码农求职小助手：Java并发编程

笔记本： 3-Java并发多线程
创建时间： 2019/9/7 21:15 **更新时间：** 2019/9/7 21:15
作者： pc941206@163.com

- 一、线程
  - 1.并行和并发有什么区别?
  - 2、线程和进程的区别?
  - 3、守护线程是什么?
  - 4、创建线程的几种方式?
  - 5、runnable 和 callable 有什么区别?
  - 6、线程状态【6种】
  - 7、sleep() 和 wait() 的区别?
  - 8、线程的 run() 和 start() 有什么区别?
  - 9、在 Java 程序中怎么保证多线程的运行安全?
  - 10、Java 线程同步的几种方法?
  - 11、Thread.interrupt() 方法
- 二、synchronized
  - 1. 说一说自己对于 synchronized 关键字的了解
  - 2. 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗?
  - 3. 讲一下 synchronized 关键字的底层原理
  - 4. 说说 JDK1.6 之后的 synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗?
    - 4.1 偏向锁
    - 4.2 轻量级锁
    - 4.3 自旋锁和自适应自旋
    - 4.4 锁消除
    - 4.5 锁粗化
  - 5. 谈谈 synchronized 和 ReenTrantLock 的区别?
    - 5.1 两者都是可重入锁
    - 5.2 synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API
    - 5.3 ReenTrantLock 比 synchronized 增加了一些高级功能相比 synchronized，ReenTrantLock 增加了一些高级功能
  - 6. synchronized 和 volatile 的区别是什么?
  - 7. 简单介绍下 volatile?
  - 8. ReentrantLock 和 ReentrantReadWriteLock
- 三、乐观锁和悲观锁
  - 1. 乐观锁和悲观锁的基本概念
    - 1.1 悲观锁：
    - 1.2 乐观锁：

        - 1.3 两种锁的使用场景
    - 2. 乐观锁常见的两种实现方式
        - 2.1 版本号机制
        - 2.2 CAS 算法
    - 3. 乐观锁的缺点
        - 3.1 ABA 问题：
        - 3.2 循环时间长开销大
        - 3.3 只能保证一个共享变量的原子操作
    - 4. CAS 和 synchronized 的使用场景
- 四、Atomic 原子类
    - 1. 原子类的基本介绍
        - 1.1 基本类型
        - 1.2 数组类型
        - 1.3 引用类型
        - 1.4 对象的属性修改类型
    - 2、说一下 atomic 的原理?
- 五、AQS：同步器【重要】
    - 1. AQS 的简单介绍
    - 2. AQS 原理
        - 2.1 AQS 原理概述
        - 2.2 AQS 对资源的共享模式
    - 3. Semaphore（信号量）：允许多个线程同时访问
    - 4. CountDownLatch （倒计时器）
        - 4.1 CountDownLatch 的三种典型用法
        - 4.2 CountDownLatch 的不足
        - 4.3 CountDownLatch 相常见面试题：
    - 5. CyclicBarrier(循环栅栏)
        - 5.1 CyclicBarrier 的应用场景
        - 5.2 CyclicBarrier 和 CountDownLatch 的区别
- 六、线程池
    - 1. 为什么要用线程池?
    - 2. 实现 Runnable 接口和 Callable 接口的区别?
    - 3. 执行 execute() 方法和 submit() 方法的区别是什么呢?
    - 4. 如何创建线程池?
    - 5. 线程池的参数
    - 6、线程池中一般设置多少线程，为什么?
    - 7、线程池线程的分配流程
- 七、Fork/Join 并行计算框架
- 八、并发容器
    - 1. JDK 提供的并发容器概述
    - 2. ConcurrentHashMap
    - 3. CopyOnWriteArrayList
        - 3.1 CopyOnWriteArrayList 简介
        - 3.2 CopyOnWriteArrayList 是如何做到的?

- 4. ConcurrentLinkedQueue
- 5. BlockingQueue
  - 5.1 BlockingQueue 简单介绍
  - 5.2 ArrayBlockingQueue
  - 5.3 LinkedBlockingQueue
  - 5.4 PriorityBlockingQueue
- 6. ConcurrentSkipListMap

- 九、其他
  - 1. ThradLocal
    - 使用场景【面试可能会问】

更多资料请关注微信公众号：**码农求职小助手**

一、线程

1.并行和并发有什么区别?

并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。

在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如 hadoop 分布式集群。所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

2、线程和进程的区别?

简而言之，**进程是程序运行和资源分配的基本单位**，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资

源，减少切换次数，从而效率更高。**线程是进程的一个实体，是 cpu 调度和分派的基本单位**，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

3、守护线程是什么?

守护线程（即 daemon thread），是个服务线程，准确地来说就是服务其他的线程。

4、创建线程的几种方式?

1、继承 Thread 类创建线程；

2、实现 Runnable 接口创建线程；

3、通过 Callable 和 Future 创建线程；

4、通过线程池创建线程。

5、runnable 和 callable 有什么区别?

Runnable 接口中的 run() 方法的返回值是 void，它做的事情只是纯粹地去执行 run() 方法中的代码而已；

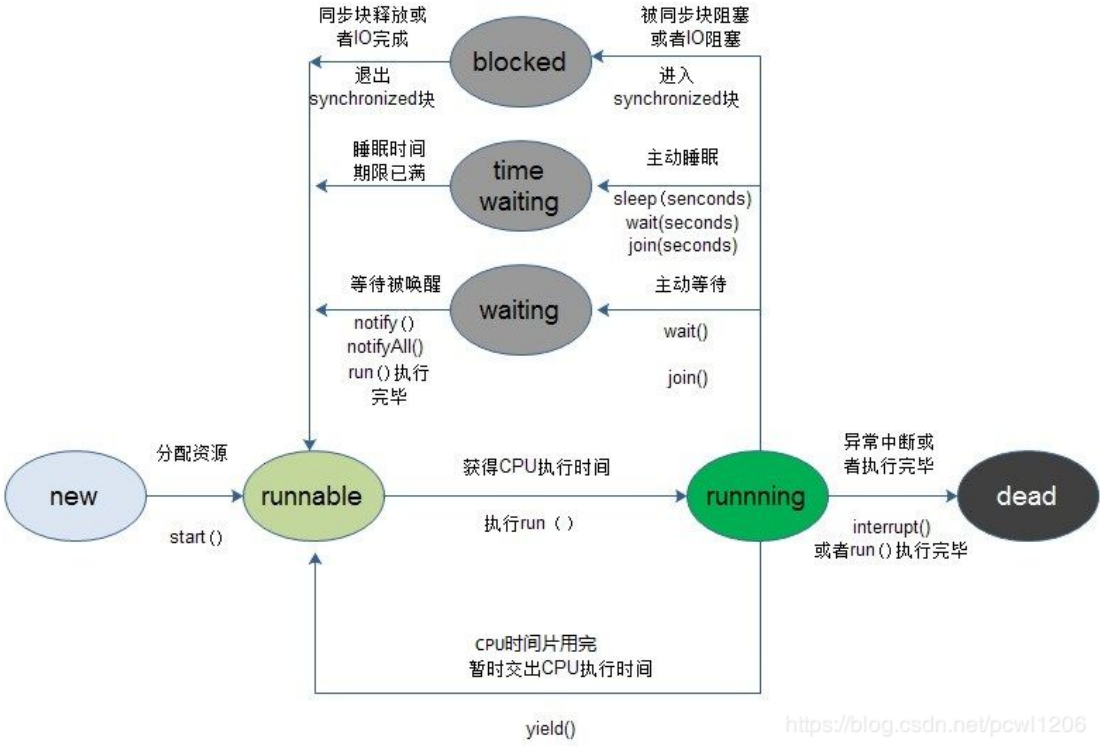
Callable 接口中的 call() 方法是有返回值的，是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。

6、线程状态【6种】

Thread的源码中定义了6种状态：new（新建）、runnnable（可运行）、blocked（阻塞）、waiting（等待）、time waiting （定时等待）和 terminated（终止）。

状态名称	说明
NEW	初始状态，线程被构建，但是还没有调用start()方法
RUNNABLE	运行状态，Java线程将操作系统中的就绪和运行两种状态笼统地称为“运行中”
BLOCKED	阻塞状态，表示线程阻塞与锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

状态转换图如下：



7、sleep() 和 wait() 的区别?

1、每个对象都有一个锁来控制同步访问，Synchronized 关键字可以和对象的锁交互，来实现同步方法或同步块。sleep() 方法正在执行的线程主动让出 CPU（然后CPU就可以去执行其他任务），在 sleep 指定时间后CPU 再回到该线程继续往下执行(注意：**sleep 方法只让出了 CPU，而并不会释放同步资源锁**)；**wait() 方法则是指当前线程让自己暂时退让出同步资源锁，以便其他正在等待该资源的线程得到该资源进而运行**，只有调用了 notify() 方法，之前调用 wait() 的线程才会解除 wait 状态，可以去参与竞争同步资源锁，进而得到执行。（注意：notify 的作用相当于叫醒睡着的人，而并不会给他分配任务，就是说 notify 只是让之前调用 wait 的线程有权利重新参与线程的调度）；

2、sleep() 方法可以在任何地方使用，而 wait() 方法则只能在同步方法或同步块中使用；

3、sleep() 是线程类（Thread）的方法，调用会暂停此线程指定的时间，但监控依然保持，不会释放对象锁，到时间自动恢复；wait() 是 Object 的方法，调用会放弃对象锁，进入等待队列，待调用 notify()/notifyAll() 唤醒指定的线程或者所有线程，才会进入锁池，不再次获得对象锁才会进入运行状态；

8、线程的 run() 和 start() 有什么区别?

