线程池生命周期：

我们需要了解的就是线程池的生命周期，回顾下，线程池的状态有：

1、RUNNING

2、SHUNDOWN

3、STOP

4、TIDYING

5、TERMINATED

这几个状态的转化关系为：

1、调用shundown()方法线程池的状态由RUNNING——>SHUTDOWN

2、调用shutdowNow()方法线程池的状态由RUNNING——>STOP

3、当任务队列和线程池均为空的时候 线程池的状态由STOP/SHUTDOWN——–>TIDYING

4、当terminated()方法被调用完成之后，线程池的状态由TIDYING———->TERMINATED状态

线程池有运行、关闭、停止、结束四种状态，结束后就会释放所有资源

平缓关闭线程池使用shutdown()

立即关闭线程池使用shutdownNow()，同时得到未执行的任务列表

检测线程池是否正处于关闭中，使用isShutdown()

检测线程池是否已经关闭使用isTerminated()

定时或者永久等待线程池关闭结束使用awaitTermination()操作

**线程生命周期**

　　线程的5种状态：

　　　　新建（New） ，就绪（Runnable），运行（Running），阻塞（Blocked），死亡（Dead）



Runtime.getRuntime().addShutdownHook(shutdownHook);

   这个方法的含义说明：

       这个方法的意思就是在jvm中增加一个关闭的钩子，当jvm关闭的时候，会执行系统中已经设置的所有通过方法addShutdownHook添加的钩子，当系统执行完这些钩子后，jvm才会关闭。所以这些钩子可以在jvm关闭的时候进行内存清理、对象销毁等操作。

**java.util.concurrent包完全建立在CAS之上的，没有CAS就不会有此包。可见CAS的重要性。**

**CAS**

CAS:Compare and Swap, 翻译成比较并交换。

java.util.concurrent包中借助CAS实现了区别于synchronouse同步锁的一种乐观锁。

**Compare And Swap**

       CAS 指的是现代 CPU 广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。这个指令会对内存中的共享数据做原子的读写操作。简单介绍一下这个指令的操作过程：首先，CPU 会将内存中将要被更改的数据与期望的值做比较。然后，当这两个值相等时，CPU 才会将内存中的数值替换为新的值。否则便不做操作。最后，CPU 会将旧的数值返回。这一系列的操作是原子的。它们虽然看似复杂，但却是 Java 5 并发机制优于原有锁机制的根本。简单来说，CAS 的含义是“我认为原有的值应该是什么，如果是，则将原有的值更新为新值，否则不做修改，并告诉我原来的值是多少”。（这段描述引自《Java并发编程实践》）

**简单的来说，CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则返回V。这是一种乐观锁的思路，它相信在它修改之前，没有其它线程去修改它**；而Synchronized是一种悲观锁，它认为在它修改之前，一定会有其它线程去修改它，悲观锁效率很低。下面来看一下AtomicInteger是如何利用CAS实现原子性操作的。

**非阻塞算法 （nonblocking algorithms）**

一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。

现代的CPU提供了特殊的指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。

拿出AtomicInteger来研究在没有锁的情况下是如何做到数据正确性的。

private volatile int value;

首先毫无以为，在没有锁的机制下可能需要借助volatile原语，保证线程间的数据是可见的（共享的）。

这样才获取变量的值的时候才能直接读取。

public final int get() {

        return value;

    }

然后来看看++i是怎么做到的。

public final int incrementAndGet() {

    for (;;) {

        int current = get();

        int next = current + 1;

        if (compareAndSet(current, next))

            return next;

    }

}

在这里采用了CAS操作，每次从内存中读取数据然后将此数据和+1后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。

而compareAndSet利用JNI来完成CPU指令的操作。

public final boolean compareAndSet(int expect, int update) {

    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

    }

整体的过程就是这样子的，利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。

其中

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

类似：

if (this == expect) {

  this = update

 return true;

} else {

return false;

}

Jdk1.5之后加入了java.util.concurrent包，这个包中主要介绍java中线程以及线程池的使用。

**线程池的作用**

减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

**线程池的静态工厂**

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

newFixedThreadPool（LinkedBlockingQueue）

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

newCachedThreadPool(SynchronousQueue)

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**ThreadPoolExecutor详解**

ThreadPoolExecutor是ExecutorService的默认实现。

public class ThreadPoolExecutor extends AbstractExecutorService {

.....

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,

long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,

long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,

long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,

RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,

long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory

threadFactory,RejectedExecutionHandler handler);

...

}

参数说明:

corePoolSize : (线程池的基本大小) 当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。

**runnableTaskQueue（任务队列）**：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列。

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，初始化要指定大小,此队列按 FIFO（先进先出）原则对元素进行排序。

LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE

SynchronousQueue：**一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态**，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

maximumPoolSize - 池中允许的最大线程数。线程池允许创建的最大线程数。**如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。**值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

unit - keepAliveTime 参数的时间单位。（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

workQueue- 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

threadFactory - 执行程序创建新线程时使用的工厂。用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

handler - RejectedExecutionHandler（饱和策略） 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序： **当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。**以下是JDK1.5提供的四种策略

AbortPolicy：直接抛出异常。

CallerRunsPolicy：只用调用者所在线程来运行任务。

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

DiscardPolicy：不处理，丢弃掉。

当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。

ThreadPoolExecutor是Executors类的底层实现。

在JDK帮助文档中，有如此一段话：

“强烈建议程序员使用较为方便的 Executors 工厂方法 Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池） Executors.newSingleThreadExecutor()（单个后台线程）

它们均为大多数使用场景预定义了设置。”

ExecutorService newFixedThreadPool (int nThreads):固定大小线程池。

1. public static ExecutorService newFixedThreadPool(int nThreads) {

2. return new ThreadPoolExecutor(nThreads, nThreads,

3. 0L, TimeUnit.MILLISECONDS,

4. new LinkedBlockingQueue<Runnable>());

5. }

可以看到，corePoolSize和maximumPoolSize的大小是一样的

（实际上，后面会介绍，**如果使用无界queue的话maximumPoolSize参数是没有意义的**），keepAliveTime和unit的设值表名什么？-就是该实现不想keep alive！最后的BlockingQueue选择了LinkedBlockingQueue，该queue有一个特点，他是无界的。

ExecutorService newSingleThreadExecutor()：单线程

1. public static ExecutorService newSingleThreadExecutor() {

2. return new FinalizableDelegatedExecutorService

3. (new ThreadPoolExecutor(1, 1,

4. 0L, TimeUnit.MILLISECONDS,

5. new LinkedBlockingQueue<Runnable>()));

6. }

ExecutorService newCachedThreadPool()：无界线程池，可以进行自动线程回收

1. public static ExecutorService newCachedThreadPool() {

2. return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

3. 60L, TimeUnit.SECONDS,

4. new SynchronousQueue<Runnable>());

}

这个实现就有意思了。首先是无界的线程池，所以我们可以发现maximumPoolSize为big big。其次BlockingQueue的选择上使用SynchronousQueue。可能对于该BlockingQueue有些陌生，简单说：该QUEUE中，**每个插入操作必须等待另一个线程的对应移除操作。** queue上的三种类型

**1. SynchronousQueue: newCacheThreadPool**

SynchronousQueue是无界的，是一种无缓冲的等待队列，但是由于该Queue本身的特性，在某次添加元素后必须等待其他线程取走后才能继续添加；可以认为SynchronousQueue**是一个缓存值为1的阻塞队列**，但是 isEmpty()方法永远返回是true，remainingCapacity() 方法永远返回是0，remove()和removeAll() 方法永远返回是false，iterator()方法永远返回空，peek()方法永远返回null。

声明一个SynchronousQueue有两种不同的方式，它们之间有着不太一样的行为。公平模式和非公平模式的区别:如果采用公平模式：SynchronousQueue会采用公平锁，并配合一个FIFO队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；但如果是非公平模式（SynchronousQueue默认）：SynchronousQueue采用非公平锁，同时配合一个LIFO队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有某些生产者或者是消费者的数据永远都得不到处理。

**2. LinkedBlockingQueue:newFixedThreadPool**

LinkedBlockingQueue是无界的，是一个无界缓存的等待队列。

**基于链表的阻塞队列，内部维持着一个数据缓冲队列（该队列由链表构成）**。当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。

**LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。**

**3. ArrayBlockingQueue**

ArrayBlockingQueue是有界的，是一个有界缓存的等待队列。

基于数组的阻塞队列，同LinkedBlockingQueue类似，内部维持着一个**定长数据缓冲队列**（该队列由数组构成）。**ArrayBlockingQueue内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置**。

ArrayBlockingQueue在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于LinkedBlockingQueue；按照实现原理来分析，ArrayBlockingQueue完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。Doug Lea之所以没这样去做，**也许是因为ArrayBlockingQueue的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。** ArrayBlockingQueue和LinkedBlockingQueue间还有一个明显的不同之处在于，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的Node对象。这在长时间内需要高效并发地处理大批量数据的系统中，其对于GC的影响还是存在一定的区别。

