UVM1.1 应用指南及源代码分析

UVM1.1 Application Guide and Source Code Analysis

张强 著

在这里，读懂 UVM

序

写这本书的难度超出了我的预料。从 8 月初开始写，一直到现在，4 个多月的时间，从刚开始的满含激情，到现在的精疲力尽。现在写出来的东西，距离我心目中的作品差距十万八千里，有太多的地方没有讲述清楚，有太多的地方需要仔细斟酌，有太多的语句需要换一种表述方式。

做为一个完美主义者，我应该努力的把所有的不完美的地方消除。可是，世间存在完美的东西么？

作为一个工程师，必须要在理想和现实之间做出一些妥协。理想总是很美好，但是现实中的资源总是有限，一个优秀的工程师不是不惜代价的实现完美，而是在现有的资源情况下做到最好。

8 月份开始写的时候，自己时间颇多，但是现在则是时间已经处于不够用的状态，已经无力再支撑自己去仔细的修改写出来的东西。

正因为如此，我停止对这本书的修改，把其对外公布。做出这个决定，我要对小学、初中、高中和大学的语文老师说一声：对不起，我给你们丢脸了；我要对那些从小到大培养我思维严谨性的数学老师、物理老师、化学老师、电路老师、编程老师等说一句：对不起，虽然我已经尽量追求严谨，但是书中依然有太多不严谨的地方；我要对广大的读者说：对不起，要阅读此书请自动开启你们的“超强纠错”功能，否则可能无法正常阅读。

写这本书，只是想把自己会的一点东西完全的落于纸上。在努力学习 UVM 的过程中，自己花费了很多时间和精力。我只想把学习的心得记录下来，希望能够给后来的人以启发。如果这本书能够给一个人带来一点点的帮助，那么我的努力就不算是白费。

这本书的前半部分（第 1 到第 9 章）介绍了 UVM 的使用，其用户群较为广泛；

而后半部分（第 10 到第 19 章）则介绍 UVM 背后的工作原理，用户群相对稀少。通常来说，一般的用户只要看懂前半部分就可以了。但是我想，世上总有像我一样有好奇心的人，不满足知其然再不知其所以然，会有人像我一样，会因为一个技术问题而彻夜难眠，如果你是这样的人，那么恭喜，这本书的后半部分就是为你准备的。

读者在阅读过程中如果有任何意见或者建议，或者发现了任何错误，请发邮件到 zhangqiang1227@gmail.com，我非常期待有人能和我探讨关于 UVM 的问题。

笔者，2011.12.11 于浙江图书馆

简要目录

详细目录

图目录

1.初识 UVM 验证平台

本章第一节将大体介绍一下感性意义上的 UVM，这里不会讲的很详细，因为关于 UVM 的历史在网上已经有太多的资料。

第二节讲述验证平台的组成，这里也只是介绍一个轮廓。

第三节开始则会教我们一步一步搭建一个UVM验证平台。或许有人会说这种方式会不会是太激进了，毕竟前一节还在教大家什么是UVM，后一节就开始尝试着一个能够运行起来的UVM验证平台。我自己最开始学习UVM是从ovm cookbook[1]开始的，那个时候，当我把那本书看完的时候，都没有搞明白一个正常的testbench应该怎么写法。其各章节之间各自为政，同时到最后也没有一章完整的把所讲述的东西给整合起来。这种感觉让我在最开始学习的时候感觉相当的难受。基于这一点，本书采取与ovm cookbook完全相反的策略，最开始的时候就给出一验证平台，之后的章节详细讲解验证平台的各个部分。读者可以与ovm cookbook对照着来看。

在传统的程序语言的教材中，第一章要讲述的肯定是 hello world 程序。因为 hello world 可以让开发让人员第一次真真切切的感受到自己的存在，当看到电脑按照自己的要求输出了东西的时候，那种喜悦感与成就感是无与伦比的。

采用这种貌似“激进”的方式，在本章结束的时候，读者就可以自己搭建起一个简单的验证平台了，这个平台虽小，不过五脏俱全，基于这个平台，后面章节讲述到的所有的内容都可以实验。

当然了，在本章描述的时候，会有很多地方让人非常不理解。先不要着急，把

[1] 《Open Verification Methodology Cookbook》，Glasser, Mark 著，Springer 出版社，2009 年第一版。不好意思的是，这本书在中国没有出版，大家只能用某度或者某歌搜索一下电子版。

不会的地方默默记着，后面会一点一点的展开。

本节最后一小节将会讲述 UVM 的执行流程。这一点对于那些好奇心极其浓的人来说，是极其必要的。但是同样的，如果你的好奇心不是那么浓，其实这一节完全可以不用看。我知道的一个 OVM 的用户，在使用 OVM 一两年之后，参与了多个项目的验证工作，但是当问到他验证整台的执行流程时，他不甚明了。不过这并不妨碍他成为一个合格的验证人员。

1.1. 什么是 UVM

UVM 是 Universal Verification Methodology 的缩写，即通用验证方法学。它起源于 OVM（Open Verification Methdology），是由 Cadence，Mentor 和 Synopsys 联合推出的新一代的验证方法学。在 2010 年 5 月，UVM 曾经出过一个 EA（early adoption）版本，这个版本里，只是单纯的把 OVM 中所有的类的前缀由 OVM 改为了 UVM，其它几乎没有任何变动。从这里可以看出 UVM 与 OVM 的深厚渊源。UVM 的第一个正式版本 1.0 是在 2011 年 2 月发布的，而截止到本书写作完成时（2011 年 12 月）的最新版 1.1 版则是在 2011 年 6 月发布的。

1.1.1. UVM 主要用在什么地方

UVM 主要用于验证数字逻辑电路的正确性。何谓验证？在数字电路的设计流程中，最开始的时候会定义需求，把需求细化成为特性列表(feature list)，之后设计人员利用 verilog 或者 systemverilog 把特性列表翻译成为 RTL 代码。在翻译的过程中，由于各种各样的原因，如设计人员自身对于 feature list 的理解不清，设计人员不小心遗漏了某种情况，这样翻译后的 RTL 代码就不能完全的反映 feature list。验证的含义就是把 feature list 和 RTL 代码比对，看看 RTL 是否能实现 feature list 的功能。被测试的 RTL 代码通常称为 DUT（Design Under Test）或者 DUV（Design Under Verification），本书统一使用 DUT 的称谓。

如果发现错误，或有建议，请联系 zhangqiang1227@gmail.com

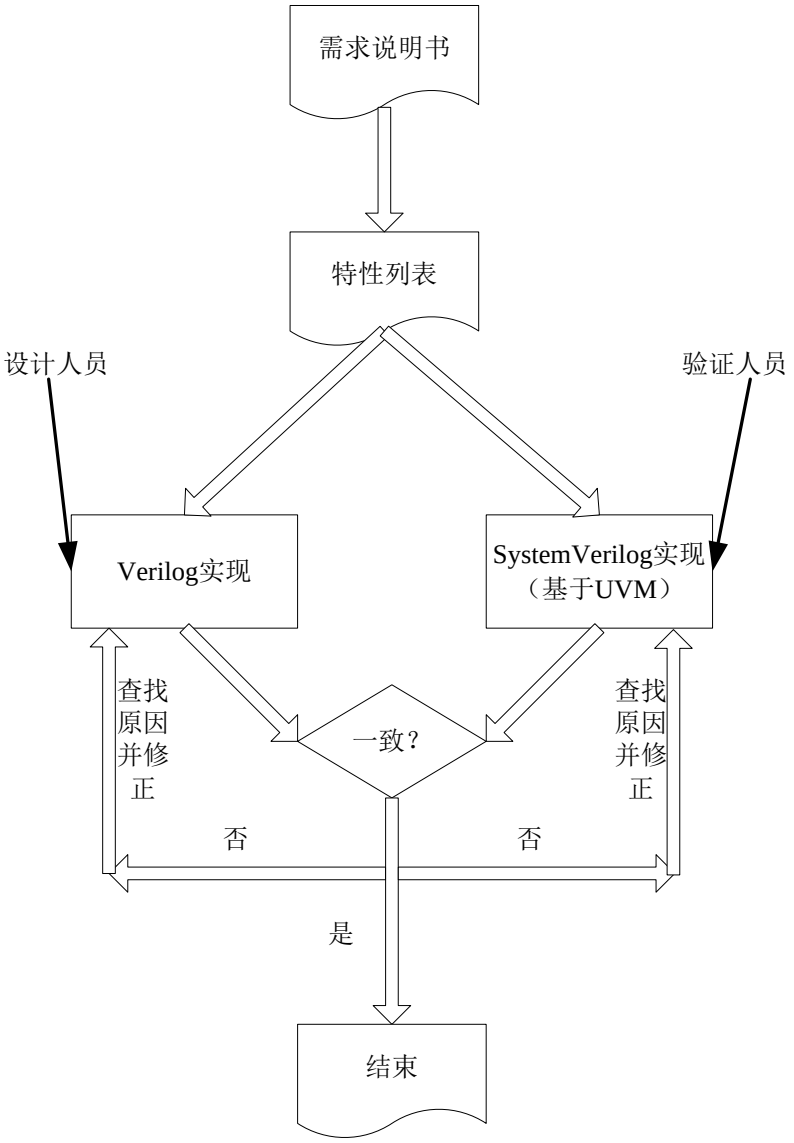


图 1-1 UVM 在数字电路设计中的位置

在上面的这幅图中，设计人员使用的是 verilog，其实这里也完全可以使用 systemverilog 实现。就如我们可以使用 c++来写 c 代码，并可以完全的用 c++的编译器来编译 c 的代码一样，systemverilog 也完全包含了 verilog，换言之，verilog 只是 systemverilog 的一个子集。

1.1.2. 何谓方法学？

当我第一次听到方法学这个词的时候完全的被震撼住了，因为我不自觉的把它跟人生观，世界观，价值观这种哲学观念放在了同一级别上，把其理解成人类探索世界认识世界时所采用的思维方式，所要遵循的原则等。但是后来当我切实了解了UVM的时候，发现这种想法实在是可笑之极[1]。

UVM 是建立在 systemverilog 平台上的一个库，它提供了一系列的接口，让我们能够更方便的进行验证。什么是库？我们大家都学习过 C 语言，其中的 stdlib.h，stdio.h，math.h 等就是 C 语言的库。库之最其本的目的就是方便人们的使用。像一个 sin 函数，如果没有 math.h，让自己来写，那要怎么写呢？使用泰勒展开吗？那要展开多少级呢？库则把这些基本的、同时又经常使用的细节给隐藏了，让我们更加方便的编写验证平台。

如下面的第一例子所示，我们期望在信息打印时同步输出时间，在 systemverilog 中只能在 display 语句中调用 time 函数。但是在 uvm 中，只要使用 uvm_warning 或者 uvm_error 或者 uvm_info 宏等，UVM 会自动帮你添加时间。在第二个例子中，经常会在出错的时候要打印一些字符，当后来问题解决的时候，我们不希望这些语句出现，于是就只能把 display 语句手工删除。但是在 UVM 中，通过设置 uvm_info 宏的 verbosity 特性，可以设置在正常运行的时候不打印这些信息，而只在调试的时候才打印，从而根本不必删除这些语句。

SystemVerilog	UVM
$display(“DRIVER: @ %0t, Cannot drive the bus”,$time);	`uvm_warning(“DRIVER”, “cannot drive the bus)
$display(“AAAAA”);	`uvm_info(“DRIVER”, “AAAAA”, UVM_DEBUG)

图 1-2 UVM 对 systemverilog 的封装

更加通俗一点说，汉语是一门美丽的语言，我们的前人留下了无数优美的篇章。如果我们说话时，引用这些篇章，那么我们说的话会很有感染力，同时也可以更好的表达自己的意思。同样的，systemverilog 也是一门优秀的语言，但是假如没有库的支撑，那么其易用性就要大打折扣。

[1] 或许多少年之后再看这句话，也会觉得这句话可笑之极。

1.2. UVM 验证平台的组成

1.2.1. 何谓验证平台

何谓验证平台？验证最基本的目的在于测试 DUT 的正确性，其最常使用的方法就是给 DUT 施加不同的输入(激励)，所以一个验证平台最重要的的功能在于产生各种各样不同的激励，并且观测 DUT 的输出结果，把此结果与期望值比较一下，判断 DUT 的正确性。注意，这里出现了一个词：期望值。什么是期望值？比如我们的 DUT 是一个加法器，那么当我们输入 1+1 时，我们期望 DUT 输出是 2。当在 DUT 计算 1+1 的结果时，验证平台也必须相应的执行同样的过程，即计算一次 1+1。在 UVM 中，完成这个过程的是参考模型（reference model）。

到此为止，可以想像一下，一个基本的验证平台会有哪几部分组成？要有一个 driver，用来把不同的激励施加给 DUT；要有一个 monitor，用来监测 DUT 的输出；要有一个 scoreboard，它专门的比较期望值与 monitor 监测到的 DUT 的输出；要有一个 reference model，它的输入跟 DUT 完全一样，它的输出送给 scoreboard，用于和 DUT 的输出比较。下图中所有的实线框和实线箭头合起来组成了一个简单的验证平台。

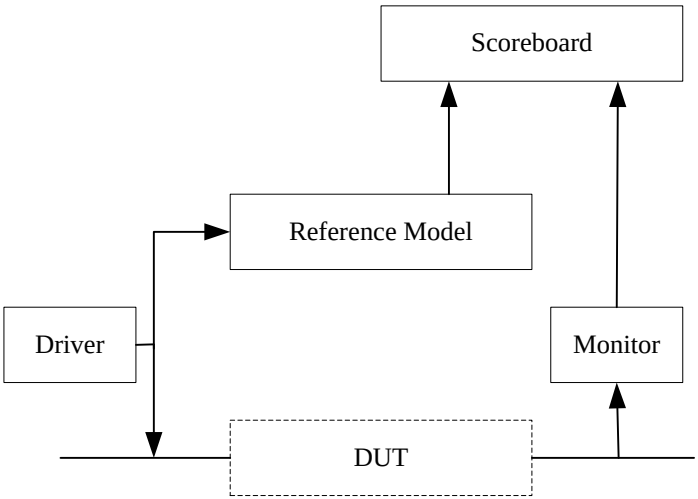


图 1-3 简单验证平台

1.2.2. UVM 验证平台

在 UVM 的验证哲学中，driver，monitor，model，scoreboard 等组成部分都是由一个类(class)来实现的。为什么要使用一个类来实现？可以这么想，如果不用类来实现，那么用什么实现？类有函数(function)，另外还可以有任务(task)，通过这些 function 和 task 可以完成 driver 的输出激励功能，完成 monitor 的监测功能，完成 model 的计算功能，完成 scoreboard 的比较功能，类中可以有成员变量，这些成员变量可以控制类的行为，如控制 driver 的行为等。所以，类是实现这些验证平台组成部分的最好选择。事实上，类也是像 systemverilog 这种面向对象编程语言中最伟大的一个发明，是面对对象的精髓所在。使用 systemverilog 而不使用其中的类就如同在中国只吃西餐不吃中餐，如同使用 windows 系统时不使用图形界面而只使用其提供的 dos 界面。所以，有不用类来实现的理由吗？

UVM 预先定义好了一个类 uvm_component，driver、monitor、model、scoreboard 等都要从这个类来派生而来。通过这种形式，把 driver、monitor、model、scoreboard 等都组织在一棵树上，这样 UVM 就可以方便的执行后面的操作。如果 uvm_component 这个概念让你相当困惑的话，那么可以暂时先忘记它，只需要记住：UVM 使用树的形式来组织管理 driver，monitor，model，scoreboard 等，这些都是树上的一个结点。整个 UVM 验证平台的各个部分就如同一棵倒置的树，如下图所示：

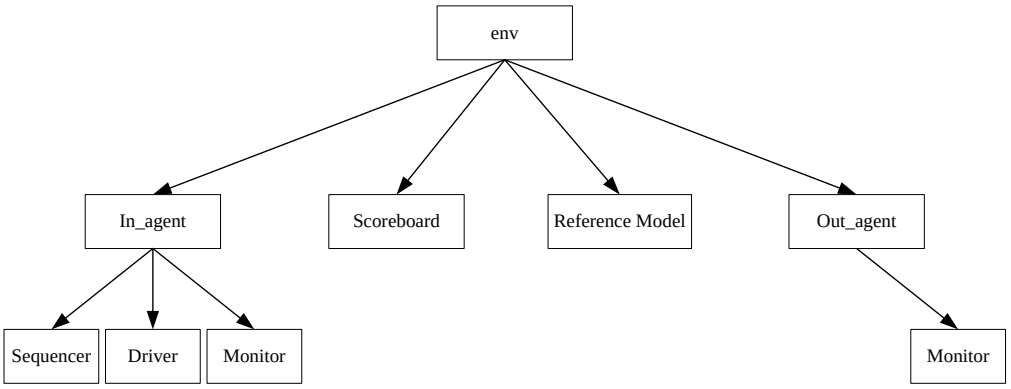


图 1-4 UVM 验证平台的树形结构

在上图中，出现了 sequencer，这是我们第一次碰到的一个概念，它是 UVM 中独有的一个概念，driver 负责向 DUT 发送数据，那么这些数据怎么来的呢？sequencer 正是用于产生这些数据的，一个 sequencer 通过启动一个 sequence，从 sequence 获取数据，并把这些数据转交给 driver。这种功能划分让 driver 不再关注数据的产生，而只负责数据的发送，职能更加清晰，更容易使用。上图中出现的另外的组件就是 In_agent 和 Out_agent，它们是 UVM 中的 agent，所谓 的 agent 其实只是简单的把 driver，

monitor 和 sequencer 封装在一起。通常来说，agent 对应的是物理接口协议，不同的接口协议对应不同的 agent，接口协议规定了数据的交换格式和方式，agent 通过 driver 和 monitor 来实现接口协议的这些内容。在一个验证平台通常会有多个 agent，如上面所示的 In_agent，里面有 driver 和 monitor，用于向 DUT 发送数据，而 Out_agent 中只有 monitor，用于监测 DUT 的输出。上图中的 env 则相当于是一个特大的容器，它把所有的 uvm_component 都包含在其内部作为其成员变量。

与图 1-4 对应的验证平台如下图所示。在这幅图中，黑色的连线表示验证平台和 DUT 的物理接口，而蓝色线表示验证平台不同的 component 之间的数据连接。

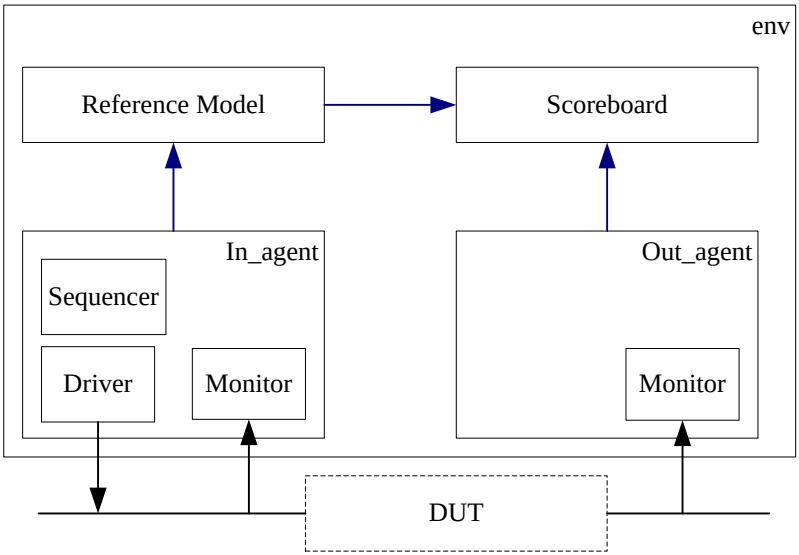


图 1-5 实际验证平台

1.3. 一个简单的 UVM 验证平台

本节介绍一个简单的 UVM 验证平台，在看本节下面各小节内容时，可能各小节之间需要前后互看，即如果当看不懂本小节时，请暂且先放一下，继续看下面的小节。当看后面的小节时，也需要回过头来回顾前面的内容。另外，本节也可能需要结合后面的第二到第七章来看。

在本节开始介绍验证平台前，将先介绍一下类的定义与实例化。关于这一点本

应在大学的编程课程里面讲述明白，只是很不幸的是，我发现很多人在大学毕业后依然不明白这两者的区别和联系。

1.3.1. 类的定义与实例化

类是面向对象编程语言中的一个基本的概念，systemverilog 就是一种面向对象的编程语言。关于类，可以扯上几十章都不为过。本文着重于介绍 UVM，因此不在这个上面展开。有兴趣的读者可以参考众多关于面向对象的书。为了更好的理解本文，请注意区分类的实例化和类的定义这两个不同的概念。所谓类的定义，就是我们用编辑器写下：

```
classs  A;
…
endclass
```

而所谓类的实例化指的是通过 new 创造出 A 的一个实例。如：

```
A  a_inst;
a_inst  =  new();
```

类的定义类似于在纸上写上了一纸条文，然后把这纸条文通知systemverilog的仿真器：我们的验证平台可能会用到这样的一个类，你要做好准备工作。而类的实例化在于通过new()，来通 知systemverilog的仿真器：你创建一个A的实例。仿真器接到new的指令后，就会在内存中划分一块空间，在划分时，会首先检查是否已经预先定义过这个类了，在已经定义过的情况下，按照定义中所指定的“条文”，分配空间，并且把这块空间的指针返回给a_inst，之后我们就可以通过a_inst来查看类中的各个成员变量，调用成员函数等。一个类，只定义了，而不实例化，是没有任何意义的 [1]。

1.3.2. UVM 验证平台中的 driver（一）

假设有如下的 DUT 定义：

```
module  dut  (clk,
              rxd,
              rx_dv,
```

[1]、其实这句话有点小小的问题，对于一些静态类，即其成员变量都是静态的，不实例化也可以正常使用。

```
                txd,
                tx_en);
input  clk;
input  [7:0]   rxd;
input          rx_dv;
output [7:0]  txd;
output         tx_en;
reg  [7:0]  txd;
reg  tx_en;

always  @(posedge  clk)  begin
          txd  <=  rxd;
          tx_en  <=  rx_dv;
end

endmodule
```

整个 dut 的功能非常简单，就是 rxd 接收数据，再直接通过 txd 发送出去。其中 rx_dv 是接收的数据有效指示，tx_en 是发送的数据有效指示。

验证要做的事情就是产生数据，然后发送到 rxd 上；在 txd 上监测输出并接收数据。检查这个数据是不是与我们之前发送过去的数据一样。如果一样，说明数据经过我们的 DUT 没有丢失，DUT 工作正常；如果不一样，说明 DUT 把数据给丢失了，那就需要仔细核查，找出问题所在。UVM 验证平台中使用 driver 来发送数据。一个实际的可以使用的 driver 如下：

```
 1`include  "uvm_macros.svh"
 2 import  uvm_pkg::*;
 3
 4 class  my_driver  extends  uvm_driver  #(my_transaction);
 5      virtual  my_if  vif;
 6      uvm_analysis_port  #(my_transaction)    ap;
 7
 8      `uvm_component_utils(my_driver)
 9
10      extern  function  new  (string  name,  uvm_component  parent);
11      extern  virtual  function  void  build_phase(uvm_phase  phase);
12      extern  virtual  task  main_phase(uvm_phase  phase);
13      extern  task  drive_one_pkt(my_transaction  req);
14      extern  task  drive_one_byte(bit  [7:0]  data);
15 endclass
16
17 function  my_driver::new  (string  name,  uvm_component  parent);
18      super.new(name,  parent);
19 endfunction
20
21 function  void  my_driver::build_phase(uvm_phase  phase);
22      super.build_phase(phase);
23      if(!uvm_config_db#(virtual  my_if)::get(this,  "",  "my_if",  vif))
24          `uvm_fatal("my_driver",  "Error  in  Getting  interface");
25      ap  =  new("ap",  this);
```

```
26 endfunction
```

uvm 中已经定义好了类 uvm_driver，我们自己的 driver 一般从这个类派生而来。第 4 行有一个 my_transaction，将会在下节中介绍。

第 5 行中定义了一个 virtual interface。为什么要有 virtual interface 这种东西？通常，我们施加激励是在一个类的 task 中施加的，而我们的 dut 是纯粹的 verilog 代码。怎么样把 verilog 代码和类连接在一起，并且让它们互相通信（所谓的通信就是我们能够在类的函数或者任务中把数据施加给 dut，然后把在类的函数或者任务中监测 dut 的输出）？这个问题是 systemverilog 的问题。

在 systemverilog 中，使用 virtual interface 把类和 verilog 实例的输入输出连接在一起。要使用 virtual interface，必须先定义好一个实际的 interface：

```
 1 interface my_if(input logic rxc, input logic txc);
 2
 3     logic [7:0]     rxd;
 4     logic             rx_dv;
 5
 6     logic [7:0]     txd;
 7     logic             tx_en;
 8
 9     // from model to dut
10     clocking drv_cb @(posedge rxc);
11         output      #1   rxd, rx_dv;
12     endclocking
13
14     clocking mon_cb @(posedge txc);
15         input       #1   txd, tx_en;
16     endclocking // tx_cb
17
18 endinterface // my_if
```

关于这个 interface 中的一些定义如果不甚理解，可以查看 systemverilog 的相关语法。当一个 interface 定义好了之后就可以在 driver 中使用这个 interface，我们通常在最顶层的 module 中（即 vcs 或者 questa 启动后寻找的第一个 module 入口，类似于 C 语言的 main 函数，默认名字为 top）把 interface“实例化”，之后把这个 interface 的一端连接 DUT，另外一端连接验证平台中的相关组件，如 driver，monitor 等。这里的连接方式在 UVM 中是通过一种称为 config_db 机制的方式来实现的。在 my_driver::build_phase 的 21 行中，通过调用 uvm_config_db::get 函数来得到从最顶层的 module 传递来的 interface。这里的语法比较古怪，不过也仅仅只是一个语法，只需要把其劳记。如果实在想搞明白这到底是怎么一回事，可以先看本书第 6 章。

另外，需要注意的是在 vif 的声明中，要加入一个关键词 virtual。在一个类的定义中，是不应该出现 my_if 这种 interface，出现了就会报语法错误。但是 systemverilog 允许这种 virtual my_if 形式的 interface，这也是类和 dut 之间通信的唯一的方法。

☆：在 UVM 验证平台中使用 virtual interface 来实现 DUT 和验证平台的通信。

my_driver 的第 6 行定义了一个成员变量 ap。它主要是用于把 driver 发送出去的东西通知给 reference model，从而 reference model 和 DUT 接收到同样的激励。ap 的类型是 uvm_analysis_port#(my_transaction)，这是一个参数化的类，是 UVM 中的一种用于传递 transaction 级别信息的通信端口，它是 TLM（Transaction Level Modeling）通信在 UVM 中的具体实现。

☆：UVM 验证平台中，各个 component 之间通信使用 TLM 方式，其中要发送信息的一方（如 monitor）一般使用 uvm_analysis_port。

第 8 行出现了 uvm_component_utils 宏。uvm_driver 是一个派生自 component 的类，对于这样一个类，在定义时一般要调用 uvm_component_utils 宏来把定义好的类通知给 uvm，其实质是 factory 机制。对于初学者来说，只要写上这么一个宏好了。

在定义一个派生自 component 的类时，要定义好三个函数(任务)：new，build_phase 和 main_phase。

new 函数在定义的时候一般都要写上 super.new(name, parent)这一句，意思就是说执行父类的 new 函数，这里的父类自然是指 uvm_driver 了。另外要指定两个参数，一个参数是 string 类型的 name，就是这个类的实例的名字；一个是 uvm_component 型的 parent，以表明这个 uvm_driver 在 uvm 整个的运行树中处于什么位置。在图 1-4 中，driver 的 parent 就是 In_agent，而 In_agent 的 parent 就是 env。

☆：派生自 uvm_component 的类在其 new 函数中要指明名字（name）和父 母（parent），其中 parent 是一个 uvm_component 形的变量，代表此实例在 UVM 树中的父结点。

在类的 build_phase 中，也要写上 super.build_phase(phase)这一句，一般 来说，在 build_phase 中要完成类的实例化工作，在图 1-4 的 UVM 树中，driver 是最底层的，没有孩子，所以这里不必进行实例化工作，只需要调用 super.build_phase(phase)一句。但是在图 1-4 的 In_agent 实现 driver，monitor 和 sequencer 的实例化，而 In_agent 的实例化则在 env 的 build_phase 中完成，那么 env 的实例化在什么地方完成呢？这个问题不着急，暂时先存着这个疑问。build_phase 中除了要完成实例化之外，还要完成 config_db 机制的 get 行为，即把其它 component 设置给此 component 的一些参数接收过来。如在 my_driver 中就把 my_if 给接收了过来（这里把 my_if 也当成是一个参数）。

除了 build_phase 之后，另外一个重要的任务就是 main_phase，将会在 1.3.4 节中介绍它。

1.3.3. UVM 验证平台中的 transaction

上节提到了 my_transaction。一般的，物理协议中的数据交换都是以帧或者包为单位的，通常一帧或者一个包中要定义好各项参数，每个包的大小不一样。以以太网为例，每个包的大小至少是 64 个 Byte。这个包中要包括源地址，目的地址，包的类型，整个包的 crc 校验数据。很少会有协议是以 bit 或者 byte 为单位来进行数据交换的。transaction 就是为了模拟这种实际情况。一个 transaction 就是一个包。my_transaction 的定义为：

```
 1 `include "uvm_macros.svh"
 2 import uvm_pkg::*;
 3
 4 class my_transaction extends uvm_sequence_item ;
 5     rand bit [47:0]          dmac;
 6     rand bit [47:0]          smac;
 7     rand bit [15:0]          ether_type;
 8     rand byte                pload[] ;    // size should be configurable
 9     rand bit [31:0]          crc;
10
11     constraint   cons_pload_size {
12         pload.size >= 46;
13         pload.size <= 1500;
14     }
15     extern function new (string name = "my_transaction");
16     `uvm_object_utils_begin(my_transaction)
17         `uvm_field_int(dmac, UVM_ALL_ON)
18         `uvm_field_int(smac, UVM_ALL_ON)
19         `uvm_field_int(ether_type, UVM_ALL_ON)
20         `uvm_field_array_int(pload, UVM_ALL_ON)
21         `uvm_field_int(crc, UVM_ALL_ON)
22     `uvm_object_utils_end
23 endclass // my_transaction
24
25 function my_transaction::new(string name = "my_transaction");
26     super.new(name);
27 endfunction
28
```

在这个定义中，5 到 9 行用于声明成员变量，dmac 是目的地址，一共是 6byte，smac 是源地址，6byte，ether_type 是 2byte，pload 中存放的就是要传输的数据，而 crc 则表示数据的校验，防止数据传送中发生错误。在定义 transaction 的时候，可能要添加一些限制条件。如我们指定一个 pload 只能是大于 46byte，小于 1500byte。11 行到 14 行就是为了实现这个限制条件。

在定义 transaction 时，同样要加入 factory 的实现。与 component 不同的是，transaction 是一个 object，它的 factory 实现要使用 uvm_object_utils 系列宏。16 到 22

如果发现错误，或有建议，请联系 zhangqiang1227@gmail.com

行就是为了实现 factory 机制，至于这里为什么加这么多宏，加上宏之后都发生了什么事情，暂且先当成一个疑问。

在了解了 transaction 之后，回头看 1.3.2 节中 my_driver 的定义：

```
4 class my_driver extends uvm_driver #(my_transaction);
```

我们知道，driver 是用于向 DUT 发送数据的，DUT 只能通过 pin 角简单的接收二进制的数据，而在验证平台中，我们则用 transaction 来处理与表征数据。所以 driver 首先需要把数据从 transaction 中提出出来，然后把其转换为 DUT 的 pin 角能够接收的数据。通常来说，一个 driver 只能接收一种 transaction，所以在 my_driver 被定义成了一个参数化的类，这个参数就是 my_driver 所能接收的 transaction 的类型。

1.3.4. UVM 验证平台中的 driver（二）

任何一个派生自uvm_component类的主要的动作都是在main_phase中完成的 [1]。一个driver的行为主要在main_phase中体现，所以定义好main_phase是关键。my_driver的main_phase及其用到的方法 [2]如下：

```
28 task my_driver::main_phase(uvm_phase phase);
29     my_transaction   req;
30     super.main_phase(phase);
31     vif.drv_cb.rxd     <= 0;
32     vif.drv_cb.rx_dv <= 1'b0;
33     while(1) begin
34         seq_item_port.get_next_item(req);
35         drive_one_pkt(req);
36         ap.write(req);
37         seq_item_port.item_done();
38     end
39 endtask
40
41 task my_driver::drive_one_pkt(my_transaction req);
42     byte unsigned        data_q[];
43     int   data_size;
44     data_size = req.pack_bytes(data_q) / 8;
45     repeat(3) @vif.drv_cb;
46     for ( int i = 0; i < data_size; i++ ) begin
47         drive_one_byte(data_q[i]); // drive data pattern
```

[1] 、这句话有点小小的问题，那就是所有的动作也可以在 run_phase 中完成。但是 main_phase 是 UVM 现在大力提倡的方式，所以这里也推荐读者使用 main_phase。

[2] 、方法是面向对象编程中的一个重要概念，通常称一个类的函数为这个类的方法。在本书中，将会把类的函数（function）和任务（task）统称为这个类的方法。

```
48      end
49      @vif.drv_cb;
50      vif.drv_cb.rx_dv <= 1'b0;
51  endtask
52
53  task my_driver::drive_one_byte(bit [7:0] data);
54      @vif.drv_cb;
55      vif.drv_cb.rxd   <= data;
56      vif.drv_cb.rx_dv <= 1'b1;
57  endtask
```

main_phase 的第一句话是 super.main_phase。这个是调用父类的 main_phase，跟 super.build_phase 类似。31 行和 32 行执行初始化的工作。33 行开始一个无限循环，在这个循环中，34 行向 seq_item_port 申请得到一个 my_transaction 类型的 item。35 行调用 drive_one_pkt 把这个 item 发送出去，36 行把发送出去的 item 放入 ap，送给 reference 一份，37 行则是和 34 行形成一对，这两句话总是会成对出现。这里的关键就是 34 行。在这一行中，出现了 seq_item_port，它是用于连接 driver 和 sequencer 的一个端口，driver 如果想要发送数据就要从这个端口中获得；sequencer 如果有数据要交给 driver，也要通过这个端口送给 driver。从这个端口申请数据要调用这个端口的 get_next_item 方法；当数据驱动完毕时，要通过调用 item_done 来告知这个端口。另外，细心的读者可能发现了，seq_item_port 并没有在 my_driver 的成员变量中，那怎么能使用呢？事实中，它在 my_driver 的父类，即 uvm_driver 类中。它的实例化也是在 uvm_driver 的 build_phase 中完成的。

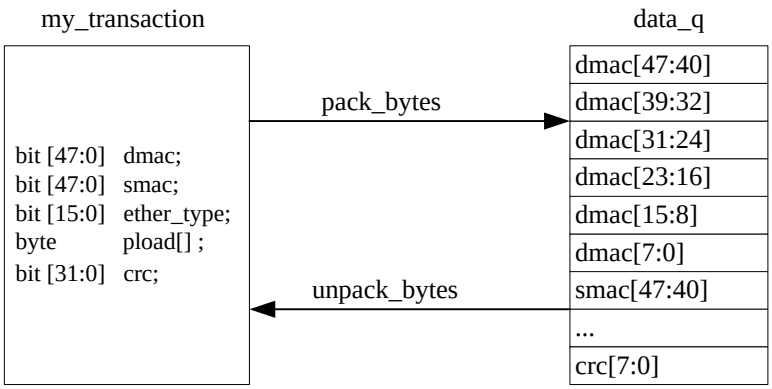


图 1-6 pack_bytes 和 unpack_bytes

driver_one_pkt 中首先使用 pack_bytes 函数把 my_transaction 中的数据变为数据流存放在 data_q 中。这里之所以能使用 pack_bytes 函数，是因为我们之前在定义 my_transaction 时使用了 uvm_field_*一系列宏。按照 uvm_field_*一系列宏书写的顺序，pack_bytes 函数把所有的字段打包成以 byte 为单位的数据流。之后，可以调用 drive_one_byte 函数把 data_q 中的数据一个个的灌注给 dut。pack_bytes 函数的原理

如图 1-6 所示。

pack_bytes 把 48 位的 dmac 分成 6 份，按照设置的大小端格式，把这 6 份放入 data_q 中。放完了 dmac 后，再放 smac，真正到把 crc 放完后为止。这里放的顺序就是按照 uvm_field_*宏的书写顺序。在上面的例子中，crc 的 uvm_field 宏是最后写的，所以这里就会被放在最后面。

☆：一个派生自 uvm_component 的类，其主要的行为发生在 main_phase 中，诸如 driver，monitor，reference model，scoreboard 等最重要的是定义好其 main_phase。

1.3.5. UVM 验证平台中的 monitor

monitor 主要用于监测 DUT 的输出，一个简单的 monitor 如下：

```
 1 `include "uvm_macros.svh"
 2 import uvm_pkg::*;
 3
 4 class my_monitor extends uvm_monitor;
 5     virtual my_if vif;
 6     uvm_analysis_port #(my_transaction)  ap;
 7     extern function new (string name, uvm_component parent);
 8     extern virtual function void build_phase(uvm_phase phase);
 9     extern virtual task main_phase(uvm_phase phase);
10     extern task receive_one_pkt(ref my_transaction get_pkt);
11     extern task get_one_byte(ref logic valid, ref logic [7:0] data);
12     `uvm_component_utils(my_monitor)
13 endclass
14
15 function my_monitor::new (string name, uvm_component parent);
16     super.new(name, parent);
17 endfunction
18
19 function void my_monitor::build_phase(uvm_phase phase);
20     super.build_phase(phase);
21     if(!uvm_config_db#(virtual my_if)::get(this, "", "my_if", vif))
22         uvm_report_fatal("my_monitor","Error in Getting interface");
23     ap = new("ap", this);
24 endfunction
25
26 task my_monitor::main_phase(uvm_phase phase);
27     logic valid;
28     logic [7:0] data;
29     my_transaction tr;
30     super.main_phase(phase);
31     while(1) begin
32         tr = new();
```

```
33          receive_one_pkt(tr);
34          ap.write(tr);
35      end
36  endtask
37
38  task my_monitor::get_one_byte(ref logic valid, ref logic [7:0] data);
39      @vif.mon_cb;
40      data  = vif.mon_cb.txd;
41      valid = vif.mon_cb.tx_en;
42  endtask
43
44  task my_monitor::receive_one_pkt(ref my_transaction get_pkt);
45      byte unsigned data_q[$];
46      byte unsigned data_array[];
47      logic [7:0] data;
48      logic valid = 0;
49      int data_size;
50      while(valid !== 1) begin
51          get_one_byte(valid, data);
52      end
53      while(valid) begin
54          data_q.push_back (data);
55          get_one_byte(valid, data);
56      end
57      data_size  = data_q.size();
58      data_array = new[data_size];
59      for ( int i = 0; i < data_size; i++ ) begin
60          data_array[i] = data_q[i];
61      end
62      get_pkt.pload = new[data_size - 18]; //da sa, e_type, crc
63      data_size = get_pkt.unpack_bytes(data_array) / 8;
64  endtask
```

my_monitor 与 driver 有些类似，它通过无限的循环来不断的监测 DUT 的输出。通过 get_one_byte 函数不断的收集 DUT 输出的数据。因为协议中只有在 tx_en 为高电平时，txd 是有效的，所以 50 行 52 行用于检测第一个合法的 txd，在检测到后，说明 DUT 开始正式发送有效数据，53 到 56 行把所有的有效数据放入 data_q 中，62 行中通过 data_size 给 get_pkt 的 pload 分配空间，63 行通过 unpack_bytes 函数，把 data_q 中的数据放入 get_pkt 中。unapck_bytes 的过程见图 1-6。

1.3.6. UVM 验证平台中的 agent

agent 在 UVM 验证平台中的主要作用就是把 monitor 和 driver 封装在一起，因为 monitor 和 driver 都是直接和 DUT 的接口打交道的，当把它们封装成为一个 agent 后，整个验证平台中就只有 agent 是与实际的物理接口打交道的。

在 UVM 验证平台的 agent 中，除了封装了 monitor 和 driver 外，还封装了 sequencer。sequencer 在本质上是不与物理接口直接打交道的，但是 sequencer 与 driver 关系非常密切，sequencer 的职能最初就是从 driver 中分离出来的，所以 UVM 中也把 sequencer 封装在一起。一个简单的 sequencer 的定义为：

```
`include "uvm_macros.svh"
import uvm_pkg::*;

class my_sequencer extends uvm_sequencer #(my_transaction);
    //Component
    extern function new( string name, uvm_component parent);
    extern function void build_phase(uvm_phase phase);

    //Register
    `uvm_component_utils(my_sequencer)
endclass

function my_sequencer::new(string name, uvm_component parent);
    super.new(name, parent);
endfunction // new

function void my_sequencer::build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction
```

这个定义是我们到现在为止看到的几乎最简单的定义。事实上，sequencer 确实几乎是 UVM 验证平台中最简单的一个 component，大部分验证平台中用到的 sequencer 都跟这个定义差不多，但是像其它的 component，如 driver，monitor 在不同的验证平台中则会千差万别。my_sequencer 是一个参数化的类，其参数是 my_transaction，用于表明这个 sequencer 只能产生 my_transaction 类型的数据，这一点与 my_driver 类似。

sequencer 定义完成后，可以看 agent 的定义了：

```
class my_agent extends uvm_agent ;
    my_sequencer    sqr;
    my_driver       drv;
    my_monitor      mon;

    extern function new(string name, uvm_component parent);
    extern virtual function void build_phase(uvm_phase phase);
    extern virtual function void connect_phase(uvm_phase phase);

    uvm_analysis_port#(my_transaction) ap;

    `uvm_component_utils_begin(my_agent)
        `uvm_field_object ( sqr          , UVM_ALL_ON)
        `uvm_field_object ( drv          , UVM_ALL_ON)
        `uvm_field_object ( mon          , UVM_ALL_ON)
```

```
16      `uvm_component_utils_end
17  endclass // my_agent
18
19  function my_agent::new(string name, uvm_component parent);
20      super.new(name, parent);
21  endfunction // new
22
23  function void my_agent::build_phase(uvm_phase phase);
24      super.build_phase(phase);
25      if (is_active == UVM_ACTIVE) begin
26          sqr = my_sequencer::type_id::create("sqr", this);
27          drv = my_driver::type_id::create("drv", this);
28      end
29      else begin
30          mon = my_monitor::type_id::create("mon", this);
31      end
32  endfunction // build_phase
33
34  function void my_agent::connect_phase(uvm_phase phase);
35      super.connect_phase(phase);
36      if (is_active == UVM_ACTIVE) begin
37          drv.seq_item_port.connect(sqr.seq_item_export);
38          this.ap = drv.ap;
39      end
40      else begin
41          this.ap = mon.ap;
42      end
43  endfunction
```

my_agent 的关键在于 build_phase。在这个函数中，通过 is_active 的值来实例化不同的成员变量。is_active 用来形容这个 agent 所扮演的角色。通常的，当一个 agent 要驱动总线时，它会实例化一个 driver，这种情况下就是 UVM_ACTIVE；当一个 agent 只是用于监测总线的输出时，它不需要把 monitor 实例化，而只需要把 monitor 实例化，如下图所示：

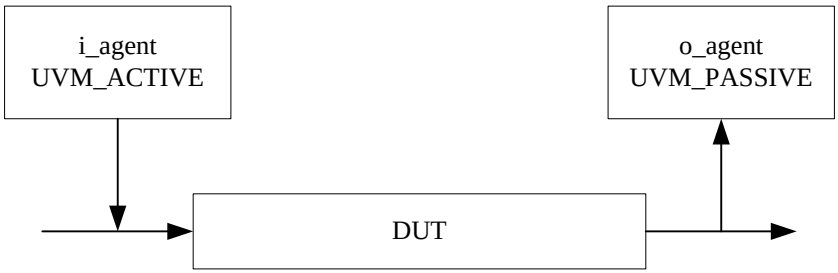


图 1-7 UVM 验证平台中的 agent

在 i_agent 中，它只需要负责向 DUT 发送数据，同时把发送的数据复制一份发送给 reference 即可，所以它是 UVM_ACTIVE 的。而在 o_agent 中，只是负责监测

DUT 的输出，所以它不需要实例化 driver，是 UVM_PASSIVE 的。

☆：agent 有两种形式，UVM_PASSIVE 和 UVM_ACTIVE，以前者方式运转的 agent 只监测总线而不驱动总线；以后者方式运行的 agent 驱动总线，也可以监测总线。

在 i_agent 中，把发送出的数据送给 reference model 有两种方式，一种是实例化一个 monitor，通过 monitor 监测 driver 发送的数据，收集之后通过 agent 的 ap 端口发送给 reference model；另外一种是 my_driver 在发送完数据后向 agent 的 ap 端口写入数据，从而传送给 reference model，也就是说，在这种方式中，根本不需要实例化一个 monitor。通常说来，前面一种方式更加正规些，合理些，但是同时也更加复杂些；后者简单些，但是其重用性没有第一种好。这里简单起见，使用第二种方式。当 is_active 为 UVM_ACTIVE 时，在 my_agent 的 build_phase 中实例化 driver 和 sequencer，同时在 connect_phase 中把 agent 的 ap 指向 driver 的 ap。当 is_active 为 UVM_PASSIVE 时，在 my_agent 的 build_phase 中实例化 monitor，同时在 connect_phase 中把 agent 的 ap 指向 monitor 的 ap。

☆：UVM 验证平台中的 agent 实现了 driver，monitor 和 sequencer 的封装，agent 是整个验证平台中唯一直接与 DUT 的物理接口打交道的组件。一个 agent 的行为主要是由其封装的 driver，monitor 和 sequencer 实现，所以 agent 的 main_phase 一般不写任何内容。agent 最重要的是其 build_phase 和 connect_phase。

1.3.7. UVM 验证平台中的 reference model

reference model 是整个 UVM 验证平台的关键与核心，因为 reference model 作的工作与 DUT 一致，scoreboard 将会根据 reference model 的输出和 DUT 进行比较，如果 reference model 中有错误，那么其输出也是不可信的，从而 scoreboard 的输出也是不可信的。

本例中的 reference model 如下：

```
1 `include "uvm_macros.svh"
2 import uvm_pkg::*;
3
4 class my_model extends uvm_component;
5
6     uvm_blocking_get_port #(my_transaction)   port;
7     uvm_analysis_port #(my_transaction)   ap;
8
```

```
 9      extern function new(string name, uvm_component parent);
10      extern function void build_phase(uvm_phase phase);
11      extern virtual  task main_phase(uvm_phase phase);
12
13      `uvm_component_utils(my_model)
14  endclass // my_model
15
16  function my_model::new(string name, uvm_component parent);
17      super.new(name, parent);
18  endfunction // new
19
20  function void my_model::build_phase(uvm_phase phase);
21      super.build_phase(phase);
22      port = new("port", this);
23      ap = new("ap", this);
24  endfunction // build_phase
25
26  task my_model::main_phase(uvm_phase phase);
27      my_transaction tr;
28      super.main_phase(phase);
29      while(1) begin
30          port.get(tr);
31          ap.write(tr);
32      end
33  endtask
34
```

第 6 行定义了一个uvm_blocking_get_port类型的端口，它与uvm_analysis_port一样，同样是TLM通信的一种端口。它用于接收一个uvm_analysis_port发送的信息，而uvm_analysis_port是发送信息的。UVM验证平台各个组件之间一般使用这种方式来实现transaction级别通信 [1]。

第 7 行定义了一个 uvm_analysis_port，用于把 reference model 的输出结果发送给 scoreboard。

由于 DUT 比较简单，只是把数据转发，没有做任何工作，因此这里的 main_phase 在 30 行接收到一个 transaction 后，31 行直接未做处理把这个 transaction 发送出去。

☆：reference model 是整个验证平台的核心。在 reference model 中使用 uvm_blocking_get_port 来获得 agent 发送来的数据。

[1] 在后面介绍 TLM 通信的章节将会看到，还有另外一种通信的方式，另外一种更加直接，不过不易于理解

1.3.8. UVM 验证平台中的 scoreboard

scoreboard 主要用于比较 reference model 和 DUT 输出是否一致，并给出比较结果。一个简单的 scoreboard 如下：

```
1  `include "uvm_macros.svh"
2  import uvm_pkg::*;
3
4  class my_scoreboard extends uvm_scoreboard;
5      my_transaction   expect_queue[$];
6      uvm_blocking_get_port #(my_transaction)   exp_port;
7      uvm_blocking_get_port #(my_transaction)   act_port;
8      `uvm_component_utils(my_scoreboard)
9
10     extern function new(string name, uvm_component parent = null);
11     extern virtual function void build_phase(uvm_phase phase);
12     extern virtual task main_phase(uvm_phase phase);
13 endclass
14
15 function my_scoreboard::new(string name, uvm_component parent = null);
16     super.new(name, parent);
17 endfunction
18
19 function void my_scoreboard::build_phase(uvm_phase phase);
20     super.build_phase(phase);
21     exp_port = new("exp_port", this);
22     act_port = new("act_port", this);
23 endfunction
24
25 task my_scoreboard::main_phase(uvm_phase phase);
26     my_transaction   get_expect,  get_actual, tmp_tran;
27     bit result;
28
29     super.main_phase(phase);
30     fork
31         while (1) begin
32             exp_port.get(get_expect);
33             expect_queue.push_back(get_expect);
34         end
35         while (1) begin
36             act_port.get(get_actual);
37             if(expect_queue.size() > 0) begin
38                 tmp_tran = expect_queue.pop_front();
39                 result = get_actual.compare(tmp_tran);
40                 if(result) begin
41                     $display("Compare SUCCESSFULLY");
42                 end
43                 else begin
44                     $display("Compare FAILED");
```

```
45                      $display("the expect pkt is");
46                      tmp_tran.print();
47                      $display("the actual pkt is");
48                      get_actual.print();
49                  end
50              end
51              else begin
52                  $display("ERROR::Received from DUT, while Expect Queue is empty");
53                  get_actual.print();
54              end
55          end
56      join
57  endtask
58
```

6 到 7 行分别定义了两个端口，一个端口是 exp_port，用于从 referencce model 的 ap 获得期望的数据；另外一个端口是 act_export，用于从 monitor 的 ap 获得实际 DUT 的输出数据。

main_phase 中使用 fork 开启了两个进程，其中一个进程用于从 reference model 中获得数据，另外一个进程用于从 monitor 中获得数据。一般说来，同样的一组数据，经过 DUT 的处理后会有一定的延迟，但是在 reference model 中不会（除非刻意这么做）。因此，从 scoreboard 的角度来看，reference model 中的数据总是先到达，所以 33 行把 reference model 的数据先放入一个队列中。由于 DUT 的输出后到达，36 行当接收到 DUT 的输出后，先查看队列中是否有记录，如果没记录，说明 reference model 没有数据输出，而 DUT 输出了，这是不期望的，52 行给出错误信息。如果队列中有记录，那么 38 行从队列中弹出第一个数据，并把此数据和 DUT 的输出比较。这里比较用到了 compare 函数，这里之所以可以用这个函数比较，是因为在定义 transaction 时，使用了一系列的 uvm_field_*宏。compare 将会逐字段比较 get_actual 和 tmp_tran，如果所有的字段都一样，那么返回 1，否则返回 0。

☆：scoreboard 中一般使用一个队列来暂存从 reference model 得到的期望数据。

1.3.9. UVM 验证平台中的 env

env 是整个 UVM 验证平台中的大容器，在图 1-4 中，env 是整个 UVM 树的最高层，它里面包含了 UVM 验证平台中的常用的组件。本例中的 UVM 验证平台如下所示：

```
1  `include "uvm_macros.svh"
2  import uvm_pkg::*;
```


如果发现错误，或有建议，请联系 zhangqiang1227@gmail.com

```
 3
 4 class my_env extends uvm_env;
 5     my_agent   i_agt;
 6     my_agent   o_agt;
 7     my_model   mdl;
 8     my_scoreboard scb;
 9
10     uvm_tlm_analysis_fifo #(my_transaction) agt_scb_fifo;
11     uvm_tlm_analysis_fifo #(my_transaction) agt_mdl_fifo;
12     uvm_tlm_analysis_fifo #(my_transaction) mdl_scb_fifo;
13
14     extern function new(string name, uvm_component parent);
15     extern virtual function void build_phase(uvm_phase phase);
16     extern virtual function void connect_phase(uvm_phase phase);
17     `uvm_component_utils(my_env)
18 endclass
19
20 function my_env::new(string name, uvm_component parent);
21     super.new(name, parent);
22 endfunction // new
23
24 function void my_env::build_phase(uvm_phase phase);
25     super.build_phase(phase);
26     i_agt = new("i_agt", this);
27     o_agt = new("o_agt", this);
28     i_agt.is_active = UVM_ACTIVE;
29     o_agt.is_active = UVM_PASSIVE;
30     mdl = new("mdl", this);
31     scb = new("scb", this);
32     agt_scb_fifo = new("agt_scb_fifo", this);
33     agt_mdl_fifo = new("agt_mdl_fifo", this);
34     mdl_scb_fifo = new("mdl_scb_fifo", this);
35 endfunction // build_phase
36
37 function void my_env::connect_phase(uvm_phase phase);
38     super.build_phase(phase);
39     i_agt.ap.connect(agt_mdl_fifo.analysis_export);
40     mdl.port.connect(agt_mdl_fifo.blocking_get_export);
41     mdl.ap.connect(mdl_scb_fifo.analysis_export);
42     scb.exp_port.connect(mdl_scb_fifo.blocking_get_export);
43     o_agt.ap.connect(agt_scb_fifo.analysis_export);
44     scb.act_port.connect(agt_scb_fifo.blocking_get_export);
45 endfunction
```

5 到 8 行定义成员变量，这里定义了两个 agent，i_agent 用于向 DUT 发送数据，而 o_agent 用于从 DUT 接收数据。在 build_phase 中，在二者实例化后，把 i_agent 配置为 UVM_ACTIVE 模式，而把 o_agent 配置为 UVM_PASSIVE 模式。

10 到 12 行定义了 3 个 fifo，这 3 个 fifo 主要用于连接 uvm_blocking_get_port 和 ap。假如没有 fifo，直接把 uvm_blocking_get_port 和 ap 直接相连接，当一个 ap 写入

数据时，有两种可能，一是 uvm_blocking_get_port 正在等待接收数据（如 scoreboard 的 32 和 36 行所示），这种情况下，ap.write()可以马上返回；二是 uvm_blocking_get_port 没有等待数据，在这种情况下一般就是 uvm_blocking_get_port 所在的 uvm_component 正在处理别的事情，无暇从 uvm_blocking_get_port 获得数据。在这种情况下，ap.write 的行为会是怎么样呢？为了应对这种情况，有必要在 uvm_blocking_get_port 和 ap 之间连接一个 fifo。在这个例子中，需要把 i_agent 和 reference model 相连接，把 reference model 和 scoreboard 连接，把 o_agent 和 sccoreboard 相连接，所以一共需要 3 个 fifo。在 connect_phase 中，把这 3 个 fifo 分别和对应端口连接。

☆：UVM 验证平台中可以使用 uvm_tlm_analysis_fifo 把 uvm_blocking_get_port 和 uvm_ analysis_port 连接。

1.3.10. UVM 验证平台中的 sequence

sequence 机制是 UVM 中最核心的东西。前面一直说，driver 要驱动数据时，向 sequencer 申请一个 item，申请到了就发送。而 sequencer 的定义前面也已经看到了，它是相当简单的，那么 sequencer 又是如何产生数据的呢？答案就是 sequnece。一个简单的 sequence 如下：

```
 1 `include "uvm_macros.svh"
 2 import uvm_pkg::*;
 3
 4 class my_sequence extends uvm_sequence #(my_transaction);
 5     my_transaction m_trans;
 6
 7     extern function new(string name = "my_sequence");
 8     virtual task body();
 9         if(starting_phase != null)
10             starting_phase.raise_objection(this);
11         repeat (10) begin
12             `uvm_do(m_trans)
13         end
14         #100;
15         if(starting_phase != null)
16             starting_phase.drop_objection(this);
17     endtask
18
19     `uvm_object_utils(my_sequence)
20 endclass
21
22 function  my_sequence::new(string name= "my_sequence");
23     super.new(name);
24 endfunction // new
```

第 4 行开始定义 my_sequence 类，它派生自 uvm_sequence，并且是一个参数化的类，其参数为 my_transaction，表明 此 sequence 只能产生 my_transaction 类型的 item。uvm_sequence 是 UVM 中内建的一个类，它实现了 sequence 机制的强大功能，它派生自 uvm_object，而不是 uvm_component，所以 19 行在使用 factory 机制时，用的是 uvm_object_utils。

sequence 中最关键的地方在于 body，当一个 sequence 启动起来后，会自动执行其 body 方法。9、10 行和 15、16 行暂且先跳过，body 的关键是 12 行，这是一个宏，这个宏每执行一次就会向 uvm_sequencer 发送一个数据，uvm_sequencer 收到一个数据就转给 uvm_driver。在我们的例子中，调用了 10 次 uvm_do 宏，所以将向 sequencer 发送 10 个数据。这里需要注意的是 uvm_do 宏产生一个 item 是不需要仿真时间（即 $time 返回的时间），它只需要消耗物理 cpu 的时间，把 item 送给 sequencer 也不需要仿真时间，但是由于 driver 需要一个或者多个时钟才能发送完一个数据，所以 driver 从 sequencer 得到数据并驱动完毕需要仿真时间。当在 driver 中调用 seq_item_port 的 item_done 方法后，一次 uvm_do 宏执行完毕，进入下一次 uvm_do 宏的执行中。当所有的 uvm_do 执行完毕后，driver 的 seq_item_port 的 get_next _item 方法就接收不到任何数据，会一直等待（阻塞）在那里。

那么 sequence 什么时候启动？如何启动？一个 sequence 可以通过如下的方式启动：

```
task my_env::main_phase(uvm_phase phase);
    my_sequence my_seq;
    super.main_phase(phase);
    my_seq = new("my_seq");
    my_seq.starting_phase = phase;
    my_seq.start(i_agt.sqr);
endtask
```

在这种方式中，首先把 sequence 实例化，之后把 main_phase 的输入参数赋值给 my_seq 的 starting_phase，这个暂且先跳过。之后调用 my_seq 的 start 函数，传入的参数是 i_agt.sqr，也即需要指明这个 sequence 会向哪个 sequencer 发送数据。如果不指明，那么这个 sequence 就是一个没头的苍蝇，不知道把产生的数据发送给谁。start 被调用后，my_seq 的 body 开始执行，于是开始发送数据。

上面的这段代码也可以写在 my_sequencer 的 main_phase 中：

```
task my_sequencer::main_phase(uvm_phase phase);
    my_sequence my_seq;
    super.main_phase(phase);
    my_seq = new("my_seq");
    my_seq.starting_phase = phase;
    my_seq.start(this);
endtask
```

当写在 my_sequencer 的 main_phase 中时，start 的参数变为了 this，这是唯一改变的地方。

在 my_sequence 的定义和启动时都出现了 startint_phase。UVM 中使用 phase 来控制验证平台的关闭。如前面所见，driver，monitor，reference model 和 scoreboard 的 main_phase 都是无限循环的，一个无限循环的验证平台是不可能停止下来的。

在verilog中，由于always语句的存在，其实质就是所有的模块都是一直在无限运行的，要想让整个仿真停止，就要调用$finish函数。在systemverilog中，这个结论依然成立。所以即使driver，monitor的main_phase中都是无限循环的也没关系，只要调用$finish就可以结束仿真。那么什么时候时候调用呢？很显然，当在sequence中发送完我们期望数量的数据时，我们就希望调用$finish。在UVM中，我们不需要显式的调用$finish，只需要使用objection机制即可。在发包之前，通过调用starting_phase.raise_objection(this)告诉UVM要开始发包了，在发送完之前不能调用$finish。在包发送完毕后，调用starting_phase. drop_objection(this)来告诉UVM可以调用$finish了。当调用drop_objection时，UVM会检查一下其它的component的objection是否已经被drop了，如果没有drop，那么不会调用 $finish，只有当所有的objection已经被drop了，才会调用$finish[1]。

☆：UVM 中通过 sequence 产生数据，转交给 sequencer，并最终交给 driver 发送出去。在 sequence 中一般通过 raise_objection 和 drop_objection 来控制仿真平台的关闭。

上面所说的启动 sequence 是一种手动启动的方式，除了手动启动之外，下节将会看到，通过某些设置，UVM 验证平台可以自动启动一个 sequence。

1.3.11. UVM 验证平台中的 case

我们知道，一个 dut 可能有许多的功能，我们的验证环境不可能只跑一次就把所有的功能都给验证完成了。不同的功能我们在不同的 case 中验证。比如在我们的例子中，有的 case 我们需要测试长包，有的 case 测试短包，有的 case 测试 crc 错误的包等。这些 case 都可以列出来。在实际应用中，这些所有的 case 都基于一个基类：uvm_test。先定义一个 my_test 类：

```
1 `include "uvm_macros.svh"
2 import uvm_pkg::*;
```

[1] 更准确的说，在所有的 objection 被 drop 后会进入下一个 phase，当所有的 phase 都执行完毕后才调用$finish，初学者可以暂且认为当所有的 objection 被 drop 后会调用$finish。

```
 3
 4 class my_test extends uvm_test;
 5
 6     my_env              env;
 7     extern function new(string name = "my_test", uvm_component parent = null);
 8     extern virtual function void build_phase(uvm_phase phase);
 9     `uvm_component_utils(my_test)
10 endclass
11
12 function my_test::new(string name = "my_test", uvm_component parent = null);
13     super.new(name,parent);
14     env = new("env", this);
15 endfunction
16
17 function void my_test::build_phase(uvm_phase phase);
18     super.build_phase(phase);
19 endfunction
20
```

这个类很简单，只是把 my_env 实例化。实际使用的 my_test 会比这个复杂些，通常会设置一些诸如 drain_time，uvm_info 优先级设置，仿真输出 log 信息存放文件等信息。这里简单起见，只实例化 my_env。

当定义好了 my_test 后，就可以定义一个 case 了：

```
26 class my_case0 extends my_test;
27     extern function new(string name = "my_case0", uvm_component parent = null);
28     extern virtual function void build_phase(uvm_phase phase);
29     `uvm_component_utils(my_case0)
30 endclass
31
32 function my_case0::new(string name = "my_case0", uvm_component parent = null);
33     super.new(name,parent);
34 endfunction // new
35
36 function void my_case0::build_phase(uvm_phase phase);
37     super.build_phase(phase);
38
39     uvm_config_db#(uvm_object_wrapper)::set(this, "env.i_agt.sqr.main_phase", "default_sequenc
e", my_sequence::type_id::get());
40 endfunction
41
```

这里定义了一个名字为 my_case0 的 case，其核心在于 39 行的 config_db 的 set 语句。这一行的意思就是通知 env.i_agt.sqr，让其在运行到 main_phase 时自动启动前面定义的 my_sequence。这里就是 sequence 的自动启动了。当运行到 main_phase 时，my_sequencer 会自动执行下面的语句：

```
task my_sequencer::main_phase(uvm_phase phase);
```

```
    my_sequence my_seq;
    super.main_phase(phase);
    my_seq = new("my_seq");
    my_seq.starting_phase = phase;
    my_seq.start(this);
endtask
```

因此经过上面的 config_db 的 set，可以把这些语句省略了。为什么要经过这么复杂的设置？很简单，一个验证平台中的 case 中可能有几百个，但是在 my_sequencer 的 main_phase 中只能写一个 my_sequence，在 my_case0 中用到了这个 sequence，但是假如在 my_case1 中用到了 my_sequence1，那么应该如何写呢？很显然，在 my_sequencer 及任何 component（如 env）的 main_phase 中启动一个 sequence 都是不方便的。通过 config_db 的 set 方式，让 sequencer 自动启动 sequence 是一个比较好的选择。如下所示定义了 my_case1：

```
 1 `include "uvm_macros.svh"
 2 import uvm_pkg::*;
 3
 4 class my_sequence1 extends uvm_sequence #(my_transaction);
 5     my_transaction m_trans;
 6
 7     extern function new(string name = "my_sequence1");
 8     virtual task body();
 9         if(starting_phase != null)
10             starting_phase.raise_objection(this);
11         repeat (10) begin
12             `uvm_do_with(m_trans, { m_trans.pload.size() == 60;})
13         end
14         #100;
15         if(starting_phase != null)
16             starting_phase.drop_objection(this);
17     endtask
18
19     `uvm_object_utils(my_sequence1)
20 endclass
21
22 function  my_sequence1::new(string name= "my_sequence1");
23     super.new(name);
24 endfunction // new
25
26 class my_case1 extends my_test;
27     extern function new(string name = "my_case1", uvm_component parent = null);
28     extern virtual function void build_phase(uvm_phase phase);
29     `uvm_component_utils(my_case1)
30 endclass
31
32 function my_case1::new(string name = "my_case1", uvm_component parent = null);
33     super.new(name,parent);
34     $display("SHORT_SEQ::NEW");
```

```
 35 endfunction // new
 36
 37 function void my_case1::build_phase(uvm_phase phase);
 38     super.build_phase(phase);
 39
 40     uvm_config_db#(uvm_object_wrapper)::set(this, "env.i_agt.sqr.main_phase", "default_sequenc
e", my_sequence1::type_id::get());
 41 endfunction
 42
```

1.3.12. UVM 验证平台中的 top

当使用仿真器进行仿真时，系统进行的第一个 module 就是 top（默认情况下，也可以指定其它名字，如 tb_top）。本例中的 top 如下：

```
 24 module top;
 25     reg clk;
 26     my_if  my_my_if(clk, clk);
 27     dut  my_dut(.clk(clk),
 28     .rxd(my_my_if.rxd),
 29     .rx_dv(my_my_if.rx_dv),
 30     .txd(my_my_if.txd),
 31     .tx_en(my_my_if.tx_en)
 32     );
 33     initial begin
 34         clk = 0;
 35         forever begin
 36             #10; clk = ~clk;
 37         end
 38     end
 39     initial begin
 40         uvm_config_db#(virtual my_if)::set(null, "uvm_test_top.env.i_agt.drv", "my_if", my_my
_if);
 41         uvm_config_db#(virtual my_if)::set(null, "uvm_test_top.env.o_agt.mon", "my_if", my_m
y_if);
 42         run_test();
 43     end
 44 endmodule
 45
```

在这个 top 中，27 到 32 行把 DUT 实例化，实例化时把 DUT 的输入输出端口和 my_if 连接在了一起，40 和 41 行通过 config_db 的 set 方式把 my_if 通知 driver 和 monitor，从而 driver 和 monitor 可以直接和 DUT 通信。这里的 set 与 driver 和 monitor 是互成一对的。如果不 set，而在 driver 中只 get，那么是 get 不到任何东西的。

33 到 38 行用于产生 DUT 所需要的时钟。整个 top 文件最关键的地方在于 42 行。

这是 UVM 中的一个全局函数，当仿真器进行到 top 后，产生运行 initial 里面的语句，当运行到 run_test 后，开始启动 UVM。

1.3.13. UVM 验证平台的启动

前面 11 小节讲述了 UVM 验证平台中用到的基本的概念。在介绍 driver 时，提到过一个 uvm_component 派生来的类要定义好三个方法：new，build_phase 和 main_phase。为什么要定义这三个方法？这三个方法被定义后又是如何被调用的呢？当 UVM 验证平台启动后，会自动的调用这几个方法，即先调用 new，再调用 build_phase，之后依次是 connect_phase，main_phase 等。这些方法是自动被调用的，无需手工干预。

☆：UVM 验证平台中 uvm_component 的一系列方法，如 new，build_phase，connect_phase，main_phase 等是自动被调用的。

那么验证平台又是如何启动的呢？当定义好一个 case 时，我们通常使用如下的方式启动这个 sequence（以 questa 为例）：

```
vsim +UVM_TESTNAME=my_case0 …
```

也就是说在编译完成后，通过在命令行中添加 UVM_TESTNAME 指定 case 的名字来启动这个 case。上节说到，仿真器首先进入 top，当执行到 run_test 后，开始启动 UVM 验证平台。UVM 验证平台根据输入的+UVM_TESTNAME 后的字符串创建一个类的实例，如在本例中将会创建一个 my_case0 的实例，假如后面的字符串是 my_case1，那么将会创建一个 my_case1 的实例。无论是 my_case0，还是 my_case1，其实例的名字都是 uvm_test_top，所以在上节 top 的 40 行 41 行在索引 monitor 和 driver 路径时，出现了 uvm_test_top。当 my_case0 的实例被创建后，接下来自动执行 my_case0 的 build_phase，创建 env。build_phase 执行完毕后，自动执行 env 的 build_phase，创建 env 下属的各成员变量。如此轮回，自上而下的执行所有 component 的 build_phase。当所有的 build_phase 执行完毕后，整个完整的 UVM 树就建立完成了，本例中的 UVM 树如下图所示：

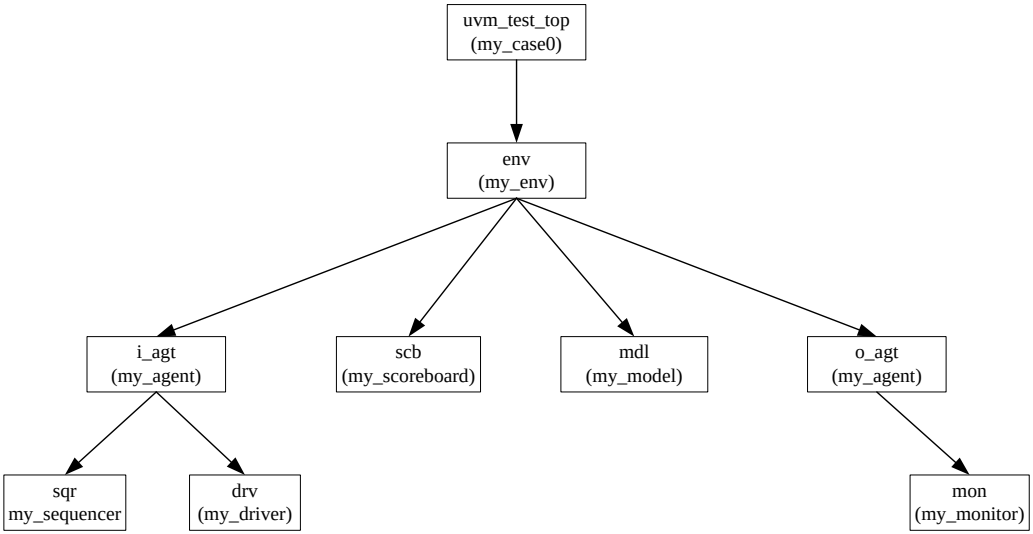


图 1-8 加入 case 概念的 UVM 树

在执行完 build_phase 后，再依次执行 connect_phase，之后执行 main_phase 等。当所有的 phase 都执行完毕后，结束仿真。整个的运行过程如下图所示：

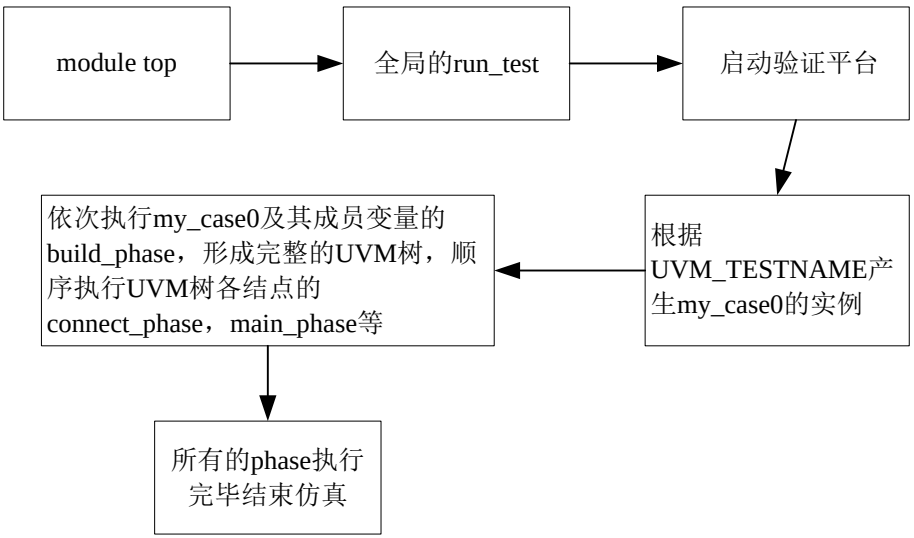


图 1-9 UVM 验证平台执行流程

1.3.10 中曾经说过，当所有的 objection 被 drop 后，将会调用$finish 函数，当时的脚注里说这种说法是不严谨的，确实如此，准确的说法是当所有的 objection 被 drop

后，将会转下下一个 phase，例如，当 main_phase 的 objection 都被 drop 后，将会进入 post_main_phase。当所有的 phase 都执行完毕后才会调用$finish 函数。

2.component 与 object

component 与 object 是 UVM 中两大最重要的概念，也是初学者最容易混淆的两个概念。本文第一节将会介绍 UVM 的树形组织结构，重点介绍 uvm_component。第二节将会介绍 uvm_object，并介绍 uvm_object 与 uvm_component 的区别。第三节将会介绍一下常用的 uvm_component 和 uvm_object，并且指明其作用。第四节将会简单的介绍一下 factory 机制。而第五节将会思考一下 uvm_object 与 uvm_component，为什么会是这样分成两大类而不是只有一个类？

2.1. UVM 的树形组织结构

在第一章中曾经提到过，UVM 是采用树形的组织结构来管理验证平台的各个部分。driver，monitor，agent，model，scoreboard，env 等都是树的一个结点。为什么要用树的形式来组织 driver 等？因为做为一个验证平台来说，它必须掌握自己治下所有的“人口”，只有这样做了，才利于大家统一步伐做事情，才能在发号施令的时候不会有漏网之鱼出现。而树形是实现这种管理的一种比较简单的方式。

2.1.1. uvm_component 中的 parent

UVM 通过 uvm_component 来实现树形结构。所有的 UVM 树的结点都是一个 uvm_component。每个 uvm_component 都有一个特点：他们在 new 的时候，需要指定一个类型为 uvm_component，名字是 parent 的变量：

```
function new(string name, uvm_component parent);
```

一般的，在使用的时候，parent 通常都是 this。假设我们有一个 A 派生自 uvm_component，在 A 中有一个 uvm_component B 的成员变量，即 A 的定义如下：

```
class A extends uvm_component;
    uvm_component B;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B = new("B", this);
    endfunction
endclass
```

在 B 实例化的时候，就把 this 指针传递给了 B，代表 A 是 B 的 parent。有兴趣的读者曾经可能会问：为什么要指定这么一个 parent 呢？B 是 A 的成员变量，那么很明显，A 就是 B 的 parent 了，就不用再在 new 的时候指定了。关于这个问题，可以在第 10 章 uvm_component 源代码分析找到答案。

2.1.2. UVM 树的根在哪里？

前面已经提到多次，UVM 树是以树的形式组织在一起的。作为一棵树来说，其树根在哪里？其树叶又是哪些呢？在第一章的例子中，我们看到，似乎树根应该就是 uvm_test。在 test 里面实例化 env，在 env 里面实例化 scoreboard，reference model，agent，在 agent 里面实例化 driver 和 monitor。scoreboard，reference model，driver 和 monitor 都是树的叶子，树到此为止，没有了更多的叶子。

关于叶子的判断，我们是正确的。但是关于树根的推断，我们是错误的。UVM 中真正的树根是一个称为 uvm_top 的东西，图 1-8 的完整版本如下：

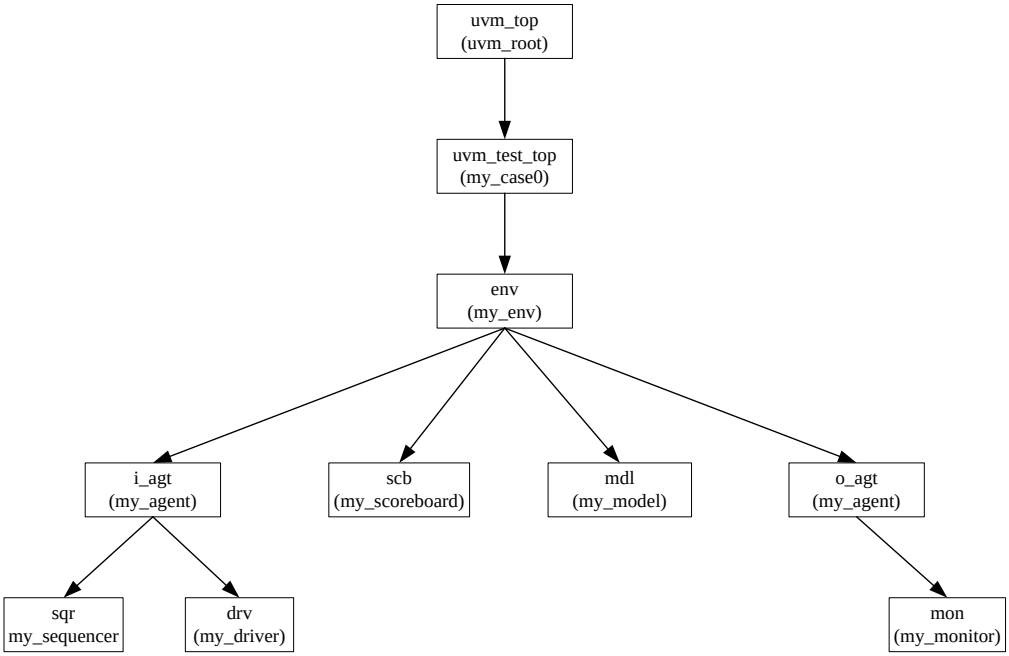


图 2-1 完整的 UVM 树

uvm_top 是一个全局变量，它是 uvm_root 的一个实例，而 uvm_root 派生自 uvm_component，所以 uvm_top 本质上是一个 uvm_component，它是树的根。uvm_test_top 的 parent 是 uvm_top，而 uvm_top 的 parent 则是 null。至于为什么不以 uvm_test 派生出来的 case（即 uvm_test_top）作为树根，而是搞了这么一个奇怪的东西做为树根，可以参考 10.2 节 uvm_root 源代码分析。

2.1.3. uvm_component 的 phase 自动执行

uvm_component 有两大特性，除了上面提到过的在 new 的时候，要指定一个 parent 外，另外一个重要的特性就是它具有 phase 自动执行的特性。当整棵树的 build_phase 执行完的时候，会自动执行 connect_phase，以此类推。这样，对于一些有执行顺序要求的语句，可以分别在不同的 phase 完成。这样可以最大程度上避免出错。关于 phase 的更多的解释，可以参考第 3 章 phase 及 objection 中的相关介绍。

2.2. uvm_object 是 UVM 中最最基本的类

2.2.1. uvm_object 与 uvm_component 是两个对等的概念吗

一般的认识中，与 uvm_component 相对等的概念就是 uvm_object。当我们自己创建一个类的时候，比如定义一个 sequence 类，一个 driver 类，要么这个类派生自 uvm_component（或者 uvm_component 的派生类，如 uvm_driver），要么这个类派生自 uvm_object（或者 uvm_object 的派生类，如 uvm_sequence）。这给我们一个感觉，似乎 uvm_object 与 uvm_component 是对等的概念，是两个对立的东西。其实不然。

uvm_object 是 UVM 中最最基本的类，你能想到的几乎所有的类都是继承自 uvm_object，包括 uvm_component。uvm_component 派生自 uvm_object 这个事实会让很多人惊讶，而这个事实说明了 uvm_component 有 uvm_object 的所有的特性，同时又有自己的一些性质。但是对于 uvm_component 的一些特性，uvm_object 则不一定具有。这是面向对象编程中经常用到的一条规律。

uvm_component 有两大特性，一是通过在 new 的时候指定 parent 来形成一种树形的组织结构，二是有 phase 的自动执行特点。uvm_object 则完全不具有这些概念。所以，你可以想像的出，如果一个类派生自 uvm_object，那么这个类的实例不可能以结点的形式出现在 UVM 树上。因为所有的 UVM 树的结点都是由 uvm_component 组成的。

图 2-2 列出了部分 UVM 类的继承关系。uvm_void 是所有常用类的基类。图中的红色表示搭建验证平台时经常使用的类，在第一章的例子中，已经实际的使用过这些类。

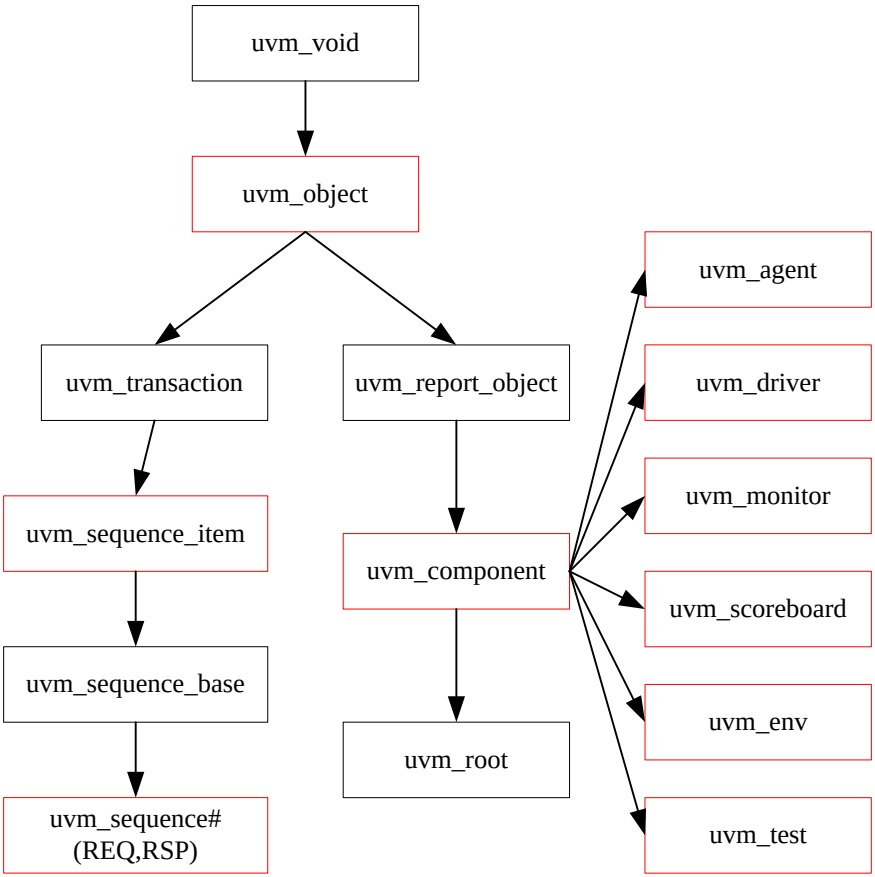


图 2-2 UVM 中常用类的继承关系

2.2.2. 有哪些类派生自 uvm_object

既然 uvm_object 是最最基本的类，那么其能力恰恰也是最差的，当然了，其扩展性也是最好的。恰如一个婴儿，其能力很差，但是我们可以把其尽量培养成书法家，艺术家等等。

说到现在 uvm_object 依然是一个相当抽象的类。验证平台中用到的什么类会派生自 uvm_object？答案是除了派生自 uvm_component 类之外的类(确实很废话-_-，本节一开始就说了，现在竟然又重复了一遍)，几乎所有的类都派生自 uvm_object。换个说法，除了 driver，monitor，agent，model，scoreboard，env 之外的几乎所有的阿猫阿狗的类，本质上都是 uvm_object，如 sequence，sequence_item，transaction，config

等。

如果你现在依然对 uvm_object 很迷茫的话，那么举一个更加通俗点的例子，uvm_object 是一个分子，用这个分子可以搭建成许多许多的东西，如既可以搭建成动物，还可以搭建成植物，更加可以搭建成没有任何意识的岩石，空气等。uvm_component 就是由其搭建成的一种高级生命，而 sequence_item 则是由其搭建成的血液，它流通在各个高级生命(uvm_component)之间，sequence 则是众多 sequence_item 的组合，config 则是由其搭建成的用于规范高级生命(uvm_component)行为方式的准则（法律条文）。

2.3. 经常用到的 uvm_object 和 uvm_component

前面其实已经分散的提到过了，这里将会总结一下。

2.3.1. 常用的 uvm_component

uvm_driver：所有的 driver 都要派生自 uvm_driver。driver 的功能主要就是向 sequencer 索要 sequence_item(transaction)，并且把 sequence_item 里的信息驱动到 DUT 的接口上，这相当于完成了从 transaction 级别到 DUT 能够接受的 pin 级别的信息的转变。

uvm_monitor：所有的 monitor 都要派生自 uvm_monitor。monitor 做的事情与 driver 相反，driver 向 DUT 的 pin 上发送数据，而 monitor 则是从 DUT 的 pin 上接收数据，并且把接收到的数据转换成 transaction 级别的 sequence_item，并且把转换后的数据发送给 scoreboard，供 scoreboard 比较。

uvm_sequencer：所有的 sequencer 都要派生自 uvm_sequencer。sequencer 的功能就是组织管理 sequence，当 driver 要求数据时，它就把 sequence 生成的 sequence_item 转发给 driver。

uvm_scoreboard：一般的 scoreboard 都要派生自 uvm_scoreboard。scoreboard 的功能就是比较 reference model 和 monitor 分别发送来的数据，根据比较结果判断 DUT 是否正确工作。

reference model：reference model 直接派生自 uvm_component。reference model

的作用就是模仿 DUT，完成与 DUT 相同的功能。DUT 是用 verilog 写成的时序电路，而 reference model 则可以直接使用 systemverilog 高级语言的特性，同时还可以通过 DPI 等接口调用其它语言来完成与 DUT 相同的功能。

uvm_agent：所有的 agent 要派生自 uvm_agent。与前面几个比起来，uvm_agent 的作用并不是那么明显。它只是把 driver 和 monitor 封装在一起，根据参数值来决定是只实例化 monitor 还是要实例化 driver 和 monitor 呢？这个主要是从可重用性的角度来考虑的。如果在做验证平台的时候不考虑可重用性，那么 agent 其实是可有可无的。

uvm_env：所有的 env 要派生自 uvm_env。env 是 environment 的缩写。env 把验证平台上用到的固定不变的 component 都封装在一起。这样，当要跑不同的 case 时，只要在 case 中实例化一个 env 就可以了。

uvm_test：所有的 case 要派生自 uvm_test。case 与 case 之间差异很大，所以从 uvm_test 派生出来的类各不相同。任何一个派生出的 case 中，都要实例化 env，只有这样，当 case 在运行的时候，才能把数据正常的发给 DUT，并正常的接收 DUT 的数据。

2.3.2. 常用的 uvm_object

2.2.2 节中说到了几种 uvm_object：

uvm_sequence_item：所有 的我们自己定义的 transaction 要从 uvm_sequence_item 派生。transaction 就是封装了一定信息的一个类，如一个 mac transaction 就是把一个 mac 帧封装在了一起，包括目的地址，源地址，帧类型，帧的数据，FCS 校验和等。driver 从 sequencer 中得到 transaction，并且把其转换成 pin 级别的信号。需要注意的是，UVM 中有一个 uvm_transaction 类，在以前的实现（OVM 的较老的版本）中，是可以直接从 uvm_transaction 来派生一个自己的 transaction，但是 在 UVM 中，是强烈不推荐从 uvm_transaction 派生一个 transaction，而要从 uvm_sequence_item 派生。事实上，uvm_sequence_itme 是从 uvm_transaction 派生而来的，因此，uvm_sequence_item 相比 uvm_transaction 添加了很多实用的成员变量和函数，从 uvm_sequence_item 直接派生，就可以使用这些新增加的成员变量和函数。

uvm_sequence：所有的 sequence 要从 uvm_sequence 派生一个。sequence 就是 sequence_item 的组合。sequence 直接与 sequencer 打交道，当 driver 向 sequencer 索要数据的时候，sequencer 会转而向 sequence 要数据，sequence 发现有 sequence_item 时，会把此 sequence_item 传递给 sequencer，并最终给 driver。

config：所有 的 config 一般直接从 uvm_object 派生。config 的主要功能就是规范

验证平台的行为方式。如规定 CPU 的 driver 在读取总线时地址信号要持续几个时钟，片选信号从什么时候开始有效等等。

除了上面几种类是派生自 uvm_object 外，还有下面几种：

uvm_reg_item：它其实派生自 uvm_sequence_item。

uvm_reg_map，uvm_mem，uvm_reg_field，uvm_reg，uvm_reg_file，uvm_reg_block 等与寄存器相关的众多的类都是派生自 uvm_object。

uvm_phase：它派生自 uvm_object，其用处主要是控制 uvm_component 的行为方式，使得 uvm_component 平滑的在各个不同的 phase 之间依次运转。

除了这些之外，其实还有更多的。不过其它的一些并不那么重要，这里不一一列出。

2.4. factory 机制

什么是 factory 机制？它是用来干吗的？当我自己学习 UVM 的时候，着实被 factory 机制给困扰了好久，始终不得其解。询问身边的人时，他们都说，factory 就是一张表格。然后一次又一次的解释表格是做什么用的，结果就是越解释越糊涂。其实，对于初学者来说，factory 只是一个宏。当涉及到 set_override 等操作时，才有必要去理解 factory 机制的原理。factory 机制更多的是体现在内部编程应用上，它为众多其它机制的实现提供了可能。换句话说，factory 机制是 UVM 的内功。那么 UVM 中的外功是什么？典型的外功是 field automation 机制，将会在 4.1 节介绍。

2.4.1. UVM 认证准生证

在定义一个类的时候，UVM 强烈推荐使用 uvm_component_utils 或 uvm_object_utils 宏来把它们注册。通过这两个宏，UVM 就知道我们自己定义一个类，这个类或是派生自 uvm_component（使用 uvm_component_utils 注册），要么就是派生自 uvm_object（使用 uvm_object_utils 注册）。那么注册之后有什么用处呢？

```
class A extends uvm_component;
    `uvm_component_utils(A)
    …
```

```
endclass
```

当使用如上代码注册 A 后，那么要创建一个 A 的实例可以这样做：

```
A a;
a = A::type_id::create("a", this);
```

还记得前面说过的 uvm_component 的 new 中的两个参数吗？一个是名字，一个是指向 parent 的指针。这里，create 的两个参数也是同样的两个参数。事实上，这个 create 会直接调用 A 的 new 函数。上面这种实例化的方法有点古怪，尤其是 type_id 是什么东西？了解这个要仔细的看源代码，不过这个 type_id 是固定的，假如有一个 B 类也同样是通过 uvm_component_utils 宏注册了：

```
class B extends uvm_component;
    `uvm_component_utils(B)
    …
endclass
```

那么 B 可以如下实例化：

```
B b;
b = B::type_id::create("b", this);
```

可见，type_id 是固定的。所以除非想研究源代码，否则就可以稍微把好奇心放一下，不要纠结于为什么会如此实例化的问题了。

假如前面的 A 根本没有使用 uvm_component_utils 实例化，那么 A 要实例化只能使用这种方式：

```
A a;
a = new("a", this);
```

这种想法看上去比上一种法子简单一些，不过上一种法子才是 UVM 推荐的。看来看去，似乎就是在实例化的时候写法变了，注册过后，就可以使用 UVM 认证的方式实例化，就像是得到了一张 UVM 认证的准生证一样；而如果没有注册，那么实例化的时候就只能使用 systemverilog 实例化的方法了。两种实例化的方法，使用前者得到的实例，可以使用 UVM 中的众多功能，而后者则不行。就像是在一个大医院里出生的孩子天生可以得到较好的医疗条件照顾，而一个在乡村卫生所出生的孩子享受的医疗资源较差。

2.4.2. override 功能

假设我们自己定义了一个 my_driver，在跑 80%的 case 的时候，这个 driver 是足够使用的。但是在剩余的 20%的 case 中，我们需要对 my_driver 的行为作出某些改

变，这时就可以用到 override 功能。使用 override 功能的第一步是要先从 my_driver 派生出一个类，把这个类的行为定义好：

```
class new_driver extends my_driver;
    …
    `uvm_component_utils(new_driver)
endclass
```

之后，在具体的 case 的 build_phase 中，调用 override 相关的函数：

```
class case_x extends base_test;
    function void build_phase(uvm_phase phase);
        …
        set_type_override_by_type(my_driver::get_type(), new_driver::get_type());
    endfunction
endclass
```

经过上述过程之后，那么在跑 case_x 的时候，系统中运行的 my_driver 就是 new_driver 类型的，其行为是 new_driver 的行为。不过这有一个前提，那就是 my_driver 在它的 agent 中实例化的时候，要使用 factory 的方式实例化：

```
class my_agent;
    my_driver drv;
    function void build_phase(uvm_phase phase);
        …
        drv = my_driver::type_id::create("drv", this);
    endfunction
endclass
```

假如不是使用上面的写法，而是使用 drv=new("drv", this)的写法进行实例化，那么 overide 功能是不能实现的。所以这也就是 UVM 为什么强烈推荐使用它独有的那种古怪的实例化的方法的一个原因。

2.4.3. 根据类名创建类的实例

在第一章最后一节中，我们曾经稍微的分析了一下 UVM 的仿真过程，当进入到 run_test 时，系统会根据输入的+UVM_TESTNAME=name 来创建一个 name 的实例。这里初看起来其实是挺容易的，但是仔细想一下，就会发现这里面的技术难度很大。要通过一个字符串来创建这个字符串所代表的类的一个实例是相当相当困难的一件事情（关于这一点可以看第 12 章 factory 机制源代码分析）。UVM 的 factory 机制实现了这种功能，它提供一个称为 create_component_by_name 的函数，这个函数的输入是一个字符串，通过此函数，可以根据类名来创建一个类的实例。

所以推荐大家派生类的时候，能够用 factory 注册就尽量用 factory 注册，一方面

是它提供了很强大的功能，另外一方面这也是整个验证平台能够运行起来的前提。

2.4.4. factory 的本质：重写了 new 函数

到现在为止，大家至少应该对 factory 有了一个初步的了解。很多人把 factory 的本质理解成是一张表，这种理解从某些方面来说确实没错。因为 factory 的内部实现中就是通过一张表来实现的。但是 factory 仅仅是一张表吗？

factory 可以创建一个新的类，在没有 factory 之前，要创建一个类的实例，只能使用 new 函数。

```
class A;
    …
endclass
A a;
a = new;
```

但是有了 factory 之后，除了可以使用类名创建实例之外，还可以通过一个代表类名字的字符串来进行实例化。除此之外还可以进行 override 等功能。

所以，从本质上来看，factory 机制其实是对 systemverilog 中 new 函数的重载。因为这个原始的 new 函数实在是太简单了，功能太少了。经过 factory 机制的改良之后，进行实例化的方法多了很多。这也体现了 UVM 编写的一个原则，一个好的库应该提供更多方便实用的接口，这种接口一方面是库自己写出来并开放给用户的，另外一方面就是把语言原始的接口给改良一下，使得更加方便用户的使用。

factory 的本质，就是重写 systemverilog 的 new 函数。关于这一点的详细内容，请参看本书第 12 章 factory 机制源代码分析。

2.5. uvm_component 与 uvm_object 的思考

为什么 UVM 中会分成 uvm_component 与 uvm_object 两大类呢？从古至今，人类在探索世界的时候，总是在不断的寻找规律，并且通过所寻找到的规律来把所遇到的事物，所看到的现象分类。因为世界太复杂，只有把有共性的万物分类，从而按照类别来认识万物，这样才能大大降低人类认识世界的难度。比如世界的生命有千万种，但是只有动物和植物两类。遇到一个生命的时候，我们会不自觉的判断它

是一个动物还是植物，并且把动物或植物的特性预加到这种生命的身上，接下来用动物或者植物的方法来研究这个生命，从而加快对于这个生命的认知过程。

UVM 很明显吸收了这种哲学，分类，然后分别管理。想像一下，假如 UVM 中不分 uvm_object 与 uvm_component，所有的东西都是 uvm_object，那是多么恐怖的一件事情？这相当于我们直接与分子打交道！废时废力，不易于使用。

systemverilog 做为一门编程语言，相当于是提供了最基本的原子，其使用起来相当麻烦。为了减少这种麻烦，我们有了 UVM，假如 UVM 中全部都是 uvm_object 的话，也就是全部都是分子，分子虽然比原子好用一些，但是依然是超脱于普通人的承受范围之外。只有当我们把分子组合成一个又一个生命体的时候，用起来才会比较顺手。

uvm_component 那么好用，为什么不把所有的东西都做成 uvm_component 的形式呢？因为 uvm_component 是高级生命体，有其自己鲜明的特征。验证平台中并不是所有的东西都有这种鲜明的特征。一个简单的例子：uvm_component 在整个仿真中是一直存在的，但是假如我们要发送一个 transaction(激励)给 DUT，此 transaction(激励)可能只需要几毫秒就可以发送完。发送完了，此 transaction(激励)的生命周期几乎就结束了，根本没有必要在整个仿真中一直持续下去。生命是多样化的，要既允许 uvm_component 这样的高级生命存在，也要允许 transaction 这种如流星一闪而逝的东西存在。

3.phase 及 objection

本章将介绍 UVM 中两个应用非常广的概念 phase 和 objection。这两个概念与 UVM 验证平台息息相关，phase 恰如铁轨，让 UVM 这趟列车在铁轨上向前运行，不会脱轨，不会跳过某一段而直接到达后一段，objection 则更像是能量，给列车提供能量，当控制着这趟列车何时终止。

第一节首先介绍一下 phase，并回答几个关于 phase 的基本的问题。第二节将会介绍 objection。

3.1. UVM 中的 phase

在第一章的例子中，我们已经提到过 phase 自动执行的概念。为什么要分成 phase？为什么要让 phase 自动执行？phase 自动执行有什么好处？为什么只有 uvm_component 才有 phase 的概念？

3.1.1. 为什么要分成 phase

verilog中有非阻塞赋值和阻塞赋值，相对应的，在仿真器中要实现NBA区域和

Active区域[1]，这样在不同的区域做不同的事情，可以避免竞争关系的存在导致的变量值不确定的情况。同样的，一个验证平台是很复杂的，要搭建一个验证平台是一件相当繁杂的事情，要正确的掌握这些步骤并理顺它们是相当艰难的一个过程。

举一个最简单的例子，一个 env 下面会实例化 agent，scoreboard，reference model 等，agent 下面又会有 sequencer，driver，monitor。而且这些组件之间还有连接关系，如 agent 中 monitor 的输出要送给 scoreboard 或 reference model，这种通信的前提是要先把两者连接起来，reference model 要和 scoreboard 连接在一起等。那么我们可以如下写：

```
scoreboard = new;
reference_model = new;
reference_model.connect(scoreboard);
agent = new;
agent.driver = new;
agent.monitor = new;
agent.monitor.connect(scoreboard);
```

这里面反应出来的问题就是最后一句话一定要放在最后写，因为连接的前提是所有的组件已经实例化了。但是相应的，reference_model.connect(scoreboard)的要求则没有那么高，只需要在上面代码中 reference_model = new 之后的任何一个地方写就可以了。我们可以看出，代码的书写顺序会影响代码的实现。

要把代码顺序的影响降低到最低，可以如下写：

```
scoreboard = new;
reference_model = new;
agent = new;
agent.driver = new;
agent.monitor = new;

reference_model.connect(scoreboard);
agent.monitor.connect(scoreboard);
```

看到了么？只要把与连接语句放在最后两行写就没有关系了。UVM 采用了这种方法，它把前面的实例化的部分都放在了 build_phase 来做，而连接关系放在了 connect_phase 来做。这就是 phase 最初始的来源。

在不同时间做不同的事情，这就是 UVM 中 phase 的设计哲学。UVM 中常用的 phase 如下：

[1] 可以参照《IEEE Std 1364-2001 IEEE Standard Verilog® Hardware Description Language》

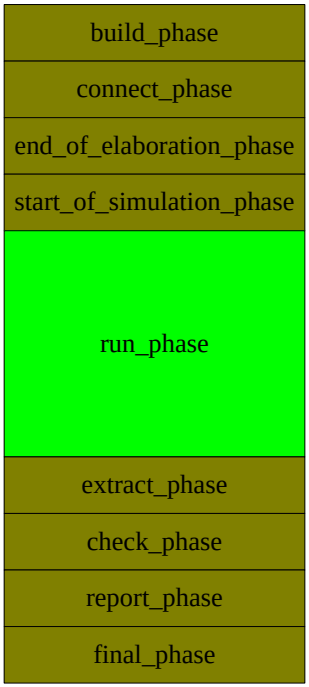


图 3-1 UVM 中的常用 phase

3.1.2. task phase 和 function phase

uvm中的phase，按照其是否消耗仿真时间 [1]的特性，可以分成两大类，一类是function phase，如build_phase，connect_phase等，这些phase都不耗费仿真时间，其实现是通过函数（function）来实现的；另外一类是run_phase等，它们是耗费仿真时间的，其实现是通过任务（task）来实现的。我们给DUT施加激励，监测DUT的输出都是在这里完成的。在图 3-1 中，只有绿色的run_phase是task phase，其它的都是function phase。

[1] 这里要区分仿真时间和运行时间。所谓仿真时间就是使用$time 函数等到的时间，而运行时间则是 CPU 的时间。直观上，我们觉得一个程序运行的快，就是 CPU 时间花费的少，即运行时间少。

3.1.3. phase 的自动执行

图 3-1 所示的 phase，是顺序执行的。只要把代码填入相应的 phase，那么这些代码就会自动执行。这看起来很强大，也是非常符合我们的思维习惯的。仍以上节的例子为例：

```
scoreboard = new;
reference_model = new;
agent = new;
agent.driver = new;
agent.monitor = new;

reference_model.connect(scoreboard);
agent.monitor.connect(scoreboard);
```

上面这段代码当 new 语句执行完成后，后面的 connect 肯定就会自动执行。现在我们引入了 phase 的概念，把前面 new 的部分包裹进了 build_phase 里面，把后面的 connect 包进了 connect_phase 里面，很自然的，当 build_phase 执行完了就应该自动执行 connect_phase。

phase 的引入在很大程度上解决了代码顺序杂乱可能会引发的问题。它本质上是通过把代码顺序强制固定来实现这个目的的，如 build_phase 的代码一定在 connect_phase 之前执行，而 connect_phase 的代码一定在 end_of_elaboration_phase 之前执行等等。遵循 UVM 的这种代码顺序划分原则，可以在很大程度上减少验证平台开发者的工作量，让其从一部分杂乱的工作中解脱出来。

3.1.4. UVM 中同一 phase 的执行顺序

上面说明了不同 phase 之间的执行顺序，由于 phase 是和 uvm_component 相伴相生的一个概念，对于每一个 uvm_component 来说，都有上面说的 phase。而验证平台中的 component 又是分层次的。如 agent 下面有 driver 和 monitor，那么同样的是 build_phase，先执行 agent 的 build_phase 还是先执行 driver 的 build_phase 或者是先执行 monitor 的 build_phase 呢？

对于这个问题，如果撇开 UVM，那么有很多种答案，如遵循自上而下的顺序，先执行 agent 的，再执行 driver 和 monitor 的，也可以遵循自下而上的顺序，先执行 driver 和 monitor 的，再执行 agent 的或者完全采用一种乱序的方式，按照随机的顺序执行。最后一种方式是不受人控制的，在编程当中，这种不受控制的代码越少越好。因此可以选择的无非就是自上而下或者自下而上。UVM 采用了自上而下的执行

顺序，这种选择其实是唯一的。

假如 UVM 不使用自上而下的方式执行 build_phase，那会是什么情况？UVM 的设计哲学就是在 build_phase 中做实例化的工作，driver 和 monitor 都是 agent 的成员变量，所以它们的实例化都要在 agent 的 build_phase 中执行。如果在 agent 的 build_phase 之前执行 driver 的 build_phase，此时 driver 还根本没有实例化，所以调用 driver.build_phase 只会引发错误。

到了这里，有必要澄清一个概念，那就是 UVM 是在 build_phase 做实例化工作。这里的实例化指的是而且也仅仅指 uvm_component 及其派生类变量的实例化，假如在其它 phase 实例化一个 uvm_component 的话，那么系统会报错的。如果是 uvm_object 的实例化，则可以在任何 phase 完成，当然也包括 build_phase 了。

除了自上而下的执行方式外，UVM 的 phase 还有一种执行方式是自下而上。事实上，除了 build_phase 之外，所有的不耗仿真时间的 phase（即 function phase）都是自下而上执行的。如对于 connect_phase，先执行 driver 和 monitor 的 connect_phase，再执行 agent 的 connect_phase。

细心的读者可能也发现了，build_phase是一种自上而下的执行顺序，但是如果是对于同一层次的uvm_component来说，那么先执行哪一个呢？像driver和monitor都是agent里面处于同一层次。这两者之间是怎么执行的呢？是按照代码书写的顺序吗？还是同时执行？UVM对这一点没有做明确的说明，读者如有兴趣，可以看本书后面关于phase的源代码分析 [1]。

本节前面说的都是不耗仿真时间的 phase，即 function phase 的执行情况，那么对于 run_phase 等的执行，它们是如何执行的呢？事实上，他们也都是按照自下而上的顺序执行。但是与前面的 function phase 自下而上执行不同的是，这种 task phase 由于是耗费时间的，所以它并不是等到“下面”的 phase（如 driver 的 run_phase）执行完了才执行“上面”的 phase（如 agent 的 run_phase），而是把这些 run_phase 通过 fork…join_none 的形式全部的启动起来。所以，自下而上的执行，其更准确的说法是自下而上的启动，同时在运行。

3.1.5. UVM 中的动态运行(run_time) phase

在图 3-1 中，列出了 UVM 中的 phase。事实上，这幅图仅列出了一部分 phase。

[1] 通过源代码，读者可以发现执行顺序是按照字典序的。这里的字典序的排序依据是 new 时指定的名字。假如 monitor 在 new 时指定的名字为”aaa”，而 driver 的名字为”bbb”，那么将会先执行 monitor 的 build_phase。反之 monitor 为”mon”，driver 为”drv”，那么将会执行 driver 的 build_phase。

在 UVM1.0 之前（即 UVM1.0EA 版和 OVM 中）这些 phase 就是所有的 phase 了。在 UVM1.0 中，新加入了一些动态运行的 phase，如图 3-2 所示。

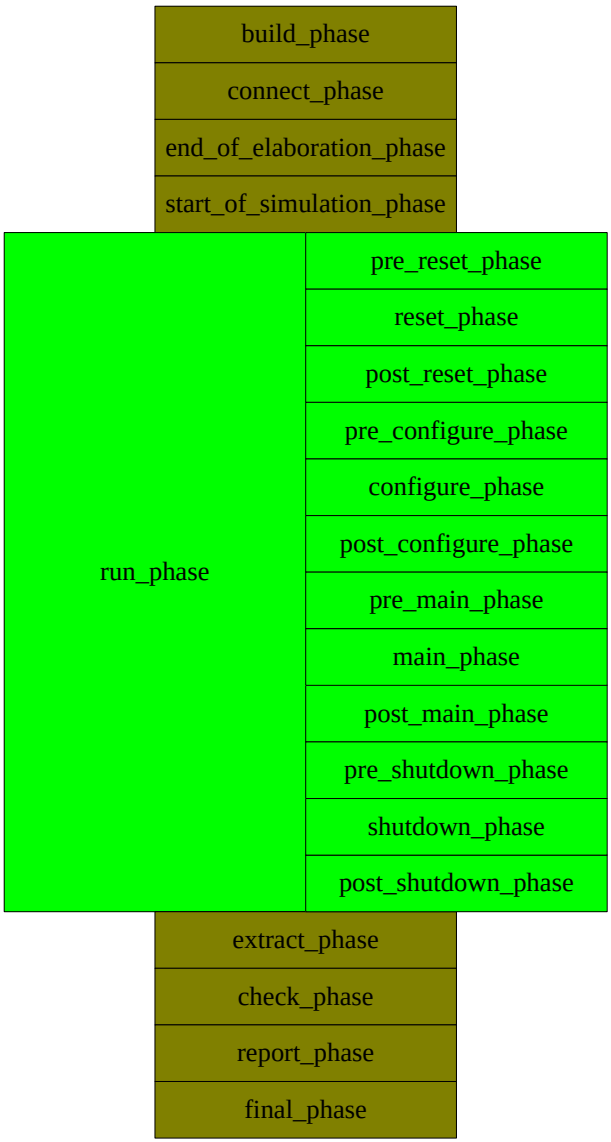


图 3-2 UVM 中所有的 phase

UVM 把 run_phase 又分割成了 12 个小的 phase，这 12 个小的 phase 各自在执行顺序方面与 run_phase 完全相同，即自下而上的启动，同时运行。这里有两个问题，第一个问题是为什么要分成小的 phase？第二个问题是这 12 个小的 phase 与 run_phase 之间关系如何？

分成小的 phase 是为了实现更加精细化的控制。如这 12 个小 phase 的名字所示，reset，configure，main，shutdown 四个 phase 是核心，这四个 phase 通常也是模拟了 DUT 的正常工作方式，在 reset_phase 对 DUT 进行复位，初始化等操作，在 configure_phase 则进行 DUT 的配置，DUT 的运行主要在 main_phase 完成，shutdown_phase 则是做一些与 DUT 断电相关的操作。通过细分，对 DUT 实现更加精确的控制。如，假设要对 DUT 在运行过程中进行一次 reset 操作，在没有这些细分的 phase 之前，这种操作要在 scoreboard，reference model 等加入一些额外的代码来保证验证平台不会出错。但是有了这些小的 phase 之后，分别在 scoreboard 和 reference model 及其它部分（如 driver，monitor 等）的 reset_phase 写好相关代码，之后如果想做一次复位操作，那么只要通过 phase 的 jump，就会自动的跳转回 reset_phase。

这里用到了一个词语：跳转。我们之前的所有的表述中，所有的 phase 都是顺序执行的，前一个 phase 执行完了才执行后一个。但是并没有说到后一个 phase 执行完了还可以再执行一次前面的 phase。而“跳转”这个词则完全打破了这种观念。它告诉我们，phase 之间还可以互相跳来跳去。其实，这种 phase 的跳转是有一定限制的，只能限于这 12 种动态运行 phase 之间互相跳转，而他们之间也不可以跳转到 run_phase。同时，原先的那些不耗费仿真时间的 function phase 之间也是不能跳转的。可以看的出来，这 12 个小的 phase 其实是自成世界的。跳转是一个比较高级的功能，对于初学者来说可以不必考虑。

回答上面的第二个问题，这 12 个动态运行的 phase 与 run_phase 之间有什么关系。从本小节的图中可以看出，这 12 个动态运行的 phase 与 run_phase 之间是并列的关系，这是不是也意味着它们之间的执行也是并列的呢？答案是确定的。对于同一个 component 来说，如果其同时定义了 run_phase 和其它 12 个小的 phase，那么其执行大体如下：

```
fork
    begin
        run_phase();
    end
    begin
        pre_reset_phase();
        reset_phase();
        post_reset_phase();
        pre_configure_phase();
        configure_phase();
        post_configure_phase();
        pre_main_phase();
        main_phase();
        post_main_phase();
        pre_shutdown_phase();
        shutdown_phase();
        post_shutdown_phase();
    end
join_none
```

这段代码只是形象的说明这 12 个小的 phase 与 run_phase 之间的关系，但是有一点要指出的是，这 12 个小的 phase 之间并不是这样顺序执行，而是每当一个小的 phase 执行完成的时候要看看其它 component 的同名的小 phase 有没有执行完，等所有的都执行完后，才会进入下一个小的 phase，也就是说有一个同步的过程。这段代码中并没有体现出这种同步的过程。

3.2. UVM 中的 objection

在没有 UVM 之间，我们自己写 testbench 时，要自己决定什么时候把 testbench 关掉，通常会调用$finish 函数，如下所示：

```
module tob_tb;
    …
    initial begin
        …//give some stimulus to DUT
        $finish();
    end
endmodule
```

UVM 中，通过 objection 机制来控制验证平台的关闭。

3.2.1. objection 是如何控制验证平台的关闭的

objection 字面的意思就是反对，异议。在验证平台中，可以通过放弃异议(drop_objection)来通知系统可以关闭验证平台。当然了，在放弃之前，首先要提起异议。想像一下，假如事先不提异议的话，如果我们跟人交流的时候，忽然说出一句：我放弃刚才的反对意见（异议），对方接着就傻眼了：你刚才不是一句话也没说吗，哪里提出什么反对意见了？所以，在 drop_objection 之前，一定要先提出异议 raise_objection：

```
task main_phase(uvm_phase pahse);
    super.main_phase(phase);
    phase.raise_objection(this);
    …
    phase.drop_objection(this);
endtask
```

在进入到某一 phase 的时候，UVM 会收集此 phase 提出的所有的 objection，并且实时监测所有的 objection 是否已经 drop 了，当发现所有的都已经 drop 后，那么就会关闭此 phase，开始进入下一个 phase。当所有的 phase 都执行完毕后，就会调用$finish 来把整个的验证平台关掉。

如果 UVM 发现此 phase 没有提出任何 objection，那么将会直接跳转到下一个 phase 中。如下所示，假如我们的验证平台中，只有(注意“只有”两个字)driver 中提起了异议，而 monitor 等都没有提起：

```
task driver::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    phase.raise_objection(this);
    #100;
    phase.drop_objection(this);
endtask

task monitor::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    while(1) begin
        …
    end
endtask
```

那么这段代码是能够执行的，当时间过了 100 个单位之后，driver 中的 objection 被 drop 了。此时，UVM 监测发现所有的 objection 都被 drop 了（因为就只有 driver raise 起了这一个 objection），虽然 monitor 中是一个 while 的无限循环，但是 UVM 根本就不会考虑 monitor，而是直接跳到下一个 phase，即 post_main_phase()。假设进入 main_phase 的时刻为 0，那么进入 post_main_phase 的时刻就为 100。如果 drive 根本就没有 raise_objection，并且其它所有的 component 的 main_phase 里面也没有 raise_objection，即 driver 变成如下情况：

```
task driver::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    #100;
endtask
```

那么在进入 main_phase 时，UVM 发现没有任何 objection 被 raise，于是虽然 driver 中有一个延时 100 个单位的代码，虽然 monitor 中有一个无限循环，UVM 都不理会，它会直接跳转到 post_main_phase，假设进入 main_phase 的时刻为 0，那么进入 post_main_phase 的时刻还是为 0！UVM 用户一定要注意：如果想执行一些耗费时间的代码，那么至少也要在此 phase 下 raise 一次 objection。这个结论只适用于 run_time 的 phase。对于 run_phase 则不适用。

3.2.2. 参数 phase 的含义

在 UVM 的所有的 phase 的自动执行函数（任务）的参数中，都有一个 phase：

```
task main_phase(uvm_phase phase);
```

这个输入参数中的 phase 是什么意思？为什么要加入这么一个东西？看了上一小节的例子，应该能够回答这个问题了。因为要便于在任何 component 的 main_phase 中都能 raise_objection，而要 raise_objection 则必须通过 phase.raise_objection 来完成，所以必须把 phase 做为参数传递到 main_phase 等任务中。可以想像，如果没有这个 phase 参数，那么想要 raise 一个 objection 就会比较麻烦了。

这里比较有意思的一个问题是：类似 build_phase 等 function phase 是否可以 raise 和 drop objection 呢？

这个问题的答案是肯定的。在 1.1 版本中是可以的，系统并不会报错。不过，一般不会这么用。phase 的引入是为了解决何时结束仿真的问题，它更多的面向类似 driver 和 scoreboard 在 main_phase 中的无限循环，而不是面向 function phase。

3.2.3. 一般在什么地方 raise_objection

在 3.2.1 节中 以 driver 中 raise_objection 为例进行了说明，但是事实上，在 driver 中 raise_objection 的时刻并不多。这是因为，driver 中通常都是一个无限循环的代码，如下所示：

```
task driver::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    while(1) begin
        seq_item_port.get_next_item(req);
        …//drive the interface according to the information in req
    end
endtask
```

如果是在 while(1)的前面 raise_objection，在 while 循环的 end 后面 drop_objection，那么由于无限循环的特性，phase.drop_objection 永远不会被执行到。

一种常见的思维是把 raise_objection 放在 get_next_item 之后，这样的话，就可以避免无限循环的问题，确实如此。但是关键问题是如果其它地方没有 raise_objection 的话，那么如前面所言，UVM 不等 get_next_item 执行完成就已经跳转到了 post_main_phase。

在 monitor 中，scoreboard 中，reference model 中，都有类似的情况，它们都是无限循环的。要解决这个问题，一种方法是不要让 driver 无限循环，这个就需要知道在每次运行中需要发多少个 item：

```
task driver::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < pkt_num; i++) begin
        seq_item_port.get_next_item(req);
        …//drive the interface according to the information in req
    end
    phase.drop_objection(this);
endtask
```

上例中，只要把 pkt_num 设置成要发送的 transaction 的数量就可以了。这种设置可以通过 config 的形式，读者可以参照 config 机制的相关章节。

另外一种方法就如在第一章的例子中那样，在 sequence 中把 sequencer 的 objection 给 raise 起来，当 sequence 完成后，再 drop 此 objection。这种方法相对上一种方法的好处就是不必设置要发送的包的数量。不过，这种方法的限制是，此 sequence 必须要做为 sequencer 的某个 phase（比如 main_phase）的 default_sequence，这个通常是比较容易的，一般都使用这种方法。

其实要解决这个问题，有很多方法，只要你有想像力，那么完全可以做到的。等到学习完了 callback 机制之后，你会切实体会到这一点的。

3.3. 用 domain 来划分不同的家庭

上面说了很多关于 objection 和 phase 的东西，并且得出了一些结论。要注意的是，有些结论只适用于同一个 domain 的情况。如果是不同的 domain 中，这些结论是不成立的。

什么是 domain？通俗点来说，UVM 中的 domain 就是家庭单位。在同一 domain 中，大家都听从同一个家长（phase）的安排，所有的 component 在同一时刻进入某一 phase（如 main_phase），又在同一时刻退出 main_phase（即使某个 component 已经执行完了 main_phase，它也要等在那里，等待其它 component 完成 main_phase），但是在不同的 domain 中，家长（phase）是不一样的，一个 domain 中的 component 不需要听从另外一个 domain 中家 长(phase)的安排,所以两个 domain 中的 component 何时进入某一 task phase 并不需要一致。

3.3.1. domain 的例子

先来看一个例子。假设我们的 DUT 分成两个相对独立的部分，这两个独立的部分可以分别复位，分别配置，分别启动，那么如果没有 domain 的概念，那么这两块独立的部分必须都同时在 reset_phase 复位，同时在 configure_phase 配置，同时进入 main_phase 开始正常工作。这种协同性当然是没有问题的，但是没有体现出独立性。图 3-3 中画出了这两个部分的 driver 位于同一 domain 的情况。

要体现出独立性，那么两个部分的 reset_phase，configure_phae，main_phase 等就不应该同步。这里就要用到 domain 的概念。图 3-4 中列出了两个 driver 位于不同 domain 的情况。

domain 把两块时钟域隔开，之后两个时钟域内的各个动态运行(run_time)的 phase 就可以不必同步。注意，这里 domain 只能把 run_time 的 phase 给隔离开来，对于其它的 phase，其实还是同步的，即两个 domain 的 run_phase 依然是同步的，其它的 function phase 也是同步的。

3.3.2. 多 domain 与单 domain 的区别

多 domain 与单 domain 的区别主要体现在 phase 和 objection 上。前面我们得到了结论，所有的 component 的 main_phase 都是同一个 phase，但是这个结论只适应于单 domain。如果两个 component 分别属于不同的 domain，那么这两个 component 的 main_phase（及其它 run_time phase）就不是同一个 phase。这个事实其实从上节的图中也可以看的出来。

由这个事实带来的一个问题就是在 objection 的控制上。我们在上面说，要在某一个 run_time 的 phase（如 main_phase）中执行语句，让仿真时间前移，那么至少要在一个 component 的相应的 phase 中 raise_objection（如至少要在 driver 的 main_phase 中 raise_objection）。在不同的 domain 中，各自的 run_time phase 各不相同（domain A 的 main_phase 与 domain B 的 main_phase 不相同），因此要在各自的 domain 中至少要找一个 component 来 raise_objection。使用的时候要注意这一点。

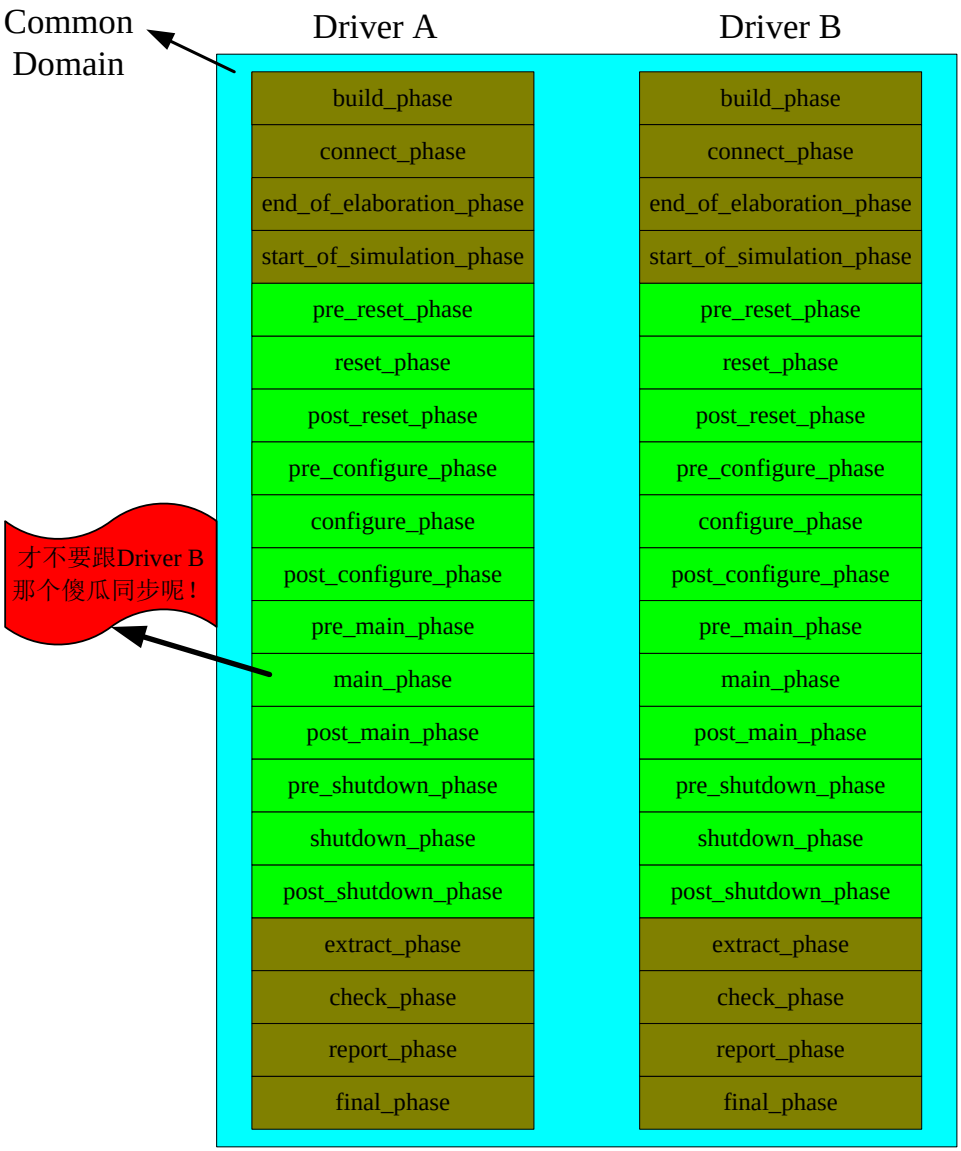


图 3-3 两个 driver 位于同一 domain

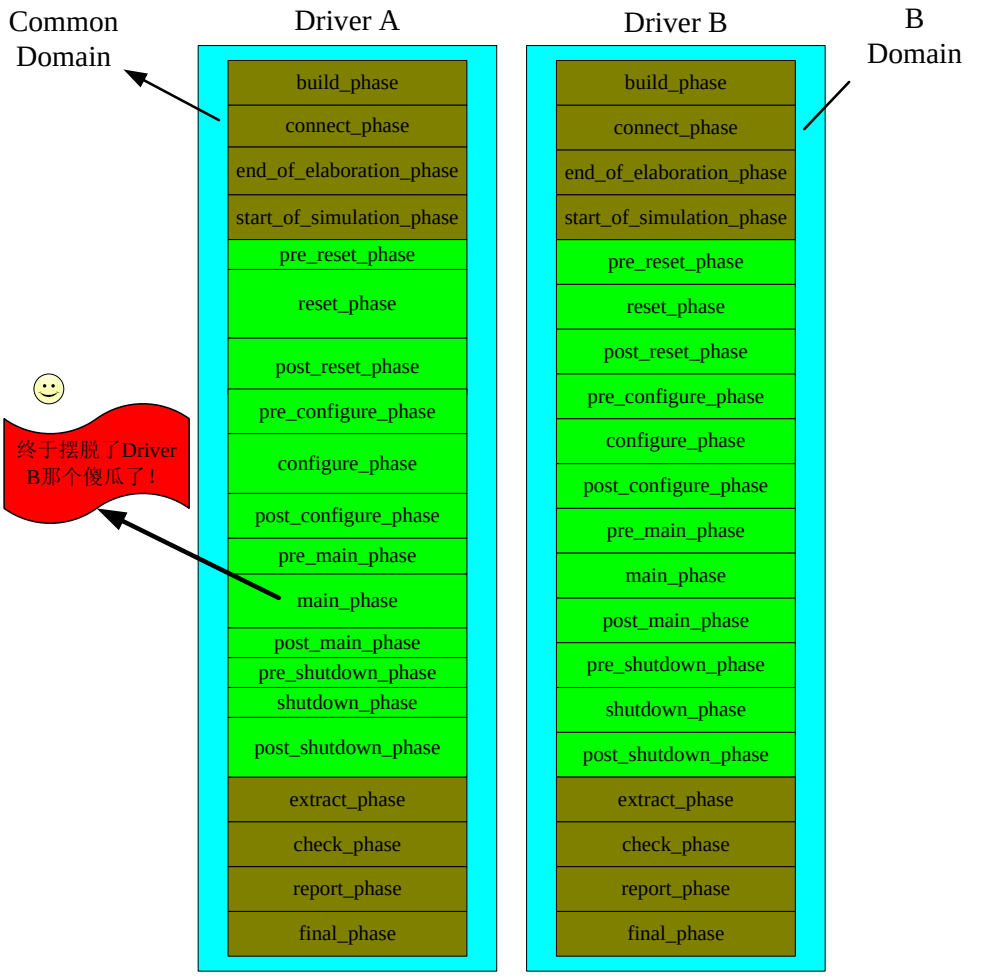


图 3-4 两个 driver 位于不同的 domain

4.transaction 及 field_automation

在第一章的例子中我们已经提到过了 transaction。现在我们回顾一下：transaction 是由我们自己定义的一个类，这个类是从 uvm_sequence_item 派生来的，在图 2-2 中已经看到，uvm_sequence_item 派生自 uvm_transaction，而 uvm_transaction 派生自 uvm_object。这里需要注意的一点是，虽然我们称呼其为 transaction，但是它并不是直接派生自 uvm_transaction。那么为什么这里不把其称呼为 sequence_item 而要称呼为一个 transaction 呢？这是因为在 TLM 定义中，它被称为 transaction。TLM 是 Transaction Level Modeling 的缩写，所谓的 transaction level 是相对 DUT 中各个 module 之间 pin 级别的通信来说的。通俗点说，DUT 中各个 module 之间通信是以 bit 为单位的，而 transaction level 则是以包为单位的。

每个 transaction 包含了我们关心的一些数据。如对于一个 my_transaction，里面包含了一个 mac 帧，这个帧里面里面包含 premble，sfd，目的地址，源地址，帧类型，帧的数据内容及 fcs 校验和等。sequence 产生出 transaction，通过 sequencer 把此 transaction 转交给 driver，driver 根据此 transaction 中的信息驱动接口信号。monitor 监测接口数据，并把数据封装成 transaction 的形式之后传递给 reference model 或者 scoreboard，由它们进行相应的处理。transaction 就是这样在整个验证平台中流动。可以这么说，transaction 是整个验证平台中流动的信息单元，是穿梭在验证平台中的子弹。

图 4-1 列出了 transaction 在一个验证平台中的流动，其中的红点表示 transaction，蓝线表示流动的路径。

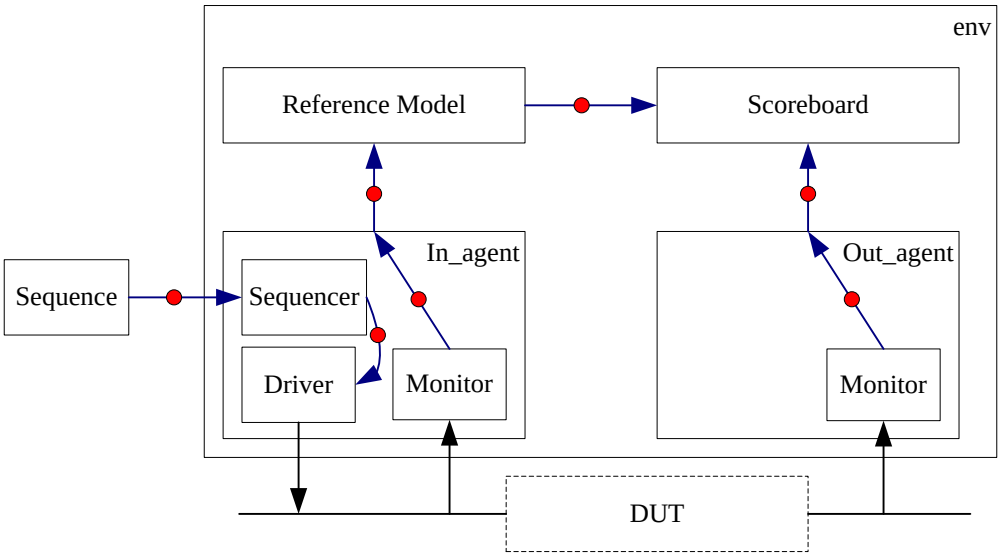


图 4-1 穿梭的 transaction

4.1. field_automation 机制

4.1.1. 为什么要使用 field_automation 机制

通常我们会如下定义一个 transacction：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] dmac;
    rand bit[47:0] smac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;
endclass
```

这里要注意的是每个成员变量前都有rand修饰符。transaction是流动在验证平台中的信息单元，reference model的行为与其接收到的信息单元是有直接关系的，相对

应的就是DUT接收到了不同的激励。因此，能够随机的产生信息单元可以最大程度上验证DUT的功能。在定义transaction时，在成员变量前加上rand修饰符，这样在调用此类的randomize函数[1]时，相应的字段就可以随机的取得一个值。

对于这样的一个 transaction，我们经常会用到许多常用的操作，如把所有的字段（成员变量）打印一下，这个就需要自己写 print 函数：

```
function void print();
    $display("dmac=%0h", dmac);
    $display("smac=%0h", smac);
    $display("eth_type=%0h", eth_type);
    for(int i = 0; I < pload.size; i++) begin
        $display("pload[%0d]=%0h", i, pload[i]);
    end
    $display("crc=%0h", crc);
endfunction
```

如在 scoreboard 中要比较 reference model 和 monitor 收集过来的两个 transaction 是否一致，这个就需要自己写 compare 函数:

```
function bit compare(mac_transaction tr);
    if(this.dmac != tr.dmac)
        return 0;
    else if(this.smac != tr.dmac)
        return 0;
    else if(this.eth_type != tr.eth_type)
        return 0;
    else if(this.crc != tr.crc)
        return 0;
    else if(this.pload.size != tr.pload.size)
        return 0;
    else begin
        for(int i = 0; i < this.pload.size; i++) begin
            if(this.pload[i] != tr.pload[i])
                return 0;
        end
    end
    return 1;
endfunction
```

可以看出，这样写起来将会是相当费时间的。而且对于这些常用的操作，如果 transaction 的定义换一下，那么这些所有的函数或任务就都需要重新写，这种代价是相当大的。仔细思考一下，这些事情的通性就是简单，重复，琐碎。那么有没有方法简化一下呢？答案是有的，这就是 field_automation 机制。

[1] 这是 systemverilog 的功能，而不是 UVM 的功能。

4.1.2. field_automation 机制的使用

UVM 中使用 field_automation 机制来完成这些事情。如对于上面的 transaction，使用 field_automation 机制之后，就可以如下定义：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] dmac;
    rand bit[47:0] smac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;

    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

加入了 uvm_object_utils_begin(mac_transaction)这样一句，这句话跟我们前面说 factory 机制时提到的 uvm_object_utils 非常相似。事实上，这其实就是 factory 机制的实现。field_automation 机制并不能单独使用宏来实现，我们不能如下定义一个 transaction：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] dmac;
    rand bit[47:0] smac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;

        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
endclass
```

这样系统是会报错的。所以 field_automation 机制的实现必须要依赖于 factory 机制的宏来实现。这个虽然看起来不合情理，其实用习惯了之后就会感觉是理所当然的。因为一个 UVM 中推荐一个 object 一定要用 factory 来实现，所以 uvm_object_utils 几乎就是必须的，所以在 uvm_field_*之间写上 uvm_object_utils_begin 就是顺理成章的。

uvm_field_*系列宏共有如下几种：

```
`define uvm_field_int(ARG,FLAG)
`define uvm_field_real(ARG,FLAG)
`define uvm_field_enum(T,ARG,FLAG)
`define uvm_field_object(ARG,FLAG)
`define uvm_field_event(ARG,FLAG)
`define uvm_field_string(ARG,FLAG)
`define uvm_field_array_enum(ARG,FLAG)
`define uvm_field_array_int(ARG,FLAG)
`define uvm_field_sarray_int(ARG,FLAG)
`define uvm_field_sarray_enum(ARG,FLAG)
`define uvm_field_array_object(ARG,FLAG)
`define uvm_field_sarray_object(ARG,FLAG)
`define uvm_field_array_string(ARG,FLAG)
`define uvm_field_sarray_string(ARG,FLAG)
`define uvm_field_queue_enum(ARG,FLAG)
`define uvm_field_queue_int(ARG,FLAG)
`define uvm_field_queue_object(ARG,FLAG)
`define uvm_field_queue_string(ARG,FLAG)
`define uvm_field_aa_int_string(ARG, FLAG)
`define uvm_field_aa_string_string(ARG, FLAG)
`define uvm_field_aa_object_string(ARG, FLAG)
`define uvm_field_aa_int_int(ARG, FLAG)
`define uvm_field_aa_int_int(ARG, FLAG)
`define uvm_field_aa_int_int_unsigned(ARG, FLAG)
`define uvm_field_aa_int_integer(ARG, FLAG)
`define uvm_field_aa_int_integer_unsigned(ARG, FLAG)
`define uvm_field_aa_int_byte(ARG, FLAG)
`define uvm_field_aa_int_byte_unsigned(ARG, FLAG)
`define uvm_field_aa_int_shortint(ARG, FLAG)
`define uvm_field_aa_int_shortint_unsigned(ARG, FLAG)
`define uvm_field_aa_int_longint(ARG, FLAG)
`define uvm_field_aa_int_longint_unsigned(ARG, FLAG)
`define uvm_field_aa_string_int(ARG, FLAG)
`define uvm_field_aa_object_int(ARG, FLAG)
```

uvm_field_array_A 表示的是动态数组，A 表示动态数组中存放的内容的类型，而 uvm_field_sarray_A 表示的是静态数组，A 表示此静态数组中存放的内容的类型。uvm_field_queue_A 表示的是队列，A 表示此队列中存放的内容的类型，uvm_field_aa_A_B 表示的是联合数组，其中的 B 表示联合数组的索引类型，A 表示联合数组中存放的内容的类型。

关于宏的参数，其中的 ARG 表示在 transaction 中定义的变量的名字，FLAG 将会在 4.1.4 中介绍。除了这两个最大众化的参数外，在 uvm_field_enum 的参数中出现了 T，这个 T 表示自定义的 enum 类型的名字。

4.1.3. field_automation 机制都做了哪些事情

经过上节的定义后，我们有很强烈的冲动想知道 uvm_field_*系列宏都做了什么事情。

uvm_field_*系列宏做的最简单的就是我们 4.1.1 节中所说的 print 函数和 compare 函数。现在，我们可以自己不用写任何代码就可以调用 print 函数了：

```
mac_transaction tr = new;
mac_transaction cp;
tr.print();
tr.compare(cp);
```

除了 compare 与 print 外，还提供了 pack 的功能。什么是 pack 功能呢？

假如没有pack，则在driver中要把一个mac_transaction给发送出去，要每次发送 8 个bit（假设数据线是有 8 位），需要这样写（其实这种写法有问题，编译是通不过的，读者能看出是什么问题来吗 [1]？这里只是做为示例，所以暂且这样写）：

```
for (i=0; i < 6; i++) begin
    drive_one_byte(mac_tran.dmac[(i+1)*8: i*8])//drive dmac
end
for (i=0; i < 6; i++) begin
    drive_one_byte(mac_tran.smac[(i+1)*8: i*8])//drive smac
end
…
```

可以看到，这是相当复杂的。当 mac_transaction 增加一个字段时，就需要全部重写 driver。有了 pack 之后，我们可以这样写：

```
byte q[];
int n;
n = mac_tran.pack_bytes(q)/8;
for( int i = 0; i < n; i++) begin
    drive_one_byte(q[i]);
end
```

就这么几句话就结束了，是不是感觉很神奇呢？那么我们来看看 pack_bytes 主要是做了什么事情。如图 1-6 所示，pack_bytes 的功能其实就是把 mac 的这些字段，按照 byte 的形式，分别装进一个数组 q 中。如 dmac[47:40]放入 q 的第一个单元中，dmac[39:32]放入 q 的第二个单元中，以此类推。这样，当在 driver 中要 drive 数据的时候，可以不必考虑具体的 transaction 的定义，只 drive 一个经过 pack 后的数组中

[1] 这里的问题就在于 systemverilog 在选择压缩数组的某几位时，其选择符不能为变量。即假设 bit[31:0] a;如果我们写上 a[msb: lsb]，要想编译通过，msb 和 lsb 不能为变量或者变量表达式。

的数据就可以了。pack_bytes 函数返回的是 q 中所有 bit 的数量之和。由于 q 的宽度是 8（一个 byte），所以 q 的大小就相当于此返回值除以 8，即上例中 n 的值。至于为什么 pack_bytes 不直接返回 bytes 的数量，而非要返回 bit 的数量，这个就是 UVM 设计的问题了。

