Lock 那点事儿

项目经理今天又接了一个客户需求，又要折磨我们这些程序员屌丝了。这个需求说起来很简单，做起来非常容易出错。我先简单描述一下：

这是一个在线文件编辑器。同一份文件，一个人在读的时候，其他人不能写；同理，一个人在写的时候，其他人也不能读。也就是说，要么读，要么写，这两件事情不能同时进行。

项目经理跟客户讲，“这个很容易实现的，我们是可以做的。”。什么都可以做，做不出来说是我们程序员能力不行，他一点责任都没有。领导发话了，不管怎么样，事情还是要做的。

看了一下需求，有两个问题，我得先问清楚，否则到时候做得不对，他又把负责推给我，我们项目经理经常搞这些让我背黑锅的事情。

“多人同时读可以吗？”

“当然可以啦！多少人来读都没关系，文件的内容不要变就行。”。

“多人同时写可以吗？”

“当然不行啦！你写别人也会写，文件不知道以哪份数据为准了。”。

他态度极其恶劣，算了，不跟他计较了，我的项目奖金还在他手里。赶紧完工，下班了还要回家抱小孩。

根据多年的项目实战经验，我写了一个超牛逼的 Data 类，来封装文件的数据。看起来是这样的：

|  |
| --- |
| public class Data {        private final char[] buffer;        public Data(int size) {          this.buffer = new char[size];          for (int i = 0; i < size; i++) {              buffer[i] = '\*';          }      }        public String read() {          StringBuilder result = new StringBuilder();          for (char c : buffer) {              result.append(c);          }          sleep(100);          return result.toString();      }        public void write(char c) {          for (int i = 0; i < buffer.length; i++) {              buffer[i] = c;              sleep(100);          }      }        private void sleep(long ms) {          try {              Thread.sleep(ms);          } catch (InterruptedException e) {              e.printStackTrace();          }      }  } |

稍微解释一下：

1. Data 类中封装了一个 char 数组类型的 buffer 成员变量。
2. 在构造器中传入一个 size，表示 buffer 的长度，并在其中创建并初始化这个 buffer，使其每个字符都为“\*”。
3. 提供两个方法，一个负责读取，另一个负责写入。在读取方法中只需遍历 buffer，将结果不断 append 到一个 StringBuilder 中，最终将其转为 String 并返回。
4. 在写入方法中传入一个字符，仍然是遍历 buffer，赋值 buffer 中的每个字符，这样可以使 buffer 中每个字符都是相同的。
5. 故意在读写方法中加入了一个 sleep() 方法，让程序运行慢一点，模拟比较耗时的操作。而且故意让写入比读取慢一点，因为将 sleep() 方法放入了 write() 方法的循环体中，而 read() 方法却没有。

当然了，以上这个示例跑通了，我想项目经理那个需求也不难实现。这也是我们平时做开发的一种习惯，先快速地写个 Demo 出来，让领导们看看，技术上走通了，我们再实现具体的需求。

好了，不就是要同时读写吗？这不就是一个典型的多线程使用场景吗？于是我快速地写了一个读取线程，让它拼命地去读取 Data 中的数据。

|  |
| --- |
| public class ReaderThread extends Thread {        private final Data data;        public ReaderThread(Data data) {          this.data = data;      }        @Override      public void run() {          while (true) {              String result = data.read();              System.out.println(Thread.currentThread().getName() + " => " + result);          }      }  } |

在 ReaderThread 中通过一个死循环去不断地读取 Data 中的数据，并将结果打印出来。

再来一个写入线程，让它使劲地向 Data 中写入数据。

|  |
| --- |
| public class WriterThread extends Thread {        private final Data data;      private final String str;      private int index = 0;        public WriterThread(Data data, String str) {          this.data = data;          this.str = str;      }        @Override      public void run() {          while (true) {              char c = next();              data.write(c);          }      }        private char next() {          char c = str.charAt(index);          index++;          if (index >= str.length()) {              index = 0;          }          return c;      }  } |

一次性可以传入一个字符串到 WriterThread 中，它将不断获取下一个字符（请见 next() 方法），并将该字符写入 Data 中。

如果让 ReaderThread 与 WriterThread 同时工作会怎样？不妨写了一个简单的 Client 类运行试试看。

|  |
| --- |
| public class Client {        public static void main(String[] args) {          Data data = new Data(10);            new ReaderThread(data).start();          new ReaderThread(data).start();          new ReaderThread(data).start();          new ReaderThread(data).start();          new ReaderThread(data).start();            new WriterThread(data, "ABCDEFGHI").start();          new WriterThread(data, "012345789").start();      }  } |

