# Unsafe

## 简介

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

## 如何使用

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

1. 私有的构造器
2. 工厂方法getUnsafe()的调用器只能被Bootloader加载，否则抛出SecurityException 异常

2、不过，我们可以通过反射机制轻松获取Unsafe的一个实例

1. **public** **static** Unsafe getUnsafe() {
2. **try** {
3. Field f = Unsafe.**class**.getDeclaredField("theUnsafe");
4. f.setAccessible(**true**);
5. **return** (Unsafe)f.get(**null**);
6. } **catch** (Exception e) {
7. **return** **null**;
8. }
9. }

## Offset方法

### objectFieldOffset

1. /\*\*\*
2. \* Returns the memory address offset of the given static field.
3. \* The offset is merely used as a means to access a particular field
4. \* in the other methods of this class.  The value is unique to the given
5. \* field and the same value should be returned on each subsequent call.
6. \* 返回指定静态field的内存地址偏移量,在这个类的其他方法中这个值只是被用作一个访问
7. \* 特定field的一个方式。这个值对于给定的field是唯一的，并且后续对该方法的调用都
8. \* 应该返回相同的值。
9. \*
10. \* @param field the field whose offset should be returned.
11. \*              需要返回偏移量的field
12. \* @return the offset of the given field.
13. \*          指定field的偏移量
14. \*/
15. **public** **native** **long** objectFieldOffset(Field field);

### arrayBaseOffset

1. /\*\*\*
2. \* Returns the offset of the first element for a given array class.
3. \* To access elements of the array class, this value may be used along with
4. \* with that returned by
5. \* <a href="#arrayIndexScale"><code>arrayIndexScale</code></a>,
6. \* if non-zero.
7. \* 获取给定数组中第一个元素的偏移地址。
8. \* 为了存取数组中的元素，这个偏移地址与<a href="#arrayIndexScale"><code>arrayIndexScale
9. \* </code></a>方法的非0返回值一起被使用。
10. \* @param arrayClass the class for which the first element's address should
11. \*                   be obtained.
12. \*       第一个元素地址被获取的class
13. \* @return the offset of the first element of the array class.
14. \*     数组第一个元素的偏移地址
15. \* @see arrayIndexScale(Class)
16. \*/
17. **public** **native** **int** arrayBaseOffset(Class arrayClass);

### arrayIndexScale

1. /\*\*\*
2. \* Returns the scale factor used for addressing elements of the supplied
3. \* array class.  Where a suitable scale factor can not be returned (e.g.
4. \* for primitive types), zero should be returned.  The returned value
5. \* can be used with
6. \* <a href="#arrayBaseOffset"><code>arrayBaseOffset</code></a>
7. \* to access elements of the class.
8. \* 获取用户给定数组寻址的换算因子.一个合适的换算因子不能返回的时候(例如：基本类型),
9. \* 返回0.这个返回值能够与<a href="#arrayBaseOffset"><code>arrayBaseOffset</code>
10. \* </a>一起使用去存取这个数组class中的元素
11. \*
12. \* @param arrayClass the class whose scale factor should be returned.
13. \* @return the scale factor, or zero if not supported for this array class.
14. \*/
15. **public** **native** **int** arrayIndexScale(Class arrayClass);

1、获得数组元素的大小，对于对象数组而言，其大小是一个固定的值(4字节或8字节)???

### staticFieldOffset

1、staticFieldOffset：能够获取到指定静态变量在类的静态地址中的偏移量

1. **public** native **long** staticFieldOffset(Field var1);

## CAS方法

### compareAndSwapInt

1. /\*\*\*
2. \* Compares the value of the integer field at the specified offset
3. \* in the supplied object with the given expected value, and updates
4. \* it if they match.  The operation of this method should be atomic,
5. \* thus providing an uninterruptible way of updating an integer field.
6. \* 在obj的offset位置比较integer field和期望的值，如果相同则更新。这个方法
7. \* 的操作应该是原子的，因此提供了一种不可中断的方式更新integer field。
8. \*
9. \* @param obj the object containing the field to modify.
10. \*            包含要修改field的对象
11. \* @param offset the offset of the integer field within <code>obj</code>.
12. \*            <code>obj</code>中整型field的偏移量
13. \* @param expect the expected value of the field.
14. \*            希望field中存在的值
15. \* @param update the new value of the field if it equals <code>expect</code>.
16. \*            如果期望值expect与field的当前值相同，设置filed的值为这个新值
17. \* @return true if the field was changed.
18. \*            如果field的值被更改则返回true
19. \*/
20. **public** **native** **boolean** compareAndSwapInt(Object obj, **long** offset,
21. **int** expect, **int** update);

### compareAndSwapLong

1. /\*\*\*
2. \* Compares the value of the long field at the specified offset
3. \* in the supplied object with the given expected value, and updates
4. \* it if they match.  The operation of this method should be atomic,
5. \* thus providing an uninterruptible way of updating a long field.
6. \* 在obj的offset位置比较long field和期望的值，如果相同则更新。这个方法
7. \* 的操作应该是原子的，因此提供了一种不可中断的方式更新long field。
8. \*
9. \* @param obj the object containing the field to modify.
10. \*               包含要修改field的对象
11. \* @param offset the offset of the long field within <code>obj</code>.
12. \*               <code>obj</code>中long型field的偏移量
13. \* @param expect the expected value of the field.
14. \*               希望field中存在的值
15. \* @param update the new value of the field if it equals <code>expect</code>.
16. \*               如果期望值expect与field的当前值相同，设置filed的值为这个新值
17. \* @return true if the field was changed.
18. \*               如果field的值被更改
19. \*/
20. **public** **native** **boolean** compareAndSwapLong(Object obj, **long** offset,
21. **long** expect, **long** update);

### compareAndSwapObject

1. /\*\*\*
2. \* Compares the value of the object field at the specified offset
3. \* in the supplied object with the given expected value, and updates
4. \* it if they match.  The operation of this method should be atomic,
5. \* thus providing an uninterruptible way of updating an object field.
6. \* 在obj的offset位置比较object field和期望的值，如果相同则更新。这个方法
7. \* 的操作应该是原子的，因此提供了一种不可中断的方式更新object field。
8. \*
9. \* @param obj the object containing the field to modify.
10. \*     包含要修改field的对象
11. \* @param offset the offset of the object field within <code>obj</code>.
12. \*          <code>obj</code>中object型field的偏移量
13. \* @param expect the expected value of the field.
14. \*               希望field中存在的值
15. \* @param update the new value of the field if it equals <code>expect</code>.
16. \*               如果期望值expect与field的当前值相同，设置filed的值为这个新值
17. \* @return true if the field was changed.
18. \*               如果field的值被更改
19. \*/
20. **public** **native** **boolean** compareAndSwapObject(Object obj, **long** offset,
21. Object expect, Object update);

### getObjectVolatile

1、getObjectVolatile保证读到最新的对象，即volatile的含义(可视性、有序性)

1. /\*\*\*
2. \* Retrieves the value of the object field at the specified offset in the
3. \* supplied object with volatile load semantics.
4. \* 获取obj对象中offset偏移地址对应的object型field的值,支持volatile load语义。
5. \*
6. \* @param obj the object containing the field to read.
7. \*    包含需要去读取的field的对象
8. \* @param offset the offset of the object field within <code>obj</code>.
9. \*       <code>obj</code>中object型field的偏移量
10. \*/
11. **public** **native** Object getObjectVolatile(Object obj, **long** offset);

## Ordered方法

### putOrderedInt

1. /\*\*\*
2. \* Sets the value of the integer field at the specified offset in the
3. \* supplied object to the given value.  This is an ordered or lazy
4. \* version of <code>putIntVolatile(Object,long,int)</code>, which
5. \* doesn't guarantee the immediate visibility of the change to other
6. \* threads.  It is only really useful where the integer field is
7. \* <code>volatile</code>, and is thus expected to change unexpectedly.
8. \* 设置obj对象中offset偏移地址对应的整型field的值为指定值。这是一个有序或者
9. \* 有延迟的<code>putIntVolatile</cdoe>方法，并且不保证值的改变被其他线程立
10. \* 即看到。只有在field被<code>volatile</code>修饰并且期望被意外修改的时候
11. \* 使用才有用。
12. \*
13. \* @param obj the object containing the field to modify.
14. \*     包含需要修改field的对象
15. \* @param offset the offset of the integer field within <code>obj</code>.
16. \*       <code>obj</code>中整型field的偏移量
17. \* @param value the new value of the field.
18. \*       field将被设置的新值
19. \* @see #putIntVolatile(Object,long,int)
20. \*/
21. **public** **native** **void** putOrderedInt(Object obj, **long** offset, **int** value);

### putOrderedLong

1. /\*\*\*
2. \* Sets the value of the long field at the specified offset in the
3. \* supplied object to the given value.  This is an ordered or lazy
4. \* version of <code>putLongVolatile(Object,long,long)</code>, which
5. \* doesn't guarantee the immediate visibility of the change to other
6. \* threads.  It is only really useful where the long field is
7. \* <code>volatile</code>, and is thus expected to change unexpectedly.
8. \* 设置obj对象中offset偏移地址对应的long型field的值为指定值。这是一个有序或者
9. \* 有延迟的<code>putLongVolatile</cdoe>方法，并且不保证值的改变被其他线程立
10. \* 即看到。只有在field被<code>volatile</code>修饰并且期望被意外修改的时候
11. \* 使用才有用。
12. \*
13. \* @param obj the object containing the field to modify.
14. \*     包含需要修改field的对象
15. \* @param offset the offset of the long field within <code>obj</code>.
16. \*       <code>obj</code>中long型field的偏移量
17. \* @param value the new value of the field.
18. \*       field将被设置的新值
19. \* @see #putLongVolatile(Object,long,long)
20. \*/
21. **public** **native** **void** putOrderedLong(Object obj, **long** offset, **long** value);

### putOrderedObject

1. /\*\*\*
2. \* Sets the value of the object field at the specified offset in the
3. \* supplied object to the given value.  This is an ordered or lazy
4. \* version of <code>putObjectVolatile(Object,long,Object)</code>, which
5. \* doesn't guarantee the immediate visibility of the change to other
6. \* threads.  It is only really useful where the object field is
7. \* <code>volatile</code>, and is thus expected to change unexpectedly.
8. \* 设置obj对象中offset偏移地址对应的object型field的值为指定值。这是一个有序或者
9. \* 有延迟的<code>putObjectVolatile</cdoe>方法，并且不保证值的改变被其他线程立
10. \* 即看到。只有在field被<code>volatile</code>修饰并且期望被意外修改的时候
11. \* 使用才有用。
12. \*
13. \* @param obj the object containing the field to modify.
14. \*     包含需要修改field的对象
15. \* @param offset the offset of the object field within <code>obj</code>.
16. \*       <code>obj</code>中long型field的偏移量
17. \* @param value the new value of the field.
18. \*       field将被设置的新值
19. \*/
20. **public** **native** **void** putOrderedObject(Object obj, **long** offset, Object value);

## Volatile方法

### putIntVolatile

1. /\*\*\*
2. \* Sets the value of the integer field at the specified offset in the
3. \* supplied object to the given value, with volatile store semantics.
4. \* 设置obj对象中offset偏移地址对应的整型field的值为指定值。支持volatile store
5. \* 语义
6. \*
7. \* @param obj the object containing the field to modify.
8. \*     包含需要修改field的对象
9. \* @param offset the offset of the integer field within <code>obj</code>.
10. \*       <code>obj</code>中整型field的偏移量
11. \* @param value the new value of the field.
12. \*       field将被设置的新值
13. \*/
14. **public** **native** **void** putIntVolatile(Object obj, **long** offset, **int** value);

### getIntVolatile

1. /\*\*\*
2. \* Retrieves the value of the integer field at the specified offset in the
3. \* supplied object with volatile load semantics.
4. \* 获取obj对象中offset偏移地址对应的整型field的值,支持volatile load语义。
5. \*
6. \* @param obj the object containing the field to read.
7. \*     包含需要去读取的field的对象
8. \* @param offset the offset of the integer field within <code>obj</code>.
9. \*       <code>obj</code>中整型field的偏移量
10. \*/
11. **public** **native** **int** getIntVolatile(Object obj, **long** offset);

### putLongVolatile

1. /\*\*\*
2. \* Sets the value of the long field at the specified offset in the
3. \* supplied object to the given value, with volatile store semantics.
4. \* 设置obj对象中offset偏移地址对应的long型field的值为指定值。支持volatile store语义
5. \*
6. \* @param obj the object containing the field to modify.
7. \*       包含需要修改field的对象
8. \* @param offset the offset of the long field within <code>obj</code>.
9. \*       <code>obj</code>中long型field的偏移量
10. \* @param value the new value of the field.
11. \*       field将被设置的新值
12. \* @see #putLong(Object,long,long)
13. \*/
14. **public** **native** **void** putLongVolatile(Object obj, **long** offset, **long** value);

### putLong

1. /\*\*\*
2. \* Sets the value of the long field at the specified offset in the
3. \* supplied object to the given value.
4. \* 设置obj对象中offset偏移地址对应的long型field的值为指定值。
5. \*
6. \* @param obj the object containing the field to modify.
7. \*      包含需要修改field的对象
8. \* @param offset the offset of the long field within <code>obj</code>.
9. \*      <code>obj</code>中long型field的偏移量
10. \* @param value the new value of the field.
11. \*      field将被设置的新值
12. \* @see #putLongVolatile(Object,long,long)
13. \*/
14. **public** **native** **void** putLong(Object obj, **long** offset, **long** value);

### getLongVolatile

1. /\*\*\*
2. \* Retrieves the value of the long field at the specified offset in the
3. \* supplied object with volatile load semantics.
4. \* 获取obj对象中offset偏移地址对应的long型field的值,支持volatile load语义。
5. \*
6. \* @param obj the object containing the field to read.
7. \*    包含需要去读取的field的对象
8. \* @param offset the offset of the long field within <code>obj</code>.
9. \*       <code>obj</code>中long型field的偏移量
10. \* @see #getLong(Object,long)
11. \*/
12. **public** **native** **long** getLongVolatile(Object obj, **long** offset);

### getLong

1. /\*\*\*
2. \* Retrieves the value of the long field at the specified offset in the
3. \* supplied object.
4. \* 获取obj对象中offset偏移地址对应的long型field的值
5. \*
6. \* @param obj the object containing the field to read.
7. \*     包含需要去读取的field的对象
8. \* @param offset the offset of the long field within <code>obj</code>.
9. \*       <code>obj</code>中long型field的偏移量
10. \* @see #getLongVolatile(Object,long)
11. \*/
12. **public** **native** **long** getLong(Object obj, **long** offset);

### putObjectVolatile

1. /\*\*\*
2. \* Sets the value of the object field at the specified offset in the
3. \* supplied object to the given value, with volatile store semantics.
4. \* 设置obj对象中offset偏移地址对应的object型field的值为指定值。支持volatile store语义
5. \*
6. \* @param obj the object containing the field to modify.
7. \*     包含需要修改field的对象
8. \* @param offset the offset of the object field within <code>obj</code>.
9. \*      <code>obj</code>中object型field的偏移量
10. \* @param value the new value of the field.
11. \*       field将被设置的新值
12. \* @see #putObject(Object,long,Object)
13. \*/
14. **public** **native** **void** putObjectVolatile(Object obj, **long** offset, Object value);

### putObject

1. /\*\*\*
2. \* Sets the value of the object field at the specified offset in the
3. \* supplied object to the given value.
4. \* 设置obj对象中offset偏移地址对应的object型field的值为指定值。
5. \*
6. \* @param obj the object containing the field to modify.
7. \*    包含需要修改field的对象
8. \* @param offset the offset of the object field within <code>obj</code>.
9. \*     <code>obj</code>中object型field的偏移量
10. \* @param value the new value of the field.
11. \*       field将被设置的新值
12. \* @see #putObjectVolatile(Object,long,Object)
13. \*/
14. **public** **native** **void** putObject(Object obj, **long** offset, Object value);

### getObjectVolatile

1. /\*\*\*
2. \* Retrieves the value of the object field at the specified offset in the
3. \* supplied object with volatile load semantics.
4. \* 获取obj对象中offset偏移地址对应的object型field的值,支持volatile load语义。
5. \*
6. \* @param obj the object containing the field to read.
7. \*    包含需要去读取的field的对象
8. \* @param offset the offset of the object field within <code>obj</code>.
9. \*       <code>obj</code>中object型field的偏移量
10. \*/
11. **public** **native** Object getObjectVolatile(Object obj, **long** offset);

## Park方法

### unpark

1. /\*\*\*
2. \* Releases the block on a thread created by
3. \* <a href="#park"><code>park</code></a>.  This method can also be used
4. \* to terminate a blockage caused by a prior call to <code>park</code>.
5. \* This operation is unsafe, as the thread must be guaranteed to be
6. \* live.  This is true of Java, but not native code.
7. \* 释放被<a href="#park"><code>park</code></a>创建的在一个线程上的阻塞.这个
8. \* 方法也可以被使用来终止一个先前调用<code>park</code>导致的阻塞.
9. \* 这个操作操作时不安全的,因此线程必须保证是活的.这是java代码不是native代码。
10. \* @param thread the thread to unblock.
11. \*           要解除阻塞的线程
12. \*/
13. **public** **native** **void** unpark(Thread thread);

### park

1. /\*\*\*
2. \* Blocks the thread until a matching
3. \* <a href="#unpark"><code>unpark</code></a> occurs, the thread is
4. \* interrupted or the optional timeout expires.  If an <code>unpark</code>
5. \* call has already occurred, this also counts.  A timeout value of zero
6. \* is defined as no timeout.  When <code>isAbsolute</code> is
7. \* <code>true</code>, the timeout is in milliseconds relative to the
8. \* epoch.  Otherwise, the value is the number of nanoseconds which must
9. \* occur before timeout.  This call may also return spuriously (i.e.
10. \* for no apparent reason).
11. \* 阻塞一个线程直到<a href="#unpark"><code>unpark</code></a>出现、线程
12. \* 被中断或者timeout时间到期。如果一个<code>unpark</code>调用已经出现了，
13. \* 这里只计数。timeout为0表示永不过期.当<code>isAbsolute</code>为true时，
14. \* timeout是相对于新纪元之后的毫秒。否则这个值就是超时前的纳秒数。这个方法执行时
15. \* 也可能不合理地返回(没有具体原因)
16. \*
17. \* @param isAbsolute true if the timeout is specified in milliseconds from
18. \*                   the epoch.
19. \*                   如果为true timeout的值是一个相对于新纪元之后的毫秒数
20. \* @param time either the number of nanoseconds to wait, or a time in
21. \*             milliseconds from the epoch to wait for.
22. \*             可以是一个要等待的纳秒数，或者是一个相对于新纪元之后的毫秒数直到
23. \*             到达这个时间点
24. \*/
25. **public** **native** **void** park(**boolean** isAbsolute, **long** time);

# Lock简介

## synchronized

1、Java中每一个对象都可以作为锁，这是synchronized实现同步的基础

1. 普通同步方法，锁是当前实例对象
2. 静态同步方法，锁是当前类的class对象
3. 同步方法块，锁是括号里面的对象

2、synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

* 如何保证内存可见性：同步代码块结束后从工作内存刷新到主内存中

3、同步方法：synchronized方法则会被翻译成普通的方法调用和返回指令如:invokevirtual、areturn指令，在VM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件的方法表中将该方法的access\_flags字段中的synchronized标志位置1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的Class在JVM的内部对象表示Klass做为锁对象

4、synchronized是重量级锁，重量级锁通过对象内部的监视器(monitor)实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高

5、当退出或者抛出异常时必须要释放锁，synchronized代码块能自动保证这一点

### HotSpotJVM底层实现

1、同步代码块是使用monitorenter和monitorexit指令实现的，同步方法依靠的是方法修饰符上的ACC\_SYNCHRONIZED实现

2、同步代码块：monitorenter指令插入到同步代码块的开始位置，monitorexit指令插入到同步代码块的结束位置，JVM需要保证每一个monitorenter都有一个monitorexit与之相对应(这就是保证在任何情况下退出synchronized代码块释放锁的原因)。任何对象都有一个monitor与之相关联，当且一个monitor被持有之后，他将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor所有权，即尝试获取对象的锁

### 对象头、monitor

#### 对象头

1、synchronized用的锁是存在Java对象头里的，那么什么是Java对象头呢？Hotspot虚拟机的对象头主要包括两部分数据：Mark Word(标记字段)、Klass Pointer(类型指针)。其中Klass Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键

2、Mark Word用于存储对象自身的运行时数据，如哈希码(HashCode)、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等。

3、Java对象头里的Mark Word里默认存储对象的HashCode，分代年龄和锁标记位。32位JVM的Mark Word的默认存储结构如下



3、在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化(约了节省空间)。Mark Word可能变化为存储以下5种数据



* 其中偏向锁和无锁状态的锁标志位相同，于是需要用是否偏向锁的标志位来判断是哪一种状态

#### monitor

1、什么是Monitor

* 我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象
* 与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁

2、Monitor是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表

3、每一个被锁住的对象都会和一个monitor关联(对象头的MarkWord中的LockWord指向monitor的起始地址)，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。其结构如下



1. Owner：初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL
2. EntryQ:关联一个系统互斥锁(semaphore)，阻塞所有试图锁住monitor record失败的线程
3. RcThis:表示blocked或waiting在该monitor record上的所有线程的个数
4. Nest:用来实现重入锁的计数
5. HashCode:保存从对象头拷贝过来的HashCode值(可能还包含GC age)
6. Candidate:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换(从阻塞到就绪然后因为竞争锁失败又被阻塞)从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁

## 锁优化

1、jdk1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销

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

### 自旋锁

1、线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁

2、所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。怎么等待呢？执行一段无意义的循环即可(自旋)

3、自旋等待不能替代阻塞，虽然它可以避免线程切换带来的开销，但是它占用了处理器的时间

1. 如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好
2. 反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，典型的占着茅坑不拉屎，这样反而会带来性能上的浪费

* 所以说，自旋等待的时间(自旋的次数)必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

4、自旋锁在JDK 1.4.2中引入，默认关闭，但是可以使用-XX:+UseSpinning开开启，在JDK1.6中默认开启。同时自旋的默认次数为10次，可以通过参数-XX:PreBlockSpin来调整；

* 如果通过参数-XX:preBlockSpin来调整自旋锁的自旋次数，会带来诸多不便。假如我将参数调整为10，但是系统很多线程都是等你刚刚退出的时候就释放了锁(假如你多自旋一两次就可以获取锁)，你是不是很尴尬
* 于是JDK1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明

#### 另一篇博客中的描述

http://www.cnblogs.com/wade-luffy/p/5969418.html

1、线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作。同时我们可以发现，很多对象锁的锁定状态只会持续很短的一段时间，例如整数的自加操作，在很短的时间内阻塞并唤醒线程显然不值得，为此引入了自旋锁

2、所谓"自旋"，就是让线程去执行一个无意义的循环，循环结束后再去重新竞争锁，如果竞争不到继续循环，循环过程中线程会一直处于running状态，但是基于JVM的线程调度，会出让时间片，所以其他线程依旧有申请锁和释放锁的机会

3、自旋锁省去了阻塞锁的时间空间(队列的维护等)开销，但是长时间自旋就变成了"忙式等待"，忙式等待显然还不如阻塞锁。所以自旋的次数一般控制在一个范围内，例如10,100等，在超出这个范围后，自旋锁会升级为阻塞锁

### 适应自旋锁

1、JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定

1. 线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多
2. 反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源

2、有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明

### 锁消除

1、为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持(逃逸分析的另一用处就是让对象在栈上而非堆中分配空间以提高效率)

2、如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于我们程序员来说这还不清楚么？我们会在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？我们虽然没有显示使用锁，但是我们在使用一些JDK的内置API时，如StringBuffer、Vector、HashTable等，这个时候会存在隐形的加锁操作

### 锁粗化

1、我们知道在使用同步锁的时候，需要让同步块的作用范围尽可能小—仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

2、在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗话的概念

3、锁粗话概念比较好理解，就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁

* 例如：vector每次add的时候都需要加锁操作，JVM检测到对同一个对象(vector)连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到for循环之外

### 轻量级锁

1、引入轻量级锁的主要目的是在多没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁，其步骤如下：

* 获取锁

1. 判断当前对象是否处于无锁状态(锁标志位01，偏向锁标志位0)

* 若是，则JVM首先将在当前线程的栈帧中建立一个名为锁记录(Lock Record)的空间，用于存储锁对象目前的Mark Word的拷贝(官方把这份拷贝加了一个Displaced前缀，即Displaced Mark Word)
* 否则执行步骤(3)

1. JVM利用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针

* 如果成功表示竞争到锁，则将锁标志位变成00(表示此对象处于轻量级锁状态)，执行同步操作
* 如果失败则执行步骤(3)

1. 判断当前对象的Mark Word是否指向当前线程的栈帧

* 如果是则表示当前线程已经持有当前对象的锁，则直接执行同步代码块
* 否则只能说明该锁对象已经被其他线程抢占了，这时轻量级锁需要膨胀为重量级锁，锁标志位变成10，后面等待的线程将会进入阻塞状态
* 释放锁(轻量级锁的释放也是通过CAS操作来进行)

1. 取出在获取轻量级锁保存在Displaced Mark Word中的数据
2. 用CAS操作将取出的数据替换当前对象的Mark Word中

* 如果成功，则说明释放锁成功
* 否则执行(3)

1. 如果CAS操作替换失败，说明有其他线程尝试获取该锁，则需要在释放锁的同时唤醒被挂起的线程

2、对于轻量级锁，其性能提升的依据是"对于绝大部分的锁，在整个生命周期内都是不会存在竞争的"，如果打破这个依据则除了互斥的开销外，还有额外的CAS操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢

3、下图是轻量级锁获取和释放过程



#### 另一篇博客中的描述

1、加锁：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋获取锁，当自旋获取锁仍然失败时，表示存在其他线程竞争锁(两条或两条以上的线程竞争同一个锁)，则轻量级锁会膨胀成重量级锁。

2、解锁：轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示同步过程已完成。如果失败，表示有其他线程尝试过获取该锁，则要在释放锁的同时唤醒被挂起的线程。

### 偏向锁

1、引入偏向锁主要目的是：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径(CAS原子指令)。

2、那么偏向锁是如何来减少不必要的CAS操作呢？我们可以查看Mark work的结构就明白了。只需要检查是否为偏向锁、锁标识为以及ThreadID即可，处理流程如下：

* 获取锁

1. 检测Mark Word是否为可偏向状态(锁标识位01，偏向锁标志位1)
2. 若为可偏向状态，则测试线程ID是否为当前线程ID

* 如果是，则执行步骤(5)
* 否则执行步骤(3)

1. 如果线程ID不为当前线程ID，则通过CAS操作竞争锁

* 竞争成功，将Mark Word的线程ID替换为当前线程ID，执行步骤(5)
* 否则执行步骤(4)

1. 通过CAS竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块
2. 执行同步代码块

* 释放锁：偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要等待全局安全点(这个时间点是上没有正在执行的代码)。其步骤如下

1. 暂停拥有偏向锁的线程，判断锁对象石是否还处于被锁定状态
2. 撤销偏向锁，恢复到无锁状态(01)或者轻量级锁的状态

2、下图是偏向锁的获取和释放流程



#### 另一篇博客中的描述

1、大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入(CAS)的开销，看起来让这个线程得到了偏护。另外，JVM对那种会有多线程加锁，但不存在锁竞争的情况也做了优化，听起来比较拗口，但在现实应用中确实是可能出现这种情况，因为线程之前除了互斥之外也可能发生同步关系，被同步的两个线程(一前一后)对共享对象锁的竞争很可能是没有冲突的。对这种情况，JVM用一个epoch表示一个偏向锁的时间戳(真实地生成一个时间戳代价还是蛮大的，因此这里应当理解为一种类似时间戳的identifier)

2、偏向锁的获取：当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要花费CAS操作来加锁和解锁，而只需简单的测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下Mark Word中偏向锁的标识是否设置成1(表示当前是偏向锁)，如果没有设置，则使用CAS竞争锁，如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程

3、偏向锁的撤销：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点(在这个时间点上没有字节码正在执行)，它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态，如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word，要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程

4、偏向锁的设置

* 关闭偏向锁：偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟-XX：BiasedLockingStartupDelay = 0。如果你确定自己应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁-XX:-UseBiasedLocking=false，那么默认会进入轻量级锁状态

### 重量级锁

1、重量锁在JVM中又叫对象监视器(Monitor)，它很像C中的Mutex，除了具备Mutex(0|1)互斥的功能，它还负责实现了Semaphore(信号量)的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列(wait队列)，前者负责做互斥，后一个用于做线程同步

2、重量级锁是使用操作系统互斥量来实现的

### 总结

1、偏向锁

* 优点：加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距
* 缺点：如果线程间存在锁竞争，会带来额外的锁撤销的消耗，适用于只有一个线程访问同步块场景

2、轻量级锁

* 竞争的线程不会阻塞，提高了程序的响应速度
* 缺点：如果始终得不到锁竞争的线程使用自旋会消耗CPU追求响应时间，锁占用时间很短

3、重量级锁

* 优点：线程竞争不使用自旋，不会消耗CPU
* 缺点：线程阻塞，响应时间缓慢

# AQS框架

## 基本概念

### CAS

1、CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做

2、在Java中CAS操作是通过sun.misc.Unsafe类来实现的，该类采用单例模式，通过getUnsafe()方法获取唯一实例，但是我们的代码也被禁止调用getUnsafe()

3、CAS的优势

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

4、CAS的劣势

1. **ABA问题**：因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A-B-A就会变成1A-2B-3A
2. **循环时间长开销大**：自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令(de-pipeline),使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突(memory order violation)而引起CPU流水线被清空(CPU pipeline flush)，从而提高CPU的执行效率
3. **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。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

5、锁的另一种分类

1. 独占锁：synchronized就是一种独占锁，它会导致所有需要此锁的线程挂起，等待锁的释放
2. 乐观锁：每次不加锁去完成操作，如果因为冲突失败就重试，直到成功

### 自旋锁

1、自旋锁是指当一个线程尝试获取某个锁时，如果该锁已被其他线程占用，就一直循环检测锁是否被释放，而不是进入线程挂起或睡眠状态

2、自旋锁适用于锁保护的临界区很小的情况，临界区很小的话，锁占用的时间就很短

3、优势

1. lock-free：不加锁(没有唤醒阻塞的系统开销)

4、劣势

1. CPU开销大
2. 无法响应中断
3. 不支持FIFO

### Ticket Lock

1、优势

1. FIFO
2. lock-free：不加锁(没有唤醒阻塞的系统开销)

2、劣势

1. 无法响应中断
2. CPU开销大
3. 多个公共线程在共享资源上自旋，开销较大

### CLH

1、全称为：Craig，Landin，and Hagersten (CLH)locks

2、优势

1. FIFO
2. lock-free：不加锁(没有唤醒阻塞的系统开销)
3. 仅仅在本地变量上自旋(多个线程不会读写同一个状态资源)

3、劣势

1. 无法响应中断
2. CPU开销较大

### LockSupport

#### suspend/resume、wait/notify

1、suspend()和resume()方法：

* 这两个方法隶属于Thread，是Thread的非静态方法
* 两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume()被调用，才能使得线程重新进入可执行状态
* 典型地，suspend()和resume()被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用resume()使其恢复
* suspend不能响应中断
* 但suspend()方法很容易引起死锁问题，已经不推荐使用了
* 如果一个目标线程t1对某一关键系统资源进行了加锁操作，然后在该加锁区块执行t1.suspend()，那么除非执行t1.resume()，否则其它线程都将无法访问该系统资源
* 如果另外一个线程t2想要占用资源，那么t2必须调用t1.resume()，如果t2调用t1.resume()之前需要获取该系统资源，那么造成死锁

2、wait()和notify()方法：

* 这两个方法隶属于Object，是Object的非静态方法
* 两个方法配套使用，wait()使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的notify()被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的notify()被调用
* **必须要在synchronized块内使用(保证调用这两个方法时，获取该对象的锁)**，但是不用编译器也不会阻止，运行时可能抛出IllegalMonitorStateException异常
* wait可以响应中断

3、初看起来它们与 suspend()和resume()方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于：suspend()和resume()方法，阻塞时都不会释放占用的锁(如果占用了的话)；而wait()和notify()方法这一对方法则相反

#### park与unpark

1、LockSupport类是Java6(JSR166-JUC)引入的一个类，提供了基本的线程同步原语。LockSupport实际上是调用了Unsafe类里的函数，归结到Unsafe里，只有两个函数

public native void unpark(Thread jthread);

public native void park(boolean isAbsolute, long time);

