C++和双重检查锁定模式(DCLP)的风险 _nodeathphoenix的专栏-CSDN博客

转自：

http://blog.jobbole.com/86392/

多线程其实就是指两个任务一前一后或者同时发生。

1 简介

当你在网上搜索设计模式的相关资料时，你一定会找到最常被提及的一个模式：单例模式(Singleton)。然而，当你尝试在项目中使用单例模式时，一定会遇到一个很重要的限制：若使用传统的实现方法(我们会在下文解释如何实现)，单例模式是非线程安全的。

程序员们为了解决这一问题付出了很多努力，其中最流行的一种解决方法是使用一个新的设计模式：双重检查锁定模式(DCLP)[13, 14]。设计DCLP的目的在于在共享资源(如单例)初始化时添加有效的线程安全检查功能。但DCLP也存在一个问题：它是不可靠的。此外，在本质上不改变传统设计模式实现的基础上，几乎找不到一种简便方法能够使DCLP在C/C++程序中变得可靠。更有趣的是，DCLP无论在单处理器还是多处理器架构中，都可能由不同的原因导致失效。

本文将为大家解释以下几个问题：

1. 为什么单例模式是非线程安全的？
2. DCLP是如何处理这个问题的？
3. 为什么DCLP在单处理器和多处理器架构下都可能失效？
4. 为什么我们很难为这个问题找到简便的解决办法？

在这个过程中，我们将澄清以下四个概念之间的关系：语句在源代码中的顺序、序列点(sequence points, 译者注[1])、 编译器和硬件优化，以及语句的实际执行顺序。最后，我们会总结一些关于如何给单例模式(或其他相似设计)添加线程安全机制的建议，使你的代码变得既可靠又高效。

2 单例模式与多线程

单例模式[7]传统的实现方法是，当对象第一次使用时将其实例化，并用一个指针指向该对象，实现代码如下：

```
1  // from the header file 以下代码来自头文件
2  class Singleton {
3  public :
4  static Singleton * instance ( ) ;
5  . . .
6  private :
7  static Singleton * pInstance ;
8  } ;
```

```
9
10 // from the implementation file 以下代码来自实现文件
11 Singleton * Singleton :: pInstance = 0 ;
12 Singleton * Singleton :: instance ( ) {
13 if ( pInstance == 0 ) {
14 pInstance = new Singleton ;
15 }
16 return pInstance ;
17 }
18
```

在单线程环境下，虽然中断会引起某些问题，但大体上这段代码可以运行得很好。如果代码运行到Singleton::instance()内部时发生中断，而中断处理程序调用的也是Singleton::instance()，可以想象你将会遇到什么样的麻烦。因此，如果撇开中断不考虑，那么这个实现在单线程环境下可以运行得很好。

很不幸，这个实现在多线程环境下不可靠。假设线程A进入instance()函数，执行第14行代码，此时线程被挂起(suspended)。在A被挂起时，它刚判断出pInstance是空值，也就是说Singleton的对象还未被创建。

现在线程B开始运行，进入instance()函数，并执行第14行代码。线程B也判断出pInstance为空，因此它继续执行第15行，创建出一个Singleton对象，并将pInstance指向该对象，然后把pInstance返回给instance()函数的调用者。

之后的某一时刻，线程A恢复执行，它接着做的第一件事就是执行第15行：创建出另一个Singleton对象，并让pInstance指向新对象。这显然违反了“单例(singleton)”的本意，因为现在我们有了两个Singleton对象。

从技术上说，第11行才是pInstance初始化的地方，但实际上，我们到第15行才将pInstance指向我们所希望它指向的内容，因此本文在提及pInstance初始化的地方，都指的是第15行。

将经典的单例实现成支持线程安全性是很容易的事，只需要在判断pInstance之前加锁(lock)即可：

```
Singleton * Singleton :: instance ( ) {

Lock lock ; // acquire lock (params omitted for simplicity) 加锁（为了简便起见，代码中忽略了加锁所需要的参数）

if ( pInstance == 0 ) {

pInstance = new Singleton ;

}

return pInstance ;

} // release lock (via Lock destructor) // 解锁（通过Lock的析构函数实现）
```

这个解决办法的缺点在于可能会导致昂贵的程序执行代价：每次访问该函数都需要进行一次加锁操作。但实际中，我们只有pInstance初始化时需要加锁。也就是说加锁操作只有instance()第一次被调用时才是必要的。如果在程序运行过程中，intance()被调用了n次，那么只有第一次调用锁起了作用。既然另外的n-1次锁操作都是没必要的，那么我们为什么还要付出n次锁操作的代价呢？DCLP就是设计来解决这个问题的。

