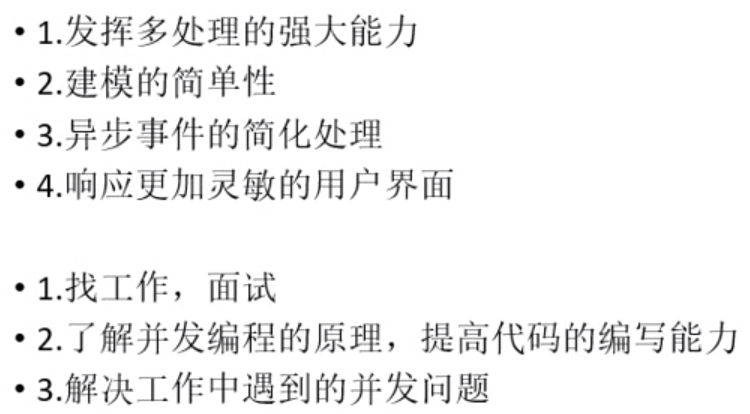
* 1. **并发编程的学习**
     1. 并发编程API的熟练使用
     2. 并发编程API的实现原理
     3. 理解java虚拟机的内存模型
     4. 了解操作系统对并发的支持
  2. **并发编程的必要性**

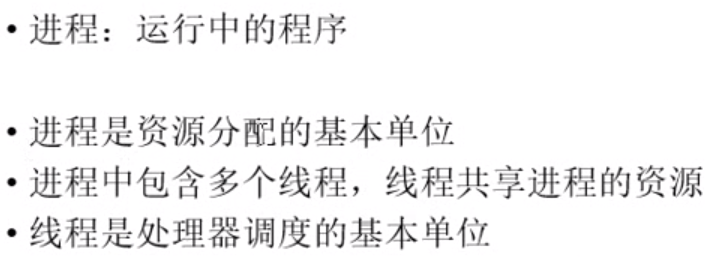


* 1. **多线程和并发**

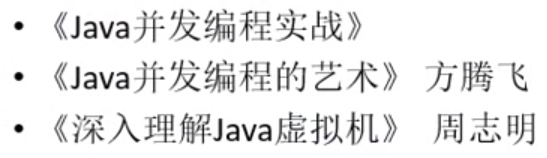
多线程：多个执行流，顺序切换执行（不是同一时刻执行，只是切换速度快）

并发： 多线程实现并发（区别并行，并行指两个或两个以上事件（或线程）在同一时刻发生，多处理器上同一时刻运行）

* 1. **进程和线程**

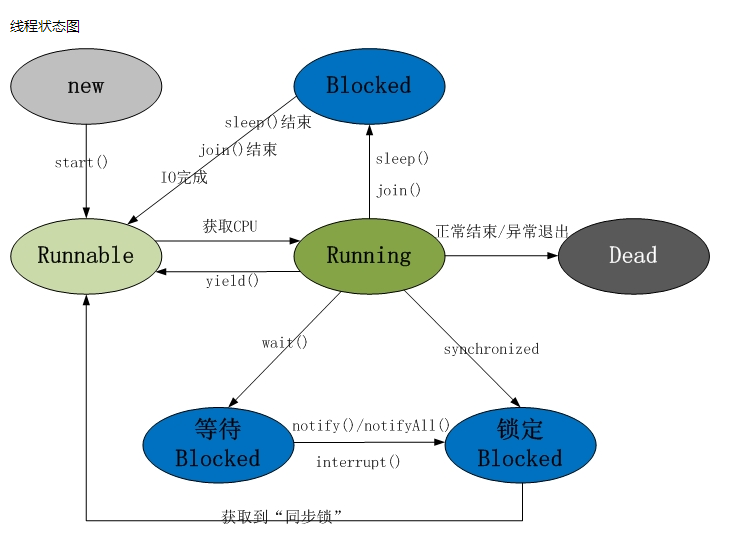


* 1. **java并发编程书籍**



**1.6线程状态的转换（**<https://blog.csdn.net/xingjing1226/article/details/81977129>**）**





Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“当前线程”等待，因为线程进入等待状态，所以线程应该释放它锁持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
OK，线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“**唤醒线程**”)，它只有在获取“该对象的同步锁”(**这里的同步锁必须和等待线程的同步锁是同一个**)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

* 1. **创建线程的方法**



* 1. **守护线程**(deamon属性) <https://blog.csdn.net/u010739551/article/details/51065923>

Daemon线程是一种支持类型的线程，因为它主要被用作程序中后台调度以及支持性工作，这意味着，当一个java虚拟机中不存在非Daemon线程的时候，JAVA虚拟机将会退出，可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程

注意：Daemon属性需要在启动线程之前设置，不能再线程启动之后设置！

注意：在构建Daemon线程是，不能依靠finally块中的内容来确保执行关闭或者清理资源的逻辑

* 1. **有返回值的线程，实现Callable接口，返回值通过Future获取**

<https://segmentfault.com/a/1190000014940144>

为什么要引入Executor框架？

如果使用new Thread(...).start()的方法处理多线程，有如下缺点：

* 开销大。对于JVM来说，每次新建线程和销毁线程都会有很大的开销。
* 线程缺乏管理。没有一个池来限制线程的数量，如果并发量很高，会创建很多的线程，而且线程之间可能会有相互竞争，这将会过多得占用系统资源，增加系统资源的消耗量。而且线程数量超过系统负荷，容易导致系统不稳定。

使用线程池的方式，有如下优点：

* 复用线程。通过复用创建的了的线程，减少了线程的创建、消亡的开销。
* 有效控制并发线程数。
* 提供了更简单灵活的线程管理。可以提供定时执行、定期执行、单线程、可变线程数等多种线程使用功能。

使用ExecutorService来创建线程池，可以使用CPU核心数来作为线程数

parallism = Runtime.getRuntime().availableProcessors(); // CPU的核心数

**2.0 Spring对创建线程的支持：使用@Async和@EableAsync注解实现**

**2.1 lambda表达式（并行流）实现并行处理**

**2.1 活锁、死锁和饥饿**

<https://blog.csdn.net/huangshulang1234/article/details/79158306>

活锁：各个线程互相谦让释放资源的锁，没有形成阻塞，但是各个线程又不能获得资源得以继续往下执行。

死锁：各个进程都占有对方的资源的锁，同时又等待各线程释放锁，导致处于一直等待状态

饥饿：优先级高的线程一直抢占优先级低的线程的CPU时间片，导致优先级低的线程无法执行

**2.2 如何避免饥饿问题**

设置合理的优先级

使用锁代替synchronized

**2.3 线程安全问题出现在什么情况下？**

（1）多线程环境下

（2）多个线程共享资源

（3）对共享资源进行非原子性操作

**2.4 javap –verbose \*\*\*.class 反编译，查看class文件的字节码**

**2.5 synchronized原理和使用**

<https://www.cnblogs.com/paddix/p/5367116.html>

内置锁（互斥）

修饰普通方法：作用域是某个对象实例，防止多个线程同时访问此对象中被synchronized修饰的方法，但是不同对象实例是不互相干预的；

修饰静态方法：作用域是类，多个线程不能同时访问此类的所有对象实例的synchronized方法（synchronized不能继承）

修饰代码块

原理：

修饰在代码块时：

（1）每个对象都有一个monitor对象，当monitor对象被占用时就会处于被锁定状态，需要通过monitorenter指令获取monitor对象，通过monitorexit指令释放

当monitor进入数为0，线程执行monitorenter指令，则获得monitor对象，同时将monitor设置为1,；

当线程已经占有monitor对象时，进入数+1（重入）；

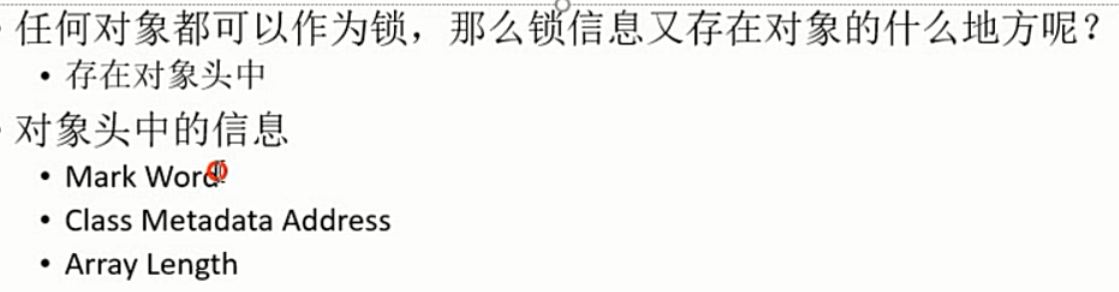
当其他线程占有monitor对象时，该线程阻塞，知道monitor进入数为0，再重新尝试获取

