多线程

1.进程与线程

进程是OS中资源分配的最小单元

线程是OS中任务分配的最小单元

创建与销毁一个线程的开销要比一个进程小得多，线程间通信也比进程间通信容易的多。

线程间通信:join()、wait/notify、yield、sleep

2.多线程常用操作方法

sleep:运行 -> 阻塞，当前线程立即交出CPU，进入阻塞态，不会释放对象锁

yield:运行 -> 就绪，系统调度交出CPU，进入就绪态，不会释放对象锁，只会让相同优先级的线程获取CPU

join:运行 -> 阻塞,进入阻塞态，会释放对象锁。(join内部就是wait方法)

wait:运行 -> 阻塞，会释放对象锁,必须与synchronized搭配使用

notify:阻塞 -> 就绪,必须在同步方法或同步代码块中使用

创建 就绪 运行 终止

阻塞

守护线程与用户线程:

默认创建的都是用户线程-主线程

守护线程:在后台执行，只要JVM中存在任意一个用户线程没有终止，守护线程就一直在运行，

当JVM中最后一个用户线程终止，守护线程会随着JVM一同停止。GC线程

3.线程同步

多个线程以下三个特性(原子性、可见性、有序性)任意一个不满足，都存在线程安全问题

解决:

3.1 synchronized实现线程安全

同步代码块

synchroinzed(锁的对象) {}

-普通对象(对象锁)

-类.class

同步方法

-成员同步方法 锁的是当前对象this

-静态同步方法 锁的是当前类.class

synchronized底层实现(对象的Monitor机制)---🡪对应的指令：monitorenter monitorexit

任意一个对象都有Monitor，synchronized对象锁实际上就是获取该对象的Monitor。

当前线程要想获取到该锁的Monitor的流程:

先判断锁对象Monitor计数器值是否为0,

为0表示此时Monitor还未被任何线程持有，当前线程获取Monitor，并且将持有线程置为自己，将Monitor的值+1.

不为0表示此时Monitor已经被线程持有，判断持有线程是否是当前线程，若是，Monitor值再次+1；

若持有线程不是当前线程，线程进入阻塞态，直到Monitor的值减为0.

可重入锁:持有锁的线程再次获取锁

JDK1.6之后关于synchronized优化：

CAS(compare and swap)

CAS(O,V,N) O:当前线程认为主内存的值 V:主内存中的实际值 N:希望更新的值

执行CAS的线程，只有首次获得CPU时间片段的线程才能修改主内存的值

例如： 有20张票，int ticket = 20;

有三个线程在卖票

线程 - 1

Ticket-1 = 20

ticket-1 = 19

线程 – 2

Ticket-2 = 20

线程 – 3

Ticket-3 = 20

CPU----线程-3执行 – 5

Ticket-3 = 15

此时执行CAS(O,V,N) ---- CAS(20,20,15)

此时 O == V 可以将 V修改为N

CPU----线程-1 执行 –1

此时执行CAS

CAS(20,15,19)

此时 O != V 不能修改主内存值

需要将O改为15 进行下一下循环

自旋：处理器上跑无用指令，但是线程不阻塞

自适应自旋：（重量级锁的优化）

JVM给一个时间段，在该时间段内，线程是自旋状态，若在该时间段内获取到锁，下一次适当延长自旋时间；否则将线程阻塞，下一次适当缩短自旋时间。

随着锁竞争的激烈程度不断升级，没有降级过程。

偏向 -> 轻量级锁 -> 重量级锁(自适应自旋)(jdk1.6之前synchronized默认实现)—线程获取锁失败进入阻塞态(OS用户态->内核态)

JDK1.6 默认先偏向锁（首先就是偏向锁）

偏向锁(乐观锁，锁一直是一个线程在来回的获取)：

当线程第一次获取锁时，将偏向锁线程置为当前线程，以后再次获取锁时，不再有加锁与减锁过程，只是简单判断下获取锁线程是否是当前线程。

轻量级锁：在不同时间段内有不同的线程尝试获取锁

每次锁的获取都需要加锁与解锁过程。

重量级锁：在同一时刻有不同线程尝试获取锁。

锁粗化

将多次连续的加减锁过程粗化为一次大的加减锁过程

public class BigLock {

private static StringBuffer sb = new StringBuffer();

public static void main(String[] args) {

sb.append(“hello”);

sb.append(“world”);

sb.append(“test”);

