千变万化的锁

1. Lock接口
2. 锁的分类
3. 乐观锁和悲观锁
4. 可重入锁和非可重入锁，以ReentrantLock为例
5. 公平锁和非公平锁
6. 共享锁和排他锁：以ReentrantReadWriteLock读写锁为例
7. 自旋锁和阻塞锁
8. 可中断锁：顾名思义，就是可以响应中断的锁
9. 锁优化

Lock接口

1. 简介、地位、作用
2. 为什么synchronized不够用？为什么需要Lock?
3. 方法介绍
4. 可见性保证

简介、地位、作用

锁是一种工具，用于控制对共享资源的访问。

Lock和synchronized,这两个是最常见的锁，它们都可以达到线程安全的目的，但是在使用上和功能上又有较大的不同。

Lock并不是用来代替synchronizedi的，而是当使用synchronized不合适或不足以满足要求的时候，来提供高级功能的。

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

Lock主要方法介绍

- 在Lock中声明了四个方法来获取锁
- lock()、tryLock0、tryLock(long time,TimeUnit unit)和lockInterruptibly()
- 那么这四个方法有何区别呢?

Lock()

lock()就是最普通的获取锁。如果锁已被其他线程获取，则进行等待

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

lock()方法不能被中断，这会带来很大的隐患：一旦陷入死锁
lock()就会陷入永久等待

tryLock()

tryLock()用来尝试获取锁，如果当前锁没有被其他线程占用
则获取成功，则返回true,否则返回false,代表获取锁失败

相比于引lock,这样的方法显然功能更强大了，我们可以根据
是否能获取到锁来决定后续程序的行为

该方法会立即返回，即便在拿不到锁时不会一直在那等

tryLock(long time, TimeUnit unit)

添加了超时时间

lockInterruptibly()

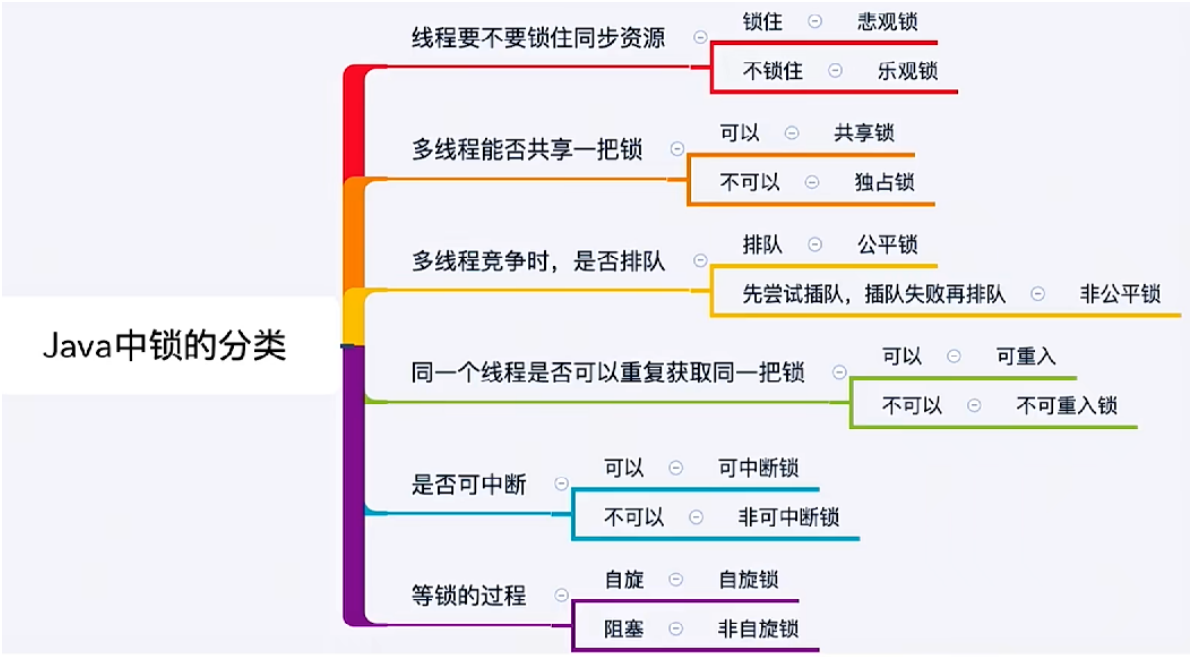
相当于tryLock(long time, TimeUnit unit)把超时时间设置为无限。在等待锁的过程中，线程可以被中断