**ArrayBlockingQueue和LinkedBlockingQueue是两个最普通、最常用的阻塞队列，一般情况下，处理多线程间的生产者消费者问题，使用这两个类足以。**

**线程池任务提交**

大概可以有两种提交方式:

1. 可以使用execute提交的任务:

但是execute方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知execute方法输入的任务是一个Runnable类的实例。

threadsPool.execute(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

});

2. 也可以使用submit 方法来提交任务:

它会返回一个future,那么我们可以通过这个future来判断任务是否执行成功，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

Future<Object> future = executor.submit(harReturnValuetask);

try {

Object s = future.get();

} catch (InterruptedException e) {

// 处理中断异常

} catch (ExecutionException e) {

// 处理无法执行任务异常

} finally {

// 关闭线程池

executor.shutdown();

}



**线程池执行流程**

从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。

其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。

最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

源码分析:

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

//如果线程数小于基本线程数，则创建线程并执行当前任务

if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {

//如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。

if (runState == RUNNING && workQueue.offer(command)) {

if (runState != RUNNING || poolSize == 0)

ensureQueuedTaskHandled(command);

}

//如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，

则创建一个线程执行任务。

else if (!addIfUnderMaximumPoolSize(command))

//抛出RejectedExecutionException异常

reject(command); // is shutdown or saturated

}

}

工作线程: 线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从Worker的run方法里看到这点：

public void run() {

try {

Runnable task = firstTask;

firstTask = null;

while (task != null || (task = getTask()) != null) {

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

**合理的配置线程池**

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

任务的性质：CPU密集型任务，IO密集型任务和混合型任务。

任务的优先级：高，中和低。

任务的执行时间：长，中和短。

任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理:

CPU密集型任务配置尽可能小的线程，如配置Ncpu+1个线程的线程池。

IO密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

数据库连接池最好也大,充分利用cpu依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

调节线程池配置的方式:

通过配置 corePoolSize和maximumPoolSizes 这对参数还有就是BlockingQueue的选择,选择策略:

使用 直接提交-等待依赖 的策略，也即SynchronousQueue。首先SynchronousQueue是无界的，也就是说他存数任务的能力是没有限制的，但是由于该Queue本身的特性，在某次添加元素后必须等待其他线程取走后才能继续添加。在这里不是核心线程便是新创建的线程，但是我们试想一样下，下面的场景。我们使用一下参数构造ThreadPoolExecutor：

new ThreadPoolExecutor(

2, 3, 30, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(),

new RecorderThreadFactory("CookieRecorderPool"),

new ThreadPoolExecutor.CallerRunsPolicy());

当核心线程已经有2个正在运行

此时继续来了一个任务（A），根据前面介绍的“如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。”,所以A被添加到queue中。

又来了一个任务（B），且核心2个线程还没有忙完，OK，接下来首先尝试1中描述，但是由于使用的SynchronousQueue，所以一定无法加入进去。

此时便满足了上面提到的“如果无法将请求加入队列，则创建新的线程，除非创建此线程超出maximumPoolSize，在这种情况下，任务将被拒绝。”，所以必然会新建一个线程来运行这个任务。 暂时还可以，但是如果这三个任务都还没完成，连续来了两个任务，第一个添加入queue中，后一个呢？queue中无法插入，而线程数达到了maximumPoolSize，所以只好执行异常策略了。

**所以在使用SynchronousQueue通常要求maximumPoolSize是无界的，这样就可以避免上述情况发生**（如果希望限制就直接使用有界队列）。对于使用SynchronousQueue的作用jdk中写的很清楚：此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。

什么意思？如果你的任务A1，A2有内部关联，A1需要先运行，那么先提交A1，再提交A2，当使用SynchronousQueue我们可以保证，A1必定先被执行，在A1么有被执行前，A2不可能添加入queue中.

总结: 使用SynchronousQueue无界并且要等待上一个线程执行结果才运行,会收到maximumPoolSize影响,所以如果要避免限制,需要设置maximumPoolSize的值很大,避免等待.

使用无界队列策略，即LinkedBlockingQueue,

不像SynchronousQueue那样有其自身的特点，对于无界队列来说，总是可以加入的（资源耗尽，当然另当别论）。换句说，永远也不会触发产生新的线程！corePoolSize大小的线程数会一直运行，忙完当前的，就从队列中拿任务开始运行。

**所以要防止任务疯长**，比如任务运行的实行比较长，而添加任务的速度远远超过处理任务的时间，而且还不断增加，如果任务内存大一些，不一会儿就爆了，呵呵。

有界队列，使用ArrayBlockingQueue。这个是最为复杂的使用，所以JDK不推荐使用也有些道理。与上面的相比，最大的特点便是可以防止资源耗尽的情况发生。举例来说，请看如下构造方法：

new ThreadPoolExecutor(

2, 4, 30, TimeUnit.SECONDS,

new ArrayBlockingQueue<Runnable>(2),

new RecorderThreadFactory("CookieRecorderPool"),

new ThreadPoolExecutor.CallerRunsPolicy());

假设，所有的任务都永远无法执行完。

对于首先来的A,B来说直接运行，接下来，如果来了C,D，他们会被放到queu中，如果接下来再来E,F，则增加线程运行E，F。但是如果再来任务，队列无法再接受了，线程数也到达最大的限制了，所以就会使用拒绝策略来处理。

总结：

ThreadPoolExecutor的使用还是很有技巧的。

使用无界queue可能会耗尽系统资源。

使用有界queue可能不能很好的满足性能，需要调节线程数和queue大小

线程数自然也有开销，所以需要根据不同应用进行调节。

通常来说对于静态任务可以归为：

数量大，但是执行时间很短

数量小，但是执行时间较长

数量又大执行时间又长

除了以上特点外，任务间还有些内在关系

**线程池关闭**

我们可以通过调用线程池的**shutdown或shutdownNow方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。**但是它们存在一定的区别，**shutdownNow首先将线程池的状态设置成STOP**，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而**shutdown只是将线程池的状态设置成SHUTDOWN状态**，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，**isShutdown方法就会返回true**。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用**isTerminaed方法会返回true。**至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow。

**3. 用户线程和守护线程有什么区别？**

当**我们在Java程序中创建一个线程，它就被称为用户线程**。**一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出**。一个守护线程创建的子线程依然是守护线程。

1. 闭锁是一种同步工具类，可以延迟线程的进度直到其到达终止状态。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭的，并且没有任何线程都能通过，当到达结束状态时，这扇门就会打开并允许所有的线程通过。当闭锁到达结束状态后，将不会在改变其状态，因此这扇门将永远保持打开状态。**闭锁可以用来确保某些活动直到其他活动都完成后才继续执行。**具体实现有CountDownLatch。
2. FutureTask也可以用作闭锁。它表示的计算是通过Callable来实现的，相当于一种可生成结果的Runnable，并且其Future.get方法是一个阻塞方法。FutureTask在Executor框架中表示异步任务，此外还可以用来表示一些时间较长的计算，这些计算可以在使用计算结果之前启动。
3. 信号量（Semaphore）：用来控制同时访问某个特定资源的操作数量，或者同时执行某个特定操作的数量。（Counting Semaphore）计数信号量可以用来实现某种资源池，或者对容器施加边界。Semaphore中管理着一组虚拟的许可（permit），许可的初始数量可通过构造函数来指定，在执行操作时可以首先获得许可（只有还剩有许可），使用完之后释放许可。如果没许可，那么acquire将阻塞直到有许可，release方法将返回一个许可给信号量。其实，这种实现并不包含真正的许可对象，许可也没有和线程相关联。
4. 栅栏（Barrier）：闭锁是一次性对象，不能被重置，而栅栏可被重置。它类似于闭锁，但是闭锁用于等待事件，栅栏用于等待线程。它能阻塞一组线程直到某个事件发生，所有线程必须同时到达栅栏位置，才能继续执行。具体实现有CyclicBarrier和Exchanger（两方栅栏）

**CyclicBarrier和CountDownLatch区别：**

1. 构造CountDownLatch的参数的数值是调用countDown的数量，而CyclicBarrier的数量是await的数量。
2. CyclicBarrier和CountDownLatch都是用于多个线程间的协调的。二者的一个很大的差别是，CountDownLatch是在多个线程进行了latch.countDown后才会触发事件，唤醒await在latch上的线程，而执行countDown的线程，执行完countDown后会继续自己线程的工作；CyclicBarrier是一个栅栏，用于同步所有调用了await方法的线程，并且等所有线程都到了await方法时，这些线程才一起返回继续各自的工作（因为使用CyclicBarrier的线程都会阻塞在await方法上，所以在线程池中使用它要特别小心，如果线程池的线程过少，那么就会发生死锁了）。
3. CountDownLatch不能循环使用，CyclicBarrier可以循环使用。

**14) 一个线程运行时发生异常会怎样？**

这是我在一次面试中遇到的一个很刁钻的Java面试题, 简单的说，如果异常没有被捕获该线程将会停止执行。**Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口**。当一个未捕获异常将造成线程中断的时候，JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理。

