3.3 保护共享数据的替代设施

互斥量是最通用的机制，但其并非保护共享数据的唯一方式。这里有很多替代方式可以在特定情况下，提供更加合适的保护。

一个特别极端(但十分常见)的情况就是，共享数据在并发访问和初始化时(都需要保护)，但是之后需要进行隐式同步。这可能是因为数据作为只读方式创建，所以没有同步问题；或者因为必要的保护作为对数据操作的一部分，所以隐式的执行。任何情况下，数据初始化后锁住一个互斥量，纯粹是为了保护其初始化过程(这是没有必要的)，并且这会给性能带来不必要的冲击。出于以上的原因， C++ 标准提供了一种纯粹保护共享数据初始化过程的机制。

3.3.1 保护共享数据的初始化过程

假设你与一个共享源，构建代价很昂贵,可能它会打开一个数据库连接或分配出很多的内存。

延迟初始化(Lazy initialization)在单线程代码很常见——每一个操作都需要先对源进行检查，为了了解数据是否被初始化，然后在其使用前决定，数据是否需要初始化：

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
{
  if(!resource_ptr)
  {
    resource_ptr.reset(new some_resource);  // 1
  }
  resource_ptr->do_something();
}
```

当共享数据对于并发访问是安全的，①是转为多线程代码时，需要保护的，但是下面天真的转换会使得线程资源产生不必要的序列化。这是因为每个线程必须等待互斥量，为了确定数据源已经初始化了。

清单 3.11 使用一个互斥量的延迟初始化(线程安全)过程

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;

void foo()
{
  std::unique_lock<std::mutex> lk(resource_mutex);  // 所有线程在此序列化
  if(!resource_ptr)
  {
    resource_ptr.reset(new some_resource);  // 只有初始化过程需要保护
  }
  lk.unlock();
  resource_ptr->do_something();
}
```

这段代码相当常见了，也足够表现出没必要的线程化问题，很多人能想出更好的一些的办法来做这件事，包括声名狼藉的双重检查锁模式：

```
void undefined_behaviour_with_double_checked_locking()
{
  if(!resource_ptr)  // 1
  {
    std::lock_guard<std::mutex> lk(resource_mutex);
    if(!resource_ptr)  // 2
    {
      resource_ptr.reset(new some_resource);  // 3
    }
  }
  resource_ptr->do_something();  // 4
}
```

指针第一次读取数据不需要获取锁①，并且只有在指针为NULL时才需要获取锁。然后，当获取锁之后，指针会被再次检查一遍② (这就是双重检查的部分)，避免另一的线程在第一次检查后再做初始化，并且让当前线程获取锁。

这个模式为什么声名狼藉呢？因为这里有潜在的条件竞争，未被锁保护的读取操作①没有与其他线程里被锁保护的写入操作③进行同步。因此就会产生条件竞争，这个条件竞争不仅覆盖指针本身，还会影响到其指向的对象；即使一个线程知道另一个线程完成对指针进行写入，它可能没有看到新创建的some_resource实例，然后调用do_something()④后，得到不正确的结果。这个例子是在一种典型的条件竞争——数据竞争， C++ 标准中这就会被指定为“未定义行为”。这种竞争肯定是可以避免的。可以阅读第5章，那里有更多对内存模型的讨论，包括数据竞争的构成。

C++标准委员会也认为条件竞争的处理很重要，所以C++标准库提供了 `std::once_flag` 和 `std::call_once` 来处理这种情况。比起锁住互斥量，并显式的检查指针，每个线程只需要使用 `std::call_once` ，在 `std::call_once` 的结束时，就能安全的知道指针已经被其他的线程初始化了。使用 `std::call_once` 比显式使用互斥量消耗的资源更少，特别是当初始化完成后。下面的例子展示了与清单3.11中的同样的操作，这里使用了 `std::call_once` 。在这种情况下，初始化通过调用函数完成，同样这样操作使用类中的函数操作符来实现同样很简单。如同大多数在标准库中的函数一样，或作为函数被调用，或作为参数被传递， `std::call_once` 可以和任何函数或可调用对象一起使用。

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;  // 1

void init_resource()
{
  resource_ptr.reset(new some_resource);
}

void foo()
{
  std::call_once(resource_flag,init_resource);  // 可以完整的进行一次初始化
  resource_ptr->do_something();
}
```

在这个例子中， `std::once_flag` ①和初始化好的数据都是命名空间区域的对象，但是 `std::call_once()` 可仅作为延迟初始化的类型成员，如同下面的例子一样：

清单3.12 使用 `std::call_once` 作为类成员的延迟初始化(线程安全)

