CPU多级缓存

CPU设置缓存意义：1）时间局部性：如果某个数据被访问，那么在不久的将来也可能被访问。2）空间局部性：如果某个数据被访问，那个与它相邻的数据很快也可能访问。

多级缓存一致性（MESI）

在一个有多核的系统中，每一个核都有自己的缓存来共享主存总线，目的是为了解决CPU运算速度与内存读写速度不匹配的矛盾 缓存一致性（MESI）是为了保证CPU多级缓存的共享一致性。MESI定义了四种状态，也就是CPU对四种操作产生不一致的状态。缓存控制器监听到本地操作和远程操作的时候，需要对cache做出修改，从而保证数据在cache之间的一致性。

1. M: Modified 修改，指的是该缓存行只被缓存在该CPU的缓存中，并且是被修改过的，因此他与主存中的数据是不一致的，该缓存行中的数据需要在未来的某个时间点（允许其他CPU读取主存相应中的内容之前）写回主存，然后状态变成E（独享）。
    2. E：Exclusive 独享缓存行只被缓存在该CPU的缓存中，是未被修改过的，与主存的数据是一致的，可以在任何时刻当有其他CPU读取该内存时，变成S（共享）状态，当CPU修改该缓存行的内容时，变成M（被修改）的状态。
2. S：Share 共享，意味着该缓存行可能会被多个CPU进行缓存，并且该缓存中的数据与主存数据是一致的，当有一个CPU修改该缓存行时，其他CPU是可以被作废的，变成I(无效的)。
3. I：Invalid 无效的，代表这个缓存是无效的，可能是有其他CPU修改了该缓存行。

Java内存模型

主内存和工作内存

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在JVM中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量与Java编程里面的变量有所不同步，它包含了实例字段、静态字段和构成数组对象的元素，但不包含局部变量和方法参数，因为后者是线程私有的，不会共享，当然不存在数据竞争问题（如果局部变量是一个reference引用类型，它引用的对象在Java堆中可被各个线程共享，但是reference引用本身在Java栈的局部变量表中，是线程私有的）。为了获得较高的执行效能，Java内存模型并没有限制执行引起使用处理器的特定寄存器或者缓存来和主内存进行交互，也没有限制即时编译器进行调整代码执行顺序这类优化措施。

JMM规定了所有的变量都存储在主内存（Main Memory）中。每个线程还有自己的工作内存（Working Memory），线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（volatile变量仍然有工作内存的拷贝，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。

线程1和线程2要想进行数据的交换一般要经历下面的步骤：

```
1．线程1把工作内存1中的更新过的共享变量刷新到主内存中去。
```

2. 线程2到主内存中去读取线程1刷新过的共享变量，然后copy一份到工作内存2中去。

内存之间相互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成：

1. lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
2. unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
3. read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
4. load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
5. use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
6. assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
7. store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
8. write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是read和load之间，store和write之间是可以插入其他指令的，如对主内存中的变量a、b进行访问时，可能的顺序是read a，read b，load b， load a。Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

```
1. 不允许read和load、store和write操作之一单独出现
```

2. 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
3. 不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中
4. 一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量
5. 一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁
6. 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。
7. 如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。
8. 对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作）。

重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

```
1. 编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。
```

2. 指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3. 内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

为了保证内存的可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。Java内存模型把内存屏障分为LoadLoad、LoadStore、StoreLoad和StoreStore四种：

屏蔽类型	指令示例	说明
LoadLoad Barriers	Load1;LoadLoad;Load2	确保Load1数据的装载，之前于Load2及所有后续装载指令的装载
StoreStore Barriers	Store1;StoreStore;Store2	确保Store1数据对其他处理器可见（刷新到内存），之前于Store2及所有后续存储指令的存储。
LoadStrore Barriers	Load1;LoadStore;Store2	确保Load1数据装载，之前于Store2及所有后续的存储指令刷新到内存
StoreLoad Barrirers	Store1;StoreLoad;Load2	确保Store1数据对其他处理器可见（刷新到内存），之前于Load2及所有后续装载指令的装载，StroeLoad Barrirers会使该屏蔽之前的所有内存访问指令（存储和装载指令）完成之后，才执行该屏障之后的内存访问指令

Java内存区

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈、常量池六个部分，这些区域都有各自的用途、创建时间、销毁时间。

1. PC寄存器/程序计数器：严格来说是一个数据结构，用于保存当前正在执行的程序的内存地址，为了线程切换后能恢复到正确的执行位置，每个线程都需要有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存,这在某种程度上有点类似于“ThreadLocal”，是线程安全的。
2. Java栈 Java Stack：Java栈总是与线程关联在一起的，每当创建一个线程，JVM就会为该线程创建对应的Java栈，在这个Java栈中又会包含多个栈帧(Stack Frame)，这些栈帧是与每个方法关联起来的，每运行一个方法就创建一个栈帧，每个栈帧会含有一些局部变量、操作栈和方法返回值等信息。每当一个方法执行完成时，该栈帧就会弹出栈帧的元素作为这个方法的返回值，并且清除这个栈帧，Java栈的栈顶的栈帧就是当前正在执行的活动栈，也就是当前正在执行的方法，PC寄存器也会指向该地址。只有这个活动的栈帧的本地变量可以被操作栈使用，当在这个栈帧中调用另外一个方法时，与之对应的一个新的栈帧被创建，这个新创建的栈帧被放到Java栈的栈顶，变为当前的活动栈。同样现在只有这个栈的本地变量才能被使用，当这个栈帧中所有指令都完成时，这个栈帧被移除Java栈，刚才的那个栈帧变为活动栈帧，前面栈帧的返回值变为这个栈帧的操作栈的一个操作数。由于Java栈是与线程对应起来的，Java栈数据不是线程共有的，所以不需要关心其数据一致性，也不会存在同步锁的问题。在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。
3. 堆 Heap: 堆是JVM所管理的内存中国最大的一块，是被所有Java线程锁共享的，不是线程安全的，在JVM启动时创建。堆是存储Java对象的地方，这一点Java虚拟机规范中描述是：所有的对象实例以及数组都要在堆上分配。Java堆是GC管理的主要区域，从内存回收的角度来看，由于现在GC基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；新生代再细致一点有Eden空间、From Survivor空间、To Survivor空间等
4. 方法区Method Area: 方法区存放了要加载的类的信息（名称、修饰符等）、类中的静态常量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当在程序中通过Class对象的getName.isInterface等方法来获取信息时，这些数据都来源于方法区。方法区是被Java线程锁共享的，不像Java堆中其他部分一样会频繁被GC回收，它存储的信息相对比较稳定，在一定条件下会被GC，当方法区要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。方法区也是堆中的一部分，就是我们通常所说的Java堆中的永久区 Permanet Generation，大小可以通过参数来设置,可以通过-XX:PermSize指定初始值，-XX:MaxPermSize指定最大值。

5. 常量池Constant Pool: 常量池本身是方法区中的一个数据结构。常量池中存储了如字符串、final变量值、类名和方法名常量。常量池在编译期间就被确定，并保存在已编译的.class文件中。一般分为两类：字面量和应用量。字面量就是字符串、final变量等。类名和方法名属于引用量。引用量最常见的是在调用方法的时候，根据方法名找到方法的引用，并以此定为到函数体进行函数代码的执行。引用量包含：类和接口的权限定名、字段的名称和描述符，方法的名称和描述符。
6. 本地方法栈Native Method Stack: 本地方法栈和Java栈所发挥的作用非常相似，区别不过是Java栈为JVM执行Java方法服务，而本地方法栈为JVM执行Native方法服务。本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。

GC垃圾回收机制

Java语言中一个显著的特点就是引入了垃圾回收机制，使c++程序员最头疼的内存管理的问题迎刃而解，它使得Java程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存。

回收方法

标记-清除法：

算法分为“标记”和“清楚”两个阶段：

```
1．首先标记出所需要回收的对象；
```

2. 在标记完成后统一回收所有被标记的对象。

缺点：

```
- 效率问题：标记和清楚两个过程的效率都不高；
- 空间问题：标记清楚之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后再程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。
```

复制算法：

将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完后，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉。

优点：

- 不用考虑内存碎片等复杂情况
- 实现简单，运行高效

缺点：

```
-  空间缩小
```

分代算法：

采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。

按照对象存活时间，分成新生代、老年代、永久代。 新生代：蜉蝣于天地，朝生夕死，用复制算法批量删。 老年代：有的人还活着，但是他已经活的太久了。所有同龄的都已逝去，所有新生的都尚未了解。没有人记得他，他已经在社会层面死亡了。用标记-清除算法抹去。 永久代：不可以删的如加载的class信息，就不删了留着吧。

优点:

```
- 解决了碎片问题
```

缺点:

```
- 代价高
```

垃圾分代

新生代:

```
- 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。
- 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。
- 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收
- 新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)
```

老年代:

```
- 在新生代中经历了N次垃圾回收后仍然存活的对象，就会被放到老年嗲中。因此，可以认为老年代中存放的都是一些生命周期较长的对象。
- 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。
```

持久带:

```
- 用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。
- Java 1.8后，使用Metaspace替代，其好处是不舍限制，杜绝了OutOfMemoryError，使用系统内存;
```

垃圾回收器

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。

```
- Serial收集器（复制算法）：新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
- Serial Old收集器（标记-整理算法）：老年代单线程收集器，Serial收集器的老年代版本；
- ParNew收集器（复制算法）：新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
- Parallel Scavenge收集器（复制算法）：新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
- Parallel Old收集器（标记-整理算法）： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版
- CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
- G1(Garbage First)收集器（标记-整理算法）： Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。
```

回收策略

- 对象优先在Eden分配，当Eden区没有足够空间进行分配时，虚拟机将发起一次MinorGC。现在的商业虚拟机一般都采用复制算法来回收新生代，将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。 当进行垃圾回收时，将Eden和Survivor中还存活的对象一次性地复制到另外一块Survivor空间上，最后处理掉Eden和刚才的Survivor空间。（HotSpot虚拟机默认Eden和Survivor的大小比例是8:1）当Survivor空间不够用时，需要依赖老年代进行分配担保。
- 大对象直接进入老年代。所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组。
- 长期存活的对象将进入老年代。当对象在新生代中经历过一定次数（默认为15）的Minor GC后，就会被晋升到老年代中。
- 动态对象年龄判定。为了更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

需要注意的是，Java的垃圾回收机制是Java虚拟机提供的能力，用于在空闲时间以不定时的方式动态回收无任何引用的对象占据的内存空间。也就是说，垃圾收集器回收的是无任何引用的对象占据的内存空间而不是对象本身。

线程安全

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

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

2. 可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized,volatile）。
3. 有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before原则）。

原子性

JDK里面提供了很多atomic类，AtomicInteger,AtomicLong,AtomicBoolean等等。它们是通过CAS完成原子性。

CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

compareAndSwapInt借助C来调用CPU底层指令实现的，AtomicIntege是在一个死循环内,不断尝试修改目标值,知道修改成功,如果竞争不激烈,那么修改成功的概率就很高,否则,修改失败的概率就很高,在大量修改失败时,这些原子操作就会进行多次循环尝试,因此性能就会受到影响。

那么竞争激烈的时候,我们应该如何进一步提高系统性能呢?一种基本方案就是可以使用热点分离,将竞争的数据进行分解.基于这个思路,一种可行的方案就是仿造ConcurrengHashMap,将热点数据分离,比如,可以将AtomicInteger的内部核心数据value分离成一个数组,每个线程访问时,通过哈希等算法映射到其中一个数字进行计数,而最终的计数结果,则为这个数组的求和累加；LongAdde类将热点数据value被分离成多个单元cell,每个cell独自维护内部的值,当前对象的实际值由所有的cell累计合成,这样,热点就进行了有效的分离,提高了并行度,LongAdder正是使用了这种思想.

CAS缺点

ABA问题：比如说一个线程one从内存位置V中取出A，这时候另一个线程two也从内存中取出A，并且two进行了一些操作变成了B，然后two又将V位置的数据变成A，这时候线程one进行CAS操作发现内存中仍然是A，然后one操作成功。

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

循环时间长开销大：自旋CAS（不成功，就一直循环执行，直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

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

synchronized

synchronized是Java中的关键字，是一种同步锁。JDK提供锁分两种：一种是synchronized，依赖JVM实现锁，因此在这个关键字作用对象的作用范围内是同一时刻只能有一个线程进行操作；另一种是LOCK，是JDK提供的代码层面的锁，依赖CPU指令，代表性的是ReentrantLock。它修饰的对象有以下几种：

1. 修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；
2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；
3. 修改一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；
4. 修改一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

在用synchronized修饰方法时要注意以下几点：

1. synchronized关键字不能继承。虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，而必须显式地在子类的这个方法中加上synchronized关键字才可以。

2. 在定义接口方法时不能使用synchronized关键字。
3. 构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

CAS与Synchronized的使用情景：

对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。 补充： synchronized在jdk1.6之后，已经改进优化。synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

可见性

volatile的可见性是通过内存屏障和禁止重排序实现的。volatile会在写操作时，会在写操作后加一条store屏障指令，将本地内存中的共享变量值刷新到主内存。volatile在进行读操作时，会在读操作前加一条load指令，从内存中读取共享变量。但是volatile不是原子性的，进行++操作不是安全的。

volatile适用于场景：1. 对变量的写操作不依赖于当前值。2. 该变量没有包含在具有其他变量不变的式子中。因此，volatile适用于状态标记量和双重检测机制。

有序性

有序性是指，在JMM中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。可以通过volatile、synchronized、lock保证有序性。 JMM具有先天的有序性，即不需要通过任何手段就可以得到保证的有序性。这称为happens-before原则.如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性。虚拟机可以随意地对它们进行重排序。
happens-before原则：

```
1. 程序次序规则：在一个单独的线程中，按照程序代码书写的顺序执行。
```

