 先从著名的c10k问题谈起。有一个叫Dan Kegel的人在网上（[http://www.kegel.com/c10k.html](http://www.kegel.com/c10k.html" \t "http://blog.csdn.net/kobejayandy/article/details/_blank)）提出：现在的硬件应该能够让一台机器支持10000个并发的client。然后他讨论了用不同的方式实现大规模并发服务的技术，归纳起来就是两种方式：一个client一个thread，用blocking I/O；多个clients一个thread，用nonblocking I/O或者asynchronous I/O。目前asynchronous I/O的支持在**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "http://blog.csdn.net/kobejayandy/article/details/_blank)**上还不是很好，所以一般都是用nonblocking I/O。大多数的实现都是用epoll()的edge triggering（传统的select()有很大的性能问题）。这就引出了thread和event之争，因为前者就是完全用线程来处理并发，后者是用事件驱动来处理并发。当然实际的系统当中往往是混合系统：用事件驱动来处理网络时间，而用线程来处理事务。由于目前**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "http://blog.csdn.net/kobejayandy/article/details/_blank)**（尤其是Linux）和程序语言的限制（**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/kobejayandy/article/details/_blank)**/C/C++等），线程无法实现大规模的并发事务。一般的机器，要保证性能的话，线程数量基本要限制几百（Linux上的线程有个特点，就是达到一定数量以后，会导致系统性能指数下降，参看SEDA的论文）。所以现在很多高性能web server都是使用事件驱动机制，比如nginx，Tornado，**[Node.js](http://lib.csdn.net/base/nodejs" \o "Node.js知识库" \t "http://blog.csdn.net/kobejayandy/article/details/_blank)**等等。事件驱动几乎成了高并发的同义词，一时间红的不得了。

        其实线程和事件，或者说同步和异步之争早就在学术领域争了几十年了。1978年有人为了平息争论，写了论文证明了用线性的process（线程的模式）和消息传递（事件的模式）是等价的，而且如果实现合适，两者应该有同等性能。当然这是理论上的。针对事件驱动的流行，2003年加大伯克利发表了一篇论文叫“Why events are a bad idea (for high-concurrency servers)”，指出其实事件驱动并没有在功能上有比线程有什么优越之处，但事件驱动编程要麻烦很多，而且特别容易出错。线程的问题，无非是目前的实现的原因。一个是线程占的资源太大，一创建就分配几个MB的stack，一般的机器能支持的线程大受限制。针对这点，可以用自动扩展的stack，创建的先少分点，然后动态增加。第二个是线程的切换负担太大，Linux中实际上process和thread是一回事，区别就在于是否共享地址空间。解决这个问题的办法是用轻量级的线程实现，通过合作式的办法来实现共享系统的线程。这样一个是切换的花费很少，另外一个可以维护比较小的stack。他们用coroutine和nonblocking I/O（用的是poll()+thread pool）实现了一个原型系统，证明了性能并不比事件驱动差。

        那是不是说明线程只要实现的好就行了呢。也不完全对。2006年还是加大伯克利，发表了一篇论文叫“The problem with threads”。线程也不行。原因是这样的。目前的程序的模型基本上是基于顺序执行。顺序执行是确定性的，容易保证正确性。而人的思维方式也往往是单线程的。线程的模式是强行在单线程，顺序执行的基础上加入了并发和不确定性。这样程序的正确性就很难保证。线程之间的同步是通过共享内存来实现的，你很难来对并发线程和共享内存来建立数学模型，其中有很大的不确定性，而不确定性是编程的巨大敌人。作者以他们的一个项目中的经验来说明，保证多线程的程序的正确性，几乎是不可能的事情。首先，很多很简单的模式，在多线程的情况下，要保证正确性，需要注意很多非常微妙的细节，否则就会导致deadlock或者race condition。其次，由于人的思维的限制，即使你采取各种消除不确定的办法，比如monitor，transactional memory，还有promise/future，等等机制，还是很难保证面面俱到。以作者的项目为例，他们有计算机科学的专家，有最聪明的研究生，采用了整套软件工程的流程：design review, code review, regression tests, automated code coverage metrics，认为已经消除了大多数问题，不过还是在系统运行4年以后，出现了一个deadlock。作者说，很多多线程的程序实际上存在并发错误，只不过由于硬件的并行度不够，往往不显示出来。随着硬件的并行度越来越高，很多原来运行完好的程序，很可能会发生问题。我自己的体会也是，程序NPE，core dump都不怕，最怕的就是race condition和deadlock，因为这些都是不确定的(non-deterministic)，往往很难重现。

        那既然线程+共享内存不行，什么样的模型可以帮我们解决并发计算的问题呢。研究领域已经发展了一些模型，目前越来越多地开始被新的程序语言采用。最主要的一个就是Actor模型。它的主要思想就是用一些并发的实体，称为actor，他们之间的通过发送消息来同步。所谓“Don’t communicate by sharing memory, share memory by communicating”。Actor模型和线程的共享内存机制是等价的。实际上，Actor模型一般通过底层的thread/lock/buffer 等机制来实现，是高层的机制。Actor模型是数学上的模型，有理论的支持。另一个类似的数学模型是CSP（communicating sequential process）。早期的实现这些理论的语言最著名的就是erlang和occam。尤其是erlang，所谓的Ericsson Language，目的就是实现大规模的并发程序，用于电信系统。Erlang后来成为比较流行的语言。