1、每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作的，方法 run() 称为线程体。通过调用 Thread 类的 start() 方法来启动一个线程。

2、start() 方法来启动一个线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码； 这时此线程是处于就绪状态， 并没有运行。

然后通过此 Thread 类调用方法 run() 来完成其运行状态， 这里方法 run() 称为线程体，它包含了要执行的这个线程的内容， run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

3、run() 方法是在本线程里的，只是线程里的一个函数，而不是多线程的。 如果直接调用 run()，其实就相当于是调用了一个普通函数而已，直接待用 run() 方法必须等待 run() 方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用 start() 方法而不是 run() 方法。

9、在 Java 程序中怎么保证多线程的运行安全?

线程安全在三个方面体现：

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，（atomic，synchronized）；

可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized，volatile）；

有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before 原则）。

10、Java 线程同步的几种方法?

详解：https://blog.csdn.net/scgyus/article/details/79499650

1、使用 Synchronized 关键字；

2、wait 和 notify；

3、使用特殊域变量 volatile 实现线程同步；

4、使用可重入锁实现线程同步；

5、使用阻塞队列实现线程同步；

6、使用信号量 Semaphore。

.....

11、Thread.interrupt() 方法

详解：https://blog.csdn.net/tianyuxingxuan/article/details/76222935

在 Java 中，线程的中断(interrupt)只是改变了线程的中断状态，至于这个中断状态改变后带来的结果，那是无法确定的，有时它更是让停止中的线程继续执行的唯一手段。不但不是让线程停止运行，反而是继续执行线程的手段。

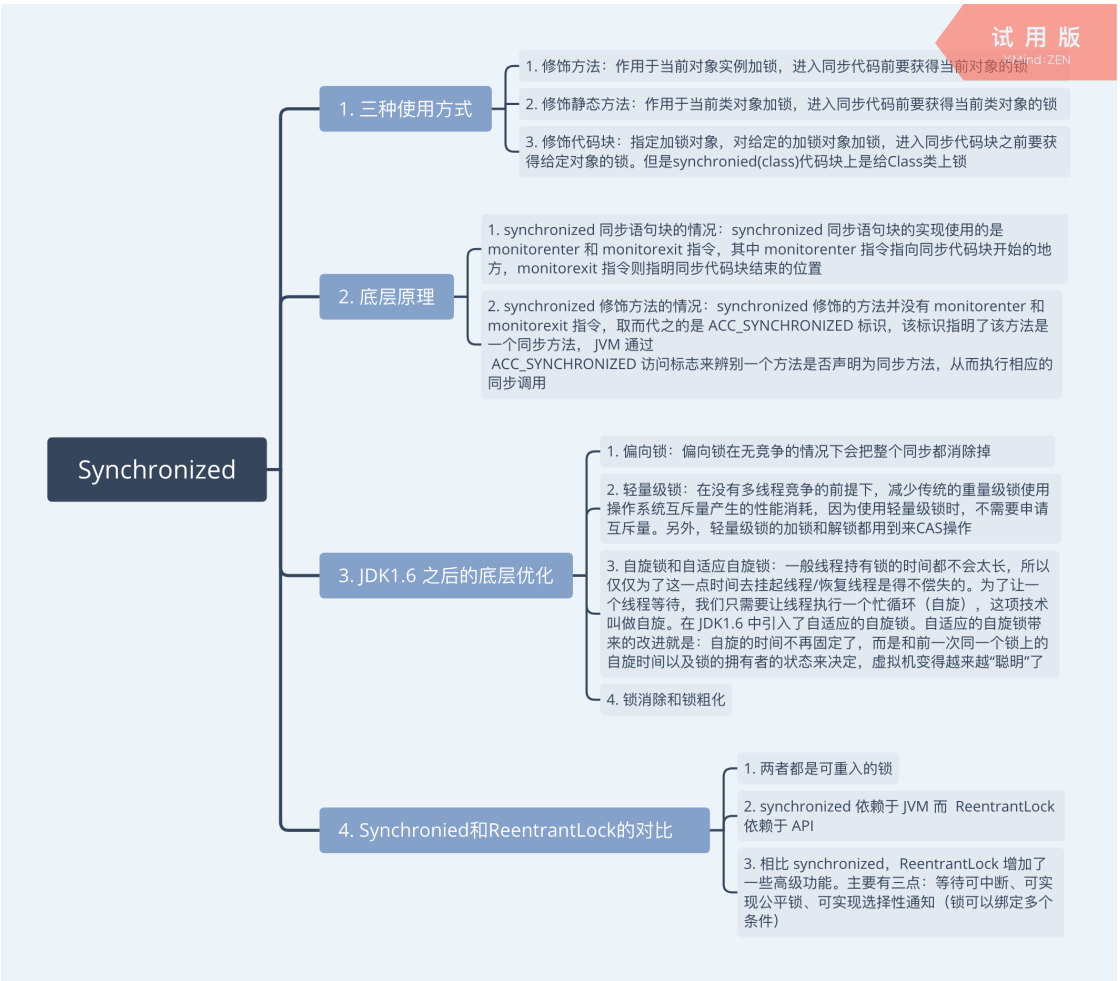
在一个线程对象上调用interrupt()方法，真正有影响的是 wait、join、sleep 方法，当然这 3 个方法包括它们的重载方法。请注意：上面这三个方法都会抛出 InterruptedException。

1、对于 wait 中的等待 notify、notifyAll 换新的线程，其实这个线程已经“暂停”执行，因为它正在某一对象的休息室中，这时如果它的中断状态被改变，那么它就会抛出异常。这个 InterruptedException 异常不是线程抛出的，而是 wait 方法，也就是对象的 wait 方法内部会不断检查在此对象上休息的线程的状态，如果发现哪个线程的状态被置为已中断，则会抛出 InterruptedException，意思就是这个线程不能再等待了，其意义就等同于唤醒它了，然后执行 catch 中的代码。

2、对于 sleep 中的线程，如果你调用了 Thread.sleep(一年)；现在你后悔了，想让它早 些醒过来，调用 interrupt() 方法就是唯一手段，只有改变它的中断状态，让它从 sleep 中将控制权转到处理异常的 catch 语句中，然后再由 catch 中的处理转换到正常的逻辑。同样，对于 join 中的线程你也可以这样处理。

二、synchronized

Synchronized 的详解



1. 说一说自己对于 synchronized 关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

2. 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗?

synchronized 关键字最主要的三种使用方式：

1.修饰实例方法： 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

2.修饰静态方法： 作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁；

3.修饰代码块： 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。 和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓冲功能！下面我已一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

双重校验锁实现对象单例（线程安全）：

```
public class Singleton {

        private volatile static Singleton uniqueInstance;

        private Singleton() {
```

```
        }

        public static Singleton getUniqueInstance() {
              // 先判断对象是否已经实例过，没有实例化过才进入加锁代码
                if (uniqueInstance == null) {
                        // 类对象加锁
                        synchronized (Singleton.class) {
                                if (uniqueInstance == null) {
                                        uniqueInstance = new Singleton();
                                }
                        }
                }
                return uniqueInstance;
        }
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

> 1.为 uniqueInstance 分配内存空间
>
> 2.初始化 uniqueInstance
>
> 3.将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出先问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

3. 讲一下 synchronized 关键字的底层原理

synchronized 关键字底层原理属于 JVM 层面。

1、synchronized 同步语句块的情况

```
public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
            System.out.println("synchronized 代码块");
```

```
            }
        }
    }
```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 javac SynchronizedDemo.java 命令生成编译后的 .class 文件，然后执行javap -c -s -v -l SynchronizedDemo.class。

```
  public void method();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter
         4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         7: ldc           #3                  // String Method 1 start
         9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        12: aload_1
        13: monitorexit
        14: goto          22
        17: astore_2
        18: aload_1
        19: monitorexit
        20: aload_2
        21: athrow
        22: return
      Exception table:
         from    to  target type
             4    14    17   any
            17    20    17   any
      LineNumberTable:
        line 5: 0
        line 6: 4
        line 7: 12
        line 8: 22
      StackMapTable: number_of_entries = 2
        frame_type = 255 /* full_frame */
          offset_delta = 17
          locals = [ class test/SynchronizedDemo, class java/lang/Object ]
          stack = [ class java/lang/Throwable ]
        frame_type = 250 /* chop */
          offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"
```

从上面我们可以看出：

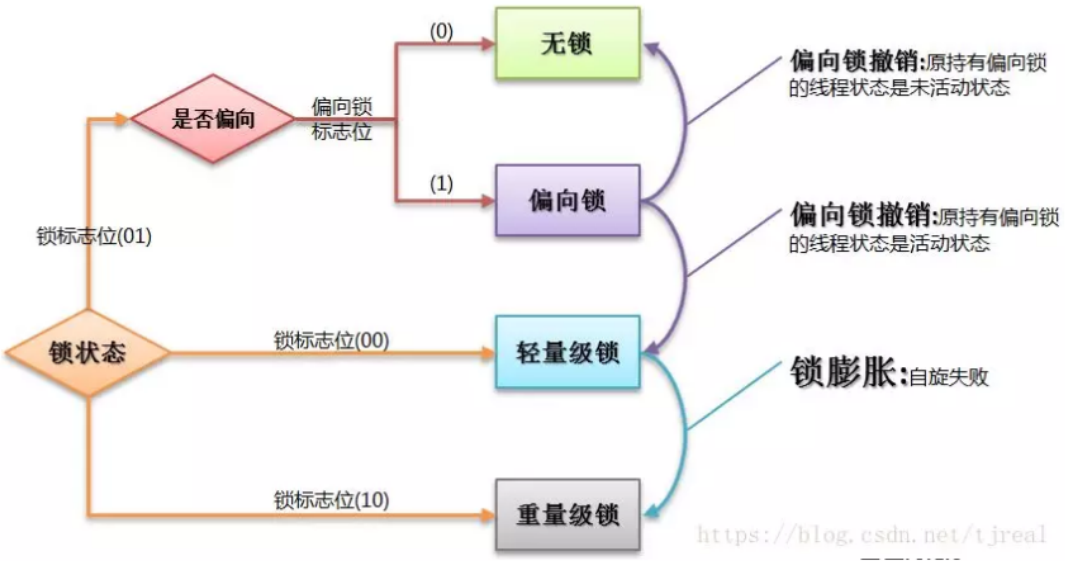
synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

2、synchronized 修饰方法的的情况