}

}

锁消除

在没有多线程共同访问的场景下，将锁直接消除。

public class BigLock {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer();

sb.append(“hello”);

sb.append(“world”);

sb.append(“test”);

}

}

死锁

死锁的产生原因：以下四个条件同时满足才会产生死锁

1. 互斥：共享资源X，Y只能被一个线程占用
2. 占有且等待：线程1已经取得共享资源X，同时在等待资源Y，并且不释放X
3. 不可抢占：其他线程无法抢占线程1已经占用的资源X
4. 循环等待：线程1等待线程2的资源，线程2等待线程1的资源

死锁的现象：程序出现“假死”现象

死锁的解决：破坏任意一个条件

查看死锁的jdk内置工具：jps jstack

Lock体系是由于synchronized无法解锁死锁问题而产生的

JDK1.5 引入Lock体系来优雅的解决死锁问题

1. Lock的使用格式

try{

lock.lock();

}catch(Exception e){

}finally{

lock.unlock();

}

1. Lock接口的重要方法（三个新特性）

响应中断

2.1 void lockInterruptibly() throws InterruptedException; 破坏死锁的第三个条件:不可抢占

非阻塞式获取锁，若获取锁失败，线程继续执行，不再阻塞

* 1. boolean tryLock(); 破坏死锁的第二个条件：占有且等待，因为tryLock()不会等待

支持超时，获取锁失败的线程等待一段时间后若还未获取到锁，线程退出

* 1. boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
  2. Lock的常用子类

ReentrantLock:可重入锁(Lock接口中常用的子类，语义与synchronized基本一致，也是独占锁的实现)

同步：互斥，任意时刻只有一个线程

同步队列：所有获取锁失败的线程进入同步队列排队获取锁

等待队列：调用wait的线程置入等待队列，等待被唤醒(notify)

面试题：synchronized与ReentrantLock的关系和区别？

1. synchronized与ReentrantLock都属于独占锁的实现，都支持可重入。
2. 区别：
3. synchronized是关键字，JVM层面的实现；ReentrantLock是Java语言层面实现的(“管程”)
4. ReentrantLock具备一些synchronized不具备的特性，如响应中断、支持超时、支持非阻塞式的获取锁，可以实现公平锁（ReentrantLock默认也是非公平锁）
5. synchronized只有一个等待队列，而Lock调用newCondition()产生多个等待队列，还支持读写锁。(生产者消费者模型中，当生产者和消费者线程都调用wait()后，进入同一个等待队列，如果全部唤醒后，最新唤醒的是生产者线程，那又将阻塞)

Condition：Lock

await/signal

变种面试题：synchronized与Lock的关系与区别：

1. synchronized与ReentrantLock都属于独占锁的实现，都支持可重入。
2. 区别：
3. synchronized是关键字，JVM层面的实现；ReentrantLock是Java语言层面的实现
4. ReentrantLock具备一些synchronized不具备的特性，如响应中断、支持超时、支持非阻塞式的获取锁，可以实现公平锁（ReentrantLock默认也是非公平锁）
5. synchronized只有一个等待队列，而Lock调用newCondition()产生多个等待队列，还支持读写锁。(生产者消费者模型中，当生产者和消费者线程都调用wait()后，进入同一个等待队列，如果全部唤醒后，最新唤醒的是生产者线程，那又将阻塞)

读写锁：读者写者问题

读线程：读读异步，读写同步

写线程：写写同步

读写锁实现：ReentrantReadWriteLock（可重入读写锁）(实现了缓存HashMap + ReentrantReadWriteLock)