修饰静态方式时，道理一样

当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。

**2.6 无锁、偏向锁、轻量级锁、重量级锁**

<https://blog.csdn.net/lengxiao1993/article/details/81568130>



Java偏向锁(Biased Locking)是Java6引入的一项多线程优化。它通过消除资源无竞争情况下的同步原语，进一步提高了程序的运行性能。

偏向锁存在的意义：

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入偏向锁。

当一个线程访问同步代码块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程再进入和退出同步块时不需要进行CAS操作来加锁和解锁

现在几乎所有的锁都是可重入的，也即已经获得锁的线程可以多次锁住/解锁监视对象，  
按照之前的HotSpot设计，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用，  
因此偏向锁的想法是一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个线程，之后的多次调用则可以避免CAS操作，  
说白了就是置个变量，如果发现为true则无需再走各种加锁/解锁流程。

它会偏向于第一个访问锁的线程，如果在接下来的运行过程中，该锁没有被其他的线程访问，则持有偏向锁的线程将永远不需要触发同步。

如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会尝试消除它身上的偏向锁，将锁恢复到标准的轻量级锁。  
(偏向锁只能在单线程下起作用)

轻量级锁加锁过程：

（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。

（2）拷贝对象头中的Mark Word复制到锁记录中。

（3）拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（4），否则执行步骤（5）。

（4）如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态

5）如果这个更新操作失败了，说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁

**2.7 单例模式和多线程安全（这个看一看项目练习）**

恶汉模式不会出现线程安全问题（因为即使在多线程下，没有出现对共享资源的非原子性操作）

**2.8 可重入锁（避免死锁）**

当一个线程得到一个对象锁后，再次请求此对象锁时是可以再次得到该对象的锁。这也证明在一个Synchronized方法/块的内部调用本类的其他Synchronized方法/块时候，是永远可以得到锁的。

**2.9 volitile关键字（比synchronized更轻量化，实现资源对多线程的可见性）**

可见性：当一个线程修改了共享变量的值，另一个线程能够读到这个共享变量修改后的值

**1.volatile关键字的两层语义**

　　一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

　　1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

　　2）禁止进行指令重排序。

实现原理：

加了volatile关键字的变量，创建对应对象实例时，会多一个Lock指令。Lock指令主要实现两个功能：

* 1. 将当前处理器缓存行的内容写会内存
  2. 这个写回内存的操作会使其他处理器中缓存了该内存地址的数据失效
* 磁盘——内存——CPU缓存

从而达成可见性（当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。）

缓存一致性协议。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取

3.0 JDK提供的原子类及其原理

原子更新基本类型

原子更新数组

原子更新抽象类型

原子更新字段

/\*\*

\* Atomically increments by one the current value.

\*

\* @return the updated value

\*/

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

**非阻塞同步：**

在并发环境下，某个线程对共享变量先进行操作，如果没有其他线程争用共享数据那操作就成功；如果存在数据的争用冲突，那就才去补偿措施，比如不断的重试机制，直到成功为止，因为这种乐观的并发策略不需要把线程挂起，也就把这种同步操作称为非阻塞同步（操作和冲突检测具备原子性）。

3.1 Lock接口

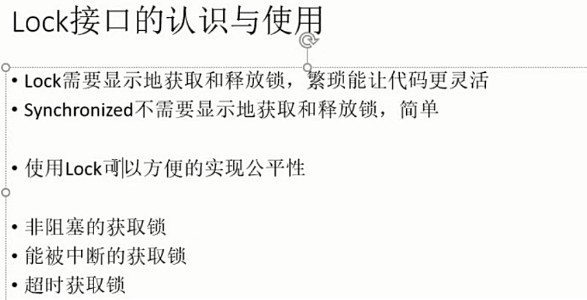
synchronized的缺陷：

当一个线程获得锁，其它线程只能等待，直到获得锁的线程释放锁。而释放锁只能是两种情况：

* 1. 同步代码块执行完毕
  2. 抛出异常由jvm自动释放

当获得锁的线程因等待I/O或sleep被阻塞时，其它线程只能一直等下去

而Lock能让等待的线程响应中断



**3.1 AbstractQueuedSynchronizer（重要）**

在java.util.concurrent.locks包中，作为提供一个框架，用于实现依赖先进先出（FIFO）等待队列的阻塞锁和相关同步器。

反正辅助ReentrantLock的实现：

阅读源码，探究ReentrantLock的lock()和unlock()的实现原理：（与自己实现可冲入锁的原理非常类似）

ReentrantLock的lock()调用同步器sync的lock()，而ReentrantLock中有两个同步器，分别是Nonfairsync和FairSync，它们都继承了AbstractQueuedSynchronizer类，读api发现，继承这个类都要同时重写几个方法

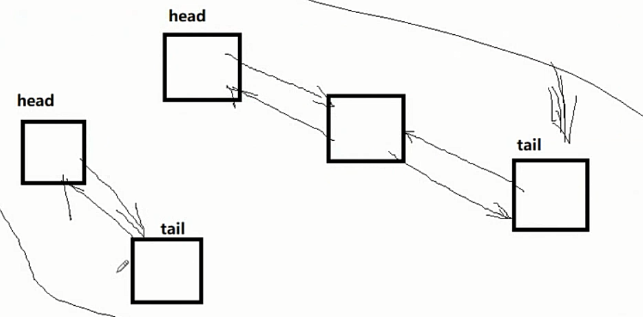
接着来看Fairsync的lock():它调用acquire(1)，传入1；

acquire(1):这个方法是AbstractQueuedSynchronizer的方法，这个方法又调用tryAcquire(1)，FairSync重写了此方法

tryAcquire(1)：通过getState()获得状态（标志是否重入）,若为0,则可以获得锁（返回true）,并将state加1，把当前获得锁的线程设为独占锁的线程；当不为0，判断当前线程是否是独占锁的线程（是否属于重入），是，则state再加1；其他情况返回false;

acquire(1)返回false时（即不能获得锁，接下来应该阻塞并放入同步队列）,继续执行acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))—>此方法应该是放入同步队列，放入后在执行*selfInterrupt*();将线程设为阻塞。

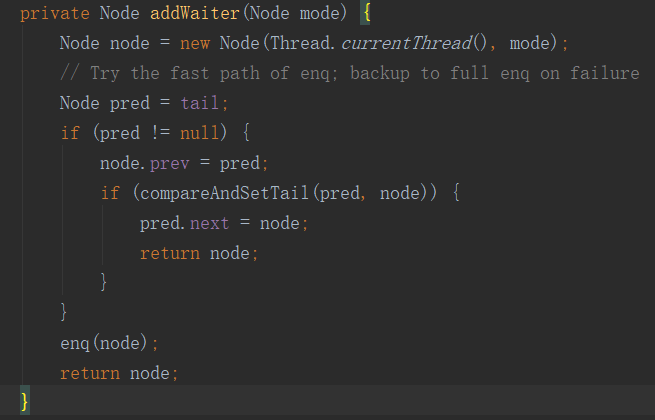
加入队列如下图：（采用自旋的方式进入队列，知道进入阻塞队列）

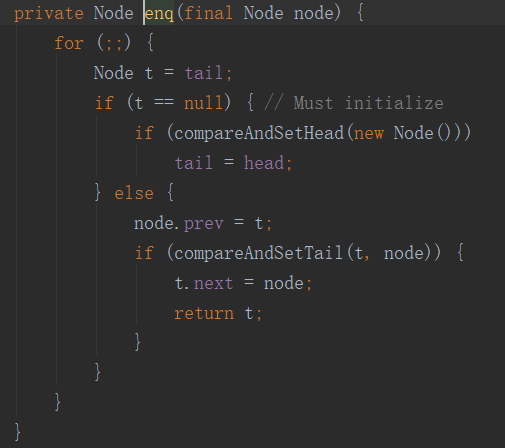


其中一个方法有点奇妙：

acquireQueued的主要作用是把已经追加到队列的线程节点（addWaiter方法返回值）进行阻塞，但阻塞前又通过tryAccquire重试是否能获得锁，如果重试成功能则无需阻塞，直接返回。