```
public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("synchronized 方法");
    }
}
```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

4. 说说 JDK1.6 之后的 synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗?



JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，它们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

4.1 偏向锁

引入偏向锁的目的和引入轻量级锁的目的很像，它们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

4.2 轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(JDK1.6 之后加入的)。轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了 CAS 操作。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生 CAS 操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

4.3 自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。 所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。

> 百度百科对自旋锁的解释：
>
> 何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过--XX:+UseSpinning 参数来开启。JDK1.6及1.6之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了。反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。自旋次数的默认值是 10 次，用户可以修改--XX:PreBlockSpin 来更改。

另外，在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：**自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。**

4.4 锁消除

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

4.5 锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小。只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

5. 谈谈 synchronized 和 ReenTrantLock 的区别?

> 简单回答（一般电话面试中可以这样简单回答，现场面试可以回答的详细些。视情况而定吧！）：
> synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock 比 synchronized 的扩展性体现在几点上：
>
> - ReentrantLock 可以对获取锁的等待时间进行设置，这样就避免了死锁；
> - ReentrantLock 可以获取各种锁的信息；
> - ReentrantLock 可以灵活地实现多路通知。
>
> 另外，二者的锁机制其实也是不一样的：ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁。synchronized 操作的应该是对象头中 mark word。

5.1 两者都是可重入锁

“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

5.2 synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

5.3 ReenTrantLock 比 synchronized 增加了一些高级功能相比 synchronized，ReenTrantLock 增加了一些高级功能

主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）；④ 性能已不是选择标准。

ReenTrantLock 提供了一种能够中断等待锁的线程的机制，通过 lock.lockInterruptibly( )来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

ReenTrantLock 可以指定是公平锁还是非公平锁。而 synchronized 只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。 ReenTrantLock 默认情况是非公平的，可以通过 ReenTrantLock 类的 ReentrantLock(boolean fair) 构造方法来制定是否是公平的。

synchronized 关键字与 wait() 和 notify/notifyAll() 方法相结合可以实现等待/通知机制，ReentrantLock 类当然也可以实现，但是需要借助于 Condition 接口与 newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知” ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择 ReenTrantLock 是一个不错的选择。

在 JDK1.6 之前，synchronized 的性能是比 ReenTrantLock 差很多。具体表示为：synchronized 关键字吞吐量随线程数的增加，下降得非常严重。而 ReenTrantLock 基本保持一个比较稳定的水平。我觉得这也侧面反映了， synchronized 关键字还有非常大的优化余地。后续的技术发展也证明了这一点，我们上面也讲了在 JDK1.6 之后 JVM 团队对 synchronized 关键字做了很多优化。JDK1.6 之后，synchronized 和 ReenTrantLock 的性能基本是持平了。所以网上那些说因为性能才选择 ReenTrantLock 的文章都是错的！JDK1.6之后，性能已经不是选择 synchronized 和 ReenTrantLock 的影响因素了！而且虚拟机在未来的性能改进中会更偏向于原生的 synchronized，所以还是提倡在 synchronized 能满足你的需求的情况下，优先考虑使用 synchronized 关键字来进行同步！优化后的 synchronized 和 ReenTrantLock 一样，在很多地方都是用到了 CAS 操作。

6. synchronized 和 volatile 的区别是什么?

1、volatile 本质是在告诉 Jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

2、volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。

3、volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。

4、volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

5、volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

7. 简单介绍下 volatile?

volatile 关键字是用来保证有序性和可见性的。这跟 Java 内存模型有关。

比如：我们所写的代码，不一定是按照我们自己书写的顺序来执行的，编译器会做重排序，CPU 也会做重排序的，这样的重排序是为了减少流水线的阻塞的，引起流水阻塞，比如数据相关性，提高 CPU 的执行效率。需要有一定的顺序和规则来保证，不然程序员自己写的代码都不知道对不对了，所以有 happens-before 规则，其中有条就是 volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；**有序性实现的是通过插入内存屏障来保证的。**

可见性：首先 Java 内存模型分为，主内存，工作内存。比如线程 A 从主内存把变量从主内存读到了自己的工作内存中，做了加 1 的操作，但是此时没有将 i 的最新值刷新会主内存中，线程 B 此时读到的还是 i 的旧值。加了 volatile 关键字的代码生成的汇编代码发现，会多出一个 lock 前缀指令。 Lock 指令对 Intel 平台的 CPU，早期是锁总线，这样代价太高了，后面提出了缓存一致性协议，MESI，来保证了多核之间数据不一致性问题。

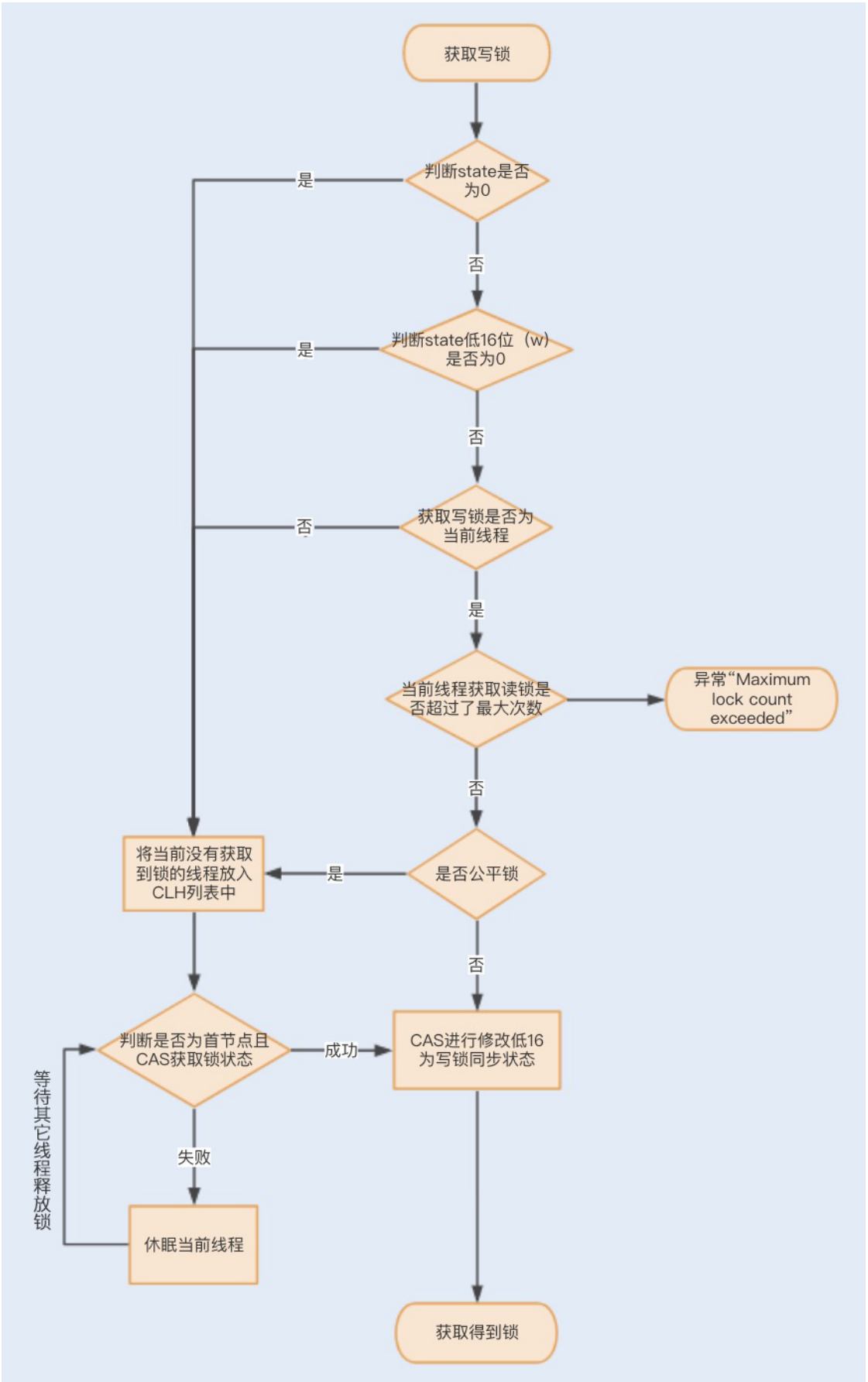
8. ReentrantLock 和 ReentrantReadWriteLock

ReentrantLock 是一个独占锁。同一时间只允许一个线程访问，而 ReentrantReadWriteLock 允许多个读线程同时访问，但是不允许写线程和读线程、写线程和写线程同时访问。读写锁内部维护了两个锁：**一个是用于读操作的 ReadLock，一个是用于写操作的 WriteLock。**

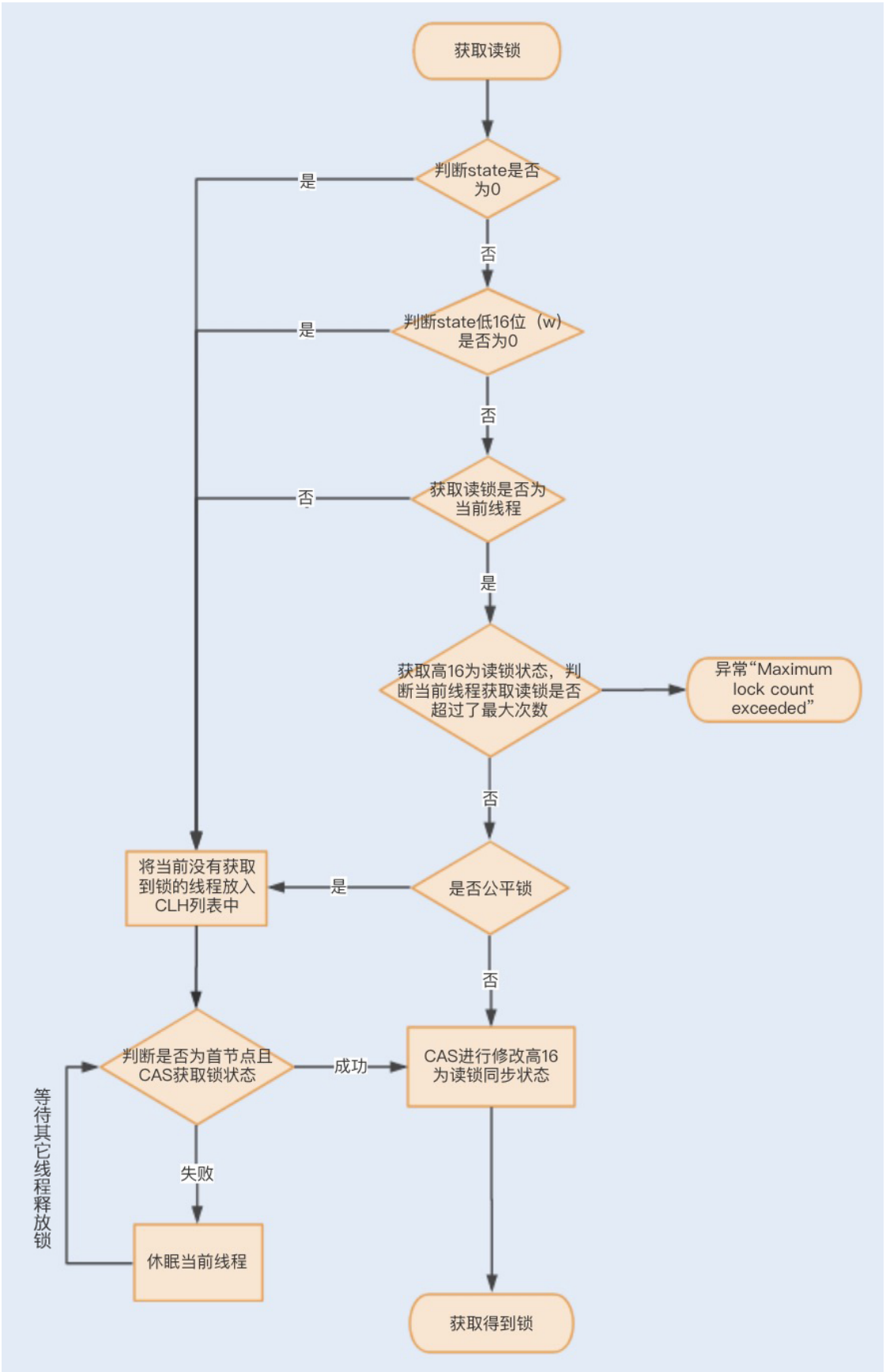
读写锁 ReentrantReadWriteLock 可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。

ReentrantReadWriteLock 基于 AQS 实现，它的自定义同步器（继承 AQS）需要在同步状态 state 上维护多个读线程和一个写线程，该状态的设计成为实现读写锁的关键。ReentrantReadWriteLock 很好的利用了高低位。来实现一个整型控制两种状态的功能，读写锁将变量切分成了两个部分，高 16 位表示读，低 16 位表示写。

- **获取写锁的过程：**


获取写锁
判断state是否为0
是
否
判断state低16位（w）是否为0
是
否
获取写锁是否为当前线程
是
否
当前线程获取读锁是否超过了最大次数
异常“Maximum lock count exceeded”
否
将当前没有获取到锁的线程放入CLH列表中
是
是否公平锁
否
判断是否为首节点且CAS获取锁状态
成功
CAS进行修改低16为写锁同步状态
失败
等待其它线程释放锁
休眠当前线程
获取得到锁

- **获取读锁的过程：**



ReentrantReadWriteLock 的优化：StampedLock：实现了乐观读锁、悲观读锁、写锁。解决 读取很多、写入很少时，ReentrantReadWriteLock 会使写入线程遭遇饥饿问题。相比于 ReentrantReadWriteLock ， StampedLock 获取读锁只是使用与或操作进行

检验，不涉及 CAS 操作，即使第一次乐观锁获取失败，也会马上升级至悲观锁，这样就可以避免一直进行 CAS 操作带来的 CPU 占用性能问题，因此 StampedLock 的效率更高。

- **ReentrantReadWriteLock 特点：**

1、写锁可以降级为读锁，但是读锁不能升级为写锁；

2、不管是 ReadLock 还是 WriteLock 都支持 Interrupt，语义与 ReentrantLock 一致；

3、WriteLock 支持 Condition 并且与 ReentrantLock 语义一致，而 ReadLock 则不能使用 Condition，否则抛出 UnsupportedOperationException 异常；

4、默认构造方法为非公平模式 ，开发者也可以通过指定 fair 为 true 设置为公平模式 。

- **升降级**

1、读锁里面加写锁，会导致死锁；

2、写锁里面是可以加读锁的，这就是锁的降级。

三、乐观锁和悲观锁

乐观锁和悲观锁详解

1. 乐观锁和悲观锁的基本概念

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

1.1 悲观锁：

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。**传统的关系型数据库里边就用到了很多这种锁机制，比如 行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java 中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。**

1.2 乐观锁：

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中

java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

1.3 两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，**像乐观锁适用于写比较少的情况下（多读场景）**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 retry，这样反倒是降低了性能，所以一般**多写的场景下用悲观锁就比较合适。**

2. 乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或者 CAS 算法实现。

2.1 版本号机制

一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加 1。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

2.2 CAS 算法

即 compare and swap（比较与交换），是一种有名的**无锁算法。**无锁编程，即**不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）**。CAS 算法涉及到三个操作数：

1、需要读写的内存值 V

2、进行比较的值 A

3、拟写入的新值 B

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

3. 乐观锁的缺点

3.1 ABA 问题：

如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。这个问题被称为 CAS 操作的 "ABA" 问题。

JDK 1.5 以后的 **AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就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。**

3.2 循环时间长开销大

自旋 CAS（也就是不成功就一直循环执行直到成功）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。

3.3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。 但是从 JDK 1.5 开始，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。所以我们可以使用锁或者利用 AtomicReference 类把多个共享变量合并成一个共享变量来操作。

4. CAS 和 synchronized 的使用场景

简单的来说 CAS 适用于写比较少的情况下（多读场景，冲突一般较少），synchronized 适用于写比较多的情况下（多写场景，冲突一般较多）

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

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

补充： Java 并发编程这个领域中 synchronized 关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在 JavaSE 1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。

synchronized 的底层实现主要依靠 Lock-Free 的队列，基本思路是：自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于 CAS。

四、Atomic 原子类

1. 原子类的基本介绍

这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下。根据操作的数据类型，可以将 JUC 包中的原子类分为 4 类：

1.1 基本类型

使用原子的方式更新基本类型：

- AtomicInteger：整型原子类
- AtomicLong：长整型原子类
- AtomicBoolean ： 布尔型原子类

1.2 数组类型

使用原子的方式更新数组里的某个元素：

- AtomicIntegerArray：整型数组原子类
- AtomicLongArray：长整型数组原子类
- AtomicReferenceArray ： 引用类型数组原子类

1.3 引用类型

- AtomicReference：引用类型原子类
- AtomicStampedReference：原子更新引用类型里的字段原子类
- AtomicMarkableReference ： 原子更新带有标记位的引用类型

1.4 对象的属性修改类型

- AtomicIntegerFieldUpdater：原子更新整型字段的更新器
- AtomicLongFieldUpdater：原子更新长整型字段的更新器

- AtomicStampedReference ： 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

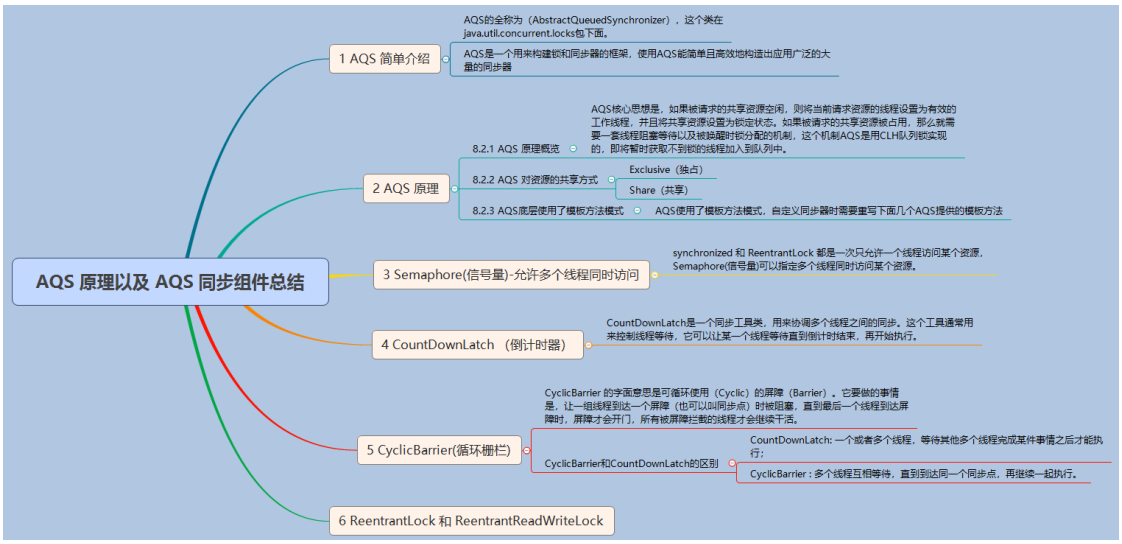
2、说一下 atomic 的原理?

Atomic 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

Atomic 系列的类中的核心方法都会调用 unsafe 类中的几个本地方法。我们需要先知道一个东西就是 Unsafe 类，全名为：sun.misc.Unsafe，这个类包含了大量的对 C 代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过 unsafe 分配内存的时候，如果自己指定某些区域可能会导致一些类似 C++ 一样的指针越界到其他进程的问题。

五、AQS：同步器【重要】

AQS



1. AQS 的简单介绍

AQS 的全称为（AbstractQueuedSynchronizer），这个类在 java.util.concurrent.locks 包下面。

AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock，Semaphore，其他的诸如

ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等皆是基于 AQS 的。当然，我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

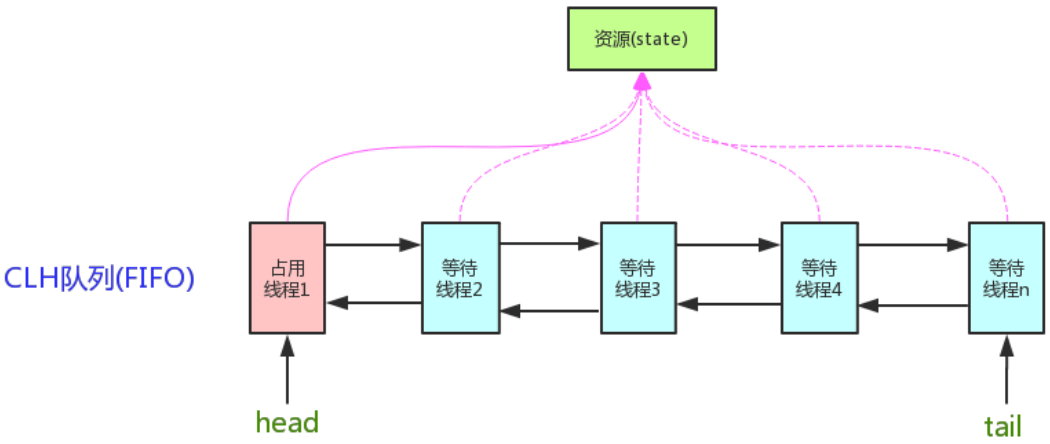
2. AQS 原理

2.1 AQS 原理概述

AQS核心思想是：**如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。**

> CLH(Craig, Landin, and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

看个 AQS(AbstractQueuedSynchronizer) 原理图：



AQS 使用一个 int 成员变量 (state) 来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state; // 共享变量，使用 volatile 修饰保证线程可见性
```

