**第 7 章　优化热点语句**

创意就在那里，它被锁在了里面，我所要做的就是移除多余的石头。

——米开朗基罗 • 博那罗蒂（1475—1564）在面对

“您如何创作杰作”的提问时如是回答

语句级别的优化可以被模式化为**从执行流中移除指令** 的过程，这与米开朗基罗雕刻其杰作的过程相似。米开朗基罗所给出的建议的问题在于，并没有指出石头中的哪部分是多余的，哪部分是杰作。

语句级别的性能优化的问题在于，除了函数调用外，没有哪条 C++ 语句会消耗许多条机器指令。通常，集中精力在这些微小的性能点上是无法获得与开发人员所付出的努力相应的性能回报的，除非开发人员找到了放大这些语句的开销、使得它们成为值得优化的热点代码的因素。这些因素包括以下几个。

**循环**

循环中的语句开销是语句各自的开销乘以它们被重复执行的次数。热点循环必须由开发人员自己找出来。分析器可以指出包含热点循环的函数，但它不会指出函数中的哪个循环是热点循环；它还可能会因为某个函数被一个或多个循环调用而指出该函数，但它也不会指出具体哪个循环是热点循环。既然分析器无法直接指出热点循环，开发人员就必须以分析器的输出结果作为线索，检查代码并找出热点循环。

**频繁被调用的函数**

函数的开销是函数自身的开销乘以它被执行的次数。分析器可以直接指出热点函数。

**贯穿整个程序的惯用法**

这是一个与 C++ 语句和惯用法有关的总类别。在这个类别中，存在着性能开销更小的选项。如果在程序中广泛地使用了这些惯用法，那么将它替换为性能开销更小的惯用法可以提升程序的整体性能。

在语句级别优化代码能够显著地改善嵌入在各种工具、装置、外设和玩具中的简单的小型处理器的性能，因为在这类处理器上，指令是直接从内存中被获取，然后一条一条被执行的。不过，由于桌面级和手持设备的处理器提供了指令级的并发和缓存，因此语句级别的优化带来的回报比优化内存分配和复制要小。

在为桌面级计算机设计的程序中，应当只对那些会被频繁调用的库函数或是程序中最底层的循环，如占用最多运行时间的图形引擎或编程语言解释器，进行语句级别的优化。

语句级别的性能优化还有一个问题：优化效果取决于编译器。对于如何为一条特定的 C++ 语句生成代码，每种编译器都会有一个或多个方案。适用于某个编译器的编程惯用法可能在另外一个编译器上毫无效果，甚至反而会降低性能。当在使用 GCC 时可以改善性能的技巧可能无法适用于 Visual C++。更关键的是，这意味着当团队升级了编译器版本后，新的编译器可能会降低他们精心优化后的代码的速度。这是语句级别的优化可能比其他性能优化手段效果更差的另一个原因。

**7.1　从循环中移除代码**

一个循环是由两部分组成的：一段被重复执行的控制语句和一个确定需要进行多少次循环的控制分支。通常情况下，移除 C++ 语句中的计算指的是移除循环中的控制语句的计算。不过在循环中，控制分支也有额外的优化机会，因为从某种意义上说，它产生了额外的开销。

请考虑代码清单 7-1 中的 for 循环，它会遍历一个字符串，找出空格并用星号替换之。

**代码清单 7-1** 　未优化的 for 循环

char s[] = "This string has many space (0x20) chars. "; ... for (size\_t i = 0; i < strlen(s); ++i) if (s[i] == ' ') s[i] = '\*';

这段代码对字符串中的每个字符都会判断循环条件 i < strlen(s) 是否成立 **1** 。调用 strlen() 的开销是昂贵的，遍历参数字符串对它的字符计数使得这个算法的开销从 *O* (*n* ) 变为了 *O* (*n* 2 )。这是一个在库函数中隐藏了循环（请参见 3.5.2 节）的典型例子。

**1** 有些读者会感到吃惊：“哇！为什么有人写出这样的代码呢？难道他们不知道 std::string 有一个常量时间开销的 length() 函数吗？”不过这类代码通常都会出现在那些需要优化的程序中。我也希望使用更加典型的示例，因为我要证明这一点。

在 Visual Studio 2010 上，这个循环进行 1000 万次迭代耗时 13 238 毫秒；而在 Visual Studio 2015 上则耗时 11 467 毫秒。VS2015 的测量速度比 VS2010 提高了 15%，这表明两个编译器对这个循环生成的代码有所不同。

**7.1.1　缓存循环结束条件值**

我们可以通过在进入循环时预计算并缓存循环结束条件值，即调用开销昂贵的 strlen() 的返回值，来提高程序性能。修改后的循环如代码清单 7-2 所示。

**代码清单 7-2** 　缓存了循环结束条件值的 for 循环

for (size\_t i = 0, len = strlen(s); i < len; ++i) if (s[i] == ' ') s[i] = '\*';

由于 strlen() 的开销实在是太大了，因此修改后的效果非常明显。对修改后的代码进行性能测试的结果是，在 VS2010 上耗时 636 毫秒，而在 VS2015 上则耗时 541 毫秒——比初始版本快了大约 20 倍。VS2015 仍然比 VS2010 快，这次快了 17%。

**7.1.2　使用更高效的循环语句**

以下是 C++ 中 for 循环语句的声明语法：

for (初始化表达式 ; 循环条件 ; 继续表达式 ) 语句

粗略地讲，for 循环会被编译为如下代码：

初始化表达式 ; L1: if ( ! 循环条件 ) goto L2; 语句 ; 继续表达式 ; goto L1; L2:

for 循环必须执行两次 jump 指令：一次是当循环条件为 false 时；另一次则是在计算了继续表达式之后。这些 jump 指令可能会降低执行速度。C++ 还有一种使用不那么广泛的、称为 do 的更简单的循环形式，它的声明语法如下：

do 语句 while ( 循环条件 ) ;

粗略地讲，do 循环会被编译为如下代码：

L1: 控制语句 if ( 循环条件 ) goto L1;

因此，将一个 for 循环简化为 do 循环通常可以提高循环处理的速度。代码清单 7-3 是一个将遍历字符串的 for 循环代码转换为 do 循环的例子。

**代码清单 7-3** 　for 循环被转换为 do 循环

size\_t i = 0, len = strlen(s); // for循环初始化表达式 do { if (s[i] == ' ') s[i] = ' '; ++i; // for循环继续表达式 } while (i < len); // for循环条件

对修改后的代码进行测试的结果是：在 Visual Studio 2010 上耗时 482 毫秒，性能提高了 12%；不过在 Visual Studio 2015 上却耗时 674 毫秒，性能降低了 25%。

**7.1.3　用递减替代递增**

缓存循环结束条件的另一种方法是用递减替代递增，将循环结束条件缓存在循环索引变量中。许多循环都有一种结束条件判断起来比其他结束条件更高效。例如，在代码清单 7-3 的循环中，一种结束条件是常量 0，而另外一种则是调用开销昂贵的 strlen() 函数。代码清单 7-4 将代码清单 7-1 中的循环重新组织，用递减替代了递增。

**代码清单 7-4** 　对循环进行递减优化

for (int i = (int)strlen(s)-1; i >= 0; --i) if (s[i] == ' ') s[i] = '\*';

请注意，我将归纳变量 i 的类型从无符号的 size\_t 变为了有符号的 int 。for 循环的结束条件是 i >= 0 。如果 i 是无符号的，从定义上说，它总是大于或等于 0，那么循环就永远无法结束。在采用递减方式时，这是一个非常典型的错误。

我对这个函数进行了相同的性能测试，结果是在 Visual Studio 2010 上的运行时间是 619 毫秒，而在 Visual Studio 2015 上的运行时间则是 571 毫秒。与代码清单 7-2 中的代码相比，我们无法确定这个结果是否表示有显著的性能提升。

**7.1.4　从循环中移除不变性代码**

在代码清单 7-2 中，结束条件被缓存起来供复用，这样更加高效。它是将具有不变性的代码移动至循环外部这个通用技巧的一个典型例子。当代码不依赖于循环的归纳变量时，它就具有循环不变性。例如，在我故意编写的代码清单 7-5 的循环中，赋值语句 j = 100; 以及子表达式 j \* x \* x 就具有循环不变性。

**代码清单 7-5** 　含有循环不变性代码的循环

int i,j,x,a[10]; ... for (i=0; i<10; ++i) { j = 100; a[i] = i + j \* x \* x; }

我们可以如代码清单 7-6 那样重写这段代码。

**代码清单 7-6** 　将循环不变性代码移动至循环外

int i,j,x,a[10]; ... j = 100; int tmp = j \* x \* x; for (i=0; i<10; ++i) { a[i] = i + tmp; }

现代编译器非常善于找出在循环中被重复计算的具有循环不变性的代码（如同这里介绍的），然后将计算移动至循环外部来改善程序性能。开发人员通常没有必要重写这段代码，因为编译器已经替我们找出了具有循环不变性的代码并重写了循环。

当在循环中有语句调用了函数时，编译器可能无法确定函数的返回值是否依赖于循环中的某些变量。被调用的函数可能很复杂，或是函数体包含在另外一个编译器看不到的编译单元中。这时，开发人员必须自己找出具有循环不变性的函数调用并将它们从循环中移除。

**7.1.5　从循环中移除无谓的函数调用**

一次函数调用可能会执行大量的指令。如果函数具有**循环不变性** （loop-invariant），那么将它移除到循环外有助于改善性能。在代码清单 7-1 中，strlen() 具有循环不变性，因此可以将它到移动到循环外部：

char\* s = "sample data with spaces"; ... for (size\_t i = 0; i < strlen(s); ++i) if (s[i] == ' ') s[i] = '\*'; // 将' '改为'\*'

