Atitit 并发处理 go与 java对比较

目录

[1. 并发编程的七个模型 线程，fp，actor，csp 数据级并行 Lambda 1](#_Toc16012)

[2. 四种并发编程模型简介 多线程，callback Actor csp 2](#_Toc7054)

[3.](#_Toc30454) *[JVM上的Thread, Thread Pool, Future, Rx, async-await, Fiber, Actor等并发编程模型](#_Toc30454)* [5](#_Toc30454)

[3.1.](#_Toc30442) *[Actor与分布式架构](#_Toc30442)* [5](#_Toc30442)

*[4.](#_Toc17463)* [5种常见的并发模型](#_Toc17463) *[Future模型 Fork/Join模型 3、Actor模型 生产者消费者模型 Master-Worker模型](#_Toc17463)* [6](#_Toc17463)

[5. Go 并发 7](#_Toc10694)

[6. 线程间通信 channel模式 8](#_Toc12772)

[6.1. 通道（channel） 8](#_Toc19351)

[6.2. 二. wait/notify 机制过时 8](#_Toc4877)

[6.3. 三. Condition 8](#_Toc16109)

[6.4. 五.线程间的通信：管道 8](#_Toc13495)

[6.5. Mq模式 消费者模式 9](#_Toc9869)

[6.6. 六. 方法 join() 的使用 9](#_Toc22812)

[7. Node.js多进程模型 9](#_Toc25642)

# **[并发编程的七个模型](https://www.cnblogs.com/barrywxx/p/10406978.html) 线程，fp，actor，csp **数据级并行 Lambda****

**线程与锁**：线程与锁模型有很多众所周知的不足，但仍是其他模型的技术基础，也是很多并发软件开发的首选。

**函数式编程**：函数式编程日渐重要的原因之一，是其对并发编程和并行编程提供了良好的支持。函数式编程消除了可变状态，所以从根本上是线程安全的，而且易于并行执行。

**Clojure之道——分离标识与状态**：编程语言Clojure是一种指令式编程和函数式编程的混搭方案，在两种编程方式上取得了微妙的平衡来发挥两者的优势。

**actor**：actor模型是一种适用性很广的并发编程模型，适用于共享内存模型和分布式内存模型，也适合解决地理分布型问题，能提供强大的容错性。

**通信顺序进程（Communicating Sequential Processes，CSP）**：表面上看，CSP模型与actor模型很相似，两者都基于消息传递。不过CSP模型侧重于传递信息的通道，而actor模型侧重于通道两端的实体，使用CSP模型的代码会带有明显不同的风格。

**数据级并行**：每个笔记本电脑里都藏着一台超级计算机——GPU。GPU利用了数据级并行，不仅可以快速进行图像处理，也可以用于更广阔的领域。如果要进行有限元分析、流体力学计算或其他的大量数字计算，GPU的性能将是不二选择。

**Lambda架构**：大数据时代的到来离不开并行——现在我们只需要增加计算资源，就能具有处理TB级数据的能力。Lambda架构综合了MapReduce和流式处理的特点，是一种可以处理多种大数据问题的架构。

# 四种并发编程模型简介 多线程，callback Actor csp

多线程编程模型

多线程模型是用于处理并发的最通用手段，在 C/C++/JAVA 等语言中广泛存在。主要特性有：

l 多个相互独立的执行流.

Callback编程模型

但是滥用回调嵌套，就会导致著名的”callback hell”问题，代码难以阅读和维护。例如下面的片段:

为了避免此类大坑，我们可以参考以下几类解决方案:

l Promises/A+规范： 它是一种用于管理异步回调的代码结构和流程，一种回调的语法糖。可以把原本嵌套的回调函数展平，使得代码逻辑更清楚。例如片段:

l Generator: 生成器/半协程方式: 可以将一个函数执行暂停，并保存上下文, 将控制权交还给调用者；当再次被调用时，能够恢复当时的暂停状态继续执行。所以generator函数的行为表现和迭代器很类似，每次触发它的时候可以获取到新的结果，而不是像传统函数全部执行结束后一口气返回一系列值。 代码片段:

l Async/Await: 可以视为Generator方式的语法糖，能够更好地展示异步调用的语义: async关键字用于表示该函数中有异步操作；await关键字表示需要等待(异步方式)后继表达式的结果。

Actor编程模型

Actor模型首先是由Carl Hewitt在1973年提出定义， 随后由Erlang OTP (Open Telecom Platform) 推广开来。Actor属于并发组件模型， 通过组件方式定义并发编程范式的高级阶段，避免使用者直接接触多线程并发或线程池等基础概念，其消息传递更加符合面向对象的原始意图。

传统多数流行的语言并发是基于多线程之间的共享内存，使用同步机制来防止写争夺。而Actors使用消息模型，每个Actors在同一时间处理最多一个消息，可以发送消息给其他Actors，保证了单独写原则，从而巧妙避免了多线程的写争夺。