3 双重检查锁定模式

DCLP的关键之处在于我们观察到的这一现象：调用者在调用instance()时，pInstance在大部分时候都是非空的，因此没必要再次初始化。所以，DCLP在加锁之前先做了一次pInstance是否为空的检查。只有判断结果为真（即pInstance还未初始化），加锁操作才会进行，然后再次检查pInstance是否为空（这就是该模式被命名为双重检查的原因）。第二次检查是必不可少的，因为，正如我们之前的分析，在第一次检验pInstance和加锁之间，可能有另一个线程对pInstance进行初始化。

以下是DCLP经典的实现代码[13, 14]:

```
Singleton * Singleton :: instance ( ) {

if ( pInstance == 0 ) { // 1st test 第一次检查

Lock lock ;

if ( pInstance == 0 ) { // 2nd test 第二次检查

pInstance = new Singleton ;

}

}

return pInstance ;

}
```

定义DCLP的文章中讨论了一些实现中的问题（例如，对单例指针加上volatile限定(译者注[3])的重要性，以及多处理器系统上独立缓存的影响，这两点我们将在下文讨论；但关于某些读写操作需要确保原子性这一点本文不予讨论），但他们都没有考虑到一个更基本的问题：DCLP的执行过程中必须确保机器指令是按一个可接受的顺序执行的。本文将着重讨论这个基本问题。

4 DCLP与指令执行顺序

我们再来思考一下初始化pInstance的这行代码:

```
pInstance = new Singleton ;
```

这条语句实际做了三件事情：

第一步：为Singleton对象分配一片内存
第二步：构造一个Singleton对象，存入已分配的内存区
第三步：将pInstance指向这片内存区

这里至关重要的一点是：我们发现编译器并不会被强制按照以上顺序执行！实际上，编译器有时会交换步骤2和步骤3的执行顺序。编译器为什么要这么做？这个问题我们留待下文解决。当前，让我们先专注于如果编译这么做了，会发生些什么。

请看下面这段代码。我们将pInstance初始化的那行代码分解成我们上文提及的三个步骤来完成，把步骤1（内存分配）和步骤3（指针赋值）写成一条语句，接着写步骤2（构造Singleton对象）。正常人当然不会这么写代码，可是编译器却有可能将我们上文写出的DCLP源码生成出以下形式的等价代码。

```
Singleton * Singleton :: instance ( ) {
if ( pInstance == 0 ) {
Lock lock ;
if ( pInstance == 0 ) {
pInstance = // Step 3 步骤3
operator new ( sizeof ( Singleton ) ) ; // Step 1 步骤1
new ( pInstance ) Singleton ; // Step 2 步骤2
}
}
return pInstance ;
}
```

一般情况下，将DCLP源码转化成这种代码是不正确的，因为在步骤2调用Singleton的构造函数时，有可能抛出异常(exception)。如果异常抛出，很重要的一点在于pInstance的值还没发生改变。这就是为什么一般来说编译器不会把步骤2和步骤3的位置对调。然而，在某些条件下，生成的这种代码是合法的。最简单的一种情况是编译器可以保证Singleton构造函数不会抛出异常（例如通过内联化后的流分析(post-inlining flow analysis），当然这不是唯一情况。有些抛出异常的构造函数会自行调整指令顺序，因此才会出现这个问题。

根据上述转化后的等价代码，我们来考虑以下场景：

1. 线程A进入instance()，检查出pInstance为空，请求加锁，而后执行由步骤1和步骤3组成的语句。之后线程A被挂起。此时，pInstance已为非空指针，但pInstance指向的内存里的Singleton对

象还未被构造出来。
2. 线程B进入instance(), 检查出pInstance非空，直接将pInstance返回(return)给调用者。之后，调用者使用该返回指针去访问Singleton对象————啊哦，显然这个Singleton对象实际上还未被构造出来呢！

只有步骤1和步骤2在步骤3之前执行，DCLP才有效，但在C/C++中我们没有办法表达这种限制。这就像一把尖刀插进DCLP的心脏：我们必须为相关指令顺序定义一些限制，但我们所使用的语言却无法表达这种限制。

是的，C/C++标准[16, 15]确实为语句的求值顺序定义了相应的限制，即序列点(sequence points)。例如，C++标准中1.9章节的第7段有句激动人心的描述：

“序列点(sequence point)是指程序运行到某个特别的时间点，在此之前的所有求值的副作用(side effect,译者注[2])应已结束，且后续求值的副作用应还未发生。”