状态信息通过 protected 类型的 getState、setState、compareAndSetState 进行操作

```
// 返回同步状态的当前值
protected final int getState() {
        return state;
}
 // 设置同步状态的值
protected final void setState(int newState) {
```

```
        state = newState;
}
// 原子地（CAS操作）将同步状态值设置为给定值 update 如果当前同步状态的值等于
expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

2.2 AQS 对资源的共享模式

- **2.2.1 AQS定义两种资源共享方式：**

1. Exclusive（独占）：只有一个线程能执行，如 ReentrantLock。又可分为公平锁和非公平锁：

> **公平锁**：按照线程在队列中的排队顺序，先到者先拿到锁；
>
> **非公平锁**：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的。

2. Share（共享）：多个线程可同时执行，如 Semaphore/CountDownLatch。Semaphore、CountDownLatch、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为 ReentrantReadWriteLock 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在上层已经帮我们实现好了。**

- **2.2.2 AQS底层使用了模板方法模式**

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

使用者继承 AbstractQueuedSynchronizer 并重写指定的方法。（这些重写方法很简单，无非是**对于共享资源 state 的获取和释放**）将 AQS 组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用，下面简单的给大家介绍一下模板方法模式，模板方法模式是一个很容易理解的设计模式之一。

> 模板方法模式是基于”继承“的，主要是为了在不改变模板结构的前提下在子类中重新定义模板中的内容以实现复用代码。举个很简单的例子假如我们要去一个地方的步骤是：购票 buyTicket()->安检 securityCheck()->乘坐某某工具回家 ride()->到达目的地 arrive()。我们可能乘坐不同的交通工具回家比如飞机或者火车，所以除

> 了 ride() 方法，其他方法的实现几乎相同。我们可以定义一个包含了这些方法的抽象类，然后用户根据自己的需要继承该抽象类然后修改 ride() 方法。

AQS 使用了模板方法模式，自定义同步器时需要重写下面几个 AQS 提供的模板方法：