2. 锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作。
3. volatile变量规则：对一个volatile变量的写操作先行发生于后面对该变量的读操作。
4. 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
5. 线程终止规则：线程的所有操作都先行发生于对此线程的终止检测，可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
6. 线程中断规则：对线程interrupt()方法的调用先行发生于发生于被中断线程的代码检测到中断时事件的发生。
7. 对象终结规则：一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize()方法的开始。
8. 传递性：如果操作A先行发生于操作B，操作B 先行发生于操作C，那么可以得出A先行发生于操作C。

安全发布对象

发布对象：使一个对象能够在当前作用域之外的代码中使用。

对象逸出：是一种错误的发布, 当一个对象还没有构造完成时 , 就使得它被其他线程看见。

四种方法：

```
1. 在静态初始化函数中初始化一个对象的引用
```

2. 将对象的引用保存在volatile类型或者AtomicReference对象中
3. 将对象的引用保存到某个正确构造对象的final类型中
4. 将对象的引用保存到某个由锁保护的域中

```
//饿汉模式
//1：适用于构造函数没有太多处理的情况，否则会造成性能的问题
//2：类在实际过程中肯定会被使用，不会造成资源的浪费
public class case1{
    privete case1(){};
    private static case1  instance = new case1();
    public static case1 getInstance(){
        return instance;
    }
}
```

```
//懒汉模式
//统一时间只允许一个线程访问，降低程序性能
public class case4{
    private case4(){};
    private static case4 instance = null;
    public static synchronized case4 getInstance(){
        if(instance == null)
            instance = new case4();
    }
    return instance;
}
```

```
//volatile + 双重检测机制实现线程安全
public class case2(){
    private case2(){};
    private volatile static case2 instance = null;
    public static case2 getInstance(){
        synchronized(case2.class){
            if(instance == null){
                instance = new case2();
            }
        }
    }
    return instance;
}
```

```
//枚举模式
public class example{
    private example(){};
    public static example getInstance(){
        reurn inerClass.INSTANCE.getIntance;
    }
    private enum inerClass{
        INSTANCE;
        private example instance;
        inerClass(){
```

```
            instance = new example;
        }
        public example getInstance(){
            return instance;
        }
    }
}
```

不可变对象：不可变对象是指对象被创建之后，其内部状态保存不变的对象。

1. 对象创建以后其状态不可改变
2. 对象所有与都是final类型
3. 对象都是正确创建的(在对象创建期间，this引用没有逸出)

final关键字：

1. 修饰类：不能被继承
2. 修饰方法：方法不能被继承类修改
3. 修饰变量：基本数据类型变量、引用类型变量

不可变类设计方法：

1. 类添加final修饰符，保证类不被继承。
2. 保证所有成员变量必须私有，并且加上final修饰
3. 不提供改变成员变量的方法
4. 通过构造器初始化所有成员，进行深拷贝
5. 在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝

线程封闭

ThreadLocal

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。

1. 一句话概括：Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响.

2. 线程数据隔离的秘诀：Thread有个ThreadLocalMap类型的属性，叫做threadLocal，该属性用来保存线程本地变量，这样每个线程都有自己的数据，就做到了不同线程间数据的隔离，保证了数据安全。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。
3. ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。从线程的角度看，目标变量就像是线程的本地变量，这也是类名中“Local"所要表达的意思。

1. 每一个Thread线程内部都有一个map

2. Map里面储存本地对象（key）和线程的变量副本（value）

3. Thread内部的map由ThreadLocal维护，由ThreadLocal负责向Map获取和设置线程的变量值

对于不同的线程，每次获取副本值时，别的线程并不能获取当前线程的副本值，形成了副本的隔离，互不干扰。

```
public class Thread implements Runnable {
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

深入解析ThreadLocal

ThreadLocal 类提供如下几个核心方法：

```
public T get()
public void set(T value)
public void remove()
```

```
* get()方法用于获取当前线程的副本变量值
* set()方法用于保存当前线程的副本变量值
* initialValue()为当前线程初始副本变量值
* remove()方法移除当前前程的副本变量值
```

get()方法

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null)
            return (T)e.value;
    }
    return setInitialValue();
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

```
protected T initialValue() {
    return null;
}
```

步骤：

```
1. 获取当前线程的ThreadLocalMap对象threadLocals
```

2. 从map中获取线程存储的K-V Entry节点
3. 从Entry节点获取存储的Value副本值返回
4. map为空的话返回初始值null，即线程变量副本为null，在使用时需要注意判断NullPointerException

set()方法

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

步骤

```
1. 获取当前线程的成员变量map
```

2. map非空，则重新将ThreadLocal和新的value副本放入到map中
3. map空，则对线程的成员变量ThreadLocalMap进行初始化创建，并将ThreadLocal和value副本放入map中

remove()方法

```
public void remove() {
 ThreadLocalMap m = getMap(Thread.currentThread());
 if (m != null)
     m.remove(this);
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

ThreadLocalMap

ThreadLocalMap是ThreadLocal的内部类，没有实现Map接口，用独立的方式实现了Map的功能，其内部的Entry也独立实现.

在ThreadLocalMap中，也是用Entry来保存K-V结构数据的。但是Entry中key只能是ThreadLocal对象，这点被Entry的构造方法已经限定死了.

```
static class Entry extends WeakReference<ThreadLocal> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal k, Object v) {
        super(k);
        value = v;
    }
}
```

Entry继承自WeakReference（弱引用，生命周期只能存活到下次GC前），但只有Key是弱引用类型的，Value并非弱引用。

Hash冲突怎么解决

和HashMap的最大的不同在于，ThreadLocalMap结构非常简单，没有next引用，也就是说ThreadLocalMap中解决Hash冲突的方式并非链表的方式，而是采用线性探测的方式，所谓线性探测，就是根据初始key的hashcode值确定元素在table数组中的位置，如果发现这个位置上已经有其他key值的元素被占用，则利用固定的算法寻找一定步长的下个位置，依次判断，直至找到能够存放的位置。

```
private static int nextIndex(int i, int len) {
    return ((i + 1 < len) ? i + 1 : 0);
}
private static int prevIndex(int i, int len) {
    return ((i - 1 >= 0) ? i - 1 : len - 1);
}
```

ThreadLocalMap的问题

由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，发生GC时弱引用Key会被回收，而Value不会回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

如何避免泄漏:

既然Key是弱引用，那么我们要做的事，就是在调用ThreadLocal的get()、set()方法时完成后再调用remove方法，将Entry节点和Map的引用关系移除，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。

应用场景：

```
private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();

//获取Session
public static Session getCurrentSession(){
    Session session =  threadLocal.get();
    //判断Session是否为空，如果为空，将创建一个session，并设置到本地线程变量中
    try {
        if(session ==null&&!session.isOpen()){
            if(sessionFactory==null){
                rbuildSessionFactory();// 创建Hibernate的SessionFactory
            }else{
                session = sessionFactory.openSession();
            }
        }
        threadLocal.set(session);
    } catch (Exception e) {
        // TODO: handle exception
    }

    return session;
}
```

每个线程访问数据库都应当是一个独立的Session会话，如果多个线程共享同一个Session会话，有可能其他线程关闭连接了，当前线程再执行提交时就会出现会话已关闭的异常，导致系统异常。此方式能避免线程争抢Session，提高并发下的安全性。

使用ThreadLocal的典型场景正如上面的数据库连接管理，线程会话管理等场景，只适用于独立变量副本的情况，如果变量为全局共享的，则不适用在高并发下使用。

总结：

- 每个ThreadLocal只能保存一个变量副本，如果想要上线一个线程能够保存多个副本以上，就需要创建多个ThreadLocal。
- ThreadLocal内部的ThreadLocalMap键为弱引用，会有内存泄漏的风险。
- 适用于无状态，副本变量独立后不影响业务逻辑的高并发场景。如果如果业务逻辑强依赖于副本变量，则不适合用ThreadLocal解决，需要另寻解决方案。

补充：

类型	回收时间	使用场景
强引用	一直存活，除非GC Roots不可达	所有程序场景
软引用	内存不足时会被回收	一般用在对内存非常敏感的资源，比如网页缓存、图片缓存
弱引用	只能存活到下一次GC前	生命周期很短的现象：比如ThreadLocal的Key
虚引用	随时会被回收，创建了可能很快会被回收	业界暂无使用场景

Java Collection集合框架

Collection

set接口:

```
1. 存入Set的每个元素都必须是唯一的，Set接口不保证维护元素的次序;
```

2. HashSet类： 为快速查找设计的Set，存入HashSet的对象必须定义hashCode()，它不保证集合的迭代顺序;
3. LinkedHashSet类： 具有HashSet的查询速度，且内部使用链表维护元素的顺序(插入的次序)。

List接口:

```
1. List按对象进入的顺序保存对象，不做排序等操作;
```

2. ArrayList类：由数组实现的List，允许对元素进行快速随机访问，但是向List中间插入与移除元素的速度很慢;
3. LinkedList类： 对顺序访问进行了优化，向List中间插入与删除的开销并不大，随机访问则相对较慢;

Queue接口:

```
1. Queue用于模拟队列这种数据结构，实现“FIFO”等数据结构。通常，队列不允许随机访问队列中的元素;
```

2. ArrayDeque类：为Queue子接口Deque的实现类，数组方式实现;
3. LinkedList类：是List接口的实现类，同时它也实现了Deque接口（Queue子接口）。因此它也可以当做一个双端队列来用，也可以当作“栈”来使用;

Collection接口基本操作

- add(Object o)：增加元素
- addAll(Collection c)：...
- clear()：...
- contains(Object o)：是否包含指定元素
- containsAll(Collection c)：是否包含集合c中的所有元素
- iterator()：返回Iterator对象，用于遍历集合中的元素
- remove(Object o)：移除元素
- removeAll(Collection c)：相当于减集合c
- retainAll(Collection c)：相当于求与c的交集
- size()：返回元素个数
- toArray()：把集合转换为一个数组

Map接口

map接口基本操作:

```
1. V put(K key, V value): 将指定的值与此映射中的指定键相关联（可选操作）。如果此映射中以前包含一个该键的映射关系，则用指定值替换旧值;
```

2. boolean containsKey(Object key): 如果此映射包含指定键的映射关系，则返回 true;
3. boolean containsValue(Object value): 如果此映射为指定值映射一个或多个键，则返回true;
4. Set<Map.Entry<K,V>> entrySet(): 返回此映射中包含的映射关系的set 视图;
5. V get(Object key): 返回指定key对应的value;
6. V remove(Object key): 删除指定key对应的key-value对;

HashMap类和Hashtable类:

```
1. Hashtable是线程安全的，而HashMap不是线程安全的;
```

2. Hashtable不允许null作为key和value，而HashMap则可以使用null作为key和value;
3. 不建议使用Hashtable类，它是一个很古老的类，从JDK1.0开始。如果要考虑线程安全，建议使用Collections工具类将HashMap转换为线程安全的;

LinkedHashMap类:

1. LinkedHashMap从HashMap类继承而来。以链表来维护内部顺序。很多方面跟LinkedHashSet类似。LinkedHashMap它可以记住key-value对的添加时的顺序，同时避免使用TreeMap时性能受到的影响。

SortedMap接口及其TreeMap现类:

```
1. 类似于SortedSet及TreeSet，TreeMap也可以自定义比较器（Comparable）实现定制排序。它的额外提供的方法也与TreeSet类似，增加了访问第一个、前一个、后一个、最后一个key-value对的方法，并提供了从TreeMap中提取子集的方法。TreeMap不允许null作为key;
```

IdentityHashMap类:

```
1. 与HashMap的不同在于，只有两个key严格相等（key1
```

== key2)时，IdentityHashMap才认为两个key相等；而对于普通HashMap而言，只要key1.equals(key2)且hashCode相同即可。同样允许null值，不能保证顺序;

EnumMap类:

```
1. EnumMap是一个与枚举类一起使用的Map实现。它的key必须是单个枚举类的枚举值。EnumMap不允许使用null作为key，但可作为value;
```

各种Map实现类选择策略:

```
1. 常情况使用HashMap，而不是Hashtable。
```

2. 如果考虑排序，那么考虑使用TreeMap。通常TreeMap比HashMap等在插入、删除操作时要慢不少，因为它需要在底层采用红黑树来管理key-value对。
3. 如果考虑插入时的顺序，那么使用LinkedHashMap是个不错的选择.
4. 如果想优化垃圾回收，建议使用WeakHashMap实现类（本文未提及）；要求key完全匹配（同一对象），则使用IdentityHashMap；还有枚举类不多说了。
5. 关于null值：Hashtable不允许key为null，也不允许value为null；TreeMap与EnumMap不允许key为null；HashMap及其子类LinkedHashMap，IdentityHashMap允许key为null;

Java AQS

AQS简介

AQS是Abstract Queued Synchronizer的简称。AQS提供了一种实现阻塞锁和一系列依赖FIFO等待队列的同步器的框架。

AQS是一个抽象类，主是是以继承的方式使用。AQS本身是没有实现任何同步接口的，它仅仅只是定义了同步状态的获取和释放的方法来供自定义的同步组件的使用。

AQS原理介绍

AQS的实现依赖内部的同步队列（FIFO双向队列），如果当前线程获取同步状态失败，AQS会将该线程以及等待状态等信息构造成一个Node，将其加入同步队列的尾部，同时阻塞当前线程，当同步状态释放时，唤醒队列的头节点。

AQS最主要的三个成员变量：

```
private transient volatile Node head;
private transient volatile Node tail;
private volatile int state;
```

上面提到的同步状态就是这个volatile int型的变量state. head和tail分别是同步队列的头结点和尾结点。假设state=0表示同步状态可用（如果用于锁，则表示锁可用），state=1表示同步状态已被占用（锁被占用）；

自定义资源共享模式

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）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

```
1. isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
```

2. tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
3. tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
4. tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
5. tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

AQS源码实现

acquire(int):

acquire是一种以独占方式获取资源，如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。该方法是独占模式下线程获取共享资源的顶层入口。获取到资源后，线程就可以去执行其临界区代码了。下面是acquire()的源码：

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

acquire方法是一种互斥模式，且忽略中断。该方法至少执行一次`tryAcquire(int)`方法，如果tryAcquire(int)方法返回true，则acquire直接返回，否则当前线程需要进入队列进行排队。函数流程如下：

```
1. tryAcquire()尝试直接去获取资源，如果成功则直接返回;
```

2. addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

tryAcquire(int)：

tryAcquire尝试以独占的方式获取资源，如果获取成功，则直接返回true，否则直接返回false。该方法可以用于实现Lock中的tryLock()方法。该方法的默认实现是抛出UnsupportedOperationException，具体实现由自定义的扩展了AQS的同步类来实现。AQS在这里只负责定义了一个公共的方法框架。这里之所以没有定义成abstract，是因为独占模式下只用实现tryAcquire-tryRelease，而共享模式下只用实现tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口。

```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
```

addWaiter(Node)：

该方法用于将当前线程根据不同的模式（Node.EXCLUSIVE互斥模式、Node.SHARED共享模式）加入到等待队列的队尾，并返回当前线程所在的结点。如果队列不为空，则以通过compareAndSetTail方法以CAS的方式将当前线程节点加入到等待队列的末尾。否则，通过enq(node)方法初始化一个等待队列，并返回当前节点。源码如下：

```
private Node addWaiter(Node mode) {
Node node = new Node(Thread.currentThread(), mode);
// Try the fast path of enq; backup to full enq on failure
Node pred = tail;
if (pred != null) {
    node.prev = pred;
    if (compareAndSetTail(pred, node)) {
        pred.next = node;
        return node;
    }
}
enq(node);
return node;
}
```

enq(node):

enq(node)用于将当前节点插入等待队列，如果队列为空，则初始化当前队列。整个过程以CAS自旋的方式进行，直到成功加入队尾为止。源码如下：

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

acquireQueued(Node, int):

acquireQueued()用于队列中的线程自旋地以独占且不可中断的方式获取同步状态（acquire），直到拿到锁之后再返回。该方法的实现分成两部分：如果当前节点已经成为头结点，尝试获取锁（tryAcquire）成功，然后返回；否则检查当前节点是否应该被park，然后将该线程park并且检查当前线程是否被可以被中断。

```
final boolean acquireQueued(final Node node, int arg) {
    //标记是否成功拿到资源，默认false
    boolean failed = true;
    try {
        boolean interrupted = false;//标记等待过程中是否被中断过
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

shouldParkAfterFailedAcquire(Node, Node):

shouldParkAfterFailedAcquire方法通过对当前节点的前一个节点的状态进行判断，对当前节点做出不同的操作，至于每个Node的状态表示，可以参考接口文档。

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        /*
         * This node has already set status asking a release
         * to signal it, so it can safely park.
         */
        return true;
    if (ws > 0) {
        /*
         * Predecessor was cancelled. Skip over predecessors and
         * indicate retry.
         */
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        /*
         * waitStatus must be 0 or PROPAGATE.  Indicate that we
         * need a signal, but don't park yet.  Caller will need to
         * retry to make sure it cannot acquire before parking.
         */
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

parkAndCheckInterrupt():

该方法让线程去休息，真正进入等待状态。park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt()。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

```
/**
 * Convenience method to park and then check if interrupted
 *
 * @return {@code true} if interrupted
 */
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

release(int):

release(int)方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}

/**
 * Wakes up node's successor, if one exists.
 *
 * @param node the node
 */
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    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);
}
```

与acquire()方法中的tryAcquire()类似，tryRelease()方法也是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，上面已经提到了，release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false。　　unparkSuccessor(Node)方法用于唤醒等待队列中下一个线程。这里要注意的是，下一个线程并不一定是当前节点的next节点，而是下一个可以用来唤醒的线程，如果这个节点存在，调用unpark()方法唤醒。　　总之，release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

acquireShared(int)

　acquireShared(int)方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。下面是acquireShared()的源码：

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

doAcquireShared(int):

将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。源码如下：

```
/**
 * Acquires in shared uninterruptible mode.
 * @param arg the acquire argument
 */
private void doAcquireShared(int arg) {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    if (interrupted)
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

跟独占模式比，还有一点需要注意的是，这里只有线程是head.next时（“老二”），才会去尝试获取资源，有剩余的话还会唤醒之后的队友。那么问题就来了，假如老大用完后释放了5个资源，而老二需要6个，老三需要1个，老四需要2个。老大先唤醒老二，老二一看资源不够，他是把资源让给老三呢，还是不让？答案是否定的！老二会继续park()等待其他线程释放资源，也更不会去唤醒老三和老四了。独占模式，同一时刻只有一个线程去执行，这样做未尝不可；但共享模式下，多个线程是可以同时执行的，现在因为老二的资源需求量大，而把后面量小的老三和老四也都卡住了。当然，这并不是问题，只是AQS保证严格按照入队顺序唤醒罢了（保证公平，但降低了并发）。实现如下：

```
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    setHead(node);
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

此方法在setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合（比如还有剩余资源），还会去唤醒后继结点，毕竟是共享模式！至此，acquireShared()也要告一段落了。让我们再梳理一下它的流程：

1. tryAcquireShared()尝试获取资源，成功则直接返回；
2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

releaseShared(int):

releaseShared(int)方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。下面是releaseShared()的源码：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于独占下可重入的考量；而共享模式下的releaseShared()则没有这种要求，共享模式实质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等待结点。

```
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
                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同步组件

- CountDownLatch
- Semaphore
- CyclicBarrier
- ReentrantLock
- Condition
- FutureTask

CountDownLatch

- CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。
- 与CountDownLatch的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。
- 其他N 个线程必须引用闭锁对象，因为他们需要通知CountDownLatch对象，他们已经完成了各自的任务。这种通知机制是通过CountDownLatch.countDown()方法来完成的；每调用一次这个方法，在构造函数中初始化的count值就减1。所以当N个线程都调 用了这个方法，count的值等于0，然后主线程就能通过await()方法，恢复执行自己的任务。

```
private final static int threadCount = 200;
public static void main(String[] args) throws Exception {

        ExecutorService exec = Executors.newCachedThreadPool();
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    test(threadNum);
                } catch (Exception e) {
                    log.error("exception", e);
                } finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await();
        log.info("finish");
        exec.shutdown();
    }

private static void test(int threadNum) throws Exception {
```

```
        Thread.sleep(100);
        log.info("{}", threadNum);
        Thread.sleep(100);
    }
```

Semaphore

1. Semaphore也叫信号量，在JDK1.5被引入，可以用来控制同时访问特定资源的线程数量，通过协调各个线程，以保证合理的使用资源。
2. Semaphore内部维护了一组虚拟的许可，许可的数量可以通过构造函数的参数指定。
3. 访问特定资源前，必须使用acquire方法获得许可，如果许可数量为0，该线程则一直阻塞，直到有可用许可。.
4. 访问资源后，使用release释放许可。
5. Semaphore和ReentrantLock类似，获取许可有公平策略和非公平许可策略，默认情况下使用非公平策略。

```
private final static int threadCount = 20;

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

       ExecutorService exec = Executors.newCachedThreadPool();

       final Semaphore semaphore = new Semaphore(3);

       for (int i = 0; i < threadCount; i++) {
           final int threadNum = i;
           exec.execute(() -> {
               try {
                   if (semaphore.tryAcquire(5000, TimeUnit.MILLISECONDS)) { // 尝试获取一个许可
                       test(threadNum);
                       semaphore.release(); // 释放一个许可
                   }
               } catch (Exception e) {
                   log.error("exception", e);
               }
           });
       }
       exec.shutdown();
   }

   private static void test(int threadNum) throws Exception {
       log.info("{}", threadNum);
       Thread.sleep(1000);
   }
```

CyclicBarrier

CyclicBarrier也是一个同步辅助类 , 它允许一组线程相互等待 , 直到到达某个公共的屏障点 , 通过它可以完成多个线程之间相互等待 ,只有当每个线程都准备好之后, 才能各自继续往下执行后续的操作, 和 CountDownLatch相似的地方就是, 它也是通过计数器来实现的. 当某个线程调用了 await()方法之后, 该线程就进入了等待状态 . 而且计数器就进行 +1 操作 , 当计数器的值达到了我们设置的初始值的时候 , 之前调用了await() 方法而进入等待状态的线程会被唤醒继续执行后续的操作. 因为 CyclicBarrier释放线程之后可以重用, 所以又称之为循环屏障 . CyclicBarrier 使用场景和 CountDownLatch 很相似 , 可以用于多线程计算数据, 最后合并计算结果的应用场景 .

```
private static CyclicBarrier barrier = new CyclicBarrier(5);

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

        ExecutorService executor = Executors.newCachedThreadPool();

        for (int i = 0; i < 10; i++) {
            final int threadNum = i;
            Thread.sleep(1000);
            executor.execute(() -> {
                try {
                    race(threadNum);
                } catch (Exception e) {
                    log.error("exception", e);
                }
            });
        }
        executor.shutdown();
    }

private static void race(int threadNum) throws Exception {
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        barrier.await();
        log.info("{} continue", threadNum);
    }
```

CyclicBarrier 与 CountDownLatch 区别:

```
1. CountDownLatch的计数器只能使用一次 ， 而 CyclicBarrier 的计数器可以使用 reset重置 循环使用
```

2. CountDownLatch 主要事项 1 个 或者 n 个线程需要等待其它线程完成某项操作之后才能继续往下执行 , 其描述的是 1 个 或者 n 个线程与其它线程的关系 ; CyclicBarrier 主要是实现了 1 个或者多个线程之间相互等待,直到所有的线程都满足条件之后, 才执行后续的操作 , 其描述的是内部各个线程相互等待的关系 .
3. CyclicBarrier 假如有 5 个线程都调用了 await() 方法 , 那这个 5 个线程就等着 , 当这 5 个线程都准备好之后, 它们有各自往下继续执行 , 如果这 5 个线程在后续有一个计算发生错误了 , 这里可以重置计数器 , 并让这 5 个线程再执行一遍 .

ReentrantLock与锁

首先要知道 Java 中的锁主要分两类锁 , 一种是 synchronize锁 , 另外一种就是 J.U.C中 提供的锁 , J.U.C里核心的锁是 ReentrantLock。

ReentrantLock (可重入锁)与 synchronize 的区别:

1. **可重入性**:ReentrantLock 字面意思就是 再进入锁 ， 所以称之为可重入锁 ， synchronize 使用的锁也是可重入的. 它俩都是同一个线程进入一次锁的计数器就自增 1,所以要等到锁的计数器下降为 0 时才释放锁 .

2. **锁的实现**:synchronize 的锁是基于 JVM 来实现的 , ReentrantLock 是jdk 实现的. 通俗的来讲就是 操作系统来控制实现和用户编码实现的区别 .
3. **性能区别**:在 synchronize 关键字优化之前, 其性能比 ReentrantLock 差 , 但是优化过后 , 在两者都可以使用的情况下, 建议使用 synchronize, 主要是其写法比较容易.
4. **功能**：synchronize 写起来更简洁 , 它是由编译器来实现锁的加锁和释放 , 而ReentrantLock 需要我们手工申明加锁和释放锁 , 为了避免手工忘记释放锁而造成死锁 , 所以建议在final里申明和释放锁.

ReentrantLock 独有的功能:

- ReentrantLock可指定是公平锁还是非公平锁 ， 公平锁就是先等待的线程先获得锁.有先来后到之说. 而synchronize是非公平锁。
- 提供了一个 Condition 类 ， 可以分组唤醒需要唤醒的线程 ， 不像 synchronize要么随机唤醒一个线程 ， 要么唤醒全部线程。
- 提供能够中断等待锁的线程的机制。

```
public static int clientTotal = 5000;

// 同时并发执行的线程数
public static int threadTotal = 200;

public static int count = 0;

private final static Lock lock = new ReentrantLock();

public static void main(String[] args) throws Exception {
    ExecutorService executorService = Executors.newCachedThreadPool();
    final Semaphore semaphore = new Semaphore(threadTotal);
    final CountDownLatch countDownLatch = new CountDownLatch(clientTotal);
    for (int i = 0; i < clientTotal ; i++) {
        executorService.execute(() -> {
            try {
                semaphore.acquire();
                add();
                semaphore.release();
            } catch (Exception e) {
                log.error("exception", e);
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    log.info("count:{}", count);
}

private static void add() {
    lock.lock();
    try {
```

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

ReentrantReadWriteLock

ReentrantReadWriteLock是Lock的另一种实现方式，我们已经知道了ReentrantLock是一个排他锁，同一时间只允许一个线程访问，而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。相对于排他锁，提高了并发性。在实际应用中，大部分情况下对共享数据（如缓存）的访问都是读操作远多于写操作，这时ReentrantReadWriteLock能够提供比排他锁更好的并发性和吞吐量。

读写锁内部维护了两个锁，一个用于读操作，一个用于写操作。所有 ReadWriteLock实现都必须保证 writeLock操作的内存同步效果也要保持与相关 readLock的联系。也就是说，成功获取读锁的线程会看到写入锁之前版本所做的所有更新。

ReentrantReadWriteLock支持以下功能：

```
- 支持公平和非公平的获取锁的方式；
- 支持可重入。读线程在获取了读锁后还可以获取读锁；写线程在获取了写锁之后既可以再次获取写锁又可以获取读锁；
- 还允许从写入锁降级为读取锁，其实现方式是：先获取写入锁，然后获取读取锁，最后释放写入锁。但是，从读取锁升级到写入锁是不允许的.
- 读取锁和写入锁都支持锁获取期间的中断；
- Condition支持。仅写入锁提供了一个 Conditon 实现；读取锁不支持 Conditon ，readLock().newCondition()会抛出 UnsupportedOperationException。
```

```
private final Map<String, Data> map = new TreeMap<>();

private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

private final Lock readLock = lock.readLock();

private final Lock writeLock = lock.writeLock();

public Data get(String key) {
    readLock.lock();
    try {
        return map.get(key);
    } finally {
        readLock.unlock();
    }
}

public Set<String> getAllKeys() {
    readLock.lock();
    try {
        return map.keySet();
    } finally {
        readLock.unlock();
```

```
    }
}

public Data put(String key, Data value) {
    writeLock.lock();
    try {
        return map.put(key, value);
    } finally {
        readLock.unlock();
    }
}

class Data {

}
```

java线程池

java线程池框架

1. Executor是最基础的执行接口；
2. ExecutorService接口继承了Executor，在其上做了一些shutdown()、submit()的扩展，可以说是真正的线程池接口；
3. AbstractExecutorService抽象类实现了ExecutorService接口中的大部分方法；
4. TheadPoolExecutor继承了AbstractExecutorService，是线程池的具体实现
5. ScheduledExecutorService接口继承了ExecutorService接口，提供了带"周期执行"功能ExecutorService；
6. ScheduledThreadPoolExecutor既继承了TheadPoolExecutor线程池，也实现了ScheduledExecutorService接口，是带"周期执行"功能的线程池；
7. Executors是线程池的静态工厂，其提供了快捷创建线程池的静态方法。