此外，标准中还声明了序列点在每条语句之后发生。看来只要你小心谨慎地将你的语句排好序，一切就都有条不紊了。

噢，奥德修斯(Odysseus, 译者注[4]), 千万别被她动人的歌声所迷惑，前方还要许多艰难困苦等着你和你的弟兄们呢！

C/C++标准根据抽象机器的可见行为(observable behavior)定义了正确的程序行为。但抽象机器里并非所有行为都可见。例如下文中这个简单的函数：

```
void Foo ( ) {
int x = 0 , y = 0 ; // Statement 1 语句1
x = 5 ; // Statement 2 语句2
y = 10 ; // Statement 3 语句3
printf ( "%d, %d" , x , y ) ; // Statement 4 语句4
}
```

这个函数看起来挺傻，但它可能是Foo调用其它内联函数展开后的结果。

C和C++的标准都保证了Foo()函数的输出是”5， 10″，因此我们也知道是这样的结果。但我们只是根据c/c++标准中给出的保证，得出我们的结论。我们其实根本不知道语句1-3是否真的会被执行。事实上，一个好的优化器会丢弃这三条语句。如果语句1-3被执行了，那么我们可以肯定语句1的执行先于语句2-4（假设调用的printf()不是个内联函数，并且结果没有被进一步优化），我们也可以肯定语句4的执行晚于语句1-3，但我们并不知道语句2和语句3的执行顺序。编译器可能让语句2先执行，也可能让语句3先执行，如果硬件支持，它甚至有可能让两条语句并行执行。这种可能性很大，因为现代处理器支持大字长以及多执行单元，两个或更多的运算器也很常见（例如，奔腾4处理器有三个整形运算器，PowerPC G4e处理器有四个，Itanium处理器有6个）。这些机器都允许编译器生成可并行执行的代码，使得处理器在一个时钟周期内能够处理两条甚至更多指令。

优化编译器会仔细地分析并重新排序你的代码，使得程序执行时，在可见行为的限制下，同一时间能做尽可能多的事情。在串行代码中发现并利用这种并行性是重新排列代码并引入乱序执行最重要的原因，但并不是唯一原因，以下几个原因也可能使编译器（和链接器）将指令重新排序：

1. 避免寄存器数据溢出；
2. 保持指令流水线连续；
3. 公共子表达式消除；

4. 降低生成的可执行文件的大小[4]。

C/C++的编译器和链接器执行这些优化操作时，只会受到C/C++标准文档中定义的抽象机器上可见行为的原则这唯一的限制。有一点很重要：这些抽象机器默认是单线程的。C/C++作为一种语言，二者都不存在线程这一概念，因此编译器在优化过程中无需考虑是否会破坏多线程程序。如果它们这么做了，请别惊讶。

既然如此，程序员怎样才能用C/C++写出能正常工作的多线程程序呢？通过使用操作系统特定的库来解决。例如Posix线程库(pthreads)[6]，这些线程库为各种同步原语的执行语义提供了严格的规范。由于编译器生成代码时需要依赖这些线程库，因此编译器不得不按线程库所约束的执行顺序生成代码。这也是为什么多线程库有一部分需要用直接用汇编语言实现，或者调用由汇编实现的系统调用（或者使用一些不可移植的语言）。换句话说，你必须跳出标准C/C++语言在你的多线程程序中实现这种执行顺序的约束。DCLP试图只使用一种语言来达到目的，所以DCLP不可靠。

通常，程序员不愿意受编译器摆布。或许你也是这类程序员之一。如果是，那你可能会忍不住调整你的代码，让pInstance的值在Singleton构造完成之前决不发生任何改变，以此巧胜编译器。你可能会试着加入一个临时变量，如下：

```
Singleton * Singleton :: instance ( ) {
if ( pInstance == 0 ) {
Lock lock ;
if ( pInstance == 0 ) {
Singleton * temp = new Singleton ; // initialize to temp 初始化temp
pInstance = temp ; // assign temp to pInstance 将temp赋值给pInstance
}
}
return pInstance ;
}
```

本质上，你已经挑起了一场代码优化之战。编译器想优化代码，可你不希望它这么做，至少你不希望它对这段代码这么做。但这决不是一场你想参与的战争。因为你的敌人老奸巨滑，满肚子诡计，要知道它可是由一群几十年来成天啥也不做一心只想着如何进行编译优化的人实现的。除非你能自己写优化编译器，否则它们总是领先于你。以上段代码为例，编译器能很轻易地通过相关性分析得出temp只是一个无关紧要的变量，因此它会直接删除该变量，将你精心写下的“不可优化”代码视为如同用传统 DCLP 方式写就的一样。游戏结束了，你输啦！

