锁

1. Lock接口

Lock简介

锁是用于通过多个线程控制对共享资源的访问的工具。通常，锁提供对共享资源的独占访问：一次只能有一个线程可以获取锁，并且对共享资源的所有访问都要求首先获取锁。 但是，一些锁可能允许并发访问共享资源，如ReadWriteLock的读写锁。

在Lock接口出现之前，Java程序是靠synchronized关键字实现锁功能的。JDK1.5之后并发包中新增了Lock接口以及相关实现类来实现锁功能。

为什么synchronized不够用？

1. 效率低:锁的释放情况少、试图获得锁时不能设定超时、**不能中断一个正在试图获得锁的线程**
2. 不够灵活(读写锁更灵活):加锁和释放的时机单一,每个锁仅有单一的条件(某个对象),可能是不够的
3. 无法知道是否成功获取到锁。
4. **Lock接口提供的synchronized关键字不具备的主要特性：**

特性	描述
尝试非阻塞地获取锁	当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁
能被中断地获取锁	获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放
超时获取锁	在指定的截止时间之前获取锁， 超过截止时间后仍旧无法获取则返回

Lock接口的实现类： ReentrantLock， ReentrantReadWriteLock.ReadLock，ReentrantReadWriteLock.WriteLock

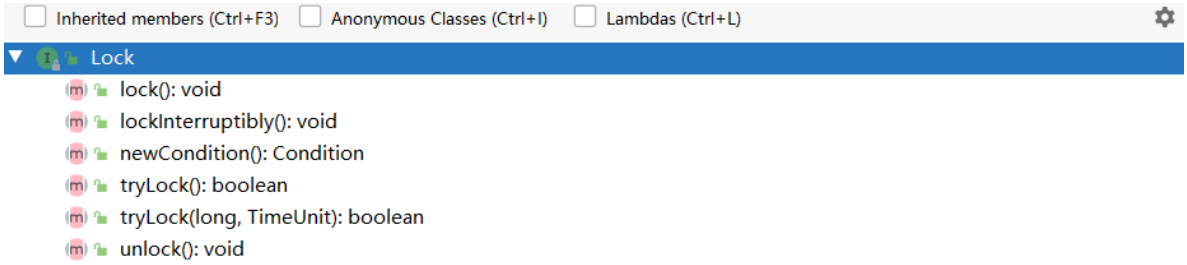
Lock的简单使用

```
Lock lock=new ReentrantLock();
    lock.lock();
try{
    //
}finally{
    lock.unlock();
}
```

注意： 最好不要把获取锁的过程写在try语句块中，因为如果在获取锁时发生了异常，异常抛出的同时也会导致锁无法被释放。

1.3 Lock接口的特性和常见方法

Lock接口基本的方法：



方法名称	描述
void lock()	获得锁。如果锁不可用，则当前线程将被禁用以进行线程调度，并处于休眠状态，直到获取锁。**Lock不会像 synchronized一样在异常时自动释放锁**，因此最佳实践是,**在 finally中释放锁**,以保证发生异常时锁一定被释放
void lockInterruptibly()	此方法相当于 try Lock(long time, Timeunit unit）把超时时间设置为无限。在等待锁的过程中,线程可以被中断获取锁，如果可用并**立即返回。**如果锁不可用，那么当前线程将被禁用以进行线程调度，并且处于休眠状态，和lock()方法不同的是在锁的获取期间**可以中断当前线程（响应中断）**。
Condition newCondition()	获取等待通知组件，该组件和当前的锁绑定，当前线程只有获得了锁，才能调用该组件的wait()方法，而调用后，当前线程将释放锁。
boolean tryLock()	1. 只有在调用时才可以获得锁。2. 如果可用，则获取锁，并**立即返回**值为**true**；如果锁不可用，则此方法将**立即返回**值为**false** 。3. 我们可以根据是否能获得锁，决定后续地一些行为。4. 该方法会立即返回，即使拿不到锁不会一直等待。
boolean tryLock(long time, TimeUnit unit)	超时获取锁，当前线程在以下三种情况下会返回： 1. 当前线程在超时时间之内获得了锁返回true；2.当前线程在超时时间内被中断返回false；3. 超时时间结束，返回false.
void unlock()	释放锁。

这里我们演示以下第二个方法：void lockInterruptibly()期间可中断

```
1  package lock.lock;
2
3  import java.util.concurrent.ThreadPoolExecutor;
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6
7  /**
8   * 描述：     第一个线程先获取到锁，然后让其睡眠，睡眠期间让第二个线程尝试获取锁，第二个线程会在获取锁期间阻塞，然后让第二个线程中断，观察获取锁期间是否会响应中断
9   */
10 public class LockInterruptibly implements Runnable {
11
12     private Lock lock = new ReentrantLock();
13 public static void main(String[] args) {
14     LockInterruptibly lockInterruptibly = new LockInterruptibly();
15     Thread thread0 = new Thread(lockInterruptibly);
16     Thread thread1 = new Thread(lockInterruptibly);
17     thread0.start();
```