Executor接口：

“执行者”接口，只提供了一个方法：

```
void execute(Runnable command);
```

可以用来执行已经提交的Runnable任务对象，这个接口提供了一种将“任务提交”与“任务执行”解耦的方法。

ExecutorService接口：

“执行者服务”接口，可以说是真正的线程池接口，在Executor接口的基础上做了一些扩展，主要是：

(A) 管理任务如何终止的 shutdown相关方法：

```
/**
 * 启动一次有序的关闭，之前提交的任务执行，但不接受新任务
 * 这个方法不会等待之前提交的任务执行完毕
 */
void shutdown();
```

```
/**
 * 试图停止所有正在执行的任务，暂停处理正在等待的任务，返回一个等待执行的任务列表
 * 这个方法不会等待正在执行的任务终止
 */
List<Runnable> shutdownNow();

/**
 * 如果已经被shutdown，返回true
 */
boolean isShutdown();

/**
 * 如果所有任务都已经被终止，返回true
 * 是否为终止状态
 */
boolean isTerminated();

/**
 * 在一个shutdown请求后，阻塞的等待所有任务执行完毕
 * 或者到达超时时间，或者当前线程被中断
 */
boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
```

(B) 可以生成用于追踪一个或多个异步任务执行结果的Future对象的 submit()相关方法:

```
/**
 * 提交一个可执行的任务，返回一个Future代表这个任务
 * 等到任务成功执行，Future#get()方法会返回null
 */
Future<?> submit(Runnable task);

/**
 * 提交一个可以执行的任务，返回一个Future代表这个任务
 * 等到任务执行结束，Future#get()方法会返回这个给定的result
 */
<T> Future<T> submit(Runnable task, T result);

/**
 * 提交一个有返回值的任务，并返回一个Future代表等待的任务执行的结果
 * 等到任务成功执行，Future#get()方法会返回任务执行的结果
 */
<T> Future<T> submit(Callable<T> task);
```

ScheduledExecutorService接口：

```
/**
 * 在给定延时后，创建并执行一个一次性的Runnable任务
 * 任务执行完毕后，ScheduledFuture#get()方法会返回null
 */
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

/**
```

```
 * 在给定延时后，创建并执行一个ScheduledFutureTask
 * ScheduledFuture 可以获取结果或取消任务
 */
public <V> ScheduledFuture<V> schedule(Callable<V> callable, ong delay, TimeUnit unit);

/**
 * 创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期
 * 也就是将在 initialDelay 后开始执行，然后在 initialDelay+period 后执行，接着在 initialDelay + 2 *
period 后执行，依此类推
 * 如果执行任务发生异常，随后的任务将被禁止，否则任务只会在被取消或者Executor被终止后停止
 * 如果任何执行的任务超过了周期，随后的执行会延时，不会并发执行
 */
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit);

/**
 * 创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟
 * 如果执行任务发生异常，随后的任务将被禁止，否则任务只会在被取消或者Executor被终止后停止
 */
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                 long initialDelay,
                                                 long delay,
                                                 TimeUnit unit);
```

ThreadPoolExecutor：

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

corePoolSize:

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

maximumPoolSize:

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize;

keepAliveTime:

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用;

workQueue：

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能；

几种排队的策略：

（1）不排队，直接提交

- 将任务直接交给线程处理而不保持它们，可使用SynchronousQueue； 如果不存在可用于立即运行任务的线程（即线程池中的线程都在工作），则试图把任务加入缓冲队列将会失败，因此会构造一个新的线程来处理新添加的任务，并将其加入到线程池中（corePoolSize-->maximumPoolSize扩容）
  Executors.newCachedThreadPool()采用的便是这种策略

（2）无界队列

- 可以使用LinkedBlockingQueue（基于链表的有界队列，FIFO），理论上是该队列可以对无限多的任务排队将导致在所有corePoolSize线程都工作的情况下将新任务加入到队列中。这样，创建的线程就不会超过corePoolSize，也因此，maximumPoolSize的值也就无效了

（3）有界队列

- 可以使用ArrayBlockingQueue（基于数组结构的有界队列，FIFO），并指定队列的最大长度；使用有界队列可以防止资源耗尽，但也会造成超过队列大小和maximumPoolSize后，提交的任务被拒绝的问题，比较难调整和控制。

threadFactory：

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名：

```
/**
* The default thread factory
*/
static class DefaultThreadFactory implements ThreadFactory {
private static final AtomicInteger poolNumber = new AtomicInteger(1);
private final ThreadGroup group;
private final AtomicInteger threadNumber = new AtomicInteger(1);
private final String namePrefix;

DefaultThreadFactory() {
    SecurityManager s = System.getSecurityManager();
    group = (s != null) ? s.getThreadGroup() :
                          Thread.currentThread().getThreadGroup();
    namePrefix = "pool-" +
                  poolNumber.getAndIncrement() +
                 "-thread-";
}

public Thread newThread(Runnable r) {
    Thread t = new Thread(group, r,
                          namePrefix + threadNumber.getAndIncrement(),
                          0);
    if (t.isDaemon())
        t.setDaemon(false);
    if (t.getPriority() != Thread.NORM_PRIORITY)
```

```
        t.setPriority(Thread.NORM_PRIORITY);
    return t;
}
}
```

Executors静态工厂里默认的threadFactory，线程的命名规则是“pool-数字-thread-数字”.

RejectedExecutionHandler（饱和策略）：

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

ThreadPoolExecutor线程池执行流程:

根据ThreadPoolExecutor源码前面大段的注释，我们可以看出，当试图通过execute方法将一个Runnable任务添加到线程池中时，按照如下顺序来处理：

1. 如果线程池中的线程数量少于corePoolSize，就创建新的线程来执行新添加的任务；
2. 如果线程池中的线程数量大于等于corePoolSize，但队列workQueue未满，则将新添加的任务放到workQueue中，按照FIFO的原则依次等待执行（线程池中有线程空闲出来后依次将队列中的任务交付给空闲的线程执行）；
3. 如果线程池中的线程数量大于等于corePoolSize，且队列workQueue已满，但线程池中的线程数量小于maximumPoolSize，则会创建新的线程来处理被添加的任务；
4. 如果线程池中的线程数量等于了maximumPoolSize，就用RejectedExecutionHandler来做拒绝处理

总结，当有新的任务要处理时，先看线程池中的线程数量是否大于corePoolSize，再看缓冲队列workQueue是否满，最后看线程池中的线程数量是否大于maximumPoolSize

另外，当线程池中的线程数量大于corePoolSize时，如果里面有线程的空闲时间超过了keepAliveTime，就将其移除线程池.

Executors静态工厂创建几种常用线程池

newFixedThreadPool:

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

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

创建一个指定工作线程数的线程池，其中参数 corePoolSize 和 maximumPoolSize 相等，阻塞队列基于 LinkedBlockingQueue

它是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是在线程池空闲时，即线程池中没有可运行任务时，它也不会释放工作线程，还会占用一定的系统资源。

newSingleThreadExecutor:

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

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

初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行，内部使用LinkedBlockingQueue作为阻塞队列.

newCachedThreadPool:

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

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

创建一个可缓存工作线程的线程池，默认存活时间60秒，线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；

在没有任务执行时，当线程的空闲时间超过keepAliveTime，则工作线程将会终止，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销。

newScheduledThreadPool：

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(
        int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
```

初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据

注意：

ScheduledExecutorService#scheduleAtFixedRate() 指的是“以固定的频率”执行，period（周期）指的是两次成功执行之间的时间

比如，scheduleAtFixedRate(command, 5, 2, second)，第一次开始执行是5s后，假如执行耗时1s，那么下次开始执行是7s后，再下次开始执行是9s后

而ScheduledExecutorService#scheduleWithFixedDelay() 指的是“以固定的延时”执行，delay（延时）指的是一次执行终止和下一次执行开始之间的延迟.

execute()内部原理

线程池状态

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
```

```
// runState is stored in the high-order bits
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;

// Packing and unpacking ctl
private static int runStateOf(int c)     { return c & ~CAPACITY; }
private static int workerCountOf(int c)  { return c & CAPACITY; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

其中ctl这个AtomicInteger的功能很强大，其高3位用于维护线程池运行状态，低29位维护线程池中线程数量

```
1. RUNNING：-1<<COUNT_BITS，即高3位为1，低29位为0，该状态的线程池会接收新任务，也会处理在阻塞队列中等待处理的任务
```

2. SHUTDOWN：0<<COUNT_BITS，即高3位为0，低29位为0，该状态的线程池不会再接收新任务，但还会处理已经提交到阻塞队列中等待处理的任务
3. STOP：1<<COUNT_BITS，即高3位为001，低29位为0，该状态的线程池不会再接收新任务，不会处理在阻塞队列中等待的任务，而且还会中断正在运行的任务
4. TIDYING：2<<COUNT_BITS，即高3位为010，低29位为0，所有任务都被终止了，workerCount为0，为此状态时还将调用terminated()方法
5. TERMINATED：3<<COUNT_BITS，即高3位为100，低29位为0，terminated()方法调用完成后变成此状态

这些状态均由int型表示，大小关系为 RUNNING<SHUTDOWN<STOP<TIDYING<TERMINATED，这个顺序基本上也是遵循线程池从 运行 到 终止这个过程。

```
- runStateOf(int c)  方法：c & 高3位为1，低29位为0的~CAPACITY，用于获取高3位保存的线程池状态
- workerCountOf(int c)方法：c & 高3位为0，低29位为1的CAPACITY，用于获取低29位的线程数量
- ctlOf(int rs, int wc)方法：参数rs表示runState，参数wc表示workerCount，即根据runState和workerCount打包合并成ctl
```

内部原理

```
/**
 * 在未来的某个时刻执行给定的任务。这个任务用一个新线程执行，或者用一个线程池中已经存在的线程执行
 * 如果任务无法被提交执行，要么是因为这个Executor已经被shutdown关闭，要么是已经达到其容量上限，任务会被当前的RejectedExecutionHandler处理
 */
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    /*
     * 如果运行的线程少于corePoolSize，尝试开启一个新线程去运行command，command作为这个线程的第一个任务
     * 如果任务成功放入队列，我们仍需要一个双重校验去确认是否应该新建一个线程（因为可能存在有些线程在我们上次检查后死了） 或者 从我们进入这个方法后，pool被关闭了
     * 所以我们需要再次检查state，如果线程池停止了需要回滚入队列，如果池中没有线程了，新开启 一个线程
     * 如果无法将任务入队列（可能队列满了），需要新开区一个线程（自己：往maxPoolSize发展）
     * 如果失败了，说明线程池shutdown 或者 饱和了，所以我们拒绝任务
```

```
     */
    int c = ctl.get();

    /**
     * 1、如果当前线程数少于corePoolSize（可能是由于addWorker()操作已经包含对线程池状态的判断，如此处没加，而入workQueue前加了）
     */
    if (workerCountOf(c) < corePoolSize) {
        //addWorker()成功，返回
        if (addWorker(command, true))
            return;

        /**
         * 没有成功addWorker()，再次获取c（凡是需要再次用ctl做判断时，都会再次调用ctl.get()）
         * 失败的原因可能是：
         * 1、线程池已经shutdown，shutdown的线程池不再接收新任务
         * 2、workerCountOf(c) < corePoolSize 判断后，由于并发，别的线程先创建了worker线程，导致workerCount>=corePoolSize
         */
        c = ctl.get();
    }

    /**
     * 2、如果线程池RUNNING状态，且入队列成功
     */
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();//再次校验位

        /**
         * 再次校验放入workerQueue中的任务是否能被执行
         * 1、如果线程池不是运行状态了，应该拒绝添加新任务，从workQueue中删除任务
         * 2、如果线程池是运行状态，或者从workQueue中删除任务失败（刚好有一个线程执行完毕，并消耗了这个任务），确保还有线程执行任务（只要有一个就够了）
         */
        //如果再次校验过程中，线程池不是RUNNING状态，并且remove(command)--workQueue.remove()成功，拒绝当前command
        if (! isRunning(recheck) && remove(command))
            reject(command);
        //如果当前worker数量为0，通过addWorker(null, false)创建一个线程，其任务为null
        //为什么只检查运行的worker数量是不是0呢？？ 为什么不和corePoolSize比较呢？？
        //只保证有一个worker线程可以从queue中获取任务执行就行了？？
        //因为只要还有活动的worker线程，就可以消费workerQueue中的任务
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);  //第一个参数为null，说明只为新建一个worker线程，没有指定firstTask
                                     //第二个参数为true代表占用corePoolSize，false占用maxPoolSize
    }
    /**
     * 3、如果线程池不是running状态 或者 无法入队列
     *   尝试开启新线程，扩容至maxPoolSize，如果addWork(command, false)失败了，拒绝当前command
     */
    else if (!addWorker(command, false))
```

```
        reject(command);
}
```

execute(Runnable command)

参数： command 提交执行的任务，不能为空 执行流程：

