第 20 章 性能设计

到目前为止，关于软件设计的讨论都集中在复杂性上。目标是使软件尽可能简单易懂。但是，如果您需要让一个系统运行的更加高效，该怎么办？性能方面的考虑应如何影响设计过程？本章讨论如何在不牺牲简洁设计的情况下实现高性能。最重要的想法仍然是简单性：简单性不仅可以改善系统的设计，而且通常可以使系统更快。

20.1 如何考虑性能

要解决的第一个问题是："在正常的开发过程中，您应该在多大程度上担心性能？" 如果您尝试优化每条语句以获得最大速度，则它将减慢开发速度并产生很多不必要的复杂性。此外，许多"优化"实际上对性能没有帮助。另一方面，如果您完全忽略了性能问题，则很容易导致整个代码中出现大量低效的设计实现，结果系统很容易比所需的速度慢 5–10 倍。在这种"木已成舟"的情况下，再想回来改进性能也很难了，因为没有任何单一的改进会产生很大的影响。

最好的方法是介于这两种极端之间，您可以利用性能相关的基本知识来选择"自然高效"但又整洁和简单的设计方案。关键是要意识到哪些操作从根本上来说是性能开销大的。以下是一些今天仍然相对开销大的操作示例：

- 网络通信：即使在数据中心内，往返消息交换也可能要花费 10 到 50 微秒，相当于数以万计的指令的执行时间。而广域网的消息往返可能需要 10 到 100 毫秒。
- 辅助存储的 I/O：磁盘的 I/O 操作通常需要 5 到 10 毫秒，这是数百万条指令的执行时间。闪存存储需要 10 到 100 微秒。新出现的非易失性存储器的速度可能高达 1 微秒，但这仍然是大约 2000 条指令的执行时间。
- 动态内存分配（C 语言中的 `malloc`， C++ 或 Java 中的 `new`）通常涉及分配、释放和垃圾回收的大量开销。
- 缓存缺失：将数据从内存提取到处理器片上的高速缓存中需要数百条指令的执行时间；在许多程序中，整体性能受缓存缺失的影响程度与受计算开销的影响程度一样大。

了解哪些操作是性能开销大的最好方法是运行-微基准测试（单独衡量单个操作成本的小程序）。在 RAMCloud 项目中，我们创建了一个提供微基准测试框架的简单程序。创建该框架花了几天时间，但是该框架使在五到十分钟内添加新的微基准测试成为可能。这使我们积累了几十个微基准测试。我们既可以使用它们来了解 RAMCloud 中使用的现有库的性能，也可以衡量为 RAMCloud 编写的新类的性能。

一旦您对什么是性能开销大的和什么是性能开销小的有了大致的了解，就可以使用该信息尽可能地选择开销小的操作。在许多情况下，更高效的方法将与较慢的方法一样简单。例如，当需要存储使用键值查找的大量对象时，可以使用哈希表或有序映射（ordered map）。两者都通常在库包中提供，并且都简单易用。但是，哈希表可以轻松地快 5 到 10 倍。因此，除非需要映射（map）提供有序属性，否则您应使用哈希表。

作为另一个示例，请考虑使用诸如 C 或 C++ 之类的语言分配的结构数组。有两种方法可以执行此操作。一种方法是让数组存储指向结构的指针，在这种情况下，您必须首先为数组分配空间，然后为每个单独的结构分配空间。而直接将结构存储在数组中效率要高得多，因此您只需为所有内容分配一大块内存。

如果提高效率的唯一方法是增加复杂性，那么选择就更困难了。如果更高效的设计仅增加了少量复杂性，并且复杂性是隐藏的，也就是说它不影响任何接口，那么它可能是值得的（但要注意：复杂性是增量产生的）。如果更快的设计增加了很多实现复杂性，或者导致了更复杂的接口，那么最好还是从更简单的方法开始，并在性能开始成为问题时再进行优化。但是，如果您有明确的证据表明性能在特定情况下很重要，那么您不妨立即实现更高效的方法。

在 RAMCloud 项目中，我们的总体目标之一是为通过数据中心网络访问存储系统的客户端机器提供尽可能低的延迟。结果，我们决定使用特殊的硬件进行联网，从而使 RAMCloud 绕过内核并直接通过网络接口控制器发送和接收数据包。尽管增加了复杂性，但我们还是做出了这个决定，因为我们从先前的测量中知道，基于内核的网络太慢了，无法满足我们的需求。在 RAMCloud 系统的其余部分，我们能够进行简单设计。把这个大问题“正确解决”会让其他事情变得更加容易。

通常来说，简单的代码往往比复杂的代码运行得更快。如果您已经通过定义规避了特殊情况和异常情况，那么就不需要代码来检查这些情况，系统就会运行速度更快。深类比浅类更高效，因为它们为每个方法调用完成了更多工作。浅类会导致更多的层级交叉，并且每个层级交叉都会增加运行开销。

