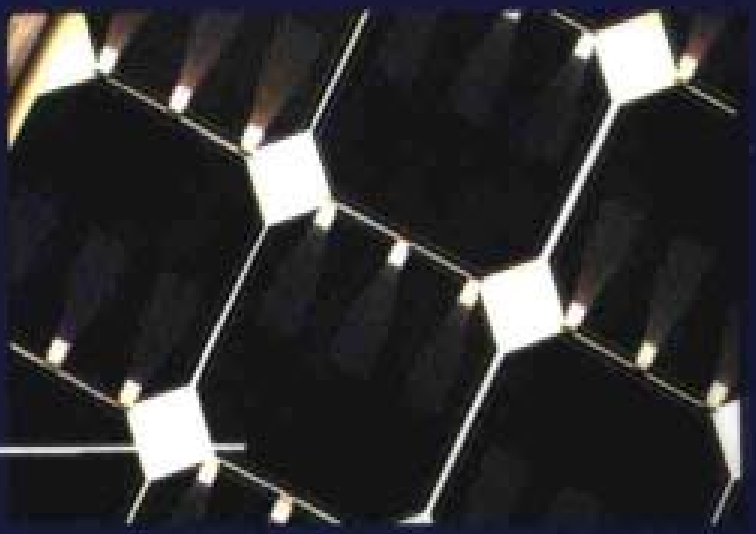
PEARSON
Addison Wesley

提高C++性能的编程技术

Efficient C++

Performance Programming Techniques



〔美〕 Dov Bulka
David Mayhew 著
常晓波
朱剑平 译

清华大学出版社

提高C++性能的编程技术

很多程序员和软件设计师都认为追求高效的C++是一种很愚蠢的做法。他们认为C++天生就慢，对性能要求很高的应用程序不适合。因此，有一些领域是C++几乎无法成功进入的，如网络、操作系统内核、设备驱动程序等。

本书反驳了这种观点。在使商业应用程序获取最大性能方面，本书的两位作者拥有第一手资料。本书说明了使用C++开发高效程序的潜力，揭示了实践中一些平常的面向对象的设计原则以及获得大幅度性能提高的C++编程技术。本书还指出了在设计和编码中产生隐含操作代价的一些常见错误。

本书关注强大和灵活性与良好性能和可伸缩性的结合，从而使两方面均达到最佳。具体主题包括临时对象、内存管理、模板、继承、虚函数、内联、引用计数、STL等。

通过本书，您将能够精通最佳性能技术的主要内容。

作者介绍

Dov Bulka在软件开发以及向市场交付大型软件产品方面有15年的经验。他曾是IBM DominoGo Web服务器的性能设计师，一些曾出现在Internet上的最大型Web站点使用了这种服务器，包括1996年亚特兰大奥运会的Web站点。他获得了杜克大学的计算机科学博士学位。

David Mayhew是StarBridge Technologies, Inc.的首席设计师。他主要从事互连构造、对等处理和PCI总线发展方面的工作，曾就职于IBM的网络软件部。他获得了Virginia Tech的计算机科学博士学位。

ISBN 7-302-06550-0

9 787302 065500 >

定价：33.00元

PEARSON

Addison Wesley

www.PearsonEd.com

提高 C ++ 性能的编程技术

Efficient C ++ Performance Programming Techniques

[美] Dov Bulka David Mayhew 著

常晓波 朱剑平 译

清华大学出版社

北 京

Efficient C++ Performance Programming Techniques
Dov Bulka David Mayhew
EISBN:0-201-37950 3

图书在版编目(CIP)数据

提高 C++ 性能的编程技术/(美)布尔卡，(美)梅休著；常晓波等译. 北京：清华大学出版社，2003
书名原文：Efficient C++ Performance Programming Techniques
ISBN 7-302-06550-0

Ⅰ. 提… Ⅱ. ①布… ②梅… ③常… Ⅲ. C 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2003)第 027069 号

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.com.cn
责任编辑：赵彤伟
版式设计：刘祎淼
印 刷 者：世界知识印刷厂
发 行 者：新华书店总店北京发行所
开 本：787×960 1/16 **印张**：16 **字数**：319 千字
版 次：2003 年 6 月第 1 版 2003 年 6 月第 1 次印刷
书 号：ISBN 7-302-06550-0/TP·4907
印 数：0001～4000
定 价：33.00 元

目　录

提高C++性能的编程技术

第1章

跟踪范例

提高C++性能的编程技术

我们所用过的软件产品一般都包括某种形式的跟踪功能。当代码超过几千行时，跟踪功能就显得十分关键。调试、维护和理解大中型软件的执行流程是重要的。不能指望在关于性能的书籍中讨论跟踪问题，但事实是(在不止一种情况下)，由于跟踪实现不当，将会导致严重的性能下降。即使存在微不足道的低效也会对性能造成巨大的影响。本章的目标不是教授人们正确的跟踪实现，而是要通过跟踪这一载体引出一些经常出现在C++代码中的重要性能原理。跟踪功能的实现引进了典型的C++性能障碍，这使它成为讨论性能的好材料。它简单且众所周知，我们不必为了突出重要的问题而陷入无关细节的海洋之中。然而，不管简单与否，通过跟踪会让人认识到许多的性能问题，几乎可以在任何一段C++代码段中遇到这类性能问题。

许多C++程序员定义一个简单的Trace类来把诊断信息输出到日志文件中。程序员可以在任何希望跟踪的函数中定义一个Trace对象，Trace类可以分别在函数的入口和出口写一条信息。Trace对象将增加附加的执行开销，但是它有助于程序员在不使用调试器的情况下找出问题。如果您的C++代码被作为本地代码嵌入在Java程序中，那么使用Java调试器跟踪这段本地代码将是一件很有挑战性的工作。

最极端的跟踪性能优化方式是把跟踪调用嵌入在#ifdef块内，从而彻底消除性能开销：

```
# ifdef TRACE
Trace t ("myFunction");                    //Constructor takes a function name argument
t.debug("Some information message");
# endif
```

#ifdef方法的弱点是：要想打开或关闭跟踪，您必须重新编译程序。很明显程序的最终用户无法这样做，除非像免费软件那样同时发行源代码。另一种方法是：可能通过与运行状态下的程序通信来动态地控制跟踪。Trace类能够在记录任何跟踪信息之前先检查跟踪状态：

```
void
Trace::debug (string &msg)
{
    if ( traceIsActive ) {
        // log message here
        }
}
```

在跟踪处于激活状态时我们不关心性能。我们假定只在确定问题时才打开跟踪。在平常的操作中，跟踪在默认情况下是处于非激活状态的，我们希望自己的代码能展示最好的性能。为了实现这一点，必须最小化跟踪开销。典型的跟踪语句如下所示：

```
t.debug ("x = " + itoa (x));  // itoa( ) converts an int to ascii
```

这条典型的跟踪语句会呈现严重的性能问题。即使关闭了跟踪，我们仍然要创建传递给 debug ( )函数的 string 参数。这一条单独的语句隐含着多步计算：

- 为“x=”创建一个临时 string 对象。
- 调用 itoa (x)。
- 为 itoa (x)返回的字符指针创建一个临时 string 对象。
- 连接上述 string 对象以创建第三个临时 string 对象。
- 从 debug ( )调用返回后清除全部三个 string 临时对象。

这样，我们把所有问题定位到三个临时 string 对象上，接着当确定跟踪处于非激活状态时把它们全部清除。创建和清除这些 string 和 trace 对象的开销至少为几百条指令。在典型的面向对象设计的代码中，如果函数短小且调用频率很高，那么跟踪的开销很容易使性能下降一个数量级。这不是牵强的主观臆想，我们曾在一个实际产品中做过真实的实验。深入地研究这种特定的可怕经历会得到有教育意义的经验。这是一次在包含 50 万行 C ++ 代码的复杂产品中添加跟踪功能的尝试。因为性能十分糟糕，所以我们的第一次尝试适得其反。

1.1 初步的跟踪实现

我们的目的是让 trace 对象记录诸如进入函数、离开函数等事件的消息，也可能记录其他位于两者之间的值得注意的信息。

```
int myFunction (int x)
{
    string name = "myFunction";
```

```
    Trace t (name);
    ...
    string moreInfo = "more interesting info";
    t.debug (moreInfo);
    ...
};      // Trace destructor logs exit event to an output stream
```

为了启用以上用法，我们开始于如下的 Trace 实现：

```
class Trace {
public:
    Trace (const string &name);
    ~Trace ( );
    void debug (const string &msg);
    static bool traceIsActive;
private:
    string theFunctionName;
};
```

Trace 的构造函数存放函数的名字。

```
inline
Trace ::Trace ( const string &name) : theFunctionName (name)
{
if (TraceIsActive){
    cout<<"Enter function"<<name<<endl;
    }
}
```

通过 debug ( )方法把附加的消息记录到日志中。

```
inline
void Trace :: debug (const string &msg)
{
if (TraceIsActive) {
    cout<<msg<<endl;
    }
}

inline
Trace :: ~Trace ( )
```

```
{
    if ( TraceIsActive) {
        cout<<"Exit function "<<theFunctionName<<endl;
    }
}
```

一旦设计、编码和测试好 Trace 类以后，即可分发并立刻将其插入到大部分代码中。Trace 对象出现在关键执行路径的大多数函数中。在随后的性能测试中，我们吃惊地发现性能直线下降为先前性能的 20%。Trace 对象的插入把性能降低到原来的五分之一。在此我们谈论关闭跟踪并且假设性能不受影响的情况。

1.1.1 发生了什么问题

程序员根据其各自不同的经历，可能对 C ++ 的性能有不同的见解。但是有几个公认的基本原理：

- I/O 的开销是昂贵的。
- 函数调用的开销是一个因素，因此我们应该内联短小的、频繁调用的函数。
- 复制对象的开销是昂贵的。最好选择按引用传递，而不是按值传递。

上述 Trace 的初步实现符合这三个原理。如果关闭跟踪，就可避免 I/O，而且所有方法都是内联的，所有的 string 参数也都是按引用传递的。我们忠于规则但却思想僵化。很显然，上述规则中所体现的智慧缺乏开发高性能 C ++ 所要求的专门技术。

经验表明，这三个原理没有涵盖 C ++ 性能的主要问题。主要问题是对不必要对象的创建(和后面的清除)，这些不必要的对象预计要使用但却没有使用。Trace 实现是一个示例，说明了无用对象对性能的破坏性影响，即使是对 Trace 对象最简单的使用，这一点也是非常明显的。对 Trace 对象的最低限度的使用是把进入函数和离开函数记录到日志中：

```
int myFunction (int x)
{
    string name = "myFunction";
    Trace t (name);
    ...
};
```

这种最低限度的跟踪调用了一系列计算：

- 创建一个作用域为 myFunction 的 string 型变量 name。
- 调用 Trace 的构造函数。

- Trace 的构造函数调用 string 的构造函数来创建一个成员 string。

在作用域的结尾，也就是函数的结尾，Trace 对象和两个 string 对象被清除：

- 清除 string 型变量 name。
- 调用 Trace 的析构函数。
- Trace 的析构函数为成员 string 调用 string 的析构函数。

在关闭跟踪的情况下，string 成员对象从未使用。有时也可能会遇到 Trace 对象本身不怎么使用的情况(在关闭跟踪的情况下)。这时，那些对象的创建和清除用到的所有计算都纯属浪费。要记住这是关闭跟踪情况下的开销。我们假设这是快车道。

究竟代价有多大呢？为了找出基准度量，我们记录了百万次迭代函数 addOne ( )的执行时间：

```
int addOne ( int x)                    // Version 0
{
    return x + 1;
}
```

正如您所知道的那样，add()ne ( )很简单，它只是一个基准点。我们准备每次分离出一个影响性能的因素。在 main ( )函数中调用 addOne ( )一百万次并测试执行时间：

```
int main ( )
{
    Trace::TraceIsActive = false ;    // Turn tracing off
    // ...
    GetSystemTime (& t1) ;            // Start timing
        for ( i = 0 ; i<j ; i + + ) {
        y = addOne (i) ;
        }

    GetSystemTime(& t2) ;             // Stop timing
    // ...

}
```

下一步，在 addOne 中添加 Trace 对象并重新测试，以此来评估性能变化，这就是 Version 1(如图 1.1 所示)。

```
int addOne ( int x )                   // Version 1. Introducing a Trace object
{
    string name = "addOne" ;
```

```
    Trace t (name);
    return x + 1;
}
```

for 循环的开销从 55ms 骤然升至 3 500ms。换句话说，addOne 的速度直线下降了 60 倍以上。这种类型的开销将会对任何软件的性能造成严重破坏。但是完全取消跟踪机制不是办法，因为我们必须有某种形式的跟踪功能。因此我们不得不重新组织以提供一种更为有效的实现。

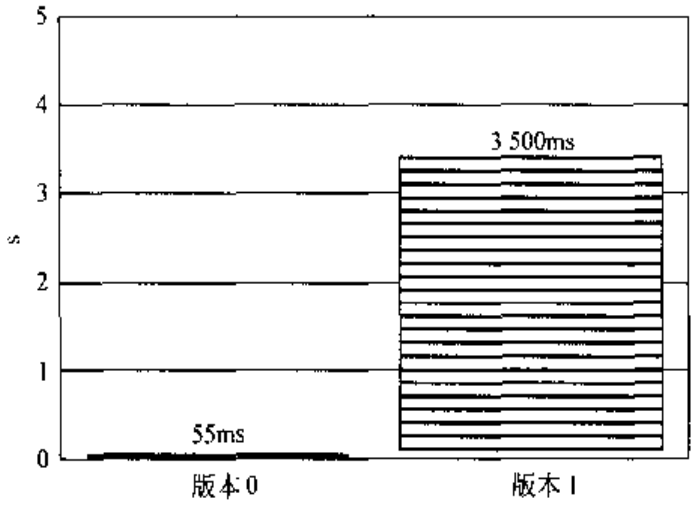


图 1.1 Trace 对象的性能开销

1.1.2 恢复计划

性能恢复计划是要在跟踪关闭时消除没有价值的对象和计算。下面从 addOne 创建并传递给 Trace 构造函数的 string 参数入手。我们把函数的 name 参数从 string 对象改成简单的 char 型指针：

```
int addOne ( int x )        // Version 2. Forget the string object.
                            // Use a char pointer instead.
{
    char    * name = "addOne";
    Trace   t (name) ;
        return x + 1;
}
```

随同此处修改，我们必须修改 Trace 构造函数本身，以便让它接收 char 型指针参数而不是 string 对象引用：

```
inline
Trace :: Trace ( const char * name) : theFunctionName (name) //Version 2
{
    if (traceIsActive) {
        cout<<"Enter function "<<name<<endl;
    }
}
```

同样，Trace :: debug ( )方法也要进行修改，它应接收一个 const * char 作为输入参数而不是 string。现在不必在创建 Trace 对象之前创建 string 对象 name 了——我们所担心的对象少了一个。这种做法会使性能提高，这在我们的测量结果中很明显。执行时间从 3 500ms 下降到了 2 500ms(如图 1.2 所示)。

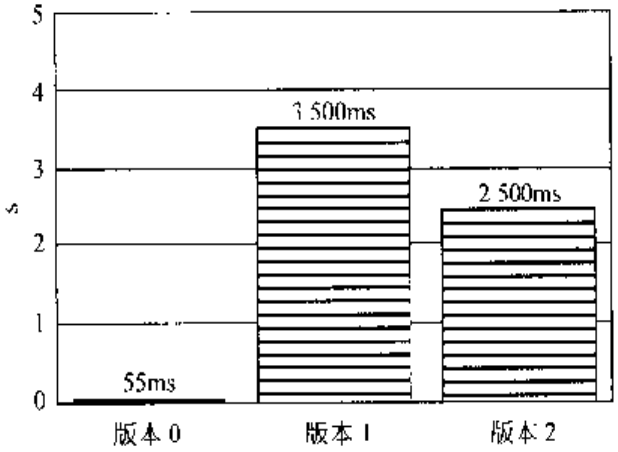


图 1.2 取消一个 string 对象后的影响

第二步是消除包含在 Trace 对象内的 string 成员对象的无条件创建。从性能的角度来看，有两种等价的解决方案。第一种是把 string 对象替换成简单 char 型指针。char 型指针通过简单的分配就可"构造"，这是廉价的。另一种解决方案是使用合成(composition)而不是聚合(aggregation)。我们不是把 string 子对象嵌入到 Trace 对象里面，而是把它换成 string 指针。与 string 对象相比，string 指针的好处是可以把 string 的创建推迟到确定跟踪处于打开状态以后。在此我们选择这种办法：

```
class Trace {     // Version 3. Use a string pointer
public:
    Trace ( const char * name) : theFunctionName(0)
    {
    if ( traceIsActive ) {   // Conditional creation
```

```
            cout<<"Enter function"<<name<<endl ;
            theFunctionName = new string(name) ;
            }
        }
        ...
    private:
        string * theFunctionName ;
    };
```

同时必须修改 Trace 的析构函数，删除 string 指针：

```
inline
Trace ::~Trace ( )
{
    if (traceIsActive) {
        cout<<"Exit function"<< * theFunctionName<<endl ;
        delete theFunctionName ;
        }
}
```

在另一次测量中显示出性能显著提高。响应时间从 2 500ms 下降到了 185ms(如图 1.3 所示)。

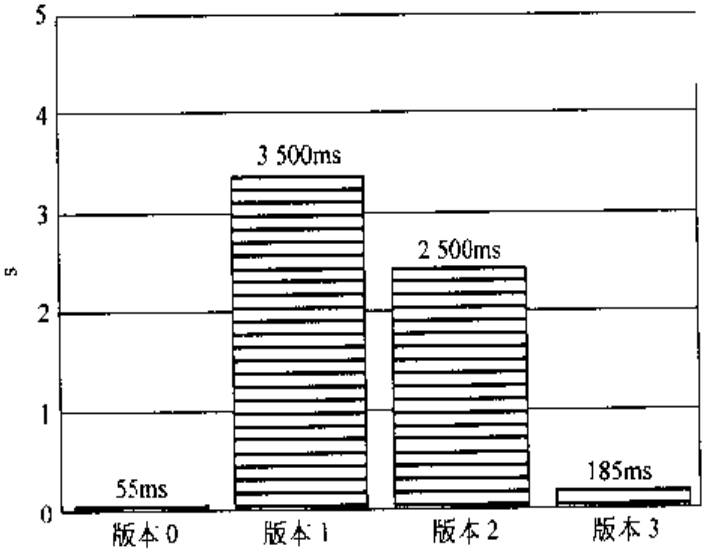


图 1.3 有条件创建 string 成员的影响

至此已经达到目的了。我们把 Trace 实现的开销从 3 500ms 降到了 185ms。比起在根本没有跟踪逻辑时的 55ms 执行时间，您可能仍然觉得 185ms 很糟糕。性能下降了

3倍多。我们又怎能算是胜利了呢?关键是原来的addOne函数(没有跟踪功能)所做的工作很少。它把输入的参数加1后立即返回。向addOne添加任何代码都会对它的执行时间有深刻的影响。如果为跟踪2条指令的行为而添加了4条指令,那么会使执行时间增至原来的3倍。相反,如果向一条已包含200条指令的执行路径添加4条指令,那么只会使执行时间增加2%。如果addOne由更多的复杂计算组成,那么添加Trace的影响几乎可以忽略不计。

同样,这与内联很类似。内联对超重量级函数的影响是可以忽略不计的。内联只有在针对小型函数时才有比较大的影响:对这些小型函数来说,调用和返回开销占支配地位。十分适合内联的函数却正好是不适合跟踪的。综上所述,Trace对象不宜添加到小型的、频繁执行的函数中。

1.2 要 点

- 对象定义会以对象的构造函数和析构函数的形式引起隐性的执行。我们之所以称其为"隐性执行"而不是"隐性开销",是因为对象的构造和清除并不总是意味着开销。如果构造函数和析构函数执行的计算总是必须的,那么可以把它们看作高效代码(内联会减轻调用和返回的开销)。正如我们所看到的那样,构造函数和析构函数并不总是具有如此"完美的"特点,而且它们会产生明显的开销。某些情况下,构造函数(或析构函数)所执行的计算是无用的。同时我们要指出,这不仅是一个C++语言的问题,更是一个设计问题。然而,这种现象在C中比较少见,因为C不支持构造函数和析构函数。
- 正是因为通过引用传递对象而不能保证良好的性能、故避免对象复制的确有所帮助,假如我们不在第一处创建和清除对象的话,这会更有益。
- 如果计算结果没什么用,就不要在其上浪费精力。当关闭跟踪时,string成员的创建就是一种无谓的开销。
- 在设计灵活性方面不要追求世界纪录。只要自己的设计针对当前问题的范围而言已足够灵活就可以了。与string对象相比,char指针有时可完成同样的工作,而且更为有效。
- 内联消除了在对小型的、频繁调用的函数进行调用时所产生的函数调用开销。内联Trace的构造函数和析构函数使得消除Trace的开销成为一件很容易的事情。

第 2 章

构造函数和析构函数

提高 C++ 性能的编程技术

理想情况下，不应该用一整章的篇幅专门讨论与构造函数和析构函数有关的性能。在这种理想情况下，构造函数和析构函数是没有开销的。它们只执行必要的初始化和清除，一般的编译器都会内联它们。以下 C 代码：

```
{
    struct X  x1;
    init ( &x1 );
    ...
    cleanup ( &x1 );
}
```

在 C++ 中如下：

```
{
    X  x1 ;
    ...
}
```

二者的开销是相同的，至少在理论上是这样。如果由此进入软件开发的世界，则会发现实际中稍有不同。我们经常会遇到继承和合成实现，就所论问题域而言，它们十分灵活和通用。它们只执行少量计算或根本不需要计算。在现实中发现，与继承和合成相关联的性能开销是不足为奇的。这是在较大问题上的局限性表现，即代码重用和性能之间的基本冲突。继承和合成与代码重用有关。通常，代码重用会对某种特定场合下不是真正需要的东西进行计算。当函数所做的工作比需求的更多时，每次调用函数都会受到性能上的打击。

2.1 继　承

继承和合成是面向对象设计中把类联系在一起的两种方式。在本节中我们要讨论继承的设计与构造函数和析构函数的开销之间的关联。我们通过一个示例开始本次讨论：线程同步构造的实现。在多线程应用程序中*，经常需要提供线程同步以限制对共享资源的并发访问。线程同步构造以各种方式出现，最常见的 3 种方式是：信号、互斥和临界区。

信号提供了限制性并发，它允许多个线程访问共享资源。线程的数量小于一个给定的最大数量。当并发线程的最大数量设为 1 时，我们用一个称为互斥（MUTual Exclusion）的特殊信号来表示。互斥通过在任一时间只允许一个线程对共享资源进行操作来保护共享资源。典型情况下共享资源对遍布于应用程序的各个代码段都是可操作的。

以一个共享队列为例，该队列中的元素个数由 enqueue ( )和 dequeue ( )例程共同管理。显然，对元素个数的修改不能由多个线程同时进行。

```
Type& dequeue ( )
{
    get_the_lock (queueLock) ;
    ...
    numberOfElements - - ;
    ...
    release_the_lock (queueLock ) ;
    ...
}
void enqueue (const Type& value )
{
    get_the_lock (queueLock) ;
    ...
    numberOfElements + + ;
    ...
    release the_lock( queueLock) ;
}
```

如果 enqueue ( )和 dequeue ( )能并发地修改 numberOfElements，那么很容易导致

---

* 有关多线程编程的基本概念和术语的更多信息，可以参阅第 15 章。

numberOfElements 包含一个错误值。对此变量的修改必须以原子操作完成。

使用互斥锁的最简单的应用程序以临界区的形式出现。临界区是指在某一时间只能由一个线程执行的单个代码段。要实现互斥，线程在进入临界区之前就必须竞争锁。成功获取锁的线程进入临界区。在退出临界区时*，该线程释放锁以便使其他线程进入。

```
get_the_lock ( CSLock ) ;
{    // Critical section beqins
     …// Protected computation
}    // Critical section ends
release_the_lock (CSLock );
```

在 dequeue ( )示例中，检查代码以及验证每一个锁操作都有与之相匹配的解锁操作是相当容易的。在实践中我们看到过包括几百行代码的例程，这些代码中包含多条返回语句。第一个问题就是由返回引起的。如果在流程中的某处获得了锁，那么在执行任何一条返回语句之前必须把锁释放。可以想象这是很难维护的，稳定性方面的漏洞随时可能出现。大型工程可能有几十人在编写代码和调试漏洞。如果在有 100 行代码的例程中添加一条返回语句，则可能忽略先前曾获得锁的情况。第二个是异常问题：如果在持有锁的时候抛出一个异常，则必须捕获该异常并手工释放锁。但这种做法不怎么高明。

C++ 对这两个难题提供了精彩的解决方案。当对象到达定义其作用域的结尾处时，会自动调用它的析构函数。可以利用自动调用的析构函数解决锁的维护问题。把锁封装在对象内部并让构造函数获得锁。然后析构函数将自动释放锁。如果在由 100 行代码组成的函数中定义这样的对象，就不必担心多条返回语句了。编译器在每条返回语句之前插入对带锁的析构函数的调用，这样总能够提前释放锁。

使用构造函数和析构函数去获取和释放共享资源[ES90 Lip96C]会产生如下所示的锁类实现：

```
class Lock {
public:
    Lock (pthread_mutex_t & key)
    : theKey (key) { pthread_mutex_lock (&theKey );
    }
        ~Lock ( ) { pthread_mutex_unlock (&theKey ); }
```

* 必须指出的是 Win32 对临界区的定义与我们所说的有一点区别。在 Win32 中，临界区由一个或多个不同的代码段组成，在任一时刻只能执行其中的一个。在 Win32 中，临界区与互斥的区别是临界区只限于单个进程，而互斥锁能够越过进程的边界并使在不同进程中运行的线程同步。我们所用术语与 Win32 所用术语之间的矛盾不会影响对 C++ 的讨论。指出这点只是为了避免发生混淆。

```
private:
    pthread_mutex_t &theKey;
};
```

编程环境通常提供多种风格的同步构造。风格的不同取决于以下几方面：

- 并发级别：信号允许多个线程共享资源，线程的数量小于给定的最大数量。互斥只允许一个线程访问共享资源。
- 嵌套：某些构造允许线程在已获得某锁的情况下再次获得该锁。而另外一些构造则会在这种锁嵌套的情况下发生死锁。
- 通知：有一些同步构造会在资源变为可用时通知所有正在等待的线程。由于除了一个线程之外，其他所有线程被唤醒后会发现它们不够快，因为资源已经由其他线程获得，所以这种方式是低效率的。更为有效的通知方案是仅唤醒一个正在等待的线程。
- 读/写锁：允许多个线程读取一个受保护的值，但是只允许一个线程修改它。
- 内核/用户空间：某些同步构造只在内核空间中有效。
- 进程间/进程内：一般情况下，在同一进程的线程间进行同步要比在不同进程的线程间进行同步更为有效。

尽管这些同步构造在语义和性能上有显著区别，但是它们共享相同的锁/解锁协议，这是非常诱人的。为此，可以把这种相似性转化到基于继承的锁类层次中，这些锁类来源于相同的基类。在所使用的某个产品中，一开始我们发现了一种实现大致如下所示：

```
class BaseLock {
public:
    //  (The LogSource object will be explained shortly)
    BaseLock (pthread_mutex_t &key, LogSource &lsrc) { };
    Virtual ~BaseLock( ) { };
};
```

显然，BaseLock 类没有太多操作。它的构造函数和析构函数都是空的。BaseLock 类的目的是要成为各种要从它派生出来的锁类的根类。各种不同风格的锁类自然要以不同的 BaseLock 子类来实现。其中一个派生类是 MutexLock：

```
class MutexLock  :public BaseLock {
public:
    MutexLock ( pthread_mutex_t &key, LogSource &lsrc);
    ~MutexLock ( );
private:
```

```
    pthread_mutex_t &theKey;
    LogSource &src;
};
```

MutexLock 的构造函数和析构函数的实现如下所示：

```
MutexLock :: MutexLock (pthread_mutex_t & aKey , const LogSource & source)
    :BaseLock (aKey , source) ,
    theKey (aKey),
    src (source)
{
    pthread_mutex_lock ( &theKey );
# if defined (DEBUG)
    cout<<"MutexLock"<<&aKey<<" create at "<<src.file ( ) <<
    "line"<<src.line ( )<<endl;
# endif
}
    MutexLock :: ~MutexLock ( )  // Destructor
{
    pthread_mutex_unlock (&theKey);
#  if defined (DEBUG)
    cout<<"MutexLock "<<&aKey<<" destroyed at "<<src.file ( )<<
    "line"<<src.line ( )<<endl;
#endif
}
```

MutexLock 的实现使用了 LogSource 对象，此前我们还没有讨论过它。LogSource 对象的用途是捕获文件名和该对象创建时的源代码行号。在记录错误和跟踪信息时，有必要指定信息源的位置。C 程序员会把(char *)用于文件名，把 int 用于行号。我们的开发人员则把它们全部封装在 LogSource 对象中。同样，我们有一个什么也不做的基类，然后是一个更为有用的派生类：

```
class BaseLogSource {
public:
    BaseLogSource ( ) { };
    virtual ~BaseLogSource ( ) { };
};
class LogSource : public BaseLogSource {
public:
```

```
    LogSource ( const char * name, int num) : filename (name),
        LineNum (num) { };
    ~LogSource ( ) { };

    char * file ( );
    int line ( );
private:
    char * filename;
    int lineNum;
};
```

这样就创建了 LogSource 对象并将之作为参数传递给 MutexLock 对象的构造函数。LogSource 对象捕获在得到锁时的源文件和行号。在调试死锁时这条信息会派上用场。

假设 sharedCounter 是一个整型变量，它可以由多个线程访问，并且需要序列化。我们通过在局部范围内插入一个锁对象来提供互斥：

```
{
    MutexLock myLock (theKey , LogSource (__FILE__ , __LINE__ ));
    sharedCounter + + ;
}
```

MutexLock 和 LogSource 对象的创建同时也引发了对它们各自基类的调用。这一小段代码调用了如下几个构造函数：

- BaseLogSource
- LogSource
- BaseLock
- MutexLock

sharedCounter 变量的值递增后，我们到了作用域的结尾，在此引发了几个相应的析构函数：

- MutexLock
- BaseLock
- LogSource
- BaseLogSource

合计起来，对共享资源的保护共用到 8 个构造函数和析构函数。重用和性能之间的矛盾是一个不断出现的主题。如果我们放弃所有这些对象，而去开发一个手工版本，通过该版本只做我们所需要的事情而不做其他任何事情，以便使范围缩小，那么计算一下其代价将是很有趣的。也就是说，只在更新 sharedCounter 的时候锁定：

```
{
    pthread_mutex_lock (&theKey);
    sharedCounter++;
    pthread_mutex_unlock( &theKey);
}
```

一眼便可以看出后面的版本比前面的版本更为有效。我们的面向对象设计付出了附加指令的代价。那些指令全部专门用于创建和清除对象。需要担心那些指令吗？这要根据上下文来定：如果是在一个性能要求严格的流程里，那么我们就应该这样。在一些特定情况下，如果总的计算开销很小且执行那些指令的代码段被调用得足够频繁，那么那些附加指令的代价就变得十分显著。我们所关心的是浪费的指令被所有计算的总指令数去除所得的比率。我们刚才所述的代码示例取自一个网关实现，它用来把数据包从一个通信适配器路由导向到另一个上去。这是一个很关键的执行路径，包含大约5 000条指令。MutexLock对象在该路径中要用很多次。累计起来会形成足够大的指令开销，使之占到全部开销的10%，这是非常显著的。

如果打算在性能要求严格的应用程序里使用C++和面向对象的设计，那么我们无法提供如此的奢侈品。在能够提供一个基于C++的方案之前，我们会很快指出设计中明显的过分行为。如果临界区就像一条语句的整型变量递增那么简单，那么为什么还需要所有这些对象工具呢？使用锁对象的好处如下：

- 包含多个返回点的复杂子程序的维护
- 从异常中恢复
- 锁操作中的多态
- 日志记录的多态

所有这些益处在我们的示例中都不是十分重要的。临界区有定义清晰的惟一出口并且整型变量的递增操作不会抛出异常，锁操作和日志记录中的多态同样是一些可以很容易地弃之不管的东西。有趣的是，正如代码段所显示的那样，程序员实际上是在熟练地做这件事情，这说明对象的构造和清除的开销被严重地忽略了。

如果在一个复杂的例程里锁对象的使用确实是有意义的，那么结果又会怎样呢？我们仍然要减少它的开销。首先考虑LogSource对象。那条信息耗费了4次函数调用：基类和派生类的构造函数和析构函数。这种奢侈品是我们在这种环境里所不能提供的。通常，在讨论C++性能时，内联用来作为一种弥补方法。尽管内联在此有所帮助，但是它并没有解决问题。在最理想的情况下，内联将为所有的构造函数和析构函数消除函数调用开销。尽管这样，LogSource对象仍然会强行带来一些性能开销。第一，它是MutexLock构造函数的一个附加参数。第二，对MutexLock的LogSource指针成员有一个分配操

作。此外，创建 LogSource 对象时，需要另外一些指令来建立其虚表指针。

在性能要求严格的执行路径中，提倡一种常识性的折中方法。即丢弃一些不重要的功能，以此换取有价值的性能。LogSource 对象必须这样做。在构造函数里，成员数据字段的分配耗费了少量指令，即使是内置类型也是如此。每个成员数据字段的开销也许不大，但是它们的开销加起来却很大。随着构造函数所初始化的数据成员数量的增多，它会相应增加。

使用 LogSource 对象的代码是用 # ifdef DEBUG 括弧括起来的，这个事实更进一步地提供了证据，说明该对象的使用并不是必需的。DEBUG 编译标志只用于开发测试阶段；发布给客户的代码是在关闭 DEBUG 的情况下进行编译的。在生产环境下执行时，我们为 LogSource 对象付出了代价，却从不会去使用它，这便是纯粹的开销。应该在其所有剩余部分使用 # ifdef 来彻底地消除 LogSource。这包括消除 MutexLock 的指针成员和构造函数参数。对 LogSource 对象局部使用 # ifdef 是开发作风散漫的一种表现。这不是十分重要的，但却是摆脱散漫的 C++ 开发作风的机会。

下一步是消除锁类层次中的 BaseLock 根类。对于 BaseLock 来说，它不提供任何数据成员，并且除了构造函数原型之外，不提供任何有意义的接口。关于 BaseLock 对整个类设计的作用是存在争议的。即使是内联已经减少了调用开销，但是 BaseLock 的虚析构函数还是会导致在 MutexLock 对象中设置虚表指针的开销。节省一次赋值也许算不了什么，但是节省每一个小的开销将会有较大帮助。内联其余的 MutexLock 构造函数和析构函数将消除这两个函数的调用开销。

把消除 LogSource 类和 BaseLock 类与内联 MutexLock 构造函数和析构函数结合使用，将会显著地削减构造数量。这样将会产生几乎和 C 手工编码效率一样的代码。编译用内联 MutexLock 产生的代码将大致等价于如下伪代码：

```
{
    MutexLock :: theKey = key;
    pthread_mutex_lock ( &MutexLock :: theKey);
    sharedCounter + + ;
    pthread_mutex_unlock (&MutexLock :: theKey);
}
```

以上 C++ 代码段几乎等价于 C 手工编码，同时我们假定它具有与之相同的效率。如果真是这样，那么对象锁便可以在不损失效率的情况下提供 C++ 新增加的强大功能。为了验证我们的假设，在此对 3 种互斥锁的实现进行测试：

- 直接调用 pthread_mutex_lock ( ) 和 pthread_mutex_unlock ( )
- 不从基类继承的独立互斥对象

· 从基类继承的互斥对象

在第一次测试中，我们只是用 pthread_mutex_lock ( ) 和 pthread_mutex_unlock ( ) 调用对来使用共享资源：

```
int main ( ) // Version 1
{
    ...
    // Start timing here
    for (i = 0;I<1000000;i + + ) {
        pthread_mutex_lock (&mutex );
        sharedCounter + + ;
        pthread_mutex_unlock (&mutex);
        }
    // Stop timing here
    ...
}
```

在版本 2 中，我们使用锁对象 SimpleMutex，在构造函数中加锁，在析构函数中解锁：

```
int main ( )            //Version 2
{
    ...
    // Start timing here
    for (i = 0;i<1000000;i + + ) {
        SimpleMutex m (mutex);
        sharedCounter + + ;
        }
    // Stop timing here
    ...
}
```

SimpleMutex 的实现如下：

```
class SimpleMutex          // Version two. Standalone lock class.
{
public:
    SimpleMutex ( pthread_mutex_t & lock ) :myLock (lock ) {acquire ( );}
    ~SimpleMutex {release ( );}
private:
    int acquire ( ) {return pthread_mutex_lock (&myLock);}
```

```
    int release ( ) {return pthread_mutex_unlock (&myLock);}
    pthread_mutex_t & myLock;
};
```

在版本 3 中加入了继承：

```
class BaseMutex      // Version 3. Base class.
{
public:
    BaseMutex ( pthread_mutex_t & lock) { };
    Virtual ~BaseMutex ( ) { };
};
class DerivedMutex  :public BaseMutex      // Version 3.
{
public:
    DerivedMutex (pthread_mutex_t &lock)
    : BaseMutex ( lock),myLock (lock) {acquire ( );}
    ~DerivedMutex ( ) {release ( );}
private:
    int acquire ( ) {return pthread_mutex_lock (&myLock); }
    int release ( ) {return pthread_mutex_unlock (&myLock);}
    pthread_mutex_t & myLock;
};
```

在测试循环中，我们用 DerivedMutex 代替 SimpleMutex：

```
int main ( )      // Version 3
{
    ...
    // Start timing here
    for ( i = 0;i<1000000;i + + ) {
    DerivedMutex m (mutex);
    sharedCounter + + ;
    }
// Stop timing here
...
}
```

运行 100 万次测试循环的计时结果验证了我们的假设。版本 1 和版本 2 在 1.01s 内执行完毕。然而版本 3 却花费了 1.62s。在版本 1 中，我们直接采用互斥调用——再没

有比这更为有效的办法了。比例的意义在于使用独立的对象根本不需要任何性能代价。构造函数和析构函数被编译器内联，从而使这种实现达到了最高效率。然而我们却为继承付出了明显的代价。基于继承的锁对象(版本 3)使性能降低了大约 60%(如图 2.1 所示)。

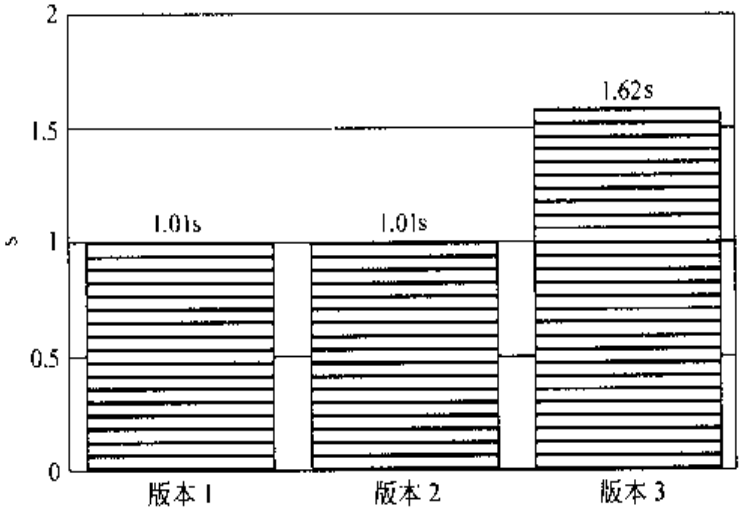


图 2.1 示例中的继承开销

创建和清除对象时常会导致性能的损失。在继承层次中，创建对象将引起其先辈的创建。清除对象也是如此。其次是与对象相关联的开销直接与派生链的长度和复杂性有关。创建对象(以及其后的清除对象)的数量与派生的复杂性成比例。

这并不是说继承从根本上就是性能的障碍。我们必须分清全部计算开销、需求开销和计算损失。全部计算开销是一次计算中所执行指令的总和。需求开销是为得到计算结果所必需的指令构成的子集。这部分计算是必需的，其余部分即为计算损失。计算损失是通过可选的设计和实现能够消除的那部分计算。为了更为具体地说明问题，我们以 SimpleMutex 类为例进行说明：

```
class SimpleMutex
{
public:
    SimpleMutex ( pthread_mutex_t & lock ) : myLock (lock ) {acquire ( );}
    ~SimpleMutex (release ( );}
private:
    int acquire ( ) {return pthread_mutex_lock (&myLock);}
    int release ( ) {return pthread_mutex_unlock (&myLock);}
```

```
    pthread_mutex_t & myLock;
};
```

SimpleMutex 的构造函数隐含着以下各项计算开销：

- 初始化 myLock 成员
- 调用 acquire ( )方法
- 在 acquire ( )方法中调用 pthread_mutex_lock ( )

第三条是需求开销。不管设计选择如何，使用何种方式，为了锁定资源都需要调用 pthread_mutex_lock ( )。第一条设置 myLock 成员则是计算损失。是基于对象的设计方式迫使我们这样做的。内联 acquire( )调用的失败会导致额外损失，实际上，编译器通过内联来清除这种损失也难以抗衡。

因此，我们不能断言复杂的继承设计一定是坏的，它们也并不总是造成性能损失。可以肯定的是开销会随着派生树尺寸的变大而增加。如果所有的计算都是有代价的，那么这些都是需求开销。在实践中，继承层次似乎并不完善*。在这种情况下，它们可能会导致计算损失。

2.2 合　　成

像继承一样，合成引入了与对象创建和清除有关的类似性能问题。在创建(或清除)对象时，同时必须创建(或清除)它所包含的成员对象。例如，上一章讨论的 Trace 对象包含一个类 string 的对象。

```
class Trace
{
    ...
    string theFunctionName;
};
```

创建 Trace 类型的对象时，构造函数会调用 string 的构造函数来初始化 string 数据成员。这种行为是递归性的：如果类 A 包含一个类 B 成员，而 B 又包含一个 C 成员，则 A 的构造函数就会调用 B 对象的构造函数。以此类推，B 又会调用 C 对象的构造函数。由于在合成(包容)层次的每一层都可能有多个属性，所以合成层次会产生一个树形结构，而不是一个简单的列表。因此合成的全部开销与合成树的尺寸有关，而合成树有可能变得非常巨大。此外，必须强调的是“代价”并不总是意味着“开销”。如果程序需要所包含对

* 在此，软件完善是指您所需的计算。即所有您所需要的，除此之外什么也不需要。

象的成熟功能，那么就不存在开销问题。这刚好是您需要做的。另一方面，Trace 示例（参见前一章）是一个“代价”与“开销”实际上相符的例子。表示正在被跟踪的函数的名字并不需要 string 对象的强大功能，因为从不用该对象做任何非常复杂的事情。因而可以很轻松地用 char 指针替换之。构造一个 char 指针比构造一个 string 对象廉价多了。可悲的是，一些 C++ 编程团体似乎没有意识到使用过于复杂的 C++ 数据类型相当于散漫的编程风格，这将导致不必要的过分行为，而不是坚持了最初的原则：“尽可能使用最简单的解决方案。”

可以想象一下巨大的派生和合成树所形成的复杂层次，会使构造和清除一个对象的代价直线上升。如果在性能敏感的流程中很有可能出现这种层次结构的话，那么在设计阶段要在思想上对此加以注意。

被包含对象的创建和清除是另一个值得注意的问题。在创建（或清除）被包含对象时无法阻止子对象的创建（或清除），因为这是编译器自动强加的。再看一下先前的 Trace 示例：

```
class Trace {
public:
    Trace ( const char * name);
    ...
private:
    string theFunctionName;
};
```

Trace 对象的创建将会创建一个 string 子对象。类似地，Trace 的析构函数将会清除该 string 子对象。这种行为在该实现中是自动的，无法阻止。为了更好地控制子对象的创建和清除，可以用指针来代替它：

```
class Trace {
public:
    Trace ( const char * name);
    // ...
private:
    string * theFunctionName;
};
```

现在就可以控制 string 对象的创建和清除了。我们仍然可以选择实现完全的初始化。这种方式将创建一个新的 sring 对象并设置一个指向它的指针：

```
Trace :: Trace (char * name) :theFunctionName (new string (name))
```

```
{
    ...
}
```

也可以选择部分初始化。我们为这些指针分配一个无效值，这意味着在使用这些对象之前必须创建它们。我们希望的使用模式是：通常情况下跟踪是关闭的，并且所创建的 Trace 对象是很少使用的。因此使创建和清除的开销达到最小是极为重要的：

```
Trace :: Trace (const char * name) : theFunctionName (0)
{
    if (traceIsActive )
        theFunctionName = new string (name);
        ...
    }
}
```

由于采用这种使用模式，第一种和第二种方式十分低效。因为通常情况下跟踪是关闭的，所以构造 string 对象的努力是没有价值的。由于向一个指针赋值 0 比构造一个新对象要廉价得多，所以第三种方式是最有效的。注意是使用模式决定了哪种方式最有效。如果使用模式是跟踪经常打开，那么第一种方式(包含子对象，而不是指针)将会是最有效的。在 Trace 对象中嵌入 string 对象要更为有效，这是因为它将占用堆栈内存而不是堆内存。分配与释放堆内存的代价要昂贵得多。基于堆栈的内存是在编译时分配的，而在函数调用返回的堆栈清除阶段释放。

2.3 缓式构造

性能优化经常会打破竞争因素之间的微妙平衡。这可能就是为什么说背离性能最大化反而是优化的原因吧。性能优化经常需要牺牲一些其他软件目标，诸如灵活性、可维护性、成本和重用之类的重要目标都经常为性能让步。解决了性能问题(或者说生成了高质量的代码)而不危及其他软件开发目标的情况是很少见的。有时我们会幸运地在没有付出代价的情况下消除简单的代码错误，从而产生更高的性能。我们要说明的第一种优化便是这样一个例子。该例子从代码段中消除了对象的创建(和随后的清除)，在这些代码段中，这些对象从来不会用到。

如果打算在性能要求苛刻的代码段中实例化一个对象，那么应该考虑开销的因素。不过，最廉价的对象是我们从不对其进行实例化的对象。

在 C、Pascal 和其他常用语言中，必须在代码块的开头定义数据类型。因此我们养成

了在例程的开头定义例程所需变量的习惯：

```
void myFunction ( )
{
    int i;
    int j;
    ...
    compute (i,j);
    ...
    return;
}
```

在C++中，自然而然地预先定义所有对象的习惯是一种浪费。这样可能会创建一些直到最后都没有用到的对象。这种情况在实际中是会发生的。在我们的C++代码中，有一个类DataPacket，它分配和回收空闲内存：

```
class DataPacket
{
public:
    DataPacket ( char * data, int sz) :size (sz) {
        buffer = new char [sz];
        ...
        }
    ~DataPacket ( ) {
        delete [ ] buffer;
        ...
        }
    ...                    // other public member functions
private:
    char * buffer;
    ...                     // other private members
};
```

在一定程度上由于内存分配和回收的原因，DataPacket在创建和清除方面的开销很大。它占用了高达400条的指令，在我们当时的环境中这是非常显著的。而且它用于性能要求较高的程序段，该程序段用于把数据从一个适配器路由导向到另一个适配器上去：

```
void routeData (char * data, int size)
{
    DataPacket packet (data, size);
```

```
    bool direction = get_direction ( );
    ...          // Some computation
    if (UPSTREAM = = direction) { // data going upstream
        computeSomething (packet);
        }
    else     {          // data going downstream
        ...             // packet is not used in this scope.
        }
}
```

packet 对象当且仅当数据发往上游时才会使用，这种情况占 50%。数据发往下游时，根本用不到 packet 对象。然而 packet 对象却在作用域的开始就无条件地创建了。在一半的时间内，这是一种完全浪费的计算周期。

正如您的想象，这里的解决方案很简单，就是在真正需要时才创建 packet 对象。对 packet 的定义应该移到使用它的块内，即数据发往上游时：

```
void routeData (char * data, int size)
{
    ...                                     // Delete definition of packet
    bool direction = get_direction ( );
    ...
    if (UPSTREAM = = direction) {           // data going upstream
        DataPacket packet (data, size);     // Add definition of packet
                                            // here ...
        computeSomething (packet);
        }
    else     {          // data going downstream
        ...             // packet is not used in this scope.
            }
}
```

尽管自从有了 C++ 语言起，C++ 专家就一直呼吁要把变量的创建延迟到第一次使用时，但是这种编码错误却仍会在实际产品代码中发现。尽管 C++ 和 C 之间的兼容性被吹捧为 C++ 的优点之一，但是兼容性却在这一方面产生了风格上的不连续。在 C++ 中坚持现在已过时的 C 声明语法会产生明显的性能开销。不幸的是实际中经常发生这种细小的错误。C++ 之所以比 C 更好，部分原因在于它能够延迟变量的创建。

在 C 中虽然也允许在任何块的开头定义变量，但这是没有什么价值的。C 程序员趋向于在函数块的开头定义所有变量是有充分根据的，因为 C 变量定义没有运行时的开

销。而C++却不是这样，在C++中必须注意对象定义的位置。

2.4 冗余构造

沿着简单但有严重编码错误的路线，这里给出另一个属于纯粹开销的例子。它两次创建了被包含对象[Mey97 item 12]、[Lip91]。

类Person包含类string的一个对象：

```
class Person {
public:
    Person (const char * s) { name = s; } // Version 0
    ...
private:
    string name;
};
```

请考虑Person的默认构造函数的如下实现：

```
Person ( const char * s) { name = s; }
```

在Person构造函数部分执行之前必须先实例化Person对象。所包含的string对象没有出现在Person :: Person(char *)初始化列表中(本例中没有该列表)。因此，为了正确地初始化string成员对象，编译器必须插入一个调用以调用string的默认构造函数。随后执行Person的构造函数部分。语句name = s;把char型指针s赋给左边的sring型对象。执行这个赋值操作要调用string :: operator = (const char *)。赋值操作所执行的计算实际覆盖了先前初始化时调用默认构造函数的结果。包含在默认string构造函数中的计算努力被彻底扔掉了。

许多优化需要某种平衡。经常为了清晰、简单、重用或其他衡量标准而牺牲速度。然而在本例中，优化根本不需要任何牺牲。可以重写Person的构造函数以找回失去的速度而不牺牲任何品质。通过指定string成员name的显式初始化可以达到这一点。

```
Person :: Person (const char * s) : name (s ) { } // Version 1.
```

除了改进了性能之外，这个构造函数将产生完全相同的Person对象。我们所做是对string :: string (const char *)的惟一一次调用，以便用s初始化name。通过以下循环我们对性能差别进行了测试：

```
for ( i = 0; i<MILLION; i + + ) {
```

```
Person p ("Pele");
}
```

图 2.2 中的图表对版本 0(隐性初始化加显式赋值)和版本 1(只有显式初始化)的执行时间进行了比较。

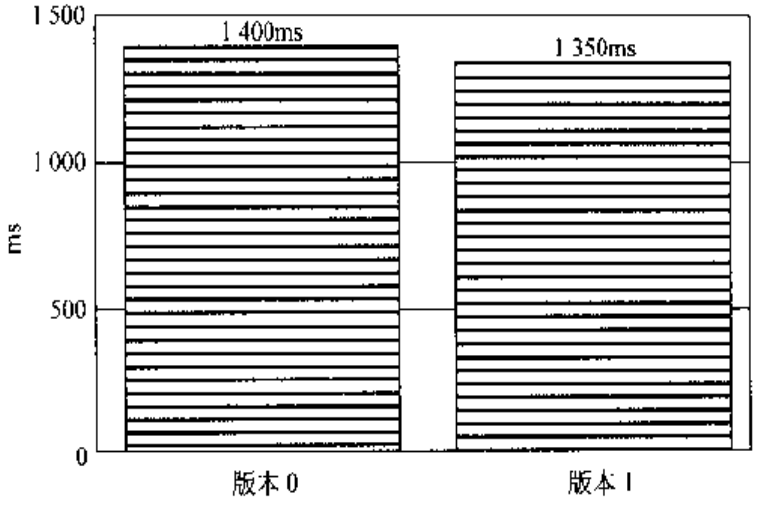


图 2.2 隐性初始化的开销在这种特定情况下可以忽略不计

幸运的是,默认的字符串构造函数是非常廉价的,从而使这种低效几乎可以忽略不计。但您不会总是这么幸运,因此不要指望它。

假设由于某种原因您不想使用标准的字符串实现,而编写自己的代码。我们把它称作 SuperString:

```
class SuperString
{
public:
    SuperString (const char * s = 0);
    SuperString (const SuperString & s);
    SuperString & operator = (const SuperString & s);

    ~SuperString ( ) {delete [ ] str;}
private:
    char * str;
};
inline
SuperString :: SuperString ( const char * s)
    : str (0 )
```

```
{
    if (s ! = 0 ) {
        str = new char [ strlen (s ) + 1];
        strcpy (str , s);
        }
}
inline
SuperString :: SuperString (const SuperString &s )
    : str (0)
{
    if (s.str) {
        str = new char[strlen (s.str) + 1];
        strcpy (str,s.str);
        }
}
SuperString& SuperString :: operator = (const SuperString& s)
{
    if (str ! = s.str)    {
        delete [ ] str;
        if (s.str) {
            str = new char [strlen (s.str) + 1];
            strcpy (str,s.str);
            }
        else str = 0;
        }
    return * this;
}
```

此外，您已决定使用 SuperString 替换 Person 实现中的 string 成员：

```
class Person {
public:
    Person (const char * s) { name = s; }        // Version 2
    ...
private:
    SuperString name;
};
```

由于没有提供接收 char 型指针参数的 SuperString 赋值操作符，所以赋值操作符要

求用 SuperString 对象引用作为参数。因此，在 Person 构造函数中的语句：

```
name = s;
```

会引发一个临时 SuperString 对象的创建。编译器将通过调用相应的 SuperString 构造函数把 char 型指针 s 转化成 SuperString。我们用伪代码表示如下 Person :: Person (char *)实现的转化：

```
Person :: Person (const char * s)
{
    name.SuperString :: SuperString ( ) ;          // Constructor; Initialize
                                                   // member "name".
    SuperString _temp;                             // Temporary object.
    _temp.SuperString :: SuperString (s);          // Construct a SuperString
                                                   // object from "s".
    name.SuperString :: operator = (_temp);        // Assign _temp to "name".
    _temp.SuperString :: ~SuperString ( );         // Destructor for temporary object.
}
```

通过对 SuperString 赋值操作符的调用把临时 SuperString 对象(temp)的内容赋予成员 name。随后调用 SuperString 的析构函数清除该临时对象。

