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.

## 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

# 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**.

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**" or "**chain locking**": 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.

## 方法介绍

### lock()

void lock() Acquires the lock.获取锁

### unlock()

void unlock() Releases the lock.释放锁

### tryLock()

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

### tryLock(long time, TimeUnit unit)

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

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

### lockInterruptibly()

void **lockInterruptibly**()

Acquires the lock unless the current thread is interrupted.

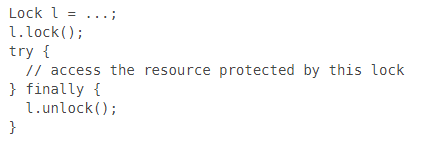
### newCondition()

Condition newCondition()

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

## 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。

# 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 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()**.

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.

**ReentrantLock**(boolean fair)

Creates an instance of ReentrantLock with the given fairness policy.

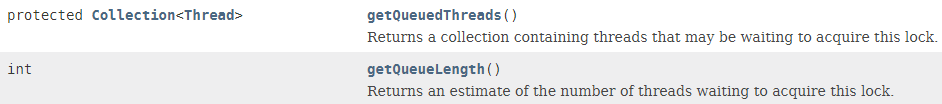
The constructor for this class **accepts an optional fairness parameter**. ①**When set true,** under contention, locks favor granting access to the longest-waiting thread. ②**Otherwise** this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display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

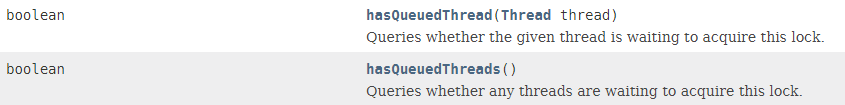
除了实现Lock接口规定的方法外，还有很多扩展方法，一部分是protected方法，用来监视lock锁的状态。具体见API。

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.

### getQueueThreads:获取线程队列

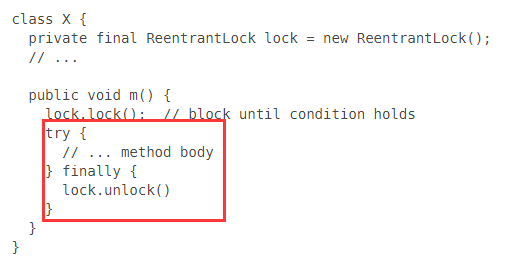


### hasQueuedThread(Thread thread)：判断线程队列中是否含有某个线程



## 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:



# ReadWriteLock接口--读写锁

## ReadWriteLock接口

public interface ReadWriteLock

ReadWriteLock接口与Lock接口**并列**(不是继承哦)，存在于java.util.concurrent.locks包中，有1个实现类：**ReentrantReadWriteLock**。

## 功能介绍

A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

All **ReadWriteLock** implementations must **guarantee** that the memory synchronization effects of writeLock operations (as specified in the Lock interface) also hold with respect to the associated readLock. **That is, a thread successfully acquiring the read lock will see all updates made upon previous release of the write lock**.

A read-write lock allows for **a greater level of concurrency** in accessing shared data than that permitted by a mutual exclusion lock. It exploits the fact that while only a single thread at a time (a writer thread) can modify the shared data, in many cases any number of threads can concurrently read the data (hence reader threads). In theory, the increase in concurrency permitted by the use of a read-write lock will lead to performance improvements over the use of a mutual exclusion lock. In practice this increase in concurrency will only be fully realized on a multi-processor, and then only if the access patterns for the shared data are suitable.

## 接口方法2个

ReadWriteLock接口只规定了2个方法：readLock()和writeLock()方法。

Lock **readLock**() Returns the lock used for reading.

Lock **writeLock**() Returns the lock used for writing.

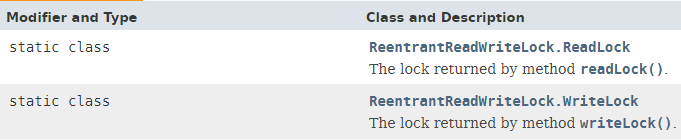
# ReentrantReadWriteLock

## ReentrantReadWriteLock简单介绍

public class **ReentrantReadWriteLock** extends **Object** implements **ReadWriteLock**, **Serializable.**

An implementation of **ReadWriteLock** supporting similar semantics to **ReentrantLock**.

## 内部包装类

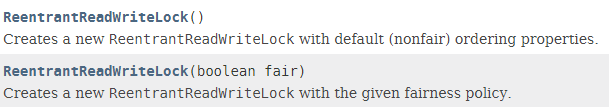


通过readLock()和writeLock()方法获取：





## 构造方法



## 方法

方法与ReentrantLock实现的方法类似。

## 使用示例