与 pack_bytes 相对应的是 unpack_bytes。其实理解了 pack_bytes 之后就非常容易理解 unpack_bytes。这个 unpack_bytes 的过程与 pack_bytes 完全相反，如图 1-6 所示，q 中前六个单元的内容被直接放入了 dmac 字段中，接下来的放入了 smac 字段中，以此类推，最后的四个单元被放入了 crc 字段中。

说到现在，读者可能发现了，pack 和 unpack 对于字段的位置非常敏感。假如把 dmac 和 smac 的位置换一下：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
    rand bit[47:0] dmac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;

    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

那么对于 unpack_bytes 来说，q 中的前六个单元的内容将会放入 smac 中，而不是 dmac 中。这种顺序是由 uvm_field_*系列宏书写的顺序决定的，所以在定义 transaction 时，一定要注意宏的顺序。

uvm_field_*宏还提供了一些其它功能，如 copy 等，这里不一一的介绍。

4.1.4. 如何排除某些字段

假设我们的 mac_transaction 中多了一个 crc_err 的字段，这个字段用于控制向激励中施加 crc 错误。transaction 之所以强大，就是因为在其中可以加入很多标志位，这些标志位不会被 driver 发送给 DUT，但却可以在验证平台之间流动，验证平台根据这些标志位来做判断可以省去很多时间。例如，某个包是 crc 错误的，那么在 reference model 中应该把这个包丢弃。reference model 如何知道这个包是 crc 错误呢？它可以检查所有的 smac，dmac，eth_type，pload，并计算出 crc 字段，最终判断是

crc 错误。另外一种方法是看一下 crc_err 标志位，如果标志位为 1，那么就直接判断是 crc 错误。当然了，这种方法的前提是前面的处理模块已经把前面的那些判断工作做完了，并根据计算结果给 crc_err 置位。这看起来只是把应该 reference model 做的事情前移了而已，似乎并没有省多少时间。如果在整个验证平台中有 10 个地方要检查 crc_err，那么只要在一个地方检查一次，把 crc_err 置位，后面的就都可以使用。即检查一次，使用多次。如果只使用一次，那么 crc_err 标志位的优势是体现不出来的。

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
    rand bit[47:0] dmac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;
    rand bit       crc_err;

    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
        `uvm_field_int(crc_err, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

那么，对于多出来的这个字段，是不是也应该用 uvm_field_int 宏来注册呢？如果不使用宏注册的话，那么当 print 的时候，我们就看不到其值了，但是如果用了宏，结果就是这个根本就不需要在 pack 和 unpack 的中出现的字段出现了。这会带来极大的问题。

UVM 考虑到了这一点，它采用后面的控制域中加入 UVM_NOPACK 的形式来实现：

```
`uvm_field_int(crc_err, UVM_ALL_ON | UVM_NOPACK)
```

使用上述语句后，那么当 pack 和 unpack 的时候，UVM 就不会考虑这个字段了。这种写法比较奇怪，是用了一个或（|）来实现的。想要知道为什么这么写吗？可以参看本书第 14 章 field_automation 机制源代码分析。

除了 UVM_NOPACK 之后，还有 UVM_NOCOMPARE，UVM_NOPRINT 等选项也可以使用。具体的可以参考 UVM 的文档。

4.2. transaction 使用时的一些技巧

transaction 在使用时有一些常用的技巧。本节将会给出几个常用的，读者可以在搭建验证平台时尝试。

4.2.1. “尽量做到”

继续使用上节的例子，我们的 mac_transaction 中，现在新扩充了 sfd_err。在默认的情况下，我们发送的 transaction 是正常的 transaction，所以 crc_err 与 sfd_err 的值都默认为 0，因此可以使用 constraint 作如下定义：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
    rand bit[47:0] dmac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;
    rand bit       crc_err;
    rand bit       sfd_err

    constraint default_cons{
        crc_err == 0;
        sfd_err == 0;
    }
    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
        `uvm_field_int(crc_err, UVM_ALL_ON | UVM_NOPACK)
        `uvm_field_int(sfd_err, UVM_ALL_ON | UVM_NOPACK)
    `uvm_object_utils_end
endclass
```

这种定义看来是不错的，但是一个潜在的问题就是，crc_err 与 sfd_err 的值永远的被限制为 0，无论怎么随机化，这两个值都不可能为 1。这与我们的初衷是相违背的，因为我们期望默认情况下为 0，在测试 error case 的情况下，其值可以为 1，也可以为 0。

我们期望在定义 transaction 时可以像上面那样定义，但是在使用时，如果不指定新的约束，表明我们使用默认的约束(default_cons)，否则就使用新的约束。即尽

量使用默认的约束，如果默认的约束与新的约束有冲突，那么就使用新的约束。换言之，就是“尽量做到”这一功能。在正常的 case 中：

```
mac_transaction tr;
tr = new();
assert(tr.randomize());
```

这表示使用默认的约束。在需要注入 error 的 case 中：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {crc_err dist {0 := 2, 1 := 1}; sfd_err dist {0 := 2, 1 := 1}});
```

这表示使用新的约束。但是很可惜，这样的方式在 systemverilog 中是行不通的。systemverilog 不支持这种语法，它会提示两个约束是冲突的。

考虑到 systemverilog 中有 dist 的语法，似乎可以以如下的方式定义 constraint：

```
    constraint default_cons{
        crc_err dist{0 := 999_999_999, 1 := 1};
        sfd_err dist{0 := 999_999_999, 1 := 1};
    }
```

上述语句的意思是，在随机化的时候，crc_err 和 sfd_err 只有 1/1_000_000_000 的可能性会取值为 1，其余的都为 0。这看似是让人非常满意的，但是其中依然潜伏着问题：如果我们测试某个 case 是正常的 case，里面不能有 error 产生，换句话说，crc_err 与 sfd_err 的值要一定为 0。上面的 constraint 明显是不能满足这种要求，因为虽然只有 1/1_000_000_000 的可能性，但是这种可能性依然存在。在跑特别长的 case 的时候，如我们发送了 1_000_000_000 个包，那么这其中有非常非常大的可能性会产生一个 crc_err 或 sfd_err 值为 1 的包。

要在 systemverilog 中解决这个问题，是一个相当困难的一件事情。这里提供两种解决方式。

第一种方式是在定义 transaction 时，constraint 依然使用如下的方式定义：

```
    constraint default_cons{
        crc_err == 0;
        sfd_err == 0;
    }
```

在正常 case 时，可以使用如下的方式随机化：

```
mac_transaction tr;
tr = new();
assert(tr.randomize());
```

在需要注入 error 的 case 时，可以使用如下的方式随机化：

```
mac_transaction tr;
tr::constraint_mode(0);//tr.default_cons::constraint_mode(0) also works
```

```
tr = new();
assert(tr.randomize() with {crc_err dist {0 := 2, 1 := 1}; sfd_err dist {0 := 2, 1 := 1}});
```

上面使用 constraint_mode(0)关闭了 default_cons。之后在随机化时可以指定新的约束条件。关闭约束有两种方法，上面使用的是关闭 tr 中所有的约束，另外一种是 tr.default_cons::constraint_mode(0)，这样会关闭某个特定的约束。在上面的例子中，只有一个约束，所以两者的效果是一样的。上面的法子看似简单，其实还有复杂的一面。使用 constraint_mode(0)关闭约束后，crc_err 与 sfd_err 都失去了约束，假如我们只想造 crc_err 的 case，而 sfd 要求正常，那么就必须如下写：

```
assert(tr.randomize() with {crc_err dist {0 := 2, 1 := 1}; sfd_err == 0;);
```

如果 default_cons 中额外定义了 100 条约束，即：

```
    constraint default_cons{
        crc_err == 0;
        sfd_err == 0;
        cons_1 == 0;
        …
        cons_100 == 0;
    }
```

在随机化时就需要如下方式书写：

```
assert(tr.randomize() with {crc_err dist {0 := 2, 1 := 1}; sfd_err == 0; cons_1 == 0; … cons_1
00 == 0);
```

这样写出来的后果将会是非常的冗长。

解决上面问题的办法之一就是把所有的约束分开书写：

```
    constraint crc_cons{
        crc_err == 0;
    }
    constraint sfd_cons{
        sfd_err == 0;
    }
    constraint cons_1{
        cons_1 == 0;
    }
        …
    constraint cons_100{
        cons_100 == 0;
    }
```

之后在随机化时使用如下方式：

```
mac_transaction tr;
tr.crc_cons::constraint_mode(0);
tr = new();
assert(tr.randomize() with {crc_err dist {0 := 2, 1 := 1}; });
```

在上面的这种解决方案中，需要调用 constraint_mode 语句。很多人不习惯这种调用方式，于是相应的有了第二种解决方案，在定义 transaction 时，加入一个 keep_default，当其值为 1 时，crc_err 和 sfd_err 都为 0。

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
    rand bit[47:0] dmac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;
    rand bit       crc_err;
    rand bit       sfd_err
    rand bit       keep_default;


    constraint default_cons{
        solve keep_default before crc_err;
        solve keep_default before sfd_err;
        keep_default == 1 -> crc_err == 0;
        keep_default == 1 -> sfd_err == 0;
        keep_default == 0 -> crc_err dist {0 := 2, 1 := 1};
        keep_default == 0 -> sfd_err dist {0 := 2, 1 := 1};
    }
    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
        `uvm_field_int(crc_err, UVM_ALL_ON | UVM_NOPACK)
        `uvm_field_int(sfd_err, UVM_ALL_ON | UVM_NOPACK)
    `uvm_object_utils_end
endclass
```

相应的，在正常 case 中可以如下进行随机化：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {keep_default == 1;});
```

在需要注入 error 的 case 中，以如下的方式随机化：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {keep_default == 0;});
```

当然了，这种情况产生的 case 中，crc_err 和 sfd_err 是混杂的。要想单独的只产生一种 error，可以用较复杂的方式实现，读者可以尝试一下。

以上两种方案各有利弊，读者可以按照自己的习惯使用。

4.2.2. 在 uvm_field_*宏前后使用 if 语句

在以太网中，有一种帧是 VLAN 帧，这种帧是在普通以太网帧基础上扩展而来的。而且并不是所有的以太网帧都是 VLAN 帧，如果一个帧是 VLAN 帧，那么其中就会有 vlan_id 等字段（具体可以详见以太网的相关协议），否则不会有这些字段。类似 vlan_id 等字段是属于帧结构的一部分，我们习惯了使用 uvm_field 系列宏来进行 pack 和 unpack，因为这个字段可能有，也可能没有，那么很直观的想法是使用动态数组的形式来实现：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
    rand bit[47:0] dmac;
    rand bit[31:0]   vlan[];
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;

    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_array_int(vlan, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

在随机化普通以太网帧时，可以使用如下的方式：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {vlan.size() == 0;});
```

协议中规定 vlan 的字段固定为 4 个 byte，所以在随机化 VLAN 帧时，可以使用如下的方式：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {vlan.size() == 1;});
```

协议中规定 vlan 的 4 个 byte 各自有其不同的含义，这 4 个 byte 分别代表 4 个不同的字段。如果使用上面的方式，问题虽然解决了，但是这 4 个字段的含义不太明确。

一个可行的解决方案是：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] smac;
```

```
    rand bit[47:0] dmac;
    rand bit[15:0] vlan_info1;
    rand bit[2:0]  vlan_info2;
    rand bit       vlan_info3;
    rand bit[11:0] vlan_info4;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;

    rand bit       is_vlan;

    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(dmac, UVM_ALL_ON)
        if(is_vlan) begin
            `uvm_field_int(vlan_info1, UVM_ALL_ON)
            `uvm_field_int(vlan_info2, UVM_ALL_ON)
            `uvm_field_int(vlan_info3, UVM_ALL_ON)
            `uvm_field_int(vlan_info4, UVM_ALL_ON)
        end
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

在随机化普通以太网帧时，可以使用如下的方式：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {is_vlan == 0;});
```

在随机化 VLAN 帧时，可以使用如下的方式：

```
mac_transaction tr;
tr = new();
assert(tr.randomize() with {is_vlan == 1;});
```

使用这种方式的 VLAN 帧，在 print 时，4 个字段的信息将会非常明显，在调用 compare 函数时，如果两个 tran 不一样，将会更加明确的指明是哪个字段不一样。

5.sequence 机制

如果说 transaction 是子弹的话，那么 sequence 无疑就是弹夹了。在整个的 UVM 验证平台中，sequence 负责 transaction 的产生，并通过 sequencer 发送给 driver，driver 则根据 transaction 里存储的信息产生激励。要产生不同的 transaction，那么就要在 sequence 中下功夫。验证中要对 DUT 施加以不同的激励，也就是施加不同的 case，在 UVM 中，不同的 case 的差异主要就是体现在 sequence 的不同上。通过控制 sequence，可以完美的产生各种不同的激励。

sequence 的另外一个重要的作用就是控制整个验证平台的关闭。在第三章的时候已经说过了控制验证平台关闭的方法。读完本章，想必读者会对验证平台的关闭有了更深的了解。

本章第一节介绍 sequence 的概念，第二节说明使用 uvm_do 宏来构建 sequence，第三节介绍 virtual sequence 的使用。

5.1. UVM 中的 sequence 机制

sequence 是什么？经过第一章之后，相信读者已经大体上有了一个印象，本节将会详细的介绍一下 sequence 机制。

5.1.1. 激励信息的产生与驱动的分离

在第三章中，UVM 把整个验证平台的运行过程分成了不同的 phase，其目的就是为了保证在特定时间做特定事情，形成规范化，制度化，从而提高效率。这种从乱中理出思路，通过分类，来保证易用性的理念是 UVM 一直推崇的。UVM 中，sequence 的设计也几乎遵循相同的设计哲学，它把数据流（激励信息）的产生从其它部分完全独立出来，从而提高了其它部分的可重用性，增加了数据流产生的灵活性。

使用 1.3.2 节中 DUT 的定义，在最原始的验证平台中，使用如下的方式产生激励：

```
module top_tb;
reg clk;
reg [7:0] rx_data;
reg rx_dv;
wire [7:0] tx_data;
wire tx_dv;
dut   my_dut(.clk(clk),
.rxd(rx_data),
.rx_dv(rx_dv),
.txd(tx_data),
.tx_en(tx_dv)
)
initial begin
    clk = 0;
    forever begin
        #10; clk = ~clk;
    end
end

initial begin
    #100;
    @posedge clk;
    rx_data <= 'HF6;
    rx_dv <= 1;
    @posedge clk;
    rx_dv <= 0;
    #100;
    $finish();
end
endmodule
```

激励的产生与驱动都是在一个 initial 模块中产生的。这种方法易用性差，要产生复杂的激励比较难。读完了前四章后，大家对 uvm_component 的概念比较熟悉了，并且也了解了一些常用的 uvm_component。driver 是用于驱动接口以给 DUT 施加激励的。所以一种自然的想法就是在 driver 的 main_phase 中来写各种各样的代码，从

而可以产生各种各样的激励：

```
task mac_driver::main_phase(uvm_phase phase);
    mac_transaction tr;
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < 10; i++) begin
        tr = new;
        assert(tr.randomize);
        drive_one_pkt(tr);
    end
    phase.drop_objection(this);
endtask
```

这种情况下，其实就相当于 driver 的 main_phase 扮演了上面 tob_tb 中的 initial 块的作用。不过比 initial 块好的地方就在于 driver 中实现了 transaction 级别的激励产生。这种想法看起来非常美，似乎也够用了，但是仔细想一下，当我们要对 DUT 施加不同的激励时，那应该怎么办呢？上面的代码中是施加了正确的包，而在下一次测试中要加入 crc 错误的包，那么可以这么写：

```
task mac_driver::main_phase(uvm_phase phase);
    mac_transaction tr;
    bit send_crc_err = 0;
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < 10; i++) begin
        tr = new;
        if(send_crc_err)
            assert(tr.randomize with {tr.crc_err == 1;});
        else
            assert(tr.randomize);
        drive_one_pkt(tr);
    end
    phase.drop_objection(this);
endtask
```

这就相当于是把整个 main_phase 给重新写了一遍，并且要发送 CRC 错误包时，还要通过别的方法把 send_crc_err 设置为 1。如果现在有了新的需求，要再测一个超长包呢？那还要再改写 main_phase，也就是说，要多测一种情况，就要多改写一次 main_phase。如果经常的改写某个任务或者函数，那么就很容易把以前对的地方给改错。所以说，这种方法是不可取的，因为它的可扩展性太差，经常会带来错误。

仔细观察 main_phase，其实只有从 tr=new 语句一直到 drive_one_pkt 之间的语句在变。有没有什么方法把这些语句从 main_phase 中独立出来呢？最好的方法就是在不同的 case 中决定这几行语句的内容。这种想法中已经包含了激励的产生与驱动的分离这个朴素的观点。drive_one_pkt 是驱动，这是 driver 应该做的事情，但是像产生什么样的包，如何产生等这些事情应该从 driver 中独立出去。

5.1.2. 数据流的独立

那么如何实现上面的想法呢？最原始的想法应该就是使用一个函数来实现。

```
function gen_pkt(ref mac_transaction tr);
    tr = new;
    assert(tr.randomize);
endfunction
task mac_driver::main_phase(uvm_phase phase);
    mac_transaction tr;
    bit send_crc_err = 0;
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < 10; i++) begin
        gen_pkt(tr);
        drive_one_pkt(tr);
    end
    phase.drop_objection(this);
endtask
```

如上所示，可以定义一个产生正常包的 gen_pkt 函数，但是如何产生一个 crc 错误包的函数呢？难道像下面这样定义吗？

```
function gen_pkt(ref mac_transaction tr);
    tr = new;
    assert(tr.randomize with {crc_err == 1;});
endfunction
```

这样带来的一个最大的问题就是 gen_pkt 函数的重复定义，很显然，这样是不允许的。为了避免重复定义，我们希望能够定义的函数的名字是不一样的，但是在 driver 的 main_phase 中又能执行这种具有不同名字的函数。

这个问题是一个相当难的问题。用单纯的 systemverilog 提供的一些接口是根本无法实现的。UVM 为了解决这个问题，引入了 sequence 机制，在解决的过程中还使用了 factory 机制，config 机制。使用 sequence 机制之后，在不同的 case 中，把不同的 sequence 设置成 sequencer 的 main_phase 的 default_sequence。当 sequencer 执行到 main_phase 时，发现有 default_sequence，那么它就会把这个 sequence 启动起来。图 5-1 中示出了 default sequence 的设置与启动。在 env.agt.sqr 中，每当进入到一个新的 run_time phase 时，就会查看是否为此 phase 设置了 default_sequence。图中的例子为 main_phase 设置了，但是没有为其它 phase，所以只会在 main_phase 启动 my_sequence。

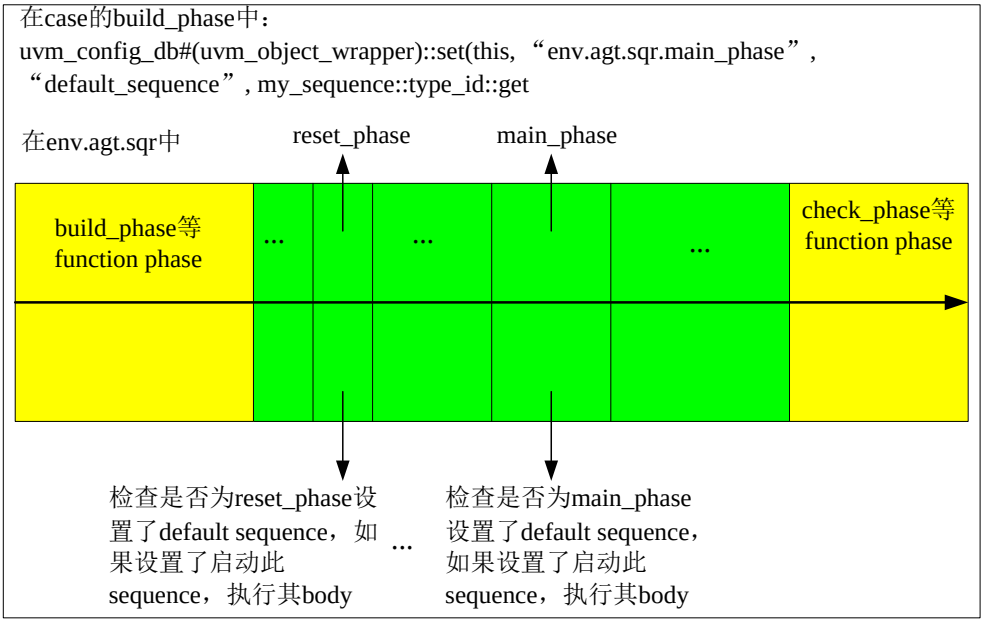


图 5-1 default sequence 的设置与启动

仔细的想一下上面的过程，sequencer 把 sequence 启动起来的过程就相当于是之前的 gen_pkt，只是执行的位置从 driver 变到了 sequencer。sequencer 把执行后产生的 transaction 交给 driver。这个其实跟在 driver 里面执行没有本质的区别。

5.1.3. sequence 的启动与执行

上一小节提到了把 sequence 启动起来这样一个概念，那么 sequence 是如何启动的呢？

假设有如下的 sequence 定义：

```
class my_sequence extends uvm_sequence #(mac_transaction);
    mac_transaction m_trans;

    extern function new(string name = "my_sequence");
    virtual task body();
        repeat (10) begin
            `uvm_do(m_trans)
        end
        #100;
    endtask
```

```
    `uvm_object_utils(my_sequence)
endclass

function    my_sequence::new(string  name=  "my_sequence");
    super.new(name);
endfunction  //  new
```

那么启动一个 sequence 就应该这么做：

```
my_sequence  my_seq;
my_seq  =  my_sequence::type_id::create("my_seq");
my_seq.start(sequencer);
```

可见，要启动一个 sequence 非常的简单，第一步就是把这个 sequence 给实例化，第二步就是调用 sequence 的 start 任务，调用时要传入一个 sequencer 参数。

当这个 sequence 启动起来的时候，这个 sequence 的 body 就会自动执行，我们可以想像，在 my_seq.start()的实现中肯定会有这么一句：

```
my_seq.body();
```

事实也确是如此。读者如果对此有兴趣，可以看本书第 15 章 sequence 机制源代码分析。

现在，我们可以推测 sequencer 的 main_phase 中启动 sequence 的过程了：它先把传递过来的 default_sequence 给实例化，然后调用此 sequence 的 start 任务，给 start 任务传递的参数是 this，因为这是一个 sequencer 的 main_phase 在启动 sequence，this 就是代表 sequencer 了。

5.1.4. 通过 sequence 来控制验证平台的关闭

伴随着 sequence 机制的使用，数据流完全的独立出来了，但是一个新的问题产生了：5.1.1 节是在 driver 中来 raise_objection 和 drop_objection。现在，数据的产生从 driver 中完全的独立出去了，那么做为 driver 来说，它只是负责从 sequencer 中索要 transaction，如果索要到了那么就把这个 transaction 驱动到接口上，如果没有，那么就等在那里。也就是说，driver 失去了控制验证平台退出的能力。

5.1.1 节中，driver 之所以能够控制验证平台的关闭，是因为 driver 同时有激励产生的功能。现在激励产生的功能已经转移到了 sequence 中，那么相应的，控制验证平台退出的功能也应该转移到 sequence 中，即在 sequence 中 raise_objection 和 drop_objection。

但是在 sequence 中进行 raise_objection 的一个问题是，raise_objection 是属于

phase 的一个函数，即我们只能以如下的方式调用此函数：

```
phase.raise_objection(this);
```

而 phase 是属于 component 的一个概念，是 component 专属的东西，而 sequence 的本质是一个 object，是没有 phase 的。那么怎么办？

这个问题其实非常简单。我们可以在 uvm_sequence 中加一个指向 phase 的指针，然后当 sequencer 在 main_phase 中启动 default_sequence 时，把 sequencer 的 main_phase 中的 phase 赋值给 sequence 中这个指针。这样在 sequence 中就可以进行 objection 操作了。

UVM 中就是这么做的。在 sequence 中，这个指向 phase 的指针的名字是 starting_phase。因此，我们可以在 sequence 中这么做：

```
task body();
    if(starting_phase != null)
        starting_phase.raise_objection(this);
    …
    if(starting_phase != null)
        starting_phase.drop_objection(this);
endtask
```

由于 starting_phase 只是一个指针，所以为了保险起见，要判断一下它是否为 null。

5.2. 写出强大的 sequence

上一节把 sequence 机制简单说明了一下。本节将着重于介绍 sequence 的应用，将会提供一些常用的方法及技巧。

5.2.1. 使用 uvm_do 系列宏

当一个 sequence 启动起来之后，UVM 会自动执行 sequence 的 body 任务，所以要产生各种各样的激励，就要写好 body 任务。

body 这个任务完成的事情几乎与 5.1.2 节中的 gen_pkt 函数相似，只是不同的是，gen_pkt 可以直接通过一个 ref 形式的参数把产生的 transaction 交给 driver，而 sequence 则需要使用通信的方式来传递，如下图所示：

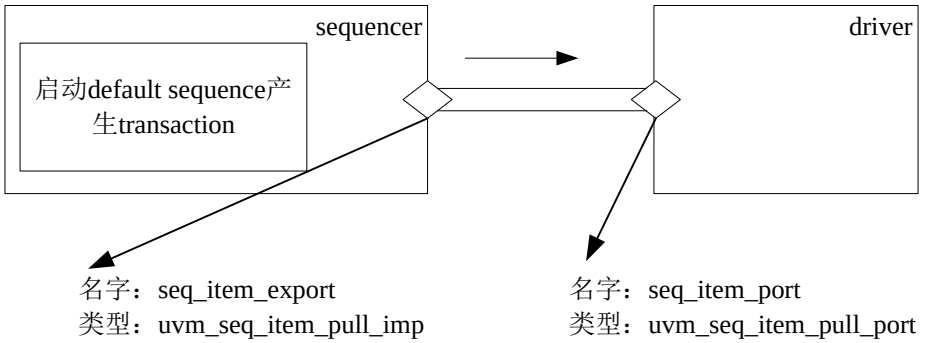


图 5-2 sequencer 与 driver 之间的通信

上图中出现了 uvm_seq_item_pull_imp 和 uvm_seq_item_pull_port，关于这两者的介绍，可以在第 7 章 UVM 的各种 port 中找到。

由于传递方式的改变，所以产生 transaction 的方式也就有了不同。一个 transaction 应该这样产生：

```
task body();
    mac_transaction tr;
    tr = mac_transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize());
    finish_item(tr);
endtask
```

要产生一个 transaction，要分成四步，第一步就是实例化，第二步就是调用 start_item，第三步就是执行 randomize 过程，第四步就是调用 finish_item。由于是牵扯到 sequencer 和 driver 的通信，因此这里就有一个主动和被动的过程。当 driver 中使用 seq_item_port.get_ next_item 主动请求一个 item（transaction）时，sequencer 才会要求 sequence 产生一个 item（transaction）；产生 完成后，sequence 就要等待 driver 把 item（transaction）取走，这需要 driver 显式的调用 seq_item_port.item_done()。当此函数被调用后，finish_item 才会返回，一个 transaction 的产生才真正的完成。

仔细观察这四步，其实只有第三步才会有一些差异，而另外的三步则永远是固定不变的。针对这种情况，UVM 推出了 uvm_do 系列宏来简化产生 transaction 的过程。如上，使用 uvm_do 宏就可以简写为：

```
task body();
    mac_transaction tr;
    `uvm_do(tr)
endtask
```

一个宏封装了产生 item 的四个步骤，相当于把所有的事情都做完了，极大的方便了使用。除了 uvm_do 宏之外，uvm_do 系列宏中另外一个经常使用的就是 uvm_do_with。它的使用方式如下：

```
task body();
    mac_transaction tr;
    `uvm_do_with(tr, {tr.crc_err == 1;})
endtask
```

使用 uvm_do_with 宏之后，那么产生 transaction 的第三步中的 assert(tr.randomize())会被替换为：

```
assert(tr.randomize() with {tr.crc_err == 1;});
```

这个跟我们自己写约束条件是一模一样的。如果想发多个包怎么办？很简单！

```
task body();
    mac_transaction tr;
    repeat(pkt_num) begin
        `uvm_do(tr);
    end
endtask
```

其中的 pkt_num 是要发送的包的数量。

5.2.2. 把 sequence 做为 uvm_do 宏的参数

假设一个产生特定 crc_err 的包的 sequence 如下：

```
class crc_seq extends uvm_sequence#(mac_transaction);
    task body();
        mac_transaction tr;
        `uvm_do_with(tr, {tr.crc_err == 1;
                          tr.da == 48'h980F;})
    endtask
    `uvm_object_utils(crc_seq)
endclass
```

另外一个 sequence 如下：

```
class long_seq extends uvm_sequence#(mac_transaction);
    task body();
        mac_transaction tr;
        `uvm_do_with(tr, {tr.pload.size() == 1500;
                          tr.sa == 48'hFFFF;
                          tr.da == 48'h4356D;})
    endtask
    `uvm_object_utils(long_seq)
endclass
```

现在，我们要写一个新的 sequence，它可以交替产生上面的两种包，那么，在新的 sequence 里面我们可以这样写：

```
class new_seq extends uvm_sequence#(mac_transaction);
    task body();
        mac_transaction tr;
        `uvm_do_with(tr, {tr.crc_err == 1;
                          tr.da == 48'h980F;})
        `uvm_do_with(tr, {tr.pload.size() == 1500;
                          tr.sa == 48'hFFFF;
                          tr.da == 48'h4356D;})
    endtask
    `uvm_object_utils(new_seq)
endclass
```

这样写起来似乎显得特别的麻烦。产生的两种不同的包中，第一个约束条件有两个，第二个约束条件有三个。但是假如约束条件有十个呢？如果我们有整个验证平台中有 30 个 case 都用到了这样的一个包，那就要在这 30 个 case 的 sequence 中加入 10 行这样的代码（假设一个约束条件写一行），这是一件相当恐怖的事情，而且特别容易出错。既然我们定义好了 crc_seq 和 long_seq，那么有没有简单的方法呢？答案是有的。在上一小节的例子中，我们把一个 mac_transaction 的变量做为 uvm_do 宏的参数。其实，uvm_do 系列宏不光可以接受一个 transaction 的变量做为参数，它还能接受一个 sequence 的变量做为参数。

```
class new_seq extends uvm_sequence#(mac_transaction);
    task body();
        crc_seq cseq;
        long_seq lseq;
        `uvm_do(cseq)
        `uvm_do(lseq)
    endtask
    `uvm_object_utils(new_seq)
endclass
```

把我们定义好的 sequence 做为 uvm_do 的参数，这样就实现了 sequence 的重用。这个功能也是非常强大的。

5.3. virtual sequence 的使用

5.3.1. 用事件做 sequence 之间的同步

到目前，为止，我们所学习的 sequence 机制就是一个 sequence 启动之后对应一个 sequencer，这个 sequence 发出 transaction，sequencer 把这个 transacton 转交给 driver。

但是考虑这样一种情况，验证平台中有两个 driver，这两 个 driver 分别做不同的事情，第一个 driver 相当于是一个 CPU，它要在 DUT 刚启动的时候，配置 DUT 的寄存器，当它配置完成后，另外一个 driver 才能发激励。这种情况在现实中是相当常见的。一块 DUT 在上电复位后虽然其默认的参数就可以工作，但是大部分情况下，CPU 还是要对 DUT 做一些配置，这样才能 DUT 工作在我们所期望的方式下。这个问题中，主要的就是有一个同步的过程，一种很自然的想法是，把这个同步的过程使用一个 event 来完成：

```
event config_over;//a global event
class config_seq extends uvm_sequence#(cpu_transaction);
    task body();
        …//some uvm_do statement, config DUT's register
        ->config_over;//trigger the global event
    endtask
    `uvm_object_utils(config_seq)
endclass

class mac_seq extends uvm_sequence#(mac_transaction);
    task body();
       @config_over;//wait the global event
       …//send mac transaction
    endtask
    `uvm_object_utils(mac_seq)
endclass
```

之后，我们通过 uvm_config_db 的方式分别把这两个 sequence 做为 cpu_sequencer 和 mac_sequencer 的 default_sequence：

```
uvm_config_db#(uvm_object_wrapper)::set(this, "env.cpu_sqr.main_phase", "default_sequence",
cpu_seq::type_id::get());

uvm_config_db#(uvm_object_wrapper)::set(this, "env.mac_sqr.main_phase", "default_sequence",
mac_seq::type_id::get());
```

当进入到 main_phase 时，这两个 sequence 会同步的启动，但是由于 mac_seq 要等待 config_over 事件的到来，所以它并不会马上产生 transaction。而 cpu_seq 则会

直接产生 transaction，交给 cpu_driver。当所有的配置工作完成后，config_over 事件被触发，于是 mac_seq 开始产生 transaction。

5.3.2. 复杂的同步：virtual sequence

上面的解决同步的方法看起来非常的简单，实用。不过这里有两个问题，第一个问题是使用了一个全局的事件 config_over。全局变量对于初写代码的人来说是非常受欢迎的一个东西，但是几乎是所有的老师及书本中都会这么说：除非有必要，否则尽量不要使用全局变量。使用全局变量的主要问题就是它是全局可见的，我们本来只是打算在 mac_seq 和 cpu_seq 中使用这个全局变量，但是假如其它的某个 sequence 也不小心使用了这个全局变量，在 cpu_seq 触发 config_over 事件之前，这个 sequence 已经触发了 config_over 事件，这样相当于是 DUT 还没有配置完成，mac_seq 就开始为 mac_driver 提供 mac_transaction 了，在某些情况下，这会造成比较严重的后果。所以应该尽量避免全局变量的使用。

第二个问题是上面只是实现了一次同步，如果是有多次同步怎么办？如 sequence A 要先执行，之后是 B，B 完了才能是 C，C 完了才能是 D，D 完了才能是 E。这依然可以使用上面的全局方法解决，只是这会显得相当的笨拙。

实现 sequence 之间同步的最好的方式就是使用 virtual sequence。从字面上理解，虚拟的 sequence。虚拟的意思就是它根本就不发送 transaction，它只是控制其它的 sequence，起统一调度的作用。

如图 5-3 所示，为了使用 virtual sequence，一般的需要一个 virtual sequencer。virtual sequencer 里面包含指向其它实际 sequencer 的指针。

```
class vsequencer extends uvm_sequencer;
    cpu_sequencer cpu_sqr;
    mac_sequencer mac_sqr;
    `uvm_component_utils(vsequencer);
endclass
```

在 test 中，可以例化 vsqr，并把相应的 sequencer 赋值给 vsqr 中的 sequencer 的指针。

```
class base_test extends uvm_test
    env env_inst;
    vsequencer vsqr;
    function build_phase(uvm_phase phase);
        …
        vsqr = vsequencer::type_id::create("vsqr", this);
        …
    endfunction
```

```
    function connect_phase(uvm_phase phase);
        …
        vsqr.cpu_sqr = env_inst.cpu_agent.cpu_sqr;
        vsqr.mac_sqr = env_inst.mac_agent.mac_sqr;
        …
    endfunction
endclass
```

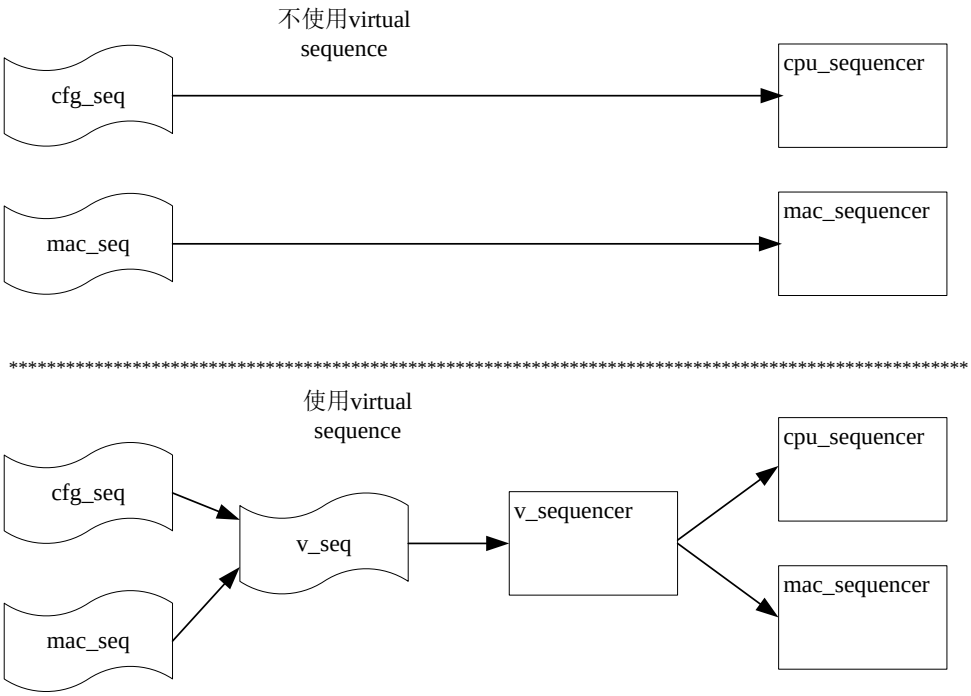


图 5-3 virtual sequence 的使用

在 virtual sequene 里面则可以使用 uvm_do 系列宏来发送 transaction：

```
class vseq extends uvm_sequence;
    task body();
        cpu_seq cseq;
        mac_seq mseq;
        `uvm_do_on(cseq, p_sequencer.cpu_sqr)
        `uvm_do_on(mseq, p_sequencer.mac_sqr)
    endtask
    `uvm_object_utils(vseq)
    `uvm_declare_p_sequencer(vsequencer)
endclass
```

这个 virtual sequence 里面出现了一些新的东西。我们没有使用前面的 uvm_do

或者 uvm_do_with 宏，而是使用了 uvm_do_on 宏。如果使用 uvm_do 宏，如下面的例子所示，那么 tr 最终会被交到启动这个 sequence 的 sequencer 手里，即 mac_sqr，再由其转交给相应的 driver。uvm_do_with 宏同样如此。

```
class mac_seq extends uvm_sequence#(mac_transaction);
    task body();
        mac_transaction tr;
        `uvm_do(tr)
    endtask
    `uvm_object_utils(mac_seq)
endclass
```

但是如果使用了 uvm_do_on 宏，如上面的 vseq 所示，如果是由 vsqr 启动这个 vseq，那么 cseq 最终会被交到 p_sequencer.cpu_sqr 手里，也即 vsqr.cpu_sqr 手里，而不是 vsqr 手里。这个就是 virtual sequence 和 virtual sequencer 中 virtual 的来源。它们各自并不产生 transaction，而只是控制其它的 sequence 为相应的 sequencer 产生 transaction。virtual sequence 和 virtual sequencer 只是起一个调度的作用。

在 vseq 的定义中，还出现了 p_sequencer 的变量，它是一个指向 sequencer 的指针，这个 sequencer 就是启动这个 vseq 的 sequencer。要使用 p_sequencer 这个变量，那么就要使用宏来显式的声明：

```
uvm_declare_p_sequencer(vsequencer)
```

经过这个宏的声明之后，UVM 就知道 p_sequencer 是一个 vsequencer 类型的变量，于是就可以使用 vsequencer 中的成员变量，如 cpu_sqr，mac_sqr 等。

为什么我们要使用这个宏，不用不行吗？因为我们要把具体的 sequence（mac_seq 和 cpu_seq）交给具体的 sequencer（mac_sqr 和 cpu_sqr），所以就要获得这些 sequencer 的指针，vsqr 中恰好有这些 sequencer 的指针。只有当我们需要在 sequence 中使用启动此 sequence 的 sequencer 中的某些成员变量时，我们才需要声明此宏。否则根本就不需要。在现实的应用中，一般就是在 virtual sequence 需要声明。除此之外几乎都不需要。

回顾一下，为了解决 sequence 的同步，之前使用了 config_over 这个全局变量的方式来解决。在 virtual sequence 中是如何解决的呢？事实上，这个问题在 virtual sequence 中根本就不是个问题。由于 virtual sequence 的 body 是顺序执行，所以只需要把有先后顺序的 sequence，顺序的使用 uvm_do 系列宏启动就可以了，没有必要去刻意的同步。这只是 virtual sequence 强大的调度功能的一个小小的体现。事实上，virtual sequence 还可以有更加强大的功能来等待读者挖掘。

在上节中，使用了两个 uvm_config_db 语句把两个 sequence 送给了相应的 sequencer 做为 default_sequence。假如验证平台中的 sequencer 是有 10 个，那么我们岂不是要写上 10 次的 uvm_config_db 语句？如果只是在一个测试 case 这样写也就罢了，问题是需要在所有的 case 中都需要写这样 10 句古怪的 uvm_config_db 的语句。这是一件很讨厌的事情。使用 virtual sequence 的另外一个好处就是，可以这把 10 句

只压缩成一句：

```
uvm_config_db#(uvm_object_wrapper)::set(this, "env.vsqr.main_phase", "default_sequence",
vseq::type_id::get());
```

这个也算是使用 virtual sequence 一个不大不小的好处了。

5.3.3. 在 sequence 中慎用 fork join_none

假设 DUT 中有四个完全相同的 MAC，那么相应的验证平台中也要有 4 个完全相同的 driver，sequencer。那么 vsequencer 就要这样定义：

```
class vsequencer extends uvm_sequencer;
    cpu_sequencer cpu_sqr;
    mac_sequencer mac_sqr[4];
    `uvm_component_utils(vsequencer);
endclass
```

当 DUT 上电复位，CPU 把寄存器配置完成后，需要四个 mac_driver 同时发送数据，在 vseq 中可以使用 fork 来使四个 driver 同时发送数据：

```
class vseq extends uvm_sequence;
    task body();
        cpu_seq cseq;
        mac_seq mseq;
        `uvm_do_on(cseq, p_sequencer.cpu_sqr)
        for(int i = 0; i < 4; i++) begin
            fork
                begin
                    int local_i = i;
                    `uvm_do_on(mseq, p_sequencer.mac_sqr[local_i])
                end
            join_none
        end
    endtask
    `uvm_object_utils(vseq)
    `uvm_declare_p_sequencer(vsequencer)
endclass
```

这里使用了 join_none，由于 join_none 的特性，系统并不等 fork 起来的进程结束就进入了下一次的 for 循环，因此上面的 for 循环的展开后如下：

```
class vseq extends uvm_sequence;
    task body();
        cpu_seq cseq;
        mac_seq mseq;
        `uvm_do_on(cseq, p_sequencer.cpu_sqr)
```

```
        fork
            `uvm_do_on(mseq,  p_sequencer.mac_sqr[0])
        join_none
        fork
            `uvm_do_on(mseq,  p_sequencer.mac_sqr[1])
        join_none
        fork
            `uvm_do_on(mseq,  p_sequencer.mac_sqr[2])
        join_none
        fork
            `uvm_do_on(mseq,  p_sequencer.mac_sqr[3])
        join_none
    endtask
    `uvm_object_utils(vseq)
    `uvm_declare_p_sequencer(vsequencer)
endclass
```

大家仔细想想看，这样会有什么问题？

我们前面说过，当 sequence 启动起来的时候，会自动执行这个 sequence 的 body 任务。当 body 执行完成的时候，那么这个 sequence 就相当于是已经完成了其使命，已经结束了。如果使用 fork join_none，由于 join_none 的特性，当使用 uvm_do_on 宏把四个 mseq 分别放在四个 mac_sqr 上执行时，系统会新起动 4 个进程，但是系统并不等待这 4 个 mseq 执行完毕就直接返回了。返回之后就到了 endtask，此时系统认为这个 sequence 已经执行完成了。执行完成之后，系统将会清理这个 sequence 之前占据的内存空间，杀死掉由其起动起来的进程，于是这 4 个启动起来的 mseq 还没有完成就直接被系统杀死掉了。也就是说，看似分别往 4 个 mac_sqr 分别丢了一个 sequence，但是事实上这个 sequence 根本没有执行。这是关键所在！

要避免这个问题有多种方法，一是使用 wait fork 语句：

```
for()  begin
    fork
        …
    join_none
end
wait  fork;
```

wait fork 语句将会等待前面被 fork 起来的进程执行完毕。

另外一种方法是使用 fork join：

```
fork
    `uvm_do_on(mseq,  p_sequencer.mac_sqr[0])
    `uvm_do_on(mseq,  p_sequencer.mac_sqr[1])
    `uvm_do_on(mseq,  p_sequencer.mac_sqr[2])
    `uvm_do_on(mseq,  p_sequencer.mac_sqr[3])
join
```

5.3.4. 在 virtual sequence 中控制验证平台的关闭

在 5.1.4 节，我们提到，在 sequence 中使用 starting_phase 来控制验证平台的关闭。前面我们也看过了，只有把此 sequence 做为 sequencer 的某动态运行 phase 的 default_sequence 时，其 starting_phase 才不为 NULL。所以，如果把某 sequence 做为 uvm_do 宏的参数，那么此 sequence 中的 starting_phase 是为 null 的。在此 sequence 中使用 starting_phase.raise_objection 是会产生问题的。这个问题其实比较容易解决，只要把父 sequence 的 starting_phase 赋值给子 sequence 的 starting_phase 就可以了，这样只要最顶层的 sequence 的 starting_phase 不为 null，那么下面所有由其启动的 sequence 的 starting_phase 也不为 null。只是可惜我们不是 UVM 的开发者，所以没有办法修改 uvm_do 系列宏。不过读者如果有这样的需求，可以给 accellera（负责开发 UVM 的官方组织）发邮件，让其在 uvm_do 宏中加入 starting_phase 的传递功能。

在 5.3.2 节中，同样的，如果在 cpu_seq 和 mac_seq 中使用 starting_phase.raise_objection 也是会产生问题的。这就促使我们思考，在哪些 sequence 中使用 raise_objection 来最终控制验证平台的关闭呢？

这里的答案比较简单，那就是在最顶层的 sequence 中。最顶层的 sequence 肯定不会做为 uvm_do 系列宏的参数（否则也就不叫最顶层的 sequence 了），通常的把最顶层的 sequence 做为 sequencer 的 default_sequence。virtual sequence 一般会充当最顶层 sequence 的角色。也就是说，一般的可以在 virtual sequence 中使用 starting_phase.drop_objection 来控制验证平台的关闭。通常说来，在一个 case 中只有一个 virtual sequence，在这唯一的一个 virtual sequence 中控制验证平台的关闭也会减少出问题的概率。

6.config 机制

前面几章分别介绍了 uvm_component 和 uvm_object 的概念，UVM 的 factory 机制，UVM 的 field_automation 机制，UVM 的 phase 和 objection 机制及 UVM 的 sequence 机制。在实际的验证平台中，除了用到上面说的外，还要用到本章将要介绍了 config 机制。学习完了本章之后，第一章中所介绍的几乎所有的内容都已经涉及了，可以利用前面的内容搭建起一个足够强大的验证平台。

本章第一节将会介绍 config 机制的由来，第二节介绍令人眼花缭乱的 config 机制的功能，第三节介绍用一个 object 来聚合多个 config 变量。

6.1.config 机制的前世今生

当我学习 config 机制的时候，我对这种机制的的必要性曾经深深的怀疑过。因为用 config 机制能够做到的事情，用其它的方法也能做到。

6.1.1. 验证平台中要配置的众多的参数

在一个验证平台中，有众多的参数要配置。如 mac_driver 中，按照以太网相关

协议的规定，在发送一帧数据前要先发送几个 preamble（即 8'h55），1 个 sfd(即 8'hD5)。那要发送几个 preamble 呢？这个可以在 driver 中设置一个 pre_num 的变量来控制。

```
class mac_driver extends uvm_driver#(mac_transaction);
    …
    int pre_num;
    local int pre_num_min;
    local int pre_num_max;
    function void build_phase(uvm_phase phase);
        super.build(phase);
        pre_num_min = 3;
        pre_num_max = 7;
    endfunction
endclass
```

之后在 main_phase 中，每发送一个 transaction 之前，先把 pre_num 随机化一次，得到一个 3 到 7 之间任意的数字，然后把这个数字作为发送 premble 的数量：

```
task mac_driver::main_phase(uvm_phase phase);
    super.main_phase(phase);
    while(1) begin
        seq_item_port.get_next_item(req);
        pre_num = $urand_ranget(pre_num_min, pre_num_max);
        …//drive this pkt, and the number of preamble is pre_num
        seq_item_port.item_done();
    end
endtask
```

在只发送正常帧的情况下，上面的代码是可以工作的。那如果现在想新增加一个 case，测试一下 DUT 对于 preamble 数量的敏感性，也就是说要发送小于 3 个的或者大于 7 个的 preamble。那应该怎么办呢？很明显的一点，这种情况是跟具体的 case 相关的，新加的这个 case 不能影响其它 case 的正常运行，同时我们也希望，mac_driver 尽量不因为加了这个 case 而做改变。

6.1.2. config 机制的本质：半个全局变量

要实现上一小节提出的这个功能，有众多的方法。一种方法是把 pre_num_max 和 pre_num_min 设置成全局变量，这样，当要发送小于 3 个的 preamble 时，可以把 pre_num_max 设置为 2，把 pre_num_min 设置为 0。当要发送大于 7 个的 preamble 时，可以把 pre_num_max 设置为 100，把 pre_num_min 设置为 8。上一章在介绍 virtual sequence 时，我们提到了类似 config_over 这种全局变量应该尽量避免使用。这里也是如此，过多的使用全局变量最后如果发现全局变量的值跟我们预期的不一样，那么排查这个全局变量是如何改变的将会是一件相当痛苦的事情。使用全局变量大大

增加了出错的概率。

与全局变量相对应的就是本地变量，但是很明显，本地变量的值要想在不同的 case 中改变会是相当困难的一件事情。那么有没有这么样的一种全局变量，即它只能被某些特定的 class 修改，而不能被其它的修改呢？也就是说，这个变量对于一些类是可见的，对于另外一些类是不可见的，似乎就像是一个半全局变量一样。

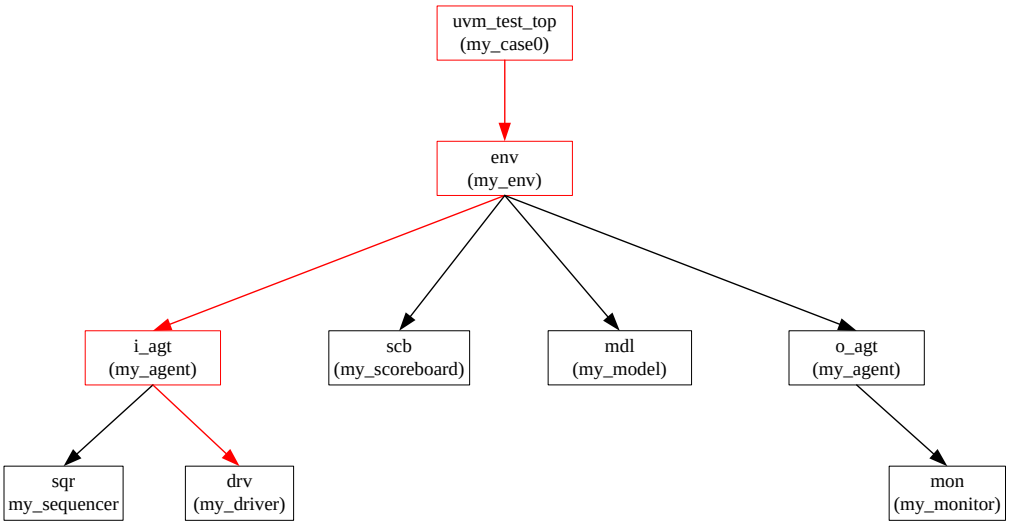


图 6-1 半全局变量

在上图中，我们希望只能在 test 和 env 及 i_agt 中才能改变 driver 中 pre_num_max 和 pre_num_min 的值。而类似 monitor，scoreboard，reference model 等都不能改变。读者可能会说，在 uvm_test_top 中可以使用如下方式给 pre_num_max 赋值：

```
env.i_agt.drv.pre_num_max = 10;
```

如果 pre_num_max 为 public（即非 local 也非 protected）的变量，那么这种方法是可行的。但是，在 pre_num_max 为 local 的情况下，这种法子是不可行的。

无论任何语言，都不会提供这种半全局变量。要想达到这种半全局变量的效果，必须通过某些特殊的机制实现。UVM 提供了 config 机制用以实现这种要求。

6.1.3. config 机制是用来传递数据的

config 机制在 UVM 验证平台中都是成对出现的。如在某个 case（派生自 uvm_test）的 build_phase 中可以做如下设置：

```
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 100);
```

那么在 driver 的 build_phase 中要这样做：

```
uvm_config_db#(int)::get(this, "", “pre_num_max”, pre_num_max);
```

这样，设置的 pre_num_max 的数值 100 就会传递给 driver 的 pre_num_max。

uvm_config_db 中的 set 和 get 都是静态函数，所以可以用双冒号的形式调用。上面 set 中的第一个参数用以说明是哪个 component 对 pre_num_max 进行了设置，一般使用填写 this，第二个参数表示从调用 uvm_config_db::set 的地方看下去，要设置的变量所在的 component 的路径。第三个参数表示一个记号，用以说明这个值是传给 driver 中的哪个变量的，第四个参数是要设置的值。get 中的第一个参数一般也是 this 即可，第二个参数填写一个空的字符串，第三个参数就是 set 中的第三个参数，这两个参数必须匹配起来，第四个参数则是要设置的变量。这里要说明的是，set 和 get 中第三个参数可以跟 get 中第四个参数可以不一样。如第四个参数是 pre_num_max，那么第三个参数可以是 pre_max，只要保持 set 和 get 中第三个参数一致即可。

打个比方来说，set 就相当于是寄出了一封信，在寄信的时候要说明是谁寄的，这就是第一个参数；要说明收信人的单位，这就是第二个参数；要说明收信人的名字，这就是第三个参数，信的内容就是第四个参数。get 就相当于是收信。收信时，要说明是谁收的，这就是 get 的第一个参数；收信人的单位，由于这本身就是在 driver 里面，所以只要填写一个空的字符串，系统会自动补齐收信人的单位；匹配一下收信人的名字，这就是第三个参数；具体的由谁来接收信的内容，这就是第四个参数。至于 get 中为什么第三个参数可以和第四个参数不同，可以这样理解：张三给李四寄了一封信，信上写了李四的名字，这样李四可以收到。但是呢，由于保密性的需要，张三只是在信上写了“四”这一个字，那么只要张三跟李四约定好了，那么李四只要一看到上面写着“四”的信就会收下来。

6.2. 强大的 config

本节说一下 config 中的一些比较强大的功能。

6.2.1. 省略 get 的 config

6.1.3 节说过，config 总是 set 和 get 成对出现的。在 build_phase 中，要写上如下

的两句话才能把 pre_num_max 和 pre_num_min 的值更新为 case 的设置值：

```
uvm_config_db#(int)::get(this, "", "pre_num_max", pre_num_max);
uvm_config_db#(int)::get(this, "", "pre_num_min", pre_num_min);
```

这里只有两个变量，尚且还能忍受，如果是有 100 个变量，那么写上这么 100 行古怪的语法，那将会是相当痛苦的一件事情。

其实在一些特定的情况下，get 是可以省略的。还记得我们每次在 build_phase 中都要写上这么一句么？

```
super.build_phase(phase);
```

读者肯定非常疑惑这句话都做了哪些事情。在特定情况下，这句话就可以完成 get 的功能。

```
class mac_driver extends uvm_driver#(mac_transaction);
    …
    int pre_num;
    int pre_num_min;
    int pre_num_max;
    `uvm_component_utils_begin(mac_driver)
        `uvm_field_int(pre_num_min, UVM_ALL_ON)
        `uvm_field_int(pre_num_max, UVM_ALL_ON)
    `uvm_component_utils_end
    function void build_phase(uvm_phase phase);
        super.build(phase);
        //uvm_config_db#(int)::get(this, "", "pre_num_max", pre_num_max);
        //uvm_config_db#(int)::get(this, "", "pre_num_min", pre_num_min);
    endfunction
endclass
```

只要在把 mac_driver 注册到 factory 时，顺便使用 field_automation 机制把要 get 的变量注册，那么当 UVM 执行到 driver 的 super.build_phase(phase)这句话时，就会自动执行那两 get 的语句。这是相当便利的。在前面介绍 field_automation 机制时，一直是以一个 transaction 为例子进行介绍的，其实对于派生自 uvm_object 的类（如 transaction）和 uvm_component 的类（如 uvm_driver）中，是都可以使用 field_automation 机制的。

不过，这里还存在一个问题，那就是如果要让 super.build_phase 能够自动 get 到设置的值的话，那么在 set 的时候，set 的第三个参数必须与要 get 的变量的名字相一致。所以 6.1.3 节中，虽然说这两个参数可以不一致，但是最好的情况下还是一致。李四的信就是给李四的，不要打什么暗语，用一个“四”来代替李四。

这个强大的功能是 config 机制和 field_automation 机制联合作用的结果，其实不能只算在 config 机制的头上的。

6.2.2. 跨层次的多重 set

前面一直强调的一个概念就是 set 与 get 总是成对出现的。但是假如 set 多次，而 get 一次，那么最终 get 到的是哪一个 set 值呢？在现实生活中，这可以理解成有好多人都给李四发了一封信，要求李四做某件事情，但是这些信是相互矛盾的。那么李四有两种方法来决定听谁的：一是以收到的时间为准，最近收到的信具有最高的权威，当同时收到两封信时，则看发信人的权威性，也即时间的优先级最高，发信人的优先级次之；二是以看发信人，哪个发信人最权威就听谁的，当同一个发信人先后发了两封信时，那么最近收到的一封权威高，也就是发信人的优先级最高，而时间的优先级低。UVM 中采用类似第二种方法的机制。

在图 6-1 中，假如 uvm_test_top 和 env 中都对 driver 的值进行了 set，在 uvm_test_top 中的 set 语句如下：

```
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 100);
```

在 env 的 set 语句如下：

```
uvm_config_db#(int)::set(this, "agent.driver", “pre_num_max”, 999);
```

那么 driver 中 get 到的值是 100 还是 999 呢？答案是 100。UVM 规定层次越高，那么它 set 的优先级越高。在整个 UVM 树中，uvm_test_top 的位置是高于 env 的，所以 uvm_test_top 中的 set 的优先级高。

看起来 UVM 似乎很势利，狗眼看人低，见到高层次的就点头哈腰，马上就从了。UVM 这样设置是有其内在道理的。对于用户来说，是 uvm_test_top 更接近用户还是 env 更接近用户呢？答案肯定是前者。我们会在 uvm_test_top 中写不同的 sequence，从而衍生出很多不同的 case 来。而对于 env，它在 uvm_test_top 中实例化。有时候，这个 env 根本就不是用户自己开发的，很可能是别人已经开发好的一个非常成熟的可重用的模块。对于这种成熟的模块，如果觉得其中某些参数不合要求，那么难道要到 env 中去修改相关的 set 参数吗？显然这是不合理的。比较合理的就是在 uvm_test_top 的 build_phase 中通过 set 的方式修改。所以说，UVM 这种看似势利的行为其实极大的方便了用户的使用。

6.2.3. 同一层次的多重 set

当跨层次来看待问题时，是高层次的 set 优先，当处于同一层次时，则是时间优先。

```
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 100);
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 109);
```

当上面两句话同时出现在 test 的 build_phase 中时，driver 最终 get 到的值将会是 109。像上面的这种用法看起来完全是胡闹，没有任何意义。但是考虑这种情况：

pre_num_max 在 99%的 case 中的值都是 7，只有 在 1%的 case 中才会是其它值。那么是不是要这么写呢？

```
class case1 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “env.agent.driver”, pre_num_max, 7);
    endfunction
endclass
…
class case99 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “env.agent.driver”, pre_num_max, 7);
    endfunction
endclass
class case100 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “env.agent.driver”, pre_num_max, 100);
    endfunction
endclass
```

前面 99 个 case 的 build_phase 里面都是同样的一句话，这样写起来是相当不具有重用性的。因为可能忽然有一天，99%的 case 中，pre_num_max 的值要变成 6，那么就需要把 99 个 case 中所有的 set 语句都要改变。这是相当耗时间的，而且是极易出错的。解决这个问题的办法就是在 base_test 的 build_phase 中进行 set，这样，当由 base_test 派生而来的 case1~case99 在执行 super.build_phase(phase)时，都会进行 set：

```
classs base_test extends uvm_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “env.agent.driver”, pre_num_max, 7);
    endfunction
endclass
class case1 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    endfunction
endclass
…
class case99 extends base_test;
    function void build_phase(uvm_phase phase);
```

```
        super.build_phase(phase);
    endfunction
endclass
```

但是对于第 100 个 case，则依然需要这么写：

```
class case100 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “env.agent.driver”, pre_num_max, 100);
    endfunction
endclass
```

case100 的 build_phase 就相当于是如下所示连续 set 了两次：

```
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 7);
uvm_config_db#(int)::set(this, "env.agent.driver", “pre_num_max”, 100);
```

按照时间优先的原则，后面 set 的将最终被 driver 得到。

6.3. 聚合 config 变量

6.3.1. 用专门的类来组织 config 变量

到这里为止，本章目前所涉及的都是少量数据在不同的 component 之间的传递。对于一个大的项目来说，要配置的参数可能有千百个。如果全部按照前面的写法，那么就会出现下面这种情况：

```
classs base_test extends uvm_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(int)::set(this, “path1”, “var1”, 7);
        …
        uvm_config_db#(int)::set(this, “path1000”, “var1000”, 999);
    endfunction
endclass
```

可以想像，这 1000 句 set 写下来将会是多么壮观的一件事情。但是壮观的同时也显示出了这是多么麻烦的一件事情。

一种比较好的方法就是把这 1000 个变量放在一个专门的类里面来实现：

```
class iconfig extends uvm_object;
    rand int var1;
    …
    rand int var1000;
    constraint default_cons{
        var1 = 7;
        …
        var1000 = 999;
    }
    `uvm_object_utils_begin(iconfig)
        `uvm_field_int(var1, UVM_ALL_ON)
        …
        `uvm_field_int(var1000, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

经过如下的定义之后，我们可以在 base_test 中这样写：

```
classs base_test extends uvm_test;
    iconfig cfg;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        cfg = iconfig::type_id::create("cfg");
        uvm_config_db#(iconfig)::set(this, "env.agent.driver", "cfg", cfg);
        uvm_config_db#(iconfig)::set(this, "env.agent.monitor", "cfg", cfg);
        …
    endfunction
endclass
```

这样，把省略了绝大多数的 set 语句。但是相应的，这样的代价就是在 driver 中必须要这样写：

```
class mac_driver extends uvm_driver#(mac_transaction);
    iconfig cfg;
    `uvm_component_utils_begin(mac_driver)
        `uvm_field_object(cfg, UVM_ALL_ON | UVM_REFERENCE)
    `uvm_component_utils_end
    extern task main_phase(uvm_phase phase);
endclass
task mac_driver::main_phase(uvm_phase phase);
    super.main_phase(phase);
    while(1) begin
        seq_item_port.get_next_item(req);
        pre_num = $urand_range(cfg.pre_num_min, cfg.pre_num_max);
        …//drive this pkt, and the number of preamble is pre_num
        seq_item_port.item_done();
    end
endtask
```

如果在某个 case 中想要改变某个变量的值，应该怎么做呢？很简单：

```
class case100 extends base_test;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        cfg.pre_num_max = 100;
        cfg.pre_num_min = 8;
        …
    endfunction
endclass
```

6.3.2. 实时的改变 config 值

有时候，可能当 DUT 运转到某一时刻时，需要改变验证平台的某些配置参数。这种情况下可以通过 virtual sequence 的方式实现。

```
class vsequencer extends uvm_sequencer;
    iconfig cfg;
    …
endclass
classs base_test extends uvm_test;
    iconfig cfg;
    vsequenccer vsqr;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        cfg = iconfig::type_id::create("cfg");
        vsqr = vsequenccer::type_id::create("vsqr", this);
        vsqr.cfg = this.cfg;
        …
    endfunction
endclass
class vseq extends uvm_sequence;
    `uvm_object_utils(vseq)
    `uvm_declare_p_sequencer(vsequencer)
    task body();
        …//send some transaction
        p_sequencer.cfg.pre_num_max = 99;
        …//send other transaction
    endtask
endclass
```

通过这种方法，可以在 sequence 中实时的改变验证平台的配置参数。不过要注意的是，这种方式一种要小心使用，因为在使用时，很可能一不小心把其它不应改变的参数的值给改了。

6.3.3. 在 sequence 中设置 driver 要发送的包的数量

在 3.2.3 节中，在提到验证平台如何关闭时，我们曾经说过可以在 sequence 中设置 driver 要发送的包的数量。到现在为止，读者想到怎么设置了吗？

我们首先需要在 iconfig 中加入一个变量 pkt_num，并把其默认值设置为某个数值，如 100。

之后在 virtual sequence 刚启动的时候可以给 pkt_num 赋值成想要设置的值：

```
class vseq extends uvm_sequence;
    …
    task body();
        p_sequencer.cfg.pkt_num = 10;
        …//send other transaction
    endtask
endclass
```

同时，在 driver 中可以这样写：

```
task mac_driver::main_phase(uvm_phase pahse);
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < cfg.pkt_num; i++) begin
        seq_item_port.get_next_item(req);
        …//drive the interface according to the information in req
    end
    phase.drop_objection(this);
endtask
```

这样就可以不用在 sequence 中 raise_objection 了。在 3.2.3 节时，我们强烈的感觉到这种做法的必要性。但是现在，其实根本不用这么做。因为只要在 sequence 中 raise_objection 和 drop_objection 就可以了，完全没有必要在 driver 中进行控制。

7.UVM 的各种 port

UVM 中内置了各种 port，用于实现 TLM 级别的通信。在阅读本章时，读者要注意其中的代码并不能直接使用。如为了说明情况，会把一个 agent 类的名字定为 agent，同时在 env 中把其实例的名字也称为 agent。这种类的名字和实例的名字一样的情况显然是不行的。

7.1. port 与 TLM

7.1.1. uvm_component 之间的通信

如果要在两个 uvm_component 之间通信，如一个 monitor 和一个 scoreboard 之间通信，你能想出多少种方法？

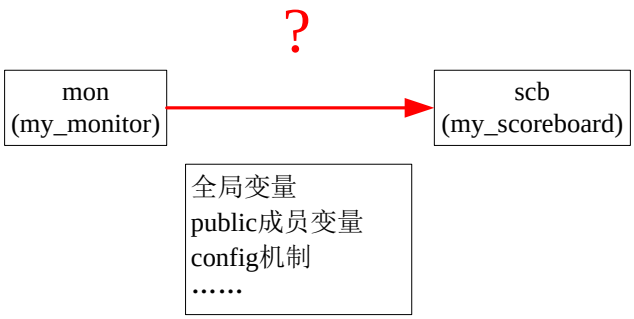


图 7-1 monitor 与 scoreboard 的通信

最简单的方法就是使用全局变量，在 monitor 里对此全局变量进行赋值，在 scoreboard 里监测此全局变量值的改变。这种方法简单，直接，不过恰如前面的章节一直强调的，要谨慎使用全局变量，滥用全局变量只会造成灾难性的后果。

稍微复杂一点的方法，在 scoreboard 中有一个变量，这个变量设置为外部可以直接访问的，即非 local 类型的，然后在 monitor 中给这个变量赋值，如图 7-2 所示。要完成这个任务，那么要在 monitor 中有一个指向 scoreboard 的指针，否则虽然 scoreboard 把这个变量给设置为非 local 类型的，但是 monitor 依然无法改变。

这种方法的问题就在于，整个 scoreboard 里面的所有的非 local 类型的变量都对 monitor 是可见的，而假如 monitor 的开发人员不小心改变了 scoreboard 中的一些变量，那么后果将可能会是致命的。

由 config 机制的半全局变量的特性，我们可以想出第三种方法来。即专门的实例化一个 config 类，在此类中有 monitor 要传给 scoreboard 的变量。在 base_test 中，可以把这个 config 类 set 给 scoreboard 和 monitor。当 monitor 要和 scoreboard 通信时，只要把此 config 类的相应变量的值改变即可。scoreboard 中则监测变量值的改变，监测到之后做相应动作。这种方法比上面的两种方法都要好，但是仍然显得有些笨拙。一是要引入一个专门的 config 类，二是一定要有 base_test 这个第三方的参与。在大多数情况下，这个第三方是不会惹麻烦的。但是永远不能保证某一个从 base_test 派生而来的类会不会改变这个 config 类中某些变量的值。也就是说，依然存在一定的风险。而且，这种方法显得有些笨拙。

解决这个问题最好的办法就是在 monitor 和 scoreboard 之间专门建立一个通道，让信息只能在这个通道内流动，scoreboard 也只能从这个通道中接收信息，这样几乎就可以保证 scoreboard 中的信息只能从 monitor 中来，而不能从别的地方来。UVM 中的各种 port 就可以实现这种功能。

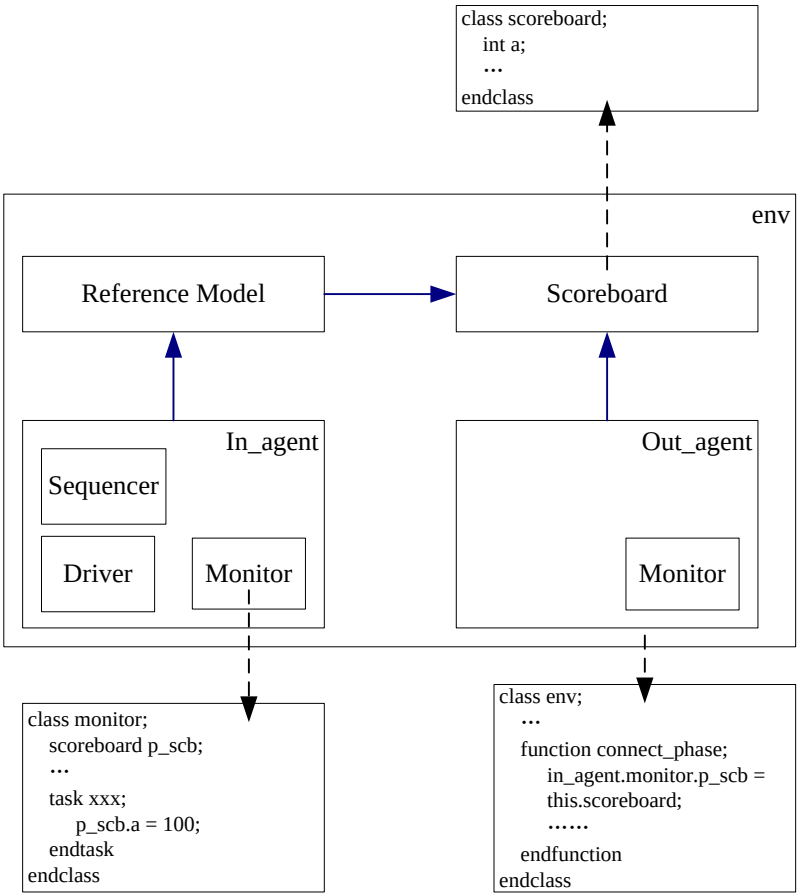


图 7-2 使用 public 成员变量实现通信

7.1.2. TLM 级别的通信

TLM 它是 Transaction Level Modeling 的缩写，所谓的 transaction level 是相对 DUT 中各个 module 之间信号线级别的通信来说的。相信读过第四章之后，读者对 transaction 这个名词有了深刻的认识。所谓的 transaction 就是把具有某一特定功能的一组信息封装在一起而成为的一个类。如 mac_transaction 就是把一个 mac 帧里的各个字段封装在了一起。

TLM 级别的通信中有几个常用的术语：

put 操作，如下图所示，通信的发起者 A 把一个 transaction 发送给 B。在这个过程中，A 称为“发起者”，而 B 称为“目标”。A 具有的端口（用方框表示）称为 PORT，而 B 的端口（用圆圈表示）称为 EXPORT。这个过程中，数据流是从 A 流向 B 的。

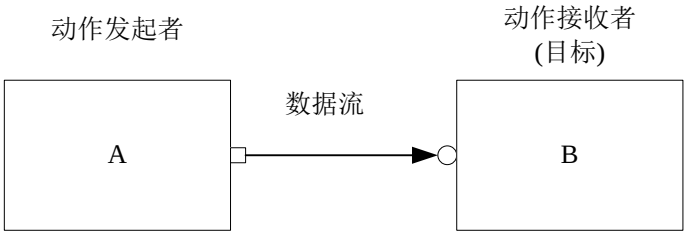


图 7-3 put 操作

get 操作，如下图所示，A 向 B 索取一个 transaction。在这个过程中，A 依然是“发起者”，B 依然是“目标”，A 上的端口依然是 PORT，而 B 上的端口依然是 EXPORT。这个过程中，数据流是从 B 流向 A 的。到这里，读者应该意识到，PORT 和 EXPORT 体现的是控制流而不是数据流。因为在 put 操作中，数据流是从 PORT 流向 PORT 的，而在 get 操作中，数据是从 EXPORT 流向 PORT 的。但是无论是 get 还是 put，其发起者拥有的都是 PORT 端口，而不是 EXPORT。作为一个 EXPORT 来说，只能被动的接收 PORT 的命令。

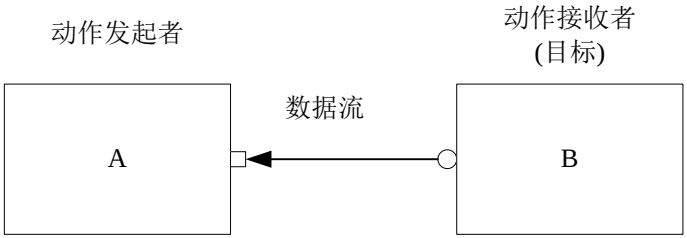


图 7-4 get 操作

transport 操作，如下图所示，transport 操作相当于一次 put 操作加一次 get 操作，这两次操作的“发起者”都是 A，目标都是 B。A 上的端口依然是 PORT，而 B 上的端口依然是 EXPORT。在这个过程中，数据流先从 A 流向 B，再从 B 流向 A。在现实世界中，相当于是 A 传递给了 B 一个 REQUEST，而 B 返回给 A 一个 RESPONSE。所以这种 transport 操作也常常被称做 request-response 操作。

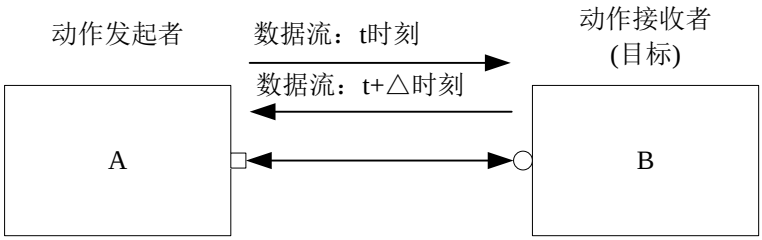


图 7-5 transport 操作

阻塞和非阻塞(blocking and non-blocking)，这两个术语对于有 verilog 代码经验的人来说是比较熟悉的，因为 verilog 中就有阻塞赋值和非阻塞赋值。TLM 中的这两个术语没有 verilog 中的那么复杂。以 put 操作为例，当 A 发起一个 put 操作，要把一个 transaction 交给 B 时，B 可能并不一定有时间立刻接收这笔 transaction。这个时候对于 A 来说有两种处理方法，一种方法是等在那里，一直等到 B 处理完事情，然后接收这个 transaction，另外一种方法是不等待，直接返回，至于后面是过一段时间继续发还是直接放弃不发了，则要看代码编写者的行为。前面一种想法相应的就是阻塞 put 操作，而后一种方法就是非阻塞 put 操作 。与 put 操作类似，get 和 transport 操作来，同样的有阻塞和非阻塞之分，这里不多说。

7.1.3. UVM 中常见的 port

为了实现 TLM 中的三种操作，UVM 中内建了三种端口：

一是 port，对应于不同的操作，有各种各样的 PORT：

```
uvm_blocking_put_port#(T);
uvm_nonblocking_put_port#(T);
uvm_put_port#(T);
uvm_blocking_get_port#(T);
uvm_nonblocking_get_port#(T);
uvm_get_port#(T);
uvm_blocking_transport_port#(REQ, RSP);
uvm_nonblocking_transport_port#(REQ, RSP);
uvm_transport_port#(REQ, RSP);
```

前 6 个定义中的参数就是这个 PORT 中的数据流类型，而后 3 个定义中参数则表示 transport 操作（request-response 操作）中发起请求时传输的数据类型和返回的数据类型。这几种 PORT 对应 TLM 中的三种操作，同时以 blocking 和 nonblocking 区分。对于名称中不含这两者的，则表示这个端口即可以用作是阻塞的，也可以用做是非阻塞的，否则只能用于阻塞的或者只能用于非阻塞的。其实 port 还有更多种，如 uvm_blocking_peek_port#(T)，这些都比较少用，因此这里不多做介绍，读者如有

兴趣可以看 UVM 的文档或本书后面关于 TLM 的源代码分析。由这种划分方法可以看出，UVM 天生的把一个 port 定为了只能执行某种操作，如对于 uvm_blocking_put_port#(T)，它只能执行非阻塞的 put 操作，想要执行阻塞的 put 操作是不行的，想要执行 get 操作，也是不行的，更不用提执行 transport 操作了。所以在定义前大家一定要想清楚了，这个端口将会用于什么操作。

二是 export，同样的也有多种 export：

```
uvm_blocking_put_export#(T);
uvm_nonblocking_put_export#(T);
uvm_put_export#(T);
uvm_blocking_get_export#(T);
uvm_nonblocking_get_export#(T);
uvm_get_export#(T);
uvm_blocking_transport_export#(REQ, RSP);
uvm_nonblocking_transport_export#(REQ, RSP);
uvm_transport_export#(REQ, RSP);
```

前 6 个定义中的参数就是这个 EXPORT 中的数据流类型，而后 3 个定义中参数则表示 transport 操作（request-response 操作）中发起请求时传输的数据类型和返回的数据类型。从 7.1.2 节中的 TLM 通信中的定义，大家看到了，PORT 是可以发起 put，get，transport 等操作，其目标对象都是 export。但是其中并没有说明 EXPORT 可以发起这三种操作。PORT 和 EXPORT 是体现的是一种控制流，在这种控制流中，PORT 具有高优先级，而 EXPORT 具有低优先级。只有高优先级的才能向低优先级的发起三种操作。

除了以上的两种端口外，UVM 中加入了第三种端口：IMP。UVM 中的 IMP 如下所示：

```
uvm_blocking_put_imp#(T, IMP);
uvm_nonblocking_put_imp#(T, IMP);
uvm_put_imp#(T, IMP);
uvm_blocking_get_imp#(T, IMP);
uvm_nonblocking_get_imp#(T, IMP);
uvm_get_imp#(T, IMP);
uvm_blocking_transport_imp#(REQ, RSP, IMP, REQ_IMP, RSP_IMP);
uvm_nonblocking_transport_imp#(REQ, RSP, IMP, REQ_IMP, RSP_IMP);
uvm_transport_imp#(REQ, RSP, IMP, REQ_IMP, RSP_IMP);
```

先来看前六个定义。这六个定义中的第一个 T 是这个 IMP 传输的数据类型。第二个参数 IMP 则让人非常费解。根据 UVM 文档中的相关解释，这个 IMP 指的是实现这个接口的一个 component。这句话怎么理解呢？

回顾图 7-3 的 put 操作，A 通过其端口 PORT 把一个 transaction 传送给 B，这个 port 在 transaction 传输的过程中起了什么作用呢？PORT 恰如一道门，EXPORT 也如此。既然是一道门，那么它们也就只是一个通行的作用，它不可能把一笔 transaction 存储下来，因为它只是一道门，没有存储作用。当我们写下 A.PORT.put(transaction)

时，此时 B.EXPORT 会通知 B 有东西过来了，那么 B 是如何知道有东西过来了呢？可以简单理解成 A.PORT.put (transaction)这个任务会调用 B.EXPORT 的一个任务，如 B.EXPORT.put(transaction)，而这个任务最终又会调用 B 的相关任务，如 B.put(transaction)。所以关于 B 的操作最终会落到 B.put 这个任务上，这个任务是属于 B 的一个任务，与 A 无关，与 A 的 PORT 无关，也与 B 的 EXPORT 无关。也就是说，这些 put 操作最终还是要由 B 这个 component 来实现，即要由一个 component 来实现接口的操作。

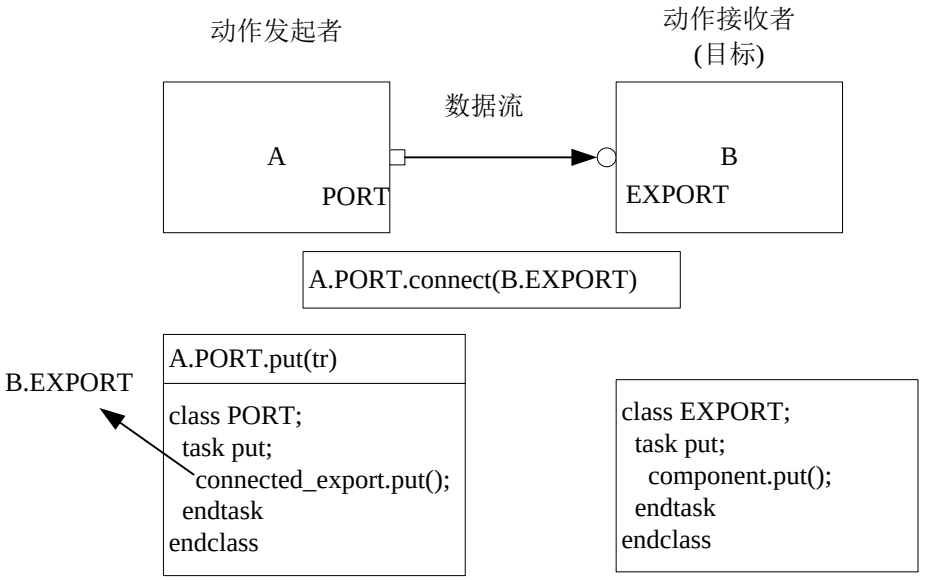


图 7-6 component 在端口通信中的作用

uvm_blocking_transport_imp 及其后的另外两个 IMP 的定义相当繁琐，而且其使用频率并不高，对于初学者来说，可以暂时先忽略它们。如果有兴趣仔细研究，那么可以看本书后面关于 TLM 的源代码分析。

IMP 的这些定义中的 blocking，nonblocking，put，get，transport 等关键字的意思并不是它们发起做相应类型的操作，而只意味着，它们可以和相应类型的 port 或者 export 进行通信，且通信时作为被动承担者。

7.2. UVM 中各种 port 的连接

按照控制流的优先级排序，UVM 中三种 port 为：PORT，EXPORT，IMP。这三种 port 之间并不是互相之间都可以连接的。

7.2.1. 使用 connect 建立连接关系

如图所示，四个端口，我们要在 A 和 B 之间通信，在 C 和 D 之间通信。为了实现这个目标，必须要在 A 和 B 之间，C 和 D 之间建立一种连接关系，否则的话，A 如何知道是和 B 通信而不是和 C 或者 D 通信？所以连接关系是一定要建立的。

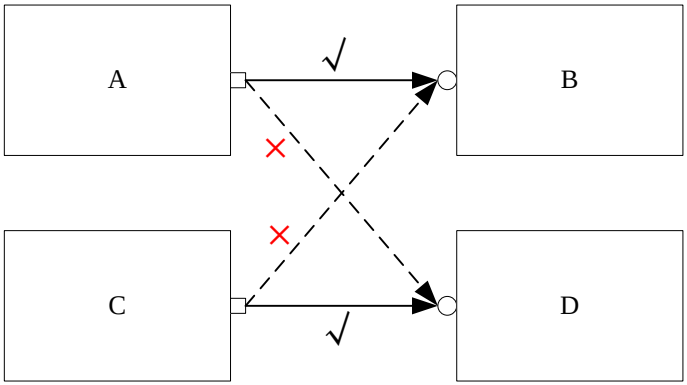


图 7-7 connect 关系的建立

UVM 中使用 connect 函数来建立连接关系。如 A 要和 B 通信（A 是发起者），那么可以这么写：

```
A.connect(B);
```

但是我们不能写 B.connect(A)。因为在通信的过程中，A 是发起者，B 是被动承担者。这种通信时的主次顺序也适用于连接时，只有发起者才能调用 connect 函数，而被动承担者则做为 connect 的参数。

7.2.2. PORT 与 IMP 的连接

在 UVM 三种端口按控制流优先级排列中，PORT 是最高优先级，IMP 是最低优先级。理所当然的，一个 PORT 可以调用 connect 函数并把 IMP 作为端口作为调用时的参数。假如有三个 component A，B 和 C，其中 C 是 A 和 B 的 parent，现在要把 A 中的 PORT 和 B 中的 IMP 连接起来，如下图所示：

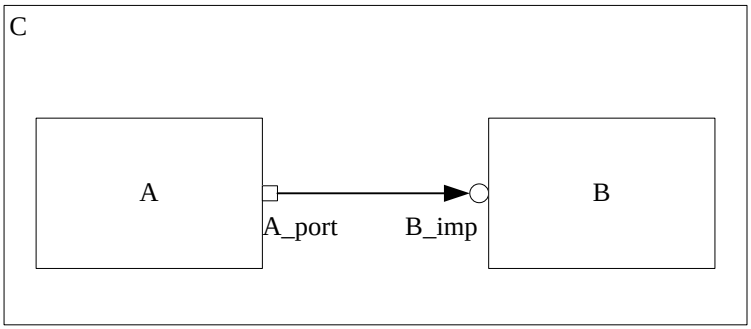


图 7-8 PORT 与 IMP 的连接

在 A 中有如下定义：

```
class A extends uvm_component;
    uvm_blocking_put_port#(mac_transaction) A_port;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        A_port = new("A_port", this);
    endfunction
endclass
```

其中 A_port 在实例化的时候比较奇怪，第一个参数是名字，而第二个参数则是一个 uvm_component 类型的 parent 变量。事实上，一个 uvm_blocking_put_port 的 new 函数如下：

```
function new(string name,
             uvm_component parent,
             int min_size = 1;
             int max_size = 1);
```

如果不看后两个参数，那么这个 new 函数其实就是一个 uvm_component 的 new 函数。这是不是暗示 port 本身也是从 uvm_component 派生而来的呢？答案是否定的。关于这一点的仔细解释，可以看本书第 17 章 TLM1.0 源代码分析。new 函数中的 min_size 和 max_size 指的是此 PORT 必须连接到的 IMP 的数量，也即这一个 PORT 可以调用多次 connect 函数，连接多个 IMP。不过如果采用默认值，即 min_size=max_size=1，那么也就只能连接一个 IMP 了。

B 中的定义如下：

```
class B extends uvm_component;
    uvm_blocking_put_imp#(mac_transaction, B) B_imp;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B_imp = new("B_imp", this);
    endfunction
    task put(mac_transaction tr);
        $display("A put a transaction to B, the transaction is");
        tr.print();
        //you could do somethis else to deal with tr.
    endtask
endclass
```

由于 A 中采用了 blocking put 的 PORT，所以在 B 中 IMP 相应的类型是 uvm_blocking_put_imp。同时，这个 IMP 有两个参数，第一个参数是将要传输的 transaction，第二个参数前面说过，就是实现接口的 uvm_component。在这里就是 B_imp 所在的 uvm_component B。IMP 的 new 函数与 PORT 的相似，第一个参数是名字，第二个参数是一个 uvm_component 的变量，一般填写 this 即可。如果 B 中只是做了这么多定义，那么当编译的时候，系统就会报错，说 B 没有一个名字为 put 的任务。回顾一下，7.1.3 节中在介绍 IMP 的时候，A_port 的 put 操作最终要落到 B 的 put 上，所以在 B 中要定义一个名字为 put 的 task。事实上，这里有如下的规律：

当 A_port 的类型是 uvm_nonblocking_put_port，B_port 的类型是 uvm_nonblocking_put _imp 时，那么就要在 B 中定义一个名字为 try_put 的任务和一个名为 can_put 的任务。

当 A_port 的类型是 uvm_put_port，B_port 的类型是 uvm_put_imp 时，那么就要在 B 中定义三个任务，一个是 put，一个是 try_put，一个是 can_put。

当 A_port 的类型是 uvm_blocking_get_port，B_port 的类型是 uvm_blocking_get_imp 时，那么就要在 B 中定义一个名字为 get 的任务。

当 A_port 的类型是 uvm_nonblocking_get_port，B_port 的类型是 uvm_nonblocking_get _imp 时，那么就要在 B 中定义一个名字为 try_get 的任务和一个名为 can_get 的任务。

当 A_port 的类型是 uvm_get_port，B_port 的类型是 uvm_get_imp 时，那么就要在 B 中定义三个任务，一个是 get，一个是 try_get，一个是 can_get。

当 A_port 的类型是 uvm_blocking_transport_port，B_port 的类型是 uvm_blocking_transport _imp 时，那么就要在 B 中定义一个名字为 transport 的任务。

当 A_port 的类型是 uvm_nonblocking_transport_port，B_port 的类型是 uvm_nonblocking_ transport_imp 时，那么就要在 B 中定义一个名字为 nb_transport 的任务。

当 A_port 的类型是 uvm_transport_port，B_port 的类型是 uvm_transport_imp 时，那么就要在 B 中定义两个任务，一个是 transport，一个是 nb_transport。

回到前面的例子中来，当 B 中定义好了 B_port 和 put 后，在 C 的 connect_phase 就需要把 A_port 和 B_port 连接在一起了：

```
class C extends uvm_component;
    A a;
    B b;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        a = A::type_id::create("a", this);
        b = B::type_id::crate("b", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        a.A_port.connect(b.B_imp);
    endfunction
endclass
```

connect 函数一定要在 connect_phase 阶段调用。连接完成后，当在 A 中往 A_port 写入一个 transaction 时，B 的 put 会马上被调用，并执行其中的代码：

```
task A::main_phase(uvm_phase phase);
    super.main_phase(phase);
    mac_transaction tr;
    for(int i = 0; i < 10; i++) begin
        tr = new;
        assert(tr.randomize());
        A_port.put(tr);
    end
endtask
```

如上所示，由于 A 往 A_port 写入了 10 个 transaction，B 的 put 会被调用 10 次。

7.2.3. EXPORT 与 IMP 的连接

PORT 可以与 IMP 想连接，同样的 EXPORT 也可以与 IMP 相连接，其连接方法与 PORT 和 IMP 的连接完全一样。

```
class A extends uvm_component;
    uvm_blocking_put_export#(mac_transaction) A_export;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        A_export = new("A_export", this);
    endfunction
endclass
```

```
class B extends uvm_component;
    uvm_blocking_put_imp#(mac_transaction, B) B_imp;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B_imp = new("B_imp", this);
    endfunction
    task put(mac_transaction tr);
        $display("A put a transaction to B, the transaction is");
        tr.print();
        //you could do somethis else to deal with tr.
    endtask
endclass
class C extends uvm_component;
    A a;
    B b;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        a = A::type_id::create("a", this);
        b = B::type_id::crate("b", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        a.A_export.connect(b.B_imp);
    endfunction
endclass
```

如上所示，就可以实现一个 EXPORT 和一个 IMP 的连接。与上一小节中的例子对比，可以发现，除了 port 变成 export 之外，其它没有任何改变。在 B 中也必须定义一个名字为 put 的任务。上一节中罗列的那些规律，对于 EXPORT 依然适用。

7.2.4. PORT 和 EXPORT 的连接

有了 PORT 和 IMP 的连接，EXPORT 和 IMP 的连接，我们很自然的期待着 PORT 和 EXPORT 能够相连接。因为在 TLM 中，PORT 和 EXPORT 相连接是贯穿始终的。即使按照我们前面总结出的规律，在 PORT，EXPORT，IMP 这三个优先级逐渐变低的排列中，前面的是可以调用 connect 函数来连接后面的。

但是很不幸的是，我们的这种期待完全错误了。读者可以试一下，如果把 7.2.2 节中，有关 imp 的都改为 export，再执行连接过程，会出现编译错误。但是，假如采用如下图的方式连接，则可以编译通过：

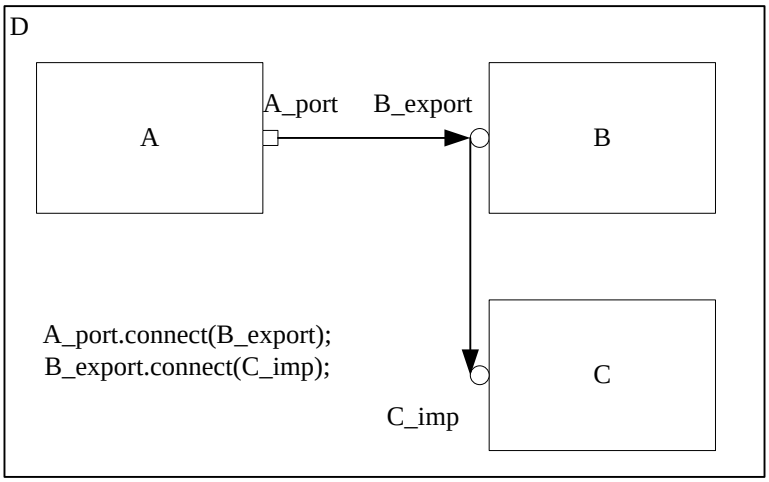


图 7-9 port、export 与 imp 的连接

这其实也就是意味着，一定要使用 IMP 来终结连接关系。PORT 和 EXPORT 都不能做为连接关系的终点。

7.2.5. UVM 中的 analysis port 和 analysis export

除了前面所说的 PORT，EXPORT，IMP 之外，事实上，UVM 还有两种常用的 port，那就 是 analysis port 和 analysis export。analysis port 和 analysis export 其实与 port 及 export 类似。都是用于传递 transaction 的。它们与 port 和 export 的区别是：

第一，默认情况下，一个 analysis port（analysis export）可以连接多个 IMP，也就是说，analysis port（analysis export）与 IMP 之间的通信是一种一对多的通信，而 PORT 和 EXPORT 与 IMP 的通信是一种一对一的通信。analysis port（analysis export）更像是一个广播。

第二，作为 PORT 和 EXPORT，有 put，get，transport 操作，虽然如前面所示，一个 PORT 要么是 put_port，要么是 get_port，要么是 transport_port，不可能是三者兼有，但是毕竟是有这三种操作。但是对于 analysis port（analysis export）来说，它只有一种操作 write。write 的意思就是广播一下，剩下的事情就与他无关了。

第三，作为 PORT 和 EXPORT，都有阻塞和非阻塞的区分。相应的 put，get，transport 操作也分成了阻塞和非阻塞的。但是对于 analysis port 和 analysis export 来说，没有阻塞和非阻塞的概念。因为它本身就是广播，不必等待与其相连的其它 port

的响应。所以不存在阻塞和非阻塞。

UVM 的这几种特性非常的实用。事实上，在验证平台上，用的最多的就是 analysis port（analysis export）。

一个 analysis port 可以和多个 IMP 相连接进行通信：

```
class A extends uvm_component;
    uvm_analysis_port#(mac_transaction) A_ap;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        A_ap = new("A_ap", this);
    endfunction
endclass
class B extends uvm_component;
    uvm_analysis_imp#(mac_transaction, B) B_imp;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B_imp = new("B_imp", this);
    endfunction
    task write(mac_transaction tr);
        $display("A write a transaction, B receives it, the transaction is");
        tr.print();
        //you could do somethis else to deal with tr.
    endtask
endclass
class D extends uvm_component;
    uvm_analysis_imp#(mac_transaction, D) D_imp;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        D_imp = new("D_imp", this);
    endfunction
    task write(mac_transaction tr);
        $display("A write a transaction, D receives it, the transaction is");
        tr.print();
        //you could do somethis else to deal with tr.
    endtask
endclass
class C extends uvm_component;
    A a;
    B b;
    D d;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        a = A::type_id::create("a", this);
        b = B::type_id::crate("b", this);
        d = D::type_id::create("d", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        a.A_ap.connect(b.B_imp);
        a.A_ap.connect(d.D_imp);
```

```
    endfunction
endclass
```

如上所示，A 中的 analysis port 与 B 和 D 的 analysis_imp 相连接。在 C 的 connect_phase 把这种连接关系建立。B 与 D 中的 IMP 的类型是 uvm_analysis_imp，是为了与 A 中的 uvm_analysis_port 相呼应。同时，由于对于一个 analysi port 有 write 操作，因此，由前面的经验，我们需要在 B 和 D 中分别写一个 write 任务。事实上，UVM 也正是这么要求的。

```
task A::main_phase(uvm_phase phase);
    super.main_phase(phase);
    mac_transaction tr;
    for(int i = 0; i < 10; i++) begin
        tr = new;
        assert(tr.randomize());
        A_ap.write(tr);
    end
endtask
```

当在 A 中向 A_ap 写入了 10 次 tr 时，B 和 D 的 write 分别被调用了 10 次。

上面只是一个 analysis port 与 IMP 相连的例子。analysis export 和 IMP 也可以这样相连接。把上面例子中的 port 改为 export 就可以。

与 PORT 和 EXPORT 直接相连会出错一样，analysis port 如果要和一个 analysis export 直接相连也会出错。只有在 analysis export 后面再连接一级 IMP，才不会出错。

7.3. 用 port 实现 monitor 和 scoreboard 的通信

7.3.1. UVM 中 port 连接时的层次关系

7.2 节主要讲述了 UVM 不同种类 port 之间的连接关系，但是没有说明同种 port 之间的连接关系。什么是同种 port 之间的连接？比如一个 uvm_blocking_put_port 可以连接 uvm_blocking_put_port，一个 uvm_analysis_port 可以连接一个 uvm_analysis_port。这种连接有意义吗？事实上，它确实是有意义的。如下图所示：（monitor，agent，scoreboard，env）

```
class monitor extends uvm_monitor;
```

```
    uvm_blocking_put_port#(mac_transaction) put_port;
    task main_phase(uvm_phase phase);
        super.main_phase(phase);
        mac_transaction tr;
        …
        put_port.put(tr);
        …
    endtask
endclass
class scoreboard extends uvm_scoreboard;
    uvm_blocking_put_imp#(mac_transaction, scoreboard) scb_imp;
    task put(mac_transaction tr);
        //do something on tr
    endtask
endclass
```

monitor 要想与 scoreboard 之间进行通信，但是由于 monitor 和 scoreboard 之间间隔了一个 agent，所以 monitor 的 put_port 无法直接连接到 scoreboard 的 scb_imp 上。

要解决这个问题有两种方法，一种方法是在 agent 里也增加一个 uvm_blocking_put_port，如 ag_put_port，然后把 monitor 的 put_port 连接到此端口上来，之后再把 ag_put_port 连接到 scoreboard 的 scb_imp 上。

这里就用到了同一种端口之间的连接。UVM 规定，孩子的 PORT 可以调用 connect 函数连接父亲的相应 PORT，父亲的 EXPORT 可以调用 connect 连接孩子的 EXPORT。因此，agent 可以这么写：

```
classs agent extends uvm_agent;
    uvm_blocking_put_port#(mac_transsaction) ag_put_port;
    …
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        ag_put_port = new(“ag_put_port”, this);
        …
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect(phase);
        monitor.put_port.connect(this.ag_put_port);
    endfunction
endclass
```

而在 env 的 connect_phase 里面可以这么写：

```
function void env::connect_phase(uvm phase);
    super.connect(phase);
    agent.ag_put_port.connect(scoreboard.scb_imp);
endfunction
```

这样就可以实现 monitor 和 scoreboard 的通信了。

另外一种方法是在 agent 中定义一个 ag_put_port，但是并不实例化它，而是令其指向 monitor 的 put_port：

```
classs agent extends uvm_agent;
    uvm_blocking_put_port#(mac_transsaction) ag_put_port;
    …
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        …
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect(phase);
        this.ag_put_port = monitor.put_port;
    endfunction
endclass
```

env 中的写法与上面相同。这样也可以实现 monitor 和 scoreboard 的通信了。

第二种方法省略了把 ag_put_port 实例化的过程，而且不用调用 connect 函数，相对简单一些。

7.3.2. 用 analysis port 实现 monitor 和 scoreboard 的通信

上面一小节说明了，使用 uvm_blocking_put_port 实现 mointor 和 scoreboard 的通信，当然了，相应的 scoreboard 中的 imp 必须是 uvm_blocking_put_imp，且必须定义一个名字为 put 的 task 来接收 monitor 传递过去的 transaction。

同样的，我们也可以使用 uvm_nonblocking_put_port，uvm_blocking_get_port，uvm_nonblocking_get_port，uvm_analysis_port 实现 monitor 和 scoreboard 的通信。通信的方法多种多样，这里仅仅说明如何用 uvm_analysis_port 实现，因为这种方式是最常用的。

在 monitor 中，我们需要定义一个 analysis port：

```
class monitor extends uvm_monitor;
    uvm_analysis_port#(mac_transaction) ap;
    task main_phase(uvm_phase phase);
        super.main_phase(phase);
        mac_transaction tr;
        …
        ap.write(tr);
        …
```

```
    endtask
endclass
```

而在 scoreboard 中，我们需要定义一个名字为 write 的 task。

```
class scoreboard extends uvm_scoreboard;
    uvm_analysis_imp#(mac_transaction, scoreboard) scb_imp;
    task write(mac_transaction tr);
        //do something on tr
    endtask
endclass
```

在 agent 中，也定义一个 analysis port，并把其指向 monitor 的 ap：

```
classs agent extends uvm_agent;
    uvm_analysis_port#(mac_transsaction) ap;
    …
    function void connect_phase(uvm_phase phase);
        super.connect(phase);
        this.ap = monitor.ap;
    endfunction
endclass
```

env 的 connect_phase 如下：

```
function void env::connect_phase(uvm phase);
    super.connect(phase);
    agent.ap.connect(scoreboard.scb_imp);
endfunction
```

可见，这种实现方式与上一小节介绍的几乎是一模一样。

7.3.3. 有多个 uvm_analysis_imp 存在的情况

上面的 monitor 和 scoreboard 之间的通信，采用一个 analysis port 和一个 anslysis imp 相连的方式实现。我们知道，对于一个 analysis imp 来说，必须在其实例化的 uvm_component 定义一个 write 的 task。在上面的例子中，scoreboard 只接收一路数据，但是现实情况中，scoreboard 除了接收 monitor 的数据之外，还要接受 reference model 的数据。相应的 scoreboard 就要再添加一个 uvm_analysis_imp 的 IMP，如 model_imp。此时问题就出现了，这个新的 IMP 也要有一个 write 任务与其对应，因为很明显，我们对接收到的两路数据的处理是不一样的。但是 write 只有一个，怎么办？

UVM 考虑到了这种情况，它采用如下的方式处理：

```
`uvm_analysis_imp_decl(_monitor)
```

```
`uvm_analysis_imp_decl(_model)
class scoreboard extends uvm_scoreboard;
    uvm_analysis_imp_monitor#(mac_transaction, scoreboard) monitor_imp;
    uvm_analysis_imp_model#(mac_transaction, scoreboard) model_imp;
    task write_monitor(mac_transaction tr);
        //do something on tr
    endtask
    task write_model(mac_transaction tr);
        //do something on tr
    endtask
endclass
```

通过宏 uvm_analysis_imp_decl，声明了两个后缀_monitor 和_model。UVM 会根据这两个后缀内建两个新的 imp：uvm_analysis_imp_monitor 和 uvm_analysis_imp_model。当与 uvm_analysis_imp_monitor 相连接的 analysis port 执行 write 任务时，会自动调用 write_monitor 任务，而与 uvm_analysis_imp_model 相连的则会自动调用 write_mondel 任务。所以，只要把后缀声明了，把 write 后面添加上相应的后缀就可以正常工作了。

7.3.4. 用 fifo 实现 monitor 和 scoreboard 的通信

上一小节中要声明两个后缀，然后再写相应的 task，这种方法看起来相当的麻烦。那么有没有简单的方法呢？另外上面的 monitor 和 scoreboard 的通信，monitor 占据主动地位，而 scoreboard 只能被动的接收。那么有没有方法也让 scoreboard 实现主动的接收呢？这两个问题的答案都是肯定的，那就是使用 fifo 来实现 monitor 和 scoreboard 的通信。

如图 7-10 所示，在 agent 和 scoreboard 之间添加一个 uvm_analysis_fifo，相应的，scoreboard 里面的端口要改成这样：

```
class scoreboard extends uvm_scoreboard;
    uvm_blocking_get_port#(mac_transaction) get_port;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    endfunction
    task main_phase(uvm_phase phase);
        mac_transaction tr;
        while(1) begin
            get_port.get(tr);
            //do something about tr
        end
    endtask
endclass
```

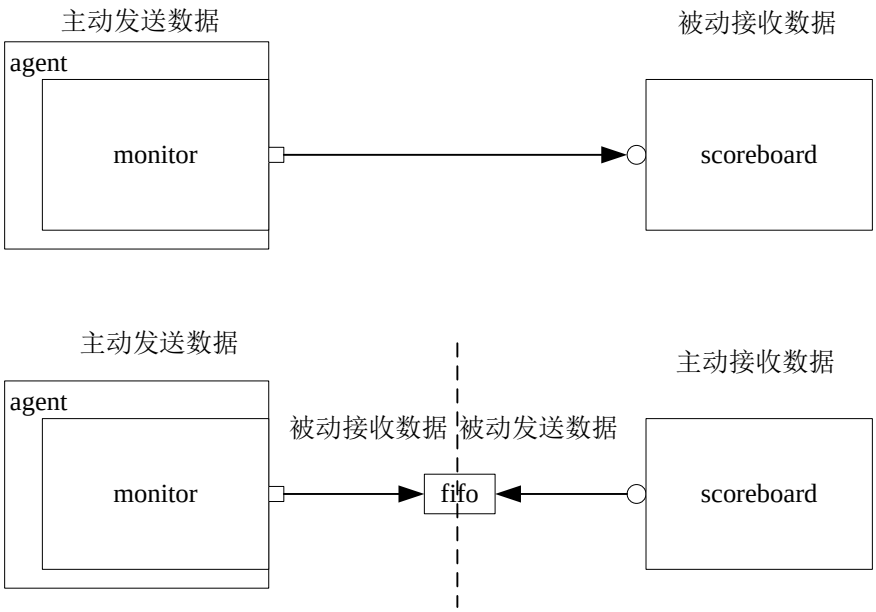


图 7-10 使用 fifo 连接 component

在 env 里面应该这样连接：

```
class env extends uvm_env;
    uvm_tlm_analysis_fifo#(mac_transaction) agent_scb_fifo;
    …
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agent_scb_fifo = new("agent_scb_fifo", this);
        …
    endfunction
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        agent.ap.connect(agent_scb_fifo.analysis_export);
        scoreboard.get_port.connect(agent_scb_fifo.blocking_get_export);
        …
    endfunction
endclass
```

而 monitor 和 agent 里面相关的代码与 7.3.2 节中的完全相同。可以看到，这样连接之后，不必在 scoreboard 中再写一个名字为 write 的任务了。scoreboard 可以按照自己的节奏工作，而不必按照 monitor 的节奏来。这种连接关系相对比较清晰。

上一小节说的当 reference model 和 monitor 同时连接到 scoreboard 此时也变得非常容易解决，只要在 scoreboard 中再声明一个 uvm_blocking_get_port，在 env 里再声明一个 fifo，然后使用 fifo 把新声明的 port 和 model 的 analysis port 连接在一起就可以了。

7.3.5. 用 fifo 还是直接用 IMP

用 fifo 还是直接用 IMP 来实现通信？

每个人对于这个问题都有各自不同的答案。除了前面说的优点之外，用 fifo 进行通信还有以下好处：在用 fifo 通信的方法中，完全的隐藏了 IMP 这个 UVM 中特有的，而 TLM 中根本就没有的东西。用户可以完全的不关心 IMP。因此，对于用户来说，只需要知道 analysis port，blocking get port 即可。这大大简化了初学者的工作量。尤其是在 scoreboard 面临多个 IMP，要给 IMP 声明一个后缀的时候，这更增加了难度。

另外，如果只是普通的端口，在 IMP 后面加一个后缀，这样是可以正常工作的。但是假如是端口数组呢？在 7.3.3 节的例子中，假如 model 中有 16 个类似端口要和 scoreboard 中相应的端口相互通信，如此多数量的端口，当然是应该使用端口数组来实现。但是如果使用给一个 IMP 声明后缀的方式，使用数组是不可能的，因为根本不知道应该声明什么样的后缀。但是使用 fifo 的方式可以完美的解决这个问题。

因此本书推荐使用 fifo 的方式来连接。如果你没有看懂本章，那么没关系，你只要看懂 7.3.4 节就 OK 了，利用这一节中讲述的内容，读者完全可以达到 UVM 验证平台中的通信要求。

8.register model 的使用

用前面几章讲述的内容，已经足可以搭建起功能强大的 UVM 验证平台了。但是搭建起来的验证平台在遇到寄存器操作时遇到了一定的问题。本章将会介绍 UVM 中的 register model 的使用。

8.1.register model 简介

8.1.1. register model 的必要性

考虑如下一个问题，当验证平台向 DUT 发了某些激励后，我们期望 DUT 中的某状态寄存器会对我们的激励有一定的反应。我们想在 scoreboard 中查看此寄存器的值是否与我们期望的值一样，应该怎么做？

就目前我们所掌握的知识来说，要查看一个寄存器的值只能通过使用 cpu_driver，向总线上发送读指令，并给出要读的寄存器地址来完成。要实现这个过程，需要启动一个 sequence，这个 sequence 会发送一个 transaction 给 cpu_driver。所以问题归结到如何在 scoreboard 的控制下来启动一个 sequence 以读取寄存器。

一个简单的想法是，设置一个全局事件（又是全局变量!），然后在 scoreboard

中触发这个事件。在 virtual sequence 中则等待这个事件的到来，等到了，则启动 sequence。这里用到了全局变量，这是我们相当忌讳的。

如果不用全局变量，那么可以用一个非全局事件来代替。利用 config 机制，分别给 virtual sequencer 和 scoreboard 设置一个 config_object，在此 object 中设置一个事件，如 rd_reg_event，然后在 scoreboard 中触发这个事件，在 virtual sequence 中则要等待这个事件的到来：

```
@p_sequencer.config_object.rd_reg_event;
```

等到了这个事件后就启动一个 sequence，开始读寄存器。

上面的这两种方法都比较麻烦。如果有了 register model，那么这个过程就可以简化为：

```
task scoreboard::main_phase(uvm_phase phase);
    …
    reg_model.STATUS_REG.read(status, value, UVM_FRONTDOOR);
    …
endtask
```

只要一句话就可以实现上述复杂的过程。像启动 sequence，并把读取结果返回这些事情，都可以由 register model 来完成。

8.1.2. register model 中一些常用的概念

uvm_reg_field：这是 register model 中最小的单位。什么是 reg field？假如有一个状态寄存器，它各个位的含义如下：

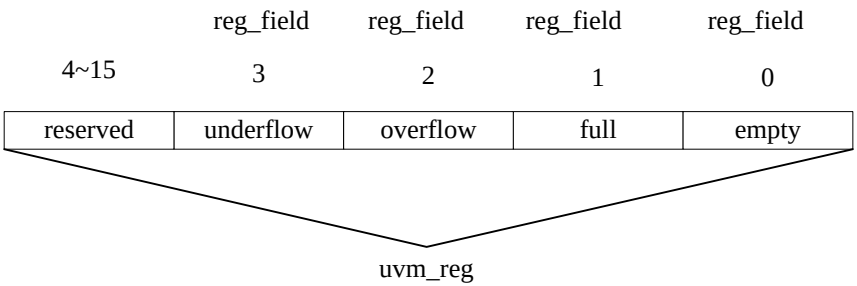


图 8-1 uvm_reg_field 和 uvm_reg

上面的这个状态寄存器一共有四个域，分别 是 empty，full，overflow，underflow。这 4 个域也就对应 register model 中的 uvm_reg_field。名字为”reserved”的并不是一个域。

uvm_reg：它比 uvm_reg_field 高一个级别，但是依然是比较小的单位。一个寄存器中至少有一个 uvm_reg_field 组成。

uvm_reg_block：它是一个比较大的单位，在其中可以加入许多的 uvm_reg，也可以加入其它的 uvm_reg_block。一个 register model 中至少包含一个 uvm_reg_block。

UVM_FRONTDOOR：它代表的是寄存器的访问方式，即通过模拟 cpu，在总线上发出读指令，进行读写操作。在这个过程中，仿真时间（不是花费的 cpu 时间，而$time 函数得到的时间）是一直往前走的。

UVM_BACKDOOR：它是与 UVM_FRONTDOOR 相对的概念。它并不通过总线进行读写操作，而是直接通过层次化的引用来改变寄存器的值。如某个寄存器的路径是：

```
top_tb.dut_inst.mac_inst.imac_inst.pre_num
```

那么在 top_tb 模块，我们可以这样给其赋值：

```
top_tb.dut_inst.mac_inst.imac_inst.pre_num = 5;
```

为什么要有 BACKDOOR 的存在？因为有时候用 FRONTDOOR 是没办法改变某些寄存器的值的。如在 DUT 中有一个写清的计数器（寄存器），当 DUT 内部发生改变时，这寄存器的值会一直增加。对于用户来说，可以对此寄存器做两种操作，一是通过总线（FRONTDOOR）来读取这个寄存器的值，二是通过写操作把这个寄存器清零。注意，是清零，而不是把一个值写入到此寄存器中。这相当于通过 FRONTDOOR 的形式只能往此寄存器中写入 0，而要写其它值是不可能的。假设现在要测试一下这个寄存器的进位功能，也即是要写入 16'hFFFF 等值，此时只有通过 BACKDOOR 的方式来进行。另外，BACKDOOR 相对 FRONTDOOR 来说，是不消耗仿真时间的（即 BACKDOOR 前后$time 的返回值不会改变），BACKDOOR 的速度比 FRONTDOOR 也快的多。

uvm_reg_map：每个寄存器都有其地址，这些地址的表现形式有的为绝对地址，有的为相对地址，uvm_reg_map 就是存储这些地址，并把这些地址转换成可以访问的地址。当 register model 使用 FRONTDOOR 方式来实现一次读或写操作时，uvm_reg_map 就会把地址转换成绝对地址，启动一个读或写的 sequence，并把读或写的结果返回。

8.1.3. register model 与 UVM 验证平台

下图示出了读取寄存器的过程，其中左图为不使用 register model，右图为使用 register model。其中红线为读取的寄存器的值。

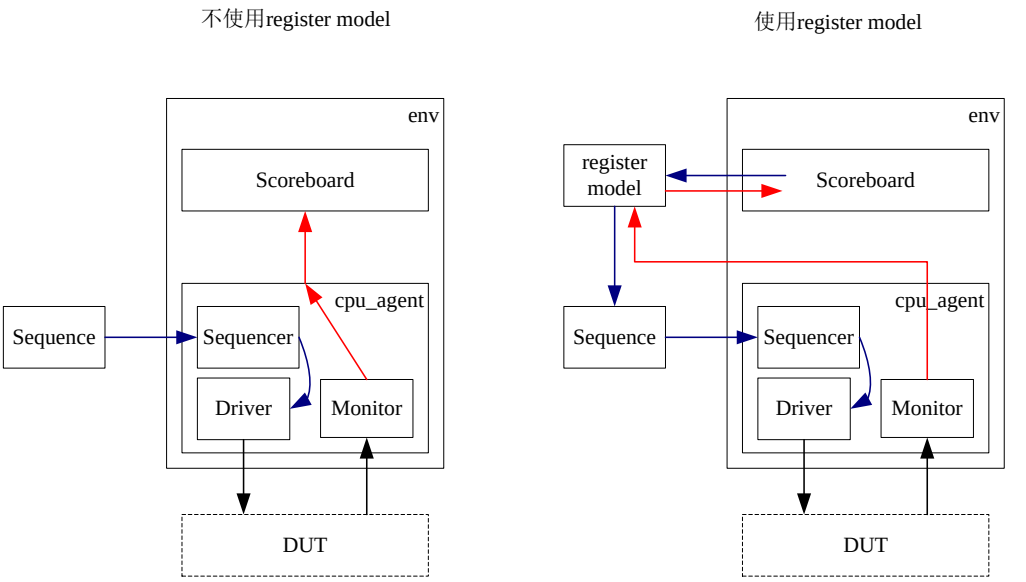


图 8-2 使用 register model 读取寄存器的流程

在没有 register model 之前，只能启动 sequence 通过 FRONTDOOR 的方式来读取寄存器，局限较大，在 scoreboard（或者其它 component）中难以控制。而有了 register model 之后，scoreboard 只与 register model 打交道，无论是发送读的指令还是读取的返回值，都可以由 register model 完成。有了 register model 后，可以在任何耗费时间的 phase 中使用 register model 以 FRONTDOOR 和 BACKDOOR 的方式来读取寄存器的值，同时还能在某些不耗费时间的 phase（如 check_phase）中使用 BACKDOOR 的方式来读取寄存器的值。

另外，register model 还提供一些任务，如 mirror，update，可以批量完成 register model 与 DUT 中相关寄存器的交互。

可见，UVM register model 的本质就是重新定义了验证平台与 DUT 的寄存器接口，让验证人员更好的组织及配置寄存器，简化流程，减少工作量。

8.2. 搭建一个简单的 register model

8.2.1. 只有一个寄存器的 register model

假设有如下的 DUT：

```
module dut (clk, data, addr, we_n, cs);
input          clk;
inout[15:0]    data;
input[15:0]    addr;
input          we_n;
input          cs;

reg [16:0] version;

initial begin
    version <= 16'h0000;
end
endmodule
```

这个 DUT 相当的简单，它只有一个寄存器 version，要为其建造 register model。首先要从 uvm_reg 派生一个通用的寄存器类：

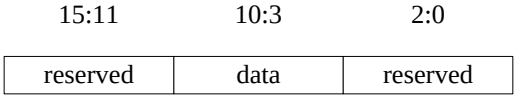
```
class my_reg extends uvm_reg;
    rand uvm_reg_field data;
    virtual function void build();
        data = uvm_reg_field::type_id::create("data");
        // parameter: parent, size, lsb_pos, access, volatile, reset value, has_reset, is_rand, indivi
dually accessible
        data.configure(this, 16, 0, "RW", 1, 0, 1, 1, 0);
    endfunction
    `uvm_object_utils(my_reg)
    function new(input string name="unnamed_my_reg");
        //parameter: name, size, has_coverage
        super.new(name, 16, UVM_NO_COVERAGE);
    endfunction
endclass
```

在new函数中，要把这个寄存器的宽度作为参数传递给super.new函数。这里的宽度并不是说这个寄存器的有效宽度，而是说这个寄存器中总共的位数[1]。如对于一个

[1] 这里其实也可以使用此寄存器的有效宽度，但是从扩展性来说，还是用此寄存器的总位数比较合理。

16 位的寄存器，其中可能只使用了 8 位，那么这里要填写的是 16，而不是 8。这个数字一般与系统总线的宽度一样。super.new中另外一个参数是是否要加入coverage的支持，这里选择即UVM_NO_COVERAGE，即不支持。

每一个派生自 uvm_reg 的类都有一个 build，这个 build 与 uvm_component 的 build_phase 并不一样，不会自动执行，需要手工调用，与 build_phase 相似的是所有的 uvm_reg_field 都在这里实例化。当 data 实例化后，要调用 data.configure 函数来配置这个字段。configure 的第一个参数就是此域（uvm_reg_field）的 parent，也即此域位于哪个寄存器中，这里当然是填写 this 了。第二个参数是此域的宽度，由于 DUT 中 version 的宽度为 16，所以这里为 16。第三个参数是此域的最低位在整个寄存器中的位置，从 0 开始计数。假如一个寄存器如图 8-3 所示，其低 3 位没有使用，高 5 位没有使用，其中只有一个字段，此字段的有效宽度为 8 位，那么在调用 configure 时，第二个参数就要填写 8，第三个参数则要填写 3，因为此 reg field 是从第 4 位开始的。



data.configure(this,8,3,…)

图 8-3 uvm_reg_field::configure 函数的参数

第四个参数表示此字段的存取方式。UVM 一共支持如下 25 种存取方式：

1、RO：读写此域都无影响

2、RW：会尽量写进去，读的话对此域无影响

3、RC：写的话无影响，读的话会清零

4、RS：写的话无影响，读的话会设置所有的位。

5、WRC：尽量写进去，读的话会清零

6、WRS：尽量写进去，读的话会设置所有的位

7、WC：写的话会清零，读的话无影响

8、WS：写的话会设置所有的位，读的话无影响

9、WSRC：写的话会设置所有的位，读的话会清零

10、WCRS：写的话会清零，读的话会设置所有的位

11、W1C：写 1 清零，写 0 无影响，读无影响

12、W1S：写 1 设置所有的位，写 0 无影响，读无影响

13、W1T：写 1 的话会翻转，写 0 的话无影响，读无影响

14、W0C：写 0 清零，写 1 无影响，读无影响

15、W0S：写 0 设置所有的位，写 1 无影响，读无影响

16、W0T：写 0 的话会翻转，写 1 的话无影响，读无影响

17、W1SRC：写 1 设置所有的位，写 0 无影响，读清零

18、W1CRS：写 1 清零，写 0 无影响，读设置所有位

19、W0SRC：写 0 设置所有的位，写 1 无影响，读清零

20、W0CRS：写 0 清零，写 1 无影响，读设置所有位

21、WO：尽可能的写，读的话会出错

22、WOC：写的话清零，读的话出错

23、WOS：写的话设置所有位，读的话会出错

24、W1：在 reset 后，第一次会尽量写进去，其它的写无影响，读的话无影响

25、WO1：在 reset 后，第一次会习题写进去，其它的无影响，读的话会出错

事实上，寄存器的种类多种多样，这 25 种有时候并不能满足用户的需求，这时候就需要自己定义寄存器的模型。

第五个参数表示是否是易失的（volatile），这个参数一般不会使用。

第六个参数表示此域上电复位后的默认值。

第七个参数表示此域是否有复位，一般的寄存器或者寄存器的域都有上电复位值，因此这里一般也填写 1。

第八个参数表示这个域是否可以随机化。这个主要是用于对寄存器进行随机写测试，如果选择了 0，那么这个域将不会随机化，而一直是复位值，否则的话将会随机出一个数值来。这一个参数当且仅当第四个参数为 RW，WRC，WRS，WO，W1，WO1 时才有效。

第九个参数表示这个域是否可以单独存取。

定义好了此通用寄存器后，我们需要在一个由 reg_block 派生的类中把其实例化：

```
class my_regmodel extends uvm_reg_block;
    rand my_reg version;
    function void build();
        default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN);
        version = my_reg::type_id::create("version", , get_full_name());
        version.configure(this, null, "version");
```

```
            version.build();
            default_map.add_reg(version, 16'h47, "RW");
        endfunction
        `uvm_object_utils(my_regmodel)
        function new(input string name="unnamed_my_regmodel");
            super.new(name, UVM_NO_COVERAGE);
        endfunction
endclass
```

同 uvm_reg 派生的类一样，每一个由 uvm_reg_block 派生的类也要定义一个 build 函数，在此函数中一般实现所有寄存器的实例化。

一个 uvm_reg_block 中一定要对应一个 uvm_reg_map，系统有一个已经声明好的 default_map，只需要在 build 中实例化，这个实例化的过程并不是直接调用 uvm_reg_map 的 new 函数，而是通过调用 uvm_reg_block 的 create_map 来实现，create_map 有众多的参数，其中第一个是名字，第二个是基地址，第三个则是系统总线的宽度，这里的单位是 byte，而不是 bit，第四个是大小端。

之后把 version 实例化，并调用 version.configure 函数。这个函数的功能主要是指定寄存器进行 BACKDOOR 操作时的路径。其第一个参数是此寄存器所在 BLOCK 的指针，这里填写 this，第二个参数是 reg_file 的指针，到现在为止还没有介绍到 reg_file 的概念，后面会介绍，这里暂时填写 null，第三个参数是此寄存器在 DUT 中的存取路径，UVM 中称为 hdl 路径。对于一个绝对路径为 a.b.c 的寄存器，这里不必填写绝对路径，只需要填写相对路径 c 即可，因此这里填写 version。当调用完 configure 时，记得要手动调用一下 verison 的 build 函数，把 version 中的域实例化。

最后一步则是把此寄存器加入到default_map中。前面我们说过，uvm_reg_map 的作用是存储所有寄存器的地址，因此必须把实例化的寄存器加入到default_map中，否则无法进行FRONTDOOR操作 [1]。add_reg函数第一个参数是要加入的指针，第二个参数是寄存器的地址，这里随机填写了一个值，第三个参数是此寄存器的存取方式。

到此为止，一个简单的 register model 已经完成。

回顾一下前面说过的 register model 中的一些常用概念。uvm_reg_field 是最小的单位，是具体的存储寄存器数值的变量，我们可以直接用这个类。uvm_reg 则是一个“空壳子”，或者用专业名词来说，它是一个纯虚类，因此是不能直接使用的，必须由其派生一个新类，在这个新类中至少加入一个 uvm_reg_field，然后这个新类才可以使用。uvm_reg_block 则是用于组织大量 uvm_reg 的一个大容器。打个比方说，uvm_reg 是一个小瓶子，其中必须装上药丸（uvm_reg_field）才有意义，这个装药丸的过程就是定义派生类的过程，而 uvm_reg_block 则是一个大盆，它中可以放许多

[1] 这句话有一个例外，那就是如果用户自定义了 FRONTDOOR，那么即使不加入到 default map 中也可以进行 FRONTDOOR 操作。但是通常情况下，不会自定义 FRONTDOOR。

小瓶子（uvm_reg），也可以放其它稍微小一点的盆(uvm_reg_block)。整个 register model 就是一个大盆（uvm_reg_block）。

8.2.2. 把 register model 集成到验证平台中

register model 的 FRONTDOOR 方式工作流程如下图所示，其中左图为读操作，右图为写操作：

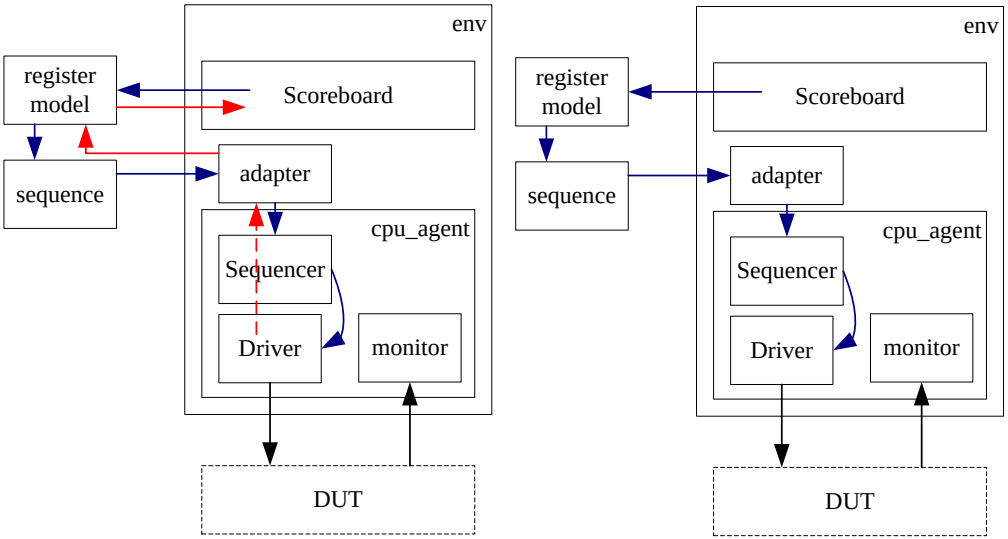


图 8-4 register model 的详细工作流程(FRONTDOOR)

register model 的 FRONTDOOR 操作可以分成读和写两种。无论是读或写，register model 都会通过 sequence 产生一个 uvm_reg_bus_op 的变量，此变量中存储着操作类型（读还是写），操作的地址，如果是写操作，还会有要写入的数据。此变量中的信息要经过一个转换器（adapter）转换之后，交给 cpu_sequencer，之后 cpu_sequencer 交给 cpu_driver，cpu_driver 实现最终的 FRONTDOOR 读写操作。因此，必须要定义好一个转换器。下面例子中列出了一个简单的转换器的代码：

```
typedef enum {CPU_RD, CPU_WR } BUS_ACC_e;
class cpu_trans extends uvm_sequence_item;
    rand bit   [15:0]        addr ;
    rand bit   [15:0]        data ;
    rand BUS_ACC_e       acc;

    `uvm_object_utils_begin(cpu_trans)
        `uvm_field_int(addr,         UVM_ALL_ON)
        `uvm_field_int(data,         UVM_ALL_ON)
```

```
        `uvm_field_enum(BUS_ACC_e, acc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
class my_adpter extends uvm_reg_adapter;
    `uvm_object_utils(cpu_adptr)
    function new(string name="my_adapter");
        super.new(name);
    endfunction : new
    function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
        cpu_trans cpu_tran;
        cpu_tran = cpu_trans::type_id::create("cpu_tran");
        cpu_tran.addr = rw.addr;
        cpu_tran.acc = (rw.kind == UVM_READ) ? CPU_RD : CPU_WR;
        if (cpu_tran.acc == CPU_WR)
            cpu_tran.data = rw.data;
        return (cpu_tran);
    endfunction : reg2bus
    function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
        cpu_trans cpu_tran;
        if(!$cast(cpu_tran, bus_item)) begin
            `uvm_fatal("adapter",
             "Provided bus_item is not of the correct type. Expecting cpu_trans")
             return;
        end
        rw.kind = (cpu_tran.acc == CPU_RD) ? UVM_READ : UVM_WRITE;
        rw.addr = cpu_tran.addr;
        rw.byte_en = 'h0;
        rw.data = cpu_tran.data;
        rw.status = UVM_IS_OK;
    endfunction : bus2reg
endclass : cpu_adptr
```

一个转换器要定义好两个函数，一是 reg2bus，其作用就是把 register model 通过 sequence 发出的 uvm_reg_bus_op 型的变量转换成 cpu_sequencer 能够接受的形式，二是 bus2reg，其作用就是当监测到总线上有操作时，把收集来的 transaction 转换成 register model 能够接受的形式，以便 register model 能够更新相应的寄存器的值。

说到这里，不得不提一下 register model 发起的读操作的数值是如何返回给 register model。如果我们的验证平台中，有一个总线的 monitor，如 cpu_monitor，那么毫无疑问，这个工作将会是 cpu_monitor 监测到读操作后，把读操作的数据封装成 cpu_trans 的形式发送出去，而一个称为 uvm_reg_predictor 的类会接收这个 transaction，并会调用转换器的 bus2reg，把 cpu_trans 转换成 uvm_reg_bus_op，register model 从后者获取读操作的数值。下图示出了 register model 读操作值返回的两种方式，其中右图示出了 uvm_reg_predictor 的工作流程：

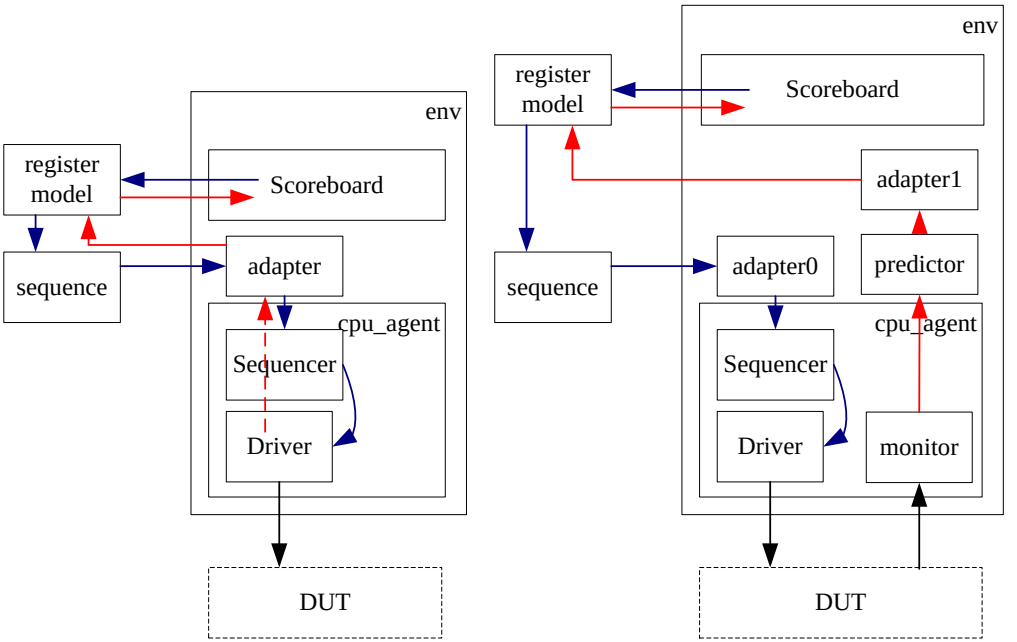


图 8-5 register model 读操作返回值的两种方式

当总线上可能有众多设备发起读写操作时，有一个 monitor 是非常有必要的。但是假如只有一个设备会使用，其实使用 monitor 是根本没有必要的。由于总线的特殊性，cpu_driver 在驱动总线进行读操作时，它也能顺便获取要读的数值，如果它把此值放入从 cpu_sequencer 获得的 cpu_trans 中时，那么 cpu_trans 中就会有读取的值，此值经过 adapter 的 bus2reg 函数，最终 register model 会收到此值。这个过程如上图中的左图所示。由于并没有实际的 transaction 的传递，所以从 driver 到 adapter 使用了虚线。

转换器写好了之后，就可以在 env 或者 test 中加入 register model 了：

```
class env extends uvm_env;
    my_regmodel reg_model;
    my_adapter   reg_sqr_adapter;
    my_adapter   mon_reg_adapter;
    uvm_reg_predictor#(iot_elcb_trans)   reg_predictor;
    …
    function void build_phase(uvm_phase phase);
        …
        reg_model = my_regmodel::type_id::create("reg_model", this);
        reg_model.configure(null, "top_tb.dut_inst");
        reg_model.build();
        reg_model.lock();
        reg_model.reset();
        reg_sqr_adapter = my_adapter::type_id::create("reg_sqr_adapter", , get_full_name());
        mon_reg_adapter  = my_adapter::type_id::create("mon_reg_adapter", , get_full_ name());
```

```
        reg_predictor = new("reg_predictor", this);
        …
    endfunction
    function void connect_phase(uvm_phase phase);
        …
        reg_model.default_map.set_sequencer(cpu_agent.cpu_sequencer, reg_sqr_adapter);
        reg_model.default_map.set_auto_predict(1);
        reg_predictor.map = reg_model.default_map;
        reg_predictor.adapter = mon_reg_adapter;
        cpu_agent.cpu_monitor.analysis_port.connect(reg_predictor.bus_in);
        …
    endfunction
    …
endclass
```

要把一个 register model 集成到 env 中，那么至少需要在 env 中定义两个成员变量，一是 reg_model，另外一个就是 reg_sqr_adapter。这种情况适用于上面说的系统中只有一个设备会读取总线时。当有多个设备时，还需要一个 reg_predictor 和一个 mon_reg_adapter，本例中用的就是这种。在 build_phase 中把所有用到的类实例化。这里要注意的是 reg_model 在实例化后还要做四件事。第一是调用 configure 函数，把 reg_model 的绝对的路径加入进去，这个路径和 reg_model 中寄存器的路径组合起来实现相关寄存器的 BACKDOOR 操作，如例子中所示，version 的完整路径将会是 top_tb.dut_inst.version。configure 的另外一个参数是 parent block。由于 reg_model 已经是最顶层的 reg_block 了，因此此处填写 null。第二是调用 build 函数，把所有的寄存器实例化。第三是调用 lock 函数，调用此函数后，reg_model 中就不能再加入新的寄存器了。第四是调用 reset 函数，如果不调用此函数，那么 reg_model 中所有寄存器的值都是不定的，调用此函数后，所有寄存器的值都将会变为设置的复位值。

前面说过，register model 的 FRONTDOOR 操作最终都将会是 uvm_reg_map 完成，因此在 connect_phase 中，需要把转换器和 cpu_sequencer 通过 set_sequencer 函数告知 reg_model 的 default_map，并把 default_map 设置为自动预测状态。最后则是把 reg_predictor 和 register model 联系起来，和转换器联系起来，和 cpu_monitor 联系起来。只有这样才能让 cpu_monitor 监测到的数据反映到 register model 中。

8.3. 复杂的 register model

8.3.1. 层次化的 register model

上节的例子中的 register model 是一个最小的，最简单的。整个实现过程中，只是把一个寄存器加入到了 uvm_reg_block 中，并在最后的 env 中例化此 reg_block。这个例子之所以这么做是因为只有一个寄存器。在现实应用中，一般的会把个 uvm_reg_block 再加入到一个 uvm_reg_block 中上，然后在 env 中例化后者。从逻辑关系上看，呈现出的是两级的 register model，如图所示：

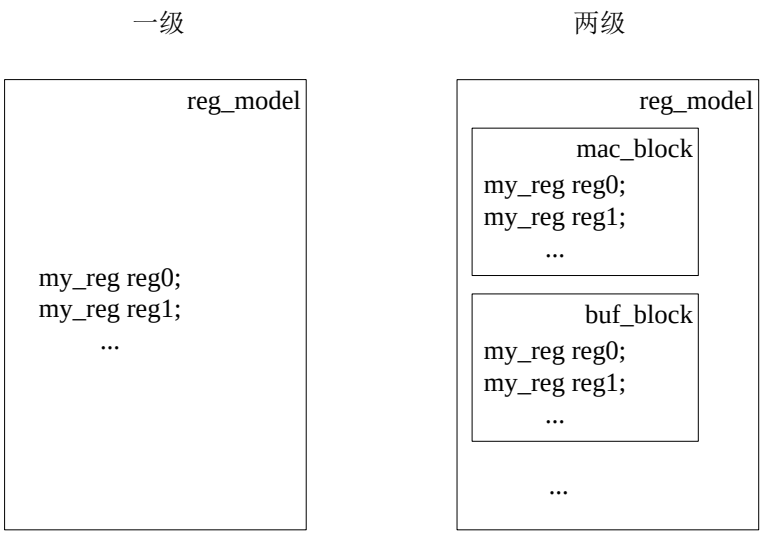


图 8-6 层次化的 register model

一般的，只会在第一级的 uvm_reg_block 中加入寄存器，而第二级的 uvm_reg_block 通常只添加 uvm_reg_block。这样从整体上来说呈现出一个比较清晰的结构。假如一个 DUT 分了三个子模块：用于缓存数据的 buf 模块，用于接收发送以太网帧的 mac 模块，buf 部分的寄存器地址为 0x1000~0x1FFF，mac 部分的为 0x2000~0x2FFF，另外全局寄存器地址为 0x0000~0x0FFF，那么可以如下定义 register model：