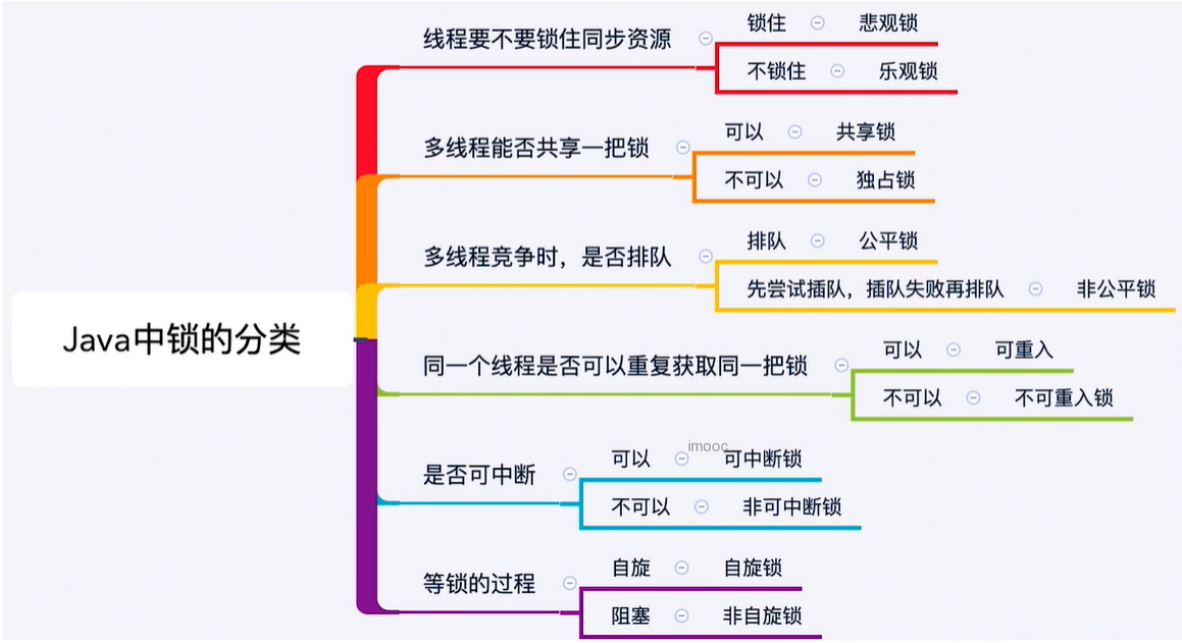
```
    thread1.start();

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    thread1.interrupt();
}
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "尝试获取锁");
        try {
            lock.lockInterruptibly();
            try {
                System.out.println(Thread.currentThread().getName() + "获取到了锁");
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "睡眠期间被中断了");
            } finally {
                lock.unlock();
                System.out.println(Thread.currentThread().getName() + "释放了锁");
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + "获得锁期间被中断了");
        }
    }
}
```

关于锁的可见性，参考JMM那篇的笔记，同样遵循happens-before原则这里不再赘述。

2. 锁的分类

这些分类,是从各种不同角度出发去看的,这些分类并不是互相矛盾的，也就是多个类型可以并存：有可能个锁 同时属于两种类型，比如ReentrantLock既是互斥锁，又是可重入锁。



2.1 线程要不要锁住同步资源

锁住——悲观锁

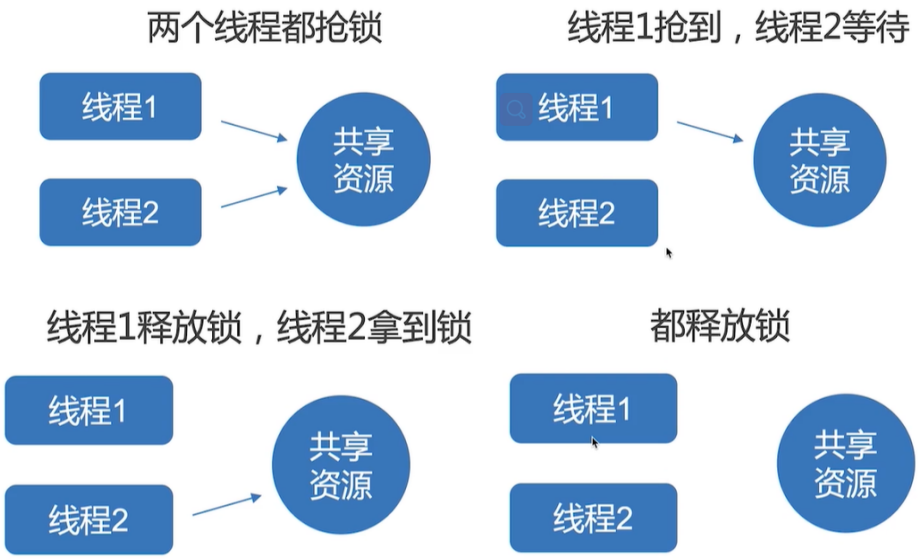
悲观锁即互斥同步锁

如果我不锁住这个资源,别人就会来争抢,就会造成数据结果错误,所以每次悲观锁为了确保结果的正确性,**会在每次获取并修改数据时,先把数据锁住**,让别人无法访问该数据,这样就可以确保数据内容万无一失

像Synchronized 、Lock相关类都是悲观锁。

数据库 select for update 也是悲观锁

悲观锁的抢锁过程



悲观锁应用场景:

适合并发**写入多**的情况,适用于临界区**持锁时间比较长**的情况,悲观锁可**以避免大量的无用自旋等消耗**,典型情况

1. 临界区有IO操作
2. 临界区代码复杂或者循环量大

3. 临界区竞争非常激烈

不锁住——乐观锁

乐观锁即非互斥同步锁

总是**假设最好的情况**，每次去读数据的时候都认为别人不会修改，所以**不会上锁**， 但是在更新的时候会判断一下在此期间有没有其他线程更新该数据， 可以使用版本号机制和CAS算法实现。 乐观锁**适用于多读的应用类型**，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。

