10.2 定位并发错误的技术

之前的章节，我们了解了与并发相关的错误类型，以及如何在代码中体现出来的。这些信息可以帮助我们来判断，的代码中是否存在有隐藏的错误。

最简单直接的就是直接看代码。虽然看起来比较明显，但是要彻底的修复问题，却是很难的。读刚写完的代码，要比读已经存在的代码容易的多。同理，当在审查别人写好的代码时，给出一个通读结果是很容易的，比如：与你自己的代码标准作对比，以及高亮标出显而易见的问题。为什么要花时间来仔细梳理代码？想想之前提到的并发相关的问题——也要考虑非并发问题。(也可以在很久以后做这件事。不过，最后bug依旧存在)我们可以在审阅代码的时候，考虑一些具体的事情，并且发现问题。

即使已经很对代码进行了很详细的审阅，依旧会错过一些bug，这就需要确定一下代码是否做了对应的工作。因此，在测试多线程代码时，会介绍一些代码审阅的技巧。

10.2.1 代码审阅——发现潜在的错误

在审阅多线程代码时，重点要检查与并发相关的错误。如果可能，可以让同事/同伴来审阅。因为不是他们写的代码，他们将会考虑这段代码是怎么工作的，就可能会覆盖到一些你没有想到的情况，从而找出一些潜在的错误。审阅人员需要有时间去做审阅——并非在休闲时间简单的扫一眼。大多数并发问题需要的不仅仅是一次快速浏览——通常需要在找到问题上花费很多时间。

如果你让你的同时来审阅代码，他/她肯定对你的代码不是很熟悉。因此，他/她会从不同的角度来看你的代码，然后指出你没有注意的事情。如果你的同事都没有空，你可以叫你的朋友，或传到网络上，让网友审阅(注意，别传一些机密代码上去)。实在没有人审阅你的代码，不要着急——你还可以做很多事情。对于初学者，可以将代码放置一段时间——先去做应用的另外的部分，或是阅读一本书籍，亦或出去溜达溜达。在休息之后，当再集中注意力做某些事情(潜意识会考虑很多问题)。同样，当你做完其他事情，回头再看这段代码，就会有些陌生——你可能会从另一个角度来看你自己以前写的代码。

另一种方式就是自己审阅。可以向别人详细的介绍你所写的功能，可能并不是一个真正的人——可能要对一只玩具熊或一只橡皮鸡来进行解释，并且我个人觉得写一些比较详细的注释是非常有益的。在解释过程中，会考虑每一行过后，会发生什么事情，有哪些数据被访问了，等等。问自己

关于代码的问题，并且向自己解释这些问题。我觉得这是种非常有效的技巧——通过自问自答，对每个问题认真考虑，这些问题往往都会揭示一些问题，也会有益于任何形式的代码审阅。

审阅多线程代码需要考虑的问题

审阅代码的时候考虑和代码相关的问题，以及有利于找出代码中的问题。问题审阅者需要在代码中找到相应的回答或错误。我认为下面这些问题必须要问(当然，不是一个综合性的列表)，你也可以找一些其他问题来帮助你找到代码的问题。

这里，列一下我的清单：

- 并发访问时，那些数据需要保护？
- 如何确定访问数据受到了保护？
- 是否会有多个线程同时访问这段代码？
- 这个线程获取了哪个互斥量？
- 其他线程可能获取哪些互斥量？
- 两个线程间的操作是否有依赖关系？如何满足这种关系？
- 这个线程加载的数据还是合法数据吗？数据是否被其他线程修改过？
- 当假设其他线程可以对数据进行修改，这将意味着什么？并且，怎么确保这样的事情不会发生？

最后一个问题，我最喜欢，因为它让我着实的去考虑线程之间的关系。通过假设一个bug和一行代码相关联，你就可以扮演侦探来追踪bug出现的原因。为了让你自己确定代码里面没有bug，需要考虑代码运行的各种情况。在数据被多个互斥量所保护的时候，这种方式尤其有用，比如：使用线程安全队列(第6章)，可以对队头和队尾使用独立的互斥量:就是为了确保在持有一个互斥量的时候，访问是安全的，这里必须确保持有其他互斥量的线程不能同时访问同一元素。还需要特别关注的是，对公共数据的显式处理，使用一个指针或引用的方式让其他代码来获取数据。

