# Java 多线程

# 基础知识：

## Tread基本方法：

Thread.sleep():睡眠,running 🡪 timeWaiting -> runnable

Thraed.yield() 放弃cpu

thread.join() 当前线程加入 thread，状态变为waiting，thread结束之后，当前线程thread变为runnable.

thread.interrupt(): 打断线程的waiting/timeWaiting状态，可以在waitting/timeWaitting执行，也可以在runnable/running状态执行，使用isInterrupted 返回true，一但进入 waiting/timeWaiting状态，那么直接抛出异常；抛出异常之后，interrupt状态自动清除;注意，LockSupport.park()不会抛出异常，而是会放弃恢复挂起线程，并且不会主动清除interrurt状态；

thread.isInterrupted(): 判断当前interrupt状态

Thread.interrupted(): 清除当前interrupt状态（只能自己的线程执行）

## Object基本方法：

前提知识：需要先获取对应object的锁，否则会抛出异常

object.wait()：

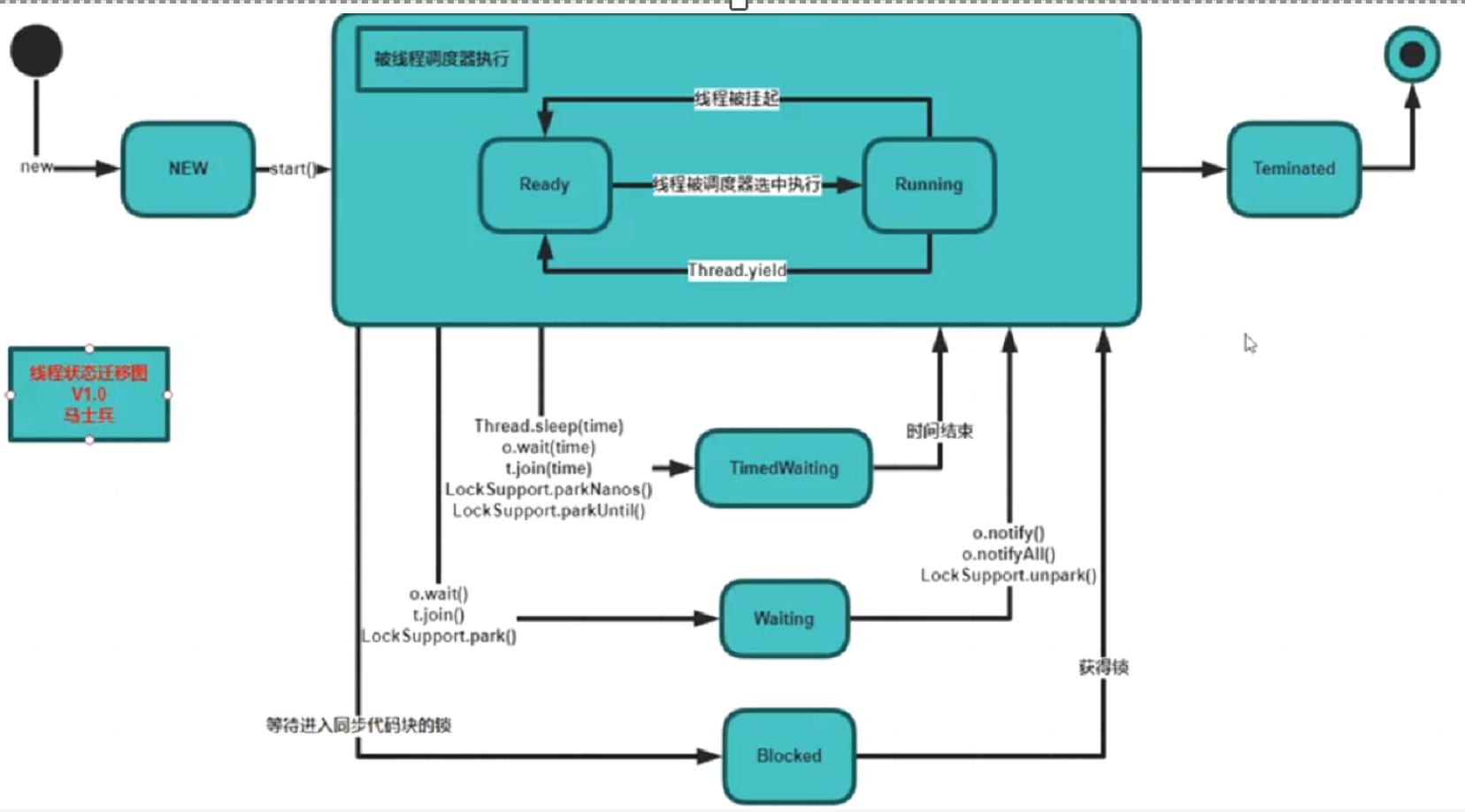
进入object的等待队列中去，直到notify通知唤醒/或者interrupt打断；wait会自动释放锁，被notify唤醒之后，需要重新获取锁才可以继续执行。

object.notify()：

随机唤醒一个wait队列中的线程的ObjectWaiter对象。

## 状态转移：

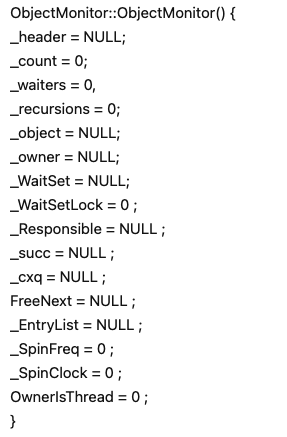
new/runnable/running/waiting/timewaiting/blocked/terminated



# Synchronized：

## 基本原理：

每个对象都对应一个ObjectMonitor对象，其结构如下：



关键字段有：

1:count：重入的次数

2:owner：线程所有者

3:waitSet：执行wait进入等待队列的thread

4:entryList：获取锁失败的，blocked状态的线程

锁的原理：

JVM层使用ObjectMonitor + 底层使用 mutex（二值信号量）

JVM命令：monitorEnter/monitorExit

大致流程：

1:线程进入synchronized，查看owner是否是自己

1-1:如果是自己那么count++，重入锁成功

1-2:如果为空，那么cas修改owner，修改成功，那么count++，获取锁成 功；如果修改失败，进入3

2:把自己加入entryList队列，当前线程状态变为blocked

## 锁优化：

### 1:锁升级

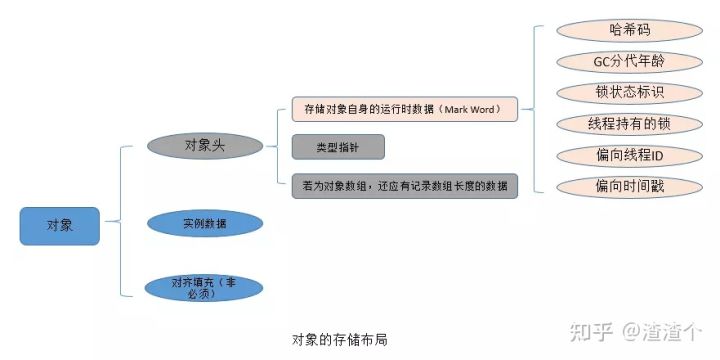
#### 背景：

因为synchronized中使用mutex实现的话，效率比较低，因为mutex使用到了操作系统的二值信号量，需要通过系统调用，涉及用户态->内核态的切换过程，所以为了优化，引入了锁升级。

#### 前置知识：

锁相关的信息是保存在对象中的，具体保存在对象头中，其具体结构如下：

##### 对象：



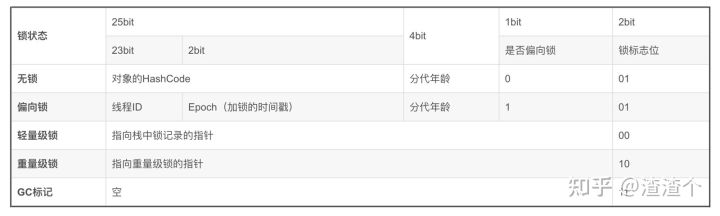
对象头：原信息  
实例数据：对象的数据  
对齐填充：jvm要求对象的起始地址必须是8字节的整数倍，所以对象大小要是8Byte整数倍