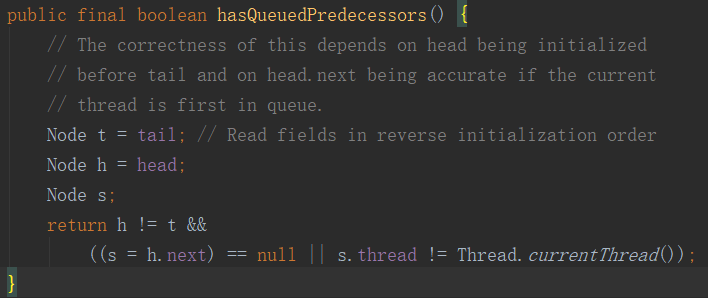
下面是将结点加入同步队列的源码：





**3.2 公平锁**

参考前面的FairSync和UnfairSync同步器源码，主要区别就是在实现上多了一个判断hasQueuedPredecessors()：判断等待队列中有没有比当前请求线程更前的结点。（等待队列就是一个FIFO队列）

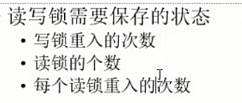


**3.3 读写锁**

我们通过分析ReentrantReadWriteLock源码来分析读写锁。

注意：里面的锁的实现都是委托同步器实现的，而同步器又是通过继承AQS实现（AbstractQueuedSynchronized）实现，所以最终要的是看AQS中的tryAcquire()、tryAcqureiShared()、tryRelease()、tryReleaseShared()方法的实现

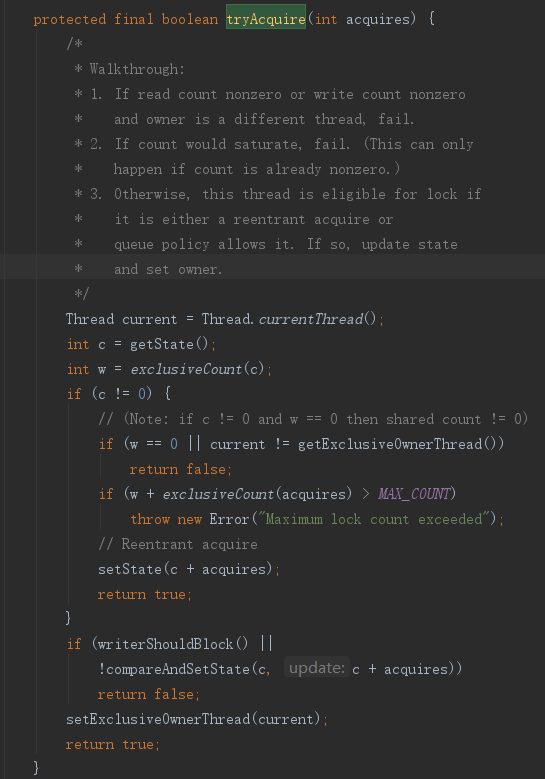
前面的可重入锁用一个int值（state）来表示锁的状态，数值可以表示同一线程重入的次数，那对于读写锁，有几个状态需要解决：





接下来，我们来读一下源码：

**获取写锁（排它锁）：**



int，16位，高16位表示读，低16位表示写状态（数值表示重入次数）

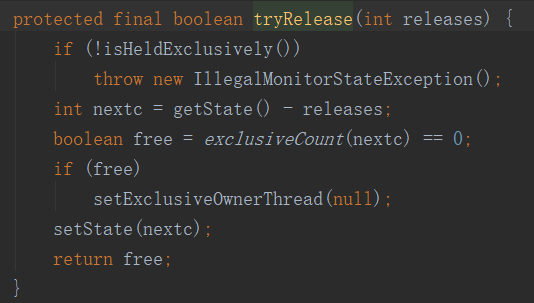
首先getState()获取c（表示是否已被占有或是否第一次进来）

若c==0，则compareAndSetState(c, c + acquires)，设置状态为1，再设置当前线程占用锁setExclusiveOwnerThread(current)；

若c!=0，比较写锁重入数w = *exclusiveCount*(c)，当c!=0且w==0时，即表示已有线程获取该对象读锁（shared count != 0），则当前线程获取写锁失败，return false；

否则设置状态setState(c + acquires)

释放写锁（排它锁）：



isHeldExclusively()判断当前线程是否独占这把锁，若不是，报错（若当前线程不是独占这把锁又怎么能释放它呢）。

如果是，getState() – releases，再判断状态是否是0，若为0，则为线程执行完毕；若不是，则为线程之前重入，现在退出而已；

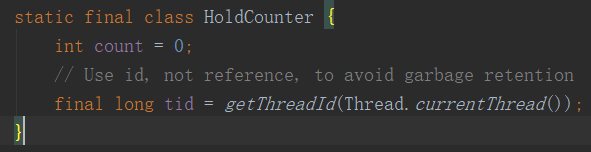
获取读锁（共享锁）：

*exclusiveCount*(c) != 0 &&getExclusiveOwnerThread() != current

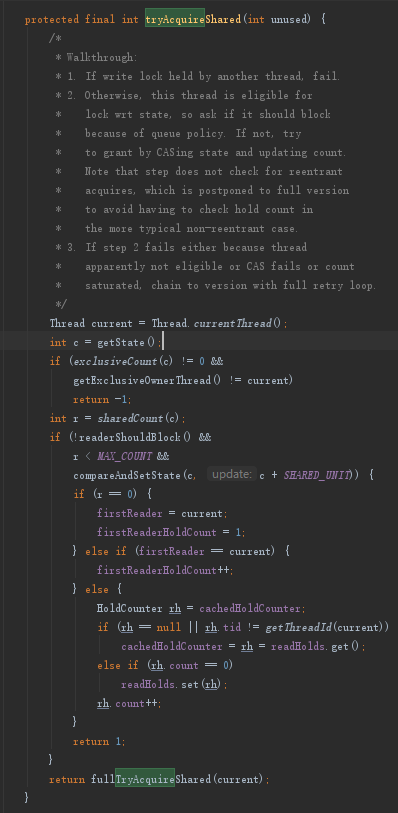
上述若为真，表示资源已被线程获得写锁且非当前线程，获取读锁请求失败；

readerShouldBlock()：保证公平，若不被阻塞，compareAndSetState(c, c + *SHARED\_UNIT*))更新c，

当c==0，设拿到锁的线程为当前线程，并设重入数为1，firstReader = current;firstReaderHoldCount = 1;若firstReader == current，firstReaderHoldCount++;其他情况表示其他线程获取读锁，则使用HoldCounter计重入数



而HoldCounter是用ThreadLocal进行存储，保证安全性



**3.4 锁降级和锁升级**

锁降级：将写锁降级为读锁

即当前线程在写锁没有释放的时候，获取读锁，再释放写锁

锁升级：将读锁升级为写锁

在当前线程没有释放读锁的时候，获取写锁，在释放读锁

锁降级的必要性：

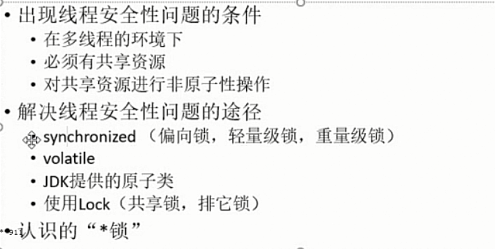
锁降级中读锁的获取是否必要呢？答案是必要的。主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁， 假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

  这时因为可能存在一个事务线程不希望自己的操作被别的线程中断，而这个事务操作可能分成多部分操作更新不同的数据（或表）甚至非常耗时。如果长时间用写锁独占，显然对于某些高响应的应用是不允许的，所以在完成部分写操作后，退而使用读锁降级，来允许响应其他进程的读操作。只有当全部事务完成后才真正释放锁。