20.2 修改前（和修改后）的测量

但是假设您的系统仍然太慢，即使您已经按照上面描述的方式设计了它。根据您对什么导致了性能问题的直觉，很容易匆忙进行性能调整。不要这样做！程序员对性能的直觉是不可靠的。即使对于有经验的开发人员也是如此。如果您开始根据直觉进行修改，你会把时间浪费在实际上无法提高性能的事情上，并且在这个过程中可能会使系统变得更加复杂。

进行任何更改之前，请测量系统的现有行为。这有两个目的。首先，这些测量将找到性能调整能产生最大影响的地方。仅仅测量顶层的系统性能是不够的，这可能会告诉您系统速度太慢，但不会告诉您原因。您需要进行更深入的测量，以详细确定影响整体性能的因素。目标是找到系统中当前花费了大量时间的、少量非常具体的、以及您有改进想法的地方。测量的第二个目的是提供基线，以便您可以在进行更改后重新测量性能，以确保性能确实得到改善。如果更改并未在性能上产生可测量的差异，则将它们撤销（除非它们使系统更简单）。保留复杂性是没有意义的，除非它提供了显著的速度提升。

20.3 围绕关键路径进行设计

到这一步，假设您已经仔细分析了性能并确定了一段速度缓慢到足以影响整个系统的性能的代码。改善其性能的最佳方法是进行“根本性的”更改，例如引入缓存，或使用其他算法（例如，平衡树还是列表）。我们决定绕过内核进行 RAMCloud 中的网络通信的决定是一个根本性修正的示例。如果您找到了一个根本性的修正，则可以使用前面各章中讨论的设计技术来实现它。

不幸的是，有时会出现一些没有根本解决办法的情况。这就把我们带到本章的核心问题，即如何重新设计现有代码，使其运行更快。这应该是您不得已才采取的方法，并且不应该经常发生，但是在某些情况下它可能会带来很大的不同。关键思想是围绕关键路径设计代码。

首先，问您自己在通常情况下执行所需任务必须执行的最少代码量是多少。忽略任何现有的代码结构。想象一下您正在编写一个仅实现关键路径的新方法，这是在最常见的情况下必须执行的最少代码量。当前的代码可能充满特殊情况，但在此练习中，请忽略它们。当前的代码可能会在关键路径上涉及多个方法调用，想象一下您可以将所有相关代码放在一个方法中。当前代码可能还使用了各种变量和数据结构，请只考虑关键路径所需的数据，并假定一些最适合关键路径的数据结构。例如，将多个变量合并为一个值可能是有意义的。假设您可以完全重新设计系统，以最大程度地减少执行关键路径所必须包含的代码。我们把这段代码称为“理想的代码”。

理想的代码可能会与您现有的类结构冲突，并且可能不切实际，但它提供了一个很好的目标：这代表了可能是最简单和最快的代码。下一步是寻找一种新设计，使其尽可能接近理想状态，同时又要保持整洁的结构。您可以应用本书前面各章中的所有设计思想，但要保持（大部分）理想代码的完整性。您可能需要在理想代码上添加一些额外的代码，以便实现整洁的抽象。例如，如果代码涉及哈希表查找，引入一个额外的方法调用到一个通用的哈希表类是可以的。根据我的经验，几乎总是能找到一种简洁明了但是又非常接近理想状态的设计。

在此过程中发生的最重要的事情之一是从关键路径中移除特殊情况。当代码运行缓慢时，通常是因为它必须处理各种不同的情况，并且代码的结构也是为简化所有不同情况的处理而设计的。每个特殊情况都以额外的条件语句和/或方法调用的形式向关键路径添加了一些代码。每一个这种添加都会使代码变慢。重新设计性能时，请尝试减少必须检查的特殊情况的数量。理想情况下，开头应该有一个 `if` 语句，该语句可以通过一个测试检测所有特殊情况。在正常情况下，只需要进行这一项测试，之后就可以执行关键路径，而无需对于特殊情况进行其他测试。如果初始测试失败（这意味着发生了特殊情况），则代码可以分叉到关键路径之外的位置以进行处理。对于特殊情况来说，性能并不是那么重要，因此您可以将特殊情况下的代码设计得更简单而不用太追求性能。

20.4 示例：RAMCloud 缓冲区

让我们考虑一个例子，在这个例子中对 RAMCloud 存储系统的 `Buffer` 类进行了优化，在最常见的操作中实现了约两倍的性能提升。

RAMCloud 使用 `Buffer` 对象管理可变长度的内存数组，例如远程过程调用的请求和响应消息。`Buffer` 的设计旨在减少内存复制和动态存储分配的开销。`Buffer` 中看上去存储的是一个线性的字节数组，但为了提高效率，它允许底层存储将其划分为多个不连续的内存块，如图 20.1 所示。`Buffer` 是通过追加数据块创建的。每个块要么是外部的，要么是内部的。如果块是外部的，则其存储空间由调用方拥有，`Buffer` 中保存对此存储的引用。外部块通常用于大型块，以避免内存复制。如果块是内部的，则 `Buffer` 拥有该块的存储，调用者提供的数据将被复制到 `Buffer` 的内部存储中。

