# 基本概念

## 程序

可执行文件

## 进程

操作系统进行资源分配的基本单位，进程是静态的概念(静态资源:CPU,内存,外部设备等)

## 线程

线程是调度执行的基本单位，多个线程共享同一个进程里面的所有的资源，线程是动态的概念。一个程序在运行的时候会产生不同的执行路径(同时在运行的多条路径)就叫多线程。双击可执行文件后会进入内存，变成一个进程，进程是如何执行的呢？(程序是如何执行的)真正开始执行是以线程为单位来执行，操作系统会找到主线程，然后将主线程的代码一条一条交给CPU来执行。如果主线程运行中开启了其他线程，再来线程之间的来回切换。

## 线程切换(Context Switch)

CPU的组成包括如下几个部分：

1. ALU:计算单元
2. Registers:寄存器组,用于存储数据
3. PC:程序计数器（Program Counter）也是一种寄存器，保存了将要取出的下一条指令的内存地址，指令取出后，就会更新该寄存器指向下一条指令
4. Cache:用来缓存内存中的数据，避免直接从内存中获取，提升CPU的运算周期效率。

程序执行过程：指令放PC，数据放Registers，然后ALU进行计算；线程切换则是将PC和Registers中正在执行的线程的数据保存到Cache中，然后将另一个线程的数据load到CPU的PC和Registers中来。CPU只负责计算，操作系统负责具体指令和数据是属于哪个线程的。

设置线程数量的一般计算公式：N_threads = N_cpu * U_cpu * (1 + W/C)

1. N_cpu是处理器的核的数目，可以通过Runtime.getRuntime().availableProcessors()得到
2. U_cpu是期望的CPU利用率(该值应该介于0到1之间)
3. W/C是等待时间与计算时间的比率

## 纤程/协程

绿色线程，用户管理的(而不是OS管理的)线程

## 并发(Concurrent)

当有多个线程在操作时,如果系统只有一个CPU,则它根本不可能真正同时进行一个以上的线程,它只能把CPU运行时间划分成若干个时间段,再将时间段分配给各个线程执行,在一个时间段的线程代码运行时,其它线程处于挂起状态.这种方式我们称之为并发(Concurrent).

## 并行(Parallel)

当系统有一个以上CPU时,则线程的操作有可能非并发.当一个CPU执行一个线程时,另一个CPU可以执行另一个线程,两个线程互不抢占CPU资源,可以同时进行,这种方式我们称之为并行(Parallel)

## 调用状态

阻塞（Blocking）和非阻塞（Non-Blocking）

## 通信机制

同步（synchronous）和异步（asynchronous）

## 锁

monitor

## 临界区

critical section 每个线程中访问临界资源(临界资源是一次仅允许一个线程使用的共享资源)的那段代码称为临界区。每次只准许一个线程进入临界区，进入后不允许其他线程进入，即：临界区是被synchronized锁定的代码区域，如果临界区执行时间长、语句多，叫做锁的粒度比较粗

# CPU缓存

为什么需要CPU  cache？因为CPU的频率太快了，快到主存跟不上，这样在处理器时钟周期内，CPU常常需要等待主存，浪费资源。所以cache的出现，是为了缓解CPU和内存之间速度的不匹配问题（结构：cpu -> cache -> memory）。CPU cache有什么意义？cache的容量远远小于主存，因此出现cache miss在所难免，既然cache不能包含CPU所需要的所有数据，那么cache的存在真的有意义吗？当然是有意义的——局部性原理。
A. 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；
B. 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

## 硬件内存架构

计算机在执行程序的时候，每条指令都是在CPU中执行的，而执行的时候，又免不了要和数据打交道。而计算机上面的数据，是存放在主存当中的，也就是计算机的物理内存。以多核CPU为例，每个CPU核都包含一组「CPU寄存器」，这些寄存器本质上是在CPU内存中。CPU在这些寄存器上执行操作的速度要比在主内存(RAM)中执行的速度快得多。因为CPU速率高，内存速率慢，为了让存储体系可以跟上CPU的速度，所以中间又加上Cache层，就是我们说的「CPU高速缓存」。

## CPU多级缓存

由于CPU的运算速度远远超越了1级缓存的数据I\O能力，CPU厂商又引入了多级的缓存结构。通常L1、L2是每个CPU核有一个，L3是多个核共用一个。

1. 各种寄存器，用来存储本地变量和函数参数，访问一次需要1cycle，耗时小于1ns；
2. L1 Cache，一级缓存，本地core的缓存，分成32K的数据缓存L1d和32k指令缓存L1i，访问L1需要3cycles，耗时大约1ns；
3. L2 Cache，二级缓存，本地core的缓存，被设计为L1缓存与共享的L3缓存之间的缓冲，大小为256K，访问L2需要12cycles，耗时大约3ns；

4. L3 Cache，三级缓存，在同插槽的所有core共享L3缓存，分为多个2M的段，访问L3需要38cycles，耗时大约12ns；

大致可以得出结论，缓存层级越接近于CPUcore，容量越小，速度越快，同时其造价也更贵。所以为了支撑更多的热点数据，同时追求最高的性价比，多级缓存架构应运而生。

```
# Linux查看CPU的缓存的指令
[root@vbox ~]# ll  /sys/devices/system/cpu/cpu0/cache/
总用量 0
drwxr-xr-x 2 root root 0 10月  2 12:18 index0
drwxr-xr-x 2 root root 0 10月  2 12:18 index1
drwxr-xr-x 2 root root 0 10月  2 12:18 index2
drwxr-xr-x 2 root root 0 10月  2 12:18 index3
# L1数据缓存
[root@vbox ~]# cat  /sys/devices/system/cpu/cpu0/cache/index0/size
32K
# L1指令缓存
[root@vbox ~]# cat  /sys/devices/system/cpu/cpu0/cache/index1/size
32K
# L2
[root@vbox ~]# cat  /sys/devices/system/cpu/cpu0/cache/index2/size
256K
# L3
[root@vbox ~]# cat  /sys/devices/system/cpu/cpu0/cache/index3/size
4096K
```

## CacheLine

Cache又是由很多个「缓存行」(CacheLine)组成的。CacheLine是Cache和RAM交换数据的最小单位。Cache存储数据是固定大小为单位的，称为一个CacheEntry，这个单位称为CacheLine或CacheBlock。给定Cache容量大小和cache-line-size的情况下，它能存储的条目个数(number-of-cache-entries)就是固定的。因为Cache是固定大小的，所以它从主内存获取数据也是固定大小。对于X86来讲，是64Bytes。对于ARM来讲，较旧的架构的CacheLine是32Bytes，但一次内存访存只访问一半的数据也不太合适，所以它经常是一次填两个CacheLine，叫做DoubleFill。

例如：遍历一个长度为16的long数组data[16]，原始数据自然存在于主内存中，访问过程描述如下

1. 访问data[0]，CPUcore尝试访问CPUCache，未命中。
2. 尝试访问主内存，操作系统一次访问的单位是一个CacheLine的大小—64字节，这意味着：既从主内存中获取到了data[0]的值，同时将data[0]~data[7]加入到了CPUCache之中。
3. 访问data[1]~data[7]，CPUcore尝试访问CPUCache，命中直接返回。
4. 访问data[8]，CPUcore尝试访问CPUCache，未命中。
5. 尝试访问主内存。重复步骤2

实验验证：

```
public class MainClass {

    static long[][] arr;

    /*
```

```
     * 使用了横向遍历和纵向遍历两种顺序对这个二位数组进行遍历，遍历总次数相同，只不过循环的方向不同，观察他们
的耗时。
     * 多次运行后可以发现：横向遍历的耗时大约为15ms，纵向遍历的耗时大约为40ms，前者比后者快了1倍有余。
     * 上述现象出现的原因就是CPU-Cache&Cache-Line。因为横向遍历arr= new long[1024 * 1024][8]
     * 每次内层寻访中的数据都存储在连续内存地址中，所以一次访问会将剩下7个数据一起加入CPUCache中，
     * 而纵向访问内存循环的数据不是在连续地址中所以每一次都要从内存中取。
     */
    public static void main(String[] args) throws InterruptedException {
        arr = new long[1024 * 1024][8];
        long sum = 0;
        // 横向遍历
        long marked = System.currentTimeMillis();
        for (int i = 0; i < 1024 * 1024; i += 1) {
            for (int j = 0; j < 8; j++) {
                sum += arr[i][j];
            }
        }
        System.out.println("Loop times:" + (System.currentTimeMillis() - marked) + "ms");
        sum = 0;
        marked = System.currentTimeMillis();
        // 纵向遍历
        for (int i = 0; i < 8; i += 1) {
            for (int j = 0; j < 1024 * 1024; j++) {
                sum += arr[j][i];
            }
        }
        System.out.println("Loop times:" + (System.currentTimeMillis() - marked) + "ms");
    }
}
```

## 伪共享

伪共享指的是多个线程同时读写同一个缓存行的不同变量时导致的CPU缓存失效。尽管这些变量之间没有任何关系，但由于在主内存中邻近，存在于同一个缓存行之中，它们的相互覆盖会导致频繁的缓存未命中，需要重新刷缓存，引发性能下降。伪共享问题的解决方法便是字节填充，只需要保证不同线程的变量存在于不同的CacheLine即可，使用多余的字节来填充可以做点这一点，这样就不会出现伪共享问题。

实验验证：

```
package org.duo.cacheline;

import java.util.concurrent.CountDownLatch;

public class CacheLinePadding {
    public static long count = 10_0000_0000L;

    /**
     * 1) 不加volatile情况，使不使用缓存对齐，由于StoreBuffer和Invalidate Queues的存在对性能影响不大
     * 2) 加volatile
     * 使用缓存对齐:由于两个对象处于两个不同的缓存行内，所以使用缓存锁，并且每个线程锁定不同的缓存行不存在锁
竞争所以性能高
     * 不使用缓存对齐:由于两个对象处于同一缓存行内，所以使用总线锁，并且存在锁竞争相当于串行所以性能低
     * <p>
     * 加不加volatile缓存一致性协议都存在。只不过加了volatile之后由于缓存行被锁定，所以另一个线程要等待锁的释放，并且通过缓存一致性协议读取最新的值
     */
    private static class T {
        private long p1, p2, p3, p4, p5, p6, p7;
        public volatile long x = 0L;
        //public long x = 0L;
        private long p9, p10, p11, p12, p13, p14, p15;
    }

    public static T[] arr = new T[2];

    static {
        arr[0] = new T();
        arr[1] = new T();
    }

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(2);
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (long i = 0; i < count; i++) {
                    arr[0].x = i;
                }
                latch.countDown();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (long i = 0; i < count; i++) {
                    arr[1].x = i;
                }
                latch.countDown();
            }
        });
        final long start = System.nanoTime();
        t1.start();
        t2.start();
        latch.await();
        System.out.println((System.nanoTime() - start) / 100_0000);

    }
}
```

## 缓存带来的问题

缓存的加入是为了解决CPU运算能力和内存读写能力的不匹配问题，简单来说就是为了提升资源利用率。那么在多CPU（一个CPU对应一个或者多个核心）或者多核心下，每个核心都会有一个一级缓存或者二级缓存，也就是说一二级缓存是核心独占的（类似JMM模型，线程的工作内存是线程独占的，主内存是共享的）而三级缓存和主内存是共享的，这样就将导致CPU缓存一致性问题。多个cache与内存之间的数据同步该怎么做？主要有两类：总线锁和缓存一致性协议

## 总线锁

在早期的CPU当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为 CPU 和其他部件进行通信都是通过总线来进行的，如果对总线加锁的话，也就是说阻塞了其他 CPU 对其他部件访问（如内存），从而使得只能有一个 CPU 能使用内存。但是这种方式会有一个问题，由于在锁住总线期间，其他 CPU无法访问内存，导致效率低下。

## 缓存一致性协议(MESI)

缓存一致性协议可以分为两类：“窥探（snooping）”协议和“基于目录的（directory-based）”协议，MESI协议属于一种"窥探协议"。窥探协议的基本思想：所有cache与内存，cache与cache（cache之间也会有数据传输）之间的传输都发生在一条共享的总线上，而所有的cpu都能看到这条总线，同一个指令周期中，只有一个cache可以读写内存，所有的内存访问都要经过仲裁（arbitrate）。窥探协议的思想是，cache不但与内存通信时和总线打交道，而且它会不停地窥探总线上发生的数据交换，跟踪其他cache在做什么。所以当一个cache代表它所属的cpu去读写内存时，其它cpu都会得到通知，它们以此来使自己的cache保持同步。

并非所有情况都会使用缓存一致性协议，如被操作的数据不能被缓存在CPU内部或操作数据跨越多个缓存行(状态无法标识)，则处理器仍然会调用总线锁定。

每个Cache line有两个标志位：dirty和valid标志来标记缓存行的有4个状态，MESI是指4种状态的首字母。协议中最重要的内容有两部分：cache line的状态以及消息通知机制：

### Cache Line的状态

- M(Modified)：描述：该CacheLine有效，数据被修改了，和内存中的数据不一致，数据只存在于本Cache中。CacheLine只有处于Exclusive状态才能被修改。此外，已修改CacheLine如果被丢弃或标记为Invalid，那么先要把它的内容回写到内存中。监听任务：必须时刻监听所有试图读该缓存行相对于主存的操作，这种操作必须在CPU将该缓存行写回主存并将状态变成S（共享）状态之前被延迟执行。
- E(Exclusive)：描述：该CacheLine有效，数据和内存中的数据一致，数据只存在于本Cache中。独占该内存地址，其他处理器的CacheLine不能同时持有它，如果其他处理器原本也持有同一CacheLine，那么它会马上变成"Invalid"状态。监听任务：缓存行也必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成S（共享）状态。
- S(Shared)：描述：该Cache line有效，数据和内存中的数据一致，数据存在于很多Cache中。处于该状态下的cache line只能被cpu读取，不能写入，因为此时还没有独占。监听任务：缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
- I(Invalid)：描述：该Cache line无效。监听任务：无

对于M和E状态而言总是精确的，他们在和该缓存行的真正状态是一致的。而S状态可能是非一致的，如果一个缓存将处于S状态的缓存行作废了，而另一个缓存实际上可能已经独享了该缓存行，但是该缓存却不会将该缓存行升迁为E状态，这是因为其它缓存不会广播他们作废掉该缓存行的通知，同样由于缓存并没有保存该缓存行的copy的数量，因此（即使有这种通知）也没有办法确定自己是否已经独享了该缓存行。

cpu有读取数据的动作，有独占的动作，有独占后更新数据的动作，有更新数据之后回写内存的动作，根据"窥探协议"的规范，每个动作都需要通知到其他cpu，于是有以下的消息机制：

## 消息通知机制

- Read，cpu发起读取数据请求，请求中包含需要读取的数据地址。
- Read Response，作为Read消息的响应，该消息可能是内存响应的，也可能是某cpu响应的(比如该地址在某cpu cache Line中为Modified状态，则该cpu必须返回该地址的最新数据)。
- Invalidate，cpu发起"我要独占一个cache line，其他cpu请失效对应的cache line"的消息，消息中包含了内存地址，所有的其它cpu需要将对应cache line置为Invalid状态。
- Invalidate ACK，收到Invalidate消息的cpu在将对应cache line置为Invalid后，返回Invalid ACK。
- Read Invalidate，相当于Read消息+Invalidate消息，即取得数据并且独占它，将收到一个Read Response和所有其它cpu的Invalidate ACK。
- Write back，写回消息，即将状态为Modified的cache line写回到内存，通常在该行将被替换时使用。现代cpu cache基本都采用"写回(Write Back)"而非"直写(Write Through)"的方式。

结合cache line状态以及消息机制，来看看cpu之间是如何协作的。假设有个四核cpu系统，每个cpu只有一个cache line，每个cache line大小为1个字节，内存地址空间一共两个字节的数据，地址分别为0x0和0x8，有如下操作序列：

| | **cup0** | **cup1** | **cup2** | **cup3** | **main** | **memory** |
|---|---|---|---|---|---|---|
| | cacheline | cacheline | cacheline | cacheline | 0x0 | 0x8 |
| cpu0:read from 0x0 | 0/shared | | | | latest | latest |
| cpu3:read from 0x0 | 0/shared | | | 0/shared | latest | latest |
| cpu0:read from 0x8 | 8/shared | | | 0/shared | latest | latest |
| cpu2:exclusive 0x0 | 8/shared | | 0/exclusive | | latest | latest |
| cpu2:modified 0x0 | 8/shared | | 0/modified | | expire | latest |
| cpu1:atomic inc in 0x0 | 8/shared | 0/modified | | | expire | latest |
| cpu1:read from 0x8 | 8/shared | 8/shared | | | latest | latest |

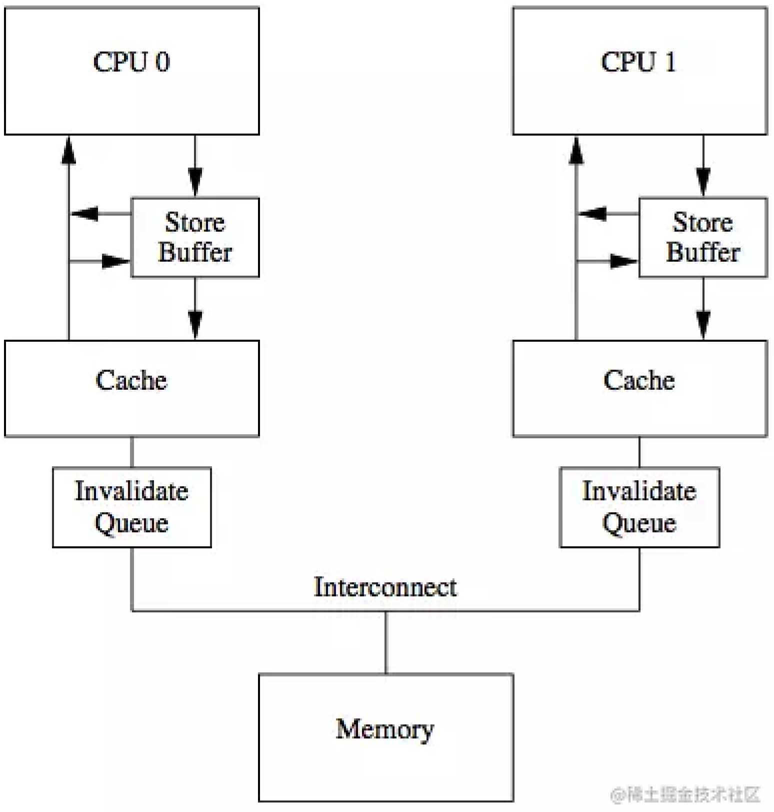
1. 初始状态，4个cpu的cache line都为Invalid状态（黑色表示Invalid）。
2. cpu0发送Read消息，加载0x0的数据，数据从内存返回，cache line状态变为Shared。
3. cpu3发送Read消息，加载0x0的数据，数据从内存返回，cache line状态变为Shared。
4. cpu0发送Read消息，加载0x8的数据，导致cache line被替换，由于之前状态为Shared，即与内存中数据一致，可直接覆盖，而无需回写。
5. cpu2发送Read Invalidate消息，从内存返回最新数据，cpu3返回Invalidate ACK，并将状态变为Invalid，cpu2获得独占权，状态变为Exclusive。
6. cpu2修改cache line中的数据，cache line状态为Modified，同时内存中0x0的数据过期。
7. cpu1 对地址0x0的数据执行原子(atomic)递增操作，发出Read Invalidate消息，cpu2将返回Read Response(而不是内存)，包含最新数据，并返回Invalidate ACK，同时cache line状态变为Invalid。最后cpu1获得独占权，cache line状态变为Modified，数据为递增后的数据，而内存中的数据仍然为过期状态。

8. cpu1 加载0x8的数据，此时cache line将被替换，由于之前状态为Modified，因此需要先执行写回操作，此时内存中0x0的数据得以更新。

这就是缓存一致性协议，一个状态机，仅此而已。因为该协议的存在，每个cpu就可以放心操作属于自己的cache，而不需要担心本地cache中的数据会不会已经被其他cpu修改。但在缓存一致性协议的框架下工作性能不高，但这并不是协议的问题，协议本身逻辑很严谨。性能差在哪里？假如某数据存在于其他cpu的cache中，那自己每次需要修改数据时，都需要发送Read Invalidate消息，除了等待最新数据的返回，还需要等待其他cpu的Invalidate ACK才能继续执行其他指令，这是一种同步行为，于是便有了cpu的优化：处理器指令重排(指令还未结束便去执行其它指令的行为称之为，指令重排)

## 处理器指令重排

指令重排的实现:硬件层面的优化——Store Buffer, Invalid Queue，以及解决Store Buffer, Invalid Queue引入带来的全局顺序性问题——内存屏障指令。



### Store Buffer(存储缓存)

StoreBuffer即存储缓存。位于内核和缓存之间。当处理器需要处理将计算结果写入在缓存中处于shared状态的数据时，需要通知其他内核将该缓存置为 Invalid（无效），引入StoreBuffer后将不再需要处理器去等待其他内核的响应结果，只需要把修改的数据写到StoreBuffer，通知其他内核，然后当前内核即可去执行其它指令。当收到其他内核的响应结果后，再把StoreBuffer中的数据写回缓存，并修改状态为M。（很类似分布式中，数据一致性保障的异步确认）

StoreBuffer带来的问题，例如有如下代码：

```
//初始状态下，假设a，b值都为0，并且a存在cpu1的cache line中(Shared状态)，代码在cpu0中执行，
a = 1;
b = a + 1;
assert(b == 2);
```

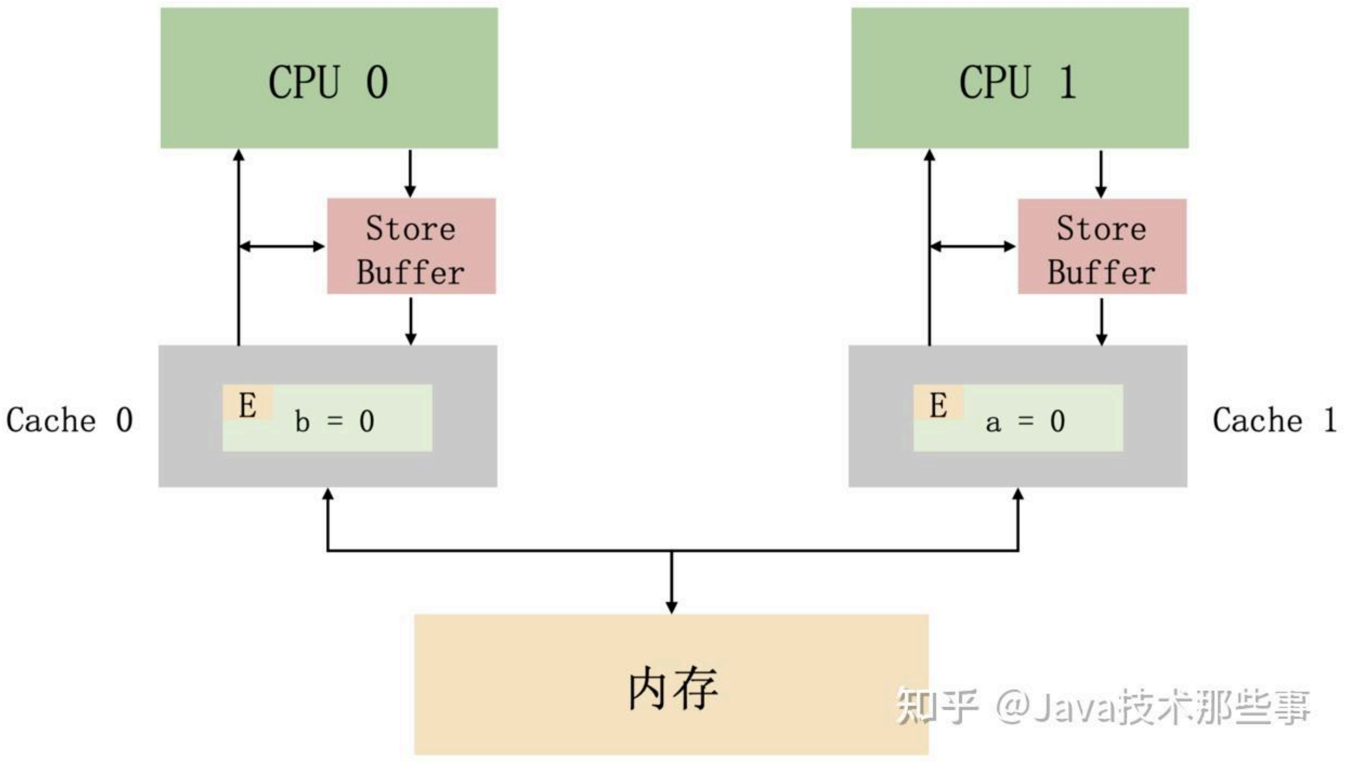
可能出现如下操作序列:

1. cpu0 要写入a，将a=1写入store buffer，并发出Read Invalidate消息，继续其他指令。
2. cpu1 收到Read Invalidate，返回Read Response(包含a=0的cache line)和Invalidate ACK，cpu0 收到Read Response，更新cache line(a=0)。
3. cpu0 开始执行b=a+1，此时cache line中还没有加载b，于是发出Read Invalidate消息，从内存加载b=0，同时cache line中已有a=0，于是得到b=1，状态为Modified状态。
4. cpu0 得到 b=1，断言失败。
5. cpu0 将store buffer中的a=1推送到cache line，然而为时已晚。

造成这个问题的根源在于对同一个cpu存在对a的两份拷贝，一份在cache，一份在store buffer，而cpu计算b=a+1时，a和b的值都来自cache。仿佛代码的执行顺序发生了变化：b = a + 1在a = 1之前执行。

store buffer可能导致破坏程序顺序的问题，硬件工程师在store buffer的基础上，又实现了"store forwarding"技术: cpu可以直接从store buffer中加载数据，即支持将cpu存入store buffer的数据传递(forwarding)给后续的加载操作，而不经由cache，从而提高了CPU的利用率，也能保证了在同一CPU，读写都能顺序执行。注意，这里的读写顺序执行说的是同一CPU，因为，store buffer 的引入并不能保证多CPU全局顺序执行。例如有如下代码：

```
//初始状态下，假设a，b值都为0，a存在于cpu1的cache中，b存在于cpu0的cache中，均为Exclusive状态，cpu0执行foo函数，cpu1执行bar函数
void foo() {
    a = 1;
    b = 1;
}
void bar() {
    while (b == 0) continue;
    assert(a == 1)
}
```



可能出现如下操作序列:

1. cpu1执行while(b == 0)，由于cpu1的Cache中没有b，发出Read b消息
2. cpu0执行a=1，由于cpu0的cache中没有a，因此它将a(当前值1)写入到store buffer并发出Read Invalidate a消息
3. cpu0执行b=1，由于b已经存在在cache中，且为Exclusive状态，因此可直接执行写入
4. cpu0收到Read b消息，将cache中的b(当前值1)返回给cpu1，将b写回到内存，并将cache Line状态改为Shared
5. cpu1收到包含b的cache line，结束while (b == 0)循环
6. cpu1执行assert(a == 1)，由于此时cpu1 cache line中的a仍然为0并且有效(Exclusive)，断言失败
7. cpu1收到Read Invalidate a消息，返回包含a的cache line，并将本地包含a的cache line置为Invalid，然而已经为时已晚。
8. cpu0收到cpu1传过来的cache line，然后将store buffer中的a(当前值1)刷新到cache line

出现这个问题的原因在于cpu不知道a,b之间的数据依赖，cpu0对a的写入需要和其他cpu通信，因此有延迟，而对b的写入直接修改本地cache就行，因此b比a先在cache中生效，导致cpu1读到b=1时，a还存在于store buffer中。从代码的角度来看，仿佛foo函数中b = 1又在a = 1之前执行。

foo函数的代码，即使是store forwarding也阻止不了它被cpu“重排”，虽然这并没有影响foo函数的正确性，但会影响到所有依赖foo函数赋值顺序的线程。到目前为止，可以发现”指令重排“的其中一个本质，cpu为了优化指令的执行效率，引入了store buffer（forwarding），而又因此导致了指令执行顺序的变化。要保证这种顺序一致性，靠硬件是优化不了了，需要在软件层面支持，硬件设计师给开发者提供了内存屏障（memory-barrier）指令，Linux操作系统将写屏障指令封装成了smp_wmb()函数，cpu执行smp_mb()的思路是，会先把当前store buffer中的数据刷到cache之后，再执行屏障后的“写入操作”，该思路有两种实现方式：一是等store buffer生效：等store buffer生效就是内存屏障后续的写必须等待store buffer里面的值都收到了对应的响应消息，都被写到缓存行里面；二是进store buffer排队：进store buffer排队就是内存屏障后续的写直接写到store buffer排队，等 store buffer前面的写全部被写到缓存行。两种方式都需要等，但是等store buffer生效是在CPU等，而进store buffer排队是进store buffer等。所以，进store buffer排队也会相对高效一些，大多数的系统采用的也是这种方式。以第二种实现逻辑为例，看看以下代码执行过程：

```
//初始状态下，假设a，b值都为0，a存在于cpu1的cache中，b存在于cpu0的cache中，均为Exclusive状态，cpu0执行foo函数，cpu1执行bar函数
void foo() {
    a = 1;
    smp_wmb()
    b = 1;
}
void bar() {
    while (b == 0) continue;
    assert(a == 1)
}
```

1. cpu1执行while(b == 0)，由于cpu1的cache中没有b，发出Read b消息。
2. cpu0执行a=1，由于cpu0的cache中没有a，因此它将a(当前值1)写入到store buffer并发出Read Invalidate a消息。
3. cpu0看到smp_wmb()内存屏障，它会标记当前store buffer中的所有条目(即a=1被标记)。
4. cpu0执行b=1，尽管b已经存在在cache中(Exclusive)，但是由于store buffer中还存在被标记的条目，因此b不能直接写入，只能先写入store buffer中。
5. cpu0收到Read b消息，将cache中的b(当前值0)返回给cpu1，并将cache line状态改为Shared。

6. cpu1收到包含b的cache line，继续while (b == 0)循环。
7. cpu1收到Read Invalidate a消息，返回包含a的cache line，并将本地的cache line置为Invalid。
8. cpu0收到cpu1传过来的包含a的cache line，然后将store buffer中的a(当前值1)刷新到cache line，并且将cache line状态置为Modified。
9. 由于cpu0的store buffer中被标记的条目已经全部刷新到cache，此时cpu0可以尝试将store buffer中的b=1刷新到cache，但是由于包含B的cache line已经不是Exclusive而是Shared，因此需要先发Invalidate b消息。
10. cpu1收到Invalidate b消息，将包含b的cache line置为Invalid，返回Invalidate ACK。
11. cpu1继续执行while(b == 0)，此时b已经不在cache中，因此发出Read消息。
12. cpu0收到Invalidate ACK，将store buffer中的b=1写入Cache。
13. cpu0收到Read消息，返回包含b新值的cache line。
14. cpu1收到包含b的cache line，可以继续执行while(b == 0)，终止循环，然后执行assert(a == 1)，此时a不在其cache中，因此发出Read消息。
15. cpu0收到Read消息，返回包含a新值的cache line。
16. cpu1收到包含a的cache line，断言为真。

## Invalidate Queues(失效队列)

简单说处理器修改数据时，需要通知其它内核将该缓存中的数据置为Invalid（失效），于是将该数据放到了Store Bufferes处理。那收到失效指令的这些内核会立即处理这种失效消息吗？答案是不会的，因为就算是一个内核缓存了该数据并不意味着马上要用，这些内核会将失效通知放到Invalidate Queues，然后快速返回Invalidate Acknowledge消息。后续收到失效通知的内核将会从该queues中逐个处理该命令。

加入了invalid queue之后，cpu在处理任何cache line的MSEI状态前，都必须先看invalid queue中是否有该cache line的Invalid消息没有处理。另外，它也再一次破坏了内存的一致性。看看以下代码执行过程：

```
//初始状态下，假设a，b的初始值为0，a在cpu0，cpu1中均为Shared状态，b在cpu0独占(Exclusive状态)，cpu0执行foo，cpu1执行bar
void foo() {
    a = 1;
    smp_wmb()
    b = 1;
}
void bar() {
    while (b == 0) continue;
    assert(a == 1)
}
```

1. cpu0执行a=1，由于其有包含a的cache line，将a写入store buffer，并发出Invalidate a消息。
2. cpu1执行while(b == 0)，它没有b的cache，发出Read b消息。
3. cpu1收到cpu0的Invalidate a消息，将其放入Invalidate Queue，返回Invalidate ACK。
4. cpu0收到Invalidate ACK，将store buffer中的a=1刷新到cache line，标记为Modified。
5. cpu0看到smp_wmb()内存屏障，但是由于其store buffer为空，因此它可以直接跳过该语句。
6. cpu0执行b=1，由于其cache独占b，因此直接执行写入，cache line标记为Modified。
7. cpu0收到cpu1发的Read b消息，将包含b的cache line写回内存并返回该cache line，本地的cache line标记为Shared。

8. cpu1收到包含b(当前值1)的cache line，结束while循环。
9. cpu1执行assert(a == 1)，由于其本地有包含a旧值的cache line，读到a初始值0，断言失败。
10. cpu1这时才处理Invalid Queue中的消息，将包含a旧值的cache line置为Invalid。

问题在于第9步中cpu1在读取a的cache line时，没有先处理Invalid Queue中该cache line的Invalid操作，怎么办？其实cpu还提供了**读屏障指令**，Linux将其封装成smp_rmb()函数，将该函数插入到bar函数中，就像这样：

```
void foo() {
    a = 1;
    smp_wmb()
    b = 1;
}
void bar() {
    while (b == 0) continue;
    smp_rmb()
    assert(a == 1)
}
```

和smp_wmb()类似，cpu执行smp_rmb()的时，会先把当前invalidate queue中的数据处理掉之后，再执行屏障后的“读取操作”；smp_mb(): 同时具有读屏障和写屏障功能。

## 总结

MESI协议，可以保证缓存的一致性，但是无法保证全局顺序性，为了解决此类问题，CPU提供了一种通过指令告知CPU什么指令不能重排，什么指令能重排的机制，就是内存屏蔽。内存屏障有两个作用，处理store buffer和invalidate queue，保持全局顺序性。但很多情况下，只需要处理store buffer和invalidate queue中的其中一个即可，所以很多系统将内存屏障细分成了读屏障（read memory barrier）和写屏障（write memory barrier）。读屏障用于处理invalidate queue，写屏障用于处理store buffer。不同的CPU架构对内存屏障的实现是不尽相同的，以场景的 X86 架构下，不同的内存屏障对应的指令分别是：

- 读屏障：lfence
- 写屏障：sfence
- 读写屏障：mfence

Linux操作系统面向cpu抽象出了自己的一套内存屏障函数，它们分别是：

- smp_rmb(): 在invalid queue的数据被刷完之后再执行屏障后的读操作。
- smp_wmb(): 在store buffer的数据被刷完之后再执行屏障后的写操作。
- smp_mb(): 同时具有读屏障和写屏障功能。

> 1.当一个CPU进行写入时，首先会给其它CPU发送Invalid消息，然后把当前写入的数据写入到Store Buffer中，然后异步在某个时刻真正的写入到Cache中。
> 2.当前CPU核如果要读Cache中的数据，需要先扫描Store Buffer之后再读取Cache。
> 3.但是此时其它CPU核是看不到当前核的Store Buffer中的数据的，要等到Store Buffer中的数据被刷到了Cache之后才会触发失效操作。
> 4.而当一个CPU核收到Invalid消息时，会把消息写入自身的Invalidate Queue中，随后异步将其设为Invalid状态。
> 5.和Store Buffer不同的是，当前CPU核心使用Cache时并不扫描Invalidate Queue部分，所以可能会有极短时间的脏读问题。

## volatile与lock前缀指令

实际通过hsdis工具将代码转换为汇编指令来看volatile的变量在赋值前都加了lock指令，下面详细分析一下lock指令的作用和为什么加上lock指令后就能保证volatile关键字的内存可见性：

lock前缀指令的定义是总线锁，也就是lock前缀指令是通过锁住总线保证可见性和禁止指令重排序的。虽然"总线锁"的说法过于老旧了，现在的系统更多的是"锁缓存行"。但需要注意的是，lock 前缀指令的核心思想还是"锁"，这和内存屏障有着本质的区别，只不过lock前缀指令一部分功能能达到内存屏障的效果。lock前缀指令的作用：

1. 锁总线，其它CPU对内存的读写请求都会被阻塞，直到锁释放，不过实际后来的处理器都采用锁缓存替代锁总线，因为锁总线的开销比较大，锁总线期间其他CPU没法访问内存
2. lock后的写操作会回写已修改的数据，同时让其它CPU相关缓存行失效，从而重新从主存中加载最新的数据
3. 不是内存屏障却能完成类似内存屏障的功能，阻止屏障两边的指令重排序

volatile关键字的实现原理：

工作内存Work Memory其实就是对CPU寄存器和高速缓存的抽象，或者说每个线程的工作内存也可以简单理解为CPU寄存器和高速缓存。那么当写两条线程Thread-A与Threab-B同时操作主存中的一个volatile变量i时，Thread-A写了变量i，那么：

1. Thread-A发出LOCK#指令
2. 发出的LOCK#指令锁总线（或锁缓存行），同时让Thread-B高速缓存中的缓存行内容失效
3. Thread-A向主存回写最新修改的i

Thread-B读取变量i，那么：

1. Thread-B发现对应地址的缓存行被锁了，等待锁的释放，缓存一致性协议会保证它读取到最新的值

# 原子性

# 可见性

# 有序性

## 指令重排序

| 名称 | 说明 |
|---|---|
| 取指 IF | (InstrucTIon Fetch)从内存中取出指令 |
| 译码和取寄存器操作数 ID | (InstrucTIon Decode)把指令送到指令译码器进行译码，产生相应控制信号 |
| 执行或者有效地址计算 EX | (InstrucTIon Execute)指令执行，在执行阶段的最常见部件为算术逻辑部件运算器(ArithmeTIc Logical Unit，ALU) |
| 存储器访问 MEM | 访存（Memory Access）是指存储器访问指令将数据从存储器中读出，或者写入存储器的过程 |
| 写回 WB | 写回（Write-Back）是指将指令执行的结果写回通用寄存器组的过程 |

单周期处理器(时钟周期1000ps)

```
IF ID EX MEM WB
               IF ID EX MEM WB
                              IF ID EX MEM WB
```

流水线处理器(时钟周期200ps)指令不是一步可以执行完毕的，每个步骤涉及的硬件可能不同，指令的某一步骤在执行的时候，只会占用某个类型的硬件，例如：当A指令在取指的时候，B指令是可以进行译码、执行或者写回等操作的。所以可以使用流水线技术来执行指令。可以看到，当第2条指令执行时，第1条指令只是完成了取值操作。假如每个步骤需要1毫秒，那么如果指令2等待指令1执行完再执行，就需要等待5毫秒。而使用流水线后，只需要等待1毫秒。

```
IF ID EX MEM WB
   IF ID EX  MEM WB
      IF ID  EX  MEM WB
```

例如：A = B + C的处理，在ADD指令上的大叉表示一个中断，也就是在这里停顿了一下，因为R2中的数据还没准备好。由于ADD的延迟，后面的指令都要慢一个节拍。

```
LW  R1,B     IF ID EX MEM WB                         //LW表示load，把B的值加载到R1寄存器中
LW  R2,C        IF ID EX  MEM WB                     //LW表示load，把C的值加载到R2寄存器中
ADD R3,R1,R2       IF ID  X   EX MEM WB              //ADD是加法，把R1、R2的值相加，并存放到R3中
SW  A,R3              IF  X   ID EX  MEM WB          //SW表示store存储，将R3寄存器的值保存到变量A中
```

再看复杂一点的情况

```
A = B + C
D = E + F
LW  R1,B     IF ID EX MEM WB
LW  R2,C        IF ID EX  MEM WB
ADD R3,R1,R2       IF ID  X   EX MEM WB
SW  A,R3              IF  X   ID EX  MEM WB
LW  R4,E                  X   IF ID  EX  MEM WB
LW  R5,F                         IF  ID  EX  MEM WB
ADD R6,R4,R5                         IF  ID  X   EX  MEM WB
SW  D,R6                                 IF  X   ID  EX  MEM WB
```

可见上图中有不少停顿。为了减少停顿，我们只需要将LW R4,E和LW R5,F移动到前面执行。

```
LW  R1,B     IF ID EX MEM WB
LW  R2,C        IF ID EX  MEM WB
LW  R4,E           IF ID  EX  MEM WB
ADD R3,R1,R2          IF  ID  EX  MEM WB
LW  R5,F                  IF  ID  EX  MEM WB
SW  A,R3                      IF  ID  EX  MEM WB
ADD R6,R4,R5                      IF  ID  EX  MEM WB
SW  D,R6                              IF  ID  EX  MEM WB
```

可见指令重排序对提高CPU性能十分必要，但是要遵循happens-before规则

1. 程序顺序原则：一个线程内保证语义的串行性(比如a=1;b=a+1;)

2. volatile规则：volatile变量的写，先发生于读，这保证了volatile变量的可见性
3. 锁规则：解锁（unlock）必然发生在随后的加锁（lock）前
4. 传递性：A先于B，B先于C，那么A必然先于C
5. 线程的start()方法先于它的每一个动作
6. 线程的所有操作先于线程的终结（Thread.join()）
7. 线程的中断（interrupt()）先于被中断线程的代码
8. 对象的构造函数执行结束先于finalize()方法

指令重排实例：

```
package org.duo.ordering;

public class Ordering {
    private static int x = 0, y = 0;
    private static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        int i = 0;
        for (; ; ) {
            i++;
            x = 0;
            y = 0;
            a = 0;
            b = 0;
            Thread one = new Thread(new Runnable() {
                public void run() {
                    //由于线程one先启动，下面这句话让它等一等线程two. 可根据自己电脑的实际性能适当调整等
待时间.
                    //shortWait(100000);
                    a = 1;
                    x = b;
                }
            });

            Thread other = new Thread(new Runnable() {
                public void run() {
                    b = 1;
                    y = a;
                }
            });
            one.start();
            other.start();
            one.join();
            other.join();
            String result = "第" + i + "次 (" + x + "," + y + "）";
            if (x == 0 && y == 0) {
                System.err.println(result);
                break;
            } else {
                //System.out.println(result);
            }
        }
    }
    public static void shortWait(long interval) {
        long start = System.nanoTime();
        long end;
        do {
            end = System.nanoTime();
        } while (start + interval >= end);
    }
}
```

## 线程的状态

Thread类中可以找到这个枚举，它定义了线程的相关状态:

```
    public enum State {
        /**
         * Thread state for a thread which has not yet started.
         * 尚未启动的线程的线程状态。
         */
        NEW,

        /**
         * Thread state for a runnable thread.  A thread in the runnable
         * state is executing in the Java virtual machine but it may
         * be waiting for other resources from the operating system
         * such as processor.
         * 可运行线程的线程状态。 处于可运行状态的线程正在 Java 虚拟机中执行，
         * 但它可能正在等待来自操作系统的其他资源，例如处理器
         */
        RUNNABLE,

        /**
         * Thread state for a thread blocked waiting for a monitor lock.
         * A thread in the blocked state is waiting for a monitor lock
         * to enter a synchronized block/method or
         * reenter a synchronized block/method after calling
         * {@link Object#wait() Object.wait}.
         * 线程阻塞等待监视器锁的线程状态。
         * 处于阻塞状态的线程正在等待监视器锁进入同步块/方法或调用后重新进入同步块/方法
         */
        BLOCKED,

        /**
         * Thread state for a waiting thread.
         * A thread is in the waiting state due to calling one of the
         * following methods:
         * <ul>
         *   <li>{@link Object#wait() Object.wait} with no timeout</li>
         *   <li>{@link #join() Thread.join} with no timeout</li>
         *   <li>{@link LockSupport#park() LockSupport.park}</li>
         * </ul>
         *
         * <p>A thread in the waiting state is waiting for another thread to
         * perform a particular action.
         *
         * For example, a thread that has called <tt>Object.wait()</tt>
         * on an object is waiting for another thread to call
         * <tt>Object.notify()</tt> or <tt>Object.notifyAll()</tt> on
         * that object. A thread that has called <tt>Thread.join()</tt>
         * is waiting for a specified thread to terminate.
         * 等待线程的线程状态。
         * 由于调用以下方法之一，线程处于等待状态
         *  Object#wait()
         *  #join() Thread.join
         *  LockSupport#park()
         */
        WAITING,

        /**
         * Thread state for a waiting thread with a specified waiting time.
         * A thread is in the timed waiting state due to calling one of
         * the following methods with a specified positive waiting time:
         * <ul>
         *   <li>{@link #sleep Thread.sleep}</li>
         *   <li>{@link Object#wait(long) Object.wait} with timeout</li>
         *   <li>{@link #join(long) Thread.join} with timeout</li>
         *   <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>
         *   <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>
         * </ul>
         * 具有指定等待时间的等待线程的线程状态。
         * 由于使用指定的正等待时间调用以下方法之一，线程处于定时等待状态
         *  #sleep Thread.sleep
         *  Object#wait(long) Object.wait
         *  #join(long) Thread.join
         *  LockSupport#parkNanos LockSupport.parkNanos
         *  LockSupport#parkUntil LockSupport.parkUntil
         */
        TIMED_WAITING,

        /**
         * Thread state for a terminated thread.
         * The thread has completed execution.
         * 已终止线程的线程状态
         * 线程已完成执行
         */
        TERMINATED;
    }
```

## sleep

sleep()是Thread类中的方法。在指定时间内让当前正在执行的线程进入阻塞状态，让出cpu给其他线程，但不会释放“锁标志”。当指定的时间到了又会自动恢复运行状态。sleep可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会sleep是静态方法，是谁调的谁去睡觉，就算是在main线程里调用了线程b的sleep方法，实际上还是main去睡觉，想让线程b去睡觉要在b的代码中掉sleep。

## yield

只是使当前线程重新回到可执行状态（yield()将导致线程从运行状态转到可运行状态），所以执行yield()线程有可能在进入到可执行状态后马上又被执行. 只能使同优先级的线程有执行的机会。同样, yield()也不会释放锁资源。sleep和yield的区别在于, sleep可以使优先级低的线程得到执行的机会, 而yield只能使同优先级的线程有执行的机会。

## wait

wait()是object类中的方法，当调用时会释放对象锁，进入等待队列，待调用notify()和notifyAll()唤醒指定线程或者所有线程。使用使线程阻塞在当前对象锁的wait方法、以及唤醒当前对象锁的等待线程使用notify或notifyAll方法，都必须拥有相同的对象锁，否则也会抛出IllegalMonitorStateException异常。

sleep(100L)表示:调用sleep后线程进入计时等待状态(TIMED_WAITING)，让出cpu给其他线程；100毫秒后回到可运行状态(RUNNABLE)，虽然sleep没有释放锁，但是在指定的sleep时间后，也许另外一个线程正在使用CPU，那么这时候操作系统是不会重新分配CPU的，直到那个线程挂起或结束；即使这个时候恰巧轮到操作系统进行CPU分配，那么当前线程也不一定就是总优先级最高的那个，CPU还是可能被其他线程抢占去。即Thread.sleep(2000)，2000ms后线程进入就绪状态,如果很长时间都没有获得CPU的执行权，有可能导致睡了大于2000ms。

wait(100L)表示:调用wait后线程进入计时等待状态(TIMED_WAITING)，100毫秒如果锁没有被其他线程占用，则再次得到锁，然后回到可运行状态(RUNNABLE)，但是如果锁被其他线程占用，则进入阻塞状态(BLOCKED)等待os调用分配资源。

Thread.sleep(0)的作用，就是触发操作系统立刻重新进行一次CPU竞争，重新计算优先级。竞争的结果也许是当前线程仍然获得CPU控制权，也许会换成别的线程获得CPU控制权。这也是我们在大循环里面经常会写一句Thread.sleep(0)，因为这样就给了其他线程比如Paint线程获得CPU控制权的权利，这样界面就不会假死在哪里。

## park

通过LockSupport.park()方法，也可以让线程进入休眠。它的底层也是调用了Unsafe类的park方法。调用park方法进入休眠后，线程状态为WAITING。LockSupport.park()的实现原理是通过二元信号量做的阻塞，要注意的是，这个信号量最多只能加到1。我们也可以理解成获取释放许可证的场景。unpark()方法会释放一个许可证，park()方法则是获取许可证，如果当前没有许可证，则进入休眠状态，直到许可证被释放了才被唤醒。无论执行多少次unpark()方法，也最多只会有一个许可证。park、unpark方法和wait、notify()方法有一些相似的地方。都是休眠，然后唤醒。但是wait、notify方法有一个不好的地方，就是在编程的时候必须能保证wait方法比notify方法先执行。如果notify方法比wait方法晚执行的话，就会导致因wait方法进入休眠的线程接收不到唤醒通知的问题。而park、unpark则不会有这个问题，可以先调用unpark方法释放一个许可证，这样后面线程调用park方法时，发现已经有许可证了，就可以直接获取许可证而不用进入休眠状态了。另外，和wait方法不同，执行park进入休眠后并不会释放持有的锁。park方法不会抛出`InterruptedException`，但是它也会响应中断。当外部线程对阻塞线程调用interrupt方法时，park阻塞的线程也会立刻返回。

# Thread的中断机制

线程的thread.interrupt()方法是中断线程，将会设置该线程的中断状态位，即设置为true，中断的结果线程是死亡、还是等待新的任务或是继续运行至下一步，就取决于这个程序本身。它并不像stop方法那样会中断一个正在运行的线程。

| 方法 | 含义 |
|---|---|
| interrupted | 判断某个线程是否已被发送过中断请求。（该方法调用后会将中断标示位清除，即重新设置为false） |
| isInterrupted | 判断某个线程是否已被发送过中断请求。（线程的中断状态不受该方法的影响） |
| interrupt | 中断线程，将会设置该线程的中断状态位，即设置为true， |

Java的中断是一种协作机制。也就是说调用线程对象的interrupt方法并不一定就中断了正在运行的线程，它只是要求线程自己在合适的时机中断自己。每个线程都有一个boolean的中断状态（这个状态不在Thread的属性上），interrupt方法仅仅只是将该状态置为true。对正常运行的线程调用interrupt()并不能终止他，只是改变了interrupt标示符。一般说来，如果一个方法声明抛出InterruptedException，表示该方法是可中断的,比如wait,sleep,join，也就是说可中断方法会对interrupt调用做出响应（例如sleep响应interrupt的操作包括清除中断状态，抛出InterruptedException）,异常都是由可中断方法自己抛出来的，并不是直接由interrupt方法直接引起的。JVM会不断的轮询监听阻塞在Object.wait,Thread.sleep等方法上的线程的interrupted标志位，发现其设置为true后，会停止阻塞并抛出InterruptedException异常，同时将中断标示位重新设置为false。

1. 没有任何语言方面的需求一个被中断的线程应该终止。中断一个线程只是为了引起该线程的注意，被中断线程可以决定如何应对中断。
2. 对于处于sleep，join等操作的线程，如果被调用interrupt()后，会抛出InterruptedException，然后线程的中断标志位会由true重置为false，因为线程为了处理异常已经重新处于就绪状态。
3. 不可中断的操作，包括进入synchronized段以及Lock.lock()，inputSteam.read()等，调用interrupt()对于这几个问题无效，因为它们都不抛出中断异常。如果拿不到资源，它们会无限期阻塞下去。

使用中断信号量中断非阻塞状态的线程

```java
package org.duo.interrupt;

public class InterruptActiveThread extends Thread {
    public static void main(String args[]) throws Exception {
        InterruptActiveThread thread = new InterruptActiveThread();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        // 发出中断请求
        thread.interrupt();
        Thread.sleep(3000);
        System.out.println("Stopping application...");
    }

    public void run() {
        // 每隔一秒检测是否设置了中断标示
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread is running...");
            long time = System.currentTimeMillis();
            // 使用while循环模拟sleep
            while ((System.currentTimeMillis() - time < 1000)) {
            }
        }
        System.out.println("Thread exiting under request...");
    }
}
```

使用thread.interrupt()中断阻塞状态线程

```
package org.duo.interrupt;

public class InterruptBlockThread extends Thread {


    public static void main(String[] args) throws InterruptedException {

        InterruptBlockThread thread = new InterruptBlockThread();
        System.out.println("Starting thread...");
        thread.start();
        Thread.sleep(3000);
        System.out.println("Asking thread to stop...");
        thread.interrupt();// 等中断信号量设置后再调用
        Thread.sleep(3000);
        System.out.println("Stopping application...");

    }

    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread running...");
            try {
                /*
                 * 如果线程阻塞，将不会去检查中断信号量stop变量，所以thread.interrupt()
                 * 会使阻塞线程从阻塞的地方抛出异常，让阻塞线程从阻塞状态逃离出来，并进行异常块进行相应的处
理
                 */
                Thread.sleep(1000);// 线程阻塞，如果线程收到中断操作信号将抛出异常
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted...");
                /*
                 * 对于处于sleep，join等操作的线程，如果被调用interrupt()后，会抛出
InterruptedException，
                 * 然后线程的中断标志位会由true重置为false，因为线程为了处理异常已经重新处于就绪状态。
                 */
                System.out.println(this.isInterrupted());// false

                // 中不中断由自己决定，如果需要真正中断线程，则需要重新设置中断位，如果
                // 不需要，则不用调用
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Thread exiting under request...");
    }
}
```

# 抽象队列同步器AQS及应用

## AQS的原理

Abstract：因为它并不知道怎么上锁，模板方法设计模式即可，暴露上锁的逻辑

Queue：线程阻塞队列

Synchronizer：同步

CAS + state 完成多线程抢锁逻辑

Queue 完成抢不到锁的线程排队

子类只需要实现自己的获取锁逻辑和释放锁逻辑即可，至于排队、阻塞等待、唤醒机制均由AQS来完成。

## AQS的特征

- 阻塞等待队列
- 共享/独占
- 公平/非公平
- 可重入
- 允许中断

除了Lock外，Java.concurrent.util当中同步器的实现如Latch,Barrier,BlockingQueue等， 都是基于AQS框架实现

- 一般通过定义内部类Sync继承AQS
- 将同步器所有调用都映射到Sync对应的方法

AQS定义两种资源共享方式

- Exclusive-独占，只有一个线程能执行，如ReentrantLock
- Share-共享，多个线程可以同时执行，如Semaphore/CountDownLatch

AQS定义两种队列

- 同步等待队列：AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先入先出线程等待队列，Java中的CLH 队列是原CLH队列的一个变种,线程由原自旋机制改为阻塞机制。
- 条件等待队列：Condition是一个多线程间协调通信的工具类，使得某个，或者某些线程一起等待某个条件（Condition）,只有当该条件具备时，这些等待线程才会被唤醒，从而重新争夺锁。

## AbstractOwnableSynchronizer

```
package java.util.concurrent.locks;
```

```
public abstract class AbstractOwnableSynchronizer
    implements java.io.Serializable {

    /** Use serial ID even though all fields transient. */
    private static final long serialVersionUID = 3737899427754241961L;

    /**
     * Empty constructor for use by subclasses.
     */
    protected AbstractOwnableSynchronizer() { }

    /**
     * 当前拥有锁的线程.
     */
    private transient Thread exclusiveOwnerThread;

    /**
     * Sets the thread that currently owns exclusive access.
     * A {@code null} argument indicates that no thread owns access.
     * This method does not otherwise impose any synchronization or
     * {@code volatile} field accesses.
     * @param thread the owner thread
     */
    protected final void setExclusiveOwnerThread(Thread thread) {
        exclusiveOwnerThread = thread;
    }

    /**
     * Returns the thread last set by {@code setExclusiveOwnerThread},
     * or {@code null} if never set.  This method does not otherwise
     * impose any synchronization or {@code volatile} field accesses.
     * @return the owner thread
     */
    protected final Thread getExclusiveOwnerThread() {
        return exclusiveOwnerThread;
    }
}
```