如果你使用杀伤力大点的武器，试图扩大temp的作用域(例如将temp设成static)，编译器照样能用相同的分析法得出相同的结论。游戏结束了，你输啦！

于是你请求支援，将temp声明成extern，并将其定义到单独的编译单元中，想以此让编译器不知道你的意图。真为你感到难过啊！因为有些编译器似乎带有“优化夜视镜”，它们能利用过程间分析来发现你对temp动的小脑筋，再一次优化了temp。永远记住，它们是”优化”编译器。它们的目的就是追踪不必要的代码并优化之。游戏结束了，你输啦！

再换一种办法，你试着在另一个文件中定义一个辅助函数来消除内联，这样可以强迫编译器假设构造函数可能会抛出异常，从而延迟pInstance的赋值。好办法，值得一试！但有一些构建环境会

采取链接时内联，随之而来的是更多的代码优化。好了，游！戏！结！束！你！又！输！啦！

一个基本的问题，你没有能力改变：你希望利用约束条件让指令按顺序执行，但你所使用的语言不提供任何实现方法。

5 volatile关键字的成名之路

我们非常希望某些特定的指令可以按顺序执行，这引发了许多关volatile关键字的思考：volatile是否能够从总体上帮助多线程程序，特别是对DCLP有所帮助？这一节中，我们将注意力集中在C++里volatile关键字的语义上，然后进一步集中讨论volatile对DCLP的影响。关于volatile更深入的讨论，可以参考本文结尾附上的补充说明。

C++标准文档[15] 1.9节中有如下信息（斜体为本文作者所加）：

“C++抽象机器上的可见行为包括：*volatile*数据的读写顺序，以及对输入输出(I/O)库函数的调用。将声明成*volatile*的数据作为左值来访问对象，修改对象，调用输入输出库函数，抑或调用其他有以上相似操作的函数，都会产生*副作用（side effects）*（译者注[2]），即执行环境状态发生的改变。”

我们早前的观察如下：

(1) C/C++标准文档中保证所有的副作用（side effects）将在程序运行到序列点时完成，
(2) 并且，序列点发生在每个c++语句结束之时，

结合上述事实，如果我们想确保正确指令执行顺序，那么我们所要做的就是将合适的数据声明成volatile，并谨慎安排语句顺序。

早期分析显示我们需要将pInstance声明成volatile，DCLP[13,14]的相关论文中也给出了这一结论。然而，福尔摩斯大侦探，你一定注意到：为了确保正确的指令顺序，Singleton对象本身也必须声明成volatile。原版的DCLP论文中并没有指出这一点，这很重要，但他们疏忽了。

为了让大家理解为什么仅将pInstance声明为volatile是不够的，我们考虑以下代码：

```
1  class Singleton {
2  public :
3  static Singleton * instance ( ) ;
4  . . .
5  private :
6  static Singleton * volatile pInstance ; // volatile added 加上volatile声明
7  int x ;
8  Singleton ( ) : x ( 5 ) { }
9  } ;
10 // from the implementation file 实现文件内容如下
11 Singleton * volatile Singleton :: pInstance = 0 ;
12 Singleton * Singleton :: instance ( ) {
```

```
13 if ( pInstance == 0 ) {
14 Lock lock ;
15 if ( pInstance == 0 ) {
16 Singleton * volatile temp = new Singleton ; // volatile added 加上volatile声明
17 pInstance = temp ;
18 }
19 }
20 return pInstance ;
21 }
```

将构造函数内联化后，代码展开如下：

```
if ( pInstance == 0 ) {
Lock lock ;
if ( pInstance == 0 ) {
Singleton * volatile temp =
static_cast < Singleton * > ( operator new ( sizeof ( Singleton ) ) ) ;
temp -> x = 5 ; // inlined Singleton constructor
pInstance = temp ;
}
}
```

虽然temp是volatile的，但*temp却不是，这就意味着temp->x也不是。我们现在已经明白非volatile数据在赋值时执行顺序可能会发生变化，因此我们也很容易得知编译器可以改变temp->x赋值与pInstance赋值的顺序。如果编译器这么做了，那么pInstance就将赋值为还未完全初始化的temp，因为它的成员变量x还未初始化，这就可能导致另一个线程读取到这个未初始化的x。