代码清单 7-7 展示了修改后的循环的模样。

**代码清单 7-7** 　循环中的 strlen() 具有循环不变性

char\* s = "sample data with spaces"; ... size\_t end = strlen(s); for (size\_t i = 0; i < end; ++i) if (s[i] == ' ') s[i] = '\*'; // 将' '改为'\*'

在代码清单 7-8 中，strlen() 返回的值不具有循环不变性，因为移除一个空格字符会缩短字符串的长度。因此，end 条件不能被移动到循环外部。

**代码清单 7-8** 　strlen() 不具有循环不变性的循环

char\* s = "sample data with spaces"; size\_t i; ... for (i = 0; i < strlen(s); ++i) if (s[i] == ' ') strcpy(&s[i], &s[i+1]); // 移除空格 s[i] = '\0';

没有一个简单的规则可以确定在某种情况下一个函数是否具有循环不变性。代码清单 7-8 向我们展示了一个函数在某个循环中具有循环不变性，但在另外一个循环中却不具有循环不变性的情况。在这种情况下，相比于编译器彻底但有限的分析能力，开发人员的判断更加有效（重复调用 strlen() 并非是这个函数唯一的降低性能的问题。作为练习，剩下的问题请读者自己找出来）。

有一种函数永远都可以被移动到循环外部，那就是返回值只依赖于函数参数而且没有副作用的**纯函数** （pure function）。如果在循环出现了这种函数，而且在循环中不会改变它的参数，那么这个函数就具有循环不变性，可以将其移动到循环外。在代码清单 7-8 中，函数 strlen() 就是一个纯函数。在第一个循环中永远不会改变它的参数 s ，因此，对 strlen() 的调用具有循环不变性；而在第二个循环中，对 strcpy() 的调用改变了 s ，因此，对 strlen() 的调用不具有循环不变性。

下面是涉及数学函数 sin() 和 cos() 的另外一个例子，这两个函数的返回值分别是以弧度表示的正弦值和余弦值。许多数学函数都是纯函数，因此，这种情况经常发生在数学计算中。代码清单 7-9 中的函数会对一副有 16 个顶点的图形进行图像旋转变换。这个变化的性能测试结果在 VS2010 上执行 100 万次耗时 7502 毫秒，在 VS2015 上执行 100 万次耗时 6864 毫秒，后者相比前者有 15% 的性能优势。

**代码清单 7-9** 　包含具有循环不变性的纯函数的 rotate()

void rotate(std::vector<Point>& v, double theta) { for (size\_t i = 0; i < v.size(); ++i) { double x = v[i].x\_, y = v[i].y\_; v[i].x\_ = cos(theta)\*x - sin(theta)\*y; v[i].y\_ = sin(theta)\*x + cos(theta)\*y; } }

函数 sin(theta) 和 cos(theta) 只依赖于函数参数 theta ，并不依赖于循环变量。如代码清单 7-10 所示，我们可以将它们移动到循环外部。

**代码清单 7-10** 　将具有循环不变性的纯函数移动到循环外部后的 rotate\_invariant()

void rotate\_invariant(std::vector<Point>& v, double theta) { double sin\_theta = sin(theta); double cos\_theta = cos(theta); for (size\_t i = 0; i < v.size(); ++i) { double x = v[i].x\_, y = v[i].y\_; v[i].x\_ = cos\_theta\*x - sin\_theta\*y; v[i].y\_ = sin\_theta\*x + cos\_theta\*y; } }

对修改后的函数进行测试的结果是性能大约提高了 3%，分别达到 7382 毫秒（VS2010）和 6620 毫秒（VS2015）。

在 PC 上，相比于上一节中将 strlen() 移动到循环外部的性能改善效果，这里的性能提升并不明显，因为数学函数通常只会对保存在寄存器中的一两个数字进行运算，而且不会像 strlen() 一样访问内存。如果是在 20 世纪 90 年代的老式 PC 或不具有浮点指令的嵌入式处理器上，这种性能提升可能会更加明显，因为正弦和余弦的计算开销更大。

有时候，在循环中调用的函数根本就不会工作或者只是进行一些无谓的工作。我们当然可以移除这些函数。可能会有读者认为“不过，称职的开发人员不会调用一个进行无谓工作的函数”。但是，要想在一个项目长达数年的生命周期中记住所有调用了该函数的地方，并在每次函数被修改后都检查该函数的所有调用，是非常困难的。

下面这段伪代码是我在整个职业生涯中反复使用的编程惯用法：

UsefulTool subsystem; InputHandler input\_getter; ... while (input\_getter.more\_work\_available()) { subsystem.initialize(); subsystem.process\_work(input\_getter.get\_work()); }

在这个模式中，程序会不断地初始化 subsystem ，然后要求其进行下一项工作。这段代码中可能存在一个只有通过检查 UsefulTool::initialize() 才能确定是否确实存在的问题。程序可能只是在执行第一项工作前需要调用 initialize() ，或是只是在执行第一项工作前以及在处理出错后需要调用 initialize() 。通常，process\_work() 会在退出 initialize() 建立的类不变性时建立相同的类不变性。在每次循环中都调用 initialize() 只是在重复执行与 process\_work() 相同的代码。如果是这样的话，可以如下这样将 initialize() 移动到循环外部：

UsefulTool subsystem; InputHandler input\_getter; ... subsystem.initialize(); while (input\_getter.more\_work\_available()) { subsystem.process\_work(input\_getter.get\_work()); }

责备开发人员草率地编写代码有些自以为是。有时候，initialize() 的行为会发生变化，其中的部分代码可能会被移动至 process\_work() 中。有时候，项目中会缺少项目文档或者项目计划很紧张，抑或是 initialize() 的目的不够明确，而开发人员只是保守地编写了代码。但是我确实多次碰到过明明只需要进行一次初始化，却在每次进行一项工作前都初始化的情况。

如果迫切地需要缩短程序执行时间，那么就值得检查循环中的每处函数调用，看看是否真的需要它们。

**7.1.6　从循环中移除隐含的函数调用**

普通的函数调用很容易识别，它们有函数名，在圆括号中有参数表达式列表。C++ 代码还可能会隐式地调用函数，而没有这种很明显的调用语句。当一个变量是以下类型之一时就可能会发生这种情况：

* 声明一个类实例（调用构造函数）
* 初始化一个类实例（调用构造函数）
* 赋值给一个类实例（调用赋值运算符）
* 涉及类实例的计算表达式（调用运算符成员函数）
* 退出作用域（调用在作用域中声明的类实例的析构函数）
* 函数参数（每个参数表达式都会被复制构造到它的形参中）
* 函数返回一个类的实例（调用复制构造函数，可能是两次）
* 向标准库容器中插入元素（元素会被移动构造或复制构造）
* 向矢量中插入元素（如果矢量重新分配了内存，那么所有的元素都需要被移动构造或是复制构造）

这些函数调用被**隐藏** 起来了。你从表面上看不出带有名字和参数列表的函数调用。它们看起来更像赋值和声明。我们很容易误以为这里没有发生函数调用。我在 6.5 节中已经讨论过这些内容了。

如果将函数签名从通过值传递实参修改为传递指向类的引用或指针，有时候可以在进行隐式函数调用时移除形参构建。在 4.2.3 节中，我已经在字符串中证明过这一点了，而对于其他复制数据的对象，我则在 6.5.2 节中证明过了。

如果将函数签名修改为通过输出参数返回指向类实例的引用或指针时，可以在进行隐式函数调用时移除函数返回值的复制。我已经在 4.2.5 节中证明了这对字符串是有效的，在 6.5.3 节中则已证明对任何其他复制数据的对象都是有效的。

如果赋值语句和初始化声明具有循环不变性，那么我们可以将它们移动到循环外部。有时，即使需要每次都将变量传递到循环中，你也可以将声明移动到循环外部，并在每次循环中都执行一次开销较小的函数调用。例如，std::string 是一个含有动态分配内存的字符数组的类。在以下代码中：

for (...) { std::string s("<p>"); ... s += "</p>"; }

在 for 循环中声明 s 的开销是昂贵的。在循环语句块的反大括号的位置将会调用 s 的析构函数，而析构函数会释放为 s 动态分配的内存，因此当下一次进入循环时，一定会重新分配内存。这段代码可以被优化为：

std::string s; for (...) { s.clear(); s += "<p>"; ... s += "</p>"; }

现在，不会再在每次循环中都调用 s 的析构函数了。这不仅仅是在每次循环中都节省了一次函数调用，同时还带来了其他效果——由于 s 内部的动态数组会被复用，因此当向 s 中添加字符时，可能会移除一次对内存管理器的调用。

这种行为不仅仅适用于字符串或是那些含有动态内存的类。类实例中还可能会含有取自操作系统的资源，如一个窗口或是文件句柄，抑或可能会在它自身的构造函数和析构函数中进行一些开销昂贵的处理。

**7.1.7　从循环中移除昂贵的、缓慢改变的调用**

有些函数调用虽然并不具有循环不变性，但是也可能变得具有循环不变性。一个典型的例子是在日志应用程序中调用获取当前时间的函数。它只需要几条指令即可从操作系统获取当前时间，但是却需要花费些时间来格式化显示时间。代码清单 7-11 是一个将当前时间转换为以空字符结尾的字符数组的函数。

**代码清单 7-11** 　timetoa() ：将时间格式化为字符数组

# include <ctime> char\* timetoa(char \*buf, size\_t bufsz) { if (buf == 0 || bufsz < 9) return nullptr; // 无效参数 time\_t t = std::time(nullptr); // 从操作系统中获取时间 tm tm = \*std::localtime(&t); // 将时间分解为时分秒 size\_t sz = std::strftime(buf, bufsz, "%c", &tm); // 格式化到缓存中 if (sz == 0) strcpy(buf, "XX:XX:XX"); // 错误 return buf; }