可见性保证

- 可见性
- happens-before
- Lock的加解锁和synchronized有同样的内存语义，也就是说下一个线程加锁后可以看到所有前一个线程解锁前发生的所有操作

锁的分类

- 这些分类，是从各种不同角度出发去看的
- 这些分类并不是互斥的，也就是多个类型可以并存：有可能一个锁同时属于两种类型
- 比如ReentrantLock既是互斥锁，又是可重入锁



为什么会诞生互斥同步锁

互斥同步锁的劣势

- 阻塞和唤醒带来的性能劣势
- 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

悲观锁

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，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

Java中悲观锁的实现就是synchronized和Lock相关类

乐观锁

认为自己在处理操作的时候不会有其他线程来干扰，所以并不会锁住被操作对象
在更新的时候，去对比在我修改的期间数据有没有被其他人改变过如果没被改变过，就说明真的是只有我自己在操作，那我就正常去修改数据

如果数据和我一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

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

使用场景

悲观锁：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:

- 临界区有IO操作
- 临界区代码复杂或循环量大
- 临界区竞争非常激烈

乐观锁：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

ReetrantLock使用案例

- 普通方法1：预定电影院座位
- 普通方法2：打印字符串

ReentrantLock重入原理

当前线程想多次获取锁，可以多次获取，通过计数来计算获取次数。

公平锁和非公平锁

- 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。
- 什么是合适的时机呢?
- 实际情况并不是这样的，Java设计者这样设计的目的，是为了提高效率
- 避免唤醒带来的空档期

公平锁和非公平锁的优缺点

	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，吞吐量更小
不公平锁	更快，吞吐量更大	有可能产生线程饥饿，也就是某些线程在长时间内，始终得不到执行

共享锁和排他锁

- 排他锁，又称为独占锁、独享锁
- 共享锁，又称为读锁，获得共享锁之后，可以查看但无法修改和删除数据，其他线程此时也可以获取到共享锁，也可以查看但无法修改和删除数据
- 共享锁和排它锁的典型是读写锁ReentrantReadWriteLock,其中读锁是共享锁，写锁是独享锁

读写锁的作用

- 在没有读写锁之前，我们假设使用ReentrantLock,那么虽然我们保证了线程安全，但是也浪费了一定的资源：多个读操作同时进行，并没有线程安全问题
- 在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的提高了程序的执行效率

读写锁规则

- 多个线程只申请读锁，都可以申请到
- 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。
- 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。
- 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，要多一写）

换一种思路更容易理解：读写锁只是**一把锁**，可以通过两种方式锁定**读锁定和写锁定。**读写锁可以同时**被一个或多个线程读锁定**，也可以被**单一线程写锁定。**但是永远不能同时对这把锁**进行读锁定和写锁定。**

这里是把"获取写锁"理解为"把读写锁进行写锁定"，相当于是换了一种思路，不过原则是不变的，就是要么是**一个或多个线程同时有读锁（同时读锁定）**，要么是**一个线程有写锁（进行写锁定）**，但是两者不会同时出现

读写锁插队策略

公平锁：不允许插队
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

锁的升降级

- 为什么需要升降级
- 支持锁的降级，不支持升级
- 为什么不支持升级？死锁

共享锁和排它锁总结

1. ReentrantReadWriteLock:实现了ReadWriteLock接口，最主要的有两个方法：readLockO和writeLock(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。
3. 插队策略：为了防止饥饿，读锁不能插队
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。**

总结：要么多读，要多一写

自旋锁和阻塞锁

- 阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间
- 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“**稍等一下**”，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。
- 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，知道被唤醒

自旋锁的缺点

- 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，但是随着自旋时间的增长，开销也是线性增长的

自旋锁的适用场景

- 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
- 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），不太合适

可中断锁

- 在Java中，synchronized就不是可中断锁，而lock是可中断锁因为tryLock(time)和lockInterruptibly都能响应中断。
- 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

锁优化

JVM对锁的优化

- 自旋锁和自适应
- 锁消除
- 锁粗化

自己优化策略

1. 缩小同步代码快
2. 尽量不要锁住方法
3. 减少请求锁的次数
4. 避免人为制造”热点“
5. 锁中尽量不要再包含锁
6. 选择合适的锁类型或合适的工具类