2、unpark函数为线程提供"许可(permit)"，线程调用park函数则等待"许可"

* 这个有点像信号量，但是这个"许可"是不能叠加的，"许可"是一次性的
* 比如线程B连续调用了三次unpark函数，当线程A调用park函数就使用掉这个"许可"，如果线程A再次调用park，则进入等待状态

3、park和unpark的灵活之处

* unpark函数可以先于park调用，这个正是它们的灵活之处
* 一个线程它有可能在别的线程unpark之前，或者之后，或者同时调用了park，那么因为park的特性，它可以不用担心自己的park的时序问题，否则，如果park必须要在unpark之前，那么给编程带来很大的麻烦！！
* 在Java5里是用wait/notify/notifyAll来同步的。wait/notify机制有个很蛋疼的地方是，比如线程B要用notify通知线程A，那么线程B要确保线程A已经在wait调用上等待了，否则线程A可能永远都在等待。编程的时候就会很蛋疼
* park/unpark模型真正解耦了线程之间的同步，线程之间不再需要一个Object或者其它变量来存储状态，不再需要关心对方的状态
* park可以响应中断，但不是通过抛出InterruptedException的方式来中断，中断后中断标志位是true

4、HotSpot里park/unpark的实现

## AQS简介

1、谈到并发，不得不谈ReentrantLock，而谈到ReentrantLock，不得不谈**AbstractQueuedSynchronizer(AQS)**

2、类如其名，抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch

### AQS：CLH的变体

1、AQS通过为Node增加predecessor字段，实现了CLH无法处理timeouts以及cancellation，如果一个节点的前继节点的状态为cancel，那么该节点可以跳过该前继节点继续往前寻找

2、AQS相对于CLH的一个改进是：提供了快速定位后继节点的方法

* 在CLH自旋锁中，一个节点状态的改变，在下一次自旋中将被该节点的后继节点侦测到，因此link是不需要的
* 在阻塞同步器中，一个节点必须明确地唤醒(unpark)其后继节点
* 注意：对于目前没有可用的技术可以实现**双向链表的lock-free原子插入**，因此next指针的维护仅仅是简单的赋值，并没有使用CAS，因此是非并发安全的(可能其他线程依赖某节点的next指针时，其尚未赋值，虽然在未来的某时刻，它会被正确地赋值)
* next字段仅仅提供了一种快速访问后继节点的**尝试：通过next无法获取到某节点的后继节点并不代表真的没有后继节点**

3、AQS相对于CLH的一个改进是：设计节点的状态，用于控制阻塞，而非通过自旋来阻塞当前线程

1. tryAcquire执行权限的控制

* 在AQS框架中，一个queued thread仅能通过执行tryAcquire方法且返回true来正常返回，tryAcquire通过其子类来实现
* 控制的含义在于：确保一个active的线程只有在处于队列头部的时候才允许执行tryAcquire方法，否则就被阻塞
* 并不需要为每个节点设置一个状态来表示其是否允许执行tryAcquire，在AQS中仅仅需要判断当前节点的前继是否为head来判断该节点是否被允许执行tryAcquire

1. CANCEL状态的控制

* 每个线程的CANCEL状态则需要被设计为Node节点的字段

1. 阻塞状态的控制

* 节点的状态用于避免park与unpark的无效调用，虽然park与unpark的调用效率与阻塞原语的效率相近
* 在调用park方法之前，线程需要设置"signal me"状态，然后在调用park之前再次检查同步和节点状态
* 释放锁的线程会清除状态
* 这也避免了要求一个releasing的线程取确定其后继直到其后继设定了signal状态

4、AQS与其他CLH locks的变体的主要区别是AQS依赖于垃圾回收机制来管理节点的存储

1. 依赖于垃圾回收仍然需要nulling那些指向这些废弃对象的引用

### 框架



1、它维护了一个volatile int state(代表共享资源)和一个FIFO线程等待队列(多线程争用资源被阻塞时会进入此队列)。这里volatile是核心关键词。state的访问方式有三种

1. getState()
2. setState()
3. compareAndSetState()

2、AQS定义两种资源共享方式：

1. Exclusive(独占，只有一个线程能执行，如ReentrantLock)
2. Share(共享，多个线程可同时执行，如Semaphore/CountDownLatch)

3、不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护(如获取资源失败入队/唤醒出队等)，AQS已经在顶层实现好了。**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，成功则返回true，失败则返回false

* 以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0(即释放锁)为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的(state会累加)，这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的
* 再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N(注意N要与线程个数一致)。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作
* 一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock

4、同步队列基本结构

* Node是构成同步队列的基础，AQS拥有首节点(head)和尾节点(tail)，没有成功获取资源的线程将会放入到队列的尾部
* 同步器中包含了两个节点类型的引用，一个指向头节点(head)，一个指向尾节点(tail),没有获取到锁的线程，加入到队列的过程必须保证线程安全，因此同步器提供了一个基于ＣＡＳ的设置尾节点的方法CompareAndSetTail(Node expect,Node update)，它需要传递当前线程认为的尾节点和当前节点，只有设置成功后，当前节点才能正式与之前的尾节点建立关联





5、同步队列遵循FIFO，首节点是获取锁成功的节点，首节点的线程在释放锁时，将会唤醒后继节点，而后继节点将会在获取到锁时，将自己设置位首节点，过程如下所示



6、设置首节点是由成功获取锁的线程来完成的，由于只有一个线程能够成功获取锁，因此设置首节点不需要CAS操作

### 伪代码

if (!tryAcquire(arg)){

node = create andenqueue new node;

pred = node's effective predecessor;

while (pred is not headnode || !tryAcquire(arg)) {

if (pred's signal bitisset)

park();

else

compareAndSet pred's signal bit to true;

pred = node's effective predecessor;

}

head = node;

}

if (tryRelease(arg) &&head node's signal bit is set) {

compareAndSet head'ssignal bit to false;

unpark head's successor,if oneexists

}

## 源码详解

### 静态内部类Node

1、Node是AbstractQueuedSynchronizer的一个静态内部类

2、字段

*表示节点正处在共享模式下等待的标记*

static final Node SHARED = new Node();

*表示节点正在以独占模式等待的标记*

static final Node EXCLUSIVE = null;

*waitStatus值，表示线程已取消*

static final int CANCELLED = 1;

*waitStatus值，后继节点将当前节点设为SIGNAL，意味着当前节点释放后，有义务唤醒后继节点*

static final int SIGNAL = -1;

*waitStatus值，表示线程正在等待条件*

static final int CONDITION = -2;

*waitStatus值，共享模式的头结点可能处于此状态，表示无条件往下传播，引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒*

static final int PROPAGATE = -3;

*状态字段，仅接受值：*

1. *SIGNAL:值为-1 ，后继节点的线程处于等待状态，而当前节点的线程如果释放了资源或者被取消，将会通知后继节点，使后继节点的线程得以运行*
2. *CANCELLED:值为1，由于在同步队列中等待的线程等待超时或者被中断，需要从同步队列中取消等待，节点进入该状态将不会变化*
3. *CONDITION: 值为-2，节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了singal方法后，该节点将会从等待队列中转移到同步队列中，加入到对资源的获取中*
4. *PROPAGATE: 值为-3，表示下一次共享模式资源获取将会无条件地传播下去???*
5. *INITIAL: 初始状态值为0，在此状态下，独占模式不会释放后继节点*

volatile int waitStatus;

*链接到前驱节点，当前节点/线程依赖它来检查waitStatus。在入同步队列时被设置，并且仅在移除同步队列时才归零(为了GC的目的)。被取消的线程永远不会成功获取，并且线程只取消自身，而不是任何其他节点*

volatile Node prev;

*链接到后续节点，当前节点/线程释放时释放。在入同步队列期间分配，在绕过取消的前驱节点时调整，并在出同步队列时取消(为了GC的目的)。enq操作不会分配前驱节点的next字段，直到附加之后，因此看到一个为null的next字段不一定意味着该节点在队列的末尾。但是，如果next字段显示为null,我们可以从尾部扫描prev，仔细检查。被取消的节点的next字段被设置为指向节点本身而不是null，以使isOnSyncQueue更方便操作。调用isOnSyncQueue时，如果节点(始终是放置在条件队列上的节点)正等待在同步队列上重新获取，则返回true*

volatile Node next;

volatile Thread thread;

Node nextWaiter;

* 状态为0代表刚初始化，其后面没有需要唤醒的节点，或者为已经释放资源的节点

### 重要字段

1、首尾节点

private transient volatile Node head;// head所指的标杆结点，就是当前获取到资源的那个结点或null(null即代表队列尚未初始化)

private transient volatile Node tail;//当队列初始化完毕后，该tail为队列中最后一个节点(并非尾后节点，是有效节点)，若tail==null则队列为空

2、状态

private volatile int state;

### acquire(int)

1、此方法是独占模式下线程获取共享资源的顶层入口。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。这也正是lock()的语义，当然不仅仅只限于lock()。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();

}

2、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以"死循环"(适量的自旋)的方式获取资源，**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(意思就是被吞了)。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上

3、独占式资源获取流程如下



#### tryAcquire(int)

1、此方法尝试去获取独占资源。如果获取成功，则直接返回true，否则直接返回false。这也正是tryLock()的语义，还是那句话，当然不仅仅只限于tryLock()。如下是tryAcquire()的源码：

protected boolean tryAcquire(int arg) {

throw new UnsupportedOperationException();

}

2、AQS只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现。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

#### addWaiter(Node)

1、此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点，如下是源码

private Node addWaiter(Node mode) {

*以给定模式构造结点。mode有EXCLUSIVE(独占)和SHARED(共享)*

*当前线程会被绑定到这个Node节点中去*

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

*尝试快速方式直接放到队尾，以下情况会失败：*

1. *当tail为null，此时可能该队列尚未初始化*
2. *当调用compareAndSetTail执行CAS时失败，即当代表当前线程的Node节点插入失败(有代表其他线程插入成功)*

Node pred = tail;

if (pred != null) {

*此时node还不知道是否可以入队(下一句CAS操作不知道是否能返回true)，但是修改node节点的prev指针是无害的!!!*

node.prev = pred;

if (compareAndSetTail(pred, node)) {

*这里修改了队列中原有节点的指针，如果在CAS之后，该句之前有其他线程访问队列，如果依赖节点的next指针，可能会出现问题*

pred.next = node;

return node;

}

}

*上一步失败则通过enq入队*

enq(node);

return node;

}

##### enq(Node)

1、此方法用于将node加入队尾。源码如下

private Node enq(final Node node) {

*不断循环，直到成功加入队尾*

for (;;) {

Node t = tail;

if (t == null) {

*此时队列是空的，通过CPU完成CAS执行队列初始化动作，此时head和tail指向同一个标记节点，队列为空(已初始化完毕，但是没有有效节点)*

*记这个new Node()节点为DummyNode(非静态字段都是0或null，该节点无关联线程，自然也不会主动释放)，这个DummyNode节点如何唤醒其后继节点???*

if (compareAndSetHead(new Node()))

*compareAndSetHead对head的期望值为null*

tail = head;

} else {

node.prev = t;

*利用CPU的CAS指令尝试放入尾部，该CAS的线程安全性保证同一时刻只有一个节点插入到尾部，也就是node.prev一定是正确的*

if (compareAndSetTail(t, node)) {

*如果此处其他线程正在执行unparkSuccessor，查找后继有效节点，此时是不能依赖于next指针的，因为此时尚未赋值，会导致从前往后无法遍历整个队列，因此必须从后往前遍历，因为prev是保证已经连上的*

t.next = node;

return t;

}

}

}

}

2、通过compareAndSetTail(Node expect,Node update)方法来确保节点能够被线程安全的添加到同步队列的尾部。在enq(final Node node)方法中，同步器通过"死循环"来保证节点的正确添加，在死循环中，只有通过CAS将节点设置为尾节点后，当前线程才能从该方法返回，否则，当前线程不断得尝试设置。**enq(final Node node)方法将并发添加节点的请求通过CAS变得串行化了**

#### acquireQueued(Node, int)

1、通过tryAcquire()和addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了。该线程下一步：进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了

2、节点进入到同步队列后，进入了一个自旋的过程，每个节点都在自省的观察，当条件满足，获取到了资源时，就可以从这个自旋中退出，否则继续自旋并且阻塞节点的线程

3、源码如下

final boolean acquireQueued(final Node node, int arg) {

*标记是否成功拿到资源*

boolean failed = true;

try {

*标记是否被中断过，因为如果产生中断会被吞，因此要将该参数传出，以供外层函数重新恢复中断*

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

*如果该节点的前继节点p是head，有以下三种情况*

1. *可能是p节点释放资源唤醒node节点*
2. *可能是node节点被中断了*
3. *p为DummyNode节点，该节点没有设置thread字段，此时执行tryAcquire(arg)仍然有可能失败，原因如下：*

* *为什么当前线程会执行到这里的原因就是因为当前线程执行tryAcquire()失败，即代表了有个线程t\_out正在占用资源，该线程t\_out从未进入过队列。如果此时还未释放资源，因此node节点再次尝试获取会失败*
* *在这种情况下，DummyNode通知其后继节点是通过t\_out执行release来触发的，详见2.6*

if (p == head && tryAcquire(arg)) {

* *如果获取成功，那么将head指向该节点，因此head指向的节点就是当前获取到资源的节点或者null*
* *setHead函数做三件事：*

1. *将head置为node*
2. *置空node.thread字段*
3. *置空node.prev字段*

* *这段代码只有获取了锁的线程才能够执行，因此不会引发线程安全问题*

setHead(node);

p.next = null;

*标记已经成功拿到资源*

failed = false;

return interrupted;

}

*如果node的前继节点并非head，或者node抢占资源失败，那么进入睡眠*

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

如果失败了，就将该节点的状态变为cancel

cancelAcquire(node);

}

}

4、只有当线程的前驱节点是头节点才能继续获取资源，原因如下

1. 头节点是成功获取到资源的节点，而头节点的线程释放了资源后，将会唤醒后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点
2. 维护同步队列的FIFO原则

##### shouldParkAfterFailedAcquire(Node, Node)

1、此方法主要用于检查状态，看看自己是否真的可以去休息了，因为在队列中的节点并非全部有效节点，那些放弃执行的节点仍然会处于队列中，等待这种放弃的节点自然是不可行的，并且这些放弃的节点是由该方法来清除的

2、源码如下

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

int ws = pred.waitStatus;

if (ws == Node.SIGNAL)

* *如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了*
* *一种极端情况：此时head正在执行，且状态为SIGNAL，在node节点park之前，head唤醒了node节点，即当前获取锁的线程先unpark后继线程，然后后继线程再park，这样也是可以的，这正是park/unpark的灵活之处*

return true;

if (ws > 0) {

* ***此时ws的状态只能是CANCEL***
* *如果前驱放弃了，那就一直往前找，直到找到第一个处于正常等待状态(SIGNAL或0)的节点，并排在它的后边*
* *注意，这里并没有重置这些异常节点的next与prev字段，因为可以保证的是，node节点，与第一个找到的正常节点并没有指向这些异常节点，这些异常节点经过GC的可达性分析将会被判为不可达(废弃节点相互引用，也会被GC回收)*

do {

node.prev = pred = pred.prev;//这里修改node的prev字段

*这里while不用去判断pred是否为空，因为至少head是正在执行的线程不会>0*

} while (pred.waitStatus > 0);

pred.next = node;

* *以上修改node节点的prev指针和pred的next指针的语句是并发安全的，因为其他线程对于队列的修改并不在这之中*

} else {

* *此时ws的状态可以是0,-2(CONDITION),-3(PROPAGATE)*
* *如果前驱正常，那就把前驱的状态设置成SIGNAL，告诉它拿完号后通知自己一下*
* *队列初始化插入的DummyNode节点，其waitState为0，是通过这里将其变为SIGNAL的*
* *waitStatus must be 0 or PROPAGATE ???*
* *这里尝试将pred的状态改为SINGNAL(可能此时pred节点状态会改为CANCELL，因此可能失败)*

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

*但是不立即park，返回false后通过acquiredQueue再次尝试获取锁，为什么需要返回false*

1. *前继节点状态为CANCEL，向前找到有效节点，删除那些无效节点*
2. *前继节点状态正常，但不为SIGNAL，设置前继节点状态SIGNAL可能会失败*

return false;

}

3、整个流程中

1. 如果前驱节点状态是SIGNAL，那么去休息(在休息之前，前节点可能会执行release，此时先unpark再park)
2. 如果前驱节点状态为CANCEL，找到有效的前继节点，并再次尝试拿号(适量自旋)
3. 如果前驱结点的状态不是SIGNAL，尝试(此时该节点可能被CANCELL，或者被释放变为状态0)将其状态改为SIGNAL，同时可以再尝试下看有没有机会轮到自己拿号(适量自旋)

##### parkAndCheckInterrupt()

1、如果线程找好安全休息点后，那就可以安心去休息了。**此方法就是让线程去休息，真正进入等待状态**

private final boolean parkAndCheckInterrupt() {

*调用park()使线程进入waiting状态*

LockSupport.park(this);

*如果被唤醒，查看自己是不是被中断的，并且重置中断标志位*

return Thread.interrupted();

}

2、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()

##### 小结

1、看了shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()，现在让我们再回到acquireQueued()，总结下该函数的具体流程：

1. 结点进入队尾后，检查状态，找到安全休息点
2. 调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己
3. 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1

#### 小结

1、再来总结下acquire的流程

1. 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回
2. 没成功，则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休息，有机会时(轮到自己，会被unpark())会去尝试获取资源。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。

2、再用流程图总结一下



### release(int)

1、此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()

public final boolean release(int arg) {

if (tryRelease(arg)) {

*h节点有以下两种情况*

1. *h节点此时关联的线程此时正在释放资源*
2. *h节点为DummyNode(此时调用release()的线程压根没有入队过)*

Node h = head;

*如果h为空或者h的状态为0，则不唤醒后继节点*

if (h != null && h.waitStatus != 0)

*无论h节点是当前线程关联的节点还是DummyNode，都会唤醒h之后的有效节点*

unparkSuccessor(h);

return true;

}

return false;

}

#### tryRelease(int)

1、此方法尝试去释放指定量的资源。下面是tryRelease()的源码

protected boolean tryRelease(int arg) {

throw new UnsupportedOperationException();

}

2、跟tryAcquire()一样，这个方法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)

1、此方法用于唤醒等待队列中下一个线程。下面是源码

private void unparkSuccessor(Node node) {

int ws = node.waitStatus;

if (ws < 0)

*由于该方法会唤醒其有效后继，因此该节点的SIGNAL状态可以被清空了(SIGNAL就是后继节点让该节点释放时取唤醒它)*

*可能失败原因???*

compareAndSetWaitStatus(node, ws, 0);

*s代表node的下一个有效节点*

*这里进行了一次尝试：直接定位当前节点的后继节点，但不一定成功，因为next指针是不可靠的*

1. *Node的next指针的赋值并没有使用CAS，可能在赋值前就会有访问*
2. *如果某节点的next指针非空，那么它一定指向的是正确的节点*
3. *next的不可依赖性仅仅指使用时机，而非指向*

Node s = node.next;

if (s == null || s.waitStatus > 0) {

s = null;

* *s为空可能是由于node的next节点尚未赋值，并非后继无节点了*
* *这里必须从后往前找，因为next指针是不可靠的，prev指针是可靠的*

for (Node t = tail; t != null && t != node; t = t.prev)

if (t.waitStatus <= 0)

s = t;

}

if (s != null)

*唤醒下一个有效节点*

LockSupport.unpark(s.thread);

}

2、这个函数并不复杂。一句话概括：**用unpark()唤醒等待队列中最前边的那个未放弃线程，这里我们也用s来表示吧**。此时，再和acquireQueued()联系起来，s被唤醒后，进入if (p == head && tryAcquire(arg))的判断(即使p!=head也没关系，它会再进入shouldParkAfterFailedAcquire()寻找一个安全点。这里既然s已经是等待队列中最前边的那个未放弃线程了，那么通过shouldParkAfterFailedAcquire()的调整，s也必然会跑到head的next结点，下一次自旋p==head就成立啦)，然后s把自己设置成head标杆结点，表示自己已经获取到资源了，acquire()也返回了！

#### 小结

1、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)

1、此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);

}

2、这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：

1. 负值代表获取失败
2. 0代表获取成功，但没有剩余资源
3. 正数表示获取成功，还有剩余资源，其他线程还可以去获取

3、所以这里acquireShared()的流程就是：

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，直到获取到资源为止才返回

4、共享式获取与独占式获取的最主要区别在于同一时刻能否有多个线程同时获取到资源。通过调用acquireShared(int arg)方法可以共享式得获取资源

#### doAcquireShared(int)

1、此方法用于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。下面是doAcquireShared()的源码

private void doAcquireShared(int arg) {

*加入队列尾部*

final Node node = addWaiter(Node.SHARED);

*标志是否成功*

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

*如果node是head的后继节点，因为head是拿到资源的线程(或DummyNode)，此时node被唤醒，很可能是head(或者t\_out)用完资源来唤醒自己的*

if (p == head) {

*尝试获取资源*

int r = tryAcquireShared(arg);

*获取成功*

if (r >= 0) {

*将head指向自己，还有剩余资源可以再唤醒之后的线程，如果node节点的后继节点依然成功获取资源，那么node节点将被直接移出队列*

setHeadAndPropagate(node, r);

p.next = null; // help GC

*如果等待过程中被打断过，此时将中断补上*

if (interrupted)

selfInterrupt();

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&  
parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

2、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。因为老大先唤醒老二，老二一看资源不够自己用继续park()，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

3、**只要进了队列，只要前一个不被唤醒，后续的节点就不会被唤醒**

##### setHeadAndPropagate(Node, int)

1、源码如下

private void setHeadAndPropagate(Node node, int propagate) {

Node h = head; // Record old head for check below

setHead(node);

*如果还有剩余量，继续唤醒下一个邻居线程*

if (propagate > 0 || h == null || h.waitStatus < 0) {

Node s = node.next;

if (s == null || s.isShared())

doReleaseShared();

}

}

2、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

#### 小结

1、acquireShared()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。

2、其实跟acquire()的流程大同小异，只不过多了个自己拿到资源后，还会去唤醒后继队友的操作(共享)

### releaseShared()

1、此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。下面是releaseShared()的源码

public final boolean releaseShared(int arg) {

if (tryReleaseShared(arg)) {

doReleaseShared();

return true;

}

return false;

}

2、此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。

3、跟独占模式下的release()相似，但有一点稍微需要注意：

1. 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，这主要是基于可重入的考量
2. 共享模式下的releaseShared()则没有这种要求，一是共享的实质--多线程可并发执行；二是共享模式基本也不会重入吧(至少我还没见过)，所以自定义同步器可以根据需要决定返回值

4、**该方法与独占式主要区别在于tryReleaseShared(int arg)方法必须确保资源线程安全释放。一般是通过循环和CAS来保证的。因为释放资源的操作会同时来自多个线程**

#### doReleaseShared()

1、源码如下

private void doReleaseShared() {

*该函数必须保证能够有效传递，即便有其他in-progress线程正在执行acquireShared或者releaseShared*

*如果当前状态为0，必须保证能够有效传递，因此设为PROPAGATE???*

for (;;) {

Node h = head;

if (h != null && h != tail) {

int ws = h.waitStatus;

if (ws == Node.SIGNAL) {

*可能失败的原因：有多个活动线(只有一个是队列头，其他在队列之外)程正在release，为了避免多次唤醒后继节点，于是采用CAS操作*

*为什么要在这里执行将节点状态从SIGNAL到0的转变，因为unparkSuccessor中会将节点从负状态转换为0状态，如果多个线程同时执行unparkSuccessor，将会导致多发park信号量，因此必须保证只有一个线程可以唤醒其后继节点*

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

continue;// loop to recheck cases

*无论活动线程是否在队列中，只能唤醒头节点的后继*

unparkSuccessor(h);

*如果唤醒成功，那么该后继将会取代当前头结点成为队列的头结点，从而会导致该函数最后判断返回false，继续循环传播*

*如果唤醒失败，那么后继又会重新将该头结点设为SIGNAL状态，并且头状态没有变化退出循环，此次传播以失败告终*

}

*只有这一处会将节点设为PROPAGATE状态，PROPAGATE是一个非持久状态*

*如果头结点的状态为0，可能情况如下：*

1. *上一次循环已经将头结点状态置为0，因此需要将其置为PROPAGATE，以表示正处于传播状态，避免再次进入该if语句执行CAS操作，节省开销*
2. 后面不可能没有节点，因为head==tail进不到这里

*为什么要将其从0变为PROPAGATE：假设上一次循环t1将head从SIGNAL，变为0，并唤醒了后继。此时如果紧跟着t2释放资源，也会执行该函数，会尝试将0变为PROPAGATE，如果成功，t2完成任务。此时如果t3释放资源，看到head状态时PROPAGATE，就不会执行任何CAS操作*

else if (ws == 0 &&  
!compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

continue; // loop on failed CAS

}

*如果头结点没有变化，说明唤醒的线程获取不了锁(需要的资源大于现有的资源)*

if (h == head) // loop if head changed

break;

}

}

2、这个方法就一个目的，就是把当前结点设置为SIGNAL或者PROPAGATE，如果当前结点不为空且不是尾结点，先判断当前结点的状态位是否为SIGNAL，如果是就设置为0，**因为共享模式下更多使用PROPAGATE来传播**，SIGNAL会被经过两步改为PROPAGATE:

1. compareAndSetWaitStatus(h, Node.SIGNAL, 0)
2. compareAndSetWaitStatus(h, 0, Node.PROPAGATE)

3、为什么要经过两步呢

1. 博客上给出的原因：原因在unparkSuccessor方法：

private void unparkSuccessor(Node node) {

int ws = node.waitStatus;

if (ws < 0)

compareAndSetWaitStatus(node, ws, 0);

...

}

* 如果直接从SIGNAL到PROPAGATE,那么到unparkSuccessor方法里面又被设置为0：SIGNAL--PROPAGATE---0----PROPAGATE
* 对头结点相当于多做了一次compareAndSet操作，其实性能也殊途同归啦

1. 我的理解

* 如果头结点为SIGNAL状态，那么将其设为0，并唤醒后继节点，执行doAcquireShared方法，如果获取失败，又会将其有效前继设为SIGNAL然后park
* 如果头结点为0状态，那么将其设为PROPAGATE，此时无需再次唤醒其后继节点，因为之前已经做过尝试了，避免无效的park/unpark开销

4、在acquireShared(int arg)方法中，同步器调用tryAcquireShared(int arg)方法尝试获取资源，其返回值为int类型，当返回值大于0时，表示能够获取资源。因此，在共享式获取的自旋过程中，成功获取资源并且退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。共享式释放资源状态是通过调用releaseShared(int arg)方法

# AbstractQueuedSynchronizer

## 继承体系

C:\Users\liuye\AppData\Local\Temp\1495589793(1).png

### AbstractOwnableSynchronizer

1. **public** **abstract** **class** AbstractOwnableSynchronizer
2. **implements** java.io.Serializable {
4. /\*\* Use serial ID even though all fields transient. \*/
5. **private** **static** **final** **long** serialVersionUID = 3737899427754241961L;
7. /\*\*
8. \* Empty constructor for use by subclasses.
9. \*/
10. **protected** AbstractOwnableSynchronizer() { }
12. /\*\*
13. \* The current owner of exclusive mode synchronization.
14. \*/
15. **private** **transient** Thread exclusiveOwnerThread;
17. /\*\*
18. \* Sets the thread that currently owns exclusive access.
19. \* A {@code null} argument indicates that no thread owns access.
20. \* This method does not otherwise impose any synchronization or
21. \* {@code volatile} field accesses.
22. \* @param thread the owner thread
23. \*/
24. **protected** **final** **void** setExclusiveOwnerThread(Thread thread) {
25. exclusiveOwnerThread = thread;
26. }
28. /\*\*
29. \* Returns the thread last set by {@code setExclusiveOwnerThread},
30. \* or {@code null} if never set.  This method does not otherwise
31. \* impose any synchronization or {@code volatile} field accesses.
32. \* @return the owner thread
33. \*/
34. **protected** **final** Thread getExclusiveOwnerThread() {
35. **return** exclusiveOwnerThread;
36. }
37. }

## 常量

### Node#SHARED

1. /\*\* Marker to indicate a node is waiting in shared mode \*/
2. **static** **final** Node SHARED = **new** Node();

1、标记一个节点处于共享模式下

### Node#EXCLUSIVE

1. /\*\* Marker to indicate a node is waiting in exclusive mode \*/
2. **static** **final** Node EXCLUSIVE = **null**;

1、标记一个节点处于独占模式下

### Node#CANCELLED

1. /\*\* waitStatus value to indicate thread has cancelled \*/
2. **static** **final** **int** CANCELLED =  1;

1、标记一个线程(节点)已被取消

### Node#SIGNAL

1. /\*\* waitStatus value to indicate successor's thread needs unparking \*/
2. **static** **final** **int** SIGNAL    = -1;

1、当该节点释放资源时，需要唤醒其后继节点

### Node#CONDITION

1. /\*\* waitStatus value to indicate thread is waiting on condition \*/
2. **static** **final** **int** CONDITION = -2;

1、标记一个节点在一个条件变量上阻塞

### Node#PROPAGATE

1. /\*\*
2. \* waitStatus value to indicate the next acquireShared should
3. \* unconditionally propagate
4. \*/
5. **static** **final** **int** PROPAGATE = -3;

1、标记下一个acquireShared应该无条件传播???

