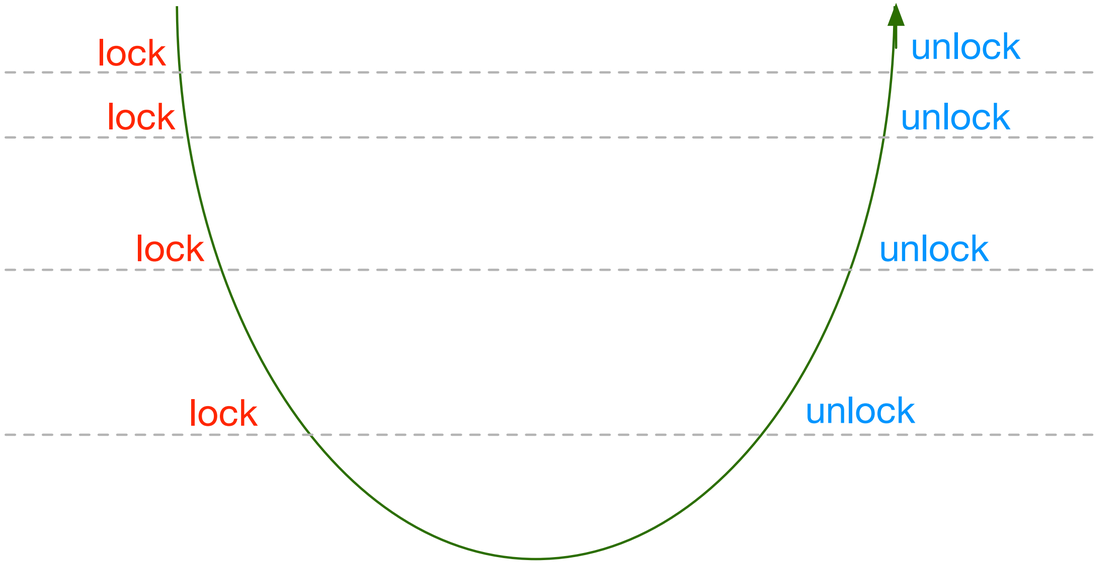
可重入锁

ReentrantLock

java.util.concurrent.locks.ReentrantLock



- 先加锁(lock)、后解锁(unlock)
- 可多次重复加锁，多次加锁需多次解锁
- 有公平锁（按等待次序获得锁）选项，默认非公平锁

构造函数	ReentrantLock()	
	ReentrantLock(boolean fair)	是否公平锁
加锁	lock()	获得锁，否则一直等待
	lockInterruptibly()	获得锁，或者线程被中断(InterruptedException)
	tryLock()	立即返回，true表示已经获得锁
	tryLock(long timeout, TimeUnit unit)	如果获得锁，马上返回；如果没获得锁，尝试等待一段时间，在此时间内如果获得锁，立即返回；超时返回false
释放锁	unlock	解锁

```
public class LockTest {
    private ReentrantLock lock = new ReentrantLock();

    public void m1(){
        System.out.println("1.    [" + Thread.currentThread().getName() + "]");
①       lock.lock();
        try {
            System.out.println("2.    [" + Thread.currentThread().getName() + "]");
            m2();
           System.out.println("3.    [" + Thread.currentThread().getName() + "]");
        }finally {
②           lock.unlock();
            System.out.println("4.    [" + Thread.currentThread().getName() + "]");
        }
    }

    public void m2(){
        System.out.println("5.    [" + Thread.c
②       lock.lock();
        try {
            System.out.println("6.    [" + Thre
        }finally {
①           lock.unlock();
            System.out.println("7.    [" + Thre
        }
    }
}
```

T-1: 1 2 5 6 7 3 4

T-2: 5 6 7

```
public static void main(String[] argvs){
    final LockTest lt = new LockTest();
    new Thread(() -> {
        lt.m1();
    }, "T-1").start();
    new Thread(() -> {
        lt.m2();
    }, "T-2").start();
}
```

```java
public void m1() throws InterruptedException{
    lock.lock();
     try {
        m2();
    }finally {
         lock.unlock();
    }
}

public void m2() throws InterruptedException{
    lock.lock();
     try {
        Thread.sleep(100);
     }finally {
         lock.unlock();
    }
}
```

```java
public void m1() throws InterruptedException{
    synchronized (this){
        m2();
    }
}

public void m2() throws InterruptedException{
    synchronized (this){
        Thread.sleep(100);
     }
}
```

Lock比synchronized多的功能

- 公平锁
- 等待加锁的过程可中断, (synchronized不可）
- 可实现共享锁和独占锁
- Condition
- 查询锁状态

建议

```
lock.lock();
try {
    // 程序处理部分
}finally {
    lock.unlock();
}
```

用try{} finally{lock.unlock();}写法，确保unlock被执行