```
1、isHeldExclusively() // 该线程是否正在独占资源。只有用到 condition 才需要去实现它。
2、tryAcquire(int) // 独占方式。尝试获取资源，成功则返回true，失败则返回false。
3、tryRelease(int) // 独占方式。尝试释放资源，成功则返回true，失败则返回false。
4、tryAcquireShared(int) // 共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
5、tryReleaseShared(int) // 共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

默认情况下，每个方法都抛出 UnsupportedOperationException。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS 类中的其他方法都是 final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以 ReentrantLock 为例，state 初始化为 0，表示未锁定状态。A 线程 lock() 时，会调用 tryAcquire() 独占该锁并将 state+1。此后，其他线程再 tryAcquire() 时就会失败，直到A 线程 unlock() 到 state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证 state 是能回到零态的。

再以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 countDown() 一次，state 会 CAS(Compare and Swap) 减 1。等到所有子线程都执行完后(即 state=0)，会 unpark() 主调用线程，然后主调用线程就会从 await() 函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，它们也只需实现 tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock。

3. Semaphore（信号量）：允许多个线程同时访问

synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore (信号量)可以指定多个线程同时访问某个资源。

执行 acquire 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 release 方法增加一个许可证，这可能会释放一个阻塞的 acquire 方法。然而，其实并没有

实际的许可证这个对象，Semaphore 只是维持了一个可获得许可证的数量。 **Semaphore 经常用于限制获取某种资源的线程数量。**

当然一次也可以一次拿取和释放多个许可证，不过一般没有必要这样做。

除了 acquire方法（阻塞）之外，另一个比较常用的与之对应的方法是 tryAcquire 方法，该方法如果获取不到许可就立即返回 false。

Semaphore 有两种模式，公平模式和非公平模式。

> 公平模式： 调用 acquire 的顺序就是获取许可证的顺序，遵循 FIFO；
>
> 非公平模式： 抢占式的。

Semaphore 对应的两个构造方法如下：

```
public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }

public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
    }
```

这两个构造方法，都必须提供许可的数量，第二个构造方法可以指定是公平模式还是非公平模式，默认非公平模式。

4. CountDownLatch （倒计时器）

CountDownLatch 是一个同步工具类，它允许一个或多个线程一直等待，直到其他线程的操作执行完后再执行。在 Java 并发中，CountDownLatch 的概念是一个常见的面试题，所以一定要确保你很好的理解了它。

4.1 CountDownLatch 的三种典型用法

1、某一线程在开始运行前等待 n 个线程执行完毕。 将 CountDownLatch 的计数器初始化为 n： new CountDownLatch(n)，每当一个任务线程执行完毕，就将计数器减 1：countDownLatch.countDown()，当计数器的值变为 0 时，在 CountDownLatch 上 await() 的线程就会被唤醒。一个典型应用场景就是：**启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。**

2、实现多个线程开始执行任务的最大并行性。 注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 CountDownLatch 对象，将其计数器初始化为 1：new CountDownLatch(1)，多个线程在开始执行任务前首先

countDownLatch.await()，当主线程调用 countDown() 时，计数器变为0，多个线程同时被唤醒。

3、死锁检测：一个非常方便的使用场景是，你可以使用 n 个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

4.2 CountDownLatch 的不足

CountDownLatch 是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当 CountDownLatch 使用完毕后，它不能再次被使用。

4.3 CountDownLatch 相常见面试题：

- 解释一下CountDownLatch概念?
- CountDownLatch 和 CyclicBarrier 的不同之处?
- 给出一些 CountDownLatch 使用的例子?
- CountDownLatch 类中主要的方法?

5. CyclicBarrier(循环栅栏)

CyclicBarrier 和 CountDownLatch 非常类似，它也是可以实现线程间等待的技术，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，**让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。**CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

5.1 CyclicBarrier 的应用场景

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 Shee t保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

5.2 CyclicBarrier 和 CountDownLatch 的区别

CountDownLatch 是计数器，只能使用一次，而CyclicBarrier 的计数器提供 reset 功能，可以多次使用。 但是我不那么认为它们之间的区别仅仅就是这么简单的一点。我们

来从 jdk 作者设计的目的来看，javadoc 是这么描述它们的：

> CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.(CountDownLatch: 一个或者多个线程，等待其他多个线程完成某件事情之后才能执行；)
>
> CyclicBarrier : A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.(CyclicBarrier : 多个线程互相等待，直到到达同一个同步点，再继续一起执行。)

对于 CountDownLatch 来说，重点是“一个线程（多个线程）等待”，而其他的 N 个线程在完成“某件事情”之后，可以终止，也可以等待。而对于 CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有的线程都必须等待。

CountDownLatch 是计数器，线程完成一个记录一个，只不过计数不是递增而是递减，而 CyclicBarrier 更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为0时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为0时，无法重置	计数达到指定值时，计数置为0重新开始
调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响	调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

六、线程池

1. 为什么要用线程池?