## AbstractQueuedSynchronizer

```
package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import sun.misc.Unsafe;

public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {

    private static final long serialVersionUID = 7373984972572414691L;

    /**
     * Creates a new {@code AbstractQueuedSynchronizer} instance
     * with initial synchronization state of zero.
     */
    protected AbstractQueuedSynchronizer() { }

    // AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先入先出线程等待队列，Java中的CLH 队列是原CLH队列的一个变种,线程由原自旋机制改为阻塞机制
    static final class Node {
        /** 标记节点未共享模式 */
        static final Node SHARED = new Node();
        /** 标记节点为独占模式 */
        static final Node EXCLUSIVE = null;

        /** 在同步队列中等待的线程出现异常、等待超时或者被中断等，需要从同步队列中取消等待(即废弃掉) */
        static final int CANCELLED =  1;
        /** 后继节点的线程处于等待状态，而当前的节点如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行。 */
        static final int SIGNAL    = -1;
        /** 节点在等待队列中，节点的线程等待在Condition上，当其他线程对Condit ion调用了signal()方法后，该节点会从等待队列中转移到同步队列中，加入到同步状态的获取中 */
        static final int CONDITION = -2;
        /**
         * 表示下一次共享式同步状态获取将会被无条件地传播下去
         */
        static final int PROPAGATE = -3;

        /**
         * 标记当前节点的信号量状态 (1,0,-1,-2,-3)5种状态(0表示初始状态)
         * 使用CAS更改状态，volatile保证线程可见性
         */
        volatile int waitStatus;

        /**
         * 前驱节点，当前节点加入到同步队列中被设置
         */
        volatile Node prev;

        /**
         * 后继节点
         */
        volatile Node next;

        /**
         * 节点同步状态的线程
         */
        volatile Thread thread;
        /**
         * 等待队列中的后继节点，如果当前节点是共享的，那么这个字段是一个SHARED常量，
         * 也就是说节点类型(独占和共享)和等待队列中的后继节点共用同一个字段。
         */
        Node nextWaiter;

        /**
         * Returns true if node is waiting in shared mode.
         */
        final boolean isShared() {
            return nextWaiter == SHARED;
        }

        /**
         * Returns previous node, or throws NullPointerException if null.
         * Use when predecessor cannot be null.  The null check could
         * be elided, but is present to help the VM.
         *
         * @return the predecessor of this node
         */
        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        Node() {    // Used to establish initial head or SHARED marker
        }

        Node(Thread thread, Node mode) {     // Used by addWaiter
            this.nextWaiter = mode;
            this.thread = thread;
        }

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

    /**
     * Head of the wait queue, lazily initialized.  Except for
     * initialization, it is modified only via method setHead.  Note:
     * If head exists, its waitStatus is guaranteed not to be
     * CANCELLED.
     */
    private transient volatile Node head;

    /**
     * Tail of the wait queue, lazily initialized.  Modified only via
     * method enq to add new wait node.
     */
    private transient volatile Node tail;

    /**
     * 同步状态(state的初始值=0,表示无锁状态可以进行加锁).
     */
    private volatile int state;

    /**
     * Returns the current value of synchronization state.
     * This operation has memory semantics of a {@code volatile} read.
     * @return current state value
     */
    protected final int getState() {
        return state;
    }

    /**
     * Sets the value of synchronization state.
     * This operation has memory semantics of a {@code volatile} write.
     * @param newState the new state value
     */
    protected final void setState(int newState) {
        state = newState;
    }

    /**
     * Atomically sets synchronization state to the given updated
     * value if the current state value equals the expected value.
     * This operation has memory semantics of a {@code volatile} read
     * and write.
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful. False return indicates that the actual
     *         value was not equal to the expected value.
     */
    protected final boolean compareAndSetState(int expect, int update) {
        // See below for intrinsics setup to support this
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
    }

    // Queuing utilities

    /**
     * The number of nanoseconds for which it is faster to spin
     * rather than to use timed park. A rough estimate suffices
     * to improve responsiveness with very short timeouts.
     */
    static final long spinForTimeoutThreshold = 1000L;

    /**
     * 节点加入CLH同步队列
     * 第一次通过addWaiter调用enq往队列添加节点的时候AbstractQueuedSynchronizer的tail肯定为空
     * 所以下面的代码中第一次循环会先初始化一个空节点并且通过CAS操作将它设置为head，并且将tail也指向此节点
     * 此时由于没有return会继续第二次循环，
     * 在第二次循环中由于已经对head和tail做过初始化操作所以此时tail已经不为空了，那么会走else的逻辑
     * 将node.prev->t并通过CAS操作将node设置为tail，然后将t.next->node(t为第一次初始化创建的空节点)
     * 在上面的操作都完成后会return结束自旋操作
     */
    private Node enq(final Node node) {
        // 自旋操作，直到node添加成功为止
        for (;;) {
            Node t = tail;
            if (t == null) { // 第一次入队，没有dummy node(空的头结点)的存在，需先创建它
                // 队列为空需要初始化，创建空的头节点
                // hasQueuedPredecessors在先后读取完tail和head后，如果这二者只有一个为null（另一个不为null），那么只可能出现“head不为null，tail为null”的情况：
                // 从if (compareAndSetHead(new Node()))到tail = head;的间隙可知，除非另一个线程恰好在当前线程第一次入队的tail = head操作前;之前读取了tali，那么才可能发生 此时head不为null，tail为null的情况。
                //  否则，其他情况下。head和tail要都为null，要么都不为null。
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else { // 至少有一个node，尝试入队
                node.prev = t;
                // 当前节点置为尾部
                if (compareAndSetTail(t, node)) {
                    // 前驱节点的next指针指向当前节点
                    t.next = node;
                    return t;
                }
            }
        }
    }

    /**
     * Creates and enqueues node for current thread and given mode.
     *
     * @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
     * @return the new node
     */
    private Node addWaiter(Node mode) {
        // 将当前线程构建成Node类型
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        // 第一次调用addWaiter方法的时候由于没有对AbstractQueuedSynchronizer的tail赋值所以tail肯定为空，下面的条件不成立
        if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        // 自旋
        enq(node);
        return node;
    }

    /**
     * Sets head of queue to be node, thus dequeuing. Called only by
     * acquire methods.  Also nulls out unused fields for sake of GC
     * and to suppress unnecessary signals and traversals.
     *
     * @param node the node
     */
    private void setHead(Node node) {
        head = node;
        node.thread = null;
        node.prev = null;
    }

    /**
     * Wakes up node's successor, if one exists.
     *
     * @param node the node
     */
    private void unparkSuccessor(Node node) {

        // 获取wait状态
        int ws = node.waitStatus;
        if (ws < 0)
            // 将等待状态waitStatus设置为初始值0
            compareAndSetWaitStatus(node, ws, 0);

        /*
         * 若后继结点为空，或状态为CANCEL（已失效），则从后尾部往前遍历找到最前的一个处于正常阻塞状态的结点
         * 进行唤醒
         */
        Node s = node.next;
        if (s == null || s.waitStatus > 0) {
            s = null;
            for (Node t = tail; t != null && t != node; t = t.prev)
                if (t.waitStatus <= 0)
                    s = t;
        }
        if (s != null)
            LockSupport.unpark(s.thread);
    }

    /**
     * 不依靠参数，直接在调用中获取head，并在一定情况unparkSuccessor这个head的后继线程，被唤醒的线程可能
因为获取不到共享锁而再次阻塞
     */
    private void doReleaseShared() {
        for (;;) {
            Node h = head;
            if (h != null && h != tail) {
                int ws = h.waitStatus;
                if (ws == Node.SIGNAL) {
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                        continue;            // loop to recheck cases
                    // 唤醒head的后继节点，若后继结点为空，或状态为CANCEL（已失效），则从后尾部往前遍历找到最前的一个处于正常阻塞状态的结点进行唤醒
                    unparkSuccessor(h);
                }
                else if (ws == 0 &&
                         !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                    continue;                // loop on failed CAS
            }
            if (h == head)                   // loop if head changed
                break;
        }
    }

    /**
     * AQS独占锁和共享锁最主要的区别就是在于这个方法了，对于独占锁来说，当前线程获取到锁之后，将自己设置为头结点就可以了，不需要再做其他的事情了，但是对于共享锁来说，因为锁是共享的，所以当前线程获取到锁了，要尝试唤醒另外一个以共享模式在队列中等待的线程去获取共享锁，另外一个线程被唤醒后，同样要做相同的事情将唤醒一直传递下去。
     * 入参node所代表的线程一定是当前执行的线程，propagate则代表tryAcquireShared的返回值，由于有if (r >= 0)的保证，propagate必定为>=0，这里返回值的意思是：如果>0，说明我这次获取共享锁成功后，还有剩余共享锁可以获取；如果=0，说明我这次获取共享锁成功后，没有剩余共享锁可以获取。
     *
     */
    private void setHeadAndPropagate(Node node, int propagate) {
        // 为下面的检查记录旧的头结点
        Node h = head;
        // 将当前节点设置为头结点，执行完后，h保存了旧的head，但现在head已经变成node了
        setHead(node);
        // h == null和(h = head) == null和s == null是为了防止空指针异常发生的标准写法，但这不代表就一定会发现它们为空的情况。这里的话，h == null和(h = head) == null是不可能成立，因为只要执行过addWaiter，CHL队列至少也会有一个node存在的；但s == null是可能发生的，比如node已经是队列的最后一个节点。
        // 如果propagate > 0成立的话，说明还有剩余共享锁可以获取，那么短路后面条件
        // 如果propagate = 0成立的话，说明没有剩余共享锁可以获取了，按理说不需要唤醒后继的。也就是说，很多情况下，调用doReleaseShared，会造成acquire thread不必要的唤醒。之所以说不必要，是因为唤醒后因为没有共享锁可以获取而再次阻塞了
        // 继续看，如果propagate > 0不成立，而h.waitStatus < 0成立。这说明旧head的status<0。根据doReleaseShared的逻辑，会发现在unparkSuccessor之前就会CAS设置head的status为0的，因为head的status为0代表一种中间状态（中间状态即：head的后继节点的线程刚刚被唤醒，还没有执行到setHeadAndPropagate中的setHead方法改变head），或者代表head是tail。而这里旧head的status<0，只能是由于doReleaseShared里的compareAndSetWaitStatus(h, 0, Node.PROPAGATE)的操作。出现这种情况，一定是因为有另一个线程在前一个线程处于中间状态的时候也结束运行然后调用doReleaseShared才能造成。也就是说前一个线程结束运行刚刚唤醒后继节点的线程还处于中间状态时，又有线程结束了运行释放出了共享锁。所以propagate == 0只能代表当时tryAcquireShared后没有共享锁剩余，但之后的时刻很可能又有共享锁释放出来了。
        // 比如一个线程工作结束后执行doReleaseShared的compareAndSetWaitStatus(h, Node.SIGNAL, 0))和unparkSuccessor(h)，唤醒CLH队列中的第一个线程，此时线程刚好被唤醒，还没开始执行接下来的逻辑(就是还没有走到setHeadAndPropagate中的setHead)。
        // 此时另一个线程也结束运行，执行doReleaseShared准备唤醒等待队列中的线程，此时head的waitStatus=0(由于被唤醒的线程还没来得及改变head)，所以会执行compareAndSetWaitStatus(h, 0, Node.PROPAGATE))，并且不会执行唤醒操作。
        // 接下来被唤醒的线程继续执行setHeadAndPropagate方法中的setHead及接下来的判断，那么此时第一个h.waitStatus < 0成立，所以会继续去唤醒新的head之后的节点。
        // 继续看，如果propagate > 0不成立，且h.waitStatus < 0不成立，而第二个h.waitStatus < 0成立。注意，第二个h.waitStatus < 0里的h是新head（很可能就是入参node）。第一个h.waitStatus < 0不成立很正常，因为它一般为0（考虑别的线程可能不会那么碰巧读到一个中间状态）。第二个h.waitStatus < 0成立也很正常，因为只要新head不是队尾，那么新head的status肯定是SIGNAL。所以这种情况只会造成不必要的唤醒。
        // 综上所叙，大部分情况都会进入if里面执行doReleaseShared(当新的head就是tail的时候，也就是说当setHead后队列中只剩下最后一个节点的时候(head == tail)由于head的status==0所以下面的if就进不去了。)
        if (propagate > 0 || h == null || h.waitStatus < 0 ||
            (h = head) == null || h.waitStatus < 0) {
            Node s = node.next;
            // s == null完全可能成立，当node是队尾时。此时会调用doReleaseShared，但doReleaseShared里会检测队列中是否存在两个node。
            // 当s != null且s.isShared()，也会调用doReleaseShared。
            // 源码注释自己也说了，if判断这么写是可能造成不必要的唤醒的。
            // The conservatism in both of these checks may cause unnecessary wake-ups, but only when there are multiple racing acquires/releases, so most need signals now or soon anyway.
            if (s == null || s.isShared())
                doReleaseShared();
        }
        // 总结
        // setHeadAndPropagate函数用来设置新head，并在一定情况下调用doReleaseShared。
        // 调用doReleaseShared时，可能会造成acquire thread不必要的唤醒。个人认为，作者这么写，是为了防止一些未知的bug，毕竟当一个线程刚获得共享锁后，它的后继很可能也能获取。
        // 可以猜想，doReleaseShared的实现必须是无伤大雅的，因为有时调用它是没有必要的。
        // PROPAGATE状态存在的意义是它的符号和SIGNAL相同，都是负数，所以能用< 0检测到。因为线程刚被唤醒，但还没设置新head前，当前head的status是0，此时另一个线程也结束运行那么就会把0变成PROPAGATE；接着被唤醒线程可以根据旧head的status<0判断出又有共享锁释放出来了，所以它会继续去唤醒等待队列中的线程。
    }

    // Utilities for various versions of acquire

    /**
     * Cancels an ongoing attempt to acquire.
     *
     * @param node the node
     */
    private void cancelAcquire(Node node) {
        // Ignore if node doesn't exist
        if (node == null)
            return;

        node.thread = null;

        // Skip cancelled predecessors
        Node pred = node.prev;
        while (pred.waitStatus > 0)
            node.prev = pred = pred.prev;

        // predNext is the apparent node to unsplice. CASes below will
        // fail if not, in which case, we lost race vs another cancel
        // or signal, so no further action is necessary.
        Node predNext = pred.next;

        // Can use unconditional write instead of CAS here.
        // After this atomic step, other Nodes can skip past us.
        // Before, we are free of interference from other threads.
        node.waitStatus = Node.CANCELLED;

        // If we are the tail, remove ourselves.
        if (node == tail && compareAndSetTail(node, pred)) {
            compareAndSetNext(pred, predNext, null);
        } else {
            // If successor needs signal, try to set pred's next-link
            // so it will get one. Otherwise wake it up to propagate.
            int ws;
            if (pred != head &&
                ((ws = pred.waitStatus) == Node.SIGNAL ||
                 (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&
                pred.thread != null) {
                Node next = node.next;
                if (next != null && next.waitStatus <= 0)
                    compareAndSetNext(pred, predNext, next);
            } else {
                unparkSuccessor(node);
            }

            node.next = node; // help GC
        }
    }

    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            // 若前驱结点的状态是SIGNAL，意味着当前结点可以被安全地park
            return true;
        if (ws > 0) {
            // 前驱节点状态如果被取消状态，将被移除出队列
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            // 当前驱节点waitStatus为0 or PROPAGATE状态时
            // 将其设置为SIGNAL状态，然后当前结点才可以被安全地park
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        // 如果前驱节点的状态不是-1那么就会返回false，在acquireQueued中会继续自旋判断
        // 一直到当前节点的前驱节点的waitStatus为-1的时候shouldParkAfterFailedAcquire函数才会返回
  true，所以除了tail节点其他节点的waitStatus为-1(tail节点的waitStatus为0)
        // 那么acquireQueued的parkAndCheckInterrupt才会被执行，才会将当前线程挂起
        return false;
    }

    /**
```

```
     * Convenience method to interrupt current thread.
     */
    static void selfInterrupt() {
        Thread.currentThread().interrupt();
    }

    /**
     * 使用LockSupport.park挂起当前线程
     */
    private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);
        // 返回当前线程是否被其他线程触发过中断请求，也就是thread.interrupt(); 如果有触发过中断请求，那么这个方法会返回当前的中断标识true，并且对中断标识进行复位标识已经响应过了中断请求。如果返回true，意味着在acquire方法中会执行selfInterrupt()。
        // 这里清除标志位的原因：
        // 1) 用来区分线程是被哪种方式唤醒，如果是被unpark唤醒，那么parkAndCheckInterrupt会返回false，那么在acquireQueued方法中局部变量interrupted就是false，那么被唤醒后返回到acquire方法中就不会执行selfInterrupt方法。
        // 2) LockSupport.park方法让当前线程进入waiting状态，调用LockSupport.unpark和Thread.interrupt方法都可以唤醒被挂起的线程，但是如果被interrupt唤醒的话如果不通过Thread.interrupted方法清除中断标志位，那么就无法利用LockSupport.park使该线程再次进入waiting状态。
        // 在调用acquireQueued方法阻塞线程的过程中，如果线程被调用interrupt方法从parkAndCheckInterrupt方法中被唤醒后，会再次进入acquireQueued方法中去获取锁，如果获取锁失败会再次进入parkAndCheckInterrupt方法中，如果不通过Thread.interrupted方法清除中断标志位，那么LockSupport.park就无法挂起当前线程的。
        return Thread.interrupted();
    }

    /*
     * 已经在队列当中的Thread节点，准备阻塞等待获取锁
     */
    final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                // 找到当前结点的前驱结点
                final Node p = node.predecessor();
                // 如果前驱结点是头结点的节点，才tryAcquire(再获取一次锁)，其他结点是没有机会tryAcquire的。
                if (p == head && tryAcquire(arg)) {
                    // 获取同步状态成功，将AbstractQueuedSynchronizer的head设置为当前节点(相当于出队列)，在setHead方法内执行了head=node;node.thread = null;node.prev = null;所以相当于将当前节点设置为了一个所有属性都为空的头节点
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
                // 第一次循环通过shouldParkAfterFailedAcquire将前置节点的SIGNAL修改成-1
                // 第二次循环才能通过parkAndCheckInterrupt阻塞当前线程
                // 通过shouldParkAfterFailedAcquire将前置节点改为-1的原因
                // 因为持有锁的线程T0在释放锁的时候，得判断head节点的waitStatus是否!=0
```

```java
                // 如果!=0成立，会再把waitStatus改成0，接着唤醒排队的第一个线程T1，
                // T1被唤醒后接着走，去抢锁，可能会失败(在非公平锁环境下)，失败后T1可能再次被阻塞，
                // head节点状态需要再一次经历两轮循环将waitStatus改成-1
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in exclusive interruptible mode.
     * @param arg the acquire argument
     */
    private void doAcquireInterruptibly(int arg)
        throws InterruptedException {
        final Node node = addWaiter(Node.EXCLUSIVE);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    throw new InterruptedException();
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in exclusive timed mode.
     *
     * @param arg the acquire argument
     * @param nanosTimeout max wait time
     * @return {@code true} if acquired
     */
    private boolean doAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (nanosTimeout <= 0L)
            return false;
        final long deadline = System.nanoTime() + nanosTimeout;
        final Node node = addWaiter(Node.EXCLUSIVE);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return true;
                }
                nanosTimeout = deadline - System.nanoTime();
                if (nanosTimeout <= 0L)
                    return false;
                if (shouldParkAfterFailedAcquire(p, node) &&
                    nanosTimeout > spinForTimeoutThreshold)
                    LockSupport.parkNanos(this, nanosTimeout);
                if (Thread.interrupted())
                    throw new InterruptedException();
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * 共享锁获取流程(执行doAcquireShared的当前线程想要获取到共享锁)
     */
    private void doAcquireShared(int arg) {
        // addWaiter将当前线程包装成一个共享模式的node放到队尾上去
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    // 执行到tryAcquireShared后可能有两种情况:
                    // 如果tryAcquireShared的返回值>=0，说明线程获取共享锁成功了，那么调用
setHeadAndPropagate，然后函数即将返回。
                    int r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        if (interrupted)
                            selfInterrupt();
                        failed = false;
                        return;
                    }
                }
                // 如果tryAcquireShared的返回值<0，说明线程获取共享锁失败了，那么调用
shouldParkAfterFailedAcquire
```

```
                // 这个shouldParkAfterFailedAcquire一般来说，得至少执行两遍才能将返回true：第一次shouldParkAfterFailedAcquirenode把前驱设置为SIGNAL状态，第二次检测到SIGNAL才返回true
                // 既然上一条说了，shouldParkAfterFailedAcquirenode一般执行两遍，那么很有可能第二遍的时候，发现自己的前驱突然变成head了并且获取共享锁成功，又或者本来第一遍的前驱就是head但第二遍获取共享锁成功了。不用觉得第一遍的SIGNAL白设置了，因为设置前驱SIGNAL本来就是为了让前驱唤醒自己的，现在自己处于醒着的状态就获得了共享锁，那就接着执行setHeadAndPropagate就好
                // 剩下的就是常见情况了。线程调用两次shouldParkAfterFailedAcquire，和一次parkAndCheckInterrupt后，便阻塞了。之后就只能等待别人unpark自己了，以后如果自己唤醒了，又会走以上这个流程
                // 总之，执行doAcquireShared的线程一定会是局部变量node所代表的那个线程（即这个node的thread成员）
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in shared interruptible mode.
     * @param arg the acquire argument
     */
    private void doAcquireSharedInterruptibly(int arg)
        throws InterruptedException {
        // 为当前线程和给定模式创建并排队节点
        final Node node = addWaiter(Node.SHARED);
        // 获取共享锁是否成功标记
        boolean failed = true;
        try {
            for (;;) {
                // 获取当前节点的前置节点
                final Node p = node.predecessor();
                // 如果前任节点是头节点，则表示当前节点是下一个应该获得锁的节点
                if (p == head) {
                    // 尝试获取共享锁
                    int r = tryAcquireShared(arg);
                    // 根据上面说明的tryAcquireShared方法返回值的含义，这个判断成立代表获取共享锁成功
                    if (r >= 0) {
                        // 设置头结点并传播
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        failed = false;
                        return;
                    }
                }
                //
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    throw new InterruptedException();
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in shared timed mode.
     *
     * @param arg the acquire argument
     * @param nanosTimeout max wait time
     * @return {@code true} if acquired
     */
    private boolean doAcquireSharedNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (nanosTimeout <= 0L)
            return false;
        final long deadline = System.nanoTime() + nanosTimeout;
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    int r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        failed = false;
                        return true;
                    }
                }
                nanosTimeout = deadline - System.nanoTime();
                if (nanosTimeout <= 0L)
                    return false;
                if (shouldParkAfterFailedAcquire(p, node) &&
                    nanosTimeout > spinForTimeoutThreshold)
                    LockSupport.parkNanos(this, nanosTimeout);
                if (Thread.interrupted())
                    throw new InterruptedException();
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    // Main exported methods

    /*
     *这个方法主要由子类实现加锁的逻辑，因为AQS并不知道你怎么用这个state来上锁
     *比如ReentrantLock的FairSync中对tryAcquire实现如下：
     *
     *  protected final boolean tryAcquire(int acquires) {
```

```
     *      final Thread current = Thread.currentThread();
     *      // 获取AbstractQueuedSynchronizer定义的同步状态state
     *      int c = getState();
     *      // 当c==0的时候表示目前没有任何线程持有锁，可以进行加锁
     *      if (c == 0) {
     *          // 由于是公平锁所以先判断阻塞队列中是否有排队的线程
     *          // 如果没有对state通过CAS改成acquires
     *          if (!hasQueuedPredecessors() &&
     *              compareAndSetState(0, acquires)) {
     *              // 如果CAS修改state成功则将AbstractOwnableSynchronizer中定义的
exclusiveOwnerThread设置为当前线程，并且返回加锁成功
     *              setExclusiveOwnerThread(current);
     *              return true;
     *          }
     *      }
     *      // 如果c不等于0但是持有锁的线程就是当前线程
     *      else if (current == getExclusiveOwnerThread()) {
     *          // 将state值增加acquires
     *          int nextc = c + acquires;
     *          if (nextc < 0)
     *              throw new Error("Maximum lock count exceeded");
     *          setState(nextc);
     *          return true;
     *      }
     *      // 如果上述两个条件都不满足，则返回加锁失败
     *      return false;
     *  }
     */
    protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * 这个方法主要由子类实现释放锁的逻辑
     */
    protected boolean tryRelease(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Attempts to acquire in shared mode. This method should query if
     * the state of the object permits it to be acquired in the shared
     * mode, and if so to acquire it.
     *
     * <p>This method is always invoked by the thread performing
     * acquire.  If this method reports failure, the acquire method
     * may queue the thread, if it is not already queued, until it is
     * signalled by a release from some other thread.
     *
     * <p>The default implementation throws {@link
     * UnsupportedOperationException}.
     *
     * @param arg the acquire argument. This value is always the one
     *        passed to an acquire method, or is the value saved on entry
     *        to a condition wait.  The value is otherwise uninterpreted
     *        and can represent anything you like.
     * @return a negative value on failure; zero if acquisition in shared
     *         mode succeeded but no subsequent shared-mode acquire can
     *         succeed; and a positive value if acquisition in shared
     *         mode succeeded and subsequent shared-mode acquires might
     *         also succeed, in which case a subsequent waiting thread
     *         must check availability. (Support for three different
     *         return values enables this method to be used in contexts
     *         where acquires only sometimes act exclusively.)  Upon
     *         success, this object has been acquired.
     * @throws IllegalMonitorStateException if acquiring would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if shared mode is not supported
     */
    protected int tryAcquireShared(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Attempts to set the state to reflect a release in shared mode.
     *
     * <p>This method is always invoked by the thread performing release.
     *
     * <p>The default implementation throws
     * {@link UnsupportedOperationException}.
     *
     * @param arg the release argument. This value is always the one
     *        passed to a release method, or the current state value upon
     *        entry to a condition wait.  The value is otherwise
     *        uninterpreted and can represent anything you like.
     * @return {@code true} if this release of shared mode may permit a
     *         waiting acquire (shared or exclusive) to succeed; and
     *         {@code false} otherwise
     * @throws IllegalMonitorStateException if releasing would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if shared mode is not supported
     */
    protected boolean tryReleaseShared(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Returns {@code true} if synchronization is held exclusively with
     * respect to the current (calling) thread.  This method is invoked
     * upon each call to a non-waiting {@link ConditionObject} method.
     * (Waiting methods instead invoke {@link #release}.)
     *
     * <p>The default implementation throws {@link
     * UnsupportedOperationException}. This method is invoked
     * internally only within {@link ConditionObject} methods, so need
     * not be defined if conditions are not used.
     *
     * @return {@code true} if synchronization is held exclusively;
     *         {@code false} otherwise
     * @throws UnsupportedOperationException if conditions are not supported
     */
    protected boolean isHeldExclusively() {
        throw new UnsupportedOperationException();
    }

    // acquire：尝试获取锁(在子类中实现)
    // addWaiter:加锁失败则进入排队队列
    // acquireQueued:阻塞加入队列中的线程，等待获取锁
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            // 如果当前线程在调用acquireQueued中被中断过(即线程在被挂起的事情被中断过)，那么在
parkAndCheckInterrupt方法中被唤醒后，会调用Thread.interrupted清除中断标志，然后在线程获取锁后由于
acquireQueued的局部变量interrupted被赋值为true，所以会执行下面的selfInterrupt再产生一个中断请求。
            selfInterrupt();
    }

    /**
     * Acquires in exclusive mode, aborting if interrupted.
     * Implemented by first checking interrupt status, then invoking
     * at least once {@link #tryAcquire}, returning on
     * success.  Otherwise the thread is queued, possibly repeatedly
     * blocking and unblocking, invoking {@link #tryAcquire}
     * until success or the thread is interrupted.  This method can be
     * used to implement method {@link Lock#lockInterruptibly}.
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @throws InterruptedException if the current thread is interrupted
     */
    public final void acquireInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (!tryAcquire(arg))
            doAcquireInterruptibly(arg);
    }

    /**
     * Attempts to acquire in exclusive mode, aborting if interrupted,
     * and failing if the given timeout elapses.  Implemented by first
     * checking interrupt status, then invoking at least once {@link
     * #tryAcquire}, returning on success.  Otherwise, the thread is
     * queued, possibly repeatedly blocking and unblocking, invoking
```

```java
     * {@link #tryAcquire} until success or the thread is interrupted
     * or the timeout elapses.  This method can be used to implement
     * method {@link Lock#tryLock(long, TimeUnit)}.
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @param nanosTimeout the maximum number of nanoseconds to wait
     * @return {@code true} if acquired; {@code false} if timed out
     * @throws InterruptedException if the current thread is interrupted
     */
    public final boolean tryAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        return tryAcquire(arg) ||
            doAcquireNanos(arg, nanosTimeout);
    }

    /**
     * Releases in exclusive mode.  Implemented by unblocking one or
     * more threads if {@link #tryRelease} returns true.
     * This method can be used to implement method {@link Lock#unlock}.
     *
     * @param arg the release argument.  This value is conveyed to
     *        {@link #tryRelease} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @return the value returned from {@link #tryRelease}
     */
    public final boolean release(int arg) {
        if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }

    /**
     * Acquires in shared mode, ignoring interrupts.  Implemented by
     * first invoking at least once {@link #tryAcquireShared},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquireShared} until success.
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquireShared} but is otherwise uninterpreted
     *        and can represent anything you like.
     */
    public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0)
            doAcquireShared(arg);
    }

    /**
     * 可以响应中断的共享锁获取方法
     */
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        // 尝试以共享模式获得锁
        // tryAcquireShared方法返回值的含义:
        // 方法返回值小于0时，代表获取共享锁失败
        // 方法返回值等于0时，因为后续线程没有办法获取共享锁，所以后续的线程是不需要唤醒的
        // 返回值大于0，因为后续线程是有可能成功获取共享锁的，所以当前线程获取到共享锁的同时也要尝试唤醒后续
的线程来获取共享锁
        if (tryAcquireShared(arg) < 0)
            // 采用共享中断模式
            doAcquireSharedInterruptibly(arg);
    }

    /**
     * Attempts to acquire in shared mode, aborting if interrupted, and
     * failing if the given timeout elapses.  Implemented by first
     * checking interrupt status, then invoking at least once {@link
     * #tryAcquireShared}, returning on success.  Otherwise, the
     * thread is queued, possibly repeatedly blocking and unblocking,
     * invoking {@link #tryAcquireShared} until success or the thread
     * is interrupted or the timeout elapses.
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquireShared} but is otherwise uninterpreted
     *        and can represent anything you like.
     * @param nanosTimeout the maximum number of nanoseconds to wait
     * @return {@code true} if acquired; {@code false} if timed out
     * @throws InterruptedException if the current thread is interrupted
     */
    public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        return tryAcquireShared(arg) >= 0 ||
            doAcquireSharedNanos(arg, nanosTimeout);
    }

    /**
     * 释放共享锁
     */
    public final boolean releaseShared(int arg) {
        // 只有共享锁完全释放，才能调用下面的doReleaseShared方法唤醒head节点的后继节点
        if (tryReleaseShared(arg)) {
            // 此处的doReleaseShared方法与setHeadAndPropagate方法中锁唤醒的节点有所差别
            // setHeadAndPropagate方法只唤醒head后继的共享锁节点
            // doReleaseShared方法则会唤醒head后继的独占锁或共享锁
            doReleaseShared();
            return true;
        }
        return false;
    }

    // Queue inspection methods

    /**
     * Queries whether any threads are waiting to acquire. Note that
     * because cancellations due to interrupts and timeouts may occur
     * at any time, a {@code true} return does not guarantee that any
     * other thread will ever acquire.
     *
     * <p>In this implementation, this operation returns in
     * constant time.
     *
     * @return {@code true} if there may be other threads waiting to acquire
     */
    public final boolean hasQueuedThreads() {
        return head != tail;
    }

    /**
     * Queries whether any threads have ever contended to acquire this
     * synchronizer; that is if an acquire method has ever blocked.
     *
     * <p>In this implementation, this operation returns in
     * constant time.
     *
     * @return {@code true} if there has ever been contention
     */
    public final boolean hasContended() {
        return head != null;
    }

    /**
     * Returns the first (longest-waiting) thread in the queue, or
     * {@code null} if no threads are currently queued.
     *
     * <p>In this implementation, this operation normally returns in
     * constant time, but may iterate upon contention if other threads are
     * concurrently modifying the queue.
     *
     * @return the first (longest-waiting) thread in the queue, or
     *         {@code null} if no threads are currently queued
     */
    public final Thread getFirstQueuedThread() {
        // handle only fast path, else relay
        return (head == tail) ? null : fullGetFirstQueuedThread();
    }

    /**
```

```java
     * Version of getFirstQueuedThread called when fastpath fails
     */
    private Thread fullGetFirstQueuedThread() {
        /*
         * The first node is normally head.next. Try to get its
         * thread field, ensuring consistent reads: If thread
         * field is nulled out or s.prev is no longer head, then
         * some other thread(s) concurrently performed setHead in
         * between some of our reads. We try this twice before
         * resorting to traversal.
         */
        Node h, s;
        Thread st;
        if (((h = head) != null && (s = h.next) != null &&
             s.prev == head && (st = s.thread) != null) ||
            ((h = head) != null && (s = h.next) != null &&
             s.prev == head && (st = s.thread) != null))
            return st;

        /*
         * Head's next field might not have been set yet, or may have
         * been unset after setHead. So we must check to see if tail
         * is actually first node. If not, we continue on, safely
         * traversing from tail back to head to find first,
         * guaranteeing termination.
         */

        Node t = tail;
        Thread firstThread = null;
        while (t != null && t != head) {
            Thread tt = t.thread;
            if (tt != null)
                firstThread = tt;
            t = t.prev;
        }
        return firstThread;
    }

    /**
     * Returns true if the given thread is currently queued.
     *
     * <p>This implementation traverses the queue to determine
     * presence of the given thread.
     *
     * @param thread the thread
     * @return {@code true} if the given thread is on the queue
     * @throws NullPointerException if the thread is null
     */
    public final boolean isQueued(Thread thread) {
        if (thread == null)
            throw new NullPointerException();
        for (Node p = tail; p != null; p = p.prev)
            if (p.thread == thread)
```

```
                return true;
        return false;
    }

    /**
     * Returns {@code true} if the apparent first queued thread, if one
     * exists, is waiting in exclusive mode.  If this method returns
     * {@code true}, and the current thread is attempting to acquire in
     * shared mode (that is, this method is invoked from {@link
     * #tryAcquireShared}) then it is guaranteed that the current thread
     * is not the first queued thread.  Used only as a heuristic in
     * ReentrantReadWriteLock.
     */
    final boolean apparentlyFirstQueuedIsExclusive() {
        Node h, s;
        return (h = head) != null &&
            (s = h.next)  != null &&
            !s.isShared()         &&
            s.thread != null;
    }

    /**
     * 判断有没有别的线程排队排在当前线程的前面(即判断当前线程是不是CHL队列的第一个线程)
     */
    public final boolean hasQueuedPredecessors() {
        // The correctness of this depends on head being initialized
        // before tail and on head.next being accurate if the current
        // thread is first in queue.
        Node t = tail; // Read fields in reverse initialization order
        Node h = head;
        Node s;
        // 1.分析h != t返回false的情况。此时hasQueuedPredecessors返回false。
        //  1.当h和t都为null，返回false。此时说明队列为空，还从来没有Node入过队。
        //  2.当h和t都指向同一个Node，也返回false。此时说明队列中只有一个dummy node，那说明没有线程在队列中。
        // 2.分析h != t返回true，且(s = h.next) == null返回true，直接短路后面。此时hasQueuedPredecessors返回true。
        //  1.既然h != t返回true，说明h和t不相等，先考虑特殊情况（上面讲到的出现“head不为null，tail为null”的情况），那么说明有一个线程正在执行enq，且它正好执行到if (compareAndSetHead(new Node()))到tail = head;的间隙。但这个线程肯定不是当前线程，所以不用判断后面短路的s.thread != Thread.currentThread()了，因为当前线程连enq都没开始执行，但另一个线程都开始执行enq了，那不就是说明当前线程排在别人后面了。
        // 3.分析h != t返回true，且(s = h.next) == null返回false，且s.thread != Thread.currentThread()返回true。此时hasQueuedPredecessors返回true。如果s.thread != Thread.currentThread()返回false。此时hasQueuedPredecessors返回false。
        //  1.现在知道head不为null，而且head.next也不为null了（(s = h.next) == null返回false）。我们也知道队列中第一个等待的线程存放在head.next里（注意，head为dummy node，不存放线程），那么如果head.next的线程不是当前线程，那即说明当前线程已经排在别人线程后面了。
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
    }


    // Instrumentation and monitoring methods
```

```
/**
 * Returns an estimate of the number of threads waiting to
 * acquire.  The value is only an estimate because the number of
 * threads may change dynamically while this method traverses
 * internal data structures.  This method is designed for use in
 * monitoring system state, not for synchronization
 * control.
 *
 * @return the estimated number of threads waiting to acquire
 */
public final int getQueueLength() {
    int n = 0;
    for (Node p = tail; p != null; p = p.prev) {
        if (p.thread != null)
            ++n;
    }
    return n;
}

/**
 * Returns a collection containing threads that may be waiting to
 * acquire.  Because the actual set of threads may change
 * dynamically while constructing this result, the returned
 * collection is only a best-effort estimate.  The elements of the
 * returned collection are in no particular order.  This method is
 * designed to facilitate construction of subclasses that provide
 * more extensive monitoring facilities.
 *
 * @return the collection of threads
 */
public final Collection<Thread> getQueuedThreads() {
    ArrayList<Thread> list = new ArrayList<Thread>();
    for (Node p = tail; p != null; p = p.prev) {
        Thread t = p.thread;
        if (t != null)
            list.add(t);
    }
    return list;
}

/**
 * Returns a collection containing threads that may be waiting to
 * acquire in exclusive mode. This has the same properties
 * as {@link #getQueuedThreads} except that it only returns
 * those threads waiting due to an exclusive acquire.
 *
 * @return the collection of threads
 */
public final Collection<Thread> getExclusiveQueuedThreads() {
    ArrayList<Thread> list = new ArrayList<Thread>();
    for (Node p = tail; p != null; p = p.prev) {
        if (!p.isShared()) {
```

```java
                Thread t = p.thread;
                if (t != null)
                    list.add(t);
            }
        }
        return list;
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire in shared mode. This has the same properties
     * as {@link #getQueuedThreads} except that it only returns
     * those threads waiting due to a shared acquire.
     *
     * @return the collection of threads
     */
    public final Collection<Thread> getSharedQueuedThreads() {
        ArrayList<Thread> list = new ArrayList<Thread>();
        for (Node p = tail; p != null; p = p.prev) {
            if (p.isShared()) {
                Thread t = p.thread;
                if (t != null)
                    list.add(t);
            }
        }
        return list;
    }

    /**
     * Returns a string identifying this synchronizer, as well as its state.
     * The state, in brackets, includes the String {@code "State ="}
     * followed by the current value of {@link #getState}, and either
     * {@code "nonempty"} or {@code "empty"} depending on whether the
     * queue is empty.
     *
     * @return a string identifying this synchronizer, as well as its state
     */
    public String toString() {
        int s = getState();
        String q  = hasQueuedThreads() ? "non" : "";
        return super.toString() +
            "[State = " + s + ", " + q + "empty queue]";
    }


    // Internal support methods for Conditions

    /**
     * 该方法用于判断节点node是否在同步队列上
     */
    final boolean isOnSyncQueue(Node node) {
        /*
         * 节点在同步队列上时，其状态可能为 0、SIGNAL、PROPAGATE 和 CANCELLED 其中之一，
         * 但不会为 CONDITION，所以可已通过节点的等待状态来判断节点所处的队列。
         * 或者如果prev为null也一定是在条件队列。
         *
         * 同步队列里的节点prev为null只可能是获取到锁后调用setHead清为null,
         * 新入队的节点prev值是不会为null的。
         * 另外,条件队列里节点是用nextWaiter来维护的，不用next和prev。
         */
        if (node.waitStatus == Node.CONDITION || node.prev == null)
            return false;
        /*
         * 如果next不为null,一定是在同步队列的。因为条件队列使用的是nextWaiter指向后继节点的，
         * 条件队列上节点的next指针均为null。但仅以node.next != null条件断定节点在同步队列是不充分的。
         * 节点在入队过程中，是先设置node.prev，后设置node.next。如果设置完node.prev后，线程被切换了，
         * 此时node.next仍然为null，但此时node确实已经在同步队列上了，所以这里还需要进行后续的判断。
         * 这里值得一提的是在AQS的cancelAcquire方法中,
         * 一个节点将自己移除出队列的时候会把自己的next域指向自己。
         * 即node.next = node;
         *
         * 从GC效果上来看node.next = node和node.next = null无异,
         * 但是这对此处next不为null一定在同步队列上来说,
         * 这样可以将节点在同步队列上被取消的情况与普通情况归一化判断。
         */
        if (node.next != null) // If has successor, it must be on queue
            return true;
        /*
         * 有可能node.prev的值不为null,但还没在队列中,因为入队时CAS队列的tail可能失败。
         * 这是从tail向前遍历一次,确定是否已经在同步队列上。
         */
        return findNodeFromTail(node);
    }

    /**
     * 从队列尾部向前遍历判断节点是否在队列中。
     */
    private boolean findNodeFromTail(Node node) {
        Node t = tail;
        for (;;) {
            if (t == node)
                return true;
            if (t == null)
                return false;
            t = t.prev;
        }
    }

    /**
     * 这个方法用于将条件队列中的节点转移到同步队列中
     */
    final boolean transferForSignal(Node node) {
        /*
         * 如果将节点的等待状态由 CONDITION 设为 0 失败，则表明节点被取消。
         * 因为 transferForSignal 中不存在线程竞争的问题，所以下面的 CAS
         * 失败的唯一原因是节点的等待状态为 CANCELLED。
         */
        if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
            return false;

        /*
         * 调用 enq 方法将 node 转移到同步队列中，并返回 node 的前驱节点 p
         */
        Node p = enq(node);
        int ws = p.waitStatus;
        /*
         * 如果前驱节点的等待状态 ws > 0，则表明前驱节点处于取消状态，此时应唤醒 node 对应的
         * 线程去获取同步状态。如果 ws <= 0，这里通过 CAS 将节点 p 的等待设为 SIGNAL。
         * 这样，节点 p 在释放同步状态后，才会唤醒后继节点 node。如果 CAS 设置失败，则应立即
         * 唤醒 node 节点对应的线程。以免因 node 没有被唤醒导致同步队列挂掉。
         */
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
            LockSupport.unpark(node.thread);
        return true;
    }

    /**
     * 判断中断发生的时机，分为两种：
     * 1. 中断在节点被转移到同步队列前发生，此时返回 true
     * 2. 中断在节点被转移到同步队列期间或之后发生，此时返回 false
     */
    final boolean transferAfterCancelledWait(Node node) {
        // 中断在节点被转移到同步队列前发生，此时自行将节点转移到同步队列上，并返回true
        if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
            enq(node);
            return true;
        }
        /*
         * 如果上面的条件分支失败了，则表明已经有线程在调用 signal/signalAll 方法了，这两个
         * 方法会先将节点等待状态由CONDITION设置为0后，再调用enq方法转移节点。下面判断节
         * 点是否已经在同步队列上的原因是，signal/signalAll 方法可能仅设置了等待状态，还没
         * 来得及转移节点就被切换走了。所以这里用自旋的方式判断signal/signalAll 是否已经完
         * 成了转移操作。这种情况表明了中断发生在节点被转移到同步队列期间。
         */
        while (!isOnSyncQueue(node))
            Thread.yield();
        return false;
    }

    /**
     * 这个方法用于完全释放同步状态。这里解释一下完全释放的原因：为了避免死锁的产生，锁的实现上
     * 一般应该支持重入功能。对应的场景就是一个线程在不释放锁的情况下可以多次调用同一把锁的
     * lock 方法进行加锁，且不会加锁失败，如失败必然导致导致死锁。锁的实现类可通过 AQS 中的整型成员
     * 变量 state 记录加锁次数，每次加锁，将 state++。每次 unlock 方法释放锁时，则将 state--，
     * 直至 state = 0，线程完全释放锁。用这种方式即可实现了锁的重入功能。
     */
    final int fullyRelease(Node node) {
        boolean failed = true;
        try {
            // 获取同步状态数值
            int savedState = getState();
            // 调用 release 释放指定数量的同步状态
            if (release(savedState)) {
                failed = false;
                return savedState;
            } else {
                throw new IllegalMonitorStateException();
            }
        } finally {
            // 如果 relase 出现异常或释放同步状态失败，此处将 node 的等待状态设为 CANCELLED
            if (failed)
                node.waitStatus = Node.CANCELLED;
        }
    }

    // Instrumentation methods for conditions

    /**
     * Queries whether the given ConditionObject
     * uses this synchronizer as its lock.
     *
     * @param condition the condition
     * @return {@code true} if owned
     * @throws NullPointerException if the condition is null
     */
    public final boolean owns(ConditionObject condition) {
        return condition.isOwnedBy(this);
    }

    /**
     * Queries whether any threads are waiting on the given condition
     * associated with this synchronizer. Note that because timeouts
     * and interrupts may occur at any time, a {@code true} return
     * does not guarantee that a future {@code signal} will awaken
     * any threads.  This method is designed primarily for use in
     * monitoring of the system state.
     *
     * @param condition the condition
     * @return {@code true} if there are any waiting threads
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final boolean hasWaiters(ConditionObject condition) {
        if (!owns(condition))
            throw new IllegalArgumentException("Not owner");
        return condition.hasWaiters();
    }

    /**
```

```
     * Returns an estimate of the number of threads waiting on the
     * given condition associated with this synchronizer. Note that
     * because timeouts and interrupts may occur at any time, the
     * estimate serves only as an upper bound on the actual number of
     * waiters.  This method is designed for use in monitoring of the
     * system state, not for synchronization control.
     *
     * @param condition the condition
     * @return the estimated number of waiting threads
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final int getWaitQueueLength(ConditionObject condition) {
        if (!owns(condition))
            throw new IllegalArgumentException("Not owner");
        return condition.getWaitQueueLength();
    }

    /**
     * Returns a collection containing those threads that may be
     * waiting on the given condition associated with this
     * synchronizer.  Because the actual set of threads may change
     * dynamically while constructing this result, the returned
     * collection is only a best-effort estimate. The elements of the
     * returned collection are in no particular order.
     *
     * @param condition the condition
     * @return the collection of threads
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final Collection<Thread> getWaitingThreads(ConditionObject condition) {
        if (!owns(condition))
            throw new IllegalArgumentException("Not owner");
        return condition.getWaitingThreads();
    }

    /**
     * condition队列与之前分析过的AQS CLH队列区分开，CLH队列是用来排队获取锁的线程的，condition队列是用来存放在condition条件上等待的线程的，两者有不同的作用也是两个不同的队列。
     * ConditionObject是通过基于单链表的条件队列来管理等待线程的。
     * java.util.concurrent.locks.Condition是JUC提供的与Java的Object中wait/notify/notifyAll类似功能的一个接口，通过此接口，线程可以在某个特定的条件下等待/唤醒。与wait/notify/notifyAll操作需要获得对象监视器类似，一个Condition实例与某个互斥锁绑定，在此Condition实例进行等待/唤醒操作的调用也需要获得互斥锁，线程被唤醒后需要再次获取到锁，否则将继续等待。而与原生的wait/notify/notifyAll等API不同的地方在于，Condition具有更丰富的功能，例如等待可以响应/不响应中断，可以设定超时时间或是等待到某个具体时间点。
     * 此外一把互斥锁可以绑定多个Condition，这意味着在同一把互斥锁上竞争的线程可以在不同的条件下等待，唤醒时可以根据条件来唤醒线程，这是Object中的wait/notify/notifyAll不具备的机制
     * Condition接口的主要实现类是AQS的内部类ConditionObject，它内部维护了一个队列，可以称之为条件队列，在某个Condition上等待的线程被signal/signalAll后，ConditionObject会将对应的节点转移到外部类AQS的等待队列中，线程需要获取到AQS等待队列的锁，才可以继续恢复执行后续的用户代码。
     */
    public class ConditionObject implements Condition, java.io.Serializable {
        private static final long serialVersionUID = 1173984872572414699L;
        /** condition 队列的第一个节点. */
        private transient Node firstWaiter;
        /** condition 队列的最后一个节点. */
        private transient Node lastWaiter;

        /**
         * Creates a new {@code ConditionObject} instance.
         */
        public ConditionObject() { }

        // Internal methods

        /**
         * 将当先线程封装成节点，并将节点添加到条件队列尾部
         */
        private Node addConditionWaiter() {
            Node t = lastWaiter;
            /*
             * 如果条件队列中最后一个waiter节点状态为取消，则调用unlinkCancelledWaiters清理队列。
             * fullyRelease 内部调用 release 发生异常或释放同步状态失败时，节点的等待状态会被设置为CANCELLED。所以这里要清理一下已取消的节点
             */
            if (t != null && t.waitStatus != Node.CONDITION) {
                unlinkCancelledWaiters();
                // 重新读取lastWaiter
                t = lastWaiter;
            }
            // 创建节点，并将节点置于队列尾部
            Node node = new Node(Thread.currentThread(), Node.CONDITION);
            // t如果为null，初始化firstWaiter为当前节点。
            if (t == null)
                firstWaiter = node;
            else
                // 将队尾的next连接到node。
                t.nextWaiter = node;
            lastWaiter = node;
            return node;
        }

        /**
         * Removes and transfers nodes until hit non-cancelled one or
         * null. Split out from signal in part to encourage compilers
         * to inline the case of no waiters.
         * @param first (non-null) the first node on condition queue
         */
        private void doSignal(Node first) {
            do {
                /*
                 * 将firstWaiter指向first节点的nextWaiter节点，while循环将会用到更新后的
                 * firstWaiter作为判断条件。
                 */
                if ( (firstWaiter = first.nextWaiter) == null)
                    lastWaiter = null;
                first.nextWaiter = null;
            /*
             * 调用transferForSignal将节点转移到同步队列中，如果失败，且firstWaiter
             * 不为null，则再次进行尝试。transferForSignal成功了，while循环就结束了。
             */
            } while (!transferForSignal(first) &&
                     (first = firstWaiter) != null);
        }

        /**
         * Removes and transfers all nodes.
         * @param first (non-null) the first node on condition queue
         */
        private void doSignalAll(Node first) {
            lastWaiter = firstWaiter = null;
            do {
                Node next = first.nextWaiter;
                first.nextWaiter = null;
                transferForSignal(first);
                first = next;
            } while (first != null);
        }

        /**
         * 清理等待状态为 CANCELLED 的节点
         */
        private void unlinkCancelledWaiters() {
            Node t = firstWaiter;
            // 指向上一个等待状态为非 CANCELLED 的节点
            Node trail = null;
            while (t != null) {
                Node next = t.nextWaiter;
                if (t.waitStatus != Node.CONDITION) {
                    t.nextWaiter = null;
                    /*
                     * trail 为 null，表明 next 之前的节点等待状态均为 CANCELLED，此时更新
                     * firstWaiter 引用的指向。
                     * trail 不为 null，表明 next 之前有节点的等待状态为 CONDITION，这时将
                     * trail.nextWaiter 指向 next 节点。
                     */
                    if (trail == null)
                        firstWaiter = next;
                    else
                        trail.nextWaiter = next;
                    // next 为 null，表明遍历到条件队列尾部了，此时将 lastWaiter 指向 trail
                    if (next == null)
                        lastWaiter = trail;
                }
                else
                    // t.waitStatus = Node.CONDITION，则将 trail 指向 t
                    trail = t;
                t = next;
            }
        }

        // public methods

        /**
         * 将条件队列中的头结点转移到同步队列中
         */
        public final void signal() {
            // 查线程是否获取了独占锁，未获取独占锁调用 signal 方法是不允许的
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            Node first = firstWaiter;
            if (first != null)
                // 将头结点转移到同步队列中
                doSignal(first);
        }

        /**
         * Moves all threads from the wait queue for this condition to
         * the wait queue for the owning lock.
         *
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        public final void signalAll() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            Node first = firstWaiter;
            if (first != null)
                doSignalAll(first);
        }

        /**
         * Implements uninterruptible condition wait.
         * <ol>
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * </ol>
         */
        public final void awaitUninterruptibly() {
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            boolean interrupted = false;
            while (!isOnSyncQueue(node)) {
                LockSupport.park(this);
                if (Thread.interrupted())
                    interrupted = true;
            }
            if (acquireQueued(node, savedState) || interrupted)
                selfInterrupt();
        }

        /*
         * For interruptible waits, we need to track whether to throw
         * InterruptedException, if interrupted while blocked on
         * condition, versus reinterrupt current thread, if
         * interrupted while blocked waiting to re-acquire.
         */

        /** Mode meaning to reinterrupt on exit from wait */
        private static final int REINTERRUPT =  1;
        /** Mode meaning to throw InterruptedException on exit from wait */
        private static final int THROW_IE    = -1;

        /**
         * 检测线程在等待期间是否发生了中断
         */
        private int checkInterruptWhileWaiting(Node node) {

            return Thread.interrupted() ?
                (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
                0;
        }

        /**
         * 根据中断类型做出相应的处理：
         * THROW_IE：抛出 InterruptedException 异常
         * REINTERRUPT：重新设置中断标志，向后传递中断
         */
        private void reportInterruptAfterWait(int interruptMode)
            throws InterruptedException {
            if (interruptMode == THROW_IE)
                throw new InterruptedException();
            else if (interruptMode == REINTERRUPT)
                selfInterrupt();
        }

        /**
         * Implements interruptible condition wait.
         */
        public final void await() throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
            // 加到条件队列中
```

```
Node node = addConditionWaiter();
// 完全释放互斥锁(无论锁是否可以重入)，如果没有持锁，会抛出异常
int savedState = fullyRelease(node);
int interruptMode = 0;
/*
 * 判断节点是否在同步队列上，如果不在则阻塞线程。
 * 循环结束的条件：
 * 1. 其他线程调用 singal/singalAll，node 将会被转移到同步队列上。node 对应线程将
 *    会在获取同步状态的过程中被唤醒，并走出 while 循环。
 * 2. 线程在阻塞过程中产生中断
 */
while (!isOnSyncQueue(node)) {
    LockSupport.park(this);
    /*
     * 检测中断模式，这里有两种中断模式，如下：
     * THROW_IE：
     *     中断在 node 转移到同步队列“前”发生，需要当前线程自行将 node 转移到同步队
     *     列中，并在随后抛出 InterruptedException 异常。
     *
     * REINTERRUPT：
     *     中断在 node 转移到同步队列“期间”或“之后”发生，此时表明有线程正在调用
     *     singal/singalAll 转移节点。在该种中断模式下，再次设置线程的中断状态。
     *     向后传递中断标志，由后续代码去处理中断。
     */
    if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
        break;
}
/*
 * 被转移到同步队列的节点 node 将在 acquireQueued 方法中重新获取同步状态，注意这里
 * 的这里的 savedState 是上面调用 fullyRelease 所返回的值，与此对应，可以把这里的
 * acquireQueued 作用理解为 fullyAcquire（并不存在这个方法）。
 *
 * 如果上面的 while 循环没有产生中断，则 interruptMode = 0。但 acquireQueued 方法
 * 可能会产生中断，产生中断时返回 true。这里仍将 interruptMode 设为 REINTERRUPT，
 * 目的是继续向后传递中断，acquireQueued 不会处理中断。
 */
if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
    interruptMode = REINTERRUPT;
/*
 * 正常通过 singal/singalAll 转移节点到同步队列时，nextWaiter 引用会被置空。
 * 若发生线程产生中断（THROW_IE）或 fullyRelease 方法出现错误等异常情况，
 * 该引用则不会被置空
 */
if (node.nextWaiter != null) // clean up if cancelled
    // 清理等待状态非 CONDITION 的节点
    unlinkCancelledWaiters();
/*
 * 根据 interruptMode 觉得中断的处理方式：
 *   THROW_IE：抛出 InterruptedException 异常
 *   REINTERRUPT：重新设置线程中断标志
 */
if (interruptMode != 0)
    reportInterruptAfterWait(interruptMode);
```

```java
        }

        /**
         * Implements timed condition wait.
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * </ol>
         */
        public final long awaitNanos(long nanosTimeout)
                throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            final long deadline = System.nanoTime() + nanosTimeout;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (nanosTimeout <= 0L) {
                    transferAfterCancelledWait(node);
                    break;
                }
                if (nanosTimeout >= spinForTimeoutThreshold)
                    LockSupport.parkNanos(this, nanosTimeout);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
                nanosTimeout = deadline - System.nanoTime();
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return deadline - System.nanoTime();
        }

        /**
         * Implements absolute timed condition wait.
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * <li> If timed out while blocked in step 4, return false, else true.
         * </ol>
         */
        public final boolean awaitUntil(Date deadline)
                throws InterruptedException {
            long abstime = deadline.getTime();
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            boolean timedout = false;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (System.currentTimeMillis() > abstime) {
                    timedout = transferAfterCancelledWait(node);
                    break;
                }
                LockSupport.parkUntil(this, abstime);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return !timedout;
        }

        /**
         * Implements timed condition wait.
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * <li> If timed out while blocked in step 4, return false, else true.
         * </ol>
         */
        public final boolean await(long time, TimeUnit unit)
                throws InterruptedException {
            long nanosTimeout = unit.toNanos(time);
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            final long deadline = System.nanoTime() + nanosTimeout;
            boolean timedout = false;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (nanosTimeout <= 0L) {
                    timedout = transferAfterCancelledWait(node);
                    break;
                }
                if (nanosTimeout >= spinForTimeoutThreshold)
                    LockSupport.parkNanos(this, nanosTimeout);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
                nanosTimeout = deadline - System.nanoTime();
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return !timedout;
        }

        //  support for instrumentation

        /**
         * Returns true if this condition was created by the given
         * synchronization object.
         *
         * @return {@code true} if owned
         */
        final boolean isOwnedBy(AbstractQueuedSynchronizer sync) {
            return sync == AbstractQueuedSynchronizer.this;
        }

        /**
         * Queries whether any threads are waiting on this condition.
         * Implements {@link AbstractQueuedSynchronizer#hasWaiters(ConditionObject)}.
         *
         * @return {@code true} if there are any waiting threads
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final boolean hasWaiters() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
                if (w.waitStatus == Node.CONDITION)
                    return true;
            }
            return false;
        }

        /**
```

```
         * Returns an estimate of the number of threads waiting on
         * this condition.
         * Implements {@link
AbstractQueuedSynchronizer#getWaitQueueLength(ConditionObject)}.
         *
         * @return the estimated number of waiting threads
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final int getWaitQueueLength() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            int n = 0;
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
                if (w.waitStatus == Node.CONDITION)
                    ++n;
            }
            return n;
        }

        /**
         * Returns a collection containing those threads that may be
         * waiting on this Condition.
         * Implements {@link AbstractQueuedSynchronizer#getWaitingThreads(ConditionObject)}.
         *
         * @return the collection of threads
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final Collection<Thread> getWaitingThreads() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            ArrayList<Thread> list = new ArrayList<Thread>();
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
                if (w.waitStatus == Node.CONDITION) {
                    Thread t = w.thread;
                    if (t != null)
                        list.add(t);
                }
            }
            return list;
        }
    }

    /**
     * Setup to support compareAndSet. We need to natively implement
     * this here: For the sake of permitting future enhancements, we
     * cannot explicitly subclass AtomicInteger, which would be
     * efficient and useful otherwise. So, as the lesser of evils, we
     * natively implement using hotspot intrinsics API. And while we
     * are at it, we do the same for other CASable fields (which could
     * otherwise be done with atomic field updaters).
     */
```

```java
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long stateOffset;
    private static final long headOffset;
    private static final long tailOffset;
    private static final long waitStatusOffset;
    private static final long nextOffset;

    static {
        try {
            stateOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("state"));
            headOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("head"));
            tailOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("tail"));
            waitStatusOffset = unsafe.objectFieldOffset
                (Node.class.getDeclaredField("waitStatus"));
            nextOffset = unsafe.objectFieldOffset
                (Node.class.getDeclaredField("next"));

        } catch (Exception ex) { throw new Error(ex); }
    }

    /**
     * CAS head field. Used only by enq.
     */
    private final boolean compareAndSetHead(Node update) {
        return unsafe.compareAndSwapObject(this, headOffset, null, update);
    }

    /**
     * CAS tail field. Used only by enq.
     */
    private final boolean compareAndSetTail(Node expect, Node update) {
        return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
    }

    /**
     * CAS waitStatus field of a node.
     */
    private static final boolean compareAndSetWaitStatus(Node node,
                                                         int expect,
                                                         int update) {
        return unsafe.compareAndSwapInt(node, waitStatusOffset,
                                        expect, update);
    }

    /**
     * CAS next field of a node.
     */
    private static final boolean compareAndSetNext(Node node,
                                                   Node expect,
                                                   Node update) {
```

```
        return unsafe.compareAndSwapObject(node, nextOffset, expect, update);
    }
}
```

## CountDownLatch

```
package java.util.concurrent;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class CountDownLatch {
    /**
     * Synchronization control For CountDownLatch.
     * Uses AQS state to represent count.
     */
    private static final class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 4982264981922014374L;

        Sync(int count) {
            setState(count);
        }

        int getCount() {
            return getState();
        }

        protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }

        protected boolean tryReleaseShared(int releases) {
            // Decrement count; signal when transition to zero
            for (;;) {
                int c = getState();
                if (c == 0)
                    return false;
                int nextc = c-1;
                if (compareAndSetState(c, nextc))
                    return nextc == 0;
            }
        }
    }

    private final Sync sync;

    /**
     * Constructs a {@code CountDownLatch} initialized with the given count.
     *
     * @param count the number of times {@link #countDown} must be invoked
     *        before threads can pass through {@link #await}
     * @throws IllegalArgumentException if {@code count} is negative
     */
```

```java
    public CountDownLatch(int count) {
        if (count < 0) throw new IllegalArgumentException("count < 0");
        this.sync = new Sync(count);
    }

    /**
     * Causes the current thread to wait until the latch has counted down to
     * zero, unless the thread is {@linkplain Thread#interrupt interrupted}.
     *
     * <p>If the current count is zero then this method returns immediately.
     *
     * <p>If the current count is greater than zero then the current
     * thread becomes disabled for thread scheduling purposes and lies
     * dormant until one of two things happen:
     * <ul>
     * <li>The count reaches zero due to invocations of the
     * {@link #countDown} method; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread.
     * </ul>
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     *
     * @throws InterruptedException if the current thread is interrupted
     *         while waiting
     */
    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    /**
     * Causes the current thread to wait until the latch has counted down to
     * zero, unless the thread is {@linkplain Thread#interrupt interrupted},
     * or the specified waiting time elapses.
     *
     * <p>If the current count is zero then this method returns immediately
     * with the value {@code true}.
     *
     * <p>If the current count is greater than zero then the current
     * thread becomes disabled for thread scheduling purposes and lies
     * dormant until one of three things happen:
     * <ul>
     * <li>The count reaches zero due to invocations of the
     * {@link #countDown} method; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     * <li>The specified waiting time elapses.
     * </ul>
     *
     * <p>If the count reaches zero then the method returns with the
     * value {@code true}.
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.  If the time is less than or equal to zero, the method
     * will not wait at all.
     *
     * @param timeout the maximum time to wait
     * @param unit the time unit of the {@code timeout} argument
     * @return {@code true} if the count reached zero and {@code false}
     *         if the waiting time elapsed before the count reached zero
     * @throws InterruptedException if the current thread is interrupted
     *         while waiting
     */
    public boolean await(long timeout, TimeUnit unit)
        throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }

    /**
     * Decrements the count of the latch, releasing all waiting threads if
     * the count reaches zero.
     *
     * <p>If the current count is greater than zero then it is decremented.
     * If the new count is zero then all waiting threads are re-enabled for
     * thread scheduling purposes.
     *
     * <p>If the current count equals zero then nothing happens.
     */
    public void countDown() {
        sync.releaseShared(1);
    }

    /**
     * Returns the current count.
     *
     * <p>This method is typically used for debugging and testing purposes.
     *
     * @return the current count
     */
    public long getCount() {
        return sync.getCount();
    }
    /**
     * Returns a string identifying this latch, as well as its state.
     * The state, in brackets, includes the String {@code "Count ="}
     * followed by the current count.
     *
     * @return a string identifying this latch, as well as its state
     */
    public String toString() {
        return super.toString() + "[Count = " + sync.getCount() + "]";
    }
}
```

# JUC锁机制

## ReentrantLock

```
package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.Collection;

public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    private final Sync sync;

    /**
     * Base of synchronization control for this lock. Subclassed
     * into fair and nonfair versions below. Uses AQS state to
     * represent the number of holds on the lock.
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = -5179523762034025860L;

        /**
         * Performs {@link Lock#lock}. The main reason for subclassing
         * is to allow fast path for nonfair version.
         */
        abstract void lock();

        /**
         * 非公平锁标准获取锁方法(这个方法只有NonfairSync类中被调用，所以放在公共的Sync类中是不合适的)
         */
        final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            // 当执行到这里时，正好持有锁的线程释放了锁，那么可以继续尝试抢锁
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            // 当前线程就是持有锁的线程，表明锁重入
            else if (current == getExclusiveOwnerThread()) {
                // 利用state整型变量进行次数记录。
                int nextc = c + acquires;
                // 如果超过了int表示范围(超出了整型的最大值)，表明符号溢出，所以抛出异常
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            // 返回false 表明需要AQS来将当前线程放入阻塞队列，然后进行阻塞操作等待唤醒获取锁
            return false;
        }

        // 公平锁和非公平锁公用方法，因为在释放锁的时候并不区分是否公平
        protected final boolean tryRelease(int releases) {
            int c = getState() - releases;
            // 如果当前线程不是上锁的线程
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
            // 不是重入锁，那么当前线程一定是释放锁了，然后我们把当前AQS用于保存当前对象的变量
ExclusiveOwnerThread设置为null，表明释放锁成功
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null);
            }
            // 注意：此时state全局变量没有改变，也就意味着在setState之前，没有别的线程能够获取锁，这时保
证了以上操作的原子性。
            setState(c);
            // AQS会根据free的值判断，是否去唤醒正在等待锁的线程
            return free;
        }

        protected final boolean isHeldExclusively() {
            // While we must in general read state before owner,
            // we don't need to do so to check if current thread is owner
            return getExclusiveOwnerThread() == Thread.currentThread();
        }

        final ConditionObject newCondition() {
            return new ConditionObject();
        }

        // Methods relayed from outer class

        final Thread getOwner() {
            return getState() == 0 ? null : getExclusiveOwnerThread();
        }
        final int getHoldCount() {
            return isHeldExclusively() ? getState() : 0;
        }

        final boolean isLocked() {
            return getState() != 0;
        }

        /**
         * Reconstitutes the instance from a stream (that is, deserializes it).
         */
        private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
            s.defaultReadObject();
            setState(0); // reset to unlocked state
        }
    }

    /**
     * Sync object for non-fair locks
     */
    static final class NonfairSync extends Sync {
        private static final long serialVersionUID = 7316153563782823691L;

        /**
         * 由ReentrantLock调用获取锁
         */
        final void lock() {
            // 非公平锁，直接抢锁，不管有没有线程排队
            if (compareAndSetState(0, 1))
                // 上锁成功，那么表示当前线程为获取锁的线程
                setExclusiveOwnerThread(Thread.currentThread());
            else
                // 抢锁失败，进入AQS的标准获取锁流程
                acquire(1);
        }

        protected final boolean tryAcquire(int acquires) {

            // 使用父类提供的获取非公平锁的方法来获取锁
            return nonfairTryAcquire(acquires);
        }
    }

    /**
     * Sync object for fair locks
     */

    static final class FairSync extends Sync {
        private static final long serialVersionUID = -3000897897090466540L;
        /**
         * 由ReentrantLock调用
         */
        final void lock() {
            // 没有尝试抢锁，直接进入AQS标准获取锁流程
            acquire(1);
        }

        /**
         * AQS调用，子类自己实现获取锁的流程
         */
        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            // 此时有可能正好持有锁的线程释放了锁，也有可能本身就没有线程获取锁
            if (c == 0) {
                // 注意：这里和非公平锁的区别在于：hasQueuedPredecessors会去检查队列中是否有线程正在排队，如果有线程排队的话，进入else if，判断持有锁的线程是不是当前线程，如果是的话表示锁重入，如果不是的话返回false让AQS来将当前线程放入阻塞队列，然后进行阻塞操作等待唤醒获取锁，
                // hasQueuedPredecessors如果返回false，表示没有线程在排队所以通过CAS抢锁
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    // 抢锁成功
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            // 当前线程就是持有锁的线程，那么这里时锁重入
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
    }

    /**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     */
    public ReentrantLock() {
        // 默认为非公平锁。原因：因为大量测试下来，发现非公平锁性能优于公平锁
        sync = new NonfairSync();
    }

    /**
     * Creates an instance of {@code ReentrantLock} with the
     * given fairness policy.
     *
     * @param fair {@code true} if this lock should use a fair ordering policy
     */
    public ReentrantLock(boolean fair) {
        // 由fair变量来表明选择锁类型
        sync = fair ? new FairSync() : new NonfairSync();
    }

    /**
     * Acquires the lock.
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately, setting the lock hold count to one.
     *
     * <p>If the current thread already holds the lock then the hold
     * count is incremented by one and the method returns immediately.
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until the lock has been acquired,
     * at which time the lock hold count is set to one.
     */
    public void lock() {
        sync.lock();
    }

    /**
     * Acquires the lock unless the current thread is
     * {@linkplain Thread#interrupt interrupted}.
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately, setting the lock hold count to one.
     *
     * <p>If the current thread already holds this lock then the hold count
     * is incremented by one and the method returns immediately.
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of two things happens:
     *
     * <ul>
     *
     * <li>The lock is acquired by the current thread; or
     *
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread.
     *
     * </ul>
     *
     * <p>If the lock is acquired by the current thread then the lock hold
     * count is set to one.
     *
     * <p>If the current thread:
     *
     * <ul>
     *
     * <li>has its interrupted status set on entry to this method; or
```

```java
     *
     * <li>is {@linkplain Thread#interrupt interrupted} while acquiring
     * the lock,
     *
     * </ul>
     *
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     *
     * <p>In this implementation, as this method is an explicit
     * interruption point, preference is given to responding to the
     * interrupt over normal or reentrant acquisition of the lock.
     *
     * @throws InterruptedException if the current thread is interrupted
     */
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    /**
     * Acquires the lock only if it is not held by another thread at the time
     * of invocation.
     *
     * <p>Acquires the lock if it is not held by another thread and
     * returns immediately with the value {@code true}, setting the
     * lock hold count to one. Even when this lock has been set to use a
     * fair ordering policy, a call to {@code tryLock()} <em>will</em>
     * immediately acquire the lock if it is available, whether or not
     * other threads are currently waiting for the lock.
     * This &quot;barging&quot; behavior can be useful in certain
     * circumstances, even though it breaks fairness. If you want to honor
     * the fairness setting for this lock, then use
     * {@link #tryLock(long, TimeUnit) tryLock(0, TimeUnit.SECONDS) }
     * which is almost equivalent (it also detects interruption).
     *
     * <p>If the current thread already holds this lock then the hold
     * count is incremented by one and the method returns {@code true}.
     *
     * <p>If the lock is held by another thread then this method will return
     * immediately with the value {@code false}.
     *
     * @return {@code true} if the lock was free and was acquired by the
     *         current thread, or the lock was already held by the current
     *         thread; and {@code false} otherwise
     */
    public boolean tryLock() {
        return sync.nonfairTryAcquire(1);
    }

    /**
     * Acquires the lock if it is not held by another thread within the given
     * waiting time and the current thread has not been
     * {@linkplain Thread#interrupt interrupted}.
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately with the value {@code true}, setting the lock hold count
     * to one. If this lock has been set to use a fair ordering policy then
     * an available lock <em>will not</em> be acquired if any other threads
     * are waiting for the lock. This is in contrast to the {@link #tryLock()}
     * method. If you want a timed {@code tryLock} that does permit barging on
     * a fair lock then combine the timed and un-timed forms together:
     *
     *  <pre> {@code
     * if (lock.tryLock() ||
     *     lock.tryLock(timeout, unit)) {
     *   ...
     * }}</pre>
     *
     * <p>If the current thread
     * already holds this lock then the hold count is incremented by one and
     * the method returns {@code true}.
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of three things happens:
     *
     * <ul>
     *
     * <li>The lock is acquired by the current thread; or
     *
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     *
     * <li>The specified waiting time elapses
     *
     * </ul>
     *
     * <p>If the lock is acquired then the value {@code true} is returned and
     * the lock hold count is set to one.
     *
     * <p>If the current thread:
     *
     * <ul>
     *
     * <li>has its interrupted status set on entry to this method; or
     *
     * <li>is {@linkplain Thread#interrupt interrupted} while
     * acquiring the lock,
     *
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.  If the time is less than or equal to zero, the method
     * will not wait at all.
```

```java
     *
     * <p>In this implementation, as this method is an explicit
     * interruption point, preference is given to responding to the
     * interrupt over normal or reentrant acquisition of the lock, and
     * over reporting the elapse of the waiting time.
     *
     * @param timeout the time to wait for the lock
     * @param unit the time unit of the timeout argument
     * @return {@code true} if the lock was free and was acquired by the
     *         current thread, or the lock was already held by the current
     *         thread; and {@code false} if the waiting time elapsed before
     *         the lock could be acquired
     * @throws InterruptedException if the current thread is interrupted
     * @throws NullPointerException if the time unit is null
     */
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }

    /**
     * Attempts to release this lock.
     *
     * <p>If the current thread is the holder of this lock then the hold
     * count is decremented.  If the hold count is now zero then the lock
     * is released.  If the current thread is not the holder of this
     * lock then {@link IllegalMonitorStateException} is thrown.
     *
     * @throws IllegalMonitorStateException if the current thread does not
     *         hold this lock
     */
    public void unlock() {
        sync.release(1);
    }

    /**
     * Returns a {@link Condition} instance for use with this
     * {@link Lock} instance.
     *
     * <p>The returned {@link Condition} instance supports the same
     * usages as do the {@link Object} monitor methods ({@link
     * Object#wait() wait}, {@link Object#notify notify}, and {@link
     * Object#notifyAll notifyAll}) when used with the built-in
     * monitor lock.
     *
     * <ul>
     *
     * <li>If this lock is not held when any of the {@link Condition}
     * {@linkplain Condition#await() waiting} or {@linkplain
     * Condition#signal signalling} methods are called, then an {@link
     * IllegalMonitorStateException} is thrown.
     *
     * <li>When the condition {@linkplain Condition#await() waiting}
     * methods are called the lock is released and, before they
     * return, the lock is reacquired and the lock hold count restored
     * to what it was when the method was called.
     *
     * <li>If a thread is {@linkplain Thread#interrupt interrupted}
     * while waiting then the wait will terminate, an {@link
     * InterruptedException} will be thrown, and the thread's
     * interrupted status will be cleared.
     *
     * <li> Waiting threads are signalled in FIFO order.
     *
     * <li>The ordering of lock reacquisition for threads returning
     * from waiting methods is the same as for threads initially
     * acquiring the lock, which is in the default case not specified,
     * but for <em>fair</em> locks favors those threads that have been
     * waiting the longest.
     *
     * </ul>
     *
     * @return the Condition object
     */
    public Condition newCondition() {
        return sync.newCondition();
    }

    /**
     * Queries the number of holds on this lock by the current thread.
     *
     * <p>A thread has a hold on a lock for each lock action that is not
     * matched by an unlock action.
     *
     * <p>The hold count information is typically only used for testing and
     * debugging purposes. For example, if a certain section of code should
     * not be entered with the lock already held then we can assert that
     * fact:
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *   public void m() {
     *     assert lock.getHoldCount() == 0;
     *     lock.lock();
     *     try {
     *       // ... method body
     *     } finally {
     *       lock.unlock();
     *     }
     *   }
     * }}</pre>
     *
     * @return the number of holds on this lock by the current thread,
     *         or zero if this lock is not held by the current thread
```

```java
     */
    public int getHoldCount() {
        return sync.getHoldCount();
    }

    /**
     * Queries if this lock is held by the current thread.
     *
     * <p>Analogous to the {@link Thread#holdsLock(Object)} method for
     * built-in monitor locks, this method is typically used for
     * debugging and testing. For example, a method that should only be
     * called while a lock is held can assert that this is the case:
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *
     *   public void m() {
     *       assert lock.isHeldByCurrentThread();
     *       // ... method body
     *   }
     * }}</pre>
     *
     * <p>It can also be used to ensure that a reentrant lock is used
     * in a non-reentrant manner, for example:
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *
     *   public void m() {
     *       assert !lock.isHeldByCurrentThread();
     *       lock.lock();
     *       try {
     *           // ... method body
     *       } finally {
     *           lock.unlock();
     *       }
     *   }
     * }}</pre>
     *
     * @return {@code true} if current thread holds this lock and
     *         {@code false} otherwise
     */
    public boolean isHeldByCurrentThread() {
        return sync.isHeldExclusively();
    }

    /**
     * Queries if this lock is held by any thread. This method is
     * designed for use in monitoring of the system state,
     * not for synchronization control.
     *
     * @return {@code true} if any thread holds this lock and
     *         {@code false} otherwise
     */
    public boolean isLocked() {
        return sync.isLocked();
    }

    /**
     * Returns {@code true} if this lock has fairness set true.
     *
     * @return {@code true} if this lock has fairness set true
     */
    public final boolean isFair() {
        return sync instanceof FairSync;
    }

    /**
     * Returns the thread that currently owns this lock, or
     * {@code null} if not owned. When this method is called by a
     * thread that is not the owner, the return value reflects a
     * best-effort approximation of current lock status. For example,
     * the owner may be momentarily {@code null} even if there are
     * threads trying to acquire the lock but have not yet done so.
     * This method is designed to facilitate construction of
     * subclasses that provide more extensive lock monitoring
     * facilities.
     *
     * @return the owner, or {@code null} if not owned
     */
    protected Thread getOwner() {
        return sync.getOwner();
    }

    /**
     * Queries whether any threads are waiting to acquire this lock. Note that
     * because cancellations may occur at any time, a {@code true}
     * return does not guarantee that any other thread will ever
     * acquire this lock.  This method is designed primarily for use in
     * monitoring of the system state.
     *
     * @return {@code true} if there may be other threads waiting to
     *         acquire the lock
     */
    public final boolean hasQueuedThreads() {
        return sync.hasQueuedThreads();
    }

    /**
     * Queries whether the given thread is waiting to acquire this
     * lock. Note that because cancellations may occur at any time, a
     * {@code true} return does not guarantee that this thread
     * will ever acquire this lock.  This method is designed primarily for use
     * in monitoring of the system state.
     *
     * @param thread the thread
     * @return {@code true} if the given thread is queued waiting for this lock
     * @throws NullPointerException if the thread is null
     */
    public final boolean hasQueuedThread(Thread thread) {
        return sync.isQueued(thread);
    }

    /**
     * Returns an estimate of the number of threads waiting to
     * acquire this lock.  The value is only an estimate because the number of
     * threads may change dynamically while this method traverses
     * internal data structures.  This method is designed for use in
     * monitoring of the system state, not for synchronization
     * control.
     *
     * @return the estimated number of threads waiting for this lock
     */
    public final int getQueueLength() {
        return sync.getQueueLength();
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire this lock.  Because the actual set of threads may change
     * dynamically while constructing this result, the returned
     * collection is only a best-effort estimate.  The elements of the
     * returned collection are in no particular order.  This method is
     * designed to facilitate construction of subclasses that provide
     * more extensive monitoring facilities.
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedThreads() {
        return sync.getQueuedThreads();
    }

    /**
     * Queries whether any threads are waiting on the given condition
     * associated with this lock. Note that because timeouts and
     * interrupts may occur at any time, a {@code true} return does
     * not guarantee that a future {@code signal} will awaken any
     * threads.  This method is designed primarily for use in
     * monitoring of the system state.
     *
     * @param condition the condition
     * @return {@code true} if there are any waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public boolean hasWaiters(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return sync.hasWaiters((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns an estimate of the number of threads waiting on the
     * given condition associated with this lock. Note that because
     * timeouts and interrupts may occur at any time, the estimate
     * serves only as an upper bound on the actual number of waiters.
     * This method is designed for use in monitoring of the system
     * state, not for synchronization control.
     *
     * @param condition the condition
     * @return the estimated number of waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public int getWaitQueueLength(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return
sync.getWaitQueueLength((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns a collection containing those threads that may be
     * waiting on the given condition associated with this lock.
     * Because the actual set of threads may change dynamically while
     * constructing this result, the returned collection is only a
     * best-effort estimate. The elements of the returned collection
     * are in no particular order.  This method is designed to
     * facilitate construction of subclasses that provide more
     * extensive condition monitoring facilities.
     *
     * @param condition the condition
     * @return the collection of threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    protected Collection<Thread> getWaitingThreads(Condition condition) {
        if (condition == null)
```

```
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return
sync.getWaitingThreads((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns a string identifying this lock, as well as its lock state.
     * The state, in brackets, includes either the String {@code "Unlocked"}
     * or the String {@code "Locked by"} followed by the
     * {@linkplain Thread#getName name} of the owning thread.
     *
     * @return a string identifying this lock, as well as its lock state
     */
    public String toString() {
        Thread o = sync.getOwner();
        return super.toString() + ((o == null) ?
                                   "[Unlocked]" :
                                   "[Locked by thread " + o.getName() + "]");
    }
}
```

