**多线程**

# 基本概念

线程状态图：



**线程共包括以下5种状态。**

**1. 新建状态(New): 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。**

**2. 就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。**

**3. 运行状态(Running) : 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。**

**4. 阻塞状态(Blocked) : 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：**

**(01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。**

**(02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。**

**(03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。**

**5. 死亡状态(Dead) : 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。**

# 多线程的代价

## 上下文切换开销

当CPU从执行一个线程切换到执行另外一个线程的时候，它需要先存储当前线程的本地的数据，程序指针等，然后载入另一个线程的本地数据，程序指针等，最后才开始执行。这种切换称为“上下文切换”(“context switch”)。CPU会在一个上下文中执行一个线程，然后切换到另外一个上下文中执行另外一个线程。

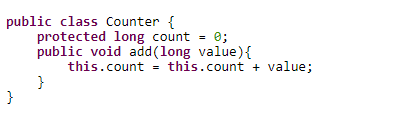
上下文切换并不廉价。如果没有必要，应该减少上下文切换的发生。

例如：服务端响应客户端的简单请求（耗时很小），这里没有使用线程池，导致大量的请求到服务端，然后服务端会给每个请求创建一个线程，然后导致大量的线程上下文切换，严重影响性能。

## 增加资源消耗

1. 每个线程都需要占用CPU资源。
2. 都需要占用一定内存。

# 竞态条件与临界区



竞态条件：即同时竞争获取count变量。

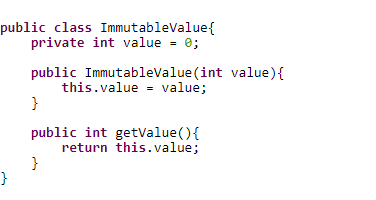
临界区：add方法会产生竞争。

说明：由于add操作非原子性，所以在多线程下，会导致计数不准备，例如count初始为1，A,B两个线程执行add，理想状态是count=3，但是A,B可能同时读取count=1，然后最终结果是2；这里涉及到了JMM（java内存模型）。

# 线程安全与不可变性

当多个线程同时访问同一个资源，并且其中的一个或者多个线程对这个资源进行了写操作，才会产生竞态条件。多个线程同时读同一个资源不会产生竞态条件。

我们可以通过创建不可变的共享对象来保证对象在线程间共享时不会被修改，从而实现线程安全。如下示例：



ImmutableValue类的成员变量value是通过构造函数赋值的，并且在类中没有set方法。这意味着一旦ImmutableValue实例被创建，value变量就不能再被修改，这就是不可变性。但你可以通过getValue()方法读取这个变量的值。

引申：多线程下线程安全的单利模式（Singleton）

# Java内存模型

1. 线程间的通信
   1. 共享内存：通过共享对象来完成隐式通信；java并发就是用的共享内存。
   2. 消息传递：典型的消息传递为线程间的wait/notify。
2. Java内存模型



* 1. 线程间的共享变量存储在主存，每个线程都会有这个共享变量的本地拷贝（带来线程不安全的原因，例如A还没来得及刷入主存，B就获取了老值）
  2. A线程对X赋值1，然后将X刷入主存；B线程读取主存，然后将X的值存储在本地内存。
  3. JMM通过控制主存与每个线程的本地内存之间的交互，来提供内存可见性保证。

1. JVM对JMM的实现



* 1. 每个线程都会有自己的线程栈，然后分配一定的本地内存（JVM中线程太多，也会导致内存不够用）。
  2. 基本数据类型直接保存在线程栈中；对象存储在堆中，栈中只存储一个引用（即存储对象的内存物理地址）；
  3. 基本数据类型，线程间如果要共享，必须通过主存来实现互相通信；共享对象则不用，因为对象都在堆（Object3，如果有变更操作，需要对obj3加锁，否则会导致数据不一致），每个线程中保存一个对象的引用。