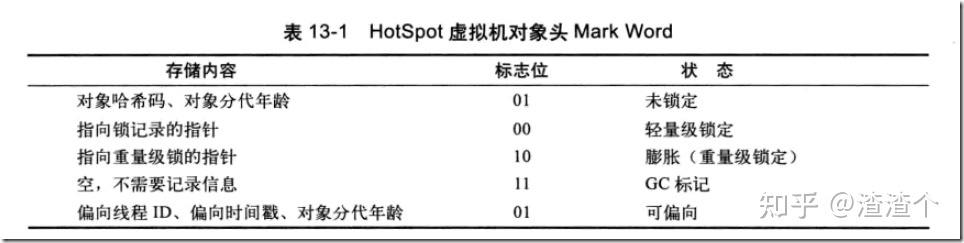
##### 对象头：



类型指针：指向堆区的 Class对象  
数组长度(如果是数组的话才有)  
MarkWord

##### MarkWord：





对象相关的元信息都保存在了对象-对象头-markword中了，其中锁信息就在这

无锁：hash｜分代年龄｜偏向锁｜锁标志

偏向锁：线程id｜epoch｜分代年龄｜锁标志

轻量级锁：LockRecord指针|锁标志

重量级锁：重量级锁指针|锁标志

#### 升级流程：

##### 前置知识：

1:由对象头可以知道，偏向锁会覆盖hashcode，所以如果对象算出了hashcode，那么偏向锁不可以使用，如果此时已经是偏向锁，那么偏向锁升级为重量级锁。

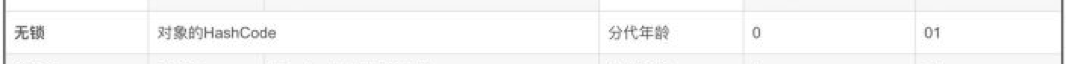
2:我们可以配置是否启用偏向锁

3:锁的升级过程不可逆，但是锁释放之后，就会变为无锁

##### 具体流程：

###### 无锁：

最开始的状态是无锁，此时有线程来获取锁，先判断是否支持偏向锁，如果不支持偏向锁，那么直接申请轻量级锁；如果支持，那么申请偏向锁



###### 偏向锁：

首先查看markword中的线程id是不是自己的，如果是自己的，那么就说明获取锁成功；如果不是自己的，那么判断此线程是否存活（偏向锁不会立即清除锁）；如果不存活，cas修改线程id，获取锁，如果cas失败或者线程存活，那么说明存在竞争，此时需要升级为轻量级锁；升级完成之后，还是由之前的线程获取锁。

释放锁：偏向锁不会主动释放，需要JVM到安全局点（没有字节码执行时），暂定所有的线程，检测持有偏向锁的线程是否存活，不存活则释放偏向锁。



###### 轻量级锁：

当偏向锁由竞争，那么就会升级为轻量级锁

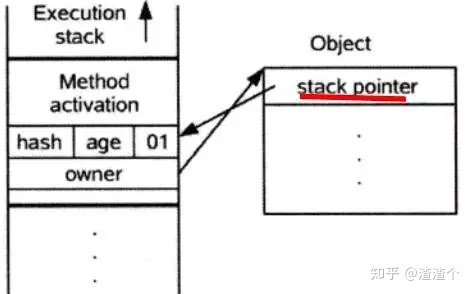
1:线程在栈空间中创建LockRecord对象，对象中保存MarkWord内容+对象头Markword的地址

2:线程cas修改MarkWord中的指针的位置为线程LockRecord地址

3:修改成功那么获取锁成功；修改失败进入4

4:修改失败说明已经有现成持有锁，此时线程进入自旋，cas修改指针，次数达到一定次数之后，升级为重量级锁

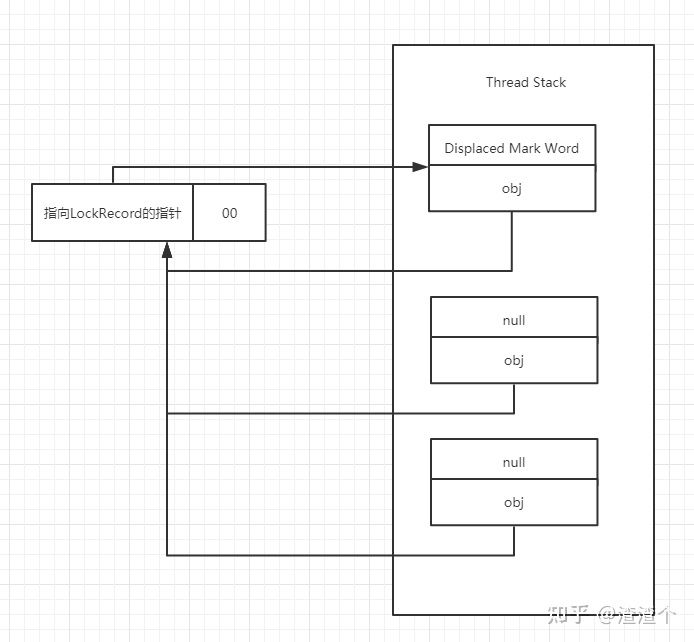




释放锁：

1：通过CAS把LockRecored中的MarkWord备份使用CAS恢复  
2：恢复成功，则成功替换锁  
3：恢复失败，说经锁已经升级，那么释放锁的同时，唤醒挂起的线程。

