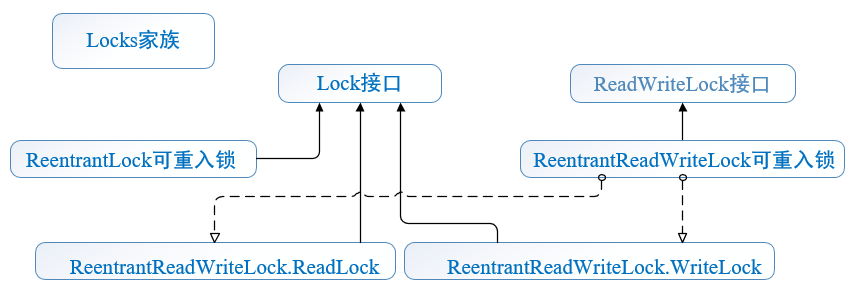
java.util.concurrent.locks包介绍

# java.util.concurrent.locks包-锁包

## 锁locks

java.util.concurrent.locks包package是java.util.concurrent包的一个子包，这个包主要定义了锁的一些接口和实现类。

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.



since:1.5

## 接口与类

java.util.concurrent.locks锁包中定义了**3个接口**：Lock、ReadWriteLock、Condition。

java.util.concurrent.locks锁包中定义了9个类：

AbstractOwnableSynchronizer、AbstractQueuedLongSynchronizer、AbstractQueuedSynchronizer、LockSupport、**ReentrantLock**、**ReentrantReadWriteLock**、**ReentrantReadWriteLock.ReadLock**、**ReentrantReadWriteLock.WriteLock**、StampedLock。

**最重要的锁实现类是：ReentrantLock、ReentrantReadWriteLock**。

## locks包描述

Interfaces and classes providing a framework for **locking** and **waiting** for conditions that is distinct from **built-in synchronization and monitors**. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.

**Lock**:The **Lock** interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is **ReentrantLock**.

**ReadWriteLock**:The **ReadWriteLock** interface similarly defines locks that may be shared among readers but are exclusive to writers. Only a single implementation, **ReentrantReadWriteLock**, is provided, since it covers most standard usage contexts. But programmers may create their own implementations **to cover nonstandard requirements**.

**Condition**:The **Condition** interface describes condition variables that may be associated with Locks. These are similar in usage to the implicit monitors accessed using Object.wait, but offer extended capabilities. In particular, multiple **Condition** objects may be associated with a single Lock. To avoid compatibility issues, the names of Condition methods are different from the corresponding Object versions.

The **AbstractQueuedSynchronizer** class serves as a useful superclass for defining locks and other synchronizers that rely on queuing blocked threads. The **AbstractQueuedLongSynchronizer** class provides the same functionality but extends support to 64 bits of synchronization state. Both extend class **AbstractOwnableSynchronizer**, a simple class that helps record the thread currently holding exclusive synchronization. The **LockSupport** class provides lower-level blocking and unblocking support that is useful for those developers implementing their own customized lock classes.

# Lock接口

## Lock接口

Lock接口存在于java.util.concurrent.locks包中，有3个实现类：**ReentrantLock**, **ReentrantReadWriteLock**.**ReadLock**, **ReentrantReadWriteLock**.**WriteLock。**

## 功能介绍

Lock implementations provide **more extensive locking operations** than can be obtained using synchronized methods and statements. They allow **more flexible structuring**, may have quite different properties, and may support multiple **associated Condition objects**.

Lock接口的实现类比同步方法或语句具有更多扩展操作，具有更灵活的结构，更多不同的属性，支持多个关联的**Condition对象**。

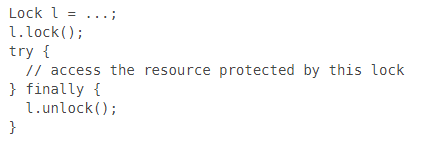
A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a **ReadWriteLock**.(但是，一些锁允许对共享资源的并发访问，如ReadWriteLock中的读操作)

The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur **in a block-structured way**: when multiple locks are acquired they must be released in the opposite order, and all locks must be released **in the same lexical scope** in which they were acquired.