## ReentrantReadWriteLock

将原来的锁分为两把锁：读锁、写锁。适用于读多写少的场景，读锁可以并发，写锁与其他锁互斥。写写互斥、写读互斥、读读兼容。

线程进入读锁的前提条件：

1. 没有其他线程的写锁，
2. 没有写请求或者有写请求，但调用线程和持有锁的线程是同一个。

线程进入写锁的前提条件：

1. 没有其他线程的读锁
2. 没有其他线程的写锁

而读写锁有以下三个重要的特性：

1. 公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。
2. 重进入：读锁和写锁都支持线程重进入。
3. 锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁。

```
package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.Collection;

/**
 * ReentrantReadWriteLock实现了ReadWriteLock接口，ReadWriteLock接口定义了获取读锁和写锁的规范，具体需要实现类去实现；同时其还实现了Serializable接口，表示可以进行序列化，在源代码中可以看到ReentrantReadWriteLock实现了自己的序列化逻辑。
 */
public class ReentrantReadWriteLock
        implements ReadWriteLock, java.io.Serializable {
    private static final long serialVersionUID = -6992448646407690164L;
    /** 读锁 */
    private final ReentrantReadWriteLock.ReadLock readerLock;
    /** 写锁 */
    private final ReentrantReadWriteLock.WriteLock writerLock;
    /** Performs all synchronization mechanics */
    final Sync sync;

    /**
     * 使用默认（非公平）的公平策略创建一个新的ReentrantReadWriteLock
     */
    public ReentrantReadWriteLock() {
        this(false);
    }

    /**
     * 使用给定的公平策略创建一个新的 ReentrantReadWriteLock
     */
    public ReentrantReadWriteLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
        readerLock = new ReadLock(this);
        writerLock = new WriteLock(this);
    }

    /** 返回用于写入操作的锁 */
    public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
    /** 返回用于读取操作的锁 */
    public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }

    /**
     * Sync抽象类继承自AQS抽象类，Sync类提供了对ReentrantReadWriteLock的支持
     * NonfairSync继承自Sync类、FairSync继承自Sync类（通过构造函数传入的布尔值决定要构造哪一种Sync实
例）
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 6317671515068378041L;
        // 同步状态(AbstractQueuedSynchronizer的state变量)在重入锁的实现中是表示被同一个线程重复获取
的次数，即一个整形变量来维护，但是之前的那个表示仅仅表示是否锁定，而不用区分是读锁还是写锁。而读写锁需要在同步
状态（一个整形变量）上维护多个读线程和一个写线程的状态。
        // 高16位为读锁，低16位为写锁
        static final int SHARED_SHIFT   = 16;
        // 读锁单位(每执行一次ReadLock.lock()增加SHARED_UNIT)
        static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
        // 读锁最大数量
        static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
        // 掩码，直接将状态state和EXCLUSIVE_MASK做与运算来获取写锁数量
        static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
        /** 获取读锁数量  */
        static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
        /** 获取写锁数量  */
        static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }

        /**
         * HoldCounter中仅有count和tid两个变量，其中count代表着计数器，tid是线程的id，HoldCounter不能起到绑定对象的作用，只是记录线程tid而已
         * HoldCounter也持有线程Id，这样在释放锁的时候才能知道ReadWriteLock里面缓存的上一个读取线程（cachedHoldCounter）是否是当前线程。这样做的好处是可以减少ThreadLocal.get()的次数，因为这也是一个耗时操作。需要说明的是这样HoldCounter绑定线程id而不绑定线程对象的原因是避免HoldCounter和ThreadLocal互相绑定而GC难以释放它们（尽管GC能够智能的发现这种引用而回收它们，但是这需要一定的代价），所以其实这样做只是为了帮助GC快速回收对象而已。
         */
        static final class HoldCounter {
            // 读线程重入的次数
            int count = 0;
            // 该线程的tid字段的值
            final long tid = getThreadId(Thread.currentThread());
        }

        /**
         * HoldCounter就是绑定在线程上的一个计数器，而ThradLocalHoldCounter则是正真将线程和HoldCounter绑定的ThreadLocal
         * ThreadLocalHoldCounter重写了ThreadLocal的initialValue方法，ThreadLocal类可以将线程与对象相关联。在没有进行set的情况下，get到的均是initialValue方法里面生成的那个HolderCounter对象。
         */
        static final class ThreadLocalHoldCounter
            extends ThreadLocal<HoldCounter> {
            // 重写初始化方法，在没有进行set的情况下，获取的都是该HoldCounter值
            public HoldCounter initialValue() {
                return new HoldCounter();
            }
        }

        /**
         * 当前线程持有的可重入读锁数.
         * 仅在构造函数和readObject中初始化.
         * 每当线程的读取保持计数变到0时删除.
         */
        private transient ThreadLocalHoldCounter readHolds;

        /**
         * 最后一个成功获取读锁的线程的HoldCounter
         */
        private transient HoldCounter cachedHoldCounter;

        /**
         * 第一个成功获取读锁的线程.
         * 第一个成功获取读锁的线程的HoldCounter.
         * firstReader是不会放入到readHolds中的，如果读锁仅有一个的情况下就会避免查找readHolds，提高性能
         */
        private transient Thread firstReader = null;
        private transient int firstReaderHoldCount;
        Sync() {
            readHolds = new ThreadLocalHoldCounter();
            setState(getState()); // ensures visibility of readHolds
        }

        /*
         * Acquires and releases use the same code for fair and
         * nonfair locks, but differ in whether/how they allow barging
         * when queues are non-empty.
         */

        /**
         * Returns true if the current thread, when trying to acquire
         * the read lock, and otherwise eligible to do so, should block
         * because of policy for overtaking other waiting threads.
         */
        abstract boolean readerShouldBlock();

        /**
         * Returns true if the current thread, when trying to acquire
         * the write lock, and otherwise eligible to do so, should block
         * because of policy for overtaking other waiting threads.
         */
        abstract boolean writerShouldBlock();

        /*
         * 写锁的释放
         */
        protected final boolean tryRelease(int releases) {
            // 若锁的持有者不是当前线程，抛出异常
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            int nextc = getState() - releases;
            // 如果独占模式重入次数为0了，说明独占模式被释放
            boolean free = exclusiveCount(nextc) == 0;
            if (free)
                // 若写锁的新线程数为0，则将锁的持有者设置为null
                setExclusiveOwnerThread(null);
            // 设置写锁的新线程数
            // 不管独占模式是否被释放，更新独占重入数
            setState(nextc);
            return free;
        }

        /**
         * 写锁的获取
         */
        protected final boolean tryAcquire(int acquires) {

            Thread current = Thread.currentThread();
            // 状态获取
            int c = getState();
            // 写线程数量（即获取独占锁的重入次数）
            int w = exclusiveCount(c);
            // 当前同步状态state != 0，说明已经有其他线程获取了读锁或写锁
            if (c != 0) {
                // 当前state不为0，此时：如果写锁状态为0说明读锁此时被占用，返回false；
                // 如果写锁状态不为0且写锁没有被当前线程持有，返回false
                if (w == 0 || current != getExclusiveOwnerThread())
                    return false;
                // 判断同一线程获取写锁是否超过最大次数（65535），支持可重入
                if (w + exclusiveCount(acquires) > MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");
                // 更新状态
                // 此时当前线程已持有写锁，现在是重入，所以只需要修改锁的数量即可。
                setState(c + acquires);
                return true;
            }
            // 到这里说明此时c=0,读锁和写锁都没有被获取
            // writerShouldBlock判断当前线程是否需要阻塞，
            // 在非公平策略下总是返回false从而进入下面的CAS抢锁操作，
            // 在公平锁策略下，会通过AQS的hasQueuedPredecessors方法来判断有没有别的线程排队排在当前线
程的前面(即判断当前线程是不是CHL队列的第一个线程)
            if (writerShouldBlock() ||
                !compareAndSetState(c, c + acquires))
                return false;
            // 设置锁为当前线程所有
            setExclusiveOwnerThread(current);
            return true;
        }

        /**
         * 读锁的释放
         */
        protected final boolean tryReleaseShared(int unused) {
            Thread current = Thread.currentThread();
            // 当前线程为第一个读线程
            if (firstReader == current) {
                // assert firstReaderHoldCount > 0;
                // 读线程占用的资源数为1
                if (firstReaderHoldCount == 1)
                    firstReader = null;
                else
                    firstReaderHoldCount--;
            // 当前线程不为第一个读线程
            } else {
                // 那么首先会获取缓存计数器（最后一个读锁线程对应的计数器 ）
                HoldCounter rh = cachedHoldCounter;
                // 若计数器为空或者tid不等于当前线程的tid值
                if (rh == null || rh.tid != getThreadId(current))
                    // 则获取当前线程的计数器
                    rh = readHolds.get();
                int count = rh.count;
                // 如果计数器的计数count小于等于1，则移除当前线程对应的计数器
                if (count <= 1) {
                    readHolds.remove();
                    if (count <= 0)
                        throw unmatchedUnlockException();
                }
                --rh.count;
            }
            // 死循环确保成功设置状态state：读锁数量减一
            for (;;) {
                int c = getState();
                int nextc = c - SHARED_UNIT;
                if (compareAndSetState(c, nextc))
                    // Releasing the read lock has no effect on readers,
                    // but it may allow waiting writers to proceed if
                    // both read and write locks are now free.
                    return nextc == 0;
            }
        }

        private IllegalMonitorStateException unmatchedUnlockException() {
            return new IllegalMonitorStateException(
                "attempt to unlock read lock, not locked by current thread");
        }

        /**
         * 读锁的获取
         */
        protected final int tryAcquireShared(int unused) {
            // 获取当前线程
            Thread current = Thread.currentThread();
            // 获取状态
            int c = getState();
            // 如果写锁线程数!= 0，且独占锁不是当前线程，则返回失败
            // 如果写锁线程数!= 0，且独占锁是当前线程，锁降级
            if (exclusiveCount(c) != 0 &&
                getExclusiveOwnerThread() != current)
                return -1;
            // 读锁数量
            int r = sharedCount(c);
            // 当前读线程不需要阻塞、并且小于最大值、并且CAS抢锁成功
            // 这里如果不进行读阻塞判断的话会产生写锁饥饿的线程：
            // 比如系统中的读线程一直源源不断的在读取数据，这个时候来了一个写线程，由于获取不到写锁那么它肯定会进入排队的队列中，由于读线程太多，所以在某一个线程完成读取后调用tryReleaseShared方法释放锁的时候，由于不断的有新的读线程获取到读锁，使得始终无法将AQS的state的值变成0也就无法唤醒阻塞中的写线程，因此产生了写锁饥饿的情况
            // 如果在这里对读线程进行阻塞判断：当阻塞队列的第一个等待节点是互斥模式(非公平)或者第一个等待节点不是当前线程(公平锁)的时候当前读线程都会进入阻塞队列，这样的话当阻塞队里有等待节点后新的读线程就无法直接获取读锁了必须进入队列中排队，等老的读线程全部完成读操作后就会释放全部的读锁而且由于新的读线程都进入队列中排队了，所以最终会使得state的值变成0，这样就会唤醒在阻塞队列中排队的写线程。这样做是为了让后续的写线程有抢占写锁的机会，不会因为一直有读线程或者锁降级情况的存在而造成后续写线程的饥饿等待。
            if (!readerShouldBlock() &&
                r < MAX_COUNT &&
                compareAndSetState(c, c + SHARED_UNIT)) {
                // 更新成功后会在firstReaderHoldCount中或readHolds(ThreadLocal类型的)的本线程副本中记录当前线程重入数，这是为了实现jdk1.6中加入的getReadHoldCount()方法的，这个方法能获取当前线程重入共享锁的次数(state中记录的是多个线程的总重入次数)，
                // 原理：如果当前只有一个线程的话，还不需要动用ThreadLocal，直接往firstReaderHoldCount这个成员变量里存重入数，当有第二个线程来的时候，就要动用ThreadLocal变量readHolds了，每个线程拥有自己的副本，用来保存自己的重入数。
                // 读锁数量为0，表示第一个读线程，第一个读线程firstRead是不会加入到readHolds中
                if (r == 0) {
                    // 设置第一个读线程
                    firstReader = current;
                    // 读线程占用的资源数为1
                    firstReaderHoldCount = 1;
                // 当前线程为第一个读线程，表示第一个读锁线程重入
                } else if (firstReader == current) {
                    // 占用资源数加1
                    firstReaderHoldCount++;
                // 读锁数量不为0并且不为第一个读线程(表示第二个读线程进来获取锁了)
                } else {
                    // cachedHoldCounter代表的是最后一个获取读锁的线程的计数器。
                    HoldCounter rh = cachedHoldCounter;
                    // 如果最后一个线程计数器是null或者不是当前线程，那么就新建一个HoldCounter对象
                    if (rh == null || rh.tid != getThreadId(current))
                        // 给当前线程新建一个HoldCounter(由ThreadLocal的get方法可知，当前线程的threadLocals为空时，会将threadLocals设置为initialValue方法返回的值，并返回。由于ThreadLocalHoldCounter重写了ThreadLocal的initialValue方法，所以执行get方法后线程与HoldCounter对象就关联上了，所以每次线程切换的时候等于说将当前线程关联上一个新的HoldCounter)
                        cachedHoldCounter = rh = readHolds.get();
                    // 如果不是null，且count是0，(这个地方什么情况会进入？)
                    else if (rh.count == 0)
                        // 加入到readHolds中
                        readHolds.set(rh);
                    // 计数+1
                    rh.count++;
                }
                return 1;
            }
            // 在Doug Lea的代码中经常会看到这种优化，他会把常见的场景前置保证性能；也就是说百分之九十的情况不会走到下面的fullTryAcquireShared方法中来。fullTryAcquireShared方法中的处理逻辑跟上面的代码非常类似，只是增加死循环保证线程能正常返回。
            return fullTryAcquireShared(current);
        }

        /**
         * Full version of acquire for reads, that handles CAS misses
         * and reentrant reads not dealt with in tryAcquireShared.
         */
        final int fullTryAcquireShared(Thread current) {
            /*
             * This code is in part redundant with that in
             * tryAcquireShared but is simpler overall by not
             * complicating tryAcquireShared with interactions between
             * retries and lazily reading hold counts.
             */
            HoldCounter rh = null;
            for (;;) {
                int c = getState();
                // 如果“锁”被“写锁”持有，并且获取锁的线程不是current线程；则返回-1。
                if (exclusiveCount(c) != 0) {
                    if (getExclusiveOwnerThread() != current)
                        return -1;
                    // else we hold the exclusive lock; blocking here
                    // would cause deadlock.
                // 需要阻塞等待
                } else if (readerShouldBlock()) {
                    // 进入这里，说明写锁被释放，读锁被阻塞
                    // 那么逻辑就很清楚了，这里是处理读锁重入的，在tryReleaseShared方法中可知，当某个
读线程的读锁完全释放后(该线程持有的读锁数量归零，即：firstReader==null或者HoldCounter.count==0)，再通
过tryAcquireShared获取锁的话，就不能再算锁重入了，重入锁值的是当前线程持有锁并在没有释放之前再来获取同一把
锁。
                    // Make sure we're not acquiring read lock reentrantly
                    // 下面的if和else都是判断是不是重入锁的
                    // 当firstReader == current表示当前线程持有读锁，所以是锁的重入，直接进行下半部分
的cas操作获取读锁。因为如果是第一个线程完全释放读锁后再来获取读锁，那么由于在tryReleaseShared方法中
firstReader被置为空，那么firstReader == current不成立。同样的当某个线程曾经持有过读锁但是完全释放后，由
于在tryReleaseShared中会remove掉readHolds中该线程的HoldCounter,所以再来获取读锁的话readHolds.get()
返回的是初始化的HoldCounter，因此rh.count==0，于是fullTryAcquireShared返回-1，那么该线程将进入AQS排队
队列。
                    //
                    if (firstReader == current) {
                        // assert firstReaderHoldCount > 0;
                    } else {
                        if (rh == null) {
                            // 最后一个持有读锁的线程
                            rh = cachedHoldCounter;
                            // 如果最后一个持有读锁的线程时不等于当前线程
                            if (rh == null || rh.tid != getThreadId(current)) {
                                // 则判断在readHolds中是否有当前线程关联的HoldCounter，如有的话说明
是锁重入，进入下半部分的cas操作获取读锁
                                rh = readHolds.get();
                                if (rh.count == 0)
                                    readHolds.remove();
                            }
                        }
                        // 如果当前线程从来没有获取过读锁(第一次调用tryAcquireShared方法获取锁)，那么
在readHolds肯定没有保存过HoldCounter那么它的count肯定是0.则返回-1,进入aqs的doAcquireShared将当前线程
加入到aqs的等待队列
                        if (rh.count == 0)
                            return -1;
                    }
                }
                if (sharedCount(c) == MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");
                // 如果readerShouldBlock不要求阻塞,或readerShouldBlock要求阻塞但是当前线程不是第一
次获取读锁.则进入下面的cas操作
                if (compareAndSetState(c, c + SHARED_UNIT)) {
                    if (sharedCount(c) == 0) {
```

```java
                    firstReader = current;
                    firstReaderHoldCount = 1;
                } else if (firstReader == current) {
                    firstReaderHoldCount++;
                } else {
                    if (rh == null)
                        rh = cachedHoldCounter;
                    if (rh == null || rh.tid != getThreadId(current))
                        rh = readHolds.get();
                    else if (rh.count == 0)
                        readHolds.set(rh);
                    rh.count++;
                    cachedHoldCounter = rh; // cache for release
                }
                return 1;
            }
        }
    }

    /**
     * Performs tryLock for write, enabling barging in both modes.
     * This is identical in effect to tryAcquire except for lack
     * of calls to writerShouldBlock.
     */
    final boolean tryWriteLock() {
        Thread current = Thread.currentThread();
        int c = getState();
        if (c != 0) {
            int w = exclusiveCount(c);
            if (w == 0 || current != getExclusiveOwnerThread())
                return false;
            if (w == MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
        }
        if (!compareAndSetState(c, c + 1))
            return false;
        setExclusiveOwnerThread(current);
        return true;
    }

    /**
     * Performs tryLock for read, enabling barging in both modes.
     * This is identical in effect to tryAcquireShared except for
     * lack of calls to readerShouldBlock.
     */
    final boolean tryReadLock() {
        Thread current = Thread.currentThread();
        for (;;) {
            int c = getState();
            if (exclusiveCount(c) != 0 &&
                getExclusiveOwnerThread() != current)
                return false;
            int r = sharedCount(c);
```

```java
                if (r == MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");
                if (compareAndSetState(c, c + SHARED_UNIT)) {
                    if (r == 0) {
                        firstReader = current;
                        firstReaderHoldCount = 1;
                    } else if (firstReader == current) {
                        firstReaderHoldCount++;
                    } else {
                        HoldCounter rh = cachedHoldCounter;
                        if (rh == null || rh.tid != getThreadId(current))
                            cachedHoldCounter = rh = readHolds.get();
                        else if (rh.count == 0)
                            readHolds.set(rh);
                        rh.count++;
                    }
                    return true;
                }
            }
        }

        protected final boolean isHeldExclusively() {
            // While we must in general read state before owner,
            // we don't need to do so to check if current thread is owner
            return getExclusiveOwnerThread() == Thread.currentThread();
        }

        // Methods relayed to outer class

        final ConditionObject newCondition() {
            return new ConditionObject();
        }

        final Thread getOwner() {
            // Must read state before owner to ensure memory consistency
            return ((exclusiveCount(getState()) == 0) ?
                    null :
                    getExclusiveOwnerThread());
        }

        final int getReadLockCount() {
            return sharedCount(getState());
        }

        final boolean isWriteLocked() {
            return exclusiveCount(getState()) != 0;
        }

        final int getWriteHoldCount() {
            return isHeldExclusively() ? exclusiveCount(getState()) : 0;
        }

        final int getReadHoldCount() {
            if (getReadLockCount() == 0)
                return 0;

            Thread current = Thread.currentThread();
            if (firstReader == current)
                return firstReaderHoldCount;

            HoldCounter rh = cachedHoldCounter;
            if (rh != null && rh.tid == getThreadId(current))
                return rh.count;

            int count = readHolds.get().count;
            if (count == 0) readHolds.remove();
            return count;
        }

        /**
         * Reconstitutes the instance from a stream (that is, deserializes it).
         */
        private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
            s.defaultReadObject();
            readHolds = new ThreadLocalHoldCounter();
            setState(0); // reset to unlocked state
        }

        final int getCount() { return getState(); }
    }

    /**
     * Nonfair version of Sync
     */
    static final class NonfairSync extends Sync {
        private static final long serialVersionUID = -8159625535654395037L;
        final boolean writerShouldBlock() {
            return false; // writers can always barge
        }
        final boolean readerShouldBlock() {
            /* As a heuristic to avoid indefinite writer starvation,
             * block if the thread that momentarily appears to be head
             * of queue, if one exists, is a waiting writer.  This is
             * only a probabilistic effect since a new reader will not
             * block if there is a waiting writer behind other enabled
             * readers that have not yet drained from the queue.
             */
            return apparentlyFirstQueuedIsExclusive();
        }
    }

    /**
     * Fair version of Sync
     */
    static final class FairSync extends Sync {
        private static final long serialVersionUID = -2274990926593161451L;
        final boolean writerShouldBlock() {
            return hasQueuedPredecessors();
        }
        final boolean readerShouldBlock() {
            return hasQueuedPredecessors();
        }
    }

    /**
     * The lock returned by method {@link ReentrantReadWriteLock#readLock}.
     */
    public static class ReadLock implements Lock, java.io.Serializable {
        private static final long serialVersionUID = -5992448646407690164L;
        private final Sync sync;

        /**
         * Constructor for use by subclasses
         *
         * @param lock the outer lock object
         * @throws NullPointerException if the lock is null
         */
        protected ReadLock(ReentrantReadWriteLock lock) {
            sync = lock.sync;
        }

        /**
         * Acquires the read lock.
         *
         * <p>Acquires the read lock if the write lock is not held by
         * another thread and returns immediately.
         *
         * <p>If the write lock is held by another thread then
         * the current thread becomes disabled for thread scheduling
         * purposes and lies dormant until the read lock has been acquired.
         */
        public void lock() {
            sync.acquireShared(1);
        }

        /**
         * Acquires the read lock unless the current thread is
         * {@linkplain Thread#interrupt interrupted}.
         *
         * <p>Acquires the read lock if the write lock is not held
         * by another thread and returns immediately.
         *
         * <p>If the write lock is held by another thread then the
         * current thread becomes disabled for thread scheduling
         * purposes and lies dormant until one of two things happens:
         *
         * <ul>
         *
         * <li>The read lock is acquired by the current thread; or
         *
         * <li>Some other thread {@linkplain Thread#interrupt interrupts}
         * the current thread.
         *
         * </ul>
         *
         * <p>If the current thread:
         *
         * <ul>
         *
         * <li>has its interrupted status set on entry to this method; or
         *
         * <li>is {@linkplain Thread#interrupt interrupted} while
         * acquiring the read lock,
         *
         * </ul>
         *
         * then {@link InterruptedException} is thrown and the current
         * thread's interrupted status is cleared.
         *
         * <p>In this implementation, as this method is an explicit
         * interruption point, preference is given to responding to
         * the interrupt over normal or reentrant acquisition of the
         * lock.
         *
         * @throws InterruptedException if the current thread is interrupted
         */
        public void lockInterruptibly() throws InterruptedException {
            sync.acquireSharedInterruptibly(1);
        }

        /**
         * Acquires the read lock only if the write lock is not held by
         * another thread at the time of invocation.
         *
         * <p>Acquires the read lock if the write lock is not held by
         * another thread and returns immediately with the value
         * {@code true}. Even when this lock has been set to use a
         * fair ordering policy, a call to {@code tryLock()}
         * <em>will</em> immediately acquire the read lock if it is
         * available, whether or not other threads are currently
         * waiting for the read lock.  This &quot;barging&quot; behavior
         * can be useful in certain circumstances, even though it
         * breaks fairness. If you want to honor the fairness setting
         * for this lock, then use {@link #tryLock(long, TimeUnit)
         * tryLock(0, TimeUnit.SECONDS) } which is almost equivalent
         * (it also detects interruption).
         *
         * <p>If the write lock is held by another thread then
         * this method will return immediately with the value
         * {@code false}.
         *
         * @return {@code true} if the read lock was acquired
         */
        public boolean tryLock() {
            return sync.tryReadLock();
        }

        /**
         * Acquires the read lock if the write lock is not held by
         * another thread within the given waiting time and the
         * current thread has not been {@linkplain Thread#interrupt
         * interrupted}.
         *
         * <p>Acquires the read lock if the write lock is not held by
         * another thread and returns immediately with the value
         * {@code true}. If this lock has been set to use a fair
         * ordering policy then an available lock <em>will not</em> be
         * acquired if any other threads are waiting for the
         * lock. This is in contrast to the {@link #tryLock()}
         * method. If you want a timed {@code tryLock} that does
         * permit barging on a fair lock then combine the timed and
         * un-timed forms together:
         *
         *  <pre> {@code
         * if (lock.tryLock() ||
         *     lock.tryLock(timeout, unit)) {
         *   ...
         * }}</pre>
         *
         * <p>If the write lock is held by another thread then the
         * current thread becomes disabled for thread scheduling
         * purposes and lies dormant until one of three things happens:
         *
         * <ul>
         *
         * <li>The read lock is acquired by the current thread; or
         *
         * <li>Some other thread {@linkplain Thread#interrupt interrupts}
         * the current thread; or
         *
         * <li>The specified waiting time elapses.
         *
         * </ul>
         *
         * <p>If the read lock is acquired then the value {@code true} is
         * returned.
         *
         * <p>If the current thread:
         *
         * <ul>
         *
         * <li>has its interrupted status set on entry to this method; or
         *
         * <li>is {@linkplain Thread#interrupt interrupted} while
         * acquiring the read lock,
         *
         * </ul> then {@link InterruptedException} is thrown and the
         * current thread's interrupted status is cleared.
         *
         * <p>If the specified waiting time elapses then the value
         * {@code false} is returned.  If the time is less than or
         * equal to zero, the method will not wait at all.
         *
         * <p>In this implementation, as this method is an explicit
         * interruption point, preference is given to responding to
         * the interrupt over normal or reentrant acquisition of the
         * lock, and over reporting the elapse of the waiting time.
         *
         * @param timeout the time to wait for the read lock
         * @param unit the time unit of the timeout argument
         * @return {@code true} if the read lock was acquired
         * @throws InterruptedException if the current thread is interrupted
         * @throws NullPointerException if the time unit is null
         */
        public boolean tryLock(long timeout, TimeUnit unit)
                throws InterruptedException {
            return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
        }

        /**
         * Attempts to release this lock.
         *
         * <p>If the number of readers is now zero then the lock
         * is made available for write lock attempts.
         */
        public void unlock() {
            sync.releaseShared(1);
        }

        /**
         * Throws {@code UnsupportedOperationException} because
         * {@code ReadLocks} do not support conditions.
         *
         * @throws UnsupportedOperationException always
         */
        public Condition newCondition() {
            throw new UnsupportedOperationException();
        }

        /**
         * Returns a string identifying this lock, as well as its lock state.
         * The state, in brackets, includes the String {@code "Read locks ="}
         * followed by the number of held read locks.
         *
         * @return a string identifying this lock, as well as its lock state
         */
        public String toString() {
            int r = sync.getReadLockCount();
            return super.toString() +
                "[Read locks = " + r + "]";
        }
    }

    /**
     * The lock returned by method {@link ReentrantReadWriteLock#writeLock}.
     */
    public static class WriteLock implements Lock, java.io.Serializable {
        private static final long serialVersionUID = -4992448646407690164L;
        private final Sync sync;

        /**
         * Constructor for use by subclasses
         *
         * @param lock the outer lock object
         * @throws NullPointerException if the lock is null
         */
        protected WriteLock(ReentrantReadWriteLock lock) {
            sync = lock.sync;
        }

        /**
         * Acquires the write lock.
         *
         * <p>Acquires the write lock if neither the read nor write lock
         * are held by another thread
         * and returns immediately, setting the write lock hold count to
         * one.
         *
         * <p>If the current thread already holds the write lock then the
         * hold count is incremented by one and the method returns
         * immediately.
         *
         * <p>If the lock is held by another thread then the current
         * thread becomes disabled for thread scheduling purposes and
         * lies dormant until the write lock has been acquired, at which
         * time the write lock hold count is set to one.
         */
        public void lock() {
            sync.acquire(1);
        }

        /**
         * Acquires the write lock unless the current thread is
         * {@linkplain Thread#interrupt interrupted}.
         *
         * <p>Acquires the write lock if neither the read nor write lock
         * are held by another thread
         * and returns immediately, setting the write lock hold count to
         * one.
         *
         * <p>If the current thread already holds this lock then the
         * hold count is incremented by one and the method returns
         * immediately.
         *
         * <p>If the lock is held by another thread then the current
         * thread becomes disabled for thread scheduling purposes and
         * lies dormant until one of two things happens:
         *
         * <ul>
         *
         * <li>The write lock is acquired by the current thread; or
         *
         * <li>Some other thread {@linkplain Thread#interrupt interrupts}
         * the current thread.
         *
         * </ul>
         *
         * <p>If the write lock is acquired by the current thread then the
         * lock hold count is set to one.
         *
         * <p>If the current thread:
         *
         * <ul>
         *
         * <li>has its interrupted status set on entry to this method;
         * or
         *
         * <li>is {@linkplain Thread#interrupt interrupted} while
         * acquiring the write lock,
         *
         * </ul>
         *
         * then {@link InterruptedException} is thrown and the current
         * thread's interrupted status is cleared.
         *
         * <p>In this implementation, as this method is an explicit
         * interruption point, preference is given to responding to
         * the interrupt over normal or reentrant acquisition of the
         * lock.
         *
         * @throws InterruptedException if the current thread is interrupted
         */
        public void lockInterruptibly() throws InterruptedException {
            sync.acquireInterruptibly(1);
        }

        /**
         * Acquires the write lock only if it is not held by another thread
         * at the time of invocation.
         *
         * <p>Acquires the write lock if neither the read nor write lock
         * are held by another thread
         * and returns immediately with the value {@code true},
         * setting the write lock hold count to one. Even when this lock has
         * been set to use a fair ordering policy, a call to
         * {@code tryLock()} <em>will</em> immediately acquire the
         * lock if it is available, whether or not other threads are
         * currently waiting for the write lock.  This &quot;barging&quot;
         * behavior can be useful in certain circumstances, even
         * though it breaks fairness. If you want to honor the
         * fairness setting for this lock, then use {@link
         * #tryLock(long, TimeUnit) tryLock(0, TimeUnit.SECONDS) }
         * which is almost equivalent (it also detects interruption).
         *
         * <p>If the current thread already holds this lock then the
         * hold count is incremented by one and the method returns
         * {@code true}.
         *
         * <p>If the lock is held by another thread then this method
         * will return immediately with the value {@code false}.
         *
         * @return {@code true} if the lock was free and was acquired
         * by the current thread, or the write lock was already held
         * by the current thread; and {@code false} otherwise.
         */
        public boolean tryLock( ) {
            return sync.tryWriteLock();
        }

        /**
         * Acquires the write lock if it is not held by another thread
         * within the given waiting time and the current thread has
         * not been {@linkplain Thread#interrupt interrupted}.
         *
         * <p>Acquires the write lock if neither the read nor write lock
         * are held by another thread
         * and returns immediately with the value {@code true},
         * setting the write lock hold count to one. If this lock has been
         * set to use a fair ordering policy then an available lock
         * <em>will not</em> be acquired if any other threads are
         * waiting for the write lock. This is in contrast to the {@link
         * #tryLock()} method. If you want a timed {@code tryLock}
         * that does permit barging on a fair lock then combine the
         * timed and un-timed forms together:
         *
         *  <pre> {@code
         * if (lock.tryLock() ||
         *     lock.tryLock(timeout, unit)) {
         *   ...
         * }}</pre>
         *
         * <p>If the current thread already holds this lock then the
         * hold count is incremented by one and the method returns
         * {@code true}.
         *
         * <p>If the lock is held by another thread then the current
```

```
     * thread becomes disabled for thread scheduling purposes and
     * lies dormant until one of three things happens:
     *
     * <ul>
     *
     * <li>The write lock is acquired by the current thread; or
     *
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     *
     * <li>The specified waiting time elapses
     *
     * </ul>
     *
     * <p>If the write lock is acquired then the value {@code true} is
     * returned and the write lock hold count is set to one.
     *
     * <p>If the current thread:
     *
     * <ul>
     *
     * <li>has its interrupted status set on entry to this method;
     * or
     *
     * <li>is {@linkplain Thread#interrupt interrupted} while
     * acquiring the write lock,
     *
     * </ul>
     *
     * then {@link InterruptedException} is thrown and the current
     * thread's interrupted status is cleared.
     *
     * <p>If the specified waiting time elapses then the value
     * {@code false} is returned.  If the time is less than or
     * equal to zero, the method will not wait at all.
     *
     * <p>In this implementation, as this method is an explicit
     * interruption point, preference is given to responding to
     * the interrupt over normal or reentrant acquisition of the
     * lock, and over reporting the elapse of the waiting time.
     *
     * @param timeout the time to wait for the write lock
     * @param unit the time unit of the timeout argument
     *
     * @return {@code true} if the lock was free and was acquired
     * by the current thread, or the write lock was already held by the
     * current thread; and {@code false} if the waiting time
     * elapsed before the lock could be acquired.
     *
     * @throws InterruptedException if the current thread is interrupted
     * @throws NullPointerException if the time unit is null
     */
    public boolean tryLock(long timeout, TimeUnit unit)
```

```java
                throws InterruptedException {
            return sync.tryAcquireNanos(1, unit.toNanos(timeout));
        }

        /**
         * Attempts to release this lock.
         *
         * <p>If the current thread is the holder of this lock then
         * the hold count is decremented. If the hold count is now
         * zero then the lock is released.  If the current thread is
         * not the holder of this lock then {@link
         * IllegalMonitorStateException} is thrown.
         *
         * @throws IllegalMonitorStateException if the current thread does not
         * hold this lock
         */
        public void unlock() {
            sync.release(1);
        }

        /**
         * Returns a {@link Condition} instance for use with this
         * {@link Lock} instance.
         * <p>The returned {@link Condition} instance supports the same
         * usages as do the {@link Object} monitor methods ({@link
         * Object#wait() wait}, {@link Object#notify notify}, and {@link
         * Object#notifyAll notifyAll}) when used with the built-in
         * monitor lock.
         *
         * <ul>
         *
         * <li>If this write lock is not held when any {@link
         * Condition} method is called then an {@link
         * IllegalMonitorStateException} is thrown.  (Read locks are
         * held independently of write locks, so are not checked or
         * affected. However it is essentially always an error to
         * invoke a condition waiting method when the current thread
         * has also acquired read locks, since other threads that
         * could unblock it will not be able to acquire the write
         * lock.)
         *
         * <li>When the condition {@linkplain Condition#await() waiting}
         * methods are called the write lock is released and, before
         * they return, the write lock is reacquired and the lock hold
         * count restored to what it was when the method was called.
         *
         * <li>If a thread is {@linkplain Thread#interrupt interrupted} while
         * waiting then the wait will terminate, an {@link
         * InterruptedException} will be thrown, and the thread's
         * interrupted status will be cleared.
         *
         * <li> Waiting threads are signalled in FIFO order.
         *
         * <li>The ordering of lock reacquisition for threads returning
         * from waiting methods is the same as for threads initially
         * acquiring the lock, which is in the default case not specified,
         * but for <em>fair</em> locks favors those threads that have been
         * waiting the longest.
         *
         * </ul>
         *
         * @return the Condition object
         */
        public Condition newCondition() {
            return sync.newCondition();
        }

        /**
         * Returns a string identifying this lock, as well as its lock
         * state.  The state, in brackets includes either the String
         * {@code "Unlocked"} or the String {@code "Locked by"}
         * followed by the {@linkplain Thread#getName name} of the owning thread.
         *
         * @return a string identifying this lock, as well as its lock state
         */
        public String toString() {
            Thread o = sync.getOwner();
            return super.toString() + ((o == null) ?
                                       "[Unlocked]" :
                                       "[Locked by thread " + o.getName() + "]");
        }

        /**
         * Queries if this write lock is held by the current thread.
         * Identical in effect to {@link
         * ReentrantReadWriteLock#isWriteLockedByCurrentThread}.
         *
         * @return {@code true} if the current thread holds this lock and
         *         {@code false} otherwise
         * @since 1.6
         */
        public boolean isHeldByCurrentThread() {
            return sync.isHeldExclusively();
        }

        /**
         * Queries the number of holds on this write lock by the current
         * thread.  A thread has a hold on a lock for each lock action
         * that is not matched by an unlock action.  Identical in effect
         * to {@link ReentrantReadWriteLock#getWriteHoldCount}.
         *
         * @return the number of holds on this lock by the current thread,
         *         or zero if this lock is not held by the current thread
         * @since 1.6
         */
        public int getHoldCount() {
```

```java
            return sync.getWriteHoldCount();
        }
    }

    // Instrumentation and status

    /**
     * Returns {@code true} if this lock has fairness set true.
     *
     * @return {@code true} if this lock has fairness set true
     */
    public final boolean isFair() {
        return sync instanceof FairSync;
    }

    /**
     * Returns the thread that currently owns the write lock, or
     * {@code null} if not owned. When this method is called by a
     * thread that is not the owner, the return value reflects a
     * best-effort approximation of current lock status. For example,
     * the owner may be momentarily {@code null} even if there are
     * threads trying to acquire the lock but have not yet done so.
     * This method is designed to facilitate construction of
     * subclasses that provide more extensive lock monitoring
     * facilities.
     *
     * @return the owner, or {@code null} if not owned
     */
    protected Thread getOwner() {
        return sync.getOwner();
    }

    /**
     * Queries the number of read locks held for this lock. This
     * method is designed for use in monitoring system state, not for
     * synchronization control.
     * @return the number of read locks held
     */
    public int getReadLockCount() {
        return sync.getReadLockCount();
    }

    /**
     * Queries if the write lock is held by any thread. This method is
     * designed for use in monitoring system state, not for
     * synchronization control.
     *
     * @return {@code true} if any thread holds the write lock and
     *         {@code false} otherwise
     */
    public boolean isWriteLocked() {
        return sync.isWriteLocked();
    }
    /**
     * Queries if the write lock is held by the current thread.
     *
     * @return {@code true} if the current thread holds the write lock and
     *         {@code false} otherwise
     */
    public boolean isWriteLockedByCurrentThread() {
        return sync.isHeldExclusively();
    }

    /**
     * Queries the number of reentrant write holds on this lock by the
     * current thread.  A writer thread has a hold on a lock for
     * each lock action that is not matched by an unlock action.
     *
     * @return the number of holds on the write lock by the current thread,
     *         or zero if the write lock is not held by the current thread
     */
    public int getWriteHoldCount() {
        return sync.getWriteHoldCount();
    }

    /**
     * Queries the number of reentrant read holds on this lock by the
     * current thread.  A reader thread has a hold on a lock for
     * each lock action that is not matched by an unlock action.
     *
     * @return the number of holds on the read lock by the current thread,
     *         or zero if the read lock is not held by the current thread
     * @since 1.6
     */
    public int getReadHoldCount() {
        return sync.getReadHoldCount();
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire the write lock.  Because the actual set of threads may
     * change dynamically while constructing this result, the returned
     * collection is only a best-effort estimate.  The elements of the
     * returned collection are in no particular order.  This method is
     * designed to facilitate construction of subclasses that provide
     * more extensive lock monitoring facilities.
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedWriterThreads() {
        return sync.getExclusiveQueuedThreads();
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire the read lock.  Because the actual set of threads may
     * change dynamically while constructing this result, the returned
     * collection is only a best-effort estimate.  The elements of the
     * returned collection are in no particular order.  This method is
     * designed to facilitate construction of subclasses that provide
     * more extensive lock monitoring facilities.
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedReaderThreads() {
        return sync.getSharedQueuedThreads();
    }

    /**
     * Queries whether any threads are waiting to acquire the read or
     * write lock. Note that because cancellations may occur at any
     * time, a {@code true} return does not guarantee that any other
     * thread will ever acquire a lock.  This method is designed
     * primarily for use in monitoring of the system state.
     *
     * @return {@code true} if there may be other threads waiting to
     *         acquire the lock
     */
    public final boolean hasQueuedThreads() {
        return sync.hasQueuedThreads();
    }

    /**
     * Queries whether the given thread is waiting to acquire either
     * the read or write lock. Note that because cancellations may
     * occur at any time, a {@code true} return does not guarantee
     * that this thread will ever acquire a lock.  This method is
     * designed primarily for use in monitoring of the system state.
     *
     * @param thread the thread
     * @return {@code true} if the given thread is queued waiting for this lock
     * @throws NullPointerException if the thread is null
     */
    public final boolean hasQueuedThread(Thread thread) {
        return sync.isQueued(thread);
    }

    /**
     * Returns an estimate of the number of threads waiting to acquire
     * either the read or write lock.  The value is only an estimate
     * because the number of threads may change dynamically while this
     * method traverses internal data structures.  This method is
     * designed for use in monitoring of the system state, not for
     * synchronization control.
     *
     * @return the estimated number of threads waiting for this lock
     */
    public final int getQueueLength() {
        return sync.getQueueLength();
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire either the read or write lock.  Because the actual set
     * of threads may change dynamically while constructing this
     * result, the returned collection is only a best-effort estimate.
     * The elements of the returned collection are in no particular
     * order.  This method is designed to facilitate construction of
     * subclasses that provide more extensive monitoring facilities.
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedThreads() {
        return sync.getQueuedThreads();
    }

    /**
     * Queries whether any threads are waiting on the given condition
     * associated with the write lock. Note that because timeouts and
     * interrupts may occur at any time, a {@code true} return does
     * not guarantee that a future {@code signal} will awaken any
     * threads.  This method is designed primarily for use in
     * monitoring of the system state.
     *
     * @param condition the condition
     * @return {@code true} if there are any waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public boolean hasWaiters(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return sync.hasWaiters((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns an estimate of the number of threads waiting on the
     * given condition associated with the write lock. Note that because
     * timeouts and interrupts may occur at any time, the estimate
     * serves only as an upper bound on the actual number of waiters.
     * This method is designed for use in monitoring of the system
     * state, not for synchronization control.
     *
     * @param condition the condition
     * @return the estimated number of waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public int getWaitQueueLength(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return
sync.getWaitQueueLength((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns a collection containing those threads that may be
     * waiting on the given condition associated with the write lock.
     * Because the actual set of threads may change dynamically while
     * constructing this result, the returned collection is only a
     * best-effort estimate. The elements of the returned collection
     * are in no particular order.  This method is designed to
     * facilitate construction of subclasses that provide more
     * extensive condition monitoring facilities.
     *
     * @param condition the condition
     * @return the collection of threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    protected Collection<Thread> getWaitingThreads(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return
sync.getWaitingThreads((AbstractQueuedSynchronizer.ConditionObject)condition);
    }

    /**
     * Returns a string identifying this lock, as well as its lock state.
     * The state, in brackets, includes the String {@code "Write locks ="}
     * followed by the number of reentrantly held write locks, and the
     * String {@code "Read locks ="} followed by the number of held
     * read locks.
     *
     * @return a string identifying this lock, as well as its lock state
     */
    public String toString() {
        int c = sync.getCount();
        int w = Sync.exclusiveCount(c);
        int r = Sync.sharedCount(c);

        return super.toString() +
            "[Write locks = " + w + ", Read locks = " + r + "]";
    }

    /**
     * Returns the thread id for the given thread.  We must access
     * this directly rather than via method Thread.getId() because
     * getId() is not final, and has been known to be overridden in
     * ways that do not preserve unique mappings.
     */
    static final long getThreadId(Thread thread) {
        return UNSAFE.getLongVolatile(thread, TID_OFFSET);
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    private static final long TID_OFFSET;
    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> tk = Thread.class;
            TID_OFFSET = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("tid"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }

}
```

## Semaphore

semaphore它可以限制访问公共资源的线程数量,可以用来做流量的限制。它就是一个共享锁,是基于AQS实现的，通过state变量来实现共享。通过调用acquire方法,对state值减去一,当调用release的时候,对state值加一。当state变量小于0的时候，在AQS队列中阻塞等待。

```
package java.util.concurrent;
import java.util.Collection;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;


public class Semaphore implements java.io.Serializable {
    private static final long serialVersionUID = -3222578661600680210L;
    /** All mechanics via AbstractQueuedSynchronizer subclass */
    private final Sync sync;

    /**
     * Synchronization implementation for semaphore.  Uses AQS state
     * to represent permits. Subclassed into fair and nonfair
     * versions.
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 1192457210091910933L;

        Sync(int permits) {
            setState(permits);
        }

        final int getPermits() {
            return getState();
        }

        /**
         * Nonfair获取锁的业务逻辑:
         * 1.获取同步状态值
         * 2.每个线程进来就减去请求的值，此处请求的值是1.然后用可用同步状态值减去请求的值得到同步状态剩余的
值。
         * 3.如果同步状态剩余的值<0代表的是许可证已经用完，则让当前线程要放在AQS队列中阻塞等待；同步状态剩
余的值>=0并且CAS操作成功把可用值改为剩余可用的值，就返回剩下可用的值，表示获取锁成功
         */
        final int nonfairTryAcquireShared(int acquires) {
            for (;;) {
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }

        /**
         * Nonfair释放锁的业务逻辑:
         * 1.获取同步状态值
         * 2.当前同步状态值+释放值得到最新的一个值
         * 3.如果得到最新的状态值小于当前获取到的状态值那么就抛出异常
         * 4.如果CAS操作把当前得到的值更新为最新的值，那么就返回true
         */
        protected final boolean tryReleaseShared(int releases) {
            for (;;) {
                int current = getState();
                int next = current + releases;
                if (next < current) // overflow
                    throw new Error("Maximum permit count exceeded");
                if (compareAndSetState(current, next))
                    return true;
            }
        }

        final void reducePermits(int reductions) {
            for (;;) {
                int current = getState();
                int next = current - reductions;
                if (next > current) // underflow
                    throw new Error("Permit count underflow");
                if (compareAndSetState(current, next))
                return;
            }
        }

        final int drainPermits() {
            for (;;) {
                int current = getState();
                if (current == 0 || compareAndSetState(current, 0))
                    return current;
            }
        }
    }

    /**
     * 可以看到NonfairSync类继承了Sync，而Sync继承了AQS,从这里其实可以看出来semaphore是基于AQS实现的
     */
    static final class NonfairSync extends Sync {
        private static final long serialVersionUID = -2694183684443567898L;

        /**
         * 假设初始化的10个信号量，传递在NonfairSync构造器中,NonfairSync构造器调用了super方法，super
方法会调用NonfairSync的父类，也就是Sync的构造器。
         * 而在Sync的构造器中调用了AQS的setState方法，所以说当new Semaphore(10)时候，实际上是在AQS的
框架中初始化了一个同步状态为10的值。
         */
        NonfairSync(int permits) {
            super(permits);
        }

        protected int tryAcquireShared(int acquires) {
            return nonfairTryAcquireShared(acquires);
        }
    }

    /**
     * Fair version
     */
    static final class FairSync extends Sync {
        private static final long serialVersionUID = 2014338818796000944L;

        FairSync(int permits) {
            super(permits);
        }

        protected int tryAcquireShared(int acquires) {
            for (;;) {
                if (hasQueuedPredecessors())
                    return -1;
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }
    }

    /**
     * 从构造器里面可以看出来semaphore默认实现的是非公平锁
     */
    public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }

    /**
     * Creates a {@code Semaphore} with the given number of
     * permits and the given fairness setting.
     *
     * @param permits the initial number of permits available.
     *        This value may be negative, in which case releases
     *        must occur before any acquires will be granted.
     * @param fair {@code true} if this semaphore will guarantee
     *        first-in first-out granting of permits under contention,
     *        else {@code false}
     */
    public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
    }

    /**
     * 调用了AQS中的共享可中断模式,每次改变的状态值是1
     * 如果线程中断则抛出异常,然后调用了tryAcquireShared方法，此方法在AQS中是一个方法
     * AQS内部用的是模版方法的设计模式，也就是顶层已经设计好了架构,tryAcquireShared方法就是其中的一个。
NonfairSync继承了Sync,而Sync继承了AQS,我们实例化了Semaphore,而Semaphore在构造器中实例化了
NonfairSync，所以AQS调用tryAcquireShared方法的时候，实际上调用的是NonfairSync重写的方法,此处也是java
多态的一个表现。从这里可以看出来Doug Lea是宗师级的别的人物,让AQS和上层业务解耦,AQS只关注底层数据的处理,上层
交个用户来实现。
     */
    public void acquire() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    /**
     * Acquires a permit from this semaphore, blocking until one is
     * available.
     *
     * <p>Acquires a permit, if one is available and returns immediately,
     * reducing the number of available permits by one.
     *
     * <p>If no permit is available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until
     * some other thread invokes the {@link #release} method for this
     * semaphore and the current thread is next to be assigned a permit.
     *
     * <p>If the current thread is {@linkplain Thread#interrupt interrupted}
     * while waiting for a permit then it will continue to wait, but the
```

```java
     * time at which the thread is assigned a permit may change compared to
     * the time it would have received the permit had no interruption
     * occurred.  When the thread does return from this method its interrupt
     * status will be set.
     */
    public void acquireUninterruptibly() {
        sync.acquireShared(1);
    }

    /**
     * Acquires a permit from this semaphore, only if one is available at the
     * time of invocation.
     *
     * <p>Acquires a permit, if one is available and returns immediately,
     * with the value {@code true},
     * reducing the number of available permits by one.
     *
     * <p>If no permit is available then this method will return
     * immediately with the value {@code false}.
     *
     * <p>Even when this semaphore has been set to use a
     * fair ordering policy, a call to {@code tryAcquire()} <em>will</em>
     * immediately acquire a permit if one is available, whether or not
     * other threads are currently waiting.
     * This &quot;barging&quot; behavior can be useful in certain
     * circumstances, even though it breaks fairness. If you want to honor
     * the fairness setting, then use
     * {@link #tryAcquire(long, TimeUnit) tryAcquire(0, TimeUnit.SECONDS) }
     * which is almost equivalent (it also detects interruption).
     *
     * @return {@code true} if a permit was acquired and {@code false}
     *         otherwise
     */
    public boolean tryAcquire() {
        return sync.nonfairTryAcquireShared(1) >= 0;
    }

    /**
     * Acquires a permit from this semaphore, if one becomes available
     * within the given waiting time and the current thread has not
     * been {@linkplain Thread#interrupt interrupted}.
     *
     * <p>Acquires a permit, if one is available and returns immediately,
     * with the value {@code true},
     * reducing the number of available permits by one.
     *
     * <p>If no permit is available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until
     * one of three things happens:
     * <ul>
     * <li>Some other thread invokes the {@link #release} method for this
     * semaphore and the current thread is next to be assigned a permit; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     * <li>The specified waiting time elapses.
     * </ul>
     *
     * <p>If a permit is acquired then the value {@code true} is returned.
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * to acquire a permit,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.  If the time is less than or equal to zero, the method
     * will not wait at all.
     *
     * @param timeout the maximum time to wait for a permit
     * @param unit the time unit of the {@code timeout} argument
     * @return {@code true} if a permit was acquired and {@code false}
     *         if the waiting time elapsed before a permit was acquired
     * @throws InterruptedException if the current thread is interrupted
     */
    public boolean tryAcquire(long timeout, TimeUnit unit)
        throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }

    /**
     * Releases a permit, returning it to the semaphore.
     *
     * <p>Releases a permit, increasing the number of available permits by
     * one.  If any threads are trying to acquire a permit, then one is
     * selected and given the permit that was just released.  That thread
     * is (re)enabled for thread scheduling purposes.
     *
     * <p>There is no requirement that a thread that releases a permit must
     * have acquired that permit by calling {@link #acquire}.
     * Correct usage of a semaphore is established by programming convention
     * in the application.
     */
    public void release() {
        sync.releaseShared(1);
    }

    /**
     * Acquires the given number of permits from this semaphore,
     * blocking until all are available,
     * or the thread is {@linkplain Thread#interrupt interrupted}.
     *
     * <p>Acquires the given number of permits, if they are available,
```

```java
     * and returns immediately, reducing the number of available permits
     * by the given amount.
     *
     * <p>If insufficient permits are available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until
     * one of two things happens:
     * <ul>
     * <li>Some other thread invokes one of the {@link #release() release}
     * methods for this semaphore, the current thread is next to be assigned
     * permits and the number of available permits satisfies this request; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread.
     * </ul>
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * for a permit,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     * Any permits that were to be assigned to this thread are instead
     * assigned to other threads trying to acquire permits, as if
     * permits had been made available by a call to {@link #release()}.
     *
     * @param permits the number of permits to acquire
     * @throws InterruptedException if the current thread is interrupted
     * @throws IllegalArgumentException if {@code permits} is negative
     */
    public void acquire(int permits) throws InterruptedException {
        if (permits < 0) throw new IllegalArgumentException();
        sync.acquireSharedInterruptibly(permits);
    }

    /**
     * Acquires the given number of permits from this semaphore,
     * blocking until all are available.
     *
     * <p>Acquires the given number of permits, if they are available,
     * and returns immediately, reducing the number of available permits
     * by the given amount.
     *
     * <p>If insufficient permits are available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until
     * some other thread invokes one of the {@link #release() release}
     * methods for this semaphore, the current thread is next to be assigned
     * permits and the number of available permits satisfies this request.
     *
     * <p>If the current thread is {@linkplain Thread#interrupt interrupted}
     * while waiting for permits then it will continue to wait and its
     * position in the queue is not affected.  When the thread does return
     * from this method its interrupt status will be set.
     *
     * @param permits the number of permits to acquire
     * @throws IllegalArgumentException if {@code permits} is negative
     */
    public void acquireUninterruptibly(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        sync.acquireShared(permits);
    }

    /**
     * Acquires the given number of permits from this semaphore, only
     * if all are available at the time of invocation.
     *
     * <p>Acquires the given number of permits, if they are available, and
     * returns immediately, with the value {@code true},
     * reducing the number of available permits by the given amount.
     *
     * <p>If insufficient permits are available then this method will return
     * immediately with the value {@code false} and the number of available
     * permits is unchanged.
     *
     * <p>Even when this semaphore has been set to use a fair ordering
     * policy, a call to {@code tryAcquire} <em>will</em>
     * immediately acquire a permit if one is available, whether or
     * not other threads are currently waiting.  This
     * &quot;barging&quot; behavior can be useful in certain
     * circumstances, even though it breaks fairness. If you want to
     * honor the fairness setting, then use {@link #tryAcquire(int,
     * long, TimeUnit) tryAcquire(permits, 0, TimeUnit.SECONDS) }
     * which is almost equivalent (it also detects interruption).
     *
     * @param permits the number of permits to acquire
     * @return {@code true} if the permits were acquired and
     *         {@code false} otherwise
     * @throws IllegalArgumentException if {@code permits} is negative
     */
    public boolean tryAcquire(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        return sync.nonfairTryAcquireShared(permits) >= 0;
    }

    /**
     * Acquires the given number of permits from this semaphore, if all
     * become available within the given waiting time and the current
     * thread has not been {@linkplain Thread#interrupt interrupted}.
     *
     * <p>Acquires the given number of permits, if they are available and
     * returns immediately, with the value {@code true},
     * reducing the number of available permits by the given amount.
     *
     * <p>If insufficient permits are available then
     * the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of three things happens:
```

```
     * <ul>
     * <li>Some other thread invokes one of the {@link #release() release}
     * methods for this semaphore, the current thread is next to be assigned
     * permits and the number of available permits satisfies this request; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     * <li>The specified waiting time elapses.
     * </ul>
     *
     * <p>If the permits are acquired then the value {@code true} is returned.
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * to acquire the permits,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     * Any permits that were to be assigned to this thread, are instead
     * assigned to other threads trying to acquire permits, as if
     * the permits had been made available by a call to {@link #release()}.
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.  If the time is less than or equal to zero, the method
     * will not wait at all.  Any permits that were to be assigned to this
     * thread, are instead assigned to other threads trying to acquire
     * permits, as if the permits had been made available by a call to
     * {@link #release()}.
     *
     * @param permits the number of permits to acquire
     * @param timeout the maximum time to wait for the permits
     * @param unit the time unit of the {@code timeout} argument
     * @return {@code true} if all permits were acquired and {@code false}
     *         if the waiting time elapsed before all permits were acquired
     * @throws InterruptedException if the current thread is interrupted
     * @throws IllegalArgumentException if {@code permits} is negative
     */
    public boolean tryAcquire(int permits, long timeout, TimeUnit unit)
        throws InterruptedException {
        if (permits < 0) throw new IllegalArgumentException();
        return sync.tryAcquireSharedNanos(permits, unit.toNanos(timeout));
    }

    /**
     * Releases the given number of permits, returning them to the semaphore.
     *
     * <p>Releases the given number of permits, increasing the number of
     * available permits by that amount.
     * If any threads are trying to acquire permits, then one
     * is selected and given the permits that were just released.
     * If the number of available permits satisfies that thread's request
     * then that thread is (re)enabled for thread scheduling purposes;
     * otherwise the thread will wait until sufficient permits are available.
     * If there are still permits available
     * after this thread's request has been satisfied, then those permits
     * are assigned in turn to other threads trying to acquire permits.
     *
     * <p>There is no requirement that a thread that releases a permit must
     * have acquired that permit by calling {@link Semaphore#acquire acquire}.
     * Correct usage of a semaphore is established by programming convention
     * in the application.
     *
     * @param permits the number of permits to release
     * @throws IllegalArgumentException if {@code permits} is negative
     */
    public void release(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        sync.releaseShared(permits);
    }

    /**
     * Returns the current number of permits available in this semaphore.
     *
     * <p>This method is typically used for debugging and testing purposes.
     *
     * @return the number of permits available in this semaphore
     */
    public int availablePermits() {
        return sync.getPermits();
    }

    /**
     * Acquires and returns all permits that are immediately available.
     *
     * @return the number of permits acquired
     */
    public int drainPermits() {
        return sync.drainPermits();
    }

    /**
     * Shrinks the number of available permits by the indicated
     * reduction. This method can be useful in subclasses that use
     * semaphores to track resources that become unavailable. This
     * method differs from {@code acquire} in that it does not block
     * waiting for permits to become available.
     *
     * @param reduction the number of permits to remove
     * @throws IllegalArgumentException if {@code reduction} is negative
     */
    protected void reducePermits(int reduction) {
        if (reduction < 0) throw new IllegalArgumentException();
        sync.reducePermits(reduction);
    }
    /**
     * Returns {@code true} if this semaphore has fairness set true.
     *
     * @return {@code true} if this semaphore has fairness set true
     */
    public boolean isFair() {
        return sync instanceof FairSync;
    }

    /**
     * Queries whether any threads are waiting to acquire. Note that
     * because cancellations may occur at any time, a {@code true}
     * return does not guarantee that any other thread will ever
     * acquire.  This method is designed primarily for use in
     * monitoring of the system state.
     *
     * @return {@code true} if there may be other threads waiting to
     *         acquire the lock
     */
    public final boolean hasQueuedThreads() {
        return sync.hasQueuedThreads();
    }

    /**
     * Returns an estimate of the number of threads waiting to acquire.
     * The value is only an estimate because the number of threads may
     * change dynamically while this method traverses internal data
     * structures.  This method is designed for use in monitoring of the
     * system state, not for synchronization control.
     *
     * @return the estimated number of threads waiting for this lock
     */
    public final int getQueueLength() {
        return sync.getQueueLength();
    }

    /**
     * Returns a collection containing threads that may be waiting to acquire.
     * Because the actual set of threads may change dynamically while
     * constructing this result, the returned collection is only a best-effort
     * estimate.  The elements of the returned collection are in no particular
     * order.  This method is designed to facilitate construction of
     * subclasses that provide more extensive monitoring facilities.
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedThreads() {
        return sync.getQueuedThreads();
    }

    /**
     * Returns a string identifying this semaphore, as well as its state.
     * The state, in brackets, includes the String {@code "Permits ="}
     * followed by the number of permits.
     *
     * @return a string identifying this semaphore, as well as its state
     */
    public String toString() {
        return super.toString() + "[Permits = " + sync.getPermits() + "]";
    }
}
```

# 异步执行任务

## FutureTask

FutureTask实现了RunnableFuture接口，即Runnable接口和Future接口。其中Runnable接口对应了FutureTask名字中的Task，代表FutureTask本质上也是表征了一个任务。而Future接口就对应了FutureTask名字中的Future，表示了我们对于这个任务可以执行某些操作，例如，判断任务是否执行完毕，获取任务的执行结果，取消任务的执行等。所以简单来说，FutureTask本质上就是一个Task，我们可以把它当做简单的Runnable对象来使用。但是它又同时实现了Future接口，因此我们可以对它所代表的Task进行额外的控制操作。

```
package java.util.concurrent;
import java.util.concurrent.locks.LockSupport;


public class FutureTask<V> implements RunnableFuture<V> {
    /**
     * state属性是贯穿整个FutureTask的最核心的属性，该属性的值代表了任务在运行过程中的状态，
     * 随着任务的执行，状态将不断地进行转变，从上面的定义中可以看出，总共有7种状态：包括了1个初始态，2个中间态和4个终止态。
     * 注意，这里的执行完毕是指传入的Callable对象的call方法执行完毕，或者抛出了异常。所以这里的COMPLETING的名字显得有点迷惑性，它并不意味着任务正在执行中，而意味着call方法已经执行完毕，正在设置任务执行的结果。
     * 而将一个任务的状态设置成终止态只有三种方法：
     * set
     * setException
     * cancel
     */
    private volatile int state;                // 任务的状态
    private static final int NEW          = 0; // 任务的初始状态
    private static final int COMPLETING   = 1; // 正在设置任务结果
    private static final int NORMAL       = 2; // 任务正常执行完毕
    private static final int EXCEPTIONAL  = 3; // 任务执行过程中发生异常
    private static final int CANCELLED    = 4; // 任务被取消
    private static final int INTERRUPTING = 5; // 正在中断运行任务的线程
    private static final int INTERRUPTED  = 6; // 任务被中断

    /** 要执行的任务本身，即FutureTask中的“Task”部分，为Callable类型，这里之所以用Callable而不用Runnable是因为FutureTask实现了Future接口，需要获取任务的执行结果 */
    private Callable<V> callable;
```

```java
    /** 任务的执行结果或者抛出的异常，为Object类型，也就是说outcome可以是任意类型的对象，所以当我们将正常的执行结果返回给调用者时，需要进行强制类型转换，返回由Callable定义的V类型 */
    private Object outcome; // non-volatile, protected by state reads/writes
    /** 任务的执行者 */
    private volatile Thread runner;
    /** 指向栈顶节点的指针 */
    private volatile WaitNode waiters;

    /**
     * 根据任务的状态，汇报执行结果
     */
    @SuppressWarnings("unchecked")
    private V report(int s) throws ExecutionException {
        // 根据当前state状态，返回正常执行的结果，或者抛出指定的异常。
        Object x = outcome;
        if (s == NORMAL)
            return (V)x;
        if (s >= CANCELLED)
            throw new CancellationException();
        throw new ExecutionException((Throwable)x);
    }

    /**
     * FutureTask共有2个构造函数，这2个构造函数一个是直接传入Callable对象， 一个是传入一个Runnable对象和一个指定的result, 然后通过Executors工具类将它适配成callable对象, 所以这两个构造函数的本质是一样的:
     * 用传入的参数初始化callable成员变量
     * 将FutureTask的状态设为NEW
     */
    public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        this.callable = callable;
        this.state = NEW;       // ensure visibility of callable
    }

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

    /*
     * 判断任务是否被取消
     */
    public boolean isCancelled() {
        return state >= CANCELLED;
    }

    /*
     * 判断任务是否已经执行完毕
     */
    public boolean isDone() {
        return state != NEW;
    }
    /*
     * 取消正在执行的任务
     *
     * FutureTask实现cancel方法的几个规范
     * 首先，对于“任务已经执行完成了或者任务已经被取消过了，则cancel操作一定是失败的(返回false)”这两条，是通过简单的判断state值是否为NEW实现的，因为我们前面说过了，只要state不为NEW，说明任务已经执行完毕了。从代码中可以看出，只要state不为NEW，则直接返回false。
     * 其次，如果state还是NEW状态，则根据mayInterruptIfRunning的值将state的状态由NEW设置成INTERRUPTING或者CANCELLED
     * 总结，ancel方法实际上完成以下两种状态转换之一:
     * 1.NEW -> CANCELLED (对应于mayInterruptIfRunning=false)
     * 2.NEW -> INTERRUPTING -> INTERRUPTED (对应于mayInterruptIfRunning=true)
     * 对于第一条路径，虽说cancel方法最终返回了true，但它只是简单的把state状态设为CANCELLED，并不会中断线程的执行。但是这样带来的后果是，任务即使执行完毕了，也无法设置任务的执行结果，因为前面分析run方法的时候，设置任务结果有一个中间态，而这个中间态的设置，是以当前state状态为NEW为前提的。
     * 对于第二条路径，虽说对正在执行的线程发出了中断信号，但是，响不响应这个中断是由执行任务的线程自己决定的。当线程在执行过程中被中断的话，无论线程有没有响应中断都是拿不到执行结果的，因为无论set还是setException都要求设置前线程的状态为NEW
     */
    public boolean cancel(boolean mayInterruptIfRunning) {
        // 在以下三种情况之一的，cancel操作一定是失败的：
        // 1.任务已经执行完成了
        // 2.任务已经被取消过了
        // 3.任务因为某种原因不能被取消
        // 其它情况下，cancel操作将返回true。值得注意的是，cancel操作返回true并不代表任务真的就是被取消了，这取决于发动cancel状态时，任务所处的状态：
        // 如果发起cancel时任务还没有开始运行，则随后任务就不会被执行；
        // 如果发起cancel时任务已经在运行了，则这时就需要看mayInterruptIfRunning参数了：
        // 1.如果mayInterruptIfRunning 为true, 则当前在执行的任务会被中断
        // 2.如果mayInterruptIfRunning 为false, 则可以允许正在执行的任务继续运行，直到它执行完
        if (!(state == NEW &&
              UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                  mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
            return false;
        try {    // in case call to interrupt throws exception
            if (mayInterruptIfRunning) {
                // 中断当前正在执行任务的线程，最后将state的状态设为INTERRUPTED
                try {
                    Thread t = runner;
                    if (t != null)
                        t.interrupt();
                } finally { // final state
                    UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
                }
            }
        } finally {
            // 中断操作完成后，还需要通过finishCompletion()来唤醒所有在Treiber栈中等待的线程
            finishCompletion();
        }
        return true;
    }
    /**
     * 获取执行结果
     */
    public V get() throws InterruptedException, ExecutionException {
        int s = state;
        // 当任务还没有执行完毕或者正在设置执行结果时，使用awaitDone方法等待任务进入终止态，注意，
awaitDone的返回值是任务的状态，而不是任务的结果。任务进入终止态之后，就根据任务的执行结果来返回计算结果或者抛
出异常。
        if (s <= COMPLETING)
            s = awaitDone(false, 0L);
        return report(s);
    }

    /**
     * @throws CancellationException {@inheritDoc}
     */
    public V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        if (unit == null)
            throw new NullPointerException();
        int s = state;
        if (s <= COMPLETING &&
            (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)
            throw new TimeoutException();
        return report(s);
    }

    /**
     * Protected method invoked when this task transitions to state
     * {@code isDone} (whether normally or via cancellation). The
     * default implementation does nothing.  Subclasses may override
     * this method to invoke completion callbacks or perform
     * bookkeeping. Note that you can query status inside the
     * implementation of this method to determine whether this task
     * has been cancelled.
     */
    protected void done() { }

    /**
     * 任务执行成功，就使用set(result)设置结果
     */
    protected void set(V v) {
        // 开始通过CAS操作将state属性由原来的NEW状态修改为COMPLETING状态，COMPLETING是一个非常短暂的中
间态，表示正在设置执行的结果。
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
            // 状态设置成功后，就把任务执行结果赋值给outcome, 然后直接把state状态设置成NORMAL
            outcome = v;
            UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
            // 调用了finishCompletion()来完成执行结果的设置
            finishCompletion();
        }
    }
```

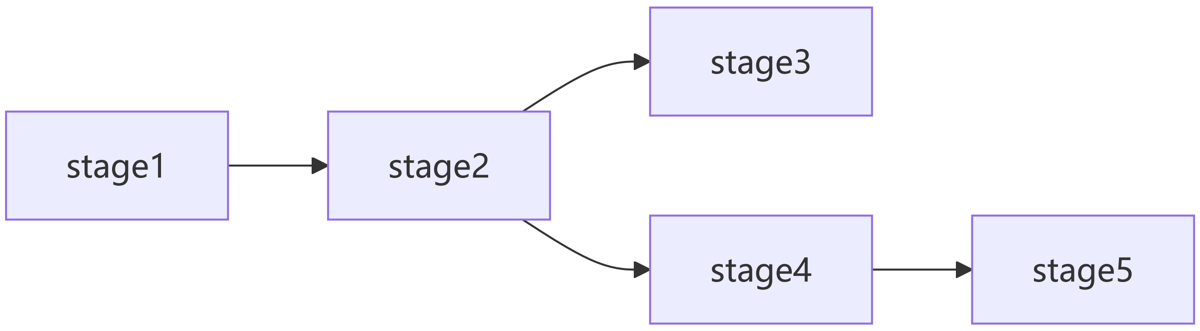
```java
    /**
     * 任务执行失败，用setException(ex)设置抛出的异常
     */
    protected void setException(Throwable t) {
        // 开始通过CAS操作将state属性由原来的NEW状态修改为COMPLETING状态，COMPLETING是一个非常短暂的中间态，表示正在设置执行的结果。
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
            // 状态设置成功后，就把异常对象赋值给outcome, 然后直接把state状态设置成EXCEPTIONAL
            outcome = t;
            // UNSAFE.putOrderedInt不保证可见性，但是保证不会发生重排序，即不会重排序到上一行代码中
            UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state
            // 调用了finishCompletion()来完成执行结果的设置
            finishCompletion();
        }
    }

    /*
     * 由于FutureTask实现了RunnableFuture，RunnableFuture继承了Runnable，而线程池只执行Runnable的run方法，所以run方法，是FutureTask放入线程池后执行的第一个方法。
     *
     */
    public void run() {
        // 状态必须是NEW(新建)状态，能且成功的将其runner修改为当前线程
        // 代表FutureTask保存了当前正在执行他的线程，所以就可以通过runner对象，在cancel中中断线程
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return;
        try {
            Callable<V> c = callable;
            // 查看内部执行的执行体，即用户定义的函数callable是否为空，且当前状态是否已经被改变。即执行前状态必须是NEW
            if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                    // 执行用户定义函数
                    result = c.call();
                    // 执行成功
                    ran = true;
                } catch (Throwable ex) {
                    // 捕捉用户定义的函数执行的异常结果
                    result = null;
                    // 标志执行失败
                    ran = false;
                    // 设置执行失败的结果outcome
                    setException(ex);
                }
                if (ran)
                    // 执行成功，设置正常执行结果outcome
                    set(result);
            }
        } finally {
            // 执行完毕后，不持有Thread的引用，方便垃圾回收(这个FutureTask已经执行完了，可以被回收掉了，如果在这个FutureTask内还持有Thread的引用那就不能进行垃圾回收，所以这里要将将要被垃圾回收掉的FutureTask对象中的runner设置为空)
            runner = null;
            // 判断状态是否被中断，如果是中断处理，那么调用handlePossibleCancellationInterrupt处理中断信息。
            // 注意：如果已经在set方法或者setException方法中将state状态设置成NORMAL或者EXCEPTIONAL了的话就无法再被别的线程取消了，因为在cancel方法中需要通过CAS操作将state的状态从NEW修改到INTERRUPTING或者CANCELLED。
            int s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
    }

    /**
     * Executes the computation without setting its result, and then
     * resets this future to initial state, failing to do so if the
     * computation encounters an exception or is cancelled.  This is
     * designed for use with tasks that intrinsically execute more
     * than once.
     *
     * @return {@code true} if successfully run and reset
     */
    protected boolean runAndReset() {
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return false;
        boolean ran = false;
        int s = state;
        try {
            Callable<V> c = callable;
            if (c != null && s == NEW) {
                try {
                    c.call(); // don't set result
                    ran = true;
                } catch (Throwable ex) {
                    setException(ex);
                }
            }
        } finally {
            // runner must be non-null until state is settled to
            // prevent concurrent calls to run()
            runner = null;
            // state must be re-read after nulling runner to prevent
            // leaked interrupts
            s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
        return ran && s == NEW;
    }
    /**
     * 自旋等待，直到state状态转换成终止态
     */
    private void handlePossibleCancellationInterrupt(int s) {
        // 该方法是一个自旋操作，如果当前的state状态是INTERRUPTING在原地自旋，直到state状态转换成终止态
        if (s == INTERRUPTING)
            while (state == INTERRUPTING)
                Thread.yield(); // wait out pending interrupt
    }

    /**
     * 在FutureTask中，队列的实现是一个单向链表，它表示所有等待任务执行完毕的线程的集合。FutureTask实现了Future接口，可以获取Task的执行结果，那么如果获取结果时，任务还没有执行完毕怎么办呢？那么获取结果的线程就会在一个等待队列中挂起，直到任务执行完毕被唤醒。这一点有点类似于AQS中的sync queue。
     * 并发编程中使用队列通常是将当前线程包装成某种类型的数据结构扔到等待队列中
     *FutureTask中的这个单向链表是当做栈来使用的，确切来说是当做Treiber栈来使用的，(可以简单的把Treiber栈当做是一个线程安全的栈，它使用CAS来完成入栈出栈操作)。为啥要使用一个线程安全的栈呢，因为同一时刻可能有多个线程都在获取任务的执行结果，如果任务还在执行过程中，则这些线程就要被包装成WaitNode扔到Treiber栈的栈顶，即完成入栈操作，这样就有可能出现多个线程同时入栈的情况，因此需要使用CAS操作保证入栈的线程安全，对于出栈的情况也是同理。由于FutureTask中的队列本质上是一个Treiber栈，那么使用这个队列就只需要一个指向栈顶节点的指针就行了，在FutureTask中，就是waiters属性：
     * /** Treiber stack of waiting threads */
     * private volatile WaitNode waiters;
     */
    static final class WaitNode {
        volatile Thread thread;
        volatile WaitNode next;
        WaitNode() { thread = Thread.currentThread(); }
    }

    /**
     * 完成执行结果的设置
     */
    private void finishCompletion() {
        for (WaitNode q; (q = waiters) != null;) {
            // 将waiters属性的值由原值设置为null, 我们知道，waiters属性指向了Treiber栈的栈顶节点，可以说是代表了整个Treiber栈，将该值设为null的目的就是清空整个栈。如果设置不成功，则if语句块不会被执行，又进行下一轮for循环，而下一轮for循环的判断条件又是waiters!=null，并且为下一次CAS操作重新给q赋值 ，由此我们知道，虽然最外层的for循环乍只是为了确保waiters属性被成功设置成null，本质上相当于一个自旋操作。
            // 通过CAS保证可见证，在set和setException方法中使用putOrderedInt更新state的状态时只保证了有序性没有保证可见性，但是在这里由于CAS的底层是lock指令实现的，而lock指令能保证可见性和有序性，因此当执行下面的CAS操作的时候也会将前面putOrderedInt的内容刷到主存中。所以在执行完下面这个操作之后，其他的线程就能看到state的状态了。
            if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
                // 将waiters属性设置成null以后，接下来的死循环才是真正的遍历节点，可以看出，循环内部就是一个普通的遍历链表的操作，Treiber栈里面存放的WaitNode代表了当前等待任务执行结束的线程，这个循环的作用也正是遍历链表中所有等待的线程，并唤醒他们。
                for (;;) {
                    Thread t = q.thread;
                    if (t != null) {
                        q.thread = null;
                        LockSupport.unpark(t);
                    }
                    WaitNode next = q.next;
                    if (next == null)
                        break;
                    q.next = null; // 断开链接帮助GC
                    q = next;
                }
                break;
            }
        }

        // 将Treiber栈中所有挂起的线程都唤醒后，下面就是执行done方法：
        // 这个方法是一个空方法，从注释上看，它是提供给子类覆写的，以实现一些任务执行结束前的额外操作。
        done();
        // done方法之后就是callable属性的清理了（callable = null）。
        callable = null;        // to reduce footprint
    }

    /**
     * FutureTask中会涉及到两类线程，一类是执行任务的线程，它只有一个，FutureTask的run方法就由该线程来执行；一类是获取任务执行结果的线程，它可以有多个，这些线程可以并发执行，每一个线程都是独立的，都可以调用get方法来获取任务的执行结果。如果任务还没有执行完，则这些线程就需要进入Treiber栈中挂起，直到任务执行结束，或者等待的线程自身被中断。
     *
     */
    private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        WaitNode q = null;
        boolean queued = false;
        for (;;) {
            // 首先一开始，先检测当前线程是否被中断了，这是因为get方法是阻塞式的，如果等待的任务还没有执行完，则调用get方法的线程会被扔到Treiber栈中挂起等待，直到任务执行完毕。但是，如果任务迟迟没有执行完毕，则我们也有可能直接中断在Treiber栈中的线程，以停止等待。
            if (Thread.interrupted()) {
                // 当检测到线程被中断后，调用了removeWaiter:
                // removeWaiter的作用是将参数中的node从等待队列（即Treiber栈）中移除。如果此时线程还没有进入Treiber栈，则q=null，那么removeWaiter(q)啥也不干。
                removeWaiter(q);
                // 直接抛出InterruptedException异常
                throw new InterruptedException();
            }

            int s = state;
            // 如果任务已经进入终止态（s > COMPLETING），我们就直接返回任务的状态;
            if (s > COMPLETING) {
                if (q != null)
                    q.thread = null;
                return s;
            }
            // 任务执行完成，正在设置任务结果阶段，在原地自旋等待(用的是Thread.yield)
            // 否则，如果任务正在设置执行结果（s == COMPLETING），我们就让出当前线程的CPU资源继续等待
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            // 否则，就说明任务还没有执行，或者任务正在执行过程中，那么这时，如果q现在还为null, 说明当前线
程还没有进入等待队列，于是新建了一个WaitNode, WaitNode的构造函数之前已经看过了，就是生成了一个记录了当前线
程的节点；
            else if (q == null)
                q = new WaitNode();
            // 如果q不为null，说明代表当前线程的WaitNode已经被创建出来了，则接下来如果queued=false，表
示当前线程还没有入队
            else if (!queued)
                // 通过CAS操作将新建的q节点添加到waiters链表的头节点之前，其实就是Treiber栈的入栈操作
                // 这个CAS操作就是为了保证同一时刻如果有多个线程在同时入栈，则只有一个能够操作成功，也即
Treiber栈的规范。
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                     q.next = waiters, q);
            // 如果timed为true表示，执行带超时时间的get
            else if (timed) {
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                // 获取任务执行结果的线程就会在Treiber栈中被LockSupport.park(this)挂起了
                LockSupport.parkNanos(this, nanos);
            }
            // 以上的条件都不满足(对应任务还没有执行完)，执行不带超时时间的get
            else
                // 获取任务执行结果的线程就会在Treiber栈中被LockSupport.park(this)挂起了
                // 被挂起的线程在下面两种情况被唤醒：
                // 1.任务执行完毕了，在finishCompletion方法中会唤醒所有在Treiber栈中等待的线程
                // 2.等待的线程自身因为被中断等原因而被唤醒
                LockSupport.park(this);
        }
    }

    /**
     * 将当前线程从等待的Treiber栈中移除
     */
    private void removeWaiter(WaitNode node) {
        if (node != null) {
            // 把要出栈的WaitNode的thread属性设置为null
            node.thread = null;
            retry:
            for (;;) {          // restart on removeWaiter race
                // 如果要删除的节点就位于栈顶，由于一开始我们就将该节点的thread属性设为了null，因此，前
面的q.thread != null 和 pred != null都不满足，直接进入到最后一个else if 分支：采用了CAS比较，将栈顶元
素设置成原栈顶节点的下一个节点。
                // 如果要删除的节点不在栈顶，当要移除的节点不在栈顶时，会一直遍历整个链表，直到找到
q.thread == null的节点，找到之后将进入else if (pred != null)，这里因为节点不在栈顶，则其必然是有前驱节
点pred的，这时，只是简单的让前驱节点指向当前节点的下一个节点，从而将目标节点从链表中剔除。
                for (WaitNode pred = null, q = waiters, s; q != null; q = s) {
                    s = q.next;
                    // 当前节点的thread不为空，将pred设置为当前节点继续循环
                    if (q.thread != null)
```

```
                        pred = q;
                    // 当前节点的thread为空，前一个节点不为空
                    else if (pred != null) {
                        // 将前一个节点的next指向当前节点的下一个节点(相当于删除链表中
node.thread==null的节点)
                        pred.next = s;
                        // 注意，后面多加的那个if判断是很有必要的，因为removeWaiter方法并没有加锁，所
以可能有多个线程在同时执行，WaitNode的两个成员变量thread和next都被设置成volatile，这保证了它们的可见性，
如果在这时发现了pred.thread==null，那就意味着它已经被另一个线程标记了，将在另一个线程中被拿出waiters链表，
而当前目标节点的原后继节点现在是接在这个pred节点上的，因此，如果pred已经被其他线程标记为要拿出去的节点，我们
现在这个线程再继续往后遍历就没有什么意义了，所以这时就调到retry处，从头再遍历。如果pred节点没有被其他线程标
记，那我们就接着往下遍历，直到整个链表遍历完。
                        if (pred.thread == null) // check for race
                            continue retry;
                    }
                    else if (!UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                          q, s))
                        // 当CAS操作不成功时，程序会回到retry处重来
                        // 但即使CAS操作成功了，程序依旧会遍历完整个链表，找寻node.thread == null 的
节点，并将它们一并从链表中剔除。
                        continue retry;
                }
                break;
            }
        }
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    // state属性代表了任务的状态
    private static final long stateOffset;
    // runner属性代表了执行FutureTask中的Task的线程
    // 记录执行任务的线程的原因：为了中断或者取消任务做准备的，只有知道了执行任务的线程是谁，才能去中断它。
    private static final long runnerOffset;
    // waiters属性代表了指向栈顶节点的指针
    private static final long waitersOffset;
    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> k = FutureTask.class;
            stateOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("state"));
            runnerOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("runner"));
            waitersOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("waiters"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }

}
```