在性能测试实验中，timetoa() 花费了大约 700 纳秒完成了获取和格式化时间的处理。但是这个时间太长了，它相当于连接两个文本字符串到文件中的时间的两倍。在相同的性能测试中，语句

out << "Fri Jan 01 00:00:00 2016" << " Test log line test log line test log line\n";

只花费了 372 纳秒，而语句

out << timetoa(buf, sizeof(buf)) << " Test log line test log line test log line\n";

则花费了 1042 纳秒。

日志记录必须尽可能地高效，否则会降低程序的性能。如果这降低了程序性能就糟糕了，如果性能的下降改变了程序行为，进而导致在打开日志记录后程序的 bug 消失就更糟了。在这个例子中，获取当前时间决定了记录日志的开销。

相比于现代计算机的指令执行速度，时间的改变非常慢。很明显，我的程序可以在两次时标之间记录 100 万行日志。因此，连续调用 timetoa() 两次获取到的当前时间可能是相同的。如果需要一次记录许多行日志，那么就没有理由在记录每条时都去获取当前时间。

我进行了一项测试来模拟程序请求当前时间时的日志行为，然后使用相同的时间以 10 行日志为一组输出日志。与预想相同，这项测试的输出结果是平均每行耗时 376 纳秒。

**7.1.8　将循环放入函数以减少调用开销**

如果程序要遍历字符串、数组或是其他数据结构，并会在每次迭代中都调用一个函数，那么可以通过一种称为**循环倒置** （loop inversion）的技巧来提高程序性能。循环倒置是指将在循环中调用函数变为在函数中进行循环。这需要改变函数的接口，不再接收一条元素作为参数，而是接收整个数据结构作为参数。按照这种方式修改后，如果数据结构中包含 *n* 条元素，那么可以节省 *n* -1 次函数调用。

我们来看一个非常简单的例子。下面这个函数的功能是用点（“.”）替代非打印字符：

# include <ctype> void replace\_nonprinting(char& c) { if (!isprint(c)) c = '.'; }

当想替换一个字符串中所有的非打印字符时，可以在程序中循环中调用 replace\_nonprinting() ：

for (unsigned i = 0, e = str.size(); i < e; ++i) replace\_nonprinting(str[i]);

如果编译器无法对 replace\_nonprinting() 内联展开，那么当需要处理的字符串是“Ring the carriage bell\x07\x07!!”时，它会调用这个函数 26 次。

库的设计者可以重载 replace\_nonprinting() 函数来处理整个字符串：

void replace\_nonprinting(std::string& str) { for (unsigned i = 0, e = str.size(); i < e; ++i) if (!isprint(str[i])) c = '.'; }

现在，循环在函数内部了，这样可以节省 *n* -1 次对 replace\_nonprinting() 的调用。

请注意，必须将 replace\_nonprinting() 的实现代码复制到新的重载函数中。仅仅在新的重载函数的循环中调用之前的函数是没有效果的。下面的版本实际上只是在循环中调用了之前的函数：

void replace\_nonprinting(std::string& str) { for (unsigned i = 0, e = str.size(); i < e; ++i) replace\_nonprinting(str[i]); }

**7.1.9　不要频繁地进行操作**

我们来看一个启发式问题：“在一个程序的主循环中每秒处理约 1000 个事务，那么它应当每隔多长时间检测一次是否有终止命令呢？”

答案当然是“视情况而定”。事实上，这取决于两件事情：程序需要以多快的速度响应终止请求，以及程序检查终止命令的开销。

如果程序的响应目标是需要在一秒内停止程序，而且在检测到停止命令后需要平均 500 ± 100 毫秒来停止程序，那么它需要每 400 毫秒（1000 - (500 + 100) = 400 毫秒）检测一次。更频繁地检测只会是浪费。

另一个因素是检测终止命令的开销。如果主循环是 Windows 消息循环，那么终止命令就是 Windows 的 WM\_CLOSE 消息。由于此时开销包含在了事件分发中，因此不会发生额外的检测开销。如果信号处理函数会设置一个 bool 标识位，那么每次在循环中检测这个标识位的开销非常微小。

但是如果是在嵌入式设备上用循环轮询键盘按键，而且必须对按键消除抖动 **2** 50 毫秒时会怎样呢？每次测试程序进入循环都会在每个事务的开销上加上 50 毫秒按键轮询开销，将处理速度从每秒 1000 个事务降低为每秒 1/0.051 = 20 个事务。这个结果让人难以接受。

**2** 当一个真实的机械按键被按下时会建立一个连接，而这个连接最初是断断续续的。在这种初始的断续状态下，一瞬间去查看这个连接的状态会误认为按键没有被按下。“消除抖动”使得按键被按下的消息被推迟至连接变为连续状态后才发送。50 毫秒是一个常用的消除抖动间隔。

如果程序只以 400 毫秒的间隔轮询键盘按下事件，那么对循环性能的影响就没有那么大了。这里的数学计算有些冗长，我们跳过这部分。每个事务大约耗时 1 毫秒（因为每秒 1000 个事务）。那么要每 400 毫秒进行一次耗时 50 毫秒的轮询，轮询必须从 350 毫秒开始，即 1000 毫秒 2.5 次。这样，事务的处理速率就是每秒 1000 -(2.5×50) = 875 个事务。

代码清单 7-12 展示了检测按键是否按下的代码。

**代码清单 7-12** 　不要频繁地检测事件

void main\_loop(Event evt) { static unsigned counter = 1; if ((counter % 350) == 0) if (poll\_for\_exit()) exit\_program(); // 不返回 ++counter; switch (evt) { ... } }

执行会每毫秒进入一次 main\_loop() （假设事件的发生时间是毫秒级）。每次通过循环时都会增加计数值。当计数值达到 350 时，程序会调用 poll\_for\_exit() ，这将会花费 50 毫秒。如果在 poll\_for\_exit() 中检测到退出键按下，代码会调用 exit\_program() ，这将会花费 400~600 毫秒来停止程序。

这种非正式的轮询方法展示了如何在两次轮询间进行更多的计算。不过，它也带有许多假设条件。

* 它假设每毫秒都发生事件，而不是有时候每 2 毫秒或是每 5 毫秒，而且即使在没有任何工作时，这个事件的发生速率也不会降低。
* 它假设轮询总是精确地耗时 50 毫秒。
* 它假设调试器永远不会获取程序控制权，即使开发人员在检查变量的值时，也不会有任何一个事件需要花费半分钟。

一种更加稳妥的方法是，测量两个事件之间经过的时间以及从进入 poll\_for\_exit() 到退出 poll\_for\_exit() 之间经过的时间。

如果面对上面列举出的各种限制条件，开发人员仍然想要实现每秒 1000 个事务的响应指标，那么他必须找出并发实现主循环与轮询键盘按键事件的方法。典型的实现方式有中断、多核心处理和使用硬件的键盘扫描器。

**7.1.10　其他优化技巧**

在互联网上有许多关于循环的底层优化技巧资料。例如，有些资料指出 ++i 通常比 i++ 更加高效，因为不需要保存或是返回任何中间值。有些资料建议展开循环来减少循环条件测试语句和循环条件增长语句的执行次数。

这些建议的问题在于它们并非总是有效果。你可能花费了很多时间来进行这些实验，但是却观察不到任何改善效果。这些建议来自于猜想而非实验结果，或者可能在某个特定的日子里在某种特定的编译器上有效果。这些建议也可能来自关于编译器设计的教材，它们所描述的性能优化技巧实际上编译器已经替我们做了。这 30 多年来，现代 C++ 编译器已经非常善于将循环内的代码移动到循环外部了。事实上，编译器比绝大多数程序员的编程能力更加优秀。这也是为什么使用类似的性能优化技巧的结果总是让人沮丧，以及为什么本节中的内容并不会太多。

**7.2　从函数中移除代码**

与循环一样，函数也包含两部分：一部分是由一段代码组成的函数体，另一部分是由参数列表和返回值类型组成的函数头。与优化循环一样，这两部分也可以独立优化。

尽管执行函数体的开销可能会非常大，但是调用函数的开销与调用大多数 C++ 语句的开销一样，是非常小的。不过，当函数被多次调用时，累积的开销可能会变得巨大，因此减少这种开销非常重要。

**7.2.1　函数调用的开销**

函数是编程中最古老和最重要的抽象概念。程序员先定义一个函数，接着就可以在代码中的其他地方调用这个函数。每次调用时，计算机都会在执行代码中保存它的位置，将控制权交给函数体，接着会返回到函数调用后的下一条语句，高效地将函数体插入到指令执行流中。

这种便利性可不是免费的。每次程序调用一个函数时，都会发生类似下面这样的处理（依赖于处理器体系结构和优化器设置）。

(1) 执行代码将一个栈帧推入到调用栈中来保存函数的参数和局部变量。

(2) 计算每个参数表达式并复制到栈帧中。

(3) 执行地址被复制到栈帧中并生成返回地址。

(4) 执行代码将执行地址更新为函数体的第一条语句（而不是函数调用后的下一条语句）。

(5) 执行函数体中的指令。

(6) 返回地址被从栈帧中复制到指令地址中，将控制权交给函数调用后的语句。

(7) 栈帧被从栈中弹出。

不过，关于函数开销也有一些好消息。带有函数的程序通常都会比带有被内联展开的大型函数的程序更加紧凑。这有利于提高缓存和虚拟内存的性能。而且，函数调用与非函数调用的其他开销都相同，这使得提高会被频繁地调用的函数的性能成为了一种有效的优化手段。