1. 内存屏障
   1. 读屏障：强制线程去主存里面获取最新数据。（在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据）
   2. 写屏障：强制线程将写后的最新数据从线程本地内存刷入主存。（在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见）

# Java同步块（SyncMethod）

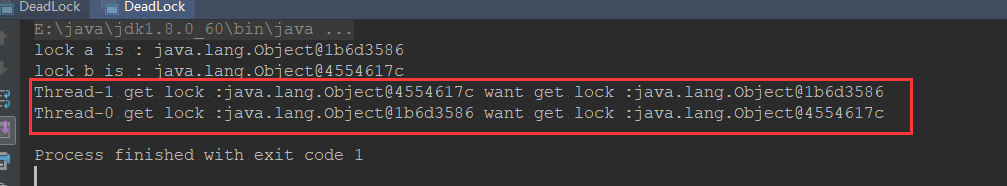
1. 实例方法同步
   1. 加的是对象锁，多线程使用不同的对象不用等待，使用了相同对象才等待。
2. 静态方法同步
   1. 加的是类锁，所线程只要调用了该方法，都必须等待。
3. 实例方法同步块（区别为方法可以并发访问，只有到同步块才等待）
   1. 同步块锁的是对象，同实例方法同步
   2. 同步块锁的是类，同静态方法同步
4. 静态方法同步块（区别为方法可以并发访问，只有到同步块才等待）
   1. 同步块锁对象，同实例方法同步
   2. 同步块锁类，同静态方法同步

# 死锁/避免方法

## java死锁

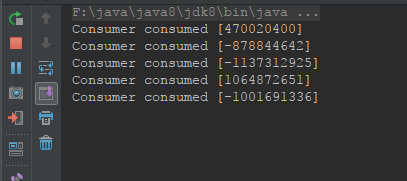
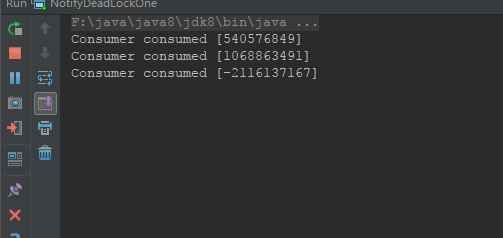
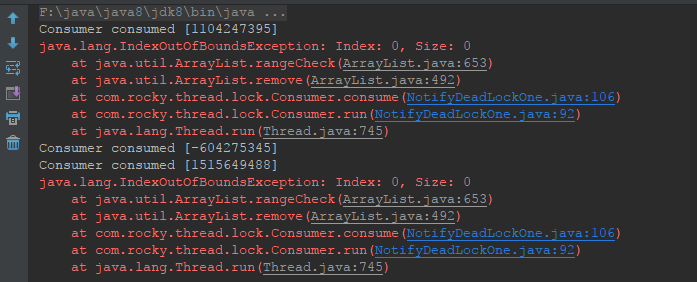
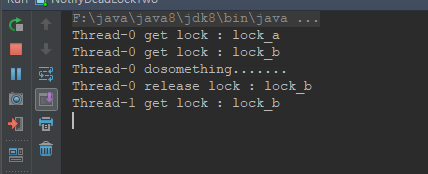
### 非顺序获取锁(DeadLock)

