# 基础

## 三目运算符中的自动拆箱问题

对于条件表达式b?x:y，一个条件表达式从不会既计算x，又计算y。条件运算符是右结合的，也就是说，从右向左分组计算。例如，a?b:c?d:e将按a?b:（c?d:e）执行。

Map<String, Boolean> map = new HashMap<>();

Boolean b = map != null ? map.get("test") : false;

运行以上的代码会报空指针

这段反编译之后

Map map = new HashMap();

Boolean b = Boolean.valueOf(map == null ? false : ((Boolean)map.get("test")).booleanValue());

这是一种自动拆箱的操作。

这段代码为什么会自动拆箱呢？这其实是三目运算符的语法规范。

简单的来说就是：当第二，第三位操作数分别为基本类型和对象时，其中的对象就会拆箱为基本类型进行操作。

结果就是：由于使用了三目运算符，并且第二、第三位操作数分别是基本类型和对象。所以对对象进行拆箱操作，由于该对象为null，所以在拆箱过程中调用null.booleanValue()的时候就报了NPE（空指针）。

如果代码这么写，就不会报错：

Map<String,Boolean> map = new HashMap<String, Boolean>();

Boolean b = (map!=null ? map.get("test") : Boolean.FALSE);

就是保证了三目运算符的第二第三位操作数都为对象类型。这样就不会发生自动拆箱操作

# 集合

## HashMap

**为什么Map桶中个数超过8才转为红黑树**

为什么要转换:因为Map中桶的元素初始化是链表保存的，其查找性能是O(n)，而树结构能将查找性能提升到O(log(n))。当链表长度很小的时候，即使遍历，速度也非常快，但是当链表长度不断变长，肯定会对查询性能有一定的影响，所以才需要转成树。

链表长度达到8就转成红黑树，当长度降到6就转成普通bin。为什么长度为8的时候转成红黑树？说白了就是trade-off，空间和时间的权衡：当hashCode离散性很好的时候，树型bin用到的概率非常小，因为数据均匀分布在每个bin中，几乎不会有bin中链表长度会达到阈值。但是在随机hashCode下，离散性可能会变差，然而JDK又不能阻止用户实现这种不好的hash算法，因此就可能导致不均匀的数据分布。不过理想情况下随机hashCode算法下所有bin中节点的分布频率会遵循泊松分布，我们可以看到，一个bin中链表长度达到8个元素的概率为0.00000006，几乎是不可能事件。所以，之所以选择8，不是拍拍屁股决定的，而是根据概率统计决定的。

# 加密算法

加密算法可以归结为三大类：哈希算法、对称加密算法、非对称加密算法。

哈希算法：如MD5算法，SHA系列算法

对称加密算法：DES算法，3DES算法，AES算法

非对称加密算法：RSA算法

# 线程和线程池

## 什么是线程

线程（英语：thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

一个进程至少得有一个线程，单线程只能干一件事情，无法并发和并行。进程之间是隔离的，进程是拥有资源的基本单位， 线程是CPU调度的基本单位，

JVM是一个进程，所以只能多线程编程。

## Java中线程具有五种状态：

初始化；可运行；运行中；阻塞；销毁



但是，线程不同状态之间的转化是谁来实现的呢？是JVM吗？

并不是。JVM需要通过操作系统内核中的TCB（Thread Control Block）模块来改变线程的状态，这一过程需要耗费一定的CPU资源。

## 什么是协程

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程

最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

由于Java的原生语法中并没有实现协程（某些开源框架实现了协程，但是很少被使用）

## 线程池

单个线程: Executors.newSingleThreadExecutor();

缓存线程: Executors.newCachedThreadPool();

固定线程Executors.newFixedThreadPool(2);

定时线程: Executors.newScheduledThreadPool(3);

## ThreadPoolExecutor

ThreadPoolExecutor 继承 AbstractExecutorService；AbstractExecutorService 实现 ExecutorService， ExecutorService 继承 Executor