### spinForTimeoutThreshold

1. /\*\*
2. \* The number of nanoseconds for which it is faster to spin
3. \* rather than to use timed park. A rough estimate suffices
4. \* to improve responsiveness with very short timeouts.
5. \*/
6. **static** **final** **long** spinForTimeoutThreshold = 1000L;

1、对于超时等待的acquire方法而言，如果剩余等待事件小于1000ns，那么仅采用自旋而非阻塞来等待

## 字段

### head

1. /\*\*
2. \* Head of the wait queue, lazily initialized.  Except for
3. \* initialization, it is modified only via method setHead.  Note:
4. \* If head exists, its waitStatus is guaranteed not to be
5. \* CANCELLED.
6. \*/
7. **private** **transient** **volatile** Node head;

1、阻塞队列的队列头，延迟加载

2、除了队列初始化之外，其他情况下仅能通过setHead进行赋值

3、如果队列头存在，那么其waitStatus能保证不处于CANCELLED状态

### tail

1. /\*\*
2. \* Tail of the wait queue, lazily initialized.  Modified only via
3. \* method enq to add new wait node.
4. \*/
5. **private** **transient** **volatile** Node tail;

1、阻塞队列尾，延迟加载

2、仅能通过enq来添加新节点

### state

1. /\*\*
2. \* The synchronization state.
3. \*/
4. **private** **volatile** **int** state;

1、资源数量，注意不要与Node#waitStatus字段搞混了

### unsafe字段

1. /\*\*
2. \* Setup to support compareAndSet. We need to natively implement
3. \* this here: For the sake of permitting future enhancements, we
4. \* cannot explicitly subclass AtomicInteger, which would be
5. \* efficient and useful otherwise. So, as the lesser of evils, we
6. \* natively implement using hotspot intrinsics API. And while we
7. \* are at it, we do the same for other CASable fields (which could
8. \* otherwise be done with atomic field updaters).
9. \*/
10. **private** **static** **final** Unsafe unsafe = Unsafe.getUnsafe();
11. **private** **static** **final** **long** stateOffset;
12. **private** **static** **final** **long** headOffset;
13. **private** **static** **final** **long** tailOffset;
14. **private** **static** **final** **long** waitStatusOffset;
15. **private** **static** **final** **long** nextOffset;
17. **static** {
18. **try** {
19. stateOffset = unsafe.objectFieldOffset
20. (AbstractQueuedSynchronizer.**class**.getDeclaredField("state"));
21. headOffset = unsafe.objectFieldOffset
22. (AbstractQueuedSynchronizer.**class**.getDeclaredField("head"));
23. tailOffset = unsafe.objectFieldOffset
24. (AbstractQueuedSynchronizer.**class**.getDeclaredField("tail"));
25. waitStatusOffset = unsafe.objectFieldOffset
26. (Node.**class**.getDeclaredField("waitStatus"));
27. nextOffset = unsafe.objectFieldOffset
28. (Node.**class**.getDeclaredField("next"));
30. } **catch** (Exception ex) { **throw** **new** Error(ex); }
31. }

## 内部类

### Node

1. /\*\*
2. \* Wait queue node class.
3. \*
4. \* <p>The wait queue is a variant of a "CLH" (Craig, Landin, and
5. \* Hagersten) lock queue. CLH locks are normally used for
6. \* spinlocks.  We instead use them for blocking synchronizers, but
7. \* use the same basic tactic of holding some of the control
8. \* information about a thread in the predecessor of its node.  A
9. \* "status" field in each node keeps track of whether a thread
10. \* should block.  A node is signalled when its predecessor
11. \* releases.  Each node of the queue otherwise serves as a
12. \* specific-notification-style monitor holding a single waiting
13. \* thread. The status field does NOT control whether threads are
14. \* granted locks etc though.  A thread may try to acquire if it is
15. \* first in the queue. But being first does not guarantee success;
16. \* it only gives the right to contend.  So the currently released
17. \* contender thread may need to rewait.
18. \*
19. \* <p>To enqueue into a CLH lock, you atomically splice it in as new
20. \* tail. To dequeue, you just set the head field.
21. \* <pre>
22. \*      +------+  prev +-----+       +-----+
23. \* head |      | <---- |     | <---- |     |  tail
24. \*      +------+       +-----+       +-----+
25. \* </pre>
26. \*
27. \* <p>Insertion into a CLH queue requires only a single atomic
28. \* operation on "tail", so there is a simple atomic point of
29. \* demarcation from unqueued to queued. Similarly, dequeuing
30. \* involves only updating the "head". However, it takes a bit
31. \* more work for nodes to determine who their successors are,
32. \* in part to deal with possible cancellation due to timeouts
33. \* and interrupts.
34. \*
35. \* <p>The "prev" links (not used in original CLH locks), are mainly
36. \* needed to handle cancellation. If a node is cancelled, its
37. \* successor is (normally) relinked to a non-cancelled
38. \* predecessor. For explanation of similar mechanics in the case
39. \* of spin locks, see the papers by Scott and Scherer at
40. \* http://www.cs.rochester.edu/u/scott/synchronization/
41. \*
42. \* <p>We also use "next" links to implement blocking mechanics.
43. \* The thread id for each node is kept in its own node, so a
44. \* predecessor signals the next node to wake up by traversing
45. \* next link to determine which thread it is.  Determination of
46. \* successor must avoid races with newly queued nodes to set
47. \* the "next" fields of their predecessors.  This is solved
48. \* when necessary by checking backwards from the atomically
49. \* updated "tail" when a node's successor appears to be null.
50. \* (Or, said differently, the next-links are an optimization
51. \* so that we don't usually need a backward scan.)
52. \*
53. \* <p>Cancellation introduces some conservatism to the basic
54. \* algorithms.  Since we must poll for cancellation of other
55. \* nodes, we can miss noticing whether a cancelled node is
56. \* ahead or behind us. This is dealt with by always unparking
57. \* successors upon cancellation, allowing them to stabilize on
58. \* a new predecessor, unless we can identify an uncancelled
59. \* predecessor who will carry this responsibility.
60. \*
61. \* <p>CLH queues need a dummy header node to get started. But
62. \* we don't create them on construction, because it would be wasted
63. \* effort if there is never contention. Instead, the node
64. \* is constructed and head and tail pointers are set upon first
65. \* contention.
66. \*
67. \* <p>Threads waiting on Conditions use the same nodes, but
68. \* use an additional link. Conditions only need to link nodes
69. \* in simple (non-concurrent) linked queues because they are
70. \* only accessed when exclusively held.  Upon await, a node is
71. \* inserted into a condition queue.  Upon signal, the node is
72. \* transferred to the main queue.  A special value of status
73. \* field is used to mark which queue a node is on.
74. \*
75. \* <p>Thanks go to Dave Dice, Mark Moir, Victor Luchangco, Bill
76. \* Scherer and Michael Scott, along with members of JSR-166
77. \* expert group, for helpful ideas, discussions, and critiques
78. \* on the design of this class.
79. \*/
80. **static** **final** **class** Node {
81. /\*\* Marker to indicate a node is waiting in shared mode \*/
82. **static** **final** Node SHARED = **new** Node();
83. /\*\* Marker to indicate a node is waiting in exclusive mode \*/
84. **static** **final** Node EXCLUSIVE = **null**;
86. /\*\* waitStatus value to indicate thread has cancelled \*/
87. **static** **final** **int** CANCELLED =  1;
88. /\*\* waitStatus value to indicate successor's thread needs unparking \*/
89. **static** **final** **int** SIGNAL    = -1;
90. /\*\* waitStatus value to indicate thread is waiting on condition \*/
91. **static** **final** **int** CONDITION = -2;
92. /\*\*
93. \* waitStatus value to indicate the next acquireShared should
94. \* unconditionally propagate
95. \*/
96. **static** **final** **int** PROPAGATE = -3;
98. /\*\*
99. \* Status field, taking on only the values:
100. \*   SIGNAL:     The successor of this node is (or will soon be)
101. \*               blocked (via park), so the current node must
102. \*               unpark its successor when it releases or
103. \*               cancels. To avoid races, acquire methods must
104. \*               first indicate they need a signal,
105. \*               then retry the atomic acquire, and then,
106. \*               on failure, block.
107. \*   CANCELLED:  This node is cancelled due to timeout or interrupt.
108. \*               Nodes never leave this state. In particular,
109. \*               a thread with cancelled node never again blocks.
110. \*   CONDITION:  This node is currently on a condition queue.
111. \*               It will not be used as a sync queue node
112. \*               until transferred, at which time the status
113. \*               will be set to 0. (Use of this value here has
114. \*               nothing to do with the other uses of the
115. \*               field, but simplifies mechanics.)
116. \*   PROPAGATE:  A releaseShared should be propagated to other
117. \*               nodes. This is set (for head node only) in
118. \*               doReleaseShared to ensure propagation
119. \*               continues, even if other operations have
120. \*               since intervened.
121. \*   0:          None of the above
122. \*
123. \* The values are arranged numerically to simplify use.
124. \* Non-negative values mean that a node doesn't need to
125. \* signal. So, most code doesn't need to check for particular
126. \* values, just for sign.
127. \*
128. \* The field is initialized to 0 for normal sync nodes, and
129. \* CONDITION for condition nodes.  It is modified using CAS
130. \* (or when possible, unconditional volatile writes).
131. \*/
132. **volatile** **int** waitStatus;
134. /\*\*
135. \* Link to predecessor node that current node/thread relies on
136. \* for checking waitStatus. Assigned during enqueuing, and nulled
137. \* out (for sake of GC) only upon dequeuing.  Also, upon
138. \* cancellation of a predecessor, we short-circuit while
139. \* finding a non-cancelled one, which will always exist
140. \* because the head node is never cancelled: A node becomes
141. \* head only as a result of successful acquire. A
142. \* cancelled thread never succeeds in acquiring, and a thread only
143. \* cancels itself, not any other node.
144. \*/
145. **volatile** Node prev;
147. /\*\*
148. \* Link to the successor node that the current node/thread
149. \* unparks upon release. Assigned during enqueuing, adjusted
150. \* when bypassing cancelled predecessors, and nulled out (for
151. \* sake of GC) when dequeued.  The enq operation does not
152. \* assign next field of a predecessor until after attachment,
153. \* so seeing a null next field does not necessarily mean that
154. \* node is at end of queue. However, if a next field appears
155. \* to be null, we can scan prev's from the tail to
156. \* double-check.  The next field of cancelled nodes is set to
157. \* point to the node itself instead of null, to make life
158. \* easier for isOnSyncQueue.
159. \*/
160. **volatile** Node next;
162. /\*\*
163. \* The thread that enqueued this node.  Initialized on
164. \* construction and nulled out after use.
165. \*/
166. **volatile** Thread thread;
168. /\*\*
169. \* Link to next node waiting on condition, or the special
170. \* value SHARED.  Because condition queues are accessed only
171. \* when holding in exclusive mode, we just need a simple
172. \* linked queue to hold nodes while they are waiting on
173. \* conditions. They are then transferred to the queue to
174. \* re-acquire. And because conditions can only be exclusive,
175. \* we save a field by using special value to indicate shared
176. \* mode.
177. \*/
178. Node nextWaiter;
180. /\*\*
181. \* Returns true if node is waiting in shared mode.
182. \*/
183. **final** **boolean** isShared() {
184. **return** nextWaiter == SHARED;
185. }
187. /\*\*
188. \* Returns previous node, or throws NullPointerException if null.
189. \* Use when predecessor cannot be null.  The null check could
190. \* be elided, but is present to help the VM.
191. \*
192. \* @return the predecessor of this node
193. \*/
194. **final** Node predecessor() **throws** NullPointerException {
195. Node p = prev;
196. **if** (p == **null**)
197. **throw** **new** NullPointerException();
198. **else**
199. **return** p;
200. }
202. Node() {    // Used to establish initial head or SHARED marker
203. }
205. Node(Thread thread, Node mode) {     // Used by addWaiter
206. **this**.nextWaiter = mode;
207. **this**.thread = thread;
208. }
210. Node(Thread thread, **int** waitStatus) { // Used by Condition
211. **this**.waitStatus = waitStatus;
212. **this**.thread = thread;
213. }
214. }

## Utils方法

## 方法

### acquire

1. /\*\*
2. \* Acquires in exclusive mode, ignoring interrupts.  Implemented
3. \* by invoking at least once {@link #tryAcquire},
4. \* returning on success.  Otherwise the thread is queued, possibly
5. \* repeatedly blocking and unblocking, invoking {@link
6. \* #tryAcquire} until success.  This method can be used
7. \* to implement method {@link Lock#lock}.
8. \*
9. \* @param arg the acquire argument.  This value is conveyed to
10. \*        {@link #tryAcquire} but is otherwise uninterpreted and
11. \*        can represent anything you like.
12. \*/
13. **public** **final** **void** acquire(**int** arg) {
14. **if** (!tryAcquire(arg) &&
15. acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
16. selfInterrupt();
17. }

#### tryAcquire

1. /\*\*
2. \* Attempts to acquire in exclusive mode. This method should query
3. \* if the state of the object permits it to be acquired in the
4. \* exclusive mode, and if so to acquire it.
5. \*
6. \* <p>This method is always invoked by the thread performing
7. \* acquire.  If this method reports failure, the acquire method
8. \* may queue the thread, if it is not already queued, until it is
9. \* signalled by a release from some other thread. This can be used
10. \* to implement method {@link Lock#tryLock()}.
11. \*
12. \* <p>The default
13. \* implementation throws {@link UnsupportedOperationException}.
14. \*
15. \* @param arg the acquire argument. This value is always the one
16. \*        passed to an acquire method, or is the value saved on entry
17. \*        to a condition wait.  The value is otherwise uninterpreted
18. \*        and can represent anything you like.
19. \* @return {@code true} if successful. Upon success, this object has
20. \*         been acquired.
21. \* @throws IllegalMonitorStateException if acquiring would place this
22. \*         synchronizer in an illegal state. This exception must be
23. \*         thrown in a consistent fashion for synchronization to work
24. \*         correctly.
25. \* @throws UnsupportedOperationException if exclusive mode is not supported
26. \*/
27. **protected** **boolean** tryAcquire(**int** arg) {
28. **throw** **new** UnsupportedOperationException();
29. }

1、交给子类去实现其同步语意

#### addWaiter

1. /\*\*
2. \* Creates and enqueues node for current thread and given mode.
3. \*
4. \* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
5. \* @return the new node
6. \*/
7. **private** Node addWaiter(Node mode) {
8. //将线程封装成一个Node，其状态为0
9. Node node = **new** Node(Thread.currentThread(), mode);
10. // Try the fast path of enq; backup to full enq on failure
11. // 下面几行先尝试进行入队动作，如果成功了，直接返回即可，若失败了，通过enq入队
12. Node pred = tail;
13. **if** (pred != **null**) {
14. node.prev = pred;
15. **if** (compareAndSetTail(pred, node)) {
16. pred.next = node;
17. **return** node;
18. }
19. }
20. enq(node);
21. **return** node;
22. }

#### enq

1. /\*\*
2. \* Inserts node into queue, initializing if necessary. See picture above.
3. \* @param node the node to insert
4. \* @return node's predecessor
5. \*/
6. **private** Node enq(**final** Node node) {
7. //CAS操作经典做法：死循环+CAS==>串行化入队操作
8. **for** (;;) {
9. Node t = tail;
10. **if** (t == **null**) { // Must initialize
11. //所谓第一个节点的延迟加载，该节点不关联任何线程，当然它也不会唤醒其后继节点，但是会由外界的线程来唤醒
12. **if** (compareAndSetHead(**new** Node()))
13. tail = head;
14. } **else** {
15. node.prev = t;
16. //CAS操作入队尾
17. **if** (compareAndSetTail(t, node)) {
18. //设置next字段的值，该值若为null，并不一定代表节点没有后继
19. t.next = node;
20. //该函数唯一出口
21. **return** t;
22. }
23. }
24. }
25. }

#### acquireQueued

1. /\*\*
2. \* Acquires in exclusive uninterruptible mode for thread already in
3. \* queue. Used by condition wait methods as well as acquire.
4. \*
5. \* @param node the node
6. \* @param arg the acquire argument
7. \* @return {@code true} if interrupted while waiting
8. \*/
9. **final** **boolean** acquireQueued(**final** Node node, **int** arg) {
10. **boolean** failed = **true**;
11. **try** {
12. //由于此过程中，线程的中断信号会被吃掉，因此当函数返回时需要还原中断状态
13. **boolean** interrupted = **false**;
14. **for** (;;) {
15. **final** Node p = node.predecessor();
16. //当前节点的前继节点时队头时，当前节点才有资格尝试获取资源
17. **if** (p == head && tryAcquire(arg)) {
18. //以下几行代码是线程安全的(共享模式下???)，因为只有当前线程能够获取到资源
19. setHead(node);
20. p.next = **null**; // help GC
21. failed = **false**;
22. //该函数唯一出口
23. **return** interrupted;
24. }
25. //将前继节点标记为SIGNAL后，阻塞自己
26. **if** (shouldParkAfterFailedAcquire(p, node) &&
27. parkAndCheckInterrupt())
28. interrupted = **true**;
29. }
30. } **finally** {
31. **if** (failed)
32. cancelAcquire(node);
33. }
34. }

#### shouldParkAfterFailedAcquire

1. /\*\*
2. \* Checks and updates status for a node that failed to acquire.
3. \* Returns true if thread should block. This is the main signal
4. \* control in all acquire loops.  Requires that pred == node.prev.
5. \*
6. \* @param pred node's predecessor holding status
7. \* @param node the node
8. \* @return {@code true} if thread should block
9. \*/
10. **private** **static** **boolean** shouldParkAfterFailedAcquire(Node pred, Node node) {
11. **int** ws = pred.waitStatus;
12. //当前继节点已经被标记为SIGNAL就可以放心地将自己阻塞，等待前继节点释放资源时唤醒自己，但是如果在阻塞自己之前被其他线程修改了前继节点的状态怎么办(???)
13. **if** (ws == Node.SIGNAL)
14. /\*
15. \* This node has already set status asking a release
16. \* to signal it, so it can safely park.
17. \*/
18. **return** **true**;
19. //只有CANCEL状态大于0
20. **if** (ws > 0) {
21. /\*
22. \* Predecessor was cancelled. Skip over predecessors and
23. \* indicate retry.
24. \*/
25. //重新组织一下链表
26. **do** {
27. node.prev = pred = pred.prev;
28. } **while** (pred.waitStatus > 0);
29. pred.next = node;
30. } **else** {
31. /\*
32. \* waitStatus must be 0 or PROPAGATE.  Indicate that we
33. \* need a signal, but don't park yet.  Caller will need to
34. \* retry to make sure it cannot acquire before parking.
35. \*/
36. //将前继节点用CAS操作改为SIGNAL状态，而且不关心结果如何，返回到acquireQueued，确定是否需要parking
37. compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
38. }
39. **return** **false**;
40. }

#### parkAndCheckInterrupt

1. /\*\*
2. \* Convenience method to park and then check if interrupted
3. \*
4. \* @return {@code true} if interrupted
5. \*/
6. **private** **final** **boolean** parkAndCheckInterrupt() {
7. LockSupport.park(**this**);
8. **return** Thread.interrupted();
9. }

### release

1. /\*\*
2. \* Releases in exclusive mode.  Implemented by unblocking one or
3. \* more threads if {@link #tryRelease} returns true.
4. \* This method can be used to implement method {@link Lock#unlock}.
5. \*
6. \* @param arg the release argument.  This value is conveyed to
7. \*        {@link #tryRelease} but is otherwise uninterpreted and
8. \*        can represent anything you like.
9. \* @return the value returned from {@link #tryRelease}
10. \*/
11. **public** **final** **boolean** release(**int** arg) {
12. **if** (tryRelease(arg)) {
13. Node h = head;
14. //队列不为空，并且队列头状态不是0状态时，可以看到并不是只有SIGNAL状态才会唤醒后继节点
15. **if** (h != **null** && h.waitStatus != 0)
16. unparkSuccessor(h);
17. **return** **true**;
18. }
19. **return** **false**;
20. }

#### tryRelease

1. /\*\*
2. \* Attempts to set the state to reflect a release in exclusive
3. \* mode.
4. \*
5. \* <p>This method is always invoked by the thread performing release.
6. \*
7. \* <p>The default implementation throws
8. \* {@link UnsupportedOperationException}.
9. \*
10. \* @param arg the release argument. This value is always the one
11. \*        passed to a release method, or the current state value upon
12. \*        entry to a condition wait.  The value is otherwise
13. \*        uninterpreted and can represent anything you like.
14. \* @return {@code true} if this object is now in a fully released
15. \*         state, so that any waiting threads may attempt to acquire;
16. \*         and {@code false} otherwise.
17. \* @throws IllegalMonitorStateException if releasing would place this
18. \*         synchronizer in an illegal state. This exception must be
19. \*         thrown in a consistent fashion for synchronization to work
20. \*         correctly.
21. \* @throws UnsupportedOperationException if exclusive mode is not supported
22. \*/
23. **protected** **boolean** tryRelease(**int** arg) {
24. **throw** **new** UnsupportedOperationException();
25. }

1、其语意交给子类实现

#### unparkSuccessor

1. /\*\*
2. \* Wakes up node's successor, if one exists.
3. \*
4. \* @param node the node
5. \*/
6. **private** **void** unparkSuccessor(Node node) {
7. /\*
8. \* If status is negative (i.e., possibly needing signal) try
9. \* to clear in anticipation of signalling.  It is OK if this
10. \* fails or if status is changed by waiting thread.
11. \*/
12. **int** ws = node.waitStatus;
13. //当前状态小于0，很可能是SIGNAL状态，尝试将状态置为0，不关心是否成功，或者成功后又被等待线程改为SIGNAL状态
14. **if** (ws < 0)
15. //这里可能会被其他线程改成别的状态，例如在共享模式下将状态成功改为0后进入该方法，在条件判断前，被另一个共享模式下的线程改为PROPAGATE状态，于是条件为真，又被另一个线程在此处改为SIGNAL状态
16. compareAndSetWaitStatus(node, ws, 0);
18. /\*
19. \* Thread to unpark is held in successor, which is normally
20. \* just the next node.  But if cancelled or apparently null,
21. \* traverse backwards from tail to find the actual
22. \* non-cancelled successor.
23. \*/
24. //用非可靠的next字段定位后继节点，如果为空不代表后继没有节点
25. Node s = node.next;
26. //如果为空或者处于CANCELL状态，那么从后往前找到最前面的正常状态的节点
27. **if** (s == **null** || s.waitStatus > 0) {
28. s = **null**;
29. **for** (Node t = tail; t != **null** && t != node; t = t.prev)
30. **if** (t.waitStatus <= 0)
31. s = t;
32. }
33. //s不可能是那个队列初始化时生成的空节点，因为向前找节点时，不会遍历到head节点
34. **if** (s != **null**)
35. LockSupport.unpark(s.thread);
36. }

### acquireShared

1. /\*\*
2. \* Acquires in shared mode, ignoring interrupts.  Implemented by
3. \* first invoking at least once {@link #tryAcquireShared},
4. \* returning on success.  Otherwise the thread is queued, possibly
5. \* repeatedly blocking and unblocking, invoking {@link
6. \* #tryAcquireShared} until success.
7. \*
8. \* @param arg the acquire argument.  This value is conveyed to
9. \*        {@link #tryAcquireShared} but is otherwise uninterpreted
10. \*        and can represent anything you like.
11. \*/
12. **public** **final** **void** acquireShared(**int** arg) {
13. **if** (tryAcquireShared(arg) < 0)
14. doAcquireShared(arg);
15. }

#### tryAcquireShared

1. /\*\*
2. \* Attempts to acquire in shared mode. This method should query if
3. \* the state of the object permits it to be acquired in the shared
4. \* mode, and if so to acquire it.
5. \*
6. \* <p>This method is always invoked by the thread performing
7. \* acquire.  If this method reports failure, the acquire method
8. \* may queue the thread, if it is not already queued, until it is
9. \* signalled by a release from some other thread.
10. \*
11. \* <p>The default implementation throws {@link
12. \* UnsupportedOperationException}.
13. \*
14. \* @param arg the acquire argument. This value is always the one
15. \*        passed to an acquire method, or is the value saved on entry
16. \*        to a condition wait.  The value is otherwise uninterpreted
17. \*        and can represent anything you like.
18. \* @return a negative value on failure; zero if acquisition in shared
19. \*         mode succeeded but no subsequent shared-mode acquire can
20. \*         succeed; and a positive value if acquisition in shared
21. \*         mode succeeded and subsequent shared-mode acquires might
22. \*         also succeed, in which case a subsequent waiting thread
23. \*         must check availability. (Support for three different
24. \*         return values enables this method to be used in contexts
25. \*         where acquires only sometimes act exclusively.)  Upon
26. \*         success, this object has been acquired.
27. \* @throws IllegalMonitorStateException if acquiring would place this
28. \*         synchronizer in an illegal state. This exception must be
29. \*         thrown in a consistent fashion for synchronization to work
30. \*         correctly.
31. \* @throws UnsupportedOperationException if shared mode is not supported
32. \*/
33. **protected** **int** tryAcquireShared(**int** arg) {
34. **throw** **new** UnsupportedOperationException();
35. }

#### doAcquireShared

1. /\*\*
2. \* Acquires in shared uninterruptible mode.
3. \* @param arg the acquire argument
4. \*/
5. **private** **void** doAcquireShared(**int** arg) {
6. //将节点入队，并标记为共享模式
7. **final** Node node = addWaiter(Node.SHARED);
8. **boolean** failed = **true**;
9. **try** {
10. **boolean** interrupted = **false**;
11. **for** (;;) {
12. //p一定不是null，否则会抛出NPE
13. **final** Node p = node.predecessor();
14. //当前节点的前继节点为头结点时，当前节点才有资格尝试获取资源
15. //这一点与独占模式完全相同，意味着当某节点未获取资源时，排在该节点后面的所有节点都无法获取资源，即便要求的资源其实很少
16. **if** (p == head) {
17. **int** r = tryAcquireShared(arg);
18. //当成功获取资源时，详见tryAcquireShared注释
19. **if** (r >= 0) {
20. //???
21. setHeadAndPropagate(node, r);
22. p.next = **null**; // help GC
23. **if** (interrupted)
24. //恢复中断现场
25. selfInterrupt();
26. failed = **false**;
27. **return**;
28. }
29. }
30. **if** (shouldParkAfterFailedAcquire(p, node) &&
31. parkAndCheckInterrupt())
32. interrupted = **true**;
33. }
34. } **finally** {
35. **if** (failed)
36. cancelAcquire(node);
37. }
38. }

#### setHeadAndPropagate

1. /\*\*
2. \* Sets head of queue, and checks if successor may be waiting
3. \* in shared mode, if so propagating if either propagate > 0 or
4. \* PROPAGATE status was set.
5. \*
6. \* @param node the node
7. \* @param propagate the return value from a tryAcquireShared
8. \*/
9. **private** **void** setHeadAndPropagate(Node node, **int** propagate) {
10. //当前节点node此时一定是队列中第二个节点，而且此时在队列中仅有当前线程是非阻塞的，那么h节点为什么可能是null???
11. Node h = head; // Record old head for check below
12. //在node节点成为队头后，其后继节点可能被其他释放资源的线程唤醒了，因此继续设置队头，导致head会被更新
13. setHead(node);
14. /\*
15. \* Try to signal next queued node if:
16. \*   Propagation was indicated by caller,
17. \*     or was recorded (as h.waitStatus either before
18. \*     or after setHead) by a previous operation
19. \*     (note: this uses sign-check of waitStatus because
20. \*      PROPAGATE status may transition to SIGNAL.)
21. \* and
22. \*   The next node is waiting in shared mode,
23. \*     or we don't know, because it appears null
24. \*
25. \* The conservatism in both of these checks may cause
26. \* unnecessary wake-ups, but only when there are multiple
27. \* racing acquires/releases, so most need signals now or soon
28. \* anyway.
29. \*/
30. // propagate > 0：在调用该方法的三个地方，传入的propagate都是保证>=0的，当还有剩余资源时，有必要继续唤醒队列中的节点
31. // h == null：怎么可能是null???
32. **if** (propagate > 0 || h == **null** || h.waitStatus < 0 ||
33. (h = head) == **null** || h.waitStatus < 0) {
34. Node s = node.next;
35. **if** (s == **null** || s.isShared())
36. doReleaseShared();
37. }
38. }

### releaseShared

1. /\*\*
2. \* Releases in shared mode.  Implemented by unblocking one or more
3. \* threads if {@link #tryReleaseShared} returns true.
4. \*
5. \* @param arg the release argument.  This value is conveyed to
6. \*        {@link #tryReleaseShared} but is otherwise uninterpreted
7. \*        and can represent anything you like.
8. \* @return the value returned from {@link #tryReleaseShared}
9. \*/
10. **public** **final** **boolean** releaseShared(**int** arg) {
11. **if** (tryReleaseShared(arg)) {
12. doReleaseShared();
13. **return** **true**;
14. }
15. **return** **false**;
16. }

#### tryReleaseShared

1. /\*\*
2. \* Attempts to set the state to reflect a release in shared mode.
3. \*
4. \* <p>This method is always invoked by the thread performing release.
5. \*
6. \* <p>The default implementation throws
7. \* {@link UnsupportedOperationException}.
8. \*
9. \* @param arg the release argument. This value is always the one
10. \*        passed to a release method, or the current state value upon
11. \*        entry to a condition wait.  The value is otherwise
12. \*        uninterpreted and can represent anything you like.
13. \* @return {@code true} if this release of shared mode may permit a
14. \*         waiting acquire (shared or exclusive) to succeed; and
15. \*         {@code false} otherwise
16. \* @throws IllegalMonitorStateException if releasing would place this
17. \*         synchronizer in an illegal state. This exception must be
18. \*         thrown in a consistent fashion for synchronization to work
19. \*         correctly.
20. \* @throws UnsupportedOperationException if shared mode is not supported
21. \*/
22. **protected** **boolean** tryReleaseShared(**int** arg) {
23. **throw** **new** UnsupportedOperationException();
24. }

1、交给子类实现其语意

#### doReleaseShared

1. /\*\*
2. \* Release action for shared mode -- signals successor and ensures
3. \* propagation. (Note: For exclusive mode, release just amounts
4. \* to calling unparkSuccessor of head if it needs signal.)
5. \*/
6. **private** **void** doReleaseShared() {
7. /\*
8. \* Ensure that a release propagates, even if there are other
9. \* in-progress acquires/releases.  This proceeds in the usual
10. \* way of trying to unparkSuccessor of head if it needs
11. \* signal. But if it does not, status is set to PROPAGATE to
12. \* ensure that upon release, propagation continues.
13. \* Additionally, we must loop in case a new node is added
14. \* while we are doing this. Also, unlike other uses of
15. \* unparkSuccessor, we need to know if CAS to reset status
16. \* fails, if so rechecking.
17. \*/
18. **for** (;;) {
19. Node h = head;
20. //h为空代表队列为空，h==tail代表队列就一个头节点，而头结点一定是拿到资源的节点
21. **if** (h != **null** && h != tail) {
22. **int** ws = h.waitStatus;
23. **if** (ws == Node.SIGNAL) {
24. **if** (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
25. **continue**;            // loop to recheck cases
26. //若成功将SIGNAL变为0，则唤醒后继节点
27. unparkSuccessor(h);
28. }
29. //仅此一处会将头节点状态改为PROPAGATE状态
30. //如果处于CANCEL状态或者CONDITION状态呢
31. **else** **if** (ws == 0 &&
32. !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
33. **continue**;                // loop on failed CAS
34. }
35. //头结点若发生变化(有一个线程执行了unparkSuccessor唤醒某节点n并且成功获取资源，然后将n设置为新的队头节点)则继续循环
36. **if** (h == head)                   // loop if head changed
37. **break**;
38. }
39. }

### acquireInterruptibly

1. /\*\*
2. \* Acquires in exclusive mode, aborting if interrupted.
3. \* Implemented by first checking interrupt status, then invoking
4. \* at least once {@link #tryAcquire}, returning on
5. \* success.  Otherwise the thread is queued, possibly repeatedly
6. \* blocking and unblocking, invoking {@link #tryAcquire}
7. \* until success or the thread is interrupted.  This method can be
8. \* used to implement method {@link Lock#lockInterruptibly}.
9. \*
10. \* @param arg the acquire argument.  This value is conveyed to
11. \*        {@link #tryAcquire} but is otherwise uninterpreted and
12. \*        can represent anything you like.
13. \* @throws InterruptedException if the current thread is interrupted
14. \*/
15. **public** **final** **void** acquireInterruptibly(**int** arg)
16. **throws** InterruptedException {
17. //若已经被中断了，则直接抛出中断异常
18. **if** (Thread.interrupted())
19. **throw** **new** InterruptedException();
20. **if** (!tryAcquire(arg))
21. doAcquireInterruptibly(arg);
22. }

#### doAcquireInterruptibly

1. /\*\*
2. \* Acquires in exclusive interruptible mode.
3. \* @param arg the acquire argument
4. \*/
5. **private** **void** doAcquireInterruptibly(**int** arg)
6. **throws** InterruptedException {
7. //将当前线程封装成独占模式的节点，并且添加到队列尾部
8. **final** Node node = addWaiter(Node.EXCLUSIVE);
9. **boolean** failed = **true**;
10. **try** {
11. **for** (;;) {
12. **final** Node p = node.predecessor();
13. //同样的只有当前节点为队列第二个节点时才有资格尝试获取节点
14. **if** (p == head && tryAcquire(arg)) {
15. setHead(node);
16. p.next = **null**; // help GC
17. failed = **false**;
18. **return**;
19. }
20. **if** (shouldParkAfterFailedAcquire(p, node) &&
21. parkAndCheckInterrupt())
22. //如果阻塞过程中被中断过，那么直接抛出中断异常
23. **throw** **new** InterruptedException();
24. }
25. } **finally** {
26. **if** (failed)
27. cancelAcquire(node);
28. }
29. }

### acquireSharedInterruptibly

1. /\*\*
2. \* Acquires in shared mode, aborting if interrupted.  Implemented
3. \* by first checking interrupt status, then invoking at least once
4. \* {@link #tryAcquireShared}, returning on success.  Otherwise the
5. \* thread is queued, possibly repeatedly blocking and unblocking,
6. \* invoking {@link #tryAcquireShared} until success or the thread
7. \* is interrupted.
8. \* @param arg the acquire argument.
9. \* This value is conveyed to {@link #tryAcquireShared} but is
10. \* otherwise uninterpreted and can represent anything
11. \* you like.
12. \* @throws InterruptedException if the current thread is interrupted
13. \*/
14. **public** **final** **void** acquireSharedInterruptibly(**int** arg)
15. **throws** InterruptedException {
16. //若已经被中断了，则直接抛出中断异常
17. **if** (Thread.interrupted())
18. **throw** **new** InterruptedException();
19. **if** (tryAcquireShared(arg) < 0)
20. doAcquireSharedInterruptibly(arg);
21. }

#### doAcquireSharedInterruptibly

1. /\*\*
2. \* Acquires in shared interruptible mode.
3. \* @param arg the acquire argument
4. \*/
5. **private** **void** doAcquireSharedInterruptibly(**int** arg)
6. **throws** InterruptedException {
7. //将线程封装成共享模式的节点，然后添加到队列尾部
8. **final** Node node = addWaiter(Node.SHARED);
9. **boolean** failed = **true**;
10. **try** {
11. **for** (;;) {
12. **final** Node p = node.predecessor();
13. **if** (p == head) {
14. **int** r = tryAcquireShared(arg);
15. **if** (r >= 0) {
16. //???
17. setHeadAndPropagate(node, r);
18. p.next = **null**; // help GC
19. failed = **false**;
20. **return**;
21. }
22. }
23. **if** (shouldParkAfterFailedAcquire(p, node) &&
24. parkAndCheckInterrupt())
25. //直接抛出异常响应中断
26. **throw** **new** InterruptedException();
27. }
28. } **finally** {
29. **if** (failed)
30. cancelAcquire(node);
31. }
32. }