线程1获取A后，再次获取B，线程先获取B，再次获取A，然后两个线程互相等待。



解决方法：顺序获取锁，尽量避免在获取了一个临界资源后，再次获取另外一个，即避免锁嵌套。

### Notify导致的死锁

1. 生产者/消费者模式，在单个线程下使用notify唤醒对方线程，一切运行正常,每秒消费者消费一个消息。(NotifyDeadLockOne)
   1. 
2. 生产者/消费者模式，在多线程下使用notify唤醒对方线程，会造成互相等待唤醒而造成死锁,成功消费了3个消息就卡住不动了。(NotifyDeadLockOne)
   1. 
   2. 原因：假设有P1-P3三个生产者，C1-C3三个消费者，第一次P1放入一个消息，然后P1进入等待队列，并唤醒C1进行消费，C1消费完后进入等待，唤醒P2进行生产，P2生产完后使用notify随机唤醒了P3，P3判断队列里面有消息，然后自己又会进入等待队列，这样就导致了所有的线程都在等待。（notify是随机唤醒一个线程，如果随机唤醒了生产者，就会导致死锁）
   3. 解决方案：使用notifyAll来唤醒所有等待线程。
   4. 引伸问题：如果代码中的while判断换成if判断是否可行？
      1. 如果换成if多线程并发下会导致数组越界(单个生产者/消费者不会有问题，因为消费者被唤醒了，队列里面肯定有数据；多线程下，notifyall时候，可能消费者再次获取锁，如果使用if就不会再次判断队列是否有值，而直接或消费)
      2. 
3. Notify另外一种死锁情况，线程1获取了A,B两个锁，然后调用b.wait()释放B锁，进入等待队列，线程2获取B锁，然后再获取A锁，造成死锁。(NotifyDeadLockTwo)
   1. 
   2. 解决方案：类似非顺序获取锁，因此还是需要顺序获取锁，以及尽量避免锁嵌套获取。

## 数据库死锁

### 两个事务互相等待

出现原因：

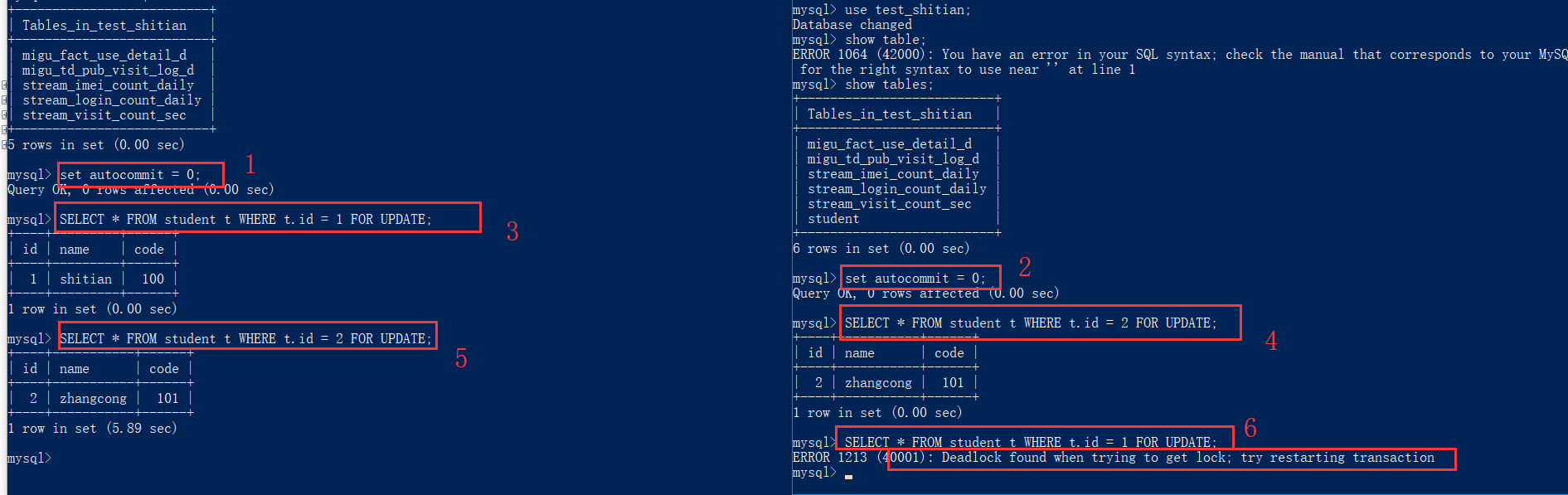
一个用户A 访问表A（锁住了表A），然后又访问表B；另一个用户B 访问表B（锁住了表B），然后企图访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续，同样用户B要等用户A释放表A才能继续，这就死锁就产生了。

|  |  |
| --- | --- |
| TX1 | TX2 |
| SET autocommit = 0;(取消自动提交) |  |
| SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 4 FOR UPDATE; |  |
|  | SET autocommit = 0; |
|  | SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 5 FOR UPDATE; |
| SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 5 FOR UPDATE; |  |
|  | SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 4 FOR UPDATE; |
|  | Lock wait timeout exceeded; try restarting transaction |