倒数第二个问题也很重要，因为这是很容易产生错误的地方：先释放再获取一个互斥量的前提是，其他线程可能会修改共享数据。虽然很明显，但当互斥锁不是立即可见——可能因为是内部对象——就会不知不觉的掉入陷阱中。在第6章，已经了解到这种情况是怎么引起条件竞争，以及如何给细粒度线程安全数据结构带来麻烦。不过，非线程安全栈将top()和pop()操作分开是有意义的，当多线程会并发的访问这个栈，问题会马上出现，因为在两个操作的调用间，内部互斥锁已经被释放，并且另一个线程对栈进行了修改。解决方案就是将两个操作合并，就能用同一个锁来对操作的执行进行保护，就消除了条件竞争的问题。

OK，你已经审阅过代码了(或者让别人看过)。现在，你确信代码没有问题。

就像需要用味觉来证明，你现在吃的东西——怎么测试才能确认你的代码没有bug呢？

10.2.2 通过测试定位并发相关的错误

写单线程应用时，如果时间充足，测试起来相对简单。原则上，设置各种可能的输入(或设置成感兴趣的情况)，然后执行应用。如果应用行为和输出正确，就能判断其能对给定输入集给出正确的答案。检查错误状态(比如：处理磁盘满载错误)就会比处理可输入测试复杂的多，不过原理是一样的——设置初始条件，然后让程序执行。

测试多线程代码的难度就要比单线程大好几个数量级，因为不确定是线程的调度情况。因此，即使使用测试单线程的输入数据，如果有条件变量潜藏在代码中，那么代码的结果可能会时对时错。只是因为条件变量可能会在有些时候，等待其他事情，从而导致结果错误或正确。

因为与并发相关的bug相当难判断，所以在设计并发代码时需要格外谨慎。设计的时候，每段代码都需要进行测试，以保证没有问题，这样才能在测试出现问题的时候，剔除并发相关的bug——例如，对队列的push和pop，分别进行并发的测试，就要好于直接使用队列测试其中全部功能。这种思想能帮你在设计代码的时候，考虑什么样的代码是可以用来测试正在设计的这个结构——本章后续章节中看到与设计测试代码相关的内容。

测试的目的就是为了消除与并发相关的问题。如果在单线程测试的时候，遇到了问题，那这个问题就是普通的bug，而非并发相关的bug。当问题发生在*未测试区域*(in the wild)，也就是没有在测试范围之内，像这样的情况就要特别注意。bug出现在应用的多线程部分，并不意味着该问题是一个多线程相关的bug。使用线程池管理某一级并发的时候，通常会有一个可配置的参数，用来指定工作线程的数量。当手动管理线程时，就需要将代码改成单线程的方式进行测试。不管哪种方式，将多线程简化为单线程后，就能将与多线程相关的bug排除掉。反过来说，当问题在单芯系统中消失(即使还是以多线程方式)，不过问题在多芯系统或多核系统中出现，就能确定你被多线程相关的bug坑了，可能是条件变量的问题，还有可能是同步或内存序的问题。

测试并发的代码很多，不过通过测试的代码结构就没那么多了；对结构的测试也很重要，就像对环境的测试一样。

如果你依旧将测试并发队列当做一个测试例，你就需要考虑这些情况：

- 使用单线程调用push()或pop()，来确定在一般情况下队列是否工作正常
- 其他线程调用pop()时，使用另一线程在空队列上调用push()
- 在空队列上，以多线程的方式调用push()
- 在满载队列上，以多线程的方式调用push()
- 在空队列上，以多线程的方式调用pop()
- 在满载队列上，以多线程的方式调用pop()
- 在非满载队列上(任务数量小于线程数量)，以多线程的方式调用pop()
- 当一线程在空队列上调用pop()的同时，以多线程的方式调用push()
- 当一线程在满载队列上调用pop()的同时，以多线程的方式调用push()

- 当多线程在空队列上调用pop()的同时，以多线程方式调用push()
- 当多线程在满载队列上调用pop()的同时，以多线程方式调用push()