一种比较吸引人的解决办法是将*pInstance与pInstance一样限定成volatile，该方法的美化版是将Singleton声明成volatile，这样所有Singleton变量都将是volatile的。

```
1 class Singleton {
2 public :
3 static volatile Singleton * volatile instance ( ) ;
4 . . .
5 private :
```

```
6  // one more volatile added 加入另一个volatile声明
7  static volatile Singleton * volatile pInstance ;
8  } ;
9  // from the implementation file 实现文件内容如下
10 volatile Singleton * volatile Singleton :: pInstance = 0 ;
11 volatile Singleton * volatile Singleton :: instance ( ) {
12 if ( pInstance == 0 ) {
13 Lock lock ;
14 if ( pInstance == 0 ) {
15 // one more volatile added 加入另一个volatile声明
16 volatile Singleton * volatile temp =
17 new volatile Singleton ;
18 pInstance = temp ;
19 }
20 }
21 return pInstance ;
22 }
```

(至此，读者可能会提出一个合理的疑问：为什么不将Lock也声明成volatile的？毕竟在我们试图给pInstance或temp赋值前将lock初始化至关重要。因为Lock由多线程库提供，所以我们可以假设Lock的说明文档已经给出了足够的限制，使其在执行过程中无需声明为volatile即可保证执行顺序。我们所知的所有线程库都是这么做的。从本质上说，使用线程库中的实体，如对象、函数等，都会导致给程序强行加入“硬序列点(hard sequence points)”，即适用于所有线程的序列点。为了达到本文的目的，我们假设这类“硬序列点”可以阻止编译器在代码优化时对指令进行重新排序：源代码中使用库实体之前的语句所对应的指令，不会被移到使用库实体的语句指令之后，反之，使用库实体之后的语句指令也不会被移到使用库实体的语句指令之前。真正的线程库不会有如此严格的限制，但这些细节对本文的讨论并不重要。)

现在我们希望我们上述完全加入了volatile限定的代码已经满足标准文档的说明，能够保证该段代码在多线程环境中正确运行，然而我们还有可能面临失败，原因有二。

第一，标准文档中对可见行为的约束仅针对标准中定义的抽象机器，而所谓的抽象机器对执行过程中的多线程毫无概念。因此，虽然标准文档避免编译器在一个线程中重新排列volatile数据的读写顺序，但它对跨线程的重新排序没有任何约束。至少大部分编译器的实现者是这么解释的。因此，现实中，许多编译器都可以将上述源代码生成非线程安全的代码。如果你的多线程程序在加上volatile声明时可以正确运行，但不加声明却发生错误，那么，要么是你的编译器小心地实现对volatile的处理使其在多线程时正确运行（这种可能性较少），要么就是你运气挺好（这种可能性挺大）。但无论是哪种原因，你的代码都不可移植。

第二，正如声明为const的对象要成为const得等它的构造函数执行完成后一样，限制成volatile的对象也要等到其构造函数退出。请看以下语句：

```
volatile Singleton * volatile temp = new volatile Singleton ;
```

创建的temp对象要直到以下表达式执行完成之后才能成为volatile的

这意味着我们又回到了之前的境况：内存分配指令与对象初始化指令可能被任意调换顺序。

尽管有些尴尬，但这个问题我们能够解决。在Singleton构造函数中，当它的每个数据成员初始化时，我们都使用cast将其强制转换为volatile，这样可以避免改变初始化指令的执行位置。以下代码就是Singleton构造函数实现代码的例子。（为了简化代码，我们依然沿用之前的代码，使用赋值语句代替初始化列表。这对我们解决当前问题没有任何影响。

```
Singleton ( )
{
static_cast < volatile int & > ( x ) = 5 ; // note cast to volatile 注意强制转换成volatile
}
```

对pInstance加入适当的volatile限定，并将内联函数展开，我们可以得到如下代码：

```
1  class Singleton {
2  public :
3  static Singleton * instance ( ) ;
4  . . .
5  private :
6  static Singleton * volatile pInstance ;
7  int x ;
8  . . .
9  } ;
10 Singleton * Singleton :: instance ( )
11 {
12 if ( pInstance == 0 ) {
13 Lock lock ;
14 if ( pInstance == 0 ) {
15 Singleton * volatile temp =
16 static_cast < Singleton * > ( operator new ( sizeof ( Singleton ) ) ) ;
17 static_cast < volatile int & > ( temp -> x ) = 5 ;
```

```
18
    pInstance = temp ;
19
    }
20
    }
21
    }
22
```