解决方案：

这种死锁比较常见，是由于程序的BUG产生的，除了调整的程序的逻辑没有其它的办法。仔细分析程序的逻辑，对于数据库的多表操作时，尽量按照相同的顺序进行处理，尽量避免同时锁定两个资源，如操作A和B两张表时，总是按先A后B的顺序处理， 必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源。

引伸：

1. InnoDB引擎会把写操作(insert/update/delete/select for update)当做事务来处理；
2. 所以单条UPDATE语句是不需要显式开启事务的.
3. MySQL默认会自动提交事务,每一条单独的查询都是一个事务；所以使用 SELECT FOR UPDATE 时应该关闭自动提交后手动开启事务 START TRANSACTION；否则 SELECT FOR UPDATE 的锁在事务自动提交后就释放了.
4. 

### 共享锁升级独占锁

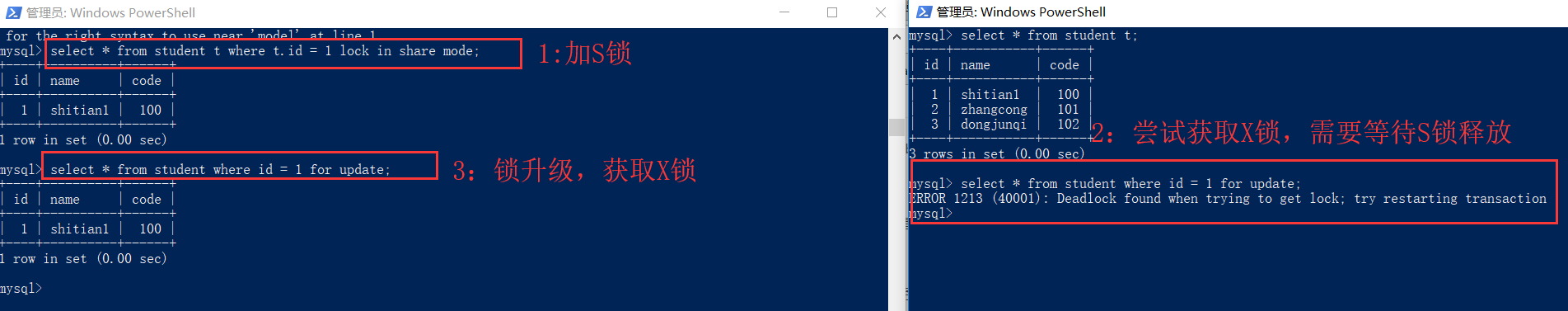
出现原因：

用户A查询一条纪录（获取共享锁S）；用户B修改该条纪录（获取排它锁X，但是需要等待A的共享锁S释放），这时用户A需要修改该条记录，锁的性质由查询的共享锁企图上升到独占锁（尝试获取排它锁X，不会释放S锁），而用户B里的独占锁由于A有共享锁存在所以必须等A释放掉共享锁，而A由于B的独占锁而无法上升的独占锁也就不可能释放共享锁，于是出现了死锁。这种死锁比较隐蔽。