1. **函数调用的基本开销**

有许多细节问题都会降低 C++ 中函数调用的速度，这些问题也构成了函数调用优化的基础。

**函数参数**

除了计算参数表达式的开销外，复制每个参数的值到栈中也会发生开销。如果只有几个小型的参数，那么可能可以很高效地将它们传递到寄存器中；但是如果有很多参数，那么至少其中一部分需要通过栈传递。

**成员函数调用（与函数调用）**

每个成员函数都有一个额外的隐藏参数：一个指向 this 类实例的指针，而成员函数正是通过它被调用的。这个指针必须被写入到调用栈上的内存中或是保存在寄存器中。

**调用和返回**

调用和返回对程序的功能没有任何影响。我们可以通过用函数体替代函数调用来移除这些开销。的确，当函数很小且在函数被调用之前已经定义了函数时，许多编译器都会试图内联函数体。如果不能内联函数，调用和返回就会产生开销。

调用函数要求执行地址被写入到栈帧中来生成返回地址。

函数返回要求执行地址从栈中被读取出来并加载到执行指针中。在调用和返回时，执行连续地工作于非连续的内存地址上。正如在 2.2.7 节中所讲过的，计算机能够高效地执行连续指令。不过，当程序执行需要跨越非连续地址时，可能会发生流水线停顿和高速缓存未命中。

1. **虚函数的开销**

在 C++ 中可以将任何成员函数定义为**虚函数** 。继承类能够通过定义一个具有相同函数签名的成员函数来重写基类的虚成员函数。这样，不论是在继承类实例上调用虚函数还是在一个指向基类类型的指针或是引用上调用虚函数，都可以使用新的函数体。程序在解引类实例时会选择调用哪个函数。因此，程序是在运行时通过类实例的实际类型来确定要调用哪个重写函数的。

每个带有虚成员函数的实例都有一个无名指针指向一张称为**虚函数表** （vtable）的表，这张表指向类中可见的每个虚函数签名所关联的函数体。虚函数表指针通常都是类实例的第一个字段，这样解引时的开销更小。

由于虚函数调用会从多个函数体中选择一个执行，调用虚函数的代码会解引指向类实例的指针，来获得指向虚函数表的指针。这段代码会为虚函数表加上索引（也就是说，代码会在虚函数表上加上一段小的整数偏移量并解引该地址）来得到函数的执行地址。因此，实际上这里会为所有的虚函数调用额外地加载两次非连续的内存，每次都会增加高速缓存未命中的几率和发生流水线停顿的几率。虚函数的另一个问题是编译器难以内联它们。编译器只有在它能同时访问函数体和构造实例的代码（这样编译器才能决定调用虚函数的哪个函数体）时才能内联它们。

1. **继承中的成员函数调用**

当一个类继承另一个类时，继承类的成员函数可能需要进行一些额外的工作。

**继承类中定义的虚成员函数**

如果继承关系最顶端的基类没有虚成员函数，那么代码必须要给 this 类实例指针加上一个偏移量，来得到继承类的虚函数表，接着会遍历虚函数表来获取函数执行地址。这些代码会包含更多的指令字节，而且这些指令通常都比较慢，因为它们会进行额外的计算。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

**多重继承的继承类中定义的成员函数调用**

代码必须向 this 类实例指针中加上一个偏移量来组成指向多重继承类实例的指针。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

**多重继承的继承类中定义的虚成员函数调用**

对于继承类中的虚成员函数调用，如果继承关系最顶端的基类没有虚成员函数，那么代码必须要给 this 类实例指针加上一个偏移量来得到继承类的虚函数表，接着会遍历虚函数表来获取函数执行地址。代码还必须向 this 类实例指针加上潜在的不同的偏移量来组成继承类的类实例指针。这种开销在小型嵌入式处理器上非常显著，但是在桌面级处理器上，指令级别的并发掩盖了大部分这种额外的开销。

**虚多重继承**

为了组成虚多重继承类的实例的指针，代码必须解引类实例中的表，来确定要得到指向虚多重继承类的实例的指针时需要加在类实例指针上的偏移量。如前所述，当被调用的函数是虚函数时，这里也会产生额外的间接开销。

1. **函数指针的开销**

C++ 提供了函数指针，这样当通过函数指针调用函数时，代码可以在运行时选择要执行的函数体。除了基本的函数调用和返回开销外，这种机制还会产生其他额外的开销。

**函数指针（指向非成员函数和静态成员函数的指针）**

C++ 允许在程序中定义指向函数的指针。程序员可以通过函数指针显式地选择一个具有特定签名（由参数列表和返回类型组成）的非成员函数。当函数指针被解引后，这个函数将会在运行时会被调用。通过将一个函数赋值给函数指针，程序可以显式地通过函数指针选择要调用的函数。

代码必须解引指针来获取函数的执行地址。编译器也不太可能会内联这些函数。

**成员函数指针**

成员函数指针声明同时指定了函数签名和解释函数调用的上下文中的类。程序通过将函数赋值给函数指针，显式地选择通过成员函数指针调用哪个函数。

成员函数指针有多种表现形式，一个成员函数只能有一种表现形式。它必须足够通用才能够在以上列举的各种复杂的场景下调用任意的成员函数。我们有理由认为一个成员函数指针会出现最差情况的性能。

1. **函数调用开销总结**

因此，C 风格的不带参数的 void 函数的调用开销是最小的。如果能够内联它的话，就没有开销；即使不能内联，开销也仅仅是两次内存读取加上两次程序执行的非局部转移 **3** 。

如果基类没有虚函数，而虚函数在多重虚拟继承的继承类中，那么这是最坏的情况。不过幸运的是，这种情况非常罕见。在这种情况下，代码必须解引类实例中的函数表来确定加到类实例指针上的偏移量，构成虚拟多重继承函数的实例的指针，接着解引该实例来获取虚函数表，最后索引虚函数表得到函数执行地址。

此时，读者可能会惊讶函数调用的开销居然如此之大，抑或是惊叹 C++ 居然如此高效地实现了这么复杂的特性。这两种看法都是合理的。需要理解的是正是有了函数调用开销，才有优化的机会。坏消息是除非函数会被频繁地调用，否则移除一处非连续内存读取并不足以改善性能；好消息则是分析器会直接指出调用最频繁的函数，让开发人员能够快速地集中精力于最佳优化对象。

**3** 即非局部跳转。——译者注

**7.2.2　简短地声明内联函数**

移除函数调用开销的一种有效方式是内联函数。要想内联函数，编译器必须能够在函数调用点访问函数定义。那些函数体在类定义中的函数会被隐式地声明为内联函数。通过将在类定义外部定义的函数声明为存储类内联，也可以明确地将它们声明为内联函数。此外，如果函数定义出现在它们在某个编译单元中第一次被使用之前，那么编译器还可能会自己选择内联较短的函数。尽管 C++ 标准说 inline 关键字只是对编译器的“提示”，但是实际上为了编译器自己的销量，它们必须善于内联函数。

当编译器内联一个函数时，那么它还有可能会改善代码，包括移除调用和返回语句。有些数学计算可能会在编译时完成。如果编译器能够确定当参数为某个特定值时有些分支永远不会执行，那么编译器会移除这些分支。因此，内联是一种通过在编译时进行计算来移除多余计算的改善性能的手段。

函数内联可能是最强力的代码优化武器。事实上，Visual Studio 中“调试”版本与“正式”版本（或是在 GCC 的 -d 选项与 -O 选项）的性能区别，主要源于“调试”版本关闭了函数内联。

**7.2.3　在使用之前定义函数**

在第一次调用函数之前定义函数（提供函数体）给了编译器优化函数调用的机会。当编译器编译对某个函数的调用时发现该函数已经被定义了，那么编译器能够自主选择内联这次函数调用。如果编译器能够同时找到函数体，以及实例化那些发生虚函数调用的类变量、指针或是引用的代码，那么这也同样适用于虚函数。

**7.2.4　移除未使用的多态性**

在 C++ 中，虚成员函数多用来实现运行时**多态性** 。多态性允许成员函数根据不同的调用对象，从多个不同但语义上有关联的方法中选择一个执行。

要实现多态行为，可以在基类中定义虚成员函数。然后任何继承类都能够选择使用特化行为来重写基类函数的行为。这些不同的实现是通过每个继承类都必须有不同的实现的语义概念关联在一起的。

多态的一个典型例子是定义在表示绘制在屏幕上的图形对象的 DrawableObject 的继承类中的 Draw() 函数。当调用 drawObjPtr->Draw() 时，程序会通过解引 drawObjPtr 所指向的实例中的虚函数表来选择使用 Draw() 的哪种实现。当类实例是 Triangle 的实例时，Draw() 的实现会画出一个三角形；而当类实例是 Rectangle 的实例时，则会画出一个长方形，等等。由于 DrawableObject::Draw() 被声明为虚函数，因此程序会调用合适的继承类的 Draw() 成员函数。当程序必须在运行时从多种实现中选择一种执行时，虚函数表是一种非常高效的机制，它的间接开销只有两次额外的内存读取以及与这两次内存读取相关的流水线停顿。

不过，多态仍然可能会带来不必要的性能开销。例如，一个类的本来的设计目的是方便实现派生类的层次结构，但是最后却没有实现这些派生类；或者一个函数被声明为虚函数是希望利用多态性，但这个函数却永远没有被实现。在上面的例子中，所有的可绘制对象可能都被实现为按顺序连接在一起的点，这样就总是会使用基类中的 Draw() 。当不会重写该方法时，移除 DrawableObject 的 Draw() 函数声明中的 virtual 关键字可以提高 Draw() 的调用速度。