重入实现：多个LockRecord对象实现

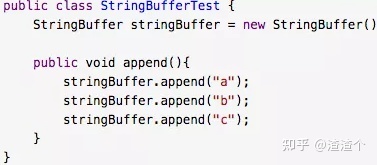


###### 重量级锁：

之前有说到

### 2:锁粗化

### 如果一系列连续的操作都是在同一个对象反复加锁释放锁，那么就会放大锁的范围，避免重复加锁解锁。 Example1：



### 3:锁消除

如果检测到 同步代码块中的数据 不会发生‘逃逸’（不会被其他线程访问到），那么就删除锁

# AQS（AbstractQueuedSynchronizer）

<https://javadoop.com/post/AbstractQueuedSynchronizer>

<https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html>

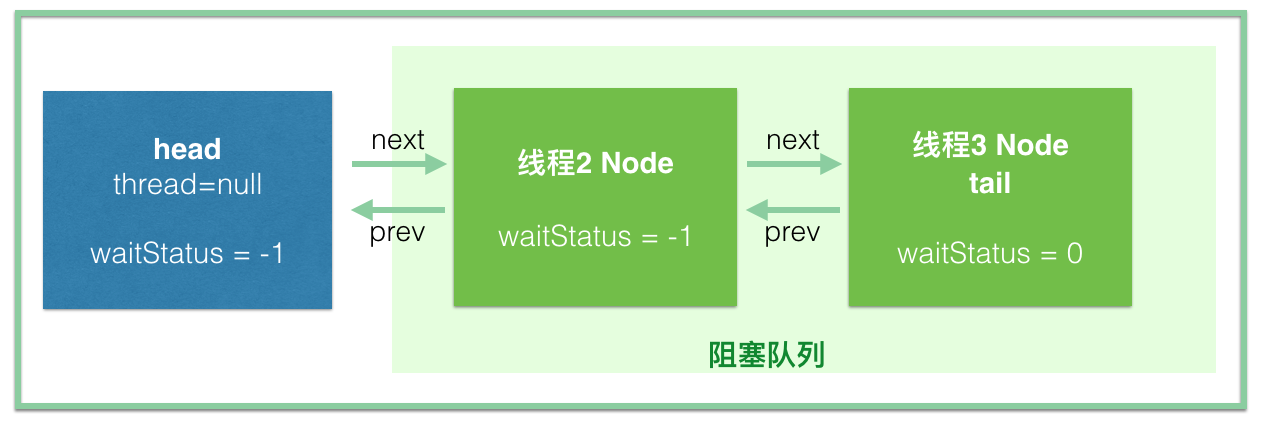
## 核心原理：

### 核心类：

ownerThread：代表当前持有锁的线程

state：重入次数，相当于count

queue：等待锁的Node队列



### 辅助类

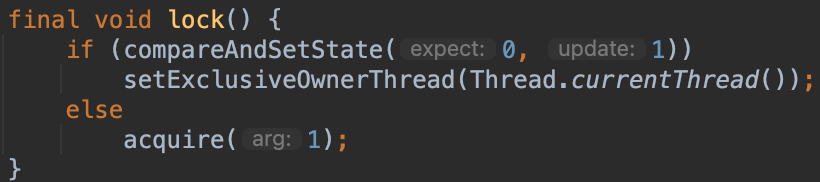
LockSupport：用于挂起当前线程

Node：双向队列的Node节点：由Thread，waitStatus，next，pre组成；其中waitStatus代表着下一个节点是否需要被唤醒

## AQS源码-获取资源：

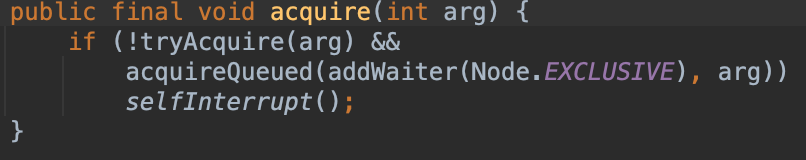
通过ReentrantLock的unfairLock去看

### 1:lock.lock()



上来直接先抢一次锁，不管队列中有没有等待锁的线程。这也就是为什么叫非公平锁。

### 2:AQS.acquire()

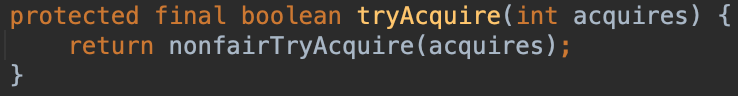


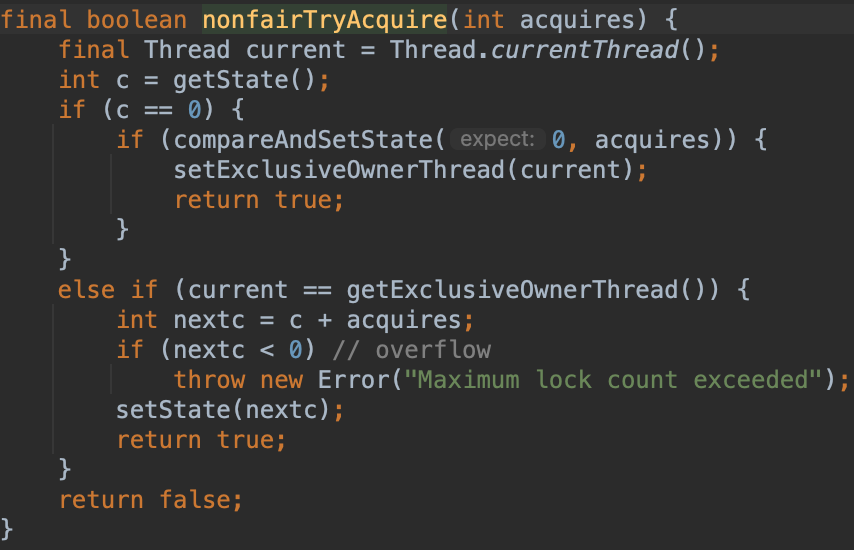
先tryAcquire()获取锁，如果获取成功，直接返回

如果获取失败，那么添加到队列中去

### 3:tryAcquire()

模版涉及模式，tryAcquire()逻辑由子类实现





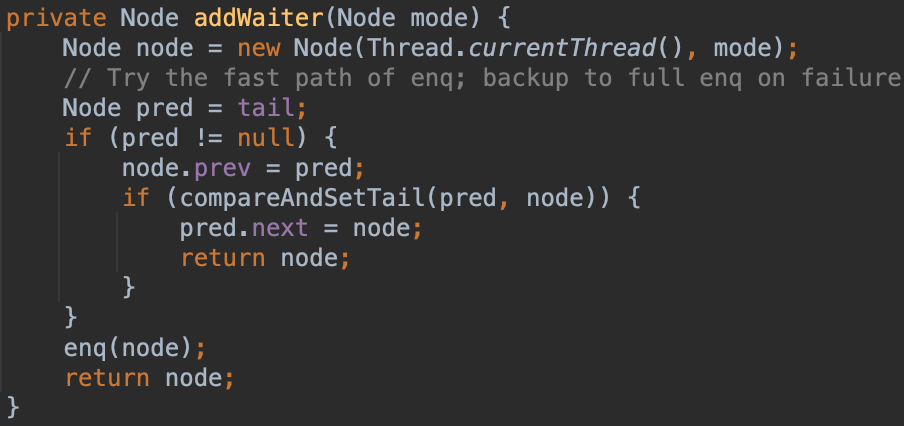
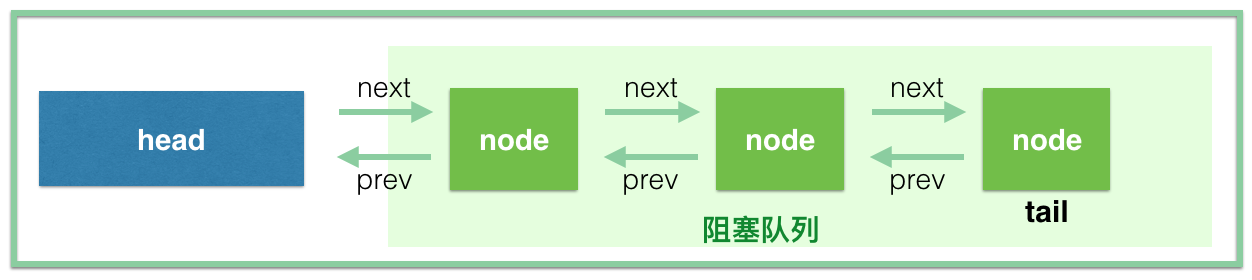
3-1:判断state是不是0，并且队列中头节点的Thread是不是自己，就直接抢一次（Unfair的又一次体现）