我们总共有两个 SuperString 构造函数，其中一个调用赋值操作符，另一个调用析构函数。这种实现的性能损失十分严重(见图 2.3)。

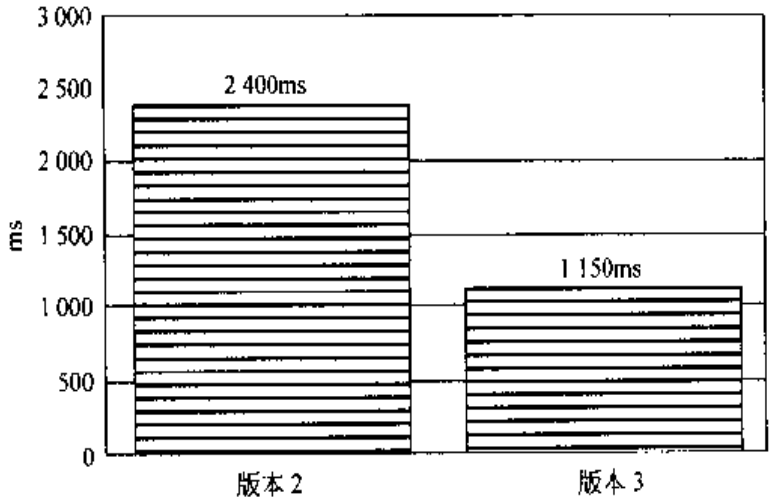


图 2.3 隐性初始化更有效的影响

版本 3 通过 SuperString 的显式初始化解决了这个问题：

```
Person :: Person (const char * s) : name (s) { }     // Version 3. Explicit
                                                     // initialization
```

版本2是在构造函数部分既使用隐性初始化又使用显式赋值的一种版本。版本3使用显式初始化但没有赋值操作。在我们自己编写的SuperString中，版本3要快2倍多。如果提供了一个接收char型指针参数的赋值操作符，那么性能损失将不会那么严重。那样会消除对临时SuperString对象得要求。版本3要稍快于使用编译器string实现的相应版本(版本1)，这一点是值得注意的。我们自己编写的SuperString不提供任何string类所提供的丰富功能。如果代码不必提供太多灵活性的话，那么代码就会运行得更快，这是经常遇到的情况。

2.5 要 点

- 构造函数和析构函数可以像手工编写的C代码一样有效。然而在实践中，它们经常包含着以多余计算形式出现的开销。
- 对象的创建(或清除)引发对父对象和成员对象的递归创建(或清除)。要当心复杂层次中对象的组合性使用。它们增加了更长的创建和清除开销。
- 要确保所编写的代码实际使用了所有创建的对象和这些对象所执行的计算。我们鼓励程序员进入他们所使用的类。这个建议不会受到OOP鼓吹者的欢迎。毕竟OOP鼓吹类的使用要像封装的黑匣子实体一样，不鼓励人们看到内幕。怎样平衡这两条相互对立的建议呢？由于这个问题与所处的环境有关，因此没有简单的答案。尽管对于80%的代码，黑匣子都能工作得十分完美，但它却可能对性能要求苛刻的另外20%造成很大破坏。它也是与应用程序相关的。有的应用程序重视可维护性和灵活性，而另一些应用程序则有可能把性能放在最为重要的位置。作为程序员，应当清楚到底想把哪个方面最大化。
- 对象的生命周期不是无偿的。对象的创建和清除会消耗CPU周期。除非打算要使用一个对象，否则就不要创建它。通常情况下，要把对象的创建推迟到要使用它的块中。
- 编译器在执行构造函数部分之前必须初始化被包含的成员对象。应该在初始化阶段完成成员对象的创建。这将降低随后在构造函数部分调用赋值操作符的开销。在某些情况下，这样也会避免产生临时对象。

第 3 章

虚 函 数

提高 C++ 性能的编程技术

编程语言发展的趋向是使编程工作更为简单，这是通过把一些负担从程序员身上转移到编译器、解释器、汇编语言或链接器上而实现的。程序正变得更容易开发、维护和扩展。随着发展而出现的问题是这种发展常常是一种总和为零的游戏，在某一方面的收获就意味着另一方面的失去。某些特定情况下，编程的进步经常转化为原有速度的损失。函数调用的动态绑定是 C++ 比 C 进步的方面之一。它把类型判断的任务从程序员转移给了编译器，这很好。但另一方面，它却会对开销造成负面影响，这正是我们所要研究的。

3.1 虚函数的构造

如果真的想避免使用虚函数，可以通过提供自己的类型解析代码来模拟动态绑定。假设您正在维护一个动物园动物的类层次[Lip91]，其中 ZooAnimal 是基类：

```
class ZooAnimal {
public:
    ...
    virtual void draw ( );
    int resolveType ( ) { return myType; }
private:
    int myType;
    ...
}
```

动物园中的其他动物都从 ZooAnimal 派生的。resolveType ( )方法将使您能够在运行时把一头熊与一只猴子区分开来。

```
class Bear : public ZooAnimal {
public:
```

```
    Bear (const char * name) : myName (name), myType (BEAR) { }
    Void draw ( );
    ...
};
```

每一只动物在其构造函数中都会设置相应的类型。

如果您要画出动物园中的所有动物，那么就要按照如下的代码行来结束[Lip91]：

```
void drawAllAnimals (ZooAnimal * pz )        // pointer to first animal in the list
{
for (ZooAnimal * p = pz; p ; p = p->next) {
    switch (p->resolveType ( )) {
    case BEAR:
        ( ( Bear * ) p) ->draw ( );
            break;
    case MONKEY:
        ( ( Monkey * ) p)->draw ( );
            break;
    ...                                  // Handle all other animals currently in the zoo.
    }
  }
}
```

这段代码维护起来是很头痛的。只要有动物离开了动物园，就不得不从 switch 语句中把它删除，而一旦有动物进入，也必须把它添加进去。虚函数的动态绑定使您可以避开这种依赖性。由于 ZooAniml :: draw ( )是一个虚函数，所以可以充分利用运行时动态绑定：

```
void drawAllAnimals (ZooAnimal * pz )    // pointer to first animal in the list
{
for (ZooAnimal * p = pz; p ; p = p->next) {
    p->draw ( );
    }
```

这段代码仍就能把一头熊与一只猴子区分开来。那它是怎样实现的呢？要便于后期绑定，我们就必须有一种在运行时解析虚函数调用的方法，这与编译时解析是相反的。如果类 X 定义了一个虚函数或是它派生于这样的类，那么就会由编译器为类 X 产生一个虚函数表(vtbl)。虚函数表拥有为该指定类所定义的所有虚函数的指针。每个类有一个虚函数表，该类的每个对象都有一个隐藏的指向该表的指针。之所以是隐藏的，是因为只有

编译器才知道 vptr 在对象内部的偏移量[Lip96]。编译器在对象的构造函数中插入代码以正确地初始化 vptr。

虚函数似乎会从以下几个方面造成性能损失：

- 必须在构造函数内初始化 vptr。
- 虚函数是通过指针间接调用的。我们必须先得到指向函数表的指针，然后访问正确的函数偏移量。
- 内联是编译时的选择。由于虚函数的类型判断发生在运行时，所以编译器不能内联虚函数。

如果公平地对待 C++，则不应该把头两条视为性能损失。不管以何种方式，即使您倾向于以前的方式而避免使用动态绑定，都将不得不付出代价。在构造函数中设置 vptr 的开销等价于在我们的 Bear 实现中初始化类型成员的开销。

```
class Bear : public ZooAnimal {
    ...
    Bear (const char * name) : myName (name), myType (BEAR) { }
    ...
};
```

第二条的开销等价于必须用来区分 Bear :: draw ( ) 和 Monkey :: draw ( )的 switch 语句逻辑的开销。

```
switch (p->resolveType ( )) {
case BEAR:
    ( ( Bear * ) p)->draw ( );
        break;
case MONKEY:
    ( ( Monkey * ) p)->draw ( );
        break;
... // Handle all other animals currently in the zoo.
}
```

这样，虚函数的真正代价只有第三条。正如 Meyers 在[Mey96]的第 24 款(Item 24)中所提到的：无法内联虚函数是虚函数最大的性能损失。

在某些情况也许可以在编译时决定虚函数的调用[Lip96]，但是这是一种例外情况(在第 8～10 章对此有更详细的讨论)。由于调用其函数的对象的类型不能在编译时决定，所以大多数虚函数调用要在运行时决定。无法由编译器决定的事实对内联造成了负面影响。内联是一种编译时选择，它需要知道特定的函数。如果在编译时无法决定要调

用哪个函数(像虚函数这种典型的情况),则无法对它进行内联。

评估虚函数的性能损失等价于评估无法内联该函数所造成的损失。这种损失没有固定的代价,而要依赖于函数的复杂性和它的调用频率。一方面是那些调用频繁的小函数。它们从内联中受益最大而现在却无法进行内联,因此将产生巨大的性能损失。另一方面是很少调用的复杂函数。在第8~10章将更详细地讨论内联及其性能影响。

当包含多重继承或虚继承时,对象的创建会涉及附加的代价。这种附加的代价来自于必须设置多个vptr以及所增加的虚函数调用间接程度。按照多重继承和虚继承设计的对象是一个有趣的问题,但是这种性能影响是微小的,在通常的代码中不会有显著的性能影响。在此不再对这些问题深入讨论。在"Inside the Object Model"[Lip96]中有对这些问题更为详细的讨论。

如果特定的虚函数给您带来了性能问题,应该怎么办呢?为了消除虚函数调用,必须允许编译器在编译时决定函数的绑定。通过对类选择进行硬编码或是将其作为模板参数来传递,您可以绕过动态绑定。下面通过一个具体例子进行讨论。

3.2 模板和继承

只能在运行时决定的虚函数调用将阻止内联。通常这会造成性能问题,该问题是必须解决的。函数的动态绑定是继承的结果。消除动态绑定的一种方法是用基于模板的设计来代替继承。在这种情况下,模板可带来更好的性能,这是因为它把决定的步骤从运行时推到编译时。对于我们所关心的问题,编译时是无偿的。

继承和模板的设计空间有些重叠。我们将讨论这样一个例子。

假设您想开发一个线程安全的字符串类,它可以由Win32环境[BW97]下的并发线程安全地管理。在该环境中,可以选择诸如临界区、互斥和信号之类的多种同步方案,我们只提几个。为了使用这些方案,您希望字符串类提供灵活性,在不同的时间可能由于某种原因,您更愿意选择其中的某一个而不是另一个[BW97]。继承是一种满足同步机制共性的合理选择。

Locker抽象基类将声明公共接口:

```
class Locker {
public:
    Locker ( ) { }
    virtual ~Locker ( ) { }
    virtual void lock ( ) = 0;
    virtual void unlock ( ) = 0;
```

```
}
```

CriticalSectionLock 和 MutexLock 将从 Locker 基类派生：

```
class CriticalSectionLock : public Locker { …};
class MutexLock : public Locker { …;}
```

由于您不想重新设计，所以选择从现有的标准 string 派生线程安全的字符串。其他设计选项是：

- 硬编码：可以从 string 派生出三个类：CriticalSectionString、MutexString 和 SemaphoreString，每个类实现其自身包含的同步机制。
- 继承：可以只派生出一个 ThreadSafeString 类，它包含指向 Locker 对象的指针。在运行时使用多态来选择特定的同步机制。
- 模板：创建一个由 Locker 类型参数化的基于模板的字符串类。

下面我们将更详细地讨论每一种设计选项。

3.2.1 硬编码

标准的 string 类将被当作基类。每个从它派生出的类支持一种特定的同步机制。例如 CriticalSectionString：

```
class CriticalSectionString : public string {
public:
    …
    int length ( );
private:
    CriticalSectionLock cs;
};
int CriticalSectionString :: length ( )
{
    cs.lock ( );
    int len = string :: length ( );
    cs.unclock ( );

    return len;
}
```

从 string 父类中得到实际字符串长度，但是要把它放到临界区中以保护计算的完整性。MutexString 和 SemaphoreString 可用类似的方式实现，它们分别使用互斥和信号。

这种设计选择有性能上的优势。即使 lock ( )和 unlock ( )方法是虚函数，它们也可由适当的编译器静态地解析。这三个线程安全的字符串中的每一个在编译时都被确定成一个使用特定同步方式的类。因此，编译器可以绕过动态绑定并选择使用正确的 lock ( )和 unlock ( )方法。更为重要的是，它允许编译器对所有这些调用使用内联。这种设计的缺点是需要为每种同步方式编写单独的字符串类，这将导致代码可重用性的降低。

3.2.2 继承

为每一种同步机制实现单独的字符串类是一件痛苦的事情。另一种选择是把同步选择放到构造函数的参数中：

```
class ThreadSafeString : public string {
public:
    ThreadSafeString (const char * s, Locker * lockPtr)
    : string (s), pLock (lockPtr) { }
    ...
    int length ( );
private:
    Locker * pLock;
}
```

length ( )方法实现如下：

```
int ThreadSafeString :: length ( )
{
    pLock->lock ( );
    int len = string :: length ( );
    pLock->unlock ( );
        return len;
}
```

这个类可以根据传递给构造函数的 Locker 指针使用所有可用的同步方案。可以像如下代码那样使用临界区：

```
{
    CriticalSectionLock cs;
    ThreadSafeString csString ("Hello", &cs);
    ...
}
```

也可以像以下代码那样选择使用互斥锁：

```
{
    MutexLock mtx;
    ThreadSafeString csString ("Hello",&mtx);
    ...
}
```

这种实现要比前一种简洁得多，但它确实导致了性能损失：对虚函数 lock ( )和 unlock ( )的调用只能在执行时决定，因此不能对它们使用内联。

3.2.3 模板

基于模板的设计集中了两方面的优点：重用和效率。ThreadSafeString 是作为一个模板实现的，其参数由 Locker 模板参数确定：

```
template <class LOCKER>
class ThreadSafteString : public string {
public:
    ThreadSafeString (const char * s) : string (s ) { }
    ...
    int length ( );
private:
    LOCKER lock;
}
```

length 方法的实现与前一种设计类似：

```
template <class LOCKER>
inline
int ThreadSafeString <LOCKER> :: length ( )
{
    lock.lock ( );
    int len = string :: length ( );
    lock.unlock ( );

    return len;
}
```

如果打算使用一种临界区保护方式，那么应该用 CriticalSectionLock 来实例化该模板：

```
{
    ThreadSafeString <CriticalSectionLock> csString = "hello";
    ...
}
```

也可以使用互斥：

```
{
    ThreadSafeString <MutexLock> mtxString = "hello";
    ...
}
```

这种设计也避免了对 lock ( )和 unlock ( )的虚函数调用。ThreadSafeString 声明在模板实例化时选择一种特定的同步类型。就像硬编码一样，这使得编译器可以确定虚函数调用并对它们进行内联。

正如您可以看到的那样，通过把计算从执行时推到编译时，并在编译时使用内联，模板就可以对性能产生积极的作用。如果把模板看成一种名称更好听的宏，那么在读过 Todd Veldhuizen 在"C ++ Gems"[Lip96C]中关于表达式模板的文章后，您将明确改变自己的想法。

3.3 要　　点

- 虚函数的代价起源于无法内联函数调用，其函数调用是在运行时绑定的。惟一潜在的性能问题是从内联(如果存在的话)获得的速度。对那些代价并非取决于调用和返回开销的函数来说，内联效率不是什么问题。
- 比起继承来，模板可提供更好的性能。它们把类型的确定推到了编译时，我们认为这种做法是没有什么代价的。

第 4 章

返回值优化

提高 C ++ 性能的编程技术

任何时候只要跳过了对象的创建和清除，就会获得性能上的收益。本章中将讨论一种通常由编译器实现的优化，它用于加速源代码，是通过对源代码进行转化并消除对象的创建来实现的。这种优化被称作返回值优化(RVO，Return Value Optimization)。在深入研究返回值优化之前，需要理解按值返回的工作方式。我们将通过一个示例进行讨论。

4.1 按值返回的构造

Complex 类实现复数的一种表示法：

```
class Complex
{
    // Complex addition operator
    friend Complex operator + (const Complex&, const Complex&);
public:
    // Default constructor.
    // Value defaults to 0 unless otherwise specified.
    Complex (double r = 0.0, double i = 0.0) : real (r), imag (i ) { }

    // Copy constructor
    Complex (const Complex& c ) : real (c.real), imag (c.imag) { }

    // Assignment operator
    Complex& operator = (const Complex& c);

    ~Complex ( ) { }
private:
    double real;
```

```
    double imag;
};
```

加法操作符按值返回一个 Complex 对象：

```
Complex operator + (const Complex& a, const Complex&b )
{
    Complex retVal;
    retVal.real = a.real + b.real;
    retVal.imag = a.imag + b.imag;
    return retVal;
}
```

假设 c1、c2 和 c3 是 Complex 型的，并且我们执行了如下操作：

```
c3 = c1 + c2;
```

那么我们是如何把 c1＋c2 的值放到 c3 中去的呢？编译器常用的一种技术是创建一个临时__result 对象，作为第三个参数把结果传递给 Complex :: operator＋ ( )。它是按引用传递的。因此编译器把

```
Complex& Complex ::operator + (const complex& c1, const Complex& c2)
{
    ...
}
```

重写为一个稍微不同的函数：

```
void Complex_Add (const Complex& _ _result,
                  const Complex& c1,
                  const Complex& c2)
{
    ...
}
```

这时原来的源代码：

```
c3 = c1 + c2;
```

就被转换成(伪代码)：

```
struct Complex _ _tempResult;              // Storage. No constructor here.
Complex_Add ( _ _tempResult,c1,c2);        // All arguments passed by reference.
```

```
c3 = _ _tempResult;                    // Feed result back into left-hand-side.
```

这种按值返回的实现提供了一种优化机会，这种优化是通过消除局部对象 RetVal（在 operator+（）内）和直接把返回值计算到_ _tempResult 临时对象中来实现的。这就是所谓的返回值优化。

4.2 返回值优化

在不进行任何优化的情况下，编译器为 Complex_Add（）产生的代码（伪代码）如下所示：

```
void Complex_Add ( const Complex& _ _tempResult,
                   const Complex& c1,
                   const Complex& c2 )
{
    struct Complex retVal;
    retVal.Complex :: Complex ( );                    // Construct retVal
    retVal.real = a.real + b.real;
    retVal.imag = a.imag + b.imag;
        _ _tempResult.Complex :: Complex (retVal);    // Copy - construct
                                                      // _ _tempResult
    retVal.Complex :: ~Complex ( );                   // Destroy retVal
    return;
}
```

编译器可以通过消除局部对象 retVal 并把它替换成__tempResult 来优化 Complex_Add( )。这就是返回值优化：

```
void Complex_Add ( const Complex& _ _tempResult,
                   const Complex& c1,
                   const Complex& c2 )
{
        _ _tempResult.Complex :: Complex ( );         // Construct _ _tempResult
        _ _tempResult.real = a.real + b.real;
        _ _tempResult.imag = a.imag + b.imag;
        return;
}
```

返回值优化消除了局部对象 retVal，从而省去了构造函数和析构函数的计算。

为了对整个讨论有一个定量的感觉，我们测试了 RVO 对执行速度的影响。我们编写了两种版本的 operator+，其中一个是经过优化的，另一个则没有。测试代码包含一百万次循环迭代：

```
int main ( )
{
    Complex a (1,0);
    Complex b (2,0);
    Complex c;
        // Begin timing here
    for( int i = 1000000; i>0; i- - ) {
        c = a + b;
        }
        // stop timing here
}
```

版本 2 没有经过返回值优化，其执行时间为 1.89s。版本 1 经过了返回值优化，执行得快多了，仅为 1.30s(如图 4.1 所示)。

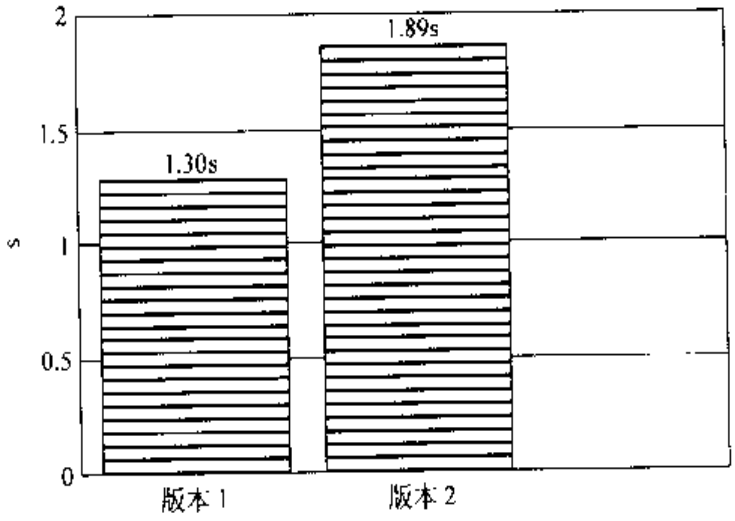


图 4.1 返回值优化(RVO)的加速作用

当然，编译器优化要维护原来计算的正确性。对于返回值优化来说，做到这一点并不总是很容易的。由于 RVO 不是必须的，所以编译器将不对复杂的函数执行 RVO。例如，如果函数有多个 return 语句返回不同名称的对象，那么就不会使用 RVO。如果想使用 RVO，则必须使用相同名称的对象。

经测试,有一种编译器拒绝对以下特定版本的 operator+使用 RVO:

```
Complex operator + (const Complex& a, const Complex&b )
// operator + version 1.
{
    Complex retVal;
    retVal.real = a.real + b.real;
    retVal.imag = a.imag + b.imag;
    return retVal;
}
```

然而该编译器确实对以下版本使用 RVO:

```
Complex operator + (const Complex& a, const Complex&b )
// operator + version 2.
{
    double r = a.real + b.real;
    double i = a.imag + b.imag;
    return Complex (r,i);
}
```

我们推测存在这种区别的原因在于:版本 1 使用了一个命名变量(retVal)作为返回值,而版本 2 使用的是未命名变量。版本 2 在返回语句中使用了构造函数调用但从未对它命名。也许这种特定的编译器实现选择了避免对已命名变量进行优化。

另外的一些证据加强了我们的推测。我们又测试了另外两个版本的 operator+:

```
Complex operator + (const Complex& a, const Complex& b ) // operator + version 3.
{
    Complex retVal (a.real + b.real, a.imag + b.imag);
    return retVal;
}
```

和

```
Complex operator + (const Complex& a, const Complex&b ) // operator + version 4.
{
    return Complex (a.real + b.real, a.imag + b.imag);
}
```

正像我们所推测的那样,该编译器对版本 4 使用了 RVO 而对版本 3 则没有使用。

此外，必须定义一个复制构造函数以“打开”返回值优化。如果有关的类没有定义复制构造函数，那么 RVO 就会悄悄地关闭。

4.3 计算性构造函数

在编译器无法使用 RVO 时，可以用计算性构造函数的方式“轻轻推它一下”（请参见 J. Shopiro 的 [Car92，Lip96]。）我们的编译器不对版本 1 使用 RVO：

```
Complex operator + (const Complex& a, const Complex& b)
// operator + version 1.
{
    Complex retVal;

    retVal.real = a.real + b.real;
    retVal.imag = a.imag + b.imag;
    return retVal;
}
```

该实现创建了一个默认的 Complex 对象并推迟了对其成员变量的设置，随后它使用输入对象的信息填充了成员数据。Complex retVal 对象的产生延续了几个步骤。计算性构造函数把这几个步骤概括到一个调用中去并消除了已命名的局部变量：

```
Complex operator + (const Complex& a, const Complex& b )  // operator +
                                                           // version 5.
{
    return Complex (a, b);
}
```

计算性构造函数使用版本 5 通过将其两个输入参数相加，从而创建了一个新的 Complex 对象：

```
Complex ::Complex (const Complex& x, const Complex& y)
    :real (x.real + y.real), imag (x.imag + y.imag)
{
}
```

现在，与加法运算符的版本 1 相比，编译器更有可能对版本 5 使用 RVO。如果您打算把同样的思想用于其他算术运算符，则必须增加第三个参数以区别分别用于加、减、乘和除的计算性构造函数的符号。这里给出对计算性构造函数的批评：它竭尽全力为效率

着想而引入了“别扭的”构造函数。我们对此争论的所持的相对观点是：在某种时间或地点，性能问题会压倒其他问题。这个问题是与上下文相关的，它没有所谓正确的答案。

4.4 要　点

- 如果必须按值返回一个对象，那么返回值优化将通过消除局部对象的创建和清除需求来提高性能。
- RVO 的应用取决于编译器实现的判断力。需要参考编译器的文档或通过实验来确实是否及何时使用 RVO。
- 通过部署计算性构造函数会有更多使用 RVO 的机会。

第5章

临时对象

提高C++性能的编程技术

在众多的性能问题中，并非所有的问题都具有相同的重要性。性能问题的显著程度直接与其本身的开销和在一般程序中出现的频率有关。在对虚继承的复杂性及其对执行速度的影响(微小)并不知情的情况下，写出高效率的C++代码是有可能的。另一方面，临时对象的产生无疑不属于这种潜在影响小的范畴。除非理解了临时对象的来源、它们的代价以及怎样在可能时消除它们，否则写出高效代码的可能性是很小的。

临时对象对新的C++开发人员来说可能是个很意外的事情，因为这些对象是编译器悄悄产生的，不出现在源代码中。只有训练有素的眼睛才能检查出将导致编译器在“幕后”插入临时对象的代码段。

接下来，我们将列举一些例子，在这些例子中，临时对象可能会出现在编译器所生成的代码中。

5.1 对象定义

假设类Rational声明如下：

```
class Rational
{
friend Rational operator + (const Rational&, const Rational &);
public:
    Rational ( int a = 0, int b = 1 ) : m (a), n (b) { }
Private:
    int m; // Numberator
    int n; // Denominator
}
```

我们可以用几种等价的方式实例化类型Rational的对象：

```
Rational  r1(100);                  // 1
Rational  r2 = Rational (100);      // 2
Rational  r3 = 100;                 // 3
```

只有第一种形式才能保证不管编译器如何实现，都不会产生临时对象。如果使用第二或第三种形式，则都有可能引入临时对象，这依赖于编译器的实现。以第三种形式为例：

```
Rational  r3 = 100;                 // 3
```

这种形式可能使编译器使用构造函数 Rational :: Rational (int,int)把整数 100 变成类型 Rational 的一个临时变量，然后使用复制构造函数从新创建的临时对象对 r3 初始化：

```
{    // C++ pseudo code
     Rational  r3;
     Rational  _temp;
     _temp.Rational :: Rational (100,1);          // Construct the temporary
     r3.Rational Rational :: Rational (_temp);    // Copy - construct r3
     _temp.Rational :: ~ Rational ( );            // Destroy the temporary
     ...
}
```

这里的全部代价是两个构造函数和一个析构函数。在第一种形式中，对于

```
Rational  r1(100);                               // 1
```

我们只付出了一个构造函数的代价。

然而在实践中，大多数构造函数将优化临时对象。此处给出的三种初始化形式在效率上是等价的。

5.2 类型不匹配

上面的例子是一种常见的类型不匹配的特殊情况。我们试图用一个整数来初始化一个 Rational 类型的对象。类型不匹配的常见情况是需要类型 X 的时候却提供了一个其他类型。不过，编译器需要把提供的类型转换成所需的 X 类型的对象。在这个过程中可能会生成临时对象。请看如下代码：

```
{
     Rational  r;
     r = 100;
```

```
    ...
}
```

此处的 Rational 类没有声明接收整型参数的赋值操作符。然后，编译器希望右边是个 Rational 对象，此时左右有些矛盾。编译器必须想办法把我们提供的整型变量转换成 Rational 类型的对象。幸运的是(也许对于性能来说是不幸的)，我们有一个构造函数知道该怎么做：

```
class Rational
{
public:
    // If only one integer is provided, the second one will default to 1
    Rational ( int a = 0, int b = 1) : m (a), n (b) { }
    ...
};
```

该构造函数知道怎样由一个整型参数创建 Rational 对象。源代码语句

```
r = 100;
```

被转换成如下 C ++ 伪代码：

```
Rational _temp;     // Place holder for temporary

_temp. Rational :: Rational (100,1);          // Construct temporary
r. Rational :: operator = (_temp);            // Assign temporary to r
_temp. Rational :: ~ Rational ( );            // Destroy the temporary
```

这种由编译器实现的类型转换上的自由为编程提供了方便。但在您的源代码中，有些地方的性能考虑可能要压倒方便性。新的 C ++ 标准使您能够限制编译器并禁止这种转换。把构造函数声明为 explicit 就可以实现这一点：

```
class Rational
{
public:
    explicit Rational (int a = 0, int b = 1) : m (a), n (b) { }
    ...
};
```

explicit 关键字会告诉编译器您反对把该构造函数用作转换构造函数。

另一种方法是重载 Rational :: operator = ( )函数，使它接收一个整型参数，这样也

可以消除该临时对象：

```
class Rational
{
public:
    … //  as before
    Rational & operator = (int a ) { m = a; n = 1; return * this ; }
};
```

同样的原理可以推广到所有的函数调用。假设 g ( )是任意一个函数调用，它接收一个 string 引用参数：

```
void g ( const string& s)
{
    …
}
```

这里，必须重载 g ( )让它接收一个 char * 型参数，否则对 g ("message")的调用将引发临时 string 对象的创建：

```
void g ( const char * s)
{
    …
}
```

Cargil[Car92]指出了一种有趣的曲解，这是一种因类型不匹配而产生临时对象的情况。在下面的代码段中，operator+ ( )要求两个 Complex 对象作为参数。为此会创建出一个表示常量 1.0 的 Complex 对象：

```
Complex a,b;
…
for ( int ; i<100; i + + ) {
    a = i * b + 1.0;
    }
```

问题是在每次循环时都会反复创建这个临时对象。把常量表达式移到循环体外是一种很普通的和众所周知的优化方法。在 a = i * b +1.0;中产生临时对象是一种在每次循环中其值都不变的计算。在这种情况下，我们为什么还要重复地做它呢？下面采取一种一劳永逸的办法：

```
Complex  one (1.0);
```

```
for ( int i = 0; i<100; i + + ) {
    a = i * b + one;
    }
```

这样便把临时对象转换成了一个已命名的 Complex 对象。虽然它需要一次构造的代价，但是这仍然要胜过每次都重复创建临时对象。

5.3 按值传递

在按值传递一个对象时，使用实参来初始化形参等价于下面的形式[ES90]：

```
T formalArg = actualArg;
```

其中 T 是类的类型。假设 g ( )是一个在调用时需要 T 型参数的函数：

```
void g (T formalArg)
{
    ...
}
```

对 g ( )的典型调用可能是这样：

```
T t;
g (t);
```

g ( )的激活记录会在堆栈中有一个用于局部参数 formArg 的占位符。编译器必须把其内容复制给 g ( )的堆栈上的 formArg。对此，常用技术会产生一个临时对象[Lip96]。

编译器将创建一个临时对象并拷贝构造之，这是通过把 t 用作输入参数来实现的。然后临时对象将被当作实参传递给 g ( )。这个新创建的对象随后按引方式用传递给 g ( )。用 C++ 伪代码表示如下：

```
T _temp;
_temp.T :: T (t);          // copy construct _temp from t
g (_temp);                 // pass _temp by reference
_temp.T :: ~T ( );         // Destroy _temp
```

创建和清除临时对象的代价相对昂贵。如果可能的话，应该按指针或引用来传递对象以避免临时对象的创建。但是有些时候除了按值传递对象之外别无选择。有关这方面

的讨论，请参阅[Mey 97]中的第 23 款(Item 23)。

5.4 按值返回

另一个会导致临时对象创建的地方是函数返回值。如果编写了一个按值(与引用或指针相对)返回对象的函数，则很容易导致创建临时对象。请考虑一个简单的例子 f ( )：

```
string f ( )
{
    string s;
    ...                         // Compute "s"
    return s;
}
```

f ( )的返回值是一个 string 类型的对象。为了存放该返回值，需要创建一个临时对象。例如：

```
string p;
...
p = f ( );
```

存放 f ( )返回值的临时对象稍后赋值给左部对象 p。对于更具体的例子，可以考虑 string operator+。该操作符将实现 string“+”操作的直观解释。它接收两个输入的 string 对象，返回一个新的 string 对象，该新的 string 对象表示给定字符串的连接结果。该操作的一种可能的实现看起来可能是如下这样：

```
string operator + ( const string & s, const string& p)
{
    char * buffer = new char[s.length ( ) + p.length ( ) +1];

    strcpy (buffer , s.str);            // Copy first character string
    strcat (buffer , p.str);            // Add second character string
    string result (buffer);             // Create return object
        delete buffer;
    return result;
}
```

下面的代码段是对 string operator+的一种典型调用：

```
{
    string s1 = "Hello";
    string s2 = "World";
    string s3;
    s3 = s1 + s2;              // s3 <- "HelloWorld"
    ...
}
```

其中的语句：

```
s3 = s1 + s2;
```

会引发以下几个函数调用：

- operator + (const string &, const string&);= =>字符串合并操作。这是由 s1 +s2 引发的。
- string :: string (const char *);= =>构造函数。由在 operator +( )内执行 string result (buffer)引发。
- string :: string (const string &);= =>我们需要一个临时对象来存放 operator + ( )的返回值。复制构造函数将使用返回的 result string 创建该临时对象。
- string :: ~string ( );= =>在 operator +( )函数退出之前，它要清除 result string 对象，该对象的生存期限于局部范围内。
- string :: operator = (const string &);= =>为了把 operator+( )产生的临时对象赋值给左部对象 s3，要调用赋值操作符。
- string :: ~string ( );= =>要清除返回值所使用的临时对象。

对于一条源代码语句来说，6 个函数调用是很高的代价。即使它们中的大多数是内联的，也需要执行其逻辑。第 4 章讨论的返回值优化可以帮助我们清除 result string 对象。要注意构造函数和析构函数调用。我们是否也能够清除临时对象呢？那样可消除另外两个函数调用。

为什么语句：

```
s3 = s1 + s2;
```

在第一个位置上产生临时对象？这是因为我们没有权利去修改 string s3 的旧内容并用 s1+s2 产生的新内容来覆盖它。赋值操作负责把 string s3 由旧内容转换为新内容。编译器不允许跳过 string :: operator = ( )，因此临时对象是必需的。但是如果 s3 本来就是一个没有旧内容的新 string 对象，那么会出现什么情况呢？在这种情况下，编译器就不必担心旧内容，编译器可以使用 s3 而不是临时对象来存放。s1+s2 的结果直接复制构造到 string s3 对象中。s3 取代了临时对象，于是临时对象不再需要。简单说就是形式：

```
{
    string s1 = "Hello";
    string s2 = "World";
    string s3 = s1 + s2;          //  No temporary here.
    ...
}
```

要比以下形式可取：

```
{
    string s1 = "Hello";
    string s2 = "World";
    string s3;
    s3 = s1 + s2;                 // Temporary generated here.
    ...
}
```

5.5 使用 op=（）消除临时对象

在以上的讨论中，我们向编译器提供了一个已有的对象，让编译器对它工作，这样它就不会再创建临时对象了。同样的思想也可以应用于其他情况。假设 s3 没有旧值而且我们不必使用如下语句从零做起来初始化 s3：

```
string s3 = s1 + s2;
```

如果我们遇到这样一种情况：

```
{
    string s1,s2,s3;
    ...
    s3 = s1 + s2;
    ...
}
```

那么我们仍然可以阻止临时对象的创建。我们可以通过使用 string operator+=（）并使用+=代替+去重写代码来实现这一点，所以

```
s3 = s1 + s2;                     // Temporary generated here
```

要重写成这样：

```
s3 = s1;          // operator = ( ). No temporary.
s3 += s2;         // operator + = ( ). No temporary
```

如果 string :: operator+= ( )和 operator+ ( )以一致的风格实现(像它们应该做的那样,都实现"加"),那么这两段代码在语义上是等价的。惟一的区别是性能。尽管它们都调用了复制构造函数和操作符函数,但是前者会产生临时对象而后者不会,因此后者更为有效。

正如[Mey96]中指出的:

```
s5 = s1 + s2 + s3 + s4;          //  Three temporaries generated.
```

比

```
s5  = s1;
s5 += s2;
s5 += s3;
s5 += s4;
```

要优雅得多。但是在一个性能要求苛刻的执行路径中,常常需要为了有利于性能而放弃优雅。第二种"丑陋的"形式更为有效,因为它不会产生临时对象。

5.6 要 点

- 临时对象会以构造函数和析构函数的形式损失两倍的性能。
- 把构造函数声明为 explicit,可阻止编译器在背后使用类型转换。
- 编译器经常为了解决类型不匹配问题而创建临时对象。通过函数重载可以避免这种情况。
- 在可能的情况下,应尽量避免对象复制,应按引用来传递和返回对象。
- 在<op>可能是+、-、* 或/的情况下使用<op>=操作符可以消除临时对象。

第6章

单线程内存池

提高C++性能的编程技术

频繁的内存分配与释放是降低应用程序性能的重要原因。性能降低的主要原因在于默认的内存管理器是通用的。应用程序可能以一种特定的方式使用内存并为不需要的功能而遭受性能上的损失。可以通过开发专门的内存管理器来对付这种情况。专用内存管理器的设计空间是多维的。我们至少可以想起两个方面：大小和并发。大小方面分为不同的两点：

- 固定大小：分配单一固定大小内存块的内存管理器。
- 可变大小：分配任意大小内存块的内存管理器。预先不知道所需内存的大小。

类似地，并发方面也可以分为两点：

- 单线程：内存管理器受限于单个线程，内存由单个线程使用并且不会越过线程的边界。该类的内存管理器不涉及彼此都在执行的多个线程。
- 多线程：这种内存管理器用于并发执行的多个线程。这种实现中包含互斥执行的代码段。在任何时刻，这些代码段中的任何一个只能有一个线程在执行。

现在，我们已经有了四种不同风格的专用管理器：与大小有关的（固定大小，可变大小），与并发有关的（单线程，多线程）。本章我们要讨论专用管理器中单线程方面的情况及其性能关系。当然，我们的目标是开发比默认管理器快得多的另一种内存管理器。同时，我们不想开发太多的内存管理器。最终目标是把速度与尽可能多的灵活性和重用性结合起来。

6.1 版本0：全局函数new（）和delete（）

从设计上来说，默认的内存管理器是通用的。这就是调用全局函数new（）和delete（）所得到的。这两个函数的实现不能作任何简化假设。它们在进程环境中管理内存，而由于进程可以产生多个线程，所以new（）和delete（）必须能够在多线程环境中执行。此外，各请求所需的内存大小可能各不相同。这种灵活性以牺牲速度为代价。需要

的计算越多，消耗的时钟周期就越多。

通常情况下，客户端的代码不需要全局函数 new ( )和 delete( )的全部强大功能。客户端代码可能只(或大多数情况下)需要特定大小的内存块。客户端代码可能在单线程中执行，默认函数 new ( )和 delete( )所提供的并发保护并不被真正地需要。如果的确是这样，则使用这些函数的全部强大功能是对 CPU 时钟周期的一种浪费。通过定制内存分配模式以更好地满足您的特定需求，这样可以获得显著的效率。

假设您的代码要求为表示有理数的对象频繁地分配和释放内存：

```
class Rational {
public:
    Rational ( int a = 0, int b = 1) : n (a), d (b) { }
Private:
    int n;     // Numerator
    int d;     // Denominator
};
```

为了测量全局函数 new ( )和 delete( )的基准性能，我们执行如下测试：

```
int main ( )
{
    Rational * array[1000];
    ...
    // Start timing here
    for ( int j = 0; j < 500; j + + ) {
        for ( int i = 0;  i< 1000; i + + ) {
            array[i] = new Rational (i);
            }
        for (i = 0; i<1000; i + + ) {
            delete array[i];
            }
        }
    // Stop timing here
    ...
}
```

每执行一次最外层的循环都会执行 1 000 次 Rational 对象的分配和释放操作。500 次循环就得到 100 万次操作。

此代码段用去了 1 500ms。通过定制专用 Rational 内存管理器会获得什么样的

速度呢?回答这个问题需要我们编写自己的内存管理器并为 Rational 类重载 new( )和 delete ( )。

6.2 版本 1：专用 Rational 内存管理器

为了避免频繁地用到默认的管理器，Rational 类将维护一个用于预先分配的 Rational 对象的静态链表，该链表将作为一个可用对象的空闲列表。需要 Rational 对象时，可以从空闲列表中取出一个。对该对象操作完以后，再把它放回空闲列表供将来分配。

我们声明一个辅助结构来链接该空闲列表中的相邻元素。

```
class NextOnFreeList {
public:
    NextOnFreeList * next;
};
```

空闲列表被声明成一个由 NextOnFreeList 元素构成的链表。

```
class Rational {
...
static NextOnFreeList * freelist;
};
```

如果空闲列表是 NextOnFreeList 结构的列表，那么您可能想知道 Rational 对象存放在哪里。尽管该空闲列表的每一个元素都被声明成了 NextOnFreeList 结构，但是它同时也是 Rational 对象。为了在对象之间遍历，需要用每个 Rational 对象的头几个字节指向空闲列表的下一个对象。我们可以通过把对象转换成指向 NextOnFreeList 类型的指针来实现这一点。

这样，该空闲列表将具有双重角色，既是 Rational 对象的序列，又是 NextOnFreeList 元素的序列(如图 6.1 所示)。

空闲列表作为 Rational 类的静态成员声明。Rational 的 new ( )和 delete ( )管理该静态列表。这些操作符重载了相应的全局操作符。

```
class Rational {
public:
    Rational ( int a = 0, int b = 1 ) : n (a), d (b) { }

    inline void * operator new (size_t size);
    inline void operator delete (void * doomed, size_t size);
```

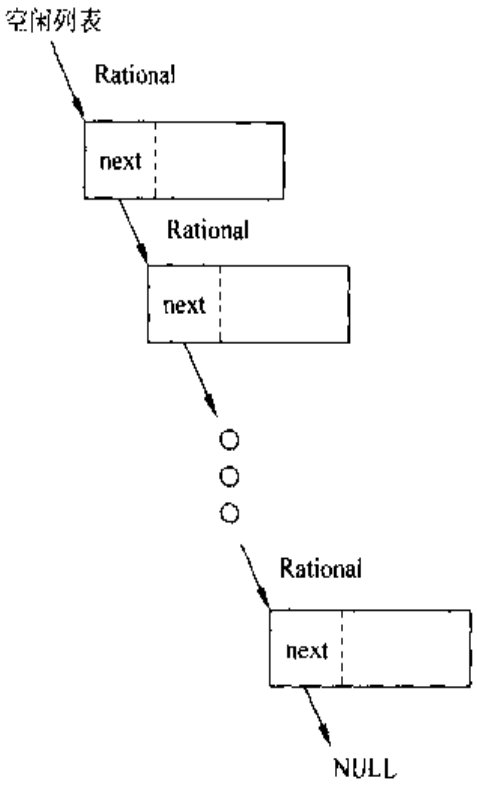


图 6.1 Rational 对象的空闲列表

```
    static void newMemPool ( ) { expandTheFreeList ( ); }
    static void deleteMemPool ( );
private:
    static NextOnFreeList * freeList;          // A free list of Rational objects.
    static void expandTheFreeList ( );
    enum { EXPANSION_SIZE = 32 };
    int n;                                     // Numerator
    int d;                                     // Denominator
};
```

操作符 new ( )从空闲列表中分配一个新的 Rational 对象。如果该空闲列表为空,那么它将得到扩展。我们先把空闲列表的头部去掉,在调整空闲列表的指针后再返回它。

```
inline
void * Rational :: operator new (size_t size)
{
    if ( 0 = = freeList ) { // If the list is empty, fill it up.
        expandTheFreeList ( );
        }
```

```
    NextOnFreeList * head = freeList;
    freeList = head->next;
    return head;
}
```

操作符 delete ( )向空闲列表返回 Rational 对象，这只需简单地在空闲列表前端添加 Rational 对象即可实现。

```
inline
void Rational :: operator delete (void * doomed, size_t size)
{
NextOnFreeList * head = static_cast <NextOnFreeList * > doomed;
head->next = freeList;
freeList = head;
}
```

该空闲列表用完以后，必须从堆里分配更多的 Rational 对象。这里我们必须指出一个细节：在 Rational 和 NextOnFreeList 类型之间进行转换有点危险。我们要确保空闲列表的每个元素都足够大，以便为每种类型服务。当我们用 Rational 对象填充该空闲列表时，必须记得比较 Rational 和 NextOnFreeList 的大小并分配两者之中较大的。

```
void Rational :: expandTheFreeList ( )
{
    // We must allocate an object large enough to contain the next pointer.
    size_t size = (sizeof (Rational) > sizeof ( NextOnFreeList * )) ?
        sizeof (Rational) : sizeof ( NextOnFreeList * );
    NextOnFreeList * runner =
    static_cast <NextOnFreeList * > new char [size];
    freeList = runner;
    for ( int = 0; i< EXPASION_SIZE; i + + ) {
        runner ->next = static_cast <NextOnFreeList * > new char [size];
        runner = runner->next;
        }
    runner ->next = 0;
}
```

本质上，expandTheFreeList ( )的这种实现不是最佳的。它对空闲列表的每个元素调用 operator new 一次。如果只调用 operator new 一次，得到一大块内存，然后我们自己把它分成多个元素，那将会更为有效[Mey 97]。孤立地看，这是一种正确的观察。然

而，我们实际上是在创建一个内存管理器，其思想是要尽量少地执行内存扩展和收缩，否则必须重新查看实现代码并进行修正。在我们的实现中空闲列表从不收缩，它将增大到一个固定的大小并停留在那种情况下。如果要实现一种更为有效的 expandTheFreeList ( ) 版本，将没有什么害处，但是似乎不会对整体性能有什么影响。

```
void Rational :: deleteMemPool ( )
{
    nextOnFreeList * nextPtr;
    for ( nextPtr = freeList; nextPtr! = NULL; nextPtr = freeList ) {
        freeList = freeList - >next;
        delete [ ] nextPtr;
        }
}
```

我们重复了下面的性能测试，但是这一次对 new 和 delete 的使用调用了 Rational 类重载的操作符：

```
NextOnFreeList * Rational :: freeList = 0;

int main ( )
{
    ...
    Rational * array[1000];
    Rational :: newMemPool ( );
    // Start timing here
    for ( int j = 0; j < 500; j + + ) {
        for ( int i = 0;  i< 1000; i + + ) {
            array[i] = new Rational (i);
            }
        for ( i = 0; i<1000; i + + ) {
            delete array[i];
            }
        }
    // Stop timing here
    Rational :: deleteMemPool ( );
    ...
}
```

现在，对 new ( )和 delete ( )的调用导致对我们所实现的 Rational :: operator new ( )

和 delete ( )的调用。该循环的执行时间从 1 500ms 降到了 43ms。这已经不只是一个数量级的进步了(图 6.2)。

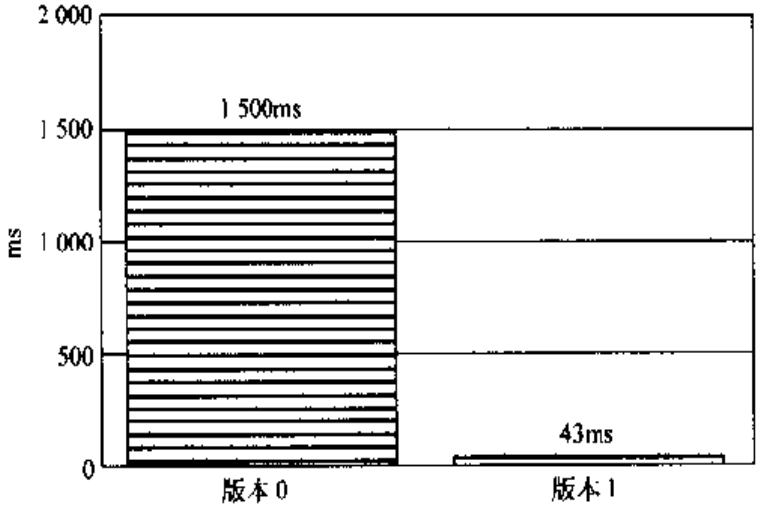


图 6.2 全局 new ( )和 delete ( )与 Rational 内存池的比较

速度上的提高来自何处？由于我们是在单线程环境中执行，所以我们的 Rational 内存管理子程序不受并发问题的干扰。由子我们不用担心任何临界区，所以不对内存管理进行保护。我们也充分利用了所有分配的内存空间大小相同的事实——即为 Rational 对象的大小。固定大小的内存分配要简单得多，要执行的计算少了很多(比如查找下一个能够满足需求的足够大的内存块)。只需简单地调整空闲列表的指针，而且我们正是这样做的。

6.3 版本 2：固定大小对象的内存池

版本 1 只限于管理 Rational 对象。如果我们想使内存管理器用于其他大小不同的类该怎么办呢？为每一个类重复管理逻辑显然是对开发时间的不必要浪费。如果我们看一下 Rational 内存管理器的实现，就会明显地看出内存管理逻辑实际上独立于特定的 Rational 类。惟一有关的是对象的大小——这是内存池模板实现的良好候选。内存池将管理某种类型的一个可用对象池。模板实现将允许我们把管理的特定对象多样化。

```
template < class T >
class MemoryPool {
public:
    MemoryPool (size_t size = EXPANSION_SIZE);
    ~MemoryPool ( );
```

```
    // Allocate a T element from the free list.
    inline void * alloc (size_t size );
    // Return a T element to the free list.
    inline void free (void * someElement);
private:
    // next element on the free list.
    MemoryPool<T> * next;
    // If the freeList is empty, expand it by this amount.
    enum { EXPANSION_SIZE = 32};
    // Add free elements to the free list
    void expandTheFreeList ( int howMany = EXPANSION_SIZE);
};
```

MemoryPool 的构造函数对空闲列表进行初始化，size 参数指定空闲列表的初始长度。

```
template < class T >
MemoryPool <T> :: MemoryPool (size_t size)
{
    expandTheFreeList (size);
}
```

析构函数遍历空闲列表并删除全部元素。

```
template < class T >
MemoryPool <T> :: ~MemoryPool ( )
{
    MemoryPool<T> * nextPtr = next;
    for (nextPtr = next; nextPtr ! = NULL; nextPtr = next ) {
        next = next - >next;
        delete [ ] nextPtr;
        }
}
```

alloc ( )成员函数为 T 元素分配足够的空间。如果空闲列表耗尽，则可以调用 expandTheFreeList ( )来补充它。

```
template < class T >
inline
void * MemoryPool <T> :: alloc ( size_t )
{
```

```
    if (! next) {
        expandTheFreeList ( );
        }

    MemoryPool<T> * head = next;
    next = head - >next;
        return head;
}
```

free ( )成员函数通过把 T 元素放回空闲列表来释放 T 元素。

```
template < class T >
inline
void MemoryPool <T> :: free ( void * doomed)
{
    MemoryPool<T> * head = static_cast <MemoryPool<T> * > doomed;
    head - >next = next;
    next = head;
}
```

expandTheFreeList ( )用于向空闲列表添加新元素。新元素从堆中分配并一起链接到链表中。当空闲列表耗尽时会调用这个函数。

```
template < class T >
void MemoryPool <T> :: expandTheFreeList ( int howMany )
{
    // We must allocate an object large enough to contain the
    // next pointer.
    size_t size = (sizeof (T) > sizeof (MemoryPool<T> * )) ?
        sizeof (T) : sizeof (MemoryPool<T> * );

    MemoryPool<T> * runner = static_cast <MemoryPool<T> * > new char [size];
    next = runner;
    for ( int i = 0; i < howMany ;  i + + ) {
    runner - >next = static_cast <MemoryPool<T> * > new char [size];
    runner = runner - >next;
    }
  runner - >next = 0;
}
```

Rational对象不再需要维护它自己的空闲列表，这个职责委派给了MemoryPool类。

```
class Rational {
public:
    Rational ( int a = 0, int b = 1 ) : n (a), d (b) { }
    void * operator new (size_t size ) { return memPool - >alloc (size ); }
    void operator delete (void * doomed,size_t size)
                { memPool - >free (doomed) ; }
    static void newMemPool ( ) { memPool = new MemoryPool <Rational>; }
    static void deleteMemPool ( ) { delete memPool; }
private:
    int n;                  // Numerator
    int d;                  // Denominator
    static MemoryPool <Rational> * memPool;
};
```

我们去除了freeList静态成员指针及其有关函数，并用指向内存池的指针来代替它。在前面的Rational实现中，模板内存池在其定义中定制成Rational内存池。

我们的测试循环基本上与上一个相同：

```
MemoryPool <Rational> * Rational :: memPool = 0;
int main ( )
{
    ...
    Rational * array[1000];
    Rational :: newMemPool ( );
    // Start timing here
    for ( int j = 0; j < 500; j + + ) {
        for ( int i = 0;  i< 1000; i + + ) {
            array[i] = new Rational (i);
            }
        for ( i = 0; i<1000; i + + ) {
            delete array[i];
            }
        }
    // Stop timing here
    Rational :: deleteMemPool ( );
    ...
}
```

前面的 Rational 空闲列表内存管理器(版本 1)执行循环共消耗了 43ms，我们希望这次测量与上次相同。由于某些原因，内存池的模板版本(版本 2)耗时 63ms(如图 6.3 所示)，这比前一个版本要慢一些。同时，版本 2 产生的汇编代码有一些附加的指令。但是，它仍然比全局函数 new ( )和 delete ( )胜出不止一个数量级。

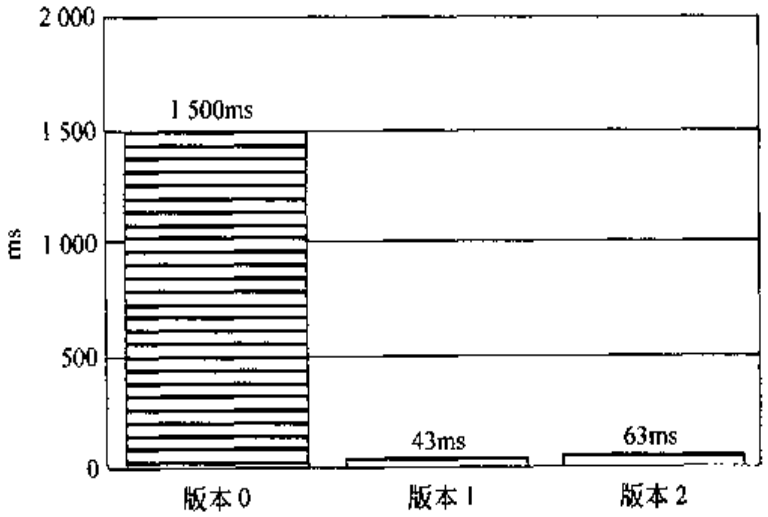


图 6.3 为普通对象添加模板内存池

到现在为止，我们一直着眼于内存空间分配设计的一个方面。那就是在单线程环境中固定大小分配方面的特定问题。现在通过放宽大小方面的限制来扩大选择范围。我们仍停留在单线程的领域，但是要取消对所有内存空间的分配都按相同大小进行的假设。

