Java 常用数据结构

Ning Ding

February 2024

1 并发编程的发展和价值

并发就是同一时刻，只有一个执行，但是一个时间段内，两个线程都执行了。并发的实现依赖于CPU切换线程，因为切换的时间特别短，所以基本对于用户是无感知的。

高并发可以通过分布式技术去解决，将并发流量分到不同的物理服务器上。但除此之外，还有很多其它优化手段：比如使用缓存系统，将所有的，静态内容放到CDN等；还可以使用多线程技术将一台服务器的服务能力最大化。

1.1 进程与线程

- 进程：进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。
- 线程：线程是进程的一个执行路径，一个进程中至少有一个线程，进程中的多个线程共享进程的资源。

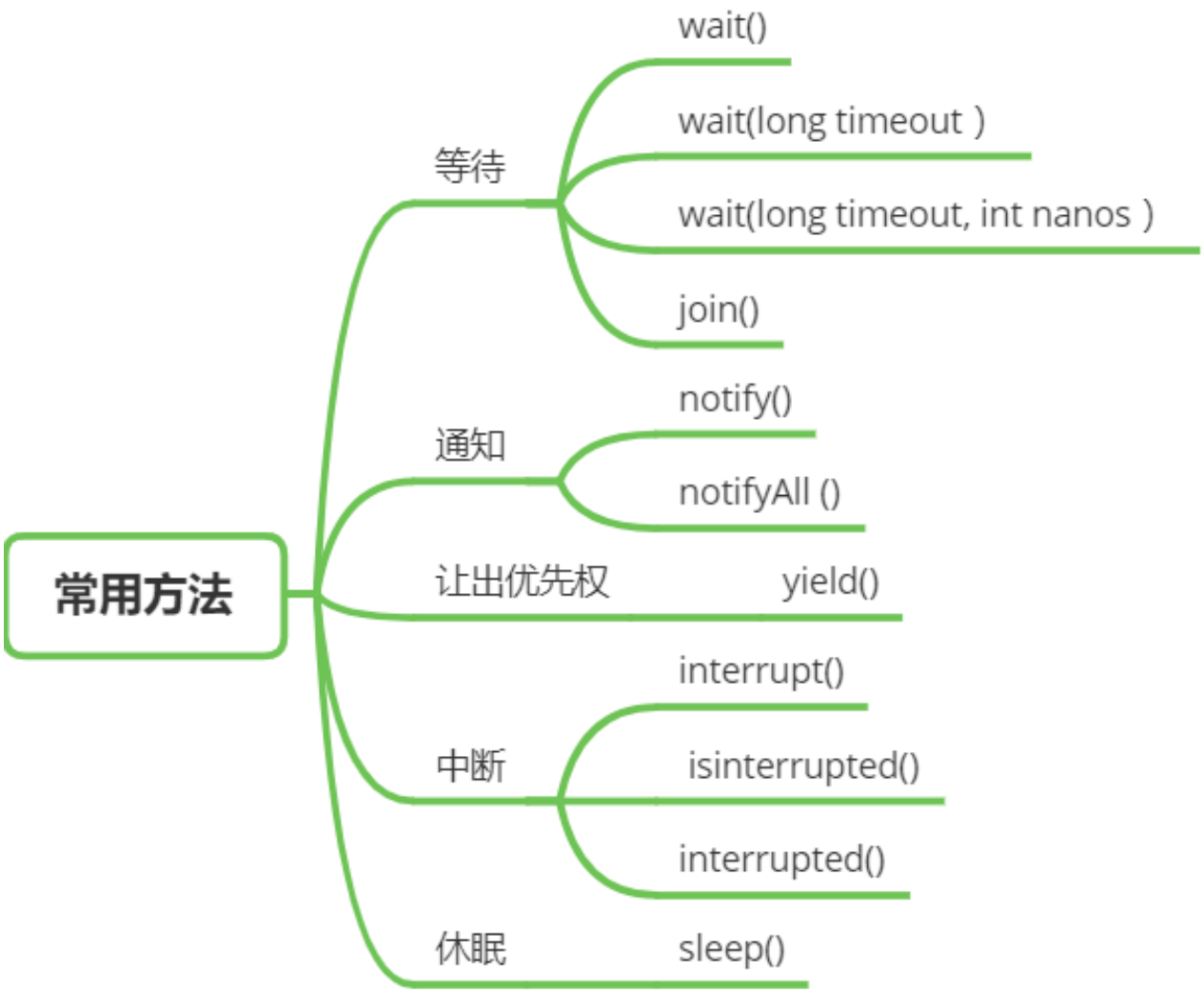
操作系统在分配资源时是把资源分配给进程的，但是CPU 资源比较特殊，它是被分配到线程的，因为真正要占用CPU运行的是线程，所以也说线程是CPU分配的基本单位。

比如在Java中，当我们启动main 函数其实就启动了一个JVM进程，而main 函数在的线程就是这个进程中的一个线程，也称主线程。

一个进程中有多个线程，多个线程共用进程的堆和方法区资源，但是每个线程有自己的程序计数器和栈。

CPU、内存、I/O 设备的速度是有极大差异的，为了合理利用CPU 的高性能，平衡这三者的速度差异，计算机体系结构、操作系统、编译程序都做出了贡献，主要体现为:

- CPU 增加了缓存，以均衡与内存的速度差异；// 导致可见性问题
- 操作系统增加了进程、线程，以分时复用CPU，进而均衡CPU 与I/O 设备的速度差异；// 导致原子性问题
- 编译程序优化指令执行次序，使得缓存能够得到更加合理地利用。// 导致有序性问题



1.2 线程安全的实现方法

1.2.1 互斥同步

synchronized 和ReentrantLock。

互斥同步最主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步。

互斥同步属于一种悲观的并发策略，总是认为只要不去做正确的同步措施，那就肯定会出现问题。无论共享数据是否真的会出现竞争，它都要进行加锁(这里讨论的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁)、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作。

1.2.2 非阻塞同步

比较并交换(CAS): 先进行操作: 如果没有其它线程争用共享数据，那操作就成功了，否则采取补偿措施(不断地重试，直到成功为止)。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步。乐观锁需要操作和冲突检测这两个步骤具备原子性，这里就不能再使用互斥同步来保证了，只能靠硬件来完成。CAS 指令需要有3 个操作数，分别是内存地址V、旧的预期值A 和新值B。当执行操作时，只有当V 的值等于A，才将V 的值更新为B。

1.2.3 无同步方案

要保证线程安全，并不是一定就要进行同步。如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性。

栈封闭: 多个线程访问同一个方法的局部变量时，不会出现线程安全问题，因为局部变量存储在虚拟机栈中，属于线程私有的。

线程本地存储(Thread Local Storage): 如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行。如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。

1.3 创建线程

没有返回值的:
继承Thread类，重写run()方法，调用start()方法启动线程

```
public static class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("This is child thread");
    }
}

public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
}
```

实现Runnable 接口，重写run()方法

```
public class RunnableTask implements Runnable {
    public void run() {
        System.out.println("Runnable!");
    }

    public static void main(String[] args) {
        RunnableTask task = new RunnableTask();
        new Thread(task).start();
    }
}
```

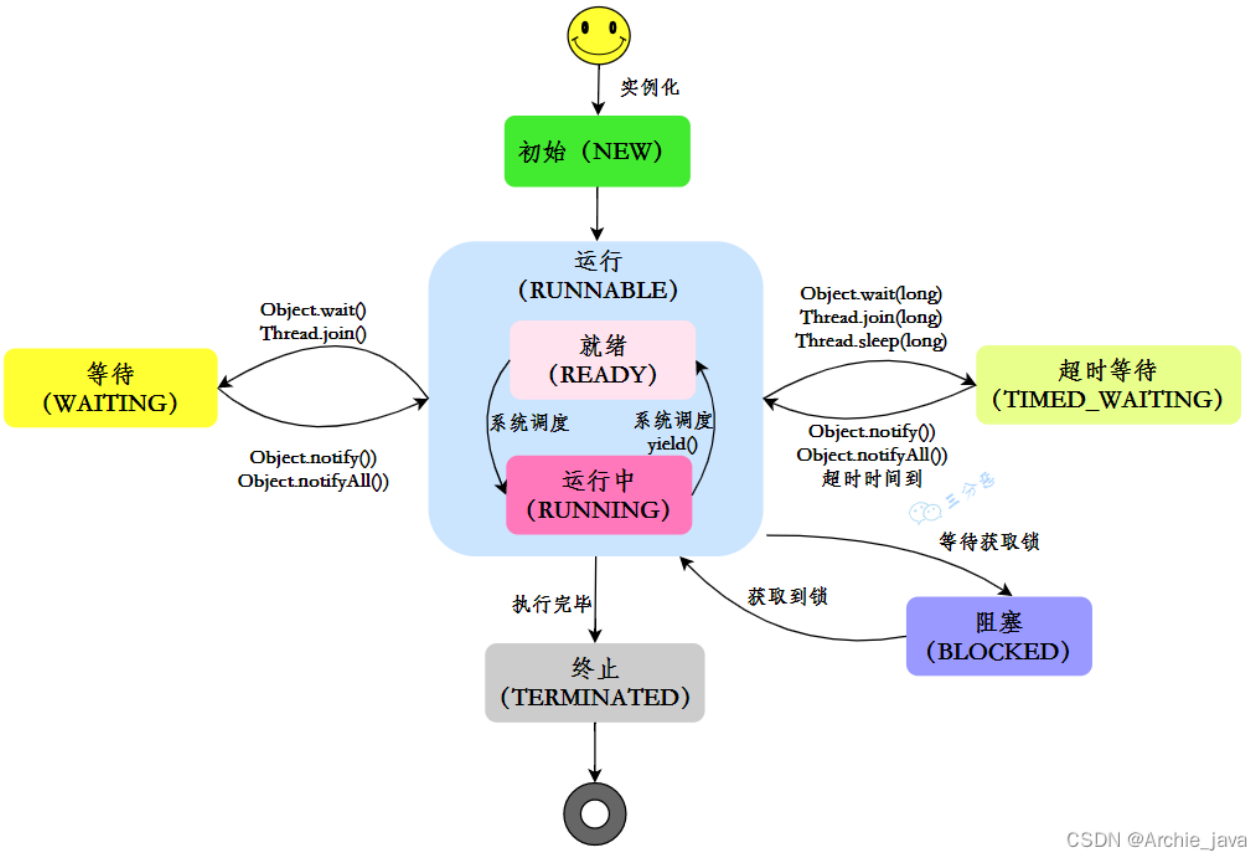
有返回值的:
实现Callable接口，重写call()方法，这种方式可以通过FutureTask获取任务执行的返回值

```
public class CallerTask implements Callable<String> {
    public String call() throws Exception {
        return "Hello,i am running!";
    }

    public static void main(String[] args) {
        //创建异步任务
        FutureTask<String> task=new FutureTask<String>(new CallerTask());
        //启动线程
        new Thread(task).start();
        try {
```

```
            //等待执行完成，并获取返回结果
            String result=task.get();
            System.out.println(result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

1.4 线程的状态

状态	说明
NEW	初始状态：线程被创建，但还没有调用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的称作"运行"
BLOCKED	阻塞状态：表示线程阻塞于锁
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 WAITIND，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



CSDN @Archie_java

1.5 线程的终止

通过stop、suspend等指令，可以终止一个线程或者挂起一个线程。但是使用此类方法强行中断可能会导致线程中任务未执行完就被强行中断，最终导致数据产生问题。

一般线程会在run()执行完毕后自动销毁，除非

1. run()方法中存在无限循环，如while(true)
2. 线程中存在一些阻塞的操作，比如sleep、wait、join等。

1.6 interrupt方法

Java 中的线程中断是一种线程间的协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是被中断的线程根据中断状态自行处理。

```
public class InterruptDemo {
    private static int i;
    public static void main(String[] args) throws InterruptedException {
        Thread thread=new Thread(()->{
        while(!Thread.currentThread().isInterrupted()){ //默认情况
    下isInterrupted返回false、通过thread.interrupt变成了true
            i++;
        }
            System.out.println("Num:"+i);
        },"interruptDemo");
    thread.start();
    TimeUnit.SECONDS.sleep(1);
    thread.interrupt(); //加和不加的效果
    }
}
```

这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅

1.6.1 处于阻塞状态下的线程中断

我们平时在线程中使用的sleep、wait、join等操作，它都会抛出一个InterruptedException异常，为什么会抛出异常，是因为它在阻塞期间，必须要能够响应被其他线程发起中断请求之后的一个响应，而这个响应是通过InterruptedException来体现的。

但是这里需要注意的是，在这个异常中如果不做任何处理的话，我们是无法去中断线程的，因为当前的异常只是响应了外部对于这个线程的中断命令，同时，线程的中断状态也会复位，如果需要中断，则还需要在catch中添加下面的代码

