> 主要讲解Java中常见的锁。

前言

并发编程系列应该快接近尾声，锁可能是这个系列的最后一篇，重要的基本知识应该都涵盖了。然后对于书籍《Java并发编程实战》，最后面的几章，我也只看了锁的部分，这篇文章主要是对该书中锁的内容进行一个简单的总结。

死锁

> 死锁是指一组互相竞争资源的线程因互相等待，导致"永久"阻塞的现象。

锁顺序死锁

我们先看一个死锁的示例，我们先定义个BankAccount对象，来存储基本信息，代码如下：

```
public class BankAccount {
    private int id;
    private double balance;
    private String password;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
}
```

接下来，我们使用细粒度锁来尝试完成转账操作：

```
public class BankTransferDemo {
    public void transfer(BankAccount sourceAccount, BankAccount targetAccount, double amount) {
        synchronized(sourceAccount) {
            synchronized(targetAccount) {
                if (sourceAccount.getBalance() > amount) {
                    System.out.println("Start transfer.");
                    sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                    targetAccount.setBalance(targetAccount.getBalance() + amount);
                }
            }
        }
    }
}
```

如果进行下述调用，就会产生死锁：

```
transfer(myAccount, yourAccount, 10);
transfer(yourAccount, myAccount, 10);
```

如果执行顺序不当，那么A可能获取myAccount的锁并等待yourAccount的锁，然而B此时持有yourAccount的锁，并正在等待myAccount的锁。

通过顺序来避免死锁

由于我们无法控制参数的顺序，如果要解决这个问题，必须定义锁的顺序，并在整个应用程序中按照这个顺序来获取锁。我们可以通过Object.hashCode返回的值，来定义锁的顺序：

```
public class BankTransferDemo {

    private static final Object tieLock = new Object();

    public void transfer(BankAccount sourceAccount, BankAccount targetAccount, double amount) {

        int sourceHash = System.identityHashCode(sourceAccount);
        int targetHash = System.identityHashCode(targetAccount);

        if (sourceHash < targetHash) {
            synchronized(sourceAccount) {
                synchronized(targetAccount) {
                    if (sourceAccount.getBalance() > amount) {
                        sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                        targetAccount.setBalance(targetAccount.getBalance() + amount);
                    }
                }
            }
        } else if (sourceHash > targetHash) {
            synchronized(targetAccount) {
                synchronized(sourceAccount) {
                    if (sourceAccount.getBalance() > amount) {
                        sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                        targetAccount.setBalance(targetAccount.getBalance() + amount);
                    }
                }
            }
        } else {
            synchronized (tieLock) {
                synchronized(targetAccount) {
                    synchronized(sourceAccount) {
                        if (sourceAccount.getBalance() > amount) {
                            sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                            targetAccount.setBalance(targetAccount.getBalance() + amount);
                        }
                    }
                }
            }
        }
    }
}
```

无论你入参怎么变化，通过hash值的大小，我们永远是先锁住hash值小的数据，再锁hash值大的数据，这样就保证的锁的顺序。

但是在极少数情况下，两个对象的Hash值相同，如果顺序错了，仍可能导致死锁，所以在获取两个锁之前，使用"加时赛（Tie-Breaking）"锁，保证每次只有一个线程以未知的顺序获取到该锁。但是如果程序经常出现Hash冲突的情况，这里会成为并发的瓶颈，因为final变量是内存可见，会让所有的线程都阻塞到该锁上，不过这种概率会很低。

在协作对象之间发生死锁

这里我就只简单说明一下，就是有两个对象A和B，A.action_A1()会调用B中的方法action_B1()，同时B.action_B2()会调用A中的方法action_A2()，由于这四个方法action_A1()、action_A2()、action_B1()、action_B2()都通过synchronized加锁，我们知道都通过synchronized在方法上加的是对象锁，所以可能存在A调用B的方法时，B也正在调用A的方法，导致互相等待出现死锁的情况。

> 具体的示例，大家可以参考《Java并发编程实战》书籍第174页的内容。

ReentrantLock

使用方法

在协调对象的访问时可以使用的机制只有synchronized和volatile，Java 5.0增加了一种新的机制：ReentrantLock。ReentrantLock并不是一种替代内置锁的方法，而是当内置锁机制不适用时，作为一种可选的高级功能。