所以总结下锁降级的意义应该就是：在一边读一边写的情况下提高性能。

这里说的【数据的可见性】并不是说内存可见性问题，而是指写锁操作的数据结果，对同一个方法里的读锁读取的时候不可见。假设线程A修改了数据，释放了写锁，这个时候线程T获得了写锁，修改了数据，然后也释放了写锁，线程A读取数据的时候，读到的是线程T修改的，并不是线程A自己修改的，那么在使用修改后的数据时，就会出现错误的结果。书上说的【当前线程无法感知线程T的数据更新】，是说线程A使用数据时，并不知道别的线程已经更改了数据，所以使用的是错误的结果。

**3.5 线程安全总结**



有什么锁？

偏向锁

轻量级锁

重量级锁

自旋锁

读锁（共享锁）

写锁（排他锁、独占锁）

活锁

死锁

公平锁

非公平锁

**3.6 线程间的通信**

wait()：必须在同步块之中，或者synchronized修饰的方法中，执行完之后会释放对象锁

notify():随机从等待队列中唤醒一个线程

notifyAll()：唤醒全部等待线程，也要在同步K块中，并且有锁对象调用

应用场景：生产者消费者问题中出现

**3.7 Condition源码分析**（<https://javadoop.com/2017/07/20/AbstractQueuedSynchronizer-2/>）

Condition：

线程之间除了同步互斥，还要考虑通信。在Java5之前我们的通信方式为：wait 和 notify。Condition的优势是支持多路等待，即可以定义多个Condition，每个condition控制线程的一条执行通路。传统方式只能是一路等待

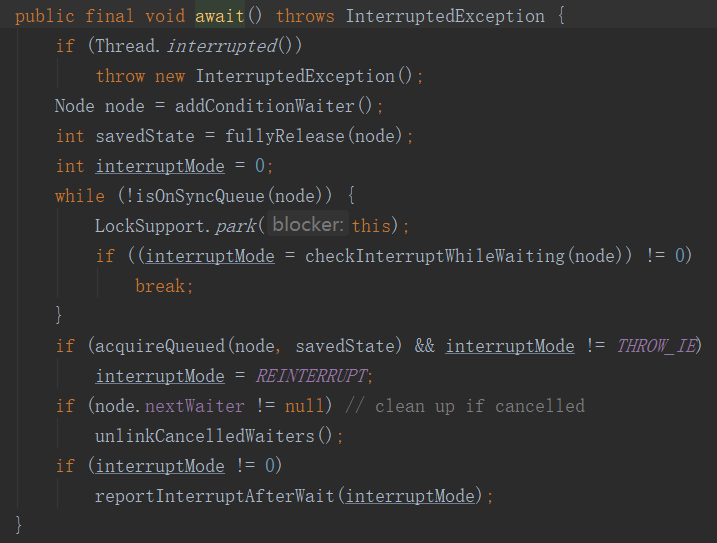
Condition提供不同于Object 监视器方法的行为和语义，如受保证的通知排序，或者在执行通知时不需要保持一个锁。

使用：

Lock lock = new ReentrantLock();  
Condition ca = lock.newCondition();

lock.lock();  
while(signal != 0){  
 try {  
 Thread.*sleep*(100);  
 ca.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}  
System.*out*.println("a");  
signal ++;  
cb.signal();  
lock.unlock();

Condition是由ReentrantLock的newCondition()创建，而接着又是同步锁Sync的newCondition()产生，是一个ConditionObject类实例（Reentrant内部类），我们一般只用到await()和signal()方法，我们重点看这两个方法：

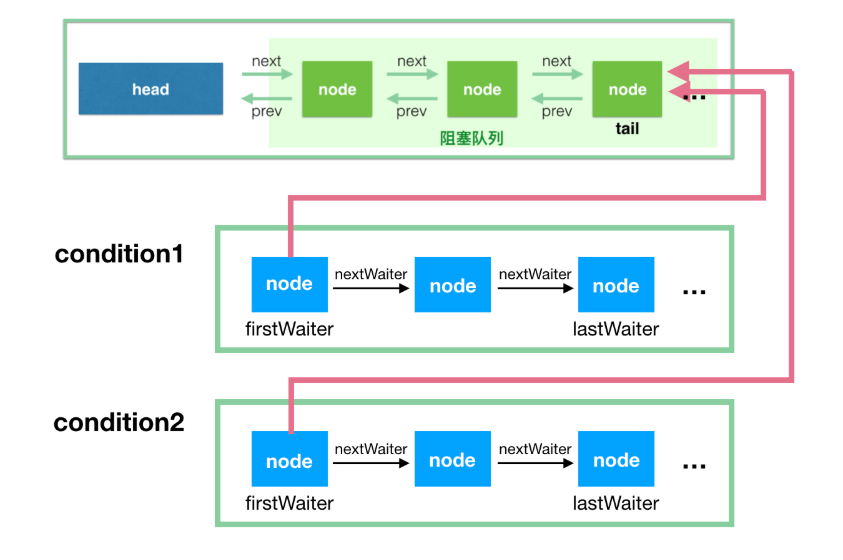


await()方法是会释放锁的，若Thread.*interrupted*()==true,即线程被阻塞中，那即表示没有拥有锁，不能调用await()方法，抛出异常；

Condition实例实质上被绑定到一个锁上。一个锁内部可以有多个Condition，即有多路等待和通知。

则可以存在多个等待队列（注意区别线程结点的同步队列，等待队列的结点需要放在同步队列中抢占CPU

时间片才能执行）。



同步队列与条件队列：

1. 条件队列和阻塞队列的节点，都是 Node 的实例，因为条件队列的节点是需要转移到阻塞队列中去的；
2. 我们知道一个 ReentrantLock 实例可以通过多次调用 newCondition() 来产生多个 Condition 实例，这里对应 condition1 和 condition2。注意，ConditionObject 只有两个属性 firstWaiter 和 lastWaiter；
3. 每个 condition 有一个关联的**条件队列**，如线程 1 调用 condition1.await() 方法即可将当前线程 1 包装成 Node 后加入到条件队列中，然后阻塞在这里，不继续往下执行，条件队列是一个单向链表；
4. 调用condition1.signal() 触发一次唤醒，此时唤醒的是队头，会将condition1 对应的**条件队列**的 firstWaiter（队头） 移到**阻塞队列的队尾**，等待获取锁，获取锁后 await 方法才能返回，继续往下执行。

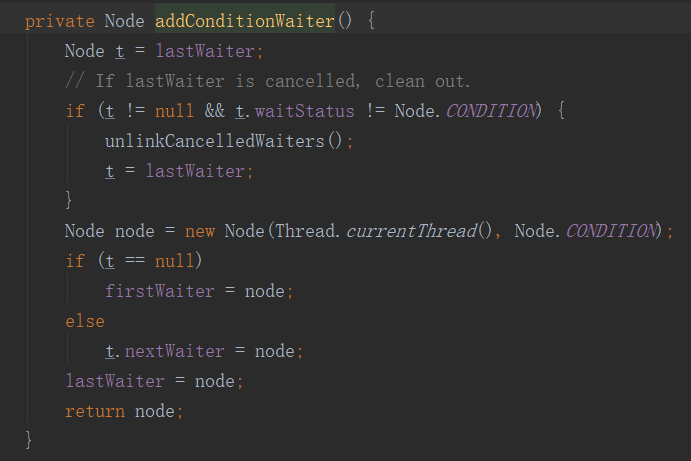
**3.8 并发工具类使用以及源码分析**

CountDownLatch

CyclicBarrier

Semaphore

Idea有案例

接着addConditionWaiter()添加到等待队列中

unlinkCancelledWaiters： 这个方法会遍历整个条件队列，然后会将已取消的所有节点清除出队列

// 释放锁，返回值是释放锁之前的 state 值

// await() 之前，当前线程是必须持有锁的，这里肯定要释放掉

int savedState = fullyRelease(node);

// 如果不在阻塞队列中，注意了，是阻塞队列

while (!isOnSyncQueue(node)) {

// 线程挂起

LockSupport.park(this);

接下来，// 被唤醒后，将进入阻塞队列，等待获取锁

即退出了while (!isOnSyncQueue(node))循环

**3.8 join()**

join():当前线程等待调用join()的线程执行完毕再往下执行

查看join()的源码：

调用Thread的join()，然后再调用join(long),传入参数0，join(long)是使用synchronized修饰的方法，则锁对象是当前对象实例；当参数为0时，调用wait()，则当前线程阻塞，转到调用join()的线程执行。当调用线程执行完毕之后（终结），会调用notifyAll().