```
public class InterruptDemo {
    private static int i;
    public static void main(String[] args) throws InterruptedException {
        Thread thread=new Thread(()->{
            while(!Thread.currentThread().isInterrupted()){
                try {
                TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                e.printStackTrace();
                Thread.currentThread().interrupt(); //再次中断
                }
            }
            System.out.println("Num:"+i);
        },"interruptDemo");
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt();
        System.out.println(thread.isInterrupted());
    }
}
```

Interrupt可以被用来唤醒阻塞下的线程，或修改中断标记(false -¿ true)

2 原子性

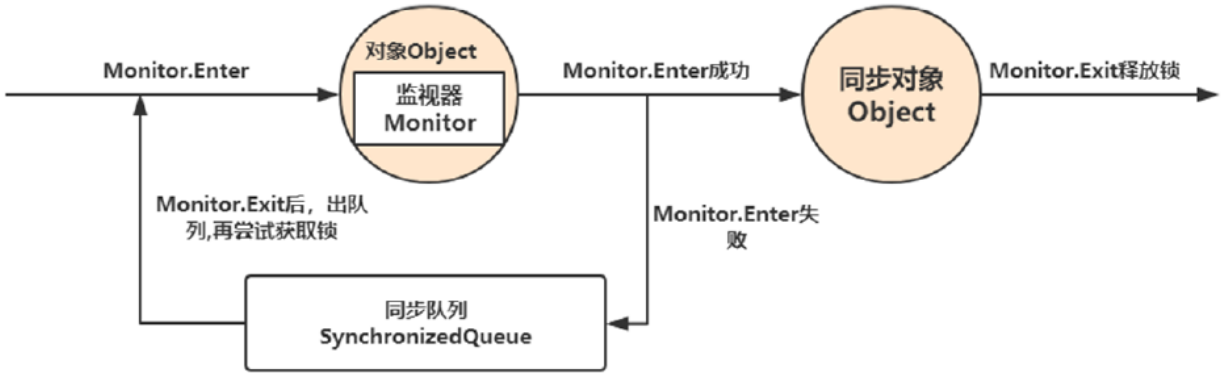
原子性：即一个操作或者多个操作要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。

在Java中除了提供Lock API外还在语法层面上提供了synchronized关键字来实现互斥同步原语:

- 一把锁只能同时被一个线程获取，没有获得锁的线程只能等待;
- 每个实例都对应有自己的一把锁(this),不同实例之间互不影响；例外：锁对象是*.class以及synchronized修饰的是static方法的时候，所有对象公用同一把锁
- synchronized修饰的方法，无论方法正常执行完毕还是抛出异常，都会释放锁

2.1 Synchronized的原理

实例对象结构里有对象头，对象头里面有一块结构叫Mark Word，Mark Word指针指向了monitor。所谓的Monitor其实是一种同步工具，也可以说是一种同步机制。在Java虚拟机（HotSpot）中，Monitor是由ObjectMonitor实现的，可以叫做内部锁，或者Monitor锁。每个同步对象都有一个自己的Monitor(监视器锁)，加锁过程如下图所示：



Synchronized在JVM里的实现都是基于进入和退出Monitor对象来实现方法同步和代码块同步，每一个对象在同一时间只与一个monitor(锁)相关联，而一个monitor在同一时间只能被一个线程获得，一个对象在尝试获得与这个对象相关联的Monitor锁的所有权的时候，monitorenter指令会发生如下3中情况之一：

- monitor计数器为0，意味着目前还没有被获得，那这个线程就会立刻获得然后把锁计数器+1，一旦+1，别的线程再想获取，就需要等待
- 如果这个monitor已经拿到了这个锁的所有权，又重入了这把锁，那锁计数器就会累加，变成2，并且随着重入的次数，会一直累加
- 这把锁已经被别的线程获取了，等待锁释放

monitorexit指令：释放对于monitor的所有权，释放过程很简单，就是讲monitor的计数器减1，如果减完以后，计数器不是0，则代表刚才是重入进来的，当前线程还继续持有这把锁的所有权，如果计数器变成0，则代表当前线程不再拥有该monitor的所有权，即释放锁。

可重入锁：又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。

2.2 JVM中锁的优化

简单来说在JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的Mutex Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行，这种切换的代价是非常昂贵的；然而在现实中的大部分情况下，同步方法是运行在单线程环境(无锁竞争环境)如果每次都调用Mutex Lock那么将严重的影响程序的性能。不过在jdk1.6中对锁的实现引入了大量的优化，如锁粗化(Lock Coarsening)、锁消除(Lock Elimination)、轻量级锁(Lightweight Locking)、偏向锁(Biased Locking)、适应性自旋(Adaptive Spinning)等技术来减少锁操作的开销。

- 锁粗化(Lock Coarsening)：也就是减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁。

- 锁消除(Lock Elimination)：通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本的Stack上进行对象空间的分配(同时还可以减少Heap上的垃圾收集开销)。

- 轻量级锁(Lightweight Locking)：这种锁实现的背后基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态(即单线程执行环境)，在无锁竞争的情况下完全可以避免调用操作系统层面的重量级互斥锁，取而代之的是在monitorenter和monitorexit中只需要依靠一条CAS原子指令就可以完成锁的获取及释放。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒。

- 偏向锁(Biased Locking)：是为了在无锁竞争的情况下避免在锁获取过程中执行不必要的CAS原子指令，因为CAS原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本地延迟。

- 适应性自旋(Adaptive Spinning)：当线程在获取轻量级锁的过程中执行CAS操作失败时，在进入与monitor相关联的操作系统重量级锁(mutex semaphore)前会进入忙等待(Spinning)然后再次尝试，当尝试一定的次数后如果仍然没有成功则调用与该monitor关联的semaphore(即互斥锁)进入到阻塞状态。

2.2.1 关于Synchronized锁的升级

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。锁膨胀方向：无锁→ 偏向锁→ 轻量级锁→ 重量级锁(此过程是不可逆的)。

这么设计的目的，其实是为了减少重量级锁带来的性能开销，尽可能的在无锁状态下解决线程并发问题，其中偏向锁和轻量级锁的底层实现是基于自旋锁，它相对于重量级锁来说，算是一种无锁的实现。

2.2.2 自旋锁

很多情况下，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和回复阻塞线程并不值得。在如今多处理器环境下，完全可以让另一个没有获取到锁的线程在门外等待一会(自旋)，但不放弃CPU的执行时间。等待持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需要让线程执行一个忙循环(自旋)。

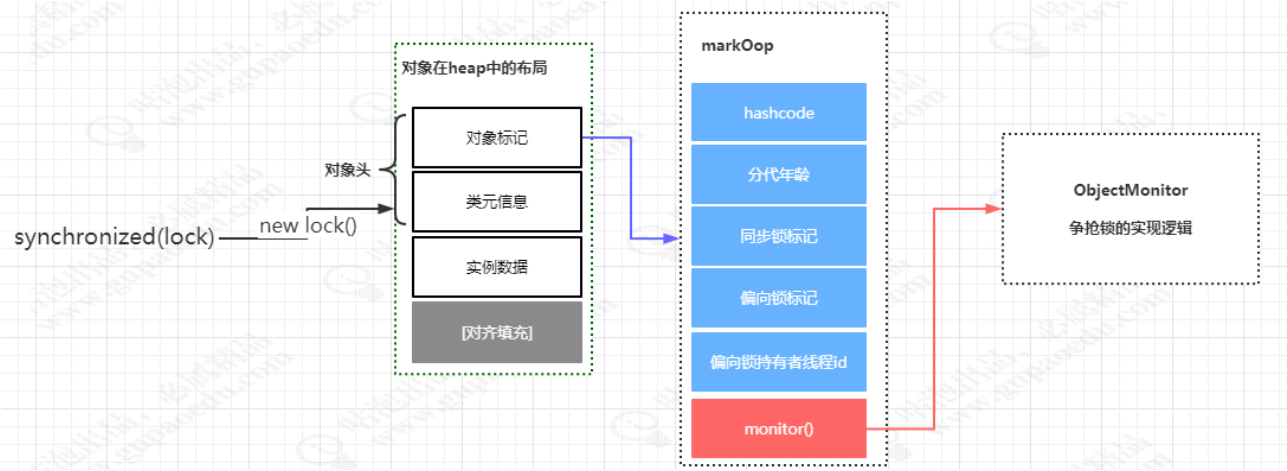
在线程自旋时，始终会占用CPU的时间片，如果锁占用的时间太长，那么自旋的线程会白白消耗掉CPU资源。因此自旋等待的时间必须要有一定的限度，如果自旋超过了限定的次数仍然没有成功获取到锁，就应该使用传统的方式去挂起线程了，在JDK定义中，自旋锁默认的自旋次数为10次，用户可以使用参数`-XX:PreBlockSpin`来更改。

2.2.3 自适应自旋锁

在JDK 1.6中引入了自适应自旋锁。这就意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定的。如果在同一个锁对象上，自旋等待刚刚成功获取过锁，并且持有锁的线程正在运行中，那么JVM会认为该锁自旋获取到锁的可能性很大，会自动增加等待时间。比如增加到100此循环。相反，如果对于某个锁，自旋很少成功获取锁。那再以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，JVM对程序的锁的状态预测会越来越准确，JVM也会越来越聪明。

2.2.4 轻量级锁

如果要理解轻量级锁，那么必须先要了解HotSpot虚拟机中对象头的内存布局。在对象头中(Object Header)存在两部分。第一部分用于存储对象自身的运行时数据，HashCode、GC Age、锁标记位、是否为偏向锁。等。一般为32位或者64位(视操作系统位数定)。官方称之为Mark Word，它是实现轻量级锁和偏向锁的关键。另外一部分存储的是指向方法区对象类型数据的指针(Klass Point)，如果对象是数组的话，还会有一个额外的部分用于存储数据的长度。



轻量级锁的获取：

1. 进行加锁操作时，jvm会判断是否已经时重量级锁，如果不是，则会在当前线程栈帧中划出一块空间，作为该锁的锁记录，并且将锁对象MarkWord复制到该锁记录中

2. 复制成功之后，jvm使用CAS操作将对象头MarkWord更新为指向锁记录的指针，并将锁记录里的owner指针指向对象头的MarkWord。如果成功，则执行‘3’，否则执行‘4’

3. 更新成功，则当前线程持有该对象锁，并且对象MarkWord锁标志设置为‘00’，即表示此对象处于轻量级锁状态

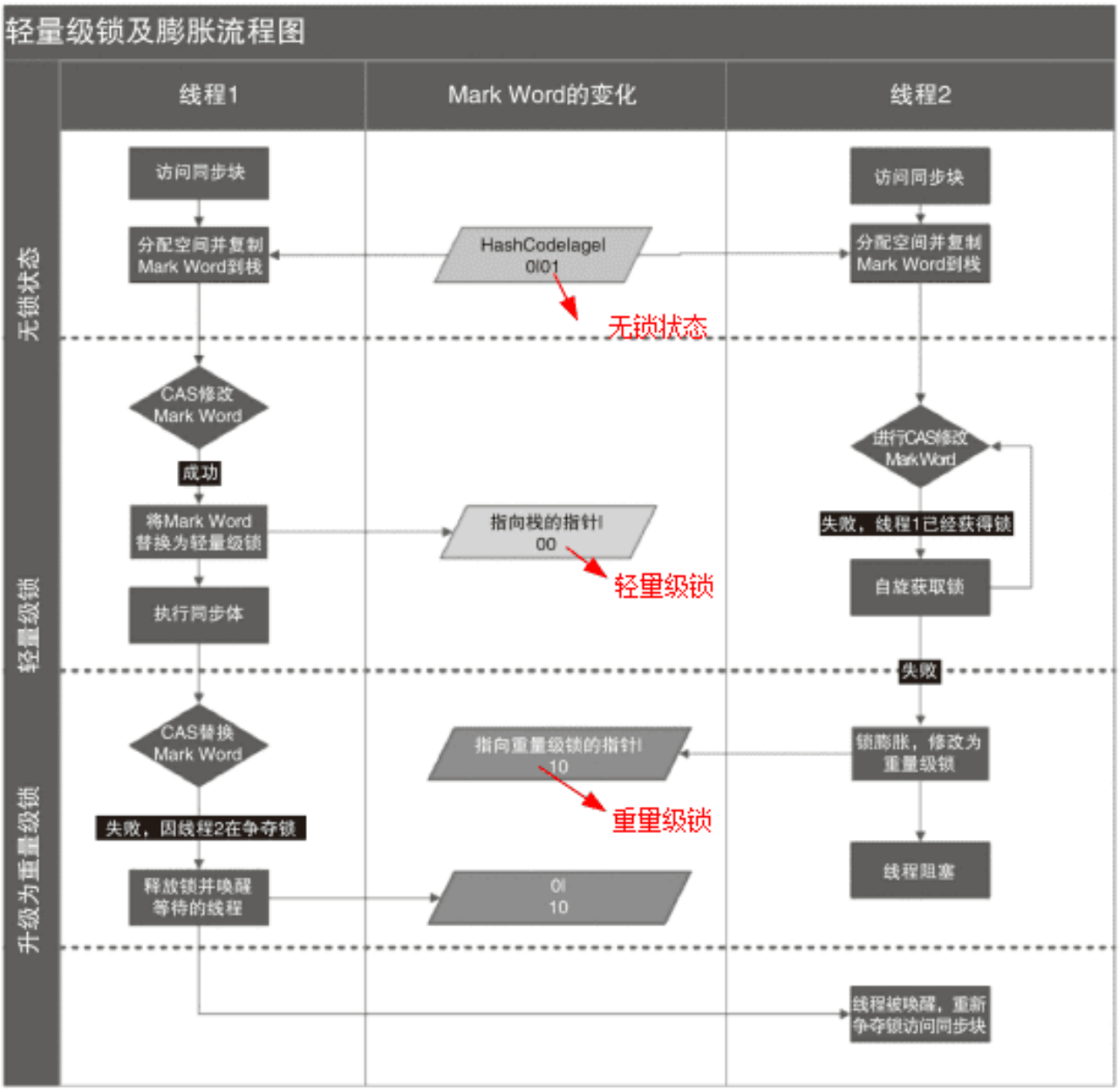
4. 更新失败，jvm先检查对象MarkWord是否指向当前线程栈帧中的锁记录，如果是则执行‘5’，否则执行‘4’

5. 表示锁重入；然后当前线程栈帧中增加一个锁记录第一部分（Displaced Mark Word）为null，并指向Mark Word的锁对象，起到一个重入计数器的作用。

6. 表示该锁对象已经被其他线程抢占，则进行自旋等待（默认10次），等待次数达到阈值仍未获取到锁，则升级为重量级锁

轻量级解锁时，会使用原子的CAS操作将Displaced Mark Word替换回到对象头中，如果成功，则表示没有发生竞争关系。如果失败，表示当前锁存在竞争关系。锁就会膨胀成重量

级锁。两个线程同时争夺锁，导致锁膨胀的流程图如下：



2.2.5 偏向锁

在大多实际环境下，锁不仅不存在多线程竞争，而且总是由同一个线程多次获取，那么在同一个线程反复获取所释放锁中，其中并还没有锁的竞争，那么这样看上去，多次的获取锁和释放锁带来了很多不必要的性能开销和上下文切换。

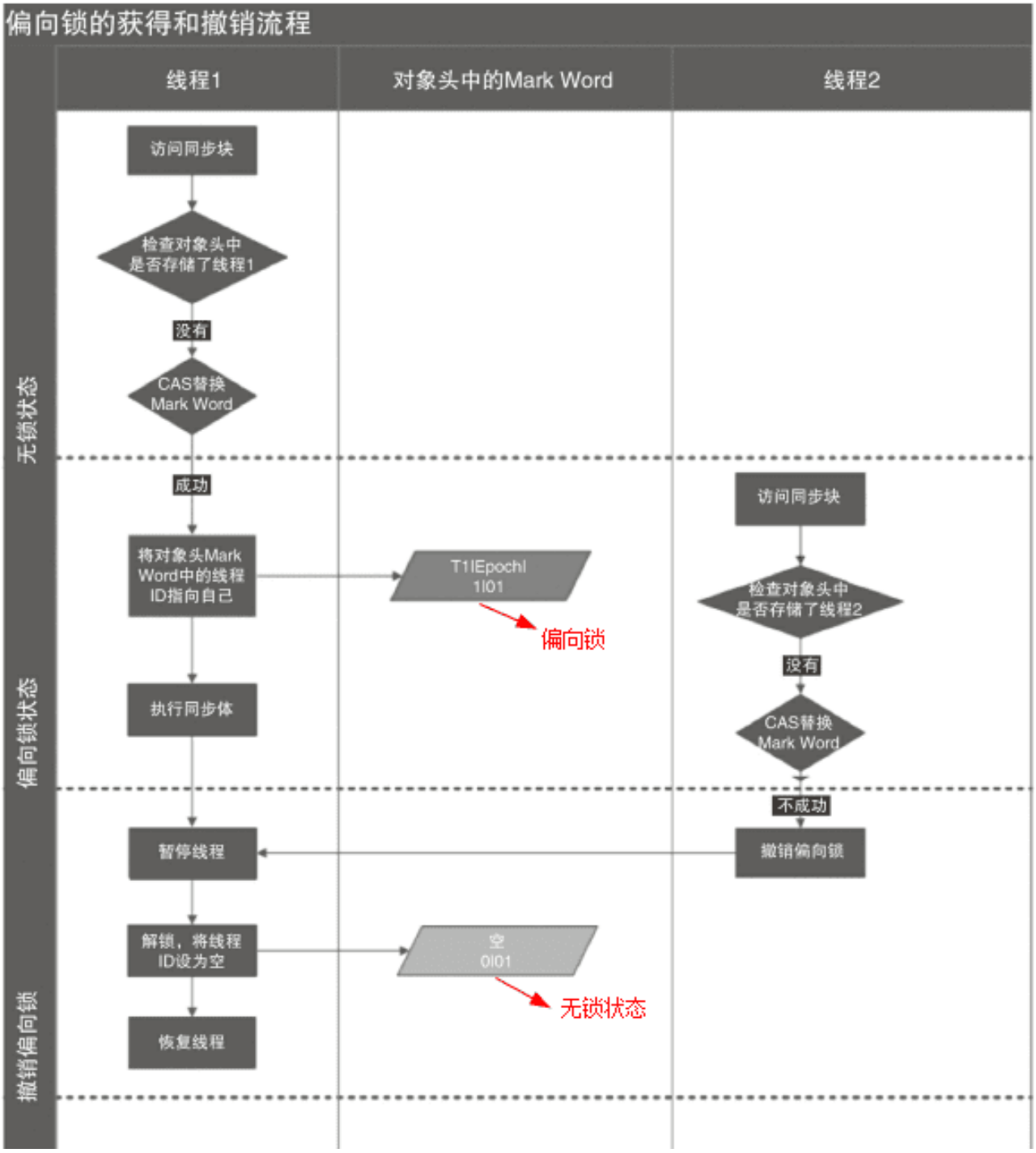
偏向锁的获取:

1. 判断是否为可偏向状态–MarkWord中锁标志是否为‘01’，是否偏向锁是否为‘1’
2. 如果是可偏向状态，则查看线程ID是否为当前线程，如果是，则进入步骤’5’，否则进入步骤‘3’
3. 通过CAS操作竞争锁，如果竞争成功，则将MarkWord中线程ID设置为当前线程ID，然后执行‘5’；竞争失败，则执行‘4’
4. CAS获取偏向锁失败表示有竞争。当达到safepoint时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块

5. 执行同步代码

偏向锁的撤销:

1. 偏向锁不会主动释放(撤销)，只有遇到其他线程竞争时才会执行撤销，由于撤销需要知道当前持有该偏向锁的线程栈状态，因此要等到safepoint时执行，此时持有该偏向锁的线程（T）有‘2’，‘3’两种情况；
2. 撤销—-T线程已经退出同步代码块，或者已经不再存活，则直接撤销偏向锁，变成无锁状态—-该状态达到阈值20则执行批量重偏向
3. 升级—-T线程还在同步代码块中，则将T线程的偏向锁升级为轻量级锁，当前线程执行轻量级锁状态下的锁获取步骤—-该状态达到阈值40则执行批量撤销



2.3 Wait/notify

在Object类中有一些函数可以用于线程的等待与通知。Wait和Notify只有在Synchronized中才能使用，这确保发起wait/notify的线程拥有此实例的锁（it ensures that the thread calling wait() actually holds the lock (monitor) associated with the object）。

线程等待:
`wait()`：当一个线程A调用一个共享变量的wait()方法时，线程A会被阻塞挂起，将线程放入堵塞队列WaitQueue，发生下面几种情况才会返回:

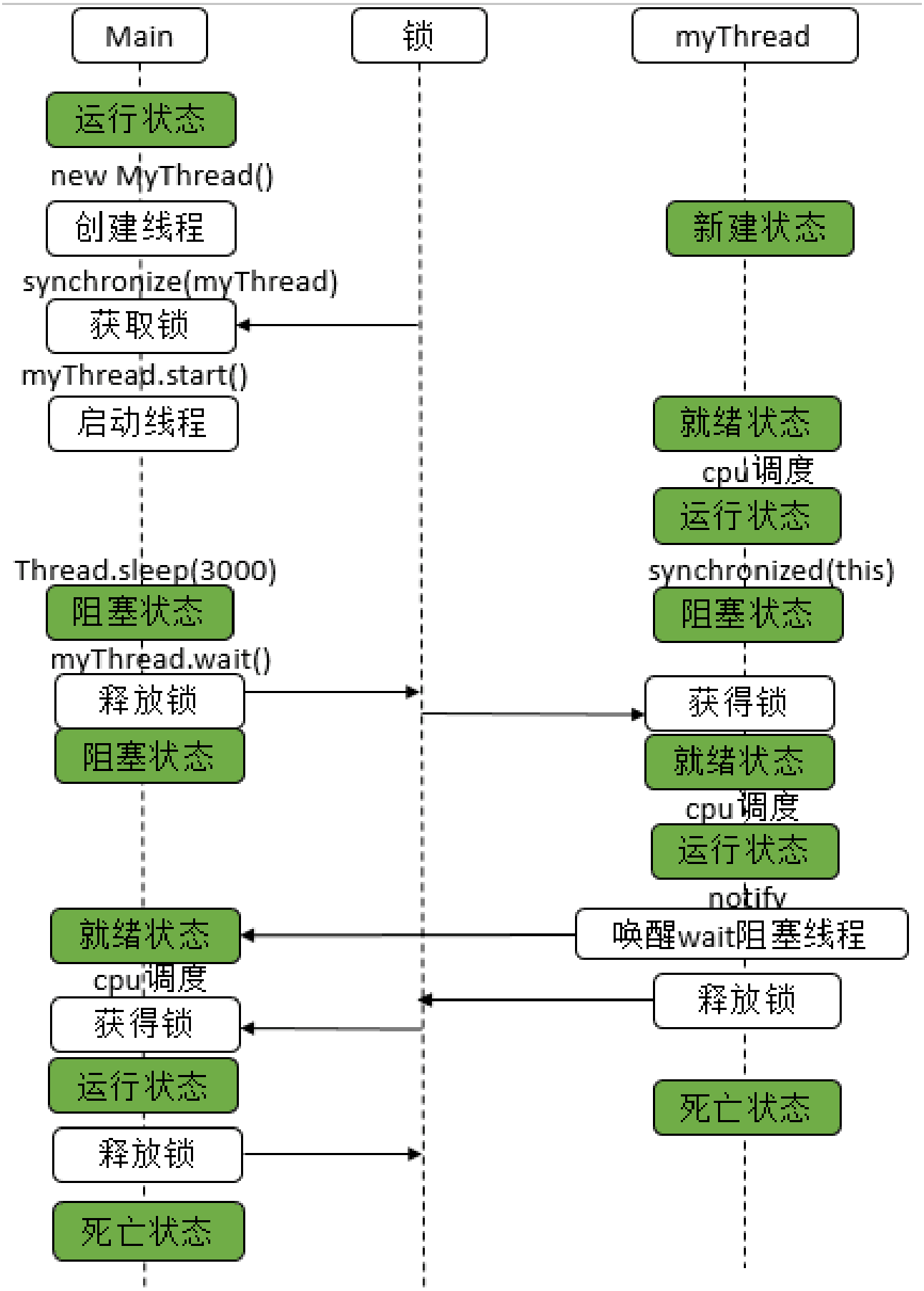
- 线程A调用了共享对象`notify()`或者`notifyAll()`方法;
- 其他线程调用了线程A的`interrupt()` 方法，线程A抛出`InterruptedException`异常返回。

wait(long timeout) ： 这个方法相比wait() 方法多了一个超时参数，它的不同之处在于，如果线程A调用共享对象的wait(long timeout)方法后，没有在指定的timeout ms时间内被其它线程唤醒，那么这个方法还是会因为超时而返回。唤醒线程，将在WaitQueue中的线程放回同步队列:

- `notify()` ： 一个线程A调用共享对象的`notify()` 方法后，会唤醒一个在这个共享变量上调用wait 系列方法后被挂起的线程。一个共享变量上可能会有多个线程在等待，具体唤醒哪个等待的线程是随机的。
- `notifyAll()` ： 不同于在共享变量上调用`notify()` 函数会唤醒被阻塞到该共享变量上的一个线程，`notifyAll()`方法则会唤醒所有在该共享变量上由于调用wait 系列方法而被挂起的线程。

例子:

```
class MyThread extends Thread {

    public void run() {
        synchronized (this) {
            System.out.println("before notify");
            notify();
            System.out.println("after notify");
        }
    }
}

public class WaitAndNotifyDemo {
    public static void main(String[] args) throws InterruptedException {
        MyThread myThread = new MyThread();
        synchronized (myThread) {
            try {
                myThread.start();
                // 主线程睡眠3s
                Thread.sleep(3000);
                System.out.println("before wait");
                // 阻塞主线程
                myThread.wait();
                System.out.println("after wait");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```


Main
锁
myThread
运行状态
new MyThread()
创建线程
新建状态
synchronize(myThread)
获取锁
myThread.start()
启动线程
就绪状态
cpu调度
运行状态
Thread.sleep(3000)
synchronized(this)
阻塞状态
阻塞状态
myThread.wait()
释放锁
获得锁
阻塞状态
就绪状态
cpu调度
运行状态
notify
就绪状态
唤醒wait阻塞线程
cpu调度
释放锁
获得锁
运行状态
死亡状态
释放锁
死亡状态

2.4 锁消除

锁消除是指虚拟机即时编译器再运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持。意思就是：JVM会判断再一段程序中的同步明显不会逃逸出去从而被其他线程访问到，那JVM就把它们当作栈上数据对待，认为这些数据是线程独有的，不需要加同步。此时就会进行锁消除。

2.5 CAS

CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性的。CAS 指令包含3 个参数：共享变量的内存地址A、预期的值B 和共享变量的新值C。

只有当内存中地址A 处的值等于B 时，才能将内存中地址A 处的值更新为新值C。作为一条CPU 指令，CAS 指令本身是能够保证原子性的。

2.5.1 ABA 问题

并发环境下，假设初始条件是A，去修改数据时，发现是A就会执行修改。但是看到的虽然是A，中间可能发生了A变B，B又变回A的情况。此时A已经非彼A，数据即使成功修改，也可能有问题。

解决方法：加版本号
每次修改变量，都在这个变量的版本号上加1，这样，刚刚A-¿B-¿A，虽然A的值没变，但是它的版本号已经变了，再判断版本号就会发现此时的A已经被改过了。参考乐观锁的版本号，这种做法可以给数据带上了一种实效性的检验。

Java提供了AtomicStampReference类，它的compareAndSet方法首先检查当前的对象引用值是否等于预期引用，并且当前印戳（Stamp）标志是否等于预期标志，如果全部相等，则以原子方式将引用值和印戳标志的值更新为给定的更新值。

2.5.2 循环性能开销

自旋CAS，如果一直循环执行，一直不成功，会给CPU带来非常大的执行开销。在Java中，很多使用自旋CAS的地方，会有一个自旋次数的限制，超过一定次数，就停止自旋。

2.5.3 只能保证一个变量的原子操作

CAS 保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS 目前无法直接保证操作的原子性的。

- 可以考虑改用锁来保证操作的原子性
- 可以考虑合并多个变量，将多个变量封装成一个对象，通过AtomicReference来保证原子性。

3 可见性

一个线程对共享变量的修改，另外一个线程能够立刻看到。

```
1 public class VolatileDemo {
2     public static boolean stop=false;
3 
4     public static void main(String[] args) throws InterruptedException {
5         Thread t1=new Thread(()->{
```

```
            int i=0;
            while(!stop){
                i++;
            }
        });
        t1.start();
        System.out.println("begin start thread");
        Thread.sleep(1000);
        stop=true;
    }
}
```

t1线程中用到了stop这个属性，接在在main线程中修改了stop 这个属性的值来使得t1线程结束，但是t1线程并没有按照期望的结果执行。

3.1 缓存

在整个计算机的发展历程中，除了CPU、内存以及I/O设备不断迭代升级来提升计算机处理性能之外，还有一个非常核心的矛盾点，就是这三者在处理速度的差异。CPU的计算速度是非常快的，其次是内存、最后是IO设备（比如磁盘），也就是CPU的计算速度远远高于内存以及磁盘设备的I/O速度。

CPU在做计算时，和内存的IO操作是无法避免的，而这个IO过程相对于CPU的计算速度来说是非常耗时，基于这样一个问题，所以在CPU层面设计了高速缓存，CPU 将未来最有可能被用到的内存数据加载进缓存。如果下次访问内存时，数据已经在缓存中了，这就是缓存命中，它获取目标数据的速度非常快。如果数据没在缓存中，这就是缓存缺失，此时要启动内存数据传输，而内存的访问速度相比缓存差很多。通过这样一个机制可以减少CPU和内存的交互开销从而提升CPU的利用率。

对于主流的x86平台，cpu的缓存行（cache）分为L1、L2、L3总共3级。当CPU 执行运算的时候，它先去L1 查找所需的数据，再去L2，然后是L3，最后如果这些缓存中都没有，所需的数据就要去主内存拿。走得越远，运算耗费的时间就越长。所以如果你在做一些很频繁的事，要确保数据在L1 缓存中。

3.2 缓存一致性问题

CPU高速缓存的出现，虽然提升了CPU的利用率，但是同时也带来了另外一个问题–缓存一致性问题:

在多线程环境中，当多个线程并行执行加载同一块内存数据时，由于每个CPU都有自己独立的L1、L2缓存，所以每个CPU的这部分缓存空间都会缓存到相同的数据，并且每个CPU执行相关指令时，彼此之间不可见，就会导致未执行指令的其他cpu依然缓存着旧值。

那么，要解决这一问题，就需要一种机制，来同步两个不同核心里面的缓存数据。要实现的这个机制的话，要保证做到下面这2 点:

1. 某个CPU 核心里的Cache 数据更新时，必须要传播到其他核心的Cache，这个称为写传播（*Write Propagation*）
2. 某个CPU 核心里对数据的操作顺序，必须在其他核心看起来顺序是一样的，这个称为事务的串形化（*Transaction Serialization*）
    - CPU 核心对于Cache 中数据的操作，需要同步给其他CPU 核心;
    - 要引入「锁」的概念，如果两个CPU 核心里有相同数据的Cache，那么对于这个Cache 数据的更新，只有拿到了「锁」，才能进行对应的数据更新。

3.2.1 总线Bus

CPU要和存储设备进行交互，必须要通过总线设备，在获取到总线控制权后才能启动数据信息的传输，而CPU要想从主存读写数据，那么就必须向总线发起一个总线事务（读事务或写事务）来从主存读取或者写入数据。

要解决缓存一致性问题，首先要解决的是多个CPU核心之间的数据传播问题。最常见的一种解决方案呢，叫作总线嗅探（Bus Snooping）。当A 号CPU 核心修改了L1 Cache 中i 变量的值，通过总线把这个事件广播通知给其他所有的核心，然后每个CPU 核心都会监听总线上的广播事件，并检查是否有相同的数据在自己的L1 Cache 里面，如果B 号CPU 核心的L1 Cache 中有该数据，那么也需要把该数据更新到自己的L1 Cache。总线嗅探只是保证了某个CPU 核心的Cache 更新数据这个事件能被其他CPU 核心知道，但是并不能保证事务串形化。

导致缓存不一致的另外一个问题在于，CPU操作共享数据的顺序性，想让并发的操作变得有序，那么常用的方式就是让操作的资源具备独占性，这也就是我们常用的的方式加锁，当一个CPU对操作的资源加了锁，那么其它CPU就只能等待，只有等前一个释放了锁（资源占用权），后面的才能获得执行权，从而保证整体操作的顺序性。

而实现这个机制的功能就叫“总线仲裁”，在多个CPU同时申请对总线的使用权时，为避免产生总线冲突，需由总线仲裁来合理地控制和管理系统中需要占用总线的申请者，在多个申请者同时提出总线请求时，以一定的优先算法仲裁哪个应获得对总线的使用权。

3.2.2 基于总线锁解决缓存一致性

在CPU1要做i++操作的时候，其在总线上发出一个LOCK#信号，其他处理器就不能操作缓存了该共享变量内存地址的缓存，也就是阻塞了其他CPU，这就使得同一时刻只有一个core可以访问共享内存(也就是i),从而解决了缓存不一致的问题,你不是多个core 缓存了变量i，然后修改导致彼此之间不一致吗，那我只让一个core 读内存，其他的core不能访问内存，实现串行化，自然也没有缓存一致性问题了，但是这种方法的代价是，cpu的利用率直线下降，只要一个CPU占用了总线，那么其它的CPU就无法与主存进行通讯而只能等待前一个执行完成，所以为了减小锁的粒度，引入了缓存锁

3.2.3 基于缓存锁解决缓存一致性

所谓“缓存锁”是指多个cpu想写同一个cache line的数据，加上LOCK前缀,去总线申请缓存锁，如果可以，那就会把当前cache line锁住，然后其他cpu如果此时也要对这个内存块的cache line操作，也需要去总线申请锁，如果失败，需要等待锁释放,这种实现是基于mesi协议的。

3.2.4 缓存一致性协议(MESI)

多线程编程时, 另外一个核的线程想要访问当前核内L1、L2 缓存行的数据, 该怎么办呢:

有人说可以通过第2 个核直接访问第1 个核的缓存行，这是当然是可行的，但这种方法不够快。跨核访问需要通过Memory Controller（内存控制器，是计算机系统内部控制内存并且通过内存控制器使内存与CPU 之间交换数据的重要组成部分），典型的情况是第2 个核经常访问第1 个核的这条数据，那么每次都有跨核的消耗.。更糟的情况是，有可能第2 个核与第1 个核不在一个插槽内，况且Memory Controller 的总线带宽是有限的，扛不住这么多数据传输。所以，CPU 设计者们更偏向于另一种办法：如果第2 个核需要这份数据，由第1 个核直接把数据内容发过去，数据只需要传一次。

那么什么时候会发生缓存行的传输呢？答案很简单：当一个核需要读取另外一个核的脏缓存行时发生。

- M（修改，Modified）：本地处理器已经修改缓存行，即是脏行，它的内容与内存中的内容不一样，并且此cache 只有本地一个拷贝(专有)；
- E（专有，Exclusive）：缓存行内容和内存中的一样，而且其它处理器都没有这行数据；
- S（共享，Shared）：缓存行内容和内存中的一样, 有可能其它处理器也存在此缓存行的拷贝；
- I（无效，Invalid）：缓存行失效, 不能使用。

下面说明这四个状态是如何转换的：

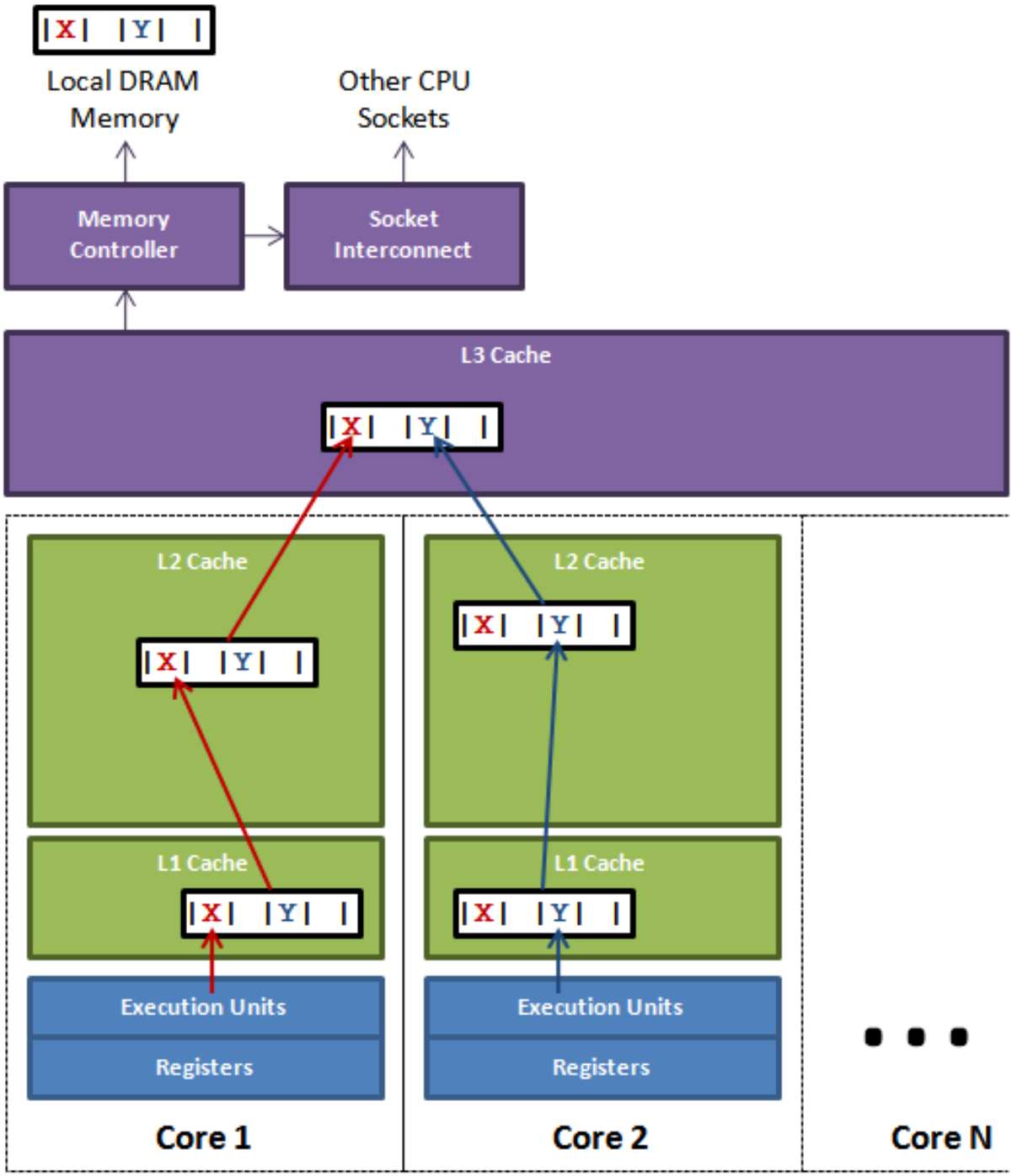
- 初始：一开始时，缓存行没有加载任何数据，所以它处于I 状态。
- 本地写（Local Write）：如果本地处理器写数据至处于I 状态的缓存行，则缓存行的状态变成M。
- 本地读（Local Read）：如果本地处理器读取处于I 状态的缓存行，很明显此缓存没有数据给它。此时分两种情况：(1)其它处理器的缓存里也没有此行数据，则从内存加载数据到此缓存行后，再将它设成E 状态，表示只有我一家有这条数据，其它处理器都没有；(2)其它处理器的缓存有此行数据，则将此缓存行的状态设为S 状态。（备注：如果处于M状态的缓存行，再由本地处理器写入/读出，状态是不会改变的）
- 远程读（Remote Read）：假设我们有两个处理器c1 和c2，如果c2 需要读另外一个处理器c1 的缓存行内容，c1 需要把它缓存行的内容通过内存控制器(Memory Controller)发送给c2，c2 接到后将相应的缓存行状态设为S。在设置之前，内存也得从总线上得到这份数据并保存。
- 远程写（Remote Write）：其实确切地说不是远程写，而是c2 得到c1 的数据后，不是为了读，而是为了写。也算是本地写，只是c1 也拥有这份数据的拷贝，这该怎么办呢？c2 将发出一个RFO (Request For Owner) 请求，它需要拥有这行数据的权限，其它处理器的相应缓存行设为I，除了它自己，谁不能动这行数据。这保证了数据的安全，同时处理RFO 请求以及设置I的过程将给写操作带来很大的性能消耗。

3.2.5 缓存行

缓存系统中是以缓存行（cache line）为单位存储的。缓存行通常是64 字节（译注：本文基于64 字节，其他长度的如32 字节等不适本文讨论的重点），并且它有效地引用主内存中的一块地址。一个Java 的long 类型是8 字节，因此在一个缓存行中可以存8 个long 类型的变量。所以，如果你访问一个long 数组，当数组中的一个值被加载到缓存中，它会额外加载另外7 个，以致你能非常快地遍历这个数组。事实上，你可以非常快速的遍历在连续的内存块中分配的任意数据结构。而如果你在数据结构中的项在内存中不是彼此相邻的（如链表），你将得不到免费缓存加载所带来的优势，并且在这些数据结构中的每一个项都可能会出现缓存未命中。

3.2.6 伪共享

如果存在这样的场景，有多个线程操作不同的成员变量，但是相同的缓存行，这个时候会发生什么？。没错，伪共享（False Sharing）问题就发生了！有张Disruptor 项目的经典示例图，如下：



上图中，一个运行在处理器core1上的线程想要更新变量X 的值，同时另外一个运行在处理器core2 上的线程想要更新变量Y 的值。但是，这两个频繁改动的变量都处于同一条缓存行。两个线程就会轮番发送RFO 消息，占得此缓存行的拥有权。当core1 取得了拥有权开始更新X，则core2 对应的缓存行需要设为I 状态。当core2 取得了拥有权开始更新Y，则core1 对应的缓存行需要设为I 状态(失效态)。轮番夺取拥有权不但带来大量的RFO 消息，而且如果某个线程需要读此行数据时，L1 和L2 缓存上都是失效数据，只有L3 缓存上是同步好的数据。从前一篇我们知道，读L3 的数据非常影响性能。更坏的情况是跨槽读取，L3 都要miss，只能从内存上加载。

表面上X 和Y 都是被独立线程操作的，而且两操作之间也没有任何关系。只不过它们共享了一个缓存行，但所有竞争冲突都是来源于共享。

3.2.7 如何避免伪共享?