读锁：ReadLock，多个线程在同一时刻可以共同取得该读锁

写锁：WriteLock，独占锁，多个线程在同一时刻只有一个线程可以取得该锁

共享锁：多个线程可以同时取得该锁 读锁 ReadLock

共享锁！=无锁

当写线程开始工作，所有其他线程(包含读线程)全部进入阻塞状态。

JDK1.8 StampedLock 更加乐观的锁实现，性能比ReentrantReadWrriteLock还高

wait/notify：在synchronized中使用

当等待队列中的线程被唤醒后，插入到同步队列的队尾。

公平锁：等待时间最长的线程优先获取锁。（synchronized不具备公平锁）

interrupt()方法只是将线程状态置为interrupt，并不是将线程中断。

InterruptedException异常与wait()/sleep()/interrupt()/interrupted()方法搭配使用，才能产生。

synchronized不响应中断

juc包下工具类：CAS+Lock

1. 闭锁CountDownLatch（CountDownLatch对象在计数器减为0时不可恢复，只会阻塞调用await方法的线程，对象不可重复使用）

public CountDownLatch(int count) ：count表示需要等待的线程个数；

public void countDown() ：计数器值-1（类似运动员线程）；

public void await() throws InterruptedException ：等待线程调用该方法进入阻塞态，直到计数器减为0。

1. 循环栅栏CyclicBarrier（计数器的值可以恢复reset(),CyclicBarrier的对象可以重复使用）

public CyclicBarrier(int parties) ：parties表示需要有多少个线程同时暂停以及恢复执行

public int await() ：cyclicBarrier 计数器-1，当减为0时，所有阻塞线程同时恢复执行

public CyclicBarrier(int parties, Runnable barrierAction) ：多个线程在恢复执行之前，任意挑选一个线程执行barrierAction任务后，再同时恢复执行。

1. 交换器Exchanger

Exchanger用于两个线程交换数据，当缓冲区只有一个线程时，该线程会阻塞直到配对成功再交换数据恢复执行。

1. 信号量Semaphore（应用在类似8个工人，5台设备这种不匹配的问题）

public Semaphore(int permits) ：表示许可的数量

public Semaphore(int permits, boolean fair) ：等待时间最长的线程最先获取到许可

public void acquire() ：申请许可，尝试获取许可（获取一个许可）

public void acquire(int permits) ：申请许可，尝试获取多个许可

public void release() ：释放许可

线程池

ExecutorService：普通调度池

void execute(Runnable r)

Future submit(Callable || Runnable)

ScheduleExecutorService：定时调度池

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,

long initialDelay,

long period,

TimeUnit unit);

ThreadPoolExecutor：线程池核心类

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler);

常见的阻塞队列：

LinkedBlockingQueue：基于链表的无界阻塞队列---------内置的固定大小线程池就采用次队列

SynchronousQueue：不存储元素的无界阻塞队列---------内置的缓冲线程池就采用此队列

Executors：线程池的工具类

----单线程池

newSingleThreadExecutor()

----固定线程池

newFixedThreadPool (int nThreads)

----缓存池（当提交任务速度>>任务处理速度，不断产生新线程；任务处理速度>提交任务速度，只有一个线程）

newCachedThreadPool()

----定时调度池

newScheduledThreadPool(int corePoolSize)

核心池

最大线程池

阻塞队列

拒绝策略

线程池考点：

为何推荐线程池来新建线程？

使用线程池有一下优点：

（1） 降低资源消耗:通过重复利用已创建的线程降低线程创建和销毁带来的消耗。  
（2） 提高响应速度:当任务到达时，任务可以不需要等待线程创建就能立即执行。  
（3） 提高线程的可管理性:使用线程池可以统一进行线程分配、调度和监控。

1. 线程池的工作流程
2. 如何自定义线程池

----核心线程池类ThreadPoolExecutor参数配置

----线程池工作线程Worker

1. 在何种场景下选用何种线程池

----任务同步处理时，选择单线程池

----系统资源紧张，适用于负载较重的服务器，选择固定大小线程池

----服务器负载较轻，适用于处理很多短期异步小任务，选择缓存池

----需要执行定时任务的场合，选择定时调度池

JMM：Java内存模型（关于并发程序的内存模型----逻辑模型）

1. JMM的工作流程

工作内存：每个线程创建时分配的空间，线程私有。所有变量的读写均在工作内存中完成

主内存：所有线程共享的内存区域，存放所有共享变量(类的实例变量、静态变量、常量)