这是我所能想到的场景，可能还有更多，之后你需要考虑测试环境的因素：

- “多线程”是有多少个线程(3个，4个，还是1024个？)
- 系统中是否有足够的处理器，能让每个线程运行在属于自己的处理器上
- 测试需要运行在哪种处理器架构上
- 在测试中如何对“同时”进行合理的安排

这些因素的考虑会具体到一些特殊情况。四个因素都需要考虑，第一个和最后一个会影响测试结构本身(在10.2.5节中会介绍)，另外两个就和实际的物理测试环境相关了。使用线程数量相关的测试代码需要独立测试，可通过很多结构化测试获得最合适的调度方式。在了解这些技巧前，先来了解一下如何让你的应用更容易测试。

10.2.3 可测试性设计

测试多线程代码很困难，所以你需要将其变得简单一些。很重要的一件事就是，在设计代码时，考虑其的可测试性。可测试的单线程代码设计已经说烂了，而且其中许多建议，在现在依旧适用。通常，如果代码满足一下几点，就很容易进行测试：

- 每个函数和类的关系都很清楚。
- 函数短小精悍。
- 测试用例可以完全控制被测试代码周边的环境。
- 执行特定操作的代码应该集中测试，而非分布式测试。
- 需要在完成编写后，考虑如何进行测试。

以上这些在多线程代码中依旧适用。实际上，我会认为对多线程代码的可测试性要比单线程的更为重要，因为多线程的情况更加复杂。最后一个因素尤为重要：即使不在写完代码后，去写测试用例，这也是一个很好的建议，能让你在写代码之前，想想应该怎么去测试它——用什么作为输入，什么情况看起来会让结果变得糟糕，以及如何激发代码中潜在的问题，等等。

并发代码测试的一种最好的方式：去并发化测试。如果代码在线程间的通讯路径上出现问，就可以让一个已通讯的单线程进行执行，这样会减小问题的难度。在对数据进行访问的应用进行测试时，可以使用单线程的方式进行。这样线程通讯和对特定数据块进行访问时只有一个线程，就达到了更容易测试的目的。

例如，当应用设计为一个多线程状态机时，可以将其分为若干块。将每个逻辑状态分开，就能保证对于每个可能的输入事件、转换或其他操作的结果是正确的；这就是使用了单线程测试的技巧，测试用例提供的输入事件将来自于其他线程。之后，核心状态机和消息路由的代码，就能保证时间能以正确的顺序，正确的传递给可单独测试的线程上，不过对于多并发线程，需要为测试专门设计简单的逻辑状态。

或者，如果将代码分割成多个块(比如：读共享数据/变换数据/更新共享数据)，就能使用单线程来测试变换数据的部分。麻烦的多线程测试问题，转换成单线程测试读和更新共享数据，就会简单许多。

一件事需要小心，就是某些库会用其内部变量存储状态，当多线程使用同一库中的函数，这个状态就会被共享。这的确是一个问题，并且这个问题不会马上出现在访问共享数据的代码中。不过，随着你对这个库的熟悉，就会清楚这样的情况会在什么时候出现。之后，可以适当的加一些保护和同步，或使用B计划——让多线程安全并发访问的功能。

将并发代码设计的有更好的测试性，要比以代码分块的方式处理并发相关的问题好很多。当然，还要注意对非线程安全库的调用。10.2.1节中那些问题，也需要在审阅自己代码的时候格外注意。虽然，这些问题和测试(可测试性)没有直接的关系，但带上“测试帽子”时候，就要考虑这些问题了，并且还要考虑如何测试已写好的代码，这就会影响设计方向的选择，也会让测试做的更加容易一些。

我们已经了解了如何能让测试变得更加简单，以及将代码分成一些“并发”块(比如，线程安全容器或事件逻辑状态机)以“单线程”的形式(可能还通过并发块和其他线程进行互动)进行测试。

下面就让我们了解一下测试多线程代码的技术。

10.2.4 多线程测试技术

想通过一些技巧写一些较短的代码，来对函数进行测试，比如：如何处理调度序列上的bug？

这里的确有几个方法能进行测试，让我们从蛮力测试(或称压力测试)开始。

蛮力测试

代码有问题的时候，就要求蛮力测试一定能看到这个错误。这就意味着代码要运行很多遍，可能会有很多线程在同一时间运行。要是有bug出现，只能线程出现特殊调度的时候；代码运行次数的增加，就意味着bug出现的次数会增多。当有几次代码测试通过，你可能会对代码的正确性有一些