**停下来思考**

设计人员希望 DrawableObject 成为一组具有继承关系的类层次的根对象，而性能优化开发人员则希望改善程序性能，因为 Draw() 成员函数根本没有实现。这两者之间存在矛盾。假设实验指出 Draw() 成员函数就是性能问题的元凶，那么设计人员可能会让步。如果有必要，以后再加上 virtual 关键字也是很容易的。聪明的开发人员如果没有充足的理由是不会去破坏设计的，而且也不会要求修改所有的虚函数。

**7.2.5　放弃不使用的接口**

在 C++ 中可以使用虚成员函数实现接口——一组通用函数的声明。这些函数描述了对象行为，而且它们在不同的情况下有不同的实现方式。基类通过声明一组纯虚函数（有函数声明，但没有函数体的函数）定义接口。由于纯虚函数没有函数体，因此 C++ 不允许实例化接口基类。继承类可以通过重写（定义）接口基类中的所有纯虚函来实现接口。C++ 中接口惯用法的优点在于，继承类必须实现接口中声明的所有函数，否则编译器将不会允许程序创建继承类的实例。

例如，开发人员可以使用接口类来隔离操作系统依赖性，特别是当设计人员预计需要为多个操作系统实现程序时。我们可以通过下面的接口类 file 来定义读写文件的类。这个 file 被称为**抽象基类** ，因为它无法被实例化：

// file.h——接口 class File { public: virtual ~File() {} virtual bool Open(Path& p) = 0; virtual bool Close() = 0; virtual int GetChar() = 0; virtual unsigned GetErrorCode() = 0; };

在其他代码中定义的 Windowsfile 继承类提供了这些函数在 Windows 操作系统上的实现。 C++11 中的关键字 override 是可选关键字，它告诉编译器当前的声明会重写基类中虚函数的声明。当指定了 override 关键字后，如果在基类中没有虚函数声明，编译器会报出警告消息：

// Windowsfile.h——接口 # include "File.h" class WindowsFile : public File { // C++11风格的声明 public: ~File() {} bool Open(Path& p) override; bool Close() override; int GetChar() override; unsigned GetErrorCode() override; };

除了头文件外，还有一个包含了这些重写函数的 Windows 版实现的 windowsfile.cpp 文件：

// windowsfile.cpp——Windows版的实现 # include "WindowsFile.h" bool WindowsFile::Open(Path& p) { ... } bool WindowsFile::Close() { ... } ...

有时，一个程序虽然定义了接口，但是只提供了一种实现。在这种情况下，通过移除接口，即移除 file.h 类定义中的 virtual 关键字并提供 file 的成员函数的实现，可以节省虚函数调用（特别是频繁地对 GetChar() 的调用）的开销。

**停下来思考**

正如前一节中所提到的，开发人员清晰地定义接口的愿景（这当然是好事）与性能优化开发人员改善性能（如果 GetChar() 被分析器标记为热点函数，那么也有问题）的渴求之间存在矛盾。在程序稳定后，判断有无其他实现方法会更加容易。这里的知识可以帮助我们选择到底是优化原来的设计还是保留原来的设计。如果性能优化开发人员并非设计接口的人，那么当他提议进行修改时，必须做好被驳回的准备。他人可能会建议他拿出性能数据来证明修改的合理性。

1. **在链接时选择接口实现**

虚函数允许程序在运行时从多个实现中选择一种。接口允许设计人员指定在开发过程中必须编写哪些函数，以使一个对象可以在程序中被使用。使用 C++ 虚函数实现接口惯用法的问题在于，虚函数为设计时问题提供的是一个带有运行时开销的运行时解决方案。

在上一节中，我们定义了一个名为 file 的接口来隔离操作系统依赖性。在继承类 Windowsfile 中我们实现了这个接口。如果要将这个程序移植到 Linux 上，那么还需要给这段代码加上一个 file 接口的继承类 Linuxfile ，但是 Windowsfile 和 Linuxfile 永远不会在同一个程序中被实例化。它们使得底层调用只会被实现在一种操作系统上。这样就不会发生虚函数的调用开销。而且，如果这个程序会读取一个大文件，file::GetChar() 可能会变为需要优化的热点代码。

如果无需在运行时做出选择的话，那么开发人员可以使用链接器来从多个实现中选择一种。具体做法是不声明 C++ 接口，而是在头文件中直接声明（但不实现）成员函数，就像它们是标准库函数一样：

// file.h——接口 class File { public: File(); bool Open(Path& p); bool Close(); int GetChar(); unsigned GetErrorCode(); };

在 windowsfile.cpp 文件中有如下 Windows 的实现代码：

// windowsfile.cpp——Windows的实现代码 # include "File.h" 　 bool File::Open(Path& p) { ... } 　 bool File::Close() { ... } ...

在另外一个名为 linuxfile.cpp 的相似文件中包含了 Linux 的实现。Visual Studio 工程文件引用 windowsfile.cpp，Linux 的 makefile 则引用 linuxfile.cpp。选择哪个实现会由链接器根据参数列表来做出决定。现在，调用 GetChar() 已经达到最高性能了。（请注意，还有其他方法可以优化 GetChar() 这样的函数，包括 7.1.8 节中提到的循环倒置技巧。）

在链接时选择实现的优点是使得程序具有通用性，而缺点则是部分决定被放在了 .cpp 文件中，部分决定被放在了 makefile 或是工程文件中。

1. **在编译时选择接口实现**

在上一节中，链接器选择了 file 抽象类的一种实现方法。这是可行的，因为 file 的实现依赖于操作系统。如果某个程序只能在一个操作系统上执行，那么没有必要在运行时选择实现。

如果对于两种 file 实现使用不同的编译器（例如对 Window 版本使用 Visual Studio，对 Linux 版本使用 GCC），那么可以在编译时使用 #ifdef 来选择实现。头文件不需要做任何改变。下面是一个名为 file.cpp 的源文件，其中预处理宏会选择实现：

// file.cpp——实现 # include "File.h" # ifdef \_WIN32 bool File::Open(Path& p) { ... } bool File::Close() { ... } ... # else // Linux bool File::Open(Path& p) { ... } 　 bool File::Close() { ... } ... # endif

这个方法要求能够使用预处理宏来选择所希望的实现。有些开发人员喜欢这种方法，因为可以在 .cpp 文件中做更多决定。另外一些开发人员则认为在一个文件中编写两种实现方式是凌乱且非面向对象的。

**7.2.6　用模板在编译时选择实现**

C++ 模板特化是另外一种在编译时选择实现的方法。利用模板，开发人员可以创建具有通用接口的类群，但是它们的行为取决于模板的类型参数。模板参数可以是任意类型——具有自己的一组成员函数的类类型或是具有内建运算符的基本类型。因此，存在两种接口：模板类的 public 成员，以及由在模板参数上被调用的运算符和函数所定义的接口。抽象基类中定义的接口是非常严格的，继承类必须实现在抽象基类中定义的所有函数。而通过模板定义的接口就没有这么严格了。只有参数中那些实际会被模板的某种特化所调用的函数才需要被定义。

模板的特性是一把双刃剑：一方面，即使开发人员在某个模板特化中忘记实现接口了，编译器也不会立即报出错误消息；但另一方面，开发人员也能够选择不去实现那些在上下文中没被用到的函数。

从性能优化的角度看，多态类层次与模板实例之间的最重要的区别是，通常在编译时整个模板都是可用的。在大多数用例下，C++ 都会内联函数调用，用多种方法改善程序性能（正如 7.2.2 节所指出的）。

模板编程提供了一种强力的优化手段。对于那些不熟悉模板的开发人员来说，需要学习如何高效地使用 C++ 的这个特性。

**7.2.7　避免使用 PIMPL 惯用法**

PIMPL 是“Pointer to IMPLementation”的缩写，它是一种用作**编译防火墙** ——一种防止修改一个头文件会触发许多源文件被重编译的机制——的编程惯用法。20 世纪 90 年代是 C++ 的快速成长期，在那时使用 PIMPL 是合理的，因为在那个年代，大型程序的编译时间是以小时为单位计算的。下面是 PIMPL 的工作原理。

假设 BigClass （代码清单 7-13）是一个被其他类广泛使用的类，它有一些内联函数，而且使用了 Foo 类、Bar 类和 Baz 类。一般情况下，bigclass.h、foo.h、bar.h 或是 baz.h 的任何改动，哪怕只是代码注释中的一个字符发生了变化，都会触发许多引用了 bigclass.h 的文件被重编译。

**代码清单 7-13** 　实现 PIMPL 惯用法之前的 bigclass.h

# include "foo.h" # include "bar.h" # include "baz.h" class BigClass { public: BigClass(); void f1(int a) { ... } void f2(float f) { ... } Foo foo\_; Bar bar\_; Baz baz\_; };

要实现 PIMPL，开发人员要定义一个新的类，在本例中，我们将其命名为 Impl 。bigclass.h 的修改如代码清单 7-14 所示。

**代码清单 7-14** 　实现 PIMPL 惯用法之后的 bigclass.h

class Impl; class BigClass { public: BigClass(); void f1(int a); char f2(float f); Impl\* impl; };

C++ 允许声明一个指向未完成类型，即一个还没有定义的对象的指针。在本例中，Impl 就是一个未完成类型。这样的代码之所以能够工作，是因为所有指针的大小都是相同的，因此编译器知道如何预留指针的存储空间。在实现 PIMPL 后，BigClass 的对外可见的定义不再依赖 foo.h、bar.h 或 baz.h 了。在 bigclass.cpp 中有 Impl 的完整定义（代码清单 7-15）。

