JAVA 锁相关知识

乐观锁和悲观锁的区别

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去取数的时候都认为别人会修改，所以每次在取数的时候都会上锁，这样别人想使用这个数据时就会 block 直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

一个典型的倚赖数据库的悲观锁调用：
select * from account where name=”jake” for update
这条 sql 语句锁定了 account 表中所有符合检索条件（ name=”jake” ）的记录。 本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。 Hibernate 的悲观锁，也是基于数据库的锁机制实现。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去取数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别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 write_condition 机制的其实都是提供的乐观锁。

何谓数据版本号机制？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个如 “version” 字段来实现。 读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据，抛出异常或者 Retry。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

JAVA 中的 Java 中的读写锁和互斥锁

Java 中的读写锁

假设你的程序中涉及到对一些共享资源的读和写操作，且写操作没有读操作那么频繁。在没有写操作的时候，两个线程同时读一个资源没有任何问题，所以应该允许多个线程能在同时读取共享资源。但是如果有一个线程想去写这些共享资源，就不应该再有其它线程对该资源进行读或写，也就是说：读-读能共存，读-写不能共存，写-写不能共存。这就需要一个读/写锁来解决这个问题。
java.util.concurrent 包中 ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer， 读取锁可以由多个 reader 线程同时保持。 写入锁是独占的，即在任何时候：1.只有一个线程在写入；
2.线程正在读取的时候，写入操作等待； 3.线程正在写入的时候，其他线程的

写入操作和读取操作都要等待；

二、读写锁的机制

1、"读-读"不互斥，比如当前有 10 个线程去读（没有线程去写），这个 10 个线程可以并发读，而不会堵塞。

2、 "读-写"互斥，当前有写的线程的时候，那么读取线程就会堵塞，反过来，有读的线程在使用的时候，写的线程也会堵塞，就看谁先拿到锁了。

3、 "写-写"互斥，写线程都是互斥的，如果有两个线程去写，A 线程先拿到锁就先写，B 线程就堵塞直到 A 线程释放锁。

三、读写锁的适用场景

1、读多写少的高并发环境下，可以说这个场景算是最适合使用 ReadWriteLock 了。

注意： 在同一线程中，持有读锁后，不能直接调用写锁的 lock 方法 ，否则会造成死锁。

例子:

```
package com.ei.db.connection;

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockExample
{

    public static void main(String[] args)
    {
        // 创建账户
        MyCount myCount = new MyCount("4238920615242830", 10000);
        // 创建用户，并指定账户
        User user = new User("Tommy", myCount);

        // 分别启动 3 个“读取账户金钱”的线程 和 3 个“设置账户金钱”的线程
        for (int i = 0; i < 3; i++)
        {
            user.getCash();
            user.setCash((i + 1) * 1000);
```

```
            }
        }
}

class User
{
        private String name; //用户名
        private MyCount myCount; //所要操作的账户
        private ReadWriteLock myLock; //执行操作所需的锁对象

        User(String name, MyCount myCount)
        {
                this.name = name;
                this.myCount = myCount;
                this.myLock = new ReentrantReadWriteLock();
        }

        public void getCash()
        {
                new Thread()
                {
                        public void run()
                        {
                                myLock.readLock().lock();
                                try
                                {
                                        System.out.println(Thread.currentThread().getName()    +    "    getCash
start");
                                        myCount.getCash();
                                        Thread.sleep(1);
                                        System.out.println(Thread.currentThread().getName() + " getCash end");
                                }
                                catch (InterruptedException e)
                                {
                                }
                                finally
                                {
                                        myLock.readLock().unlock();
                                }
                        }
                }.start();
        }

        public void setCash(final int cash)
```

```
        {
            new Thread()
            {
                public void run()
                {
                    myLock.writeLock().lock();
                    try
                    {
                        System.out.println(Thread.currentThread().getName() + " setCash start");
                        myCount.setCash(cash);
                        Thread.sleep(1);
                        System.out.println(Thread.currentThread().getName() + " setCash end");
                    }
                    catch (InterruptedException e)
                    {
                    }
                    finally
                    {
                        myLock.writeLock().unlock();
                    }
                }
            }.start();
        }
    }

    class MyCount
    {
        private String id; //账号
        private int cash; //账户余额

        MyCount(String id, int cash)
        {
            this.id = id;
            this.cash = cash;
        }

        public String getId()
        {
            return id;
        }

        public void setId(String id)
        {
            this.id = id;
```

```
    }

    public int getCash()
    {
        System.out.println(Thread.currentThread().getName() + " getCash cash=" + cash);
        return cash;
    }

    public void setCash(int cash)
    {
        System.out.println(Thread.currentThread().getName() + " setCash cash=" + cash);
        this.cash = cash;
    }
}
```

Java 中的互斥锁

两种互斥锁机制：

1、synchronized

2、ReentrantLock

在 java5 之前，实现同步主要是使用 synchronized。它是 Java 语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

Synchronized 的用法：

```
synchronized (lockObject)
{
  // 更新共享变量
}
```

采用 ReentrantLock 可以完全替代替换 synchronized 传统的锁机制。ReentrantLock 实现更具可伸缩性，使用 ReentrantLock 的总体开支通常要比 synchronized 少得多。