上面为个人思考过程，有错!!

**但是我们需要知道的是，调用wait方法必须要获取锁**，所以join方法是被synchronized修饰的，synchronized修饰在方法层面相当于synchronized(this),this就是previousThread（调用线程）本身的实例。

有很多人不理解join为什么阻塞的是主线程呢? 当主线程调用previousThread.join()时，主线程会获得prvviousThread对象的锁。不理解的原因是阻塞主线程的方法是放在previousThread这个实例作用，让大家误以为应该阻塞previousThread线程。实际上主线程会持有previousThread这个对象的锁，然后调用wait方法去阻塞，而这个方法的调用者是在主线程中的。所以造成主线程阻塞。

Thread.join其实底层是通过wait/notifyall来实现线程的通信达到线程阻塞的目的；当线程执行结束以后，会触发两个事情，第一个是设置native线程对象为null、第二个是通过notifyall方法，让等待在previousThread对象锁上的wait方法被唤醒。

**既然A调用B线程的join，那为什么A会被阻塞？**

因为，A中调用B线程的join的时候，当前运行的线程是A而不是B,这个时候可以这么理解，把join看成其它方法比如test方法，A调用B的test的时候B就是一个普通对象，并不是说B线程去执行join方法，还是A线程在执行B的join方法，然后再去调用wait方法，从而A线程进入等待状态。

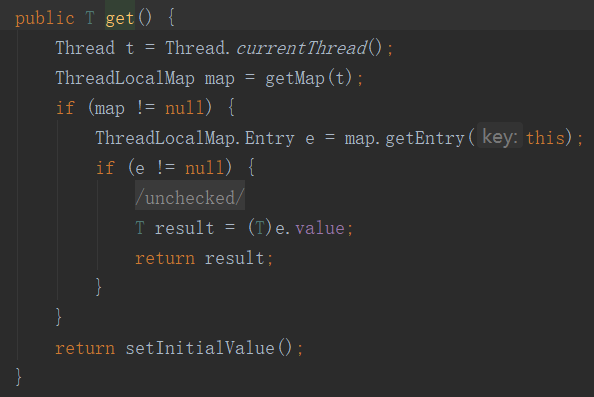
**3.9 ThreadLocal**

相当于线程的局部变量，每个线程都有，所以线程安全。

分析ThreadLocal的源码：

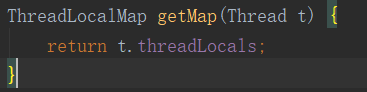
首先我们主要关注其中的get()、set(T value)、setInitialValue()

get():

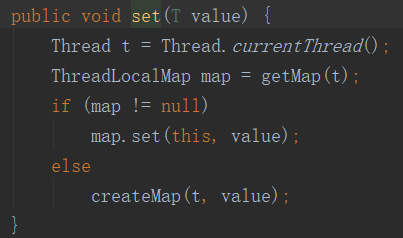


可以看出从ThreadLocalMap中获取Entry在取值，map的key为当前线程实例，value为Object;

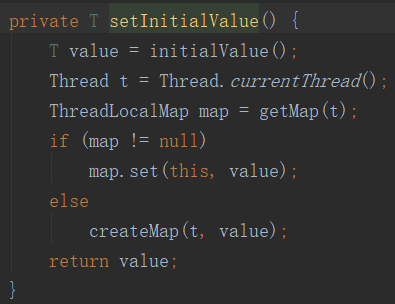
而ThreadLocalMap是Thread的一个属性，在getMap(t)中就返回Thread的ThreadLocalMap



set():



setInitialValue():



**4.0 Future源码解析**

Runnable : run()是被线程调用的，在run()中是异步执行的；

Callable : call()是由Future的run方法调用的，不是异步执行的

下面用我的语言叙述一次Future的执行过程：

//FutureTask实现RunnableFuture接口，而RunnableFuture继承了Runnable、Future接口  
FutureTask<Integer> task = new FutureTask<Integer>(new Callable\_Thread());  
//task可以作为Thread的参数  
Thread t1 = new Thread(task);  
t1.start();  
//获取返回值  
Integer result = task.get();

如上面注释所说，FutureTask实际上是Runnable的子接口，所以可以作为Thread的参数，然后线程自动调用FutureTask实现的run().

返回值其实是如何实现的呢？

public void run() {  
 if (state != *NEW* ||  
 !*UNSAFE*.compareAndSwapObject(this, *runnerOffset*,  
 null, Thread.*currentThread*()))  
 return;  
 try {  
 Callable<V> c = callable;  
 if (c != null && state == *NEW*) {  
 V result;  
 boolean ran;  
 try {  
 result = c.call();  
 ran = true;  
 } catch (Throwable ex) {  
 result = null;  
 ran = false;  
 setException(ex);  
 }  
 if (ran)  
 set(result);  
 }  
 } finally {  
 // runner must be non-null until state is settled to  
 // prevent concurrent calls to run()  
 runner = null;  
 // state must be re-read after nulling runner to prevent  
 // leaked interrupts  
 int s = state;  
 if (s >= *INTERRUPTING*)  
 handlePossibleCancellationInterrupt(s);  
 }  
}

可以看到，run()调用了callable的call(),返回结果

public FutureTask(Runnable runnable, V result) {  
 this.callable = Executors.*callable*(runnable, result);  
 this.state = *NEW*; // ensure visibility of callable  
}

public static <T> Callable<T> callable(Runnable task, T result) {  
 if (task == null)  
 throw new NullPointerException();  
 return new RunnableAdapter<T>(task, result);  
}

static final class RunnableAdapter<T> implements Callable<T> {  
 final Runnable task;  
 final T result;  
 RunnableAdapter(Runnable task, T result) {  
 this.task = task;  
 this.result = result;  
 }  
 public T call() {  
 task.run();  
 return result;  
 }  
}

FutureTask有一个构造方法，传入Runnable类型实例，并将接受返回值的参数（泛型）传入；Executors.*callable*(runnable, result)返回一个RunnableAdapter(实现了Callable接口)，这里设计得很巧妙，实际上上面的call()调用RunnableAdapter的call()，然后运行Runnable类型对象的run(),返回result（相当于RunnableAdapter整合了Runnable和返回值功能）

最后的get()是如何实现等到task的run()完成再返回值？

public V get() throws InterruptedException, ExecutionException {  
 int s = state;  
 if (s <= *COMPLETING*)  
 s = awaitDone(false, 0L);  
 return report(s);  
}

private int awaitDone(boolean timed, long nanos)  
 throws InterruptedException {  
 final long deadline = timed ? System.*nanoTime*() + nanos : 0L;  
 WaitNode q = null;  
 boolean queued = false;  
 for (;;) {  
 if (Thread.*interrupted*()) {  
 removeWaiter(q);  
 throw new InterruptedException();  
 }  
  
 int s = state;  
 if (s > *COMPLETING*) {  
 if (q != null)  
 q.thread = null;  
 return s;  
 }  
 else if (s == *COMPLETING*) // cannot time out yet  
 Thread.*yield*();  
 else if (q == null)  
 q = new WaitNode();  
 else if (!queued)  
 queued = *UNSAFE*.compareAndSwapObject(this, *waitersOffset*,  
 q.next = waiters, q);  
 else if (timed) {  
 nanos = deadline - System.*nanoTime*();  
 if (nanos <= 0L) {  
 removeWaiter(q);  
 return state;  
 }  
 LockSupport.*parkNanos*(this, nanos);  
 }  
 else  
 LockSupport.*park*(this);  
 }  
}

FutureTask使用state来表示状态，有一下几个状态：

private volatile int state;  
private static final int *NEW* = 0;  
private static final int *COMPLETING* = 1;  
private static final int *NORMAL* = 2;  
private static final int *EXCEPTIONAL* = 3;  
private static final int *CANCELLED* = 4;  
private static final int *INTERRUPTING* = 5;  
private static final int *INTERRUPTED* = 6;