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {...}

corePoolSize : 核心线程数的大小

maximumPoolSize : 线程池中允许的最大线程数

keepAliveTime : 空闲线程允许的最大的存活时间

unit : 存活时间的单位

workQueue : 阻塞任务队列

threadFactory : 线程工厂用来创建线程

handler : 拒绝策略，针对当队列满了时新来任务的处理方式

**线程池的执行过程：**

在刚刚创建线程池的时候，内部线程的数量是 0，当首个任务进行添加的时候，会根据参数的配置进行线程的创建，并随着任务数的增加，会逐渐创建新的线程直到线程数达到核心线程的大小。这时，如果再添加任务会将任务放置到阻塞队列当中,如果队列装不下了这时会判断当前线程的数量是否达到了最大线程数，如果未达到，新添加的任务会由最大重新创建线程并立马执行。如果队列满了，而且已经达到最大线程数了，这时再添加任何就会由拒绝策略来处理，默认的拒绝策略是抛出异常。

## 为什么要使用线程池

虽然线程是个轻量级的东西， 但是对于互联网应用来说，如果每个用户的请求都创建一个线程，那会非常得多，服务器也是难于承受， 再说了，众多的线程去竞争CPU，不断切换，也会让CPU调度不堪重负，很多线程将不得不等待。

线程可以预先创建，当线程池的线程刚创建时，让他们进入阻塞状态：等待某个任务的到来。 如果任务来了，那就好办，唤醒其中一个线程，让它拿到任务去执行即可。

## 线程池原理

原文：<https://mp.weixin.qq.com/s/LExpStGw7XXp0M1E_T0yKA>

https://mp.weixin.qq.com/s/DJPuYVhRGRXkkUGMuDdrSw

根据源码可以发现整个线程池大致分为 3 个部分，1. 是创建 worker 线程，2. 添加任务到 workQueue; 3.worker 线程执行具体任务

线程池每个线程会包装成worker，用来循环执行take任务。

第一步是生产者，也就是任务提供者他执行了一个 execute() 方法，本质上就是往这个内部队列里放了一个任务。

之前已经创建好了的 Worker 线程会执行一个 while 循环 ---> 不停的从这个 内部队列里获取任务。(这一步是竞争的关系，都会抢着从队列里获取任务，由这个队列内部实现了线程安全。)

获取得到一个任务后，其实也就是拿到了一个 Runnable 对象(也就是 execute(Runnabletask) 这里所提交的任务)，接着执行这个 Runnable 的 run() 方法，而不是 start()，为什么线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？线程池这个场景中却恰好就是要利用它只是一个普通方法调用。如果我们往一个核心、最大线程数为 2 的线程池里丢了 1000 个任务，那么它会额外的创建 1000 个线程，同时每个任务都是异步执行的，一下子就执行完毕了。

所谓线程池本质是一个hashSet。多余的任务会放在阻塞队列中。

只有当阻塞队列满了后，才会触发非核心线程的创建。所以非核心线程只是临时过来打杂的。直到空闲了，然后自己关闭了。

线程池提供了两个钩子（beforeExecute，afterExecute）给我们，我们继承线程池，在执行任务前后做一些事情。

线程池原理关键技术：锁（lock,cas）、阻塞队列、hashSet（资源池）

当线程池在执行一个任务时抛出未捕获异常，这个任务还会执行吗？



右图放大后的那一块，也就是内部队列并没有其他线程往里边丢任务执行 execute() 方法。

而一旦发生未捕获的异常后， Worker1 被回收，顺带的它所调度的线程 task1（这个task1 也就是在执行一个 while 循环消费左图中的那个队列） 也会被回收掉。

新创建的 Worker2 会取代 Worker1 继续执行 while 循环从内部队列里获取任务，但此时这个队列就一直会是空的，所以也就是处于 Waiting 状态。

原文：<https://mp.weixin.qq.com/s/SpxYhfExN-GwTYflkhvKRg>

## executor与submit的区别

1、接收的参数不一样

2、submit有返回值，而execute没有

3、submit方便Exception处理：意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

# 多线程

## 一个线程OOM，进程里其他线程还能运行么？

答案是还能运行

在多线程环境下，每个线程拥有一个栈和一个程序计数器。栈和程序计数器用来保存线程的执行历史和线程的执行状态，是线程私有的资源。其他的资源（比如堆、地址空间、全局变量）是由同一个进程内的多个线程共享。