```
class global_blk extends uvm_reg_block;
    …
endclass
```

```
class buf_blk extends uvm_reg_block;
    …
endclass
class mac_blk extends uvm_reg_block;
    …
endclass

class register_model extends uvm_reg_block;
    global_blk gb_ins;
    buf_blk    bb_ins;
    mac_blk    mb_ins;

    function void build();
        default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN, 0);
        gb_ins = global_blk::type_id::create("gb_ins", , get_full_name());
        gb_ins.configure(this, "global_reg");
        gb_ins.build();
        gb_ins.lock_model();
        default_map.add_submap(gb_ins.default_map, 16'h0);

        bb_ins = buf_blk::type_id::create("bb_ins", , get_full_name());
        bb_ins.configure(this, "buf_reg");
        bb_ins.build();
        bb_ins.lock_model();
        default_map.add_submap(bb_ins.default_map, 16'h1000);

        mb_ins = mac_blk::type_id::create("mb_ins", , get_full_name());
        mb_ins.configure(this, "mac_reg");
        mb_ins.build(0;
        mb_ins.lock_model();
        default_map.add_submap(mb_ins.default_map, 16'h2000);
    endfunction
    …
endclass
```

要把一个子 reg_block 加入到父 reg_block 中，第一步是先实例化子 reg_block，第二步是调用子 reg_block 的 configure 函数，在这个函数中要说明这个子 reg_block 的路径，这个路径不是绝对路径，而是相对于父 reg_block 来说的路径，第三步是调用子 reg_block 的 build 函数，第四步是调用子 reg_block 的 lock_model 函数，第五步则是把子 reg_block 的 default_map 以子 map 的形式加入到父 reg_block 的 default_map 中。这是可以理解的，因为一般在子 reg_block 中定义寄存器的时候，我们给定的都是寄存器的偏移地址，其实际物理地址还要再加上一个基地址。前面说过，寄存器的 FRONTDOOR 的读写操作最终都要通过 default_map 来完成。很显然，子 reg_block 的 default_map 是不知道寄存器的基地址的，它只知道寄存器的偏移地址，只有把它加入到父 reg_block 的 default_map，同时在加入的时候告诉这个子 map 的偏移地址，这样父 reg_block 的 default_map 就可以完成 FRONTDOOR 操作。

因此，读完本节，大家可以总结出，一般的把具有同一基地址的寄存器作为整

体加入到一个 uvm_reg_block 中，而不同的基地址对应不同的 uvm_reg_block。每个 uvm_reg_block 一般都有与其对应的物理地址空间。对于本节的所说的子 reg_block，其里面还可以加入小的 reg_block，这相当于是把地址空间再次细化。

8.3.2. reg file 用以区分不同的 hdl 路径

到现在为止，前面引入了 uvm_reg_field，uvm_reg，uvm_reg_block 的概念，这三者的组合已经可以组成一个可以使用的 register model 了。然后，前面也说过了，UVM 的 register model 还有一个称为 uvm_reg_file 的概念。这个类的引入主要是用于区分不同的 hdl 路径。

假设有两个寄存器 regA 和 regB，它们的 hdl 路径为 top_tb.mac_reg.fileA.regA，top_tb.mac_inst.fileB.regB，我们延续上一节的例子，设 top_tb.mac_reg 下面所有寄存器的基地址为 0x2000，这样在第二级的 reg_block 中，加入 mac 模块的时候，其 hdl 路径要写成：

```
mb_ins.configure(this, “mac_reg”);
```

相应的，在 mac_blk 的 build 中，要通过如下方式把 regA 和 regB 的路径告知 register model：

```
regA.configure(this, null, "fileA.regA");
…
regB.configure(this, null, "fileB.regB");
```

当 fileA 中的寄存器只有一个 regA 时，这种写法是没有问题的，但是假如 fileA 中有几十个寄存器时，那么很显然，fileA.*会几十次的出现在这几十个寄存器的 configure 函数里。假如有一天，fileA 的名字忽然变为了 filea_inst，那么就需要把这几十行中所有 fileA 替换成 filea_inst，这个过程很容易出错。

为了适应这种情况，UVM 的 register model 中引入了 uvm_reg_file 的概念。uvm_reg_file 同 uvm_reg 一样是一个纯虚类，不能直接使用，必须使用其派生类：

```
class regfile extends uvm_reg_file;
    function new(input string name="unnamed_regfile");
        super.new(name);
    endfunction
    `uvm_object_utils(regfile)
endclass
class mac_blk extends uvm_reg_block;
    rand regfile file_a;
    rand regfile file_b;
    rand my_reg regA;
    rand my_reg regB;
```

```
    function void build();
        …
        file_a = regfile::type_id::create("file_a", , get_full_name());
        file_a.configure(this, null, "fileA");
        file_b = regfile::type_id::create("file_b", , get_full_name());
        file_b.configure(this, null, "fileA");
        …
        regA.configure(this, file_a, "regA");
        …
        regB.configure(this, file_b, "regB");
        …
    endfunction
endclass
```

如上所示，先从 uvm_reg_file 派生一个类，然后在 mac_blk 中实例化此类，之后调用其 configure 函数，此函数的第一个参数是其所在的 reg_block 的指针，第二个参数是假设此 reg_file 是另外一个 reg_file 的 parent，那么这里就填写其父 reg_file 的指针。这里我们只有这一级 reg_file，因此填写 null。第三个参数则是此 reg_file 的 hdl 路径。当把 reg_file 定义好了后，在调用寄存器的 configure 参数时，就可以把其第二个参数设为 reg_file 的指针。

加入了 reg_file 的概念后，当 fileA 变为 filea_inst 时，只需要把 file_a 的 configure 参数值改变一下即可，其它的不用做任何改变。这大大减少了出错的概率。

8.3.3. 具有多个域的寄存器

前面所有例子中的寄存器都是只有一个域的，如果一个寄存器有多个域时，那么在建立模型时会稍有改变。

设某个寄存器有三个域，其中最低两位为 filedA，接着 3 位为 filedB，接着 4 位为 filedC，其余位未使用。

这个寄存器从逻辑上来看是一个寄存器，但是从物理上来看，即它的 DUT 的实现中，是三个寄存器，因此这一个寄存器实际上对应着三个不同的 hdl 路径：fieldA，fieldB，fieldC。对于这种情况，前面介绍的模型建立方法已经不适用了。

```
class three_field_reg extends uvm_reg;
    rand uvm_reg_field fieldA;
    rand uvm_reg_field fieldB;
    rand uvm_reg_field fieldC;
    function void build();
        fieldA = uvm_reg_field::create("fieldA");
        fieldB = uvm_reg_field::create("fieldB");
        fieldC = uvm_reg_field::create("fieldC");
    endfunction
```

```
    `uvm_object_utils(three_field_reg)
endclass
class my_block extends uvm_reg_block;
    rand three_field_reg tf_reg;
    …
    function void build();
        …
        tf_reg = three_field_reg::type_id::create("tf_reg", , get_full_name());
        tf_reg.configure(this, null, "");
        tf_reg.build();
        tf_reg.fieldA.configure(tf_reg, 2, 0, "RW", 1, 0, 1, 1, 1);
        tf_reg.add_hdl_path_slice("fieldA", 0, 2);
        tf_reg.fieldB.configure(tf_reg, 3, 2, "RW", 1, 0, 1, 1, 1);
        tf_reg.add_hdl_path_slice("fieldB", 2, 3);
        tf_reg.fieldC.configure(tf_reg, 4, 5, "RW", 1, 0, 1, 1, 1);
        tf_reg.add_hdl_path_slice("fieldC", 5, 4);
        default_map.add_reg(tf_reg, 'h76, "RW");
        …
    endfunction
    …
endclass
```

这里要先从 uvm_reg 派生一个类，在此类中加入 3 个 uvm_reg_field。在 reg_block 中把此类实例化后，调用 rf_reg.configure 时，要注意，最后一个代表 hdl 路径的参数已经变为了空的字符串，在调用 tf_reg.build 之后要调用 tf_reg.fieldA 的 configure 函数。

在 8.2.1 节中，uvm_reg_filed 的 configure 是在所在类的 build 中被调用的，而不是在 uvm_reg_block 的 build 中调用的。这两者有什么区别呢？如果是在所定义的 uvm_reg 类中调用，那么此 uvm_reg 其实就已经定型了，不能更改了。如 8.2.1 节中定义了具有一个域的 uvm_reg 派生类，现在假如有一个新的寄存器，它也是只有一个域，但是这个域并不是如 8.2.1 中那样占据了 16bit，而只占据了低 8 位，那么此时就需要重新从 uvm_reg 派生一个类，然后再重新定义。假如 8.2.1 节中所定义的 my_reg 并没有在其 build 中调用 data 的 configure 函数，那么就不必重新定义。因为没有调用 configure 之前，这个域是不确定的。所以说，应该尽量在 uvm_reg_block 的 build 中调用 uvm_reg_field 的 configure，而不是在 uvm_reg 的 build 中调用。

调用完 fieldA 的 configure 函数后，需要把 fieldA 的 hdl 路径加入到 tf_reg 中，此时用到的函数是 add_hdl_path_slice。这个函数的第一个参数是要加入的路径，第二个参数则是此路径适用的域在此寄存器中的起始位数，如 fieldA 是从 0 开始的，而 fieldB 是从 2 开始的，第三个参数则是此路径适应的域的位宽。

8.3.4. 跨越多个地址的寄存器

实际的 DUT 中，有些寄存器会同时占据多个地址。如某 16 位的系统，其时间设置寄存器为 64 位的，即占据四个地址。

```
module dut;
    …
    reg [63:0] timer;
    …
endmodule
```

设其四个地址为 0x80~0x83，采用小端方式存储，即小地址放低位数据，大地址放高位数据。在 register model 中，为了模拟这个寄存器，有两种方法：一是把个地址拆分成四个独立的地址：

```
class my_block extends uvm_reg_block;
    rand my_reg timer0;
    rand my_reg timer1;
    rand my_reg timer2;
    rand my_reg timer3;
    …
    function void build();
        …
        timer0.configure(this, null, “timer[15:0]”);
        timer0.build();
        default_map.add_reg(timer0, ‘h80, “RW”);

        timer1.configure(this, null, “timer[31:16]”);
        timer1.build();
        default_map.add_reg(timer1, ‘h81, “RW”);

        timer2.configure(this, null, “timer[47:32]”);
        timer2.build();
        default_map.add_reg(timer2, ‘h82, “RW”);

        timer3.configure(this, null, “timer[63:48]”);
        timer3.build();
        default_map.add_reg(timer3, ‘h83, “RW”);

        …
    endfunction
    …
endclass
```

当需要读取寄存器数据时，需要分别调用 timer0~timer3 的 read（或者 peek），最终把得到的四个值组合在一起成为真正的的 timer 值。这里需要注意的是，在调用 configure 函数输入 timer 的 hdl 路径时，要把具体的位数使用中括号表示出来。

另外一种方法是如下所示：

```
class timer_reg extends uvm_reg;
      rand uvm_reg_field data;
      virtual function void build();
            data = uvm_reg_field::type_id::create("data");
            // parameter: parent, size, lsb_pos, access, volatile, reset value, has_reset, is_rand, indivi
dually accessible
            data.configure(this, 64, 0, "RW", 1, 0, 1, 1, 0);
      endfunction
      `uvm_object_utils(timer_reg)
      function new(input string name="unnamed_timer_reg");
            //parameter: name, size, has_coverage
            super.new(name, 64, UVM_NO_COVERAGE);
      endfunction
endclass
class my_block extends uvm_reg_block;
    rand timer_reg timer;
    …
    function void build();
        default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN, 0);
        …
        timer.configure(this, null, “timer”);
        timer.build();
        default_map.add_reg(timer, ‘h80, “RW”);

    endfunction
    …
endclass
```

这种方法相对简单，定义一个 timer_reg，其构造函数中指名此寄存器的大小为 64 位，此寄存器中只有一个域，此域的宽度也为 64。之后在 my_block 中把其实例化即可。在调用 default_map 的 add_reg 函数时，要指定寄存器的地址，这里只需要指明最小的一个地址即可。这是因为我们在前面实例化 default_map 时，已经指明了它使用 UVM_LITTLE_ENDIAN 形式，同时总线的宽度为 2byte，即 16bit，UVM 会自动根据这些信息计算出此寄存器占据 4 个地址。当使用 FRONTDOOR 的形式读写此寄存器时，register model 一共会进行四次读写操作，即发出四个 transaction，这四个 transaction 的地址从 0x80 一直递增到 0x83。

8.3.5. 在 register model 中加入存储器

要在 register model 中加入存储器非常容易。在一个 16 位的系统中加入一块 1024x16 的存储器的过程如下：

```
class my_memory extends uvm_mem;
```

```
    function new(string name);
        super.new(name, 1024, 16);
    endfunction
    `uvm_object_utils(my_memory)
endclass
class my_block extends uvm_reg_block;
    my_memory mm;
    …
    function void build();
        …
        mm = my_memory::type_id::create("mm", , get_full_name());
        mm.configure(this, "top_tb.stat.counter.memory");
        default_map.add_mem(mm, 'h0);
        …
    endfunction
    …
endclass
```

首先从 uvm_mem 派生一个类 my_memory，在其 new 函数中调用 super.new 函数。这个函数有三个参数，第一个是名字，第二个是存储器的所有单元的数量，第三个是每个单元的宽度，例子中加入的是一块 1024*16bits 的 memory。在 my_block 的 build 函数中，把存储器实例化，调用其 configure 函数，第一个参数是所在 reg_block 的指针，第二个参数是此块存储器的 hdl 路径。之后调用 default_map.add_mem 函数，把此块存储器的信息加入到 default_map 中，以便后面可以进行 FRONTDOOR 操作。

要对此存储器进行读写，可以通过调用 read，write，peek，poke 实现。这四个函数在调用的时候需要加入一个 offset 的参数，说明是读此取此存储器的哪个地址。

上面的存储器的宽度与系统总线位宽恰好相同。假如存储器的宽度大于系统总线位宽时，情况会略有不同。如在一个 16 位的系统中加入 512x32 的存储器：

```
class my_memory extends uvm_mem;
    function new(string name);
        super.new(name, 512, 32);
    endfunction
    `uvm_object_utils(my_memory)
endclass
```

在派生 my_memory 时，就要在其 new 函数中指明其宽度为 32，在 my_block 中加入此 memory 的方法与前面的相同。很明显，这里加入的这块存储器的一个单元占据两个物理地址，一共占据了 1024 个地址。那么当我们使用 read，write，peek，poke 时，输入的参数 offset 是代表实际的物理地址呢还是指的某一个存储单元？答案指的是存储单元。在访问这块 512x32 的存储器时，offset 的最大值是 511，而不是 1023。当指定了一个 offset，使用 FORNTDOOR 操作读写时，由于一个 offset 对应的是两个物理地址，所以 register model 会在总线上进行两次读写操作。

8.4. register model 中的常用操作

8.4.1. register model 对 DUT 寄存器的模拟

由于 DUT 中寄存器的值有可能是实时变更的，这种变更对 register model 来说，其行为能够预测，但是行为发生的时间不能确定，因此，register model 中的寄存器的值有时候与 DUT 中相关寄存器的值并不一致。对于任一个寄存器，register model 中会有一个专门的变量用于最大可能的与 register model 保持同步，这个变量在 register model 中称为 DUT 的镜像值（mirrored value）。

除了 DUT 的镜像值外，register model 中还有一个是渴望值（desired value）。如目前 DUT 中某寄存器目前的值为'h9，register model 中的镜像值也为'h9，但是我们希望往此寄存器中写入一个'h8，此时一种方法是直接调用 write 函数，把'h8 写入，另外一种方法是通过 set 函数把渴望值设置为'h8，之后调用 update 函数，update 函数会检查渴望值和镜像值是否一致，如果不一致，那么将会把渴望值写入到 DUT 中。

8.4.2. 常用操作对镜像值和渴望值的影响

read&write：使用 BACKDOOR 或 FRONTDOOR 的方式从 DUT 中读取或写入指定寄存器的值，当操作完成后，会根据读写的结果更新 register model 中的渴望值和镜像值（二者相等）。

peek&poke：peek 几乎等同于使用 BACKDOOR 方式的 read，poke 几乎等同于 BACKDOOR 方式的 write，与 read 和 write 的区别是 peek 和 poke 不会模仿寄存器的行为。如对于一个读清的寄存器来说，进行 read 操作，那么无论是 BACKDOOR 还是 FRONTDOOR，那么 DUT 中此寄存器的值在 read 操作之后都会变为 0，而 peek 则是会得到寄存器的值，但是 DUT 中寄存器的值依然保持不变。peek 和 poke 操作之后，register model 中的渴望值和镜像值都会更新（二者一致）。

get&set：这两个操作都是只针对渴望值的。只有渴望值会改变，镜像值不会改变。

randomize：randomize 之后，渴望值将会变为随机出来的数值，镜像值不会改变。

update：这个操作会检查寄存器的渴望值和镜像值是否一致，如果不一致，那么

就会变渴望值写入到 DUT 中，并且更新镜像值，使其与渴望值一致。每个由 uvm_reg 派生来的类都会有 update 操作。另外，每个由 uvm_reg_block 派生来的类也有 update 操作，它将会调用加入到此 block 中的所有寄存器的 update。

mirror：它将会读取 DUT，并且更新镜像值和渴望值。读取的方式可以选择 BACKDOOR 或者 FRONTDOOR。与 update 相类似，每个 uvm_reg 派生来的类都有 mirror 操作，每个由 uvm_reg_block 派生来的类也都有 mirror 操作。mirror 操作可以指定是否报告 DUT 中寄存器的值与 register model 中镜像值不一致。如果选择了这一项，那么就可以检查某些计数器的值是否 DUT 中相应计数器的值一致。

predict：它将会把 register model 中的镜像值和渴望值都更新为要设置的值。它与 set 的区别是后者只更新渴望值而不更新镜像值。这个操作非常有用。假如 DUT 中有一个寄存器用于统计包的数量，当收到一个包时，DUT 中的寄存器的数值加 1，而验证平台中要想同步的更新 register model 中相关寄存器的值就需要用到 predict 操作。

9.callback 的使用

在 UVM 验证平台中，callback 的最大用处就是提高验证平台的复用性。很多情况下，我们期望在一个项目中开发的验证平台能够用于另外一个项目。但是，通常来说，完全的复用是比较难实现的，两个不同的项目之间或多或少会有一些差异。如果把两个项目不同的地方使用 callback 来做，而把相同的地方写成一个完整的 env，这样复用时，env 可以完全的复用，只要改变相关的 callback 即可。

9.1. callback 简介

9.1.1. 最简单的 callback 函数

先来看一个最简单的 callback 函数。前面介绍过的 mac_transaction 为例：

```
class mac_transaction extends uvm_sequence_item;
    rand bit[47:0] dmac;
    rand bit[47:0] smac;
    rand bit[15:0] eth_type;
    rand byte      pload[];
    rand bit[31:0] crc;
```

```
    `uvm_object_utils_begin(mac_transaction)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(eth_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

这个 transaction 的最后一个字段是 crc 校验信息。这个信息必须在整个 transaction 的数据都固定之后才能计算出来。

```
mac_transaction tr;
assert(tr.randomize());
tr.calc_crc();
```

执行前两句之后，tr 中的 crc 字段的值是一个随机的值，我们要把其设置成真正的反正这个 transaction 数据的 crc 信息，需要在 randoimize()之后调用一个 calc_crc，calc_crc 是一个自定义的函数。

这个调用 calc_crc 的过程有点繁琐，因为每次 randomize 之后都要调用一次，如果有一次，忘记调用了，这很可能会成为验证平台的一个隐患，非常隐蔽，不容易发现。我们期望有一种方法，能够在 randomize 之后自动调用 calc_crc 函数。randomize 是 systemverilog 提供的一个函数，同时 systemverilog 还提供了一个 post_randomize() 函数，当 randomize()之后，系统会自动调用 post_randomize 函数，像如上的三句话，执行时实际上如下：

```
mac_transaction tr;
assert(tr.randomize());
tr.post_randomize();
tr.calc_crc();
```

其中 tr.post_randomize 是自动调用的，所以如果我们能够定义 post_randomize 函数，在其中执行 calc_crc 函数，那么就可以达到我们的目的了：

```
function void mac_transaction::post_randomize();
    super.post_randomize();
    this.calc_crc();
endfunction
```

像上面的 post_randomize 就是 systemverilog 提供的一个 callback 函数。这也是最简单的 callback 函数。

本节的这个例子似乎与本章开始时候谈的 callback 不一样。不过如果你把 systemverilog 语言的开发过程作为一个项目 A，我们自己在做的是一个项目 B，B 要用到 A 项目中的 post_randomize 函数，同时要对此函数做出一些改变。这样就可以理解 callback 的含义了。

9.1.2. callback：让一切丰富多彩

世界是丰富多彩的，而程序又是固定的。程序的设计者不是程序的使用者，所以作为程序的使用者来说，总是希望能够程序的设计者提供一些接口来满足自己的应用需求。作为这两者之间的一个协调，callback 出现了。如上面所示的例子，如果 systemverilog 的设计者一意孤行，他将会只提供 randomize 函数，此函数执行完成之后就完成任务了，不做任何事情。幸运的是，他听取了用户的意见，加入了一个 post_randomize 的 callback 函数，这样可以让用户实现各自的想法。

由上面这个例子，我们可以看出，第一，程序的开发者其实是不需要 callback 的，它完全是由程序的使用者要求的。第二，程序的开发者必须能够准确的获取使用者的需求，知道使用者希望在程序的什么地方提供 callback 接口，如果无法获取使用者的需求，那么程序的开发者只能尽可能的预测使用者的需求。

对于一个 VIP 来说，一个很容易预测到的需求是在 driver 中，在发送 transaction 之前，用户可能会针对 transaction 做某些动作，因此应该提供一个 pre_tran 的接口，如用户 A 可能在 pre_tran 中把要发送的内容的最后 4 个 byte 设置为发送的包的序号，这样在包出现比对错误的时候，可以快速的定位，B 用户可能在整个包发送之前先在线路上发送几个特殊的字节，C 用户可能把整个包的长度给截去一部分，D 用户……总之不同的用户会有不同的需求。正是 callback 的存在，满足了这种需求，扩大的 VIP 的应用范围。

9.2. UVM 中 callback 的使用

9.2.1. UVM 中的 callback

9.1.1 节中讲述了一个最简单的 callback，那是 systemverilog 中自带的 callback。我们说这个 callback 简单，因为它只牵扯到了一个类：mac_transaction。考虑如下的一个 callback：

```
task  mii_driver::main_phase();
    …
    while(1)  begin
```

```
        seq_item_port.get_next_item(req);
        pre_tran(req);
        …
    end
endtask
```

假设这是一个成熟的 VIP 中的 driver，考虑如何实现这个 pre_tran 这个 callback 呢？它应该是 mii_driver 的一个函数（任务）。如果按照上面的 post_randomize 的经验，那么我们应该从 mii_driver 派生出一个类来，然后重写 pre_tran 这个函数（任务）。这种想法是行不通的，因为这是一个完整的 VIP，我们虽然从 mii_driver 派生了一个类，但是这个这个 VIP 中正常运行时使用的依然是 mii_driver，而不是它的派生类。我们的派生类根本就没有实例化过，所以 pre_tran 从来不会运行。

为了解决这个问题，UVM 中新引入了一个类：

```
task  mii_driver::main_phase();
    …
    while(1)  begin
        seq_item_port.get_next_item(req);
        A.pre_tran(req);
        …
    end
endtask
```

这样的话，我们可以避免把 mii_driver 重新定义一次，我们只需要重新定义 A 的 pre_tran 就可以了。重新派生 A 的代价是要远小于 mii_driver 的。

在使用的时候，我们只要从 A 派生一个类，然后把这个类实例化，重新定义其 pre_tran 函数，于是 callback 的目的就达到了。看起来似乎一切顺利，但是忽略了一点。因为我们从 A 派生了一个类，把它实例化，但是作为 mii_driver 来说，怎么知道 A 派生了一个类呢？又怎么知道 A 实例化了呢？为了应付这个问题，UVM 中又引入了一个类，假设这个类称为 A_pool，意思就是专门存放 A 或者 A 的派生类的一个池子。我们约定会执行这个池子中所有实例的 pre_tran 函数（任务），即：

```
task  mii_driver::main_phase();
    …
    while(1)  begin
        seq_item_port.get_next_item(req);
        foreach(A_pool[i])  begin
            A_pool[i].pre_tran(req);
        end
        …
    end
endtask
```

这样，在使用的时候，只要从 A 派生一个类，把其实例化，并加入到 A_pool 中，那么系统运行到上面的 foreach(A_pool[i])语句时，会知道加入了一个实例，于是就会调用其 pre_tran 函数（任务）。

有了 A 和 A_pool，真正的 callback 就可以实现了。UVM 中的 callback 机制就是类似，不过其代码实现非常复杂。下面一节将会讲述具体的怎么用法。

9.2.2. pre_tran 功能的具体实现

要实现真正的 pre_tran，需要首先定义好上节所说的类 A：

```
class A extends uvm_callback;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
    endtask
endclass
```

这里要注意的是 A 类一定要从 uvm_callback 派生，另外还需要定义一个 pre_tran 的任务，此任务的类型一定要是 virtual 的，因为从 A 派生的类需要重载这个任务。

接下来定义好一个 A_pool 类：

```
typedef uvm_callbacks#(mii_driver, A) A_pool;
```

A_pool 的定义相当简单，只需要一个 typedef 语句即可。另外，在这个声明中除了要指明这是一个 A 类型的池子外，还要指明这个池子将会用于哪个类。在本例中，mii_driver 将会使用这个池子，所以要把此池子声明为 mii_driver 专用的。之后，在 mii_driver 中要做如下声明：

```
class mii_driver extends uvm_driver#(mii_transaction);
    …
    `uvm_register_cb(mii_driver, A)
endclass
```

这个声明与 A_pool 的类似，要指明 mii_driver 和 A。在 mii_driver 的 main_phase 中调用 pre_tran 时并不如上节所示的那么简单，而是调用了一个宏来实现：

```
task mii_driver::main_phase();
    …
    while(1) begin
        seq_item_port.get_next_item(req);
        `uvm_do_callbacks(mii_driver, A, pre_tran(this, req))
        …
    end
endtask
```

uvm_do_callback 宏的第一个参数是调用 pre_tran 的类的名字，这里自然是 mii_driver，第二个参数是哪个类具有 pre_tran，这里是 A，第三个参数是调用的是哪个函数（任务），这里是 pre_tran，在指明是 pre_tran 时，要顺便给出 pre_tran 的参数。

本节到现在为止是 VIP 的开发者应该做的事情，作为使用 VIP 的用户来说，需要做如下事情：

首先从 A 派生一个类：

```
class my_callback extends A;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
        $display("pre_tran, the transaction is");
        tr.print();
    endtask
    `uvm_object_utils(my_callback)
endclass
```

其次，在 base_test 中把 my_callback 实例化：

```
class base_test extends uvm_test;
    …
    my_callback my_cb;
    …
    function void connect_phase(uvm_phase phase);
        my_cb = my_callback::type_id::create("my_cb");
        A_pool::add(mii_env.agent.driver, my_cb);
        …
    endfunction
endclass
```

my_callback 的实例化是在 connect_phase 中完成的，实例化完成后需要把 my_cb 加入到 A_pool 中。同时，在加入的时候，要指定是给哪个 mii_driver 使用的。因为很可能整个 base_test 中例化了多个 mii_env，所以要把 mii_driver 的路径作为 add 函数的第一个参数。

至此，一个简单的 callback 就完成了。这个 callback 几乎是涵盖中 UVM 中所有可能用到的 callback 的知识，大部分的 callback 的使用都与这个例子相似。

9.3. callback 与 sequence 机制

9.3.1. callback 与 sequence 机制有关系吗

仔细回想一下 5.1.2 节中，我们当时在引入 sequence 机制时，曾经举过一个例子：

```
task mac_driver::main_phase(uvm_phase phase);
    mac_transaction tr;
    bit send_crc_err = 0;
    super.main_phase(phase);
    phase.raise_objection(this);
    for(int i = 0; i < 10; i++) begin
        gen_pkt(tr);
        drive_one_pkt(tr);
    end
    phase.drop_objection(this);
endtask
```

当时，我们费尽力气想要达到一个目的：在不同的 case 中调用不同的 gen_pkt 函数，我们当时怎么也想不出好的方法来，最后还是引入了 sequence 机制。现在，我们看一下 mii_driver 中的 pre_tran，是不是与这个 gen_pkt 相似呢？有了 callback 机制，我们可以在每一个 case 中分别定义一个 A 的派生类，重载 pre_tran，然后把这个派生类的实例在每个 case 的 connect_phase 中加入到 A_pool 中，这也意味着我们的可以做到在不同的 case 中调用不同的 pre_tran。有了这一层的了解，读者可以思考，是不是可以用 callback 机制实现 gen_pkt 函数，从而替代 sequence 呢？

其实，这是完全可以的。只是这么做，可能会有一些其它问题，如什么时候 raise_objection。毕竟 UVM 到现在为止是 sequence 机制的 UVM，sequence 机制是摆在 UVM 首位的，所以类似 objection 这种机制的都是与 sequence 机制配合的，而不是与 callback 机制配合的。

10. uvm_component 源代码分析

从本章开始，将进入源代码分析部分。这部分的的内容中，一般会与具体的操作相结合，如分析 register model 的一个 write 操作时，将会按照操作的流程，一点一点的讲下去，而不是把 register model 中的类一个个的讲下去，那样未免会枯燥无味。

程序=数据结构+算法。因此，在介绍源代码时，将会用很大一部分精力拿来介绍每个类的存储数据的方式。

本章第一节简析 uvm_component 的源代码。前面已经说过，component 有两大特性：每个 component 都有一个 parent 来组成 UVM 的树形结构；每个 component 都有 phase 的概念。第一节将会着重讲述第一个特性，其第二个特性将会放在第 13 章讲述。事实上，uvm_component 的源代码实现中还包含了很多为实现其它特性而写作的代码，如 factory 机制，report 机制等。这部分代码将会在介绍各自机制时详细阐述。本章第二节重点讲述了 uvm_root 的单实例实现，并且简要分析了整个 UVM 平台的启动流程。

10.1. uvm_component

10.1.1. uvm_component 的派生图

图 10-1 所示为 uvm_component 类的派生关系图。恰如前面我们已经知道的，uvm_component 派生自 uvm_object，而且并不是直接派生，在两者之间还有一个 uvm_report_object。

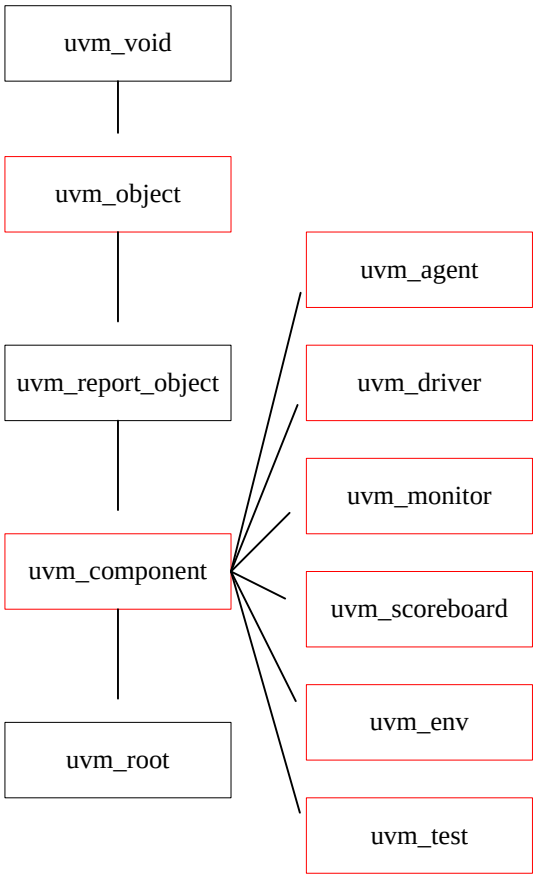


图 10-1 uvm_component 及 uvm_root 类派生图

uvm_object 是一个相对简单的类，它提供了一些基本的接口，如用于 field_automation 机制的 print，copy，pack 等，一些鉴别身份的，如 get_name，

get_type_name，get_full_name 等。其中关于 field_automation 机制的一些代码相对比较复杂。这个留在后面讲述 field_automation 机制时再回来仔细研究。

uvm_report_object 用于提供 UVM 中信息报告机制的一系列接口，如 uvm_report_error 等。由于这里只是一些接口，并无实质性的内容，因此不多做介绍。在后面将会专门拿出一章来介绍这种信息报告机制。

无论是 uvm_object 还是 uvm_report_object，更多的只是提供一些接口，而相对来说，uvm_component 里面的东西则有更具有实质性意义。对于 uvm_component 来说，有两大特点，一是它是一种树形组织结构中的一个结点，在其 new 的时候都要指定一个 parent，二是它有 phase 的概念。关于 phase 的部分将会放在 phase 机制中讲述，所以在本章中重点讲述 uvm_component 的树形组织结构。

10.1.2. 为什么要指定一个 parent

在每个 uvm_component 实例化时，都要指定一个 uvm_component 类型的 parent 变量。从字面意思理解，parent 指的是父母。每一个 uvm_component 在 new 的时候都需要指定它的父母？是的。曾经有个初学者对这个问题非常不解，为什么要指定一个 parent 呢？假设我们有一个 A 派生自 uvm_component，在 A 中有一个 uvm_component B 的成员变量，即 A 的定义如下：

```
class A extends uvm_component;
    uvm_component B;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B = new("B", this);
    endfunction
endclass
```

天然的，我们会有一种想法，那就是既然 B 是 A 的成员变量，那么很明显，A 就是 B 的 parent 了，就不用在 new 的时候指定了。即 B 的在实例化时可以这样写：

```
B = new("B");
```

这种看法看似可行，其实忽略了一点，B 是 A 的成员变量，那么在 systemverilog 仿真器一级，这种关系是确定的，可知的。假定我们有下面的类：

```
class C extends uvm_component;
    A A_INST;
    function test();
        …
        A_INST.B = …;
        A_INST.C = …;
        …
```

```
    endfunction
endclass
```

我们可以在 C 类的 test 函数中使用 A_INST.B 来得到 B 的值或者给 B 赋值，但是我们不能用 A_INST.C 来给 C 赋值。因为 C 根本就不存在于 A 里面。systemverilog 仿真器会检测这种成员变量的从属关系。但是关键问题是它即使检测到了后，它也不会告诉 A：你有一个成员变量 B，没有一个成员变量 C。A 是属于用户写出来的代码，仿真器只负责检查这些代码的合理性，它不会主动的发消息给代码，所以 A 根本就没有办法知道自己有这么一个孩子。

换个角度来说，如果在 test 中想到得 A 中所有的孩子的指针，应该怎么办？你可能会说，因为 A 是我们自己写出的，它就只有一个孩子，并且孩子的名字叫 B，所以我们可以直接使用 A_INST.B 就可以了。问题是，假设我们要把整棵 UVM 树给遍历一下，即要找到每个结点及结点的孩子的指针，那如何写呢？似乎根本就没有办法实现。你可能也会问：我们有遍历的必要性吗？有，当然有，比如我们要遍历一下，看看整棵 UVM 树中有多少个结点；比如我们要遍历下，打印出整棵树的拓扑结构。用户的需求是无止境的，所以做为一个库来说，提供树的遍历功能是必须的。

那怎么办？很简单，当 B 在实例化的时候，指定一个 parent 的变量，同时在每一个 component 的内部维护一个数组 m_children，当 B 实例化时，就把 B 的指针加入到 A 的 m_children 数组中。只有这样才能让 A 知道 B 是自己的孩子，同时也才能让 B 知道 A 是自己的父母。当 B 有了自己的孩子时，那么就在 B 的 m_children 中加入孩子的指针。

10.1.3. uvm_component 的树形组织结构的实现

在 uvm_component 的定义中，有两个联合数组是最关键的：

```
文件：src/base/uvm_component.svh
类：uvm_component
1621    protected       uvm_component  m_children[string];
1622    protected       uvm_component  m_children_by_handle[uvm_component];
```

这两个联合数组实现了 UVM 的树形组织结构。m_children 联合数组的索引是 string 类型的，其存储的内容则是 uvm_component 类型的。m_children_by_handle 的索引是 uvm_component 类型的，存储的内容是 uvm_component 类型的。

另外，还有一个 uvm_component 类型的成员变量 m_parent 用于存储这个 component 的父亲的指针：

```
文件：src/base/uvm_component.svh
```

```
类：uvm_component
1620    /*protected*/ uvm_component m_parent;
```

m_parent，m_children 和 m_children_by_handle 三个成员变量实现了 UVM 的树形结构。

在 uvm_component::new 中有如下语句：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：new

1700 function uvm_component::new (string name, uvm_component parent);
        …
1763    m_parent = parent;
        …

1767    if (!m_parent.m_add_child(this))
1768      m_parent = null;
        …
1790 endfunction
```

在每一个 uvm_component 实例化时，都要求输入一个 uvm_component 类型的 parent 变量，作为整个 UVM 树中此结点的父节点。new 函数又会调用 parent.m_add_child 函数，由于 parent 是一个 uvm_component 类型的变量，这相当于调用了 uvm_component 的 m_add_child 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：m_add_child

1796 function bit uvm_component::m_add_child(uvm_component child);
1797
1798    if (m_children.exists(child.get_name()) &&
1799        m_children[child.get_name()] != child) begin
1800        `uvm_warning("BDCLD",
1801          $sformatf("A child with the name '%0s' (type=%0s) already exists.",
1802              child.get_name(), m_children[child.get_name()].get_type_name()))
1803        return 0;
1804    end
1805
1806    if (m_children_by_handle.exists(child)) begin
1807        `uvm_warning("BDCHLD",
1808          $sformatf("A child with the name '%0s' %0s %0s'",
1809                    child.get_name(),
1810                    "already exists in parent under name '",
1811                    m_children_by_handle[child].get_name()))
1812        return 0;
1813      end
1814
```

```
1815     m_children[child.get_name()] = child;
1816     m_children_by_handle[child] = child;
1817     return 1;
1818  endfunction
```

先跳过两个 if 语句，看关于两个联合数组的操作。在 m_children 中插入了一条记录，此条记录的索引是 child.get_name。这个会返回什么呢？

```
class A extends uvm_component;
    uvm_component B;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        B = new("B", this);
    endfunction
endclass
```

如上面的例子，在调用 B = new("B", this)时，传入的名字是 B，那么对于 A 来说，child.get_name 就相当于是"B"，所以对于 A 来说，m_children 中相当加入了这么一条记录：

```
m_children["B"] = B;
```

把 B 的指针存入了 A 的 m_children 数组里，此条记录的索引是 B 的名字"B"。对于 m_children_by_handle 则更加简单，其索引和内容都是 B 的指针。对于一棵比较完整的树，如下所示，假如 A 是整棵树的根，而 D 和 E 是整棵树的叶子，那么将会有如下的数据结构：

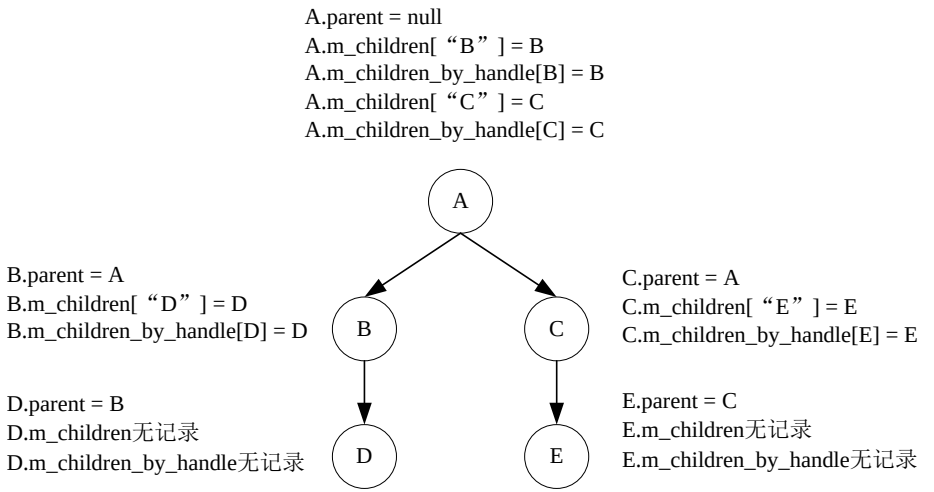


图 10-2 uvm_component 树形结构的代码实现

对于一棵完整的uvm_component树，由于根节点没有parent，所以根节点的parent的值为 null，而对于叶子节点来说，它没有任何的孩子，所以其 m_children 和

m_children_by_handle 中没有任何记录，是两个空的联合数组。

10.2. uvm_root

10.2.1. uvm_root 的应用

uvm_root是UVM中比较特殊的一个类。它的特殊之处就在于，在整个UVM验证平台中，有且只有uvm_root的一个实例存在 [1]，通常我们把这个实例叫做top。在很多地方会用到这个实例的指针。如在每个uvm_component的new中有如下语句：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：new

1700 function uvm_component::new (string name, uvm_component parent);
1701   string error_str;
1702   uvm_root top;
       …
1712   top = uvm_root::get();
       …
1790 endfunction
```

我们只从名字就可以知道，uvm_root::get();就是得到系统中唯一的一个 uvm_root 实例的指针。关于这个 get 函数，下一节会详细讲述。uvm_component 的两大机制：树形组织结构和 phase 机制的实施都离不开 uvm_root。所以要说重要程度，可能 uvm_root 的重要性要大于 uvm_component。事实上，从图 10-1 中可以看出 uvm_root 是从 uvm_component 派生而来的，而且 uvm_root 是单实例的。下面一节将会分析 uvm_root 的单实例实现。

[1] 设计模式中鼎鼎大名的 singleton 单态模式。

10.2.2. uvm_root 的单实例实现

考虑如下一段代码，假设是由我们自己写 uvm_root：

```
class A;
  extern function new();
endclass
const A a;
a=new();
```

这样可以生成 A 的一个普通实例，而且也可以在代码的其它地方生成 A 的实例。但是假设构造函数 new 变成如下格式：

```
class A;
  extern proctected function new();
endclass
const A a;
a=new();
```

那么在执行上面最后一行时，编译器会报错说 new()是 proctected 类型的，不能在外部被调用。一个 proctected 类型的函数，意味着它要么只能被它本身的成员变量或者成员函数调用，要么被由它派生而来的类的成员变量或者成员函数调用。所以，在外部调用是不可能的。

那么在这种情况下怎么样才能实例化一个 A 呢？虽然外部不能调用，但是 A 的成员函数是可以调用 new()的。只是问题是一般的成员函数都是在类的实例化后才能调用，此时类的实例已经产生了，再调用 new()有什么意思？所以说，普通的成员函数是不可能实现的。要使用静态成员函数。

什么是静态函数？在 systemverilog 中，函数分为两类，一类是静态的，一类是非静态的。如下所示，是定义了一个普通的非静态函数：

```
class A;
    function void non_static_function();
        …
    endfunction
endclass
```

对于一个非静态函数，只有当 A 实例化之后才能被调用：

```
class B;
    A a;
    function void test();
        a = new();
        a.non_static_function();
    endfunction
endclass
```

而对于非静态函数，并不需要一个类实例化就能调用，而且其调用的方式与非静态函数不一样。非静态函数是用一个点号(.)来调用，而静态函数是用双冒号来调用(::)：

```
class C;
    static function void static_function();
        …
    endfunction
endclass
class D;
    function void test();
        C::static_function();
    endfunction
endclass
```

有了静态函数的概念，可以使用如下的方式来解决我们的问题：

```
class A;
  extern proctected function new();
  extern static function A get();
endclass
function A A::get();
  A m_inst;
  m_inst=new();
  return m_inst;
endfunction
const A a=A::get();
```

到现在为止，解决了 new 的 protected 类型问题。但是这样可以生成无限多个 A 的实例，这个跟刚用普通的构造函数是一样的，为了解决这个问题，达到只有一个实例的目标，我们可以这样约定：

第一，只有在 get 函数中才调用 A 的构造函数 new，其它函数中都不能调用。也就是说，要想得到 A 的实例，只能通过调用 get 得到，调用其它函数是根本不可能得到的。

第二，可以记录 get 被调用的次数，如果这是第一次调用，那么就执行 A 的 new 函数，如果是第二次或者更多次的调用，那么就把第一次通过 new 出来的实例给返回。

通过以上两点，就可以让系统中唯一的存在一个 A 的实例。第一点是相当容易实现的。对于第二点，由于要记录 get 被调用的次数，可以考虑在 A 中增加一个成员变量用于完成这个功能。很显然，普通的变量是无法满足要求的，因为普通成员变量是与具体的类的实例相关的，必须使用静态成员变量，它不依赖于具体的类的实例：

```
class A;
  static bit first_called = 0;
  extern proctected function new();
```

```
   extern static function A get();
endclass
function A A::get();
   A m_inst;
   if(first_called==0) begin
      m_inst=new();
      first_called=1;
   end
   return m_inst;
endfunction
const A a=A::get();
```

看看上面这个 get 函数？它实现了我们的功能了吗？没有！因为每次进入 get 的时候都要重新声明一次 m_inst，当第一次调用 get 的时候，会调用构造函数 new 把 m_inst 实例化，但是当第二次调用的时候，由于 first_called 已经被赋值为 1，所以会直接返回 m_inst，而此时 m_inst 还等于 null。要解决这个问题，需要把 m_inst 声明成 A 类的一个静态成员变量：

```
class A;
   static bit first_called = 0;
   static A m_inst;
   extern proctected function new();
   extern static function A get();
endclass
function A A::get();
   if(first_called==0) begin
      m_inst=new();
      first_called=1;
   end
   return m_inst;
endfunction
const A a=A::get();
```

经过上面的步骤之后，我们的整个系统中只有一个 A 的实例目标达到了。实际中 uvm_root 的代码与上面相似，只是少了一个类似 first_called 的变量：

```
文件：src/base/uvm_root.svh
类：uvm_root

 68 class uvm_root extends uvm_component;
 69
 70    extern static function uvm_root get();
       …
165    extern `_protected function new ();
       …
179    static local uvm_root m_inst;
       …
220 endclass

231 const uvm_root uvm_top = uvm_root::get();
```

```
269 function uvm_root uvm_root::get();
270   if (m_inst == null) begin
271     m_inst = new();
272     void'(uvm_domain::get_common_domain());
273     m_inst.m_domain = uvm_domain::get_uvm_domain();
274   end
275   return m_inst;
276 endfunction
```

为什么这里不需要一个 first_called 的变量呢？因为 m_inst 就完全可以完成 first_called 的功能。第一次调用 get 的时候，m_inst 是等于 null 的，而后面调用的时候，已经不等于 null 了，所以可以通过判断 m_inst 的值来判断是不是第一次调用 get。

这段代码中的 m_inst 是 local 类型的，这是一种良好的编程风格。另外，这里的 new 函数是一个`_protected 类型的，这个宏的定义位于 uvm_macros.svh 文件中：

```
文件：src/uvm_macros.svh
类：无
// Default settings
 28 `define _protected protected
```

另外，在这个文件中还有一些与编译器相关的关于这个宏的定义，后面将会逐渐的涉及，这里暂且不必理会。可见_protected 其实就是 protected。

10.2.3. 回顾 uvm_component 的 new 函数

在 10.1.3 节中，我们曾经通过分析 uvm_component::new 函数，得到了 UVM 树形结构的建立方式。当时我们只说明了树的主体部分是如何建立的，但是对于整棵树的根节点是如何建立的则没有说明。

uvm_root 是整棵 UVM 树的根节点，其 new 函数中有如下语句：

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：new

282 function uvm_root::new();
      …
286   super.new("__top__", null);
      …
298 endfunction
```

这个 new 函数不需要任何参数，它调用 super.new，传入的 name 参数为__top__，而 parent 参数为 null。

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：new

1700 function uvm_component::new (string name, uvm_component parent);
        …
1704    super.new(name);
1705
1706    // If uvm_top, reset name to "" so it doesn't show in full paths then return
1707    if (parent==null && name == "__top__") begin
1708       set_name(""); // *** VIRTUAL
1709       return;
1710    end
        …
1790 endfunction
```

在 uvm_component 的 new 函数中，当发现传入的参数是 null 和__top__的时候，就知道这是由 uvm_root 调用的，因此直接返回。在这个过程中，并没有给 m_parent 赋值，所以 m_parent 的值为 null。另外，也并没有调用 m_parent.m_add_child 函数，因为 parent 等于 null，调用 m_add_child 也是无意义的。

前面说过，当一个 uvm_component 的变量在实例化的时候必须传入一个 parent 变量，但是假如传入的 parent 变量为 null 的时候会怎么样呢？

```
class A extends uvm_component;
   function new(string name);
      super.new(name, null);
   endfunction
endclass
```

这种情况下还怎么保证整个系统中只有一棵树呢？

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：new

1700 function uvm_component::new (string name, uvm_component parent);
1701    string error_str;
1702    uvm_root top;
        …
1712    top = uvm_root::get();
        …
1741    if (parent == null)
1742       parent = top;
        …
1763    m_parent = parent;
        …
1790 endfunction
```

可见，如下图所示，如果一个 component 在实例化时，其 parent 被设置为 null，

那么这个 component 的实例的 parent 将会被系统设置为系统中唯一的 uvm_root 的实例 uvm_top，从而保证整个系统中只有一棵树，所以树的结点都是 uvm_top 的子结点。

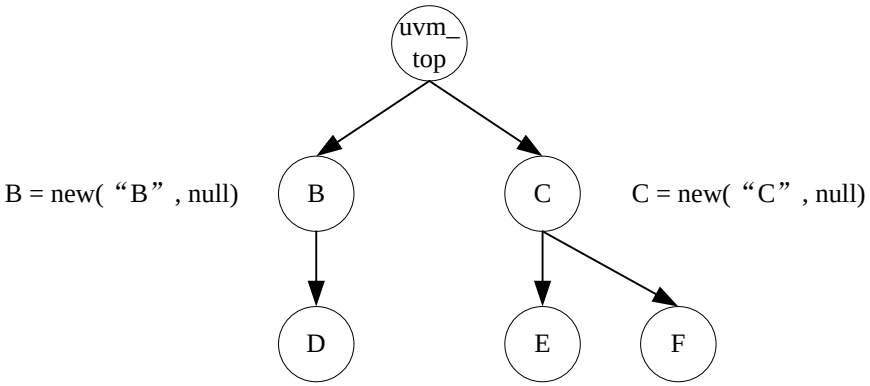


图 10-3 实例化时 parent 设置为 null 的 uvm_component

10.2.4. run_test 函数

在第一章时曾经分析过，在系统启动的时候，仿真器会调用一个全局的函数 run_test，而 run_test 会调用 uvm_root 的 run_test，下面用我们到现在为止的源代码知识简单分析一下 run_test 的执行过程：

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

304 task uvm_root::run_test(string test_name="");
305
306   uvm_factory factory= uvm_factory::get();
307   bit testname_plusarg;
308   int test_name_count;
309   string test_names[$];
310   string msg;
311   uvm_component uvm_test_top;
312
313   process phase_runner_proc; // store thread forked below for final cleanup
314
315   testname_plusarg = 0;
316
323   uvm_objection::m_init_objections();
324
325 `ifndef UVM_NO_DPI
```

```
326
327    // Retrieve the test names provided on the command line.  Command line
328    // overrides the argument.
329    test_name_count = clp.get_arg_values("+UVM_TESTNAME=", test_names);
330
331    // If at least one, use first in queue.
332    if (test_name_count > 0) begin
333      test_name = test_names[0];
334      testname_plusarg = 1;
335    end
336
337    // If multiple, provided the warning giving the number, which one will be
338    // used and the complete list.
339    if (test_name_count > 1) begin
340      string test_list;
341      string sep;
342      for (int i = 0; i < test_names.size(); i++) begin
343        if (i != 0)
344          sep = ", ";
345        test_list = {test_list, sep, test_names[i]};
346      end
347      uvm_report_warning("MULTTST",
348        $sformatf("Multiple (%0d) +UVM_TESTNAME arguments provided on the command
line.  '%s' will be used.  Provided list: %s.", test_name_count, test_n      ame, test_list), UVM_N
ONE);
349    end
350
```

第 323 行用于初始化 objection 机制中的相关参数，这里暂且不必理会。第 325 行，根据 UVM_NO_DPI 宏是否定义，系统呈现两个分支。从名字可以看出来，就是系统中是否不使用 DPI。一般说来，都不会定义这个宏。所以执行第一个分支。从 327 一直到 349 行，这段代码的意思就是得到命令行中+UVM_TESTNAME=后面所跟字符串。在运行 UVM 验证平台时，会通过+UVM_TESTNAME=casename 来指定运行哪个 case，所以这几行代码就是为了得到要运行的 case 的名字。由于使用者可能多次输入+UVM_TESTNAME=casename，所以这里要考虑把多出来的 casename 给去掉，只保留第一次碰到的+UVM_TESTNAME 中所指定的 case。这段代码用到了 uvm_cmdline_processsor 中的相关函数。我们没有接触过这个类，不过幸运的是，从其函数名字中我们就可以知道这个函数做了什么事情。这个类相对简单，后面会有专门的章节介绍，因此这里不多阐述。

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

351 `else
352
353       // plusarg overrides argument
354    if ($value$plusargs("UVM_TESTNAME=%s", test_name)) begin
```

```
355       `uvm_info("NO_DPI_TSTNAME", "UVM_NO_DPI defined--getting UVM_TESTNAME
directly, without DPI", UVM_NONE)
356       testname_plusarg = 1;
357    end
358
359 `endif
```

351 行到 359 行相对是当系统中定义了 UVM_NO_DPI 时所走的一个分支。这个分支其实做的事情与上面的分支相似，只是换了另外一种方式来实现。

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

361    // if test now defined, create it using common factory
362    if (test_name != "") begin
363       if(m_children.exists("uvm_test_top")) begin
364          uvm_report_fatal("TTINST",
365                "An uvm_test_top already exists via a previous call to run_test", UVM_NONE);
366          #0; // forces shutdown because $finish is forked
367       end
368       $cast(uvm_test_top, factory.create_component_by_name(test_name,
369             "", "uvm_test_top", null));
370
371       if (uvm_test_top == null) begin
372          msg = testname_plusarg ? {"command line +UVM_TESTNAME=",test_name} :
373                                   {"call to run_test(",test_name,")"};
374          uvm_report_fatal("INVTST",
375             {"Requested test from ",msg, " not found." }, UVM_NONE);
376       end
377    end
```

362 到 377 行则是用于创建指定的 case 的实例。这里首先判断一下是不是已经创建了这样的一个实例，正常情况是不会出现的，只有当 UVM 出错的时候才会走 364 行的分支。整个 run_test 关键的部分在于 368 和 369 行的 cast 函数中调用的 factory.create_component_ by_name，这个函数将会根据输入的 case 的名字来创建这个 case 的一个实例。这用到了 factory 机制，后面的章节将会详细介绍。371 行 376 用于判断实例创建是否成功，如果没有成功，说明这个 case 根本没有在 factory 中注册过，会给出出错提示。一般的，当我们在指定 case 的时候如果输错了名字就会给出这个错误提示。

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

379    if (m_children.num() == 0) begin
380       uvm_report_fatal("NOCOMP",
381             {"No components instantiated. You must either instantiate",
382              " at least one component before calling run_test or use",
```

```
383                 " run_test to do so. To run a test using run_test,",
384                 " use +UVM_TESTNAME or supply the test name in",
385                 " the argument to run_test(). Exiting simulation."}, UVM_NONE);
386       return;
387     end
388
389     uvm_report_info("RNTST", {"Running test ",test_name, "..."}, UVM_LOW);
```

379 到 387 行做的事情也是检测系统中是否创建了一个 case 的实例，这点与 371 到 376 类似。由于 371 行到 376 行是在 362 行的 if 分支里面，所以只用于判断输入的 case 的名字不等于"'时创建 case 的实例是否成功。当输入的 case 为一个空字符串时，说明根本没有指定 case 名字，自然也就不会创建 case 的实例。那么为什么可以根据 m_children 中记录的数量来判断是不是已经创建了一个实例呢？回顾 9.1.3 节，在 uvm_component 的 new 函数中，当一个 uvm_component 实例化的时候，会在其 parent 的 m_children 中插入一条记录。所以当有一个 uvm_component 被新创建的时候，如果其 parent 为 null，那么就会在 uvm_root 的 m_children 中插入一条记录，如果其 parent 不为 null，那么这个 parent 可能是 uvm_top，也可能是其它的，设为 A。如果是 uvm_top，那么自然会在 uvm_top.m_children 中插入一条记录，如果是 A 的话，那么可以想像，在 A 实例化的时候，或者在 A 的 parent 及其更早的祖先实例化的时候，会在 uvm_top.m_children 中插入一条记录。因此，由于 uvm_root 的单实例特性，可以根据其 m_children 中记录的数量来判断是不是有 case 实例化。

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

391     // phase runner, isolated from calling process
392     fork begin
393       // spawn the phase runner task
394       phase_runner_proc = process::self();
395       uvm_phase::m_run_phases();
396     end
397     join_none
398     #0; // let the phase runner start
399
400     wait (m_phase_all_done == 1);
401
402     // clean up after ourselves
403     phase_runner_proc.kill();
404
405     report_summarize();
406
407     if (finish_on_completion)
408       $finish;
409
410 endtask
```

391 行到最后，则是与 phase 机制相关的，这个在介绍后面的 phase 机制的时候

会仔细阐述。

11. report 机制源代码分析

UVM 中的信息报告机制相对来说比较简单，不过其作用却比较大，主要用于打印信息等。本章第一节以 uvm_error 宏为例来进行分行，第二节分析 uvm_report_server 类的源代码。

11.1. `uvm_error 宏的执行

一般的，我们会使用 uvm_error 宏来说明一些错误信息，如在一个 env.agent.driver 中：

```
`uvm_error("driver", "the config object is null")
```

这个宏的展开如下：

```
文件：src/macros/uvm_message_defines.svh
类：无

140 `define uvm_error(ID,MSG) \
141     begin \
142       if (uvm_report_enabled(UVM_NONE,UVM_ERROR,ID)) \
143         uvm_report_error (ID, MSG, UVM_NONE, `uvm_file, `uvm_line); \
144     end
```

11.1.1. uvm_report_enabled

这里会首先调用 uvm_report_enabled 函数，这个函数有两个版本，假如是在一个 component 中调用 uvm_error 宏的话，那么使用的是这个 component 自己的函数，而这个函数是在 uvm_report_object 中定义的：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object
函数/任务：uvm_report_enabled

447    function int uvm_report_enabled(int verbosity,
448                                    uvm_severity severity=UVM_INFO, string id="");
449      if (get_report_verbosity_level(severity, id) < verbosity ||
450          get_report_action(severity,id) == uvm_action'(UVM_NO_ACTION))
451        return 0;
452      else
453        return 1;
454    endfunction
```

假如是在一个 sequence 中或者是一个派生自 uvm_object 的类中使用这个宏的话，调用的是全局的函数：

```
文件：src/base/uvm_globalst.svh
类：无，为全局函数
函数/任务：uvm_report_enabled

115 function bit uvm_report_enabled (int verbosity,
116                                   uvm_severity severity=UVM_INFO, string id="");
117   uvm_root top;
118   top = uvm_root::get();
119   return top.uvm_report_enabled(verbosity,severity,id);
120 endfunction
```

这实际上是调用了 uvm_root 的 uvm_report_enabled 函数，也即 uvm_report_object::uvm_ report_enabled 函数。这里有一个 ID 的概念。对于同一个 component，可能会有多个记录信息，如在 main_phase 中我们可能想使用 main 作为 ID，在 shutdown_phase 中使用 shutdown 作为 ID。在我们的例子中，是使用”driver”作为 ID。

get_report_verbosity_level 是 uvm_report_object 定义的一个函数，它的作用是返回当前 component 的报告作息冗余级别，UVM 中共有如下几种冗余级别：

```
文件：src/base/uvm_object_globals.svh
类：无

304 typedef enum
305 {
```

```
306    UVM_NONE   = 0,
307    UVM_LOW    = 100,
308    UVM_MEDIUM = 200,
309    UVM_HIGH   = 300,
310    UVM_FULL   = 400,
311    UVM_DEBUG  = 500
312  } uvm_verbosity;
```

get_report_verbosity_level 的定义如下：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object

 81  class uvm_report_object extends uvm_object;
 82
 83    uvm_report_handler m_rh;
       …
 90    function new(string name = "");
 91      super.new(name);
 92      m_rh = new();
 93    endfunction
       …
413    function int get_report_verbosity_level(uvm_severity severity=UVM_INFO, string id="");
414      return m_rh.get_verbosity_level(severity, id);
415    endfunction
       …
539  endclass
```

m_rh 是一个 uvm_report_handler 类型的变量，它会在每个 component 实例化的时候被实例化，也就是说，每个 component 对应一个 m_rh，此变量用于记录这个 component 的一些报告信息，如是否单独对此 component 设置了报告冗余度级别(verbosity_lever)。get_report_verbosity_level 这个函数最终调用的是 uvm_report_handler 的 get_verbosity_level 函数：

```
文件：src/base/uvm_report_handler.svh
类：uvm_report_handler

 55  class uvm_report_handler;
       …
236    function int get_verbosity_level(uvm_severity severity=UVM_INFO, string id="" );
237
238      uvm_id_verbosities_array array;
239      if(severity_id_verbosities.exists(severity)) begin
240        array = severity_id_verbosities[severity];
241        if(array.exists(id)) begin
242          return array.get(id);
243        end
244      end
245
246      if(id_verbosities.exists(id)) begin
247        return id_verbosities.get(id);
```

```
248         end
249
250         return  m_max_verbosity_level;
251
252     endfunction
            …
622 endclass : uvm_report_handler
```

这个函数首先会判断 severity_id_verbosities 是否存在着与 severity 相对应的记录。severity_id_verbosities 是一个联合数组，其索引是 uvm_severity 类型：

```
文件：src/base/uvm_object_globals.svh
类：无

247 typedef bit [1:0] uvm_severity;
```

而其内容则是 uvm_id_verbosities_array：

```
文件：src/base/uvm_report_handler.svh
类：无

52 typedef uvm_pool#(string, int) uvm_id_verbosities_array;
```

可见，其内容实质上是一个 uvm_pool，而 uvm_pool 的本质是一个派生自 uvm_object 的联合数组：

```
文件：src/base/uvm_pool.svh
类：uvm_pool

 35 class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;
        …
 42     protected T pool[KEY];
        …
232 endclass
```

所以 severity_id_verbosities 是这样的一个联合数组：它的索引是 uvm_severity(整数型)，而其内容则是一个 uvm_object，在这个 object 中有着一个联合数组，这个联合数组的索引是 string 类型的，而内容则是整数类型的。

到目前为止，我们没有往这个数组里面写过任何记录，所以它依然是空的，所以直接跳过这几行。接下来将会检查 id_verbosities 是否有与输入的 id 相匹配的记录。id_verbosities 是一个 uvm_id_verbosities_array 类型的变量：

```
文件：src/base/uvm_report_handler.svh
类：uvm_report_handler

 67     uvm_id_verbosities_array  id_verbosities;
```

所以 246 行的意思其实就是说检查一下 id_verbosities 内部的联合数组中是否存在与 ID 相匹配的记录。从这里我们可以看到，UVM 对于这种信息报告的控制到了

非常精细的地步。例如假如在 main_phase 中我们使用 main 这个 ID 来报告信息，而在 shutdown_phase 中使用 shutdown 这个 ID，那么我们可以分别对这两个 ID 的信息报告冗余级别进行设置，如可以把 main 设置为 UVM_HIGH，而把 shutdown 设置为 UVM_MEDIUM。

如果 id_verbosities 中也没有记录的话，那么将会直接返回 m_max_verbosity_level，这是一个 int 类型的变量，会在系统初始化的时候设置为 UVM_MEDIUM。

回到 uvm_report_enabled 函数。在 get_report_verbosity_level 返回后还要看一下 get_report _action 函数，这个函数与 get_report_verbossity_level，最终调用的是 uvm_report_handler 的 get_action 函数：

```
文件：src/base/uvm_report_handler.svh
类：uvm_report_handler
函数/任务：get_action

264    function uvm_action get_action(uvm_severity severity, string id);
265
266      uvm_id_actions_array array;
267      if(severity_id_actions.exists(severity)) begin
268        array = severity_id_actions[severity];
269        if(array.exists(id))
270          return array.get(id);
271      end
272
273      if(id_actions.exists(id))
274        return id_actions.get(id);
275
276      return severity_actions[severity];
277
278    endfunction
```

这个函数与 get_report_verbossity_level 几乎就是姐妹函数，它根据 severity_id_actions 和 id_actions 中的记录来给出返回值。这两者都是 uvm_id_actions_array 类型的变量：

```
文件：src/base/uvm_report_handler.svh
类：无

 50 typedef uvm_pool#(string, uvm_action) uvm_id_actions_array;
```

uvm_action 的定义如下：

```
文件：src/base/uvm_object_globals.svh
类：无

275 typedef int uvm_action;
276
```

```
277 typedef enum
278 {
279    UVM_NO_ACTION  = 'b000000,
280    UVM_DISPLAY    = 'b000001,
281    UVM_LOG        = 'b000010,
282    UVM_COUNT      = 'b000100,
283    UVM_EXIT       = 'b001000,
284    UVM_CALL_HOOK  = 'b010000,
285    UVM_STOP       = 'b100000
286 } uvm_action_type;
```

也就是说 UVM 除了对每个 component 的每个 ID 进行信息报告冗余级别控制之外，还针对每个冗余级别进行不同的行为控制。这两者可以相互补充，如把 main 设置为 UVM_MEDIUM，同时把 shutdown 的 UVM_MEDIUM 设置为 UVM_NO_ACTION。

11.1.2. uvm_report_error 函数

与 uvm_report_enabled 函数一样，它也有两个版本，一个是属于 uvm_report_object 的，一个是全局的，而全局的最终会调用 uvm_root 即 uvm_report_object 的这个函数，所以重点看 uvm_report_object 的这个函数：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object

 81 class uvm_report_object extends uvm_object;
 82
 83   uvm_report_handler m_rh;
      …
124   virtual function void uvm_report_error( string id,
125                                           string message,
126                                           int verbosity = UVM_LOW,
127                                           string filename = "",
128                                           int line = 0);
129     m_rh.report(UVM_ERROR, get_full_name(), id, message, verbosity,
130                 filename, line, this);
131   endfunction
      …
539 endclass
```

这个函数最终会调用 uvm_report_handler 的 report 函数，而这个 report 函数最终会调用 uvm_report_server 的 report 函数：

```
文件：src/base/uvm_report_handler.svh
类：uvm_report_handler
```

```
函数/任务：report

318    virtual function void report(
319          uvm_severity severity,
320          string name,
321          string id,
322          string message,
323          int verbosity_level,
324          string filename,
325          int line,
326          uvm_report_object client
327          );
328
329       uvm_report_server srvr;
330       srvr = uvm_report_server::get_server();
          …
345       srvr.report(severity,name,id,message,verbosity_level,filename,line,client);
346
347    endfunction
```

所以，下一节我们分析 uvm_report_server 类。

11.2. uvm_report_server

11.2.1. 类的实例化

这个类是派生自 uvm_object 的一个类：

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server

 40 typedef class uvm_report_catcher;
 41 class uvm_report_server extends uvm_object;
 42
 43    local int max_quit_count;
 44    local int quit_count;
 45    local int severity_count[uvm_severity];
 46
 47    // Needed for callbacks
 48    function string get_type_name();
 49       return "uvm_report_server";
```

```
 50    endfunction
 51
 52    // Variable: id_count
 53    //
 54    // An associative array holding the number of occurences
 55    // for each unique report ID.
 56
 57    protected int id_count[string];
 58
 59    bit enable_report_id_count_summary=1;
 60
 61
 62    // Function: new
 63    //
 64    // Creates the central report server, if not already created. Else, does
 65    // nothing. The constructor is protected to enforce a singleton.
 66
 67    function new();
 68       set_name("uvm_report_server");
 69       set_max_quit_count(0);
 70       reset_quit_count();
 71       reset_severity_counts();
 72    endfunction
 73
 74
 75    static protected uvm_report_server m_global_report_server = get_server();
       …
 99    static function uvm_report_server get_server();
100       if (m_global_report_server == null)
101          m_global_report_server = new;
102       return m_global_report_server;
103    endfunction
       …
520 endclass
```

这里最有意思的一点在于 new 函数前面的注释中，作者的意思是想把 new 声明成 protected 类型的，但是很遗憾，cadence 的仿真器目前还不支持 protected 类型，所以这里即使设置了也是没有用处。按照 UVM 的解释，一旦当 cadence 的仿真器开始支持 protected 时，将会这个关键字加上。上一章中 uvm_root 的 new 函数前面也是加了 protected 关键字的。其实那里加的 protected 是用一个宏来实现的，当使用 cadence 的仿真器时，这个 protected 会被替换成一个空格。为了便于理解，大家可以在这里把 new 当成是 protected 类型的。

如果 new 是 protected 类型的，大家看到 get_server 和 m_global_report_server 是不是感觉很眼熟呢？确实不错，它跟 uvm_root 的单实例实现是完全一样的。所以它其实也是单实例的。

这个类有几个成员变量。其中 max_quit_count 表示系统出现多少个 UVM_ERROR 时，会自动退出仿真，0 表示无限多个。

UVM 中把报告信息的优先级分成如下几个等级：

```
文件：src/base/uvm_object_globals.svh
类：无

249 typedef enum uvm_severity
250 {
251    UVM_INFO,
252    UVM_WARNING,
253    UVM_ERROR,
254    UVM_FATAL
255 } uvm_severity_type;
```

quit_count 表示当前已经有多少个 UVM_ERROR 发生了。severity_count 是一个联合数组，用于统计本次仿真中出现了多少个 UVM_EEOR，多少个 UVM_INFO，多少个 UVM_WARNING 等，会在仿真完成时打印出来。id_count 也是一个联合数组，用于统计在本次仿真中每个 id 有多少个信息显示。如下显示出一份报告，用到的就是这两个联合数组：

```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  568
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[conv_env]    19
[conv_mdl]   256
[conv_scb]   256
[demo]     4
[demo_seq]    32
```

图 11-1 UVM 仿真汇总报告

11.2.2. report 函数

我们最关心的还是 report 函数，其定义如下：

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数/任务：report

243    virtual function void report(
244          uvm_severity severity,
245          string name,
246          string id,
247          string message,
248          int verbosity_level,
249          string filename,
250          int line,
251          uvm_report_object client
252          );
253        string m;
254        uvm_action a;
255        UVM_FILE f;
256        bit report_ok;
257        uvm_report_handler rh;
258
259        rh = client.get_report_handler();
260
261        // filter based on verbosity level
262
```

259 行会返回一个 uvm_report_handler 的指针。这里的 client 指的是什么呢？我们是在 env.agent.driver 中调用的 uvm_error 宏，这里的 client 指的就是这个 driver。所以这里其实是调用 uvm_component 的 get_report_handler 函数，这个函数是在 uvm_component 的基类 uvm_report_object 中定义的：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object
函数/任务：get_report_handler

489    function uvm_report_handler get_report_handler();
490      return m_rh;
491    endfunction
```

前面已经提到过 m_rh，它是 uvm_report_object 的一个成员变量，在 new 的时候实例化，所以这里得到的将是每个 uvm_component 独有的一个 uvm_report_handler 的实例指针。

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数/任务：report

263        if(!client.uvm_report_enabled(verbosity_level, severity, id)) begin
264            return;
265        end
266
```

```
267       // determine file to send report and actions to execute
268
269       a = rh.get_action(severity, id);
270       if( uvm_action_type'(a) == UVM_NO_ACTION )
271         return;
```

263 到 265 行会调用 uvm_report_enabled 函数，这个函数前面已经有过介绍，它将会调用 get_report_verbosity_level，并根据返回的值来决定是否打印信息。

269 到 271 行则用于判断是否对这个 ID 设置了 UVM_NO_ACTION。其实 263 到 271 行做的事情与 uvm_error 宏中的 uvm_report_enabled 函数做的事情是完全一样的。那为什么这里还要再做一次？因为并不是所有的人都习惯使用 uvm_error 宏，而习惯于直接调用 uvm_report_error 函数。事实上，在 UVM1.0ea 版本及 OVM 中是没有 uvm_error 宏的。这个宏是在 1.0 版本中新出现的。所以这么做也是为了与以前的兼容。

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数/任务：report
272
273       f = rh.get_file_handle(severity, id);
274
275       // The hooks can do additional filtering.  If the hook function
276       // return 1 then continue processing the report.  If the hook
277       // returns 0 then skip processing the report.
278
279       if(a & UVM_CALL_HOOK)
280         report_ok = rh.run_hooks(client, severity, id,
281                                  message, verbosity_level, filename, line);
282       else
283         report_ok = 1;
284
```

273 行是得到用于输出信息的文件。这里用到了 uvm_report_handler 的 get_file_handle 函数。这个函数与 10.1.1 节中介绍的 get_report_verbossity_level 和 get_action 函数非常相似，这里不多做介绍。不过从这里也可以看的出来，UVM 可以对不同的 ID 设置不同的输出文件，这再次体现出了 UVM 的精细控制功能。尤其是当项目非常大的时候，这种信息的控制将会非常的重要。

280 行则调用 run_hooks 函数来再次确认是否要真正的打印信息。run_hooks 的定义如下：

```
文件：src/base/uvm_report_handler.svh
类：uvm_report_handler
函数/任务：run_hooks

165   virtual function bit run_hooks(uvm_report_object client,
166                                  uvm_severity severity,
167                                  string id,
```

```
168                                         string message,
169                                         int verbosity,
170                                         string filename,
171                                         int line);
172
173       bit ok;
174
175       ok = client.report_hook(id, message, verbosity, filename, line);
176
177       case(severity)
178         UVM_INFO:
179           ok &= client.report_info_hook     (id, message, verbosity, filename, line);
180         UVM_WARNING:
181           ok &= client.report_warning_hook(id, message, verbosity, filename, line);
182         UVM_ERROR:
183           ok &= client.report_error_hook    (id, message, verbosity, filename, line);
184         UVM_FATAL:
185           ok &= client.report_fatal_hook    (id, message, verbosity, filename, line);
186       endcase
187
188       return ok;
189
190     endfunction
```

这里将会根据 client，即 uvm_report_object 的 report_*_hook 函数来决定返回值。这几个函数在 uvm_report_object 的定义都相当简单，以 report_error_hook 为例：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object
函数/任务：report_error_hook

191     virtual function bit report_error_hook(
192                 string id, string message, int verbosity, string filename, int line);
193       return 1;
194     endfunction
```

这里看似无任何意义，其实是向用户提供了接口，可以更精准的用于控制信息的输出。用户可以重载这些函数来控制信息的打印。

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数/任务：report

285       if(report_ok)
286         report_ok = uvm_report_catcher::process_all_report_catchers(
287                             this, client, severity, name, id, message,
288                             verbosity_level, a, filename, line);
289
290       if(report_ok) begin
291         m = compose_message(severity, name, id, message, filename, line);
292         process_report(severity, name, id, message, a, f, filename,
```

```
293                             line, m, verbosity_level, client);
294       end
295
296     endfunction
```

回到 report 函数。286 行将会调用 uvm_report_catcher 的 process_all_report_catchers 函数。

uvm_report_catcher是一个派生自uvm_callback的类[1]。它的主要用处就是在真正的打印信息之前可以再次控制要打印的信息。用户可以定义自己的processs_all_report_catchers函数，从而实现信息打印的多样化。在本例中，并没有加入任何的callback函数，所以接下来会到 291 行，这一行将会调用compose_messsage函数，把要打印的信息，如时间，ID，文件行数等组织在一个字符串里，之后把这个字符串作为参数，调用processs_report函数：

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数：process_report

309     virtual function void process_report(
310         uvm_severity severity,
311         string name,
312         string id,
313         string message,
314         uvm_action action,
315         UVM_FILE file,
316         string filename,
317         int line,
318         string composed_message,
319         int verbosity_level,
320         uvm_report_object client
321         );
322       // update counts
323       incr_severity_count(severity);
324       incr_id_count(id);
325
326       if(action & UVM_DISPLAY)
327         $display("%s",composed_message);
328
329       // if log is set we need to send to the file but not resend to the
330       // display. So, we need to mask off stdout for an mcd or we need
331       // to ignore the stdout file handle for a file handle.
332       if(action & UVM_LOG)
333         if( (file == 0) || (file != 32'h8000_0001) ) //ignore stdout handle
334         begin
335           UVM_FILE tmp_file = file;
336           if( (file&32'h8000_0000) == 0) //is an mcd so mask off stdout
```

[1] 关于 callback，可以在第 19 章找到其代码分析。

```
337             begin
338                 tmp_file = file & 32'hffff_fffe;
339             end
340             f_display(tmp_file,composed_message);
341         end
342
343     if(action & UVM_EXIT) client.die();
344
```

323 到 324 行把相应的计数器加 1，326 行则决定是否把信息输出到屏幕上，332 到 341 行用于把信息送到 log 文件里。343 行则决定是否调用 die 函数，这个函数定义在 uvm_report_object 类中：

```
文件：src/base/uvm_report_object.svh
类：uvm_report_object
函数/任务：die

271   virtual function void die();
272     // make the pre_abort callbacks
273     uvm_root top = uvm_root::get();
274     top.m_do_pre_abort();
275
276     report_summarize();
277     $finish;
278   endfunction
```

这里会调用 uvm_root 的 m_do_pre_abort 函数，进行仿真结束前的清理工作，之后调用 report_summarize 函数，把所有的统计信息打印出来，如有多少个 UVM_ERROR 出现等，每个不同的 ID 有多少条信息打印。这个函数比较简单，不多做阐述。最后会调用$finish 退出仿真。例如如果是一个 uvm_fatal 宏的话，这里就会结束仿真。

```
文件：src/base/uvm_report_server.svh
类：uvm_report_server
函数：process_report

345     if(action & UVM_COUNT) begin
346       if(get_max_quit_count() != 0) begin
347           incr_quit_count();
348         if(is_quit_count_reached()) begin
349           client.die();
350         end
351       end
352     end
353
354     if (action & UVM_STOP) $stop;
355
356   endfunction
```

345 到 352 行用来查看 UVM_ERROR 数是否已经达到了预先设定的值，如果是

的话那也会直接结束仿真。

354 行则根据 action 来调用系统的$stop 函数，以挂起仿真。

11.2.3. UVM 对于信息打印的精细控制

读完了上节，大家对 UVM 的信息输出控制肯定吧为观止。总结一下，UVM 中一共可以通过如下的方式来实现输出信息的控制：

第一，通过控制 ID 来实现不同的输出控制。

第二，通过设置不同 ID 的信息报告冗余级别。

第三，通过设置不同 ID 的 action。

第四，从 uvm_report_catcher 派生一个类，然后把这个 callback 加入到 callback 池中。

12. factory 机制源代码分析

factory 机制可以说是整个 UVM 的基石，其重要性不言而喻。本章第一和第二节讲述 factory 机制的基本原理，第三节讲述 factory 机制的典型应用，第四节讲述用于 component 的 factory 机制，第五节讲述常用的胜于 factory 机制的宏，第六节讲述 override 功能。本章最关键的在于第二节，而难点在于第六节。不过相对来说，第六节的重要性并不高。所以在阅读本章时，只要读懂第二节就可以，第六节可以在用到时再细看。

12.1. 根据字符串创建一个类的实例

在 2.4.3 节中，我们反复强调了一个事实，factory 机制的一大特点就是根据类的名字来创建类的实例。在 10.2.4 节中讲述 uvm_root::run_test 时，就直接看到了这种用法。为什么这里要反复强调这个功能呢？

12.1.1. 创建类的实例的方法

一般的面向对象的编程语言中，要创建一个类的实例，有两种方法，一种是在类的可见的作用范围之内，直接创建：

```
class A
  …
endclass
class B;
    A a;
    function new();
        a = new();
    endfunction
endclass
```

另外一种是使用参数化的类：

```
class parameterized_class # (type T)
    T t;
    function new();
        t = new();
    endfunction
endclass
class A;
    …
endclass
class B;
    parameterized_classs#(A) pa;
    function new();
        pa = new();
    endfunction
endclass
```

这样 pa 实例化的时候，其内部就创建了一个属于 A 类型的实例 t。但是，如何通过一个字符串来创建一个类？当然了，这里的前提是这个字符串代表一个类的名字。

```
class A;
    …
endclass
class B;
    string type_string;
    function new();
        type_string = “A”;
        //how to create an instance of A according to type_string???
    endfunction
endclass
```

据我所知，没有任何语言会内建一种如上的机制：即通过一个字符串来创建此字符串所代表的类的一个实例。如果要实现这种功能，需要自己做，factory 机制正是用于实现上述功能。

12.2. uvm_object_utils 宏

UVM 中有两大关键类，uvm_object 和 uvm_component。一个 uvm_object 在定义时一般要调用 uvm_object_utils 宏，而一个 uvm_component 在定义时要调用 uvm_component_utils 宏。factory 所有的操作都通过这两个宏来完成。

本节将会介绍 uvm_object_utils 宏。

12.2.1. uvm_object_utils 宏展开

uvm_object_utils 宏的定义如下：

```
文件：src/macros/uvm_object_defines.svh
类：无

 218 `define uvm_object_utils(T) \
 219     `uvm_object_utils_begin(T) \
 220     `uvm_object_utils_end
```

其中 uvm_object_utils_end 非常简单：

```
文件：src/macros/uvm_object_defines.svh
类：无

 237 `define uvm_object_utils_end \
 238         end \
 239       endfunction \
```

重点看 uvm_object_utils_begin:

```
文件：src/macros/uvm_object_defines.svh
类：无

 226 `define uvm_object_utils_begin(T) \
 227       `m_uvm_object_registry_internal(T,T)    \
 228       `m_uvm_object_create_func(T) \
 229       `m_uvm_get_type_name_func(T) \
 230       `uvm_field_utils_begin(T)
```

12.2.2. m_uvm_object_registry_internal 宏

宏的定义为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 407 `define m_uvm_object_registry_internal(T,S) \
 408     typedef uvm_object_registry#(T,`"S`") type_id; \
 409     static function type_id get_type(); \
 410       return type_id::get(); \
 411     endfunction \
 412     virtual function uvm_object_wrapper get_object_type(); \
 413       return type_id::get(); \
 414     endfunction
```

这个宏的第一句是一个 typedef 语句。factory 机制的注册都是在这个 typedef 语句中实现的。一个 typedef 语句能够实现注册过程？你没有看错。这其中很巧妙的用到了类的静态成员变量。这个 typedef 语句声明了一个类，这个类是一个参数化的类，类的名字是 uvm_object_registry，而类的参数有两个，第一个是输入的类型 T，第二个是一个字符串 S。

类 uvm_object_registry 的原型如下：

```
文件：src/base/uvm_registry.svh
类：uvm_object_registry

182 class uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")
183                                                 extends uvm_object_wrapper;
184   typedef uvm_object_registry #(T,Tname) this_type;
      …
212   local static this_type me = get();
      …
219   static function this_type get();
220     if (me == null) begin
221       uvm_factory f = uvm_factory::get();
222       me = new;
223       f.register(me);
224     end
225     return me;
226   endfunction
      …
296 endclass
```

这个类派生自 uvm_object_wrapper。这是我们第一次遇到这个类，其定义为：

```
文件：src/base/uvm_factory.svh
类：uvm_object_wrapper
```

```
641  virtual  class  uvm_object_wrapper;
        …
674  endclass
```

这是一个纯虚类，它只提供了一些接口函数，是不能直接实例化的，必须要把其派生之后才能实例化。

回到 uvm_object_registry 上来，这个类的内部有一个私有的静态成员变量 me，这个变量的类型是 this_type，即这个类本身，也就是说这个成员变量指向这个类的一个实例。me 在定义的时候就通过 get 初始化了。get 首先检查 me 是否为 null，如果不为 null，说明 me 已经是非空了，因此直接返回 me 的值。如果为 null 的话，则创建一个 me 的实例，然后通过 factory 的 register 函数注册到 factory 中。

本节暂时先不介绍 uvm_factory 和其 get 及 register 等方法，后面会有详细介绍。这里暂时把 uvm_factory 看成一个黑盒子，通过 register 把 me 推进这个盒子中。也就是实现了注册过程。

现在回到 typedef uvm_object_registry#(T,`"S`") type_id 这句话来。前面说过，这句话实现了 factory 的注册机制。可是我们也看过了 uvm_object_registry 这个类的实现，似乎并没有把我们 T 这个类注册进 factory 中，而只是把 uvm_object_registry 类的一个成员变量 me 注册到了 factory 中。到底应该怎么理解呢？

这里要区分参数化的类的一个概念。

```
typedef  uvm_object_registry#(T1,"S1")  type1
typedef  uvm_object_registry#(T2,"S2")  type2
```

这里是两个参数化的类，当这两句话在编译时，编译器就会认为 uvm_object_registry# (T1,"S1")是一个类，uvm_object_registry#(T2,"S2")是另外一个类，而且这两个类都与 class uvm_object_registry#(type T=uvm_object, string Tname="<unknown>")是完全不一样的类。这是由参数化的类决定的。因此，对于 type1 和 type2，各自有一个 me 的静态成员变量，我们可以使用 type1::me 和 type2::me 来引用这个静态成员变量。

uvm_object_registry#(T1,"S1")中通过 get 静态函数把 type1::me 注册到了 factory 中，而 uvm_object_registry#(T2,"S2")通过 get 静态函数把 type2::me 注册到了 factory 中。

现在 factory 中有两个 me 了，这两个 me 是 type1::me 和 type2::me。比较一下这两个 me，就会发现，me 是 uvm_object_registry#(T1,"S1")这个类的一个静态成员变量，对于这一个类来说，只有这么一个，而 uvm_object_registry#(T1,"S1")是与什么相关的？是与 T1 相关的，对于一个 T1，就会有一个 uvm_object_registry#(T1,"S1")，从而会有一个 me，注册了 uvm_object_registry#(T1,"S1")的 me，就相当于把 T1 的信息告诉给了 factory，也即是间接实现了 T1 的注册。

所以，所谓的把一个 object 或者 component 注册到 factory 中，其实不是注册的这个 object 或者 component，而是注册的 uvm_object_registry#(T,"S")的一个实例。类似于曲线救国。这里也可以理解成是把 T 类型外面包了一层皮，然后把包了皮后的 object 和 component 注册到 factory 中。对，这是一层皮，一个 wrapper，所以这就是为什么 uvm_object_registry 的基类名字叫 uvm_object_wrapper。

回到 m_uvm_object_registry_internal(T,T)这个宏的定义来，后面实现了两个函数 get_type 和 get_object_type，比较简单，这里就不多阐述。

12.2.3. uvm_object_utils_begin 宏的其它部分

uvm_object_utils_begin 的其它部分如下：

文件：src/macros/uvm_object_defines.svh
类：无

```
382 `define m_uvm_object_create_func(T) \
383     function uvm_object create (string name=""); \
384       T tmp; \
385       tmp = new(); \
386       if (name!="") \
387         tmp.set_name(name); \
388       return tmp; \
389     endfunction
        …
395 `define m_uvm_get_type_name_func(T) \
396     const static string type_name = `"T`"; \
397     virtual function string get_type_name (); \
398       return type_name; \
399     endfunction
```

这两个宏来很简单。且不涉及到具体的 factory 机制，因此不多做阐述。后面的 uvm_field_utils_begin(T)宏则是用于实现 uvm 的 field automation 机制的实现，在后面的章节中会做阐述。

12.2.4. uvm_factory 类

在 12.2.2 节中讲述到 uvm_object_registry 的 get 函数时，对 uvm_factory 一带而过，这节详细说一下这个类，以及 class uvm_object_registry #(type T=uvm_object,

string Tname="< unknown>")中的静态变量 me 是如何注册到其中的。

uvm_factory 的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

  69 class uvm_factory;
  70
  71   extern `_protected function new ();
  72
  73   extern static function uvm_factory get();
       …
 325 endclass
     …
 711 const uvm_factory factory = uvm_factory::get();
     …
 722 function uvm_factory uvm_factory::get();
 723   if (m_inst == null) begin
 724     m_inst = new();
 725   end
 726   return m_inst;
 727 endfunction
     …
 732 function uvm_factory::new ();
 733 endfunction
```

看到这些，大家是不是似曾相识呢？不错，这里跟 uvm_root 的单实例实现完全一样。所以 uvm_factory 其实也是一个单实例的类。

重点来看 register 函数。其定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：register

 739 function void uvm_factory::register (uvm_object_wrapper obj);
 740
 741   if (obj == null) begin
 742     uvm_report_fatal ("NULLWR", "Attempting to register a null object with the factory",
UVM_NONE);
 743   end
 744   if (obj.get_type_name() == "" || obj.get_type_name() == "<unknown>") begin
 745     //uvm_report_warning("EMPTNM", {"Factory registration with ",
 746     //  "unknown type name prevents name-based operations. "});
 747   end
 748   else begin
 749     if (m_type_names.exists(obj.get_type_name()))
 750       uvm_report_warning("TPRGED", {"Type name '",obj.get_type_name(),
 751         "' already registered with factory. No string-based lookup ",
 752         "support for multiple types with the same type name."}, UVM_NONE);
 753     else
```

```
754          m_type_names[obj.get_type_name()] = obj;
755      end
756
```

这个函数的实现相对来说比较复杂，一点一点的来看。

741 到 743 行比较简单，就是查看一下传递进来的参数是不是一个 null 值。744 到 747 行同样简单，略过不提。

748 到 755 行把 obj 这个要注册的指向某个实例的指针放入 m_type_names 中，在放入之前先检查一下 m_type_names 中是否已经有了这条记录。

m_type_names 是一个联合数组，其定义为：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

307    protected uvm_object_wrapper     m_type_names[string];
```

这个联合数组的索引是 string 类型的，其存储的内容是 uvm_object_wrapper 类型的。

结合 12.2.2 节，把 uvm_object_registry#(T1,"S1")的静态成员变量 me 作为参数传入到 register 中，那么 m_type_names 中就会增加一条记录，这条记录的索引就是 me.get_type_name()的返回值，即"S1"，这条记录的内容就是 me 所指向的实例。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：register

757    if (m_types.exists(obj)) begin
758      if (obj.get_type_name() != "" && obj.get_type_name() != "<unknown>")
759        uvm_report_warning("TPRGED", {"Object type '",obj.get_type_name(),
760                                      "' already registered with factory. "}, UVM_NONE);
761    end
762    else begin
763      m_types[obj] = 1;
764      // If a named override happens before the type is registered, need to copy
765      // the override queue.
766      // Note:Registration occurs via static initialization, which occurs ahead of
767      // procedural (e.g. initial) blocks. There should not be any preexisting overrides.
768      if(m_inst_override_name_queues.exists(obj.get_type_name())) begin
769        m_inst_override_queues[obj] = new;
770        m_inst_override_queues[obj].queue = m_inst_override_name_queues[obj.get_type_nam
e()].queue;
771        m_inst_override_name_queues.delete(obj.get_type_name());
772      end
773      if(m_wildcard_inst_overrides.size()) begin
774        if(! m_inst_override_queues.exists(obj))
775              m_inst_override_queues[obj] = new;
776        foreach (m_wildcard_inst_overrides[i]) begin
```

```
 777                 if(uvm_is_match( m_wildcard_inst_overrides[i].orig_type_name, obj.get_type_name
())) 
 778                     m_inst_override_queues[obj].queue.push_back(m_wildcard_inst_overrides[i]);
 779             end
 780         end
 781 
 782     end
 783 
 784 endfunction
```

757 行检查 m_types 中是否已经有了要注册的记录。m_types 也是一个联合数组，其定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

 305     protected bit                           m_types[uvm_object_wrapper];
```

这个联合数组的索引是 uvm_object_wrapper，而存储的内容是 bit 类型的。

如果 m_types 中没有找到相关的记录则进入 else 分支，763 行会往 m_types 中加入一条记录，这个记录的索引是 uvm_object_registry#(T1,"S1")的静态成员变量 me 所指向的实例，而内容是 1。代表着这个 uvm_object_registry#(T1,"S1")已经在 factory 中注册过了。

768 到 780 行则主要是用于 override 的，后面会介绍，这里暂且略过。

```
文件：src/base/uvm_registry.svh
类：uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")

182 class uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")
183                                                 extends uvm_object_wrapper;
184     typedef uvm_object_registry #(T,Tname) this_type;
        …
212     local static this_type me = get();
        …
219     static function this_type get();
220       if (me == null) begin
221         uvm_factory f = uvm_factory::get();
222         me = new;
223         f.register(me);
224       end
225       return me;
226     endfunction
        …
296 endclass
```

总结一下如上所谓的注册，其实质就是把 uvm_object_registry#(T1,"S1")的 me 放入 m_type_names 中，并把"S1"作为索引；把 me 作为 m_types 的索引，其内容为 1。

12.3. factory 机制的应用

12.3.1. 根据类名创建类的一个实例

在之前的介绍中，一直强调 factory 的根据类名来创建类的实例的功能。现在距离这个目标越来越近了。

```
class A extends uvm_object;
    `uvm_object_utils(A)
    ...
endclass
class test;
    uvm_object a;
    function new();
        uvm_factory f = uvm_factory::get();
        a = f.create_object_by_name("A");
endclass
```

如上，所创建的类的实例就是 A 类型的，而不是 uvm_object 类型的，我们看一下 create_object_by_name 是如何工作的。如下为这个函数的定义：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：create_object_by_name

1052 function uvm_object uvm_factory::create_object_by_name (string requested_type_name,
1053                                                          string parent_inst_path="",
1054                                                          string name="");
1055
1056    uvm_object_wrapper wrapper;
1057    string inst_path;
1058
1059    if (parent_inst_path == "")
1060       inst_path = name;
1061    else if (name != "")
1062       inst_path = {parent_inst_path,".",name};
1063    else
1064       inst_path = parent_inst_path;
1065
1066    m_override_info.delete();
1067
1068    wrapper = find_override_by_name(requested_type_name, inst_path);
1069
1070    // if no override exists, try to use requested_type_name directly
```

```
1071    if (wrapper==null) begin
1072      if(!m_type_names.exists(requested_type_name)) begin
1073        uvm_report_warning("BDTYP",{"Cannot create an object of type '",
1074         requested_type_name,"' because it is not registered with the factory."}, UVM_NON
E);
1075        return null;
1076      end
1077      wrapper = m_type_names[requested_type_name];
1078    end
1079
1080    return wrapper.create_object(name);
1081
1082  endfunction
```

1059 到 1068 行的主要用 override 发生的时候，应该使用 override 之后的类型来创建实例。这个下节会介绍。

1071 行到 1078 行是当 override 不存在的时候才会走的分支，它的主要作用就是到 m_type_names 这个数组中去寻找 requested_type_name，并把 m_type_names 中记录的内容作为 wrapper 的值。具体到我们的例子中，就寻找 m_type_names 中索引为字符串"A"的记录，这条记录的内容就是之前我们注册时放入的 uvm_object_registry#(myclass,"myclass")::me，把其赋值给 wrapper。

create_object_by_name 最关键的一句其实就是它的 return 语句。return 语句调用 wrapper.create_object 方法。在我们的例子中，就是调用 uvm_object_registry#(A, "A")::me. create_object()。

回过头来看一下 class uvm_object_registry #(type T=uvm_object, string Tname= "<unknown>")类的 create_object 函数：

```
文件：src/base/uvm_registry.svh
类：uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")
函数/任务：create_object

193    virtual function uvm_object create_object(string name="");
194      T obj;
195      obj = new();
196      if (name!="")
197        obj.set_name(name);
198      return obj;
199    endfunction
```

具体到我们的例子中，uvm_object_registry#(A,"A")的相当于是：

```
virtual function uvm_object create_object(string name="");
  A obj;
  obj = new();
  if (name!="")
    obj.set_name(name);
  return obj;
```

```
endfunction
```

为什么这里可以创建 A 类型的实例？因为 uvm_object_registry 是一个参数化的类，myclass 是作为一个类型参数传递进来的！这恰好是本章在最开始的时候所讲述的两种创建类的实例的方法中的一种。

小结一下，factory 机制中根据类名来创建类的实例所用到的技术：一是参数化的类，二是静态变量和静态函数。这两者是 factory 机制实现的根本所在。

12.3.2. factory 机制下独特的实例化的方法

大家现在都对使用 factory 机制来创建一个类的实例的古怪写法记忆犹新：

```
class A extends uvm_object;
    …
    `uvm_object_utils(A)
endclass
A a;
a = A::type_id::create("a");
```

关于 type_id，它的定义其实是隐含在 m_uvm_object_registry_internal 宏，也即隐藏在 uvm_object_utils 宏里的：

```
文件：src/macros/uvm_object_defines.svh
类：无

 407 `define m_uvm_object_registry_internal(T,S) \
 408     typedef uvm_object_registry#(T,`"S`") type_id; \
         …
```

在我们的例子中就是：

```
typedef uvm_object_registry#(A,"A") type_id;
```

uvm_object_registry#(A,"A")中的 create 函数如下：

```
文件：src/base/uvm_registry.svh
类：uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")
函数/任务：create

237   static function T create (string name="", uvm_component parent=null,
238                             string contxt="");
239     uvm_object obj;
240     uvm_factory f = uvm_factory::get();
241     if (contxt == "" && parent != null)
242       contxt = parent.get_full_name();
```

```
243       obj = f.create_object_by_type(get(),contxt,name);
244       if (!$cast(create, obj)) begin
245         string msg;
246         msg = {"Factory did not return an object of type '",type_name,
247           "'. A component of type '",obj == null ? "null" : obj.get_type_name(),
248           "' was returned instead. Name=",name," Parent=",
249           parent==null?"null":parent.get_type_name()," contxt=",contxt};
250         uvm_report_fatal("FCTTYP", msg, UVM_NONE);
251       end
252     endfunction
```

create 最终会调用 uvm_factory 的 create_object_by_type：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：create_object_by_type

1088 function uvm_object uvm_factory::create_object_by_type (uvm_object_wrapper requested_typ
e,
1089                                                         string parent_inst_path="",
1090                                                         string name="");
1091
1092    string full_inst_path;
1093
1094    if (parent_inst_path == "")
1095      full_inst_path = name;
1096    else if (name != "")
1097      full_inst_path = {parent_inst_path,".",name};
1098    else
1099      full_inst_path = parent_inst_path;
1100
1101    m_override_info.delete();
1102
1103    requested_type = find_override_by_type(requested_type, full_inst_path);
1104
1105    return requested_type.create_object(name);
1106
1107  endfunction
```

与 create_object_by_name 类似，create_object_by_type 最终会调用 uvm_object_registry# (A,"A")类的 create_object 函数，后者已经在上节中有过介绍，这里不多做阐述。

12.3.3. factory 机制的反思

通过前面的章节，已经把 factory 机制的基本原理讲述的差不多了。似乎这种机

制也并没有多么高明，而且看起来相当复杂。那么，这种机制最高明的地方在哪里？

在回答这个问题之前，先想一个问题，为什么在 factory 中要引入 uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")这么一个类？引入之后绕来绕去，绕的巨复杂无比？

factory 机制的核心就是一个联合数组，m_type_names。这个联合数组的索引是 string 类型的，其存储的内容是 uvm_object_wrapper 类型的。

想像一下，在 systemverilog 中，我们要往任何的数组或者 queue 中存放东西，存放的永远是值，而不可能是一个类型。我们只能说其中存放了某个类型的值，而不能说存放了一个类型。形象点说，假如我们定义好了一个 A 类，这个类的名字就是 A，我们要把这个类存放在联合数组中，那么应该存什么？第一，我们可以存放"A"这个字符串，但是这只是字符串，而不是类。第二，我们可以声明如下的一个数组：

```
A   testarray[3];
```

这样，testarray 中就可以存放 A 类型的实例指针了，注意，存放的是 A 类型的实例的指针，而不可能是 A 这个类。类是一个抽象的概念，是不可能存放在一个数组里面的！我们只有先把这个类实例化了，然后才能把实例的指针放入数组里面。

基于这样的一种考虑，那么我们可以把一个 A 类的指针直接放入 m_type_names 中，干吗非要引入 uvm_object_registry#(A,"A")这样的一个类呢？

```
class A extends uvm_object;
    `uvm_object_utils(A)
    …
endclass
```

在上面这个类中，我们要把一个 A 实例的指针放入 m_type_names 数组中，其索引是字符串"A"，这样要想实现 create_object_by_name("A")时，可以直接到 m_type_names 中搜索"A"，找出的是一个 A 类型的实例，要想创建一个新的实例，那么就 m_type_names["A"].copy()可以了。这样就可以直接省去 uvm_object_registry 这个类的，节省了存储空间，而且变的非常简单了。

看上去确实是挺不错的主意，但是关键是，怎么样在 A 定义的时候把 A 的一个实例的指针放入 m_type_names 中呢？这是很难的事情，可以用静态成员变量来实现。

```
class A extends uvm_object;
    static A m_a = A:;get();
    static function A get();
        if (m_a == null) begin
            uvm_factory f=uvm_factory::get();
            m_a = new();
            f.register(m_a);
        end
```

```
        return m_a;
    endfunction
    …
endclass
```

如上就可以实现我们要的功能。但是上述过程比较复杂，如果作为 UVM 的用户，每从 uvm_object 类派生一个类的时候，都要这么做，那么我相信每个用户都会很快放弃 UVM 的。UVM 的伟大之处就在于，把上述过程给标准化了，通过调用一个 uvm_object_utils 的宏来完成上述事情。通过使用 uvm_object_registry#(A,"A")这样一个中间类，在这个类中做与上面类似的事情，即使用一些静态变量和静态函数，通过静态变量的方法产生 uvm_object_registry #(A,"A")的一个实例，把其加入到 factory 的 m_type_names 中去。仔细回味，我们才觉 uvm_object_registry 类的巧妙。

12.4. uvm_component_utils 宏

12.4.1. uvm_component_utils 宏的展开

这个宏的展开如下：

```
文件：src/macros/uvm_object_defines.svh
类：无

 303 `define uvm_component_utils(T) \
 304     `m_uvm_component_registry_internal(T,T) \
 305     `m_uvm_get_type_name_func(T) \
```

其中 m_uvm_get_type_name_func 的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 395 `define m_uvm_get_type_name_func(T) \
 396     const static string type_name = `"T`"; \
 397     virtual function string get_type_name (); \
 398       return type_name; \
 399     endfunction
```

相对来说比较简单，而 m_uvm_component_registry_internal 则相对复杂。

12.4.2. m_uvm_component_registry_internal

m_uvm_component_registry_internal 的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 435 `define m_uvm_component_registry_internal(T,S) \
 436     typedef uvm_component_registry #(T,`"S`") type_id; \
 437     static function type_id get_type(); \
 438       return type_id::get(); \
 439     endfunction \
 440     virtual function uvm_object_wrapper get_object_type(); \
 441       return type_id::get(); \
 442     endfunction
```

这个宏与 12.2.2 节介绍的 m_uvm_object_registry_internal 几乎一模一样，唯一的区别就是 uvm_component_registry 和 uvm_object_registry 的区别。而这两个类几乎是完全对称的两个类，其函数原型分别为：

```
class uvm_component_registry #(type T=uvm_component, string Tname="<unknown>")
                                                    extends uvm_object_wrapper;
class uvm_object_registry #(type T=uvm_object, string Tname="<unknown>")
                                                  extends uvm_object_wrapper;
```

类的声明都如此相似，类的内容也完全相似。前面我们仔细的介绍了 uvm_object_registry 类，因此这里不多阐述 uvm_component_registry 类，有兴趣的读者可以按照前面的步骤试着分析一下这个类。

12.5. 其它用于 factory 注册的宏

12.5.1. uvm_object_param_utils 宏

这是一个参数化的 uvm_object_utils 宏。前面我们从 uvm_object 派生一个普通类的时候，使用的的是 uvm_object_utils 宏，而如果派生出一个参数化的类的话，则需要使用这个宏：

```
class para_class#(type T=int) extends uvm_object;
    typedef para_class#(type T=int) my_type;
    `uvm_object_param_utils(my_type)
    …
endclass
```

这个宏的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 222 `define uvm_object_param_utils(T) \
 223     `uvm_object_param_utils_begin(T) \
 224     `uvm_object_utils_end
```

其中 uvm_object_utils_end 前面已经见过了，这里不再赘述。uvm_object_param_utils_ begin 的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 232 `define uvm_object_param_utils_begin(T) \
 233     `m_uvm_object_registry_param(T)   \
 234     `m_uvm_object_create_func(T) \
 235     `uvm_field_utils_begin(T)
```

对比一下前面分析过的 uvm_object_utils_begin 宏：

```
文件：src/macros/uvm_object_defines.svh
类：无

 226 `define uvm_object_utils_begin(T) \
 227     `m_uvm_object_registry_internal(T,T)   \
 228     `m_uvm_object_create_func(T) \
 229     `m_uvm_get_type_name_func(T) \
 230     `uvm_field_utils_begin(T)
```

后者多了一个 m_uvm_get_type_name_func，且后者的 m_uvm_object_registry_internal 宏在前者中被 m_uvm_object_registry_param 取代了。m_uvm_object_registry_param 的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 420 `define m_uvm_object_registry_param(T) \
 421     typedef uvm_object_registry #(T) type_id; \
 422     static function type_id get_type(); \
 423       return type_id::get(); \
 424     endfunction \
 425     virtual function uvm_object_wrapper get_object_type(); \
 426       return type_id::get(); \
 427     endfunction
```

而 m_uvm_object_registry_internal 的宏的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 407 `define m_uvm_object_registry_internal(T,S) \
 408     typedef uvm_object_registry#(T,`"S`") type_id; \
 409     static function type_id get_type(); \
 410       return type_id::get(); \
 411     endfunction \
 412     virtual function uvm_object_wrapper get_object_type(); \
 413       return type_id::get(); \
 414     endfunction
```

这两个宏之间的差别在于 typedef 语句，m_uvm_object_registry_param 宏没有传入名字信息，而 m_uvm_object_registry_internal 则传入了。所以对于`uvm_object_param_utils (my_type)来说，最终在 factory 的 m_types 中插入的记录的索引将会是 uvm_object_registry# (my_type,<unknown>)类型的，其中的<unknown>为 uvm_object_registry 类的默认参数。

这里又出现了参数化的类，而 uvm_object_registry 本身就是一个参数化的类，所以这相当于这个参数化的类的参数是一个参数的类。这句话相当绕口。

```
class para_class#(type T=int) extends uvm_object;
    typedef para_class#(type T=int) my_type;
    `uvm_object_param_utils(my_type)
    …
endclass
```

假设有如下两句话：

```
typedef para_class#(int) int_class;
typedef para_class#(string) string_class;
```

那么对于 int_class 来说，相当于是 factory 中多了索引是如下的一条记录：

```
uvm_object_registry#(para_class#(int),<unknown>)::me
```

而对于 string_class 来说，相当于是多了索引是如下的一条记录：

```
uvm_object_registry#(para_class#(string),<unknown>)::me
```

很明显，这两个索引是不一样的。因此对于这种由 uvm_object 派生而来并且使用 uvm_object_param_utils 注册的参数化的类，不能使用 create_object_by_name 来创建实例了。因为假如要使用这个函数，那么应该传入什么参数来代表要创建的类的名字呢？但是可以使用 create_object_by_type 函数来创建。

12.5.2. uvm_component_utils_begin 宏

这个宏的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 310 `define uvm_component_utils_begin(T) \
 311     `m_uvm_component_registry_internal(T,T) \
 312     `m_uvm_get_type_name_func(T) \
 313     `uvm_field_utils_begin(T)
```

相比 uvm_component_utils 宏，它只是多了最后一句，用于实现 field_autumation 机制。本书后面章节将会在讲述 field_automation 机制时仔细阐述。

12.5.3. uvm_component_param_utils 宏

这个宏的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 307 `define uvm_component_param_utils(T) \
 308     `m_uvm_component_registry_param(T) \
```

而 m_uvm_component_registry_param 的展开为：

```
文件：src/macros/uvm_object_defines.svh
```

```
类：无

 450 `define m_uvm_component_registry_param(T) \
 451     typedef uvm_component_registry #(T) type_id; \
 452     static function type_id get_type(); \
 453       return type_id::get(); \
 454     endfunction \
 455     virtual function uvm_object_wrapper get_object_type(); \
 456       return type_id::get(); \
 457     endfunction
```

它与 12.5.1 节中 uvm_object_param_utils 宏最终展开的内容相同。它与 uvm_object_ param_utils 的区别和联系就如同 uvm_component_utils 和 uvm_object_utils 的区别和联系。本文在这里不重复叙述。

12.6. override 功能

override 功能也是 UVM 中一个比较重要的功能。通常的，我们会在 env 或者具体的 case 中使用 override 功能：

```
class case_x extends base_test;
    function void build_phase(uvm_phase phase);
        …
        set_type_override_by_type(my_driver::get_type(), new_driver::get_type());
    endfunction
endclass
```

set_type_override_by_type 是 uvm_component 的一个函数，其定义如下：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：set_type_override_by_type

2098 function void uvm_component::set_type_override_by_type (uvm_object_wrapper original_type,
2099                                                         uvm_object_wrapper overri
de_type,
2100                                                         bit       replace=1);
2101     factory.set_type_override_by_type(original_type, override_type, replace);
2102 endfunction
```

它会调用 uvm_factory 类的 set_type_overide_by_type 函数。与这个函数相对应的，还有 set_在分析这个函数之前，我们先看一下 uvm_factory 用于记录 override 信息的数据结构。

12.6.1. 用于 override 功能的数据结构

factory 机制的 override 功能是通过几个队列实现的。最重要一个队列如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

 309    protected uvm_factory_override m_type_overrides[$];
```

这个队列中存储的是 uvm_factory_override 类型的变量。uvm_factory_override 是专门用于组织 factory 中的 override 信息的一个类。其定义为：

```
文件：src/base/uvm_factory.svh
类：uvm_factory_override

 684 class uvm_factory_override;
 685    string full_inst_path;
 686    string orig_type_name;
 687    string ovrd_type_name;
 688    bit selected;
 689    uvm_object_wrapper orig_type;
 690    uvm_object_wrapper ovrd_type;
        …
 704 endclass
```

这个类中记录了原始类型的名字，原始类型，override 类型的名字，override 类型及 full_inst_path 信息。前面四个我们可以理解，那么这一个变量是有什么用处呢？变量用于存储被 override 的 uvm_component 的路径。

另外两个用于记录 override 信息的队列是：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

 313    protected uvm_factory_override      m_wildcard_inst_overrides[$];
 314
 315    local uvm_factory_override        m_override_info[$];
```

与 m_type_overrides 队列类似，这两个队列中存放的内容同样也是 uvm_factory_override 类型的。

12.6.2. set_type_overide_by_type 函数

uvm_factory 类中 set_type_override_by_type 的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_type_override_by_type

 790 function void uvm_factory::set_type_override_by_type (uvm_object_wrapper original_type,
 791                                                        uvm_object_wrapper override
_type,
 792                                                        bit replace=1);
 793   bit replaced;
 794
 795   // check that old and new are not the same
 796   if (original_type == override_type) begin
 797     if (original_type.get_type_name() == "" || original_type.get_type_name() == "<unknow
n>")
 798       uvm_report_warning("TYPDUP", {"Original and override type ",
 799                                     "arguments are identical"}, UVM_NONE);
 800     else
 801       uvm_report_warning("TYPDUP", {"Original and override type ",
 802                                     "arguments are identical: ",
 803                                     original_type.get_type_name()}, UVM_NONE);
 804     return;
 805   end
 806
 807   // register the types if not already done so, for the benefit of string-based lookup
 808   if (!m_types.exists(original_type))
 809     register(original_type);
 810
 811   if (!m_types.exists(override_type))
 812     register(override_type);
 813
 814
```

795 到 805 行会比较一下 override 前后的两个类型是不是同一个类型，如果是同一类型就没有必要 override。

808 到 812 行则会检查一下输入的两个类型是不是已经在 factory 中注册过了，如果没有，那么就会注册。也就是说，要保证在 override 之前，这两种类型已经在 factory 的 m_types 数组中存在了。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_type_override_by_type

 815   // check for existing type override
```

```
816    foreach (m_type_overrides[index]) begin
817      if (m_type_overrides[index].orig_type == original_type ||
818          (m_type_overrides[index].orig_type_name != "<unknown>" &&
819           m_type_overrides[index].orig_type_name != "" &&
820           m_type_overrides[index].orig_type_name == original_type.get_type_name())) begin
821        string msg;
822        msg = {"Original object type '",original_type.get_type_name(),
823               "' already registered to produce '",
824               m_type_overrides[index].ovrd_type_name,"'"};
825        if (!replace) begin
826          msg = {msg, ".  Set 'replace' argument to replace the existing entry."};
827          uvm_report_info("TPREGD", msg, UVM_MEDIUM);
828          return;
829        end
830        msg = {msg, ".  Replacing with override to produce type '",
831                    override_type.get_type_name(),"'."};
832        uvm_report_info("TPREGR", msg, UVM_MEDIUM);
833        replaced = 1;
834        m_type_overrides[index].orig_type = original_type;
835        m_type_overrides[index].orig_type_name = original_type.get_type_name();
836        m_type_overrides[index].ovrd_type = override_type;
837        m_type_overrides[index].ovrd_type_name = override_type.get_type_name();
838      end
839    end
840
841    // make a new entry
842    if (!replaced) begin
843      uvm_factory_override override;
844      override = new(.orig_type(original_type),
845                     .orig_type_name(original_type.get_type_name()),
846                     .full_inst_path("*"),
847                     .ovrd_type(override_type));
848
849      m_type_overrides.push_back(override);
850    end
851
852  endfunction
```

816 行到 839 行则用于查看系统中已经有的 override 信息。当在整个验证平台中第一次调用 override 系列函数时，m_type_overrides 队列中是空的，所以暂且先跳过这几行。

842 到 850 行则用于向 m_type_overrides 队列中插入一条记录。这条记录会调用 uvm_factory_override 的 new 函数来创建。在调用 new 的时候会输入几个参数。其中的 original_type 即是 my_driver::get_type() 。get_type 位于 m_uvm_component_registry_internal 宏内：

```
文件：src/macros/uvm_object_defines.svh
类：无

 435 `define m_uvm_component_registry_internal(T,S) \
```

```
436     typedef uvm_component_registry #(T,`"S`") type_id; \
437     static function type_id get_type(); \
438       return type_id::get(); \
439     endfunction \
440     virtual function uvm_object_wrapper get_object_type(); \
441       return type_id::get(); \
442     endfunction
```

其返回值就是 type_id 的 get 函数的返回值，在我们的例子中就是 uvm_component_ registry#(my_driver, “my_driver”)中的静态成员变量 me。

original_type.get_type_name 即是上述静态成员变量的 get_type_name，在我们的例子中是”my_driver”。

override_type 即是 uvm_component_registry#(new_driver, “new_driver”)中的那个静态成员变量 me。

同时，传入的路径信息为*，表示整个 UVM 树中所有的结点都会执行 override 操作。

根据这些信息，生成一条记录，插入到 m_type_overrides 队列中。

842 到 850 行的插入新记录的动作只有在 replaced 为 0 的时候才会执行。replaced 是 793 行定义的一个变量，816 行到 839 行会对这个变量进行赋值，但是当第一次调用 override 系列函数时，这几行不会执行，所以执行到 842 行时，replaced 的值依然为初始值 0。

现在回过头来看 816 到 839 行 假设前面已经通过调用 set_type_override_by_type 函数用 new_driver 把 my_driver 给 override 掉了，现在再调用一次：

```
set_type_override_by_type(my_driver::get_type(), new_driver2::get_type());
```

由于前面已经调用过 override 系列函数，那么 m_type_overrides 队列中已经有了记录，所以 816 到 839 行会执行。816 到 820 行用于判断 m_type_overrides 是不是已经有了关于 my_driver 记录。在我们的例子中，m_type_overrides 中已经插入了这样的一条记录，所以 821 行到 837 行会执行。825 行会检查一下 replace 参数是否为 0，如果为 0，那么将会打印一条信息，直接返回。这个的意思就是说系统中已经有 new_driver 把 my_driver 给 override 了，如果想用 new_driver2 再 override，那么必须把 replace 位置为 1。而 replace 的默认值为 1，所以 825 到 829 的分支一般不会被执行。833 行用于把 replaced 变量置位，已经是已经进行 override 过了，那么 842 到 850 行就不会被执行，避免向 m_type_overrides 队列中插入多余的记录。834 到 837 行则用于更新 m_type_overrides 中的相关信息。

12.6.3. 类型被 override 时实例的创建

在某个 case 中调用了 override 系列函数：

```
set_type_override_by_type(my_driver::get_type(), new_driver::get_type());
```

那么我们来看当 drv = my_driver::type_id::create(“drv”, this);会发生什么事情：

首先这个 create 是属于 class uvm_component_registry #(my_driver, “my_driver”)的一个函数，其定义为;

```
文件：src/base/uvm_registry.svh
类：uvm_component_registry #(type T=uvm_component, string Tname="<unknown>")
函数/任务：create

108    static function T create(string name, uvm_component parent, string contxt="");
109      uvm_object obj;
110      uvm_factory f = uvm_factory::get();
111      if (contxt == "" && parent != null)
112        contxt = parent.get_full_name();
113      obj = f.create_component_by_type(get(),contxt,name,parent);
114      if (!$cast(create, obj)) begin
115        string msg;
116        msg = {"Factory did not return a component of type '",type_name,
117          "'. A component of type '",obj == null ? "null" : obj.get_type_name(),
118          "' was returned instead. Name=",name," Parent=",
119          parent==null?"null":parent.get_type_name()," contxt=",contxt};
120        uvm_report_fatal("FCTTYP", msg, UVM_NONE);
121      end
122    endfunction
```

111 行的条件符合，所以 contxt 将会被赋值为”uvm_test_done.env.agent”。

113 行会调用 uvm_factory 的 create_component_by_type 函数：

传入的第一个参数是 uvm_component_registry #(my_driver, “my_driver”)中的静态成员变量 me，第二个参数是 contxt，这里是”uvm_test_done.env.agent”，第三个参数是字符串”drv”，第四个参数则是 parent，这里是 this，指代 drv 的父 component，为一个 agent。

factory 的 create_component_by_type 函数如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：create_component_by_type

1149 function uvm_component uvm_factory::create_component_by_type (uvm_object_wrapper requ
ested_type,
```

```
1150                                                                    string parent_inst_path
="",
1151                                                                   string name,
1152                                                                    uvm_component paren
t);
1153    string full_inst_path;
1154
1155    if (parent_inst_path == "")
1156      full_inst_path = name;
1157    else if (name != "")
1158      full_inst_path = {parent_inst_path,".",name};
1159    else
1160      full_inst_path = parent_inst_path;
1161
1162    m_override_info.delete();
1163
1164    requested_type = find_override_by_type(requested_type, full_inst_path);
1165
1166    return requested_type.create_component(name, parent);
1167
1168  endfunction
```

1155 到 1160 的 if 语句中，将会执行第二个分支，1158 行会把 full_inst_path 的值置为”uvm_test_done.env.agent.drv”。

1162 行删除 m_override_info 中的信息，这个暂且先略过。

override 功能的关键发生在 1164 行，它会调用 find_override_by_type 函数。传入了两个参数，第一个参数就是传递给 create_component_by_type 的第一个参数，即 uvm_component_registry #(my_driver, “my_driver”)中的静态成员变量 me，第二个参数是 full_inst_path，这里为字符串”uvm_test_done.env.agent.drv”。

find_override_by_type 函数如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：find_override_by_type

1278 function uvm_object_wrapper uvm_factory::find_override_by_type(uvm_object_wrapper reques
ted_type,
1279                                                                    string full_inst_pat
h);
1280
1281    uvm_object_wrapper override;
1282    uvm_factory_queue_class qc = null;
1283    if (m_inst_override_queues.exists(requested_type))
1284      qc = m_inst_override_queues[requested_type];
1285
1286    foreach (m_override_info[index]) begin
1287      if ( //index != m_override_info.size()-1 &&
1288           m_override_info[index].orig_type == requested_type) begin
```

```
1289          uvm_report_error("OVRDLOOP", "Recursive loop detected while finding override.",
UVM_NONE);
1290          if (!m_debug_pass)
1291            debug_create_by_type (requested_type, full_inst_path);
1292
1293          return requested_type;
1294        end
1295      end
1296
1297      // inst override; return first match; takes precedence over type overrides
1298      if (full_inst_path != "" && qc != null)
1299        for (int index = 0; index < qc.queue.size(); ++index) begin
1300          if ((qc.queue[index].orig_type == requested_type ||
1301               (qc.queue[index].orig_type_name != "<unknown>" &&
1302                qc.queue[index].orig_type_name != "" &&
1303                qc.queue[index].orig_type_name == requested_type.get_type_name())) &&
1304              uvm_is_match(qc.queue[index].full_inst_path, full_inst_path)) begin
1305            m_override_info.push_back(qc.queue[index]);
1306            if (m_debug_pass) begin
1307              if (override == null) begin
1308                override = qc.queue[index].ovrd_type;
1309                qc.queue[index].selected = 1;
1310              end
1311            end
1312            else begin
1313              if (qc.queue[index].ovrd_type == requested_type)
1314                return requested_type;
1315              else
1316                return find_override_by_type(qc.queue[index].ovrd_type,full_inst_path);
1317            end
1318          end
1319        end
1320
```

这个函数比较长。1283 行与 1284 行是用于检查是否有具体的实例的 override。由于 factory 的 override 有两种，一种是整个验证平台中相关类的所有实例都进行 override，另外一种是只针对特定的实例进行 override。我们的例子中没有对具体的实例进行 override，另外 m_override_info 中的所有信息已经在 1162 行被删除了，所以 1283 到 1319 行之间的语句都不会执行。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：find_override_by_type

1321      // type override - exact match
1322      foreach (m_type_overrides[index]) begin
1323        if (m_type_overrides[index].orig_type == requested_type ||
1324            (m_type_overrides[index].orig_type_name != "<unknown>" &&
1325             m_type_overrides[index].orig_type_name != "" &&
1326             requested_type != null &&
1327             m_type_overrides[index].orig_type_name == requested_type.get_type_name())) begi
```

```
n
1328          m_override_info.push_back(m_type_overrides[index]);
1329          if (m_debug_pass) begin
1330            if (override == null) begin
1331              override = m_type_overrides[index].ovrd_type;
1332              m_type_overrides[index].selected = 1;
1333            end
1334          end
1335          else begin
1336            if (m_type_overrides[index].ovrd_type == requested_type)
1337              return requested_type;
1338            else
1339              return find_override_by_type(m_type_overrides[index].ovrd_type,full_inst_path);
1340          end
1341        end
1342      end
1343
1344      // type override with wildcard match
1345      //foreach (m_type_overrides[index])
1346      //   if (uvm_is_match(index,requested_type.get_type_name())) begin
1347      //      m_override_info.push_back(m_inst_overrides[index]);
1348      //      return find_override_by_type(m_type_overrides[index],full_inst_path);
1349      //   end
1350
1351      if (m_debug_pass && override != null)
1352        if (override == requested_type)
1353          return requested_type;
1354        else
1355          return find_override_by_type(override,full_inst_path);
1356
1357      return requested_type;
1358
1359  endfunction
```

1322 到 1342 行才是这个例子中会用到的。1323 行检查 m_type_overrides 中是否有对应 my_driver 的记录。恰好找到了一条，把这条记录放入 m_override_info 中。这条记录中的 orig_type 为 my_driver 类型，而 ovrd_type 为 new_driver 类型。1329 到 1334 行的这个分支是用于 debug 时才会用到的，因此在这里不会执行。1336 到 1340 这个分支中，1336 行检查 m_type_overrides 中找到的这条记录中 ovrd_type 是不是就是要查找的类型，在本例中就相当于是看 new_driver 是不是等于 my_driver，很显然是不等于的，所以会调用 find_override_by_type 函数，这相当于是递归调用，只是这次调用传入的第一个参数将会是代表 new_driver 的 uvm_component_registry #(new_driver, “new_driver”)的静态成员变量 me，第二个参数是字符串 my_driver。

这一次调用时 1283 到 1284 行与 1298 到 1319 行依然不会执行。但是由于此时 m_override_info 中已经有了记录，所以 1286 到 1295 行会执行。这几行的目的主要是为了防止形成环路的 override，即用 B 把 A 给 override，而用 C 把 B 给 override 了，接下来又用 A 把 C 给 override 了。这种情况是不允许的。

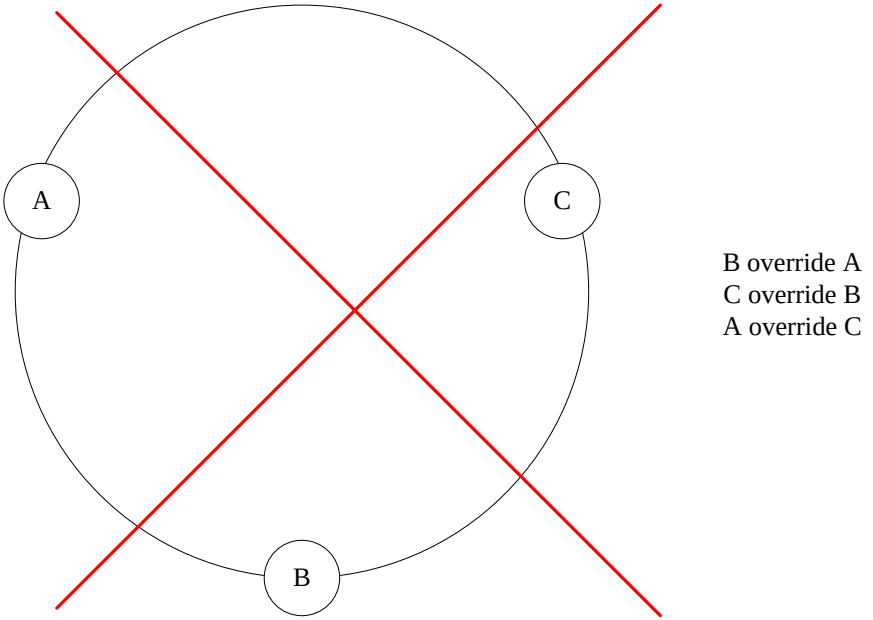


图 12-1 形成环路的 override

由于这次传递进来的是 new_driver 类型，m_type_overrides 中没有相关的记录，1322 行到 1327 行不会被匹配到，所以 1322 行到 1342 会将会被直接跳过。由于这是非 debug 模式，所以 1351 到 1355 行将不会被执行。于是系统会直接返回 new_driver 类型，即 uvm_component_registry #(new_driver, “new_driver”)的静态成员变量 me。

这个 find_override_by_type 函数为什么要做成递归的方式呢？因为假设系统中有一个 A，结果使用 override 语句把 A 用 B 给 override 了，而后面又用 override 语句把 B 用 C 给 override 了，此时创建 A 的实例，得到的应该是 C 的实例。只有在 find_override_by_type 中递归调用，才能完整的实现这一功能。

现在回到 create_component_by_type 函数。第 1164 行返回了 find_override_by_type 的结果。后面依据这个结果，调用 create_component 函数创建实例。如果在没有 override，那么最终会调用 uvm_component_registry #(my_driver, “my_driver”)::me.create_component 创建一个 my_driver 类型的实例，而现在 my_driver 被 new_driver 给 override 了，所以最终会调用 uvm_component_registry #(new_driver, “new_driver”)::me.create_component 创建一个 new_driver 类型的实例。也就是说 drv = my_driver::type_id::create(“drv”, this);最终得到的 drv 将会是一个 new_driver 类型的实例。这恰好就是我们预期的 override 功能。

12.6.4. set_type_override_by_name

前面三节讲述了 set_type_override_by_type 功能。除了这一功能外，UVM 中还提供其它的 override 方式。本节讲述 set_type_override_by_name。

uvm_component 中有函数 set_type_override：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：set_type_override

2087 function void uvm_component::set_type_override (string original_type_name,
2088                                                  string override_type_name,
2089                                                  bit    replace=1);
2090   factory.set_type_override_by_name(original_type_name,
2091                                     override_type_name, replace);
2092 endfunction
```

这个函数最终会调用 uvm_factory 的 set_type_override_by_name 函数。如果我们在某个 case 中有如下语句：

```
set_type_override("my_driver", "new_driver");
```

那么接下来会调用：

```
factory.set_type_override_by_name("my_driver", "new_driver", 1);
```

这个函数的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_type_override_by_name

 858 function void uvm_factory::set_type_override_by_name (string original_type_name,
 859                                                       string override_type_name,
 860                                                       bit replace=1);
 861   bit replaced;
 862
 863   uvm_object_wrapper original_type=null;
 864   uvm_object_wrapper override_type=null;
 865
 866   if(m_type_names.exists(original_type_name))
 867     original_type = m_type_names[original_type_name];
 868
 869   if(m_type_names.exists(override_type_name))
 870     override_type = m_type_names[override_type_name];
 871
 872   // check that type is registered with the factory
 873   if (override_type == null) begin
```

```
 874          uvm_report_error("TYPNTF", {"Cannot register override for original type '",
 875          original_type_name,"' because the override type '",
 876          override_type_name, "' is not registered with the factory."}, UVM_NONE);
 877        return;
 878      end
 879
 880      // check that old and new are not the same
 881      if (original_type_name == override_type_name) begin
 882          uvm_report_warning("TYPDUP", {"Requested and actual type name ",
 883          " arguments are identical: ",original_type_name,". Ignoring this override."}, UVM_N
ONE);
 884        return;
 885      end
 886
```

866 到 885 行比较简单，就是查看一下 new_driver 是不是已经在 factory 中注册过了，以及 original_type_name 与 ovrride_type_name 是否一样，这里就是 my_driver 与 new_driver 是否一样。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_type_override_by_name

 887      foreach (m_type_overrides[index]) begin
 888        if (m_type_overrides[index].orig_type_name == original_type_name) begin
 889          if (!replace) begin
 890            uvm_report_info("TPREGD", {"Original type '",original_type_name,
 891              "' already registered to produce '",m_type_overrides[index].ovrd_type_name,
 892              "'.  Set 'replace' argument to replace the existing entry."}, UVM_MEDIUM);
 893            return;
 894          end
 895          uvm_report_info("TPREGR", {"Original object type '",original_type_name,
 896            "' already registered to produce '",m_type_overrides[index].ovrd_type_name,
 897            "'.  Replacing with override to produce type '",override_type_name,"'."}, UVM_M
EDIUM);
 898          replaced = 1;
 899          m_type_overrides[index].ovrd_type = override_type;
 900          m_type_overrides[index].ovrd_type_name = override_type_name;
 901        end
 902      end
 903
 904      if (original_type == null)
 905        m_lookup_strs[original_type_name] = 1;
 906
 907      if (!replaced) begin
 908        uvm_factory_override override;
 909        override = new(.orig_type(original_type),
 910                       .orig_type_name(original_type_name),
 911                       .full_inst_path("*"),
 912                       .ovrd_type(override_type));
 913
```

```
914         m_type_overrides.push_back(override);
915 //        m_type_names[original_type_name] = override.ovrd_type;
916     end
917
918 endfunction
```

887 行到 902 行以及 907 行到 916 行与 set_type_override_by_type 的相关部分完全一样，同样的会在 m_type_overrides 中插入一条用 new_driver 替换 my_driver 的记录，这里不重复阐述。这里比较有趣的是 904 到 905 行，意思就是当 my_driver 没有在 factory 中注册过的时候，在联合数组中插入一条记录：m_lookup_strs[“my_driver”] = 1。搜索整个 uvm_factory.svh 文件，可以看到这个联合数组只有在 m_debug_create 时才会用到，即它只是用于 debug 功能的，因此这里不多做阐述。

可见，set_type_overide_by_name 与 set_type_override_by_type 几乎完全一样，其唯一的区别就是传入的参数有差异。

12.6.5. set_inst_override_by_type

uvm_component 中提供如下函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：set_inst_override_by_type

2128 function void uvm_component::set_inst_override_by_type (string relative_inst_path,
2129                                                          uvm_object_wrapper original_type,
2130                                                          uvm_object_wrapper override_type);
2131     string full_inst_path;
2132
2133     if (relative_inst_path == "")
2134       full_inst_path = get_full_name();
2135     else
2136       full_inst_path = {get_full_name(), ".", relative_inst_path};
2137
2138     factory.set_inst_override_by_type(original_type, override_type, full_inst_path);
2139
2140 endfunction
```

这个函数用于实现对具体的某一个实例进行 override。

设要替换的实例的路径为：env.agent.drv，则在某 case 的 build_pahse 进行如下的 override：

```
set_inst_override_by_type(“env.agent.drv”, my_driver::get_type(), new_driver::get_type ());
```

在 case 的 build_phase 运行 get_full_name 将会得到的是”uvm_test_done”字符串，所以最终会调用：

```
factory.set_inst_override_by_type(my_driver::get_type(), new_driver::get_type(), “uvm_test_done.env.agent. drv”);
```

set_inst_override_by_type 函数的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_inst_override_by_type

 954 function void uvm_factory::set_inst_override_by_type (uvm_object_wrapper original_type,
 955                                                       uvm_object_wrapper override_type,
 956                                                       string full_inst_path);
 957
 958   uvm_factory_override override;
 959
 960   // register the types if not already done so
 961   if (!m_types.exists(original_type))
 962     register(original_type);
 963
 964   if (!m_types.exists(override_type))
 965     register(override_type);
 966
 967   if (check_inst_override_exists(original_type,override_type,full_inst_path))
 968     return;
 969
 970   if(!m_inst_override_queues.exists(original_type))
 971     m_inst_override_queues[original_type] = new;
 972
 973   override = new(.full_inst_path(full_inst_path),
 974                  .orig_type(original_type),
 975                  .orig_type_name(original_type.get_type_name()),
 976                  .ovrd_type(override_type));
 977
 978
 979   m_inst_override_queues[original_type].queue.push_back(override);
 980
 981 endfunction
```

961 到 965 行用于保证 my_driver 和 new_driver 在 factory 中注册过了。

967 行会调用 check_inst_override_exists 函数：

```
check_inst_override_exists(my_driver::get_type(), new_driver::get_type(), “uvm_test_done. env.agent. drv”);
```

这个函数的定义如下：

```
文件：src/base/uvm_factory.svh
```

```
类：uvm_factory
函数/任务：check_inst_override_exists

 923 function bit uvm_factory::check_inst_override_exists (uvm_object_wrapper original_type,
 924                                                       uvm_object_wrapper override_type,
 925                                                       string full_inst_path);
 926   uvm_factory_override override;
 927   uvm_factory_queue_class qc;
 928
 929   if (m_inst_override_queues.exists(original_type))
 930     qc = m_inst_override_queues[original_type];
 931   else
 932     return 0;
 933
 934   for (int index=0; index<qc.queue.size(); ++index) begin
 935
 936     override = qc.queue[index];
 937     if (override.full_inst_path == full_inst_path &&
 938         override.orig_type == original_type &&
 939         override.ovrd_type == override_type &&
 940         override.orig_type_name == original_type.get_type_name()) begin
 941     uvm_report_info("DUPOVRD",{"Instance override for '",
 942       original_type.get_type_name(),"' already exists: override type '",
 943       override_type.get_type_name(),"' with full_inst_path '",
 944       full_inst_path,"'"},UVM_HIGH);
 945       return 1;
 946     end
 947   end
 948   return 0;
 949 endfunction
```

这个函数用到了 m_inst_override_queue，其定义为：

```
文件：src/base/uvm_factory.svh
类：uvm_factory

 311   protected uvm_factory_queue_class m_inst_override_queues[uvm_object_wrapper];
```

这是一个联合数组，其索引是 uvm_object_wrapper 类型的，也即我们前面说的 uvm_object_registry 或者 uvm_component_registry 类型，而其内容则是 uvm_factory_queue_ class 类型：

```
文件：src/base/uvm_factory.svh
类：uvm_factory_queue_class

  30 class uvm_factory_queue_class;
  31   uvm_factory_override queue[$];
  32 endclass
```

可见，这只是一个简单的队列，也就是说一条 m_inst_override_queues 对应着一个队列，这个队列是与具体的 uvm_object_registry 或者 uvm_component_registry 的变

量一一对应。

回到 check_inst_override_exists 中来，此时 929 行会检查 m_inst_override_queues 是否存在着与 my_driver 相对应的一条记录。很明显这里是没有的，所以会直接返回 0。如果跳开我们的这个例子，假设 m_inst_override_queues 中有与 my_driver 相对应的一条记录，那么把这条记录取出，然后 934 到 947 会检查这条记录相应的队列中是否有与我们正在做的动作完全相同的一条记录。

换句话说，假如我们在 case 的 build_phase 中调用了两次 set_inst_override_by_type()，且给出的参数完全一样，那么在第二次调用的时候会给出信息说明已经 override 过了。

check_inst_override_exists 函数返回为 0 时，代表之前没有进行过 override，而 1 则代表之前已经进行过一次完全相同的 override。我们的例子是返回 0 的。

回到 set_inst_override_by_type 中来，967 行如果检测到系统中已经进行过一次相同的 override，那么就直接返回。970 行到 971 行则用于检查 m_inst_override_queues 是否有对应 my_driver 的一条记录。如果没有的话，那么就新建一条。973 行到 979 行用于新建一条 override 信息，并且把这条信息插入到 m_inst_override_queues 相应记录的队列中。

从上面可以看出，m_inst_override_queues 中的记录是按类型存放的，假如系统中有两个 my_driver 的实例，我们执行如下的操作：

```
set_inst_override_by_type("env.agent1.drv", my_driver::get_type(), new_driver::get_type ());
set_inst_override_by_type("env.agent2.drv", my_driver::get_type(), new_driver::get_type ());
```

那么在执行第一次 override 时，m_inst_override_queues 中还没有与 my_driver 相关的记录，于是就新建了一条，并把与 agent1.drv 相关的 override 信息放入这条记录的队列中（如图所示）。当执行第二次的 override 时，m_inst_override_queues 中已经有了与 my_driver 相对应的记录，所以就直接把一条与 agent2.drv 相关的 override 信息放入这条记录的队列中（如图所示）

12.6.6. 实例被 override 时实例的创建

12.6.3 节讲述了类型被 override 时，一个实例的创建过程。当时在讲述 find_override_by_type 时，特意略过了 factory 中有实例被 override 的记录的情况。

假如在某个 case 中调用了如下函数：

```
set_inst_override_by_type("env.agent.drv", my_driver::get_type(), new_driver::get_type ());
```

那么 m_inst_override_queues 中就有了一条与 my_driver 相关的记录。

而在 agent 的 build_phase 使用 factory 的功能创建一个实例时，最终会调用 factory 的 create_component_by_type 函数，而这个函数又会调用：