### tryAcquireNanos

1. /\*\*
2. \* Attempts to acquire in exclusive mode, aborting if interrupted,
3. \* and failing if the given timeout elapses.  Implemented by first
4. \* checking interrupt status, then invoking at least once {@link
5. \* #tryAcquire}, returning on success.  Otherwise, the thread is
6. \* queued, possibly repeatedly blocking and unblocking, invoking
7. \* {@link #tryAcquire} until success or the thread is interrupted
8. \* or the timeout elapses.  This method can be used to implement
9. \* method {@link Lock#tryLock(long, TimeUnit)}.
10. \*
11. \* @param arg the acquire argument.  This value is conveyed to
12. \*        {@link #tryAcquire} but is otherwise uninterpreted and
13. \*        can represent anything you like.
14. \* @param nanosTimeout the maximum number of nanoseconds to wait
15. \* @return {@code true} if acquired; {@code false} if timed out
16. \* @throws InterruptedException if the current thread is interrupted
17. \*/
18. **public** **final** **boolean** tryAcquireNanos(**int** arg, **long** nanosTimeout)
19. **throws** InterruptedException {
20. //若已经被中断了，则直接抛出中断异常
21. **if** (Thread.interrupted())
22. **throw** **new** InterruptedException();
23. **return** tryAcquire(arg) ||
24. doAcquireNanos(arg, nanosTimeout);
25. }

#### doAcquireNanos

1. /\*\*
2. \* Acquires in exclusive timed mode.
3. \*
4. \* @param arg the acquire argument
5. \* @param nanosTimeout max wait time
6. \* @return {@code true} if acquired
7. \*/
8. **private** **boolean** doAcquireNanos(**int** arg, **long** nanosTimeout)
9. **throws** InterruptedException {
10. //剩余时间不够了，直接返回false
11. **if** (nanosTimeout <= 0L)
12. **return** **false**;
13. //截止时间
14. **final** **long** deadline = System.nanoTime() + nanosTimeout;
15. //将线程封装成独占模式节点，添加到队列尾
16. **final** Node node = addWaiter(Node.EXCLUSIVE);
17. **boolean** failed = **true**;
18. **try** {
19. **for** (;;) {
20. **final** Node p = node.predecessor();
21. **if** (p == head && tryAcquire(arg)) {
22. setHead(node);
23. p.next = **null**; // help GC
24. failed = **false**;
25. **return** **true**;
26. }
27. //若剩余时间不足，不再继续等待，直接返回false
28. nanosTimeout = deadline - System.nanoTime();
29. **if** (nanosTimeout <= 0L)
30. **return** **false**;
31. //如果剩余时间小于spinForTimeoutThreshold，则自旋等待
32. **if** (shouldParkAfterFailedAcquire(p, node) &&
33. nanosTimeout > spinForTimeoutThreshold)
34. LockSupport.parkNanos(**this**, nanosTimeout);
35. **if** (Thread.interrupted())
36. //直接抛出异常响应中断
37. **throw** **new** InterruptedException();
38. }
39. } **finally** {
40. **if** (failed)
41. cancelAcquire(node);
42. }
43. }

### tryAcquireSharedNanos

1. /\*\*
2. \* Attempts to acquire in shared mode, aborting if interrupted, and
3. \* failing if the given timeout elapses.  Implemented by first
4. \* checking interrupt status, then invoking at least once {@link
5. \* #tryAcquireShared}, returning on success.  Otherwise, the
6. \* thread is queued, possibly repeatedly blocking and unblocking,
7. \* invoking {@link #tryAcquireShared} until success or the thread
8. \* is interrupted or the timeout elapses.
9. \*
10. \* @param arg the acquire argument.  This value is conveyed to
11. \*        {@link #tryAcquireShared} but is otherwise uninterpreted
12. \*        and can represent anything you like.
13. \* @param nanosTimeout the maximum number of nanoseconds to wait
14. \* @return {@code true} if acquired; {@code false} if timed out
15. \* @throws InterruptedException if the current thread is interrupted
16. \*/
17. **public** **final** **boolean** tryAcquireSharedNanos(**int** arg, **long** nanosTimeout)
18. **throws** InterruptedException {
19. //若已经被中断了，则直接抛出中断异常
20. **if** (Thread.interrupted())
21. **throw** **new** InterruptedException();
22. **return** tryAcquireShared(arg) >= 0 ||
23. doAcquireSharedNanos(arg, nanosTimeout);
24. }

#### doAcquireSharedNanos

1. /\*\*
2. \* Acquires in shared timed mode.
3. \*
4. \* @param arg the acquire argument
5. \* @param nanosTimeout max wait time
6. \* @return {@code true} if acquired
7. \*/
8. **private** **boolean** doAcquireSharedNanos(**int** arg, **long** nanosTimeout)
9. **throws** InterruptedException {
10. //剩余时间不够了，直接返回false
11. **if** (nanosTimeout <= 0L)
12. **return** **false**;
13. //截止时间
14. **final** **long** deadline = System.nanoTime() + nanosTimeout;
15. //将线程封装成共享模式节点，添加到队列尾
16. **final** Node node = addWaiter(Node.SHARED);
17. **boolean** failed = **true**;
18. **try** {
19. **for** (;;) {
20. **final** Node p = node.predecessor();
21. **if** (p == head) {
22. **int** r = tryAcquireShared(arg);
23. **if** (r >= 0) {
24. //???
25. setHeadAndPropagate(node, r);
26. p.next = **null**; // help GC
27. failed = **false**;
28. **return** **true**;
29. }
30. }
31. nanosTimeout = deadline - System.nanoTime();
32. //若剩余时间不足，不再继续等待，直接返回false
33. **if** (nanosTimeout <= 0L)
34. **return** **false**;
35. //如果剩余时间小于spinForTimeoutThreshold，则自旋等待
36. **if** (shouldParkAfterFailedAcquire(p, node) &&
37. nanosTimeout > spinForTimeoutThreshold)
38. LockSupport.parkNanos(**this**, nanosTimeout);
39. **if** (Thread.interrupted())
40. //直接抛出异常响应中断
41. **throw** **new** InterruptedException();
42. }
43. } **finally** {
44. **if** (failed)
45. cancelAcquire(node);
46. }
47. }

# ConcurrentHashMap(JDK 1.8)

1、HashMap、ConcurrentHashMap只会扩张不会收缩

## 常量

### MAXIMUM\_CAPACITY

1. /\*\*
2. \* The largest possible table capacity.  This value must be
3. \* exactly 1<<30 to stay within Java array allocation and indexing
4. \* bounds for power of two table sizes, and is further required
5. \* because the top two bits of 32bit hash fields are used for
6. \* control purposes.
7. \*/
8. **private** **static** **final** **int** MAXIMUM\_CAPACITY = 1 << 30;

1、容器的最大容量，确切等于230。hash字段的最高两位用于存储控制信息

### DEFAULT\_CAPACITY

1. /\*\*
2. \* The default initial table capacity.  Must be a power of 2
3. \* (i.e., at least 1) and at most MAXIMUM\_CAPACITY.
4. \*/
5. **private** **static** **final** **int** DEFAULT\_CAPACITY = 16;

1、默认的table大小，必须是2的幂次，最大为MAXIMUM\_CAPACITY

### MAX\_ARRAY\_SIZE

1. /\*\*
2. \* The largest possible (non-power of two) array size.
3. \* Needed by toArray and related methods.
4. \*/
5. **static** **final** **int** MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;

1、数组最大容量

### DEFAULT\_CONCURRENCY\_LEVEL

1. /\*\*
2. \* The default concurrency level for this table. Unused but
3. \* defined for compatibility with previous versions of this class.
4. \*/
5. **private** **static** **final** **int** DEFAULT\_CONCURRENCY\_LEVEL = 16;

1、默认的并发级别，虽然定义了，但是没有使用，为了兼容以前的版本

### LOAD\_FACTOR

1. /\*\*
2. \* The load factor for this table. Overrides of this value in
3. \* constructors affect only the initial table capacity.  The
4. \* actual floating point value isn't normally used -- it is
5. \* simpler to use expressions such as {@code n - (n >>> 2)} for
6. \* the associated resizing threshold.
7. \*/
8. **private** **static** **final** **float** LOAD\_FACTOR = 0.75f;

1、负载因子，实际过程中用的是n - (n >>> 2)来表示n\* LOAD\_FACTOR

2、当ConcurrentHashMap中存储元素个数/table表大小大于0.75时，会进行扩容，即边界值为table.length\*LOAD\_FACTORY，即sizeCtl的值

### TREEIFY\_THRESHOLD

1. /\*\*
2. \* The bin count threshold for using a tree rather than list for a
3. \* bin.  Bins are converted to trees when adding an element to a
4. \* bin with at least this many nodes. The value must be greater
5. \* than 2, and should be at least 8 to mesh with assumptions in
6. \* tree removal about conversion back to plain bins upon
7. \* shrinkage.
8. \*/
9. **static** **final** **int** TREEIFY\_THRESHOLD = 8;

1、当由于插入操作导致某个槽位中Node数量超过8时，将会用红黑树来组织这些Node

### UNTREEIFY\_THRESHOLD

1. /\*\*
2. \* The bin count threshold for untreeifying a (split) bin during a
3. \* resize operation. Should be less than TREEIFY\_THRESHOLD, and at
4. \* most 6 to mesh with shrinkage detection under removal.
5. \*/
6. **static** **final** **int** UNTREEIFY\_THRESHOLD = 6;

1、当由于删除操作导致某个槽位中Node数量低于6时，将会用链表来组织这些Node

2、为什么UNTREEIFY\_THRESHOLD和TREEIFY\_THRESHOLD设置为不同的值？我觉得这是为了避免插入删除交替执行时会造成Node组织结构在链表和红黑树之间不断地切换

### MIN\_TREEIFY\_CAPACITY

1. /\*\*
2. \* The smallest table capacity for which bins may be treeified.
3. \* (Otherwise the table is resized if too many nodes in a bin.)
4. \* The value should be at least 4 \* TREEIFY\_THRESHOLD to avoid
5. \* conflicts between resizing and treeification thresholds.
6. \*/
7. **static** **final** **int** MIN\_TREEIFY\_CAPACITY = 64;

1、bin中节点被树化所对应的容器的最小容量，否则容器应该扩容，因为一个槽位容纳了太多的节点

### MIN\_TRANSFER\_STRIDE

1. /\*\*
2. \* Minimum number of rebinnings per transfer step. Ranges are
3. \* subdivided to allow multiple resizer threads.  This value
4. \* serves as a lower bound to avoid resizers encountering
5. \* excessive memory contention.  The value should be at least
6. \* DEFAULT\_CAPACITY.
7. \*/
8. **private** **static** **final** **int** MIN\_TRANSFER\_STRIDE = 16;

1、扩容时每个CPU最少分得的槽位数量

### RESIZE\_STAMP\_BITS

1. /\*\*
2. \* The number of bits used for generation stamp in sizeCtl.
3. \* Must be at least 6 for 32bit arrays.
4. \*/
5. **private** **static** **int** RESIZE\_STAMP\_BITS = 16;

1、sizeCtl中用于生成一个stamp所用的位数

### MAX\_RESIZERS

1. /\*\*
2. \* The maximum number of threads that can help resize.
3. \* Must fit in 32 - RESIZE\_STAMP\_BITS bits.
4. \*/
5. **private** **static** **final** **int** MAX\_RESIZERS = (1 << (32 - RESIZE\_STAMP\_BITS)) - 1;

1、表扩张或收缩时，最多可以用多少个线程来并发执行

### RESIZE\_STAMP\_SHIFT

1. /\*\*
2. \* The bit shift for recording size stamp in sizeCtl.
3. \*/
4. **private** **static** **final** **int** RESIZE\_STAMP\_SHIFT = 32 - RESIZE\_STAMP\_BITS;

1、与RESIZE\_STAMP\_BITS相对应，用于左移resizeStamp方法的返回值，来生成一个负数

### Node#控制位状态解释

1. /\*
2. \* Encodings for Node hash fields. See above for explanation.
3. \*/
4. **static** **final** **int** MOVED     = -1; // hash for forwarding nodes
5. **static** **final** **int** TREEBIN   = -2; // hash for roots of trees
6. **static** **final** **int** RESERVED  = -3; // hash for transient reservations
7. **static** **final** **int** HASH\_BITS = 0x7fffffff; // usable bits of normal node hash

1、Node节点hash值所对应的含义

1. MOVED：当前Node是ForwardingNode
2. TREEBIN：当前Node是TreeBin节点(红黑树)
3. RESERVED：当前Node是???

### NCPU

1. /\*\* Number of CPUS, to place bounds on some sizings \*/
2. **static** **final** **int** NCPU = Runtime.getRuntime().availableProcessors();

1、获取CPU数量

## 字段

### table

1. /\*\*
2. \* The array of bins. Lazily initialized upon first insertion.
3. \* Size is always a power of two. Accessed directly by iterators.
4. \*/
5. **transient** **volatile** Node<K,V>[] table;

1、bins的数组，延迟加载(直到第一次插入操作)，大小一定是2的幂次，可被迭代器直接访问

2、为什么大小一定是2的幂次

1. 其他方法中会用到i = (table.length - 1) & hash来进行求模运算，由于table的大小是2的幂次，因此table.length-1是"000...11...1"的样子，直接用于运算即可求模
2. 在表扩张过程中，设原表长度为n，某个槽位i的节点会分为两类，一类放入新表的槽位i，一类放入新表的槽位i+1，如下图所示

* 如果hash的第X位是0，那么在表扩张后，仍然处于槽位i，即

(nextTable.length - 1) & hash == (table.length - 1) & hash

* 如果hash的第X位是1，那么在表扩张后，处于的槽位是i+2X=i+table.length，即

(nextTable.length - 1) & hash == (table.length - 1) & hash + table.length



### nextTable

1. /\*\*
2. \* The next table to use; non-null only while resizing.
3. \*/
4. **private** **transient** **volatile** Node<K,V>[] nextTable;

1、扩容时会用到，正常情况下为null

### baseCount

1. /\*\*
2. \* Base counter value, used mainly when there is no contention,
3. \* but also as a fallback during table initialization
4. \* races. Updated via CAS.
5. \*/
6. **private** **transient** **volatile** **long** baseCount;

1、计数值，通常在无竞争的情况下被使用，通过CAS操作来更新

### sizeCtl

1. /\*\*
2. \* Table initialization and resizing control.  When negative, the
3. \* table is being initialized or resized: -1 for initialization,
4. \* else -(1 + the number of active resizing threads).  Otherwise,
5. \* when table is null, holds the initial table size to use upon
6. \* creation, or 0 for default. After initialization, holds the
7. \* next element count value upon which to resize the table.
8. \*/
9. **private** **transient** **volatile** **int** sizeCtl;

1、table初始化或者调整大小的控制信息

1. -1：table正在进行初始化
2. -N(N>1)：~~正在resizing，N-1就是参与resizing的线程数量~~，实际代码中的情况好像与注释描述的不一致，当进行resizing时，会将sizeCtl赋值为一个很小的负值

(resizeStamp(n) << RESIZE\_STAMP\_SHIFT) + 2

1. 其他情况：

* 当table未初始化，表示table需要初始化的大小
* 当table初始化完成，表示table需要扩容时的边界大小，超过此值则需要进行扩容操作

### transferIndex

1. /\*\*
2. \* The next table index (plus one) to split while resizing.
3. \*/
4. **private** **transient** **volatile** **int** transferIndex;

1、ConcurrentHashMap含有一个属性transferIndex(原table.length+1)，表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride

### cellsBusy

1. /\*\*
2. \* Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
3. \*/
4. **private** **transient** **volatile** **int** cellsBusy;

1、<未完成>：不懂啥意思

### counterCells

1. /\*\*
2. \* Table of counter cells. When non-null, size is a power of 2.
3. \*/
4. **private** **transient** **volatile** CounterCell[] counterCells;

1、<未完成>：不懂啥意思

### views

1. // views
2. **private** **transient** KeySetView<K,V> keySet;
3. **private** **transient** ValuesView<K,V> values;
4. **private** **transient** EntrySetView<K,V> entrySet;

1、<未完成>：不懂啥意思

### unsafe字段

1. **private** **static** **final** sun.misc.Unsafe U;
2. //以下常量用于计算偏移量，因为Unsafe的CAS操作是基于字段偏移量的
3. **private** **static** **final** **long** SIZECTL;  //对应于sizeCtl
4. **private** **static** **final** **long** TRANSFERINDEX;  //对应于transferIndex
5. **private** **static** **final** **long** BASECOUNT;  //对应于baseCount
6. **private** **static** **final** **long** CELLSBUSY;  //对应于cellsBusy
7. **private** **static** **final** **long** CELLVALUE;  //对应于CounterCell#value
8. **private** **static** **final** **long** ABASE;  //Node[]数字头元素的偏移量
9. **private** **static** **final** **int** ASHIFT;  //对应于Node[]元素(引用，一般4字节)大小的偏移量，由于4字节是00000...100，即1<<2，因此ASHIFT为2
11. **static** {
12. **try** {
13. U = sun.misc.Unsafe.getUnsafe();
14. Class<?> k = ConcurrentHashMap.**class**;
15. SIZECTL = U.objectFieldOffset
16. (k.getDeclaredField("sizeCtl"));
17. TRANSFERINDEX = U.objectFieldOffset
18. (k.getDeclaredField("transferIndex"));
19. BASECOUNT = U.objectFieldOffset
20. (k.getDeclaredField("baseCount"));
21. CELLSBUSY = U.objectFieldOffset
22. (k.getDeclaredField("cellsBusy"));
23. Class<?> ck = CounterCell.**class**;
24. CELLVALUE = U.objectFieldOffset
25. (ck.getDeclaredField("value"));
26. Class<?> ak = Node[].**class**;
27. ABASE = U.arrayBaseOffset(ak);
28. **int** scale = U.arrayIndexScale(ak);
29. **if** ((scale & (scale - 1)) != 0)
30. **throw** **new** Error("data type scale not a power of two");
31. ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
32. } **catch** (Exception e) {
33. **throw** **new** Error(e);
34. }
35. }

## 内部类

### Node

1. /\*\*
2. \* Key-value entry.  This class is never exported out as a
3. \* user-mutable Map.Entry (i.e., one supporting setValue; see
4. \* MapEntry below), but can be used for read-only traversals used
5. \* in bulk tasks.  Subclasses of Node with a negative hash field
6. \* are special, and contain null keys and values (but are never
7. \* exported).  Otherwise, keys and vals are never null.
8. \*/
9. **static** **class** Node<K,V> **implements** Map.Entry<K,V> {
10. **final** **int** hash;
11. **final** K key;
12. **volatile** V val;
13. **volatile** Node<K,V> next;
15. Node(**int** hash, K key, V val, Node<K,V> next) {
16. **this**.hash = hash;
17. **this**.key = key;
18. **this**.val = val;
19. **this**.next = next;
20. }
22. **public** **final** K getKey()       { **return** key; }
23. **public** **final** V getValue()     { **return** val; }
24. **public** **final** **int** hashCode()   { **return** key.hashCode() ^ val.hashCode(); }
25. **public** **final** String toString(){ **return** key + "=" + val; }
26. **public** **final** V setValue(V value) {
27. **throw** **new** UnsupportedOperationException();
28. }
30. **public** **final** **boolean** equals(Object o) {
31. Object k, v, u; Map.Entry<?,?> e;
32. **return** ((o **instanceof** Map.Entry) &&
33. (k = (e = (Map.Entry<?,?>)o).getKey()) != **null** &&
34. (v = e.getValue()) != **null** &&
35. (k == key || k.equals(key)) &&
36. (v == (u = val) || v.equals(u)));
37. }
39. /\*\*
40. \* Virtualized support for map.get(); overridden in subclasses.
41. \*/
42. Node<K,V> find(**int** h, Object k) {
43. Node<K,V> e = **this**;
44. **if** (k != **null**) {
45. **do** {
46. K ek;
47. **if** (e.hash == h &&
48. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
49. **return** e;
50. } **while** ((e = e.next) != **null**);
51. }
52. **return** **null**;
53. }
54. }

### TreeNode

1. **static** **final** **class** TreeNode<K,V> **extends** Node<K,V> {
2. TreeNode<K,V> parent;  // red-black tree links
3. TreeNode<K,V> left;
4. TreeNode<K,V> right;
5. TreeNode<K,V> prev;    // needed to unlink next upon deletion
6. **boolean** red;
8. TreeNode(**int** hash, K key, V val, Node<K,V> next,
9. TreeNode<K,V> parent) {
10. **super**(hash, key, val, next);
11. **this**.parent = parent;
12. }
14. Node<K,V> find(**int** h, Object k) {
15. **return** findTreeNode(h, k, **null**);
16. }
18. /\*\*
19. \* Returns the TreeNode (or null if not found) for the given key
20. \* starting at given root.
21. \*/
22. **final** TreeNode<K,V> findTreeNode(**int** h, Object k, Class<?> kc) {
23. **if** (k != **null**) {
24. TreeNode<K,V> p = **this**;
25. **do**  {
26. **int** ph, dir; K pk; TreeNode<K,V> q;
27. TreeNode<K,V> pl = p.left, pr = p.right;
28. **if** ((ph = p.hash) > h)
29. p = pl;
30. **else** **if** (ph < h)
31. p = pr;
32. **else** **if** ((pk = p.key) == k || (pk != **null** && k.equals(pk)))
33. **return** p;
34. **else** **if** (pl == **null**)
35. p = pr;
36. **else** **if** (pr == **null**)
37. p = pl;
38. **else** **if** ((kc != **null** ||
39. (kc = comparableClassFor(k)) != **null**) &&
40. (dir = compareComparables(kc, k, pk)) != 0)
41. p = (dir < 0) ? pl : pr;
42. **else** **if** ((q = pr.findTreeNode(h, k, kc)) != **null**)
43. **return** q;
44. **else**
45. p = pl;
46. } **while** (p != **null**);
47. }
48. **return** **null**;
49. }
50. }

### TreeBin

### ForwardingNode

1. **static** **final** **class** ForwardingNode<K,V> **extends** Node<K,V> {
2. **final** Node<K,V>[] nextTable;
3. ForwardingNode(Node<K,V>[] tab) {
4. **super**(MOVED, **null**, **null**, **null**);
5. **this**.nextTable = tab;
6. }
8. Node<K,V> find(**int** h, Object k) {
9. // loop to avoid arbitrarily deep recursion on forwarding nodes
10. outer: **for** (Node<K,V>[] tab = nextTable;;) {
11. Node<K,V> e; **int** n;
12. **if** (k == **null** || tab == **null** || (n = tab.length) == 0 ||
13. (e = tabAt(tab, (n - 1) & h)) == **null**)
14. **return** **null**;
15. **for** (;;) {
16. **int** eh; K ek;
17. **if** ((eh = e.hash) == h &&
18. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
19. **return** e;
20. **if** (eh < 0) {
21. **if** (e **instanceof** ForwardingNode) {
22. tab = ((ForwardingNode<K,V>)e).nextTable;
23. **continue** outer;
24. }
25. **else**
26. **return** e.find(h, k);
27. }
28. **if** ((e = e.next) == **null**)
29. **return** **null**;
30. }
31. }
32. }
33. }

1、ForwardingNode：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用

2、**只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点已经被处理过了(槽位即便没有节点也需要在移动后放置ForwardingNode节点)**

## Utils方法

### spread

1. /\*\*
2. \* Spreads (XORs) higher bits of hash to lower and also forces top
3. \* bit to 0. Because the table uses power-of-two masking, sets of
4. \* hashes that vary only in bits above the current mask will
5. \* always collide. (Among known examples are sets of Float keys
6. \* holding consecutive whole numbers in small tables.)  So we
7. \* apply a transform that spreads the impact of higher bits
8. \* downward. There is a tradeoff between speed, utility, and
9. \* quality of bit-spreading. Because many common sets of hashes
10. \* are already reasonably distributed (so don't benefit from
11. \* spreading), and because we use trees to handle large sets of
12. \* collisions in bins, we just XOR some shifted bits in the
13. \* cheapest possible way to reduce systematic lossage, as well as
14. \* to incorporate impact of the highest bits that would otherwise
15. \* never be used in index calculations because of table bounds.
16. \*/
17. **static** **final** **int** spread(**int** h) {
18. **return** (h ^ (h >>> 16)) & HASH\_BITS;
19. }

1、该方法用于对hashCode()做一些调整，这样调整的原因有

* 使最高位(符号位为0)
* 在一个小表中，一系列浮动的键将会保持连续性
* 由于表的大小一般不会太大，因此高位对于槽位计算的影响将会微乎其微，因此将高位的影响向低位扩展，这样h ^ (h >>> 16)得到的值的低16位将由高位和低位共同决定，高16位仍由高位决定

### tableSizeFor

1. /\*\*
2. \* Returns a power of two table size for the given desired capacity.
3. \* See Hackers Delight, sec 3.2
4. \*/
5. **private** **static** **final** **int** tableSizeFor(**int** c) {
6. **int** n = c - 1;
7. n |= n >>> 1;
8. n |= n >>> 2;
9. n |= n >>> 4;
10. n |= n >>> 8;
11. n |= n >>> 16;
12. **return** (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;
13. }

1、将c调整为大于c的2的幂次方，真尼玛神奇

### comparableClassFor

1. /\*\*
2. \* Returns x's Class if it is of the form "class C implements
3. \* Comparable<C>", else null.
4. \*/
5. **static** Class<?> comparableClassFor(Object x) {
6. **if** (x **instanceof** Comparable) {
7. Class<?> c; Type[] ts, as; Type t; ParameterizedType p;
8. **if** ((c = x.getClass()) == String.**class**) // bypass checks
9. **return** c;
10. **if** ((ts = c.getGenericInterfaces()) != **null**) {
11. **for** (**int** i = 0; i < ts.length; ++i) {
12. **if** (((t = ts[i]) **instanceof** ParameterizedType) &&
13. ((p = (ParameterizedType)t).getRawType() ==
14. Comparable.**class**) &&
15. (as = p.getActualTypeArguments()) != **null** &&
16. as.length == 1 && as[0] == c) // type arg is c
17. **return** c;
18. }
19. }
20. }
21. **return** **null**;
22. }

### compareComparables

1. /\*\*
2. \* Returns k.compareTo(x) if x matches kc (k's screened comparable
3. \* class), else 0.
4. \*/
5. @SuppressWarnings({"rawtypes","unchecked"}) // for cast to Comparable
6. **static** **int** compareComparables(Class<?> kc, Object k, Object x) {
7. **return** (x == **null** || x.getClass() != kc ? 0 :
8. ((Comparable)k).compareTo(x));
9. }

### table access methods

1. /\*
2. \* Volatile access methods are used for table elements as well as
3. \* elements of in-progress next table while resizing.  All uses of
4. \* the tab arguments must be null checked by callers.  All callers
5. \* also paranoically precheck that tab's length is not zero (or an
6. \* equivalent check), thus ensuring that any index argument taking
7. \* the form of a hash value anded with (length - 1) is a valid
8. \* index.  Note that, to be correct wrt arbitrary concurrency
9. \* errors by users, these checks must operate on local variables,
10. \* which accounts for some odd-looking inline assignments below.
11. \* Note that calls to setTabAt always occur within locked regions,
12. \* and so in principle require only release ordering, not
13. \* full volatile semantics, but are currently coded as volatile
14. \* writes to be conservative.
15. \*/
17. @SuppressWarnings("unchecked")
18. //由于ASHIFT是数组元素的字节大小的偏移量，而i是数组索引，因此i<<ASHIFT就是i个数组元素的字节数量，ABASE就是数组首元素的偏移量，因此整体的((long)i << ASHIFT) + ABASE就是下标为i的数组元素在内存中的偏移量
19. **static** **final** <K,V> Node<K,V> tabAt(Node<K,V>[] tab, **int** i) {
20. **return** (Node<K,V>)U.getObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE);
21. }
23. **static** **final** <K,V> **boolean** casTabAt(Node<K,V>[] tab, **int** i,
24. Node<K,V> c, Node<K,V> v) {
25. **return** U.compareAndSwapObject(tab, ((**long**)i << ASHIFT) + ABASE, c, v);
26. }
28. **static** **final** <K,V> **void** setTabAt(Node<K,V>[] tab, **int** i, Node<K,V> v) {
29. U.putObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE, v);
30. }

### resizeStamp

1. /\*\*
2. \* Returns the stamp bits for resizing a table of size n.
3. \* Must be negative when shifted left by RESIZE\_STAMP\_SHIFT.
4. \*/
5. **static** final **int** resizeStamp(**int** n) {
6. **return** Integer.numberOfLeadingZeros(n) | (1 << (RESIZE\_STAMP\_BITS - 1));
7. }

1、返回值左移RESIZE\_STAMP\_SHIFT必须是个负数

## 方法

### put

1. **public** V put(K key, V value) {
2. **return** putVal(key, value, **false**);
3. }

#### putVal

1. /\*\* Implementation for put and putIfAbsent \*/
2. **final** V putVal(K key, V value, **boolean** onlyIfAbsent) {
3. **if** (key == **null** || value == **null**) **throw** **new** NullPointerException();
4. //对hashCode()的返回值进行改造，作为Node的hash值
5. **int** hash = spread(key.hashCode());
6. **int** binCount = 0;
7. **for** (Node<K,V>[] tab = table;;) {
8. Node<K,V> f; **int** n, i, fh;
9. //当表未初始化时，初始化表
10. **if** (tab == **null** || (n = tab.length) == 0)
11. tab = initTable();
12. //i是table中定位索引位置，n是table的大小
13. //有个小问题：如果添加元素时，表进行了一次扩容，那么下面的语句不会将元素添加到原来的table中去吗：不会，槽位为空说明至少该槽位尚未转移，可以先通过CAS插入进去，后续的参与并发扩容的线程会正确地维护这个新插入的节点
14. //另一个问题：为什么需要依赖Unsafe. getObjectVolatile来读取节点，table本身是有volatile关键字的，不是已经能保证每次读取操(字节码指令use)作能得到最新的值么???
15. **else** **if** ((f = tabAt(tab, i = (n - 1) & hash)) == **null**) {
16. **if** (casTabAt(tab, i, **null**,
17. **new** Node<K,V>(hash, key, value, **null**)))
18. **break**;                   // no lock when adding to empty bin
19. }
20. //如果f的hash值为-1，说明当前f是ForwardingNode节点，意味着有其他线程正在扩容，则一起进行扩容操作
21. **else** **if** ((fh = f.hash) == MOVED)
22. tab = helpTransfer(tab, f);
23. **else** {
24. //其余情况把新的Node节点按链表或红黑树的方式插入到合适的位置，这个过程采用同步内置锁实现并发
25. V oldVal = **null**;
26. **synchronized** (f) {
27. //再次进行判断，防止被其他线程修改
28. **if** (tabAt(tab, i) == f) {
29. //f节点是一个链表节点
30. **if** (fh >= 0) {
31. binCount = 1;
32. //将新值更新到链表中(若存在键相同的节点，则更新值，否则插入到链表尾部)
33. **for** (Node<K,V> e = f;; ++binCount) {
34. K ek;
35. **if** (e.hash == hash &&
36. ((ek = e.key) == key ||
37. (ek != **null** && key.equals(ek)))) {
38. oldVal = e.val;
39. //onlyIfAbsent为true表示不允许键已存在
40. **if** (!onlyIfAbsent)
41. e.val = value;
42. **break**;
43. }
44. Node<K,V> pred = e;
45. **if** ((e = e.next) == **null**) {
46. pred.next = **new** Node<K,V>(hash, key,
47. value, **null**);
48. **break**;
49. }
50. //可能会有else吗
51. }
52. }
53. //f节点是TreeBin节点(一颗红黑树)，则通过红黑树插入
54. **else** **if** (f **instanceof** TreeBin) {
55. Node<K,V> p;
56. //手动置为安全数值，避免下面进行的将链表转换成红黑树的操作
57. binCount = 2;
58. //putTreeVal方法查找或添加一个节点，当添加节点时返回null，若节点已存在，则返回该节点
59. **if** ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
60. value)) != **null**) {
61. oldVal = p.val;
62. **if** (!onlyIfAbsent)
63. p.val = value;
64. }
65. }
66. }
67. }
68. **if** (binCount != 0) {
69. //当节点数量大于阈值，需要将链表转为红黑树
70. **if** (binCount >= TREEIFY\_THRESHOLD)
71. treeifyBin(tab, i);
72. **if** (oldVal != **null**)
73. **return** oldVal;
74. **break**;
75. }
76. }
77. }
78. //更新baseCount计数值，若有必要，则进行扩容
79. addCount(1L, binCount);
80. **return** **null**;
81. }

