第16条：记住80-20法则。

在这些任务中，更麻烦的通常是定位瓶颈。有两种根本不同的方法来处理这个问题：大多数人的方法和正确的方法。

然而，并不是所有的分析师都能做到这一点。你想要一个直接测量你感兴趣的资源的方法。例如，如果您的程序太慢，您需要一个分析器来告诉您程序的不同部分花费了多少时间。这样，您就可以专注于那些地方，在这些地方，局部效率的显著提高也会带来整体效率的显著提高。

请记住，分析器只能告诉您程序在特定运行（或一组运行）中的行为，因此如果您使用不具有代表性的输入数据分析程序，您将得到同样不具有代表性的配置文件。反过来，这可能会导致您针对不常见的用途优化软件的行为，对常见用途的总体影响甚至可能是负面的。

您必须确保每个数据集都代表软件的客户端（或至少是最重要的客户端）是如何使用软件的。通常很容易获得代表性的数据集，因为许多客户很乐意让您在分析时使用他们的数据。毕竟，您将调整您的软件以满足他们的需求，这对您和他们都有好处。

第17条：考虑使用惰性评估。

当你使用惰性计算时，你以这样一种方式编写你的类，它们推迟计算，直到需要这些计算的结果。如果不需要结果，计算就不会执行，软件的客户端和父母都不会知道。

* **写时复制（COW）** 是一种优化技术，通过共享数据减少复制开销。
* **共享数据对客户端透明**，只有在修改数据时才需要复制。
* **适用于读多写少的场景**，但在频繁修改数据的场景中可能不适用。
* **懒惰的方法可以节省资源**，但在需要修改数据时必须付出复制的代价。

区分读和写

惰性抓取

**惰性抓取（Lazy Fetch）** 是一种编程技术，主要用于延迟加载数据或资源，直到真正需要时才进行加载。这种技术的核心思想是避免在程序初始化或对象创建时立即加载所有可能用到的数据，而是等到数据被实际访问或使用时才进行加载。惰性抓取通常用于优化性能，特别是在处理大量数据或资源密集型操作时。

惰性表达式求值

1. **急切实值 vs 惰性求值**：
   * **急切实值（Eager Evaluation）**：立即执行计算并返回结果。例如，m3 = m1 + m2 会立即计算 m1 和 m2 的和，并存储结果。
   * **惰性求值（Lazy Evaluation）**：推迟计算，直到真正需要结果时才执行。例如，m3 = m1 + m2 只会记录 m3 是 m1 和 m2 的和，而不会立即计算。
2. **惰性求值的实现**：
   * 在惰性求值中，m3 不会立即存储计算结果，而是存储一个数据结构，记录 m3 是 m1 和 m2 的和。
   * 这个数据结构可能只包含指向 m1 和 m2 的指针，以及一个表示操作类型（如加法）的枚举值。
3. **惰性求值的优势**：
   * **节省计算资源**：如果 m3 从未被使用，那么计算 m1 + m2 的开销就被完全避免了。
   * **节省内存**：不需要立即分配存储结果的内存。
   * **动态优化**：如果 m3 的定义被覆盖（如 m3 = m4 \* m1），之前的计算可以被完全丢弃，避免不必要的计算。

第18条：摊销预期计算的成本。

文本

描述已自动生成

我在这篇文章中提出的建议--通过缓存和预取等过于急切的策略来分摊预期计算的成本--与我在第17篇中提出的关于懒惰计算的建议并不矛盾。惰性计算是一种技术，当您必须支持其结果并不总是需要的操作时，它可以提高程序的效率。过度急切求值是一种提高程序效率的技术，当您必须支持几乎总是需要其结果或其结果经常需要多次的操作时。这两种方法都比一般的急切评估更难实现，但它们都可以在程序中产生显着的性能改进，这些程序的行为特征证明了额外的编程工作是合理的。

项目19：理解临时对象的起源。

在C++中，真正的临时对象是不可见的--它们不会出现在你的源代码中。当一个非堆对象被创建但没有命名时，它们就会出现。这种未命名对象通常出现在以下两种情况之一时：当应用隐式类型转换以使函数调用成功时，以及当函数返回对象时。理解如何以及为什么创建和销毁这些临时对象是很重要的，因为它们的构造和销毁所伴随的成本会对程序的性能产生明显的影响。

* 隐式类型转换在按值传递和常量引用参数时是允许的。
* 非常量引用参数禁止隐式类型转换，以避免意外的行为。
* 常量引用参数允许隐式类型转换，因为临时对象是只读的。

任何时候你看到一个reference-to-const参数，都有可能会创建一个临时参数来绑定到这个参数。任何时候你看到一个函数返回一个对象，一个临时的将被创建（稍后被销毁）。

第20条：促进返回值优化。

这段文字的核心思想是：**通过返回构造函数参数而不是局部对象，可以帮助编译器优化掉临时对象的开销**。

**2. 返回值优化（RVO）**

RVO 是一种编译器优化技术，允许编译器消除函数返回时创建的临时对象。具体来说，编译器可以直接在调用者的栈帧中构造返回值，从而避免额外的拷贝构造和析构。

**3. 命名返回值优化（NRVO）**

NRVO 是 RVO 的扩展，适用于函数返回命名局部对象的情况。NRVO 允许编译器将局部对象直接构造在返回值的位置，从而避免额外的拷贝构造和析构。

第21项：重载以避免隐式类型转换。

这个C++游戏是有规则的，其中之一就是每个重载操作符必须至少有一个用户定义类型的参数。

第22条：考虑使用op=而不是独立的op。

* 赋值版本的运算符（如 +=、-=）通常比独立版本的运算符（如 +、-）更高效，因为它们直接修改左操作数，而不需要创建临时对象。
* 独立版本的运算符会导致临时对象的构造、拷贝和析构，从而增加性能开销。
* 在实际编程中，优先使用赋值版本运算符，并通过它们来实现独立版本运算符，以提高代码的效率和可维护性。

项目23：考虑替代图书馆。

由于不同的库体现了关于效率、可扩展性、可移植性、类型安全和其他问题的不同设计决策，因此有时可以通过切换到设计者更重视性能考虑而不是其他因素的库来显著提高软件的效率。

项目24：理解虚函数、多重继承、虚基类和RTTI的开销。

虚拟表和虚拟表指针通常分别称为vtbls和vptrs

vtbl通常是指向函数的指针数组

**你必须为每个包含虚函数的类预留虚表空间**。

运行时类型识别（RTTI）的成本

然而，从效率的角度来看，你不太可能比编译器生成的实现更好地自己编写这些特性。