如果数据和我开始拿到的不一样了,说明其他人在这段时间内改过数据,那我就不能继续刚才的更新数据过程了,我会选择**放弃、报错、重试**等策略

乐观锁的实现一般都是利用CAS算法来实现的（CAS的核心思想就是在一个原子操作内，把数据进行对比并且交换，在此期间是不会有其他线程打断的，CAS后面会详细讲解）

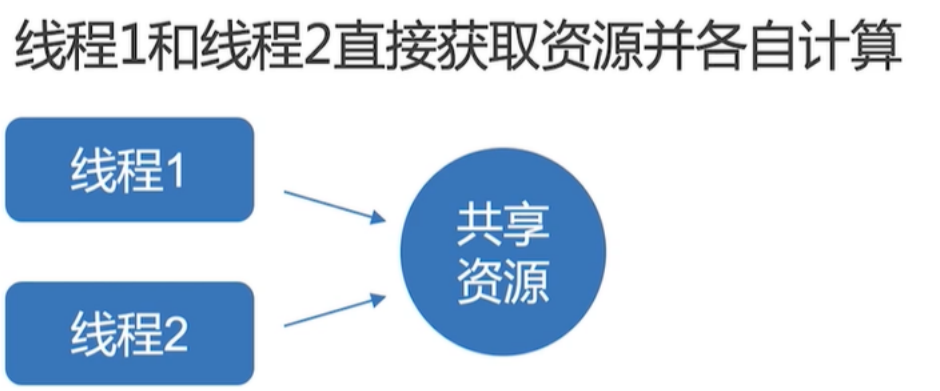
1.为什么会诞生非互斥同步锁一互斥同步锁的劣势

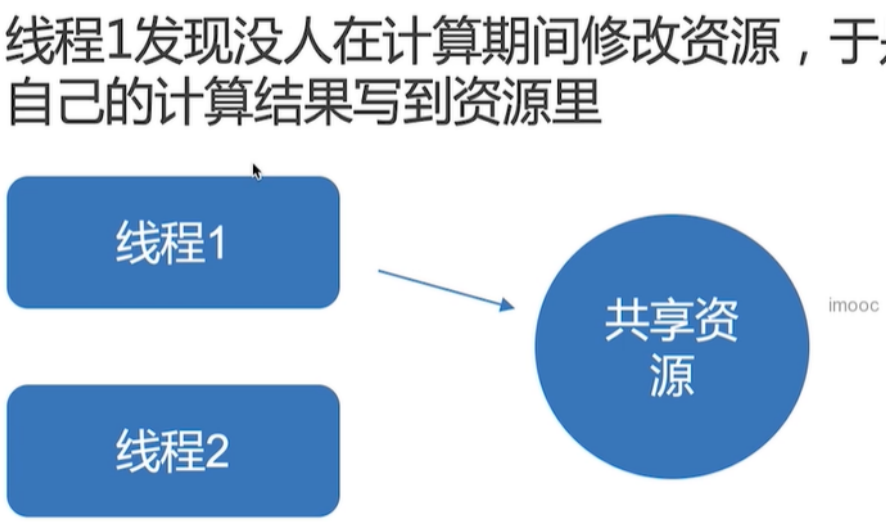
阻塞和唤醒带来的**性能劣势**，用户核心态和用户态的切换，检查是否有被阻塞线程，有线程上下文切换

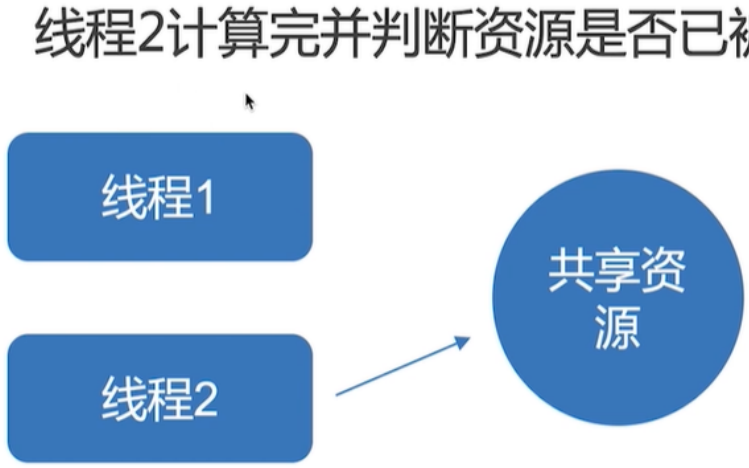
悲观锁可能会有第二个问题——陷入永久阻塞：如果持有锁的线程被永久阻塞比如遇到了无限循环、死锁等活跃性问题,那么等待该线程释放锁的那几个悲催的线程，将永远也得不到执行

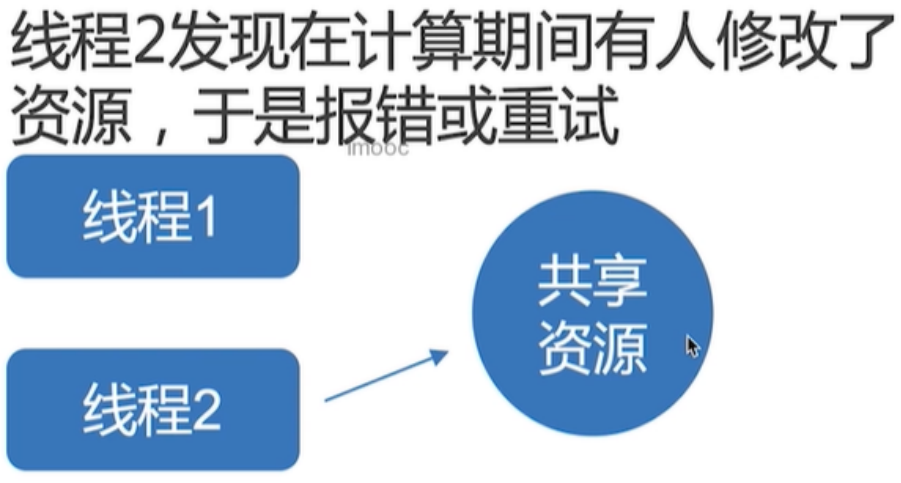
优先级反转：如果阻塞的线程优先级比较高，而持有锁的优先级比较低，就会导致优先级反转，我们本身设置的优先级高的线程，希望它多运行，结果它被阻塞，这就导致优先级高的反而运行少。

乐观锁的抢锁流程：









乐观锁的应用场景：

适合并发写入少,大部分是读取的场景,不加锁的能让读取性能大幅提高。

1. 乐观锁的典型例子就是**原子类、并发容器**等
2. **Git**：Git就是乐观锁的典型例子,当我们往远端仓库push的时候,git会检查远端仓库的版本是不是领先于我们现在的版本,如果远程仓库的版本号和本地的不一样,就表示有其他人修改了远端代码了,我们的这次提交就失败;如 果远端和本地版本号一致,我们就可以顺利提交版本到远端仓库

3. **数据库中的乐观锁应用：**

   添加一个字段 version，先查询这个更新语句的 `version: select * from table` 然后 `update set num=2 version= version+1 where version= 1 and id=5`

乐观锁和悲观锁的开销对比：

悲观锁的原始开销要高于乐观锁，但是特点是一劳永逸，临界区持锁时间就算越来越差，也不会对互斥锁的开销造成影响；

相反,虽然乐观锁一开始的开销比悲观锁小,但是如果自旋时间很长或者不停重试,那么消耗的资源也会越来越多

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

2.2 多线程竞争时，是否排队——公平和非公平锁

ReentrantLock又分为公平锁和非公平锁，但是他们都是通过维护一个节点队列来实现，只不过公平锁每次都取头结点执行，而非公平锁每次可能随机取节点执行。无论那种情况，每当一个线程获得锁之后都会使状态加一，当状态为0时会通知后面排队的节点（或者是在非公平锁中某一个正好合适的线程执行）。