解决方案：

A）使用乐观锁：通过版本控制实现无锁并发（查询时候获取版本号，更新时候比对版本号是否相同），当高并发下比较容易大量的更新不成功。

B）使用悲观锁：查询时候就使用select for update获取排它锁，性能严重下降。



### 9.2.3 mysql更新导致死锁

出现原因： mysql的innodb默认使用的是行锁，支持表锁；行锁锁的是索引，并不是记录；

update table set num = 100 where normal\_index = 1；

第一步：使用普通索引列normal\_index，所以获取了这个普通索引的锁。

第二步：获取主键锁，然后根据主键索引锁定该记录，修改数据。

Update table set normal\_index = 2 where pk = 100；

第一步：根据主键索引修改数据，所以先获取主键索引。

第二步：因为需要更新普通索引列数据，所以还需要获取普通索引。

如果上述的两个sql同时执行，就会导致死锁，互相等待。

解决方案：更新尽量带上主键，保证优先获取主键锁，避免死锁。

# 阻塞队列

# 饥饿与公平

1. 导致饥饿的原因
   1. 高优先级线程吞噬所有的低优先级线程的CPU时间
      1. 每个线程都有优先级，可以人工设置，优先级（1-10）越高，获取CPU时间越多，优先级表示的行为状态依赖运行的平台，尽量不要人工修改优先级。
   2. 线程永远阻塞在同步代码块外
      1. Synchronized为非公平锁，所以可能存在某一线程永远获取不到锁
   3. 线程等待被唤醒
      1. Notify是随机唤醒一个线程，所以也可能存在某一线程永远唤醒不过来
2. 使用公平锁，解决线程饥饿
   1. 使用ReentrantLock（true），默认是非公平锁，传true表示使用公平锁
3. 公平锁：所有线程按照先后顺序进入等待队列，新来的线程永远加入队尾
   1. 公平锁性能一般会比非公平锁低
4. 非公平锁：线程释放锁后，唤醒队列中的第一个线程，然后来尝试获取锁；此时来了一个新线程，非公平锁允许该新线程也来获取锁，成功则执行，失败则加入队尾。
   1. 非公平锁快的原因：恢复一个阻塞的线程到该线程变为可运行，之间存在比较多的延迟，例如加载上次执行的中间信息等。
      1. A释放锁，然后唤醒B，B处于阻塞到就绪的恢复状态；C新到（C耗时很小），然后获取锁成功，并且执行完，然后释放锁；此时B恢复完成，然后获取锁成功，然后运行；这就是一个双赢的局面。
      2. 如果线程占有锁的时间很长，或者在获取锁的耗时也很长，则使用公平锁（这种情况下，插队带来的好处基本不会出现，因为获取锁的时间很长，唤醒的线程已经彻底恢复了）。

# 常用的实现多线程方式

1. 创建方式
   1. 继承Thread对象
   2. 实现Runnable接口
   3. 实现Callable接口（CallableTest）
2. 不同之处
   1. 继承Thread不适合资源共享，每个new出来的Thread都是新的。
   2. Callable规定的方法是call()，而Runnable规定的方法是run()
   3. 运行Callable任务可拿到一个Future对象，Future表示异步计算的结果。它提供了检查计算是否完成的方法,以等待计算的完成,并检索计算的结果.通过Future对象可了解任务执行情况,也可取消任务的执行,还可阻塞式获取任务执行的结果
      1. Future的cancel方法，入参为boolean，如果为true，则表示中断线程；为false，如果该线程已经在运行则不处理，让线程跑完，线程在等待队列，则后续线程池不处理该线程（简单来说，传入false参数只能取消还没有开始的任务，若任务已经开始了，就任由其运行下去。）。
   4. call()方法可抛出异常，而run()方法是不能抛出异常的，只能try/catch