当一个线程抛出OOM异常后，即使是堆内存溢出，它所占据的内存资源会全部被释放掉，从而不会影响其他线程的运行！如果是栈溢出，结论也是一样。

JvisualVM看堆的变化

## 多线程有什么用？

（1）发挥多核CPU的优势

（2）防止阻塞

（3）便于建模：假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

## start()方法和run()方法的区别？

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

## Runnable接口和Callable接口的区别？

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

## CyclicBarrier和CountDownLatch的区别？

两个看上去有点像的类，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

（1）CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

（2）CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

（3）CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

## volatile/'vɒlətaɪl/关键字的作用？

（1）多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据

（2）代码底层执行不像我们看到的高级语言----Java程序这么简单，它的执行是Java代码-->字节码-->根据字节码执行对应的C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

## 什么是线程安全

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

（1）不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

（2）绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

（3）相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。

（4）线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

## Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

（1）获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java

（2）打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈，

## 一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

## 如何在两个线程之间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的。

## sleep方法和wait方法有什么区别

这个问题常问，sleep方法和wait方法都可以用来放弃CPU一定的时间，不同点在于如果线程持有某个对象的监视器，sleep方法不会放弃这个对象的监视器，wait方法会放弃这个对象的监视器

## 生产者消费者模型的作用是什么

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

## ThreadLocal有什么用

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

## 为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别？

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

## 为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

## 怎么检测一个线程是否持有对象监视器

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着"某条线程"指的是当前线程。

## synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。

ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

## ConcurrentHashMap的并发度是什么

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

## FutureTask是什么

FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

## Linux环境下如何查找哪个线程使用CPU最长

（1）获取项目的pid，jps或者ps -ef | grep java，这个前面有讲过

（2）top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

## 怎么唤醒一个阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

## 不可变对象对多线程有什么帮助

前面有提到过的一个问题，不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

## 什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

## 如果你提交任务时，线程池队列已满，这时会发生什么

如果使用的是无界队列LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务

如果使用的是有界队列比如ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，会根据maximumPoolSize的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue继续满，那么则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy

## Java中用到的线程调度算法是什么

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

## Thread.sleep(0)的作用是什么

为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

## 什么是自旋

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是

## 什么是Java内存模型

Java内存模型定义了一种多线程访问Java内存的规范。

（1）Java内存模型将内存分为了主内存和工作内存。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于操作主内存和工作内存中的变量

（3）定义了volatile变量的使用规则

（4）happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的

## 什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

## 什么是乐观锁和悲观锁

（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

（2）悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

## 什么是AQS

简单说一下AQS，AQS全称为AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

## 单例模式的线程安全性

首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

## Semaphore有什么作用

Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

## Hashtable的size()方法中明明只有一条语句"return count"，为什么还要做同步

（1）同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性

（2）CPU执行代码，执行的不是Java代码，这点很关键，一定得记住。Java代码最终是被翻译成机器码执行的，机器码才是真正可以和硬件电路交互的代码。即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句"return count"假设被翻译成了三句汇编语句执行，一句汇编语句和其机器码做对应，完全可能执行完第一句，线程就切换了。

## 线程类的构造方法、静态块是被哪个线程调用的

线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：

（1）Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的

（2）Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的

## 同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越小越好。

虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：

a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考其他有关线程池的文章。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

## java程序，主进程需要等待多个子进程结束之后再执行后续的代码，有哪些方案可以实现

**1.join方法**

比如在线程B中调用了线程A的join()方法，直到线程A执行完毕后，才会继续执行线程B。

public static void main(String[] args) throws InterruptedException {

Vector<Thread> vector = new Vector<>();

for(int i=0;i<5;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

}

});

vector.add(childThread);

childThread.start();

}

for(Thread thread : vector) {

thread.join();

}

System.out.println("主线程被执行");

}

**2.等待多线程完成的CountDownLatch**

public static void main(String[] args) throws InterruptedException {

final CountDownLatch latch = new CountDownLatch(5);

for(int i=0;i<5;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

latch.countDown();

}

});

childThread.start();

}

latch.await();//阻塞当前线程直到latch中的值

System.out.println("主线程被执行");

