效率

Item 16 谨记80-20法则

> 一个程序80%的资源用于20%的代码身上。这不只是一个动人的口号而已，它是系统性能议题上的一个准则，有广大的应用价值和坚实的实证基础。

> 大部分程序员的程序的性能特质，都有错误的直觉，因为程序的性能特质倾向高度的非直觉性。结果，无数的努力关注在一些绝对无法提升整体性能的而程序段落上头，形成严重的精力浪费。

> 如果你的软件够快，没有人会在乎你执行爱多语句；如果你的软件过慢，也没有人会因为你执行了很少的语句而原谅你。他们所在乎的只是：他们讨厌等待，而如果你的程序让他们等待，他们就会讨厌你。

当然知道语句被执行或函数被调用的频繁都，有时候可以让你对你的软件行为有更深刻的了解。此外，语句的执行次数和函数的调用次数可以间接了解你无法直接测量的软件行为。

数据是最好测试的方法，但是不要被单个数据蒙蔽双眼，这不是算法考试，必须要以最优的最坏复杂度取胜。但是不论如何，最佳办法是尽可能以最多的数据来分析你的软件，这些数据要尽力保证可以重现。

Item 17 考虑使用lazy evaluation

这一个骚操作来自于生活，能拖就尽量拖下去。这个方法遵守的是只计算必要的部分，不必要的部分尽力去除。

Reference Counting（引用计数）

赋值操作时，只需要把东西绑在另一个东西上即可，不需要立即进行复制操作，在进行修改的时候才需要对新变量进行松绑，因为两个东西的值已经不一样了。总的来讲，在你真正需要之前，这个东西对于你没有意义。

区分读和写

```
string s = "Homer's Iliad";
// ...
cout << s[3];
s[3] = 'x';
```

第一个`operator[]`调用的是读取，第二个则是写入。虽然我们可以说啊只有读取到修改过的地方时才有必要把修改放进去，但是实际上在调用`operator[]`的时候，我们无法区分两者。在后续的实现中可能会有所改观。

Lazy Fetching（缓式取出）

假如我们的对象管理的是大型对象，或者需要远程连接、大量IO等等操作，此时对象管理不需要一次性把所有的都拿过来，这种等待会令客户炸毛，所以这时建议用指针替代具体对象，在没有访问时从赋值`nullptr`，需要访问时附上具体的值。

考虑到读取的方法通常会被设置为`const`方法，返回值也是`const`的，此时不允许修改内部的值，也不允许调用非`const`方法，此时最好的解决方案是给成员变量加上`mutable`关键字，或者从另一种角度考虑，上`const_cast`或者强制类型转换，虽然个人更倾向于`const_cast`。这里的指针通常也包括包装后的指针。

Lazy Expression Evaluation（表达式缓评估）

拖延战术通常还用在具体运算中，比如

```
template<class T>
class Matrix { /* ... */ };
Matrix<int> m1(1000,1000),m2(1000,1000);
// ...
Matrix<int> m3 = m1 + m2;
```

如果要立即做的话，复杂度可能还可以接受，但是应该会有大量不必要做的运算吧。如果觉得加法还行的话，乘法呢？$O(n^3)$的复杂度不是开玩笑的。

此时只处理那一行、那一列或者单独一个元素，岂不是真香？不然在其的科学计算软件是怎么撑过来的？虽然当下计算机跑得更快了，但是数据量更大了，客户也更没有耐心了，所以当代很多的矩阵程序库也有Lazy evaluation。

Item 18 分期摊还预期的计算成本

有很多时候，能不做的事情就不要去做，但是也有时候，该做的事情就一定要做，特别是能顺手做了的和预期会一直要做的。但是不论选择哪一种，都是为了——减少等待，减少不必要的操作，平摊时间。