**代码清单 7-15** 　包含 Impl 的定义的 bigclass.cpp

# include "foo.h" # include "bar.h" # include "baz.h" # include "bigclass.h" class Impl { void g1(int a); void g2(float f); Foo foo\_; Bar bar\_; Baz baz\_; }; void Impl::g1(int a) { ... } char Impl::g2(float f) { ... } void BigClass::BigClass() { impl\_ = new Impl; } void BigClass::f1(int a) { impl\_ -> g1(a); } char BigClass::f2(float f) { return impl\_ -> g2(f) }

实现了 PIMPL 后，在编译时，对 foo.h、bar.h 或 baz.h，或者是对 Impl 的实现的改动都会导致 bigclass.cpp 被重编译，但是 bigclass.h 不会改变，这样就限制了重编译的范围。

在运行时情况就不同了。PIMPL 给程序带来了延迟。之前 BigClass 中的成员函数可能会被内联，而现在则会发生一次成员函数调用。而且，现在每次成员函数调用都会调用 Impl 的成员函数。使用了 PIMPL 的工程往往会在很多地方使用它，导致形成了多层嵌套函数调用。更甚者，这些额外的函数调用层次使得调试变得更加困难。

2016 年，PIMPL 已经不是必需的了，因为编译时间可能已经减少至了 20 世纪 90 年代的 1%。而且，即使是在 20 世纪 90 年代，也只有当 BigClass 是一个非常大的类，依赖于许多头文件时，才需要使用 PIMPL。这样的类违背了许多面向对象编程原则。采用将 BigClass 分解，使接口功能更加集中的方法，可能与 PIMPL 同样有效。

**7.2.8　移除对 DDL 的调用**

在 Windows 上，当 DLL 被按需加载后在程序中显式地设置函数指针，或是在程序启动时自动地加载 DLL 时隐式地设置函数指针，然后通过这个函数指针调用**动态链接库** （dynamic link library，DDL）。Linux 上也有动态链接库，实现也是相同的。

有些 DLL 调用是必需的。例如，应用程序可能需要实现第三方插件库。其他情况下，DLL 则不是必需的。例如，有时之所以使用 DLL 仅仅是因为它们修复了一些 bug。经验证明 bug 修复通常都是批量的，一次性覆盖了程序中的各个地方。这限制了在一个 DLL 中修复所有 bug 的可能性，破坏了 DLL 的用途。

另外一种改善函数调用性能的方式是不使用 DLL，而是使用对象代码库并将其链接到可执行程序上。

**7.2.9　使用静态成员函数取代成员函数**

每次对成员函数的调用都有一个额外的隐式参数：指向成员函数被调用的类实例的 this 指针。通过对 this 指针加上偏移量可以获取类成员数据。虚成员函数必须解引 this 指针来获得虚函数表指针。

有时，一个成员函数中的处理仅仅使用了它的参数，而不用访问成员数据，也不用调用其他的虚成员函数。在这种情况下，this 指针没有任何作用。

我们应当将这样的成员函数声明为静态函数。静态成员函数不会计算隐式 this 指针，可以通过普通函数指针，而不是开销更加昂贵的成员函数指针找到它们（请参见 7.2.1 节中的“函数指针的开销”）。

**7.2.10　将虚析构函数移至基类中**

任何有继承类的类的析构函数都应当被声明为虚函数。这是有必要的，这样 delete 表达式将会引用一个指向基类的指针，继承类和基类的析构函数都会被调用。

另外一个在继承层次关系顶端的基类中声明虚函数的理由是：确保在基类中有虚函数表指针。

继承层次关系中的基类处于一个特殊的位置。如果在这个基类中有虚成员函数声明，那么虚函数表指针在其他继承类中的偏移量是 0；如果这个基类声明了成员变量且没有声明任何虚成员函数，但是有些继承类却声明了虚成员函数，那么每个虚成员函数调用都会在 this 指针上加上一个偏移量来得到虚函数表指针的地址。确保在这个基类中至少有一个成员函数，可以强制虚函数表指针出现在偏移量为 0 的位置上，这有助于产生更高效的代码。

而析构函数则是最佳候选。如果这个基类有继承类，它就必须是虚函数。在类实例的生命周期中析构函数只会被调用一次，因此只要不是那些在程序中会被频繁地构造和析构的非常小的类（而且通常情况下，几乎不会让这些小的类去继承子类），将其设置为虚函数后产生的开销是最小的。

这看似是非常罕见的情况，不需要太过关注，不过我参与过的几个项目中都存在这种情况：在重要类层次的基类中有引用计数、事务 ID 或者其他类似的变量。这个基类对可能继承它的类没有任何了解。通常，类层次关系中的第一个类都是一个声明了一组虚成员函数的抽象基类。基类肯定知道的一件事情就是实例最终会被销毁。

**7.3　优化表达式**

在语句级别下面是涉及基本数据类型（整数、浮点类型和指针）的数学计算。这也是最后的优化机会。如果一个热点函数中只有一条表达式，那么它可能是唯一的优化机会。

**停下来思考**

现代编译器非常善于优化涉及基本数据类型的表达式。从其他所有方面到性能优化，它们都非常擅长。但是它们不够勇敢。只有当它们能够确保改动不会影响程序行为时，才会进行优化表达式。

开发人员尽管没有编译器那么细致，但是比编译器更聪明。开发人员能够优化那些编译器无法确定优化是否安全的代码，因为开发人员可以推断设计和在其他代码模块中定义的函数的意图，而编译器则看不见这些。

因此，在这种非常罕见的情况下，开发人员可以比编译器做得更好。

优化表达式在每次只执行一条指令的小型处理器上有很好的效果。在桌面级的具有多段流水线的处理器中，虽然也可以测试到有改进效果，但并不明显。因此，并不太值得在这里投入大量精力进行优化。只有在那些必须通过热点循环或者函数再最后提升一点性能的极其罕见的情况下才需要这么做。

**7.3.1　简化表达式**

C++ 会严格地以运算符的优先级和可结合性的顺序来计算表达式。只有像 ((a\*b)+(a\*c)) 这样书写表达式时才会进行 a\*b+a\*c 的计算，因为 C++ 的优先级规则规定乘法的优先级高于加法。C++ 编译器绝对不会使用分配律将表达式重新编码为像 a\*(b+c) 这样的更高效的形式。只有像 ((a+b)+c) 这样书写表达式才会进行 a+b+c 的计算，因为 + 运算符具有左结合性。编译器绝对不会重写表达式为 (a+(b+c)) ，尽管在进行整数和实数数学计算时其结果并不会发生改变。

C++ 之所以让程序员手动优化表达式，是因为 C++ 的 int 类型的模运算并非是整数的数学运算，C++ 的 float 类型的近似计算也并非真正的数学运算。C++ 必须给予程序员足够的权力来清晰地表达他的意图，否则编译器会对表达式进行重排序，从而导致控制流程发生各种变化。这意味着开发人员必须尽可能使用最少的运算符来书写表达式。

用于计算多项式的霍纳法则（Horner Rule）证明了以一种更高效的形式重写表达式有多么厉害。尽管大多数 C++ 开发人员并不会每天都进行多项式计算，但是我们都很熟悉它。

多项式 *y* = *ax* 3 + *bx* 2 + *cx* + *d* 在 C++ 中可以写为：

y = a\*x\*x\*x + b\*x\*x + c\*x + d;

这条语句将会执行 6 次乘法运算和 3 次加法运算。我们可以根据霍纳法则重复地使用分配律来重写这条语句：

y = (((a\*x + b)\*x) + c)\*x + d;

这条优化后的语句只会执行 3 次乘法运算和 3 次加法运算。通常，霍纳法则可以将表达式的乘法运算次数从 *n* (*n* -1) 减少为 *n* ，其中 *n* 是多项式的维数。

**a / b \* c ：警示故事**

C++ 之所以不会重排序算术表达式是因为这非常危险。数值分析是一个非常大的主题，围绕这个主题可以写出一本书。下面是一个我们容易弄错的例子。

如果表达式被写作 ((a / b) \* c) ，那么 C++ 编译器会 a / b \* c 这样进行计算。但这个表达式存在一个问题。如果 a 、b 和 c 都是整数类型，那么 a/b 的结果将不精确。所以，如果 a = 2 、b = 3 且 c = 10 ，那么 a / b \* c 的结果就是 2 / 3 \* 10 = 0，而我们所期待的结果是 6。问题在于 a / b 的非精确性被放大了 c 倍，导致得出了一个错得离谱的结果。精通数学的开发人员可能会将表达式修改为 c \* a / b ，这样编译器在计算时就像是给表达式加上括号，使其变为 ((c \* a) / b) ，然后得出结果 2 \* 10 / 3 = 6。

问题解决了，对吗？实际上并没有。如果先做乘法，那么存在着溢出的风险。如果 a = 86 400（一天中的秒数）、b = 90 000（视频采样中使用的常量）且 c = 1 000 000（一秒中的微秒数），那么表达式 c \* a 会溢出 32 位无符号数据类型的范围。原来的表达式虽然计算误差很大，但是却比修改后的表达式要更好。

开发人员是唯一必须知道表达式的写法、它的参数的数量级并对其输出结果负责的人。编译器不会帮助我们完成这项任务，这也是它不会优化表达式的原因。

**7.3.2　将常量组合在一起**

编译器可以帮们做的一件事是计算常量表达式。请看下面这个表达式

seconds = 24 \* 60 \* 60 \* days;

或是

seconds = days \* (24 \* 60 \* 60);

编译器会计算表达式中的常量部分，产生类似下面的表达式：

seconds = 86400 \* days;

但是，如果程序员这样写：

seconds = 24 \* days \* 60 \* 60;

编译器只能在运行时进行乘法计算了。

因此，我们应当总是用括号将常量表达式组合在一起，或是将它们放在表达式的左端，或者更好的一种的做法是，将它们独立出来初始化给一个常量，或者将它们放在一个常量表达式（constexpr）函数中（如果你的编译器支持 C++11 的这一特性）。这样编译器能够在编译时高效地计算常量表达式。