什么是公平和非公平锁

公平指的是按照线程请求的顺序,来分配锁；非公平指的是,不完全按照请求的顺序,**在一定情况下,可以插队。**

注意:非公平也同样不提倡"插队"行为,这里的非公平,指的是**"在合适的时机"**插队,而不是盲目插队

什么是合适的时机：

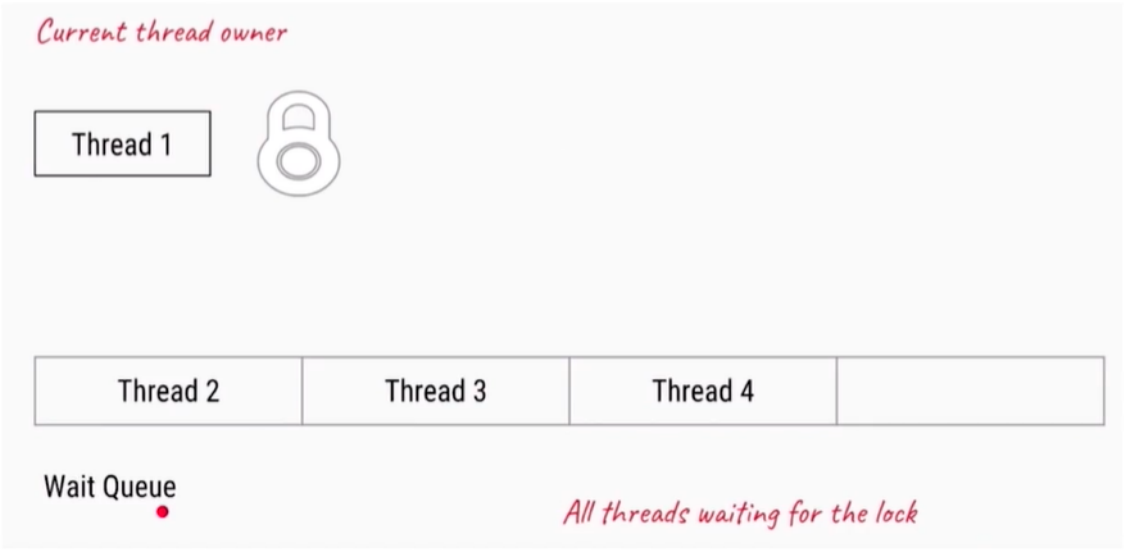
火车票被插队的例子：

为什么要有非公平锁

避免唤醒带来的空档期，把挂起的线程恢复过来的这段时间，是有开销的，如果是公平的，必须排队，那么这段时间任何线程都无法拿到锁，

公平的情况

如果在创建 Reentrantlock对象时,参数填写为true那么这就是个公平锁: 假设线程1234是按顺序调用lock()的



Thread1 先拿到锁，后续等待的线程会到 wait queue里,按照顺序依次执行

在线程1执行 unlock（）释放锁之后,由于此时线程2的等待时间最久,所以线程2先得到执行,然后是线程3和线程4.

不公平的情况

如果在线程1释放锁的时候,线程5恰好去执行lock()

由于 Reentrantlock发现此时并没有线程持有lck这把锁(线程2还没来得及获取到,因为获取需要时间)

线程5可以插队,直接拿到这把锁,这也是 Reentrantlock**默认**的公平策略,也就是"不公平"

代码演示：演示公平和非公平的效果

特例

针对 tryLock方法,它不遵守设定的公平的规则。例如,当有线程执行 tryLock的时候,一旦有线程释放了锁,那么这个正在 tryLock的线程就能获取到锁,即使在它之前已经有其他线程在等待队列里了

公平和不公平的优缺点

	优势	**劣势**
公平锁	各线程公平平等,每个线程 在等待一段时间后,总有执行的机会	更慢，吞吐量小
非公平锁	g更快，吞吐量更大	可能产生饥饿

源码分析

公平锁

```
/**...*/
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

非公平锁

```
/**...*/
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

通过上图中的源代码对比，我们可以明显的看出公平锁与非公平锁的lock()方法唯一的区别就在于公平锁在获取同步状态时多了一个限制条件：hasQueuedPredecessors()。

hasQueuedPredecessors() 也是 AQS 中的方法，它主要是用来 查询是否有任何线程在等待获取锁的时间比当前线程长，也就是说每个等待线程都是在一个队列中，此方法就是判断队列中在当前线程获取锁时，是否有等待锁时间比自己还长的队列，如果当前线程之前有排队的线程，返回 true，如果当前线程位于队列的开头或队列为空，返回 false。

2.3 同一个线程是否可以重复获取同一把锁

可以——可重入

ReentrantLocks意为可重入锁，也就是获得锁的同一个线程可以多次获得锁而不会阻塞。但是同一时间内锁只能被同一个线程持有。

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者`class`），不会因为之前已经获取过还没释放而阻塞。Java 中`ReentrantLock`和`synchronized`都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。下面用示例代码来进行分析：

```
1  public class Widget {
2      public synchronized void doSomething() {
3          System.out.println("方法1执行...");
4          doOthers();
5      }
6
7      public synchronized void doOthers() {
8          System.out.println("方法2执行...");
9      }
10 }
```

在上面的代码中，类中的两个方法都是被内置锁`synchronized`修饰的，`doSomething()`方法中调用`doOthers()`方法。因为内置锁是可重入的，所以同一个线程在调用`doOthers()`时可以直接获得当前对象的锁，进入`doOthers()`进行操作。

如果是一个不可重入锁，那么当前线程在调用`doOthers()`之前需要将执行`doSomething()`时获取当前对象的锁释放掉，实际上该对象锁已被当前线程所持有，且无法释放。所以此时会出现死锁。

之前我们说过`ReentrantLock`和`synchronized`都是重入锁，那么我们通过重入锁`ReentrantLock`以及非可重入锁`NonReentrantLock`的源码来对比分析一下为什么非可重入锁在重复调用同步资源时会出现死锁。

不可以——不可重入锁

可重入锁和不可重入锁原理分析

首先`ReentrantLock`和`NonReentrantLock`都继承父类 AQS，其父类 AQS 中维护了一个同步状态`status`来计数重入次数，`status`初始值为`0`。

当线程尝试获取锁时，可重入锁先尝试获取并更新`status`值，如果`status == 0`表示没有其他线程在执行同步代码，则把`status`置为`1`，当前线程开始执行。如果`status != 0`，则判断当前线程是否是获取到这个锁的线程，如果是的话执行`status+1`，且当前线程可以再次获取锁。而非可重入锁是直接去获取并尝试更新当前`status`的值，如果`status != 0`的话会导致其获取锁失败，当前线程阻塞。