信心。如果连续运行10次都通过，你就会更有信心。如果你运行十亿次都通过了，那么你就会认为这段代码没有问题了。

自信的来源是每次测试的结果。如果你的测试粒度很细，就像测试之前的线程安全队列，那么蛮力测试会让你对这段代码持有高度的自信。另一方面，当测试对象体积较大的时候，调度序列将会很长，即使运行了十亿次测试用例，也不让你对这段代码产生什么信心。

蛮力测试的缺点就是，可能会误导你。如果写出来的测试用例就为了不让有问题的情况发生，那么怎么运行，测试都不会失败，可能会因环境的原因，出现几次失败的情况。最糟糕的情况就是，问题不会出现在你的测试系统中，因为在某些特殊的系统中，这段代码就会出现问题。除非代码运行在与测试机系统相同的系统中，不过特殊的硬件和操作系统的因素结合起来，可能就会让运行环境与测试环境有所不同，问题可能就会随之出现。

这里有一个经典的案例，在单处理器系统上测试多线程应用。因为每个线程都在同一个处理器上运行，任何事情都是串行的，并且还有很多条件竞争和乒乓缓存，这些问题可能在真正的多处理器系统中，根本不会出现。还有其他变数：不同处理器架构提供不同的的同步和内存序机制。比如，在x86和x86-64架构上，原子加载操作通常是相同的，无论是使用memory_order_relaxed，还是memory_order_seq_cst(详见5.3.3节)。这就意味着在x86架构上使用松散内存序没有问题，但在有更精细的内存序指令集的架构(比如：SPARC)下，这样使用就可能产生错误。

如果你希望你的应用能跨平台使用，就要在相关的平台上进行测试。这就是我把处理器架构也列在测试需要考虑的清单中的原因(详见10.2.2)。

要避免误导的产生，关键点在于成功的蛮力测试。这就需要进行仔细考虑和设计，不仅仅是选择相关单元测试，还要遵守测试系统设计准则，以及选定测试环境。保证代码分支被尽可能的测试到，尽可能多的测试线程间的互相作用。还有，需要知道哪部分被测试覆盖到，哪些没有覆盖。

虽然，蛮力测试能够给你一些信心，不过其不保证能找到所有的问题。如果有时间将下面的技术应用到你的代码或软件中，就能保证所有的问题都能被找到。

组合仿真测试

名字比较口语化，我需要解释一下这个测试是什么意思：使用一种特殊的软件，用来模拟代码运行的真实情况。你应该知道这种软件，能让一台物理机上运行多个虚拟环境或系统环境，而硬件环境则由监控软件来完成。除了环境是模拟的以外，模拟软件会记录对数据序列访问，上锁，以及对每个线程的原子操作。然后使用C++内存模型的规则，重复的运行，从而识别条件竞争和死锁。

虽然，这种组合测试可以保证所有与系统相关的问题都会被找到，不过过于零碎的程序将会在这种测试中耗费太长时间，因为组合数目和执行的操作数量将会随线程的增多呈指数增长态势。这

个测试最好留给需要细粒度测试的代码段，而非整个应用。另一个缺点就是，代码对操作的处理，往往会依赖与模拟软件的可用性。

所以，测试需要在正常情况下，运行很多次，不过这样可能会错过一些问题；也可以在一些特殊情况下运行多次，不过这样更像是为了验证某些问题。

还有其他的测试选项吗？

第三个选项就是使用一个库，在运行测试的时候，检查代码中的问题。

使用专用库对代码进行测试

虽然，这个选择不会像组合仿真的方式提供彻底的检查，不过可以通过特别实现的库(使用同步原语)来发现一些问题，比如：互斥量，锁和条件变量。例如，访问某块公共数据的时候，就要将指定的互斥量上锁。数据被访问后，发现一些互斥量已经上锁，就需要确定相关的互斥量是否被访问线程锁住；如果没有，测试库将报告这个错误。当需要测试库对某块代码进行检查时，可以对对应的共享数据进行标记。