**7.3.3　使用更高效的运算符**

有些数学运算符在计算时比其他运算符更低效。例如，如今，所有处理器（除了最小型的处理器）都可以在一个内部时钟周期中执行一次位移或是加法操作。某些专业的数字信号处理器芯片有单周期乘法器，但是对于 PC，乘法是一种类似于我们在小学学到的十进制乘法的迭代计算。除法是一种更复杂的迭代处理。这种开销结构为性能优化提供了机会。

例如，整数表达式 x\*4 可以被重编码为更高效的 x<<2 。任何差不多的编译器都可以优化这个表达式。但是如果表达式是 x\*y 或 x\*func() 会怎样呢？许多情况下，编译器都无法确定 y 或 func() 的返回值一定是 2 的幂。这时就需要依靠程序员了。如果其中一个参数可以用指数替换掉 2 的幂，那么开发人员就可以重写表达式，用位移运算替代乘法运算。

另一种优化是用位移运算和加法运算替代乘法。例如，整数表达式 x\*9 可以被重写为 x\*8+x\*1 ，进而可以重写为 (x<<3)+x 。当常量运算子中没有许多置为 1 的位时，这种优化最有效，因为每个置为 1 的位都会扩展为一个位移和加法表达式。在拥有指令缓存和流水线执行单元的桌面级或是手持级处理器上，以及在长乘法被实现为子例程调用的小型处理器上，这种优化同样有效。与所有性能优化方法一样，我们必须测试性能结果来确保在某种处理器上它确实提高了性能，但通常情况下确实都是这样的。

**7.3.4　使用整数计算替代浮点型计算**

浮点型计算的开销是昂贵的。浮点数值内部的表现比较复杂，它带有一个整数型尾数、一个独立的指数以及两个符号。PC 上实现了浮点型计算单元的硬件可能占到芯片面积的 20%。有些多核处理器会共享一个单独的浮点型计算单元，但是却在每个核心上都有多个独立的整数计算单元。

即使是在具有浮点型计算硬件单元的处理器上，即使对计算结果的整数部分进行了舍入处理，而不是截取处理，计算整数结果仍然能够比计算浮点型结果快至少 10 倍。如果是在没有浮点型计算硬件单元的小型处理器上用函数库进行浮点型计算，那么整数的计算速度会快得更多。但是我们仍然可以看到，有些开发人员在明明可以使用整数计算时，却使用浮点型计算。

代码清单 7-16 中展示了我遇到得最多的进行舍入操作的代码。它将整数参数转化为浮点类型，然后进行除法操作，最后对结果进行舍入操作。

**代码清单 7-16** 　对浮点类型进行舍入操作得到整数值

unsigned q = (unsigned)round((double)n / (double)d));

在我的 PC 上对该处理重复执行 1 亿次的测试结果是耗时 3125 毫秒。

要想得到舍入后的整数部分，需要先知道除法结果的余数。余数的取值范围是 0 至 *d* -1，其中 *d* 是除数。如果余数大于或等于除数的二分之一，那么整数部分应当向上舍入。对于有符号整数来说，这个公式会复杂一点点。

C++ 中提供了来自 C 运行时库的 ldiv() 函数，它会生成一种同时包含整数和余数的结构。代码清单 7-17 展示了一个使用 ldiv() 函数对除法结果进行舍入的函数。

**代码清单 7-17** 　使用 ldiv() 对整数除法进行舍入

inline unsigned div0(unsigned n, unsigned d) { auto r = ldiv(n, d); return (r.rem >= (d >> 1)) ? r.quot + 1 : r.quot; }

这个函数并不完美。ldiv() 接收整数类型的参数，但是调用时传递的确是 signed 或 unsigned 。当两个参数都是正数时，ldiv() 认为它们是整数，可以得到正确的结果。对 div0() 的测试结果是执行 1 亿次耗时 435 毫秒，比原来的浮点型版本快了 6 倍。

代码清单 7-18 展示了一个计算两个无符号参数的商在舍入后的结果的函数。

**代码清单 7-18** 　对整数除法的结果舍入

inline unsigned div1(unsigned n, unsigned d) { unsigned q = n / d; unsigned r = n % d; return r >= (d >> 1) ? q + 1 : q; }

div1() 会计算商和余数。(d >> 1) 是 d/2 的一种高效但威力稍弱的形式，它会计算出除数 d 的二分之一。如果余数大于或等于除数的一半，那么就会对商加 1。编译器所进行的一项优化是这个函数成功的关键。x86 机器对两个整数进行除法的指令会同时得到商和余数。 Visual C++ 编译器非常聪明，当执行这段代码时它只会执行一次这个指令。与浮点型计算测试一样，我对这个函数也进行了相同的测试，结果是耗时 135 毫秒，速度是原来的 22 倍，优化结果令人满意。

代码清单 7-19 是另外一种对 unsigned 舍入的方法，它更快，但也有代价。

**代码清单 7-19** 　对整数除法结果舍入

inline unsigned div2(unsigned n, unsigned d) { return (n + (d >> 1)) / d; }

div2() 在进行除法之前，在分子 n 上加上了除数 d 的二分之一。div2() 的缺点在于，如果分子很大，那么 n + (d >> 1) 可能会溢出。如果开发人员知道参数的数量级没有问题，那么就可以使用这个非常高效的 div2() 函数。对其进行测试的结果是耗时 102 毫秒（比那个常用的浮点型计算版本快了 30 倍）。

**7.3.5　双精度类型可能会比浮点型更快**

在我的 i7 PC 上运行 Visual C++ 时，双精度类型的计算速度比浮点类型的计算速度更快。首先我会向读者展示测试结果，接下来再推测为什么会出现这种现象。

下面这段代码会循环计算物体的下落距离，这是一段典型的浮点型计算：

float d, t, a = -9.8f, v0 = 0.0f, d0 = 100.0f; for (t = 0.0; t < 3.01f; t += 0.1f) { d = a\*t\*t + v0\*t + d0;

运行这段循环 1000 万次耗时 1889 毫秒。

将变量和常量类型修改为双精度后的代码如下：

double d, t, a = -9.8, v0 = 0.0, d0 = 100.0; for (t = 0.0; t < 3.01; t += 0.1) { d = a\*t\*t + v0\*t + d0;

运行此版本的循环 1000 万次只耗时 989 毫秒，几乎比之前快了一倍。

为什么会 出现 这种现 象呢？ Visual C++ 生成的浮 点型指令 会引 用老式 的“x87 FPU coprocessor”寄存器栈。在这种情况下，所有的浮点计算都会以 80 位格式进行。当单精度 float 和双精度 double 值被移动到 FPU 寄存器中时，它们都会被加长。对 float 进行转换的时间可能比对 double 进行转换的时间更长。

有多种编译浮点型计算的方式。在 x86 平台上，使用 SSE 寄存器允许直接以四种不同大小完成计算。使用了 SSE 指令的编译器的行为可能会与为非 x86 处理器进行编译的编译器不同。

**7.3.6　用闭形式替代迭代计算**

C++ 和位操作是怎样的呢？ C++ 中丰富的计算和位逻辑运算符只是将位移动来移动去，还是从设备寄存器和网络包获得信息位以及将信息位放到设备寄存器和网络包的需求使得 C++ 变为今天这个样子？

有许多特殊情况都需要对置为 1 的位计数，找到最高有效位，确定一个字的奇偶校验位，确定一个字的位是否是 2 的幂，等等。大多数这些问题都可以通过简单地遍历字中的所有位来解决。这种解决方法的时间开销为 *O* (*n* )，其中 *n* 是字的位数。也可能还有一些效率更高的迭代解决方法。但是对于某些问题，还有更快更紧凑的**闭形式** 解决方法：计算的时间开销为常量，不进行任何迭代。

例如，考虑一个简单的用于确定一个整数是否是 2 的幂的迭代算法。所有这些值都只有 1 个置为 1 的位，因此算出置为 1 的位的数量是一种解决方法。代码清单 7-20 展示了这种算法的一种简单的实现。

**代码清单 7-20** 　判断一个整数是否是 2 的幂的迭代算法的一种实现

inline bool is\_power\_2\_iterative(unsigned n) { for (unsigned one\_bits = 0; n != 0; n >>= 1) if ((n & 1) == 1) if (one\_bits != 0) return false; else one\_bits += 1; return true; }

对这种方式的测试结果是耗时 549 毫秒。

这个问题同样有一种闭形解决方法。如果 *x* 是 2 的 *n* 阶幂，那么它只在第 *n* 位有一个置为 1 的位（以最低有效位作为第 0 位）。接着，我们用 *x* -1 作为当置为 1 的位在第 *n* -1,…,0 位时的位掩码，那么 *x* & (*x* -1) 等于 0。如果 *x* 不是 2 的幂，那么它就有不止一个置为 1 的位，那么使用 *x* -1 作为掩码计算后只会将最低有效位置为 0，*x* & (*x* -1) 不再等于 0。

代码清单 7-21 展示了一个判断 *x* 是否是 2 的幂的闭形式的函数。

**代码清单 7-21** 　判断一个整数是否是 2 的幂的闭形式

inline bool is\_power\_2\_closed(unsigned n) { return ((n != 0) && !(n & (n - 1))); }

使用这个修改后的函数进行测试的结果是耗时 238 毫秒，比之前的版本快了 2.3 倍。其实还有更快的方法。Rick Regan 在他的网页（<http://www.exploringbinary.com/ten-ways-to-check-if-an-integer-is-a-power-of-two-in-c/> ）上记录了 10 种方法，而且都附有时间测量结果。

**买一本 *Hacker's Delight***

上节给出了一些高效进行位操作的建议，但这只能算是激发读者深入学习的例子，还有其他成百上千的改善计算性能的小技巧。

每位对优化表达式感兴趣的开发人员，其书柜里都应该有一本 Henry S. Warren, Jr 的 *Hacker's Delight* **4** ，现在它已经发行了第二版了。哪怕你对编写高效的表达式只有一丁点儿兴趣，阅读 *Hacker's Delight* 就像是打开了你的第一个乐高玩具箱或是使用分立元件制作你的第一个电路一样。Warren 同时还为这本书制作并维护了一个网站（<http://hackersdelight.org/> ），上面有很多有趣的链接和讨论。

想要免费了解 *Hacker's Delight* 一书中的部分内容，你可以在互联网上浏览 MIT 人工智能实验室 Memo 239——它也被亲切地叫作 HAKMEM**5** 。HAKMEM 诞生于最快的处理器的速度比现在的手机处理器还慢 10 000 倍的年代，它是 *Hacker's Delight* 一书的概念上的原型，里面介绍了许多位操作的技巧。

**4** 中文版《高效程序的奥秘》，冯速译，机械工业出版社，2004 年 5 月。——译者注

**5** Beeler, Michael, Gosper, R. William, and Schroeppel, Rich,“HAKMEM,”Memo 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1972.