释放锁时，可重入锁同样先获取当前`status`的值，在当前线程是持有锁的线程的前提下。如果`status-1 == 0`，则表示当前线程所有重复获取锁的操作都已经执行完毕，然后该线程才会真正释放锁。而非可重入锁则是在确定当前线程是持有锁的线程之后，直接将`status`置为`0`，将锁释放。

具体的源码分析会放到AQS讲解的章节中去，这里现有一个大概的认识。

2.4 多线程能否共享一把锁

什么是共享锁和排它锁

- 可以共享一把锁共享锁
- 不可以——独占锁（排他锁、互斥锁）

独占锁又叫做排他锁、互斥锁，是指锁在同一时刻只能被一个线程拥有，其他线程想要访问资源，就会被阻塞。JDK 中 synchronized和 JUC 中 Lock 的实现类就是互斥锁。

共享锁指的是锁**能够被多个线程所拥有**，如果某个线程对资源加上共享锁后，则其他线程**只能对资源再加共享锁**，不能加排它锁。**获得共享锁的线程只能读数据，不能修改数据。**

读写锁的作用

重入锁 ReentrantLock 是排他锁，排他锁在同一时刻仅有一个线程可以进行访问，但是在大多数场景下，大部分时间都是提供读服务，而写服务占有的时间较少。然而，读服务不存在数据竞争问题，如果一个线程在读时禁止其他线程读势必会导致性能降低。所以就提供了读写锁。

ReentrantReadWriteLock具体用法

ReentrantReadWriteLock 类的大体结构如下：

```
/** 内部类  读锁 */
private final ReentrantReadWriteLock.ReadLock readerLock;
/** 内部类  写锁 */
private final ReentrantReadWriteLock.WriteLock writerLock;

final Sync sync;

/** 使用默认（非公平）的排序属性创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock() {
    this(false);
}

/** 使用给定的公平策略创建一个新的 ReentrantReadWriteLock */
public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
```

```
}

/** 返回用于写入操作的锁 */
@Override
public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
/** 返回用于读取操作的锁 */
@Override
public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }

abstract static class Sync extends AbstractQueuedSynchronizer {
    /**
     * 省略其余源代码
     */
}
public static class WriteLock implements Lock, java.io.Serializable {
    /**
     * 省略其余源代码
     */
}

public static class ReadLock implements Lock, java.io.Serializable {
    /**
     * 省略其余源代码
     */
}
```

我们看到 ReentrantReadWriteLock 有两把锁：ReadLock 和 WriteLock，也就是一个读锁一个写锁，合在一起叫做读写锁。再进一步观察可以发现 **ReadLock 和 WriteLock 是靠内部类 Sync 实现的锁。** Sync 是继承于 AQS 子类的，AQS 是并发的根本，这种结构在CountDownLatch、ReentrantLock、Semaphore里面也都存在。

在 **ReentrantReadWriteLock** 里面，读锁和写锁的锁主体都是 **Sync**，但读锁和写锁的加锁方式不一样。读锁是共享锁，写锁是独享锁。读锁的共享锁可保证并发读非常高效，而读写、写读、写写的过程互斥，因为读锁和写锁是分离的。所以ReentrantReadWriteLock的并发性相比一般的互斥锁有了很大提升。

参考：https://juejin.im/post/5d832ff3e51d4561ea1a9512#heading-25

读锁和写锁的交互方式

1. 多个线程只申请读锁,都可以申请到
2. 如果有一个线程已经占用了读锁,则此时其他线程如果要申请写锁,则申请写锁的线程会一直等待释放读锁。
3. 如果有一个线程已经占用了写锁,则此时其他线程如果申请写锁或者读锁,则申请的线程会一直等待释放写锁
4. 一句话总结:要么是一个或多个线程同时有读锁,要么是一个线程有写锁,但是两者不会同时出现(**要么多读,要么一写**)

读写锁插队策略：

- 公平锁
  - 不允许插人
- 非公平锁
  - 写锁可以随时插队
  - 读锁仅在**等待队列头结点不是想获取写锁的线程**的时候可以插队，会先盘对等待队列对头是不是想获取写锁，如果是，就不会选择插队，避免写的线程发生饥饿。

读写锁插队代码演示

1. 演示读锁可以共享，读锁的下一个线程是写锁不可以插队

```
package lock.readwrite;

import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 描述：     TODO
 */
public class CinemaReadWriteQueue {

    private static ReentrantReadWriteLock reentrantReadWriteLock = new ReentrantReadWriteLock(false);
    private static ReentrantReadWriteLock.ReadLock readLock = reentrantReadWriteLock.readLock();
    private static ReentrantReadWriteLock.WriteLock writeLock = reentrantReadWriteLock.writeLock();

    private static void read() {
        readLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了读锁，正在读取");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放读锁");
            readLock.unlock();
        }
    }

    private static void write() {
        writeLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到了写锁，正在写入");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放写锁");
            writeLock.unlock();
        }
    }

    public static void main(String[] args) {
        new Thread(()->write(),"Thread1").start();
        new Thread(()->read(),"Thread2").start();
        new Thread(()->read(),"Thread3").start();
        new Thread(()->write(),"Thread4").start();
        new Thread(()->read(),"Thread5").start();

```

```
    }
}
```

```
Thread1得到了写锁，正在写入
Thread1释放写锁
Thread2得到了读锁，正在读取
Thread3得到了读锁，正在读取
Thread3释放读锁
Thread2释放读锁
Thread4得到了写锁，正在写入
Thread4释放写锁
Thread5得到了读锁，正在读取
Thread5释放读锁
```

可以看到Thread2和Thread3线程同时获取到读锁，可以并发执行读操作

而Thread5无法和Thread2和Thread3同时获取到读锁，由于下一个等待队列中的Thread4是想要获得写锁的，故无法插队。

2. 演示读线程可以插队的情况：等待队列中的第一个线程不是想获取写锁

```
package lock.readwrite;

import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 描述：     演示非公平和公平的ReentrantReadWriteLock的策略
 */
public class NonfairBargeDemo {

    private static ReentrantReadWriteLock reentrantReadWriteLock = new ReentrantReadWriteLock(
            false);

    private static ReentrantReadWriteLock.ReadLock readLock = reentrantReadWriteLock.readLock();
    private static ReentrantReadWriteLock.WriteLock writeLock = reentrantReadWriteLock.writeLock();

    private static void read() {
        System.out.println(Thread.currentThread().getName() + "开始尝试获取读锁");
        readLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到读锁，正在读取");
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
```