1. JMM三大特性

只有以下三个特性同时满足，才是线程安全的代码

原子性：基本数据类型的访问读写都属于原子性操作

若需要更大范围的原子性，需要使用synchronized或lock来保证。

可见性：任意一个线程修改了共享变量的值，其他线程能够立即得知此修改

synchronized、volatile（可见性、不可指令重排）、final实现可见性

有序性：逻辑上写在前面的代码优先发生于写在后面的代码

1. volatile变量的特殊规则（volatile的写操作优先于读操作）
   1. 可见性

volatile boolean shutdownRequested;

public void shutdown() {  
shutdownRequested = true;

}

public void work() {  
while(!shutdownRequested) {  
//do stuff

}

}

* 1. 禁止指令重排

Int x = 1;

Int y = 2;

----------- volatile int z = 3; ----------

X += 2;

Y += 3;

1. Volatile代码既不会提前也不会滞后执行
2. Volatile之前的代码已经全部执行完毕，且结果对后续代码可见；volatile之后的代码都还没有开始执行。

**Double-Check singleton**

public class SingleTon {

private int age;

private String name;

private static volatile SingleTon singleTon;

private SingleTon(){

age = 0;

name = null;

}

public static SingleTon getInstance() {

if(singleTon == null) {

synchronized (SingleTon.class) {

if(singleTon == null) {

// a.堆开空间

// b.属性初始化

// c.栈开空间指向堆内存(此时single !=null)

singleTon = new SingleTon();//这里存在指令重排问题，有可能出现c在b之前执行，就是导致产生的single!=null，但是属性没有初始化

}

}

}

return singleTon;

}

}

深浅拷贝：拷贝的是对象

浅拷贝：拷贝对象内部的引用复用，并不会产生新对象

深拷贝：拷贝对象内部的引用也会产生新对象

实现深拷贝：

1. 递归实现Cloneable接口
2. 使用序列化（推荐）

内存区域划分

线程私有(随着线程的创建而创建，随着线程销毁而回收，生命周期与线程相同，不同线程间相互隔离)

程序计数器：记录当前线程所执行的字节码行号，唯一一块不产生OOM的区域，native方法值为0；

虚拟机栈：Java方法的内存模型（局部变量表是其一个子集）

本地方法栈：本地方法的内存模型 -Xss设置栈的大小 StackOverflowError OOM

Hotspot：本地方法栈和虚拟机栈合二为一

线程共享

堆（GC堆）：所有的对象实例以及数组实例 -Xms设置堆的最小值 -Xmx设置堆的最大值

方法区（JDK8之前叫永久代）：变量、常量、加载的类信息

运行时常量池(方法区的一部分)：符号引用、字面量(接写出来的值，例如：10,”hello”)

判断对象是否存活 -> 对象是否原地去世？(finalize()) -去世> 如何回收已不再存活对象(GC算法)

判断对象是否存活：

1. 引用计数法(C++智能指针、PythonGC)：无法解决循环引用问题(Java没有采用它)
2. 可达性分析算法

GC Roots：

1. 类中的静态变量、常量
2. 栈中的局部变量

\*\*\*\*\*JDK1.2之后关于引用的扩充\*\*\*\*\*

强引用：程序中普通存在的，类似Test test = new Test();只要对象被强引用指向，GC无法回收此对象。

软引用：描述有用但不必须对象(缓存对象)，JDK1.2之后SoftReference来描述软引用。当对象仅被软引用指向，内存够用时不回收，即将发生OOM，一次性回收掉仅被软引用指向的对象。

弱引用：描述有用但不必须对象，仅能存活到下一次GC之前，GC开始时，无论内存是否够用，都会回收掉仅被弱引用指向的对象，JDK1.2之后WeakReference来描述弱引用。

虚引用：与对象的生存周期无关，当对象被GC时，由JVM发出一个回收通知，JDK1.2之后PhantomReference描述虚引用。

对象自我拯救finalize()

final、finally、finalize区别？

- final：用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖和类不可被继承（不能再派生出新的子类）