CSP编程模型

CSP（Communicating Sequential Processes）是由Tony Hoare在1978的论文上首次提出的。 它是处理并发编程的一种设计模式或者模型，指导并发程序的设计，提供了一种并发程序可实践的组织方法或者设计范式。通过此方法，可以减少并发程序引入的其它缺点，减少和规避并发程序的常见缺点和bug，并且可以被数学理论所论证。

# *JVM上的Thread, Thread Pool, Future, Rx, async-await, Fiber, Actor等并发编程模型*

## ***Actor与分布式架构***

可以看出，相比于async-await或Fiber，actor就是一种状态机，是较为底层、不易用的编程模型。但是actor附带了成熟的分布式能力。

我感觉actor很像异步版的EJB。EJB中有stateless session bean和stateful session bean，actor也可按stateless和stateful来分类。

# 5种常见的并发模型 *Future模型 Fork/Join模型 3、Actor模型 生产者消费者模型 Master-Worker模型*

*前言*

*并发在现在已经是十分常见的问题了，由于人类信息量的增加，很多信息都需要并发处理，原有的串行处理已经很难满足现实的需求。*

*今天我们来讲一讲5种常见的并发模型*

*1、Future模型 Fork/Join模型 3、Actor模型 生产者消费者模型 Master-Worker模型*

*2、Fork/Join模型*

*将任务分割成足够小的小任务，然后让不同的线程来做这些分割出来的小事情，然后完成之后再进行join，将小任务的结果组装成大任务的结果*

*3、Actor模型*

*4、生产者消费者模型*

*核心是使用一个缓存来保存任务。开启一个/多个线程来生产任务，然后再开启一个/多个来从缓存中取出任务进行处理。*

*这样的好处是任务的生成和处理分隔开，生产者不需要处理任务，只负责向生成任务然后保存到缓存。而消费者只需要从缓存中取出任务进行处理。使用的时候可以根据任务的生成情况和处理情况开启不同的线程来处理。*

*5、Master-Worker模型*

*核心思想:系统有两个进程协作工作*

*Master进程，负责接收和分配任务；*

*Worker进程，负责处理子任务。*

*当Worker进程将子任务处理完成后，结果返回给Master进程，由Master进程做归纳汇总，最后得到最终的结果*

# Go 并发

Go 语言支持并发，我们只需要通过 go 关键字来开启 goroutine 即可。

goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。

goroutine 语法格式：

func say(s string) {  
        **for** i := 0; i < 5; i++ {  
                time.Sleep(100 \* time.Millisecond)  
                fmt.Println(s)  
        }  
}  
  
func main() {  
        **go** say("world")  
        say("hello")  
}

Java需要使用thread类对方法来执行某个函数

# 线程间通信 channel模式

## 通道（channel）

通道（channel）是用来传递数据的一个数据结构。

通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。

## **二. wait/notify 机制过时**

## **三. Condition**

Condition是在java 1.5中出现的，它用来替代传统的Object的wait()/notify()实现线程间的协作，它的使用依赖于 Lock，Condition、Lock 和 Thread 三者之间的关系如下图所示。**相比使用Object的wait()/notify()，使用Condition的await()/signal()这种方式能够更加安全和高效地实现线程间协作。Condition是个接口，基本的方法就是await()和signal()方法。Condition依赖于Lock接口，生成一个Condition的基本代码是lock.newCondition() 。 必须要注意的是，Condition 的 await()/signal() 使用都必须在lock保护之内，也就是说，必须在lock.lock()和lock.unlock之间才可以使用。**事实上，Conditon的await()/signal() 与 Object的wait()/notify() 有着天然的对应关系：

* Conditon中的await()对应Object的wait()；

## **五.线程间的通信：管道**

PipedInputStream类 与 PipedOutputStream类 用于在应用程序中创建管道通信。一个PipedInputStream实例对象必须和一个PipedOutputStream实例对象进行连接而产生一个通信管道。PipedOutputStream可以向管道中写入数据，PipedIntputStream可以读取PipedOutputStream向管道中写入的数据，这两个类主要用来完成线程之间的通信。一个线程的PipedInputStream对象能够从另外一个线程的PipedOutputStream对象中读取数据，如下图所示：

## Mq模式 消费者模式

## **六. 方法 join() 的使用**

1). join() 的定义

**假如在main线程中调用thread.join方法，则main线程会等待thread线程执行完毕或者等待一定的时间。**详细地，如果调用的是无参join方法，则等待thread执行完毕；如果调用的是指定了时间参数的join方法，则等待一定的时间。join()方法有三个重载版本：

public final synchronized void join(long milli

# Node.js多进程模型

并发模型比较 - 云+社区 - 腾讯云