下面看一个简单的示例：

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

除了上述不可替换synchronized的原因，就是需要手动通过lock.unlock()释放该锁，如果忘记释放，那将是个非常严重的问题。

通过tryLock避免顺序死锁

还是沿用上面的死锁示例，我们通过tryLock()进行简单改造：

```
public boolean transfer(BankAccount sourceAccount, BankAccount targetAccount, double amount, long timeout, TimeUnit unit) {
    long stopTime = System.nanoTime() + unit.toNanos(timeout);
    while (true) {
        if (sourceAccount.lock.tryLock()) {
            try {
                if (targetAccount.lock.tryLock()) {
                    try {
                        if (sourceAccount.getBalance() > amount) {
                            sourceAccount.setBalance(sourceAccount.getBalance() - amount);
                            targetAccount.setBalance(targetAccount.getBalance() + amount);
                        }
                    } finally {
                        targetAccount.lock.unlock();
                    }
                }
            } finally {
                sourceAccount.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime) {
            return false;
        }
        // Sleep一会...
    }
}
```

我们先尝试获取sourceAccount的锁，如果获取成功，再尝试获取targetAccount的锁，如果获取失败，我们就释放sourceAccount的锁，避免长期占用sourceAccount锁而导致的死锁问题。

带有时间限制的加锁

我们也可以对tryLock()指定超时时间，如果等待的时间超时，不会一直等待，直接执行后续的逻辑：

```
long stopTime = System.nanoTime() + unit.toNanos(timeout);
while (true) {
    long nanosToLock = unit.toNanos(timeout);
    if (sourceAccount.lock.tryLock(nanosToLock, TimeUnit.NANOSECONDS)) {
        try {
            // 省略...
        } finally {
            sourceAccount.lock.unlock();
        }
    }
    if (System.nanoTime() < stopTime) {
        return false;
    }
    // Sleep一会...
}
```

synchronized vs ReentrantLock

ReentrantLock在加锁和内存上提供的语义与内置锁相同，此外它还提供了一些其他的功能，包括定时的锁等待、可中断的锁等待、公平性，以及实现非块结构的加锁。ReentrantLock的性能上似乎优于内置锁，其中在Java 6.0中略有胜出，而在Java 5.0中则远远胜出，那是否我们都用ReentrantLock，直接废弃掉synchronized么？

与显示锁相比，内置锁仍然具有很大的优势。内置锁为许多开发人员所熟悉，并且简洁紧凑。ReentrantLock的危险性比同步机制要高，如果忘记在finally块中调用unlock，那么虽然代码表面上看起来能正常运行，但实际上已经埋下了一颗定时炸弹，并很有可能伤及其它代码。仅当内置锁不能满足需求时，才可以考虑使用ReentrantLock。

> 使用原则：ReentrantLock可以作为一种高级工具，当需要一些高级功能，比如可定时的、可轮训与可中断的锁获取操作，公平队列，以及非块结构的锁。否则，还是优先使用synchronized。

然后有一点需要重点强调一下，synchronized和ReentrantLock都是可重入锁，可重入的概念，请参考文章《【Java并发编程系列3】synchronized》。

读写锁

读写锁的使用和Go中的读写锁用法一致，先看读写锁接口定义：

```
public interface ReadWriteLock {
    /**
     * 返回读锁
     */
    Lock readLock();
    /**
     * 返回写锁
     */
    Lock writeLock();
}
```

ReadWriteLock管理一组锁，一个是只读的锁，一个是写锁。
Java并发库中ReetrantReadWriteLock实现了ReadWriteLock接口并添加了可重入的特性。

下面看一下使用姿势：

```
public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }
    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key,value);
        } finally {
            w.unlock();
        }
    }
    public V get(Object key) {
        r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }
}
```

这样可以多个线程去读取数据，但是只有一个线程可以去写数据，然后读和写不能同时进行。

其它

自旋锁

这个仅作为扩展知识，觉得有些意思，就写进来，那么什么是自旋锁呢？

> 自旋锁的定义：当一个线程尝试去获取某一把锁的时候，如果这个锁此时已经被别人获取(占用)，那么此线程就无法获取到这把锁，该线程将会等待，间隔一段时间后会再次尝试获取。这种采用循环加锁 -> 等待的机制被称为自旋锁(spinlock)。