While **the scoping mechanism** for synchronized methods and statements makes it much easier to program **with monitor locks,** and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. For example, some algorithms for traversing concurrently accessed data structures require the use of "**hand-over-hand(手牵手)**" or "**chain locking(链式锁)**": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

## 使用示例

**With this increased flexibility** comes **additional responsibility**. The absence of **block-structured locking** removes the automatic release of locks that occurs with synchronized methods and statements. In most cases, the following idiom should be used:



When **locking and unlocking** occur in different scopes, care must be taken to ensure that all code that is executed while the lock is held is protected by **try-finally or try-catch** to ensure that the lock is released when necessary.



获取锁之后，记得释放锁，且最好在**finally**中释放锁。

## 其他功能

### tryLock

Lock implementations provide **additional functionality** overthe use of **synchronized methods and statements** by providing a non-blocking attempt to acquire a lock (**tryLock()),** an attempt to acquire the lock that can be interrupted (**lockInterruptibly**(), and an attempt to acquire the lock that can timeout (**tryLock**(long, TimeUnit)).

A Lock class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering, non-reentrant usage, or deadlock detection. If an implementation provides such specialized semantics(语义) then the implementation must document those semantics.

Note that **Lock** instances are just normal objects and can themselves be used as the target in a synchronized statement. Acquiring the monitor lock of a Lock instance has no specified relationship with invoking any of the lock() methods of that instance. It is recommended that to avoid confusion you never use **Lock** instances in this way, except within their own implementation.

Except where noted, passing a **null** value for any parameter will result in a **NullPointerException** being thrown.

## Memory Synchronization

All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in The Java Language Specification (17.4 Memory Model):

### A successful lock operation has the same memory synchronization effects as a successful Lock action.

### A successful unlock operation has the same memory synchronization effects as a successful Unlock action.

Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

## Implementation Considerations

The three forms of lock acquisition(获得) (**interruptible, non-interruptible, and timed)** may differ in their performance characteristics, ordering guarantees, or other implementation qualities. Further, the ability to interrupt the ongoing acquisition of a lock may not be available in a given **Lock** class. Consequently, an implementation is not required to define exactly the same guarantees or semantics for all three forms of lock acquisition, nor is it required to support interruption of an ongoing lock acquisition. An implementation is required to clearly document the semantics and guarantees provided by each of the locking methods. It must also obey the interruption semantics as defined in this interface, **to the extent** that interruption of lock acquisition is supported: which is either totally, or only on method entry.

As interruption generally implies **cancellation**, and **checks** for interruption are often infrequent, an implementation can favor responding to an interrupt over normal method return. This is true even if it can be shown that the interrupt occurred after another action may have unblocked the thread. An implementation should document this behavior.

## 方法介绍

主要是**lock与unlock、trylock、lockInterruptibly、newCondition**。

### lock() 与 unlock()

void lock() Acquires the lock.获取锁

void unlock() Releases the lock.释放锁

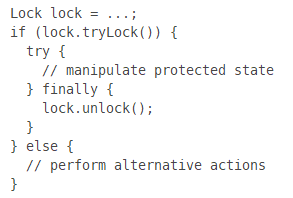
### tryLock() 与tryLock(long time, TimeUnit unit)

boolean **tryLock**() Acquires the lock **only if** it is free at the time of invocation.

Acquires the lock if it is available and returns immediately with the value **true**. If the lock is not available then this method will return immediately with the value **false**.

Returns: **true** if the lock was acquired and **false** otherwise

使用示例：



This usage ensures that the lock is unlocked if it was acquired, and doesn't try to unlock if the lock was not acquired.

boolean **tryLock**(long time, TimeUnit unit) throws InterruptedException

Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

Parameters: **time** - the maximum time to wait for the lock；

**unit** - the time unit of the time argument

**Returns: true** if the lock was acquired and **false** if the waiting time elapsed before the lock was acquired**.**

### lockInterruptibly()

void **lockInterruptibly**()

Acquires the lock unless the current thread is interrupted.

Acquires the lock if it is available and returns immediately.

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

* The lock is acquired by the current thread; or
* Some other thread interrupts the current thread, and interruption of lock acquisition is supported.

If the current thread:

* has its interrupted status set on entry to this method; or
* is interrupted while acquiring the lock, and interruption of lock acquisition is supported,

then InterruptedException is thrown and the current thread's interrupted status is cleared.

### newCondition()

Condition newCondition()

**Returns a new Condition instance** that is bound to this Lock instance.

Before waiting on the condition the lock must be held by the current thread. A call to **Condition.await()** will atomically release the lock before waiting and re-acquire the lock before the wait returns.