```
1. 如果线程池当前线程数量少于corePoolSize，则addWorker(command, true)创建新worker线程，如创建成功返回，如没创建成功，则执行后续步骤；
addWorker(command, true)失败的原因可能是：
    - A、线程池已经shutdown，shutdown的线程池不再接收新任务
    - B、workerCountOf(c) < corePoolSize 判断后，由于并发，别的线程先创建了worker线程，导致workerCount>=corePoolSize
```

2. 如果线程池还在running状态，将task加入workQueue阻塞队列中，如果加入成功，进行double-check，如果加入失败（可能是队列已满），则执行后续步骤； double-check主要目的是判断刚加入workQueue阻塞队列的task是否能被执行
    - A、如果线程池已经不是running状态了，应该拒绝添加新任务，从workQueue中删除任务
    - B、如果线程池是运行状态，或者从workQueue中删除任务失败（刚好有一个线程执行完毕，并消耗了这个任务），确保还有线程执行任务（只要有一个就够了）
    3. 如果线程池不是running状态 或者 无法入队列，尝试开启新线程，扩容至maxPoolSize，如果addWork(command, false)失败了，拒绝当前command

addWorker() -- 添加worker线程

```
/**
 * 检查根据当前线程池的状态和给定的边界(core or maximum)是否可以创建一个新的worker
 * 如果是这样的话，worker的数量做相应的调整，如果可能的话，创建一个新的worker并启动，参数中的firstTask作为worker的第一个任务
 * 如果方法返回false，可能因为pool已经关闭或者调用过了shutdown
 * 如果线程工厂创建线程失败，也会失败，返回false
 * 如果线程创建失败，要么是因为线程工厂返回null，要么是发生了OutOfMemoryError
 */
private boolean addWorker(Runnable firstTask, boolean core) {
    //外层循环，负责判断线程池状态
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c); //状态

        // Check if queue empty only if necessary.
        /**
         * 线程池的state越小越是运行状态，runnbale=-1，shutdown=0,stop=1,tidying=2，terminated=3
         * 1、如果线程池state已经至少是shutdown状态了
         * 2、并且以下3个条件任意一个是false
         *   rs == SHUTDOWN          (隐含：rs>=SHUTDOWN) false情况： 线程池状态已经超过shutdown，可能是stop、tidying、terminated其中一个，即线程池已经终止
         *   firstTask == null       (隐含：rs==SHUTDOWN) false情况： firstTask不为空，rs==SHUTDOWN 且 firstTask不为空，return false，场景是在线程池已经shutdown后，还要添加新的任务，拒绝
         *   ! workQueue.isEmpty()   (隐含：rs==SHUTDOWN，firstTask==null) false情况：workQueue为空，当firstTask为空时是为了创建一个没有任务的线程，再从workQueue中获取任务，如果workQueue已经
```

```
为空，那么就没有添加新worker线程的必要了
         * return false，即无法addWorker()
         */
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        //内层循环，负责worker数量+1
        for (;;) {
            int wc = workerCountOf(c); //worker数量

            //如果worker数量>线程池最大上限CAPACITY（即使用int低29位可以容纳的最大值）
            //或者( worker数量>corePoolSize 或  worker数量>maximumPoolSize )，即已经超过了给定的边
界
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;

            //调用unsafe CAS操作，使得worker数量+1，成功则跳出retry循环
            if (compareAndIncrementWorkerCount(c))
                break retry;

            //CAS worker数量+1失败，再次读取ctl
            c = ctl.get();  // Re-read ctl

            //如果状态不等于之前获取的state，跳出内层循环，继续去外层循环判断
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
            // else CAS失败时因为workerCount改变了，继续内层循环尝试CAS对worker数量+1
        }
    }

    /**
     * worker数量+1成功的后续操作
     * 添加到workers Set集合，并启动worker线程
     */
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        final ReentrantLock mainLock = this.mainLock;
        w = new Worker(firstTask); //1、设置worker这个AQS锁的同步状态state=-1
                                   //2、将firstTask设置给worker的成员变量firstTask
                                   //3、使用worker自身这个runnable，调用ThreadFactory创建一个线
程，并设置给worker的成员变量thread
        final Thread t = w.thread;
        if (t != null) {
            mainLock.lock();
            try {
                //-----------------------------------------这部分代码是上锁的
```

```
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                // 当获取到锁后，再次检查
                int c = ctl.get();
                int rs = runStateOf(c);

                //如果线程池在运行running<shutdown 或者 线程池已经shutdown，且firstTask==null（可能是workQueue中仍有未执行完成的任务，创建没有初始任务的worker线程执行）
                //worker数量-1的操作在addWorkerFailed()
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable   线程已经启动，抛非法线程状态异常
                        throw new IllegalThreadStateException();

                    workers.add(w);//workers是一个HashSet<Worker>

                    //设置最大的池大小largestPoolSize，workerAdded设置为true
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
              //-------------------------------------------
            }
            finally {
                mainLock.unlock();
            }

            //如果往HashSet中添加worker成功，启动线程
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        //如果启动线程失败
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

参数：

firstTask： worker线程的初始任务，可以为空

core：true：将corePoolSize作为上限，false：将maximumPoolSize作为上限

addWorker方法有4种传参的方式：

1. addWorker(command, true)
2. addWorker(command, false)

3. addWorker(null, false)
4. addWorker(null, true)

在execute方法中就使用了前3种，结合这个核心方法进行以下分析 第一个：线程数小于corePoolSize时，放一个需要处理的task进Workers Set。如果Workers Set长度超过corePoolSize，就返回false 第二个：当队列被放满时，就尝试将这个新来的task直接放入Workers Set，而此时Workers Set的长度限制是maximumPoolSize。如果线程池也满了的话就返回false 第三个：放入一个空的task进workers Set，长度限制是maximumPoolSize。这样一个task为空的worker在线程执行的时候会去任务队列里拿任务，这样就相当于创建了一个新的线程，只是没有马上分配任务 第四个：这个方法就是放一个null的task进Workers Set，而且是在小于corePoolSize时，如果此时Set中的数量已经达到corePoolSize那就返回false，什么也不干。实际使用中是在prestartAllCoreThreads()方法，这个方法用来为线程池预先启动corePoolSize个worker等待从workQueue中获取任务执行 **执行流程：** 1、判断线程池当前是否为可以添加worker线程的状态，可以则继续下一步，不可以return false： A、线程池状态>shutdown，可能为stop、tidying、terminated，不能添加worker线程 B、线程池状态==shutdown，firstTask不为空，不能添加worker线程，因为shutdown状态的线程池不接收新任务 C、线程池状态==shutdown，firstTask==null，workQueue为空，不能添加worker线程，因为firstTask为空是为了添加一个没有任务的线程再从workQueue获取task，而workQueue为空，说明添加无任务线程已经没有意义 2、线程池当前线程数量是否超过上限（corePoolSize 或 maximumPoolSize），超过了return false，没超过则对workerCount+1，继续下一步 3、在线程池的ReentrantLock保证下，向Workers Set中添加新创建的worker实例，添加完成后解锁，并启动worker线程，如果这一切都成功了，return true，如果添加worker入Set失败或启动失败，调用addWorkerFailed()逻辑。

内部类Worker

```
/**
 * Worker类大体上管理着运行线程的中断状态 和 一些指标
 * Worker类投机取巧的继承了AbstractQueuedSynchronizer来简化在执行任务时的获取、释放锁
 * 这样防止了中断在运行中的任务，只会唤醒(中断)在等待从workQueue中获取任务的线程
 * 解释:
 *    为什么不直接执行execute(command)提交的command，而要在外面包一层worker呢？？
 *    主要是为了控制中断
 *    用什么控制？？
 *    用AQS锁，当运行时上锁，就不能中断，TreadPoolExecutor的shutdown()方法中断前都要获取worker锁
 *    只有在等待从workQueue中获取任务getTask()时才能中断
 * worker实现了一个简单的不可重入的互斥锁，而不是用ReentrantLock可重入锁
 * 因为我们不想让在调用比如setCorePoolSize()这种线程池控制方法时可以再次获取锁(重入)
 * 解释:
 *    setCorePoolSize()时可能会interruptIdleWorkers()，在对一个线程interrupt时会要w.tryLock()
 *    如果可重入，就可能会在对线程池操作的方法中中断线程，类似方法还有:
 *    setMaximumPoolSize()
 *    setKeppAliveTime()
 *    allowCoreThreadTimeOut()
 *    shutdown()
 * 此外，为了让线程真正开始后才可以中断，初始化lock状态为负值(-1)，在开始runWorker()时将state置为0，而state>=0才可以中断
 *
 *
 * Worker继承了AQS，实现了Runnable，说明其既是一个可运行的任务，也是一把锁（不可重入）
 */
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;

    /** Thread this worker is running in.  Null if factory fails. */
    final Thread thread; //利用ThreadFactory和 Worker这个Runnable创建的线程对象

    /** Initial task to run.  Possibly null. */
    Runnable firstTask;

    /** Per-thread task counter */
    volatile long completedTasks;

    /**
     * Creates with given first task and thread from ThreadFactory.
     * @param firstTask the first task (null if none)
     */
    Worker(Runnable firstTask) {
        //设置AQS的同步状态private volatile int state，是一个计数器，大于0代表锁已经被获取
        setState(-1); // inhibit interrupts until runWorker
                      // 在调用runWorker()前，禁止interrupt中断，在interruptIfStarted()方法中会判
断 getState()>=0
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this); //根据当前worker创建一个线程对象
                                                          //当前worker本身就是一个runnable任务，
也就是不会用参数的firstTask创建线程，而是调用当前worker.run()时调用firstTask.run()
    }

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

    // Lock methods
    //
    // The value 0 represents the unlocked state. 0代表“没被锁定”状态
    // The value 1 represents the locked state. 1代表“锁定”状态

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

    /**
     * 尝试获取锁
     * 重写AQS的tryAcquire()，AQS本来就是让子类来实现的
     */
    protected boolean tryAcquire(int unused) {
        //尝试一次将state从0设置为1，即“锁定”状态，但由于每次都是state 0->1，而不是+1，那么说明不可重入
        //且state==-1时也不会获取到锁
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread()); //设置exclusiveOwnerThread=当前线
程
```

```
            return true;
        }
        return false;
    }

    /**
     * 尝试释放锁
     * 不是state-1，而是置为0
     */
    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(); }

    /**
     * 中断（如果运行）
     * shutdownNow时会循环对worker线程执行
     * 且不需要获取worker锁，即使在worker运行时也可以中断
     */
    void interruptIfStarted() {
        Thread t;
        //如果state>=0、t!=null、且t没有被中断
        //new Worker()时state==-1，说明不能中断
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}
```

Worker类:

Worker类本身既实现了Runnable，又继承了AbstractQueuedSynchronizer（以下简称AQS），所以其既是一个可执行的任务，又可以达到锁的效果 **new Worker()** 1、将AQS的state置为-1，在runWoker()前不允许中断 2、待执行的任务会以参数传入，并赋予firstTask 3、用Worker这个Runnable创建Thread

之所以Worker自己实现Runnable，并创建Thread，在firstTask外包一层，是因为要通过Worker控制中断，而firstTask这个工作任务只是负责执行业务 **Worker控制中断主要有以下几方面**： 1、初始AQS状态为-1，此时不允许中断interrupt()，只有在worker线程启动了，执行了runWoker()，将state置为0，才能中断 不允许中断体现在：

```
    - A、shutdown()线程池时，会对每个worker tryLock()上锁，而worker类这个AQS的tryAcquire()方法是固定将state从0->1，故初始状态state==-1时tryLock()失败，没发interrupt()
    - B、shutdownNow()线程池时，不用tryLock()上锁，但调用worker.interruptIfStarted()终止worker，interruptIfStarted()也有state>0才能interrupt的逻辑
```

2、为了防止某种情况下，在运行中的worker被中断，runWorker()每次运行任务时都会lock()上锁，而shutdown()这类可能会终止worker的操作需要先获取worker的锁，这样就防止了中断正在运行的线程

Worker实现的AQS为不可重入锁，为了是在获得worker锁的情况下再进入其它一些需要加锁的方法

Worker和Task的区别： Worker是线程池中的线程，而Task虽然是runnable，但是并没有真正执行，只是被Worker调用了run方法，后面会看到这部分的实现。

runWorker() -- 执行任务

```
/**
 * 我们可能使用一个初始化任务开始，即firstTask为null
 * 然后只要线程池在运行，我们就从getTask()获取任务
 * 如果getTask()返回null，则worker由于改变了线程池状态或参数配置而退出
 * 其它退出因为外部代码抛异常了，这会使得completedAbruptly为true，这会导致在processWorkerExit()方法中替换当前线程
 *
 * 在任何任务执行之前，都需要对worker加锁去防止在任务运行时，其它的线程池中断操作
 * clearInterruptsForTaskRun保证除非线程池正在stoping，线程不会被设置中断标示
 *
 * 每个任务执行前会调用beforeExecute()，其中可能抛出一个异常，这种情况下会导致线程die（跳出循环，且completedAbruptly==true），没有执行任务
 * 因为beforeExecute()的异常没有cache住，会上抛，跳出循环
 *
 * 假定beforeExecute()正常完成，我们执行任务
 * 汇总任何抛出的异常并发送给afterExecute(task, thrown)
 * 因为我们不能在Runnable.run()方法中重新上抛Throwables，我们将Throwables包装到Errors上抛（会到线程的UncaughtExceptionHandler去处理）
 * 任何上抛的异常都会导致线程die
 *
 * 任务执行结束后，调用afterExecute()，也可能抛异常，也会导致线程die
 * 根据JLS Sec 14.20，这个异常（finally中的异常）会生效
 *
 * @param w the worker
 */
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
                // new Worker()是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
    boolean completedAbruptly = true; //是否“突然完成”，如果是由于异常导致的进入finally，那么completedAbruptly==true就是突然完成的
    try {
        /**
         * 如果task不为null，或者从阻塞队列中getTask()不为null
         */
        while (task != null || (task = getTask()) != null) {
            w.lock(); //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

            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
```

```
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            /**
             * clearInterruptsForTaskRun操作
             * 确保只有在线程stoping时，才会被设置中断标示，否则清除中断标示
             * 1、如果线程池状态>=stop，且当前线程没有设置中断状态，wt.interrupt()
             * 2、如果一开始判断线程池状态<stop，但Thread.interrupted()为true，即线程已经被中断，又清除了中断标示，再次判断线程池状态是否>=stop
             *    是，再次设置中断标示，wt.interrupt()
             *    否，不做操作，清除中断标示后进行后续步骤
             */
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt(); //当前线程调用interrupt()中断

            try {
                //执行前（子类实现）
                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 {
                    //执行后（子类实现）
                    afterExecute(task, thrown); //这里就考验catch和finally的执行顺序了，因为要以thrown为参数
                }
            }
            finally {
                task = null; //task置为null
                w.completedTasks++; //完成任务数+1
                w.unlock(); //解锁
            }
        }

        completedAbruptly = false;
    }
    finally {
        //处理worker的退出
        processWorkerExit(w, completedAbruptly);
```

```
    }
}
```

runWorker(Worker w) 执行流程：

```
1. Worker线程启动后，通过Worker类的run()方法调用runWorker(this)
```