其中一个解决思路，就是让不同线程操作的对象处于不同的缓存行即可：通过缓存行填

充(Padding)。

我们知道一条缓存行有64 字节，而Java 程序的对象头固定占8 字节(32位系统)或12 字节(64 位系统默认开启压缩, 不开压缩为16 字节)，所以我们只需要填6 个无用的长整型补上6*8=48字节，让不同的VolatileLong 对象处于不同的缓存行，就避免了伪共享(64 位系统超过缓存行的64 字节也无所谓，只要保证不同线程不操作同一缓存行就可以)。

另外一种技术是使用编译指示，来强制使每一个变量对齐: 编译器使用`__declspec( align(n) )` 此处n=64，按照cache line 边界对齐。

当使用数组时，在cache line 尾部填充padding 来保证数据元素在cache line 边界开始。如果不能够保证数组按照cache line 边界对齐，填充数据结构【数组元素】使之是cache line 大小的两倍。下面的代码显式了填充数据结构使之按照cache line 对齐。并且通过`__declspec( align(n) )` 语句来保证数组也是对齐的。如果数组是动态分配的，你可以增加分配的大小，并调整指针来对其到cache line 边界。

4 有序性

有序性：即程序执行的顺序按照代码的先后顺序执行。

4.1 指令重排

在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。重排序分3种类型:

1. 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2. 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

指令重排也是有一些限制的，有两个规则happens-before和as-if-serial来约束。

happens-before的定义：

- 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
- 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序并不非法

happens-before和我们息息相关的有六大规则：

1. 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
2. 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
3. volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4. 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

5. start()规则：如果线程A执行操作`ThreadB.start()`（启动线程B），那么A线程的`ThreadB.start()`操作happens-before于线程B中的任意操作。

6. join()规则：如果线程A执行操作`ThreadB.join()`并成功返回，那么线程B中的任意操作happens-before于线程A从`ThreadB.join()`操作成功返回。

as-if-serial语义的意思是：不管怎么重排序（编译器和处理器为了提高并行度），单线程程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守as-if-serial语义。

为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

5 Volatile

相比synchronized的加锁方式来解决共享变量的内存可见性问题，volatile就是更轻量的选择，它没有上下文切换的额外开销成本。

volatile可以确保对某个变量的更新对其他线程马上可见，一个变量被声明为volatile 时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存当其它线程读取该共享变量，会从主内存重新获取最新值，而不是使用当前线程的本地内存中的值。

5.0.1 Volatile可见性实现

- 为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存(L1，L2 或其他)后再进行操作，但操作完不知道何时会写到内存。
- 如果对声明了volatile 的变量进行写操作，JVM 就会向处理器发送一条lock 前缀的指令，将这个变量所在缓存行的数据写回到系统内存。
- 为了保证各个处理器的缓存是一致的，实现了缓存一致性协议(MESI)，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。
- 所有多核处理器下还会完成：当处理器发现本地缓存失效后，就会从内存中重读该变量数据，即可以获取当前最新值。

volatile 变量通过这样的机制就使得每个线程都能获得该变量的最新值。

5.0.2 volatile 有序性实现

volatile 变量的内存有序性是基于内存屏障(Memory Barrier)实现:

内存屏障，又称内存栅栏，是一个CPU 指令。在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序，JMM 为了保证在不同的编译器和CPU 上有相同的结果，通过插入特定类型的内存屏障来禁止+ 特定类型的编译器重排序和处理器重排序，插入一条内存屏障会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier 指令重排序。

为了实现volatile 内存语义时，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎是不可能的，为此，JMM 采取了保守的策略:

- 在每个volatile 写操作的前面插入一个StoreStore 屏障: 禁止上面的普通写和下面的volatile 写重排序。
- 在每个volatile 写操作的后面插入一个StoreLoad 屏障: 防止上面的volatile 写与下面可能有的volatile 读/写重排序。
- 在每个volatile 读操作的后面插入一个LoadLoad 屏障: 禁止下面所有的普通读操作和上面的volatile 读重排序。
- 在每个volatile 读操作的后面插入一个LoadStore 屏障: 禁止下面所有的普通写操作和上面的volatile 读重排序。

5.0.3 volatile 的应用场景

使用volatile 必须具备的条件:

- 对变量的写操作不依赖于当前值。
- 该变量没有包含在具有其他变量的不变式中。
- 只有在状态真正独立于程序内其他内容时才能使用volatile。

5.1 JMM

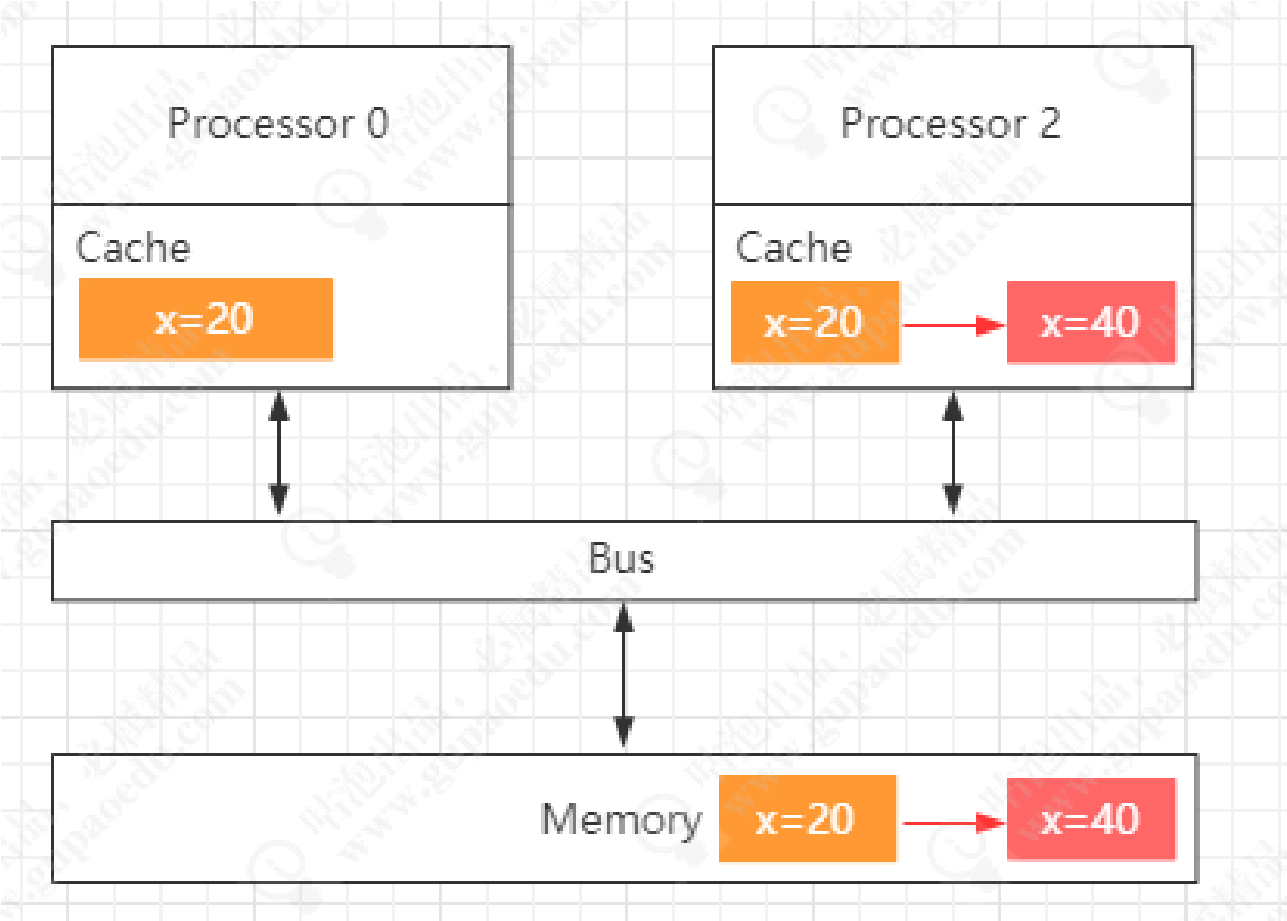
首先，我们都知道Java程序是运行在Java虚拟机上的，同时我们也知道，JVM是一个跨语言跨平台的实现，也就是Write Once、Run Anywhere。

那么JVM如何实现在不同平台上都能达到线程安全的目的呢？所以这个时候JMM出来了，Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范

JMM定义了共享内存中多线程程序读写操作的行为规范：在虚拟机中把共享变量存储到内存以及从内存中取出共享变量的底层实现细节。

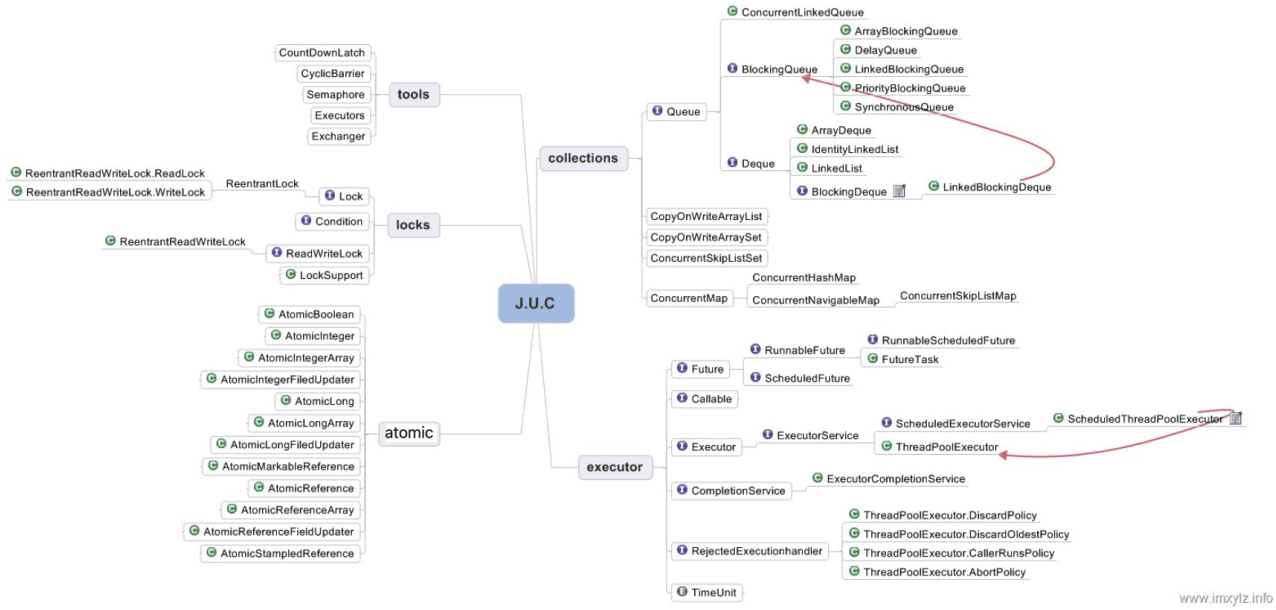
Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了这个线程中用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行，流程图如下:



6 JUC

JUC，即`java.util.concurrent`包的缩写，是java原生的并发包和一些常用的工具类。



6.1 AbstractQueuedSynchronizer

AbstractQueuedSynchronizer 抽象同步队列，简称AQS ， 它是Java并发包的根基，并发包中的锁就是基于AQS实现的。没有抢占到锁的线程会被放入AQS。

- AQS是基于一个FIFO的双向队列，其内部定义了一个节点类Node，Node 节点内部的SHARED 用来标记该线程是获取共享资源时被阻挂起后放入AQS 队列的，EXCLUSIVE用来标记线程是取独占资源时被挂起后放入AQS 队列
- AQS 使用一个volatile 修饰的int 类型的成员变量state 来表示同步状态，修改同步状态成功即为获得锁，volatile 保证了变量在多线程之间的可见性，修改State 值时通过CAS机制来保证修改的原子性
- 获取state的方式分为两种，独占方式和共享方式，一个线程使用独占方式获取了资源，其它线程就会在获取失败后被阻塞。一个线程使用共享方式获取了资源，另外一个线程还可以通过CAS的方式进行获取。
- 如果共享资源被占用，需要一定的阻塞等待唤醒机制来保证锁的分配，AQS 中会将竞争共享资源失败的线程添加到一个变体的CLH 队列中。

先简单了解一下CLH：Craig、Landin and Hagersten 队列，是单向链表实现的队列。申请线程只在本地变量上自旋，它不断轮询前驱的状态，如果发现前驱节点释放了锁就结束自旋

- AQS 中队列是个双向链表，也是FIFO 先进先出的特性
- 通过Head、Tail 头尾两个节点来组成队列结构，通过volatile 修饰保证可见性
- Head 指向节点为已获得锁的节点，是一个虚拟节点，节点本身不持有具体线程
- 获取不到同步状态，会将节点进行自旋获取锁，自旋一定次数失败后会将线程阻塞，相对于CLH 队列性能较好

```
static final class Node {
    // 模式，分为共享与独占
    // 共享模式
    static final Node SHARED = new Node();
    // 独占模式
    static final Node EXCLUSIVE = null;
    // 结点状态
    // CANCELLED，值为1，表示当前的线程被取消
    // SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark
    // CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中
    // PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行
    // 值为0，表示当前节点在sync队列中，等待着获取锁
    static final int CANCELLED =  1;
    static final int SIGNAL    = -1;
    static final int CONDITION = -2;
    static final int PROPAGATE = -3;

    // 结点状态
    volatile int waitStatus;
    // 前驱结点
    volatile Node prev;
    // 后继结点
    volatile Node next;
    // 结点所对应的线程
    volatile Thread thread;
    // 下一个等待者
    Node nextWaiter;

    // 结点是否在共享模式下等待
    final boolean isShared() {
```

```
        return nextWaiter == SHARED;
    }
    //...
    // 构造方法
        Node(Thread thread, Node mode) {     // Used by addWaiter
        this.nextWaiter = mode;
        this.thread = thread;
    }

    // 构造方法
    Node(Thread thread, int waitStatus) { // Used by Condition
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}
```

6.1.1 acquire方法

该方法以独占模式获取(资源)，忽略中断，即线程在aquire过程中，中断此线程是无效的。源码如下:

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

当一个线程调用acquire时，调用方法流程如下:

1. 首先调用tryAcquire方法，调用此方法的线程会试图在独占模式下获取对象状态。此方法应该查询是否允许它在独占模式下获取对象状态，如果允许，则获取它。在AbstractQueuedSynchronizer源码中默认会抛出一个异常，即需要子类去重写此方法完成自己的逻辑。

2. 若tryAcquire失败，则调用addWaiter方法，addWaiter方法完成的功能是将调用此方法的线程封装成为一个结点并放入Sync queue。

3. 调用acquireQueued方法，此方法完成的功能是Sync queue中的结点不断尝试获取资源，若成功，则返回true，否则，返回false。

acquireQueue方法:

```
// sync队列中的结点在独占且忽略中断的模式下获取(资源)
final boolean acquireQueued(final Node node, int arg) {
    // 标志
    boolean failed = true;
    try {
        // 中断标志
        boolean interrupted = false;
        for (;;) { // 无限循环
            // 获取node节点的前驱结点
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) { // 前驱为头节点并且成功获得锁
                setHead(node); // 设置头节点
                p.next = null; // help GC
                failed = false; // 设置标志
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())//通过locksupport进行阻塞
```

```
                interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

6.1.2 release方法

```
public final boolean release(int arg) {
    if (tryRelease(arg)) { // 释放成功
        // 保存头节点
        Node h = head;
        if (h != null && h.waitStatus != 0) // 头节点不为空并且头节点状态不为0
            unparkSuccessor(h); //释放头节点的后继结点
        return true;
    }
    return false;
}s
```

其中，tryRelease的默认实现是抛出异常，需要具体的子类实现，如果tryRelease成功，那么如果头节点不为空并且头节点的状态不为0，则释放头节点的后继结点

6.2 locksupport

LockSupport用来创建锁和其他同步类的基本线程阻塞原语。简而言之，当调用LockSupport.park时，表示当前线程将会等待，直至获得许可，当调用LockSupport.unpark时，必须把等待获得许可的线程作为参数进行传递，好让此线程继续运行。

和Wait/Notify不同，Wait/Notify只能使用在Synchronize锁中，而park/unpark只能使用在lock中。

引入sun.misc.Unsafe类中的park和unpark函数，因为LockSupport的核心函数都是基于Unsafe类中定义的park和unpark函数，下面给出两个函数的定义:

```
public native void park(boolean isAbsolute, long time);
public native void unpark(Thread thread);
```

- park函数，阻塞线程，并且该线程在下列情况发生之前都会被阻塞: ①调用unpark函数，释放该线程的许可。②该线程被中断。③设置的时间到了。并且，当time为绝对时间时，isAbsolute为true，否则，isAbsolute为false。当time为0时，表示无限等待，直到unpark发生。

- unpark函数，释放线程的许可，即激活调用park后阻塞的线程。这个函数不是安全的，调用这个函数时要确保线程依旧存活。

JUC的park函数:

```
public static void park() {
    // 获取许可，设置时间为无限长，直到可以获取许可
    UNSAFE.park(false, 0L);
}

public static void park(Object blocker) {
    // 获取当前线程
    Thread t = Thread.currentThread();
    // 设置Blocker
```

```
    setBlocker(t, blocker);
    // 获取许可
    UNSAFE.park(false, 0L);
    // 重新可运行后再此设置Blocker
    setBlocker(t, null);
}
```

其中parkBlocker是用于记录线程是被谁阻塞的。可以通过LockSupport的getBlocker获取到阻塞的对象。主要用于监控和分析线程用的。

unpark函数:
此函数表示如果给定线程的许可尚不可用，则使其可用。如果线程在park 上受阻塞，则它将解除其阻塞状态。否则，保证下一次调用park 不会受阻塞。如果给定线程尚未启动，则无法保证此操作有任何效果。具体函数如下:

```
public static void unpark(Thread thread) {
    if (thread != null) // 线程为不空
        UNSAFE.unpark(thread); // 释放该线程许可
}
```

6.3 接口：Lock

Lock为接口类型，Lock实现提供了比使用synchronized方法和语句可获得的更广泛的锁定操作。此实现允许更灵活的结构，可以具有差别很大的属性，可以支持多个相关的Condition对象。

6.4 ReentrantLock详解

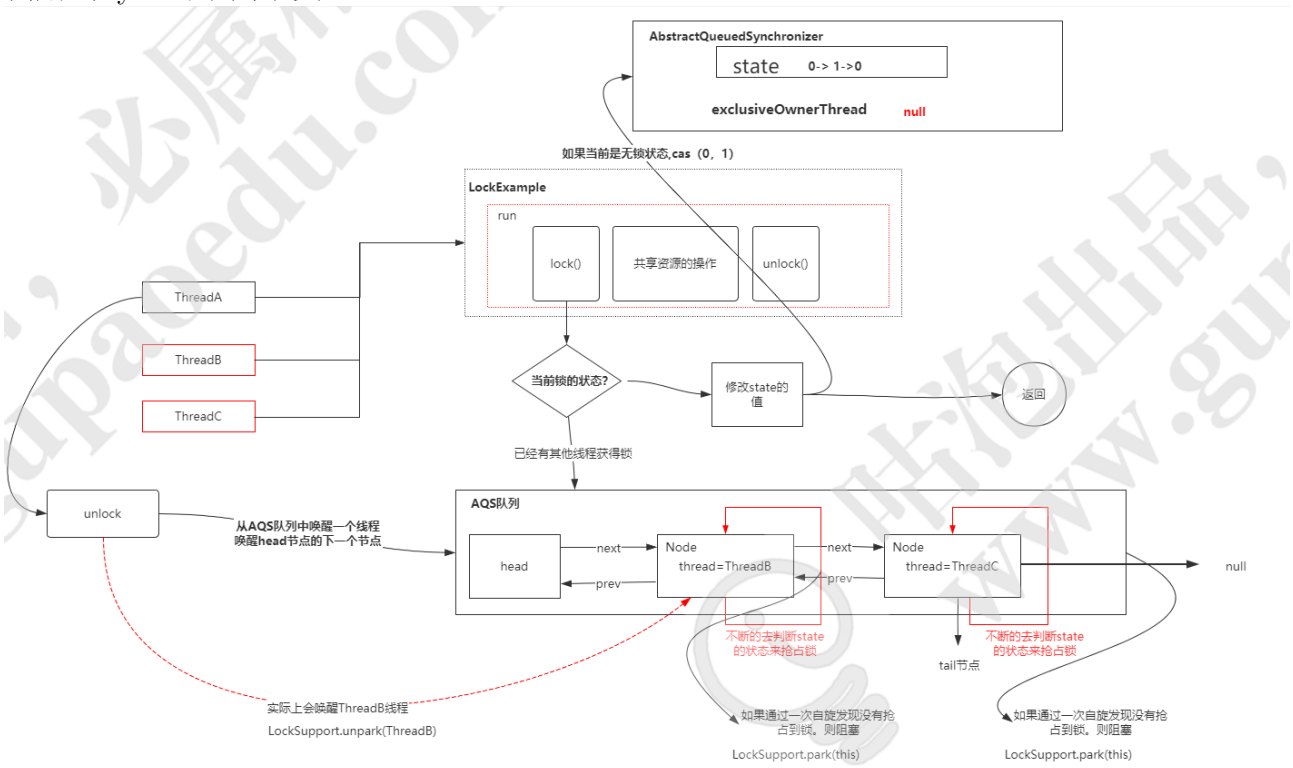
ReentrantLock 是可重入的独占锁，只能有一个线程可以获取该锁，其它获取该锁的线程会被阻塞而被放入该锁的阻塞队列里面。ReentrantLock的加锁操作:

```
// 创建非公平锁
ReentrantLock lock = new ReentrantLock();
// 获取锁操作
lock.lock();
try {
    // 执行代码逻辑
} catch (Exception ex) {
    // ...
} finally {
     // 解锁操作
    lock.unlock();
}
```

参数以及构造函数:

```
public class ReentrantLock implements Lock, java.io.Serializable {
    // 序列号
    private static final long serialVersionUID = 7373984872572414699L;
    // 同步队列
    private final Sync sync;

    public ReentrantLock() {
        // 默认非公平策略
        sync = new NonfairSync();
    }
}
```

通过分析ReentrantLock的源码，可知对其操作都转化为对Sync对象的操作，由于Sync继承了AQS，所以基本上都可以转化为对AQS的操作。如将ReentrantLock的lock函数转化为对Sync的lock函数的调用，而具体会根据采用的策略(如公平策略或者非公平策略)的不同而调用到Sync的不同子类。



new ReentrantLock()构造函数默认创建的是非公平锁NonfairSync。

公平锁FairSync:

1. 公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁

2. 公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大

```
// 公平锁
static final class FairSync extends Sync {
    // 版本序列化
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() {
        // 以独占模式获取对象，忽略中断
        acquire(1);
    }

    // 尝试公平获取锁
    protected final boolean tryAcquire(int acquires) {
        // 获取当前线程
        final Thread current = Thread.currentThread();
        // 获取状态
        int c = getState();
        if (c == 0) { // 状态为0
            if (!hasQueuedPredecessors() &&
```

```
                    compareAndSetState(0, acquires)) { // 不存在已经等待更久的线程并且比较并且设置状态成功(原子操作)
                    // 设置当前线程独占
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) { // 状态不为0，即资源已经被线程占据
                // 下一个状态
                int nextc = c + acquires;
                if (nextc < 0) // 超过了int的表示范围
                    throw new Error("Maximum lock count exceeded");
                // 设置状态
                setState(nextc);
                return true;
            }
            return false;
        }
}
```

非公平锁NonfairSync:

1. 非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁

2. 非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU 不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁

```
// 非公平锁
static final class NonfairSync extends Sync {
    // 版本号
    private static final long serialVersionUID = 7316153563782823691L;

    // 获得锁
    final void lock() {
        if (compareAndSetState(0, 1)) // 比较并设置状态成功，状态0表示锁没有被占用
            // 把当前线程设置独占了锁
            setExclusiveOwnerThread(Thread.currentThread());
        else // 锁已经被占用，或者set失败
            // 以独占模式获取对象，忽略中断
            acquire(1);
    }

    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
//以下代码是写在抽象类Sync里的
final boolean nonfairTryAcquire(int acquires) {
        // 当前线程
        final Thread current = Thread.currentThread();
        // 获取状态
        int c = getState();
        if (c == 0) { // 表示没有线程正在竞争该锁
            if (compareAndSetState(0, acquires)) { // 比较并设置状态成功，状态0表示锁没有被占用
                // 设置当前线程独占
```

```
                setExclusiveOwnerThread(current);
                return true; // 成功
            }
        }
        else if (current == getExclusiveOwnerThread()) { // 当前线程拥有该锁
            int nextc = c + acquires; // 增加重入次数
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            // 设置状态
            setState(nextc);
            // 成功
            return true;
        }
        // 失败
        return false;
    }
```

默认创建的对象lock()的时候：

- 如果锁当前没有被其它线程占用，并且当前线程之前没有获取过该锁，则当前线程会获取到该锁，然后设置当前锁的拥有者为当前线程，并设置AQS 的状态值为1 ，然后直接返回。如果当前线程之前已经获取过该锁，则这次只是简单地把AQS 的状态值加1后返回。
- 如果该锁已经被其他线程持有，非公平锁会尝试去获取锁，获取失败的话，则调用该方法线程会被放入AQS 队列阻塞挂起。

6.4.1 synchronized和ReentrantLock的区别

- 锁的实现：synchronized是Java语言的关键字，基于JVM实现。而ReentrantLock是基于JDK的API层面实现的（一般是`lock()`和`unlock()`方法配合try/finally 语句块来完成。）
- 性能：在JDK1.6锁优化以前，synchronized的性能比ReenTrantLock差很多。但是JDK6开始，增加了适应性自旋、锁消除等，两者性能就差不多了。
- ReentrantLock 比synchronized 增加了一些高级功能，如等待可中断、可实现公平锁、可实现选择性通知。
  - ReentrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制
  - ReentrantLock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。
  - synchronized与`wait()`和`notify()`/`notifyAll()`方法结合实现等待/通知机制，ReentrantLock类借助Condition接口与`newCondition()`方法实现。
  - ReentrantLock需要手工声明来加锁和释放锁，一般跟finally配合释放锁。而synchronized不用手动释放锁。

6.5 Condition

在Synchronized加锁状态时，是使用wait/notify/notifyAll进行线程间的通信。那么在使用ReentrantLock锁时，是如何实现线程间通信问题的呢？在JUC中既然提供了Lock，也提供了用作其线程间通信的方式，再次引入了Condition。

Condition要结合Lock对象实现，两者是同时存在的。准确来说先有Lock对象，然后再创

建Condition对象。线程调用这些方法时候，需要提前获取到Condition对象关联的锁，Condition对象是由Lock对象（Lock.newConditoin()）创建出来的，也就是说Condition是依赖Lock对象的。

唤醒线程：

```
    public class ConditionNotify implements Runnable {

    private Lock lock;

    private Condition condition;

    public ConditionNotify(Lock lock, Condition condition) {
        this.lock = lock;
        this.condition = condition;
    }

    @Override
    public void run() {
        try {
            lock.lock();

            System.out.println("begin - notify");
            condition.signal();
            System.out.println("end - notify");

        }catch (Exception e){

        }finally {
            lock.unlock();
        }
    }

}
```

阻塞线程：

```
public class ConditionWait implements Runnable {

    private Lock lock;

    private Condition condition;

    public ConditionWait(Lock lock, Condition condition) {
        this.lock = lock;
        this.condition = condition;
    }

    @Override
    public void run() {
        try {
            lock.lock();

            System.out.println("begin - wait");
            condition.await();
            System.out.println("end - wait");

        }catch (InterruptedException e){

        }finally {
            lock.unlock();
```

```
        }
    }
}
```

运行:

```
public class Main {

    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();

        // 保证两个线程获取的是同一把锁和同一个COndition

        new Thread(new ConditionWait(lock, condition)).start();
        new Thread(new ConditionNotify(lock, condition)).start();
    }
}
//output:
//begin - await
//begin - Notify
//end - Notify
//end - await
```

6.5.1 condition

Condition对象使用执行lock.newCondition()获取的，会创建一个ConditionObject对象返回。我们先来看一下ConditionObject的结构：

```
private transient Node firstWaiter;
private transient Node lastWaiter;
```

它有两个成员对象，分别表示头结点和尾节点构成一个单向队列，当调用await方法时，会将线程加入的这个队列中。

6.5.2 Await

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //添加到等待队列
    Node node = addConditionWaiter();
    //完整的释放锁(考虑重入问题)
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        //上下文切换（程序计数器、寄存器）用户态-内核态的转化（上下文切换）
        LockSupport.park(this); //阻塞当前线程(当其他线程调用signal()方法时，该线程会从这个位置去执行)
        //要判断当前被阻塞的线程是否是因为interrupt()唤醒
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    //重新竞争锁，savedState表示的是被释放的锁的重入次数.
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
```

```
}
```

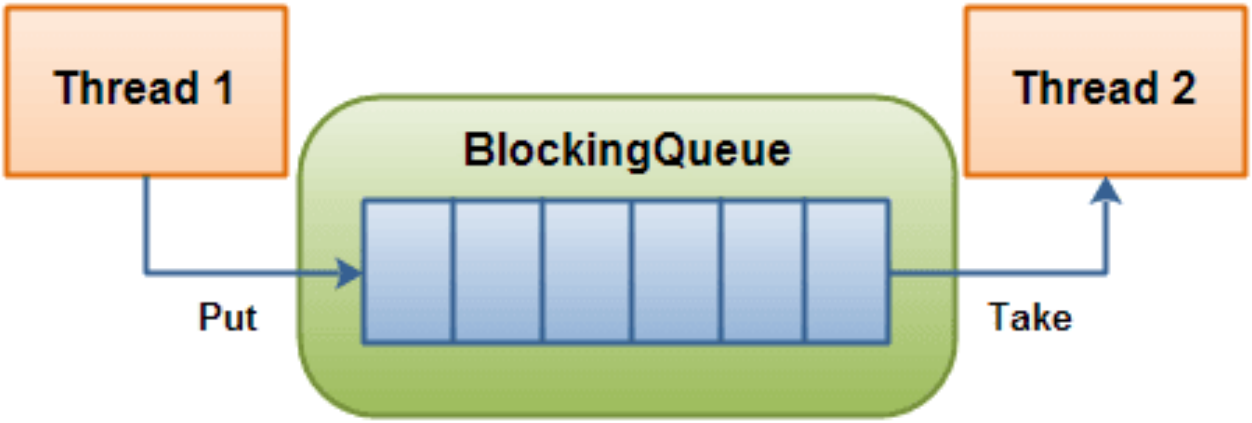
6.5.3 signal

- 要把被阻塞的线程，先唤醒(signal、signalAll)
- 把等待队列中被唤醒的线程转移到AQS队列中

```
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        // 将Condition队列中的线程移除，加入到
        doSignal(first);
}

//唤醒等待队列中的一个线程
private void doSignal(Node first) {
    do {
        if ( (firstWaiter = first.nextWaiter) == null)
        lastWaiter = null;
        first.nextWaiter = null;
        } while (!transferForSignal(first) &&
    (first = firstWaiter) != null);
}

final boolean transferForSignal(Node node) {
    //检查线程是否失效
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

     // 加入到AQS队列
    Node p = enq(node);
    int ws = p.waitStatus;
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        // 如果没有线程占用锁，就唤醒该线程
        LockSupport.unpark(node.thread);
    return true;
}
```

6.6 阻塞队列Blocking Queue

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。下图是对这个原理的阐述:



一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
        implements BlockingQueue<E>, java.io.Serializable {

    // 通过数组来实现的队列
    final Object[] items;
    //记录队首元素的下标
    int takeIndex;
    //记录队尾元素的下标
    int putIndex;
    //队列中的元素个数
    int count;
    //通过ReentrantLock来实现同步
    final ReentrantLock lock;
    //有2个条件对象，分别表示队列不为空和队列不满的情况
    private final Condition notEmpty;
    private final Condition notFull;
    //迭代器
    transient Itrs itrs;

    //offer方法用于向队列中添加数据
    public boolean offer(E e) {
        // 可以看出添加的数据不支持null值
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        //通过重入锁来实现同步
        lock.lock();
        try {
          //如果队列已经满了的话直接就返回false，不会阻塞调用这个offer方法的线程
            if (count == items.length)
                return false;
            else {
               //如果队列没有满，就调用enqueue方法将元素添加到队列中
                enqueue(e);
                return true;
            }
        } finally {
```

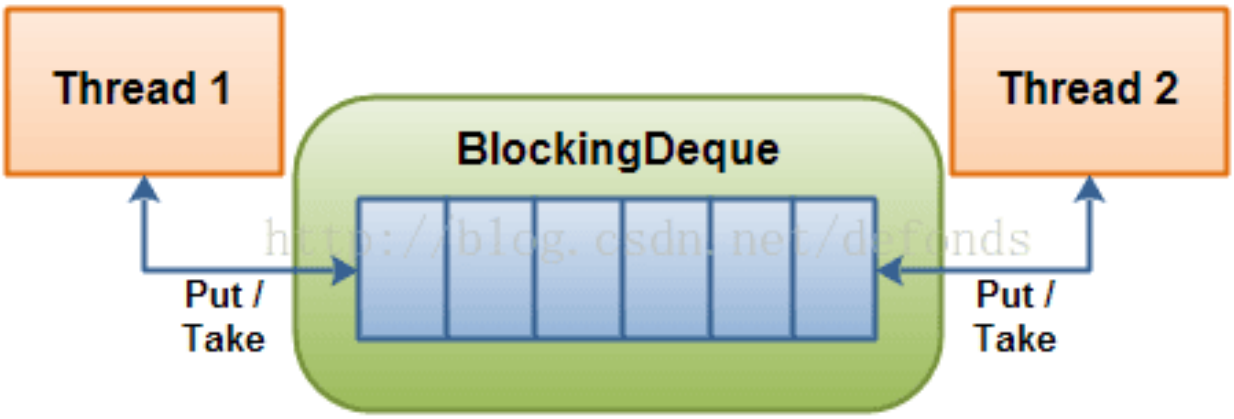
```
            lock.unlock();
        }
    }

    //将数据添加到队列中的具体方法
    private void enqueue(E x) {
        // assert lock.getHoldCount() == 1;
        // assert items[putIndex] == null;
        final Object[] items = this.items;
        items[putIndex] = x;
        //通过循环数组实现的队列，当数组满了时下标就变成0了
        if (++putIndex == items.length)
            putIndex = 0;
        count++;
        //激活因为notEmpty条件而阻塞的线程，比如调用take方法的线程
        notEmpty.signal();
    }

    //将数据从队列中取出的方法
    private E dequeue() {
        // assert lock.getHoldCount() == 1;
        // assert items[takeIndex] != null;
        final Object[] items = this.items;
        @SuppressWarnings("unchecked")
        E x = (E) items[takeIndex];
        //将对应的数组下标位置设置为null释放资源
        items[takeIndex] = null;
        if (++takeIndex == items.length)
            takeIndex = 0;
        count--;
        if (itrs != null)
            itrs.elementDequeued();
        //激活因为notFull条件而阻塞的线程，比如调用put方法的线程
        notFull.signal();
        return x;
    }

    //put方法和offer方法不一样的地方在于，如果队列是满的话，它就会把调用put方法的线程阻塞，直到队列里有空间
    public void put(E e) throws InterruptedException {
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        //因为后面调用了条件变量的await()方法，而await()方法会在中断标志设置后抛出InterruptedException异常后退出，
        // 所以在加锁时候先看中断标志是不是被设置了，如果设置了直接抛出InterruptedException异常，就不用再去获取锁了
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                //如果队列满的话就阻塞等待，直到notFull的signal方法被调用，也就是队列里有空间了
                notFull.await();
            //队列里有空间了执行添加操作
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }

```