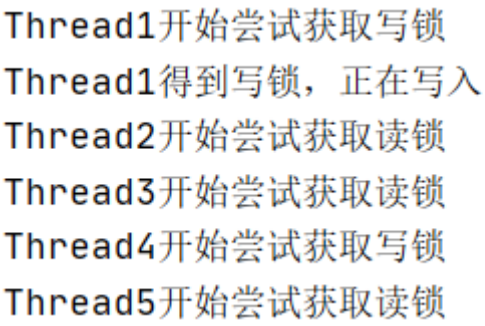
```
            System.out.println(Thread.currentThread().getName() + "释放读锁");
            readLock.unlock();
        }
    }

    private static void write() {
        System.out.println(Thread.currentThread().getName() + "开始尝试获取写锁");
        writeLock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "得到写锁，正在写入");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + "释放写锁");
            writeLock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(()->write(),"Thread1").start();
        Thread.sleep(10);
        new Thread(()->read(),"Thread2").start();
        new Thread(()->read(),"Thread3").start();
        new Thread(()->write(),"Thread4").start();
        new Thread(()->read(),"Thread5").start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                Thread thread[] = new Thread[1000];
                for (int i = 0; i < 1000; i++) {
                    thread[i] = new Thread(() -> read(), "子线程创建的Thread" + i);
                }
                for (int i = 0; i < 1000; i++) {
                    thread[i].start();
                }
            }
        }).start();
    }
}
```

这里用子线程new 出1000个子线程的方法，去演示不断插队，我们主要看，在Thread1释放写锁后，是否后尝试获取锁的线程优先于Thread2或Thread3线程拿到读锁

结果发现：



我们一开始明明是Thread2和Thread3先要尝试获取锁，但是Thread1释放锁后，后面尝试获取锁的线程反而先拿到了锁，发生了插队现象

读写锁插队规则的源码分析

```
/**
 * Fair version of Sync
 */
static final class FairSync extends Sync {
    private static final long serialVersionUID = -2274990926593161451L;

    // writer想获取写锁的线程
    final boolean writerShouldBlock() {
        // 调用AQS中的方法
        return hasQueuedPredecessors();
    }
    final boolean readerShouldBlock() {
        return hasQueuedPredecessors();
    }
}

static final class NonfairSync extends Sync {
    final boolean writerShouldBlock() {
        return false;
    }

    final boolean readerShouldBlock() {
        return apparentlyFirstQueuedIsExclusive();
    }
}
```

FairSync 和 NonfairSync 的区别在于 writerShouldBlock 和 readerShouldBlock 两个方法的具体实现，这两个方法的含义是是否应该被阻塞，如果返回true，代表应该被阻塞。返回false表示不该被阻塞。

在公平锁中writerShouldBlock()和readerShouldBlock()都去调用**AQS**中的`hasQueuedPredecessors()`；这个方法的作用在其注释中已经写明：

Queries whether any threads have been waiting to acquire longer than the current thread.

查询是否有线程等待获取的时间长于当前线程

如果有，返回true，没有放回false

```
public final boolean hasQueuedPredecessors() {
    // 每一个队列中的线程封装为一个node
    Node h, s;
    // 把头节点赋给head，如果有头结点
    if ((h = head) != null) {
        if ((s = h.next) == null || s.waitStatus > 0) {
            s = null; // traverse in case of concurrent cancellation
            for (Node p = tail; p != h && p != null; p = p.prev) {
                // 这里有个waitStatus，查看下面说明，waitStatus<=0代表的是，有等待被唤醒的线程，或者有初始化好的线程
                if (p.waitStatus <= 0)
                    s = p;
            }
        }
        if (s != null && s.thread != Thread.currentThread())
            // 如果没有，则返回true
            return true;
    }
    // 没有头节点，说明整个队列为空，直接返回false
    return false;
}
```

在尝试获取锁的方法中，有用到

```
@ReservedStackAccess
protected final boolean tryAcquire(int acquires) {
    /*
     * Walkthrough:
     * 1. If read count nonzero or write count nonzero
     *    and owner is a different thread, fail.
     * 2. If count would saturate, fail. (This can only
     *    happen if count is already nonzero.)
     * 3. Otherwise, this thread is eligible for lock if
     *    it is either a reentrant acquire or
     *    queue policy allows it. If so, update state
     *    and set owner.
     */
    Thread current = Thread.currentThread();
    int c = getState();
    int w = exclusiveCount(c);
    if (c != 0) {
        // (Note: if c != 0 and w == 0 then shared count != 0)
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
```

```
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // Reentrant acquire
        setState(c + acquires);
        return true;
    }
    // 这里调用了上面的wirterShouldBlock()方法
    if (writerShouldBlock() ||
        !compareAndSetState(c, c + acquires))
        return false;
    setExclusiveOwnerThread(current);
    return true;
}
```

我们来看一下非公平锁的实现

```
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = -8159625535654395037L;
    final boolean writerShouldBlock() {
        return false; // writers can always barge
    }
    final boolean readerShouldBlock() {
        /* As a heuristic to avoid indefinite writer starvation,
         * block if the thread that momentarily appears to be head
         * of queue, if one exists, is a waiting writer.  This is
         * only a probabilistic effect since a new reader will not
         * block if there is a waiting writer behind other enabled
         * readers that have not yet drained from the queue.
         */
        return apparentlyFirstQueuedIsExclusive();
    }
}
```

我们可以看到，在非公平锁的`writerShouldBlock()`方法直接返回false，代表永远不被阻塞，也就是非公平锁的实现中，想持有write锁的线程是**永远**可以插队的。

对于readerShouldBlock()来说，调用了`apparentlyFirstQueuedIsExclusive()`，这个方法会盘对队列中第一个等待线程是否想要获取排他锁（这里就是写锁），如果是，返回true，不让读线程插队，原因在注释中也已经写明 to avoid indefinite writer starvation , 避免写发生饥饿。

进入apparentlyFirstQueuedIsExclusive()

```
final boolean apparentlyFirstQueuedIsExclusive() {
    Node h, s;
    return (h = head) != null &&
        (s = h.next)  != null &&
        !s.isShared()         &&
        s.thread != null;
}
```