我开启了 5 个 ReaderThread 与 2 个 WriterThread，模拟读得多写得少的情况，并将不同的数据写入 Data 中。

运行一下！

…  
Thread-1 => AA0A0A00A0  
Thread-4 => AA0A0A00A0  
Thread-3 => AA0A0A00A0  
Thread-2 => AA0A0A00A0  
Thread-0 => AA0A0A00A0  
…

为何每次读取出来的数据不一致呢？应该是输出 10 个相同的字符才对啊！Data 的 buffer 中每个字符不是应该相同吗？

如果把这个结果给项目经理看，他肯定要搞死我的。

哦！想到了！在多线程开发中，资源的访问一定要做到“共享互斥”，也就是说要“上锁”，这招还是架构师前几天才教我的，我怎能不用？

于是我用了 Java 多线程中超牛逼的 synchronized 关键字，将它放到了 read() 与 write() 方法上，这样就可以保证 synchronized 方法在同一时刻只能被一个线程调用了，其他线程将会阻挡在外。

废话少说，赶紧加两个 synchronized 运行看看吧。

|  |
| --- |
| public class Data {        ...        public synchronized String read() {          ...      }        public synchronized void write(char c) {          ...      }        ...  } |

再运行一把！

…  
Thread-0 => 1111111111  
Thread-4 => CCCCCCCCCC  
Thread-3 => CCCCCCCCCC  
Thread-2 => CCCCCCCCCC  
Thread-1 => CCCCCCCCCC  
…

终于搞定啦！这下子项目经理应该满意了吧？

“不错！这效果很好啊，同时写同时读，而且每次读出来的数据都一样，技术上应该是走通了，这个需求应该可以实现了吧？” 项目经理问。

“没问题啊！小意思！” 我高兴的答。

“这是一个在线文件编辑器，你考虑过性能问题吗？” 架构师突然问了一句。

“性能很好啊！”

“你可以在 ReaderThread 中每调用 10 次 read() 方法，就打印 1 次所耗时间看看。”

“好啊！”

这还不简单，我快速地给 ReaderThread 的 run() 方法中加了几行代码，测试一下运行所消耗的时间。

|  |
| --- |
| public class ReaderThread extends Thread {        ...        @Override      public void run() {          while (true) {              long begin = System.currentTimeMillis();              for (int i = 0; i < 10; i++) {                  String result = data.read();                  System.out.println(Thread.currentThread().getName() + " => " + result);              }              long time = System.currentTimeMillis() - begin;              System.out.println(Thread.currentThread().getName() + " -- " + time + "ms");          }      }  } |

跑起来吧！

…  
Thread-2 => IIIIIIIIII  
Thread-2 — 24802ms  
Thread-3 => IIIIIIIIII  
Thread-3 — 24901ms  
Thread-4 => IIIIIIIIII  
Thread-4 — 25001ms  
Thread-0 => 3333333333  
…  
Thread-0 => 1111111111  
Thread-0 — 55305ms  
Thread-4 => CCCCCCCCCC  
Thread-3 => CCCCCCCCCC  
Thread-2 => CCCCCCCCCC  
Thread-1 => CCCCCCCCCC  
Thread-1 — 58705ms  
Thread-2 => CCCCCCCCCC  
…

我随意挑选了其中这 5 个 ReaderThread 所消耗的时间，平均值是：37742.8 毫秒，折合 37.8 秒。

我心里也没谱了，这性能到底是否需要优化呢？于是我带着测试结果，去向架构师请教。

他看到了这样的结果，微笑着摇了摇头。从他鄙视而又猥琐的表情上，我可以推测，这次他又要在我面前露一手了。

来吧，我给你写一个 ReadWriteLock，你自己去看吧。

随后，架构师用他熟练的手指，疯狂地在键盘上敲了一堆让我一知半解的东西。

|  |
| --- |
| public class ReadWriteLock {        private int readThreadCounter = 0;      // 正在读取的线程数（0个或多个）      private int waitingWriteCounter = 0;    // 等待写入的线程数（0个或多个）      private int writeThreadCounter = 0;     // 正在写入的线程数（0个或1个）      private boolean writeFlag = true;       // 是否对写入优先（默认为是）        // 读取加锁      public synchronized void readLock() throws InterruptedException {          // 若存在正在写入的线程，或当写入优先时存在等待写入的线程，则将当前线程设置为等待状态          while (writeThreadCounter > 0 || (writeFlag && waitingWriteCounter > 0)) {              wait();          }          // 使正在读取的线程数加一          readThreadCounter++;      }        // 读取解锁      public synchronized void readUnlock() {          // 使正在读取的线程数减一          readThreadCounter--;          // 读取结束，对写入优先          writeFlag = true;          // 通知所有处于 wait 状态的线程          notifyAll();      }        // 写入加锁      public synchronized void writeLock() throws InterruptedException {          // 使等待写入的线程数加一          waitingWriteCounter++;          try {              // 若存在正在读取的线程，或存在正在写入的线程，则将当前线程设置为等待状态              while (readThreadCounter > 0 || writeThreadCounter > 0) {                  wait();              }          } finally {              // 使等待写入的线程数减一              waitingWriteCounter--;          }          // 使正在写入的线程数加一          writeThreadCounter++;      }        // 写入解锁      public synchronized void writeUnlock() {          // 使正在写入的线程数减一          writeThreadCounter--;          // 写入结束，对读取优先          writeFlag = false;          // 通知所有处于等待状态的线程          notifyAll();      }  } |

