并发编程含义比较广泛，包含多线程编程、多进程编程及分布式程序等

目录

[1. “共享内存系统”，消息传递系统”。 1](#_Toc29285)

[1.1. 共享模式 多进程 多线程 1](#_Toc14793)

[1.2. Actor消息模式 事件驱动 2](#_Toc18483)

[1.3. csp消息模式 事件驱动 2](#_Toc29602)

[2. 优缺点比较 3](#_Toc8191)

[2.1. 线程模式简单 3](#_Toc14044)

[2.2. 消息模式编程复杂 3](#_Toc26263)

[2.3. 消息驱动稳定性较高 3](#_Toc25053)

[3. 用线性的process（线程的模式）和消息传递（事件的模式）是等价的 3](#_Toc23128)

[4. 并发编程含义比较广泛，包含多线程编程、多进程编程及分布式程序等。本章讲解的并发含义属于多线程编程 4](#_Toc21748)

[5. 称为 goroutine 5](#_Toc4139)

[6. CSP和Actor的区别 6](#_Toc6098)

[7. goroutine和coroutine的区别抢占式vs 协作式 7](#_Toc26024)

[8. rf 8](#_Toc23477)

# “共享内存系统”，消息传递系统”。

## 共享模式 多进程 多线程

程之间通信只能采用共享内存的方式。为了保证共享内存的有效性，我们采取了很多措施，比如加锁等，来避免死锁或资源竞争。实践证明，我们很难面面俱到，往往会在工程中遇到各种奇怪的故障和问题。  
  
我们可以将之前的线程加共享内存的方式归纳为“共享内存系统”，虽然共享内存系统是一种有效的并发模式，但它也暴露了众多使用上的问题。计算机科学家们在近 40 年的研究中又产生了一种新的系统模型，称为“消息传递系统”。

线程间共享状态的各种操作都被封装在线程之间传递的消息中，这通常要求：发送消息时对状态进行复制，并且在消息传递的边界上交出这个状态的所有权。从逻辑上来看，这个操作与共享内存系统中执行的原子更新操作相同，但从物理上来看则非常不同。由于需要执行复制操作，所以大多数消息传递的实现在性能上并不优越，但线程中的状态管理工作通常会变得更为简单。

## Actor消息模式 事件驱动

## csp消息模式 事件驱动

最早被广泛应用的消息传递系统是由 C. A. R. Hoare 在他的 Communicating Sequential Processes 中提出的。在 CSP 系统中，所有的并发操作都是通过独立线程以异步运行的方式来实现的。这些线程必须通过在彼此之间发送消息，从而向另一个线程请求信息或者将信息提供给另一个线程。使用类似 CSP 的系统将提高编程的抽象级别。

并发指在同一时间内可以执行多个任务。

消息机制认为每个并发单元是自包含的、独立的个体，并且都有自己的变量，但在不同并发单元间这些变量不共享。每个并发单元的输入和输出只有一种，那就是消息。这有点类似于进程的概念，每个进程不会被其他进程打扰，它只做好自己的工作就可以了。不同进程间靠消息来通信，它们不会共享内存。  
  
Go语言提供的消息通信机制被称为 channel，关于 channel 的介绍将在后续的学习中为大家讲解。现在，让我们用 Go语言社区的那句著名的口号来结束这一小节：“不要通过共享内存来通信，而应该通过通信来共享内存。”

# 优缺点比较

## 线程模式简单

## 消息模式编程复杂

## 消息驱动稳定性较高

# 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。

# 并发编程含义比较广泛，包含多线程编程、多进程编程及分布式程序等。本章讲解的并发含义属于多线程编程

从以上几个例子可以看到，串行程序在很多场景下无法满足我们的要求。下面我们归纳了并发程序的几条优点，让大家认识到并发势在必行：

* 并发能更客观地表现问题模型；
* 并发可以充分利用 CPU 核心的优势，提高程序的执行效率；
* 并发能充分利用 CPU 与其他硬件设备固有的异步性。

现在我们已经意识到并发的好处了，那么到底有哪些方式可以实现并发执行呢？就目前而言，并发包含以下几种主流的实现模型。

#### 1) 多进程

多进程是在操作系统层面进行并发的基本模式。同时也是开销最大的模式。在 Linux 平台上，很多工具链正是采用这种模式在工作。比如某个 Web 服务器，它会有专门的进程负责网络端口的监听和链接管理，还会有专门的进程负责事务和运算。这种方法的好处在于简单、进程间互不影响，坏处在于系统开销大，因为所有的进程都是由内核管理的。

