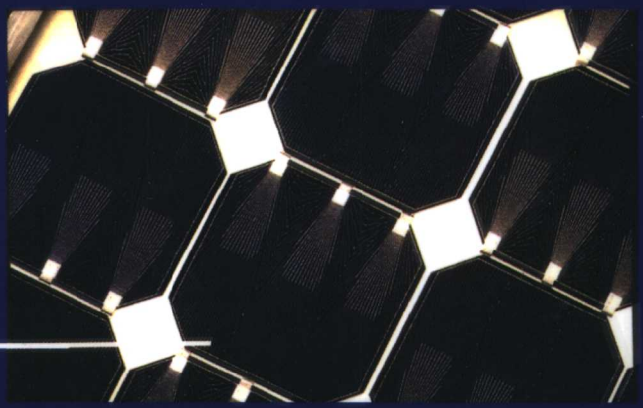
PEARSON
Addison Wesley

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



Pearson Education
清华大学出版社

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

序 …… 8

引言 …… 10

第 1 章　跟踪范例 …… 1

1.1　初步的跟踪实现 …… 2

1.1.1　发生了什么问题 …… 4

1.1.2　恢复计划 …… 6

1.2　要点 …… 9

第 2 章　构造函数和析构函数 …… 10

2.1　继承 …… 11

2.2　合成 …… 21

2.3　缓式构造 …… 23

2.4　冗余构造 …… 26

2.5　要点 …… 30

第 3 章　虚函数 …… 31

3.1　虚函数的构造 …… 31

3.2　模板和继承 …… 34

3.2.1　硬编码 …… 35

3.2.2　继承 …… 36

3.2.3　模板 …… 37

3.3 要点…… 38

第4章 返回值优化 …… 39
4.1 按值返回的构造…… 39
4.2 返回值优化…… 41
4.3 计算性构造函数…… 44
4.4 要点…… 45

第5章 临时对象 …… 46
5.1 对象定义…… 46
5.2 类型不匹配…… 47
5.3 按值传递…… 50
5.4 按值返回…… 51
5.5 使用 op=（）消除临时对象…… 53
5.6 要点…… 54

第6章 单线程内存池 …… 55
6.1 版本 0：全局函数 new（）和 delete（） …… 55
6.2 版本 1：专用 Rational 内存管理器 …… 57
6.3 版本 2：固定大小对象的内存池 …… 61
6.4 版本 3：单线程可变大小内存管理器 …… 65
6.5 要点…… 72

第7章 多线程内存池 …… 73
7.1 版本 4：实现 …… 73
7.2 版本 5：快速锁定 …… 76
7.3 要点…… 80

第8章 内联基础 …… 81
8.1 什么是内联…… 81
8.2 方法调用代价…… 85
8.3 为何使用内联…… 89

8.4 内联详述 …… 90
8.5 内联虚方法 …… 91
8.6 通过内联获得性能 …… 92
8.7 要点 …… 93

第 9 章 内联——性能方面的考虑 …… 94
9.1 调用间优化 …… 94
9.2 为何不使用内联 …… 99
9.3 开发阶段和编译时的内联考虑 …… 102
9.4 基于配置的内联 …… 102
9.5 内联规则 …… 106
9.5.1 惟一 …… 106
9.5.2 微小 …… 106
9.6 要点 …… 107

第 10 章 内联技巧 …… 108
10.1 条件内联 …… 108
10.2 选择性内联 …… 109
10.3 递归内联 …… 111
10.4 对静态局部变量进行内联 …… 115
10.5 与体系结构有关的注意事项：多寄存器集 …… 117
10.6 要点 …… 118

第 11 章 标准模板库 …… 119
11.1 渐近复杂度 …… 119
11.2 插入 …… 120
11.3 删除 …… 127
11.4 遍历 …… 130
11.5 查找 …… 131
11.6 函数对象 …… 133
11.7 比 STL 更好 …… 135