- 比如求一个数据结构的最大最小或者平均等等之类的变量，只要有这方面的需求，能够平摊成本，岂不是很香？
- 磁盘缓存的设计中，如果某处的数据被需要，那么邻近的数据也会被需要，于是乎就有预先取出这个说法，在各个级别的都可以见到。
- 动态内存分配。分配超额内存时，不要求直接分配这么多，而是要求多分配超出部分的两倍内存，因为要很多内存也有理由要更多内存。

空间可以交换时间，这个历史不仅仅和计算机一样古老，在各行各业，甚至从生物界都可以有很多证明。

Item 19 了解临时对象的来源

首先要明确规定的是，临时对象是指没有变量名的，但是在变量转移过程中不得不进行赋值的变量。如果在函数运行过程中定义了所谓的“临时变量”，那么一个更合适的叫法其实是“局部变量”。只要产生一个non-heap object对象而没有为它命名，便产生了一个临时对象。这类通常发生于两种情况：一是隐式类型转换被实行以求函数调用能够成功；二是函数返回对象时。

最常见的是把`char*`转换到`const string&`的函数。

```
size_t countChar(const string& str,char ch);
countChar("huajihuaji",'j');
```

因为C++是一个强类型语言，在不是完全匹配的情况下（先不考虑`template`），编译器会优先考虑进行类型转换，产生临时变量之后进入函数。在函数执行完以后，临时变量会被销毁。对于这种可能是不必要的花费，建议重新设计代码使得转换不会发生，要么重新设计软件，让这类转换不再需要。

如果传递方式是by value或者by reference-to-`const`时，这类转换才会发生。如果对象被传递给一个reference-to-non-`const`，并不会发生此类转换，此时传入其他类型或者一个非左值就会报错。这是基于reference的初衷来决定的，程序员把东西传入引用时，绝大部分都是希望对象能被修改，而修改需要直到这东西放在哪里，而且要稳定存在（特别是不是右值）。如果传入一个不是同类的东西，修改又如何谈起？

除此以外，函数的返回值也经常会产生临时对象，比如`operator+`的返回值通常是`const T`，此时就会产生临时对象的复制：运算产生临时对象，临时对象放入寄存器，然后又复制到外面去。通常来说只能特殊情况特殊处理，比如`operator+`可以改成`operator+=`，但是更多时候，编译器会帮你减少这类对象复制的频率，这种方法被称为“优化”。

临时对象可能很耗成本，所以你应该尽可能消除它们。然而更重要的是，如何训练出锐利的眼力，看出可能产生临时对象的地方。任何时候只要你看到reference to-`const`参数，就极有可能产生一个临时对象以绑定到这个参数上。任何时候只要看到函数返回一个对象，就会产生临时对象。学习找出这些架构，你对幕后成本的洞察力就会有显著的提升。

Item 20 协助完成“返回值优化”

通常来说返回一个对象总是会带来成本，这极有可能会引发你的强迫症去优化掉这个东西。比如在经常举例的Rational中：

```
1 const Rational operator*(const Rational& lhs, const Rational& rhs){
2     return Rational(lhs.numerator() * rhs.numerator(), lhs.denumerator() *
  rhs.denumerator());
3 }
```

如果返回值是一个`const Rational&`，那么会得到一个很尴尬的结果：你的操作符拿不回一个有效的对象，因为那个结果在函数执行完了以后就消失了。当然一个更扯淡的方法时返回`const Rational*`，别说了你自己用用就知道有多扯了。

如果说你真的希望优化的话，建议上`inline`，当然不要对大程序这样子搞，CPU翻页翻来翻去也是很痛苦的。如果你比较佛性的话，那还是算了，编译器帮你打点好一切，通常来说编译器也不会在这里坑你。所以不要过于追求细枝末节上的效率，除非你真的希望自己打点所有东西，不然你想得到的别人为啥想不到？

Item 21 利用重载技术避免隐式转换

Item 22 考虑以操作符复合形式取代其独身形式

Item 23 考虑使用其他程序库

Item 24 了解virtual functions、multiple inheritance、virtual base classes、runtime type identification的成本