```
92      //poll方法用于从队列中取数据，不会阻塞当前线程
93      public E poll() {
94          final ReentrantLock lock = this.lock;
95          lock.lock();
96          try {
97              //如果队列为空的话会直接返回null，否则调用dequeue方法取数据
98              return (count == 0) ? null : dequeue();
99          } finally {
100             lock.unlock();
101         }
102     }
103
104     //take方法也是用于取队列中的数据，但是和poll方法不同的是它有可能会阻塞当前的线程
105     public E take() throws InterruptedException {
106         final ReentrantLock lock = this.lock;
107         lock.lockInterruptibly();
108         try {
109             //当队列为空时，就会阻塞当前线程
110             while (count == 0)
111                 notEmpty.await();
112             //直到队列中有数据了，调用dequeue方法将数据返回
113             return dequeue();
114         } finally {
115             lock.unlock();
116         }
117     }
118
119     //返回队首元素
120     public E peek() {
121         final ReentrantLock lock = this.lock;
122         lock.lock();
123         try {
124             return itemAt(takeIndex); // null when queue is empty
125         } finally {
126             lock.unlock();
127         }
128     }
129
130     // 此外，还有一些其他方法...
131 }
```

6.7 BlockingDeque

BlockingDeque 类是一个双端队列，在不能够插入元素时，它将阻塞住试图插入元素的线程；在不能够抽取元素时，它将阻塞住试图抽取的线程。deque(双端队列) 是”Double Ended Queue” 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。

在线程既是一个队列的生产者又是这个队列的消费者的时候可以使用到BlockingDeque。如果生产者线程需要在队列的两端都可以插入数据，消费者线程需要在队列的两端都可以移除数据，这个时候也可以使用BlockingDeque。



6.8 延迟队列DelayQueue

An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired. The head of the queue is that Delayed element whose delay expired furthest in the past. If no delay has expired there is no head and poll will return null. Expiration occurs when an element's getDelay(TimeUnit.NANOSECONDS) method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using take or poll, they are otherwise treated as normal elements. For example, the size method returns the count of both expired and unexpired elements. This queue does not permit null elements.

6.9 其他阻塞队列

- ArrayBlockingQueue 基于数组结构
- LinkedBlockingQueue 基于链表结构
- PriorityBlcokingQueue 基于优先级队列
- 同步队列SynchronousQueue
  - A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one. You cannot peek at a synchronous queue because an element is only present when you try to remove it; you cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate. The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements.

6.10 CountDownLatch

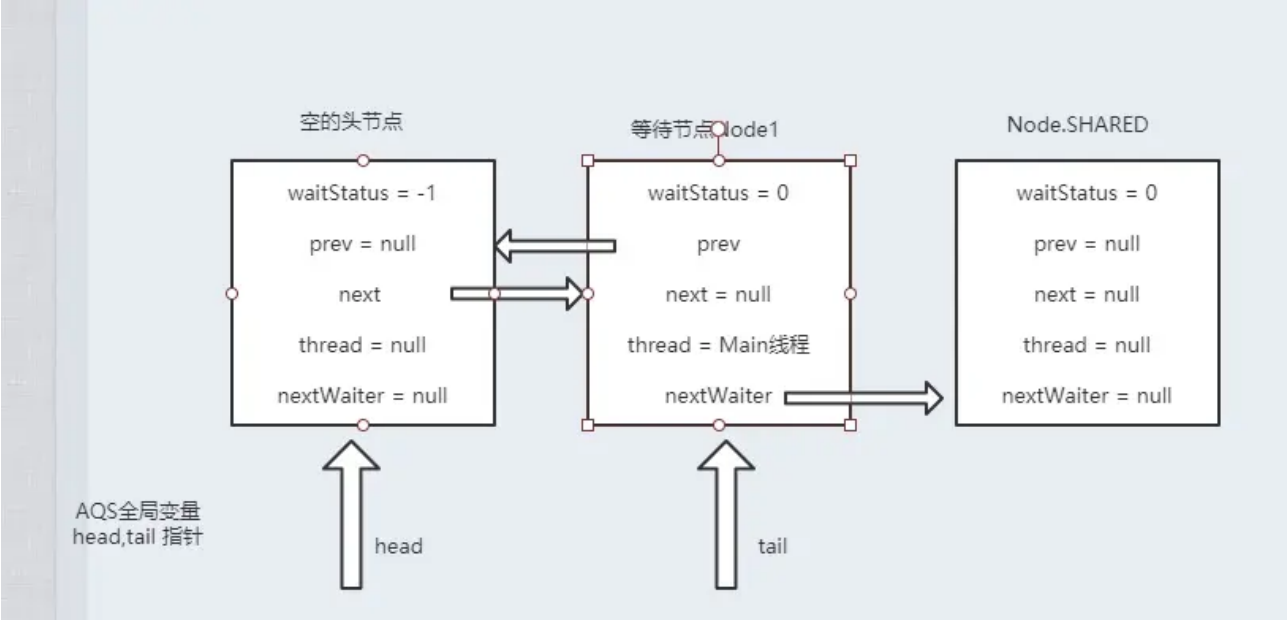
闭锁是一种同步工具类，可以延迟线程的进度，直到其到达终止状态。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭的，并且没有任何线程能够通过，当达到结束状态时，这扇门会打开并允许所有的线程通过。当闭锁到达结束状态后，将不会再改变状态，因此这扇门将永远保持打开状态。闭锁可以确保某些活动直到其他活动都完成之后

才继续执行。

CountDownLatch是一种灵活的闭锁实现，它允许一个或多个线程等待一组事件的产生。闭锁状态包括一个计数器，该计数器初始化为一个正数，表示需要等待的事件数量。countDown方法递减计数器，表示有一个事件已经发生了，而await方法会一直阻塞直到计数器为0，或者等待中的线程中断，或者等待超时。

其底层是由AQS提供支持，所以其数据结构可以参考AQS的数据结构，而AQS的数据结构核心就是两个虚拟队列：同步队列sync queue 和条件队列condition queue，不同的条件会有不同的条件队列。CountDownLatch典型的用法是将一个程序分为n个互相独立的可解决任务，并创建值为n的CountDownLatch。当每一个任务完成时，都会在这个锁存器上调用countDown，等待问题被解决的任务调用这个锁存器的await，将他们自己拦住，直至锁存器计数结束。

CountDownLatch是共享锁，当countdown归零的时候，所有调用其`await()`的线程都会同时被notify，然后同时抢占到锁。当线程执行`await()`的时候，使用AQS中`addWaiter()`的共享模式将线程加入同步队列：



ws = -1表示节点是通知状态，表示该节点出队后，必须唤醒其后续的节点线程。执行countdown并通过CAS成功设置为0的那个线程将会同时承担起唤醒队列中第一个节点线程的任务，此节点恢复执行之后，其发现状态值为通知状态，所以会唤醒后续节点。

6.11 Semaphore

Semaphore称为计数信号量，它允许n个任务同时访问某个资源，可以将信号量看做是在向外分发使用资源的许可证，只有成功获取许可证，才能使用资源。可以限制资源的访问(控流)。

说明: Semaphore与ReentrantLock的内部类的结构相同，类内部总共存在Sync、NonfairSync、FairSync三个类，NonfairSync与FairSync类继承自Sync类，Sync类继承自AbstractQueuedSynchronizer抽象类。

```
//Demo
import java.util.concurrent.Semaphore;

class MyThread extends Thread {
```

```
    private Semaphore semaphore;

    public MyThread(String name, Semaphore semaphore) {
        super(name);
        this.semaphore = semaphore;
    }

    public void run() {
        int count = 3;
        System.out.println(Thread.currentThread().getName() + " trying to acquire");
        try {
            semaphore.acquire(count);
            System.out.println(Thread.currentThread().getName() + " acquire successfully");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release(count);
            System.out.println(Thread.currentThread().getName() + " release successfully");
        }
    }
}

public class SemaphoreDemo {
    public final static int SEM_SIZE = 10;

    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(SEM_SIZE);
        MyThread t1 = new MyThread("t1", semaphore);
        MyThread t2 = new MyThread("t2", semaphore);
        t1.start();
        t2.start();
        int permits = 5;
        System.out.println(Thread.currentThread().getName() + " trying to acquire");
        try {
            semaphore.acquire(permits);
            System.out.println(Thread.currentThread().getName() + " acquire successfully");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
            System.out.println(Thread.currentThread().getName() + " release successfully");
        }
    }
}
------
//可能的output:
//main trying to acquire
//main acquire successfully
//t1 trying to acquire
//t1 acquire successfully
//t2 trying to acquire
//t1 release successfully
```

```
//main release successfully
//t2 acquire successfully
//t2 release successfully
```

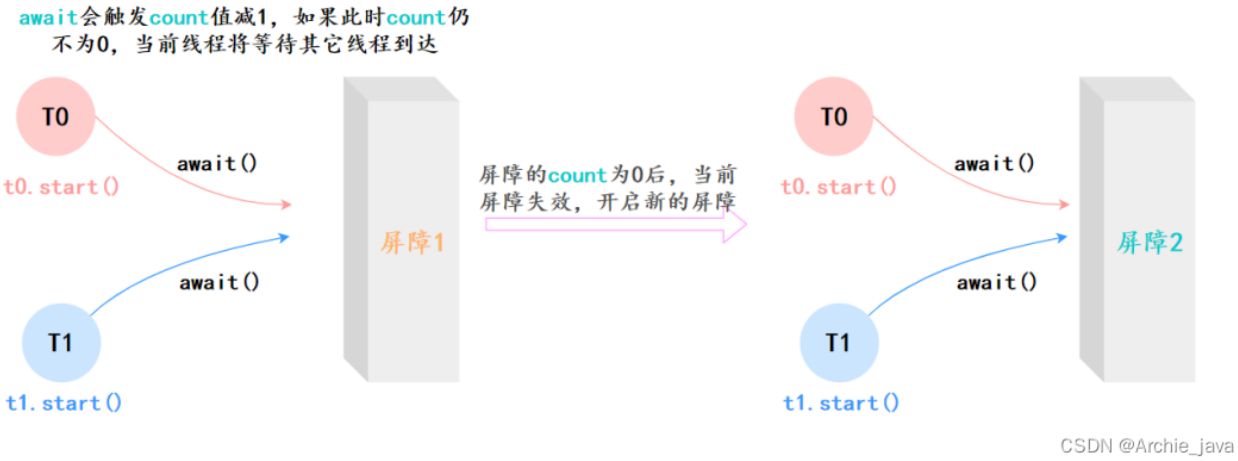
不同于CyclicBarrier和ReentrantLock，单独使用Semaphore是不会使用到AQS的条件队列的，其实，只有进行await操作才会进入条件队列，其他的都是在同步队列中，只是当前线程会被park。

6.11.1 场景问题

- semaphore初始化有10个令牌，11个线程同时各调用1次acquire方法，会发生什么？
  - 答案：拿不到令牌的线程阻塞，不会继续往下运行。
- semaphore初始化有10个令牌，一个线程重复调用11次acquire方法，会发生什么？
  - 答案：线程阻塞，不会继续往下运行。可能你会考虑类似于锁的重入的问题，很好，但是，令牌没有重入的概念。你只要调用一次acquire方法，就需要有一个令牌才能继续运行。
- semaphore初始化有1个令牌，1个线程调用一次acquire方法，然后调用两次release方法，之后另外一个线程调用acquire(2)方法，此线程能够获取到足够的令牌并继续运行吗？
  - 答案：能，原因是release方法会添加令牌，并不会以初始化的大小为准。

6.12 CyclicBarrier

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运行。



6.12.1 CyclicBarrier和CountDownLatch有什么区别

CyclicBarrier	CountDownLatch
CyclicBarrier是可重用的，其中的线程会等待所有的线程完成任务。届时，屏障将被拆除，并可以选择性地做一些特定的动作。	CountDownLatch是一次性的，不同的线程在同一个计数器上工作，直到计数器为0.
CyclicBarrier面向的是线程数	CountDownLatch面向的是任务数
在使用CyclicBarrier时，你必须在构造中指定参与协作的线程数，这些线程必须调用await()方法	使用CountDownLatch时，则必须要指定任务数，至于这些任务由哪些线程完成无关紧要
CyclicBarrier可以在所有的线程释放后重新使用	CountDownLatch在计数器为0时不能再使用
在CyclicBarrier中，如果某个线程遇到了中断、超时等问题时，则处于await的线程都会出现问题	在CountDownLatch中，如果某个线程出现问题，其他线程不受影响

7 ThreadLocal

ThreadLocal是一个将在多线程中为每一个线程创建单独的变量副本的类; 当使用ThreadLocal来维护变量时, ThreadLocal会为每个线程创建单独的变量副本, 避免因多线程操作共享变量而导致的数据不一致的情况。

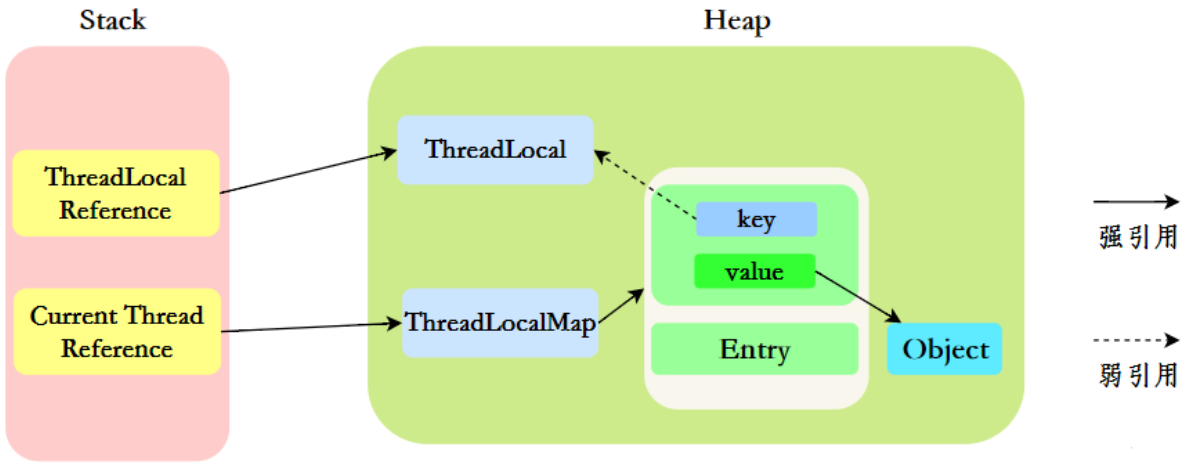
```
//创建一个ThreadLocal变量,任何一个线程都能并发访问localVariable
public static ThreadLocal<String> localVariable = new ThreadLocal<>();

//线程可以在任何地方使用localVariable，写入变量。
localVariable.set("demo");

//线程在任何地方读取的都是它写入的变量。
localVariable.get();
```

7.1 原理

7.1.1 如何实现线程隔离

- Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，每个线程都有一个属于自己的ThreadLocalMap。
- ThreadLocalMap内部维护着Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal的弱引用，value是ThreadLocal的泛型值。
- 每个线程在往ThreadLocal里设置值的时候，都是往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离。
- ThreadLocal本身不存储值，它只是作为一个key来让线程往ThreadLocalMap里存取值。

7.1.2 ThreadLocal 内存泄露



ThreadLocalMap中使用的key 为ThreadLocal 的弱引用(只要垃圾回收机制一运行，不管JVM的内存空间是否充足，都会回收该对象占用的内存)。弱引用很容易被回收，如果ThreadLocal (ThreadLocalMap的Key)被垃圾回收器回收了，但是ThreadLocalMap生命周期和Thread是一样的，它这时候如果不被回收，就会出现这种情况：ThreadLocalMap的key没了，value还在，这就会造成了内存泄漏问题。

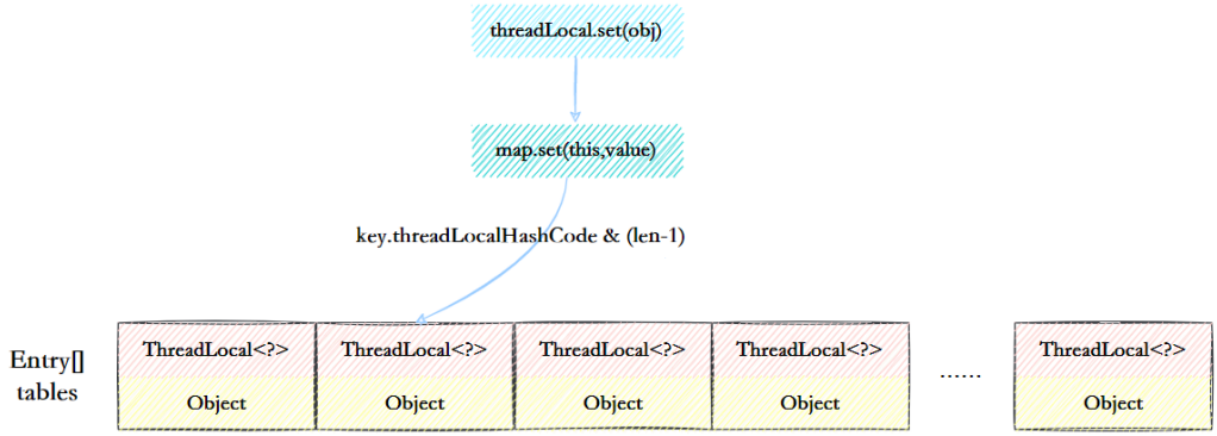
解决内存泄漏问题: 使用完ThreadLocal后，及时调用remove()方法释放内存空间。

```
ThreadLocal<String> localVariable = new ThreadLocal();
try {
    localVariable.set("demo");
    ...
} finally {
    localVariable.remove();
}
```

那为什么key还要设计成弱引用:
key设计成弱引用同样是为了防止内存泄漏。假如key被设计成强引用，如果ThreadLocal Reference (例子中的localVariable) 被销毁，此时它指向ThreadLocal的强引用就没有了，但是此时key还强引用指向ThreadLocal，就会导致ThreadLocal不能被回收，这时候就发生了内存泄漏的问题。

7.1.3 ThreadLocalMap的结构

ThreadLocalMap虽然被叫做Map，其实它是没有实现Map接口的，该方法仅仅用了一个Entry数组来存储Key, Value; Entry并不是链表形式, 而是每个bucket里面仅仅放一个Entry;




```java
//一个table数组，存储Entry类型的元素，Entry是ThreaLocal弱引用作为key，Object作为value的结构。
private Entry[] table;

//散列方法
//把对应的key映射到table数组的相应下标,取出key的threadLocalHashCode，然后和table数组长度减一&运算（相当于取余）
int i = key.threadLocalHashCode & (table.length - 1);