6.4 版本3：单线程可变大小内存管理器

加速固定大小内存的分配和释放只能到此为止。有一类应用程序要求可变内存分配。Web 服务器就是个很好的例子。Web 服务器的典型实现就是一个巨大的字符串计数器。只要扫一眼代码就会发现处理单个 HTTP 请求要求大量的字符串处理。许多这样的调用需要分配空间来创建新字符串或复制字符串。另外，事先无法知道需要分配多少字符串及其空间大小。潜在的情况是字符串可能非常大。特定 HTTP 请求的内容确定了字符串管理的本质。固定大小的内存管理器无法满足这种需求。然而依赖于全局函数 new ( )和 delete ( )却是不可能的，因为全局函数 new ( )和 delete ( )的代价太昂贵了。它们消耗几百条指令，更有甚者，它们包含互斥临界区，这阻止了并发线程的执行，因此损害了可伸缩性。这将会破坏 Web 服务器的速度和可伸缩性。这正是我们自己编写可变大小内存管理器的最佳之处。

实现可变内存管理器的方式有很多[ALG95]。下面讨论我们在自己的 Web 服务器中所用的方式。

MemoryChunk 类代替了前面版本中的 NextOnFreeList 类。它用于把各种大小的内存块串成一个块序列。

```
class MemoryChunk {
public:
    MemoryChunk (MemoryChunk * nextChunk, size_t chunkSize);
    ~MemoryChunk ( ) {delete mem;}
    inline void *alloc (size_t size);
    inline void free (void* someElement);
    // Pointer to next memory chunk on the list.
    MemoryChunk *nextMemChunk ( )  {return next;}
    // How much space do we have left on this memory chunk?
    size_t spaceAvailable ( )
        { return chunkSize - bytesAlreadyAllocated; }
    // this is the default size of a single memory chunk.
    enum { DEFAULT_CHUNK_SIZE = 4096 };
private:
    MemoryChunk *next;
    void                *mem;
    // The size of a single memory chunk.
    Size_t chunkSize;
    // This many bytes already allocated on the current memory chunk.
    Size_t bytesAlreadyAllocated;
};
```

MemoryChunk 类是 NextOnFreeList 的一个更为干净的版本。它把 next 指针从用于已分配对象的实际内存中分离了出来。它使用显式的 next 和 mem 指针，它们不需要转换。图 6.4 给出了图形表示。

MemoryChunk 的构造函数首先确定内存块的大小。它使用这个值从堆中分配它的私有存储空间。构造函数同时使 next 成员指针指向输入参数 nextChunk。nextChunk 是链表以前的头部，我们使新创建的 MemoryChunk 成为 MemoryChunk 对象链表的新头部。由于这是一个崭新的 MemoryChunk，所以该块的已分配字节数被设成 0：

```
MemoryChunk :: MemoryChunk (MemoryChunk * nextChunk, size_t reqSize)
{
    chunkSize = (reqSize > DEFAULT_CHUNK_SIZE) ?
```

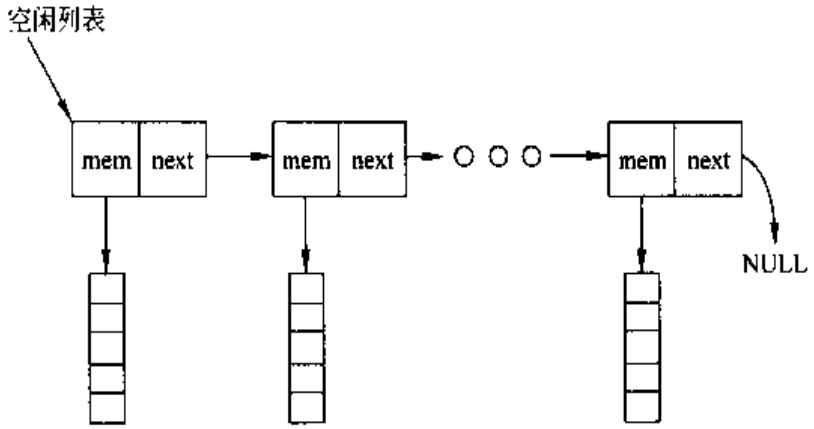


图 6.4 可变大小内存空闲列表

```
            reqSize : DEFAULT_CHUNK_SIZE;
    next = nextChunk;
    bytesAlreadyAllocated = 0;
    mem = new char [chunkSize];
}
```

析构函数释放构造函数所获得的内存：

```
MemoryChunk :: ~MemoryChunk ( ) { delete [ ] mem; }
```

内存分配请求由 alloc( )方法处理。它返回一个指针，指向由 mem 所指向的 MemoryChunk 私有存储空间中的可用空间。它通过更新本块中已分配字节数来跟踪可用空间的大小。

```
void * MemoryChunk :: alloc (size_t requestSize)
{
    void * addr = static_cast <void * >
        (static_cast <size_t> mem + bytesAlreadyAllocated);
    bytesAlreadyAllocated + = requestSize;
    return addr;
}
```

在此实现中，我们不用操心空闲内存段的释放。当对象删除以后，整个内存块将会被释放并送回到堆中：

```
inline void MemoryChunk :: free (void * doomed) { }
```

MemoryChunk 只是一个辅助类，它由 ByteMemoryPool 类用来实现可变内存管理器：

```
class ByteMemoryPool {
public:
    ByteMemoryPool (size_t initSize =
                    MemoryChunk :: DEFAULT_CHUNK_SIZE);
    ~ByteMemoryPool ( );
    // Allocate memory from private pool.
    inline void *alloc (size_t size);
    // Free memory previously allocated from the pool
    inline void free (void* someElement);
private:
    // A list of memory chunks. This is our private storage.
    MemoryChunk *listOfMemoryChunks;
    // Add one memory chunk to our private storage
    void expandStorage (size_t reqSize);
};
```

尽管内存块列表可能包含不止一个块，但是只有第一块中拥有可用于分配的内存。其他块表示已经分配了的内存。列表的第一个元素是惟一能够分配可用内存的块。

构造函数接收一个用于指定单个内存块大小的 initSize 参数。构造函数由此来设置单个内存块的大小。expandStorage( )方法使 listOfMemoryChunks 指向一个已分配的 MemoryChunk 对象：

```
// Construct the ByteMemoryPool object. Build the private storage.
ByteMemoryPool :: ByteMemoryPool(size_t initSize)
{
    expandStorage ( initSize);
}
```

析构函数遍历内存块列表并释放它们：

```
ByteMemoryPool :: ~ByteMemoryPool ( )
{
    MemoryChunk *memChunk = listOfMemoryChunks;
    while (memChunk) {
        listOfMemoryChunks = memChunk->nextMemChunk ( );
        delete memChunk;
        memChunk = listOfMemoryChunks;
        }
}
```

ByteMemoryPool :: alloc ( )确保有足够的可用空间，然后把分配的任务委派给列表顶部的 MemoryChunk：

```
void* ByteMemoryPool :: alloc ( size_t requestSize )
{
    size_t space = listOfMemoryChunks->spaceAvailable ( );
    if (space < requestSize ) {
        expandStorage ( requestSize);
        }
    return listOfMemoryChunks->alloc ( requestSize);
}
```

与前面等价的 MemoryPool<T> :: alloc( )和 Rational :: operator new ( )的实现相比，ByteMemoryPool :: alloc ( )的实现有点过于复杂。以 MemoryPool<T> :: alloc( )为例，我们检查列表的状态，如果它是空的，那么就对它进行扩充。在计算方面，所有随后要做的事就是返回该元素到列表头部。而对于 ByteMemoryPool :: alloc ( )来说，我们要计算的东西要多很多。由于正在处理未知请求空间大小的情况，所以在 MemoryChunk 中必须有足够的可用空间。否则，就必须调用 expandStorage ( )分配一个新的 MemoryChunk 并把它放到块列表的头部。总之，我们现在确保有足够的空间来满足请求。下一步，调用 MemoryChunk :: alloc ( )来计算要返回给调用者的内存地址并调整它的相应记录以反映该块的已分配字节数量。所有这些附加的计算都将在某种程度上降低性能。这就是扩展功能的代价。

释放先前分配内存的工作委派给了列表头部的 MemoryChunk：

```
inline
void ByteMemoryPool :: free (void * doomed)
{
    listOfMemoryChunk->free (doomed);
}
```

记住 ByteMemoryPool :: free ( )方法非常简单——它不做任何事情。为什么我们这么不急于释放内存呢？这是因为 ByteMemoryPool 实现不需要重用以前分配的内存。如果需要更多的内存，那么我们将创建一个新的内存块并把它用于将来的分配。在池的析构中，内存被释放回堆中。ByteMemoryPool 析构函数把所有内存块释放回堆中。

这不像听起来那样浪费。我们使用这种方案处理 HTTP 请求。在每个请求的开始要创建一个 ByteMemoryPool 对象。内存块的大小是 4 096 字节，这对于 99%的 HTTP

请求都是十分充足的。因此在典型情况下，我们不必扩展私有存储空间，因为单个块的大小够用了。由于 HTTP 请求的处理都是短暂的任务，所以没有必要担心释放和重用空闲的私有空间。在一个池被释放期间（请求结束时）立即卸载整个块会更有效。这种方法还适用于以不同形式重复出现在本书中的一个有关信息的例子：特定的环境允许简化假设。简化假设又提供了大幅度优化的机会。您不希望使用适用于更大问题的解决方案。这里出现的另一个问题是性能有时会危及重用。为了促进性能，您有时可能需要为特定的环境定制有针对性的解决方案。那些方案不追求超出目标的特强灵活性和重用性，它们只适用于狭窄的领域。

如果出现了存储块用尽这种不希望出现的情况，那么我们就通过创建新的内存块并把它添加到内存块列表的头部来进行扩展：

```
void ByteMemoryPool :: expandStorage (size_t reqSize)
{
    listOfMemoryChunks = new MemoryChunk (listOfMemoryChunks, reqSize);
}
```

为了测试内存管理器，我们修改了 Rational 类的实现，把 MemoryPool<Rational> 对象替换成 ByteMemoryPool：

```
class Rational {
public:
    Rational ( int a = 0, int b = 1 ) : n (a), d (b) { }

    void * operator new (size_t size ) { return memPool ->alloc (size ); }
    void operator delete (void * doomed,size_t size)
                { memPool ->free (doomed) ; }
    static void newMemPool ( ) { memPool = new ByteMemoryPool; }
    static void deleteMemPool ( ) { delete memPool; }
private:
    int n;              // Numerator
    int d;              // Denominator
    static ByteMemoryPool * memPool;
};
```

通过这个内存池实现，我们再一次测量了分配和释放 Rational 对象的测试循环：

```
MemoryPool <Rational> * Rational :: memPool = 0;
int main ( )
{
```

```
    ...
    Rational * array[1000];
    Rational :: newMemPool ( );
    // Start timing here
    for ( int j = 0; j < 500; j + + ) {
        for ( int i = 0;  i< 1000; i + + ) {
            array[i] = new Rational (i);
            }
        for ( i = 0; i<1000; i + + ) {
            delete array[i];
            }
        }
    // Stop timing here
    Rational :: deleteMemPool ( );
    ...
}
```

执行该循环用了 140ms。图 6.5 中比较了以下几种情况下的执行速度：

- 版本 1，Rational 内存管理器。
- 版本 2，对象内存管理器的模板实现。
- 版本 3，可变大小内存管理器。

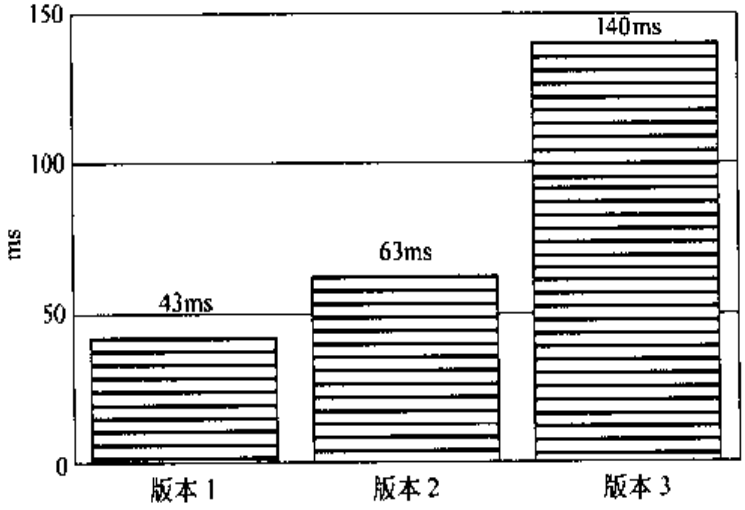


图 6.5 可变大小内存池通常慢于固定大小的内存池

不出所料，版本 3 要比版本 1 和版本 2 慢，这主要因为增加了分配逻辑的复杂性。随着内存管理方面的功能越来越强大，我们必须牺牲一些速度。

6.5 要 点

- 灵活性的代价是牺牲速度。随着内存管理的功能和灵活性的增加，执行速度会下降。
- 全局内存管理器(由 new ( )和 delete ( )实现)是通用的，因而开销也是昂贵的。
- 专用内存管理器比全局内存管理器要快一个数量级以上。
- 如果主要需要分配固定大小的内存块，那么专用的固定大小内存管理器将带来显著的性能提高。
- 如果主要需要分配限于单线程的内存块，那么也会有类似的性能提高。单线程内存管理器将通过避免全局函数 new ( )和 deletc ( )所必须处理的并发问题从而帮助提高性能。

第7章

多线程内存池

提高C++性能的编程技术

上一章我们的视线完全停留在单线程环境范围内。内存池为单个线程所拥有，并发问题被忽略。现在，把设计延伸到多线程环境。内存池将不再属于某个特定线程，它将由应用程序的所有线程共享。

为单线程环境开发的分配器将无法在多线程环境中正确地工作。为了允许多个线程并发地分配和释放内存，必须在分配器方法中添加互斥锁。可以复制单线程的实现并在适当的位置添加锁定，但这种生硬的机制要求为每种内存池和每种锁定方案提供不同的实现。这是一种适合于使用模板实现的情况。这样一种多线程内存池的实现可以由内存池类型来确定其参数。它将使用锁保护简单地改写 alloc ( )和 free ( )(上一章所开发的)。通过把锁类型也变成一个参数，这能够以不同的锁方案来实例化内存池，我们由此更进了一步。这是另一种程度的自由。

7.1 版本4：实现

版本4实现多线程内存池。它可以处理符合接口要求的任何类型的池和锁：

```
template <class POOLTYPE , class LOCK>
class MTMemoryPool {
public:
    // Allocate an element from the freeList;
    inline void* alloc (size_t size);
    // Return an element to the freeList.
    inline void free (void* someElement);
private:
    POOLTYPE stPool;   // Single - threaded pool.
    LOCK         theLock;
```

```
};
```

alloc ( )方法把分配任务委派给内存池成员，相应地把锁定任务委派给锁成员：

```
template <class M, class L>
inline
void* MTMemoryPool<M,L>::alloc (size_t size)
{
    void * mem;
    theLock.lock ( );
    mem = stPool.alloc (size);
    theLock.unlock ( );
    return mem;
}
template <class M, class L>
inline
void* MTMemoryPool<M,L>:: free (void* doomed)
{
    theLock.lock ( );
    stPool.free (doomed);
    theLock.unlock ( );
}
```

要实例化 MemPool 模板，就需要提供内存池类型和锁类型。对于内存池，我们将重用上一章所开发的内容。对于锁，我们从如下代码开始：

```
class ABClock { // Abstract base class
public:
    virtual ~ABClock ( ) { }
    virtual void lock ( ) = 0;
    virtual void unlock ( ) = 0;
};
class MutexLock:public ABClock {
public:
    MutexLock ( ) {pthread_mutex_init (&lock, NULL);}
    ~MutexLock ( ) { pthread_mutex_destroy ( &lock);}
    inline void lock ( ) {pthread_mutex_lock (&lock);}
    inline void unlock ( ) {pthread_mutex_unlock ( &lock);}
private:
    pthread_mutex_t lock;
```

```
};
```

为了实例化多线程内存池，还需要修改 Rational 的实现。核心内存池是 MemoryPool<Rational>类型的，MutexLock 提供锁定服务：

```
class Rational {
public:
    ...
    static void newMemPool ( ) {
        memPool = new MTMemoryPool <MemoryPool <Rational>, MutexLock>;
    }
private:
    ...
    static MTMemoryPool <MemoryPool<Rational>, MutexLock> * memPool;
};
```

我们的测试程序保持不变。它仍然是先前用于测试 Rational 对象分配和释放的同一个循环：

```
for ( int j = 0; j < 500; j++ ) {
    for ( int i = 0; i< 1000; i++ ) {
        array[i] = new Rational (i);
        }
    for ( i= 0; i<1000; i++ ) {
        delete array[i];
        }
    }
```

我们最后一次测试 MemoryPool 的执行时间时，版本 2 用时 63ms。因此当我们看到版本 4 的执行时间增加到 1 300ms 时有点吃惊。图 7.1 对以下 3 种情况进行了比较：

- 版本 0，全局 new ( )和 delete ( )。
- 版本 2，MemoryPool：固定对象内存管理器的单线程模板实现。
- 版本 4，MTMemoryPool：多线程 MemoryPool。

在版本 4 的实现中，某些地方出了问题。经过快速分析后发现，执行时间完全由 pthreads 库的锁定调用决定。分配内存包含 pthread_mutex_lock( )调用，后面紧跟着 pthread_mutex_unlock ( )调用。同样的事情出现在释放内存的过程中。总之，每次主循环的执行都要引起两次 pthread_mutex_lock( )调用和另外两次 pthread_mutex_unlock ( )调用：其中一对用于分配对象，另一对用于释放对象。由于在当前环境中锁定是必要的，所以我们惟一的希望是对锁定调用进行大幅改进。

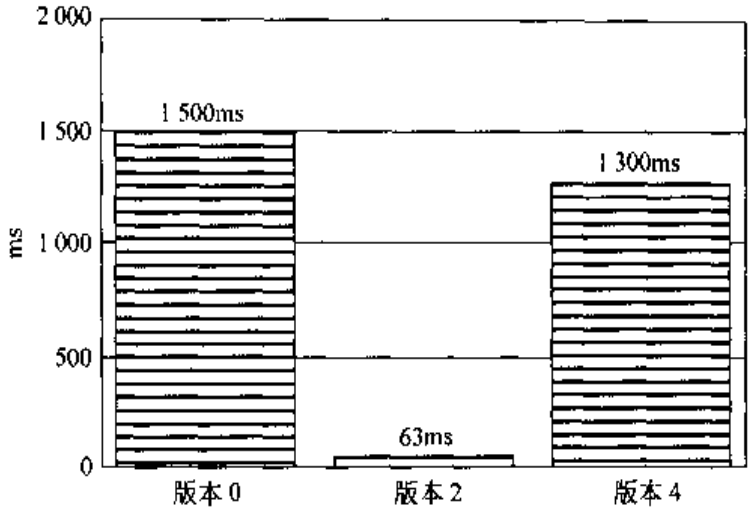


图 7.1 多线程内存池与单线程内存池的对比

7.2 版本5：快速锁定

在使用 pthreads 锁定调用时出了什么问题？以前曾提到过，有时并不真正需要默认内存管理所提供的全部强大功能和灵活性。现在我们在锁定问题上遇到了类似问题。有时并不真正需要 pthreads 库锁定服务所提供的全部强大功能。例如，pthread_mutex_lock( )必须检查确认调用线程还没有拥有锁，否则会造成死锁。而 pthread_mutex_unlock ( )则必须检查确认调用线程是真正拥有锁的线程。所有这些细小的检查和计算都消耗了宝贵的 CPU 时钟周期。

假设我们在版本 4 中并不真正需要所有这些锁定灵活性。还假设应用程序对锁定服务的使用是简单的，我们可以保证正在锁定的线程还没有拥有锁。再假设我们可以保证正在解锁的线程是首先锁定的线程。现在有了一种锁方案，它远不及 pthreads 库所提供的锁方案复杂，但使用这种方案，可以从费时的锁方案中解脱出来。我们应该做的是实现新的锁类，它使用更快和更原始的“积木块”来提供锁定。沿着为速度牺牲可移植性的方向，我们的下一个锁类实现是平台相关的。然而，每一种平台都提供一种比 pthreads 库要快很多的原始基本积木块。在此，我们为 AIX 定制 PrimitiveLock 实现。

版本 5 简单地把 MutexLock 替换成一个名为 PrimitiveLock 的更快的实现：

```
class PrimitiveLock:public ABClock {
public:
    PrimitiveLock ( ) {
```

```
            _clear_lock ( (static_cast<atomic_p>) &_lock, LOCK_FREE);}
    ~PrimitiveLock ( ) { }
    inline void lock ( ) { // Spin lock
        while (! _check_lock ( (static_cast<atomic_p>) &_lock,
                    LOCK_FREE,LOCK_BUSY));
        }
    inline void unlock ( ) {
        _clear_lock ( (static_cast<atomic_p>) &_lock, LOCK_FREE);
        }
private:
    int _lock;
    enum {LOCK_FREE = 0, LOCK_BUSY = 1};
};
```

MTMemoryPool 的实现保持不变。这就是模板的强大功能。为了使用 PrimitiveLock 实例化 MTMemoryPool，只需要对 Rational 类稍加修改：

```
class Rational {
public:
    ...
    static void newMemPool ( ) {
        memPool = new MTMemoryPool <MemoryPool <Rational>, PrimitiveLock>;
    }
private:
    ...
    static MTMemoryPool <MemoryPool<Rational>, PrimitiveLock> *memPool;
};
```

通过使用这个更快的锁定类，执行时间缩短到了 900ms(如图 7.2 所示)。

到现在为止，我们一直把 MTMemoryPool 模板作为多线程解决方案。实际上它要灵活得多。可以在单线程环境中对其进行实例化，它将和单线程方案执行得一样快。通过使用一个什么都不做的 DummyLock 类实例化 MTMemoryPool 可以实现这一点：

```
class DummyLock:public ABClock {
public:
    inline void lock ( ) { }
    inline void unlock ( ) { }
};
```

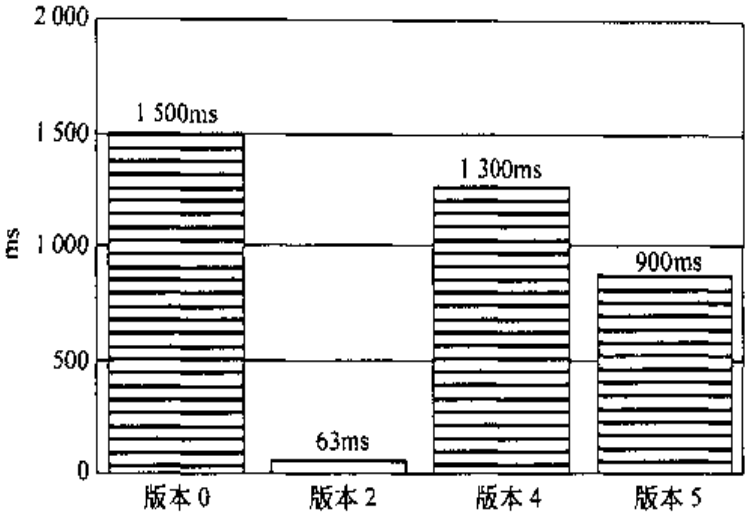


图 7.2 使用快速锁定的多线程内存池

当按以下语句的方式使用 DummyLock 类来实例化 MTMemoryPool 时：

```
MTMemoryPool <MemoryPool<Rational>,DummyLock> myRationalPool;
```

本质上等同于：

```
MemoryPool<Rational> myRationalPool;
```

如果编译器执行了内联，那么这两种形式的性能是相同的。内联将使 DummyLock::lock ( )和 DummyLock::unlock ( )缩短成零条指令。

上一章我们开发了一个名为 ByteMemoryPool 的单线程、可变大小分配器。我们可以按照与扩展 MemoryPool 相同的方式把 ByteMemoryPool 扩展到多线程环境中。我们只是简单地使用 ByteMemoryPool 参数来实例化 MTMemoryPool 模板。要测试它只需要修改 Rational 实现中的两行代码：

```
class Rational {
public:
    ...
    static void newMemPool ( ) {
        memPool = new MTMemoryPool <ByteMemoryPool , MutexLock>;
    }
private:
    ...
    static MTMemoryPool <ByteMemoryPool, MutexLock> * memPool;
};
```

图 7.3 对在第 5 章和第 6 章中开发的各种内存池类的性能进行了对比：

- 版本 0，全局 new ( )和 delete ( )。
- 版本 2，MemoryPool：固定对象内存管理器的单线程模板实现。
- 版本 4，MTMemoryPool：多线程 MemoryPool。
- 版本 5，本质上是版本 4，它使用了更快的、平台相关的锁。
- 版本 6，使用 ByteMemoryPool 和快速锁定的多线程可变大小内存管理器。

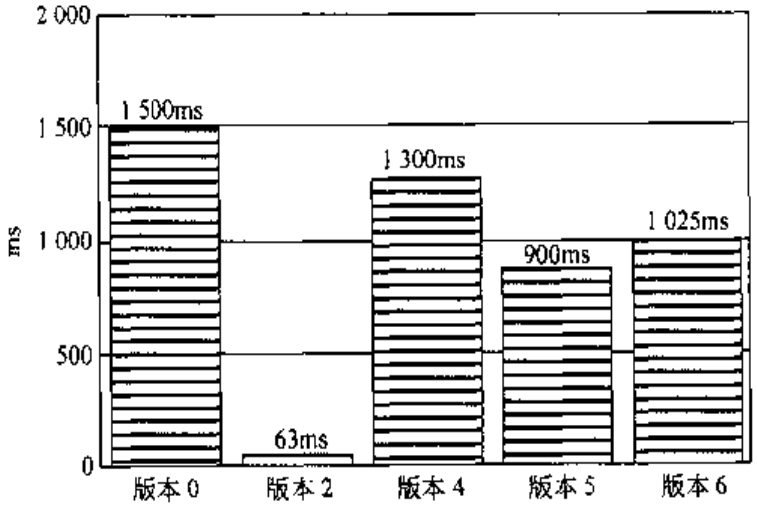


图 7.3 不同风格内存池的对比

在第 5 章和第 6 章中，给出了各种各样的内存分配器，但是它们还很不完善。我们没有探讨内存分配器的全部内容，只是谈到其中的一个小子集。考虑如下代码所定义的可变大小分配器：

```
MemPool < ByteMemoryPool, PrimitiveLock> myPool;
```

该分配器在 SMP 环境中有严重的缺点，因为它不可伸缩。我们的 ByteMemoryPool 实现通过管理单个 MemoryChunk 来分配内存。该 MemoryChunk 是不能并发访问的单一资源。如果两个线程正在试图分配内存，那么它们必须被串行化。为了打破这种可伸缩性瓶颈，可以把临界资源分成多个资源。例如，我们可以用更小的单位(少于 512 字节)分配其中一个资源，而使用大一些的单位(512 字节或更多)分配另一个资源。现在，分别需要 100 和 1 000 字节的两个请求可以互不影响地并发处理。在此不深入探讨这种特殊的分配器。我们将在第 15 章中全面地讨论可伸缩性问题。

7.3 要 点

- 全局内存管理器[由 new ( )和 delete ( )实现]是通用的,因而开销也是昂贵的。
- 如果基本上限于在单线程环境中分配内存块,则可以显著地改进性能。单线程内存管理器比多线程内存管理器要快得多。
- 如果开发了一套有效的单线程分配器,那么通过使用模板可以轻松地把它们扩展到多线程环境中。

第 8 章

内联基础

提高 C++ 性能的编程技术

本章离开了设计领域并开始着眼于与设计很不相关的速度机制。这些技术将提高任一程序的性能，而与其整体设计质量无关。不过，这些机械性的技巧无法使设计糟糕的程序变快——只能使其变得比原来更快一些。尽管它们能够改变某一问题的数据集大小（其他较为低效的算法也可以对此进行有效的处理），但是它们无法使选择排序法比快速排序法更快。这些技术有一些是无偿的。这就是说，它们是纯粹的性能收获，对其他软件质量特征没有任何负面影响。然而，这些技术中的大多数需要设计者为了获得更高性能而牺牲一些设计特性，如代码大小、可移植性、可扩展性或通用性等。几乎所有这些技术都对可维护性有负面影响。

8.1 什么是内联

内联用一种类似于宏的展开的方式代替方法调用，它在调用方法内部将被调用方法展开。指明内联意图的机制有两种：一种是用保留字 inline 给方法的定义加上前缀；另一种方法是在头部声明中定义方法。我们使用一个代码示例更容易说明：

```
class GatedInt {
    int x;
public:
    int get ( ) {
        return x;
    }
void set ( int arg) {
    x = arg;
    }
};
```

由于 GatedInt::get 和 GatedInt::set 在声明内定义，所以它们都将被内联。另一种情况是，要内联的方法可以定义在类声明的外面，但是必须放在头文件内或头文件包含的文件内。

```
class GatedInt {
    int x;
public:
    int get ( ) { };
void set (int arg) { };
};
    inline int GatedInt::get ( )
{
    return x;
}
    inline int GatedInt::set ( int arg )
{
    x = arg;
}
```

内联方法在被调用时和其他方法一样，但是在编译时和其他方法不同。用于内联方法的代码需要被展开，这意味着任何访问内联方法的代码必须访问该方法的定义。内联方法的定义必须合并到调用它的方法中去，这意味着内联方法内的任何改变都要求重新编译所有使用该方法的模块。内联潜在地提供显著的性能提高的代价之一是增加了编译时间。某些情况下这种增加可能是不大的，但在另一些情况下它又可能是极大的。在最极端的情况下，对内联方法的修改可能要求对整个程序进行彻底的重新编译。这一般使得对任何东西的内联到代码开发的较后阶段成为一种不错的思想。

应用程序可以这样调用这些方法：

```
int main ( )
{
    GatedInt gi;
    gi.set (12);
    cout <<gi.get ( );
}
```

方法以连续代码块的形式出现在程序中。这些连续的代码块包含方法所执行的操作。就我们的示例程序来说，get、set 和 main 将各自以独立的代码块出现，这些代码块包含机器指令，这些机器指令是编译器为执行这些代码块各自的操作而选择的。在我们的

示例程序中，main 实例化 GatedInt 的一个实例，把直接量 12 压入堆栈并调用 GatedInt::set。随后 main 调用 GatedInt::get 并把返回值压入堆栈，以便 iostream::operator<<能够输出该值。GatedInt::get 返回私有成员变量 GatedInt::x 的值，GatedInt::set 则把输入参数 arg 的值赋给 GatedInt::x。如果忽略操作系统为加载和运行这个程序所做的工作，则该程序的外联(不是内联)版本将进行 3 次调用。在头两次调用中，完成调用和返回所需要工作的开销比在被调用方法内实际执行代码可能会多一个数量级。

相反，内联 get 和 set 将使 main 程序仅包含单个对 iostream::operator<<的调用。在 get 和 set 被内联后和 main 被编译之前，主程序在逻辑上基本上是这样：

```
int main ( )
{
    GatedInt gi;
    {
        gi::x = 12;
    }
    int temp = gi::x;
    cout << temp;
}
```

注意我们是说逻辑上，实际上编译器会做得比这更好。main 经过优化后将简化成这样：

```
int main ( )
{
    cout << 12;
}
```

逻辑上，编译器为内联一个方法所使用的过程是这样的：内联方法的连续代码块被复制到调用方法的调用点处。内联方法内的任何局部变量在块内分配。内联方法的输入参数和返回值被映射到调用方法的局部变量空间。如果内联方法有多个返回点，则这些返回点就变成内联块尾部的分支(可怕的 GOTO)。所有与调用(对于与创建新块相关联的 SP 修改可能例外)有关的痕迹及随之而来的所有性能损失都被消除了。然而，避免调用只是与内联有关的性能项的一半。假设我们有两个方法：y 和 build_mask：

```
int x::y (int a)
{
    ...
    int b = 6;
```

```
    …     // b is not modified within this section
    int m = build_mask (b);
    …     // m is not modified in this section
    int n = m + 1;
    …
}

inline
int build_mask (int q)
{
    if (q > WORD_SIZE) return -1;
    else if (q > 0) return ( 1 << q ) -1;
    else return 0;
}
```

把 build_mask 内联到 y 中不经优化的结果将会是：

```
int x::y (int a)
{
    …
    int b =6;
    …     // b is not modified within this section
    int m;
    {
        int _temp_q = 6;
        int _temp;
        if (_temp_q > WORD_SIZE) _temp = -1;
        else if (_temp_q > 0) _temp = (1 << q) - 1;
        else _temp = 0;
        m = _temp;
    }
    …     // m is not modified in this section
    int n = m + 1;
    …
}
```

然而，优化后的结果将会是：

```
int x::y (int a)
{
```

```
    ...
    int b = 6;
    ...     // b is not modified within this section
    int m = 0x3F;;
    ...     // m is not modified in this section
    int n = 0x40;
    ...
}
```

调用间优化是内联性能等式的另一半。一个经过良好优化的编译器会使内联方法块边界的任何痕迹变得认不出来。通过优化，方法的大部分(在某些情况下是全部)将不再存在。编译器可以重排大量的方法。因此，把内联方法看成是维持某种内聚度在逻辑上是有用的。内联的主要好处之一就是并非必须这样做。

8.2 方法调用代价

要完全理解与内联性能收益有关的一切问题，必须理解与方法调用(过程调用)和返回有关的所有因素。这将有助于理解我们要避免什么，以及为什么避免这些方面就会显著地提高程序性能。

大多数系统有3或4个"管家"寄存器：一个 Instruction Pointer(尽管它不对程序计数，但它还是经常被称为 Program Counter)、一个 Link Register、一个 Statck Pointer、一个 Frame Pointer 和一个 Argument Pointer；或分别把它们称为 IP、LR、SP、FP 和 AP。您可能已经注意到这里列出了5个寄存器，而我们却说一个系统有3或4个寄存器。这是因为系统要混合搭配、组合使用它们的缘故。我们还不知道哪种系统使用的寄存器会少于这些寄存器中的三种，我们也不知道哪种系统会使用全部这5种寄存器。

下面对有关各种维护寄存器的功能进行简要说明。

Instruction Pointer(IP)包含下一次要执行的指令的地址。方法调用包括跳转到被调用方法的指令和对 IP 的相应修改。然而不能仅仅重写 IP，在修改它之前必须保存它原来的值，否则将没有办法返回调用方法。

Link Register(LR)包含调用当前方法的方法的 IP 的地址。这是方法执行完以后要返回的地方。LR 通常与系统结构的调用指令操作相联系，它的值是自动设置的，就像执行调用的副产品一样。它是单个寄存器，不是寄存器堆栈。如果方法本身要调用其他方法，那么必须把 LR 保存起来以防被重写。这是因为如果调用者的标识被破坏，就难以有效地从调用中返回。在某些体系结构中，自动地或是通过显式地把调用方法的 IP 压入程序的进程堆栈来实现 LR 的功能，后一种情况下，体系结构不会包含显式的 LR。

方法中的局部(自动)变量在进程的堆栈上分配。Stack Pointer(SP)跟踪已消耗的堆栈数量。每次调用都要消耗堆栈空间,每次返回都会释放先前分配的堆栈空间。类似于调用者 IP 和 LR,返回之后,必须通过对传递到堆栈的参数进行可能的调整来恢复堆栈。这意味着 SP 也必须作为方法调用序列的一部分保存起来。

Argument Pointer(AP)和 Frame Pointer(FP)是非常依赖于系统的。它们在某些体系结构中一个都没有,在另一些体系结构中只有一个,而在某些体系结构这两个寄存器都有。FP 用于标识堆栈上两个区域间的边界:调用方法在其中一个区域上保存那些其状态需要保留的寄存器,在另一个区域上存放被调用方法的自动变量。通常 SP 在方法执行期间有很大的易变性。FP 通常被用作方法中局部变量的不易变的引用指针。

良好的调用性能要求只保存那些被方法用到的寄存器。在每个调用中保存整个寄存器集是不必要的浪费。然而只保存部分寄存器会产生潜在的可变大小内存分配,这种情况发生在传递给方法的参数和方法的自动变量之间。如果可变数量的寄存器存储与一个给定的方法相关联(这就是说,所保存的寄存器值的数量独立于调用方法的状态),那么就需要一个 AP 来指明传递给方法的参数在堆栈中的位置(如图 8.1 所示)。

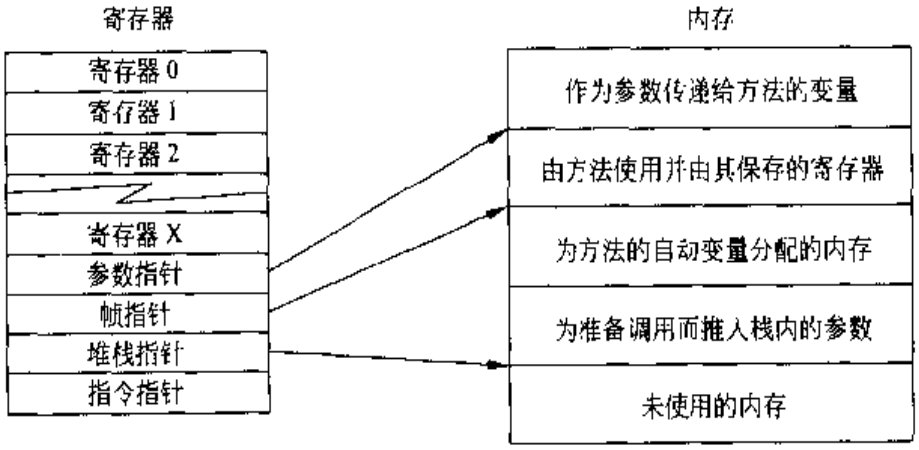


图 8.1　调用帧寄存器映射

一些新的处理系统使用了调用者/被调者保存模式。这种模式中,在不对被调用方法部分(被调用方法能够在不对调用方法产生任何负面影响的情况下重写自己的内容)采取任何行动的情况下,调用方法保证寄存器能够让被调用方法使用,并且某些寄存器在能够使用之前要由被调用方法进行保存。然而,即使是最成熟的调用和返回机制也不能完全避免在每次调用和返回时至少会有一些寄存器存储和恢复。

使用这些寄存器的典型调用序列包括如下步骤:

- 调用方法将要传递给被调用方法的参数进行排列。这通常意味着把参数压入堆栈,一般是以相反顺序压栈。在所有参数压入堆栈后,SP 将指向第一个参数。

- 要返回的指令地址被压入堆栈，然后调用指令转移到被调用方法的第一条指令处。
- 被调用方法把调用方法的 SP、AP 和 FP 保存到堆栈上，对“管家”进行调整，让它们反映被调用方法的上下文。
- 被调用方法同样保存(压入堆栈)将用到的任何其他寄存器。(必须做到这一点，以便方法返回后调用方法的上下文不会受到干扰。这通常是另外的 3 个或 4 个寄存器。)

清除先前调用序列的典型返回序列包括以下步骤：

- 如果方法返回一个值，那么该值通常从寄存器 0 中返回，有时也会是寄存器 1。这意味着寄存器 0 或寄存器 1 必须是便笺式寄存器(不作为方法调用和返回的一部分而保存和恢复的寄存器)。通过寄存器返回可简化为了返回而清除堆栈的工作。
- 从堆栈里把寄存器(由于前面被方法使用而被保存的)恢复到原来的位置。
- 必须把为调用者的 FP 和 AP 所保存的值恢复到它们的相应位置。
- 必须调整 SP，以便使它指向在把方法的第一个参数压入堆栈以前所指的位置。
- 从堆栈中获得返回地址并把它放到 IP 中，强制返回到调用方法中紧跟调用点的位置。

与方法调用相关联的一次简单数据移动意味着有 6～8 个寄存器(4 个维护寄存器和 2～4 个由方法使用的寄存器)被保存，然后其中 4 个被修改。这通常需要 12 个时钟周期(写入或读出内存的数据移动很少真正是一个时钟周期)，有些时候会超过 40 个时钟周期。因而与方法调用有关的工作在机器时钟周期方面的代价是十分昂贵的。不幸的是，这才只是开销的一半。方法返回时，必须清除所做的对调用有影响的事情。应从堆栈中恢复以前保存的值，机器的状态必须回到与调用前相接近的状态。这意味着方法调用通常要求大约 25～100 个时钟周期的开销，这有时还是保守的估计。

所谓“保守”，部分原因与参数的准备和获得有关。作为调用开端的一部分而压入堆栈的参数通常已被映射到被调用方法的内存映像中。对于引用来说，这总是正确的。对于指针和对象来说，这有时也是正确的。因而存在一种附加的开销，它与调用前把参数压入堆栈和在被调用方法内把它们从堆栈读回来相关联。某些情况下，这些参数通过寄存器传递，这样可提供良好的性能(尽管不是无偿的)，然而规范的机制涉及使用内存来传递参数。

如果方法返回一个值，特别是该返回值是一个对象时，也会产生一种代价，它来源于把被调用方法产生的对象复制到调用方法为返回值保留的内存中。对于一个巨大的对象来说，这项开销可能是显著的，特别是使用复杂的复制构造函数来做这项工作时(在这种情况下，我们产生了两份调用/返回开销：一份用于显式调用的方法，另一份被复制构造函数用于返回对象)。再考虑到调用者/被调用者通信因素和系统维护因素，方法调用的代

价大约是25～250个时钟周期。通常情况下被调用方法越大，这种开销越大，直到保存和恢复所有处理器的寄存器、传递大量的参数和调用用户定义方法以实现返回值构造等开销代价达到最大。

对异常处理的使用可能显著地减少内联返回值优化的全部潜能。返回值复制在逻辑上是由复制构造函数作为被调用方法的一部分所完成的原子级基本操作。这意味着在逻辑上，如果在执行返回语句之前抛出了异常，那么返回值就不会返回，要存放方法返回值的变量将保持不变。这在本质上要求在出现异常时为返回值使用复制语义。这也是在某种情况下避免使用异常的正当理由。某些针对异常的返回值优化已经成为可能。例如，如果存放返回值的变量其作用域与内联方法处于相同的try块内，那么就可以对返回值进行优化。不幸的是，尽管在大多数情况下这是相对容易确定的，但是需要调用间的优化，其开销是昂贵的，并且有点复杂。

使用内联的另一个优点是不需要为执行被调用方法而进行跳转。跳转，即使是无条件跳转，也会对现代的处理器产生负面影响。由于需要的指令不在预取缓存中，所以频繁跳转涉及执行管道的延迟。跳转也会要求算术单元的服务以确定跳转目标，因此将会延迟时间，直到得知跳转地址为止。延迟意味着在对指令流进行重定向期间，处理器将处于闲置状态。在每次方法调用时这种情况都会发生两次，一次是在方法被调用时，另一次是方法返回时。

还有一些与调用性能相抵触的因素。编译器可以很好地优化小型的代码窗口，就是说，编译器可以很好地优化单个方法，但是它们势必无法跨越方法调用边界进行优化。这意味着这种简单的优化就像把写入相同位置的两个连续内存块中的头一个删除一样，如果存在插入调用的话，那么这种优化将无法执行。例如：

```
int x = 10;
x = 20;
```

作为单个代码块来看，这有点无聊。很明显第一个语句 x = 10;的赋值操作可以忽略(假设x不是某种易失性内存位置，例如FIFO)。任何好的编译器都将把第一条和第二条语句合并成一条简单的int x = 20;语句。不幸的是，这种编码行为经常发生，并且未经编译器检测和优化，编译器在连续的赋值之间插入一条返回指令。例如：

```
int a::b(int& i)
{
    i = 10;
    ...
}
```

```
int a::c ( )
{
    ...
    int k = 0;
    ret = this->b(k);
    ...
}
```

对语句块进行了改变，把两个方法（其中一个调用另一个）变成了单个方法（包含了全部两个方法的功能）。这种改变尽管在逻辑上对于最终结果没有什么意义，但是它在改善编译器对低效编码进行优化的能力上却是至关重要的。因此，在某些情况下，方法调用最为显著的代价是无法跨越方法边界进行优化。

8.3 为何使用内联

内联可能是 C++ 中可用的效果最为显著的自动性能增强技术。在不进行任何重写的情况下，相对较大的系统的性能可以迅速改观。在为一个网络构架进行工作时，我们使用了一个 10 000 行代码的子系统并把它的性能提高了不止一个数量级。改进的最后一步与内联工作系统有关。最后一步仅花费两天时间，却使系统性能提高了 40%。内联工作之所以需要两天时间，是因为最初的设计是不使用内联的。如果在开发该系统期间使用了现在所知道的有关内联的一切，那么我们将通过改变 makefile 和重新编译来完成同样的内联工作。

这 40%的性能提高不是仔细配置和“快速路径”分析的结果，而是对方法集的粗略检查和对显而易见的内联编译候选项的选择。如果我们对性能有更高的要求，那么会更为严肃地对待创建“直线快速路径”，这是自动性能技术的真正目标。[1]

程序的快速路径是程序中支持程序执行的正常、无差错和一般使用情况的部分。通常少于 10%的程序代码依赖于这种快速路径。包含少于 1%程序代码的快速路径是不正常的。示范直线执行的代码使跳转（条件执行、循环和调用）的数量达到最小。这种特性对于高度优化性能的实现是至关重要的，我们将在第 16 章对此进行讨论。内联允许从快速路径中消除调用。在大多数系统中，调用是惟一最为昂贵的结构性实体。一般来说，依

---

1. 令人难过的是，我们所使用的系统严重受害于本书所述的每一个性能缺点，最终由于无法避免这种痛苦而被否决。谢天谢地，我们不用设计该系统了，但我们确实学到了大量的在从事 OO 设计时避免这种痛苦的经验性方法。像该系统一样的大型工程的失败大大地损害了 OO 设计的名声。我们认为该工程不是由于 OO 设计失败的，而是由于把性能作为第二考虑因素才失败的。在性能上的失败，会导致整个设计的失败。

靠大量小型方法作为主要构件的程序都不存在获得杰出性能的希望，除非确定并内联了这些方法的适当子集。

8.4 内联详述

尽管从表面上看相对简单，但是内联的概念本身是复杂的。尽管我们认为理解并正确使用内联是重要的，但是我们也认为内联最终是一种基于编译和配置的将由编译器/配置器/优化器来完成的优化动作。我们还认为自动化的工具最终将比您完成更多有关内联的精确工作。不幸的是，当前的编译器缺乏机动性，即使是在接近自动优化的内联方面。因此，学习所有与内联有关的奥妙和处理实现内联所需复杂句法规则的任务就落在了您的身上。也许将来有希望能够忘掉有关内联的一切，但除非是您在向后来者谈及这个话题："我记得有一段时间我们必须手工处理代码路径优化，孩子，那是一段美好的过去。"很有可能，在说这些话的时候，您将在思考，至少是有关内联的东西："孩子，那是一段糟糕的日子。"C++ 集成开发环境对文件语义的不妥当保存违背了对编译器机动性所期望的水平。

保留字"inline"是对编译器的一种建议。它告诉编译器，由于性能方面的原因，如果方法的代码能以内联的方式扩展而不是调用，那将会是一个不错的主意。编译器并没有被强迫同意内联请求。编译器可以内联方法，也可以不内联方法，这要看它是否愿意或能够。这就是说编译器可能在没有明确告诉它要内联的情况下对方法进行内联(价值不大的方法内联有时会对优化产生副作用)，也可能在明确告诉它要内联方法的情况下而没有内联方法。

一般情况下，编译器仍缺乏内联典型方法的经验。例如，有些编译器拒绝内联包含循环的方法，有些编译器不能处理内联方法或声明为虚函数的方法中的静态变量。某些情况下，编译器可能无法决定被调用方法变量空间的映射。实际上，有些编译器除了间接调用、存取器方法(除了设置和返回属性值外很少做其他事情的方法)、包含几行简单赋值语句(可能带一些相关计算)的方法之外，不会再内联其他方法。因此，主动内联的程序最后可能很轻易被编译器所阻止，这样的编译器无法满足程序员全部的或大量的内联请求。

对于内联还应注意一些引人注意的副作用。即使内联方法定义常常存在于单独的.inl文件中，但它们在逻辑上是类头文件的一部分。然后头文件及其在逻辑上所包含的.inl文件由使用它们的.c或.cpp*文件所包含。在把源文件编译成目标文件以后，就不

---

* 从现在开始，我们将把包含实现代码的C++源文件称作".cpp"文件。如果您所熟悉的编程环境使用".c"命名法，那么请使用相应的语法转换。

需在目标文件中包含任何指示来指明该目标文件包含了某些方法的内联实例。一般情况下，这就是说目标文件已经完全解析了内联方法，不再需要记录它的存在(不存在链接要求)。因此，尽管语言明确禁止，但是一个源文件仍可以和内联方法的定义一起编译，另一个源文件却可以和实际上是同一方法的另一个版本一起编译。注意到这一点的编译器将不把这种状况报告为一个错误，但是我们还不知道哪种编译器会这样做。

8.5 内联虚方法

我们在前面提到有些编译器拒绝内联虚方法。这可能看起来理所当然。毕竟，虚方法的绑定要延迟到运行时。一般情况下认为虚方法是通过函数指针表间接调用的。尽管这种关于虚方法的观点通常是正确的，但并不总是这样。以下面的代码段为例：

```
inline
virtual
int x::y ( char * a)
{
    ...
}

void z (char * b)
{
    x_base * x_pointer = new x (some_arguments_maybe);
    x x_instance (maybe_some_more_arguments);
    x_pointer->y (b);
    x_instance.y (b);
}
```

y ( )是一个虚方法，但是其绑定并不总是需要延迟到运行时。在以上例于中，y ( ) 由 x_pointer 和 x_instancc 调用。延迟虚方法绑定建立在使用对象指针的基础之上。任何由对象实例对虚方法进行的调用都将产生对虚方法实例的间接调用，该虚方法实例与该对象的类型相关联。在调用绑定中不存在延迟。这样的延迟将会产生相反的结果：对象的类型在编译时已知，没有必要也没有可能使用多态性。运行时虚方法确定的附加开销将由编译器在可能时随时避免，同时由于对象实例没有多态性，所以编译器总是创建对象实例对虚方法的间接调用。

在某些实例中，正如上面所讲的那个实例，在编译时也可以知道潜在的多态性指针类型，并且对虚方法的直接调用是可能的。对象的指针可以是多态的，但是如果与对象指针

相关联的对象是可见的，且没有使用实际类型未知的对象给对象指针赋值，那么编译器可以在编译时确定应该调用方法的哪个虚实例，同时能够产生直接调用而不是虚调用。对于 x_pointer 来说，它简单地确定（动态对象实例的创建是可见的，同时也没有插入对 x_pointer 的赋值，这种赋值可能改变这个基类指针所引用对象的类型）关于哪种 y ( ) 虚方法应该被调用的多态解决方案。现实中，很多虚方法调用是在编译时解析的。例如，许多在虚方法内由 this 指针对虚方法进行的调用都至少存在潜在的编译时解决方案。

这意味着如果编译器足够完善，则虚方法调用能够潜在被内联。因此，如果配置文件指明一些虚方法需要大量的程序执行时间，那么就存在通过内联这些方法来消除一些方法调用开销的可能。这同样说明如果编译器能够内联虚方法并且选择了内联，那么几乎可以保证既会有内联调用的实例，也会有对同一方法虚调用的实例。

8.6 通过内联获得性能

请考查内联一个简单的存取方法（存取对象属性的方法）所带来的影响：

```
int x::get_y ( )
{
    return y;
}
```

方法本身可能只需 3 或 4 个时钟周期。但如果调用它的话，即使是这么小的一个方法，它也会产生多至 20 个时钟周期的开销。如果内联该方法，那么它将只消耗 1 或 2 个时钟周期（比原来的指令数少 2 条是因为不再有调用和返回开销）。因此，内联可以把速度提高 10 倍。同时应该记住，开销中所节省的 20 个时钟周期中的大多数都有与其相关联的指令。这意味着除了节省了 20 个时钟周期的开销之外，我们还缩小了代码长度，其比例是每静态内联一个方法调用（调用内联方法的程序中调用位置的数量）就会减少 12 条指令。

请考查如下程序：

```
# include <iostreame.h>

// inline
int calc ( int a, int b)
{
    return a + b;
}
```

```
main ( )
{
    int x[1000];
    int y[1000];
    int z[1000];

    for ( int i = 0; i < 1000; ++i) {
        for ( int j = 0; j < 1000; ++j) {
            for ( int k = 0; k < 1000; ++k) {
                z[i] = calc (y[j], x[k]);
                }
            }
        }
}
```

为了测试内联对执行速度的影响，我们两次运行了该程序。在第一次运行时，内联了calc，消耗了8s(由此可以看出您的计算机比我的计算机快多少)。第二次运行时，对calc使用外联(不是内联)，消耗了62s。只是通过消除调用开销，就把性能提高了8倍之多。内联版本的代码尺寸也减小了30字节，在这种情况下，把代码尺寸减小了10%以上。内联方法的结果是使程序显著加快并使程序更小一些。

8.7 要　　点

- 内联就是用方法代码替换方法调用。
- 内联通过消除调用开销而提高性能，并允许进行调用间优化。
- 内联主要是一种执行时优化，尽管它同样也能够产生较小的可执行映像。

第 9 章

内联——性能方面的考虑

提高 C++ 性能的编程技术

9.1 调用间优化

正如我们讨论过的，通过避免昂贵的方法调用所带来的性能增益只是内联在性能方面的一半，另一半就是调用间优化。调用间优化允许编译器对方法执行源代码级和机器代码级的优化，这是建立在对方法调用的一种更为广泛的上下文关系认识的基础之上的。这些优化的方式是在编译时做一些事情，以避免运行时再去做这些事情。例如，下面这样一些简单的事情：

```
float x = 90.0;
...                          // nothing that changes x's value
float y = sin (x);
```

