**多线程**

**关于HashMap，HashSet，和ArrayList集合的线程安全以及解决方案**

​ 其实HashMap，HashSet，和ArrayList接口的这些都是线程不安全的，他们在多线程操作的时候都会产生线程安全的问题，具体体现在多线程的添加的时候都会出现ConcurrentModificationException这个异常，那是由于多线程的修改同一个数据引起的

那么如何解决呢

解决ArrayList可以使用的方法

使用Vector修饰  
List<String> map = new Vector<>();  
​  
使用Collections的工具类给List上锁  
List<String> map = Collections.synchronizedList(new ArrayList<>());  
​  
使用JUC工具包CopyOnWriteArrayList的写时复制List  
List<String> map = new CopyOnWriteArrayList<>();

解决HashMap的问题

使用Collections的工具类给Map上锁  
Map<String,Object> hash = Collections.synchronizedMap(new HashMap<>());  
​  
使用ConcurrentHashMap并发工具包  
Map<String,Object> hash = new ConcurrentHashMap<>();

解决HashSet的问题

使用Collections的工具类给Set上锁  
Set<String> set = Collections.synchronizedSet(new HashSet<>());  
​  
使用JUC工具包CopyOnWriteArraySet的写时复制List  
Set<String> set = new CopyOnWriteArraySet<>();

**AbstractQueuedSynchronizer（AQS）（JUC并发包实现的基类）**

AbstractQueuedSynchronizer的使用依靠继承来完成，子类通过继承自AQS并实现所需的方法来管理同步状态。

提供了一个基于FIFO队列，可以用于构建锁或者其他同步装置的基础框架

AQS可以同时实现排它锁和共享锁模式

使用方法是继承，子类通过继承并通过实现它的方法管理其状态｛acquire 和release｝的方法操纵状态

**怎么检测一个线程是否拥有锁**

​ java.lang.Thread中有一个方法叫holdsLock() ，返回true表示当前线程有锁

**同步和异步有什么区别**

同步异步通常形容方法的调用过程

同步：

​ 同步就是方法的执行必须有序，也就是我调用一个方法，这个方法必须执行完我才能继续执行；即有序执行

异步：

​ 异步就是在方法调用的时候不用等待也能执行其他的方法，例如我在调用一个方法时我还能调用其他的方法，不必等待前面的方法执行完再执行下一个

**并发、并行有什么区别**

​ 并发：多个任务轮流争取CPU的资源，多个任务交替执行，看起来好像多个程序一起执行其实还是多个任务在共享一个CPU

​ 并行：每个线程使用一个CPU，真正的一起执行

**临界区（锁，资源争抢）**

​ 表示一个公共资源或者共享数据，可以被多个线程使用;每一次，只能有一个线程使用它，一旦临界区资源被专用(加锁)，其他线程想要使用这个资源，就必须等待；

**阻塞、非阻塞有什么区别**

​ 阻塞：当一个线程占用了临界资源（被上锁的临界资源），那么当其他线程想要访问临界资源的时候，就必须等待，等待会导致线程挂起，即将当前线程放入到等待队列中，直到被唤醒；

​ 非阻塞：线程之间的执行不受共享资源的状态影响，即不会被挂起；

**线程之间的锁机制会引发哪些问题**

​ 死锁：线程间相互的调用，所引起的锁资源获取不到导致无限等待循环，例如A需要执行B后才能释放锁，C需要执行A后释放锁，B有需要C执行后释放锁，导致锁释放不出来。没人拿的到这个锁造成的死锁

​ 活锁：两个线程互相主动将资源释放，让给对方，但没有一个线程可以同时拿到所有资源而正常执行

​ 饥饿： 线程因为某种原因(例如线程的优先级比较低)一直等待资源，无法获取所需要的资源，导致一直无法执行

**进程，线程的区别是什么**

​ 进程：是系统进行资源分配和调度的基本单位，进程是基本执行的实体，是线程的容器，运行一个程序，就相当于运行一个进程。当我们程序有很多小的任务需要执行的时候，我们使用进程进行任务之间的切换，比较费时，且数据交换麻烦，因此引入线程

​ 线程：是程序执行的最小单位，线程之间可共享数据，线程之间切换的成本较低，一个进程包含很多线程；

**多线程的一些性质**

​ 原子性：Atomicity

​ 原子性指的是一个操作不可能被中断，即如果多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰；

​ 可见性：Visibility

​ 可见性是指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道这个修改；对于串行来说，不存在这个问题，此问题只针对于并行；

​ 有序性：Ordering

​ 当程序执行时，有可能会进行指令重拍，重拍后的指令和原指令的顺序未必一致；

**线程的状态有哪些**

​ 创建New

​ 新建状态表示刚刚创立的简称，这个线程还没有开始执行，需要等到start()方法调用时，才表示线程执行；

​ 启动Runnable

​ 运行状态表示线程所需的一些资源都已经准备好了，可以执行

​ 阻塞Blocked

​ 阻塞状态，表示当遇到Synchronized同步块是，就会进行blocked阻塞状态，这个线程会暂停执行，直到获取请求的锁

​ 停止Waiting

​ Waiting/Time Waiting:Waiting等待状态会进入一个无时间限制的等待

​ Time\_Waiting会进行一个有时间的等待；

​ 结束Terminated

​ 有限等待Time\_Waiting

**线程有哪些方法**

终止线程stop()