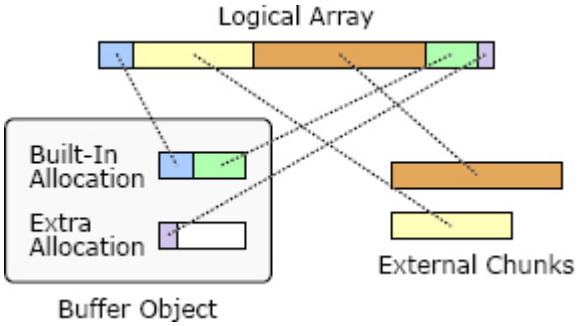

图 20.1：`Buffer` 对象使用内存块的集合来存储线性字节数组。内部块为 `Buffer` 拥有，并在 `Buffer` 销毁时释放；外部块不为 `Buffer` 所有。

每个 `Buffer` 内置一个小的分配器，这是一个可用于存储内部块的内存块。如果此空间已用完，则 `Buffer` 需要额外分配内存，这些分配的内存必须在 `Buffer` 销毁时进行释放。内部块对于小块来说是很方便的，因为其内存复制的成本可以忽略不计。图 20.1 显示了一个具有 5 个块的 `Buffer`：第一个块是内部的，接下来的两个块是外部的，最后两个块是内部的。

`Buffer` 类本身代表一个“根本性的修复”，因为它消除了开销大的内存拷贝，而如果没有它的话，就需要进行拷贝。例如，在 RAMCloud 存储系统中组装包含短标头和大的对象内容的响应消息时，RAMCloud 使用带有两个块的 `Buffer`。第一个块是包含标头的内部块；第二个块是一个外部块，它引用 RAMCloud 存储系统中的对象内容。这样就可以在不复制大对象的情况下将响应收集到 `Buffer` 中。

除了允许不连续块的基本设计外，在最初的实现中，我们并没有尝试优化 `Buffer` 类的代码。然而，随着时间的流逝，我们注意到 `Buffer` 越来越多地被使用。例如，在执行每个远程过程调用的期间，至少会创建四个 `Buffer` 对象。最终，我们发现优化 `Buffer` 的实现可能会对整体系统性能产生显著影响。我们决定看看是否可以提高 `Buffer` 类的性能。

`Buffer` 最常见的操作是使用内部块为少量新数据分配空间。例如，在为请求和响应消息创建标头时就会发生这种情况。我们决定使用将此操作作为优化的关键路径。在最简单的情况下，可以通过扩大 `Buffer` 中最后一个现有块来分配空间。但是，只有在最后一个现有块是内部块，并且在其分配中仍有足够的空间来容纳新数据时才有可能。理想的代码将执行一次检查，以确认简单方法是否可行，然后将调整现有块的大小。

图 20.2 展示了关键路径的原始代码，该代码以 `Buffer::alloc` 方法开头。在最快的使用场景下，`Buffer::alloc` 调用 `Buffer::allocateAppend`，后者再调用 `Buffer::Allocation::allocateAppend`。从性能的角度来看，此代码有两个问题。第一个问题是要单独检查多个特殊情况，并且有些还是重复的。首先，`Buffer::allocateAppend` 检查了 `Buffer` 当前是否有任何分配。然后代码检查了两次以查看当前分配是否有足够的空间容纳新数据：一次在 `Buffer::Allocation::allocateAppend` 中，一次在其返回值被 `Buffer::allocateAppend` 测试时。此外，该代码没有尝试直接扩展最后一个块，而是在不考虑最后一个块的情况下分配了新空间。然后，`Buffer::alloc` 检查该空间是否恰好与最后一块相邻，在这种情况下，它将新空间与现有块合并，这也导致了额外的检查。总体而言，该代码在关键路径上测试了 6 个不同的条件。

```
char* Buffer::alloc(int numBytes)
{
    char* data = allocateAppend(numBytes);
    Buffer::Chunk* lastChunk = this->chunksTail;
    if ((lastChunk != NULL && lastChunk->isInternal()) &&
            (data - lastChunk->length == lastChunk->data)) {
        // Fast path: grow the existing Chunk.
        lastChunk->length += numBytes;
        this->totalLength += numBytes;
    } else {
        // Creates a new Chunk out of the allocated data.
        append(data, numBytes);
    }
    return data;
}

// Allocates new space at the end of the Buffer; uses space at the end
// of the last current allocation, if possible; otherwise creates a
// new allocation. Returns a pointer to the new space.
char* Buffer::allocateAppend(int size) {
    void* data;
    if (this->allocations != NULL) {
        data = this->allocations->allocateAppend(size);
        if (data != NULL) {
            // Fast path
            return data;
        }
    }
    data = newAllocation(0, size)->allocateAppend(size);
    assert(data != NULL);
    return data;
}

// Tries to allocate space at the end of an existing allocation.
Returns
// a pointer to the new space, or NULL if not enough room.
char* Buffer::Allocation::allocateAppend(int size) {
    if ((this->chunkTop - this->appendTop) < size)
        return NULL;
    char *retVal = &data[this->appendTop];
    this->appendTop += size;
    return retVal;
}
```