我们可以看到有这么一行代码：!s.isShared()，表明不能是共享锁，也就是只能是写锁才可能最终返回true

读写锁的升降级

读写锁有一个特性就是锁降级。锁降级就意味着写锁是可以降级为读锁的，但是需要遵循先获取写锁、再获取读锁再释放写锁的次序。注意如果当前线程先获取写锁，然后释放写锁，再获取读锁这个过程不能称之为锁降级，锁降级一定要遵循那个次序。

在获取读锁的方法 #tryAcquireShared(int unused) 中，有一段代码就是来判读锁降级的：

```
int c = getState();
//exclusiveCount(c)计算写锁
//如果存在写锁，且锁的持有者不是当前线程，直接返回-1
//存在锁降级问题，后续阐述
if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current)
    return -1;
//读锁
int r = sharedCount(c);
复制代码
```

代码演示：演示不支持读锁升级为写锁

```
public class ReadWriteTest {

    public static void main(String[] args) {
        ReadWriteLock rtLock = new ReentrantReadWriteLock();
        rtLock.readLock().lock();
        System.out.println("get readLock.");
        rtLock.writeLock().lock();
        System.out.println("blocking");

    }
}

// 程序会在获取写锁的地方被阻塞
```

代码演示：演示写锁可以降级为读锁

```
public class ReadWriteTest {

    public static void main(String[] args) {
        ReadWriteLock rtLock = new ReentrantReadWriteLock();
        rtLock.writeLock().lock();
        System.out.println("writeLock");
        rtLock.readLock().lock();
        System.out.println("get read lock");
    }
}
// 可以顺利执行完毕
```

锁降级中读锁的获取释放为必要？肯定是必要的。试想，假如当前线程 A 不获取读锁而是直接释放了写锁，这个时候另外一个线程 B 获取了写锁，那么这个线程 B 对数据的修改是不会对当前线程 A 可见的。如果获取了读锁，则线程B在获取写锁过程中判断如果有读锁还没有释放则会被阻塞，只有当前线程 A 释放读锁后，线程 B 才会获取写锁成功。

此处参考文章：https://juejin.im/post/6844903952274685959#heading-40

为什么不支持升级

如果有多个线程都想升级，而这些线程又都持有读锁，但是升级为写锁，必须其他线程释放读锁，这样就会导致死锁

这里是以ReentrantReadWriteLock为例的，有些锁可能只允许一个线程去升级锁，这样就可以避免死锁，不同锁实现可能不同。

总结

1. Reentrantreadwritelocke实现了 Readwritelock接口,最主要 的有两个方法: read Lock)和whitelock用来获取读锁和写锁
2. 锁申请和释放策略
    1. 多个线程只申请读锁,都可以申请到
    2. 如果有一个线程已经占用了读锁,则此时其他线程如果要申请写锁,则申请写锁的线程会一直等待释放读锁。
    3. 如果有一个线程已经占用了写锁,则此时其他线程如果申请写锁或者读锁,则申请的线程会一直等待释放写锁。
    4. 要么是个或多个线程同时有读锁,要么是一个线程有写锁,但是两者不会同时出现。
3. 插队策略:为了防止饥饿,读锁不能插队（具体来说是不能插写想获取写锁线程的队）
4. 升降级策略:只能降级,不能升级
5. 适用场合:相比于 Reentrantlock适用于一般场合,ReentrantreadwriteLock适用于读多写少的情况,合理使用可以进 一步提高并发效率

2.5 是否可中断

在Java中, synchronized就不是可中断锁,而Lock是可中断锁,因为 tryLock(time)和 lockInterruptibly都可以响应中断。

如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了,想先处理其他事情，我们可以中断它,这种就是可中断锁

可以——可中断

不可以——非可中断锁

2.6 等锁的过程

自旋不释放CPU资源——自旋锁

如果不使用自旋锁，那么我们**需要阻塞或唤醒一个Java线程**，阻塞和唤醒需要操作系统切换CPU状态来完成,这种状态转换需要**耗费处理器时间**

而如果同步代码块中的内容过于简单,状态转换消耗的时间有可能比用户代码执行的时间还要长

在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失

如果物理机器**有多个处理器**,能够让两个或以上的线程同时**并行**执行,我们就可以让后面那个请求锁的线程不放弃CPU的执行时间,看看持有锁的线程是否很快就会释放锁

而为了让当前线程“不断检测",我们需让当前线程进行**自旋**，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以**不必阻塞而是直接获取同步资源**,从而避免切换线程的开销。这就是自旋锁。

自旋锁缺点:

- 如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源，在自旋的过程中,一直消耗cpu,所以虽然自旋锁的起始开销低于悲观锁（非自旋锁），但是随着自旋时间的增长,开销也是线性增长的。
- 自旋锁要占用CPU，如果是计算密集型任务，这一优化通常得不偿失，减少锁的使用是更好的选择。
- 如果锁竞争的时间比较长，那么自旋通常不能获得锁，白白浪费了自旋占用的CPU时间。这通常发生在*锁持有时间长，且竞争激烈*的场景中，此时应主动禁用自旋锁。

> 使用-XX:-UseSpinning参数关闭自旋锁优化；-XX:PreBlockSpin参数修改默认的自旋次数。

在iava1.5版本及以上的并发框架`java.util.concurrent的atmoic`包下的类基本都是自旋锁的实现`Atomiclnteger`的实现:自旋锁的实现原理是**CAS**,`AtomicInteger`中调用`unsafe`进行自增操作的源码中的`do - while循环`就是一个自旋操作，**如果修改过程中遇到其他线程竞争导致没修改成功**,就在**while里死循环**,直至修改成功

自己实现一个简单的自旋锁：

```
package lock.spinlock;

import java.util.concurrent.atomic.AtomicReference;

/**
 * 描述：       自旋锁
 */
public class SpinLock {

    private AtomicReference<Thread> sign = new AtomicReference<>();

    public void lock() {
        Thread current = Thread.currentThread();
        while (!sign.compareAndSet(null, current)) {
            System.out.println("自旋获取失败，再次尝试");
        }
    }

    public void unlock() {
        Thread current = Thread.currentThread();
        sign.compareAndSet(current, null);
    }

    public static void main(String[] args) {
        SpinLock spinLock = new SpinLock();
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "开始尝试获取自旋锁");
                spinLock.lock();
                System.out.println(Thread.currentThread().getName() + "获取到了自旋锁");
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    spinLock.unlock();
```

```
                    System.out.println(Thread.currentThread().getName() +
"释放了自旋锁");
                }
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

核心代码：