3-2:获取失败，判断是不是重入，是的话state++

3-3:返回获取锁失败

### AQS.addWaiter：

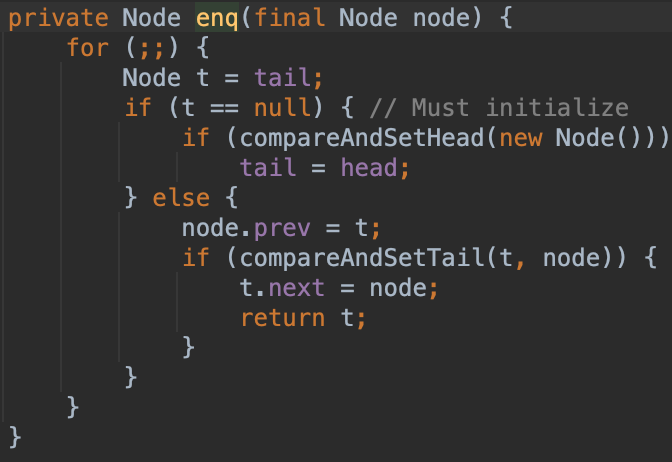
目的入队列，队列结构如下，一个头节点，两个指针，head+tail



1:创建一个Node（Tread+mode）

2:添加到末尾，并返回

3:如果头节点不存在，或者添加末尾失败（存在竞争），那么enq一直加

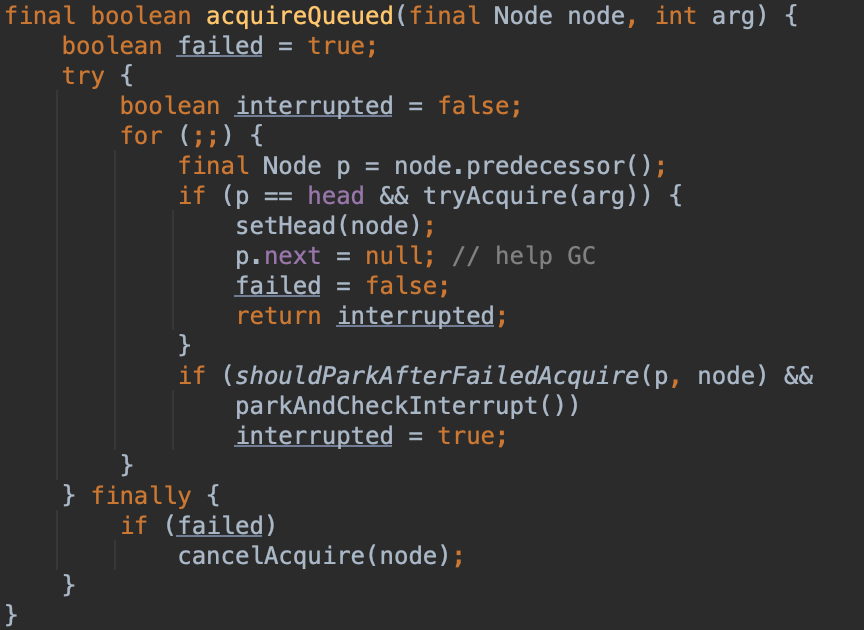


3-1:t==null说明头节点不存在，初始化一个头节点，并把tail也指向head

3-2:否则，不停的cas，把当前节点设置到尾部

### AQS.acquireQueued():

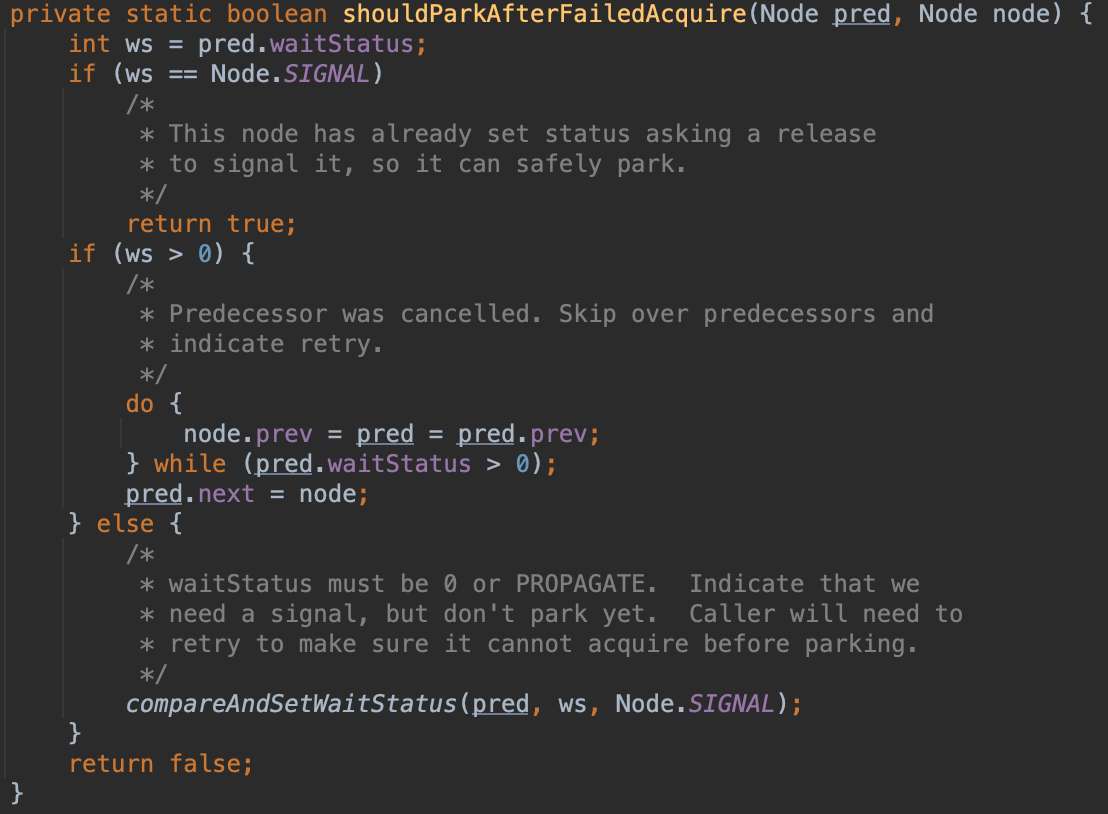
挂起和恢复的逻辑：



1:第一次进来，或者挂起恢复会执行该语句；先判断是不是head，节点，如果是的话，抢一下锁，如果抢成功，返回isInterrupted；

2:否则判断是否需要 park，如果需要则进行park

2-1:判断是否需要park()