现在，x的赋值必须先于pInstance的赋值，因为它们都是volatile的。

不幸的是，所有这一切都无法解决我们的第一个问题：C++的抽象机器是单线程的，C++编译器无论如何都可能为上述代码生成非线程安全的代码。否则，不优化这些代码会导致很大的效率问题。进行了这么多讨论之后，我们又回到了原点。可等一等，还有另一个问题：多处理器。

6 多处理器上的DCLP

假设你的机器有多个处理器，每个都有各自的高速缓存，但所有处理器共享内存空间。这种架构需要设计者精确定义一个处理器该如何向共享内存执行写操作，又该何时执行这样的操作，并使其对其他处理器可见。我们很容易想象这样的场景：当某一个处理器在自己的高速缓存中更新的某个共享变量的值，但它并没有将该值更新至共享主存中，更不用说将该值更新到其他处理器的缓存中了。这种缓存间共享变量值不一致的情况被称为缓存一致性问题(cache coherency problem)。

假设处理器A改变了共享变量x的值，之后又改变了共享变量y的值，那么这些新值必须更新至主存中，这样其他处理器才能看到这些改变。然而，由于按地址顺序递增刷新缓存更高效，所以如果y的地址小于x的地址，那么y很有可能先于x更新至主存中。这样就导致其他处理器认为y值的改变是先于x值的。

对DCLP而言，这种可能性将是一个严重的问题。正确的Singleton初始化要求先初始化Singleton对象，再初始化pInstance。如果在处理器A上运行的线程是按正确顺序执行，但处理器B上的线程却将两个步骤调换顺序，那么处理器B上的线程又会导致pInstance被赋值为未完成初始化的Singleton对象。

解决缓存一致性问题的一般方法是使用内存屏障(memory barriers), 例如使用栅栏(fences, 译者注[6])：即在共享内存的多处理器系统中，用以限制对某些可能会对共享内存进行读写的指令进行重新排序，能被编译器、链接器，或者其他优化实体识别的指令。对DCLP而言，我们需要使用内存关卡以保证pInstance的赋值在Singleton初始化完成之后。下面这段伪代码与参考文献[1]中的一个例子非常相似，我们只在需要加入内存关卡之处加入相应的注释，因为实际的代码是平台相关的（通常使用汇编）。

```
1  Singleton * Singleton :: instance ( ) {
2  Singleton * tmp = pInstance ;
3  . . . // insert memory barrier 加入内存屏障
4  if ( tmp == 0 ) {
5  Lock lock ;
6  tmp = pInstance ;
7  if ( tmp == 0 ) {
```

```
8  tmp = new Singleton ;
9  . . . // insert memory barrier 加入内存屏障
10 pInstance = tmp ;
11 }
12 }
13 return tmp ;
14 }
```

Arch Robison指出这种解决办法杀伤力过大了（他是参考文献[12]的作者，但这些观点是私下与他交流时提及的）：从技术上说，我们并不需要完整的双向屏障。第一道屏障可以防止另一个线程先执行Singleton构造函数之后的代码，第二道屏障可以防止pInstance初始化的代码先于Singleton对象的初始化。有一组称作“请求”和“释放”操作可以比单纯用硬件支持内存关卡(如Itainum处理器)具有更高的效率。

但无论如何，只要你的机器支持内存屏障，这是DCLP一种可靠的实现方法。所有可以重新排列共享内存的写入操作指令顺序的处理器都支持各种不同的内存屏障。有趣的是，在单处理器系统中，同样的方法也适用。因为内存关卡本质上是“硬序列点”，即从硬件层面防止可能引发麻烦的指令重排序。

7 结论以及**DCLP**的替代方法

从以上讨论中我们可以得出许多经验。首先，请记住一点：基于分时的单处理机并行机制与真正跨多处理器的并行是完全不同的。这就是为什么在单处理器架构下针对某个编译器的线程安全的解决办法，在多处理器架构下就不可用了。即使你使用相同的编译器，也可能导致这个问题（这是个一般性结论，不仅仅存在于DCLP中）。

第二，尽管从本质上讲DCLP并不局限于单例模式，但以DCLP的方式使用单例模式往往会导致编译器去优化跟线程安全有关的语句。因此，你必须避免用DCLP实现Singleton模式。由于DCLP每次调用instance()时都需要加一个同步锁，如果你(或者你的客户)很在意加锁引起的性能问题，你可以建议你的客户将instance()返回的指针缓存起来，以达到最小化调用instance()的目的。例如，你可以建议他不要这么写代码：

```
Singleton :: instance ( ) -> transmogrify ( ) ;
Singleton :: instance ( ) -> metamorphose ( ) ;
Singleton :: instance ( ) -> transmute ( ) ;
clients do things this way :
```