#### initTable

1. **private** **final** Node<K,V>[] initTable() {
2. Node<K,V>[] tab; **int** sc;
3. //在调用本方法前，都会进行类似"(tab = table) == null || tab.length == 0"这样的判断，为什么这里还需要再次判断：因为可能有多个线程同时触发了该方法，而只有一个线程能够执行初始化操作
4. **while** ((tab = table) == **null** || tab.length == 0) {
5. //已经有其他线程正在执行初始化操作，因此当前线程让出CPU时间，进行自旋，直至初始化结束(退出while循环)
6. **if** ((sc = sizeCtl) < 0)
7. Thread.yield(); // lost initialization race; just spin
8. //尝试用CAS操作更改sizeCtl，将其置为-1，初始化只能有一个线程。若成功，那么执行初始化操作的就是当前线程，否则会自旋直至其他线程初始化table完毕
9. **else** **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, -1)) {
10. **try** {
11. //这里为什么需要再次进行判断???
12. **if** ((tab = table) == **null** || tab.length == 0) {
13. **int** n = (sc > 0) ? sc : DEFAULT\_CAPACITY;
14. @SuppressWarnings("unchecked")
15. Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n];
16. table = tab = nt;
17. //等价于0.75\*n，用位运算效率较高
18. sc = n - (n >>> 2);
19. }
20. } **finally** {
21. sizeCtl = sc;
22. }
23. **break**;
24. }
25. }
26. **return** tab;
27. }

1、sizeCtl默认为0，当调用带有初始化大小的构造方法时，会将该指定大小通过方法tableSizeFor调整为2的幂次

#### treeifyBin

1. **private** final **void** treeifyBin(Node<K,V>[] tab, **int** index) {
2. Node<K,V> b; **int** n, sc;
3. **if** (tab != null) {
4. **if** ((n = tab.length) < MIN\_TREEIFY\_CAPACITY)
5. tryPresize(n << 1);
6. **else** **if** ((b = tabAt(tab, index)) != null && b.hash >= 0) {
7. synchronized (b) {
8. **if** (tabAt(tab, index) == b) {
9. TreeNode<K,V> hd = null, tl = null;
10. **for** (Node<K,V> e = b; e != null; e = e.next) {
11. TreeNode<K,V> p =
12. **new** TreeNode<K,V>(e.hash, e.key, e.val,
13. null, null);
14. **if** ((p.prev = tl) == null)
15. hd = p;
16. **else**
17. tl.next = p;
18. tl = p;
19. }
20. setTabAt(tab, index, **new** TreeBin<K,V>(hd));
21. }
22. }
23. }
24. }
25. }

#### addCount

1. /\*\*
2. \* Adds to count, and if table is too small and not already
3. \* resizing, initiates transfer. If already resizing, helps
4. \* perform transfer if work is available.  Rechecks occupancy
5. \* after a transfer to see if another resize is already needed
6. \* because resizings are lagging additions.
7. \*
8. \* @param x the count to add
9. \* @param check if <0, don't check resize, if <= 1 only check if uncontended
10. \*/
11. **private** final **void** addCount(**long** x, **int** check) {
12. CounterCell[] as; **long** b, s;
13. // counterCells不为空时，baseCount是不准确的，必须用sumCount重新计算，或者counterCells为空，但是baseCount更新失败，也需要进入下面的逻辑
14. **if** ((as = counterCells) != null ||
15. !U.compareAndSwapLong(**this**, BASECOUNT, b = baseCount, s = b + x)) {   //下面这堆看不懂???
16. CounterCell a; **long** v; **int** m;
17. boolean uncontended = **true**;
18. **if** (as == null || (m = as.length - 1) < 0 ||
19. (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
20. !(uncontended =
21. U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
22. fullAddCount(x, uncontended);
23. **return**;
24. }
25. **if** (check <= 1)
26. **return**;
27. s = sumCount();
28. }
29. //到这里s必定为b+x或者sumCount(),即ConcurrentHashMap#size()，即元素数量
30. **if** (check >= 0) {
31. Node<K,V>[] tab, nt; **int** n, sc;
32. //进入while循环的条件分析：元素数量大于sizeCtl，table不为空，且table容量小于最大容量
33. **while** (s >= (**long**)(sc = sizeCtl) && (tab = table) != null &&
34. (n = tab.length) < MAXIMUM\_CAPACITY) {
35. **int** rs = resizeStamp(n);
36. //当前表正在扩容
37. **if** (sc < 0) {
38. //下面这堆条件看不懂(是一些异常情况么)???
39. **if** ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||
40. sc == rs + MAX\_RESIZERS || (nt = nextTable) == null ||
41. transferIndex <= 0)
42. **break**;
43. //如果能运行到这里，当前线程即将参与并发扩容
44. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, sc + 1))
45. transfer(tab, nt);
46. }
47. //在transfer中会用(rs << RESIZE\_STAMP\_SHIFT) + 2来判断是否还有其他线程正在处理，因此这个值得含义非常重要，必须通过CAS串行化
48. **else** **if** (U.compareAndSwapInt(**this**, SIZECTL, sc,
49. (rs << RESIZE\_STAMP\_SHIFT) + 2))
50. transfer(tab, null);
51. s = sumCount();
52. }
53. }
54. }

#### sumCount

1. final **long** sumCount() {
2. CounterCell[] as = counterCells; CounterCell a;
3. **long** sum = baseCount;
4. **if** (as != null) {
5. **for** (**int** i = 0; i < as.length; ++i) {
6. **if** ((a = as[i]) != null)
7. sum += a.value;
8. }
9. }
10. **return** sum;
11. }

1、counterCells有什么用???

#### transfer

1. **private** final **void** transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
2. **int** n = tab.length, stride;
3. //stride即是每个分片的大小，目前有最低要求16，即每个分片至少需要16个桶。stride的计算依赖于CPU的核数，如果只有1个核，那么此时就不用分片，即stride=n。其他情况就是 (n >>> 3) / NCPU
4. **if** ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN\_TRANSFER\_STRIDE)
5. stride = MIN\_TRANSFER\_STRIDE; // subdivide range
6. **if** (nextTab == null) {            // initiating
7. **try** {
8. @SuppressWarnings("unchecked")
9. //新的table大小为之前的两倍
10. Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n << 1];
11. nextTab = nt;
12. } **catch** (Throwable ex) {      // try to cope with OOME
13. sizeCtl = Integer.MAX\_VALUE;
14. **return**;
15. }
16. nextTable = nextTab;
17. //初值赋值为n，表示从n开始到最后的桶都已经迁移完毕
18. transferIndex = n;
19. }
20. **int** nextn = nextTab.length;
21. ForwardingNode<K,V> fwd = **new** ForwardingNode<K,V>(nextTab);
22. boolean advance = **true**;
23. boolean finishing = **false**; // to ensure sweep before committing nextTab   //i是当前槽位序号，bound是当前线程所分得的分片的下边界，当前线程处理的分片区域为[bound,bound+stride)
24. **for** (**int** i = 0, bound = 0;;) {
25. Node<K,V> f; **int** fh;
26. //下面的循环进行i索引的递减以及分片操作
27. **while** (advance) {
28. **int** nextIndex, nextBound;
29. //当前线程需要继续在当前分片上执行节点转移操作，或者finishing(当前线程是最后一个线程了)
30. **if** (--i >= bound || finishing)
31. advance = **false**;
32. //transferIndex<=0表示所有的桶已经全部分配出去了，当前线程没有任务可以领取了
33. **else** **if** ((nextIndex = transferIndex) <= 0) {
34. i = -1;
35. advance = **false**;
36. }
37. //这里利用CAS操作进行分片，保证并发条件下分片的串行化，分得的桶数量为stride
38. **else** **if** (U.compareAndSwapInt
39. (**this**, TRANSFERINDEX, nextIndex,
40. nextBound = (nextIndex > stride ?
41. nextIndex - stride : 0))) {
42. bound = nextBound;  //设置当前线程的下边界
43. i = nextIndex - 1;  //当前线程所分配的最后一个槽位，依次向前进行处理
44. advance = **false**;
45. }
46. }
47. //当槽位号越界时
48. **if** (i < 0 || i >= n || i + n >= nextn) {
49. **int** sc;
50. //当扩容完成时的收尾工作，只有一个线程来完成，如何保证只有一个线程执行下面的代码：只有最后一个线程才会将finishing设为1
51. **if** (finishing) {
52. nextTable = null;  //将nextTable置空，该字段只有扩容时有效
53. table = nextTab;  //更新table
54. sizeCtl = (n << 1) - (n >>> 1);  //为n的1.5倍，即下次需要扩容的边界，注意n是原table的大小，因此1.5n/2n=0.75即扩容因子LOAD\_FACTORY
55. **return**;
56. }
57. //用CAS操作将sizeCtl-1，表明当前线程即将退出transfer或者作为最后一个线程即将处理收尾工作
58. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc = sizeCtl, sc - 1)) {
59. //参与扩容的线程未全部结束时，其他线程完成任务后的处理逻辑
60. //addCount方法中，在第一个线程执行transfer方法时会将sizeCtl赋值为(resizeStamp(n) << RESIZE\_STAMP\_SHIFT)+2，因此这里用一个相同的值来判断是否还有其他线程存在，与sizeCtl字段注释的说明不一致
61. **if** ((sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT)
62. **return**;
63. //只有最后一个线程才会执行后续的检查以及收尾工作
64. finishing = advance = **true**;
65. i = n; // recheck before commit
66. }
67. }
68. //当前槽位中没有元素，将fwd(ForwardingNode)作为该槽位中的元素，为了避免其他线程进行put操作插入到这个槽位中，所以即便是空槽也得放上占位节点fwd
69. **else** **if** ((f = tabAt(tab, i)) == null)
70. advance = casTabAt(tab, i, null, fwd);
71. //若当前节点标记为MOVED(放的是ForwardingNode)，则说明已经迁移完毕，将advice设为true，然后通过上面的while循环来递减i
72. //小问题：既然分片操作是通过CAS串行化的，那么为什么会出现以经迁移过的节点???
73. **else** **if** ((fh = f.hash) == MOVED)
74. advance = **true**; // already processed
75. //否则需要对该槽位(i)进行迁移操作
76. **else** {
77. //在锁定的情况下对该槽位进行迁移操作，putVal方法中也有synchronized方法块，将一个槽位的插入或者迁移过程串行化
78. synchronized (f) {
79. **if** (tabAt(tab, i) == f) {
80. Node<K,V> ln, hn;
81. //f是一个Node节点
82. **if** (fh >= 0) {
83. //n是一个2的幂次，只有最高位(对于32位的int来说，记为第X位)是1，因此得到的runBit要么是0要么是n，通过这种方式将一个链表中的元素划分为类，一类第X位为0，一类第X位为1
84. **int** runBit = fh & n;
85. Node<K,V> lastRun = f;
86. //遍历这个链表
87. **for** (Node<K,V> p = f.next; p != null; p = p.next) {
88. **int** b = p.hash & n;
89. //这里为什么要替换runBit的值：出于效率的考虑，如果某个节点开始往后都是同一类，那么最后lastRun就是这最后一串的起始节点，在下面的循环中将不再遍历这些节点，提高效率，真他妈煞费苦心
90. //另外，槽位由(table.length-1)&hash表达式确定，表扩张后，容量是2倍，1111&hash -> 11111&hash，因此结果取决于第4位，也就是原表的最高位，因此根据最高位的划分，可以将原表中某槽位的节点分为两类放入新表的两个槽位中
91. **if** (b != runBit) {
92. runBit = b;
93. lastRun = p;
94. }
95. }
96. **if** (runBit == 0) {
97. ln = lastRun;
98. hn = null;
99. }
100. **else** {
101. hn = lastRun;
102. ln = null;
103. }
104. //从f到lastRun遍历这些节点，按照第X位0或1进行分类
105. **for** (Node<K,V> p = f; p != lastRun; p = p.next) {
106. **int** ph = p.hash; K pk = p.key; V pv = p.val;
107. **if** ((ph & n) == 0)
108. ln = **new** Node<K,V>(ph, pk, pv, ln);
109. **else**
110. hn = **new** Node<K,V>(ph, pk, pv, hn);
111. }
112. //为什么需要CAS操作???
113. setTabAt(nextTab, i, ln);
114. setTabAt(nextTab, i + n, hn);
115. setTabAt(tab, i, fwd);
116. advance = **true**;
117. }
118. //f是一个TreeBin节点，即槽位中放的是一棵红黑树
119. **else** **if** (f instanceof TreeBin) {
120. TreeBin<K,V> t = (TreeBin<K,V>)f;
121. TreeNode<K,V> lo = null, loTail = null;
122. TreeNode<K,V> hi = null, hiTail = null;
123. **int** lc = 0, hc = 0;
124. **for** (Node<K,V> e = t.first; e != null; e = e.next) {
125. **int** h = e.hash;
126. TreeNode<K,V> p = **new** TreeNode<K,V>
127. (h, e.key, e.val, null, null);
128. //一样的套路，依据第X位是0或1，将节点分为两类
129. **if** ((h & n) == 0) {
130. **if** ((p.prev = loTail) == null)
131. lo = p;
132. **else**
133. loTail.next = p;
134. loTail = p;
135. ++lc;
136. }
137. **else** {
138. **if** ((p.prev = hiTail) == null)
139. hi = p;
140. **else**
141. hiTail.next = p;
142. hiTail = p;
143. ++hc;
144. }
145. }
146. //当ln节点的数量到达转为链表的阈值时，将红黑树转为链表
147. ln = (lc <= UNTREEIFY\_THRESHOLD) ? untreeify(lo) :
148. //如果hc为0，说明所有节点都是同一类，红黑树不需要重新构造
149. (hc != 0) ? **new** TreeBin<K,V>(lo) : t;
150. hn = (hc <= UNTREEIFY\_THRESHOLD) ? untreeify(hi) :
151. //如果hc为0，说明所有节点都是同一类，红黑树不需要重新构造
152. (lc != 0) ? **new** TreeBin<K,V>(hi) : t;
153. //为什么需要CAS操作???
154. setTabAt(nextTab, i, ln);
155. setTabAt(nextTab, i + n, hn);
156. setTabAt(tab, i, fwd);
157. advance = **true**;
158. }
159. }
160. }
161. }
162. }
163. }

#### helpTransfer

1. /\*\*
2. \* Helps transfer if a resize is in progress.
3. \*/
4. final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
5. Node<K,V>[] nextTab; **int** sc;
6. **if** (tab != null && (f instanceof ForwardingNode) &&
7. (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
8. **int** rs = resizeStamp(tab.length);
9. **while** (nextTab == nextTable && table == tab &&
10. (sc = sizeCtl) < 0) {
11. //在addCount方法中也有下面这些判断条件(可能是一些异常情况吧)
12. **if** ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||
13. sc == rs + MAX\_RESIZERS || transferIndex <= 0)
14. **break**;
15. //当前线程将参与到并发扩容的过程中去
16. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, sc + 1)) {
17. transfer(tab, nextTab);
18. **break**;
19. }
20. }
21. **return** nextTab;
22. }
23. **return** table;
24. }

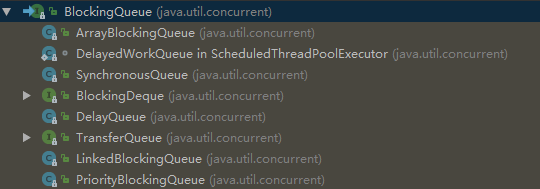
### get

1. /\*\*
2. \* Returns the value to which the specified key is mapped,
3. \* or {@code null} if this map contains no mapping for the key.
4. \*
5. \* <p>More formally, if this map contains a mapping from a key
6. \* {@code k} to a value {@code v} such that {@code key.equals(k)},
7. \* then this method returns {@code v}; otherwise it returns
8. \* {@code null}.  (There can be at most one such mapping.)
9. \*
10. \* @throws NullPointerException if the specified key is null
11. \*/
12. **public** V get(Object key) {
13. Node<K,V>[] tab; Node<K,V> e, p; **int** n, eh; K ek;
14. **int** h = spread(key.hashCode());
15. **if** ((tab = table) != null && (n = tab.length) > 0 &&
16. (e = tabAt(tab, (n - 1) & h)) != null) {
17. **if** ((eh = e.hash) == h) {
18. **if** ((ek = e.key) == key || (ek != null && key.equals(ek)))
19. **return** e.val;
20. }
21. //当eh小于0，代表当前节点转移到新表中了或者当前节点是TreeBin节点，若是TreeBin节点，那么会调用TreeBin的find方法，若为ForwardingNode节点，则会调用ForwardingNode的find方法在新表中查找
22. **else** **if** (eh < 0)
23. **return** (p = e.find(h, key)) != null ? p.val : null;
24. **while** ((e = e.next) != null) {
25. **if** (e.hash == h &&
26. ((ek = e.key) == key || (ek != null && key.equals(ek))))
27. **return** e.val;
28. }
29. }
30. **return** null;
31. }

# ThreadLocal

# ArrayBlockingQueue

## 继承体系



### BlockingQueue接口

1. **public** **interface** BlockingQueue<E> **extends** Queue<E> {
2. /\*\*
3. \* Inserts the specified element into this queue if it is possible to do
4. \* so immediately without violating capacity restrictions, returning
5. \* {@code true} upon success and throwing an
6. \* {@code IllegalStateException} if no space is currently available.
7. \* When using a capacity-restricted queue, it is generally preferable to
8. \* use {@link #offer(Object) offer}.
9. \*
10. \* @param e the element to add
11. \* @return {@code true} (as specified by {@link Collection#add})
12. \* @throws IllegalStateException if the element cannot be added at this
13. \*         time due to capacity restrictions
14. \* @throws ClassCastException if the class of the specified element
15. \*         prevents it from being added to this queue
16. \* @throws NullPointerException if the specified element is null
17. \* @throws IllegalArgumentException if some property of the specified
18. \*         element prevents it from being added to this queue
19. \*/
20. **boolean** add(E e);
22. /\*\*
23. \* Inserts the specified element into this queue if it is possible to do
24. \* so immediately without violating capacity restrictions, returning
25. \* {@code true} upon success and {@code false} if no space is currently
26. \* available.  When using a capacity-restricted queue, this method is
27. \* generally preferable to {@link #add}, which can fail to insert an
28. \* element only by throwing an exception.
29. \*
30. \* @param e the element to add
31. \* @return {@code true} if the element was added to this queue, else
32. \*         {@code false}
33. \* @throws ClassCastException if the class of the specified element
34. \*         prevents it from being added to this queue
35. \* @throws NullPointerException if the specified element is null
36. \* @throws IllegalArgumentException if some property of the specified
37. \*         element prevents it from being added to this queue
38. \*/
39. **boolean** offer(E e);
41. /\*\*
42. \* Inserts the specified element into this queue, waiting if necessary
43. \* for space to become available.
44. \*
45. \* @param e the element to add
46. \* @throws InterruptedException if interrupted while waiting
47. \* @throws ClassCastException if the class of the specified element
48. \*         prevents it from being added to this queue
49. \* @throws NullPointerException if the specified element is null
50. \* @throws IllegalArgumentException if some property of the specified
51. \*         element prevents it from being added to this queue
52. \*/
53. **void** put(E e) **throws** InterruptedException;
55. /\*\*
56. \* Inserts the specified element into this queue, waiting up to the
57. \* specified wait time if necessary for space to become available.
58. \*
59. \* @param e the element to add
60. \* @param timeout how long to wait before giving up, in units of
61. \*        {@code unit}
62. \* @param unit a {@code TimeUnit} determining how to interpret the
63. \*        {@code timeout} parameter
64. \* @return {@code true} if successful, or {@code false} if
65. \*         the specified waiting time elapses before space is available
66. \* @throws InterruptedException if interrupted while waiting
67. \* @throws ClassCastException if the class of the specified element
68. \*         prevents it from being added to this queue
69. \* @throws NullPointerException if the specified element is null
70. \* @throws IllegalArgumentException if some property of the specified
71. \*         element prevents it from being added to this queue
72. \*/
73. **boolean** offer(E e, **long** timeout, TimeUnit unit)
74. **throws** InterruptedException;
76. /\*\*
77. \* Retrieves and removes the head of this queue, waiting if necessary
78. \* until an element becomes available.
79. \*
80. \* @return the head of this queue
81. \* @throws InterruptedException if interrupted while waiting
82. \*/
83. E take() **throws** InterruptedException;
85. /\*\*
86. \* Retrieves and removes the head of this queue, waiting up to the
87. \* specified wait time if necessary for an element to become available.
88. \*
89. \* @param timeout how long to wait before giving up, in units of
90. \*        {@code unit}
91. \* @param unit a {@code TimeUnit} determining how to interpret the
92. \*        {@code timeout} parameter
93. \* @return the head of this queue, or {@code null} if the
94. \*         specified waiting time elapses before an element is available
95. \* @throws InterruptedException if interrupted while waiting
96. \*/
97. E poll(**long** timeout, TimeUnit unit)
98. **throws** InterruptedException;
100. /\*\*
101. \* Returns the number of additional elements that this queue can ideally
102. \* (in the absence of memory or resource constraints) accept without
103. \* blocking, or {@code Integer.MAX\_VALUE} if there is no intrinsic
104. \* limit.
105. \*
106. \* <p>Note that you <em>cannot</em> always tell if an attempt to insert
107. \* an element will succeed by inspecting {@code remainingCapacity}
108. \* because it may be the case that another thread is about to
109. \* insert or remove an element.
110. \*
111. \* @return the remaining capacity
112. \*/
113. **int** remainingCapacity();
115. /\*\*
116. \* Removes a single instance of the specified element from this queue,
117. \* if it is present.  More formally, removes an element {@code e} such
118. \* that {@code o.equals(e)}, if this queue contains one or more such
119. \* elements.
120. \* Returns {@code true} if this queue contained the specified element
121. \* (or equivalently, if this queue changed as a result of the call).
122. \*
123. \* @param o element to be removed from this queue, if present
124. \* @return {@code true} if this queue changed as a result of the call
125. \* @throws ClassCastException if the class of the specified element
126. \*         is incompatible with this queue
127. \*         (<a href="../Collection.html#optional-restrictions">optional</a>)
128. \* @throws NullPointerException if the specified element is null
129. \*         (<a href="../Collection.html#optional-restrictions">optional</a>)
130. \*/
131. **boolean** remove(Object o);
133. /\*\*
134. \* Returns {@code true} if this queue contains the specified element.
135. \* More formally, returns {@code true} if and only if this queue contains
136. \* at least one element {@code e} such that {@code o.equals(e)}.
137. \*
138. \* @param o object to be checked for containment in this queue
139. \* @return {@code true} if this queue contains the specified element
140. \* @throws ClassCastException if the class of the specified element
141. \*         is incompatible with this queue
142. \*         (<a href="../Collection.html#optional-restrictions">optional</a>)
143. \* @throws NullPointerException if the specified element is null
144. \*         (<a href="../Collection.html#optional-restrictions">optional</a>)
145. \*/
146. **public** **boolean** contains(Object o);
148. /\*\*
149. \* Removes all available elements from this queue and adds them
150. \* to the given collection.  This operation may be more
151. \* efficient than repeatedly polling this queue.  A failure
152. \* encountered while attempting to add elements to
153. \* collection {@code c} may result in elements being in neither,
154. \* either or both collections when the associated exception is
155. \* thrown.  Attempts to drain a queue to itself result in
156. \* {@code IllegalArgumentException}. Further, the behavior of
157. \* this operation is undefined if the specified collection is
158. \* modified while the operation is in progress.
159. \*
160. \* @param c the collection to transfer elements into
161. \* @return the number of elements transferred
162. \* @throws UnsupportedOperationException if addition of elements
163. \*         is not supported by the specified collection
164. \* @throws ClassCastException if the class of an element of this queue
165. \*         prevents it from being added to the specified collection
166. \* @throws NullPointerException if the specified collection is null
167. \* @throws IllegalArgumentException if the specified collection is this
168. \*         queue, or some property of an element of this queue prevents
169. \*         it from being added to the specified collection
170. \*/
171. **int** drainTo(Collection<? **super** E> c);
173. /\*\*
174. \* Removes at most the given number of available elements from
175. \* this queue and adds them to the given collection.  A failure
176. \* encountered while attempting to add elements to
177. \* collection {@code c} may result in elements being in neither,
178. \* either or both collections when the associated exception is
179. \* thrown.  Attempts to drain a queue to itself result in
180. \* {@code IllegalArgumentException}. Further, the behavior of
181. \* this operation is undefined if the specified collection is
182. \* modified while the operation is in progress.
183. \*
184. \* @param c the collection to transfer elements into
185. \* @param maxElements the maximum number of elements to transfer
186. \* @return the number of elements transferred
187. \* @throws UnsupportedOperationException if addition of elements
188. \*         is not supported by the specified collection
189. \* @throws ClassCastException if the class of an element of this queue
190. \*         prevents it from being added to the specified collection
191. \* @throws NullPointerException if the specified collection is null
192. \* @throws IllegalArgumentException if the specified collection is this
193. \*         queue, or some property of an element of this queue prevents
194. \*         it from being added to the specified collection
195. \*/
196. **int** drainTo(Collection<? **super** E> c, **int** maxElements);
197. }

### AbstractQueue

1. **public** **abstract** **class** AbstractQueue<E>
2. **extends** AbstractCollection<E>
3. **implements** Queue<E> {
5. /\*\*
6. \* Constructor for use by subclasses.
7. \*/
8. **protected** AbstractQueue() {
9. }
11. /\*\*
12. \* Inserts the specified element into this queue if it is possible to do so
13. \* immediately without violating capacity restrictions, returning
14. \* <tt>true</tt> upon success and throwing an <tt>IllegalStateException</tt>
15. \* if no space is currently available.
16. \*
17. \* <p>This implementation returns <tt>true</tt> if <tt>offer</tt> succeeds,
18. \* else throws an <tt>IllegalStateException</tt>.
19. \*
20. \* @param e the element to add
21. \* @return <tt>true</tt> (as specified by {@link Collection#add})
22. \* @throws IllegalStateException if the element cannot be added at this
23. \*         time due to capacity restrictions
24. \* @throws ClassCastException if the class of the specified element
25. \*         prevents it from being added to this queue
26. \* @throws NullPointerException if the specified element is null and
27. \*         this queue does not permit null elements
28. \* @throws IllegalArgumentException if some property of this element
29. \*         prevents it from being added to this queue
30. \*/
31. **public** **boolean** add(E e) {
32. **if** (offer(e))
33. **return** **true**;
34. **else**
35. **throw** **new** IllegalStateException("Queue full");
36. }
38. /\*\*
39. \* Retrieves and removes the head of this queue.  This method differs
40. \* from {@link #poll poll} only in that it throws an exception if this
41. \* queue is empty.
42. \*
43. \* <p>This implementation returns the result of <tt>poll</tt>
44. \* unless the queue is empty.
45. \*
46. \* @return the head of this queue
47. \* @throws NoSuchElementException if this queue is empty
48. \*/
49. **public** E remove() {
50. E x = poll();
51. **if** (x != **null**)
52. **return** x;
53. **else**
54. **throw** **new** NoSuchElementException();
55. }
57. /\*\*
58. \* Retrieves, but does not remove, the head of this queue.  This method
59. \* differs from {@link #peek peek} only in that it throws an exception if
60. \* this queue is empty.
61. \*
62. \* <p>This implementation returns the result of <tt>peek</tt>
63. \* unless the queue is empty.
64. \*
65. \* @return the head of this queue
66. \* @throws NoSuchElementException if this queue is empty
67. \*/
68. **public** E element() {
69. E x = peek();
70. **if** (x != **null**)
71. **return** x;
72. **else**
73. **throw** **new** NoSuchElementException();
74. }
76. /\*\*
77. \* Removes all of the elements from this queue.
78. \* The queue will be empty after this call returns.
79. \*
80. \* <p>This implementation repeatedly invokes {@link #poll poll} until it
81. \* returns <tt>null</tt>.
82. \*/
83. **public** **void** clear() {
84. **while** (poll() != **null**)
85. ;
86. }
88. /\*\*
89. \* Adds all of the elements in the specified collection to this
90. \* queue.  Attempts to addAll of a queue to itself result in
91. \* <tt>IllegalArgumentException</tt>. Further, the behavior of
92. \* this operation is undefined if the specified collection is
93. \* modified while the operation is in progress.
94. \*
95. \* <p>This implementation iterates over the specified collection,
96. \* and adds each element returned by the iterator to this
97. \* queue, in turn.  A runtime exception encountered while
98. \* trying to add an element (including, in particular, a
99. \* <tt>null</tt> element) may result in only some of the elements
100. \* having been successfully added when the associated exception is
101. \* thrown.
102. \*
103. \* @param c collection containing elements to be added to this queue
104. \* @return <tt>true</tt> if this queue changed as a result of the call
105. \* @throws ClassCastException if the class of an element of the specified
106. \*         collection prevents it from being added to this queue
107. \* @throws NullPointerException if the specified collection contains a
108. \*         null element and this queue does not permit null elements,
109. \*         or if the specified collection is null
110. \* @throws IllegalArgumentException if some property of an element of the
111. \*         specified collection prevents it from being added to this
112. \*         queue, or if the specified collection is this queue
113. \* @throws IllegalStateException if not all the elements can be added at
114. \*         this time due to insertion restrictions
115. \* @see #add(Object)
116. \*/
117. **public** **boolean** addAll(Collection<? **extends** E> c) {
118. **if** (c == **null**)
119. **throw** **new** NullPointerException();
120. **if** (c == **this**)
121. **throw** **new** IllegalArgumentException();
122. **boolean** modified = **false**;
123. **for** (E e : c)
124. **if** (add(e))
125. modified = **true**;
126. **return** modified;
127. }
129. }

## 字段

### items

1. /\*\* The queued items \*/
2. **final** Object[] items;

1、队列中的元素，即一个数组，数组大小在初始化后就固定了，不会扩容，也就是说队列的最大长度是固定的

### takeIndex

1. /\*\* items index for next take, poll, peek or remove \*/
2. **int** takeIndex;

1、队列头索引

### putIndex

1. /\*\* items index for next put, offer, or add \*/
2. **int** putIndex;

2、队列尾索引

### count

1. /\*\* Number of elements in the queue \*/
2. **int** count;

1、队列中元素的个数，size()方法的返回值

### lock

1. /\*\* Main lock guarding all access \*/
2. **final** ReentrantLock lock;

1、利用AQS框架实现的锁，是否可冲入依赖于构造函数。

2、大部分对于队列的操作都会使用该锁进行锁定

### notEmpty

1. /\*\* Condition for waiting takes \*/
2. **private** **final** Condition notEmpty;

1、条件变量

### notFull

1. /\*\* Condition for waiting puts \*/
2. **private** **final** Condition notFull;

1、条件变量

### itrs

1. /\*\*
2. \* Shared state for currently active iterators, or null if there
3. \* are known not to be any.  Allows queue operations to update
4. \* iterator state.
5. \*/
6. **transient** Itrs itrs = **null**;

1、迭代器

## 内部类(未完成)

## 方法

### 构造方法

1. /\*\*
2. \* Creates an {@code ArrayBlockingQueue} with the given (fixed)
3. \* capacity and the specified access policy.
4. \*
5. \* @param capacity the capacity of this queue
6. \* @param fair if {@code true} then queue accesses for threads blocked
7. \*        on insertion or removal, are processed in FIFO order;
8. \*        if {@code false} the access order is unspecified.
9. \* @throws IllegalArgumentException if {@code capacity < 1}
10. \*/
11. **public** ArrayBlockingQueue(**int** capacity, **boolean** fair) {
12. **if** (capacity <= 0)
13. **throw** **new** IllegalArgumentException();
14. **this**.items = **new** Object[capacity];
15. lock = **new** ReentrantLock(fair);
16. notEmpty = lock.newCondition();
17. notFull =  lock.newCondition();
18. }

### offer

1. /\*\*
2. \* Inserts the specified element at the tail of this queue if it is
3. \* possible to do so immediately without exceeding the queue's capacity,
4. \* returning {@code true} upon success and {@code false} if this queue
5. \* is full.  This method is generally preferable to method {@link #add},
6. \* which can fail to insert an element only by throwing an exception.
7. \*
8. \* @throws NullPointerException if the specified element is null
9. \*/
10. **public** **boolean** offer(E e) {
11. //入队的元素必须非空
12. checkNotNull(e);
13. **final** ReentrantLock lock = **this**.lock;
14. //enqueue操作非线程安全，因此必须加锁
15. lock.lock();
16. **try** {
17. //队列元素已满，直接返回false
18. **if** (count == items.length)
19. **return** **false**;
20. **else** {
21. //队列尚未满，执行入队操作
22. enqueue(e);
23. **return** **true**;
24. }
25. } **finally** {
26. lock.unlock();
27. }
28. }