判断pre节点的状态是不是 SIGNAL，如果是那么return true

如果>0, 那么说明上一个节点被cancel了，那么一直往前找，一直找一个新爹

否则，设置前一个节点为SIGNAL，return false

return false的话，下次循环还会进来，return true的。

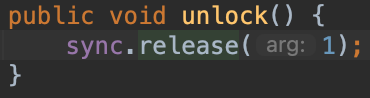
3:park逻辑：

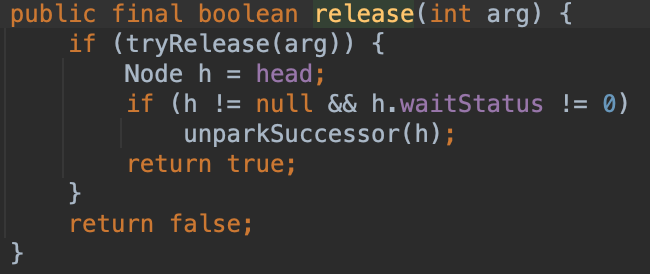


## AQS源码-获取资源：

还是拿ReentrantLock来说：

### ReentrantLock.unlock()->AQS.release()

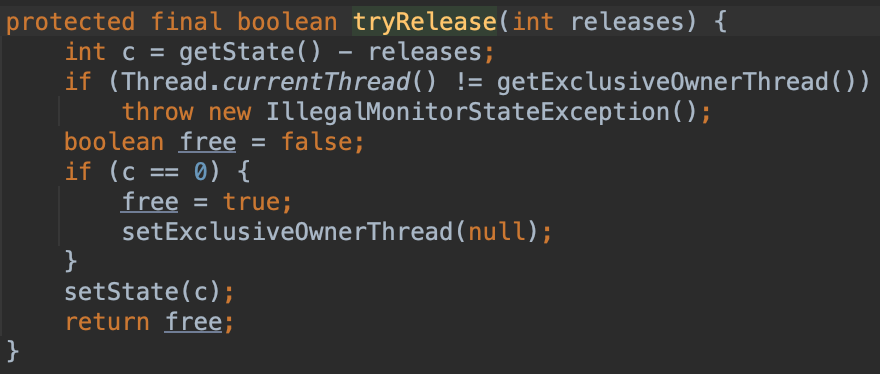




1:释放资源

2:唤醒下一个等待节点

### ReentrantLock.tryRelease()



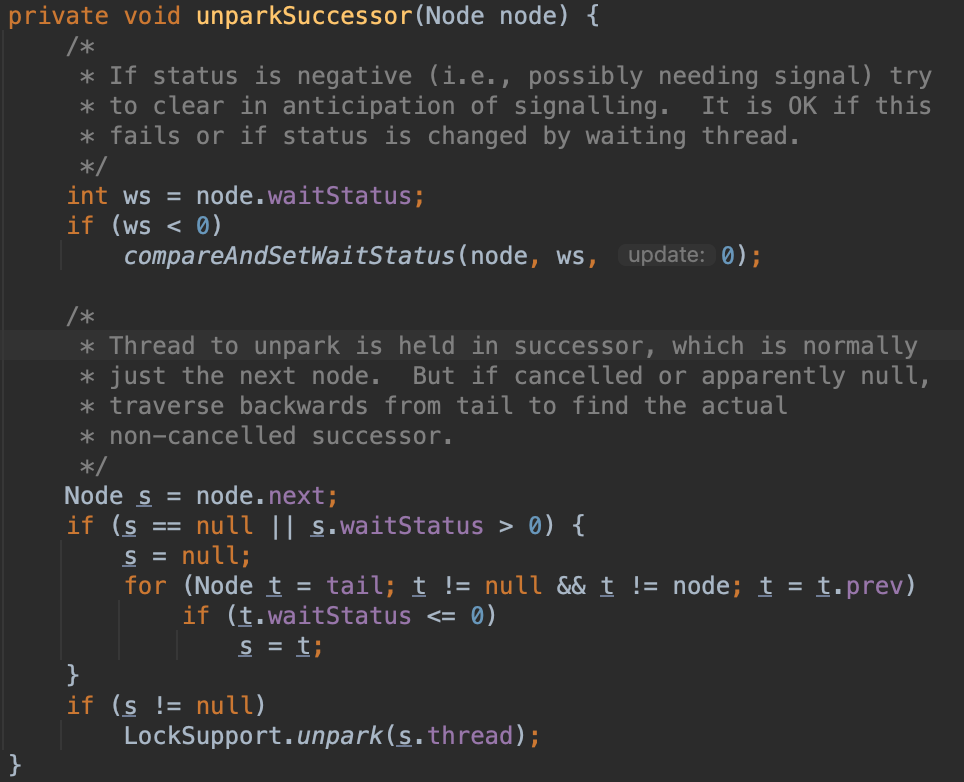
1:判断当前线程是不是锁的owner

2:如果state=0，设置owner为0

3:设置state

4:返回是否已经释放锁（重入的场景下，state != 0是不会释放锁的）

### AQS.unparkSuccessor()



从后往前找，一直找到最后一个没有cancel的节点，调用LockSupport.unpark()方法，唤醒节点

### 从后往前问题：

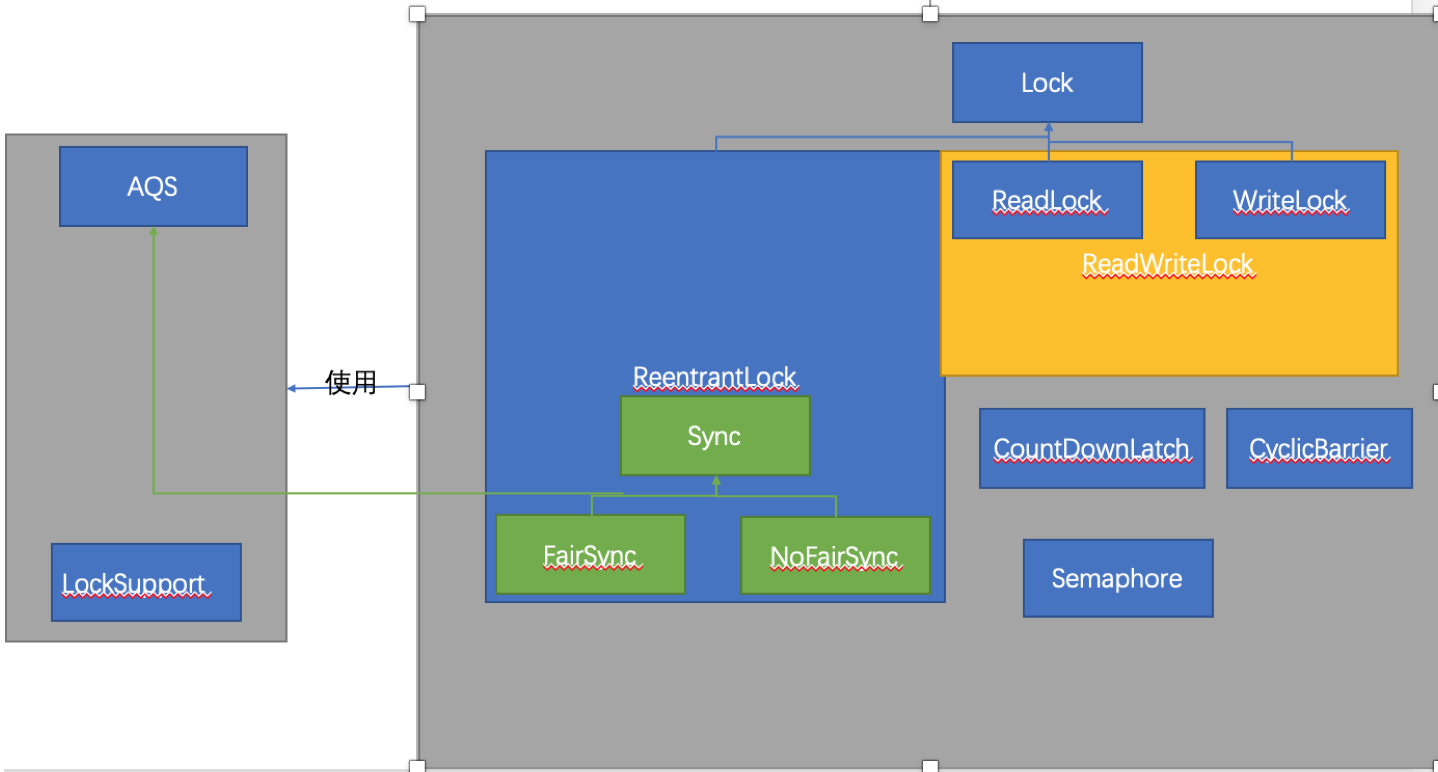
<https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html>

以内在addWaiter的时候，是先node.pre = pre; tail = pre; oriTail.next = tail；

这个过程非原子的，如果还没有执行最后一步，此时唤醒逻辑执行，那么会丢掉最后一个节点。

### SelfInterrupt问题：

因为要保证锁在 挂起的时候（LockSupport.park（））,不能被打断，但是因为interrupt会打断park，所以在打断之后，先清除interrupt状态，在被唤醒之后，再补充一个interrupt。保证interrupt逻辑。



# JUC相关类（ReentrantLock，ReadWriteLock）

## ReentrantLock:

### 锁的使用方式：

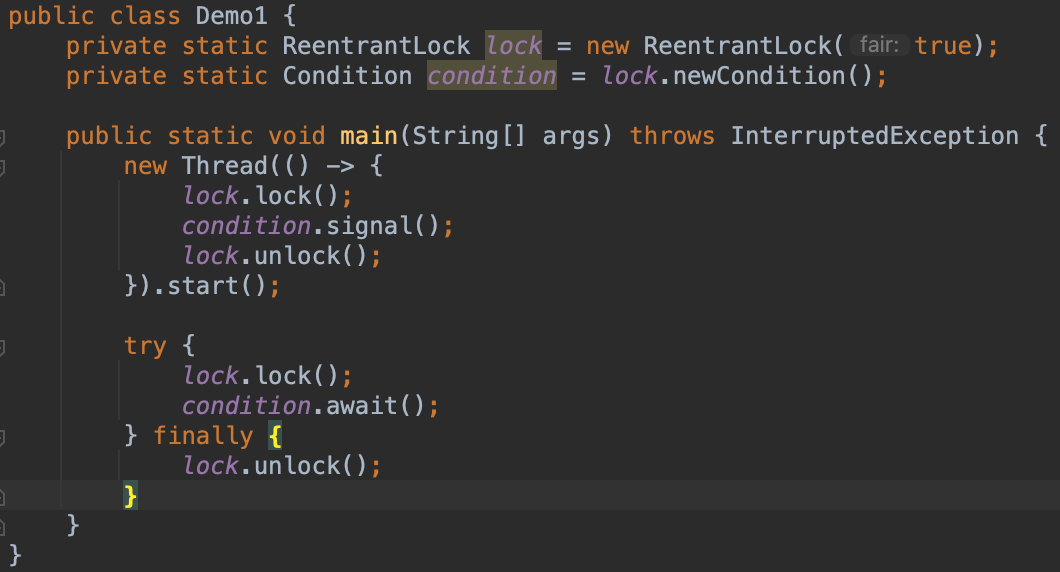
加锁：lock()

释放锁：unlock()



### 线程同步功能使用方式：

使用condition：



ReentrantLock和Synchronized都有加锁，同步的相关功能，reentrant的同步更加灵活一些，可以有多个同步队列，而原始的Synchronized相关的同步队列只有一个，所以在生产消费者模型时，ReentrantLock能创建两个Condiditon，定点同步，更优雅高效。

