现在考虑在没有多线程的情形下，如何编写Web服务器。一种可能的方式是，使其像一个线程一样运行。Web服务器的主循环获得请求，检查请求，并且在取下一个请求之前完成整个工作。在等待磁盘操作时，服务器就空转，并且不处理任何到来的其他请求。如果该Web服务器运行在惟一的机器上，通常情形都是这样，那么在等待磁盘操作时CPU只能空转。结果导致每秒钟只有很少的请求被处理。可见线程较好地改善了Web服务器的性能，而且每个线程是按通常方式顺序编程的。

到现在为止，我们有了两个可能的设计：多线程Web服务器和单线程Web服务器。假设没有多线程可用，而系统设计者又认为由于单线程所造成的性能降低是不能接受的，那么如果可以使用read系统调用的非阻塞版本，还存在第三种可能的设计。在请求到来时，这个惟一的线程对请求进行考察。如果该请求能够在高速缓存中得到满足，那么一切都好，如果不能，则启动一个非阻塞的磁盘操作。

服务器在表格中记录当前请求的状态，然后去处理下一个事件。下一个事件可能是一个新工作的请求，或是磁盘对先前操作的回答。如果是新工作的请求，就开始该工作。如果是磁盘的回答，就从表格中取出对应的信息，并处理该回答。对于非阻塞磁盘I/O而言，这种回答多数会以信号或中断的形式出现。

在这一设计中，前面两个例子中的“顺序进程”模型消失了。每次服务器从为某个请求工作的状态切换到另一个状态时，都必须显式地保存或重新装入相应的计算状态。事实上，我们以一种困难的方式模拟了线程及其堆栈。这里，每个计算都有一个被保存的状态，存在一个会发生且使得相关状态发生改变的事件集合，我们把这类设计称为有限状态机（finite-state machine）。有限状态机这一概念广泛地应用在计算机科学中。

现在很清楚多线程必须提供的是什么了。多线程使得顺序进程的思想得以保留下来；这种顺序进程阻塞了系统调用（如磁盘I/O），但是仍旧实现了并行性。对系统调用进行阻塞使程序设计变的较为简单，而且并行性改善了性能。单线程服务器虽然保留了阻塞系统调用的简易性，但是却放弃了性能。第三种处理方法运用了非阻塞调用和中断，通过并行性实现了高性能，但是给编程增加了困难。在图2-10中给出了上述模式的总结。

模型	特性
多线程	并行性、阻塞系统调用
单线程进程	无并行性、阻塞系统调用
有限状态机	并行性、非阻塞系统调用、中断

图2-10 构造服务器的三种方法

有关多线程作用的第三个例子是那些必须处理极大量数据的应用。通常的处理方式是，读进一块数据，对其处理，然后再写出数据。这里的问题是，如果只能使用阻塞系统调用，那么在数据进入和数据输出时，会阻塞进程。在有大量计算需要处理的时候，让CPU空转显然是浪费，应该尽可能避免。

多线程提供了一种解决方案，有关的进程可以用一个输入线程、一个处理线程和一个输出线程构造。输入线程把数据读入到输入缓冲区中；处理线程从输入缓冲区中取出数据，处理数据，并把结果放到输出缓冲区中；输出线程把这些结果写到磁盘上。按照这种工作方式，输入、处理和输出可以全部同时进行。当然，这种模型只有当系统调用只阻塞调用线程而不是阻塞整个进程时，才能正常工作。

2.2.2 经典的线程模型

既然我们已经明白为什么线程会有用以及如何使用它们，不如让我们用更近一步的眼光来审查一下上面的想法。进程模型基于两种独立的概念：资源分组处理与执行。有时，将这两种概念分开会更有益，这也引入了“线程”这一概念。我们将先来看经典的线程模型，之后我们会来研究“模糊进程与线程分界线”的Linux线程模型。

理解进程的一个角度是，用某种方法把相关的资源集中在一起。进程有存放程序正文和数据以及其他资源的地址空间。这些资源中包括打开的文件、子进程、即将发生的报警、信号处理程序、账号信息等。把它们都放到进程中可以更容易管理。

另一个概念是，进程拥有一个执行的线程，通常简写为线程（thread）。在线程中有一个程序计数器，用来记录接着要执行哪一条指令。线程拥有寄存器，用来保存线程当前的工作变量。线程还拥有一个堆