我看出来了，架构师特意写了很多注释，免得我总是去烦他。

代码不解释了，看看注释吧，有疑问可以给我留言哦！

此时，Data 类还需要稍作改写。

|  |
| --- |
| public class Data {        ...        private final ReadWriteLock lock = new ReadWriteLock(); // 创建读写锁        ...        public String read() throws InterruptedException {          lock.readLock(); // 读取上锁          try {              return doRead(); // 执行读取操作          } finally {              lock.readUnlock(); // 读取解锁          }      }        public void write(char c) throws InterruptedException {          lock.writeLock(); // 写入上锁          try {              doWrite(c); // 执行写入操作          } finally {              lock.writeUnlock(); // 写入解锁          }      }        private String doRead() {          StringBuilder result = new StringBuilder();          for (char c : buffer) {              result.append(c);          }          sleep(100);          return result.toString();      }        private void doWrite(char c) {          for (int i = 0; i < buffer.length; i++) {              buffer[i] = c;              sleep(100);          }      }        ...  } |

同样的 Client 类，我再运行一把试试看，性能是否有提高呢？

…  
Thread-1 => 4444444444  
Thread-2 — 14000ms  
Thread-0 — 14001ms  
Thread-3 — 14000ms  
Thread-4 — 14000ms  
Thread-1 — 14001ms  
Thread-4 => IIIIIIIIII  
…

平均下来是 14000.4 毫秒，折合 14.0 秒，比以前快了 63%，而且输出的结果都比以前平稳（以前忽高忽低的）。

果然是架构师，真让我们这些程序员崇拜啊！

最后架构师过来，看到我在那里得意地笑。他拍拍我的肩，对我说：“别乐了，其实 JDK 1.5 中已经有 ReadWriteLock 了，我这个只不过是一个精简版而已，去看看 java.util.concurrent.locks.ReadWriteLock 吧，你一定会震精！”。

看来我真是孤陋寡闻啊，打开 JDK API 看到了 ReadWriteLock：

|  |
| --- |
| public interface ReadWriteLock {        Lock readLock();        Lock writeLock();  } |

可以通过 ReadWriteLock 接口来获取 ReadLock 与 WriteLock，它们都是 Lock 对象，这也是一个接口。

官方提供了一个 ReadWriteLock 接口的实现类 java.util.concurrent.locks.ReentrantReadWriteLock。

|  |
| --- |
| public interface Lock {        void lock();        void lockInterruptibly() throws InterruptedException;        boolean tryLock();        boolean tryLock(long time, TimeUnit unit) throws InterruptedException;        void unlock();        Condition newCondition();  } |

该接口中，有两个非常重要的方法：lock() 与 unlock()，分别表示“上锁”与“解锁”。

尝试用一下 JDK 的 ReadWriteLock 吧。

|  |
| --- |
| public class Data {        ...        private final ReadWriteLock lock = new ReentrantReadWriteLock(); // 创建读写锁      private final Lock readLock = lock.readLock();    // 获取读锁      private final Lock writeLock = lock.writeLock();  // 获取写锁        ...        public String read() throws InterruptedException {          readLock.lock(); // 读取上锁          try {              return doRead(); // 执行读取操作          } finally {              readLock.unlock(); // 读取解锁          }      }        public void write(char c) throws InterruptedException {          writeLock.lock(); // 写入上锁          try {              doWrite(c); // 执行写入操作          } finally {              writeLock.unlock(); // 写入解锁          }      }        ...  } |

再次运行一把看看效果。

使用了 JDK 的 ReadWriteLock，性能与自己实现的 ReadWriteLock 差不多，大家不妨自己试一下吧。

此外 JDK 还提供了一个更加简单的 ReentrantLock，它可以取代 synchronized，确保获取更高的吞吐率，一般可以这样来做：

以前的做法：

|  |
| --- |
| public synchronized void foo() {      ...  } |

现在的做法：

|  |
| --- |
| private final Lock lock = new ReentrantLock();    public void foo() {      lock.lock();      try {          ...      } finally {          lock.unlock();      }  } |