2. 执行任务之前，首先worker.unlock()，将AQS的state置为0，允许中断当前worker线程
3. 开始执行firstTask，调用task.run()，在执行任务前会上锁wroker.lock()，在执行完任务后会解锁，为了防止在任务运行时被线程池一些中断操作中断
4. 在任务执行前后，可以根据业务场景自定义beforeExecute() 和 afterExecute()方法
5. 无论在beforeExecute()、task.run()、afterExecute()发生异常上抛，都会导致worker线程终止，进入processWorkerExit()处理worker退出的流程
6. 如正常执行完当前task后，会通过getTask()从阻塞队列中获取新任务，当队列中没有任务，且获取任务超时，那么当前worker也会进入退出流程

getTask() -- 获取任务

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

    /**
     * 外层循环
     * 用于判断线程池状态
     */
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        /**
         * 对线程池状态的判断，两种情况会workerCount-1，并且返回null
         * 线程池状态为shutdown，且workQueue为空（反映了shutdown状态的线程池还是要执行workQueue中剩余的任务的）
         * 线程池状态为stop（shutdownNow()会导致变成STOP）（此时不用考虑workQueue的情况）
         */
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount(); //循环的CAS减少worker数量，直到成功
            return null;
        }

        boolean timed;      // Are workers subject to culling?
                            // 是否需要定时从workQueue中获取

        /**
         * 内层循环
         * 要么break去workQueue获取任务
         * 要么超时了，worker count-1
         */
        for (;;) {
```

```
            int wc = workerCountOf(c);
            timed = allowCoreThreadTimeOut || wc > corePoolSize; //allowCoreThreadTimeOut默认
为false
                                                                 //如果allowCoreThreadTimeOut
为true，说明corePoolSize和maximum都需要定时

            //如果当前执行线程数<maximumPoolSize，并且timedOut 和 timed 任一为false，跳出循环，开始从
workQueue获取任务
            if (wc <= maximumPoolSize && ! (timedOut && timed))
                break;

            /**
             * 如果到了这一步，说明要么线程数量超过了maximumPoolSize（可能maximumPoolSize被修改了）
             * 要么既需要计时timed==true，也超时了timedOut==true
             * worker数量-1，减一执行一次就行了，然后返回null，在runWorker()中会有逻辑减少worker线程
             * 如果本次减一失败，继续内层循环再次尝试减一
             */
            if (compareAndDecrementWorkerCount(c))
                return null;

            //如果减数量失败，再次读取ctl
            c = ctl.get();  // Re-read ctl

            //如果线程池运行状态发生变化，继续外层循环
            //如果状态没变，继续内层循环
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }

        try {
            //poll() - 使用  LockSupport.parkNanos(this, nanosTimeout) 挂起一段时间，
interrupt()时不会抛异常，但会有中断响应
            //take() - 使用 LockSupport.park(this) 挂起，interrupt()时不会抛异常，但会有中断响应
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :    //大于corePoolSize
                workQueue.take();                                        //小于等于
corePoolSize

            //如获取到了任务就返回
            if (r != null)
                return r;

            //没有返回，说明超时，那么在下一次内层循环时会进入worker count减一的步骤
            timedOut = true;
        }
        /**
              * blockingQueue的take()阻塞使用LockSupport.park(this)进入wait状态的，对
LockSupport.park(this)进行interrupt不会抛异常，但还是会有中断响应
              * 但AQS的ConditionObject的await()对中断状态做了判断，会报告中断状态
reportInterruptAfterWait(interruptMode)
              * 就会上抛InterruptedException，在此处捕获，重新开始循环
              * 如果是由于shutdown()等操作导致的空闲worker中断响应，在外层循环判断状态时，可能return
```

```
null
                */
            catch (InterruptedException retry) {
                timedOut = false; //响应中断，重新开始，中断状态会被清除
            }
        }
    }
```

getTask() 执行流程： 1、首先判断是否可以满足从workQueue中获取任务的条件，不满足return null A、线程池状态是否满足： （a）shutdown状态 + workQueue为空 或 stop状态，都不满足，因为被shutdown后还是要执行workQueue剩余的任务，但workQueue也为空，就可以退出了 （b）stop状态，shutdownNow()操作会使线程池进入stop，此时不接受新任务，中断正在执行的任务，workQueue中的任务也不执行了，故return null返回 B、线程数量是否超过maximumPoolSize 或 获取任务是否超时 （a）线程数量超过maximumPoolSize可能是线程池在运行时被调用了setMaximumPoolSize()被改变了大小，否则已经addWorker()成功不会超过maximumPoolSize （b）如果 当前线程数量>corePoolSize，才会检查是否获取任务超时，这也体现了当线程数量达到maximumPoolSize后，如果一直没有新任务，会逐渐终止worker线程直到corePoolSize 2、如果满足获取任务条件，根据是否需要定时获取调用不同方法： A、workQueue.poll()：如果在keepAliveTime时间内，阻塞队列还是没有任务，返回null B、workQueue.take()：如果阻塞队列为空，当前线程会被挂起等待；当队列中有任务加入时，线程被唤醒，take方法返回任务 3、在阻塞从workQueue中获取任务时，可以被interrupt()中断，代码中捕获了InterruptedException，重置timedOut为初始值false，再次执行第1步中的判断，满足就继续获取任务，不满足return null，会进入worker退出的流程

processWorkerExit() -- worker线程退出

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    /**
     * 1、worker数量-1
     * 如果是突然终止，说明是task执行时异常情况导致，即run()方法执行时发生了异常，那么正在工作的worker线程数量需要-1
     * 如果不是突然终止，说明是worker线程没有task可执行了，不用-1，因为已经在getTask()方法中-1了
     */
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted 代码和注释正好相反啊
        decrementWorkerCount();

    /**
     * 2、从workers Set中移除worker
     */
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        completedTaskCount += w.completedTasks; //把worker的完成任务数加到线程池的完成任务数
        workers.remove(w); //从HashSet<Worker>中移除
    } finally {
        mainLock.unlock();
    }

    /**
     * 3、在对线程池有负效益的操作时，都需要“尝试终止”线程池
     * 主要是判断线程池是否满足终止的状态
     * 如果状态满足，但还有线程池还有线程，尝试对其发出中断响应，使其能进入退出流程
     * 没有线程了，更新状态为tidying->terminated
     */
```

```
    tryTerminate();

    /**
     * 4、是否需要增加worker线程
     * 线程池状态是running 或 shutdown
     * 如果当前线程是突然终止的，addWorker()
     * 如果当前线程不是突然终止的，但当前线程数量 < 要维护的线程数量，addWorker()
     * 故如果调用线程池shutdown()，直到workQueue为空前，线程池都会维持corePoolSize个线程，然后再逐渐销毁这corePoolSize个线程
     */
    int c = ctl.get();
    //如果状态是running、shutdown，即tryTerminate()没有成功终止线程池，尝试再添加一个worker
    if (runStateLessThan(c, STOP)) {
        //不是突然完成的，即没有task任务可以获取而完成的，计算min，并根据当前worker数量判断是否需要addWorker()
        if (!completedAbruptly) {
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize; //allowCoreThreadTimeOut默认为false，即min默认为corePoolSize

            //如果min为0，即不需要维持核心线程数量，且workQueue不为空，至少保持一个线程
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;

            //如果线程数量大于最少数量，直接返回，否则下面至少要addWorker一个
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }

        //添加一个没有firstTask的worker
        //只要worker是completedAbruptly突然终止的，或者线程数量小于要维护的数量，就新添一个worker线程，即使是shutdown状态
        addWorker(null, false);
    }
}
```

参数： worker： 要结束的worker completedAbruptly： 是否突然完成（是否因为异常退出） 执行流程： 1、worker数量-1 A、如果是突然终止，说明是task执行时异常情况导致，即run()方法执行时发生了异常，那么正在工作的worker线程数量需要-1 B、如果不是突然终止，说明是worker线程没有task可执行了，不用-1，因为已经在getTask()方法中-1了 2、从Workers Set中移除worker，删除时需要上锁mainlock 3、tryTerminate()：在对线程池有负效益的操作时，都需要"尝试终止"线程池，大概逻辑： 判断线程池是否满足终止的状态 A、如果状态满足，但还有线程池还有线程，尝试对其发出中断响应，使其能进入退出流程 B、没有线程了，更新状态为tidying->terminated 4、是否需要增加worker线程，如果线程池还没有完全终止，仍需要保持一定数量的线程 线程池状态是running 或 shutdown A、如果当前线程是突然终止的，addWorker() B、如果当前线程不是突然终止的，但当前线程数量 < 要维护的线程数量，addWorker() 故如果调用线程池shutdown()，直到workQueue为空前，线程池都会维持corePoolSize个线程，然后再逐渐销毁这corePoolSize个线程

终止池原理

shutdown()

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock(); //上锁

    try {
        //判断调用者是否有权限shutdown线程池
        checkShutdownAccess();

        //CAS+循环设置线程池状态为shutdown
        advanceRunState(SHUTDOWN);

        //中断所有空闲线程
        interruptIdleWorkers();

        onShutdown(); // hook for ScheduledThreadPoolExecutor
    }
    finally {
        mainLock.unlock(); //解锁
    }

    //尝试终止线程池
    tryTerminate();
}
```

shutdown()执行流程：

1、上锁，mainLock是线程池的主锁，是可重入锁，当要操作workers set这个保持线程的HashSet时，需要先获取mainLock，还有当要处理largestPoolSize、completedTaskCount这类统计数据时需要先获取mainLock

2、判断调用者是否有权限shutdown线程池

3、使用CAS操作将线程池状态设置为shutdown，shutdown之后将不再接收新任务

4、中断所有空闲线程 interruptIdleWorkers()

5、onShutdown()，ScheduledThreadPoolExecutor中实现了这个方法，可以在shutdown()时做一些处理

6、解锁

7、尝试终止线程池 tryTerminate()

可以看到shutdown()方法最重要的几个步骤是：更新线程池状态为shutdown、中断所有空闲线程、tryTerminated()尝试终止线程池

interruptIdleWorkers():

```
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock(); //上锁
    try {
        for (Worker w : workers) {
            Thread t = w.thread;

            if (!t.isInterrupted() && w.tryLock()) {
                try {
```

```
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock(); //解锁
    }
}
```

interruptIdleWorkers() 首先会获取mainLock锁，因为要迭代workers set，在中断每个worker前，需要做两个判断：

1、线程是否已经被中断，是就什么都不做

2、worker.tryLock() 是否成功

tryTerminated():

```
final void tryTerminate() {
    //这个for循环主要是和进入关闭线程池操作的CAS判断结合使用的
    for (;;) {
        int c = ctl.get();

        /**
         * 线程池是否需要终止
         * 如果以下3中情况任一为true，return，不进行终止
         * 1、还在运行状态
         * 2、状态是TIDYING、或 TERMINATED，已经终止过了
         * 3、SHUTDOWN 且 workQueue不为空
         */
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;

        /**
         * 只有shutdown状态 且 workQueue为空，或者 stop状态能执行到这一步
         * 如果此时线程池还有线程（正在运行任务，正在等待任务）
         * 中断唤醒一个正在等任务的空闲worker
         * 唤醒后再次判断线程池状态，会return null，进入processWorkerExit()流程
         */
        if (workerCountOf(c) != 0) { // Eligible to terminate 资格终止
            interruptIdleWorkers(ONLY_ONE); //中断workers集合中的空闲任务，参数为true，只中断一个
            return;
        }

        /**
         * 如果状态是SHUTDOWN，workQueue也为空了，正在运行的worker也没有了，开始terminated
         */
```

```
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //CAS：将线程池的ctl变成TIDYING（所有的任务被终止，workCount为0，为此状态时将会调用
terminated()方法），期间ctl有变化就会失败，会再次for循环
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    terminated(); //需子类实现
                } 
                finally {
                    ctl.set(ctlOf(TERMINATED, 0)); //将线程池的ctl变成TERMINATED
                    termination.signalAll(); //唤醒调用了 等待线程池终止的线程 awaitTermination()
                }
                return;
            }
        }
        finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
        // 如果上面的CAS判断false，再次循环
    }
}
```

tryTerminate() 执行流程：

1、判断线程池是否需要进入终止流程（只有当shutdown状态+workQueue.isEmpty 或 stop状态，才需要）

2、判断线程池中是否还有线程，有则 interruptIdleWorkers(ONLY_ONE) 尝试中断一个空闲线程（正是这个逻辑可以再次发出中断信号，中断阻塞在获取任务的线程）

3、如果状态是SHUTDOWN，workQueue也为空了，正在运行的worker也没有了，开始terminated,会先上锁，将线程池置为tidying状态，之后调用需子类实现的 terminated()，最后线程池置为terminated状态，并唤醒所有等待线程池终止这个Condition的线程.

shutdownNow()

```
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock(); //上锁

    try {
        //判断调用者是否有权限shutdown线程池
        checkShutdownAccess();

        //CAS+循环设置线程池状态为stop
        advanceRunState(STOP);

