# 2018年11月21日09:21:45>五类安全三种方案

第一：五类安全问题

1. 不可变
2. 绝对的线程安全
3. 相对的线程安全
4. 线程兼容
5. 线程对立

解决线程的方案

1可重入代码

2本地线程存储

3 互斥

第二：到底是为了并发还是并行

1. 线程与锁

主要使用互斥

1. 函数式编程

可重入代码，只要输入相同的值，出来的值也一定是相同的

1. 通信顺序进程
2. 数据集并行（并行，利用GPU）
3. Lambda架构
4. Cloujure之道
5. Actor（这个可能要单独了解，书中介绍了一种我不知道的语言，只能体会思维）

给我的感觉就是，除了线程和锁，函数式编程似乎可以原生的支持并发与并行，其本身同时具有的无状态，不变性，可重入性

不管是actor还是channel都是基于事件和消息，而且函数编程似乎确实是高并发中的主流

# 2018年11月22日09:21:15>并发问题可以总结一下为这样

1.共享资源的竞争态

2.资源的利用

3.到底需要多少线程

针对一下问题：

线程之间共享资源

线程之间互相依赖

线程间相互独立

有如下解决方式：

* **Mutex**（Lock）（Go 里的 sync 包, Java 的 concurrent 包）通过互斥量来保护数据，但有了锁，明显就降低了并发度。
* **Semaphore** 通过信号量来控制并发度或者作为线程间信号（signal）通知。
* **Volatile** Java 专门引入了 volatile 关键词来，来降低只读情况下的锁的使用。
* **Compare-and-swap** 通过硬件提供的 CAS 机制保证原子性（atomic），也是降低锁的成本的机制。

对于资源的利用上，提出线程池方案，异步回调，**GreenThread/Coroutine/Fiber（总结起来代码片段可中断，整体保证顺序执行）**

**Actor模型和CSP模型**

**Actor模型我是这样理解的：**

**你对美女打招呼，但是美女是否给予回应你是做不了主的，也就是说不会占用你的资源，会占用美女的资源。相当于每个actor是个独立的个体。actor关注消息是否被恢复或者期望相应的actor进行回复**

**CSP模型很像常见的消息队列，你放到channel（actor里面是mailbox，但是透明的，actor并不知道）里面，不知道也不关注谁会回复，只要回复就好，对消息负责，不对收发消息的负责**

**最后**

**最后**

**最后**

**你会发现，有两个问题依然残留**

**对于资源的共享而产生的静态除了互斥加锁原子操作似乎没有更好的方式，而这些操作都会降低并发度，当然也可以选择使用不便的数据，这样确实就可以保证**

**同样到底需要多少线程的问题依然没有得到解决**