而应该将上述代码改写成：

```
Singleton * const instance =
Singleton :: instance ( ) ; // cache instance pointer 用变量存instance()指针
instance -> transmogrify ( ) ;
```

```
instance -> metamorphose ( ) ;

instance -> transmute ( ) ;
```

要实现这个想法有个有趣的办法，就是鼓励用户尽量在每个需要使用singleton对象的线程开始时，只调用一次instance()，之后该线程就可直接使用缓存在局部变量中的指针。使用该方法的代码，对每个线程都只需要承担一次instance()调用的代价即可。

在采用“缓存调用结果”这一建议之前，我们最好先验证一下这样是否真的能够显著地提高性能。加入一个线程库提供的锁，以确保Singleton初始化时的线程安全性，然后计时，看看这样的代价是否真正值得我们担心。

第三，请不要使用延迟初始化(lazily-initialized)的方式，除非你必须这么做。单例模式的经典实现方法就是基于这种方式：除非有需求，否则不进行初始化。替代方法是采用提前初始化(eager initialization)方式，即在程序运行之初就对所需资源进行初始化。因为多线程程序在运行之初通常只有一个线程，我们可以将某些对象的初始化语句写在程序只存在一个线程之时，这样就不用担心多线程所引起的初始化问题了。在很多情况下，将singleton对象的初始化放在程序运行之初的单线程模式下（例如，在进入main函数之前初始化），是最简便最高效且线程安全的singleton实现方法。

采用提前初始化的另一种方法是用单状态模式(Monostate模式)[2]代替单例模式。不过，Monostate模式属于另一个话题，特别是关于构成它的状态的非局部静态对象初始化顺序的控制，是一个完全不同的问题。Effective C++[9]一书中对Monostate的这个问题给出了介绍，很讽刺的是，关于这一问题，书中给出的方案是使用Singleton变量来避免（这个变量并不能保证线程安全[17]）。

另一种可能的方法是每个线程使用一个局部singleton来替代全局singleton，在线程内部存储singleton数据。延迟初始化可以在这种方法下使用而无需考虑线程问题，但这同时也带来了新的问题：一个多线程程序中竟然有多个“单例”。

最后，DCLP以及它在C/C++语言中的问题证实了这么一个结论：想使用一种没有线程概念的语言来实现具有线程安全性的代码(或者其他形式的并发式代码)有着固有的困难。编程中对多线程的考虑很普遍，因为它们是代码生成中的核心问题。正如Peter Buhr的观点，指望脱离语言，只靠库函数来实现多线程简直就是痴心妄想。如果你这么做了，要么

(1) 库最终会在编译器生成代码时加入各种约束（pthreads库正是如此）

要么

(2) 编译器以及其它代码生成工具的优化功能将被禁用，即使针对单线程代码也不得不如此。多线程、无线程概念的编程语言、优化后的代码，这三者你只能挑选两个。例如，Java和.net CLI解决矛盾的办法是将线程概念加入其语言结构中[8, 12]。

8 致谢

本文发表前曾邀请Doug Lea, Kevlin Henney,Doug Schmidt, Chuck Allison, Petru Marginean, Hendrik Schober, David Brownell, Arch Robison, Bruce Leasure, and James Kanze审阅及校稿。他们的点评建议为本文的发表做了很大的贡献，并使我们对DCLP、多线程、指令重排序、编译器优化这些概念又有了进一步的理解。出版后，我们还加入了Fedor Pikus, Al Stevens, Herb Sutter, and John Hicken几人的点评建议。

9 关于作者

Scott Meyers曾出版了三本《Effective C++》的书，并出任Addison-Wesley所著的《有效的软件开发系统丛书(Effective Software Developement Series)》的顾问编辑。目前他专注于研究提高软

件质量的基本原则。他的主页是：http://aristeia.com.

Andrei Alexandrescu是《Modern C++ Design》一书的作者，(译者注[5])，他还写过大量文章，其中大部分都是作为专栏作家为CUJ所写。目前他正在华盛顿大学攻读博士学位，专攻编程语言方向。他的主页是：http://moderncppdesign.com.