        //中断所有线程，包括正在运行任务的
        interruptWorkers();

        tasks = drainQueue(); //将workQueue中的元素放入一个List并返回
```

```
    }
    finally {
        mainLock.unlock(); //解锁
    }

    //尝试终止线程池
    tryTerminate();

    return tasks; //返回workQueue中未执行的任务
}
```

shutdownNow() 和 shutdown()的大体流程相似，差别是：

1、将线程池更新为stop状态

2、调用 interruptWorkers() 中断所有线程，包括正在运行的线程

3、将workQueue中待处理的任务移到一个List中，并在方法最后返回，说明shutdownNow()后不会再处理workQueue中的任务

```
private void interruptWorkers() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers)
            w.interruptIfStarted();
    } finally {
        mainLock.unlock();
    }
}
```

interruptWorkers() 很简单，循环对所有worker调用 interruptIfStarted()，其中会判断worker的AQS state是否大于0，即worker是否已经开始运作，再调用Thread.interrupt()

需要注意的是，对于运行中的线程调用Thread.interrupt()并不能保证线程被终止，task.run()内部可能捕获了InterruptException，没有上抛，导致线程一直无法结束.

awaitTermination()

参数： timeout：超时时间 unit： timeout超时时间的单位 **返回：** true：线程池终止 false：超过timeout指定时间在发出一个shutdown请求后，在以下3种情况发生之前，awaitTermination()都会被阻塞 1、所有任务完成执行 2、到达超时时间 3、当前线程被中断

awaitTermination() 循环的判断线程池是否terminated终止 或 是否已经超过超时时间，然后通过termination这个Condition阻塞等待一段时间

termination.awaitNanos() 是通过 LockSupport.parkNanos(this, nanosTimeout)实现的阻塞等待

阻塞等待过程中发生以下具体情况会解除阻塞（对上面3种情况的解释）：

1、如果发生了 termination.signalAll()（内部实现是 LockSupport.unpark()）会唤醒阻塞等待，且由于ThreadPoolExecutor只有在 tryTerminated()尝试终止线程池成功，将线程池更新为terminated状态后才会signalAll()，故awaitTermination()再次判断状态会return true退出

2、如果达到了超时时间 termination.awaitNanos() 也会返回，此时nano==0，再次循环判断return false，等待线程池终止失败

3、如果当前线程被 Thread.interrupt()，termination.awaitNanos()会上抛InterruptException，awaitTermination()继续上抛给调用线程，会以异常的形式解除阻塞

故终止线程池并需要知道其是否终止可以用如下方式：

```
executorService.shutdown();
try{
    while(!executorService.awaitTermination(500, TimeUnit.MILLISECONDS)) {
        LOGGER.debug("Waiting for terminate");
    }
}
catch (InterruptedException e) {
    //中断处理
}
```

java并发容器

ConcurrentHashMap

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因：所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap的结构

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

ConcurrentHashMap的初始化

ConcurrentHashMap初始化方法是通过initialCapacity，loadFactor, concurrencyLevel几个参数来初始化segments数组，段偏移量segmentShift，段掩码segmentMask和每个segment里的HashEntry数组。

初始化segments数组：

```
if (concurrencyLevel > MAX_SEGMENTS)
   concurrencyLevel = MAX_SEGMENTS;
   int sshift = 0;
   int ssize = 1;
   while (ssize < concurrencyLevel) {
       ++sshift;
       ssize <<= 1;
   }
   segmentShift = 32 - sshift;
   segmentMask = ssize - 1;
   this.segments = Segment.newArray(ssize);
```

由上面的代码可知segments数组的长度ssize通过concurrencyLevel计算得出。为了能通过按位与的哈希算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个是大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14，15或16，ssize都会等于16，即容器里锁的个数也是16。注意concurrencyLevel的最大大小是65535，意味着segments数组的长度最大为65536，对应的二进制是16位。

初始化segmentShift和segmentMask:

这两个全局变量在定位segment时的哈希算法里需要使用，sshift等于ssize从1向左移位的次数，在默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。segmentShift用于定位参与hash运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的，后面的测试中我们可以看到这点。segmentMask是哈希运算的掩码，等于ssize减1，即15，掩码的二进制各个位的值都是1。因为ssize的最大长度是65536，所以segmentShift最大值是16，segmentMask最大值是65535，对应的二进制是16位，每个位都是1。

初始化每个Segment:

输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个segment的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个segment。

```
if (initialCapacity > MAXIMUM_CAPACITY)
          initialCapacity = MAXIMUM_CAPACITY;
      int c = initialCapacity / ssize;
      if (c * ssize < initialCapacity)
          ++c;
      int cap = 1;
      while (cap < c)
          cap <<= 1;
      for (int i = 0; i < this.segments.length; ++i)
          this.segments[i] = new Segment<K,V>(cap, loadFactor);
}
```

上面代码中的变量cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以cap不是1，就是2的N次方。segment的容量threshold = (int)cap*loadFactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过运算cap等于1，threshold等于零。

定位Segment

既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到Segment。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再哈希。

```
private static int hash(int h) {
h += (h << 15) ^ 0xffffcd7d; h ^= (h >>> 10);
h += (h << 3); h ^= (h >>> 6);
h += (h << 2) + (h << 14); return h ^ (h >>> 16);
}
```

再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的Segment上，从而提高容器的存取效率。

ConcurrentHashMap的get操作

Segment的get操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到segment，再通过哈希算法定位到元素，代码如下：

```
public V get(Object key) {
       int hash = hash(key.hashCode());
       return segmentFor(hash).get(key, hash);
   }
```

get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读，我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

```
transient volatile int count;
volatile V value;
```

在定位元素的代码里我们可以发现定位HashEntry和定位Segment的哈希算法虽然一样，都与数组的长度减去一相与，但是相与的值不一样，定位Segment使用的是元素的hashcode通过再哈希后得到的值的高位，而定位HashEntry直接使用的是再哈希后的值。其目的是避免两次哈希后的值一样，导致元素虽然在Segment里散列开了，但是却没有在HashEntry里散列开。

```
hash >>> segmentShift) & segmentMask//定位Segment所使用的hash算法
int index = hash & (tab.length - 1);// 定位HashEntry所使用的hash算法
```

ConcurrentHashMap的Put操作

由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

是否需要扩容。在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

ConcurrentHashMap的size操作

如果我们要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，我们是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是拿到之后可能累加前使用的count发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计size的时候把所有Segment的put，remove和clean方法全部锁住，但是这种做法显然非常低效。

因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put , remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

ConcurrentLinkedQueue

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。

ConcurrentLinkedQueue的结构

ConcurrentLinkedQueue由head节点和tair节点组成，每个节点（Node）由节点元素（item）和指向下一个节点的引用(next)组成，节点与节点之间就是通过这个next关联起来，从而组成一张链表结构的队列。默认情况下head节点存储的元素为空，tair节点等于head节点。

Copy-On-Write容器

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

CopyOnWriteArrayList的实现原理

```
public boolean add(T e) {
final ReentrantLock lock = this.lock;
lock.lock();
try {

    Object[] elements = getArray();
```

```
    int len = elements.length;
    // 复制出新数组

    Object[] newElements = Arrays.copyOf(elements, len + 1);
    // 把新元素添加到新数组里

    newElements[len] = e;
    // 把原数组引用指向新数组

    setArray(newElements);

    return true;

} finally {

    lock.unlock();

}
```

读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList。

```
public E get(int index) {
    return get(getArray(), index);
}
```

CopyOnWrite的缺点

内存占用问题：因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

数据一致性问题：CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

LinkedBlockingQueue

java.util.concurrent.LinkedBlockingQueue 是一个基于单向链表的、范围任意的（其实是有界的）、FIFO 阻塞队列。访问与移除操作是在队头进行，添加操作是在队尾进行，并分别使用不同的锁进行保护，只有在可能涉及多个节点的操作才同时对两个锁进行加锁。

队列是否为空、是否已满仍然是通过元素数量的计数器（count）进行判断的，由于可以同时在队头、队尾并发地进行访问、添加操作，所以这个计数器必须是线程安全的，这里使用了一个原子类 AtomicInteger，这就决定了它的容量范围是： 1 – Integer.MAX_VALUE。

由于同时使用了两把锁，在需要同时使用两把锁时，加锁顺序与释放顺序是非常重要的：必须以固定的顺序进行加锁，再以与加锁顺序的相反的顺序释放锁。

头结点和尾结点一开始总是指向一个哨兵的结点，它不持有实际数据，当队列中有数据时，头结点仍然指向这个哨兵，尾结点指向有效数据的最后一个结点。这样做的好处在于，与计数器 count 结合后，对队头、队尾的访问可以独立进行，而不需要判断头结点与尾结点的关系。

属性

```
/**
 * 节点类，用于存储数据
 */
static class Node<E> {
    E item;
    Node<E> next;

    Node(E x) { item = x; }
}

/** 阻塞队列的大小，默认为Integer.MAX_VALUE */
private final int capacity;

/** 当前阻塞队列中的元素个数 */
private final AtomicInteger count = new AtomicInteger();

/**
 * 阻塞队列的头结点
 */
transient Node<E> head;

/**
 * 阻塞队列的尾节点
 */
private transient Node<E> last;

/** 获取并移除元素时使用的锁，如take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();

/** notEmpty条件对象，当队列没有数据时用于挂起执行删除的线程 */
private final Condition notEmpty = takeLock.newCondition();

/** 添加元素时使用的锁如 put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();

/** notFull条件对象，当队列数据已满时用于挂起执行添加的线程 */
private final Condition notFull = putLock.newCondition();
```

从上面的属性我们知道，每个添加到LinkedBlockingQueue队列中的数据都将被封装成Node节点，添加的链表队列中，其中head和last分别指向队列的头结点和尾结点。与ArrayBlockingQueue不同的是，LinkedBlockingQueue内部分别使用了takeLock 和 putLock 对并发进行控制，也就是说，添加和删除操作并不是互斥操作，可以同时进行，这样也就可以大大提高吞吐量。

这里如果不指定队列的容量大小，也就是使用默认的Integer.MAX_VALUE，如果存在添加速度大于删除速度时候，有可能会内存溢出，这点在使用前希望慎重考虑。

另外，LinkedBlockingQueue对每一个lock锁都提供了一个Condition用来挂起和唤醒其他线程。

构造函数

```
public LinkedBlockingQueue() {
    // 默认大小为Integer.MAX_VALUE
    this(Integer.MAX_VALUE);
}

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

public LinkedBlockingQueue(Collection<? extends E> c) {
    this(Integer.MAX_VALUE);
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        int n = 0;
        for (E e : c) {
            if (e == null)
                throw new NullPointerException();
            if (n == capacity)
                throw new IllegalStateException("Queue full");
            enqueue(new Node<E>(e));
            ++n;
        }
        count.set(n);
    } finally {
        putLock.unlock();
    }
}
```

默认的构造函数和最后一个构造函数创建的队列大小都为Integer.MAX_VALUE，只有第二个构造函数用户可以指定队列的大小。第二个构造函数最后初始化了last和head节点，让它们都指向了一个元素为null的节点。最后一个构造函数使用了putLock来进行加锁，但是这里并不是为了多线程的竞争而加锁，只是为了放入的元素能立即对其他线程可见。

入队方法

LinkedBlockingQueue提供了多种入队操作的实现来满足不同情况下的需求，入队操作有如下几种：

- void put(E e);
- boolean offer(E e);
- boolean offer(E e, long timeout, TimeUnit unit)。

put(E e)：

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
```

```
    final AtomicInteger count = this.count;
    // 获取锁中断
    putLock.lockInterruptibly();
    try {
        //判断队列是否已满，如果已满阻塞等待
        while (count.get() == capacity) {
            notFull.await();
        }
        // 把node放入队列中
        enqueue(node);
        c = count.getAndIncrement();
        // 再次判断队列是否有可用空间，如果有唤醒下一个线程进行添加操作
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    // 如果队列中有一条数据，唤醒消费线程进行消费
    if (c == 0)
        signalNotEmpty();
}
```

enqueue(Node node):

```
private void enqueue(Node<E> node) {
    last = last.next = node;
}
```

signalNotEmpty():

```
private void signalNotEmpty() {
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
}

private void signalNotFull() {
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        notFull.signal();
    } finally {
        putLock.unlock();
    }
}
```

offer(E e):

```
public boolean offer(E e) {
    if (e == null) throw new NullPointerException();
    final AtomicInteger count = this.count;
    if (count.get() == capacity)
        return false;
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        // 队列有可用空间，放入node节点，判断放入元素后是否还有可用空间，
        // 如果有，唤醒下一个添加线程进行添加操作。
        if (count.get() < capacity) {
            enqueue(node);
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        }
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
    return c >= 0;
}
```

offer(E e, long timeout, TimeUnit unit)：

```
public boolean offer(E e, long timeout, TimeUnit unit)
        throws InterruptedException {

    if (e == null) throw new NullPointerException();
    long nanos = unit.toNanos(timeout);
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        // 等待超时时间nanos，超时时间到了返回false
        while (count.get() == capacity) {
            if (nanos <= 0)
                return false;
            nanos = notFull.awaitNanos(nanos);
        }
        enqueue(new Node<E>(e));
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally {
        putLock.unlock();
    }
    if (c == 0)
```

```
        signalNotEmpty();
    return true;
}
```

该方法只是对offer方法进行了阻塞超时处理，使用了Condition的awaitNanos来进行超时等待，这里为什么要用while循环？因为awaitNanos方法是可中断的，为了防止在等待过程中线程被中断，这里使用while循环进行等待过程中中断的处理，继续等待剩下需等待的时间。

出队方法

入队列的方法说完后，我们来说说出队列的方法。LinkedBlockingQueue提供了多种出队操作的实现来满足不同情况下的需求，如下：

- E take();
- E poll();
- E poll(long timeout, TimeUnit unit);

take()：

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        // 队列为空，阻塞等待
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        // 队列中还有元素，唤醒下一个消费线程进行消费
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    // 移除元素之前队列是满的，唤醒生产线程进行添加元素
    if (c == capacity)
        signalNotFull();
    return x;
}
```

dequeue()：

