并发系统可以采用多种并发编程模型来实现。并发模型指定了系统中的线程如何通过协作来完成分配给它们的作业。不同的并发模型采用不同的方式拆分作业，同时线程间的协作和交互方式也不相同。这篇并发模型教程将会较深入地介绍目前（2015年）比较流行的几种并发模型。

## 并发模型与分布式系统之间的相似性

本文所描述的并发模型类似于分布式系统中使用的很多体系结构。在并发系统中线程之间可以相互通信。在分布式系统中进程之间也可以相互通信（进程有可能在不同的机器中）。线程和进程之间具有很多相似的特性。这也就是为什么很多并发模型通常类似于各种分布式系统架构。

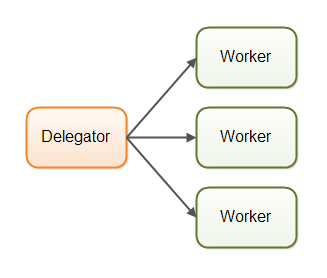
当然，分布式系统在处理网络失效、远程主机或进程宕掉等方面也面临着额外的挑战。但是运行在巨型服务器上的并发系统也可能遇到类似的问题，比如一块CPU失效、一块网卡失效或一个磁盘损坏等情况。虽然出现失效的概率可能很低，但是在理论上仍然有可能发生。

由于并发模型类似于分布式系统架构，因此它们通常可以互相借鉴思想。例如，为工作者们（线程）分配作业的模型一般与分布式系统中的负载均衡系统比较相似。同样，它们在日志记录、失效转移、幂等性等错误处理技术上也具有相似性。

【注：幂等性，一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同】

## 并行工作者

第一种并发模型就是我所说的并行工作者模型。



在并行工作者模型中，委派者（Delegator）将传入的作业分配给不同的工作者。每个工作者完成整个任务。工作者们并行运作在不同的线程上，甚至可能在不同的CPU上。

如果在某个汽车厂里实现了并行工作者模型，每台车都会由一个工人来生产。工人们将拿到汽车的生产规格，并且从头到尾负责所有工作。

在Java应用系统中，并行工作者模型是最常见的并发模型（即使正在转变）。java.util.concurrent包中的许多并发实用工具都是设计用于这个模型的。你也可以在Java企业级（J2EE）应用服务器的设计中看到这个模型的踪迹。

### 并行工作者模型的优点

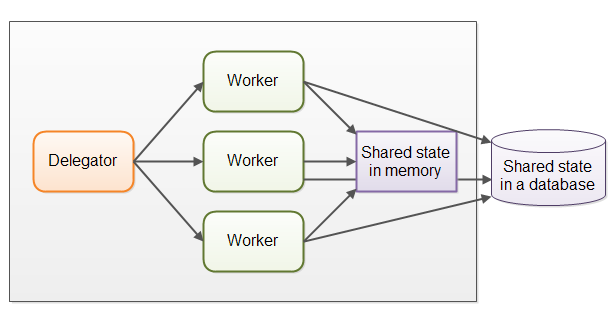
很容易理解，只需添加更多的工作者来提高系统的并行度。

例如，如果你正在做一个网络爬虫，可以试试使用不同数量的工作者抓取到一定数量的页面，然后看看多少数量的工作者消耗的时间最短（意味着性能最高）。由于网络爬虫是一个IO密集型工作，最终结果很有可能是你电脑中的每个CPU或核心分配了几个线程。每个CPU若只分配一个线程可能有点少，因为在等待数据下载的过程中CPU将会空闲大量时间。

### 并行工作者的缺点

* 共享状态可能会很复杂

共享的工作者经常需要访问一些共享数据，无论是内存中的或者共享的数据库中的。下图展示了并行工作者模型是如何变得复杂的：



有些共享状态是在像作业队列这样的通信机制下。但也有一些共享状态是业务数据，数据缓存，数据库连接池等。

一旦共享状态潜入到并行工作者模型中，将会使情况变得复杂起来。线程需要以某种方式存取共享数据，以确保某个线程的修改能够对其他线程可见（数据修改需要同步到主存中，不仅仅将数据保存在执行这个线程的CPU的缓存中）。线程需要避免竞态，死锁以及很多其他共享状态的并发性问题。

此外，在等待访问共享数据结构时，线程之间的相互等待将会丢失部分并行性。许多并发数据结构是阻塞的，意味着在任何一个时间只有一个或很少的线程能够访问。这样会导致在这些共享数据结构上出现竞争状态。在执行需要共享数据结构部分的代码时，高竞争基本上会导致执行时出现一定程度的串行化。