```
find_override_by_type(uvm_component_registry#(my_driver, “my_driver”)::me, “uvm_test_done.env.agent.drv”);
```

这个函数已经在 12.6.3 节已经列出了，因此这里不另外列出。

1283 行时会判断 m_inst_override_queues 中是否有与 my_driver 相关的记录。这里找到了一条，于是令 qc 指向这条记录。

find_override_by_type 是一个递归调用的函数，而在第一次调用的时候，m_override_info 中是空的，所以 1286 到 1295 行跳过。

由于满足 1298 行的条件，所以接下来程序会进入 1299 行的分支。我们的 qc 中只有一条记录（用 new_driver override my_driver 的记录），1300 行的条件满足，于是这条记录被放放 m_override_info 中。

1306 到 1311 行的分支依然是 debug 功能时才会用到。

1313 到 1316 的语句将会执行 1316 的分支此时调用传入的参数时：

find_override_by_type(uvm_component_registry#(new_driver, “new_driver”)::me, “uvm_test_done.env.agent.drv”);

第二次调用这个函数时，1284 行不会被执行，因为 m_inst_override_queues 中没有与 new_driver 相对应的记录。所以 qc 的值依然为 null。

1286 到 1295 行会执行到，11.6.3 节已经说过，这几行主要是为了防止递归的 override。本例中没有这种情况，所以直接跳过。

由于 qc 的值依然为 null，所以 1298 到 1319 行不会被执行。

1322 到 1342 行是与类型 override 相关的，在本节的例子中 m_type_overrides 中没有记录，所以会直接跳过。

由于是处于非 debug 模式，所以 1351 行到 1355 行依然不会被执行。

最后 1357 行直接返回 new_driver 类型。

上面说的是恰好被 override 的实例进行实例化的情况，即我们设置了 env.agent.drv 的 driver 进行 override，在 env.agent 中进行 drv 实例化的情况。假如在 env.agent2 中同样有一个 my_driver 类型的 drv 要进行实例化，那会发生什么情况呢？此时 find_override_by_type 依然会被调用：

```
find_override_by_type(uvm_component_registry#(my_driver, “my_driver”)::me, “uvm_test_done.env.agent2. drv”);
```

此时传入的参数发生了变化，注意到这个函数的 1304 行会对路径进行检查，此时 m_inst_override_queues 相应记录的队列中的那条记录的路径是

“uvm_test_done.env.agent. drv”，所以不匹配，于是 1305 到 1318 的分支不会被执行。系统会直接跳到 1357 行返回 my_driver 类型。这种行为恰好是我们所期待的。

12.6.7. set_inst_override_by_name 函数

uvm_component 中提供 set_inst_override 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：set_inst_override


2108 function void  uvm_component::set_inst_override (string relative_inst_path,
2109                                                   string original_type_name,
2110                                                   string override_type_name);
2111   string full_inst_path;
2112
2113   if (relative_inst_path == "")
2114     full_inst_path = get_full_name();
2115   else
2116     full_inst_path = {get_full_name(), ".", relative_inst_path};
2117
2118   factory.set_inst_override_by_name(
2119                                     original_type_name,
2120                                     override_type_name,
2121                                     full_inst_path);
2122 endfunction
```

在某个 case 的 build_phase 中进行如下的 override：

```
set_inst_override(“env.agent.drv”, “my_driver”, “new_driver”);
```

那么最终会调用:

```
factory.set_inst_override_by_name(“my_driver”, “new_driver”, “uvm_test_done.env.agent. drv”);
```

这个函数的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_inst_override_by_name

 987 function void uvm_factory::set_inst_override_by_name (string original_type_name,
 988                                                       string override_type_name,
 989                                                       string full_inst_path);
 990
 991   uvm_factory_override override;
```

```
 992    uvm_object_wrapper original_type=null;
 993    uvm_object_wrapper override_type=null;
 994
 995    if(m_type_names.exists(original_type_name))
 996      original_type = m_type_names[original_type_name];
 997
 998    if(m_type_names.exists(override_type_name))
 999      override_type = m_type_names[override_type_name];
1000
1001    // check that type is registered with the factory
1002    if (override_type == null) begin
1003      uvm_report_error("TYPNTF", {"Cannot register instance override with type name '",
1004      original_type_name,"' and instance path '",full_inst_path,"' because the type it's suppose
d ",
1005      "to produce, '",override_type_name,"', is not registered with the factory."}, UVM_NON
E);
1006      return;
1007    end
1008
1009    if (original_type == null)
1010        m_lookup_strs[original_type_name] = 1;
1011
1012    override = new(.full_inst_path(full_inst_path),
1013                   .orig_type(original_type),
1014                   .orig_type_name(original_type_name),
1015                   .ovrd_type(override_type));
1016
```

995 行到 999 行用于从 factory 的注册信息中提取与 my_driver 和 new_driver 相关的记录。1002 到 1007 行用于保证 new_driver 已经在 factory 中注册过了。

1009 到 1010 行则主要是用于 debug 功能，前面在讲述 set_type_override_by_name 时已经有描述。

1012 到 1015 行用于新建一条 override 记录。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：set_inst_override_by_name

1017    if(original_type != null) begin
1018      if (check_inst_override_exists(original_type,override_type,full_inst_path))
1019        return;
1020      if(!m_inst_override_queues.exists(original_type))
1021        m_inst_override_queues[original_type] = new;
1022      m_inst_override_queues[original_type].queue.push_back(override);
1023    end
1024    else begin
1025      if(m_has_wildcard(original_type_name)) begin
1026        foreach(m_type_names[i]) begin
1027          if(uvm_is_match(original_type_name,i)) begin
1028            this.set_inst_override_by_name(i, override_type_name, full_inst_path);
```

```
1029                end
1030              end
1031              m_wildcard_inst_overrides.push_back(override);
1032          end
1033          else  begin
1034            if(!m_inst_override_name_queues.exists(original_type_name))
1035              m_inst_override_name_queues[original_type_name]  =  new;
1036            m_inst_override_name_queues[original_type_name].queue.push_back(override);
1037          end
1038      end
1039
1040  endfunction
```

1017 到 1038 行是这个函数的主体部分，在我们的例子中会执行 1018 到 1023 行的分支。1018 行检查之前系统是否已经有过一次完全相同的 override 了，有的话会直接返回。1020 到 1022 行会把刚才新建的 override 记录信息放入 m_inst_override_queues 相应记录的队列中。1025 到 1037 行的分支只有在 original_type 为 null 的时候才会执行。original_type 为什么可能为 null 呢？

假如在 case 中如下两种 override：

```
set_inst_override("env.agent.drv",  "my_driv",  "new_driver");
```

或：

```
set_inst_override("env.agent.drv",  "my_driv*",  "new_driver");
```

那么 original_type 就会为 null，因为系统中不存在 my_driv 或者 my_driv*的类型。对于 my_driv*，会执行 1025 到 1032 的分支，系统发现这是个通配符之后，最终会匹配到 my_driver，然后剩下的事情就跟前面很像了，调用 set_inst_override_by_name，这次调用给出的是完整的名字，所以会执行 1018 到 1022 的分支。之后 1031 行会往 m_wildcard_inst_overrides 队列中插入一条 override 记录。

对于 my_driv 则比较复杂，会执行 1034 到 1036 行，往 m_inst_override_name_queues 中对应的队列中插入一条记录。关于这一点，下节将会介绍。

12.6.8. find_override_by_name 函数

假设在 env.agent 中，drv 是 my_driver 类型的，使用如下方式创建：

```
drv  =  factory.create_component_by_name("my_driv",  get_full_name(),  "drv",  this);
```

而在某个 case 中使用了如下的语句：

```
set_inst_override("env.agent.drv", "my_driv", "new_driver");
```

在这个前提下，我们分析这个函数。create_component_by_name 最终会调用 find_override_by_name：

```
find_override_by_name("my_driv", "uvm_test_done.env.agent.drv");
```

这个函数的定义如下：

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：find_override_by_name

1189 function uvm_object_wrapper uvm_factory::find_override_by_name (string requested_type_na
me,
1190                                                                  string full_inst_pa
th);
1191    uvm_object_wrapper rtype = null;
1192    uvm_factory_queue_class qc;
1193
1194    uvm_object_wrapper override;
1195
1196    if (m_type_names.exists(requested_type_name))
1197      rtype = m_type_names[requested_type_name];
1198
1199 /***
1200    if(rtype == null) begin
1201      if(requested_type_name != "") begin
1202        uvm_report_warning("TYPNTF", {"Requested type name ",
1203            requested_type_name, " is not registered with the factory. The instance override t
o ",
1204            full_inst_path, " is ignored"}, UVM_NONE);
1205      end
1206      m_lookup_strs[requested_type_name] = 1;
1207      return null;
1208    end
1209 ***/
1210
```

1196 行与 1197 行查看 factory 中是否已经注册了与输入的参数相匹配的类型。这里由于输入了"my_driv"，所以 rtype 为 null。

```
文件：src/base/uvm_factory.svh
类：uvm_factory
函数/任务：find_override_by_name

1211    if (full_inst_path != "") begin
1212      if(rtype == null) begin
1213        if(m_inst_override_name_queues.exists(requested_type_name))
1214          qc = m_inst_override_name_queues[requested_type_name];
1215      end
1216      else begin
```

```
1217          if(m_inst_override_queues.exists(rtype))
1218            qc = m_inst_override_queues[rtype];
1219        end
1220        if(qc != null)
1221          for(int index = 0; index<qc.queue.size(); ++index) begin
1222            if (uvm_is_match(qc.queue[index].orig_type_name, requested_type_name) &&
1223                uvm_is_match(qc.queue[index].full_inst_path, full_inst_path)) begin
1224              m_override_info.push_back(qc.queue[index]);
1225              if (m_debug_pass) begin
1226                if (override == null) begin
1227                  override = qc.queue[index].ovrd_type;
1228                  qc.queue[index].selected = 1;
1229                end
1230              end
1231              else begin
1232                if (qc.queue[index].ovrd_type.get_type_name() == requested_type_name)
1233                  return qc.queue[index].ovrd_type;
1234                else
1235                  return find_override_by_type(qc.queue[index].ovrd_type,full_inst_path);
1236              end
1237            end
1238          end
1239      end
          …
1272  endfunction
```

1211 行则根据输入的 full_inst_path 来判断。假设我们输入的是非空的，所以 1212 到 1238 的这个分支将会执行。

1212 到 1219 行的语句将会执行 1213 与 1214 的分支。由于 m_inst_override_name_queues 中已经插入了一条用”new_driver”替换”my_driv”的记录，所以 qc 将会指向这条记录所在的队列。这个队列中除了这一条记录外，再无其它记录。

由于条件都满足，接下来会进入到 1224 行，m_override_info 中将会插入一条用 new_driver 替换 my_driv 的记录。

1225 到 1230 行是用于 debug 的，直接跳过。

1232 到 1235 的语句将会执行 1235 行的分支，即直接调用函数：

```
find_override_by_type(uvm_component_registry#(new_driver, “new_driver”), “uvm_test_ done.env.ag
ent.drv”);
```

根据前面我们章节，这会直接返回 new_driver 类型。所以

```
drv = factory.create_component_by_name(“my_driv”, get_full_name(), “drv”, this);
```

这句话最终将会产生一个 new_driver 类型的实例。

12.6.9. override 功能总结

set_type_override_by_type 的参数是 wrapper 型的变量，它会向 m_type_overrides 中插入一条记录。

set_type_override 的参数是字符串，它会向 m_type_overrides 中插入一条记录。如果输入的 original_name 中有通配符或者名字是不完整的，那么依然会插入一条记录，只是记录中的 original_type 是 null。也就是说，它其实是不支持正则表达式及不完整的名字的。

set_inst_override_by_type 的参数是 wrapper 型的变量，它会向 m_inst_override_queues 中插入一条记录。

set_inst_override 的参数是字符串，如果字符串是完整的，那么它会向 m_inst_override_ queues 中插入一条记录；如果字符串中有通配符，那么将会在 m_wildchard_inst_overrides 中插入一条记录，并查找所有可能匹配的类型，在 m_inst_override_queues 中插入一条记录；如果字符串是不完整的字符串，那么将会在 m_inst_override_name_queues 中插入一条记录，之后通过名字查找时可以找出匹配的记录。

13. phase 机制源代码分析

phase 机制是 uvm_component 拥有的两大特性之一，也是 UVM 中非常重要的一个功能。自 UVM1.0 以来，引入了 domain 和运行时 phase 的概念，更是大大扩展了 phase 的易用性。

本章前三节讲述基本的 phase 机制，domain 等，第四节讲述 objection 机制，第五节讲述 phase 机制的高级应用。

13.1. 探索 phase

一直以来，UVM 都在告诉我们一个事实，phase 是自动执行的。那现在我们就来看一下，phase 是如何自动执行的。

13.1.1. 从 run_test 说起

10.2.4 节在介绍 uvm_root 的时候，曾经提起过这函数：

文件：src/base/uvm_root.svh
类：uvm_root

```
函数/任务：run_test

391    // phase runner, isolated from calling process
392    fork begin
393       // spawn the phase runner task
394       phase_runner_proc = process::self();
395       uvm_phase::m_run_phases();
396    end
397    join_none
398    #0; // let the phase runner start
399
400    wait (m_phase_all_done == 1);
401
402    // clean up after ourselves
403    phase_runner_proc.kill();
404
405    report_summarize();
406
407    if (finish_on_completion)
408       $finish;
409
410 endtask
```

当时对于 392 行以后的内容一笔带过。这里回过头来，再仔细看一下。

392 行到 397 行是一个 fork join_none 语句。394 行得到此进程的句柄（指针）。这里用到了 process，process 是 systemverilog 中的（而不是 UVM 中的）一个内建的类，其定义为：

```
class process;
    enum state {FINISHED, RUNNING, WAITING, SUSPENDED, KILLED};

    static function process self();
    function state status();
    task kill();
    task await();
    task suspend();
    task resume();
endclass
```

进程类型的对象在进程产生的时候在内部产生。用户不能够产生进程类型的对象；调用 new 不会产生一个新的进程，相反，它还会导致一个错误。进程类不能被扩展。对进程类的扩展会导致一个编译错误。self()函数返回当前进程的句柄，也就是说，一个指向进行调用的进程的句柄。394 行正是使用 self 函数得到了 fork join_none 这个进程的句柄。得到这个句柄之后，就可以通过控制句柄来对进程进行控制，如 403 行就调用了这个句柄的 kill 函数，用于杀死整个 fork join_none 进程。

395 行调用 uvm_phase 类的静态函数 m_run_phases。

13.1.2. m_run_phases 函数

这个函数是 uvm_phase 类的一个静态函数，其定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数：m_run_phases

 579    extern static task m_run_phases();

1798 task uvm_phase::m_run_phases();
1799    uvm_root top = uvm_root::get();
1800
1801    // initiate by starting first phase in common domain
1802    begin
1803       uvm_phase ph = uvm_domain::get_common_domain();
1804       void'(m_phase_hopper.try_put(ph));
1805    end
1806
1807    forever begin
1808       uvm_phase phase;
1809       uvm_process proc;
1810       m_phase_hopper.get(phase);
1811       fork
1812          begin
1813             proc = new(process::self());
1814             phase.execute_phase();
1815          end
1816       join_none
1817       m_phase_top_procs[phase] = proc;
1818       #0;  // let the process start running
1819    end
1820 endtask
```

1803 行通过静态函数得到了一个 uvm_phase 类型的变量，之后尝试着把这个变量放入到 m_phase_hooper 中。关于这个静态函数，我们先稍微放一下。

m_phase_hooper 是一个静态的成员变量，其定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

 575    local static mailbox #(uvm_phase) m_phase_hopper = new();
```

这是一个 mailbox，专门用于传递 uvm_phase 类型的变量。

1807 到 1819 行是一个无限循环。这几行中用到了一个 uvm_process 类，这个类的定义如下：

文件：src/base/uvm_phase.svh
类：uvm_process

```
 36 class uvm_process;
 37
 38   protected process m_process_id;
 39
 40   function new(process pid);
 41     m_process_id = pid;
 42   endfunction
 43
 44   function process self();
 45     return m_process_id;
 46   endfunction
 47
 48   virtual function void kill();
 49     m_process_id.kill();
 50   endfunction
 51
 52 `ifdef UVM_USE_FPC
 53   virtual function process::state status();
 54     return m_process_id.status();
 55   endfunction
 56
 57   task await();
 58     m_process_id.await();
 59   endtask
 60
 61   task suspend();
 62    m_process_id.suspend();
 63   endtask
 64
 65   function void resume();
 66    m_process_id.resume();
 67   endfunction
 68 `else
 69   virtual function int status();
 70     return m_process_id.status();
 71   endfunction
 72 `endif
 73
 74 endclass
```

这个类比较简单，纯粹是为了封装 processs 类而设置的。其内部有一个 process_id。简单点来说，大家可以把它与 process 类当成等价的就可以了。

1810 行从 m_phase_hopper 取得信件，如果取不到，就一直阻塞在那里，如果取到了，那么就启动一个 fork 进程，在这个进程中，首先得到这个进程的句柄，之后就是调用 phase. execute_phase。1817 行则把刚刚得到的进程的句柄放入一个静态的联合数组 m_phase_top_ procs 中。这个联合数组的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

 576    local static uvm_process m_phase_top_procs[uvm_phase];
```

这个联合数组的索引为 uvm_phase 类型的变量，而存储的内容为 uvm_process 类型的。

1814 行的 execute_phase 从名字来看相当简单，也能明白其大概意思。所以整个函数看下来，关键点就在于 1803 行的那个 get_common_domain 到底返回的是什么。下一节将从这个函数展开。

13.2. 初识 uvm_domain

13.2.1. get_common_domain

这是一个静态函数，其定义如下：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：get_common_domain

100    static function uvm_domain get_common_domain();
101
102      uvm_domain domain;
103      uvm_phase schedule;
104
105      if (m_common_domain != null)
106        return m_common_domain;
107
```

这个函数相对来说并不复杂，里面的语句都是类似 add，find 之类的，没有涉及到一些古怪的算法之类的东西，所以相对来说比较容易分析。

105 行提到了 m_common_domain：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
```

```
67    static local uvm_domain m_common_domain;
```

这是一个 uvm_domain 类的静态成员变量。105 行判断这个变量是否为 null，由于这个变量是私有类型的，也就是说只有这个类的成员函数才能修改它。搜索整个类，只在 get_common_domain 里才能见到对这个变量的修改，因此，可以判定，当 get_common_domain 第一次调用的时候，这个变量一定是 null。

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：get_common_domain

108       domain = new("common");
109       domain.add(uvm_build_phase::get());
110       domain.add(uvm_connect_phase::get());
111       domain.add(uvm_end_of_elaboration_phase::get());
112       domain.add(uvm_start_of_simulation_phase::get());
113       domain.add(uvm_run_phase::get());
114       domain.add(uvm_extract_phase::get());
115       domain.add(uvm_check_phase::get());
116       domain.add(uvm_report_phase::get());
117       domain.add(uvm_final_phase::get());
118       m_domains["common"] = domain;
119
```

108 行到 117 行之间实例化了一个 uvm_domain 类型的成员变量，并且把我们所熟知的几大函数 phase 加入到这个 uvm_domain 中。118 行用到了 m_domains，这是一个静态联合数组：

```
文件：src/base/uvm_domain.svh
类：uvm_domain

69    static local uvm_domain m_domains[string];
```

这个联合数组的索引是 string 类型的，而内容则是 uvm_domain 类型的。118 行往这个数组中插入一条记录，记录的索引是 common，而内容则是刚刚实例化的 uvm_domain。

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：get_common_domain

120       // for backward compatibility, make common phases visible;
121       // same as uvm_<name>_phase::get().
122       build_ph                 = domain.find(uvm_build_phase::get());
123       connect_ph               = domain.find(uvm_connect_phase::get());
124       end_of_elaboration_ph    = domain.find(uvm_end_of_elaboration_phase::get());
125       start_of_simulation_ph = domain.find(uvm_start_of_simulation_phase::get());
126       run_ph                   = domain.find(uvm_run_phase::get());
127       extract_ph               = domain.find(uvm_extract_phase::get());
128       check_ph                 = domain.find(uvm_check_phase::get());
```

```
129       report_ph                    = domain.find(uvm_report_phase::get());
130       m_common_domain  =  domain;
131
132       domain  =  get_uvm_domain();
133       m_common_domain.add(domain,
134                                .with_phase(m_common_domain.find(uvm_run_phase::get())));
135
136
137       return  m_common_domain;
138
139    endfunction
```

122 到 129 行可以直接忽略，它是为了与老的 OVM 兼容。

最精彩的地方出现在 130 行，直接把我们刚刚实例化的成员变量赋值给了 m_common_ domain。所以，现在我们可以大体上判定，get_common_domain 中返回的就是一个插入了 build_phase，connect_phase 等各种 phase 的 uvm_domain。

不过，到目前为止，m_common_domain 中并没有加入 12 个动态运行的 phase。132 到 134 行就是做这件事情的。132 行通过调用 get_uvm_domain 函数来得到这些运行时的 phase，之后 133 和 134 行这把 12 个动态运行时的 phase 与 run_phase 并行的加入了 m_common_domain 中。这个 add 函数后面会专门的介绍。不过大体上到现在为止整个函数的意思应该比较明了。

13.2.2. get_uvm_domain

上面介绍到通过调用 get_uvm_domain 得到了 12 个动态运行的 phase，这里重点介绍下这个函数。函数的定义如下：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：get_uvm_domain

168    static  function  uvm_domain  get_uvm_domain();
169
170       if  (m_uvm_domain  ==  null)  begin
171          m_uvm_domain  =  new("uvm");
172          m_uvm_schedule  =  new("uvm_sched",  UVM_PHASE_SCHEDULE);
173          add_uvm_phases(m_uvm_schedule);
174          m_uvm_domain.add(m_uvm_schedule);
175       end
176       return  m_uvm_domain;
177    endfunction
```

170 行判断 m_uvm_domain 的值是否为 null。这个变量也是一个静态变量：

```
文件：src/base/uvm_domain.svh
类：uvm_domain

 68    static local uvm_domain m_uvm_domain; // run-time phases
```

变量的类型是 uvm_domain 类型的，而且是 local 的，整个 uvm_domain 类中只有 get_uvm_domain 函数会对这个变量进行修改。所以其赋值过程一定是在 get_uvm_domain 中完成的。第一次运行 get_uvm_domain 时，这个变量的值一定是 null。171 行把这个变量实例化。172 行用到了 m_uvm_schedule，这是一 uvm_phase 类型的静态成员变量：

```
文件：src/base/uvm_domain.svh
类：uvm_domain

 70    static local uvm_phase m_uvm_schedule;
```

由于是 local 类型的，这个变量的值只能被此类的函数修改。我们暂且先不管 uvm_phase 的 new 函数传递进去的 UVM_PHASE_SCHEDULE 是什么意思，可以先把它其当成一个容器，173 行调用 add_uvm_phase 函数向这个变量中插入了动态运行时的 phase：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：add_uvm_phase

146    static function void add_uvm_phases(uvm_phase schedule);
147
148      schedule.add(uvm_pre_reset_phase::get());
149      schedule.add(uvm_reset_phase::get());
150      schedule.add(uvm_post_reset_phase::get());
151      schedule.add(uvm_pre_configure_phase::get());
152      schedule.add(uvm_configure_phase::get());
153      schedule.add(uvm_post_configure_phase::get());
154      schedule.add(uvm_pre_main_phase::get());
155      schedule.add(uvm_main_phase::get());
156      schedule.add(uvm_post_main_phase::get());
157      schedule.add(uvm_pre_shutdown_phase::get());
158      schedule.add(uvm_shutdown_phase::get());
159      schedule.add(uvm_post_shutdown_phase::get());
160
161    endfunction
```

这段代码的意思也是相当明白，虽然不知道 add 函数是怎么工作的，但是我们有理由相信这 12 个运行时的 phase 已经以某种形式存在于 m_uvm_schedule 中。

174 行把 m_uvm_schedule 加入到 m_uvm_domain 中。所以 m_uvm_domain 中存放的就是 12 个动态运行时的 phase。get_uvm_domain 的返回值就是存储了这 12 个动态运行 phase 的 m_uvm_domain。

13.2.3. uvm_domain 类小结

uvm_domain 类共有四个静态成员变量：

```
文件：src/base/uvm_domain.svh
类：uvm_domain

 65 class uvm_domain extends uvm_phase;
 66
 67    static local uvm_domain m_common_domain;
 68    static local uvm_domain m_uvm_domain; // run-time phases
 69    static local uvm_domain m_domains[string];
 70    static local uvm_phase m_uvm_schedule;
       …
190 endclass
```

经过前面两节的分析，m_common_domain 中存放的就是我们正常运行时所有的 phase，包括各种函数 phase，run_phase 及 12 个动态运行时的 phase。

m_uvm_domain 中存放的是一个 uvm_phase 类型的变量 m_uvm_schedule，而这个变量中存放着 12 个动态运行时的 phase。

m_uvm_schedule 存放的是 12 个动态运行的 phase。

m_domains 则用于记录所有的 uvm_domain 信息。uvm_domain 的 new 函数如下：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：new

183    function new(string name);
184       super.new(name,UVM_PHASE_DOMAIN);
185       if (m_domains.exists(name))
186          `uvm_error("UNIQDOMNAM", $psprintf("Domain created with non-unique name '%s'
", name))
187       m_domains[name] = this;
188    endfunction
```

从这段代码中可以看出，每当有一个 uvm_domain 实例化的时候，就会往 m_domains 中插入一条记录，而这条记录的内容就是这个实例的指针。

在 13.2.1 节中，有一个 m_common_domain 实例化，在 13.2.2 节中，有一个 m_uvm_domain 实例化，所以现在 m_domains 中有了这两条记录。事实上，当一个验证平台中如果自己不另外定义 uvm_domain 的话，那么 m_domains 中的记录只有这两条。

13.3. 浅探 uvm_phase

uvm_phase 是一个相对比较复杂的类，上面分析的 uvm_domain 其实就是派生自 uvm_phase。上面提到了 add 函数就是 uvm_phase 的一个成员函数，本节分析这个类。

13.3.1. uvm_phase 的类型

13.2.2 节中到 m_uvm_schedule 实例时，提到了一个 UVM_PHASE_SCHEDULE。m_uvm_schedule 是一个 uvm_phase 的变量。uvm_phase 的构造函数的原型如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

 155  class uvm_phase extends uvm_object;
         …
 170     extern function new(string name="uvm_phase",
 171                         uvm_phase_type phase_type=UVM_PHASE_SCHEDULE,
 172                         uvm_phase parent=null);
         …
 622  endclass
```

uvm_phase_type 是一个枚举变量，其定义为：

```
文件：src/base/uvm_object_globals.svh
类：无

472  typedef enum { UVM_PHASE_IMP,
473                 UVM_PHASE_NODE,
474                 UVM_PHASE_TERMINAL,
475                 UVM_PHASE_SCHEDULE,
476                 UVM_PHASE_DOMAIN,
477                 UVM_PHASE_GLOBAL
478  } uvm_phase_type;
```

UVM_PHASE_IMP 的意思相对难懂一些，这里暂且先跳过。UVM_PHASE_NODE 代表普通的 phase。UVM_PHASE_SCHEDULE 代表几个普通 phase 联合。UVM_PHASE_ TERMINAL 代表一个 schedule 中的终结 node。UVM_PHASE_DOMAIN 代表整个 phase 图，其中可以包含 schedule 和 phase。也就是说，UVM_PHASE_DOMAIN 是最大的单位，它里面包含了大量的 phase。而 UVM_PHASE_SCHEDULE 其实纯粹是伴随着 12 个动态运行的 phase 的引入而引入的一个概念。如果验证平台中不重新定义 uvm_domain，那么读者可以简单理解成 UVM_PHASE_SCHEDULE 其实就等于 12 个动态运行 phase 的联合体，如图 13-1

所示。

由于 UVM 的运行是按照图 13-1，从上往下按照 phase 推进的。13.2 节中，我们看到，通过 add 函数，可以把 phase 依次加入到整个 domain 中。假如上面的图中现在没有 12 个运行时的 phase，那么我们要现在要想办法把这 12 个 phase 给加进去，那应该怎么办呢？add 函数提供一个 with_phase 的参数，可以让两个 phase 并行的运行。但是问题是我们这里是有 12 个 phase，是没有办法使用这个参数的。所以为了使用这个参数，我们把这 12 个 phase 打包成一个 UVM_PHASE_SCHEDULE，然后把其通过 add 函数加入到 domain 中，这就达到了并行运行的目的。所以，可以看的出来，UVM_PHASE_SCHEDULE 纯粹就是为这 12 个动态运行的 phase 引入的概念。

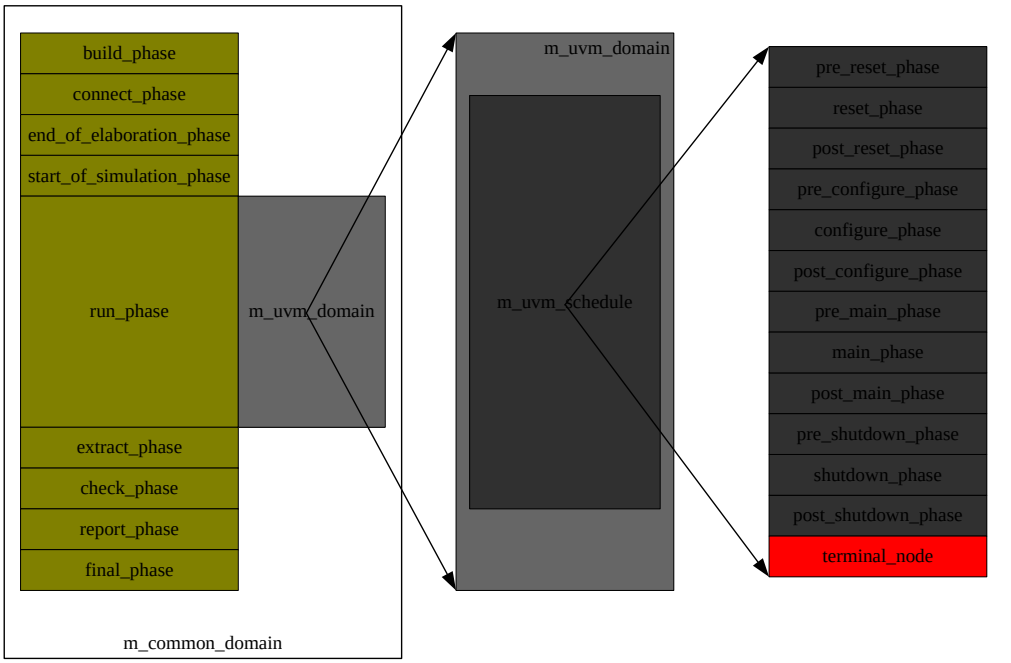


图 13-1 UVM 的 phase 运行图

13.3.2. 构造函数 new

uvm_phase 的 new 函数如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：new
```

```
642 function uvm_phase::new(string name="uvm_phase",
643                            uvm_phase_type phase_type=UVM_PHASE_SCHEDULE,
644                            uvm_phase parent=null);
645   super.new(name);
646   m_phase_type = phase_type;
647
648   if (name == "run")
649     phase_done = uvm_test_done_objection::get();
650   else begin
651     phase_done = new(name);
652   end
653
654   m_state = UVM_PHASE_DORMANT;
655   m_run_count = 0;
656   m_parent = parent;
657
```

646 行把传入的 phase 类型赋值给了 m_phase_type 成员变量。648 到 651 行提到了 phase_done，它的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

554   uvm_objection phase_done; // phase done objection
```

这是一个 uvm_objection 类型的成员变量。由于牵到了 objection 机制，我们暂且先跳过。后面章节会讲述。

654 行给 m_state 赋值，这个变量的定义为：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

520   local uvm_phase_state      m_state;
```

这是一个 uvm_phase_state 类型的私有成员变量，uvm_pahse_state 是一个枚举类型，它用于标识一个 phase 的执行情况，是已经执行完了还是根本没有执行还是正在执行？它的定义为：

```
文件：src/base/uvm_object_globals.svh
类：无

521     typedef enum { UVM_PHASE_DORMANT        = 1,
522                    UVM_PHASE_SCHEDULED      = 2,
523                    UVM_PHASE_SYNCING        = 4,
524                    UVM_PHASE_STARTED        = 8,
525                    UVM_PHASE_EXECUTING      = 16,
526                    UVM_PHASE_READY_TO_END   = 32,
527                    UVM_PHASE_ENDED          = 64,
528                    UVM_PHASE_CLEANUP        = 128,
```

```
529                     UVM_PHASE_DONE          = 256,
530                     UVM_PHASE_JUMPING       = 512
531                     } uvm_phase_state;
```

这个定义位于 uvm_object_globals.svh 文件中，关于这些类型的详细说明将会在介绍 execute_phase 时详细阐述。654 行给 m_state 赋值成 UVM_PHASE_DORMANT 表示目前这个 phase 什么都没有做过。

655 行把 m_run_count 初始化成为 0，这个变量主要用于记录这个 phase 被执行了多少次，其实际意义并不大，读者可以直接忽略。

656 行给 m_parent 赋值。这个变量其实是伴随着 UVM_PHASE_SCHEDULE 的引入而引入的。大家还记得 uvm_domain::m_uvm_schedule 中加入了 12 个动态运行的 phase，m_uvm_schedule 就是这 12 个 phase 的 parent。当其它 uvm_phase 在实例化的时候，并不需要指定一个 parent，如 m_uvm_schedule 自己就是一个没有 parent 的 phase，build_phase 也是一个没有 parent 的 phase。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：new

 658    begin
 659      uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();
 660      string val;
 661      if (clp.get_arg_value("+UVM_PHASE_TRACE", val))
 662        m_phase_trace = 1;
 663      else
 664        m_phase_trace = 0;
 665      if (clp.get_arg_value("+UVM_USE_OVM_RUN_SEMANTIC", val))
 666        m_use_ovm_run_semantic = 1;
 667      else
 668        m_use_ovm_run_semantic = 0;
 669    end
 670
 671
 672    if (parent == null && (phase_type == UVM_PHASE_SCHEDULE ||
 673                           phase_type == UVM_PHASE_DOMAIN )) begin
 674      //m_parent = this;
 675      m_end_node = new({name,"_end"}, UVM_PHASE_TERMINAL, this);
 676      this.m_successors[m_end_node] = 1;
 677      m_end_node.m_predecessors[this] = 1;
 678    end
 679
 680  endfunction
```

661 到 664 行则用于从命令行得到是否使用 phase 跟踪来了解运行过程中 phase 的切换过程。665 到 668 行则判断是否使用 OVM 的 phase 运行机制，这纯粹是为了与 OVM 兼容，读者可以直接跳过。

672 行到 678 行则是在创建一个 UVM_PHASE_SCHEDULE 或一个

UVM_PHASE_DOMAIN 时，需要在这个 phase 的后面添加一个 UVM_PHASE_TERMINAL 型的终端结点。为什么要有这么个东西？因为一个 DOMAIN 或者 SCHEDULE 是按照顺序执行 phase 的，但是 phase 总有要执行完的时候，那怎么判断这是最后一个 phase 了呢？有很多种方法可以做这件事情，UVM 选了想对来说比较简单的一点。如上节中的图所示，对于 m_uvm_schedule 来说，当运行到 post_shutdown_phase 时会自动向前到达一个终端结点，此时一系统一年结点类型是终端结点，直接就退出了。那么对于普通的 phase 来说，为什么不需要这么个终端结点？因为普通 phase 运行完了，接下来要运行其它要正常运行的 phase，不能是这么一个会终结的 phase。

这里的关键是两个联合数组：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

 544    protected bit   m_predecessors[uvm_phase];
 545    protected bit   m_successors[uvm_phase];
```

这两个联合数组的索引都是 uvm_phase 类型的，而存储的内容是 bit 类型的。这两个数组用于标记整个 UVM 的 phase 运行图中每个结点的前后关系。一般说来，一个 phase 只有一个先驱者（predecessor），一个后继者（successor），但是有两个例外，一是 start_of_simulation _phase，它有两个后继者，分别是 run_phase 和包裹了 12 个动态运行 phase 的 uvm_domain:: m_uvm_schedule，二是 extract_phase，它有两个先驱者，即 run_phase 和 uvm_domain:: m_uvm_schedule。

13.3.3. uvm_build_phase

13.3.1 节在介绍 get_common_domain 函数时，我们曾经用到了 add 函数，而作为 add 函数的参数是这样的：

```
domain.add(uvm_build_phase::get());
```

在后面讲解 add 函数前，我们先把 add 的参数搞明白。uvm_build_phase 的定义为：

```
文件：src/base/uvm_common_phases.svh
类：uvm_build_phase

 52 class uvm_build_phase extends uvm_topdown_phase;
 53     virtual function void exec_func(uvm_component comp, uvm_phase phase);
 54         uvm_component comp_;
 55         if ($cast(comp_,comp))
 56           comp_.build_phase(phase);
```

```
 57      endfunction : exec_func
 58      local static uvm_build_phase m_inst;
 59      static const string type_name = "uvm_build_phase";
 60      static function uvm_build_phase get();
 61          if(m_inst == null) begin
 62              m_inst = new();
 63          end
 64          return m_inst;
 65      endfunction : get
 66      `_protected function new(string name="build");
 67          super.new(name);
 68      endfunction : new
 69      virtual function string get_type_name();
 70          return type_name;
 71      endfunction : get_type_name
 72  endclass : uvm_build_phase
```

这是一个派生自 uvm_topdown_phase 的类，其本身并不复杂，声明了非常简单的几个函数和成员变量。在这里，我们又看到了熟悉的一幕：一个私有的静态成员变量 m_inst，一个 protected 类型的 new 函数，一个用于返回 m_inst 值的 get 函数。这三者的组合我们在 uvm_root 中见过，在 uvm_factory 中见过，它们用于实现一个单实例的类。所以 uvm_build_phase 类也是单实例的。

关于这个类的基类 uvm_topdown_phase，其定义如下：

```
文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase

 35  virtual class uvm_topdown_phase extends uvm_phase;
        …
 42     function new(string name);
 43        super.new(name,UVM_PHASE_IMP);
 44     endfunction
        …
110  endclass
```

这个类的实现并不复杂，但是基于目前我们的认识，理解起来还有点困难。它的本质是一个 uvm_phase，新派生出来的功能主要是规范它自身所具有的 top_down 的功能。后面章节会详细介绍这个类。这里只说明一下它的 new 函数。它调用 uvm_phase 的 new 函数时，传入的类型是 UVM_PHASE_IMP，我们在 13.3.1 节讲述类型时，特意跳过了这个类型，现在我们终于找到了它的一个代表。UVM 中的 UVM_PHASE_IMP 其实指的就是类似于 uvm_build_phase，uvm_connect_phase 等一些单实例的 phase。

与 uvm_build_phase 一起声明的，还有 uvm_connect_phase 等一些常规的 phase，这些被声明的 phase 都是单实例的，它们分别派生自不同的基类，uvm_connect_phase，uvm_end_of_elaboration_phase，uvm_start_of_simulation_phase，uvm_extract_phase，uvm_check_phase，uvm_report_phase 的基类是 uvm_bottomup_phase，而

uvm_final_phase 则与 uvm_build_phase 一样，是派生自 uvm_topdown_phase 的，uvm_run_phase 则是派生自 uvm_task_phase 中。

这些声明中并不包括 12 个动态运行的 phase。这 12 个动态运行的 phase 的声明位于 uvm_runtime_phases.svh 文件中。它们的声明与 uvm_build_phase 的声明极其的相似，也是单实例的。另外它跟 uvm_run_phase 一样派生自 uvm_task_phase，因此这里不重复说明。

13.3.4. add 函数

add 函数是到现在为止一直我们一直使用但是不知道其原理的一个函数。这个函数相对复杂，其定义为：

文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：add

```
687 function void uvm_phase::add(uvm_phase phase,
688                              uvm_phase with_phase=null,
689                              uvm_phase after_phase=null,
690                              uvm_phase before_phase=null);
691   uvm_phase new_node, begin_node, end_node;
692   if (phase == null)
693     `uvm_fatal("PH/NULL", "add: phase argument is null")
694
695   if (with_phase != null && with_phase.get_phase_type() == UVM_PHASE_IMP) begin
696     string nm = with_phase.get_name();
697     with_phase = find(with_phase);
698     if (with_phase == null)
699       `uvm_fatal("PH_BAD_ADD",
700         {"cannot find with_phase '",nm,"' within node '",get_name(),"'"})
701   end
702
703   if (before_phase != null && before_phase.get_phase_type() == UVM_PHASE_IMP) begin
704     string nm = before_phase.get_name();
705     before_phase = find(before_phase);
706     if (before_phase == null)
707       `uvm_fatal("PH_BAD_ADD",
708         {"cannot find before_phase '",nm,"' within node '",get_name(),"'"})
709   end
710
711   if (after_phase != null && after_phase.get_phase_type() == UVM_PHASE_IMP) begin
712     string nm = after_phase.get_name();
713     after_phase = find(after_phase);
714     if (after_phase == null)
715       `uvm_fatal("PH_BAD_ADD",
```

```
716             {"cannot find after_phase '",nm,"' within node '",get_name(),"'"})
717     end
718
719     if (with_phase != null && (after_phase != null || before_phase != null))
720       `uvm_fatal("PH_BAD_ADD",
721           "cannot specify both 'with' and 'before/after' phase relationships")
722
```

函数一共有四个参数，第一个参数是加入的 phase，第二个是 with_phase，用于表明这个新加入的 phase 与原来的 DOMAIN 或者 SCHEDULE 中的哪个 phase 是并行的，当在 m_uvm_domain 中加入 m_uvm_schedule 时，就用到了这个参数，第三个是 after_phase，表明这个新加入的 phase 位于原 DOMAIN 或者 SCHEDULE 中哪个 phase 的后面，第四个 before_phase，表明这个新加入的 phase 位于原 DOMAIN 或者 SCHEDULE 中哪个 phase 的前面。

692 行到 693 行检查 phase 的有效性。

在看接下来的代码前，一定要理清一个事实，那就是到底是什么情况下才能调用 add 函数。在我们前面的例子中，我们用到了四次 add 函数：

```
domain.add(uvm_build_phase::get());
schedule.add(uvm_pre_reset_phase::get());
m_uvm_domain.add(m_uvm_schedule);
domain = get_uvm_domain();
m_common_domain.add(domain,
                    .with_phase(m_common_domain.find(uvm_run_phase::get())));
```

在这四次不同的调用中，作为调用者要么是一个 UVM_PHASE_SCHEDULE，要么是一个 UVM_PHASE_DOMAIN。而作为 add 的参数的 phase，则有三种情况，uvm_build_phase 的属性是 UVM_PHASE_IMP，m_uvm_schedule 的属性是 UVM_PHASE_SCHEDULE，而 domain 的属性是 UVM_PHASE_DOMAIN。

接下来我们先看第一种情况的调用。在这种调用方式中，with_phase，after_phase 和 before_phase 均为 null，所以 695 到 721 行可以直接跳过。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：add

723     if (before_phase == this || after_phase == m_end_node || with_phase == m_end_node)
724       `uvm_fatal("PH_BAD_ADD",
725           "cannot add before begin node, after end node, or with end nodes")
726
727     // If we are inserting a new "leaf node"
728     if (phase.get_phase_type() == UVM_PHASE_IMP) begin
729       new_node = new(phase.get_name(),UVM_PHASE_NODE,this);
730       new_node.m_imp = phase;
731       begin_node = new_node;
732       end_node = new_node;
```

```
733     end
734     // We are inserting an existing schedule
735     else begin
736       begin_node = phase;
737       end_node   = phase.m_end_node;
738       phase.m_parent = this;
739     end
740
741     // If 'with_phase' is us, then insert node in parallel
742     /*
743     if (with_phase == this) begin
744       after_phase = this;
745       before_phase = m_end_node;
746     end
747     */
748
```

723 行提到了 m_end_node。13.3.2 在介绍 uvm_phase 的构造函数 new 的时候，曾经提到过，当一个 UVM_PHASE_SCHEDULE 或者一个 UVM_PHASE_DOMAIN 在初始化的时候，会实例化一个 UVM_PHASE_TERMINAL 类型的 uvm_phase，作为终端结点。这里的 m_end_node 就是指的这个终端结点。很明显，723 行的条件也是不成立的。

728 行判断 add 的参数是什么类型的，现在分析的是第一种调用方式，所以是 UVM_PHASE_IMP 类型的，于是执行 729 到 732 行的分支。729 行实例化一个 UVM_PHASE_NODE 的结点，并且把这个 domain 作为此结点的 parent。730 行把 m_imp 的值赋值为传入的参数，即 uvm_build_phase::get()，这个函数的返回值是 uvm_build_phase 的一个单实例。之后同时把 begin_node 和 end_node 初始化为新实例化的这个结点。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：add

749     // If no before/after/with specified, insert at end of this schedule
750     if (with_phase == null && after_phase == null && before_phase == null) begin
751       before_phase = m_end_node;
752     end
753
754
755     if (m_phase_trace) begin
756       uvm_phase_type typ = phase.get_phase_type();
757       `uvm_info("PH/TRC/ADD_PH",
758         {get_name()," (",m_phase_type.name(),") ADD_PHASE: phase=",phase.get_full_name
(),"  (",
759         typ.name(),", inst_id=",$sformatf("%0d",phase.get_inst_id()),")",
760         " with_phase=",     (with_phase == null)     ? "null" : with_phase.get_name(),
761         " after_phase=",   (after_phase == null)   ? "null" : after_phase.get_name(),
762         " before_phase=", (before_phase == null) ? "null" : before_phase.get_name(),
```

```
763          " new_node=",       (new_node == null)       ? "null" : {new_node.get_name(),
764                                                                    " inst_id=",
765                                                                     $sformatf("%0d",new_n
ode.get_ inst_id())},
766          " begin_node=",    (begin_node == null)    ? "null" : begin_node.get_name(),
767          " end_node=",       (end_node == null)       ? "null" : end_node.get_name()},UVM_
DEBUG)
768    end
769
770
```

750 行，在 with_phase，after_phase 和 before_phase 均为 null 的情况下，把 m_end_node 作为 before_phase 的值。这是很容易理解的，因为三个参数都不加，所以就是没有什么约束条件，那当然是加在最后的终端结点之前了。

755 到 768 行是为了调试时打印信息用的，直接跳过。

文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：add

```
771    // INSERT IN PARALLEL WITH 'WITH' PHASE
772    if (with_phase != null) begin
773       begin_node.m_predecessors = with_phase.m_predecessors;
774       end_node.m_successors = with_phase.m_successors;
775       foreach (with_phase.m_predecessors[pred])
776          pred.m_successors[begin_node] = 1;
777       foreach (with_phase.m_successors[succ])
778          succ.m_predecessors[end_node] = 1;
779    end
780
781
782    // INSERT BEFORE PHASE
783    else if (before_phase != null && after_phase == null) begin
784       begin_node.m_predecessors = before_phase.m_predecessors;
785       end_node.m_successors[before_phase] = 1;
786       foreach (before_phase.m_predecessors[pred]) begin
787          pred.m_successors.delete(before_phase);
788          pred.m_successors[begin_node] = 1;
789       end
790       before_phase.m_predecessors.delete();
791       before_phase.m_predecessors[end_node] = 1;
792    end
793
794
795    // INSERT AFTER PHASE
796    else if (before_phase == null && after_phase != null) begin
797       end_node.m_successors = after_phase.m_successors;
798       begin_node.m_predecessors[after_phase] = 1;
799       foreach (after_phase.m_successors[succ]) begin
800          succ.m_predecessors.delete(after_phase);
801          succ.m_predecessors[end_node] = 1;
```

```
802       end
803       after_phase.m_successors.delete();
804       after_phase.m_successors[begin_node] = 1;
805     end
806
807
808     // IN BETWEEN 'BEFORE' and 'AFTER' PHASES
809     else if (before_phase != null && after_phase != null) begin
810       if (!after_phase.is_before(before_phase)) begin
811         `uvm_fatal("PH_ADD_PHASE",{"Phase '",before_phase.get_name(),
812                    "' is not before phase '",after_phase.get_name(),"'"})
813       end
814       // before and after? add 1 pred and 1 succ
815       begin_node.m_predecessors[after_phase] = 1;
816       end_node.m_successors[before_phase] = 1;
817       after_phase.m_successors[begin_node] = 1;
818       before_phase.m_predecessors[end_node] = 1;
819       if (after_phase.m_successors.exists(before_phase)) begin
820         after_phase.m_successors.delete(before_phase);
821         before_phase.m_successors.delete(after_phase);
822       end
823     end
824
825 endfunction
```

由于刚才给 before_phase 赋值了，所以 772 到 823 的语句执行第二个分支。784 到 791 这几句话的意思就是把刚刚实例化的结点真正的插入到 m_end_node 之前，这主要涉及到三步：

第一，把新结点的 m_predecessors 更新为 m_end_node 的 m_predecessors。

第二，那些以 m_end_node 作为 m_successors 的结点，把它们的 m_successors 更新为新加入的结点。

第三，把 m_end_node 的 m_predecessors 更新为新插入的结点。

第一种调用方式是把一个 UVM_PHASE_IMP 类型的 uvm_phase 加入到一个 UVM_PHASE_DOMAIN 中，这里并不是真的把这个 phase 给加到这个 domain 中，而是新实例化了一个结点，把这个新实例化的结点加入到了 domain 中。这个新实例化的结点中，有一个指向输入的 phase 的指针。

第二种调用方式第一种完全一样，只是调用者变成了一个 UVM_PHASE_SCHEDULE 型的变量，不过这并没有改变函数的流程。

第三种与第一种类似，区别在于 728 与 739 行的语句时，执行的是第二个分支。在这个分支中并没有新实例化一个 uvm_phase，也就是说，当把一个 UVM_PHASE_SCHEDULE 类型的 uvm_phase 加入时，是真真切切的把这个变量给加进来了。

第四种调用方式与前面三种差异比较大。

695 行的条件符合，会直接进入这个 if 语句。这里的关键是 697 行调用 find 函数。find 函数的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：find

1066 function uvm_phase uvm_phase::find(uvm_phase phase, bit stay_in_scope=1);
1067   // TBD full search
1068   //$display({"\nFIND node "",phase.get_name(),"" within ",get_name()," (scope ",m_phase_ty
pe.name (),")", (stay_in_scope) ? " staying within scope" :       ""});
1069   if (phase == m_imp || phase == this)
1070     return phase;
1071   find = m_find_predecessor(phase,stay_in_scope,this);
1072   if (find == null)
1073     find = m_find_successor(phase,stay_in_scope,this);
1074 endfunction
```

1069 出现了 m_imp。m_imp 这个变量在前面新建一个 UVM_PHASE_NODE 时，出现过，而在第四种调用方式中，这是一个 UVM_PHASE_DOMAIN 在调用，其 m_imp 是等于 null 的，所以 phase==m_imp 是不成立的，另外 phase==this 这个条件也不成立。所以会直接跳到 1071 行。这里会调用 m_find_predecessor 函数，这个函数的定义为：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_find_predecessor

 972 function uvm_phase uvm_phase::m_find_predecessor(uvm_phase phase, bit stay_in_scope=1,
uvm_phase orig_phase=null);
 973   uvm_phase found;
 974   //$display("  FIND PRED node "",phase.get_name(),"" (id=",$sformatf("%0d",phase.get_inst
_id()),") - checking against ",get_name()," (",m_phase_type.      name()," id=",$sformatf("%0d",get
_inst_id()),(m_imp== null)?"":{"/",$sformatf("%0d",m_imp.get_inst_id())},")");
 975   if (phase == m_imp || phase == this)
 976     return this;
 977   foreach (m_predecessors[pred]) begin
 978     uvm_phase orig;
 979     orig = (orig_phase==null) ? this : orig_phase;
 980     if (!stay_in_scope ||
 981         (pred.get_schedule() == orig.get_schedule()) ||
 982         (pred.get_domain() == orig.get_domain())) begin
 983       found = pred.m_find_predecessor(phase,stay_in_scope,orig);
 984       if (found != null)
 985         return found;
 986     end
 987   end
 988   return null;
 989 endfunction
```

这个函数相对来说比较简单，它通过递归的方式，来查询输入的参数 phase 是不是这个 phase 的 predecessor 或者是其 predecessor 的 predecessor。

也就是说，如图 13-2 所示，假设在 E 中调用 m_find_predecessor 函数，传入的参数是 B，那么接下来会递归调用 D，C，B 的 m_find_predecessor 函数，当调用到 B 的 m_find_predecessor 函数时，795 行的条件 B==this 满足了，于是直接返回 B。

在我们的这个例子中，m_find_predecesssor 是会直接返回 null 的。因为这是一个 domain，它的 m_predecessors 数组是空的。

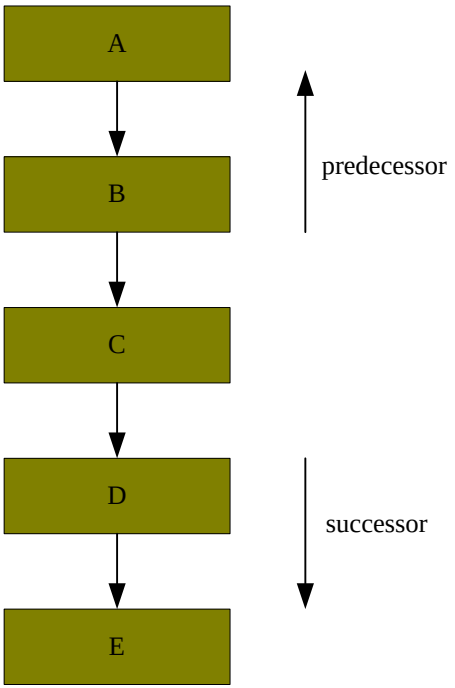


图 13-2 predecessor 与 successor

返回到 find 函数的 1072 行，这时条件满足了，于是进入 1073 行调用 m_find_successor 函数。这个函数的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_find_successor

1018 function uvm_phase uvm_phase::m_find_successor(uvm_phase phase, bit stay_in_scope=1, uvm_phase orig_phase=null);
1019     uvm_phase found;
1020     //$display("  FIND SUCC node '",phase.get_name(),"' (id=",$sformatf("%0d",phase.get_inst_id()),") - checking against ",get_name()," (",m_phase_type.        name()," id=",$sformatf("%0d",get_inst_id()),(m_imp== null)?"":{"/",$sformatf("%0d",m_imp.get_inst_id())},")");
1021     if (phase == m_imp || phase == this) begin
```

```
1022        return this;
1023        end
1024      foreach (m_successors[succ]) begin
1025        uvm_phase orig;
1026        orig = (orig_phase==null) ? this : orig_phase;
1027        if (!stay_in_scope ||
1028            (succ.get_schedule() == orig.get_schedule()) ||
1029            (succ.get_domain() == orig.get_domain())) begin
1030          found = succ.m_find_successor(phase,stay_in_scope,orig);
1031          if (found != null) begin
1032            return found;
1033            end
1034        end
1035      end
1036      return null;
1037  endfunction
```

这个函数跟 m_find_predecessor 完全相似，只是它往后找，而后者往前找而已。在我们的例子中，是要在 m_common_domain 中查找与 uvm_run_phase 对应的结点。在这之前，已经把这个结点加入进去了，所以这里会直接把这个结点的指针返回。

回到 add 函数的 698 行，这里 with_phase 不为 null，所以会跳出这个 if 分支。反思一下这里为什么要调用 find 函数？因为要确保这个 with_phase 是已经存在在这个 domain 里面的。同样的，703 到 717 行也是为了确保 before_phase 和 after_phase 是已经存在于这个 domain 里面的了。

719 行的语句用于避免一个待加入的结点一方面要和这个 domain 中的某一个结点同步运行，另外一方面却又要求在这个节点之前（或之后）运行。这是一种自相矛盾是行为，当然应该避免了。

728 到 739 与第三种调用方式相似，不重复阐述。750 行的语句现在是不满足条件了，所以不会给 before_phase 赋值了。772 到 823 的语句中会直接执行第一个分支。在这个分支中，把 start_of_simulation_phase 的 m_successsors 中加入了一条记录，extract_phase 的 m_predecessors 中加入了一条记录，同时把新加入的这个结点的 m_predecessors 设置为 start_of_simulation_phase，把 m_successors 设置为 extract_phase。

到此，add 函数讲述完毕。通过这种方式，形成了 13.3.1 节中的运行图。

13.3.5. 正常情况运行的 execute_phase：普通函数 phase

终于来到了 uvm_phase 中最重要，同时也几乎是控制着整个 UVM 运行的一个 task。从复杂度上来说，这个 task 也几乎是 UVM 所有函数和 task 中最难的一个。在介绍这个 task 之前，先回顾一下 m_run_phases 是如何调用这个 task 的：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_run_phases

1798 task uvm_phase::m_run_phases();
1799   uvm_root top = uvm_root::get();
1800
1801   // initiate by starting first phase in common domain
1802   begin
1803     uvm_phase ph = uvm_domain::get_common_domain();
1804     void'(m_phase_hopper.try_put(ph));
1805   end
1806
1807   forever begin
1808     uvm_phase phase;
1809     uvm_process proc;
1810     m_phase_hopper.get(phase);
1811     fork
1812       begin
1813         proc = new(process::self());
1814         phase.execute_phase();
1815       end
1816     join_none
1817     m_phase_top_procs[phase] = proc;
1818     #0;  // let the process start running
1819   end
1820 endtask
```

为了方便，把代码重新贴过来了。现在我们知道 get_common_domain 得到的是 uvm_domain::m_common_domain，所以 1814 行其实是 m_common_domain.execute_phase，也就是说，execute_phase 的调用者是一个 domain。在此前提下，我们分析这个 task。

函数的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase
```

```
1122 task uvm_phase::execute_phase();
1123
1124   uvm_task_phase task_phase;
1125   uvm_root top;
1126   top = uvm_root::get();
1127
1128   // If we got here by jumping forward, we must wait for
1129   // all its predecessor nodes to be marked DONE.
1130   // (the next conditional speeds this up)
1131   // Also, this helps us fast-forward through terminal (end) nodes
1132   foreach (m_predecessors[pred])
1133     wait (pred.m_state == UVM_PHASE_DONE);
1134
1135
1136   // If DONE (by, say, a forward jump), return immed
1137   if (m_state == UVM_PHASE_DONE)
1138     return;
1139
1140
1141   //---------
1142   // SYNCING:
1143   //---------
1144   // Wait for phases with which we have a sync()
1145   // relationship to be ready. Sync can be 2-way -
1146   // this additional state avoids deadlock.
1147   if (m_sync.size()) begin
1148     m_state = UVM_PHASE_SYNCING;
1149     foreach (m_sync[i]) begin
1150       wait (m_sync[i].m_state >= UVM_PHASE_SYNCING);
1151     end
1152   end
1153
1154   m_run_count++;
1155
1156
1157   if (m_phase_trace) begin
1158     `UVM_PH_TRACE("PH/TRC/STRT","Starting phase",this,UVM_LOW)
1159   end
1160
1161
```

1132 与 1133 行是在等待所有这个 phase 的先驱 phase 全部执行完毕。这里用到了一个 phase 的执行状态的概念，前面已经提及过了。UVM_PHASE_DOMANT 表示这个 domain 中的 phase 什么都没有做过。UVM_PHASE_SCHEDULED 表示至少有一个先驱 phase 已经完成，正在等待所有其它的先驱 phase 完成。UVM_PHASE_SYNCING 表示所有的先驱 phase 已经完成，正跨 domian 检查各个需要同步的 phase。UVM_PHASE_STARTED 表示 phase 已经开始，调用 phase_started callback 函数。UVM_PHASE_EXECUTING 表示 phase 正在执行。UVM_PHASE_READY_TO_END 表示 phase 已经结束，正在等待完成，如 run 结束后等待 post_shutdown 完成。UVM_PHASE_ENDED 表示 phase 已经执行完，调用

phase_ended callback 函数。UVM_PHASE_CLEANUP 表示 phase 相关的各进程被 kill 掉。UVM_PHASE_DONE 表示本 phase 已经执行完毕，接下来的 phase 可以执行了。

1137 行则判断当前 phase 是不是已经执行完毕，如果完毕了，直接返回。正常情况下不会出现这种情况，只有当 phase jump 时才会出现。

1147 到 1152 行主要用于有 phase 需要同步时才会出现。一般的，这种同步只有在跨 domain 时才会出现，因此这里暂且先跳过。前面说过，1154 行的计数器没有太多意义，直接跳过。1157 到 1159 行用于调试。接下来在此 task 中还有许多用于调试的，直接跳过，不再单独说明。

文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

```
1162    // If we're a schedule or domain, then "fake" execution
1163    if (m_phase_type != UVM_PHASE_NODE) begin
1164      m_state = UVM_PHASE_STARTED;
1165      #0;
1166      m_state = UVM_PHASE_EXECUTING;
1167      #0;
1168    end
1169
1170
1171    else begin // PHASE NODE
          …
1297    end
1298
1299
1300    //---------
1301    // JUMPING:
1302    //---------
1303
1304    // If jump_to() was called then we need to kill all the successor
1305    // phases which may still be running and then initiate the new
1306    // phase.  The return is necessary so we don't start new successor
1307    // phases.  If we are doing a forward jump then we want to set the
1308    // state of this phase's successors to UVM_PHASE_DONE.  This
1309    // will let us pretend that all the phases between here and there
1310    // were executed and completed.  Thus any dependencies will be
1311    // satisfied preventing deadlocks.
1312    // GSA TBD insert new jump support
1313
1314    if (m_phase_type == UVM_PHASE_NODE) begin
          …
1350    end
1351
1352    // WAIT FOR PREDECESSORS:  // WAIT FOR PREDECESSORS:
1353    // function phases only
1354    if (task_phase == null)
1355      m_wait_for_pred();
```

```
1356
1357
```

1163 行用于判断是不是调用 execute_phase 的是不是一个 UVM_PHASE_NODE，由于这里是一个 domain，所以进入 1164 行的分支。这个分支其实就是假装执行一下，之后直接跳出到达 1314 行，而 1314 行条件不满足，执行到 1354 行。task_phase 是一个在 execute_phase 中定义的变量，初始值为 null，到目前为止没有对其赋值，所以这 1354 行的条件满足，进入 1355 行，调用 m_wait_for_pred。m_wait_for_pred 的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_wait_for_pred

1421  task uvm_phase::m_wait_for_pred();
1422
1423    if(!(m_jump_fwd || m_jump_bkwd)) begin
1424
1425      bit done;
1426      bit successors[uvm_phase];
1427      bit pred_of_succ[uvm_phase];
1428
1429      // get all successors
1430      foreach (m_successors[succ])
1431        successors[succ] = 1;
1432
1433      // replace TERMINAL or SCHEDULE nodes with their successors
1434      do begin
1435        done=1;
1436        foreach (successors[succ]) begin
1437          if (succ.get_phase_type() != UVM_PHASE_NODE) begin
1438            successors.delete(succ);
1439            foreach (succ.m_successors[next_succ])
1440              successors[next_succ] = 1;
1441            done = 0;
1442          end
1443        end
1444      end while(!done);
1445
```

1423 行检查有没有 phase 的跳转，只有在不跳转的情况下才执行。1430，1431 行把 m_successors 数组的内容复制到 successors 中来。1433 到 1444 行用于把 successors 中的 UVM_PHASE_TERMINAL，UVM_PHASE_SCHEDULE 和 UVM_PHASE_DOMAIN 剔除。如果这是一个 start_of_simulation_phase 在调用这个函数的话，那么它有两个 successors，一个是 run_phase，另外一个是包含 12 个运行时 phase 的 m_uvm_domain，会把后者剔除。但是此处是 domain，其 successors 中不存在这样的，所以这几行不会执行。

```
文件：src/base/uvm_phase.svh
```

```
类：uvm_phase
函数/任务：m_wait_for_pred

1446        // get all predecessors to these successors
1447        foreach (successors[succ])
1448          foreach (succ.m_predecessors[pred])
1449            pred_of_succ[pred] = 1;
1450
1451        // replace any terminal nodes with their predecessors, recursively.
1452        // we are only interested in "real" phase nodes
1453        do begin
1454          done=1;
1455          foreach (pred_of_succ[pred]) begin
1456            if (pred.get_phase_type() != UVM_PHASE_NODE) begin
1457              pred_of_succ.delete(pred);
1458              foreach (pred.m_predecessors[next_pred])
1459                pred_of_succ[next_pred] = 1;
1460              done =0;
1461            end
1462          end
1463        end while (!done);
1464
1465
```

1446 到 1463 行做的事情类似于 1430 到 1444，不过这里是把 precedessors 中的非 UVM_PHASE_NODE 结点剔除。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_wait_for_pred

1466        // remove ourselves from the list
1467        pred_of_succ.delete(this);
1468
1469        // wait for predecessors to successors (real phase nodes, not terminals)
1470        // mostly debug msgs
1471        foreach (pred_of_succ[sibling]) begin
1472
1473          if (m_phase_trace) begin
1474            string s;
1475            s = $sformatf("Waiting for phase '%s' (%0d) to be READY_TO_END. Current sta
te is %s",
1476                  sibling.get_name(),sibling.get_inst_id(),sibling.m_state.name());
1477            `UVM_PH_TRACE("PH/TRC/WAIT_PRED_OF_SUCC",s,this,UVM_HIGH)
1478          end
1479
1480          sibling.wait_for_state(UVM_PHASE_READY_TO_END, UVM_GTE);
1481
1482          if (m_phase_trace) begin
1483            string s;
1484            s = $sformatf("Phase '%s' (%0d) is now READY_TO_END. Releasing phase",
1485                  sibling.get_name(),sibling.get_inst_id());
```

```
1486            `UVM_PH_TRACE("PH/TRC/WAIT_PRED_OF_SUCC",s,this,UVM_HIGH)
1487          end
1488
1489        end
1490
1491        if (m_phase_trace) begin
1492          if (pred_of_succ.num()) begin
1493            string s = "( ";
1494            foreach (pred_of_succ[pred])
1495              s = {s, pred.get_full_name()," "};
1496            s = {s, ")"};
1497            `UVM_PH_TRACE("PH/TRC/WAIT_PRED_OF_SUCC",
1498                  {"*** All pred to succ ",s," in READY_TO_END state, so ending phase ***"},this,UVM_HIGH)
1499          end
1500          else begin
1501            `UVM_PH_TRACE("PH/TRC/WAIT_PRED_OF_SUCC",
1502                  "*** No pred to succ other than myself, so ending phase ***",this,UVM_HIGH)
1503          end
1504        end
1505
1506      end
1507      #0; // LET ANY WAITERS WAKE UP
1508
1509  endtask
```

1467 行把自己从 pred_of_succ 中去除，这样整个 pred_of_succ 中全部都是当前 phase 的兄弟姐妹，于是 1480 行等待这些兄弟姐妹到达 UVM_PHASE_REDAY_TO_END。事实上，当前的是一个 domain，根本就没有兄弟姐妹，所以其实 1480 行也不会执行。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase


1358      //-------
1359      // ENDED:
1360      //-------
1361      // execute 'phase_ended' callbacks
1362      if (m_phase_trace)
1363        `UVM_PH_TRACE("PH_END","ENDING PHASE",this,UVM_HIGH)
1364      m_state = UVM_PHASE_ENDED;
1365      if (m_imp != null)
1366        m_imp.traverse(top,this,UVM_PHASE_ENDED);
1367      #0; // LET ANY WAITERS WAKE UP
1368
1369      //---------
1370      // CLEANUP:
1371      //---------
```

```
1372    // kill this phase's threads
1373    m_state = UVM_PHASE_CLEANUP;
1374    if (m_phase_proc != null) begin
1375      m_phase_proc.kill();
1376      m_phase_proc = null;
1377    end
1378    #0; // LET ANY WAITERS WAKE UP
1379
1380    end
1381
1382
1383    //------
1384    // DONE:
1385    //------
1386    if (m_phase_trace)
1387      `UVM_PH_TRACE("PH/TRC/DONE","Completed phase",this,UVM_LOW)
1388    m_state = UVM_PHASE_DONE;
1389    m_phase_proc = null;
1390    #0; // LET ANY WAITERS WAKE UP
1391
1392
1393
1394    //-----------
1395    // SCHEDULED:
1396    //-----------
1397    // If more successors, schedule them to run now
1398    m_phase_top_procs.delete(this);
1399    if (m_successors.size() == 0) begin
1400      top.m_phase_all_done=1;
1401    end
1402    else begin
1403      // execute all the successors
1404      foreach (m_successors[succ]) begin
1405        if(succ.m_state < UVM_PHASE_SCHEDULED) begin
1406          succ.m_state = UVM_PHASE_SCHEDULED;
1407            #0; // LET ANY WAITERS WAKE UP
1408          void'(m_phase_hopper.try_put(succ));
1409          if (m_phase_trace)
1410            `UVM_PH_TRACE("PH/TRC/SCHEDULED",{"Scheduled from phase ",get_full_
name()},succ,UVM_LOW)
1411        end
1412      end
1413    end
1414
1415  endtask
```

回到 execute_phase 中来，程序运行到 1365 行，由于这是一个 domain，其 m_imp=null，所以也直接跳过。1373 到 1380 行做一些清理工作，1386 到 1390 行是简单的赋值语句，直接跳过。接下来真正重要的其实是发生在 1399 行，判断一下是不是有后继者。做为一个 domain 来说，其主要的成员都体现在其后继者中，所以进入 else 分支。在这个 else 分支中，判断后继者的状态。我们现在分析的是

m_common_domain 的 execute_phase，而 m_common_domain 只有一个后继者，那就是插入 uvm_build_phase 时新实例化的那个结点。这个结点的状态被初始化成 UVM_PHASE_DOMANT，满足 1405 行的条件，于是把这个代表 uvm_build_phase 的 phase 放入 m_phase_hooper 中。之后整个 execute_phase 就结束了。什么？这就结束了？确实是不错。现在就结束了，是因为我们分析的是一个 UVM_PHASE_DOMAIN 调用的 execute_phase。接下来将会分析一个 UVM_PHASE_NODE 是如何调用这个 task 的。

往 m_phase_hooper 放入一个 phase 后，注意到 m_run_phases 函数中 1810 行是处于一个无限循环中，所以一旦放入后，1810 行马上会得到这个 phase，之后启动一个进程，执行 execute_phase。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1162    // If we're a schedule or domain, then "fake" execution
1163    if (m_phase_type != UVM_PHASE_NODE) begin
           …
1168    end
1169
1170
1171    else begin // PHASE NODE
1172
1173       //---------
1174       // STARTED:
1175       //---------
1176       m_state = UVM_PHASE_STARTED;
1177       m_imp.traverse(top,this,UVM_PHASE_STARTED);
1178       #0; // LET ANY WAITERS WAKE UP
1179
1180
```

现在回到 execute_phase，此时 1163 行的条件不满足，进入 else 分支。1177 行调用 m_imp 的 traverse 函数。读者还记得 m_imp 是什么吗？当我们把 uvm_build_phase 加入到 domain 中时，并没有真正的加入，而是新实例化了一个结点，把这个结点加入了 domain 中，并且把这个结点的 m_imp 设置为了 uvm_build_phase 类唯一的静态实例。所以这里的 m_imp 其实就是 uvm_build_phase::get()。这里用到了 traverse task。这个 task 的定义位于 uvm_topdown_phase 类中（uvm_build_phase 派生自这个类）：

```
文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase
函数/任务：traverse

 52    virtual function void traverse(uvm_component comp,
 53                                   uvm_phase phase,
```

```
 54                                          uvm_phase_state state);
 55       string name;
 56       uvm_domain phase_domain = phase.get_domain();
 57       uvm_domain comp_domain = comp.get_domain();
 58
 59       if (m_phase_trace)
 60       `uvm_info("PH_TRACE",$sformatf("topdown-phase phase=%s state=%s comp=%s comp.d
omain=%s phase.domain=%s",
 61                 phase.get_name(), state.name(), comp.get_full_name(),comp_domain.get_name(),phas
e_domain. get_name()),
 62                 UVM_DEBUG)
 63
 64       if (phase_domain == uvm_domain::get_common_domain() ||
 65           phase_domain == comp_domain) begin
 66           case (state)
 67             UVM_PHASE_STARTED: begin
 68               comp.m_current_phase = phase;
 69               comp.m_apply_verbosity_settings(phase);
 70               comp.phase_started(phase);
 71               end
                …
 91           endcase
 92       end
 93       if(comp.get_first_child(name))
 94         do
 95           traverse(comp.get_child(name), phase, state);
 96         while(comp.get_next_child(name));
 97     endfunction
```

64 行的条件直接满足，因为 uvm_build_phase 所在的 domain 恰好是 m_common_domain。65 行的判断主要是为了跨 domain 时才会用到，这里暂时先忽略。

进入 66 行的 case 语句，由于这里传入的是 UVM_PHASE_STARTED，所以进入 68 行的分支。在调用 traverse 时传入的 comp 是 uvm_root，所以这里先把 uvm_root 的 m_current_phase 赋值，调用 m_apply_verbosity_settings 函数。从函数的名字可以看出这个是为了设置信息报告冗余度级别，所以直接跳过。我们比较感兴趣的是 70 行的 phase_started。这个函数是在 uvm_component 中定义的：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：phase_started

2359 function void uvm_component::phase_started(uvm_phase phase);
2360 endfunction
```

这是一个空函数，是 UVM 提供给用户的一个接口。如果用户想在 phase 启动之前作一些动作的话，可以重载这个函数。

回到 traverse 上来，93 行到 96 行则开始从上而下递归的调用 traverse。当整棵

UVM 树遍历完毕后，返回到 execute_phase 中来。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1181        //if (m_imp.get_phase_type() != UVM_PHASE_TASK) begin
1182        if (!$cast(task_phase,m_imp)) begin
1183
1184          //-----------
1185          // EXECUTING: (function phases)
1186          //-----------
1187          m_state = UVM_PHASE_EXECUTING;
1188          #0; // LET ANY WAITERS WAKE UP
1189          m_imp.traverse(top,this,UVM_PHASE_EXECUTING);
1190
1191        end
```

1182 行判断这个 phase 是不是一个 task phase。这是 uvm_build_phase，所以不是一个 task phase，满足 if 的条件。于是给 m_state 置位，同时再次调用 traverse，这一次传递进去的则是 UVM_PHASE_EXECUTING。

```
文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase
函数/任务：traverse

 66             case (state)
                  …
 72               UVM_PHASE_EXECUTING: begin
 73                 if (!(phase.get_name() == "build" && comp.m_build_done)) begin
 74                   uvm_phase ph = this;
 75                   comp.m_phasing_active++;
 76                   if (comp.m_phase_imps.exists(this))
 77                     ph = comp.m_phase_imps[this];
 78                   ph.execute(comp, phase);
 79                   comp.m_phasing_active--;
 80                 end
 81                 end
```

在 tranverse 的 73 行，用于判断当前的 phase 是不是 build_phase，以及 build_phase 是不是执行完了。如果已经执行完了，那么将不会调用接下来的代码而是会直接跳转到 93 行。什么情况下 build_phase 会已经执行完呢？当我们手动调用 build_phase 会出现这种情况。在 UVM1.1 之前，这种行为是允许的，但是从 UVM1.1 开始，这种用法已经被丢弃，所以读者尽量避免使用。

75 行出现了一个 m_phasing_active 变量，这是一个 uvm_component 的成员变量，它的用意主要就是监测用户是否手动调用了 build_phase。我们可以看 uvm_component 的 build_phase：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：build_phase

2271 function void uvm_component::build_phase(uvm_phase phase);
2272     m_build_done = 1;
2273     build();
2274 endfunction
```

build_phase 会直接调用 build：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：build

2278 function void uvm_component::build();
2279     m_build_done = 1;
2280     apply_config_settings(print_config_matches);
2281     if(m_phasing_active == 0) begin
2282         uvm_report_warning("UVM_DEPRECATED", "build()/build_phase() has been called explicitly, outside of the phasing system. This usage of build is deprecated and may lead to unexpected behavior.");
2283     end
2284 endfunction
```

在这段代码中，给出警告。如果 m_phasing_active 为 0，则说明是用户手动调用了 build_phase。因为如果是系统调用，如 75 行那样，会让这个变量的值不为 0。

76 行与 77 行判断用户是否需要执行另外的 phase 而不是执行 build_phase。一般的这种情况不会出现。

78 行调用 execute。其定义位于 uvm_topdown_phase 中：

```
文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase
函数/任务：execute

104     protected virtual function void execute(uvm_component comp,
105                                             uvm_phase phase);
106       comp.m_current_phase = phase;
107       exec_func(comp,phase);
108     endfunction
```

直接调用 exec_func，而这个函数的定义则位于 uvm_build_phase 中：

```
文件：src/base/uvm_common_phases.svh
类：uvm_build_phase
函数/任务：exec_func

 53     virtual function void exec_func(uvm_component comp, uvm_phase phase);
 54         uvm_component comp_;
```

```
55          if ($cast(comp_,comp))
56             comp_.build_phase(phase);
57      endfunction : exec_func
```

这个函数会直接又转而直接调用相应 component 的 build_phase。现在我们传入的参数是 uvm_root，所以最开始调用的是 uvm_root 的 build_phase。当执行完成后返回到 traverse 的 93 行，之后开始调用各个子 component 的 build_phase。这是一种自上而下的执行，所以说 build_phase 是一种自上而下执行的 phase。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1163    if (m_phase_type != UVM_PHASE_NODE) begin
           …
1168    end
1169
1170
1171    else begin // PHASE NODE
           …
1182       if (!$cast(task_phase,m_imp)) begin
              …
1191       end
1192       else begin
              …
1295       end
1296
1297    end
```

整棵 UVM 树的 build_phase 执行完毕后，回到 execute_phase 的 1191 行，此时跳出 if 分支，来到 1314 行（本节前面部分已经贴过此部分代码，这里不重复粘贴）。1314 行的条件满足，不过 1316 行的条件不满足，只有当出现 phase jump 的时候，1316 行的条件会满足。后面的章节会讲述 phase jump 的情况。1354 行判断此 phase 是不是一个函数 phase，条件满足，于是调用 m_wait_for_pred。前面说过，这个 task 的主要作用就是等待兄弟姐妹 phase 的状态达到 UVM_PHASE_READY_TO_END。由于 build_phase 根本没有兄弟姐妹，所以其实这个 task 可以直接忽略。

1364 行把 phase 的状态设置为 UVM_PHASE_ENDED，并接着调用 traverse，这次传递进去的是 UVM_PHASE_ENDED。在 traverse 中，86 行会直接调用 phase_ended 函数，这个函数与 70 行的 phase_started 一样，都是没有实质内容的函数。所以直接返回。

回到 executing_phase 的 1373 行，这里将进行一些清理工作，1388 行把 phase 的状态设置为 UVM_PHASE_DONE，接下来在 1408 行把代表 connect_phase 的结点放入 m_phase_hooper 中，之后可以预见的是执行此结点的 execute_phase。

connect_phase，end_of_elaboration_phase，start_of_simulation_phase 的执行过程

与 build_phase 类似，不过由于它们都派生自 uvm_bottomup_phase，所以其 traverse 有其独特之处：

```
文件：src/base/uvm_bottomup_phase.svh
类：uvm_bottomup_phase
函数/任务：traverse
   //
 52    virtual function void traverse(uvm_component comp,
 53                                   uvm_phase phase,
 54                                   uvm_phase_state state);
 55      string name;
 56      uvm_domain phase_domain =phase.get_domain();
 57      uvm_domain comp_domain = comp.get_domain();
 58
 59      if (comp.get_first_child(name))
 60        do
 61          traverse(comp.get_child(name), phase, state);
 62        while(comp.get_next_child(name));
 63
 64      if (m_phase_trace)
 65      `uvm_info("PH_TRACE",$sformatf("bottomup-phase phase=%s state=%s comp=%s comp.
domain=%s phase.domain=%s",
 66              phase.get_name(), state.name(), comp.get_full_name(),comp_domain.get_name(),phas
e_domain. get_name()),
 67              UVM_DEBUG)
 68
 69      if (phase_domain == uvm_domain::get_common_domain() ||
 70          phase_domain == comp_domain) begin
 71        case (state)
 72          UVM_PHASE_STARTED: begin
 73            comp.m_current_phase = phase;
 74            comp.m_apply_verbosity_settings(phase);
 75            comp.phase_started(phase);
 76            end
 77          UVM_PHASE_EXECUTING: begin
 78            uvm_phase ph = this;
 79            if (comp.m_phase_imps.exists(this))
 80              ph = comp.m_phase_imps[this];
 81            ph.execute(comp, phase);
 82            end
 83          UVM_PHASE_READY_TO_END: begin
 84            comp.phase_ready_to_end(phase);
 85            end
 86          UVM_PHASE_ENDED: begin
 87            comp.phase_ended(phase);
 88            comp.m_current_phase = null;
 89            end
 90          default:
 91            `uvm_fatal("PH_BADEXEC","bottomup phase traverse internal error")
 92        endcase
 93      end
```

```
94    endfunction
```

把这段代码与 uvm_topdown_phase 的 traverse 对比一下就能看出差异。对于 topdown 来说，是自上而下执行的，由于传给 traverse 的参数是最顶层的 uvm_root，所以先执行 uvm_root 的 build_phase，之后再一个一个的执行各个子 component 的 build_phase。但是对于 bottomup 来说，则是先去执行所有子 component 的相应 phase(如 connect_phase)，等到全部执行完毕后，再返回来执行自己的 connect_phase，也就说 uvm_root 的 connect_phase 将会在最后时刻执行。这刚好就是 bottomup 的意思。

13.3.6. 正常情况运行的 execute_phase：task phase

本节分析当 task phase 调用 execute_phase 时，其执行流程。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1394    //------------
1395    // SCHEDULED:
1396    //------------
1397    // If more successors, schedule them to run now
1398    m_phase_top_procs.delete(this);
1399    if (m_successors.size() == 0) begin
1400      top.m_phase_all_done=1;
1401    end
1402    else begin
1403      // execute all the successors
1404      foreach (m_successors[succ]) begin
1405        if(succ.m_state < UVM_PHASE_SCHEDULED) begin
1406          succ.m_state = UVM_PHASE_SCHEDULED;
1407            #0; // LET ANY WAITERS WAKE UP
1408          void'(m_phase_hopper.try_put(succ));
1409          if (m_phase_trace)
1410            `UVM_PH_TRACE("PH/TRC/SCHEDULED",{"Scheduled from phase ",get_full_
name()},succ,UVM_LOW)
1411        end
1412      end
1413    end
```

当 start_of_simulation_phase 的 execute_phase 执行到 1404 行时，由于它有两个 successor，第一个是 run_phase，第二个则是 uvm_domain::m_uvm_domain，所以对于 m_run_phases 来说，在 1807 行到 1819 行会同时 fork 起两个进程来，其中一个调用 run_phase 的 execute_phase，另外一个则执行 m_uvm_domain 的 execute_phase。

由于 m_uvm_domain 是一个 domain，所以它会假装执行，很快的运行到 1404 行（具体的代码见上节），这个 domain 中只有一个 successor，那就是 uvm_domain::m_uvm_schedule。于是接下来会经过 m_run_phases，开启 m_uvm_schedule 的 execute_phase 进程。由于 m_uvm_schedule 也不是一个真正的结点，所以如同 m_uvm_domain 一样，会假装执行，运行到 1404 行，把其唯一的一个 successor pre_reset_phase 放入 m_phase_hooper 中。之后由 m_run_phases 开启 pre_reset_phase 的 execute_phase 进程。由于 m_uvm_domain 和 m_uvm_schedule 的执行没有耗费任何时间，所以 pre_reset_phase 和 run_phase 是同一时刻被 fork 起来的两个进程，它们是同时运行的。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1163    if (m_phase_type != UVM_PHASE_NODE) begin
          …
1168    end
1169
1170
1171    else begin // PHASE NODE
          …
1182      if (!$cast(task_phase,m_imp)) begin
            …
1191      end
1192      else begin
1193
1194          fork : master_phase_process
1195            begin
1196
1197              m_phase_proc = process::self();
1198
1199              //-----------
1200              // EXECUTING: (task phases)
1201              //-----------
1202              m_state = UVM_PHASE_EXECUTING;
1203              task_phase.traverse(top,this,UVM_PHASE_EXECUTING);
1204
1205              wait(0); // stay alive for later kill
1206
1207            end
1208          join_none
1209
1210          uvm_wait_for_nba_region(); //Give sequences, etc. a chance to object
1211
          …
1295      end
1296
1297    end
```

回到 execute_phase，1194 到 1208 行通过 fork 开启了一个进程。在此进程之内调用 traverse，遍历整棵 UVM 树，执行相应的 phase。这里用到的 traverse 与上面一节介绍的不同，上一节的是属于函数 phase 的 traverse，而这个则是属于 task phase 的 traverse，其定义如下：

```
文件：src/base/uvm_task_phase.svh
类：uvm_task_phase
函数/任务：traverse
  //
 76    virtual function void traverse(uvm_component comp,
 77                                   uvm_phase phase,
 78                                   uvm_phase_state state);
 79      phase.m_num_procs_not_yet_returned = 0;
 80      m_traverse(comp, phase, state);
 81    endfunction
```

79 行把 m_num_procs_not_yet_returned 初始化为 0，之后调用 m_traverse:

```
文件：src/base/uvm_task_phase.svh
类：uvm_task_phase
函数/任务：m_traverse

 83    function void m_traverse(uvm_component comp,
 84                             uvm_phase phase,
 85                             uvm_phase_state state);
 86      string name;
 87      uvm_domain phase_domain =phase.get_domain();
 88      uvm_domain comp_domain = comp.get_domain();
 89
 90      if (comp.get_first_child(name))
 91        do
 92          m_traverse(comp.get_child(name), phase, state);
 93        while(comp.get_next_child(name));
 94
 95      if (m_phase_trace)
 96        `uvm_info("PH_TRACE",$sformatf("topdown-phase phase=%s state=%s comp=%s comp.d
omain=%s phase.domain=%s",
 97                phase.get_name(), state.name(), comp.get_full_name(),comp_domain.get_name(),phas
e_domain. get_name()),
 98                UVM_DEBUG)
 99
100      if (phase_domain == uvm_domain::get_common_domain() ||
101          phase_domain == comp_domain) begin
102        case (state)
103          UVM_PHASE_STARTED: begin
104            comp.m_current_phase = phase;
105            comp.m_apply_verbosity_settings(phase);
106            comp.phase_started(phase);
107            end
108          UVM_PHASE_EXECUTING: begin
109            uvm_phase ph = this;
```

```
110                 if (comp.m_phase_imps.exists(this))
111                   ph = comp.m_phase_imps[this];
112                 ph.execute(comp, phase);
113                 end
114               UVM_PHASE_READY_TO_END: begin
115                 comp.phase_ready_to_end(phase);
116                 end
117               UVM_PHASE_ENDED: begin
118                 comp.phase_ended(phase);
119                 comp.m_current_phase = null;
120                 end
121               default:
122                 `uvm_fatal("PH_BADEXEC","task phase traverse internal error")
123             endcase
124           end
125
126       endfunction
```

这个 m_traverse 与我们前面见过的 uvm_bottomup_phase::traverse 相似，都是先遍历 UVM 树中最底层的结点，最后才访问到 uvm_root。这似乎是给我们一种错觉，那就是对于 task phase（如 run_phase，main_phase 等）也是采用的自底向上的执行策略。这种说法是没有问题的，不过与 uvm_bottomup_phase 相比，这里的自底向上的区别在于并不等待相应的 phase 执行完毕后才返回，而是把进程 fork 起来后就直接返回。为什么这么说呢？我们看接下来的分析。

由于 112 行会调用 execute，所以接下来分析 execute：

```
文件：src/base/uvm_task_phase.svh
类：uvm_task_phase
函数/任务：execute

133     protected virtual function void execute(uvm_component comp,
134                                             uvm_phase phase);
135
136       fork
137         begin
138           uvm_sequencer_base seqr;
139
140           phase.m_num_procs_not_yet_returned++;
141
142           if ($cast(seqr,comp))
143             seqr.start_phase_sequence(phase);
144
145           exec_task(comp,phase);
146
147           phase.m_num_procs_not_yet_returned--;
148
149         end
150       join_none
151
```

```
152     endfunction
```

140 行与 147 行的用法我们似曾相识。在介绍 build_phase 时，其 traverse 中通过如下的方式来探测是不是手动的调用了 build_phase。

```
文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase
函数/任务：traverse
 75                     comp.m_phasing_active++;
 76                     if (comp.m_phase_imps.exists(this))
 77                        ph = comp.m_phase_imps[this];
 78                     ph.execute(comp, phase);
 79                     comp.m_phasing_active--;
```

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：build

2278  function void uvm_component::build();
2279     m_build_done = 1;
2280     apply_config_settings(print_config_matches);
2281     if(m_phasing_active == 0) begin
2282        uvm_report_warning("UVM_DEPRECATED", "build()/build_phase() has been called explicitly, outside of the phasing system. This usage of build is depr      ecated and may lead to unexpected behavior.");
2283     end
2284  endfunction
```

这里的用法是只具其形，但是不具其意。整个 uvm 的代码中除了在这里给 m_num_procs_not_yet_returned 赋值外，并没有在其它地方使用。所以我们可以猜测，这应该是 UVM 还在开发中的一个功能。

142 行与 143 行是与 sequence 机制相关的，其意思就是如果这是一个 sequencer 的话，那么就看看是否给此 sequencer 设置了 default_sequence，如果有则启动。这里不多做介绍，后面有专门的章节介绍 sequence 机制。

145 行调用 exec_task，而 exec_task 接下来会调用相应的 component 的 task，如 pre_reset_phase()，main_phase()等。

这里的关键是 150 行。由于使用了 join_none 的形式，所以并不等待这些 task 执行完毕就直接退出 execute 了。所以说类似 run_phase 等这些花费时间的 task，它们是通过自底向上的方式 fork 起来，但是却是同时运行的。

回到 execute_phase，其 1194 到 1208 行之间也是以 join_none 的形式的进程，所以会直接运行到 1210 行。这个 task 与 objection 机制相关，暂且先跳过。

```
文件：src/base/uvm_phase.svh
类：uvm_phase
```

函数/任务：execute_phase

```
1163    if (m_phase_type != UVM_PHASE_NODE) begin
           …
1168    end
1169
1170
1171    else begin // PHASE NODE
           …
1182       if (!$cast(task_phase,m_imp)) begin
              …
1191       end
1192       else begin
              …
1212          // Now wait for one of three criterion for end-of-phase.
1213          fork
1214          begin // guard
1215
1216             do begin
1217
1218              // if looped back from READY_TO_END, change state
1219              m_state = UVM_PHASE_EXECUTING;
1220              #0;
1221
1222              fork
1223
1224                 // WAIT_FOR_ALL_DROPPED
1225                 begin
1226                    // OVM semantic: don't end until objection raised or stop request
1227                    if (phase_done.get_objection_total(top) ||
1228                          m_use_ovm_run_semantic && m_imp.get_name() == "run") begin
1229                       if (!phase_done.m_top_all_dropped)
1230                          phase_done.wait_for(UVM_ALL_DROPPED, top);
1231                       `UVM_PH_TRACE("PH/TRC/EXE/ALLDROP","PHASE EXIT ALL_DR
OPPED",this,UVM_DEBUG)
1232                    end
1233                    else begin
1234                        if (m_phase_trace)
1235                           `UVM_PH_TRACE("PH/TRC/SKIP","No objections raised, skipping
phase",this,UVM_LOW)
1236                    end
1237                 end
1238
1239                 // TIMEOUT
1240                 begin
1241                    if (top.phase_timeout == 0)
1242                       wait(top.phase_timeout != 0);
1243                    `uvm_delay(top.phase_timeout)
1244                    if ($time == `UVM_DEFAULT_TIMEOUT) begin
1245                       `uvm_error("PH_TIMEOUT",
1246                             $sformatf("Default phase timeout of %0t hit. All processes are wait
ing, indicating a probable testbench issue. Phase '%0s' ready        to end",
```

```
1247                                    top.phase_timeout, get_name()))
1248                    end
1249                    else begin
1250                       `uvm_error("PH_TIMEOUT",
1251                           $sformatf("Phase timeout of %0t hit, phase '%0s' ready to end",
1252                                    top.phase_timeout, get_name()))
1253                    end
1254                    phase_done.clear(this);
1255                    `UVM_PH_TRACE("PH/TRC/EXE/3","PHASE EXIT TIMEOUT",this,UVM
_DEBUG)
1256                 end
1257
1258              join_any
1259              disable fork;
1260
1261              phase_done.clear(this);
1262
1263              // If jump is pending, do not allow prolonging of phase
1264              if(!m_jump_fwd && !m_jump_bkwd) begin
1265
1266                 //--------------
1267                 // READY_TO_END:
1268                 //--------------
1269
1270                  `UVM_PH_TRACE("PH_READY_TO_END","PHASE READY TO END",thi
s,UVM_DEBUG)
1271                 //m_state = UVM_PHASE_READY_TO_END;
1272                 m_ready_to_end_count++;
1273                 if (m_ready_to_end_count < max_ready_to_end_iter) begin
1274                    if (m_phase_trace)
1275                       `UVM_PH_TRACE("PH_READY_TO_END_CB","CALLING READY_T
O_END CB",this,UVM_HIGH)
1276                    if (m_imp != null)
1277                       m_imp.traverse(top,this,UVM_PHASE_READY_TO_END);
1278                 end
1279                 #0; // LET ANY WAITERS WAKE UP
1280
1281                 // WAIT FOR PREDECESSORS
1282                 if (!phase_done.get_objection_total(top)) begin
1283                    m_state = UVM_PHASE_READY_TO_END;
1284                    m_wait_for_pred();
1285                 end
1286
1287              end
1288
1289           end
1290           while (phase_done.get_objection_total(top));
1291
1292        end
1293        join // guard
1294
1295     end
```

```
1296
1297     end
```

从 1213 到 1293 行则是用于监控结束仿真程序的。这里用到了如下的一个结构：

```
fork
   do  begin
      fork
         process1;
         process2;
      join_any
      othe_statements;
   end  while()
join
```

process1 是位于 1225 到 1237 行的语句。1227 行用于判断是不是有 objection 被 raise 起来了。假如对于当前的 phase，没有任何 objection 被 raise 起来，那么将会直接跳到 else 分支，并最终会退出 fork…join_any，到达 other_statement 语句。这里用到了 objection 机制，后面会专门的阐述。在这里，大家再次看到了，对于一个运行时的 phase，至少要调用一次 raise_objection，这样才能保证相应的代码被执行。process2 则是超时退出。

1261 行执行一些清理工作，1264 行判断是否发生了 phase 的 jump。1272 行把 m_ready_to_end_count 的值加 1。1273 行则根据 m_ready_to_end_count 的值来判断是否调用一些相关的 callback。这种判断方式比较令人费解，可能是牵到 UVM 一些正在开发但是还没有正式发布的功能。1277 行再次调用 traverse，这次传入的参数是 UVM_PHASE_READY_TO_END。在 m_traverse 的 115 行，当传入这个参数时，会调用 component 的 phase_ready_to_end：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：phase_ready_to_end

2372  function void uvm_component::phase_ready_to_end (uvm_phase phase);
2373  endfunction
```

这是一个空函数，用户可以通过扩展这个函数来实现自己的一些目标。

回到 execute_phase，这里 1282 行判断是否已经把所有的 objection 都被 drop 了。如果依然还有 objection，那么由后面的 while 语句可以看出，会继续进入下一次的循环。否则的就执行 1283 行。正常情况下，运行到此处时肯定是所有的 objection 都已经被 drop 了，不过如果遇到了 phase 的 jump 则可能会出现没有 drop 的情况（确认一下），这个在后面分析 phase 的 jump 的时候会仔细介绍。

1284 行调用 m_wait_for_pred，前面调用的时候，这个 task 并没有起到什么作用，在这里，则可能会起到作用。这个 task 的意思是等待兄弟姐妹 phase 执行完毕。对于 run_phase 来说，它的兄弟姐妹 phase 是什么？是动态运行 phase 中的最后一个

post_shutdown_phase。所以在这里，run_phase 会等待 post_shutdown_phase 完成，同样的 post_shutdown_phase 也会等待 run_phase 完成。那么这种互相等待会不会带来死锁的情况？不会。因为 1283 行已经把此 phase 的状态赋值成了 UVM_PHASE_READY_TO_END。如果在此之前 post_shutdown_phase 已经在等待 run_phase 了，那么由于 1283 行，相当于是 run_phase 完成了，所以由 post_shutdown_phase 调用的 m_wait_for_pred 会返回。反过来亦如此。

接下来的代码则与上节讲述的没有区别，不重复讲述。

13.3.7. 同一层次的 component 的 build_phase 的执行

在 3.1.4 节中曾经提到过如下问题：uvm 的 build_phase 是自上而下执行的，但是对于同一层次的 uvm_component 来说，执行顺序是怎么样的呢？像 driver 和 monitor 都是 agent 里面处于同一层次。这两者之间是怎么执行的呢？是按照代码书写的顺序吗？还是同时执行？

13.3.5 节在讲述 build_phase 的执行时，有下述代码：

文件：src/base/uvm_topdown_phase.svh
类：uvm_topdown_phase
函数/任务：traverse

```
 52    virtual function void traverse(uvm_component comp,
 53                                   uvm_phase phase,
 54                                   uvm_phase_state state);
 55      string name;
 56      uvm_domain phase_domain = phase.get_domain();
 57      uvm_domain comp_domain = comp.get_domain();
 58
 59      if (m_phase_trace)
 60      `uvm_info("PH_TRACE",$sformatf("topdown-phase phase=%s state=%s comp=%s comp.domain=%s phase.domain=%s",
 61              phase.get_name(), state.name(), comp.get_full_name(),comp_domain.get_name(),phase_domain. get_name()),
 62              UVM_DEBUG)
 63
 64      if (phase_domain == uvm_domain::get_common_domain() ||
 65          phase_domain == comp_domain) begin
 66          case (state)
 67            UVM_PHASE_STARTED: begin
 68              comp.m_current_phase = phase;
 69              comp.m_apply_verbosity_settings(phase);
 70              comp.phase_started(phase);
 71              end
                …
```

```
91              endcase
92        end
93        if(comp.get_first_child(name))
94          do
95            traverse(comp.get_child(name), phase, state);
96          while(comp.get_next_child(name));
97    endfunction
```

这里的关键是 93 行到 96 行。可以看出，这里是先执行第一个孩子及其后代的 build_phase，再执行第二个孩子及其后代的 build_phase，以此类推。这里的第一个、第二个孩子是怎么确定的呢？

先看 get_first_child 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：get_first_child

1841  function int uvm_component::get_first_child(ref string name);
1842    return m_children.first(name);
1843  endfunction
```

再看 get_next_child 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：get_next_child


1849  function int uvm_component::get_next_child(ref string name);
1850    return m_children.next(name);
1851  endfunction
```

可见，孩子的顺序就是 m_children 中 component 的顺序。那么 m_children 中孩子的顺序是如何确定的呢？在 uvm_component 的 new 函数中：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：new


1767    if (!m_parent.m_add_child(this))
1768      m_parent = null;
```

这里会调用 parent 的 m_add_child 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：m_add_child

1796  function bit uvm_component::m_add_child(uvm_component child);
```

```
1797
1798    if (m_children.exists(child.get_name()) &&
1799        m_children[child.get_name()] != child) begin
1800        `uvm_warning("BDCLD",
1801          $sformatf("A child with the name '%0s' (type=%0s) already exists.",
1802              child.get_name(), m_children[child.get_name()].get_type_name()))
1803        return 0;
1804    end
1805
1806    if (m_children_by_handle.exists(child)) begin
1807        `uvm_warning("BDCHLD",
1808          $sformatf("A child with the name '%0s' %0s %0s'",
1809                    child.get_name(),
1810                    "already exists in parent under name '",
1811                    m_children_by_handle[child].get_name()))
1812        return 0;
1813      end
1814
1815    m_children[child.get_name()] = child;
1816    m_children_by_handle[child] = child;
1817    return 1;
1818  endfunction
```

1815 行把 child 加入到 parent 的 m_children 数组中。m_children 是一个联合数组：

```
文件：src/base/uvm_component.svh
类：uvm_component

1621    protected        uvm_component m_children[string];
```

m_children 的索引类型是字符串类型的，那么其记录的内容是如何排列的呢？是按照加入的顺序吗？

在 IEEE Std1800-2009，即《IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language》的 7.8.2 节(Page 112)中有如下说明：

Associative arrays that specify a string index have the following properties:

— Indices can be strings or string literals of any length. Other types are illegal and shall result in a type check error.

— An empty string “” index is valid.

— The ordering is lexicographical (lesser to greater).

其中的第三条说明数组中记录的内容是以字典顺序排列的，而不是按照加入的顺序。假设某 component 名字为 pa，下面有 A，B，C，D 四个 component，其定义的顺序为：

```
uvm_component B;
uvm_component D;
uvm_component C;
```

```
uvm_component A;
```

而这四个 component 实例化的顺序为：

```
C = new("C", this);
B = new("B", this);
D = new("D", this);
A = new("A", this);
```

即加入 pa 的 m_children 的顺序为 C，B，D，A，但是这四条记录将会在 m_children 中排列为 A，B，C，D，最终执行 build_phase 的时候不是按照 C，B，D，A 的顺序执行，也不是 B，D，C，A（即声明的顺序）执行，而是按照 A，B，C，D 的顺序执行。

13.4. objection 机制

上面一节在讲述一个 task phase 在调用 execute_phase 时用到了 objection 机制。本节将会详细的讲述。

13.4.1. uvm_phase 中的 phase_done

在 uvm_phase 的 new 函数中，648 到 542 行会实例化一个 phase_done 的变量：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：new

 648    if (name == "run")
 649      phase_done = uvm_test_done_objection::get();
 650    else begin
 651      phase_done = new(name);
 652    end
```

如果是 run_phase 的话，那么就把 phase_done 的值赋为 uvm_test_done_objection::get 的返回值。我们见识了 uvm_root 的 get，见识了 uvm_factory 的 get，所以这里我们很容易猜测，这是返回一个 uvm_test_done_objection 类型的静态变量。事实确实如此：

```
文件：src/base/uvm_objection.svh
类：uvm_test_done_objection

 960 //------------------------------------------------------------------------------
 961 //
 962 // Class- uvm_test_done_objection DEPRECATED
 963 //
 964 // Provides built-in end-of-test coordination
 965 //------------------------------------------------------------------------------
 966
 967 class uvm_test_done_objection extends m_uvm_test_done_objection_base;
 968
 969   protected static uvm_test_done_objection m_inst;
       …
1208   static function uvm_test_done_objection get();
1209     if(m_inst == null)
1210       m_inst = uvm_test_done_objection::type_id::create("run");
1211     return m_inst;
1212   endfunction
1213
1214 endclass
```

uvm_test_done_objection 是一个派生自 uvm_objection 的类，这里纯粹是为了与 OVM 兼容，在这个类的代码中，大家随处可见 DEPRECATED 这个词。因此这里将不会过多的讲述这个类。

回到 uvm_phase 的 new 函数中，651 行，如果不是 run，那么就新实例化一个 uvm_objection 类型的变量。

在 13.3.1 节讲述的 UVM 的 phase 运行图中，大家可以看到完整的 phase。这些 phase 结点各自有一个 phase_done，而且，更加要注意的是，这一幅图是在 0 时刻就已经建立好了，所以相应的，在 0 时刻就有了很多 uvm_objection 的实例。这么多的实例，实例，都放在 uvm_objection::m_objections 中：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：new

 135   function new(string name="");
 136     uvm_cmdline_processor clp;
 137     string trace_args[$];
 138     super.new(name);
 139     set_report_verbosity_level(m_top.get_report_verbosity_level());
 140
 141     // Get the command line trace mode setting
 142     clp = uvm_cmdline_processor::get_inst();
 143     if(clp.get_arg_matches("+UVM_OBJECTION_TRACE", trace_args)) begin
 144       m_trace_mode=1;
 145     end
 146     m_objections.push_back(this);
```

```
147     endfunction
```

从这个构造函数可以看出，每当有一个 uvm_objection 被实例化时，就会把这个实例的指针放入 m_objections 中。

13.4.2. raise_objection

通常，我们会在某个 sequence 中以如下的方式 raise_objection：

```
if(starting_phase != null)
   starting_phase.raise_objection(this);
```

我们之前已经提到过，这个 starting_phase 其实就是这个 sequence 对应的 sequencer，假设此 sequencer 的路径是 uvm_test_done.env.agent.sqr，后面会用到这个路径。

raise_objection 的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：raise_objection

 316    virtual function void raise_objection (uvm_object obj=null,
 317                                           string description="",
 318                                           int count=1);
 319      if(obj == null)
 320        obj = m_top;
 321      m_cleared = 0;
 322      m_raise (obj, obj, description, count);
 323    endfunction
```

这里首先判断传入的 obj 参数是否为 null，如果是的话，则把 uvm_root 的作为 obj。把 m_cleared 置为 0，这是为了后面跳出 phase 做准备工作的。之后调用 m_raise：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_raise

 328    function void m_raise (uvm_object obj,
 329                           uvm_object source_obj,
 330                           string description="",
 331                           int count=1);
 332
 333      if (m_total_count.exists(obj))
 334        m_total_count[obj] += count;
 335      else
```

```
336         m_total_count[obj] = count;
337
338       if (source_obj==obj) begin
339         if (m_source_count.exists(obj))
340           m_source_count[obj] += count;
341         else
342           m_source_count[obj] = count;
343       end
344
345       if (m_trace_mode)
346         m_report(obj,source_obj,description,count,"raised");
347
348       raised(obj, source_obj, description, count);
349
350       // If this object is still draining from a previous drop, then
351       // raise the count and return. Any propagation will be handled
352       // by the drain process.
353       if (m_draining.exists(obj))
354         return;
355
356       if (!m_hier_mode && obj != m_top)
357         m_raise(m_top,source_obj,description,count);
358       else if (obj != m_top)
359         m_propagate(obj, source_obj, description, count, 1, 0);
360
361     endfunction
```

m_raise 有两个参数，一个是 obj，另外一个是 source_obj，这两个参数的意思让人误解，不过慢慢看后面就明白了。333 到 343 行用到了两个联合数组，其定义分别为：

```
文件：src/base/uvm_objection.svh
类：uvm_objection

  65    protected int        m_source_count[uvm_object];
  66    protected int        m_total_count [uvm_object];
```

这两个联合数组的索引都是 uvm_object 型的，而存放的数据是 int 类型的。333 到 336 行向 m_total_count 插入或更新一条记录。如果我们连续两次调用如下语句：

```
starting_phase.raise_objection(this);
```

那么第一次调用的时候，m_total_count 是没有关于这个 sequence 的记录的，会执行 336 行。当第二次调用的时候，已经有了关于这个 sequence 的一条记录，那么就会执行 334 行。

338 到 343 的情况与上面相似，会在 m_source_count 中插入或者更新一条记录。不过这里的区别是 338 行加了一个判断条件来判断 obj 与 source_obj 是不是指向同一个 object。暂且把这个疑问搁置，后面会有涉及。

345 与 346 行是用于追踪 objection 变化状态时打印信息，直接跳过。后面会有很多关于这种打印信息，不另外说明，直接跳过。

348 调用 raised 函数：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：raised

 651    virtual function void raised (uvm_object obj,
 652                                  uvm_object source_obj,
 653                                  string description,
 654                                  int count);
 655      uvm_component comp;
 656      if ($cast(comp,obj))
 657        comp.raised(this, source_obj, description, count);
 658      if (m_events.exists(obj))
 659         ->m_events[obj].raised;
 660    endfunction
```

这个函数比较简洁，656 行判断传入的 obj 是不是一个 component。如果我们这么写：

```
task my_driver::main_phase(uvm_phase phase);
    super.main_phase(phase);
    phase.raise_objection(this);
    …
endtask
```

此时由于传给 raise_objection 的参数是 this，即 my_driver 的指针，因此这是一个 component，所以 657 行会执行 my_driver 的 raised 函数，这个函数位于 uvm_component 中：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：raised

1051    virtual function void raised (uvm_objection objection, uvm_object source_obj,
1052          string description, int count);
1053    endfunction
```

这是一个空函数，用户可以重载这个函数，以便在 raise_objection 之前做一些特殊的动作。

在我们现在的例子中，是在一个 sequence 内部 raise_objection，所以 657 行不会执行。658 行判断 m_events 中是否有 obj 的记录。m_events 的定义为：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
```

```
69    protected uvm_objection_events m_events [uvm_object];
```

这是一个联合数组，索引是 uvm_object 弄变量，而存储内容是 uvm_objection_events 类型。uvm_objection_events 是一个比较简单的类：

```
文件：src/base/uvm_objection.svh
类：uvm_objection_events

34 class uvm_objection_events;
35    int waiters;
36    event raised;
37    event dropped;
38    event all_dropped;
39 endclass
```

这个类只是封装了几个简单的事件。

658 与 659 行触发一个事件，这个事件虽然触发了，但是并没有代码在等待这个事件的发生。事实上，用户可以自定义一些代码，利用这个被触发的事件。

回到 raise_objection。353 与 354 行主要是用于处理当 drop_objection 之后，由于设置了 drain_time，objection 并没有真正的 drop，在真正的被 drop 之前，又重新 raise_objection。这种情况在后面介绍 drop_objection 时会说明。

整个 objection 机制比较精彩的地方反映在 356 到 359 行。356 行出现了变量 m_hier_mode：

```
文件：src/base/uvm_objection.svh
类：uvm_objection

80    /*protected*/ bit m_hier_mode = 1;
```

它的默认值为 1。当它为 0 的时候，356 行的判断条件成立，此时会调用 m_raise 函数，不过此时第一个参数变成了 m_top。m_top 就是 uvm_root 的单实例。

在这次调用的时候，333 到 336 的代码会执行，但是 338 到 343 的代码不会执行，因为 338 行会判断 obj 是否与 source_obj 指向同一个东西。在这次调用中，obj 指的是 uvm_root，而 source_obj 指的是 my_sequence。两者不相等，所以 m_source_count 中不会增加也不会更新一条记录。在这种情况下，raise_objection 执行完毕后，m_source_count 中只有一条记录：

```
m_source_count[my_sequence]=1
```

而 m_total_count 中则有两条记录：

```
m_total_count[my_sequence]=1;
m_total_count[uvm_root]=1;
```

我们接着看 356 行的条件不满足，进入到 358 行，这里会判断一下 obj 是否为 uvm_root，接下来会调用 m_propagate 函数。这个函数的定义为：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_propagate

 239    function void m_propagate (uvm_object obj,
 240                               uvm_object source_obj,
 241                               string description,
 242                               int count,
 243                               bit raise,
 244                               int in_top_thread);
 245      if (obj != null && obj != m_top) begin
 246        obj = m_get_parent(obj);
 247        if(raise)
 248          m_raise(obj, source_obj, description, count);
 249        else
 250          m_drop(obj, source_obj, description, count, in_top_thread);
 251      end
 252    endfunction
```

246 行得到 obj 的 parent。对于一个 component 来说，其 parent 的概念比较清楚，就是在其 new 函数中指定的 parent，但是对于一个 sequence 来说，这个 parent 指什么呢？

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_get_parent

 208    function uvm_object m_get_parent(uvm_object obj);
 209      uvm_component comp;
 210      uvm_sequence_base seq;
 211      if ($cast(comp, obj)) begin
 212        obj = comp.get_parent();
 213      end
 214      else if ($cast(seq, obj)) begin
 215          obj = seq.get_sequencer();
 216      end
 217      else
 218        obj = m_top;
 219      if (obj == null)
 220        obj = m_top;
 221      return obj;
 222    endfunction
```

从这段代码中可以看出，对于一个 sequence 来说，其 parent 就是 sequencer。

回到 m_propagate，247 行根据 raise 的值来判断接下来是调用 m_raise 还是 m_drop。刚刚我们传入的值为 1，所以这里会依然调用 m_raise，不过这次调用，传入的 obj 参数则变成了这个 sequence 的 sequencer。

那么在 m_raise 的 333 到 336 的代码会执行，但是 338 到 343 的不会执行。所以

运行到现在，m_source_count 中的记录依然只有一条：

```
m_source_count[my_sequence]=1
```

而 m_total_count 中则有两条：

```
m_total_count[my_sequence]=1;
m_total_count[env.agent.sqr]=1;
```

接下来我们注意到又运行到 359 行，调用 m_propagate，从调用 m_raise。这个过程会一直持续到传入 m_raise 的 obj 参数为 uvm_root 为止。整个递归调用全部结束后，m_source_count 中的记录依然只有一条：

```
m_source_count[my_sequence]=1;
```

而 m_total_count 中则有 5 条：

```
m_total_count[my_sequence]=1;
m_total_count[env.agent.sqr]=1;
m_total_count[env.agent]=1;
m_total_count[env]=1;
m_total_count[uvm_root]=1;
```

也就是说，当一个 object 把 raise_objection 时，不光这个 object 自己在 m_total_count 中有了一条记录，这个 object 的所有的先祖也有了一条记录，也即沿着整棵 UVM 树向上，所有的结点都在 m_total_count 中有了一条记录。而 m_source_count 中存放的是发起 raise_objection 的 object 的记录。

不过这种 object 的所有先祖都在 m_total_count 中有记录是在 m_hier_mode 为 1 的前提下才会发生的。为 0 有时候上面说过，则只是此 object 及 uvm_root 会在 m_total_count 中有记录。而默认的情况为 1，一般用户不需要改变这个值。

13.4.3. drop_objection

上节分析了 raise_objection，本节分析 drop_objection。通常的，我们在 sequence 中这么做：

```
if(starting_phase != null)
    starting_phase.drop_objection(this);
```

drop_objection 的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：drop_objection
```

```
423   virtual function void drop_objection (uvm_object obj=null,
424                                         string description="",
425                                         int count=1);
426     if(obj == null)
427       obj = m_top;
428     m_drop (obj, obj, description, count, 0);
429   endfunction
```

它会直接调用 m_drop：

文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_drop

```
434   function void m_drop (uvm_object obj,
435                         uvm_object source_obj,
436                         string description="",
437                         int count=1,
438                         int in_top_thread=0);
439
440     if (!m_total_count.exists(obj) || (count > m_total_count[obj])) begin
441       if(m_cleared)
442         return;
443       uvm_report_fatal("OBJTN_ZERO", {"Object \"", obj.get_full_name(),
444         "\" attempted to drop objection "",this.get_name(),"" count below zero"});
445       return;
446     end
447
448     if (obj == source_obj) begin
449       if (!m_source_count.exists(obj) || (count > m_source_count[obj])) begin
450         if(m_cleared)
451           return;
452         uvm_report_fatal("OBJTN_ZERO", {"Object \"", obj.get_full_name(),
453           "\" attempted to drop objection "",this.get_name(),"" count below zero"});
454         return;
455       end
456       m_source_count[obj] -= count;
457     end
458
459     m_total_count[obj] -= count;
460
461     if (m_trace_mode)
462       m_report(obj,source_obj,description,count,"dropped");
463
464     dropped(obj, source_obj, description, count);
465
466     // if count != 0, no reason to fork
467     if (m_total_count[obj] != 0) begin
468       if (!m_hier_mode && obj != m_top)
469         m_drop(m_top,source_obj,description, count, in_top_thread);
470       else if (obj != m_top) begin
471         this.m_propagate(obj, source_obj, description, count, 0, in_top_thread);
```

```
472         end
473
474       end
475       else begin
476         // need to make sure we are safe from the dropping thread terminating
477         // while the drain time is being honored. Can call immediatiately if
478         // we are in the top thread, otherwise we have to schedule it.
479         if(!m_draining.exists(obj)) m_draining[obj] = 1;
480         else m_draining[obj] = m_draining[obj]+1;
481
482         if(in_top_thread) begin
483           m_forked_drop(obj, source_obj, description, count, in_top_thread);
484         end
485         else
486         begin
487           uvm_objection_context_object ctxt;
488           if(m_context_pool.size())
489             ctxt = m_context_pool.pop_front();
490           else
491             ctxt = new;
492           ctxt.obj = obj;
493           ctxt.source_obj = source_obj;
494           ctxt.description = description;
495           ctxt.count = count;
496           m_scheduled_list.push_back(ctxt);
497         end
498       end
499
500   endfunction
```

m_drop 与 m_raise 一样，也同样有两个容易混淆的参数，一个是 obj，另外一个是 source_obj。相信看了上节，读者应该对这两个参数有了很深的了解了。

440 行判断输入的 obj 是否在 m_total_count 中有一条记录，以及要 drop 的 objection 的数量是不是大于被 raise_objection 的数量。这里的 441 行判断 m_cleared 值。这个变量只有在 clear 函数中会被置位，后面会详细介绍 clear。这里没有调用 clear，所以其值依然为 0。

448 到 457 行在 m_source_count 中把相应的 count 给减去。459 把 m_total_count 中的 count 数减去。之后 464 行调用 dropped 函数，这个函数与 raised 函数极其相似：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：dropped

668   virtual function void dropped (uvm_object obj,
669                                  uvm_object source_obj,
670                                  string description,
671                                  int count);
672     uvm_component comp;
673     if($cast(comp,obj))
```

```
674             comp.dropped(this, source_obj, description, count);
675         if (m_events.exists(obj))
676             ->m_events[obj].dropped;
677     endfunction
```

由于几乎与 raised 几乎一模一样，因此不多做介绍。这里会直接从这个函数返回到 467 行。从这一行开始进入 m_drop 函数的关键所在。

这一行判断 m_total_count 与这个 obj 有关的记录的数值是否为 0，如果不为 0，说明还有 objection 存在，那么接下来根据 m_hier_mode 的值调用 m_drop 或者 m_propagate。而 m_propatate 又会调用 m_drop。这也是一个递归调用的过程，假如我们 raise_objection 调用了两次：

```
starting_phase.raise_objection(this);
starting_phase.raise_objection(this);
```

那么调用完成后，m_total_count 和 m_source_count 的数值分别为：

```
m_source_count[my_sequence]=2;
m_total_count[my_sequence]=2;
m_total_count[env.agent.sqr]=2;
m_total_count[env.agent]=2;
m_total_count[env]=2;
m_total_count[uvm_root]=2;
```

那么调用一次 drop_objection 之后，这些值相应的更新为：

```
m_source_count[my_sequence]=1;
m_total_count[my_sequence]=1;
m_total_count[env.agent.sqr]=1;
m_total_count[env.agent]=1;
m_total_count[env]=1;
m_total_count[uvm_root]=1;
```

假如此时再调用一次 drop_objection，那么将会进入 479 行的分支。479 与 480 行则会在 m_draining 联合数组中插入或者更新一条记录。这个联合数组的定义为：

```
文件：src/base/uvm_objection.svh
类：uvm_objection

  68    protected int        m_draining      [uvm_object];
```

482 行判断 in_top_thread 的值。关于这个变量，到现在为止我们调用 m_propagate 和 m_drop 时输入的值都为 0。关于这个变量的用处，后面会介绍。

487 到 496 行会把一个 uvm_objection_context_object 型的变量放入 m_scheduled_list 中。这里用到了两个联合数组：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
```

```
77    local uvm_objection_context_object m_context_pool[$];
78    local uvm_objection_context_object m_scheduled_list[$];
```

这是两个队列，其中存储的数据是 uvm_objection_context_object 类型。uvm_objection_context_object 类的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection_context_object

1218 // Have a pool of context objects to use
1219 class uvm_objection_context_object;
1220    uvm_object obj;
1221    uvm_object source_obj;
1222    string description;
1223    int count;
1224 endclass
```

这是一个相当简单的类，只封装了几个简单的数据，且没有函数。

至此，整个函数结束。至于把一条记录放入 m_scheduled_list 中作什么用，下节将会介绍。

13.4.4. m_forked_drop

在我们分析源代码时，最早是在 uvm_root 的 run_test 的 323 行遇到了与 objection 相关的代码：

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：run_test

323    uvm_objection::m_init_objections();
```

m_init_objections 是一个静态函数，其定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_init_objections

 506    static function void m_init_objections();
 507      fork begin
 508        while(1) begin
 509          wait(m_objections.size() != 0);
 510          foreach(m_objections[i]) begin
 511            automatic uvm_objection obj = m_objections[i];
 512            fork
 513              begin
```

```
514                     obj.m_background_proc = process::self();
515                     obj.m_execute_scheduled_forks();
516                   end
517                 join_none
518               end
519               m_objections.delete();
520             end
521           end join_none
522         endfunction
```

这是一个外层包含着 fork…join_none 语句，509 行会等待 m_objections 里面有记录出现。12.4.1 节已经说过，当在 0 时刻时，m_objections 中已经有很了很多实例的指针。因此进入到 510 行。在接下来的几行中，做的事情就是对于每一个 uvm_objection 的实例，调用 m_execute_scheduled_forks，这个 task 的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_execute_scheduled_forks

528     // background process; when non
529     task m_execute_scheduled_forks;
530       uvm_objection_context_object ctxt;
531       while(1) begin
532         wait(m_scheduled_list.size() != 0);
533         if(m_scheduled_list.size() != 0) begin
534         ctxt = m_scheduled_list.pop_front();
535         m_forked_drop(ctxt.obj, ctxt.source_obj, ctxt.description, ctxt.count, 1);
536         m_context_pool.push_back(ctxt);
537           end
538       end
539     endtasks
```

530 行用到了 532 行会等待 m_scheduled_list 中有记录出现。在上一节的末尾中，m_drop 把一条记录放入了 m_scheduled_list 中，于是此处把这条记录取出，调用 m_forked_drop 函数：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_forked_drop

545     function void m_forked_drop (uvm_object obj,
546                                  uvm_object source_obj,
547                                  string description="",
548                                  int count=1,
549                                  int in_top_thread=0);
550
551       int diff_count;
552
553       fork    // join_none, so this can be a function
554       begin   //  also serves as guard proc to embedded disable fork
```

```
555
556         fork
557           begin
558             if (m_drain_time.exists(obj))
559               `uvm_delay(m_drain_time[obj])
560
561              if (m_trace_mode)
562                m_report(obj,source_obj,description,count,"all_dropped");
563
564             all_dropped(obj,source_obj,description, count);
565
566             // wait for all_dropped cbs to complete
567             wait fork;
568           end
569           wait (m_total_count.exists(obj) && m_total_count[obj] != 0);
570         join_any
571         disable fork;
572
573         m_draining[obj] = m_draining[obj] - 1;
574
575         if(m_draining[obj] == 0) begin
576
577         m_draining.delete(obj);
578
579         if(!m_total_count.exists(obj))
580           diff_count = -count;
581         else
582           diff_count = m_total_count[obj] - count;
583
584         // no propagation if a re-raise cancels the drop
585         if (diff_count != 0) begin
586           bit reraise;
587
588           if (diff_count > 0)
589             reraise = 1;
590           reraise = diff_count > 0 ? 1 : 0;
591
592           if (diff_count < 0)
593             diff_count = -diff_count;
594
595           // we are ready to delete the 0-count entries for the current
596           // object before propagating up the hierarchy.
597           if (m_source_count.exists(obj) && m_source_count[obj] == 0)
598               m_source_count.delete(obj);
599
600           if (m_total_count.exists(obj) && m_total_count[obj] == 0)
601              m_total_count.delete(obj);
602
603           if (!m_hier_mode && obj != m_top) begin
604             if (reraise)
605               m_raise(m_top,source_obj,description,diff_count);
606             else
```

```
607                 m_drop(m_top,source_obj,description, diff_count, 1);
608             end
609             else begin
610               if (obj != m_top)
611                 this.m_propagate(obj, source_obj, description, diff_count, reraise, 1);
612             end
613           end
614
615           end //m_draining == 0
616         end
617         join_none
618
619       endfunction
```

要注意的是，这里调用的时候传入的 in_top_thread 参数为 1。这个函数的函数体是一个 fork…join_none 语句，所以在 fork 里面有耗费时间的语句是允许的。

558 行判断 m_drain_time 中是否有 obj 的记录。m_drain_time 是一个联合数组：

```
文件：src/base/uvm_objection.svh
类：uvm_objection

  67    protected time       m_drain_time   [uvm_object];
```

这个联合数组的索引是 uvm_object 类型，而存储的内容则是 time 型的。当使用 set_drain_time 时，会在这个数组中插入一条记录：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：set_drain_time

 633    // AE: set_drain_time(drain,obj=null)?
 634    function void set_drain_time (uvm_object obj=null, time drain);
 635      if (obj==null)
 636        obj = m_top;
 637      m_drain_time[obj] = drain;
 638      m_set_hier_mode(obj);
 639    endfunction
```

假设我们在某个地方(如某个 component 的 main_phase 中)设置了：

```
phase.set_drain_time(1000);
```

那么相应的，在 m_drain_time 中多了这么一条记录：

```
m_drain_time[uvm_root] = 1000;
```

回到 558 行，这里会判断是否有一条 obj 的记录。obj 是什么？我们在一个 sequence 中调用了 drop_objection，之后 m_drop 把一条记录放入了 m_scheduled_list 中，这条记录中的 obj 指的是这个 sequence，所以这里的 obj 指的还是这个 sequence。而 m_drain_time 中并没有这个 sequence 的记录，于是会跳到 564 行执行 all_dropped

函数：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：all_dropped

 687    virtual task all_dropped (uvm_object obj,
 688                              uvm_object source_obj,
 689                              string description,
 690                              int count);
 691      uvm_component comp;
 692      if($cast(comp,obj))
 693        comp.all_dropped(this, source_obj, description, count);
 694      if (m_events.exists(obj))
 695        ->m_events[obj].all_dropped;
 696      if (obj == m_top)
 697        m_top_all_dropped = 1;
 698    endtask
```

这里依然会调用 component 的函数 all_dropped，这也是个空函数，用户可以重载这个函数。694 行则触发 all_dropped 事件。

接下来会跳到 571 行。573 行把 m_draining 中有关于 obj 的记录的数值减 1。在 drop_objection 的 479 与 480 行，我们曾经把这个的值更新过。所以这里的条件符合，于是 577 行把 m_draining 中与此 obj 相关的记录删除。

579 到 582 行判断在 drop_objection 之后，而在 drain_time 没有完结之前，有没有新的 raise_objection。

假如没有被 raise，那么 diff_count 的值将会执行 582 行的分支，由于 count 的值为 1，而 m_total_count 中有关此 obj 的值已经变为了 0，所以 diff_count 的值将会是 -1。

假设有一个 objection 被 raise 起来，那么 m_total_count 的值将会是 1，相应的 diff_count 的值将会是 0，假如有两个 objection 被 raise 起来，那么 diff_count 的值将会是 1。

585 到 590 行根据 diff_count 的值给 reraise 进行赋值，这里排除了 diff_count 为 0 的情况。如果 reraise 表示有新的 objection 重新被 raise，那么 diff_count 为 0 的情况下，reraise 应该为 1，这看起来非常像一个 bug，是不是这样呢？我们暂且存下这个疑问，接着往后看。597 到 601 行比较容易理解，把 m_total_count 和 m_source_count 的有关于 obj 的记录删除，前提当然是这两条记录均已经为 0 了。

603 行又出现了 m_hier_mode，关于此变量，前面说过，一般为 1，所以直接进入 609 行的 else 分支。在这个分支中将会调用 m_propagate。相应的，把 reraise 作为参数。在 m_propagate 中将会根据这个参数的值来决定调用 m_raise 还是 m_drop。另外，此次调用时，最后一个参数为 1，表明这是最顶层的进程调用的这个函数。

假设没有新的 raise_objection，于是 m_propagate 将会调用 m_drop。这次调用时传入的 obj 将会变成了 sequence 的 parent，即其 sequencer。程序运行到 482 行时，由于此次调用是由最顶层的守护程序引起的，传入的 in_top_thread 参数为 1，所以会执行 483 行的分支。而在这里又会调用 m_forked_drop。这次调用，运行到 558 行时，判断条件依然不成立，于是接下来的执行过程跟我们刚才分析的一样，运行到 611 行时又会调用一次 m_propagate。m_propagate 又将会调用 m_drop。如此往复，直到传入 m_drop 的第一个参数是 uvm_root 时，情况才会改变。此时 483 行调用 m_forked_join 时，传入的第一个参数是 uvm_root。

在 m_forked_drop 中，558 行的判断条件成立了，于是延时 m_drain_time[uvm_root] 的时间。564 行调用 all_dropped，这次调用的时候会触发 m_events[uvm_root].all_dropped 事件。大家还记得 execute_phase 的 1230 行有如下的语句：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：execute_phase

1230                        phase_done.wait_for(UVM_ALL_DROPPED, top);
```

这里的 wait_for 的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：wait_for

 722    task wait_for(uvm_objection_event objt_event, uvm_object obj=null);
 723
 724        if (obj==null)
 725          obj = m_top;
 726
 727        if (!m_events.exists(obj)) begin
 728          m_events[obj] = new;
 729        end
 730
 731        m_events[obj].waiters++;
 732        case (objt_event)
 733          UVM_RAISED:          @(m_events[obj].raised);
 734          UVM_DROPPED:         @(m_events[obj].dropped);
 735          UVM_ALL_DROPPED: @(m_events[obj].all_dropped);
 736        endcase
 737
 738        m_events[obj].waiters--;
 739
 740        if (m_events[obj].waiters == 0)
 741          m_events.delete(obj);
 742
 743     endtask
```

这里将会在 m_events 中插入一条 uvm_root 的记录。由于输入的是 UVM_ALL_DROPPED，所以将会等待 m_events[uvm_root].all_dropped 事件的触发。all_dropped 函数满足了这个触发条件，所以 execute_phase 的等待进程结束。注意，此时是是 drop_objection 之后，加上了 drain_time 之后，也就是说，execute_phase 的后续动作都是在 drain_time 之后的了。

回到 m_forked_drop 中来，程序运行到 610 行时，由于输入的 obj 是 uvm_root，所以这里的条件不满足，于是直接退出，不会再次调用 m_propagate 开启新的进程。

假设有新的 raise_objection，那么 raise_objection 会调用 m_raise，而后者运行到 354 行时会直接返回，因为此时 m_draining 中有此 obj 记录，表明此 obj 正在处理 drain_time 期间。这里看来似乎有问题，因为只更新了此 obj 的 m_total_count，而其 parent 的则没有更新。其实不然，在有新的 raise_objection 时，611 行调用 m_propagate，传入的 reraise 为 1，接下来会调用 m_raise，从而把此 obj 的 parent 的 m_total_count 更新。

现在回头看一下 585 到 590 行我们认为是 bug 的那段代码，假如由我们来写，让 diff_count 为 0 的时候，reraise 为 1，那么 611 行的语句会调用 m_raise，此时传入的 diff_count 为 0，也就是让所有的父 component 的 m_total_count 都加 0。这跟不回其实效果是一样的，所以 585 到 590 直接把 diff_count 为 0 的情况排除了。所以这不是 bug，而是非常巧妙的设计。

13.4.5. systemverilog 中关于函数的调度语义

上一节在讲述 m_forked_drop 语句时，其 579 到 582 行胜于判断在 drop_objection 之后，而在 drain_time 没有完结之前，有没有新的 raise_objection。假设有一个 objection 被 raise 起来，那么 m_total_count 的值将会是 1，相应的 diff_count 的值将会是 0，假如有两个 objection 被 raise 起来，那么 diff_count 的值将会是 1。

这里其实存在着一些潜在的问题。我们回顾 m_raise 函数：

文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：m_raise

```
328    function void m_raise (uvm_object obj,
329                           uvm_object source_obj,
330                           string description="",
331                           int count=1);
332
333      if (m_total_count.exists(obj))
334        m_total_count[obj] += count;
```

```
335       else
336          m_total_count[obj] = count;
337
338       if (source_obj==obj) begin
339          if (m_source_count.exists(obj))
340             m_source_count[obj] += count;
341          else
342             m_source_count[obj] = count;
343       end
344
345       if (m_trace_mode)
346          m_report(obj,source_obj,description,count,"raised");
347
348       raised(obj, source_obj, description, count);
349
350       // If this object is still draining from a previous drop, then
351       // raise the count and return. Any propagation will be handled
352       // by the drain process.
353       if (m_draining.exists(obj))
354          return;
355
356       if (!m_hier_mode && obj != m_top)
357          m_raise(m_top,source_obj,description,count);
358       else if (obj != m_top)
359          m_propagate(obj, source_obj, description, count, 1, 0);
360
361    endfunction
```

353 行，表示当系统处于 draining 状态（上一次的 drain_time 还没有结束）时，那么 m_raise 函数会直接返回，接下来的用于向上层的 component 增加 objection 统计数的 356 到 359 行将不会被执行。当 333 到 336 行把 m_total_count 的值改变后，m_forked_drop 函数的 569 行将会马上得到这一信息，于是 577 行会把 obj 从 m_draining 中删除。这样带来的问题是，当 m_raise 执行到 353 行时发现 m_draining 中已经没有 obj 了，于是接下来继续执行 356 到 359 的语句，向上层的 component 增加 objection 统计数，同时 m_forked_drop 后面的代码也会向上层的 component 增加 objection 统计数，这样会出现重复计数，从而出现错误。这说到底其实是竞争的问题。那么到底会不会这样呢？我们把上述过程简化成如下的代码：

```
function raise_objection
    m_total_count++;
    ….
    statement A;
endfunction
function m_forked_drop
    fork
        wait(m_total_count != 0);
        …
        statement B;
    join_none
endfunction
```

上面的问题可以归结为 statement A 与 statement B 谁先执行的问题。这里其实牵扯到了 systemverilog 的调度语义。

在 IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language(IEEE Std 1800™-2009)的 4.9.7 节（Page 32）中有如下语句：

Subroutine argument passing is by value, and it copies in on invocation and copies out on return. The copy-out-on-the-return function behaves in the same manner as does any blocking assignment.