# 可完成的异步执行任务

## CompletionStage

CompletionStage代表一个异步回调的动作，异步执行的动作，而动作与动作之间可以用stage关联起来，它们之间有触发的先后顺序。


$$\begin{tikzcd}
 & & \text{stage3} & \\
\text{stage1} \arrow[r] & \text{stage2} \arrow[ur, bend left] \arrow[dr, bend right] & & \\
 & & \text{stage4} \arrow[r] & \text{stage5}
\end{tikzcd}$$

stage产生结果(生产者)：Supplier 、Function

stage接受结果(消费者)：Consumer、Function

| 方法类型 | 是否需要输入 | 是否输出 |
|---|---|---|
| applyXXX | 是 | 是 |
| acceptXXX | 是 | 否 |
| runXXX | 否 | 否 |

上图中stage1回调stage2的时候，stage2可以是同步执行也可以是异步执行，同步执行就是当前执行stage2的线程和stage1是同一个线程，异步执行就是当前执行stage2的线程和stage1不是同一个线程。

```java
package java.util.concurrent;
import java.util.function.Supplier;
import java.util.function.Consumer;
import java.util.function.BiConsumer;
import java.util.function.Function;
import java.util.function.BiFunction;
import java.util.concurrent.Executor;

/**
 * CompletionStage代表异步计算中的一个阶段或步骤。该接口定义了多种不同的方式，将CompletionStage实例与其他实例或代码链接在一起，比如完成时调用的方法。CompletionStage的接口一般都返回新的CompletionStage，表示执行完一些逻辑后，生成新的CompletionStage，构成链式的阶段型的操作。
 */
public interface CompletionStage<T> {

    /**
     * 接受上一个阶段的执行结果作为此阶段的入参，返回一个新的CompletionStage，当此阶段正常完成时，将使用此阶段的结果作为提供函数的参数(下一个阶段的类型为supplied的函数的参数)来执行。
     * Apply：用的是Function所以它既有入参，也有返回值。
     *
     * fn:用于计算返回的CompletionStage值的函数
     */
    public <U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn);

    /**
     * 同thenApply，只是执行线程和上一阶段执行线程不是同一个
     *
     */
    public <U> CompletionStage<U> thenApplyAsync
        (Function<? super T,? extends U> fn);

    /**
     * 返回一个新的CompletionStage，使用提供的Executor执行，当此阶段正常完成时，将使用此阶段的结果作为提供函数的参数来执行。
     */
    public <U> CompletionStage<U> thenApplyAsync
        (Function<? super T,? extends U> fn,
         Executor executor);

    /**
     * 返回一个新的CompletionStage，当此阶段正常完成时，将使用此阶段的结果作为提供函数的参数(下一个阶段的类型为supplied的函数的参数)来执行。
     * Accept:用的是Consumer，所以相当于消费者，只有入参没有返回值，所以返回的返回一个新的CompletionStage中的泛型为Void
     *
     *  action:在完成返回的CompletionStage之前要执行的操作
     */
    public CompletionStage<Void> thenAccept(Consumer<? super T> action);

    /**
     * 同thenAccept，只是执行线程和上一阶段执行线程不是同一个
     */
    public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);

    /**
     * 返回一个新的CompletionStage，使用提供的Executor执行，当此阶段正常完成时，将使用此阶段的结果作为提供函数的参数来执行。
     */
    public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action,
                                                 Executor executor);
    /**
     * 返回一个新的CompletionStage，在返回的CompletionStage之前要执行action
     * Run:不接受入参，也没有返回值
     */
    public CompletionStage<Void> thenRun(Runnable action);

    /**
     * 同thenRun，新线程执行
     */
    public CompletionStage<Void> thenRunAsync(Runnable action);

    /**
     * 返回一个新的CompletionStage，在返回的CompletionStage之前使用提供的Executor执行action
     */
    public CompletionStage<Void> thenRunAsync(Runnable action,
                                               Executor executor);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage both complete normally, is executed with the two
     * results as arguments to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param <U> the type of the other CompletionStage's result
     * @param <V> the function's return type
     * @return the new CompletionStage
     */
    public <U,V> CompletionStage<V> thenCombine
        (CompletionStage<? extends U> other,
         BiFunction<? super T,? super U,? extends V> fn);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, is executed using this stage's
     * default asynchronous execution facility, with the two results
     * as arguments to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param <U> the type of the other CompletionStage's result
     * @param <V> the function's return type
     * @return the new CompletionStage
     */
    public <U,V> CompletionStage<V> thenCombineAsync
        (CompletionStage<? extends U> other,
         BiFunction<? super T,? super U,? extends V> fn);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, is executed using the supplied
     * executor, with the two results as arguments to the supplied
     * function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
```

```java
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @param <U> the type of the other CompletionStage's result
     * @param <V> the function's return type
     * @return the new CompletionStage
     */
    public <U,V> CompletionStage<V> thenCombineAsync
        (CompletionStage<? extends U> other,
         BiFunction<? super T,? super U,? extends V> fn,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage both complete normally, is executed with the two
     * results as arguments to the supplied action.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param <U> the type of the other CompletionStage's result
     * @return the new CompletionStage
     */
    public <U> CompletionStage<Void> thenAcceptBoth
        (CompletionStage<? extends U> other,
         BiConsumer<? super T, ? super U> action);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, is executed using this stage's
     * default asynchronous execution facility, with the two results
     * as arguments to the supplied action.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param <U> the type of the other CompletionStage's result
     * @return the new CompletionStage
     */
    public <U> CompletionStage<Void> thenAcceptBothAsync
        (CompletionStage<? extends U> other,
         BiConsumer<? super T, ? super U> action);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, is executed using the supplied
     * executor, with the two results as arguments to the supplied
     * function.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @param <U> the type of the other CompletionStage's result
     * @return the new CompletionStage
     */
    public <U> CompletionStage<Void> thenAcceptBothAsync
        (CompletionStage<? extends U> other,
         BiConsumer<? super T, ? super U> action,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage both complete normally, executes the given action.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterBoth(CompletionStage<?> other,
                                              Runnable action);
    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, executes the given action using
     * this stage's default asynchronous execution facility.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,
                                                   Runnable action);

    /**
     * Returns a new CompletionStage that, when this and the other
     * given stage complete normally, executes the given action using
     * the supplied executor.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,
                                                   Runnable action,
                                                   Executor executor);
    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed with the
     * corresponding result as argument to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> applyToEither
        (CompletionStage<? extends T> other,
         Function<? super T, U> fn);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed using this
     * stage's default asynchronous execution facility, with the
     * corresponding result as argument to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> applyToEitherAsync
        (CompletionStage<? extends T> other,
         Function<? super T, U> fn);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed using the
     * supplied executor, with the corresponding result as argument to
     * the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param fn the function to use to compute the value of
     * the returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> applyToEitherAsync
        (CompletionStage<? extends T> other,
         Function<? super T, U> fn,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed with the
     * corresponding result as argument to the supplied action.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> acceptEither
        (CompletionStage<? extends T> other,
         Consumer<? super T> action);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed using this
     * stage's default asynchronous execution facility, with the
     * corresponding result as argument to the supplied action.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> acceptEitherAsync
        (CompletionStage<? extends T> other,
         Consumer<? super T> action);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, is executed using the
     * supplied executor, with the corresponding result as argument to
     * the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @return the new CompletionStage
     */
    public CompletionStage<Void> acceptEitherAsync
        (CompletionStage<? extends T> other,
         Consumer<? super T> action,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, executes the given action.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterEither(CompletionStage<?> other,
                                                Runnable action);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, executes the given action
     * using this stage's default asynchronous execution facility.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterEitherAsync
        (CompletionStage<?> other,
         Runnable action);

    /**
     * Returns a new CompletionStage that, when either this or the
     * other given stage complete normally, executes the given action
     * using the supplied executor.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param other the other CompletionStage
     * @param action the action to perform before completing the
     * returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @return the new CompletionStage
     */
    public CompletionStage<Void> runAfterEitherAsync
        (CompletionStage<?> other,
         Runnable action,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * normally, is executed with this stage as the argument
     * to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param fn the function returning a new CompletionStage
     * @param <U> the type of the returned CompletionStage's result
     * @return the CompletionStage
     */
    public <U> CompletionStage<U> thenCompose
        (Function<? super T, ? extends CompletionStage<U>> fn);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * normally, is executed using this stage's default asynchronous
     * execution facility, with this stage as the argument to the
     * supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param fn the function returning a new CompletionStage
     * @param <U> the type of the returned CompletionStage's result
     * @return the CompletionStage
     */
    public <U> CompletionStage<U> thenComposeAsync
        (Function<? super T, ? extends CompletionStage<U>> fn);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * normally, is executed using the supplied Executor, with this
     * stage's result as the argument to the supplied function.
     *
     * See the {@link CompletionStage} documentation for rules
     * covering exceptional completion.
     *
     * @param fn the function returning a new CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @param <U> the type of the returned CompletionStage's result
     * @return the CompletionStage
     */
    public <U> CompletionStage<U> thenComposeAsync
        (Function<? super T, ? extends CompletionStage<U>> fn,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * exceptionally, is executed with this stage's exception as the
     * argument to the supplied function.  Otherwise, if this stage
     * completes normally, then the returned stage also completes
     * normally with the same value.
     *
     * @param fn the function to use to compute the value of the
     * returned CompletionStage if this CompletionStage completed
     * exceptionally
     * @return the new CompletionStage
     */
    public CompletionStage<T> exceptionally
        (Function<Throwable, ? extends T> fn);

    /**
     * Returns a new CompletionStage with the same result or exception as
     * this stage, that executes the given action when this stage completes.
     *
     * <p>When this stage is complete, the given action is invoked with the
     * result (or {@code null} if none) and the exception (or {@code null}
     * if none) of this stage as arguments.  The returned stage is completed
     * when the action returns.  If the supplied action itself encounters an
     * exception, then the returned stage exceptionally completes with this
     * exception unless this stage also completed exceptionally.
     *
     * @param action the action to perform
     * @return the new CompletionStage
     */
    public CompletionStage<T> whenComplete
        (BiConsumer<? super T, ? super Throwable> action);

    /**
     * Returns a new CompletionStage with the same result or exception as
     * this stage, that executes the given action using this stage's
     * default asynchronous execution facility when this stage completes.
     *
     * <p>When this stage is complete, the given action is invoked with the
     * result (or {@code null} if none) and the exception (or {@code null}
     * if none) of this stage as arguments.  The returned stage is completed
     * when the action returns.  If the supplied action itself encounters an
     * exception, then the returned stage exceptionally completes with this
     * exception unless this stage also completed exceptionally.
     *
     * @param action the action to perform
     * @return the new CompletionStage
     */
    public CompletionStage<T> whenCompleteAsync
```

```
        (BiConsumer<? super T, ? super Throwable> action);

    /**
     * Returns a new CompletionStage with the same result or exception as
     * this stage, that executes the given action using the supplied
     * Executor when this stage completes.
     *
     * <p>When this stage is complete, the given action is invoked with the
     * result (or {@code null} if none) and the exception (or {@code null}
     * if none) of this stage as arguments.  The returned stage is completed
     * when the action returns.  If the supplied action itself encounters an
     * exception, then the returned stage exceptionally completes with this
     * exception unless this stage also completed exceptionally.
     *
     * @param action the action to perform
     * @param executor the executor to use for asynchronous execution
     * @return the new CompletionStage
     */
    public CompletionStage<T> whenCompleteAsync
        (BiConsumer<? super T, ? super Throwable> action,
         Executor executor);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * either normally or exceptionally, is executed with this stage's
     * result and exception as arguments to the supplied function.
     *
     * <p>When this stage is complete, the given function is invoked
     * with the result (or {@code null} if none) and the exception (or
     * {@code null} if none) of this stage as arguments, and the
     * function's result is used to complete the returned stage.
     *
     * @param fn the function to use to compute the value of the
     * returned CompletionStage
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> handle
        (BiFunction<? super T, Throwable, ? extends U> fn);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * either normally or exceptionally, is executed using this stage's
     * default asynchronous execution facility, with this stage's
     * result and exception as arguments to the supplied function.
     *
     * <p>When this stage is complete, the given function is invoked
     * with the result (or {@code null} if none) and the exception (or
     * {@code null} if none) of this stage as arguments, and the
     * function's result is used to complete the returned stage.
     *
     * @param fn the function to use to compute the value of the
     * returned CompletionStage
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> handleAsync
        (BiFunction<? super T, Throwable, ? extends U> fn);

    /**
     * Returns a new CompletionStage that, when this stage completes
     * either normally or exceptionally, is executed using the
     * supplied executor, with this stage's result and exception as
     * arguments to the supplied function.
     *
     * <p>When this stage is complete, the given function is invoked
     * with the result (or {@code null} if none) and the exception (or
     * {@code null} if none) of this stage as arguments, and the
     * function's result is used to complete the returned stage.
     *
     * @param fn the function to use to compute the value of the
     * returned CompletionStage
     * @param executor the executor to use for asynchronous execution
     * @param <U> the function's return type
     * @return the new CompletionStage
     */
    public <U> CompletionStage<U> handleAsync
        (BiFunction<? super T, Throwable, ? extends U> fn,
         Executor executor);

    /**
     * Returns a {@link CompletableFuture} maintaining the same
     * completion properties as this stage. If this stage is already a
     * CompletableFuture, this method may return this stage itself.
     * Otherwise, invocation of this method may be equivalent in
     * effect to {@code thenApply(x -> x)}, but returning an instance
     * of type {@code CompletableFuture}. A CompletionStage
     * implementation that does not choose to interoperate with others
     * may throw {@code UnsupportedOperationException}.
     *
     * @return the CompletableFuture
     * @throws UnsupportedOperationException if this implementation
     * does not interoperate with CompletableFuture
     */
    public CompletableFuture<T> toCompletableFuture();

}
```

## CompletableFuture

CompletableFuture 可完成的异步执行的任务，主要解决异步回调处理的复杂性

```
package java.util.concurrent;
import java.util.function.Supplier;
import java.util.function.Consumer;
```

```java
import java.util.function.BiConsumer;
import java.util.function.Function;
import java.util.function.BiFunction;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.Executor;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.CancellationException;
import java.util.concurrent.CompletionException;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.locks.LockSupport;

/**
 * CompletableFuture<T>类，它实现了新的CompletionStage<T>接口，并对Future<T>进行了扩展。
 */
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {

    // 存放执行结果，正常结果或者抛出的异常都要存放，所以是Object。任务执行完毕后，result会变成非null。
    volatile Object result;       // Either the result or boxed AltResult
    // stack是一个链栈，存放与this对象直接关联的Completion对象。Completion对象是用来驱动某一个
CompletableFuture对象，所谓的驱动，就是使得这个CompletableFuture对象的result成员变为非null。
    volatile Completion stack;    // Top of Treiber stack of dependent actions

    final boolean internalComplete(Object r) { // CAS from null to r
        return UNSAFE.compareAndSwapObject(this, RESULT, null, r);
    }

    final boolean casStack(Completion cmp, Completion val) {
        return UNSAFE.compareAndSwapObject(this, STACK, cmp, val);
    }

    /** Returns true if successfully pushed c onto stack. */
    final boolean tryPushStack(Completion c) {
        Completion h = stack;
        lazySetNext(c, h);
        return UNSAFE.compareAndSwapObject(this, STACK, h, c);
    }

    /** Unconditionally pushes c onto stack, retrying if necessary. */
    final void pushStack(Completion c) {
        do {} while (!tryPushStack(c));
    }

    /* ------------- Encoding and decoding outcomes -------------- */
    // 任务执行完毕后，result会变成非null。但如果执行结果就是null该怎么办。所以用这个对象来包装一下null。
    static final class AltResult { // See above
        final Throwable ex;        // null only for NIL
        AltResult(Throwable x) { this.ex = x; }
    }
    /** The encoding of the null value. */
    static final AltResult NIL = new AltResult(null);

    /** Completes with the null value, unless already completed. */
    final boolean completeNull() {
        return UNSAFE.compareAndSwapObject(this, RESULT, null,
                                           NIL);
    }

    /** Returns the encoding of the given non-exceptional value. */
    final Object encodeValue(T t) {
        return (t == null) ? NIL : t;
    }

    /** Completes with a non-exceptional result, unless already completed. */
    final boolean completeValue(T t) {
        return UNSAFE.compareAndSwapObject(this, RESULT, null,
                                           (t == null) ? NIL : t);
    }

    /**
     * Returns the encoding of the given (non-null) exception as a
     * wrapped CompletionException unless it is one already.
     */
    static AltResult encodeThrowable(Throwable x) {
        return new AltResult((x instanceof CompletionException) ? x :
                             new CompletionException(x));
    }

    /** Completes with an exceptional result, unless already completed. */
    final boolean completeThrowable(Throwable x) {
        return UNSAFE.compareAndSwapObject(this, RESULT, null,
                                           encodeThrowable(x));
    }

    /**
     * Returns the encoding of the given (non-null) exception as a
     * wrapped CompletionException unless it is one already.  May
     * return the given Object r (which must have been the result of a
     * source future) if it is equivalent, i.e. if this is a simple
     * relay of an existing CompletionException.
     */
    static Object encodeThrowable(Throwable x, Object r) {
        if (!(x instanceof CompletionException))
            x = new CompletionException(x);
        else if (r instanceof AltResult && x == ((AltResult)r).ex)
            return r;
        return new AltResult(x);
    }

    /**
     * Completes with the given (non-null) exceptional result as a
     * wrapped CompletionException unless it is one already, unless
     * already completed.  May complete with the given Object r
     * (which must have been the result of a source future) if it is
     * equivalent, i.e. if this is a simple propagation of an
     * existing CompletionException.
     */
    final boolean completeThrowable(Throwable x, Object r) {
        return UNSAFE.compareAndSwapObject(this, RESULT, null,
                                           encodeThrowable(x, r));
    }

    /**
     * Returns the encoding of the given arguments: if the exception
     * is non-null, encodes as AltResult.  Otherwise uses the given
     * value, boxed as NIL if null.
     */
    Object encodeOutcome(T t, Throwable x) {
        return (x == null) ? (t == null) ? NIL : t : encodeThrowable(x);
    }

    /**
     * Returns the encoding of a copied outcome; if exceptional,
     * rewraps as a CompletionException, else returns argument.
     */
    static Object encodeRelay(Object r) {
        Throwable x;
        return (((r instanceof AltResult) &&
                 (x = ((AltResult)r).ex) != null &&
                 !(x instanceof CompletionException)) ?
                new AltResult(new CompletionException(x)) : r);
    }

    /**
     * Completes with r or a copy of r, unless already completed.
     * If exceptional, r is first coerced to a CompletionException.
     */
    final boolean completeRelay(Object r) {
        return UNSAFE.compareAndSwapObject(this, RESULT, null,
                                           encodeRelay(r));
    }

    /**
     * Reports result using Future.get conventions.
     */
    private static <T> T reportGet(Object r)
        throws InterruptedException, ExecutionException {
        if (r == null) // by convention below, null means interrupted
            throw new InterruptedException();
        if (r instanceof AltResult) {
            Throwable x, cause;
            if ((x = ((AltResult)r).ex) == null)
                return null;
            if (x instanceof CancellationException)
```

```
                throw (CancellationException)x;
            if ((x instanceof CompletionException) &&
                (cause = x.getCause()) != null)
                x = cause;
            throw new ExecutionException(x);
        }
        @SuppressWarnings("unchecked") T t = (T) r;
        return t;
    }

    /**
     * Decodes outcome to return result or throw unchecked exception.
     */
    private static <T> T reportJoin(Object r) {
        if (r instanceof AltResult) {
            Throwable x;
            if ((x = ((AltResult)r).ex) == null)
                return null;
            if (x instanceof CancellationException)
                throw (CancellationException)x;
            if (x instanceof CompletionException)
                throw (CompletionException)x;
            throw new CompletionException(x);
        }
        @SuppressWarnings("unchecked") T t = (T) r;
        return t;
    }

    /* ------------- Async task preliminaries -------------- */

    /**
     * A marker interface identifying asynchronous tasks produced by
     * {@code async} methods. This may be useful for monitoring,
     * debugging, and tracking asynchronous activities.
     *
     * @since 1.8
     */
    public static interface AsynchronousCompletionTask {
    }

    private static final boolean useCommonPool =
        (ForkJoinPool.getCommonPoolParallelism() > 1);

    /**
     * Default executor -- ForkJoinPool.commonPool() unless it cannot
     * support parallelism.
     */
    private static final Executor asyncPool = useCommonPool ?
        ForkJoinPool.commonPool() : new ThreadPerTaskExecutor();

    /** Fallback if ForkJoinPool.commonPool() cannot support parallelism */
    static final class ThreadPerTaskExecutor implements Executor {
        public void execute(Runnable r) { new Thread(r).start(); }
    }

    /**
     * Null-checks user executor argument, and translates uses of
     * commonPool to asyncPool in case parallelism disabled.
     */
    static Executor screenExecutor(Executor e) {
        if (!useCommonPool && e == ForkJoinPool.commonPool())
            return asyncPool;
        if (e == null) throw new NullPointerException();
        return e;
    }

    // Modes for Completion.tryFire. Signedness matters.
    static final int SYNC   =  0;
    static final int ASYNC  =  1;
    static final int NESTED = -1;

    /* ------------- Base Completion classes and operations -------------- */

    // Completion继承了ForkJoinTask<Void>，但也仅仅是为了套上ForkJoinTask的壳子，因为
CompletableFuture默认的线程池是ForkJoinPool.commonPool()。但它也实现了Runnable，这使得它也能被一个普
通线程正常执行。Completion有很多继承的子类，它们分别实现了tryFire方法。
    @SuppressWarnings("serial")
    abstract static class Completion extends ForkJoinTask<Void>
        implements Runnable, AsynchronousCompletionTask {
        volatile Completion next;      // Treiber stack link

        /**
         * Performs completion action if triggered, returning a
         * dependent that may need propagation, if one exists.
         *
         * @param mode SYNC, ASYNC, or NESTED
         */
        abstract CompletableFuture<?> tryFire(int mode);

        /** Returns true if possibly still triggerable. Used by cleanStack. */
        abstract boolean isLive();

        public final void run()                { tryFire(ASYNC); }
        public final boolean exec()            { tryFire(ASYNC); return true; }
        public final Void getRawResult()       { return null; }
        public final void setRawResult(Void v) {}
    }

    static void lazySetNext(Completion c, Completion next) {
        UNSAFE.putOrderedObject(c, NEXT, next);
    }

    /**
     * 在当前stage完成后，弹出并尝试触发所有可访问的依赖项
     * 对tryFire的调用可能是这种过程：postComplete -> tryFire -> postFire -> postComplete，这样是会造成递归调用的。但是现在，该函数通过tryFire(NESTED)的特殊处理，并且在遇到第二层次以下的后续任务时，先把后续任务放到树形结构中的第二层次，再来执行，从而解决了递归调用的问题。tryFire(NESTED)的特殊处理是指，当参数为NESTED时，只处理这个Completion对象的当前stage，不处理当前stage的后续stage。
     */
    final void postComplete() {
        // A--(stack)-->B---->C
        // f初始指向当前stage对象，注意，遍历过程中，f可能不再指向this，而是树形结构下面层次的节点
        CompletableFuture<?> f = this; Completion h;
        //1. 如果当前f的stack不为null，那么短路后面
        //2. 如果当前f的stack为null，且f是this（f != this不成立），说明整个树形结构已经被遍历完毕，退出循环
        //3. 如果当前f的stack为null，且f不是this，那么让f恢复为this，再查看this的stack是否为null(恢复f为this反正在，将f更新为树形结构中的非root节点后，并且将f的后续任务全部压入root(this对象)的栈后，此时f.stack肯定为空了，后续会继续判断f != this，也为真，如是就将f恢复为this，再判断f.stack是不是为空)
        while ((h = f.stack) != null ||
               (f != this && (h = (f = this).stack) != null)) {
            CompletableFuture<?> d; Completion t;
            // 使得f的栈顶指针下移
            if (f.casStack(h, t = h.next)) {
                // 如果h是有后继的
                if (t != null) {
                    // 如果f不是树形结构的root，就把f的后续任务压入root(this对象)的栈，直到全部压入
                    if (f != this) {
                        pushStack(h);
                        continue;
                    }
                    // 如果h是有后继的，需要断开h的next指针
                    h.next = null;    // detach
                }
                //Completion执行tryFire(NESTED)后，有两种情况：
                //1. 以上图为例，可能会返回Completion对象的当前stage。因为当前线程在tryFire中执行了h的当前stage，所以f会更新为树形结构中的非root的节点。(出现这种情况只有可能在supplyAsync后调用thenApply方法，因为在thenApply方法调用uniApplyStage方法的时候，线程池传入的是null，所以UniApply对象的executor为null。这样在调用当前stage的tryFire方法的时候，当前stage的线程池为null，所以uniApply的claim返回true，这样会在当前线程执行h的当前stage)
                //2. 也有可能会返回null。因为当前线程在tryFire中没有执行h的当前stage，这说明h的当前stage的stack不需要当前线程来处理了，所以要把f恢复为this
                f = (d = h.tryFire(NESTED)) == null ? this : d;
            }
        }
    }

    /** 一个Completion对象的当前stage如果已经完成，那么它的dep成员肯定为null.(因为在UniApply的tryFire方法中，在当前stage完成后会将dep赋为null的处理) */
    final void cleanStack() {
        // 遍历过程中，可能的结构为p -> q -> s,  q为当前遍历节点
        for (Completion p = null, q = stack; q != null;) {
            Completion s = q.next;
            if (q.isLive()) { // 如果q是有效节点，更新q，用p保存旧q
                p = q;
                q = s;
            }
            else if (p == null) { // 如果q是栈顶，那么移动栈顶
                casStack(q, s);
                q = stack;
            }
            else { // 如果q不是栈顶，那么使得p -> q -> s变成p -> s，达到移除q的目的
                p.next = s;
                // 但这种移除成功的前提是q是一个有效节点
                if (p.isLive()) // 如果是有效的移除
                    q = s;
                else { // 如果不是有效的移除
                    p = null;  // 需要从新开始整个循环
                    q = stack;
                }
            }
        }
    }

    /* ------------- One-input Completions -------------- */

    /** A Completion with a source, dependent, and executor. */
    @SuppressWarnings("serial")
    abstract static class UniCompletion<T,V> extends Completion {
        Executor executor;                 // executor to use (null if none)
        CompletableFuture<V> dep;          // the dependent to complete
        CompletableFuture<T> src;          // source for action

        UniCompletion(Executor executor, CompletableFuture<V> dep,
                      CompletableFuture<T> src) {
            this.executor = executor; this.dep = dep; this.src = src;
        }

        /**
         * Returns true if action can be run. Call only when known to
         * be triggerable. Uses FJ tag bit to ensure that only one
         * thread claims ownership.  If async, starts as task -- a
         * later call to tryFire will run action.
         */
        final boolean claim() {
            Executor e = executor;
            if (compareAndSetForkJoinTaskTag((short)0, (short)1)) {
                if (e == null)
                    return true;
                executor = null; // disable
                e.execute(this);
            }
            return false;
        }

        final boolean isLive() { return dep != null; }
    }

    /** Pushes the given completion (if it exists) unless done. */
    final void push(UniCompletion<?,?> c) {
        if (c != null) {
            while (result == null && !tryPushStack(c))
                lazySetNext(c, null); // clear on failure
        }
    }

    /**
     * UniApply#tryFire成功执行了这个UniApply对象的当前stage后，将会调用当前stage的postFire。
     */
    final CompletableFuture<T> postFire(CompletableFuture<?> a, int mode) {
        // 前一个stage的后续任务还没做完(stack!=null说明前一个stage的后续任务还没有完成，因为在stage完成后的postComplete处理中会遍历stack中的每一个Completion，然后调用其tryFire方法，在遍历完成后会将stack赋为空)
        if (a != null && a.stack != null) {
            //1. mode为NESTED。说明就是postComplete调用过来的，那么只清理一下栈中无效节点即可。
            //2. mode为SYNC或ASYNC，但前一个stage还没执行完。不知道何时发生，因为调用postFire的前提就是前一个stage已经执行完
            if (mode < 0 || a.result == null)
                a.cleanStack();
            ////3. mode为SYNC或ASYNC，但前一个stage已经执行完了。特殊时序可能发生的，那么帮忙完成前一个stage的的后续任务
            else
                a.postComplete();
        }
        // 当前stage的后续任务还没做完
        if (result != null && stack != null) {
            // mode为NESTED。说明就是postComplete调用过来的.
            if (mode < 0)
                return this;
            // mode为SYNC或ASYNC，那么调用postComplete
            else
                postComplete();
        }
        return null;
    }

    // src代表前一个stage, dep代表当前stage。UniApply对象将两个stage组合在一起了
    @SuppressWarnings("serial")
    static final class UniApply<T,V> extends UniCompletion<T,V> {
        Function<? super T,? extends V> fn;
        UniApply(Executor executor, CompletableFuture<V> dep,
                 CompletableFuture<T> src,
                 Function<? super T,? extends V> fn) {
            super(executor, dep, src); this.fn = fn;
        }
        // 在tryFire函数中，如果d.uniApply(a = src, fn, mode > 0 ? null : this)返回了true，说明当前线程执行了当前stage的函数式接口（这说明没有提供线程池，当前线程同步执行了stage），自然接下来会去处理后续stage（d.postFire(a, mode)）。
        // 在提供了线程池的情况下，且前一个stage没有抛出异常正常执行完的情况下，tryFire函数中的d.uniApply(a = src, fn, mode > 0 ? null : this)必然会返回false，因为它把uniApply对象提交给了线程池来执行。当前线程将不会去处理后续stage了（d.postFire(a, mode)）。
        // 提交给线程池后，因为uniApply对象也是一个Runnable对象，它的run函数为:
        // public final void run()                { tryFire(ASYNC); }
        // 线程池将提供一个线程来执行tryFire(ASYNC)，之后d.uniApply(a = src, fn, mode > 0 ? null
: this)将会直接执行当前stage的函数式接口，返回true后，再去处理后续stage（d.postFire(a, mode)）。
        // tryFire在下面几处被调用：
        // uniApplyStage中的同步调用，c.tryFire(SYNC)
        // 执行前一个stage的线程，在run的d.postComplete()中，会调用tryFire(NESTED)
        // 上面两处，tryFire的this对象都是当前stage。并且，这说明tryFire可能会有多线程的竞争问题
        // 比如 当前线程入栈后，正要执行c.tryFire(SYNC)，此时执行前一个stage的线程刚完事正要触发后续
stage（run的d.postComplete()中）。
        // 因此，会在CompletableFuture#uniApply#claim中使用CAS保证函数式接口只被提交一次给Executor
        final CompletableFuture<V> tryFire(int mode) {
            CompletableFuture<V> d; CompletableFuture<T> a;
            // 1. 如果dep为null，说明当前stage已经被执行过了()
            // 2. 如果uniApply返回false，说明当前线程无法执行当前stage。返回false有可能是因为：
            //     1. 前一个stage没执行完呢
            //     2. 前一个stage执行完了，但当前stage已经被别的线程执行了。如果提供了线程池，那么肯定属
于被别的线程执行了。
            if ((d = dep) == null ||
                !d.uniApply(a = src, fn, mode > 0 ? null : this))
                return null;
            // 执行到这里，说明dep不为null，而且uniApply返回true，说明当前线程执行了当前stage
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    /**
     * uniApply函数返回后，然后回到tryFire函数，紧接着马上执行dep = null（这代表当前stage已经执行完
毕）。这样可以使得某种特殊时序下，执行前一个stage的线程不会通过tryFire函数中的(d = dep) == null的检查，
进而直接返回null不去执行d.postFire(a, mode);。总之，这是为了避免对同一个CompletableFuture对象调用它的
成员函数postComplete。但这种防止线程竞争的手段不是完全有效的，因为“紧接着马上执行dep = null”不是原子性，两
个线程的执行顺序也有可能穿插执行，有可能当前线程还没来得及“执行dep = null”，执行前一个stage的线程就开始执行
tryFire函数了。
     * 所以claim函数是用来保护函数式接口只被执行一次的：
     * 如果该Completion包含有Executor，那么claim函数每次都会返回false。并且通过CAS保证函数式接口只被提
交一次给Executor(即最终执行postComplete的只有一个Executor)
     * 如果该Completion没有Executor，那么此函数第一次返回true，之后每次返回false。当Completion返回
true的时候说明当前stage的函数式接口被当前线程执行。因此应该有当前线程来执行CompletableFuture的
postComplete方法
     *
     * uniApply的返回值含义：
     * 返回false，代表前一个stage还没完成。也可代表，当前stage的函数式接口已经被别的线程执行了。
     * 返回true，代表前一个stage已经完成，并且当前stage的函数式接口被当前线程执行了。
     * 函数返回后，回到tryFire，将执行d.postFire(a, mode)，因为执行完毕了当前stage的函数式接口，当前线
程就得处理当前stage的后续任务。
     */
    // this永远是当前stage，a参数永远是前一个stage
    final <S> boolean uniApply(CompletableFuture<S> a,
                               Function<? super S,? extends T> f,
                               UniApply<S,T> c) {
        Object r; Throwable x;
        // 前后两个条件只是优雅的避免空指针异常，实际不可能发生。
        // 如果 前一个stage的result为null，说明前一个stage还没执行完毕
        if (a == null || (r = a.result) == null || f == null)
```

```
            return false;
        // 执行到这里，说明前一个stage执行完毕
        // 如果this即当前stage的result不为null，说当前stage还没执行。
        tryComplete: if (result == null) {
            // 如果前一个stage的执行结果为null或者抛出异常
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    // 如果前一个stage抛出异常，那么直接让当前stage的执行结果也为这个异常，都不用执行
Function了
                    completeThrowable(x, r);
                    break tryComplete;
                }
                // 如果前一个stage的执行结果为null
                // 那么让r变成null
                r = null;
            }
            try {
                //1. c为null，这说明c还没有入栈，没有线程竞争。直接执行当前stage即f.apply(s)
                //2. c不为null，这说明c已经入栈了，有线程竞争执行当前stage。
                if (c != null && !c.claim())
                    // claim返回了false，说明当前线程不允许执行当前stage，直接返回
                    return false;
                // claim返回了true，说明当前线程允许接下来执行当前stage
                @SuppressWarnings("unchecked") S s = (S) r;
                completeValue(f.apply(s));
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        // 如果走到这里说明当前stage的result不为null，即当前stage已经执行完毕，那么直接返回true
        return true;
    }

    private <V> CompletableFuture<V> uniApplyStage(
        Executor e, Function<? super T,? extends V> f) {
        if (f == null) throw new NullPointerException();
        // 从CompletableFuture<V> d = new CompletableFuture<V>()和return d来看，还是和之前一样，
new出来一个CompletableFuture对象后就尽快返回。
        CompletableFuture<V> d =  new CompletableFuture<V>();
        // 如果Executor e为null（当前stage是可以允许被同步执行的），并且此时前一个stage已经结束了，这种
情况应该让当前线程来同步执行当前stage。但我们其实不知道前一个stage是否结束，所以通过d.uniApply(this, f,
null)检测前一个stage是否已经结束。如果d.uniApply(this, f, null)返回true，说明发现了前一个stage已经结
束，并且当前线程执行完毕当前stage，所以这种情况就会直接return d。
        // d.uniApply(this, f, null)的第三个实参为null，这代表与当前stage相关联的Completion对象还没
有入栈（还没push(c)），即不可能有别的线程与当前线程来竞争执行当前stage。这样d.uniApply(this, f, null)里
面的逻辑就变简单了，要么发现前一个stage还没执行完，直接返回false；要么发现前一个stage执行完毕，那么执行当前
stage后，返回true。
        //进入此分支有两种情况：
        //1. 要么e不为null，如果前一个stage已经执行完毕：当前线程在c.tryFire(SYNC)中把接管的当前stage
转交给e执行。如果前一个stage还没执行完毕：当前线程会直接返回，等到执行前一个stage的线程来把当前stage转交给e
执行。
        //2. 要么e为null，但前一个stage还没执行完毕。所以只能入栈等待
        if (e != null || !d.uniApply(this, f, null)) {
            UniApply<T,V> c = new UniApply<T,V>(e, d, this, f);
            push(c);
            //（考虑e为null）入栈后需要避免，入栈后刚好前一个stage已经执行完毕的情况。这种特殊情况，如果不
执行c.tryFire(SYNC)，当前stage永远不会完成。
            //（考虑e不为null）入栈后需要避免，入栈前前一个stage已经执行完毕的情况。
            //下面这句，有可能发现前一个stage已经执行完毕，然后马上执行当前stage
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniAccept<T> extends UniCompletion<T,Void> {
        Consumer<? super T> fn;
        UniAccept(Executor executor, CompletableFuture<Void> dep,
                  CompletableFuture<T> src, Consumer<? super T> fn) {
            super(executor, dep, src); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d; CompletableFuture<T> a;
            if ((d = dep) == null ||
                !d.uniAccept(a = src, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final <S> boolean uniAccept(CompletableFuture<S> a,
                                Consumer<? super S> f, UniAccept<S> c) {
        Object r; Throwable x;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        tryComplete: if (result == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    completeThrowable(x, r);
                    break tryComplete;
                }
                r = null;
            }
            try {
                if (c != null && !c.claim())
                    return false;
                @SuppressWarnings("unchecked") S s = (S) r;
                f.accept(s);
                completeNull();
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }
```

```java
    private CompletableFuture<Void> uniAcceptStage(Executor e,
                                                   Consumer<? super T> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.uniAccept(this, f, null)) {
            UniAccept<T> c = new UniAccept<T>(e, d, this, f);
            push(c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniRun<T> extends UniCompletion<T,Void> {
        Runnable fn;
        UniRun(Executor executor, CompletableFuture<Void> dep,
               CompletableFuture<T> src, Runnable fn) {
            super(executor, dep, src); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d; CompletableFuture<T> a;
            if ((d = dep) == null ||
                !d.uniRun(a = src, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final boolean uniRun(CompletableFuture<?> a, Runnable f, UniRun<?> c) {
        Object r; Throwable x;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        if (result == null) {
            if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)
                completeThrowable(x, r);
            else
                try {
                    if (c != null && !c.claim())
                        return false;
                    f.run();
                    completeNull();
                } catch (Throwable ex) {
                    completeThrowable(ex);
                }
        }
        return true;
    }

    private CompletableFuture<Void> uniRunStage(Executor e, Runnable f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.uniRun(this, f, null)) {
            UniRun<T> c = new UniRun<T>(e, d, this, f);
            push(c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniWhenComplete<T> extends UniCompletion<T,T> {
        BiConsumer<? super T, ? super Throwable> fn;
        UniWhenComplete(Executor executor, CompletableFuture<T> dep,
                        CompletableFuture<T> src,
                        BiConsumer<? super T, ? super Throwable> fn) {
            super(executor, dep, src); this.fn = fn;
        }
        final CompletableFuture<T> tryFire(int mode) {
            CompletableFuture<T> d; CompletableFuture<T> a;
            if ((d = dep) == null ||
                !d.uniWhenComplete(a = src, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final boolean uniWhenComplete(CompletableFuture<T> a,
                                  BiConsumer<? super T,? super Throwable> f,
                                  UniWhenComplete<T> c) {
        Object r; T t; Throwable x = null;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        if (result == null) {
            try {
                if (c != null && !c.claim())
                    return false;
                if (r instanceof AltResult) {
                    x = ((AltResult)r).ex;
                    t = null;
                } else {
                    @SuppressWarnings("unchecked") T tr = (T) r;
                    t = tr;
                }
                f.accept(t, x);
                if (x == null) {
                    internalComplete(r);
                    return true;
                }
            } catch (Throwable ex) {
                if (x == null)
                    x = ex;
            }
            completeThrowable(x, r);
        }
        return true;
    }

    private CompletableFuture<T> uniWhenCompleteStage(
        Executor e, BiConsumer<? super T, ? super Throwable> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<T> d = new CompletableFuture<T>();
        if (e != null || !d.uniWhenComplete(this, f, null)) {
            UniWhenComplete<T> c = new UniWhenComplete<T>(e, d, this, f);
            push(c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniHandle<T,V> extends UniCompletion<T,V> {
        BiFunction<? super T, Throwable, ? extends V> fn;
        UniHandle(Executor executor, CompletableFuture<V> dep,
                  CompletableFuture<T> src,
                  BiFunction<? super T, Throwable, ? extends V> fn) {
            super(executor, dep, src); this.fn = fn;
        }
        final CompletableFuture<V> tryFire(int mode) {
            CompletableFuture<V> d; CompletableFuture<T> a;
            if ((d = dep) == null ||
                !d.uniHandle(a = src, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final <S> boolean uniHandle(CompletableFuture<S> a,
                                BiFunction<? super S, Throwable, ? extends T> f,
                                UniHandle<S,T> c) {
        Object r; S s; Throwable x;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        if (result == null) {
            try {
                if (c != null && !c.claim())
                    return false;
                if (r instanceof AltResult) {
                    x = ((AltResult)r).ex;
                    s = null;
                } else {
                    x = null;
                    @SuppressWarnings("unchecked") S ss = (S) r;
                    s = ss;
                }
                completeValue(f.apply(s, x));
```

```java
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <V> CompletableFuture<V> uniHandleStage(
        Executor e, BiFunction<? super T, Throwable, ? extends V> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<V> d = new CompletableFuture<V>();
        if (e != null || !d.uniHandle(this, f, null)) {
            UniHandle<T,V> c = new UniHandle<T,V>(e, d, this, f);
            push(c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniExceptionally<T> extends UniCompletion<T,T> {
        Function<? super Throwable, ? extends T> fn;
        UniExceptionally(CompletableFuture<T> dep, CompletableFuture<T> src,
                         Function<? super Throwable, ? extends T> fn) {
            super(null, dep, src); this.fn = fn;
        }
        final CompletableFuture<T> tryFire(int mode) { // never ASYNC
            // assert mode != ASYNC;
            CompletableFuture<T> d; CompletableFuture<T> a;
            if ((d = dep) == null || !d.uniExceptionally(a = src, fn, this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final boolean uniExceptionally(CompletableFuture<T> a,
                                   Function<? super Throwable, ? extends T> f,
                                   UniExceptionally<T> c) {
        Object r; Throwable x;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        if (result == null) {
            try {
                if (r instanceof AltResult && (x = ((AltResult)r).ex) != null) {
                    if (c != null && !c.claim())
                        return false;
                    completeValue(f.apply(x));
                } else
                    internalComplete(r);
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private CompletableFuture<T> uniExceptionallyStage(
        Function<Throwable, ? extends T> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<T> d = new CompletableFuture<T>();
        if (!d.uniExceptionally(this, f, null)) {
            UniExceptionally<T> c = new UniExceptionally<T>(d, this, f);
            push(c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class UniRelay<T> extends UniCompletion<T,T> { // for Compose
        UniRelay(CompletableFuture<T> dep, CompletableFuture<T> src) {
            super(null, dep, src);
        }
        final CompletableFuture<T> tryFire(int mode) {
            CompletableFuture<T> d; CompletableFuture<T> a;
            if ((d = dep) == null || !d.uniRelay(a = src))
                return null;
            src = null; dep = null;
            return d.postFire(a, mode);
        }
    }

    final boolean uniRelay(CompletableFuture<T> a) {
        Object r;
        if (a == null || (r = a.result) == null)
            return false;
        if (result == null) // no need to claim
            completeRelay(r);
        return true;
    }

    @SuppressWarnings("serial")
    static final class UniCompose<T,V> extends UniCompletion<T,V> {
        Function<? super T, ? extends CompletionStage<V>> fn;
        UniCompose(Executor executor, CompletableFuture<V> dep,
                   CompletableFuture<T> src,
                   Function<? super T, ? extends CompletionStage<V>> fn) {
            super(executor, dep, src); this.fn = fn;
        }
        final CompletableFuture<V> tryFire(int mode) {
            CompletableFuture<V> d; CompletableFuture<T> a;
            if ((d = dep) == null ||
                !d.uniCompose(a = src, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; fn = null;
            return d.postFire(a, mode);
        }
    }

    final <S> boolean uniCompose(
        CompletableFuture<S> a,
        Function<? super S, ? extends CompletionStage<T>> f,
        UniCompose<S,T> c) {
        Object r; Throwable x;
        if (a == null || (r = a.result) == null || f == null)
            return false;
        tryComplete: if (result == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    completeThrowable(x, r);
                    break tryComplete;
                }
                r = null;
            }
            try {
                if (c != null && !c.claim())
                    return false;
                @SuppressWarnings("unchecked") S s = (S) r;
                CompletableFuture<T> g = f.apply(s).toCompletableFuture();
                if (g.result == null || !uniRelay(g)) {
                    UniRelay<T> copy = new UniRelay<T>(this, g);
                    g.push(copy);
                    copy.tryFire(SYNC);
                    if (result == null)
                        return false;
                }
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <V> CompletableFuture<V> uniComposeStage(
        Executor e, Function<? super T, ? extends CompletionStage<V>> f) {
        if (f == null) throw new NullPointerException();
        Object r; Throwable x;
        if (e == null && (r = result) != null) {
            // try to return function result directly
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    return new CompletableFuture<V>(encodeThrowable(x, r));
                }
                r = null;
            }
            try {
                @SuppressWarnings("unchecked") T t = (T) r;
                CompletableFuture<V> g = f.apply(t).toCompletableFuture();
                Object s = g.result;
                if (s != null)
                    return new CompletableFuture<V>(encodeRelay(s));
                CompletableFuture<V> d = new CompletableFuture<V>();
                UniRelay<V> copy = new UniRelay<V>(d, g);
                g.push(copy);
                copy.tryFire(SYNC);
                return d;
            } catch (Throwable ex) {
                return new CompletableFuture<V>(encodeThrowable(ex));
            }
        }
        CompletableFuture<V> d = new CompletableFuture<V>();
        UniCompose<T,V> c = new UniCompose<T,V>(e, d, this, f);
        push(c);
        c.tryFire(SYNC);
        return d;
    }

    /* ------------- Two-input Completions -------------- */

    /** A Completion for an action with two sources */
    @SuppressWarnings("serial")
    abstract static class BiCompletion<T,U,V> extends UniCompletion<T,V> {
        CompletableFuture<U> snd; // second source for action
        BiCompletion(Executor executor, CompletableFuture<V> dep,
                     CompletableFuture<T> src, CompletableFuture<U> snd) {
            super(executor, dep, src); this.snd = snd;
        }
    }

    /** A Completion delegating to a BiCompletion */
    @SuppressWarnings("serial")
    static final class CoCompletion extends Completion {
        BiCompletion<?,?,?> base;
        CoCompletion(BiCompletion<?,?,?> base) { this.base = base; }
        final CompletableFuture<?> tryFire(int mode) {
            BiCompletion<?,?,?> c; CompletableFuture<?> d;
            if ((c = base) == null || (d = c.tryFire(mode)) == null)
                return null;
            base = null; // detach
            return d;
        }
        final boolean isLive() {
            BiCompletion<?,?,?> c;
            return (c = base) != null && c.dep != null;
        }
    }

    /** Pushes completion to this and b unless both done. */
    final void bipush(CompletableFuture<?> b, BiCompletion<?,?,?> c) {
        if (c != null) {
            Object r;
            while ((r = result) == null && !tryPushStack(c))
                lazySetNext(c, null); // clear on failure
            if (b != null && b != this && b.result == null) {
                Completion q = (r != null) ? c : new CoCompletion(c);
                while (b.result == null && !b.tryPushStack(q))
                    lazySetNext(q, null); // clear on failure
            }
        }
    }

    /** Post-processing after successful BiCompletion tryFire. */
    final CompletableFuture<T> postFire(CompletableFuture<?> a,
                                        CompletableFuture<?> b, int mode) {
        if (b != null && b.stack != null) { // clean second source
            if (mode < 0 || b.result == null)
                b.cleanStack();
            else
                b.postComplete();
        }
        return postFire(a, mode);
    }

    @SuppressWarnings("serial")
    static final class BiApply<T,U,V> extends BiCompletion<T,U,V> {
        BiFunction<? super T,? super U,? extends V> fn;
        BiApply(Executor executor, CompletableFuture<V> dep,
                CompletableFuture<T> src, CompletableFuture<U> snd,
                BiFunction<? super T,? super U,? extends V> fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<V> tryFire(int mode) {
            CompletableFuture<V> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.biApply(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final <R,S> boolean biApply(CompletableFuture<R> a,
                                CompletableFuture<S> b,
                                BiFunction<? super R,? super S,? extends T> f,
                                BiApply<R,S,T> c) {
        Object r, s; Throwable x;
        if (a == null || (r = a.result) == null ||
            b == null || (s = b.result) == null || f == null)
            return false;
        tryComplete: if (result == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
```

```java
                    completeThrowable(x, r);
                    break tryComplete;
                }
                r = null;
            }
            if (s instanceof AltResult) {
                if ((x = ((AltResult)s).ex) != null) {
                    completeThrowable(x, s);
                    break tryComplete;
                }
                s = null;
            }
            try {
                if (c != null && !c.claim())
                    return false;
                @SuppressWarnings("unchecked") R rr = (R) r;
                @SuppressWarnings("unchecked") S ss = (S) s;
                completeValue(f.apply(rr, ss));
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <U,V> CompletableFuture<V> biApplyStage(
        Executor e, CompletionStage<U> o,
        BiFunction<? super T,? super U,? extends V> f) {
        CompletableFuture<U> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
            throw new NullPointerException();
        CompletableFuture<V> d = new CompletableFuture<V>();
        if (e != null || !d.biApply(this, b, f, null)) {
            BiApply<T,U,V> c = new BiApply<T,U,V>(e, d, this, b, f);
            bipush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class BiAccept<T,U> extends BiCompletion<T,U,Void> {
        BiConsumer<? super T,? super U> fn;
        BiAccept(Executor executor, CompletableFuture<Void> dep,
                 CompletableFuture<T> src, CompletableFuture<U> snd,
                 BiConsumer<? super T,? super U> fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.biAccept(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final <R,S> boolean biAccept(CompletableFuture<R> a,
                                 CompletableFuture<S> b,
                                 BiConsumer<? super R,? super S> f,
                                 BiAccept<R,S> c) {
        Object r, s; Throwable x;
        if (a == null || (r = a.result) == null ||
            b == null || (s = b.result) == null || f == null)
            return false;
        tryComplete: if (result == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    completeThrowable(x, r);
                    break tryComplete;
                }
                r = null;
            }
            if (s instanceof AltResult) {
                if ((x = ((AltResult)s).ex) != null) {
                    completeThrowable(x, s);
                    break tryComplete;
                }
                s = null;
            }
            try {
                if (c != null && !c.claim())
                    return false;
                @SuppressWarnings("unchecked") R rr = (R) r;
                @SuppressWarnings("unchecked") S ss = (S) s;
                f.accept(rr, ss);
                completeNull();
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <U> CompletableFuture<Void> biAcceptStage(
        Executor e, CompletionStage<U> o,
        BiConsumer<? super T,? super U> f) {
        CompletableFuture<U> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
            throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.biAccept(this, b, f, null)) {
            BiAccept<T,U> c = new BiAccept<T,U>(e, d, this, b, f);
```

```java
            bipush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class BiRun<T,U> extends BiCompletion<T,U,Void> {
        Runnable fn;
        BiRun(Executor executor, CompletableFuture<Void> dep,
              CompletableFuture<T> src,
              CompletableFuture<U> snd,
              Runnable fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.biRun(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final boolean biRun(CompletableFuture<?> a, CompletableFuture<?> b,
                        Runnable f, BiRun<?,?> c) {
        Object r, s; Throwable x;
        if (a == null || (r = a.result) == null ||
            b == null || (s = b.result) == null || f == null)
            return false;
        if (result == null) {
            if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)
                completeThrowable(x, r);
            else if (s instanceof AltResult && (x = ((AltResult)s).ex) != null)
                completeThrowable(x, s);
            else
                try {
                    if (c != null && !c.claim())
                        return false;
                    f.run();
                    completeNull();
                } catch (Throwable ex) {
                    completeThrowable(ex);
                }
        }
        return true;
    }

    private CompletableFuture<Void> biRunStage(Executor e, CompletionStage<?> o,
                                               Runnable f) {
```

```java
        CompletableFuture<?> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
            throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.biRun(this, b, f, null)) {
            BiRun<T,?> c = new BiRun<>(e, d, this, b, f);
            bipush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class BiRelay<T,U> extends BiCompletion<T,U,Void> { // for And
        BiRelay(CompletableFuture<Void> dep,
                CompletableFuture<T> src,
                CompletableFuture<U> snd) {
            super(null, dep, src, snd);
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null || !d.biRelay(a = src, b = snd))
                return null;
            src = null; snd = null; dep = null;
            return d.postFire(a, b, mode);
        }
    }

    boolean biRelay(CompletableFuture<?> a, CompletableFuture<?> b) {
        Object r, s; Throwable x;
        if (a == null || (r = a.result) == null ||
            b == null || (s = b.result) == null)
            return false;
        if (result == null) {
            if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)
                completeThrowable(x, r);
            else if (s instanceof AltResult && (x = ((AltResult)s).ex) != null)
                completeThrowable(x, s);
            else
                completeNull();
        }
        return true;
    }

    /** Recursively constructs a tree of completions. */
    static CompletableFuture<Void> andTree(CompletableFuture<?>[] cfs,
                                           int lo, int hi) {
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (lo > hi) // empty
            d.result = NIL;
        else {
            CompletableFuture<?> a, b;
            int mid = (lo + hi) >>> 1;
            if ((a = (lo == mid ? cfs[lo] :
                      andTree(cfs, lo, mid))) == null ||
                (b = (lo == hi ? a : (hi == mid+1) ? cfs[hi] :
                      andTree(cfs, mid+1, hi)))  == null)
                throw new NullPointerException();
            if (!d.biRelay(a, b)) {
                BiRelay<?,?> c = new BiRelay<>(d, a, b);
                a.bipush(b, c);
                c.tryFire(SYNC);
            }
        }
        return d;
    }

    /* ------------- Projected (Ored) BiCompletions -------------- */

    /** Pushes completion to this and b unless either done. */
    final void orpush(CompletableFuture<?> b, BiCompletion<?,?,?> c) {
        if (c != null) {
            while ((b == null || b.result == null) && result == null) {
                if (tryPushStack(c)) {
                    if (b != null && b != this && b.result == null) {
                        Completion q = new CoCompletion(c);
                        while (result == null && b.result == null &&
                               !b.tryPushStack(q))
                            lazySetNext(q, null); // clear on failure
                    }
                    break;
                }
                lazySetNext(c, null); // clear on failure
            }
        }
    }

    @SuppressWarnings("serial")
    static final class OrApply<T,U extends T,V> extends BiCompletion<T,U,V> {
        Function<? super T,? extends V> fn;
        OrApply(Executor executor, CompletableFuture<V> dep,
                CompletableFuture<T> src,
                CompletableFuture<U> snd,
                Function<? super T,? extends V> fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<V> tryFire(int mode) {
            CompletableFuture<V> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.orApply(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final <R,S extends R> boolean orApply(CompletableFuture<R> a,
                                          CompletableFuture<S> b,
                                          Function<? super R, ? extends T> f,
                                          OrApply<R,S,T> c) {
        Object r; Throwable x;
        if (a == null || b == null ||
            ((r = a.result) == null && (r = b.result) == null) || f == null)
            return false;
        tryComplete: if (result == null) {
            try {
                if (c != null && !c.claim())
                    return false;
                if (r instanceof AltResult) {
                    if ((x = ((AltResult)r).ex) != null) {
                        completeThrowable(x, r);
                        break tryComplete;
                    }
                    r = null;
                }
                @SuppressWarnings("unchecked") R rr = (R) r;
                completeValue(f.apply(rr));
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <U extends T,V> CompletableFuture<V> orApplyStage(
        Executor e, CompletionStage<U> o,
        Function<? super T, ? extends V> f) {
        CompletableFuture<U> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
            throw new NullPointerException();
        CompletableFuture<V> d = new CompletableFuture<V>();
        if (e != null || !d.orApply(this, b, f, null)) {
            OrApply<T,U,V> c = new OrApply<T,U,V>(e, d, this, b, f);
            orpush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class OrAccept<T,U extends T> extends BiCompletion<T,U,Void> {
        Consumer<? super T> fn;
        OrAccept(Executor executor, CompletableFuture<Void> dep,
                 CompletableFuture<T> src,
                 CompletableFuture<U> snd,
                 Consumer<? super T> fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.orAccept(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final <R,S extends R> boolean orAccept(CompletableFuture<R> a,
                                           CompletableFuture<S> b,
                                           Consumer<? super R> f,
                                           OrAccept<R,S> c) {
        Object r; Throwable x;
        if (a == null || b == null ||
            ((r = a.result) == null && (r = b.result) == null) || f == null)
            return false;
        tryComplete: if (result == null) {
            try {
                if (c != null && !c.claim())
                    return false;
                if (r instanceof AltResult) {
                    if ((x = ((AltResult)r).ex) != null) {
                        completeThrowable(x, r);
                        break tryComplete;
                    }
                    r = null;
                }
                @SuppressWarnings("unchecked") R rr = (R) r;
                f.accept(rr);
                completeNull();
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private <U extends T> CompletableFuture<Void> orAcceptStage(
        Executor e, CompletionStage<U> o, Consumer<? super T> f) {
        CompletableFuture<U> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
            throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.orAccept(this, b, f, null)) {
            OrAccept<T,U> c = new OrAccept<T,U>(e, d, this, b, f);
            orpush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class OrRun<T,U> extends BiCompletion<T,U,Void> {
        Runnable fn;
        OrRun(Executor executor, CompletableFuture<Void> dep,
              CompletableFuture<T> src,
              CompletableFuture<U> snd,
              Runnable fn) {
            super(executor, dep, src, snd); this.fn = fn;
        }
        final CompletableFuture<Void> tryFire(int mode) {
            CompletableFuture<Void> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null ||
                !d.orRun(a = src, b = snd, fn, mode > 0 ? null : this))
                return null;
            dep = null; src = null; snd = null; fn = null;
            return d.postFire(a, b, mode);
        }
    }

    final boolean orRun(CompletableFuture<?> a, CompletableFuture<?> b,
                        Runnable f, OrRun<?,?> c) {
        Object r; Throwable x;
        if (a == null || b == null ||
            ((r = a.result) == null && (r = b.result) == null) || f == null)
            return false;
        if (result == null) {
            try {
                if (c != null && !c.claim())
                    return false;
                if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)
                    completeThrowable(x, r);
                else {
                    f.run();
                    completeNull();
                }
            } catch (Throwable ex) {
                completeThrowable(ex);
            }
        }
        return true;
    }

    private CompletableFuture<Void> orRunStage(Executor e, CompletionStage<?> o,
                                               Runnable f) {
        CompletableFuture<?> b;
        if (f == null || (b = o.toCompletableFuture()) == null)
```

```
            throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        if (e != null || !d.orRun(this, b, f, null)) {
            OrRun<T,?> c = new OrRun<>(e, d, this, b, f);
            orpush(b, c);
            c.tryFire(SYNC);
        }
        return d;
    }

    @SuppressWarnings("serial")
    static final class OrRelay<T,U> extends BiCompletion<T,U,Object> { // for Or
        OrRelay(CompletableFuture<Object> dep, CompletableFuture<T> src,
                CompletableFuture<U> snd) {
            super(null, dep, src, snd);
        }
        final CompletableFuture<Object> tryFire(int mode) {
            CompletableFuture<Object> d;
            CompletableFuture<T> a;
            CompletableFuture<U> b;
            if ((d = dep) == null || !d.orRelay(a = src, b = snd))
                return null;
            src = null; snd = null; dep = null;
            return d.postFire(a, b, mode);
        }
    }

    final boolean orRelay(CompletableFuture<?> a, CompletableFuture<?> b) {
        Object r;
        if (a == null || b == null ||
            ((r = a.result) == null && (r = b.result) == null))
            return false;
        if (result == null)
            completeRelay(r);
        return true;
    }

    /** Recursively constructs a tree of completions. */
    static CompletableFuture<Object> orTree(CompletableFuture<?>[] cfs,
                                            int lo, int hi) {
        CompletableFuture<Object> d = new CompletableFuture<Object>();
        if (lo <= hi) {
            CompletableFuture<?> a, b;
            int mid = (lo + hi) >>> 1;
            if ((a = (lo == mid ? cfs[lo] :
                      orTree(cfs, lo, mid))) == null ||
                (b = (lo == hi ? a : (hi == mid+1) ? cfs[hi] :
                      orTree(cfs, mid+1, hi)))  == null)
                throw new NullPointerException();
            if (!d.orRelay(a, b)) {
                OrRelay<?,?> c = new OrRelay<>(d, a, b);
                a.orpush(b, c);
                c.tryFire(SYNC);
            }
        }
        return d;
    }

    /* ------------- Zero-input Async forms -------------- */

    @SuppressWarnings("serial")
    static final class AsyncSupply<T> extends ForkJoinTask<Void>
            implements Runnable, AsynchronousCompletionTask {
        CompletableFuture<T> dep; Supplier<T> fn;
        AsyncSupply(CompletableFuture<T> dep, Supplier<T> fn) {
            this.dep = dep; this.fn = fn;
        }

        public final Void getRawResult() { return null; }
        public final void setRawResult(Void v) {}
        public final boolean exec() { run(); return true; }

        public void run() {
            CompletableFuture<T> d; Supplier<T> f;
            if ((d = dep) != null && (f = fn) != null) {
                // 为了防止内存泄漏，方便GC.
                dep = null; fn = null;
                if (d.result == null) {
                    try {
                        // task被异步执行完毕后，会调用completeValue或completeThrowable来为result
成员赋值。stage完成的标志，就是它的result成员非null。
                        d.completeValue(f.get());
                    } catch (Throwable ex) {
                        d.completeThrowable(ex);
                    }
                }
                d.postComplete();
            }
        }
    }

    static <U> CompletableFuture<U> asyncSupplyStage(Executor e,
                                                     Supplier<U> f) {
        if (f == null) throw new NullPointerException();
        // CompletableFuture对象是new出来以后就直接返回的，但是刚new的CompletableFuture对象的result
成员是为null，因为task还没有执行完。
        CompletableFuture<U> d = new CompletableFuture<U>();
        // task的执行交给了e.execute(new AsyncSupply<U>(d, f))
        e.execute(new AsyncSupply<U>(d, f));
        return d;
    }

    @SuppressWarnings("serial")
    static final class AsyncRun extends ForkJoinTask<Void>
            implements Runnable, AsynchronousCompletionTask {
        CompletableFuture<Void> dep; Runnable fn;
```

```java
        AsyncRun(CompletableFuture<Void> dep, Runnable fn) {
            this.dep = dep; this.fn = fn;
        }

        public final Void getRawResult() { return null; }
        public final void setRawResult(Void v) {}
        public final boolean exec() { run(); return true; }

        public void run() {
            CompletableFuture<Void> d; Runnable f;
            if ((d = dep) != null && (f = fn) != null) {
                dep = null; fn = null;
                if (d.result == null) {
                    try {
                        f.run();
                        d.completeNull();
                    } catch (Throwable ex) {
                        d.completeThrowable(ex);
                    }
                }
                d.postComplete();
            }
        }
    }

    static CompletableFuture<Void> asyncRunStage(Executor e, Runnable f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<Void> d = new CompletableFuture<Void>();
        e.execute(new AsyncRun(d, f));
        return d;
    }

    /* ------------- Signallers -------------- */

    // 配合get或者join使用的，实现对想获取执行结果的线程的阻塞和唤醒的功能。
    @SuppressWarnings("serial")
    static final class Signaller extends Completion
        implements ForkJoinPool.ManagedBlocker {
        long nanos;                    // wait time if timed
        final long deadline;           // non-zero if timed
        volatile int interruptControl; // > 0: interruptible, < 0: interrupted
        volatile Thread thread;

        Signaller(boolean interruptible, long nanos, long deadline) {
            this.thread = Thread.currentThread();
            this.interruptControl = interruptible ? 1 : 0;
            this.nanos = nanos;
            this.deadline = deadline;
        }
        final CompletableFuture<?> tryFire(int ignore) {
            Thread w; // no need to atomically claim
            if ((w = thread) != null) {
                thread = null;
                LockSupport.unpark(w);
            }
            return null;
        }
        public boolean isReleasable() {
            if (thread == null)
                return true;
            if (Thread.interrupted()) {
                int i = interruptControl;
                interruptControl = -1;
                if (i > 0)
                    return true;
            }
            if (deadline != 0L &&
                (nanos <= 0L || (nanos = deadline - System.nanoTime()) <= 0L)) {
                thread = null;
                return true;
            }
            return false;
        }
        public boolean block() {
            if (isReleasable())
                return true;
            else if (deadline == 0L)
                LockSupport.park(this);
            else if (nanos > 0L)
                LockSupport.parkNanos(this, nanos);
            return isReleasable();
        }
        final boolean isLive() { return thread != null; }
    }

    /**
     * Returns raw result after waiting, or null if interruptible and
     * interrupted.
     */
    private Object waitingGet(boolean interruptible) {
        Signaller q = null;
        boolean queued = false;
        int spins = -1;
        Object r;
        while ((r = result) == null) {
            if (spins < 0)
                spins = (Runtime.getRuntime().availableProcessors() > 1) ?
                    1 << 8 : 0; // Use brief spin-wait on multiprocessors
            else if (spins > 0) {
                if (ThreadLocalRandom.nextSecondarySeed() >= 0)
                    --spins;
            }
            else if (q == null)
                q = new Signaller(interruptible, 0L, 0L);
            else if (!queued)
                queued = tryPushStack(q);
            else if (interruptible && q.interruptControl < 0) {
                q.thread = null;
                cleanStack();
                return null;
            }
            else if (q.thread != null && result == null) {
                try {
                    ForkJoinPool.managedBlock(q);
                } catch (InterruptedException ie) {
                    q.interruptControl = -1;
                }
            }
        }
        if (q != null) {
            q.thread = null;
            if (q.interruptControl < 0) {
                if (interruptible)
                    r = null; // report interruption
                else
                    Thread.currentThread().interrupt();
            }
        }
        postComplete();
        return r;
    }

    /**
     * Returns raw result after waiting, or null if interrupted, or
     * throws TimeoutException on timeout.
     */
    private Object timedGet(long nanos) throws TimeoutException {
        if (Thread.interrupted())
            return null;
        if (nanos <= 0L)
            throw new TimeoutException();
        long d = System.nanoTime() + nanos;
        Signaller q = new Signaller(true, nanos, d == 0L ? 1L : d); // avoid 0
        boolean queued = false;
        Object r;
        // We intentionally don't spin here (as waitingGet does) because
        // the call to nanoTime() above acts much like a spin.
        while ((r = result) == null) {
            if (!queued)
                queued = tryPushStack(q);
            else if (q.interruptControl < 0 || q.nanos <= 0L) {
                q.thread = null;
                cleanStack();
                if (q.interruptControl < 0)
                    return null;
                throw new TimeoutException();
            }
            else if (q.thread != null && result == null) {
                try {
                    ForkJoinPool.managedBlock(q);
                } catch (InterruptedException ie) {
                    q.interruptControl = -1;
                }
            }
        }
        if (q.interruptControl < 0)
            r = null;
        q.thread = null;
        postComplete();
        return r;
    }

    /* ------------- public methods -------------- */

    /**
     * Creates a new incomplete CompletableFuture.
     */
    public CompletableFuture() {
    }

    /**
     * Creates a new complete CompletableFuture with given encoded result.
     */
    private CompletableFuture(Object r) {
        this.result = r;
    }

    /**
     * Async任务的执行过程
     * 1.执行前一个stage的线程来执行UniApply内部类对象的tryFire(NESTED)。
     *   即在supplyAsync->asyncSupplyStage->AsyncSupply的run方法中，在本stage完成后会执行
postComplete，而postComplete中会执行tryFire(对于thenApplyAsync中的stage来说是前一个stage)
     * 2.接着执行当前stage.uniApply()。(在UniApply中dep指的是当前stage，src值的是前一个stage)
     * 3.执行UniApply内部类对象的claim方法。(由于mode是从前一个stage的线程执行UniApply是传下来的，所以
是NESTED，因此d.uniApply(a = src, fn, mode > 0 ? null : this)的第三个参数是this)
     * 4.由于提供了线程池，claim会把任务提交给线程池（e.execute(this)），把保存的线程池清理掉（executor
= null），然后返回false。执行前一个stage的线程接着层层返回，最终tryFire(NESTED)返回null。
     * 前面的步骤说明的是如何将异步任务提交到线程池中，下面则是线程池如何执行任务。
     * 5.线程池选择一个线程，开始对同一个UniApply内部类对象执行tryFire(ASYNC)。(UniApply 继承
UniCompletion，而UniCompletion 又继承 Completion，Completion则继承ForkJoinTask实现Runnable，run
方法中会调用tryFire(ASYNC);所以任务在执行的时候，执行的就是子类中的tryFire方法)
     * 6.由于通过Completion的run方法执行任务的时候，传入tryFire中的参数ASYNC=1，执行当前
stage.uniApply(a = src, fn, ASYNC > 0 ? null : this)，第三个实参肯定为null。
     * 7.if (c != null && !c.claim())不会执行，因为第三个参数为null。接着直接执行
completeValue(f.apply(s));
     * 8.这个线程以“同步”的方式来执行了任务。但它对于执行前一个stage的线程来说是异步的。
     */
    public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
        return asyncSupplyStage(asyncPool, supplier);
    }

    /**
```