被转换成

```
float x = 90.0;
...
float y = 1.0;               // sin (90) = 1
```

不像在单个方法的上下文中，本例在调用/被调用方法上下文中变得十分常见，在这种上下文中，变量在一个方法中初始化，然后作为参数传递给另一个方法。我们再从另一种上下文环境来考虑上例：

```
enum TrigFuns {SIN, COS, TAN}

float calc_trig (TRIG_FUNS fun, float val)
{
    switch (fun) {
        case SIN:          return sin (val);
```

```
        case COS:    return cos (val);
        case TAN:    return tan (val);
        }
}

TrigFuns get_trig_fun ( )
{
    return SIN;
}

float get_float ( )
{
    return 90;
}

void calculator ( )
{
    ...
    TrigFuns tf = get_trig_fun ( );
    float value = get_float ( );
    reg0 = calc_trig (tf, value );
    ...
}
```

如果把 get_trig_fun、get_float 和 calculator 全部内联，那么上面的代码段将解析成一条简单的 reg0 = 1.0 语句。相反，如果没有内联，那么由于在单个方法内没有实施优化所需的任何信息，并且通常情况下只有方法内的优化才可用，所以编译器将无法发现这种简单的优化。这是一个有关内联如何让编译器实行代码优化的简单例子，它基于那些在没有内联影响的情况下需要调用间优化的需求。

与简单的避免调用相比，这种调用间代码优化可能获得的性能更为显著。另一方面，从避免调用来获得性能是有保障的，尽管它们不显著，但却是常用的。代码优化与编译器密切相关，高级优化可能使编译过程耗费大量时间，有时候实际上是在分解代码。内联调用间的优化是寓言中的兔子，而避免调用则是与内联性能获取相关的乌龟。

不幸的是，优化有时使用对变量易混淆和易变性的假设，这是与源代码相冲突的。优化会分解那些使用对象间预期关系(但这些关系不被保障)的代码。例如，如果程序假设变量以某种顺序保存在堆栈中，那么该程序很容易会在某种编译器上走样。这种编译器只选择以寄存器为存储变量的基础，而不为变量保留任何堆栈空间。幸运的是，许多简单的优化不需要作出大量的假设。如果编译器优化对您很有价值，那么应该参考编译器构

造的详细说明。

尽管在编译器优化方面的话题扯得太远，已超出我们所讨论的范围，但是我们能够指出直接量在一些最为有效的优化中所扮演的重要角色。上一个例子很精彩，它说明了直接量所提供的方法，在该例中，直接量的三角函数 sin ( )在编译时进行解析。如果直接量和相关变量在同一方法内定义和使用，则可以轻松地实施这种优化。如果直接量以参数的形式传递进来，那么这将是不可能的。

看一下另一个简单的例子：

```
int i = 100;
...                 // nothing that changes i's value
if ( i > 10 ) {
    ...             // 20 instructions
    }
else {
    ...             // 50 more instructions
    }
```

这段代码直接简化为：

```
int   = 100;
...                 // nothing that changes i's value
...                 // 20 instructions
```

这把代码尺寸减小了几百字节，同时清除了条件判断和分支。在这种情况下，没有巨大的收获，但这是有效的。您可能已注意到把 i 设成 100，而后又转去测试它是很愚蠢的，是的，事情确实是这样。在单个方法的上下文中，这是愚蠢的。相反的是，如果变量在一个方法内初始化，然后又传递到另一个方法中去，这种事情则是经常性发生的。在对输入参数进行范围检测的保护性编程例程中，这是一种非常常见的事情。

请考虑如下包含 case 语句的方法：

```
inline
bool is_hex (char c, int& value)
{
    switch (c ) {
        case '0': value = 0; break;
        case '1': value = 1; break;
        case '2': value = 2; break;
        case '3': value = 3; break;
```

```
        case '4': value = 4; break;
        case '5': value = 5; break;
        case '6': value = 6; break;
        case '7': value = 7; break;
        case '8': value = 8; break;
        case '9': value = 9; break;
        case 'a': value = 10; break;
        case 'A': value = 10; break;
        case 'b': value = 11; break;
        case 'B': value = 11; break;
        case 'c': value = 12; break;
        case 'C': value = 12; break;
        case 'd': value = 13; break;
        case 'D': value = 13; break;
        case 'e': value = 14; break;
        case 'E': value = 14; break;
        case 'f': value = 15; break;
        case 'F': value = 15; break;
        default: return false;
        }
    return true;
}
```

有一些跳转表机制，它们实际上能够使这种判断成为相对有效的方法并且其代码的范围检查版本无疑将会更快一些，但是这个版本说明了一个重要观点。您一般情况下不会内联这样的例程。根据编译器的完善程度不同，该例程会包含 10～100 条指令不等，它只是看起来大。但是即使在其最小的实现中，该例程也将产生一些代码扩展。相反，我们研究一下在 is_hex ( )内联版本中直接量输入参数的影响。直接量输入参数允许编译器把代码缩小成单独的一条赋值语句。可能看起来不是通过直接量来调用该方法，但是可以想象以下代码中两层内联的影响：

```
inline
int parse_hex ( char * cp)
{
    int ret = 0;
    int temp;
    while ( is_hex ( * cp, temp)) {
```

```
        ret = (ret << 4) + temp;
        ++cp;
        }

    return ret;
}

main()
{
    ...
    char* alpha_number = "12345678";
    ...
    int bin_number = parse_hex(alpha_number);
    ...
}
```

优秀的编译器会把对 parse_hex( )的内联调用压缩成使用直接量整数 0x12345678 所进行的简单赋值。这种对内联的使用把几百条运行时指令替换成了一条单独的直接赋值指令。不仅如此,它还允许在已知 bin_number 取值的基础上进行其他优化。

通过向接收单个参数的递归方法传递直接量(就像斐波纳契数列产生器这种在教授递归时所使用的古老备用示例),我们可以说明在内联方法和直接量之间进行优化的更为复杂的可能性。(不管它的递归定义如何,使用这么陈旧的示例都是很痛苦的,特别是在它如此不适合于递归解决方案的情况下。然而,它的简便性和人们对它的普遍熟悉程度使得它成为易于使用的目标示例,且不说通过编译时确定直接量所获得的几乎无法令人置信的性能。)

```
inline
int get_fib( int x )
{
    if (x <= 0) {
        return 0;
        }
    if (x == 1) {
        return 1;
        }
    if (x == 2) {
```

```
        return 1;
        }
    else {
        return get_fib ( x - 1 ) + get_fib ( x - 2 );
        }
}
```

使用直接量参数调用 get_fib 可以在编译时得到解析。例如:get_fib (10)可以在编译时由直接量整数 55 替换,否则将会产生 109 次方法调用。大多数编译器当前还不能提供与内联方法编译时优化有关的这种层次的完善,不过,将来这种优化将会变得易于实现。虽说暂时有些昂贵,需要调用间优化,但是很久以后,这种优化将根本不再与内联相关,它将简单地变成一个例程。

您可能开始认识到,内联更应该是编译器要进行的优化,而不是由程序员进行的优化。我们同意这种说法。许多真正有价值的内联是具有选择性的。这就是说,只在某些情况下才内联方法。直接量与内联方法之间的相互影响是一个很好的例子。如果编译器能够在编译时确定方法的重要输入参数,那么编译器或许能够进行一种十分划算的优化。不幸的是,由于这些优化不能普遍应用于方法调用的所有情况,所以它们超出了 C++ 的基本内联协议。第 10 章将讨论程序控制的选择性内联。目前,这些技术可以通过认真的人工调整来实现,不过,我们希望以后编译器能为我们完成这些事情。

9.2 为何不使用内联

如果内联这么好,我们何不干脆对一切进行内联? 这个简单的问题有着很复杂的答案。我们从内联条件开始寻找答案。假设在编译时内联一个包含 550 字节源代码的方法。并假设被调用方法中有 50 字节与调用的开始和结束(方法调用开销)相关。如果我们假设的方法被静态地调用 12 次(从程序中的 12 个不同位置进行调用),我们恰好把程序尺寸增大了 5 450 条指令((每次内联 550 条指令－调用开销的 50 条指令) * 12) 其他被调用版本的 550 条),同时我们因内联每个方法的执行而提高了执行性能,假如说只有 10%。(假设这个巨大的方法有 50 个时钟周期的调用开销以及它需要执行 500 个时钟周期。这里纯粹是假设。一些包含 500 条机器指令的方法可能平均需要 10 个时钟周期的执行时间,而另一些可能需要上百万个时钟周期。)这样,我们使这个方法的代码尺寸增加了 10 倍,而对于每次调用只有微小的性能提高。当类推到程序中所有能够内联的方法时,这种数量级的代码膨胀将对性能产生巨大的负面影响,如缓存失败和页面错误,这将使所假设的任何主要收获变得微小。另一方面,过分内联的程序将执行较少的指令,但

是需要更长的时间来执行。因此，不内联所有方法的原因之一就是无法容忍内联所产生的代码膨胀。

随着磁盘空间和内存容量的成倍增长，可以假定可执行映像的大小已不成问题，特别是为促进程序的执行而采用了虚拟内存管理机制以后。然而代码尺寸不单单是存储问题。单个内联方法的多个实例意味着每个实例要拥有自己的地址，因而也将消耗相互独立的缓存，结果是减少了有效缓存空间从而使其容量成倍丢失。例如，假设均匀地从程序的4个不同的位置调用一个内联方法。第一次执行该方法时，由于必然的缓存失败，代码缓存将出现错误。这种必然的缓存失败与程序对内联方法的初始化调用(执行)相关。由于内联代码在每次内联时都各不相同，所以在从另一个内联代码位置再次执行该方法时，缓存将再次丢失。即使是内联代码相同，内联指令也将出现在进程代码空间的不同地址中。在我们这个有4次频繁调用的例子中，内联方法所消耗的时间将是外联的4倍。内联至少要比外联版本多出现4次缓存错误。我们之所以说至少4次，是因为该方法的一个或多个缓存实例被清除出缓存并需要重新加载的可能性更大。单个外联实例将有很高的使用率因而被清除的可能性要小得多。由于内联而导致的缓存性能降低可能掩盖与调用和返回开销相关的性能提高，特别是对于大型方法更是如此。

因不加选择地使用内联而引发的缓存错误可能变成程序的高度退化。假设执行流的页面恰好装进处理器的缓存，就是说，由于程序代码空间中负责大部分程序执行的部分全部装进缓存，所以指令流几乎不出现缓存错误。现在设想我们把通过不加选择的内联而把页面增大两倍的情况。由于缓存不再足够以容纳页面，所以变大了的代码路径将频繁地产生错误。这种缓存错误的增加将严重地影响性能，它抵消了通过避免方法调用和返回而从另一方面获得的性能提高。因此，不内联所有方法的第二个原因是可能无法忍受代码膨胀的副作用。

内联代码膨胀有时会出现另一个退化特征。内联一个方法有时会产生指数级的代码增长，当相对庞大的方法被多层内联时会出现这种情况。以A、B、C和D 4个方法为例，每个方法都包含500字节的指令。

```
int D ( )
{
    … // 500 code bytes of functionality
}

int C ( )
{
    D ( );
    … // 500 code bytes of functionality
```

```
    D( );
}

int B( )
{
    C( );
    … // 500 code bytes of functionality
    C( );
}

int A( )
{
    B( );
    … // 500 code bytes of functionality
    B( );
}

int main( )
{
    A( ); A( ); A( ); A( ); A( );
    A( ); A( ); A( ); A( ); A( );
}
```

A 调用了 B 两次，B 调用了 C 两次，C 调用了 D 两次，main 调用了 A 十次。内联 A，B，C 和 D 将把与这 4 个方法相关的代码尺寸增加 70K 字节。为了提供这种功能使代码尺寸增加了 37 倍。尽管上述例子非常简单，一般也不会完全像我们所展示的那样出现，然而事实是过度内联会使代码尺寸膨胀。因此，不内联所有方法的第三个原因是无法容忍代码膨胀。

如果您刚刚往回看了几页并对自己说："嗨，不进行内联的这前三个原因都是同一个！"那么您是正确的。如果没有察觉到，那说明您的注意力不集中，可能需要来点咖啡因了。代码膨胀是不全部进行内联的主要原因。随着编译器质量的提高以及编译器在内联方法内解析局部静态产物能力的加强，这一点会变得更为正确。

还有其他一些不进行内联的原因，尤其是大型工程和使用复杂方法的情况。内联方法会带来复杂性，这种复杂性不仅取决于方法的接口，还取决于方法的实现。因此，在程序开发期间要求经常改动的方法不是特别适合内联。就像这样一条一般的规律：任何减小代码尺寸的内联都是好的，任何显著增大代码尺寸的内联都是不佳的。第二条规律就是说不应内联其实现容易发生改变的方法。

不对一切进行内联的另一个原因是有些方法不能内联：例如，不能内联递归方法。假设某个方法 A 调用它自己。由于编译器不断地试图把 A 插入 A，所以任何内联 A 的尝试都将导致无限循环。（实际上在某些情况下，特别聪明的编译器能够内联递归函数，特别是在控制递归的变量作为直接量传递进来的情况下。）因此，通常不能内联递归方法（尽管以后我们要讨论一些获得类似最终效果的机制）。有时可以内联间接递归的方法。例如，A 调用 B，B 又调用 A，这里尽管是间接递归，但是 B 可以被内联。

9.3 开发阶段和编译时的内联考虑

逻辑上内联方法必须出现在其类的头文件中。这对于内联方法的调用者生成它们的代码体是必要的。不幸的是，这意味着对内联方法体的任何改变都要求重新编译所有使用它的模块，而不仅仅是重新链接。对于大型程序来说，这实际上增加了程序开发时间，这种增加源于每次编译所消耗的附加时间。

由于单个断点不能用于跟踪内联方法的入口和出口，所以内联方法的调试是很复杂的。存在几种使用观察点来完成同样工作的方式，尽管这样做有损调试器的性能。跨越调用方法和被调用方法的源代码边界来跟踪变量名同样是困难的。

内联方法通常不出现在程序的配置表（该表在样本执行的基础上指示程序执行行为）中。有时对内联方法的“调用”对配置表是不可见的。一旦内联了一个方法，就必须假定只要包含它的方法执行，它就会执行，尽管在有些情况下这是不必要的。就像使用汇编语言重写方法一样，这使确定极限性能测量的工作变得复杂。

正确工作的编译器不会产生作为内联副产品的错误。而编译器是十分复杂的软件，C++ 也是十分复杂的语言，因此短期内似乎不存在理想的没有差错的 C++ 编译器。这就使得内联优化自身产生错误成为可能。由于编译器自由地决定其所接受的内联请求的复杂程度，所以在内联方面，出现这种情况的可能性相对较小。在另一方面，内联引起的错误通常难以发现，就像大多数由编译器所产生的错误一样。

9.4 基于配置的内联

有一些内联策略看起来微不足道，而有一些则肯定有积极作用。还有一些递归方法迫切要求通过调用展开来获得良好的性能。然而，事实是我们可能为了解决问题而对内联投入了大量精力，显著地增大了程序尺寸和编译时间，而最后只看到十分微小的性能提高，甚至是性能下降。相反，我们可以内联很多重要的方法并看到显著的性能提高。无意义地使用内联与显著地提高程序速度的区别在于选择正确的内联方法。发现正确方法的

最好方式是通过配置。当存在能够用于产生配置数据的代表性数据样本时，这种方法是最有效的。

配置是一种性能测试技术，它依靠配置工具（软件套件）给程序提供一种工具（插入测试代码），这样就可以在程序样本执行期间对程序性能进行定性。配置质量直接依赖于产生配置的样本数据质量。配置文件以不同的形式和大小出现，它们的可能输入范围也有很大差别。但是在一般情况下，所有的配置文件都要提供如下信息：正在执行的方法，这些方法执行的频率。我们所熟悉的大多数配置文件还提供执行跟踪。使用这些信息，程序员可以进行信息化的内联决策。

方法有两种不同的尺寸：动态的和静态的。静态方法尺寸是将方法编译成机器代码后的字节数。这是每次内联方法时加入到程序中的附加代码数量的上限。动态方法尺寸是每次调用方法时执行的指令数，或者每次调用消耗的执行时间。这将提供调用/返回开销与方法执行时间之比。尽管有时包含较少指令的方法需要出人意料的大量执行时间，但是通常在方法尺寸和执行时间之间存在很强的相关性。带有多层循环嵌套的方法就是一个例子，但是看起来简单的方法有时也会由于缓存交互不佳等原因而导致效率低下（与不可缓存数据交互的方法就是一个良好例子）。

静态方法尺寸是编译的产物，也就是说，它是在编译时确定的。动态方法尺寸会显著地受到运行时产物的影响。这依赖于在产生配置数据时对良好样本数据的使用。如果用于产生配置数据的程序样本的执行具有代表性，那么就会产生合理而正确的动态方法尺寸。相反，不具代表性的样本数据可能提供不稳定的结果。配置常常可能不提供有关方法的任何数据，也就是说，有些方法在执行过后很少看到。这样的方法一般不会出现在配置中，或者说它们的调用频率非常低。

配置文件既要计算指令，也要测量时间。时间是一个精确的度量标准，但是指令数更容易获得并且可以提供良好的数据。使用这些数据可以进行信息化的内联决策。基于指令数进行性能测试的问题是：

- 即使是声称对大多数指令执行一条单独指令的体系结构，实际上也存在某些指令需要的时间比其他指令长。加载和存储指令通常需要逻辑和算术指令的两倍时长。
- 在单条指令执行时间变化很大的体系结构中，指令数可能具有很强的误导性。有些指令可能在一个时钟周期内执行完，而另外一些则有可能需要几十个时钟周期。因此，10 条快的指令所消耗的时钟周期可能少于一条慢的指令。
- 指令数完全忽略了低层体系结构对性能的影响。这包括缓存效率、跳转对管道的影响、寄存器相对于内存的使用和内存访问等待时间等。

幸运的是，配置文件通常会提供可比指令数度量，这进一步强调了指令执行时间的差

异，对于相对庞大的指令数量，这种差异将趋向于达到平均数。不幸的是，内联决策通常是在较少指令数度量的基础上作出的，与系统体系结构细节之间的交互很容易使原始指令数度量失色。

我们遇到过一个例子，在该例中用指令数作为度量不是特别有效。一位同事发明了一种计算 TCP/IP 校验和的算法，这种算法明显地要更快一些。但是他在采用这种新技术时遇到很大的阻力，原因是这种新方法的配置比以前的方法展现出更为庞大的指令数。直到其他程序员放弃性能等价于指令数这种狭隘的观点，并采用了性能是时间消耗问题这种视野更为开阔的观点以后，这位同事的那个更庞大但更快的算法才得以采用。这就是说，即使更有可能提供指令数度量，也不要设想指令数能够说明一切。

一个简单的配置会说明方法被调用的次数以及每个方法消耗的全部执行时间所占的百分比。一般情况下，该信息足以对配置候选方案进行合理的选择。对产生庞大动态尺寸的方法进行内联时，即使是它们有很高的调用率也很少会获得大幅度性能提高。其调用/返回开销与总执行时间之间的比率非常小，性能上的微小提高通常不易看到。另一方面，如果内联一个动态尺寸小而调用率高的方法，将会获得显著的性能提高。通常情况下小的动态尺寸和小的静态尺寸是强相关的。然而有时候小的动态尺寸会与产生巨大静态尺寸的方法相关联。内联这样的方法将导致代码膨胀，要小心行事。有时可以重写这样的方法，把核心功能抽取到内联例程中。以后我们将介绍这项技术。

表 9.1 提供了一套合理的内联方针。

表 9.1 内联决策矩阵

动态频率	静态尺寸		
	大型(超过 20 行代码)	中型(5～20 行代码)	小型(少于 5 行代码)
低(80%以下的调用频率)	不内联	不内联	如果您有时间和耐心，则可以内联
中等(80%～95%的调用频率)	不内联	重写该方法以展示出其快速路径并进行内联	总是内联
高(95%以上的调用频率)	重写该方法以展示出其快速路径并进行内联	选择性地内联高频率静态调用点	总是内联

尽管基本的程序配置将说明哪些方法执行几率高，但是通常不提供调用点频率。也就是说，如果方法被调用了 1 000 次并且在代码中有 20 个静态调用点，那么基本的配置有可能说明每次调用与每个调用点之间的百分比。对中等大小而动态和静态调用频率高的方法进行内联可以从该信息中获得很大益处。如果方法静态出现只占小部分，说明动

态执行占有重要分量，那么选择内联(这是只内联某些方法调用的一种机制，我们将在下一章进行讨论)其高利用率的部分是可取的。如果代码膨胀不成为问题，那么对这种方法的内联通常是可以接受的。请记住，对二级缓存和换页的影响是代码膨胀的副产品。

程序往往拥有快速路径，以致产生了许多大型方法。有时，可以将方法重写以展示出重要的、所期望的方法行为。插一句题外话，可以把方法中的错误校正部分转移到另一个方法中去。以下面的方法为例：

```
int x::y (int a, int b)
{
    if ( /* sanity check on input values for a and b */ ) {
        ...          // inline error handling (30 lines )
        }
    ...              // Real work (5 lines )
}
```

方法 x::y 大约有 40 条语句，这使它成为一个相对较大的方法。如果方法有高调用频率及大量调用点，那么从表面上看，内联 x::y 不会特别有效，它可能产生超过 1K 的代码映像尺寸。如果它不是一个叶节点，内联可能造成一些我们前面谈到的组合性代码尺寸爆炸。相反，在 x::y 内隐藏着另一个方法，它可以被非常有效地内联。请考虑将 x::y 划分成两个方法后的情形：

```
void x::handle_input_error ( int& a, int& b)
{
// error handling (30 lines)
}

inline
int x::new_y ( int a, int b)
{
    if ( /* sanity check on input values for a and b */ ) {
        handle_input_error ( a, b);
        }
    ...              // Real work of the method (5 lines )
}
```

new_y( )的代码尺寸减小到 10 行。尽管仍然不少，但是作为内联方法来说已经比较容易承受了。这只有在参数输入错误不频繁发生的情况下才有效。可以采用这种为内联而重写方法的做法，这往往是要把方法表示成小动态尺寸和大静态尺寸。重写这种类型

的方法可以在不增加代码尺寸的情况下使性能显著提高。

9.5 内联规则

在作出内联决策时可以使用一些相对简单的规则。惟一和微小总是意味着代码尺寸和性能方面的收益。微小可以在不考虑使用频率的情况下进行内联，惟一正好是由于使用频率才内联的。通常这些不费思索的内联是足够的，否则配置将指明最有可能的其他候选方法。下一章将介绍有益地内联较大方法的其他机制。

9.5.1 惟一

惟一方法是指在程序中只有一个调用点的方法。这并不意味着它在程序执行时只被调用了一次，惟一方法可能存在于循环中从而被调用成百万次，但是只要方法在程序中只有一个调用点，那它就是惟一方法。惟一方法天生就适合内联。只有一个调用点的事实说明不需要考虑惟一方法的尺寸和调用频率，惟一方法内联后，合成后的代码将比以前更小更快。性能提高可能不大，即使这样，您自己所做的内联努力也无法保证会产生这种性能提高。

通常，惟一方法同样难以识别。方法的惟一属性通常是临时的并可能与环境有关的。另外，惟一属性不是设计的产物。后一种类型总是显式地内联，前一种类型可以被内联，但是要求监视其惟一属性以便避免负面结果。

如果编译器能自动识别和内联所有的惟一方法就好了。不幸的是，做到这一点要求对程序的调用树进行全局分析。尽管全局调用树的产生是相对简单的，但是如果程序包含大量的独立编译模块，则代价有点昂贵了。当然，这不是在代码开发和调试阶段完成的分析类型，但是它是一种有用的后期优化。注意，对于大型程序来说，这种分析可能是处理器密集型的工作——在等待的时候拿点东西来读(如果喜欢感伤型的，那么托尔斯泰的小说可能比较合适)。

像前面提到的那样，编译器通常不允许内联复杂的方法。这使得对惟一方法的例程内联更成为一种理论上的收益，而不是现实的收益。能够内联所有惟一方法的完善的编译器总能带来好处，但是在编译器仍处于相对不完善阶段的特定情况下，要使惟一方法的内联成为标准功能还有很长一段路要走。

9.5.2 微小

微小方法通常是少于4条源代码级语句的方法，这些源代码级的语句被编译成10条以下的汇编语言指令。这些方法特别小，以至不会有任何使代码显著膨胀的可能性。较

小的微小方法实际上将减小总的代码尺寸，较大的微小方法可能会使总的代码尺寸稍有增大。全局微小方法内联的总效果可保证代码尺寸不受影响。在 C++ 中，微小方法经常出现。存取器方法、间接调用和简单的操作符重载往往是小型的方法，通常只有一行代码，但是这些方法经常使用。内联这些方法将显著地减少与维护正确的对象封装层次相关的性能损失。

有些编译器将自动地内联那些在复杂的上下文中可见的微小方法。这意味着如果编译器能够识别出微小方法，会将它们内联到分离的编译模块中，但是对那些跨模块的微小方法调用将无法进行内联。这种自动内联是一种非常方便的功能，这种功能可扩展到全局的基于编译的微小方法内联。编译器是测量调用开销和查看方法指令数量的最佳位置。这将总是使编译器能够内联 code、size、neutral 或较小的方法。更为完善的基于编译器的内联决策可以建立在程序员所提供的可接受代码扩充度量的基础之上，或是建立在有关代码尺寸和性能相对重要性说明的基础之上。这种能力很容易合并到自动惟一方法内联中去(合并使得可以把完整的雨果巨著加入到您的阅读清单中)。

要注意在内联微小方法时需要确保内联后的微小方法仍保持微小。如果内联的微小方法范围增长，程序员则应准备降级该方法。当该方法不是叶方法(叶方法不包含方法调用，它们执行后返回)时这一点尤其正确。意料之外的代码爆炸可能由内联非叶(内部)方法导致。叶方法的代码扩充量很容易确定。首先计算程序对该方法的静态调用次数，然后用该方法包含的指令数减去调用该方法需要的指令数，用所得的差乘以静态调用次数即可。在内联非叶方法时，任何由叶方法扩充所产生的扩充量都要翻倍。

9.6 要　点

- 直接量参数和内联结合在一起，为编译器显著地提高性能提供了充分的机会。
- 内联可能产生与预期相反的结果，过度内联几乎肯定如此。内联可能会增大代码尺寸，而代码尺寸较大会比代码尺寸较小遇到更多的缓存失败和页面错误。
- 非微小方法的内联决策应建立在样本执行配置文件的基础上，而非主观臆想的基础上。
- 对于那些静态尺寸大、动态尺寸小而调用频率高的方法，可以考虑重写，以便抽取它们的重要动态特性，然后内联动态部分。
- 微小和惟一方法总是可以内联的。

第 10 章

内联技巧

提高 C++ 性能的编程技术

有许多可以使内联更为有效的“技巧”。本章致力于这些技巧。作为编译选项和优化，这些机制正变得更加有用。在投入大量精力亲自实践这些技巧之前，请首先检查您的编译器可以做什么事情。

10.1 条件内联

编译、调试和配置等与内联有关的消极因素对把内联推迟到开发周期的较后阶段起到了有力的推动作用，通常是把内联推迟到大部分调试完成以后。理想状况下，内联决策是建立在配置结果的基础之上的。大多数编译器能够使用编译行开关来阻止内联，但是如果您的编译器不能如此，那么条件内联是构造您自己的编译行开关的一种简单方式。

令人难过的是，配置要求程序员做出适当的努力。没有人愿意在服务于性能的内联状态与服务于正常测试的外联状态之间反复改变方法。幸运的是可用预处理器来使方法在内联与外联之间轻松地转移。基本思想是使用编译行参数向编译器传递宏定义。输入参数用于定义名为 INLINE 的宏，也可忽略输入参数使 INLINE 保持未定义的状态。这种技术依赖于把所有潜在要内联的方法的定义与要外联的方法划分开。外联方法包括在标准的 .c 文件中，潜在要内联的方法放在 .inl 文件中。如果要内联 .inl 文件中的方法，可以在编译命令中使用－D 选项来定义 INLINE。下例说明了如何使用该编译时内联开关：

文件：x.h：

```
# if ! defined (_X_H_)
# define _X_H_

class X {
    ...
```

```
    int y ( int a );
    ...
};

# if defined (INLINE)
# include x.inl
# endif
# endif   // _X_H_
```

文件：x.inl：

```
# if ! defined ( INLINE )
# define inline
# endif

inline
int x::y ( int a)
{
    ...
}
```

文件：x.c：

```
# if ! defined (INLINE)
# include x.inl
# endif
```

.h 文件以通常的方式使用。如果定义了 INLINE，那么.h 文件将包含 .inl 文件，加在每个方法前面的 inline 指示符将不受影响。如果没有定义 INLINE，则.h 文件将不包含内联后的方法，而是把这些方法包含在.c 文件中，inline 指示符将被从每个方法的前面清除。这种机制为内联方法在内联状态和外联状态之间转换提供了一种方便的灵活性。这种技术是一种要么全有要么全无的方法。此外要注意在创建内联或外联的子部分时的情况。

10.2 选择性内联

C++ 最为严重的缺点之一是其语法和内联机制的灵活性。尽管这通常是有用的，但却是一种令人痛苦的缺陷。找不到用于选择性内联的机制，也就是说，无法在某些地方内联一个方法，而在别处又不内联该方法。这使内联决策成了一种要么全是要么全不是的

选择，从而忽略了快速路径优化的现实情况。例如，假设一个程序的快速路径（典型的执行序列）包含对一个方法的两次静态调用，在快速路径之外还有另外 20 次对该方法的静态调用。没有一种机制可以在两个重要的调用位置内联该方法而在其他调用位置使用一般的调用机制。我们将向您展示实现这一点的办法——这是相当简单和非常有效的。不幸的是，它要求训练有素，因而难以推广使用这种机制。

选择性内联方法的最简单方式是复制该方法的一个外联版本并把它放到 .h 文件或 .inl 文件中。然后把该方法的复制版本的名称改为 inline__whatever_the_original_method_name_was，在相应的 .h 文件中为该方法添加一个原型，使用字符串 inline_给希望有条件内联的方法名称实例加上前缀。这种机制笨拙但有效。它有明显的缺点，那就是需要分别维护同一方法的两个版本。该缺点可以这样解决：为需要内联的方法复制另一份实例，放到 .inl 文件中并改名为 _whatever_it_was_before，使它成为内联方法，然后把 .c 文件中原来版本的代码体和 inline__whatever_the_original_method_name_was 替换成对名为 _whatever_it_was_before 的现有内联方法的调用。下例将有助于把这一切说得更清楚。

假设我们想选择性地内联以下方法：

```
int x::y ( int a)
{
    ...
}
```

该方法位于名为 x.c 的文件中。通过向所提到的文件中添加以下代码将选择性地内联该方法。

文件：x.h：

```
class x {

public:  // assuming the original method "y" had public visibility
    ...
    int inline_y ( int a);
    int y (int a);                  // this should already have been here
    ...
};
    # include "x.inl"               // this may already have been here also
```

文件：x.inl：

```
inline
```

```
int x::inline_y ( int a)
{
    ...              /. original implementation of y
}
```

文件:x.c:

```
int x::y (int a)
{
    return inline_y (a);
}
```

这样便给出了两种版本的 y:一种是内联的,名称为 inline_y,另一种是外联的,名称为 y。我们还有 x::y 代码体的单个共享定义,就是位于 inline_y 定义中的那个。这样还有其他便利,即单个方法体还会为方法内的任何静态变量产生惟一的实例。

10.3 递归内联

正如以前所提到的,直接递归方法不能内联。对于许多依靠递归来插入、删除和搜索的相对简单的数据结构来说这很成问题。这些递归方法通常是直接递归而且很小。如果我们被迫在不使用内联或其他调用压缩方式的情况下执行这些方法,那么性能将遭到很大的损害。下面介绍一些迭代调用压缩和递归调用展开方面的机制。

一些递归方法是尾部递归。尾部递归由一个方法所证明,该方法递归下降,直到达到其基本情形。此时要进行一些操作,然后方法终止,有可能返回一个值。与尾部递归相关联的返回序列的显著特征是,决定了基本情形以后,可能除了由基本情形方案标识的返回值外,不会有什么别的操作。典型的二叉树搜索就是尾部递归方法的典型示例:

```
binary_tree * binary_tree::find ( int key ) {
    if ( key = = id ) {
        return this;
        }
    else if ( key > id ) {
        if ( right ) return right find (key );
        }
    else {
        if ( left ) return left.find ( key );
        }
        return 0;
```

```
}
```

像我们所能看到的那样，除了返回一个指向指定对象的指针外，在基本情形满足后不会采取任何操作。递归所产生的调用堆栈的上下文并不重要。事实上，如果编译器简单地保留一个容纳 this 的变量，那么该方法可以在不产生新方法上下文的情况下执行。请考虑另一种实现：

```
binary_tree* binary_tree::find ( int key ) {
    binary_tree *temp = this;
    while (temp) {
        if ( key == temp->id ) {
            return this;
            }
        else if ( key > id ) {
            temp = right;
            }
        else  {
            temp = left;
            }
        }
    return 0;
}
```

这种实现说明尾部递归方法能够转换成迭代方式。优秀的编译器可识别出尾部递归并把它从尾部递归方法转换成。这种优化使得任何使用内联来提高性能的进一步尝试都不是太有价值。作为一般的规则，特别是在性能十分重要的情况下，只要迭代方案相当简单，就应注意使用迭代方案代替递归方案。

当迭代方案比较复杂而且性能很重要时，可以使用一些相对简单的机制来展开递归方法。假设我们对提高非尾部递归方法的性能感兴趣，比如说在二叉树中产生 id 字段的中缀列表的方法：

```
void binary_tree::key_out ( )
{
    if ( left ) left->key_out ( );
    cout << id << endl;
    if ( right ) right->key_out ( );
}
```

可以使用内联把 key_out 按通常方式展开一次：

```
inline
void binary_tree::UNROLLED_key_out ( )
{
    if ( left ) left  >key_out ( );
    cout << id << endl;
    if ( right ) right->key_out ( );
}

void binary_tree::key_out ( )
{
    if ( left ) left-> UNROLLED_key_out ( );
    cout << id << endl;
    if ( right ) right-> UNROLLED_key_out ( );
}
```

程序员调用 key_out 方法，key_out 方法再依次调用 UNROLLED_key_out 方法。然后 UNROLLED_key_out 再间接递归调用 key_out 方法。可以在标准版本中内联展开的版本，这将导致标准版本在其自身内部嵌入两个自身版本。这种版本大约是原来版本的两倍大，但是比原来的版本要快 2～3 倍。这种对内联的使用方法可生成一种易于学习的版本，程序员自己可能已经写出来了：

```
void binary_tree::key_out ( )
{
    if ( left ) {
        if ( left->left ) left->left->key_out ( );
            cout << left->id << endl;
            if ( left->right ) left->right->key_out ( );
            }
    cout << id << endl;
    if ( right ) {
        if ( right->left ) right->left->key_out ( );
        cout << right->id << cout;
        if ( right->right ) right->right->key_out ( );
        }
}
```

单层展开提供了最大的性价比，然而如果需要的话可以再次展开。以下是 key_out 方法的 4 层迭代版本：

```
inline
void binary_tree::UNROLLED3_key_out ( )
{
    if ( left ) left ->key_out ( );
    cout << id << endl;
    if ( right ) right ->key_out ( );
}

inline
void binary_tree::UNROLLED2_key_out ( )
{
    if ( left ) left -> UNROLLED3_key_out ( );
    cout << id << endl;
    if ( right ) right -> UNROLLED3_key_out ( );
}

inline
void binary_tree::UNROLLED1_key_out ( )
{
    if ( left ) left -> UNROLLED2_key_out ( );
    cout << id << endl;
    if ( right ) right -> UNROLLED2_key_out ( );
}

void binary_tree::key_out ( )
{
    if ( left ) left -> UNROLLED1_key_out ( );
    cout << id << endl;
    if ( right ) right -> UNROLLED1_key_out ( );
}
```

展开到这种程度可以显著地提高性能，然而本质上相同的一个方法的4个版本使得对展开递归的维护很容易产生错误。使用旧式的C #define宏扩展可以把展开方法统一起来，但是这样显得很不雅观，并且在维护时要特别小心。以下是key_out方法的宏版本。

```
# define KEY_OUT_MARCO ( inline_arg, my_label, call_label )        \
                                                                   \
inline_arg                                                         \
```

```
void binary_tree::UNROLLED##my_label##_key_out()                    \
{                                                                   \
    if(left) left->UNROLLED##call_label##_key_out();                \
    cout<<id<<endl;                                                 \
    if(right) right->UNROLLED##call_label##_key_out();              \
}

KEY_OUT_MARCO(inline,3,0)
KEY_OUT_MARCO(inline,2,3)
KEY_OUT_MARCO(inline,1,2)
KEY_OUT_MARCO(\\t    ,C,1)

inline
void x::key_out()
{
    UNROLLED_key_out();
}
```

请记住:仍然需要在 binary_tree 的头文件中放一个 key_out()方法的入口。

这种宏机制能够在不需要太多额外努力的情况下进行附加层次的展开。但是必须记住:尽管这段代码在源程序中所占用的空间现在看起来比较少,但是在 C++ 预编译器把宏展开以及编译器完成内联之后,将会产生和以前一样的代码膨胀。在大力推荐宏展开这种脆弱的机制时,我们是很痛苦的,但是在需要这种极端措施时,它的优点(只有递归方法的一个版本)通常要胜过它的弱点。

4 层展开相对于两层展开所得到的额外性能提高只是略微低于第一层展开所获得的性能。8 层展开将另外获得 4 层展开所得性能的 2~3 倍。然而应该记住:过多的展开层次会导致过度的代码膨胀。4 份复制的版本是原版本的 4 倍大。8 份复制的版本是原版本大小的 64 倍,这是一种明显的膨胀。但是在某些情况下,为了获得性能优势,即使如此也是值得的。

10.4 对静态局部变量进行内联

对基于编译的内联解决方案来说,局部静态变量可能十分成问题。某些编译器拒绝内联任何包含静态变量声明的方法。有些编译器允许内联所有静态变量,但是会在运行时错误地创建这些内联变量的多个实例。毫无疑问,甚至可能有一些编译器提供全世界最差的两种做法:它们不内联带有静态变量的方法,但是它们却在每个相互分离的编译模块中创建该方法的独立实例。当然,某种程度上这种困难是 C++ 处理内联方法的方式

发生改变的结果。当前的语言规范要求对内联方法进行外部链接，因而将强制对任何局部静态变量只产生一个实例。不幸的是，并不是所有的编译器都能满足该规范的要求。请检查一下您的编译器。还没有升级的编译器会存在以下问题。

与内联包含静态变量的方法有关的问题是确定静态变量的惟一性和确保静态变量得到初始化。内联的方法在头文件中定义。内联这样的方法需要所有使用该头文件的模块共享静态变量的惟一实例。从逻辑上说，静态局部变量实际上就是限定范围内的全局变量。这意味着链接器要足够智能化以检测出对这种静态变量的使用，创建该变量，在全局数据空间内保留空间，为该变量创建初始化代码(要不就是确保它得到初始化)，最后把所有对该变量的内联引用链接到在全局空间内为该变量新创建的惟一实例。尽管从概念上说这不是太困难，但是在相互分开的编译模块内确定共享静态变量的位置把写编译器的人给弄糊涂了。当然这是一个短期问题，但是其间您需要知道与使用这些封装在方法内的静态变量有关的困难。

确定您的编译器是否能够正确处理静态变量的内联是比较简单的。下面的程序正好能够完成此事。

```
z.h:
inline
int test ( )
{
    static int i  = 0;
    return + + i;
}

y.cpp:

# include "z.h"
void test_a ( )
{
    int i  = test ( );
    cout << i;
}

x.cpp:

 # include "z.h"
void test_b ( )
```

```
{
    int i = test ( );
    cout << i;
}

main.cpp:

int main ( ) {
    test_a ( );
    test_b ( );
    cout << endl;
    return 0;
}
```

如果编译器创建了正确的代码,那么分别编译的三个 .cpp 文件及随后的链接将产生一个输出为 12 的可执行程序。如果输出为 11,则说明编译器在内联包含静态变量的方法方面存在问题。同时需注意编译器指明 test 方法根本没有内联的警告信息。

如果由于性能而要求内联包含静态变量的方法,而您的编译器又不能正确处理这种构造,那么可以采用一种简单的做法。通过创建静态类属性(静态成员变量)可以达到相同的最终结果。这样会在限定的范围内创建变量的惟一实例。不幸的是,这种机制明显地扩大了该静态变量的可见性,但是如果有适当的记录,这通常不会成为问题。

10.5 与体系结构有关的注意事项:多寄存器集

并不是所有的体系结构都有同样的调用/返回性能。有些机器(比如 SPARC 体系结构)在限定的调用深度范围内,会表现出接近于零的调用/返回性能损失。这些体系结构能够为调用进行很快的新寄存器帧分配,从而消除了在进行调用时保存旧寄存器内容的需要。通过简单地把以前分配的寄存器集变成当前寄存器集,方法就可以返回,因此不再需要在返回时对寄存器进行恢复。由于这些体系结构可以在寄存器中传递参数(调用者和被调用者共享寄存器组),因此不再需要在调用前把变量压入系统的内存堆栈。

每次调用都分配新寄存器集的能力显著地减少了编译器中寄存器分配方案方面的压力。充足的寄存器使得频繁地向寄存器分配用到的原子变量变得容易起来。无论是在代码尺寸方面,还是在执行效率方面,这些寄存器都可以用最低的代价进行访问。

随着寄存器存储和获取不再成为问题以及调用间变量映射成本的降低,现在只有基于管道的调用效果和无法跨越调用边界进行优化这两方面还对内联的明显好处寄予厚

望。当由于因没有进行内联而改善的缓存特性发挥作用后，随着寄存器分配器压力的减小，内联在那些非微小方法上的性能效力变得令人怀疑。

不幸的是，这些多寄存器集体系结构的效力会随着调用深度的增加而明显减弱。如果程序的调用深度超过了处理器物理寄存器集的数量，那么这些体系结构就要开始经受极大的调用代价，通常会比典型的单寄存器集体系结构付出更大的代价。其原因在于寄存器集的溢出要求把整个寄存器集保存到内存中，最后再从内存中恢复。这说明内联的直接效果是可以忽略的，而内联在调用深度上的间接影响会产生显著的性能收益。

这使希望使用内联的人们陷入了进退两难的境地：一般情况下，内联非微小方法几乎不是同等有效。即使是看起来能明显从内联受益的情况，在这些多寄存器集体系结构中也会使内联失去效果。另一方面，内联确实减少了总的调用深度，因而也减少了寄存器集发生溢出的可能性。不存在很好的权威性答案。一般情况下，在这些机器上内联不会损害性能，同时也不能大幅提高性能。我们的建议是：如果没有运行时测试资料直接支持内联决策，就应该避免内联非微小方法。

10.6 要　　点

- 内联可以提高性能。目标是要找到程序的快速路径并对其进行内联，尽管内联该路径可能并不容易。
- 条件内联会阻止内联。这减少了编译时间，简化了早期开发阶段的调试工作。
- 选择性内联是只在某些位置对方法进行内联的一项技术。选择性内联只在性能重要的路径中内联方法调用，从而可以在某种程度上抵消由于内联方法而带来的代码尺寸膨胀方面的潜在可能性。
- 递归内联是一种让人别扭但对提高递归方法性能有效的技术。
- 要注意局部静态变量。
- 内联针对的是消除调用开销，在使用内联之前请先确认您机器上的真正调用代价。

第 11 章

标准模板库

提高 C++ 性能的编程技术

标准模板库(Standard Template Library,STL)是一种容器与通用算法之间强有力的合并。如果从性能的角度来思考,头脑中立即会出现几个问题:

- STL 与不同容器和算法在渐近复杂度方面的性能保证捆绑在一起出现。这究竟是怎么回事?
- STL 由许多容器构成。面对一个给定的计算任务,到底应该使用哪一个?在特定情况下还有更好的吗?
- STL 的性能怎么样?如果使用自己开发的容器和算法,是否可以做得更好?

本章我们将就这些问题以及相关问题进行探讨。

即使我们通常会提到"STL 性能",但是应该说明一下,STL 包含满足不同性能水平的各种不同实现。本章的衡量标准是一个普通实现的反映。不过,我们的观测应该适用于一般实现。

11.1 渐近复杂度

渐近复杂度是对算法性能的粗略估计。它把算法集映射到特定的性能标准集中。如果要计算包含 N 个整数的向量中所有元素的和,那么必须对每个整数检查一次,并且仅仅是一次,因此其复杂度是 N 的序数,我们把它称为 $O(N)$。另一方面,假设要创建一个包含 N 个元素的向量,由于某种原因,您选择把这些元素插入到向量的前端。在向量的前端每插入一个元素都需要依次把全部已有元素移动一个位置。这导致共计 $(1+2+3+\cdots+N)$ 次向量元素的移动,结果等于 $(N/2)(N+1)$ 次移动。即使是需要 $(N*N/2)+(N/2)$ 次移动,我们仍然说这种算法的复杂度是 $O(N*N)$。这是因为渐近性能标准集忽略常量倍数和次要因素。因而,$N*N$ 和 $7N*N$ 的复杂度是一样的:都是 $O(N*N)$。因此我们说渐近复杂度是一种简单的粗略估计。尽管它对于数学家研究算法复杂度来说很不错,但是对于程序员来说这是不够的。我们还是要关注常量倍数。对于我们来说,

2N 的算法和 4N 的算法是不一样的。前者比后者要快 100%。

STL 对其算法渐近复杂度的保证是一种良好的开始。它告诉我们所用的算法是所有这类算法中最好的[MS96]。然而，我们仍然要深入研究一下被数学家忽略的常量倍数和次要因素。我们不会涉及所有容器的所有操作，相反，只讨论流行容器的一些常见操作，下面就从插入开始。

11.2 插 入

如果性能十分重要的路径中包含了向容器插入大量元素的代码，您会使用什么样的容器呢？为了深入探讨这个问题，我们将对一些容器进行测试并对其结果进行讨论。这次插入练习将向数组、向量、列表和多重集插入 100 万个随机元素。每个插入测试带有 3 个参数：

- 指向正在测试的目标容器的指针
- 指向 data 数组的指针，该数组包含要插入到容器中的元素
- data 数组的大小

下面给出用于各种容器的插入测试：

```
template <class T>
void arrayInsert ( T *a, T *collection, int size )
{
    for ( int k = 0; k < size; k++ ) {
        a [k] = collection [k];
        }
}

template <class T>
void vectorInsert ( vector<T> *v, T *collection, int size )
{
    for ( int k = 0; k < size; k++ ) {
        v->push_back ( collection [k]);
        }
}

template <class T>
void listInsert ( list<T> *l, T *collection, int size )
{
```

```
    for ( int k = 0; k < size; k++ ) {
        l->push_back ( collection[k]);
        }
}

template <class T>
void multisetInsert ( multiset<T> * s, T * collection, int size )
{
    for ( int k = 0; k < size; k++ ) {
        s->insert ( collection[k]);
        }
}
```

各种插入测试均使用由随机元素组成的数组(collection)作为输入参数。然后把这些随机元素插入到测试的容器中。对于整型数来说,我们使用 STL 的函数 generator ( ) 来生成随机数:

```
int * genIntData ( int size )
{
    int * data = new int[size];

    // generate random integers and place them in array data
    generate (&data[0], &data[size], rand );
        return data;
}
```

我们使用这些测试来向每个容器插入 100 万个随机整数。执行速度如图 11.1 所示,单位是毫秒(ms)。

在这种情况下,数组要胜出其他容器很多。与其他容器相比,数组有一个明显的优点:从一开始起,它就足够大,可以包含全部 100 万个 data 的元素。相反,向量事先不知道集合会有多大。列表和多重集也不知道。更为糟糕的是,列表和多重集还要为其他开销而斗争。对于每一个 data 值,列表除了要前后设置指向列表元素的指针外,还要分配一个列表元素来容纳数据。多重集在任何时候都要按排序后的顺序来维护它的集合。由于这个原因,多重集在这次特殊的测试中最后完成。必须指出的是,我们进行的这次特殊的测试并不说明数组总是比其他容器好。每种容器都有胜过其他容器的特定场合。如果某种容器在任何情况下都好于其他容器,那么其他容器就不会存在了。例如,如果需要一个集合,它的查找速度是最重要的,那么多重集容器将胜过数组、向量和列表容器。

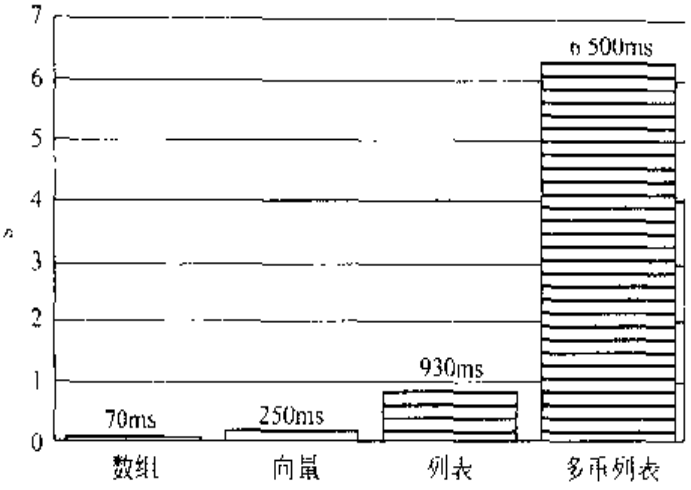


图 11.1 插入速度

数组和向量都是使用连续内存块的顺序容器。在事先不知道集合大小的情况下向量容器会更有用。向量会动态增长，这样程序员就可以不考虑集合大小了。要更好地理解向量和数组之间的性能差别，必须首先把向量大小和向量容量区分开来。向量的大小是向量当前包含的元素个数。向量的容量是向量在分配更多的内存以适应其增长之前所能够容纳元素的最大数量。在向向量中插入第一个元素的时候，典型的 STL 实现会分配一大块内存从而使向量的容量超过它的初始大小(值为 1)。随后的插入将增大向量的大小，而容量将保持不变。如果集合连续增长，那么最后向量的大小将达到它的容量。下一次插入将促使向量实现扩大其容量。这必须经过下列步骤[Lip91]：

- 分配一块更大的内存块，以便为其他元素提供空间。
- 把现有元素复制到新分配的内存块中。这时会为旧集合中的每个元素调用复制构造函数。
- 清除旧的集合并释放它所占用的内存。这时会为旧集合中的每个元素调用析构函数。

这些步骤可能导致昂贵的开销。出于这个原因，我们希望出现向量大小超过其容量的次数最少。分配和释放内存已经够糟糕的了，而为旧集合中的每个元素调用构造函数和析构函数的代价可能更昂贵。当向量元素涉及大的构造函数和析构函数方法时就是这种情况。例如，我们把整数集合替换成 BigInt 对象集合来看一下。

BigInt 类是从 Tom Cargil 的《C++ Programming Style》[Car92]书中借用过来的。BigInt 类把正整数表示成二进制编码的十进制数。例如，数 123 在内部由 3 字节的字符数组来表示，每个字节代表一位数。可以使用下列方式创建和操作 BigInt 对象：

```
BigInt a = 123;
```

```
BigInt b = "456";
BigInt c = a + b;
```

以下是 BigInt 定义的一部分：

```
class BigInt {
public:
    BigInt ( const char * );
    BigInt ( unsigned = 0 );
    BigInt ( const BigInt& );
    ~BigInt ( );
    ...

private:
    char     * digits;
    unsigned ndigits;
    unsigned size;                          // size of allocated string
    ...
};
```

在我们的讨论过程中只使用 BigInt 实现的一部分。这一部分包括构造函数和析构函数，代码如下：

```
BigInt ::BigInt (unsigned u)                // Constructor
{
    unsigned v = u;
        for ( ndigits = 1; (v /= 10) > 0; ++ndigits) {
            ;                               // Count the number
            }                               // of digits in u
        digits = new char [size = ndigits];

    for ( unsigned i  = 0; i < ndigits; ++i ) {
        digits[i] = u % 10;                 // Peel off the
        u /= 10;                            // digits of u
        }
    }

BigInt ::BigInt ( const BigInt& copyFrom )  // Copy constructor
{
    size = ndigits = copyFrom.ndigits;
```

```
    digits = new char[size];

    for ( unsigned i = 0; i < ndigits; ++i ) {
        digits[i] = copyFrom.digits[i];
        }
}
    BigInt ::~ BigInt ( )                    // Destructor
{
    delete [ ] digits;
}
```

为了测试向量容量增长对性能的影响，我们重复了向量的插入测试，不过这次把整数元素替换成了 BigInt 对象。图 11.2 对比了 100 万个整数和 100 万个 BigInt 对象的插入时间。

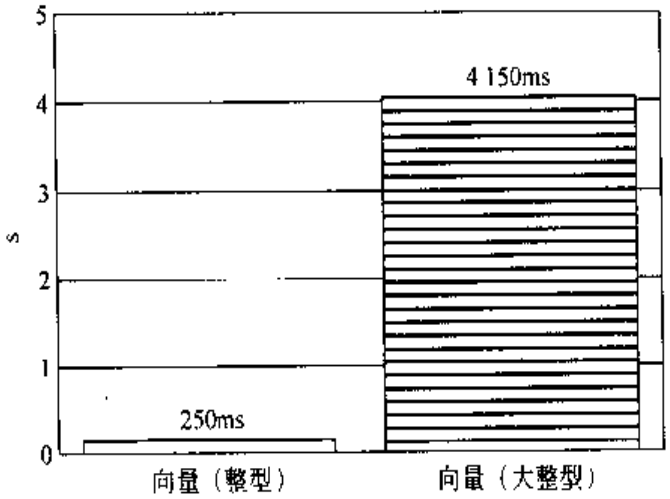


图 11.2 对象插入速度

性能上的惊人差别反映了简单整数和 BigInt 对象在复制和清除代价上的差别。如果您发现自己处于这样的情况：对象的复制构造函数和析构函数相当昂贵，而向量的容量很可能增长，那么您仍然可以通过保存指针而不是对象来避免这种代价。指向对象的指针不需要相应的构造函数和析构函数。复制指针的代价实质上与复制整数相同。在向量中插入 100 万个 BigInt 指针的时间与 100 万个整数几乎相等，图 11.3 给出了它们各自的时间。

列表不在连续的内存块中放置元素，因而不必面对容量问题以及相关的性能开销问题。结果是，在 BigInt 插入测试中列表容器比向量容器表现得更好一些（如图 11.4 所示）。