函数调用是子例程的一种（另外一种是 task）。这 里明确说明了 function 类似于一个阻塞赋值语句。一个阻塞赋值语句在其执行完毕后，其产生的影响才会扩展到其它语句。因此，上面的问题中，raise_objection 是一个函数，在函数体一开始的时候就改变了 m_total_count 的值。但是这个值的改变并不会马上影响到 m_forked_drop 函数，而要等到整个 raise_objection 执行完毕后才会影响到 m_forked_drop 函数。也就是说，函数做为一个整体执行，而不会分散开来。所以，statement A 一定是先于 statement B 执行的。从而，m_raise 与 m_forked_drop 函数都能正常工作，即整个 objection 可以正常工作，我们之前的担心是多余的，竞争情况不会出现。

13.4.6. 可以写一点与 execute_phase 相关的

13.5. phase 的高级应用

13.5.1. phase 的 jump

phase 的 jump 一直是前面在介绍 execute_phase 时回避的一点。本节详细阐述。

在某个 component 的 main_phase，我们可以通过如下的方式跳转：

```
task  main_phase(uvm_phase  phase);
    super.main_phase(phase);
```

```
    phase.raise_objecton(this);
    …
    phase.jump(uvm_reset_phase::get());
    …
    phase.drop_objection(this);
endtask
```

这里的 jump 的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：jump

1663 function void uvm_phase::jump(uvm_phase phase);
1664   uvm_phase d;
1665   // TBD refactor
1666
1667   if ((m_state <  UVM_PHASE_STARTED) ||
1668       (m_state >  UVM_PHASE_READY_TO_END) )
1669   begin
1670     `uvm_error("JMPPHIDL", { "Attempting to jump from phase \"",
1671       get_name(), "\" which is not currently active (current state is ",
1672       m_state.name(), "). The jump will not happen until the phase becomes ",
1673       "active."})
1674   end
1675
1676
1677
1678   // A jump can be either forward or backwards in the phase graph.
1679   // If the specified phase (name) is found in the set of predecessors
1680   // then we are jumping backwards.  If, on the other hand, the phase is in the set
1681   // of successors then we are jumping forwards.  If neither, then we
1682   // have an error.
1683   //
1684   // If the phase is non-existant and thus we don't know where to jump
1685   // we have a situation where the only thing to do is to uvm_report_fatal
1686   // and terminate_phase.  By calling this function the intent was to
1687   // jump to some other phase. So, continuing in the current phase doesn't
1688   // make any sense.  And we don't have a valid phase to jump to.  So we're done.
1689
1690   d = m_find_predecessor(phase,0);
1691   if (d == null) begin
1692     d = m_find_successor(phase,0);
1693     if (d == null) begin
1694       string msg;
1695       $sformat(msg,{"phase %s is neither a predecessor or successor of ",
1696                     "phase %s or is non-existant, so we cannot jump to it.  ",
1697                     "Phase control flow is now undefined so the simulation ",
1698                     "must terminate"}, phase.get_name(), get_name());
1699       `uvm_fatal("PH_BADJUMP", msg);
1700     end
1701     else begin
```

```
1702          m_jump_fwd = 1;
1703          `uvm_info("PH_JUMPF",$sformatf("jumping forward to phase %s", phase.get_name()),
1704                    UVM_DEBUG);
1705        end
1706      end
1707      else begin
1708        m_jump_bkwd = 1;
1709        `uvm_info("PH_JUMPB",$sformatf("jumping backward to phase %s", phase.get_name()),
1710                  UVM_DEBUG);
1711      end
1712
1713      m_jump_phase = d;
1714      m_terminate_phase();
1715  endfunction
```

1667 到 1674 行判断当前 phase 的状态，如果 phase 还没有执行或者已经执行完毕，那么是不能跳转的。

1690 行在当前 phase 的先驱 phase 中寻找是否能找到输入的 phase。如果找到了，则把 m_jump_bkwd 赋值为 1，说明是向后跳转，否则就再到后继 phase 中看看能否找到，如果找不到，说明输入的 phase 根本不存在，不会发生跳转；找到了则把 m_jump_fwd 置为 1，说明是向前跳转。

1713 把 m_jump_phase 赋值为要跳转的 phase，1714 行则调用 m_terminate_phase。这个函数的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：m_terminate_phase

1826  function void uvm_phase::m_terminate_phase();
1827    phase_done.clear(this);
1828  endfunction
```

这里就是调用 uvm_objection 的 clear 函数。这个函数的定义如下：

```
文件：src/base/uvm_objection.svh
类：uvm_objection
函数/任务：clear

  98    virtual function void clear(uvm_object obj=null);
  99      string name;
 100      if (obj==null)
 101        obj=m_top;
 102      name = obj.get_full_name();
 103      if (name == "")
 104        name = "uvm_top";
 105      else
 106        name = obj.get_full_name();
 107      if (!m_top_all_dropped && get_objection_total(m_top))
 108        uvm_report_warning("OBJTN_CLEAR",{"Object '",name,
```

```
109                 "' cleared objection counts for ",get_name()});
110       //Should there be a warning if there are outstanding objections?
111       m_source_count.delete();
112       m_total_count.delete();
113       m_draining.delete();
114       m_top_all_dropped = 0;
115       m_cleared = 1;
116       if (m_events.exists(m_top))
117         ->m_events[m_top].all_dropped;
118       m_background_proc.kill();
119
120       fork
121         begin
122           m_background_proc = process::self();
123           m_execute_scheduled_forks();
124         end
125       join_none
126     endfunction
```

111 到 113 行把相关的联合数组清空，115 行把 m_cleared 置位为 0。这里的关键是 117 及 118 行。117 行触发 all_dropped 事件，上节已经说过了，这会让 execute_phase 的 1230 等待进程结束。118 行则把 m_background_proc 进程给杀死。这是个什么进程？注意到 m_init_objections 的 514 行，这里启动了 m_background_proc 进程。120 到 125 行把刚刚杀死的进程重新启动起来。为什么需要重新启动起来？假如这是一个向后跳转，跳转到 reset_phase，那么接下来还要再运行一次 main_phase，那么由于这个进程是负责最终启动 m_forked_drop 的，也就是说用于结束仿真，所以如果不重新启动起来，那么后面的 main_phase 就没有办法运行了。

可见 m_terminate_phase 的作用就是让当前的 main_phase 结束，这反应在 execute_phase 的 1230 行的等待进程终结。之后继续运行到 1264 行时，此时条件不满足了，会直接跳转到 1314 行。

1314 与 1316 行的条件均满足，于是 1330 行会调用 traverse，传入的参数是 UVM_PHASE_ENDED。前面已经说过，这里主要是调用相应 component 的 phase_ended，这里不重复阐述。

1333 行把状态设置为跳转态。1335 行则用于杀死 m_phase_proc 进程。这个进程是在 execute_phase 的 1197 行启动起来的。

1340 行，如果发现是向前跳转的话，那么调用 clear_successors 函数：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：clear_successors

1750 function void uvm_phase::clear_successors(uvm_phase_state state = UVM_PHASE_DORMA
NT,
1751       uvm_phase end_state=null);
1752   if(this == end_state)
```

```
1753        return;
1754     clear(state);
1755     foreach(m_successors[succ])  begin
1756        succ.clear_successors(state,  end_state);
1757     end
1758  endfunction
```

这是一个递归调用的函数，用到了 clear 函数：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：clear

1737  function  void  uvm_phase::clear(uvm_phase_state  state  =  UVM_PHASE_DORMANT);
1738     m_state  =  state;
1739     m_phase_proc  =  null;
1740     phase_done.clear(this);
1741  endfunction
```

clear_successsors 函数的意思就是把后续的 phase 设置为特定值。1341 行调用的时候的参数是 UVM_PHASE_DONE，m_jump_phase。

假如我们在 main_phase 中这么写：

```
phase.jump(uvm_shutdown_phase::get());
```

那么这里将会把 post_shutdown_phase 和 pre_shutdown_phase 的状态设置为 UVM_PHASE_DONE，表示已经执行过了。

回到 execute_phase，1343 行又一次调用了 clear_successors 函，不过这次使用的是默认的参数。

如果是向前跳转到 shutdown_phase，那么这里就相当于是 shutdown_phase.clear_successors，它将会把它自己及所有的后继结点，即 shutdown_phase，post_shutdown_phase 设置为 UVM_PHASE_DOMAINT 状态，不过这两个结点本来就是这个状态，所以这句话其实对向前跳转是没有意义的。

如果是向后跳转到 reset_phase，那么由于从 reset_phase 一直到 main_phase，它们的状态都已经是执行了，这里会把从 reset_phase 一直到最后的 post_shutdown_phase 的状态设置为 UVM_PHASE_DOMAINT，以便重新把这些 phase 执行一次。

1344 和 1345 行把两个成员变量复位为 0，它们的使命已经完成了。1346 行把要跳转的 phase 放入了 m_phase_hooper，等待 m_run_phases 来取。1348 行把相关进程的记录从 m_phase_top_procs 中删除。这条记录是在 m_run_phases 的 1817 行被放入的。到此整个跳转已经结束。

13.5.2. domain 的使用

前面讲述的所有的都是基于系统默认的 domain 的情况，本节讲述使用多个 domain 的情况。通常的在某个 component 中可以这么做：

```
class my_comp extends uvm_component;
    uvm_domain new_domain;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        new_domain = new("new_domain");
        set_domain(new_domain);
    endfunction
    …
endclass
```

如上就可以把 my_comp 的实例将会从属于 new_domain，而不是系统默认的 domain。在 new_domain 实例化的时候，uvm_domain::m_domains 将会插入一条新的记录：

```
m_domains["new_domain"]=new_domain;
```

set_domain 是 uvm_component 的一个函数，其定义如下：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：set_domain

2410 function void uvm_component::set_domain(uvm_domain domain, int hier=1);
2411
2412    // build and store the custom domain
2413    m_domain = domain;
2414    define_domain(domain);
2415    if (hier)
2416       foreach (m_children[c])
2417          m_children[c].set_domain(domain);
2418 endfunction
```

这里把 m_domain 的值赋值为新的 domain，之后调用 define_domain 函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：define_domain

2386 function void uvm_component::define_domain(uvm_domain domain);
2387    uvm_phase schedule;
2388    //schedule = domain.find(uvm_domain::get_uvm_schedule());
2389    schedule = domain.find_by_name("uvm_sched");
2390    if (schedule == null) begin
2391       uvm_domain common;
```

```
2392        schedule = new("uvm_sched", UVM_PHASE_SCHEDULE);
2393        uvm_domain::add_uvm_phases(schedule);
2394        domain.add(schedule);
2395        common = uvm_domain::get_common_domain();
2396        if (common.find(domain,0) == null)
2397           common.add(domain,.with_phase(uvm_run_phase::get()));
2398     end
2399
2400  endfunction
```

2389 行查找是否有一个名为 uvm_sched 的结点，这个 domain 是我们刚刚实例化的，它里面是空的，所以这里 schedule 的值为 null。2392 行实例化一个 UVM_PHASE_SCHEDULE 型的结点，之后调用 add_uvm_phases 函数。还记得这个函数吗？本章一开始的时候就介绍了它，这里再把它贴出来：

```
文件：src/base/uvm_domain.svh
类：uvm_domain
函数/任务：add_uvm_phase

146     static function void add_uvm_phases(uvm_phase schedule);
147
148       schedule.add(uvm_pre_reset_phase::get());
149       schedule.add(uvm_reset_phase::get());
150       schedule.add(uvm_post_reset_phase::get());
151       schedule.add(uvm_pre_configure_phase::get());
152       schedule.add(uvm_configure_phase::get());
153       schedule.add(uvm_post_configure_phase::get());
154       schedule.add(uvm_pre_main_phase::get());
155       schedule.add(uvm_main_phase::get());
156       schedule.add(uvm_post_main_phase::get());
157       schedule.add(uvm_pre_shutdown_phase::get());
158       schedule.add(uvm_shutdown_phase::get());
159       schedule.add(uvm_post_shutdown_phase::get());
160
161     endfunction
```

函数的功能就是把 12 个动态运行的 phase 加入到这个新实例化的 UVM_PHASE_ SCHEDULE 结点里。2394 行则把这个 schedule 加入到我们新实例化的 domain 里面。这样的步骤其实跟我们之前讨论 m_uvm_domain 的生成过程一模一样。其实运行到 2394 行之后，这个新实例化的 domain 跟 m_uvm_domain 几乎就是一模一样的了，里面的内容相同。

2397 行则把这个新的 domain 加入到 m_common_domain 中，与 run_phase 并行执行。这与 m_uvm_domain 加入 common_domain 中时一模一样。

回到 set_domain 的 1415 行，这里会判断 hier 的值，如果为 1 的话，那么把这个 component 的所有的后代都加入到这个新的 domain 里面。如果为 0 的话，那么仅仅只有这个 component 是从属于这个新的 domain 的。所以可以控制这个参数值来达到想要的不同效果。

新实例化的 domain 被加入到 m_common_domain 中之后会如何运行呢？

我们来到 execute_phase 的 1404 行，假设此时是 start_of_simulation_phase，那么此时它有三个后继者，一是 run_phase，二是 m_uvm_domain，三是新加入的 domain。前面说过，m_uvm_domain 的 pre_reset_phase 其实会和 run_phase 在同一时刻被 fork 起来执行，同样的，新加入的 domain 的 pre_reset_phase 也会在这一时刻被 fork 起来执行。不过新的 domain 的 pre_resset_phase 不必和 m_uvm_phase 的 pre_reset_phase 同步，二者不必相互等待对方完成，这就是新建 domain 的优势所在。利用这一特性，可以做成很多事情。

13.5.3. 进程的同步

上节讲述了不同的 domain 之间，其内部相应的动态运行的 phase 不必同步。但是有时候还是要有同步的情况出现，如我们需要 common_domain 的 reset_phaset 和新 domain 的 configure_phase 同步，那么可以这么写：

```
uvm_domain common_domain;
common_domain=uvm_domain::get_common_domain();
common_domain.sync(new_domain, uvm_reset_phase::get(), uvm_configure_phase::get());
```

sync 的定义如下：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：sync

1539 function void uvm_phase::sync(uvm_domain target,
1540                                 uvm_phase phase=null,
1541                                 uvm_phase with_phase=null);
1542   if (!this.is_domain()) begin
1543     `uvm_fatal("PH_BADSYNC","sync() called from a non-domain phase schedule node");
1544   end
1545   else if (target == null) begin
1546     `uvm_fatal("PH_BADSYNC","sync() called with a null target domain");
1547   end
1548   else if (!target.is_domain()) begin
1549     `uvm_fatal("PH_BADSYNC","sync() called with a non-domain phase schedule node as
target");
1550   end
1551   else if (phase == null && with_phase != null) begin
1552     `uvm_fatal("PH_BADSYNC","sync() called with null phase and non-null with phase");
1553   end
1554   else if (phase == null) begin
1555     // whole domain sync - traverse this domain schedule from begin to end node and sy
nc each node
```

```
1556        int visited[uvm_phase];
1557        uvm_phase queue[$];
1558        queue.push_back(this);
1559        visited[this] = 1;
1560        while (queue.size()) begin
1561          uvm_phase node;
1562          node = queue.pop_front();
1563          if (node.m_imp != null) begin
1564            sync(target, node.m_imp);
1565          end
1566          foreach (node.m_successors[succ]) begin
1567            if (!visited.exists(succ)) begin
1568              queue.push_back(succ);
1569              visited[succ] = 1;
1570            end
1571          end
1572        end
1573      end else begin
1574        // single phase sync
1575        // this is a 2-way ('with') sync and we check first in case it is already there
1576        uvm_phase from_node, to_node;
1577        int found_to[$], found_from[$];
1578        if(with_phase == null) with_phase = phase;
1579        from_node = find(phase);
1580        to_node = target.find(with_phase);
1581        if(from_node == null || to_node == null) return;
1582        found_to = from_node.m_sync.find_index(node) with (node == to_node);
1583        found_from = to_node.m_sync.find_index(node) with (node == from_node);
1584        if (found_to.size() == 0) from_node.m_sync.push_back(to_node);
1585        if (found_from.size() == 0) to_node.m_sync.push_back(from_node);
1586      end
1587    endfunction
```

1542 行要求只有一个 domain 才能调用 sync 函数。1551 行禁止 phase 是 null，而 with_phase 不是 null 的调用方式，1554 行，则说明 phase 是 null，同时 with_phase 也是 null，1573 行则是 phase 不是 null，而 with_phase 可能是 null，也可能不是 null 的情况。我们先看 1573 行的这个分支，我们的例子中用的也是这个分支。

1578 行，如果 with_phase 是 null 的话，那么 with_phase 就与 phase 相同。1579 行用到了 find 函数：

```
文件：src/base/uvm_phase.svh
类：uvm_phase
函数/任务：find

1066  function uvm_phase uvm_phase::find(uvm_phase phase, bit stay_in_scope=1);
1067      // TBD full search
1068      //$display({"\nFIND node '",phase.get_name(),"' within ",get_name()," (scope ",m_phase_ty
pe.name(), ")", (stay_in_scope) ? " staying within scope" :        ""});
1069      if (phase == m_imp || phase == this)
1070        return phase;
```

```
1071     find = m_find_predecessor(phase,stay_in_scope,this);
1072     if (find == null)
1073         find = m_find_successor(phase,stay_in_scope,this);
1074  endfunction
```

这个函数从这个 phase 的所有先驱者和后继者中查找输入的 phase 是否在其先驱者或后继者中，查找到后就直接返回。from_node 这里指的就是 reset_phase，而 to_node 指的是 configure_phase，1582 行用于检查 from_node（即 reset_phase）的 m_sync 中是否有 to_node（即 configure_phase），1583 行用于检查 to_node（即 configure_phase）的 m_sync 中是否有 from_node（即 reset_phase）。

这里用到了一个 m_sync，这是一个队列，存放的内容是 uvm_phase 类型：

```
文件：src/base/uvm_phase.svh
类：uvm_phase

 553     local uvm_phase m_sync[$];   // schedule instance to which we are synced
```

很明显，这两个 m_sync 中互相都没有对方，于是接下来 1584 行在 from_node（即 reset_phase）的 m_sync 中插入了一条 to_node（即 configure_phase）的记录，1585 行在 to_node（即 configure_phase）的 m_sync 中插入了一条 from_node（即 reset_phase）的记录。

假如在执行 1573 行的分支时，with_phase 等于 null:

```
common_domain.sync(new_domain, uvm_reset_phase::get());
```

那么接下来在 common_domain 和 new_domain 的各自的 reset_phase 中，都会插入一条对方的 reset_phase 的记录。

假如 sync 如下调用：

```
common_domain.sync(new_domain);
```

此时执行 1554 行的分支，第一次进入 while 循环时，node=common_domain，它的 m_imp 为 null，之后则把 common_domain 的后继者 uvm_build_phase 放入了 queue 中，进行第二次 while 循环，此时 1563 行的条件满足，于是调用 sync 函数，这次调用执行的是 1573 行的分支。在这个分支中，1580 行的查找结果是 null，因为 new_domain 中并没有 build_phase，只有 12 个动态运行的 phase，于是直接返回到上一次 sync 调用的 while 循环中，此时把 build_phase 的后继者 connect_phase 作为参数调用 sync，其结果与 build_phase 类似。这种情况只有在当把 pre_reset_phase 调用时才会出现转变，此时会在 common_domain 和 new_domain 各自的 pre_reset_phase 的 m_sync 中插入一条关于对方 pre_reset_phase 的记录。接下来一直到 post_shutdown_phase 的情况也如此。也就是说，如果调用 sync 时 phase 和 with_phase 均为 null 的话，那么各自的 12 个 phase 将会和对方的 12 个 phase 同步。

当 m_sync 非空时，execute_phase 在 1150 行会等待 m_sync 中的 phase 达到指定

的状态。从而实现了 phase 的同步。

phase 中，同一级别的 driver 和 monitor 的执行顺序
super.*_phase 都做了什么事情。

14. field_automation 机制源代码分析

field_automation 机制在 UVM 的实现中并不算复杂，但是却非常实用，它实现了一些重复性工作的自动化执行。或许，它可以给我们一个提示：最复杂的东西并不一定是最有用的。恰如 20/80 原理提示的那样，用 20%的精力做出来的东西，是 80%的情况下会用到的；用 80%的精力做出来的东西，只有 20%的情况下才会被使用。

UMV 中把类分成了 uvm_object 与 uvm_component 两大类，相应的 field_automation 机制的实现在这两者之间有一定的差异。本章第一节讲述简单的 field_automation 机制，第二节深入剖析 field_automation 机制的原理。

14.1. 简单的 field_automation

14.1.1. 一个简单的例子

考虑如下的一个简单的例子：

```
class my_cfg extends uvm_object;
    rand bit [7:0] num;
    `uvm_object_utils_begin(my_cfg)
        `uvm_field_int(num, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

对于 uvm_object_utils_begin，其展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 226 `define uvm_object_utils_begin(T) \
 227       `m_uvm_object_registry_internal(T,T)    \
 228       `m_uvm_object_create_func(T) \
 229       `m_uvm_get_type_name_func(T) \
 230       `uvm_field_utils_begin(T)
```

这几个展开的宏中，除了 uvm_field_utils_begin 之外，其它的几个在 factory 机制时已经分析过。对于 uvm_field_utils_begin 宏，其实在 uvm_object_utils 之中也出现过：

```
文件：src/macros/uvm_object_defines.svh
类：无

 218 `define uvm_object_utils(T) \
 219     `uvm_object_utils_begin(T) \
 220     `uvm_object_utils_end
```

显然，uvm_object_utils 的展开中也是包括了 uvm_field_utils_begin 宏的。

而 uvm_object_utils_end 的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 237 `define uvm_object_utils_end \
 238         end \
 239       endfunction \
```

所以 uvm_field_int 是插在 uvm_field_utils_begin 与 uvm_object_utils_end 之间的，无论 uvm_field_int 宏是否出现，当使用 factory 机制时，uvm_field_utils_begin 是一定存在的。

14.1.2. uvm_field_utils_begin 宏

宏的展开为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 135 `define uvm_field_utils_begin(T) \
 136     function void __m_uvm_field_automation (uvm_object tmp_data__, \
 137                                             int what__, \
 138                                             string str__); \
 139     begin \
 140       T local_data__; /* Used for copy and compare */ \
 141       typedef T ___local_type____; \
 142       string string_aa_key; /* Used for associative array lookups */ \
 143       /* Type is verified by uvm_object::compare() */ \
 144       super.__m_uvm_field_automation(tmp_data__, what__, str__); \
 145       if(tmp_data__ != null) \
 146         /* Allow objects in same hierarchy to be copied/compared */ \
 147         if(!$cast(local_data__, tmp_data__)) return;
```

这个宏给人的第一印象就是相当不美观，其中变量的命名非常的不符合阅读习惯，前后加了很多下划线。这主要是为了避免这些变量名字和用户自定义的变量名字相同从而导致 field_automation 机制失效或出错。

我们把这个宏与 uvm_object_utils_end 宏合并，并把 T 用 my_cfg 来代替再展开：

```
function void __m_uvm_field_automation (uvm_object tmp_data__,
                                        int what__,
                                        string str__);
begin
  my_cfg local_data__;
  typedef T ___local_type____;
  string string_aa_key;
  super.__m_uvm_field_automation(tmp_data__, what__, str__);
  if(tmp_data__ != null)
    if(!$cast(local_data__, tmp_data__)) return;
end
endfunction
```

这样看起来就清晰了许多。这其实是重载了一个函数__m_uvm_field_automation，这个函数的原型位于 uvm_object 中：

```
文件：src/base/uvm_object.svh
类：uvm_object
函数/任务：__m_uvm_field_automation

1125 function void uvm_object::__m_uvm_field_automation (uvm_object tmp_data__,
1126                                                      int        what__,
1127                                                      string     str__ );
1128   return;
1129 endfunction
```

这是一个空函数，里面没有任何内容。整个由 uvm_object_utils_end 声明的函数只是一个函数头，我们看不出更多的东西来，比较让我们费解的是这个函数的几个参数。为了理解这几个参数是什么意思，我们要看一下我们平时是如何使用 field_automation 机制。

uvm_object 中有一个 copy 函数，它是应用 field_automation 的典型代表：

```
文件：src/base/uvm_object.svh
类：uvm_object
函数/任务：copy

1010 function void uvm_object::copy (uvm_object rhs);
1011   //For cycle checking
1012   static int depth;
1013   if((rhs !=null)  && (uvm_global_copy_map.get(rhs) != null)) begin
1014     return;
1015   end
1016
1017   if(rhs==null) begin
1018     uvm_report_warning("NULLCP", "A null object was supplied to copy; copy is ignored
", UVM_NONE);
1019     return;
1020   end
1021
1022   uvm_global_copy_map.set(rhs, this);
1023   ++depth;
1024
1025   __m_uvm_field_automation(rhs, UVM_COPY, "");
1026   do_copy(rhs);
1027
1028   --depth;
1029   if(depth==0) begin
1030     uvm_global_copy_map.clear();
1031   end
1032 endfunction
```

1025 行调用了__m_uvm_field_automation 函数，传入的第一个参数就是要拷贝的对像的指针，第二个参数用于表明这是一个 copy 操作，第三个参数是一个空的字符串。对于第三个参数，它的应用出现在 set_*_local 函数里，这里的*有 int，string，object。这三个函数的意思是把类中的某个成员变量变成 local 形式的。以 int 为例：

```
文件：src/base/uvm_object.svh
类：uvm_object
函数/任务：set_int_local

 923 function void  uvm_object::set_int_local (string         field_name,
 924                                            uvm_bitstream_t value,
 925                                            bit             recurse=1);
 926   if(__m_uvm_status_container.cycle_check.exists(this)) return;
 927   __m_uvm_status_container.cycle_check[this] = 1;
 928
 929   this.__m_uvm_status_container.status = 0;
 930   this.__m_uvm_status_container.bitstream = value;
 931
 932   __m_uvm_field_automation(null, UVM_SETINT, field_name);
 933
 934   if(__m_uvm_status_container.warning && !this.__m_uvm_status_container.status) begin
 935     uvm_report_error("NOMTC", $sformatf("did not find a match for field %s", field_nam
e),UVM_NONE);
 936   end
 937   __m_uvm_status_container.cycle_check.delete(this);
 938
 939 endfunction
```

932 行调用__m_uvm_field_automation 时，第三个参数设置为要设置的变量的名字。

至少我们应该明白，第一个参数表示当涉及到第二个对象的 field_objection 时，这个对象的指针，第二个参数表示 field_automation 的类型，第三个参数仅用于 set_*_local 系列函数，表明要设置的变量的名字。

14.1.3. 操作的类型

之前在介绍 field_automation 机制时，介绍到可以通过某些参数来把某个字段的相应功能去除，如：

```
`uvm_field_int(a, UVM_ALL_ON | UVM_NOCOPY)
```

类似于 UVM_NOCOPY 的定义位于 uvm_object_global.svh 文件中：

```
文件：src/base/uvm_object_globals.svh
类：无

181 //A=ABSTRACT  Y=PHYSICAL
182 //F=REFERENCE, S=SHALLOW, D=DEEP
183 //K=PACK, R=RECORD, P=PRINT, M=COMPARE, C=COPY
184 //-------------------------- AYFSD  K  R  P  M  C
```

```
185  parameter  UVM_DEFAULT       = 'b000010101010101;
186  parameter  UVM_ALL_ON        = 'b000000101010101;
187  parameter  UVM_FLAGS_ON      = 'b000000101010101;
188  parameter  UVM_FLAGS_OFF     = 0;
189
190  //Values  are  or'ed  into  a  32  bit  value
191  //and  externally
192  parameter  UVM_COPY            = (1<<0);
193  parameter  UVM_NOCOPY          = (1<<1);
194  parameter  UVM_COMPARE         = (1<<2);
195  parameter  UVM_NOCOMPARE       = (1<<3);
196  parameter  UVM_PRINT           = (1<<4);
197  parameter  UVM_NOPRINT         = (1<<5);
198  parameter  UVM_RECORD          = (1<<6);
199  parameter  UVM_NORECORD        = (1<<7);
200  parameter  UVM_PACK            = (1<<8);
201  parameter  UVM_NOPACK          = (1<<9);
202  //parameter  UVM_DEEP          = (1<<10);
203  //parameter  UVM_SHALLOW       = (1<<11);
204  //parameter  UVM_REFERENCE     = (1<<12);
205  parameter  UVM_PHYSICAL        = (1<<13);
206  parameter  UVM_ABSTRACT        = (1<<14);
207  parameter  UVM_READONLY        = (1<<15);
208  parameter  UVM_NODEFPRINT      = (1<<16);
209
```

可见，所谓的 UVM_ALL_ON，UVM_NOCOPY 等都是一些数字，这些数字的低 10 位用于表明操作的类型。这里一共有五种操作：copy，compare，print，record 和 pack。通过查看标志位的情况就可以知道是否执行某种操作。如对于例子中的 UVM_ALL_ON | UVM_NOCOPY，其值为'b000000101010111，在执行 copy 操作时，先看一下倒数第二位，发现此位被置起了，于是就不对此字段执行 copy 操作。

14.1.4. uvm_field_int 宏

有了前面的准备，我们把加了实际参数的 uvm_field_int 宏展开来：

```
文件：src/macros/uvm_object_defines.svh
类：无

542  `define  uvm_field_int(ARG,FLAG)  \
543     begin  \
544        case  (what__)  \
545           UVM_CHECK_FIELDS:  \
                 …
549           UVM_COPY:  \
550              begin  \
```

```
551                 if(local_data__ == null) return; \
552                 if(!((FLAG)&UVM_NOCOPY)) ARG = local_data__.ARG; \
553               end \
554           UVM_COMPARE: \
                  …
599         endcase \
600       end
```

我们以 copy 操作为例来进行分析，因此这里只列出了与 copy 操作相关的。这个看起来依然有点头大，不过我们可以结合前面的__m_uvm_field_automation 来看，这相当于是声明了如下的一个函数：

```
1 function void __m_uvm_field_automation (uvm_object tmp_data__,
2                                         int what__,
3                                         string str__);
4 begin
5    my_cfg local_data__;
6    typedef T ___local_type____;
7    string string_aa_key;
8    super.__m_uvm_field_automation(tmp_data__, what__, str__);
9    if(tmp_data__ != null)
10    if(!$cast(local_data__, tmp_data__)) return;
11   begin
12   case (what_)
13    …
14    UVM_COPY:
15    begin
16       if(local_data__ == null) return;
17        if(!((FLAG)&UVM_NOCOPY)) ARG = local_data__.ARG;
18    end
19    …
20    endcase
21    end
22end
23endfunction
```

当调用 copy 函数时，copy 最终会调用__m_uvm_field_automation，并把第二个参数设置为 UVM_COPY。函数会首先检查 tmp_data__即输入的参数是否为 null，如果不为 null，如对于我们的 copy 函数，会看一下输入的这个指针是不是 my_cfg 类型的，因为只有相同类同的类才能互相拷贝。

12 行根据输入的第二个参数，选择 14 行的分支。16 行判断要拷贝的数据源是否存在。17 行中的 FLAG 指的是我们之前输入的 UVM_ALL_ON | UVM_NOCOPY，其值为'b000000101010111。而 NOCOPY 的值为'b000000000000010，两者与操作的结果为 1，于是 17 行的判断条件不会成立，ARG 字段就不会执行拷贝操作。

当有更多的字段加进来时，只是相当于加入了 11 到 21 行的 begin…end。这样，当执行 copy 操作时，会逐字段的判断这个字段是否应该 copy。通过这种方式，UVM 实现了 field_automation 机制。

14.2. 高级的 field_automation 机制

14.2.1. __m_uvm_status_container

上节介绍了使用 field_automation 机制来实现 copy 操作，看上去非常的简单。不过如果你以为 field_automation 机制就是这么简单，那就错了。考虑 uvm_field_int 宏中与 compare 相关的部分：

```
文件：src/macros/uvm_object_defines.svh
类：无

 542 `define uvm_field_int(ARG,FLAG) \
 543   begin \
 544     case (what__) \
 545       UVM_CHECK_FIELDS: \
             … 
 554       UVM_COMPARE: \
 555         begin \
 556           if(local_data__ == null) return; \
 557           if(!((FLAG)&UVM_NOCOMPARE)) begin \
 558             if(ARG !== local_data__.ARG) begin \
 559               void'(__m_uvm_status_container.comparer.compare_field(`"ARG`", ARG, loc
al_data__.ARG, $bits(ARG))); \
 560               if(__m_uvm_status_container.comparer.result && (__m_uvm_status_containe
r.comparer.show_max <= __m_uvm_status_container.comparer.result) ) return; \
 561             end \
 562           end \
 563         end \
 564       UVM_PACK: \
             … 
 599     endcase \
 600   end
```

这里出现了我们不熟悉的一个变量__m_uvm_status_container，它是 uvm_object 的一个成员变量：

```
文件：src/base/uvm_object.svh
类：uvm_object

 778   static /*protected*/ uvm_status_container __m_uvm_status_container = new;
```

这是一个静态的，uvm_status_container 类型的成员变量。uvm_status_container 的定义位于 uvm_misc.svh 文件中，它的成员变量中有我们几个是不熟悉的：

```
文件：uvm_misc.svh
类：uvm_status_container

195 class uvm_status_container;
      ...
245   uvm_scope_stack scope   = new;
      …
260   uvm_comparer      comparer;
261   uvm_packer        packer;
262   uvm_recorder      recorder;
263   uvm_printer       printer;
264 endclass
```

uvm_scope_stack 的定义也位于 uvm_misc.svh 文件中，它主要的数据结构如下：

```
文件：uvm_misc.svh
类：uvm_scope_stack

 52 class uvm_scope_stack;
 53   local string m_arg = "";
 54   local string m_stack[$];
      …
183 endclass
```

成员变量都是字符串类型的，用于存放 UVM 中的路径，并提供了一些函数来对这些路径进行存取操作。其中比较重要的是 get，down，up，down_element 及 up_element。这几个函数的应用现在我们还没出现，等到出现时我们在具体的情景下分析。

uvm_status_container 中的 uvm_comparer，uvm_packer，uvm_recorder，uvm_printer 将会在后面陆续介绍。

14.2.2. compare 等操作

对于 uvm_object 中 uvm_status_container 类型静态成员变量 __m_uvm_status_container 来说，comparer 指的是一个全局的 comparer。

```
文件：src/base/uvm_object.svh
类：uvm_object
函数/任务：compare

1046 function bit  uvm_object::compare (uvm_object rhs,
1047                                     uvm_comparer comparer=null);
1048   bit t, dc;
1049   static int style;
```

```
1050    bit done;
1051    done = 0;
1052    if(comparer != null)
1053      __m_uvm_status_container.comparer = comparer;
1054    else
1055      __m_uvm_status_container.comparer = uvm_default_comparer;
1056    comparer = __m_uvm_status_container.comparer;
        …
1110  endfunction
```

在 compare 函数中，会传入一个 comparer 的变量，并且根据这个参数给__m_uvm_status_container 的 comparer 赋值。一般调用 compare 时，是不需要输入这个参数的。因此这个 comparer 的值就变成了 uvm_default_comparer，这是一个全局变量，它的定义位于 uvm_object_globals.svh 文件中：

```
文件：src/base/uvm_object_globals.svh
类：无

658 uvm_comparer uvm_default_comparer = new(); // uvm_comparer::init();
```

uvm_comparer 类是一个专用于比较数据的类，定义位于 uvm_comparer.svh 文件中，它的主体是 compare_field，compare_field_int，compare_field_real，compare_object，compare_string，分别用于比较整数，整数，实数，类，字符串。

先看 compare_field_int：

```
文件：src/base/uvm_comparer.svh
类：uvm_comparer
函数/任务：compare_field_int

202   virtual function bit compare_field_int (string name,
203                                           logic[63:0] lhs,
204                                           logic[63:0] rhs,
205                                           int size,
206                                           uvm_radix_enum radix=UVM_NORADIX);
207     logic [63:0] mask;
208     string msg;
209
210     mask = -1;
211     mask >>= (64-size);
212     if((lhs & mask) !== (rhs & mask)) begin
213       uvm_object::__m_uvm_status_container.scope.set_arg(name);
214       case (radix)
215         UVM_BIN: begin
216                    $swrite(msg, "lhs = 'b%0b : rhs = 'b%0b",
217                                 lhs&mask, rhs&mask);
218                  end
219         UVM_OCT: begin
220                    $swrite(msg, "lhs = 'o%0o : rhs = 'o%0o",
221                                 lhs&mask, rhs&mask);
222                  end
```

```
223             UVM_DEC: begin
224                     $swrite(msg, "lhs = %0d : rhs = %0d",
225                             lhs&mask, rhs&mask);
226                     end
227             UVM_TIME: begin
228                     $swrite(msg, "lhs = %0t : rhs = %0t",
229                       lhs&mask, rhs&mask);
230             end
231             UVM_STRING: begin
232                     $swrite(msg, "lhs = %0s : rhs = %0s",
233                             lhs&mask, rhs&mask);
234                     end
235             UVM_ENUM: begin
236                     //Printed as decimal, user should cuse compare string for enum val
237                     $swrite(msg, "lhs = %0d : rhs = %0d",
238                             lhs&mask, rhs&mask);
239                     end
240             default: begin
241                     $swrite(msg, "lhs = 'h%0x : rhs = 'h%0x",
242                             lhs&mask, rhs&mask);
243                     end
244         endcase
245         print_msg(msg);
246         return 0;
247       end
248       return 1;
249     endfunction
```

主要的比较工作是 212 行的 if 的条件语句。这里用到了 mask，其值是根据 size 的值来判断的，假如 size=5，那么可以计算出 mask 的值为 64’b11111。212 行根据这个 mask 的值，从而会只比较要比较的两个数的低 5 位，而把高位忽略。这输入的两个数据不一样时，214 到 244 行用于根据输入的参数决定以什么进制来输出这两个数字，可以有二进制，十进制，十六进制甚至还可以是字符串形式。213 行把一个字符串，也就是比较的字段的名字放在了__m_uvm_status_container 的 scope 中，这句话其实是为了配合 245 行的。对于 print_msg 函数：

```
文件：src/base/uvm_comparer.svh
类：uvm_comparer
函数/任务：print_msg

342     function void print_msg (string msg);
343       result++;
344       if(result <= show_max) begin
345           msg = {"Miscompare for ", uvm_object::__m_uvm_status_container.scope.get(), ": ",
msg};
346           uvm_report_info("MISCMP", msg, UVM_LOW);
347       end
348       miscompares = { miscompares, uvm_object::__m_uvm_status_container.scope.get(), ": ",
msg, "\n" };
349     endfunction
```

函数的参数中仅仅包含如下的一个结果：

```
lhs=xxxx : rhs=xxxx
```

但是没有并没有说明是哪个字段的，于是真正的打印之前，需要把字段的名字也打印出来。345 行就是从__m_uvm_status_container 的 scope 得到刚刚放进去的字符串，这样新的打印信息将会是这种形式：

```
Miscompare for num : lhs=xxxx : rhs=xxxx
```

至于具体的 set_arg 与 get 函数，这里不再详细介绍，读者有兴趣可以自己去研究下。

compare_field 与 compare_field_int 相似，只是后者的数据的位宽不大于 64 位，而前者最大可以达到 4096 位。

compare_field_real，compare_string 的形式都与 compare_field_int 相似，不重复阐述。compare_object 与上述实现不同：

```
文件：src/base/uvm_comparer.svh
类：uvm_comparer
函数/任务：compare_object

285    virtual function bit compare_object (string name,
286                                         uvm_object lhs,
287                                         uvm_object rhs);
288      if (rhs == lhs)
289        return 1;
290
291      if (policy == UVM_REFERENCE && lhs != rhs) begin
292        uvm_object::__m_uvm_status_container.scope.set_arg(name);
293        print_msg_object(lhs, rhs);
294        return 0;
295      end
296
297      if (rhs == null || lhs == null) begin
298        uvm_object::__m_uvm_status_container.scope.set_arg(name);
299        print_msg_object(lhs, rhs);
300        return 0;   //miscompare
301      end
302
303      uvm_object::__m_uvm_status_container.scope.down(name);
304      compare_object = lhs.compare(rhs, this);
305      uvm_object::__m_uvm_status_container.scope.up();
306
307    endfunction
```

291 行出现了 policy：

```
文件：src/base/uvm_comparer.svh
类：uvm_comparer
```

```
40    uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY;
```

这是一个 uvm_recursion_policy_enum 类型的变量：

```
文件：src/base/uvm_object_globals.svh
类：无

131 // Enum: uvm_recursion_policy_enum
132 //
133 // Specifies the policy for copying objects.
134 //
135 // UVM_DEEP      - Objects are deep copied (object must implement copy method)
136 // UVM_SHALLOW   - Objects are shallow copied using default SV copy.
137 // UVM_REFERENCE - Only object handles are copied.
138
139 typedef enum {
140   UVM_DEFAULT_POLICY = 0,
141   UVM_DEEP           = 'h400,
142   UVM_SHALLOW        = 'h800,
143   UVM_REFERENCE      = 'h1000
144 } uvm_recursion_policy_enum;
```

它决定了在比较两个类的实例的时候，是仅仅比较句柄还是要具体的比较类中每个字段是否一致。

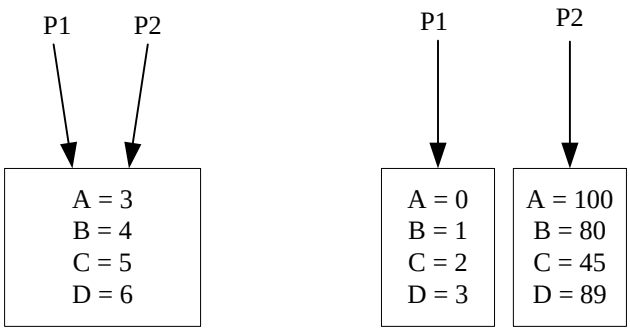


图 14-1 句柄与实例

如上图所示中，ABCD 都是成员变量。对于上面的左图，如果仅仅比较句柄的话，得出的结果是一样的，因为 P1 和 P2 两个句柄指向的是同一个实例，如果比较的是类中每个字段是否一致的话，那么由于这两个本来就是同一实例，其各个字段当然一样了，所以比较的结果也是一致的。

对于上面的右图，如果仅仅比较句柄的话，比较的结果一定是不一样的，因为 P1 和 P2 两个句柄指向两个不同的实例。如果是逐个字段比较的话，则得到的结果可能是一样的（在上例中是不一样的）。

这两种比较哪种比较符合实际呢？假如 scoreboard 一方面从 model 收到了一个

transaction，另一方面也从 monitor 收到了一个 transaction，这两 个 transaction 是两个不同的实例。所以如果采用句柄比较的话，那么比较结果一定是不一样的，这是不符合我们的预期的。所以一般的还是采用逐字段比较。

回到 compare_object，291 行采用的就是句柄比较，而默认情况下，由于 policy 的值是 UVM_DEFAULT_POLICY，所以不会进行这种比较。如果不是这种比较，那么接下来就要进行逐字段比较了，在这种比较中，一定要确保两个实例是存在的，不能是 null。

304 行调用这个 object 的 compare 函数，这相当于是一种递归调用。这里 303 行与 304 行也是为了打印信息做准备的，不详细阐述。

print，pack，unpack，record 等操作均与 compare 操作类似，都是有一个全局的专门负责这种操作的类，而__m_uvm_field_automation 会调用这个类的相关函数来完成这些操作，这些操作实现起来虽然繁琐，但是比较简单，其分析方式与 compare 操作类似，因此这里不多做解释。

14.2.3. set_*_local 操作

除上了节所说的几种常用的 field_automation 操作外，UVM 还提供 set_*_local 操作。考虑如下一种情况：

```
class my_class extends uvm_object;
  local int a;
  local int b;
  local int c;
  …
endclass
```

对于编程来说，一种极端的情况是所有的变量都要设置成 local 形式的，这样才能达到封装的目的。但是有时候难免需要对某些值需要设置，此时就有了相应的 set 函数：

```
function set_a(int value);
  a = value;
endfunction
function set_b(int value);
function set_c(int value);
```

多一个变量就要多写一个 set 函数。UVM 通过 field_automation 机制，实现了无论变量多少（前提是这些变量都是 int 型的），只用一个函数就可以了。

```
set_int_local(“a”, 3);
set_int_local(“b”, 4);
```

这种功能对于特别喜欢使用 local 来定义成员变量的人特别有用。

以 set_int_local 为例，看一下其是如何实现的：

```
文件：src/base/uvm_object.svh
类：uvm_object
函数/任务：set_int_local

 923 function void  uvm_object::set_int_local (string         field_name,
 924                                            uvm_bitstream_t value,
 925                                            bit             recurse=1);
 926   if(__m_uvm_status_container.cycle_check.exists(this)) return;
 927   __m_uvm_status_container.cycle_check[this] = 1;
 928
 929   this.__m_uvm_status_container.status = 0;
 930   this.__m_uvm_status_container.bitstream = value;
 931
 932   __m_uvm_field_automation(null, UVM_SETINT, field_name);
 933
 934   if(__m_uvm_status_container.warning && !this.__m_uvm_status_container.status) begin
 935     uvm_report_error("NOMTC", $sformatf("did not find a match for field %s", field_nam
e),UVM_NONE);
 936   end
 937   __m_uvm_status_container.cycle_check.delete(this);
 938
 939 endfunction
```

930 行把要设置的值放入 bitstream 变量中，932 行调用__m_uvm_fieldautomation。

uvm_field_int 中与其相关的宏为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 542 `define uvm_field_int(ARG,FLAG) \
 543   begin \
 544     case (what__) \
           …
 580       UVM_SETINT: \
 581         begin \
 582           bit matched; \
 583           __m_uvm_status_container.scope.set_arg(`"ARG`"); \
 584           matched = uvm_is_match(str__, __m_uvm_status_container.scope.get()); \
 585           if(matched) begin \
 586             if((FLAG)&UVM_READONLY) begin \
 587               uvm_report_warning("RDONLY", $sformatf("Readonly argument match %s i
s ignored",  \
 588                                  __m_uvm_status_container.get_full_scope_arg()), UVM_NONE); \
 589             end \
 590             else begin \
 591               if (__m_uvm_status_container.print_matches) \
 592                 uvm_report_info("STRMTC", {"set_int()", ": Matched string ", str__, "
```

```
to field ", __m_uvm_status_container.get_full_scope_arg()}, UVM_LOW); \
593                     ARG = uvm_object::__m_uvm_status_container.bitstream; \
594                     uvm_object::__m_uvm_status_container.status = 1; \
595                   end \
596                 end \
597                 __m_uvm_status_container.scope.unset_arg(`"ARG`"); \
598               end \
599         endcase \
600     end
```

584 行判断 set_int_local 输入的字符串是不是此类中定义的字段。例如，当我们输入”a”时，那么只有在 uvm_field_int(a, …)才能匹配到，但是在 uvm_field_int(b, …)中则匹配不到。如果匹配到了，那么 586 行会判断一下这个字段是不是只读的，如果是只读的是不能设置的。593 行从 bitstream 中取出刚刚放入的值，并且赋值给要设置的变量。这里之所以要经历一个把设置值放入 bitstream 再取出的过程，是因为如果不这样做，那么就需要把这个值通过__m_uvm_fieldautomation 来传递，这样此函数就多了一个参数。对于 set_int_local 来说，此参数是 int 型的，但是对于 set_string_local 来说，此参数是 string 类型的。这样会带来函数的参数过多的问题。

14.2.4. 自动 get_config 功能的实现

前面讲述过，应用了 field_automation 机制之后，那么在应用 config 机制的时候，可以省去 get。这一功能是通过 uvm_component 的 build_phase 实现的：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：build_phase

2271 function void uvm_component::build_phase(uvm_phase phase);
2272     m_build_done = 1;
2273     build();
2274 endfunction
```

build_phase 会调用 build：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：build

2278 function void uvm_component::build();
2279     m_build_done = 1;
2280     apply_config_settings(print_config_matches);
2281     if(m_phasing_active == 0) begin
2282         uvm_report_warning("UVM_DEPRECATED", "build()/build_phase() has been called explicitly, outside of the phasing system. This usage of build is deprecated and may lead to u
```

```
nexpected behavior.");
2283     end
2284  endfunction
```

而 build 又最终会调用 apply_config_settings：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：apply_config_settings

2972 function void uvm_component::apply_config_settings (bit verbose=0);
2973
2974    uvm_resource_pool rp = uvm_resource_pool::get();
2975    uvm_queue#(uvm_resource_base) rq;
2976    uvm_resource_base r;
2977    string name;
2978    string search_name;
2979    int unsigned i;
2980    int unsigned j;
2981
2982    __m_uvm_field_automation (null, UVM_CHECK_FIELDS, "");
2983
2984    if(verbose)
2985       $display("applying configuration settings for %s", get_full_name());
2986
2987    rq = rp.lookup_scope(get_full_name());
2988    rp.sort_by_precedence(rq);
2989
2990    // rq is in precedence order now, so we have to go through in reverse
2991    // order to do the settings.
2992    for(int i=rq.size()-1; i>=0; --i) begin
2993
2994       r = rq.get(i);
2995       name = r.get_name();
2996
2997       // does name have brackets [] in it?
2998       for(j = 0; j < name.len(); j++)
2999          if(name[j] == "[" || name[j] == ".")
3000             break;
3001
3002       // If it does have brackets then we'll use the name
3003       // up to the brackets to search __m_uvm_status_container.field_array
3004       if(j < name.len())
3005          search_name = name.substr(0, j-1);
3006       else
3007          search_name = name;
3008
3009       if(!uvm_resource_pool::m_has_wildcard_names &&
3010          !__m_uvm_status_container.field_array.exists(search_name))
3011          continue;
3012
3013       if(verbose)
```

```
3014          $display("applying %s [%s] in %s", name, __m_uvm_status_container.field_array[sear
ch_name],
3015                                                  get_full_name());
3016
3017        begin
3018        uvm_resource#(uvm_bitstream_t) rbs;
3019        if($cast(rbs, r))
3020          set_int_local(name, rbs.read(this));
3021        else begin
3022          uvm_resource#(int) ri;
3023          if($cast(ri, r))
3024            set_int_local(name, ri.read(this));
3025          else begin
3026            uvm_resource#(int unsigned) riu;
3027            if($cast(riu, r))
3028              set_int_local(name, riu.read(this));
3029            else begin
3030              uvm_resource#(string) rs;
3031              if($cast(rs, r))
3032                set_string_local(name, rs.read(this));
3033              else begin
3034                uvm_resource#(uvm_object) ro;
3035                if($cast(ro, r))
3036                  set_object_local(name, ro.read(this), 0);
3037              end
3038            end
3039          end
3040        end
3041        end
3042
3043      end
3044
3045      __m_uvm_status_container.field_array.delete();
3046
3047  endfunction
```

2982 行调用 __m_uvm_field_automation 函数，传入的参数是 UVM_CHECK_FIELDS。uvm_field_int 宏中与此相关的代码为：

```
文件：src/macros/uvm_object_defines.svh
类：无

 542 `define uvm_field_int(ARG,FLAG) \
 543   begin \
 544     case (what__) \
 545       UVM_CHECK_FIELDS: \
 546         begin \
 547           __m_uvm_status_container.do_field_check(`"ARG`", this); \
 548         end \
 549       UVM_COPY: \
           …
 599     endcase \
```

```
600     end
```

这里只是纯粹的调用了 uvm_status_container 的 do_field_check 函数：

```
文件：src/base/uvm_misc.svh
类：uvm_status_container
函数/任务：do_field_check

216     function void do_field_check(string field, uvm_object obj);
217       `ifdef UVM_ENABLE_FIELD_CHECKS
218         if (field_array.exists(field))
219           uvm_report_error("MLTFLD", $sformatf("Field %s is defined multiple times in type
'%s'",
220                 field, obj.get_type_name()), UVM_NONE);
221       `endif
222         field_array[field] = 1;
223     endfunction
```

这个函数也很简单只是查看一下 field_array 数组中是否已经有了要检查的字段。有了就报错，否则会直接把此字段放入 field_array 中。对于

```
`uvm_field_int(a, …)
…
`uvm_field_int(z, …)
```

经过 2982 行调用__m_uvm_field_automation 后，field_array 中将会包含 a…z 所有这些字段。2987 行用到了 resource 机制（其实也就是 config 机制）的一些内容，后面章节会对其详细介绍。2987 行的意思是得到所有通过 uvm_config_db::set 功能设置到此 component 的记录，2988 行把这些记录排序。这种排序是按照层次结构来排序的，高层的放在最顶端，而底层设置放在底端。如假设在 base_test 中和 env 中同时对此 component 的某个字段进行了设置，由于 env 是在 base_test 中实例化的，其层次低于 base_test，所以 base_test 中的那条 set 记录将会放在最顶端，而 env 中的则放在这条记录的后面。由于采用了这种排序，所以 2992 行将会从最底层的记录开始遍历。config 机制的原则就是高层的优先级高于低层的优先级，所以从最底层记录开始遍历，并进行设置。如果有高层的记录，那么将会把底层的记录覆盖。

2998 到 3011 行主要是为了处理字段的名字中有[]的情况。如对于如下的一个定义：

```
int a[3];
```

那么在 set 的时候，指定名字的时候要指定成”a[n]”，其中 n 为 0 到 2 的数字。3017 到 3041 行通过 set_*_local 来进行具体的设置。由于不知道这条 set 记录的类型，所以这里通过不停的 cast 方式来判断到底是哪种类型的。3045 行把 field_array 数组清空，以便下一个 uvm_component 的 apply_config_settings 使用。

14.2.5. 小结

在应用 field_automation 机制时，有很多中间的临时数据需要交换。对于这些数据，可以使用全局变量来实现，但是全局变量的坏处前面已经多次提及。在不得不用的情况下，可以对这些全局变量进行一些包装，让使用这些全局变量不那么容易，这可以大大减少由于误操作导致全局变量的值被改变的情况。

uvm_status_container 就是这么一个类，它是一个全局变量，它封装了可能用到的很多全局变量。这些全局变量要使用的时候要加上长长的前缀：

uvm_object::__m_uvm_status_container.field_array

（后面写一下关于联合数组索引的）

15. sequence 机制源代码分析

本章讲述 sequence 机制。sequence 机制是 UVM 的核心机制。其实现相对复杂，不过比起它的前辈 OVM 来说，这里的 sequence 机制比 OVM 的 sequence 机制要简单些，但是实现的功能并没有打折扣。第一节从 uvm_do 宏开始谈起，仔细分析宏的执行过程，第二节讲述 sequence 机制发送 sequence_item 的一些关键步骤。第三节讲述验证平台中常用的 default sequence 的启动流程及 p_sequencer 的由来。第四节讲述带有 response 的 sequence。

15.1. uvm_do 系列宏

uvm_do 系列宏是 UVM 中 sequence 机制的具体实现，通过这些宏，实现了 transaction 的产生，与 driver 的交互。

15.1.1. 宏的展开

uvm_do 系列宏主要有以下几个：

文件：src/base/uvm_sequence_defines.svh

```
类：无

 90 `define uvm_do(SEQ_OR_ITEM) \
 91   `uvm_do_on_pri_with(SEQ_OR_ITEM, m_sequencer, -1, {})

101 `define uvm_do_pri(SEQ_OR_ITEM, PRIORITY) \
102   `uvm_do_on_pri_with(SEQ_OR_ITEM, m_sequencer, PRIORITY, {})

113 `define uvm_do_with(SEQ_OR_ITEM, CONSTRAINTS) \
114   `uvm_do_on_pri_with(SEQ_OR_ITEM, m_sequencer, -1, CONSTRAINTS)

125 `define uvm_do_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS) \
126   `uvm_do_on_pri_with(SEQ_OR_ITEM, m_sequencer, PRIORITY, CONSTRAINTS)

162 `define uvm_do_on(SEQ_OR_ITEM, SEQR) \
163   `uvm_do_on_pri_with(SEQ_OR_ITEM, SEQR, -1, {})

174 `define uvm_do_on_pri(SEQ_OR_ITEM, SEQR, PRIORITY) \
175   `uvm_do_on_pri_with(SEQ_OR_ITEM, SEQR, PRIORITY, {})

187 `define uvm_do_on_with(SEQ_OR_ITEM, SEQR, CONSTRAINTS) \
188   `uvm_do_on_pri_with(SEQ_OR_ITEM, SEQR, -1, CONSTRAINTS)
```

可以看出，这些所有的宏最终都是调用的 uvm_do_on_pri_with 宏：

```
文件：src/base/uvm_sequence_defines.svh
类：无

199 `define uvm_do_on_pri_with(SEQ_OR_ITEM, SEQR, PRIORITY, CONSTRAINTS) \
200   begin \
201   uvm_sequence_base __seq; \
202   `uvm_create_on(SEQ_OR_ITEM, SEQR) \
203   if (!$cast(__seq,SEQ_OR_ITEM)) start_item(SEQ_OR_ITEM, PRIORITY);\
204   if (!SEQ_OR_ITEM.randomize() with CONSTRAINTS ) begin \
205     `uvm_warning("RNDFLD", "Randomization failed in uvm_do_with action") \
206   end\
207   if (!$cast(__seq,SEQ_OR_ITEM)) finish_item(SEQ_OR_ITEM, PRIORITY); \
208   else __seq.start(SEQR, this, PRIORITY, 0); \
209   end
```

这个宏有四个参数，第一个是要发送的 sequence 或者是 sequence_item，第二个参数是 sequencer，也就是产生出来的 sequence_item 要交给哪个 sequencer，第三个参数是优先级，第四个参数是 sequence_item 或者 sequence 在随机化的约束条件。

由第一个参数 SEQ_OR_ITEM 的字面意思，我们可以理解这是一个 sequence 或者一个 sequence_item。这两种情况下，宏的行为是不一样的。下面将分这两种情况分别介绍。

15.1.2. uvm_create_on 宏

202 行调用了 uvm_create_on 宏：

```
文件：src/base/uvm_sequence_defines.svh
类：无

146 `define uvm_create_on(SEQ_OR_ITEM, SEQR) \
147   begin \
148   uvm_object_wrapper w_; \
149   w_ = SEQ_OR_ITEM.get_type(); \
150   $cast(SEQ_OR_ITEM , create_item(w_, SEQR, `"SEQ_OR_ITEM`"));\
151   end
```

149 行调用 get_type 函数。这点可能让人费解。因为此时 SEQ_OR_ITEM 依然是一个 null 的指针，做为一个 null 的指针来说，又怎么可以调用函数呢？因为很简单，get_type 是一个静态函数。它是在 uvm_object 中定义的。通常意义上，我们认为静态函数应该这样调用：

```
uvm_object::get_type();
```

但是事实上，也可以通过具体的变量的名字来调用：

```
uvm_object a;
a.get_type();
```

当我们使用 uvm_object_utils 宏时，会把 uvm_object 中定义的 get_type 函数重载：

```
文件：src/macros/uvm_object_defines.svh
类：无

 407 `define m_uvm_object_registry_internal(T,S) \
 408     typedef uvm_object_registry#(T,`"S`") type_id; \
 409     static function type_id get_type(); \
 410       return type_id::get(); \
 411     endfunction \
 412     virtual function uvm_object_wrapper get_object_type(); \
 413       return type_id::get(); \
 414     endfunction
```

w_的值将会是使用 uvm_object_utils 宏注册在 factory 中的静态变量值。如假设使用了此宏注册了 my_sequence，那么：

```
w_ = uvm_object_registry#(my_sequence, “my_sequence”)::me
```

前面已经说过，通过 me 可以唯一的确定一个类型。因此，当 SEQ_OR_ITEM 是一个 sequence 时，那么 w_就是代表一个 sequence；如果是一个 sequence_item 时，w_就代表一个 sequence_item。

150 行调用了 create_item 函数，它是 uvm_sequence_base 中定义的函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：create_item

 713    protected function uvm_sequence_item create_item(uvm_object_wrapper type_var,
 714                                                     uvm_sequencer_base l_sequencer,
 string name);
 715
 716      uvm_factory f_ = uvm_factory::get();
 717      $cast(create_item,  f_.create_object_by_type( type_var, this.get_full_name(), name ));
 718
 719      create_item.set_use_sequence_info(1);
 720      create_item.set_parent_sequence(this);
 721      create_item.set_sequencer(l_sequencer);
 722      create_item.set_depth(get_depth() + 1);
 723      create_item.reseed();
 724
 725    endfunction
```

函数比较简单，就是通过 factory 机制来实例化一个 SEQ_OR_ITEM，并且设置 parent_sequence，sequencer，depth 等信息。

719 行调用了 set_use_sequence_info 函数，它的定义位于 uvm_sequence_item 中：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：set_use_sequence_info

109    function void set_use_sequence_info(bit value);
110      m_use_sequence_info = value;
111    endfunction
```

函数相当简单，只是把输入的值赋值给 m_use_sequence_info 变量。这个变量的作用就是控制着 sequence 的信息，如 sequencer，parent_sequence，sequence_id 等信息在 print，copy，record 等时的行为。假如此位为 1，那么这些信息将会 print 和 copy，否则将不会使用。

这里比较让人疑惑的是由于 SEQ_OR_ITEM 可能是一个 sequence_item，也可能是一个 sequence，那么假如是一个 sequence 的话，那么还如何调用属于 uvm_sequence_item 的函数呢？uvm_sequence 的继承关系如下：

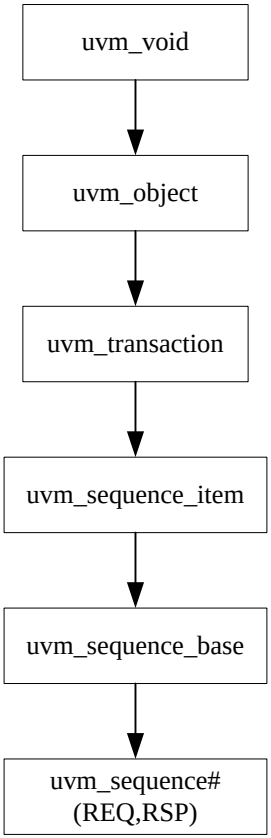


图 15-1 uvm_sequence 的继承关系

从图中可以看出，uvm_sequence 其实是派生自 uvm_sequence_item 的，所以 uvm_sequence_item 的函数同样也是 uvm_sequence 的成员函数。

720 行调用了 set_parent_sequence 函数，它也是由 uvm_sequence_item 定义的：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：set_parent_sequence

169    function void set_parent_sequence(uvm_sequence_base parent);
170      m_parent_sequence = parent;
171    endfunction
```

m_parent_sequence 是一个属于 uvm_sequence_item 的，uvm_sequence_base 类型的成员变量。它记录了 sequenc 之间的从属关系。

```
class my_seq extends uvm_sequence#(my_transaction);
   task body();
      my_transaction tr;
```

```
        `uvm_do(tr)
    endtask
    …
endclass
class new_seq extends uvm_sequence#(my_transaction);
    task body();
        new_seq ns;
        `uvm_do(ns)
    endtask
endclass
```

在上面的例子中，tr 的 parent_sequence 就是 my_seq，而 ns 的 parent_sequence 就是 new_seq。由于有这种层次关系，那么最顶层的 sequence 的 parent_sequence 将会是什么呢？很简单，其值将会是默认值 null。

721 行调用了 set_sequencer 函数，同样是 uvm_sequence_item 的成员函数：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：set _sequencer

149    virtual function void set_sequencer(uvm_sequencer_base sequencer);
150        m_sequencer = sequencer;
151        m_set_p_sequencer();
152    endfunction
```

简单的给 m_sequencer 赋值。这里比较让人疑惑的是 m_set_p_sequencer 函数。此函数的定义为：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：m_set_p _sequencer

265    virtual function void m_set_p_sequencer();
266        return;
267    endfunction
```

这是一个空函数，派生类将会重载这个函数。这里牵扯到 p_sequencer 和 m_sequencer 的区别问题，后面章节将会专门的讲述这个。

722 行将会先调用 get_depth 函数，它也是 uvm_sequence_item 的成员函数：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：get_depth

202    function int get_depth();
203
204        // If depth has been set or calculated, then use that
205        if (m_depth != -1) begin
206            return (m_depth);
```

```
207       end
208
209       // Calculate the depth, store it, and return the value
210       if (m_parent_sequence == null) begin
211         m_depth = 1;
212       end else begin
213         m_depth = m_parent_sequence.get_depth() + 1;
214       end
215
216       return (m_depth);
217     endfunction
```

这个函数可以结合上面的 new_seq 和 my_seq 的例子来分析。假设 new_seq 是最顶层的 sequence（如直接把其做为某 sequencer 的 main_phase 的 default_sequence），那么根据 210 行，其 m_depth 将会是 1；根据 213 行，my_seq 的 m_depth 将会是 2；tr 的 m_depth 将会是 3。

722 行通过 get_depth 与 set_depth 的值来设置 m_depth 的值。如在 my_seq 的 uvm_do(tr)中，get_depth 的值将会得到 my_seq 的 m_depth 值，即 2。通过 set_depth 值，把 2+1=3 设置为 tr 的 m_depth 值。

723 将会调用 reseed 函数。这里就是重新设置系统的随机数种子，不多做介绍。

回到 create_item 中，假设 uvm_create_on 宏 150 行中传入的 w_代表的是一个 sequence，那么最终 create_item 函数返回的是一个 uvm_sequence_base 类型的指针，否则将会是一个 uvm_sequence_item 类型的指针。

uvm_create_on 宏 150 行通过 cast 把 create_item 返回的指针赋值给 SEQ_OR_ITEM，从而 SEQ_OR_ITEM 要么就是一个 uvm_sequence_base 类型的指针，要么是一个 uvm_sequ ence_item 类型的指针。

15.1.3. SEQ_OR_ITEM 是一个 sequence_item

uvm_do_on_pri_with 宏的 203 行会判断经过 uvm_create_on 实例化后的 SEQ_OR_ITEM 是否是一个 sequence。

由上节的 sequence 继承关系图可以看出，假如 SEQ_OR_ITEM 是一个 sequence_item 的话，那么 203 行 cast 的返回值将会是 0，因为一个 item 不是 uvm_sequence_base 类型的。从而会调用 start_item 函数。本节将不会展开 start_item，后面会仔细分析。

204 到 206 行将会执行 SEQ_OR_ITEM 的随机化。

207 行将会再次判断 SEQ_OR_ITEM 是不是一个 sequence_item，是的话将会执

行 finish_item。与 start_item 一样，将会在后面专门分析这两个函数。

15.1.4. SEQ_OR_ITEM 是一个 sequence

当 SEQ_OR_ITEM 是一个 sequence 时，203 行和 207 行的 start_item 及 finish_item 将不会执行。这里将会调用的是 uvm_sequence_base 的 start 函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：start

 238    virtual task start (uvm_sequencer_base sequencer,
 239                        uvm_sequence_base parent_sequence = null,
 240                        int this_priority = -1,
 241                        bit call_pre_post = 1);
 242
 243      if (parent_sequence != null) begin
 244          set_parent_sequence(parent_sequence);
 245          set_use_sequence_info(1);
 246          if (sequencer == null) sequencer = parent_sequence.get_sequencer();
 247          reseed();
 248      end
 249      set_sequencer(sequencer);
 250
 251      if (!(m_sequence_state != CREATED ||
 252            m_sequence_state != STOPPED ||
 253            m_sequence_state != FINISHED)) begin
 254        uvm_report_fatal("SEQ_NOT_DONE",
 255            {"Sequence ", get_full_name(), " already started"},UVM_NONE);
 256      end
 257
 258      if (this_priority < -1) begin
 259        uvm_report_fatal("SEQPRI", $psprintf("Sequence %s start has illegal priority: %0d",
 260                                             get_full_name(),
 261                                             this_priority), UVM_NONE);
 262      end
 263      if (this_priority < 0) begin
 264          if (parent_sequence == null) this_priority = 100;
 265          else this_priority = parent_sequence.get_priority();
 266      end
 267
 268      // Check that the response queue is empty from earlier runs
 269      clear_response_queue();
 270
```

243 到 249 行设置 parent_sequence，sequencer 等信息。

251 行出现了 m_sequence_state 变量，它是 uvm_sequence_state 类型的：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 136    protected uvm_sequence_state m_sequence_state;
```

uvm_sequence_state 的定义位于 uvm_object_globals.svh 文件中：

```
文件：src/base/uvm_object_globals.svh
类：无

401 typedef enum
402 {
403    CREATED     = 1,
404    PRE_START  = 2,
405    PRE_BODY   = 4,
406    BODY         = 8,
407    POST_BODY  = 16,
408    POST_START= 32,
409    ENDED        = 64,
410    STOPPED     = 128,
411    FINISHED   = 256
412 } uvm_sequence_state;
```

CREATED 表示此 sequence 已经实例化，接下来从 PRE_START 一直到 POST_START 其实是代表 5 个不同的 task 或者 function：pre_start，pre_body，body，post_body，post_start 等。STOPPED 则是表示一种不正常的中止，FINISHED 表示正常的结束。

当实例化之后，m_sequence_state 默认将会是 CREATED，254 行的语句不会执行。

258 到 266 行设置 priority 属性。假设有多个 sequence 同时启动，这几个 sequence 共用一个 sequencer，那么 sequencer 将会根据 priority 的特性选择把哪个 sequence_item 送给 driver。如果不做设置，那么 14.4.2 中的 new_seq，my_seq 和 tr 的 priority 将全部为 100。如果是想要自己设置，那么设置的 priority 属性大于 100 的话会优先发送，小于 100 的话则会最后发送。

269 行调用了 clear_response_queue 函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：clear_response_queue

 968    virtual function void clear_response_queue();
 969      response_queue.delete();
 970    endfunction
```

它做的事情就是把 response_queue 清空：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 146    protected uvm_sequence_item response_queue[$];
```

这是一个队列，存储的内容是 uvm_sequene_item 类型。作用是存放 driver 发送给 sequence 的 response。因为这些 response 可能并不会马上被 sequence 取走，所以需要有一个缓存来把这些 response 暂存下来。这将在 15.4 节介绍 sequence 的 response 时详细介绍。

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：start

 271      if (parent_sequence != null)
 272        m_parent_sequence   = parent_sequence;
 273      m_sequencer             = sequencer;
 274      m_priority              = this_priority;
 275
 276      m_set_p_sequencer();
 277
 278      if (m_sequencer != null) begin
 279          if (m_parent_sequence == null) begin
 280            m_tr_handle = m_sequencer.begin_tr(this, get_name());
 281          end else begin
 282            m_tr_handle = m_sequencer.begin_child_tr(this, m_parent_sequence.m_tr_handle,
 283                                                     get_root_sequence_name());
 284          end
 285      end
 286
```

271 到 274 行 parent_sequence，sequencer，priority 等属性。276 行又调用 m_set_p_sequencer 函数，在 15.1.2 中介绍 create_item 时曾经见过这个函数，后面会专门讲述它。

278 行判断 m_sequencer 的值是否为 null。在正常的情况下，这个值一般不是 null，因为如果它为 null，那么产生的 sequence_item 通过谁来转交给 driver 呢？不过这个值确实有为 null 的情况，在 register model 的相关 sequence 启动时会遇到这种情况，后面将会仔细说明。

如果这个 sequence 是最顶层的 sequence（其 parent_sequence 为 null），那么 280 行调用 begin_tr 函数，否则将会执行 282 行调用 begin_child_tr 函数。这两个函数是 uvm_sequencer 的成员变量，但是它们并不是在 uvm_sequencer 中定义的，而是在 uvm_component 中定义的。uvm_sequencer 的继承关系如图：

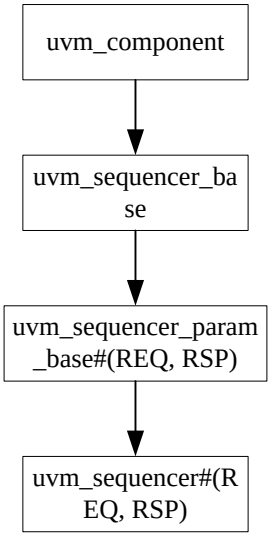


图 15-2 uvm_sequencer 的继承关系

begin_tr 函数定义如下：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：begin_tr

2575 function integer uvm_component::begin_tr (uvm_transaction tr,
2576                                            string stream_name ="main",
2577                                            string label="",
2578                                            string desc="",
2579                                            time begin_time=0,
2580                                            integer parent_handle=0);
2581   return m_begin_tr(tr, parent_handle, (parent_handle!=0), stream_name, label, desc, begin_time);
2582 endfunction
```

它将会直接调用 m_begin_tr 函数。begin_child_tr 函数如下：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：begin_child_tr

2587 function integer uvm_component::begin_child_tr (uvm_transaction tr,
2588                                            integer parent_handle=0,
2589                                            string stream_name="main",
2590                                            string label="",
2591                                            string desc="",
2592                                            time begin_time=0);
2593   return m_begin_tr(tr, parent_handle, 1, stream_name, label, desc, begin_time);
2594 endfunction
```

它也会直接调用 m_begin_tr 函数，只是传入的参数不同。m_begin_tr 是 uvm_component 的成员函数，其主要作用是在系统中记录一些信息，如开始时间，并触发事件，标志着一个 transaction 开始了。

无论是 280 行还是 282 行，其结果将是返回一个句柄，此句柄代表这个唯一的 sequence 已经在系统中记录，开始启动。

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：start

 287       // Ensure that the sequence_id is intialized in case this sequence has been stopped pre
viously
 288       set_sequence_id(-1);
 289       // Remove all sqr_seq_ids
 290       m_sqr_seq_ids.delete();
 291
 292       // Register the sequence with the sequencer if defined.
 293       if (m_sequencer != null) begin
 294          void'(m_sequencer.m_register_sequence(this));
 295       end
 296
```

288 行调用 set_sequence_id：

```
文件：src/seq/uvm_sequence_item.svh
类：uvm_sequence_item
函数/任务：set_sequence_id

 66    function void set_sequence_id(int id);
 67       m_sequence_id = id;
 68    endfunction
```

把此 sequence 的 m_sequence_id 的值设置为-1。

290 行把 m_seq_sqr_ids 中的记录全部删除：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 144    protected int m_sqr_seq_ids[int];
```

这是一个联合数组，其内容和索引都是 int 类型，它用于记录 sequence 和 sequencer 之间的关系。

294 行调用 sequencer 的 m_register_sequence 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_register_sequencer
```

```
559 function int uvm_sequencer_base::m_register_sequence(uvm_sequence_base sequence_ptr);
560
561     if (sequence_ptr.m_get_sqr_sequence_id(m_sequencer_id, 1) > 0)
562         return sequence_ptr.get_sequence_id();
563
564     sequence_ptr.m_set_sqr_sequence_id(m_sequencer_id, g_sequence_id++);
565     reg_sequences[sequence_ptr.get_sequence_id()] = sequence_ptr;
566     return sequence_ptr.get_sequence_id();
567 endfunction
```

这里出现了 m_sequencer_id，它是 uvm_sequencer_base 的一个成员变量：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

 47     protected int                       m_sequencer_id;
```

另外，uvm_sequencer_base 中与 register_sequence 相关的还有下面几个变量：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

 55     local static int                    g_sequence_id = 1;
 56     local static int                    g_sequencer_id = 1;
```

在 uvm_sequencer_base 的 new 函数中：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：new

451 function uvm_sequencer_base::new (string name, uvm_component parent);
452     super.new(name, parent);
453     m_sequencer_id = g_sequencer_id++;
454     m_lock_arb_size = -1;
455 endfunction
```

g_sequencer_id 是一个静态变量，每实例化一个 sequencer，那么 把 m_sequence_id 的值设置为 g_sequencer_id 的值，同时 g_sequencer_id 的值将会增加。这意味着整个验证平台中每一个 sequencer，无论这个 sequencer 是什么类型的，一定有一个唯一的数字与其对应，这个数字是 1 开始计数的，sequencer 的实例越多，这个数字就越大。

m_register_sequence 的 561 行调用 m_get_sqr_sequence_id 函数，我们先跳过这个函数直接看 564 行的 m_set_sqr_sequence_id 函数，这是 uvm_sequence_base 的一个成员函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：m_set_sqr_sequence_id
```

```
1182    function void m_set_sqr_sequence_id(int sequencer_id, int sequence_id);
1183      m_sqr_seq_ids[sequencer_id] = sequence_id;
1184      set_sequence_id(sequence_id);
1185    endfunction
```

在调用这个函数时传入了这个 sequencer 的 id，同时还传入了 g_sequence_id。而 g_sequence_id 也是一个静态成员变量。每当有一个 sequence 通过 register_sequence 注册到某个 sequencer 时，这个数字将会加 1。对于每一个 sequence，都有唯一的一个 id 与其对应。这个值将会和 sequencer_id 值一起做为一条记录插入到 m_sqr_seq_ids 中，同时把此值将会通过 set_sequence_id 函数作为 m_sequence_id 的值。我们还记得初始的时候是把 m_sequence_id 的值设置为-1，这就意味着还没有把此 sequence 注册到 sequencer 中。因为一旦注册之后，那么 m_sequence_id 的值将会是一个大于 1 的数字。

回过头来看 561 行的 m_get_sqr_sequence_id 函数，这也是 uvm_sequence_base 的一个成员函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：m_get_sqr_sequence_id

1164    function int m_get_sqr_sequence_id(int sequencer_id, bit update_sequence_id);
1165      if (m_sqr_seq_ids.exists(sequencer_id)) begin
1166        if (update_sequence_id == 1) begin
1167          set_sequence_id(m_sqr_seq_ids[sequencer_id]);
1168        end
1169        return m_sqr_seq_ids[sequencer_id];
1170      end
1171
1172      if (update_sequence_id == 1)
1173        set_sequence_id(-1);
1174
1175      return -1;
1176    endfunction
```

如果已经通过 m_register_sequence 把此 sequence 注册到了 sequencer 中，那么 m_sqr_seq_ids 肯定会有一条相应的记录，否则的话将会直接返回-1，表明没有注册过。561 行的判断条件就是为了避免重复注册。

565 行向 reg_sequences 中插入一条记录：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

  46    protected uvm_sequence_base     reg_sequences[int];
```

这是一个联合数组，其索引是 int，而其内容是 uvm_sequence_base。565 行把 sequence_id 及其指针放入 reg_sequences 中。

可见，所谓的注册，一方面是给 sequence 的 m_sequence_id 赋值，给 sequence 的 m_sqr_seq_ids 中插入一条记录，另一方面是在 sequencer 的 reg_sequences 中插入一条记录。通过这种注册，sequence 从 sequencer 中获得 m_sequence_id 的值，有了这个 id，它才能在整个验证平台中通行。而作为颁发 m_sequence_id 的 sequencer，它自然要记录自己颁发了哪个 sequence_id，并把其颁发给了谁。

文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：start

```
297       fork
298         begin
299           m_sequence_process = process::self();
300
301           m_sequence_state = PRE_START;
302           #0;
303           pre_start();
304
305           if (call_pre_post == 1) begin
306             m_sequence_state = PRE_BODY;
307             #0;
308             pre_body();
309           end
310
311           if (parent_sequence != null) begin
312             parent_sequence.pre_do(0);      // task
313             parent_sequence.mid_do(this); // function
314           end
315
316           m_sequence_state = BODY;
317           #0;
318           body();
319
320           m_sequence_state = ENDED;
321           #0;
322
323           if (parent_sequence != null) begin
324             parent_sequence.post_do(this);
325           end
326
327           if (call_pre_post == 1) begin
328             m_sequence_state = POST_BODY;
329             #0;
330             post_body();
331           end
332
333           m_sequence_state = POST_START;
334           #0;
335           post_start();
336
337           m_sequence_state = FINISHED;
```

```
338             #0;
339
340           end
341       join
342
343       if (m_sequencer != null) begin
344          m_sequencer.end_tr(this);
345       end
346
347       // Clean up any sequencer queues after exiting; if we
348       // were forcibly stoped, this step has already taken place
349       if (m_sequence_state != STOPPED) begin
350          if (m_sequencer != null)
351             m_sequencer.m_sequence_exiting(this);
352       end
353
354       #0; // allow stopped and finish waiters to resume
355
356    endtask
```

回到 start 函数的 297 行，从 297 到 341 行比较简单，通过调用 pre_body，body 等，让 sequence 的状态从 PRE_START 一直演变到 FINISHED。这就是为什么当一个 sequence 启动起来后，系统会自动的执行 body 等的原因所在。

344 行的 end_tr 与 begin_tr 相对应。这里也不多做介绍。

349 行判断这个 sequence 是自然执行完毕还是被别的进程杀死根本就没有执行完。如果是自然执行完毕，那么其状态将会是 FINISHED，如果是被杀死，其状态将会是 STOPPED。在 FINISHED 状态下，需要调用 sequencer 的 m_sequence_exiting 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_sequence_exiting

1249 function void uvm_sequencer_base::m_sequence_exiting(uvm_sequence_base sequence_ptr);
1250    remove_sequence_from_queues(sequence_ptr);
1251 endfunction
```

这个函数会直接调用 remove_sequence_from_queues 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：remove_sequence_from_queues

1181 function void uvm_sequencer_base::remove_sequence_from_queues(
1182                                                 uvm_sequence_base sequence_ptr);
1183    int i;
1184    int seq_id;
1185
1186    seq_id = sequence_ptr.m_get_sqr_sequence_id(m_sequencer_id, 0);
```

```
1187
1188    // Remove all queued items for this sequence and any child sequences
1189    i = 0;
1190    do
1191      begin
1192        if (arb_sequence_q.size() > i) begin
1193          if ((arb_sequence_q[i].sequence_id == seq_id) ||
1194              (is_child(sequence_ptr, arb_sequence_q[i].sequence_ptr))) begin
1195            if (sequence_ptr.get_sequence_state() == FINISHED)
1196              `uvm_error("SEQFINERR", $psprintf("Parent sequence '%s' should not finish b
efore all items from itself and items from descendent sequences      are processed.  The item
request from the sequence '%s' is being removed.", sequence_ptr.get_full_name(), arb_sequence_q
[i].sequence_ptr.get_full_nam      e()))
1197            arb_sequence_q.delete(i);
1198            m_update_lists();
1199          end
1200          else begin
1201            i++;
1202          end
1203        end
1204      end
1205    while (i < arb_sequence_q.size());
1206
1207    // remove locks for this sequence, and any child sequences
1208    i = 0;
1209    do
1210      begin
1211        if (lock_list.size() > i) begin
1212          if ((lock_list[i].get_inst_id() == sequence_ptr.get_inst_id()) ||
1213              (is_child(sequence_ptr, lock_list[i]))) begin
1214            if (sequence_ptr.get_sequence_state() == FINISHED)
1215              `uvm_error("SEQFINERR", $psprintf("Parent sequence '%s' should not finish b
efore locks from itself and descedent sequences are removed.  T      he lock held by the child
sequence '%s' is being removed.",sequence_ptr.get_full_name(), lock_list[i].get_full_name()))
1216            lock_list.delete(i);
1217            m_update_lists();
1218          end
1219          else begin
1220            i++;
1221          end
1222        end
1223      end
1224    while (i < lock_list.size());
1225
1226    // Unregister the sequence_id, so that any returning data is dropped
1227    m_unregister_sequence(sequence_ptr.m_get_sqr_sequence_id(m_sequencer_id, 1));
1228  endfunction
```

1186 行得到 sequence 的 id，1190 到 1205 行将会从 arb_sequence_q 中删除关于这条 sequence 的记录。arb_sequence_q 是 sequencer 用于仲裁的一个队列，当有多个 sequence 同时请求发送 sequence_item 时，它通过这个队列来进行仲裁。关于这一点，

后面会仔细介绍。

1209 到 1224 行则用于删除 lock_list 中的记录。由于一个 sequence 可以把一个 sequencer 给 lock 住，由于此 sequence 中已经结束，所以理应把其 lock 信息给删除。

其实无论是仲裁信息，还是 lock 信息，在一个 sequence 自然执行完毕的情况下，lock_list 和 arb_sequence_q 中都不会有此 sequence 的记录。这里做这些其实是为了应对此 sequence 不是正常中止，而是被其它进程杀死的情况。

1227 行调用 m_unregistered_sequences 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_unregistered_sequences

 595  function void uvm_sequencer_base::m_unregister_sequence(int sequence_id);
 596    if (!reg_sequences.exists(sequence_id))
 597      return;
 598    reg_sequences.delete(sequence_id);
 599  endfunction
```

这个与 register_sequence 是相对应的。register_sequence 在 reg_sequences 中插入一条记录，这个函数则从其中删除一条记录，即解除注册。

到此，uvm_sequence_base 的 start 函数分析完毕。

15.2. sequence_item 的产生与发送

15.2.1. start_item

15.1.3 节跳过了 satrt_item 函数，本节重点分析：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：start_item

 737    virtual task start_item (uvm_sequence_item item,
 738                             int set_priority = -1,
 739                             uvm_sequencer_base sequencer=null);
 740      uvm_sequence_base seq;
```

```
741
742       if(item == null) begin
743          uvm_report_fatal("NULLITM",
744              {"attempting to start a null item from sequence '",
745               get_full_name(), "'"}, UVM_NONE);
746          return;
747       end
748
749       if($cast(seq, item)) begin
750          uvm_report_fatal("SEQNOTITM",
751              {"attempting to start a sequence using start_item() from sequence '",
752               get_full_name(), "'. Use seq.start() instead."}, UVM_NONE);
753          return;
754       end
755
756       if (sequencer == null)
757             sequencer = item.get_sequencer();
758
759       if(sequencer == null)
760             sequencer = get_sequencer();
761
762       if(sequencer == null) begin
763             uvm_report_fatal("SEQ",{"neither the item's sequencer nor dedicated sequencer has
    been supplied to start item in ",get_full_name()},UVM_NONE);
764           return;
765       end
766
767       if (sequencer == null)
768          sequencer = item.get_sequencer();
769
770       if (sequencer == null) begin
771             uvm_report_fatal("STRITM", "sequence_item has null sequencer", UVM_NONE);
772       end
773
774       item.set_use_sequence_info(1);
775       item.set_sequencer(sequencer);
776       item.set_parent_sequence(this);
777       item.reseed();
778
779       if (set_priority < 0)
780          set_priority = get_priority();
781
782       sequencer.wait_for_grant(this, set_priority);
783
784       `ifndef UVM_DISABLE_AUTO_ITEM_RECORDING
785          void'(sequencer.begin_child_tr(item, m_tr_handle, item.get_root_sequence_name()));
786       `endif
787
788       pre_do(1);
789
790    endtask
```

749 行判断传入 start_item 的参数是否为一个 sequence。只有是 sequence_item 才能作为参数传入 start_item 中。

756 到 772 行设置 sequencer 信息。经过这几行语句的设置之后，sequencer 一定不为 null，只有在 sequencer 不为 null 的情况下，一个 sequence_item 才可能正常发送。

774 到 777 的语句在分析 uvm_create_on 宏时已经介绍过，这里不重复阐述。

782 行调用 wait_for_grant 函数，这是 uvm_sequencer_base 的一个成员函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：wait_for_grant

 975 task uvm_sequencer_base::wait_for_grant(uvm_sequence_base sequence_ptr,
 976                                          int item_priority = -1,
 977                                          bit lock_request = 0);
 978   uvm_sequence_request req_s;
 979   int my_seq_id;
 980
 981   if (sequence_ptr == null)
 982     uvm_report_fatal("uvm_sequencer",
 983         "wait_for_grant passed null sequence_ptr", UVM_NONE);
 984
 985   my_seq_id = m_register_sequence(sequence_ptr);
 986
 987   // If lock_request is asserted, then issue a lock.  Don't wait for the response, since
 988   // there is a request immediately following the lock request
 989   if (lock_request == 1) begin
 990     req_s = new();
 991     req_s.grant = 0;
 992     req_s.sequence_id = my_seq_id;
 993     req_s.request = SEQ_TYPE_LOCK;
 994     req_s.sequence_ptr = sequence_ptr;
 995     req_s.request_id = g_request_id++;
 996     arb_sequence_q.push_back(req_s);
 997   end
 998
 999   // Push the request onto the queue
1000   req_s = new();
1001   req_s.grant = 0;
1002   req_s.request = SEQ_TYPE_REQ;
1003   req_s.sequence_id = my_seq_id;
1004   req_s.item_priority = item_priority;
1005   req_s.sequence_ptr = sequence_ptr;
1006   req_s.request_id = g_request_id++;
1007   arb_sequence_q.push_back(req_s);
1008   m_update_lists();
1009
1010   // Wait until this entry is granted
```

```
1011     // Continue to point to the element, since location in queue will change
1012     m_wait_for_arbitration_completed(req_s.request_id);
1013
1014     // The wait_for_grant_semaphore is used only to check that send_request
1015     // is only called after wait_for_grant.  This is not a complete check, since
1016     // requests might be done in parallel, but it will catch basic errors
1017     req_s.sequence_ptr.m_wait_for_grant_semaphore++;
1018
1019  endtask
```

985 行得到 sequence 的 id，989 到 997 行则是用于有 lock 请求时，1000 到 1017 行则是系统的仲裁机制。1007 行把一个请求放入了 arb_sequence_q 中：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

   41     protected uvm_sequence_request arb_sequence_q[$];
```

这是一个队列，其存储的内容是 uvm_sequence_request 类型的：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequence_request

1615  class uvm_sequence_request;
1616     bit              grant;
1617     int              sequence_id;
1618     int              request_id;
1619     int              item_priority;
1620     uvm_sequencer_base::seq_req_t    request;
1621     uvm_sequence_base sequence_ptr;
1622  endclass
```

这个类主要用于记录 sequence 向 sequencer 发出的请求信息。在请求信息中，包括了 sequence 的 id，sequence 的指针及请求的类型：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

   37     typedef enum {SEQ_TYPE_REQ,
   38                   SEQ_TYPE_LOCK,
   39                   SEQ_TYPE_GRAB} seq_req_t;
```

请求的类型一共有三种，其中前两种分别对应发送 item 的请求，lock 请求及 grab 请求。最后一个看似是 grab 请求，但是实际上在 grab 操作时，用的是 SEQ_TYPE_LOCK，而不是 SEQ_TYPE_GRAB，后面将会详细介绍。这里使用的是发送 item 的请求。

1008 之后调用 m_update_lists 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
```

```
函数/任务：m_update_lists

 524 function void uvm_sequencer_base::m_update_lists();
 525    m_lock_arb_size++;
 526 endfunction
```

函数相对简单，把 m_lock_arb_size 的值为 1，表示有新的请求产生。

1012 行通过 m_wait_for_arbitration_completed 来等待仲裁完成，关于仲裁机制，下节将会介绍。1017 行则主要是用于保证 send_request 是在 wait_for_grant 之后被调用。

从 wait_for_grant 返回之后，785 行将会调用 begin_child_tr 函数，给要发送的 item 记录一些初始信息，如时间等。788 行调用回调函数 pre_do。

15.2.2. UVM 的 sequence 的仲裁机制

wait_for_grant 的 1012 行调用了 m_wait_for_arbitration_completed 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_wait_for_arbitration_completed

 906 task uvm_sequencer_base::m_wait_for_arbitration_completed(int request_id);
 907    int lock_arb_size;
 908
 909    // Search the list of arb_wait_q, see if this item is done
 910    forever
 911       begin
 912          lock_arb_size   = m_lock_arb_size;
 913
 914          if (arb_completed.exists(request_id)) begin
 915             arb_completed.delete(request_id);
 916             return;
 917          end
 918          wait (lock_arb_size != m_lock_arb_size);
 919       end
 920 endtask
```

这里比较简单，就是不断的查看 arb_completed 中是否有与 request_id 相对应的记录。如果有，说明已经仲裁完毕，此 sequence 可以发送 item。每当 m_lock_arb_size 的值变化就检查一次 arb_completed 数组，因为后面将会看到，每当仲裁完毕时，m_lock_arb_size 将会变化，同时 arb_completed 中会插入记录。但是不仅仅只看 m_lock_arb_size 的变化来判断是否完成仲裁，因为 m_lock_arb_size 的变化可能是因为由于有新的 request 放入 arb_sequence_q 中引起的。

m_lock_arb_size 的变化要依靠 driver 的行为来改变。当在 driver 调用 get_next_item 时，会直接调用 uvm_sequencer 的 get_next_item：

```
文件：src/seq/uvm_sequencer.svh
类：uvm_sequencer
函数/任务：get_next_item

175 task uvm_sequencer::get_next_item(output REQ t);
176   REQ req_item;
177
178   // If a sequence_item has already been requested, then get_next_item()
179   // should not be called again until item_done() has been called.
180
181   if (get_next_item_called == 1)
182     uvm_report_error(get_full_name(),
183       "Get_next_item called twice without item_done or get in between", UVM_NONE);
184
185   if (!sequence_item_requested)
186     m_select_sequence();
187
188   // Set flag indicating that the item has been requested to ensure that item_done or get
189   // is called between requests
190   sequence_item_requested = 1;
191   get_next_item_called = 1;
192   m_req_fifo.peek(t);
193 endtask
```

181 与 185 行的判断条件是为了避免一个 item 重复的被发送的情况。186 行调用 m_select_sequence 函数，之后 192 行会阻塞在那里等待 m_req_fifo 中放入要发送的 item。

m_select_sequence 是在 uvm_sequencer_base 中定义的：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_select_sequence

 653 task uvm_sequencer_base::m_select_sequence();
 654     int selected_sequence;
 655
 656     // Select a sequence
 657     do begin
 658       wait_for_sequences();
 659       selected_sequence = m_choose_next_request();
 660       if (selected_sequence == -1) begin
 661         m_wait_for_available_sequence();
 662       end
 663     end while (selected_sequence == -1);
 664     // issue grant
 665     if (selected_sequence >= 0) begin
 666       m_set_arbitration_completed(arb_sequence_q[selected_sequence].request_id);
```

```
667             arb_sequence_q.delete(selected_sequence);
668             m_update_lists();
669         end
670 endtask
```

658 行调用 wait_for_sequences，这个其实只是简单的#0 延时，并无实质性内容。

659 行调用 m_choose_next_request，此函数会返回可用的 sequence 在 arb_sequence_q 中的序号。如果没有可用的 sequence，那么会返回-1，此时 661 行会执行，一直等到有可用的 sequence。

666 行调用 m_set_arbitration_completed：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_set_arbitration_completed

926 function void uvm_sequencer_base::m_set_arbitration_completed(int request_id);
927     arb_completed[request_id] = 1;
928 endfunction
```

它比较简单，只是在 arb_completed 中插入一条记录。

667 行把选定的 sequence 从仲裁队列中删除，668 行更新 m_lock_arb_size 的值。此值改变，那么 m_wait_for_arbitration_completed 的 918 行将会被释放，表示此 sequence 已经获得了请求，后续的可以调用 finish_item，把 sequence_item 发送给 driver。

这里的关键是 m_choose_next_request 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_choose_next_request

681 function int uvm_sequencer_base::m_choose_next_request();
682     int i, temp;
683     int avail_sequence_count;
684     int sum_priority_val;
685     integer avail_sequences[$];
686     integer highest_sequences[$];
687     int highest_pri;
688     string  s;
689
690     avail_sequence_count = 0;
691
692     grant_queued_locks();
693
694     for (i = 0; i < arb_sequence_q.size(); i++) begin
695       // Search for available sequences.  If in SEQ_ARB_FIFO arbitration,
696       // then just return the first available sequence.  Otherwise,
697       // create a list for arbitration purposes.
```

```
698       if (i < arb_sequence_q.size())
699          if (arb_sequence_q[i].request == SEQ_TYPE_REQ)
700             if (is_blocked(arb_sequence_q[i].sequence_ptr) == 0)
701                if (arb_sequence_q[i].sequence_ptr.is_relevant() == 1) begin
702                   if (m_arbitration == SEQ_ARB_FIFO) begin
703                      return i;
704                   end
705                   else avail_sequences.push_back(i);
706                end
707    end
708
```

这个函数是仲裁机制的具体实现。692 行调用 grant_queued_locks，这是处理 lock 请求时要考虑的，这里暂且先跳过。

700 行用到了 is_blocked 函数，这个也是与 lock 请求相关的，暂且跳过，

701 行判断 is_relevant 函数，这是 uvm_sequence_base 的一个成员函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：is_relevant

524    virtual function bit is_relevant();
525       is_rel_default = 1;
526       return 1;
527    endfunction
```

它用于判断这个 sequence 是不是有效，如果无效，那么 sequencer 在仲裁时将不会考虑这个 sequence。这是 sequence 主动控制自己发送 item 行为的一种方式，通过重载 is_relevant 函数，sequence 可以动态控制是否发送 item。

回到 m_choose_next_request 函数，在 698 到 701 行的条件都满足的情况下，那么根据设置的仲裁算法来决定让哪一个 sequence 发送 item。如果算法是 SEQ_ARB_FIFO，那么将会直接返回当前的请求，否则则把所有可用的请求放入 avail_sequences 中。

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_choose_next_request

709    // Return immediately if there are 0 or 1 available sequences
710    if (m_arbitration == SEQ_ARB_FIFO) begin
711       return -1;
712    end
713    if (avail_sequences.size() < 1)   begin
714       return -1;
715    end
716
717    if (avail_sequences.size() == 1) begin
```

```
718       return avail_sequences[0];
719     end
720
721     // If any locks are in place, then the available queue must
722     // be checked to see if a lock prevents any sequence from proceeding
723     if (lock_list.size() > 0) begin
724       for (i = 0; i < avail_sequences.size(); i++) begin
725         if (is_blocked(arb_sequence_q[avail_sequences[i]].sequence_ptr) != 0) begin
726           avail_sequences.delete(i);
727           i--;
728         end
729       end
730       if (avail_sequences.size() < 1)
731         return -1;
732       if (avail_sequences.size() == 1)
733         return avail_sequences[0];
734     end
735
```

如果 710 与 713 行的条件满足，说明 694 到 707 行的语句根据就没有找到可用的 request，这也就意味着根本就没有 sequence 要求发送 item，此时会直接返回-1。

717 行，如果只有一个 sequence 提出了请求，那么无论是采用什么仲裁方法，都只有选择这个 sequence。

723 到 734 则用于处理有 lock 请求存在时的情况，先跳过。

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_choose_next_request

736     //  Weighted Priority Distribution
737     // Pick an available sequence based on weighted priorities of available sequences
738     if (m_arbitration == SEQ_ARB_WEIGHTED) begin
739       sum_priority_val = 0;
740       for (i = 0; i < avail_sequences.size(); i++) begin
741         sum_priority_val += m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]);
742       end
743
744       temp = $urandom_range(sum_priority_val-1, 0);
745
746       sum_priority_val = 0;
747       for (i = 0; i < avail_sequences.size(); i++) begin
748         if ((m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]) +
749               sum_priority_val) > temp) begin
750           return avail_sequences[i];
751         end
752         sum_priority_val += m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]);
753       end
754       uvm_report_fatal("Sequencer", "UVM Internal error in weighted arbitration code", UV
M_NONE);
755     end
```

```
 756
 757     //   Random Distribution
 758     if (m_arbitration == SEQ_ARB_RANDOM) begin
 759       i = $urandom_range(avail_sequences.size()-1, 0);
 760       return avail_sequences[i];
 761     end
 762
 763     //   Strict Fifo
 764     if ((m_arbitration == SEQ_ARB_STRICT_FIFO) || m_arbitration == SEQ_ARB_STRICT_
RANDOM) begin
 765       highest_pri = 0;
 766       // Build a list of sequences at the highest priority
 767       for (i = 0; i < avail_sequences.size(); i++) begin
 768         if (m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]) > highest_pri) begin
 769           // New highest priority, so start new list
 770           highest_sequences.delete();
 771           highest_sequences.push_back(avail_sequences[i]);
 772           highest_pri = m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]);
 773         end
 774         else if (m_get_seq_item_priority(arb_sequence_q[avail_sequences[i]]) == highest_pri)
begin
 775           highest_sequences.push_back(avail_sequences[i]);
 776         end
 777       end
 778
 779       // Now choose one based on arbitration type
 780       if (m_arbitration == SEQ_ARB_STRICT_FIFO) begin
 781         return(highest_sequences[0]);
 782       end
 783
 784       i = $urandom_range(highest_sequences.size()-1, 0);
 785       return highest_sequences[i];
 786     end
 787
 788     if (m_arbitration == SEQ_ARB_USER) begin
 789       i = user_priority_arbitration( avail_sequences);
 790
 791       // Check that the returned sequence is in the list of available sequences.  Failure to
 792       // use an available sequence will cause highly unpredictable results.
 793       highest_sequences = avail_sequences.find with (item == i);
 794       if (highest_sequences.size() == 0) begin
 795         uvm_report_fatal("Sequencer",
 796             $sformatf("Error in User arbitration, sequence %0d not available\n%s",
 797                       i, convert2string()), UVM_NONE);
 798       end
 799       return(i);
 800     end
 801
 802     uvm_report_fatal("Sequencer", "Internal error: Failed to choose sequence", UVM_NONE);
 803
 804   endfunction
```

从 738 到 800 行，则是根据不同的算法来处理请求。这里的算法定义位于 uvm_object_globals.svh 文件中：

```
文件：src/base/uvm_object__globals.svh
类：无

347 // Enum: uvm_sequencer_arb_mode
348 //
349 // Specifies a sequencer's arbitration mode
350 //
351 // SEQ_ARB_FIFO             - Requests are granted in FIFO order (default)
352 // SEQ_ARB_WEIGHTED         - Requests are granted randomly by weight
353 // SEQ_ARB_RANDOM           - Requests are granted randomly
354 // SEQ_ARB_STRICT_FIFO      - Requests at highest priority granted in fifo order
355 // SEQ_ARB_STRICT_RANDOM - Requests at highest priority granted in randomly
356 // SEQ_ARB_USER             - Arbitration is delegated to the user-defined
357 //                            function, user_priority_arbitration. That function
358 //                            will specify the next sequence to grant.
359
360
361 typedef enum
362 {
363   SEQ_ARB_FIFO,
364   SEQ_ARB_WEIGHTED,
365   SEQ_ARB_RANDOM,
366   SEQ_ARB_STRICT_FIFO,
367   SEQ_ARB_STRICT_RANDOM,
368   SEQ_ARB_USER
369 } uvm_sequencer_arb_mode;
```

SEQ_ARB_FIFO 表示先到的请求先发出，SEQ_ARB_WEIGHTED 则是给每个请求授予一定的权重，根据不同的权重，随机的选择，SEQ_ARB_RANDOM 则是纯粹的随机，即所有的请求的权重相同，SEQ_ARB_STRICT_FIFO 则是表示先把请求按照 priority 分类，然后把有最高 priority 值的按时间顺序发出，SEQ_ARB_STRICT_RANDOM 则是把请求按照 priority 分类，然后从最高 priority 值的请求中随机挑选一个，SEQ_ARB_USER 则是用户自定义的行为。

接下来我们看一下当有 lock 请求时，仲裁机制是如何完成的。lock 请求有两种，一种是 grab，一种是 lock。grab 的定义为：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：grab

1157 task uvm_sequencer_base::grab(uvm_sequence_base sequence_ptr);
1158   m_lock_req(sequence_ptr, 0);
1159 endtask
```

lock 的定义为：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：lock

1149 task uvm_sequencer_base::lock(uvm_sequence_base sequence_ptr);
1150     m_lock_req(sequence_ptr, 1);
1151 endtask
```

这两者都会直接调用 m_lock_req，只是传入的最后一个参数不同。m_lock_req 的定义为：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_lock_req

1085 task uvm_sequencer_base::m_lock_req(uvm_sequence_base sequence_ptr, bit lock);
1086     int my_seq_id;
1087     uvm_sequence_request new_req;
1088
1089     if (sequence_ptr == null)
1090       uvm_report_fatal("uvm_sequence_controller",
1091                                "lock_req passed null sequence_ptr", UVM_NONE);
1092
1093     my_seq_id = m_register_sequence(sequence_ptr);
1094     new_req = new();
1095     new_req.grant = 0;
1096     new_req.sequence_id = sequence_ptr.get_sequence_id();
1097     new_req.request = SEQ_TYPE_LOCK;
1098     new_req.sequence_ptr = sequence_ptr;
1099     new_req.request_id = g_request_id++;
1100
1101     if (lock == 1) begin
1102       // Locks are arbitrated just like all other requests
1103       arb_sequence_q.push_back(new_req);
1104     end else begin
1105       // Grabs are not arbitrated - they go to the front
1106       // TODO:
1107       // Missing: grabs get arbitrated behind other grabs
1108       arb_sequence_q.push_front(new_req);
1109       m_update_lists();
1110     end
1111
1112     // If this lock can be granted immediately, then do so.
1113     grant_queued_locks();
1114
1115     m_wait_for_arbitration_completed(new_req.request_id);
1116 endtask
```

1093 行得到 sequence 的 id，1094 到 1099 行产生一个 SEQ_TYPE_LOCK 的请求，注意，这里是不区分 lock 还是 grab 的。

lock 函数调用时，传入的 lock 为 1，此时这个新产生的请求将会直接送入 arb_sequene_q 中，并且是放入这个队列的最后一个。但是如果是 grab 调用时，则是会放入队列的顶端，这样在发送下一个 item 时，首要的就是处理这个请求。1113 行调用 grant_queued_locks：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：grant_queued_locks

 616 function void uvm_sequencer_base::grant_queued_locks();
 617   int i, temp;
 618
 619   for (i = 0; i < arb_sequence_q.size(); i++) begin
 620
 621     // Check for lock requests.  Any lock request at the head
 622     // of the queue that is not blocked will be granted immediately.
 623     temp = 0;
 624     if (i < arb_sequence_q.size()) begin
 625       if (arb_sequence_q[i].request == SEQ_TYPE_LOCK) begin
 626         temp = (is_blocked(arb_sequence_q[i].sequence_ptr) == 0);
 627       end
 628     end
 629
 630     // Grant the lock request and remove it from the queue.
 631     // This is a loop to handle multiple back-to-back locks.
 632     // Since each entry is deleted, i remains constant
 633     while (temp) begin
 634       lock_list.push_back(arb_sequence_q[i].sequence_ptr);
 635       m_set_arbitration_completed(arb_sequence_q[i].request_id);
 636       arb_sequence_q.delete(i);
 637       m_update_lists();
 638
 639       temp = 0;
 640       if (i < arb_sequence_q.size()) begin
 641         if (arb_sequence_q[i].request == SEQ_TYPE_LOCK) begin
 642           temp = is_blocked(arb_sequence_q[i].sequence_ptr) == 0;
 643         end
 644       end
 645     end
 646   end
 647 endfunction
```

619 行遍历所有的在 arb_sequence_q 中的记录。625 行用于检查记录是否 SEQ_TYPE_LOCK 类型，626 将会调用 is_blocked 函数。我们先跳过这个函数，看 634 行，这一行中将会把一个发出 lock 请求的 sequence 放入 lock_list 中。也就是说，lock_list 中存放的是所有发出 lock 请求的 sequence 的指针。回过来看 is_blocked 函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
```

```
函数/任务：is_blocked

1044 function bit uvm_sequencer_base::is_blocked(uvm_sequence_base sequence_ptr);
1045
1046    if (sequence_ptr == null)
1047       uvm_report_fatal("uvm_sequence_controller",
1048                        "is_blocked passed null sequence_ptr", UVM_NONE);
1049
1050       foreach (lock_list[i]) begin
1051          if ((lock_list[i].get_inst_id() !=
1052               sequence_ptr.get_inst_id()) &&
1053              (is_child(lock_list[i], sequence_ptr) == 0)) begin
1054             return 1;
1055          end
1056       end
1057       return 0;
1058 endfunction
```

如果 lock_list 中没有记录，那么将会直接返回 0，也就是意味着没有 sequence 提出 lock 请求。1051 到 1053 行用于判断 lock_list 中的记录内容是不是由这个 sequence 或者此 sequence 的 parent_sequence 发出的。如果都不是，说明有其它的 sequence 提交了 lock 请求，此 sequencer 已经被其它的 sequence 锁定了，于是会直接返回 1。

回到 grant_queued_locks，当 arb_sequence_q 中有一条记录是 SEQ_TYPE_LOCK 类型的，并且发出这条记录的 sequence 是被 sequencer 允许发送 item 的（即 is_blocked 返回值为 0），那么 temp 将会为 1，执行 633 行的 while 循环。

634 行把发出此 SEQ_TYPE_LOCK 类型请求的 sequence 放入 lock_list 中，635 行在 arb_completed 中插入一条关于此条请求的记录，表示已经仲裁完毕，636 行从 arb_sequence_q 中删除此条记录，640 行到 644 行是在重复 624 到 628 行的事情。

看一下 unlock 和 ungrab：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base

1165 function void uvm_sequencer_base::unlock(uvm_sequence_base sequence_ptr);
1166    m_unlock_req(sequence_ptr);
1167 endfunction

1173 function void  uvm_sequencer_base::ungrab(uvm_sequence_base sequence_ptr);
1174    m_unlock_req(sequence_ptr);
1175 endfunction
```

这两个函数都是调用 m_unlock_req：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_unlock_req
```

```
1124 function void uvm_sequencer_base::m_unlock_req(uvm_sequence_base sequence_ptr);
1125   int my_seq_id;
1126
1127   if (sequence_ptr == null) begin
1128     uvm_report_fatal("uvm_sequencer",
1129                      "m_unlock_req passed null sequence_ptr", UVM_NONE);
1130   end
1131   my_seq_id = m_register_sequence(sequence_ptr);
1132
1133   foreach (lock_list[i]) begin
1134     if (lock_list[i].get_inst_id() == sequence_ptr.get_inst_id()) begin
1135       lock_list.delete(i);
1136       m_update_lists();
1137       return;
1138     end
1139   end
1140   uvm_report_warning("SQRUNL",
1141              {"Sequence '", sequence_ptr.get_full_name(),
1142               "' called ungrab / unlock, but didn't have lock"}, UVM_NONE);
1143 endfunction
```

这里做的事情主要就是从 lock_list 中删除相关的记录。

因此总结一下 grant_queued_locks 做的事情，它会检查 arb_sequence_q 中的记录，如果发现 SEQ_TYPE_LOCK 类型的，那么将会把发出这条记录的 sequence 放入 lock_list 中，表示此 sequence 中已经把 sequencer 给锁定了。假设 arb_sequence_q 中有 3 条 SEQ_TYPE_LOCK 记录，分别是由 seq1，seq2，seq3 发出的，那么调用 grant_queued_locks 时，将处理 seq1 的请求，让 seq1 获得 sequencer 的使用权。arb_sequence_q 中关于 seq1 的记录被删除。seq2 和 seq3 的记录还一直存在。当 seq1 把 sequencer 给释放之后，在 m_choose_next_request 时，会再次调用 grant_queued_locks，此时它将会把 sequencer 的使用权赋予 seq2，直到 seq2 把 sequencer 给释放。

我们回顾 m_choose_next_request 的 700 行，那里调用了 is_blocked 函数，当时我们直接跳过了，现在回过头来想一下，这里的意思是就是看一下 arb_sequence_q 中的要求发送 item 的 sequence 是不是准许被 sequencer 发送 item（即有没有其它的 sequence 把 sequencer 锁定了），如果允许的话，再看一下这个 sequence 自身的 is_realevant 函数，并最终根据算法来决定让哪些 sequence 发送 item。

15.2.3. finish_item

这是 uvm_sequence_base 中定义的一个函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：finish_item

 801    virtual task finish_item (uvm_sequence_item item,
 802                                      int set_priority = -1);
 803
 804      uvm_sequencer_base sequencer;
 805
 806      sequencer = item.get_sequencer();
 807
 808      if (sequencer == null) begin
 809           uvm_report_fatal("STRITM", "sequence_item has null sequencer", UVM_NONE);
 810      end
 811
 812      mid_do(item);
 813      sequencer.send_request(this, item);
 814      sequencer.wait_for_item_done(this, -1);
 815      `ifndef UVM_DISABLE_AUTO_ITEM_RECORDING
 816      sequencer.end_tr(item);
 817      `endif
 818      post_do(item);
 819
 820    endtask
```

806 行得到此 item 的 sequencer，812 行调用回调函数 mid_do。813 行调用 send_request，这是在 uvm_sequencer_param_base 中定义的一个函数：

```
文件：src/seq/uvm_sequencer_param_base.svh
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)
函数/任务：send_request

266 function void uvm_sequencer_param_base::send_request(uvm_sequence_base sequence_ptr,
267                                                      uvm_sequence_item t,
268                                                      bit rerandomize = 0);
269    REQ param_t;
270
271    if (sequence_ptr == null) begin
272      uvm_report_fatal("SNDREQ", "Send request sequence_ptr is null", UVM_NONE);
273    end
274
275    if (sequence_ptr.m_wait_for_grant_semaphore < 1) begin
276      uvm_report_fatal("SNDREQ", "Send request called without wait_for_grant", UVM_NON
E);
277    end
278    sequence_ptr.m_wait_for_grant_semaphore--;
279
280    if ($cast(param_t, t)) begin
281      if (rerandomize == 1) begin
282        if (!param_t.randomize()) begin
283          uvm_report_warning("SQRSNDREQ", "Failed to rerandomize sequence item in send
```

```
_request");
284             end
285         end
286         if (param_t.get_transaction_id() == -1) begin
287             param_t.set_transaction_id(sequence_ptr.m_next_transaction_id++);
288         end
289         m_last_req_push_front(param_t);
290     end else begin
291         uvm_report_fatal(get_name(),$sformatf("send_request failed to cast sequence item"), UV
M_NONE);
292     end
293
294     param_t.set_sequence_id(sequence_ptr.m_get_sqr_sequence_id(m_sequencer_id, 1));
295     t.set_sequencer(this);
296     if (m_req_fifo.try_put(param_t) != 1) begin
297         uvm_report_fatal(get_full_name(),
298                          $sformatf("Sequencer send_request not able to put to fifo, depth; %
0d", m_req_fifo.size()), UVM_NONE);
299     end
300
301     m_num_reqs_sent++;
302     // Grant any locks as soon as possible
303     grant_queued_locks();
304 endfunction
```

271 到 273 行保证 sequence 不为 null，275 到 278 行则是用于保证 wait_for_grant 一定要在此函数之前被调用，在介绍 wait_for_grant 时已经提过这一点。

280 行把要发送的 item 转换为派生类的指针。因为接下来可能牵扯到随机化。如果只是使用 uvm_sequence_item 类型的 t，那么随机化的是 t 中的变量，但是对于那些 param_t 中的成员变量则不能随机化。如对于下面的定义：

```
class my_tran extends uvm_sequence_item;
    rand int a;
    …
endclass
```

如果调用 t 的 randomize，那么 a 是不会随机化的，因为 a 不是 uvm_sequence_item 的成员变量。只有调用 param_t 的 randomize 才能让 a 随机化。

281 行根据 rerandomize 的值来判断是否需要再一次进行随机化。uvm_do_on_pri_with 宏中，在调用 finish_item 之间已经进行过一次随机化了。

286 到 288 行判断 item 的 transaction_id 是否已经设置过了，如果没有设置过，那么就进行设置。287 行用到了 uvm_sequence_base 的 m_next_transaction_id 变量：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 137                int                     m_next_transaction_id = 1;
```

每当有一个新的 item 从 sequence 中产生时，这个变量的值就加 1。

289 调用 m_last_req_push_front，这是 uvm_sequencer_param_base 中定义的函数：

```
文件：src/seq/uvm_sequencer_param_base.svh
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)
函数/任务：m_last_req_push_front

407 function void uvm_sequencer_param_base::m_last_req_push_front(REQ item);
408   if(!m_num_last_reqs)
409     return;
410
411   if(m_last_req_buffer.size() == m_num_last_reqs)
412     void'(m_last_req_buffer.pop_back());
413
414   this.m_last_req_buffer.push_front(item);
415 endfunction
```

与这个函数相关的变量：

```
文件：src/seq/uvm_sequencer_param_base.svh
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)

 41   protected int m_num_last_reqs = 1;
```

可见，在真正的把 item 发送出去之前，sequencer 在其 m_last_req_buffer 中默认保存了最后发送的 item。

回到 send_request，294 行把 sequence 的 id 信息存放在由此 sequence 产生的每一个 item 中。295 行把 item 的 sequencer 赋值为这个 sequence。

296 行向 m_req_fifo 中放入要发送的 item。还记得 uvm_sequencer 的 get_next_item 的 192 行么？那里会等待 m_req_fifo 中有新的 item。这里放入 item 之后，uvm_sequencer 的 get_next_item 将会把这个 item 取走，交给 driver。

301 行记录总共发送了多少个 item。

303 行则再次的调用 grant_queued_locks，意思就是尽快的处理 arb_sequence_q 中的 lock 请求。

到此，send_request 分析完毕，回到 finish_item 中来。814 行调用 wait_for_item_done 函数，这是在 uvm_sequence_base 中定义的函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：wait_for_item_done

1025 task uvm_sequencer_base::wait_for_item_done(uvm_sequence_base sequence_ptr,
1026                                             int transaction_id);
1027   int sequence_id;
1028
```

```
1029    sequence_id = sequence_ptr.m_get_sqr_sequence_id(m_sequencer_id, 1);
1030    m_wait_for_item_sequence_id = -1;
1031    m_wait_for_item_transaction_id = -1;
1032
1033    if (transaction_id == -1)
1034      wait (m_wait_for_item_sequence_id == sequence_id);
1035    else
1036      wait ((m_wait_for_item_sequence_id == sequence_id &&
1037             m_wait_for_item_transaction_id == transaction_id));
1038 endtask
```

这个函数比较简单，1030 与 1031 行复位两个变量，1033 到 1037 行根据这两个变量的值来判断 driver 是否已经调用了 item_done 函数。要了解这两个变量的具体用法，就要看 item_done 函数，这是在 uvm_sequencer 中定义的函数：

```
文件：src/seq/uvm_sequencer.svh
类：uvm_sequencer
函数/任务：item_done

246 function void uvm_sequencer::item_done(RSP item = null);
247   REQ t;
248
249   // Set flag to allow next get_next_item or peek to get a new sequence_item
250   sequence_item_requested = 0;
251   get_next_item_called = 0;
252
253   if (m_req_fifo.try_get(t) == 0) begin
254     uvm_report_fatal(get_full_name(), {"Item_done() called with no outstanding requests.",
255       " Each call to item_done() must be paired with a previous call to get_next_item()."});
256   end else begin
257     m_wait_for_item_sequence_id = t.get_sequence_id();
258     m_wait_for_item_transaction_id = t.get_transaction_id();
259   end
260
261   if (item != null) begin
262     seq_item_export.put_response(item);
263   end
264
265   // Grant any locks as soon as possible
266   grant_queued_locks();
267 endfunction
```

当一个 driver 发送完一个 item 时，会调用 seq_item_port 的 item_done，而这个 item_done 实际上就是 uvm_sequencer 的 item_done。253 行从 m_req_fifo 中试探着取，看是否能取到。这里为什么要这么做？因为 257 与 258 行要设置两个变量的值，而这两个变量的值与发送的 item 直接相关。而在 dirver 中调用 item_done 时，并没有把发送的 item 做为参数传递进来，所以这里需要把刚刚 driver 发送的 item 再次取出来。item_done 有一个输入的参数，这个牵扯到了 response 机制，后面会仔细介绍。

回到 wait_for_item_done，当 item_done 被调用后，那么两个变量的值被设置，于是 wait_for_item_done 将会直接返回，finish_item 的 816 行将会调用 end_tr 函数。这个也不多做介绍。818 行调用回调函数 post_do。

15.3. sequence 的常用功能

15.3.1. default_sequence 的自动启动

关于 default_sequence 的自动启动功能，在 phase 机制中，介绍 uvm_task_phase 的时候曾经提及。uvm_task_phase 的 excute 函数如下：

```
文件：src/base/uvm_task_phase.svh
类：uvm_task_phase
函数/任务：execute

133    protected virtual function void execute(uvm_component comp,
134                                            uvm_phase phase);
135
136      fork
137        begin
138          uvm_sequencer_base seqr;
139
140          phase.m_num_procs_not_yet_returned++;
141
142          if ($cast(seqr,comp))
143            seqr.start_phase_sequence(phase);
144
145          exec_task(comp,phase);
146
147          phase.m_num_procs_not_yet_returned--;
148
149        end
150      join_none
151
152    endfunction
```

142 行会判断这个 component 是不是一个 uvm_sequencer_base 类型的变量，如果是的话，那么就调用 start_phase_sequence，这是一个在 uvm_sequencer_base 中定义的成员函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：start_phase_sequence

1343 function void uvm_sequencer_base::start_phase_sequence(uvm_phase phase);
1344       uvm_object_wrapper wrapper;
1345       uvm_sequence_base  seq;
1346       uvm_factory f = uvm_factory::get();
1347
1348       // default sequence instance?
1349       if (!uvm_config_db #(uvm_sequence_base)::get(
1350             this, {phase.get_name(),"_phase"}, "default_sequence", seq) || seq == null) begin
1351         // default sequence object wrapper?
1352         if (uvm_config_db #(uvm_object_wrapper)::get(
1353                   this, {phase.get_name(),"_phase"}, "default_sequence", wrapper) && wrapp
er != null) begin
1354           // use wrapper is a sequence type
1355           if(!$cast(seq , f.create_object_by_type(
1356                 wrapper, get_full_name(), wrapper.get_type_name()))) begin
1357             `uvm_warning("PHASESEQ", {"Default sequence for phase '",
1358                          phase.get_name(),"' %s is not a sequence type"})
1359             return;
1360           end
1361         end
1362         else begin
1363           `uvm_info("PHASESEQ", {"No default phase sequence for phase '",
1364                                  phase.get_name(),"'"}, UVM_FULL)
1365           return;
1366         end
1367       end
1368
1369       `uvm_info("PHASESEQ", {"Starting default sequence '",
1370          seq.get_type_name(),"' for phase '", phase.get_name(),"'"}, UVM_FULL)
1371
1372       seq.print_sequence_info = 1;
1373       seq.set_sequencer(this);
1374       seq.reseed();
1375       seq.starting_phase = phase;
1376
1377       if (!seq.is_randomized && !seq.randomize()) begin
1378         `uvm_warning("STRDEFSEQ", {"Randomization failed for default sequence '",
1379          seq.get_type_name(),"' for phase '", phase.get_name(),"'"})
1380          return;
1381       end
1382
1383       fork
1384         seq.start(this);
1385       join_none
1386
1387 endfunction
```

1349 行与 1350 行判断是否把一个 sequence 的实例作为 default_sequence 设置给

了此 phase。也即是检测如下方式的 config：

```
my_seq ms;
ms = my_seq::type_id::create("ms");
uvm_config_db#(uvm_sequence_base)::set(this, "env.agent.sqr.main_phase", "default_sequence", ms);
```

这种设置方法必须把一个 sequence 实例化之后才能作为 set 的参数。

1352 行则是上述检测没有检测到的情况下，检查下面这种设置：

```
uvm_config_db#(uvm_object_wrapper)::set(this, "env.agent.sqr.main_phase", "default_sequence", my_seq::type_ id::get());
```

1355 行把根据第二种检测方式得到的变量进行实例化，并且查看创建的实例是不是一个 uvm_sequence_base 类型的实例，如果不是，那么这就表明通过 config_db 方式设置的不是一个 uvm_sequence_base 型的 uvm_object_wrapper。

1363 到 1365 行在前两种方式都没有检测到 default_sequence 的情况下，给出信息。

1372 到 1381 行做 sequence 启动前的准备工作。这里需要重视的是 1375 行。在 uvm_sequence_base 中有如下的成员变量：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 475    uvm_phase starting_phase;
```

在 uvm_sequence_base 实例化的时候，这个变量是没有实例化的。1375 行给其赋值之后，在 sequence 的 body 或者 pre_body 及 post_body 才可以这么做：

```
starting_phase.raise_objection(this);
starting_phase.drop_objection(this);
```

所以说在使用 starting_phase 的时候一定要小心，使用前一定要判断一下其是否为 null。

```
if(starting_phase != null)
  starting_phase.raise_objection(this);
```

如果不加判断，那么经常会不小心就会出错。一种常见的出错情况如下：

```
class my_seq extends uvm_sequence#(my_transaction);
    task body();
        starting_phase.raise_objection(this);
        …
        starting_phase.drop_objection(this);
    endtask
    …
endclass
class my_vseq extends uvm_sequence;
```

```
    task body();
        my_seq ms;
        `uvm_do_on(ms, p_sequencer.sqr)
        …
    endtask
    …
endclass
```

在 case 的 build_phase 里面把 my_vseq 作为某 virtual_sequencer 的 default_sequence，这样 my_vseq 中的 starting_phase 就是一个非 null 的变量，但是前面我们分析 uvm_do 系列宏时，并没有看到系统会把 parent_sequence 的 starting_phase 赋值给子 sequence 的 starting_phase，所以 my_seq 的 starting_phase 实质上等于 null。

另外，如果是自己启动 sequencce，那么也可能会有这种情况出现：

```
class my_vseq2 extends uvm_sequence;
    task body();
        my_seq ms;
        ms = my_seq::type_id::create("ms");
        ms.start(p_sequencer.sqr);
        …
    endtask
    …
endclass
```

可以通过显示的赋值来把 starting_phase 的值从 my_vseq2 传递到 my_seq：

```
ms.starting_phase = this.starting_phase;
ms.start(p_sequencer.sqr);
```

回到 start_phase_sequence 中，1383 行真正的启动 sequence，start 函数又会顺序的调用 pre_start，pre_body，body，post_body，post_start 等，从而完成 sequence 的启动与执行。

15.3.2. p_sequencer 与 m_sequencer 的区别

前面曾经介绍过，如果要使用 sequencer 中的某些变量，如下面：

```
class my_sequencer extends uvm_sequencer;
    event irq_event;
    …
endclass
```

如果要在 sequence 中使用 irq_event，那么必须使用宏来显式的声明：

```
class my_seq extends uvm_sequence;
    `uvm_declare_p_sequencer(my_sequencer)
```

```
    taks  body();
        @p_sequencer.irq_event;
        …
    endtask
    …
endclass
```

我们来看一下 uvm_declare_p_sequencer 宏：

```
文件：src/macros/uvm_sequence_defines.svh
类：无

446 `define uvm_declare_p_sequencer(SEQUENCER) \
447     SEQUENCER p_sequencer;\
448     virtual function void m_set_p_sequencer();\
449         super.m_set_p_sequencer(); \
450         if( !$cast(p_sequencer, m_sequencer)) \
451             `uvm_fatal("DCLPSQ", \
452             $sformatf("%m %s Error casting p_sequencer, please verify that this sequence/seque
nce item is intended to execute on this type of sequencer", get_full_name())) \
453     endfunction
```

这个宏相当简单，定义了一个 SEQUENCE 类型的变量 p_sequencer，在我们的例子中就是定义了一个 my_sequencer 类型的变量 p_sequencer，之后重载 m_set_p_sequencer 函数。这种重载只是单纯的把 m_sequencer 的值赋给了 p_sequencer。

在 uvm_sequence_base 的 start 函数的 276 行，会调用 m_set_p_sequencer，如果没有使用 uvm_declare_p_sequencer 宏，那么这将会是一个空函数，但是经过宏的声明后，这个函数被重载了。于是 p_sequencer 与 m_sequencer 从本质上来说是指向同一个 sequencer 的实例。

为什么要有一个 p_sequencer 呢？因为 m_sequencer 是一个 uvm_sequencer_base 类型的变量，对于上面的 irq_event，m_sequencer 中是没有这个成员变量的，而 irq_event 是属于 uvm_sequencer_base 的派生类（my_sequencer）中定义的成员变量。要使用这个成员变量，必须要声明一个 my_sequencer 类型的 p_sequencer。

15.4. sequence 的 response

15.4.1. put_respont 与 get_response

如果在一个 sequence，发送完一个 item 之后，等待 driver 返回一个 transaction，那么通常可以这么写：

```
task body()
  my_transaction tr, rsp;
  ‘uvm_do(tr)
  get_response(rsp);
  //do something according to rsp
endtask
```

而在 driver 中则需要这么写：

```
task my_driver::main_phase(uvm_phase phase);
    …
    while(1) begin
        seq_item_port.get_next_item(req);
        …
        rsp.set_id_info(req);
        seq_item_port.put_response(rsp);
        seq_item_port.item_done();
    end
endtask
```

get_response 是在 uvm_sequence 中定义的：

```
文件：src/seq/uvm_sequence.svh
类：uvm_sequence
函数/任务：get_response

105    virtual task get_response(output RSP response, input int transaction_id = -1);
106       uvm_sequence_item rsp;
107       get_base_response( rsp, transaction_id);
108       $cast(response,rsp);
109    endtask
```

它会直接的调用 get_base_response，这是在 uvm_sequence_base 中定义的函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：get_base_response
```

```
 996    virtual task get_base_response(output uvm_sequence_item response, input int transaction_i
d = -1);
 997
 998      int queue_size, i;
 999
1000      if (response_queue.size() == 0)
1001        wait (response_queue.size() != 0);
1002
1003      if (transaction_id == -1) begin
1004        response = response_queue.pop_front();
1005        return;
1006      end
1007
1008      forever begin
1009        queue_size = response_queue.size();
1010        for (i = 0; i < queue_size; i++) begin
1011          if (response_queue[i].get_transaction_id() == transaction_id)
1012            begin
1013              $cast(response,response_queue[i]);
1014              response_queue.delete(i);
1015              return;
1016            end
1017        end
1018        wait (response_queue.size() != queue_size);
1019      end
1020    endtask
```

整个函数比较清晰，1003 到 1006 行表明不需要 transaction_id 进行匹配时，那么只要 response_queue 中有了新的记录就取出并作为最终的返回值。responset_queue 的定义如下：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 146    protected uvm_sequence_item response_queue[$];
```

这是一个队列，它存放的内容是 uvm_sequence_item 类型的。

1008 到 1019 行是用于需要 transaction_id 进行精确匹配的时候。

可以推测，put_response 的行为一定会影响 response_queue。当我们在 driver 中调用 put_response，其实质是调用 uvm_sequencer 的 put_response。关于这一点，可以参考 TLM1.0 源代码分析部分相关解释。put_response 是在 uvm_sequencer_param_base 中定义的函数：

```
文件：src/seq/uvm_sequencer_param_base
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)
函数/任务：put_response

310 function void uvm_sequencer_param_base::put_response (RSP t);
311   uvm_sequence_base sequence_ptr;
```

```
312
313     if (t == null) begin
314       uvm_report_fatal("SQRPUT", "Driver put a null response", UVM_NONE);
315     end
316
317     m_last_rsp_push_front(t);
318     m_num_rsps_received++;
319
320     // Check that set_id_info was called
321     if (t.get_sequence_id() == -1) begin
322   `ifndef CDNS_NO_SQR_CHK_SEQ_ID
323       uvm_report_fatal("SQRPUT", "Driver put a response with null sequence_id", UVM_NO
NE);
324   `endif
325       return;
326     end
327
328     sequence_ptr = m_find_sequence(t.get_sequence_id());
329
330     if (sequence_ptr != null) begin
331       // If the response_handler is enabled for this sequence, then call the response handler
332       if (sequence_ptr.get_use_response_handler() == 1) begin
333         sequence_ptr.response_handler(t);
334         return;
335       end
336
337       sequence_ptr.put_response(t);
338     end
339     else begin
340       uvm_report_info("Sequencer",
341                       $sformatf("Dropping response for sequence %0d, sequence not found.
  Probable cause: sequence exited or has been killed",
342                                 t.get_sequence_id()));
343     end
344   endfunction
```

313 到 315 行用于保证要返回给 sequence 的 rsp 不为 null。317 行调用 m_last_rsp_push_front：

```
文件：src/seq/uvm_sequencer_param_base
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)
函数/任务：m_last_rsp_push_front

449 function void uvm_sequencer_param_base::m_last_rsp_push_front(RSP item);
450     if(!m_num_last_rsps)
451       return;
452
453     if(m_last_rsp_buffer.size() == m_num_last_rsps)
454       void'(m_last_rsp_buffer.pop_back());
455
456     this.m_last_rsp_buffer.push_front(item);
457 endfunction
```

这里做的事情其实与 sequencer 发送 item 给 driver 非常相似。sequencer 会保留最后发送的 item。这个可以参考分析 finish_item 时调用 m_last_req_push_front。同样的，这里 sequencer 也把要发送给 sequence 的 rsp 也保留一份。

321 到 326 行检查这个 rsp 是属于哪个 sequence 的，如果 sequence_id 为-1，则说明根本没有进行设置过。在上面的 drive 的代码中，如果不调用 rsp.set_id_info(req) 而直接使用 put_response，那么就会出现这种情况。

328 行调用 m_find_sequence 函数。这是在 uvm_sequencer_base 中定义的成员函数：

```
文件：src/seq/uvm_sequencer_base.svh
类：uvm_sequencer_base
函数/任务：m_find_sequence

 573 function uvm_sequence_base uvm_sequencer_base::m_find_sequence(int sequence_id);
 574   uvm_sequence_base seq_ptr;
 575   int               i;
 576
 577   // When sequence_id is -1, return the first available sequence.  This is used
 578   // when deleting all sequences
 579   if (sequence_id == -1) begin
 580     if (reg_sequences.first(i)) begin
 581       return(reg_sequences[i]);
 582     end
 583     return(null);
 584   end
 585
 586   if (!reg_sequences.exists(sequence_id))
 587     return null;
 588   return reg_sequences[sequence_id];
 589 endfunction
```

其做的事情就是遍历 reg_sequences，看看输入的 sequence_id 是否存在。如果不存在，那么执行 339 行的分支，给出错误提示，否则执行 331 行的分支。在这个分支中，332 到 335 行用到了 get_use_response_handler 函数，下一节将会讲述。在这里，将会执行 337 行的语句，即调用 sequence 的 put_response 函数，这是在 uvm_sequence 中定义的函数：

```
文件：src/seq/uvm_sequence.svh
类：uvm_sequence
函数/任务：put_response

117   virtual function void put_response(uvm_sequence_item response_item);
118     RSP response;
119     if (!$cast(response, response_item)) begin
120       uvm_report_fatal("PUTRSP", "Failure to cast response in put_response", UVM_NON
E);
121     end
```

```
122         put_base_response(response_item);
123     endfunction
```

这个函数会直接调用 put_base_response 函数，这是在 uvm_sequence_base 中定义的函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：put_base_response

 973     virtual function void put_base_response(input uvm_sequence_item response);
 974         if ((response_queue_depth == -1) ||
 975                 (response_queue.size() < response_queue_depth)) begin
 976             response_queue.push_back(response);
 977             return;
 978         end
 979         if (response_queue_error_report_disabled == 0) begin
 980             uvm_report_error(get_full_name(), "Response queue overflow, response was dropped",
 UVM_NONE);
 981         end
 982     endfunction
```

974 与 975 行用于判断 sequence 中 response_queue 是否可以容纳新的 response。response_queue_depth 为-1 表示可以无限制放入 response，而如果其不为-1，那么就要查看 response_queue 中记录的数量是否超过了设置的 response_queue_depth：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base

 147     protected int                       response_queue_depth = 8;
```

response_queue_depth 的默认值为 8，可见默认情况下，是不能无限制的放入 response 的。

976 行把 response 放入 response_queue 中。这样等待在那里的 get_base_response 将会得到这个 response。

979 到 981 行则是表明 response_queue 中已经满了，不能再放入新的 response 了。

15.4.2. 使用 response_handler

上节讲述的 get_response 和 put_response 是一一对应的。当在 sequence 中启动 get_response 时，进程就会阻塞在那里，一直到 response_queue 中被放入新的记录。如果 driver 能够马上把 response 通过 put_response 的方式传回 sequence，那么 sequence

被阻塞的进程就会得到释放，可以接着发送下一个 transaction 给 driver。但是假如 driver 需要延时较长的一段时间才能把 transaction 传回，此时 driver 和 sequence 相当于是都被阻塞在那里了。

上述的情况主要是源于 sequence 和 driver 之间 put_response 和 get_response 的同步。假如把这种同步分散开来，那么将会得到不同的结果。在这种情况下需要使用 response_handler。

```
class my_seq extends uvm_sequence#(my_transaction);
    task pre_body();
        use_response_handler(1);
    endtask
    task body();
        …
    endtask
    function void response_handler(uvm_sequence_item response);
        my_transaction rsp;
        if(!$cast(rsp, response))
            `uvm_error("response_handler", "response item type error!");
        else
            rsp.print();
        …
    endfunction
    …
endclass
```

要使用 response_handler，那么首先需要调用 use_response_handler 函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：use_response_handler

 895    function void use_response_handler(bit enable);
 896        m_use_response_handler = enable;
 897    endfunction
```

函数比较简单，只是单纯的给 m_use_response_handler 赋值。

经过这样的设置后，在 body 中，可以不用使用 get_response 函数了。当一个 item 发送完毕后可以接下来发送新的 item。

上节在介绍 uvm_sequencer_param_base 的 put_response 函数时，在其 332 行时遇到过使用 response_handler 的情况：

```
文件：src/seq/uvm_sequencer_param_base
类：uvm_sequencer_param_base #(type REQ = uvm_sequence_item, type RSP = REQ)
函数/任务：put_response

310 function void uvm_sequencer_param_base::put_response (RSP t);
        …
```

```
330     if (sequence_ptr != null) begin
331       // If the response_handler is enabled for this sequence, then call the response handler
332       if (sequence_ptr.get_use_response_handler() == 1) begin
333         sequence_ptr.response_handler(t);
334         return;
335       end
336
337       sequence_ptr.put_response(t);
338     end
339     else begin
340       uvm_report_info("Sequencer",
341                       $sformatf("Dropping response for sequence %0d, sequence not found.
  Probable cause: sequence exited or has been killed",
342                                 t.get_sequence_id()));
343     end
344   endfunction
```

这里通过调用 get_use_response_handler 来得到 m_use_response_handler 的值，如果使用了那么就执行 333 行，调用 response_handler，334 行退出。

response_handler 是一个在 uvm_sequence_base 中定义的函数：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：response_handler

 914    virtual function void response_handler(uvm_sequence_item response);
 915      return;
 916    endfunction
```

这是一个空函数，用户必须重载这个函数，在其中填入自己想要的代码，这样才能真正的实现 response 机制。上面的例子就重载了 response_handler，其中仅仅是把 response 打印了一遍。在实际使用时，response_handler 将会远比这个要复杂。

16. config_db 机制源代码分析

config_db 机制是 UVM 中用于在不同 component 之间共享资源的一种机制，它满足了资源共享的要求，同时又避免了全局变量的弊端。其实，本章或许称为 resource_db 机制更加贴切一点，因为所有的工作都是由 resource_db 机制来完成的，config_db 机制只是在外面套了一个套子而已。config 机制是由 OVM 流传下来的，只是为了保持与 OVM 兼容，所以才搞出一个专门的 config_db 机制。

16.1. 基本的数据结构

resource_db 机制主要就是用于资源的共享。既然说到共享，那么有三个问题是不得不考虑的：一是资源是存放在什么地方的，比如我们把资源放在一个队列中，也可以放在一个联合数组中，还可以放在一个动态数组中；二是资源是以什么形式存放的，这个问题看起来比较简单，因为我们可以直接存放值就行了，如要共享一个 int 型的资源，那么就直接把这个 int 的值放入一个联合数组就可以了，这个想法固然不错，但是问题在于后面要用到资源的查找等功能，所以仅仅只存放值是不行的；三是资源是如何存取的，即怎么样通过 set 函数把要共享的资源放入要存放的地方，怎么使用 get 函数把资源从存放的地方取出。

在 resource_db 机制中，这三个功能分别通过三个类来实行，实现第一个功能的是 uvm_resource_pool 类，实现第二个功能的是 uvm_resource#(type T)类，实现第三个功能的是 uvm_resource_db#(type T)类。本节先介绍第二和第二个功能，后两节介

绍第三个功能

16.1.1. 资源的存放形式

resource_db 机制中，使用 uvm_resource#(type T)类来组织各种各样的资源。这个类派生自 uvm_resource_base：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)

1393 class uvm_resource #(type T=int) extends uvm_resource_base;
```

而 uvm_resource_base 是一个纯虚类，它派生自 uvm_object：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base

 199 virtual class uvm_resource_base extends uvm_object;
```

当一个资源被共享时，需要记录这个资源的哪些信息呢？或者举一个例子：

```
uvm_config_db#(int)::set(this, “tb.env.agent.driver”, “ifg_num”, 8);
```

在上面的例子中，我们至少可以猜测，为 ifg_num 这个资源要存放两点，一是这个资源是要共享给谁的，这里是“tb.env.agent.driver”，二是”8”这个值。另外，由本书前半部分关于 config_db 机制的介绍，config_db 的 set 是有优先级的，所以系统还需要存放是由谁来进行了 set，是由最顶层的 case 设置的还是由 tb 或者 env 来设置的？

在 uvm_resource_base 中有使用一个字符串变量 scope 来存放目标路径信息：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base

 201    protected string scope;
```

使用一个整型变量来存放优先级信息：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base

 218    int unsigned precedence;
```

假如在 case 的 build_phase 使用 config_db 机制 set 了 ifg_num，而在 env 中同样 set 了一次，那么在 case 中 set 的那一次可以给予其 1000 的优先级，而在 env 中则在

1000 的基础上减去 1。这样在 driver 中使用 get 函数得到 ifg_num 的信息时，会发现两条同样的记录，系统会选择 precedence 较大的那一条，从而实现了 config_db 机制的优先级问题。

关于具体值的存放，那是由 uvm_resource#(type T)中的变量 val 来实现的：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)

1401    protected T val;
```

因为牵扯到了类型 T，所以这个成员变量只能在 uvm_resource#(type T)中定义，而不能在 uvm_resource_base 中定义。

16.1.2. 资源的存放地点

resource_db 机制中，使用 uvm_resource_pool 类来存放各种各样的资源，这是一个单实例的类，有一个全局的变量 uvm_resources：

```
文件：src/base/uvm_resource.svh
类：无

1658 const uvm_resource_pool uvm_resources = uvm_resource_pool::get();
```

get 函数比较简单：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：get

 680    static function uvm_resource_pool get();
 681      if(rp == null)
 682        rp = new();
 683      return rp;
 684    endfunction
```

这里出现了变量 rp：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool

 659    static local uvm_resource_pool rp = get();
```

关于这种单实例的实现方法，在 uvm_root 中已经详细介绍过，这里不再重复。

在 uvm_resource_pool 中，使用两个联合数组来存放资源：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool

 661    uvm_resource_types::rsrc_q_t rtab [string];
 662    uvm_resource_types::rsrc_q_t ttab [uvm_resource_base];
```

这里出现了 uvm_resource_types 类：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_types

 111 class uvm_resource_types;
 112
 113    // types uses for setting overrides
 114    typedef bit[1:0] override_t;
 115    typedef enum override_t { TYPE_OVERRIDE = 2'b01,
 116                              NAME_OVERRIDE = 2'b10 } override_e;
 117
 118     // general purpose queue of resourcex
 119    typedef uvm_queue#(uvm_resource_base) rsrc_q_t;
 120
 121    // enum for setting resource search priority
 122    typedef enum { PRI_HIGH, PRI_LOW } priority_e;
 123
 124    // access record for resources.  A set of these is stored for each
 125    // resource by accessing object.  It's updated for each read/write.
 126    typedef struct
 127    {
 128       time read_time;
 129       time write_time;
 130       int unsigned read_count;
 131       int unsigned write_count;
 132    } access_t;
 133
 134 endclass
```

这个类是用来组织各种资源类型的。uvm_resource_types::rsrc_q_t 是一个队列，这个队列存放的内容是 uvm_resource_base 类型。因此，rtab 是一个联合数组，这个联合数组的索引是 string，而其内容则是一个队列。ttab 也是一个联合数组，其索引是 uvm_resource_base 类型，而内容也是一个队列。

rtab 中的不同的 queue 中存放的 uvm_resource_base 的名字是不一样的，同一 queue 中存放的名字是一样的，这个名字就是 queue 的索引。ttab 中不同 queue 存放的 uvm_resource_base 的类型是不一样的，这里的类型主要体现在由其派生而来的 uvm_resource#(type T)中的 T 是不一样的。同一 queue 中存放的类型是一样的。在 rtab 中，对于同一个名字，其 queue 里面存放的东西可能是不同类型的，因为不同类型的资源都可以叫同一名字，都会被放入 rtab 中的同一条记录的队列中。同样的，ttab 中同一条记录对应的 queue 里存放的东西可能并不都是同一名字。

16.2. 资源的写入

本节和下一节介绍资源的存取实现。本节先介绍资源的写入。

16.2.1. uvm_resource_db 类

资源的存取通过 uvm_resource_db 类来实现。这个类的所有的成员变量都是静态的，所以对于这个类来说，是否实例化并不重要。类的原型为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)