当不止一个互斥量同时被一个线程持有，测试库也会对锁的序列进行记录。如果其他线程以不同的顺序进行上锁，即使在运行的时候测试用例没有发生死锁，测试库都会将这个行为记录为“有潜在死锁”可能。

当测试多线程代码的时候，另一种库可能会用到，以线程原语实现的库，比如：互斥量和条件变量；当多线程代码在等待，或是被条件变量通过notify_one()提醒的某个线程，测试者可以通过线程，获取到锁。就可以让你来安排一些特殊的情况，以验证代码是否会在这些特定的环境下产生期望的结果。

C++标准库实现中，某些测试工具已经存在于标准库中，没有实现的测试工具，可以基于标准库进行实现。

了解完各种运行测试代码的方式，将让我们来了解一下，如何以你想要的调度方式来构建代码。

10.2.5 构建多线程测试代码

10.2.2节中提过，需要找一种合适的调度方式来处理测试中“同时”的部分，现在就是解决这个问题的时候。

在特定时间内，你需要安排一系列线程，同时去执行指定的代码段。最简单的情况：两个线程的情况，就很容易扩展到多个线程。

首先，你需要知道每个测试的不同之处：

- 环境布置代码，必须首先执行
- 线程设置代码，需要在每个线程上执行
- 线程上执行的代码，需要有并发性
- 在并发执行结束后，后续代码需要对代码的状态进行断言检查

这几条后面再解释，先让我们考虑一下10.2.2节中的一个特殊的情况：一个线程在空队列上调用push()，同时让其他线程调用pop()。

通常，布置环境的代码比较简单：创建队列即可。线程在执行pop()的时候，没有线程设置代码。线程设置代码是在执行push()操作的线程上进行的，其依赖与队列的接口和对象的存储类型。如果存储的对象需要很大的开销才能构建，或必须在堆上分配的对象，那么最好在线程设置代码中进行构建或分配；这样，就不会影响到测试结果。另外，如果队列中只存简单的int类型对象，构建int对象时就不会有太多额外的开销。实际上，已测试代码相对简单——一个线程调用push()，另一个线程调用pop()——那么，“完成后”的代码到底是什么样子呢？

在这个例子中，pop()具体做的事情，会直接影响“完成后”代码。如果有数据块，返回的肯定就是数据了，push()操作就成功的向队列中推送了一块数据，并在在数据返回后，队列依旧是空的。如果pop()没有返回数据块，也就是队列为空的情况下，操作也能执行，这样就需要两个方向的测试：要不pop()返回push()推送到队列中的数据块，之后队列依旧为空；要不pop()会示意队列中没有元素，但同时push()向队列推送了一个数据块。这两种情况都是真实存在的；你需要避免的情况是：pop()示意队列中没有数据的同时，队列还是空的，或pop()返回数据块的同时，队列中还有数据块。为了简化测试，可以假设pop()是可阻塞的。在最终代码中，需要用断言判断弹出的数据与推入的数据，还要判断队列为空。

现在，了解了各个代码块，就需要保证所有事情按计划进行。一种方式是使用一组 `std::promise` 来表示就绪状态。每个线程使用一个promise来表示是否准备好，然后让 `std::promise` 等待(复制)一个 `std::shared_future` ;主线程会等待每个线程上的promise设置后，才按下“开始”键。这就能保证每个线程能够同时开始，并且在准备代码执行完成后，并发代码就可以开始执行了；任何的线程特定设置都需要在设置线程的promise前完成。最终，主线程会等待所有线程完成，并且检查其最终状态。还需要格外关心的是——异常，所有线程在准备好的情况下，才按下“开始”键；否则，未准备好的线程就不会运行。