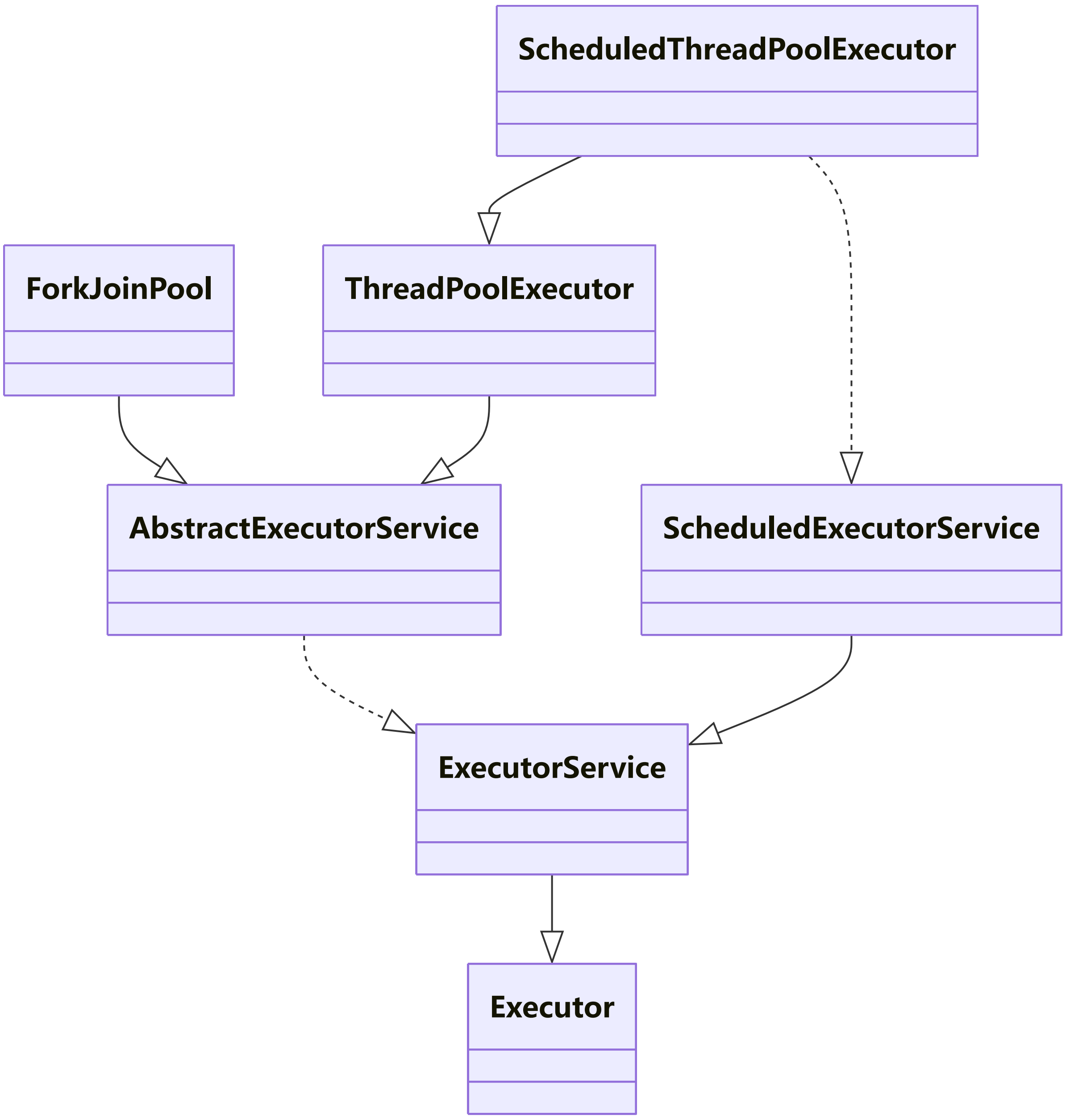
```java
     * Returns a new CompletableFuture that is asynchronously completed
     * by a task running in the given executor with the value obtained
     * by calling the given Supplier.
     *
     * @param supplier a function returning the value to be used
     * to complete the returned CompletableFuture
     * @param executor the executor to use for asynchronous execution
     * @param <U> the function's return type
     * @return the new CompletableFuture
     */
    public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,
                                                       Executor executor) {
        return asyncSupplyStage(screenExecutor(executor), supplier);
    }

    /**
     * Returns a new CompletableFuture that is asynchronously completed
     * by a task running in the {@link ForkJoinPool#commonPool()} after
     * it runs the given action.
     *
     * @param runnable the action to run before completing the
     * returned CompletableFuture
     * @return the new CompletableFuture
     */
    public static CompletableFuture<Void> runAsync(Runnable runnable) {
        return asyncRunStage(asyncPool, runnable);
    }

    /**
     * Returns a new CompletableFuture that is asynchronously completed
     * by a task running in the given executor after it runs the given
     * action.
     *
     * @param runnable the action to run before completing the
     * returned CompletableFuture
     * @param executor the executor to use for asynchronous execution
     * @return the new CompletableFuture
     */
    public static CompletableFuture<Void> runAsync(Runnable runnable,
                                                   Executor executor) {
        return asyncRunStage(screenExecutor(executor), runnable);
    }

    /**
     * Returns a new CompletableFuture that is already completed with
     * the given value.
     *
     * @param value the value
     * @param <U> the type of the value
     * @return the completed CompletableFuture
     */
    public static <U> CompletableFuture<U> completedFuture(U value) {
        return new CompletableFuture<U>((value == null) ? NIL : value);
    }

    /**
     * Returns {@code true} if completed in any fashion: normally,
     * exceptionally, or via cancellation.
     *
     * @return {@code true} if completed
     */
    public boolean isDone() {
        return result != null;
    }

    /**
     * Waits if necessary for this future to complete, and then
     * returns its result.
     *
     * @return the result value
     * @throws CancellationException if this future was cancelled
     * @throws ExecutionException if this future completed exceptionally
     * @throws InterruptedException if the current thread was interrupted
     * while waiting
     */
    public T get() throws InterruptedException, ExecutionException {
        Object r;
        return reportGet((r = result) == null ? waitingGet(true) : r);
    }

    /**
     * Waits if necessary for at most the given time for this future
     * to complete, and then returns its result, if available.
     *
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @return the result value
     * @throws CancellationException if this future was cancelled
     * @throws ExecutionException if this future completed exceptionally
     * @throws InterruptedException if the current thread was interrupted
     * while waiting
     * @throws TimeoutException if the wait timed out
     */
    public T get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        Object r;
        long nanos = unit.toNanos(timeout);
        return reportGet((r = result) == null ? timedGet(nanos) : r);
    }

    /**
     * Returns the result value when complete, or throws an
     * (unchecked) exception if completed exceptionally. To better
     * conform with the use of common functional forms, if a
     * computation involved in the completion of this
     * CompletableFuture threw an exception, this method throws an
     * (unchecked) {@link CompletionException} with the underlying
     * exception as its cause.
     *
     * @return the result value
     * @throws CancellationException if the computation was cancelled
     * @throws CompletionException if this future completed
     * exceptionally or a completion computation threw an exception
     */
    public T join() {
        Object r;
        return reportJoin((r = result) == null ? waitingGet(false) : r);
    }

    /**
     * Returns the result value (or throws any encountered exception)
     * if completed, else returns the given valueIfAbsent.
     *
     * @param valueIfAbsent the value to return if not completed
     * @return the result value, if completed, else the given valueIfAbsent
     * @throws CancellationException if the computation was cancelled
     * @throws CompletionException if this future completed
     * exceptionally or a completion computation threw an exception
     */
    public T getNow(T valueIfAbsent) {
        Object r;
        return ((r = result) == null) ? valueIfAbsent : reportJoin(r);
    }

    /**
     * If not already completed, sets the value returned by {@link
     * #get()} and related methods to the given value.
     *
     * @param value the result value
     * @return {@code true} if this invocation caused this CompletableFuture
     * to transition to a completed state, else {@code false}
     */
    public boolean complete(T value) {
        boolean triggered = completeValue(value);
        postComplete();
        return triggered;
    }

    /**
     * If not already completed, causes invocations of {@link #get()}
     * and related methods to throw the given exception.
     *
     * @param ex the exception
     * @return {@code true} if this invocation caused this CompletableFuture
     * to transition to a completed state, else {@code false}
     */
    public boolean completeExceptionally(Throwable ex) {
        if (ex == null) throw new NullPointerException();
        boolean triggered = internalComplete(new AltResult(ex));
        postComplete();
        return triggered;
    }

// 从supplyAsync + thenApply(thenApplyAsync)理解 下面是测试例子
//    public static void main(String[] args) {
//
//        CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() ->
{
//            String supplyAsyncResult = Thread.currentThread().getName();
////            try {
////                Thread.sleep(1000);
////            } catch (InterruptedException e) {
////                e.printStackTrace();
////            }
//            System.out.println(supplyAsyncResult);
//            return supplyAsyncResult;
////        }).thenApplyAsync(r -> {  //添加后续任务
//        }).thenApply(r -> {  //添加后续任务
//            String thenApplyResult = Thread.currentThread().getName();
//            System.out.println(thenApplyResult);
//            return thenApplyResult;
//        });
//        try {
//            completableFuture.get();
//        } catch (InterruptedException | ExecutionException e) {
//            e.printStackTrace();
//        }
//    }

    // thenApply不会传入Executor，因为它优先让当前线程（例子中是main线程）来执行后续stage的task。具体的
说：
    // 当发现前一个stage已经执行完毕时，直接让当前线程来执行后续stage的task。
    // 当发现前一个stage还没执行完毕时，则把当前stage包装成一个UniApply对象，放到前一个stage的栈中。执行
前一个stage的线程，执行完毕后，接着执行后续stage的task。
    // 总之，要么是一个异步线程走到底(这个异步线程是CompletableFuture.supplyAsync时启动的)，要么让当前
线程来执行后续stage（因为异步线程已经结束，而且你又没有给Executor，那只好让当前线程来执行咯）。
    public <U> CompletableFuture<U> thenApply(
        Function<? super T,? extends U> fn) {
        return uniApplyStage(null, fn);
    }

    // thenApplyAsync会传入一个Executor，因为它总是让 Executor线程池里面的线程来执行后续stage的task。
具体的说：
    // 当发现前一个stage已经执行完毕时，直接让Executor来执行。
    // 当发现前一个stage还没执行完毕时，则等到执行前一个stage的线程执行完毕后，再让Executor来执行。
    // 总之，无论哪种情况，执行后一个stage的线程肯定不是当前添加后续stage的线程（例子中是main线程）了。
    public <U> CompletableFuture<U> thenApplyAsync(
        Function<? super T,? extends U> fn) {
        return uniApplyStage(asyncPool, fn);
    }

    public <U> CompletableFuture<U> thenApplyAsync(
```

```java
        Function<? super T,? extends U> fn, Executor executor) {
        return uniApplyStage(screenExecutor(executor), fn);
    }

    public CompletableFuture<Void> thenAccept(Consumer<? super T> action) {
        return uniAcceptStage(null, action);
    }

    public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action) {
        return uniAcceptStage(asyncPool, action);
    }

    public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action,
                                                   Executor executor) {
        return uniAcceptStage(screenExecutor(executor), action);
    }

    public CompletableFuture<Void> thenRun(Runnable action) {
        return uniRunStage(null, action);
    }

    public CompletableFuture<Void> thenRunAsync(Runnable action) {
        return uniRunStage(asyncPool, action);
    }

    public CompletableFuture<Void> thenRunAsync(Runnable action,
                                                Executor executor) {
        return uniRunStage(screenExecutor(executor), action);
    }

    public <U,V> CompletableFuture<V> thenCombine(
        CompletionStage<? extends U> other,
        BiFunction<? super T,? super U,? extends V> fn) {
        return biApplyStage(null, other, fn);
    }

    public <U,V> CompletableFuture<V> thenCombineAsync(
        CompletionStage<? extends U> other,
        BiFunction<? super T,? super U,? extends V> fn) {
        return biApplyStage(asyncPool, other, fn);
    }

    public <U,V> CompletableFuture<V> thenCombineAsync(
        CompletionStage<? extends U> other,
        BiFunction<? super T,? super U,? extends V> fn, Executor executor) {
        return biApplyStage(screenExecutor(executor), other, fn);
    }

    public <U> CompletableFuture<Void> thenAcceptBoth(
        CompletionStage<? extends U> other,
        BiConsumer<? super T, ? super U> action) {
        return biAcceptStage(null, other, action);
    }
    public <U> CompletableFuture<Void> thenAcceptBothAsync(
        CompletionStage<? extends U> other,
        BiConsumer<? super T, ? super U> action) {
        return biAcceptStage(asyncPool, other, action);
    }

    public <U> CompletableFuture<Void> thenAcceptBothAsync(
        CompletionStage<? extends U> other,
        BiConsumer<? super T, ? super U> action, Executor executor) {
        return biAcceptStage(screenExecutor(executor), other, action);
    }

    public CompletableFuture<Void> runAfterBoth(CompletionStage<?> other,
                                                Runnable action) {
        return biRunStage(null, other, action);
    }

    public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                     Runnable action) {
        return biRunStage(asyncPool, other, action);
    }

    public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                     Runnable action,
                                                     Executor executor) {
        return biRunStage(screenExecutor(executor), other, action);
    }

    public <U> CompletableFuture<U> applyToEither(
        CompletionStage<? extends T> other, Function<? super T, U> fn) {
        return orApplyStage(null, other, fn);
    }

    public <U> CompletableFuture<U> applyToEitherAsync(
        CompletionStage<? extends T> other, Function<? super T, U> fn) {
        return orApplyStage(asyncPool, other, fn);
    }

    public <U> CompletableFuture<U> applyToEitherAsync(
        CompletionStage<? extends T> other, Function<? super T, U> fn,
        Executor executor) {
        return orApplyStage(screenExecutor(executor), other, fn);
    }

    public CompletableFuture<Void> acceptEither(
        CompletionStage<? extends T> other, Consumer<? super T> action) {
        return orAcceptStage(null, other, action);
    }

    public CompletableFuture<Void> acceptEitherAsync(
        CompletionStage<? extends T> other, Consumer<? super T> action) {
        return orAcceptStage(asyncPool, other, action);
    }

    public CompletableFuture<Void> acceptEitherAsync(
        CompletionStage<? extends T> other, Consumer<? super T> action,
        Executor executor) {
        return orAcceptStage(screenExecutor(executor), other, action);
    }

    public CompletableFuture<Void> runAfterEither(CompletionStage<?> other,
                                                  Runnable action) {
        return orRunStage(null, other, action);
    }

    public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                       Runnable action) {
        return orRunStage(asyncPool, other, action);
    }

    public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                       Runnable action,
                                                       Executor executor) {
        return orRunStage(screenExecutor(executor), other, action);
    }

    public <U> CompletableFuture<U> thenCompose(
        Function<? super T, ? extends CompletionStage<U>> fn) {
        return uniComposeStage(null, fn);
    }

    public <U> CompletableFuture<U> thenComposeAsync(
        Function<? super T, ? extends CompletionStage<U>> fn) {
        return uniComposeStage(asyncPool, fn);
    }

    public <U> CompletableFuture<U> thenComposeAsync(
        Function<? super T, ? extends CompletionStage<U>> fn,
        Executor executor) {
        return uniComposeStage(screenExecutor(executor), fn);
    }

    public CompletableFuture<T> whenComplete(
        BiConsumer<? super T, ? super Throwable> action) {
        return uniWhenCompleteStage(null, action);
    }

    public CompletableFuture<T> whenCompleteAsync(
        BiConsumer<? super T, ? super Throwable> action) {
        return uniWhenCompleteStage(asyncPool, action);
    }

    public CompletableFuture<T> whenCompleteAsync(
        BiConsumer<? super T, ? super Throwable> action, Executor executor) {
        return uniWhenCompleteStage(screenExecutor(executor), action);
    }

    public <U> CompletableFuture<U> handle(
        BiFunction<? super T, Throwable, ? extends U> fn) {
        return uniHandleStage(null, fn);
    }

    public <U> CompletableFuture<U> handleAsync(
        BiFunction<? super T, Throwable, ? extends U> fn) {
        return uniHandleStage(asyncPool, fn);
    }

    public <U> CompletableFuture<U> handleAsync(
        BiFunction<? super T, Throwable, ? extends U> fn, Executor executor) {
        return uniHandleStage(screenExecutor(executor), fn);
    }

    /**
     * Returns this CompletableFuture.
     *
     * @return this CompletableFuture
     */
    public CompletableFuture<T> toCompletableFuture() {
        return this;
    }

    // not in interface CompletionStage

    /**
     * Returns a new CompletableFuture that is completed when this
     * CompletableFuture completes, with the result of the given
     * function of the exception triggering this CompletableFuture's
     * completion when it completes exceptionally; otherwise, if this
     * CompletableFuture completes normally, then the returned
     * CompletableFuture also completes normally with the same value.
     * Note: More flexible versions of this functionality are
     * available using methods {@code whenComplete} and {@code handle}.
     *
     * @param fn the function to use to compute the value of the
     * returned CompletableFuture if this CompletableFuture completed
     * exceptionally
     * @return the new CompletableFuture
     */
    public CompletableFuture<T> exceptionally(
        Function<Throwable, ? extends T> fn) {
        return uniExceptionallyStage(fn);
    }

    /* ------------- Arbitrary-arity constructions -------------- */

    /**
     * Returns a new CompletableFuture that is completed when all of
     * the given CompletableFutures complete.  If any of the given
```

```java
     * CompletableFutures complete exceptionally, then the returned
     * CompletableFuture also does so, with a CompletionException
     * holding this exception as its cause.  Otherwise, the results,
     * if any, of the given CompletableFutures are not reflected in
     * the returned CompletableFuture, but may be obtained by
     * inspecting them individually. If no CompletableFutures are
     * provided, returns a CompletableFuture completed with the value
     * {@code null}.
     *
     * <p>Among the applications of this method is to await completion
     * of a set of independent CompletableFutures before continuing a
     * program, as in: {@code CompletableFuture.allOf(c1, c2,
     * c3).join();}.
     *
     * @param cfs the CompletableFutures
     * @return a new CompletableFuture that is completed when all of the
     * given CompletableFutures complete
     * @throws NullPointerException if the array or any of its elements are
     * {@code null}
     */
    public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) {
        return andTree(cfs, 0, cfs.length - 1);
    }

    /**
     * Returns a new CompletableFuture that is completed when any of
     * the given CompletableFutures complete, with the same result.
     * Otherwise, if it completed exceptionally, the returned
     * CompletableFuture also does so, with a CompletionException
     * holding this exception as its cause.  If no CompletableFutures
     * are provided, returns an incomplete CompletableFuture.
     *
     * @param cfs the CompletableFutures
     * @return a new CompletableFuture that is completed with the
     * result or exception of any of the given CompletableFutures when
     * one completes
     * @throws NullPointerException if the array or any of its elements are
     * {@code null}
     */
    public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) {
        return orTree(cfs, 0, cfs.length - 1);
    }

    /* ------------- Control and status methods -------------- */

    /**
     * If not already completed, completes this CompletableFuture with
     * a {@link CancellationException}. Dependent CompletableFutures
     * that have not already completed will also complete
     * exceptionally, with a {@link CompletionException} caused by
     * this {@code CancellationException}.
     *
     * @param mayInterruptIfRunning this value has no effect in this
     * implementation because interrupts are not used to control
     * processing.
     *
     * @return {@code true} if this task is now cancelled
     */
    public boolean cancel(boolean mayInterruptIfRunning) {
        boolean cancelled = (result == null) &&
            internalComplete(new AltResult(new CancellationException()));
        postComplete();
        return cancelled || isCancelled();
    }

    /**
     * Returns {@code true} if this CompletableFuture was cancelled
     * before it completed normally.
     *
     * @return {@code true} if this CompletableFuture was cancelled
     * before it completed normally
     */
    public boolean isCancelled() {
        Object r;
        return ((r = result) instanceof AltResult) &&
            (((AltResult)r).ex instanceof CancellationException);
    }

    /**
     * Returns {@code true} if this CompletableFuture completed
     * exceptionally, in any way. Possible causes include
     * cancellation, explicit invocation of {@code
     * completeExceptionally}, and abrupt termination of a
     * CompletionStage action.
     *
     * @return {@code true} if this CompletableFuture completed
     * exceptionally
     */
    public boolean isCompletedExceptionally() {
        Object r;
        return ((r = result) instanceof AltResult) && r != NIL;
    }

    /**
     * Forcibly sets or resets the value subsequently returned by
     * method {@link #get()} and related methods, whether or not
     * already completed. This method is designed for use only in
     * error recovery actions, and even in such situations may result
     * in ongoing dependent completions using established versus
     * overwritten outcomes.
     *
     * @param value the completion value
     */
    public void obtrudeValue(T value) {
        result = (value == null) ? NIL : value;
        postComplete();
    }

    /**
     * Forcibly causes subsequent invocations of method {@link #get()}
     * and related methods to throw the given exception, whether or
     * not already completed. This method is designed for use only in
     * error recovery actions, and even in such situations may result
     * in ongoing dependent completions using established versus
     * overwritten outcomes.
     *
     * @param ex the exception
     * @throws NullPointerException if the exception is null
     */
    public void obtrudeException(Throwable ex) {
        if (ex == null) throw new NullPointerException();
        result = new AltResult(ex);
        postComplete();
    }

    /**
     * Returns the estimated number of CompletableFutures whose
     * completions are awaiting completion of this CompletableFuture.
     * This method is designed for use in monitoring system state, not
     * for synchronization control.
     *
     * @return the number of dependent CompletableFutures
     */
    public int getNumberOfDependents() {
        int count = 0;
        for (Completion p = stack; p != null; p = p.next)
            ++count;
        return count;
    }

    /**
     * Returns a string identifying this CompletableFuture, as well as
     * its completion state.  The state, in brackets, contains the
     * String {@code "Completed Normally"} or the String {@code
     * "Completed Exceptionally"}, or the String {@code "Not
     * completed"} followed by the number of CompletableFutures
     * dependent upon its completion, if any.
     *
     * @return a string identifying this CompletableFuture, as well as its state
     */
    public String toString() {
        Object r = result;
        int count;
        return super.toString() +
            ((r == null) ?
             (((count = getNumberOfDependents()) == 0) ?
              "[Not completed]" :
              "[Not completed, " + count + " dependents]") :
             (((r instanceof AltResult) && ((AltResult)r).ex != null) ?
                "[Completed exceptionally]" :
                "[Completed normally]"));
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    private static final long RESULT;
    private static final long STACK;
    private static final long NEXT;
    static {
        try {
            final sun.misc.Unsafe u;
            UNSAFE = u = sun.misc.Unsafe.getUnsafe();
            Class<?> k = CompletableFuture.class;
            RESULT = u.objectFieldOffset(k.getDeclaredField("result"));
            STACK = u.objectFieldOffset(k.getDeclaredField("stack"));
            NEXT = u.objectFieldOffset
                (Completion.class.getDeclaredField("next"));
        } catch (Exception x) {
            throw new Error(x);
        }
    }
}
```

# Java常用线程池体系

1. Executor：线程池顶级接口；
2. ExecutorService：线程池次级接口，对Executor做了一些扩展，增加了一些功能；
3. ScheduledExecutorService：对ExecutorService做了一些扩展，增加了一些定时任务相关的功能；
4. AbstractExecutorService：抽象类，运用模板方法设计模式实现了一部分方法；
5. ThreadPoolExecutor：普通线程池类，包含最基本的一些线程池操作相关的方法实现；
6. ScheduledThreadPoolExecutor：定时任务线程池类，用于实现定时任务相关功能；
7. ForkJoinPool：新型线程池类，Java7中新增的线程池类，基于工作窃取理论实现，运用于大任务拆小任务，任务无限多的场景；
8. Executors：线程池工具类，定义了一些快速实现线程池的方法；



## Executor

```java
package java.util.concurrent;

public interface Executor {

    /**
     * 执行无返回值任务
     * 根据Executor的实现判断，可能是在新线程、线程池、线程调用中执行
     */
    void execute(Runnable command);
}
```

## ExecutorService

```java
package java.util.concurrent;
import java.util.List;
import java.util.Collection;

public interface ExecutorService extends Executor {

    // 关闭线程池，不再接受新任务，但已经提交的任务会执行完成
    void shutdown();

    /**
     * 立即关闭线程池，尝试停止正在运行的任务，未执行的任务不再执行
     * 被迫停止及未执行的任务将以列表的形式返回
     */
    List<Runnable> shutdownNow();

    // 检查线程池是否已关闭
    boolean isShutdown();

    // 检查线程池是否已终止，只有在shutdown()或shutdownNow()之后调用才有可能为true
    boolean isTerminated();

    // 在指定时间内线程池达到终止状态了才会返回true
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;

    // 执行有返回值的任务，任务的返回值为task.call()的结果
    <T> Future<T> submit(Callable<T> task);

    /**
     * 执行有返回值的任务，任务的返回值为这里传入的result
     * 当然只有当任务执行完成了调用get()时才返回
     */
    <T> Future<T> submit(Runnable task, T result);

    /**
     * Submits a Runnable task for execution and returns a Future
     * representing that task. The Future's {@code get} method will
     * return {@code null} upon <em>successful</em> completion.
     *
     * @param task the task to submit
     * @return a Future representing pending completion of the task
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     * @throws NullPointerException if the task is null
     */
    Future<?> submit(Runnable task);

    // 批量执行任务，只有当这些任务都完成了这个方法才返回
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

    /**
     * 在指定时间内批量执行任务，未执行完成的任务将被取消
     * 这里的timeout是所有任务的总时间，不是单个任务的时间
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                  long timeout, TimeUnit unit)
        throws InterruptedException;

    // 返回任意一个已完成任务的执行结果，未执行完成的任务将被取消
    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

    // 在指定时间内如果有任务已完成，则返回任意一个已完成任务的执行结果，未执行完成的任务将被取消
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

## AbstractExecutorService

```
package java.util.concurrent;
import java.util.*;

public abstract class AbstractExecutorService implements ExecutorService {

    protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
        return new FutureTask<T>(runnable, value);
    }

    protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        return new FutureTask<T>(callable);
    }

    public Future<?> submit(Runnable task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<Void> ftask = newTaskFor(task, null);
        execute(ftask);
        return ftask;
    }

    public <T> Future<T> submit(Runnable task, T result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task, result);
        execute(ftask);
        return ftask;
    }

    public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);
        execute(ftask);
        return ftask;
    }

```

```java
    private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,
                              boolean timed, long nanos)
        throws InterruptedException, ExecutionException, TimeoutException {
        if (tasks == null)
            throw new NullPointerException();
        int ntasks = tasks.size();
        if (ntasks == 0)
            throw new IllegalArgumentException();
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(ntasks);
        ExecutorCompletionService<T> ecs =
            new ExecutorCompletionService<T>(this);

        try {
            // Record exceptions so that if we fail to obtain any
            // result, we can throw the last exception we got.
            ExecutionException ee = null;
            final long deadline = timed ? System.nanoTime() + nanos : 0L;
            Iterator<? extends Callable<T>> it = tasks.iterator();

            // Start one task for sure; the rest incrementally
            futures.add(ecs.submit(it.next()));
            --ntasks;
            int active = 1;

            for (;;) {
                Future<T> f = ecs.poll();
                if (f == null) {
                    if (ntasks > 0) {
                        --ntasks;
                        futures.add(ecs.submit(it.next()));
                        ++active;
                    }
                    else if (active == 0)
                        break;
                    else if (timed) {
                        f = ecs.poll(nanos, TimeUnit.NANOSECONDS);
                        if (f == null)
                            throw new TimeoutException();
                        nanos = deadline - System.nanoTime();
                    }
                    else
                        f = ecs.take();
                }
                if (f != null) {
                    --active;
                    try {
                        return f.get();
                    } catch (ExecutionException eex) {
                        ee = eex;
                    } catch (RuntimeException rex) {
                        ee = new ExecutionException(rex);
                    }
                }
            }

            if (ee == null)
                ee = new ExecutionException();
            throw ee;

        } finally {
            for (int i = 0, size = futures.size(); i < size; i++)
                futures.get(i).cancel(true);
        }
    }

    public <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException {
        try {
            return doInvokeAny(tasks, false, 0);
        } catch (TimeoutException cannotHappen) {
            assert false;
            return null;
        }
    }

    public <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                           long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        return doInvokeAny(tasks, true, unit.toNanos(timeout));
    }

    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException {
        if (tasks == null)
            throw new NullPointerException();
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
        boolean done = false;
        try {
            for (Callable<T> t : tasks) {
                RunnableFuture<T> f = newTaskFor(t);
                futures.add(f);
                execute(f);
            }
            for (int i = 0, size = futures.size(); i < size; i++) {
                Future<T> f = futures.get(i);
                if (!f.isDone()) {
                    try {
                        f.get();
                    } catch (CancellationException ignore) {
                    } catch (ExecutionException ignore) {
                    }
                }
            }
            done = true;
            return futures;
        } finally {
```

```java
                if (!done)
                    for (int i = 0, size = futures.size(); i < size; i++)
                        futures.get(i).cancel(true);
            }
        }

        public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                             long timeout, TimeUnit unit)
            throws InterruptedException {
            if (tasks == null)
                throw new NullPointerException();
            long nanos = unit.toNanos(timeout);
            ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
            boolean done = false;
            try {
                for (Callable<T> t : tasks)
                    futures.add(newTaskFor(t));

                final long deadline = System.nanoTime() + nanos;
                final int size = futures.size();

                // Interleave time checks and calls to execute in case
                // executor doesn't have any/much parallelism.
                for (int i = 0; i < size; i++) {
                    execute((Runnable)futures.get(i));
                    nanos = deadline - System.nanoTime();
                    if (nanos <= 0L)
                        return futures;
                }

                for (int i = 0; i < size; i++) {
                    Future<T> f = futures.get(i);
                    if (!f.isDone()) {
                        if (nanos <= 0L)
                            return futures;
                        try {
                            f.get(nanos, TimeUnit.NANOSECONDS);
                        } catch (CancellationException ignore) {
                        } catch (ExecutionException ignore) {
                        } catch (TimeoutException toe) {
                            return futures;
                        }
                        nanos = deadline - System.nanoTime();
                    }
                }
                done = true;
                return futures;
            } finally {
                if (!done)
                    for (int i = 0, size = futures.size(); i < size; i++)
                        futures.get(i).cancel(true);
            }
        }
```

```
}
```

## ThreadPoolExecutor

```
package java.util.concurrent;

import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.*;

public class ThreadPoolExecutor extends AbstractExecutorService {

    // ctl贯穿在线程池的整个生命周期中。主要作用是用来保存线程数量和线程池的状态。一个int数值是4个字节32个bit位，这里采用高3位来保存运行状态，低29位来保存线程数量。
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    // 32-3 =29
    private static final int COUNT_BITS = Integer.SIZE - 3;
    // CAPACITY是个常量，值为：00011111 11111111 11111111 11111111
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

    // 运行状态保存在int值的高3位(所有数值左移29位)
    private static final int RUNNING    = -1 << COUNT_BITS; //接收新任务,并执行队列中的任务
    private static final int SHUTDOWN   =  0 << COUNT_BITS; //不接收外部新任务,但是执行队列中的任务
    private static final int STOP       =  1 << COUNT_BITS; //不接收外部新任务,也不执行队列中的任务,中断正在执行中的任务
    private static final int TIDYING    =  2 << COUNT_BITS;//所有的任务都已结束,线程数量为 0,处于该状态的线程池即将调用 terminated()方法
    private static final int TERMINATED =  3 << COUNT_BITS;//terminated()方法执行完成

    // 读取状态，先将CAPACITY进行非(~)运算，得到结果 11100000 00000000 00000000 00000000
    // 再通过&（按位与）运算，保留高3位，把低29位全部变为0;
    private static int runStateOf(int c)     { return c & ~CAPACITY; }
    // 读取数量，可以把高3位变为0，低29位保留。
    private static int workerCountOf(int c)  { return c & CAPACITY; }
    private static int ctlOf(int rs, int wc) { return rs | wc; }

    /*
     * Bit field accessors that don't require unpacking ctl.
     * These depend on the bit layout and on workerCount being never negative.
     */

    private static boolean runStateLessThan(int c, int s) {
        return c < s;
    }
    private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
    }

    private static boolean isRunning(int c) {
        return c < SHUTDOWN;
    }

    /**
     * Attempts to CAS-increment the workerCount field of ctl.
     */
    private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
    }

    /**
     * Attempts to CAS-decrement the workerCount field of ctl.
     */
    private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
    }

    /**
     * Decrements the workerCount field of ctl. This is called only on
     * abrupt termination of a thread (see processWorkerExit). Other
     * decrements are performed within getTask.
     */
    private void decrementWorkerCount() {
        do {} while (! compareAndDecrementWorkerCount(ctl.get()));
    }

    /**
     * The queue used for holding tasks and handing off to worker
     * threads.  We do not require that workQueue.poll() returning
     * null necessarily means that workQueue.isEmpty(), so rely
     * solely on isEmpty to see if the queue is empty (which we must
     * do for example when deciding whether to transition from
     * SHUTDOWN to TIDYING).  This accommodates special-purpose
     * queues such as DelayQueues for which poll() is allowed to
     * return null even if it may later return non-null when delays
     * expire.
     */
    private final BlockingQueue<Runnable> workQueue;

    /**
     * Lock held on access to workers set and related bookkeeping.
     * While we could use a concurrent set of some sort, it turns out
     * to be generally preferable to use a lock. Among the reasons is
     * that this serializes interruptIdleWorkers, which avoids
     * unnecessary interrupt storms, especially during shutdown.
     * Otherwise exiting threads would concurrently interrupt those
     * that have not yet interrupted. It also simplifies some of the
     * associated statistics bookkeeping of largestPoolSize etc. We
     * also hold mainLock on shutdown and shutdownNow, for the sake of
     * ensuring workers set is stable while separately checking
     * permission to interrupt and actually interrupting.
     */
    private final ReentrantLock mainLock = new ReentrantLock();

    /**
     * Set containing all worker threads in pool. Accessed only when
     * holding mainLock.
     */
    private final HashSet<Worker> workers = new HashSet<Worker>();

    /**
     * Wait condition to support awaitTermination
     */
    private final Condition termination = mainLock.newCondition();

    /**
     * Tracks largest attained pool size. Accessed only under
     * mainLock.
     */
    private int largestPoolSize;

    /**
     * Counter for completed tasks. Updated only on termination of
     * worker threads. Accessed only under mainLock.
     */
    private long completedTaskCount;

    /*
     * All user control parameters are declared as volatiles so that
     * ongoing actions are based on freshest values, but without need
     * for locking, since no internal invariants depend on them
     * changing synchronously with respect to other actions.
     */

    /**
     * Factory for new threads. All threads are created using this
     * factory (via method addWorker).  All callers must be prepared
     * for addWorker to fail, which may reflect a system or user's
     * policy limiting the number of threads.  Even though it is not
     * treated as an error, failure to create threads may result in
     * new tasks being rejected or existing ones remaining stuck in
     * the queue.
     *
     * We go further and preserve pool invariants even in the face of
     * errors such as OutOfMemoryError, that might be thrown while
     * trying to create threads.  Such errors are rather common due to
     * the need to allocate a native stack in Thread.start, and users
     * will want to perform clean pool shutdown to clean up.  There
     * will likely be enough memory available for the cleanup code to
     * complete without encountering yet another OutOfMemoryError.
     */
    private volatile ThreadFactory threadFactory;

    /**
     * Handler called when saturated or shutdown in execute.
     */
    private volatile RejectedExecutionHandler handler;

    /**
     * 等待工作的空闲线程的超时时间（以纳秒为单位）
     * 以下两部分线程将使用此超时时间:
     *   1) 当allowCoreThreadTimeOut为false时，超过corePoolSize部分的线程
     *   2) 当allowCoreThreadTimeOut为true时，所有的线程
     * 否则它们将永远等待新工作
     */
    private volatile long keepAliveTime;

    /**
     * 如果为false（默认），核心线程即使在空闲时也保持活动状态.
     * 如果为 true，核心线程使用 keepAliveTime 超时等待工作.
     */
    private volatile boolean allowCoreThreadTimeOut;

    /**
     * 核心线程数
     * 使用完成后不回收(占用OS资源)
     */
    private volatile int corePoolSize;

    /**
     * 最大线程数
     */
    private volatile int maximumPoolSize;

    /**
     * 任务队列已经满了并且没有线程可以执行时，对新来的任务的处理策略
     */
    private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();

    /**
     * Permission required for callers of shutdown and shutdownNow.
     * We additionally require (see checkShutdownAccess) that callers
     * have permission to actually interrupt threads in the worker set
     * (as governed by Thread.interrupt, which relies on
     * ThreadGroup.checkAccess, which in turn relies on
     * SecurityManager.checkAccess). Shutdowns are attempted only if
     * these checks pass.
     *
     * All actual invocations of Thread.interrupt (see
     * interruptIdleWorkers and interruptWorkers) ignore
     * SecurityExceptions, meaning that the attempted interrupts
     * silently fail. In the case of shutdown, they should not fail
     * unless the SecurityManager has inconsistent policies, sometimes
     * allowing access to a thread and sometimes not. In such cases,
     * failure to actually interrupt threads may disable or delay full
     * termination. Other uses of interruptIdleWorkers are advisory,
     * and failure to actually interrupt will merely delay response to
     * configuration changes so is not handled exceptionally.
     */
    private static final RuntimePermission shutdownPerm =
        new RuntimePermission("modifyThread");

    /* The context to be used when executing the finalizer, or null. */
    private final AccessControlContext acc;


    private final class Worker
        extends AbstractQueuedSynchronizer
        implements Runnable
    {
        /**
         * This class will never be serialized, but we provide a
         * serialVersionUID to suppress a javac warning.
         */
        private static final long serialVersionUID = 6138294804551838833L;

        // 注意了，这才是真正执行task的线程，从构造函数可知是由ThreadFactury创建的
        final Thread thread;
        // 这就是需要执行的task
        Runnable firstTask;
        // 完成的任务数，用于线程池统计
        volatile long completedTasks;

        /**
         * Creates with given first task and thread from ThreadFactory.
         * @param firstTask the first task (null if none)
         */
        Worker(Runnable firstTask) {
            // 初始状态-1，防止在调用runWorker（），也就是真正执行task前中断thread
            setState(-1);
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this);
        }

        /** Delegates main run loop to outer runWorker  */
        public void run() {
            runWorker(this);
        }

        // Lock methods
        //
        // The value 0 represents the unlocked state.
        // The value 1 represents the locked state.

        protected boolean isHeldExclusively() {
            return getState() != 0;
        }

        protected boolean tryAcquire(int unused) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }

        public void lock()        { acquire(1); }
        public boolean tryLock()  { return tryAcquire(1); }
        public void unlock()      { release(1); }
        public boolean isLocked() { return isHeldExclusively(); }

        void interruptIfStarted() {
            Thread t;
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                }
            }
        }
    }

    /*
     * Methods for setting control state
     */

    /**
     * Transitions runState to given target, or leaves it alone if
     * already at least the given target.
     *
     * @param targetState the desired state, either SHUTDOWN or STOP
     *        (but not TIDYING or TERMINATED -- use tryTerminate for that)
     */
    private void advanceRunState(int targetState) {
        for (;;) {
            int c = ctl.get();
            if (runStateAtLeast(c, targetState) ||
                ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
                break;
        }
    }

    /**
```

```
     * 根据线程池状态进行判断是否结束线程池
     */
    final void tryTerminate() {
        for (;;) {
            int c = ctl.get();
            /*
             * 当前线程池的状态为以下几种情况时，直接返回：
             * 1. RUNNING，因为还在运行中，不能停止；
             * 2. TIDYING或TERMINATED，因为线程池中已经没有正在运行的线程了；
             * 3. SHUTDOWN并且等待队列非空，这时要执行完workQueue中的task；
             */
            if (isRunning(c) ||
                runStateAtLeast(c, TIDYING) ||
                (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
                return;
            // 如果线程数量不为0，则中断一个空闲的工作线程，并返回
            if (workerCountOf(c) != 0) { // Eligible to terminate
                // interruptIdleWorkers(ONLY_ONE)的作用：
                // shutdown方法中，会中断所有空闲的工作线程，但如果在执行shutdown时工作线程没有空闲，那么这里就必须接着去调用interruptIdleWorkers方法中断线程，如果工作线程数大于零，那么线程池就一直无法关闭
                // 这里只中断一个空闲的工作线程的原因：在interruptIdleWorkers中会调用interrupt将空闲线程从runWorker的getTask中唤醒，空闲线程被唤醒后会执行processWorkerExit方法，在processWorkerExit方法中又会调用tryTerminate，所以这里只中断一个线程即可。被中断的线程又会出继续中断线程池中其他的没有中断的线程，直到workerCountOf(c)===0之后，就会执行下面的逻辑将ctl的状态设置为TERMINATED。
                interruptIdleWorkers(ONLY_ONE);
                return;
            }

            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // 这里尝试设置状态为TIDYING，如果设置成功，则调用terminated方法
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        // terminated方法默认什么都不做，留给子类实现
                        terminated();
                    } finally {
                        // 设置状态为TERMINATED
                        ctl.set(ctlOf(TERMINATED, 0));
                        // 唤醒调用所有等待线程池终止的线程 awaitTermination
                        termination.signalAll();
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }

    /*
     * Methods for controlling interrupts to worker threads.
     */

    /**
     * If there is a security manager, makes sure caller has
     * permission to shut down threads in general (see shutdownPerm).
     * If this passes, additionally makes sure the caller is allowed
     * to interrupt each worker thread. This might not be true even if
     * first check passed, if the SecurityManager treats some threads
     * specially.
     */
    private void checkShutdownAccess() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkPermission(shutdownPerm);
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                for (Worker w : workers)
                    security.checkAccess(w.thread);
            } finally {
                mainLock.unlock();
            }
        }
    }

    /**
     * Interrupts all threads, even if active. Ignores SecurityExceptions
     * (in which case some threads may remain uninterrupted).
     */
    private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                w.interruptIfStarted();
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * 中断空闲线程
     */
    private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers) {
                Thread t = w.thread;
                // 判断线程是否为空闲线程是通过AQS中的state是否为0来实现的
                // 如果线程的中断标志位为false并且获取锁成功，则中断线程
                // 通过锁来判断该线程是不是在运行中，因为在runWorker方法中会先加锁，然后再运行任务
                if (!t.isInterrupted() && w.tryLock()) {
```

```
                try {
                    // 会使线程从runWorker的getTask中被唤醒
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}

/**
 * Common form of interruptIdleWorkers, to avoid having to
 * remember what the boolean argument means.
 */
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}

private static final boolean ONLY_ONE = true;

/*
 * Misc utilities, most of which are also exported to
 * ScheduledThreadPoolExecutor
 */

/**
 * Invokes the rejected execution handler for the given command.
 * Package-protected for use by ScheduledThreadPoolExecutor.
 */
final void reject(Runnable command) {
    handler.rejectedExecution(command, this);
}

/**
 * Performs any further cleanup following run state transition on
 * invocation of shutdown.  A no-op here, but used by
 * ScheduledThreadPoolExecutor to cancel delayed tasks.
 */
void onShutdown() {
}

/**
 * State check needed by ScheduledThreadPoolExecutor to
 * enable running tasks during shutdown.
 *
 * @param shutdownOK true if should return true if SHUTDOWN
 */
```

```
    final boolean isRunningOrShutdown(boolean shutdownOK) {
        int rs = runStateOf(ctl.get());
        return rs == RUNNING || (rs == SHUTDOWN && shutdownOK);
    }

    /**
     * Drains the task queue into a new list, normally using
     * drainTo. But if the queue is a DelayQueue or any other kind of
     * queue for which poll or drainTo may fail to remove some
     * elements, it deletes them one by one.
     */
    private List<Runnable> drainQueue() {
        BlockingQueue<Runnable> q = workQueue;
        ArrayList<Runnable> taskList = new ArrayList<Runnable>();
        q.drainTo(taskList);
        if (!q.isEmpty()) {
            for (Runnable r : q.toArray(new Runnable[0])) {
                if (q.remove(r))
                    taskList.add(r);
            }
        }
        return taskList;
    }

    private boolean addWorker(Runnable firstTask, boolean core) {
        retry: //goto 语句,避免死循环
        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);

            /*
             * 下面这个条件只有在execute方法中的第二个判断成立时才有可能
             * 当核心线程数已满，队列未满时补充新的线程到线程池中
             */
            // 线程池已经shutdown后，还要添加新的任务，拒绝
            // （第二个判断）SHUTDOWN状态不接受新任务，但仍然会执行已经加入任务队列的任务,所以当进入
SHUTDOWN状态，而传进来的任务为空，并且任务队列不为空的时候，是允许添加新线程的(补充新的线程：execute方法中
的第二个条件),如果把这个条件取反，就表示不允许添加worker
            // firstTask == nul表示不可接受外部任务
            // workQueue.isEmpty()表示需要执行内部队列的任务
            if (rs >= SHUTDOWN &&
                ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty()))
                return false;

            for (;;) {
                // 获得Worker工作线程数
                int wc = workerCountOf(c);
                // 如果工作线程数大于默认容量大小或者大于核心线程数大小，则直接返回false表示不能再添加
worker。
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
                    return false;
                // 通过cas来增加工作线程数，如果cas失败，则直接重试
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                // 再次获取ctl
                c = ctl.get();  // Re-read ctl
                // 这里如果不相等，说明线程的状态发生了变化，继续重试
                if (runStateOf(c) != rs)
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop
            }
        }

        // 上面的代码主要是对worker数量做原子+1操作,下面的逻辑才是正式构建一个worker
        boolean workerStarted = false;//工作线程是否启动的标识
        boolean workerAdded = false;//工作线程是否已经添加成功的标识
        Worker w = null;
        try {
            // 构建一个 Worker，可以看到构造方法里面传入了一个Runnable对象
            w = new Worker(firstTask);
            // 从worker对象中取出线程
            final Thread t = w.thread;
            if (t != null) {
                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());

                    // 只有当前线程池是正在运行状态，[或是SHUTDOWN且firstTask为空]，才能添加到
workers集合中
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        // 任务刚封装到work ，还没start,你封装的线程就是alive，那肯定是有问题的要抛异
常出去的

                        if (t.isAlive()) // precheck that t is startable
                            throw new IllegalThreadStateException();
                        // 将新创建的Worker添加到workers集合中
                        workers.add(w);
                        int s = workers.size();
                        // 如果集合中的工作线程数大于最大线程数，这个最大线程数表示线程池曾经出现过的最大
线程数

                        if (s > largestPoolSize)
                            largestPoolSize = s; // 更新线程池出现过的最大线程数
                        workerAdded = true; // 表示工作线程创建成功了
                    }
                } finally {
                    mainLock.unlock();
                }
                // 如果worker添加成功
                if (workerAdded) {
                    // 启动线程
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }

    /**
     * Rolls back the worker thread creation.
     * - removes worker from workers, if present
     * - decrements worker count
     * - rechecks for termination, in case the existence of this
     *   worker was holding up termination
     */
    private void addWorkerFailed(Worker w) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            if (w != null)
                workers.remove(w);
            decrementWorkerCount();
            tryTerminate();
        } finally {
            mainLock.unlock();
        }
    }

    private void processWorkerExit(Worker w, boolean completedAbruptly) {

        // 如果是因为用户的异常导致worker的中止，则将workercount减一
        if (completedAbruptly)
            decrementWorkerCount();

        // 将completedTaskCount进行增加，表示总共完成的任务数，并且从WorkerSet中将对应的Worker移除
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            completedTaskCount += w.completedTasks;
            workers.remove(w);
        } finally {
            mainLock.unlock();
        }

        // 调用tryTemiate，进行判断当前的线程池是否处于SHUTDOWN状态，判断是否要终止线程
        tryTerminate();

        //判断当前的线程池状态，如果当前线程池状态比STOP大的话，就不处理
        int c = ctl.get();
```

```
        if (runStateLessThan(c, STOP)) {
            // 判断是否是意外退出，如果不是意外退出的话，那么就会判断最少要保留的核心线程数，如果
allowCoreThreadTimeOut被设置为true的话，那么说明核心线程在设置的KeepAliveTime之后，也会被销毁。
            if (!completedAbruptly) {
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                // 如果最少保留的Worker数为0的话，那么就会判断当前的任务队列是否为空，如果任务队列不为空
的话而且线程池没有停止，那么说明至少还需要1个线程继续将任务完成。
                if (min == 0 && ! workQueue.isEmpty())
                    min = 1;
                // 判断当前的Worker是否大于min，也就是说当前的Worker总数大于最少需要的Worker数的话，那
么就直接返回，因为剩下的Worker会继续从WorkQueue中获取任务执行。
                if (workerCountOf(c) >= min)
                    return; // replacement not needed
            }
            // 如果当前运行的Worker数比当前所需要的Worker数少的话，那么就会调用addWorker，添加新的
Worker，也就是新开启线程继续处理任务。
            addWorker(null, false);
        }
    }

    /**
     * Performs blocking or timed wait for a task, depending on
     * current configuration settings, or returns null if this worker
     * must exit because of any of:
     * 1. There are more than maximumPoolSize workers (due to
     *    a call to setMaximumPoolSize).
     * 2. The pool is stopped.
     * 3. The pool is shutdown and the queue is empty.
     * 4. This worker timed out waiting for a task, and timed-out
     *    workers are subject to termination (that is,
     *    {@code allowCoreThreadTimeOut || workerCount > corePoolSize})
     *    both before and after the timed wait, and if the queue is
     *    non-empty, this worker is not the last thread in the pool.
     *
     * @return task, or null if the worker must exit, in which case
     *         workerCount is decremented
     */
    private Runnable getTask() {
        boolean timedOut = false; // Did the last poll() time out?

        for (;;) {
            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);

            // Are workers subject to culling?
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

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

            try {
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }

    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        // 表示当前worker线程允许中断，因为 new Worker默认的state=-1,此处是调用Worker类的
tryRelease()方法，将state置为0，而interruptIfStarted()中只有state>=0才允许调用中断
        w.unlock(); // allow interrupts
        // 当用户重写的beforeExecute方法和afterExecute方法出现异常时，由于代码中没有处理异常，所以
while循环里的completedAbruptly=false这句代码时走不到的，所以调用processWorkerExit方法时
completedAbruptly是等于true的
        boolean completedAbruptly = true;
        try {
            // 注意这个while循环,在这里实现了[线程复用]
            // 如果task为空，则通过getTask(从workQueue)来获取任务,如果队列中没有任务则阻塞等待
            while (task != null || (task = getTask()) != null) {
                // 上锁，不是为了防止并发执行任务，为了在shutdown()时不终止正在运行的worker
                // shutdown中会掉用interruptIdleWorkers，而interruptIdleWorkers在进行中断时会使用
tryLock来判断该工作线程是否正在处理任务，如果tryLock返回true，说明该工作线程当前未执行任务，这时才可以被中
断。如果当前线程阻塞在getTask方法中，而getTask方法中没有mainLock，
                w.lock();
                // 如果线程池状态是stop并且当前线程是没有设置中断位的，则给当前线程设置中断位。
                // 如果线程池的状态不是stop，使用Thread.interrupted()判断当前线程的中断状态，如果是
true，再继续判断线程池的状态是否为stop如果是stop,并且当前线程是没有设置中断位，则给当前线程设置中断。
                // 根据shutdownNow方法的定义，线程池是STOP状态就要设置中断位。因为是多线程环境，主线程
正在把线程池的状态设置为stop，此时工作线程也在给自己设置中断位。当指令重排序的时候可能会使得线程池的状态还没有
设置完，工作线程会先设置了中断位。因此就出现了runWorker里面的操作，如果中断位的设置早于stop的设置，就把中段
位清理掉，重新设置。
                //设置中断位不是说立即把线程正在执行的任务也停了，只是设置中断位由工作线程自行去处理，所以
正常状态下线程还是会把最后一个任务执行完毕后再回到while循环的判断条件getTask中，由于此时状态为STOP所以直接
worker数量减一后返回null结束while循环，再在执行完processWorkerExit后结束线程。
                if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    // beforeExecute：钩子函数，留给子类去重写
                    // 这里默认是没有实现的，在一些特定的场景中可以自己继承ThreadpoolExecutor自己重写
                    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 {
                    // 置空任务(这样下次循环开始时,task依然为null,需要再通过getTask()取)
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            // 有两种情况会终止while循环
            // 当上面while中有业务异常抛出（由于代码中try之后只有finally没有catch所有异常还是会往外抛，
所以while循环会中止然后进入外层try的finally里面）
            // 线程池中止的时候
            processWorkerExit(w, completedAbruptly);
        }
    }

    // Public constructors and methods

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default thread factory and rejected execution handler.
     * It may be more convenient to use one of the {@link Executors} factory
     * methods instead of this general purpose constructor.
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
```

```java
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default rejected execution handler.
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @param threadFactory the factory to use when the executor
     *        creates a new thread
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue}
     *         or {@code threadFactory} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             threadFactory, defaultHandler);
    }
    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default thread factory.
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @param handler the handler to use when execution is blocked
     *        because the thread bounds and queue capacities are reached
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue}
     *         or {@code handler} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              RejectedExecutionHandler handler) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), handler);
    }

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters.
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @param threadFactory the factory to use when the executor
     *        creates a new thread
     * @param handler the handler to use when execution is blocked
     *        because the thread bounds and queue capacities are reached
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue}
     *         or {@code threadFactory} or {@code handler} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory,
                              RejectedExecutionHandler handler) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
                null :
                AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }

    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();

        int c = ctl.get();
        // 当前池中线程比核心数少，新建一个线程执行任务
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        // 核心池已满，但任务队列未满，添加到队列中
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            // 任务成功添加到队列以后，再次检查是否需要添加新的线程，因为已存在的线程可能被销毁了
            if (! isRunning(recheck) && remove(command))
                // 如果线程池处于非运行状态，并且把当前的任务从任务队列中移除成功，则拒绝该任务
                reject(command);
            // 因为设置了allowCoreThreadTimeOut为true，那么之前的线程可能已被销毁完，所以这里新建一个
线程
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 核心池已满，队列已满，试着创建一个新线程（当没有达到最大线程数时创建成功）
        else if (!addWorker(command, false))
            // 如果创建新线程失败了，说明线程池被关闭或者线程池完全满了，拒绝任务
            reject(command);
    }

    /**
     * Initiates an orderly shutdown in which previously submitted
     * tasks are executed, but no new tasks will be accepted.
     * Invocation has no additional effect if already shut down.
     *
     * <p>This method does not wait for previously submitted tasks to
     * complete execution.  Use {@link #awaitTermination awaitTermination}
     * to do that.
     *
     * @throws SecurityException {@inheritDoc}
     */
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 安全策略判断(无视即可)
            checkShutdownAccess();
            // SHUTDOWN:不接受新任务，但是执行queue任务
            advanceRunState(SHUTDOWN);
            // 中断空闲线程
            interruptIdleWorkers();
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            mainLock.unlock();
        }
        // 尝试结束线程池
        tryTerminate();
    }

    /**
     * Attempts to stop all actively executing tasks, halts the
     * processing of waiting tasks, and returns a list of the tasks
     * that were awaiting execution. These tasks are drained (removed)
     * from the task queue upon return from this method.
     *
     * <p>This method does not wait for actively executing tasks to
     * terminate.  Use {@link #awaitTermination awaitTermination} to
     * do that.
     *
     * <p>There are no guarantees beyond best-effort attempts to stop
     * processing actively executing tasks.  This implementation
     * cancels tasks via {@link Thread#interrupt}, so any task that
     * fails to respond to interrupts may never terminate.
     *
     * @throws SecurityException {@inheritDoc}
     */
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 安全策略判断(无视即可)
            checkShutdownAccess();
            // 不接受新任务，也不执行queue任务
            advanceRunState(STOP);
            // 中断所有线程
            interruptWorkers();
            // 取出队列中的任务
            tasks = drainQueue();
        } finally {
            mainLock.unlock();
        }
        // 尝试结束线程池
        tryTerminate();
        return tasks;
    }

    public boolean isShutdown() {
        return ! isRunning(ctl.get());
    }

    /**
     * Returns true if this executor is in the process of terminating
     * after {@link #shutdown} or {@link #shutdownNow} but has not
     * completely terminated.  This method may be useful for
     * debugging. A return of {@code true} reported a sufficient
     * period after shutdown may indicate that submitted tasks have
     * ignored or suppressed interruption, causing this executor not
     * to properly terminate.
     *
     * @return {@code true} if terminating but not yet terminated
     */
    public boolean isTerminating() {
        int c = ctl.get();
        return ! isRunning(c) && runStateLessThan(c, TERMINATED);
    }

    public boolean isTerminated() {
        return runStateAtLeast(ctl.get(), TERMINATED);
    }

    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (;;) {
                if (runStateAtLeast(ctl.get(), TERMINATED))
                    return true;
                if (nanos <= 0)
                    return false;
                nanos = termination.awaitNanos(nanos);
            }
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Invokes {@code shutdown} when this executor is no longer
     * referenced and it has no threads.
     */
    protected void finalize() {
        SecurityManager sm = System.getSecurityManager();
        if (sm == null || acc == null) {
            shutdown();
        } else {
            PrivilegedAction<Void> pa = () -> { shutdown(); return null; };
            AccessController.doPrivileged(pa, acc);
        }
    }

    /**
     * Sets the thread factory used to create new threads.
     *
     * @param threadFactory the new thread factory
     * @throws NullPointerException if threadFactory is null
     * @see #getThreadFactory
     */
    public void setThreadFactory(ThreadFactory threadFactory) {
        if (threadFactory == null)
            throw new NullPointerException();
        this.threadFactory = threadFactory;
    }

    /**
     * Returns the thread factory used to create new threads.
     *
     * @return the current thread factory
     * @see #setThreadFactory(ThreadFactory)
     */
    public ThreadFactory getThreadFactory() {
        return threadFactory;
    }

    /**
     * Sets a new handler for unexecutable tasks.
     *
     * @param handler the new handler
     * @throws NullPointerException if handler is null
     * @see #getRejectedExecutionHandler
     */
    public void setRejectedExecutionHandler(RejectedExecutionHandler handler) {
        if (handler == null)
            throw new NullPointerException();
        this.handler = handler;
    }

    /**
     * Returns the current handler for unexecutable tasks.
     *
     * @return the current handler
     * @see #setRejectedExecutionHandler(RejectedExecutionHandler)
     */
    public RejectedExecutionHandler getRejectedExecutionHandler() {
        return handler;
    }

    /**
     * Sets the core number of threads.  This overrides any value set
     * in the constructor.  If the new value is smaller than the
     * current value, excess existing threads will be terminated when
     * they next become idle.  If larger, new threads will, if needed,
     * be started to execute any queued tasks.
     *
     * @param corePoolSize the new core size
     * @throws IllegalArgumentException if {@code corePoolSize < 0}
     * @see #getCorePoolSize
     */
    public void setCorePoolSize(int corePoolSize) {
        if (corePoolSize < 0)
            throw new IllegalArgumentException();
        int delta = corePoolSize - this.corePoolSize;
        this.corePoolSize = corePoolSize;
        if (workerCountOf(ctl.get()) > corePoolSize)
            interruptIdleWorkers();
        else if (delta > 0) {
            // We don't really know how many new threads are "needed".
            // As a heuristic, prestart enough new workers (up to new
            // core size) to handle the current number of tasks in
            // queue, but stop if queue becomes empty while doing so.
            int k = Math.min(delta, workQueue.size());
            while (k-- > 0 && addWorker(null, true)) {
                if (workQueue.isEmpty())
                    break;
            }
        }
    }

    /**
```

```
     * Returns the core number of threads.
     *
     * @return the core number of threads
     * @see #setCorePoolSize
     */
    public int getCorePoolSize() {
        return corePoolSize;
    }

    /**
     * Starts a core thread, causing it to idly wait for work. This
     * overrides the default policy of starting core threads only when
     * new tasks are executed. This method will return {@code false}
     * if all core threads have already been started.
     *
     * @return {@code true} if a thread was started
     */
    public boolean prestartCoreThread() {
        return workerCountOf(ctl.get()) < corePoolSize &&
            addWorker(null, true);
    }

    /**
     * Same as prestartCoreThread except arranges that at least one
     * thread is started even if corePoolSize is 0.
     */
    void ensurePrestart() {
        int wc = workerCountOf(ctl.get());
        if (wc < corePoolSize)
            addWorker(null, true);
        else if (wc == 0)
            addWorker(null, false);
    }

    /**
     * Starts all core threads, causing them to idly wait for work. This
     * overrides the default policy of starting core threads only when
     * new tasks are executed.
     *
     * @return the number of threads started
     */
    public int prestartAllCoreThreads() {
        int n = 0;
        while (addWorker(null, true))
            ++n;
        return n;
    }

    /**
     * Returns true if this pool allows core threads to time out and
     * terminate if no tasks arrive within the keepAlive time, being
     * replaced if needed when new tasks arrive. When true, the same
     * keep-alive policy applying to non-core threads applies also to
     * core threads. When false (the default), core threads are never
     * terminated due to lack of incoming tasks.
     *
     * @return {@code true} if core threads are allowed to time out,
     *         else {@code false}
     *
     * @since 1.6
     */
    public boolean allowsCoreThreadTimeOut() {
        return allowCoreThreadTimeOut;
    }

    /**
     * Sets the policy governing whether core threads may time out and
     * terminate if no tasks arrive within the keep-alive time, being
     * replaced if needed when new tasks arrive. When false, core
     * threads are never terminated due to lack of incoming
     * tasks. When true, the same keep-alive policy applying to
     * non-core threads applies also to core threads. To avoid
     * continual thread replacement, the keep-alive time must be
     * greater than zero when setting {@code true}. This method
     * should in general be called before the pool is actively used.
     *
     * @param value {@code true} if should time out, else {@code false}
     * @throws IllegalArgumentException if value is {@code true}
     *         and the current keep-alive time is not greater than zero
     *
     * @since 1.6
     */
    public void allowCoreThreadTimeOut(boolean value) {
        if (value && keepAliveTime <= 0)
            throw new IllegalArgumentException("Core threads must have nonzero keep alive
times");
        if (value != allowCoreThreadTimeOut) {
            allowCoreThreadTimeOut = value;
            if (value)
                interruptIdleWorkers();
        }
    }

    /**
     * Sets the maximum allowed number of threads. This overrides any
     * value set in the constructor. If the new value is smaller than
     * the current value, excess existing threads will be
     * terminated when they next become idle.
     *
     * @param maximumPoolSize the new maximum
     * @throws IllegalArgumentException if the new maximum is
     *         less than or equal to zero, or
     *         less than the {@linkplain #getCorePoolSize core pool size}
     * @see #getMaximumPoolSize
     */
    public void setMaximumPoolSize(int maximumPoolSize) {
        if (maximumPoolSize <= 0 || maximumPoolSize < corePoolSize)
            throw new IllegalArgumentException();
        this.maximumPoolSize = maximumPoolSize;
        if (workerCountOf(ctl.get()) > maximumPoolSize)
            interruptIdleWorkers();
    }

    /**
     * Returns the maximum allowed number of threads.
     *
     * @return the maximum allowed number of threads
     * @see #setMaximumPoolSize
     */
    public int getMaximumPoolSize() {
        return maximumPoolSize;
    }

    /**
     * Sets the time limit for which threads may remain idle before
     * being terminated.  If there are more than the core number of
     * threads currently in the pool, after waiting this amount of
     * time without processing a task, excess threads will be
     * terminated.  This overrides any value set in the constructor.
     *
     * @param time the time to wait.  A time value of zero will cause
     *        excess threads to terminate immediately after executing tasks.
     * @param unit the time unit of the {@code time} argument
     * @throws IllegalArgumentException if {@code time} less than zero or
     *         if {@code time} is zero and {@code allowsCoreThreadTimeOut}
     * @see #getKeepAliveTime(TimeUnit)
     */
    public void setKeepAliveTime(long time, TimeUnit unit) {
        if (time < 0)
            throw new IllegalArgumentException();
        if (time == 0 && allowsCoreThreadTimeOut())
            throw new IllegalArgumentException("Core threads must have nonzero keep alive
times");
        long keepAliveTime = unit.toNanos(time);
        long delta = keepAliveTime - this.keepAliveTime;
        this.keepAliveTime = keepAliveTime;
        if (delta < 0)
            interruptIdleWorkers();
    }

    /**
     * Returns the thread keep-alive time, which is the amount of time
     * that threads in excess of the core pool size may remain
     * idle before being terminated.
     *
     * @param unit the desired time unit of the result
     * @return the time limit
     * @see #setKeepAliveTime(long, TimeUnit)
     */
    public long getKeepAliveTime(TimeUnit unit) {
        return unit.convert(keepAliveTime, TimeUnit.NANOSECONDS);
    }

    /* User-level queue utilities */

    /**
     * Returns the task queue used by this executor. Access to the
     * task queue is intended primarily for debugging and monitoring.
     * This queue may be in active use.  Retrieving the task queue
     * does not prevent queued tasks from executing.
     *
     * @return the task queue
     */
    public BlockingQueue<Runnable> getQueue() {
        return workQueue;
    }

    /**
     * Removes this task from the executor's internal queue if it is
     * present, thus causing it not to be run if it has not already
     * started.
     *
     * <p>This method may be useful as one part of a cancellation
     * scheme.  It may fail to remove tasks that have been converted
     * into other forms before being placed on the internal queue. For
     * example, a task entered using {@code submit} might be
     * converted into a form that maintains {@code Future} status.
     * However, in such cases, method {@link #purge} may be used to
     * remove those Futures that have been cancelled.
     *
     * @param task the task to remove
     * @return {@code true} if the task was removed
     */
    public boolean remove(Runnable task) {
        boolean removed = workQueue.remove(task);
        tryTerminate(); // In case SHUTDOWN and now empty
        return removed;
    }

    /**
     * Tries to remove from the work queue all {@link Future}
     * tasks that have been cancelled. This method can be useful as a
     * storage reclamation operation, that has no other impact on
     * functionality. Cancelled tasks are never executed, but may
     * accumulate in work queues until worker threads can actively
     * remove them. Invoking this method instead tries to remove them now.
     * However, this method may fail to remove tasks in
     * the presence of interference by other threads.
     */
    public void purge() {
        final BlockingQueue<Runnable> q = workQueue;
        try {
```