图 11.4 引出另一个重点：每种容器都有自己的长处和弱点。尽管向量在插入 100 万

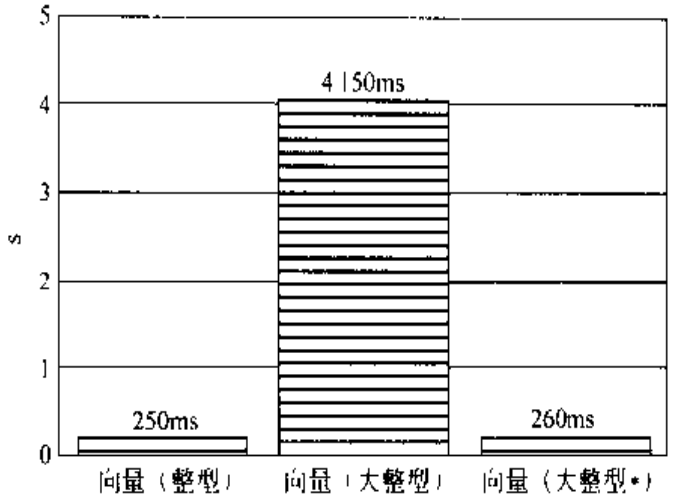


图 11.3 对象插入与指针插入的比较

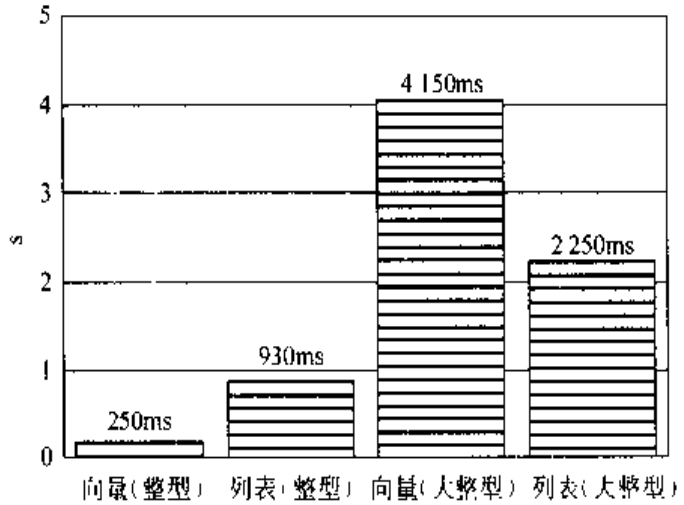


图 11.4 列表插入与向量插入的比较

个整数方面胜过列表，但是列表在使用 BigInt 对象的类似测试中却超过了向量。

对于由向量容量增长而产生的性能难题，还有另外一个可选的解决方案。许多情况下，可以估计向量的容量，让该容量在特定的情况下足够大。在进行这种智能化猜测的情况下，可以继续进行并提前保存这个必要的容量。向量的 reserve(n)方法保证向量的容量等于或大于 n 个元素。这需要修改一行我们的测试代码：

```
vector<BigInt> * v = new vector<BigInt>;
v->reserve (size );
vecotrInsert ( v, dataBigInt, size );
```

保留需要的容量使性能提高了 1/2 以上。图 11.5 显示了在保留容量和不保留容量两种情况下执行 100 万个 BigInt 插入的时间。

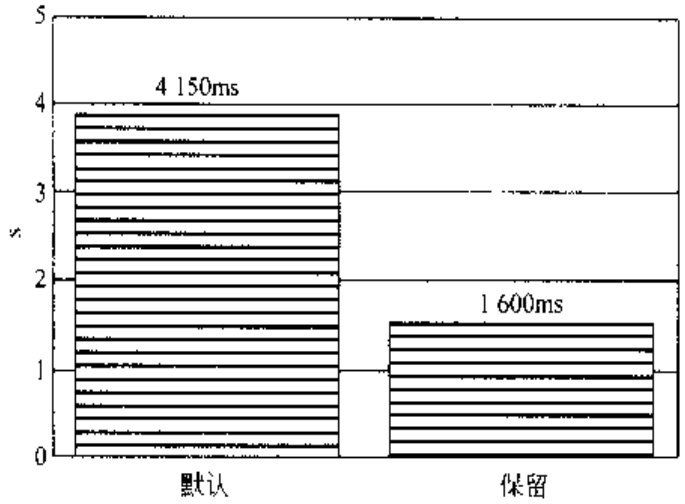


图 11.5 在保留容量和不保留容量两种情况下的向量插入

到现在为止，我们已经向数组、向量和列表的尾部插入了元素。在事先知道集合大小的情况下数组容器完成得最好。在不知道集合大小的情况下，对于整数插入来说，向量要胜过列表容器。如果我们需要在集合的中间或是前端插入元素，那么哪一种容器会最好呢？这可能要把以前的性能结果颠倒过来了。下面，针对把元素插入容器的前端这个任务来比较向量和列表的性能。我们使用了如下两个函数：

```
template <class T>
void vectorInsertFront ( vector<T> * v, T * collection, int size )
{
    for ( int k = 0; k < size; k + + ) {
        v - >insert ( v - >begin( ), collection[k] );
        }
}

template <class T>
void listInsertFront ( list<T> * l, T * collection, int size )
{
    for ( int k = 0; k < size; k + + ) {
        l - >push_front (collection[k] );
        }
}
```

向量容器在从前端插入方面的性能极为糟糕。其性能太差，所以我们不得不把集合的大小限制为 10 000。每在向量的前端插入一个元素都要求移动所有的现有元素，以便为新元素提供空间。换句话说，把集合变成原来的 10 倍大将导致执行时间变成原来的 100 倍。由于没有那么多的时间，所以我们把测试限制为 10 000 个元素。相反，列表容器在前端插入元素时消耗的时间不变，不管集合是多大。图 11.6 显示了执行时间上的这种惊人差别。

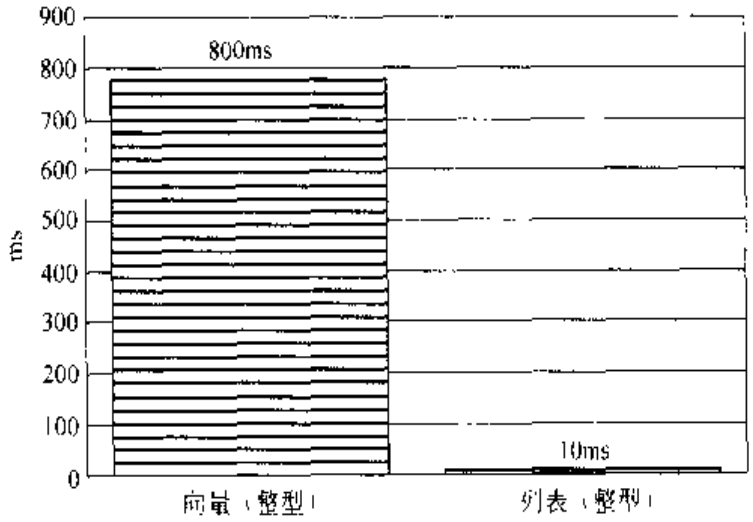


图 11.6 从前端插入

最后的测试突出了 STL 的另一个性能优点：STL 设计阻止您去做一些实在不明智的事，比如说需要在前端插入元素时选择向量容器。因此，STL 不为向量容器提供 push_front ( )方法。如果一定要坚持做这种非常低效的事情，将不得不亲自去实施，就像我们在 vectorInsertFront ( )代码中所做的那样。

11.3 删 除

关于删除操作的性能在很多方面都与插入操作相似。许多与插入效率有关的结论同样适用于删除。* 例如：

- 向量在进行尾部插入(删除)时很优秀。由于这种操作与集合的大小无关，所以它是一种时间固定的操作。
- 除了在尾部操作之外，任何情况下对于元素的插入(删除)来说向量都是一种极坏

* 可以在[MS96]中找到那些结论以及更多的性能保证。包括对所有容器和算法的复杂性保证。

的选择。这种插入(删除)的代价与插入(删除)点到向量的最后一个元素之间的距离成比例。

- 双向队列在向集合的前端和尾部插入(删除)时是高效的(时间固定)。对于其他任何地方的插入(删除)它都是低效的。
- 无论向(从)集合的什么地方插入(删除),列表都是高效的(时间固定)。

作为一种全面的探讨,我们对从向量和列表中删除 100 万个元素进行了测试。对于列表和向量都使用 STL 提供的 pop_back ( )方法。用 pop_back ( )方法来删除容器中的最后一个元素。

```
template <class T>
void vectorDelete ( vector<T> * v )
{
    while (! v->empty ( ) ) {
        v->pop_back ( );
        }
}

template <class T>
void listDelete ( list<T> * l )
{
    while (! l->empty ( ) ) {
        l->pop_back ( );
        }
}
```

执行速度如图 11.7 所示。

正如插入一样,在涉及从集合的尾部删除元素时,向量要胜过列表。如果从前端删除会怎样呢? 如果一切都如插入那样,那么应该把向量和列表之间的比较结果颠倒过来。测试代码如下:

```
template <class T>
void listDeleteFront ( list<T> * l )
{
    while (! l->empty ( ) ) {
        l->pop_front ( );
        }
}
```

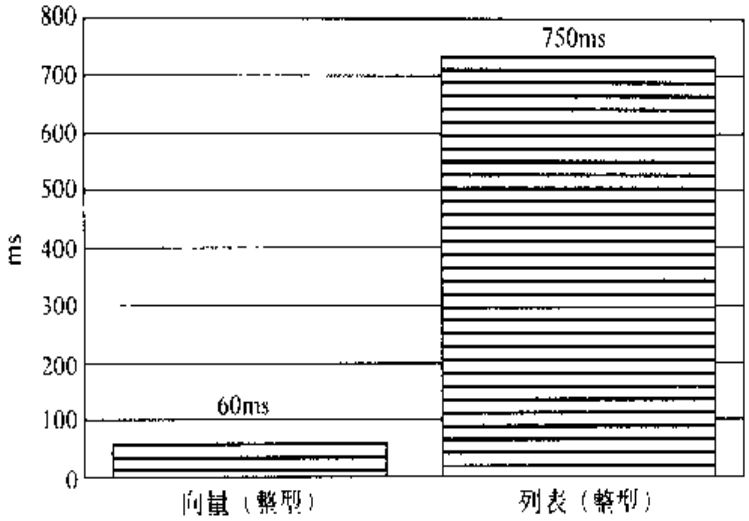


图 11.7 列表删除与向量删除的比较

```
template <class T>
void vectorDeleteFront ( vector<T> * v )
{
    while ( ! v->empty ( ) ) {
        v->erase ( v->begin ( ) );
        }
}
```

正如前端插入一样，在向量中从前端删除元素是低效的。由于这个原因，我们不得不把测试限制成 10 000 个元素。图 11.8 显示了执行时间。

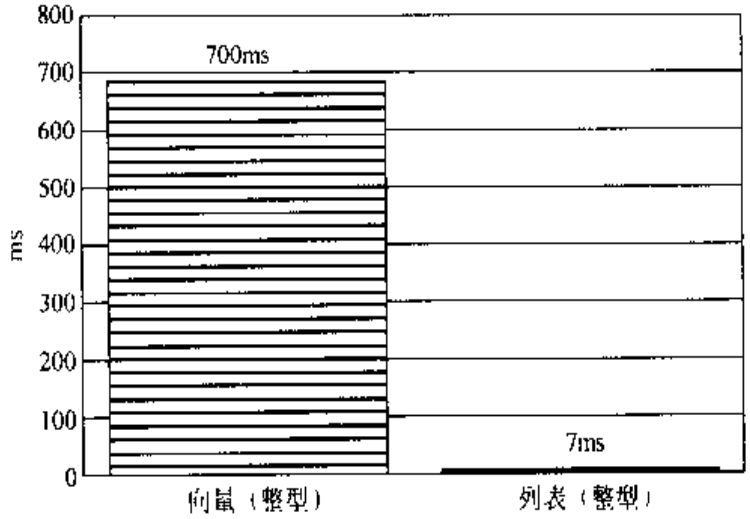


图 11.8 从前端删除元素

在这次测试中，列表胜出向量 100 倍。请注意，STL 同样没有为向量容器提供 pop_front ( )方法，这是为了阻止用户对向量执行这种低效的操作。

图 11.7 和图 11.8 显示的删除性能情况类似于图 11.1 和图 11.6 所揭示的插入性能情况。这再次证明了我们在本节开始时所得出的结论：与容器插入有关的结论同样适用于删除。

11.4 遍 历

我们经常需要以每次一个元素的方式遍历容器，在这个过程中对每个元素进行一种操作。我们以 STL 的 accumulate ( )函数来说明容器的遍历。accumulate ( )方法从头到尾遍历容器，同时把每个元素加起来。进行测试的容器是数组、向量和列表。每个容器都保存一个相同的集合，该集合由 10 000 个随机整数组成。以下是用于各种容器的测试代码：

```
void vectorTraverse ( vector<int> * v, int size )
{
    int sum = accmulate ( v->begin ( ), v->end ( ), 0);
}

void arryTraverse ( int *a, int size )
{
    int sum = accmulate ( &a[0], &a[size], 0);
}

void listTraverse ( list<int> *l, int size )
{
    int sum = accmulate ( l->begin ( ), l->end ( ), 0);
}
```

测试由对这些遍历函数的每一个调用 100 次组成。对每种容器遍历 100 次的执行时间如图 11.9 所示。

对于容器遍历，向量和数组的性能是相同的。它们都胜出列表容器很多。可见容器遍历的决定性因素是容器内存布局与系统缓存之间的交互。向量和数组都在连续的内存空间中放置它们的元素。从逻辑上说，集合中相邻的元素在内存中物理上是一个挨一个。当一个特定的元素要加载到缓存中时，会连带一些下一步要访问的相邻元素(确切的数量依赖于缓存行的大小和元素的大小)。从缓存的角度来看这是一种理想的操作。对于列

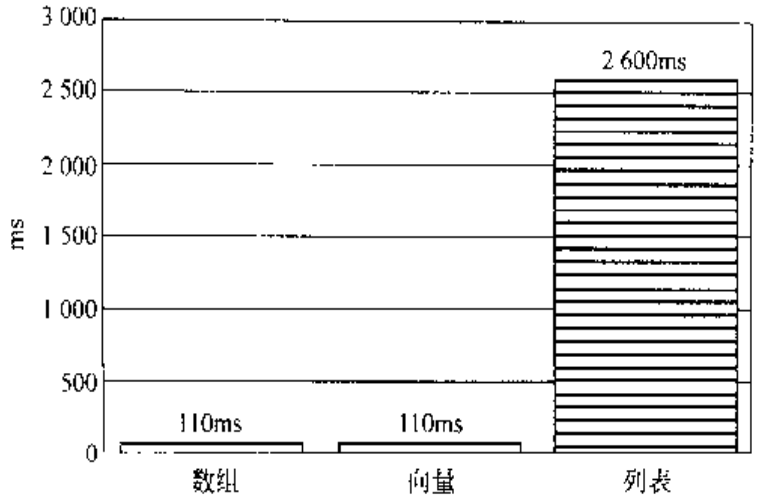


图 11.9 容器遍历速度

表容器则不是这种情况。逻辑上相邻的元素在内存中不一定相邻。此外，列表元素除了要保存元素值外，还要保存前后指针，这使得列表元素要比相应的向量元素大。即使是某些相邻的列表元素在内存中储存在相邻的位置，出于其大小的原因，也只有较少的元素会装入缓存行。因此，遍历列表容器所产生的缓存失败要远远超过遍历数组和向量。

11.5 查 找

以上对插入、删除和遍历的讨论说明了一种情况，即数组、向量和列表中的每一种都有可能胜过其他用于测试的容器。我们现在转移到另一个更为重要的操作上，在该操作中多重集容器会大展身手。这种操作就是在集合中查找特定元素。下面的代码使用 STL 的 find ( )在各种容器中完成查找操作：

```
void arrayFind ( int *a, int *collection, int size )
{
    int const value = collection[size/2];
    int *p = find (&a[0], &a[size],value);
}

void vectorFind ( vector<int> *v, int *collection, int size )
{
    int const value = collection[size/2];
    vector<int> ::iterator it =
        find ( v->begin ( ), v->end ( ), value );
```

```
}

void listFind ( list<int> * l, int * collection, int size )
{
    int const value = collection[size/2];
    list<int> ::iterator it =
        find ( l->begin ( ), l->end ( ), value );
}

// This is using the generic find ( ) which is not the
// best choice when searching a multiset.
void multisetFind ( multiset<int> * s, int * collection, int size )
{
    int const value = collection[size/2];
    multiset<int> ::iterator it =
        find ( s->begin ( ), s->end ( ), value );
}
```

性能测试包括对容器查找的100次迭代。图11.10显示了各种容器的执行时间。

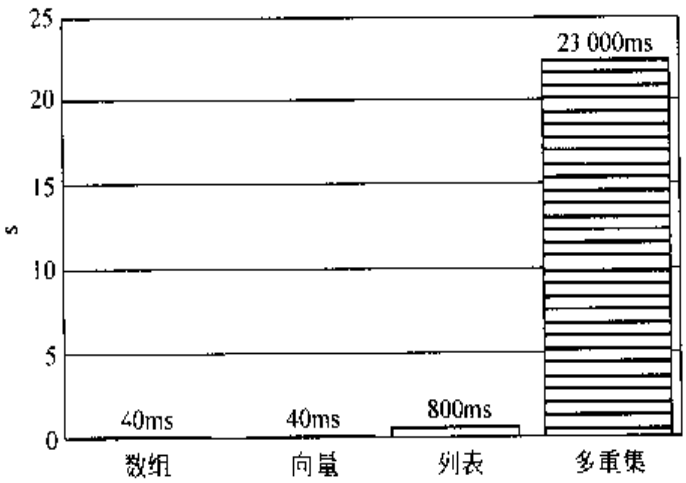


图11.10 容器查找速度

图11.10在感觉上是一种误导，它似乎说明对于查找性能来说，向量和数组是最佳选择。之所以看起来是这样，仅仅是因为我们对多重集容器错误地使用了通用的find ( )算法。如果使用多重集容器所提供的find ( )成员方法，那么会有效得多。成员find ( )方法利用了多重集是一种排序容器的事实。由于通用find ( )方法必须处理非排序的容器，所以它对此事实不加理睬。因此，通用的find ( )方法必须进行顺序查找。多重集查找的

以下版本使用了 find ( )成员方法：

```
void multisetFind ( multiset<int> * s, int * collection, int size )
{
    int const value = collection[size/2];
    multiset<int> ::iterator it =
        s->find ( value );
}
```

成员 find ( )方法胜过通用 find ( )方法不止一个数量级。通用 find ( )方法为了完成对包含 100 万个元素的多重集进行 100 次查找，共消耗了 23 000ms，而成员 find ( )在 0.06ms 内就完成了同样的测试。从图 11.11 中可以得到一种直观感受。

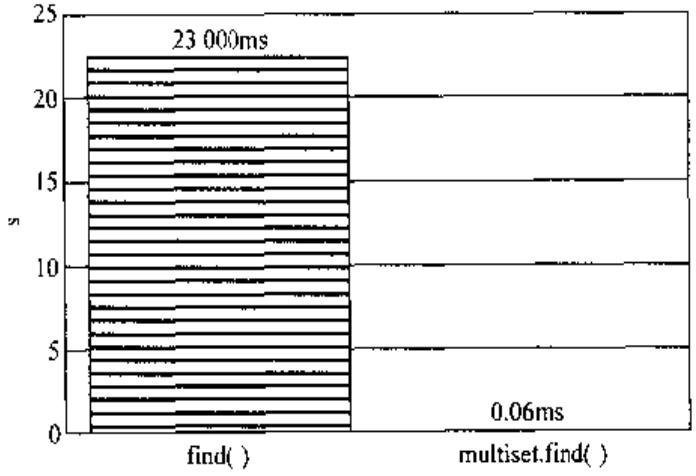


图 11.11 通用 find ( )和成员 find ( )的比较

很明显，通过使用成员 find ( )，多重集容器出现在元素查找上时，使其他竞争容器烟消云散。多重集是排序的集合这一事实需要以插入和删除操作的性能损失为基础。另一方面，有序的顺序为多重集容器在查找方面提供了很大的优势。

11.6 函数对象

默认情况下，accumulate ( )函数对集合中的所有元素使用 operator+，并返回所有元素相加的累计结果。对于整数集合来说，如果给 accumulate ( )提供的初始值是 0，那么结果会是集合中所有元素的和。accumulate ( )算法不一定局限于对象相加，而可以对集合元素进行任何操作(假定元素支持该操作)并返回累积结果[MS96]：

```
template <class InputIterator, class T>
```

```
T accumulate ( InputIterator first,
               InputIterator beyondLast,
               T initialValue )
{
    while ( first ! = beyondLast ) {
        initialValue = initialValue +  * first + + ;
        }
}
```

在 C 编程中，参数化这种操作一般通过传递函数指针实现。STL 可以做同样的事情，但是除此之外，它还以函数对象的方式为我们提供了一种更为有效的替代方式。

我们要计算整数集合的积，而不是计算和。可选择定义一个函数，该函数返回两个整数的积，然后把这个积作为参数传递给 accumulate ( ) [MS96]：

```
int mult ( int x, int y ) {return x * y ;}
...
int a[10] = {1, 2, 3, 5, 7, 11, 13, 17,19, 23};
int product = accumulate (&a[0],
                          &a[10],
                          1,
                          mult);   // Function pointer
```

第二种做法是把函数对象作为参数传递给 accumulate ( )。函数对象重载了 operator ( )的类的实例。例如：

```
class Mult {
public:
    int operator ( ) ( int x, int y ) const {return x * y; }
}
...
int a[10] = {1,2,3,5,7,11,13,17,19,23};
int product = accumulate (&a[0],
                          &a[10],
                          1,
                          mult);   // Constructor for a
                                   // function object
```

由于 STL 已经提供了一个 times 模板类，所以我们实在不必重新发明一个用于乘法的轮盘。

下面的代码是等价的[MS96]：

```
int product = accumulate (&a[0],
                          &a[10],
                          1,
                          times<int>( ) );  // Constructor for a
                                            // function object
```

我们对分别使用函数指针和函数对象的 accumulate ( )的性能关系进行了比较。图 11.12 给出的执行时间表示的是对刚才讨论的 accumulate ( )版本进行 100 万次调用的时间。

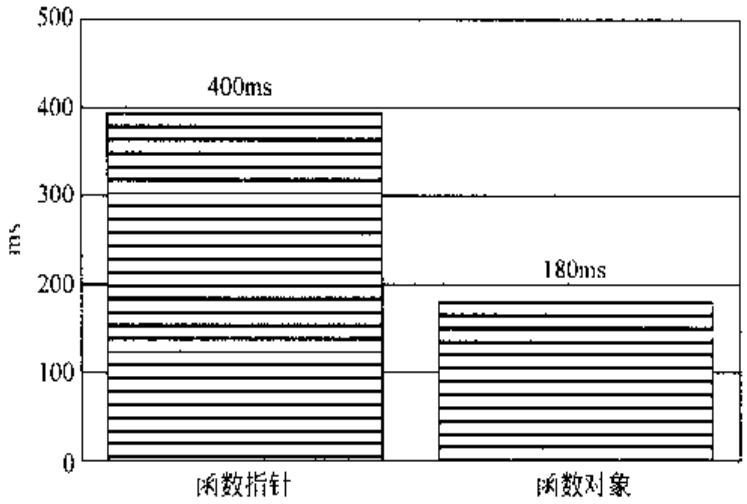


图 11.12 函数对象与函数指针的比较

使用函数对象的版本要明显胜过使用函数指针的版本.这与预期的结果一致。函数指针直到运行时才能确定,这使它们无法内联。而函数对象是在编译时确定的,这使得编译器可以自由地内联对 operator ( )的调用并显著地提高效率。

11.7 比 STL 更好

很多人认为 STL 有出色的性能。通常的倾向甚至是让程序员想都不要想通过自己的实现来超越 STL 的性能。尽管在很大程度上我们同意这种说法,但我们还是想再深入探讨一下这些观点。为了学到更多有关的知识,我们将研究一下如何超越 STL 的性能。

在本章前面遇到了 accumulate ( )函数,我们用它计算整数集合的和。让我们看一下能否用自己的实现对它加以改进:

```
int myIntegerSum ( int *a, int size )
{
```

```
    int sum = 0;
    int *begin = &a[0];
    int *end = &a[size];

    for ( int *p = begin; p != end; p++ ) {
        sum += *p;
        }
        return sum;
}
```

基于 STL 的实现是：

```
int stlIntegerSum ( int *a, int size )
{
return accumulate (&a[0], &a[size], 0 );
}
```

性能比较进行测试的结果说明 myIntegerSum ( )和 stlIntegerSum ( )的执行速度相同。我们对超越 STL 速度的尝试出现了问题。可以合理地推测这种情况是典型的，并且一般情况下超越 STL 的性能是一种规则的例外。STL 实现在以下方面加入了自己的特点：

- STL 实现使用最佳算法。
- STL 实现的设计者通常是相应领域的专家。
- 各领域的专家致力于提供灵活、强大和高效的库。这是他们的首要任务。

对于我们这些其余的人，开发可重用的容器和算法顶多只算第二目标。我们的首要任务是交付紧扣主题的应用程序。大多数情况下，我们没有时间和专门技术去和那些专家相比。

既然我们已尽力劝阻您不要试图去和 STL 竞争，那我们就改变一下方式并提供一些可能的例外，例如，假设我们的应用程序经常需要反转一个字符序列，并假设字符序列长度固定，比如是 5 个字符长。下面是基于 STL 的一种可能实现：

```
char *s = "abcde";
reverse (&s[0],&s[5]);
```

另一种选择是开发我们自己的实现，如下所示：

```
char *s = "abcde";
char temp;
```

```
temp = s[4];          // s[0] <-> s[4]
s[4] = s[0];
s[0] = temp;

temp = s[3];          // s[1] <-> s[3]
s[3] = s[1];
s[1] = temp;
```

我们循环执行该字符串反转代码100万次并记录了执行时间，如图11.13所示。

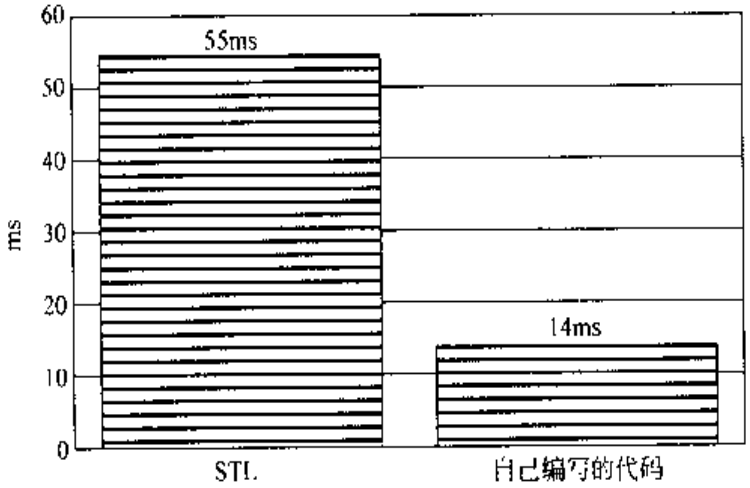


图11.13 STL与自编代码的速度比较

我们的穷举法实现胜过了STL实现，性能代价仅为STL实现的1/4。reverse()实现忽略了序列长度为5的事实。它必须处理序列长度因而也更为强大。我们的穷举方式似乎很不优雅，它十分依赖于序列是5个字符长的事实，因而不能处理任何其他序列长度。这是针对于特定任务的非常专一的实现。

虽然这个例子有点做作，但是它确实得出了至少三条与STL实现有关的论点：

- 为了使自己实现的版本胜过STL版本，需要付出很大努力。
- 超越STL是有可能的。
- 如果您打算超越基于STL的实现，那么最好知道一些相应领域里STL不清楚的东西。否则，最好还是不要浪费时间。

我们可以得出一个与超越STL性能有关的更为鲜明的结论。即无论一种实现多么出色，总会存在一些显得不太理想的计算环境。该实现不可能在任何时间对任何人都是最好的。我们以列表容器为例。其中，列表容器支持插入、删除和size()成员函数，该成员函数返回当前存储在列表中的元素数量。在实现size()成员方法时我们基本上有两

种设计选择[BM97]：

(1) 我们可以定义一个数据成员字段，让它连续跟踪当前的列表大小。每次对于列表的插入和删除操作，都要立即对该成员进行重新求值和更新。这种必要的更新使得插入和删除会变慢，但是 size ( )函数会非常快(时间固定)。

(2) 只是在需要的时候才计算列表的大小。由于不需要更新当前的大小值，所以插入和删除将会更快一些，但是 size ( )函数会慢得多。size ( )计算将需要用一次顺序遍历来计算列表中当前元素的数量。

STL 实现必须在这两种设计中进行选择。无论作出什么样的选择，都存在损害性能的情况。插入和删除起决定作用的工作将从设计选项(1)受益。另一方面，对大型列表进行频繁的 size ( )操作的工作将从选项(2)受益。由于这个原因，对 STL 的性能进行改进的可能性在理论上总是存在的。

11.8 要 点

- STL 是抽象、灵活和效率的非同寻常的综合。
- 对于某种特定的使用方式，一些容器可能比另外一些更为高效，这要根据应用程序而定。
- 除非清楚一些特定领域内 STL 不清楚的事情，否则您一般无法超越 STL 实现。
- 不过，在某些特定情况下超越 STL 实现的性能是可能的。

第 12 章

引用计数

提高 C++ 性能的编程技术

就像在自然界一样，玻耳兹曼熵法则同样适用于软件世界——所有的实体都将随着时间的过去而消失。软件工程可以从设计清楚和实现简单的小范围模型开始。那些选择小范围模型(这些小范围的模型使这些软件进入市场)的软件将经常经历扩展的过程。这通常是为了响应客户不断提出的对新功能的需求和原有不足的改进。把开发新功能和解决错误合并在一起会使原来清晰的设计遭受严重破坏。随着时间的过去，设计和实现的清晰性会消失在维护和频繁的发布周期中。软件必然会朝着混乱的方向发展。区分好与不太好的惟一标准就是衰变率。

与混乱的软件有关的主要难题是内存讹误。分配的内存通过指针传递的方式遍布于系统内部。指针在模块和线程之间传递。在混乱的软件系统中，会导致两个主要的难题：

- 内存泄漏。当内存一直得不到释放时就会发生这种情况。随着时间的过去，当应用程序的内存消耗无法控制时，内存泄漏将会使应用程序终止运行。
- 过早删除。当指针的所有关系不清楚时，会导致访问已释放的内存，这将造成灾难性的破坏。

幸运的是，C++ 对这两种问题都提供了解决方案。C++ 允许您控制对象的创建、清除、复制和赋值等方面。通过开发一种称作引用计数的垃圾回收机制，可以实现这种控制。其基本思想是把对象清除的责任从客户端代码转移给对象本身。对象连续跟踪当前对它的引用数量，并在引用数量达到 0 时删除自己。换句话说，对象在没有人再使用时删除自己。对于追溯到内存讹误时期的大多数软件缺陷来说，引用计数是 C++ 武器库中非常重要的技术。

引用计数同样被吹捧为一种性能优化。人们声称引用计数能够减少内存讹误和 CPU 时钟周期。我们考虑这样一个对象：它包含一个成员，该成员指向堆存储空间。在复制该对象或者向该对象赋值时会发生什么呢？最简单的实现会执行一种深层复制，给每个对象一份对堆存储空间的私有拷贝。以如下 MyString 的实现为例：

```
class MyString {
public:
    ...
    MyString& operator = ( const MyString& rhs );
    ...
private:
    char * pdata;
};

MyString& MyString::operator = ( const MyString& rhs )
{
    if ( this = = &rhs ) return * this;
    delete [ ] pdata;
    int length = strlen ( rhs.pdata ) + 1;   // Include the terminating
                                             // null
    pdata = new char [length];               // Make room for the new array
    memcpy ( pdata, rhs.pdata, length);      // Copy the characters from the
                                             // right - hand - side object

    return * this;
}
```

当把一个 MyString 对象赋值给另一个 MyString 对象时，像这样：

```
MyString p,s;
p = s = "TWO";
```

将最终得到如图 12.1 所示的结果。

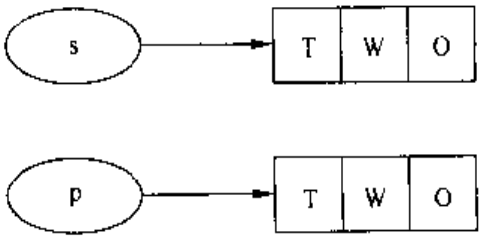


图 12.1 重复资源

无论是在内存利用方面，还是在分配堆内存和复制字符串内容所消耗的 CPU 时钟周期方面，这似乎都是一种浪费。如果利用引用计数来让两个对象都指向相同内存资源，那么代价会低廉得多。

毫无疑问，引用计数在减少内存使用方面总是一个胜利者。然而，它在执行速度方面是否总是一个胜利者却是不明确的。这正是本章的主题：什么情况下引用计数会对执行速度有利，在什么环境下它又确实会损害执行速度。

对引用计数性能的全面讨论需要对其实现细节的初步掌握。下面讨论这个话题。

12.1 实现细节

实现引用计数是一项精确而庞大的任务，想清楚地解释它也是一样。与重新发明轮盘相反，这次我们把性能讨论建立在由 Meyers 在《More Effective C ++ 》[Mey96]的第 29 条中所开发的引用计数实现的基础之上，这是因为以下几个原因：

- 就该实现所要完成的任务来说，它是一种高效的实现。
- [Mey96]对该实现解释得非常清楚。除了我们提供的资料外，您可以在那里找到更多细节。
- 许多人都已经熟悉了这个特定的实现。

让我们从 Widget 类开始并把它发展成一个使用引用计数的类。Widget 包含一个指向堆内存的成员(Meyers 在他开始的概念说明中使用了一个 String 类)：

```
class Widget {
public:
    Widget ( int size );
    Widget ( const Widget& rhs );
    ~Widget ( );

    Widget& operator = ( const Widget& rhs );

    void doThis ( );
    int showThat ( ) const;
private:
    char * somePtr;
};
```

向引用计数发展的第一步是在 Widget 中添加一个计数器成员。该计数器将连续跟踪对特定 Widget 对象的引用数量：

```
class Widget {
    ... // As before
private:
```

```
    char * somePtr;
    int refCount;
};
```

.当 refCount 达到 0 时 Widget 对象将清除自己。下一步向 Widget 类引入引用计数：

```
class RCWidget { // Reference - counted Widget class
public:
    RCWidget ( int size ) : value ( new Widget ( size ) ) { }

    void doThis ( ) { value - >doThis ( ) ; }
    int showThat ( ) const { return value - >showThat ( ); }
private:
    Widget * value;
};
```

RCWidget 作为 Widget 的代理。它展示出同样的接口并且简单地把实际工作转给它指向的 Widget 对象。这是一种基本思想，如图 12.2 所示。

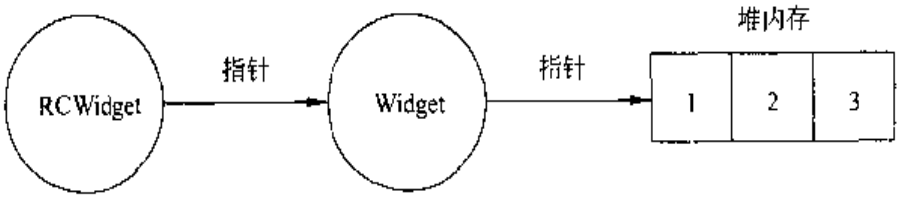


图 12.2 使用引用计数的 Widget 类的简单设计

到现在为止，我们的讨论还过于简单化。Meyers 的实现要复杂得多。他在 RCWidget 中加入了一个智能化的指针[Mey96]，而不是我们所展示的这个没有智能的指针。他还让 Widget 类以公有的方式从 RCObject 类派生，这个类是所有使用引用计数类的基类，RCObject 封装了对引用计数变量的操作。Meyers 的实现看起来如图 12.3。

这是引用计数设计的上层概念。现在继续讨论实现的细节。首先，我们把 Widget 换成一个更容易联系的更为具体的例子。BigInt 是我们从 Tom Cargil 的《C ++ Programming Style》[Car92]中借用的一个类。BigInt 类使用二进制编码的十进制数来表示正整数。例如，数字 123 在内部由一个 3 字节的字符数组表示，每个字节代表一位数字。可以按如下方式创建和操作一个 BigInt 对象：

```
BigInt a = 123;
BigInt b = "456";
BigInt c = a + b;
```

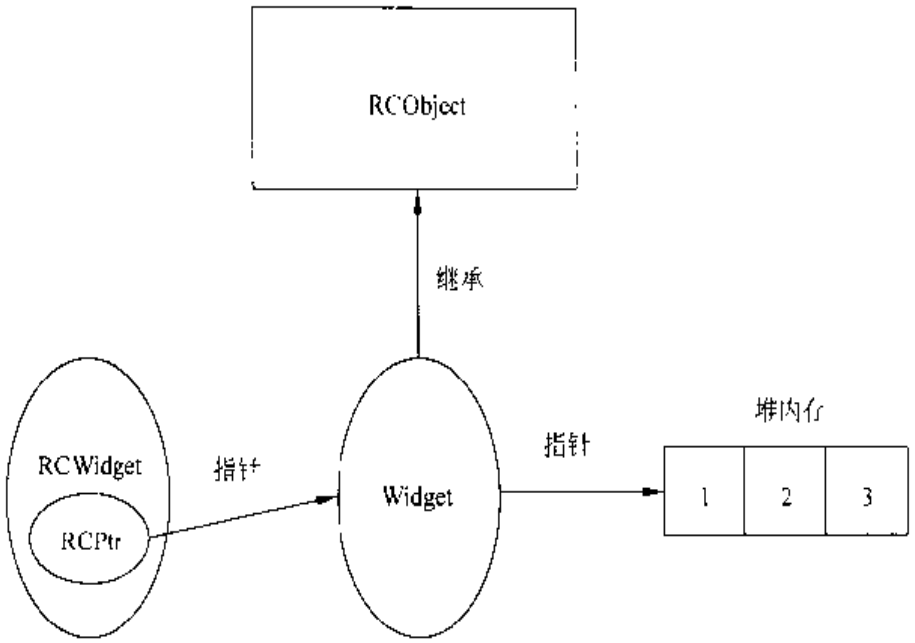


图 12.3 在引用计数设计中添加继承和智能指针

由于 BigInt 使用堆内存来存储它的数字，所以对于引用计数讨论来说它是个不错的选择。以下是 BigInt 的实现，它几乎和原实现相同[Car92]：

```
class BigInt {
friend BigInt operator + ( const BigInt&, const BigInt& );
public:
    BigInt ( const char * );
    BigInt ( unsigned = 0 );
    BigInt ( const BigInt& );
    BigInt& operator = ( const BigInt& );
    BigInt& operator + = ( const BigInt& );
    ~BigInt ( );

    char * getDigits ( ) const { return digits; }
    unsigned getNdigits ( ) const { return ndigits; }
private:
    char * digits;
    unsigned ndigits;
    unsigned size;                          // size of allocated string
    BigInt ( const BigInt&, const BigInt& );  // operational ctor
    Char fetch ( unsigned i ) const;
};
```

在讨论引用计数的过程中，我们将只使用 BigInt 实现的一部分。这部分包括一个构造函数、一个析构函数和一个赋值操作符。这足以让您掌握这三个函数了。

```
BigInt::BigInt (unsigned u)
{
    unsigned v = u;

    for ( ndigits = 1; (v/ = 10) > 0; + + ndigits)
        ;
        }

    digits = new char [size = ndigits];

    for ( unsigned i  = 0; i < ndigits; + + i ) {
      digits[i] = u % 10;
        u / = 10;
        }
}

BigInt::~ BigInt ( )
{
    delete [ ] digits;
}

BigInt& BigInt::operator = ( const BigInt& rhs )
{
    if ( this = = &rhs ) return * this;
    ndigits = rhs.ndigits;
    if ( ndigits > size ) {
        delete [ ] digits;
        digits = new char[ size = ndigits ];
        }

    for ( unsigned i  = 0; i < ndigits; + + i ) {
        digits [i] = rhs.digits [i];
        }
```

```
    return * this;
}
```

BigInt 实现的其他部分对我们的讨论没有什么作用。但出于对完整性的考虑，我们还是把它们列了出来：

```
BigInt::BigInt ( const char * s)
{
    if ( s [0] = = '\0' ) {
        s = "0";
        }

    size = ndigits = strlen (s);
    digits = new char[size];
    for ( unsigned i = 0; i < ndigits; + + i ) {
        digits[i] = s[ndigits - 1 - i] = 0';
        }
}

BigInt::BigInt ( const BigInt& copyFrom )     // Copy constructor
{
    size = ndigits = copyFrom.ndigits;
    digits = new char[size];

    for ( unsigned i = 0; i < ndigits; + + i ) {
        digits[i] = copyFrom.digits[i];
        }
}

// Operational constructor: BigInt = left + right
BigInt::BigInt ( const BigInt& left, const BigInt& right )
{
    size = 1 + ( left.ndigits > right.ndigits ?
        left.ndigits : right.ndigits );
    digits = new char[size];
    ndigits = left.ndigits;
    for ( unsigned i = 0; i < ndigits; + + i ) {
```

```
        digits[i] = left.digits[i];
        }

    *this += right;
}

inline
char BigInt::fetch( unsigned i ) const
{
    return i < ndigits ? digits[i] : 0;
}

BigInt& BigInt::operator+=( const BigInt& rhs )
{
    unsigned max = 1 + ( rhs.ndigits > ndigits ?
        rhs.ndigits : ndigits );

    if ( size < max ) {
        char *d = new char[size = max];
        for ( unsigned i = 0; i < ndigits; ++i ) {
            d[i] = digits[i];
            }
        delete [] digits;
        digits = d;
        }

    while ( ndigits < max ) {
        digits[ndigits++] = 0;
        }

    for ( unsigned i = 0; i < ndigits; ++i) {
        digits[i] += rhs.fetch(i);
        if ( digits[i] >= 10 ) {
            digits[i] -= 10;
            digits[i+1] += 1;
            }
```

```
        }

    if (digits[ndigits-1] == 0) {
        --ndigits;
    }

    return *this;
}

ostream& operator << ( ostream& os, BigInt& bi )
{
    char c;

    const char *d = bi.getDigits();

    for ( int i = bi.getNdigits() - 1; i >= 0; i-- ) {
        c = d[i] + '0';
        os << c;
    }
    os << endl;

    return os;
}

inline
BigInt operator + ( const BigInt& left, const BigInt& right )
{
    return BigInt ( left, right );
}
```

为了满足引用计数的要求，我们需要使每个 BigInt 对象都有一个相关联的引用计数。可以通过直接向 BigInt 类加入 refCount 成员或者让 BigInt 从一个基类继承这两种方式来实现这一点。Meyers 选择了后者，我们按照他的做法进行。RCObject 类是用于引用计数的基类，它封装了引用计数变量及其有关操作。

```
class RCObject {
public:
```

```
    void addReference ( ) { ++refCount; }
    void removeReference ( ) { if (--refCount == 0) delete this; }

    void markUnshareable ( ) { sharcable = false; }
    bool isShareable ( ) const { return shareable; }

    bool isShared ( ) const { return refCount > 1 ; }

protected:
    RCObject ( ) : refCount (0), shareable (true) { }
    RCObject ( const RCObject& rhs ) : refCount (0), shareable (true) { }
    RCObject& operator= ( const RCObject& rhs ) {return * this;}
    virtual ~ RCObject ( ) { }

private:
    int refCount;
    bool shareable;
};
```

现在必须把 BigInt 类改成从 RCObject 类继承：

```
class BigInt : public RCObject {
    // Same as before
}
```

我们正在向 RCBigInt 进展，这是 BigInt 类的引用计数实现。不知为什么，它需要指向真正的 BigInt 对象。可以使用一个真正的(非智能)指针或者是智能指针来实现这一点。Meyers 选择使用智能指针。(智能指针是通过重载“->”和“*”操作符而封装了一个非智能指针的对象[Mey96]。)这种特定的智能指针负责引用计数簿记：

```
template <class T>
class RCPtr {
public:
    RCPtr ( T *realPtr = 0 ) : pointee ( realPtr ) { init ( ); }
    RCPtr ( const RCPtr& rhs ) : pointee ( rhs.pointee ) { init ( ); }
    ~ RCPtr ( ) { if (pointee ) pointee->removeReference ( ); }

    RCPtr& operator= ( const RCPtr& rhs );
```

```
    T* operator->() const { return pointee; }
    T& operator*() const { return *pointee; }

private:
    T *pointee;
    Void init();
};

template <class T>
void RCPtr<T>::init()
{
    if( 0 == pointee ) return;

    if( false == pointee->isShareable() )
        pointee = new T( *pointee );
        }

    pointee->addReference();
}

template <class T>
RCPtr<T>& RCPtr<T>::operator=( const RCPtr& rhs )
{
    if (pointee != rhs.pointee) {
        if( pointee ) pointee->removeReference();
        pointee = rhs.pointee;
        init();
        }

    return *this;
}
```

最后，我们有了装配引用计数 BitInt 所需要的全部代码段，我们把这个引用计数 BigInt 类称为 RCBigInt。RCBigInt 非常简单。艰苦的工作已经由 RCPtr 和 RCObject 完成：

```
class RCBigInt {
friend RCBigInt operator+ ( const RCBigInt&, const RCBigInt& );
public:
```

```
    RCBigInt (const char * p) : value (new BigInt (p)) { }
    RCBigInt ( unsigned u = 0 ) : value ( new BigInt (u) ) { }
    RCBigInt ( const BigInt& bi ) : value ( new BigInt (bi) ) { }

    void print ( ) const { value->print ( ); }

private:
    RCPtr<BigInt> value;
};

inline
RCBigInt operator + ( const RCBigInt& left, const RCBigInt& right )
{
    return RCBigInt ( * ( left.value ) + * ( right.value ) );
}
```

在性能测试中，对于那些取决于频繁地赋值和复制 RCBigInt 对象的工作，RCBigInt 会大显身手。另一方面，与使用简单的 BigInt 相比，产生对新 BigInt 第一次引用的新 RCBigInt 对象变得更为昂贵。一旦 RCBigInt 对象引发了第一次 BigInt 引用之后，它就会在堆中创建一个 BigInt 对象并指向它。这是一种代价，如果使用基于堆栈的简单 BigInt 对象，则不需要付出这种代价。类似的结论适用于删除最后的 BigInt 引用。基础的 BigInt 对象被释放回堆中。为了量化性能得失，我们开发了赋值和创建测试案例，并分别把它们用于 BigInt 和 RCBigInt。第一次测试中测量了 BigInt 对象的创建和清除：

```
void testBigIntCreate ( int n )
{
    ...
    GetSystemTime ( &t1 );

    for ( i = 0; i < n; ++i ) {
        BigInt a = i;
        BigInt b = i + 1;
        BigInt c = i + 2;
        BigInt d = i + 3;
        }

    GetSystemTime ( &t2 );
    ...
```

```
}
```

用于 RCBigInt 创建的等价测试几乎和 BigInt 测试相同。我们只是简单地把 BigInt 对象换成 RCBigInt 对象。重要的一点是要指出下面的测试将会更多地忙于创建第一次引用然后再删除它们：

```
void testRCBigIntCreate ( int n )
{
    ...
    GetSystemTime ( &t1 );

    for ( i = 0; i < n; ++i ) {
        RCBigInt a = i;
        RCBigInt b = i + 1;
        RCBigInt c = i + 2;
        RCBigInt d = i + 3;
        }

    GetSystemTime ( &t2 );
    ...
}
```

类似地，我们还有两个相应的赋值测试案例。我们只写出用于 BigInt 的第一个。用于 RCBigInt 的那一个几乎完全相同：

```
void testBigIntAssign ( int n )
{
    ...
    BigInt a,b,c;
    BigInt d = 1;

    GetSystemTime ( &t1 );

    for ( i = 0; i < n; ++i ) {
        a = b = c = d;
        }

    GetSystemTime ( &t2 );
    ...
}
```

4个测试案例中的每一个都包含一个循环，我们让每个循环执行100万次，并对执行时间进行了测量。图12.4和图12.5给出了执行时间(以ms为单位)。

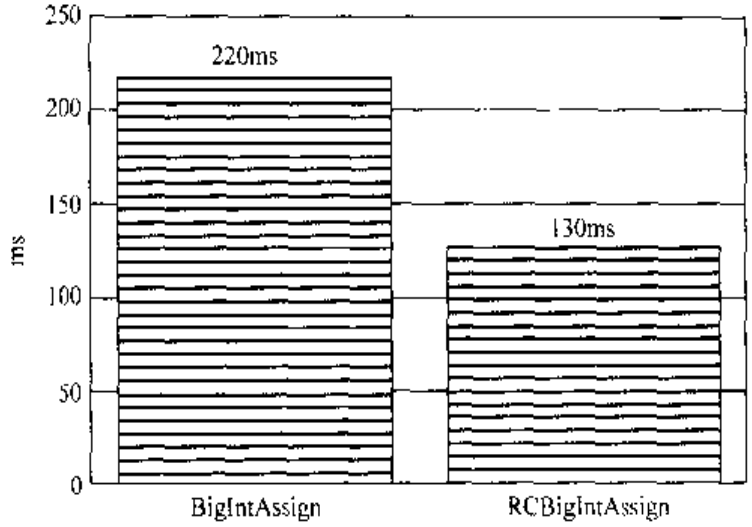


图12.4 BigInt的赋值速度

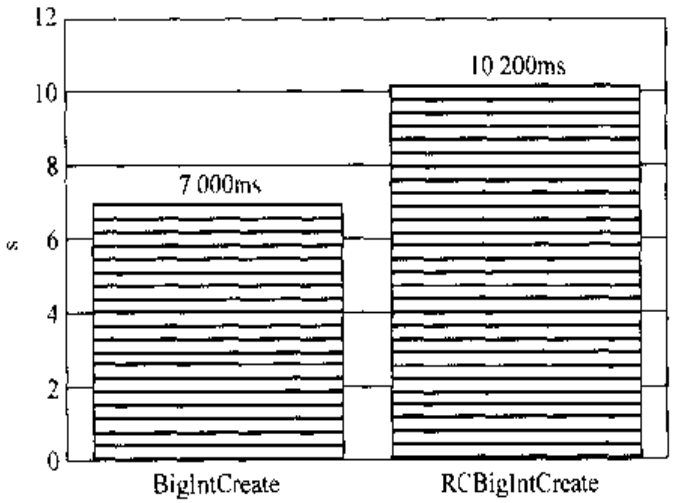


图12.5 BigInt的创建速度

和意料中的结果一样，对RCBigInt对象的赋值是非常高效的，几乎是BigInt对象赋值速度的两倍。而对象创建的结果则相反。创建和删除BigInt对象明显要快于RCBigInt对象。这应当是一种提醒，就是说引用计数并不意味着性能上的获益。这是我们第一次遇到引用计数损害性能的情况。另一方面，同样存在引用计数促进性能的情况(赋值测试)。本质问题是如何把它们区分出来。引用计数和执行速度的关系是与上下文

相关的。它依赖于几方面的因素：

- 目标对象的资源消耗主要在哪一方面？例如，如果目标对象占用大量的内存，那么在保持内存方面的失败将推动对可用内存的限制，并且将以缓存失败和页面错误的方式导致巨大的性能损失。
- 分配(释放)目标对象使用的资源所带来的开销如何？这与前一个因素不同。诸如BitInt之类的对象通常占用少量的存储空间，看起来不会耗尽可用内存。然而，即使是从堆里分配一个字节的空间，也将需要几百条指令的代价。释放一个字节的空间也是一样。
- 可能有多少对象共享目标对象的单个实例？通过使用赋值操作符和复制构造函数可以增加共享。
- 创建(清除)目标对象的第一次(最后一次)引用的频率怎样？如果使用构造函数而不是复制构造函数来创建新的引用计数对象，那么就会创建对该对象的第一次引用。这种操作是昂贵的。它包含除创建目标对象之外的更多开销。同样的结论适合于删除最后的引用。

在极端的情况下可能存在许多引用计数实例共享少量目标对象(通过复制构造函数和赋值操作符)的情况，而创建和清除第一次和最后的引用是有限的。目标对象是有限资源的巨大消费者，这些有限资源本身的获得和释放是昂贵的。对于促进引用计数性能来说这是一种典型情况。在另一种极端情况下，可能存在这样的情况：创建了许多目标对象，但是它们的引用计数很少超过一个或两个(说明少至没有共享)。目标对象的资源消耗可以忽略不计，资源本身的获得和释放是廉价的。在这种情况下，引用计数是执行时间方面的失败者。

难以判断的原因在于大多数情况将属于这两种极端的情况。即使是在我们的BigInt示例中，也可以通过提供内存池来分配由BigInt内部缓冲区消耗的存储空间，从而使天平进一步朝着与引用计数相反的方向倾斜。在这种情况下，复制BigInt对象的代价明显降低。从性能的观点来看，这使引用计数更加缺乏吸引力。下一节我们将介绍另一种不同的实现，这种实现会影响引用计数和简单对象之间的性能平衡。

12.2 已存在类

前述引用计数BigInt类实现包括一些对原BigInt类的修改。当然，这要求我们有修改源代码的特权。这种选择并不总是可行，因为目标类可能来自一个没有给出源代码的库。但仍然可以实现这种已存在的、无法插手的目标类的引用计数，不过我们需要一些设计上的修改。

以前，在能够修改源代码的情况下，我们通过从 RCObject 继承 BigInt，从而在 BigInt 中添加了引用计数器。现在，我们无法修改 BigInt 的实现，因此必须引入一个独立的类来包含引用计数并对它进行操作。Meyers 把这个类叫作 CountHolder。图 12.6 形象地给出了修改后的设计。