```
private E dequeue() {
    // 获取到head节点
    Node<E> h = head;
    // 获取到head节点指向的下一个节点
    Node<E> first = h.next;
    // head节点原来指向的节点的next指向自己，等待下次gc回收
```

```
    h.next = h; // help GC
    // head节点指向新的节点
    head = first;
    // 获取到新的head节点的item值
    E x = first.item;
    // 新head节点的item值设置为null
    first.item = null;
    return x;
}
```

poll():

```
public E poll() {
    final AtomicInteger count = this.count;
    if (count.get() == 0)
        return null;
    E x = null;
    int c = -1;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        if (count.get() > 0) {
            x = dequeue();
            c = count.getAndDecrement();
            if (c > 1)
                notEmpty.signal();
        }
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

获取元素方法

```
public E peek() {
    if (count.get() == 0)
        return null;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        Node<E> first = head.next;
        if (first == null)
            return null;
        else
            return first.item;
    } finally {
        takeLock.unlock();
    }
}
```

删除元素方法

```
public boolean remove(Object o) {
if (o == null) return false;
// 两个lock全部上锁
fullyLock();
try {
    // 从head开始遍历元素，直到最后一个元素
    for (Node<E> trail = head, p = trail.next;
         p != null;
         trail = p, p = p.next) {
        // 如果找到相等的元素，调用unlink方法删除元素
        if (o.equals(p.item)) {
            unlink(p, trail);
            return true;
        }
    }
    return false;
} finally {
    // 两个lock全部解锁
    fullyUnlock();
}
}

void fullyLock() {
putLock.lock();
takeLock.lock();
}

void fullyUnlock() {
takeLock.unlock();
putLock.unlock();
}
```

unlink():

```
void unlink(Node<E> p, Node<E> trail) {
// p的元素置为null
p.item = null;
// p的前一个节点的next指向p的next，也就是把p从链表中去除了
trail.next = p.next;
// 如果last指向p，删除p后让last指向trail
if (last == p)
    last = trail;
// 如果删除之前元素是满的，删除之后就有空间了，唤醒生产线程放入元素
if (count.getAndDecrement() == capacity)
    notFull.signal();
}
```

futureTask

简介

FutureTask是一种异步任务(或异步计算)，举个栗子，主线程的逻辑中需要使用某个值，但这个值需要负责的运算得来，那么主线程可以提前建立一个异步任务来计算这个值(在其他的线程中计算)，然后去做其他事情，当需要这个值的时候再通过刚才建立的异步任务来获取这个值，有点并行的意思，这样可以缩短整个主线程逻辑的执行时间。

框架

可以看到FutureTask实现了Runnable接口和Future接口，因此FutureTask可以传递到线程对象Thread或Excutor(线程池)来执行。

如果在当前线程中需要执行比较耗时的操作，但又不想阻塞当前线程时，可以把这些作业交给FutureTask，另开一个线程在后台完成，当当前线程将来需要时，就可以通过FutureTask对象获得后台作业的计算结果或者执行状态。

FutureTask():

```
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
}
```

futuretask内部结构：

```
public class FutureTask<V> implements RunnableFuture<V> {

    /**
     * 内部状态可能得迁转过程：
     * NEW -> COMPLETING -> NORMAL //正常完成
     * NEW -> COMPLETING -> EXCEPTIONAL //发生异常
     * NEW -> CANCELLED //取消
     * NEW -> INTERRUPTING -> INTERRUPTED //中断
     */
    private volatile int state;
    private static final int NEW          = 0;
    private static final int COMPLETING   = 1;
    private static final int NORMAL       = 2;
    private static final int EXCEPTIONAL  = 3;
    private static final int CANCELLED    = 4;
    private static final int INTERRUPTING = 5;
    private static final int INTERRUPTED  = 6;
    /** 内部的callable，运行完成后设置为null */
    private Callable<V> callable;
    /** 如果正常完成，就是执行结果，通过get方法获取；如果发生异常，就是具体的异常对象，通过get方法抛出。 */
    private Object outcome; // 本身没有volatile修饰，依赖state的读写来保证可见性。
```

```
    /** 执行内部callable的线程。 */
    private volatile Thread runner;
    /** 存放等待线程的Treiber Stack*/
    private volatile WaitNode waiters;
}
```

WaitNode：

```
static final class WaitNode {
    volatile Thread thread;
    volatile WaitNode next;
    WaitNode() { thread = Thread.currentThread(); }
}
```

FutureTask运行过程：

```
1. 创建任务，实际使用时，一般会结合线程池(ThreadPoolExecutor)使用，所以是在线程池内部创建FutureTask。
```

2. 执行任务，一般会有由工作线程(对于我们当前线程来说的其他线程)调用FutureTask的run方法，完成执行。
3. 获取结果，一般会有我们的当前线程去调用get方法来获取执行结果，如果获取时，任务并没有被执行完毕，当前线程就会被阻塞，直到任务被执行完毕，然后获取结果。
4. 取消任务，某些情况下会放弃任务的执行，进行任务取消。

源码

run()：

```
public void run() {
    // 如果state是NEW，设置线程为当前线程
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset, null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                // 调用Callable的call方法，得到结果
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                // 处理异常状态和结果
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                // 正常处理设置状态和结果
                set(result);
        }
```

```
    } finally {
        // runner必须在设置了state之后再置空，避免run方法出现并发问题。
        runner = null;
        // 这里还必须再读一次state，避免丢失中断。
        int s = state;
        if (s >= INTERRUPTING)
            // 处理可能发生的取消中断(cancel(true))。
            handlePossibleCancellationInterrupt(s);
    }
}
```

set():

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

set过程中，首先尝试将当前任务状态state从NEW改为COMPLETING。如果成功的话，再设置执行结果到outcome。然后将state再次设置为NORMAL，注意这次使用的是putOrderedInt，其实就是原子量的LazySet内部使用的方法。为什么使用这个方法？首先LazySet相对于Volatile-Write来说更廉价，因为它没有昂贵的Store/Load屏障，只有Store/Store屏障(x86下Store/Store屏障是一个空操作)，其次，后续线程不会及时的看到state从COMPLETING变为NORMAL，但这没什么关系，而且NORMAL是state的最终状态之一，以后不会在变化了。

finishCompletion():

```
private void finishCompletion() {
    for (WaitNode q; (q = waiters) != null;) {
        // 尝试将waiters设置为null。
        if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
            // 然后将waiters中的等待线程全部唤醒。
            for (;;) {
                Thread t = q.thread;
                if (t != null) {
                    q.thread = null;
                    LockSupport.unpark(t);    // 唤醒线程
                }
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }
    // 回调下钩子方法。
    done();
    // 置空callable，减少内存占用
```

```
    callable = null;
}
```

setException(Throwable t):

```
protected void setException(Throwable t) {
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = t;
        UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state
        finishCompletion();
    }
}
```

handlePossibleCancellationInterrupt(int s):

```
/**
 * 确保cancel(true)产生的中断发生在run或runAndReset方法过程中。
 */
private void handlePossibleCancellationInterrupt(int s) {
    // 如果当前正在中断过程中，自旋等待一下，等中断完成。
    if (s == INTERRUPTING)
        while (state == INTERRUPTING)
            Thread.yield(); // wait out pending interrupt
    // 这里的state状态一定是INTERRUPTED;
    // 这里不能清除中断标记，因为没办法区分来自cancel(true)的中断。
    // Thread.interrupted();
}
```

run方法总结:

- 只有state为NEW的时候才执行任务(调用内部callable的run方法)。执行前会原子的设置执行线程(runner)，防止竞争。
- 如果任务执行成功，设置执行结果，状态变更：NEW -> COMPLETING -> NORMAL。
- 如果任务执行发生异常，设置异常结果，状态变更：NEW -> COMPLETING -> EXCEPTIONAL。
- 将Treiber Stack中等待当前任务执行结果的等待节点中的线程全部唤醒，同时删除这些等待节点，将整个Treiber Stack置空。
- 最后别忘了等一下可能发生的cancel(true)中引起的中断，让这些中断发生在执行任务过程中(别泄露出去)。

runAndReset():

```
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;
}
```

该方法和run方法的区别是，run方法只能被运行一次任务，而该方法可以多次运行任务。而runAndReset这个方法不会设置任务的执行结果值,如果该任务成功执行完成后，不修改state的状态，还是可运行（NEW）状态，如果取消任务或出现异常，则不会再次执行。

get():

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);     // 如果任务还没执行完毕，等待任务执行完毕。
    return report(s);     // 如果任务执行完毕，获取执行结果。
}
```

awaitDone():

```
private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
    // 先算出到期时间。
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
        if (Thread.interrupted()) {
            // 如果当前线程被中断，移除等待节点q，然后抛出中断异常。
            removeWaiter(q);
            throw new InterruptedException();
        }

        int s = state;
        if (s > COMPLETING) {
            // 如果任务已经执行完毕
            if (q != null)
                q.thread = null;     // 如果q不为null，将q中的thread置空。
```

```
            return s;    // 返回任务状态。
        }
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();    // 如果当前正在完成过程中，出让CPU。
        else if (q == null)
            q = new WaitNode();    // 创建一个等待节点。
        else if (!queued)
            // 将q(包含当前线程的等待节点)入队。
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                    q.next = waiters, q);
        else if (timed) {
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) {
                //如果超时，移除等待节点q
                removeWaiter(q);
                //返回任务状态。
                return state;
            }
            //超时的话，就阻塞给定时间。
            LockSupport.parkNanos(this, nanos);
        }
        else
            //没设置超时的话，就阻塞当前线程。
            LockSupport.park(this);
    }
}
```

removeWaiter():

```
private void removeWaiter(WaitNode node) {
    if (node != null) {
        //将node的thread域置空。
        node.thread = null;
        //下面过程中会将node从等待队列中移除，以thread域为null为依据，
        //如果过程中发生了竞争，重试。
        retry:
        for (;;) {          // restart on removeWaiter race
            for (WaitNode pred = null, q = waiters, s; q != null; q = s) {
                s = q.next;
                if (q.thread != null)
                    pred = q;
                else if (pred != null) {
                    pred.next = s;
                    if (pred.thread == null) // check for race
                        continue retry;
                }
                else if (!UNSAFE.compareAndSwapObject(this, waitersOffset,
                        q, s))
                    continue retry;
            }
            break;
        }
```

```
    }
}
```

report(int s):

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

get方法总结:

```
- 首先检查当前任务的状态，如果状态表示执行完成，进入第2步。
```

- 获取执行结果，也可能得到取消或者执行异常，get过程结束。 如果当前任务状态表示未执行或者正在执行，那么当前线程放入一个新建的等待节点，然后进入Treiber Stack进行阻塞等待。
    - 如果任务被工作线程(对当前线程来说是其他线程)执行完毕，执行完毕时工作线程会唤醒Treiber Stack上等待的所有线程，所以当前线程被唤醒，清空当前等待节点上的线程域，然后进入第2步。
- 当前线程在阻塞等待结果过程中可能被中断，如果被中断，那么会移除当前线程在Treiber Stack上对应的等待节点，然后抛出中断异常，get过程结束。
- 当前线程也可能执行带有超时时间的阻塞等待，如果超时时间过了，还没得到执行结果，那么会除当前线程在Treiber Stack上对应的等待节点，然后抛出超时异常，get过程结束。

cancel(boolean):

```
public boolean cancel(boolean mayInterruptIfRunning) {
    if (!(state == NEW &&
          UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                  mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
        return false;
    try {
        // mayInterruptIfRunning并且有正在运行的线程，调用interrupt中断，最后设置状态为INTERRUPTED
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
            }
        }
    } finally {
        finishCompletion();
    }
    return true;
}
```

fork/join

Fork/Join框架

Fork/Join框架是Java7提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

工作窃取算法

假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

Fork/Join框架

第一步分割任务。首先我们需要有一个fork类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停的分割，直到分割出的子任务足够小。

第二步执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join使用两个类来完成以上两件事情：

- ForkJoinTask：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类：
  - RecursiveAction：用于没有返回结果的任务
  - RecursiveTask ： 用于有返回结果的任务。
- ForkJoinPool ： ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

源码实现

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

ForkJoinTask的fork方法实现原理。当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步的执行这个任务，然后立即返回结果。代码如下

```
public final ForkJoinTask fork() {
        ((ForkJoinWorkerThread) Thread.currentThread())
            .pushTask(this);
        return this;
}
```

pushTask方法把当前任务存放在ForkJoinTask 数组queue里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下：

```
final void pushTask(ForkJoinTask t) {
        ForkJoinTask[] q; int s, m;
        if ((q = queue) != null) {    // ignore if queue removed
            long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;
            UNSAFE.putOrderedObject(q, u, t);
            queueTop = s + 1;         // or use putOrderedInt
            if ((s -= queueBase) <= 2)
                pool.signalWork();
    else if (s == m)
                growQueue();
        }
    }
```

ForkJoinTask的join方法实现原理。Join方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看ForkJoinTask的join方法的实现，代码如下：

```
public final V join() {
        if (doJoin() != NORMAL)
            return reportResult();
        else
            return getRawResult();
}
private V reportResult() {
        int s; Throwable ex;
        if ((s = status) == CANCELLED)
            throw new CancellationException();
if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)
            UNSAFE.throwException(ex);
        return getRawResult();
}
```

首先，它调用了doJoin()方法，通过doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有四种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。

- 如果任务状态是已完成，则直接返回任务结果。
- 如果任务状态是被取消，则直接抛出CancellationException。
- 如果任务状态是抛出异常，则直接抛出对应的异常

```
private int doJoin() {
        Thread t; ForkJoinWorkerThread w; int s; boolean completed;
        if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
            if ((s = status) < 0)
 return s;
            if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {
                try {
                    completed = exec();
                } catch (Throwable rex) {
                    return setExceptionalCompletion(rex);
```

```
                }
                if (completed)
                    return setCompletion(NORMAL);
            }
            return w.joinTask(this);
        }
        else
            return externalAwaitDone();
    }
```

在doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为NORMAL，如果出现异常，则纪录异常，并将任务状态设置为EXCEPTIONAL。