```
class X
{
private:
  connection_info connection_details;
  connection_handle connection;
  std::once_flag connection_init_flag;

  void open_connection()
  {
    connection=connection_manager.open(connection_details);
  }
public:
  X(connection_info const& connection_details_):
```

```
      connection_details(connection_details_)
  {}
  void send_data(data_packet const& data)  // 1
  {
    std::call_once(connection_init_flag,&X::open_connection,this);  // 2
    connection.send_data(data);
  }
  data_packet receive_data()  // 3
  {
    std::call_once(connection_init_flag,&X::open_connection,this);  // 2
    return connection.receive_data();
  }
};
```

例子中第一个调用send_data()①或receive_data()③的线程完成初始化过程。使用成员函数open_connection()去初始化数据，也需要将this指针传进去。和其在在标准库中的函数一样，其接受可调用对象，比如 `std::thread` 的构造函数和 `std::bind()` ，通过向 `std::call_once()` ②传递一个额外的参数来完成这个操作。

值得注意的是， `std::mutex` 和 `std::one_flag` 的实例就不能拷贝和移动，所以当你使用它们作为类成员函数，如果你需要用到他们，你就得显示定义这些特殊的成员函数。

还有一种情形的初始化过程中潜存着条件竞争：其中一个局部变量被声明为static类型。这种变量的在声明后就已经完成初始化；对于多线程调用的函数，这就意味着这里有条件竞争——抢着去定义这个变量。在很多在前C++11编译器(译者：不支持C++11标准的编译器)，在实践过程中，这样的条件竞争是确实存在的，因为在多线程中，每个线程都认为他们是第一个初始化这个变量线程；或一个线程对变量进行初始化，而另外一个线程要使用这个变量时，初始化过程还没完成。在C++11标准中，这些问题都被解决了：初始化及定义完全在一个线程中发生，并且没有其他线程可在初始化完成前对其进行处理，条件竞争终止于初始化阶段，这样比在之后再去处理好的多。在只需要一个全局实例情况下，这里提供一个 `std::call_once` 的替代方案

```
class my_class;
my_class& get_my_class_instance()
{
  static my_class instance;  // 线程安全的初始化过程
  return instance;
}
```

多线程可以安全的调用get_my_class_instance()①函数，不用为数据竞争而担心。

对于很少有更新的数据结构来说，只在初始化时保护数据。在大多数情况下，这种数据结构是只读的，并且多线程对其并发的读取也是很愉快的，不过一旦数据结构需要更新，就会产生竞争。

3.3.2 保护很少更新的数据结构

试想，为了将域名解析为其相关IP地址，我们在缓存中的存放了一张DNS入口表。通常，给定DNS数目在很长的一段时间内保持不变。虽然，在用户访问不同网站时，新的入口可能会被添加到表中，但是这些数据可能在其生命周期内保持不变。所以定期检查缓存中入口的有效性，就变的十分重要了；但是，这也需要一次更新，也许这次更新只是对一些细节做了改动。

虽然更新频度很低，但更新也有可能发生，并且当这个可缓存被多个线程访问，这个缓存就需要处于更新状态时得到保护，这也为了确保每个线程读到都是有效数据。

没有使用专用数据结构时，这种方式是符合预期，并且为并发更新和读取特别设计的(更多的例子在第6和第7章中介绍)。这样的更新要求线程独占数据结构的访问权，直到其完成更新操作。当更新完成，数据结构对于并发多线程访问又会是安全的。使用 `std::mutex` 来保护数据结构，显的有些反应过度(因为在没有发生修改时，它将削减并发读取数据的可能性)。这里需要另一种不同的互斥量，这种互斥量常被称为“读者-作者锁”，因为其允许两种不同的使用方式：一个“作者”线程独占访问和共享访问，让多个“读者”线程并发访问。

虽然这样互斥量的标准提案已经交给标准委员会，但是 `C++` 标准库依旧不会提供这样的互斥量[3]。因为建议没有被采纳，这个例子在本节中使用的是Boost库提供的实现(Boost采纳了这个建议)。你将在第8章中看到，这种锁的也不能包治百病，其性能依赖于参与其中的处理器数量，同样也与读者和作者线程的负载有关。为了确保增加复杂度后还能获得性能收益，目标系统上的代码性能就很重要。

比起使用 `std::mutex` 实例进行同步，不如使用 `boost::shared_mutex` 来做同步。对于更新操作，可以使用 `std::lock_guard<boost::shared_mutex>` 和 `std::unique_lock<boost::shared_mutex>` 上锁。作为 `std::mutex` 的替代方案，与 `std::mutex` 所做的一样，这就能保证更新线程的独占访问。因为其他线程不需要去修改数据结构，所以其可以使用 `boost::shared_lock<boost::shared_mutex>` 获取访问权。这与使用 `std::unique_lock` 一样，除非多线程要在同时获取同一个 `boost::shared_mutex` 上有共享锁。唯一的限制：当任一线程拥有一个共享锁时，这个线程就会尝试获取一个独占锁，直到其他线程放弃他们的锁；同样的，当任一线程拥有一个独占锁时，其他线程就无法获得共享锁或独占锁，直到第一个线程放弃其拥有的锁。

如同之前描述的那样，下面的代码清单展示了一个简单的DNS缓存，使用 `std::map` 持有缓存数据，使用 `boost::shared_mutex` 进行保护。

清单3.13 使用 `boost::shared_mutex` 对数据结构进行保护