### ReentrantLock和Synchronized的异同：

1:前者使用JavaAPI实现，后者需要依赖os的二值信号量，所以前者更高效

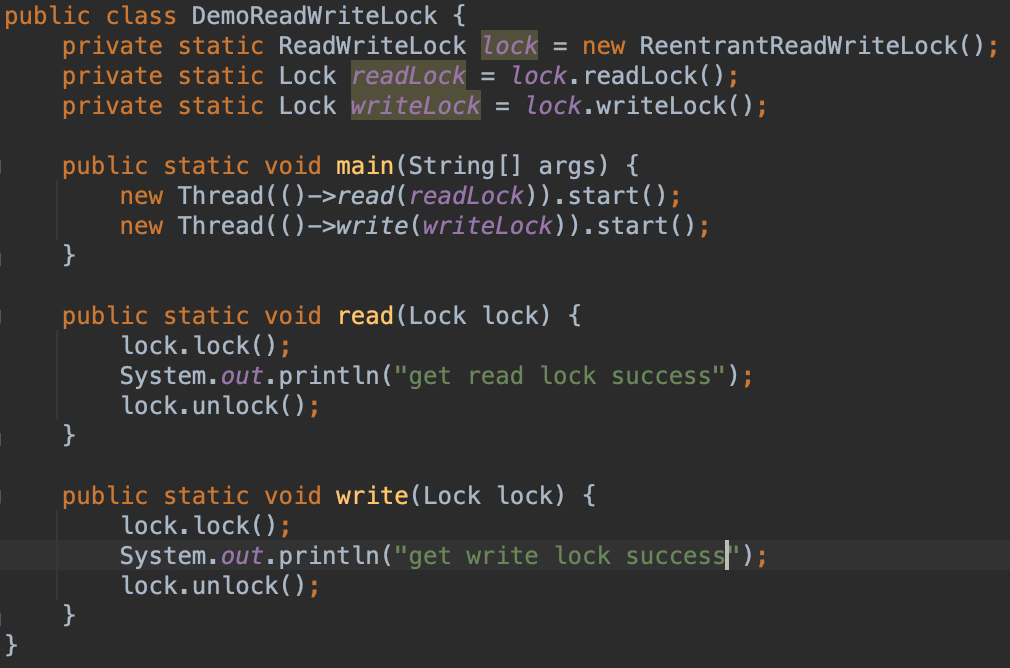
2:在同步功能方面，前者可以有多个同步队列，后者只有一个；两者都需要先获取锁才能await/wait, notify/signal；

3:前者需要手动释放锁（抛异常不会释放锁），后者自动释放

4:前者支持锁可interrupt；还支持设置获取锁的超时时间

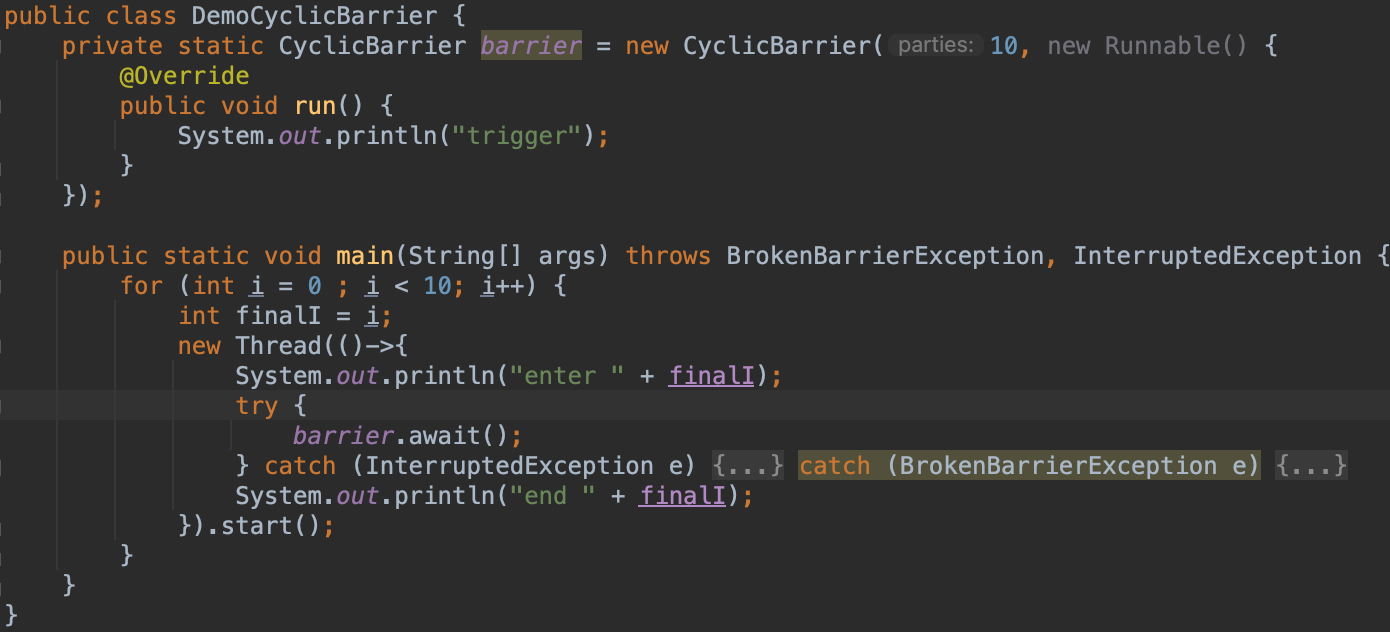
## ReadWriteLock:

背景：读写锁，可以类比mysql的读锁 和 写锁，一样的道理



## CyclicBarrier:

原理：线程栅栏，指定数据的线程执行了await之后，统一放行



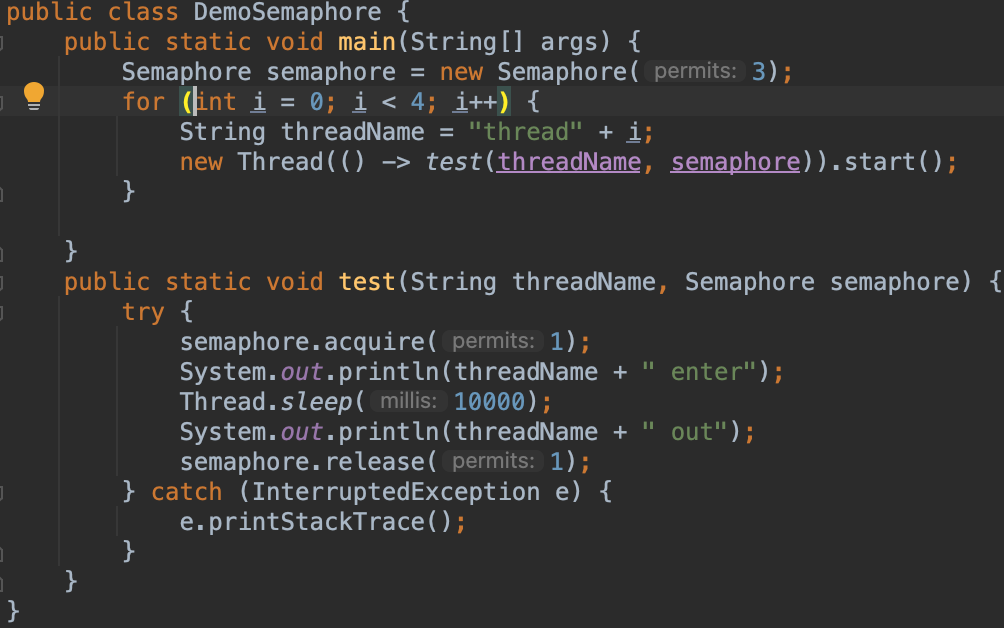
CountDownLatch:

原理：倒计时放行，有初始值，countdown一次，减一次，到0，解放所有的同步Thread



## Semaphore:

使用场景：限流



# 多线程基本概念

### 1:原子性

#### 说明：

同一个代码块，同时只有一个线程进入，这个就是代码块的原子性

#### 实现：

java中，使用锁可以实现原子性

#### 注意：

数据库的原子性，os的原子操作，多线程的原子性 这几个是有区别的，注意区分；

### 2:可见性

#### 说明：

因为现代CPU都是有多个核心，每个core有每个core的缓存，多个线程跑在不同的core中 ，共享的变量修改，是否能保证其他线程修改后可见。这就是可见性

#### 实现：

java中，volatile和锁都可以保证可见性

p s：可以看看CPU的架构，以及操作

### 3:有序性

## 说明：

操作系统为了优化，通常会将指令进行重新排序，在单线程的情况下，os的as-fi-serial保证了单线程情况下的程序正确性。但是多线程的情况下，需要使内存屏障，或者通过加锁，使多线程变单线程解决该问题。