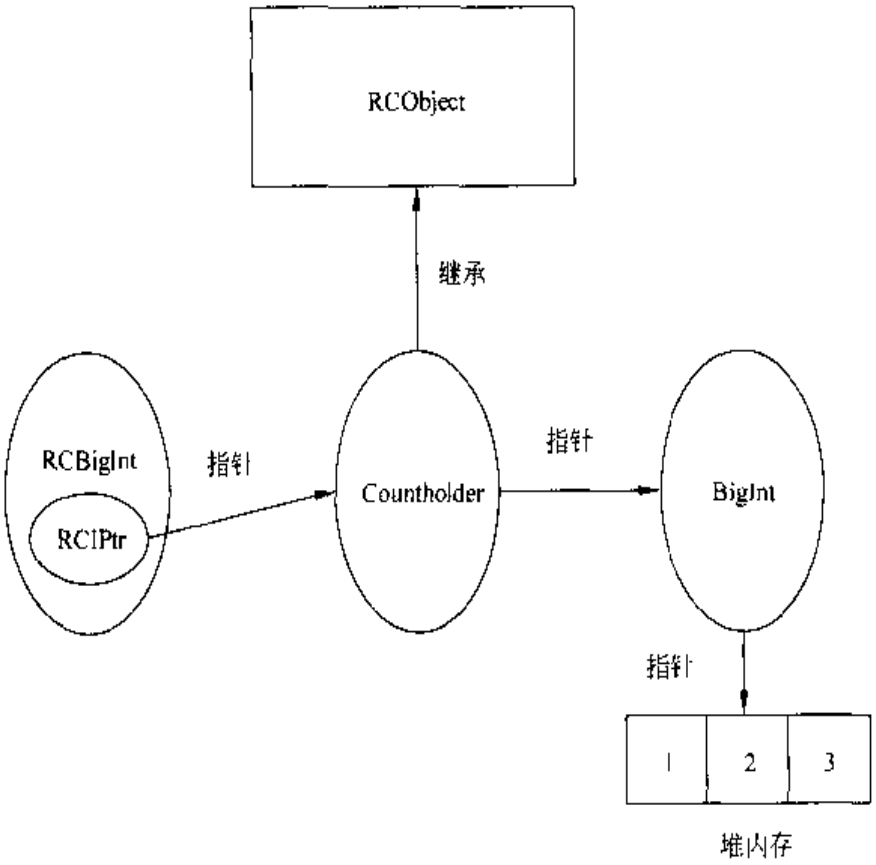


图 12.6 对已存在的 BigInt 进行引用计数

除了从 BigInt 类中分离出引用计数之外，我们还用 PCIPtr 对象替换了 RCPtr 对象（以前是嵌入在 RCBigInt 类中的）。PCIPtr 是一个与 RCPtr 一样的智能指针，但是它是通过一个指向 CounterHoler 的指针来间接指向 BigInt。RCPtr 以前是直接指向 BigInt。从实现的观点来看我们有 3 个问题要解决。首先，BigInt 类恢复到它原来的形式，而不再从 RCObject 继承。第二，RCBigInt 类现在包含一个 PCIPtr 成员，而不是 RCPtr：

```
class RCBigInt {
    ...                          // The same as before
private:
    PCIPtr<BigInt> value;   // PCIPtr not RCPTr
};
```

第三，我们必须提供 PCIPtr 的实现。PCIPtr 的实现还以嵌套类的形式引入了 Counter-Holer：

```
template <class T>
class RCIPtr {
public:
    RCIPtr ( T *realPtr = 0 );
    RCIPtr ( const RCIPtr& rhs );
    ~ RCIPtr ( );

    RCIPtr& operator = ( const RCIPtr& rhs );

    T* operator -> ( ) const { return counter->pointee; }
    T& operator * ( ) const { return * ( counter->pointee); }

private:
    struct CountHolder : public RCObject {
        ~CountHolder ( ) { delete pointee; }
        T *pointee;
    };

    RCIPtr<T>::CountHolder *counter;
    void init ( );
};

template <class T>
void RCIPtr<T>::init ( )
{
    if ( 0 == counter ) return;

    if ( false == counter->isShareable ( ) ) {
        counter = new CountHolder;
        counter->pointee = new T ( *counter->pointee );
        }

    counter->addReference ( );
}
```

```
template <class T>
RCIPtr<T>::RCIPtr ( T *realPtr )
    : counter ( new CountHolder )
{
    counter->pointee = realPtr;
    init ( );
}

template <class T>
RCIPtr<T>::RCIPtr (const RCIPtr& rhs ) : counter ( rhs.counter )
{
    init ( );
}

template <class T>
RCIPtr<T>::~RCIPtr ( )
{
    if ( counter ) counter->removeReference ( );
}

template <class T>
RCIPtr<T>& RCIPtr<T>::operator = ( const RCIPtr& rhs )
{
    if ( counter != rhs.counter ) {
        if ( counter ) counter->removeReference ( );

        counter = rhs.counter;
        init ( );
        }
    return *this;
}
```

RCIPtr<T>::operator=的实现在代价上类似于相应的 RCPtr 的实现。因此，赋值测试(testRCBigInt)的执行速度对于 RCBigInt 的两种版本来说是相同的。对于创建测试(testRCBigIntCreate)则不然。这次测试首先创建对新 BigInt 对象的引用，然后再以相反的顺序删除它们。从 RCPtr 到 RCIPtr 的转变使第一次 BigInt 引用的创建及删除变

得要昂贵得多。我们不仅要分配新的基于堆的 BigInt 对象，还必须分配新的 CountHolder 对象，此外在清除时必须删除它们。BigInt 和 RCBigInt 之间的性能差距在这种情况下变得更大。图 12.7 给出了这次测试得到的新的执行速度。

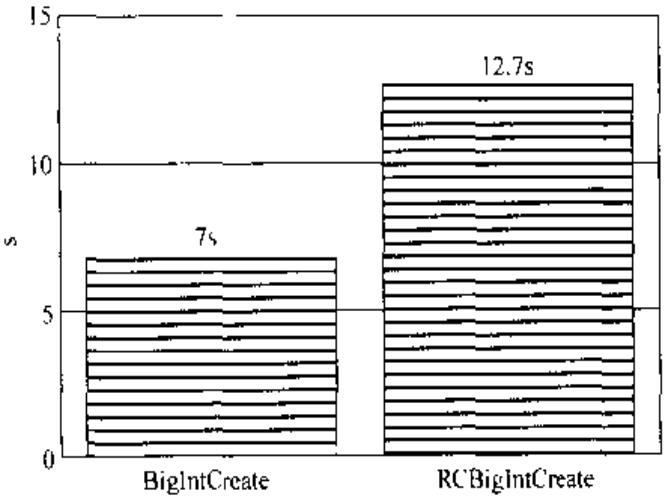


图 12.7 BigInt 的创建速度

12.3 并发引用计数

我们已经研究了引用计数的两种版本，并已知道其中一种(使用 RCPtr)要快于另一种(使用 RCIPtr)。在引用计数方面还有其他重要的曲解需要我们进行评估。第三种不同的实现在多线程执行环境中处理引用计数。在这种情况下，可能有多个线程并发地访问引用计数对象。因此，必须保护包含引用计数的变量以便使更新成为原子操作。原子更新要求锁定机制。我们在第 7 章中讨论了锁定类，在此把有关的代码复制过来：

```
class MutexLock {
public:
    MutexLock ( ) {mutex = CreateMutex ( NULL, FALSE, NULL ); }
    virtual ~ MutexLock ( ) {CloseHandle ( mutex ); }
    void lock ( ) {WaitForSingleObject ( mutex, INFINITE ); }
    void unlock ( ) {ReleaseMutex ( mutex ); }
private:
    HANDLE mutex;
};
```

给 RCBigInt 类添加并发控制需要在它的类声明中改动一行代码。我们给 RCIPtr 智

能指针添加 MutexLock 模板参数：

```
class RCBigInt {
    … // Same as before
private:
    RCIPtr<BigInt, MutexLock> value;        // Add MutexLock
};
```

处理并发访问的任务就交给了 RCIPtr 类。我们对它的模板声明进行了扩充以便允许另外一个锁类参数。该锁用于串行化对需要原子更新的变量所进行的访问：

```
template <class T, class LOCK>
class RCIPtr {
public:
    …
private:
    struct CountHolder : public RCObject {
        ~CountHolder ( ) { delete pointee; }
        T * pointee;
        LOCK key;
    };

    RCIPtr<T, LOCK>::CountHolder * counter;
    void init ( );
};
```

init ( )方法不受并发的影响因而没有修改。如果必要，原子访问将在调用者的 init ( ) 中处理：

```
template <class T, class L>
void RCIPtr<T, L>::init ( )
{
    if ( 0 == counter ) return;
        if ( false == counter->isShareable ( ) ) {
        counter = new CountHolder;
        counter->pointee = new T ( *counter->pointee );
        }
        counter->addReference ( );
}
```

其余的方法会显式地操作该锁。它的实现是简单的——所有需要串行化的操作都用一对 lock ( )和 unlock ( )调用括起来：

```
template <class T, class L>
RCIPtr<T, L>::RCIPtr ( T * realPtr )
: counter ( new CountHolder )
{
    counter->pointee = realPtr;
    init ( );
}

template <class T, class L>
RCIPtr<T, L>::RCIPtr ( const RCIPtr& rhs )
: counter ( rhs.counter )
{
    if ( rhs.counter ) rhs.counter->key.lock ( );
    init ( );
    if (rhs.counter ) rhs.counter->key.unlock ( );
}

template <class T, class L>
RCIPtr<T, L>::~RCIPtr ( )
{
    if ( counter ) {
        counter->key.lock ( );
        counter->removeReference ( );
        counter->key.unlock ( );
        }
}

template <class T, class L>
RCIPtr<T, L>& RCIPtr<T, L >::operator = ( const RCIPtr& rhs )
{
    if ( counter != rhs.counter ) {
        if ( counter ) {
            counter->key.lock ( );
```

```
            counter->removeReference();
            counter->key.unlock();
            }

        counter = rhs.counter;
    if ( rhs.counter ) rhs.counter->key.lock();
    init();
    if (rhs.counter ) rhs.counter->key.unlock();
        }
        return *this;
}
```

要求提供原子更新的其他计算使 RCBigInt 赋值操作的代价略有增加。测试的执行时间从 130ms 增加到了 150ms。但是，使用 RCBigInt 对象赋值仍然要比 BigInt 对象廉价，在这种情况下引用计数仍然是一个胜利者，如图 12.8 所示。

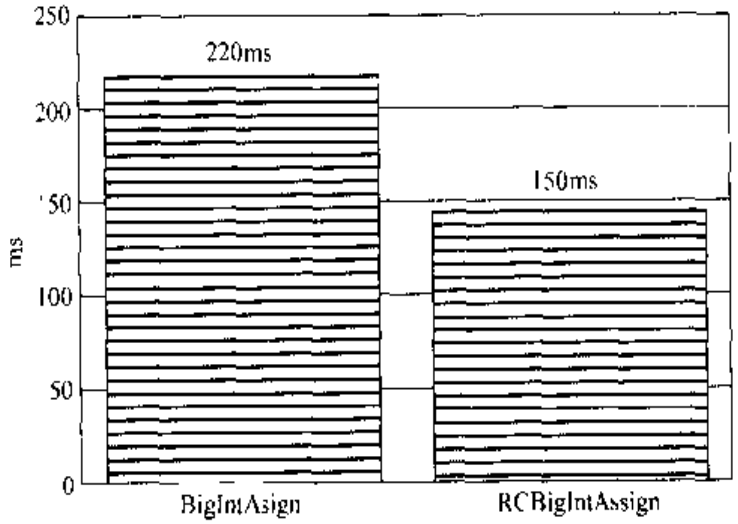


图 12.8 多线程 BigInt 的赋值速度

转移到多线程环境中对于对象的赋值没有太大的影响。但是，它确实对引用的创建和清除产生了巨大的影响。由于我们必须在堆中创建(或清除)BigInt 和 CountHolder 对象，所以 BigInt 引用的创建(清除)本来已属昂贵。紧接着，我们必须加上创建(清除)MutexLock 对象的代价。在这种情况下，简单 BigInt 类和引用计数 BigInt 类的性能差别已经几乎上升到 9 倍，其中简单 BigInt 更为有利一些，如图 12.9 所示。

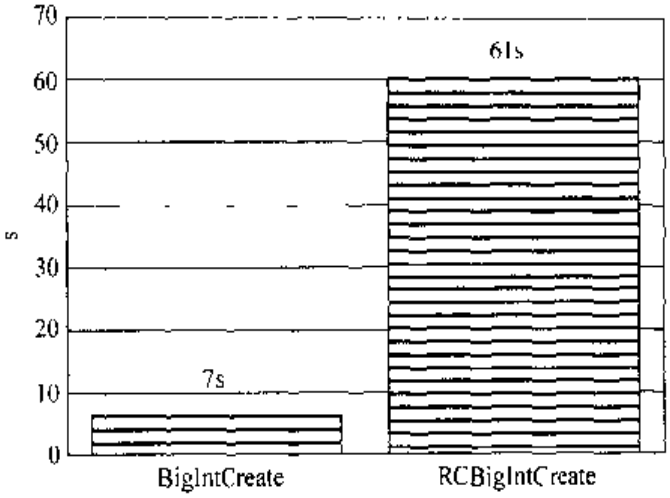


图 12.9 多线程 BigInt 创建速度

12.4 要 点

引用计数不是天生的性能获益者。引用计数、执行速度和资源节约形成了一种微妙的相互关系。如果性能是重要的考虑事项，则需要仔细地评估这种相互关系。根据使用环境的不同，引用计数可能有助于也可能有损于性能。下列任何一项都会增加引用计数的有利性：

- 目标对象消耗大量的资源。
- 资源的分配和释放很昂贵。
- 高度共享：由于使用赋值操作符和复制构造函数，所以引用计数可能比较大。
- 引用的创建和清除相对廉价。

如果情况相反，就应该跳过引用计数，向未计数的简单对象倾斜。

第 13 章

代码优化

提高 C ++ 性能的编程技术

如果我们的讨论只集中于特定领域或是特定应用程序方面的问题，那么对于大多数人来说将不会有多大用途，而只会对工作在特定领域或是相关应用程序中的程序员有益。为了吸引更多的读者，我们必须致力于遍布编程领域并可能出现在随便一个应用程序中的性能问题，同时不对问题范围进行任何假设。

尽管性能问题的表现形式无限多样，但是可以把它们划分成有限种类。每一类代表一组相同的性能缺陷。表面上它们可能看起来不一样，但是本质是相同的。以冗余计算为例，它可能有多种形式。请看下面不随循环改变的计算：

```
for ( i = 0; i < 100; i + + ) {
    a[i] = m * n;
}
```

每次循环都要计算 m * n 的值。只要把不随循环改变的计算移到循环外面，就可避免冗余计算：

```
int k = m * n;

for ( i = 0; i < 100; i + + ) {
    a[i] = k;
}
```

计算浪费还有另一种形式。在下例中，每个例程都要访问对当前线程而言属于私有的信息。pthread_getspecific( )返回一个指向线程私有数据结构的指针。由于 f( )和 g( )函数都需要访问该数据，所以每个函数都要调用 pthread_getspecific( )：

```
static pthread_key_t global_key;

void f ( )
```

```
{
    void * x = pthread_getspecific ( global_key);
    g ( );
    ...
}

void g ( )
{
    void * y = pthread_getspecific ( global_key);
...
}
```

由于 f ( )在执行的同一个线程中调用 g ( ),所以将在同一个执行流程中调用两次 pthread_getspecific ( ),这是一种浪费。如果 pthread_getspecific ( )是相当昂贵的调用,情况则会更糟。我们应该把 pthread_getspecific ( )调用的结果传递给 g ( )以消除对它的第二次调用:

```
static pthread_key_t global_key;

void f ( )
{
    void * x = pthread_getspecific ( global_key );
    g (x );   // Pass a pointer to thread_private data as a parameter
    ...
}

void g (void * y )
{
...
}
```

这两个例子是同种低效的不同表现:冗余计算。我们正在重新计算已知的结果:第一个例子中(m * n)的值和第二个例子中 pthread_getspecific ( )的返回值。

本章着眼于应用程序编码阶段引入的性能问题。由于这些性能问题太多,所以我们不会给出每一种实例。我们将涉及那些已被发现占主导地位的性能问题。这类问题通常是小范围的问题:解决它们不需要看太多的代码。把常量放到循环外面是这类代码优化的一个例子。确实没有必要理解完整的程序设计,只要理解具体的 for 循环便足够了。

性能方面的编码问题要比深层的设计问题易于解决。在第一次加入到 Web 服务器

开发团队的时候，我们不负责深层次的设计优化。我们对整个服务器设计没有足够的理解。从这一点上来说，我们无法对它进行大手术。但是我们立刻发现了刚才描述的有关pthread_getspecific ( )的低效。在处理 HTTP 请求过程中，至少要调用这个函数 100 次。通过给那些需要它的例程传递返回值，我们轻松地将服务器的吞吐量提高了 10%。

13.1 缓　　存

在现代处理器中，缓存通常与数据和指令缓存相关联。然而，缓存的时机出现在更广泛的环境中，包括应用程序级的编码和设计。缓存就是记住频繁计算和计算代价高的计算结果，这样就不必一遍又一遍地重复那些计算。

检测并在编码的舞台上实现缓存时机是比较容易的。在循环内对常量表达式求值是一种众所周知的性能低效的做法。例如：

```
for (…; ! done; …) {
    done = patternMatch ( pat1, pat2, isCaseSenstive ( ) );
    }
```

patternMatch ( )函数对两个字符串样本进行比较，第三个参数指明是否要求区分大小写。选择本身由函数调用生成，即 isCaseSenstive ( )。该函数只是简单地对区分大小写的 UNIX 平台和不区分大小写的 PC 平台进行区分。该选择是固定的。它独立于循环并且在每次迭代之间不会改变。因此应该在循环外面调用它：

```
int isSensitive =  isCaseSensitive ( );

for (…; ! done; …) {
    done = patternMatch ( pat1, pat2, isSenstive );
    }
```

现在对是否区分大小写只计算一次，便可把结果缓存在局部变量中并反复使用。

13.2 预先计算

预先计算与高速缓存密切相关。在缓存某个计算结果时，所付出的代价是在关键性能路径中完成一次计算。如果是预先计算，那么甚至不需要这种第一次的代价。可以在关键性能路径之外执行这种预先计算(比如说初始化时)，因而不会付出在关键性能路径内进行昂贵计算这样的代价。

为便于举例，我们回到 Web 服务器的实现上，以此为例：Web 服务器把相当一部分时间用在了操作字符串上。如果能够加快字符串操作的速度，性能将得到显著提高。我们通常会遇到的字符串操作任务是把字符串和字符转换成大写。如果浏览器发送给我们一个“Accept”请求报头，那么服务器程序必须能够在不区分大小写的情况下识别它。我们不应对“accept:”，“AcCept”或其他小写和大写字符组合形式进行区分。（Accept 报头告诉服务器浏览器可以接受哪些文档类型—— 如 text/html，image/gif...）

由于 memcmp (header,“ACCEPT:”,7)是区分大小写的，所以我们在调用 memcmp ( ) 之前必须把字符串转换成大写形式。可以这样做：

```
for ( char * p = header; * p ; p + + ) {          // Uppercase the header
    * p = toupper ( * p);
    }

it ( 0 = = memcmp ( header, "ACCEPT:", 7 ) ) {
    ...
}
```

在关键性能路径中，将会发现重复调用 toupper ( )的代价是无法接受的，即使是作为内联函数来实现 toupper ( )，但它仍将包含如下形式的条件语句：

```
return ( c > = 'a' && c< = 'z' )? c - ( 'a' - 'A' ) : c;
```

这是针对 ASCII 码的实现。对于 EBCDIC 字符集(EBCDIC 的字母表顺序不是相连的)来说甚至更为复杂。

我们不想为可能的库调用甚至内联的条件语句付出代价。选择先计算每个可能字符的相应大写值：

```
void initLookupTable ( )
{
    for (int i = 0; i < 256;  i + + ) {
        uppercaseTable[i] = toupper (i);
        }
}
```

由于 uppercaseTable 在启动时进行初始化，所以不管 islower ( )和 toupper ( )有多昂贵都没关系，其性能不是什么问题。

现在只需两条指令就可以把字符转换成大写形式。这明显地加快了字符串操作：

```
for ( char * p = header; * p ; p + + ) {
```

```
        *p = uppercaseTable[*p];
        }

    if( 0 == memcmp( header, "ACCEPT:", 7 ) ){
        ...
        }
```

在这种情况下，我们为关键性能路径所要求的其他字符函数预先计算出表查询：包括 islower ( )、isspace ( )、isdigit ( )和其他函数。

13.3 降低灵活性

您作为某一行业的专家，在提高性能之前通常要进行简化问题的假设。

一般情况下，Web 服务器需要知道客户发起请求所用的 IP 地址。IP 地址以类似于 9.37.37.205 的点分十进制数表示。起初，对于每一次请求，我们的代码都分配足够的堆存储空间来容纳客户的 IP 地址。在请求处理结束时再释放该空间。调用用于堆内存的 new ( )和 delete ( )是很昂贵的，因为必须为每次请求这样做，所以情况会更糟糕。尽管使用内存池可以减轻这种负担，但是还无法全部消除 new ( )和 delete ( )调用。

如果 IP 地址可能无限大，那么我们除了分配空闲的内存空间外别无选择。然而，我们是 TCP/IP 领域的专家，我们知道 IP 地址当前不会超过 15 个字节。最长的地址是：

```
xxx.xxx.xxx.xxx
```

如果再增加一个结束符 NULL，长度是 16 个字节。下一代 IP 地址(IPv6)将会更长一些，但仍然是有限的。由于 IP 地址的长度是有限的，所以以局部变量的形式把 IP 地址存储在堆栈上会更为有效：

```
char clientIPAddr[256];
```

在未来很多年内这对于我们的领域来说已经足够了。尽管 32 字节已经足够，但是出于安全方面的考虑我们还是选择了 256 个字节。这个有限的字符数组消除了昂贵的 new ( )和 delete ( )调用。我们为了获得更好的性能而牺牲了并不需要用到的灵活性。

13.4 80-20 规则：提高常用路径的速度

80-20 规则有许多方面的应用：80%的执行情况会遍历 20%的代码，80%的时间消耗在执行路径所遇到的 20%的函数之上。80-20 规则即“过早的调整是一种罪过”这种理论

得以形成的决定性因素。如果随便地调整了所能想到的一切，那么不仅浪费了80%的努力，而且会把设计弄得无法修正。

HTTP规范是一份100页的文档，它描述了Web服务器必须处理的所有可能的HTTP请求。目前，大多数经过Web的HTTP请求都很简单。它们只包含请求潜在包含的HTTP报头的一个小子集。找出这些典型的请求报头是非常简单的：由于Microsoft和Netscape在浏览器市场占有决定性的份额，所以所要做的就是查看一下这两种浏览器所发送的请求报头。这是80-20规则的另一种表现——20%的可能输入将占用80%的时间。

HTTP请求报头决定了请求的类型并且通常是经历分开的执行路径。对程序员资源的有效使用是调整那些出现在80%的时间中的20%的请求类型。

80-20思想还有另一点表现。不仅要集中于典型的执行路径，而且要充分利用一个事实，即大多数输入数据会局限于整个输入空间中很小的范围内。下面是一些例子。

HTTP的Accept报头是HTTP请求的一部分，它提出了浏览器所接受的文档格式。HTTP规范允许把任意的大小写字符组合作为Accept报头。在阅读字符串标记并希望确定它是否是Accept报头时，执行以下代码即可：

```
memcmp ( "Accept:" , header, 7 )
```

我们需要执行区分大小写的字符串比较。我们实现了一个用于进行区分大小写字符串比较的自产版本：

```
int memCaseCmp ( char * , char * , int ) {...}
```

为了满足HTTP要求，正确的操作应该是：

```
if ( 0 = = memCaseCmp ( "accept:", header, 7 ) ) {
    // This is the Accept header
    }
```

然而，memCaseCmp ( )并不是廉价的。它既需把整个字符串转换成大写形式，又要调用memcmp ( )。这正是领域专家必须发挥专长的地方。就像我们说过的那样，Microsoft和Netscape在浏览器市场占有决定性的份额。它们的浏览器所发送的Accept报头正好是"Accept:"。这不仅是HTTP规范所允许的许多种大小写字符组合中的一种，更是我们将在95%的时间内要接收的字符组合，因此它是我们所关心的。下面的测试力争利用这一点：

```
if ( ( 0 = = memcmp ( "Accept:" , header, 7 ) ) ||          // An intelligent gamble ...
    ( 0 = = memCaseCmp ( "accept:", header, 7 ) ) )
```

```
// This is the Accept header
}
```

对于95%的输入，关于memcmp()测试都将成功，并且不会调用昂贵的memCaseCmp()。

最后的代码示例正确地把我们带入了有关求值顺序的另一个80-20问题。通常条件表达式是由子表达式的逻辑组合所生成。我们判断以下的表达式：

```
if (e1 || e2) {...}
```

或：

```
if (e1 && e2) {...}
```

这时，求值的顺序与性能有很大的关系。以第一个if语句为例：

```
if (e1 || e2) {...}
```

如果if (e1 || e2)求值的结果是FALSE，我们就不说了。此时两个子表达式e1和e2都必须进行求值。然而，当(e1 || e2)求值的结果是TRUE时，如果e1为TRUE，那么将不会对e2求值，表达式的总代价将降低。在以下的讨论中，我们把注意力缩小在(e1 || e2)为TRUE的情况。

如果e1和e2都可能为TRUE，那么计算代价较小的子表达式应该放在计算顺序中的前面。相反，如果e1和e2有相同的计算代价，则最有可能为TRUE的那一个应该出现在前面。在更为普遍的情况中，令：

p1 =假定(e1 || e2)为TRUE情况下e1为TRUE的条件概率

c1 =对e1进行求值的计算代价

我们尽量使计算(e1 || e2)的预期代价最小：

```
cost = c1 + (1 - p1) * c2
```

我们的例子只处理两个子表达式的情况，但是很容易把它扩展到逻辑OR表达式中的任意数量的子表达式。

前面我们曾遇到一个用于HTTP Accept报头的条件语句判断：

```
if ( ( 0 == memcmp ( "Accept:", str, 7 ) ) ||
    ( 0 == memCaseCmp ("accept", str, 7 ) ) ) {

    // This is the Accept header
    }
```

在该例中，

```
e1是（0 == memcmp("Accept:", str, 7））

e2是（0 == memCaseCmp("accept:", str, 7））
```

在这种情况下，if语句为TRUE，由于e2是一种区分大小写的字符串比较，所以它在100%的情况下将计算为TRUE。因此p2 = 1.0。e1只是在90%的时间内为TURE。因此p1 = 0.9。计算代价大约是：

```
c1 = 10条指令
c2 = 100条指令
```

我们的预期代价为：

```
cost = 10 + 0.1 * 100 = 20
```

如果颠倒一下e1和e2的顺序，则预期代价将为：

```
cost = c2 + (1 - p2) * c1 = 100 + 0 * 10 = 100
```

因此，if（e1 || e2）要比if（e2 || e1）好。

对于像if（e1 && e2）这样的逻辑AND表达式存在类似的逻辑。假设整个表达式为FALSE，那么我们不必计算出所有的子表达式，而是尽早结束求值。这种情况下的概率稍有不同：

```
p1 = 假设(e1 && e2)为FALSE的情况下e1为FALSE的条件概率
```

在实践中很容易遇到这样的选择，例如，当规范要求实现判断并处理如下一些很少见的组合时：

```
if（rareCondition_1（）&& rareCondition_2（））{
// Handle it
}
```

由于这种组合很少见，所以大多数情况下这种条件将不会成立。我们希望它在第一个表达式上就尽快为假。顺便说一句，这就是符合HTTP 1.1的服务器要比符合HTTP 1.0的实现慢的原因。HTTP 1.1规范要复杂得多，它强制服务器对各种深奥的情况进行很多检查，而这些深奥的情况很少发生。例如，满足HTTP 1.1规范的浏览器可能请求一个巨大文档的一部分，而不是整个文档。在实践中，绝大多数请求是请求整个文档。然而，服务器仍然要检查部分文档请求并正确地处理它。

13.5 缓式计算

为一项最后可能不需要的计算付出性能代价可不怎么样。这在复杂的代码中却经常发生。不应该执行"只在某种情况下"才需要的昂贵计算，昂贵计算应该只在需要时才执行。这通常意味着把计算延迟到真正需要的时候再进行，故亦称为缓式计算(Lazy Evaluation)[Mey96,ES90]。

回溯到纯粹使用C编程的时代，那时我们习惯了在每个例程的开始预先定义所有的变量。在C++中，对象的定义触发构造函数和析构函数，这可能是昂贵的。它导致了一些我们希望避免的立即求值。缓式计算原则建议我们应该把对象的定义放到使用它们的范围之内。如果我们不准备使用对象，那么为对象的构造函数和析构函数付出代价则没有意义。这听起来没什么意思，但在实践中是会发生的。

在一个C++网关程序中，我们有一段运行于AIX内核的性能很重要的代码。这段代码在上下游通信适配器之间为消息选择路由。在我们所使用的对象中，有一个对象在构造和清除时非常昂贵：

```
int route ( Message * msg )
{
    ExpensiveClass upstream ( msg );

    if (goingUpstream ) {
        ...     // do something with expensive object
        ;
    // upstream object not used here.

    return SUCCESS;
}
```

upstream对象是非常昂贵的。在原来的代码中，它在函数的顶部立即构造。只在一半的时间中会有消息路由到上游。当消息路由到下游时(50%的时间)，根本不使用upstream对象。对于下游消息来说，对upstream对象的求值纯粹是浪费。更好的方案是在真正需要upstream对象的场合定义这个昂贵的对象：

```
int route ( Message * msg )
{
    if (goingUpstream ) {
    ExpensiveClass upstream ( msg );
```

```
        // do something with the expensive object
        }
    // upstream object not used here.
        return SUCCESS;
}
```

在[Mey97]中，Meyers 得出另一个重要的结论，即不仅应该延迟对象的创建，将其放到正确的范围内，还应该把创建延迟到具备了有效创建所需要的各种成分以后。例如：

```
void f ( )
{
    string s;                    // 1
    ...
    char * p = "Message";        // 2
    ...
    s = p;                       // 3
    ...
}
```

在语句 1 中，我们使用默认的 String::String ( )构造函数创建了 String 对象 s。后来，在语句 3 中，我们把一些实际内容装入了 s。对象 s 实际上直到语句 3 执行完以后才完成。因此，这就是一种低效的创建。它需要以下操作：

- 默认的 string 构造器，在语句 1 中；
- 赋值操作符，在语句 3 中。

缓式计算原则可以把这些操作减少成惟一的对构造函数的调用。这是通过把 string s 构造延迟到具备了需要的部分之后来实现的：

```
void f ( )
{
    char * p = "Message";
    string s (p);              // invoke String::String (char * )
    ...
}
```

通过延迟 s 的创建，我们只需要进行一步创建——这明显地会有更高的效率。

13.6 无用计算

缓式计算涉及那些不总是需要的计算，它依赖于执行流程。而无用计算针对的是那些我们从不需要的计算。这些完全是没有意义的计算，不管执行流程如何，都不会使用这

些计算的结果。

有关无用计算的一个微妙的例子就是对成员对象所进行的无用初始化。

```
class Stutent {
public:
    Student ( char * nm );
    ...
private:
    string name;
};
```

Student 构造函数把输入的字符指针转换成 string 对象来表示学生的姓名：

```
Student::Student ( char * nm )
{
    name = nm;
    ...
}
```

在 Student 的构造函数体执行之前，C++ 保证已经创建所有的成员对象。在我们的例子中就是 string 型 name 对象。由于我们没有明确地告诉编译器怎样做，所以编译器隐含地插入了对 string 默认构造函数的调用。该调用在执行 Student 的构造函数体之前进行。构造函数体之后紧跟着：

```
name = nm;
```

这条语句实际上清除了 name 对象以前的内容。我们从来没有用到编译器产生的对 string 默认构造函数的调用所产生的结果。通过对 Student 构造函数的初始化列表明确地指明 string 构造函数，可以消除这种没有意义的计算：

```
Student::Student ( char * nm ) : name (nm )  // explicit string constructor
{
    ...
}
```

由于明确地告诉了编译器使用哪个 string 构造函数，所以编译器不会再产生对 string 默认构造函数的隐含调用[Mey97]。因此我们获得了对成员 string 对象的单步创建。

13.7 系统体系结构

内存访问的代价是相差很大的。在某种特定的 RISC 体系结构中，如果数据位于数据缓存中，那么数据访问需要一个 CPU 时钟周期；如果数据位于主存中(缓存失败)，那

么需要 8 个时钟周期；如果数据在磁盘中(页面错误)，那么需要 400 000 个时钟周期。尽管确切的时钟周期数量可能不同，但不同的处理器体系结构在总的关系上是相同的：缓存成功、缓存失败和页面错误之间的速度差别是不同数量级的差别。

在访问数据时，数据缓存区是第一个要查找的地方。如果数据不在缓存中，硬件就会产生一个缓存失败。缓存失败导致数据从 RAM 或磁盘加载到缓存中。缓存以缓存行为单元进行加载。缓存行通常比我们正在寻找的特定数据项大。在 4 字节整数上的缓存失败可能造成 128 字节的缓存行加载到缓存中。由于相关数据项在内存中位于附近的可能性大，所以这是有用的。如果随后的指令要访问同一缓存行中的其他数据项，那么我们会在第一次请求该数据时就受益于缓存成功。尽我们所能，应该帮助我们的代码展示这种引用的位置性。

以类 X 为例：

```
class X {
public:
    x( ) : a( 1 ) , c( 2 ) { }
    ...
private:
    int a;
    char b[4096];              // buffer
    int c;
};
```

X::X( )构造函数初始化成员 a 和 c。符合标准的编译器将按照声明的顺序来排列对象 X：成员 a 和成员 c 将由 4 096 字节的成员 b 所分离，并不会位于相同的缓存行中。X::X( )在访问 c 时可能会遭到缓存失败。

由于 a 和 c 由相邻的指令访问，所以可以通过把它们放在相邻的位置而使得缓存更为友好。

```
class X {
      ...
private:
    int a;
    int c;
    char b[4096];              // buffer
};
```

现在 a 和 c 更有可能位于相同缓存行。由于 a 在 c 之前访问，所以在需要 c 时，几乎

可以保证 c 位于数据缓存之内。

13.8 内存管理

分配和释放堆内存是昂贵的。从性能的角度考虑，使用不需要显式管理的内存要廉价得多。定义成局部变量的对象位于堆栈上。该对象所占用的内存是留给相应函数的堆栈内存的一部分，对象在该函数的范围内定义。局部对象的一种选择是使用 new ( )和 delete ( )来获取和释放堆内存：

```
void f ( )
{
    X * xPtr = new X;
    ...
    delete xPtr;
}
```

另一种性能更佳的选择是定义 X 类型的局部变量：

```
void f ( )
{
    X x;
    ...
}    // no need to delete x.
```

在后一种实现中，对象 x 位于堆栈上，不需要事先分配，也不需要在退出时释放，在 f ( )返回时堆栈内存自动释放，从而我们节省了调用 new ( )和 delete ( )的高昂代价。

类似的问题存在于成员数据中。但这次不是堆与堆栈内存了，而是要选择在包含对象中嵌入指针还是整个对象的问题：

```
class Z {
public:
    Z ( ) : xPtr ( new X ) {...}
    ~Z ( ) {delete xPtr; ...}

private:
    X * xPtr;
    ...
}
```

在构造函数中调用 new ( )和在析构函数调用 delete ( )显著地增加了 Z 对象的代价。通过在 Z 中嵌入 X 对象可以消除内存管理代价：

```
class Z {
public:
    Z ( ) {...}
    ~Z ( ) {...}

private:
    X x;
    ...
}
```

通过把 X 指针替换成 X 对象，我们牺牲掉了多态地使用该成员的可选项。如果确实需要灵活性，则这种优化是不合适的。再次重申：性能是一种交易。

13.9 库和系统调用

计算机语言的发展不断地简化着解决复杂问题的方案设计和编码任务。随着语言在表现能力上的发展，我们使用它们来解决高度复杂的问题。从理论上说，一台使用简单语法的图灵机和我们现在所拥有的任何编程语言一样强大。我们的意思是，从功能上说，可以使用图灵机编写任何算法，只不过这是一个很困难的编程环境。对于汇编语言也一样。我们不使用汇编语言开发 Web 服务器的原因在于其可行性：使用低级语言来开发复杂问题的软件解决方案太困难且太耗时了。

如果想把两个整数相加，那么这在 C ++ 之类的语言中是很简单的：

```
k = i + j;
```

如果要在汇编语言中尝试，将碰到一些必须自己处理的微小细节：

- 把整数 i 装入寄存器 X；
- 把整数 j 装入寄存器 Y；
- 把寄存器器 X 和 Y 的内容相加；
- 把结果存储到整数 k 的内存地址中。

在高级语言中，所有微小的细节由编译器处理，我们通常忽略它。这种简单性导致了更高的生产力和承担更为复杂的编程任务的能力。

有几种隐藏复杂性的方式：我们可以把它隐藏在硬件中和软件中。软件隐藏通过编译器把我们的源代码转换成汇编指令来实现。我们也可以自己实现：把复杂性封装在系

统调用、库和类实现中。

以连接两个字符串为例，需要分配足够大的内存来容纳两个字符串再加上一个结束符 null：

```
{
    char * s0;
    char * s1 = "Hello";
    char * s2 = "World";

    s0 = ( char * ) malloc ( strlen ( s1 ) + strlen ( s2 ) + 1 );
    strcpy ( s0,s1 );            // Copy first string
    strcat ( s0,s2 );            // Concatenate the second
    ...
}
```

在 C++ 中，所有这些细节都封装在 string 类中实现。重载 string 的“+”操作符将把我们的客户代码简化成：

```
{
    string s1 = "Hello";
    string s2 = "World";

    string s0 = s1 + s2;
    ...
}
```

当我们上升到编程简单性的更高层次以后，细节工作并没有消失，而是在别的地方完成。从性能的观点来看，我们不能忽略背后的工作。这是性能工程师不同于其他开发者的地方，其他开发者惟一的考虑是迅速把所有需要的功能放到一起并组装起来。而性能开发者必须知道隐含的代价，因为这对系统性能有巨大的影响。

作为具体的例子，我们看一下 pthreads 库的实现。该库是为 UNIX 平台上的用户提供线程服务的简单接口。pthread_mutex_lock ( )和 pthread_mutex_unlock ( )调用提供互斥锁。如果锁定了资源，那么就排斥了其他访问，直到对资源解锁为止。您所看不到的是，在调用 pthread_mutex_lock ( )时，pthread 库的实现要对您的线程还没有拥有锁这一点进行核对，这样就防止了死锁。在调用 pthread_mutex_unlock ( )时会对调用者真正拥有锁这一点进行核对。除非该线程是事前锁定资源的那个线程，否则就不能对资源进行解锁。所有这些必要的错误检查增加了执行路径长度并降低了性能。

在许多情况下，应用程序对锁的使用可能非常简单，这时所有这种开销都是浪费。您

的代码设计可能已经确保了这种前提:要进行锁定的线程现在没有拥有锁,要进行解锁的线程是事先锁定资源的线程。在这种情况下,可以创建一种简单的锁模式,这种模式建立在本地操作系统所提供的原始积木块基础之上,这显然更为有效。如果不了解库的实现细节,那么可能不会知道自己正在为并非真正需要的功能而付出性能上的代价。

我们仍然停留在 pthread 库的范围之内来举一个有关的例子。您可能让每个线程都与一块私有的数据区域相关联。线程的全局变量可能存储在该区域[NBF96]。如果需要访问线程的私有数据,那么需要一个指向内存中该线程专用区域的指针。可以通过调用 pthread_getspecific ( )来得到该指针。pthreads 库实现通过维护当前线程与其私有数据之间的关联来实现这种神奇的功能。由于线程可能有来有往,所以这种关联是动态的。必须串行化对该集合的访问。结果是,对 pthread_getspecific ( )的调用隐藏了内部的串行化代码。我们在获取与特定线程相关联的数据时必须锁定该集合。这可能是您需要知道的事情,以便弄清楚是否要关心可伸缩性。如果多个线程对 pthread_getspecific ( )执行频繁的调用,那么串行化逻辑将由于线程彼此将对方关在外面而造成可伸缩性阻塞。

一旦您知道了 pthread_getspecific ( )的内幕,就可能选择使用其他方法来四处传递特定于线程的数据。不是让数据项成为线程的全局变量,而是可以以函数调用参数的形式来向各处传递数据项。如果确实需要频繁地访问线程的全局变量,则可以执行一次 pthread_getspecific ( )调用,然后以函数调用参数的形式向各处传递该指针。

13.10 编译器优化

在不需要从您这里对源代码进行任何干涉的情况下,差不多的编译器都能代替您采用相当多的重要优化。第一个出现在脑海中的优化是寄存器分配。当变量位于寄存器中时,加载和存储它要更快一些。否则,我们将不得不花费一些时钟周期来从其他地方得到它。在这种情况下,如果变量不在缓存中,那么事情会更糟。在现代的体系结构中,一次内存访问需要 5 个时钟周期以上。如果变量位于寄存器中,就可以避免这一切。由于可以对执行路径中的许多方法使用寄存器分配,所以寄存器分配是一种重要的优化。当涉及循环索引变量时这会特别明显。每次循环迭代时都会访问和更新那些变量。

我们特别注意的第二个优化就是内联。第 8、9 和 10 章已经强调了内联可以显著地改善许多 C++ 程序性能的观点。另一个值得 C++ 程序员注意的优化是我们在第 4 章中讨论的返回值优化。还有很多种编译器优化。有关各种编译器优化的更为详尽的内容可以参阅[O98]。我们要努力说明的是没有必要列出所有可能的优化,但是一定要很清楚地知道,您不能把它们看成是想当然的。

通常情况下,编译的默认设置根本不执行任何优化。这意味着这些重要的速度优化

将不会发生——即使您在代码中加上了 register 和 inline 关键字也没有用。编译器有权忽略它们，并且它经常这样做。为了增加优化的机会，必须通过向命令行添加开关(UNIX 类的平台上使用-O)或是在 GUI 界面上选择速度优化选项人为地激活编译器优化。

由于编译器优化会随应用程序的不同而不同，所以很难精确地量化编译器优化的影响。以我们的经历，曾经看到各种应用程序有 20%～40%的速度提高。这显然是“诱人的果子”。

13.11 要 点

编码优化在范围上是局部性的，并且不需要理解整个程序的设计。当您参加到不理解其设计且正在进行的工作中去时，这是良好的起点。

最快的代码是从不执行的代码。尝试以下几方面去剔除那些昂贵的计算：

- 您要使用该结果吗？这听起来没有什么意思，但是它确实会发生。有时我们执行计算但从不使用其结果。
- 您现在需要该结果吗？把计算延迟到真正需要它的地方。过早的计算可能从不为任何执行流程所需要。
- 您是否已经知道该结果？我们曾经看到过昂贵的计算被执行，而其结果在两行代码前就已知道。如果已在该流程的早些时候计算出了该结果，那么应让该结果成为可重用的。

有些时候可能无法剔除该计算，这时就必须执行该计算。现在的挑战是让它加快：

- 该计算是否过于通用？您只需和该领域所要求的一样灵活就够了，而不是更灵活。要利用简化的假设，以降低灵活性来增加速度。
- 有些灵活性隐藏在库调用中。通过为库调用开发您自己的版本，可能获得速度提高。这些库调用要被足够频繁地调用，以便对得起您的努力。要让您自己熟悉那些所使用的库和系统调用的隐含代价。
- 使内存管理调用的数量达到最少。在很多编译器中它们都是昂贵的。
- 如果考虑一下可能输入数据的集合，则可以发现 20%可能输入的数据出现在 80%的时间里。应以损害其他不经常出现的场合为代价提高典型输入的处理速度。
- 缓存、RAM 和磁盘访问之间的速度差别是明显的。要编写缓存友好的代码。

第 14 章

设计优化

提高 C ++ 性能的编程技术

性能优化可以粗略地分为两种主要类型:编码和设计。我们把编码优化定义为不需要理解整个问题领域或应用程序执行流程的优化。按照定义,编码优化是局部化的,并很好地独立于周围的代码。设计优化是所有剩余的优化,即除了"诱人的果子"之外的一切。这些优化是全局性的——它们依赖于其他组件以及相隔很远的代码部分。设计优化的作用遍及全部代码。这不完全是一种精确的数学定义,有些优化将由于这种不精确的定义而陷入困境。通过本章提供的例子,所有的思路将变得更为清晰一些。

14.1 设计灵活性

在过去的 10 年中,我们遇到很多 C ++ 工程,它们远远不能满足所要求的性能。那些性能失败的核心原因是设计和实现时为了过度的灵活性和可重用性而进行的调整。由于可重用代码方面的成见,生产出来的软件缺乏高性能,甚至无法使用,更不用说重用了。C ++ 把 OO 编程带入了主流,并且为广泛的编程团体充满热情地采用 OO 铺平了道路。OO 的教育组织把 C 程序员转化成了新的一代人,这一代人的主要注意力就是创建通用、灵活和可重用的软件。问题是性能和灵活性通常互相排斥。这不是不良设计或实现的产物。我们讨论的是两股朝两个相反方向拉动的力量之间的基本拉力。在数学上,如果 $X * Y$ 的结果是常数,那么 X 越大,Y 必将变得越小。这是针对于某一阶段而言的。一般来说,性能和灵活性之间存在着类似的关系。

如果您试图找出性能和灵活性能够真正和平共处的对称点,那么您可能举出标准模板库(Standard Template Library,STL)作为证据。STL 是强大的、通用的、灵活的和可扩展的。在某些情况下 STL 所产生代码的性能可以和硬编码的 C 相抗衡。这好像违背了我们的论点。现在来深入探讨一下。

以 STL 的 accumulate ( )函数为例,它可以累加任意对象集合。它接收 3 个参数:

* 指向集合开始位置的迭代器

- 指向集合结束位置的迭代器
- 初始值

accumulate ( )函数将从头完成把对象累加到其初始值中的工作。它可以累加不同的对象集合:整数、浮点数、字符、字符串,甚至是用户定义的数据类型。它不关心集合是不是数组、列表、向量或其他集合,它所关心的是要传递两个控制迭代起止位置的迭代器。此外,accumulate ( )还能够执行其他二元操作,而不仅仅是相加。可以以可选参数的形式指定二元操作符。

很明显,这是一段很好的代码。它是高度灵活和通用的。有时,accumulate ( )甚至会表现出极好的性能。它可以累加如下的整数数组:

```
sum = accumulate (&x[0], &x[100], 0 );
```

它和如下自己编写的版本一样快:

```
int sum = 0;
for ( int i = 0; i < 100;  i++ )
{
    sum += x[i];
}
```

这相当不错。accumulate ( )函数的实现有点像以下[MS96]中的代码行:

```
template <class InputIterator, class T>
T accumulate ( InputIterator first,
               InputIterator beyondLast,
               T initialValue )
{
    while ( first != beyondLast ) {
            initialValue = initialValue + *first++;
            }
}
```

accumulate ( )实现的优势源于它很少对集合及其包含的对象进行假设。它只知道两件事情:

- 从 first 迭代器开始,它将顺序进行,最后遇到 beyondLast 迭代器。
- 集合中包含的对象是+操作符可接受的参数。

这两个微小的假设使得 accumulate ( )具有极大的灵活性。但它们同样会变成它的性能弱点。对于高性能的整数相加来说这些假设可能是足够了,但是其他具有复杂结构

的类将不得不放弃对这些类的特定知识的利用。以 string 为例，假设您想连接这样的一个字符串集合：

```
vector<string> vectorx;

for (i = 0; i < 100; i++) {
    vectorx.push_back ("abcd");
    }
```

第一种选择是使用 accumulate ( )。除非有迫不得已的原因，否则您一定不想重新发明罗盘：

```
string empty;                // initial value for result string.

result = accumulate ( vectorx.begin ( ), vectorx.end ( ), empty );
```

这会完成所需要的工作，但是它会有多快呢？我们对如下代码的 100 000 次迭代进行了计时：

```
string empty;                // initial value for result string.

for (i = 0; i < 100000;  i++) {
result = accumulate ( vectorx.begin ( ), vectorx.end ( ), empty );
    }
```

这段代码消耗了 29s。能够用手工自产的解决方案战胜 STL 吗？我们进行了以下尝试：

```
void stringSum1 ( vector<string> vs, string& result )
{
    int i = 0;
    int totalInputLength = 0;
    int vectorSize = vs.size ( );
    int * stringSize = new int[vectorSize];

    for (i = 0; i < vectorSize;  i++) {
        stringSize[i] = vs[i].length ( );
        totalInputLength += stringSizes[i];
        }

    char * s = new char [totalInputLength + 1];
```

```
    int sp = 0;
for (i = 0; i < vectorSize;  i + +) {
        memcpy ( &s[sp], vs[i].c_str ( ), stringSizes[i] );
        sp + = stringSizes[i];
        }

    delete stringSizes;
    result = s;

    return;
}
```

同样，我们对这种方案进行了计时：

```
string result;              // initial value for result string

for (i = 0; i < 100000;  i + +) {
    result = stringSum1 ( vectorx, result );
    }
```

我们自产的解决方案要快得多——只有5s，速度提高了5倍多。由于我们选择了为提高性能而牺牲了通用性，所以超过accumulate ( )很多。我们的stringSum1 ( )实现进行了如下特定于string的假设：

- 我们知道现在要相加的惟一对象就是字符串。
- 我们知道string相加需要内存分配来容纳处理后的string结果。
- 我们知道通过调用string::length ( )，每个string对象能够告诉我们它的大小。

我们使用这些假设事先算出结果string将会是多大。我们使用该知识在一个步骤中分配足够大的内存缓冲区来容纳结果string。这是我们的实现比accumulate ( )函数快得多的主要原因。accumulate ( )函数为分配另外的空闲内存缓冲区要进行多次调用。它无法事先知道到底需要多大的缓冲区。更为糟糕的是，每当accumulate ( )用完缓冲区空间之后，它需要把旧缓冲区的内容复制到新获得的更大一些的缓冲区中。

这是一个说明如何减少灵活性从而提高效率的不错的例子。我们的stringSum1 ( )无论如何都无法与accumulate ( )的灵活性和强大相比。除了string对象之外我们不能累加任何东西。这就相当于日光灯泡与激光之间的差别。不是激光比日光灯泡包含更多的能量，而是它只把光束聚焦于狭窄的范围。尽管我们不能用激光来照亮房间，但是却可以用它在砖墙上钻一个洞。

我们的目的不是挑剔STL，绝对不是这样。您应该只考虑在性能热点上替代STL。

由于这些热点难以预知，所以在默认情况下，应该在软件开发的早期阶段全面使用 STL。这里讨论的要点是强调一个原则，这个原则由于人们疯狂地冲向创建可重用软件的潮流而被抛开：在软件性能和灵活性之间有一种基本的平衡。在通常的应用程序中特定的环境和简化假设是很常见的。软件库是通用的，但软件不是。缩小代码的视野会产生更高的效率。

14.2 缓　　存

第 13 章已经讨论了与编码有关的缓存优化。不过在设计舞台上同样有大量的缓存机会。本节中我们描述 3 种设计优化来代表这一类的优化。

14.2.1 Web 服务器时间戳

在 HTTP 请求服务期间，存在多个时间，服务器必须对这些时间采集时间戳以便记录当前时间。例如，在服务器和浏览器之间建立了 TCP 连接之后，服务器为这个特定的连接启动一个计时器。如果在该连接上的请求到达之前计时器超时，那么将会终止连接。这可防止攻击者利用从不提交任何请求的连接淹没服务器。这些僵死的连接会占用服务器资源（如线程等）进而挂起服务器。为了对付向服务器采取的这种类型的攻击而特地设计了僵死连接的服务器终止。

Web 服务器还为每次请求产生了日志项。这对于了解站点的流行程度、正在访问站点的人以及在您的站点上最为需要的特定信息是非常有用的。请求日志包含了时间戳，用来指明为请求提供服务的时间。

我们在最后一次计算中发现，每次请求处理会引起 7 个时间戳。由于一次请求只延续几 ms，而需要的时间间隔是以 s 为单位测量的，所以为每次请求产生惟一的时间戳并重用它将更为高效。在某些平台上，时间戳的产生是昂贵的。在某种特定的平台上，一次典型请求的计算代价是 120 000 条指令。每个时间戳调用花费 1 500 条指令，7 次调用的合计大约是 10 500 条指令。通过缓存第一个时间戳的结果，我们消除了 6 次时间戳计算并节省了 9 000 条指令。在请求处理速度方面大约有 8%的提高。

我们把第一次时间戳计算的结果储存起来，并以函数调用参数的形式把它依次传给其他 6 个函数。这是一个在堆栈上缓存结果的例子。这个特定值从不用在内存中存储。

14.2.2 数据扩展

缓存以前的结果有许多不同的方式。其中之一是作为对象自身内的数据成员。如果您发现自己在不断地重复使用一个对象来获得相关的结果，那么把结果作为对象的数据

成员储存起来并在将来的计算中重用它将会更好一些。

Web服务器消耗了大量的时间来操作字符串。字符串在代码间传送的过程中，多次计算同一字符串的长度是常有的事。在本地string类实现中，您可以看到自己正在进行过多的strlen()调用：

```
class string {
public:
    ...
    int length() const {return strlen(str); }
private:
    char * str;
}
```

增加存储字符串长度的数据成员会更为有效：

```
class string {
public:
    ...
    int length() const {return strLen; }
private:
    char * str;
    int strLen
}
```

当string对象创建或修改时会计算成员strLen。随后对计算字符串长度的调用将返回已经计算好的长度从而避免了昂贵的strlen()调用。

记住字符串的长度还有另外一个好处：mem*()调用系列(memcpy()、memcmp()等)要比str*()调用家族(strcpy()、strcmp()等)更为高效。mem*()调用家族的高效率来源于这些函数不需要为可能的null结束符而检查每个字节。mem*()调用系列需要字符串的长度作为参数。如果储存了string对象的长度，可用mem*()调用系列来获得更高的效率。

14.2.3 公用代码陷阱

典型的Web服务器实现支持多个紧密相关的协议。大多数实现既支持HTTP，也支持安全套接字层(Secure Socket Layer，SSL)(SSL的作用是为Web文档提供安全连接，本质上是一种安全的、加密的HTTP。)自然，大多数实现Web连接的代码是HTTP和SSL公用的。这有利于重用，但是要注意保护性能。在我们以前的一个Web服务器实现

中，我们尽最大努力重用而遭到了性能损失。在处理请求的执行路径中，有许多区分 SSL 代码和 HTTP 代码分支的判断点。每一个这样的分支点都需如下形式的条件判断语句：

```
bool HTHandle ( StreamSocket& mySocket )
{
    ...
    if (mySocket->SSL ( ) ) {
       requestHandler->perform_the_SSL_action ( );
       }
    else {
       requestHandler->perform_the_HTTP_action ( );
       }
    ...
}
```

这种重复计算是一种执行速度上的负担。如果在一次请求的生命周期中执行 20 次这样的检查，那么就会有 19 次是多余的。HTHandle ( )的调用者已经知道请求的类型(SSL 或 HTTP)，但是我们没有把这个知识记下来并重用它。

由于大多数这种检查聚集在少量的例程中，因此解决这种冗余的一种可能方式就是复制每一个例程，比如说，把例程 HTHandle ( )拆分成 HTHandle_HTTP ( )和 HTHandle_SSL( )。现在，每个例程都知道了自己正在处理的连接类型并能够避免 HTTP/SSL 选择。