```java
            Iterator<Runnable> it = q.iterator();
            while (it.hasNext()) {
                Runnable r = it.next();
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled())
                    it.remove();
            }
        } catch (ConcurrentModificationException fallThrough) {
            // Take slow path if we encounter interference during traversal.
            // Make copy for traversal and call remove for cancelled entries.
            // The slow path is more likely to be O(N*N).
            for (Object r : q.toArray())
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled())
                    q.remove(r);
        }

        tryTerminate(); // In case SHUTDOWN and now empty
    }

    /* Statistics */

    /**
     * Returns the current number of threads in the pool.
     *
     * @return the number of threads
     */
    public int getPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // Remove rare and surprising possibility of
            // isTerminated() && getPoolSize() > 0
            return runStateAtLeast(ctl.get(), TIDYING) ? 0
                : workers.size();
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate number of threads that are actively
     * executing tasks.
     *
     * @return the number of threads
     */
    public int getActiveCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            int n = 0;
            for (Worker w : workers)
                if (w.isLocked())
                    ++n;
            return n;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the largest number of threads that have ever
     * simultaneously been in the pool.
     *
     * @return the number of threads
     */
    public int getLargestPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            return largestPoolSize;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate total number of tasks that have ever been
     * scheduled for execution. Because the states of tasks and
     * threads may change dynamically during computation, the returned
     * value is only an approximation.
     *
     * @return the number of tasks
     */
    public long getTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            long n = completedTaskCount;
            for (Worker w : workers) {
                n += w.completedTasks;
                if (w.isLocked())
                    ++n;
            }
            return n + workQueue.size();
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate total number of tasks that have
     * completed execution. Because the states of tasks and threads
     * may change dynamically during computation, the returned value
     * is only an approximation, but one that does not ever decrease
     * across successive calls.
     *
     * @return the number of tasks
     */
    public long getCompletedTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            long n = completedTaskCount;
            for (Worker w : workers)
                n += w.completedTasks;
            return n;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns a string identifying this pool, as well as its state,
     * including indications of run state and estimated worker and
     * task counts.
     *
     * @return a string identifying this pool, as well as its state
     */
    public String toString() {
        long ncompleted;
        int nworkers, nactive;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            ncompleted = completedTaskCount;
            nactive = 0;
            nworkers = workers.size();
            for (Worker w : workers) {
                ncompleted += w.completedTasks;
                if (w.isLocked())
                    ++nactive;
            }
        } finally {
            mainLock.unlock();
        }
        int c = ctl.get();
        String rs = (runStateLessThan(c, SHUTDOWN) ? "Running" :
                     (runStateAtLeast(c, TERMINATED) ? "Terminated" :
                      "Shutting down"));
        return super.toString() +
            "[" + rs +
            ", pool size = " + nworkers +
            ", active threads = " + nactive +
            ", queued tasks = " + workQueue.size() +
            ", completed tasks = " + ncompleted +
            "]";
    }

    /* Extension hooks */
    /**
     * Method invoked prior to executing the given Runnable in the
     * given thread.  This method is invoked by thread {@code t} that
     * will execute task {@code r}, and may be used to re-initialize
     * ThreadLocals, or to perform logging.
     *
     * <p>This implementation does nothing, but may be customized in
     * subclasses. Note: To properly nest multiple overridings, subclasses
     * should generally invoke {@code super.beforeExecute} at the end of
     * this method.
     *
     * @param t the thread that will run task {@code r}
     * @param r the task that will be executed
     */
    protected void beforeExecute(Thread t, Runnable r) { }

    /**
     * Method invoked upon completion of execution of the given Runnable.
     * This method is invoked by the thread that executed the task. If
     * non-null, the Throwable is the uncaught {@code RuntimeException}
     * or {@code Error} that caused execution to terminate abruptly.
     *
     * <p>This implementation does nothing, but may be customized in
     * subclasses. Note: To properly nest multiple overridings, subclasses
     * should generally invoke {@code super.afterExecute} at the
     * beginning of this method.
     *
     * <p><b>Note:</b> When actions are enclosed in tasks (such as
     * {@link FutureTask}) either explicitly or via methods such as
     * {@code submit}, these task objects catch and maintain
     * computational exceptions, and so they do not cause abrupt
     * termination, and the internal exceptions are <em>not</em>
     * passed to this method. If you would like to trap both kinds of
     * failures in this method, you can further probe for such cases,
     * as in this sample subclass that prints either the direct cause
     * or the underlying exception if a task has been aborted:
     *
     *  <pre> {@code
     * class ExtendedExecutor extends ThreadPoolExecutor {
     *   // ...
     *   protected void afterExecute(Runnable r, Throwable t) {
     *     super.afterExecute(r, t);
     *     if (t == null && r instanceof Future<?>) {
     *       try {
     *         Object result = ((Future<?>) r).get();
     *       } catch (CancellationException ce) {
     *           t = ce;
     *       } catch (ExecutionException ee) {
     *           t = ee.getCause();
     *       } catch (InterruptedException ie) {
     *           Thread.currentThread().interrupt(); // ignore/reset
     *       }
     *     }
     *     if (t != null)
     *       System.out.println(t);
     *   }
     * }}</pre>
     *
     * @param r the runnable that has completed
     * @param t the exception that caused termination, or null if
     * execution completed normally
     */
    protected void afterExecute(Runnable r, Throwable t) { }

    /**
     * Method invoked when the Executor has terminated.  Default
     * implementation does nothing. Note: To properly nest multiple
     * overridings, subclasses should generally invoke
     * {@code super.terminated} within this method.
     */
    protected void terminated() { }

    /* Predefined RejectedExecutionHandlers */

    /**
     * A handler for rejected tasks that runs the rejected task
     * directly in the calling thread of the {@code execute} method,
     * unless the executor has been shut down, in which case the task
     * is discarded.
     */
    public static class CallerRunsPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code CallerRunsPolicy}.
         */
        public CallerRunsPolicy() { }

        /**
         * Executes task r in the caller's thread, unless the executor
         * has been shut down, in which case the task is discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }

    /**
     * A handler for rejected tasks that throws a
     * {@code RejectedExecutionException}.
     */
    public static class AbortPolicy implements RejectedExecutionHandler {
        /**
```

```
         * Creates an {@code AbortPolicy}.
         */
        public AbortPolicy() { }

        /**
         * Always throws RejectedExecutionException.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         * @throws RejectedExecutionException always
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException("Task " + r.toString() +
                                                 " rejected from " +
                                                 e.toString());
        }
    }

    /**
     * A handler for rejected tasks that silently discards the
     * rejected task.
     */
    public static class DiscardPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardPolicy}.
         */
        public DiscardPolicy() { }

        /**
         * Does nothing, which has the effect of discarding task r.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
    }

    /**
     * A handler for rejected tasks that discards the oldest unhandled
     * request and then retries {@code execute}, unless the executor
     * is shut down, in which case the task is discarded.
     */
    public static class DiscardOldestPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardOldestPolicy} for the given executor.
         */
        public DiscardOldestPolicy() { }

        /**
         * Obtains and ignores the next task that the executor
         * would otherwise execute, if one is immediately available,
         * and then retries execution of task r, unless the executor
         * is shut down, in which case task r is instead discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);
            }
        }
    }
}
```

ThreadPool的5中状态:
RUNNING:接收新任务和进程队列任务
SHUTDOWN:不接受新任务，但是接收进程队列任务(即：还接受已经在队列中的任务)
STOP:不接受新任务也不接受进程队列任务，并且打断正在进行中的任务
TIDYING:所有任务终止，待处理任务数量为0，线程转换为TIDYING，将会执行terminated钩子函数
TERMINATED:terminated()执行完成

状态之间的转换:
RUNNING -> SHUTDOWN:调用shutdown()方法
(RUNNING or SHUTDOWN) -> STOP:调用shutdownNow()方法
SHUTDOWN -> TIDYING:队列和线程池都是空的
STOP -> TIDYING:线程池为空
TIDYING -> TERMINATED:钩子函数terminated()执行完成

4中丢弃策略:
CallerRunsPolicy：由主线程去执行被拒绝的任务（比：线程池核心线程数设置为2，最大线程数也设置为2，任务队列的大小设置为3，当提交第6个任务的时候会触发拒绝策略，而如果此时配置了CallerRunsPolicy策略的话，主线程会直接执行第六个任务）
AbortPolicy：抛RejectedExecutionException异常，（默认的处理方式）
DiscardPolicy：对拒绝任务直接无声抛弃，没有异常信息
DiscardOldestPolicy：对拒绝任务不抛弃，而是抛弃队列里面等待最久的一个线程，然后把拒绝任务加到队列。

## ScheduledThreadPoolExecutor

继承自ThreadPoolExecutor，主要用来在给定的延迟之后运行任务，或者定期执行任务。ScheduledThreadPoolExecutor继承ThreadPoolExecutor来重用线程池的功能，实现了ScheduledExecutorService接口，该接口定义了schedule等任务调度的方法。它的主要实现方式如下:

1. 将任务封装成ScheduledFutureTask对象，ScheduledFutureTask基于相对时间，不受系统时间的改变所影响；
2. ScheduledFutureTask实现了java.lang.Comparable接口和java.util.concurrent.Delayed接口，所以有两个重要的方法：compareTo和getDelay。compareTo方法用于比较任务之间的优先级关系，如果距离下次执行的时间间隔较短，则优先级高；getDelay方法用于返回距离下次任务执行时间的时间间隔；
3. ScheduledThreadPoolExecutor定义了一个DelayedWorkQueue，它是一个有序队列，会通过每个任务按照距离下次执行时间间隔的大小来排序；
4. ScheduledFutureTask继承自FutureTask，可以通过返回Future对象来获取执行的结果。

```java
package java.util.concurrent;
import static java.util.concurrent.TimeUnit.NANOSECONDS;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
import java.util.*;

public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService {

    /*
     * This class specializes ThreadPoolExecutor implementation by
     *
     * 1. Using a custom task type, ScheduledFutureTask for
     *    tasks, even those that don't require scheduling (i.e.,
     *    those submitted using ExecutorService execute, not
     *    ScheduledExecutorService methods) which are treated as
     *    delayed tasks with a delay of zero.
     *
     * 2. Using a custom queue (DelayedWorkQueue), a variant of
     *    unbounded DelayQueue. The lack of capacity constraint and
     *    the fact that corePoolSize and maximumPoolSize are
     *    effectively identical simplifies some execution mechanics
     *    (see delayedExecute) compared to ThreadPoolExecutor.
     *
     * 3. Supporting optional run-after-shutdown parameters, which
     *    leads to overrides of shutdown methods to remove and cancel
     *    tasks that should NOT be run after shutdown, as well as
     *    different recheck logic when task (re)submission overlaps
     *    with a shutdown.
     *
     * 4. Task decoration methods to allow interception and
     *    instrumentation, which are needed because subclasses cannot
     *    otherwise override submit methods to get this effect. These
     *    don't have any impact on pool control logic though.
     */

    /**
     * 表示在池关闭之后是否继续执行已经存在的周期任务
     */
    private volatile boolean continueExistingPeriodicTasksAfterShutdown;

    /**
     * 表示在池关闭之后是否继续执行已经存在的非周期任务
     */
    private volatile boolean executeExistingDelayedTasksAfterShutdown = true;

    /**
     * True if ScheduledFutureTask.cancel should remove from queue
     */
    private volatile boolean removeOnCancel = false;
    /**
     * Sequence number to break scheduling ties, and in turn to
     * guarantee FIFO order among tied entries.
     */
    private static final AtomicLong sequencer = new AtomicLong();

    /**
     * Returns current nanosecond time.
     */
    final long now() {
        return System.nanoTime();
    }

    private class ScheduledFutureTask<V>
            extends FutureTask<V> implements RunnableScheduledFuture<V> {

        /** Sequence number to break ties FIFO */
        private final long sequenceNumber;

        /** The time the task is enabled to execute in nanoTime units */
        private long time;

        /**
         * Period in nanoseconds for repeating tasks.  A positive
         * value indicates fixed-rate execution.  A negative value
         * indicates fixed-delay execution.  A value of 0 indicates a
         * non-repeating task.
         */
        private final long period;

        /** The actual task to be re-enqueued by reExecutePeriodic */
        RunnableScheduledFuture<V> outerTask = this;

        /**
         * Index into delay queue, to support faster cancellation.
         */
        int heapIndex;

        /**
         * Creates a one-shot action with given nanoTime-based trigger time.
         */
        ScheduledFutureTask(Runnable r, V result, long ns) {
            super(r, result);
            this.time = ns;
            this.period = 0;
            this.sequenceNumber = sequencer.getAndIncrement();
        }

        /**
         * Creates a periodic action with given nano time and period.
         */
        ScheduledFutureTask(Runnable r, V result, long ns, long period) {
            super(r, result);
            this.time = ns;
            this.period = period;
            this.sequenceNumber = sequencer.getAndIncrement();
        }

        /**
         * Creates a one-shot action with given nanoTime-based trigger time.
         */
        ScheduledFutureTask(Callable<V> callable, long ns) {
            super(callable);
            this.time = ns;
            this.period = 0;
            this.sequenceNumber = sequencer.getAndIncrement();
        }

        public long getDelay(TimeUnit unit) {
            // 执行时间减去当前系统时间
            return unit.convert(time - now(), NANOSECONDS);
        }

        public int compareTo(Delayed other) {
            if (other == this) // compare zero if same object
                return 0;
            if (other instanceof ScheduledFutureTask) {
                ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
                long diff = time - x.time;
                if (diff < 0)
                    return -1;
                else if (diff > 0)
                    return 1;
                else if (sequenceNumber < x.sequenceNumber)
                    return -1;
                else
                    return 1;
            }
            long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECONDS);
            return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;
        }

        /**
         * 判断是否为周期性任务
         */
        public boolean isPeriodic() {
            return period != 0;
        }

        /**
         * scheduleAtFixedRate和scheduleWithFixedDelay的区别在setNextRunTime方法中就可以看出来
         * setNextRunTime方法会在run方法中执行完任务后调用。
         */
        private void setNextRunTime() {
            long p = period;
            // 固定频率，上次执行时间加上周期时间
```

```
            if (p > 0)
                time += p;
            // 相对固定延迟执行，使用当前系统时间加上周期时间
            else
                time = triggerTime(-p);
        }

        public boolean cancel(boolean mayInterruptIfRunning) {
            boolean cancelled = super.cancel(mayInterruptIfRunning);
            if (cancelled && removeOnCancel && heapIndex >= 0)
                remove(this);
            return cancelled;
        }

        /**
         * Overrides FutureTask version so as to reset/requeue if periodic.
         */
        public void run() {
            // 是否是周期性任务
            boolean periodic = isPeriodic();
            // 当前线程池运行状态下如果不可以执行任务，取消该任务
            if (!canRunInCurrentRunState(periodic))
                cancel(false);
            // 如果不是周期性任务，调用FutureTask中的run方法执行
            else if (!periodic)
                ScheduledFutureTask.super.run();
            // 如果是周期性任务，调用FutureTask中的runAndReset方法执行
            // runAndReset方法不会设置执行结果，所以可以重复执行任务
            else if (ScheduledFutureTask.super.runAndReset()) {
                // 计算下次执行该任务的时间
                setNextRunTime();
                // 重复执行任务
                reExecutePeriodic(outerTask);
            }
        }
    }

    /**
     * 判断调用线程池shutdown()方法后是否应该执行此任务。
     * 可以通过setContinueExistingPeriodicTasksAfterShutdownPolicy方法设置在线程池关闭时，周期任务
继续执行，默认为false，也就是线程池关闭时，不再执行周期任务
     * isRunningOrShutdown方法中通过目前线程池的状态以及出入的参数shutdownOK来决定是否运行次任务
     * 如果线程池的状态是RUNNING则不看shutdownOK直接返回true，
     * 如果线程池的状态是SHUTDOWN那么返回shutdownOK的值
     */
    boolean canRunInCurrentRunState(boolean periodic) {
        return isRunningOrShutdown(periodic ?
                                   continueExistingPeriodicTasksAfterShutdown :
                                   executeExistingDelayedTasksAfterShutdown);
    }

    /**
     * Main execution method for delayed or periodic tasks.  If pool
     * is shut down, rejects the task. Otherwise adds task to queue
     * and starts a thread, if necessary, to run it.  (We cannot
     * prestart the thread to run the task because the task (probably)
     * shouldn't be run yet.)  If the pool is shut down while the task
     * is being added, cancel and remove it if required by state and
     * run-after-shutdown parameters.
     *
     * @param task the task
     */
    private void delayedExecute(RunnableScheduledFuture<?> task) {
        // 如果线程池已经关闭，使用拒绝策略拒绝任务
        if (isShutdown())
            reject(task);
        else {
            // 添加到阻塞队列中
            super.getQueue().add(task);
            // 通过canRunInCurrentRunState方法判断在线程池关闭时，周期任务是否继续执行
            if (isShutdown() &&
                !canRunInCurrentRunState(task.isPeriodic()) &&
                remove(task))
                task.cancel(false);
            else
                // 确保线程池中至少有一个线程启动
                // 该方法在ThreadPoolExecutor中实现
                ensurePrestart();
        }
    }

    /**
     * Requeues a periodic task unless current run state precludes it.
     * Same idea as delayedExecute except drops task rather than rejecting.
     *
     * @param task the task
     */
    void reExecutePeriodic(RunnableScheduledFuture<?> task) {
        if (canRunInCurrentRunState(true)) {
            super.getQueue().add(task);
            if (!canRunInCurrentRunState(true) && remove(task))
                task.cancel(false);
            else
                ensurePrestart();
        }
    }

    /**
     * Cancels and clears the queue of all tasks that should not be run
     * due to shutdown policy.  Invoked within super.shutdown.
     */
    @Override void onShutdown() {
        BlockingQueue<Runnable> q = super.getQueue();
        boolean keepDelayed =
            getExecuteExistingDelayedTasksAfterShutdownPolicy();
        boolean keepPeriodic =
```

```java
            getContinueExistingPeriodicTasksAfterShutdownPolicy();
        if (!keepDelayed && !keepPeriodic) {
            for (Object e : q.toArray())
                if (e instanceof RunnableScheduledFuture<?>)
                    ((RunnableScheduledFuture<?>) e).cancel(false);
            q.clear();
        }
        else {
            // Traverse snapshot to avoid iterator exceptions
            for (Object e : q.toArray()) {
                if (e instanceof RunnableScheduledFuture) {
                    RunnableScheduledFuture<?> t =
                        (RunnableScheduledFuture<?>)e;
                    if ((t.isPeriodic() ? !keepPeriodic : !keepDelayed) ||
                        t.isCancelled()) { // also remove if already cancelled
                        if (q.remove(t))
                            t.cancel(false);
                    }
                }
            }
        }
        tryTerminate();
    }

    /**
     * 修改或替换用于执行runnable的任务。此方法可重写用于管理内部任务的具体类。默认实现只返回给定任务。
     */
    protected <V> RunnableScheduledFuture<V> decorateTask(
        Runnable runnable, RunnableScheduledFuture<V> task) {
        return task;
    }

    /**
     * 修改或替换用于执行callable的任务。此方法可重写用于管理内部任务的具体类。默认实现只返回给定任务。
     */
    protected <V> RunnableScheduledFuture<V> decorateTask(
        Callable<V> callable, RunnableScheduledFuture<V> task) {
        return task;
    }

    /**
     * 因为ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，所以这里都是调用的
ThreadPoolExecutor类的构造方法。注意传入的阻塞队列是DelayedWorkQueue类型的对象。
     */
    public ScheduledThreadPoolExecutor(int corePoolSize) {
        // 这里直接调用父类也就是ThreadPoolExecutor的构造方法，而在ThreadPoolExecutor中，只有当
workQueue为有限队列时候非核心线程数：maximumPoolSize才有效，
        // ScheduledThreadPoolExecutor里面的DelayedWorkQueue是自定义的队列，而且DelayedWorkQueue
的构造方法中没有限制队列大小的变量，所以这里的队列为无限队列，所以它的非核心线程数以及超时时间参数都是无效的。
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue());
    }
    /**
     * 因为ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，所以这里都是调用的ThreadPoolExecutor类的构造方法。注意传入的阻塞队列是DelayedWorkQueue类型的对象。
     */
    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       ThreadFactory threadFactory) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue(), threadFactory);
    }

    /**
     * 因为ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，所以这里都是调用的ThreadPoolExecutor类的构造方法。注意传入的阻塞队列是DelayedWorkQueue类型的对象。
     */
    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       RejectedExecutionHandler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue(), handler);
    }

    /**
     * 因为ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，所以这里都是调用的ThreadPoolExecutor类的构造方法。注意传入的阻塞队列是DelayedWorkQueue类型的对象。
     */
    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       ThreadFactory threadFactory,
                                       RejectedExecutionHandler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue(), threadFactory, handler);
    }

    /**
     * 用于获取下一次执行的具体时间.
     */
    private long triggerTime(long delay, TimeUnit unit) {
        return triggerTime(unit.toNanos((delay < 0) ? 0 : delay));
    }

    /**
     * Returns the trigger time of a delayed action.
     */
    long triggerTime(long delay) {
        // 这里的delay < (Long.MAX_VALUE >> 1是为了判断是否要防止Long类型溢出，
        // 如果delay的值小于Long类型最大值的一半，则直接返回delay，否则需要进行防止溢出处理。
        return now() +
            ((delay < (Long.MAX_VALUE >> 1)) ? delay : overflowFree(delay));
    }

    /**
     * 限制队列中所有节点的延迟时间在Long.MAX_VALUE之内，防止在compareTo方法中溢出
     */
    private long overflowFree(long delay) {
        // 获取队列中的第一个节点
        Delayed head = (Delayed) super.getQueue().peek();
        if (head != null) {
            // 获取延迟时间
            long headDelay = head.getDelay(NANOSECONDS);
            // 如果延迟时间小于0，并且 delay - headDelay 超过了Long.MAX_VALUE
            // 将delay设置为 Long.MAX_VALUE + headDelay 保证delay小于Long.MAX_VALUE
            if (headDelay < 0 && (delay - headDelay < 0))
                delay = Long.MAX_VALUE + headDelay;
        }
        return delay;
    }

    /**
     * 在delay时间后开始执行commod任务
     */
    public ScheduledFuture<?> schedule(Runnable command,
                                       long delay,
                                       TimeUnit unit) {
        if (command == null || unit == null)
            throw new NullPointerException();
        // 把传入的任务封装成一个RunnableScheduledFuture对象，其实也就是ScheduledFutureTask对象
        // decorateTask默认什么功能都没有做，子类可以重写该方法
        RunnableScheduledFuture<?> t = decorateTask(command,
            new ScheduledFutureTask<Void>(command, null,
                                          triggerTime(delay, unit)));
        // 通过调用delayedExecute方法来延时执行任务。
        delayedExecute(t);
        return t;
    }

    /**
     * 在delay时间后开始执行commod任务
     */
    public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                                           long delay,
                                           TimeUnit unit) {
        if (callable == null || unit == null)
            throw new NullPointerException();
        // 把传入的任务封装成一个RunnableScheduledFuture对象，其实也就是ScheduledFutureTask对象
        // decorateTask默认什么功能都没有做，子类可以重写该方法
        RunnableScheduledFuture<V> t = decorateTask(callable,
            new ScheduledFutureTask<V>(callable,
                                       triggerTime(delay, unit)));
        // 通过调用delayedExecute方法来延时执行任务。
        delayedExecute(t);
        return t;
    }

    /**
     * initialDelay:系统启动后，需要等待多久才开始执行。
     * period:为固定周期时间，按照一定频率来重复执行任务。
     * 如果period设置的是3秒，command执行要5秒；那么等上一次任务执行完就立即执行，也就是任务与任务之间的差
异是5s；
```

```
 * 如果period设置的是3s，command执行要2s；那么需要等到command执行完后再等1S后再次执行下一次任务。
 */
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    if (period <= 0)
        throw new IllegalArgumentException();
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
                                      null,
                                      triggerTime(initialDelay, unit),
                                      unit.toNanos(period));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    sft.outerTask = t;
    delayedExecute(t);
    return t;
}

/**
 * initialDelay:系统启动后，需要等待多久才开始执行
 * period:固定周期时间，按照一定频率来重复执行任务。
 * 这个方式必须等待上一个任务结束才开始计时period。
 * 如果设置的period为3s;任务执行耗时为5S那么下次任务执行时间为第8S。
 */
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                 long initialDelay,
                                                 long delay,
                                                 TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    if (delay <= 0)
        throw new IllegalArgumentException();
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
                                      null,
                                      triggerTime(initialDelay, unit),
                                      unit.toNanos(-delay));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    sft.outerTask = t;
    delayedExecute(t);
    return t;
}

/**
 * Executes {@code command} with zero required delay.
 * This has effect equivalent to
 * {@link #schedule(Runnable,long,TimeUnit) schedule(command, 0, anyUnit)}.
 * Note that inspections of the queue and of the list returned by
 * {@code shutdownNow} will access the zero-delayed
 * {@link ScheduledFuture}, not the {@code command} itself.
```

```java
     *
     * <p>A consequence of the use of {@code ScheduledFuture} objects is
     * that {@link ThreadPoolExecutor#afterExecute afterExecute} is always
     * called with a null second {@code Throwable} argument, even if the
     * {@code command} terminated abruptly.  Instead, the {@code Throwable}
     * thrown by such a task can be obtained via {@link Future#get}.
     *
     * @throws RejectedExecutionException at discretion of
     *         {@code RejectedExecutionHandler}, if the task
     *         cannot be accepted for execution because the
     *         executor has been shut down
     * @throws NullPointerException {@inheritDoc}
     */
    public void execute(Runnable command) {
        schedule(command, 0, NANOSECONDS);
    }

    // Override AbstractExecutorService methods

    /**
     * @throws RejectedExecutionException {@inheritDoc}
     * @throws NullPointerException       {@inheritDoc}
     */
    public Future<?> submit(Runnable task) {
        return schedule(task, 0, NANOSECONDS);
    }

    /**
     * @throws RejectedExecutionException {@inheritDoc}
     * @throws NullPointerException       {@inheritDoc}
     */
    public <T> Future<T> submit(Runnable task, T result) {
        return schedule(Executors.callable(task, result), 0, NANOSECONDS);
    }

    /**
     * @throws RejectedExecutionException {@inheritDoc}
     * @throws NullPointerException       {@inheritDoc}
     */
    public <T> Future<T> submit(Callable<T> task) {
        return schedule(task, 0, NANOSECONDS);
    }

    /**
     * Sets the policy on whether to continue executing existing
     * periodic tasks even when this executor has been {@code shutdown}.
     * In this case, these tasks will only terminate upon
     * {@code shutdownNow} or after setting the policy to
     * {@code false} when already shutdown.
     * This value is by default {@code false}.
     *
     * @param value if {@code true}, continue after shutdown, else don't
     * @see #getContinueExistingPeriodicTasksAfterShutdownPolicy
     */
    public void setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean value) {
        continueExistingPeriodicTasksAfterShutdown = value;
        if (!value && isShutdown())
            onShutdown();
    }

    /**
     * Gets the policy on whether to continue executing existing
     * periodic tasks even when this executor has been {@code shutdown}.
     * In this case, these tasks will only terminate upon
     * {@code shutdownNow} or after setting the policy to
     * {@code false} when already shutdown.
     * This value is by default {@code false}.
     *
     * @return {@code true} if will continue after shutdown
     * @see #setContinueExistingPeriodicTasksAfterShutdownPolicy
     */
    public boolean getContinueExistingPeriodicTasksAfterShutdownPolicy() {
        return continueExistingPeriodicTasksAfterShutdown;
    }

    /**
     * Sets the policy on whether to execute existing delayed
     * tasks even when this executor has been {@code shutdown}.
     * In this case, these tasks will only terminate upon
     * {@code shutdownNow}, or after setting the policy to
     * {@code false} when already shutdown.
     * This value is by default {@code true}.
     *
     * @param value if {@code true}, execute after shutdown, else don't
     * @see #getExecuteExistingDelayedTasksAfterShutdownPolicy
     */
    public void setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean value) {
        executeExistingDelayedTasksAfterShutdown = value;
        if (!value && isShutdown())
            onShutdown();
    }

    /**
     * Gets the policy on whether to execute existing delayed
     * tasks even when this executor has been {@code shutdown}.
     * In this case, these tasks will only terminate upon
     * {@code shutdownNow}, or after setting the policy to
     * {@code false} when already shutdown.
     * This value is by default {@code true}.
     *
     * @return {@code true} if will execute after shutdown
     * @see #setExecuteExistingDelayedTasksAfterShutdownPolicy
     */
    public boolean getExecuteExistingDelayedTasksAfterShutdownPolicy() {
        return executeExistingDelayedTasksAfterShutdown;
    }
    /**
     * Sets the policy on whether cancelled tasks should be immediately
     * removed from the work queue at time of cancellation.  This value is
     * by default {@code false}.
     *
     * @param value if {@code true}, remove on cancellation, else don't
     * @see #getRemoveOnCancelPolicy
     * @since 1.7
     */
    public void setRemoveOnCancelPolicy(boolean value) {
        removeOnCancel = value;
    }

    /**
     * Gets the policy on whether cancelled tasks should be immediately
     * removed from the work queue at time of cancellation.  This value is
     * by default {@code false}.
     *
     * @return {@code true} if cancelled tasks are immediately removed
     *         from the queue
     * @see #setRemoveOnCancelPolicy
     * @since 1.7
     */
    public boolean getRemoveOnCancelPolicy() {
        return removeOnCancel;
    }

    /**
     * Initiates an orderly shutdown in which previously submitted
     * tasks are executed, but no new tasks will be accepted.
     * Invocation has no additional effect if already shut down.
     *
     * <p>This method does not wait for previously submitted tasks to
     * complete execution.  Use {@link #awaitTermination awaitTermination}
     * to do that.
     *
     * <p>If the {@code ExecuteExistingDelayedTasksAfterShutdownPolicy}
     * has been set {@code false}, existing delayed tasks whose delays
     * have not yet elapsed are cancelled.  And unless the {@code
     * ContinueExistingPeriodicTasksAfterShutdownPolicy} has been set
     * {@code true}, future executions of existing periodic tasks will
     * be cancelled.
     *
     * @throws SecurityException {@inheritDoc}
     */
    public void shutdown() {
        super.shutdown();
    }

    /**
     * Attempts to stop all actively executing tasks, halts the
     * processing of waiting tasks, and returns a list of the tasks
     * that were awaiting execution.
     *
     * <p>This method does not wait for actively executing tasks to
     * terminate.  Use {@link #awaitTermination awaitTermination} to
     * do that.
     *
     * <p>There are no guarantees beyond best-effort attempts to stop
     * processing actively executing tasks.  This implementation
     * cancels tasks via {@link Thread#interrupt}, so any task that
     * fails to respond to interrupts may never terminate.
     *
     * @return list of tasks that never commenced execution.
     *         Each element of this list is a {@link ScheduledFuture},
     *         including those tasks submitted using {@code execute},
     *         which are for scheduling purposes used as the basis of a
     *         zero-delay {@code ScheduledFuture}.
     * @throws SecurityException {@inheritDoc}
     */
    public List<Runnable> shutdownNow() {
        return super.shutdownNow();
    }

    /**
     * Returns the task queue used by this executor.  Each element of
     * this queue is a {@link ScheduledFuture}, including those
     * tasks submitted using {@code execute} which are for scheduling
     * purposes used as the basis of a zero-delay
     * {@code ScheduledFuture}.  Iteration over this queue is
     * <em>not</em> guaranteed to traverse tasks in the order in
     * which they will execute.
     *
     * @return the task queue
     */
    public BlockingQueue<Runnable> getQueue() {
        return super.getQueue();
    }

    /**
     * Specialized delay queue. To mesh with TPE declarations, this
     * class must be declared as a BlockingQueue<Runnable> even though
     * it can only hold RunnableScheduledFutures.
     */
    static class DelayedWorkQueue extends AbstractQueue<Runnable>
        implements BlockingQueue<Runnable> {

        /*
         * A DelayedWorkQueue is based on a heap-based data structure
         * like those in DelayQueue and PriorityQueue, except that
         * every ScheduledFutureTask also records its index into the
         * heap array. This eliminates the need to find a task upon
         * cancellation, greatly speeding up removal (down from O(n)
         * to O(log n)), and reducing garbage retention that would
         * otherwise occur by waiting for the element to rise to top
         * before clearing. But because the queue may also hold
         * RunnableScheduledFutures that are not ScheduledFutureTasks,
         * we are not guaranteed to have such indices available, in
         * which case we fall back to linear search. (We expect that
         * most tasks will not be decorated, and that the faster cases
         * will be much more common.)
         *
         * All heap operations must record index changes -- mainly
         * within siftUp and siftDown. Upon removal, a task's
         * heapIndex is set to -1. Note that ScheduledFutureTasks can
         * appear at most once in the queue (this need not be true for
         * other kinds of tasks or work queues), so are uniquely
         * identified by heapIndex.
         */

        private static final int INITIAL_CAPACITY = 16;
        private RunnableScheduledFuture<?>[] queue =
            new RunnableScheduledFuture<?>[INITIAL_CAPACITY];
        private final ReentrantLock lock = new ReentrantLock();
        private int size = 0;

        /**
         * Thread designated to wait for the task at the head of the
         * queue.  This variant of the Leader-Follower pattern
         * (http://www.cs.wustl.edu/~schmidt/POSA/POSA2/) serves to
         * minimize unnecessary timed waiting.  When a thread becomes
         * the leader, it waits only for the next delay to elapse, but
         * other threads await indefinitely.  The leader thread must
         * signal some other thread before returning from take() or
         * poll(...), unless some other thread becomes leader in the
         * interim.  Whenever the head of the queue is replaced with a
         * task with an earlier expiration time, the leader field is
         * invalidated by being reset to null, and some waiting
         * thread, but not necessarily the current leader, is
         * signalled.  So waiting threads must be prepared to acquire
         * and lose leadership while waiting.
         */
        private Thread leader = null;

        /**
         * Condition signalled when a newer task becomes available at the
         * head of the queue or a new thread may need to become leader.
         */
        private final Condition available = lock.newCondition();

        /**
         * Sets f's heapIndex if it is a ScheduledFutureTask.
         */
        private void setIndex(RunnableScheduledFuture<?> f, int idx) {
            if (f instanceof ScheduledFutureTask)
                ((ScheduledFutureTask)f).heapIndex = idx;
        }
```

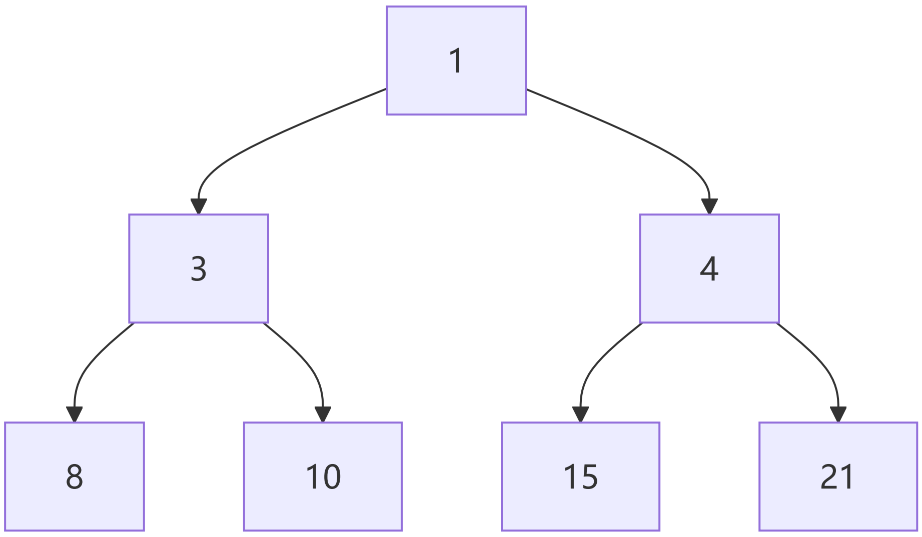
```java
        /**
         * Sifts element added at bottom up to its heap-ordered spot.
         * Call only when holding lock.
         */
        private void siftUp(int k, RunnableScheduledFuture<?> key) {
            while (k > 0) {
                int parent = (k - 1) >>> 1;
                RunnableScheduledFuture<?> e = queue[parent];
                if (key.compareTo(e) >= 0)
                    break;
                queue[k] = e;
                setIndex(e, k);
                k = parent;
            }
            queue[k] = key;
            setIndex(key, k);
        }

        /**
         * Sifts element added at top down to its heap-ordered spot.
         * Call only when holding lock.
         */
        private void siftDown(int k, RunnableScheduledFuture<?> key) {
            int half = size >>> 1;
            while (k < half) {
                int child = (k << 1) + 1;
                RunnableScheduledFuture<?> c = queue[child];
                int right = child + 1;
                if (right < size && c.compareTo(queue[right]) > 0)
                    c = queue[child = right];
                if (key.compareTo(c) <= 0)
                    break;
                queue[k] = c;
                setIndex(c, k);
                k = child;
            }
            queue[k] = key;
            setIndex(key, k);
        }

        /**
         * Resizes the heap array.  Call only when holding lock.
         */
        private void grow() {
            int oldCapacity = queue.length;
            int newCapacity = oldCapacity + (oldCapacity >> 1); // grow 50%
            if (newCapacity < 0) // overflow
                newCapacity = Integer.MAX_VALUE;
            queue = Arrays.copyOf(queue, newCapacity);
        }

        /**
         * Finds index of given object, or -1 if absent.
         */
        private int indexOf(Object x) {
            if (x != null) {
                if (x instanceof ScheduledFutureTask) {
                    int i = ((ScheduledFutureTask) x).heapIndex;
                    // Sanity check; x could conceivably be a
                    // ScheduledFutureTask from some other pool.
                    if (i >= 0 && i < size && queue[i] == x)
                        return i;
                } else {
                    for (int i = 0; i < size; i++)
                        if (x.equals(queue[i]))
                            return i;
                }
            }
            return -1;
        }

        public boolean contains(Object x) {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                return indexOf(x) != -1;
            } finally {
                lock.unlock();
            }
        }

        public boolean remove(Object x) {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                int i = indexOf(x);
                if (i < 0)
                    return false;

                setIndex(queue[i], -1);
                int s = --size;
                RunnableScheduledFuture<?> replacement = queue[s];
                queue[s] = null;
                if (s != i) {
                    siftDown(i, replacement);
                    if (queue[i] == replacement)
                        siftUp(i, replacement);
                }
                return true;
            } finally {
                lock.unlock();
            }
        }

        public int size() {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                return size;
            } finally {
                lock.unlock();
            }
        }

        public boolean isEmpty() {
            return size() == 0;
        }

        public int remainingCapacity() {
            return Integer.MAX_VALUE;
        }

        public RunnableScheduledFuture<?> peek() {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                return queue[0];
            } finally {
                lock.unlock();
            }
        }

        public boolean offer(Runnable x) {
            if (x == null)
                throw new NullPointerException();
            RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x;
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                int i = size;
                if (i >= queue.length)
                    grow();
                size = i + 1;
                if (i == 0) {
                    queue[0] = e;
                    setIndex(e, 0);
                } else {
                    siftUp(i, e);
                }
                if (queue[0] == e) {
                    leader = null;
                    available.signal();
                }
            } finally {
                lock.unlock();
            }
            return true;
        }
```

```
    public void put(Runnable e) {
        offer(e);
    }

    public boolean add(Runnable e) {
        return offer(e);
    }

    public boolean offer(Runnable e, long timeout, TimeUnit unit) {
        return offer(e);
    }

    /**
     * Performs common bookkeeping for poll and take: Replaces
     * first element with last and sifts it down.  Call only when
     * holding lock.
     * @param f the task to remove and return
     */
    private RunnableScheduledFuture<?> finishPoll(RunnableScheduledFuture<?> f) {
        int s = --size;
        RunnableScheduledFuture<?> x = queue[s];
        queue[s] = null;
        if (s != 0)
            siftDown(0, x);
        setIndex(f, -1);
        return f;
    }

    public RunnableScheduledFuture<?> poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            RunnableScheduledFuture<?> first = queue[0];
            if (first == null || first.getDelay(NANOSECONDS) > 0)
                return null;
            else
                return finishPoll(first);
        } finally {
            lock.unlock();
        }
    }

    public RunnableScheduledFuture<?> take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                RunnableScheduledFuture<?> first = queue[0];
                if (first == null)
                    available.await();
                else {
                    long delay = first.getDelay(NANOSECONDS);
                    if (delay <= 0)
                        return finishPoll(first);
                    first = null; // don't retain ref while waiting
                    if (leader != null)
                        available.await();
                    else {
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;
                        try {
                            available.awaitNanos(delay);
                        } finally {
                            if (leader == thisThread)
                                leader = null;
                        }
                    }
                }
            }
        } finally {
            if (leader == null && queue[0] != null)
                available.signal();
            lock.unlock();
        }
    }

    public RunnableScheduledFuture<?> poll(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                RunnableScheduledFuture<?> first = queue[0];
                if (first == null) {
                    if (nanos <= 0)
                        return null;
                    else
                        nanos = available.awaitNanos(nanos);
                } else {
                    long delay = first.getDelay(NANOSECONDS);
                    if (delay <= 0)
                        return finishPoll(first);
                    if (nanos <= 0)
                        return null;
                    first = null; // don't retain ref while waiting
                    if (nanos < delay || leader != null)
                        nanos = available.awaitNanos(nanos);
                    else {
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;
                        try {
                            long timeLeft = available.awaitNanos(delay);
                            nanos -= delay - timeLeft;
                        } finally {
                            if (leader == thisThread)
```

```java
                                leader = null;
                            }
                        }
                    }
                }
            } finally {
                if (leader == null && queue[0] != null)
                    available.signal();
                lock.unlock();
            }
        }

        public void clear() {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                for (int i = 0; i < size; i++) {
                    RunnableScheduledFuture<?> t = queue[i];
                    if (t != null) {
                        queue[i] = null;
                        setIndex(t, -1);
                    }
                }
                size = 0;
            } finally {
                lock.unlock();
            }
        }

        /**
         * Returns first element only if it is expired.
         * Used only by drainTo.  Call only when holding lock.
         */
        private RunnableScheduledFuture<?> peekExpired() {
            // assert lock.isHeldByCurrentThread();
            RunnableScheduledFuture<?> first = queue[0];
            return (first == null || first.getDelay(NANOSECONDS) > 0) ?
                null : first;
        }

        public int drainTo(Collection<? super Runnable> c) {
            if (c == null)
                throw new NullPointerException();
            if (c == this)
                throw new IllegalArgumentException();
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                RunnableScheduledFuture<?> first;
                int n = 0;
                while ((first = peekExpired()) != null) {
                    c.add(first);   // In this order, in case add() throws.
                    finishPoll(first);
                    ++n;
                }
                return n;
            } finally {
                lock.unlock();
            }
        }

        public int drainTo(Collection<? super Runnable> c, int maxElements) {
            if (c == null)
                throw new NullPointerException();
            if (c == this)
                throw new IllegalArgumentException();
            if (maxElements <= 0)
                return 0;
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                RunnableScheduledFuture<?> first;
                int n = 0;
                while (n < maxElements && (first = peekExpired()) != null) {
                    c.add(first);   // In this order, in case add() throws.
                    finishPoll(first);
                    ++n;
                }
                return n;
            } finally {
                lock.unlock();
            }
        }

        public Object[] toArray() {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                return Arrays.copyOf(queue, size, Object[].class);
            } finally {
                lock.unlock();
            }
        }

        @SuppressWarnings("unchecked")
        public <T> T[] toArray(T[] a) {
            final ReentrantLock lock = this.lock;
            lock.lock();
            try {
                if (a.length < size)
                    return (T[]) Arrays.copyOf(queue, size, a.getClass());
                System.arraycopy(queue, 0, a, 0, size);
                if (a.length > size)
                    a[size] = null;
                return a;
            } finally {
                lock.unlock();
            }
        }

        public Iterator<Runnable> iterator() {
            return new Itr(Arrays.copyOf(queue, size));
        }

        /**
         * Snapshot iterator that works off copy of underlying q array.
         */
        private class Itr implements Iterator<Runnable> {
            final RunnableScheduledFuture<?>[] array;
            int cursor = 0;     // index of next element to return
            int lastRet = -1;   // index of last element, or -1 if no such

            Itr(RunnableScheduledFuture<?>[] array) {
                this.array = array;
            }

            public boolean hasNext() {
                return cursor < array.length;
            }

            public Runnable next() {
                if (cursor >= array.length)
                    throw new NoSuchElementException();
                lastRet = cursor;
                return array[cursor++];
            }

            public void remove() {
                if (lastRet < 0)
                    throw new IllegalStateException();
                DelayedWorkQueue.this.remove(array[lastRet]);
                lastRet = -1;
            }
        }
    }
}
```

## DelayedWorkQueue

ScheduledThreadPoolExecutor之所以要自己实现阻塞的工作队列，是因为ScheduledThreadPoolExecutor要求的工作队列有些特殊。DelayedWorkQueue是一个基于堆的数据结构，类似于DelayQueue和PriorityQueue。在执行定时任务的时候，每个任务的执行时间都不同，所以DelayedWorkQueue的工作就是按照执行时间的升序来排列，执行时间距离当前时间越近的任务在队列的前面（注意：这里的顺序并不是绝对的，堆中的排序只保证了子节点的下次执行时间要比父节点的下次执行时间要大，而叶子节点之间并不一定是顺序的，下文中会说明）。堆结构如下图所示：



可见，DelayedWorkQueue是一个基于最小堆结构的队列。堆结构可以使用数组表示，可以转换成如下的数组：

| 1 | 3 | 4 | 8 | 10 | 15 | 21 |
|---|---|---|---|---|---|---|

在这种结构中，可以发现有如下特性：
假设，索引值从0开始，子节点的索引值为k，父节点的索引值为p，则：

1. 一个节点的左子节点的索引为：k = p * 2 + 1；
2. 一个节点的右子节点的索引为：k = (p + 1) * 2；
3. 一个节点的父节点的索引为：p = (k - 1) / 2。

为什么要使用DelayedWorkQueue呢？

定时任务执行时需要取出最近要执行的任务，所以任务在队列中每次出队时一定要是当前队列中执行时间最靠前的，所以自然要使用优先级队列。DelayedWorkQueue是一个优先级队列，它可以保证每次出队的任务都是当前队列中执行时间最靠前的，由于它是基于堆结构的队列，堆结构在执行插入和删除操作时的最坏时间复杂度是 O(logN)。

DelayedWorkQueue的属性

```
// 队列初始容量
private static final int INITIAL_CAPACITY = 16;
// 根据初始容量创建RunnableScheduledFuture类型的数组
private RunnableScheduledFuture<?>[] queue =
    new RunnableScheduledFuture<?>[INITIAL_CAPACITY];
private final ReentrantLock lock = new ReentrantLock();
private int size = 0;
// leader线程
private Thread leader = null;
// 当较新的任务在队列的头部可用时，或者新线程可能需要成为leader，则通过该条件发出信号
private final Condition available = lock.newCondition();
```

注意这里的leader，它是Leader-Follower模式的变体，用于减少不必要的定时等待。什么意思呢？对于多线程的网络模型来说：所有线程会有三种身份中的一种：leader和follower，以及一个干活中的状态：proccesser。它的基本原则就是，永远最多只有一个leader。而所有follower都在等待成为leader。线程池启动时会自动产生一个Leader负责等待网络IO事件，当有一个事件产生时，Leader线程首先通知一个Follower线程将其提拔为新的Leader，然后自己就

去干活了，去处理这个网络事件，处理完毕后加入Follower线程等待队列，等待下次成为Leader。这种方法可以增强CPU高速缓存相似性，及消除动态内存分配和线程间的数据交换。

offer方法

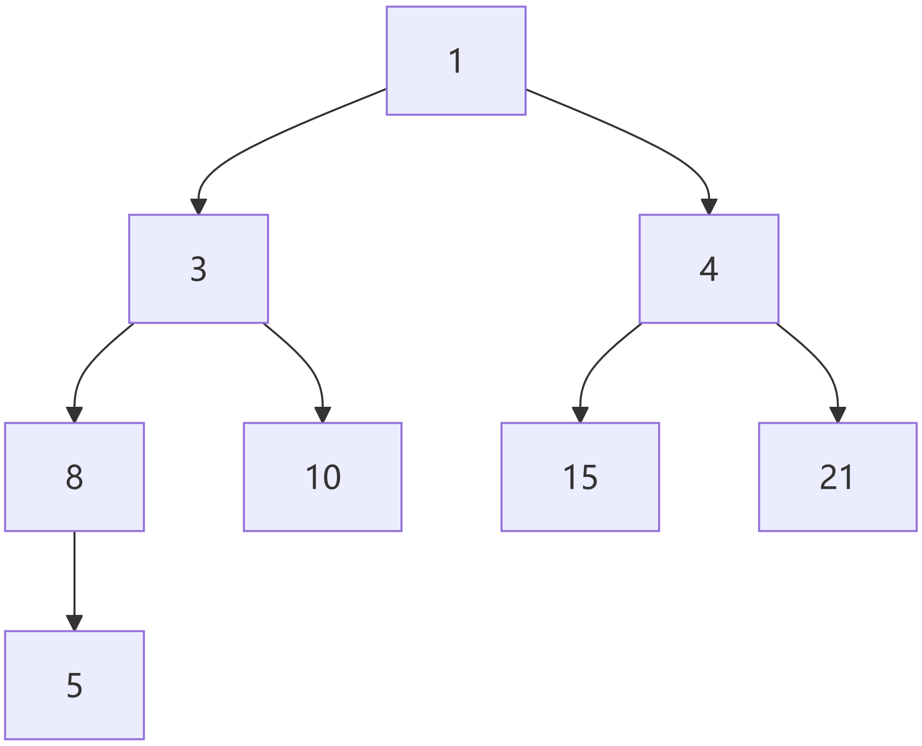
```java
public boolean offer(Runnable x) {
    if (x == null)
        throw new NullPointerException();
    RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x;
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        int i = size;
        // queue是一个RunnableScheduledFuture类型的数组，如果容量不够需要扩容
        if (i >= queue.length)
            grow();
        size = i + 1;
        // i == 0 说明堆中还没有数据
        if (i == 0) {
            queue[0] = e;
            setIndex(e, 0);
        } else {
            // i != 0 时，需要对堆进行重新排序
            siftUp(i, e);
        }
        // 如果传入的任务已经是队列的第一个节点了，这时available需要发出信号
        if (queue[0] == e) {
            // leader设置为null为了使在take方法中的线程在通过available.signal();后会执行
available.awaitNanos(delay);
            leader = null;
            available.signal();
        }
    } finally {
        lock.unlock();
    }
    return true;
}
```

siftUp方法

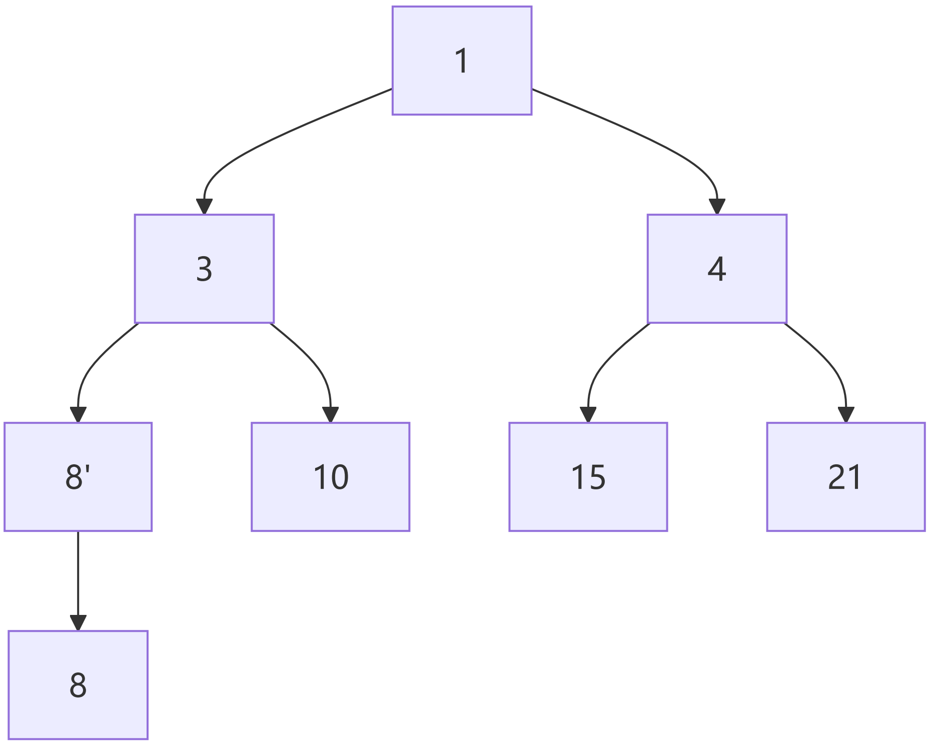
```java
private void siftUp(int k, RunnableScheduledFuture<?> key) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        RunnableScheduledFuture<?> e = queue[parent];
        if (key.compareTo(e) >= 0)
            break;
        queue[k] = e;
        setIndex(e, k);
        k = parent;
    }
    queue[k] = key;
    setIndex(key, k);
}
```

代码很好理解，就是循环的根据key节点与它的父节点来判断，如果key节点的执行时间小于父节点，则将两个节点交换，使执行时间靠前的节点排列在队列的前面。假设新入队的节点的延迟时间（调用getDelay()方法获得）是5，执行过程如下：

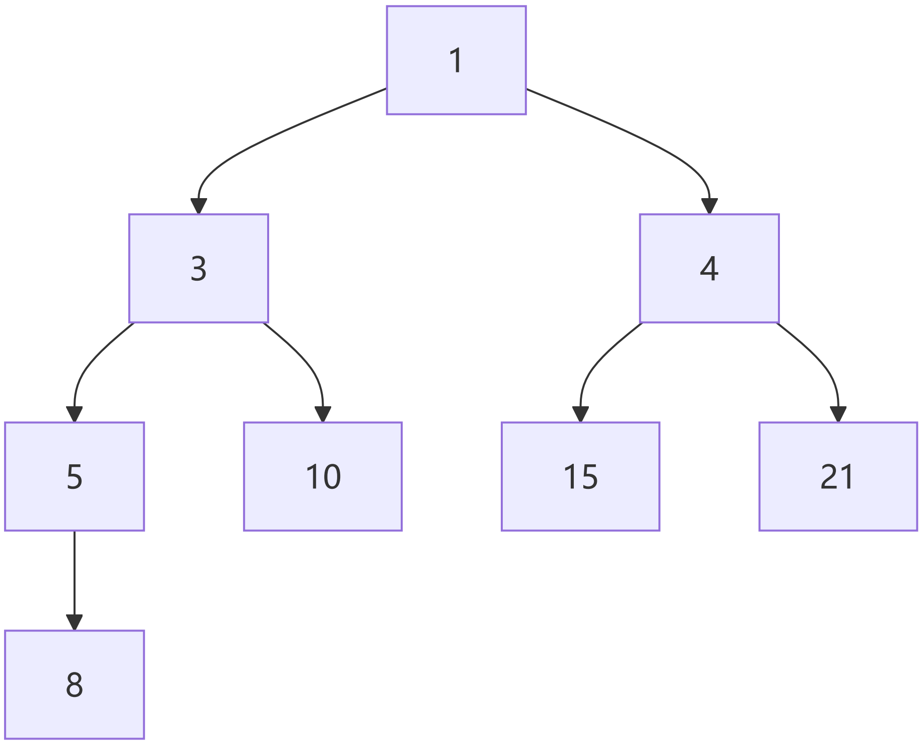
1. 先将新的节点添加到数组的尾部，这时新节点的索引k为7：



2. 计算新父节点的索引：parent = (k - 1) >>> 1， parent = 3，那么queue[3]的时间间隔值为8，因为 5 < 8 ，将执行queue[7] = queue[3]：



3. 这时将k设置为3，继续循环，再次计算parent为1，queue[1]的时间间隔为3，因为 5 > 3 ，这时退出循环，最终k为3：



可见，每次新增节点时，只是根据父节点来判断，而不会影响兄弟节点。另外，setIndex方法只是设置了ScheduledFutureTask中的heapIndex属性(即设置在队列中的索引)：

```java
private void setIndex(RunnableScheduledFuture<?> f, int idx) {
    if (f instanceof ScheduledFutureTask)
        ((ScheduledFutureTask)f).heapIndex = idx;
}
```

take方法

```java
public RunnableScheduledFuture<?> take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            RunnableScheduledFuture<?> first = queue[0];
            if (first == null)
                available.await();
            else {
                // 计算当前时间到执行时间的时间间隔
                long delay = first.getDelay(NANOSECONDS);
                if (delay <= 0)
                    return finishPoll(first);
                first = null; // don't retain ref while waiting
                if (leader != null)
                    // leader不为空，阻塞线程(进入条件队列)
                    available.await();
                else {
                    // leader为空，则把leader设置为当前线程
                    Thread thisThread = Thread.currentThread();
                    leader = thisThread;
                    try {
                        // 阻塞到执行时间
                        available.awaitNanos(delay);
                    } finally {
                        // 设置leader = null，让其他线程执行available.awaitNanos(delay);
                        if (leader == thisThread)
                            leader = null;
                    }
                }
            }
        }
    } finally {
        // 如果leader不为空，则说明leader的线程正在执行available.awaitNanos(delay);
        // 如果queue[0] == null，说明队列为空
        if (leader == null && queue[0] != null)
            available.signal();
        lock.unlock();
    }
}
```

在ThreadPoolExecutor的getTask方法中，工作线程会循环地从workQueue中取任务。但定时任务却不同，在take方法中，要保证只有在到指定的执行时间的时候任务才可以被取走。再来说一下leader的作用，这里的leader是为了减少不必要的定时等待，当一个线程成为leader时，它只等待下一个节点的时间间隔，但其它线程无限期等待。leader线程必须在从take()或poll()返回之前signal其它线程，除非其他线程成为了leader。举例来说，如果没有leader，当十个线程在等待同一个任务的时候，都会执行available.awaitNanos(delay)，那么在delay时间后十个线程又被同时唤醒但是只有一个线程能拿到任务，这是完全不合理的，所以这里增加了leader，如果leader不为空，则说明队列中第一个节点已经在等待出队，这时其它的线程会一直阻塞，减少了无用的阻塞。

poll方法

```
public RunnableScheduledFuture<?> poll(long timeout, TimeUnit unit)
    throws InterruptedException {
    long nanos = unit.toNanos(timeout);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            RunnableScheduledFuture<?> first = queue[0];
            if (first == null) {
                if (nanos <= 0)
                    return null;
                else
                    nanos = available.awaitNanos(nanos);
            } else {
                long delay = first.getDelay(NANOSECONDS);
                // 如果delay <= 0，说明已经到了任务执行的时间，返回。
                if (delay <= 0)
                    return finishPoll(first);
                // 如果nanos <= 0，说明已经超时，返回null
                if (nanos <= 0)
                    return null;
                first = null; // don't retain ref while waiting
                // nanos < delay 说明需要等待的时间小于任务要执行的延迟时间
                // leader != null 说明有其它线程正在对任务进行阻塞
                // 这时阻塞当前线程nanos纳秒
                if (nanos < delay || leader != null)
                    nanos = available.awaitNanos(nanos);
                else {
                    Thread thisThread = Thread.currentThread();
                    leader = thisThread;
                    try {
                        // 这里的timeLeft表示delay减去实际的等待时间
                        long timeLeft = available.awaitNanos(delay);
                        // 计算剩余的等待时间
                        nanos -= delay - timeLeft;
                    } finally {
                        if (leader == thisThread)
                            leader = null;
                    }
                }
            }
        }
    } finally {
        if (leader == null && queue[0] != null)
            available.signal();
        lock.unlock();
    }
}
```

finishPoll方法

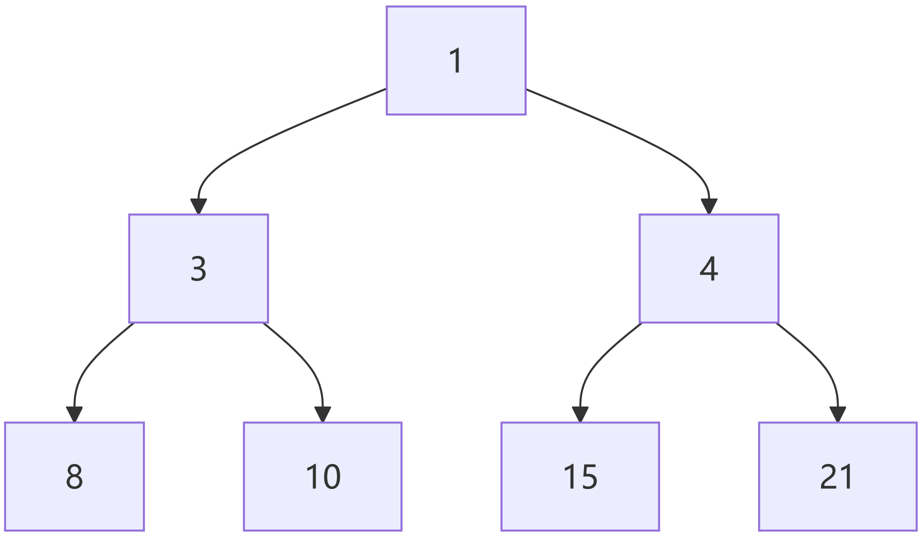
```
private RunnableScheduledFuture<?> finishPoll(RunnableScheduledFuture<?> f) {
    // 数组长度-1
    int s = --size;
    // 取出最后一个节点
    RunnableScheduledFuture<?> x = queue[s];
    queue[s] = null;
    // 长度不为0，则从第一个元素开始排序，目的是要把最后一个节点放到合适的位置上
    if (s != 0)
        siftDown(0, x);
    setIndex(f, -1);
    return f;
}
```

siftDown方法(siftDown方法使堆从k开始向下调整):

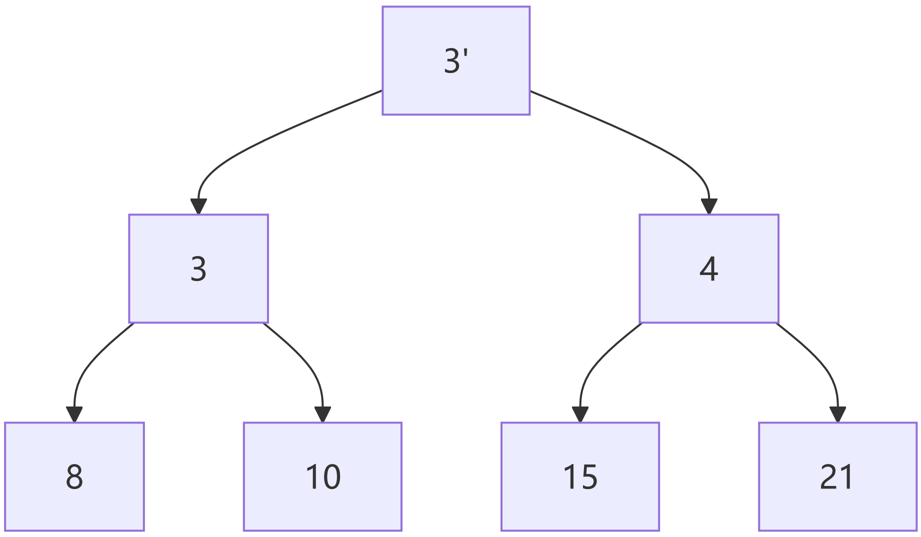
```
private void siftDown(int k, RunnableScheduledFuture<?> key) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        RunnableScheduledFuture<?> c = queue[child];
        int right = child + 1;
        if (right < size && c.compareTo(queue[right]) > 0)
            c = queue[child = right];
        if (key.compareTo(c) <= 0)
            break;
        queue[k] = c;
        setIndex(c, k);
        k = child;
    }
    queue[k] = key;
    setIndex(key, k);
}
```

假设k=0，那么执行如下步骤:

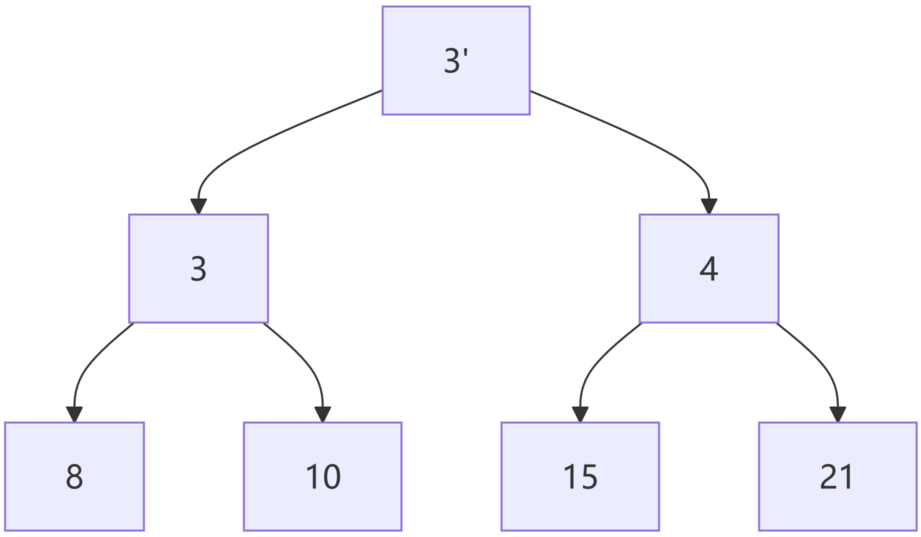
1. 获取左子节点，child = 1 ， 获取右子节点， right = 2 :



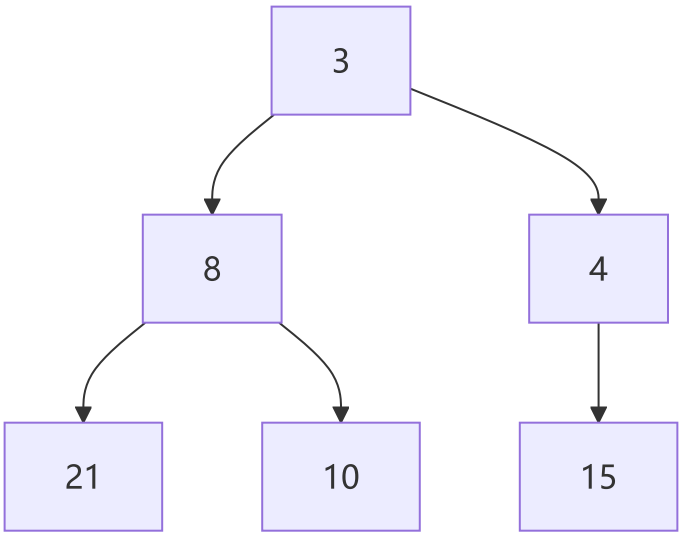
2. 由于 right < size ， 这时比较左子节点和右子节点时间间隔的大小，这里 3 < 7 ， 所以 c = queue[child] ；
3. 比较key的时间间隔是否小于c的时间间隔，这里不满足，继续执行，把索引为k的节点设置为c，然后将k设置为child，；



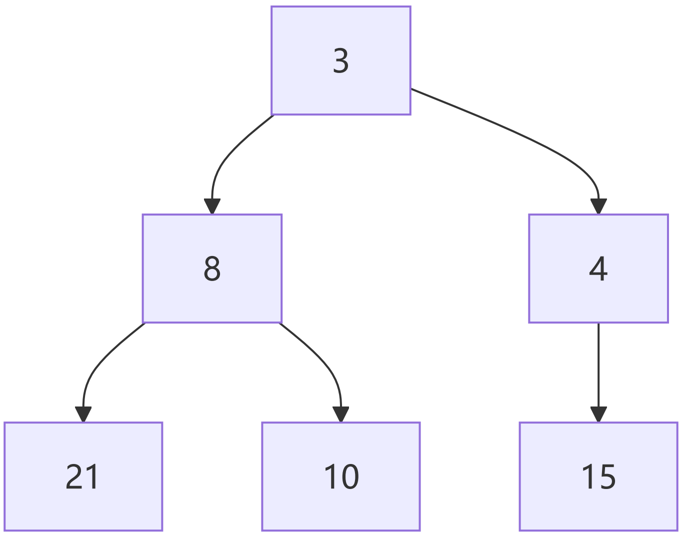
4. 因为 half = 3 ， k = 1 ， 继续执行循环，这时的索引变为：



5. 这时再经过如上判断后，将k的值为3，最终的结果如下：



6. 最后，如果在finishPoll方法中调用的话，会把索引为0的节点的索引设置为-1，表示已经删除了该节点，并且size也减了1，最后的结果如下：



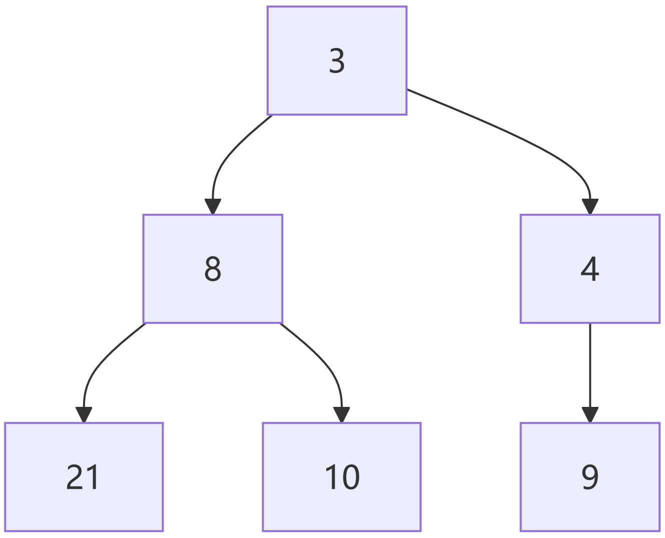
可见，siftdown方法在执行完并不是有序的，但可以发现，子节点的下次执行时间一定比父节点的下次执行时间要大，由于每次都会取左子节点和右子节点中下次执行时间最小的节点，所以还是可以保证在take和poll时出队是有序的。

1.

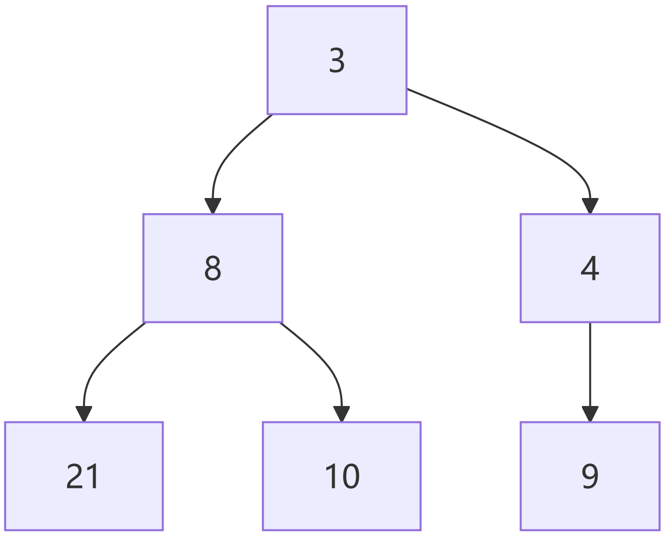
remove方法