现在的非阻塞并发算法也许可以降低竞争并提升性能，但是非阻塞算法的实现比较困难。

可持久化的数据结构是另一种选择。在修改的时候，可持久化的数据结构总是保护它的前一个版本不受影响。因此，如果多个线程指向同一个可持久化的数据结构，并且其中一个线程进行了修改，进行修改的线程会获得一个指向新结构的引用。所有其他线程保持对旧结构的引用，旧结构没有被修改并且因此保证一致性。（注：这里的可持久化数据结构不是指持久化存储，而是一种数据结构，比如Java中的String类，以及CopyOnWriteArrayList类）

可持久化的数据结构的表现往往不尽人意。比如说，一个可持久化的链表需要在头部插入一个新的节点，并且返回指向这个新加入的节点的一个引用（这个节点指向了链表的剩余部分）。所有其他现场仍然保留了这个链表之前的第一个节点，对于这些线程来说链表仍然是会改变的。它们无法看到新加入的元素。这种可持久化的列表采用链表来实现。不幸的是链表在现代硬件上表现的不太好。链表中的每个元素都是一个独立的对象，这些对象可以遍布在整个计算机内存中。现代CPU能够更快的进行顺序访问，所以你可以在现代的硬件上用数组实现的列表，以获得更高的性能。数组可以顺序的保存数据。CPU缓存能够一次加载数组的一大块进行缓存，一旦加载完成CPU就可以直接访问缓存中的数据。这对于元素散落在RAM中的链表来说，不太可能做得到。

* 无状态工作者

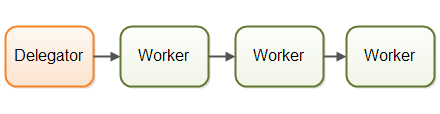
共享状态能够被系统中的其他线程修改。所以工作者在每次需要的时候必须重读状态，以确保每次都能访问到最新的副本，不管共享状态是保存在内存中还是在外部数据库中。工作者无法在内部保存这个状态（但是每次需要的时候可以重读）称为无状态的。

* 任务顺序是不确定的

作业执行顺序是不确定的。无法保证哪个作业最先或者最后被执行。并行工作者模式的这种非确定性的特性，使得很难在任何特定的时间点推断系统的状态。这也使得它也更难（如果不是不可能的话）保证一个作业在其他作业之前被执行。

## 流水线模式

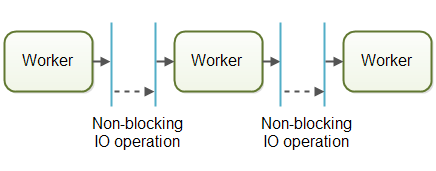
第二种并发模型称之为流水线并发模型。其他开发者可能会根据平台或社区选择其他称呼（反应器系统，或者事件驱动系统）



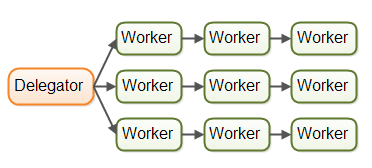
类似于工厂生产线上的工人们那样组织工作者。每个工作者只负责作业中的部分工作。当完成了自己的这部分工作时工作者会将作业转发给下一个工作者。每个工作者在自己的线程中运行，并且不会和其他工作者共享模型。有时也被称为无共享并行模式。

通常使用非阻塞的IO来设计使用流水线并发模型的系统。非阻塞IO意味着，一旦某个工作者开始一个IO操作的时候（比如读取文件或从网络连接中读取数据），这个工作者不会一直等待IO操作的结束。此时CPU可以做一些其他事情。当IO操作完成的时候，IO操作的结果（比如读出的数据或者数据写完的状态）被传递给下一个工作者。

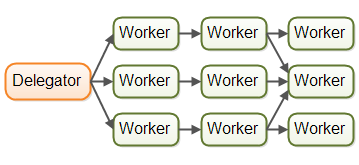
有了非阻塞IO，就可以使用IO操作确定工作者之间的边界。工作者会尽可能多运行直到遇到并启动一个IO操作。然后交出作业的控制权。当IO操作完成的时候，在流水线上的下一个工作者继续操作，直到它也遇到并启动一个IO操作。



在实际应用中，作业有可能不会沿着单一流水线进行。由于大多数系统可以执行多个作业，作业从一个工作者流向另一个工作者取决于作业需要做的工作。在实际中可能会有多个不同的虚拟流水线同时运行。这是现实当中作业在流水线系统中可能的移动情况：