*\* Possible state transitions:  
\* NEW -> COMPLETING -> NORMAL  
\* NEW -> COMPLETING -> EXCEPTIONAL  
\* NEW -> CANCELLED  
\* NEW -> INTERRUPTING -> INTERRUPTED*

由上面源码看出，当state<= COMPLETING,即未完成，进入awaitDone方法中，再次判断，当state> COMPLETING,返回状态state,调用report(state)方法;若其他情况，先生成结点WaitNode（包含线程和后继结点指针）加入链表中，然后挂起等待；

上面的run()调用call()之后会返回result，然后调用FutureTask的set(result)方法把result放到outcome属性中（Object类型，如果发生异常，outcome就接收异常对象）

protected void set(V v) {  
 if (*UNSAFE*.compareAndSwapInt(this, *stateOffset*, *NEW*, *COMPLETING*)) {  
 outcome = v;  
 *UNSAFE*.putOrderedInt(this, *stateOffset*, *NORMAL*); // final state  
 finishCompletion();  
 }  
}

private void finishCompletion() {  
 // assert state > COMPLETING;  
 for (WaitNode q; (q = waiters) != null;) {  
 if (*UNSAFE*.compareAndSwapObject(this, *waitersOffset*, q, null)) {  
 for (;;) {  
 Thread t = q.thread;  
 if (t != null) {  
 q.thread = null;  
 LockSupport.*unpark*(t);  
 }  
 WaitNode next = q.next;  
 if (next == null)  
 break;  
 q.next = null; // unlink to help gc  
 q = next;  
 }  
 break;  
 }  
 }  
  
 done();  
  
 callable = null; // to reduce footprint  
}

将值放入outcome后就需要循环唤醒链表中所有的等待结点，然后之前挂起的线程继续执行，调用report(state)返回结果

private V report(int s) throws ExecutionException {  
 Object x = outcome;  
 if (s == *NORMAL*)  
 return (V)x;  
 if (s >= *CANCELLED*)  
 throw new CancellationException();  
 throw new ExecutionException((Throwable)x);  
}

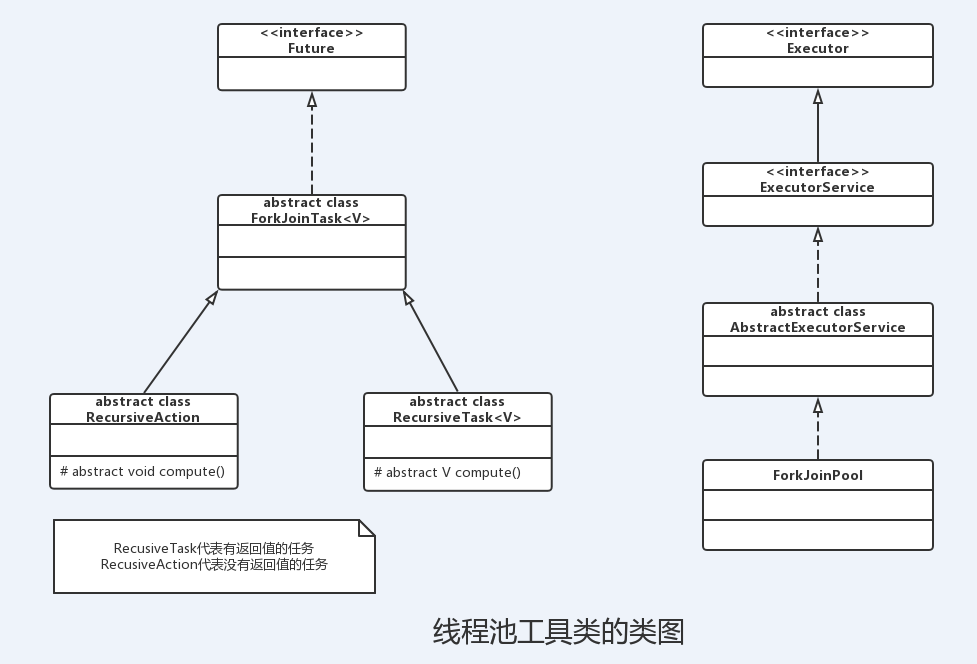
根据state决定返回结果还是抛出异常

总的来书，就是线程调用run(),run中执行call(),返回结果；若还没有结果就有线程取的话，就生成等待结点加入链表，set()完值后再唤醒链表中所有挂起的线程

4.1 ForkJoin框架详解<https://www.cnblogs.com/lixuwu/p/7979480.html>

ForkJoinPool的优势：可以充分利用多核CPU，通过发一个大任务分成多个小任务，然后通过多线程并行并发地执行任务，最后将结果合并。

ForkJoinPool的优势在于，可以充分利用多cpu，多核cpu的优势，把一个任务拆分成多个“小任务”，把多个“小任务”放到多个处理器核心上并行执行；当多个“小任务”执行完成之后，再将这些执行结果合并起来即可。这种思想值得学习。



ForkJoinTask的工作原理：

<http://blog.dyngr.com/blog/2016/09/15/java-forkjoinpool-internals/>

<https://blog.csdn.net/niyuelin1990/article/details/78658251>

4.2 同步容器和并发容器

Vector（线程安全，性能不高，在修改方法上加了synchronized）

---> ArrayList（线程不安全，方法没有加synchronized，使用Collections.*synchronizedList*(list)进行线程安全化，原理是类似代理加锁，用SynchronizedList包装List

---> [CopyOnWriteArrayList](http://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/CopyOnWriteArrayList.html#CopyOnWriteArrayList--)（）,类似读写分离吧

HashTable（线程安全）

---> HashMap（线程不安去，Collections.*synchronizedMap*(map)）

---> ConcurrentMap （不锁整个map表，实现分区域加锁）

4.3 CopyOnWriteArray原理分析

add(E d):

public boolean add(E e) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 Object[] newElements = Arrays.*copyOf*(elements, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

在操作前先加ReentrantLock,然后获取原数组，原数组长度，接着复制原数组元素到新数组，长度+1，再添加新元素到新数组，最后把原数组引用指向新数组

可以看出，这样性能是会降低的

然后set(E e)和remove()也差不多

4.3 ConcurrentLinkedQueue分析（非阻塞队列）

这个结构的安全性如何来呢？

注意：ConcurrentLinkedQueue全是采用的非阻塞算法，里面没有使用任何锁，全是基于CAS操作实现的。

在操作结点时使用了以下一些cas操作（原子操作）：

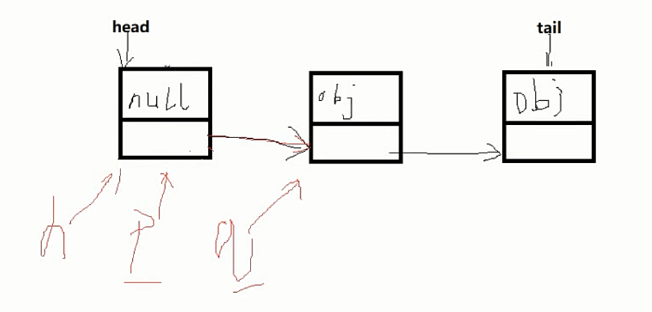
p.casNext(null, newNode)

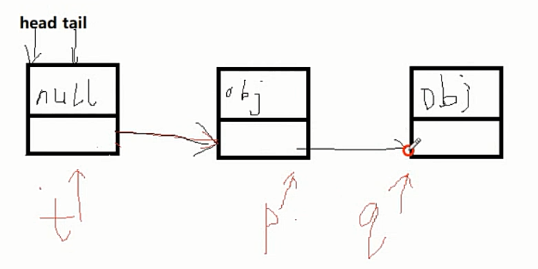
p.casItem(item, null)

对一个队列来说，插入满足FIFO特性，插入元素总是在队列最末尾的地方进行插入，而取（移除）元素总是从队列的队头。所有要想能够彻底弄懂ConcurrentLinkedQueue自然而然是从offer方法和poll方法开始。

差不多就是链表结构的增删改查

不同的就是当在队列胃添加一个结点之后，不是马上将tail引用指向最后的结点，而是下一次添加时再判断进行添加；在队头删除结点时将head的下一结点数据（element）区域置null，head引用也不马上指向被置空结点