**7.4　优化控制流程惯用法**

如在 2.2.7 节中所讲过的，由于当指令指针必须被更新为非连续地址时在处理器中会发生流水线停顿，因此计算比控制流程更快。C++ 编译器会努力地减少指令指针更新的次数。了解这些知识有助于我们编写更快的代码。

**7.4.1　用 switch 替代 if-else 　if-else**

if-else if-else 语句中的流程控制是线性的：首先测试 if 条件，如果结果为真，执行第一个代码块；否则，接着测试 else if 条件，如果为真，则执行该条件所匹配的代码块。

如果测试一个变量的值 *n* 次，那么需要 *n* 个 if-then-else if 语句块。如果所有的条件为真的概率都是一样的，那么 if-then-else if 将会进行 *O* (*n* ) 次判断。如果这段代码执行得非常频繁，例如在事件分发代码或是指令分发代码中，那么开销将会显著地增加。

switch 语句也会测试一个变量是否等于这 *n* 个值，但是由于 switch 语句的形式比较特殊，它用 switch 的值与一系列常量进行比较，这样编译器可以进行一系列有效的优化。

一种常见的情况是被测试的常量是一组连续值或是近似一组连续值，这时 switch 语句会被编译为 jump 指令表，其索引是要测试的值或是派生于要测试的值的表达式。switch 语句会执行一次索引操作，然后跳转到表中的地址。无论有多少种要比较的情况，每次比较处理的开销都是 *O* (1)。我们在程序中不必对各种要比较的情况排序，因为编译器会排序 jump 指令表。

如果这些被测试的常量不是连续值，而是互相之间相差很大的数值，那么 jump 指令表会变得异常庞大，难以管理。编译器可能仍然会排序这些要测试的常量并生成执行二分查找的代码。对于一个会与 *n* 个值进行比较的 switch 语句，这种查找的最差情况的开销是 *O* (log2 *n* )。在任何情况下，编译器编译 switch 语句后产生的代码都不会比编译 if-then 语句后产生的代码的速度慢。

有时，if-elseif-else 逻辑的某个条件分支的可能性非常大。在这种情况下，如果首先测试最可能出现的条件的话，if 语句的摊销性能可能会接近常量。

**7.4.2　用虚函数替代 switch 或 if**

在 C++ 出现之前，如果开发人员想要在程序中引入多态行为，那么他们必须编写一个带有标识变量的结构体或是联合体，然后通过这个标识变量来辨别出当前使用的是哪个结构体或是联合体。程序中应该会有很多类似下面的代码：

if (p->animalType == TIGER) { tiger\_pounce(p->tiger); } else if (p->animalType == RABBIT) { rabit\_hop(p->rabbit); } else if (...)

经验丰富的开发人员都知道这个反模式是面向对象编程的典型代表。但是新手开发人员要想熟练掌握面向对象思想是需要时间的。我在很多软件产品中看到过下面这样不纯粹的面向对象的 C++ 代码：

Animal::move() { if (this->animalType == TIGER) { pounce(); } else if (this->animalType == RABBIT) { hop(); } else if (...) ... }

从性能优化的角度看，这段代码的问题在于使用了 if 语句来识别对象的继承类型。C++ 类已经包含了一种机制来实现此功能：虚成员函数和作为识别器的虚函数表指针。

虚函数调用会通过索引虚函数表得到虚函数体的地址。这个操作的开销总是常量时间。因此，基类中的虚成员函数 move() 会被继承类中表示各种动物的 pounce 、hop 或 swim 等函数重写。

**7.4.3　使用无开销的异常处理**

异常处理是应当在设计阶段就进行优化的项目之一。错误传播方法的设计会影响程序中的每一行代码，因此改造程序的异常处理的代价可能会非常昂贵。可以说，使用异常处理可以使程序在通常运行时更加快速，在出错时表现得更加优秀。

有些开发人员对 C++ 的异常处理持有怀疑态度。一般认为，异常处理会使程序变得更加庞大和更加慢，因此关闭编译器的异常处理开关是一项优化。

其实真相比较复杂。确实，如果程序不使用异常处理，那么关闭编译器的异常处理开关可以使得程序变得更小，而且可能更快。Jeff Preshing 在他的博客（<http://preshing.com/20110807/the-cost-of-enabling-exception-handling/> ）中发表了文章，说测量到性能差距在 1.4% 和 4% 之间。但是不清楚不使用异常处理的程序的运行状况如何。C++ 标准库中的所有容器都使用 new 表达式来抛出异常。许多其他库，包括本书中会讲解的流 I/O 和并发库（请参见第 12 章）都会抛出异常。dynamic\_cast 运算符也会抛出异常。如果关掉了异常处理，无法确定当程序遇到异常被抛出的情况时会如何。

如果程序不抛出异常，它可能会完全忽略错误码。那么在这种情况下，开发人员就会得到报应了。另外一种情况是，程序必须在各层函数调用之间耐心地、小心地传递错误码，然后在调用库函数的地方将错误码从一种形式转换为另一种形式并相应地释放资源。而且，无论每次运算是成功还是失败，都不能遗漏这些处理。

如果有异常，处理错误的部分开销就被从程序执行的正常路径转移至错误路径上。除此之外，编译器会通过调用在抛出异常和 try/catch 代码块之间的执行路径上的所有自动变量的析构函数，自动地回收资源。这简化了程序执行的正常路径的逻辑，从而提升性能。

在 C++ 的早期，每个栈帧都包含一个异常上下文：一个指向包含所有被构建的对象的链表的指针，因此当异常穿过栈帧被抛出时，这些对象也必须被销毁。随着程序的执行，这个上下文会被动态地更新。这并非大家所希望看到的，因为这导致了在程序执行的正常路径上增加了运行时开销。这可能会是高开销的异常处理之源。后来出现了一种新的实现方式，它的原理是将那些需要被销毁的对象映射到指令指针值上。除非抛出了异常，否则这种机制不会发生任何运行时开销。Visual Studio 会在构建 64 位应用程序时使用这种无开销机制，而在构建 32 位应用程序时则会使用旧机制。Clang 则提供了一个编译器选项 **6** 让开发人员选择使用哪种机制。

**6** Clang 可以使用 /EH 指定异常处理模型。——译者注

**不要使用异常规范**

异常规范是对函数声明的修饰，指出函数可能会抛出什么异常。不带有异常规范的函数抛出异常可能不会有任何惩罚。而带有异常规范的函数可能只会抛出在规范中列出的异常。但是如果它抛出了其他异常，那么程序会被 terminate() 无条件地立即终止。

异常规范有两个问题。一个问题是开发人员很难知道被调用的函数可能会抛出什么异常，特别是在使用不熟悉的库时。这使得使用了异常规范的程序变得脆弱且可能会突然停止。

第二个问题是异常规范对性能有负面影响。程序必须要检查被抛出的异常，就像是每次对带有异常规范的函数的调用都进入了一个 try/catch 代码块一样。

C++11 弃用了传统的异常规范。

在 C++11 中引入了一种新的异常规范，称为 noexcept 。声明一个函数为 noexcept 会告诉编译器这个函数不可能抛出任何异常。如果这个函数抛出了异常，那么如同在 throw() 规范中一样，terminate() 将会被调用。不过不同的是，编译器要求将移动构造函数和移动赋值语句声明为 noexcept 来实现移动语义（请参见 6.6 节中有关移动语义的讨论）。在这些函数上的 noexcept 规范的作用就像是发表了一份声明，表明对于某些对象而言，移动语义比强异常安全保证更重要。我知道这非常晦涩。

**7.5　小结**

* 除非有一些因素放大了语句的性能开销，否则不值得进行语句级别的性能优化，因为所能带来的性能提升不大。
* 循环中的语句的性能开销被放大的倍数是循环的次数。
* 函数中的语句的性能开销被放大的倍数是其在函数中被调用的次数。
* 被频繁地调用的编程惯用法的性能开销被放大的倍数是其被调用的次数。
* 有些 C++ 语句（赋值、初始化、函数参数计算）中包含了隐藏的函数调用。
* 调用操作系统的函数的开销是昂贵的。
* 一种有效的移除函数调用开销的方法是内联函数。
* 现在几乎不再需要 PIMPL 编程惯用法了。如今程序的编译时间只有发明 PIMPL 的那个年代的 1% 左右。
* double 计算可能会比 float 计算更快。