作业甚至有可能被转发到超过一个工作者上并发处理。



### 反应器，事件驱动系统

采用流水线并发模型的系统有时候也称为反应器系统或事件驱动系统。系统内的工作者对系统内出现的事件做出反应，这件事也有可能来自于外部世界或者发自其它工作者。事件可以是传入的HTTP请求，也可以是某个文件成功加载到内存中等。比较流行的反应器/事件驱动平台似乎是这几个：

Vert.x

AKKa

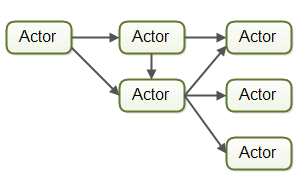
Node.JS(JavaScript)

Vert.x是相当有趣的（特别是对于我这样使用Java/JVM的人来说）

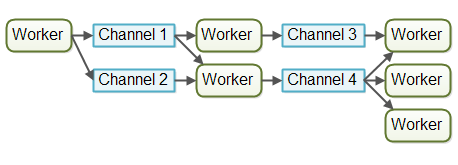
### Actors和Channels

Actors和channels是两种比较类似的流水线（或反应器/事件驱动）模型。

在Actor模型中每个工作者被称为actor。Actor之间可以直接异步地发送和处理消息。Actor可以被用来实现一个或多个像前文描述的那样的作业处理流水线。下图给出了Actor模型：



而在Channel模型中，工作者之间不直接进行通信。相反，它们在不同的通道中发布自己的消息（事件）。其他工作者们可以在这些通道上监听消息，发送者无需知道谁在监听。下图给出了Channel模型：



channel模型似乎更加灵活。一个工作者无需知道谁在后面的流水线上处理作业。只需知道作业（或消息等）需要转发给哪个通道。通道上的监听者可以随意订阅或者取消订阅，并不会影响向这个通道发送消息的工作者。这使得工作者之间具有松散的耦合。

### 流水线模型的优点

* 无需共享的状态
* 有状态的工作者
* 较好的硬件整合（Hardware Conformity）
* 合理的作作业顺序

### 流水线模型的缺点

作业执行往往分配到多个工作者上，并因此分布到项目中的多个类上。这样导致在追踪某个作业到底被什么代码执行时变得困难。

同样，这也加大了代码编写的难度。有时会将工作者的代码写成回调处理的形式。若在代码中嵌入过多的回调处理，往往会出现所谓的回调地狱（callback hell）现象。

## 函数式并行（Functional Parallelism）

第三种并发模型是函数式并行模型，这是也2015讨论的比较多的一种模型。函数式并行的基本思想是采用函数调用实现程序。函数可以看作是”代理人（agents）“或者”actor“，函数之间可以像流水线模型（AKA反应器或者事件驱动系统）那样互相发送消息。某个函数调用另一个函数，这个过程类似于消息发送。

函数都是通过拷贝来传递参数的，所以除了接收函数外没有实体可以操作数据。这对于避免共享数据的竞态来说是很有必要的。同样也使得函数的执行类似于原子操作。每个函数调用的执行独立于任何其他函数的调用。

一旦每个函数调用都可以独立的执行，它们就可以分散在不同的CPU上执行了。这也就意味着能够在多处理器上并行的执行使用函数式实现的算法。

Java7中的java.util.concurrent包里包含的ForkAndJoinPool能够帮助我们实现类似于函数式并行的一些东西。而Java8中并行streams能够用来帮助我们并行的迭代大型集合。记住有些开发者对ForkAndJoinPool进行了批判（你可以在我的ForkAndJoinPool教程里面看到批评的链接）。

函数式并行里面最难的是确定需要并行的那个函数调用。跨CPU协调函数调用需要一定的开销。某个函数完成的工作单元需要达到某个大小以弥补这个开销。如果函数调用作用非常小，将它并行化可能比单线程、单CPU执行还慢。

我个人认为（可能不太正确），你可以使用反应器或者事件驱动模型实现一个算法，像函数式并行那样的方法实现工作的分解。使用事件驱动模型可以更精确的控制如何实现并行化（我的观点）。

此外，将任务拆分给多个CPU时协调造成的开销，仅仅在该任务是程序当前执行的唯一任务时才有意义。但是，如果当前系统正在执行多个其他的任务时（比如web服务器，数据库服务器或者很多其他类似的系统），将单个任务进行并行化是没有意义的。不管怎样计算机中的其他CPU们都在忙于处理其他任务，没有理由用一个慢的、函数式并行的任务去扰乱它们。使用流水线（反应器）并发模型可能会更好一点，因为它开销更小（在单线程模式下顺序执行）同时能更好的与底层硬件整合。