```
public boolean remove(Object x) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        int i = indexOf(x);
        if (i < 0)
            return false;

        setIndex(queue[i], -1);
        int s = --size;
        RunnableScheduledFuture<?> replacement = queue[s];
        queue[s] = null;
        if (s != i) {
            // 从i开始向下调整
            siftDown(i, replacement);
            // 如果queue[i] == replacement，说明i是叶子节点
            // 如果是这种情况，不能保证子节点的下次执行时间比父节点的大
            // 这时需要进行一次向上调整
            if (queue[i] == replacement)
                siftUp(i, replacement);
        }
        return true;
    } finally {
        lock.unlock();
    }
}
```

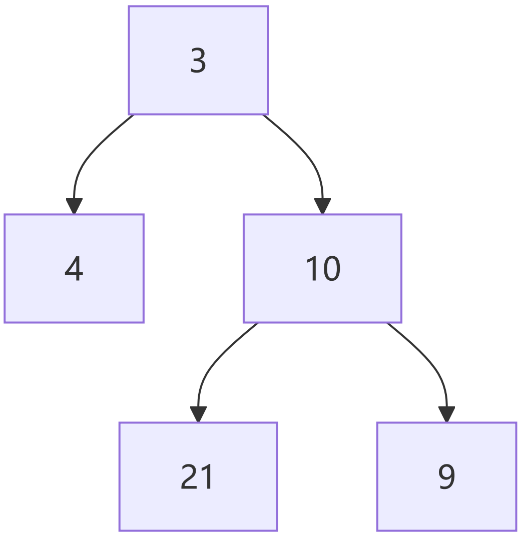
假设初始的堆结构如下：



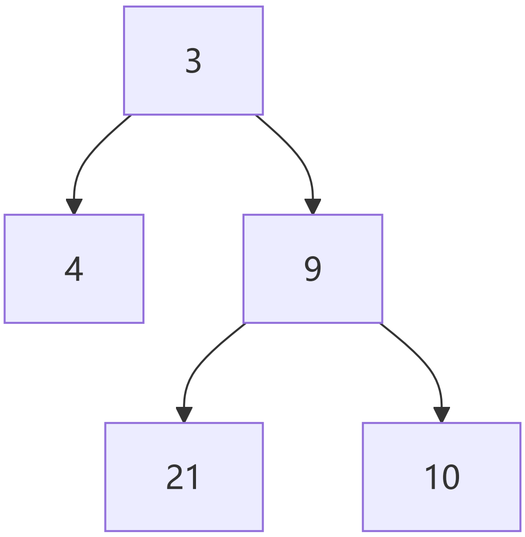
这时要删除8的节点，那么这时 k = 1，key为最后一个节点：



这时通过上文对siftDown方法的分析，siftDown方法执行后的结果如下：



这时会发现，最后一个节点的值比父节点还要小，所以这里要执行一次siftUp方法来保证子节点的下次执行时间要比父节点的大，所以最终结果如下：



总结

1. 与Timer执行定时任务的比较，相比Timer，ScheduedThreadPoolExecutor有什么优点；
2. ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，所以它也是一个线程池，也有coorPoolSize和workQueue，ScheduledThreadPoolExecutor特殊的地方在于，自己实现了优先工作队列DelayedWorkQueue；
3. ScheduedThreadPoolExecutor实现了ScheduledExecutorService，所以就有了任务调度的方法，如schedule，scheduleAtFixedRate和scheduleWithFixedDelay，同时注意他们之间的区别；
4. 内部类ScheduledFutureTask继承自FutureTask，实现了任务的异步执行并且可以获取返回结果。同时也实现了Delayed接口，可以通过getDelay方法获取将要执行的时间间隔；
5. 周期任务的执行其实是调用了FutureTask类中的runAndReset方法，每次执行完不设置结果和状态。
6. DelayedWorkQueue是一个基于最小堆结构的优先队列，并且每次出队时能够保证取出的任务是当前队列中下次执行时间最小的任务。同时注意一下优先队列中堆的顺序，堆中的顺序并不是绝对的，但要保证子节点的值要比父节点的值要大，这样就不会影响出队的顺序。

onShutdown方法

```
/**
 * onShutdown方法是ThreadPoolExecutor中的钩子方法，在ThreadPoolExecutor中什么都没有做，该方法是在执行shutdown方法时被调用
 */
@Override void onShutdown() {
    BlockingQueue<Runnable> q = super.getQueue();
    // 获取在线程池已 shutdown 的情况下是否继续执行现有延迟任务
    boolean keepDelayed =
        getExecuteExistingDelayedTasksAfterShutdownPolicy();
    // 获取在线程池已 shutdown 的情况下是否继续执行现有定期任务
    boolean keepPeriodic =
        getContinueExistingPeriodicTasksAfterShutdownPolicy();
    // 如果在线程池已 shutdown 的情况下不继续执行延迟任务和定期任务
    // 则依次取消任务，否则则根据取消状态来判断
    if (!keepDelayed && !keepPeriodic) {
        for (Object e : q.toArray())
            if (e instanceof RunnableScheduledFuture<?>)
                ((RunnableScheduledFuture<?>) e).cancel(false);
        // 当workQueue为空后，线程在ThreadPoolExecutor的getTask处会返回null，这样所有的线程在执行完
最后一个任务后就退出
        q.clear();
    }
    else {
        // Traverse snapshot to avoid iterator exceptions
        for (Object e : q.toArray()) {
            if (e instanceof RunnableScheduledFuture) {
                RunnableScheduledFuture<?> t =
                    (RunnableScheduledFuture<?>)e;
                // 如果有在 shutdown 后不继续的延迟任务或周期任务，则从队列中删除并取消任务
                if ((t.isPeriodic() ? !keepPeriodic : !keepDelayed) ||
                    t.isCancelled()) { // also remove if already cancelled
                    if (q.remove(t))
                        t.cancel(false);
                }
            }
        }
    }
    tryTerminate();
}
```

## ForkJoinPool

分治思想
在计算机科学中，分治法是一种很重要的算法。字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解就变成了子问题解的合并。这个技巧是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换(快速傅立叶变换)……，如果你是搞大数据的，MapReduce就是分支思想的典型，
分治法所能解决的问题一般具有以下几个特征：
该问题的规模缩小到一定的程度就可以容易地解决
该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。
利用该问题分解出的子问题的解可以合并为该问题的解；
该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题

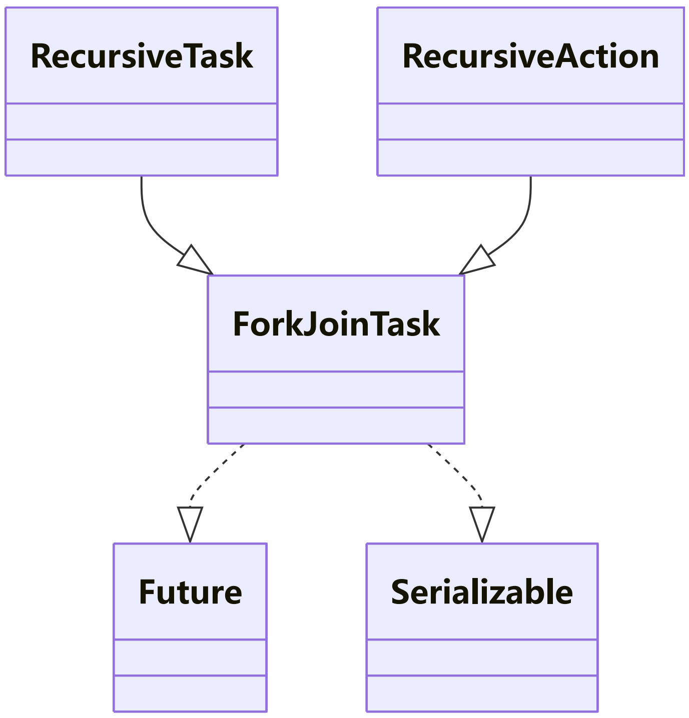
### ForkJoin

有子任务，自然要用到多线程。之前说过，执行子任务的线程不允许单独创建，要用线程池管理。秉承相同设计理念，再结合分治算法，ForkJoin框架中就出现了ForkJoinPool和ForkJoinTask。

### ForkJoinTask

ForkJoinTask实现了Future接口（那就是具有Future接口的特性），同样如其名，fork()和join()自然是它的两个核心方法

1. fork():异步执行一个子任务（上面说的拆分）
2. join():阻塞当前线程等待子任务的执行结果（上面说的合并）



ForkJoinTask是一个抽象类，在分治模型中，它还有两个抽象子类 RecursiveAction(没有返回值)和 RecursiveTask(有返回值)。两个类里面都定义了一个抽象方法compute()，需要子类重写实现具体逻辑。子类遵循分治思想重写这个方法：

1. 什么时候进一步拆分任务？
2. 什么时候满足最小可执行任务，即不再进行拆分？
3. 什么时候汇总子任务结果

## fork

```
/**
 * Fork方法的逻辑很简单，如果当前线程是ForkJoinWorkerThread类型，也就是说已经通过上文注册的Worker，那么直接调用push方法将task放到当前线程拥有的WorkQueue中，否则就再调用externalPush
 * 如果当前线程不是ForkJoinWorkerThread说明，是在主线程中new出来一个ForkJoinTask，然后再直接调用fork方法(因此他使用的是ForkJoinPool默认的静态进程池)，所以说它不像上面那样先创建一个ForkJoinPool，然后再继承RecursiveTask定义自己的task，然后再将task提交到ForkJoinPool中，然后再有task来fork出子线程。
 */
public final ForkJoinTask<V> fork() {
    Thread t;
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
        ((ForkJoinWorkerThread)t).workQueue.push(this);
    else
        ForkJoinPool.common.externalPush(this);
    return this;
}
```

## join

```
/**
 *join的核心调用在doJoin
 */
public final V join() {
    int s;
    if ((s = doJoin() & DONE_MASK) != NORMAL)
        reportException(s);
    return getRawResult();
}

private int doJoin() {
    int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
    // 有结果，直接返回
    return (s = status) < 0 ? s :
    ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) ?
        (w = (wt = (ForkJoinWorkerThread)t).workQueue).
        tryUnpush(this) && (s = doExec()) < 0 ? s :
    wt.pool.awaitJoin(w, this, 0L) :
    externalAwaitDone();
}
// 将上面的级联三元运算符变成下面的if/else:
private int doJoin() {
  int s;
  Thread t;
  ForkJoinWorkerThread wt;
  ForkJoinPool.WorkQueue w;

  if((s = status) < 0) { // 有结果，直接返回
    return s;
  }else {
    if((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
      // 如果是 ForkJoinWorkerThread Worker
      if((w = (wt = (ForkJoinWorkerThread) t).workQueue).tryUnpush(this) // 类似上面提到的
scan，但是是从本工作队列里取出等待的任务
         // 取出了任务，就去执行它，并返回结果
         && (s = doExec()) < 0) {
        return s;
      }else {
        // 也有可能别的线程把这个任务偷走了，那就执行内部等待方法
        return wt.pool.awaitJoin(w, this, 0L);
      }
    }else {
      // 如果不是 ForkJoinWorkerThread，执行外部等待方法
      return externalAwaitDone();
    }
  }
}
```

## ForkJoinWorkerThread

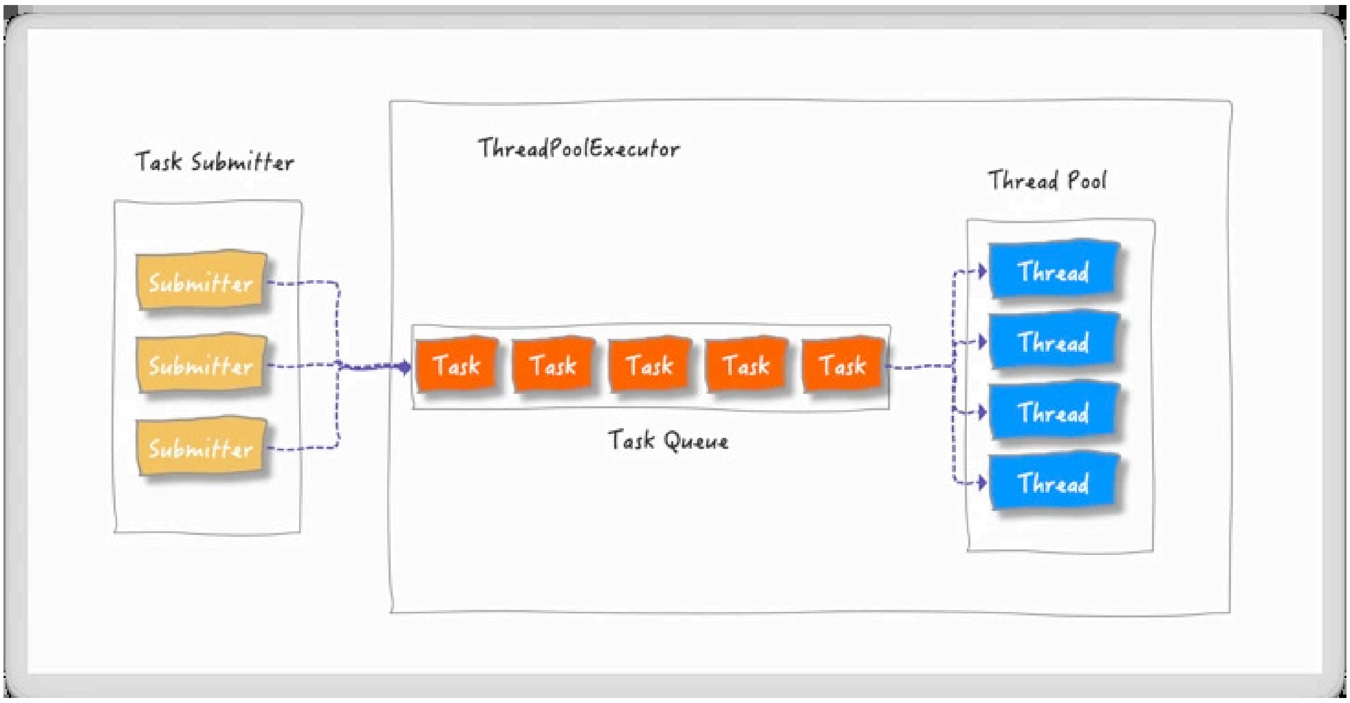
```java
package java.util.concurrent;

import java.security.AccessControlContext;
import java.security.ProtectionDomain;

public class ForkJoinWorkerThread extends Thread {

    final ForkJoinPool pool;                // the pool this thread works in
    final ForkJoinPool.WorkQueue workQueue; // work-stealing mechanics

    /**
     * Creates a ForkJoinWorkerThread operating in the given pool.
     *
     * @param pool the pool this thread works in
     * @throws NullPointerException if pool is null
     */
    protected ForkJoinWorkerThread(ForkJoinPool pool) {
        // Use a placeholder until a useful name can be set in registerWorker
        super("aForkJoinWorkerThread");
        this.pool = pool;
        this.workQueue = pool.registerWorker(this);
    }

    /**
     * Version for InnocuousForkJoinWorkerThread
     */
    ForkJoinWorkerThread(ForkJoinPool pool, ThreadGroup threadGroup,
                         AccessControlContext acc) {
        super(threadGroup, null, "aForkJoinWorkerThread");
        U.putOrderedObject(this, INHERITEDACCESSCONTROLCONTEXT, acc);
        eraseThreadLocals(); // clear before registering
        this.pool = pool;
        this.workQueue = pool.registerWorker(this);
    }

    /**
     * Returns the pool hosting this thread.
     *
     * @return the pool
     */
    public ForkJoinPool getPool() {
        return pool;
    }

    /**
     * Returns the unique index number of this thread in its pool.
     * The returned value ranges from zero to the maximum number of
     * threads (minus one) that may exist in the pool, and does not
     * change during the lifetime of the thread.  This method may be
     * useful for applications that track status or collect results
     * per-worker-thread rather than per-task.
     *
     * @return the index number
     */
    public int getPoolIndex() {
        return workQueue.getPoolIndex();
    }

    /**
     * Initializes internal state after construction but before
     * processing any tasks. If you override this method, you must
     * invoke {@code super.onStart()} at the beginning of the method.
     * Initialization requires care: Most fields must have legal
     * default values, to ensure that attempted accesses from other
     * threads work correctly even before this thread starts
     * processing tasks.
     */
    protected void onStart() {
    }

    /**
     * Performs cleanup associated with termination of this worker
     * thread.  If you override this method, you must invoke
     * {@code super.onTermination} at the end of the overridden method.
     *
     * @param exception the exception causing this thread to abort due
     * to an unrecoverable error, or {@code null} if completed normally
     */
    protected void onTermination(Throwable exception) {
    }

    /**
     * This method is required to be public, but should never be
     * called explicitly. It performs the main run loop to execute
     * {@link ForkJoinTask}s.
     */
    public void run() {
        if (workQueue.array == null) { // only run once
            Throwable exception = null;
            try {
                onStart();
                pool.runWorker(workQueue);
            } catch (Throwable ex) {
                exception = ex;
            } finally {
                try {
                    onTermination(exception);
                } catch (Throwable ex) {
                    if (exception == null)
                        exception = ex;
                } finally {
                    pool.deregisterWorker(this, exception);
                }
            }
        }
    }

    /**
     * Erases ThreadLocals by nulling out Thread maps.
     */
    final void eraseThreadLocals() {
        U.putObject(this, THREADLOCALS, null);
        U.putObject(this, INHERITABLETHREADLOCALS, null);
    }

    /**
     * Non-public hook method for InnocuousForkJoinWorkerThread
     */
    void afterTopLevelExec() {
    }

    // Set up to allow setting thread fields in constructor
    private static final sun.misc.Unsafe U;
    private static final long THREADLOCALS;
    private static final long INHERITABLETHREADLOCALS;
    private static final long INHERITEDACCESSCONTROLCONTEXT;
    static {
        try {
            U = sun.misc.Unsafe.getUnsafe();
            Class<?> tk = Thread.class;
            THREADLOCALS = U.objectFieldOffset
                (tk.getDeclaredField("threadLocals"));
            INHERITABLETHREADLOCALS = U.objectFieldOffset
                (tk.getDeclaredField("inheritableThreadLocals"));
            INHERITEDACCESSCONTROLCONTEXT = U.objectFieldOffset
                (tk.getDeclaredField("inheritedAccessControlContext"));

        } catch (Exception e) {
            throw new Error(e);
        }
    }

    /**
     * A worker thread that has no permissions, is not a member of any
     * user-defined ThreadGroup, and erases all ThreadLocals after
     * running each top-level task.
     */
    static final class InnocuousForkJoinWorkerThread extends ForkJoinWorkerThread {
        /** The ThreadGroup for all InnocuousForkJoinWorkerThreads */
        private static final ThreadGroup innocuousThreadGroup =
            createThreadGroup();

        /** An AccessControlContext supporting no privileges */
        private static final AccessControlContext INNOCUOUS_ACC =
            new AccessControlContext(
                new ProtectionDomain[] {
```

```
                new ProtectionDomain(null, null)
            });

    InnocuousForkJoinWorkerThread(ForkJoinPool pool) {
        super(pool, innocuousThreadGroup, INNOCUOUS_ACC);
    }

    @Override // to erase ThreadLocals
    void afterTopLevelExec() {
        eraseThreadLocals();
    }

    @Override // to always report system loader
    public ClassLoader getContextClassLoader() {
        return ClassLoader.getSystemClassLoader();
    }

    @Override // to silently fail
    public void setUncaughtExceptionHandler(UncaughtExceptionHandler x) { }

    @Override // paranoically
    public void setContextClassLoader(ClassLoader cl) {
        throw new SecurityException("setContextClassLoader");
    }

    /**
     * Returns a new group with the system ThreadGroup (the
     * topmost, parent-less group) as parent.  Uses Unsafe to
     * traverse Thread.group and ThreadGroup.parent fields.
     */
    private static ThreadGroup createThreadGroup() {
        try {
            sun.misc.Unsafe u = sun.misc.Unsafe.getUnsafe();
            Class<?> tk = Thread.class;
            Class<?> gk = ThreadGroup.class;
            long tg = u.objectFieldOffset(tk.getDeclaredField("group"));
            long gp = u.objectFieldOffset(gk.getDeclaredField("parent"));
            ThreadGroup group = (ThreadGroup)
                u.getObject(Thread.currentThread(), tg);
            while (group != null) {
                ThreadGroup parent = (ThreadGroup)u.getObject(group, gp);
                if (parent == null)
                    return new ThreadGroup(group,
                                           "InnocuousForkJoinWorkerThreadGroup");
                group = parent;
            }
        } catch (Exception e) {
            throw new Error(e);
        }
        // fall through if null as cannot-happen safeguard
        throw new Error("Cannot create ThreadGroup");
    }
}
```

```
}
```

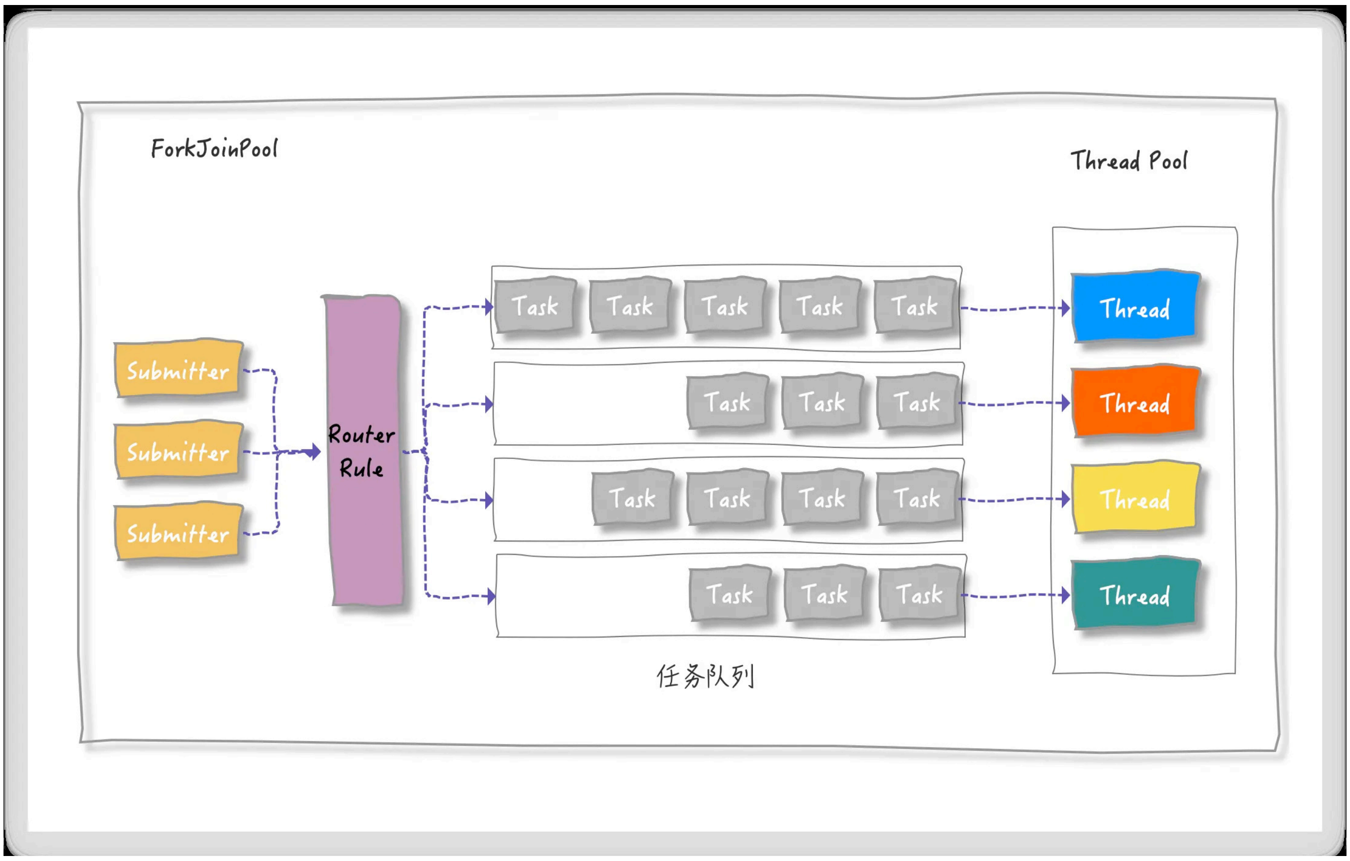
## ForkJoinPool

先来回忆一下 ThreadPoolExecutor 的实现原理



这是典型的生产者/消费者模式，消费者线程都从一个共享的 Task Queue中消费提交的任务。ThreadPoolExecutor简单的并行操作主要是为了执行时间不确定的任务（I/O 或定时任务等）
治思想其实也可以理解成一种父子任务依赖的关系，当依赖层级非常深，用 ThreadPoolExecutor 来处理这种关系很显然是不太现实的，所以 ForkJoinPool 作为功能补充就出现了

任务拆分后有依赖关系，还得减少线程之间的竞争，那就让线程执行属于自己的task就可以了，所以较ThreadPoolExecutor的单个TaskQueue的形式，ForkJoinPool是多个TaskQueue的形式，简单用图来表示，就是这样的：



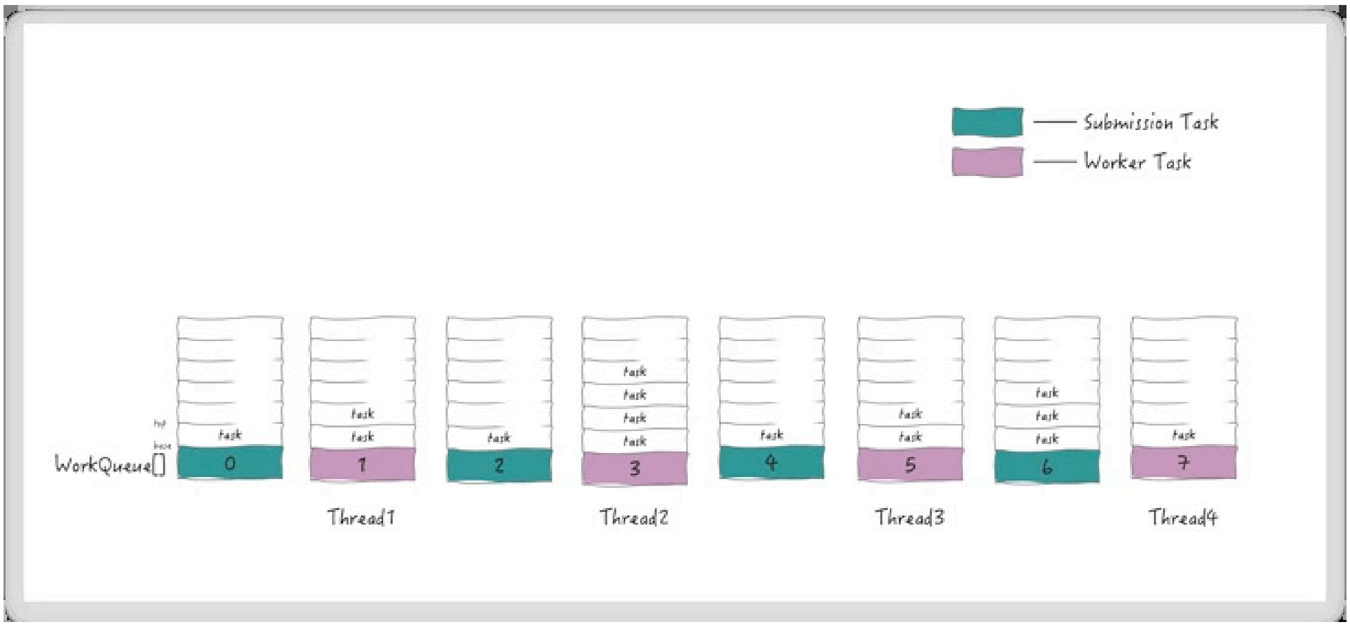
有多个任务队列，所以在ForkJoinPool中就有一个数组形式的成员变量WorkQueue[]。那问题又来了
任务队列有多个，提交的任务放到哪个队列中呢？（上图中的Router Rule部分）
这就需要一套路由规则，从上面的代码Demo中可以理解，提交的任务主要有两种：

1. 有外部直接提交的（submission task）
2. 也有任务自己fork出来的（worker task）

为了进一步区分这两种task，Doug Lea就设计一个简单的路由规则：

1. 将submission task放到 WorkQueue 数组的「偶数」下标中
2. 将worker task放在WorkQueue的「奇数」下标中，并且只有奇数下标才有线程(worker)与之相对

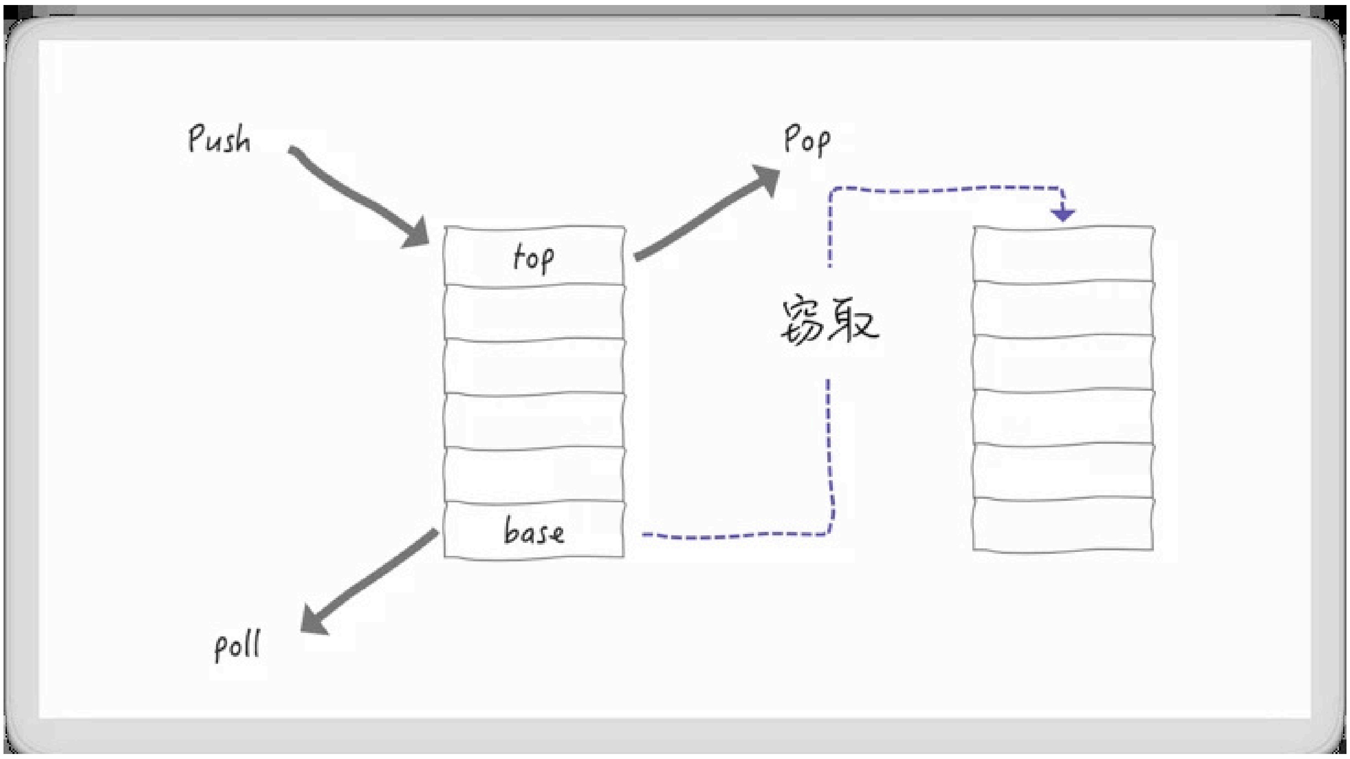
应局部丰富一下上图就是这样：



每个任务执行时间都是不一样的（当然是在CPU眼里），执行快的线程的工作队列的任务就可能是空的，为了最大化利用CPU资源，就允许空闲线程拿取其它任务队列中的内容，这个过程就叫做work-stealing(工作窃取)当前线程要执行一个任务，其他线程还有可能过来窃取任务，这就会产生竞争，为了减少竞争，Work Queue就设计成了一个双端队列：

1. 支持LIFO(last-in-first-out) 的push（放）和pop（拿）操作——操作top端
2. 支持FIFO(first-in-first-out) 的poll（拿）操作——操作base端

线程（worker）操作自己的WorkQueue默认是LIFO操作(可选FIFO)，当线程（worker）尝试窃取其他WorkQueue里的任务时，这个时候执行的是FIFO操作，即从base端窃取，用图丰富一下就是这样的：



这样的好处非常明显了：

1. LIFO 操作只有对应的 worker 才能执行，push和pop不需要考虑并发
2. 拆分时，越大的任务越在WorkQueue的base端，尽早分解，能够尽快进入计算

从 WorkQueue 的成员变量的修饰符中也能看出一二了(base 有 volatile 修饰，而 top 却没有)：

```
volatile int base;         // index of next slot for poll
int top;                   // index of next slot for push
```

ForkJoinPool 里有三个重要的角色：

1. ForkJoinWorkerThread（继承 Thread）：就是上面说的线程（Worker）
2. WorkQueue：双向的任务队列
3. ForkJoinTask：Worker执行的对象

```
package java.util.concurrent;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.AbstractExecutorService;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.RejectedExecutionException;
import java.util.concurrent.RunnableFuture;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.security.AccessControlContext;
import java.security.ProtectionDomain;
import java.security.Permissions;

/**
 * ForkJoinPool里有三个重要的角色：
 * ForkJoinWorkerThread（继承Thread）：就是上面说的线程（Worker）
 * WorkQueue：双向的任务队列
 * ForkJoinTask：Worker 执行的对象
 */
@sun.misc.Contended
public class ForkJoinPool extends AbstractExecutorService {

    /*
     * Implementation Overview
     */

    /**
     * If there is a security manager, makes sure caller has
     * permission to modify threads.
     */
    private static void checkPermission() {
        SecurityManager security = System.getSecurityManager();
        if (security != null)
```

```
            security.checkPermission(modifyThreadPermission);
    }

    // Nested classes

    /**
     * Factory for creating new {@link ForkJoinWorkerThread}s.
     * A {@code ForkJoinWorkerThreadFactory} must be defined and used
     * for {@code ForkJoinWorkerThread} subclasses that extend base
     * functionality or initialize threads with different contexts.
     */
    public static interface ForkJoinWorkerThreadFactory {
        /**
         * Returns a new worker thread operating in the given pool.
         *
         * @param pool the pool this thread works in
         * @return the new worker thread
         * @throws NullPointerException if the pool is null
         */
        public ForkJoinWorkerThread newThread(ForkJoinPool pool);
    }

    /**
     * Default ForkJoinWorkerThreadFactory implementation; creates a
     * new ForkJoinWorkerThread.
     */
    static final class DefaultForkJoinWorkerThreadFactory
        implements ForkJoinWorkerThreadFactory {
        public final ForkJoinWorkerThread newThread(ForkJoinPool pool) {
            return new ForkJoinWorkerThread(pool);
        }
    }

    /**
     * Class for artificial tasks that are used to replace the target
     * of local joins if they are removed from an interior queue slot
     * in WorkQueue.tryRemoveAndExec. We don't need the proxy to
     * actually do anything beyond having a unique identity.
     */
    static final class EmptyTask extends ForkJoinTask<Void> {
        private static final long serialVersionUID = -7721805057305804111L;
        EmptyTask() { status = ForkJoinTask.NORMAL; } // force done
        public final Void getRawResult() { return null; }
        public final void setRawResult(Void x) {}
        public final boolean exec() { return true; }
    }

    // Constants shared across ForkJoinPool and WorkQueue

    // Bounds
    static final int SMASK        = 0xffff;        // short bits == max index
    // 00000000000000001111111111111111
    static final int MAX_CAP      = 0x7fff;        // max #workers - 1
```

```java
    static final int EVENMASK     = 0xfffe;        // even short bits
    static final int SQMASK       = 0x007e;        // max 64 (even) slots

    // Masks and units for WorkQueue.scanState and ctl sp subfield
    static final int SCANNING     = 1;             // false when running tasks
    static final int INACTIVE     = 1 << 31;       // must be negative
    static final int SS_SEQ       = 1 << 16;       // version count

    // Mode bits for ForkJoinPool.config and WorkQueue.config
    // 11111111111111110000000000000000
    static final int MODE_MASK    = 0xffff << 16;  // top half of int
    static final int LIFO_QUEUE   = 0;
    // 00000000000000010000000000000000
    static final int FIFO_QUEUE   = 1 << 16;
    // 10000000000000000000000000000000
    static final int SHARED_QUEUE = 1 << 31;       // must be negative

    /**
     * Queues supporting work-stealing as well as external task
     * submission. See above for descriptions and algorithms.
     * Performance on most platforms is very sensitive to placement of
     * instances of both WorkQueues and their arrays -- we absolutely
     * do not want multiple WorkQueue instances or multiple queue
     * arrays sharing cache lines. The @Contended annotation alerts
     * JVMs to try to keep instances apart.
     */
    @sun.misc.Contended
    static final class WorkQueue {

        /**
         * 初始队列容量
         */
        static final int INITIAL_QUEUE_CAPACITY = 1 << 13;

        /**
         * 最大队列容量
         */
        static final int MAXIMUM_QUEUE_CAPACITY = 1 << 26; // 64M

        // Instance fields
        // WorkQueue是一个双端队列，线程池有runState，WorkQueue有scanState
        volatile int scanState;    // versioned, <0: inactive; odd:scanning
        // 小于零：inactive (未激活状态)
        // 奇数：scanning （扫描状态）
        // 偶数：running （运行状态）
        int stackPred;             // pool stack (ctl) predecessor 前任池（WorkQueue[]）索引，
由此构成一个栈
        int nsteals;               // number of steals 偷取的任务个数
        int hint;                  // randomization and stealer index hint 记录偷取者的索引
        // WorkQueue中也有个config，但是和ForkJoinPool中的是不一样的，WorkQueue中的config记录了该
WorkQueue在WorkQueue[]数组的下标以及mode
        int config;                // pool index and mode
        // 操作线程池需要锁，操作队列也是需要锁的，qlock就是操作队列的锁
        volatile int qlock;        // 1: locked, < 0: terminate; else 0
        // 1: 锁定
        // 0：未锁定
        // 小于零：终止状态
        volatile int base;         // index of next slot for poll
        int top;                   // index of next slot for push
        ForkJoinTask<?>[] array;   // the elements (initially unallocated) 任务数组
        final ForkJoinPool pool;   // the containing pool (may be null)
        final ForkJoinWorkerThread owner; // owning thread or null if shared 当前工作队列的工作
线程，共享模式下为null
        volatile Thread parker;    // == owner during call to park; else null 调用park阻塞期间
为owner，其他情况为null
        volatile ForkJoinTask<?> currentJoin;  // task being joined in awaitJoin 记录当前join
来的任务
        volatile ForkJoinTask<?> currentSteal; // mainly used by helpStealer 记录从其他工作队列
偷取过来的任务

        WorkQueue(ForkJoinPool pool, ForkJoinWorkerThread owner) {
            this.pool = pool;
            this.owner = owner;
            // Place indices in the center of array (that is not yet allocated)
            base = top = INITIAL_QUEUE_CAPACITY >>> 1;
        }

        /**
         * Returns an exportable index (used by ForkJoinWorkerThread).
         */
        final int getPoolIndex() {
            return (config & 0xffff) >>> 1; // ignore odd/even tag bit
        }

        /**
         * Returns the approximate number of tasks in the queue.
         */
        final int queueSize() {
            int n = base - top;       // non-owner callers must read base first
            return (n >= 0) ? 0 : -n; // ignore transient negative
        }

        /**
         * Provides a more accurate estimate of whether this queue has
         * any tasks than does queueSize, by checking whether a
         * near-empty queue has at least one unclaimed task.
         */
        final boolean isEmpty() {
            ForkJoinTask<?>[] a; int n, m, s;
            return ((n = base - (s = top)) >= 0 ||
                    (n == -1 &&           // possibly one task
                     ((a = array) == null || (m = a.length - 1) < 0 ||
                      U.getObject
                      (a, (long)((m & (s - 1)) << ASHIFT) + ABASE) == null)));
        }
```

```java
    /**
     * Pushes a task. Call only by owner in unshared queues.  (The
     * shared-queue version is embedded in method externalPush.)
     *
     * @param task the task. Caller must ensure non-null.
     * @throws RejectedExecutionException if array cannot be resized
     */
    final void push(ForkJoinTask<?> task) {
        ForkJoinTask<?>[] a; ForkJoinPool p;
        int b = base, s = top, n;
        if ((a = array) != null) {    // ignore if queue removed
            int m = a.length - 1;     // fenced write for task visibility
            U.putOrderedObject(a, ((m & s) << ASHIFT) + ABASE, task);
            U.putOrderedInt(this, QTOP, s + 1);
            if ((n = s - b) <= 1) {
                if ((p = pool) != null)
                    p.signalWork(p.workQueues, this);
            }
            else if (n >= m)
                growArray();
        }
    }

    /**
     * Initializes or doubles the capacity of array. Call either
     * by owner or with lock held -- it is OK for base, but not
     * top, to move while resizings are in progress.
     */
    final ForkJoinTask<?>[] growArray() {
        ForkJoinTask<?>[] oldA = array;
        int size = oldA != null ? oldA.length << 1 : INITIAL_QUEUE_CAPACITY;
        if (size > MAXIMUM_QUEUE_CAPACITY)
            throw new RejectedExecutionException("Queue capacity exceeded");
        int oldMask, t, b;
        ForkJoinTask<?>[] a = array = new ForkJoinTask<?>[size];
        if (oldA != null && (oldMask = oldA.length - 1) >= 0 &&
            (t = top) - (b = base) > 0) {
            int mask = size - 1;
            do { // emulate poll from old array, push to new array
                ForkJoinTask<?> x;
                int oldj = ((b & oldMask) << ASHIFT) + ABASE;
                int j    = ((b &    mask) << ASHIFT) + ABASE;
                x = (ForkJoinTask<?>)U.getObjectVolatile(oldA, oldj);
                if (x != null &&
                    U.compareAndSwapObject(oldA, oldj, x, null))
                    U.putObjectVolatile(a, j, x);
            } while (++b != t);
        }
        return a;
    }

    /**
     * Takes next task, if one exists, in LIFO order.  Call only
     * by owner in unshared queues.
     */
    final ForkJoinTask<?> pop() {
        ForkJoinTask<?>[] a; ForkJoinTask<?> t; int m;
        if ((a = array) != null && (m = a.length - 1) >= 0) {
            for (int s; (s = top - 1) - base >= 0;) {
                long j = ((m & s) << ASHIFT) + ABASE;
                if ((t = (ForkJoinTask<?>)U.getObject(a, j)) == null)
                    break;
                if (U.compareAndSwapObject(a, j, t, null)) {
                    U.putOrderedInt(this, QTOP, s);
                    return t;
                }
            }
        }
        return null;
    }

    /**
     * Takes a task in FIFO order if b is base of queue and a task
     * can be claimed without contention. Specialized versions
     * appear in ForkJoinPool methods scan and helpStealer.
     */
    final ForkJoinTask<?> pollAt(int b) {
        ForkJoinTask<?> t; ForkJoinTask<?>[] a;
        if ((a = array) != null) {
            int j = (((a.length - 1) & b) << ASHIFT) + ABASE;
            if ((t = (ForkJoinTask<?>)U.getObjectVolatile(a, j)) != null &&
                base == b && U.compareAndSwapObject(a, j, t, null)) {
                base = b + 1;
                return t;
            }
        }
        return null;
    }

    /**
     * Takes next task, if one exists, in FIFO order.
     */
    final ForkJoinTask<?> poll() {
        ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
        while ((b = base) - top < 0 && (a = array) != null) {
            int j = (((a.length - 1) & b) << ASHIFT) + ABASE;
            t = (ForkJoinTask<?>)U.getObjectVolatile(a, j);
            if (base == b) {
                if (t != null) {
                    if (U.compareAndSwapObject(a, j, t, null)) {
                        base = b + 1;
                        return t;
                    }
                }
                else if (b + 1 == top) // now empty
                    break;
            }
        }
        return null;
    }

    /**
     * Takes next task, if one exists, in order specified by mode.
     */
    final ForkJoinTask<?> nextLocalTask() {
        return (config & FIFO_QUEUE) == 0 ? pop() : poll();
    }

    /**
     * Returns next task, if one exists, in order specified by mode.
     */
    final ForkJoinTask<?> peek() {
        ForkJoinTask<?>[] a = array; int m;
        if (a == null || (m = a.length - 1) < 0)
            return null;
        int i = (config & FIFO_QUEUE) == 0 ? top - 1 : base;
        int j = ((i & m) << ASHIFT) + ABASE;
        return (ForkJoinTask<?>)U.getObjectVolatile(a, j);
    }

    /**
     * Pops the given task only if it is at the current top.
     * (A shared version is available only via FJP.tryExternalUnpush)
    */
    final boolean tryUnpush(ForkJoinTask<?> t) {
        ForkJoinTask<?>[] a; int s;
        if ((a = array) != null && (s = top) != base &&
            U.compareAndSwapObject
            (a, (((a.length - 1) & --s) << ASHIFT) + ABASE, t, null)) {
            U.putOrderedInt(this, QTOP, s);
            return true;
        }
        return false;
    }

    /**
     * Removes and cancels all known tasks, ignoring any exceptions.
     */
    final void cancelAll() {
        ForkJoinTask<?> t;
        if ((t = currentJoin) != null) {
            currentJoin = null;
            ForkJoinTask.cancelIgnoringExceptions(t);
        }
        if ((t = currentSteal) != null) {
            currentSteal = null;
            ForkJoinTask.cancelIgnoringExceptions(t);
        }
        while ((t = poll()) != null)
```

```
                ForkJoinTask.cancelIgnoringExceptions(t);
        }

        // Specialized execution methods

        /**
         * Polls and runs tasks until empty.
         */
        final void pollAndExecAll() {
            for (ForkJoinTask<?> t; (t = poll()) != null;)
                t.doExec();
        }

        /**
         * Removes and executes all local tasks. If LIFO, invokes
         * pollAndExecAll. Otherwise implements a specialized pop loop
         * to exec until empty.
         */
        final void execLocalTasks() {
            int b = base, m, s;
            ForkJoinTask<?>[] a = array;
            if (b - (s = top - 1) <= 0 && a != null &&
                (m = a.length - 1) >= 0) {
                if ((config & FIFO_QUEUE) == 0) {
                    for (ForkJoinTask<?> t;;) {
                        if ((t = (ForkJoinTask<?>)U.getAndSetObject
                             (a, ((m & s) << ASHIFT) + ABASE, null)) == null)
                            break;
                        U.putOrderedInt(this, QTOP, s);
                        t.doExec();
                        if (base - (s = top - 1) > 0)
                            break;
                    }
                }
                else
                    pollAndExecAll();
            }
        }

        /**
         * Executes the given task and any remaining local tasks.
         */
        final void runTask(ForkJoinTask<?> task) {
            if (task != null) {
                // 记录当前的任务是偷来的
                scanState &= ~SCANNING; // mark as busy
                // doExec方法才是真正执行任务的关键，它是链接我们自定义compute方法的核心
                // ForkJoinTask-->doExec-->exec
                // 在RecursiveTask和RecursiveAction都重写exec方法，而exec方法里面调用的又是抽象方法
compute(这个compute方法就是我们在继承RecursiveTask或RecursiveAction时必须重写的方法)。
                // 到这里，已经看到本质了，绕了这么一大圈，终于和我们自己重写的compute方法联系到了一起
                (currentSteal = task).doExec();
                U.putOrderedObject(this, QCURRENTSTEAL, null); // release for GC
                execLocalTasks();
                ForkJoinWorkerThread thread = owner;
                // 累加偷来的数量
                if (++nsteals < 0)      // collect on overflow
                    transferStealCount(pool);
                // 任务执行完后，就重新更新scanState为SCANNING
                scanState |= SCANNING;
                if (thread != null)
                    thread.afterTopLevelExec();
            }
        }

        /**
         * Adds steal count to pool stealCounter if it exists, and resets.
         */
        final void transferStealCount(ForkJoinPool p) {
            AtomicLong sc;
            if (p != null && (sc = p.stealCounter) != null) {
                int s = nsteals;
                nsteals = 0;            // if negative, correct for overflow
                sc.getAndAdd((long)(s < 0 ? Integer.MAX_VALUE : s));
            }
        }

        /**
         * If present, removes from queue and executes the given task,
         * or any other cancelled task. Used only by awaitJoin.
         *
         * @return true if queue empty and task not known to be done
         */
        final boolean tryRemoveAndExec(ForkJoinTask<?> task) {
            ForkJoinTask<?>[] a; int m, s, b, n;
            if ((a = array) != null && (m = a.length - 1) >= 0 &&
                task != null) {
                while ((n = (s = top) - (b = base)) > 0) {
                    for (ForkJoinTask<?> t;;) {      // traverse from s to b
                        long j = ((--s & m) << ASHIFT) + ABASE;
                        if ((t = (ForkJoinTask<?>)U.getObject(a, j)) == null)
                            return s + 1 == top;     // shorter than expected
                        else if (t == task) {
                            boolean removed = false;
                            if (s + 1 == top) {      // pop
                                if (U.compareAndSwapObject(a, j, task, null)) {
                                    U.putOrderedInt(this, QTOP, s);
                                    removed = true;
                                }
                            }
                            else if (base == b)      // replace with proxy
                                removed = U.compareAndSwapObject(
                                    a, j, task, new EmptyTask());
                            if (removed)
                                task.doExec();
                            break;
```

```java
                }
                else if (t.status < 0 && s + 1 == top) {
                    if (U.compareAndSwapObject(a, j, t, null))
                        U.putOrderedInt(this, QTOP, s);
                    break;                  // was cancelled
                }
                if (--n == 0)
                    return false;
            }
            if (task.status < 0)
                return false;
        }
    }
    return true;
}

/**
 * Pops task if in the same CC computation as the given task,
 * in either shared or owned mode. Used only by helpComplete.
 */
final CountedCompleter<?> popCC(CountedCompleter<?> task, int mode) {
    int s; ForkJoinTask<?>[] a; Object o;
    if (base - (s = top) < 0 && (a = array) != null) {
        long j = (((a.length - 1) & (s - 1)) << ASHIFT) + ABASE;
        if ((o = U.getObjectVolatile(a, j)) != null &&
            (o instanceof CountedCompleter)) {
            CountedCompleter<?> t = (CountedCompleter<?>)o;
            for (CountedCompleter<?> r = t;;) {
                if (r == task) {
                    if (mode < 0) { // must lock
                        if (U.compareAndSwapInt(this, QLOCK, 0, 1)) {
                            if (top == s && array == a &&
                                U.compareAndSwapObject(a, j, t, null)) {
                                U.putOrderedInt(this, QTOP, s - 1);
                                U.putOrderedInt(this, QLOCK, 0);
                                return t;
                            }
                            U.compareAndSwapInt(this, QLOCK, 1, 0);
                        }
                    }
                    else if (U.compareAndSwapObject(a, j, t, null)) {
                        U.putOrderedInt(this, QTOP, s - 1);
                        return t;
                    }
                    break;
                }
                else if ((r = r.completer) == null) // try parent
                    break;
            }
        }
    }
    return null;
}
```

```java
        /**
         * Steals and runs a task in the same CC computation as the
         * given task if one exists and can be taken without
         * contention. Otherwise returns a checksum/control value for
         * use by method helpComplete.
         *
         * @return 1 if successful, 2 if retryable (lost to another
         * stealer), -1 if non-empty but no matching task found, else
         * the base index, forced negative.
         */
        final int pollAndExecCC(CountedCompleter<?> task) {
            int b, h; ForkJoinTask<?>[] a; Object o;
            if ((b = base) - top >= 0 || (a = array) == null)
                h = b | Integer.MIN_VALUE;  // to sense movement on re-poll
            else {
                long j = (((a.length - 1) & b) << ASHIFT) + ABASE;
                if ((o = U.getObjectVolatile(a, j)) == null)
                    h = 2;                  // retryable
                else if (!(o instanceof CountedCompleter))
                    h = -1;                 // unmatchable
                else {
                    CountedCompleter<?> t = (CountedCompleter<?>)o;
                    for (CountedCompleter<?> r = t;;) {
                        if (r == task) {
                            if (base == b &&
                                U.compareAndSwapObject(a, j, t, null)) {
                                base = b + 1;
                                t.doExec();
                                h = 1;      // success
                            }
                            else
                                h = 2;      // lost CAS
                            break;
                        }
                        else if ((r = r.completer) == null) {
                            h = -1;         // unmatched
                            break;
                        }
                    }
                }
            }
            return h;
        }

        /**
         * Returns true if owned and not known to be blocked.
         */
        final boolean isApparentlyUnblocked() {
            Thread wt; Thread.State s;
            return (scanState >= 0 &&
                    (wt = owner) != null &&
                    (s = wt.getState()) != Thread.State.BLOCKED &&
                    s != Thread.State.WAITING &&
                    s != Thread.State.TIMED_WAITING);
        }

        // Unsafe mechanics. Note that some are (and must be) the same as in FJP
        private static final sun.misc.Unsafe U;
        private static final int  ABASE;
        private static final int  ASHIFT;
        private static final long QTOP;
        private static final long QLOCK;
        private static final long QCURRENTSTEAL;
        static {
            try {
                U = sun.misc.Unsafe.getUnsafe();
                Class<?> wk = WorkQueue.class;
                Class<?> ak = ForkJoinTask[].class;
                QTOP = U.objectFieldOffset
                    (wk.getDeclaredField("top"));
                QLOCK = U.objectFieldOffset
                    (wk.getDeclaredField("qlock"));
                QCURRENTSTEAL = U.objectFieldOffset
                    (wk.getDeclaredField("currentSteal"));
                ABASE = U.arrayBaseOffset(ak);
                int scale = U.arrayIndexScale(ak);
                if ((scale & (scale - 1)) != 0)
                    throw new Error("data type scale not a power of two");
                ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
            } catch (Exception e) {
                throw new Error(e);
            }
        }
    }

    // static fields (initialized in static initializer below)

    /**
     * Creates a new ForkJoinWorkerThread. This factory is used unless
     * overridden in ForkJoinPool constructors.
     */
    public static final ForkJoinWorkerThreadFactory
        defaultForkJoinWorkerThreadFactory;

    /**
     * Permission required for callers of methods that may start or
     * kill threads.
     */
    private static final RuntimePermission modifyThreadPermission;

    /**
     * Common (static) pool. Non-null for public use unless a static
     * construction exception, but internal usages null-check on use
     * to paranoically avoid potential initialization circularities
     * as well as to simplify generated code.
```

```
     */
    static final ForkJoinPool common;

    /**
     * Common pool parallelism. To allow simpler use and management
     * when common pool threads are disabled, we allow the underlying
     * common.parallelism field to be zero, but in that case still report
     * parallelism as 1 to reflect resulting caller-runs mechanics.
     */
    static final int commonParallelism;

    /**
     * Limit on spare thread construction in tryCompensate.
     */
    private static int commonMaxSpares;

    /**
     * Sequence number for creating workerNamePrefix.
     */
    private static int poolNumberSequence;

    /**
     * Returns the next sequence number. We don't expect this to
     * ever contend, so use simple builtin sync.
     */
    private static final synchronized int nextPoolId() {
        return ++poolNumberSequence;
    }

    // static configuration constants

    /**
     * Initial timeout value (in nanoseconds) for the thread
     * triggering quiescence to park waiting for new work. On timeout,
     * the thread will instead try to shrink the number of
     * workers. The value should be large enough to avoid overly
     * aggressive shrinkage during most transient stalls (long GCs
     * etc).
     */
    private static final long IDLE_TIMEOUT = 2000L * 1000L * 1000L; // 2sec

    /**
     * Tolerance for idle timeouts, to cope with timer undershoots
     */
    private static final long TIMEOUT_SLOP = 20L * 1000L * 1000L;  // 20ms

    /**
     * The initial value for commonMaxSpares during static
     * initialization. The value is far in excess of normal
     * requirements, but also far short of MAX_CAP and typical
     * OS thread limits, so allows JVMs to catch misuse/abuse
     * before running out of resources needed to do so.
     */
```

```java
    private static final int DEFAULT_COMMON_MAX_SPARES = 256;

    /**
     * Number of times to spin-wait before blocking. The spins (in
     * awaitRunStateLock and awaitWork) currently use randomized
     * spins. Currently set to zero to reduce CPU usage.
     *
     * If greater than zero the value of SPINS must be a power
     * of two, at least 4.  A value of 2048 causes spinning for a
     * small fraction of typical context-switch times.
     *
     * If/when MWAIT-like intrinsics becomes available, they
     * may allow quieter spinning.
     */
    private static final int SPINS  = 0;

    /**
     * Increment for seed generators. See class ThreadLocal for
     * explanation.
     */
    private static final int SEED_INCREMENT = 0x9e3779b9;

    /*
     * Bits and masks for field ctl, packed with 4 16 bit subfields:
     * AC: Number of active running workers minus target parallelism
     * TC: Number of total workers minus target parallelism
     * SS: version count and status of top waiting thread
     * ID: poolIndex of top of Treiber stack of waiters
     *
     * When convenient, we can extract the lower 32 stack top bits
     * (including version bits) as sp=(int)ctl.  The offsets of counts
     * by the target parallelism and the positionings of fields makes
     * it possible to perform the most common checks via sign tests of
     * fields: When ac is negative, there are not enough active
     * workers, when tc is negative, there are not enough total
     * workers.  When sp is non-zero, there are waiting workers.  To
     * deal with possibly negative fields, we use casts in and out of
     * "short" and/or signed shifts to maintain signedness.
     *
     * Because it occupies uppermost bits, we can add one active count
     * using getAndAddLong of AC_UNIT, rather than CAS, when returning
     * from a blocked join.  Other updates entail multiple subfields
     * and masking, requiring CAS.
     */

    // Lower and upper word masks
    private static final long SP_MASK    = 0xffffffffL;
    private static final long UC_MASK    = ~SP_MASK;

    // Active counts 活跃线程数
    private static final int  AC_SHIFT   = 48;
    private static final long AC_UNIT    = 0x0001L << AC_SHIFT;
    private static final long AC_MASK    = 0xffffL << AC_SHIFT;
    // Total counts 总线程数
    private static final int  TC_SHIFT   = 32;
    private static final long TC_UNIT    = 0x0001L << TC_SHIFT;
    private static final long TC_MASK    = 0xffffL << TC_SHIFT;
    private static final long ADD_WORKER = 0x0001L << (TC_SHIFT + 15); // sign

    // runState有6种状态切换，按注释所言，只有SHUTDOWN状态是负数，其他都是整数，在并发环境更改状态必然要用到锁，ForkJoinPool对线程池加锁和解锁分别由lockRunState和unlockRunState来实现(这两个方法可以暂且不用深入理解，可以暂时跳过，只需要理解它们是帮助安全更改线程池状态的锁即可)
    // runState bits: SHUTDOWN must be negative, others arbitrary powers of two
    private static final int  RSLOCK     = 1;           //线程池被锁定
    private static final int  RSIGNAL    = 1 << 1;      //线程池有线程需要唤醒
    private static final int  STARTED    = 1 << 2;      //线程池已经初始化
    private static final int  STOP       = 1 << 29;     //线程池停止
    private static final int  TERMINATED = 1 << 30;     //线程池终止
    private static final int  SHUTDOWN   = 1 << 31;     //线程池关闭

    // Instance fields
    volatile long ctl;                   // main pool control
    volatile int runState;               // lockable status
    final int config;                    // parallelism, mode
    int indexSeed;                       // to generate worker index
    volatile WorkQueue[] workQueues;     // main registry
    final ForkJoinWorkerThreadFactory factory;
    final UncaughtExceptionHandler ueh;  // per-worker UEH
    final String workerNamePrefix;       // to create worker name string
    volatile AtomicLong stealCounter;    // also used as sync monitor

    /**
     * Acquires the runState lock; returns current (locked) runState.
     * 获取runState锁；返回当前（锁定）运行状态.
     * 从方法注释中看到，该方法一定会返回 locked 的 runState，也就是说一定会加锁成功
     */
    private int lockRunState() {
        int rs;
        // 因为RSLOCK=1，如果runState&RSLOCK==0，则说明目前没有加锁，进入或运算的下半段CAS
        // 先通过CAS尝试加锁，尝试成功直接返回，尝试失败则要调用awaitRunStateLock方法
        return ((((rs = runState) & RSLOCK) != 0 ||
                 !U.compareAndSwapInt(this, RUNSTATE, rs, rs |= RSLOCK)) ?
                awaitRunStateLock() : rs);
    }

    /**
     * Spins and/or blocks until runstate lock is available.  See
     * above for explanation.
     */
    private int awaitRunStateLock() {
        Object lock;
        boolean wasInterrupted = false;
        for (int spins = SPINS, r = 0, rs, ns;;) {
            // 判断是否加锁（==0表示未加锁）
            if (((rs = runState) & RSLOCK) == 0) {
                // 通过CAS加锁
                if (U.compareAndSwapInt(this, RUNSTATE, rs, ns = rs | RSLOCK)) {
                    if (wasInterrupted) {
                        try {
                            // 重置线程终端标记
                            Thread.currentThread().interrupt();
                        } catch (SecurityException ignore) {
                        }
                    }
                    // 加锁成功返回最新的runState，for循环的唯一正常出口
                    return ns;
                }
            }
            else if (r == 0)
                r = ThreadLocalRandom.nextSecondarySeed();
            else if (spins > 0) {
                r ^= r << 6; r ^= r >>> 21; r ^= r << 7; // xorshift
                if (r >= 0)
                    --spins;
            }
            // 如果是其他线程正在初始化占用锁，则调用yield方法让出CPU，让其快速初始化
            // 当完整的初始化ForkJoinPool时，直接利用了stealCounter这个原子变量，因为初始化时（调用
externalSubmit时），才会对StealCounter赋值。所以，这里的逻辑是，当状态不是STARTED或者stealCounter为
空，让出线程等待，也就是说，别的线程还没初始化完全，让其继续占用锁初始化即可
            else if ((rs & STARTED) == 0 || (lock = stealCounter) == null)
                Thread.yield();   // initialization race
            // 如果其它线程持有锁，并且线程池已经初始化，则将唤醒位标记为1
            // 不要让无限自旋尝试，如果资源不满足就等待，如果资源满足了就通知，所以，如果
(runState&RSIGNAL)==0成立，说明有线程需要唤醒，直接唤醒就好，否则也别浪费资源，主动等待一会
            else if (U.compareAndSwapInt(this, RUNSTATE, rs, rs | RSIGNAL)) {
                // 进入互斥锁
                synchronized (lock) {
                    // 再次判断，如果等于0，说明进入互斥锁前刚好有线程进行了唤醒，就不用等待，直接进行唤醒
操作即可，否则就进入等待
                    if ((runState & RSIGNAL) != 0) {
                        try {
                            lock.wait();
                        } catch (InterruptedException ie) {
                            if (!(Thread.currentThread() instanceof
                                  ForkJoinWorkerThread))
                                wasInterrupted = true;
                        }
                    }
                    else
                        lock.notifyAll();
                }
            }
        }
    }

    /**
     * Unlocks and sets runState to newRunState.
     *
     * @param oldRunState a value returned from lockRunState
     * @param newRunState the next value (must have lock bit clear).
     */
    private void unlockRunState(int oldRunState, int newRunState) {
        if (!U.compareAndSwapInt(this, RUNSTATE, oldRunState, newRunState)) {
            Object lock = stealCounter;
            runState = newRunState;              // clears RSIGNAL bit
            if (lock != null)
                synchronized (lock) { lock.notifyAll(); }
        }
    }

    // Creating, registering and deregistering workers

    /**
     * Tries to construct and start one worker. Assumes that total
     * count has already been incremented as a reservation.  Invokes
     * deregisterWorker on any failure.
     *
     * @return true if successful
     */
    private boolean createWorker() {
        ForkJoinWorkerThreadFactory fac = factory;
        Throwable ex = null;
        ForkJoinWorkerThread wt = null;
        try {
            // 如果工厂已经存在了，就用factory来创建线程，会去注册线程，这里的this就是ForkJoinPool对象
            // Worker线程是在fac.newThread中与WorkQueue关联上的。
            // newThread-->new ForkJoinWorkerThread-->ForkJoinPool.registerWorker
            // 而在DefaultForkJoinWorkerThreadFactory的newThread方法，会new一个
ForkJoinWorkerThread然后返回，而在创建ForkJoinWorkerThread的时候会将this也就是ForkJoinPool对象传给
ForkJoinWorkerThread；也就是说ForkJoinWorkerThread中会拥有ForkJoinPool对象。
            // 并且在ForkJoinWorkerThread的构造函数中，还会接受在registerWorker中创建的WorkQueue，
保存在自己的成员变量中。
            if (fac != null && (wt = fac.newThread(this)) != null) {
                // 启动线程
                // wt是ForkJoinWorkerThread，ForkJoinWorkerThread继承自Thread，调用start()方法
后，自然要调用自己重写的run()方法，
                // 而在ForkJoinWorkerThread重写的run方法中，又会调用ForkJoinPool.runWorker方法，
处理workQueue中的任务。方法的重点自然是进入到ForkJoinPool.runWorker方法中了。
                wt.start();
                return true;
            }
        } catch (Throwable rex) {
            ex = rex;
        }
        // 如果创建线程失败，就要逆向注销线程，包括前面对ctl等的操作
        deregisterWorker(wt, ex);
        return false;
    }

    /**
     * Tries to add one worker, incrementing ctl counts before doing
     * so, relying on createWorker to back out on failure.
     *
     * @param c incoming ctl value, with total count negative and no
     * idle workers.  On CAS failure, c is refreshed and retried if
     * this holds (otherwise, a new worker is not needed).
     */
    private void tryAddWorker(long c) {
        // 初始化添加worker表识
        boolean add = false;
        do {
            // 因为要添加Worker，所以AC和TC都要加一
            long nc = ((AC_MASK & (c + AC_UNIT)) |
                       (TC_MASK & (c + TC_UNIT)));
            // ctl还没被改变
            if (ctl == c) {
                int rs, stop;                 // check if terminating
                if ((stop = (rs = lockRunState()) & STOP) == 0)
                    // 更新ctl的值
                    add = U.compareAndSwapLong(this, CTL, c, nc);
                unlockRunState(rs, rs & ~RSLOCK);
                if (stop != 0)
                    break;
                // ctl值更新成功，开始真正的创建Worker
                if (add) {
                    createWorker();
                    break;
                }
            }
            // 重新获取ctl的值赋给c，并且没有达到最大线程数，并且没有空闲的线程
        } while (((c = ctl) & ADD_WORKER) != 0L && (int)c == 0);
    }

    /**
     * Callback from ForkJoinWorkerThread constructor to establish and
     * record its WorkQueue.
     *
     * @param wt the worker thread
     * @return the worker's queue
     */
    final WorkQueue registerWorker(ForkJoinWorkerThread wt) {
        UncaughtExceptionHandler handler;
        // 这里线程被设置为守护线程(所以ForkJoinPool中的线程为守护线程，当主线程结束后ForkJoinPool中的
线程也会退出，这一点和ThreadPool是不同的)
        wt.setDaemon(true);                           // configure thread
        if ((handler = ueh) != null)
            wt.setUncaughtExceptionHandler(handler);
        // 创建一个WorkQueue，并且设置当前WorkQueue的owner是当前线程
        WorkQueue w = new WorkQueue(this, wt);
        int i = 0;                                    // assign a pool index
        // 提取出WorkQueue的模式
        int mode = config & MODE_MASK;
        // 加锁
        int rs = lockRunState();
        try {
            WorkQueue[] ws; int n;                    // skip if no array
            // 判断ForkJoinPool的WorkQueue[]都初始化完全
            if ((ws = workQueues) != null && (n = ws.length) > 0) {
                // 一种魔数计算方式，用以减少冲突
                int s = indexSeed += SEED_INCREMENT;  // unlikely to collide
                // 假设WorkQueue的初始长度是16，那这里的m就是15，最终目的就是为了得到一个奇数
                int m = n - 1;
                // 和得到偶数的计算方式一样，得到一个小于m的奇数i
                i = ((s << 1) | 1) & m;               // odd-numbered indices
                // 如果这个槽位不为空，说明已经被其他线程初始化过了，也就是有冲突，选取别的槽位
                if (ws[i] != null) {                  // collision
                    int probes = 0;                   // step by approx half n
                    // 步长加2，也就保证step还是奇数
                    int step = (n <= 4) ? 2 : ((n >>> 1) & EVENMASK) + 2;
                    // 一直遍历，直到找到空槽位，如果都遍历了一遍，那就需要对WorkQueue[]扩容了
                    while (ws[i = (i + step) & m] != null) {
                        if (++probes >= n) {
                            workQueues = ws = Arrays.copyOf(ws, n <<= 1);
                            m = n - 1;
                            probes = 0;
                        }
                    }
                }
                // 初始化一个随机数
                w.hint = s;                           // use as random seed
                // config记录索引值和模式
                w.config = i | mode;
                // 扫描状态也记录为索引值
                w.scanState = i;                      // publication fence
                // 把初始化好的WorkQueue放到ForkJoinPool的WorkQueue[]数组中
                ws[i] = w;
            }
        } finally {
            // 解锁
            unlockRunState(rs, rs & ~RSLOCK);
        }
        // 设置worker的前缀名，用于业务区分
        wt.setName(workerNamePrefix.concat(Integer.toString(i >>> 1)));
        // 返回当前线程创建的WorkQueue，回到上一层调用栈，也就将WorkQueue注册到ForkJoinWorkerThread
里面了
        return w;
    }

    /**
     * Final callback from terminating worker, as well as upon failure
     * to construct or start a worker.  Removes record of worker from
     * array, and adjusts counts. If pool is shutting down, tries to
     * complete termination.
     *
     * @param wt the worker thread, or null if construction failed
     * @param ex the exception causing failure, or null if none
     */
```

```java
    final void deregisterWorker(ForkJoinWorkerThread wt, Throwable ex) {
        WorkQueue w = null;
        if (wt != null && (w = wt.workQueue) != null) {
            WorkQueue[] ws;                           // remove index from array
            int idx = w.config & SMASK;
            int rs = lockRunState();
            if ((ws = workQueues) != null && ws.length > idx && ws[idx] == w)
                ws[idx] = null;
            unlockRunState(rs, rs & ~RSLOCK);
        }
        long c;                                       // decrement counts
        do {} while (!U.compareAndSwapLong
                     (this, CTL, c = ctl, ((AC_MASK & (c - AC_UNIT)) |
                                           (TC_MASK & (c - TC_UNIT)) |
                                           (SP_MASK & c))));
        if (w != null) {
            w.qlock = -1;                             // ensure set
            w.transferStealCount(this);
            w.cancelAll();                            // cancel remaining tasks
        }
        for (;;) {                                    // possibly replace
            WorkQueue[] ws; int m, sp;
            if (tryTerminate(false, false) || w == null || w.array == null ||
                (runState & STOP) != 0 || (ws = workQueues) == null ||
                (m = ws.length - 1) < 0)              // already terminating
                break;
            if ((sp = (int)(c = ctl)) != 0) {         // wake up replacement
                if (tryRelease(c, ws[sp & m], AC_UNIT))
                    break;
            }
            else if (ex != null && (c & ADD_WORKER) != 0L) {
                tryAddWorker(c);                      // create replacement
                break;
            }
            else                                      // don't need replacement
                break;
        }
        if (ex == null)                               // help clean on way out
            ForkJoinTask.helpExpungeStaleExceptions();
        else                                          // rethrow
            ForkJoinTask.rethrow(ex);
    }