**17) 为什么wait, notify 和 notifyAll这些方法不在thread类里面？**

这是个设计相关的问题，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在Object类里是有意义的，还有不把它放在Thread类里的原因。**一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。**简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

**19) 什么是FutureTask？**

在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。**一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。**

1. **Java中interrupted 和 isInterruptedd方法的区别？**

**interrupt**

**interrupt方法用于中断线程。调用该方法的线程的状态为将被置为"中断"状态。**

**注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出interruptedException的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。**

**interrupted() 和 isInterrupted()**的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变。

**interrupt（）**是用来设置中断状态的。返回true说明中断状态被设置了而不是被清除了。我们调用sleep、wait等此类可中断（throw InterruptedException）方法时，一旦方法抛出InterruptedException，当前调用该方法的线程的中断状态就会被jvm自动清除了，就是说我们调用该线程的**isInterrupted** 方法时是返回false。如果你想保持中断状态，可以再次调用interrupt方法设置中断状态。这样做的原因是，java的中断并不是真正的中断线程，而只设置标志位（中断位）来通知用户。如果你捕获到中断异常，说明当前线程已经被中断，不需要继续保持中断位。

interrupted是静态方法，返回的是当前线程的中断状态。例如，如果当前线程被中断（没有抛出中断异常，否则中断状态就会被清除），你调用**interrupted**方法，第一次会返回true。然后，当前线程的中断状态被方法内部清除了。第二次调用时就会返回false。如果你刚开始一直调用isInterrupted，则会一直返回true，除非中间线程的中断状态被其他操作清除了。

**synchronized：**

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好，不管用没用过5.0多线程包的程序员都能理解。

**ReentrantLock:**

ReentrantLock提供了多样化的同步，比如有时间限制的同步，可以被Interrupt的同步（synchronized的同步是不能Interrupt的）等。在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

**Atomic:**

和上面的类似，不激烈情况下，性能比synchronized略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic的性能会优于ReentrantLock一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个Atomic的变量，多于一个同步无效。因为他不能在多个Atomic之间同步。

所以，我们写同步的时候，优先考虑synchronized，如果有特殊需要，再进一步优化。ReentrantLock和Atomic如果用的不好，不仅不能提高性能，还可能带来灾难。

synchronized原语和ReentrantLock在一般情况下没有什么区别，**但是在非常复杂的同步应用中，请考虑使用ReentrantLock**，特别是遇到下面2种需求的时候。

1.某个线程在等待一个锁的控制权的这段时间需要中断

2.需要分开处理一些wait-notify，ReentrantLock里面的Condition应用，能够控制notify哪个线程

3.具有公平锁功能，每个到来的线程都将排队等候

下面细细道来……

先说第一种情况，ReentrantLock的lock机制有2种，忽略中断锁和响应中断锁，这给我们带来了很大的灵活性。比如：如果A、B2个线程去竞争锁，A线程得到了锁，B线程等待，但是A线程这个时候实在有太多事情要处理，就是一直不返回，B线程可能就会等不及了，想中断自己，不再等待这个锁了，转而处理其他事情。这个时候ReentrantLock就提供了2种机制，第一，B线程中断自己（或者别的线程中断它），但是ReentrantLock不去响应，继续让B线程等待，你再怎么中断，我全当耳边风（synchronized原语就是如此）；第二，B线程中断自己（或者别的线程中断它），ReentrantLock处理了这个中断，并且不再等待这个锁的到来，完全放弃。

**22) 为什么你应该在循环中检查等待条件?**

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，**在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因**，你可以在Eclipse中创建模板调用wait和notify试一试。

**23) Java中的同步集合与并发集合有什么区别？**

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的**可扩展性更高**。在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用**锁分离和内部分区**等现代技术提高了可扩展性。

**24） Java中堆和栈有什么不同？**

为什么把这个问题归类在多线程和并发面试题里？因为栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

**25） 什么是线程池？ 为什么要使用它？**

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

**26） 如何写代码来解决生产者消费者问题？**

在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道怎么进行线程间通信来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用Semaphore 或者 BlockingQueue来实现生产者消费者模型

1. **如何避免死锁？**

Java多线程中的死锁  
死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：

* 互斥条件：一个资源每次只能被一个进程使用。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
* 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

**避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。**

**28) Java中活锁和死锁有什么区别？**

这是上题的扩展，活锁和死锁类似，**不同之处在于处于活锁的线程或进程的状态是不断改变的，活锁可以认为是一种特殊的饥饿。**一个现实的活锁例子是两个人在狭小的走廊碰到，两个人都试着避让对方好让彼此通过，但是因为避让的方向都一样导致最后谁都不能通过走廊。简单的说就是，**活锁和死锁的主要区别是前者进程的状态可以改变但是却不能继续执行。**

**29） 怎么检测一个线程是否拥有锁？**

在java.lang.Thread中有一个方法叫**holdsLock()**，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

**30) 你如何在Java中获取线程堆栈？**

对于不同的操作系统，有多种方法来获得Java进程的线程堆栈。当你获取线程堆栈时，JVM会把所有线程的状态存到日志文件或者输出到控制台。在Windows你可以使用**Ctrl + Break组合键来获取线程堆栈**，**Linux下用kill -3命令**。你也可以**用jstack这个工具来获取**，**它对线程id进行操作**，**你可以用jps这个工具找到id。**

如何输出线程堆栈？

Java虚拟机提供了线程转储（thread dump）的后门，通过这个后门可以把线程堆栈打印出来。通常我们将堆栈信息重定向到一个文件中，便于我们分析，由于信息量太大，很可能超出控制台缓冲区的最大行数限制造成信息丢失。这里介绍一个jdk自带的打印线程堆栈的工具，jstack用于打印出给定的Java进程ID或core file或远程调试服务的Java堆栈信息。

示例：$jstack –l 23561 >> xxx.dump

命令 : $jstack [option] pid >> 文件

>>表示输出到文件尾部，实际运行中，往往一次dump的信息，还不足以确认问题，建议产生三次dump信息，如果每次dump都指向同一个问题，我们才确定问题的典型性。

9）Java 中怎么获取一份线程 dump 文件？

在 Linux 下，你可以通过命令 **kill -3 PID** （Java 进程的进程 ID）来获取 Java 应用的 dump 文件。在 Windows 下，你可以按下 **Ctrl + Break** 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。如果你使用Tomcat。

**32） Java中synchronized 和 ReentrantLock 有什么不同？**

Java在过去很长一段时间只能通过synchronized关键字来实现互斥，它有一些缺点。比如你不能扩展锁之外的方法或者块边界，尝试获取锁时不能中途取消等。Java 5 通过Lock接口提供了更复杂的控制来解决这些问题。 ReentrantLock 类实现了 Lock，**它拥有与 synchronized 相同的并发性和内存语义且它还具有可扩展性。**

**ReentrantLock**

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

1.**等待可中断**，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。

2.**公平锁**，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

3.**锁绑定多个条件**，一个ReentrantLock对象可以同时绑定多个对象。

**总结**

**ReentrantLock提供了内置锁类似的功能和内存语义。**

此外，ReetrantLock还提供了其它功能，包括定时的锁等待、可中断的锁等待、公平性、以及实现非块结构的加锁、Condition，对线程的等待和唤醒等操作更加灵活，一个ReentrantLock可以有多个Condition实例，所以更有扩展性，不过ReetrantLock需要显式的获取锁，并在finally中释放锁，否则后果很严重。

ReentrantLock在性能上似乎优于Synchronized，其中在jdk1.6中略有胜出，在1.5中是远远胜出。那么为什么不放弃内置锁，并在新代码中都使用ReetrantLock？

在java1.5中， **内置锁与ReentrantLock相比有例外一个优点：在线程转储中能给出在哪些调用帧中获得了哪些锁，并能够检测和识别发生死锁的线程。ReentrantLock的非块状特性意味着，获取锁的操作不能与特定的栈帧关联起来，而内置锁却可以。**

**因为内置锁是JVM的内置属性，所以未来更可能提升synchronized而不是ReentrantLock的性能。**例如对线程封闭的锁对象消除优化，通过增加锁粒度来消除内置锁的同步。

**35） Java中ConcurrentHashMap的并发度是什么？**

ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

concurrencyLevel，能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，**在Java8之前实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。**正确地估计很重要，当低估，数据结构将根据额外的竞争，从而导致线程试图写入当前锁定的段时阻塞；相反，如果高估了并发级别，你遇到过大的膨胀，由于段的不必要的数量; 这种膨胀可能会导致性能下降，由于高数缓存未命中。

**在Java8里，仅仅是为了兼容旧版本而保留。唯一的作用就是保证构造map时初始容量不小于concurrencyLevel。**

**36） Java中Semaphore是什么？**

Java中的**Semaphore是一种新的同步类，它是一个计数信号**。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如数据库连接池。

**37）如果你提交任务时，线程池队列已满。会时发会生什么？**

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常。

**38) Java线程池中submit() 和 execute()方法有什么区别？**

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

**39) 什么是阻塞式方法？**

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

**40) Swing是线程安全的吗？ 为什么？**

你可以很肯定的给出回答，Swing不是线程安全的，但是你应该解释这么回答的原因即便面试官没有问你为什么。当我们说swing不是线程安全的常常提到它的组件，这些组件不能在多线程中进行修改，所有对GUI组件的更新都要在AWT线程中完成，而Swing提供了同步和异步两种回调方法来进行更新。