线程中断lnterrupt()

睡眠sleep()

等待wait(),notify()

挂起suspend()

等待线程结束join()、谦让yield()：

synchronized关键字

**Java实现多线程有哪几种方式**

​ 有四种

​ 1、继承Thread类

​ 2、实现Runnable接口

​ 3、实现Callable接口

​ 4、线程池

**Callable和Future的了解**

​ Callable接口代表一段可以调用并返回结果的代码

​ Future接口表示异步任务，是还没有完成的任务给出的未来结果

​ 所以说Callable用于产生结果，Future用于获取结果。

**CyclicBarrier和CountDownLatch的区别是什么？**

​ CountDownLatch，使用的是减计数的方式，计算为0时释放所有等待的线程，计数为0时无法重置，调用countDown方法进行计数减一，调用await方法进行阻塞，对计数没有任何的影响

​ CyclicBarrier，使用的是加计数的方式，计数达到指定值时释放，计数重置为0重新开始，调用await时+1，如果加1不等于构造方法的值则线程阻塞，他是可重复利用的

**线程池的接口和依赖的关系**

​ Executor 父类接口

​ ExecutorService 继承Executor

​ AbstractExecutorService 实现ExecutorService

​ ThreadPoolExecutor 继承AbstractExecutorService

**线程池的几个参数**

int corePoolSize //核心线程数,允许同时执行任务的最大线程数

int maximumPoolSize // 最大线程数，允许同时处理任务的最大线程数

long keepAliveTime // 超出核心线程数的空闲线程的最大存活时间

TimeUnit unit //空闲线程存活时间单位

BlockingQueue<Runnable> workQueue // 阻塞任务队列，存储待执行的任务

ThreadFactory threadFactory // 线程工厂，用于创建线程，可指定线程命名规则

RejectedExecutionHandler handler // 饱和策略(拒绝策略)，当线程池阻塞队列已满时对新任务的处理

​ ThreadPoolExecutor.AbortPolicy; //丢弃任务并抛出RejectedExecutionException异常。

​ ThreadPoolExecutor.DiscardPolicy; //也是丢弃任务，但是不抛出异常。

​ ThreadPoolExecutor.DiscardOldestPolicy;//丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

​ ThreadPoolExecutor.CallerRunsPolicy; //由调用线程处理该任务

**常见的线程池有哪几种**

​ **1. newSingleThreadExecutor**

​ 创建一个单线程的线程池。这个线程池仅仅有一个线程在工作，也就是相当于单线程串行运行全部任务。假设这个唯一的线程由于异常结束，那么会有一个新的线程来替代它。此线程池保证全部任务的运行顺序依照任务的提交顺序运行。

​ **2. newFixedThreadPool**

​ 创建固定大小的线程池。每次提交一个任务就创建一个线程。直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变。假设某个线程由于运行异常而结束，那么线程池会补充一个新线程。

​ **3. newCachedThreadPool**

​ 创建一个可缓存的线程池。假设线程池的大小超过了处理任务所须要的线程.那么就会回收部分空暇（60秒不运行任务）的线程，当任务数添加时。此线程池又可以智能的加入新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小全然依赖于操作系统（或者说JVM）可以创建的最大线程大小。

​ **4. newScheduledThreadPool**

​ 创建一个大小无限的线程池。此线程池支持定时以及周期性运行任务的需求。

**如何自定义一个线程池**

​ 首先是我们要知道线程池的核心是什么其实就是从ThreadPoolExecutor中创建来的，那么我们来看一下ThreadPoolExecutor的使用，分别介绍了他的7个参数以及用法，首先

5 是核心线程数

10 是最大线程数（最大线程数中包含核心线程数）

3L 表示超时时间3，单位不确定

TimeUnit.MICROSECONDS 表示超时时间单位

​ TimeUnit.DAYS //天 TimeUnit.HOURS //小时 TimeUnit.MINUTES //分钟 TimeUnit.SECONDS //秒 TimeUnit.MILLISECONDS //毫秒 TimeUnit.NANOSECONDS //毫微秒