## Lock接口的源代码

package java.util.concurrent.locks;

import java.util.concurrent.TimeUnit;

public interface Lock {

void lock();

void unlock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

**Condition** newCondition();

}

# ReentrantLock可重入锁

reentrant 英 [riː'entrənt] 美 n. 凹角；再进入 adj. 再进去的；凹角的

**reentrant functions** 可重入函数

**可重入**：如果一个函数在同一时刻可以被多个线程安全地调用，就称该函数是**线程安全**的。线程安全函数解决多个线程调用函数时访问共享资源的冲突问题。 **可重入(Reentrant)：**函数可以由多于一个线程并发使用，而不必担心数据错误。可重入函数可以在任意时刻被中断，稍后再继续运行，不会丢失数据。

## ReentrantLock可重入锁

public class **ReentrantLock** extends Object implements **Lock, Serializable**

ReentrantLock可重入锁存在于java.util.concurrent.locks包中，实现了2个接口：Serializable, Lock。

## 功能介绍

A reentrant mutual exclusion **Lock** with the same basic behavior and semantics(语义) as the implicit monitor lock accessed using synchronized methods and statements, **but with extended capabilities.**

A **ReentrantLock** is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods **isHeldByCurrentThread()**, and **getHoldCount()**.

The constructor for this class accepts **an optional fairness parameter**. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. **Note however**, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed **tryLock()** method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

In addition to implementing *the Lock interface*, this class defines a number of public and protected methods **for inspecting the state of the lock**. Some of these methods are only useful for instrumentation and monitoring.

Serialization of this class behaves in the same way as **built-in locks**: a deserialized lock is in the unlocked state, **regardless of** its state when serialized.

This lock supports a maximum of **2147483647 recursive locks** by the same thread. Attempts to exceed this limit result in Error throws from locking methods.

## 构造方法

**ReentrantLock**()

Creates an instance of ReentrantLock. This is equivalent to using ReentrantLock(false).

**ReentrantLock**(boolean fair)

Creates an instance of ReentrantLock with the given fairness policy.

Parameters: fair - true if this lock should use a fair ordering policy

公平策略的含义：

公平策略：确保让**等待时间最长**的线程(the longest-waiting thread)获取锁。

The constructor for this class **accepts an optional fairness parameter**. ①**When set true,** under contention, locks favor granting access to the longest-waiting thread. ②**Otherwise** this lock does not guarantee any particular access order.

Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. **Note however**, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed **tryLock**() method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

## 方法介绍

除了实现Lock接口规定的方法外，还有很多扩展方法，**一部分是protected方法，用来监视lock锁的状态**。具体见API。

In addition to implementing the Lock interface, this class defines a number of public and protected methods **for inspecting the state of the lock**. Some of these methods are only useful for instrumentation and monitoring.

### Lock接口的方法：lock与unlock、trylock、lockInterruptibly、newCondition

**lockInterruptibly**(): Acquires the lock unless the current thread is interrupted.

**newCondition()**:Returns a Condition instance for use with this Lock instance.

### getQueuedThreads:获取线程队列

protected Collection<Thread> getQueuedThreads()

Returns a collection containing threads that may be waiting to acquire this lock.

### getWaitingThreads(Condition condition)

protected Collection<Thread> getWaitingThreads(**Condition** condition)

Returns a collection containing those threads that may be waiting on the given condition associated with this lock.

int getWaitQueueLength(**Condition** condition)

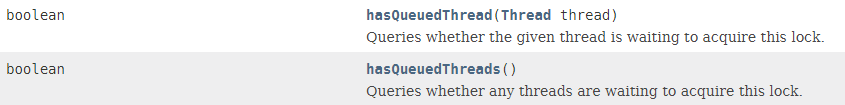
Returns an estimate of the number of threads waiting on the given condition associated with this lock.

### public final int getQueueLength()

Returns an estimate of the number of threads waiting to acquire this lock. The value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use **in monitoring of the system state**, **not for synchronization control.**

Returns: the estimated number of threads waiting for this lock

### hasQueuedThread(Thread thread)：判断线程队列中是否含有某个线程



### protected Thread getOwner()

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

Returns: the owner, or null if not owned

### public int getHoldCount()

查询当前线程保持此锁的次数，也就是执行此线程执行lock方法的次数。

Queries the number of holds on this lock by the current thread.

A thread has a hold on a lock for each lock action that is not matched by an unlock action.