**41） Java中invokeAndWait 和 invokeLater有什么区别？**

这两个方法是Swing API 提供给Java开发者用来从当前线程而不是事件派发线程更新GUI组件用的。InvokeAndWait()同步更新GUI组件，比如一个进度条，一旦进度更新了，进度条也要做出相应改变。如果进度被多个线程跟踪，那么就调用invokeAndWait()方法请求事件派发线程对组件进行相应更新。而invokeLater()方法是异步调用更新组件的。

**42) Swing API中那些方法是线程安全的？**

这个问题又提到了swing和线程安全，虽然组件不是线程安全的但是有一些方法是可以被多线程安全调用的，比如repaint(), revalidate()。 JTextComponent的setText()方法和JTextArea的insert() 和 append() 方法也是线程安全的。

**43) 如何在Java中创建Immutable对象？**

这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：**通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。**

**16）我们能创建一个包含可变对象的不可变对象吗？**

是的，我们是可以创建一个包含可变对象的不可变对象的，你只需要谨慎一点，不要共享可变对象的引用就可以了，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。

**44） Java中的ReadWriteLock是什么？**

一般而言，读写锁是用来提升并发程序性能的锁分离技术的成果。Java中的ReadWriteLock是Java 5 中新增的一个接口，一个ReadWriteLock维护一对关联的锁，一个用于只读操作一个用于写。在没有写线程的情况下一个读锁可能会同时被多个读线程持有。写锁是独占的，你可以使用JDK中的ReentrantReadWriteLock来实现这个规则，它最多支持65535个写锁和65535个读锁。

**45) 多线程中的忙循环是什么?**

忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

**46）volatile 变量和 atomic 变量有什么不同？**

这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

**47) 如果同步块内的线程抛出异常会发生什么？**

这个问题坑了很多Java程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁

**49） 如何在Java中创建线程安全的Singleton？**

这是上面那个问题的后续，如果你不喜欢双检锁而面试官问了创建Singleton类的替代方法，你可以利用JVM的类加载和静态变量初始化特征来创建Singleton实例，或者是利用枚举类型来创建Singleton，我很喜欢用这种方法。

**50) 写出3条你遵循的多线程最佳实践**

这种问题我最喜欢了，我相信你在写并发代码来提升性能的时候也会遵循某些最佳实践。以下三条最佳实践我觉得大多数Java程序员都应该遵循：

**给你的线程起个有意义的名字。**

这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

**避免锁定和缩小同步的范围**

锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。

**多用同步类少用wait 和 notify**

首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

**多用并发集合少用同步集合**

这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。

**51) 如何强制启动一个线程？**

这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

**52) Java中的fork join框架是什么？**

fork join框架是JDK7中出现的一款高效的工具，Java开发人员可以通过它充分利用现代服务器上的多处理器。它是专门为了那些可以递归划分成许多子模块设计的，目的是将所有可用的处理能力用来提升程序的性能。fork join框架一个巨大的优势是它使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行。

**53） Java多线程中调用wait() 和 sleep()方法有什么不同？**

Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

**什么是Daemon线程**

Daemon线程（守护线程，后台线程）是值程序运行时在后台提供一种通用服务的线程，并且这种线程不是程序中不可或缺的部分（例如垃圾回收线程就是一种后台线程）。所有非后台线程结束时会杀死所有的后台线程。只要有非后台线程没结束，程序就不会结束。

可以通过Thread的setDaemon方法将线程设置为Daemon线程，该方法必须在线程启动之前调用

Daemon线程创建的线程也是Daemon线程

Daemon不应该访问数据库、文件等资源，因为它随时有可能被中断（甚至无法执行finall中的语句）

**什么是可重入锁（ReentrantLock）？**

ReentrantLock由最近成功获取锁，还没有释放的线程所拥有，当锁被另一个线程拥有时，调用lock方法的线程可以成功获取锁。如果锁已经被当前线程拥有，当前线程会立即返回。在AQS里面有一个state字段，在ReentrantLock中表示锁被持有的次数，它是一个volatile类型的整型值，因此对它的修改可以保证其他线程可以看到。ReentrantLock顾名思义就是锁可以重入，一个线程持有锁，state=1,如果它再次调用lock方法，那么他将继续拥有这把锁，state=2.当前可重入锁要完全释放，调用了多少次lock方法，还得调用等量的unlock方法来完全释放锁。

AbstractQueuedSynchronizer又称为队列同步器(后面简称AQS)，它是用来构建锁或其他同步组件的基础框架，内部通过一个int类型的成员变量state来控制同步状态,当state=0时，则说明没有任何线程占有共享资源的锁，当state=1时，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待，AQS内部通过内部类Node构成FIFO的同步队列来完成线程获取锁的排队工作，同时利用内部类ConditionObject构建等待队列，当Condition调用wait()方法后，线程将会加入等待队列中，而当Condition调用signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步队列等待，而另一种则是等待队列(可有多个)，通过Condition调用await()方法释放锁后，将加入等待队列

**synchronized和java.util.concurrent.locks.Lock的异同？**

synchronized和ReentrantLock都可以用于线程同步，synchronized中使用的IntrinsicLock和ReentrantLock都是可重入的。它们之间的区别如下：

synchronized是Java语言的一个特性，得到虚拟机的直接支持，Lock是concurrent包下的类

synchronized在进入同步方法或同步代码块时会自动获取锁，在返回同步方法或者退出同步代码块时会自动释放锁，但是ReentrantLock必须显式地获取锁，并且一定要在finally中显式释放锁，如果忘了显式释放，获取的锁无法被其他线程获取，有可能造成死锁。

ReentrantLock提供了更大的灵活性

可以通过tryLock实现轮询或定时获取锁，可用于避免死锁的发生

线程在等待内置锁而阻塞时无法响应中断（因为线程认为它肯定可以获取锁，所以不会响应中断请求）。ReentrantLock的lockInterruptibly方法能够在获取锁的过程中保持对中断的响应

synchronized方法和synchronized块都是基于块结构的加锁，ReentrantLock可用于非块结构加锁（例如ConcurrentHashMap中的分段锁)

ReentrantLock可以实现公平锁。synchronized使用的内置锁和ReentrantLock默认都是非公平的，ReentrantLock在构造时可选择公平锁。

公平锁是指如果有其它线程持有锁，或者阻塞队列中有其它线程，新请求锁的线程会被放到阻塞队列的末尾，按照先来后到的顺序；非公平锁是指只有锁被其它线程持有时，才会被放入阻塞队列，如果没有其它线程持有，所有线程都可以获取锁。

ps1：即使是公平锁，使用可轮询的tryLock，线程仍然可以插队。

ps2：使用公平锁会降低吞吐率和性能，只有持有锁的时间相对较长（请求锁的平均时间间隔较长），并且公平性确实很重要时才使用公平锁。默认实现的统计公平性已经足够了。

ReentrantLock作为一种高级工具提供了更好的灵活性，只有在synchronized无法满足需求时，才应该考虑使用ReentrantLock。

**SynchronizedMap和ConcurrentHashMap有什么区别？**

SynchronizedMap的方法都是synchronized的，对SynchronizedMap进行get、put、isEmpty、size等操作都会对整个表加锁；ConcurrentHashMap使用了分段锁（Lock Striping）技术，默认情况下ConcurrentHashMap中包含16个ReentrantLock，每个ReentrantLock守护若干个桶。对ConcurrentHashMap执行读操作时不会加锁，对ConcurrentHashMap的写操作会代理到每个分段锁上，相当于只锁住了整个表的1/16，提高了并发度。

但是，ConcurrentHashMap提供的是弱一致性，从ConcurrentHashMap读取数据（get，containsKey, size等操作）时使用的是原子语义，有可能读到过时数据。如果需要强一致性，必须使用synchronizedMap。

**CopyOnWriteArrayList可以用于什么应用场景？**

CopyOnWriteArrayList（写时复制）是一个线程安全的容器，它的线程安全性在于：对容器的修改可以和读操作同时进行，只要读操作能看到完成修改后的结果即可。从容器中读时不需要加锁，对容器中的元素进行修改时，先复制一份所有元素的副本，然后在新的副本上进行操作。CopyOnWriteArrayList适用于读操作远远多于写操作的场景。例如，缓存。

**同步有几种实现方法？**

1. 使用synchronized实现同步

2. 对于有条件的同步使用wait和notify/notifyAll

3. 显式地使用lock/unlock和await, singnal/signalAll

4. 更高的抽象级别——直接使用阻塞队列

**多读少写的场景应该使用哪个并发容器，为什么使用它？**

比如你做了一个搜索引擎，搜索引擎每次搜索前需要判断搜索关键词是否在黑名单里，黑名单每天更新一次。

CopyOnWriteArrayList这个容器适用于多读少写。读写并不是在同一个对象上。在写时会大面积复制数组，所以写的性能差，在写完成后将读的引用改为执行写的对象