#### 2) 多线程

多线程在大部分操作系统上都属于系统层面的并发模式，也是我们使用最多的最有效的一种模式。目前，我们所见的几乎所有工具链都会使用这种模式。它比多进程的开销小很多，但是其开销依旧比较大，且在高并发模式下，效率会有影响。

#### 3) 基于回调的非阻塞/异步 IO

这种架构的诞生实际上来源于多线程模式的危机。在很多高并发服务器开发实践中，使用多线程模式会很快耗尽服务器的内存和 CPU 资源。

而这种模式通过事件驱动的方式使用异步 IO，使服务器持续运转，且尽可能地少用线程，降低开销，它目前在 Node.js 中得到了很好的实践。但是使用这种模式，编程比多线程要复杂，因为它把流程做了分割，对于问题本身的反应不够自然。

#### 4) 协程

协程（Coroutine）本质上是一种用户态线程，不需要操作系统来进行抢占式调度，且在真正的实现中寄存于线程中，因此，系统开销极小，可以有效提高线程的任务并发性，而避免多线程的缺点。使用协程的优点是编程简单，结构清晰；缺点是需要语言的支持，如果不支持，则需要用户在程序中自行实现调度器。目前，原生支持协程的语言还很少。

# 称为 goroutine

虽然，线程池为逻辑编写者提供了线程分配的抽象机制。但是，如果面对随时随地可能发生的并发和线程处理需求，线程池就不是非常直观和方便了。能否有一种机制：使用者分配足够多的任务，系统能自动帮助使用者把任务分配到 CPU 上，让这些任务尽量并发运作。这种机制在 [Go语言](http://c.biancheng.net/golang/" \t "http://c.biancheng.net/view/_blank)中被称为 **goroutine**。  
  
goroutine 是 Go语言中的轻量级线程实现，由 Go 运行时（runtime）管理。Go 程序会智能地将 goroutine 中的任务合理地分配给每个 CPU。  
  
Go 程序从 main 包的 main() 函数开始，在程序启动时，Go 程序就会为 main() 函数创建一个默认的 goroutine

# CSP和Actor的区别

* CSP进程通常是同步的(即任务被推送进Channel就立即执行，如果任务执行的线程正忙，则发送者就暂时无法推送新任务)，Actor进程通常是异步的(消息传递给Actor后并不一定马上执行).
* CSP中的Channel通常是匿名的, 即任务放进Channel之后你并不需要知道是哪个Channel在执行任务，而Actor是有“身份”的，你可以明确的知道哪个Actor在执行任务.
* 在CSP中，我们只能通过Channel在任务间传递消息, 在Actor中我们可以直接从一个Actor往另一个Actor传输数据.
* CSP中消息的交互是同步的，Actor中支持异步的消息交互

# goroutine和coroutine的区别抢占式vs 协作式

< 并发和并行的区别Go语言通道（chan） >

C#、Lua、Python 语言都支持 coroutine 特性。coroutine 与 goroutine 在名字上类似，都可以将函数或者语句在独立的环境中运行，但是它们之间有两点不同：

goroutine 可能发生并行执行；

但 coroutine 始终顺序执行

狭义地说，goroutine 可能发生在多线程环境下，goroutine 无法控制自己获取高优先度支持；coroutine 始终发生在单线程，coroutine 程序需要主动交出控制权，宿主才能获得控制权并将控制权交给其他 coroutine。  
  
goroutine 间使用 channel 通信，coroutine 使用 yield 和 resume 操作。  
  
goroutine 和 coroutine 的概念和运行机制都是脱胎于早期的操作系统。  
  
coroutine 的运行机制属于协作式任务处理，早期的操作系统要求每一个应用必须遵守操作系统的任务处理规则，应用程序在不需要使用 CPU 时，会主动交出 CPU 使用权。如果开发者无意间或者故意让应用程序长时间占用 CPU，操作系统也无能为力，表现出来的效果就是计算机很容易失去响应或者死机。  
  
goroutine 属于抢占式任务处理，已经和现有的多线程和多进程任务处理非常类似。应用程序对 CPU 的控制最终还需要由操作系统来管理，操作系统如果发现一个应用程序长时间大量地占用 CPU，那么用户有权终止这个任务。

# rf

(9+条消息)Actor并发模型&基于共享内存线程模型 - kobejayandy的专栏 - CSDN博客.mhtml