#### checkNotNull

1. /\*\*
2. \* Throws NullPointerException if argument is null.
3. \*
4. \* @param v the element
5. \*/
6. **private** **static** **void** checkNotNull(Object v) {
7. **if** (v == **null**)
8. **throw** **new** NullPointerException();
9. }

#### enqueue

1. /\*\*
2. \* Inserts element at current put position, advances, and signals.
3. \* Call only when holding lock.
4. \*/
5. **private** **void** enqueue(E x) {
6. // assert lock.getHoldCount() == 1;
7. // assert items[putIndex] == null;
8. **final** Object[] items = **this**.items;
9. items[putIndex] = x;
10. **if** (++putIndex == items.length)
11. //这里将putIndex置为0有什么用途：由于是数组，因此有一个循环的特点
12. putIndex = 0;
13. count++;
14. //???有待于研究ReentrantLock
15. notEmpty.signal();
16. }

### offer(timeout)

1. /\*\*
2. \* Inserts the specified element at the tail of this queue, waiting
3. \* up to the specified wait time for space to become available if
4. \* the queue is full.
5. \*
6. \* @throws InterruptedException {@inheritDoc}
7. \* @throws NullPointerException {@inheritDoc}
8. \*/
9. **public** **boolean** offer(E e, **long** timeout, TimeUnit unit)
10. **throws** InterruptedException {
11. //入队元素必须非空
12. checkNotNull(e);
13. **long** nanos = unit.toNanos(timeout);
14. **final** ReentrantLock lock = **this**.lock;
15. //有待于研究ReentrantLock???
16. lock.lockInterruptibly();
17. **try** {
18. //当队列已满
19. **while** (count == items.length) {
20. //剩余等待时间小于0，则直接返回false
21. **if** (nanos <= 0)
22. **return** **false**;
23. //进行阻塞等待???此时lock是否会释放???好像是一定要释放的
24. nanos = notFull.awaitNanos(nanos);
25. }
26. //当队列未满，入队即可
27. enqueue(e);
28. **return** **true**;
29. } **finally** {
30. lock.unlock();
31. }
32. }

### put

1. /\*\*
2. \* Inserts the specified element at the tail of this queue, waiting
3. \* for space to become available if the queue is full.
4. \*
5. \* @throws InterruptedException {@inheritDoc}
6. \* @throws NullPointerException {@inheritDoc}
7. \*/
8. **public** **void** put(E e) **throws** InterruptedException {
9. //检查元素非空
10. checkNotNull(e);
11. **final** ReentrantLock lock = **this**.lock;
12. //有待于研究ReentrantLock???
13. lock.lockInterruptibly();
14. **try** {
15. **while** (count == items.length)
16. //阻塞直到队列不再满，或者被中断，此函数之后感觉锁会被释放
17. notFull.await();
18. //入队操作
19. enqueue(e);
20. } **finally** {
21. lock.unlock();
22. }
23. }

### poll

1. **public** E poll() {
2. **final** ReentrantLock lock = **this**.lock;
3. lock.lock();
4. **try** {
5. //若队列为空，则返回null，否则返回队列头元素
6. **return** (count == 0) ? **null** : dequeue();
7. } **finally** {
8. lock.unlock();
9. }
10. }

#### dequeue

1. /\*\*
2. \* Extracts element at current take position, advances, and signals.
3. \* Call only when holding lock.
4. \*/
5. **private** E dequeue() {
6. // assert lock.getHoldCount() == 1;
7. // assert items[takeIndex] != null;
8. **final** Object[] items = **this**.items;
9. @SuppressWarnings("unchecked")
10. E x = (E) items[takeIndex];
11. items[takeIndex] = **null**;
12. **if** (++takeIndex == items.length)
13. //这里将takeIndex赋值为0有什么意图：由于是数组，因此有一个循环的特点
14. takeIndex = 0;
15. count--;
16. **if** (itrs != **null**)
17. itrs.elementDequeued();
18. notFull.signal();
19. **return** x;
20. }

### poll(timeout)

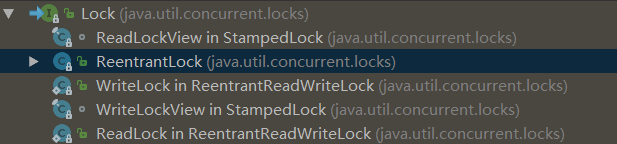
1. **public** E poll(**long** timeout, TimeUnit unit) **throws** InterruptedException {
2. **long** nanos = unit.toNanos(timeout);
3. **final** ReentrantLock lock = **this**.lock;
4. //有待于研究ReentrantLock???
5. lock.lockInterruptibly();
6. **try** {
7. //队列为空
8. **while** (count == 0) {
9. //剩余等待时间小于0
10. **if** (nanos <= 0)
11. **return** **null**;
12. //在条件量上进行等待，应该会释放锁???
13. nanos = notEmpty.awaitNanos(nanos);
14. }
15. //将队列头元素出队
16. **return** dequeue();
17. } **finally** {
18. lock.unlock();
19. }
20. }

### take

1. **public** E take() **throws** InterruptedException {
2. **final** ReentrantLock lock = **this**.lock;
3. //有待于研究ReentrantLock???
4. lock.lockInterruptibly();
5. **try** {
6. //队列为空
7. **while** (count == 0)
8. //在条件变量上进行等待，应该会释放锁???
9. notEmpty.await();
10. **return** dequeue();
11. } **finally** {
12. lock.unlock();
13. }
14. }

# ReentrantLock

## 继承体系



### Lock接口

1. **public** **interface** Lock {
3. /\*\*
4. \* Acquires the lock.
5. \*
6. \* <p>If the lock is not available then the current thread becomes
7. \* disabled for thread scheduling purposes and lies dormant until the
8. \* lock has been acquired.
9. \*
10. \* <p><b>Implementation Considerations</b>
11. \*
12. \* <p>A {@code Lock} implementation may be able to detect erroneous use
13. \* of the lock, such as an invocation that would cause deadlock, and
14. \* may throw an (unchecked) exception in such circumstances.  The
15. \* circumstances and the exception type must be documented by that
16. \* {@code Lock} implementation.
17. \*/
18. **void** lock();
20. /\*\*
21. \* Acquires the lock unless the current thread is
22. \* {@linkplain Thread#interrupt interrupted}.
23. \*
24. \* <p>Acquires the lock if it is available and returns immediately.
25. \*
26. \* <p>If the lock is not available then the current thread becomes
27. \* disabled for thread scheduling purposes and lies dormant until
28. \* one of two things happens:
29. \*
30. \* <ul>
31. \* <li>The lock is acquired by the current thread; or
32. \* <li>Some other thread {@linkplain Thread#interrupt interrupts} the
33. \* current thread, and interruption of lock acquisition is supported.
34. \* </ul>
35. \*
36. \* <p>If the current thread:
37. \* <ul>
38. \* <li>has its interrupted status set on entry to this method; or
39. \* <li>is {@linkplain Thread#interrupt interrupted} while acquiring the
40. \* lock, and interruption of lock acquisition is supported,
41. \* </ul>
42. \* then {@link InterruptedException} is thrown and the current thread's
43. \* interrupted status is cleared.
44. \*
45. \* <p><b>Implementation Considerations</b>
46. \*
47. \* <p>The ability to interrupt a lock acquisition in some
48. \* implementations may not be possible, and if possible may be an
49. \* expensive operation.  The programmer should be aware that this
50. \* may be the case. An implementation should document when this is
51. \* the case.
52. \*
53. \* <p>An implementation can favor responding to an interrupt over
54. \* normal method return.
55. \*
56. \* <p>A {@code Lock} implementation may be able to detect
57. \* erroneous use of the lock, such as an invocation that would
58. \* cause deadlock, and may throw an (unchecked) exception in such
59. \* circumstances.  The circumstances and the exception type must
60. \* be documented by that {@code Lock} implementation.
61. \*
62. \* @throws InterruptedException if the current thread is
63. \*         interrupted while acquiring the lock (and interruption
64. \*         of lock acquisition is supported)
65. \*/
66. **void** lockInterruptibly() **throws** InterruptedException;
68. /\*\*
69. \* Acquires the lock only if it is free at the time of invocation.
70. \*
71. \* <p>Acquires the lock if it is available and returns immediately
72. \* with the value {@code true}.
73. \* If the lock is not available then this method will return
74. \* immediately with the value {@code false}.
75. \*
76. \* <p>A typical usage idiom for this method would be:
77. \*  <pre> {@code
78. \* Lock lock = ...;
79. \* if (lock.tryLock()) {
80. \*   try {
81. \*     // manipulate protected state
82. \*   } finally {
83. \*     lock.unlock();
84. \*   }
85. \* } else {
86. \*   // perform alternative actions
87. \* }}</pre>
88. \*
89. \* This usage ensures that the lock is unlocked if it was acquired, and
90. \* doesn't try to unlock if the lock was not acquired.
91. \*
92. \* @return {@code true} if the lock was acquired and
93. \*         {@code false} otherwise
94. \*/
95. **boolean** tryLock();
97. /\*\*
98. \* Acquires the lock if it is free within the given waiting time and the
99. \* current thread has not been {@linkplain Thread#interrupt interrupted}.
100. \*
101. \* <p>If the lock is available this method returns immediately
102. \* with the value {@code true}.
103. \* If the lock is not available then
104. \* the current thread becomes disabled for thread scheduling
105. \* purposes and lies dormant until one of three things happens:
106. \* <ul>
107. \* <li>The lock is acquired by the current thread; or
108. \* <li>Some other thread {@linkplain Thread#interrupt interrupts} the
109. \* current thread, and interruption of lock acquisition is supported; or
110. \* <li>The specified waiting time elapses
111. \* </ul>
112. \*
113. \* <p>If the lock is acquired then the value {@code true} is returned.
114. \*
115. \* <p>If the current thread:
116. \* <ul>
117. \* <li>has its interrupted status set on entry to this method; or
118. \* <li>is {@linkplain Thread#interrupt interrupted} while acquiring
119. \* the lock, and interruption of lock acquisition is supported,
120. \* </ul>
121. \* then {@link InterruptedException} is thrown and the current thread's
122. \* interrupted status is cleared.
123. \*
124. \* <p>If the specified waiting time elapses then the value {@code false}
125. \* is returned.
126. \* If the time is
127. \* less than or equal to zero, the method will not wait at all.
128. \*
129. \* <p><b>Implementation Considerations</b>
130. \*
131. \* <p>The ability to interrupt a lock acquisition in some implementations
132. \* may not be possible, and if possible may
133. \* be an expensive operation.
134. \* The programmer should be aware that this may be the case. An
135. \* implementation should document when this is the case.
136. \*
137. \* <p>An implementation can favor responding to an interrupt over normal
138. \* method return, or reporting a timeout.
139. \*
140. \* <p>A {@code Lock} implementation may be able to detect
141. \* erroneous use of the lock, such as an invocation that would cause
142. \* deadlock, and may throw an (unchecked) exception in such circumstances.
143. \* The circumstances and the exception type must be documented by that
144. \* {@code Lock} implementation.
145. \*
146. \* @param time the maximum time to wait for the lock
147. \* @param unit the time unit of the {@code time} argument
148. \* @return {@code true} if the lock was acquired and {@code false}
149. \*         if the waiting time elapsed before the lock was acquired
150. \*
151. \* @throws InterruptedException if the current thread is interrupted
152. \*         while acquiring the lock (and interruption of lock
153. \*         acquisition is supported)
154. \*/
155. **boolean** tryLock(**long** time, TimeUnit unit) **throws** InterruptedException;
157. /\*\*
158. \* Releases the lock.
159. \*
160. \* <p><b>Implementation Considerations</b>
161. \*
162. \* <p>A {@code Lock} implementation will usually impose
163. \* restrictions on which thread can release a lock (typically only the
164. \* holder of the lock can release it) and may throw
165. \* an (unchecked) exception if the restriction is violated.
166. \* Any restrictions and the exception
167. \* type must be documented by that {@code Lock} implementation.
168. \*/
169. **void** unlock();
171. /\*\*
172. \* Returns a new {@link Condition} instance that is bound to this
173. \* {@code Lock} instance.
174. \*
175. \* <p>Before waiting on the condition the lock must be held by the
176. \* current thread.
177. \* A call to {@link Condition#await()} will atomically release the lock
178. \* before waiting and re-acquire the lock before the wait returns.
179. \*
180. \* <p><b>Implementation Considerations</b>
181. \*
182. \* <p>The exact operation of the {@link Condition} instance depends on
183. \* the {@code Lock} implementation and must be documented by that
184. \* implementation.
185. \*
186. \* @return A new {@link Condition} instance for this {@code Lock} instance
187. \* @throws UnsupportedOperationException if this {@code Lock}
188. \*         implementation does not support conditions
189. \*/
190. Condition newCondition();
191. }

## 字段

### sync

1. /\*\* Synchronizer providing all implementation mechanics \*/
2. **private** **final** Sync sync;

1、持有的继承自AQS的同步队列对象，锁操作主要是基于对这个对象的调用

## 内部类

### Sync

1. **abstract** **static** **class** Sync **extends** AbstractQueuedSynchronizer {
2. **private** **static** **final** **long** serialVersionUID = -5179523762034025860L;
4. /\*\*
5. \* Performs {@link Lock#lock}. The main reason for subclassing
6. \* is to allow fast path for nonfair version.
7. \*/
8. **abstract** **void** lock();
10. /\*\*
11. \* Performs non-fair tryLock.  tryAcquire is implemented in
12. \* subclasses, but both need nonfair try for trylock method.
13. \*/
14. **final** **boolean** nonfairTryAcquire(**int** acquires) {
15. **final** Thread current = Thread.currentThread();
16. **int** c = getState();
17. //c为资源状态，必须是0才能够获取，注意不是AQS#Node节点的状态!!!
18. **if** (c == 0) {
19. //获取资源成功，那么将锁关联当前线程
20. **if** (compareAndSetState(0, acquires)) {
21. setExclusiveOwnerThread(current);
22. **return** **true**;
23. }
24. }
25. //如果此时是重入的情况
26. **else** **if** (current == getExclusiveOwnerThread()) {
27. **int** nextc = c + acquires;
28. **if** (nextc < 0) // overflow
29. **throw** **new** Error("Maximum lock count exceeded");
30. //更新资源状态，不需要CAS操作，因为当前线程已经独占资源，不会有冲突
31. setState(nextc);
32. **return** **true**;
33. }
34. **return** **false**;
35. }
37. **protected** **final** **boolean** tryRelease(**int** releases) {
38. **int** c = getState() - releases;
39. //当前执行tryRelease的线程不是锁关联的线程，这是不可能的
40. **if** (Thread.currentThread() != getExclusiveOwnerThread())
41. **throw** **new** IllegalMonitorStateException();
42. **boolean** free = **false**;
43. //如果剩余的资源是0，那么意味着锁可以被释放
44. **if** (c == 0) {
45. free = **true**;
46. setExclusiveOwnerThread(**null**);
47. }
48. //设置资源状况，也不用CAS操作
49. setState(c);
50. **return** free;
51. }
53. **protected** **final** **boolean** isHeldExclusively() {
54. // While we must in general read state before owner,
55. // we don't need to do so to check if current thread is owner
56. **return** getExclusiveOwnerThread() == Thread.currentThread();
57. }
59. **final** ConditionObject newCondition() {
60. **return** **new** ConditionObject();
61. }
63. // Methods relayed from outer class
65. **final** Thread getOwner() {
66. **return** getState() == 0 ? **null** : getExclusiveOwnerThread();
67. }
69. **final** **int** getHoldCount() {
70. **return** isHeldExclusively() ? getState() : 0;
71. }
73. **final** **boolean** isLocked() {
74. **return** getState() != 0;
75. }
77. /\*\*
78. \* Reconstitutes the instance from a stream (that is, deserializes it).
79. \*/
80. **private** **void** readObject(java.io.ObjectInputStream s)
81. **throws** java.io.IOException, ClassNotFoundException {
82. s.defaultReadObject();
83. setState(0); // reset to unlocked state
84. }
85. }

### NonfairSync

1. /\*\*
2. \* Sync object for non-fair locks
3. \*/
4. **static** **final** **class** NonfairSync **extends** Sync {
5. **private** **static** **final** **long** serialVersionUID = 7316153563782823691L;
7. /\*\*
8. \* Performs lock.  Try immediate barge, backing up to normal
9. \* acquire on failure.
10. \*/
11. **final** **void** lock() {
12. //当前状态只有是0并且竞争成功时，将当前线程绑定到该锁中
13. //意味着一旦state设置为1，必然走的是acquire(1)的流程
14. **if** (compareAndSetState(0, 1))
15. setExclusiveOwnerThread(Thread.currentThread());
16. **else**
17. acquire(1);
18. }
20. **protected** **final** **boolean** tryAcquire(**int** acquires) {
21. **return** nonfairTryAcquire(acquires);
22. }
23. }

### FairSync

1. /\*\*
2. \* Sync object for fair locks
3. \*/
4. **static** **final** **class** FairSync **extends** Sync {
5. **private** **static** **final** **long** serialVersionUID = -3000897897090466540L;
7. **final** **void** lock() {
8. acquire(1);
9. }
11. /\*\*
12. \* Fair version of tryAcquire.  Don't grant access unless
13. \* recursive call or no waiters or is first.
14. \*/
15. **protected** **final** **boolean** tryAcquire(**int** acquires) {
16. **final** Thread current = Thread.currentThread();
17. **int** c = getState();
18. **if** (c == 0) {
19. //首先检查是否有已经在等待的线程(不一定完全准确，因为依赖的是Node节点的next字段)，若没有，则通过CAS操作抢占资源，若成功了，则获取锁
20. **if** (!hasQueuedPredecessors() &&
21. compareAndSetState(0, acquires)) {
22. setExclusiveOwnerThread(current);
23. **return** **true**;
24. }
25. }
26. //这一部分操作就与NonfairSync一样了
27. **else** **if** (current == getExclusiveOwnerThread()) {
28. **int** nextc = c + acquires;
29. **if** (nextc < 0)
30. **throw** **new** Error("Maximum lock count exceeded");
31. setState(nextc);
32. **return** **true**;
33. }
34. **return** **false**;
35. }
36. }

## 方法

### 构造方法

1. /\*\*
2. \* Creates an instance of {@code ReentrantLock}.
3. \* This is equivalent to using {@code ReentrantLock(false)}.
4. \*/
5. **public** ReentrantLock() {
6. //默认非公平锁
7. sync = **new** NonfairSync();
8. }
10. /\*\*
11. \* Creates an instance of {@code ReentrantLock} with the
12. \* given fairness policy.
13. \*
14. \* @param fair {@code true} if this lock should use a fair ordering policy
15. \*/
16. **public** ReentrantLock(**boolean** fair) {
17. sync = fair ? **new** FairSync() : **new** NonfairSync();
18. }

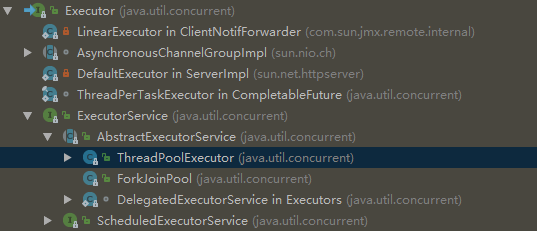
### 其他方法

1、其他方法都是转调用sync的相应方法，暂不分析了

# Condition

# ThreadPoolExecutor

## 继承体系



### Executor接口

1. **public** **interface** Executor {
3. /\*\*
4. \* Executes the given command at some time in the future.  The command
5. \* may execute in a new thread, in a pooled thread, or in the calling
6. \* thread, at the discretion of the {@code Executor} implementation.
7. \*
8. \* @param command the runnable task
9. \* @throws RejectedExecutionException if this task cannot be
10. \* accepted for execution
11. \* @throws NullPointerException if command is null
12. \*/
13. **void** execute(Runnable command);
14. }

1、该接口只有一个execute方法，执行一个Runnable任务，没有返回值

### ExecutorService

1. **public** **interface** ExecutorService **extends** Executor {
3. /\*\*
4. \* Initiates an orderly shutdown in which previously submitted
5. \* tasks are executed, but no new tasks will be accepted.
6. \* Invocation has no additional effect if already shut down.
7. \*
8. \* <p>This method does not wait for previously submitted tasks to
9. \* complete execution.  Use {@link #awaitTermination awaitTermination}
10. \* to do that.
11. \*
12. \* @throws SecurityException if a security manager exists and
13. \*         shutting down this ExecutorService may manipulate
14. \*         threads that the caller is not permitted to modify
15. \*         because it does not hold {@link
16. \*         java.lang.RuntimePermission}{@code ("modifyThread")},
17. \*         or the security manager's {@code checkAccess} method
18. \*         denies access.
19. \*/
20. **void** shutdown();
22. /\*\*
23. \* Attempts to stop all actively executing tasks, halts the
24. \* processing of waiting tasks, and returns a list of the tasks
25. \* that were awaiting execution.
26. \*
27. \* <p>This method does not wait for actively executing tasks to
28. \* terminate.  Use {@link #awaitTermination awaitTermination} to
29. \* do that.
30. \*
31. \* <p>There are no guarantees beyond best-effort attempts to stop
32. \* processing actively executing tasks.  For example, typical
33. \* implementations will cancel via {@link Thread#interrupt}, so any
34. \* task that fails to respond to interrupts may never terminate.
35. \*
36. \* @return list of tasks that never commenced execution
37. \* @throws SecurityException if a security manager exists and
38. \*         shutting down this ExecutorService may manipulate
39. \*         threads that the caller is not permitted to modify
40. \*         because it does not hold {@link
41. \*         java.lang.RuntimePermission}{@code ("modifyThread")},
42. \*         or the security manager's {@code checkAccess} method
43. \*         denies access.
44. \*/
45. List<Runnable> shutdownNow();
47. /\*\*
48. \* Returns {@code true} if this executor has been shut down.
49. \*
50. \* @return {@code true} if this executor has been shut down
51. \*/
52. **boolean** isShutdown();
54. /\*\*
55. \* Returns {@code true} if all tasks have completed following shut down.
56. \* Note that {@code isTerminated} is never {@code true} unless
57. \* either {@code shutdown} or {@code shutdownNow} was called first.
58. \*
59. \* @return {@code true} if all tasks have completed following shut down
60. \*/
61. **boolean** isTerminated();
63. /\*\*
64. \* Blocks until all tasks have completed execution after a shutdown
65. \* request, or the timeout occurs, or the current thread is
66. \* interrupted, whichever happens first.
67. \*
68. \* @param timeout the maximum time to wait
69. \* @param unit the time unit of the timeout argument
70. \* @return {@code true} if this executor terminated and
71. \*         {@code false} if the timeout elapsed before termination
72. \* @throws InterruptedException if interrupted while waiting
73. \*/
74. **boolean** awaitTermination(**long** timeout, TimeUnit unit)
75. **throws** InterruptedException;
77. /\*\*
78. \* Submits a value-returning task for execution and returns a
79. \* Future representing the pending results of the task. The
80. \* Future's {@code get} method will return the task's result upon
81. \* successful completion.
82. \*
83. \* <p>
84. \* If you would like to immediately block waiting
85. \* for a task, you can use constructions of the form
86. \* {@code result = exec.submit(aCallable).get();}
87. \*
88. \* <p>Note: The {@link Executors} class includes a set of methods
89. \* that can convert some other common closure-like objects,
90. \* for example, {@link java.security.PrivilegedAction} to
91. \* {@link Callable} form so they can be submitted.
92. \*
93. \* @param task the task to submit
94. \* @param <T> the type of the task's result
95. \* @return a Future representing pending completion of the task
96. \* @throws RejectedExecutionException if the task cannot be
97. \*         scheduled for execution
98. \* @throws NullPointerException if the task is null
99. \*/
100. <T> Future<T> submit(Callable<T> task);
102. /\*\*
103. \* Submits a Runnable task for execution and returns a Future
104. \* representing that task. The Future's {@code get} method will
105. \* return the given result upon successful completion.
106. \*
107. \* @param task the task to submit
108. \* @param result the result to return
109. \* @param <T> the type of the result
110. \* @return a Future representing pending completion of the task
111. \* @throws RejectedExecutionException if the task cannot be
112. \*         scheduled for execution
113. \* @throws NullPointerException if the task is null
114. \*/
115. <T> Future<T> submit(Runnable task, T result);
117. /\*\*
118. \* Submits a Runnable task for execution and returns a Future
119. \* representing that task. The Future's {@code get} method will
120. \* return {@code null} upon <em>successful</em> completion.
121. \*
122. \* @param task the task to submit
123. \* @return a Future representing pending completion of the task
124. \* @throws RejectedExecutionException if the task cannot be
125. \*         scheduled for execution
126. \* @throws NullPointerException if the task is null
127. \*/
128. Future<?> submit(Runnable task);
130. /\*\*
131. \* Executes the given tasks, returning a list of Futures holding
132. \* their status and results when all complete.
133. \* {@link Future#isDone} is {@code true} for each
134. \* element of the returned list.
135. \* Note that a <em>completed</em> task could have
136. \* terminated either normally or by throwing an exception.
137. \* The results of this method are undefined if the given
138. \* collection is modified while this operation is in progress.
139. \*
140. \* @param tasks the collection of tasks
141. \* @param <T> the type of the values returned from the tasks
142. \* @return a list of Futures representing the tasks, in the same
143. \*         sequential order as produced by the iterator for the
144. \*         given task list, each of which has completed
145. \* @throws InterruptedException if interrupted while waiting, in
146. \*         which case unfinished tasks are cancelled
147. \* @throws NullPointerException if tasks or any of its elements are {@code null}
148. \* @throws RejectedExecutionException if any task cannot be
149. \*         scheduled for execution
150. \*/
151. <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)
152. **throws** InterruptedException;
154. /\*\*
155. \* Executes the given tasks, returning a list of Futures holding
156. \* their status and results
157. \* when all complete or the timeout expires, whichever happens first.
158. \* {@link Future#isDone} is {@code true} for each
159. \* element of the returned list.
160. \* Upon return, tasks that have not completed are cancelled.
161. \* Note that a <em>completed</em> task could have
162. \* terminated either normally or by throwing an exception.
163. \* The results of this method are undefined if the given
164. \* collection is modified while this operation is in progress.
165. \*
166. \* @param tasks the collection of tasks
167. \* @param timeout the maximum time to wait
168. \* @param unit the time unit of the timeout argument
169. \* @param <T> the type of the values returned from the tasks
170. \* @return a list of Futures representing the tasks, in the same
171. \*         sequential order as produced by the iterator for the
172. \*         given task list. If the operation did not time out,
173. \*         each task will have completed. If it did time out, some
174. \*         of these tasks will not have completed.
175. \* @throws InterruptedException if interrupted while waiting, in
176. \*         which case unfinished tasks are cancelled
177. \* @throws NullPointerException if tasks, any of its elements, or
178. \*         unit are {@code null}
179. \* @throws RejectedExecutionException if any task cannot be scheduled
180. \*         for execution
181. \*/
182. <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,
183. **long** timeout, TimeUnit unit)
184. **throws** InterruptedException;
186. /\*\*
187. \* Executes the given tasks, returning the result
188. \* of one that has completed successfully (i.e., without throwing
189. \* an exception), if any do. Upon normal or exceptional return,
190. \* tasks that have not completed are cancelled.
191. \* The results of this method are undefined if the given
192. \* collection is modified while this operation is in progress.
193. \*
194. \* @param tasks the collection of tasks
195. \* @param <T> the type of the values returned from the tasks
196. \* @return the result returned by one of the tasks
197. \* @throws InterruptedException if interrupted while waiting
198. \* @throws NullPointerException if tasks or any element task
199. \*         subject to execution is {@code null}
200. \* @throws IllegalArgumentException if tasks is empty
201. \* @throws ExecutionException if no task successfully completes
202. \* @throws RejectedExecutionException if tasks cannot be scheduled
203. \*         for execution
204. \*/
205. <T> T invokeAny(Collection<? **extends** Callable<T>> tasks)
206. **throws** InterruptedException, ExecutionException;
208. /\*\*
209. \* Executes the given tasks, returning the result
210. \* of one that has completed successfully (i.e., without throwing
211. \* an exception), if any do before the given timeout elapses.
212. \* Upon normal or exceptional return, tasks that have not
213. \* completed are cancelled.
214. \* The results of this method are undefined if the given
215. \* collection is modified while this operation is in progress.
216. \*
217. \* @param tasks the collection of tasks
218. \* @param timeout the maximum time to wait
219. \* @param unit the time unit of the timeout argument
220. \* @param <T> the type of the values returned from the tasks
221. \* @return the result returned by one of the tasks
222. \* @throws InterruptedException if interrupted while waiting
223. \* @throws NullPointerException if tasks, or unit, or any element
224. \*         task subject to execution is {@code null}
225. \* @throws TimeoutException if the given timeout elapses before
226. \*         any task successfully completes
227. \* @throws ExecutionException if no task successfully completes
228. \* @throws RejectedExecutionException if tasks cannot be scheduled
229. \*         for execution
230. \*/
231. <T> T invokeAny(Collection<? **extends** Callable<T>> tasks,
232. **long** timeout, TimeUnit unit)
233. **throws** InterruptedException, ExecutionException, TimeoutException;
234. }