下面的代码，构建了这样的测试。

清单10.1 对一个队列并发调用push()和pop()的测试用例

```
void test_concurrent_push_and_pop_on_empty_queue()
{
  threadsafe_queue<int> q;  // 1

  std::promise<void> go,push_ready,pop_ready;  // 2
  std::shared_future<void> ready(go.get_future());  // 3

  std::future<void> push_done;  // 4
  std::future<int> pop_done;

  try
  {
    push_done=std::async(std::launch::async,  // 5
                         [&q,ready,&push_ready]()
                         {
                           push_ready.set_value();
                           ready.wait();
                           q.push(42);
                         }
      );
    pop_done=std::async(std::launch::async,  // 6
                        [&q,ready,&pop_ready]()
                        {
                          pop_ready.set_value();
                          ready.wait();
                          return q.pop();  // 7
                        }
      );
    push_ready.get_future().wait();  // 8
    pop_ready.get_future().wait();
    go.set_value();  // 9

    push_done.get();  // 10
    assert(pop_done.get()==42);  // 11
    assert(q.empty());
  }
  catch(...)
  {
    go.set_value();  // 12
    throw;
  }
}
```

首先，环境设置代码中创建了空队列①。然后，为准备状态创建promise对象②，并且为go信号获取一个 `std::shared_future` 对象③。再后，创建了future用来表示线程是否结束④。这些都需要放在try块外面，再设置go信号时抛出异常，就不需要等待其他城市线程完成任务了(这将会产生死锁——如果测试代码产生死锁，测试代码就是不理想的代码)。

try块中可以启动线程⑤⑥——使用 `std::launch::async` 保证每个任务在自己的线程上完成。注意，使用 `std::async` 会让你任务更容易成为线程安全的任务；这里不用普通 `std::thread` ，因为其析构函数会对future进行线程汇入。lambda函数会捕捉指定的任务(会在队列中引用)，并且为promise准备相关的信号，同时对从go中获取的ready做一份拷贝。

如之前所说，每个任务集都有自己的ready信号，并且会在执行测试代码前，等待所有的ready信号。而主线程不同——等待所有线程的信号前⑧，提示所有线程可以开始进行测试了⑨。

最终，异步调用等待线程完成后⑩⑪，主线程会从中获取future，再调用get()成员函数获取结果，最后对结果进行检查。注意，这里pop操作通过future返回检索值⑦，所以能获取最终的结果⑪。

当有异常抛出，需要通过对go信号的设置来避免悬空指针的产生，再重新抛出异常⑫。future与之后声明的任务相对应④，所以future将会被首先销毁，如果future都没有就绪，析构函数将会等待相关任务完成后执行操作。

虽然，像是使用测试模板对两个调用进行测试，但使用类似的东西是必要的，这样会便于测试的进行。例如，启动一个线程就是一个很耗时的过程，如果没有线程在等待go信号时，推送线程可能会在弹出线程开始之前，就已经完成了；这样就失去了测试的作用。以这种方式使用future，就是为了保证线程都在运行，并且阻塞在同一个future上。future解除阻塞后，将会让所有线程运行起来。当你熟悉了这个结构，其就能以同样的模式创建新的测试用例。测试两个以上的线程，这种模式很容易进行扩展。

目前，我们已经了解了多线程代码的正确性测试。

虽然这是最最重要的问题，但是其不是我们做测试的唯一原因：多线程性能的测试同样重要。

下面就让我们来了解一下性能测试。

10.2.6 测试多线程代码性能

选择以并发的方式开发应用，就是为了能够使用日益增长的处理器数量；通过处理器数量的增加，来提升应用的执行效率。因此，确定性能是否有真正的提高就很重要了(就像其他优化一样)。

并发效率中有个特别的问题——可扩展性——你希望代码能很快的运行24次，或在24芯的机器上对数据进行24(或更多)次处理，或其他等价情况。你不会希望，你的代码运行两次的数据和在双芯机器上执行一样快的同时，在24芯的机器上会更慢。如8.4.2节中所述，当有重要的代码以单线程方式运行时，就会限制性能的提高。因此，在做测试之前，回顾一下代码的设计结构是很有必要的；这样就能判断，代码在24芯的机器上时，性能会不会提高24倍，或是因为有串行部分的存在，最大的加速比只有3。

在对数据访问的时候，处理器之间会有竞争，会对性能有很大的影响。需要合理的权衡性能和处理器的数量，处理器数量太少，就会等待很久；处理器过多，又会因为竞争的原因等待很久。

因此，在对应的系统上通过不同的配置，检查多线程的性能就很有必要，这样可以得到一张性能伸缩图。最起码，(如果条件允许)你应该在一个单处理器的系统上和一个多处理核芯的系统上进行测试。