一种更好的解决方案是从基类 RequestHandler 中获得 SSLRequestHandler 和 HTTPRequestHandcr。现在，HTHandle ( )变成了虚函数，HTTP/SSL 选择只需要通过虚函数的确定，而且仅需确定一次。代码只是简单地调用：

```
RequestHandler->HTHandle ( mySocket )
```

而 HTHandle ( )将被确定为正确的例程。利用虚函数调用解决方案的优点是只需要进行一次选择点判断来区分 SSL 和 HTTP。这比在合并的 SSL/HTTP HTHandle ( )实现中多次重复访问这种选择要高效得多。

实际上，我们的服务器实现要比这里的描述复杂得多。我们的服务器不仅是 HTTP 和 SSL 服务器，而且还是代理服务器和 FTP 服务器。避免在 HTTP/SSL/FTP/代理之间进行多达几百次的重复选择是一种真正的设计挑战。

类似的问题存在于当前的 TCP/IP 实现中。当前处于主导地位的 IP 实现是第 4 版(V4)。新兴的下一代 IP 实现是第 6 版(V6)，也被人们称为 IPNG(IP 下一代，IP Next Generation)。在最近几年，TCP/IP 供应商在抓紧向其现有 V4 TCP/IP 实现提供 V6 支

持。由于 V4 和 V6 是紧密相关的，所以它们会执行许多通用代码。单独使用 V4 的基本代码并在必要的地方加上 V6 分支，这样做可能非常诱人。但这将因如下形式的条件语句而产生代码垃圾：

```
if (V4 ) {
    do_one_thing ( );
    }
else {  // V6
    do_something_else ( );
    }
```

使用这种实现，无论是 V4 还是 V6，其性能都要有所下降。所以需要一种更为高效的设计来消除那些多处存在的分支选择点。一旦执行流程确定了是 V4 还是 V6，那么就应该保留这一点并重用它。

14.3 高效的数据结构

通常把软件的性能等同于实现中所使用的算法和数据结构。有时候认为算法和数据结构是软件性能最为重要的因素。不管事实上是否是这样，使用高效的算法毫无疑问是软件效率的必要条件。如果选择了低效的算法，那么即使再多的细微调整也没有用。您可以把所有的变量都塞到寄存器中、内联所有的函数，以及展开所有的循环。您的冒泡排序程序仍然不会在任何地方接近随便的一种快速排序实现。

有关这个主题的讨论超出了本书的范围。高效的算法和数据结构已成为热门研究的焦点，许多书籍专门致力于这个主题[AHU74，Knu73]。

正像其重要性一样，高效的算法也是必要的，但它们不是保证应用程序性能的充分条件。我们的经验是：软件的性能是几个必要因素综合的结果，无法选出任何一个来作为最重要的因素。人类的心脏无疑是一个重要的器官，但是如果要生存的话，还需其他器官。专注于算法并把其作为惟一的性能问题是一种明显的错误。

14.4 缓式计算

许多性能优化通过更为高效地执行计算来获得速度。优化过程中性能的收益不仅通过加快计算速度来获得，还通过完全消除计算来获得。

缓式计算问题对于大型代码更有可能出现。比如说，如果编写一个推销员旅行路线

问题(Traveling Salesman Problem,TSP)的解决方案,问题描述占用了一节,其实现少于100行代码。该TSP问题描述和实现完全可以被一个开发者所理解。这是典型的小型工程。在这种情况下,发现完全无用且易于避免的昂贵计算的可能性很小。对这种小规模代码进行优化通常从根本上就很困难[Ben82]。

当转移到大规模的编程工程上时,经常会发现自己陷入了沼泽之中。大规模的工程可能包含几十万行代码。问题描述不再是占用一小段,比如编辑HTTP协议规范很可能需要几百页。问题规范通常不会再是活动目标。它在不同的版本之间进行修改,要受到市场趋势及客户需求的影响。

新增的复杂性会导致这样的情况:一个组织中没有哪个人能对整个实现情况有深入全面的理解。开发团体被分成较小的团队,这些团队专注于整个解决方案的特定部分。

大型工程实现的失误导致大量明显的性能方面的编码和设计错误。只有在这种复杂的环境中延迟求值才能表现出调整武器的作用。

getpeernam ( )

TCP/IP套接字连接是远程套接字和本地套接字之间的通信管道。getpeername ( )用于计算远程套接字的套接字IP地址。然后您的应用程序可以把它转换成点分十进制符号,像9.37.37.205一样。在Web服务器中,由每次请求产生的典型日志记录包含远程客户端的IP地址。这是对商业Web站点非常有用的信息,这些站点希望知道浏览站点的人以及他们正在查看的内容。因此,假设您是HTTP日志团队的成员,您的任务是往日志中添加客户端的IP地址。您需要调用getpeername ( ),但是到底应该把它插入到哪里呢?您不知道整个服务器实现的全部细节,但是您猜想客户端的IP地址很有可能到达应用程序的其他部分。在每次请求的开始,就在accept ( )调用建立新的套接字连接之后调用getpeername ( ),这听起来是个好主意。如果服务器的任何部分在任何时候需要客户端的IP地址,则它将已经成为可用的了。

类似的问题存在于getsockname ( )中。与getpeername ( )类似,getsockname ( )计算本地套接字的IP地址。我们为什么会需要本地的套接字IP地址呢?其原因在于这样做可以在一台单独的物理服务器上运行多个Web站点。当您在同一台物理服务器上运行www.lotus.com和www.tivoli.com时,如何把这些请求路由到正确的站点呢?您如何知道一次对index.html的请求是针对于Lotus站点还是针对于Tivoli站点的呢?把它们分开的方式之一是在不同的IP地址上运行各个站点,然后可以根据请求所到达的IP地址把Web站点区分开。这就是getsockname ( )出现的原因。它告诉我们与本地套接字相关联的IP地址。

getpeername ( )和getsockname ( )是昂贵的系统调用。此外,如果您花些时间来研

究一下内部的服务器设计，将会发现这些调用并不总是必须的。如果系统管理员选择了关闭日志，那么 getpeername ( )将不再需要。如果服务器只运行惟一的 Web 站点，那么可能也不再需要调用 getsockname ( )。如果选择把这些调用放在每次请求的开始，那么您将在所有的请求上遭受性能损失，而不管它们是否真正需要这些值。

为您最后可能不需要的计算遭受性能损失是很糟的。这在复杂的代码中经常发生。昂贵的计算应该只在需要的情况下执行。这通常意味着把计算延迟到真正需要它们的时候，因此称为延迟计算。除了我们在这里讨论的情况之外，还有大量其他有关延迟计算的例子[Mey96，ES90]。

现在说明这种想法。假设有类 X，它包含一个指向 Z 型数据的数据成员，Z 的计算非常昂贵：

```
class X {
public:
    X ( );
    ~X ( );

    Z     * get_veryExpensive ( );
    void set_veryExpensive ( );
private:
    Z   * veryExpensive;
    ...
}
```

如果数据成员 veryExpensive 对 X 对象的使用是必须的，那么可能需要在 X 对象的构造函数中对它进行计算。然而，我们现在是谈论延迟计算，所以假设这里不是这种情况。这就是说即使在 veryExpensive 不可用的情况下，X 对象的某些操作仍然很有意义。例如，即使我们不知道一个套接字的 IP 地址，该套接字仍然是有用的。我们仍然能够发送和接收数据。在这种情况下，X 的构造函数应该把 veryExpensive 设置成某种默认值，该默认值说明它还没有被计算：

```
X::X ( )
{
    veryExpensive = 0;  // Lazy evaluation.
    ...
}
```

如果您在任何时候需要 veryExpensive 的值，那么可以调用如下方法：

```
Z * get_veryExpensive ( )
{
    if ( 0 = = veryExpensive ) {
        set_veryExpensive ( );  // Compute it.
        }

    return veryExpensive;
}
```

get_veryExpensive ( )方法首先进行检查,以便弄清楚要查询的值是否已经计算过。如果还没有,那么就执行该计算。在任何情况下,都会得到要查询的值。这种实现确保把这种昂贵的计算延迟到您明确地需要它时。现在,如果还远远没到使用它的时候就要去查询它,那么您自己决定好了。

让我们研究一个有关套接字地址的更为具体的例子。

```
class StreamSocket {
public:

    StreamSocket ( int listenSocket );
    ...
    struct sockaddr * get_peername ( );
    void            set_peername ( );
private:
    int               sockfd;         // Socket descriptor
    struct sockaddr   remoteSockAddr;
    struct sockaddr * peername;       // Will point to remoteSockAddr
    ...
};
```

构造函数将把对等名称设置成默认值以表明它的不可用:

```
StreamSocket::StreamSocket ( int listenSock )
{
    ...
    peername = 0;  // Not computed, yet.
}
```

如果需要,get_peername ( )将对 peername 进行计算:

```
int StreamSocket::get_peername ( )
```

```
{
    if ( 0 == peername ) {
        set_peername ( );
        }
        return peername;

}
```

尽管网络编程不是我们的主题，但仍需简单介绍 getpeername ( )。实际上，getpeername ( )调用可能根本就不需要。即使需要远程套接字的套接字地址，也可避免该调用。在服务器端，必须调用 accept(int sockfd，struct sockaddr * remoteAddr，int addrlen)来建立连接。accept 调用计算远程套接字的 IP 地址并把它存储在第二个参数 struct sockaddr * remoteAddr 中。

```
StreamSocket::StreamSocket ( int listenSock )
{
    int addrLen;

    accept ( listenSock, &remoteSockAddr, &addLen );
    peername = &remoteSockAddr;          // Now it's available.
    ...
}
```

由于必须执行 accept ( )来创建新的连接，所以同样可以作为 accept ( )调用的无偿副产品来正确地得到对等名称。

14.5 无用计算

您在实践中可能遇到这样一种编程习惯，那就是自动地用零填充大型数据结构。可以通过调用 calloc ( )分配用零填充的内存块来实现，或者也可以通过调用 memset (void * block, 0, int blockLen)自己来实现。

在 Web 服务器上，我们使用缓冲的套接字对象来存储进来的请求：

```
class BufferedStreamSocket {
private:
    int     sockfd;              // Socket descriptor
    char    inputBuffer[4096];
    ...
```

```
};
```

BufferedStreamSocket 的构造函数自动地用零填充输入缓冲区：

```
BufferedStreamSocket::BufferedStreamSocket(...)
{
    memset( buffer,0,4096 );
    ...
}
```

我们一般不反对 memset ( )调用，它有自己的地位，只有在一无所获时我们才反对它。

在对源代码进行详细的检查之后，我们发现没有在任何地方用到用零填充输入缓存区这种操作。我们从来没有假定它，也从来没有使用它。很显然，缓冲区"只是在某种情况下"才执行 memset ( )。在从套接字读取数据时，数据被复制到用户提供的缓冲区中，同时会获得读取字节的数量，这就是您所需要的。读取字节的数量告诉您数据缓冲区的逻辑结束位置。这不是以 null 结束的字符串。由于我们从不使用它，所以构造函数调用 memset ( )纯粹是一种时间上的浪费。

无用计算的例子通常看起来有点无聊，您可能认为自己从不会做这样的事情。然而在实践中，大量场合都会遇到这种计算。在复杂的、大规模的代码中更有可能出现这种情况。

14.6 失效代码

这也是一个属于那类听起来无聊但在实践中经常发生的提示。我们正在谈论的是那些不再用于某种目的但仍保留在执行路径中的代码。这在学校的编程作业或小型程序中不太可能发生，而是发生在大型的编程任务中。驱动软件实现的需求是一种活动的目标，在不同的版本之间它都保持着发展的趋势。可能添加新功能和删除对旧功能的支持。实现本身随着缺陷的解决和提高而持续变化。这种需求和实现的持续变化导致了无生命的泡沫(从未执行)和失效(执行但不需要)代码。让我们回到 HTTP 服务器的讨论中来，以此作为一个具体的例子。

最初的 Web 服务器产品既支持 HTTP 协议，又支持 Secure-HTTP(S-HTTP)协议。此后不久，SSL 便作为 S-HTTP 的替代产品出现。有一段时间我们支持所有这三种协议。随后，SSL 大力发展并把 S-HTTP 抛进了被人遗忘的角落。自然，我们服从了市场趋向并删除了对 S-HTTP 的支持。

在WWW狂热的早些时候，产品的发布周期从1～2年减少到了2～3个月。因此，即使弃用了对S-HTTP的支持，也没有人自寻烦恼去清除源代码。S-HTTP代码到处都是，清除它的所有痕迹是一种容易产生错误、令人厌烦而耗时的工作。源于S-HTTP的失效代码在几个方面产生了性能上的代价。首先，有许多数据成员实例被S-HTTP带到各种对象里面。这些成员被初始化和清除，由于这些计算的结果在本质上是失效的，所以这些计算耗费了不必要的CPU时钟周期。另外，有大量用于把S-HTTP从HTTP中分开的if then else选择点：

```
if (/* HTTP */) {
    ...
    }
else { /* S-HTTP */
    ...
    }
```

通过增加时钟周期以及向处理器指令管道引入泡沫般无用的指令，这些无用的分支点损害了性能。

其他由失效代码造成的微小损害是增加了可执行程序的大小、内存占用以及源代码长度。包含失效和无用代码的源代码难以理解、维护和扩展。由于所有这些原因，我们强烈建议把删除无用代码和失效代码作为一项例行的源代码维护工作。

14.7 要　点

- 在软件性能和灵活性之间存在一种基本的拉力。对于在80%的时间内执行的20%的软件部分，性能通常在损失灵活性的基础上得以提高。
- 像出现在微小的代码细节中一样，缓存也会出现在整个程序设计中。通过简单地储存以前计算的结果，通常可以避免较大的计算缺陷。
- 使用高效的算法和数据结构是高效软件的必要但不充分条件。
- 有些计算可能只在某些情况下需要。这些计算应该放到它们必须被使用的执行路径中。如果过早地执行某种计算，其结果可能并没有被用到。
- 大规模的软件通常有混乱的趋势。混乱的软件所产生的一种副作用是失效代码的执行：这些代码从前用于某种目的，但现在已不再需要。定期地清除失效代码和其他无用代码将如同全面的软件保健一样能促进软件性能。

第 15 章

可伸缩性

提高 C++ 性能的编程技术

面对提高 C++ 应用程序运行速度这样的任务，您通常有以下选择：

- 调整您的应用程序。通过优化代码来缩短路径长度。到现在为止，这一直是我们讨论的焦点，也就是提高单处理器（单个 CPU）计算机上单个线程的执行速度。
- 升级处理器的速度。更快一些的 CPU 将使那些受限于 CPU 的工作得以更快地执行。
- 添加处理器。多处理器计算机包括多个 CPU。理论上，多个 CPU 将胜过单个 CPU，每添加一个都会增加相同的速度。这就是可伸缩性涉及的全部内容。

应用程序代码面临的可伸缩性挑战就是要与增加的处理能力保持一致。当您把应用从单处理器移植到两路多处理器之后，如果能将执行速度翻倍那很不错。如果在一个 4 路多处理器上，您的应用程序执行速度会是以前的 4 倍吗？如果在一个 8 路多处理器上，您的应用程序执行速度会是以前的 8 倍吗？这就是可伸缩性方面的挑战。尽管大多数应用程序不会表现出这种完美的线性比例，但是我们的目标是要尽量地接近这一点。

要理解多处理器上的软件性能问题，就必须对底层体系结构有基本的掌握。由于多处理器是由单处理器发展而来，所以我们要对单处理器计算机体系结构进行一次极其短暂的回顾，并由此开始下一步的研究。主流的单处理器体系结构由单个 CPU 和单个主存模块组成。数据和指令位于主存中。由于 CPU 的速度至少要比主存快一个数量级，所以我们在处理器（=CPU）和主存之间插入了一块附加的（非常）快速内存。这块快速内存也被称为缓存。缓存提供对程序数据和指令更快速的访问。缓存可以被分隔成两个物理单元，一个用于数据，另一个用于指令。我们将忽略这种差别并以一个单独的逻辑实体来引用它。为了取得指令本身，对于每条指令，处理器都至少需要访问一次内存。对于大多数指令，可能需要用于数据的其他内存引用。当处理器需要内存访问时，它首先查看缓存。由于缓存的命中率高达 90%，所以与内存之间的缓慢往返是不频繁的。图 15.1 对整个情况进行了概述。

应用程序是单个进程或是一组协同工作的进程。每个进程由一个或多个线程组成。

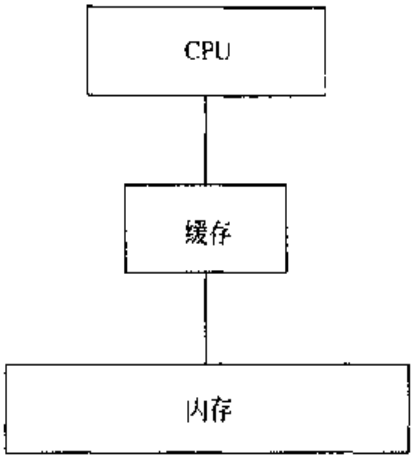


图 15.1 单处理器体系结构

在现代操作系统中，调度实体是线程。操作系统不会执行一个进程，而是执行一个线程。准备执行的线程放在系统的 Run Queue(运行队列)中。该序列头部的线程是下一个即将被执行的线程(如图 15.2 所示)。

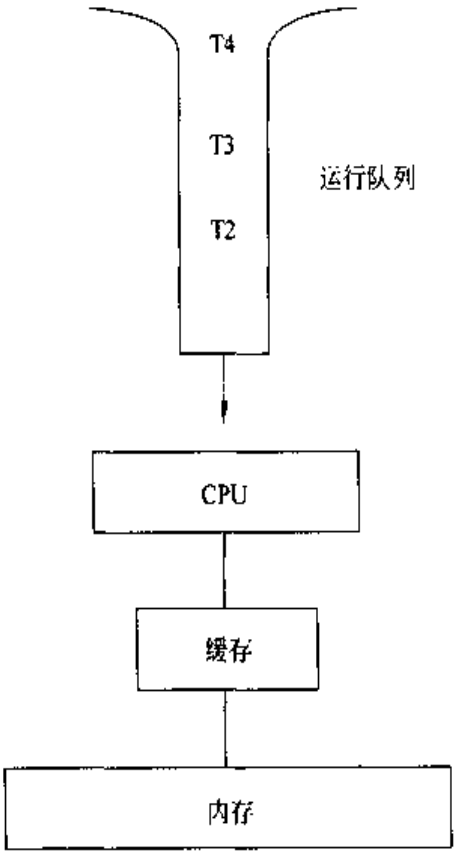


图 15.2 线程是调度实体

现代的单处理器计算机由抢先式多任务操作系统控制。这种操作系统创造一种程序并发执行的假象。从用户的角度来看，似乎是多个程序在同时执行。正如您所知道的那样，实际上并不是这样。在硬件一级，只存在一个 CPU，在任意时刻都只有一个线程在执行。单处理器上的多任务是通过让线程轮流使用 CPU 实现的。

15.1 SMP 体系结构

名词 SMP 本身已经说明问题：

- 它是 MP，也就是多处理器。指系统包括多个同样的 CPU。
- 它是对称的。所有的处理器有相同的系统表现，它们有相同的能力。例如，它们对任何内存位置和任何 I/O 设备都进行同样的访问。
- 其他一切都是惟一的。惟一的内存系统，惟一的操作系统拷贝，惟一的 Run Rueue。

图 15.3 表现了 SMP 的情况。

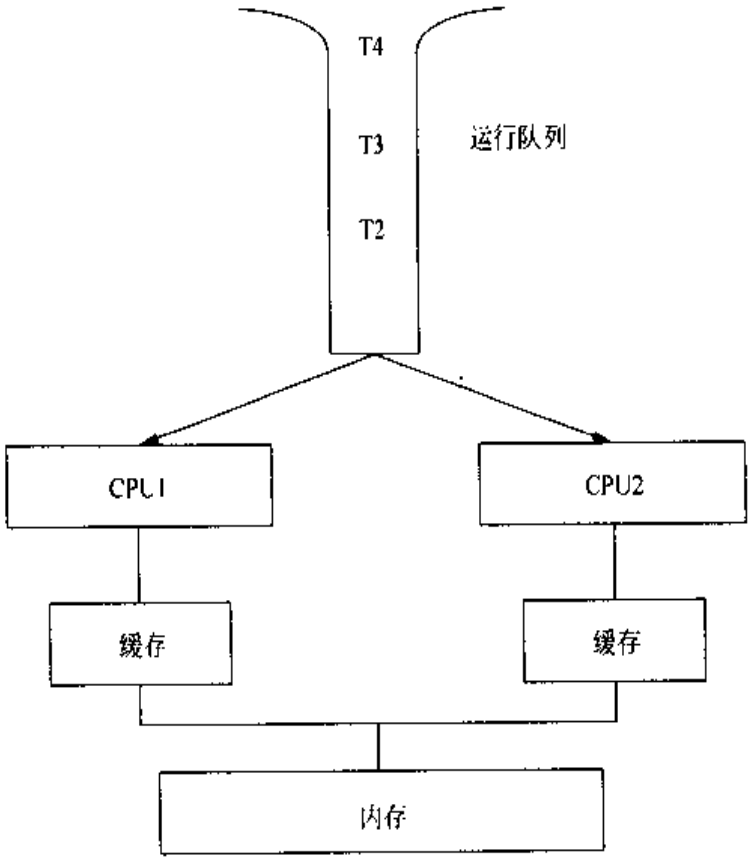


图 15.3 SMP 体系结构

除非应用程序代码非要那么做，否则线程不会与任何特定的CPU产生任何密切关系。同一个线程在某个时间片内可能在CPU1上执行，在下一个时间片内则有可能在CPU2上执行。由图15.3得出的另一个结论是多个线程确实在同时执行。当线程T1在CPU1执行时，线程T2可能在CPU2上执行得很好。由于调度实体是线程，所以没有什么可以阻止线程T1和T2属于同一个进程。在这种情况下，它们很有可能将在一定的规则基础上访问相同的内存地址。这会带给我们另一个问题：只有一条连接处理器和内存系统的总线。总线将成为SMP系统的主要瓶颈，它也是256路SMP计算机系统不常见的原因——总线竞争将带给它们摩擦性中断。

总线竞争的解决方案是采用大型缓存，每个处理器一个。尽管大型缓存将使总线往返变得不频繁，但是这又产生了一个新的问题：如果两个不同的缓存都包含变量x的一份拷贝，并且有一个处理器更新了它的私有拷贝，那么将会怎么样呢？另一个缓存可能会包含一个失效的错误x值，发生这种情况是令人讨厌的。这就是缓存一致性问题。SMP体系结构为此提供了一种硬件解决方案。细节不是十分重要，我们需要知道的是，当变量x更新后，所有其他缓存拷贝和主存拷贝都将得到更新。缓存一致性的硬件解决方案对于应用程序软件是完全不可见的。

SMP体系结构提供了潜在的可伸缩性。很明显，处理器的数量呈现一种简单的上限。在8路SMP上，无法让速度提高8倍以上。实际中的可伸缩性限制可能比这还要紧张。下面我们就来探索实际中的这种限制。

15.2 Amdahl法则

针对应用程序的连续部分将对其潜在可伸缩性进行封顶这个事实，Amdahl法则(Amdahl's Law)对其进行了量化。像矩阵乘法这样的例子是最好的说明，它包含三个阶段：

- 初始化：读取矩阵中的数据值。
- 相乘：让两个矩阵相乘。
- 表示：表示结果矩阵。

让我们进一步假设全部计算用10ms，它们可以分解成以下部分(尽管这些数字完全是虚构的，但有助于说明问题)：

- 初始化：2ms
- 相乘：6ms
- 表示：2ms

这些年来，开发了许多利用多处理器体系结构来加快矩阵相乘阶段速度的聪明算法，

而对初始化和表示阶段却没有这样做。实际上，这两个阶段在计算方面基本上是顺序进行的。理想情况下，无限多的并行处理器在理论上可以把相乘阶段减少为 0ms。但是它们对其他两个阶段根本就不起作用，如图 15.4 所示。

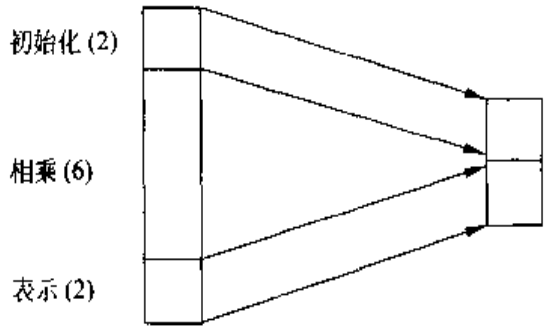


图 15.4 潜在的提速受到限制

无限多的并行处理器实际只能把 10ms 的计算减少到 4ms。可见在任何 SMP 系统上，不管处理器的数量是多少，2.5 倍都是这个应用程序速度提高的上限。

这个矩阵示例展示了一种响应时间限制。同样的思想适用于吞吐量。请考虑一下我们的 Web 服务器的原始设计，这是一个三阶段的管道(如图 15.5 所示)。

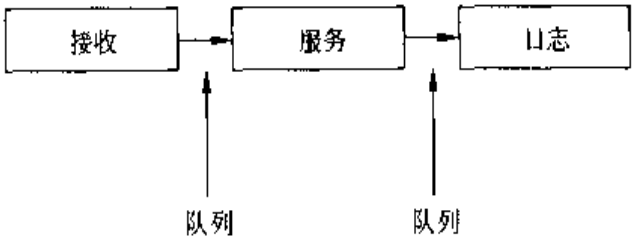


图 15.5 一个 Web 服务器的特定设计

让我们假设整个管道消耗 10ms，如下所示(再说一次，这些数字是虚构的)：

- 接收：2ms
- 服务：6ms
- 日志：2ms

在这 10ms 之内，我们可以在单处理器上以每 s100 次请求的速度提供服务。我们的服务器设计成使用单个线程来执行接收阶段。然后把接收到的连接排成服务阶段的序列。可用线程池消耗掉该序列中的请求，从而完成服务阶段。一旦服务阶段完成，就有日志记录被排成日志阶段的序列。和接收阶段一样，日志阶段也是由单个线程完成。接下来只有服务阶段能够受益于 SMP。由于接收和日志阶段是以单线程执行，所以它们本来就是顺序进行的。

不管并行处理器的数量有多少，我们都只能用一个线程来接收新的连接。由于需要

2ms来建立各个新连接，所以这个线程每 s 至多接收 500 个连接。这对我们的 Web 服务器设计产生了至多 5 倍的比例限制，即使是把我们的 Web 服务器从单处理器移植到 12 路的 SMP 上也是如此。

顺序(单线程)计算是通往可伸缩性道路上的主要障碍。下面几节我们将列举消除顺序计算或者至少是使它最小化的途径。在开始之前，需要简要解释一些术语。

15.3　多线程和同步术语

在本书中，这已不是第一次遇到"同步"这个术语。我们已经使用过诸如同步、串行化、临界区、竞争条件和锁之类的术语*。我们还没有真正地解释过它们，现在是回过头来解释这些术语的时候了。第一步是提出这样的问题：所有这些术语要致力于什么样的问题？

以这样一个简单代码语句为例：

```
x = x + 1;
```

如果有两个线程执行该语句，那么我们希望 x 中存储的最终值是 x+2。任何其他值都是错误的。这条代码语句存在的问题在于它不是原子执行的。实际上它被分解成了一些汇编指令：

```
load x, r5          // load the value of x into register r5
add r5, 1           // add 1 to register 5
store r5, x         // store the value of register r5 into x
```

如果两个线程在几乎相同的时间内执行这段代码，那么它们的指令可能以某种方式交替进行，这将导致把 x+1 作为 x 的新值存储，而不是 x+2。这种线程执行上的不适当交替被称为竞争条件。这种竞争条件的解决方案是确保这段汇编代码以原子方式执行。我们把这个代码块称为临界区。一旦一个线程进入了临界区，那么其他线程要等第一个线程离开后才能进入。在这种情况下，我们称这两个线程是同步的(或者说被串行化了)，同时称该临界区是互斥的。由于 x 由不止一个线程访问，所以把它称为共享数据。临界区通常是围绕着对共享数据的访问而言的。我们使用锁来确保对共享数据的安全访问(以及正确执行)——一种与共享数据相关联的标志(bit 或 int 型)。线程在进入临界区之前必须检查锁的状态。如果锁是关闭的，那么说明当前没有其他线程在临界区内执行。

---

* Java 重新定义了术语"串行化"，并用它表示一些与同步完全不同和无关的事情。它对于我们理解这些术语是根本没有帮助的。

从而，通过授予其访问权限并且改变锁的状态来反映这一点。（硬件保证对锁的测试和设置是原子性的。）如果锁是打开的，那么线程必须等待，直到当前在临界区内执行的线程把锁关闭以后。当临界区被锁保护起来以后，我们通常说它的执行是顺序进行的。多个希望进入临界区的线程必须排队，并且在某一时刻只有一个进入。

没有了这些术语挡道，我们就可以转移到对各种方式的讨论上来，这些方式用来消除顺序执行所固有的可伸缩性瓶颈。

15.4 把一个任务分解成多个子任务

要释放潜在的可伸缩性，应用程序必须把一个计算任务分解成可以并行执行多个子任务。Web 服务器是一个很好的例子。其计算任务是当多个客户端请求到达指定的 HTTP 端口时为它们提供服务。如果您的 Web 服务器是单线程的，那么在某一时刻它只能完成一次请求，所有其他请求必须在队列中等待。要使其具有可伸缩性，我们必须对服务任务进行分解。为队列中全部请求服务的任务可以分解成为单个请求服务的较小任务。通过对较小的任务启动多线程，我们可以实现真正的并行机制和可伸缩性。把 Web 服务分解成几个并行子任务提高了以下几个性能指标：

- 每个单独请求的响应时间
- 服务器的吞吐量（每 s 处理的请求个数）
- 对服务器 CPU 更为充分的利用

Web 服务器要执行频繁的 I/O 操作。如果服务器由单个线程组成，那么在等待 I/O 完成期间不能做任何事情。多线程的服务器将简单地转变成执行其他任务从而使 CPU 保持任务饱满。结果是获得更好的吞吐量和更高的 CPU 利用率。客户头痛的一件事是看到自己的 12 路 SMP 服务器只有 25%的 CPU 利用率。

有一些商业 Web 服务器设计成了单线程的。这些供应商有意忽略了那些通常要求 SMP 服务器的大容量 Web 站点。相反，单线程的服务器着眼于运行在低端单处理器服务器上的小容量 Web 站点。

15.5 缓存共享数据

一旦能跳过对临界区的访问，就能看到可伸缩性方面的收获。这种思想最简单的应用就是通过缓存共享数据，从而在将来的访问中不需要进行锁定和解锁共享资源就能重用它。这实际上是我们对原来的 Web 服务器设计和实现所进行的单方面最大的可伸缩性优化。最初，它不仅没按比例增大，而且还表现出一种分数式的比例— 当我们从单处

理器移植到4路SMP时，吞吐量降低了一半。有些方面发生了严重的错误。

在本章的前面部分已经描述了我们的Web服务器的整体设计。在线程和请求之间我们使用1对1的对应关系。可用线程池消耗掉连接队列中的请求。一旦把一个请求指派给特定的线程，那么在请求处理期间，它将始终限于该线程。在整个请求活动期间，有很多对象属于我们正在处理的请求。我们需要让那些对象对需要它们的例程成为可用的。我们可能已经以函数调用参数的形式把它们传递到了堆栈当中，但是这将强迫我们向几百层深的嵌套调用传递10～15个对象引用。这不仅不优雅，而且将引起函数和过程调用开销。

我们做出了更为优雅的选择，那就是把所有相关的数据包装到一个单独的数据结构中。这里给出它的轮廓：

```
class ThreadSpecificData {
    HTRequest      * reqPtr;          // Attributes of current HTTP request
    StreamSocket   socket;            // Socket used by current HTTP connection
    ...
};
```

ThreadSpecificData对象包含线程在请求活动期间所调用的各种过程所需要的全部数据。现在我们需要让这些数据为特定的线程所私有。pthreads库提供了实现这一目标的工具。pthread_setspecific ( )函数创建调用线程和数据指针之间的关联：

```
void initThread (...)
{
    ...
    ThreadSpecificData * tsd = new ThreadSpecificData;
    pthread_setspecific ( global_key, tsd );
    ...
}
```

当线程需要访问其私有数据时，它使用pthread_getspecific ( )调用来得到指向其私有数据的指针：

```
ThreadSpecificData * HTInitRequest ( StreamSocket sock )
{
    ...
    ThreadSpecificData * tsd = pthread_getspecific ( global_key );
    tsd->socket = sock;
    tsd->reqPtr = new HTRequest;
```

```
    ...
    return tsd;
}
```

处理单个 HTTP 请求需要调用许多函数和过程。大多数调用需要访问存储在 ThreadSpecificData 对象中的信息。因此，我们的代码被几百个 pthread_getspecific ( ) 弄得很杂乱，其中在为单个请求提供服务的过程中，每个线程大约要进行 100 次这样的调用。

在某些地方，以某种方式，pthreads 库维护着线程和它们的私有数据之间的关联。由于线程可能出现和消失，所以这种关联是动态的并且必须用锁保护起来。在对这种关联进行锁定的时候，您希望确保这种关联不会莫名其妙地消失。我们最后得到的是这样一种情况：为不同的请求提供服务的多个线程同时请求相同的锁，每次请求都会发生 100 次。正如您可以想象的那样，线程在等待轮到它们获得热门锁的过程中，以空转的方式消耗了它们的大部分时间。

对锁的激烈竞争在 SMP 体系结构中产生了另一个严重的问题。锁字可能在每个处理器缓存中包含拷贝。由于它经常更新(当锁被获取和释放时)，这将触发一场缓存一致性风暴，其原因是 SMP 硬件设法使某一个缓存之外的所有缓存中的过时拷贝失效。缓存一致性是糟糕的。它们击中了 SMP 体系结构最为脆弱的可伸缩性因素 ---在处理器与内存子系统之间以及各自相互之间进行连接的总线。

为了释放我们的 Web 服务器在可伸缩性方面的潜力，我们不得不避免对 pthread_getspecific( )进行频繁调用。我们必须找到一种传递线程数据的替代方式。这种替代是非常简单的，尽管线程数据被频繁更新，但是在线程的全部生命周期内指向线程数据的指针是不变的。它本质上是一个只读指针，它自动地建议进行缓存。我们通过在请求的开始进行惟一的一次 pthread_gespecific ( )调用来得到该指针，然后把它作为函数(或者过程)调用参数传递到堆栈中。通过这种方法，我们把 100 次 pthread_gespecific ( )调用减少到了 1 次，特别是消除了对其内部锁的竞争。

```
void HTHandleRequest (StreamSocket sock )
{
    ...
    ThreadSpecificData * tsd = HTInitRequest (sock );

    readRequest (tsd);
    mapURLToFilename (tsd);
    authenticate (tsd);
    authorize (tsd);
```

```
    sendData (tsd);
    log (tsd);
    cleanup (tsd);
    ...
}
```

通过把数据指针传递到堆栈，我们消除了对维护线程与数据关联的 pthreads 容器所进行的频繁调用。我们有效地减少了该容器在不同线程之间共享的频率。这种共享情况的减少在大多数可伸缩性优化中是一种不断出现的主题。

15.6 无 共 享

减少共享是好事，但还是没有彻底消除竞争好。与共享资源相反，有时可以通过给每个线程分配自己的私有资源来完全消除共享。让我们回顾前一节：每个 Web 服务器工作者线程(在我们的实现中)都维护一个指向其 ThreadSpecificData 类型私有数据对象的指针：

```
class ThreadSpecificData {
    HTRequest * reqPtr;          // Atttributes of current HTTP request.
    ...
};
```

reqPtr 成员是一个指向 HTRequest 对象的指针，HTRequest 对象包含当前 HTTP 请求的信息。在每个请求的开始，工作者线程都创建一个崭新的 HTRequest 类型的对象：

```
ThreadSpecificData * HTInitRequest (StreamSocket sock )
{
    ...
    tsd->reqPtr = new HTRequest;
    ...
}
```

在请求清除阶段对 HTRequest 对象进行清除：

```
void cleanup (ThreadSpecificData * tsd)
{
    delete tsd->reqPtr;
    ...
}
```

尽管表面上可能看不出来，但是通过调用 new ( )和 delete ( )，这里确实会出现一些线程冲突。使用全局 new ( )和 delete ( )会在工作者线程之间产生竞争。这些调用管理着进程范围内由进程的所有线程所共享的内存。访问由 new ( )和 delete ( )管理的内部数据结构必须使用互斥锁进行保护。它们的实现中有几个部分注定是要顺序进行的。

针对这种情况的解决方案是相当简单的。HTRequest 对象不应该从每次请求的堆中分配。应该把它嵌入属于线程的 ThreadSpecificData 对象中，并重复使用于随后的请求：

```
class ThreadSpecificData {
    HTRequest req;      // An object, not a pointer to one.
    ...
};
```

现在，我们不是分配和释放 HTRequest 对象，而是简单地重用已有的 HTRequest：

```
ThreadSpecificData * HTInitRequest (StreamSocket sock )
{
    ...
    tsc->req.init ( );
    ...
}
```

HTRequest 对象在清除期间被重用：

```
void cleanup (ThreadSpecificData * tsd)
{
    tsd->req.reset ( );
    ...
}
```

现在，由于 HTRequest::init ( )和 HTRequest::reset ( )都是对线程的私有对象工作，所以它们都不包含任何串行化代码。

当无法事先确定需要资源的多少实例时，共享资源池成为一种必要。然而，如果确实知道将只需要固定数量的实例，那么就应该继续进行并使它们成为私有的。在我们的例子中，对于每个请求，所需要的是 HTRequest 的单个实例。通过使 HTRequest 对象成为线程的私有对象，就彻底消除了对资源(这里是指内存管理数据结构)的共享。

15.7 部分共享

以上部分我们遇到了资源共享方面的两个对立的极端：公共共享池和资源的线程私有实例。在这两个极端之间存在着共享的折中方案，即部分共享资源池。

如果每个线程都要求资源的单个实例，那么可以通过使实例成为线程的私有对象轻松地消除竞争。如果无法事先确定所需的实例数量，那么就需要使用由所有线程共享的资源池。这种共享的资源通常会成为线程之间的竞争热点，这种情况会严重地降低性能和可伸缩性。线程在空转状态下花费大量的时钟周期。部分共享提供了一种避免对资源进行激烈竞争的途径。

最初，我们从单个资源池为所有线程服务开始，如图 15.6 所示。我们的目标是通过减少竞争资源的线程数量来减少线程之间的竞争。为了该目标，我们把单个资源池转换成了多个相同的子池。与吸收全部火力的单个线程相比，我们更愿意使用两个池来使竞争减少一半，或者使用四个池使竞争减少至四分之一（如图 15.7 所示）。

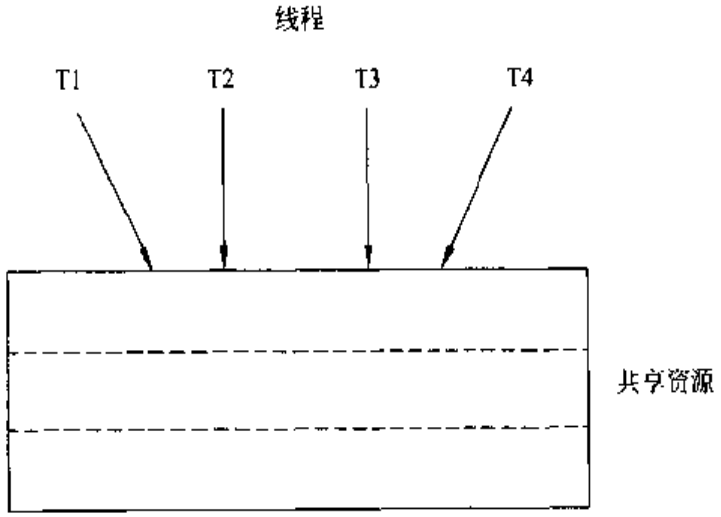


图 15.6 单个共享资源

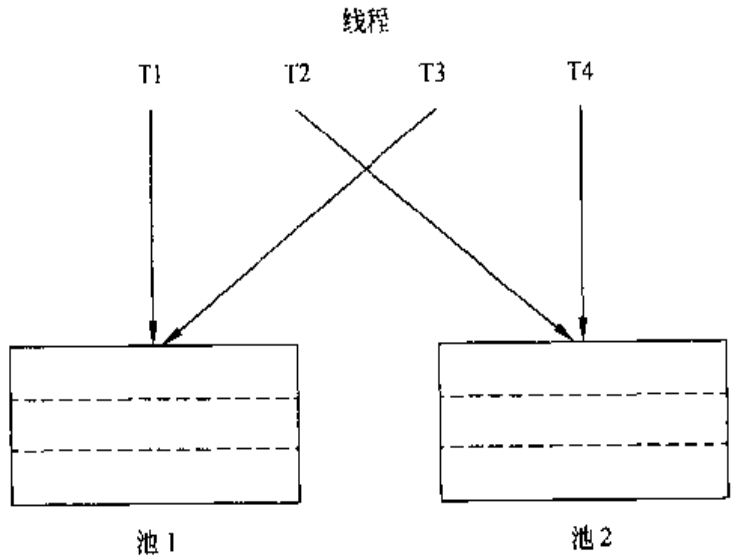


图 15.7 分解单个共享资源

以前全部聚集在单个池上的线程现在被分散到多个子池上。通过生成随机数，通过线程 ID 进行散列，或者通过其他随机快速的分散技巧，可以完成把线程请求分散为各个池的操作。尽管每个子池仍然必须用互斥进行保护，但是线程竞争将比原来水平减少数倍。

随着子池数量的增加，线程之间的竞争相应减少。最终，可能达到一个极端，此时子池的数量大约等于线程的数量。这种情况几乎等同于包含线程私有资源这种没有竞争的方式。

据我们推测，在大多数实际情形中，把单个池分成 2 个、4 个或 8 个子池足以把竞争减少到可以忽略的程度。但有一点要注意，如果超越了正常情形，那么将会出现收益递减。最终的判断标准是经验数据，这些经验数据取决于应用程序。

15.8 锁的粒度

大多数 Web 服务器的合理实现都会为网管提供一些原始统计数据。例如，我们的实现保持对 HTTP 和 HTTPS(SSL)请求数量的跟踪：

```
class HTStats {
    int httpReqs;
    int sslReqs;
    pthread_mutex_t lock;
    ...
};
```

由于可能有多个线程试图并发地操作该统计量，所以必须使用互斥锁对它进行保护：

```
void HTStats::addHttpReq ( )      // Increment the counter for HTTP requests.
{
    pthread_mutex_lock (&lock );
    httpReqs + + ;
    pthread_mutex_unlock (&lock);
}
void HTStats::addSslReq ( )       // Increment the counter for SSL requests. *
{
    pthread_mutex_lock (&lock );
    sslReqs + + ;
```

* 这里原是 HTTPS，但根据上下文，这里应为 SSL。

```
    pthread_mutex_unlock (&lock);
}
```

正如您能够从这段代码中看到的那样，HTStats 类使用单个锁来保护对它的所有计数器进行的操作。通常，把多个无关的资源融合到单个锁的保护之下不是个好主意。这扩大了临界区的范围并在不同的相互独立的线程间产生了冲突。该规则惟一可能的例外是满足以下两个条件的情况：

- 所有的共享资源总是一起被操作。
- 对共享资源的操作中没有任何一个会消耗大量的 CPU 时钟周期。

对于对 HTTP 和 SSL 请求进行计数的情况，尽管共享计数器满足第二个条件（更新是迅速的），但是违背了第一个条件：更新某一个计数器的线程不访问其他计数器。SSL 线程将更新 SSL 计数器，但是它对 HTTP 计数器的状态不感兴趣。就像把单个资源池分解成多个子池一样，我们更愿意使用两个不同的锁来保护两个计数器，这样就可以把对锁的竞争减少一半。

```
class HTStats {
    int httpReqs;
    pthread_mutex_t  lockHttp;

    char smpDmz[CACHE_LINE_SIZE];          // Will explain this one later
    int sslReqs;
    pthread_mutex_t lockSsl;
    ...
};
```

现在，对 HTTP 和 SSL 计数器的更新将使用不同的锁：

```
void HTStats::addHttpReq ( )                // Increment the counter for HTTP requests.
{
    pthread_mutex_lock (&lockHttp);
    httpReqs + + ;
    pthread_mutex_unlock (&lockHttp);
}
void HTStats::addSslReq ( )                 // Increment the counter for SSL requests. *
    pthread_mutex_lock (&lockSsl);
```

* 这里原是 HTTP，但根据上下文，这里应为 SSL。

```
    sslReqs++;
    pthread_mutex_unlock(&lockSsl);
}
```

该实现提高了可伸缩性并防止 SSL 和 HTTP 线程组互相挡路。另一方面，我们可以使用 HTStats 来展示一个例子。在这个例子中，锁的融合实际上是有意义的。假设我们同样跟踪为每种协议服务的累积字节数：

```
class HTStats {
    int httpRes;
    int httpBytes;              // Add this
    pthread_mutex_t lockHttp;
    char smpDmz[CACHE_LINE_SIZE];
    int sslReqs;
    int sslBytes;               // Add this
    pthread_mutex_t lockSsl;
    ...
};
```

更新 HTTP 请求数的 HTTP 线程总是要同样更新发送的字节数。这两个计数器一起更新并且涉及整数相加这样的快速操作。由于这些原因，我们选择把它们融合到同个锁中：

```
void HTStats::updateHttpCounters(int nBytes ) // Increment HTTP counters.
{
    pthread_mutex_lock(&lockHttp);
    httpReqs++;
    httpBytes += nBytes;
    pthread_mutex_unlock(&lockHttp);
}
```

通过分离 SSL 和 HTTP 统计量，减少了对各个锁的竞争。原来只包含一个锁的设计造成了 SSL 和 HTTP 线程之间的人为竞争。另一方面，我们确实选择了把对两个相关 HTTP 统计量的更新融合到一起的策略。否则将不得不为每个计数器浪费一对锁定和解锁调用。

到目前为止，一切都顺利。不过 HTStats 类定义中的 smpDmz 字符数组的作用是什么呢？这是我们的 SMP 非管制区。它必须处理伪共享，下面对此进行讨论。

15.9 伪 共 享

缓存的原子单元是行。一般缓存行可容纳大量字节。128 字节的缓存行是典型的。当 4 字节的整数从主存中加载时，不会孤立地加载它。包含它的整行都要一次性地加载到缓存中去。类似地，当另外的缓存(运行于不同的处理器上)使整数失效时，整个缓存行都要失效。此后，变量的物理内存布局会在 SMP 可伸缩性中扮演一个角色。

以前面讨论的 HTStats 类为例。如果清除了 smpDmz 字符数组，那么最终将会使两个锁互相接近，如以下代码所示：

```
class HTStats {
    int httpRes;
    int httpBytes;
    pthread_mutex_t lockHttp;
    int sslReqs;              // 4 bytes
    int sslBytes;             // 4 bytes
    pthread_mutex_t lockSsl;
    ...
};
```

这两个锁(lockHttp 和 lockSsl)只有 8 个字节的距离。它们很可能最终位于相同的缓存行中。假设我们使用 SMP 系统，在这个系统上，线程 T1 正在处理器 P1 上执行一个 HTTP 请求，线程 T2 正在处理器 P2 上执行一个 SSL 请求。当 T1 获得 HTTP 锁(lockHttp)后，它在 P1 的缓存中修改该锁变量的值。在 P2 上的同一缓存行现在失效。当 T2 试图获取 lockSsl 时，它会得到缓存失败，这是因为该缓存行已经失效。

您可以想象一下这种情况会怎样发展：运行在 P1 和 P2 上的线程连续地互相让对方的包含两个锁的缓存行失效。要记住我们的线程不是在应用程序的共享资源上冲突，而是在缓存行上冲突，这是类的内存布局与缓存一致性之类的硬件体系结构问题共同作用的结果。由于把缓存命中率降低到了 90%以下，所以缓存一致性风暴会严重地降低性能和可伸缩性。

15.10 Thuudering Herd

如果线程在获取繁忙的锁时失败，那么它会以下面两种方式之一继续：

- 在一个死循环中等待，每次循环都试图获取该锁。这只对短时的锁有效。持续长

时间的旋转锁将导致线程在等待共享资源变成可用时的空转过程中消耗 CPU。

- 转入睡眠状态并等待通知。锁被释放以后，会把正在等待的线程唤醒。我们把这种类型的锁叫作简单锁，以便与旋转锁进行区分。

如果大量的线程并发地竞争简单锁，除了其中一个外，其他的都进入睡眠状态。当锁最后变成可用时会怎样呢？这又会出现两种情况：

- 等待队列中只有一个线程被唤醒。通常情况下，将会是等待队列中优先级最高的线程。
- 唤醒所有正在等待的线程。这就会触发 Thundering Herd[Cam91]。

唤醒所有正在等待的线程所存在的问题是：除了一个线程之外，其他所有线程都将失败并转入睡眠。假如线程的上下文切换不是廉价的，那么我们就会遭到性能和可伸缩性方面的巨大损失。Thundering Herd 事件并不神秘，对于许多商业服务器的实现来说它是一种实实在在的威胁。我们自己就曾在设计和实现 Web 服务器时遇到这一情况。

在描述接收－服务—日志三阶段管道时，您可能奇怪为什么我们把接收阶段局限于单线程。这个决定是设法避免 Thundering Herd 的直接结果。

在服务器线程打算接收新的 TCP/IP 连接时，它必须执行 accept ( )调用。accept ( )调用告诉 TCP/IP 调用线程希望接收指定 TCP/IP 端口上的下一个可用连接。

对于每一个端口，TCP/IP 都维护一个进入连接队列和等待接收新连接的线程队列。连接按照到达的顺序移交给接收线程。如果没有线程在等待，则将连接排队。如果没有连接可用，则将等待线程排队。

现在的情况是：大量的线程在连接队列为空时在某端口(比如端口 80)上执行 accept ( )。所有的线程都被添加到等待队列。在内部，所有这些线程都进入等待信号的睡眠状态。一段时间以后，建立了连接。在大多数平台上，此时会向所有等待线程发送信号从而唤醒它们，这会造成一种强烈的冲击，以求获得 CPU(因此称为“Thundering Herd”)。某一个幸运的线程会首先夺取 CPU 并接收指定端口上惟一可用的连接。其他线程将仅仅浪费一次上下文切换，其结果是发现该端口的连接队列再次为空。除了一个线程之外，其他的线程都将返回到等待队列中并进入睡眠状态。如果等待队列包含 100 个线程，那么只有一个会获取新连接，另外 99 个线程将在不完成任何有用工作的情况下浪费一次昂贵的上下文切换。这种 CPU 冲击会导致服务器萎缩并严重损害吞吐量。当吞吐量下降时，系统管理员最可能采取的行动是增加服务器线程的数量，而这只会使问题恶化。当出现 Thundering Herd 时，更多的线程则意味着更差的性能。

我们的 Web 服务器的实现运行于许多不同的平台上，这是我们广泛宣传的口号之一。最初，除了一种平台之外，所有其他平台都受到 accept ( )的 Thundering Herd 问题的不利影响。在这些平台上，我们不得不使用一个线程来接收新连接。使用多个线程执

行 accept ( )比使用单个线程执行 accept ( )表现出了更糟糕的性能。这即是使用过多线程的场景之一。

Thundering Herd 不止限于 TCP/IP。它在任何大量线程竞争过少资源的情况下都有可能发生。最有可能遇到的地方是锁定。锁本身是被多个线程竞争的单个资源。此时的解决方案是研究一下在您的平台上可用的锁定模式特征。

某些平台提供全部两种类型的简单锁——一种类型是唤醒所有正在等待的线程,另一种类型是只唤醒一个线程。如果有选择的余地,那么您应该使用唤醒单线程的锁定模式。

15.11 读/写锁

消除同步问题的另一种方式是放松一个且只有一个线程可以独占访问共享资源的要求。串行化共享数据访问的需要来源于共享数据可能被访问它的某一线程所修改这种事实。为此我们必须只让那些旨在修改共享数据(写)的线程进行独占访问。相反,允许只对读取共享数据(读)感兴趣的线程并发地访问共享数据。

读/写锁是这样一种锁:它们允许多个读操作访问共享数据,而不是等待独占访问。设法获得共享数据读访问的线程将在以下两种情况下获得读访问:

- 没有任何其他线程获得访问。
- 获得访问的线程全部是读操作。

如果写线程获得访问,那么所有的读线程都必须等到写线程离开临界区之后。当且仅当没有任何线程获得对共享资源的访问的情况下写线程才能获得访问。

许多平台提供读/写锁。如果对于特定的平台这类锁不可用,那么您可以从可用的原始同步代码起步创建自己的锁。有关这方面的实现请参阅[NBF96]。

如果所有的线程都要修改共享资源,那么读/写锁将不再有用。实际上,它们将损害性能,这是因为它们的实现本来就比较复杂,从而也就比简单的锁更慢。然而,如果您的共享数据主要用于读,那么通过消除读线程之间的竞争,读写锁将提高可伸缩性。

15.12 要 点

- SMP 是当前的主流多处理器体系结构。它包含多个系统处理器,它们通过惟一的总线与惟一的内存系统相连接。总线是 SMP 体系结构中的薄弱环节。每个处理器包含一个大型缓存意味着总线竞争受到控制。
- Amdahl 法则给出了应用程序的潜在可伸缩性的上限。被串行化的计算部分造成

了对可伸缩性的限制。

实现可伸缩性的技巧是减少(如果可能就消除)串行化代码。下面是一些可以达到该目标的步骤：

- 任务分解：把单一的任务分解成多个子任务，这有益于并发线程并行执行。
- 代码移出：临界区应该包括重要的代码而不是其他任何代码。不直接操作共享资源的代码不应该放在临界区内。
- 利用缓存：有时，通过缓存以前访问的结果，可能消除对临界区的访问。
- 无共享：如果需要少量的、数量固定的资源实例，那么应该避免使用公有资源池。应把那些实例变成线程私有的并重用它们。
- 部分共享：使用两个同样的池使竞争减少一半会更好一些。
- 锁粒度：除非资源总是同时更新，否则不要把资源融合到同一个锁的保护之下。
- 伪共享：不要在类定义中把两个热门锁放在非常接近的地方。您肯定不希望它们共享相同的缓存行并触发缓存一致性风暴。
- Thundering Herd：研究一下锁定调用的特征。当锁被释放时，是唤醒所有正在等待的线程还是只唤醒一个？唤醒所有线程的锁定特征会威胁到应用程序的可伸缩性。
- 系统和库调用：研究一下它们的实现特征。它们中的一些隐含着大量的串行化代码。
- 读/写锁：以读为主的共享数据会受益于这类锁。读/写锁消除了读线程之间的竞争。

