1.2 为什么使用并发？

主要原因有两个：关注点分离(SOC)和性能。事实上，它们应该是使用并发的唯一原因；如果你观察得足够仔细，所有因素都可以归结到其中的一个原因(或者可能是两个都有。当然，除了像“就因为我愿意”这样的原因之外)。

1.2.1 为了分离关注点

编写软件时，分离关注点是个好主意；通过将相关的代码与无关的代码分离，可以使程序更容易理解和测试，从而减少出错的可能性。即使一些功能区域中的操作需要在同一时刻发生的情况下，依旧可以使用并发分离不同的功能区域；若不显式地使用并发，就得编写一个任务切换框架，或者在操作中主动地调用一段不相关的代码。

考虑一个有用户界面的处理密集型应用——DVD播放程序。这样的应用程序，应具备这两种功能：一，要从光盘中读出数据，对图像和声音进行解码，之后把解码出的信号输出至视频和音频硬件，从而实现DVD的无误播放；二，还需要接受来自用户的输入，当用户单击“暂停”、“返回菜单”或“退出”按键的时候执行对应的操作。当应用是单个线程时，应用需要在回放期间定期检查用户的输入，这就需要把“DVD播放”代码和“用户界面”代码放在一起，以便调用。如果使用多线程方式来分隔这些关注点，“用户界面”代码和“DVD播放”代码就不再需要放在一起：一个线程可以处理“用户界面”事件，另一个进行“DVD播放”。它们之间会有交互(用户点击“暂停”)，不过任务间需要人为的进行关联。

这会给响应性带来一些错觉，因为用户界面线程通常可以立即响应用户的请求，在当请求传达给忙碌线程，这时的相应可以是简单地显示代表忙碌的光标或“请等待”字样的消息。类似地，独立的线程通常用来执行那些必须在后台持续运行的任务，例如，桌面搜索程序中监视文件系统变化的任务。因为它们之间的交互清晰可辨，所以这种方式会使每个线程的逻辑变的更加简单。

在这种情况下，线程的数量不再依赖CPU中的可用内核的数量，因为对线程的划分是基于概念上的设计，而不是一种增加吞吐量的尝试。

1.2.2 为了性能

多处理器系统已经存在了几十年，但直到最近，它们也只在超级计算机、大型机和大型服务器系统中才能看到。然而，芯片制造商越来越倾向于多核芯片的设计，即在单个芯片上集成2、4、16或更多的处理器，从而获取更好的性能。因此，多核台式计算机、多核嵌入式设备，现在越来越普遍。它们计算能力的提高不是源自使单一任务运行的更快，而是并行运行多个任务。在过去，程序员曾坐看他们的程序随着处理器的更新换代而变得更快，无需他们这边做任何事。但是现在，就像Herb Sutter所说的，“没有免费的午餐了。”[1] *如果想要利用日益增长的计算能力，那就必须设计多任务并发式软件*。程序员必须留意这个，尤其是那些迄今都忽略并发的人们，现在很有必要将其加入工具箱中了。

两种方式利用并发提高性能：第一，将一个单个任务分成几部分，且各自并行运行，从而降低总运行时间。这就是任务并行（*task parallelism*）。虽然这听起来很直观，但它是一个相当复杂的过程，因为在各个部分之间可能存在着依赖。区别可能是在过程方面——一个线程执行算法的一部分，而另一个线程执行算法的另一个部分——或是在数据方面——每个线程在不同的数据部分上执行相同的操作（第二种方式）。后一种方法被称为数据并行（*data parallelism*）。

第一种并行方式影响的算法常被称为易并行(*embarrassingly parallel*)算法。尽管易并行算法的代码会让你感觉到头痛，但这对于你来说是一件好事：我曾遇到过自然并行(*naturally parallel*)和便利并发(*conveniently concurrent*)的算法。易并行算法具有良好的可扩展特性——当可用硬件线程的数量增加时，算法的并行性也会随之增加。这种算法能很好的体现*人多力量大*。如果算法中有不易并行的部分，你可以把算法划分成固定(不可扩展)数量的并行任务。第8章将会再来讨论，在线程之间划分任务的技巧。