线程池提供了一种限制和管理资源（包括执行一个任务）的方式。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

使用线程池的好处：

> 1、**降低资源消耗**：通过重复利用已创建的线程降低线程创建和销毁造成的消耗;
>
> 2、**提高响应速度**：当任务到达时，任务可以不需要的等到线程创建就能立即执行;
>
> 3、**提高线程的可管理性**：线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2. 实现 Runnable 接口和 Callable 接口的区别?

如果想让线程池执行任务的话需要实现的 Runnable 接口或 Callable 接口。Runnable 接口或 Callable 接口实现类都可以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行。**两者的区别在于 Runnable 接口不会返回结果但是 Callable 接口可以返回结果。**

> 备注： 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。（Executors.callable（Runnable task）或 Executors.callable（Runnable task，Object resule））。

3. 执行 execute() 方法和 submit() 方法的区别是什么呢?

1、**execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否;**

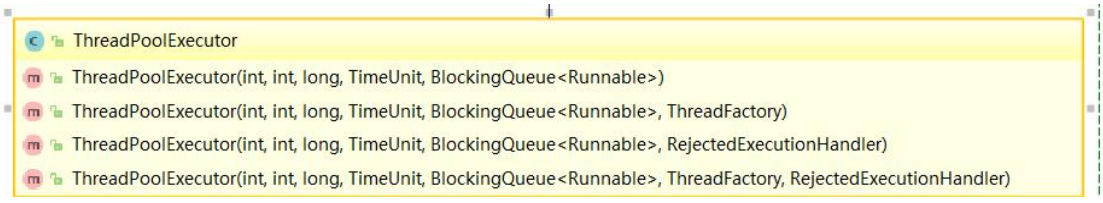
2、**submit()方法用于提交需要返回值的任务。**线程池会返回一个 future 类型的对象，通过这个 future 对象可以判断任务是否执行成功，并且可以通过 future 的 get() 方法来获取返回值，get() 方法会阻塞当前线程直到任务完成，而使用 get(long timeout，TimeUnit unit) 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

4. 如何创建线程池?

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

> Executors 返回线程池对象的弊端如下：
> FixedThreadPool 和 SingleThreadExecutor： 允许请求的队列长度为 Integer.MAX_VALUE，可能堆积大量的请求，从而导致 OOM。
> CachedThreadPool 和 ScheduledThreadPool： 允许创建的线程数量为 Integer.MAX_VALUE ，可能会创建大量线程，从而导致 OOM。

方式一：通过构造方法实现



方式二：通过 Executor 框架的工具类 Executors 来实现

我们可以创建三种类型的 ThreadPoolExecutor：

1. FixedThreadPool： 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

2. SingleThreadExecutor：方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先进先出的顺序执行队列中的任务。

3. CachedThreadPool： 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

5. 线程池的参数

1、**corePoolSize（线程池的基本大小）**：当提交一个任务到线程池时，如果当前 poolSize < corePoolSize 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。

2、**maximumPoolSize（线程池最大数量）**：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。

3、**keepAliveTime（线程活动保持时间）**：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，可以调大时间，提高线程的利用率。

4、**TimeUnit（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）和纳秒（NANOSECONDS，千分之一微秒）。

5、**workQueue（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，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 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。
>
> 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。

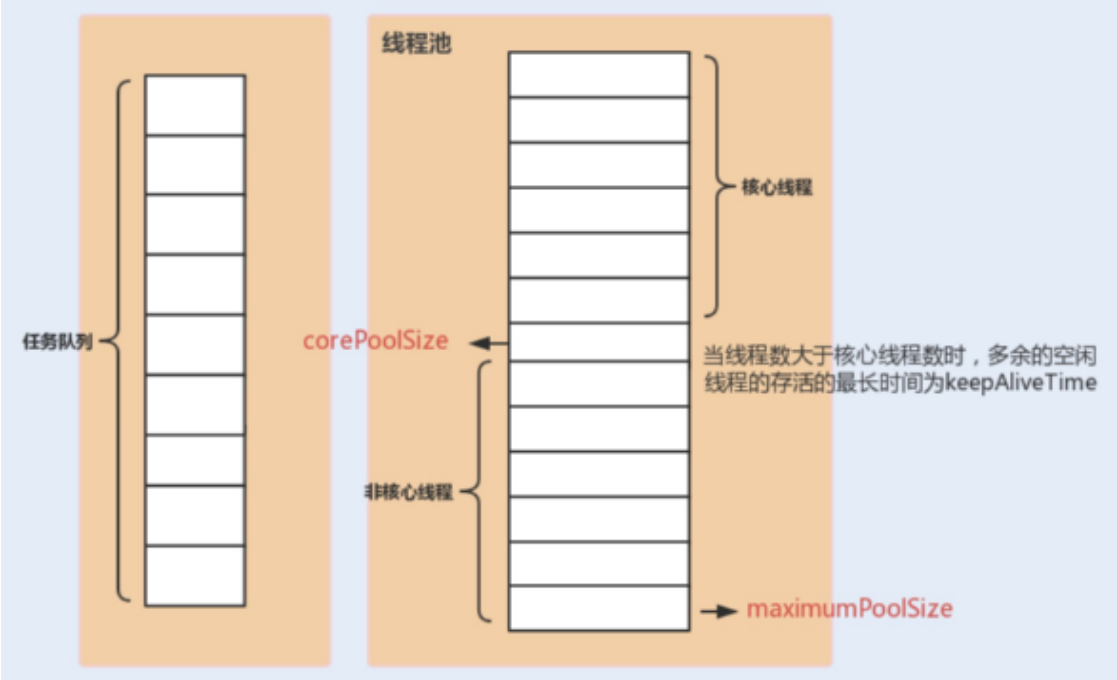
6、**threadFactory**：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。使用开源框架guava提供的ThreadFactoryBuilder可以快速给线程池里的线程设置有意义的名字，代码如下。

```
new ThreadFactoryBuilder().setNameFormat("XX-task-%d").build();
```

7、**RejectExecutionHandler（饱和策略）**：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。在 JDK 1.5 中 Java 线程池框架提供了以下 4 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存储不能处理的任务。



6、线程池中一般设置多少线程，为什么?

好文章，有案例分析：https://blog.csdn.net/zhanht/article/details/79513134

主要考虑下面几个方面：

1、线程池中线程执行任务的性质：

计算密集型的任务比较占 cpu，所以一般线程数设置的大小 等于或者略微大于 cpu 的核数；但 IO 型任务主要时间消耗在 IO 等待上，cpu 压力并不大，所以线程数一般设置较大，例如 多线程访问数据库，数据库有128个表，可能就直接考虑使用 128 个线程。

2、CPU 使用率：

当线程数设置较大时，会有如下几个问题：第一，线程的初始化，切换，销毁等操作会消耗不小的 cpu 资源，使得 cpu 利用率一直维持在较高水平。第二，线程数较大时，任务会短时间迅速执行，任务的集中执行也会给 cpu 造成较大的压力。第三， 任务的集中支持，会让 cpu 的使用率呈现锯齿状，即短时间内 cpu 飙高，然后迅速下降至闲置状态，cpu 使用的不合理，应该减小线程数，让任务在队列等待，使得 cpu 的使用率应该持续稳定在一个合理，平均的数值范围。所以 cpu 在够用时，不宜过大，不是越大越好。可以通过上线后，**观察机器的 cpu 使用率和 cpu 负载两个参数来判断线程数是否合理。**可通过命令查看 cpu 使用率是否主要花在线程切换上。cpu 负载是正在执行的线程和等待执行的线程之和，注意这个等待不是指线程的 wait 那种等待，而是指线程处于 running 状态但是还没有被 cpu 调度的等待，负载较高，意味着 cpu 竞争激烈，进而说明线程设置较大，在抢 cpu 资源。负载的值一般约等于 cpu 核数是比较合理的数值。