The hold count information is typically only used **for testing and debugging purposes**. For example, if a certain section of code should not be entered with the lock already held then we can assert that fact:

class X {

ReentrantLock lock = new ReentrantLock();

// ...

public void m() {

assert lock.getHoldCount() == 0;

lock.lock();

try {

// ... method body

} finally {

lock.unlock();

}

}

}

Returns: the number of holds on this lock by the current thread, or zero if this lock is not held by the current thread.

### public boolean isHeldByCurrentThread()

Queries if this lock is held by the current thread.

Returns: true if current thread holds this lock and false otherwise

### public final boolean isFair()

Returns true if this lock has fairness set true.

### public boolean isLocked()

**Queries if this lock is held by any thread**. This method is designed for use in monitoring of the system state, not for synchronization control.

Returns: true if any thread holds this lock and false otherwise

getHoldCount() 查询当前线程保持此锁的次数，也就是执行此线程执行lock方法的次数

getQueueLength（）返回正等待获取此锁的线程估计数，比如启动10个线程，1个线程获得锁，此时返回的是9

getWaitQueueLength（Condition condition）返回等待与此锁相关的给定条件的线程估计数。比如10个线程，用同一个condition对象，并且此时这10个线程都执行了condition对象的await方法，那么此时执行此方法返回10

hasWaiters(Condition condition)查询是否有线程等待与此锁有关的给定条件(condition)，对于指定contidion对象，有多少线程执行了condition.await方法

hasQueuedThread(Thread thread)查询给定线程是否等待获取此锁

hasQueuedThreads()是否有线程等待此锁

isFair()该锁是否公平锁

isHeldByCurrentThread() 当前线程是否保持锁锁定，线程的执行lock方法的前后分别是false和true

isLock()此锁是否有任意线程占用；

lockInterruptibly（）如果当前线程未被中断，获取锁；

tryLock（）尝试获得锁，仅在调用时锁未被线程占用，获得锁；

tryLock(long timeout TimeUnit unit)如果锁在给定等待时间内没有被另一个线程保持，则获取该锁；

## tryLock和lock和lockInterruptibly的区别

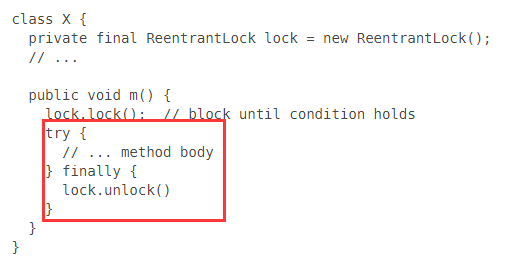
tryLock能获得锁就返回true，不能就立即返回false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回false；

**lock**能获得锁就返回true，不能的话一直等待获得锁；

**lock和lockInterruptibly**，如果两个线程分别执行这两个方法，但此时中断这两个线程，前者不会抛出异常，而后者会抛出异常。

## 典型使用示例

It is recommended practice to always immediately follow a call to lock with a try block, most typically in a before/after construction such as: :(**try—finally**)



# ReentrantLock的源码分析

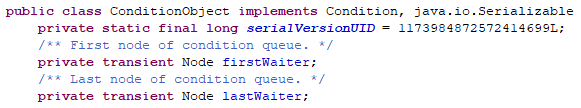
在ReentrantLock中与AbstractQueuedSynchronizer有很大的关系。在ReentrantLock中定义了一个抽象内部类Sync,该类是ReentrantLock的核心。

abstract static class **Sync** extends **AbstractQueuedSynchronizer。**

**AbstractQueuedSynchronizer中定义了两个重要的内部类Node和ConditionObject。**

public class **ConditionObject** implements **Condition**, java.io.Serializable

在ConditionObject的内部封装了两个Node对象firstWaiter和lastWaiter。



static final class **Node，其源码为：**

static final class Node {

/\*\* Marker to indicate a node is waiting in shared mode \*/

static final Node SHARED = **new Node();**

/\*\* Marker to indicate a node is waiting in exclusive mode \*/

static final Node EXCLUSIVE = null;

/\*\* waitStatus value to indicate thread has cancelled \*/

static final int CANCELLED = 1;

/\*\* waitStatus value to indicate successor's thread needs unparking \*/

static final int SIGNAL = -1;

/\*\* waitStatus value to indicate thread is waiting on condition \*/

static final int CONDITION = -2;

/\*\*

\* waitStatus value to indicate the next acquireShared should

\* unconditionally propagate

\*/

static final int PROPAGATE = -3;

volatile int waitStatus;

**volatile Node prev;**

**volatile Node next;**

/\*\*

\* **The thread that enqueued this node**. Initialized on

\* construction and nulled out after use.