```
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>

class dns_entry;

class dns_cache
{
  std::map<std::string,dns_entry> entries;
  mutable boost::shared_mutex entry_mutex;
public:
  dns_entry find_entry(std::string const& domain) const
  {
    boost::shared_lock<boost::shared_mutex> lk(entry_mutex);  // 1
    std::map<std::string,dns_entry>::const_iterator const it=
       entries.find(domain);
    return (it==entries.end())?dns_entry():it->second;
  }
  void update_or_add_entry(std::string const& domain,
                           dns_entry const& dns_details)
  {
    std::lock_guard<boost::shared_mutex> lk(entry_mutex);  // 2
    entries[domain]=dns_details;
  }
};
```

清单3.13中，find_entry()使用 `boost::shared_lock<>` 来保护共享和只读权限①；这就使得多线程可以同时调用find_entry()，且不会出错。另一方面，update_or_add_entry()使用 `std::lock_guard<>` 实例，当表格需要更新时②，为其提供独占访问权限；update_or_add_entry()函数调用时，独占锁会阻止其他线程对数据结构进行修改，并且阻止线程调用find_entry()。

3.3.3 嵌套锁

当一个线程已经获取一个 `std::mutex` 时(已经上锁)，并对其再次上锁，这个操作就是错误的，并且继续尝试这样做的话，就会产生未定义行为。然而，在某些情况下，一个线程尝试获取同一个互斥量多次，而没有对其进行一次释放是可以的。之所以可以，是因为 `C++` 标准库提供了 `std::recursive_mutex` 类。其功能与 `std::mutex` 类似，除了你可以从同一线程的单个实例上获取多个锁。互斥量锁住其他线程前，你必须释放你拥有的所有锁，所以当你调用lock()三次时，你也必须调用unlock()三次。正确使用 `std::lock_guard<std::recursive_mutex>` 和 `std::unique_lock<std::recursive_mutex>` 可以帮你处理这些问题。

大多数情况下，当你需要嵌套锁时，就要对你的设计进行改动。嵌套锁一般用在可并发访问的类上，所以其拥互斥量保护其成员数据。每个公共成员函数都会对互斥量上锁，然后完成对应的功能，之后再解锁互斥量。不过，有时成员函数会调用另一个成员函数，这种情况下，第二个成员函数也会试图锁住互斥量，这就会导致未定义行为的发生。“变通的”解决方案会将互斥量转为嵌套锁，第二个成员函数就能成功的进行上锁，并且函数能继续执行。

但是，这样的使用方式是不推荐的，因为其过于草率，并且不合理。特别是，当锁被持有时，对应类的不变量通常正在被修改。这意味着，当不变量正在改变的时候，第二个成员函数还需要继续执行。一个比较好的方式是，从中提取出一个函数作为类的私有成员，并且让其他成员函数都对其进行调用，这个私有成员函数不会对互斥量进行上锁(在调用前必须获得锁)。然后，你仔细考虑一下，在这种情况调用新函数时，数据的状态。

[3] Howard E. Hinnant, “Multithreading API for C++0X—A Layered Approach,” C++ Standards Committee Paper N2094, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html.