- finally：作为异常处理的一部分，他只能在try/catch语句时，并且附带一个语句块，表示这段语句语句最终一定能被执行，经常被用在需要释放资源的情况下。

- finalize: 是Object类的一个方法，在垃圾回收执行器执行时会调用被回收对象的finalize方法，可以覆盖此方法来实现对其他资源的回收，例如关闭文件。需要注意：一旦垃圾回收器准备好释放对象占用的空间，将首先调用其finalize方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。

如何进行垃圾回收？

Java采用分代回收算法，将内存划分为新生代与老年代，其中新生代采用复制算法，老年代采用标记-整理算法。

新生代(对象默认在新生代产生,对象存活率较低)：复制算法（Eden、Survior(from,to)）

老年代(对象存活率较高)：标记-整理算法

请解释MinorGC与FullGC(Major GC)

MinorGC又叫新生代GC，采用复制算法，效率比FullGC高。

FullGC又叫老年代GC，采用标记-整理算法，效率一般比MinorGC慢10倍。

出现了MajorGC，经常会伴随着至少一次的MinorGC。

类集（多线程+数组结构）----JDK1.2

1. ArrayList、LinkedList、Vector的关系和区别（源码）
2. ArrayList基于数组实现List，采用懒加载策略，当一次调用add方法时，数组才会初始化为10（默认值）。ArrayList扩容为原先数组的1.5倍，ArrayList采用异步操作，线程不安全，性能较高，普通的插入ArrayList已经能够满足要求，因为默认为尾插。
3. Vector(JDK1.0)基于数组实现List，当产生Vector对象时，就初始化为大小为10的数组，扩容为原先数组的2倍，Vector采用同步方法保证线程安全，性能低（读读互斥）。
4. LinkedList基于双向循环链表实现的List，采用异步处理，线程不安全。在任意位置插入或删除时考虑使用LinkedList或需要使用队列的场合。
5. 请描述fail-fast策略------ ConcurrentModificationException（java.util包下除了TreeMap，其他都是fail-fast）

在迭代输出的过程中修改了集合的结构(remove、add)抛出此异常，为了保证多线程场景下多线程取得数据正确性。

如何产生：

ModCount记录当前集合被修改次数

expectedModCount迭代器内部记录当前集合修改次数，当取得集合迭代器时赋值为当前modCount；

modCount != expectedModCount

什么是快速失败策略：优先考虑异常情况，当异常产生时，直接抛出，程序终止。

如何解决：

1. 迭代输出时尽量不要修改集合结构
2. 使用juc包下的线程安全集合如CopyOnWriteArrayList、ConcurrentHashMap(fail-safe集合)
3. Set与Map有什么关系

Set实际就是Map，Set将元素存储到了内部Map的Key，共享一个null的Object对象作为value值。

* 1. hashcode与equals
  2. 如何实现第三方类的比较(内部排序、外部排序)

1. HashMap、Hashtable、TreeMap的关系与区别
2. 为何在线程安全的场合要使用ConcurrentHashMap，ConcurrentHashMap如何高效实现线程安全？
3. ConcurrentHashMap在JDK1.7与JDK1.8的区别

递归函数：

----语义：函数能干嘛

----出口在哪（终止条件）

数据库事务ACID特性：

A：原子性：一组操作要么同时发生，要么一个都不发生

C：持久性：指的是只要事务成功结束，它对数据库所做的更新就必须**永久保存**下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。

I：隔离性：指的是在并发环境中，当不同的事务同时操作相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，**事务不会查看到中间状态的数据**。

D：一致性：在一个事务执行之前和执行之后数据库都必须处于一致性状态。如果事务成功地完成，那么系统中所有变化将正确地应用，系统处于有效状态。如果在事务中出现错误，那么系统中的所有变化将自动地回滚，系统返回到原始状态。

只要有一个字符串变量的拼接，javac会将String变量转换为StringBuilder

StringBuilder sb = sb.append(str2).append(str3)

toString()方法中会new String()

字符串常量拼接，内存还是在常量池中