\*/

volatile Thread thread;

**Node nextWaiter;**

/\*\*

\* Returns true if node is waiting in shared mode.

\*/

final boolean **isShared**() {

return nextWaiter == SHARED;

}

/\*\*

\* Returns previous node, or throws NullPointerException if null.

\* Use when predecessor cannot be null. The null check could

\* be elided, but is present to help the VM.

\* @return the predecessor of this node

\*/

final Node **predecessor**() throws NullPointerException {

Node p = prev;

if (p == null)

throw new NullPointerException();

else

return p;

}

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

}

Node(Thread thread, Node mode) { // Used by addWaiter

this.nextWaiter = mode;

this.thread = thread;

}

Node(Thread thread, int waitStatus) { // Used by Condition

this.waitStatus = waitStatus;

this.thread = thread;

}

}

# Condition接口

## 继承关系

存在于java.util.concurrent.locks包中。

public interface **Condition**

All Known Implementing Classes:

AbstractQueuedLongSynchronizer.ConditionObject, AbstractQueuedSynchronizer.ConditionObject

## 功能介绍

Condition factors out the Object **monitor** methods (**wait, notify and notifyAll**) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. **Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.(Lock取代了同步方法或同步语句，Condition取代了对象锁)。**

**Conditions** (also known as condition queues or condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true. Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition. The key property that waiting for a condition provides is that it atomically releases the associated lock and suspends the current thread, just like Object.wait.

**A Condition implementation** can provide behavior and semantics that is different from that of the Object monitor methods, such as guaranteed ordering for notifications, or not requiring a lock to be held when performing notifications. If an implementation provides such specialized semantics then the implementation must document those semantics.

**Note** that Condition instances are just normal objects and can themselves be used as the target in a synchronized statement, and can have *their own monitor wait and notification methods invoked*. Acquiring the monitor lock of a Condition instance, or using its monitor methods, has no specified relationship with acquiring the Lock associated with that Condition or the use of its waiting and signalling methods. It is recommended that to avoid confusion you never use ***Condition*** instances in this way, except perhaps within their own implementation.

Except where noted, passing a null value for any parameter will result in a NullPointerException being thrown.

## 如何创建*Condition*实例？

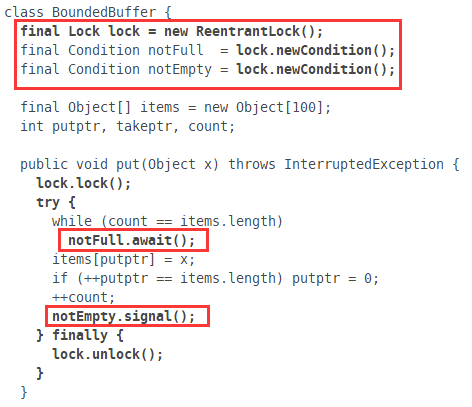
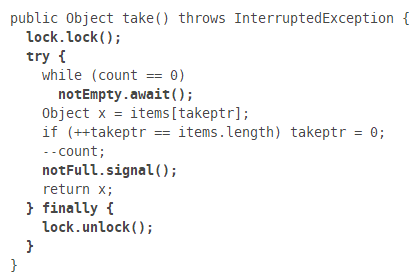
**A Condition instance is intrinsically bound to a lock**. To obtain a Condition instance for a particular Lock instance use its **newCondition()** method.

***Condition condition = lock.newCondition();***

注意：**Condition**只有两类方法await和signal，分别对应Object的wait和notify方法，只有获取了对应的Lock锁之后，才能调用Condition的**await和signal**方法。

## 使用场景：查看数组阻塞队列ArrayBlockingQueue就是这种实现。

As an example, suppose we **have a bounded buffer** which supports **put and take** methods. If a take is attempted on an empty buffer, then the thread will block until an item becomes available; if a put is attempted on a full buffer, then the thread will block until a space becomes available. We would like to keep waiting put threads and take threads in separate wait-sets so that we can use the optimization of only **notifying** a single thread at a time when items or spaces become available in the buffer. **This can be achieved using two Condition instances**.