第 16 章

系统体系结构相关性

提高 C++ 性能的编程技术

由于我们希望提供尽可能多的实用信息，因而一直在尝试避免向性能问题摇手说“不”。前面的大多数性能讨论是相当独立的。如果您希望知道如何用 C++ 编程，那么这可能有意义。现在我们要进入一个很大程度上来源于非编程概念的领域。硬件相关性的基础是硬件的工作机制。不幸的是，除非您对硬件体系结构有很强的背景，否则其中有些内容可能是难以理解的。

计算机科学已经变成了软件科学，其中，硬件实际功能如何变幻莫测的问题留给了火车司机（工程师）。一般情况下，这种远离深层硬件功能知识的变化还没有损害到程序员。然而，如果没有把高层软件结构转化成低层硬件序列的知识，就无法获得优化的软件性能。对所需信息进行分类不同于您通过阅读书中的某一章来获得所需知识。它需要通过研究 Hennessy 和 Patterson[HP97]以及 Patterson 和 Hennessy[PH96]这样的大部头才能开始理解。

本章包含大量信息，不过大部分都是“注意！”和“谨记”之类的信息。我们曾考虑过全部放弃这些讨论，但又坚信这些讨论会使那些具有必要背景的人受益。至少，对于那些还没有深入钻研过性能等式中硬件一端的人来说，这会为他们打开通向最终性能的大门。

16.1 内存层次

对性能的全部讨论最后必须集中在内存使用及内存使用模式上。通常算法复杂性最明显的表现集中在算法所需要的内存访问数量和类型上。现在的情况是计算很快而访问很慢，这是千真万确的。

在一般的计算机内存层次中，通常至少有 5 个级别。一些内存层次主级别有时会包含子级别。内存层次从最快（最短的访问时间）到最慢（最长的访问时间）依次包括：寄存器，L1（第一级）芯片内缓存，L2（第二级）芯片外缓存，主存（以各种变体出现的半导体动态随机访问存储器：DRAM、SDRAM、RAMBUS、SyncLink 等）和磁盘存储器。一些新处

理器带有二级芯片内缓存，我们在逻辑上认为它们是单个 L1 缓存，其本身由一个较小而极快的存储器和一个较大而很快的存储器组成。大多数硬盘现在带有自己的本地缓存，带有这种硬盘的系统其存储层次可以粗略认为包括两个子系统：一个相对快而小的存储器和一个相对慢而大的存储器。

对内存访问进行简单研究的一种强烈趋势是依据访问时间和时钟周期。这种观念忽略了延迟和带宽之间的交互。访问时间是一个延迟问题：得到数据需要多长时间？总线宽度和区间长度是带宽问题：数据到达时一次可以获得多少数据。这与消防带的概念相似：打开阀门需要多长时间？消防带打开之后每 s 能喷出多少水来？水箱中有多少水？

延迟不会和带宽以同样的比例提高。或者说，我们可以用更短的平均访问时间来移动数据块，但是对内存单个字节的访问速度提高很慢。例如，在过去的 5 年内（大约到 1999 年），处理器的速度从 90MHz 提高到了 500MHz，但是主存的时间只是从 120ns 提高到 50ns（即使是最快的 8nsSDRAM，每次访问也需要 6 个时钟周期）。这说明，处理器速度是以前的 5 倍多，而主存只是以前的 2.5 倍。相反，内存的带宽性能做得较好，能与处理器的性能并驾齐驱。在过去的 5 年内，内存的带宽翻了一倍，当前的 DRAM 区间特性显著地增加了有效内存带宽。磁盘的性能特性类似。访问时间没有跟上，而带宽没有受到同样的影响。

如果熟读一下计算机主存访问特性资料，则会看到一些类似“6-1-1-1-2-1-1-1”的东西。这是说最开始的访问需要 6 个总线周期，接下来的 3 次访问中的每一次需要一个总线周期，随后的一次访问需要 2 个总线周期，然后是 3 次访问，每次一个总线周期。如果考虑到现在一个总线周期通常是 4 个处理器时钟周期，那么这就意味着每次转向主存时，处理器就要空闲 24 个时钟周期，整个缓存行读取将需要大约 60 个指令周期。然而并不总是这样的，有些完善的体系结构支持多个未完成的内存要求，它们在真正需要之前请求数据，但是普遍原则仍然普遍有效——先匆忙然后等待。

就当前的技术来讲（大约是 1999 年），内存层次中每个级别的访问延迟如表 16.1 所示。

表 16.1 内存访问速度

内存级别	延迟
磁盘	10ms
DRAM	50ns
L2 缓存	4～8ns
L1 缓存	2ns
寄存器	2ns

从这些数字似乎可以看出寄存器和缓存具有大致相同的性能，但根本不是这么回事。计算既需要延迟又需要带宽。寄存器组每个时钟周期通常可以处理 3 个操作数。L1 缓存每个时钟周期通常只能处理半个操作数，这是因为每次缓存访问需要一两次寄存器访问来生成访问缓存所使用的有效地址。因此，寄存器组的带宽大约是 L1 缓存的 6 倍。因此虽然 L1 缓存有很好的延迟特性，但它的带宽并不是特别出色。*

16.2 寄存器：内存之王

寄存器是内存层次中延迟时间最短、带宽最大、规格开销最少的实体。像我们所描述过的那样，它们的效率至少是 L1 缓存的两倍。通常情况下，寄存器组至少是 3 的倍数，也就是说，一个寄存器组通常可以在一个时钟周期内读两个操作数并写一个操作数。这意味着，简单的 600MHz、64 位 RISC 体系结构将具有大约每秒 14.4 千兆字节的最大寄存器带宽。这比相当大的 L1 缓存带宽要大 3～6 倍，比 64 位宽的 100MHz 外部 DRAM 接口的有效带宽要大 30 倍。

寄存器在机器代码语句中直接寻址。指令格式中的 5 位可以访问一个操作数（5 位的寄存器说明符可以寻址 32 个寄存器之一）。这意味着一条包含 3 个操作数的 32 位指令具有每条 32 位指令 3 个操作数的有效寄存器带宽，或者说操作数和指令之间是三比一。内存层次中所有的其他级别都建立在虚拟或物理寻址或者由操作系统管理的存储系统（文件系统）索引的基础之上。要获得一个基于内存的操作数，至少需要完整的地址位来加载它，某些地方可能需要加上另外的指令来存储结果值。在使用 32 位指令和 32 位数据地址的 RISC 处理器上，这意味着对单个操作数进行任何有意义的操作都将需要 64 位。因此寄存器与 L1 缓存相比，操作开销要少 6 倍，带宽要大 3～6 倍，延迟要少一倍。寄存器的优点比内存层次中其后的级别要相应更大一些。

可见寄存器确实很棒。让我们就把所有的数据放到寄存器中，然后使用内存的其他层。如果程序少于两打原子变量，该解决方案会工作得很好。然而即使是任意一个尺寸很小的程序也会发现寄存的范围太有限了。许多处理器现在带有 32 个通用寄存器，很快会发布包含 100 多个寄存器的体系结构。然而即使能够设计大的“不可思议”的寄存器（不需考虑物理规则的寄存器），寄存器直接寻址的特性也会使其成为十分困难的大型内存映射媒介。直接寻址为寄存器提供了很快的速度，但是也使它们无法嵌入数组和进行

* 这里讨论的是一种普遍情况，尽管对于大多数硬件是正确的，但是对于某种特定硬件可能不完全正确。尽管超流水线和超标量体系结构以不同的方式偏向于寄存器和缓存的有效速度，但是一般情况下，不管您正在使用哪种处理器，寄存器都会比缓存提供更大的带宽。

动态索引(不需要自行修改代码，在这种情况下是一种强制性能的能力，但是仍然充满了困难)。创建指向基于寄存器变量的指针是极其困难的。变量到寄存器的映射是上下文相关的，也就是说，寄存器 3 总是寄存器 3，不管我们可能处于调用树中的何种深度(这实际上只是在大多数情况下正确，支持重叠寄存器窗口的体系结构不重新映射寄存器)。

寄存器很像原子数据的仓库，但是它们太少了，而且它们的寻址特性也太有局限性了，因此不能把它们作为惟一的内存存储器。但是，它们对合适的临时数据类型是很棒的。让我们探讨一下有哪些合适的临时数据类型，并看一下编译器可以为变量到寄存器映射提供哪些帮助，同时我们也要看一下当编译器不能做出最佳选择时我们能提供什么帮助。

编译器编写者和所有人一样知道良好寄存器映射的重要性。他们尽力向寄存器分配变量，但是有点倾向于保守。他们理解与 C++ 之类的语言相关的别名问题，这些语言允许不受限制的指针操作。指针允许以不可思议的方式给变量起别名。尽管我们知道没有必要再支持这种类型的编程，但是同时意识到，改变编译器方法(对现有代码进行大的改动)在许多环节上被认为是不明智的。

保留字 register 告诉编译器不需把变量放到内存里面，或者更准确地说，不需给变量分配内存地址。这只是一个建议，就像 inline 指示符一样，它有可能被编译器忽略。有的编译器完全忽略所有的寄存器指示符。我们只能假设这类编译器的编写者确信他们的寄存器分配算法能比程序员做出更好的选择，这是一种在某些情况正确，在另一些情况下又很不正确的信念。

我们在前面说过，理想情况下，编译器将自动内联并优化。对于基于寄存器的变量更是如此。对于寄存器分配或者确定什么时候应该让变量基于寄存器，编译器能够比程序员做得更好。如果编译器能够利用配置信息(这些信息为编译器确定方法的主要执行路径提供基础知识)，那么编译器就会对寄存器分配做出优化选择。似乎有这样一种趋势，编译器编写者正在放弃这种方式，而我们却为之向他们喝彩，正如以前所说，我们相信基于配置的优化将为自动化优化过程提供最大的可能性，同时也是把程序员从显式的、低层的优化决策中解脱出来的最为有效的技术。程序员应该考虑类的实现决策，而不是把时间花费在低层的基于寄存器的问题上。不幸的是，许多编译器远没有达到优化的基于寄存器的决策，因此一些程序员的直接协助也就成为很平常的事情。

以下有 3 种方法：a、b 和 c，每个方法都包含一个局部变量 i，它在方法 a 中是基于寄存器的，在方法 b 中不是基于寄存器的，在方法 c 中可能是也可能不是一个好的选择，这要依赖于我们对该方法常用执行路径的了解。

```
int x::a ( int x, int& y)
{
```

```
    int i = -x + y;
    y = i - 10;
    i = x;
    y -= i;

    return i + 10;              // i used 5 times in 5 instructions with
                                // no intervening calls
}

int x::b ( int x, int& y)
{
    int i = -x;
    y = test (x);
    y = -y;
    int j = test (y);
    y = -j + 12;
    count << j << y;

    return i + j;               // i used 2 times in 8 instructions with
                                // 2 intervening calls between its first
                                // and last usage
}

int x::c ( int x, int& y)
{
    int i = -x + 100;
    if ( i < 0 ) {
        y = test(x) + i;
        }
    y =   y;
    int j = 15;
    if ( i <= 0 ) {
        j = test (y) + x;
        }
    y -= -j + 12;
    return i + j;               // i used 5 times in 9 instructions but with
```

```
                    // potentially two intervening calls between
                    // its first and its last usage
}
```

选择是否让一个变量基于寄存器的重要标准是变量将被加载和存储的次数。如果变量是基于寄存器的，那么通常在方法执行调用时要保存变量所在的寄存器，然后在被调用方法返回时将它恢复。合计起来，变量第一次使用到最后一次使用之间每次方法调用需要一次加载和一次保存。如果变量不是基于寄存器的，那么定义它的方法每次执行调用时不需要加载和保存它，但是每次定义它的方法需要它时都需要加载和保存它。这使得计算调用和使用位置进行选择变得相对简单。编译器对此做得相对较好。存在大量的条件调用方法是编译器惟一需要帮助的时候。这些实例使得配置信息十分有用，然而由于大多数编译器没有任何自动的反馈回路，所以程序员有时必须通过指定某变量应该基于寄存器以对编译器提供适当的提示。

正确地让变量基于寄存器可以使某些编译器产生的个别方法在性能上提高一个数量级。在大多数这样的情况下，编译器将识别出这些优化可能性并根据寄存器自行执行，但是检查一下是不会有什么坏处的。像内联选择一样，基于寄存器选择应该在配置的基础上进行。只有位于程序重要路径中的代码才应该考虑寄存器编译指示说明。通过找出基于寄存器的最佳候选，对其使用寄存器编译指示符，重新编译并重新配置可以测量编译相关质量指标。任何执行速度上的提高都说明要么需要程序员显式地给出寄存器分配的选择，要么需要增加编译器提供的优化级别。需要注意的是：应该在启用编译器优化最高功能级别的情况下收集配置数据。未优化代码的配置使编译器看起来像是由那些对性能一无所知的人编写的。大多数编译器在执行有价值的自动优化之前都需要显式地启用这些功能。

16.3 磁盘和内存结构

文件结构是可以由整整一本书来单独讲述的主题，在任何重要细节上深入讨论文件结构都超出了本书的范围。然而，我们将要讨论一些更为重要的原因，这些原因解释了为什么理解文件结构很重要，为什么在数据永久性要求文件存储时以及动态数据尺寸需要使用大量虚拟存储时要使用适当的文件结构。

B+树和 b * 树是与易变有序类实例存储相关联的规范文件结构。对于那些具有顺序访问特征或者不经常修改的数据来说，简单线性文件或索引文件是可行的结构。通常认为这些文件结构更适合数据的永久存储，并认为它们不太适合管理运行时数据。大型数据集的运行时管理需要与系统的虚拟内存机制更为接近的存储机制。

不幸的是，我们遇到过许多把虚拟内存看成“神奇长生不老药”的程序员。他们认为这种机制把无限的内存消耗变成了可能的和可满足的。这种观点是某些可执行程序映像膨胀的部分原因，这种现象非常普遍。在较新的操作系统中，存在通过把永久数据集（文件）映射到系统的虚拟存储而访问永久数据集的能力，有时这种能力加剧了依赖虚拟存储来管理特大运行时数据集的倾向。尽管这种把简单的顺序管理数据映射到随机访问内存模型中操作的方法使得软件易于编写，但是可能会产生不是特别有效和快速的程序。这并不是说文件映射是一种不适当的策略。对于某些数据类型，把文件映射到虚拟存储中会获得显著的性能收益。我们只是对不加分辨地依靠操作系统的虚拟内存存储机制提出警告。

虚拟内存并不深奥。操作系统代码负责维护您的数据在内存中的驻留，而不是由您自己的代码来负责，这不一定会使数据访问变快。磁盘访问大约和系统的虚拟换页系统一样昂贵，这是由于它使用了显式的内存管理代码。

访问磁盘平均需要大约 12～20ms（这是磁盘厂商提供的数字，通常认为这是经过极度优化的，特别是在繁忙的系统上），这共计有 300 万个时钟周期的处理器延迟。典型的磁盘访问包括至少两次上下文转换以及低级设备接口的执行，这总共有至少几千条指令的开销。尽管磁盘访问延迟可能被繁忙系统上的其他活动所掩盖，但是设备接口所消耗的时钟周期不会被掩盖。磁盘访问是昂贵的。任意地依赖系统的虚拟内存系统可能对程序的性能造成非常恶劣的影响。

分配存储空间时，目标总是应该维护尽量多的数据局域性。如果数据集是巨大的，那么这种局域性应该考虑成页局域性。应该对数据进行组织以便需要最少的页访问，这正是以前所述的 b＋树结构存在的目的。

对于有序数据来说（暂时先忽略散列数据），二分查找法通常是最快的查找机制，其查找复杂性是 $\log_2 N$。这意味着只要 30 次比较就可以从一个包含 1 亿个有序元素的数组中把一个元素找出来。然而，如果这 30 次比较中有 20 次出现页面错误，那么真正的效率就不是特别好。需要更多比较次数但是产生较少页面错误的解决方案会更好一些。毕竟，页面错误的计算开销需要几千条指令，延迟需要几万条指令。相反，比较操作一般需要 6 条以下的指令。在页面错误的处理过程中会进行很多次比较。

很多情况下，自然数据局域性能够非常有效地使用系统的虚拟内存能力。数据表现出良好的时间和空间局域性。这意味着程序可以访问非常庞大的数据，但是在某一时刻只使用该数据中的一个小子集。该活动数据集称为程序的工作集，它倾向于只占用程序所用全部内存的一个小比例，经过证明它是相当稳定的（工作集内的平均页面使用可能非常长）。当访问新的页面而不再访问旧的页面时，程序的工作集可以逐渐地把页面调入或调出。程序的调用堆栈准确地说明了这种类型的局域性。相反，当把不同的工作集加载

到内存中去的时候，程序可能出现一系列页面错误，随后是一段很少发生页面错误的时间。从一个较大的处理子系统转移到另一个时会出现这种行为。

有些程序具备良好的数据局域性，而其他一些程序可能展示出对数据的稀疏使用，这将导致频繁的页面错误和不佳的性能。这有时是没有成功地管理好程序的主要类管理存储的结果。我们已经对类的存储管理进行了一般性讨论，现在感兴趣的是如何管理需要庞大存储的类。

一般情况下，程序代码倾向于拥有非常好的局域性，代码存储管理是不必要的。当程序代码的工作集大小成为问题时，就需要根据执行配置来重新组织代码，以便把通常相关的代码分配在相同的一个或多个文件中。这在某些极端情况下会出现问题。值得注意的是，这时是方便地使用 C ++ 的 namespace 能力的时候。非常庞大的程序可以受益于单个编译单元中主要方法在空间上的聚集。由于每个聚集的方法都拥有自己的名称空间标识符，所以 namespaces 使之成为一个相对简单的操作。总之，开始增加对基于 namespace 的分区的依赖，而不是基于文件分配的分区是一个不错的主意。下一代的集成开发环境会废弃文件观念并转移到自动名称空间管理约定上来。（我们热烈欢迎这种转变）

因性能原因而重新组织代码是另一方面的考虑，它可使反馈给编译器的配置提供明显的自动优化。方法的代码不关心它处于可执行模块的什么位置，也没有必要把类的方法放在同一个区域，尽管说在榨取最后一点性能时您确实需要至少维护可维护性的外观。如果您确实把类的实现分散出了文件范围，那么一定要在头文件中注明能够找到每个方法实现的地方，同时要说明为什么您会做这么奇怪的事情。

尽管代码只是偶尔需要位置优化，但是管理数据的存储通常是必要的。尽管面向对象的重用性往往是被夸大了，但是类的存储管理却以混合类和模板重用的方式为它进行了很好的开脱。大量的非常完善的存储管理算法可以简单地加以重用。这些解决方案中有一些是很不错的，它们是包含几千行代码的数据管理机制的通用实现。在已经存在通过重用可以轻松使用的解决方案的情况下，我们在有目的地选择编写自己的 b+树之前一定要认真地考虑一下。

不幸的是，完善的存储结构的易用性也使得对错误解决方案的使用更加容易。对于您计划使用的结构，理解隐含其后的低层概念，或者对您所选择的存储结构在计算效率方面有合理的理由，这都是不错的观点。幸运的是，使用用户定义类及其相关构造函数/析构函数指定机制，可以使给定类对象的基础存储机制对类用户透明。这可以很轻松地从一种存储管理模式转移到另一种。我们建议在开始时使用简单的机制，然后把更为完善的存储机制用于配置的性能度量要求。

例如，您可以从简单的数组开始，或者把 STL vector 作为大型顺序列表的存储机制。如果当前配置在这种实现中没有表现出任何缺陷，那么我们就达到目的。如果配置指明

我们在排序和查找数组(vector)时花费了太多时间,那么可以选择一种散列表访问机制。或者也许我们可以采取一种 b * 树或是索引键方法。这些有完善管理机制的类具有本质上相同的接口。它们的可用性使我们可以轻松地用一种管理实现来替换另一种管理实现。只在配置指示需要时才采用更为完善的管理代码,这使得我们能够在需要时使用复杂的机制,而在其他任何地方使用简单的机制。要记住,复杂性是正确性和可维护性的敌人。在能够解决问题的情况下使用最简单的解决方案。所谓节省就是说最简单的解决方案就是最好的——让您的软件保持节省。

在这次有关认真仔细的全部讨论中,我们没有提到一个最有效的 C++面向硬件的优化:使用更多的内存和更快的处理器!有时我们发现自己旋转优化之轮来设法榨取另外 10%的性能收益,有时这会以很大的代价为基础,而我们没有做另一件显而易见的事情:重新指定软件的最低硬件配置要求。与在 100 个内存位置上设法优化相比,从 200MHz 处理器和 32M 的 RAM 升级到 500MHz 的处理器和 256M 的 RAM 能够把代码的性能提高更多。硬件是相对廉价的;不要为承认您正在解决大型问题以及大型软件解决方案需要访问大型硬件平台而担心。

16.4 缓存影响

缓存不仅提供对以前所访问数据的更快访问,它还提供一个包围着以前所访问数据的小的预取区域。缓存获取和管理着数据行。典型的缓存行包含 32 字节的数据,这些数据按照 32 字节的边界对齐。除了完整的行之外,典型的缓存缺乏管理任何其他数据的能力。这意味着如果一条指令访问位于地址 100 的单个字节,那么缓存将把位于地址 96～127 之间的字节加载到行中。当期望的数据是从内存中获得时,最初对地址 100 的访问很有可能导致缓存失败并导致处理器停止运转几十个时钟周期(或者是,如果数据是从另一个级别的缓存中获取的,那么处理器将停止运转 6 个时钟周期),但是随后对字节 101,102,103,…,127 的访问可能不会有问题,除非访问相邻位置之前已经过去了很长时间。

在编写代码时应该考虑这种自动缓存数据块的操作。请考虑类 lla 的如下两个实现,该类封装了一个链表数组,lla 对象可以根据优先级参数链接到这些链表中。第一种实现使用两个相互独立的数组 first 和 last 的指针来维护优先级列表:

```
class lla
{
public:
    ...
    void insert ( );
```

```
    private:
    ...
    int priority;
    lla * next;

    static lla * first[1024];
    static lla * last[1024];
}

void lla::insert ( )
{
    if (first[priority]) last[priority]->next = this;
    else first[priority] = this;
    last[priority] = this;
};
```

第二种实现使用由 first/last 对组成的数组：

```
class lla {
public:
    ...
    void insert ( )

private:
    ...
    int priority;
    lla * next;

    struct pairs {
        lla * first;
        lla * last;
    };
    struct pairs ptrs[1024];
};

void lla::insert ( )
{
    if (ptrs[priority].first) ptrs[priority].last->next = this;
    else ptrs[priority].first = this;
```

```
    ptrs[priority].last = this;
}
```

第二种实现在操作上看起来可能不太明显，但是它考虑了两种实现的缓存效率。在第一种实现中，对first[priority]的测试不会影响对last[priority]的加载。对first[priority]的第一次访问所产生的缓存失败不会影响last[priority]出现的可能性。在第二种实现中，first和last pairs以相邻的存储字分配。如果在内存中正确地对齐了pairs数组，那么最开始对first[priority]的访问将确保相应的last[priority]出现在缓存中。

保持相关数据在一起的要求不仅出现在缓存方面，对于最初访问可能出现页面错误也是如此。在这个例子中，第一种实现中的first和last数组都相当庞大，在first[priority]上出现的页面错误并不一定导致把last[priority]加载到内存。尽管任何明显层次的结构使用往往会把两个数组全部放入内存，不明显层次的结构使用往往对性能影响微小，但最好是第一次就正确地完成工作。

16.5 缓存颠簸

在多处理器系统中包含缓存所带来的一个值得注意的影响是缓存相关协议对缓存性能的影响。缓存相关协议是内存/缓存控制器所使用的一种机制。内存/缓存控制器维护内存的相关视图，这些内存与许多在其他方面与缓存不相关的内存相对立。这个问题很简单：处理器A希望向内存地址100写入而处理器B希望从内存地址100读取。如果B以前从地址100读取过并且在它的缓存中包含一份拷贝，那么其他读取操作将不会到系统的总线去重新获取它。如果处理器计划快速运转，那么这会是您所期望的工作。然而，如果处理器A在B第一次读该地址和第二次读该地址之间向地址100写入了一个新值，那么如果没有相关协议的话，处理器B将从地址100读到失效的(旧的，不正确的)内容，而不是最新的(新的，正确的)内容。

缓存相关协议的基础是内存属主说明和对读取所进行的同位缓存进入的明确验证。这基本上相当于这样一种协议，该协议允许多个缓存共享单个数据项拷贝，但是在某一时间只有一个缓存对该数据项拥有写权限。如果属主(进行写访问的那个缓存)不是正在更新数据，那么其他缓存可以自己拥有该缓存行的拷贝，但是每当缓存行的属主修改该缓存行时，它就会告诉其他缓存让它们的拷贝失效。通过使其他拷贝失效可强制其他缓存再一次读取该数据之前获取另一份拷贝。这将允许属主缓存向它的同位缓存提供该缓存行惟一的正确值。有时缓存行将刷新内存，以便维护内存系统的全面一致性。

缓存相关协议可能需要缓存间的直接通信。大量的这种通信可能降低系统的性能。这是因为缓存间的通道由系统资源所共享，必须在缓存间串行化这种通信。如果内存总

线由于其他缓存事务而已经处于繁忙状态，那么就会强制缓存进行等待。这种用于缓存更新访问的等待称为缓存颠簸。缓存颠簸通常由那些不为其进程维护任何处理器关系的多处理器系统产生。处理器关系相当于跟踪进程最后一次执行所使用的处理器，并尽可能在下一次执行时运行于同一个处理器上。

让我们考虑 2 路 SMP 系统上关系密切的空闲进程序列的执行结果。进程继续在先前运行它的处理器上运行的概率是 50%。如果在调用和终止之间进程被换出 10 次，那么全部时间都在同一个处理器上运行的概率是千分之一。随着处理器比例的增加，缓存影响就变得更为糟糕。在 16 路 SMP 系统上只有 1/16 的机会在同一个处理器上连续运行某个进程，如果经过 10 次上下文交换，那么这种概率只有万亿分之一。

正如已经讨论过的那样，在缓存中维护一个过程的工作集是获取良好性能的关键，缓存失败会使处理器空闲几十个时钟周期。缓存颠簸是在大型 SMP 系统中获得良好性能所必须处理的防垢产物。某些 SMP 系统已开始为处理器的指派提供关系机制。尽管这可能是非常超前的硬件概念，但是这是在多处理器系统中获得良好性能所必须解决的问题。这纯粹是操作系统问题。尽管对于其重要性有一些不同的看法，但是我们相信把问题解决以后，为一些较高优先级的线程维护处理器关系将会获得显著的性能收益。特别是在下一代的多线程处理器上。如果把处理器关系特性过滤成多线程或多处理器编程接口，那么理解这些 API 所提供能力的实质是很重要的。

16.6 避免跳转

现代的处理器生往对跳转不是特别友好。在那些“幸福的往日”里，处理器指令将在下一条指令出现之前完成。这说明如果处理器计算了跳转的目标或者设置了跳转选择所依赖的条件代码，那么下一条指令可以立即使用这条指令的结果。尽管这不是特别快，但是这有利于简单的处理器体系结构。

几乎所有现代的处理器都已管道化。管道化的处理器把指令分解成几个阶段。简单的管道可能包含 5 个阶段：获取指令、指令解码、获取操作数、执行操作和存储结果。一般情况下，5 阶段的管道意味着在指令把结果存储到寄存器时，下一条指令将使用 ALU 来进行一些计算，随后的指令将从寄存器中获得其操作数，后面的指令将被解码，此后仍然要获取下一条指令。在任何指定的时钟周期内，处理器都处于从事这 5 个协同工作的活动中(5 条顺序指令的执行)。

管道化有利于快速处理：理想情况下每条指令可以在每个时钟周期内完成，处理器的基本时钟周期可能由于与其非管道先辈有关的五个因素之一而减少(尽管这是过于简化的分析，但是这有助于您形成一种概念性的认识)。当管道保待填满时会充分地提高性

能。不幸的是,跳转(特别是条件跳转)往往会延迟管道。管道延迟在指令不能进入管道时发生,指令不能进入的原因是指令的进入要附带一个还没有确定的值。例如,假设您使用如下两条汇编语句:

```
CMP r1, r2      // compare r1 to r2 and set the condition codes
BLT x           // branch if the condition codes indicate that the
                // result was less than zero
```

比较指令可以在时钟周期 100 内进入管道,跳转指令将在时钟周期 101 内进入。理想状况下我们将在时钟周期 102 内加载下一条指令,但是跳转中的条件要求比较指令完成其减法操作并设置条件代码,而这些操作直到时钟周期 104 才能完成。在我们等待确定是否要跳的过程中,这将产生两个时钟周期的延迟。如果不存在跳转,我们可以使用这两个时钟周期去完成一些有用的工作。

尽管这里对管道化的讨论的确过于简单,但是确实说明了管道化处理器中跳转问题的核心部分。完善的处理器使用跳转预测,不顺序执行,与/或跳转延迟切口来减少跳转的代价。相反的是,这类处理器往往也会有较长的管道(操作完成与获取指令之间有更多的阶段),缺乏多线程处理器方法(尽管这牺牲了带宽延迟,但仍然是一项新技术),小的跳转仍将保持难以满意的昂贵代价。

跳转看起来明显较多的地方是参数完整性检查。我们看到过大量函数实例,它们包含 90%的参数检查,10%的计算。不幸的是,大多数这样的参数检查是条件判断。在确定这种条件判断对性能有负面影响的情况下,应该设法把过度保守的编程移出重要的代码路径。我们看到过大量实例,在这些实例中,一个值通过 3 或 4 层函数调用传递,在每层调用中都对它进行同样的范围检查。这种对条件跳转的反复使用大量消耗了时钟周期而没有获得等同的收益。

最快的代码是直线代码:没有条件判断、没有循环、没有调用、没有返回。一般情况下,程序的重要路径看起来越像直线,就执行得越快。请记住:带有大量跳转的短代码序列要比不带跳转的长代码序列执行更长的时间。

16.7 简单计算胜过小分支

正像我们以前所讨论的那样,跳转在性能方面是昂贵的。尽管跳转常常不可避免,但是有时跳转是用于代替计算,这可能是严重的性能失误。以下面的代码序列为例:

```
const int X_MAX = 16;
...
```

```
++x;
if ( x >= X_MAX ) {
    x = 0;
    }
```

这实际上只是个位掩码操作，可以用以下代码序列轻松地完成同样的工作：

```
const int X_MASK = 15;
...
x = (x + 1 ) & X_MASK;
```

第一种情况包含一次加载、一次递增、一次存储、一次条件判断和另外一次存储。即使是在使用短管道的机器上，条件判断也将消耗 3 个时钟周期，因此条件判断解决方案的效率代价大约等同于执行 7 条机器代码指令。没有条件判断的版本包含一次加载，一次递增，可能是一条或两条指令的掩码生成序列(用来把值 15 放入寄存器或是立即字段)，一次按位 AND 操作和一次存储。这在最糟糕的情况下需要 6 条指令，一般情况下只需要 4 条指令，其速度几乎比有条件判断版本快两倍。在带有长管道的机器上计算版本将相应地更快一些。

要注意尽管这些代码序列在本质上是等价的，但又不完全是这样。第一种解决方案比第二种更为灵活。X_MAX 不要求按 2 的乘方对齐，而 X_MASK 则要求。因此，就例子中的值来说，它们的功能相同，尽管更快一些的解决方案比第一种解决方案存在更大的局限性。一般情况下，如果性能是最重要的，那么希望为快速的计算而牺牲某些灵活性，这些快速的计算避免了对跳转的使用。

16.8 线程影响

多重处理和多执行路线是两种关系密切相关而又不同的并发编程类型。尽管两个概念都是出现在较大的环境中，但是我们还是要把讨论局限在它们对单一软件系统的影响上，而不是操作系统环境中多个本质上相互独立的软件系统之间的交互。

多重处理是一个古老的概念。多重处理允许多个大型的、相互独立的进程相互通信。进程通常是存在于操作系统中的调度实体。在历史上，多重处理是系统设计师惟一可用的机制，这些设计师希望能够执行多个伪并发异步活动。InterProcess Communication (IPC)(通常是套接字函数库)曾用来简化进程间通信。fork( )和 exec( )曾用于创建进程，而进程终结(process termination)用来联结进程。进程通过套接字调用、管道、共享内存和信号灯协同，这些办法没一个易于使用，而且每个都特定于操作系统。

多执行路线是较新的概念。线程是子进程，是较大进程的片段，它们可以独立地调

度。用极度简化的方式来说，线程就是多重处理，其中一组轻量级进程可以访问一段共享内存区域。共享内存区域的一组进程中的每个轻量级进程都将成为一个线程，作为一个整体的进程组称为进程或任务。线程是任务内的调度实体。大多数操作系统把线程看作惟一的调度实体，也就是说，不能调度任务，只能调度任务内的线程。大多数线程系统把任务看成是内存管理结构而把线程看成是调度结构。几乎在所有的操作系统文档中都可以找到更多有关任务调度的讨论。

任何来自应用程序观点的多任务讨论都将包括对阻塞的讨论。通常大多数 I/O 请求可以分为两种不同的类型：阻塞和非阻塞，或者说同步和异步。同步 I/O 在允许其进程上下文继续之前要等待所请求的 I/O 完成，也就是说，它要阻塞，等待到完全满足其 I/O 请求为止。异步 I/O 在立即得到满足的情况下执行尽可能多的 I/O 请求事务，并把所取得的进展返回给其进程上下文，但是不管所完成的请求事务是多么少，它都不会阻塞。通常，还有一种中间 I/O 请求能力，这种请求类型在特定的时间内表现得与同步 I/O 一样，而有时它们就变成了异步 I/O。尽管这种同步和异步操作之间的中间层向任务提供了值得注意的操作能力，但是从性能的观点来看，它在其操作方面本质上是一种同步操作。

当线程同步请求共享资源或不可用数据时，它就会阻塞。发出请求的线程随后必须等待系统来满足其请求，通常是要求完成 I/O 操作。在线程等待完成其相关 I/O 操作的时候，处理器会把它交换出去，让其他没有阻塞的线程执行。完成所请求的 I/O 之后，线程再一次变成可调度的，最后将把它交换回处理器。一个被阻塞的、拥有很高优先级的线程通常是一完成 I/O 请求就会恢复处理。

典型的程序是同步单线程，因此，在某一时刻它只能做一件事。一旦发出了不能立即完成的请求，就要阻塞整个程序，等待操作系统完成该请求。相反，一些完善的程序会编写成多个物理和逻辑处理流，它们在逻辑上是并发的。这允许执行程序多个相互独立的方面，不用考虑它们中是否有一个在某个特定时刻不能完全满足其 I/O 请求。例如，对于由 10 个线程组成的多线程程序来说，即使 10 个线程中的 9 个被阻塞，整个程序仍然可以朝前进行。对于依靠异步 I/O 的程序来说，即使其 I/O 请求没有返回全部期望的数据，程序仍然可以继续执行。尽管要更为复杂一些，但是与简单的同步单线程解决方案相比，编写支持并发执行概念的软件能够提供更好的性能，特别是在延迟特性方面。这种多线程的优点在真正并行执行多个操作的多处理器系统上变得更为明显。

尽管多线程是一种非常有价值的功能，在适当的系统中它可以产生巨大的性能收益，但是对线程概念的错误应用可能导致明显的性能问题。多线程意味着上下文切换，而上下文切换可能消耗上千个处理周期。多线程也会频繁需要对于共享内存区域的锁。锁同样可能消耗大量处理周期，而对程序的完成起不到任何推动作用。尽管现在多线程相对

容易实现，但是我们发现要做好是困难的。十分常见的情况是：程序被分解成线程，这些线程相互之间缺乏必要的独立性，因而也就无法使线程高效。我们看到过许多把程序分解成线程的例子。这些程序包含太多的锁和信号，最终结果是串行化线程执行（可以同时运行的线程超过一个是异常而不是正常情况）。综合考虑上下文切换的开销以后，一些这样的程序和同步轮询单一程序运行得一样快，比同步多线程程序运行得要更快一些。理解多线程代价因素的关键是理解上下文切换的性能影响。

16.9 上下文切换

上下文切换是什么？它为什么那么昂贵？上下文切换就是把一个进程（线程）移出处理器然后把另一个进程移进处理器。这包括保存进程和处理器的状态。需要保存进程的状态以便维护进程执行点的准确记录。需要维护处理器的状态以便相关进程继续执行时处理器能够退回到这种状态。处理器状态是进程状态的一部分，但不是全部。

进程上下文是一个操作系统结构，因此是非常依赖于 OS 的。OS 中的一个结构对它进行管理，该结构包含进程状态的有关信息，像已分配内存范围、页映射表、打开文件指针、子进程、处理器状态变量（寄存器、程序计数器、可能的翻译后援缓冲器（TLB））和进程状态变量等，进程状态变量有优先级、属主、调用时间和当前状态（可以运行、被阻塞、正在运行）等。大多数进程状态位于内存中并且只是偶尔被访问，但是进程上下文的处理器部分一直由处理器使用，并且必须把它们放到处理器中用于进程的执行。每当进程换出时，处理器状态中与进程相关的部分必须由处理器转移到内存中，每当进程换入时，必须再把这部分加载到处理器。

上下文切换有三种主要的代价：处理器上下文转移、缓存和 TLB 丢失以及调度开销。让我们逐个进行讨论。处理器上下文是处理器状态中特定于进程的部分，通常包括整个寄存器组，如果程序计数器和堆栈指针没有正确地包括在寄存器组中，那么还应包括它们，还有任何其他状态标志（通常包括在一个或两个处理器状态字中）、页表指针、有效地址范围索引，还有可能包括一部分 TLB（这是用来缓存页表项的一个结构）。页表用来管理构成进程线性虚拟地址空间的各页。保存进程状态通常要求向内存写入 20 到 50 个字的数据。这是上下文切换的直接代价，在某些情况下是最廉价的部分，其代价大约只需耗费 100 个时钟周期。但是要记住，保存一个上下文至少要 100 个时钟周期，然后至少需要另外 100 个时钟周期来加载另一个上下文。对于那些包含更多寄存器和/或更多处理器来存放进程上下文的机器来说，将相应地需要更多的上下文切换开销。

上下文切换和缓存之间的交互对程序性能的影响常常是最为有害的。结构和缓存类型将对系统性能的这种特性产生显著的影响。缓存可以根据访问它们所使用的地址类型

分成两类。虚拟编址缓存使用来自进程虚拟地址空间的地址访问。虚拟地址与进程无关。这就是说，系统中的每个进程都有可能使用虚拟地址 100，但是虚拟地址 100 的物理定位将可能是几百个不同的物理地址。纯粹依赖于虚拟地址的缓存在每次上下文切换时都需刷新/使失效。这将意味着每次上下文切换之后，进程都需要重建自己的缓存上下文。

假设进程的工作集由 200 个缓存行组成。这意味着上下文切换之后从头重建该工作集将需要 200 次缓存失败。以每次失败 12 个时钟周期计算，这相当于另外 1 200 个时钟周期的上下文切换开销。这可能是依靠虚拟编址缓存的系统很少的原因之一。有一些机制能够减少这些缓存类型的影响。例如，可以把进程标识字段添加到缓存行的地址标志中。这将允许同一个虚拟地址的多个实例出现在缓存中(每个都将包含不同的进程 ID)，从而将消除上下文切换时对刷新缓存的需求。这也允许缓存高效地处理多个进程，从而减少了冷启动缓存(使用逻辑上为空的缓存)的损失。

缓存的另一种类型是物理编址缓存，本质上与带有 ID 标志的虚拟缓存相同。区别是这种缓存不是依靠标志中的 ID/地址对，而是强制系统在向缓存提交之前把虚拟地址转换成物理地址。事实上，由于这种机制使用页映射，所以地址的低 9～14 位可以立即提交，只有地址中其余的高位才真正需要转换。从缓存延迟方面来说，这原是很简单的。物理缓存也使得把缓存行移出缓存更为容易。它们的内存地址在缓存行的标志中指定，并且独立于进程相关的页映射。大多数处理器使用物理映射缓存。这使上下文切换透明于缓存，也就是说，上下文切换不需要使缓存行失效。物理映射缓存比虚拟映射缓存具有更好的上下文切换特性，但是即使是物理映射缓存，也存在着容量问题。缓存只能容纳那么多的数据。以包含 32K 字节 L1 缓存的处理器为例，其中缓存行的大小是 32 字节。这样的缓存包含 1 024 个缓存行。如果包含这种缓存的机器对由 10 个进程组成的进程组使用多重处理，那么当进程换入处理器时，对于这些进程中的每一个，与其相关的处理器驻留特性将决定缓存所包含已有数据的数量。

假设进程 A 被换出而进程 B 被换入，而 B 只执行了很短时间后就被换出，A 又被换了回来。在这种情况下，可以合理地预测那 1 024 个缓存行将仍然包含大量 A 的上下文。相反，假设 A 由于页面错误而被换出，那么在 A 重新成为可运行进程之前可能需要上百万个时钟周期。这时候另外 9 个进程可能将使用全部缓存。当 A 再会到处理器时，缓存中不大可能还驻留着 A 的上下文。L2 缓存大小日前持续增大的显著原因就是试图构造足够大的缓存，以便为处理缓存上下文提供一点优越性。

在大多数现代处理器中，数据和指令缓存只是缓存的一种形式。翻译后援缓冲器(TLB)是另一种类型的缓存，它用于进程的页映射表。TLB 的大小非常不确定，有时小至 8 个项，有时会超过几百个项。TLB 像寄存器组一样是系统相关的。有些 TLB 标记

了进程标识并且能够超越上下文切换，另一些则没有，在上下文切换时需要使它们失效。TLB 的命中率往往是很高的，单个 TLB 项可以映射大量的数据。相反，TLB 失败可能是非常昂贵的——比缓存失败要昂贵 2～5 倍。TLB 的冷启动是上下文切换性能代价的另一个重要方面。动态恢复一打 TLB 项的代价可能很容易达到 500 个时钟周期的界限。

最后一个主要的上下文切换代价是调度程序的执行。调度程序通常在大多数中断的最后阶段执行。调度程序将决定是否继续运行被中断的进程，或者是否把一个不同的进程加载回处理器。调度程序是操作系统核心功能的一部分。事实上，微内核类的操作系统所包含的内核只比调度程序大一点。尽管调度是相对简单的，并且不需要消耗大量的操作周期，但是也存在复杂性，如进程老化和动态优先级指派等，它们会消耗大约 50～500 个时钟周期。

综上所述，第一，上下文切换的直接代价（处理器状态转移和调度程序执行）消耗大约 250～1 000 个时钟周期。第二，上下文切换代价（缓存和 TLB 失败）可能消耗另外几千个时钟周期。这使得上下文切换的代价适度，而且，如果没有特别的原因，则对于高性能的程序不是十分必要的事。

16.10 内核交叉

对于同步多线程来说，异步轮询可能是一个合理的选择。尽管不是一种简单的软件技术，但是在一些系统上，它能够提供显著的性能收益。这个问题围绕着内核交叉的代价展开，而内核交叉与上下文切换相关。我们已经讨论了上下文切换，因此要深入探讨内核交叉概念。

当程序显式地调用需要内核权限来执行的服务例程时，就会发生内核交叉。这时，会发生两件事中的一件：胖系统调用或瘦系统调用，这取决于操作系统所使用的机制和所请求服务的类型。

胖系统调用就是在最糟糕的情况下，把全部上下文切换到内核。请求服务的进程被换出而内核服务例程被换入。调度程序通常在服务例程确定了下一步要运行的进程后执行。如果服务请求得到了满足，发出请求的进程仍然是可运行的，进程没有消耗掉它的全部执行量程（进程可以连续占用处理器的最大时间量），没有其他优先级更高的进程在等待，那么请求服务的进程将被换回到处理器。因此，为胖系统调用服务几乎需要两次完全的上下文切换（调度程序只运行一次）。

有一些内核服务请求可以在不执行完全上下文切换的情况下得到满足。瘦系统调用只需要保存一些寄存器并改变处理优先级和系统级别。这样的调用可以在对性能影响相对较小以及向调用例程返回一些值的情况下完成，这些返回值看起来比一个非常昂贵的

方法调用稍微多一些。处理器厂商逐步加深了对内核交叉代价的认识，并开始把专用的内核入口和出口能力构造在硬件内。这些系统可以使用几百个时钟周期进出内核，这比胖内核交叉的开销大约减少一个数量级。

在使用高效内核入口的系统上，一打异步操作消耗的时间可能少于一个同步操作。这使多线程方法的采用（基于单个进程内部管理的I/O向量）可能成为系统管理的多线程的一种有价值的替代。轮询（异步读写操作）重要的I/O向量使单个进程能够避免在系统管理的多线程内进行频繁的锁定和串行化，同时避免线程间的上下文切换。如果对I/O向量的快速读取没有产生可处理的数据，那么“选择”同步形式的I/O调用可能使进程进入睡眠状态，直到其他数据到达为止。相反，如果对I/O向量轮询了12次而状态没有任何改变，那么上下文切换可能更快一些。

16.11 线程选择

有3种主要的线程方法：单个、小型和大型。单线程最简单的形式就是只包含一个控制线程的典型程序。然而，在更为复杂的形式中，单线程程序可能使用异步I/O来完成逻辑上并发和独立的操作。单线程异步模式依靠程序内部的结构来监视程序各部分的状态，同时它也依靠轮询来测试这些组成部分内的I/O可用性。使用异步I/O可以确保：第一，没有子任务会因为一些I/O请求没有立即得到满足而阻塞整个任务；第二，程序的控制元素能够确定何时放弃对处理器的控制以及何时继续处理。这样的系统往往围绕一个中心控制循环来构造，该循环串行执行各个子任务。尽管这往往是一种非常复杂的方法，但是在一些环境中，这种方法能够比等价的多线程提供更好的性能。同时如果使用不当的话，它会埋头子一种没有结果的轮询循环而不能取得任务进展。基于这种异步模式构造的软件允许逻辑上的多线程结构存在于不支持多线程的操作系统的上层。通过消除对一些特性（这些特性可能产生分歧、平台依赖性和线程特性）的依赖，这种模式也可以使多平台间的移植问题变得轻松。

大型线程是一种把处理请求作为处理实体的机制。可以认为每个重要的对象方法都是一个线程。在创建重要对象时向其分配自己的线程，该线程的惟一目的是和对象一起走完从生到灭的过程。尽管这种机制可能非常有效，但可能受累于频繁的上下文切换。这种方法在概念上可能最简单，它把核心对象当作半独立的同步程序。不幸的是，如果线程对象依赖太多的共享资源，那么这种半独立性有时可能成为一种幻想。串行化信号需要阻止对临界代码区进行并发访问，这可能导致线程在为锁定、唤醒风暴（当资源变得可用时，系统对一组线程的唤醒，此时除了一个线程之外，所有其他刚被唤醒的线程在消耗了一次上下文切换后重新进入睡眠状态）或者这两种情况而空转的过程中白白消耗时钟

周期。

小型线程是一种把单线程元素和大型线程元素混合使用的机制。与完全完成单个处理请求相比，小型线程往往更集中于完成子任务。在小型线程环境中，对象在其从生到灭的过程中往往会从一个线程转移到另一个线程。尽管小型多线程往往要比大型多线程更为复杂，但是它受累于线程串行化问题的可能性要小一些，它减少了与大型线程有关的上下文切换开销，一般能比单线提供更好的可伸缩性。

以一个简单的程序为例，它从分解成3个线程开始，一个用来管理输入，一个实现程序的主要功能，最后一个管理输出。假设这些线程通过管道进行通信。尽管对于程序执行来说这是一种十分有名望的模式，但是在错误的条件下它可能是一种昂贵的模式。假设该程序主要对基于字符的输入感兴趣，输入线程收集字符，完成一些简单的输入过滤和编辑，然后把它所接收到的字符传递给中心功能线程。如果输入速度很慢，那么就意味着在每个字符到达时该程序都要在输入线程和中心处理线程之间执行一次上下文切换。上下文切换可能消耗上千条指令。我们必须提出这样的问题：使用单个程序对输入流进行异步轮询是不是要比连续的上下文切换消耗较少的时钟周期？

这个问题没有固定的答案。有些时候上下文切换更为昂贵，而另一些时候异步轮询更为昂贵。针对于特定应用的正确答案与这样几个问题相联系：对于新的输入是否需立即处理？希望的速度是快还是慢？输入线程可以完成多少输入处理？应该运行在哪种硬件上？使用错误的线程模式可能严重地影响系统性能，当然，选择正确的模式可以显著地提高性能。不幸的是，作为后期实现性能优化的一部分来修改程序的线程模式不是件容易的事。

线程决策最重要的方面可能是将要支持程序运行的硬件。您是在为桌面系统（在这种系统中，响应延迟可能是最为重要的性能要求）编写程序吗？或者您是在编写服务器软件（在这种系统中，延迟不像吞吐量那么重要，同时资源的使用要受到更大的约束）吗？您希望硬件具备一些特定功能以便使上下文切换廉价一点吗？您是定位于多处理器系统吗？

对于要求响应延迟最小的程序来说，同步多线程解决方案要比异步轮询解决方案更好。这是因为异步程序依赖于控制循环，并且当输入到达时，缺乏一种通知它们的机制。这里的主要问题是对设备进行异步轮询是有偿的，要消耗时钟周期。在这方面的工作做得越多，而同时又没发现任何数据，那么程序的性能就会越差。对输入的查找越少，轮询循环消耗的性能就越少，但是程序的响应延迟也就越长。如果延迟不成问题，那么异步轮询解决方案可以提供最好的性能；如果延迟是个问题，那么多线程可能是个正确的方向。

繁忙的服务器通常受到资源限制。服务器可能不欢迎您的程序事先分配1 000个服务线程。相反，服务器逐步变成了多处理器系统。单一的软件不能像其他模式那样把自

已分散到多个处理器上。即使是在程序已经被数据淹没的情况下，这样做也可能导致处理器空闲。

当前正在开发新的处理器体系结构系列，这些体系结构将显著地改善多线程的效率代价。多线程化的体系结构有能力完成基本上是零开销的上下文切换。通过在异步模式中自己来管理逻辑上的同步线程模式，从而来管理逻辑上的同步线程模式，这是复杂和昂贵的。从长远观点来看，认可这种复杂和昂贵是轻率的。从短期观点来看，在制定重大的设计决策时，必须考虑是使用同步模式还是使用异步模式。令人难过的是，这种类型的决策不能推迟到系统构建好之后，再将其作为系统基于配置优化的一部分来"解决"。改变系统的线程模式不是一件小事，事实上顺着程序的重要路径有条件地修改它是不可能的事。线程模式非常适合在设计阶段早些定型。

16.12 要　　点

- 要使用的内存离处理器越远，访问它需要的时间就越长。离处理器最近的资源即寄存器，寄存器在容量上有限制，但速度极快。对寄存器的优化可能是非常有价值的。
- 虚拟内存不是无偿的。不加选择地依赖系统管理的虚拟结构可能造成非常显著的性能分流，通常是负面的。
- 上下文切换是昂贵的，要避免使用它们。
- 最后，尽管我们知道在内部管理的异步 I/O 具有重要的地位，但是我们也认为处理器体系结构的未来转变将使单一线程方法的优势显著减少。

参考文献

[ALG95] J. Alger. *Secrets of the C ++ Masters*. AP Professional, Chestnut Hill, MA (1995).

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA (1974).

[Ben82] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ(1982).

[BM97] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice-Hall PTR, Englewood Cliffs, NJ (1997).

[BR95] A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, Reading, MA (1995).

[BR97] R. Booth. *Inner Loops*. Addison-Wesley, Reading, MA (1997).

[BW97] J. Beveridge and R. Wiener. *Multithreading Applications in Win32*. Addison-Wesley, Reading, MA (1997).

[Cam91] M. Campbell *et al*. "The Parallelization of UNIX System V Release 4.0," *Proceedings of the Winter* 1991 *USENIX Conference*.

[Car92] T. Cargill. C ++ *Programming Style*. Addison-Wesley, Reading, MA (1992).

[CE95] M. Carroll and M. Ellis. *Designing and Coding Reusable* C ++. Addison-Wesley, Reading, MA (1995).

[CL95] M. P. Cline and G. A. Lomow. C ++ *FAQs: Frequently Asked Questions*. Addison-Wesley, Reading, MA (1995).

[CL99] M. Cline, G. Lomow, and M. Girou. C ++ *FAQs, Second Edition*. Addison-Wesley, Reading, MA, (1999).

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C ++ Reference Manual*. Addison-Wesley, Reading, MA (1990).

[GH95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA (1995).

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, (1996).

[KP74] B. W. Kemighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, NY (1974).

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ (1988).

[Knu97] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume* 1. *Third Edition*. Addison-Wesley, Reading, MA (1997).

[Knu97] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Volume* 2, *Third Edition*. Addison-Wesley, Reading, MA (1998).

[Knu97] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume* 3, *Second Edition*. Addison-Wesley, Reading, MA (1998).

[Lak96] J. Lakos. Large Scale C++ Software Design. Addison-Wesley, Reading, MA (1996).

[Lew1] T. Lewis. "The Next 10 000 Years: Part I," *IEEE Computer*. April 1996.

[Lew2] T. Lewis. "The Next 10 000 Years: Part I," *IEEE Computer*. May 1996.

[Lip91] S. B. Lippman and J. Lajoie. C++ *Primer, Third Edition*. Addison-Wesley, Reading, MA(1998).

[Lip96C] S. B. Lippman, Editor. C++ *Gems*. SIGS Books and Multimedia (1996).

[Lip96I] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, Reading, MA (1996).

[McC93] S. McConnell. *Code Complete*. Microsoft Press, Redmond, WA (1993).

[Mey96] S. Meyers. *More Effective* C++. Addison-Wesley, Reading, MA (1996).

[Mey97] S. Meyers. *Effective C++, Second Edition*. Addison-Wesley, Reading, MA (1998).

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA (1996).

[Mur93] R. B. Murray. C++ *Strategies and Tactics*. Addison-Wesley, Reading, MA (1993).

[NBF96] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Cambridge, MA (1996).

[O98] J. M. Orost. "The Bench++ Benchmark Suite," *Dr. Dobbs Journal*, October 1998.

[Pat] D. A. Patterson. "Microprocessors in 2020," *Scientific American*, September 1995.

[PH97] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA (1997).

[Str97] B. Stroustrup. *The C ++ Programming Language, Third Edition*. Addison-Wesley, Reading, MA (1997).