#### 实现：

1:使用volatile，来禁止修饰的对象的相关的操作的指令重排

2:使用锁，多线程-> 单线程来解决

ps：可以看看CPU的指令重拍相关知识，以及总线锁/cache line锁

# Validate

## 背景：

## 用来解决多线程中的可见性 和 有序性的问题

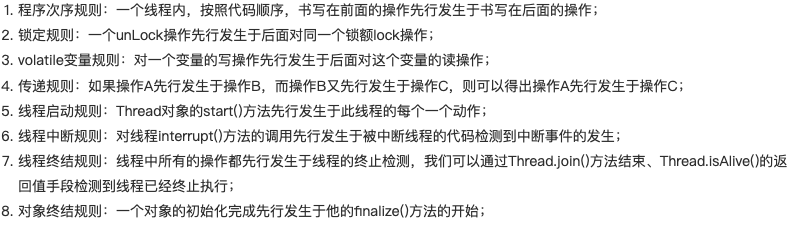
## 原理：

有序性是使用的内存屏障来保障



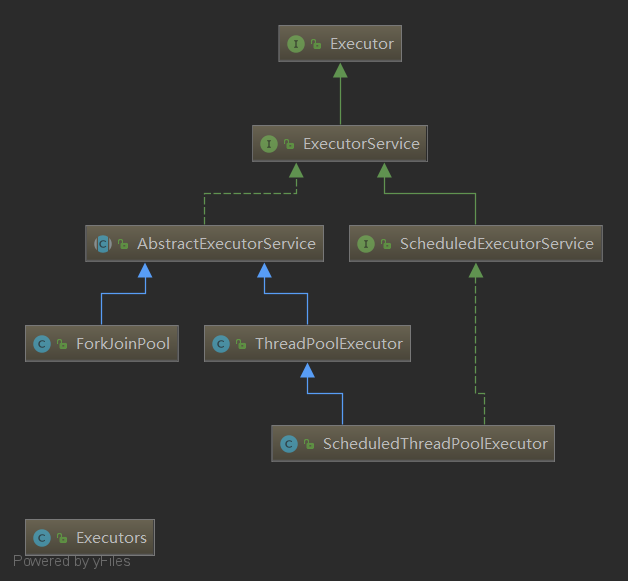
JVM中的内存屏障的实现，其实是底层使用lock 指令，通过锁总线/锁cache line，把volatile对象的相关的操作强制加锁，保证了有序性，和可见性（lock指令会触发MESI缓存一致性协议）。