4.4 LinkedBlockingQueue（阻塞队列）

LinkedBlockingQueue是一个线程安全的阻塞队列，它实现了BlockingQueue接口，BlockingQueue接口继承自java.util.Queue接口，并在这个接口的基础上增加了take和put方法，这两个方法正是队列操作的阻塞版本。

public void put(E e) throws InterruptedException {  
 if (e == null) throw new NullPointerException();  
 // Note: convention in all put/take/etc is to preset local var  
 // holding count negative to indicate failure unless set.  
 int c = -1;  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock putLock = this.putLock;  
 final AtomicInteger count = this.count;  
 putLock.lockInterruptibly();  
 try {  
 /\*  
 \* Note that count is used in wait guard even though it is  
 \* not protected by lock. This works because count can  
 \* only decrease at this point (all other puts are shut  
 \* out by lock), and we (or some other waiting put) are  
 \* signalled if it ever changes from capacity. Similarly  
 \* for all other uses of count in other wait guards.  
 \*/  
 while (count.get() == capacity) {  
 notFull.await();  
 }  
 enqueue(node);  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
}

public E take() throws InterruptedException {  
 E x;  
 int c = -1;  
 final AtomicInteger count = this.count;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lockInterruptibly();  
 try {  
 while (count.get() == 0) {  
 notEmpty.await();  
 }  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}

原理：通过一个可重入锁ReentrantLock和两个Condition条件对象来实现阻塞。有notFull和notEmpty两个条件对象，LinkedBlockingQueue构造方法可以传入最大结点数，当添加时（put），若队列数已经达到最大结点数，则会调用notFull.await()，只有等到调用take()从队列移除后，才会调用notFull.notify();当移除时，队列结点数==0时，调用notEmpty.await()方法，知道添加再唤醒

可以用于生产者消费者案例中

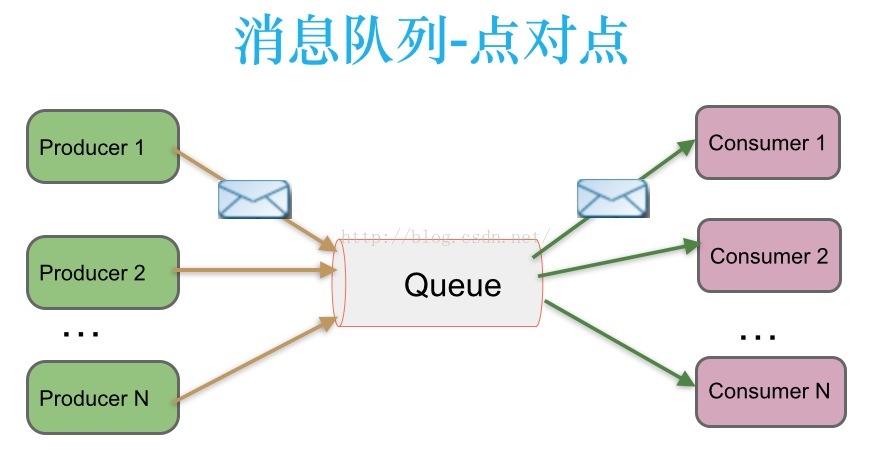
随便补充一点：尽量使用\*\*.isEmpty()代替\*\*.size()>0

4.5 消息队列 <https://blog.csdn.net/HEYUTAO007/article/details/50131089>

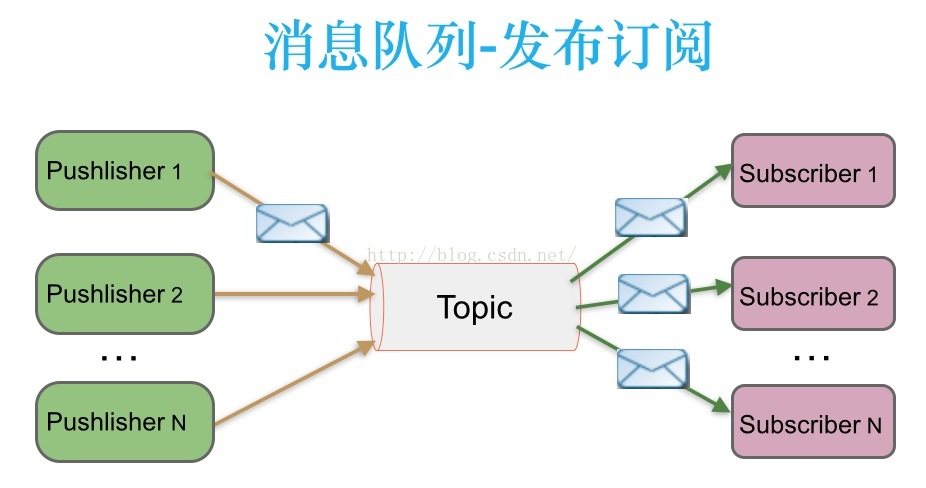
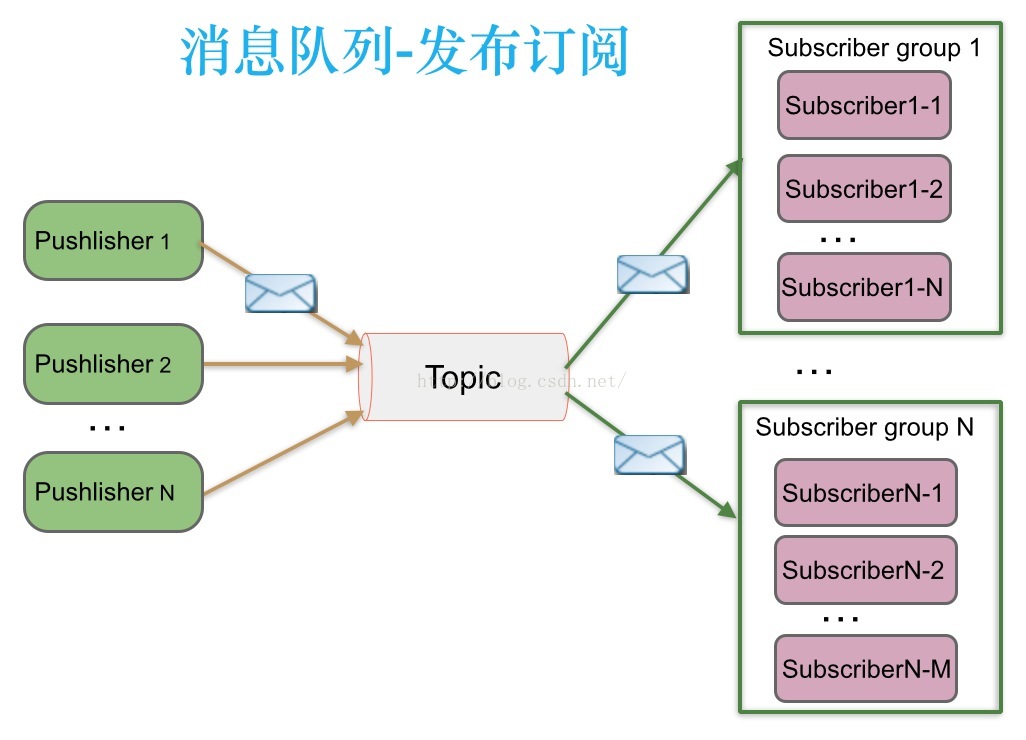
JMS（Java Message Service）规范目前支持两种消息模型：点对点（point to point， queue）和发布/订阅（publish/subscribe，topic）。

**点对点：Queue，不可重复消费**

消息生产者生产消息发送到queue中，然后消息消费者从queue中取出并且消费消息。  
消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。



**发布/订阅：Topic，可以重复消费**

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费。  
  
支持订阅组的发布订阅模式：  
发布订阅模式下，当发布者消息量很大时，显然单个订阅者的处理能力是不足的。实际上现实场景中是多个订阅者节点组成一个订阅组负载均衡消费topic消息即分组订阅，这样订阅者很容易实现消费能力线性扩展。可以看成是一个topic下有多个Queue，每个Queue是点对点的方式，Queue之间是发布订阅方式。  