 53 class uvm_resource_db #(type T=uvm_object);
```

这个类本身并没有派生自任何类，而是自成一家。其中的 set 函数负责向 uvm_resource_ pool 中写入资源：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set

138    static function void set(input string scope, input string name,
139                             T val, input uvm_object accessor = null);
140
141      rsrc_t rsrc = new(name, scope);
142      rsrc.write(val, accessor);
143      rsrc.set();
144
145      if(uvm_resource_db_options::is_tracing())
146        m_show_msg("RSRCDB/SET", "Resource","set", scope, name, accessor, rsrc);
147    endfunction
```

函数有四个参数，其中的 scope 表示路径信息，name 表示名字，而 val 则是要共享的资源，最后一个 accessor 比较古怪，我们可以先把其做为一个疑问。

141 行出现了 rsrc_t，其定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)

 55    typedef uvm_resource #(T) rsrc_t;
```

141 行实例化了一个 uvm_resource#(T)的变量 rsrc 中，142 行通过调用 write 函

数把 val 的值写入 rsrc 中。这样，rsrc 中就有了 val 的值，有了 scope 的值。143 行则把 rsrc 写入到全局的 uvm_resource_pool 中。下面分别介绍 uvm_resource#(T)的 new 函数，write 函数和 set 函数。

16.2.2. uvm_resource#(T)的 new 函数

uvm_resource#(type T)的 new 函数的定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：new

1403    function new(string name="", scope="");
1404      super.new(name, scope);
1405    endfunction
```

这里仅仅只是调用了 uvm_resource_base 的 new 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base
函数/任务：new

 234    function new(string name = "", string s = "*");
 235      super.new(name);
 236      set_scope(s);
 237      modified = 0;
 238      read_only = 0;
 239      precedence = default_precedence;
 240      if(uvm_has_wildcard(name))
 241        m_is_regex_name = 1;
 242    endfunction
```

我们以一个例子来讲解这个 new 函数：

```
uvm_resource_db#(int)::set(“a.b.c”, “blk_num”, 8);
```

236 行调用 set_scope 函数，传入的参数是”a.b.c”：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base
函数/任务：set_scope

 390    function void set_scope(string s);
 391      scope = uvm_glob_to_re(s);
 392    endfunction
```

这个函数相当简单，只是调用了 uvm_glob_to_re。uvm_glob_to_re 是一个字符串处理函数，它把输入的字符串转换成正则表达式的形式。这里用到了 DPI 调用，使用 C 函数来完成转换。如对于上面的 a.b.c，经过转换后就变成了/a\.b\.c/，即在前后各加了一个代表正则表达式的斜线，同时把.使用反斜丝转义。在”a.b.c”的表述中，”.”只是起分隔作用，这种用法是 glob，但是在正则表达式中，”.”则是元字符，有特殊含义，关于这一点，可以看正则表达式的相关书籍。因此，为了在正则表达式中表述”.”的本来意思，那么就需要使用反斜线来进行转义。除了”.”号外，这个函数还会对”*”号，”+”号，”[“号，”]”号等属于 glob 的字符替换成正则表达式的字符。这里牵扯到了 glob 与正则表达式的对应关系，关于这一点，可以参考相关书籍。假设本来输入的就是” /a.b.c/”，那么输出会是什么？答案是” /a.b.c/”因为这个函数会检测输入本身是否是一个正则表达式，这种检测是通过查看第一个字符是不是”/”来完成的。

回到 set_scope 函数，这个函数的主要用意就是给类的成员变量 scope 进行初始化，存入这个资源的目标路径（即这个资源是要被谁使用的）。

new 函数的 239 行设置了优先级信息，把其设置为默认的 default_precedence：

static int unsigned default_precedence = 1000;

也就是说，默认情况下，所有资源的优先级都为 1000。

240 行调用 uvm_has_wildcard 函数，传入的参数是 name：

文件：src/base/uvm_misc.svh
类：无
函数/任务：uvm_has_wildcard

```
558 function automatic bit uvm_has_wildcard (string arg);
559   uvm_has_wildcard = 0;
560
561   //if it is a regex then return true
562   if( (arg.len() > 1) && (arg[0] == "/") && (arg[arg.len()-1] == "/") )
563     return 1;
564
565   //check if it has globs
566   foreach(arg[i])
567     if( (arg[i] == "*") || (arg[i] == "+") || (arg[i] == "?") )
568       uvm_has_wildcard = 1;
569
570 endfunction
```

这个函数的主要检查输入的 name 中是否有通配符，如*等字符。如果有的话，则把 m_is_regex_name 赋值为 1。

new 函数主要完成了两件事情：一是设置 scope 信息，二是设置 precedence 信息。

16.2.3. uvm_resource#(T)的 write 函数

函数的定义如下：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：write

1574    function void write(T t, uvm_object accessor = null);
1575
1576      if(is_read_only()) begin
1577        uvm_report_error("resource", $sformatf("resource %s is read only -- cannot modify",
get_name()));
1578        return;
1579      end
1580
1581      record_write_access(accessor);
1582
1583      // set the value and set the dirty bit
1584      val = t;
1585      modified = 1;
1586    endfunction
```

1576 到 1579 行检查这个资源是不是只读的，如果是只读的，那么是不能进行 write 操作的，从而给出出错提示。

1581 行调用 record_write_acccess 函数，这是在 uvm_resource_base 中定义的函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base
函数/任务：record_write_acccess

 511    function void record_write_access(uvm_object accessor = null);
 512
 513      string str;
 514
 515      // If an accessor object is supplied then get the accessor record.
 516      // Otherwise create a new access record.  In either case populate
 517      // the access record with information about this access.  Check
 518      // first that auditing is turned on
 519
 520      if(uvm_resource_options::is_auditing()) begin
 521        if(accessor != null) begin
 522          uvm_resource_types::access_t access_record;
 523          string str;
 524          str = accessor.get_full_name();
 525          if(access.exists(str))
 526            access_record = access[str];
```

```
527             else
528                init_access_record(access_record);
529             access_record.write_count++;
530             access_record.write_time = $realtime;
531             access[str] = access_record;
532          end
533       end
534    endfunction
```

520 行调用 uvm_resource_options::is_auditing 函数。uvm_resource_options 类是一个专门用于设置 resource_db 机制相关信息的一个类：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_options

158 class uvm_resource_options;
159
160    static local bit auditing = 1;
       …
168    static function void turn_on_auditing();
169       auditing = 1;
170    endfunction
       …
178    static function void turn_off_auditing();
179       auditing = 0;
180    endfunction
       …
186    static function bit is_auditing();
187       return auditing;
188    endfunction
189 endclass
```

这个类的核心就是控制 auditing 功能。在默认情况下，这个功能是打开的。所以 record_write_acccess 的 520 行在默认情况下会返回 1。521 行检查传入的 accessor 是否为 null。

522 行出现了 uvm_resource_types 类：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_types

111 class uvm_resource_types;
112
113    // types uses for setting overrides
114    typedef bit[1:0] override_t;
115    typedef enum override_t { TYPE_OVERRIDE = 2'b01,
116                              NAME_OVERRIDE = 2'b10 } override_e;
117
118     // general purpose queue of resourcex
119    typedef uvm_queue#(uvm_resource_base) rsrc_q_t;
120
```

```
121    // enum for setting resource search priority
122    typedef enum { PRI_HIGH, PRI_LOW } priority_e;
123
124    // access record for resources.  A set of these is stored for each
125    // resource by accessing object.  It's updated for each read/write.
126    typedef struct
127    {
128      time read_time;
129      time write_time;
130      int unsigned read_count;
131      int unsigned write_count;
132    } access_t;
133
134  endclass
```

这个类主要有两个用途，一是设置 override 信息，二是提供了一些 typedef，进行重命名。如果在 uvm_resource_db::set 时指定了 accessor，那么对于每一个 accessor，都有一条 uvm_resource_types::access_t 的记录存放在 uvm_resource_base 的 access 联合数组中：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base

209    uvm_resource_types::access_t access[string];
```

这条记录记载了对于这个资源的访问的相关信息。

回到 record_write_access。525 到 528 行查看 access 中是否有 accessor 的记录，如果没有插入，否则把这条记录提取出来。528 行用到了 init_access_record：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base
函数/任务：init_access_record

571    function void init_access_record (inout uvm_resource_types::access_t access_record);
572      access_record.read_time = 0;
573      access_record.write_time = 0;
574      access_record.read_count = 0;
575      access_record.write_count = 0;
576    endfunction
577
578  endclass
```

函数比较简单，只是把 access_t 的各字段均初始化为 0。

529 到 530 行把关于此次 write 操作的相关信息存储在 access_record 中，531 行则更新 acccess 中的相关记录。可以看出来，access 数组的索引是 accessor 的 get_full_name 的返回值，而内容则表明了此 accessor 对此资源的访问情况。

可见，auditing 功能只是 UVM 提供的一个用于记录资源访问时间，访问次数的

机制，其对于整个 resource_db 机制并无太大影响，它不属于 resource_db 机制的核心功能，甚至可以直接跳过去。

回到 write 函数，1584 行是这个函数的核心，它把输入的 val 值赋值给了 val。从而经过 write 函数后，这个 uvm_resource#(T)的实例中 val 值被设置了，再加上此前的 scope 和 precedence 在 new 的时候被设置，加入 uvm_resource_pool 的条件都满足了，因此可以加入到 uvm_resource_pool 中了。

16.2.4. uvm_resource_pool 的 set 函数

uvm_resource#(type T)的 set 函数如下：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：set

1454    function void set();
1455      uvm_resource_pool rp = uvm_resource_pool::get();
1456      rp.set(this);
1457    endfunction
```

1455 行得到全局的 uvm_resource_pool 的指针，1456 行调用 uvm_resource_pool 的 set 函数，传入的参数是 this：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：set

 720    function void set (uvm_resource_base rsrc,
 721                       uvm_resource_types::override_t override = 0);
 722
 723      uvm_resource_types::rsrc_q_t rq;
 724      string name;
 725      uvm_resource_base type_handle;
 726
 727      // If resource handle is null then there is nothing to do.
 728      if(rsrc == null)
 729        return;
 730
 731      // insert into the name map.  Resources with empty names are
 732      // anonymous resources and are not entered into the name map
 733      name = rsrc.get_name();
 734      if(name != "") begin
 735        if(rtab.exists(name))
 736          rq = rtab[name];
```

```
737         else
738           rq = new();
739
740         // Insert the resource into the queue associated with its name.
741         // If we are doing a name override then insert it in the front of
742         // the queue, otherwise insert it in the back.
743         if(override & uvm_resource_types::NAME_OVERRIDE)
744           rq.push_front(rsrc);
745         else
746           rq.push_back(rsrc);
747
748         rtab[name] = rq;
749       end
750
```

723 行声明了 uvm_resource_types::rsrc_q_t 型的变量 rq。uvm_resource_types::rsrc_q_t 的定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_types

119   typedef uvm_queue#(uvm_resource_base) rsrc_q_t;
```

意味着 rq 是一个队列，其中存放的内容是 uvm_resource_base 类型的。

728 行判断 rsrc 是否为 null。rsrc 是输入的参数，根据 uvm_resource#(T)的 set 函数，这个 rsrc 指的是要加入的资源的指针。

733 行到 749 行把输入的 rsrc 插入到 rtab 中。这里要注意的是，rsrc 是不能直接插入到 rtab 中的，因为 rtab 中的每一条记录是一个队列，rsrc 只能插入到某条记录的队列中。另外，734 行会判断 rsrc 的名字是否为空，如果为空，那么就不插入。rtab 中记录的索引是字符串，即资源的名字。733 行得到资源的名字，735 行看看是否在 rtab 中有了索引为为个名字的记录，如果有，那么把这条记录对应的队列提取出来，否则就新建一个队列。经过 735 行到 738 行之后，rq 就会指向一个实际的队列，而不是为 null。

743 到 746 行根据 override 的信息，决定把 rsrc 是插入到 rq 所指向队列的最前面还是最后面。这里出现了 uvm_resource_types::NAME_OVERRIDE，其定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_types

115   typedef enum override_t { TYPE_OVERRIDE = 2'b01,
116                             NAME_OVERRIDE = 2'b10 } override_e;
```

默认情况下，set 函数的 override 参数的值为 0，所以 743 行的条件不满足，也即不进行名字替换，所以把 rsrc 插入到 rq 的最后面。假设还有一个 rsrc1，在之前已经放入了 rq 中（即 735 行的条件满足），这个 rsrc1 的 scope 与 rsrc 的 scope 是一样的。默认情况下，rsrc 是在 rsrc1 后面，当依据 scope 进行查找时，会先找到 rsrc1。

假如设置了 NAME_OVERRIDE，即输入的 override 的值为 2’b10，那么 rsrc 会放在 rsrc1 前面，这样当依据 scope 进行查找时，会首先找到 rsrc。

748 行把 rq 插入到 rtab 中。

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：set

 751       // insert into the type map
 752       type_handle = rsrc.get_type_handle();
 753       if(ttab.exists(type_handle))
 754          rq = ttab[type_handle];
 755       else
 756          rq = new();
 757
 758       // insert the resource into the queue associated with its type.  If
 759       // we are doing a type override then insert it in the front of the
 760       // queue, otherwise insert it in the back of the queue.
 761       if(override & uvm_resource_types::TYPE_OVERRIDE)
 762          rq.push_front(rsrc);
 763       else
 764          rq.push_back(rsrc);
 765       ttab[type_handle] = rq;
 766
 767       //optimization for name lookups. Since most environments never
 768       //use wildcarded names, don't want to incurr a search penalty
 769       //unless a wildcarded name has been used.
 770       if(rsrc.m_is_regex_name)
 771          m_has_wildcard_names = 1;
 772    endfunction
```

752 到 765 行把 rsrc 插入到 ttab 中。同 rtab 一样，并不是直接把 rsrc 插入，而是先把 rsrc 放入一个队列中，然后再把这个队列插入到 ttab 中。与 rsrc 插入到 rtab 中不同的是，所有的 rsrc 一定会插入到 ttab 中，但是并不是所有的 rsrc 都会插入到 rtab 中。当 rsrc 的 get_name 返回为空的时候，rsrc 就不会插入到 rtab 中，但是依然会插入到 ttab 中。

752 行调用了 get_type_handle 函数，这是在 uvm_resource#(type T)中定义的函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：get_type_handle

1434    function uvm_resource_base get_type_handle();
1435       return get_type();
1436    endfunction
```

直接调用了 get_type 函数，也是在 uvm_resource#(type T)中定义的：

```
文件：src/base/uvm_resource.svh
```

```
类：uvm_resource#(type T=int)
函数/任务：get_type

1421    static function this_type get_type();
1422      if(my_type == null)
1423        my_type = new();
1424      return my_type;
1425    endfunction
```

这个函数的主要意思就是查看 my_type 是否为 null，如果 是 null，那么就实例化，否则直接返回 null。my_type 是 uvm_resource#(type T)的成员变量：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)

1395    typedef uvm_resource#(T) this_type;
1396
1397    // singleton handle that represents the type of this resource
1398    static this_type my_type = get_type();
```

这是一个静态成员变量，因此是与类 uvm_resource#(type T)对应的，一个 T 对应一个 uvm_resource#(type T)类，从而对应一个 my_type。也就是说，my_type 从某种意义上来说是代表了 T 这个类型。因此，uvm_resource_pool 的 set 函数的 752 行通过 get_type_handle 函数得到了代表 T 类型的指针，753 行查看 ttab 中是否有此指针为索引的记录。因此，ttab 中所有记录的索引是类型的指针。

753 到 756 行得到要插入 rsrc 的队列的指针，761 到 764 行根据 override 的值来决定是把 rsrc 是放在队列的前面还是后面。这个与 rsrc 插入到 rtab 中类似，不多做介绍。

765 行把队列插入到 ttab 中。770 到 771 行根据 rsrc 的名字中是否有通配符来给 m_has_wildcard_names 赋值。m_has_wildcard_names 主要是用于后面的读取操作中，关于其作用后面会提到。

16.2.5. uvm_config_db 的 set_default 函数

除了上面提到的 set 函数外，uvm_config_db#(T)类中还有其它的函数用于向 uvm_resource_pool 中写入资源。本节及接下来的几小节将会介绍这几个函数。本节先介绍 set_default 函数，其定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set_default
```

```
 98    static function rsrc_t set_default(string scope, string name);
 99
100      rsrc_t r;
101
102      r = new(name, scope);
103      r.set();
104      return r;
105    endfunction
```

这个函数与 set 函数几乎完全一样，只是它没有调用 uvm_resource#(T)的 write 函数。因此，经过 set_default 函数之后，如果 name 参数不为空，那么 uvm_resource_pool 的 rtab 和 ttab 相应的队列中将各自插入一条记录，否则将会只在 ttab 的相应队列中插入一条记录。插入的这条记录的 val 值是默认的值，而没有经过设置，也即 default 值，这就是 set_default 名称的由来。

16.2.6. uvm_config_db 的 set_anonymous 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set_anonymous

155    static function void set_anonymous(input string scope,
156                                       T val, input uvm_object accessor = null);
157
158      rsrc_t rsrc = new("", scope);
159      rsrc.write(val, accessor);
160      rsrc.set();
161
162      if(uvm_resource_db_options::is_tracing())
163        m_show_msg("RSRCDB/SETANON","Resource", "set", scope, "", accessor, rsrc);
164    endfunction
```

这个函数与 set 函数极其相似，唯一的区别在于因为在实例化 rsrc 时，输入的名字为空，所以它只会向 uvm_resource_pool 的 ttab 中插入一条记录，而不会向 rtab 中插入记录。

16.2.7. uvm_config_db 的 set_override 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set_override

173    static function void set_override(input string scope, input string name,
174                                      T val, uvm_object accessor = null);
175      rsrc_t rsrc = new(name, scope);
176      rsrc.write(val, accessor);
177      rsrc.set_override();
178
179      if(uvm_resource_db_options::is_tracing())
180        m_show_msg("RSRCDB/SETOVRD", "Resource","set", scope, name, accessor, rsrc);
181    endfunction
```

与 set 函数也比较像，唯一区别是 177 行调用的是 rsrc 的 set_override 函数，而不是 set 函数。uvm_resource#(T)的 set_override 函数定义如下：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：set_override

1469    function void set_override(uvm_resource_types::override_t override = 2'b11);
1470      uvm_resource_pool rp = uvm_resource_pool::get();
1471      rp.set(this, override);
1472    endfunction
```

这里会直接调用 uvm_resource_pool 的 set 函数。注意的是，这里传入的 override 数值是 2’b11，uvm_resource_pool 的 set 函数的 743 行 761 行的条件都满足，即同时进行 NAME_OVERRIDE 和 TYPE_OVERRIDE。

总结下，如果 name 参数不为空，那么 uvm_resource_pool 的 rtab 和 ttab 相应的队列中将各自插入一条记录，否则将会只在 ttab 的相应队列中插入一条记录。插入的这条记录是放在相应队列的最前端。

16.2.8. uvm_config_db 的 set_override_type 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
```

```
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set_override_type

193    static function void set_override_type(input string scope, input string name,
194                                           T val, uvm_object accessor = null);
195      rsrc_t rsrc = new(name, scope);
196      rsrc.write(val, accessor);
197      rsrc.set_override(uvm_resource_types::TYPE_OVERRIDE);
198
199      if(uvm_resource_db_options::is_tracing())
200        m_show_msg("RSRCDB/SETOVRDTYP","Resource", "set", scope, name, accessor, rsr
c);
201    endfunction
```

函数与 set_override 类似，只是 197 行在调用 uvm_resource#(T)的 set_override 函数时，传入了 TYPE_OVERRIDE 参数，从而 uvm_resource#(T)的 set_override 在调用 uvm_resource_pool 的 set 函数时会入 TYPE_OVERRIDE 参数。

函数的执行结果就是，如果 name 参数不为空，那么 uvm_resource_pool 的 rtab 和 ttab 相应的队列中将各自插入一条记录，否则将会只在 ttab 的相应队列中插入一条记录。ttab 中插入的这条记录是放在相应队列的最前端，而 rtab 中插入的这条记录（假如插入了）是放在相应队列的最后端。

16.2.9. uvm_config_db 的 set_override_name 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：set_override_name

211    static function void set_override_name(input string scope, input string name,
212                                           T val, uvm_object accessor = null);
213      rsrc_t rsrc = new(name, scope);
214      rsrc.write(val, accessor);
215      rsrc.set_override(uvm_resource_types::NAME_OVERRIDE);
216
217      if(uvm_resource_db_options::is_tracing())
218        m_show_msg("RSRCDB/SETOVRDNAM","Resource", "set", scope, name, accessor, rsr
c);
219    endfunction
```

函数与 set_override_type 类似，只是 215 行在调用 uvm_resource#(T)的 set_override 函数时，传入了 NAME_OVERRIDE 参数，从而 uvm_resource#(T)的 set_override 在调用 uvm_resource_pool 的 set 函数时会入 NAME_OVERRIDE 参数。

函数的执行结果就是，如果 name 参数不为空，那么 uvm_resource_pool 的 rtab 和 ttab 相应的队列中将各自插入一条记录，否则将会只在 ttab 的相应队列中插入一条记录。ttab 中插入的这条记录是放在相应队列的最后端，而 rtab 中插入的这条记录（假如插入了）是放在相应队列的最前端。

16.2.10. uvm_config_db 的 write_by_name 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：write_by_name

281    static function bit write_by_name(input string scope, input string name,
282                                      T val, input uvm_object accessor = null);
283
284      rsrc_t rsrc = get_by_name(scope, name);
285
286      if(uvm_resource_db_options::is_tracing())
287        m_show_msg("RSRCDB/WR","Resource", "written", scope, name, accessor, rsrc);
288
289      if(rsrc == null)
290        return 0;
291
292      rsrc.write(val, accessor);
293
294      return 1;
295
296    endfunction
```

284 行调用 get_by_name 函数，函数的定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：get_by_name

 85    static function rsrc_t get_by_name(string scope,
 86                                       string name,
 87                                       bit rpterr=1);
 88
 89      return rsrc_t::get_by_name(scope, name, rpterr);
 90    endfunction
```

它会直接调用 uvm_resource#(T)的 get_by_name 函数：

```
文件：src/base/uvm_resource.svh
```

```
类：uvm_resource#(type T=int)
函数/任务：get_by_name

1485    static function this_type get_by_name(string scope,
1486                                          string name,
1487                                          bit rpterr = 1);
1488
1489      uvm_resource_pool rp = uvm_resource_pool::get();
1490      uvm_resource_base rsrc_base;
1491      this_type rsrc;
1492      string msg;
1493
1494      rsrc_base = rp.get_by_name(scope, name, my_type, rpterr);
1495      if(rsrc_base == null)
1496        return null;
1497
1498      if(!$cast(rsrc, rsrc_base)) begin
1499        if(rpterr) begin
1500          $sformat(msg, "Resource with name %s in scope %s has incorrect type", name, s
cope);
1501          `uvm_warning("RSRCTYPE", msg);
1502        end
1503        return null;
1504      end
1505
1506      return rsrc;
1507
1508    endfunction
```

1494 行直接调用 uvm_resource_pool 的 get_by_name 函数。这里出现了 rpterr 参数，是用于在出错的时候是否提示错误。默认情况下，rpterr 的值为 1，也就意味着会报告出错信息。uvm_resource_pool 的 get_by_name 函数如下：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：get_by_name

 981    function uvm_resource_base get_by_name(string scope = "",
 982                                           string name,
 983                                           uvm_resource_base type_handle,
 984                                           bit rpterr = 1);
 985
 986      uvm_resource_types::rsrc_q_t q;
 987      uvm_resource_base rsrc;
 988
 989      q = lookup_name(scope, name, type_handle, rpterr);
 990
 991      if(q.size() == 0) begin
 992        push_get_record(name, scope, null);
 993        return null;
 994      end
```

```
 995
 996       rsrc = get_highest_precedence(q);
 997       push_get_record(name, scope, rsrc);
 998       return rsrc;
 999
1000    endfunction
```

989 行调用 lookup_name 函数，其定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：lookup_name

 877    function uvm_resource_types::rsrc_q_t lookup_name(string scope = "",
 878                                                      string name,
 879                                                       uvm_resource_base type_handle
= null,
 880                                                      bit rpterr = 1);
 881       uvm_resource_types::rsrc_q_t rq;
 882       uvm_resource_types::rsrc_q_t q = new();
 883       uvm_resource_base rsrc;
 884       uvm_resource_base r;
 885
 886       // resources with empty names are anonymous and do not exist in the name map
 887       if(name == "")
 888         return q;
 889
 890       // Does an entry in the name map exist with the specified name?
 891       // If not, then we're done
 892       if((rpterr && !spell_check(name)) || (!rpterr && !rtab.exists(name))) begin
 893         return q;
 894       end
 895
 896       rsrc = null;
 897       rq = rtab[name];
 898       for(int i=0; i<rq.size(); ++i) begin
 899         r = rq.get(i);
 900         // does the scope match?
 901         if(r.match_scope(scope) &&
 902             // does the type match?
 903             ((type_handle == null) || (r.get_type_handle() == type_handle)))
 904           q.push_back(r);
 905       end
 906
 907       return q;
 908
 909    endfunction
```

887 行判断 name 是否为空，name 为空的记录是不可能在 rtab 中查找到的。因此 888 行会直接返回一个空的 q。

892 行调用了 spell_check 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：spell_check

 693    function bit spell_check(string s);
 694      return uvm_spell_chkr#(uvm_resource_types::rsrc_q_t)::check(rtab, s);
 695    endfunction
```

这个函数会调用 uvm_spell_chkr 的 check 函数来判断输入的名字是否合法：

```
文件：src/base/uvm_spell_chkr.svh
类：uvm_spell_chkr #(type T=int);
函数/任务：check

 67   static function bit check (tab_t strtab, string s);
 68
 69     string key;
 70     int distance;
 71     int unsigned min;
 72     string min_key[$];
 73
 74     if(strtab.exists(s)) begin
 75       return 1;
 76     end
 77
 78     min = max;
 79     foreach(strtab[key]) begin
 80       distance = levenshtein_distance(key, s);
 81
 82       // A distance < 0 means either key, s, or both are empty.  This
 83       // should never happen here but we check for that condition just
 84       // in case.
 85       if(distance < 0)
 86         continue;
 87
 88       if(distance < min) begin
 89         // set a new minimum.  Clean out the queue since previous
 90         // alternatives are now invalidated.
 91         min = distance;
 92         min_key.delete();
 93         min_key.push_back(key);
 94         continue;
 95       end
 96
 97       if(distance == min) begin
 98         min_key.push_back(key);
 99       end
100
101     end
102
103     $display("%s not located", s);
104
```

```
105       // if (min == max) then the string table is empty
106       if(min == max) begin
107         $display("   no alternatives to suggest");
108         return 0;
109       end
110
111       // dump all the alternatives with the minimum distance
112       foreach(min_key[i]) begin
113         $display("   did you mean %s?", min_key[i]);
114       end
115
116       return 0;
117
118     endfunction
```

整个函数相对来说比较简单，74 行查看输入的名字是否与 rtab 中的某条记录的索引完全一样，如果完全一样，那么直接返回 1。否则接下来的代码将会查看是否某条记录的索引与输入的名字差几个字符，但是无论差几个字符，返回值都将为 0。这里不详细介绍这个函数。

回到 lookup_name 函数，如果 name 不在 rtab 的索引中，那么 892 行的条件将会满足，因此 893 行会直接返回一个空的队列。

897 行取出 rtab 中对应 name 的那个队列，898 到 905 行则是遍历这个队列，把其中符合条件的记录放入 q 中。这里的条件有两种：一是 scope 完全的匹配，并且 type_handle 为 null；二是 scope 完全的匹配，并且记录的 type_handle 与输入的 type_handle 完全一样。也即表示必须是所需要类型的记录。

907 行返回 q。因此，lookup_name 函数的结果就是返回 rtab 中索引为 name 的队列中 scope 与输入的 scope 吻合，type_handle 与输入的 type_handle 吻合的记录，由于记录可能有多个，因此这些记录被放入一个队列中返回。

回到 uvm_resource_pool 的 get_by_name 函数，991 行判断返回的队列是否为空，并做相关处理。992 行用到了 push_get_record 函数，这个函数只是用于记录下对 uvm_resource_pool 的读取信息，对于 uvm_resource_pool 的整体功能实现并没有太大影响，这里不多做介绍。

996 行调用了 get_highest_precedence 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：get_highest_precedence

 918     function uvm_resource_base get_highest_precedence(ref uvm_resource_types::rsrc_q_t q);
 919
 920       uvm_resource_base rsrc;
 921       uvm_resource_base r;
 922       int unsigned i;
 923       int unsigned prec;
```

```
924
925       if(q.size() == 0)
926         return null;
927
928       // get the first resources in the queue
929       rsrc = q.get(0);
930       prec = rsrc.precedence;
931
932       // start searching from the second resource
933       for(int i = 1; i < q.size(); ++i) begin
934         r = q.get(i);
935         if(r.precedence > prec) begin
936           rsrc = r;
937           prec = r.precedence;
938         end
939       end
940
941       return rsrc;
942
943     endfunction
```

函数比较简单，就是把输入的 q 中 precendence 值最高的记录返回。

998 行把 get_highest_precedence 的返回值返回。因此，uvm_resource_pool 的 get_by_name 函数返回的是 rtab 中索引为 name 的队列中 scope 与输入的 scope 吻合，type_handle 与输入的 type_handle 吻合的记录中 precendence 最高的那条记录。

回到 uvm_resource#(T)的 get_by_name 函数，1495 行用于检查 uvm_resource_pool 的 get_by_name 函数是否返回了一条非空的记录。1498 到 1504 行检查返回的记录是否是属于 this_type 类型的。由于在 uvm_resource_pool 的 lookup_name 中已经做了这种检查，因此这里的检查似有重复的嫌疑。这几行的核心在于 1498 行的 cast 语句，1506 行直接返回的 rsrc 就是在 cast 语句中实现的类型转换。

回到 uvm_resource_db 的 write_by_name 函数，284 行通过调用 get_by_name 函数得到了 rtab 中对应 scope 和 name 的记录。292 行把新的值写入到这条记录中。

因此，write_by_name 函数并不会在 rtab 和 ttab 的相应队列中新插入记录，而只是会更新记录。注意，这里的更新不只更新 rtab 中相应队列中的相应记录，同时也会更新 ttab 中相应队列中的相应记录。因为无论是 rtab 还是 ttab，其相应队列中的相应记录指向的都是同一条记录，记录的内容变了，看上去 rtab 和 ttab 的相应队列的相应记录同步改变了。

16.2.11. uvm_resource_db 的 write_by_type 函数

函数的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：write_by_type

310    static function bit write_by_type(input string scope,
311                                      input T val, input uvm_object accessor = null);
312
313      rsrc_t rsrc = get_by_type(scope);
314
315      if(uvm_resource_db_options::is_tracing())
316        m_show_msg("RSRCDB/WRTYP", "Resource","written", scope, "", accessor, rsrc);
317
318      if(rsrc == null)
319        return 0;
320
321      rsrc.write(val, accessor);
322
323      return 1;
324    endfunction
```

函数与 write_by_name 几乎一模一样，不同的是 313 行调用的是 get_by_type，而 write_by_name 中调用的则是 get_by_name。get_by_type 的定义如下：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：get_by_type

 74    static function rsrc_t get_by_type(string scope);
 75      return rsrc_t::get_by_type(scope, rsrc_t::get_type());
 76    endfunction
```

这里直接调用的是 uvm_resource#(T)的 get_by_type 函数，其定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource#(type T=uvm_object)
函数/任务：get_by_type

1517    static function this_type get_by_type(string scope = "",
1518                                          uvm_resource_base type_handle);
1519
1520      uvm_resource_pool rp = uvm_resource_pool::get();
1521      uvm_resource_base rsrc_base;
1522      this_type rsrc;
1523      string msg;
1524
```

```
1525        if(type_handle == null)
1526          return null;
1527
1528        rsrc_base = rp.get_by_type(scope, type_handle);
1529        if(rsrc_base == null)
1530          return null;
1531
1532        if(!$cast(rsrc, rsrc_base)) begin
1533          $sformat(msg, "Resource with specified type handle in scope %s was not located", s
cope);
1534          `uvm_warning("RSRCNF", msg);
1535          return null;
1536        end
1537
1538        return rsrc;
1539
1540      endfunction
```

这个函数与 uvm_resource#(T)的 get_by_name 几乎完全一样，不同的是 1528 行调用的是 uvm_resource_pool 的 get_by_type，而 get_by_name 中则是调用的是 uvm_resource_pool 的 get_by_name。uvm_resource_pool 的 get_by_type 定义如下：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：get_by_type

1037    function uvm_resource_base get_by_type(string scope = "",
1038                                           uvm_resource_base type_handle);
1039
1040      uvm_resource_types::rsrc_q_t q;
1041      uvm_resource_base rsrc;
1042
1043      q = lookup_type(scope, type_handle);
1044
1045      if(q.size() == 0) begin
1046        push_get_record("<type>", scope, null);
1047        return null;
1048      end
1049
1050      rsrc = q.get(0);
1051      push_get_record("<type>", scope, rsrc);
1052      return rsrc;
1053
1054    endfunction
```

这个函数与 uvm_resource_pool 的 get_by_name 几乎一样，不同的是 1043 行调用的是 lookup_type：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：lookup_type
```

```
1009    function uvm_resource_types::rsrc_q_t lookup_type(string scope = "",
1010                                                      uvm_resource_base type_handl
e);
1011
1012      uvm_resource_types::rsrc_q_t q = new();
1013      uvm_resource_types::rsrc_q_t rq;
1014      uvm_resource_base r;
1015      int unsigned i;
1016
1017      if(type_handle == null || !ttab.exists(type_handle)) begin
1018        return q;
1019      end
1020
1021      rq = ttab[type_handle];
1022      for(int i = 0; i < rq.size(); ++i) begin
1023        r = rq.get(i);
1024        if(r.match_scope(scope))
1025          q.push_back(r);
1026      end
1027
1028      return q;
1029
1030    endfunction
```

1017 行检查输入的 type_handle 是否为 null，以及 ttab 中是否存在索引为 type_handle 的一条记录。1021 到 1028 行把 ttab 中索引为 type_handle 的记录所对应的队列中 scope 与输入的 scope 一致的记录取出，放入新建的队列中，并且返回。这一点与 lookup_name 一致。

因此 uvm_resource_db 的 write_by_type 函数与 write_by_name 函数几乎是完全一样，区别就是一个根据输入的参数是 scope 查找，而另外一个则根据输入的 scope 和 name 查找。这两个函数都是只更新 rtab 和 ttab 中相应队列中的记录，而不会向相应队列中插入一条新的记录。

16.3. 资源的读出

16.3.1. read_by_name 和 read_by_type 函数

resource_db 机制中，资源的读出是由 uvm_resource_db 类中的 read_by_name 和 read_by_type 完成的。read_by_name 的定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：read_by_name

227   static function bit read_by_name(input string scope,
228                                    input string name,
229                                    inout T val, input uvm_object accessor = null);
230
231     rsrc_t rsrc = get_by_name(scope, name);
232
233     if(uvm_resource_db_options::is_tracing())
234       m_show_msg("RSRCDB/RDBYNAM","Resource", "read", scope, name, accessor, rsrc);
235
236     if(rsrc == null)
237       return 0;
238
239     val = rsrc.read(accessor);
240
241     return 1;
242
243   endfunction
```

而 read_by_type 的定义为：

```
文件：src/base/uvm_resource_db.svh
类：uvm_resource_db #(type T=uvm_object)
函数/任务：read_by_type

251   static function bit read_by_type(input string scope,
252                                    inout T val,
253                                    input uvm_object accessor = null);
254
255     rsrc_t rsrc = get_by_type(scope);
256
257     if(uvm_resource_db_options::is_tracing())
258       m_show_msg("RSRCDB/RDBYTYP", "Resource","read", scope, "", accessor, rsrc);
259
```

```
260       if(rsrc == null)
261         return 0;
262
263       val = rsrc.read(accessor);
264
265       return 1;
266
267     endfunction
```

这两个函数比较类似，先通过 get_by_type 或者 get_by_name 得到要读取的资源的指针，之后调用 uvm_resource#(T)的 read 函数。

16.3.2. uvm_resource#(T)的 read 函数

函数的定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource#(type T=int)
函数/任务：read

1559    function T read(uvm_object accessor = null);
1560      record_read_access(accessor);
1561      return val;
1562    endfunction
```

函数相当简单，先调用 record_read_access 函数记录读取信息，之后直接返回 val 值。

16.4. uvm_config_db 类对 resource_db 机制的扩展

16.4.1. uvm_config_db 的 set 函数

uvm_config_db 是从 uvm_resource_db 派生而来的，它对 uvm_resource_db 的一些功能进行了扩展，这种扩展主要体现在对资源的写入和读取上。其实在资源的写

入操作上，它重载了 uvm_resource_db 的 set 函数，而在资源的读取操作上，它新建了一个称为 get 的函数。本节先介绍 set 函数，下一节介绍 get 函数。set 函数的定义为：

```
文件：src/base/uvm_config_db.svh
类：uvm_config_db#(type T=int)
函数/任务：set

147    static function void set(uvm_component cntxt,
148                             string inst_name,
149                             string field_name,
150                             T value);
151
152      uvm_root top;
153      uvm_phase curr_phase;
154      uvm_resource#(T) r;
155      bit exists = 0;
156
157      //take care of random stability during allocation
158      process p = process::self();
159      string rstate = p.get_randstate();
160      top = uvm_root::get();
161      curr_phase = top.m_current_phase;
162
163      if(cntxt == null)
164        cntxt = top;
165      if(inst_name == "")
166        inst_name = cntxt.get_full_name();
167      else if(cntxt.get_full_name() != "")
168        inst_name = {cntxt.get_full_name(), ".", inst_name};
169
170      r = m_get_resource_match(cntxt, field_name, inst_name);
171
172      if(r == null) begin
173        uvm_pool#(string, uvm_resource#(T)) pool = new;
174        string key = {inst_name,field_name};
175        m_rsc[cntxt] = pool;
176        r = new(field_name, inst_name);
177        pool.add(key, r);
178      end
179      else begin
180        exists = 1;
181      end
182
183      if(curr_phase != null && curr_phase.get_name() == "build")
184        r.precedence -= cntxt.get_depth();
185
186      r.write(value, cntxt);
187
188      if(exists) begin
189        uvm_resource_pool rp = uvm_resource_pool::get();
```

```
190         rp.set_priority_name(r, uvm_resource_types::PRI_HIGH);
191       end
192       else begin
193         //Doesn't exist yet, so put it in resource db at the head.
194         r.set_override();
195       end
196
197       //trigger any waiters
198       if(m_waiters.exists(field_name)) begin
199         m_uvm_waiter w;
200         for(int i=0; i<m_waiters[field_name].size(); ++i) begin
201           w = m_waiters[field_name].get(i);
202           if(uvm_re_match(inst_name,w.inst_name) == 0)
203               ->w.trigger;
204         end
205       end
206
207       p.set_randstate(rstate);
208
209       if(uvm_resource_db_options::is_tracing())
210         m_show_msg("CFGDB/SET", "Configuration","set", inst_name, field_name, cntxt, r);
211     endfunction
```

为了便于理解，以一个例子来讲述这个函数，假设我们在某个 case 的 build_phase 中如下下语句：

```
uvm_config_db#(top_config)::set(this, “env.agent.drv”, “top_config”, top_cfg)
```

我们看这句话是如何执行的。它首先定义了一个称为 uvm_config_db#(top_config) 的类，这个类内部有一个静态联合数组 m_rsc，此数组的索引是 uvm_component，内容是 uvm_pool：

```
文件：src/base/uvm_config_db.svh
类：uvm_config_db#(type T=int)

 65    static uvm_pool#(string,uvm_resource#(T)) m_rsc[uvm_component];
```

关于 uvm_pool，已经多次提到过了，它的本质是一个联合数组，在上面的例子中，联合数组的索引是 string 类型的，而内容则是 uvm_resource#(T)。

set 函数的 159 行和 207 行出现了 get_randstate 和 set_randstate，这是两个 systemverilog 的函数，用在这里是为了保证在插入操作时，保证随机数产生器的稳定性。

163 行到 168 行是为了给 inst_name 赋值。经过这几行代码后，inst_name 的值变为了”uvm_test_top.env.agent.drv”。

170 行调用 m_get_resourcce_match 函数：

```
文件：src/base/uvm_config_db.svh
类：uvm_config_db#(type T=int)
```

```
函数/任务：m_get_resource_match

282    static function uvm_resource#(T) m_get_resource_match(uvm_component cntxt,
283              string field_name, string inst_name);
284      uvm_pool#(string,uvm_resource#(T)) pool;
285      string lookup;
286
287      if(!m_rsc.exists(cntxt)) begin
288        return null;
289      end
290
291      lookup = {inst_name,field_name};
292      pool = m_rsc[cntxt];
293
294      if(!pool.exists(lookup)) return null;
295
296      return pool.get(lookup);
297    endfunction
```

287 行检查 m_rsc 中是否有对应 cntxt 的记录。这里我们输入的 cntxt 是 this，即是指的此 case 实例的指针。由于这是我们第一次调用 set 函数，所以 m_rsc 中是空的，所以 288 行会直接返回。

回到 set 函数，172 行发现 m_rsc 中没有对应 cntxt 的记录，所以 173 行到 177 行会向 m_rsc 中插入一条新的记录。插入的记录的索引是 cntxt，而内容则是一个 pool，向这个 pool 中插入了一条记录，记录的索引是”uvm_test_done.env.agent.drvtop_config”，而内容则是 uvm_resource#(top_config)的一个实例的指针。需要注意的是，此时插入的紧紧是这条记录的指针，但是这条记录的值并没有写入。真正的值写入到记录中是在 186 行完成的。另外这条记录接下来也会被插入到 uvm_resource_pool 中。183 和 184 行设置要插入记录的 precedence 信息。

188 行到 195 行则向 uvm_resource_pool 中插入记录。在我们的例子中，执行的是 194 行的分支，即调用 set_override 函数。前面分析这个函数的时候，介绍过其默认的参数下，将会实行 NAME_OVERRIDE 和 TYPE_OVERRIDE，即把这条记录插入到 uvm_resource_pool 的 rtab 和 ttab 相应对应的最前端。

198 到 205 行则用于释放所有等待此 set 函数完成的进程。

回过头来看一下在下面这种情况下：

```
uvm_config_db#(top_config)::set(this, “env.agent.drv”, “top_config”, top_cfg)
uvm_config_db#(top_config)::set(this, “env.agent.drv”, “top_config”, new_top_cfg)
```

当第二次执行 set 的时候，m_get_resourcce_match 函数将会返回一个之前插入到 m_rsc 中的记录，所以 set 函数会执行 190 行的分支，这里用到了 set_prority_name 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：set_prority_name

1251    function void set_priority_name(uvm_resource_base rsrc,
1252                                    uvm_resource_types::priority_e pri);
1253
1254      string name;
1255      string msg;
1256      uvm_resource_types::rsrc_q_t q;
1257
1258      if(rsrc == null) begin
1259        uvm_report_warning("NULLRASRC", "attempting to change the serach priority of a
null resource");
1260        return;
1261      end
1262
1263      name = rsrc.get_name();
1264      if(!rtab.exists(name)) begin
1265        $sformat(msg, "Resrouce named %s not found in name map; cannot change its sear
ch priority", name);
1266        uvm_report_error("RNFNAME", msg);
1267        return;
1268      end
1269
1270      q = rtab[name];
1271      set_priority_queue(rsrc, q, pri);
1272
1273    endfunction
```

1258 到 1268 行都是检查输入参数的有效性。函数的核心在于 1271 行，这里调用了 set_priority_queue 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：set_priority_queue

1184    local function void set_priority_queue(uvm_resource_base rsrc,
1185                                           ref uvm_resource_types::rsrc_q_t q,
1186                                           uvm_resource_types::priority_e pri);
1187
1188      uvm_resource_base r;
1189      int unsigned i;
1190
1191      string msg;
1192      string name = rsrc.get_name();
1193
1194      for(i = 0; i < q.size(); i++) begin
1195        r = q.get(i);
1196        if(r == rsrc) break;
1197      end
```

```
1198
1199        if(r != rsrc) begin
1200          $sformat(msg, "Handle for resource named %s is not in the name name; cannot cha
nge its priority", name);
1201          uvm_report_error("NORSRC", msg);
1202          return;
1203        end
1204
1205        q.delete(i);
1206
1207        case(pri)
1208          uvm_resource_types::PRI_HIGH:  q.push_front(rsrc);
1209          uvm_resource_types::PRI_LOW:    q.push_back(rsrc);
1210        endcase
1211
1212      endfunction
```

1194 行到 1197 行得到 r 在 q 中的位置。r 是我们已经插入到 uvm_resource_pool 中的记录，而 q 则是 rtab 中对应名字为”top_config”的队列。1199 到 1203 行检查是否在 q 中找到了 r。1205 行从 q 中删除 r，这一句是为了 1208 和 1209 行做准备的。假如这里不删除，那么 1208 或 1209 任何一句都将会向 q 中插入一个 r，这样 q 中就会有两个 r。1208 和 1209 行则根据输入的参数决定是把 r 放在 q 的最前端还是最后端。在我们的例子中，输入的参数是 PRI_HIGH，因此会放在最前端。

16.4.2. uvm_config_db 的 get 函数

本节依然以一个例子来进行讲解，假设在 case 的 build_phase 中执行了如下语句：

```
uvm_config_db#(top_config)::set(this, “env.agent.drv”, “top_config”, top_cfg)
```

而在 env.agent.drv 的 build_phase 中有如下语句：

```
uvm_config_db#(top_config)::get(this, “”, “top_config”, p_top_cfg)
```

get 函数的定义如下：

```
文件：src/base/uvm_config_db.svh
类：uvm_config_db#(type T=int)
函数/任务：get

 85    static function bit get(uvm_component cntxt,
 86                            string inst_name,
 87                            string field_name,
 88                            inout T value);
 89  //TBD: add file/line
 90      int unsigned p=0;
```

```
 91       uvm_resource#(T) r, rt;
 92       uvm_resource_pool rp = uvm_resource_pool::get();
 93       uvm_resource_types::rsrc_q_t rq;
 94
 95       if(cntxt == null)
 96         cntxt = uvm_root::get();
 97       if(inst_name == "")
 98         inst_name = cntxt.get_full_name();
 99       else if(cntxt.get_full_name() != "")
100         inst_name = {cntxt.get_full_name(), ".", inst_name};
101
102       rq = rp.lookup_regex_names(inst_name, field_name, uvm_resource#(T)::get_type());
103       r = uvm_resource#(T)::get_highest_precedence(rq);
104
105       if(uvm_resource_db_options::is_tracing())
106         m_show_msg("CFGDB/GET", "Configuration","read", inst_name, field_name, cntxt, r);
107
108       if(r == null)
109         return 0;
110
111       value = r.read(cntxt);
112
113       return 1;
114     endfunction
```

95 到 100 行为 inst_name 及 cntxt 赋值。经过这几行代码后，inst_name 的值变为了”uvm_test_done.env.agent.drv”。

102 行调用了 uvm_resource_pool 的 lookup_regex_names 函数，其定义为：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_pool
函数/任务：lookup_regex_names

1065     function uvm_resource_types::rsrc_q_t lookup_regex_names(string scope,
1066                                                              string name,
1067                                                              uvm_resource_base type
_handle = null);
1068
1069       uvm_resource_types::rsrc_q_t rq;
1070       uvm_resource_types::rsrc_q_t result_q;
1071       int unsigned i;
1072       uvm_resource_base r;
1073
1074       //For the simple case where no wildcard names exist, then we can
1075       //just return the queue associated with name.
1076       if(!m_has_wildcard_names) begin
1077         result_q = lookup_name(scope, name, type_handle, 0);
1078         return result_q;
1079       end
1080
1081       result_q = new();
```

```
1082
1083        foreach (rtab[re]) begin
1084          rq = rtab[re];
1085          for(i = 0; i < rq.size(); i++) begin
1086            r = rq.get(i);
1087            if(uvm_re_match(uvm_glob_to_re(re),name) == 0)
1088              // does the scope match?
1089              if(r.match_scope(scope) &&
1090                // does the type match?
1091                ((type_handle == null) || (r.get_type_handle() == type_handle)))
1092                result_q.push_back(r);
1093          end
1094        end
1095        return result_q;
1096      endfunction
```

这段代码的意思就是从 rtab 中查找是否与输入的 name 对应的记录。这里的支持正则表达式。不过，一般说来，无论是在 get 还是在 set 时，都不会在其第三个字段中使用正则表达式，因此 lookup_regex_names 函数 1076 行的条件满足，1078 行会直接返回。这个返回值是查找到的所有的名字为”top_config”，scope 为”uvm_test_done.env.agent.drv”，且类型为 uvm_resource#(top_config)的记录，这些记录被放入一个队列中。

get 函数的 103 行从返回的队列中查找 precendence 值最高的一条记录，111 行得到这条记录的值。

在介绍 lookup_regex_names 函数时提到了正则表达式的问题。我们一般不会在 set 或者 get 时在第三个字段中使用正则表达式，但是我们经常会在 set 的第二个字段中使用通配符：

```
uvm_config_db#(top_config)::set(this, “*.agent.drv”, “top_config”, top_cfg)
```

这样，插入到 uvm_resource_pool 中的这条记录的 scope 就将会为”uvm_test_done.*. agent.drv”。那么在 get 的时候如何正确得到值呢？lookup_regex_names 函数在 1077 行调用了 lookup_name 函数，而在此函数中会调用 uvm_resource_base 的 match_scope 函数：

```
文件：src/base/uvm_resource.svh
类：uvm_resource_base
函数/任务：match_scope

 409        int err = uvm_re_match(scope, s);
 410        return (err == 0);
 411      endfunction
```

409 行的 uvm_re_match 就是支持正则表达式的比较，这里即是比较”uvm_test_done.*. agent.drv”与”uvm_test_done.env.agent.drv”是否匹配。在支持通配符的情况下，这两者是匹配的，从而 get 函数可以得到正确的值。

17. TLM1.0 源代码分析

相对于前面的 factory 机制，phase 机制或者 sequence 机制来说，TLM 的源代码都是相对简单许多，但是从另外一方面来说，它是比较繁琐的。

17.1. TLM 端口简介

17.1.1. UVM 中两类 TLM 端口

从本质上来说，UVM 中有两类 TLM 端口，一类是用于 driver 和 sequencer 之间连接的端口，一类是用于其它 component 之间连接的端口，如 monitor 和 scoreboard。

对于第一类来说，有下述三种端口，它们的原型分别是：

文件：src/tlm1/sqr_connections.svh

```
54 class uvm_seq_item_pull_port #(type REQ=int, type RSP=REQ)
55   extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));

73 class uvm_seq_item_pull_export #(type REQ=int, type RSP=REQ)
74   extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));
```

```
 89 class uvm_seq_item_pull_imp #(type REQ=int, type RSP=REQ, type IMP=int)
 90   extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));
```

而对于第二类来说，端口种类则多很多，以 uvm_analysis_port 来例：

```
文件：src/tlm1/uvm_analysis_port.svh

 56 class uvm_analysis_port # (type T = int)
 57   extends uvm_port_base # (uvm_tlm_if_base #(T,T));
```

可见，无论是哪一类 port，它们都是派生自 uvm_port_base 基类的。这两类 port 的最主要区别是，第一类 port 在派生自 uvm_port_base 时，传递的参数是 uvm_sqr_if_base 类型，而第二类参数则是 uvm_tlm_if_base 类型。

17.1.2. uvm_tlm_if_base

类的原型如下:

```
文件：src/tlm1/uvm_tlm_ifs.svh
类：uvm_tlm_if_base #(type T1=int, type T2=int)

 48 virtual class uvm_tlm_if_base #(type T1=int, type T2=int);
```

这个类没有派生自任何类，在类的内部，定义了三类接口：

第一类是阻塞性质的普通方法，put，get，peek，transport(T1,T2)。所谓 peek 的意思就是拿到了一笔 transaction，但是不 consuming 它。如果下次 peek 或者 get，可以得到同一笔 transaction。其中的 transport 是双向通信的。

```
文件：src/tlm1/uvm_tlm_ifs.svh
类：uvm_tlm_if_base #(type T1=int, type T2=int)

 59   virtual task put( input T1 t );
 60     uvm_report_error("put", `UVM_TASK_ERROR, UVM_NONE);
 61   endtask

 76   virtual task get( output T2 t );
 77     uvm_report_error("get", `UVM_TASK_ERROR, UVM_NONE);
 78   endtask

 94   virtual task peek( output T2 t );
 95     uvm_report_error("peek", `UVM_TASK_ERROR, UVM_NONE);
 96   endtask

185   virtual task transport( input T1 req , output T2 rsp );
186     uvm_report_error("transport", `UVM_TASK_ERROR, UVM_NONE);
```

```
187    endtask
```

第二类是非阻塞性质的普通方法，try_put，try_get，try_peek，nb_transport(T1,T2)。相应的还有 can_put，can_get，can_peek，can_put 意思就是在 put 或者 try_put 之间看看是不是对方已经做好了接收的准备了。can_get 意思就是看看对方是不是已经把 transaction 放在那里等待取走了。其中的 nb_transport 是双向通信的：

```
文件：src/tlm1/uvm_tlm_ifs.svh
类：uvm_tlm_if_base #(type T1=int, type T2=int)

108    virtual function bit try_put( input T1 t );
109      uvm_report_error("try_put", `UVM_FUNCTION_ERROR, UVM_NONE);
110      return 0;
111    endfunction

118    virtual function bit can_put();
119      uvm_report_error("can_put", `UVM_FUNCTION_ERROR, UVM_NONE);
120      return 0;
121    endfunction

134    virtual function bit try_get( output T2 t );
135      uvm_report_error("try_get", `UVM_FUNCTION_ERROR, UVM_NONE);
136      return 0;
137    endfunction

145    virtual function bit can_get();
146      uvm_report_error("can_get", `UVM_FUNCTION_ERROR, UVM_NONE);
147      return 0;
148    endfunction

162    virtual function bit try_peek( output T2 t );
163      uvm_report_error("try_peek", `UVM_FUNCTION_ERROR, UVM_NONE);
164      return 0;
165    endfunction

172    virtual function bit can_peek();
173      uvm_report_error("can_ppeek", `UVM_FUNCTION_ERROR, UVM_NONE);
174      return 0;
175    endfunction

200    virtual function bit nb_transport(input T1 req, output T2 rsp);
201      uvm_report_error("nb_transport", `UVM_FUNCTION_ERROR, UVM_NONE);
202      return 0;
203    endfunction
```

第三类是用于广播的 write，是属于 analysis 的，向所有连接的 port 写。这是一个 function，而不是 task。而前两类所有的都是 task：

```
文件：src/tlm1/uvm_tlm_ifs.svh
类：uvm_tlm_if_base #(type T1=int, type T2=int)
```

```
213    virtual function void write( input T1 t );
214      uvm_report_error("write", `UVM_FUNCTION_ERROR, UVM_NONE);
215    endfunction
```

这三类接口都是必须要要派生类里重新定义才能使用的，如果直接使用的话，那么系统会给出出错提示的。

17.1.3. uvm_sqr_if_base

类的原型如下：

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

 37 virtual class uvm_sqr_if_base #(type T1=uvm_object, T2=T1);
```

与 uvm_tlm_if_base 一样，这个类也没有派生自任何类，在类的内部，定义了如下几个接口：

1、get_next_item，是一个 task，对于这个我们已经熟知，当在 driver 中调用 get_next_item 时，就是调用的这个 task：

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

 58    virtual task get_next_item(output T1 t);
 59      uvm_report_error("get_next_item", `UVM_SEQ_ITEM_TASK_ERROR, UVM_NONE);
 60    endtask
```

2、try_next_item，也是一个 task，对于这个我们也已经熟知，当在 driver 中调用 try_next_item 时，就是调用的这个 task。与 get_next_item 相比，它是非阻塞的：

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

 82    virtual task try_next_item(output T1 t);
 83      uvm_report_error("try_next_item", `UVM_SEQ_ITEM_TASK_ERROR, UVM_NONE);
 84    endtask
```

3、item_done，是一个 function，它可以带一个 rsp 的参数，默认下为 Null：

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

105    virtual function void item_done(input T2 t = null);
106      uvm_report_error("item_done", `UVM_SEQ_ITEM_FUNCTION_ERROR, UVM_NONE);
```

```
107     endfunction
```

4、get，也是取一个 sequence_item，不过它调用完后可以不用调用 item_done，这是一个 task。

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

159     virtual task get(output T1 t);
160       uvm_report_error("get", `UVM_SEQ_ITEM_TASK_ERROR, UVM_NONE);
161     endtask
```

5、peek，取一个 sequence_item，同时此 item 依然保留在 sequence 的 fifo 里面，直到 item_done 被调用后才会消除，这也是一个 task。

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

183     virtual task peek(output T1 t);
184       uvm_report_error("peek", `UVM_SEQ_ITEM_TASK_ERROR, UVM_NONE);
185     endtask
```

6、put 就是把一个 response 返回给 sequence，在 put 之间必须设置 transaction id 等信息。是非阻塞的。

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

201     virtual task put(input T2 t);
202       uvm_report_error("put", `UVM_SEQ_ITEM_TASK_ERROR, UVM_NONE);
203     endtask
```

7、put_respone，是一个函数。

```
文件：src/tlm1/sqr_ifs.svh
类：uvm_sqr_if_base #(type T1=uvm_object, T2=T1)

210     virtual function void put_response(input T2 t);
211       uvm_report_error("put_response", `UVM_SEQ_ITEM_FUNCTION_ERROR, UVM_NON
E);
212     endfunction
```

17.2. uvm_port_base 类

本节介绍 uvm_port_base 类。在介绍之前，要先介绍两个类：uvm_port_component_base 和 uvm_port_component 类。

17.2.1. uvm_port_component_base 类

类的原型如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component_base

 50 virtual class uvm_port_component_base extends uvm_component;
```

这是一个派生自 uvm_component 的类，因此具有 uvm_component 的所有的特性。这个类是一个纯虚类，它提供了几个接口：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component_base

 62    pure virtual function void get_connected_to(ref uvm_port_list list);

 70    pure virtual function void get_provided_to(ref uvm_port_list list);

 74    pure virtual function bit is_port();

 78    pure virtual function bit is_export();

 86    pure virtual function bit is_imp();
```

get_connected_to 用于返回所有的此 port 主动连接的端口的列表，如一个 analysis_port 可以连接到多个 IMP 类型的端口，那么此函数将会返回所有的这些端口的列表。PORT，EXPORT 和 IMP 均可以使用此方法。

get_provided_to 用于返回所有的此 port 被动连接的端口的列表。如对于一个 IMP 来说，可能有多个 port，export 连接到此端口，此函数将会返回这些端口的列表。只有 IMP，EXPORT 才能使用此方法。此函数与 get_connected_to 的区别在于一个是被动连接的，一个是主动连接的。

is_port，is_export，is_imp，这三个接口用于判断端口是否是一个 PORT，EXPORT 和 IMP。

这几个接口都是没有任何实质内容的接口，它们都需要在派生类里重载才能使用。

除了这几个接口外，由于它是派生自 uvm_component，它具有 build_phase 函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component_base
函数/任务：build_phase

 89    virtual function void build_phase(uvm_phase phase);
 90      build(); //for backward compat
 91      return;
 92    endfunction
```

它重载了 build_phase，没有调用 super.build_phase，从而把自动 config 给关闭了。

17.2.2. uvm_port_component

类的原型如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component #(type PORT=uvm_object)

108 class uvm_port_component #(type PORT=uvm_object) extends uvm_port_component_base;
```

它实现了 uvm_port_component_base 的定义的各种方法，如 is_port，get_connected_to 等。它当中有一个最重要的成员变量：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component #(type PORT=uvm_object)

110   PORT m_port;
```

如 is_port，get_connected_to 等方法其实都是 m_port.is_port，m_port.get_connected_to 等。

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component #(type PORT=uvm_object)

136   virtual function void get_connected_to(ref uvm_port_list list);
137     m_port.get_connected_to(list);
138   endfunction

140   virtual function void get_provided_to(ref uvm_port_list list);
141     m_port.get_provided_to(list);
142   endfunction
```

```
143
144     function bit is_port ();
145       return m_port.is_port();
146     endfunction
147
148     function bit is_export ();
149       return m_port.is_export();
150     endfunction
151
152     function bit is_imp ();
153       return m_port.is_imp();
154     endfunction
```

17.2.3. uvm_port_base 的基本定义

类的原型如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

195 virtual class uvm_port_base #(type IF=uvm_void) extends IF;
```

这个定义是让人非常不能理解的一个定义。为什么要有 IF 这么个东西呢？我们放在实际的环境中来考虑这个问题。

```
class uvm_analysis_port # (type T = int)
  extends uvm_port_base # (uvm_tlm_if_base #(T,T));
```

这是 uvm_analysis_port 的定义，这个定义也很让人费解，但是如果把 uvm_port_base 部分分解开：

```
class uvm_analysis_port # (type T = int)
  extends uvm_port_base # (uvm_tlm_if_base #(T,T)) extends uvm_tlm_if_base#(T, T);
```

或者省略一点：

```
class uvm_analysis_port # (type T = int)
  extends uvm_port_base extends uvm_tlm_if_base#(T, T);
```

这里用到了 T 类型，如果 uvm_port_base 是一个非参数化的类，要想把 T 从 uvm_analysis_port 的定义中传递到 uvm_tlm_if_base 是很困难的。为了传递给 uvm_tlm_if_base，可以这样写：

```
class uvm_analysis_port#(type T = int)
  extends uvm_port_base#(type T) extends uvm_tlm_if_base#(T,T);
```

但是这样一来的话，所有的 uvm_port_base 都要派生自 uvm_tlm_if_base。如果

想要派生自 uvm_sqr_if_base 就不可能了。为考虑这两种情况，所以要在 uvm_port_base 的声明中使用 IF。

```
class uvm_analysis_port # (type T = int)
  extends uvm_port_base # (uvm_tlm_if_base #(T,T)) extends uvm_tlm_if_base#(T, T);
```

我们前面已经知道，uvm_tlm_if_base 是一个单独的类，它没有派生自任何的类，而 uvm_analysis_port 最终是派生自 uvm_tlm_if_base，所以这不免让我们疑惑：原来 uvm_analysis_port 真的是什么都不是，它不是一个 component，也不是一个 object。其实这样说法并没有什么不对的地方，不过为了便于理解，我们来看下面的两个变量：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

198    typedef uvm_port_base #(IF) this_type;

204    uvm_port_component #(this_type) m_comp;
```

在 new 函数中，有如下语句：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：new

241       m_comp = new(name, parent, this);
```

而 uvm_port_component 的 new 函数是这么定义的：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component #(type PORT=uvm_object)
函数/任务：new

112    function new (string name, uvm_component parent, PORT port);
113       super.new(name,parent);
114       if (port == null)
115          uvm_report_fatal("Bad usage", "Null handle to port", UVM_NONE);
116       m_port = port;
117    endfunction
```

因此，uvm_port_base 内部有一个 m_comp 的指针，这个指针指向一个 uvm_port_component，而后者内部有一个 m_port 的指针，这个指针指向一个 uvm_port_base。所以一个 uvm_port_base 和一个 uvm_port_component 其实是严格一一对应的。虽然 uvm_port_base 是派生自 uvm_tlm_if_base，但是由于它与 uvm_port_component 的一一对应的关系，也可以理解成 uvm_port_base 的行为在很大程度上是跟一个 uvm_component 非常像的。每一个 uvm_port_component 都是在 UVM 树上占有一席之地的，所以 uvm_port_base 也在 UVM 树上占有一席之地。

如果上面的解释依然让你困扰，那么想想假如 uvm_port_component 的 new 函数

是这么写的：

```
function new (string name, uvm_component parent);
  super.new(name,parent);
  m_port = new;
  m_port.m_comp = this;
endfunction
```

这样，这个 port 都变成了 uvm_port_component 的一个成员变量，在分析的时候大家会把 uvm_port_component 的放在第一感觉的位置上，觉得 port 是它的成员变量。而不是把 uvm_port_base 放在主要的位置上，觉得 uvm_port_component 是它的一个成员变量。

uvm_port_base 中有如下两个成员变量：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

208    local int                  m_min_size;
209    local int                  m_max_size;
```

这两个变量分别表明最少最多要有多少个其它的端口连接到此 port 上。通过这两个变量做了严格的限制。如对于一个 PORT 和一个 EXPORT 来说，这两个值都为 1，表明只能有一个连接到上面。而对于 uvm_analysis_port 来说，则 m_min_size 为 0，而 m_max_size 没有限制。因为 analysis_port 本身就是一个广播性质的。作为广播来说，它不会对受众做出限制，一个听众没有也没有关系，有无数个观众也没有关系。

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

207    local uvm_port_type_e     m_port_type;
```

上面的成员变量用于表明此 port 的类型。uvm_port_type_e 的定义如下：

```
文件：src/base/uvm_object_globals.svh
类：无

322 // Enum: uvm_port_type_e
323 //
324 // Specifies the type of port
325 //
326 // UVM_PORT               - The port requires the interface that is its type
327 //                        parameter.
328 // UVM_EXPORT             - The port provides the interface that is its type
329 //                        parameter via a connection to some other export or
330 //                        implementation.
331 // UVM_IMPLEMENTATION - The port provides the interface that is its type
332 //                        parameter, and it is bound to the component that
333 //                        implements the interface.
```

```
334
335 typedef enum
336 {
337    UVM_PORT ,
338    UVM_EXPORT ,
339    UVM_IMPLEMENTATION
340 } uvm_port_type_e;
```

可见，一共有三种类型，一是 UVM_PORT，一是 UVM_EXPORT，一是 UVM_IMPLEM ENTATION。

对于一个 port 来说，如 uvm_put_port：

```
文件：src/tlm1/uvm_ports.svh
类：uvm_put_port#(type T=int)

 92 class uvm_put_port #(type T=int)
 93    extends uvm_port_base #(uvm_tlm_if_base #(T,T));
 94    `UVM_PORT_COMMON(`UVM_TLM_PUT_MASK,"uvm_put_port")
 95    `UVM_PUT_IMP (this.m_if, T, t)
 96 endclass
```

它会调用 UVM_PORT_COMMON 宏：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

181 `define UVM_PORT_COMMON(MASK,TYPE_NAME) \
182    function new (string name, uvm_component parent, \
183                  int min_size=1, int max_size=1); \
184       super.new (name, parent, UVM_PORT, min_size, max_size); \
185       m_if_mask = MASK; \
186    endfunction \
187    `UVM_TLM_GET_TYPE_NAME(TYPE_NAME)
```

可见，在调用 super.new 时，会传入 UVM_PORT 来表明这是一个 PORT。

同样的，EXPORT 和 IMP 也都会在 new 的时候传入相应的参数。

uvm_port_base 实现了 is_port，is_export，is_imp 等成员函数，这些函数都非常简单：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

337    function bit is_port ();
338       return m_port_type == UVM_PORT;
339    endfunction

343    function bit is_export ();
344       return m_port_type == UVM_EXPORT;
345    endfunction
```

```
352    function bit is_imp ();
353      return m_port_type == UVM_IMPLEMENTATION;
354    endfunction
```

由于有了 m_port_type 变量，所以这里可以根据这个变量的值直接给出结果。

17.2.4. connect 函数

在 env 的 connect_phase，我们通常要调用 connect 函数，如：

```
A.port.connect(B.imp);
```

下面我们分析 connect 函数，这个函数是 uvm_port_base 中最重要的一个函数，函数的定义如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：connect

427    virtual function void connect (this_type provider);
428      uvm_root top = uvm_root::get();
429      if (end_of_elaboration_ph.get_state() == UVM_PHASE_EXECUTING || // TBD tidy
430            end_of_elaboration_ph.get_state() == UVM_PHASE_DONE ) begin
431          m_comp.uvm_report_warning("Late Connection",
432            {"Attempt to connect ",this.get_full_name()," (of type ",this.get_type_name(),
433             ") at or after end_of_elaboration phase.  Ignoring."});
434          return;
435       end
436
437      if (provider == null) begin
438        m_comp.uvm_report_error(s_connection_error_id,
439                                "Cannot connect to null port handle", UVM_NONE);
440        return;
441      end
442
443      if ((provider.m_if_mask & m_if_mask) != m_if_mask) begin
444        m_comp.uvm_report_error(s_connection_error_id,
445           {provider.get_full_name(),
446            " (of type ",provider.get_type_name(),
447            ") does not provide the complete interface required of this port (type ",
448            get_type_name(),")"}, UVM_NONE);
449        return;
450      end
451
452      // IMP.connect(anything) is illegal
453      if (is_imp()) begin
```

```
454           m_comp.uvm_report_error(s_connection_error_id,
455             $sformatf(
456 "Cannot call an imp port's connect method. An imp is connected only to the component pa
ssed in its constructor. (You attempted to bind this imp to %s)"      , provider.get_full_name()),
UVM_NONE);
457           return;
458         end
459
460         // EXPORT.connect(PORT) are illegal
461         if (is_export() && provider.is_port()) begin
462           m_comp.uvm_report_error(s_connection_error_id,
463             $sformatf(
464 "Cannot connect exports to ports Try calling port.connect(export) instead. (You attempted to
bind this export to %s).", provider.get_full_name()), UVM_      NONE);
465           return;
466         end
467
468         void'(m_check_relationship(provider));
469
470         m_provided_by[provider.get_full_name()] = provider;
471         provider.m_provided_to[get_full_name()] = this;
472
473     endfunction
```

429 到 435 行用于保证要在 end_of_elaboration_phase 之前调用 connect 函数，也即在 connect_phase 或者 build_phase。一般的，由于存在一些实例的实例化先后顺序问题，都在 connect_phase 中调用此函数。

437 不对劲 431 行用于避免要连接的 port 为 null 的情况。一个 port 是不能和 null 连接的。

443 到 450 行用于限制连接的种类，只有两个具有相同或相近的 if_mask 的 port 才能相连。这里出现了 m_if_mask：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

201     protected int unsigned   m_if_mask;
```

其定义比较简单，在 new 函数中对其进行赋值，如上节的 UVM_PORT_COMMON 宏中，在 new 中对 m_if_mask 进行赋值。

关于各个 mask 的定义位于 uvm_tlm_defines.svh 文件中：

```
文件：src/macros/uvm_tlm_defines.svh

560 `define UVM_TLM_BLOCKING_PUT_MASK              (1<<0)
561 `define UVM_TLM_BLOCKING_GET_MASK              (1<<1)
562 `define UVM_TLM_BLOCKING_PEEK_MASK             (1<<2)
563 `define UVM_TLM_BLOCKING_TRANSPORT_MASK       (1<<3)
564
```

```
565 `define UVM_TLM_NONBLOCKING_PUT_MASK           (1<<4)
566 `define UVM_TLM_NONBLOCKING_GET_MASK           (1<<5)
567 `define UVM_TLM_NONBLOCKING_PEEK_MASK          (1<<6)
568 `define UVM_TLM_NONBLOCKING_TRANSPORT_MASK  (1<<7)
569
570 `define UVM_TLM_ANALYSIS_MASK                  (1<<8)
571
572 `define UVM_TLM_MASTER_BIT_MASK                (1<<9)
573 `define UVM_TLM_SLAVE_BIT_MASK                 (1<<10)
```

上面只是列出了部分 mask 的定义。每个 port 在实例化的时候都指定好 m_if_mask，然后在 connect 时，可以通过检查 m_if_mask 的值看看两个 port 是否可以连接。

453 到 458 行表明，一个 IMP 是不能主动的调用 connect 函数，它只能作为 connect 函数的参数，被别的 port 连接。

461 到 466 行表明，export.connect(port)是不被允许的。

468 行调用 m_check_relationship，用于再次检查两个连接的端口之间的连接是否是合法的，函数的定义如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：m_check_relationship

613   local function bit  m_check_relationship (this_type provider);
614     string s;
615     this_type from;
616     uvm_component from_parent;
617     uvm_component to_parent;
618     uvm_component from_gparent;
619     uvm_component to_gparent;
620
621     // Checks that the connection is between ports that are hierarchically
622     // adjacent (up or down one level max, or are siblings),
623     // and check for legal direction, requirer.connect(provider).
624
625     // if we're an analysis port, allow connection to anywhere
626     if (get_type_name() == "uvm_analysis_port")
627       return 1;
628
629     from        = this;
630     from_parent  = get_parent();
631     to_parent    = provider.get_parent();
632
633     // skip check if we have a parentless port
634     if (from_parent == null || to_parent == null)
635       return 1;
636
637     from_gparent = from_parent.get_parent();
```

```
638       to_gparent     = to_parent.get_parent();
639
640       // Connecting port-to-port: CHILD.port.connect(PARENT.port)
641       //
642       if (from.is_port() && provider.is_port() && from_gparent != to_parent) begin
643         s = {provider.get_full_name(),
644              " (of type ",provider.get_type_name(),
645              ") is not up one level of hierarchy from this port. ",
646              "A port-to-port connection takes the form ",
647              "child_component.child_port.connect(parent_port)"};
648         m_comp.uvm_report_warning(s_connection_warning_id, s, UVM_NONE);
649         return 0;
650       end
651
652       // Connecting port-to-export: SIBLING.port.connect(SIBLING.export)
653       // Connecting port-to-imp:       SIBLING.port.connect(SIBLING.imp)
654       //
655       else if (from.is_port() && (provider.is_export() || provider.is_imp()) &&
656                  from_gparent != to_gparent) begin
657           s = {provider.get_full_name(),
658              " (of type ",provider.get_type_name(),
659              ") is not at the same level of hierarchy as this port. ",
660              "A port-to-export connection takes the form ",
661              "component1.port.connect(component2.export)"};
662         m_comp.uvm_report_warning(s_connection_warning_id, s, UVM_NONE);
663         return 0;
664       end
665
666       // Connecting export-to-export: PARENT.export.connect(CHILD.export)
667       // Connecting export-to-imp:       PARENT.export.connect(CHILD.imp)
668       //
669       else if (from.is_export() && (provider.is_export() || provider.is_imp()) &&
670                  from_parent != to_gparent) begin
671         s = {provider.get_full_name(),
672              " (of type ",provider.get_type_name(),
673              ") is not down one level of hierarchy from this export. ",
674              "An export-to-export or export-to-imp connection takes the form ",
675              "parent_export.connect(child_component.child_export)"};
676         m_comp.uvm_report_warning(s_connection_warning_id, s, UVM_NONE);
677         return 0;
678       end
679
680       return 1;
681     endfunction
```

函数主要检查 child，parent 之间的互相的连接关系。这个函数的意思相对比较明了，因此在这里不多做介绍。

470 行和 471 行分别在 m_provided_by 及 m_provided_to 插入一条记录。这是两个联合数组，它们的定义为：

文件：src/base/uvm_port_base.svh

```
类：uvm_port_base #(type IF=uvm_void)

205    local this_type m_provided_by[string];
206    local this_type m_provided_to[string];
```

索引是 string 类型的，而存放的内容是 uvm_port_base 类型的。以 A.connect(B)为例，经过 470 行和 471 行，那么 A 的 m_provided_by 中将会有一条 B 的记录，而 B 的 m_provided_to 中将会有一条 A 的记录。由于 IMP 只能作为 connect 的参数，所以可以推测，对于一个 IMP 来说，其 m_provided_by 中将永远是空的，不会有任何记录存在。

17.2.5. resolve_bindings

resolve_bindings 会自动的被调用。我们知道，在分析 phase 机制的时候，我们分析过，在每个 phase 开始执行之前，会调用 phase_started，普通的 uvm_component 的 phase_started 是一个空函数，但是 uvm_root 则对重载了此函数：

```
文件：src/base/uvm_root.svh
类：uvm_root
函数/任务：phase_started

188    function void phase_started(uvm_phase phase);
189      if (phase == end_of_elaboration_ph) begin
190        do_resolve_bindings();
191        if (enable_print_topology) print_topology();
192
193        begin
194              uvm_report_server srvr;
195            srvr = get_report_server();
196            if(srvr.get_severity_count(UVM_ERROR) > 0) begin
197              uvm_report_fatal("BUILDERR", "stopping due to build errors", UVM_NONE);
198            end
199        end
200      end
201    endfunction
```

189 行判断是不是接下来要执行的是 end_of_elaboration_phase，如果是，那么 190 行会调用 do_resolve_bindings。这也就意味着，resolve_bindings 是在 connect_phase 完毕之后，end_of_elaboration_phase 开始之前被调用的。

do_resolve_bindings 是 uvm_component 定义的一个函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：do_resolve_bindings
```

```
2545 function void uvm_component::do_resolve_bindings();
2546    foreach( m_children[s] )
2547      m_children[s].do_resolve_bindings();
2548    resolve_bindings();
2549 endfunction
```

根据这个函数，可以看的出来，系统会自底向上的调用 uvm_component 的 resolve_bindings 函数。对于普通的 component 来说，其 resolve_bindings 函数是一个空函数：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：resolve_bindings

2537 function void uvm_component::resolve_bindings();
2538    return;
2539 endfunction
```

前面已经介绍过，一个 port 是和一个 uvm_component 对应的。uvm_port_component 中重载了这个函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_component #(type PORT=uvm_object)
函数/任务：resolve_bindings

124    virtual function void resolve_bindings();
125      m_port.resolve_bindings();
126    endfunction
```

它会调用与其对应的 port 的 resolve_bindings。因此，每一个 port 内部的 resolve_bindings 是会自动被调用的。

函数的定义如下：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：resolve_bindings

712    virtual function void resolve_bindings();
713      if (m_resolved) // don't repeat ourselves
714        return;
715
716      if (is_imp()) begin
717        m_imp_list[get_full_name()] = this;
718      end
719      else begin
720        foreach (m_provided_by[nm]) begin
721          this_type port;
722          port = m_provided_by[nm];
723          port.resolve_bindings();
```

```
724             m_add_list(port);
725           end
726       end
727
728       m_resolved = 1;
729
730       if (size() < min_size() ) begin
731         m_comp.uvm_report_error(s_connection_error_id,
732           $sformatf("connection count of %0d does not meet required minimum of %0d",
733           size(), min_size()), UVM_NONE);
734       end
735
736       if (max_size() != UVM_UNBOUNDED_CONNECTIONS && size() > max_size() ) beg
in
737         m_comp.uvm_report_error(s_connection_error_id,
738           $sformatf("connection count of %0d exceeds maximum of %0d",
739           size(), max_size()), UVM_NONE);
740       end
741
742       if (size())
743         set_if(0);
744
745     endfunction
```

整个函数的关键在于 716 到 726 行。以一个例子来说明：

```
A.connect(B);
```

其中 A 是一个 PORT，而 B 是一个 IMP，那么在 B 的 resolve_bindings 函数中，716 行的条件是满足的，于是 B 的 m_imp_list 中插入了一条自己的记录。m_imp_list 是一个联合数组：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

211     local this_type              m_imp_list[string];
```

这个联合数组的索引是 string 类型的，而存放的内容则是 uvm_port_base 类型的。

当 A 的 resolve_bindings 被调用时，716 行的条件不满足，于是进入 719 行的分支。在这个分支中将会遍历 m_provided_by 中所有的记录。对于 A 来说，其 m_provided_by 中只有一条记录：B。于是接下来会调用 B 的 resolve_bindings。这里可能就产生冲突了，因为前面已经说过了，系统会自下而上的调用相应的 resolve_bindings 函数。那么 B 的 resolve_bindings 可能会被重复调用。为了避免这种情况，uvm_port_base 中使用变量 m_resolved 来标志着 resolve_bindings 是否被调用过。当 resolve_bindings 被调用时，728 行会让 m_resolved 赋值为 1。当下一次调用时，根据 713 行，将会直接返回。

接头上看 719 行的分支。723 行执行完毕后，即 B.resolve_bindings 之后，B 的

m_imp_list 中有了一条自己的记录。724 行以 B 为参数调用 m_add_list 函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：m_add_list

688    local function void m_add_list                (this_type provider);
689      string sz;
690      this_type imp;
691
692      for (int i = 0; i < provider.size(); i++) begin
693        imp = provider.get_if(i);
694        if (!m_imp_list.exists(imp.get_full_name()))
695          m_imp_list[imp.get_full_name()] = imp;
696      end
697
698    endfunction
```

692 行用到了 size 函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：size

363    function int size ();
364      return m_imp_list.num();
365    endfunction
```

它其实就是 m_imp_list 中记录的数量，在我们的例子中为 1。

693 行调用 B 的 get_if 函数，传入的参数为 0：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：get_if

755    function uvm_port_base #(IF) get_if(int index=0);
756      string s;
757      if (size()==0) begin
758        m_comp.uvm_report_warning("get_if",
759          "Port size is zero; cannot get interface at any index", UVM_NONE);
760        return null;
761      end
762      if (index < 0 || index >= size()) begin
763        $sformat(s, "Index %0d out of range [0,%0d]", index, size()-1);
764        m_comp.uvm_report_warning(s_connection_error_id, s, UVM_NONE);
765        return null;
766      end
767      foreach (m_imp_list[nm]) begin
768        if (index == 0)
769          return m_imp_list[nm];
770        index--;
```

```
771       end
772     endfunction
```

757 到 766 行检查传入的参数及 size 的合理性。

767 到 771 行则返回 m_imp_list 中序号为 index 的记录。这是为了处理有多个 IMP 连接到同一个 port 上的情况。在本例中，将会直接返回 B 的 m_imp_list 中唯一的一条记录，即 B 自身。

回到 m_add_list 函数，695 行把得到的记录插入了自己的 m_imp_list 中。

resolve_bindings 的 730 到 740 行用到了 min_size 和 max_size 函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)

309     function int max_size ();
310       return m_max_size;
311     endfunction

319     function int min_size ();
320       return m_min_size;
321     endfunction
```

min_size 和 max_size 分别为此 port 在实例化时传入的 min_size 和 max_size。730 行到 740 行检查连接到此 port 上的端口的数量是否符合要求。

743 行调用 set_if 函数：

```
文件：src/base/uvm_port_base.svh
类：uvm_port_base #(type IF=uvm_void)
函数/任务：set_if

368     function void set_if (int index=0);
369       m_if = get_if(index);
370       if (m_if != null)
371         m_def_index = index;
372     endfunction
```

这段代码的意思就是把 m_if 的值设置 m_imp_list 中第一条记录。对于 A 来说，resolve_bindings 被调用完后，那么其 m_if 将会指向 B，而 B 的 m_if 将会指向自身。

resolve_bindings 其实是为了应付 connect 的传递行为，如：

```
port_b.connect(imp);
port_a.connect(port_b);
```

这种连接方式是允许的，当 resolve_bindings 执行完毕后，port_a 的 m_if 指向的是 imp，port_b 的 m_if 指向的是 imp，imp 的 m_if 指向的是自身。port_a，port_b 及 imp 的 m_imp_list 都只有一条记录，这条记录中记录的是 imp。

这是当 m_imp_list 中只有一条记录的情况。假如有多条记录，如一个 analysis_port 可以连接多个 imp：

```
ap.connect(imp_0);
ap.connect(imp_1);
…
ap.connect(imp_n);
```

那么 ap 的 m_imp_list 中存放的将会是 imp_0 到 imp_n，共有 n 条记录，而其 m_if 指向的是这些 imp 中的第一个，即 imp_0。

17.3. 常用的 port 的定义

17.3.1. uvm_*_imp

我们已经知道，做为 IMP 来说，它是只能做为 connect 函数的参数的。并且大部分的工作都是由 IMP 来实现的。以 uvm_blocking_put_imp 为例来说明：

```
文件：src/tlm1/uvm_imps.svh
类：uvm_blocking_put_imp #(type T=int, type IMP=int)

 82 class uvm_blocking_put_imp #(type T=int, type IMP=int)
 83   extends uvm_port_base #(uvm_tlm_if_base #(T,T));
 84   `UVM_IMP_COMMON(`UVM_TLM_BLOCKING_PUT_MASK,"uvm_blocking_put_imp",IMP)
 85   `UVM_BLOCKING_PUT_IMP (m_imp, T, t)
 86 endclass
```

有两个参数，第一个是 T，第二个是 IMP。如果离开实际环境看这两个参数会相当的困惑。如果我们是在一个 scoreboard 中使用这个 IMP 的话，我们会这么声明：

```
class my_scoreboard extends uvm_scoreboard;
  uvm_blocking_put_imp #( mac_transaction, my_scoreboard) my_imp;
  …
endclass
```

也就是说，第一个参数 T 其实是此 IMP 传递的 transaction 的类型，而第二个参数则是此 IMP 所在的 uvm_component 类的名称。

uvm_blocking_put_imp 调用了两个宏，其中 UVM_IMP_COMMON 的展开为：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

205 `define UVM_IMP_COMMON\
206   local IMP m_imp; \
207   function new (string name, IMP imp); \
208     super.new (name, imp, UVM_IMPLEMENTATION, 1, 1); \
209     m_imp = imp; \
210     m_if_mask = MASK; \
211   endfunction \
212   `UVM_TLM_GET_TYPE_NAME(TYPE_NAME)
```

206 行声明了一个 m_imp 变量，并在 209 行时对其赋值。如果展开来看，就相当于是有如下的代码：

```
class uvm_blocking_put_imp#(mac_transaction, my_scoreboard) extends …;
   my_scoreboard m_imp;
   function new(string name, my_scoreboard imp);
     super.new(…);
     m_imp = imp;
     …
   endfunction
   …
endclass
```

而在 my_scoreboard 中把 my_imp 时进行实例化的时候，会对 m_imp 进行赋值：

```
function my_scoreboard::new(string name, uvm_component parent);
   super.new(name, parent);
   my_imp = new("my_imp", this);
endfunction
```

也就相当于是 m_imp 其实指的就是此 IMP 所在的 component 的指针。

212 行调用了 UVM_TLM_GET_TYPE_NAME 宏：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

176 `define UVM_TLM_GET_TYPE_NAME(NAME) \
177   virtual function string get_type_name(); \
178     return NAME; \
179   endfunction
```

这里比较简单中，只是声明了一个 get_type_name 的函数。

回到 uvm_blocking_put_imp 的定义中，其调用的另外一个宏是 UVM_BLOCKING_ PUT_IMP：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无
```

```
 97 `define UVM_BLOCKING_PUT_IMP(imp, TYPE, arg) \
 98   task put (TYPE arg); \
 99     imp.put(arg); \
100   endtask
```

这个宏也相当简单，它只是重载了 put。put 是哪里来的？这里的 put 是属于 uvm_blocking_put_imp，而 uvm_blocking_put_imp 是从 uvm_tlm_if_base 派生来的。在本章一开始的时候就分析了这个类，它里面定义了 put，get 等许多的接口。由于这只是一个 uvm_blocking_put_imp，所以它只重载了 put，但是像 get，try_get 等调用的时候，依然会使用 uvm_tlm_if_base 的 get，try_get，给出出错提示。这里我们也可以推测，其实对于 uvm_*_imp 来说，它的本质上只是重载了*号对应的接口，这里的*可以是 nonblocking_put，blocking_get 等等。对于 nonblocking_put 来说，它要重载 try_put，can_put，而对于 blocking_get 来说，它要重载 get。

这里的所谓的重载其实也很简单，UVM_BLOCKING_PUT_IMP 的 99 行调用了 m_imp 的 put 函数。这也意味着，在 my_scoreboard 中必须要有一个 put 函数，否则函数就会出错。同样的，try_put，get 等的重载也与此相同。也就是说，假如 my_scoreboard 中用的是 uvm_blocking_get_imp，那么就需要在 my_scoreboard 中预先定义好一个 get，假如用的是 uvm_nonblocking_put_imp，那么就需要在 my_scoreboard 中预先定义好 try_put，try_get。否则系统在编译时就会出错。所以，大部分的工作是在 my_scoreboard 中来完成的，IMP 通过调用它所在的 component 的相关函数来完成端口的功能。对于 IMP 本身来说，它其实本身不具体任何端口的处理特性，端口的行为完全取决于在 my_scoreboard 中定义的相关函数。甚至从某种意义上说，我们还可以疯狂的这样做：在 my_scoreboard 中定义 put 时，把其定义成是一个非阻塞的，而把 try_put 定义成是阻塞的。没有人会对此说什么，只要用的习惯就 OK。不过还是强烈不推荐这样做的！

17.3.2. uvm_*_port 与 uvm_*_export

以 uvm_blocking_put_port 来例说明：

```
文件：src/tlm1/uvm_ports.svh
类：uvm_blocking_put_port #(type T=int)

 80 class uvm_blocking_put_port #(type T=int)
 81   extends uvm_port_base #(uvm_tlm_if_base #(T,T));
 82   `UVM_PORT_COMMON(`UVM_TLM_BLOCKING_PUT_MASK,"uvm_blocking_put_port")
 83   `UVM_BLOCKING_PUT_IMP (this.m_if, T, t)
 84 endclass
```

这里调用了 UVM_PORT_COMMON 宏：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

181 `define UVM_PORT_COMMON(MASK,TYPE_NAME) \
182   function new (string name, uvm_component parent, \
183                 int min_size=1, int max_size=1); \
184     super.new (name, parent, UVM_PORT, min_size, max_size); \
185     m_if_mask = MASK; \
186   endfunction \
187   `UVM_TLM_GET_TYPE_NAME(TYPE_NAME)
```

宏的展开并不复杂，只是给 m_if_mask 赋值。关于这个变量的意义，在介绍 uvm_port_base 的 connect 函数时已经有过介绍，这里不多做介绍。

uvm_blocking_put_port 用的第二个宏是 UVM_BLOCKING_PUT_IMP：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

 97 `define UVM_BLOCKING_PUT_IMP(imp, TYPE, arg) \
 98   task put (TYPE arg); \
 99     imp.put(arg); \
100   endtask
```

这个宏在上节介绍 uvm_blocking_put_imp 时也出现过，当时传入的 imp 参数是 m_imp，即 IMP 所在的 my_scoreboard 的指针。而在这里，传入的则是 m_if。在上节中已经说过，m_if 其实指的是连接到此 PORT 的 imp 的指针。

假如有如下连接：

```
comp_a.blocking_put_port.connect(comp_b.blocking_put_imp);
```

那么当我们在 comp_a 中使用 put 时：

```
blocking_put_port.put(tr);
```

将会调用 blocking_put_imp 的 put，而这个 put，上一节已经说过，将会调用 blocking_put_imp 所在 comp_b 的 put。

uvm_*_export 与 uvm_*_port 几乎完全是相同的。假如有如下连接：

```
comp_a.blocking_put_export.connect(comp_b.blocking_put_imp);
```

那么当我们在 comp_a 中使用 put 时：

```
blocking_put_export.put(tr);
```

将会调用 blocking_put_imp 的 put，最终会调用 blocking_put_imp 所在 comp_b 的 put。

那么假如是一个 port 连接一个 export 呢？

```
comp_a.blocking_put_port.connect(comp_b.blocking_put_export);
```

这种连接方式是允许的，只是问题在于，当在 comp_a 中使用 put 时，由于没有任何的 IMP 连接到 blocking_put_port 上，所以 m_if 将会为 null，也就是会调用 null.put。这是不被请允许的。这也就意味着，一个 port 连接一个 export，在 connect 的时候是允许的，但是当使用 put 等方法的时候则会出错。对于这种情况，可以让 export 再连接一个 imp：

```
comp_b.blocking_put_export.connect(comp_c.blocking_put_imp);
```

这样，当在 comp_a 中使用 put 时，最终会使用 blocking_put_imp 的 put，而后者又会调用 comp_c 的 put。这也就意味着，UVM 中，一个 TLM 的连接关系，必须以一个 imp 作为整个连接关系的终止。而能以一个 export 作为终结点。

17.3.3. uvm_analysis_*

uvm_analysis_port 的定义为：

```
文件：src/tlm1/uvm_analysis_port.svh
类：uvm_analysis_port # (type T = int)

 56 class uvm_analysis_port # (type T = int)
 57   extends uvm_port_base # (uvm_tlm_if_base #(T,T));
 58
 59   function new (string name, uvm_component parent);
 60     super.new (name, parent, UVM_PORT, 0, UVM_UNBOUNDED_CONNECTIONS);
 61     m_if_mask = `UVM_TLM_ANALYSIS_MASK;
 62   endfunction
 63
 64   virtual function string get_type_name();
 65     return "uvm_analysis_port";
 66   endfunction
 67
 68   // Method: write
 69   // Send specified value to all connected interface
 70   function void write (input T t);
 71     uvm_tlm_if_base # (T, T) tif;
 72     for (int i = 0; i < this.size(); i++) begin
 73       tif = this.get_if (i);
 74       if ( tif == null )
 75         uvm_report_fatal ("NTCONN", {"No uvm_tlm interface is connected to ", get_full_
name(), " for executing write()"}, UVM_NONE);
 76       tif.write (t);
 77     end
 78   endfunction
 79
```

```
80 endclass
```

这里需要注意的是 60 行时传入了 0 和 UVM_UNBODUNDED_CONNECTIONS 两个参数，表示可以有任意个 IMP 来连接到此 port 上来。

70 行重写了 write 函数。由于可以有任意个 IMP 连接到此 port 上来，由 16.2.5 节的分析可知，这些 IMP 都存放在 m_imp_list 中，可以通过 get_if 来取得所有的这些 IMP。72 行遍历所有的这些 IMP，76 行调用这些 IMP 的 write 函数。这些 IMP 的 wirte 函数又将会调用所在的 component 的 write 函数。从而达到广播的目的。

uvm_analysis_export 与 uvm_analysis_port 的定义相似，这里不重复阐述。uvm_analysis_imp 与上述两者有区别：

文件：src/tlm1/uvm_analysis_port.svh
类：uvm_analysis_export # (type T = int)

```
126 class uvm_analysis_export #(type T=int)
127   extends uvm_port_base #(uvm_tlm_if_base #(T,T));
128
129   // Function: new
130   // Instantiate the export.
131   function new (string name, uvm_component parent = null);
132     super.new (name, parent, UVM_EXPORT, 1, UVM_UNBOUNDED_CONNECTIONS);
133     m_if_mask = `UVM_TLM_ANALYSIS_MASK;
134   endfunction
135
136   virtual function string get_type_name();
137     return "uvm_analysis_export";
138   endfunction
139
140   // analysis port differs from other ports in that it broadcasts
141   // to all connected interfaces. Ports only send to the interface
142   // at the index specified in a call to set_if (0 by default).
143   function void write (input T t);
144     uvm_tlm_if_base #(T, T) tif;
145     for (int i = 0; i < this.size(); i++) begin
146       tif = this.get_if (i);
147       if (tif == null)
148           uvm_report_fatal ("NTCONN", {"No uvm_tlm interface is connected to ", get_full
_name(), " for executing write()"}, UVM_NONE);
149       tif.write (t);
150     end
151   endfunction
152
153 endclass
```

这里调用了 UVM_IMP_COMMON 宏，前面已经介绍过这个宏。从整个定义来看，uvm_analysis_imp 其实更像是一个普通的 imp，它不具有任何的 analysis 的性质，即不具备任何广播的性质。这也是可以理解的，因为一个 IMP 是放在整个连接关系的终结点上的，对于这种终结点，是不存在广播的概念的。如果非要找出与其它 IMP

的区别，那么对于下例：

```
ap.connect(imp);
```

这里的 ap 是 analysis_port 或者 analysis_export。对于上述连接关系，imp 只能是 analysis_imp，而不能是其它的 IMP。

17.3.4. fifo 的使用

使用 IMP 作为终结点的一个问题是必须在 IMP 所在的 component 中定义好相应的函数，如一个 analysis_imp 存在于 scoreboard 中，那么必须在 scoreboard 中定义好 write 函数。

要解决这个问题，可以采用 FIFO。假如 comp_a 与 comp_b 之间要进行通信，那么可以在 comp_a 与 comp_b 之间加入一个 comp_c。

在 comp_a 中定义好一个 analysis_port，在 comp_c 中定义好一个 analysis_imp 与其相连接，并在 comp_c 中定义好一个 write 函数，此函数把从 comp_a 使用 write 函数传递过来的 transaction 放入一块缓存中。

在 comp_b 中定义好一个 blocking_get_port，在 comp_c 中定义好一个 blocking_get_imp 与其相连，并在 comp_c 中定义好一个 get 函数，这样当 comp_b 中通过 get 函数取得 transaction 时，调用 comp_c 的 get，而这个 get 又会从之前 write 写入的缓存中取得数据。

comp_c 内部把所有的工作都做好了，对于 comp_a 和 comp_b 来说，就不存在 IMP 的问题了，对于他们来说只有 PORT 或者 EXPORT。

comp_c 其实就是 fifo 的雏形。

UVM 的 TLM 实现中，fifo 有两种，一是 uvm_analysis_fifo，一是 uvm_tlm_fifo：

```
文件：src/tlm1/uvm_tlm_fifos.svh
类：uvm_tlm_fifo #(type T=int)与 uvm_tlm_analysis_fifo #(type T = int)

 46 class uvm_tlm_fifo #(type T=int) extends uvm_tlm_fifo_base #(T);

200 class uvm_tlm_analysis_fifo #(type T = int) extends uvm_tlm_fifo #(T);
```

我们先看 uvm_tlm_fifo_base 基类：

```
文件：src/tlm1/uvm_tlm_fifos.svh
类：uvm_tlm_fifo_base #(type T=int)

 48 virtual class uvm_tlm_fifo_base #(type T=int) extends uvm_component;
```

```
 49
 50   typedef uvm_tlm_fifo_base #(T) this_type;

 65   uvm_put_imp #(T, this_type) put_export;

 85   uvm_get_peek_imp #(T, this_type) get_peek_export;

 99   uvm_analysis_port #(T) put_ap;

113   uvm_analysis_port #(T) get_ap;

118   uvm_put_imp      #(T, this_type) blocking_put_export;
119   uvm_put_imp      #(T, this_type) nonblocking_put_export;

124   uvm_get_peek_imp #(T, this_type) blocking_get_export;
125   uvm_get_peek_imp #(T, this_type) nonblocking_get_export;
126   uvm_get_peek_imp #(T, this_type) get_export;
127
128   uvm_get_peek_imp #(T, this_type) blocking_peek_export;
129   uvm_get_peek_imp #(T, this_type) nonblocking_peek_export;
130   uvm_get_peek_imp #(T, this_type) peek_export;
131
132   uvm_get_peek_imp #(T, this_type) blocking_get_peek_export;
133   uvm_get_peek_imp #(T, this_type) nonblocking_get_peek_export;

143   function new(string name, uvm_component parent = null);
144     super.new(name, parent);
145
146     put_export = new("put_export", this);
147     blocking_put_export    = put_export;
148     nonblocking_put_export = put_export;
149
150     get_peek_export = new("get_peek_export", this);
151     blocking_get_peek_export    = get_peek_export;
152     nonblocking_get_peek_export = get_peek_export;
153     blocking_get_export         = get_peek_export;
154     nonblocking_get_export      = get_peek_export;
155     get_export                  = get_peek_export;
156     blocking_peek_export        = get_peek_export;
157     nonblocking_peek_export     = get_peek_export;
158     peek_export                 = get_peek_export;
159
160     put_ap = new("put_ap", this);
161     get_ap = new("get_ap", this);
162
163   endfunction
```

可以看到，这个类中真正实例化的成员变量其实只有四个：put_export，get_peek_export，put_ap，get_ap。并且 put_export 和 get_peek_export 的本质也是一个 IMP。所谓的 blocking_get_export 等其实根本不是 export，而只是 IMP。

uvm_tlm_fifo_base 只是提供了几个 port，真正的实现我们前面说的 comp_c 的功

能是在 uvm_tlm_fifo 中实现的。在 uvm_tlm_fifo 中如下的成员变量：

```
文件：src/tlm1/uvm_tlm_fifos.svh
类：uvm_tlm_fifo #(type T=int)

 50    local mailbox #( T ) m;
```

这就是 comp_c 中所说的缓存，用一个 mailbox 实现缓存的功能。这个类具体的实现了 put，get 等接口。以 put 和 get 来例：

```
文件：src/tlm1/uvm_tlm_fifos.svh
类：uvm_tlm_fifo #(type T=int)

113    virtual task put( input T t );
114       m.put( t );
115       put_ap.write( t );
116    endtask
117
118    virtual task get( output T t );
119       m_pending_blocked_gets++;
120       m.get( t );
121       m_pending_blocked_gets--;
122       get_ap.write( t );
```

每当有一个 transaction 通过 put 方法放入此 fifo 中时，一方面会向 mailbox 中放入，另外还通过 pu_ap 把此 transaction 广播出去。

每当有 get 请求从此 fifo 中取得 transaction 时，会从 mailbox 中得到一个 transaction，并且把此 transaction 通过 get_ap 广播出去。

uvm_analysis_fifo 派生自 uvm_tlm_fifo，它只是增加了 write 方法：

```
文件：src/tlm1/uvm_tlm_fifos.svh
类：uvm_analysis_fifo #(type T=int)

200 class uvm_tlm_analysis_fifo #(type T = int) extends uvm_tlm_fifo #(T);

213    uvm_analysis_imp #(T, uvm_tlm_analysis_fifo #(T)) analysis_export;

223    function new(string name ,  uvm_component parent = null);
224       super.new(name, parent, 0); // analysis fifo must be unbounded
225       analysis_export = new("analysis_export", this);
226    endfunction
227
228    const static string type_name = "uvm_tlm_analysis_fifo #(T)";
229
230    virtual function string get_type_name();
231       return type_name;
232    endfunction
233
234    function void write(input T t);
```

```
235       void'(this.try_put(t)); // unbounded => must succeed
236     endfunction
237
238 endclass
```

17.3.5. sequencer 与 driver 之间的连接关系

在 uvm_driver 中如下成员变量：

```
文件：src/comps/uvm_driver.svh
类：  uvm_driver #(type REQ=uvm_sequence_item, type RSP=REQ)

 53     uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;
```

而在 uvm_sequencer 中有如下成员变量：

```
文件：src/seq/uvm_sequencer.svh
类：uvm_sequencer #(type REQ=uvm_sequence_item, RSP=REQ)

 59     uvm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;
```

这本质上其实是一个 IMP，而不是如其名字所说的是一个 EXPORT。

uvm_seq_item_pull_port 的定义为：

```
文件：src/tlm1/sqr_connections.svh
类：uvm_seq_item_pull_port #(type REQ=int, type RSP=REQ)

 54 class uvm_seq_item_pull_port #(type REQ=int, type RSP=REQ)
 55     extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));
 56     `UVM_SEQ_PORT(`UVM_SEQ_ITEM_PULL_MASK, "uvm_seq_item_pull_port")
 57     `UVM_SEQ_ITEM_PULL_IMP(this.m_if, REQ, RSP, t, t)
 58
 59     bit print_enabled = 0;
 60
 61 endclass
```

56 行调用了 UVM_SEQ_PORT 宏：

```
文件：src/tlm1/uvm_tlm_imps.svh
类：无

189 `define UVM_SEQ_PORT(MASK,TYPE_NAME) \
190     function new (string name, uvm_component parent, \
191                   int min_size=0, int max_size=1); \
192       super.new (name, parent, UVM_PORT, min_size, max_size); \
193       m_if_mask = MASK; \
```

```
194     endfunction \
195     `UVM_TLM_GET_TYPE_NAME(TYPE_NAME)
```

这个宏与我们前面所见识的 UVM_PORT_COMMON 宏并没有本质的区别，不重复阐述。

57 行调用 UVM_SEQ_ITEM_PULL_IMP 宏：

```
文件：src/macros/uvm_tlm_defines.svh
类：无

548 `define UVM_SEQ_ITEM_PULL_IMP(imp, REQ, RSP, req_arg, rsp_arg) \
549   task get_next_item(output REQ req_arg); imp.get_next_item(req_arg); endtask \
550   task try_next_item(output REQ req_arg); imp.try_next_item(req_arg); endtask \
551   function void item_done(input RSP rsp_arg = null); imp.item_done(rsp_arg); endfunction \
552   task wait_for_sequences(); imp.wait_for_sequences(); endtask \
553   function bit has_do_available(); return imp.has_do_available(); endfunction \
554   function void put_response(input RSP rsp_arg); imp.put_response(rsp_arg); endfunction \
555   task get(output REQ req_arg); imp.get(req_arg); endtask \
556   task peek(output REQ req_arg); imp.peek(req_arg); endtask \
557   task put(input RSP rsp_arg); imp.put(rsp_arg); endtask
```

这里定义了 get_next_item，item_done 函数。当在 driver 中使用 get_next_item 时：

```
seq_item_port.get_next_item(req);
```

其实质调用的是 seq_item_export 的 get_next_item，而这个函数又会调用 seq_item_export 所在的 component，即 uvm_sequencer 的 get_next_item。其它如 item_done，put_response 等同样如此。

18. register model 源代码分析

UVM 的 register model 是沿用了 synopsys 的寄存器解决方案 RAL。UVM 是由 OVM 派生来的，而在 OVM 中并没有 register model。register model 是一个相对独立的模块。这在 UVM 的源代码的组织形式中可以看出来：在 src 目录下，它单独占据了一个 reg 目录。本节讲述 register model 的源代码。

18.1. 基本的数据结构

18.1.1. 存储数据的基本单位：uvm_reg_field

恰如前面所说，UVM 的 register model 中最基本的存储数据的单位是 uvm_reg_field，它用来代表寄存器中的某几个域。uvm_reg_field 的原型为：

文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field

```
38  class uvm_reg_field extends uvm_object;
```

它是派生自 uvm_object 的一个类，因此具有 uvm_object 的一切特性。在

uvm_reg_field 中真正存储数据的有三个成员变量：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field

   44      rand   uvm_reg_data_t   value; // Mirrored after randomize()
   45
   46      local uvm_reg_data_t   m_mirrored; // What we think is in the HW
   47      local uvm_reg_data_t   m_desired;   // Mirrored after set()
```

首先来看 uvm_reg_data_t 类型：

```
文件：src/reg/uvm_reg_model.svh
类：无

 58 typedef   bit unsigned [`UVM_REG_DATA_WIDTH-1:0]   uvm_reg_data_t ;
```

这个定义位于 uvm_reg_model.svh 文件中，它本质上是 bit 类型，其宽度为 UVM_REG_DATA_WIDTH，这个宽度值是一个宏，其默认值为 64：

```
文件：src/macros/uvm_reg_defines.svh
类：无

 31 `ifndef UVM_REG_ADDR_WIDTH
 32  `define UVM_REG_ADDR_WIDTH 64
 33 `endif
```

我们可以通过改变这个宏的值来变更 register model 所能表示的最大寄存器的值。

上面的三个成员变量中，只有一个 value 是非 local 类型的，也即是在外部可见的，同时，也只有 value 是 rand 类型的，表示它可以进行随机化。当我们要对寄存器进行大规模的随机读写时，就需要用到 value 的 rand 特性。关于这三个的具体的用法，我们在后面慢慢展开。

除了存放数据外，uvm_reg_field 还有一些其它常用的成员变量：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field

   48      local string               m_access;
   49      local uvm_reg              m_parent;
   50      local int unsigned         m_lsb;
   51      local int unsigned         m_size;
   52      local bit                  m_volatile;
   53      local uvm_reg_data_t       m_reset[string];

   60      local bit                  m_individually_accessible = 0;
```

这几位都会在调用 field 的 configure 函数时进行初始化操作。其中 m_access 表

示这个 reg_field 的存储特性，如 RW，RO 等，m_parent 表示此 reg_field 所在的 uvm_reg 的指针，m_lsb 表示这个 reg_field 的最低位在 uvm_reg 中的位置，这个位对于 uvm_reg_field 来说并无太大用处，但是对于 uvm_reg 来说，这个位可以用于检查一个 uvm_reg 中的 uvm_reg_field 是否重叠，是否合法等。m_size 用于表示此 uvm_reg_field 占据了几个 bit。而 m_volatile 则表示此 uvm_reg_field 的 volatile 特性，这一性质用的并不多。m_reset 表示此 uvm_reg_field 复位后的值。注意的是，这里的 m_reset 是一个联合数组，表示一个寄存器可以有多个复位值，如硬复位，软复位等。实际上，一般来说只会用到一种复位，即硬复位。m_individeally_accessible 表示此 uvm_reg_field 是否可以进行 field 级别的读写操作。

18.1.2. 逻辑上比较独立的数据单位：uvm_reg

uvm_reg 是逻辑上比较独立的一个数据单位，一个 uvm_reg 中至少包含一个 uvm_reg_field。

uvm_reg 的原型为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg

  37 virtual class uvm_reg extends uvm_object;
```

与 uvm_reg_field 一样，它也是派生自 uvm_object。与 uvm_reg_field 不同的是，它在定义中加入了 virtual 关键字，表示它不能直接实例化，而必须派生后，实例化其派生类。这个派生的过程其实就是把 uvm_reg_field 加入到其中的过程。只有加入了 uvm_reg_field，uvm_reg 才是一个有内涵的寄存器，而不只是一个空壳子。

uvm_reg 中比较重要的几个成员变量如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg

  40      local uvm_reg_block      m_parent;
  41      local uvm_reg_file       m_regfile_parent;
  42      local int unsigned       m_n_bits;
  43      local int unsigned       m_n_used_bits;
  44      protected bit            m_maps[uvm_reg_map];
  45      protected uvm_reg_field  m_fields[$];    // Fields in LSB to MSB order

  61      local uvm_object_string_pool
  62          #(uvm_queue #(uvm_hdl_path_concat)) m_hdl_paths_pool;
```

其中的 m_parent 表示此 uvm_reg 所在的 uvm_reg_block，m_regfile_parent 表示

此 uvm_reg 所在的 uvm_reg_file 的指针，m_n_bits 表示此寄存器的位数，m_n_used_bits 表示此寄存器已经用了多少位。这两个变量比较容易混淆。m_n_bits 是在此寄存器 new 的时候指定的，如无特殊情况，此值一般等于系统的位宽，而 m_n_used_bits 则在 new 的时候被初始化为 0，后面当 add_field 的时候，此变量的值会根据 uvm_reg_field 的大小不断的增加。当 m_n_used_bits 大于 m_n_bits 时，表示此 uvm_reg 中所加入的 uvm_reg_field 的位宽和已经超过了限制，会给出警告信息。m_maps 表示此寄存器所在的 uvm_reg_map 信息。这里也是一个联合数组的形式，表示一个寄存器可以位于多个 uvm_reg_map 中，但是一般说来，一个寄存器只属于一个 uvm_reg_map 中。另外这里需要注意的是，这里虽然是一个联合数组，但是用到的仅仅是其索引。uvm_reg_map 用于统一的组织管理 register model 中寄存器的物理地址信息，当使用 frontdoor 操作时，会用到 uvm_reg_map。m_fields 联合数组是 uvm_reg 中最核心的东西。它记录了加入到此 uvm_reg 中所有的 uvm_reg_field 的指针。m_hdl_paths_pool 用于存储此寄存器的 backdoor 操作时用到的路径信息。uvm_object_string_pool 的原型为：

```
文件：src/base/uvm_pool.svh
类：uvm_object_string_pool #(type T=uvm_object)

247 class uvm_object_string_pool #(type T=uvm_object) extends uvm_pool #(string,T);
```

这是一个派生自 uvm_pool 的类。uvm_pool 的本质是一个联合数组，此联合数组的索引是 uvm_pool 的第一个参数，而存储内容是第二个参数。在这里，m_hdl_paths_pool 的本质上是一个联合数组，此联合数组的索引是 string 类型，而存储的内容则是一个队列，此队列的索引是 uvm_hdl_path_concat 类型。uvm_hdl_path_concat 的原型为：

```
文件：src/reg/uvm_reg_model.svh
类：uvm_hdl_path_concat

345 class uvm_hdl_path_concat;
```

它没有派生自任何类，其核心是一个动态数组的成员变量：

```
文件：src/reg/uvm_reg_model.svh
类：uvm_hdl_path_concat

350     uvm_hdl_path_slice slices[];
```

uvm_hdl_path_slice 是 register model 中用于组织 hdl 路径信息的一个结构体：

```
文件：src/reg/uvm_reg_model.svh
类：无

124 typedef struct {
125     string path;
126     int offset;
127     int size;
```

```
128 } uvm_hdl_path_slice;
```

这里的 path 即是真正的路径信息，而 offset 表示一个 uvm_reg_field 的最低位在 uvm_reg 中位置，size 表示一个 uvm_reg_field 的位宽。

18.1.3. 比较大的容器：uvm_reg_block

uvm_reg_block 是 register model 中比较大的数据单位，它用于组织多个寄存器。一般来说，一个 uvm_reg_block 对应的是地址划分中的某一地址范围，它并不代表实际的某个文件。urvm_reg_block 的原型为：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

 38 virtual class uvm_reg_block extends uvm_object;
```

与 uvm_reg 一样，它也是派生自 uvm_object 的，同时也是 virtual 类型的，即必须派生后才能进行实例化。这个派生的过程就是把其它的 uvm_reg_block 和大量的 uvm_reg 加入的过程。

uvm_reg_block 中比较重要的几个成员变量如下：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

 40     local uvm_reg_block   parent;

 43     local int unsigned    blks[uvm_reg_block];
 44     local int unsigned    regs[uvm_reg];

 46     local int unsigned    mems[uvm_mem];
 47     local bit             maps[uvm_reg_map];

 55     local uvm_object_string_pool #(uvm_queue #(string)) hdl_paths_pool;
```

其中的 parent 表示此 uvm_reg_block 所在的 uvm_reg_block，即父 uvm_reg_block，而 blks 则记录了加入到此 uvm_reg_block 的其它 uvm_reg_block 的信息，regs 表示了加入到此 uvm_reg_block 中的其它 uvm_reg 的信息，mems 表示加入到此 uvm_reg_block 中的所有的 uvm_mem 的信息，uvm_mem 表示的是 memory，maps 表示此 uvm_reg_block 所在 uvm_reg_map 信息。hdl_paths_pool 记录了 hdl 路径信息。

18.1.4. 略显单薄的 uvm_reg_file

uvm_reg_file 在整个 register model 中的地位略显单薄。它的原型为：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file

34 virtual class uvm_reg_file extends uvm_object;
```

同 uvm_reg 一样，它也是派生自 uvm_object，且同样的是 virtual 类型的。它其中比较重要的成员变量如下：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file

 36      local uvm_reg_block        parent;
 37      local uvm_reg_file     m_rf;

 39      local uvm_object_string_pool #(uvm_queue #(string)) hdl_paths_pool;
```

其中的 parent 表示此 uvm_reg_file 所在的 uvm_reg_block，m_rf 表示此 uvm_reg_file 所在的 uvm_reg_file，而 hdl_paths_pool 用于记录路径信息。

18.1.5. memory 的模型 uvm_mem

register model 中用于模拟实际的 memory 的是 uvm_mem，其原型为：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem

  40 class uvm_mem extends uvm_object;
```

与 register model 中大部分类一样，它也是派生自 uvm_object。在本书前半部分介绍 register model 时说过，register model 会为每一个寄存器保存镜像值，但是不会为 memory 保存。因此，相对于 uvm_reg_field 来说，其内部并没有存储 memory 数据的数据结构。其中重要的成员变量如下：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem

  47      local string                   m_access;
  48      local longint unsigned     m_size;
  49      local uvm_reg_block          m_parent;
```

```
50      local bit                       m_maps[uvm_reg_map];
51      local int unsigned              m_n_bits;

59      local uvm_object_string_pool
60                      #(uvm_queue #(uvm_hdl_path_concat)) m_hdl_paths_pool;
```

m_access 表示此 memory 是存取策略，m_size 表示 memory 的大小，m_parent 表示此 uvm_mem 所在的 uvm_reg_block，m_maps 表示此 uvm_mem 所在的 uvm_reg_map，m_n_bits 则表示此 memory 每个单元的位宽。与 uvm_reg 一样，m_hdl_paths_pool 是用于存放 hdl 路径信息。

18.1.6. 实现 FRONTDOOR 操作的 uvm_reg_map

uvm_reg_map 负责 register model 中的 FRONTDOOR 操作，所有的对于寄存器或者 memory 的 FRONTDOOR 操作最终都要由 uvm_reg_map 中的相关函数来完成。这个类的原型为：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map

52  class uvm_reg_map extends uvm_object;
```

这也是一个派生自 uvm_object 的类。一般说来，一个 uvm_reg_block 对应一个地址区间，而每一个 uvm_reg_block 都有一个 default_map，此 default_map 完全的表征了此地址区间。uvm_reg_map 类中比较易懂且重要的成员变量如下：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map

57      local uvm_reg_addr_t        m_base_addr;

62      local uvm_reg_adapter       m_adapter;
63      local uvm_sequencer_base    m_sequencer;

66      local uvm_reg_block         m_parent;

70      local uvm_reg_map           m_parent_map;

72      local uvm_reg_addr_t        m_submaps[uvm_reg_map];         // value=offset of submap
at this level

75      local uvm_reg_map_info      m_regs_info[uvm_reg];
76      local uvm_reg_map_info      m_mems_info[uvm_mem];
77
78      local uvm_reg               m_regs_by_offset[uvm_reg_addr_t];
```

其中的 m_base_addr 表示此 uvm_reg_map 所代表的地址空间的基地址，m_adapter 和 m_sequencer 用于 FRONTDOOR 操作时向 driver 发送 item。当某个寄存器要进行读写操作时，首先产生一个 uvm_reg_item 的 uvm_sequence_item，里面记载了要读写的寄存器的信息，读写的类型等等，然后把此 uvm_reg_item 传递给 uvm_reg_map，uvm_reg_map 则根据此 item 产生一个 uvm_reg_bus_op 的操作标志，里面记载了地址信息，读写类型等，之后唤醒 m_adapter，把此操作标志转换成验证平台的 sequencer 能够接受的 transaction 类型，之后通过 m_sequencer 把此 transaction 发送给 driver。

18.1.7. uvm_reg_item 与 uvm_reg_bus_op

uvm_reg_item 的原型及重要的成员变量如下：

```
文件：src/reg/uvm_reg_item.svh
类：uvm_reg_item

 41 class uvm_reg_item extends uvm_sequence_item;

 49    uvm_elem_kind_e element_kind;

 58    uvm_object element;

 65    rand uvm_access_e kind;

 73    rand uvm_reg_data_t value[];

 84    rand uvm_reg_addr_t offset;

 92    uvm_status_e status;

103    uvm_reg_map local_map;

112    uvm_reg_map map;

119    uvm_path_e path;

126    rand uvm_sequence_base parent;

134    int prior = -1;

142    rand uvm_object extension;
       …
239 endclass
```

uvm_reg_item 本质上是一个 uvm_sequence_item，其中的 element_kind 表示产生

这个 uvm_reg_item 的是一个 uvm_reg，还是一个 uvm_reg_field 或者是一个 memory 等 而 element 则表示产生这个 uvm_reg_item 的 uvm_reg 或 uvm_reg_field 或 memory 的指针。kind 表示这是一个读操作还是一个写操作，value 在读操作时表示从 DUT 读出来的值，而在写操作时表示要写入 DUT 的值，offset 主要是用于 memory 的读写操作中，表示地址偏移。status 则是在读写操作完成时用于指示读写操作的状态，local_map 表示 uvm_reg，uvm_reg_field 或者 uvm_mem 所在的 uvm_reg_map 的指针，根据 local_map，可以得到 uvm_reg，uvm_reg_field 的地址，可以得到 uvm_mem 的基地址，map 则表示调用 write 和 read 任务时，传入的一个参数，一般来说，这个参数为 null，系统最终会让 map 的值等于 local_map 的值，path 表示这个操作是 BACKDOOR 还是 FRONTDOOR。如果是 BACKDOOR 操作，那么 uvm_reg，uvm_reg_field 或者 uvm_mem 自行完成，如果是 FRONTDOOR 操作，那么最终会由 uvm_reg_map 来完成。在进行 FRONTDOOR 操作时要启动一个 sequence，parent 就表示此 sequence 的指针。prior 表示产生 item 的优先级，关于优先级的问题曾经在 sequence 机制中提及过。在实际的应用中，这个变量一般不会使用。extension 一般不会用到，可以忽略不计。

在进行 FRONTDOOR 操作时，一个 uvm_reg_item 会转换成 uvm_reg_bus_op，其定义为：

```
文件：src/reg/uvm_reg_item.svh
类：无

257 typedef struct {

263    uvm_access_e kind;

270    uvm_reg_addr_t addr;

279    uvm_reg_data_t data;

287    int n_bits;

303    uvm_reg_byte_en_t byte_en;

311    uvm_status_e status;
312
313 } uvm_reg_bus_op;
```

kind 用于区分读写操作，addr 表示要操作的地址，data 表示读写回来的数据，n_bits 则表示 data 中的有效位数，byte_en 主要是用于按 byte 访问时，status 表征读写操作是否成功，这个状态最终会传递回 uvm_reg_item。

18.2. 模型的建立

18.2.1. 把 uvm_reg_field 加入到 uvm_reg 中

以例子来进行介绍当把 uvm_reg_field 加入到 uvm_reg 中时，系统内部都做了哪些工作：

```
class my_reg extends uvm_reg;
    rand uvm_reg_field data;
    virtual function void build();
        data = uvm_reg_field::type_id::create("data");
        // parameter: parent, size, lsb_pos, access, volatile, reset value, has_reset, is_rand, indivi
dually accessible
        data.configure(this, 16, 0, "RW", 1, 0, 1, 1, 0);
    endfunction
    `uvm_object_utils(my_reg)
    function new(input string name="unnamed_my_reg");
        //parameter: name, size, has_coverage
        super.new(name, 16, UVM_NO_COVERAGE);
    endfunction
endclass
```

这个例子是前面中出现的一个例子，首先来看 new 函数，调用了 uvm_reg 的 new，传入了三个参数，分别是 name，16 和 UVM_NO_COVERAGE。uvm_reg 的 new 函数为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：new

1151 function uvm_reg::new(string name="", int unsigned n_bits, int has_coverage);
1152     super.new(name);
1153     if (n_bits == 0) begin
1154         `uvm_error("RegModel", $sformatf("Register \"%s\" cannot have 0 bits", get_name
()));
1155         n_bits = 1;
1156     end
1157     m_n_bits          = n_bits;
1158     m_has_cover       = has_coverage;
1159     m_atomic          = new(1);
1160     m_n_used_bits = 0;
1161     m_locked          = 0;
1162     m_is_busy         = 0;
1163     m_is_locked_by_field = 1'b0;
```

```
1164      m_hdl_paths_pool = new("hdl_paths");
1165
1166      if (n_bits > m_max_size)
1167          m_max_size = n_bits;
1168
1169  endfunction: new
```

1153 行 1156 行检测输入的第二个参数是否合法，由于一个寄存器至少需要有 1bit，所以输入为 0 时是非法的。1157 行一直到最后是给内部的一些成员变量赋值。这里需要关注的就是 m_n_bits 变量，它记载了这个寄存器的位宽。1166 行到 1167 行是给 m_max_size 赋值，这是一个静态成员变量：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg

   59      local static int unsigned m_max_size = 0;
```

它标志着系统中最大的寄存器的宽度。

把 uvm_reg_field 加入到 uvm_reg 中是通过在 uvm_reg 的 build 中调用 uvm_reg_field 的 configure 函数来实现的。在上面的 build 中，先对 data 进行实例化，之后调用 configure 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：configure

 774  function void uvm_reg_field::configure(uvm_reg            parent,
 775                                         int unsigned      size,
 776                                         int unsigned      lsb_pos,
 777                                         string            access,
 778                                         bit               volatile,
 779                                         uvm_reg_data_t    reset,
 780                                         bit               has_reset,
 781                                         bit               is_rand,
 782                                         bit               individually_accessible);
 783      m_parent = parent;
 784      if (size == 0) begin
 785          `uvm_error("RegModel",
 786              $sformatf("Field \"%s\" cannot have 0 bits", get_full_name()));
 787          size = 1;
 788      end
 789
 790      m_size          = size;
 791      m_volatile    = volatile;
 792      m_access        = access.toupper();
 793      m_lsb           = lsb_pos;
 794      m_cover_on    = UVM_NO_COVERAGE;
 795      m_written     = 0;
 796      m_check         = UVM_CHECK;
 797      m_individually_accessible = individually_accessible;
```

```
798
799     if (has_reset)
800         set_reset(reset);
801     else
802         uvm_resource_db#(bit)::set({"REG::", get_full_name()},
803                                    "NO_REG_HW_RESET_TEST", 1);
804
805     m_parent.add_field(this);
806
```

这个 configure 函数首先对一些成员变量进行赋值。799 行到 803 行进行 reset 设置。这里用到了 set_reset 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：set_reset

1314 function void uvm_reg_field::set_reset(uvm_reg_data_t value,
1315                                        string kind = "HARD");
1316     m_reset[kind] = value & ((1<<m_size) - 1);
1317 endfunction: set_reset
```

它主要就是在 m_reset 联合数组中插入一条记录，记录的索引是“HARD”。一般情况下都会只有这一种 reset。

在没有 reset 情况下，802 行将会向 uvm_resource_pool 中写入一条记录，表明此 uvm_reg_field 没有 reset。

805 行调用 add_field 函数，这里的 m_parent 指的就是此 uvm_reg_field 所在的 uvm_reg。uvm_reg 的 add_field 函数如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：add_field

1187 function void uvm_reg::add_field(uvm_reg_field field);
1188     int offset;
1189     int idx;
1190
1191     if (m_locked) begin
1192         `uvm_error("RegModel", "Cannot add field to locked register model");
1193         return;
1194     end
1195
1196     if (field == null) `uvm_fatal("RegModel", "Attempting to register NULL field");
1197
1198     // Store fields in LSB to MSB order
1199     offset = field.get_lsb_pos();
1200
1201     idx = -1;
1202     foreach (m_fields[i]) begin
```

```
1203            if (offset < m_fields[i].get_lsb_pos()) begin
1204                int j = i;
1205                m_fields.insert(j, field);
1206                idx = i;
1207                break;
1208            end
1209        end
1210        if (idx < 0) begin
1211            m_fields.push_back(field);
1212            idx = m_fields.size()-1;
1213        end
1214
1215        m_n_used_bits += field.get_n_bits();
1216
1217        // Check if there are too many fields in the register
1218        if (m_n_used_bits > m_n_bits) begin
1219            `uvm_error("RegModel",
1220                $sformatf("Fields use more bits (%0d) than available in register \"%s\" (%0d)",
1221                    m_n_used_bits, get_name(), m_n_bits));
1222        end
1223
```

函数相对来说比较复杂。1191 行检测此 register 是否已经被 lock 住了。所谓的被 lock 的意思就是此 reg 不能再加入任何 field 了，也就是说此 reg 的结构已经固定了。一般的，当整个的 register model 建立完成后，会在最顶层的 uvm_reg_block 调用 lock 相关函数，把整个模型锁起来。

1196 行检查 field 的有效性。1199 行调用此 uvm_reg_field 的 get_lst_pos 函数，这个函数比较简单，如其名字所述，返回此 uvm_reg_field 的最低位在 uvm_reg 中的位置。我们什么时候设置了这个值？在 uvm_reg_field 的 configure 的 793 行，把输入的 lsb 参数赋值给了 m_lsb。

1201 到 1213 行则是向 m_fields 中插入要加入的 field。如果此前没有加入进去过，那么就直接加入。如果此前已经有 uvm_reg_field 加入了，且是顺序加入的，那么 1210 行的条件将会满足。这里的所谓的顺序加入是按照 lsb 从小到大的顺序加入，如下所示：

```
fieldA.configure(this, 2, 0, “RW”, 1, 0, 1, 1, 1);
fieldB.configure(this, 3, 2, “RW”, 1, 0, 1, 1, 1);
fieldC.configure(this, 4, 5, “RW”, 1, 0, 1, 1, 1);
```

假如不是顺序加入的，如下所示：

```
fieldC.configure(this, 4, 5, “RW”, 1, 0, 1, 1, 1);
fieldA.configure(this, 2, 0, “RW”, 1, 0, 1, 1, 1);
fieldB.configure(this, 3, 2, “RW”, 1, 0, 1, 1, 1);
```

fieldC 的 lsb 为 4，加入 fieldA 和 fieldB 时，将会执行 1205 行的语句。因此，经过 1201 到 1213 行，m_fields 中的记录是按照 lsb 从小到大的顺序排列的，且 idx 的

值将会是此 field 在 m_fields 中的序号数，即 m_fields[idx]就是刚刚加入的 field。

1215 行得到此 reg 中所有已经加入的 uvm_reg_field 的位宽之和。1218 行检查刚刚加入的这个 field 是否导致位宽和超出了此 uvm_reg 的大小。在上面的 fieldA，fieldB，fieldC 顺序加入的例子中，当 fieldA 加入完成后，m_n_used_bits 为 2，fieldB 加入完成后，此值为 5，而 fieldC 加入后，此值为 9。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：add_field

1224      // Check if there are overlapping fields
1225      if (idx > 0) begin
1226          if (m_fields[idx-1].get_lsb_pos() +
1227              m_fields[idx-1].get_n_bits() > offset) begin
1228              `uvm_error("RegModel", $sformatf("Field %s overlaps field %s in register \"%s\"
",
1229                                                  m_fields[idx-1].get_name(),
1230                                                  field.get_name(), get_name()));
1231          end
1232      end
1233      if (idx < m_fields.size()-1) begin
1234          if (offset + field.get_n_bits() >
1235              m_fields[idx+1].get_lsb_pos()) begin
1236              `uvm_error("RegModel", $sformatf("Field %s overlaps field %s in register \"%s\"
",
1237                                                  field.get_name(),
1238                                                  m_fields[idx+1].get_name(),
1239                                                get_name()));
1240          end
1241      end
1242  endfunction: add_field
```

1225 到 1241 行检查所加入的 field 是否有位重叠现象。其中 1232 行之前检查是否与比较 lsb 小的 field 重合，在上面的乱序例子中就是检查 fieldB 是否与 fieldA 的位重叠；1233 行之后检查是否与比其 lsb 大的 field 重合，在上面的乱序例子中就是检查 fieldB 是否与 fieldC 重叠。

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：configure

 807      if (!m_policy_names.exists(m_access)) begin
 808          `uvm_error("RegModel", {"Access policy '",access,
 809           "' for field '",get_full_name(),"' is not defined. Setting to RW"})
 810          m_access = "RW";
 811      end
 812
 813      if (size > m_max_size)
 814          m_max_size = size;
```

```
815
816     // Ignore is_rand if the field is known not to be writeable
817     // i.e. not "RW", "WRC", "WRS", "WO", "W1", "WO1"
818     case (access)
819       "RO", "RC", "RS", "WC", "WS",
820         "W1C", "W1S", "W1T", "W0C", "W0S", "W0T",
821         "W1SRC", "W1CRS", "W0SRC", "W0CRS", "WSRC", "WCRS",
822         "WOC", "WOS": is_rand = 0;
823     endcase
824
825     if (!is_rand)
826       value.rand_mode(0);
827
828 endfunction: configure
```

回到 uvm_reg_field 的 configure 函数，807 行到 811 行检测输入的关于此 uvm_reg_field 的存储策略是否合法。UVM 中内建了 25 种策略，这基本上代表了大部分的需求。如果这些不能满足，那么就需要自己建立了。

813 行到 814 行给 m_max_size 进行赋值：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field

 63     local static int m_max_size = 0;
```

m_max_size 也是一个静态的变量，它表示整个 register model 中最大的 uvm_reg_field 的位宽。这一点与 uvm_reg 中的 m_max_size 相似。

818 行到 826 行设置此 uvm_reg_field 是否可以进行随机化。如果定义了类似于 RO 这种存取策略，那么是不能进行随机化的。如果不能进行随机化，那么 826 行 826 行把 value 的随机化功能关闭。

因此，uvm_reg_field 的 configure 的执行效果就是把相关成员变量赋值，如 m_lsb，reset 等，另外它还调用了 uvm_reg 的 add_field 函数，把此 field 加入到了 uvm_reg 的 m_fields 数组中。

18.2.2. 把 uvm_reg 加入到 uvm_reg_block 中

以一个例子来介绍：

```
class my_regmodel extends uvm_reg_block;
    rand my_reg version;
    function void build();
        default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN);
        version = my_reg::type_id::create("version", , get_full_name());
```

```
            version.configure(this, null, "version");
            version.build();
            default_map.add_reg(version, 16'h47, "RW");
        endfunction
        `uvm_object_utils(my_regmodel)
        function new(input string name="unnamed_my_regmodel");
            super.new(name, UVM_NO_COVERAGE);
        endfunction
endclass
```

首先要注意的就是 new 函数，这里调用了 uvm_reg_block 的 new 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：new

 954 function uvm_reg_block::new(string name="", int has_coverage=UVM_NO_COVERAGE);
 955     super.new(name);
 956     hdl_paths_pool = new("hdl_paths");
 957     this.has_cover = has_coverage;
 958     // Root block until registered with a parent
 959     m_roots[this] = 0;
 960 endfunction: new
```

函数比较简单，完成一些基本的初始化操作。需要注意的是 959 行出现了 m_roots 联合数组：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

  42     local static bit        m_roots[uvm_reg_block];
```

这个联合数组是一个静态的成员变量，它记载了系统中所有例化的 uvm_reg_block 的指针。如果一个 blk 最终不是最顶层的 uvm_reg_block，那么 add_block 函数将会把其从 m_roots 中删除。另外在 lock_model 函数中将会检测此 uvm_reg_block 是否是最顶层的 uvm_reg_block，如果是的话，就会把此其值设置为 1，后面会详细介绍。

把 uvm_reg 加入到 uvm_reg_block 中，最关键的就调用 uvm_reg 的 configure 函数，上面的例子就是调用 version 的 configure 函数。后面还调用了 version 的 build 函数，前面已经介绍过了，其中主要就是把此 uvm_reg 的 uvm_reg_field 实例化，并且加入到此 uvm_reg 中。uvm_reg 的 configure 函数如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：configure

1174 function void uvm_reg::configure (uvm_reg_block blk_parent,
1175                                   uvm_reg_file regfile_parent=null,
```

```
1176                                                    string hdl_path = "");
1177      m_parent = blk_parent;
1178      m_parent.add_reg(this);
1179      m_regfile_parent = regfile_parent;
1180      if (hdl_path != "")
1181        add_hdl_path_slice(hdl_path, -1, -1);
1182  endfunction: configure
```

1177 行设置 m_parent 值，1178 行调用 uvm_reg_block 的 add_reg 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：add_reg

 995  function void uvm_reg_block::add_reg(uvm_reg rg);
 996      if (this.is_locked()) begin
 997          `uvm_error("RegModel", "Cannot add register to locked block model");
 998          return;
 999      end
1000
1001      if (this.regs.exists(rg)) begin
1002          `uvm_error("RegModel", {"Register '",rg.get_name(),
1003              "' has already been registered with block '",get_name(),"'"})
1004          return;
1005      end
1006
1007      regs[rg] = id++;
1008  endfunction: add_reg
```

996 行到 999 行检测是否已经被 lock 住，被 lock 后是不能再向 uvm_reg_block 中加入 uvm_reg 的，恰如一个 uvm_reg 被 lock 后是不能往其中加入 uvm_reg_field 的。

1001 行到 1005 行避免重复加入，像下面这样，如果连续两次调用 configure 函数，那么就会给出出错提示的：

```
version.configure(this, null, "version");
version.configure(this, null, "version");
```

1007 行向 regs 中插入一条记录，记录的索引是要加入的 uvm_reg，而记录的内容则是 id：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

  65      local static int id = 0;
```

id 是一个静态的变量，这表明所有的 uvm_reg_block 的实例共用这一个变量。所以 id 的值就表示 register model 中所有已经加入到 uvm_reg_block 的 uvm_reg 的数量。

回到 uvm_reg 的 configure 函数，1179 行设置 m_regfile_parent 变量值，1180 到 1181 行则根据输入的 hdl_path 的情况决定是否调用 add_hdl_path_slice 函数。一般的，假如整个寄存器中只有一个字段的话，会在调用这个寄存器的 configure 函数时指定此寄存器的 hdl 路径，这个路径本质上是其中的 uvm_reg_field 的路径。但是当一个寄存器中有多个 uvm_reg_field 时，由于每个 field 都有各自的路径，即有多个路径，而在 configure 的参数中只能指定一个参数，这种情况下就需要手工调用 add_hdl_path_slice 函数。add_hdl_path_slice 函数的定义如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：add_hdl_path_slice

1365 function void uvm_reg::add_hdl_path_slice(string name,
1366                                           int offset,
1367                                           int size,
1368                                           bit first = 0,
1369                                           string kind = "RTL");
1370     uvm_queue #(uvm_hdl_path_concat) paths = m_hdl_paths_pool.get(kind);
1371     uvm_hdl_path_concat concat;
1372
1373     if (first || paths.size() == 0) begin
1374         concat = new();
1375         paths.push_back(concat);
1376     end
1377     else
1378         concat = paths.get(paths.size()-1);
1379
1380   concat.add_path(name, offset, size);
1381 endfunction
```

1370 行用到了 uvm_pool 的 get 函数：

```
文件：src/base/uvm_pool.svh
类：uvm_pool
函数/任务：get

 86   virtual function T get (KEY key);
 87     if (!pool.exists(key)) begin
 88       T default_value;
 89       pool[key] = default_value;
 90     end
 91     return pool[key];
 92   endfunction
```

get 函数将会返回 uvm_pool 中索引为 kind 的一条记录。一般说来，在调用 add_hdl_path_slice 时，最后一个参数 kind 都不会传入任何值，所以 m_hdl_paths_pool 中其中只有一条记录，这条记录的索引就是”RTL”，而记录的内容是一个队列。

1173 行根据 first 的值与刚才得到的队列中记录的数量来决定是否向此队列中新

加入一条记录。假如这是我们这一次调用 add_hdl_path_slice，那么很明显队列中是没有任何内容的，将会执行这 1374 行 1375 行的分支；假如这是我们第二次调用，且调用时传入的 firtst 为默认的 0，那么将会执行 1378 行的分支，把队列中的最后一条记录取出；假如这是我们第二次调用，且传入的 first 的值为 1，那么将会执行 1174 行的分支，向队列中新插入一条记录。因此，如果所有的调用中，first 的值为 0，那么队列中记录的数量一直为 1，即只有一个 uvm_hdl_path_concat 的实例，否则将会多于 1 个。

1380 行向 concat 中加入 hdl 路径。这里用到了 add_path 函数：

```
文件：src/reg/uvm_reg_model.svh
类：uvm_hdl_path_concat
函数/任务：add_path

368     function void add_path(string path,
369                            int unsigned offset = -1,
370                            int unsigned size = -1);
371         uvm_hdl_path_slice t;
372         t.offset = offset;
373         t.path   = path;
374         t.size   = size;
375
376         add_slice(t);
377     endfunction
```

371 到 374 行实例化一个 uvm_hdl_path_slice 类型变量 t，376 行调用 add_slice 函数：