自旋锁的原理

自旋锁的原理比较简单，如果持有锁的线程能在短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞状态，它们只需要等一等(自旋)，等到持有锁的线程释放锁之后即可获取，这样就避免了用户进程和内核切换的消耗。

因为自旋锁避免了操作系统进程调度和线程切换，所以自旋锁通常适用在时间比较短的情况下。由于这个原因，操作系统的内核经常使用自旋锁。但是，如果长时间上锁的话，自旋锁会非常耗费性能，它阻止了其他线程的运行和调度。线程持有锁的时间越长，则持有该锁的线程将被 OS(Operating System) 调度程序中断的风险越大。如果发生中断情况，那么其他线程将保持旋转状态(反复尝试获取锁)，而持有该锁的线程并不打算释放锁，这样导致的是结果是无限期推迟，直到持有锁的线程可以完成并释放它为止。

解决上面这种情况一个很好的方式是给自旋锁设定一个自旋时间，等时间一到立即释放自旋锁。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cpu 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁。

自旋锁的实现

```
public class SpinLockTest {
    private AtomicBoolean available = new AtomicBoolean(false);
    public void lock(){
        // 循环检测尝试获取锁
        while (!tryLock()){
            // doSomething...
        }
    }
    public boolean tryLock(){
        // 尝试获取锁，成功返回true，失败返回false
        return available.compareAndSet(false,true);
    }
    public void unLock(){
        if(!available.compareAndSet(true,false)){
            throw new RuntimeException("释放锁失败");
        }
    }
}
```

这种简单的自旋锁有一个问题：无法保证多线程竞争的公平性。对于上面的 SpinlockTest，当多个线程想要获取锁时，谁最先将available设为false谁就能最先获得锁，这可能会造成某些线程一直都未获取到锁造成线程饥饿。就像我们下课后蜂拥的跑向食堂，下班后蜂拥地挤向地铁，通常我们会采取排队的方式解决这样的问题，类似地，我们把这种锁叫排队自旋锁(QueuedSpinlock)。计算机科学家们使用了各种方式来实现排队自旋锁，如TicketLock，MCSLock，CLHLock。

锁的特性

Java 中的锁有很多，可以按照不同的功能、种类进行分类，下面是我对 Java 中一些常用锁的分类，包括一些基本的概述：

- 从线程是否需要对资源加锁可以分为"悲观锁"和"乐观锁"
- 从资源已被锁定，线程是否阻塞可以分为"自旋锁"
- 从多个线程并发访问资源，也就是Synchronized可以分为无锁、偏向锁、轻量级锁和重量级锁
- 从锁的公平性进行区分，可以分为"公平锁"和"非公平锁"
- 从根据锁是否重复获取可以分为"可重入锁"和"不可重入锁"
- 从那个多个线程能否获取同一把锁分为"共享锁"和"排他锁"

具体可以参考文章《不懂什么是锁？看看这篇你就明白了》：https://mp.weixin.qq.com/s?__biz=MzkwMDE1MzkwNQ==&mid=2247496038&idx=1&sn=10b96d79a1ff5a24c49523cdd2be43a4&chksm=c04ae638f73d6f2e1ead614f2452ebaeab26cf77b095d6f634654699a1084365e7f5cf6ca4f9&token=1816689916&lang=zh_CN#rd

总结

这篇文章主要讲解了死锁，死锁的解决方式，ReentrantLock，ReentrantLock和内置锁synchronized的比较，最后也讲解了自旋锁，前面内容是核心部分，自旋锁仅仅作为扩展知识。

锁的内容目前总结完了，所以Java并发编程系列我就先学到这个地方，后续如果学习到了其它Java并发知识，会持续维护这个系列。之前给自己定了Flag，今年需要把Java的基础知识都学完，所以我下个系列将会是Spring，希望和我一样的Java小白，能一起共同进步。

学习交流

可以扫下面二维码，关注「楼仔」公众号。

长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏👏👏

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。