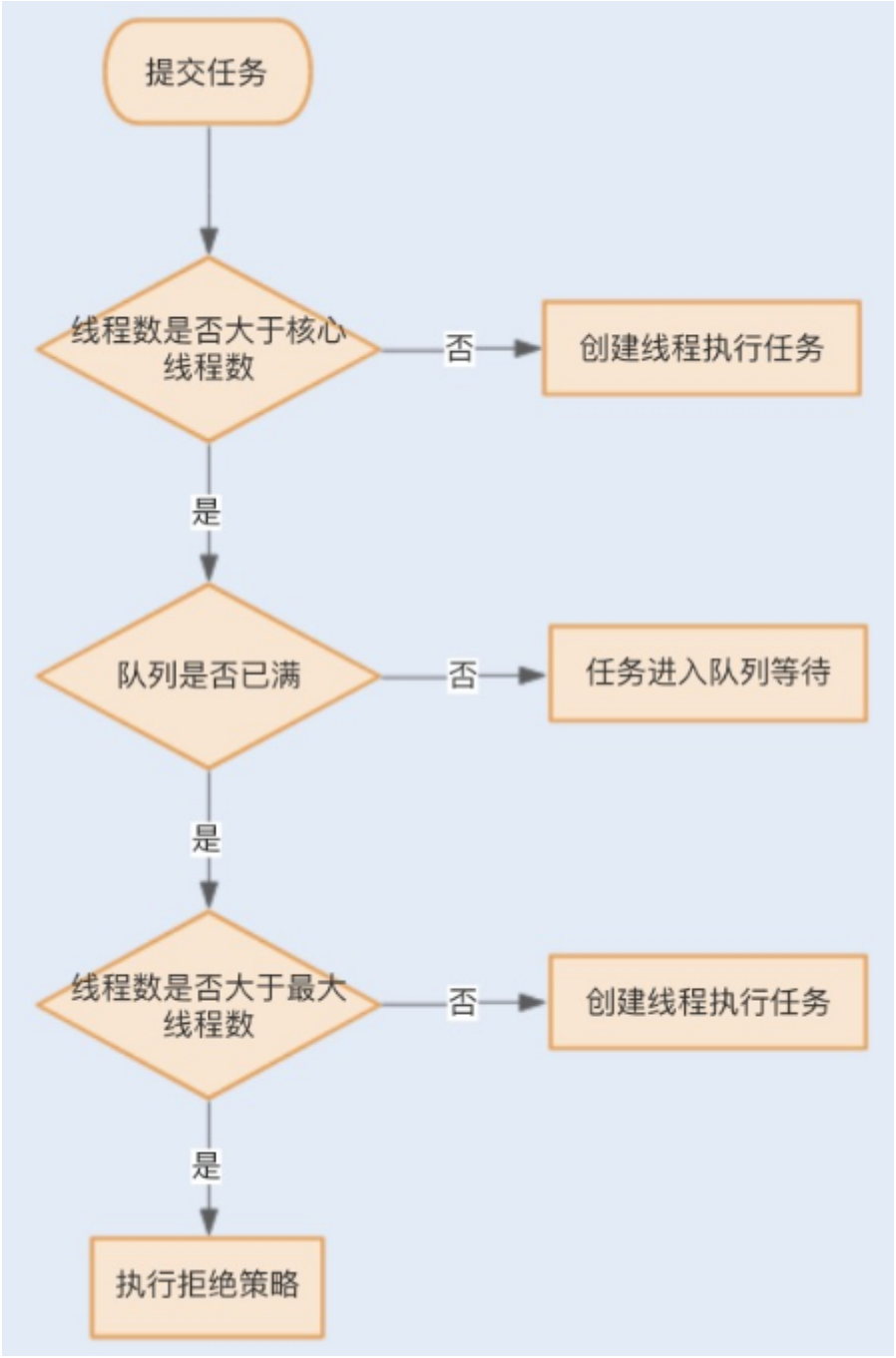
3、内存使用率：

线程数过多和队列的大小都会影响此项数据，队列的大小应该通过前期计算线程池任务的条数，来合理的设置队列的大小，不宜过小，让其不会溢出，因为溢出会走拒绝策略，多少会影响性能，也会增加复杂度，因为你得好好考量你的拒绝策略的选择，拒绝策略包括 AbortPolicy(抛异常)， CallerRunsPolicy(主线程执行) 和 DiscardPolicy(丢弃)。也不宜多大，过大用不上，还会消耗较大的内存。

4、下游系统抗并发能力：

多线程给下游系统造成的并发等于你设置的线程数，例如：如果是多线程访问数据库，你就考虑数据库的连接池大小设置，数据库并发太多影响其 QPS，会把数据库打挂等问题。 如果访问的是下游系统的接口，你就得考虑下游系统是否能抗的住这么多并发量，不能把下游系统打挂了。

7、线程池线程的分配流程

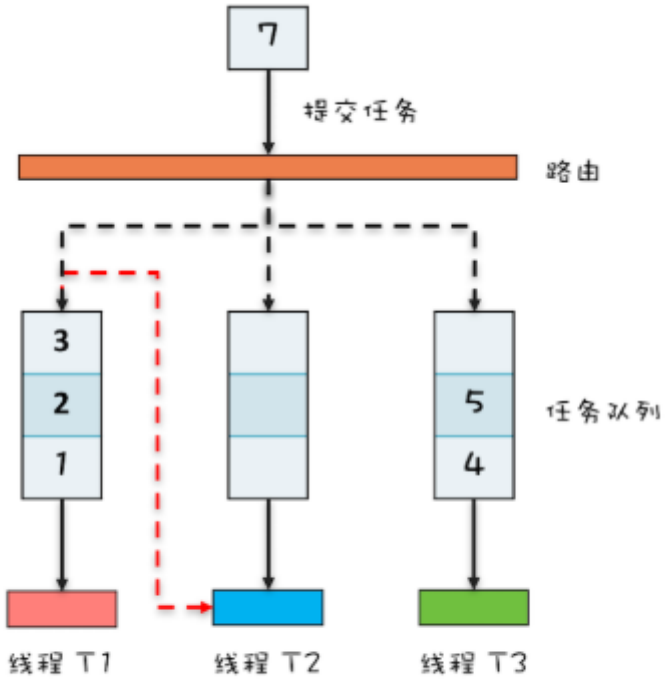


七、Fork/Join 并行计算框架

Fork/Join 并行计算框架主要解决的是分治任务。分治的核心思想是“分而治之”：将一个大的任务拆分成小的子任务的结果聚合起来从而得到最终结果。

Fork/Join 并行计算框架的核心组件是 ForkJoinPool。ForkJoinPool 支持任务窃取机制，能够让所有的线程的工作量基本均衡，不会出现有的线程很忙，而有的线程很闲的情况，所以性能很好。

ForkJoinPool 中的任务队列采用的是双端队列，工作线程正常获取任务和“窃取任务”分别是从任务队列不同的端消费，这样能避免很多不必要的数据竞争。



ForkJoinPool 工作原理图

八、并发容器

1. JDK 提供的并发容器概述

JDK 提供的这些容器大部分在 java.util.concurrent 包中。

ConcurrentHashMap: 线程安全的 HashMap；

CopyOnWriteArrayList: 线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector；

ConcurrentLinkedQueue: 高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列；

BlockingQueue: 这是一个接口，JDK 内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道；

ConcurrentSkipListMap: 跳表的实现。这是一个 Map，使用跳表的数据结构进行快速查找。

2. ConcurrentHashMap

我们知道 HashMap 不是线程安全的，在并发场景下如果要保证一种可行的方式是使用 Collections.synchronizedMap() 方法来包装我们的 HashMap。但这是通过使用一个

全局的锁来同步不同线程间的并发访问，因此会带来不可忽视的性能问题。

所以就有了 HashMap 的线程安全版本 —— ConcurrentHashMap 的诞生。在 ConcurrentHashMap 中，无论是读操作还是写操作都能保证很高的性能：在进行读操作时(几乎)不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。

3. CopyOnWriteArrayList

3.1 CopyOnWriteArrayList 简介

```
public class CopyOnWriteArrayList<E> extends Object
        implements List<E>, RandomAccess, Cloneable, Serializable
```

在很多应用场景中，读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据，毕竟读取操作是安全的。

这和 ReentrantReadWriteLock 读写锁的思想非常类似，也就是读读共享、写写互斥、读写互斥、写读互斥。JDK 中提供了 CopyOnWriteArrayList 类比相比于在读写锁的思想又更进一步。为了将读取的性能发挥到极致，CopyOnWriteArrayList 读取是完全不用加锁的，并且更厉害的是：写入也不会阻塞读取操作。只有写入和写入之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。

3.2 CopyOnWriteArrayList 是如何做到的?

CopyOnWriteArrayList 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，我们并不需要修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

从 CopyOnWriteArrayList 的名字就能看出 CopyOnWriteArrayList 是满足 CopyOnWrite 的 ArrayList，所谓 CopyOnWrite 也就是说：在计算机，如果你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后呢，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉了。

CopyOnWriteArrayList 读取操作没有任何同步控制和锁操作，理由就是内部数组 array 不会发生修改，只会被另外一个 array 替换，因此可以保证数据安全。

CopyOnWriteArrayList 写入操作 add() 方法在添加集合的时候加了锁，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

4. ConcurrentLinkedQueue

Java 提供的线程安全的 Queue 可以分为阻塞队列和非阻塞队列，其中阻塞队列的典型例子是 BlockingQueue，非阻塞队列的典型例子是 ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。 阻塞队列可以通过加锁来实现，非阻塞队列可以通过 CAS 操作实现。

从名字可以看出，ConcurrentLinkedQueue 这个队列使用链表作为其数据结构. ConcurrentLinkedQueue 应该算是在高并发环境中性能最好的队列了。它之所有能有很好的性能，是因为其内部复杂的实现。

ConcurrentLinkedQueue 内部代码我们就不分析了，大家知道**ConcurrentLinkedQueue 主要使用 CAS 非阻塞算法来实现线程安全**就好了。

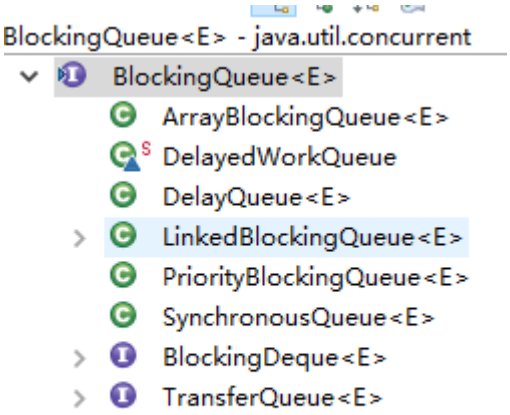
ConcurrentLinkedQueue 适合在对性能要求相对较高，同时对队列的读写存在多个线程同时进行的场景，即如果对队列加锁的成本较高则适合使用无锁的 ConcurrentLinkedQueue 来替代。

5. BlockingQueue

5.1 BlockingQueue 简单介绍

上面我们已经提到了 ConcurrentLinkedQueue 作为高性能的非阻塞队列。下面我们要讲到的是阻塞队列 —— BlockingQueue。阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是 **BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。**

BlockingQueue 是一个接口，继承自 Queue，所以其实现类也可以作为 Queue 的实现来使用，而 Queue 又继承自 Collection 接口。下面是 BlockingQueue 的相关实现类：



下面主要介绍一下：ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue，这3个 BlockingQueue 的实现类。

5.2 ArrayBlockingQueue

ArrayBlockingQueue 是 BlockingQueue 接口的有界队列实现类，底层采用数组来实现。ArrayBlockingQueue 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。当队列容量满时，尝试将元素放入队列将导致操作阻塞；尝试从一个空队列中取一个元素也会同样阻塞。