**如何实现乐观锁（CAS）？如何避免ABA问题？**

传统的基于synchronized和显式lock（互斥同步）的方式是一种悲观锁，这种策略认为，只要不做正确的同步措施（加锁），一定会出现问题。CAS（Compare and swap）是一种非阻塞的乐观加锁策略，CAS直接使用处理器提供的cmpxchg提供同步。CAS的三个参数内存地址值V，预测值A和新值B。在执行CAS操作时，处理器先比较内存中的值和预测值是否相等，如果相等就用新值B更新；如果和预测值不相等，表示内存中的值已被修改，更新失败。更新失败的线程不会被阻塞，可以再次尝试更新。

ABA问题是指，如果CAS中V的值首先由A变为B，再由B变为A，CAS比较时会认为V的值没有发生变化。解决ABA问题可以引入另外一个变量表示V的版本号，即使由A变为B再由B变为A，V的版本号是不一样的。Java的Concurrent包中提供了AtomicStampedReference和AtomicMarkableReference来解决ABA问题。

**什么场景下可以使用volatile替换synchronized？**

只需要保证共享资源的可见性的时候可以使用volatile替代，synchronized保证可操作的原子性一致性和可见性。

volatile适用于新值不依赖于旧值的情形。

1写N读

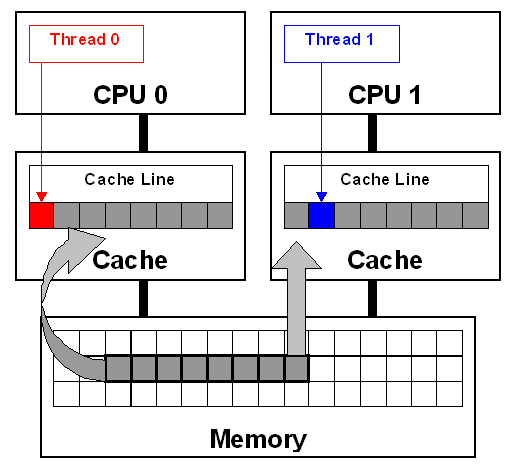
不与其他变量构成不变性条件时候使用volatile

**如何让一段程序并发的执行，并最终汇总结果？**

1. CountDownLatch：允许一个或者多个线程等待前面的一个或多个线程完成，构造一个CountDownLatch时指定需要CountDown的点的数量，每完成一点就count down一下，当所有点都完成，latch.wait就解除阻塞。

2. CyclicBarrier：可循环使用的Barrier，它的作用是让一组线程到达一个Barrier后阻塞，直到所有线程都到达Barrier后才能继续执行。CountDownLatch的计数值只能使用一次，CyclicBarrier可以通过使用reset重置；还可以指定到达栅栏后优先执行的任务。

3. fork/join框架，fork把大任务分解成多个小任务，然后汇总多个小任务的结果得到最终结果。使用一个双端队列，当线程空闲时从双端队列的另一端领取任务。

7）什么是多线程环境下的伪共享（false sharing）？  
伪共享是多线程系统（每个处理器有自己的局部缓存）中一个众所周知的性能问题。伪共享发生在不同处理器的上的线程对变量的修改依赖于相同的缓存行，如下图所示：  
[](http://jbcdn2.b0.upaiyun.com/2015/11/2bccd7f52a70db95aa72524ef3a55164.gif" \o "最近5年133个Java面试问题列表" \t "http://www.importnew.com/_blank)

### 有经验程序员的 Java 面试题

伪共享问题很难被发现，因为线程可能访问完全不同的全局变量，内存中却碰巧在很相近的位置上。如其他诸多的并发问题，避免伪共享的最基本方式是仔细审查代码，根据缓存行来调整你的数据结构。

8）什么是 Busy spin？我们为什么要使用它？  
Busy spin 是一种在不释放 CPU 的基础上等待事件的技术。它经常用于避免丢失 CPU 缓存中的数据（如果线程先暂停，之后在其他CPU上运行就会丢失）。所以，如果你的工作要求低延迟，并且你的线程目前没有任何顺序，这样你就可以通过循环检测队列中的新消息来代替调用 sleep() 或 wait() 方法。它唯一的好处就是你只需等待很短的时间，如几微秒或几纳秒。LMAX 分布式框架是一个高性能线程间通信的库，该库有一个 BusySpinWaitStrategy 类就是基于这个概念实现的，使用 busy spin 循环 EventProcessors 等待屏障。

9）Java 中怎么获取一份线程 dump 文件？  
在 Linux 下，你可以通过命令 kill -3 PID （Java 进程的进程 ID）来获取 Java 应用的 dump 文件。在 Windows 下，你可以按下 Ctrl + Break 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。如果你使用Tomcat。

Java最初被设计为一种安全的受控环境。尽管如此，HotSpot还是包含了一个后门sun.misc.Unsafe，提供了一些可以直接操控内存和线程的底层操作。Unsafe被JDK广泛应用于java.nio和并发包等实现中，这个不安全的类提供了一个观察HotSpot JVM内部结构并且可以对其进行修改，但是不建议在生产环境中使用。

Unsafe类是如此地不安全，以至于JDK开发者增加了很多特殊限制来访问它。

**私有的构造器**

**工厂方法getUnsafe()的调用器只能被Bootloader加载，否则抛出SecurityException 异常**

不过，我们可以通过反射机制轻松获取Unsafe的一个实例。

public static Unsafe getUnsafe() {

try {

Field f = Unsafe.class.getDeclaredField("theUnsafe");

f.setAccessible(true);

return (Unsafe)f.get(null);

} catch (Exception e) {

/\* ... \*/

}

}

**条件变量Condition**

条件变量很大一个程度上是为了解决Object.wait/notify/notifyAll难以使用的问题。

public class ConditionObject implements Condition, java.io.Serializable {

/\*\* First node of condition queue. \*/

private transient Node firstWaiter;

/\*\* Last node of condition queue. \*/

private transient Node lastWaiter;

public final void signal() {}

public final void signalAll() {}

public final void awaitUninterruptibly() {}

public final void await() throws InterruptedException {}

}

Synchronized中，所有的线程都在同一个object的条件队列上等待。而ReentrantLock中，每个condition都维护了一个条件队列。

每一个Lock可以有任意数据的Condition对象，Condition是与Lock绑定的，所以就有Lock的公平性特性：如果是公平锁，线程为按照FIFO的顺序从Condition.await中释放，如果是非公平锁，那么后续的锁竞争就不保证FIFO顺序了。

Condition接口定义的方法，await对应于Object.wait，signal对应于Object.notify，signalAll对应于Object.notifyAll。特别说明的是Condition的接口改变名称就是为了避免与Object中的wait/notify/notifyAll的语义和使用上混淆。

**CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作。**

**ABA问题。**因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那 么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时 候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

**从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。**这个类的 compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的 值设置为给定的更新值。

public boolean compareAndSet

(V expectedReference,//预期引用

V newReference,//更新后的引用

int expectedStamp, //预期标志

int newStamp) //更新后的标志

**循环时间长开销大。**自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的 提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现 的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

**只能保证一个共享变量的原子操作。**当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量 i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。**从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的 原子性，你可以把多个变量放在一个对象里来进行CAS操作。**

**Volatile的使用优化**

**著名的Java并发编程大师Doug lea在JDK7的并发包里新增一个队列集合类LinkedTransferQueue，他在使用Volatile变量时，用一种追加字节的方式来优化队列出队和入队的性能。**

**追加字节能优化性能？这种方式看起来很神奇，但如果深入理解处理器架构就能理解其中的奥秘。让我们先来看看LinkedTransferQueue这个类，它使用一个内部类类型来定义队列的头队列（Head）和尾节点（tail），而这个内部类PaddedAtomicReference相对于父类AtomicReference只做了一件事情，就将共享变量追加到64字节。我们可以来计算下，一个对象的引用占4个字节，它追加了15个变量共占60个字节，再加上父类的Value变量，一共64个字节。**

**/\*\* head of the queue \*/**

**private transient final PaddedAtomicReference<QNode> head;**

**/\*\* tail of the queue \*/**

**private transient final PaddedAtomicReference<QNode> tail;**

**static final class PaddedAtomicReference <T> extends AtomicReference <T> {**

**// enough padding for 64bytes with 4byte refs**

**Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;**

**PaddedAtomicReference(T r) {**

**super(r);**

**}**

**}**

**public class AtomicReference <V> implements java.io.Serializable {**

**private volatile V value;**

**//省略其他代码**

**｝**

**为什么追加64字节能够提高并发编程的效率呢？ 因为对于英特尔酷睿i7，酷睿， Atom和NetBurst， Core Solo和Pentium M处理器的L1，L2或L3缓存的高速缓存行是64个字节宽，不支持部分填充缓存行，这意味着如果队列的头节点和尾节点都不足64字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头尾节点，当一个处理器试图修改头接点时会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作是需要不停修改头接点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。Doug lea使用追加到64字节的方式来填满高速缓冲区的缓存行，避免头接点和尾节点加载到同一个缓存行，使得头尾节点在修改时不会互相锁定。**