# 常用方法解析

## 线程等待/唤醒

## 线程让步

## 线程休眠

## 线程加入

## 线程中断

# 如何中断线程

# 并发编程的三个概念

## 原子：一个操作要么全部成功，要么全部失败。

## 可见：多线程模式下，A线程对共享对象做了修改，其他线程要能立刻看到

## 有序：程序执行的顺序按照代码的顺序（JVM为了提高效率会指令重排序，JVM保证重排序后，结果不会有变化）

**//线程1:**

context = loadContext(); //语句1

inited = true; //语句2

**//线程2:**

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

说明：上述情况多线程下可能会有问题

1. 线程1先执行，然后由于指令重排序，语句2先执行
2. 线程2执行while循环，发现不满足条件，跳出循环获取context对象操作
3. 线程1此时还没有来得及初始化context
4. 线程2会报空指针

# Volatile关键字

## 保证可见性

通过内存屏障实现，对于volatile修饰的关键词做如下两步操作：

1. 线程写入操作，会在写入后，加入写屏障，强制数据刷入主存。
2. 线程读操作时候，会在读之前，加入读屏障，强制从主存读数据。

## 保证部分有序性

**//线程1:**

**Volatile context = loadContext(); //语句1**

**inited = true; //语句2**

还是以上面的线程1为例，给context加上volatile修饰，jvm在指令重排序的时候，保证context上面的指令与context下面的指令不能重排序，即上面的指令能到下面，下面的指令不能到上面，上面的指令可以重排序，下面的指令也能重排序；

# ThreadLocal

## 概述

## 导致内存泄漏的情况

# 锁

## 内置锁（Sync）

## ReentrantLock

### 公平锁

### 非公平锁

## 8.3锁重入

## 8.4读写锁

## 8.5条件锁

## 8.6分布式系统中使用内置锁是否有效

# 线程池

## 线程池参数含义

## 线程池类层次结构

## 源码解析

## 线程池处理流程

# 多线程工具类

## CountDownLatch

## CyclicBarrier

## Semaphore

## Exchange

## Phaser

## Fork/Join

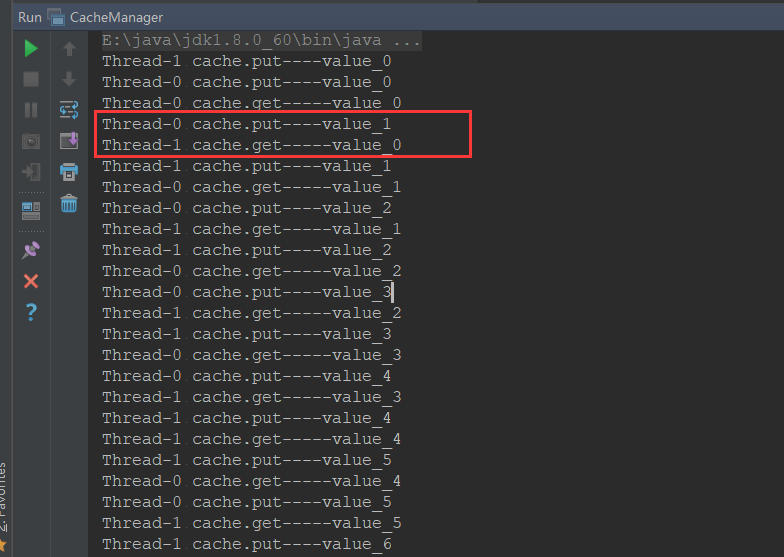
# JUC（java.util.concurrent）原子类（CAS）

# 多线程下静态变量与实例变量

# 并发编程陷阱

## 同步不完全(CacheManager)

一个简单的缓存类，put操作使用同步，get操作没有使用同步，在多个线程同时put/get同一个key的时候可能导致读取老的数据【将get操作也设置为同步就可以解决该问题，可以使用读写锁优化】。



如图，线程0已经put值为“value\_1”，但是线程1还是读取的老的值“value\_0”。

总结：可以并发读，不可以并发写，读时不能写，写时不能读。

## InterruptException无处不在

## 使用Thread.interrupt()中断线程

## Volatile与变量脏读

## Double check

## 高并发环境下使用性能较低的map

## 读多写少使用Sync锁导致性能下降

## 不要吞食CountDownLatch的线程异常