ArrayBlockingQueue 默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到 ArrayBlockingQueue。而非公平性则是指访问 ArrayBlockingQueue 的顺序不是遵守严格的时间顺序，有可能存在，当 ArrayBlockingQueue 可以被访问时，长时间阻塞的线程依然无法访问到 ArrayBlockingQueue。如果保证公平性，通常会降低吞吐量。如果需要获得公平性的 ArrayBlockingQueue，可采用如下代码：

```
private static ArrayBlockingQueue<Integer]]> blockingQueue = new
ArrayBlockingQueue<Integer]]>(10,true);
```

5.3 LinkedBlockingQueue

LinkedBlockingQueue 底层基于单向链表实现的阻塞队列，可以当做无界队列也可以当做有界队列来使用，同样满足 FIFO 的特性，与 ArrayBlockingQueue 相比起来具有更高的吞吐量，为了防止 LinkedBlockingQueue 容量迅速增，损耗大量内存。通常在创建 LinkedBlockingQueue 对象时，会指定其大小，如果未指定，容量等于 Integer.MAX_VALUE。

相关构造方法:

```
  // 某种意义上的无界队列
    public LinkedBlockingQueue() {
            this(Integer.MAX_VALUE);
    }

    // 有界队列
    public LinkedBlockingQueue(int capacity) {
            if (capacity <= 0) throw new IllegalArgumentException();
            this.capacity = capacity;
            last = head = new Node<E]]>(null);
    }
```

5.4 PriorityBlockingQueue

PriorityBlockingQueue 是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现 compareTo() 方法来指定元素排序规则，或者初始化时通过构造器参数 Comparator 来指定排序规则。

PriorityBlockingQueue 并发控制采用的是 ReentrantLock，队列为无界队列（ArrayBlockingQueue 是有界队列，LinkedBlockingQueue 也可以通过在构造函数中传入 capacity 指定队列最大的容量，但是 PriorityBlockingQueue 只能指定初始的队列大小，后面插入元素的时候，如果空间不够的话会自动扩容）。

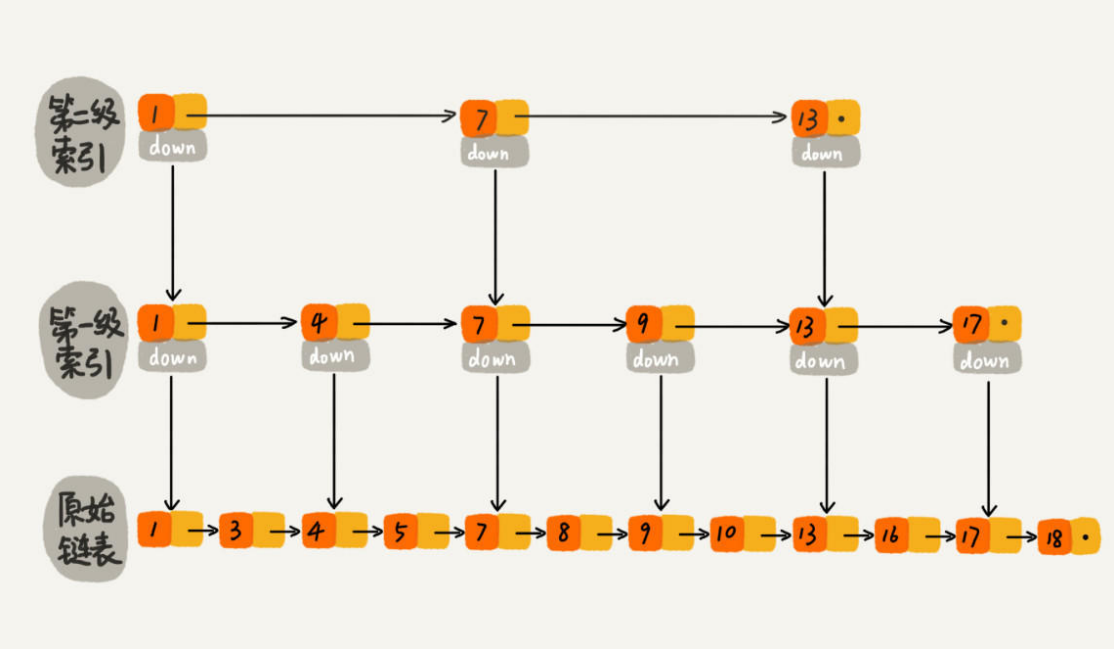
简单地说，它就是 PriorityQueue 的线程安全版本。不可以插入 null 值，同时，插入队列的对象必须是可比较大小的（comparable），否则报 ClassCastException 异常。它的插入操作 put 方法不会 block，因为它是无界队列（take 方法在队列为空的时候会阻塞）。

6. ConcurrentSkipListMap

为了引出 ConcurrentSkipListMap，先带着大家简单理解一下跳表。对于一个单链表，即使链表是有序的，如果我们想要在其中查找某个数据，也只能从头到尾遍历链表，这样效率自然就会很低，跳表就不一样了。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。

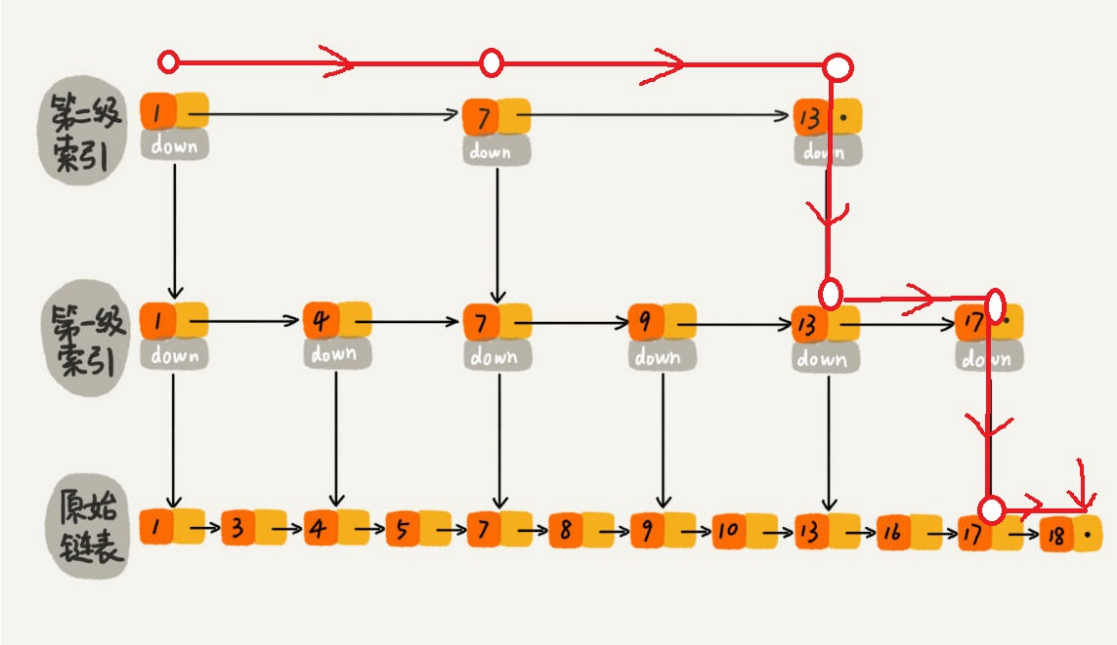
但一个重要的区别是：**对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。**这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是 O(logn) 所以在并发数据结构中，JDK 使用跳表来实现一个 Map。

跳表的本质是同时维护了多个链表，并且链表是分层的。



最低层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集。

跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。如上图所示，在跳表中查找元素18。



查找 18 的时候原来需要遍历 18 次，现在只需要 7 次即可。针对链表长度比较大的时候，构建索引查找效率的提升就会非常明显。

从上面很容易看出，跳表是一种利用空间换时间的算法。

使用跳表实现 Map 和使用哈希算法实现 Map 的另外一个不同之处是：哈希并不会保存元素的顺序，而跳表内所有的元素都是排序的。因此在对跳表进行遍历时，你会得到一个有序的结果。所以，如果你的应用需要有序性，那么跳表就是你不二的选择。JDK 中实现这一数据结构的类是：ConcurrentSkipListMap。

九、其他

1. ThradLocal

参考：https://mp.weixin.qq.com/s/myMkqMRHlPRLlHvShu8f9g

1、Java 的 Web 项目大部分都是基于 Tomcat。每次访问都是一个新的线程，**每一个线程都独享一个 ThreadLocal**，我们可以在接收请求的时候 set 特定内容，在需要的时候 get 这个值。

2、**ThreadLocal 提供 get 和 set 方法，为每一个使用这个变量的线程都保存有一份独立的副本。**

```
public T get() { }
public void set(T value) { }
public void remove() { }
protected T initialValue() { }
```

（1）get() 方法是用来获取 ThreadLocal 在当前线程中保存的变量副本；

（2）set() 用来设置当前线程中变量的副本；

（3）remove() 用来移除当前线程中变量的副本；

（4）initialValue() 是一个 protected 方法，一般是用来在使用时进行重写的，如果在没有 set 的时候就调用 get，会调用 initialValue方法初始化内容。

- **使用场景【面试可能会问】**

在调用 API 接口的时候传递了一些公共参数，这些公共参数携带了一些设备信息（是安卓还是 ios），服务端接口根据不同的信息组装不同的格式数据返回给客户端。假定服务器端需要通过设备类型(device)来下发下载地址，当然接口也有同样的其他逻辑，我们只要在返回数据的时候判断好是什么类型的客户端就好了。

上面这种场景就可以将传进来的参数 device 设置到 ThreadLocal 中。用的时候取出来就行。避免了参数的层层传递。