首先要保证并发的正确性， 然后在此基础上实现高效。

13.1线程安全

定义： 当有多个线程同时访问一个对象时， 如果不需要额外的同步机制， 不管他们怎么交替执行， 都用的这个对象都能获得正确的结果， 那么这个对象就是线程安全的 。 （要求太严格了）

分类：

1. 不可变对象 。 一般用final修饰， 不可变对象一定是线程安全的 。 如果共享数据是一个基本类型， 就用一个final修饰； 如果是一个对象，那就需要保证对象的任何行为都不会对其状态产生影响。 => String , Long , Double , BigInteger

2). 绝对线程安全。 符合定义的， 太严格了

3）. 相对线程安全 。 我们统称的安全线程， 只需保证 对这个对象的单独的操作是正确的； 但是往往一系列的操作是需要额外的同步措施的 。

4）线程兼容 。 对象本身不安全， 但是调用者可以使用一定的措施 使之安全运行.

5）线程对立。 不管如何都是无法成为安全的对象 。

线程安全的实现方法：

1） 互斥（阻塞）同步 。 临界区 ， 互斥量， 信号量 。 synchronize 使用的是管程，用核心态的完成线程的挂起， 阻塞 。 区别于重入锁 ReentrantLock锁：

a. 等待可中断 。如果一个等待锁的时间太长， 那么可以放弃等待， 改为处理其他事情

b.公平锁。 按照线程申请锁的时间作为优先级来获得锁

c.锁绑定多个条件 。 不用使用多个synchronize了， 可以一次使用多个条件作为锁。

但是JVM更加偏向于原生的synchronize，所以在能用它实现业务的情况就是用它吧。

2）乐观锁 -- 非阻塞同步 。 先进行操作， 如果没有线程争取共享数据， 那么操作就成功了； 如果共享数据产生了冲突， 就采用其他的补偿措施（不断地重试， 直到成功）。

3）无同步方案 。 如果一个方法不涉及共享数据， 自然不用去同步了 。

=> 可重入代码， 线程本地储存（Web交互模型）。

13.2 锁优化

1）自旋锁 与 自适应自旋。 如果一个线程不会持有一个锁很久， 那么等待这个锁的线程， 就可以做几次空操作（默认10次）。

2） 锁消除。 对一些代码要求同步， 但是被检测到不存在共享数据竞争， 那么可以进行消除。

3）锁粗化。 如果一系列的连续操作， 都对同一个对象加锁 ， 或者锁在循环中出现， 可以适当粗化锁，减少消耗。

4）轻量级锁。 减少重量级锁。

5）偏向锁。 偏向第一个获得它的线程。