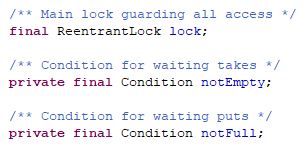
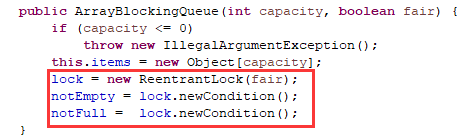
使用一把锁**Lock对象**，两个Condition分别为***notEmpty***和***notFull***。

(The ***ArrayBlockingQueue*** class provides this functionality, so there is no reason to implement this sample usage class.)

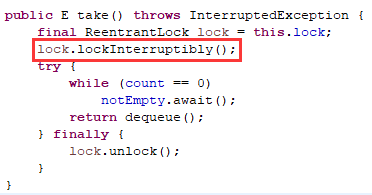
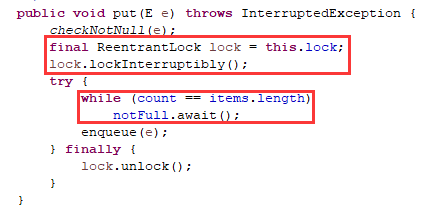
## ArrayBlockingQueue数组阻塞队列



只定义了一把锁，也就是说在同一时刻要么take要么put，不能同时put和take。

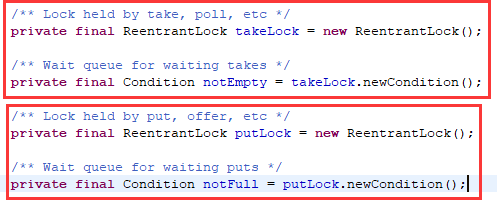
**take()和put(E e)**方法：

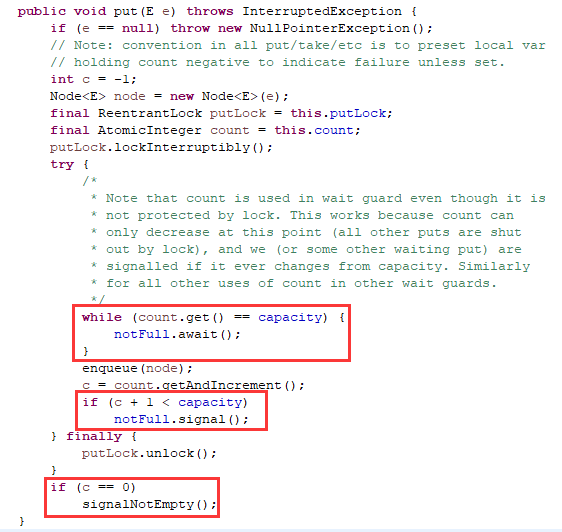
 

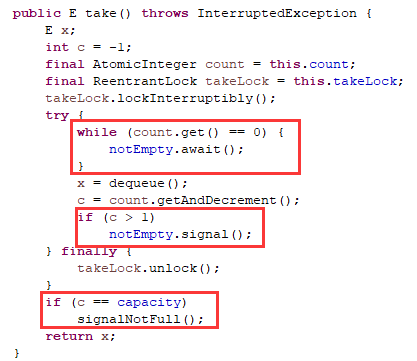
## 链表阻塞队列LinkedBlockingQueue中Condition的使用

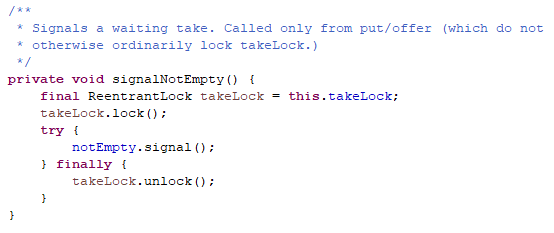
利用Condition实例对象来控制队列的put和take。当队列为空时，take需要阻塞，持有takeLock锁，Condition为notEmpty，当队列已满时，put需要阻塞，持有putLock锁，Condition为notFull。

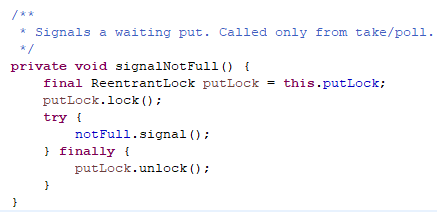
定义了两个Lock，一个takeLock，一个是putLock，说明在同一时刻可以同时take和put。putLock上绑定的就是**notFull**，takeLock上绑定的是**notEmpty**。











## 接口方法：两大类await和signal

### await与awaitXxx

通过创建Condition对象来使线程await，必须先执行**lock.lock方法**获得锁。

void **await**()

Causes the current thread to wait until it is signalled or interrupted.

boolean **await**(long time, TimeUnit unit)

Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

long **awaitNanos**(long nanosTimeout)

Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

void **awaitUninterruptibly**()

Causes the current thread to wait until it is signalled.

boolean **awaitUntil**(**Date deadline**)

Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses

### signal

void **signal**()： Wakes up one waiting thread.

void **signalAll**()：Wakes up all waiting threads.