```
文件：src/reg/uvm_reg_model.svh
类：uvm_hdl_path_concat
函数/任务：add_slice

360     function void add_slice(uvm_hdl_path_slice slice);
361         slices = new [slices.size()+1] (slices);
362         slices[slices.size()-1] = slice;
363     endfunction
```

slices 是一个动态数组，这里的意思就是把此动态数组的大小加 1，且保持原来的数据不变，并新插入一条记录。

因此，整个 add_hdl_path_slice 的效果就是向 m_hdl_paths_pool 中索引为”RTL” 的记录对应的队列中的最后一条记录的 uvm_hdl_path_concat 的实例的动态数组中插入一条记录，这条记录记载了此 uvm_reg_field 的大小，lsb 及路径。假如一个寄存器中有多个字段，那么通常采用如下的方式在 uvm_reg_block 中来加载 hdl 路径信息：

```
tf_reg.configure(this, null, “”);
tf_reg.build();
tf_reg.fieldA.configure(tf_reg, 2, 0, “RW”, 1, 0, 1, 1, 1);
tf_reg.add_hdl_path_slice(“fieldA”, 0, 2);
```

```
tf_reg.fieldB.configure(tf_reg, 3, 2, “RW”, 1, 0, 1, 1, 1);
tf_reg.add_hdl_path_slice(“fieldB”, 2, 3);
tf_reg.fieldC.configure(tf_reg, 4, 5, “RW”, 1, 0, 1, 1, 1);
tf_reg.add_hdl_path_slice(“fieldC”, 5, 4);
```

在这种添加方式中，直接调用 add_hdl_path_slice 函数。经过这样三次调用后，m_hdl_paths_pool 中的记录只有一条，其索引为”RTL”。这条记录对应的队列中也只有一条记录，即只有一个 uvm_hdl_path_concat 的实例，但是这个 uvm_hdl_path_concat 的实例的动态数组中有了三条记录。

小结一下，uvm_reg 的 configure 函数除了设置一些 uvm_reg 的成员变量值之外，做的第一件事情就是在 uvm_reg_block 的 regs 数组中插入了一条记录，记录的索引是这个 uvm_reg 的指针，记录的内容是 register model 中已经加入到 uvm_reg_block 的 uvm_reg 的数量；做的第二件事情就是在 hdl_path 不为 null 的情况下，向 m_hdl_paths_pool 中索引为”RTL”的记录对应的队列中的最后一条记录的 uvm_hdl_path_concat 的实例的动态数组中插入一条记录，这条记录记载了此 uvm_reg_field 的大小，lsb 及路径。

18.2.3. 把 uvm_reg 加入到 uvm_reg_map 中

上节中只是谈了如何调用 uvm_reg 的 configure 函数把 uvm_reg 加入到 uvm_reg_block 中，而忽略了关于 uvm_reg_map 的代码。本节讲述如何把 uvm_reg 加入到 uvm_reg_map 中。每一个 uvm_reg_block 都至少对应一个（且一般也只对应一个）uvm_reg_map，这个 map 被称为 default_map：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

 168      uvm_reg_map  default_map;
```

在 uvm_reg_block 的 build 中，首先需要把 default_map 实例化：

```
default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN);
```

这里调用了 create_map 函数，这是 uvm_reg_block 的一个成员函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：create_map

1793 function uvm_reg_map uvm_reg_block::create_map(string name,
1794                                                uvm_reg_addr_t base_addr,
1795                                                int unsigned n_bytes,
1796                                                uvm_endianness_e endian,
```

```
1797                                                        bit byte_addressing=1);
1798
1799      uvm_reg_map    map;
1800
1801      if (this.locked) begin
1802          `uvm_error("RegModel", "Cannot add map to locked model");
1803          return null;
1804      end
1805
1806      map = uvm_reg_map::type_id::create(name,,this.get_full_name());
1807      map.configure(this,base_addr,n_bytes,endian,byte_addressing);
1808
1809      this.maps[map] = 1;
1810      if (maps.num() == 1)
1811        default_map = map;
1812
1813      return map;
1814  endfunction
```

1801 行检查此 uvm_reg_block 是否已经被 lock 住了 ，被 lock 住的 uvm_reg_block 是不能实例化 default_map 的。

1806 行实例化一个 uvm_reg_map，1807 行调用 uvm_reg_map 的 configure 函数，传入的参数就是 create_map 的参数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：configure

 609  function void uvm_reg_map::configure(uvm_reg_block       parent,
 610                                       uvm_reg_addr_t     base_addr,
 611                                       int unsigned       n_bytes,
 612                                       uvm_endianness_e   endian,
 613                                       bit                byte_addressing=1);
 614      m_parent        = parent;
 615      m_n_bytes       = n_bytes;
 616      m_endian        = endian;
 617      m_base_addr     = base_addr;
 618      m_byte_addressing = byte_addressing;
 619  endfunction: configure
```

函数比较简单，只是单纯的给此 map 的一些内部的成员变量赋值。回到 create_map 函数，1809 行把刚刚实例化并且配置好的 uvm_reg_map 加入到 maps 联合数组中 1810 行检查此 uvm_reg_block 中已经实例化的 uvm_reg_map 的数量，如果只有一个那么就把这个赋值给 default_map。从这里也可以看出来，在上面的调用中完全可以这样做：

```
create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN);
```

这样系统也会把新实例化的 uvm_reg_map 赋值给 default_map。

当一个 map 实例化完成后，那么就可以向其中添加 uvm_reg：

```
default_map.add_reg(version, 16'h47, "RW");
```

这里用到了 add_reg 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：add_reg

 624 function void uvm_reg_map::add_reg(uvm_reg rg,
 625                                     uvm_reg_addr_t offset,
 626                                     string rights = "RW",
 627                                     bit unmapped=0,
 628                                     uvm_reg_frontdoor frontdoor=null);
 629
 630   if (m_regs_info.exists(rg)) begin
 631     `uvm_error("RegModel", {"Register '",rg.get_name(),
 632                "' has already been added to map '",get_name(),"'"})
 633     return;
 634   end
 635
 636   if (rg.get_parent() != get_parent()) begin
 637     `uvm_error("RegModel",
 638       {"Register '",rg.get_full_name(),"' may not be added to address map '",
 639        get_full_name(),"' : they are not in the same block"})
 640     return;
 641   end
 642
 643   rg.add_map(this);
 644
 645   begin
 646   uvm_reg_map_info info = new;
 647   info.offset     = offset;
 648   info.rights     = rights;
 649   info.unmapped = unmapped;
 650   info.frontdoor = frontdoor;
 651   m_regs_info[rg] = info;
 652   end
 653 endfunction
```

630 行检查此 reg 是否已经加入到此 uvm_reg_block 中了，636 行则用于检查要加入的 uvm_reg 与此 map 的 parent block 是否是同一个 uvm_reg_block。只有属于同一个 uvm_reg_block，才能加入。

643 行调用 uvm_reg 的 add_map 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：add_map

1554 function void uvm_reg::add_map(uvm_reg_map map);
```

```
1555     m_maps[map] = 1;
1556  endfunction
```

函数相当简单，只是在 uvm_reg 的 m_maps 中加入了一条记录。

645 到 652 行实例化一个 uvm_reg_map_info 的变量，并把其加入到 m_regs_info 联合数组中。因此，m_regs_info 联合数组的索引就是所有加入到此 uvm_reg_block 的 uvm_reg 实例的指针，而内容则是对应此 uvm_reg 实例的信息，如地址，存取策略等信息。注意的是，这里的地址只是一个相对地址，即相对此 uvm_reg_map 的基地址的偏移地址。

因此，uvm_reg_map 的 add_reg 函数的效果一是把 map 的信息写入到了待加入的 uvm_reg 的 m_maps 数组中，二是把此 reg 的信息加入到了此 map 的 m_regs_info 数组中。

18.2.4. 把 uvm_mem 加入到 uvm_reg_block 中

同样的，以一个例子来说明如何把一块 memory 加入到 uvm_reg_block 中：

```
class my_memory extends uvm_mem;
    function new(string name);
        super.new(name, 1024, 16);
    endfunction
    `uvm_object_utils(my_memory)
endclass
class my_block extends uvm_reg_block;
    my_memory mm;
    …
    function void build();
        …
        mm = my_memory::type_id::create("mm", , get_full_name());
        mm.configure(this, "top_tb.stat.counter.memory");
        default_map.add_mem(mm, 'h0);
        …
    endfunction
    …
endclass
```

在 my_memory 的 new 函数中，调用了 uvm_mem 的 new 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：new

 934 function uvm_mem::new (string                 name,
 935                        longint unsigned size,
```

```
936                                 int unsigned        n_bits,
937                                 string              access = "RW",
938                                 int                 has_coverage = UVM_NO_COVERAGE);
939
940     super.new(name);
941     m_locked = 0;
942     if (n_bits == 0) begin
943         `uvm_error("RegModel", {"Memory '",get_full_name(),"' cannot have 0 bits"})
944         n_bits = 1;
945     end
946     m_size          = size;
947     m_n_bits        = n_bits;
948     m_backdoor      = null;
949     m_access        = access.toupper();
950     m_has_cover     = has_coverage;
951     m_hdl_paths_pool = new("hdl_paths");
952
953     if (n_bits > m_max_size)
954         m_max_size = n_bits;
955
956 endfunction: new
```

942 行检查输入的 n_bits 的合法性，保证一个 memory 的单元的位宽至少为 1。946 到 951 行给相关的变量赋值。经过赋值后，m_size 存放的就是此 memory 一共有多少个单元，而 m_n_bits 则表示每个单元的数量。954 行的 m_max_size 与 uvm_reg_field 和 uvm_reg 的 m_max_size 类似，这里表示系统中所有 memory 中最大的位宽：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem

  62     local static int unsigned   m_max_size = 0;
```

在 my_block 的 build 函数中，首先需要对 mm 进行实例化，之后调用 uvm_mem 的 configure 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：configure

 961 function void uvm_mem::configure(uvm_reg_block   parent,
 962                                    string           hdl_path="");
 963
 964     if (parent == null)
 965       `uvm_fatal("REG/NULL_PARENT","configure: parent argument is null")
 966
 967     m_parent = parent;
 968
 969     if (m_access != "RW" && m_access != "RO") begin
 970         `uvm_error("RegModel", {"Memory '",get_full_name(),"' can only be RW or RO"})
```

```
971         m_access = "RW";
972     end
973
974     begin
975         uvm_mem_mam_cfg cfg = new;
976
977         cfg.n_bytes      = ((m_n_bits-1) / 8) + 1;
978         cfg.start_offset = 0;
979         cfg.end_offset   = m_size-1;
980
981         cfg.mode     = uvm_mem_mam::GREEDY;
982         cfg.locality = uvm_mem_mam::BROAD;
983
984         mam = new(get_full_name(), cfg, this);
985     end
986
987     m_parent.add_mem(this);
988
989     if (hdl_path != "") add_hdl_path_slice(hdl_path, -1, -1);
990 endfunction: configure
```

964 行保证 parent 不为 null。967 行给 m_parent 赋值，969 到 972 行保证 memory 的存取策略只能是 RW 或者 RO。

974 到 985 行用于 uvm_vreg 的相关操作，平时一般较少用到。

987 行调用 uvm_reg_block 的 add_mem 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：add_mem

1030 function void uvm_reg_block::add_mem(uvm_mem mem);
1031     if (this.is_locked()) begin
1032         `uvm_error("RegModel", "Cannot add memory to locked block model");
1033         return;
1034     end
1035
1036     if (this.mems.exists(mem)) begin
1037         `uvm_error("RegModel", {"Memory '",mem.get_name(),
1038             "' has already been registered with block '",get_name(),"'"})
1039          return;
1040     end
1041     mems[mem] = id++;
1042 endfunction: add_mem
```

1031 到 1034 行保证此 uvm_reg_block 还没有被 lock。

1036 行检查此 memory 是已经加入到此 uvm_reg_block 中，避免重复加入。1041 行向 mems 数组中插入一条记录，记录的索引是要加入的 uvm_mem 的指针，而内容

则是 id。前面介绍过，id 中记录了所有已经加入到 register_model 中的寄存器的数量，因此，这里需要把这种说法稍微纯正一下，id 中记录了所有已经加入到 register model 中的 uvm_reg 和 uvm_mem 的数量。

回到 uvm_mem 的 configure 函数，989 行调用 add_hdl_path_slice 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：add_hdl_path_slice

2178 function void uvm_mem::add_hdl_path_slice(string name,
2179                                           int offset,
2180                                           int size,
2181                                           bit first = 0,
2182                                           string kind = "RTL");
2183     uvm_queue #(uvm_hdl_path_concat) paths=m_hdl_paths_pool.get(kind);
2184     uvm_hdl_path_concat concat;
2185
2186     if (first || paths.size() == 0) begin
2187         concat = new();
2188         paths.push_back(concat);
2189     end
2190     else
2191         concat = paths.get(paths.size()-1);
2192
2193     concat.add_path(name, offset, size);
2194 endfunction
```

这个函数与 uvm_reg 的 add_hdl_path_slice 函数完全一样。因此，add_hdl_path_slice 的效果就是向 m_hdl_paths_pool 中索引为”RTL”的记录对应的队列中的最后一条记录的 uvm_hdl_path_concat 的实例的动态数组中插入一条记录，这条记录记载了此 uvm_mem 的大小，lsb 及路径。

因此，uvm_mem 的 configure 函数除了设置一些 uvm_mem 的成员变量值之外，做的第一件事情就是在 uvm_reg_block 的 mems 数组中插入了一条记录，记录的索引是这个 uvm_mem 的指针，记录的内容是 register model 中已经加入到 uvm_reg_block 的 uvm_reg 和 uvm_mem 的数量；做的第二件事情就是在 hdl_path 不为 null 的情况下，向 m_hdl_paths_pool 中索引为”RTL”的记录对应的队列中的最后一条记录的 uvm_hdl_path_concat 的实例的动态数组中插入一条记录，这条记录记载了此 uvm_mem 的大小，lsb 及路径。

18.2.5. 把 uvm_mem 加入到 uvm_reg_map 中

在 uvm_reg_block 的 build 中除了需要把 uvm_mem 通过调用 configure 函数加入

到 uvm_reg_block 中外，还要调用 uvm_reg_map 的 add_mem 函数把此 uvm_mem 加入到 uvm_reg_block 的 default_map 中：

```
default_map.add_mem(mm, ‘h0);
```

uvm_reg_map 的 add_mem 函数定义为：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：add_mem

 770 function void uvm_reg_map::add_mem(uvm_mem mem,
 771                                       uvm_reg_addr_t offset,
 772                                       string rights = "RW",
 773                                       bit unmapped=0,
 774                                       uvm_reg_frontdoor frontdoor=null);
 775     if (m_mems_info.exists(mem)) begin
 776         `uvm_error("RegModel", {"Memory '",mem.get_name(),
 777                    "' has already been added to map '",get_name(),"'"})
 778         return;
 779     end
 780
 781     if (mem.get_parent() != get_parent()) begin
 782         `uvm_error("RegModel",
 783             {"Memory '",mem.get_full_name(),"' may not be added to address map '",
 784              get_full_name(),"' : they are not in the same block"})
 785         return;
 786     end
 787
 788     mem.add_map(this);
 789
 790     begin
 791     uvm_reg_map_info info = new;
 792     info.offset     = offset;
 793     info.rights     = rights;
 794     info.unmapped = unmapped;
 795     info.frontdoor = frontdoor;
 796     m_mems_info[mem] = info;
 797     end
 798 endfunction: add_mem
```

这个函数与 uvm_reg_map 的 add_reg 非常像，因此简单介绍。788 行调用 uvm_mem 的 add_map 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：add_map

1018 function void uvm_mem::add_map(uvm_reg_map map);
1019    m_maps[map] = 1;
1020 endfunction
```

函数相当简单，只是在 m_maps 中插入一条记录，记录了此 uvm_mem 所从属的 uvm_map 的信息。

790 到 797 行向 m_mems_info 中插入一条记录，这条记录记载了此块 memory 的基地址，存取策略等。

因此，uvm_reg_map 的 add_mem 函数的效果一是把 map 的信息写入到了待加入的 uvm_mem 的 m_maps 数组中，二是把此 uvm_mem 的信息加入到了此 map 的 m_mems_info 数组中。

18.2.6. 把 uvm_reg_file 加入到 uvm_reg_block 中

依然以一个例子来介绍：

```
class regfile extends uvm_reg_file;
    function new(input string name="unnamed_regfile");
        super.new(name);
    endfunction
    `uvm_object_utils(regfile)
endclass
class mac_blk extends uvm_reg_block;
    rand regfile file_a;

    function void build();
        …
        file_a = regfile::type_id::create(“file_a”, , get_full_name());
        file_a.configure(this, null, “fileA”);
    endfunction
endclass
```

要把一个 uvm_reg_file 加入到 uvm_reg_block 中，最关键的是调用 uvm_reg_file 的 configure 函数：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file
函数/任务：configure

236 function void uvm_reg_file::configure(uvm_reg_block blk_parent, uvm_reg_file regfile_parent,
string hdl_path = "");
237       this.parent = blk_parent;
238       this.m_rf = regfile_parent;
239       this.add_hdl_path(hdl_path);
240 endfunction: configure
```

237 行给 parent 赋值，238 行给代表 parent reg file 的 m_rf 赋值，239 行调用 add_hdl_path 函数：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file
函数/任务：add_hdl_path

283 function void uvm_reg_file::add_hdl_path(string path, string kind = "RTL");
284
285    uvm_queue #(string) paths;
286
287    paths = hdl_paths_pool.get(kind);
288
289    paths.push_back(path);
290
291 endfunction
```

287 行从 hdl_paths_pool 中得到一条索引为”RTL”的记录，这条记录的内容是一个队列，队列中存放的是 string 类型。289 行把要加入的 hdl 路径加入到此队列中。注意到这里的队列中存放的是 string 类型，而在 uvm_mem 和 uvm_reg 的 m_hdl_paths_pool 的队列中存放的数据是 uvm_hdl_path_concat 类型的。

因此，uvm_reg_file 的 configure 执行效果就在给一些成员变量赋值，同时往 hdl_paths_pool 的索引为”RTL”的记录对应的队列中插入一条记录，记录的内容就是此 uvm_reg_file 的 hdl 路径。

18.2.7. 把子 uvm_reg_block 加入到父 uvm_reg_block 中

一般来说，常用的 register model 至少是两级的，即有一个顶层的 uvm_reg_block，在此之下，有很多子 uvm_reg_block。因此，这里就牵扯到向 uvm_reg_block 中加入 uvm_reg_block。

```
class register_model extends uvm_reg_block;
    global_blk gb_ins;
    …
    function void build();
        default_map = create_map("default_map", 0, 2, UVM_LITTLE_ENDIAN, 0);
        gb_ins = global_blk::type_id::create("gb_ins", , get_full_name());
        gb_ins.configure(this, "global_reg");
        gb_ins.build();
        gb_ins.lock_model();
        default_map.add_submap(gb_ins.default_map, 16'h0);
        …
    endfunction
    …
```

```
endclass
```

如上所示，把子 uvm_reg_block 加入到父 uvm_reg_block 中需要在父 uvm_reg_block 的 build 函数中：一是调用子 uvm_reg_block 的 configure 函数时，第一个参数要设置为 this，即父 uvm_reg_block 的指针；二是需要调用 uvm_reg_map 的 add_submap 函数。

先来看 uvm_reg_block 的 configure 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：configure

 965 function void uvm_reg_block::configure(uvm_reg_block parent=null, string hdl_path="");
 966    this.parent = parent;
 967    if (parent != null)
 968       this.parent.add_block(this);
 969    add_hdl_path(hdl_path);
 970
 971    uvm_resource_db#(uvm_reg_block)::set(get_full_name(),
 972                                         "uvm_reg::*", this);
 973 endfunction
```

966 行给 parent 赋值，968 行调用 parent uvm_reg_block 的 add_block 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：add_block

 978 function void uvm_reg_block::add_block (uvm_reg_block blk);
 979    if (this.is_locked()) begin
 980       `uvm_error("RegModel", "Cannot add subblock to locked block model");
 981       return;
 982    end
 983    if (this.blks.exists(blk)) begin
 984       `uvm_error("RegModel", {"Subblock '",blk.get_name(),
 985          "' has already been registered with block '",get_name(),"'"})
 986       return;
 987    end
 988    blks[blk] = id++;
 989    if (m_roots.exists(blk)) m_roots.delete(blk);
 990 endfunction
```

979 到 982 行保证在加入子 uvm_reg_block 的时候，父 uvm_reg_block 没有被锁定。983 到 987 行避免重复加入。988 行向 blks 数组中插入一条记录，记录的索引是子 uvm_reg_block，而内容是 id。前面说过，id 表示在执行这句代码前已经加入到 register model 中的 uvm_reg，uvm_mem 数量之和。因此这里需要再对这种说法进行修正一下：id 表示在执行这句代码前已经加入到 register model 中的 uvm_reg，uvm_mem 和 uvm_reg_block 的数量之和。

989 行则用于删除 m_roots 中的关于子 uvm_reg_block 的记录。m_roots 中最终会只存放最顶层的 uvm_reg_block 的指针。因此，当一个子 uvm_reg_block 被加入到父 uvm_reg_block 中时，这个子 uvm_reg_block 就一定不是最顶层的，因此可以直接删除。

configure 函数的 969 行调用 add_hdl_path 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：add_hdl_path

1966 function void uvm_reg_block::add_hdl_path(string path, string kind = "RTL");
1967
1968    uvm_queue #(string) paths;
1969
1970    paths = hdl_paths_pool.get(kind);
1971
1972    paths.push_back(path);
1973
1974 endfunction
```

这个函数与 uvm_reg_file 的 add_hdl_path 完全一样，不重复讲述。

configure 函数的 971 行向 uvm_resource_pool 中写入此个 uvm_reg_block 实例的信息，这样可以保证 register model 中所有的 uvm_reg_block 都可以在 uvm_resource_pool 中找到。

因此，uvm_reg_block 的 configure 函数的执行效果：第一，向 blks 中插入一条记录，记录的索引是加入到此 uvm_reg_block 的子 uvm_reg_block 的指针，内容则是此前已经加入到 register model 中的所有 uvm_reg，uvm_mem，uvm_reg_block 的数量之和；第二，把 m_roots 中关于子 uvm_reg_block 的相关记录删除；第三，向 hdl_paths_pool 的索引为”RTL”的记录对应的队列中插入一条记录，记录的内容就是此 uvm_reg_file 的 hdl 路径。

uvm_reg_map 的 add_submap 函数定义如下：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：add_submap

 903 function void uvm_reg_map::add_submap (uvm_reg_map child_map,
 904                                         uvm_reg_addr_t offset);
 905     uvm_reg_map parent_map;
 906
 907     if (child_map == null) begin
 908         `uvm_error("RegModel", {"Attempting to add NULL map to map '",get_full_name(),"'"})
 909         return;
 910     end
```

```
911
912     parent_map = child_map.get_parent_map();
913
914     // Can not have more than one parent (currently)
915     if (parent_map != null) begin
916         `uvm_error("RegModel", {"Map '", child_map.get_full_name(),
917                    "' is already a child of map '",
918                    parent_map.get_full_name(),
919                    "'. Cannot also be a child of map '",
920                    get_full_name(),
921                    "'"})
922         return;
923     end
924
925     begin : parent_block_check
926       uvm_reg_block child_blk = child_map.get_parent();
927       if (child_blk == null) begin
928           `uvm_error("RegModel", {"Cannot add submap '",child_map.get_full_name(),
929                      "' because it does not have a parent block"})
930           return;
931       end
932       if (get_parent() != child_blk.get_parent()) begin
933           `uvm_error("RegModel",
934             {"Submap '",child_map.get_full_name(),"' may not be added to this ",
935             "address map, '", get_full_name(),"', as the submap's parent block, '",
936             child_blk.get_full_name(),"', is not a child of this map's parent block, '",
937             m_parent.get_full_name(),"'"})
938         return;
939       end
940     end
941
942     begin : n_bytes_match_check
943         if (m_n_bytes > child_map.get_n_bytes(UVM_NO_HIER)) begin
944             `uvm_warning("RegModel",
945                 $sformatf("Adding %0d-byte submap '%s' to %0d-byte parent map '%s'",
946                           m_n_bytes, child_map.get_full_name(),
947                           child_map.get_n_bytes(UVM_NO_HIER), get_full_name()));
948         end
949     end
950
951     child_map.add_parent_map(this,offset);
952
953     set_submap_offset(child_map, offset);
954
955 endfunction: add_submap
```

907 行检查输入的参数的有效性。912 行到 923 行保证 child_map 没有加入到其它的 uvm_reg_map 中。

925 到 940 行一方面保证要加入的 child_map 已经从属于某个 uvm_reg_block 了，另外一方面保证要加入的 child_map 所从属的 uvm_reg_block 和此 map 在同一个

uvm_reg_block 中。

942 到 949 行检查要加入的 child_map 和此 map 的位宽是否一致。如果此 map 的位宽大于 child_map 的位宽，那么给出警告信息。

951 行调用 child_map 的 add_parent_map 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：add_parent_map

 973 function void uvm_reg_map::add_parent_map(uvm_reg_map parent_map, uvm_reg_addr_t off
set);
 974
 975     if (parent_map == null) begin
 976         `uvm_error("RegModel",
 977             {"Attempting to add NULL parent map to map '",get_full_name(),"'"})
 978         return;
 979     end
 980
 981     if (m_parent_map != null) begin
 982         `uvm_error("RegModel",
 983             $sformatf("Map \"%s\" already a submap of map \"%s\" at offset 'h%h",
 984                       get_full_name(), m_parent_map.get_full_name(),
 985                       m_parent_map.get_submap_offset(this)));
 986         return;
 987     end
 988
 989     m_parent_map = parent_map;
 990     m_parent_maps[parent_map] = offset; // prep for multiple parents
 991     parent_map.m_submaps[this] = offset;
 992
 993 endfunction: add_parent_map
```

975 到 979 行保证输入参数的有效性，981 到 987 行保证此 map 只拥有一个 parent map，且一旦设定，不能更改。989 行给 m_parent_map 赋值，990 行向 m_parent_maps 中插入一条记录。从这句后面的注释来看，这是为了后面支持多个 parent map 的情况。在目前的 UVM1.1 版本中，这个数组并没有太多的用处。991 行向 parent map 的 m_submaps 中插入一条记录，记录的索引是此 child map，而内容则是此 child map 相对于 parent map 的偏移地址。

回到 add_submap 函数，953 行调用了 set_submap_offset 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：set_submap_offset

1429 function void uvm_reg_map::set_submap_offset(uvm_reg_map submap, uvm_reg_addr_t offse
t);
1430     if (submap == null) begin
```

```
1431        `uvm_error("REG/NULL","set_submap_offset: submap handle is null")
1432        return;
1433      end
1434      m_submaps[submap] = offset;
1435      if (m_parent.is_locked()) begin
1436        uvm_reg_map root_map = get_root_map();
1437        root_map.Xinit_address_mapX();
1438      end
1439    endfunction
```

1430 到 1433 行保证输入参数的有效性，1434 行向 m_submaps 中插入一条记录，记录的索引是 child_map，而内容是此 child_map 的偏移地址。在之前的 chile_map 的 add_parent_map 中已经做过同样的事情了。两次操作插入的记录完全一样，因此最终 m_submaps 会只被插入一条记录，而不会出错。

1435 行到 1438 行则是用于在 parent uvm_reg_block 已经被 lock 的情况下，进行地址的初始化操作。关于这一点，将会在一下节中详细介绍。

18.3. register model 的锁定

一般的，在最顶层的 uvm_reg_block 的 build 中，当所有的 uvm_reg_block 和 uvm_reg 及 uvm_mem 被加入完毕后，需要调用 lock_model 函数来进行锁定。本节介绍整个 register model 的锁定。

18.3.1. uvm_reg_block 的 lock_model 函数

函数的定义为：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：lock_model

1062  function void uvm_reg_block::lock_model();
1063
1064      if (is_locked())
1065        return;
1066
1067      locked = 1;
```

```
1068
1069      foreach (regs[rg_]) begin
1070          uvm_reg rg = rg_;
1071          rg.Xlock_modelX();
1072      end
1073
1074      foreach (mems[mem_]) begin
1075          uvm_mem mem = mem_;
1076          mem.Xlock_modelX();
1077      end
1078
1079      foreach (blks[blk_]) begin
1080          uvm_reg_block blk=blk_;
1081          blk.lock_model();
1082      end
1083
1084      if (this.parent == null) begin
1085          int max_size = uvm_reg::get_max_size();
1086
1087          if (uvm_reg_field::get_max_size() > max_size)
1088              max_size = uvm_reg_field::get_max_size();
1089
1090          if (uvm_mem::get_max_size() > max_size)
1091              max_size = uvm_mem::get_max_size();
1092
1093          if (max_size > `UVM_REG_DATA_WIDTH) begin
1094              `uvm_fatal("RegModel", $sformatf("Register model requires that UVM_REG_DAT
A_WIDTH be defined as %0d or greater. Currently defined as %0d",          max_size, `UVM_RE
G_DATA_WIDTH))
1095          end
1096
```

1064 行检查此 uvm_reg_block 是否已经 lock 了。如果已经被 lock 了，也即 lock_model 已经被调用过一次了，那么这里将会直接返回。

1067 行把 locked 赋值，表明 lock_model 已经被调用过了。

1069 行到 1072 调用所有加入到此 uvm_reg_block 的 uvm_reg 的 Xlock_modelX 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：Xlock_modelX

1247  function void uvm_reg::Xlock_modelX();
1248      if (m_locked)
1249          return;
1250      m_locked = 1;
1251  endfunction
```

函数比较简单，只是把 m_locked 置为 1。

1074 行到 1077 调用所有加入到此 uvm_reg_block 的 uvm_mem 的 Xlocck_modelX 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：Xlocck_modelX

1025  function  void  uvm_mem::Xlock_modelX();
1026        m_locked  =  1;
1027  endfunction:  Xlock_modelX
```

函数也比较简单，只是把 m_locked 置为 1。

1079 到 1082 行递归的调用所有加入到此 uvm_reg_block 的子 uvm_reg_block 的 lock_model 函数。

1084 行判断此 uvm_reg_block 的 parent 是否为 null。这其实相当于在判断这个 uvm_reg_block 是不是最顶层的 uvm_reg_block。只有最顶层的 uvm_reg_block，其 parent 才为 null。因此，分析 1085 到 1089 行时一定要注意，这段代码只有在最顶层的 uvm_reg_block 的 lock_model 中才会执行。

1085 行到 1095 行判断系统中所有的 uvm_reg，uvm_reg_field，uvm_mem 的位宽是否超出了 UVM_REG_DATA_WIDTH 的限制。可以通过重定义这个宏给扩展 register model 所能容忍的最大位宽。默认情况下，此值为 64，一般来说已经足够使用了。

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：lock_model

1097          Xinit_address_mapsX();
1098
1099          // Check that root register models have unique names
1100
1101          // Has this name has been checked before?
1102          if (m_roots[this] != 1) begin
1103              int n = 0;
1104
1105              foreach (m_roots[_blk]) begin
1106                  uvm_reg_block blk = _blk;
1107
1108                  if (blk.get_name() == get_name()) begin
1109                      m_roots[blk] = 1;
1110                      n++;
1111                  end
1112              end
1113
1114              if (n > 1) begin
1115                  `uvm_error("UVM/REG/DUPLROOT",
1116                             $sformatf("There are %0d root register models named \"%s\". Th
```

```
e names of the root register models have to be unique",
1117                                                    n, get_name()))
1118                end
1119            end
1120        end
1121
1122  endfunction: lock_model
```

1097 行调用 Xinit_address_mapsX 函数，这是一个比较复杂的函数，将在下节中介绍。

1102 到 1119 行则用于判断 m_roots 中最顶层的 uvm_reg_block 是否重名。这里的关键是 1108 行，由于这段代码是在最顶层的 uvm_reg_block 中执行的，因此 get_name 返回的就是最顶层的 uvm_reg_block 的 name。如果 m_roots 中有 blk 的 get_name 的返回值等于此 name，那么有两种情况，一种是此 uvm_reg_block 就是这个最顶层的 uvm_reg_block，另外一种情况是有其它的 uvm_reg_block 和最顶层的 uvm_reg_block 重名了。后面一种情况是需要避免的。

18.3.2. Xinit_address_mapsX 函数

uvm_reg_block 的 Xinit_address_mapsX 函数的定义如下：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：Xinit_address_mapsX

1898  function void uvm_reg_block::Xinit_address_mapsX();
1899      foreach (maps[map_]) begin
1900          uvm_reg_map map = map_;
1901          map.Xinit_address_mapX();
1902      end
1903          //map.Xverify_map_configX();
1904  endfunction
```

这里会依次调用每个加入到此 uvm_reg_block 的每一个 uvm_reg_map 的 Xinit_address_mapsX 函数。要注意的是，uvm_reg_block 的 Xinit_address_mapsX 是被最顶层的 uvm_reg_block 调用的，因此这里的 uvm_reg_map 也是最顶层的 uvm_reg_map。并且一般说来，每个 uvm_reg_block 只有一个 uvm_reg_map，所以这里也仅仅只会调用最顶层的 uvm_reg_block 的 default_map 的 Xinit_address_mapsX 函数。函数的定义如下：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xinit_address_mapsX
```

```
1494 function void uvm_reg_map::Xinit_address_mapX();
1495
1496     int unsigned bus_width;
1497
1498     uvm_reg_map top_map = get_root_map();
1499
1500     if (this == top_map) begin
1501       top_map.m_regs_by_offset.delete();
1502       top_map.m_regs_by_offset_wo.delete();
1503       top_map.m_mems_by_offset.delete();
1504     end
1505
1506     foreach (m_submaps[l]) begin
1507       uvm_reg_map map=l;
1508       map.Xinit_address_mapX();
1509     end
1510
```

1498 行调用 get_root_map 函数取得最顶层的 uvm_reg_map。get_root_map 的定义如下：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_root_map

1038 function uvm_reg_map uvm_reg_map::get_root_map();
1039     return (m_parent_map == null) ? this : m_parent_map.get_root_map();
1040 endfunction: get_root_map
```

函数比较简单。这里判断是否是最顶层的 uvm_reg_map 的标准就是此 map 的 m_parent_map 是否为 null。很显然，非最顶层的 uvm_reg_map 是都有父 uvm_reg_map 的。

1500 行判断是不是最顶层的 uvm_reg_map 在调用此函数，如果是的话，那么就把 m_regs_by_offset，m_regs_by_offset_wo 和 m_mems_by_offset 清空。这几个联合数组的定义如下：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map

  78     local uvm_reg              m_regs_by_offset[uvm_reg_addr_t];
  79                                 // Use only in addition to above if a RO and a WO
  80                                 // register share the same address.
  81     local uvm_reg              m_regs_by_offset_wo[uvm_reg_addr_t];
  82     local uvm_mem               m_mems_by_offset[uvm_reg_map_addr_range];
```

其中 m_regs_by_offset 的索引是 uvm_reg_addr_t 类型的，即是一个地址值，而其内容则是 uvm_reg 类型的。m_regs_by_offset_wo 与 m_regs_by_offset 相似，它主要是用于在同一张 uvm_reg_map 中，在同一个地址上，有两个寄存器，其中一个

是”RO”的，另外一个是”WO”的情况。一般情况下较少会用到。m_mems_by_offset 的索引是 uvm_reg_map_addr_range 类型的，而内容则是 uvm_mem。

1506 行到 1509 行递归的执行所有 sub_map 的 Xinit_address_mapsX 函数。

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xinit_address_mapsX

1511      foreach (m_regs_info[rg_]) begin
1512        uvm_reg rg = rg_;
1513        m_regs_info[rg].is_initialized=1;
1514        if (!m_regs_info[rg].unmapped) begin
1515            string rg_acc = rg.Xget_fields_accessX(this);
1516          uvm_reg_addr_t addrs[];
1517
1518          bus_width = get_physical_addresses(m_regs_info[rg].offset,0,rg.get_n_bytes(),addrs);
1519
1520          foreach (addrs[i]) begin
1521            uvm_reg_addr_t addr = addrs[i];
1522
1523            if (top_map.m_regs_by_offset.exists(addr)) begin
1524
1525                uvm_reg rg2 = top_map.m_regs_by_offset[addr];
1526                string rg2_acc = rg2.Xget_fields_accessX(this);
1527
1528                // If the register at the same address is RO or WO
1529                // and this register is WO or RO, this is OK
1530                if (rg_acc == "RO" && rg2_acc == "WO") begin
1531                    top_map.m_regs_by_offset[addr]      = rg;
1532                    uvm_reg_read_only_cbs::add(rg);
1533                    top_map.m_regs_by_offset_wo[addr] = rg2;
1534                    uvm_reg_write_only_cbs::add(rg2);
1535                end
1536                else if (rg_acc == "WO" && rg2_acc == "RO") begin
1537                    top_map.m_regs_by_offset_wo[addr] = rg;
1538                    uvm_reg_write_only_cbs::add(rg);
1539                    uvm_reg_read_only_cbs::add(rg2);
1540                end
1541                else begin
1542                    string a;
1543                    a = $sformatf("%0h",addr);
1544                    `uvm_warning("RegModel", {"In map '",get_full_name(),"' register '",
1545                                                rg.get_full_name(), "' maps to same address
 as register '",
1546                                                top_map.m_regs_by_offset[addr].get_full_nam
e(),"': 'h",a})
1547                end
1548            end
1549            else
1550                top_map.m_regs_by_offset[addr] = rg;
1551
```

```
1552              foreach (top_map.m_mems_by_offset[range]) begin
1553                if (addr >= range.min && addr <= range.max) begin
1554                  string a,b;
1555                  a = $sformatf("%0h",addr);
1556                  b = $sformatf("[%0h:%0h]",range.min,range.max);
1557                  `uvm_warning("RegModel", {"In map '",get_full_name(),"' register '",
1558                       rg.get_full_name(), "' with address ",a,
1559                       "maps to same address as memory '",
1560                       top_map.m_mems_by_offset[range].get_full_name(),"': ",b})
1561                end
1562              end
1563            end
1564            m_regs_info[rg].addr = addrs;
1565          end
1566      end
1567
```

1511 到 1566 行遍历加入到此 uvm_reg_map 的所有的 uvm_reg。1513 行把 m_regs_info 中每一个 uvm_reg 的 is_initialized 赋值为 1。1514 行判断 unmapped 标志位，在初始的时候，这个 unmapped 是为 0 的，所以 1514 行的条件满足。

1515 行调用 uvm_reg 的 Xget_fields_accessX 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：Xget_fields_accessX

1802 function string uvm_reg::Xget_fields_accessX(uvm_reg_map map);
1803      bit is_R = 0;
1804      bit is_W = 0;
1805
1806      foreach(m_fields[i]) begin
1807          case (m_fields[i].get_access(map))
1808            "RO",
1809              "RC",
1810              "RS":
1811                  is_R = 1;
1812
1813            "WO",
1814                "WOC",
1815                "WOS",
1816                "WO1":
1817                    is_W = 1;
1818
1819            default:
1820                return "RW";
1821          endcase
1822
1823          if (is_R && is_W) return "RW";
1824      end
1825
1826      case ({is_R, is_W})
```

```
1827        2'b01: return "WO";
1828        2'b10: return "RO";
1829      endcase
1830      return "RW";
1831  endfunction
```

函数比较简单，就是遍历此 uvm_reg 中的所有的 uvm_reg_field，得到其存取策略，要么返回 RW，那么返回 RO，要么返回 WO。假如有一个 field 是 RO 的，有一个 field 是 WO 的，那么最终会返回 RW。

1518 行调用 get_physical_addresses 函数。这个函数相对复杂，将会在下节中介绍。这里仅仅说明一下其作用。函数的第一个参数表示 uvm_reg 或者 uvm_mem 的相对地址，第二个参数仅用于 memory 操作中，表示要得到地址的单元在整块 memory 中的偏移。第三个参数则表示此 uvm_reg 的位宽，一般说来，这个位宽将不会超过总线的位宽，即假如系统是 16 位的，那么此寄存器的宽度最大为 16，当然了，这里可能出现 32 的情况，这种情况下，一个寄存器就占据了两个地址。同样的情况也适用于 memory 的每个单元的位宽，即每个单元的位宽最好不要超过系统总线的宽，当超过时，那么一个单元就会占据多个地址。第四个参数表示返回的实际的地址，这里使用了 ref 形式的参数，注意到这个地址可能有多个，因此这里使用一个动态数组来表示。

1520 行遍历返回的所有的 addrs 值。1523 行检查最顶层的 uvm_reg_map 的 m_regs_by_offset 数组中是否已经有了对应此地址的寄存器。如果有，说明有其它寄存器也使用这一地址。一般来说，这种情况是不允许的，如 1542 到 1546 行就给出警告信息。但是有两种情况也允许的：一种是原来的寄存器是 WO，而这个寄存器是 RO 的，如 1530 行的分支所示，或者反过来，如 1536 行的分支所示。由于这两种情况均比较少见，因此这里不详细介绍。

1550 行，当最顶层的的 uvm_reg_map 的 m_regs_by_offset 数组中没有此地址时，说明此地址还没有被其它寄存器占用，那么就在 m_regs_by_offset 中插入一条记录，记录的索引是地址值，而内容是此地址对应的寄存器。

1552 行到 1562 行检查系统中已经有的 memory 的地址是否和此地址冲突。

1564 行把 m_regs_info 中寄存器的 addr 字段赋值。因此，这里可见，在最顶层的 uvm_reg_map 中，其 m_regs_by_offset 中存放了所有的寄存器及其地址信息，而在每一个子 uvm_reg_map 的 m_regs_info 数组中，存放了所有加入到此 uvm_reg_map 的寄存器的信息。

1511 行到 1566 行主要用于初始化所有的 uvm_reg 的地址信息，而 1568 到 1615 行用于初始化所有的 uvm_mem 的地址信息。在进入后面的代码前，先回过头来看一下 1514 行的的条件，如果 unmapped 被置位了，那么 1511 到 1566 行将不会初始化此寄存器的地址信息。这种情况下，就需要用户自己定义一个 FRONTDOOR 函数进行此寄存器的 FRONTDOOR 操作。关于这一点，后面会讲到。

文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xinit_address_mapsX

```
1568      foreach (m_mems_info[mem_]) begin
1569        uvm_mem mem = mem_;
1570        if (!m_mems_info[mem].unmapped) begin
1571
1572          uvm_reg_addr_t addrs[],addrs_max[];
1573          uvm_reg_addr_t min, max, min2, max2;
1574          int unsigned stride;
1575
1576          bus_width = get_physical_addresses(m_mems_info[mem].offset,0,mem.get_n_bytes(),a
ddrs);
1577          min = (addrs[0] < addrs[addrs.size()-1]) ? addrs[0] : addrs[addrs.size()-1];
1578          min2 = addrs[0];
1579
1580          void'(get_physical_addresses(m_mems_info[mem].offset,(mem.get_size()-1),mem.get_n_
bytes(),addrs_max));
1581          max = (addrs_max[0] > addrs_max[addrs_max.size()-1]) ? addrs_max[0] : addrs_ma
x[addrs_max.size()-1];
1582          max2 = addrs_max[0];
1583          // address interval between consecutive mem offsets
1584          stride = (max2 - min2)/(mem.get_size()-1);
1585
1586          foreach (top_map.m_regs_by_offset[reg_addr]) begin
1587            if (reg_addr >= min && reg_addr <= max) begin
1588              string a;
1589              a = $sformatf("%0h",reg_addr);
1590              `uvm_warning("RegModel", {"In map '",get_full_name(),"' memory '",
1591                   mem.get_full_name(), "' maps to same address as register '",
1592                   top_map.m_regs_by_offset[reg_addr].get_full_name(),"': 'h",a})
1593            end
1594          end
1595
1596          foreach (top_map.m_mems_by_offset[range]) begin
1597            if (min <= range.max && max >= range.max ||
1598                min <= range.min && max >= range.min ||
1599                min >= range.min && max <= range.max) begin
1600              string a;
1601              a = $sformatf("[%0h:%0h]",min,max);
1602              `uvm_warning("RegModel", {"In map '",get_full_name(),"' memory '",
1603                   mem.get_full_name(), "' overlaps with address range of memory '",
1604                   top_map.m_mems_by_offset[range].get_full_name(),"': 'h",a})
1605              end
1606          end
1607
1608          begin
1609            uvm_reg_map_addr_range range = '{ min, max, stride };
1610            top_map.m_mems_by_offset[ range ] = mem;
1611            m_mems_info[mem].addr   = addrs;
1612            m_mems_info[mem].mem_range = range;
```

```
1613            end
1614          end
1615        end
1616
1617        // If the block has no registers or memories,
1618        // bus_width won't be set
1619        if (bus_width == 0) bus_width = m_n_bytes;
1620
1621        m_system_n_bytes = bus_width;
1622    endfunction
```

1576 到 1584 行的语句有些难以理解。以一个例子介绍，假设这是一块 4*64 的 memory，即 memory 一共有 4 个单元，每个单元有 64bit，此 memory 的相对地址为 0，且此 memory 所在的 uvm_reg_map 就是最顶层的 uvm_reg_map，此 uvm_reg_map 的基地址为 0，此 uvm_reg_map 的位宽为 16bit，所以 1576 行得到的 addrs 中将会有 4 个地址，如果是小端的话，那么分别为 0，1，2，3，如果是大端的话，那么为 3，2，1，0。如果对此不理解，可以先看一下节关于 get_physical_addresses 的介绍。无论是大端还是小端，min 的值都将会是 0，而 min2 的值则不一定，在小端情况下为 0，在大端情况下为 3。

1580 行返回的 addrs_max 中也将会有四个地址值，如果是小端的话，那么为 12，13，14，15；如果是大端的话，那么会 15，14，13，12。无论是大端还是小端，max 的值都将会是 15，而 max2 的值在小端下为 12，大端下为 15。

因此，1584 行，在大端情况下，stride 的值将会是(15-3)/(4-1)=4，而在小端情况下为(12-0)/(4-1)=4。4 即是一个单元占据的地址的数量。

1586 行到 1594 行判断 register model 中已经分配了地址的 uvm_reg 的地址是否和此 uvm_mem 的地址冲突。这里的分配指的是已经加入到了最顶层的 uvm_reg_map 的 m_regs_by_offset 联合数组中。

1596 到 1606 行则用于检测 register model 中已经分配好了地址的 uvm_mem 的地址是否和此 uvm_mem 的地址冲突。这里的分配指的是已经加入到了最顶层的 uvm_reg_map 的 m_mems_by_offset 联合数组中。

1609 行实例化一个 uvm_reg_map_addr_range 类型的变量 range，并且把此 memory 的信息加入进去。这里加入了此 memory 的最大地址，最小地址，及每个单元占据几个地址。1610 行在最顶层的 uvm_reg_map 的 m_mems_by_offset 中插入一条记录，记录的索引是刚刚实例化的 range，而内容则是与此 range 对应的 uvm_mem。

1611 行把子 uvm_reg_map 中的 m_mems_info 数组，与此 uvm_mem 对应的记录的 addr 值设置为此 uvm_mem 的第一个单元的地址。注意的是，这里不是只存取了 uvm_mem 的第一个地址，而是存放了第一个单元的地址，在本例中有 4 个地址。1612 行更新 m_mems_info 数组中与此 uvm_mem 对应的记录的 mem_range 信息。

1619 行则是在极端情况下，在此 register model 中既没有 uvm_mem 也没有

uvm_reg，把 bus_width 的值设置为 m_n_bytes 值。

1621 行把 m_n_system_bytes 值设置为所有的 uvm_reg_map 中最小的位宽值。

18.3.3. uvm_reg_map 的 get_physical_addresses 函数

上节中提到了 get_physical_addresses 函数，本节详细的讲述。函数的定义为：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_physical_addresses

1318 function int uvm_reg_map::get_physical_addresses(uvm_reg_addr_t      base_addr,
1319                                                  uvm_reg_addr_t      mem_offset,
1320                                                  int unsigned        n_bytes,
1321                                                  ref uvm_reg_addr_t  addr[]);
1322     int bus_width = get_n_bytes(UVM_NO_HIER);
1323     uvm_reg_map  up_map;
1324     uvm_reg_addr_t  local_addr[];
1325     int multiplier = m_byte_addressing ? bus_width : 1;
1326
1327     addr = new [0];
1328
1329     if (n_bytes <= 0) begin
1330         `uvm_fatal("RegModel", $sformatf("Cannot access %0d bytes. Must be greater than 0
",
1331                                          n_bytes));
1332         return 0;
1333     end
1334
1335     // First, identify the addresses within the block/system
1336     if (n_bytes <= bus_width) begin
1337         local_addr = new [1];
1338         local_addr[0] = base_addr + (mem_offset * multiplier);
1339     end else begin
1340         int n;
1341
1342         n = ((n_bytes-1) / bus_width) + 1;
1343         local_addr = new [n];
1344
1345         base_addr = base_addr + mem_offset * (n * multiplier);
1346
1347         case (get_endian(UVM_NO_HIER))
1348             UVM_LITTLE_ENDIAN: begin
1349                 foreach (local_addr[i]) begin
1350                     local_addr[i] = base_addr + (i * multiplier);
1351                 end
```

```
1352              end
1353              UVM_BIG_ENDIAN: begin
1354                  foreach (local_addr[i]) begin
1355                      n--;
1356                      local_addr[i] = base_addr + (n * multiplier);
1357                  end
1358              end
1359              UVM_LITTLE_FIFO: begin
1360                  foreach (local_addr[i]) begin
1361                      local_addr[i] = base_addr;
1362                  end
1363              end
1364              UVM_BIG_FIFO: begin
1365                  foreach (local_addr[i]) begin
1366                      local_addr[i] = base_addr;
1367                  end
1368              end
1369              default: begin
1370                  `uvm_error("RegModel",
1371                      {"Map has no specified endianness. ",
1372                       $sformatf("Cannot access %0d bytes register via its %0d byte \"%s\" int
erface",
1373                      n_bytes, bus_width, get_full_name())})
1374              end
1375          endcase
1376      end
1377
```

函数有四个参数，第一个参数表示要获取地址的 uvm_reg 或者 uvm_mem 的相对地址，第二个参数仅仅用于 memory 操作中，表示要得到地址的单元在整块 memory 中的偏移。第三个参数则表示此 uvm_reg 或者 uvm_mem 的位宽。第四个参数表示返回的实际的地址，注意到这个地址可能有多个，因此这里使用一个动态数组来表示。

1322 行得到此 uvm_reg_map 的位宽，这里是以 byte 为单位的。这个位宽是在每个 uvm_reg_map 实例化的时候指定的。

1325 行根据 m_byte_adddressing 的值来判定乘数因子。一般情况下，m_byte_addressing 的值为 0，因此 multiplier 的值为 1。

1327 行清空 addr。因为 addr 是以 ref 形式传递进来的，并且最后还要把数据传递出去。因此假如原来数组中有东西的话，是会影响最后的结果的。

1329 行保证 n_bytes 是一个大于 0 的值。

1336 行判断要获取地址的 uvm_reg 或者 uvm_mem 的位宽是否小于总线宽度。如果小于，那么说明此 uvm_reg 或者 uvm_mem 的单元只占据一个地址，因此 1337 行把 local_addr 的大小设置为 1。对于一个 uvm_reg 来说，local_addr 的值就变为了 base_addr，这个也即是此 uvm_reg 的相对地址；对于一个 uvm_mem 的某一单元来

说，假如输入的 mem_offset 的参数为 3，即要获得这块 memory 第 4 个（第 1 个的偏移为 0，第 4 个的偏移为 3）单元的地址，那么 local_addr 的值就变成了此 uvm_mem 的第一个单元的地址的值加 3。

假如 1336 行的条件不满足，那么 1342 行将根据 n_bytes 的值来确定占据几个地址。如果 n_bytes 为 64，而 bus_width 为 16，那么 1342 行的 n 的值将会为 4，即占据 4 个地址。1345 行，对于 uvm_reg 来说，base_addr 的值没有变过；对于 uvm_mem 来说，base_addr 变为了 uvm_mem 的相对地址+3*4，因为要获取的是第 4 个单元的地址，在它前面有 3 个单元，这 3 个单元要分别占据 4 个地址，所以总共要占据 12 个地址。第 4 个单元的第一个地址等于 uvm_mem 的相对地址加 12。

1347 到 1368 行根据此 uvm_reg_map 是大端还是小端来给对应的 local_addr 赋值。假如是小端，那么这 4 个地址是往上递增的；假如是大端，那么这个地址是递减的；假如是 LITTLE_FIFO 或者 BIG_FIFO，那么地址等于基地址，这两个不常用。只有大端和小端比较常用。

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_physical_addresses

1378    up_map = get_parent_map();
1379
1380     // Then translate these addresses in the parent's space
1381     if (up_map == null) begin
1382        // This is the top-most system/block!
1383        addr = new [local_addr.size()] (local_addr);
1384        foreach (addr[i]) begin
1385           addr[i] += m_base_addr;
1386        end
1387     end else begin
1388        uvm_reg_addr_t   sys_addr[];
1389        uvm_reg_addr_t   base_addr;
1390        int w, k;
1391
1392        // Scale the consecutive local address in the system's granularity
1393        if (bus_width < up_map.get_n_bytes(UVM_NO_HIER))
1394          k = 1;
1395        else
1396          k = ((bus_width-1) / up_map.get_n_bytes(UVM_NO_HIER)) + 1;
1397
1398        base_addr = up_map.get_submap_offset(this);
1399        foreach (local_addr[i]) begin
1400           int n = addr.size();
1401
1402           w = up_map.get_physical_addresses(base_addr + local_addr[i] * k,
1403                                             0,
1404                                             bus_width,
1405                                             sys_addr);
1406
```

```
1407                addr = new [n + sys_addr.size()] (addr);
1408                foreach (sys_addr[j]) begin
1409                    addr[n+j] = sys_addr[j];
1410                end
1411            end
1412            // The width of each access is the minimum of this block or the system's width
1413            if (w < bus_width)
1414                bus_width = w;
1415        end
1416
1417        return bus_width;
1418
1419    endfunction: get_physical_addresses
```

1378 行得到 parent_map 的指针。假如 parent map 为 null，说明这是最顶层的 uvm_reg_map，那么要获取的地址就等于 local_addr 的值加上此顶层 uvm_reg_map 的基地址。否则的话，1388 到 1410 行将会递归的调用 parent map 的 get_physical_addresses 函数，得到最终的物理地址。

1393 行到 1396 行判断子 uvm_reg_map 和父 uvm_reg_map 的位宽是否相同。这里先考虑两者一致时的情况，即执行 1396 行的分支，得到 k 的值为 1。1398 行得到此子 uvm_reg_map 在父 uvm_reg_map 中的相对地址，这样 1402 行在调用父 uvm_reg_map 时输入的第一个参数就是相对于父 uvm_reg_map 的相对地址。1399 行开始遍历 local_addr 中的每一个值，并且递归的调用父 uvm_reg_map 的 get_physical_addresses 函数，得到此地址最终对应的物理地址。注意这里的 1400 行，在遍历第一个 local_addr 中的数据时，addr 中还没有任何值，即其大小为 0。1407 行到 1410 行把返回的 sys_addr 中地址的值赋值给 addr。这里我们考虑子 uvm_reg_map 和父 uvm_reg_map 的位宽相同的情况，因此返回的 sys_addr 的大小为 1。于是当 local_addr 遍历完成后，最终的 addr 的大小为 4，这跟 local_addr 的大小是一样的。

当子 uvm_reg_map 和父 uvm_reg_map 的位宽是不相同，先考虑前者小于后者的情况，设前者为 16，而后者为 32，那么 1394 行给 k 赋值为 1，1405 行最终返回的 sys_addr 的大小为 1。所以最终 addr 的大小依然为 4。

当子 uvm_reg_map 的位宽为 16，而父 uvm_reg_map 的位宽为 8 时，此时 1396 行得到的 k 值为 2。1402 行在调用时传入的第一个参数要乘以 k，即 2。这是比较好理解的，恰如 1345 行当 memory 的位宽大于总线位宽时，那么此时可以认为一个 offset 值对应多个地址。在我们的例子中，一个 memory 单元占据了 4 个单元，所以 1345 行要乘以 4。而在这里，子 uvm_reg_map 的一个单元占据了 2 个地址，所以要乘以 2。1405 行最终返回的 sys_addr 的值将会是 2。这样最终 addr 的大小为 8。即一个 64bit 的单元，要在整个的 register model 中占据 8 个地址。register model 的位宽是由最小的 uvm_reg_map 的位宽决定的。

1413 行把 bus_width 的值设置为最小的位宽值，1417 行把此值返回。

18.4. uvm_reg 的 write 操作：FRONTDOOR

接下来几节开始介绍 register model 的常用操作。本节介绍 FRONTDOOR 形式的 uvm_reg 的 write 操作。在介绍之前，将会首先介绍 register model 中为了保证操作的原子性而采取的方法。

18.4.1. reset 操作及 uvm_reg 的原子操作

当一个 register model 被集成到验证平台后，必须进行 reset 操作，这样才能保证每个寄存器的值等于我们设置的初始值。也就是说，register model 并不会自动的进行 reset 操作，必须显式的调用 reset 函数进行复位操作。uvm_reg_block 的 reset 函数如下：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：reset

1574 function void uvm_reg_block::reset(string kind = "HARD");
1575
1576     foreach (regs[rg_]) begin
1577        uvm_reg rg = rg_;
1578        rg.reset(kind);
1579     end
1580
1581     foreach (blks[blk_]) begin
1582        uvm_reg_block blk = blk_;
1583        blk.reset(kind);
1584     end
1585 endfunction
```

1576 到 1579 行调用所有加入到此 uvm_reg_block 的 reset 函数。uvm_reg 的 reset 函数为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：reset

1987 function void uvm_reg::reset(string kind = "HARD");
1988     foreach (m_fields[i])
1989         m_fields[i].reset(kind);
1990     // Put back a key in the semaphore if it is checked out
1991     // in case a thread was killed during an operation
```

```
1992      void'(m_atomic.try_get(1));
1993      m_atomic.put(1);
1994  endfunction: reset
```

1988 到 1989 行调用所有此 uvm_reg 的 uvm_reg_field 的 reset 函数，其定义为：

1272 行和 1273 行检查是否定义了名字为”HARD”的复位，如果没有定义，则直接返回。

1276 到 1278 行把 uvm_reg_field 内部三个用于存储数据的变量全部赋值为 m_reset 中的相应值。m_reset 中的记录是在 uvm_reg_field 的 configure 通过调用 set_reset 函数插入的。1281 行把 m_written 赋值为 0。

回到 uvm_reg 的 reset 函数，1292 行出现了 m_atomic 变量：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg

  48      local semaphore             m_atomic;
```

m_atomic 本质上是一个 semaphore，用于实现进程的同步。1292 行到 1293 行主要是为了保证 m_atomic 中有一个键值。在 uvm_reg 的 new 函数中，1159 行有如下语句：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：new

1159      m_atomic          = new(1);
```

因此，在初始状态下，m_atomic 中已经有了一个键值。在 new 与 reset 之间某个函数把此键值通过 get 或者 try_get 方法取回，但是之后发生了意外，没有能够再放回去，从而 m_atomic 中变成空的。在这种情况下，try_get 语句将会返回 0，即不能从 m_atomic 中得到键值，于是 1993 行把一个新的键值放入。在没有意外发生的情况下，1992 行的 try_get 将会从 m_atomic 中取回一个键值，这样 m_atomic 中将会是空的，于是 1193 行的语句把一个新的键值放入。总之，无论原来 m_atomic 中是什么情况，这两句话执行之后，m_atomic 中将会有一个键值。为什么要保证 m_atomic 中有一个键值？我们来看 XatomicX 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：XatomicX

2916  task uvm_reg::XatomicX(bit on);
2917      process m_reg_process;
2918      m_reg_process=process::self();
2919
2920      if (on) begin
```

```
2921          if (m_reg_process == m_process)
2922            return;
2923          m_atomic.get(1);
2924          m_process = m_reg_process;
2925        end
2926        else begin
2927          // Maybe a key was put back in by a spurious call to reset()
2928          void'(m_atomic.try_get(1));
2929          m_atomic.put(1);
2930          m_process = null;
2931        end
2932  endtask: XatomicX
```

通常，我们要访问互斥的资源时，即只允许一个进程访问，其它进程必须在此进行访问完成之后才能访问，可以这样写：

```
XatomicX(1);
…//access the resource
XaotmicX(0);
```

后面我们将会看到，在 read，write，poke，peek，mirror 等操作中，都用到了类似的方法来保证在同一时间只有一个进程对互斥资源进行访问。

来分析 XatomicX 函数，当输入的参数为 1 时，那么将从 m_atomic 中得到一个键值，这里使用的是 get 函数。如果是有多个进程同时调用了 get 函数，那么将会有一个先得到，但是其它的进程将会阻塞在那里，一直等待 m_atomic 中有新的键值出现。得到键值的进程将会对资源进行访问，访问完成后再次调用 XatomicX 函数，传入的参数为 0，此时 2929 行将会向 m_atomic 中写入一个新的键值，于是其它等待的进程将会获得键值。

这里稍微让人疑惑的 2921 和 2922 行，牵扯到了进程的一些概念。只有通过 fork 函数才能新建立一个进程：

```
fork
    a;
    b;
join
```

像如上所示，是开启了两个进程，假如 a 语句中有 task1 调用 func1，而 func1 又调用 func2，那么 task1，func1 和 func2 共用的都是同一个进程，也即函数调用是不产生新的进程。假设上面的 a 语句和 b 语句同时对一个 uvm_reg 进行操作，操作之前和操作之后分别调用 XatomicX 函数，a 先得到了 m_atomic 中的键值，于是 2924 行把此 uvm_reg 的 m_process 赋值为 a 进程，即表示此刻 a 进程正在访问此 uvm_reg。如果 a 进程不小心连续两次调用了传入参数为 1 的 XatomicX 的函数：

```
XatomicX(1);
XatomicX(1);
…//access the resource
XaotmicX(0);
```

```
XaotmicX(0);
```

那么在第二次调用的时候，2921 行会检查两个进程是否为同一个进程，假设是同一个进程，那就表明此进程之前已经得到了此 uvm_reg 的操作权，现在又要求得到此 uvm_reg 的操作权，这显然是多余的，于是就直接返回。

小结一下，uvm_reg 的 reset 函数被调用后，uvm_reg 的各个 uvm_reg_field 值将会是预先设置的 reset 值，同时每个 uvm_reg 的 m_atomic 中将会有一个键值以保证接下来的其它函数对此 uvm_reg 进行原子操作。

回到 uvm_reg_block 的 reset 函数，在调用完所有 uvm_reg 的 reset 函数后，1581 到 1584 行递归调用所有子 uvm_reg_block 的 reset 函数，而子 uvm_reg_block 的 reset 函数又会调用所有加入其中的 uvm_reg 的 reset 函数。因此整个 reset 函数的执行效果就是所有 register model 中的 uvm_reg 的 reset 函数都会调用了。

18.4.2. uvm_reg::write

本节分析 uvm_reg 的 write 操作。由于操作涉及 FRONTDOOR 和 BACKDOOR 形式，本节先介绍 FRONTDOOR 形式，下节介绍 BACKDOOR 形式。uvm_reg 的 write 任务如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：write

2088 task uvm_reg::write(output uvm_status_e          status,
2089                     input  uvm_reg_data_t        value,
2090                     input  uvm_path_e            path = UVM_DEFAULT_PATH,
2091                     input  uvm_reg_map           map = null,
2092                     input  uvm_sequence_base parent = null,
2093                     input  int                   prior = -1,
2094                     input  uvm_object            extension = null,
2095                     input  string                fname = "",
2096                     input  int                   lineno = 0);
2097
2098   // create an abstract transaction for this operation
2099   uvm_reg_item rw;
2100
2101   XatomicX(1);
2102
2103   rw = uvm_reg_item::type_id::create("write_item",,get_full_name());
2104   rw.element          = this;
2105   rw.element_kind = UVM_REG;
2106   rw.kind             = UVM_WRITE;
2107   rw.value[0]         = value;
```

```
2108      rw.path              = path;
2109      rw.map                 = map;
2110      rw.parent            = parent;
2111      rw.prior             = prior;
2112      rw.extension       = extension;
2113      rw.fname             = fname;
2114      rw.lineno            = lineno;
2115
2116      do_write(rw);
2117
2118      status = rw.status;
2119
2120      XatomicX(0);
2121
2122  endtask
```

2101 到 2120 行保证这是一个原子操作，上节已经讲述，不在重复说明。

2103 行实例化一个 uvm_reg_item 类型的变量 rw，2104 行到 2114 行给 rw 赋值。2104 行把 element 赋值为 this，表示是由此 uvm_reg 发起的 write 操作，2105 表示发起 write 操作的是一个寄存器，而不是一个 uvm_reg_field，也不是一个 uvm_mem。2107 行把要写入的值放在 value 中。2108 行给 path 赋值，这里 path 可能是 UVM_BACKDOOR，也可以是 UVM_FRONTDOOR，如果不指定的话，那么将会是 UVM_DEFAULT_PATH。2109 行给 map 参数赋值。一般说来，map 参数都不会特意指定，因此这里一般为 null。2110 行给 parent 赋值，所谓的 parent 就是假如此次操作是 FRONTDOOR 形式，那么必然要启动一个 sequence，parent 就是此 sequence。这里还没有启动 sequence，所以自然为 null。2111 到 2114 行则给一些其它的变量赋值，相对来说不那么重要，直接跳过。

2116 行调用 do_write 任务。这是整个 write 任务的关键。将会在下节中介绍这个任务。

18.4.3. uvm_reg::do_write(一)

do_write 任务的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

2127  task uvm_reg::do_write (uvm_reg_item rw);
2128
2129      uvm_reg_cb_iter  cbs = new(this);
2130      uvm_reg_map_info map_info;
```

```
2131      uvm_reg_data_t     value;
2132
2133      m_fname    = rw.fname;
2134      m_lineno  = rw.lineno;
2135
2136      if (!Xcheck_accessX(rw,map_info,"write()"))
2137         return;
2138
```

2133 到 2134 行给 m_fname 和 m_lineno 赋值。这两个赋值对整个 write 操作没有什么影响。2136 行调用 Xcheck_accessX 函数。

18.4.4. uvm_reg:: Xcheck_accessX

Xcheck_accessX 函数的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：Xcheck_accessX

2543 function bit uvm_reg::Xcheck_accessX (input uvm_reg_item rw,
2544                                         output uvm_reg_map_info map_info,
2545                                         input string caller);
2546
2547
2548      if (rw.path == UVM_DEFAULT_PATH)
2549         rw.path = m_parent.get_default_path();
2550
2551      if (rw.path == UVM_BACKDOOR) begin
2552          if (get_backdoor() == null && !has_hdl_path()) begin
2553              `uvm_warning("RegModel",
2554                  {"No backdoor access available for register '",get_full_name(),
2555                  "' . Using frontdoor instead."})
2556              rw.path = UVM_FRONTDOOR;
2557          end
2558          else
2559             rw.map = uvm_reg_map::backdoor();
2560      end
2561
2562
2563      if (rw.path != UVM_BACKDOOR) begin
2564
2565         rw.local_map = get_local_map(rw.map,caller);
2566
2567         if (rw.local_map == null) begin
2568             `uvm_error(get_type_name(),
2569                 {"No transactor available to physically access register on map '",
```

```
2570                  rw.map.get_full_name(),"""})
2571              rw.status = UVM_NOT_OK;
2572              return 0;
2573          end
2574
2575          map_info = rw.local_map.get_reg_map_info(this);
2576
2577          if (map_info.frontdoor == null && map_info.unmapped) begin
2578                `uvm_error("RegModel", {"Register '",get_full_name(),
2579                   "' unmapped in map '",
2580                   (rw.map==null)? rw.local_map.get_full_name():rw.map.get_full_name(),
2581                   "' and does not have a user-defined frontdoor"})
2582              rw.status = UVM_NOT_OK;
2583              return 0;
2584          end
2585
2586          if (rw.map == null)
2587            rw.map = rw.local_map;
2588       end
2589       return 1;
2590   endfunction
```

如果在调用 write 时没有指定操作方式，那么 2549 行给 rw 的 path 赋值。这里用到了 uvm_reg_block 的 get_default_path 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：get_default_path

1883  function uvm_path_e uvm_reg_block::get_default_path();
1884
1885      if (this.default_path != UVM_DEFAULT_PATH)
1886          return this.default_path;
1887
1888      if (this.parent != null)
1889          return this.parent.get_default_path();
1890
1891      return UVM_FRONTDOOR;
1892
1893  endfunction
```

1885 行会检查 uvm_reg_block 的 default_path 的值：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

  51      uvm_path_e          default_path = UVM_DEFAULT_PATH;
```

在默认情况下，它的值也为 UVM_DEFAULT_PATH。1888 行和 1889 行递归的调用父 uvm_reg_block 的 get_default_path 函数。如果发现所有的都没有设置过，那么最终会返回 UVM_FRONTDOOR，否则的话将会返回设置值。因此，在默认情况

下，调用 write 任务时，不指定操作方式，那么将会使用 FRONTDOOR 方式。如果在最顶层的 uvm_reg_block 中，把其 default_path 设置为 UVM_BACKDOOR，调用 write 任务时，不指定操作方式，那么将会使用最顶层设置的 BACKDOOR 方式。

回到 Xcheck_accessX 函数，2551 行到 2560 行检查是否采用 BACKDOOR 形式，本节先分析 FRONTDOOR 形式，因此这几行不会执行。

2563 行分析使用非 FRONTDOOR 的情况。2565 行调用 get_local_map 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_local_map

1596 function uvm_reg_map uvm_reg::get_local_map(uvm_reg_map map, string caller="");
1597     if (map == null)
1598       return get_default_map();
1599     if (m_maps.exists(map))
1600       return map;
1601     foreach (m_maps[l]) begin
1602       uvm_reg_map local_map=l;
1603       uvm_reg_map parent_map = local_map.get_parent_map();
1604
1605       while (parent_map != null) begin
1606         if (parent_map == map)
1607           return local_map;
1608         parent_map = parent_map.get_parent_map();
1609       end
1610     end
1611     `uvm_warning("RegModel",
1612         {"Register '",get_full_name(),"' is not contained within map '",map.get_full_name(),"'
",
1613          (caller == "" ? "": {" (called from ",caller,")"}) })
1614     return null;
1615 endfunction
```

注意这里传入的第一个参数是 rw 的 map 变量。这个 map 变量一般为 null。1597 行判断 map 是否为 null，在为 null 的情况下，那么就调用 get_default_map 函数，我们先跳过这个函数，直接看不为 null 的情况，1599 行检查输入的 map 是否在此 uvm_reg 的 m_maps 数组中，如果是的话，说明此 uvm_reg 已经加入到此 uvm_reg_map 中了，就直接返回这个 uvm_reg_map。如果 在 m_maps 中没有找到，那么 1601 到 1610 行则递归的查找父 uvm_reg_map，看看父 uvm_reg_map 是否等于输入的 map。假如要访问的 uvm_reg 的层次关系为 blk0.blk1.blk2.reg，三块 uvm_reg_block 对应的 map 分别为 map0，map1，map2。在此 reg 的 write 中输入的 map 参数是 map0 或者 map1，那么最终将会返回 map2。假设输入的参数是 map3，那么最终返回的是 null。因为 write 任务的 map 函数可以这样理解，它指定了在此 map 上进行 write 操作。如上面的例子，可以指定在 map0，也可以指定在 map1，还可以指定在 map2 上，因为在这几张 map 上，此寄存器都是可见的。但是不能在 map3 上，因为在 map3 中不知道此

寄存器。

回过来看 1598 行的 get_default_map 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_default_map

1621 function uvm_reg_map uvm_reg::get_default_map(string caller="");
1622
1623     // if reg is not associated with any map, return null
1624     if (m_maps.num() == 0) begin
1625         `uvm_warning("RegModel",
1626            {"Register '",get_full_name(),"' is not registered with any map",
1627             (caller == "" ? "": {" (called from ",caller,")"})})
1628         return null;
1629     end
1630
1631     // if only one map, choose that
1632     if (m_maps.num() == 1) begin
1633       uvm_reg_map map;
1634       void'(m_maps.first(map));
1635       return map;
1636     end
1637
1638     // try to choose one based on default_map in parent blocks.
1639     foreach (m_maps[l]) begin
1640       uvm_reg_map map = l;
1641       uvm_reg_block blk = map.get_parent();
1642       uvm_reg_map default_map = blk.get_default_map();
1643       if (default_map != null) begin
1644         uvm_reg_map local_map = get_local_map(default_map,"get_default_map()");
1645         if (local_map != null)
1646           return local_map;
1647       end
1648     end
1649
1650     // if that fails, choose the first in this reg's maps
1651
1652     begin
1653       uvm_reg_map map;
1654       void'(m_maps.first(map));
1655       return map;
1656     end
1657
1658 endfunction
```

1624 行到 1629 行判断是否为此 uvm_reg 设置了 uvm_reg_map，如果没有设置给出警告信息，并且返回 null。

1632 行到 1636 行检查如果只设置了一个 uvm_reg_map，那么直接把这张 map

返回。

1639 到 1648 行用于看一下 m_maps 中的 map，是否为此 map 所在 uvm_reg_block 的 default_map，如果是的话，那么就把这张 map 返回，也就是说从多个 map 中选择一个，选择的标准就是此 map 是其 parent block 的 default map。以一个例子来解释，假如此 uvm_reg 的 m_maps 中有两个 map：map1 和 map2，这两张 map 都从属于 block0 中，即 block0 中有两张 uvm_reg_map，其中的 map1 是 default map，那么这里将会把 map1 作为 get_default_map 的返回值。

1652 行到 1656 行在前面的语句都不满足的条件下，那么直接返回 m_maps 中的第一个做为此 uvm_reg 的 default map。1639 到 1648 行的语句什么以时候会不满足？假如此 uvm_reg 的 m_maps 中有两个 map：map1 和 map2，这两张 map 都从属于 block0 中，即 block0 中有两张 uvm_reg_map，除此之外还有第三张 map，这张 map 是 default map，但是这张 map 并不在此 uvm_reg 的 m_maps 数组中，在这种情况下，这里 1646 行的条件将不会满足，不会真把返回。

总结一下 get_default_map 函数，如果此 uvm_reg 的 m_maps 没有记录，那么返回 null；如果有一条记录，那么直接返回其中的 map；如果有多条记录，那么查看其对应的 map 是否为父 uvm_reg_block 的 default_map，如果是直接返回；如果有多条记录，且其中所有的记录对应的 map 都不是父 uvm_reg_block 的 default_map，那么直接返回 m_maps 中第一条记录对应的 uvm_reg_map。

返回到 get_local_map 函数，总结一下，假设输入的参数为 null 时，那么其返回值就等于上面 get_default_map 的返回值；如果不为 null，那么则查看输入的 map 是否在其这个 uvm_reg 的“map 树”上。要注意的是，这里的 map 树可能是有多个的，因为 m_maps 数组中可能有多条记录，每条记录可能对应一个 map 树。如果在某棵 map 树中找到了输入的 map，那么就返回 m_maps 中此棵 map 树对应的结点。

回到 Xcheck_accessX 函数，2567 到 2573 行在没有找到 local map 的情况下，把 rw 的 status 置为 UVM_NOT_OK，直接返回 0。

2575 行调用了 uvm_reg_map 的 get_reg_map_info 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_reg_map_info

1218 function uvm_reg_map_info uvm_reg_map::get_reg_map_info(uvm_reg rg, bit error=1);
1219    uvm_reg_map_info result;
1220    if (!m_regs_info.exists(rg)) begin
1221       if (error)
1222          `uvm_error("REG_NO_MAP",{"Register '",rg.get_name(),"' not in map '",get_name(),"'
"})
1223       return null;
1224    end
1225    result = m_regs_info[rg];
1226    if(!result.is_initialized)
```

```
1227       `uvm_warning("RegModel",{"map '",get_name(),"' does not seem to be initialized correc
tly, check that the top register model is locked()"})
1228
1229     return result;
1230  endfunction
```

函数整体来说比较简单，从 m_regs_info 中查找此 uvm_reg 对应的记录，在查找的过程中给出错误提示。其中的 1226 行会检查 is_initialized 标志位，这个标志位是在 Xinit_address_mapX 中被置位的，只能经过此函数，那么 m_regs_info 中所有寄存器的 addrs 信息才是这个寄存器的全局地址信息。1229 行直接返回查找到的寄存器的 uvm_reg_map_info 信息，这个信息中包含了此寄存器的地址等信息。

2577 行判断返回的此寄存器的 uvm_reg_maop_info 信息中 frontdoor 是否为 null，且 unmapped 是否为 1。在介绍 Xinit_addresss_mapX 函数时说过，如果 unmapped 被设置为 1，那么此寄存器的地址信息就没有被 Xinit_addresss_mapX 初始化。没有初始化过，那么就不能使用系统定义的 FRONTDOOR 操作，在这种情况下如果要进行 FRONTDOOR 操作，那么需要自定义 frontdoor，如果没有自定义，那么 2578 到 2581 行给出出错提示，2582 行给 status 赋值，2583 行直接返回。

2587 行在输入的 map 为 null 情况下，把 rw 的 map 赋值为 rw 的 local_map。

2589 行直接返回 1。

小结一下 Xcheck_accessX 函数，当使用 FRONTDOOR 方式时，这个函数主要是检查要读写的寄存器的 uvm_reg_map 是否准备好了，其地址如果没有被初始化过，那么看一下是否定义了自己的 frontdoor。

18.4.5. uvm_reg::do_write(二)

继续看 uvm_reg 的 do_write 任务：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

2139      XatomicX(1);
2140
2141      m_write_in_progress = 1'b1;
2142
2143      rw.value[0] &= ((1 << m_n_bits)-1);
2144      value = rw.value[0];
2145
2146      rw.status = UVM_IS_OK;
2147
```

```
2148     // PRE-WRITE CBS - FIELDS
2149     begin : pre_write_callbacks
2150         uvm_reg_data_t    msk;
2151         int lsb;
2152
2153         foreach (m_fields[i]) begin
2154             uvm_reg_field_cb_iter cbs = new(m_fields[i]);
2155             uvm_reg_field f = m_fields[i];
2156             lsb = f.get_lsb_pos();
2157             msk = ((1<<f.get_n_bits())-1) << lsb;
2158             rw.value[0] = (value & msk) >> lsb;
2159             f.pre_write(rw);
2160             for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next()) begin
2161                 rw.element = f;
2162                 rw.element_kind = UVM_FIELD;
2163                 cb.pre_write(rw);
2164             end
2165
2166             value = (value & ~msk) | (rw.value[0] << lsb);
2167         end
2168     end
2169     rw.element = this;
2170     rw.element_kind = UVM_REG;
2171     rw.value[0] = value;
2172
2173     // PRE-WRITE CBS - REG
2174     pre_write(rw);
2175     for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2176         cb.pre_write(rw);
2177
2178     if (rw.status != UVM_IS_OK) begin
2179       m_write_in_progress = 1'b0;
2180
2181       XatomicX(0);
2182
2183       return;
2184     end
2185
```

回到 do_write 任务。2136 行在 Xcheck_accessX 返回为 0，即不满足操作条件的情况下，直接返回，即此次 write 操作失败。

2139 行开始进行一次写操作。由于一次写操作不能中止，所以这里采用 XatomicX 来保证这种原子性，正常情况下这个原子操作在 2306 行结束。2141 行给 m_write_in_proceess 赋值，表示正在进行写操作。正常情况下，2304 行把 m_write_in_process 赋值为 0，表示写操作结束。

2143 行使用与操作来截取 rw.value[0]中的有效位。例如一个寄存器是 64 位的，那么这里只会取 rw.value[0]中的最后 64 位。这里没有考虑系统总线的情况。假设系统总线是 16 位的，那么这里 value 的值依然为 64 位的。

2146 行把 status 初始化为 UVM_IS_OK。2149 到 2168 行执行此 uvm_reg 的所有 uvm_reg_field 的 callback 函数 pre_write，这里不多介绍。

由于 2161 到 2162 行对 rw 中的某些变量进行了赋值，2169 到 2171 行重新对这些变量进行赋值。尤其需要注意的是 2171 行，各个 uvm_reg_field 的 pre_write 函数可能会改变要写入各个 field 的值，因此 2171 行要写入的值更新为被改变过的值。

2174 到 2176 行执行此 uvm_reg 的所有的 callback 函数 pre_write。从这里也看可以看出，在定义 pre_write 时，可以分别定义 uvm_reg 的 pre_write，也可以定义 uvm_reg_field 的 pre_write。当两者全部定义的时候，那么需要注意的是 uvm_reg_field 的 pre_write 先执行。A 用户定义了 uvm_reg_field 的 pre_write，把值乘以 2，而 B 用户定义了 uvm_reg 的 pre_write，把要写入的值除以 2，那么最终的结果是要写入的值没有变更过。因此牵扯到改变要写入的值时要小心。

2178 行到 2184 行，如果前面的 pre_write 操作把 status 置位为 UVM_NOT_OK，那么这里将会直接退出。退出时要释放之前的原子操作时所占用的 m_atomic 中的键值。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

2186      // EXECUTE WRITE...
2187      case (rw.path)
2188
2189          // ...VIA USER BACKDOOR
2190          UVM_BACKDOOR: begin
                  …
2226          end
2227
2228          UVM_FRONTDOOR: begin
2229
2230              uvm_reg_map system_map = rw.local_map.get_root_map();
2231
2232              m_is_busy = 1;
2233
2234              // ...VIA USER FRONTDOOR
2235              if (map_info.frontdoor != null) begin
2236                  uvm_reg_frontdoor fd = map_info.frontdoor;
2237                  fd.rw_info = rw;
2238                  if (fd.sequencer == null)
2239                     fd.sequencer = system_map.get_sequencer();
2240                  fd.start(fd.sequencer, rw.parent);
2241              end
2242
2243              // ...VIA BUILT-IN FRONTDOOR
2244              else begin : built_in_frontdoor
2245
2246                  rw.local_map.do_write(rw);
```

```
2247
2248            end
2249
2250            m_is_busy = 0;
2251
```

2187 行判断操作的方式，本节分析 FRONTDOOR，因此从 2228 行开始看起。2230 行得到最顶层的 uvm_reg_map，这里用到了 get_root_map 函数，前面已经分析过此函数，就是根据 parent_map 是否为 null 来判断这是否是最顶层的 uvm_reg_map。

2232 和 2250 行分别对 m_is_busy 进行置位，2232 表示接下来要启动 sequence 进行读写，开始忙碌状态，2250 行表示忙碌结束。

2235 行判断用户是否自定义了 frontdoor 操作方式，如果自定义了，那么使用自定义的 frontdoor 来启动 sequence 进行写操作，否则执行 2246 行，调用 local_map 的 do_write 来进行写操作，需要注意的是这里调用的是 local_map 的 do_write，而不是最顶层 uvm_reg_map 的 do_write。

18.4.6. uvm_reg_map::do_write

uvm_reg_map 的 do_write 任务为：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：do_write

1668 task uvm_reg_map::do_write(uvm_reg_item rw);
1669
1670    uvm_reg_map system_map = get_root_map();
1671    uvm_reg_adapter adapter = system_map.get_adapter();
1672    uvm_sequencer_base sequencer = system_map.get_sequencer();
1673
1674    if (rw.parent == null)
1675       rw.parent = new("default_parent_seq");
1676
1677    if (adapter == null) begin
1678       rw.set_sequencer(sequencer);
1679       rw.parent.start_item(rw,rw.prior);
1680       rw.parent.finish_item(rw);
1681       rw.end_event.wait_on();
1682    end
1683    else begin
1684       do_bus_write(rw, sequencer, adapter);
1685    end
1686
1687 endtask
```

1670 行得到最顶层的 uvm_reg_map，1671 和 1672 行调用与最顶层 uvm_reg_map 相对应的 adapter 和 sequencer。这里用到了 get_adapter 函数和 get_sequencer 函数，两者的定义分别：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_adapter

1090 function uvm_reg_adapter uvm_reg_map::get_adapter(uvm_hier_e hier=UVM_HIER);
1091    if (hier == UVM_NO_HIER || m_parent_map == null)
1092       return m_adapter;
1093    return m_parent_map.get_adapter(hier);
1094 endfunction
```

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_sequencer

1081 function uvm_sequencer_base uvm_reg_map::get_sequencer(uvm_hier_e hier=UVM_HIER);
1082    if (hier == UVM_NO_HIER || m_parent_map == null)
1083       return m_sequencer;
1084    return m_parent_map.get_sequencer(hier);
1085 endfunction
```

两个函数比较像，也比较简单，不多做介绍。

1675 行实例化一个 uvm_sequence。1677 行判断 adapter 是否为 null。如果为 null，那么说明这个 sequencer 可以直接接受 uvm_reg_item 类型的变量，于是 1678 到 1680 行把要写的 item 通过 sequencer 发送出去，1681 行调用 rw 的 end_event 的 wait_on，end_event 是一个在 uvm_transaction 中定义的变量：

```
文件：src/base/uvm_transaction.svh
类：uvm_transaction

446    uvm_event end_event;
```

在 uvm_transaction 的 new 函数中，它被初始化：

```
文件：src/base/uvm_transaction.svh
类：uvm_transaction
函数/任务：new

487 function uvm_transaction::new (string name="",
488                                  uvm_component initiator = null);
489
490    super.new(name);

494    end_event = events.get("end");
495
```

```
496 endfunction // uvm_transaction
```

events 是一个 uvm_transaction 中定义的 uvm_event_pool 类型的变量：

```
文件：src/base/uvm_transaction.svh
类：uvm_transaction

412    const uvm_event_pool events = new;
```

events 的 get 函数将会在 events 的联合数组中新建一条索引为”end”，类型为 uvm_event 的记录，并把这个 uvm_event 的实例返回。因此在 new 函数中，end_event 就已经具有了初值。

uvm_event 的 wait_on 的定义为：

```
文件：src/base/uvm_event.svh
类：uvm_event
函数/任务：wait_on

 69    virtual task wait_on (bit delta=0);
 70      if (on) begin
 71        if (delta)
 72          #0;
 73        return;
 74      end
 75      num_waiters++;
 76      @on;
 77    endtask
```

这里就是会等待 uvm_event 中的成员变量 on 变为 1。那么什么时候这个值会改变呢？在 uvm_event 的 trigger 函数中：

```
文件：src/base/uvm_event.svh
类：uvm_event
函数/任务：trigger

163    virtual function void trigger (uvm_object data=null);
164      int skip;
165      skip=0;
166      if (callbacks.size()) begin
167        for (int i=0;i<callbacks.size();i++) begin
168          uvm_event_callback tmp;
169          tmp=callbacks[i];
170          skip = skip + tmp.pre_trigger(this,data);
171        end
172      end
173      if (skip==0) begin
174        ->m_event;
175        if (callbacks.size()) begin
176          for (int i=0;i<callbacks.size();i++) begin
177            uvm_event_callback tmp;
```

```
178                 tmp=callbacks[i];
179                 tmp.post_trigger(this,data);
180               end
181           end
182           num_waiters = 0;
183           on = 1;
184           trigger_time = $realtime;
185           trigger_data = data;
186       end
187     endfunction
```

可见，trigger 会让 on 值变为 1，也就是说只要某个进程触发了 end_event 的 trigger 事件，那么就会释放这个 wait_on 进程 。在 uvm_sequence_base 的 finish_item 语句中：

```
文件：src/seq/uvm_sequence_base.svh
类：uvm_sequence_base
函数/任务：finish_item

 801     virtual task finish_item (uvm_sequence_item item,
 802                               int set_priority = -1);
 803
 804       uvm_sequencer_base sequencer;
 805
 806       sequencer = item.get_sequencer();
 807
 808       if (sequencer == null) begin
 809           uvm_report_fatal("STRITM", "sequence_item has null sequencer", UVM_NONE);
 810       end
 811
 812       mid_do(item);
 813       sequencer.send_request(this, item);
 814       sequencer.wait_for_item_done(this, -1);
 815       `ifndef UVM_DISABLE_AUTO_ITEM_RECORDING
 816       sequencer.end_tr(item);
 817       `endif
 818       post_do(item);
 819
 820     endtask
```

814 行会等待 wait_for_item_done。在 sequence 机制中已经说过，这句话最终会在 driver 中调用 item_done()之后结束，从而 816 行调用 sequencer 的 end_tr。end_tr 是在 uvm_component 中定义的一个函数，其代码为：

```
文件：src/base/uvm_component.svh
类：uvm_component
函数/任务：end_tr

2682 function void uvm_component::end_tr (uvm_transaction tr,
2683                                       time end_time=0,
2684                                       bit free_handle=1);
       …
```

```
2725    e = event_pool.get("end_tr");
2726    if(e!=null)
2727       e.trigger();
2728
2729 endfunction
```

2725 到 2727 行会最终触发 end_event 事件。

回到 uvm_reg_map 的 do_write，1681 行最终会在 driver 调用了 item_done 之后返回，即一次写操作在总线上被完成了。

如果系统总线的 sequencer 不能接受 uvm_reg_item 类型的变量，那么必须设置一个 adapter，此时 do_write 会执行 1684 行的分支，调用 do_bus_write。

18.4.7. uvm_reg::do_bus_write

任务的代码为：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：do_bus_write

1716 task uvm_reg_map::do_bus_write (uvm_reg_item rw,
1717                                  uvm_sequencer_base sequencer,
1718                                  uvm_reg_adapter adapter);
1719
1720    uvm_reg_addr_t        addrs[$];
1721    uvm_reg_map           system_map = get_root_map();
1722    int unsigned          bus_width  = get_n_bytes();
1723    uvm_reg_byte_en_t     byte_en    = -1;
1724    uvm_reg_map_info      map_info;
1725    int                   n_bits;
1726    int                   lsb;
1727    int                   skip;
1728    int unsigned          curr_byte;
1729    int                   n_access_extra, n_access;
1730
1731    Xget_bus_infoX(rw, map_info, n_bits, lsb, skip);
1732    `UVM_DA_TO_QUEUE(addrs,map_info.addr)
1733
1734    // if a memory, adjust addresses based on offset
1735    if (rw.element_kind == UVM_MEM)
1736       foreach (addrs[i])
1737          addrs[i] = addrs[i] + map_info.mem_range.stride * rw.offset;
1738
```

1721 行得到最顶层的 uvm_reg_map，1722 行得到总线位宽。注意是这里得到的

总不是整个系统的总线位宽，而只是这个 map 的位宽，整个系统的位宽只会小于或者等于此位宽。

1731 行调用 Xget_bus_infoX 函数来得到总线的信息，注意这里输入的几个参数除了第一个之外，全部都是未初始化的数值：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xget_bus_infoX

1629 function void uvm_reg_map::Xget_bus_infoX(uvm_reg_item rw,
1630                                            output uvm_reg_map_info map_info,
1631                                            output int size,
1632                                            output int lsb,
1633                                            output int addr_skip);
1634
1635   if (rw.element_kind == UVM_MEM) begin
         …
1642   end
1643   else if (rw.element_kind == UVM_REG) begin
1644     uvm_reg rg;
1645     if(rw.element == null || !$cast(rg,rw.element))
1646       `uvm_fatal("REG/CAST", {"uvm_reg_item 'element_kind' is UVM_REG, ",
1647                  "but 'element' does not point to a register: ",rw.get_name()})
1648     map_info = get_reg_map_info(rg);
1649     size = rg.get_n_bits();
1650   end
1651   else if (rw.element_kind == UVM_FIELD) begin
         …
1660   end
1661 endfunction
```

这里是分三种情况来分别操作的。第一个分支先处理这是由一个 uvm_mem 发起的读操作的情况，这里暂且先跳过，后面介绍 memory 操作时会详细介绍。第三个分支是处理 uvm_reg_field 的情况，也暂且先跳过，后面会详细介绍。

我们只看第二个分支，处理 uvm_reg 的情况。1645 到 1647 行一方面完成错误检查，另外一方面完成 rw.element 到 rg 的指针转换。1648 行调用 get_reg_map_info 函数，这个函数前面已经遇到过，其作用就是从此 uvm_reg_map 的 m_regs_info 数组中找到与 rg 对应的记录，返回之后把此值赋值给作为输出参数的 map_info。之后 size 作为输出参数被赋值为此 rg 的位宽。

回到 do_bus_write。1732 行比较古怪，用到了一个宏，这个主要是为了与 cadence 的 ius 兼容。这个宏的意思就是把 rw.addr 中的地址值逐个复制到 addrs 中。

1735 到 1737 行的分支在介绍 memory 的写操作时再介绍。

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
```

函数/任务：do_bus_write

```
1739    foreach (rw.value[val_idx]) begin: foreach_value
1740
1741      uvm_reg_data_t value = rw.value[val_idx];
1742
1743      /* calculate byte_enables */
1744      if (rw.element_kind == UVM_FIELD) begin
            …
1763      end
1764
1765      foreach(addrs[i]) begin: foreach_addr
1766
1767        uvm_sequence_item bus_req;
1768        uvm_reg_bus_op rw_access;
1769        uvm_reg_data_t data;
1770
1771
1772        data = (value >> (curr_byte*8)) & ((1'b1 << (bus_width * 8))-1);
1773
1774        `uvm_info(get_type_name(),
1775            $sformatf("Writing 'h%0h at 'h%0h via map \"%s\"...",
1776                  data, addrs[i], rw.map.get_full_name()), UVM_FULL);
1777
1778        if (rw.element_kind == UVM_FIELD) begin
1779          for (int z=0;z<bus_width;z++)
1780            rw_access.byte_en[z] = byte_en[curr_byte+z];
1781        end
1782
1783        rw_access.kind      = rw.kind;
1784        rw_access.addr      = addrs[i];
1785        rw_access.data      = data;
1786        rw_access.n_bits    = (n_bits > bus_width*8) ? bus_width*8 : n_bits;
1787        rw_access.byte_en   = byte_en;
1788
```

1739 行会遍历 rw 的 vlaue 数组。对于 memory 来说，可能进行块读写，此时 value 中会有多个数值。但是对于寄存器来说，value 中是只有一个值的。1744 到 1763 处理 uvm_reg_field 的情况，暂且先跳过。

1765 行开始遍历 addrs 中的第一个地址。以一个 64bit 的寄存器，总线为 16 位来例来介绍，1772 当遇到第一个地址时，cur_byte 的值为 0，data 的值将会是要写入的值 value 的低 16 位。当第一个地址写完成时，1828 行会把 cur_byte 的值加上总线位宽，这样当写第二个地址时，cur_byte 的值变为了 2，于是 1772 行的 data 的值将会是 value[31:16]，依次类推，第三次写操作的 data 值为 value[47:32]，第四次写操作的 data 值为 value[63:48]。

1778 到 1781 行先跳过。1783 到 1787 行给 uvm_reg_bus_op 型的变量 rw_access 赋值。kind 赋值为 rw 的 kind，即写操作，addr 赋值为 addrs 中相应的地址，data 赋

值为要写入的数据，n_bits 则根据 Xget_bus_infoX 中的第三个参数的返回值来定。这个值可能是大于总线位宽的，也可能是小于的。如总线是 16 位，但是寄存器只有 8 位，那么 n_bits 被赋值为 8。否则被赋值为总线位宽。1786 与 1829 行要结合起来看。假设系统总线是 16 位的，而要读写的寄存器是 23 位的，那么 addrs 中将会有两个地址，在第一次写操作时，rw_access 的 n_bits 被赋值为 16，而第二次写操作时被赋值为 7。

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：do_bus_write
1739    foreach (rw.value[val_idx]) begin: foreach_value
          …
1765      foreach(addrs[i]) begin: foreach_addr
            …
1789        adapter.m_set_item(rw);
1790        bus_req = adapter.reg2bus(rw_access);
1791        adapter.m_set_item(null);
1792
1793        if (bus_req == null)
1794          `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] didnt return a bus transacti
on"});
1795
1796        bus_req.set_sequencer(sequencer);
1797        rw.parent.start_item(bus_req,rw.prior);
1798
1799        if (rw.parent != null && rw_access.addr == addrs[0])
1800          rw.parent.mid_do(rw);
1801
1802        rw.parent.finish_item(bus_req);
1803        bus_req.end_event.wait_on();
1804
1805        if (adapter.provides_responses) begin
1806          uvm_sequence_item bus_rsp;
1807          uvm_access_e op;
1808          // TODO: need to test for right trans type, if not put back in q
1809          rw.parent.get_base_response(bus_rsp);
1810          adapter.bus2reg(bus_rsp,rw_access);
1811        end
1812        else begin
1813          adapter.bus2reg(bus_req,rw_access);
1814        end
1815
1816        if (rw.parent != null && rw_access.addr == addrs[addrs.size()-1])
1817          rw.parent.post_do(rw);
1818
1819        rw.status = rw_access.status;
1820
1821        `uvm_info(get_type_name(),
1822            $sformatf("Wrote 'h%0h at 'h%0h via map \"%s\": %s...",
1823                data, addrs[i], rw.map.get_full_name(), rw.status.name()), UVM_FULL)
```

```
1824
1825            if (rw.status == UVM_NOT_OK)
1826                break;
1827
1828            curr_byte += bus_width;
1829            n_bits -= bus_width * 8;
1830
1831        end: foreach_addr
1832
1833        foreach (addrs[i])
1834            addrs[i] = addrs[i] + map_info.mem_range.stride;
1835
1836    end: foreach_value
1837
1838 endtask: do_bus_write
```

1789 行调用 uvm_reg_adapter 的 m_set_item 函数：

```
文件：src/reg/uvm_reg_adapter.svh
类：uvm_reg_adapter
函数/任务：m_set_item

107    virtual function void m_set_item(uvm_reg_item item);
108      m_item = item;
109    endfunction
```

其中的 m_item 的定义为：

```
文件：src/reg/uvm_reg_adapter.svh
类：uvm_reg_adapter

102    local uvm_reg_item m_item;
```

这是 uvm_reg_adapter 的一个成员变量。m_set_item 比较简单，只是给 m_item 赋值。

1790 行调用 adapter 的 reg2bus 函数，这个函数是需要用户自定义的，它把 uvm_reg_bus_op 型的变量中的信息提取出来，转换成用户自定义的 transaction 形式。

1791 行再次调用 m_set_item，把 m_item 赋值为 null。

1793 行检查 bus_req 的有效性。1796 到 1797 行设置 sequencer，调用 start_item。这都是 sequence 机制中的内容。

1799 到 1800 行调用 sequence 的 mid_do 函数。这里比较有趣的是除了要判断这个 sequence 是否为 null 之外，还要判断一个地址值。根据这个判断，仅仅是在写第一个地址时，才会执行 mid_do。

1802 到 1803 行刚刚在介绍 do_write，当 adapter 不为 null 时已经介绍过了。这里与那里相同。

1805 到 1810 行判断这个 adapter 是否提供 response，如果提供，那么 1809 行得到这个 response。这也是 sequence 机制中的内容。1810 行调用 bus2reg，把 bus_req 中的信息转换为 rw_access 的形式。如果 adapter 不提供 response，那么就省去了索要 response 的语句，直接调用 bus2reg，把 bus_req 中的信息转换为 rw_access 的形式。

1816 到 1871 行调用 post_do，仅仅是在写最后一个地址完成后才调用。

1819 行给 rw 的 status 赋值。1826 行如果发现写操作没有完成，那么中断 foreach 循环，后面的就不必再写下去了。1828 与 1829 行前面已经介绍过了。1833 到 1834 行主要是用于 memory 的写操作中，这里先跳过。

至此，uvm_reg_map 的 do_bus_write 和 do_write 任务分析完毕，下节将回到 uvm_reg 的 do_write 任务。

18.4.8. uvm_reg::do_write(三)

文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

```
2252                if (system_map.get_auto_predict()) begin
2253                    if (rw.status != UVM_NOT_OK) begin
2254                        sample(value, -1, 0, rw.map);
2255                        m_parent.XsampleX(map_info.offset, 0, rw.map);
2256                    end
2257
2258                  do_predict(rw, UVM_PREDICT_WRITE);
2259                end
2260            end
2261
2262        endcase
2263
```

2252 行调用 uvm_reg_map 的 get_auto_predict 函数。函数的定义为：

文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_auto_predict

```
543     function bit    get_auto_predict(); return m_auto_predict; endfunction
```

函数只是返回 m_atuo_predict 的值。m_auto_predict 是一个 bit 型的变量，在 uvm_reg_map 的 new 函数中将其初始化成 0：

文件：src/reg/uvm_reg_map.svh

```
类：uvm_reg_map

  64      local bit                       m_auto_predict;

 601 function uvm_reg_map::new(string name = "uvm_reg_map");
 602      super.new((name == "") ? "default_map" : name);
 603      m_auto_predict = 0;
 604 endfunction
```

我们可以通过 set_auto_predict 函数来给其赋值：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：set_auto_predict

 536      function void set_auto_predict(bit on=1); m_auto_predict = on; endfunction
```

一般来说，在 register model 中都要打开 auto predict 功能。

当写操作正常完成时，rw.status 的值将会是 UVM_IS_OK，异常完成时，则为 UVM_NOT_OK，所以 2253 行是判断是否正常完成了，在正常完成的情况下，2254 行调用 sample 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：sample

1028      protected virtual function void sample(uvm_reg_data_t   data,
1029                                             uvm_reg_data_t   byte_en,
1030                                             bit              is_read,
1031                                             uvm_reg_map      map);
1032      endfunction
```

这是一个空函数，用户可以重载它，来进行 coverage 方面的相关操作。2255 行调用了 uvm_reg_block 的 XsampleX 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：XsampleX

1525 function void uvm_reg_block::XsampleX(uvm_reg_addr_t addr,
1526                                        bit              is_read,
1527                                        uvm_reg_map      map);
1528      sample(addr, is_read, map);
1529      if (parent != null) begin
1530          // ToDo: Call XsampleX in the parent block
1531          //       with the offset and map within that block's context
1532      end
1533 endfunction
```

这个函数也是与 coverage 收集相关的函数。从注释就可以看出，这部分功能尚

且没有完成。

2258 行在 auto_predict 功能打开情况下调用 do_predict 函数。这点也正是 auto_predict 名称的由来，如果 auto_predict 没有打开，那么这里就不会自动执行。

18.4.9. uvm_reg::do_predict

uvm_reg 的 do_predict 函数如下：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_predict

1941 function void uvm_reg::do_predict(uvm_reg_item          rw,
1942                                   uvm_predict_e         kind = UVM_PREDICT_DIREC
T,
1943                                   uvm_reg_byte_en_t be = -1);
1944
1945     uvm_reg_data_t reg_value = rw.value[0];
1946     m_fname = rw.fname;
1947     m_lineno = rw.lineno;
1948
1949     rw.status = UVM_IS_OK;
1950
1951     if (m_is_busy && kind == UVM_PREDICT_DIRECT) begin
1952         `uvm_warning("RegModel", {"Trying to predict value of register '",
1953                      get_full_name(),"' while it is being accessed"})
1954         rw.status = UVM_NOT_OK;
1955         return;
1956     end
1957
1958     foreach (m_fields[i]) begin
1959         rw.value[0] = (reg_value >> m_fields[i].get_lsb_pos()) &
1960                                     ((1 << m_fields[i].get_n_bits())-1);
1961         m_fields[i].do_predict(rw, kind, be>>(m_fields[i].get_lsb_pos()/8));
1962     end
1963
1964     rw.value[0] = reg_value;
1965
1966 endfunction: do_predict
```

这里输入的参数是 rw 和 UVM_PREDICT_WRITE，表示这是一次 write 操作完成后的调用。

1945 行取得寄存器的操作值。1951 到 1956 行的语句避免下面这种情况：当一个寄存器正在写操作时，调用 do_predict，且传入的第二个参数 kind 为函数的默认

值。kind 是一个 uvm_predict_e 类型的变量：

```
文件：src/reg/uvm_reg_model.svh
类：无

255     typedef enum {
256         UVM_PREDICT_DIRECT,
257         UVM_PREDICT_READ,
258         UVM_PREDICT_WRITE
259     } uvm_predict_e;
```

uvm_predict_e 用于表征预测的类型。对于 UVM_PREDICT_READ 和 UVM_PREDICT_ WRITE，我们可以很容易理解：二者分别是当读定操作完毕时会调用。但是很难理解 UVM_PREDICT_DIRECT。事实上，考虑如下一种情况：

DUT 中有一个计数器，这个计数器用于统计 DUT 接收到的包的数量。每当接收到一个包的时候，其值就需要加 1。我们需要测试这个计数器的正确性。一种简单的方法就是在验证环境的参考模型中每收到一个包就更新 register model 中此计数器的值。最后，把此计数器的值和 DUT 中计数器的值相比较。在更新 register model 中此计数器的值时，需要调用 do_predict 函数，且传入的参数是 UVM_PREDICT_DIRECT。

1958 到 1962 行依次调用所有 uvm_reg_field 的 do_predict 函数。1959 到 1960 行比较好理解，就是得到此 uvm_reg_field 的值，比较难于理解的是 1961 行调用 do_predict 时传入的第三个参数，我们接下来先看 uvm_reg_field 的 do_preidict 函数。

18.4.10. uvm_reg_field::do_predict(一)

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_predict

1084 function void uvm_reg_field::do_predict(uvm_reg_item      rw,
1085                                         uvm_predict_e     kind = UVM_PREDICT_
DIRECT,
1086                                         uvm_reg_byte_en_t be = -1);
1087
1088    uvm_reg_data_t field_val = rw.value[0] & ((1 << m_size)-1);
1089
1090    if (rw.status != UVM_NOT_OK)
1091      rw.status = UVM_IS_OK;
1092
```

其第三个参数是 uvm_reg_byte_en_t 类型的：

```
文件：src/reg/uvm_reg_model.svh
类：无

 86 typedef   bit unsigned [`UVM_REG_BYTENABLE_WIDTH-1:0]   uvm_reg_byte_en_t ;
```

其中的宏的定义为：

```
文件：src/macros/uvm_reg_defines.svh
类：无

 54 `ifndef UVM_REG_BYTENABLE_WIDTH
 55   `define UVM_REG_BYTENABLE_WIDTH ((`UVM_REG_DATA_WIDTH-1)/8+1)
 56 `endif
```

在默认情况下，UVM_REG_DATA_WIDTH 为 64，共有 8 个 byte，所以 UVM_REG_ BYTENABLE_WIDTH 的值为 8，分别控制 8 个 byte 的使能。注意的是 uvm_reg_byte_en_t 默认是 unsigned 的，也就是说它是无符号的，但是默认情况下，无论是 uvm_reg 的 do_predict 还是 uvm_reg_field 的 do_predict，其 be 值都是-1，是一个有符号的量，这应该怎么理解呢？由于-1 的补码为全 1，所以实际上 be 的值最终会是全 1。uvm_reg 的 do_predict 变量的 1961 行中，第三个参数根据要访问的 uvm_reg_field 的 lsb 把 be 向右移。以一个例子来说明，假如是 64bit 的寄存器，有 4 个 uvm_reg_field，每个分别为 16 位，于是在调用第一个的 do_predict 时，传入的参数为 uvm_reg 的 do_predict 的 be 参数，默认情况下为全 1。调用第二个时，需要把 be 向右移 2 位，因为前面 2 位是第一个 uvm_reg_field 占据了，以此类推。因此 be 位主要是控制寄存器中某个 byte 是否进行 predict。

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_predict

1093      // Assume that the entire field is enabled
1094      if (!be[0])
1095        return;
1096
1097      m_fname = rw.fname;
1098      m_lineno = rw.lineno;
1099
1100      case (kind)
1101
1102        UVM_PREDICT_WRITE:
1103          begin
1104            uvm_reg_field_cb_iter cbs = new(this);
1105
1106            if (rw.path == UVM_FRONTDOOR || rw.path == UVM_PREDICT)
1107                field_val = XpredictX(m_mirrored, field_val, rw.map);
1108
1109            m_written = 1;
1110
1111            for (uvm_reg_cbs cb = cbs.first(); cb != null; cb = cbs.next())
```

```
1112                    cb.post_predict(this, m_mirrored, field_val,
1113                                    UVM_PREDICT_WRITE, rw.path, rw.map);
1114
1115                field_val &= ('b1 << m_size)-1;
1116
1117              end
1118
1119            UVM_PREDICT_READ:
1120              begin
                    …
1152              end
1153
1154            UVM_PREDICT_DIRECT:
1155              begin
                    …
1162              end
1163          endcase
1164
1165          // update the mirror with predicted value
1166          m_mirrored = field_val;
1167          m_desired  = field_val;
1168          this.value = field_val;
1169
1170      endfunction: do_predict
```

回到 uvm_reg_field 的 do_predict。1094 行如果发现这个 uvm_reg_field 所在的 byte 没有打开 predict 功能，那么就直接返回。

在传入的第二个参数为 UVM_PREDICT_WRITE 时，1106 到 1107 行调用 XpredictX 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：XpredictX

1022  function uvm_reg_data_t uvm_reg_field::XpredictX (uvm_reg_data_t cur_val,
1023                                                    uvm_reg_data_t wr_val,
1024                                                    uvm_reg_map    map);
1025      uvm_reg_data_t mask = ('b1 << m_size)-1;
1026
1027      case (get_access(map))
1028        "RO":    return cur_val;
1029        "RW":    return wr_val;
1030        "RC":    return cur_val;
1031        "RS":    return cur_val;
1032        "WC":    return '0;
1033        "WS":    return mask;
1034        "WRC":   return wr_val;
1035        "WRS":   return wr_val;
1036        "WSRC":  return mask;
1037        "WCRS":  return '0;
1038        "W1C":   return cur_val & (~wr_val);
```

```
1039          "W1S":     return cur_val | wr_val;
1040          "W1T":     return cur_val ^ wr_val;
1041          "W0C":     return cur_val & wr_val;
1042          "W0S":     return cur_val | (~wr_val & mask);
1043          "W0T":     return cur_val ^ (~wr_val & mask);
1044          "W1SRC":  return cur_val | wr_val;
1045          "W1CRS":  return cur_val & (~wr_val);
1046          "W0SRC":  return cur_val | (~wr_val & mask);
1047          "W0CRS":  return cur_val & wr_val;
1048          "WO":       return wr_val;
1049          "WOC":     return '0;
1050          "WOS":     return mask;
1051          "W1":       return (m_written) ? cur_val : wr_val;
1052          "WO1":     return (m_written) ? cur_val : wr_val;
1053          default: return wr_val;
1054       endcase
1055
1056       `uvm_fatal("RegModel", "uvm_reg_field::XpredictX(): Internal error");
1057       return 0;
1058  endfunction: XpredictX
```

函数不难，只是稍微有点长，其主要用意就是根据此 uvm_reg_field 的存取策略，来返回写操作后寄存器中的数值。

1111 到 1113 行则调用此 uvm_reg_field 的所有的 post_predict 函数。

1115 行根据此 uvm_reg_field 的有效位数，把 field_val 中的无效位去除。

在传入的第二个参数为 UVM_PREDICT_READ 与 UVM_PREDICT_DIRECT 时先跳过。

1166 到 1168 行把此 uvm_reg_field 中三个存放数据的变量的值更新为刚刚预测过的值。

到此 uvm_reg 的 do_predict 和 uvm_reg_field 的 do_predict 分析完毕。回到 uvm_reg 的 do_write 任务。

18.4.11. uvm_reg::do_write(四)

接着分析 uvm_reg 的 do_write 任务：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

2264       value = rw.value[0];
```

```
2265
2266      // POST-WRITE CBS - REG
2267      for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2268          cb.post_write(rw);
2269      post_write(rw);
2270
2271      // POST-WRITE CBS - FIELDS
2272      foreach (m_fields[i]) begin
2273          uvm_reg_field_cb_iter cbs = new(m_fields[i]);
2274          uvm_reg_field f = m_fields[i];
2275
2276          rw.element = f;
2277          rw.element_kind = UVM_FIELD;
2278          rw.value[0] = (value >> f.get_lsb_pos()) & ((1<<f.get_n_bits())-1);
2279
2280          for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2281              cb.post_write(rw);
2282          f.post_write(rw);
2283      end
2284
2285      rw.value[0] = value;
2286      rw.element = this;
2287      rw.element_kind = UVM_REG;
2288
2289      // REPORT
2290      if (uvm_report_enabled(UVM_HIGH)) begin
2291        string path_s,value_s;
2292        if (rw.path == UVM_FRONTDOOR)
2293          path_s = (map_info.frontdoor != null) ? "user frontdoor" :
2294                                                  {"map ",rw.map.get_full_name()};
2295        else
2296          path_s = (get_backdoor() != null) ? "user backdoor" : "DPI backdoor";
2297
2298        value_s = $sformatf("=0x%0h",rw.value[0]);
2299
2300        `uvm_info("RegModel", {"Wrote register via ",path_s,": ",
2301                               get_full_name(),value_s},UVM_HIGH)
2302      end
2303
2304      m_write_in_progress = 1'b0;
2305
2306      XatomicX(0);
2307
2308  endtask: do_write
```

在执行完 2258 行后，2267 到 2269 行调用此寄存器的 post_write 函数。2272 到 2283 调用此寄存器所有的 uvm_reg_field 的 post_write 函数。2290 到 2302 行打印信息。

2306 行释放从 m_atomicc 中得到的键值，此次写操作完毕。

至此，以 FRONTDOOR 形式的 write 操作分析完成，小结下，在没有自定义

frontdoor 情况下，FRONTDOOR 操作最终会转换为 uvm_reg_map 的 do_write 任务，do_write 任务将会查看系统是否设置了 adapter，如果没有设置那么就直接启动 sequence，让 sequencer 发送 uvm_reg_item 类型的 transactiion。如果设置了，那么调用 do_bus_write 任务。在 uvm_reg_map 的 do_write 完成后，如果 auto predict 功能打开了，uvm_reg 的 do_write 会收集 coverage 信息，并且根据写入的值更新 register model 中的寄存器的值。在整个 write 操作过程，在操作开始前，会先调用所有 uvm_reg_field 的 pre_write，再调用此 uvm_reg 的 pre_write。在把寄存器的值写入 DUT 后，register model 的值也相应的被预测后，会调用 uvm_reg 的 post_write，再调用各个 uvm_reg_field 的 post_write。

18.5. uvm_reg 的 write 操作：BACKDOOR

上节分析了以 FRONTDOOR 形式来进行 uvm_reg 的 write 操作，本节分析以 BACKDOOR 形式。

18.5.1. uvm_reg::do_write(五)

在 uvm_reg 的 do_write 函数，2190 到 2226 行为 BACKDOOR 操作相关代码：

文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_write

```
2186      // EXECUTE WRITE...
2187      case (rw.path)
2188
2189        // ...VIA USER BACKDOOR
2190        UVM_BACKDOOR: begin
2191          uvm_reg_data_t final_val;
2192          uvm_reg_backdoor bkdr = get_backdoor();
2193
2194          // Mimick the final value after a physical read
2195          rw.kind = UVM_READ;
2196          if (bkdr != null)
2197            bkdr.read(rw);
2198          else
2199            backdoor_read(rw);
```

```
2200
2201            if (rw.status == UVM_NOT_OK) begin
2202              m_write_in_progress = 1'b0;
2203              return;
2204            end
2205
2206            begin
2207                foreach (m_fields[i]) begin
2208                    uvm_reg_data_t field_val;
2209                    int lsb = m_fields[i].get_lsb_pos();
2210                    int sz  = m_fields[i].get_n_bits();
2211                    field_val = m_fields[i].XpredictX((rw.value[0] >> lsb) & ((1<<sz)-1),
2212                                                      (value >> lsb) & ((1<<sz)-1),
2213                                                      rw.local_map);
2214                    final_val |= field_val << lsb;
2215                end
2216            end
2217            rw.kind = UVM_WRITE;
2218            rw.value[0] = final_val;
2219
2220            if (bkdr != null)
2221              bkdr.write(rw);
2222            else
2223              backdoor_write(rw);
2224
2225            do_predict(rw, UVM_PREDICT_WRITE);
2226        end
2227
2228        UVM_FRONTDOOR: begin
                …
2260        end
2261
2262    endcase
2263
```

2192 行调用 get_backdoor 函数来得到用户自定义的 backdoor 操作。函数的代码来：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_backdoor

1308 function uvm_reg_backdoor uvm_reg::get_backdoor(bit inherited = 1);
1309
1310    if (m_backdoor == null && inherited) begin
1311      uvm_reg_block blk = get_parent();
1312      uvm_reg_backdoor bkdr;
1313      while (blk != null) begin
1314        bkdr = blk.get_backdoor();
1315        if (bkdr != null) begin
1316          m_backdoor = bkdr;
1317          break;
```

```
1318            end
1319            blk = blk.get_parent();
1320          end
1321        end
1322        return m_backdoor;
1323    endfunction: get_backdoor
```

如果定义了 backdoor，那么直接返回。否则整个函数就是递归的向上查找是否哪个 uvm_reg_block 定义了 backdoor，如果定义了那么就直接返回，否则返回 null。1314 行用到了 uvm_reg_block 的 get_backdoor 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：get_backdoor

1928  function uvm_reg_backdoor uvm_reg_block::get_backdoor(bit inherited = 1);
1929      if (backdoor == null && inherited) begin
1930        uvm_reg_block blk = get_parent();
1931        while (blk != null) begin
1932          uvm_reg_backdoor bkdr = blk.get_backdoor();
1933          if (bkdr != null)
1934            return bkdr;
1935          blk = blk.get_parent();
1936        end
1937      end
1938      return this.backdoor;
1939  endfunction: get_backdoor
```

与 uvm_reg 的 get_backdoor 类似，可见，如果没有自定义 backdoor，那么最终 uvm_reg 的 get_backdoor 将会返回 null。

回到 uvm_reg 的 do_write，2196 行判断是否自定义了 backdoor。我们考虑没有自定义的情况，于是 2199 行的分支，调用 backdoor_read 函数。

18.5.2. uvm_reg::backdoor_read

函数定义为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：backdoor_read

2644  task   uvm_reg::backdoor_read (uvm_reg_item rw);
2645    rw.status = backdoor_read_func(rw);
2646  endtask
```

直接调用 backdoor_read_func 函数。

18.5.3. uvm_reg:: backdoor_read_func(一)

函数的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：backdoor_read_func

2651 function uvm_status_e uvm_reg::backdoor_read_func(uvm_reg_item rw);
2652    uvm_hdl_path_concat paths[$];
2653    uvm_reg_data_t val;
2654    bit ok=1;
2655    get_full_hdl_path(paths,rw.bd_kind);
```

2655 行调用 get_full_hdl_path 函数。

18.5.4. uvm_reg::get_full_hdl_path(一)

函数代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_full_hdl_path

1442 function void uvm_reg::get_full_hdl_path(ref uvm_hdl_path_concat paths[$],
1443                                           input string kind = "",
1444                                           input string separator = ".");
1445
1446    if (kind == "") begin
1447        if (m_regfile_parent != null)
1448            kind = m_regfile_parent.get_default_hdl_path();
1449        else
1450            kind = m_parent.get_default_hdl_path();
1451    end
1452
```

1446 行会首先判断输入的 kind 的值，在调用这个函数时，传入的是 rw.bd_kind，而这个值一直没有设置过，所以这里 1446 行的条件满足。1448 行在设置了 reg file 的情况下，调用 uvm_reg_file 的 get_default_hdl_path：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file
函数/任务：get_default_hdl_path

382 function string uvm_reg_file::get_default_hdl_path();
383     if (default_hdl_path == "") begin
384         if (m_rf != null)
385             return m_rf.get_default_hdl_path();
386         else
387             return parent.get_default_hdl_path();
388     end
389     return default_hdl_path;
390 endfunction
```

这里用到了 default_hdl_path 变量：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file

 38      local string                    default_hdl_path = "RTL";
```

我们一直没有对其进行设置过，所以其值依然为 RTL。所以 get_default_hdl_path 的返回值为 RTL。

1450 行在没有设置 reg file 的情况下，调用 uvm_reg_block 的 get_default_hdl_path：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：get_default_hdl_path

2061 function string uvm_reg_block::get_default_hdl_path();
2062     if (default_hdl_path == "" && parent != null)
2063         return parent.get_default_hdl_path();
2064     return default_hdl_path;
2065 endfunction
```

这里也用到了 default_hdl_path 变量，其值与 uvm_reg_file 中的 default_hdl_path 一样，被设置为了 RTL：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block

  53      local string                default_hdl_path = "RTL";
```

所以无论是 1448 还是 1450 行，最终都将会返回 RTL。当然了，这是在默认的情况下。如果用户自定义了，那么自然会返回别的。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
```

```
函数/任务：get_full_hdl_path

1453      if (!has_hdl_path(kind)) begin
1454          `uvm_error("RegModel",
1455              {"Register ",get_full_name()," does not have hdl path defined for abstraction '",ki
nd,"'"})
1456          return;
1457      end
1458
```

1453 行调用 has_hdl_path 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：has_hdl_path

1979 function bit   uvm_reg_block::has_hdl_path(string kind = "");
1980    if (kind == "") begin
1981       kind = get_default_hdl_path();
1982    end
1983    return hdl_paths_pool.exists(kind);
1984 endfunction
```

这个函数会从 m_hdl_paths_pool 中寻找是否有对应 RTL 的记录。如果在调用 uvm_reg 的 configure 时指定了 hdl 路径，或者虽然没有指定，但是后面通过 add_hdl_path_slice 指定了路径，那么 m_hdl_paths_pool 中是会有一条索引为"RTL"的记录的。

1454 行在检测到没有对应路径的情况下，会给出错误提示，并直接返回。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_full_hdl_path

1459      begin
1460          uvm_queue #(uvm_hdl_path_concat) hdl_paths = m_hdl_paths_pool.get(kind);
1461          string parent_paths[$];
1462
1463          if (m_regfile_parent != null)
1464              m_regfile_parent.get_full_hdl_path(parent_paths, kind, separator);
1465          else
1466              m_parent.get_full_hdl_path(parent_paths, kind, separator);
1467
```

1460 行从 m_hdl_paths_pool 中取得此寄存器的本地路径所存储的队列。1464 和 1466 行分别调用 uvm_reg_file 和 uvm_reg_block 的 get_full_hdl_path 来得到各自的父路径。什么叫父路径？假设一个 uvm_reg 的完整路径为 top_tb.dut_inst.mac_inst.reg，那么 top_tb.dut_inst.mac_inst 就是父路径，而 reg 则是本地路径。

18.5.5. uvm_reg_block::get_full_hdl_path

本节先来看 uvm_reg_block 的 get_full_hdl_path：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：get_full_hdl_path

2011 function void uvm_reg_block::get_full_hdl_path(ref string paths[$],
2012                                                 input string kind = "",
2013                                                 string separator = ".");
2014
2015     if (kind == "")
2016         kind = get_default_hdl_path();
2017
2018     paths.delete();
2019     if (is_hdl_path_root(kind)) begin
2020         if (root_hdl_paths[kind] != "")
2021             paths.push_back(root_hdl_paths[kind]);
2022         return;
2023     end
2024
2025     if (!has_hdl_path(kind)) begin
2026         `uvm_error("RegModel",{"Block does not have hdl path defined for abstraction '",kin
d,"'"})
2027         return;
2028     end
2029
```

2015 到 2016 行取得路径的种类，默认情况下为”RTL”。2018 行清空 paths 数组，以防止其中的内容在最后返回后被误认为是由 get_full_hdl_path 添加进去的。

2019 行调用 is_hdl_path_root 函数：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：is_hdl_path_root

2096 function bit   uvm_reg_block::is_hdl_path_root (string kind = "");
2097    if (kind == "")
2098       kind = get_default_hdl_path();
2099
2100    return root_hdl_paths.exists(kind);
2101 endfunction
```

这个函数就是用于检查此 uvm_reg_block 的 root_hdl_paths 中是否定义了 kind 类型（在我们的例子中为”RTL”）的路径。一般的，在最顶层的 uvm_reg_block 中，我们会调用 set_hdl_path_root，把最顶层的 uvm_reg_block 的路径写入此 uvm_reg_block

的 root_hdl_paths 中。set_hdl_path_root 如下：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：set_hdl_path_root

2086 function void uvm_reg_block::set_hdl_path_root (string path, string kind = "RTL");
2087    if (kind == "")
2088       kind = get_default_hdl_path();
2089
2090    root_hdl_paths[kind] = path;
2091 endfunction
```

在定义了情况下，2020 行查看 root_hdl_paths 中是否有对应 king（即”RTL”）的记录。因此，假如这是一个最顶层的 uvm_reg_block，那么 get_full_hdl_path 的 2021 行会把此绝对路径放入 paths 数组中，并直接返回。当然了，即使不是最顶层的 uvm_reg_block，那么假如调用了 set_hdl_path_root 函数，也是会把此路径放入 paths 中，直接返回的。

假如没有设置过 root_hdl_paths 数组，2025 行到 2028 行会检查此 uvm_reg_block 的 hdl_paths_pool 中是否定义了索引为 kind，即”RTL”的路径。如果没有定义过，那么给出错误提示，并且返回。

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：get_full_hdl_path

2030      begin
2031         uvm_queue #(string) hdl_paths = hdl_paths_pool.get(kind);
2032         string parent_paths[$];
2033
2034         if (parent != null)
2035            parent.get_full_hdl_path(parent_paths, kind, separator);
2036
2037         for (int i=0; i<hdl_paths.size();i++) begin
2038            string hdl_path = hdl_paths.get(i);
2039
2040            if (parent_paths.size() == 0) begin
2041               if (hdl_path != "")
2042                  paths.push_back(hdl_path);
2043
2044               continue;
2045            end
2046
2047            foreach (parent_paths[j])  begin
2048               if (hdl_path == "")
2049                  paths.push_back(parent_paths[j]);
2050               else
2051                  paths.push_back({ parent_paths[j], separator, hdl_path });
2052            end
```

```
2053            end
2054        end
2055
2056    endfunction
```

2031 行从 hdl_paths_pool 中得到索引为 kind 的路径所在的队列。2034 与 2035 行递归的调用父 uvm_reg_block 的 get_full_hdl_path 函数，得到父路径。

在父 uvm_reg_block 没有设置过路径或者这本身就是一个最顶层的 uvm_reg_block 时，2042 行把本地路径放入 paths 联合数组中。

当父 uvm_reg_block 的 get_full_hdl_path 返回了路径时，2047 到 2052 行把这些路径加上本地路径后放入 paths 联合数组中。假设父路径为 top_tb.dut_inst，本地路径为 mac_inst，那么最终放入 paths 中的路径为 top_tb.dut_inst.mac_inst。

总结一下 uvm_reg_block 的 get_full_hdl_path 函数，假设调用过此 uvm_reg_block 的 set_hdl_path_root 函数设置过绝对路径，那么无论这个 uvm_reg_block 是不是最顶层的 uvm_reg_block，都会直接返回这个路径。否则的话，这个函数会返回父路径加上本地路径，也即此 uvm_reg_block 的绝对路径。无论哪种情况，get_full_hdl_path 最终都是返回了此 uvm_reg_block 的绝对路径。

18.5.6. uvm_reg_file::get_full_hdl_path

uvm_reg_file 的 get_full_hdl_path 如下：

```
文件：src/reg/uvm_reg_file.svh
类：uvm_reg_file
函数/任务：get_full_hdl_path

336 function void uvm_reg_file::get_full_hdl_path(ref string paths[$],
337                                                 input string kind = "",
338                                                 input string separator = ".");
339     if (kind == "")
340         kind = get_default_hdl_path();
341
342     if (!has_hdl_path(kind)) begin
343         `uvm_error("RegModel",{"Register file does not have hdl path defined for abstraction
  '",kind,"'"})
344         return;
345     end
346
347     paths.delete();
348
349     begin
350         uvm_queue #(string) hdl_paths = hdl_paths_pool.get(kind);
```

```
351         string parent_paths[$];
352
353         if (m_rf != null)
354             m_rf.get_full_hdl_path(parent_paths, kind, separator);
355         else if (parent != null)
356             parent.get_full_hdl_path(parent_paths, kind, separator);
357
358         for (int i=0; i<hdl_paths.size();i++) begin
359             string hdl_path = hdl_paths.get(i);
360
361             if (parent_paths.size() == 0) begin
362                 if (hdl_path != "")
363                     paths.push_back(hdl_path);
364
365                 continue;
366             end
367
368             foreach (parent_paths[j])  begin
369                 if (hdl_path == "")
370                     paths.push_back(parent_paths[j]);
371                 else
372                     paths.push_back({ parent_paths[j], separator, hdl_path });
373             end
374         end
375     end
376
377 endfunction
```

函数与 uvm_reg_block 的 get_full_hdl_path 非常类似，可以看出，它最终会返回此 uvm_reg_file 的绝对路径，不详细介绍。

18.5.7. uvm_reg::get_full_hdl_path(二)

回到 uvm_reg 的 get_full_hdl_path。经过 1463 到 1466 行之后，parent_paths 中装载的都将会是父路径，且此路径是绝对路径。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get_full_hdl_path

1459     begin
1460         uvm_queue #(uvm_hdl_path_concat) hdl_paths = m_hdl_paths_pool.get(kind);
1461         string parent_paths[$];
1462
1463         if (m_regfile_parent != null)
1464             m_regfile_parent.get_full_hdl_path(parent_paths, kind, separator);
1465         else
```

```
1466              m_parent.get_full_hdl_path(parent_paths, kind, separator);
1467
1468          for (int i=0; i<hdl_paths.size();i++) begin
1469              uvm_hdl_path_concat hdl_concat = hdl_paths.get(i);
1470
1471              foreach (parent_paths[j])  begin
1472                  uvm_hdl_path_concat t = new;
1473
1474                  foreach (hdl_concat.slices[k]) begin
1475                      if (hdl_concat.slices[k].path == "")
1476                          t.add_path(parent_paths[j]);
1477                      else
1478                          t.add_path({ parent_paths[j], separator, hdl_concat.slices[k].path },
1479                                       hdl_concat.slices[k].offset,
1480                                       hdl_concat.slices[k].size);
1481                  end
1482                  paths.push_back(t);
1483              end
1484          end
1485      end
1486  endfunction
```

1468 到 1484 行将会把父路径和此 uvm_reg 的本地路径拼接起来，放入 paths 数组输出。这里需要注意的是，如果本地路径为空，那么放入 paths 的就只是纯粹的父路径，里面没有任何 offset 和 size 信息；如果不为空，那么就是带着本地路径的最终的拼接后的路径，这个路径里面带有 offset 和 size 信息。无论这个 uvm_reg 中是否有多个 uvm_reg_field，假如父路径只设置了一条，那么返回的 paths 中将只有一条记录。uvm_reg 的多 field 所带来的多路径只体现在 pahts 中的那一条记录中的动态数组中。只有一个 uvm_reg_field 时，此动态数组大小为 1，有多个 uvm_reg_field 时，此动态数组大小大于 1。

18.5.8. uvm_reg:: backdoor_read_func(二)

回到 backdoor_read_func 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：backdoor_read_func

2655    get_full_hdl_path(paths,rw.bd_kind);
2656    foreach (paths[i]) begin
2657        uvm_hdl_path_concat hdl_concat = paths[i];
2658        val = 0;
2659        foreach (hdl_concat.slices[j]) begin
2660            `uvm_info("RegMem", {"backdoor_read from %s ",
```

```
2661                        hdl_concat.slices[j].path},UVM_DEBUG)
2662
2663            if (hdl_concat.slices[j].offset < 0) begin
2664                ok &= uvm_hdl_read(hdl_concat.slices[j].path,val);
2665                continue;
2666            end
2667            begin
2668                uvm_reg_data_t slice;
2669                int k = hdl_concat.slices[j].offset;
2670
2671                ok &= uvm_hdl_read(hdl_concat.slices[j].path, slice);
2672
2673                repeat (hdl_concat.slices[j].size) begin
2674                    val[k++] = slice[0];
2675                    slice >>= 1;
2676                end
2677            end
2678        end
2679
2680        val &= (1 << m_n_bits)-1;
2681
2682        if (i == 0)
2683            rw.value[0] = val;
2684
2685        if (val != rw.value[0]) begin
2686            `uvm_error("RegModel", $sformatf("Backdoor read of register %s with multiple H
DL copies: values are not the same: %0h at path '%s', and %0h          at path '%s'. Returning fi
rst value.",
2687                       get_full_name(),
2688                       rw.value[0], uvm_hdl_concat2string(paths[0]),
2689                       val, uvm_hdl_concat2string(paths[i])));
2690            return UVM_NOT_OK;
2691         end
2692         `uvm_info("RegMem",
2693             $sformatf("returned backdoor value 0x%0x",rw.value[0]),UVM_DEBUG);
2694
2695    end
2696
2697    rw.status = (ok) ? UVM_IS_OK : UVM_NOT_OK;
2698    return rw.status;
2699 endfunction
```

2655 行得到了此 uvm_reg 的绝对路径信息。2656 行遍历 paths 联合数组，2659 行遍历 hdl_concat 中的动态数组 slices 中的每一条记录。2663 行如果发现此条记录的 offset 小于 0，那么调用 uvm_hdl_read 函数，得到寄存器的值，放入 val 中。否则的话也是调用 uvm_hdl_read 函数，只是把得到的值，放在 val 的特定的位中。什么情况下 offset 会小于 0？在调用 uvm_reg 的 configure 函数时，如果指定了 hdl 路径，那么 configure 函数会调用 add_hdl_path_slice，传入的 offset 参数为-1。所以 offset 小于 0 也即意味着只有这一个 uvm_reg_field。在 offset 不为 0 的情况下，那么也即意味着有多个 uvm_reg_field。此时 val 的不同位需要多次调用 uvm_hdl_read 函数来

获取。获取到之后，2669 和 2674 行将根据 offset 的值把 val 的不同的位赋值为读取的值。

这里用到了 uvm_hdl_read 函数，这是一个用 C 语言写成的函数，其作用就是根据输入的字符串形式路径信息来获得这个这个路径所表示的寄存器的值。这里不详细展开。

2680 行根据此寄存器的位宽，去除 val 中的无效位。2682 和 2683 行根据 i 的值给 rw.value[0]赋值。如果 i 为 1，表示此寄存器只有一个路径，因此得到的值是唯一的。但是假如有两个路径，那么可能会得到两个值，在这种情况下 2685 行就会检测到错误信息，并给出提示，2690 行直接返回。

正常情况下，2697 行会根据 ok 的状态来给 rw 的 status 赋值，并且把此值返回。ok 的值是在调用 uvm_hdl_read 时候确定的，如果 uvm_hdl_read 不成功，如输入的路径找不到等，那么 ok 就会是 0，从而 rw.status 会被赋值为 UVM_NOT_OK。

18.5.9. uvm_reg::do_write(六)

回到 uvm_reg 的 do_write 函数，2199 行通过 backdoor_read 得到了要写入寄存器的当前值。2201 检测刚刚的 backdoor_read 中是否出错，如果出错了就直接返回。

2206 到 2216 行通过调用所有 uvmj_reg_field 的 XpredictX 函数，返回经过修正过后的要写入寄存器的值。为什么要经过这一步呢？假如某寄存器为 RO 形式的，其值为 9，我们意图把 8 写进去，如果通过 FRONTDOOR 的形式，那么最终此寄存器的值为 9，8 是不会被写入的。而以 BACKDOOR 形式的 write 函数是要尽量模仿 FRONTDOOR，所以这里也不能把 8 写进去。这里，经过 2206 和 2216 行之后，要写入的 final_val 值变为了 9。这样接下来通过 BACKDOOR 把 9 这个值写入，从外部来看，此寄存器还是 RO 形式的，从而 BACKDOOR 形式的 write 函数尽量模仿了 FRONTDOOR 操作。

2218 行把要写入的 rw.value[0]的值赋值为经过修正过的值。2220 到 2223 行把此值写入。2225 行调用 do_predict 函数，这里与用 FRONTDOOR 时的一样，所以不再重复说明。

18.6. uvm_reg 的 read 操作

本节介绍 uvm_reg 的 read 操作。

18.6.1. uvm_reg 的 read 与 XreadX

read 函数的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：read

2312 task uvm_reg::read(output uvm_status_e          status,
2313                    output uvm_reg_data_t        value,
2314                    input  uvm_path_e            path = UVM_DEFAULT_PATH,
2315                    input  uvm_reg_map           map = null,
2316                    input  uvm_sequence_base parent = null,
2317                    input  int                   prior = -1,
2318                    input  uvm_object            extension = null,
2319                    input  string                fname = "",
2320                    input  int                   lineno = 0);
2321      XatomicX(1);
2322      XreadX(status, value, path, map, parent, prior, extension, fname, lineno);
2323      XatomicX(0);
2324 endtask: read
```

2321 与 2323 行联合起来组成一个原子操作。2322 行直接调用 XreadX 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：XreadX

2329 task uvm_reg::XreadX(output uvm_status_e          status,
2330                      output uvm_reg_data_t        value,
2331                      input  uvm_path_e            path,
2332                      input  uvm_reg_map           map,
2333                      input  uvm_sequence_base parent = null,
2334                      input  int                   prior = -1,
2335                      input  uvm_object            extension = null,
2336                      input  string                fname = "",
2337                      input  int                   lineno = 0);
2338
2339      // create an abstract transaction for this operation
2340      uvm_reg_item rw;
```

```
2341      rw = uvm_reg_item::type_id::create("read_item",,get_full_name());
2342      rw.element      = this;
2343      rw.element_kind = UVM_REG;
2344      rw.kind         = UVM_READ;
2345      rw.value[0]     = 0;
2346      rw.path         = path;
2347      rw.map          = map;
2348      rw.parent       = parent;
2349      rw.prior        = prior;
2350      rw.extension    = extension;
2351      rw.fname        = fname;
2352      rw.lineno       = lineno;
2353
2354      do_read(rw);
2355
2356      status = rw.status;
2357      value = rw.value[0];
2358
2359  endtask: XreadX
```

2342 到 2352 行把读操作的相关参数放入 rw 中。其中要注意的是 2343 行表示这是由一个 uvm_reg 发起的读操作，2344 行表示这是一个读操作，2345 行把 rw.value[0]置为 0。在读操作中，这个值应该是最后的返回值，这里为了避免出错，把其值初始化为 0。2354 行调用 do_read 任务。

18.6.2. uvm_reg::do_read(一)

任务的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_read

2364  task uvm_reg::do_read(uvm_reg_item rw);
2365
2366      uvm_reg_cb_iter  cbs = new(this);
2367      uvm_reg_map_info map_info;
2368      uvm_reg_addr_t   value;
2369
2370      m_fname  = rw.fname;
2371      m_lineno = rw.lineno;
2372
2373      if (!Xcheck_accessX(rw,map_info,"read()"))
2374        return;
2375
2376      m_read_in_progress = 1'b1;
```

```
2377
2378      rw.status = UVM_IS_OK;
2379
2380      // PRE-READ CBS - FIELDS
2381      foreach (m_fields[i]) begin
2382          uvm_reg_field_cb_iter cbs = new(m_fields[i]);
2383          uvm_reg_field f = m_fields[i];
2384          rw.element = f;
2385          rw.element_kind = UVM_FIELD;
2386          m_fields[i].pre_read(rw);
2387          for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2388              cb.pre_read(rw);
2389      end
2390
2391      rw.element = this;
2392      rw.element_kind = UVM_REG;
2393
2394      // PRE-READ CBS - REG
2395      pre_read(rw);
2396      for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2397          cb.pre_read(rw);
2398
2399      if (rw.status != UVM_IS_OK) begin
2400        m_read_in_progress = 1'b0;
2401
2402        return;
2403      end
2404
```