**那么是不是在使用Volatile变量时都应该追加到64字节呢？不是的。在两种场景下不应该使用这种方式。第一：缓存行非64字节宽的处理器，如P6系列和奔腾处理器，它们的L1和L2高速缓存行是32个字节宽。第二：共享变量不会被频繁的写。因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定的性能消耗，共享变量如果不被频繁写的话，锁的几率也非常小，就没必要通过追加字节的方式来避免相互锁定。**

**AQS，AbstractQueuedSynchronizer，即队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentrantLock、ReentrantReadWriteLock、Semaphore、CountDownLatch、SynchronousQueue和FutureTask等）**，JUC并发包的作者（****Doug Lea****）期望它能够成为实现大部分同步需求的基础。它是JUC并发包中的核心基础组件。

AQS解决了子啊实现同步器时涉及当的大量细节问题，例如获取同步状态、FIFO同步队列。基于AQS来构建同步器可以带来很多好处。它不仅能够极大地减少实现工作，而且也不必处理在多个位置上发生的竞争问题。

在基于AQS构建的同步器中，只能在一个时刻发生阻塞，从而降低上下文切换的开销，提高了吞吐量。同时在设计AQS时充分考虑了可伸缩行，因此J.U.C中所有基于AQS构建的同步器均可以获得这个优势。

AQS的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态。

AQS使用一个int类型的成员变量state来表示同步状态，当state>0时表示已经获取了锁，当state = 0时表示释放了锁。它提供了三个方法（getState()、setState(int newState)、compareAndSetState(int expect,int update)）来对同步状态state进行操作，当然AQS可以确保对state的操作是安全的。

AQS通过内置的FIFO同步队列来完成资源获取线程的排队工作，如果当前线程获取同步状态失败（锁）时，AQS则会将当前线程以及等待状态等信息构造成一个节点（Node）并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，则会把节点中的线程唤醒，使其再次尝试获取同步状态。

AQS主要提供了如下一些方法：

* getState()：返回同步状态的当前值；
* setState(int newState)：设置当前同步状态；
* compareAndSetState(int expect, int update)：使用CAS设置当前状态，该方法能够保证状态设置的原子性；
* tryAcquire(int arg)：独占式获取同步状态，获取同步状态成功后，其他线程需要等待该线程释放同步状态才能获取同步状态；
* tryRelease(int arg)：独占式释放同步状态；
* tryAcquireShared(int arg)：共享式获取同步状态，返回值大于等于0则表示获取成功，否则获取失败；
* tryReleaseShared(int arg)：共享式释放同步状态；
* isHeldExclusively()：当前同步器是否在独占式模式下被线程占用，一般该方法表示是否被当前线程所独占；
* acquire(int arg)：独占式获取同步状态，如果当前线程获取同步状态成功，则由该方法返回，否则，将会进入同步队列等待，该方法将会调用可重写的tryAcquire(int arg)方法；
* acquireInterruptibly(int arg)：与acquire(int arg)相同，但是该方法响应中断，当前线程为获取到同步状态而进入到同步队列中，如果当前线程被中断，则该方法会抛出InterruptedException异常并返回；
* tryAcquireNanos(int arg,long nanos)：超时获取同步状态，如果当前线程在nanos时间内没有获取到同步状态，那么将会返回false，已经获取则返回true；
* acquireShared(int arg)：共享式获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式的主要区别是在同一时刻可以有多个线程获取到同步状态；
* acquireSharedInterruptibly(int arg)：共享式获取同步状态，响应中断；
* tryAcquireSharedNanos(int arg, long nanosTimeout)：共享式获取同步状态，增加超时限制；
* release(int arg)：独占式释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒；
* releaseShared(int arg)：共享式释放同步状态；

**大多数开发者都不会直接使用AQS，标准同步器类的集合能够满足绝大多数情况的需求。但如果能了解标准同步器类的实现方式，那么对于理解它们的工作原理是非常有帮助的。**

**AQS负责管理同步器类中的状态，它管理了一个整数状态信息，可以通过getState,setState以及compareAndSetState等protected类型方法来进行操作。这个整数可以用于表示任意状态。例如，ReentrantLock用它来表示所有者线程已经重复获取该锁的次数，Semaphore用它来表示剩余的许可数量，FutureTask用它来表示任务的状态（尚未开始、正在运行、已完成以及已取消）。在同步器类中还可以自行管理一些额外的状态变量，例如，ReentrantLock保存了锁的当前所有者的信息，这样就能区分某个获取操作是重入的还是竞争的。**

/\*\* \* AQS抽象类 \*/

public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer{

//指向同步队列队头

private transient volatile Node head;

//指向同步的队尾

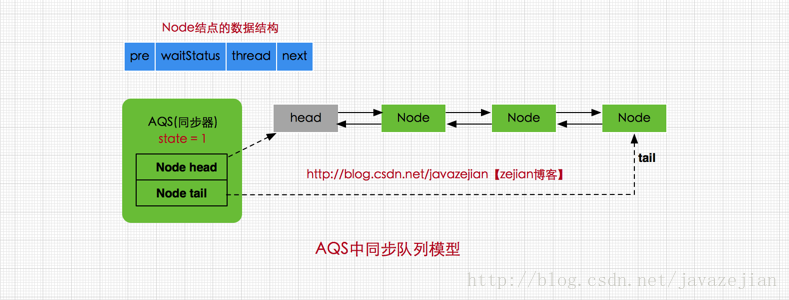
private transient volatile Node tail;

//同步状态，0代表锁未被占用，1代表锁已被占用

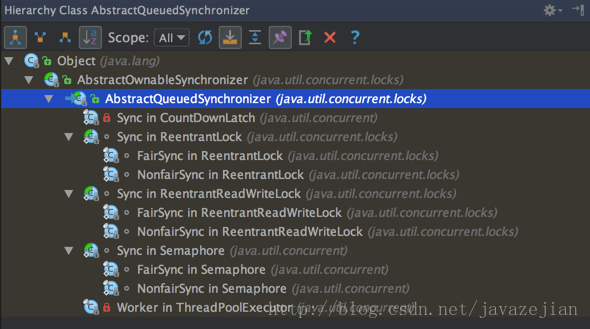
private volatile int state;

//省略其他代码......

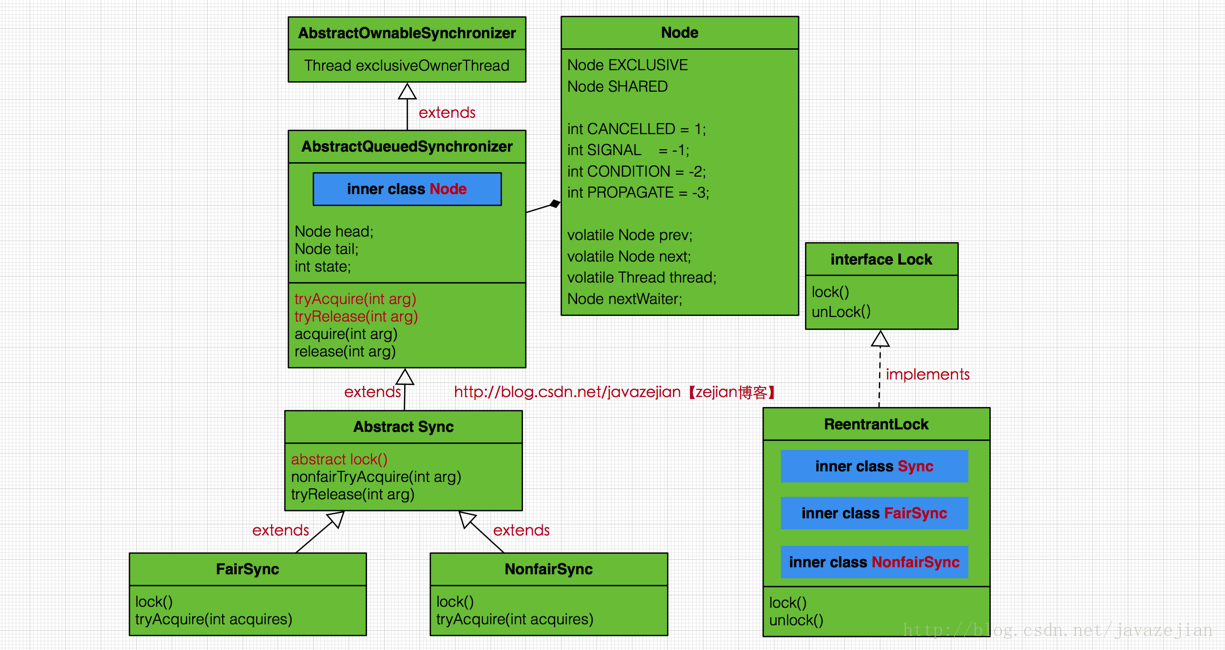
}



head和tail分别是AQS中的变量，其中head指向同步队列的头部，注意head为空结点，不存储信息。而tail则是同步队列的队尾，同步队列采用的是双向链表的结构这样可方便队列进行结点增删操作。state变量则是代表同步状态，执行当线程调用lock方法进行加锁后，如果此时state的值为0，则说明当前线程可以获取到锁(在本篇文章中，锁和同步状态代表同一个意思)，同时将state设置为1，表示获取成功。如果state已为1，也就是当前锁已被其他线程持有，那么当前执行线程将被封装为Node结点加入同步队列等待。其中Node结点是对每一个访问同步代码的线程的封装，从图中的Node的数据结构也可看出，其包含了需要同步的线程本身以及线程的状态，如是否被阻塞，是否等待唤醒，是否已经被取消等。每个Node结点内部关联其前继结点prev和后继结点next，这样可以方便线程释放锁后快速唤醒下一个在等待的线程，Node是AQS的内部类