# ReentrantLock的公平锁与非公平锁

Lock lock = new ReentrantLock(true);//公平锁

Lock lock = new ReentrantLock(false);//非公平锁(无参也是默认false)

**公平锁**指的是线程获取锁的顺序是**按照加锁顺序**来的，而非公平锁指的是**抢锁机制**，先lock的线程不一定先获得锁。

# ReentrantLock与synchronized的区别

一个condition对象的**signal（signalAll）**方法和该对象的**await方法**是一一对应的，也就是一个condition对象的**signal（signalAll）**方法不能唤醒**其他condition对象**的await方法

**ReentrantLock类可以**唤醒**指定条件**(特定Condition)的线程，而object的唤醒是**随机**的。

**Condition类和Object类**

Condition类的**awiat方法**和Object类的**wait方法**等效；

Condition类的**signal方法**和Object类的**notify方法**等效；

Condition类的**signalAll方法**和Object类的**notifyAll方法**等效；

## 和synchronized的使用

效果和synchronized一样，都可以同步执行，lock方法获得锁，unlock方法释放锁。

synchronized (new byte[0]) {

//coding

}

ReentrantLock lock = new ReentrantLock();

try{

lock.lock();

//coding

System.out.println("代码");

}finally{

lock.unlock();

}

## 见ArrayBlockingQueue的源代码。

# 总结

* Lock也可以实现线程同步，执行lock方法获得锁，执行unLock方法释放锁；
* **Lock**类可以创建**Condition**对象，Condition对象用来是线程等待和唤醒线程，需要注意的是Condition对象的唤醒的是用同一个Condition执行await方法的线程，所以也就可以实现唤醒指定类的线程；
* Lock类分公平锁和不公平锁，公平锁是按照加锁顺序来的，非公平锁是不按顺序的，也就是说先执行lock方法的锁不一定先获得锁；
* ReadWriteLock有读锁和写锁，读读共享，写写互斥，读写互斥。

# ReentrantLock源码分析: AbstractQueuedSynchronized

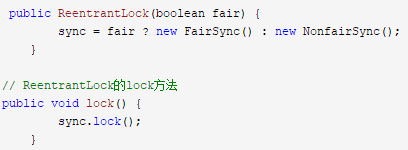
## ReentrantLock<==>AbstractQueuedSynchronized(FIFO)<==>Node

ReentrantLock通过**同步器AQS**（**AbstractQueuedSynchronized类**）来实现的，AQS根本上是通过一个**双向队列**来实现的；将**线程**构造成一个**节点Node**，一个线程先尝试获得锁，如果获取锁失败，就将该线程加到**队列尾部**;

**AbstractQueuedSynchronizer** :Provides a **framework** for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on **first-in-first-out (FIFO) wait queues**.

## lock方法原理

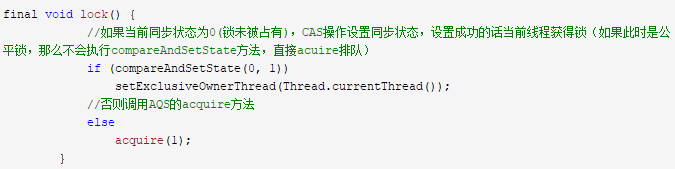
非公平锁的**lock方法**,调用的**sync（NonfairSync和fairSync的父类）的lock方法，多态的使用：FairSync和NonfairSync。**