**流行模型比较**

传统企业型消息队列ActiveMQ遵循了JMS规范，实现了点对点和发布订阅模型，但其他流行的消息队列RabbitMQ、Kafka并没有遵循JMS规范。

**消息队列的应用场景：**

个人认为消息队列的主要特点是异步处理，主要目的是减少请求响应时间和解耦。所以主要的使用场景就是将比较耗时而且不需要即时（同步）返回结果的操作作为消息放入消息队列。

**使用场景的话，举个例子：**

假设用户在你的软件中注册，服务端收到用户的注册请求后，它会做这些操作：

校验用户名等信息，如果没问题会在数据库中添加一个用户记录

如果是用邮箱注册会给你发送一封注册成功的邮件，手机注册则会发送一条短信

分析用户的个人信息，以便将来向他推荐一些志同道合的人，或向那些人推荐他

发送给用户一个包含操作指南的系统通知

等等……

但是对于用户来说，注册功能实际只需要第一步，只要服务端将他的账户信息存到数据库中他便可以登录上去做他想做的事情了。至于其他的事情，非要在这一次请求中全部完成么？值得用户浪费时间等你处理这些对他来说无关紧要的事情么？所以实际当第一步做完后，服务端就可以把其他的操作放入对应的消息队列中然后马上返回用户结果，由消息队列异步的进行这些操作。

或者还有一种情况，同时有大量用户注册你的软件，再高并发情况下注册请求开始出现一些问题，例如邮件接口承受不住，或是分析信息时的大量计算使cpu满载，这将会出现虽然用户数据记录很快的添加到数据库中了，但是却卡在发邮件或分析信息时的情况，导致请求的响应时间大幅增长，甚至出现超时，这就有点不划算了。面对这种情况一般也是将这些操作放入消息队列（生产者消费者模型），消息队列慢慢的进行处理，同时可以很快的完成注册请求，不会影响用户使用其他功能。

4.6 ConcurrentHashMap结构分析 （用分离锁实现多个线程的并发写操作）

<https://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/index.html>

**重排序**

内存模型描述了程序的可能行为。具体的编译器实现可以产生任意它喜欢的代码 -- 只要所有执行这些代码产生的结果，能够和内存模型预测的结果保持一致。这为编译器实现者提供了很大的自由，包括操作的重排序。

编译器生成指令的次序，可以不同于源代码所暗示的“显然”版本。重排序后的指令，对于优化执行以及成熟的全局寄存器分配算法的使用，都是大有脾益的，它使得程序在计算性能上有了很大的提升。

重排序类型包括：

* 编译器生成指令的次序，可以不同于源代码所暗示的“显然”版本。
* 处理器可以乱序或者并行的执行指令。
* 缓存会改变写入提交到主内存的变量的次序。

**内存可见性**

由于现代可共享内存的多处理器架构可能导致一个线程无法马上（甚至永远）看到另一个线程操作产生的结果。所以 Java 内存模型规定了 JVM 的一种最小保证：什么时候写入一个变量对其他线程可见。

在现代可共享内存的多处理器体系结构中每个处理器都有自己的缓存，并周期性的与主内存协调一致。假设线程 A 写入一个变量值 V，随后另一个线程 B 读取变量 V 的值，在下列情况下，线程 B 读取的值可能不是线程 A 写入的最新值：

* 执行线程 A 的处理器把变量 V 缓存到寄存器中。
* 执行线程 A 的处理器把变量 V 缓存到自己的缓存中，但还没有同步刷新到主内存中去。
* 执行线程 B 的处理器的缓存中有变量 V 的旧值。

**Happens-before 关系**

happens-before 关系保证：如果线程 A 与线程 B 满足 happens-before 关系，则线程 A 执行动作的结果对于线程 B 是可见的。如果两个操作未按 happens-before 排序，JVM 将可以对他们任意重排序。

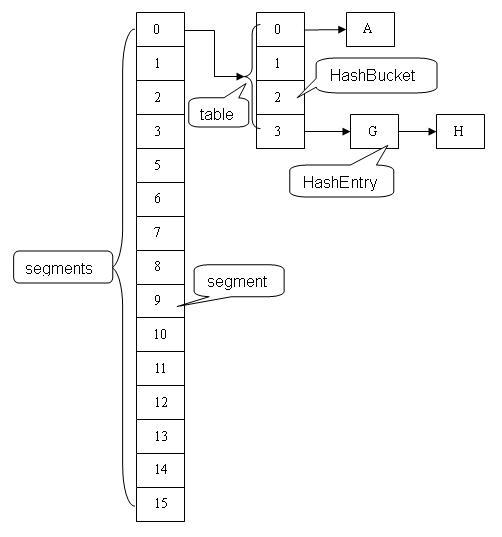
下面介绍几个与理解 ConcurrentHashMap 有关的 happens-before 关系法则：

1. 程序次序法则：如果在程序中，所有动作 A 出现在动作 B 之前，则线程中的每动作 A 都 happens-before 于该线程中的每一个动作 B。
2. 监视器锁法则：对一个监视器的解锁 happens-before 于每个后续对同一监视器的加锁。
3. Volatile 变量法则：对 Volatile 域的写入操作 happens-before 于每个后续对同一 Volatile 的读操作。
4. 传递性：如果 A happens-before 于 B，且 B happens-before C，则 A happens-before C。

**ConcurrentHashMap的结构：**

ConcurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。HashEntry 用来封装映射表的键 / 值对；Segment 用来充当锁的角色，每个 Segment 对象守护整个散列映射表的若干个桶。每个桶是由若干个 HashEntry 对象链接起来的链表。一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组。

ConcurrentHashMap 在默认并发级别会创建包含 16 个 Segment 对象的数组。每个 Segment 的成员对象 table 包含若干个散列表的桶。每个桶是由 HashEntry 链接起来的一个链表。如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。



插入三个节点后桶的结构示意图：

图 1. 插入三个节点后桶的结构示意图：

**用 HashEntery 对象的不变性来降低读操作对加锁的需求**

在代码清单“HashEntry 类的定义”中我们可以看到，HashEntry 中的 key，hash，next 都声明为 final 型。这意味着，不能把节点添加到链接的中间和尾部，也不能在链接的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变。这个特性可以大大降低处理链表时的复杂性。

同时，HashEntry 类的 value 域被声明为 Volatile 型，Java 的内存模型可以保证：某个写线程对 value 域的写入马上可以被后续的某个读线程“看”到。在 ConcurrentHashMap 中，不允许用 unll 作为键和值，当读线程读到某个 HashEntry 的 value 域的值为 null 时，便知道产生了冲突——发生了重排序现象，需要加锁后重新读入这个 value 值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。

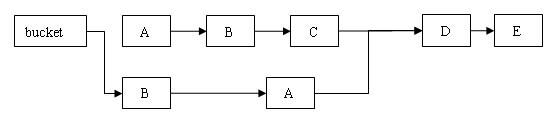
“在 Segment 中执行具体的 put 操作”中，我们可以看出：put 操作如果需要插入一个新节点到链表中时 , 会在链表头部插入这个新节点。此时，链表中的原有节点的链接并没有被修改。也就是说：插入新健 / 值对到链表中的操作不会影响读线程正常遍历这个链表。

和 get 操作一样，首先根据散列码找到具体的链表；然后遍历这个链表找到要删除的节点；最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。下面通过图例来说明 remove 操作。假设写线程执行 remove 操作，要删除链表的 C 节点，另一个读线程同时正在遍历这个链表。

**执行删除之前的原链表：**

图 4. 执行删除之前的原链表：

**执行删除之后的新链表**



**总结**

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和用同步包装器包装的 HashMap（Collections.synchronizedMap(new HashMap())），ConcurrentHashMap 拥有更高的并发性。在 HashTable 和由同步包装器包装的 HashMap 中，使用一个全局的锁来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成*串行化*的了。

在使用锁来协调多线程间并发访问的模式下，减小对锁的竞争可以有效提高并发性。有两种方式可以减小对锁的竞争：

1. 减小请求 同一个锁的 频率。
2. 减少持有锁的 时间。

**ConcurrentHashMap 的高并发性主要来自于三个方面：**

1. 用分离锁实现多个线程间的更深层次的共享访问。
2. 用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。
3. 通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。