这里以ReentrantLock为例，简单讲解ReentrantLock与AQS的关系



AbstractOwnableSynchronizer：抽象类，定义了存储独占当前锁的线程和获取的方法

AbstractQueuedSynchronizer：抽象类，AQS框架核心类，其内部以虚拟队列的方式管理线程的锁获取与锁释放，其中获取锁(tryAcquire方法)和释放锁(tryRelease方法)并没有提供默认实现，需要子类重写这两个方法实现具体逻辑，目的是使开发人员可以自由定义获取锁以及释放锁的方式。

Node：AbstractQueuedSynchronizer 的内部类，用于构建虚拟队列(链表双向链表)，管理需要获取锁的线程。

Sync：抽象类，是ReentrantLock的内部类，继承自AbstractQueuedSynchronizer，实现了释放锁的操作(tryRelease()方法)，并提供了lock抽象方法，由其子类实现。

NonfairSync：是ReentrantLock的内部类，继承自Sync，非公平锁的实现类。

FairSync：是ReentrantLock的内部类，继承自Sync，公平锁的实现类。

ReentrantLock：实现了Lock接口的，其内部类有Sync、NonfairSync、FairSync，在创建时可以根据fair参数决定创建NonfairSync(默认非公平锁)还是FairSync。

ReentrantLock内部存在3个实现类，分别是Sync、NonfairSync、FairSync，其中Sync继承自AQS实现了解锁tryRelease()方法，而NonfairSync(非公平锁)、 FairSync(公平锁)则继承自Sync，实现了获取锁的tryAcquire()方法，ReentrantLock的所有方法调用都通过间接调用AQS和Sync类及其子类来完成的。从上述类图可以看出AQS是一个抽象类，但请注意其源码中并没一个抽象的方法，这是因为AQS只是作为一个基础组件，并不希望直接作为直接操作类对外输出，而更倾向于作为基础组件，为真正的实现类提供基础设施，如构建同步队列，控制同步状态等，事实上，从设计模式角度来看，AQS采用的模板模式的方式构建的，其内部除了提供并发操作核心方法以及同步队列操作外，还提供了一些模板方法让子类自己实现，如加锁操作以及解锁操作，为什么这么做？这是因为AQS作为基础组件，封装的是核心并发操作，但是实现上分为两种模式，即共享模式与独占模式，而这两种模式的加锁与解锁实现方式是不一样的，但AQS只关注内部公共方法实现并不关心外部不同模式的实现，所以提供了模板方法给子类使用，也就是说实现独占锁，如ReentrantLock需要自己实现tryAcquire()方法和tryRelease()方法，而实现共享模式的Semaphore，则需要实现tryAcquireShared()方法和tryReleaseShared()方法，这样做的好处是显而易见的，无论是共享模式还是独占模式，其基础的实现都是同一套组件(AQS)，只不过是加锁解锁的逻辑不同罢了，更重要的是如果我们需要自定义锁的话，也变得非常简单，只需要选择不同的模式实现不同的加锁和解锁的模板方法即可，AQS提供给独占模式和共享模式的模板方法如下

//AQS中提供的主要模板方法，由子类实现。

public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer{

//独占模式下获取锁的方法

protected boolean tryAcquire(int arg)

{ throw new UnsupportedOperationException(); }

//独占模式下解锁的方法

protected boolean tryRelease(int arg)

{ throw new UnsupportedOperationException(); }

//共享模式下获取锁的方法

protected int tryAcquireShared(int arg)

{ throw new UnsupportedOperationException(); }

//共享模式下解锁的方法

protected boolean tryReleaseShared(int arg)

{ throw new UnsupportedOperationException(); }

//判断是否为持有独占锁

protected boolean isHeldExclusively()

{ throw new UnsupportedOperationException(); } }

## 基于ReetrantLock分析AQS独占模式实现过程

### ReetrantLock中非公平锁

AQS同步器的实现依赖于内部的同步队列(FIFO的双向链表对列)完成对同步状态(state)的管理，当前线程获取锁(同步状态)失败时，AQS会将该线程以及相关等待信息包装成一个节点(Node)并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会将头结点head中的线程唤醒，让其尝试获取同步状态。关于同步队列和Node结点，前面我们已进行了较为详细的分析，这里重点分析一下获取同步状态和释放同步状态以及如何加入队列的具体操作，这里从ReetrantLock入手分析AQS的具体实现，我们先以非公平锁为例进行分析

//默认构造，创建非公平锁NonfairSync

public ReentrantLock()

{ sync = new NonfairSync(); }

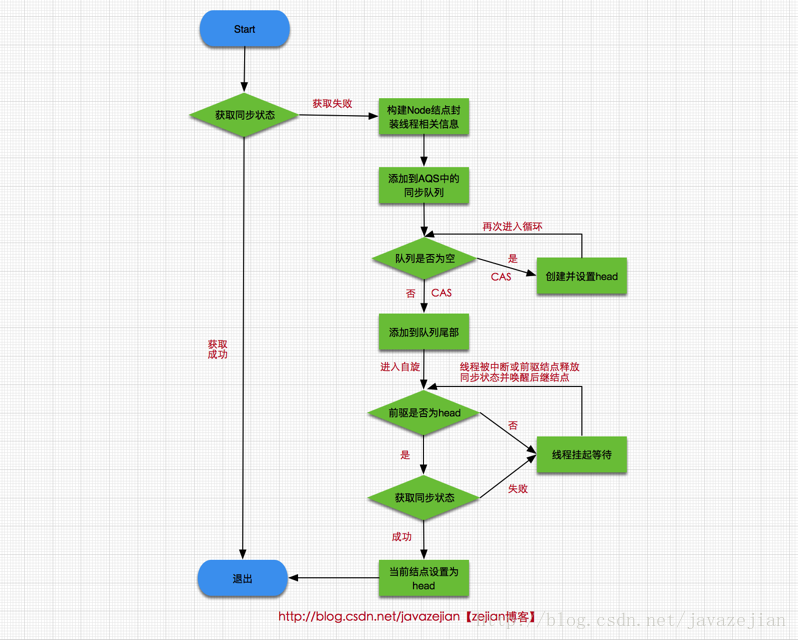
//根据传入参数创建锁类型

public ReentrantLock(boolean fair)

{ sync = fair ? new FairSync() : new NonfairSync(); }

//加锁操作

public void lock() { sync.lock(); }

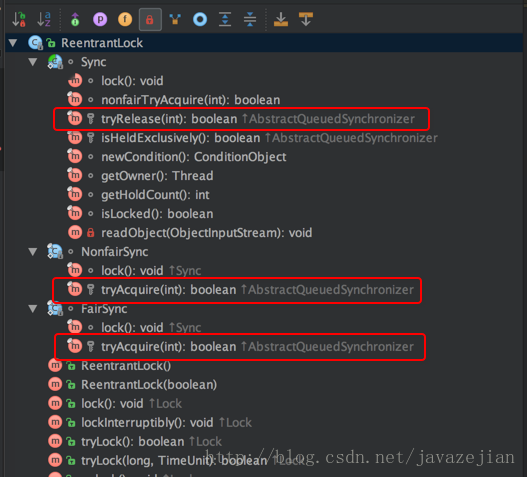


关于获取锁的操作，这里看看另外一种可中断的获取方式，即调用ReentrantLock类的lockInterruptibly()或者tryLock()方法，最终它们都间接调用到doAcquireInterruptibly()

**这就是非公平锁与公平锁最大的区别**，即公平锁在线程请求到来时先会判断同步队列是否存在结点，如果存在先执行同步队列中的结点线程，当前线程将封装成node加入同步队列等待。而非公平锁呢，当线程请求到来时，不管同步队列是否存在线程结点，直接尝试获取同步状态，获取成功直接访问共享资源，但请注意在绝大多数情况下，非公平锁才是我们理想的选择，毕竟从效率上来说非公平锁总是胜于公平锁。