10 [补充说明] volatile的发展简史

volatile产生的根源得追溯到20世纪70年代，那时Gordon Bell(PDP-11架构的设计者)提出了内存映射I/O(MMIO)的概念。在这之前，处理器为I/O端口分配针脚，并定义专门的指令访问。MMIO让I/O与内存共用相同的处理器针脚和指令集。处理器外部的硬件将某些特定的内存地址转换成I/O请求，因此对I/O端口的读写变得与访问内存一样简单。

真是个好主意！减少针脚数量是个好办法，因为针脚会降低信号传输速度、增加出错率，并使封装复杂化。而且MMIO不需要为I/O使用专门的指令集，程序只需像访问内存一样即可，剩下的工作都由硬件去完成。

或者我们应该说：“几乎”都由硬件去完成。

让我们通过以下代码来看看为什么MMIO需要引入volatile变量：

```
unsigned int * p = GetMagicAddress ( ) ;

unsigned int a , b ;

a = * p ;

b = * p ;
```

如果p指向一个端口，a和b应该能够从该端口读取到两个连续的字长(words)。然而，如果p指向一个真实的内存地址，那么a和b将分别被赋成同一个地址值，因此a和b理应相等。编译器就是基于这一假设而设计的，因此它将上述代码的最后一行优化成更高效的代码：

同样，对于相等的p, a, b，考虑如下代码：

这段代码是将两个字长写入*p中，但优化器却将*p假设成内存，把上述的两次赋值认为是重复的代码，优化了其中一次赋值。显然这样的“优化”破坏了代码的本意。类似的场景可以出现在当一个变量同时被主线代码和中断服务程序(ISR)改变时。针对这种情况，为了主线代码与中断服务程序间的通信，编译器处理冗余的读写是有必要的。

因此，在处理相同内存地址的相关代码时（如内存映射端口或与ISR相关的内存），编译器不得对其进行优化。对这类内存地址需有要特殊的处理，volatile就应运而生了。Volatile用于说明以下几个含义：

(1) 声明成volatile的变量其内容是“不稳定的”（unstable），它的值可能在编译器不知道的情况下发生变化
(2) 所有对声明成volatile的数据的写操作都是“可见的”（observable）,因此必须严格执行这些写操作
(3) 所以对声明成volatile的数据的操作都必须按源代码中的顺序执行

前两条规则保证了正确的读操作和写操作，最后一条保证I/O协议能正确处理读写混合操作。这正是C/C++中volatile关键字所保证的。

Java语言对volatile作了进一步扩展：在多线程环境下也能保证volatile的上述性质。这是一步很重要的扩展，但还不足以使volatile完全适用于线程同步性，因为volatile和非volatile操作间的顺序

依然没有明确规则。由于忽略了这一点，为了保证合适的执行顺序，大量的变量都不得不声明为volatile。

Java 1.5版本中的对volatile[10]对“请求/释放”语义有了更严格但更简单的限制：确保所有对volatile的读操作都发生在该语句后所有读写操作之前(无论这些读写操作是否为针对volatile数据)；确保所有对volatile的写操作都发生在该语句前所有读写操作之后。.NET也定义了跨线程的volatile语义，它与目前Java所用的语义基本相同。而目前C/C++中的volatile还没有类似的改动。

译者注：

[1]序列点(sequence point)是指程序运行到某个特别的时间点，在这个时间点之前的所有副作用(side effect,译者注[2])已经结束，并且后续的副作用还没发生。

[2]副作用(side effect)是指对数据对象或者文件的修改。例如，”var=99″的副作用是把var的值修改成99。

[3]volatile限定(volatile-qualifying): volatile是c/c++中的关键字，与const类似，都是对变量进行附加修饰，旨在告之编译器，该对象的值可能在编译器未监测到的情况下被改变，编译器不能武断的对引用这些对象的代码作优化处理。

[4]奥德修斯(Odysseus)：《奥德赛》是古希腊最重要的两部史诗之一（两部史诗统称为《荷马史诗》，另一部是《伊利亚特》），奥德修斯是该作品中主人公的名字。作品讲述了主人公奥德修斯10年海上历险的故事。途中，他们经过一个妖鸟岛，岛上的女巫能用自己的歌声诱惑所有过路船只的船员，使他们的船触礁沉没。智勇双全的奥德修斯抵御了女巫的歌声诱惑，带领他的船员们顺利渡过难关。本文的作者借用《奥德赛》里的这个故事告诉大家不要被C/C++的标准所迷惑:D

[5]CUJ: C/C++ Users Journal | C/C++用户日报

[6]内存屏障/栅栏(Memory Barriers / Fences): 在多线程环境下，需要采用一些技术让程序结果及时可见，一旦内存被推到缓存，就会引发一些协议消息，以确保所有共享数据的缓存一致性，这种使内存对处理器可见的技术被称为内存关卡或栅栏。(Ref: http://mechanical-sympathy.blogspot.com/2011/07/memory-barriersfences.html)