    // Signalling

    /**
     * Tries to create or activate a worker if too few are active.
     *
     * @param ws the worker array to use to find signallees
     * @param q a WorkQueue --if non-null, don't retry if now empty
     */
    final void signalWork(WorkQueue[] ws, WorkQueue q) {
        long c; int sp, i; WorkQueue v; Thread p;
        // ctl小于零，说明活动的线程数AC不够，由ForkJoinPool的构造函数可知，ctl初始化的值为-parallelism，即一开始的时候线程池中是没有活跃的线程的。
        while ((c = ctl) < 0L) {                       // too few active
            // 取ctl的低32位，如果为0，说明没有等待的线程
            if ((sp = (int)c) == 0) {                  // no idle workers
                // 取TC的高位，如果不等于0，则说明目前的工作着还没有达到并行度
                // 假设程序刚开始执行，那么活动线程数以及总线程数肯定都没达到并行度要求，这时就会调用tryAddWorker方法了
                if ((c & ADD_WORKER) != 0L)            // too few workers
                    // 添加Worker，也就是说要创建线程了
                    tryAddWorker(c);
                break;
            }
            // 未开始或者已停止，直接跳出
            if (ws == null)                            // unstarted/terminated
                break;
            // i=空闲线程栈顶端所属的工作队列索引
            if (ws.length <= (i = sp & SMASK))         // terminated
                break;
            if ((v = ws[i]) == null)                   // terminating
                break;
            // 程序执行到这里，说明有空闲线程，计算下一个scanState，增加了版本号，并且调整为active状态
            int vs = (sp + SS_SEQ) & ~INACTIVE;        // next scanState
            int d = sp - v.scanState;                  // screen CAS
            // 计算下一个ctl的值，活动线程数AC + 1，通过stackPred取得前一个WorkQueue的索引，重新设置回sp，形成最终的ctl值
            long nc = (UC_MASK & (c + AC_UNIT)) | (SP_MASK & v.stackPred);
            // 更新ctl的值
            if (d == 0 && U.compareAndSwapLong(this, CTL, c, nc)) {
                v.scanState = vs;                      // activate v
                // 如果有线程阻塞，则调用unpark唤醒即可
                if ((p = v.parker) != null)
                    U.unpark(p);
                break;
            }
            // 没有任务，直接跳出
            if (q != null && q.base == q.top)          // no more work
                break;
        }
    }

    /**
     * Signals and releases worker v if it is top of idle worker
     * stack.  This performs a one-shot version of signalWork only if
     * there is (apparently) at least one idle worker.
     *
     * @param c incoming ctl value
     * @param v if non-null, a worker
     * @param inc the increment to active count (zero when compensating)
     * @return true if successful
     */
    private boolean tryRelease(long c, WorkQueue v, long inc) {
        int sp = (int)c, vs = (sp + SS_SEQ) & ~INACTIVE; Thread p;
        if (v != null && v.scanState == sp) {          // v is at top of stack
            long nc = (UC_MASK & (c + inc)) | (SP_MASK & v.stackPred);
            if (U.compareAndSwapLong(this, CTL, c, nc)) {
                v.scanState = vs;
                if ((p = v.parker) != null)
                    U.unpark(p);
                return true;
            }
        }
        return false;
    }

    // Scanning for tasks

    /**
     * runWorker是很常规的三部曲操作：
     * scan: 通过扫描获取任务
     * runTask：执行扫描到的任务
     * awaitWork：没任务进入等待
     */
    final void runWorker(WorkQueue w) {
        // 初始化队列，并根据需要是否扩容为原来的2倍
        w.growArray();                   // allocate queue
        int seed = w.hint;               // initially holds randomization hint
        int r = (seed == 0) ? 1 : seed;  // avoid 0 for xorShift
        // 死循环更新偏移r，为扫描任务作准备
        for (ForkJoinTask<?> t;;) {
            // 扫描任务(ForkJoinPool的任务窃取机制就藏在scan方法中，另外虽然线程的启动肯定是伴随task提
交的，那既然有task提交那为啥以上来就要去偷任务呢？这是因为前面说的submission的task是放到WorkQueue数组的
「偶数」下标中，而线程时放在WorkQueue的「奇数」下标中，准确的说是只有奇数下标才有线程(worker)与之相对，所以
线程一启动就需要去偷取任务)
            if ((t = scan(w, r)) != null)
                // 扫描到就执行任务
                w.runTask(t);
            // 没扫描到将当前线程阻塞等待，如果等也等不到任务，那就跳出循环别死等了
            else if (!awaitWork(w, r))
                break;
            r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // xorshift
        }
    }

    /**
     * Scans for and tries to steal a top-level task. Scans start at a
     * random location, randomly moving on apparent contention,
     * otherwise continuing linearly until reaching two consecutive
     * empty passes over all queues with the same checksum (summing
     * each base index of each queue, that moves on each steal), at
     * which point the worker tries to inactivate and then re-scans,
     * attempting to re-activate (itself or some other worker) if
     * finding a task; otherwise returning null to await work.  Scans
     * otherwise touch as little memory as possible, to reduce
     * disruption on other scanning threads.
     *
     * @param w the worker (via its WorkQueue)
     * @param r a random seed
     * @return a task, or null if none found
     */
    private ForkJoinTask<?> scan(WorkQueue w, int r) {
        WorkQueue[] ws; int m;
        // 再次验证workQueue[]数组的初始化情况
        if ((ws = workQueues) != null && (m = ws.length - 1) > 0 && w != null) {
            // 获取当前扫描状态
            int ss = w.scanState;                     // initially non-negative
            // 随机一个起始位置，并赋值给k(死循环，注意到出口位置就好)
            for (int origin = r & m, k = origin, oldSum = 0, checkSum = 0;;) {
                WorkQueue q; ForkJoinTask<?>[] a; ForkJoinTask<?> t;
                int b, n; long c;
                // 如果k槽位不为空
                if ((q = ws[k]) != null) {
                    // base-top小于零，并且任务q不为空
                    if ((n = (b = q.base) - q.top) < 0 &&
                        (a = q.array) != null) {      // non-empty
                        // 获取base的偏移量，赋值给i
                        long i = (((a.length - 1) & b) << ASHIFT) + ABASE;
                        // 从base端获取任务，和前文的描述的steal搭配上了，是从base端steal
                        if ((t = ((ForkJoinTask<?>)
                                  U.getObjectVolatile(a, i))) != null &&
                            q.base == b) {
                            // 是active状态
                            if (ss >= 0) {
                                // 更新WorkQueue中数组i索引位置为空，并且更新base的值
                                if (U.compareAndSwapObject(a, i, t, null)) {
                                    q.base = b + 1;
                                    // n<-1,说明当前队列还有剩余任务，继续唤醒可能存在的其他线程
                                    if (n < -1)       // signal others
                                        signalWork(ws, q);
                                    // 直接返回任务
                                    return t;
                                }
                            }
                            else if (oldSum == 0 &&   // try to activate
                                     w.scanState < 0)
                                tryRelease(c = ctl, ws[m & (int)c], AC_UNIT);
                        }
                        // 如果获取任务失败，则准备换位置扫描
                        if (ss < 0)                   // refresh
                            ss = w.scanState;
                        r ^= r << 1; r ^= r >>> 3; r ^= r << 10;
                        origin = k = r & m;           // move and rescan
                        oldSum = checkSum = 0;
                        continue;
                    }
                    checkSum += b;
                }
                // k一直在变，扫描到最后，如果等于origin，说明已经扫描了一圈还没扫描到任务
                if ((k = (k + 1) & m) == origin) {    // continue until stable
                    if ((ss >= 0 || (ss == (ss = w.scanState))) &&
                        oldSum == (oldSum = checkSum)) {
                        if (ss < 0 || w.qlock < 0)    // already inactive
                            break;
                        // 准备inactive当前工作队列
                        int ns = ss | INACTIVE;       // try to inactivate
                        // 活动线程数AC减1
                        long nc = ((SP_MASK & ns) |
                                   (UC_MASK & ((c = ctl) - AC_UNIT)));
                        w.stackPred = (int)c;         // hold prev stack top
                        U.putInt(w, QSCANSTATE, ns);
                        if (U.compareAndSwapLong(this, CTL, c, nc))
                            ss = ns;
                        else
                            w.scanState = ss;         // back out
                    }
                    checkSum = 0;
                }
            }
        }
        return null;
    }

    /**
     * Possibly blocks worker w waiting for a task to steal, or
     * returns false if the worker should terminate.  If inactivating
     * w has caused the pool to become quiescent, checks for pool
     * termination, and, so long as this is not the only worker, waits
     * for up to a given duration.  On timeout, if ctl has not
     * changed, terminates the worker, which will in turn wake up
     * another worker to possibly repeat this process.
     *
     * @param w the calling worker
     * @param r a random seed (for spins)
     * @return false if the worker should terminate
     */
    private boolean awaitWork(WorkQueue w, int r) {
        if (w == null || w.qlock < 0)                 // w is terminating
            return false;
        for (int pred = w.stackPred, spins = SPINS, ss;;) {
            if ((ss = w.scanState) >= 0)
                break;
            else if (spins > 0) {
                r ^= r << 6; r ^= r >>> 21; r ^= r << 7;
                if (r >= 0 && --spins == 0) {         // randomize spins
                    WorkQueue v; WorkQueue[] ws; int s, j; AtomicLong sc;
                    if (pred != 0 && (ws = workQueues) != null &&
                        (j = pred & SMASK) < ws.length &&
                        (v = ws[j]) != null &&        // see if pred parking
                        (v.parker == null || v.scanState >= 0))
                        spins = SPINS;                // continue spinning
                }
            }
```

```
            else if (w.qlock < 0)                     // recheck after spins
                return false;
            else if (!Thread.interrupted()) {
                long c, prevctl, parkTime, deadline;
                int ac = (int)((c = ctl) >> AC_SHIFT) + (config & SMASK);
                if ((ac <= 0 && tryTerminate(false, false)) ||
                    (runState & STOP) != 0)           // pool terminating
                    return false;
                if (ac <= 0 && ss == (int)c) {        // is last waiter
                    prevctl = (UC_MASK & (c + AC_UNIT)) | (SP_MASK & pred);
                    int t = (short)(c >>> TC_SHIFT);  // shrink excess spares
                    if (t > 2 && U.compareAndSwapLong(this, CTL, c, prevctl))
                        return false;                 // else use timed wait
                    parkTime = IDLE_TIMEOUT * ((t >= 0) ? 1 : 1 - t);
                    deadline = System.nanoTime() + parkTime - TIMEOUT_SLOP;
                }
                else
                    prevctl = parkTime = deadline = 0L;
                Thread wt = Thread.currentThread();
                U.putObject(wt, PARKBLOCKER, this);   // emulate LockSupport
                w.parker = wt;
                if (w.scanState < 0 && ctl == c)      // recheck before park
                    U.park(false, parkTime);
                U.putOrderedObject(w, QPARKER, null);
                U.putObject(wt, PARKBLOCKER, null);
                if (w.scanState >= 0)
                    break;
                if (parkTime != 0L && ctl == c &&
                    deadline - System.nanoTime() <= 0L &&
                    U.compareAndSwapLong(this, CTL, c, prevctl))
                    return false;                     // shrink pool
            }
        }
        return true;
    }

    // Joining tasks

    /**
     * Tries to steal and run tasks within the target's computation.
     * Uses a variant of the top-level algorithm, restricted to tasks
     * with the given task as ancestor: It prefers taking and running
     * eligible tasks popped from the worker's own queue (via
     * popCC). Otherwise it scans others, randomly moving on
     * contention or execution, deciding to give up based on a
     * checksum (via return codes frob pollAndExecCC). The maxTasks
     * argument supports external usages; internal calls use zero,
     * allowing unbounded steps (external calls trap non-positive
     * values).
     *
     * @param w caller
     * @param maxTasks if non-zero, the maximum number of other tasks to run
     * @return task status on exit
     */
    final int helpComplete(WorkQueue w, CountedCompleter<?> task,
                           int maxTasks) {
        WorkQueue[] ws; int s = 0, m;
        if ((ws = workQueues) != null && (m = ws.length - 1) >= 0 &&
            task != null && w != null) {
            int mode = w.config;                 // for popCC
            int r = w.hint ^ w.top;              // arbitrary seed for origin
            int origin = r & m;                  // first queue to scan
            int h = 1;                           // 1:ran, >1:contended, <0:hash
            for (int k = origin, oldSum = 0, checkSum = 0;;) {
                CountedCompleter<?> p; WorkQueue q;
                if ((s = task.status) < 0)
                    break;
                if (h == 1 && (p = w.popCC(task, mode)) != null) {
                    p.doExec();                  // run local task
                    if (maxTasks != 0 && --maxTasks == 0)
                        break;
                    origin = k;                  // reset
                    oldSum = checkSum = 0;
                }
                else {                           // poll other queues
                    if ((q = ws[k]) == null)
                        h = 0;
                    else if ((h = q.pollAndExecCC(task)) < 0)
                        checkSum += h;
                    if (h > 0) {
                        if (h == 1 && maxTasks != 0 && --maxTasks == 0)
                            break;
                        r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // xorshift
                        origin = k = r & m;      // move and restart
                        oldSum = checkSum = 0;
                    }
                    else if ((k = (k + 1) & m) == origin) {
                        if (oldSum == (oldSum = checkSum))
                            break;
                        checkSum = 0;
                    }
                }
            }
        }
        return s;
    }

    /**
     * Tries to locate and execute tasks for a stealer of the given
     * task, or in turn one of its stealers, Traces currentSteal ->
     * currentJoin links looking for a thread working on a descendant
     * of the given task and with a non-empty queue to steal back and
     * execute tasks from. The first call to this method upon a
     * waiting join will often entail scanning/search, (which is OK
     * because the joiner has nothing better to do), but this method
     * leaves hints in workers to speed up subsequent calls.
```

```java
     *
     * @param w caller
     * @param task the task to join
     */
    private void helpStealer(WorkQueue w, ForkJoinTask<?> task) {
        WorkQueue[] ws = workQueues;
        int oldSum = 0, checkSum, m;
        if (ws != null && (m = ws.length - 1) >= 0 && w != null &&
            task != null) {
            do {                                       // restart point
                checkSum = 0;                          // for stability check
                ForkJoinTask<?> subtask;
                WorkQueue j = w, v;                    // v is subtask stealer
                descent: for (subtask = task; subtask.status >= 0; ) {
                    for (int h = j.hint | 1, k = 0, i; ; k += 2) {
                        if (k > m)                     // can't find stealer
                            break descent;
                        if ((v = ws[i = (h + k) & m]) != null) {
                            if (v.currentSteal == subtask) {
                                j.hint = i;
                                break;
                            }
                            checkSum += v.base;
                        }
                    }
                    for (;;) {                         // help v or descend
                        ForkJoinTask<?>[] a; int b;
                        checkSum += (b = v.base);
                        ForkJoinTask<?> next = v.currentJoin;
                        if (subtask.status < 0 || j.currentJoin != subtask ||
                            v.currentSteal != subtask) // stale
                            break descent;
                        if (b - v.top >= 0 || (a = v.array) == null) {
                            if ((subtask = next) == null)
                                break descent;
                            j = v;
                            break;
                        }
                        int i = (((a.length - 1) & b) << ASHIFT) + ABASE;
                        ForkJoinTask<?> t = ((ForkJoinTask<?>)
                                             U.getObjectVolatile(a, i));
                        if (v.base == b) {
                            if (t == null)             // stale
                                break descent;
                            if (U.compareAndSwapObject(a, i, t, null)) {
                                v.base = b + 1;
                                ForkJoinTask<?> ps = w.currentSteal;
                                int top = w.top;
                                do {
                                    U.putOrderedObject(w, QCURRENTSTEAL, t);
                                    t.doExec();        // clear local tasks too
                                } while (task.status >= 0 &&
                                         w.top != top &&
                                         (t = w.pop()) != null);
                                U.putOrderedObject(w, QCURRENTSTEAL, ps);
                                if (w.base != w.top)
                                    return;            // can't further help
                            }
                        }
                    }
                }
            } while (task.status >= 0 && oldSum != (oldSum = checkSum));
        }
    }

    /**
     * Tries to decrement active count (sometimes implicitly) and
     * possibly release or create a compensating worker in preparation
     * for blocking. Returns false (retryable by caller), on
     * contention, detected staleness, instability, or termination.
     *
     * @param w caller
     */
    private boolean tryCompensate(WorkQueue w) {
        boolean canBlock;
        WorkQueue[] ws; long c; int m, pc, sp;
        if (w == null || w.qlock < 0 ||           // caller terminating
            (ws = workQueues) == null || (m = ws.length - 1) <= 0 ||
            (pc = config & SMASK) == 0)           // parallelism disabled
            canBlock = false;
        else if ((sp = (int)(c = ctl)) != 0)      // release idle worker
            canBlock = tryRelease(c, ws[sp & m], 0L);
        else {
            int ac = (int)(c >> AC_SHIFT) + pc;
            int tc = (short)(c >> TC_SHIFT) + pc;
            int nbusy = 0;                        // validate saturation
            for (int i = 0; i <= m; ++i) {        // two passes of odd indices
                WorkQueue v;
                if ((v = ws[((i << 1) | 1) & m]) != null) {
                    if ((v.scanState & SCANNING) != 0)
                        break;
                    ++nbusy;
                }
            }
            if (nbusy != (tc << 1) || ctl != c)
                canBlock = false;                 // unstable or stale
            else if (tc >= pc && ac > 1 && w.isEmpty()) {
                long nc = ((AC_MASK & (c - AC_UNIT)) |
                           (~AC_MASK & c));       // uncompensated
                canBlock = U.compareAndSwapLong(this, CTL, c, nc);
            }
            else if (tc >= MAX_CAP ||
                     (this == common && tc >= pc + commonMaxSpares))
                throw new RejectedExecutionException(
                    "Thread limit exceeded replacing blocked worker");
            else {                                // similar to tryAddWorker
```

```java
                boolean add = false; int rs;      // CAS within lock
                long nc = ((AC_MASK & c) |
                           (TC_MASK & (c + TC_UNIT)));
                if (((rs = lockRunState()) & STOP) == 0)
                    add = U.compareAndSwapLong(this, CTL, c, nc);
                unlockRunState(rs, rs & ~RSLOCK);
                canBlock = add && createWorker(); // throws on exception
            }
        }
        return canBlock;
    }

    /**
     * Helps and/or blocks until the given task is done or timeout.
     *
     * @param w caller
     * @param task the task
     * @param deadline for timed waits, if nonzero
     * @return task status on exit
     */
    final int awaitJoin(WorkQueue w, ForkJoinTask<?> task, long deadline) {
        int s = 0;
        if (task != null && w != null) {
            ForkJoinTask<?> prevJoin = w.currentJoin;
            U.putOrderedObject(w, QCURRENTJOIN, task);
            CountedCompleter<?> cc = (task instanceof CountedCompleter) ?
                (CountedCompleter<?>)task : null;
            for (;;) {
                if ((s = task.status) < 0)
                    break;
                if (cc != null)
                    helpComplete(w, cc, 0);
                else if (w.base == w.top || w.tryRemoveAndExec(task))
                    helpStealer(w, task);
                if ((s = task.status) < 0)
                    break;
                long ms, ns;
                if (deadline == 0L)
                    ms = 0L;
                else if ((ns = deadline - System.nanoTime()) <= 0L)
                    break;
                else if ((ms = TimeUnit.NANOSECONDS.toMillis(ns)) <= 0L)
                    ms = 1L;
                if (tryCompensate(w)) {
                    task.internalWait(ms);
                    U.getAndAddLong(this, CTL, AC_UNIT);
                }
            }
            U.putOrderedObject(w, QCURRENTJOIN, prevJoin);
        }
        return s;
    }
    // Specialized scanning

    /**
     * Returns a (probably) non-empty steal queue, if one is found
     * during a scan, else null.  This method must be retried by
     * caller if, by the time it tries to use the queue, it is empty.
     */
    private WorkQueue findNonEmptyStealQueue() {
        WorkQueue[] ws; int m;  // one-shot version of scan loop
        int r = ThreadLocalRandom.nextSecondarySeed();
        if ((ws = workQueues) != null && (m = ws.length - 1) >= 0) {
            for (int origin = r & m, k = origin, oldSum = 0, checkSum = 0;;) {
                WorkQueue q; int b;
                if ((q = ws[k]) != null) {
                    if ((b = q.base) - q.top < 0)
                        return q;
                    checkSum += b;
                }
                if ((k = (k + 1) & m) == origin) {
                    if (oldSum == (oldSum = checkSum))
                        break;
                    checkSum = 0;
                }
            }
        }
        return null;
    }

    /**
     * Runs tasks until {@code isQuiescent()}. We piggyback on
     * active count ctl maintenance, but rather than blocking
     * when tasks cannot be found, we rescan until all others cannot
     * find tasks either.
     */
    final void helpQuiescePool(WorkQueue w) {
        ForkJoinTask<?> ps = w.currentSteal; // save context
        for (boolean active = true;;) {
            long c; WorkQueue q; ForkJoinTask<?> t; int b;
            w.execLocalTasks();     // run locals before each scan
            if ((q = findNonEmptyStealQueue()) != null) {
                if (!active) {      // re-establish active count
                    active = true;
                    U.getAndAddLong(this, CTL, AC_UNIT);
                }
                if ((b = q.base) - q.top < 0 && (t = q.pollAt(b)) != null) {
                    U.putOrderedObject(w, QCURRENTSTEAL, t);
                    t.doExec();
                    if (++w.nsteals < 0)
                        w.transferStealCount(this);
                }
            }
            else if (active) {      // decrement active count without queuing
                long nc = (AC_MASK & ((c = ctl) - AC_UNIT)) | (~AC_MASK & c);
                if ((int)(nc >> AC_SHIFT) + (config & SMASK) <= 0)
                    break;          // bypass decrement-then-increment
                if (U.compareAndSwapLong(this, CTL, c, nc))
                    active = false;
            }
            else if ((int)((c = ctl) >> AC_SHIFT) + (config & SMASK) <= 0 &&
                     U.compareAndSwapLong(this, CTL, c, c + AC_UNIT))
                break;
        }
        U.putOrderedObject(w, QCURRENTSTEAL, ps);
    }

    /**
     * Gets and removes a local or stolen task for the given worker.
     *
     * @return a task, if available
     */
    final ForkJoinTask<?> nextTaskFor(WorkQueue w) {
        for (ForkJoinTask<?> t;;) {
            WorkQueue q; int b;
            if ((t = w.nextLocalTask()) != null)
                return t;
            if ((q = findNonEmptyStealQueue()) == null)
                return null;
            if ((b = q.base) - q.top < 0 && (t = q.pollAt(b)) != null)
                return t;
        }
    }

    /**
     * Returns a cheap heuristic guide for task partitioning when
     * programmers, frameworks, tools, or languages have little or no
     * idea about task granularity.  In essence, by offering this
     * method, we ask users only about tradeoffs in overhead vs
     * expected throughput and its variance, rather than how finely to
     * partition tasks.
     *
     * In a steady state strict (tree-structured) computation, each
     * thread makes available for stealing enough tasks for other
     * threads to remain active. Inductively, if all threads play by
     * the same rules, each thread should make available only a
     * constant number of tasks.
     *
     * The minimum useful constant is just 1. But using a value of 1
     * would require immediate replenishment upon each steal to
     * maintain enough tasks, which is infeasible.  Further,
     * partitionings/granularities of offered tasks should minimize
     * steal rates, which in general means that threads nearer the top
     * of computation tree should generate more than those nearer the
     * bottom. In perfect steady state, each thread is at
     * approximately the same level of computation tree. However,
     * producing extra tasks amortizes the uncertainty of progress and
     * diffusion assumptions.
     *
     * So, users will want to use values larger (but not much larger)
     * than 1 to both smooth over transient shortages and hedge
     * against uneven progress; as traded off against the cost of
     * extra task overhead. We leave the user to pick a threshold
     * value to compare with the results of this call to guide
     * decisions, but recommend values such as 3.
     *
     * When all threads are active, it is on average OK to estimate
     * surplus strictly locally. In steady-state, if one thread is
     * maintaining say 2 surplus tasks, then so are others. So we can
     * just use estimated queue length.  However, this strategy alone
     * leads to serious mis-estimates in some non-steady-state
     * conditions (ramp-up, ramp-down, other stalls). We can detect
     * many of these by further considering the number of "idle"
     * threads, that are known to have zero queued tasks, so
     * compensate by a factor of (#idle/#active) threads.
     */
    static int getSurplusQueuedTaskCount() {
        Thread t; ForkJoinWorkerThread wt; ForkJoinPool pool; WorkQueue q;
        if (((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)) {
            int p = (pool = (wt = (ForkJoinWorkerThread)t).pool).
                config & SMASK;
            int n = (q = wt.workQueue).top - q.base;
            int a = (int)(pool.ctl >> AC_SHIFT) + p;
            return n - (a > (p >>>= 1) ? 0 :
                        a > (p >>>= 1) ? 1 :
                        a > (p >>>= 1) ? 2 :
                        a > (p >>>= 1) ? 4 :
                        8);
        }
        return 0;
    }

    //  Termination

    /**
     * Possibly initiates and/or completes termination.
     *
     * @param now if true, unconditionally terminate, else only
     * if no work and no active workers
     * @param enable if true, enable shutdown when next possible
     * @return true if now terminating or terminated
     */
    private boolean tryTerminate(boolean now, boolean enable) {
        int rs;
        if (this == common)                       // cannot shut down
            return false;
        if ((rs = runState) >= 0) {
            if (!enable)
                return false;
            rs = lockRunState();                  // enter SHUTDOWN phase
            unlockRunState(rs, (rs & ~RSLOCK) | SHUTDOWN);
        }

        if ((rs & STOP) == 0) {
            if (!now) {                           // check quiescence
                for (long oldSum = 0L;;) {        // repeat until stable
                    WorkQueue[] ws; WorkQueue w; int m, b; long c;
                    long checkSum = ctl;
                    if ((int)(checkSum >> AC_SHIFT) + (config & SMASK) > 0)
                        return false;             // still active workers
                    if ((ws = workQueues) == null || (m = ws.length - 1) <= 0)
                        break;                    // check queues
                    for (int i = 0; i <= m; ++i) {
                        if ((w = ws[i]) != null) {
                            if ((b = w.base) != w.top || w.scanState >= 0 ||
                                w.currentSteal != null) {
                                tryRelease(c = ctl, ws[m & (int)c], AC_UNIT);
                                return false;     // arrange for recheck
                            }
                            checkSum += b;
                            if ((i & 1) == 0)
                                w.qlock = -1;     // try to disable external
                        }
                    }
                    if (oldSum == (oldSum = checkSum))
                        break;
                }
            }
            if ((runState & STOP) == 0) {
                rs = lockRunState();              // enter STOP phase
                unlockRunState(rs, (rs & ~RSLOCK) | STOP);
            }
        }

        int pass = 0;                             // 3 passes to help terminate
        for (long oldSum = 0L;;) {                // or until done or stable
            WorkQueue[] ws; WorkQueue w; ForkJoinWorkerThread wt; int m;
            long checkSum = ctl;
            if ((short)(checkSum >>> TC_SHIFT) + (config & SMASK) <= 0 ||
                (ws = workQueues) == null || (m = ws.length - 1) <= 0) {
                if ((runState & TERMINATED) == 0) {
                    rs = lockRunState();          // done
                    unlockRunState(rs, (rs & ~RSLOCK) | TERMINATED);
                    synchronized (this) { notifyAll(); } // for awaitTermination
                }
                break;
            }
            for (int i = 0; i <= m; ++i) {
                if ((w = ws[i]) != null) {
                    checkSum += w.base;
                    w.qlock = -1;                 // try to disable
                    if (pass > 0) {
                        w.cancelAll();            // clear queue
                        if (pass > 1 && (wt = w.owner) != null) {
```

```
                        if (!wt.isInterrupted()) {
                            try {             // unblock join
                                wt.interrupt();
                            } catch (Throwable ignore) {
                            }
                        }
                        if (w.scanState < 0)
                            U.unpark(wt);     // wake up
                    }
                }
            }
        }
        if (checkSum != oldSum) {             // unstable
            oldSum = checkSum;
            pass = 0;
        }
        else if (pass > 3 && pass > m)        // can't further help
            break;
        else if (++pass > 1) {                // try to dequeue
            long c; int j = 0, sp;            // bound attempts
            while (j++ <= m && (sp = (int)(c = ctl)) != 0)
                tryRelease(c, ws[sp & m], AC_UNIT);
        }
    }
    return true;
}

// External operations

/**
 * Full version of externalPush, handling uncommon cases, as well
 * as performing secondary initialization upon the first
 * submission of the first task to the pool.  It also detects
 * first submission by an external thread and creates a new shared
 * queue if the one at index if empty or contended.
 *
 * @param task the task. Caller must ensure non-null.
 */
private void externalSubmit(ForkJoinTask<?> task) {
    int r;                                    // initialize caller's probe
    // 生成随机数
    if ((r = ThreadLocalRandom.getProbe()) == 0) {
        ThreadLocalRandom.localInit();
        r = ThreadLocalRandom.getProbe();
    }
    for (;;) {
        WorkQueue[] ws; WorkQueue q; int rs, m, k;
        boolean move = false;
        // 如果线程池的状态为终止状态，则帮助终止
        if ((rs = runState) < 0) {
            tryTerminate(false, false);     // help terminate
            throw new RejectedExecutionException();
        }
```

```
            // 再判断一次状态是否为初始化，因为在lockRunState过程中有可能状态被别的线程更改了
            else if ((rs & STARTED) == 0 ||     // initialize
                     ((ws = workQueues) == null || (m = ws.length - 1) < 0)) {
                int ns = 0;
                // 加锁
                rs = lockRunState();
                try {
                    if ((rs & STARTED) == 0) {
                        // 初始化stealcounter的值（任务窃取计数器，原子变量）
                        U.compareAndSwapObject(this, STEALCOUNTER, null,
                                               new AtomicLong());
                        // create workQueues array with size a power of two
                        // 取config的低16位（确切说是低15位），获取并行度
                        int p = config & SMASK; // ensure at least 2 slots
                        // 简单描述这个过程，就是根据不同的并行度来初始化不同大小的WorkQueue[]数组，数
组大小要求是2的n次幂，所以用表格直观理解一下并行度和队列容量的关系：
                        // 并行度p      容量
                        // 1，2         4
                        // 3，4         8
                        // 5 ～ 8       16
                        // 9 ～ 16      32
                        int n = (p > 1) ? p - 1 : 1;
                        n |= n >>> 1; n |= n >>> 2;  n |= n >>> 4;
                        n |= n >>> 8; n |= n >>> 16; n = (n + 1) << 1;
                        // 初始化WorkQueue数组
                        workQueues = new WorkQueue[n];
                        // 标记初始化完成
                        ns = STARTED;
                    }
                } finally {
                    // 解锁
                    unlockRunState(rs, (rs & ~RSLOCK) | ns);
                }
            }
            // 偶数位槽位，将任务放进偶数槽位(槽位不为空，即该槽位已经存在了WorkQueue，其实WorkQueue是
个二维数组，WorkQueue的每个槽位又是一个WorkQueue)
            else if ((q = ws[k = r & m & SQMASK]) != null) {
                // 对WorkQueue加锁
                if (q.qlock == 0 && U.compareAndSwapInt(q, QLOCK, 0, 1)) {
                    ForkJoinTask<?>[] a = q.array;
                    int s = q.top;
                    // 初始化任务提交标识
                    boolean submitted = false; // initial submission or resizing
                    try {                      // locked version of push
                        // 计算内存偏移量，放任务，更新top值
                        if ((a != null && a.length > s + 1 - q.base) ||
                            (a = q.growArray()) != null) {
                            int j = (((a.length - 1) & s) << ASHIFT) + ABASE;
                            U.putOrderedObject(a, j, task);
                            U.putOrderedInt(q, QTOP, s + 1);
                            // 提交任务成功
                            submitted = true;
                        }
                    } finally {
                        // WorkQueue解锁
                        U.compareAndSwapInt(q, QLOCK, 1, 0);
                    }
                    // 任务提交成功了
                    if (submitted) {
                        // 自然要唤醒可能存在等待的线程来处理任务了
                        signalWork(ws, q);
                        return;
                    }
                }
                // 任务提交没成功，可以重新计算随机数，再走一次流程
                move = true;                   // move on failure
            }
            // 如果找到的槽位是空，则要初始化一个WorkQueue
            else if (((rs = runState) & RSLOCK) == 0) { // create new queue
                q = new WorkQueue(this, null);
                // 设置工作队列的窃取线索值
                q.hint = r;
                // 如上面WorkQueue中config的介绍，记录当前WorkQueue在WorkQueue[]数组中的值，和队列
    模式
                q.config = k | SHARED_QUEUE;
                q.scanState = INACTIVE;
                rs = lockRunState();           // publish index
                if (rs > 0 &&  (ws = workQueues) != null &&
                    k < ws.length && ws[k] == null)
                    ws[k] = q;                 // else terminated
                // 解锁
                unlockRunState(rs, rs & ~RSLOCK);
            }
            else
                move = true;                   // move if busy
            if (move)
                r = ThreadLocalRandom.advanceProbe(r);
        }
        // 第一次执行这个方法内部的逻辑顺序应该是 Flag1--> Flag3-->Flag2
        // externalSubmit如果任务成功提交，就会调用signalWork方法了
    }

    /**
     * invoke/submit/execute方法都会调用 externalPush(task) 这个用法
     * 前面说过，task会细分成submission task和worker task，worker task是fork出来的，那从这个入口进入
    的，自然也就是submission task了，也就是说：
     * 通过invoke() | submit() | execute()等方法提交的task, 是submission task，会放到WorkQueue数
    组的偶数索引位置
     * 调用fork()方法生成出的任务，叫worker task，会放到WorkQueue数组的奇数索引位置
     *
     * @param task the task. Caller must ensure non-null.
     */
    final void externalPush(ForkJoinTask<?> task) {
        WorkQueue[] ws; WorkQueue q; int m;
        // 通过ThreadLocalRandom产生随机数，用于下面计算槽位索引
```

```
        // ThreadLocalRandom是ThreadLocal的衍生物，每个线程默认的probe是0，当线程调用
ThreadLocalRandom.current()时，会初始化seed和probe，维护在线程内部，这里就知道是生成一个随机数就好
        int r = ThreadLocalRandom.getProbe();
        int rs = runState;
        // 如果ForkJoinPool中的WorkQueue数组已经完成初始化，且根据随机数定位的index存在workQueue,且
cas的方式加锁成功
        // 二进制为：0000 0000 0000 0000 0000 0000 0111 1110
        // static final int SQMASK       = 0x007e;        // max 64 (even) slots
        // m的值代表WorkQueue数组的最大下表
        // m & r 会保证随机数r大于m的部分不可用
        // m & r & SQMASK 因为SQMASK最后一位是0，最终的结果就会是偶数
        // r != 0 说明当前线程已经初始化过一些内容
        // rs > 0 说明ForkJoinPool的runState也已经被初始化过
        if ((ws = workQueues) != null && (m = (ws.length - 1)) >= 0 &&
            (q = ws[m & r & SQMASK]) != null && r != 0 && rs > 0 &&
            // 对WorkQueue操作加锁
            U.compareAndSwapInt(q, QLOCK, 0, 1)) {
            ForkJoinTask<?>[] a; int am, n, s;
            // WorkQueue中的任务数组不为空
            if ((a = q.array) != null &&
                (am = a.length - 1) > (n = (s = q.top) - q.base)) { // 数组长度大于任务个数，不
需要扩容
                int j = ((am & s) << ASHIFT) + ABASE; // WorkQueue中的任务数组不为空
                U.putOrderedObject(a, j, task); // 向Queue中放入任务
                U.putOrderedInt(q, QTOP, s + 1); // top值加一
                U.putIntVolatile(q, QLOCK, 0); // 对WorkQueue操作解锁
                // 任务个数小于等于1，那么此槽位上的线程有可能等待，如果大家都没任务，可能都在等待，新任务
来了，唤醒，起来干活了
                if (n <= 1)
                    // 唤醒可能存在等待的线程
                    signalWork(ws, q);
                return;
            }
            // 任务入队失败，前面加锁了，这里也要解锁
            U.compareAndSwapInt(q, QLOCK, 1, 0);
        }
        // 不满足上述条件，也就是说上面的这些WorkQueue[]等都不存在，就要通过这个方法一切从头开始创建
        externalSubmit(task);
    }

    /**
     * Returns common pool queue for an external thread.
     */
    static WorkQueue commonSubmitterQueue() {
        ForkJoinPool p = common;
        int r = ThreadLocalRandom.getProbe();
        WorkQueue[] ws; int m;
        return (p != null && (ws = p.workQueues) != null &&
                (m = ws.length - 1) >= 0) ?
            ws[m & r & SQMASK] : null;
    }

    /**
```

```
     * Performs tryUnpush for an external submitter: Finds queue,
     * locks if apparently non-empty, validates upon locking, and
     * adjusts top. Each check can fail but rarely does.
     */
    final boolean tryExternalUnpush(ForkJoinTask<?> task) {
        WorkQueue[] ws; WorkQueue w; ForkJoinTask<?>[] a; int m, s;
        int r = ThreadLocalRandom.getProbe();
        if ((ws = workQueues) != null && (m = ws.length - 1) >= 0 &&
            (w = ws[m & r & SQMASK]) != null &&
            (a = w.array) != null && (s = w.top) != w.base) {
            long j = (((a.length - 1) & (s - 1)) << ASHIFT) + ABASE;
            if (U.compareAndSwapInt(w, QLOCK, 0, 1)) {
                if (w.top == s && w.array == a &&
                    U.getObject(a, j) == task &&
                    U.compareAndSwapObject(a, j, task, null)) {
                    U.putOrderedInt(w, QTOP, s - 1);
                    U.putOrderedInt(w, QLOCK, 0);
                    return true;
                }
                U.compareAndSwapInt(w, QLOCK, 1, 0);
            }
        }
        return false;
    }

    /**
     * Performs helpComplete for an external submitter.
     */
    final int externalHelpComplete(CountedCompleter<?> task, int maxTasks) {
        WorkQueue[] ws; int n;
        int r = ThreadLocalRandom.getProbe();
        return ((ws = workQueues) == null || (n = ws.length) == 0) ? 0 :
            helpComplete(ws[(n - 1) & r & SQMASK], task, maxTasks);
    }

    // Exported methods

    // Constructors

    /**
     * Creates a {@code ForkJoinPool} with parallelism equal to {@link
     * java.lang.Runtime#availableProcessors}, using the {@linkplain
     * #defaultForkJoinWorkerThreadFactory default thread factory},
     * no UncaughtExceptionHandler, and non-async LIFO processing mode.
     *
     * @throws SecurityException if a security manager exists and
     *         the caller is not permitted to modify threads
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")}
     */
    public ForkJoinPool() {
        this(Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors()),
             defaultForkJoinWorkerThreadFactory, null, false);
```

```java
    }

    /**
     * Creates a {@code ForkJoinPool} with the indicated parallelism
     * level, the {@linkplain
     * #defaultForkJoinWorkerThreadFactory default thread factory},
     * no UncaughtExceptionHandler, and non-async LIFO processing mode.
     *
     * @param parallelism the parallelism level
     * @throws IllegalArgumentException if parallelism less than or
     *         equal to zero, or greater than implementation limit
     * @throws SecurityException if a security manager exists and
     *         the caller is not permitted to modify threads
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")}
     */
    public ForkJoinPool(int parallelism) {
        this(parallelism, defaultForkJoinWorkerThreadFactory, null, false);
    }

    /**
     * Creates a {@code ForkJoinPool} with the given parameters.
     *
     * @param parallelism the parallelism level. For default value,
     * use {@link java.lang.Runtime#availableProcessors}.
     * @param factory the factory for creating new threads. For default value,
     * use {@link #defaultForkJoinWorkerThreadFactory}.
     * @param handler the handler for internal worker threads that
     * terminate due to unrecoverable errors encountered while executing
     * tasks. For default value, use {@code null}.
     * @param asyncMode if true,
     * establishes local first-in-first-out scheduling mode for forked
     * tasks that are never joined. This mode may be more appropriate
     * than default locally stack-based mode in applications in which
     * worker threads only process event-style asynchronous tasks.
     * For default value, use {@code false}.
     * @throws IllegalArgumentException if parallelism less than or
     *         equal to zero, or greater than implementation limit
     * @throws NullPointerException if the factory is null
     * @throws SecurityException if a security manager exists and
     *         the caller is not permitted to modify threads
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")}
     */
    public ForkJoinPool(int parallelism,
                        ForkJoinWorkerThreadFactory factory,
                        UncaughtExceptionHandler handler,
                        boolean asyncMode) {
        this(checkParallelism(parallelism),
             checkFactory(factory),
             handler,
             asyncMode ? FIFO_QUEUE : LIFO_QUEUE,
             "ForkJoinPool-" + nextPoolId() + "-worker-");
        checkPermission();
    }

    private static int checkParallelism(int parallelism) {
        if (parallelism <= 0 || parallelism > MAX_CAP)
            throw new IllegalArgumentException();
        return parallelism;
    }

    private static ForkJoinWorkerThreadFactory checkFactory
        (ForkJoinWorkerThreadFactory factory) {
        if (factory == null)
            throw new NullPointerException();
        return factory;
    }

    /**
     * parallelism:并行度，这并不是定义的线程数，具体线程数，以及WorkQueue的长度等都是根据这个并行度来计
算的，通过makeCommonPool方法可以知道，parallelism默认值是CPU核心线程数减1
     * factory:创建 ForkJoinWorkerThread 的工厂接口
     * handler:每个线程的异常处理器
     * mode:WorkQueue的模式，LIFO/FIFO；
     * ForkJoinWorkerThread的前缀名称
     */
    private ForkJoinPool(int parallelism,
                         ForkJoinWorkerThreadFactory factory,
                         UncaughtExceptionHandler handler,
                         int mode,
                         String workerNamePrefix) {
        this.workerNamePrefix = workerNamePrefix;
        this.factory = factory;
        this.ueh = handler;
        // parallelism & SMASK 其实就是要保证并行度的值不能大于SMASK，上面所有的构造方法在传入
parallelism的时候都会调用checkParallelism来检查合法性：在checkParallelism中可以看到当parallelism >
MAX_CAP时会抛异常， 所以parallelism的最大值就是MAX_CAP了，0x7fff肯定小于0xffff。所以config的值其实就
是：this.config = parallelism | mode;
        // 由于mode都是向右移16位，所以：
        // LIFO_QUEUE = 0 = 00000000000000000000000000000000
        // FIFO_QUEUE = 1 = 00000000000000010000000000000000
        // 所有 parallelism | mode的结果表示，config的高16位代表模式，低16位表示并行度
        this.config = (parallelism & SMASK) | mode;
        // -parallelism的补码表示为：
1111111111111111111111111111111111111111111111111000000000000001
        // 因为parallelism的原码为:0111111111111111
        // 所以-parallelism的原码：1111111111111111
        // 然后-parallelism的反码：1000000000000000
        // 最后-parallelism的补码：1000000000000001
        // 转成long之后变成了：
1111111111111111111111111111111111111111111111111000000000000001
        long np = (long)(-parallelism); // offset ctl counts
        // 线程池的核心控制线程字段
        // np << AC_SHIFT 即 np 向左移动 48 位，这样原来的低 16 位变成了高 16 位，再用 AC 掩码
（AC_MASK） 做与运算，也就是说 ctl 的 49 ～ 64 位表示活跃线程数
```

```
        // np << TC_SHIFT 即 np 向左移动 32 位，这样原来的低 16 位变成了 33 ~ 48 位，再用 TC 掩码做与运算，也就是说 ctl 的 33 ~ 48 位表示总线程数
        // ctl的构成如下：
        // 49 ~ 64表示活跃线程数(AC)，初始化值为-parallelism
        // 33 ~ 48表示总线程数(TC)，初始化值为-parallelism（初始化后 AC = TC）
        // 17 ~ 32栈顶工作线程状态和版本数（每一个线程在挂起时都会持有前一个等待线程所在工作队列的索引，由此构成一个等待的工作线程栈，栈顶是最新等待的线程），第一位表示状态 1：不活动(inactive)； 0：活动(active)，后15表示版本号，防止 ABA 问题
        // 1 ~ 16栈顶工作线程所在工作队列的索引
        // 另 sp=(int)ctl，即获取64位ctl的低32位，因为低32位都是创建出线程之后才会存在的值，所以推断出，如果sp!=0，就存在等待的工作线程，唤醒使用就行，不用创建新的线程。这样就通过ctl可以获取到有关线程所需要的一切信息了
        this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_MASK);
    }

    /**
     * Returns the common pool instance. This pool is statically
     * constructed; its run state is unaffected by attempts to {@link
     * #shutdown} or {@link #shutdownNow}. However this pool and any
     * ongoing processing are automatically terminated upon program
     * {@link System#exit}.  Any program that relies on asynchronous
     * task processing to complete before program termination should
     * invoke {@code commonPool().}{@link #awaitQuiescence awaitQuiescence},
     * before exit.
     *
     * @return the common pool instance
     * @since 1.8
     */
    public static ForkJoinPool commonPool() {
        // assert common != null : "static init error";
        return common;
    }

    // Execution methods

    /**
     * Performs the given task, returning its result upon completion.
     * If the computation encounters an unchecked Exception or Error,
     * it is rethrown as the outcome of this invocation.  Rethrown
     * exceptions behave in the same way as regular exceptions, but,
     * when possible, contain stack traces (as displayed for example
     * using {@code ex.printStackTrace()}) of both the current thread
     * as well as the thread actually encountering the exception;
     * minimally only the latter.
     *
     * @param task the task
     * @param <T> the type of the task's result
     * @return the task's result
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public <T> T invoke(ForkJoinTask<T> task) {
```

```java
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
        return task.join();
    }

    /**
     * Arranges for (asynchronous) execution of the given task.
     *
     * @param task the task
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public void execute(ForkJoinTask<?> task) {
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
    }

    // AbstractExecutorService methods

    /**
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public void execute(Runnable task) {
        if (task == null)
            throw new NullPointerException();
        ForkJoinTask<?> job;
        if (task instanceof ForkJoinTask<?>) // avoid re-wrap
            job = (ForkJoinTask<?>) task;
        else
            job = new ForkJoinTask.RunnableExecuteAction(task);
        externalPush(job);
    }

    /**
     * Submits a ForkJoinTask for execution.
     *
     * @param task the task to submit
     * @param <T> the type of the task's result
     * @return the task
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task) {
        if (task == null)
            throw new NullPointerException();
        externalPush(task);
        return task;
    }

    /**
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public <T> ForkJoinTask<T> submit(Callable<T> task) {
        ForkJoinTask<T> job = new ForkJoinTask.AdaptedCallable<T>(task);
        externalPush(job);
        return job;
    }

    /**
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public <T> ForkJoinTask<T> submit(Runnable task, T result) {
        ForkJoinTask<T> job = new ForkJoinTask.AdaptedRunnable<T>(task, result);
        externalPush(job);
        return job;
    }

    /**
     * @throws NullPointerException if the task is null
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     */
    public ForkJoinTask<?> submit(Runnable task) {
        if (task == null)
            throw new NullPointerException();
        ForkJoinTask<?> job;
        if (task instanceof ForkJoinTask<?>) // avoid re-wrap
            job = (ForkJoinTask<?>) task;
        else
            job = new ForkJoinTask.AdaptedRunnableAction(task);
        externalPush(job);
        return job;
    }

    /**
     * @throws NullPointerException       {@inheritDoc}
     * @throws RejectedExecutionException {@inheritDoc}
     */
    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) {
        // In previous versions of this class, this method constructed
        // a task to run ForkJoinTask.invokeAll, but now external
        // invocation of multiple tasks is at least as efficient.
        ArrayList<Future<T>> futures = new ArrayList<>(tasks.size());

        boolean done = false;
        try {
            for (Callable<T> t : tasks) {
                ForkJoinTask<T> f = new ForkJoinTask.AdaptedCallable<T>(t);
                futures.add(f);
                externalPush(f);
            }
            for (int i = 0, size = futures.size(); i < size; i++)
                ((ForkJoinTask<?>)futures.get(i)).quietlyJoin();
            done = true;
            return futures;
        } finally {
            if (!done)
                for (int i = 0, size = futures.size(); i < size; i++)
                    futures.get(i).cancel(false);
        }
    }

    /**
     * Returns the factory used for constructing new workers.
     *
     * @return the factory used for constructing new workers
     */
    public ForkJoinWorkerThreadFactory getFactory() {
        return factory;
    }

    /**
     * Returns the handler for internal worker threads that terminate
     * due to unrecoverable errors encountered while executing tasks.
     *
     * @return the handler, or {@code null} if none
     */
    public UncaughtExceptionHandler getUncaughtExceptionHandler() {
        return ueh;
    }

    /**
     * Returns the targeted parallelism level of this pool.
     *
     * @return the targeted parallelism level of this pool
     */
    public int getParallelism() {
        int par;
        return ((par = config & SMASK) > 0) ? par : 1;
    }

    /**
     * Returns the targeted parallelism level of the common pool.
     *
     * @return the targeted parallelism level of the common pool
     * @since 1.8
     */
    public static int getCommonPoolParallelism() {
        return commonParallelism;
    }

    /**
     * Returns the number of worker threads that have started but not
     * yet terminated.  The result returned by this method may differ
     * from {@link #getParallelism} when threads are created to
     * maintain parallelism when others are cooperatively blocked.
     *
     * @return the number of worker threads
     */
    public int getPoolSize() {
        return (config & SMASK) + (short)(ctl >>> TC_SHIFT);
    }

    /**
     * Returns {@code true} if this pool uses local first-in-first-out
     * scheduling mode for forked tasks that are never joined.
     *
     * @return {@code true} if this pool uses async mode
     */
    public boolean getAsyncMode() {
        return (config & FIFO_QUEUE) != 0;
    }

    /**
     * Returns an estimate of the number of worker threads that are
     * not blocked waiting to join tasks or for other managed
     * synchronization. This method may overestimate the
     * number of running threads.
     *
     * @return the number of worker threads
     */
    public int getRunningThreadCount() {
        int rc = 0;
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 1; i < ws.length; i += 2) {
                if ((w = ws[i]) != null && w.isApparentlyUnblocked())
                    ++rc;
            }
        }
        return rc;
    }

    /**
     * Returns an estimate of the number of threads that are currently
     * stealing or executing tasks. This method may overestimate the
     * number of active threads.
     *
     * @return the number of active threads
     */
    public int getActiveThreadCount() {
        int r = (config & SMASK) + (int)(ctl >> AC_SHIFT);
        return (r <= 0) ? 0 : r; // suppress momentarily negative values
    }

    /**
     * Returns {@code true} if all worker threads are currently idle.
     * An idle worker is one that cannot obtain a task to execute
     * because none are available to steal from other threads, and
     * there are no pending submissions to the pool. This method is
     * conservative; it might not return {@code true} immediately upon
     * idleness of all threads, but will eventually become true if
     * threads remain inactive.
     *
     * @return {@code true} if all threads are currently idle
     */
    public boolean isQuiescent() {
        return (config & SMASK) + (int)(ctl >> AC_SHIFT) <= 0;
    }

    /**
     * Returns an estimate of the total number of tasks stolen from
     * one thread's work queue by another. The reported value
     * underestimates the actual total number of steals when the pool
     * is not quiescent. This value may be useful for monitoring and
     * tuning fork/join programs: in general, steal counts should be
     * high enough to keep threads busy, but low enough to avoid
     * overhead and contention across threads.
     *
     * @return the number of steals
     */
    public long getStealCount() {
        AtomicLong sc = stealCounter;
        long count = (sc == null) ? 0L : sc.get();
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 1; i < ws.length; i += 2) {
                if ((w = ws[i]) != null)
                    count += w.nsteals;
            }
        }
        return count;
    }

    /**
     * Returns an estimate of the total number of tasks currently held
     * in queues by worker threads (but not including tasks submitted
     * to the pool that have not begun executing). This value is only
     * an approximation, obtained by iterating across all threads in
     * the pool. This method may be useful for tuning task
     * granularities.
     *
     * @return the number of queued tasks
     */
    public long getQueuedTaskCount() {
        long count = 0;
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 1; i < ws.length; i += 2) {
                if ((w = ws[i]) != null)
                    count += w.queueSize();
            }
        }
        return count;
    }

    /**
     * Returns an estimate of the number of tasks submitted to this
     * pool that have not yet begun executing.  This method may take
     * time proportional to the number of submissions.
     *
     * @return the number of queued submissions
     */
    public int getQueuedSubmissionCount() {
        int count = 0;
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 0; i < ws.length; i += 2) {
                if ((w = ws[i]) != null)
                    count += w.queueSize();
            }
        }
        return count;
    }

    /**
     * Returns {@code true} if there are any tasks submitted to this
     * pool that have not yet begun executing.
     *
     * @return {@code true} if there are any queued submissions
     */
    public boolean hasQueuedSubmissions() {
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 0; i < ws.length; i += 2) {
                if ((w = ws[i]) != null && !w.isEmpty())
                    return true;
            }
        }
        return false;
    }

    /**
     * Removes and returns the next unexecuted submission if one is
     * available.  This method may be useful in extensions to this
     * class that re-assign work in systems with multiple pools.
     *
     * @return the next submission, or {@code null} if none
     */
    protected ForkJoinTask<?> pollSubmission() {
        WorkQueue[] ws; WorkQueue w; ForkJoinTask<?> t;
        if ((ws = workQueues) != null) {
            for (int i = 0; i < ws.length; i += 2) {
                if ((w = ws[i]) != null && (t = w.poll()) != null)
                    return t;
            }
        }
        return null;
    }

    /**
     * Removes all available unexecuted submitted and forked tasks
     * from scheduling queues and adds them to the given collection,
     * without altering their execution status. These may include
     * artificially generated or wrapped tasks. This method is
     * designed to be invoked only when the pool is known to be
     * quiescent. Invocations at other times may not remove all
     * tasks. A failure encountered while attempting to add elements
     * to collection {@code c} may result in elements being in
     * neither, either or both collections when the associated
     * exception is thrown.  The behavior of this operation is
     * undefined if the specified collection is modified while the
     * operation is in progress.
     *
     * @param c the collection to transfer elements into
     * @return the number of elements transferred
     */
    protected int drainTasksTo(Collection<? super ForkJoinTask<?>> c) {
        int count = 0;
        WorkQueue[] ws; WorkQueue w; ForkJoinTask<?> t;
        if ((ws = workQueues) != null) {
            for (int i = 0; i < ws.length; ++i) {
                if ((w = ws[i]) != null) {
                    while ((t = w.poll()) != null) {
                        c.add(t);
                        ++count;
                    }
                }
            }
        }
        return count;
    }

    /**
     * Returns a string identifying this pool, as well as its state,
     * including indications of run state, parallelism level, and
     * worker and task counts.
     *
     * @return a string identifying this pool, as well as its state
     */
    public String toString() {
        // Use a single pass through workQueues to collect counts
        long qt = 0L, qs = 0L; int rc = 0;
        AtomicLong sc = stealCounter;
        long st = (sc == null) ? 0L : sc.get();
        long c = ctl;
        WorkQueue[] ws; WorkQueue w;
        if ((ws = workQueues) != null) {
            for (int i = 0; i < ws.length; ++i) {
                if ((w = ws[i]) != null) {
                    int size = w.queueSize();
                    if ((i & 1) == 0)
                        qs += size;
                    else {
                        qt += size;
                        st += w.nsteals;
                        if (w.isApparentlyUnblocked())
                            ++rc;
                    }
                }
            }
        }
        int pc = (config & SMASK);
        int tc = pc + (short)(c >>> TC_SHIFT);
        int ac = pc + (int)(c >> AC_SHIFT);
        if (ac < 0) // ignore transient negative
            ac = 0;
        int rs = runState;
        String level = ((rs & TERMINATED) != 0 ? "Terminated" :
                        (rs & STOP)       != 0 ? "Terminating" :
                        (rs & SHUTDOWN)   != 0 ? "Shutting down" :
                        "Running");
        return super.toString() +
            "[" + level +
            ", parallelism = " + pc +
            ", size = " + tc +
            ", active = " + ac +
            ", running = " + rc +
            ", steals = " + st +
            ", tasks = " + qt +
            ", submissions = " + qs +
            "]";
    }

    /**
     * Possibly initiates an orderly shutdown in which previously
     * submitted tasks are executed, but no new tasks will be
     * accepted. Invocation has no effect on execution state if this
     * is the {@link #commonPool()}, and no additional effect if
     * already shut down.  Tasks that are in the process of being
     * submitted concurrently during the course of this method may or
     * may not be rejected.
     *
     * @throws SecurityException if a security manager exists and
     *         the caller is not permitted to modify threads
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")}
     */
    public void shutdown() {
        checkPermission();
        tryTerminate(false, true);
    }

    /**
     * Possibly attempts to cancel and/or stop all tasks, and reject
     * all subsequently submitted tasks.  Invocation has no effect on
     * execution state if this is the {@link #commonPool()}, and no
     * additional effect if already shut down. Otherwise, tasks that
     * are in the process of being submitted or executed concurrently
     * during the course of this method may or may not be
     * rejected. This method cancels both existing and unexecuted
     * tasks, in order to permit termination in the presence of task
     * dependencies. So the method always returns an empty list
     * (unlike the case for some other Executors).
     *
     * @return an empty list
     * @throws SecurityException if a security manager exists and
     *         the caller is not permitted to modify threads
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")}
     */
    public List<Runnable> shutdownNow() {
        checkPermission();
        tryTerminate(true, true);
        return Collections.emptyList();
    }

    /**
     * Returns {@code true} if all tasks have completed following shut down.
     *
     * @return {@code true} if all tasks have completed following shut down
     */
    public boolean isTerminated() {
        return (runState & TERMINATED) != 0;
    }

    /**
     * Returns {@code true} if the process of termination has
     * commenced but not yet completed.  This method may be useful for
     * debugging. A return of {@code true} reported a sufficient
     * period after shutdown may indicate that submitted tasks have
     * ignored or suppressed interruption, or are waiting for I/O,
     * causing this executor not to properly terminate. (See the
     * advisory notes for class {@link ForkJoinTask} stating that
     * tasks should not normally entail blocking operations.  But if
     * they do, they must abort them on interrupt.)
     *
     * @return {@code true} if terminating but not yet terminated
     */
    public boolean isTerminating() {
        int rs = runState;
        return (rs & STOP) != 0 && (rs & TERMINATED) == 0;
    }

    /**
     * Returns {@code true} if this pool has been shut down.
     *
     * @return {@code true} if this pool has been shut down
     */
    public boolean isShutdown() {
        return (runState & SHUTDOWN) != 0;
    }

    /**
     * Blocks until all tasks have completed execution after a
     * shutdown request, or the timeout occurs, or the current thread
     * is interrupted, whichever happens first. Because the {@link
     * #commonPool()} never terminates until program shutdown, when
     * applied to the common pool, this method is equivalent to {@link
     * #awaitQuiescence(long, TimeUnit)} but always returns {@code false}.
     *
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @return {@code true} if this executor terminated and
     *         {@code false} if the timeout elapsed before termination
     * @throws InterruptedException if interrupted while waiting
     */
    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (this == common) {
            awaitQuiescence(timeout, unit);
            return false;
        }
        long nanos = unit.toNanos(timeout);
        if (isTerminated())
            return true;
        if (nanos <= 0L)
            return false;
        long deadline = System.nanoTime() + nanos;
        synchronized (this) {
            for (;;) {
                if (isTerminated())
                    return true;
                if (nanos <= 0L)
                    return false;
                long millis = TimeUnit.NANOSECONDS.toMillis(nanos);
                wait(millis > 0L ? millis : 1L);
                nanos = deadline - System.nanoTime();
            }
        }
    }

    /**
     * If called by a ForkJoinTask operating in this pool, equivalent
     * in effect to {@link ForkJoinTask#helpQuiesce}. Otherwise,
     * waits and/or attempts to assist performing tasks until this
     * pool {@link #isQuiescent} or the indicated timeout elapses.
     *
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @return {@code true} if quiescent; {@code false} if the
     * timeout elapsed.
     */
    public boolean awaitQuiescence(long timeout, TimeUnit unit) {
        long nanos = unit.toNanos(timeout);
        ForkJoinWorkerThread wt;
        Thread thread = Thread.currentThread();
        if ((thread instanceof ForkJoinWorkerThread) &&
            (wt = (ForkJoinWorkerThread)thread).pool == this) {
            helpQuiescePool(wt.workQueue);
            return true;
        }
        long startTime = System.nanoTime();
        WorkQueue[] ws;
        int r = 0, m;
        boolean found = true;
        while (!isQuiescent() && (ws = workQueues) != null &&
               (m = ws.length - 1) >= 0) {
            if (!found) {
                if ((System.nanoTime() - startTime) > nanos)
                    return false;
                Thread.yield(); // cannot block
            }
            found = false;
            for (int j = (m + 1) << 2; j >= 0; --j) {
                ForkJoinTask<?> t; WorkQueue q; int b, k;
                if ((k = r++ & m) <= m && k >= 0 && (q = ws[k]) != null &&
                    (b = q.base) - q.top < 0) {
                    found = true;
                    if ((t = q.pollAt(b)) != null)
                        t.doExec();
                    break;
                }
            }
        }
        return true;
    }

    /**
     * Waits and/or attempts to assist performing tasks indefinitely
     * until the {@link #commonPool()} {@link #isQuiescent}.
     */
    static void quiesceCommonPool() {
        common.awaitQuiescence(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    }

    /**
     * Interface for extending managed parallelism for tasks running
     * in {@link ForkJoinPool}s.
     *
     * <p>A {@code ManagedBlocker} provides two methods.  Method
     * {@link #isReleasable} must return {@code true} if blocking is
     * not necessary. Method {@link #block} blocks the current thread
     * if necessary (perhaps internally invoking {@code isReleasable}
     * before actually blocking). These actions are performed by any
     * thread invoking {@link ForkJoinPool#managedBlock(ManagedBlocker)}.
     * The unusual methods in this API accommodate synchronizers that
     * may, but don't usually, block for long periods. Similarly, they
     * allow more efficient internal handling of cases in which
     * additional workers may be, but usually are not, needed to
     * ensure sufficient parallelism.  Toward this end,
     * implementations of method {@code isReleasable} must be amenable
     * to repeated invocation.
     *
     * <p>For example, here is a ManagedBlocker based on a
     * ReentrantLock:
     *  <pre> {@code
     * class ManagedLocker implements ManagedBlocker {
     *   final ReentrantLock lock;
     *   boolean hasLock = false;
     *   ManagedLocker(ReentrantLock lock) { this.lock = lock; }
     *   public boolean block() {
     *     if (!hasLock)
     *       lock.lock();
     *     return true;
     *   }
     *   public boolean isReleasable() {
     *     return hasLock || (hasLock = lock.tryLock());
     *   }
     * }}</pre>
     *
     * <p>Here is a class that possibly blocks waiting for an
     * item on a given queue:
     *  <pre> {@code
     * class QueueTaker<E> implements ManagedBlocker {
     *   final BlockingQueue<E> queue;
     *   volatile E item = null;
     *   QueueTaker(BlockingQueue<E> q) { this.queue = q; }
     *   public boolean block() throws InterruptedException {
     *     if (item == null)
     *       item = queue.take();
     *     return true;
     *   }
     *   public boolean isReleasable() {
```

```
     *     return item != null || (item = queue.poll()) != null;
     *   }
     *   public E getItem() { // call after pool.managedBlock completes
     *     return item;
     *   }
     * }}</pre>
     */
    public static interface ManagedBlocker {
        /**
         * Possibly blocks the current thread, for example waiting for
         * a lock or condition.
         *
         * @return {@code true} if no additional blocking is necessary
         * (i.e., if isReleasable would return true)
         * @throws InterruptedException if interrupted while waiting
         * (the method is not required to do so, but is allowed to)
         */
        boolean block() throws InterruptedException;

        /**
         * Returns {@code true} if blocking is unnecessary.
         * @return {@code true} if blocking is unnecessary
         */
        boolean isReleasable();
    }

    /**
     * Runs the given possibly blocking task.  When {@linkplain
     * ForkJoinTask#inForkJoinPool() running in a ForkJoinPool}, this
     * method possibly arranges for a spare thread to be activated if
     * necessary to ensure sufficient parallelism while the current
     * thread is blocked in {@link ManagedBlocker#block blocker.block()}.
     *
     * <p>This method repeatedly calls {@code blocker.isReleasable()} and
     * {@code blocker.block()} until either method returns {@code true}.
     * Every call to {@code blocker.block()} is preceded by a call to
     * {@code blocker.isReleasable()} that returned {@code false}.
     *
     * <p>If not running in a ForkJoinPool, this method is
     * behaviorally equivalent to
     *  <pre> {@code
     * while (!blocker.isReleasable())
     *   if (blocker.block())
     *     break;}</pre>
     *
     * If running in a ForkJoinPool, the pool may first be expanded to
     * ensure sufficient parallelism available during the call to
     * {@code blocker.block()}.
     *
     * @param blocker the blocker task
     * @throws InterruptedException if {@code blocker.block()} did so
     */
    public static void managedBlock(ManagedBlocker blocker)
```

```
        throws InterruptedException {
        ForkJoinPool p;
        ForkJoinWorkerThread wt;
        Thread t = Thread.currentThread();
        if ((t instanceof ForkJoinWorkerThread) &&
            (p = (wt = (ForkJoinWorkerThread)t).pool) != null) {
            WorkQueue w = wt.workQueue;
            while (!blocker.isReleasable()) {
                if (p.tryCompensate(w)) {
                    try {
                        do {} while (!blocker.isReleasable() &&
                                     !blocker.block());
                    } finally {
                        U.getAndAddLong(p, CTL, AC_UNIT);
                    }
                    break;
                }
            }
        }
        else {
            do {} while (!blocker.isReleasable() &&
                         !blocker.block());
        }
    }

    // AbstractExecutorService overrides.  These rely on undocumented
    // fact that ForkJoinTask.adapt returns ForkJoinTasks that also
    // implement RunnableFuture.

    protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
        return new ForkJoinTask.AdaptedRunnable<T>(runnable, value);
    }

    protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        return new ForkJoinTask.AdaptedCallable<T>(callable);
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe U;
    private static final int  ABASE;
    private static final int  ASHIFT;
    private static final long CTL;
    private static final long RUNSTATE;
    private static final long STEALCOUNTER;
    private static final long PARKBLOCKER;
    private static final long QTOP;
    private static final long QLOCK;
    private static final long QSCANSTATE;
    private static final long QPARKER;
    private static final long QCURRENTSTEAL;
    private static final long QCURRENTJOIN;

    static {
        // initialize field offsets for CAS etc
        try {
            U = sun.misc.Unsafe.getUnsafe();
            Class<?> k = ForkJoinPool.class;
            CTL = U.objectFieldOffset
                (k.getDeclaredField("ctl"));
            RUNSTATE = U.objectFieldOffset
                (k.getDeclaredField("runState"));
            STEALCOUNTER = U.objectFieldOffset
                (k.getDeclaredField("stealCounter"));
            Class<?> tk = Thread.class;
            PARKBLOCKER = U.objectFieldOffset
                (tk.getDeclaredField("parkBlocker"));
            Class<?> wk = WorkQueue.class;
            QTOP = U.objectFieldOffset
                (wk.getDeclaredField("top"));
            QLOCK = U.objectFieldOffset
                (wk.getDeclaredField("qlock"));
            QSCANSTATE = U.objectFieldOffset
                (wk.getDeclaredField("scanState"));
            QPARKER = U.objectFieldOffset
                (wk.getDeclaredField("parker"));
            QCURRENTSTEAL = U.objectFieldOffset
                (wk.getDeclaredField("currentSteal"));
            QCURRENTJOIN = U.objectFieldOffset
                (wk.getDeclaredField("currentJoin"));
            Class<?> ak = ForkJoinTask[].class;
            ABASE = U.arrayBaseOffset(ak);
            int scale = U.arrayIndexScale(ak);
            if ((scale & (scale - 1)) != 0)
                throw new Error("data type scale not a power of two");
            ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
        } catch (Exception e) {
            throw new Error(e);
        }

        commonMaxSpares = DEFAULT_COMMON_MAX_SPARES;
        defaultForkJoinWorkerThreadFactory =
            new DefaultForkJoinWorkerThreadFactory();
        modifyThreadPermission = new RuntimePermission("modifyThread");

        common = java.security.AccessController.doPrivileged
            (new java.security.PrivilegedAction<ForkJoinPool>() {
                public ForkJoinPool run() { return makeCommonPool(); }});
        int par = common.config & SMASK; // report 1 even if threads disabled
        commonParallelism = par > 0 ? par : 1;
    }

    /**
     * Creates and returns the common pool, respecting user settings
     * specified via system properties.
     */
    private static ForkJoinPool makeCommonPool() {
        int parallelism = -1;
        ForkJoinWorkerThreadFactory factory = null;
        UncaughtExceptionHandler handler = null;
        try {  // ignore exceptions in accessing/parsing properties
            String pp = System.getProperty
                ("java.util.concurrent.ForkJoinPool.common.parallelism");
            String fp = System.getProperty
                ("java.util.concurrent.ForkJoinPool.common.threadFactory");
            String hp = System.getProperty
                ("java.util.concurrent.ForkJoinPool.common.exceptionHandler");
            if (pp != null)
                parallelism = Integer.parseInt(pp);
            if (fp != null)
                factory = ((ForkJoinWorkerThreadFactory)ClassLoader.
                           getSystemClassLoader().loadClass(fp).newInstance());
            if (hp != null)
                handler = ((UncaughtExceptionHandler)ClassLoader.
                           getSystemClassLoader().loadClass(hp).newInstance());
        } catch (Exception ignore) {
        }
        if (factory == null) {
            if (System.getSecurityManager() == null)
                factory = defaultForkJoinWorkerThreadFactory;
            else // use security-managed default
                factory = new InnocuousForkJoinWorkerThreadFactory();
        }
        if (parallelism < 0 && // default 1 less than #cores
            (parallelism = Runtime.getRuntime().availableProcessors() - 1) <= 0)
            parallelism = 1;
        if (parallelism > MAX_CAP)
            parallelism = MAX_CAP;
        return new ForkJoinPool(parallelism, factory, handler, LIFO_QUEUE,
                                "ForkJoinPool.commonPool-worker-");
    }

    /**
     * Factory for innocuous worker threads
     */
    static final class InnocuousForkJoinWorkerThreadFactory
        implements ForkJoinWorkerThreadFactory {

        /**
         * An ACC to restrict permissions for the factory itself.
         * The constructed workers have no permissions set.
         */
        private static final AccessControlContext innocuousAcc;
        static {
            Permissions innocuousPerms = new Permissions();
            innocuousPerms.add(modifyThreadPermission);
            innocuousPerms.add(new RuntimePermission(
                                   "enableContextClassLoaderOverride"));
            innocuousPerms.add(new RuntimePermission(
                                   "modifyThreadGroup"));
            innocuousAcc = new AccessControlContext(new ProtectionDomain[] {
                    new ProtectionDomain(null, innocuousPerms)
                });
        }

        public final ForkJoinWorkerThread newThread(ForkJoinPool pool) {
            return (ForkJoinWorkerThread.InnocuousForkJoinWorkerThread)
                java.security.AccessController.doPrivileged(
                    new java.security.PrivilegedAction<ForkJoinWorkerThread>() {
                    public ForkJoinWorkerThread run() {
                        return new ForkJoinWorkerThread.
                            InnocuousForkJoinWorkerThread(pool);
                    }}, innocuousAcc);
        }
    }

}
```