<http://blog.csdn.net/javazejian/article/details/75043422?locationNum=1&fps=1>

以上便是ReentrantLock的内部实现原理，这里我们简单进行小结，重入锁ReentrantLock，是一个基于AQS并发框架的并发控制类，其内部实现了3个类，分别是Sync、NoFairSync以及FairSync类，其中Sync继承自AQS，实现了释放锁的模板方法tryRelease(int)，而NoFairSync和FairSync都继承自Sync，实现各种获取锁的方法tryAcquire(int)。ReentrantLock的所有方法实现几乎都间接调用了这3个类，因此当我们在使用ReentrantLock时，大部分使用都是在间接调用AQS同步器中的方法，这就是ReentrantLock的内部实现原理,最后给出张类图结构



# 神奇的Condition

## 关于Condition接口

在并发编程中，每个Java对象都存在一组监视器方法，如wait()、notify()以及notifyAll()方法，通过这些方法，我们可以实现线程间通信与协作（也称为等待唤醒机制），如生产者-消费者模式，而且这些方法必须配合着synchronized关键字使用，关于这点，如果想有更深入的理解，可观看博主另外一篇博文【[深入理解Java并发之synchronized实现原理](http://blog.csdn.net/javazejian/article/details/72828483" \t "http://blog.csdn.net/javazejian/article/details/_blank)】，与synchronized的等待唤醒机制相比Condition具有更多的灵活性以及精确性，这是因为notify()在唤醒线程时是随机(同一个锁)，而Condition则可通过多个Condition实例对象建立更加精细的线程控制，也就带来了更多灵活性了，我们可以简单理解为以下两点

通过Condition能够精细的控制多线程的休眠与唤醒。

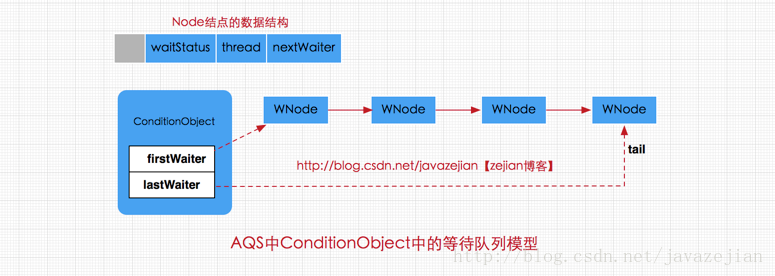
对于一个锁，我们可以为多个线程间建立不同的Condition。

Condition的实现类是AQS的内部类ConditionObject。

## Condition的实现原理

Condition的具体实现类是AQS的内部类ConditionObject，前面我们分析过AQS中存在两种队列，一种是同步队列，一种是等待队列，而等待队列就相对于Condition而言的。注意在使用Condition前必须获得锁，同时在Condition的等待队列上的结点与前面同步队列的结点是同一个类即Node，其结点的waitStatus的值为CONDITION。在实现类ConditionObject中有两个结点分别是firstWaiter和lastWaiter，firstWaiter代表等待队列第一个等待结点，lastWaiter代表等待队列最后一个等待结点

每个Condition都对应着一个等待队列，也就是说如果一个锁上创建了多个Condition对象，那么也就存在多个等待队列。等待队列是一个FIFO的队列，在队列中每一个节点都包含了一个线程的引用，而该线程就是Condition对象上等待的线程。当一个线程调用了await()相关的方法，那么该线程将会释放锁，并构建一个Node节点封装当前线程的相关信息加入到等待队列中进行等待，直到被唤醒、中断、超时才从队列中移出。Condition中的等待队列模型如下



正如图所示，Node节点的数据结构，在等待队列中使用的变量与同步队列是不同的，Condtion中等待队列的结点只有直接指向的后继结点并没有指明前驱结点，而且使用的变量是nextWaiter而不是next，这点我们在前面分析结点Node的数据结构时讲过。firstWaiter指向等待队列的头结点，lastWaiter指向等待队列的尾结点，等待队列中结点的状态只有两种即CANCELLED和CONDITION，前者表示线程已结束需要从等待队列中移除，后者表示条件结点等待被唤醒。再次强调每个Codition对象对于一个等待队列，也就是说AQS中只能存在一个同步队列，但可拥有多个等待队列。下面从代码层面看看被调用await()方法(其他await()实现原理类似)的线程是如何加入等待队列的，而又是如何从等待队列中被唤醒的

public final void await() throws InterruptedException {

//判断线程是否被中断

if (Thread.interrupted())

throw new InterruptedException();

//创建新结点加入等待队列并返回

Node node = addConditionWaiter();

//释放当前线程锁即释放同步状态

int savedState = fullyRelease(node);

int interruptMode = 0;

//判断结点是否同步队列(SyncQueue)中,即是否被唤醒

while (!isOnSyncQueue(node)) {

//挂起线程

LockSupport.park(this);

//判断是否被中断唤醒，如果是退出循环。

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

//被唤醒后执行自旋操作争取获得锁，同时判断线程是否被中断

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

// clean up if cancelled

if (node.nextWaiter != null)

//清理等待队列中不为CONDITION状态的结点

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

执行addConditionWaiter()添加到等待队列。

private Node addConditionWaiter() {

Node t = lastWaiter;

// 判断是否为结束状态的结点并移除

if (t != null && t.waitStatus != Node.CONDITION) {

unlinkCancelledWaiters();

t = lastWaiter;

}

//创建新结点状态为CONDITION

Node node = new Node(Thread.currentThread(), Node.CONDITION);

//加入等待队列

if (t == null)

firstWaiter = node;

else

t.nextWaiter = node;

lastWaiter = node;

return node;

}

await()方法主要做了3件事，一是调用addConditionWaiter()方法将当前线程封装成node结点加入等待队列，二是调用fullyRelease(node)方法释放同步状态并唤醒后继结点的线程。三是调用isOnSyncQueue(node)方法判断结点是否在同步队列中，注意是个while循环，如果同步队列中没有该结点就直接挂起该线程，需要明白的是如果线程被唤醒后就调用acquireQueued(node, savedState)执行自旋操作争取锁，即当前线程结点从等待队列转移到同步队列并开始努力获取锁。

接着看看唤醒操作singal()方法

public final void signal() {

//判断是否持有独占锁，如果不是抛出异常

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

//唤醒等待队列第一个结点的线程

if (first != null)

doSignal(first);

}

这里signal()方法做了两件事，一是判断当前线程是否持有独占锁，没有就抛出异常，从这点也可以看出只有独占模式先采用等待队列，而共享模式下是没有等待队列的，也就没法使用Condition。二是唤醒等待队列的第一个结点，即执行doSignal(first)

private void doSignal(Node first) {

do {

//移除条件等待队列中的第一个结点，

//如果后继结点为null，那么说没有其他结点将尾结点也设置为null

if ( (firstWaiter = first.nextWaiter) == null)

lastWaiter = null;

first.nextWaiter = null;

//如果被通知节点没有进入到同步队列并且条件等待队列还有不为空的节点，则继续循环通知后续结点

} while (!transferForSignal(first) &&

(first = firstWaiter) != null);

}

//transferForSignal方法final boolean transferForSignal(Node node) {

//尝试设置唤醒结点的waitStatus为0，即初始化状态

//如果设置失败，说明当期结点node的waitStatus已不为

//CONDITION状态，那么只能是结束状态了，因此返回false

//返回doSignal()方法中继续唤醒其他结点的线程，注意这里并

//不涉及并发问题，所以CAS操作失败只可能是预期值不为CONDITION，

//而不是多线程设置导致预期值变化，毕竟操作该方法的线程是持有锁的。

if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))

return false;

//加入同步队列并返回前驱结点p

Node p = enq(node);

int ws = p.waitStatus;

//判断前驱结点是否为结束结点(CANCELLED=1)或者在设置

//前驱节点状态为Node.SIGNAL状态失败时，唤醒被通知节点代表的线程

if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))

//唤醒node结点的线程

LockSupport.unpark(node.thread);

return true;

}

注释说得很明白了，这里我们简单整体说明一下，doSignal(first)方法中做了两件事，从条件等待队列移除被唤醒的节点，然后重新维护条件等待队列的firstWaiter和lastWaiter的指向。二是将从等待队列移除的结点加入同步队列(在transferForSignal()方法中完成的)，如果进入到同步队列失败并且条件等待队列还有不为空的节点，则继续循环唤醒后续其他结点的线程。到此整个signal()的唤醒过程就很清晰了，即signal()被调用后，先判断当前线程是否持有独占锁，如果有，那么唤醒当前Condition对象中等待队列的第一个结点的线程，并从等待队列中移除该结点，移动到同步队列中，如果加入同步队列失败，那么继续循环唤醒等待队列中的其他结点的线程，如果成功加入同步队列，那么如果其前驱结点是否已结束或者设置前驱节点状态为Node.SIGNAL状态失败，则通过LockSupport.unpark()唤醒被通知节点代表的线程，到此signal()任务完成，注意被唤醒后的线程，将从前面的await()方法中的while循环中退出，因为此时该线程的结点已在同步队列中，那么while (!isOnSyncQueue(node))将不在符合循环条件，进而调用AQS的acquireQueued()方法加入获取同步状态的竞争中，这就是等待唤醒机制的整个流程实现原理，流程如下图所示（注意无论是同步队列还是等待队列使用的Node数据结构都是同一个，不过是使用的内部变量不同罢了）