图 20.2：使用内部块在 `Buffer` 的末尾分配新空间的原始代码。

原始代码的第二个问题是它的层级太多，而且都很浅。这既是性能问题，也是设计问题。除了对 `Buffer::alloc` 的原始调用之外，关键路径还进行了两个额外的方法调用。每个方法调用都需要额外的时间，其中一个调用的结果必须由其调用者检查，这导致了额外的需要考虑的特殊情况。第 7 章讨论了当您从一个层级转到另一个层级时，抽象通常应该如何变化，但是图 20.2 中的所有三个方法都具有相同的签名，它们提供了基本相同的抽象。这是一个危险信号。`Buffer::allocateAppend` 几乎是一个透传方法，它的唯一作用是在需要时创建新的分配。额外的层级使代码更慢，也更复杂。

为了解决这些问题，我们重构了 `Buffer` 类，使其设计围绕性能最关键的路径进行。我们不仅考虑了上面的分配代码，还考虑了其他几个常见的执行路径，例如检索当前存储在 Buffer 中的数据的总字节数。对于这些关键路径中的每一个，我们试图确定在通常情况下必须执行的最少代码量。然后，我们围绕这些关键路径设计了类的其余部分。我们还应用了本书中的设计原则来简化整个类。例如，我们消除了浅的层并创建了更深的内部抽象，还减少了需要检查的特殊情况数量。重构后的类比原始版本小 20%（1476 行代码，而原始版本为 1886 行）。

图 20.3 展示了在 `Buffer` 的内部块中分配空间的新关键路径。新代码不仅更快，而且更容易阅读，因为它避免了浅抽象。整个路径使用单个方法来处理，它使用单个测试来排除所有特殊情况。新代码引入了新的实例变量 `availableAppendBytes` 以简化关键路径，该变量跟踪缓冲区中最后一个块之后有多少空间直接可用。如果没有可用空间，或者 `Buffer` 中的最后一个块不是内部块，或者 `Buffer` 根本不包含任何块，则 `availableAppendBytes` 为零。只需要对 `availableAppendBytes` 进行测试，即可一次性检查三个不同的特殊情况。图 20.3 中展示的就是处理还有可用空间这种常见情况的最少量代码。

```
char* Buffer::alloc(int numBytes) {
    if (this->availableAppendBytes >= numBytes) {
        // There is extra space just after the current
        // last chunk, so we can allocate the new
        // region there.
        Buffer::Chunk* chunk = this->lastChunk;
        char* result = chunk->data + chunk->length;
        chunk->length += numBytes;
        this->availableAppendBytes -= numBytes;
        this->totalLength += numBytes;
        return result;
    }
    // We're going to have to create a new chunk.
}
```

图 20.3：用于在 `Buffer` 的内部块中分配新空间的新代码。

注意：可以通过在需要时重新计算各个块的总缓冲区长度来消除对 `totalLength` 的更新。但是，这种方法对于具有许多块的大型 `Buffer` 而言将是性能开销大的，并且获取 `Buffer` 的总长度是另一种常见的操作。因此，我们选择向 `alloc` 添加少量额外开销，以确保 `Buffer` 长度始终立即可用。

新代码的速度约为旧代码的两倍：使用内部存储将 1 字节字符串附加到 `Buffer` 的总时间从 8.8 纳秒降低到了 4.75 纳秒。许多其他 `Buffer` 操作也因为这次修改而加快了速度。例如，构建一个新的 `Buffer` 并在内部存储中追加一小块、然后再销毁 `Buffer` 的时间从 24 纳秒降到了 12 纳秒。

20.5 结论

本章总体上最重要的经验是，简洁的设计和高性能是可以兼容的。重写 `Buffer` 类可将其性能提高两倍，同时简化了其设计并将代码量减少了 20%。复杂的代码通常会很慢，因为它会执行无关或冗余的工作。另一方面，如果您编写整洁、简单的代码，则系统可能会足够快到您从一开始就不必担心性能。在少数需要优化性能的情况下，关键还是简化：找到对性能最重要的关键路径，并使它们尽可能简单。