# happens-before



# 线程池

## 线程池类结构：



1: Executor：执行器基接口

关键方法：execute(Runnable)； 这个方法**没有返回值**

2:ExecutorService：线程池基接口，是线程池的基础类

关键方法：

submit(Runnable/Callback):这个方法**有返回值**

shundown(): 关闭线程池

3:AbstractExecutorService：线程池的抽象类

4:ThreadPoolExecutor：线程池的实现类，最重要！

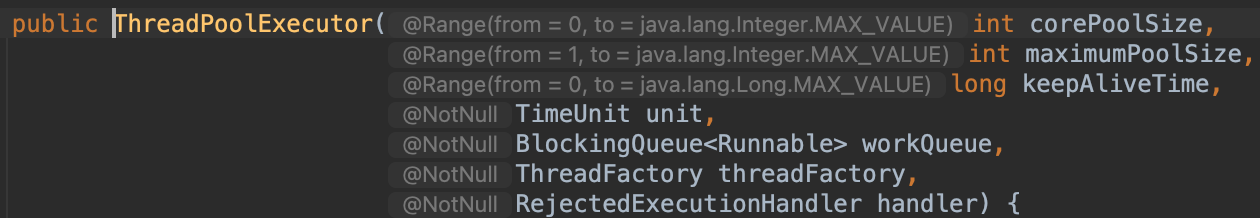
5:ScheduledThreadPoolExecutor：定时任务的线程池

6:Executors：线程池工厂

**注意：不要在业务中使用Executors线程池工厂去创建线程池，因为他创建的线程池会导致OOM，下面会讲到**

## ThreadPoolExecutor：

构造参数：



1:corePoolSize：核心线程数

2:maxPoolSize：最大线程数

3:keepAliveTime：超过最大线程数的线程的生命周期

4:timeUnit：生命周期时间单位

5:BlockQueue：用来存Runnable对象的阻塞队列

6:线程工厂：一般用来做一些线程的定制工作

7:拒绝策略

1:AbortPolicy：抛出异常

2:DiscardPolicy：丢弃

3:DiscardOldestPolicy：丢弃最老的

4:CallerRunnsPolicy：提交线程执行

5:自定义：一般可以持久化到其他地方（mysql），等任务空闲时候再执行

## 几种预置的线程池：

1: SingleThreadExecutor：只有一个线程的线程池，比new Thread来说，可以做一些任务的排序执行，还有线程的生命周期管理；

问题：Executors创建给的阻塞队列是LinkedBlockingQueue，无限大，会OOM；

2:FixedThreadExecutor：固定核心线程大小的线程池

问题：Executors创建给的阻塞队列是LinkedBlockingQueue，无限大，会OOM；

3:CachedThreadPool：core线程为0，实时创建线程，超过一定时间没有任务，那么回收

问题：Executors创建给的最大线程数，无限大，会OOM；

3:ScheduledThreadPool：可以做一些定时任务，或者延迟任务的线程池

## 阻塞队列：

### 队列：Queue

add()：添加元素，如果满了，抛出异常

remove()：取出元素，如果没有，抛出异常

offer()：添加元素，如果满了返回false，添加成功 返回true

poll()：取出元素，如果没有返回null

### 阻塞队列：BlockingQueue

put(): 添加元素，如果满了，阻塞

take()：取出元素，如果没有，阻塞

### 线程池中常用到的阻塞队列：

1：LinkedBlockingQueue：链式阻塞队列，无限长

2：ArrayBlockingQueue：数组式的阻塞队列，固定长度

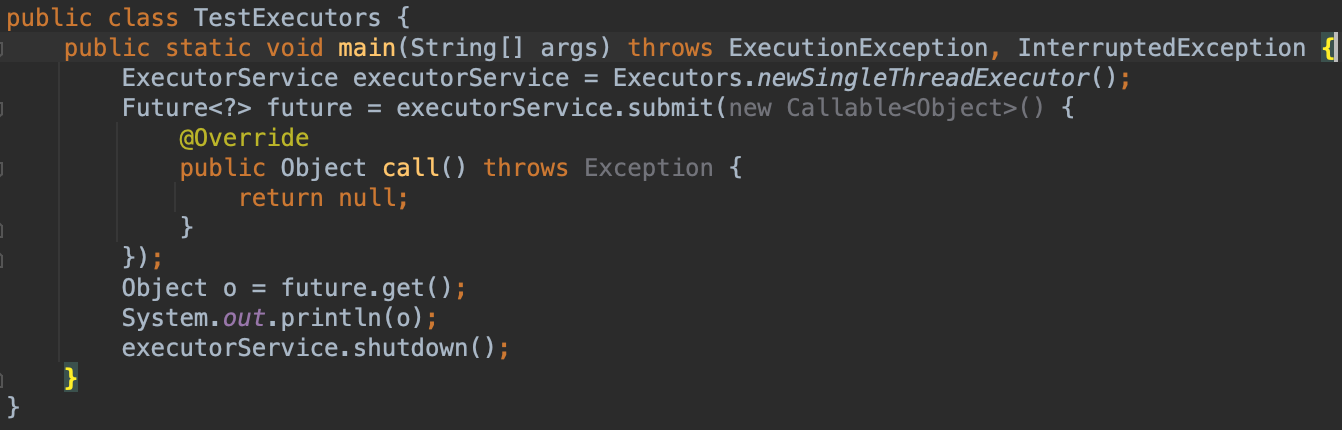
3：DelayQueue：按时间进行排序优先队列，无限长

4：PriorityBlockingQueue：按照指定优先级进行排队的阻塞队列，无限长

## 线程池相关的类

### Feature:

ExecutorSerivce.submit()函数的返回值，获取线程返回的数据，需要配置Callable使用

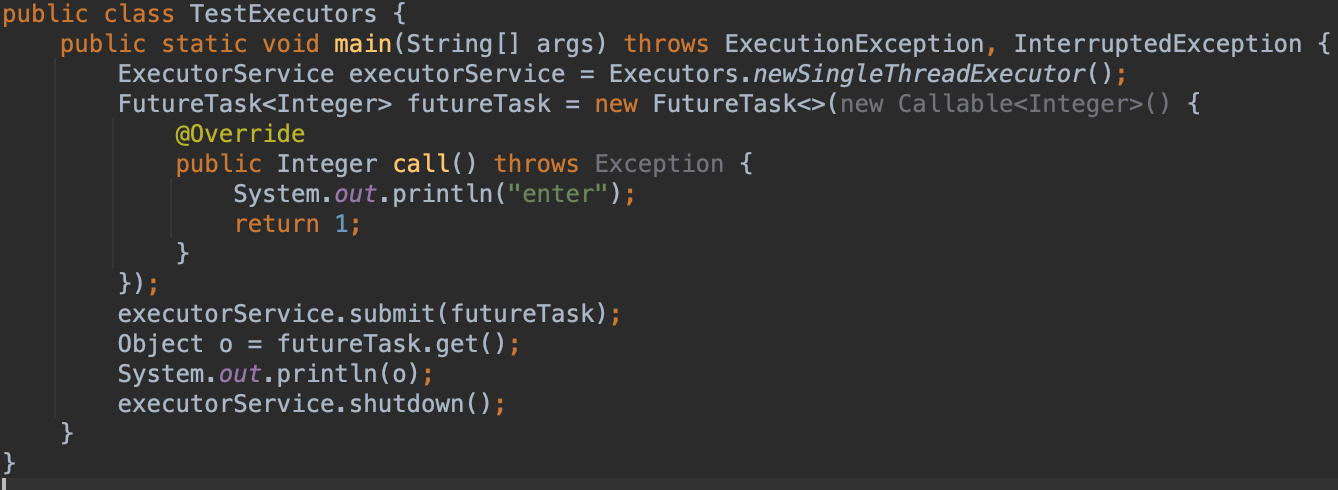


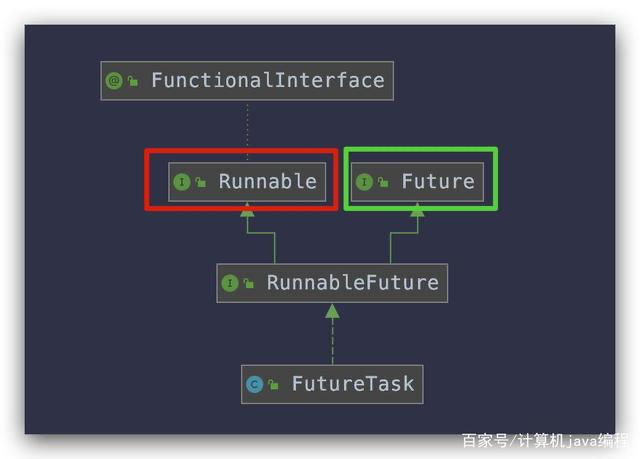
### Callable:

在ExecutorService.submti需要返回值时，传入Callable对象，而不是Runnable。见上图

## FutrueTask：

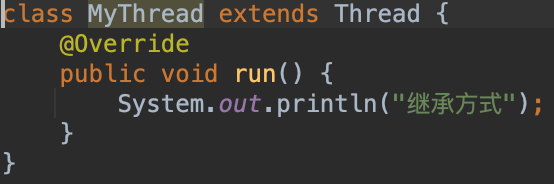
Runnable和Future的结合体，构造参数支持Callable或者Runnable+Object



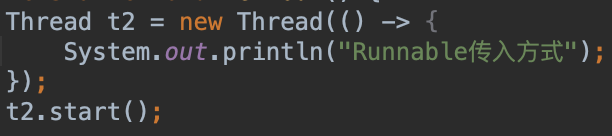


## 创建线程的几种方式

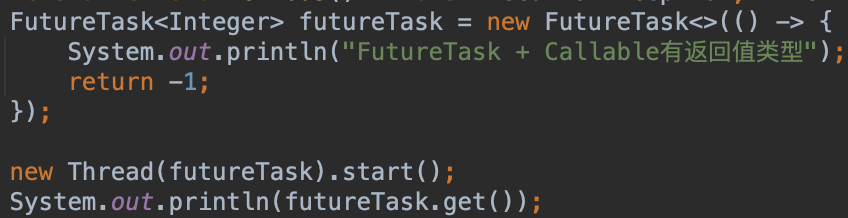
1:继承方式



2:传入Runnable方式



3:有返回值的FutureTask+Callable方式



# ThreadLocal

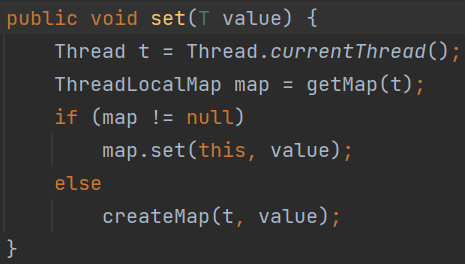
## 背景：

线程级别地共享数据，在一个线程内部可以共享。

## 原理：

Thread中会维护一个ThreadLocalMap，map存储的是key：ThraedLocal；value：ThreadLocal.set的值

## Thread.set()逻辑：



1:先获取当前调用set的Thread

2:获取Thread的map

3:给map添加一个key/value对，其中key为ThreadLocal

**注意：一个ThreadLocal对象可以存在于多个线程的map中，实际存储的value的地方是thread的ThreadLocalMap中，而非ThreadLocal中，ThreadLocal只是作为一个key。不同的Thread中同一个ThreadLocal作为key对应的value肯定不同。**

## ThreadLocal可能会存在内存的问题：

### ThreadLocal存在泄漏：

当ThreadLocal不需要之后，会设置为null，如果当前线程还没有结束，的话，ThreadLocal设置为null是没有用的，因为使用了ThreadLocal的Thrad的ThreadLocalMap中，还是存有ThreadLocal。具体路径是：Thread –> ThreadLocalMap -> Entry -> value

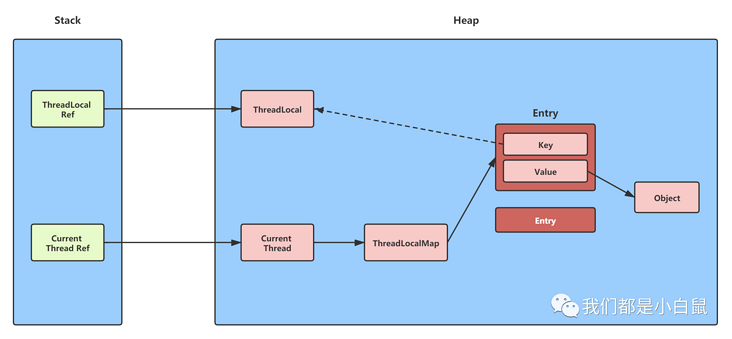
解决办法：将key（ThreadLocal）对象使用WeckReference修饰，当设置为null之后，TheadLocal就只是弱引用了，就可以被GC；

**编码时，我们只需要将ThreadLocal设置为null既可**

### value存在泄漏：

如果ThreadLocal不使用了，只设置ThreadLocal为null是没用的，因为实际数据时存储在每个线程的ThreadLocalMap中的Entry中。具体路径是Thread –> ThreadLocalMap -> Entry -> value

解决办法：如果不使用ThreadLocal对象了，那么在每个使用了对应的ThreadLocal对象的线程中，调用该threadLocal.set(null) 清除该value即可。



### 总而言之：在不使用ThreadLocal对象时

#### 1：保证每个使用了该threadLocal的线程执行了threadLocal.set(null)

#### 2：将threadLocal的引用设置为null

# 死锁：