​ TimeUnit.MICROSECONDS //微秒

Executors.defaultThreadFactory() 使用Executors默认的线程工厂

new LinkedBlockingQueue<>(30) 阻塞队列长度为30，一个项目的最大并发数就是最大线程数+阻塞队列数

​ 常用的三个阻塞队列（一共）

​ SynchronousQueue 单个阻塞队列

​ ArrayBlockingQueue 由数组构建的有界的阻塞队列

​ LinkedBlockingQueue 由链表构建的无界的阻塞队列（默认为Integer的max值也就是21亿，不推荐直接使用最大值）

new ThreadPoolExecutor.CallerRunsPolicy() 拒绝策略

​ ThreadPoolExecutor.AbortPolicy; 如果超过了阻塞队列+最大线程池的数量的并发抛出异常

​ ThreadPoolExecutor.DiscardPolicy; 如果超过了阻塞队列+最大线程池的数量的并发直接丢弃

​ ThreadPoolExecutor.DiscardOldestPolicy如果超过了阻塞队列+最大线程池的数量的并发丢弃最前面的尝试执行下一个线程，也就是超过的有可能被执行一次

​ ThreadPoolExecutor.CallerRunsPolicy 如果超过了阻塞队列+最大线程池的数量的并发如果自己处理不了那么交给调用的线程执行，也就是main线程调用线程池，但是超过了，那么线程池处理不了交给main线程进行执行

      Executor executor = new ThreadPoolExecutor(  
               5,  
               10,  
               3L,  
               TimeUnit.MICROSECONDS,  
               new LinkedBlockingQueue<>(30),  
               Executors.defaultThreadFactory(),  
               new ThreadPoolExecutor.CallerRunsPolicy());

​ 这个就是自定线程池的过程，那么我们来介绍如何合理的设置线程池参数

**合理线程池参数定制**

​ CPU密集型

​ 由于要保证线程的计算速度也就是高利用CPU的性能

​ 计算模式

​ （注：单核CPU告辞）

​ CPU核数+1

​ IO密集型

​ 分两种

​ CPU \* 2 (推荐第二种)

​ CPU / 1 - 阻塞系数 阻塞系数为0.8-0.9

**什么是JMM**

​ JMM就是Java的内存模型

**volitile关键字的作用，原理**

​ 他是一个轻量级的同步机制，保证内存可见性（Memory Visibility）常用于保持内存可见性和防止指令重排序。 所有线程都能看到共享内存的最新状态，但是他不保证原子性。

public class MutableInteger {  
   private int value;  
   public int get(){  
       return value;  
   }  
   public void set(int value){  
       this.value = value;  
   }  
}  
​  
MutableInteger不是线程安全的，因为get和set方法都是在没有同步的情况下进行的。如果线程1调用了set方法，那么正在调用的get的线程2可能会看到更新后的value值，也可能看不到。  
​

Java通过几种原子操作完成工作内存和主内存的交互

1. lock：作用于主内存，把变量标识为线程独占状态。
2. unlock：作用于主内存，解除独占状态。
3. read：作用主内存，把一个变量的值从主内存传输到线程的工作内存。
4. load：作用于工作内存，把read操作传过来的变量值放入工作内存的变量副本中。
5. use：作用工作内存，把工作内存当中的一个变量值传给执行引擎。
6. assign：作用工作内存，把一个从执行引擎接收到的值赋值给工作内存的变量。
7. store：作用于工作内存的变量，把工作内存的一个变量的值传送到主内存中。
8. write：作用于主内存的变量，把store操作传来的变量的值放入主内存的变量中。

**volatile如何保持内存可见性**

volatile的特殊规则就是：

* read、load、use动作必须**连续出现**。
* assign、store、write动作必须**连续出现**。

所以，使用volatile变量能够保证:

* 每次读取前必须先从主内存刷新最新的值。
* 每次写入后必须立即同步回主内存当中。

也就是说，**volatile关键字修饰的变量看到的随时是自己的最新值**。线程1中对变量v的最新修改，对线程2是可见的。

volatile解析

<https://www.cnblogs.com/monkeysayhi/p/7654460.html>

**synchronized关键字的用法，优缺点**

​ 可以用在代码块上，还有方法上面，synchronized 粒度有些大，在处理实际问题时存在诸多局限性。

**Lock接口有哪些实现类，使用场景是什么**

​ Lock有三个实现类

​ ReentrantLock （可重入锁）

​ ReentrantReadWriteLock.ReadLock（读锁）

​ ReentrantReadWriteLock.WriteLock（写锁）

**可重入锁的用处及实现原理**

**读写锁**

**分段锁（ConcurrentHashMap中的segment）**

**线程的状态都有哪些**

​ 创建

​ 准备

​ 运行或者阻塞

​ 死亡

​ 一共有5个状态

**sleep和wait的区别**

​ sleep是睡眠，他指定了一个时间，他会时刻的监视这个时间如果一旦到达就会继续运行，但是wait的睡眠他会让进入到一个类似等待池的状态，在这个状态中他是真正的睡眠，如果不去唤醒他会一直进入到休眠，这个时候就需要唤醒了，我们可以使用notify（在只有两个线程时）他会被其他线程唤醒，但是如果多个线程都在睡眠，这个情况下使用notify就不一定有效了，那么我们可以使用notifyall

**notify和notifyall的区别**

​ notify是唤醒一个线程，这个线程并不能我们指定的去唤醒，但是他会让正在休眠的线程中的某一个线程唤醒，而notifyall则是让所有的睡眠的线程进行唤醒