1、这个接口扩展了Executor接口，并且提供了submit等方法

### AbstractExecutorService

1. **public** **abstract** **class** AbstractExecutorService **implements** ExecutorService {
3. /\*\*
4. \* Returns a {@code RunnableFuture} for the given runnable and default
5. \* value.
6. \*
7. \* @param runnable the runnable task being wrapped
8. \* @param value the default value for the returned future
9. \* @param <T> the type of the given value
10. \* @return a {@code RunnableFuture} which, when run, will run the
11. \* underlying runnable and which, as a {@code Future}, will yield
12. \* the given value as its result and provide for cancellation of
13. \* the underlying task
14. \* @since 1.6
15. \*/
16. **protected** <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {   //将runnable封装成一个RunnableFuture
17. **return** **new** FutureTask<T>(runnable, value);
18. }
20. /\*\*
21. \* Returns a {@code RunnableFuture} for the given callable task.
22. \*
23. \* @param callable the callable task being wrapped
24. \* @param <T> the type of the callable's result
25. \* @return a {@code RunnableFuture} which, when run, will call the
26. \* underlying callable and which, as a {@code Future}, will yield
27. \* the callable's result as its result and provide for
28. \* cancellation of the underlying task
29. \* @since 1.6
30. \*/
31. **protected** <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
32. **return** **new** FutureTask<T>(callable);
33. }
35. /\*\*
36. \* @throws RejectedExecutionException {@inheritDoc}
37. \* @throws NullPointerException       {@inheritDoc}
38. \*/
39. **public** Future<?> submit(Runnable task) {
40. **if** (task == **null**) **throw** **new** NullPointerException();
41. //进行一层封装，可以持有这个任务
42. RunnableFuture<Void> ftask = newTaskFor(task, **null**);
43. execute(ftask);  //执行的还是execute方法
44. **return** ftask;
45. }
47. /\*\*
48. \* @throws RejectedExecutionException {@inheritDoc}
49. \* @throws NullPointerException       {@inheritDoc}
50. \*/
51. **public** <T> Future<T> submit(Runnable task, T result) {
52. **if** (task == **null**) **throw** **new** NullPointerException();
53. RunnableFuture<T> ftask = newTaskFor(task, result);
54. execute(ftask);
55. **return** ftask;
56. }
58. /\*\*
59. \* @throws RejectedExecutionException {@inheritDoc}
60. \* @throws NullPointerException       {@inheritDoc}
61. \*/
62. **public** <T> Future<T> submit(Callable<T> task) {
63. **if** (task == **null**) **throw** **new** NullPointerException();
64. RunnableFuture<T> ftask = newTaskFor(task);
65. execute(ftask);
66. **return** ftask;
67. }
69. /\*\*
70. \* the main mechanics of invokeAny.
71. \*/
72. **private** <T> T doInvokeAny(Collection<? **extends** Callable<T>> tasks,
73. **boolean** timed, **long** nanos)
74. **throws** InterruptedException, ExecutionException, TimeoutException {
75. **if** (tasks == **null**)
76. **throw** **new** NullPointerException();
77. **int** ntasks = tasks.size();
78. **if** (ntasks == 0)
79. **throw** **new** IllegalArgumentException();
80. ArrayList<Future<T>> futures = **new** ArrayList<Future<T>>(ntasks);
81. ExecutorCompletionService<T> ecs =
82. **new** ExecutorCompletionService<T>(**this**);
84. // For efficiency, especially in executors with limited
85. // parallelism, check to see if previously submitted tasks are
86. // done before submitting more of them. This interleaving
87. // plus the exception mechanics account for messiness of main
88. // loop.
90. **try** {
91. // Record exceptions so that if we fail to obtain any
92. // result, we can throw the last exception we got.
93. ExecutionException ee = **null**;
94. // 设定超时时间
95. **final** **long** deadline = timed ? System.nanoTime() + nanos : 0L;
96. Iterator<? **extends** Callable<T>> it = tasks.iterator();
98. // Start one task for sure; the rest incrementally
99. futures.add(ecs.submit(it.next()));
100. --ntasks;
101. **int** active = 1;
103. **for** (;;) {
104. // 转调用BlockingQueue的poll方法
105. Future<T> f = ecs.poll();
106. **if** (f == **null**) {
107. **if** (ntasks > 0) {
108. --ntasks;
109. futures.add(ecs.submit(it.next()));
110. ++active;
111. }
112. **else** **if** (active == 0)
113. **break**;
114. **else** **if** (timed) {
115. f = ecs.poll(nanos, TimeUnit.NANOSECONDS);
116. **if** (f == **null**)
117. **throw** **new** TimeoutException();
118. nanos = deadline - System.nanoTime();
119. }
120. **else**
121. f = ecs.take();
122. }
123. **if** (f != **null**) {
124. --active;
125. **try** {
126. **return** f.get();
127. } **catch** (ExecutionException eex) {
128. ee = eex;
129. } **catch** (RuntimeException rex) {
130. ee = **new** ExecutionException(rex);
131. }
132. }
133. }
135. **if** (ee == **null**)
136. ee = **new** ExecutionException();
137. **throw** ee;
139. } **finally** {
140. **for** (**int** i = 0, size = futures.size(); i < size; i++)
141. futures.get(i).cancel(**true**);
142. }
143. }
145. **public** <T> T invokeAny(Collection<? **extends** Callable<T>> tasks)
146. **throws** InterruptedException, ExecutionException {
147. **try** {
148. **return** doInvokeAny(tasks, **false**, 0);
149. } **catch** (TimeoutException cannotHappen) {
150. **assert** **false**;
151. **return** **null**;
152. }
153. }
155. **public** <T> T invokeAny(Collection<? **extends** Callable<T>> tasks,
156. **long** timeout, TimeUnit unit)
157. **throws** InterruptedException, ExecutionException, TimeoutException {
158. **return** doInvokeAny(tasks, **true**, unit.toNanos(timeout));
159. }
161. **public** <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)
162. **throws** InterruptedException {
163. **if** (tasks == **null**)
164. **throw** **new** NullPointerException();
165. ArrayList<Future<T>> futures = **new** ArrayList<Future<T>>(tasks.size());
166. **boolean** done = **false**;
167. **try** {
168. **for** (Callable<T> t : tasks) {
169. RunnableFuture<T> f = newTaskFor(t);
170. futures.add(f);
171. execute(f);
172. }
173. **for** (**int** i = 0, size = futures.size(); i < size; i++) {
174. Future<T> f = futures.get(i);
175. **if** (!f.isDone()) {
176. **try** {
177. f.get();
178. } **catch** (CancellationException ignore) {
179. } **catch** (ExecutionException ignore) {
180. }
181. }
182. }
183. done = **true**;
184. **return** futures;
185. } **finally** {
186. **if** (!done)
187. **for** (**int** i = 0, size = futures.size(); i < size; i++)
188. futures.get(i).cancel(**true**);
189. }
190. }
192. **public** <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,
193. **long** timeout, TimeUnit unit)
194. **throws** InterruptedException {
195. **if** (tasks == **null**)
196. **throw** **new** NullPointerException();
197. **long** nanos = unit.toNanos(timeout);
198. ArrayList<Future<T>> futures = **new** ArrayList<Future<T>>(tasks.size());
199. **boolean** done = **false**;
200. **try** {
201. **for** (Callable<T> t : tasks)
202. futures.add(newTaskFor(t));
204. **final** **long** deadline = System.nanoTime() + nanos;
205. **final** **int** size = futures.size();
207. // Interleave time checks and calls to execute in case
208. // executor doesn't have any/much parallelism.
209. **for** (**int** i = 0; i < size; i++) {
210. execute((Runnable)futures.get(i));
211. nanos = deadline - System.nanoTime();
212. **if** (nanos <= 0L)
213. **return** futures;
214. }
216. **for** (**int** i = 0; i < size; i++) {
217. Future<T> f = futures.get(i);
218. **if** (!f.isDone()) {
219. **if** (nanos <= 0L)
220. **return** futures;
221. **try** {
222. f.get(nanos, TimeUnit.NANOSECONDS);
223. } **catch** (CancellationException ignore) {
224. } **catch** (ExecutionException ignore) {
225. } **catch** (TimeoutException toe) {
226. **return** futures;
227. }
228. nanos = deadline - System.nanoTime();
229. }
230. }
231. done = **true**;
232. **return** futures;
233. } **finally** {
234. **if** (!done)
235. **for** (**int** i = 0, size = futures.size(); i < size; i++)
236. futures.get(i).cancel(**true**);
237. }
238. }
240. }

## 常量

### COUNT\_BITS

1. **private** **static** final **int** COUNT\_BITS = Integer.SIZE - 3;

1、用于保存线程数量的位数，29位

### CAPACITY

1. **private** **static** final **int** CAPACITY   = (1 << COUNT\_BITS) - 1;

1、最多线程数量

### RUNNING

1. **private** **static** final **int** RUNNING    = -1 << COUNT\_BITS;

1、RUNNING状态：可以接受新的任务，并且会执行已存在于队列中的任务

### SHUTDOWN

1. **private** **static** final **int** SHUTDOWN   =  0 << COUNT\_BITS;

1、SHUTDOWN状态：不再接受新的任务，但是会执行已在队列中的任务

### STOP

1. **private** **static** final **int** STOP       =  1 << COUNT\_BITS;

1、STOP状态：不再接受新的任务，也不会执行队列中的任务

### TIDYING

1. **private** **static** final **int** TIDYING    =  2 << COUNT\_BITS;

1、TIDYING状态：所有任务已经终止，workerCount为零，线程转换到状态TIDYING将运行terminate()钩子方法

### TERMINATED

1. **private** **static** final **int** TERMINATED =  3 << COUNT\_BITS;

1、TERMINATED状态：terminated() 方法执行完毕

## 字段

### ctl

1. **private** final AtomicInteger ctl = **new** AtomicInteger(ctlOf(RUNNING, 0));

1、这个变量是整个类的核心，AtomicInteger保证了对这个变量的操作是原子的，通过巧妙的操作，ThreadPoolExecutor用这一个变量保存了两个内容：

1. 所有有效线程的数量
2. 各个线程的状态(runState)

2、低29位存线程数，高3位存runState,这样runState有5个值

1. RUNNING:-536870912
2. SHUTDOWN:0
3. STOP:536870912
4. TIDYING:1073741824
5. TERMINATED:1610612736

3、状态含义解释

1. RUNNING状态：线程池正常运行，可以接受新的任务并处理队列中的任务；
2. SHUTDOWN状态：不再接受新的任务，但是会执行队列中的任务；
3. STOP状态：不再接受新任务，不处理队列中的任务

### workQueue

1. /\*\*
2. \* The queue used for holding tasks and handing off to worker
3. \* threads.  We do not require that workQueue.poll() returning
4. \* null necessarily means that workQueue.isEmpty(), so rely
5. \* solely on isEmpty to see if the queue is empty (which we must
6. \* do for example when deciding whether to transition from
7. \* SHUTDOWN to TIDYING).  This accommodates special-purpose
8. \* queues such as DelayQueues for which poll() is allowed to
9. \* return null even if it may later return non-null when delays
10. \* expire.
11. \*/
12. **private** **final** BlockingQueue<Runnable> workQueue;

1、保存任务的阻塞队列???

### mainLock

1. /\*\*
2. \* Lock held on access to workers set and related bookkeeping.
3. \* While we could use a concurrent set of some sort, it turns out
4. \* to be generally preferable to use a lock. Among the reasons is
5. \* that this serializes interruptIdleWorkers, which avoids
6. \* unnecessary interrupt storms, especially during shutdown.
7. \* Otherwise exiting threads would concurrently interrupt those
8. \* that have not yet interrupted. It also simplifies some of the
9. \* associated statistics bookkeeping of largestPoolSize etc. We
10. \* also hold mainLock on shutdown and shutdownNow, for the sake of
11. \* ensuring workers set is stable while separately checking
12. \* permission to interrupt and actually interrupting.
13. \*/
14. **private** **final** ReentrantLock mainLock = **new** ReentrantLock();

1、重入锁???

### workers

1. /\*\*
2. \* Set containing all worker threads in pool. Accessed only when
3. \* holding mainLock.
4. \*/
5. **private** **final** HashSet<Worker> workers = **new** HashSet<Worker>();

1、线程集合，仅在持有mainLock时才能访问

### termination

1. /\*\*
2. \* Wait condition to support awaitTermination
3. \*/
4. **private** **final** Condition termination = mainLock.newCondition();

1、???

### largestPoolSize

1. /\*\*
2. \* Tracks largest attained pool size. Accessed only under
3. \* mainLock.
4. \*/
5. **private** **int** largestPoolSize;

1、???

### completedTaskCount

1. /\*\*
2. \* Counter for completed tasks. Updated only on termination of
3. \* worker threads. Accessed only under mainLock.
4. \*/
5. **private** **long** completedTaskCount;

1、完成任务的数量

### threadFactory

1. /\*\*
2. \* Factory for new threads. All threads are created using this
3. \* factory (via method addWorker).  All callers must be prepared
4. \* for addWorker to fail, which may reflect a system or user's
5. \* policy limiting the number of threads.  Even though it is not
6. \* treated as an error, failure to create threads may result in
7. \* new tasks being rejected or existing ones remaining stuck in
8. \* the queue.
9. \*
10. \* We go further and preserve pool invariants even in the face of
11. \* errors such as OutOfMemoryError, that might be thrown while
12. \* trying to create threads.  Such errors are rather common due to
13. \* the need to allocate a native stack in Thread#start, and users
14. \* will want to perform clean pool shutdown to clean up.  There
15. \* will likely be enough memory available for the cleanup code to
16. \* complete without encountering yet another OutOfMemoryError.
17. \*/
18. **private** **volatile** ThreadFactory threadFactory;

1、线程工厂

### handler

1. /\*\*
2. \* Handler called when saturated or shutdown in execute.
3. \*/
4. **private** **volatile** RejectedExecutionHandler handler;
5. /\*\*
6. \* The default rejected execution handler
7. \*/
8. **private** **static** final RejectedExecutionHandler defaultHandler =
9. **new** AbortPolicy();

1、拒绝策略，当任务太多来不及处理，可拒绝该任务，下面是默认的策略，即抛出异常

1. /\*\*
2. \* A handler for rejected tasks that throws a
3. \* {@code RejectedExecutionException}.
4. \*/
5. **public** **static** **class** AbortPolicy implements RejectedExecutionHandler {
6. /\*\*
7. \* Creates an {@code AbortPolicy}.
8. \*/
9. **public** AbortPolicy() { }
11. /\*\*
12. \* Always throws RejectedExecutionException.
13. \*
14. \* @param r the runnable task requested to be executed
15. \* @param e the executor attempting to execute this task
16. \* @throws RejectedExecutionException always
17. \*/
18. **public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {
19. **throw** **new** RejectedExecutionException("Task " + r.toString() +
20. " rejected from " +
21. e.toString());
22. }
23. }

### keepAliveTime

1. /\*\*
2. \* Timeout in nanoseconds for idle threads waiting for work.
3. \* Threads use this timeout when there are more than corePoolSize
4. \* present or if allowCoreThreadTimeOut. Otherwise they wait
5. \* forever for new work.
6. \*/
7. **private** **volatile** **long** keepAliveTime;

1、线程从队列中获取任务的超时时间，也就是说如果线程空闲超过这个时间就会终止

### allowCoreThreadTimeOut

1. /\*\*
2. \* If false (default), core threads stay alive even when idle.
3. \* If true, core threads use keepAliveTime to time out waiting
4. \* for work.
5. \*/
6. **private** **volatile** **boolean** allowCoreThreadTimeOut;

1、是否允许核心线程超时后释放

### corePoolSize

1. /\*\*
2. \* Core pool size is the minimum number of workers to keep alive
3. \* (and not allow to time out etc) unless allowCoreThreadTimeOut
4. \* is set, in which case the minimum is zero.
5. \*/
6. **private** **volatile** **int** corePoolSize;

1、核心线程池大小，活动线程小于corePoolSize则直接创建，大于等于则先加到workQueue中，队列满了才创建新的线程

### maximumPoolSize

1. /\*\*
2. \* Maximum pool size. Note that the actual maximum is internally
3. \* bounded by CAPACITY.
4. \*/
5. **private** **volatile** **int** maximumPoolSize;

1、最大线程数量

## 内部类

### Worker

1. **private** **final** **class** Worker
2. **extends** AbstractQueuedSynchronizer
3. **implements** Runnable
4. {
5. /\*\*
6. \* This class will never be serialized, but we provide a
7. \* serialVersionUID to suppress a javac warning.
8. \*/
9. **private** **static** **final** **long** serialVersionUID = 6138294804551838833L;
11. /\*\* Thread this worker is running in.  Null if factory fails. \*/
12. **final** Thread thread;
13. /\*\* Initial task to run.  Possibly null. \*/
14. Runnable firstTask;
15. /\*\* Per-thread task counter \*/
16. **volatile** **long** completedTasks;
18. /\*\*
19. \* Creates with given first task and thread from ThreadFactory.
20. \* @param firstTask the first task (null if none)
21. \*/
22. Worker(Runnable firstTask) {
23. setState(-1); // inhibit interrupts until runWorker
24. **this**.firstTask = firstTask;
25. **this**.thread = getThreadFactory().newThread(**this**);
26. }
28. /\*\* Delegates main run loop to outer runWorker  \*/
29. **public** **void** run() {
30. runWorker(**this**);
31. }
33. // Lock methods
34. //
35. // The value 0 represents the unlocked state.
36. // The value 1 represents the locked state.
38. **protected** **boolean** isHeldExclusively() {
39. **return** getState() != 0;
40. }
42. **protected** **boolean** tryAcquire(**int** unused) {
43. **if** (compareAndSetState(0, 1)) {
44. //独占，且不可重入
45. setExclusiveOwnerThread(Thread.currentThread());
46. **return** **true**;
47. }
48. **return** **false**;
49. }
51. **protected** **boolean** tryRelease(**int** unused) {
52. setExclusiveOwnerThread(**null**);
53. setState(0);
54. **return** **true**;
55. }
57. **public** **void** lock()        { acquire(1); }
58. **public** **boolean** tryLock()  { **return** tryAcquire(1); }
59. **public** **void** unlock()      { release(1); }
60. **public** **boolean** isLocked() { **return** isHeldExclusively(); }
62. **void** interruptIfStarted() {
63. Thread t;
64. **if** (getState() >= 0 && (t = thread) != **null** && !t.isInterrupted()) {
65. **try** {
66. t.interrupt();
67. } **catch** (SecurityException ignore) {
68. }
69. }
70. }
71. }

1、继承自AQS框架，这个Worker是一个封装了Thread的对象，维护了了一个同步阻塞队列，感觉有点乱???为什么要继承AQS???

2、设计成不可重入的原因是为了避免任务执行的代码中修改线程池的变量

## Utils方法

### runStateOf

1. **private** **static** **int** runStateOf(**int** c)     { **return** c & ~CAPACITY; }

1、这个方法用于取出runState的值 因为CAPACITY值为：00011111111111111111111111111111，~为按位取反操作，则~CAPACITY值为：11100000000000000000000000000000，再同参数做&操作，就将低29位置0了，而高3位还是保持原先的值，也就是runState的值

### workerCountOf

1. **private** **static** **int** workerCountOf(**int** c)  { **return** c & CAPACITY; }

1、这个方法用于取出workerCount的值，因为CAPACITY值为：00011111111111111111111111111111，所以&操作将参数的高3位置0了，保留参数的低29位，也就是workerCount的值

### ctlOf

1. **private** **static** **int** ctlOf(**int** rs, **int** wc) { **return** rs | wc; }

1、将runState和workerCount存到同一个int中，就是"|"运算的意思是，假设rs的值是101000，wc的值是000111，则他们位或运算的值为101111

### isRunning

1. **private** **static** **boolean** isRunning(**int** c) {
2. **return** c < SHUTDOWN;
3. }

1、只有RUNNING状态小于0

### runStateLessThan

1. **private** **static** **boolean** runStateLessThan(**int** c, **int** s) {
2. **return** c < s;
3. }

1、判断c是否小于s

### runStateAtLeast

1. **private** **static** **boolean** runStateAtLeast(**int** c, **int** s) {
2. **return** c >= s;
3. }

1、判断c是否大于等于s

## 方法

### execute

1、execute方法主要三个步骤：

1. 活动线程小于corePoolSize的时候创建新的线程
2. 活动线程大于corePoolSize时都是先加入到任务队列当中
3. 任务队列满了再去启动新的线程，如果线程数达到最大值就拒绝任务

2、源码

1. /\*\*
2. \* Executes the given task sometime in the future.  The task
3. \* may execute in a new thread or in an existing pooled thread.
4. \*
5. \* If the task cannot be submitted for execution, either because this
6. \* executor has been shutdown or because its capacity has been reached,
7. \* the task is handled by the current {@code RejectedExecutionHandler}.
8. \*
9. \* @param command the task to execute
10. \* @throws RejectedExecutionException at discretion of
11. \*         {@code RejectedExecutionHandler}, if the task
12. \*         cannot be accepted for execution
13. \* @throws NullPointerException if {@code command} is null
14. \*/
15. **public** **void** execute(Runnable command) {
16. **if** (command == **null**)
17. **throw** **new** NullPointerException();
18. /\*
19. \* Proceed in 3 steps:
20. \*
21. \* 1. If fewer than corePoolSize threads are running, try to
22. \* start a new thread with the given command as its first
23. \* task.  The call to addWorker atomically checks runState and
24. \* workerCount, and so prevents false alarms that would add
25. \* threads when it shouldn't, by returning false.
26. \*
27. \* 2. If a task can be successfully queued, then we still need
28. \* to double-check whether we should have added a thread
29. \* (because existing ones died since last checking) or that
30. \* the pool shut down since entry into this method. So we
31. \* recheck state and if necessary roll back the enqueuing if
32. \* stopped, or start a new thread if there are none.
33. \*
34. \* 3. If we cannot queue task, then we try to add a new
35. \* thread.  If it fails, we know we are shut down or saturated
36. \* and so reject the task.
37. \*/
38. **int** c = ctl.get();
39. //当活动线程数量小于核心线程数量
40. **if** (workerCountOf(c) < corePoolSize) {
41. //将任务作为firstTask封装到Work中作为核心线程，并进行添加逻辑。第二个参数true:addWorker中会重新检查workerCount是否小于corePoolSize
42. **if** (addWorker(command, **true**))
43. **return**;
44. c = ctl.get();
45. }
46. //当活动线程数量大于核心线程数量，添加到队列中
47. **if** (isRunning(c) && workQueue.offer(command)) {
48. **int** recheck = ctl.get();
49. //处于非RUNNING状态
50. **if** (! isRunning(recheck) && remove(command))
51. reject(command);  //采用线程池指定的策略拒绝任务
52. //线程池处于RUNNING状态 || 线程池处于非RUNNING状态但是任务移除失败
53. **else** **if** (workerCountOf(recheck) == 0)
54. //这行代码是为了SHUTDOWN状态下没有活动线程了，但是队列里还有任务没执行这种特殊情况，添加一个null任务是因为SHUTDOWN状态下，线程池不再接受新任务
55. addWorker(**null**, **false**);
56. }
57. //将任务作为firstTask封装到Work中作为非核心线程，并进行添加逻辑
58. **else** **if** (!addWorker(command, **false**))
59. reject(command);
60. }

### reject

1. /\*\*
2. \* Invokes the rejected execution handler for the given command.
3. \* Package-protected for use by ScheduledThreadPoolExecutor.
4. \*/
5. final **void** reject(Runnable command) {
6. handler.rejectedExecution(command, **this**);
7. }

1、回调先前设定的策略来处理拒绝的请求

### addWork

1. /\*\*
2. \* Checks if a new worker can be added with respect to current
3. \* pool state and the given bound (either core or maximum). If so,
4. \* the worker count is adjusted accordingly, and, if possible, a
5. \* new worker is created and started, running firstTask as its
6. \* first task. This method returns false if the pool is stopped or
7. \* eligible to shut down. It also returns false if the thread
8. \* factory fails to create a thread when asked.  If the thread
9. \* creation fails, either due to the thread factory returning
10. \* null, or due to an exception (typically OutOfMemoryError in
11. \* Thread#start), we roll back cleanly.
12. \*
13. \* @param firstTask the task the new thread should run first (or
14. \* null if none). Workers are created with an initial first task
15. \* (in method execute()) to bypass queuing when there are fewer
16. \* than corePoolSize threads (in which case we always start one),
17. \* or when the queue is full (in which case we must bypass queue).
18. \* Initially idle threads are usually created via
19. \* prestartCoreThread or to replace other dying workers.
20. \*
21. \* @param core if true use corePoolSize as bound, else
22. \* maximumPoolSize. (A boolean indicator is used here rather than a
23. \* value to ensure reads of fresh values after checking other pool
24. \* state).
25. \* @return true if successful
26. \*/
27. **private** **boolean** addWorker(Runnable firstTask, **boolean** core) {
28. retry:
29. **for** (;;) {
30. **int** c = ctl.get();
31. **int** rs = runStateOf(c);
33. // Check if queue empty only if necessary.
34. //条件成立的情况(函数失败)：
35. //1.rs>SHUTDOWN;(大于SHUTDOWN时添加任务当然是失败的)
36. //2.rs==SHUTDOWN且firstTask!=null;(当处于SHUTDOWN时不再接受新任务，但是可以接受为null的任务)
37. //3.rs=SHUTDOWN且firstTask==null且workQueue为空(当处于SHUTDOWN时且任务为空，此时如果队列中没有暂未处理的任务，也是不能添加Work的)
38. //条件不成立的情况(函数可以继续往下走)：
39. //1.rs==RUNNING;(运行状态当然可以添加任务咯，至于是否成功还得看后面的逻辑)
40. //2.rs==SHUTDOWN且firstTask==null且workQueue不为空(当处于SHUTDOWN状态并且添加的任务为空时，如果队列中仍有尚未执行的任务，那么允许添加Work)
41. **if** (rs >= SHUTDOWN &&
42. ! (rs == SHUTDOWN &&
43. firstTask == **null** &&
44. ! workQueue.isEmpty()))
45. **return** **false**;
47. //此循环就是为了增加线程计数值，为什么不调用ctl的increment方法?因为还需要判断线程数量的合法性(不一定非得递增成功)，而increment是通过循环+CAS保证递增成功的。
48. **for** (;;) {
49. **int** wc = workerCountOf(c);
50. //当线程数量大于容量或者大于核心或最大线程数量时，返回false
51. **if** (wc >= CAPACITY ||
52. wc >= (core ? corePoolSize : maximumPoolSize))
53. **return** **false**;
54. //CAS操作将递增计数的操作串行化，成功则退出大循环(继续执行唯一出口)
55. **if** (compareAndIncrementWorkerCount(c))
56. **break** retry;
57. c = ctl.get();  // Re-read ctl
58. //当状态没有发生变化时继续内层循环，否则到外层循环
59. **if** (runStateOf(c) != rs)
60. **continue** retry;
61. // else CAS failed due to workerCount change; retry inner loop
62. }
63. }
64. //线程计数值自增成功才会执行到这里
65. //下面的逻辑是：将任务封装成Worker并添加到workers中，然后启动
66. **boolean** workerStarted = **false**;
67. **boolean** workerAdded = **false**;
68. Worker w = **null**;
69. **try** {
70. **final** ReentrantLock mainLock = **this**.mainLock;
71. w = **new** Worker(firstTask);
72. **final** Thread t = w.thread;
73. **if** (t != **null**) {
74. mainLock.lock();
75. //下面的try语句块将w添加到works中，由于works是HashSet，是非线程安全的，因此必须加锁，串行添加操作
76. **try** {
77. // Recheck while holding lock.
78. // Back out on ThreadFactory failure or if
79. // shut down before lock acquired.
80. **int** c = ctl.get();
81. **int** rs = runStateOf(c);
82. //条件成立的情况：1.rs==RUNNING;2.rs==SHUTDOWN且firstTask==null(SHUTDOWN状态不接受新任务了)
83. **if** (rs < SHUTDOWN ||
84. (rs == SHUTDOWN && firstTask == **null**)) {
85. //若线程已经开始，则抛出异常
86. **if** (t.isAlive()) // precheck that t is startable
87. **throw** **new** IllegalThreadStateException();
88. workers.add(w);
89. **int** s = workers.size();
90. **if** (s > largestPoolSize)
91. //largestPoolSize也是非线程安全的，因此也必须在锁内
92. largestPoolSize = s;
93. workerAdded = **true**;
94. }
95. } **finally** {
96. mainLock.unlock();
97. }
98. //启动线程
99. **if** (workerAdded) {
100. t.start();
101. workerStarted = **true**;
102. }
103. }
104. } **finally** {
105. **if** (! workerStarted)
106. addWorkerFailed(w);
107. }
108. **return** workerStarted;
109. }

1、addWorker有2种情况

1. 一种就是线程数量不足核心线程数
2. 另一种就是核心线程数已满同时任务队列已满但是线程数不足最大线程数

* 上述boolean core就是用来区分上述2种情况的

2、addWorker首先就要对线程数量自增，即ctl的后29为进行自增。这里就涉及到多线程问题，为了解决多线程问题就采用了for循环加CAS来解决，为什么没有直接用AtomicInteger的incrementAndGet?

* incrementAndGet也是内部for循环加CAS，它是要确保一定要自增成功的
* 而这里我们不一定要自增成功，还要判断当前线程的数量合不合法。 即如果core=true,则当前线程数量不能超过incrementAndGet，如果core=false，则当前线程数量不能超过maximumPoolSize

### runWorker

1. /\*\*
2. \* Main worker run loop.  Repeatedly gets tasks from queue and
3. \* executes them, while coping with a number of issues:
4. \*
5. \* 1. We may start out with an initial task, in which case we
6. \* don't need to get the first one. Otherwise, as long as pool is
7. \* running, we get tasks from getTask. If it returns null then the
8. \* worker exits due to changed pool state or configuration
9. \* parameters.  Other exits result from exception throws in
10. \* external code, in which case completedAbruptly holds, which
11. \* usually leads processWorkerExit to replace this thread.
12. \*
13. \* 2. Before running any task, the lock is acquired to prevent
14. \* other pool interrupts while the task is executing, and then we
15. \* ensure that unless pool is stopping, this thread does not have
16. \* its interrupt set.
17. \*
18. \* 3. Each task run is preceded by a call to beforeExecute, which
19. \* might throw an exception, in which case we cause thread to die
20. \* (breaking loop with completedAbruptly true) without processing
21. \* the task.
22. \*
23. \* 4. Assuming beforeExecute completes normally, we run the task,
24. \* gathering any of its thrown exceptions to send to afterExecute.
25. \* We separately handle RuntimeException, Error (both of which the
26. \* specs guarantee that we trap) and arbitrary Throwables.
27. \* Because we cannot rethrow Throwables within Runnable.run, we
28. \* wrap them within Errors on the way out (to the thread's
29. \* UncaughtExceptionHandler).  Any thrown exception also
30. \* conservatively causes thread to die.
31. \*
32. \* 5. After task.run completes, we call afterExecute, which may
33. \* also throw an exception, which will also cause thread to
34. \* die. According to JLS Sec 14.20, this exception is the one that
35. \* will be in effect even if task.run throws.
36. \*
37. \* The net effect of the exception mechanics is that afterExecute
38. \* and the thread's UncaughtExceptionHandler have as accurate
39. \* information as we can provide about any problems encountered by
40. \* user code.
41. \*
42. \* @param w the worker
43. \*/
44. final **void** runWorker(Worker w) {
45. Thread wt = Thread.currentThread();
46. //取出Work添加时所附带的firstTask，然后将该字段置为null，之后再也不会进行赋值了
47. Runnable task = w.firstTask;
48. w.firstTask = null;
49. w.unlock(); // allow interrupts
50. boolean completedAbruptly = **true**;
51. **try** {
52. //当firstTask不为空，或者从队列中取出非空任务时，执行该任务
53. //可以看出当从队列中取不出任务时当前线程就阻塞了，阻塞时是未加锁的
54. **while** (task != null || (task = getTask()) != null) {
55. w.lock();
56. // If pool is stopping, ensure thread is interrupted;
57. // if not, ensure thread is not interrupted.  This
58. // requires a recheck in second case to deal with
59. // shutdownNow race while clearing interrupt
60. //第二个条件很有意思，在执行Thread#interrupted(该静态方法会清空当前线程的终端标志位)时又检查了一遍状态，这是为了处理在清空标志位后该线程池又被执行了shutdownNow方法
61. **if** ((runStateAtLeast(ctl.get(), STOP) ||
62. (Thread.interrupted() &&
63. runStateAtLeast(ctl.get(), STOP))) &&
64. !wt.isInterrupted())
65. //对于一个在执行getTask()方法中被挂起的线程执行interrupt会发生什么?返回一个null???如何响应shutdownNow，终止线程
66. wt.interrupt();
67. **try** {
68. beforeExecute(wt, task);
69. Throwable thrown = null;
70. **try** {
71. task.run();
72. } **catch** (RuntimeException x) {
73. thrown = x; **throw** x;
74. } **catch** (Error x) {
75. thrown = x; **throw** x;
76. } **catch** (Throwable x) {
77. thrown = x; **throw** **new** Error(x);
78. } finally {
79. afterExecute(task, thrown);
80. }
81. } finally {
82. task = null;
83. w.completedTasks++;
84. w.unlock();
85. }
86. }
87. completedAbruptly = **false**;
88. } finally {
89. processWorkerExit(w, completedAbruptly);
90. }
91. }

### beforeExecute

1. /\*\*
2. \* Method invoked prior to executing the given Runnable in the
3. \* given thread.  This method is invoked by thread {@code t} that
4. \* will execute task {@code r}, and may be used to re-initialize
5. \* ThreadLocals, or to perform logging.
6. \*
7. \* <p>This implementation does nothing, but may be customized in
8. \* subclasses. Note: To properly nest multiple overridings, subclasses
9. \* should generally invoke {@code super.beforeExecute} at the end of
10. \* this method.
11. \*
12. \* @param t the thread that will run task {@code r}
13. \* @param r the task that will be executed
14. \*/
15. **protected** **void** beforeExecute(Thread t, Runnable r) { }

1、很显然，这个方法是留给子类实现，定义执行任务前的一些额外操作

### afterExecute

1. /\*\*
2. \* Method invoked upon completion of execution of the given Runnable.
3. \* This method is invoked by the thread that executed the task. If
4. \* non-null, the Throwable is the uncaught {@code RuntimeException}
5. \* or {@code Error} that caused execution to terminate abruptly.
6. \*
7. \* <p>This implementation does nothing, but may be customized in
8. \* subclasses. Note: To properly nest multiple overridings, subclasses
9. \* should generally invoke {@code super.afterExecute} at the
10. \* beginning of this method.
11. \*
12. \* <p><b>Note:</b> When actions are enclosed in tasks (such as
13. \* {@link FutureTask}) either explicitly or via methods such as
14. \* {@code submit}, these task objects catch and maintain
15. \* computational exceptions, and so they do not cause abrupt
16. \* termination, and the internal exceptions are <em>not</em>
17. \* passed to this method. If you would like to trap both kinds of
18. \* failures in this method, you can further probe for such cases,
19. \* as in this sample subclass that prints either the direct cause
20. \* or the underlying exception if a task has been aborted:
21. \*
22. \*  <pre> {@code
23. \* class ExtendedExecutor extends ThreadPoolExecutor {
24. \*   // ...
25. \*   protected void afterExecute(Runnable r, Throwable t) {
26. \*     super.afterExecute(r, t);
27. \*     if (t == null && r instanceof Future<?>) {
28. \*       try {
29. \*         Object result = ((Future<?>) r).get();
30. \*       } catch (CancellationException ce) {
31. \*           t = ce;
32. \*       } catch (ExecutionException ee) {
33. \*           t = ee.getCause();
34. \*       } catch (InterruptedException ie) {
35. \*           Thread.currentThread().interrupt(); // ignore/reset
36. \*       }
37. \*     }
38. \*     if (t != null)
39. \*       System.out.println(t);
40. \*   }
41. \* }}</pre>
42. \*
43. \* @param r the runnable that has completed
44. \* @param t the exception that caused termination, or null if
45. \* execution completed normally
46. \*/
47. **protected** **void** afterExecute(Runnable r, Throwable t) { }

