**多线程**

# 基本概念

# 多线程的代价

## 设计更加复杂

## 上下文切换开销

## 增加资源消耗

# 竞态条件与临界区

# 线程安全与共享资源

# 线程安全与不可变性

# Java内存模型

# Java同步块

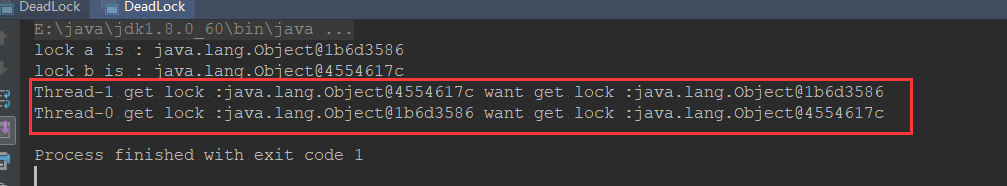
# 线程通信

# 死锁/避免方法

## java死锁

### 非顺序获取锁

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

## 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

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。

### 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

# 常用的实现多线程方式

## 创建Thread对象

## 实现Runnable接口

## 实现Callable接口

# 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

## 可见

## 有序

# Volatile关键字

## 保证可见性

## 保证部分有序性

# 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）

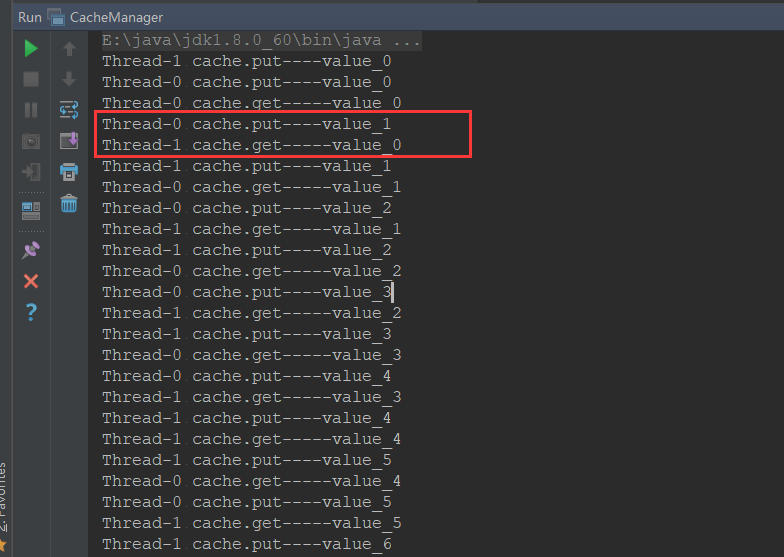
# 多线程下静态变量与实例变量

# 多线程下访问同步/非同步/静态方法

# 并发编程陷阱

## 同步不完全

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