第二种方法是使用可并行的方式，来解决更大的问题；与其同时处理一个文件，不如酌情处理2个、10个或20个。虽然，这是数据并行的一种应用(通过对多组数据同时执行相同的操作)，但着重点不同。处理一个数据块仍然需要同样的时间，但在相同的时间内处理了更多的数据。当然，这种方法也有限制，并非在所有情况下都是有益的。不过，这种方法所带来的吞吐量提升，可以让某些新功能成为可能，例如，可以并行处理图片的各部分，就能提高视频的分辨率。

1.2.3 什么时候不使用并发

知道何时**不使用**并发与知道何时**使用**它一样重要。基本上，不使用并发的唯一原因就是，收益比不上成本。使用并发的代码在很多情况下难以理解，因此编写和维护的多线程代码就会产生直接的脑力成本，同时额外的复杂性也可能引起更多的错误。除非潜在的性能增益足够大或关注点分离地足够清晰，能抵消所需的额外的开发时间以及与维护多线程代码相关的额外成本(代码正确的前提下)；否则，别用并发。

同样地，性能增益可能会小于预期；因为操作系统需要分配内核相关资源和堆栈空间，所以在启动线程时存在固有的开销，然后才能把新线程加入调度器中，所有这一切都需要时间。如果在线

程上的任务完成得很快，那么任务实际执行的时间要比启动线程的时间小很多，这就会导致应用程序的整体性能还不如直接使用“产生线程”的方式。

此外，线程是有限的资源。如果让太多的线程同时运行，则会消耗很多操作系统资源，从而使得操作系统整体上运行得更加缓慢。不仅如此，因为每个线程都需要一个独立的堆栈空间，所以运行太多的线程也会耗尽进程的可用内存或地址空间。对于一个可用地址空间为4GB(32bit)的平坦架构的进程来说，这的确是个问题：如果每个线程都有一个1MB的堆栈(很多系统都会这样分配)，那么4096个线程将会用尽所有地址空间，不会给代码、静态数据或者堆数据留有任何空间。即便64位(或者更大)的系统不存在这种直接的地址空间限制，但其他资源有限：如果你运行了太多的线程，最终也是出会问题的。尽管线程池(参见第9章)可以用来限制线程的数量，但这也并不是什么灵丹妙药，它也有自己的问题。

当客户端/服务器(C/S)应用在服务器端为每一个链接启动一个独立的线程，对于少量的链接是可以正常工作的，但当同样的技术用于需要处理大量链接的高需求服务器时，也会因为线程太多而耗尽系统资源。在这种场景下，使用线程池可以对性能产生优化（参见第9章）。

最后，运行越多的线程，操作系统就需要做越多的上下文切换，每一次切换都需要耗费本可以花在有价值工作上的时间。所以在某些时候，增加一个额外的线程实际上会降低，而非提高应用程序的整体性能。为此，如果你试图得到系统的最佳性能，可以考虑使用硬件并发(或不用)，并调整运行线程的数量。

为性能而使用并发就像所有其他优化策略一样:它拥有大幅度提高应用性能的潜力，但它也可能使代码复杂化，使其更难理解，并更容易出错。因此，只有应用中具有显著增益潜力的性能关键部分，才值得并发化。当然，如果性能收益的潜力仅次于设计清晰或关注点分离，可能也值得使用多线程设计。

假设你已经决定确实要在应用中使用并发，无论是为了性能、关注点分离，亦或是因为*多线程星期一*(multithreading Monday)(译者：可能是学习多线程的意思)。

问题又来了，对于C++程序员来说，多线程意味着什么？

[1] “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” Herb Sutter, Dr. Dobb’s Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.