ReentrantLock 的用法：

```
Lock lock = new ReentrantLock();
lock.lock();
try
{
    // 更新共享变量
}
finally
{
   lock.unlock();
}
```

例子

```
class ReentrantLockExample
        {
            int a = 0;
            ReentrantLock lock = new ReentrantLock();

            public void writer()
            {
                lock.lock(); //获取锁
                try
                {
                    a++;
                }
                finally
                {
                    lock.unlock(); //释放锁
                }
            }

            public void reader()
            {
                lock.lock(); //获取锁
                try
                {
                    int i = a;

                }
                finally
                {
                    lock.unlock(); //释放锁
                }
```

```
        }
    }
```

StampedLock

它是 java8 在 java.util.concurrent.locks 新增的一个 API。

ReentrantReadWriteLock 在沒有任何读写锁时，才可以取得写入锁，这可用于实现了悲观读（Pessimistic Reading），即如果执行中进行读取时，经常可能有另一执行要写入的需求，为了保持同步，ReentrantReadWriteLock 的读取锁定就可派上用场。

然而，如果读取执行情况很多，写入很少的情况下，使用 ReentrantReadWriteLock 可能会使写入线程遭遇饥饿（Starvation）问题，也就是写入线程迟迟无法竞争到锁定而一直处于等待状态。

StampedLock 控制锁有三种模式（写，读，乐观读），一个 StampedLock 状态是由版本和模式两个部分组成，锁获取方法返回一个数字作为票据 stamp，它用相应的锁状态表示并控制访问，数字 0 表示没有写锁被授权访问。在读锁上分为悲观锁和乐观锁。

所谓的乐观读模式，也就是若读的操作很多，写的操作很少的情况下，你可以乐观地认为，写入与读取同时发生几率很少，因此乐观地使用完全的读锁定，程序可以查看读取资料之后，是否遭到写入执行的变更，再采取后续的措施（重新读取变更信息，或者抛出异常），这一个小小改进，可大幅度提高程序的吞吐量！

下面是 java doc 提供的 StampedLock 一个例子

```
public class BankAccountWithStampedLock

{

  private final StampedLock lock = new StampedLock();

  private double balance;

  //排他写模式

  public void deposit(double amount)

  {
```

```
    long stamp = lock.writeLock();
    try
    {
      balance = balance + amount;
    } finally
    {
      lock.unlockWrite(stamp);
    }
  }
 //读模式
  public double getBalance()
  {
     long stamp = lock.readLock();
   try
   {
      return balance;
   } finally
   {
     lock.unlockRead(stamp);
   }
  }
```

乐观读模式

```
long stamp = lock.tryOptimisticRead(); //non blocking
read();
if(!lock.validate(stamp))
```

```
{ // if a write occurred, try again with a read lock

  long stamp = lock.readLock();

  try
  {
    read();
  }
  finally
  {
    lock.unlock(stamp);
  }}
```

SDK 的一个例子

```
class Point {
   private double x, y;
   private final StampedLock sl = new StampedLock();

   void move(double deltaX, double deltaY) { // an exclusively locked
method
     long stamp = sl.writeLock();
     try {
       x += deltaX;
       y += deltaY;
     } finally {
       sl.unlockWrite(stamp);
     }
   }

   double distanceFromOrigin() { // A read-only method
     long stamp = sl.tryOptimisticRead();
     double currentX = x, currentY = y;
     if (!sl.validate(stamp)) {
        stamp = sl.readLock();
        try {
          currentX = x;
          currentY = y;
        } finally {
           sl.unlockRead(stamp);
        }
     }
     return Math.sqrt(currentX * currentX + currentY * currentY);
```

```
  }

  void moveIfAtOrigin(double newX, double newY) { // upgrade
    // Could instead start with optimistic, not read mode
    long stamp = sl.readLock();
    try {
      while (x == 0.0 && y == 0.0) {
        long ws = sl.tryConvertToWriteLock(stamp);
        if (ws != 0L) {
          stamp = ws;
          x = newX;
          y = newY;
          break;
        }
        else {
          sl.unlockRead(stamp);
          stamp = sl.writeLock();
        }
      }
    } finally {
      sl.unlock(stamp);
    }
  }
}
```

StampedLock 要比 ReentrantReadWriteLock 更加廉价，也就是消耗比较小。

双重检查锁定模式与延迟初始化

下面的延迟初始化的方法不是线程安全的

```
class Foo {
   private Helper helper = null;
   public Helper getHelper()
      {
       if (helper == null)
           {
           helper = new Helper();
       }
       return helper;
```

```
    }

    // other functions and members...
}
```

可以通过加互斥锁来达到线程安全，如下面代码，但是性能达到降低

```
// Correct but possibly expensive multithreaded versionclass
Foo {
    private Helper helper = null;
    public synchronized Helper getHelper()
   {
        if (helper == null)
        {
            helper = new Helper();
        }
        return helper;
    }

    // other functions and members...
}
```

通过双重检查锁定来解决这个问题，需加 volatile 关键字

```
// Works with acquire/release semantics for volatile
// Broken under Java 1.4 and earlier semantics for volatileclass
Foo {
    private volatile Helper helper = null;
    public Helper getHelper()
   {
        Helper result = helper;//这么做为了提高性能
        if (result == null)
        {
            synchronized(this)
            {
                result = helper;
                if (result == null)
                {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }
    // other functions and members...
}
```