### 可重入锁和不可重入锁

可重入锁概念记录：可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。

<https://www.cnblogs.com/dj3839/p/6580765.html>

可重入锁：synchronized 和   ReentrantLock 都是可重入锁。

<https://www.cnblogs.com/theRhyme/p/9133804.html>

待深入探索点：java 自旋锁

# **[Java并发问题--乐观锁与悲观锁](https://www.cnblogs.com/qjjazry/p/6581568.html)以及Java并发问题**

详解：<https://www.cnblogs.com/qjjazry/p/6581568.html>

乐观锁：[顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制；乐观锁的一种实现方式-CAS](https://www.cnblogs.com/qjjazry/p/6581568.html)

悲观锁:总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。Synchronized

CAS与Synchronized的使用情景：

　　　　1、对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

　　　 2、对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

### java 中的锁 -- 偏向锁、轻量级锁、自旋锁、重量级锁

<https://blog.csdn.net/zqz_zqz/article/details/70233767>

### ReenTrantLock可重入锁（和synchronized的区别）总结

**可重入性：**

从名字上理解，ReenTrantLock的字面意思就是再进入的锁，其实synchronized关键字所使用的锁也是可重入的，两者关于这个的区别不大。两者都是同一个线程没进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

**锁的实现：**

Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，有什么区别，说白了就类似于操作系统来控制实现和用户自己敲代码实现的区别。前者的实现是比较难见到的，后者有直接的源码可供阅读。

**性能的区别：**

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

Tyfun-大数据量并发情况下，还是优先考虑使用ReenTrantLock，也具有灵活性

**功能区别：**

便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

**ReenTrantLock独有的能力：**

1.      ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

2.      ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

3.      ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

**ReenTrantLock实现的原理：**

在网上看到相关的源码分析，本来这块应该是本文的核心，但是感觉比较复杂就不一一详解了，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。**想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。**

**什么情况下使用ReenTrantLock：**

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

1. 可中断锁
2. 公平锁
3. 多condition条件

另一种就是大数据量并发时优先考虑**ReenTrantLock**