2373 行调用 Xcheck_accessX 函数，其作用就是检查要读取的寄存器是否能够按照指定的方式完成读操作，即对于 BACKDOOR 来说，检查是否设置了 hdl 路径；对于 FRONTDOOR 来说，检查是否配置了 uvm_reg_map 等。这个函数在 write 操作的 FRONTDOOR 方式一节已经讲述过了，这里不重复说明。

2376 行标志读操作开始，2378 行把 status 初始化为 UVM_IS_OK。2381 到 2389 行依次调用此 uvm_reg 的所有的 uvm_reg_field 的 pre_read 回调函数，2395 到 2397 则调用此寄存器的 pre_read 函数。2399 到 2403 行检查经过前面的 pre_read 函数之后，status 的值是否被置为 UVM_NOT_OK，如果是则直接返回，表明读操作出错。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_read

2405      // EXECUTE READ...
2406      case (rw.path)
2407
2408          // ...VIA USER BACKDOOR
2409          UVM_BACKDOOR: begin
2410              uvm_reg_backdoor bkdr = get_backdoor();
2411
```

```
2412            if (bkdr != null)
2413              bkdr.read(rw);
2414            else
2415              backdoor_read(rw);
2416
2417            value = rw.value[0];
2418
2419            // Need to clear RC fields, set RS fields and mask WO fields
2420            if (rw.status != UVM_NOT_OK) begin
2421
2422                uvm_reg_data_t wo_mask = 0;
2423
2424                foreach (m_fields[i]) begin
2425                    string acc = m_fields[i].get_access(uvm_reg_map::backdoor());
2426                    if (acc == "RC" ||
2427                        acc == "WRC" ||
2428                        acc == "W1SRC" ||
2429                        acc == "W0SRC") begin
2430                        value &= ~(((1<<m_fields[i].get_n_bits())-1)
2431                                                    << m_fields[i].get_lsb_pos());
2432                    end
2433                    else if (acc == "RS" ||
2434                                acc == "WRS" ||
2435                                acc == "W1CRS" ||
2436                                acc == "W0CRS") begin
2437                        value |= (((1<<m_fields[i].get_n_bits())-1)
2438                                                    << m_fields[i].get_lsb_pos());
2439                    end
2440                    else if (acc == "WO" ||
2441                                acc == "WOC" ||
2442                                acc == "WOS" ||
2443                                acc == "WO1") begin
2444                        wo_mask |= ((1<<m_fields[i].get_n_bits())-1)
2445                                                    << m_fields[i].get_lsb_pos();
2446                    end
2447                end
2448
2449                if (value != rw.value[0]) begin
2450                  if (bkdr != null)
2451                      bkdr.read(rw);
2452                  else
2453                      backdoor_read(rw);
2454                end
2455
2456                rw.value[0] &= ~wo_mask;
2457                do_predict(rw, UVM_PREDICT_READ);
2458            end
2459        end
2460
```

2406 行开始按照指定的读操作的方式来执行读操作，先来看 BACKDOOR 方式。

2412 到 2415 行根据用户是否设置了自己的 backdoor 来调用不同的函数进行读操作。默认情况是没有定义的，这里执行的是 backdoor_read 函数。这个函数在 write 操作的 BACKDOOR 方式一节已经讲述过，不在重复。

2417 行把读取到的值赋值给 value，2424 到 2447 行对 value 值进行操作：对于那些读清的 uvm_reg_field，把 value 中的相应位置为 0；对于那些读置位（RS，即读操作后变为 1）的 uvm_reg_field，把 value 中的相应位置为 1；对于那些 WO 的 uvm_reg_field，这里取得这些位的掩码。2449 行判断 value 的值是否与 rw.value[0] 是否一致。什么情况下会不一致？只有在存在读清或者读置位的 field 的情况下会不一致。按照 UVM 的解释，read 操作的 BACKDOOR 方式应该尽量模仿 FRONTDOOR 方式，所以这里发现不一致之后，应该把 value 的值写入到此寄存器中，即把应该读清的写入 0，把应该读置位的写入 1。但是 2450 到 2453 行的代码却让我们相当费解。从 2449 一直到 2458 整个 BACKDOOR 方式操作完成，我们都没有看到往 dut 中写东西。其实这里是 UVM 的一个 bug。在 UVM 的开发代号中，此 bug 的 ID 为 3631，有兴趣的读者可以去看看。

18.6.3. uvm_reg_field::do_predict(二)

2457 行通过调用 do_predict 函数来变更 register model 中此寄存器的值，这里传入的参数是 UVM_PREDICT_READ。在 write 操作中，曾经调用过这个函数，传入的参数是 UVM_PREDICT_WRITE。关于这两个参数的区别，在 uvm_reg 的 do_predict 中是看不出区别的，真正的区别是发生在 uvm_reg_field 的 do_predict 中体现：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_predict

1084 function void uvm_reg_field::do_predict(uvm_reg_item      rw,
1085                                         uvm_predict_e     kind = UVM_PREDICT_
DIRECT,
1086                                         uvm_reg_byte_en_t be = -1);
1087
1088    uvm_reg_data_t field_val = rw.value[0] & ((1 << m_size)-1);
1089
1090    if (rw.status != UVM_NOT_OK)
1091      rw.status = UVM_IS_OK;
1092
1093    // Assume that the entire field is enabled
1094    if (!be[0])
1095      return;
1096
1097    m_fname = rw.fname;
```

```
1098      m_lineno = rw.lineno;
1099
1100      case (kind)
1101
1102        UVM_PREDICT_WRITE:
1103          begin
1104            uvm_reg_field_cb_iter cbs = new(this);
1105
1106            if (rw.path == UVM_FRONTDOOR || rw.path == UVM_PREDICT)
1107              field_val = XpredictX(m_mirrored, field_val, rw.map);
1108
1109            m_written = 1;
1110
1111            for (uvm_reg_cbs cb = cbs.first(); cb != null; cb = cbs.next())
1112              cb.post_predict(this, m_mirrored, field_val,
1113                              UVM_PREDICT_WRITE, rw.path, rw.map);
1114
1115            field_val &= ('b1 << m_size)-1;
1116
1117          end
1118
1119        UVM_PREDICT_READ:
1120          begin
1121            uvm_reg_field_cb_iter cbs = new(this);
1122
1123            if (rw.path == UVM_FRONTDOOR || rw.path == UVM_PREDICT) begin
1124
1125              string acc = get_access(rw.map);
1126
1127              if (acc == "RC" ||
1128                  acc == "WRC" ||
1129                  acc == "W1SRC" ||
1130                  acc == "W0SRC")
1131                field_val = 0;   // (clear)
1132
1133              else if (acc == "RS" ||
1134                       acc == "WRS" ||
1135                       acc == "W1CRS" ||
1136                       acc == "W0CRS")
1137                field_val = ('b1 << m_size)-1; // all 1's (set)
1138
1139              else if (acc == "WO" ||
1140                       acc == "WOC" ||
1141                       acc == "WOS" ||
1142                       acc == "WO1")
1143                return;
1144            end
1145
1146            for (uvm_reg_cbs cb = cbs.first(); cb != null; cb = cbs.next())
1147              cb.post_predict(this, m_mirrored, field_val,
1148                              UVM_PREDICT_READ, rw.path, rw.map);
1149
```

```
1150              field_val &= ('b1 << m_size)-1;
1151
1152            end
1153
1154          UVM_PREDICT_DIRECT:
1155            begin
1156              if (m_parent.is_busy()) begin
1157                `uvm_warning("RegModel", {"Trying to predict value of field '",
1158                    get_name(),"' while register '",m_parent.get_full_name(),
1159                    "' is being accessed"})
1160                rw.status = UVM_NOT_OK;
1161              end
1162            end
1163        endcase
1164
1165        // update the mirror with predicted value
1166        m_mirrored = field_val;
1167        m_desired  = field_val;
1168        this.value = field_val;
1169
1170    endfunction: do_predict
```

可见，这里的与 do_write 的 2426 到 2446 行的几乎一样，不过这里更加清晰，对于读清的，置为 0，对于读置位的，置为全 1，对于只写的，则不做任何处理。1166 到 1168 行把 uvm_reg_field 中三个存储数据的变量的值更新。

18.6.4. uvm_reg::do_read(二)

接下来看 FRONTDOOR 方式进行的 do_read 任务：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_read

2405        // EXECUTE READ...
2406        case (rw.path)
2407
2408          // ...VIA USER BACKDOOR
2409          UVM_BACKDOOR: begin
                …
2459          end
2460
2461
2462          UVM_FRONTDOOR: begin
2463
2464            uvm_reg_map system_map = rw.local_map.get_root_map();
2465
```

```
2466            m_is_busy = 1;
2467
2468            // ...VIA USER FRONTDOOR
2469            if (map_info.frontdoor != null) begin
2470                uvm_reg_frontdoor fd = map_info.frontdoor;
2471                fd.rw_info = rw;
2472                if (fd.sequencer == null)
2473                   fd.sequencer = system_map.get_sequencer();
2474                fd.start(fd.sequencer, rw.parent);
2475            end
2476
2477            // ...VIA BUILT-IN FRONTDOOR
2478            else begin
2479                rw.local_map.do_read(rw);
2480            end
2481
2482            m_is_busy = 0;
2483
2484            if (system_map.get_auto_predict()) begin
2485                if (rw.status != UVM_NOT_OK) begin
2486                    sample(rw.value[0], -1, 1, rw.map);
2487                    m_parent.XsampleX(map_info.offset, 1, rw.map);
2488                end
2489
2490             do_predict(rw, UVM_PREDICT_READ);
2491            end
2492        end
2493
2494    endcase
2495
```

接下来看 FRONTDOOR 方式。整个 FRONTDOOR 方式的代码与 do_write 中 FRONTDOOR 方式完全一样，我们主要关注 2479 行调用的 uvm_reg_map 的 do_read 任务：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：do_read

1692 task uvm_reg_map::do_read(uvm_reg_item rw);
1693
1694   uvm_reg_map system_map = get_root_map();
1695   uvm_reg_adapter adapter = system_map.get_adapter();
1696   uvm_sequencer_base sequencer = system_map.get_sequencer();
1697
1698   if (rw.parent == null)
1699      rw.parent = new("default_parent_seq");
1700
1701   if (adapter == null) begin
1702      rw.set_sequencer(sequencer);
1703      rw.parent.start_item(rw,rw.prior);
```

```
1704        rw.parent.finish_item(rw);
1705        rw.end_event.wait_on();
1706      end
1707      else begin
1708        do_bus_read(rw, sequencer, adapter);
1709      end
1710
1711  endtask
```

它与 uvm_reg_map 的 do_write 函数几乎是完全一样的。1708 行调用 do_bus_read 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：do_bus_read

1843  task uvm_reg_map::do_bus_read (uvm_reg_item rw,
1844                                 uvm_sequencer_base sequencer,
1845                                 uvm_reg_adapter adapter);
1846
1847    uvm_reg_addr_t addrs[$];
1848    uvm_reg_map    system_map = get_root_map();
1849    int unsigned   bus_width  = get_n_bytes();
1850    uvm_reg_byte_en_t byte_en = -1;
1851    uvm_reg_map_info  map_info;
1852    int               size, n_bits;
1853    int               skip;
1854    int               lsb;
1855    int unsigned      curr_byte;
1856    int n_access_extra, n_access;
1857
1858    Xget_bus_infoX(rw, map_info, n_bits, lsb, skip);
1859    `UVM_DA_TO_QUEUE(addrs,map_info.addr)
1860    size = n_bits;
1861
1862    // if a memory, adjust addresses based on offset
1863    if (rw.element_kind == UVM_MEM)
1864      foreach (addrs[i])
1865        addrs[i] = addrs[i] + map_info.mem_range.stride * rw.offset;
1866
1867    foreach (rw.value[val_idx]) begin: foreach_value
1868
1869      /* calculate byte_enables */
1870      if (rw.element_kind == UVM_FIELD) begin
1871        int temp_be;
1872        int idx=0;
1873        n_access_extra = lsb%(bus_width*8);
1874        n_access = n_access_extra + n_bits;
1875        temp_be = n_access_extra;
1876        while(temp_be >= 8) begin
1877          byte_en[idx++] = 0;
1878          temp_be -= 8;
```

```
1879          end
1880          temp_be += n_bits;
1881          while(temp_be > 0) begin
1882             byte_en[idx++] = 1;
1883             temp_be -= 8;
1884          end
1885          byte_en &= (1<<idx)-1;
1886          for (int i=0; i<skip; i++)
1887            void'(addrs.pop_front());
1888       end
1889       rw.value[val_idx] = 0;
1890
1891       foreach (addrs[i]) begin
1892
1893          uvm_sequence_item bus_req;
1894          uvm_reg_bus_op rw_access;
1895          uvm_reg_data_logic_t data;
1896
1897
1898          `uvm_info(get_type_name(),
1899             $sformatf("Reading address 'h%0h via map \"%s\"...",
1900                       addrs[i], get_full_name()), UVM_FULL);
1901
1902          if (rw.element_kind == UVM_FIELD)
1903            for (int z=0;z<bus_width;z++)
1904              rw_access.byte_en[z] = byte_en[curr_byte+z];
1905
1906          rw_access.kind = rw.kind;
1907          rw_access.addr = addrs[i];
1908          rw_access.data = 'h0;
1909          rw_access.byte_en = byte_en;
1910          rw_access.n_bits = (n_bits > bus_width*8) ? bus_width*8 : n_bits;
1911
1912          adapter.m_set_item(rw);
1913          bus_req = adapter.reg2bus(rw_access);
1914          adapter.m_set_item(null);
1915          if (bus_req == null)
1916            `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] didnt return a bus transacti
on"});
1917
1918          bus_req.set_sequencer(sequencer);
1919          rw.parent.start_item(bus_req,rw.prior);
1920
1921          if (rw.parent != null && rw_access.addr == addrs[0]) begin
1922            rw.parent.pre_do(1);
1923            rw.parent.mid_do(rw);
1924          end
1925
1926          rw.parent.finish_item(bus_req);
1927          bus_req.end_event.wait_on();
1928
1929          if (adapter.provides_responses) begin
```

```
1930            uvm_sequence_item bus_rsp;
1931            uvm_access_e op;
1932            // TODO: need to test for right trans type, if not put back in q
1933            rw.parent.get_base_response(bus_rsp);
1934            adapter.bus2reg(bus_rsp,rw_access);
1935          end
1936          else begin
1937            adapter.bus2reg(bus_req,rw_access);
1938          end
1939
1940          data = rw_access.data & ((1<<bus_width*8)-1);
1941
1942          rw.status = rw_access.status;
1943
1944          if (rw.status == UVM_IS_OK && (^data) === 1'bx)
1945            rw.status = UVM_HAS_X;
1946
1947          `uvm_info(get_type_name(),
1948              $sformatf("Read 'h%0h at 'h%0h via map \"%s\": %s...", data,
1949                        addrs[i], get_full_name(), rw.status.name()), UVM_FULL);
1950
1951          if (rw.status == UVM_NOT_OK)
1952              break;
1953
1954          rw.value[val_idx] |= data << curr_byte*8;
1955
1956          if (rw.parent != null && rw_access.addr == addrs[addrs.size()])
1957            rw.parent.post_do(rw);
1958
1959          curr_byte += bus_width;
1960          n_bits -= bus_width * 8;
1961        end
1962
1963        if (rw.element_kind == UVM_FIELD)
1964            rw.value[val_idx] = (rw.value[val_idx] >> (n_access_extra)) & ((1<<size)-1);
1965      end
1966
1967  endtask: do_bus_read
```

这个函数与 do_bus_write 很相似，读者可以对照着来看。1858 和 1859 行得到要读取的地址。1870 到 1888 行牵扯到了 uvm_reg_field，暂且先跳过。1891 到 1961 行通过发送一个 sequence_item 给 driver 来读取每一个地址，这里与 do_bus_write 中的相关代码相似，唯一的区别是在 do_bus_write 操作中，在每发送一个 sequenct_item 之前，都要设定此 item 的数据。

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：do_read

2495
2496      value = rw.value[0]; // preserve
```

```
2497
2498      // POST-READ CBS - REG
2499      for (uvm_reg_cbs cb = cbs.first(); cb != null; cb = cbs.next())
2500          cb.post_read(rw);
2501      post_read(rw);
2502
2503      // POST-READ CBS - FIELDS
2504      foreach (m_fields[i]) begin
2505          uvm_reg_field_cb_iter cbs = new(m_fields[i]);
2506          uvm_reg_field f = m_fields[i];
2507
2508          rw.element = f;
2509          rw.element_kind = UVM_FIELD;
2510          rw.value[0] = (value >> f.get_lsb_pos()) & ((1<<f.get_n_bits())-1);
2511
2512          for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
2513              cb.post_read(rw);
2514          f.post_read(rw);
2515      end
2516
2517      rw.value[0] = value; // restore
2518      rw.element = this;
2519      rw.element_kind = UVM_REG;
2520
2521      // REPORT
2522      if (uvm_report_enabled(UVM_HIGH)) begin
2523        string path_s,value_s;
2524        if (rw.path == UVM_FRONTDOOR)
2525          path_s = (map_info.frontdoor != null) ? "user frontdoor" :
2526                                                  {"map ",rw.map.get_full_name()};
2527        else
2528          path_s = (get_backdoor() != null) ? "user backdoor" : "DPI backdoor";
2529
2530        value_s = $sformatf("=%0h",rw.value[0]);
2531
2532        `uvm_info("RegModel", {"Read  register via ",path_s,": ",
2533                               get_full_name(),value_s},UVM_HIGH)
2534      end
2535
2536      m_read_in_progress = 1'b0;
2537
2538  endtask: do_read
```

回到 uvm_reg 的 do_read 函数，2496 行得到读的数值，2499 到 2501 行调用此寄存器的 post_read 函数，2504 到 2515 行调用此寄存器所有的 uvm_reg_field 的 post_read 函数。由于在 post_read 函数中 rw 的某些变量被改变，2517 到 2519 恢复 rw 中这些被改变的变量。2522 到 2534 报告信息。2536 标志读操作结束。

回到 read 函数，2354 行从 do_read 函数返回后，2356 行把读操作过程的状态变化赋值给 status，2357 行则把读出的数值赋值给 value，而 value 是一个 output 类型的参数，所以最终用户通过 value 得到此次读操作的数值。

总结 uvm_reg 的 read 操作，它与 write 操作几乎是完全对偶的关系。只要明白了 write 操作，那么读操作也是轻而易举的。正是由于这种对偶的特性，所以接下来在分析 uvm_reg_field 的 read 和 write 操作，uvm_mem 的 read 和 write 操作时，将只分析 write 操作。

18.7. register model 的其它常用操作

18.7.1. uvm_reg 的 poke 和 peek 操作

本节分析 uvm_reg 的 poke 和 peek 操作。poke 操作对应 BACKDOOR 方式的 write 操作，peek 操作对应 BACKDOOR 方式的 read 操作，区别就是不会模仿 FRONTDOOR。

poke 函数的定义为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：poke

2704 task uvm_reg::poke(output uvm_status_e          status,
2705                    input  uvm_reg_data_t        value,
2706                    input  string                kind = "",
2707                    input  uvm_sequence_base     parent = null,
2708                    input  uvm_object            extension = null,
2709                    input  string                fname = "",
2710                    input  int                   lineno = 0);
2711
2712   uvm_reg_backdoor bkdr = get_backdoor();
2713   uvm_reg_item rw;
2714
2715   m_fname = fname;
2716   m_lineno = lineno;
2717
2718
2719   if (bkdr == null && !has_hdl_path(kind)) begin
2720     `uvm_error("RegModel",
2721       {"No backdoor access available to poke register '",get_full_name(),"'"})
2722     status = UVM_NOT_OK;
2723     return;
```

```
2724      end
2725
2726      if (!m_is_locked_by_field)
2727        XatomicX(1);
2728
2729      // create an abstract transaction for this operation
2730      rw = uvm_reg_item::type_id::create("reg_poke_item",,get_full_name());
2731      rw.element          = this;
2732      rw.path             = UVM_BACKDOOR;
2733      rw.element_kind  = UVM_REG;
2734      rw.kind             = UVM_WRITE;
2735      rw.bd_kind          = kind;
2736      rw.value[0]         = value & ((1 << m_n_bits)-1);
2737      rw.parent           = parent;
2738      rw.extension        = extension;
2739      rw.fname             = fname;
2740      rw.lineno           = lineno;
2741
2742      if (bkdr != null)
2743        bkdr.write(rw);
2744      else
2745        backdoor_write(rw);
2746
2747      status = rw.status;
2748
2749      `uvm_info("RegModel", $sformatf("Poked register \"%s\": 'h%h",
2750                                      get_full_name(), value),UVM_HIGH);
2751
2752      do_predict(rw, UVM_PREDICT_WRITE);
2753
2754      if (!m_is_locked_by_field)
2755        XatomicX(0);
2756  endtask: poke
```

2719 到 2724 行判断此寄存器是否能进行 BACKDOOR 操作，在用户没有自定义 backdoor，且没有设置此寄存器的 hdl 路径时，就认为此寄存器是不能进行 BACKDOOR 操作的，因此会给出错误提示，并直接返回。

2727 和 2755 行组成了原子操作。这里需要注意的是根据 m_is_locked_by_field 位来决定是否向 m_atomic 申请一个键值来保证此次 poke 操作的原子性，为什么要这么做？关于这一点，这里牵扯到 uvm_reg_field 的 poke 操作。因此在后面介绍 uvm_reg_field 的 poke 操作时会详细说明。

2730 行实例化一个 uvm_reg_item 型的变量，2731 到 2740 行把 poke 操作的相关信息写入此变量中。2742 行在用户自定义了 backdoor 的情况下，调用此 backdoor 的 write 函数，否则调用 backdoor_write 函数。关于 backdoor_write 函数，前面已经介绍过，这里不重复说明。

2752 行调用 do_predict 函数来进行预测，也即更新 register model 中寄存器的相关值。注意这里与使用 BACKDOOR 方式的 write 操作的区别。在 write 操作中，要

先使用 BACKDOOR 的 read 方式读出原来寄存器的数值，把读出来的数值，及将要写入的数值，通过调用 uvm_reg_field 的 XpredictX，得到一个新的值，这个值其实就是完全模拟了 FRONTDOOR 行为的一个值，之后再把这个值写入。而在 poke 操作中，则是直接写入，根本没有经过这些复杂的步骤。poke 中调用 do_predict 操作只是纯粹把 m_mirrored 值等更新为写入的数据，而 write 操作中调用 XpredictX 则是把要写入的值更新为一个比较合理的数据，但是 XpredictX 事实上是不会改变 m_mirrored 等数值的。或许，如果把 XpredictX 的名字改为 predict_frontdoor 的话会更加让人容易理解些。以一个例子来说明，假设某计数器是写 1 清的，除此之外，写任何东西进去都是没有效果的。我们如果通过 BACKDOOR 方式的 write 操作把 16'hFFFF 写进去的话，那么经过 XpredictX，写入的值就会变为 0，于是真正往里面写的数值是 0。在写完后，通过 do_predict 函数，更新 register model 中的 m_mirrored 数值，让其为 0。但是如果使用 poke 的方式，那么由于不经过 XpredictX 函数，所以写入的值没有变，真正写入的数值就是 16'hFFFF。之后经过 do_predict 函数，把 register model 中的 m_mirrored 的数值更新为 16'hFFFF。

因此，我们可以看的出来，poke 操作和 BACKDOOR 方式的 write 操作的区别：第一，poke 操作不会调用 pre_write，post_write 等函数；第二，poke 操作不会模拟 FRONTDOOR 操作中的寄存器行为。

peek 函数的定义为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：peek

2761 task uvm_reg::peek(output uvm_status_e          status,
2762                    output uvm_reg_data_t        value,
2763                    input  string                kind = "",
2764                    input  uvm_sequence_base     parent = null,
2765                    input  uvm_object            extension = null,
2766                    input  string                fname = "",
2767                    input  int                   lineno = 0);
2768
2769    uvm_reg_backdoor bkdr = get_backdoor();
2770    uvm_reg_item rw;
2771
2772    m_fname = fname;
2773    m_lineno = lineno;
2774
2775    if (bkdr == null && !has_hdl_path(kind)) begin
2776       `uvm_error("RegModel",
2777          $sformatf("No backdoor access available to peek register \"%s\"",
2778                    get_full_name()));
2779       status = UVM_NOT_OK;
2780       return;
2781    end
2782
```

```
2783      if(!m_is_locked_by_field)
2784          XatomicX(1);
2785
2786      // create an abstract transaction for this operation
2787      rw = uvm_reg_item::type_id::create("mem_peek_item",,get_full_name());
2788      rw.element          = this;
2789      rw.path             = UVM_BACKDOOR;
2790      rw.element_kind  = UVM_REG;
2791      rw.kind             = UVM_READ;
2792      rw.bd_kind          = kind;
2793      rw.parent           = parent;
2794      rw.extension        = extension;
2795      rw.fname            = fname;
2796      rw.lineno           = lineno;
2797
2798      if (bkdr != null)
2799         bkdr.read(rw);
2800      else
2801         backdoor_read(rw);
2802
2803      status = rw.status;
2804      value = rw.value[0];
2805
2806      `uvm_info("RegModel", $sformatf("Peeked register \"%s\": 'h%h",
2807                                      get_full_name(), value),UVM_HIGH);
2808
2809      do_predict(rw, UVM_PREDICT_READ);
2810
2811      if (!m_is_locked_by_field)
2812          XatomicX(0);
2813  endtask: peek
```

整个函数与 poke 函数几乎也是对偶的，唯一的区别就是 2801 行调用 backdoor_read 函数。因此这里也不详细说明。后面在介绍 uvm_reg_field 和 uvm_mem 的 poke 和 peek 操作时，将会只介绍 poke 操作。

18.7.2. uvm_reg_field 的 write 操作

UVM 的 register model 中最小的数据单位为 uvm_reg_field。这种数据单位不只体现在数据存储上，还体现在操作上。它支持 uvm_reg_field 级别的读写操作。本节介绍 uvm_reg_field 的 write 操作，至于读操作，由于它与写操作的对偶性，读者可以在看完写操作后自行研究。

write 函数的代码为：

文件：src/reg/uvm_reg_field.svh

```
类：uvm_reg_field
函数/任务：write

1386 task uvm_reg_field::write(output uvm_status_e           status,
1387                           input  uvm_reg_data_t         value,
1388                           input  uvm_path_e             path = UVM_DEFAULT_PATH,
1389                           input  uvm_reg_map            map = null,
1390                           input  uvm_sequence_base      parent = null,
1391                           input  int                    prior = -1,
1392                           input  uvm_object             extension = null,
1393                           input  string                 fname = "",
1394                           input  int                    lineno = 0);
1395
1396     uvm_reg_item rw;
1397     rw = uvm_reg_item::type_id::create("field_write_item",,get_full_name());
1398     rw.element          = this;
1399     rw.element_kind  = UVM_FIELD;
1400     rw.kind             = UVM_WRITE;
1401     rw.value[0]         = value;
1402     rw.path             = path;
1403     rw.map              = map;
1404     rw.parent           = parent;
1405     rw.prior            = prior;
1406     rw.extension        = extension;
1407     rw.fname            = fname;
1408     rw.lineno           = lineno;
1409
1410     do_write(rw);
1411
1412     status = rw.status;
1413
1414 endtask
```

函数的内容与 uvm_reg 的 write 类似，这里需要注意的是 1399 行把 element_kind 设置为 UVM_FIELD，表明这是由一个 uvm_reg_field 发起的写操作。1410 行调用了 do_write 任务：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_write

1419 task uvm_reg_field::do_write(uvm_reg_item rw);
1420
1421     uvm_reg_data_t     value_adjust;
1422     uvm_reg_map_info  map_info;
1423     uvm_reg_field      fields[$];
1424     bit  bad_side_effect;
1425
1426     m_parent.XatomicX(1);
1427     m_fname   = rw.fname;
1428     m_lineno  = rw.lineno;
```

```
1429
1430      if (!Xcheck_accessX(rw,map_info,"write()"))
1431        return;
1432
1433      m_write_in_progress = 1'b1;
1434
1435      if (rw.value[0] >> m_size) begin
1436          `uvm_warning("RegModel", {"uvm_reg_field::write(): Value greater than field '",
1437                                    get_full_name(),"'"})
1438          rw.value[0] &= ((1<<m_size)-1);
1439      end
1440
```

1430 行调用 Xcheck_accessX 函数，这个函数与 uvm_reg 的 Xcheck_accessX 函数几乎一模一样，这里不重复说明。

1433 行给 m_write_in_progress 赋值，标志写操作开始。1435 行检查要写入的数据是否超过了此 uvm_reg_field 所能存放的最大数值。如果超过了，给出警告，并且 1438 行把超过的部分直接忽略，只考虑有效的数值。

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_write

1441      // Get values to write to the other fields in register
1442      m_parent.get_fields(fields);
1443      foreach (fields[i]) begin
1444
1445          if (fields[i] == this) begin
1446              value_adjust |= rw.value[0] << m_lsb;
1447              continue;
1448          end
1449
1450          // It depends on what kind of bits they are made of...
1451          case (fields[i].get_access(rw.local_map))
1452            // These...
1453            "RO", "RC", "RS", "W1C", "W1S", "W1T", "W1SRC", "W1CRC":
1454              // Use all 0's
1455              value_adjust |= 0;
1456
1457            // These...
1458            "W0C", "W0S", "W0T", "W0SRC", "W0CRS":
1459              // Use all 1's
1460              value_adjust |= ((1<<fields[i].get_n_bits())-1) << fields[i].get_lsb_pos();
1461
1462            // These might have side effects! Bad!
1463            "WC", "WS", "WCRS", "WSRC", "WOC", "WOS":
1464                bad_side_effect = 1;
1465
1466            default:
1467                value_adjust |= fields[i].m_mirrored << fields[i].get_lsb_pos();
```

```
1468
1469            endcase
1470        end
1471
1472    `ifdef UVM_REG_NO_INDIVIDUAL_FIELD_ACCESS
1473        rw.element_kind = UVM_REG;
1474        rw.element = m_parent;
1475        rw.value = value_adjust;
1476        m_parent.do_write(rw)
1477    `else
1478
1479        if (!is_indv_accessible(rw.path,rw.local_map)) begin
              …
```

1442 行调用 uvm_reg 的 get_fields 函数，得到此 uvm_reg_field 所在的 uvm_reg 的所有的 uvm_reg_field。

1443 到 1470 行取得其它 uvm_reg_field 的值，并且把它们和要写入的值一起放入 value_adjust 中。这么做其实是为了接下来的按照整个寄存器来访问做准备的。因为并不是所有的 uvm_reg_field 都支持按照 field 访问。在不支持的情况下，那就退而求其次，对整个寄存器进行写操作。

1472 行检查是否定义了宏 UVM_REG_NO_INDIVIDUAL_FIELD_ACCESS，如果定义了，那么就把 rw 的 element_kind 设置为 UVM_REG，1476 行调用 uvm_reg 的 do_write 函数，从而完全转变成了一次 uvm_reg 的 write 操作。如果没有定义，那么 1479 行要检查此寄存器是否允许进行按 field 操作。这里用到了 is_indv_accessible 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：is_indv_accessible

1667 function bit uvm_reg_field::is_indv_accessible(uvm_path_e   path,
1668                                                uvm_reg_map  local_map);
1669     if (path == UVM_BACKDOOR) begin
1670         `uvm_warning("RegModel",
1671             {"Individual BACKDOOR field access not available for field '",
1672             get_full_name(), "'. Accessing complete register instead."})
1673         return 0;
1674     end
1675
1676     if (!m_individually_accessible) begin
1677             `uvm_warning("RegModel",
1678                 {"Individual field access not available for field '",
1679                 get_full_name(), "'. Accessing complete register instead."})
1680         return 0;
1681     end
1682
1683     // Cannot access individual fields if the container register
1684     // has a user-defined front-door
```

```
1685      if (m_parent.get_frontdoor(local_map) != null) begin
1686          `uvm_warning("RegModel",
1687                       {"Individual field access not available for field '",
1688                        get_name(), "' because register '", m_parent.get_full_name(), "' has a
 user-defined front-door. Accessing complete register ins       tead."})
1689          return 0;
1690      end
1691
1692      begin
1693        uvm_reg_map system_map = local_map.get_root_map();
1694        uvm_reg_adapter adapter = system_map.get_adapter();
1695        if (adapter.supports_byte_enable)
1696          return 1;
1697      end
1698
```

1669 到 1674 行检查是否是 BACKDOOR 方式，如果是，那么给出警告信息，并且直接返回。这里表明 UVM 中是不支持 BACKDOOR 方式的按照 field 的访问寄存器的，只支持 FRONTDOOR 形式的。

1676 行检查 m_individually_accessible 成员变量，这个成员变量的默认值为 0，它在调用此 uvm_reg_field 的 configure 函数时被赋值。因此，如果设置了此 uvm_reg_field 不支持按 field 访问，那么 1677 到 1679 行给出警告信息，并且直接返回。

1685 到 1690 行检查此 uvm_reg_field 所在的寄存器是否设置了自己的 frontdoor，如果设置了，那么也是不能进行按 field 访问的，只能按照整个寄存器的方式访问。

1693 行得到最顶层的 uvm_reg_map，1694 行得到此 uvm_reg_map 的 adapter。1695 行检查 adapter 的 supports_byte_enable，这一标志位是与协议相关的。如果这一标志位为 1，那么就直接返回 1，表明是可以按照 field 来进行访问。

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：is_indv_accessible

1699      begin
1700        int fld_idx = 0;
1701        int bus_width = local_map.get_n_bytes();
1702        uvm_reg_field fields[$];
1703        bit sole_field = 0;
1704
1705        m_parent.get_fields(fields);
1706
1707        if (fields.size() == 1) begin
1708            sole_field = 1;
1709        end
1710        else begin
1711            int prev_lsb,this_lsb,next_lsb;
1712            int prev_sz,this_sz,next_sz;
```

```
1713            int bus_sz = bus_width*8;
1714
1715            foreach (fields[i]) begin
1716                if (fields[i] == this) begin
1717                    fld_idx = i;
1718                    break;
1719                end
1720            end
1721
1722            this_lsb = fields[fld_idx].get_lsb_pos();
1723            this_sz  = fields[fld_idx].get_n_bits();
1724
1725            if (fld_idx>0) begin
1726               prev_lsb = fields[fld_idx-1].get_lsb_pos();
1727               prev_sz  = fields[fld_idx-1].get_n_bits();
1728            end
1729
1730            if (fld_idx < fields.size()-1) begin
1731               next_lsb = fields[fld_idx+1].get_lsb_pos();
1732               next_sz  = fields[fld_idx+1].get_n_bits();
1733            end
1734
1735            // if first field in register
1736            if (fld_idx == 0 &&
1737                ((next_lsb % bus_sz) == 0 ||
1738                 (next_lsb - this_sz) > (next_lsb % bus_sz)))
1739                return 1;
1740
1741            // if last field in register
1742            else if (fld_idx == (fields.size()-1) &&
1743                 ((this_lsb % bus_sz) == 0 ||
1744                  (this_lsb - (prev_lsb + prev_sz)) >= (this_lsb % bus_sz)))
1745                return 1;
1746
1747            // if somewhere in between
1748            else begin
1749                if ((this_lsb % bus_sz) == 0) begin
1750                    if ((next_lsb % bus_sz) == 0 ||
1751                         (next_lsb - (this_lsb + this_sz)) >= (next_lsb % bus_sz))
1752                        return 1;
1753                end
1754                else begin
1755                    if ( (next_lsb - (this_lsb + this_sz)) >= (next_lsb % bus_sz) &&
1756                         ((this_lsb - (prev_lsb + prev_sz)) >= (this_lsb % bus_sz)) )
1757                        return 1;
1758                end
1759            end
1760        end
1761    end
1762
1763    `uvm_warning("RegModel",
1764        {"Target bus does not support byte enabling, and the field '",
```

```
1765            get_full_name(),"' is not the only field within the entire bus width. ",
1766            "Individual field access will not be available. ",
1767            "Accessing complete register instead."})
1768
1769      return 0;
1770
1771  endfunction
```

1705 行得到此 uvm_reg_field 所在的 uvm_reg 的所有的 uvm_reg_field。1707 行如果发现这个寄存器只有这一个 field 的话，那么就会把 sole_field 置位，并且最终会跳转到 1769 行，返回 0，表示不支持按照 field 访问。因为对于只有一个 field 的寄存器来说，访问一个 field 其实就是访问整个寄存器。

在有多个 uvm_reg_field 的情况下，1715 到 1720 行得到这个 uvm_reg_field 在所有的 field 中的序号，1722 和 1723 行分别得到此 field 的 lsb 和位宽。

1725 行判断此 field 是不是最低位的 field，如果不是，说明还在更低位的，那么 1726 和 1727 行取得更低位 field 的 lsb 和位宽。

1730 行判断此 field 是不是最高位的 field，如果不是，说明还在更高位的，那么 1731 和 1732 行取得更高位 field 的 lsb 和位宽。

如果这是最低位的 field，那么 1736 到 1739 行判断此 field 是否能够按照 field 访问。以一个例子来说明，假设此寄存器有 64 位，系统总线为 16 位，此 field 的 lsb 为 0，位宽为 9，如果它的高位的 field 的 lsb 恰好是 16 或者 32 或者 48，即恰好是位宽的整数倍，那么是可以按 field 访问的；或者不是位宽的整数倍，如 lsb 为 23，那么也是可以的，但是假如是 13，那么是不可以的。这里的判断标准其实就是把整个 64 按照 16 的总线位宽分成了 4 块区域，如果这个 field 占据了某个区域，那么要想能够按照 field 访问，其它的 field 就不能占据这个区域。

1742 到 1759 行的判断条件与上面相似，不详细说明。

1769 行在不能满足按 field 访问的情况下，返回 0。

总结 is_indv_accessible 函数，其判断标准是只有 FRONTDOOR 才能进行按照 field 访问；如果用户定义了自己的 frontdoor，那么是不能按照 field 访问的；把寄存器位宽（如 64）按照总线宽度（如 16），分成几个区域，如果此 field 能够独占这个区域，那么是可以按 field 访问的，除此之外都不能按照 field 访问。把这种特性称为“局部独占总线特性”

回到 do_write 任务：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：do_write

1479      if (!is_indv_accessible(rw.path,rw.local_map)) begin
1480          rw.element_kind = UVM_REG;
```

```
1481            rw.element = m_parent;
1482            rw.value[0] = value_adjust;
1483            m_parent.do_write(rw);
1484
1485            if (bad_side_effect) begin
1486                `uvm_warning("RegModel", $sformatf("Writing field \"%s\" will cause unintended
 side effects in adjoining Write-to-Clear or Write-to-Set fi       elds in the same register", this.ge
t_full_name()));
1487            end
1488        end
1489        else begin
1490
1491            uvm_reg_map system_map = rw.local_map.get_root_map();
1492            uvm_reg_field_cb_iter cbs = new(this);
1493
1494            m_parent.Xset_busyX(1);
1495
1496            rw.status = UVM_IS_OK;
1497
1498            pre_write(rw);
1499            for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
1500                cb.pre_write(rw);
1501
1502            if (rw.status != UVM_IS_OK) begin
1503                m_write_in_progress = 1'b0;
1504                m_parent.Xset_busyX(0);
1505                m_parent.XatomicX(0);
1506
1507                return;
1508            end
1509
1510            rw.local_map.do_write(rw);
1511
1512            if (system_map.get_auto_predict())
1513                // ToDo: Call parent.XsampleX();
1514                do_predict(rw, UVM_PREDICT_WRITE);
1515
1516            post_write(rw);
1517            for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
1518                cb.post_write(rw);
1519
1520            m_parent.Xset_busyX(0);
1521
1522        end
1523
1524    `endif
1525
1526        m_write_in_progress = 1'b0;
1527        m_parent.XatomicX(0);
1528
1529    endtask: do_write
```

在不支持按照 field 访问的情况下，1480 行到 1483 行把本次 write 操作变更为一次 uvm_reg 的 write 操作。

在支持按照 field 访问的情况下，1491 行得到最顶层的 uvm_reg_map。1494 行调用所有 uvm_reg 的 Xset_busyX 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：Xset_busyX

2602 function void uvm_reg::Xset_busyX(bit busy);
2603     m_is_busy = busy;
2604 endfunction
```

这里只是简单的把 m_is_busy 标志位设置为 1，以防止其它程序在 write 操作时执行别的操作，从而出错。这里与 XatomicX 相似，都是为了保证对互斥资源的访问。

1496 行把 status 的值初始化为 UVM_IS_OK。1498 到 1500 行调用该 field 的所有的 pre_write 函数。

1502 行到 1508 行检测 status 的值，如果 pre_write 把 status 值改变了，那么就结束写操作，直接返回。

1510 行 uvm_reg_map 的 do_write 函数，前面介绍过这个函数会直接调用 uvm_reg_map 的 do_bus_write 函数，而此函数会首先调用 Xget_bus_infoX 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xget_bus_infoX

1629 function void uvm_reg_map::Xget_bus_infoX(uvm_reg_item rw,
1630                                            output uvm_reg_map_info map_info,
1631                                            output int size,
1632                                            output int lsb,
1633                                            output int addr_skip);
1634
1635   if (rw.element_kind == UVM_MEM) begin
         …
1642   end
1643   else if (rw.element_kind == UVM_REG) begin
         …
1650   end
1651   else if (rw.element_kind == UVM_FIELD) begin
1652     uvm_reg_field field;
1653     if(rw.element == null || !$cast(field,rw.element))
1654       `uvm_fatal("REG/CAST", {"uvm_reg_item 'element_kind' is UVM_FIELD, ",
1655                  "but 'element' does not point to a field: ",rw.get_name()})
1656     map_info = get_reg_map_info(field.get_parent());
1657     size = field.get_n_bits();
1658     lsb = field.get_lsb_pos();
```

```
1659        addr_skip  =  lsb/(get_n_bytes()*8);
1660      end
1661  endfunction
```

与本节相关的是从 1651 行开始。把 map_info 赋值为此 uvm_reg_field 所在的 uvm_reg 在此 uvm_reg_map 的 m_regs_info 中对应的记录值。1657 行把 size 赋值为此 uvm_reg_field 的位宽，1658 行把 lsb 赋值为这个 uvm_reg_field 的 lsb。1659 行比较有意思，指的是要跳过的地址的数量。以例子介绍，假设总线是 16 位，此 uvm_reg_field 所在的 uvm_reg 为 16 位，此 uvm_reg_field 的 lsb 为 0，那么这里的 addr_skip 将会是 0，即这个寄存器的地址值就是要访问的 uvm_reg_field 的值。假如此 uvm_reg_field 所在的 uvm_reg 为 64 位，它有 4 个 uvm_reg_field，每个占据 16 位，现在要访问的是第三个 uvm_reg_field，即其 lsb 为 32，那么 addr_skip 的值为 2，即在访问时跳过前两个地址。因为前两个地址是前面两个 uvm_reg_field 所对应的地址。

回到 do_bus_write 函数，与本节相关的是 1744 行。这段语句比较难懂，以一个例子来说明。假如总线宽度为 16，要访问的寄存器有 64 位，其中我们关心的 field 的 lsb 为 19，位宽为 18，也就意味着如果把 64 位分成 4 个域的话，那么此 field 是占据了中间两个域。

1747 行 n_access_extra 的值为 3，n_access 为 21。这里 3 表示是此寄存器所占据的第二个物理地址所对应总线的是第 4 位。n_access 表示这个 field 的终止位，由于大于 16，所以也就意味着要在第三个物理地址的第 5 位结束。1750 行把 value 的值进行调整，经过调整之后才能真正的写入。1751 到 1760 行用于对 byte_en 赋值，经过这几行后，byte_en 的值变为了'b00000111，这里 3 个 1 比较难以理解，先看 1761 到 1762 行，这里把 addrs 中的地址给丢弃了。在我们的这个例子中，skip 的值为 1，因此相当是把第一个值丢弃了，因为我们要写的 field 是从第二个地址开始的，所以第一个地址自然是没用了。从第二个地址开始算起，一直到此寄存器的最高位，一共还有 6 个 byte，这 6 个 byte 中最低的 3 个 byte 是要写入的数值。所以上面出现的 3 个 1 就是对应这里要写入的 3 个 byte。这里计算出 byte_en 有什么用处？这个就要看具体的协议实现了。如果所用的协议根本不支持按照 byte 操作，那么这个变量也就是没有作用的。做为一个成熟的验证方法学，UVM 自然要考虑这种按 byte 操作的情况。所以要在内部加入对于按 byte 访问的支持。

do_bus_write 后面的代码与与 uvm_reg 发起 write 操作时一样，不多做说明，不过这里会产生一个冗余操作。在上面的例子中，只需要访问第二个地址和第三个地址就可以了，但是实际上 UVM 会对接下来的第四个地址也进行写操作，写入的数值为 0。这会造成错误的情况，这是 UVM 的一个 bug，已经在修复 3641 的 bug（http://www.eda.org/svdb/view.php? id=3641）时修复了。

回到 uvm_reg_field 的 do_write 函数。1510 行从 uvm_reg_map 的 do_write 返回后，1512 行检查 register model 是否启用了 auto predict 功能，如果启用了，那么调用 do_predict 函数。这个函数在前面已经分析过了。1516 到 1518 行调用此 uvm_reg_field 的所有的 post_write 函数。

1520 行与 1494 行相呼应。1526 与 1527 行进行后续处理，结束原子操作。

总结一下，uvm_reg_field 的 write 操作只能以 FRONTDOOR 方式进行，BACKDOOR 方式是不支持的。这个结论同样也适用于 uvm_reg_field 的 read 操作。

18.7.3. uvm_reg_field 的 poke 操作

本节介绍 uvm_reg_field 的 poke 操作，其函数为：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：poke

1776 task uvm_reg_field::poke(output uvm_status_e          status,
1777                          input  uvm_reg_data_t        value,
1778                          input  string                kind = "",
1779                          input  uvm_sequence_base     parent = null,
1780                          input  uvm_object            extension = null,
1781                          input  string                fname = "",
1782                          input  int                   lineno = 0);
1783    uvm_reg_data_t    tmp;
1784
1785    m_fname = fname;
1786    m_lineno = lineno;
1787
1788    if (value >> m_size) begin
1789       `uvm_warning("RegModel",
1790          {"uvm_reg_field::poke(): Value exceeds size of field '",
1791           get_name(),"'"})
1792       value &= value & ((1<<m_size)-1);
1793    end
1794
1795
1796    m_parent.XatomicX(1);
1797    m_parent.m_is_locked_by_field = 1'b1;
1798
1799    tmp = 0;
1800
1801    // What is the current values of the other fields???
1802    m_parent.peek(status, tmp, kind, parent, extension, fname, lineno);
1803
1804    if (status == UVM_NOT_OK) begin
1805       `uvm_error("RegModel", {"uvm_reg_field::poke(): Peek of register '",
1806          m_parent.get_full_name(),"' returned status ",status.name()})
1807       m_parent.XatomicX(0);
1808       m_parent.m_is_locked_by_field = 1'b0;
1809       return;
```

```
1810      end
1811
1812      // Force the value for this field then poke the resulting value
1813      tmp &= ~(((1<<m_size)-1) << m_lsb);
1814      tmp |= value << m_lsb;
1815      m_parent.poke(status, tmp, kind, parent, extension, fname, lineno);
1816
1817      m_parent.XatomicX(0);
1818      m_parent.m_is_locked_by_field = 1'b0;
1819  endtask: poke
```

1788 到 1793 行检查要写入的值是否超过了此 uvm_reg_field 所能容纳的最大值，如果超过了给出警告并且把超出部分去除。

1796 行调用所在 uvm_reg 的 XatomicX 函数，开始一个原子操作。1797 行把 uvm_reg 的 m_is_locked_by_field 置位。当时在分析 uvm_reg 的 poke 函数时把这个标志位略过了。现在来具体的分析这个标志位。

1815 行会调用 uvm_reg 的 poke 函数，此时调用的时候，m_is_locked_by_field 位已经是 1 了，按照 uvm_reg 的 poke 函数的相关代码，此时 uvm_reg 就不会调用 XatomicX，而是会直接进行接下来的操作。由于 uvm_reg_field 的 poke 函数所在的进程已经取得了此 uvm_reg 的访问权，所以接下来调用 uvm_reg 的 poke 时自然不必再调用 XatomicX 申请访问权。在理解这个过程的时候，要注意访问个是跟进程相关的，而不是跟函数调用相关的。函数调用函数都是属于同一个进程。同一进程之内的代码是顺序执行的。因此不会有同一进程内的两个函数同时访问某个寄存器的情况。

那么这种情况会不会出错呢？如果用户不小心把 m_is_locked_by_field 赋值为 1，那么接下来 uvm_reg 的 poke 函数在进行操作之前将永远不会申请访问权，这很有可能造成错误结果。事实上，由于 XatomicX 函数内部已经设置了被同一进程重复申请调用的机制（可以参看前面关于这个函数的介绍章节），所以 uvm_reg 的 poke 函数的 2726 和 2754 行完全可以省略。

回到 uvm_reg_field 的 poke 函数来。1802 行调用了 uvm_reg 的 peek 函数，这个函数前面分析过。最终 tmp 中会返回此 uvm_reg 的值。1813 行和 1814 行通过移位运算，把 tmp 中此 uvm_reg_field 所在的位的值变为要写入的值。1815 行通过调用 uvm_reg 的 poke 函数把这个值写入 uvm_reg 中。

总结一下，所谓的 uvm_reg_field 的 poke 操作，其实质还是要调用所在 uvm_reg 的 poke 函数来完成。并且在 poke 之前，要先把其它 field 的值读出来，把这些值和此 field 要写入的值组合起来做为参数传递给 uvm_reg 的 poke 函数。需要注意的一点是，这里的按照 field 的 poke 操作不受上节所说的“局部独占总线特性”，任何寄存器的任何 field 都可以使用 poke 操作，当然了，前提必须是定义好了 hdl 路径。

uvm_reg_field 的 peek 函数也是通过调用 uvm_reg 的 peek 函数来完成。它也不受“局部独占总线特性”的限制，只要定义好了 hdl 路径，任何寄存器的任何 field

都可以使用 peek 函数。

18.7.4. uvm_mem 的 write 与 burst_write 操作

本节介绍 uvm_mem 的 write 与 burst_wrirte 操作。write 任务为：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：write

1434 task uvm_mem::write(output uvm_status_e          status,
1435                     input   uvm_reg_addr_t       offset,
1436                     input   uvm_reg_data_t       value,
1437                     input   uvm_path_e           path = UVM_DEFAULT_PATH,
1438                     input   uvm_reg_map          map = null,
1439                     input   uvm_sequence_base    parent = null,
1440                     input   int                  prior = -1,
1441                     input   uvm_object           extension = null,
1442                     input   string               fname = "",
1443                     input   int                  lineno = 0);
1444
1445    // create an abstract transaction for this operation
1446    uvm_reg_item rw = uvm_reg_item::type_id::create("mem_write",,get_full_name());
1447    rw.element      = this;
1448    rw.element_kind = UVM_MEM;
1449    rw.kind         = UVM_WRITE;
1450    rw.offset       = offset;
1451    rw.value[0]     = value;
1452    rw.path         = path;
1453    rw.map          = map;
1454    rw.parent       = parent;
1455    rw.prior        = prior;
1456    rw.extension    = extension;
1457    rw.fname        = fname;
1458    rw.lineno       = lineno;
1459
1460    do_write(rw);
1461
1462    status = rw.status;
1463
1464 endtask: write
```

1446 到 1458 行把写操作的相关信息放入 rw 中。需要注意的是 1448 行把 element_kind 设置为 UVM_MEM，表示这是一个 uvm_mem 发起的操作。1451 行只是把一个数值放入了 rw.value 中。由于这只是一个普通的 write 操作，所以只写一个数值。但是在 burst_write 操作中，有多个值赋值给 rw.value：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：burst_write

1505 task uvm_mem::burst_write(output uvm_status_e           status,
1506                           input  uvm_reg_addr_t         offset,
1507                           input  uvm_reg_data_t         value[],
1508                           input  uvm_path_e             path = UVM_DEFAULT_PATH,
1509                           input  uvm_reg_map            map = null,
1510                           input  uvm_sequence_base      parent = null,
1511                           input  int                    prior = -1,
1512                           input  uvm_object             extension = null,
1513                           input  string                 fname = "",
1514                           input  int                    lineno = 0);
1515
1516     uvm_reg_item rw;
1517     rw = uvm_reg_item::type_id::create("mem_burst_write",,get_full_name());
1518     rw.element          = this;
1519     rw.element_kind     = UVM_MEM;
1520     rw.kind             = UVM_BURST_WRITE;
1521     rw.offset           = offset;
1522     rw.value            = value;
1523     rw.path             = path;
1524     rw.map              = map;
1525     rw.parent           = parent;
1526     rw.prior            = prior;
1527     rw.extension        = extension;
1528     rw.fname            = fname;
1529     rw.lineno           = lineno;
1530
1531     do_write(rw);
1532
1533     status = rw.status;
1534
1535 endtask: burst_write
```

输入的 value 是一个动态数组，里面存放了多个数值，所以 1521 行的 rw.value 中也会有多个数值。另外需要注意的是 1520 行把 kind 设置为 UVM_BURST_WRITE，而在 write 的 1449 行则把这个值设置为 UVM_WRITE，从而区分了两个不同的操作。

无论是 write 还是 burst_write，都会调用 do_write 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
函数/任务：do_write

1575 task uvm_mem::do_write(uvm_reg_item rw);
1576
1577     uvm_mem_cb_iter  cbs = new(this);
1578     uvm_reg_map_info map_info;
1579
```

```
1580      m_fname   = rw.fname;
1581      m_lineno  = rw.lineno;
1582
1583      if (!Xcheck_accessX(rw, map_info, "burst_write()"))
1584        return;
1585
1586      m_write_in_progress = 1'b1;
1587
1588      rw.status = UVM_IS_OK;
1589
1590      // PRE-WRITE CBS
1591      pre_write(rw);
1592      for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
1593          cb.pre_write(rw);
1594
1595      if (rw.status != UVM_IS_OK) begin
1596          m_write_in_progress = 1'b0;
1597
1598          return;
1599      end
1600
1601      rw.status = UVM_NOT_OK;
1602
1603      // FRONTDOOR
1604      if (rw.path == UVM_FRONTDOOR) begin
1605
1606          uvm_reg_map system_map = rw.local_map.get_root_map();
1607
1608          if (map_info.frontdoor != null) begin
1609              uvm_reg_frontdoor fd = map_info.frontdoor;
1610              fd.rw_info = rw;
1611              if (fd.sequencer == null)
1612                fd.sequencer = system_map.get_sequencer();
1613              fd.start(fd.sequencer, rw.parent);
1614          end
1615          else begin
1616              rw.local_map.do_write(rw);
1617          end
1618
1619          if (rw.status != UVM_NOT_OK)
1620              for (int idx = rw.offset;
1621                    idx <= rw.offset + rw.value.size();
1622                    idx++) begin
1623                  XsampleX(map_info.mem_range.stride * idx, 0, rw.map);
1624                  m_parent.XsampleX(map_info.offset +
1625                                    (map_info.mem_range.stride * idx),
1626                                     0, rw.map);
1627              end
1628      end
1629
1630      // BACKDOOR
1631      else begin
```

```
1632          // Mimick front door access, i.e. do not write read-only memories
1633          if (get_access(rw.map) == "RW") begin
1634              uvm_reg_backdoor bkdr = get_backdoor();
1635              if (bkdr != null)
1636                  bkdr.write(rw);
1637              else
1638                  backdoor_write(rw);
1639          end
1640          else
1641              rw.status = UVM_IS_OK;
1642      end
1643
1644      // POST-WRITE CBS
1645      post_write(rw);
1646      for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
1647          cb.post_write(rw);
1648
1649      // REPORT
1650      if (uvm_report_enabled(UVM_HIGH)) begin
1651        string path_s,value_s,pre_s,range_s;
1652        if (rw.path == UVM_FRONTDOOR)
1653          path_s = (map_info.frontdoor != null) ? "user frontdoor" :
1654                                                  {"map ",rw.map.get_full_name()};
1655        else
1656          path_s = (get_backdoor() != null) ? "user backdoor" : "DPI backdoor";
1657
1658        if (rw.value.size() > 1 && uvm_report_enabled(UVM_HIGH)) begin
1659          value_s = "='{";
1660          pre_s = "Burst ";
1661          foreach (rw.value[i])
1662            value_s = {value_s,$sformatf("%0h,",rw.value[i])};
1663          value_s[value_s.len()-1]="}";
1664          range_s = $sformatf("[%0d:%0d]",rw.offset,rw.offset+rw.value.size());
1665        end
1666        else begin
1667          value_s = $sformatf("=%0h",rw.value[0]);
1668          range_s = $sformatf("[%0d]",rw.offset);
1669        end
1670
1671        `uvm_info("RegModel", {pre_s,"Wrote memory via ",path_s,": ",
1672                               get_full_name(),range_s,value_s},UVM_HIGH)
1673      end
1674
1675      m_write_in_progress = 1'b0;
1676
1677  endtask: do_write
```

1580 和 1581 行给两个变量赋值，这两个变量用于后面打印信息。1583 行调用 Xcheck_accessX 函数：

```
文件：src/reg/uvm_mem.svh
类：uvm_mem
```

函数/任务：Xcheck_accessX

```
1784 function bit uvm_mem::Xcheck_accessX(input uvm_reg_item rw,
1785                                        output uvm_reg_map_info map_info,
1786                                        input string caller);
1787
1788    if (rw.offset >= m_size) begin
1789       `uvm_error(get_type_name(),
1790          $sformatf("Offset 'h%0h exceeds size of memory, 'h%0h",
1791             rw.offset, m_size))
1792       rw.status = UVM_NOT_OK;
1793       return 0;
1794    end
1795
1796    if (rw.path == UVM_DEFAULT_PATH)
1797      rw.path = m_parent.get_default_path();
1798
1799    if (rw.path == UVM_BACKDOOR) begin
1800       if (get_backdoor() == null && !has_hdl_path()) begin
1801          `uvm_warning("RegModel",
1802             {"No backdoor access available for memory '",get_full_name(),
1803             "' . Using frontdoor instead."})
1804          rw.path = UVM_FRONTDOOR;
1805       end
1806       else
1807          rw.map = uvm_reg_map::backdoor();
1808    end
1809
1810    if (rw.path != UVM_BACKDOOR) begin
1811
1812      rw.local_map = get_local_map(rw.map,caller);
1813
1814      if (rw.local_map == null) begin
1815         `uvm_error(get_type_name(),
1816            {"No transactor available to physically access memory from map '",
1817             rw.map.get_full_name(),"'"})
1818         rw.status = UVM_NOT_OK;
1819         return 0;
1820      end
1821
1822      map_info = rw.local_map.get_mem_map_info(this);
1823
1824      if (map_info.frontdoor == null) begin
1825
1826         if (map_info.unmapped) begin
1827            `uvm_error("RegModel", {"Memory '",get_full_name(),
1828                       "' unmapped in map '", rw.map.get_full_name(),
1829                       "' and does not have a user-defined frontdoor"})
1830            rw.status = UVM_NOT_OK;
1831            return 0;
1832         end
1833
```

```
1834              if ((rw.value.size() > 1)) begin
1835                  if (get_n_bits() > rw.local_map.get_n_bytes()*8) begin
1836                      `uvm_error("RegModel",
1837                              $sformatf("Cannot burst a %0d-bit memory through a narrower data
 path (%0d bytes)",
1838                              get_n_bits(), rw.local_map.get_n_bytes()*8));
1839                      rw.status = UVM_NOT_OK;
1840                      return 0;
1841                  end
1842                  if (rw.offset + rw.value.size() > m_size) begin
1843                      `uvm_error("RegModel",
1844                           $sformatf("Burst of size 'd%0d starting at offset 'd%0d exceeds size of
 memory, 'd%0d",
1845                                  rw.value.size(), rw.offset, m_size))
1846                      return 0;
1847                  end
1848              end
1849          end
1850
1851          if (rw.map == null)
1852              rw.map = rw.local_map;
1853      end
1854
1855      return 1;
1856  endfunction
```

从总体上来看，这个函数与 uvm_reg 和 uvm_field 的 Xcheck_accessX 函数是很像的。只讲述有区别的地方。1788 行检查要访问的基地址是否超出了此 memory 的范围，如果超出了给出出错提示，并且直接返回。

1834 到 1848 行也是这个函数独有的，这几行主要是用于 burst 操作时进行检查。其中 1835 到 1841 行用于检查进行 burst 操作的 memory 的位宽是否超过总线的位宽。这是一个比较奇怪的检查。假如系统数据总线位宽是 16 位，如果一块 1024*32 的 memory 进行 burst 操作是不允许的！

1842 行到 1847 行则是用于检查进行的 burst 操作是否超出了此 memory 的范围，如果超出了，直接返回。即假设 memory 的大小为 1024，从 1020 地址开始写，要写入 8 个数据，这明显超出了 1024 的范围。

函数的其它部分与 uvm_reg 和 Xcheck_accessX 类似，不多做介绍。

回到 do_write 函数，1586 行置位 m_write_in_progress，开始写操作。1588 行初始化 status 变量。1591 到 1599 行调用 pre_write 函数，并且查看调用之后的 status 是否改变。如果变为 UVM_NOT_OK，则直接返回，不再进行写操作。

从 1604 行开始按照操作方式分别调用不同的函数进行写操作。如果以 FRONTDOOR 方式进行，那么会调用 uvm_reg_map 的 do_write 函数 而 uvm_reg_map 的 do_write 又会最终调用 uvm_reg_map 的 do_bus_write 任务（可以参看前面章节贴出的代码）。这个函数我们前面也仔细分析过，唯一有点区别的是：

一，调用的 Xget_bus_infoX 会针对 uvm_mem 做不同的处理：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：Xget_bus_infoX

1629 function void uvm_reg_map::Xget_bus_infoX(uvm_reg_item rw,
1630                                           output uvm_reg_map_info map_info,
1631                                           output int size,
1632                                           output int lsb,
1633                                           output int addr_skip);
1634
1635    if (rw.element_kind == UVM_MEM) begin
1636       uvm_mem mem;
1637       if(rw.element == null || !$cast(mem,rw.element))
1638          `uvm_fatal("REG/CAST", {"uvm_reg_item 'element_kind' is UVM_MEM, ",
1639                     "but 'element' does not point to a memory: ",rw.get_name()})
1640       map_info = get_mem_map_info(mem);
1641       size = mem.get_n_bits();
1642    end
1643    else if (rw.element_kind == UVM_REG) begin
          …
1650    end
1651    else if (rw.element_kind == UVM_FIELD) begin
          …
1660    end
1661 endfunction
```

1640 行调用 get_mem_map_info 函数：

```
文件：src/reg/uvm_reg_map.svh
类：uvm_reg_map
函数/任务：get_mem_map_info

1206 function uvm_reg_map_info uvm_reg_map::get_mem_map_info(uvm_mem mem, bit error=1);
1207    if (!m_mems_info.exists(mem)) begin
1208       if (error)
1209          `uvm_error("REG_NO_MAP",{"Memory '",mem.get_name(),"' not in map '",get_name
(),"'"})
1210       return null;
1211    end
1212    return m_mems_info[mem];
1213 endfunction
```

函数与 get_reg_map_info 一样，比较简单，从 m_mems_info 中取出对应发起写操作的 memory 的记录并返回。

1641 行把 size 设置为这块 memory 的位宽，注意这里不是 memory 的大小，也不是 burst 操作的数量，它只是纯粹的位宽。

二，do_bus_write 的 1735 行到 1737 行会把返回的地址值做一修正，加上偏移量。

因为由 Xget_bug_infoX 中取得的 map_info 中的 addrs 存放的是这块 memory 的初始地址，地址的数量与 memory 的位宽和系统位宽有关。假如系统位宽为 16，而 memory 位宽为 64，这里会存放 4 个地址。由于只是初始地址，所以要加上要写的单元的偏移量。

三，1739 行，由于牵扯到 burst 操作，这里可能会有多个数值。对于每一个数值的写操作，在分析 uvm_reg 的写操作时已经介绍过了，那么对于多个数值的写操作，地址是怎么变化的呢？这里的关键在于 1833 到 1834 行，每写一个数值就把所有的地址值加上 stride 值，这个值在分析 Xinit_address_mapX 时已经介绍过了，里面存放的是两个连续的单元的物理起始地址之差。系统总线为 16 位，假设 memory 为 16bit，那么 stride 值为 1，即一个单元对应一个地址；假如 memory 为 64，那么 stride 值为 4，即一个单元对应 4 个地址。

回到 uvm_mem 的 do_write 函数，如果采用 BACKDOOR 方式，那么在使用系统的 backdoor 情况下，1638 行调用 backdoor_write 函数，这个函数与 uvm_reg 的 backdoor_write 一模一样，因此不多做介绍。需要注意的是，1633 行会查看整块 memory 的属性，只有 RW 属性的才会进行写操作，否则根本不进行任何操作。

无论是以 BACKDOOR 还是以 FRONTDOOR 方式，在写操作完成后，1645 到 1647 行调用 post_write 函数。1650 行到 1673 行打印本次写操作的信息。1675 行把 m_write_in_progress 赋值为 0，标志写操作结束。

总结下，memory 的 write 和 burst_write 操作比较简单，它们不必和 uvm_reg 的 write 操作那样需要进行预测。

uvm_mem 的 read，burst_read，poke，peek 操作跟前面的相似，差别不大，因此不在重复说明。

18.7.5. uvm_reg 的 set 和 get 操作

经常使用 set 操作来设置 desired 值，使用 get 操作来得到 desired 值。set 函数的代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：set

1906 function void uvm_reg::set(uvm_reg_data_t   value,
1907                            string           fname = "",
1908                            int              lineno = 0);
```

```
1909      // Split the value into the individual fields
1910      m_fname = fname;
1911      m_lineno = lineno;
1912
1913      foreach (m_fields[i])
1914          m_fields[i].set((value >> m_fields[i].get_lsb_pos()) &
1915                                  ((1 << m_fields[i].get_n_bits()) - 1));
1916  endfunction: set
```

函数比较简单，调用每一个 uvm_reg_field 的 set 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：set

1213  function void uvm_reg_field::set(uvm_reg_data_t   value,
1214                                   string             fname = "",
1215                                   int                lineno = 0);
1216      uvm_reg_data_t mask = ('b1 << m_size)-1;
1217
1218      m_fname = fname;
1219      m_lineno = lineno;
1220      if (value >> m_size) begin
1221          `uvm_warning("RegModel",
1222              $sformatf("Specified value (0x%h) greater than field \"%s\" size (%0d bits)",
1223                  value, get_name(), m_size));
1224          value &= mask;
1225      end
1226
1227      case (m_access)
1228          "RO":       m_desired = m_desired;
1229          "RW":       m_desired = value;
1230          "RC":       m_desired = m_desired;
1231          "RS":       m_desired = m_desired;
1232          "WC":       m_desired = '0;
1233          "WS":       m_desired = mask;
1234          "WRC":      m_desired = value;
1235          "WRS":      m_desired = value;
1236          "WSRC":     m_desired = mask;
1237          "WCRS":     m_desired = '0;
1238          "W1C":      m_desired = m_desired & (~value);
1239          "W1S":      m_desired = m_desired | value;
1240          "W1T":      m_desired = m_desired ^ value;
1241          "W0C":      m_desired = m_desired & value;
1242          "W0S":      m_desired = m_desired | (~value & mask);
1243          "W0T":      m_desired = m_desired ^ (~value & mask);
1244          "W1SRC":  m_desired = m_desired | value;
1245          "W1CRS":  m_desired = m_desired & (~value);
1246          "W0SRC":  m_desired = m_desired | (~value & mask);
1247          "W0CRS":  m_desired = m_desired & value;
1248          "WO":       m_desired = value;
1249          "WOC":      m_desired = '0;
```

```
1250          "WOS":     m_desired = mask;
1251          "W1":      m_desired = (m_written) ? m_desired : value;
1252          "WO1":     m_desired = (m_written) ? m_desired : value;
1253          default: m_desired = value;
1254      endcase
1255      this.value = m_desired;
1256  endfunction: set
```

1216 行取得 mask 值，1220 行检查要 set 的值是否超出了此 uvm_reg_field 所能存放的最大值，超出了给出错误提示并把超出部分截掉。

1227 行到 1254 行对 m_desired 的值进行设置，这里并不是直接把输入的值赋值给 m_desired，而是要根据此 uvm_reg_field 的存取策略来进行一定的修正，例如对于 RO 的寄存器，1228 行根本不管输入的数值是多少，直接采用原来的值。

1255 行把 value 值更新为设置的值。因此，set 函数会改变 value 和 m_desired，而不会改变 m_mirrored 值。

uvm_reg 的 get 函数为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：get

1971  function uvm_reg_data_t   uvm_reg::get(string   fname = "",
1972                                          int        lineno = 0);
1973      // Concatenate the value of the individual fields
1974      // to form the register value
1975      m_fname = fname;
1976      m_lineno = lineno;
1977
1978      get = 0;
1979
1980      foreach (m_fields[i])
1981          get |= m_fields[i].get() << m_fields[i].get_lsb_pos();
1982  endfunction: get
```

函数依次调用每一个 uvm_reg_field 的 get 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：get

1261  function uvm_reg_data_t   uvm_reg_field::get(string   fname = "",
1262                                               int        lineno = 0);
1263      m_fname = fname;
1264      m_lineno = lineno;
1265      get = m_desired;
1266  endfunction: get
```

这里直接把 m_desired 的值赋值返回。

因此，无论是 set 还是 get 操作，它们均不会直接的读取 DUT，它们甚至也不会改变 register model 中与 DUT 最接近的 m_mirrored 值。因此，set 与 get 操作更像是一种一厢情愿的操作。

18.7.6. uvm_reg 的 update 操作

上节说到，uvm_reg 的 set 操作是一厢情愿的操作，要让这种操作成真，则必须调用 update 函数：

文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：update

```
2051 task uvm_reg::update(output uvm_status_e          status,
2052                      input  uvm_path_e            path = UVM_DEFAULT_PATH,
2053                      input  uvm_reg_map           map = null,
2054                      input  uvm_sequence_base parent = null,
2055                      input  int                   prior = -1,
2056                      input  uvm_object            extension = null,
2057                      input  string                fname = "",
2058                      input  int                   lineno = 0);
2059    uvm_reg_data_t upd;
2060
2061    status = UVM_IS_OK;
2062
2063    if (!needs_update()) return;
2064
2065    if (m_update_in_progress) begin
2066       @(negedge m_update_in_progress);
2067       return;
2068    end
2069
2070    m_update_in_progress = 1;
2071
2072    // Concatenate the write-to-update values from each field
2073    // Fields are stored in LSB or MSB order
2074    upd = 0;
2075    foreach (m_fields[i])
2076       upd |= m_fields[i].XupdateX() << m_fields[i].get_lsb_pos();
2077
2078    write(status, upd, path, map, parent, prior, extension, fname, lineno);
2079
2080    m_update_in_progress = 0;
2081
2082 endtask: update
```

2063 行调用 needs_update 函数：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：needs_update

2039  function bit uvm_reg::needs_update();
2040      needs_update = 0;
2041      foreach (m_fields[i]) begin
2042          if (m_fields[i].needs_update()) begin
2043              return 1;
2044          end
2045      end
2046  endfunction: needs_update
```

这里调用每一个 uvm_reg_field 的 needs_update 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：needs_update

1322  function bit uvm_reg_field::needs_update();
1323      needs_update = (m_mirrored != m_desired);
1324  endfunction: needs_update
```

函数只是简单 m_mirrored 的值与 m_desired 的值是否一样，不一致的话返回 1，表明需要进行更新。

如果不需要更新，那么 2063 行会直接返回。2065 到 2068 行是为了多个进程同时调用 update 函数。在开始真正的进行更新之前，2070 行会把 m_update_in_progress 设置为 1，在更新完成后 2080 行把其值设置为 0。由于更新一次就可以了，因此假如某个进程调用 update 函数时发现其它进程正在调用，所以它等待那个进程结束后直接返回，不必再更新一次。

2076 行调用 uvm_reg_field 的 XupdateX 函数，得到真正要写入到寄存器中的值。这个函数比较简单，不再介绍。2078 行调用 write 函数，把值写入 DUT 中。

除了 uvm_reg 有 update 操作外，uvm_reg_block 也有 update 操作：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：update

1608  task uvm_reg_block::update(output uvm_status_e    status,
1609                                    input   uvm_path_e       path = UVM_DEFAULT_PATH,
1610                                    input   uvm_sequence_base  parent = null,
1611                                    input   int                prior = -1,
1612                                    input   uvm_object         extension = null,
1613                                    input   string             fname = "",
1614                                    input   int                lineno = 0);
```

```
1615      status = UVM_IS_OK;
1616
1617      if (!needs_update()) begin
1618        `uvm_info("RegModel", $sformatf("%s:%0d - RegModel block %s does not need upda
ting",
1619                                  fname, lineno, this.get_name()), UVM_HIGH);
1620
1621      end
1622
1623      `uvm_info("RegModel", $sformatf("%s:%0d - Updating model block %s with %s path",
1624                                  fname, lineno, this.get_name(), path.name ), UVM_HIGH);
1625
1626      foreach (regs[rg_]) begin
1627          uvm_reg rg = rg_;
1628          if (rg.needs_update()) begin
1629              rg.update(status, path, null, parent, prior, extension);
1630              if (status != UVM_IS_OK && status != UVM_HAS_X) begin;
1631                `uvm_error("RegModel", $sformatf("Register \"%s\" could not be updated",
1632                                                        rg.get_full_name()));
1633                return;
1634              end
1635          end
1636      end
1637
1638      foreach (blks[blk_]) begin
1639        uvm_reg_block blk = blk_;
1640        blk.update(status,path,parent,prior,extension,fname,lineno);
1641      end
1642  endtask: update
```

整个函数比较简单，主体部分就是 1626 到 1636 行依次把添加到此 uvm_reg_block 的 uvm_reg 进行更新，1638 到 1641 行把添加到此 uvm_reg_block 的 uvm_reg_block 进行更新。

总结 update 操作，它会比较 m_mirrored 值与 m_desired 值，在两者不一致的情况下，把 m_desired 值写入 DUT 中。

18.7.7. uvm_reg 的 mirror 操作

uvm_reg 的 mirror 也是常用的一种操作，其代码为：

```
文件：src/reg/uvm_reg.svh
类：uvm_reg
函数/任务：mirror

2818  task uvm_reg::mirror(output uvm_status_e          status,
```

```
2819                            input   uvm_check_e             check = UVM_NO_CHECK,
2820                            input   uvm_path_e              path = UVM_DEFAULT_PATH,
2821                            input   uvm_reg_map              map = null,
2822                            input   uvm_sequence_base       parent = null,
2823                            input   int                     prior = -1,
2824                            input   uvm_object              extension = null,
2825                            input   string                  fname = "",
2826                            input   int                     lineno = 0);
2827     uvm_reg_data_t   v;
2828     uvm_reg_data_t   exp;
2829     uvm_reg_backdoor bkdr = get_backdoor();
2830
2831     XatomicX(1);
2832     m_fname = fname;
2833     m_lineno = lineno;
2834
2835
2836     if (path == UVM_DEFAULT_PATH)
2837       path = m_parent.get_default_path();
2838
2839     if (path == UVM_BACKDOOR && (bkdr != null || has_hdl_path()))
2840       map = uvm_reg_map::backdoor();
2841     else
2842       map = get_local_map(map, "read()");
2843
2844     if (map == null)
2845       return;
2846
2847     // Remember what we think the value is before it gets updated
2848     if (check == UVM_CHECK) begin
2849         exp = get();
2850         // Assume that WO* field will readback as 0's
2851         foreach(m_fields[i]) begin
2852             string mode;
2853             mode = m_fields[i].get_access(map);
2854             if (mode == "WO" ||
2855                 mode == "WOC" ||
2856                 mode == "WOS" ||
2857                 mode == "WO1") begin
2858                 exp &= ~(((1 << m_fields[i].get_n_bits())-1)
2859                             << m_fields[i].get_lsb_pos());
2860             end
2861         end
2862     end
2863
2864     XreadX(status, v, path, map, parent, prior, extension, fname, lineno);
2865
2866     if (status == UVM_NOT_OK) begin
2867         XatomicX(0);
2868         return;
2869     end
2870
```

```
2871      if (check == UVM_CHECK) begin
2872          // Check that our idea of the register value matches
2873          // what we just read from the DUT, minus the don't care fields
2874          uvm_reg_data_t   dc = 0;
2875
2876          foreach(m_fields[i]) begin
2877              string acc = m_fields[i].get_access(map);
2878              if (m_fields[i].get_compare() == UVM_NO_CHECK) begin
2879                  dc |= ((1 << m_fields[i].get_n_bits())-1)
2880                          << m_fields[i].get_lsb_pos();
2881              end
2882              else if (acc == "WO" ||
2883                          acc == "WOC" ||
2884                          acc == "WOS" ||
2885                          acc == "WO1") begin
2886                  // Assume WO fields will always read-back as 0
2887                  exp &= ~(((1 << m_fields[i].get_n_bits())-1)
2888                              << m_fields[i].get_lsb_pos());
2889              end
2890          end
2891
2892          if ((v|dc) !== (exp|dc)) begin
2893              `uvm_error("RegModel", $sformatf("Register \"%s\" value read from DUT (0x%h)
 does not match mirrored value (0x%h)",
2894                                                  get_full_name(), v, (exp ^ ('x & dc))));
2895
2896                foreach(m_fields[i]) begin
2897                    if(m_fields[i].get_compare() == UVM_CHECK) begin
2898                        uvm_reg_data_t mask=((1 << m_fields[i].get_n_bits())-1);
2899                        uvm_reg_data_t field = mask << m_fields[i].get_lsb_pos();
2900                        uvm_reg_data_t diff = ((v ^ exp) >> m_fields[i].get_lsb_pos()) & mask;
2901                        if(diff)
2902                            `uvm_info("RegMem",$sformatf("field %s mismatch read=%0d'h%0h
mirrored=%0d'h%0h slice [%0d:%0d]",m_fields[i].get_name(),
2903                                    m_fields[i].get_n_bits(),(v >> m_fields[i].get_lsb_pos()) & mask,
 m_fields[i].get_n_bits(),(exp >> m_fields[i].get_lsb_pos(       ))&mask,
2904                                    m_fields[i].get_lsb_pos()+m_fields[i].get_n_bits()-1,m_fields[i].get_
lsb_pos()),UVM_NONE)
2905                    end
2906                end
2907          end
2908      end
2909
2910      XatomicX(0);
2911  endtask: mirror
```

2831 与 2910 行一起组成一个原子操作保护对。2836 到 2845 行类似的代码前面已经出现过多次。

2848 行根据 check 参数的情况来决定是否把 mirror 之前寄存器的值保存下来。如果设置了 UVM_CHECK，那么后面是需要把 mirror 之前寄存器的值和 mirror 之后

的值比较的，所以需要保存。2849 行通过 get 函数得到 m_desired 值放在 exp 中，2851 行到 2861 行把 exp 中所有只写的 uvm_reg_field 的值修正为 0。2864 行调用 XreadX 函数来得到 DUT 中对应寄存器的值，并且把 register model 中的 m_mirrored，m_desired，value 等值更新。

2876 行到 2890 行遍历每一个 uvm_reg_field。如果此 field 被设置为 UVM_NO_CHECK，那么将此 field 所在的位置 1，其它 field 置 0，这样 2892 行在比较之前，先对数据进行或操作。这样可以把设置为 UVM_NO_CHECK 的字段忽略掉。这里用到了 uvm_reg_field 的 get_compare 函数：

```
文件：src/reg/uvm_reg_field.svh
类：uvm_reg_field
函数/任务：get_compare

1869 function uvm_check_e uvm_reg_field::get_compare();
1870    return m_check;
1871 endfunction
```

函数非常简单，只是返回 m_check 的值。

如果此 field 没有被设置为 UVM_NO_CHECK，那么 2882 到 2889 行检查此 field 的存取策略，如果是只写的话，把 exp 中此 field 所在的比特赋值为 0。这里做的其实与 2854 到 2860 行重复了。

2892 行如果发现 exp 的值与 DUT 的读出值不一致的话，2893 行给出错误提示信息，2896 到 2906 行则查看是哪个字段出错了，并打印出相关字段的信息。

除了 uvm_reg 有 mirror 操作之外，uvm_reg_block 也有 mirror 操作：

```
文件：src/reg/uvm_reg_block.svh
类：uvm_reg_block
函数/任务：mirror

1647 task uvm_reg_block::mirror(output uvm_status_e          status,
1648                                input  uvm_check_e           check = UVM_NO_CHECK,
1649                                input  uvm_path_e            path = UVM_DEFAULT_PATH,
1650                                input  uvm_sequence_base     parent = null,
1651                                input  int                   prior = -1,
1652                                input  uvm_object            extension = null,
1653                                input  string                fname = "",
1654                                input  int                   lineno = 0);
1655    uvm_status_e final_status = UVM_IS_OK;
1656
1657    foreach (regs[rg_]) begin
1658       uvm_reg rg = rg_;
1659       rg.mirror(status, check, path, null,
1660                 parent, prior, extension, fname, lineno);
1661       if (status != UVM_IS_OK && status != UVM_HAS_X) begin;
1662          final_status = status;
```

```
1663            end
1664        end
1665
1666        foreach (blks[blk_]) begin
1667            uvm_reg_block blk = blk_;
1668
1669            blk.mirror(status, check, path, parent, prior, extension, fname, lineno);
1670            if (status != UVM_IS_OK && status != UVM_HAS_X) begin;
1671                final_status = status;
1672            end
1673        end
1674
1675    endtask: mirror
```

1657 到 1664 行调用所有添加到此 uvm_reg_block 的 uvm_reg 的 mirror 函数，1666 到了 1673 行调用所有添加到此 uvm_reg_block 的 mirror 函数。

因此，mirror 操作比较简单，它会从 DUT 中读取的情况来更新 register model 中相应寄存器的值。如果设置了 UVM_CHECK，在操作之前把 register model 中寄存器的值预先取出；从 DUT 中读取数据之后，把读取的数据和之前预先取出的数据比较，不一致则给出错误提示信息。如果设置了 UVM_NO_CHECK，则不做比较，只是更新 register model 中寄存器的值。

19. callback 机制源代码分析

本章分析 callback 机制的源代码。第一节首先讲述 uvm_register_cb 宏，第二节讲述 callback 的 add 函数，uvm_do_callbacks 宏等的源代码。

19.1. 从 uvm_register_cb 宏说起

19.1.1. callback 的实例

在本书前半部分讲述 callback 机制的时候，曾经举了如下的一个例子，本章依然以这个例子来阐述：

```
class A extends uvm_callback;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
    endtask
endclass
typedef uvm_callbacks#(mii_driver, A) A_pool;
class mii_driver extends uvm_driver#(mii_transaction);
    …
    `uvm_register_cb(mii_driver, A)
endclass
```

19.1.2. 类的继承关系及数据结构

上面的例子中出现了 uvm_callback 类，uvm_callbacks 类。uvm_callback 类是真正的会在实际使用中被派生的类，而 uvm_callbacks 类只是单纯的组织这些 uvm_callback 类，它就像一个大池子一样，里面堆满的是各种各样的 uvm_callback。uvm_callback 类的继承关系比较简单，是从 uvm_object 派生而来的：

```
文件：src/base/uvm_callback.svh
类：uvm_callback

1116  class  uvm_callback  extends  uvm_object;
```

所以每一个 uvm_callback 从本质上来说是一个 uvm_object。而 uvm_callbacks 类则复杂许多：

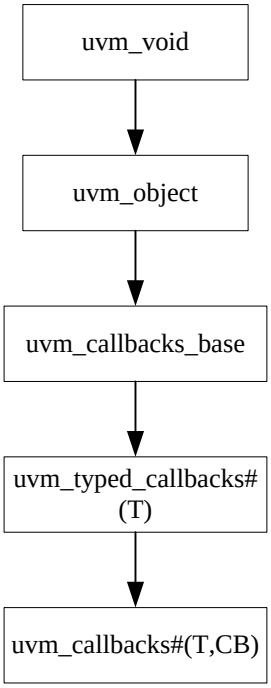


图 19-1 uvm_callbacks 类的继承关系

uvm_callbacks 的先祖类是 uvm_callbacks_base，它的原型为：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks_base

  90  class  uvm_callbacks_base  extends  uvm_object;
```

这个类中提供了两个静态成员变量是需要我们注意的：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks_base

  92    typedef uvm_callbacks_base this_type;
  93
  95    static this_type m_b_inst;
  96
  97    static uvm_pool#(uvm_object,uvm_queue#(uvm_callback)) m_pool;
```

这两个静态成员变量中的一个是 this_type 类型的，根据我们前面在分析 uvm_root 和 factory 机制时所积累的经验，我们很容易猜测这个指针是指向自己的，而且这是一个单实例的。这种猜测对不对呢？我们先放带着这个疑问，慢慢看后面的。

另外一个静态成员变量是一个 uvm_pool。关于 uvm_pool，前面已经提及过，它的本质是一个联合数组，这个联合数组的索引是 uvm_object 类型的，而存放的内容是一个 uvm_queue 的指针，这个 uvm_queue 中存放的内容是 uvm_callback。怎么理解这个有点复杂的 uvm_pool 呢？在使用 callback 时，通常采用如下的做法：

```
class my_callback extends A;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
        $display("pre_tran, the transaction is");
        tr.print();
    endtask
    `uvm_object_utils(my_callback)
endclass
class base_test extends uvm_test;
    …
    my_callback my_cb;
    …
    function void connect_phase(uvm_phase phase);
        my_cb = my_callback::type_id::create("my_cb");
        A_pool::add(mii_env.agent.driver, my_cb);
        …
    endfunction
endclass
```

可以推测，m_pool 中将会有一条记录，这条记录的索引是 mii_env.agent.driver，而这条记录的内容是一个队列，这个队列里面的第一条记录就是 my_cb。为什么会是一个队列？因为我们可以用如下的方式添加无数个 callback：

```
A_pool::add(mii_env.agent.driver, my_cb1);
A_pool::add(mii_env.agent.driver, my_cb2);
…
A_pool::add(mii_env.agent.driver, my_cbn);
```

采用队列就是为了应对这种一个 driver 有多个 callback 的情况。

uvm_callbacks 的直接父类是 uvm_typed_callbacks，类的原型为：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)

178 class uvm_typed_callbacks#(type T=uvm_object) extends uvm_callbacks_base;
```

这是一个参数化的类，也有两个静态成员变量是我们需要注意的：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)

180     static uvm_queue#(uvm_callback) m_tw_cb_q;

188     static this_type m_t_inst;
```

m_t_inst 我们可以大致的推测这是代表着自己，那么 m_tw_cb_q 呢？我们暂且把其当作一个疑问，后面看的时候不断的解释这个疑问。

uvm_callbacks 类的原型为：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)

449 class uvm_callbacks #(type T=uvm_object, type CB=uvm_callback)
450       extends uvm_typed_callbacks#(T);
```

它里面同样的也有两个静态成员变量：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)


471     local static this_type m_inst;

480     static uvm_callbacks#(T,uvm_callback) m_base_inst;
```

m_inst 可以大致理解，而 m_base_inst 也是让人很费解的。与 m_tw_cb_q 一样，我们把其当成一个疑问。

反思一下，为什么会有 uvm_callbacks 这个类？uvm_typed_callbacks 类已经是一个参数化的类了，对于每个类型 T 都有一个 uvm_typed_callbacks？如对于类 mii_driver 就有一个 uvm_typed_callbacks 类与其对应。uvm_callbacks 类是在 uvm_typed_callbacks 类的基础上加入了 uvm_callback 类型。前面的例子中，我们只是把 A 通过 register_cb 宏和 mii_driver 绑定在一起。那么假如还有一个 B 类：

```
class A extends uvm_callback;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
    endtask
endclass
typedef uvm_callbacks#(mii_driver, A) A_pool;
```

```
class B extends uvm_callback;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
    endtask
endclass
typedef uvm_callbacks#(mii_driver, B) B_pool;

class mii_driver extends uvm_driver#(mii_transaction);
    …
    `uvm_register_cb(mii_driver, A)
    `uvm_register_cb(mii_driver, B)
endclass
```

uvm_callbacks 类的存在是为了应付这种多 callback 类的情况。假如只允许有一个 callback 类可以通过 register_cb 宏和 mii_driver 绑定在一起，那么真实 uvm_typed_callbacks 类已经足够了，不需要 uvm_callbacks 类。

除了前面几个类之外，在 callback 机制中还有两个类也是至关重要的：

```
文件：src/base/uvm_callback.svh
类：uvm_typeid_base

  50 class uvm_typeid_base;
  51     static string typename;
  52     static uvm_callbacks_base typeid_map[uvm_typeid_base];
  53     static uvm_typeid_base type_map[uvm_callbacks_base];
  54 endclass
```

```
文件：src/base/uvm_callback.svh
类：uvm_typeid#(type T=uvm_object)

  64 class uvm_typeid#(type T=uvm_object) extends uvm_typeid_base;
  65     static uvm_typeid#(T) m_b_inst;
  66     static function uvm_typeid#(T) get();
  67         if(m_b_inst == null)
  68             m_b_inst = new;
  69         return m_b_inst;
  70     endfunction
  71 endclass
```

这两个类从某种程度上来说是一个小型的 factory 机制。在 factory 机制中，使用 uvm_object_registry 和 uvm_component_registry 中的静态成员变量 me 来唯一的表征一个类型。在这里，同样是使用 uvm_typeid 中的 m_b_inst 来唯一的表征一个类型。

在 callback 机制中，广泛的用到了静态成员变量。这里就牵扯到了一个问题，我们传统意义上认为一个静态成员变量是与某个类相关的，那么对于派生类来说它是否派生父类的静态成员变量呢？

假设有如下三个类：

```
class A;
    static int num;
endclass

class B extends A;
    int b;
endclass

class C extends A;
    int c;
endclass
```

可以肯定的是 A，B，C 类中肯定有一个 num 存在。问题是静态成员变量是以类为单位存在的，B 和 C 都是一个单独的类，那么 B，C 类中的 num 是与 A 中 num 无关的一个变量呢还是说其实与 A::num 是同一个变量？

systemverilog 中静态成员变量是不会继承的，虽然在子类中能够看到父类中的静态成员变量，但是系统并没有为子类给这个静态成员变量重新分配一块空间，而是直接使用的是父类静态成员变量的空间，即与父类的静态成员变量是一个东西。

因此，虽然在 uvm_typeid 中可以用 m_b_inst 来唯一表征一个 T 类型，但是所有的这些 T 类型是共用父类的 typeid_map 和 type_map 数组的。

19.1.3. uvm_register_cb 宏的展开

上面的例子中，出现了 uvm_callback 类，uvm_callbacks 类，及一个 uvm_register_cb 宏。我们首先从 uvm_register_cb 宏展开，宏的定义为：

```
文件：src/macros/uvm_callback_defines.svh
类：无

 59 `define uvm_register_cb(T,CB) \
 60   static local bit m_register_cb_``CB = uvm_callbacks#(T,CB)::m_register_pair(`"T`",`"CB`");
```

宏非常的简单，不过由于使用了一些特殊字符，这里比较难以理解，如果写成下面的方式可能更加让人易懂：

```
static local bit m_register_cb_A = uvm_callbacks#(mii_driver, A)::m_register_pair (“mii_driver”, “A”);
```

宏的本质只是声明了一个静态变量 m_register_cb_A，并且把这个变量初始化。

19.1.4. m_register_pair

在初始化的过程中用到了 m_register_pair 函数。这是属于 uvm_callbacks 类的一个静态成员函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：m_register_pair

 525    static function bit m_register_pair(string tname="", cbname="");
 526      this_type inst = get();
 527
 528      m_typename = tname;
 529      super_type::m_typename = tname;
 530      m_typeid.typename = tname;
 531
 532      m_cb_typename = cbname;
 533      m_cb_typeid.typename = cbname;
 534
 535      inst.m_registered = 1;
 536
 537      return 1;
 538    endfunction
```

函数比较简单，526 行调用 get 函数得到 uvm_callbacks#(mii_driver, A)的一个唯一的实例：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：get

 487    static function this_type get();
 488
 489      if (m_inst == null) begin
 490        uvm_typeid_base cb_base_type;
 491
 492        void'(super_type::m_initialize());
 493
 494        cb_base_type = uvm_typeid#(uvm_callback)::get();
 495        m_cb_typeid  = uvm_typeid#(CB)::get();
 496        m_typeid     = uvm_typeid#(T)::get();
 497
 498        m_inst = new;
 499
 500        if (cb_base_type == m_cb_typeid) begin
 501          $cast(m_base_inst, m_inst);
 502          // The base inst in the super class gets set to this base inst
 503          m_t_inst = m_base_inst;
 504          uvm_typeid_base::typeid_map[m_typeid] = m_inst;
```

```
505             uvm_typeid_base::type_map[m_b_inst] = m_typeid;
506           end
507           else begin
508             m_base_inst = uvm_callbacks#(T,uvm_callback)::get();
509             m_base_inst.m_this_type.push_back(m_inst);
510           end
511
512           if (m_inst == null)
513             `uvm_fatal("CB/INTERNAL","get(): m_inst is null")
514       end
515
516       return m_inst;
517     endfunction
```

这个 get 函数与我们在分析 factory 机制和 uvm_root 时分析过的 get 函数相类似，只是多添加了一些内容。492 行调用了 super_type 的 m_initialize 函数。super_type 是什么？

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)

466     typedef uvm_typed_callbacks#(T) super_type;
```

在我们的例子中，super_type 指的就是 uvm_typed_callbacks#(mii_driver)。m_initialize 函数如下：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_initialize

190     static function this_type m_initialize();
191       if(m_t_inst == null) begin
192         void'(super_type::m_initialize());
193         m_t_inst = new;
194         m_t_inst.m_tw_cb_q = new("typewide_queue");
195       end
196       return m_t_inst;
197     endfunction
```

这个函数有点类似于 get，得到全局的唯一的一个 uvm_typed_callbacks#(mii_driver)类型的实例的指针 m_t_inst。192 行再次调用 super_type 的 m_initialize 函数，这里的 super_type 指的就是 uvm_callbacks_base，其 m_initialize 函数如下：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks_base
函数/任务：m_initialize

 99     static function this_type m_initialize();
100       if(m_b_inst == null) begin
```

```
101          m_b_inst = new;
102          m_pool = new;
103        end
104        return m_b_inst;
105      endfunction
```

这个函数几乎就是一个标准的 get 函数，得到一个全局唯一的一个 uvm_callbacks_base 类型的实例的指针。另外，这个函数还把 m_pool 进行实例化。

回到 uvm_typed_callbacks#(mii_driver)的 m_initialize 函数，194 行把 m_tw_cb_q 实例化。经过这两个 m_initialize 函数，我们看到，系统把 uvm_callbacks_base 的用于存放 callback 实例的 m_pool 实例化了；把 uvm_typed_callbacks 用于存放 callback 的 m_tw_cb_q 实例化了。此时要注意一下，对于 m_pool 来说，全局是只有一个的，因为所有的派生类都是共用父类的静态成员变量的。但是对于 m_tw_cb_q 来说，则是有多少种类型 T，就有多少个这样的队列，这个队列是与类型相关的。假如我们系统中分别给 mii_driver 和 mii_monitor 都通过 register_cb 宏把两种 callback 分别绑定，那么系统中就会有两个 m_tw_cb_q，这两个分别从属于 uvm_typed_callbacks#(mii_driver)和 uvm_typed_callbacks#(mii_monitor)。

回到 get 函数。494 到 496 行通过三个 get 函数得到代表这三种类型的三个指针。在分析 uvm_typeid 函数时说过，其 m_b_inst 是一个能代表某一类型的成员变量，通过 get 函数获得 m_b_inst，其实就相当于是得到了一个能够表征一个类型的指针。498 行把 m_inst 实例化。

500 行判断 cb_base_type 与 m_cb_typeid 的值是否一样。为什么要有这样的判断？我们暂且存下这个疑问。我们先看判断条件不成立的情况，这其实也是我们的例子现在会走的分支。在这种情况下，508 行通过 uvm_callbacks#(mii_driver, uvm_callback)::get 得到的返回值赋值给 m_base_inst。在 uvm_callbacks#(mii_driver, uvm_callback)的 get 函数中，可以看出，500 行的判断条件是成立的。那么 501 行的 m_base_inst 和 m_inst 其实就是同种类型的，因此可以通过 cast 进行转换。503 行把 m_t_inst 赋值为 m_base_inst，也即 uvm_typed_callbacks#(mii_driver)::m_t_inst 的值其实是等于 uvm_callback#(mii_driver, uvm_callback)的 m_inst 的值。504 行在 uvm_typeid_base 的静态联合数组 typeid_map 中插入一条记录，记录的索引是代表 mii_driver 类型的指针 m_typeid，而记录的内容是 uvm_callback#(mii_driver, uvm_callback)唯一的全局实例 m_inst 的指针。505 行在 type_map 中插入一条记录，记录的索引是 m_b_inst，而记录的内容是类型 mii_driver。m_b_inst 是在 uvm_callbacks_base 中定义的，是唯一的 uvm_callbacks_base 类型的实例的指针。

508 行从 get 返回之后，509 行把 m_inst 放入了 m_base_inst 的 m_this_type 中。m_this_type 是在 uvm_callbacks_base 中定义的：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks_base

 92    typedef uvm_callbacks_base this_type;
```

```
108     this_type           m_this_type[$];       //one to many T->T/CB
```

注意到 m_this_type 是一个非静态变量，因此它不是与类相关的，而与具体的实例相关的。因此，对于 uvm_callbacks#(mii_driver, uvm_callback)来说，有它自己的 m_this_type，对 uvm_callbacks#(mii_driver, A)来说，有它自己的 m_this_type。509 行相当于是把全局唯一的一个 uvm_callbacks#(mii_driver, A)的实例的指针放入了 uvm_callbacks#(mii_driver, uvm_callback)的 m_this_type 中。这里其实也可以看出来，假设通过 register_cb 宏把 B，C，…Z 等 callback 和 mii_driver 绑定在了一起，那么最后的结果就是 uvm_callbacks#(mii_driver, uvm_callback)的 m_this_type 中放入了 uvm_callbacks#(mii_driver, A)到 uvm_callbacks# (mii_driver, Z)的指针。也即所有的与 mii_driver 相关的 uvm_callbacks 类的指针都放入了 uvm_callbacks#(mii_driver, uvm_callback)中。大家知道类似于 uvm_callbacks#(mii_driver, A)本身就是一个池子，这个池子中可以加入很多从 A 派生来的 callback 的实例的指针，那么 uvm_callbacks#(mii_driver, uvm_callback)就相当于是一个更加大的池子，它的 m_this_type 中有指向其它小池子的指针。而 uvm_typed_callbacks#(mii_driver)中的 m_t_inst 指向的就是这个大池子。

回到 get 函数，512 行判断 m_inst 是否为 null，并给出错误提示。516 行返回 m_inst，也即唯一的 uvm_callbacks#(mii_driver, A)这个小池子的指针。

回到 m_register_pair 函数。528 到 533 行给一些表征名字的变量赋值。经过这几行之后，uvm_callbacks#(mii_driver, A)的 m_typename=”mii_driver”，uvm_typed_callbacks#(mii_driver)的 m_typename=”mii_driver”，m_typeid 即全局唯一的 uvm_typeid#(mii_driver)的 typename=”mii_driver”， uvm_callbacks#(mii_driver, A)的 m_cb_typename=”A”，m_cb_typeid 即全局唯一的 uvm_typeid#(A)的 typename=”A”。

535 行表明 m_register_pair 已经调用过了。537 行直接返回 1。到此，整个函数分析完毕。

19.2. callback 的使用

19.2.1. uvm_callbacks::add(一)

前面已经说过，在定义好了 callback 接口后，可以使用如下的方式使用 callback:

```
class my_callback extends A;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
        $display("pre_tran, the transaction is");
        tr.print();
    endtask
    `uvm_object_utils(my_callback)
endclass
class base_test extends uvm_test;
    …
    my_callback my_cb;
    …
    function void connect_phase(uvm_phase phase);
        my_cb = my_callback::type_id::create("my_cb");
        A_pool::add(mii_env.agent.driver, my_cb);
        …
    endfunction
endclass
```

这里用到了 add 函数。这是 uvm_callbacks 的一个静态函数，其定义为：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：add

 570    static function void add(T obj, uvm_callback cb, uvm_apprepend ordering=UVM_APPEN
D);
 571      uvm_queue#(uvm_callback) q;
 572      string nm,tnm;
 573
 574      void'(get());
 575
 576      if (cb==null) begin
 577        if (obj==null)
 578          nm = "(*)";
 579        else
 580          nm = obj.get_full_name();
 581
 582        if (m_base_inst.m_typename!="")
 583          tnm = m_base_inst.m_typename;
 584        else if (obj != null)
 585          tnm = obj.get_type_name();
 586        else
 587          tnm = "uvm_object";
 588
 589        uvm_report_error("CBUNREG",
 590                         {"Null callback object cannot be registered with object ",
 591                          nm, " (", tnm, ")"}, UVM_NONE);
 592        return;
 593      end
 594
 595      if (!m_base_inst.check_registration(obj,cb)) begin
          …
```

574 行调用 get 函数，这里是为了避免 get 函数一次也没有调用的情况。在我们的例子中，由于 m_inst 其实已经不为 null 了，所以这次调用其实是直接返回 m_inst 值的。

576 到 593 进行错误检查，保证 cb 的值不为 null。其中 577 到 587 行都是为了 589 行的信息打印准备的。

19.2.2. uvm_callbacks_base:: check_registration

uvm_callbacks::add 函数的 595 行调用 check_registration 函数，注意，这里调用的是 m_base_inst 的。m_base_inst 指的是 uvm_callbacks#(mii_driver, uvm_callback) 的实例的指针。uvm_callbacks 类并没有重载这个函数，所以这里调用的是基类 uvm_callbacks_base 类的 check_registration 函数。其定义如下：

文件：src/base/uvm_callback.svh
类：uvm_callbacks_base
函数/任务：check_registration

```
138    function bit check_registration(uvm_object obj, uvm_callback cb);
139      this_type st, dt;
140
141      if (m_is_registered(obj,cb))
142        return 1;
143
144      // Need to look at all possible T/CB pairs of this type
145      foreach(m_this_type[i])
146        if(m_b_inst != m_this_type[i] && m_this_type[i].m_is_registered(obj,cb))
147          return 1;
148
149      if(obj == null) begin
150        foreach(m_derived_types[i]) begin
151          dt = uvm_typeid_base::typeid_map[m_derived_types[i] ];
152          if(dt != null && dt.check_registration(null,cb))
153            return 1;
154        end
155      end
156
157      return 0;
158    endfunction
```

141 行调用 m_is_registered 函数：

文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：m_is_registered

```
540    virtual function bit m_is_registered(uvm_object obj, uvm_callback cb);
541      if(m_is_for_me(cb) && m_am_i_a(obj)) begin
542        return m_registered;
543      end
544    endfunction
```

541 行用到了两个简单的函数：m_is_for_me 和 m_am_i_a，m_is_for_me 的定义为：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：m_is_for_me

547    virtual function bit m_is_for_me(uvm_callback cb);
548      CB this_cb;
549      return($cast(this_cb,cb));
550    endfunction
```

m_is_for_me 会检查输入的 cb 是不是 CB 类型的。其实就是指 my_cb 是不是 uvm_callback 类型的。由于 my_cb 是 my_callback 类型的一个实例，而 my_callback 是从 A 派生来的，A 又是从 uvm_callback 派生来的，所以这里的返回值将会是 1。

m_am_i_a 是在 uvm_typed_callbacks 中定义的：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_am_i_a

200    virtual function bit m_am_i_a(uvm_object obj);
201      T this_type;
202      if (obj == null)
203        return 1;
204      return($cast(this_type,obj));
205    endfunction
```

这里就是检查输入的 obj 是不是 T 类型的，也即查看 mii_env.agent.driver 是不是 mii_driver 类型的。这里也会返回 1。

从而 542 行会直接返回 m_registered。在 m_register_pair 中已经把此变量赋值为 1，所以 m_is_registered 最终会返回 1。

回到 check_registration 函数，141 行的判断条件满足，那么此函数返回 1。函数接下来的几段代码则是用于其它情况。后面会有介绍，这里先跳过。

19.2.3. uvm_callbacks::add(二)

回到 add 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：add

 595      if (!m_base_inst.check_registration(obj,cb)) begin
 596
 597        if (obj==null)
 598          nm = "(*)";
 599        else
 600          nm = obj.get_full_name();
 601
 602        if (m_base_inst.m_typename!="")
 603          tnm = m_base_inst.m_typename;
 604        else if(obj != null)
 605          tnm = obj.get_type_name();
 606        else
 607          tnm = "uvm_object";
 608
 609        uvm_report_warning("CBUNREG",
 610                           {"Callback ", cb.get_name(), " cannot be registered with obje
ct ",
 611                           nm, " because callback type ", cb.get_type_name(),
 612                           " is not registered with object type ", tnm }, UVM_NONE);
 613      end
 614
 615      if(obj == null) begin
 616
 617        if (m_cb_find(m_t_inst.m_tw_cb_q,cb) != -1) begin
 618
 619          if (m_base_inst.m_typename!="")
 620            tnm = m_base_inst.m_typename;
 621          else if (obj != null)
 622            tnm = obj.get_type_name();
 623          else tnm = "uvm_object";
 624
 625          uvm_report_warning("CBPREG",
 626                             {"Callback object ", cb.get_name(),
 627                             " is already registered with type ", tnm }, UVM_NONE);
 628        end
 629        else begin
 630          `uvm_cb_trace_noobj(cb,$sformatf("Add (%s) typewide callback %0s for type %s",
 631                              ordering.name(), cb.get_name(), m_base_inst.m_typename))
 632          m_t_inst.m_add_tw_cbs(cb,ordering);
 633        end
 634      end
```

```
635
636       else begin
637
638         `uvm_cb_trace_noobj(cb,$sformatf("Add (%s) callback %0s to object %0s ",
639                                          ordering.name(), cb.get_name(), obj.get_full_name()))
640
641         q = m_base_inst.m_pool.get(obj);
642
643         if (q==null) begin
644           q=new;
645           m_base_inst.m_pool.add(obj,q);
646         end
647
648         if(q.size() == 0) begin
649           // Need to make sure that registered report catchers are added. This
650           // way users don't need to set up uvm_report_object as a super type.
651           uvm_report_object o;
652
653           if($cast(o,obj)) begin
654             uvm_queue#(uvm_callback) qr;
655             qr = uvm_callbacks#(uvm_report_object,uvm_callback)::m_t_inst.m_tw_cb_q;
656             for(int i=0; i<qr.size(); ++i)
657                   q.push_back(qr.get(i));
658           end
659
660           for(int i=0; i<m_t_inst.m_tw_cb_q.size(); ++i)
661             q.push_back(m_t_inst.m_tw_cb_q.get(i));
662         end
663
```

595 行的条件不满足，所以接下来跳到 615 行，这里判断 obj 的值是为 null，很明显，我们的 mii_env.agent.driver 是不为 null 的，那么此处执行 638 行的分支。641 到 646 行检查 m_pool 中是否有 mii_env.agent.driver 一条记录，如果没有，那么就新插入一条，否则把这条记录的指针赋值给 q。也就是说 q 最终会指向 m_pool 中与 mii_env.agent.driver 相关的一条记录。q 是一个队列，其中存放的内容是 uvm_callback 类型的。

653 行判断 mii_env.agent.driver 是不是一个 uvm_report_object，如果是的话，那么接下来 654 到 657 行把系统中添加的与 uvm_report_object 类型相关的 callback 同样加入到 q 中。这里有点难以理解，因为它牵扯到了 m_tw_cb_q 这个队列，而到现在为止我们还不知道这个队列中会存放什么内容。后面会介绍，这里存放的是与类型相关的 callback 实例的指针。我们的例子中加入的 callback 只是适应于某一特定的实例的（mii_env.agent.driver），那假如系统中所有的 mii_driver 类型的实例都要使用某个 callback，那么这种 callback 是存放在 uvm_typed_callbacks#(mii_driver)::m_tw_cb_q 中的，其实 611 行到 633 行做的就是这样的事情，后面会有详细说明。

不过这里有一点是值得反思的，那就是为什么要把 uvm_report_object 相关的

callback 也要加入到 mii_env.agent.driver 的 callback 队列中？我们定义 callback 的时候，如果是指定了这个 callback 是适用于某一类型（而不是只适用于某个实例，在 callback 机制中的表现方式就是这个 callback 是存放在 m_tw_cb_q 队列中）的话，那么此类型的派生类也应该使用这些 callback。因为从本质上说，mii_env.agent.driver 也是一个 uvm_report_object，所以也应该使用适用于 uvm_report_object 的 callback。

继续看 add 函数，660 行到 661 行把属于 mii_driver 类型的 callback 加入到 mii_env.agent.driver 的 callback 队列中来。因为很明显的如果一个 callback 是适用于所有 mii_driver 类型的话，那么作为 mii_driver 的一个实例，mii_env.agent.driver 的 callback 队列中理应加入这个 callback。

回过头来看一下 648 行为什么要判断 q 的大小是否为 0。假设下面两个调用：

```
A_pool::add(mii_env.agent.driver, my_cb);
A_pool::add(mii_env.agent.driver, my_cb2);
```

那么第二次调用 add 时，其实 q 中已经放入了属于 uvm_report_object 类型的 callback，也已经加入了属于 mii_driver 类型的 callback，所以不必再加一次了。

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：add

 664          //check if already exists in the queue
 665          if(m_cb_find(q,cb) != -1) begin
 666            uvm_report_warning("CBPREG", { "Callback object ", cb.get_name(), " is already
registered",
 667                                           " with object ", obj.get_full_name() }, UVM_NONE);
 668          end
 669          else begin
 670            if(ordering == UVM_APPEND)
 671              q.push_back(cb);
 672            else
 673              q.push_front(cb);
 674          end
 675        end
 676    endfunction
```

665 行调用 m_cb_find 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_cb_find

 225    static function int m_cb_find(uvm_queue#(uvm_callback) q, uvm_callback cb);
 226      for(int i=0; i<q.size(); ++i)
 227        if(q.get(i) == cb)
 228          return i;
 229      return -1;
 230    endfunction
```

函数其实相当简单，就是为了查看一下要加入的 cb 是不是已经存在于 callback 队列中了。如下面的情况：

```
A_pool::add(mii_env.agent.driver, my_cb);
A_pool::add(mii_env.agent.driver, my_cb);
```

那么在第二次调用时，665 行的条件就会满足。而第一次调用时，则不会满足。

670 行到 673 行把要加入的 cb 放入 q 中。这里会判断 ording 的值。ording 是 add 函数的一个输入的参数，用于表明把新加入的这个 callback 是放在 callback 队列的前面还是后面。如下所示：

```
A_pool::add(mii_env.agent.driver, my_cb);
A_pool::add(mii_env.agent.driver, my_cb2);
```

假如 my_cb 和 my_cb2 会处理同一数据，如果先用 my_cb 处理跟再用 my_cb2 处理结果跟先用 my_cb2 再用 my_cb 处理的结果不一样，那么此时就需要指定 ording 的顺序。这个参数平时一般不会用到。

到此，callback 实例加入到队列中的动作完成。小结一下，假如是往某特定的实例加入 callback，那么系统最终会在 m_pool 中插入一条有关这个实例的记录，这条记录中包含了要加入的 callback，同时还包括适用于 uvm_report_object 类型的 callback（前提是此实例所属的类型是从 uvm_report_object 派生来的，及适用于此实例所属类型的所有 callback。

19.2.4. 加入适用于某一类型的 callback

上一节讲述了适用于某一类型的 callback，本节详细解释。假设我们在加入 callback 时是这样加入的：

```
my_cb = my_callback::type_id::create("my_cb");
A_pool::add(null, my_cb);
```

那么 add 函数的 595 行调用 check_registration 会依然返回 1（读者可以看上节的代码仔细想一下）。

由于 obj 为 null，所以 615 行的条件满足。617 行会判断要加入的 cb，即 my_cb 是不是已经在 uvm_typed_callbacks#(mii_driver)::m_tw_cb_q 中了。如果有了，说明已经加入过了，所以不必加入，只是给出错误提示。如果没有，那么 632 行会调用 m_add_tw_cbs。这是在 uvm_callbacks_base 中定义，在 uvm_typed_callbacks 中重载的函数：

```
文件：src/base/uvm_callback.svh
```

```
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_add_tw_cbs

 233    virtual function void m_add_tw_cbs(uvm_callback cb, uvm_apprepend ordering);
 234      super_type cb_pair;
 235      uvm_object obj;
 236      T me;
 237      uvm_queue#(uvm_callback) q;
 238      if(m_cb_find(m_t_inst.m_tw_cb_q,cb) == -1) begin
 239          if(ordering == UVM_APPEND)
 240              m_t_inst.m_tw_cb_q.push_back(cb);
 241          else
 242              m_t_inst.m_tw_cb_q.push_front(cb);
 243      end
 244      if(m_t_inst.m_pool.first(obj)) begin
 245        do begin
 246          if($cast(me,obj)) begin
 247            q = m_t_inst.m_pool.get(obj);
 248            if(q==null) begin
 249              q=new;
 250              m_t_inst.m_pool.add(obj,q);
 251            end
 252            if(m_cb_find(q,cb) == -1) begin
 253              if(ordering == UVM_APPEND)
 254                q.push_back(cb);
 255              else
 256                q.push_front(cb);
 257            end
 258          end
 259        end while(m_t_inst.m_pool.next(obj));
 260      end
 261      foreach(m_derived_types[i]) begin
 262        cb_pair = uvm_typeid_base::typeid_map[m_derived_types[i] ];
 263        if(cb_pair != this)
 264          cb_pair.m_add_tw_cbs(cb,ordering);
 265      end
 266    endfunction
```

238 行检查是否已经加入到 m_tw_cb_q 中，没有加入，则 239 到 242 行根据 ording 加入进去。

244 行用到 first 函数，它是 uvm_pool 的一个函数，因为 obj 是一个 uvm_object 类型的变量，所以 first 返回值表示是否在 m_pool 中找到了一条索引为 uvm_object 类型及其派生类的记录。

244 到 260 行用于遍历 m_pool 中所有的记录，如果这条记录的索引为 T 类型的，即 mii_driver 类型的，那么就向此条记录的 queue 中加入要加入的 callback。以下面的例子解释：

```
A_pool::add(mii_env.agent.driver, my_cb);
A_pool::add(null, my_cb2);
```

经过第一次调用 add 后，那么 m_pool 中有了一条索引为 mii_env.agent.driver 的记录，第二次 add 调用时，将会向这条记录所指向的队列中插入 my_cb2。这其实是容易理解的，因为后加入的 my_cb2 我们是希望适用于所有的 mii_driver 的实例的，它同样应该是 mii_env.agent.driver 的一个 callback。

261 到 265 行的语句会让人有些费解，这里暂且先跳过，后面章节会有介绍。

19.2.5. uvm_do_callbacks

如前，经过上面两步后，我们看定义在 mii_driver 的 uvm_do_callbacks 宏是如何运转的：

```
task mii_driver::main_phase();
    …
    while(1) begin
        seq_item_port.get_next_item(req);
        `uvm_do_callbacks(mii_driver, A, pre_tran(this, req))
        …
    end
endtask
```

宏的展开为：

```
文件：src/macros/uvm_callback_defines.svh
类：无

139 `define uvm_do_callbacks(T,CB,METHOD) \
140   `uvm_do_obj_callbacks(T,CB,this,METHOD)
```

这里直接调用 uvm_do_obj_callbacks 宏：

```
文件：src/macros/uvm_callback_defines.svh
类：无

162 `define uvm_do_obj_callbacks(T,CB,OBJ,METHOD) \
163     begin \
164       uvm_callback_iter#(T,CB) iter = new(OBJ); \
165       CB cb = iter.first(); \
166       while(cb != null) begin \
167         `uvm_cb_trace_noobj(cb,$sformatf(`"Executing callback method 'METHOD' for callba
ck %s (CB) from %s (T)`",cb.get_name(), OBJ.get_full_name())) \
168           cb.METHOD; \
169           cb = iter.next(); \
170       end \
171     end
```

164 行声明了一个 uvm_callback_iter#(T,CB)，即 uvm_callback_item#(mii_driver, A)

类型的变量 iter，并把其实例化。

uvm_callback_iter 是 callback 机制中专门用于查找与实例相关的 callback 的一个类，其原型如下：

```
文件：src/base/uvm_callback.svh
类：uvm_callback_iter#(type T = uvm_object, type CB = uvm_callback)

1013 class uvm_callback_iter#(type T = uvm_object, type CB = uvm_callback);
```

它没有派生自任何类。其 new 函数为：

```
文件：src/base/uvm_callback.svh
类：uvm_callback_iter#(type T = uvm_object, type CB = uvm_callback)
函数/任务：new

1024     function new(T obj);
1025         m_obj = obj;
1026     endfunction
```

m_obj 是其中定义的一个 T 类型的变量：

```
文件：src/base/uvm_callback.svh
类：uvm_callback_iter#(type T = uvm_object, type CB = uvm_callback)

1016     local T     m_obj;
```

在我们的例子中，m_obj 其实就是 mii_driver 类型的变量，它指向的是 mii_env.agent. driver。

165 行调用 uvm_callback_iter 的 first 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callback_iter#(type T = uvm_object, type CB = uvm_callback)
函数/任务：first

1034     function CB first();
1035         m_cb = uvm_callbacks#(T,CB)::get_first(m_i, m_obj);
1036         return m_cb;
1037     endfunction
```

它会调用 uvm_callbacks#(mii_driver, A)的 get_first 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：get_first

 825    static function CB get_first (ref int itr, input T obj);
 826      uvm_queue#(uvm_callback) q;
 827      CB cb;
 828      void'(get());
```

```
829       m_get_q(q,obj);
830       for(itr = 0; itr<q.size(); ++itr)
831          if($cast(cb, q.get(itr)) && cb.callback_mode())
832             return cb;
833       return null;
834    endfunction
```

828 行再次调用 get，这里的调用将会直接返回，因为 m_inst 已经不是 null 了。829 行调用 m_get_q 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：m_get_q

799    static function void m_get_q (ref uvm_queue #(uvm_callback) q, input T obj);
800       if(!m_base_inst.m_pool.exists(obj)) begin //no instance specific
801          q = (obj == null) ? m_t_inst.m_tw_cb_q : m_t_inst.m_get_tw_cb_q(obj);
802       end
803       else begin
804          q = m_base_inst.m_pool.get(obj);
805          if(q==null) begin
806             q=new;
807             m_base_inst.m_pool.add(obj,q);
808          end
809       end
810    endfunction
```

这个函数稍微有点复杂。800 行判断 m_pool 是否有索引为 mii_env.agent.driver 的记录，如果有的话，那么 804 行将把这条记录取出，赋值给 q，由于 q 是一个 ref 形式的参数，所以 uvm_callbacks#(mii_driver, A)的 get_first 函数最终会得到 m_pool 中与 mii_env.agent.driver 相关的记录。在我们的例子中，如果我们通过如下方式添加 callback，就会是这种情况：

```
A_pool::add(mii_env.agent.driver, my_cb);
```

但是假如是使用如下的方式：

```
A_pool::add(null, my_cb);
```

那么我们知道，在 m_pool 中是不会有关于 mii_env.agent.driver 的记录的，而只会在 uvm_typed_callbacks#(mii_driver)::m_tw_cb_q 中有一条 my_cb 的记录。这种情况下会执行 801 行的分支。由于 obj 即 mii_env.agent.driver 不为 null，所以将会调用 m_get_tw_cb_q 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_get_tw_cb_q

208    virtual function uvm_queue#(uvm_callback) m_get_tw_cb_q(uvm_object obj);
209       if(m_am_i_a(obj)) begin
```

```
210         foreach(m_derived_types[i]) begin
211           super_type dt;
212           dt = uvm_typeid_base::typeid_map[m_derived_types[i] ];
213           if(dt != null && dt != this) begin
214             m_get_tw_cb_q = dt.m_get_tw_cb_q(obj);
215             if(m_get_tw_cb_q != null)
216               return m_get_tw_cb_q;
217           end
218         end
219         return m_t_inst.m_tw_cb_q;
220       end
221       else
222         return null;
223   endfunction
```

209 行先判断 obj，即 mii_env.agent.driver 是不是 mii_driver 类型的，这里的答案是肯定的。210 到 218 行又出现了 m_derived_types，在 m_add_tw_cbs 函数中，这个联合数组曾经出现过一次，与前面相同，这里同样先跳过。

219 行将直接返回 uvm_typed_callbacks#(mii_driver)::m_tw_cb_q。也就是说，uvm_callbacks#(mii_driver, A) 的 get_first 函数最终会得到 uvm_typed_callbacks#(mii_driver):: m_tw_cb_q，也即所有的 mii_driver 类型应该调用的 callback。

回到 uvm_callbacks#(mii_driver, A)的 get_first 函数，830 行将会遍历 q 中所有的记录，如果这条记录中的 callback 是 A 类型的，那么就直接返回。无论是用下面的方式添加 callback：

```
A_pool::add(mii_env.agent.driver, my_cb);
```

还是用下面的方式添加 callback：

q 中最终都将会是 my_cb，从而会将 my_cb 的指针返回。那么什么情况下 831 行的条件将会不满足呢？

假如除了 A 之外，还另外有一个 B，即：

```
class B extends uvm_callback;
…
endclass
typedef uvm_callbacks#(mii_driver, B) B_pool;
class mii_driver extends uvm_driver;
    …
    `uvm_register_cb(mii_driver, A);
    `uvm_register_cb(mii_driver, B);
endclass
```

然后在从 B 派生一个实例，并将其加入到 B_pool 中：

```
class B_cb extends B;
    …
```

```
endclass
B_cb my_b_cb;
my_b_cb = B_cb::type_id::create("my_b_cb");
B_pool::add(mii_env.agent.driver, my_b_cb);
```

那么最终 831 行中 q.get(itr)将会是得到 my_b_cb，它是属于 B 类型的，而不是 A 类型的，所以 831 行的判断条件不成立。

831 行还出现了 callback_mode 函数：

```
文件：src/base/uvm_callback.svh
类：uvm_callback
函数/任务：callback_mode

1135    function bit callback_mode(int on=-1);
1136      if(on == 0 || on == 1) begin
1137        `uvm_cb_trace_noobj(this,$sformatf("Setting callback mode for %s to %s",
1138                get_name(), ((on==1) ? "ENABLED":"DISABLED")))
1139      end
1140      else begin
1141        `uvm_cb_trace_noobj(this,$sformatf("Callback mode for %s is %s",
1142                get_name(), ((m_enabled==1) ? "ENABLED":"DISABLED")))
1143      end
1144      callback_mode = m_enabled;
1145      if(on==0) m_enabled=0;
1146      if(on==1) m_enabled=1;
1147    endfunction
```

这个函数比较有意思，一个函数完成了两件事情。如果输入的参数为-1，那么就返回 m_enabled 的值，如果输入的为 0 或 1，那么就表示让 m_enabled 的值为 0 或 1。也就是说这个函数还有设置 m_enabled 功能。

通过控制 m_enabled 功能，那么可以暂时的让某个 callback 失效，假如在 base_test 中加入了一个 callback：

```
A_pool::add(mii_env.agent.driver, my_cb);
```

但是在某个 case 中，我们不想用这个 callback，那么不必调用 delete 函数，可以直接在 case 中这样写：

```
my_cb.callback_mode(0);
```

因此 831 行一方面是要检查从 q 中出来的 callback 是不是所要类型，另外还要查看这个 callback 是不是失效了。

uvm_callbacks#(mii_driver, A)的 get_first 最终将会把 q 中第一个符合条件的 callback 返回给 uvm_callback_iter#(mii_driver, A)的 first 函数，从而 first 函数的返回值就是第一个符合条件的 callback。

回到 uvm_do_obj_callbacks 宏。在 cb 不为 null 的情况下，那么将会执行 168 行，

cb.METHOD，即 cb.pre_tran。在我们的例子中就是 my_cb.pre_tran。当执行完毕后，169 行将会从系统中查看第二个符合条件的 callback，一直到没有符合条件的 callback 为止。

19.2.6. 子类继承父类的 callback

在上面的分析中，我们遇到过 m_derived_types 联合数组。本节分析其中存储的是什么内容。假如我们已经有了这样的定义：

```
class A extends uvm_callback;
    virtual task pre_tran(mii_driver mii_drv, ref mii_transaction tr);
    endtask
endclass
typedef uvm_callbacks#(mii_driver, A) A_pool;
class mii_driver extends uvm_driver#(mii_transaction);
    …
    `uvm_register_cb(mii_driver, A)
endclass
task mii_driver::main_phase();
    …
    `uvm_do_callbacks(mii_driver, A, pre_tran(this, req))

endtask
```

现在我们从 mii_driver 派生一个新的类，并且也希望能够使用 A 类型的 callback，那么可以这么做：

```
class derived_drived extends mii_driver;
    `uvm_set_super_type(derived_driver, mii_drived);
    …
endclass
task derived_driver::main_phase();
    …
    `uvm_do_callbacks(mii_driver, A, pre_tran(this, req))

endtask
```

我们看一下 set_super_type 宏：

```
文件：src/macros/uvm_callback_defines.svh
类：无

 98 `define uvm_set_super_type(T,ST) \
 99    static local bit m_register_``T``ST = uvm_derived_callbacks#(T,ST)::register_super_type(`"T`
",`"ST`");
```

放在我们例子中就是：

```
static local_bit m_register_derived_drivermii_driver = uvm_derived_callbacks# (derived_driver, mii
_driver):: register_super_type(“derived_driver”, “mii_driver”);
```

register_super_type 函数是 uvm_derived_callbacks 的一个静态函数。uvm_derived_callbacks 的原型为：

```
文件：src/base/uvm_callback.svh
类：uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callback)

 942 class uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callb
ack)
 943       extends uvm_callbacks#(T,CB);
```

register_super_type 函数的定义为：

```
文件：src/base/uvm_callback.svh
类：uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callback)
函数/任务：register_super_type

 967    static function bit register_super_type(string tname="", sname="");
 968       this_user_type u_inst = this_user_type::get();
 969       this_type      inst = this_type::get();
 970       uvm_callbacks_base s_obj;
 971
 972       this_user_type::m_t_inst.m_typename = tname;
 973
 974       if(sname != "") m_s_typeid.typename = sname;
 975
 976       if(u_inst.m_super_type != null) begin
 977          if(u_inst.m_super_type == m_s_typeid) return 1;
 978          uvm_report_warning("CBTPREG", { "Type ", tname, " is already registered to super
type ",
 979             this_super_type::m_t_inst.m_typename, ". Ignoring attempt to register to super type
",
 980             sname}, UVM_NONE);
 981          return 1;
 982       end
 983       if(this_super_type::m_t_inst.m_typename == "")
 984          this_super_type::m_t_inst.m_typename = sname;
 985       u_inst.m_super_type = m_s_typeid;
 986       u_inst.m_base_inst.m_super_type = m_s_typeid;
 987       s_obj = uvm_typeid_base::typeid_map[m_s_typeid];
 988       s_obj.m_derived_types.push_back(m_typeid);
 989       return 1;
 990    endfunction
```

这个函数里出现了 this_user_type，this_type，this_super_type，其定义分别为：

```
文件：src/base/uvm_callback.svh
类：uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callback)
```

```
945    typedef  uvm_derived_callbacks#(T,ST,CB)  this_type;
946    typedef  uvm_callbacks#(T)                this_user_type;
947    typedef  uvm_callbacks#(ST)               this_super_type;
```

968 行 u_inst 最终的值将会是 uvm_callbacks# (derived_driver, uvm_callback)全局唯一的一个实例。

969 行调用了 get 函数，其定义为：

```
文件：src/base/uvm_callback.svh
类：uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callback)
函数/任务：get

957    static function this_type get();
958       m_user_inst = this_user_type::get();
959       m_super_inst = this_super_type::get();
960       m_s_typeid = uvm_typeid#(ST)::get();
961       if(m_d_inst == null) begin
962          m_d_inst = new;
963       end
964       return m_d_inst;
965    endfunction
```

这里出现了四个成员变量，其定义分别为：

```
文件：src/base/uvm_callback.svh
类：uvm_derived_callbacks#(type T=uvm_object, type ST=uvm_object, type CB=uvm_callback)

950    static this_type m_d_inst;
951    static this_user_type m_user_inst;
952    static this_super_type m_super_inst;
955    static uvm_typeid_base m_s_typeid;
```

get 函数被调用后，m_d_inst 将会指向唯一的 uvm_derived_callbacks#(derived_driver, mii_driver, uvm_callback)的实例，m_user_inst 将会指向 uvm_callbacks# (derived_driver, uvm_callback)全局唯一的一个实例。m_super_inst 将会指向 uvm_callbacks# (mii_driver, uvm_callback)全局唯一的一个实例。m_s_typeid 将会指向代表 mii_driver 类型的唯一的实例 uvm_typeid#(mii_driver)::m_b_inst，它唯一的表征了类型 mii_driver。

register_super_type 函数 969 行的 inst 最终会指向唯一的 uvm_derived_callbacks# (derived_driver, mii_driver, uvm_callback)的实例。

972 行把 uvm_typed_callbacks#(derived_driver)唯一全局实例的 m_typename 设置为 derived_driver。

973 行则设置 uvm_typeid#(mii_driver)::m_b_inst 的 typename。

976 行检查是否已经设置过了 uvm_callbacks# (derived_driver, uvm_callback)的

m_super_type 值。m_super_type 是在 uvm_callbacks_base 中定义的变量：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks_base

 109    uvm_typeid_base  m_super_type;          //one  to  one  relation
```

如果设置过了，那么 977 行检查设置的父类型是不是 mii_driver，是的话就直接返回，否则 978 行给出警告信息，981 行返回。

在我们的例子中，很明显是没有设置过的。所以 985 行把 uvm_callbacks# (derived_driver, uvm_callback)的 m_super_type 设置为 m_s_typeid，而后者唯一的表征了 mii_driver，也就是把 m_super_type 设置为了 mii_driver。

986 行把 uvm_callbacks# (derived_driver, uvm_callback)::m_base_inst 的 m_super_type 设置为 mii_driver。m_base_inst 是什么？是 uvm_callbacks# (derived_driver, uvm_callback)的唯一全局实例，这是一个大池子，它管理着所有与 derived_driver 相关的小池子。

986 行的设置其实与 985 行是等价的，二者实际上是对同一个实例的 m_super_type 进行设置。不过两者也有不等价的情况。在我们的例子中，是 uvm_derived_callbacks# (derived_driver, mii_driver, uvm_callback)，假如是 uvm_derived_callbacks# (derived_driver, mii_driver, C)，其中 C 是从 uvm_callback 派生来的，读者可以仔细的想一下，这种情况下这两句话将会是不等价的。

987 行从 typeid_map 中取出一条记录，这条记录的索引是 mii_driver 类型，内容是什么？在分析 register_cb 宏时，曾经提到过 get 函数，那个函数中会向 typeid_map 插入索引为 mii_driver 一条记录，记录的内容是 uvm_callback#(mii_driver, uvm_callback)唯一的全局实例 m_inst 的指针，也即是管理所有的 mii_driver 的 callback 的大池子，这个大池子通过管理众多小池子来管理 mii_driver 的所有的 callback。

988 行给 uvm_callback#(mii_driver, uvm_callback)的 m_derived_types 中插入一条记录，记录的内容就是 m_typeid，它唯一的表征了 derived_driver。

因此总结一下，所谓的 register_super_type，其本质就是在父类的 uvm_callback# (mii_driver, uvm_callback)的 m_derived_types 中插入子类的一条记录，并把子类的 uvm_callbacks# (derived_driver, uvm_callback)的 m_super_type 设置为父类，即 mii_driver。

19.2.7. 子类继承父类 callback 的使用

上节介绍了 set_super_type 宏，通过此宏分别在相应的类中设置了父子关系。

在上节的前提上，我们看下面的例子：

```
my_cb = my_callback::type_id::create("my_cb");
A_pool::add(null, my_cb);
```

前面分析过，add 函数最终会执行 632 行的分支，调用 m_add_tw_cbs：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_add_tw_cbs

233   virtual function void m_add_tw_cbs(uvm_callback cb, uvm_apprepend ordering);
234     super_type cb_pair;
        …
261     foreach(m_derived_types[i]) begin
262       cb_pair = uvm_typeid_base::typeid_map[m_derived_types[i] ];
263       if(cb_pair != this)
264         cb_pair.m_add_tw_cbs(cb,ordering);
265     end
266   endfunction
```

前面已经介绍过 260 行之前的语句向 m_tw_cb_q 中插入要加入的 cb，并把其扩展到 m_pool 中。261 行遍历 m_derived_types 数组。m_derived_types 数组中有什么内容？上节介绍过，经过 set_super_type 宏之后，uvm_callback#(mii_driver, uvm_callback)的 m_derived_types 中有一条关于 derived_driver 的记录，但是像 uvm_callback#(mii_driver, A)的 m_derived_types 中则没有。我们是调用的 A_pool，即 uvm_callback#(mii_driver, A)的 add 函数，那么是否意味着 261 行里面将会一条记录都没有？其实不然，我们回顾 add 函数，其 632 行调用的并不是 uvm_callback#(mii_driver, A)自己的 m_add_tw_cbs 函数，而是调用的 m_t_inst 即 uvm_callback#(mii_driver, uvm_callback)的 m_add_tw_cbs 函数，所以 261 行这里会有一条记录的。

262 行出现了 cb_pair，它的定义位于 234 行。而 super_type 在这里指的是 uvm_callbacks_base，所以 262 行是从 typeid_map 中找出一条记录。typeid_map 中什么时候有了关于 derived_driver 的一条记录？回顾 register_super_type 的 969 行，调用了 get 函数，而这里的 get 函数的 958 行又会调用 uvm_callbacks#(derived_driver, uvm_callback)::get，在这个 get 中将会向 typeid_map 中插入一条索引为 derived_driver，而内容为 uvm_callbacks#(derived_driver, uvm_callback)唯一实例的记录。

263 行会检查 cb_pair 的值是否为 this，即比较一下 uvm_callbacks#(derived_driver, uvm_callback)与 uvm_callbacks#(mii_driver, uvm_callback)是否相同，这里很明显是不

一样的，所以将会执行 264 行，调用 uvm_callbacks#(derived_driver, uvm_callback)的 m_add_tw_cbs 宏，这样做的效果就是把要加入的 my_cb 加入到了 uvm_typed_callbacks#(derived_driver)的 m_tw_cb_q 中。也即是说，假如我们通过 set_super_type 宏把 deriver_driver 和 mii_driver 设置了关联，并且之前通过 register_cb 宏把 A 和 mii_driver 绑定在了一起，那么类似如下的语句也将会在 uvm_typed_callbacks#(derived_driver)的 m_tw_cb_q 中插入一条 my_cb 的记录。

```
A_pool::add(null, my_cb);
```

接下来我们看 uvm_do_callbacks 宏的执行情况：

```
task derived_driver::main_phase();
    …
    `uvm_do_callbacks(mii_driver, A, pre_tran(this, req))

endtask
```

要注意的是这里传入的第一个参数是父类 mii_driver，而不是 derived_driver。与前面分析 uvm_do_callbacks 宏相似，这里会调用 uvm_callbacks#(mii_driver, A)的 get_first 函数，只是传入的 m_obj 是一个 derived_driver 类型的变量，当然了，它在本质上也是一个 mii_driver 类型的变量。

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：get_first

 825    static function CB get_first (ref int itr, input T obj);
 826      uvm_queue#(uvm_callback) q;
 827      CB cb;
 828      void'(get());
 829      m_get_q(q,obj);
 830      for(itr = 0; itr<q.size(); ++itr)
 831        if($cast(cb, q.get(itr)) && cb.callback_mode())
 832          return cb;
 833      return null;
 834    endfunction
```

函数的 829 行调用 m_get_q：

```
文件：src/base/uvm_callback.svh
类：uvm_callbacks#(type T=uvm_object, type CB=uvm_callback)
函数/任务：m_get_q

 799    static function void m_get_q (ref uvm_queue #(uvm_callback) q, input T obj);
 800      if(!m_base_inst.m_pool.exists(obj)) begin //no instance specific
 801        q = (obj == null) ? m_t_inst.m_tw_cb_q : m_t_inst.m_get_tw_cb_q(obj);
 802      end
 803      else begin
 804        q = m_base_inst.m_pool.get(obj);
 805        if(q==null) begin
```

```
806             q=new;
807             m_base_inst.m_pool.add(obj,q);
808           end
809       end
810   endfunction
```

其 801 行将会调用 m_t_inst 即 uvm_callbacks#(mii_driver, uvm_callback)的 m_get_tw_cb_q：

```
文件：src/base/uvm_callback.svh
类：uvm_typed_callbacks#(type T=uvm_object)
函数/任务：m_get_tw_cb_q

208   virtual function uvm_queue#(uvm_callback) m_get_tw_cb_q (uvm_object obj);
209     if(m_am_i_a(obj)) begin
210       foreach(m_derived_types[i]) begin
211         super_type dt;
212         dt = uvm_typeid_base::typeid_map[m_derived_types[i] ];
213         if(dt != null && dt != this) begin
214           m_get_tw_cb_q = dt.m_get_tw_cb_q(obj);
215           if(m_get_tw_cb_q != null)
216             return m_get_tw_cb_q;
217         end
218       end
219       return m_t_inst.m_tw_cb_q;
220     end
221     else
222       return null;
223   endfunction
```

函数的 210 到 218 行是我们前面分析时跳过去的部分。210 行遍历 m_derived_types 数组，这是 uvm_callbacks#(mii_driver, uvm_callback) 的 m_derived_types，因此其中有一条 derived_driver 的记录。

212 找到与 derived_driver 对应的那条记录，214 行调用 uvm_callbacks#(derived_driver, uvm_callback) 的 m_get_tw_cb_q，这会得到 uvm_typed_callbacks#(derived_driver):: m_tw_cb_q。前面刚刚说过，这里有一条 my_cb 的记录，216 行将会返回带有这条记录的 m_tw_cb_q 的指针。my_cb 将最终被 get_first 函数返回，并被 uvm_do_callback 宏的 168 行调用。

附录 A：术语

DUT：Design Under Test

DUV：Design Under Verification

UVM：Universal Verification Methodology

OVM：Open Verification Methdology

VIP：Verification Intellectual Property

TLM：Transaction Level Modeling

参考模型：reference model

附录 B：函数索引