```
while (!sign.compareAndSet(null, current)) {
    System.out.println("自旋获取失败，再次尝试");
}
```

不断去尝试直到获取成功，底层是CAS原理，后面CAS章节具体展开。

自旋锁使用场景

1. 自旋锁一般用于多核的服务器,在并发度不是特别高的情况下,比阻塞锁的效率高
2. 另外,自旋锁适用于临界区比较短小的情况,否则如果临界区很大(线程一旦拿到锁,很久以后才会释放),那也是不合适的

阻塞释放CPU资源——非自旋锁

阻塞锁和自旋锁相反,阻塞锁如果遇到没拿到锁的情况,会直接把线程阻塞,直到被唤醒。

3. 锁优化

关于锁的优化，主要分为两个大方面，都很重要，一方面是JVM对锁的性能的优化，另一方面是我们作为程序员，在代码层面，例如最佳实践、使用场景等，也可以对锁性能的提高做出贡献。

Java虚拟机对锁的优化

1. 自适应自旋锁

自适应意味着自旋的时间不再固定了，而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：

- 如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。
- 相反的，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能减少自旋时间甚至省略自旋过程，以避免浪费处理器资源。

自适应自旋解决的是"锁竞争时间不确定"的问题。JVM很难感知到确切的锁竞争时间，而交给用户分析就违反了JVM的设计初衷。*自适应自旋假定不同线程持有同一个锁对象的时间基本相当，竞争程度趋于稳定，因此，可以根据上一次自旋的时间与结果调整下一次自旋的时间。*

缺点

然而，自适应自旋也没能彻底解决该问题，*如果默认的自旋次数设置不合理（过高或过低），那么自适应的过程将很难收敛到合适的值。*

2. 偏向锁、轻量级锁、重量级锁

这几种锁并不是java类库实现的，而是虚拟机底层对 `synchronized` 代码块的不同加锁方式。为了换取性能，JVM在内置锁上做了非常多的优化，膨胀式的锁分配策略就是其一。理解偏向锁、轻量级锁、重量级锁的要解决的基本问题，几种锁的分配和膨胀过程，有助于编写并优化基于锁的并发程序。

偏向锁、轻量级锁、重量级锁会在synchronized章节详细讲解，由于篇幅较长这里先不赘述。

3. 锁消除

如果能确认某个加锁的对象不会逃逸出局部作用域，就可以进行锁删除。这意味着这个对象同时只可能被一个线程访问，因此也就没有必要防止其它线程对它进行访问了。这样的话这个锁就是可以删除的。这个便叫做锁消除。

4. 锁粗化

是一种通过分析作用域来提高内部锁的性能的手段。HotSpot虚拟机会去检查多个锁区域是否能合并成一个更大的锁区域。这种聚合被称作锁粗化，它能够减少加锁和解锁的消耗。

当HotSpot JVM发现需要加锁时，它会尝试往前查找同一个对象的解锁操作。如果能匹配上，它会考虑是否要将两个锁区域作合并，并删除一组解锁/加锁操作。

> 锁粗化是默认开启的，不过也可以通过启动参数-XX:-EliminateLocks来关掉它。

我们在写代码时如何优化锁和提高并发性能

缩小同步代码块

只锁必要的操作数据的部分，把其他不相关的、开销大的、可能被阻塞的操作，例如IO操作，通通移出互斥的范围。

当然，也不能过分小，原子操作必须包含到一个同步块中。

尽量不要锁住方法

方法这个级别太高了、范围太宽了。如果我们锁住方法，很有可能其实我们并不需要这么大锁的范围，我们可以把这个锁住方法的锁优化成代码块，代码会锁住的部分往往比方法要小。

减小锁的粒度（把锁拆小）

减少请求锁的次数

如果只看这个标题，同学们可能会很困惑，我明明是有请求锁的需求，你凭什么让我减少次数呢？如果我减少次数的话，是不是就无法保证线程安全了呢？实际情况不总是这样的，有的时候我们可以通过减少请求锁的次数来优化性能，让我们来举一个具体的非常好的例子：

在很多日志框架里。最主要的思路是把系统的打印日志的能力进行一系列包装，比如说可以设置打印的级别设置、格式等等。

但是对于日志框架，有一个很重要的点，那就是当多个线程同时进行日志打印的时候，可能如果只是采用刚才的那种简单的架构设计，那么多个线程同时打印便会造成线程竞争，因为这是一个IO操作，多个线程需要操作同一个日志文件，需要加锁。

这里我们对架构进行改进，把多个线程的打印日志的这种需求收集到一起，然后统一由某一个线程去执行写入文件的操作，这样的话便不会有过多的锁的竞争的情况，可以大大提高打印日志的效率，我们相当于是使用了一个消息队列来作为我们的中间层，这也是我们现有的优质日志框架Lof4j2的一个思路。

避免人为制造“热点”

这里的“热点”，指的是一个大家都想同时访问的，并且需要互斥同步的资源。有的时候。我们会人为制造出这种资源，从而导致了性能的下降。

比如一个典型的例子就是在HashMap中，size()方法是获取到当前容器大小，size的第一种实现方式是每次调用size方法的时候，就去遍历一遍，但是这样的时间复杂度是O(n)。

所以就有一种优化方法是，每一次增加元素的时候，比如put方法，我就去更新计数器，这样一来，如果需要知道集合的大小，我只需要读取一下之前已经更新过的数值就可以了，把复杂度降到了O(1)，这是一个很不错的优化。

可是这有一个问题，那就是如果并发地去放置元素，就会导致这个计数器被多线程同时使用，也就带来了性能问题，因为多个线程需要去竞争锁。

一个相当不错的解决方案是，在JDK7的ConcurrentHashMap中，不同的segment之间不共用同一个计数器，而仅仅是在最终真正需要获取集合容量的时候，再把多个计数器的值相加，这样的思维，我们在之前的LongAdder中已经分析过了，是一种非常好的思路，可以大大提高并发的能力。

锁中尽量不要再包含锁

否则很容易死锁，演示一个死锁的例子，之前写过

选择合适的锁类型或合适的工具类

不同的锁适合不同的场景。放弃互斥锁，可以提高并发性能。

监控CPU的利用率