//每创建一个ThreadLocal对象，它就会新增0x61c88647，这个值很特殊，它是斐波那契数也叫黄金分割数。hash增量为这个数字，带来的好处就是hash 分布非常均匀。
 private static final int HASH_INCREMENT = 0x61c88647;

private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

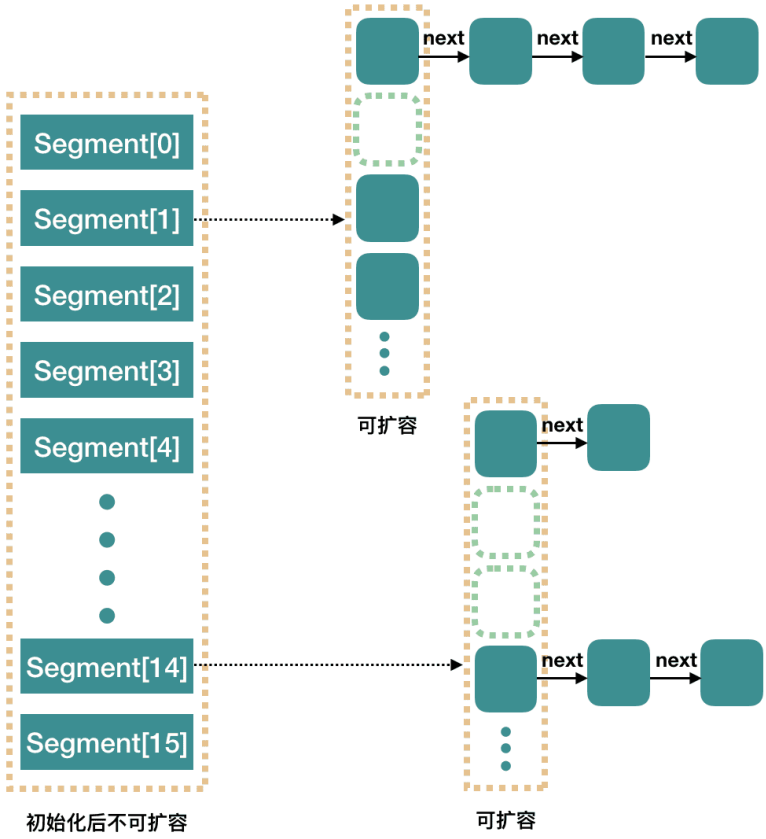
ThreadLocalMap没有使用链表，自然也不是用链地址法来解决冲突了，它用的是另外一种方式——开放定址法(Open Hashing)。开放定址法是什么意思呢？简单来说，就是这个坑被人占了，那就接着去找空着的坑。

7.2 Concurrent HashMap

7.2.1 JDK 1.7

在JDK1.5 1.7版本，Java使用了分段锁机制实现ConcurrentHashMap。简而言之，ConcurrentHashMap在对象中保存了一个Segment数组，即将整个Hash表划分为多个分段；而每个Segment元素，即每个分段则类似于一个Hashtable；这样，在执行put操作时首先根据hash算法定位到元素属于哪个Segment，然后对该Segment加锁即可。因此，ConcurrentHashMap在多线程并发编程中可是实现多线程put操作。

Java7 ConcurrentHashMap 结构



concurrencyLevel：并行级别、并发数、Segment 数。默认是16，也就是说ConcurrentHashMap有16 个Segments，所以理论上，这个时候，最多可以同时支持16 个线程并发写，只要它们的操作分别分布在不同的Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

初始化：

```
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // Find power-of-two sizes best matching arguments
    int sshift = 0;
    int ssize = 1;
    // 计算并行级别ssize，因为要保持并行级别是2 的n 次方
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    // 我们这里先不要那么烧脑，用默认值，concurrencyLevel 为16，sshift 为4
    // 那么计算出segmentShift 为28，segmentMask 为15，后面会用到这两个值
    this.segmentShift = 32 - sshift;
    this.segmentMask = ssize - 1;

    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;

    // initialCapacity 是设置整个map 初始的大小，
```

```
    // 这里根据initialCapacity 计算Segment 数组中每个位置可以分到的大小
    // 如initialCapacity 为64，那么每个Segment 或称之为"槽"可以分到4 个
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    // 默认MIN_SEGMENT_TABLE_CAPACITY 是2，这个值也是有讲究的，因为这样的话，对于具体的槽上，
    // 插入一个元素不至于扩容，插入第二个的时候才会扩容
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    while (cap < c)
        cap <<= 1;

    // 创建Segment 数组，
    // 并创建数组的第一个元素segment[0]
    Segment<K,V> s0 =
        new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                         (HashEntry<K,V>[])new HashEntry[cap]);
    Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
    // 往数组写入segment[0]
    UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
    this.segments = ss;
}
```

7.2.2 put 过程分析

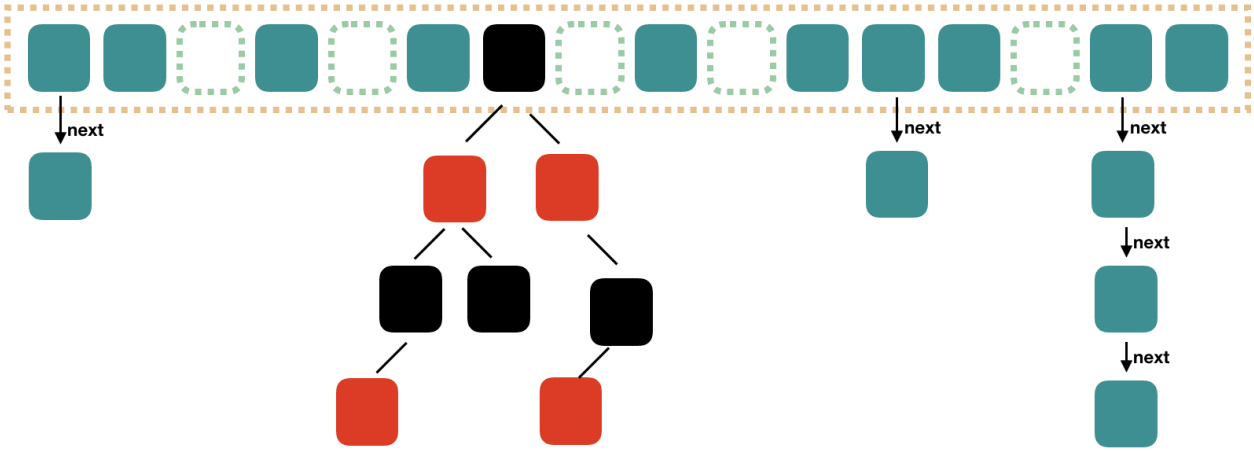
```
public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    // 1. 计算key 的hash 值
    int hash = hash(key);
    // 2. 根据hash 值找到Segment 数组中的位置j
    // hash 是32 位，无符号右移segmentShift(28) 位，剩下高4 位，
    // 然后和segmentMask(15) 做一次与操作，也就是说j 是hash 值的高4 位，也就是槽的数组下标
    int j = (hash >>> segmentShift) & segmentMask;
    // 刚刚说了，初始化的时候初始化了segment[0]，但是其他位置还是null，
    // ensureSegment(j) 对segment[j] 进行初始化
    if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
         (segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment
        s = ensureSegment(j);
    // 3. 插入新值到槽s 中
    return s.put(key, hash, value, false);
}

final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 在往该segment 写入前，需要先获取该segment 的独占锁
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        // 这个是segment 内部的数组
        HashEntry<K,V>[] tab = table;
        // 再利用hash 值，求应该放置的数组下标
        int index = (tab.length - 1) & hash;
        // first 是该数组的链表的表头
        HashEntry<K,V> first = entryAt(tab, index);

        // for循环，插入或更新值（省略）
        //...
```

```
    } finally {
        // 解锁
        unlock();
    }
    return oldValue;
}
```

7.2.3 JDK1.8

在JDK1.8中，ConcurrentHashMap的实现原理摒弃了这种设计，而是选择了与HashMap类似的数组+链表+红黑树的方式实现，而加锁则采用CAS和synchronized实现。

Java8 ConcurrentHashMap 结构




```
    /* ---------------- Fields -------------- */

    /**
     * The array of bins. Lazily initialized upon first insertion.
     * Size is always a power of two. Accessed directly by iterators.
     */
    transient volatile Node<K,V>[] table;

    /**
     * The next table to use; non-null only while resizing.
     */
    private transient volatile Node<K,V>[] nextTable;

    /**
     * Base counter value, used mainly when there is no contention,
     * but also as a fallback during table initialization
     * races. Updated via CAS.
     */
    private transient volatile long baseCount;

    /**
     * Table initialization and resizing control.  When negative, the
     * table is being initialized or resized: -1 for initialization,
     * else -(1 + the number of active resizing threads).  Otherwise,
     * when table is null, holds the initial table size to use upon
     * creation, or 0 for default. After initialization, holds the
     * next element count value upon which to resize the table.
     */
    private transient volatile int sizeCtl;

```

```
    /**
     * The next table index (plus one) to split while resizing.
     */
    private transient volatile int transferIndex;

    /**
     * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
     */
    private transient volatile int cellsBusy;

    /**
     * Table of counter cells. When non-null, size is a power of 2.
     */
    private transient volatile CounterCell[] counterCells;
```

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 得到hash 值
    int hash = spread(key.hashCode());
    // 用于记录相应链表的长度
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果数组"空"，进行数组初始化
        if (tab == null || (n = tab.length) == 0)
            // 初始化数组，后面会详细介绍
            tab = initTable();

        // 找该hash 值对应的数组下标，得到第一个节点f
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果数组该位置为空，
            // 用一次CAS 操作将这个新值放入其中即可，这个put 操作差不多就结束了，可以拉到最后面了
            // 如果CAS 失败，那就是有并发操作，进到下一个循环就好了
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // hash 居然可以等于MOVED，这个需要到后面才能看明白，不过从名字上也能猜到，肯定是因为在扩容
        else if ((fh = f.hash) == MOVED)
            // 帮助数据迁移，这个等到看完数据迁移部分的介绍后，再理解这个就很简单了
            tab = helpTransfer(tab, f);

        else { // 到这里就是说，f 是该位置的头节点，而且不为空

            V oldVal = null;
            // 获取数组该位置的头节点的监视器锁
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) { // 头节点的hash 值大于0，说明是链表
                        // 用于累加，记录链表的长度
                        binCount = 1;
                        // 遍历链表
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 如果发现了"相等"的key，判断是否要进行值覆盖，然后也就可以break 了
```

```
                                if (e.hash == hash &&
                                    ((ek = e.key) == key ||
                                     (ek != null && key.equals(ek)))) {
                                    oldVal = e.val;
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                // 到了链表的最末端，将这个新值放到链表的最后面
                                Node<K,V> pred = e;
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                                              value, null);
                                    break;
                                }
                            }
                        }
                        else if (f instanceof TreeBin) { // 红黑树
                            Node<K,V> p;
                            binCount = 2;
                            // 调用红黑树的插值方法插入新节点
                            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                                  value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }

                if (binCount != 0) {
                    // 判断是否要将链表转换为红黑树，临界值和HashMap 一样，也是8
                    if (binCount >= TREEIFY_THRESHOLD)
                        // 这个方法和HashMap 中稍微有一点点不同，那就是它不是一定会进行红黑树转换，
                        // 如果当前数组的长度小于64，那么会选择进行数组扩容，而不是转换为红黑树
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        //
        addCount(1L, binCount);
        return null;
    }
```

7.2.4 扩容: tryPresize

```
// 首先要说明的是，方法参数size 传进来的时候就已经翻了倍了
private final void tryPresize(int size) {
    // c:resize后的size为原先size 的1.5 倍，再加1，再往上取最近的2 的n 次方。
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    int sc;//next resize threshold
    while ((sc = sizeCtl) >= 0) {
```

```
        Node<K,V>[] tab = table; int n;

        // 如果没有数组，则初始化数组
        if (tab == null || (n = tab.length) == 0) {
            n = (sc > c) ? sc : c;
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    if (table == tab) {
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = nt;
                        sc = n - (n >>> 2); // 0.75 * n
                    }
                } finally {
                    sizeCtl = sc;
                }
            }
        }
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        else if (tab == table) {
            int rs = resizeStamp(n);

            if (sc < 0) {
                Node<K,V>[] nt;
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 2. 用CAS 将sizeCtl 加1，然后执行transfer 方法
                // 此时nextTab 不为null
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // resizeStamp function ensures sizeCtl gets a negative value if the below
    statement executed successfully
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                // 调用transfer 方法，此时nextTab 参数为null
                transfer(tab, null);
        }
    }
}
```

7.2.5 数据迁移: transfer

7.2.6 size()

8 Fork/Join

Fork/Join框架是Java并发工具包中的一种可以将一个大任务拆分为很多小任务来异步执行的工具，自JDK1.7引入。

Fork/Join框架主要包含三个模块:

- 任务对象: ForkJoinTask (包括RecursiveTask、RecursiveAction 和CountedCompleter)
- 执行Fork/Join任务的线程: ForkJoinWorkerThread

- 线程池: ForkJoinPool

这三者的关系是: ForkJoinPool可以通过池中的ForkJoinWorkerThread来处理ForkJoinTask任务。
ForkJoinPool 只接收ForkJoinTask 任务(在实际使用中，也可以接收Runnable/Callable 任务，但在真正运行时，也会把这些任务封装成ForkJoinTask 类型的任务)，RecursiveTask是ForkJoinTask 的子类，是一个可以递归执行的ForkJoinTask，RecursiveAction 是一个无返回值的RecursiveTask，CountedCompleter 在任务完成执行后会触发执行一个自定义的钩子函数。

8.1 核心思想

核心思想: 分治算法(Divide-and-Conquer)：把任务递归的拆分为各个子任务，这样可以更好的利用系统资源，尽可能的使用所有可用的计算能力来提升应用性能。

核心思想: work-stealing(工作窃取)算法: 大任务拆成了若干个小任务，把这些小任务放到不同的队列里，各自创建单独线程来执行队列里的任务。

那么问题来了，有的线程干活块，有的线程干活慢。干完活的线程不能让它空下来，得让它去帮没干完活的线程干活。它去其它线程的队列里窃取一个任务来执行，这就是所谓的工作窃取。

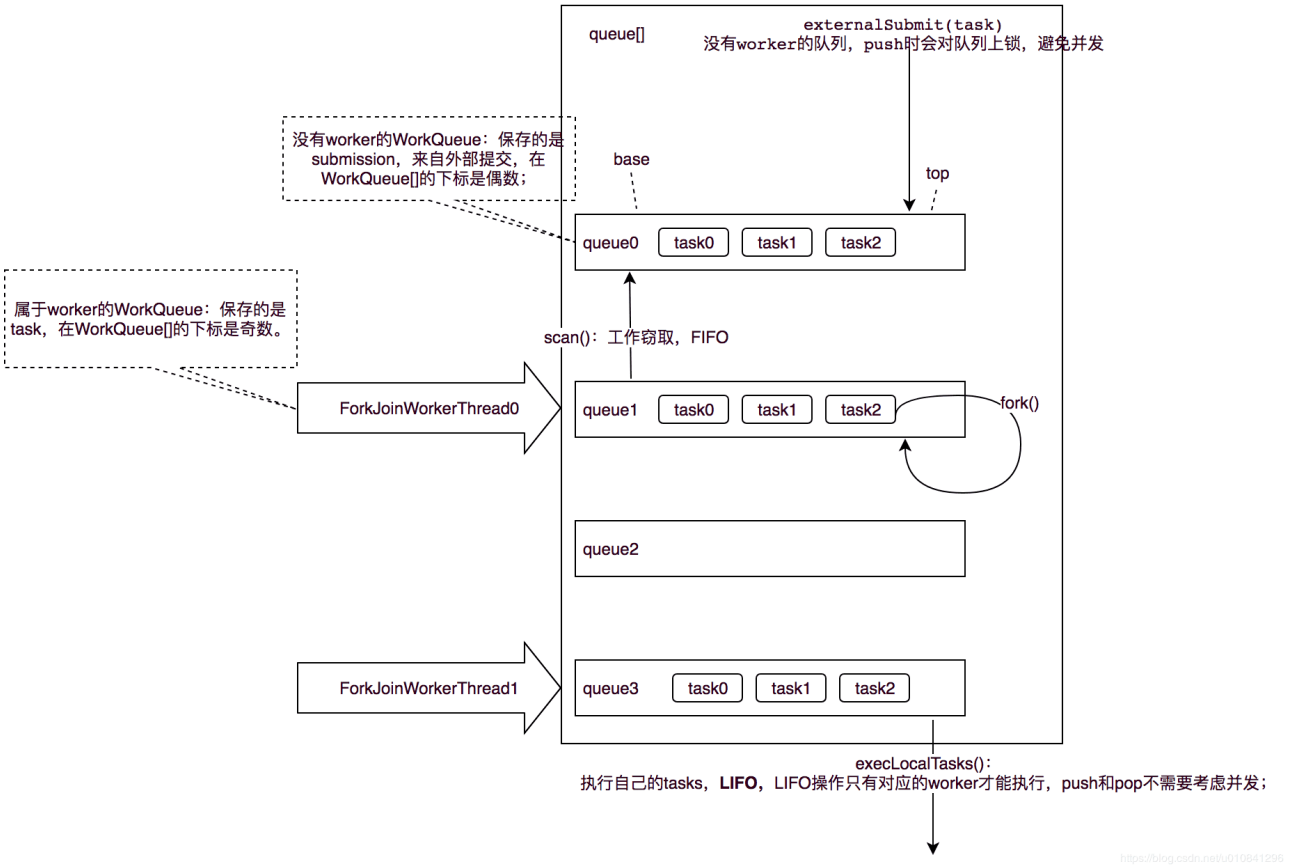
工作窃取发生的时候，它们会访问同一个队列，为了减少窃取任务线程和被窃取任务线程之间的竞争，通常任务会使用双端队列，被窃取任务线程永远从双端队列的头部拿，而窃取任务的线程永远从双端队列的尾部拿任务执行。

1、ForkJoinPool使用数组保存所有WorkQueue（下文经常出现的WorkQueue[]），

2、每个worker有属于自己的WorkQueue，但不是每个WorkQueue都有对应的worker。

2、WorkQueue是一个双端队列，同时支持LIFO(last-in-first-out)的push和pop操作，

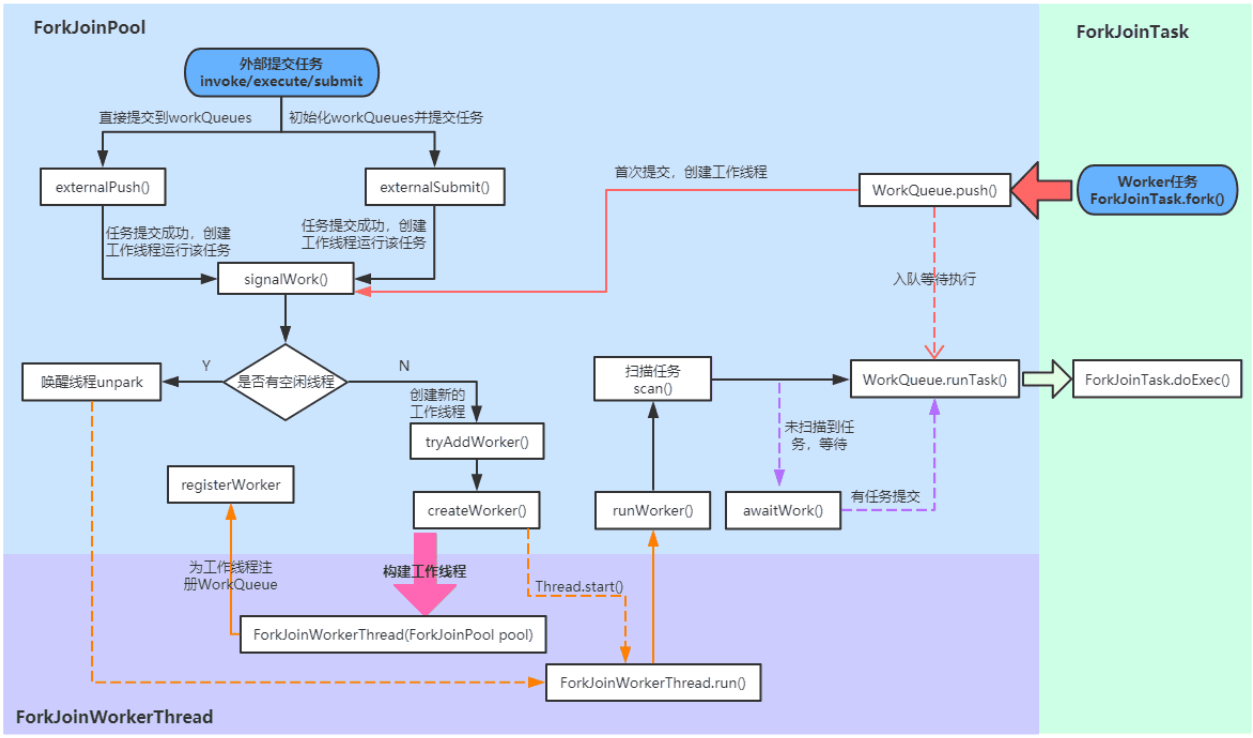
和FIFO(first-in-first-out)的poll操作，分别操作top端和base端。



8.2 Fork/Join 框架的执行流程

上图可以看出ForkJoinPool 中的任务执行分两种:

- 直接通过FJP 提交的外部任务(external/submissions task)，存放在workQueues 的偶数槽位;
- 通过内部fork 分割的子任务(Worker task)，存放在workQueues 的奇数槽位。



8.3 执行流程- 外部任务(external/submissions task)提交

向ForkJoinPool 提交任务有三种方式:

- invoke()会等待任务计算完毕并返回计算结果;
- execute()是直接向池提交一个任务来异步执行，无返回结果;
- submit()也是异步执行，但是会返回提交的任务，在适当的时候可通过task.get()获取执行结果。

这三种提交方式都都是调用externalPush()方法来完成, externalPush的执行流程很简单: 首先找到一个随机偶数槽位的workQueue，然后把任务放入这个workQueue 的任务数组中，并更新top位。如果队列的剩余任务数小于1，则尝试创建或激活一个工作线程来运行任务(防止在externalSubmit初始化时发生异常导致工作线程创建失败)。

8.4 执行流程: 子任务(Worker task)提交

子任务的提交相对比较简单，由任务的fork()方法完成。通过上面的流程图可以看到任务被分割(fork)之后调用了ForkJoinPool.WorkQueue.push()方法直接把任务放到队列中等待被执行

如果当前线程是Worker 线程，说明当前任务是fork分割的子任务，通过ForkJoinPool.workQueue.push方法直接把任务放到自己的等待队列中；否则调用ForkJoinPool.externalPush()提交到一个随机的等待队列中(外部任务)。

首先把任务放入等待队列并更新top位；如果当前WorkQueue 为新建的等待队列(top-base¡=1)，则调用signalWork方法为当前WorkQueue 新建或唤醒一个工作线程；如果WorkQueue中的任务数组容量过小，则调用growArray()方法对其进行两倍扩容，growArray()方法源码如下:

8.5 执行流程: 任务执行

在ForkJoinPool .createWorker()方法中创建工作线程后，会启动工作线程，系统为工作线程分配到CPU执行时间片之后会执行ForkJoinWorkerThread 的run()方法正式开始执行任务。

方法很简单，在工作线程运行前后会调用自定义钩子函数(onStart和onTermination)，任务的运行则是调用了ForkJoinPool.runWorker()。如果全部任务执行完毕或者期间遭遇异常，则通过ForkJoinPool.deregisterWorker关闭工作线程并处理异常信息(deregisterWorker方法我们后面会详细讲解)。

runWorker是ForkJoinWorkerThread 的主运行方法，用来依次执行当前工作线程中的任务。函数流程很简单: 调用scan方法依次获取任务，然后调用WorkQueue .runTask运行任务；如果未扫描到任务，则调用awaitWork等待，直到工作线程/线程池终止或等待超时。

scan, 扫描并尝试偷取一个任务。使用w.hint进行随机索引WorkQueue，也就是说并不一定会执行当前WorkQueue 中的任务，而是偷取别的Worker的任务来执行。如果scan方法未扫描到任务，会调用awaitWork等待获取任务。在scan方法扫描到任务之后，调用WorkQueue.runTask()来执行获取到的任务

deregisterWorker方法用于工作线程运行完毕之后终止线程或处理工作线程异常，主要就是清除已关闭的工作线程或回滚创建线程之前的操作，并把传入的异常抛给ForkJoinTask 来处理。

8.6 获取任务结果- ForkJoinTask.join() / ForkJoinTask.invoke()

ForkJoinTask的join()和invoke()方法都可以用来获取任务的执行结果(另外还有get方法也是调用了doJoin来获取任务结果，但是会响应运行时异常)，它们对外部提交任务的执行方式一致，都是通过externalAwaitDone方法等待执行结果。不同的是invoke()方法会直接执行当前任务；而join()方法则是在当前任务在队列top 位时(通过tryUnpush方法判断)才能执行，如果当前任务不在top 位或者任务执行失败调用ForkJoinPool.awaitJoin方法帮助执行或阻塞当前join 任务。(所以在官方文档中建议了我们对ForkJoinTask任务的调用顺序，一对fork-join操作一般按照如下顺序调用: a.fork(); b.fork(); b.join(); a.join();。因为任务b 是后面进入队列，也就是说它是在栈顶的(top 位)，在它fork()之后直接调用join()就可以直接执行而不会调用ForkJoinPool.awaitJoin方法去等待。)

如果当前join 任务不在Worker等待队列的top位，或者任务执行失败，调用awaitJoin方法来帮助执行或阻塞当前join 的任务。由于每次调用awaitJoin都会优先执行当前join的任务，所以首先会更新currentJoin为当前join任务

8.7 示例

采用Fork/Join来异步计算1+2+3+...+10000的结果