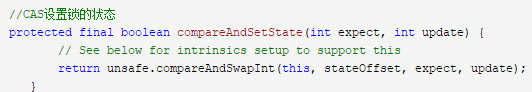


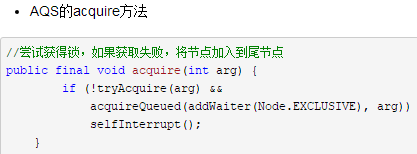
## AbstractQueuedSynchronized源码分析：



**NonfairSync**的**lock**方法，acquire的是Sync的父类AQS的acquire方法。

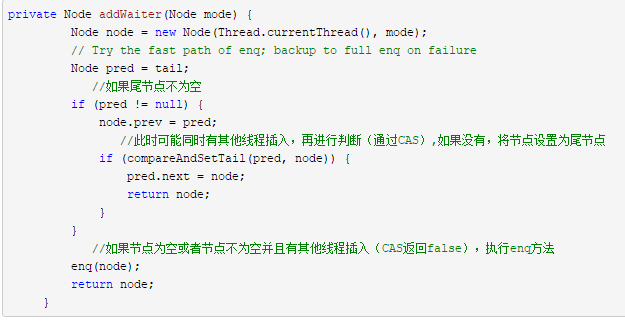




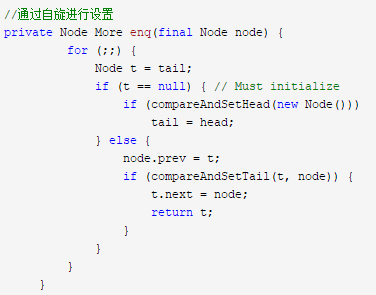




**如果获取锁失败，将节点加入尾节点。**



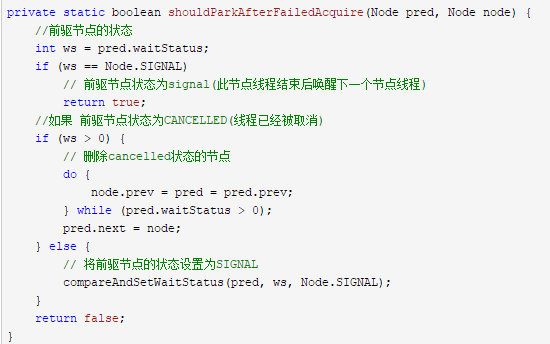
如果节点为空或者节点不为空并且有其他线程插入（CAS返回false），执行enq



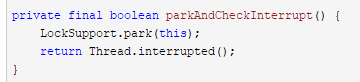
进入队列的线程尝试获得锁：



线程是否可以挂起



挂起当前线程,返回线程中断状态并重置



## Node的定义：将线程定义成一个Node

static final class Node {

/\*\* Marker to indicate a node is waiting in shared mode \*/

static final Node SHARED = new Node();

/\*\* Marker to indicate a node is waiting in exclusive mode \*/

static final Node EXCLUSIVE = null;

/\*\* waitStatus value to indicate thread has cancelled \*/

static final int CANCELLED = 1;

/\*\* waitStatus value to indicate successor's thread needs unparking \*/

static final int SIGNAL = -1;

/\*\* waitStatus value to indicate thread is waiting on condition \*/

static final int CONDITION = -2;

/\*\*

\* waitStatus value to indicate the next acquireShared should

\* unconditionally propagate

\*/

static final int PROPAGATE = -3;

**volatile int waitStatus;**

**volatile Node prev;**

**volatile Node next;**

/\*\*

\* The thread that enqueued this node. Initialized on

\* construction and nulled out after use.

\*/

**volatile Thread thread;**

/\*\*

\* Link to next node waiting on condition, or the special

\* value SHARED. Because condition queues are accessed only

\* when holding in exclusive mode, we just need a simple

\* linked queue to hold nodes while they are waiting on

\* conditions. They are then transferred to the queue to

\* re-acquire. And because conditions can only be exclusive,

\* we save a field by using special value to indicate shared

\* mode.

\*/

Node nextWaiter;

/\*\*

\* Returns true if node is waiting in shared mode.

\*/

final boolean isShared() {

return nextWaiter == SHARED;

}

/\*\*

\* Returns previous node, or throws NullPointerException if null.

\* Use when predecessor cannot be null. The null check could

\* be elided, but is present to help the VM.

\*

\* @return the predecessor of this node

\*/

final Node predecessor() throws NullPointerException {

Node p = prev;

if (p == null)

throw new NullPointerException();

else

return p;

}

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

}

Node(Thread thread, Node mode) { // Used by addWaiter

this.nextWaiter = mode;

this.thread = thread;

}

Node(Thread thread, int waitStatus) { // Used by Condition

this.waitStatus = waitStatus;

this.thread = thread;

}

}