11.8 要点 …… 138

第 12 章 引用计数 …… 139

12.1 实现细节 …… 141

12.2 已存在类 …… 153

12.3 并发引用计数 …… 157

12.4 要点 …… 161

第 13 章 代码优化 …… 162

13.1 缓存 …… 164

13.2 预先计算 …… 164

13.3 降低灵活性 …… 166

13.4 80-20 规则：提高常用路径的速度 …… 166

13.5 缓式计算 …… 170

13.6 无用计算 …… 171

13.7 系统体系结构 …… 172

13.8 内存管理 …… 174

13.9 库和系统调用 …… 175

13.10 编译器优化 …… 177

13.11 要点 …… 178

第 14 章 设计优化 …… 179

14.1 设计灵活性 …… 179

14.2 缓存 …… 183

14.2.1 Web 服务器时间戳 …… 183

14.2.2 数据扩展 …… 183

14.2.3 公用代码陷阱 …… 184

14.3 高效的数据结构 …… 186

14.4 缓式计算 …… 186

14.5 无用计算 …… 190

14.6 失效代码 …… 191

14.7 要点 …… 192

第 15 章 可伸缩性 …… 193
15.1 SMP 体系结构 …… 195
15.2 Amdahl 法则 …… 196
15.3 多线程和同步术语 …… 198
15.4 把一个任务分解成多个子任务 …… 199
15.5 缓存共享数据 …… 199
15.6 无共享 …… 202
15.7 部分共享 …… 203
15.8 锁的粒度 …… 205
15.9 伪共享 …… 208
15.10 Thundering Herd …… 208
15.11 读/写锁 …… 210
15.12 要点 …… 210

第 16 章 系统体系结构相关性 …… 212
16.1 内存层次 …… 212
16.2 寄存器:内存之王 …… 214
16.3 磁盘和内存结构 …… 217
16.4 缓存影响 …… 220
16.5 缓存颠簸 …… 222
16.6 避免跳转 …… 223
16.7 简单计算胜过小分支 …… 224
16.8 线程影响 …… 225
16.9 上下文切换 …… 227
16.10 内核交叉 …… 229
16.11 线程选择 …… 230
16.12 要点 …… 232

参考文献 …… 233

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，addOne ( )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。

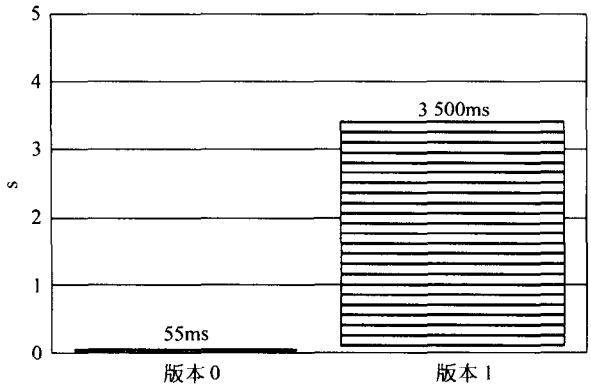


图 1.1 Trace 对象的性能开销

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)。

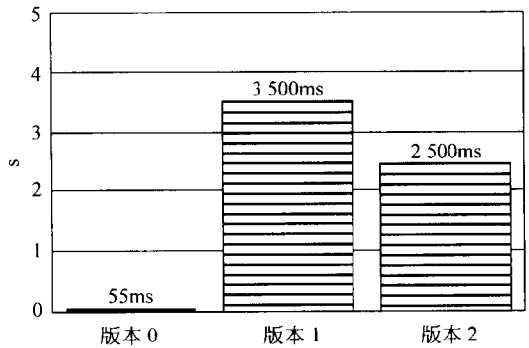


图 1.2 取消一个 string 对象后的影响

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)。

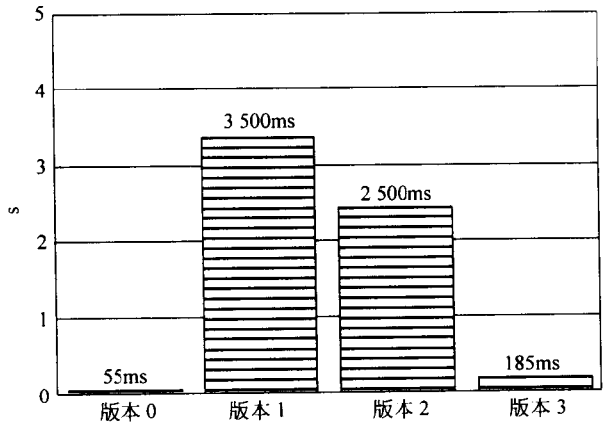


图 1.3 有条件创建 string 成员的影响

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。