```
public class Test {
  static final class SumTask extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1L;

    final int start; //开始计算的数
    final int end; //最后计算的数

    SumTask(int start, int end) {
      this.start = start;
      this.end = end;
```

```
    }

    @Override
    protected Integer compute() {
      //如果计算量小于1000，那么分配一个线程执行if中的代码块，并返回执行结果
      if(end - start < 1000) {
        System.out.println(Thread.currentThread().getName() + "begin" +
   start + "-" + end);
        int sum = 0;
        for(int i = start; i <= end; i++)
          sum += i;
        return sum;
      }
      //如果计算量大于1000，那么拆分为两个任务
      SumTask task1 = new SumTask(start, (start + end) / 2);
      SumTask task2 = new SumTask((start + end) / 2 + 1, end);
      //执行任务
      task1.fork();
      task2.fork();
      //获取任务执行的结果
      return task1.join() + task2.join();
    }
  }

  public static void main(String[] args) throws InterruptedException,
   ExecutionException {
    ForkJoinPool pool = new ForkJoinPool();
    ForkJoinTask<Integer> task = new SumTask(1, 10000);
    pool.submit(task);
    System.out.println(task.get());
  }
}
-----------------------------------
//output:
//ForkJoinPool-1-worker-1 开始执行: 1-625
//ForkJoinPool-1-worker-7 开始执行: 6251-6875
//ForkJoinPool-1-worker-6 开始执行: 5626-6250
//ForkJoinPool-1-worker-10 开始执行: 3751-4375
//ForkJoinPool-1-worker-13 开始执行: 2501-3125
//ForkJoinPool-1-worker-8 开始执行: 626-1250
//ForkJoinPool-1-worker-11 开始执行: 5001-5625
//ForkJoinPool-1-worker-3 开始执行: 7501-8125
//ForkJoinPool-1-worker-14 开始执行: 1251-1875
//ForkJoinPool-1-worker-4 开始执行: 9376-10000
//ForkJoinPool-1-worker-8 开始执行: 8126-8750
//ForkJoinPool-1-worker-0 开始执行: 1876-2500
//ForkJoinPool-1-worker-12 开始执行: 4376-5000
//ForkJoinPool-1-worker-5 开始执行: 8751-9375
//ForkJoinPool-1-worker-7 开始执行: 6876-7500
//ForkJoinPool-1-worker-1 开始执行: 3126-3750
//50005000
```

9 线程池

- 它帮我们管理线程，避免增加创建线程和销毁线程的资源损耗。因为线程其实也是一个对象，创建一个对象，需要经过类加载过程，销毁一个对象，需要走GC垃圾回收流

程，都是需要资源开销的。

- 提高响应速度: 如果任务到达了，相对于从线程池拿线程，比起重新去创建一条线程执行，速度快很多。
- 重复利用。线程用完，再放回池子，可以达到重复利用的效果，节省资源。

ThreadPoolExecutor使用详解:
线程池由一个线程集合workerSet和一个阻塞队列workQueue组成。当用户向线程池提交一个任务(也就是线程)时，线程池会先将任务放入workQueue中。workerSet中的线程会不断的从workQueue中获取线程然后执行。当workQueue中没有任务的时候，worker就会阻塞，直到队列中有任务了就取出来继续执行。

9.1 Execute原理

当一个任务提交至线程池之后:

1. 线程池首先当前运行的线程数量是否少于corePoolSize。如果是，则创建一个新的工作线程来执行任务。如果都在执行任务，则进入2.
2. 判断BlockingQueue是否已经满了，倘若还没有满，则将线程放入BlockingQueue。否则进入3.
3. 如果创建一个新的工作线程将使当前运行的线程数量超过maximumPoolSize，则交给RejectedExecutionHandler来处理任务。

当ThreadPoolExecutor创建新线程时，通过CAS来更新线程池的状态ctl.

9.2 参数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

- corePoolSize: 线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize, 即使有其他空闲线程能够执行新来的任务, 也会继续创建线程；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的`prestartAllCoreThreads()`方法，线程池会提前创建并启动所有核心线程。
- maximumPoolSize: 线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize；当阻塞队列是无界队列, 则maximumPoolSize则不起作用, 因为无法提交至核心线程池的线程会一直持续地放入workQueue.
- keepAliveTime: 线程空闲时的存活时间，即当线程没有任务执行时，该线程继续存活的时间；默认情况下，该参数只在线程数大于corePoolSize时才有用, 超过这个时间的空闲线程将被终止;
- unit: keepAliveTime的单位
- workQueue: 用来保存等待被执行的任务的阻塞队列(使用JUC中的Blocking Queue)

- handler: 线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略:
  - AbortPolicy: 直接抛出异常，默认策略;
  - CallerRunsPolicy: 用调用者所在的线程来执行任务;
  - DiscardOldestPolicy: 丢弃阻塞队列中靠最前的任务，并执行当前任务;
  - DiscardPolicy: 直接丢弃任务;

9.3 预设的类型

9.3.1 newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

线程池的线程数量达corePoolSize后，即使线程池没有可执行任务时，也不会释放线程。

FixedThreadPool的工作队列为无界队列LinkedBlockingQueue(队列容量为`Integer.MAX_VALUE`), 这会导致以下问题:

- 线程池里的线程数量不超过corePoolSize,这导致了maximumPoolSize和keepAliveTime将会是个无用参数
- 由于使用了无界队列, 所以FixedThreadPool永远不会拒绝, 即饱和策略失效

9.3.2 newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行.由于使用了无界队列, 所以SingleThreadPool永远不会拒绝, 即饱和策略失效

9.3.3 newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

线程池的线程数可达到`Integer.MAX_VALUE`，即2147483647，内部使用SynchronousQueue作为阻塞队列；和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；执行过程与前两种稍微不同:

- 主线程调用SynchronousQueue的`offer()`方法放入task, 倘若此时线程池中有空闲的线程尝试读取SynchronousQueue的task, 即调用了SynchronousQueue的`poll()`, 那么主线程将该task交给空闲线程. 否则执行(2)
- 当线程池为空或者没有空闲的线程, 则创建新的线程执行任务.
- 执行完任务的线程倘若在60s内仍空闲, 则会被终止. 因此长时间空闲的CachedThreadPool不会持有任何线程资源

9.3.4 关闭线程池

遍历线程池中的所有线程，然后逐个调用线程的interrupt方法来中断线程

- 关闭方式- shutdown: 将线程池里的线程状态设置成SHUTDOWN状态, 然后中断所有没有正在执行任务的线程
- 关闭方式- shutdownNow: 将线程池里的线程状态设置成STOP状态, 然后停止所有正在执行或暂停任务的线程. 只要调用这两个关闭方法中的任意一个, `isShutDown()` 返回true. 当所有任务都成功关闭了, `isTerminated()`返回true

9.3.5 任务的执行

线程池的工作线程通过Woker类实现，在ReentrantLock锁的保证下，把Woker实例插入到HashSet后，并启动Woker中的线程。从Woker类的构造方法实现可以发现: 线程工厂在创建线程thread时，将Woker实例本身this作为参数传入，当执行start方法启动线程thread时，本质是执行了Worker的runWorker方法。firstTask执行完成之后，通过getTask方法从阻塞队列中获取等待的任务，如果队列中没有任务，getTask方法会被阻塞并挂起，不会占用cpu资源;

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
    //workerCountOf获取线程池的当前线程数；小于corePoolSize，执行addWorker创建新线程执行command任务
        if (addWorker(command, true))
            return;
```

```
29          c = ctl.get();
30      }
31      // double check:  c, recheck
32      // 线程池处于RUNNING状态，把提交的任务成功放入阻塞队列中
33      if (isRunning(c) && workQueue.offer(command)) {
34          int recheck = ctl.get();
35          // recheck and if necessary 回滚到入队操作前，即倘若线程池shutdown状态，
    就remove(command)
36          //如果线程池没有RUNNING，成功从阻塞队列中删除任务，执行reject方法处理任务
37          if (! isRunning(recheck) && remove(command))
38              reject(command);
39          //线程池处于running状态，但是没有线程，则创建线程
40          else if (workerCountOf(recheck) == 0)
41              addWorker(null, false);
42      }
43      // 往线程池中创建新的线程失败，则reject任务
44      else if (!addWorker(command, false))
45          reject(command);
46  }
```

9.3.6 addWorker方法

```
1   private final ReentrantLock mainLock = new ReentrantLock();
2   private boolean addWorker(Runnable firstTask, boolean core) {
3       // CAS更新线程池数量
4       retry:
5       for (;;) {
6           int c = ctl.get();
7           int rs = runStateOf(c);
8
9           // Check if queue empty only if necessary.
10          if (rs >= SHUTDOWN &&
11              ! (rs == SHUTDOWN &&
12                 firstTask == null &&
13                 ! workQueue.isEmpty()))
14              return false;
15
16          for (;;) {
17              int wc = workerCountOf(c);
18              if (wc >= CAPACITY ||
19                  wc >= (core ? corePoolSize : maximumPoolSize))
20                  return false;
21              if (compareAndIncrementWorkerCount(c))
22                  break retry;
23              c = ctl.get();  // Re-read ctl
24              if (runStateOf(c) != rs)
25                  continue retry;
26              // else CAS failed due to workerCount change; retry inner loop
27          }
28      }
29
30      boolean workerStarted = false;
31      boolean workerAdded = false;
32      Worker w = null;
33      try {
34          w = new Worker(firstTask);
35          final Thread t = w.thread;
36          if (t != null) {
37              // 线程池重入锁
```

```
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                t.start();  // 线程启动，执行任务(Worker.thread(firstTask).start());
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

9.3.7 Worker类的runworker方法

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        // 先执行firstTask，再从workerQueue中取task(getTask())

        while (task != null || (task = getTask()) != null) {
            w.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            if ((runStateAtLeast(ctl.get(), STOP) ||
                    (Thread.interrupted() &&
                    runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
```

```
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

- 线程启动之后，通过unlock方法释放锁，设置AQS的state为0，表示运行可中断；
- Worker执行firstTask或从workQueue中获取任务:
  - 进行加锁操作，保证thread不被其他线程中断(除非线程池被中断)
  - 检查线程池状态，倘若线程池处于中断状态，当前线程将中断。
  - 执行beforeExecute
  - 执行任务的run方法
  - 执行afterExecute方法
  - 解锁操作

9.3.8 getTask

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {//自旋
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
```

```
                continue;
            }

            try {
            // allowCoreThreadTimeOut为false，线程即使空闲也不会被销毁；倘若为true，
    在keepAliveTime内仍空闲则会被销毁。
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
}
```

- 如果线程允许空闲等待而不被销毁timed == false，workQueue.take任务：如果阻塞队列为空，当前线程会被挂起等待；当队列中有任务加入时，线程被唤醒，take方法返回任务，并执行；
- 如果线程不允许无休止空闲timed == true, workQueue.poll任务：如果在keepAliveTime时间内，阻塞队列还是没有任务，则返回null；

9.3.9 submit

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

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    // 通过submit方法提交的Callable任务会被封装成了一个FutureTask对象。
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

通过submit方法提交的Callable任务会被封装成了一个FutureTask对象。通过Executor.execute方法提交FutureTask到线程池中等待被执行，最终执行的是FutureTask的run方法

9.3.10 任务的提交

```
public class Test{
    public static void main(String[] args) {

        ExecutorService es = Executors.newCachedThreadPool();
        Future<String> future = es.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return "future result";
            }
        });
```

```
        try {
            String result = future.get();
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在实际业务场景中，Future和Callable基本是成对出现的，Callable负责产生结果，Future负责获取结果。

- Callable接口类似于Runnable，只是Runnable没有返回值。
- Callable任务除了返回正常结果之外，如果发生异常，该异常也会被返回，即Future可以拿到异步执行任务各种结果；
- Future.get方法会导致主线程阻塞，直到Callable任务执行完成；