1、很显然，这个方法是留给子类实现，定义执行任务后的一些额外操作

### processWorkerExit

1. /\*\*
2. \* Performs cleanup and bookkeeping for a dying worker. Called
3. \* only from worker threads. Unless completedAbruptly is set,
4. \* assumes that workerCount has already been adjusted to account
5. \* for exit.  This method removes thread from worker set, and
6. \* possibly terminates the pool or replaces the worker if either
7. \* it exited due to user task exception or if fewer than
8. \* corePoolSize workers are running or queue is non-empty but
9. \* there are no workers.
10. \*
11. \* @param w the worker
12. \* @param completedAbruptly if the worker died due to user exception
13. \*/
14. **private** **void** processWorkerExit(Worker w, **boolean** completedAbruptly) {
15. **if** (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
16. decrementWorkerCount();
18. **final** ReentrantLock mainLock = **this**.mainLock;
19. //workers和completedTaskCount是非线程安全的，因此需要全局锁
20. mainLock.lock();
21. **try** {
22. completedTaskCount += w.completedTasks;
23. workers.remove(w);
24. } **finally** {
25. mainLock.unlock();
26. }
28. tryTerminate();
30. **int** c = ctl.get();
31. **if** (runStateLessThan(c, STOP)) {
32. // completedAbruptly==false：非极端情况下结束run方法
33. **if** (!completedAbruptly) {
34. //允许核心线程超时退出时，最少线程数量为0，否则为corePollSize
35. **int** min = allowCoreThreadTimeOut ? 0 : corePoolSize;
36. //当允许核心线程超时退出时，且此时队列中尚有未执行的任务，那么必须将min置为1
37. **if** (min == 0 && ! workQueue.isEmpty())
38. min = 1;
39. //当前核心线程数量大于最少核心线程数量时，该线程退出即可
40. **if** (workerCountOf(c) >= min)
41. **return**; // replacement not needed
42. }
43. //尝试开启一个新的非核心线程：1.当completedAbruptly==true;2.当workerCountOf(c) < min时，需要添加新的线程，这个新线程非当前线程
44. addWorker(**null**, **false**);
45. }
46. }

1、一个线程终止前执行的收尾工作，以及如何保持核心线程数量，并不是执着于当前线程，而是当现有线程数量小于需要的核心线程数量时，就调用addWork进行添加

### tryTerminate

1. /\*\*
2. \* Transitions to TERMINATED state if either (SHUTDOWN and pool
3. \* and queue empty) or (STOP and pool empty).  If otherwise
4. \* eligible to terminate but workerCount is nonzero, interrupts an
5. \* idle worker to ensure that shutdown signals propagate. This
6. \* method must be called following any action that might make
7. \* termination possible -- reducing worker count or removing tasks
8. \* from the queue during shutdown. The method is non-private to
9. \* allow access from ScheduledThreadPoolExecutor.
10. \*/
11. **final** **void** tryTerminate() {
12. **for** (;;) {
13. **int** c = ctl.get();
14. //以下状态直接返回
15. //1.RUNNING状态
16. //2.处于TIDYING状态
17. //3.处于SHUTDOWN状态，且队列不为空
18. **if** (isRunning(c) ||
19. runStateAtLeast(c, TIDYING) ||
20. (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
21. **return**;
22. **if** (workerCountOf(c) != 0) { // Eligible to terminate
23. interruptIdleWorkers(ONLY\_ONE);
24. **return**;
25. }
26. //此时workerCountOf(c)==0
27. **final** ReentrantLock mainLock = **this**.mainLock;
28. mainLock.lock();
29. **try** {
30. **if** (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
31. **try** {
32. terminated();
33. } **finally** {
34. ctl.set(ctlOf(TERMINATED, 0));
35. termination.signalAll();
36. }
37. **return**;
38. }
39. } **finally** {
40. mainLock.unlock();
41. }
42. // else retry on failed CAS
43. }
44. }

### terminated

1. /\*\*
2. \* Method invoked when the Executor has terminated.  Default
3. \* implementation does nothing. Note: To properly nest multiple
4. \* overridings, subclasses should generally invoke
5. \* {@code super.terminated} within this method.
6. \*/
7. **protected** **void** terminated() { }

1、很显然，这个方法是留给子类实现，定义终止后进行的额外操作

### getTask

1. /\*\*
2. \* Performs blocking or timed wait for a task, depending on
3. \* current configuration settings, or returns null if this worker
4. \* must exit because of any of:
5. \* 1. There are more than maximumPoolSize workers (due to
6. \*    a call to setMaximumPoolSize).
7. \* 2. The pool is stopped.
8. \* 3. The pool is shutdown and the queue is empty.
9. \* 4. This worker timed out waiting for a task, and timed-out
10. \*    workers are subject to termination (that is,
11. \*    {@code allowCoreThreadTimeOut || workerCount > corePoolSize})
12. \*    both before and after the timed wait, and if the queue is
13. \*    non-empty, this worker is not the last thread in the pool.
14. \*
15. \* @return task, or null if the worker must exit, in which case
16. \*         workerCount is decremented
17. \*/
18. **private** Runnable getTask() {
19. **boolean** timedOut = **false**; // Did the last poll() time out?
21. **for** (;;) {
22. **int** c = ctl.get();
23. **int** rs = runStateOf(c);
25. // Check if queue empty only if necessary.
26. //条件成立的情况：
27. //rs>=STOP，此时自然所有队列中的任务都不必再做了
28. //rs==SHUTDOWN，且队列中无任务，此时返回null，让Work退出while循环
29. **if** (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
30. decrementWorkerCount();
31. **return** **null**;
32. }
34. **int** wc = workerCountOf(c);
36. // Are workers subject to culling?
37. //当核心线程允许超时终止，或者当前线程数量已经超过核心线程数量了(在这一堆线程中，任意corePoolSize个线程都可以作为核心线程，无所谓，因此本线程此时允许超时终止)
38. **boolean** timed = allowCoreThreadTimeOut || wc > corePoolSize;
40. **if** ((wc > maximumPoolSize || (timed && timedOut))
41. && (wc > 1 || workQueue.isEmpty())) {
42. **if** (compareAndDecrementWorkerCount(c))
43. **return** **null**;
44. **continue**;
45. }
47. **try** {
48. //是否允许超时终止：若允许则用poll指定超时时间，若不允许，则用take直至取出新任务或者被中断
49. Runnable r = timed ?
50. workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
51. workQueue.take();
52. //看取出的任务是否为空，若不为空，则直接返回，否则便是超时了，中断不会走到这里，而是由下面的catch子句捕获
53. **if** (r != **null**)
54. **return** r;
55. timedOut = **true**;
56. } **catch** (InterruptedException retry) {
57. timedOut = **false**;
58. }
59. }
60. }

1、这个方法中包含了超时终止的逻辑

### shutdown

1. /\*\*
2. \* Initiates an orderly shutdown in which previously submitted
3. \* tasks are executed, but no new tasks will be accepted.
4. \* Invocation has no additional effect if already shut down.
5. \*
6. \* <p>This method does not wait for previously submitted tasks to
7. \* complete execution.  Use {@link #awaitTermination awaitTermination}
8. \* to do that.
9. \*
10. \* @throws SecurityException {@inheritDoc}
11. \*/
12. **public** **void** shutdown() {
13. **final** ReentrantLock mainLock = **this**.mainLock;
14. mainLock.lock();
15. **try** {
16. checkShutdownAccess();
17. advanceRunState(SHUTDOWN);
18. interruptIdleWorkers();
19. onShutdown(); // hook for ScheduledThreadPoolExecutor(目前为空方法)
20. } **finally** {
21. mainLock.unlock();
22. }
23. tryTerminate();
24. }

### shutdownNow

1. /\*\*
2. \* Attempts to stop all actively executing tasks, halts the
3. \* processing of waiting tasks, and returns a list of the tasks
4. \* that were awaiting execution. These tasks are drained (removed)
5. \* from the task queue upon return from this method.
6. \*
7. \* <p>This method does not wait for actively executing tasks to
8. \* terminate.  Use {@link #awaitTermination awaitTermination} to
9. \* do that.
10. \*
11. \* <p>There are no guarantees beyond best-effort attempts to stop
12. \* processing actively executing tasks.  This implementation
13. \* cancels tasks via {@link Thread#interrupt}, so any task that
14. \* fails to respond to interrupts may never terminate.
15. \*
16. \* @throws SecurityException {@inheritDoc}
17. \*/
18. **public** List<Runnable> shutdownNow() {
19. List<Runnable> tasks;
20. final ReentrantLock mainLock = **this**.mainLock;
21. mainLock.lock();
22. **try** {
23. checkShutdownAccess();
24. advanceRunState(STOP);
25. interruptWorkers();
26. tasks = drainQueue();
27. } finally {
28. mainLock.unlock();
29. }
30. tryTerminate();
31. **return** tasks;
32. }

### interruptIdleWorkers

1. /\*\*
2. \* Interrupts threads that might be waiting for tasks (as
3. \* indicated by not being locked) so they can check for
4. \* termination or configuration changes. Ignores
5. \* SecurityExceptions (in which case some threads may remain
6. \* uninterrupted).
7. \*
8. \* @param onlyOne If true, interrupt at most one worker. This is
9. \* called only from tryTerminate when termination is otherwise
10. \* enabled but there are still other workers.  In this case, at
11. \* most one waiting worker is interrupted to propagate shutdown
12. \* signals in case all threads are currently waiting.
13. \* Interrupting any arbitrary thread ensures that newly arriving
14. \* workers since shutdown began will also eventually exit.
15. \* To guarantee eventual termination, it suffices to always
16. \* interrupt only one idle worker, but shutdown() interrupts all
17. \* idle workers so that redundant workers exit promptly, not
18. \* waiting for a straggler task to finish.
19. \*/
20. **private** **void** interruptIdleWorkers(**boolean** onlyOne) {
21. **final** ReentrantLock mainLock = **this**.mainLock;
22. mainLock.lock();
23. **try** {
24. **for** (Worker w : workers) {
25. Thread t = w.thread;
26. //当未中断时，尝试获取w的锁，如果获取成功，说明w处于空闲状态，因为w在进行run方法时会将自己锁定，而这个锁是非重入的独占锁
27. **if** (!t.isInterrupted() && w.tryLock()) {
28. **try** {
29. //中断，从getTask()的阻塞状态中唤醒，退出while循环并结束线程
30. t.interrupt();
31. } **catch** (SecurityException ignore) {
32. } **finally** {
33. w.unlock();
34. }
35. }
36. //是否仅仅需要终止一个空闲线程
37. **if** (onlyOne)
38. **break**;
39. }
40. } **finally** {
41. mainLock.unlock();
42. }
43. }

1、这个方法是为什么Worker为什么需要继承AQS框架的原因

### interruptWorkers

1. /\*\*
2. \* Interrupts all threads, even if active. Ignores SecurityExceptions
3. \* (in which case some threads may remain uninterrupted).
4. \*/
5. **private** **void** interruptWorkers() {
6. final ReentrantLock mainLock = **this**.mainLock;
7. mainLock.lock();
8. **try** {
9. **for** (Worker w : workers)
10. //调用Work# interruptIfStarted
11. w.interruptIfStarted();
12. } finally {
13. mainLock.unlock();
14. }
15. }

## 问题

### 如何实现让一个线程空闲一段时间后结束

1、由于Worker的run方法(回调ThreadPoolExecutor#runWorker方法)，线程会阻塞在ThreadPoolExecutor#getTask方法中

2、在ThreadPoolExecutor#getTask会根据是否允许超时结束的设定，选择调用BlockingQueue的poll()或者take方法，其中poll方法允许指定阻塞时间，而take方法会阻塞直到获取到元素或者被中断

### 为什么Worker需要继承AQS框架

1、为了解决如下问题：有这样一个需求，要终止一个或者一批空闲着的线程，此处会有一个问题，如何判断一个线程是否处于空闲状态

2、这里线程处于空闲状态是指线程不在处理ThreadPoolExecutor所接受的任务，而在ThreadPoolExecutor#getTask中阻塞

3、ThreadPoolExecutor#runWorker方法中，在成功获取到任务后，会将自己锁定，这个时候就通过这个锁定状态将自己标记为非空闲。而当一个任务处理完毕后，又会解除锁定状态

4、ThreadPoolExecutor#interruptIdleWorkers方法中会调用Worker的tryLock方法，当成功锁定时，表明该Worker没有在处理任务，因此是空闲状态，可以进行关闭

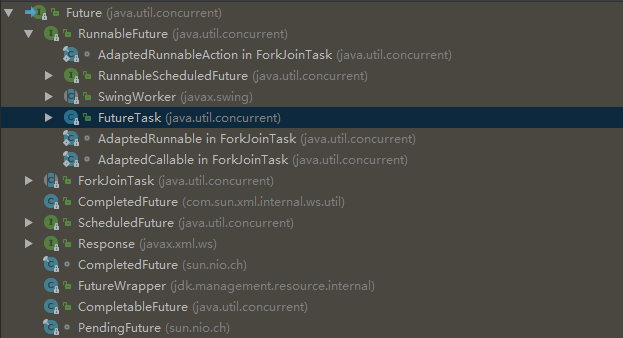
5、为什么不用一个状态量来标记，使用状态量就必须要加锁，而AQS框架提供了一种轻量级的加锁，因此性能会更高一点

concurrent结构



# FutureTask

## 继承体系



### Future接口

1. **public** **interface** Future<V> {
3. /\*\*
4. \* Attempts to cancel execution of this task.  This attempt will
5. \* fail if the task has already completed, has already been cancelled,
6. \* or could not be cancelled for some other reason. If successful,
7. \* and this task has not started when {@code cancel} is called,
8. \* this task should never run.  If the task has already started,
9. \* then the {@code mayInterruptIfRunning} parameter determines
10. \* whether the thread executing this task should be interrupted in
11. \* an attempt to stop the task.
12. \*
13. \* <p>After this method returns, subsequent calls to {@link #isDone} will
14. \* always return {@code true}.  Subsequent calls to {@link #isCancelled}
15. \* will always return {@code true} if this method returned {@code true}.
16. \*
17. \* @param mayInterruptIfRunning {@code true} if the thread executing this
18. \* task should be interrupted; otherwise, in-progress tasks are allowed
19. \* to complete
20. \* @return {@code false} if the task could not be cancelled,
21. \* typically because it has already completed normally;
22. \* {@code true} otherwise
23. \*/
24. **boolean** cancel(**boolean** mayInterruptIfRunning);
26. /\*\*
27. \* Returns {@code true} if this task was cancelled before it completed
28. \* normally.
29. \*
30. \* @return {@code true} if this task was cancelled before it completed
31. \*/
32. **boolean** isCancelled();
34. /\*\*
35. \* Returns {@code true} if this task completed.
36. \*
37. \* Completion may be due to normal termination, an exception, or
38. \* cancellation -- in all of these cases, this method will return
39. \* {@code true}.
40. \*
41. \* @return {@code true} if this task completed
42. \*/
43. **boolean** isDone();
45. /\*\*
46. \* Waits if necessary for the computation to complete, and then
47. \* retrieves its result.
48. \*
49. \* @return the computed result
50. \* @throws CancellationException if the computation was cancelled
51. \* @throws ExecutionException if the computation threw an
52. \* exception
53. \* @throws InterruptedException if the current thread was interrupted
54. \* while waiting
55. \*/
56. V get() **throws** InterruptedException, ExecutionException;
58. /\*\*
59. \* Waits if necessary for at most the given time for the computation
60. \* to complete, and then retrieves its result, if available.
61. \*
62. \* @param timeout the maximum time to wait
63. \* @param unit the time unit of the timeout argument
64. \* @return the computed result
65. \* @throws CancellationException if the computation was cancelled
66. \* @throws ExecutionException if the computation threw an
67. \* exception
68. \* @throws InterruptedException if the current thread was interrupted
69. \* while waiting
70. \* @throws TimeoutException if the wait timed out
71. \*/
72. V get(**long** timeout, TimeUnit unit)
73. **throws** InterruptedException, ExecutionException, TimeoutException;
74. }

### RunnableFuture接口

1. **public** **interface** RunnableFuture<V> **extends** Runnable, Future<V> {
2. /\*\*
3. \* Sets this Future to the result of its computation
4. \* unless it has been cancelled.
5. \*/
6. **void** run();
7. }

1、该接口继承了Runnable以及Future接口

### RunnableAdapter

1. /\*\*
2. \* A callable that runs given task and returns given result
3. \*/
4. **static** **final** **class** RunnableAdapter<T> **implements** Callable<T> {
5. **final** Runnable task;
6. **final** T result;
7. RunnableAdapter(Runnable task, T result) {
8. **this**.task = task;
9. **this**.result = result;
10. }
11. **public** T call() {
12. task.run();
13. //这个result与taks的run方法无任何关系，传个非空的result有什么意思???
14. **return** result;
15. }
16. }

1、该类并不在Future的继承体系中，该类是Callable的实现类，该类的的作用就是将Runnable封装成Callable<T>，其run方法会回调封装task的run方法，由于Future支持返回任务运行结果，因此Future内部必须持有Callable而不是Runnable

## 常量

### NEW

1. **private** **static** **final** **int** NEW          = 0;

1、初始化状态，以及正在执行run方法且未被终止或中断时的状态

### COMPLETING

1. **private** **static** **final** **int** COMPLETING   = 1;

1、正在完成状态，从NEW变为COMPLETING，仅在方法set以及setException中进行CAS赋值

### NORMAL

1. **private** **static** **final** **int** NORMAL       = 2;

1、正常结束状态

### EXCEPTIONAL

1. **private** **static** **final** **int** EXCEPTIONAL  = 3;

1、???

### CANCELLED

1. **private** **static** **final** **int** CANCELLED    = 4;

1、被终止状态

### INTERRUPTING

1. **private** **static** **final** **int** INTERRUPTING = 5;

1、正在被中断

### INTERRUPTED

1. **private** **static** **final** **int** INTERRUPTED  = 6;

1、已经中断

### 转换关系

1、NEW -> COMPLETING -> NORMAL

2、NEW -> COMPLETING -> EXCEPTIONAL

3、NEW -> CANCELLED

4、NEW -> INTERRUPTING -> INTERRUPTED

## 字段

### callable

1. /\*\* The underlying callable; nulled out after running \*/
2. **private** Callable<V> callable;

1、可以理解为可调用的任务，FutureTask的构造方法会将Runnable通过一个适配器RunnableAdapter(实现了Callable接口)封装成一个Callable

2、因为Future支持get操作获取任务的结果，因此内部应该包含的是Callable而不是Runnable

### outcome

1. /\*\* The result to return or exception to throw from get() \*/
2. **private** Object outcome; // non-volatile, protected by state reads/writes

1、get()方法返回的结果，是Callable执行后返回的结果

### runner

1. /\*\* The thread running the callable; CASed during run() \*/
2. **private** **volatile** Thread runner;

1、持有的线程引用

### waiters

1. /\*\* Treiber stack of waiting threads \*/
2. **private** **volatile** WaitNode waiters;

1、等待线程链表表头，只有在方法awaitDone中会被赋值

### unsafe字段

1. **private** **static** **final** sun.misc.Unsafe UNSAFE;
2. **private** **static** **final** **long** stateOffset;
3. **private** **static** **final** **long** runnerOffset;
4. **private** **static** **final** **long** waitersOffset;
5. **static** {
6. **try** {
7. UNSAFE = sun.misc.Unsafe.getUnsafe();
8. Class<?> k = FutureTask.**class**;
9. stateOffset = UNSAFE.objectFieldOffset
10. (k.getDeclaredField("state"));
11. runnerOffset = UNSAFE.objectFieldOffset
12. (k.getDeclaredField("runner"));
13. waitersOffset = UNSAFE.objectFieldOffset
14. (k.getDeclaredField("waiters"));
15. } **catch** (Exception e) {
16. **throw** **new** Error(e);
17. }
18. }

1、获取各个字段的偏移量，用于Unsafe的CAS操作

## 内部类

### WaitNode

1. /\*\*
2. \* Simple linked list nodes to record waiting threads in a Treiber
3. \* stack.  See other classes such as Phaser and SynchronousQueue
4. \* for more detailed explanation.
5. \*/
6. **static** **final** **class** WaitNode {
7. **volatile** Thread thread;
8. **volatile** WaitNode next;
9. WaitNode() { thread = Thread.currentThread(); }
10. }

1、一个封装了线程的节点的链表

## 方法

### 构造方法

1. **public** FutureTask(Runnable runnable, V result) {
2. //Executors.callable是一个静态方法，返回一个RunnableAdapter
3. **this**.callable = Executors.callable(runnable, result);
4. **this**.state = NEW;       // ensure visibility of callable
5. }

1、由于FutureTask内部持有的是Callable，因此必须将Runnable适配成Callable

2、RunnableAdapter详见9.1.3

### run

1. **public** **void** run() {
2. //执行run方法时，FutureTask的状态必须是NEW，并且必须为runner正确赋值
3. **if** (state != NEW ||
4. !UNSAFE.compareAndSwapObject(**this**, runnerOffset,
5. **null**, Thread.currentThread()))
6. **return**;
7. **try** {
8. Callable<V> c = callable;
9. **if** (c != **null** && state == NEW) {
10. V result;
11. **boolean** ran;
12. **try** {
13. //回调callable的call方法
14. //对于用RunnableAdapter封装成的Callable来说，返回的一定是调用RunnableAdapter构造方法时传入的参数，与Runnable的run方法无任何关系
15. result = c.call();
16. ran = **true**;
17. } **catch** (Throwable ex) {
18. result = **null**;
19. ran = **false**;
20. setException(ex);
21. }
22. **if** (ran)
23. set(result);
24. }
25. } **finally** {
26. // runner must be non-null until state is settled to
27. // prevent concurrent calls to run()
28. runner = **null**;
29. // state must be re-read after nulling runner to prevent
30. // leaked interrupts
31. **int** s = state;
32. **if** (s >= INTERRUPTING)
33. handlePossibleCancellationInterrupt(s);
34. }
35. }

1、只有在run方法运行时才能设置其关联的线程，这是必然的

#### set

1. /\*\*
2. \* Sets the result of this future to the given value unless
3. \* this future has already been set or has been cancelled.
4. \*
5. \* <p>This method is invoked internally by the {@link #run} method
6. \* upon successful completion of the computation.
7. \*
8. \* @param v the value
9. \*/
10. **protected** **void** set(V v) {
11. //将NEW通过CAS操作变为COMPLETING，这个过程为什么会失败呢(可能有其他线程在执行cancel将NEW变为CANCELL状态或者INTERRUPTING状态)
12. //失败的话，如何唤醒等待任务结果的线程呢(cancel方法会执行finishCompletion方法，因此会唤醒等待任务结果的线程)
13. **if** (UNSAFE.compareAndSwapInt(**this**, stateOffset, NEW, COMPLETING)) {
14. outcome = v;
15. UNSAFE.putOrderedInt(**this**, stateOffset, NORMAL); // final state
16. finishCompletion();
17. }
18. }

#### setException

1. /\*\*
2. \* Causes this future to report an {@link ExecutionException}
3. \* with the given throwable as its cause, unless this future has
4. \* already been set or has been cancelled.
5. \*
6. \* <p>This method is invoked internally by the {@link #run} method
7. \* upon failure of the computation.
8. \*
9. \* @param t the cause of failure
10. \*/
11. **protected** **void** setException(Throwable t) {
12. //将NEW通过CAS操作变为COMPLETING，这个过程为什么会失败呢(可能有其他线程在执行cancel将NEW变为CANCELL状态或者INTERRUPTING状态)
13. //失败的话，如何唤醒等待任务结果的线程呢(cancel方法会执行finishCompletion方法，因此会唤醒等待任务结果的线程)
14. **if** (UNSAFE.compareAndSwapInt(**this**, stateOffset, NEW, COMPLETING)) {
15. outcome = t;
16. UNSAFE.putOrderedInt(**this**, stateOffset, EXCEPTIONAL); // final state
17. finishCompletion();
18. }
19. }

#### finishCompletion

1. /\*\*
2. \* Removes and signals all waiting threads, invokes done(), and
3. \* nulls out callable.
4. \*/
5. **private** **void** finishCompletion() {
6. // assert state > COMPLETING;
7. //外层循环是为了配合waiter字段的CAS赋值操作
8. **for** (WaitNode q; (q = waiters) != **null**;) {
9. **if** (UNSAFE.compareAndSwapObject(**this**, waitersOffset, q, **null**)) {
10. //一旦进入到这里，那么当内存循环退出时，外层循环必然会退出
11. //遍历等待队列，依次唤醒所有等待线程
12. **for** (;;) {
13. Thread t = q.thread;
14. **if** (t != **null**) {
15. q.thread = **null**;
16. //唤醒该线程
17. LockSupport.unpark(t);
18. }
19. WaitNode next = q.next;
20. **if** (next == **null**)
21. **break**;
22. q.next = **null**; // unlink to help gc
23. q = next;
24. }
25. **break**;
26. }
27. }
29. done();  //该方法在FutureTask中是个空方法，交给子类来实现一些额外的生命周期
31. callable = **null**;        // to reduce footprint
32. }

1、FutureTask所关联的任务执行完毕了，必须唤醒那些等待该任务结果而被阻塞的线程

### cancel

1. **public** **boolean** cancel(**boolean** mayInterruptIfRunning) {
2. //条件成立的情况：
3. //1.state!=NEW
4. //2.根据参数是否允许中断，将状态通过CAS改为INTERRUPTING或CANCELLED，且CAS操作失败
5. **if** (!(state == NEW &&
6. UNSAFE.compareAndSwapInt(**this**, stateOffset, NEW,
7. mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
8. **return** **false**;
9. **try** {    // in case call to interrupt throws exception
10. **if** (mayInterruptIfRunning) {
11. **try** {
12. //获得当前Future关联的线程
13. Thread t = runner;
14. **if** (t != **null**)
15. //进行中断
16. t.interrupt();
17. } **finally** { // final state
18. //为什么要调用这个方法，而不用CAS类或者Volatile类的方法
19. UNSAFE.putOrderedInt(**this**, stateOffset, INTERRUPTED);
20. }
21. }
22. } **finally** {
23. finishCompletion();
24. }
25. **return** **true**;
26. }

### get

1. /\*\*
2. \* @throws CancellationException {@inheritDoc}
3. \*/
4. **public** V get() **throws** InterruptedException, ExecutionException {
5. **int** s = state;
6. //当任务还未完成(正常或者异常完成)时，需要进行等待
7. **if** (s <= COMPLETING)
8. //等待任务完成，将当前访问该结果的线程添加到waiters链表中
9. s = awaitDone(**false**, 0L);
10. **return** report(s);
11. }
13. /\*\*
14. \* @throws CancellationException {@inheritDoc}
15. \*/
16. **public** V get(**long** timeout, TimeUnit unit)
17. **throws** InterruptedException, ExecutionException, TimeoutException {
18. **if** (unit == **null**)
19. **throw** **new** NullPointerException();
20. **int** s = state;
21. **if** (s <= COMPLETING &&
22. (s = awaitDone(**true**, unit.toNanos(timeout))) <= COMPLETING)
23. **throw** **new** TimeoutException();
24. **return** report(s);
25. }

#### awaitDone

1. /\*\*
2. \* Awaits completion or aborts on interrupt or timeout.
3. \*
4. \* @param timed true if use timed waits
5. \* @param nanos time to wait, if timed
6. \* @return state upon completion
7. \*/
8. **private** **int** awaitDone(**boolean** timed, **long** nanos)
9. **throws** InterruptedException {
10. //设定等待时间
11. **final** **long** deadline = timed ? System.nanoTime() + nanos : 0L;
12. WaitNode q = **null**;  //封装当前线程的节点
13. **boolean** queued = **false**;
14. **for** (;;) {
15. //如果当前线程被中断过意味着无论如何都不能再将当前线程添加到等待节点链表中去了，对除去链表中没有关联线程的节点???
16. **if** (Thread.interrupted()) {
17. removeWaiter(q);
18. **throw** **new** InterruptedException();
19. }
21. **int** s = state;
22. //该任务已经结束，因此可以返回了
23. **if** (s > COMPLETING) {
24. **if** (q != **null**)
25. //将该节点关联线程置空
26. q.thread = **null**;
27. **return** s;
28. }
29. //任务即将结束，稍等片刻
30. **else** **if** (s == COMPLETING) // cannot time out yet
31. Thread.yield();
32. //状态为NEW，q尚未初始化，进行初始化，将当前线程封装到WaitNode中
33. **else** **if** (q == **null**)
34. q = **new** WaitNode();
35. //状态为NEW，q已经初始化完毕，但尚未进入等候链表
36. **else** **if** (!queued)
37. queued = UNSAFE.compareAndSwapObject(**this**, waitersOffset,
38. q.next = waiters, q);
39. //状态为NEW，q已经初始化完毕，且已经进入等待链表，且允许等待超时
40. **else** **if** (timed) {
41. nanos = deadline - System.nanoTime();
42. //检查是否已经超过指定的等待时间，若已经超过则直接返回即可
43. **if** (nanos <= 0L) {
44. //当前线程关联的节点已经可以出链表了
45. removeWaiter(q);
46. **return** state;
47. }
48. //阻塞剩余时间
49. LockSupport.parkNanos(**this**, nanos);
50. }
51. **else**
52. //阻塞直至中断或者被唤醒
53. LockSupport.park(**this**);
54. }
55. }

1、正常情况下，请求任务返回值的线程所关联的节点会被添加到等待链表中，等待任务完成被唤醒，或者超时自动唤醒

#### removeWaiter

1. /\*\*
2. \* Tries to unlink a timed-out or interrupted wait node to avoid
3. \* accumulating garbage.  Internal nodes are simply unspliced
4. \* without CAS since it is harmless if they are traversed anyway
5. \* by releasers.  To avoid effects of unsplicing from already
6. \* removed nodes, the list is retraversed in case of an apparent
7. \* race.  This is slow when there are a lot of nodes, but we don't
8. \* expect lists to be long enough to outweigh higher-overhead
9. \* schemes.
10. \*/
11. **private** **void** removeWaiter(WaitNode node) {
12. **if** (node != **null**) {
13. //此方法的目的是将node节点的关联线程置空，然后用下面的过程，将这个置空的节点除去
14. node.thread = **null**;
15. retry:
16. **for** (;;) {          // restart on removeWaiter race
17. **for** (WaitNode pred = **null**, q = waiters, s; q != **null**; q = s) {
18. //以下节点链接操作在并发下是安全的???为什么
19. s = q.next;
20. //q节点关联的线程不为空，将其赋值给pred，也就是说pred若不是nul，则一定指向一个关联有线程的节点
21. **if** (q.thread != **null**)
22. pred = q;
23. //pred不为空，但是q没有关联线程，此时将pred与s相连，相当于将q从链表中除去了
24. **else** **if** (pred != **null**) {
25. pred.next = s;
26. //为什么要检查一下???awaitDone中会将thread字段置为空???
27. //如果重新检查为空了，那么必须继续循环将该节点除去
28. **if** (pred.thread == **null**) // check for race
29. **continue** retry;
30. }
31. //运行到这里，情况应该是q关联的线程为空，并且pred为空，因此s节点应当作为表头，用CAS操作替换链表头，若失败了，说明至少有两个线程在执行。则重新回到大循环
32. **else** **if** (!UNSAFE.compareAndSwapObject(**this**, waitersOffset,
33. q, s))
34. **continue** retry;
35. }
36. **break**;
37. }
38. }
39. }

### report

1. /\*\*
2. \* Returns result or throws exception for completed task.
3. \*
4. \* @param s completed state value
5. \*/
6. @SuppressWarnings("unchecked")
7. **private** V report(**int** s) **throws** ExecutionException {
8. Object x = outcome;
9. **if** (s == NORMAL)
10. **return** (V)x;
11. **if** (s >= CANCELLED)
12. **throw** **new** CancellationException();
13. **throw** **new** ExecutionException((Throwable)x);
14. }

# 神奇的位运算

## 判断是否是2的幂次

1. **if** ((x& (x- 1)) != 0)

## 模2的幂次

1. (x-1)&y

1、其中x是2的幂次，y是任意整数

## 找出不小于给定值得最小2的幂次

1. **private** **static** final **int** tableSizeFor(**int** c) {
2. **int** n = c - 1;
3. n |= n >>> 1;
4. n |= n >>> 2;
5. n |= n >>> 4;
6. n |= n >>> 8;
7. n |= n >>> 16;
8. **return** (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;
9. }