}

**3.同步屏障CyclicBarrier**

public static void main(String[] args) throws InterruptedException, BrokenBarrierException {

final CyclicBarrier barrier = new CyclicBarrier(5);

for(int i=0;i<4;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

try {

barrier.await();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (BrokenBarrierException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

});

childThread.start();

}

barrier.await();//阻塞当前线程直到latch中的值

System.out.println("主线程被执行");

}

CylicBarrier是控制一组线程的同步，初始化的参数：5的含义是包括主线程在内有5个线程，所以只能有四个子线程，这与CountDownLatch是不一样的。

countDownLatch和cyclicBarrier有什么区别呢，他们的区别：countDownLatch只能使用一次，而CyclicBarrier方法可以使用reset()方法重置，所以CyclicBarrier方法可以能处理更为复杂的业务场景。cyclicBarrier只会在最后一个线程结束时才会给主线程发送信号，而countDownLatch会在每个线程结束时都发送一个信号

5.FutureTast可用于闭锁，类似于CountDownLatch的作用

public class Test5 {

public static void main(String[] args) {

MyThread td = new MyThread();

//1.执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。

FutureTask<Integer> result1 = new FutureTask<>(td);

new Thread(result1).start();

FutureTask<Integer> result2 = new FutureTask<>(td);

new Thread(result2).start();

FutureTask<Integer> result3 = new FutureTask<>(td);

new Thread(result3).start();

Integer sum;

try {

sum = result1.get();

sum = result2.get();

sum = result3.get();

//这里获取三个sum值只是为了同步，并没有实际意义

System.out.println(sum);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ExecutionException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} //FutureTask 可用于 闭锁 类似于CountDownLatch的作用，在所有的线程没有执行完成之后这里是不会执行的

System.out.println("主线程被执行");

}

}

class MyThread implements Callable<Integer> {

@Override

public Integer call() throws Exception {

int sum = 0;

Thread.sleep(1000);

for (int i = 0; i <= 10; i++) {

sum += i;

}

System.out.println("子线程被执行");

return sum;

}

}

**6.使用callable+future**

Callable+Future最终也是以Callable+FutureTask的形式实现的。 在这种方式中调用了： Future future = executor.submit(task);

public class Test6 {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executor = Executors.newCachedThreadPool();

Task task = new Task();

Future<Integer> future1 = executor.submit(task);

Future<Integer> future2 = executor.submit(task);

//获取线程执行结果，用来同步

Integer result1 = future1.get();

Integer result2 = future2.get();

System.out.println("主线程执行");

executor.shutdown();

}

}

class Task implements Callable<Integer>{

@Override public Integer call() throws Exception {

int sum = 0;

//do something;

System.out.println("子线程被执行");

return sum;

}

}

1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。CountDownLatch类实际上是使用计数器的方式去控制的，不难想象当我们初始化CountDownLatch的时候传入了一个int变量这个时候在类的内部初始化一个int的变量，每当我们调用countDownt()方法的时候就使得这个变量的值减1，而对于await()方法则去判断这个int的变量的值是否为0，是则表示所有的操作都已经完成，否则继续等待。

实际上如果了解AQS的话应该很容易想到可以使用AQS的共享式获取同步状态的方式来完成这个功能。而CountDownLatch实际上也就是这么做的。

# ClassLoader

## ClassLoader 做什么的？

顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

## 延迟加载

JVM 运行并不是一次性加载所需要的全部类的，它是按需加载，也就是延迟加载。程序在运行的过程中会逐渐遇到很多不认识的新类，这时候就会调用 ClassLoader 来加载这些类。加载完成后就会将 Class 对象存在 ClassLoader 里面，下次就不需要重新加载了。

比如你在调用某个类的静态方法时，首先这个类肯定是需要被加载的，但是并不会触及这个类的实例字段，那么实例字段的类别 Class 就可以暂时不必去加载，但是它可能会加载静态字段相关的类别，因为静态方法会访问静态字段。而实例字段的类别需要等到你实例化对象的时候才可能会加载。

## 各司其职

JVM 运行实例中会存在多个 ClassLoader，不同的 ClassLoader 会从不同的地方加载字节码文件。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA\_HOME/lib/rt.jar 文件中，比如 java.util.\*、java.io.\*、java.nio.\*、java.lang.\* 等等。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA\_HOME/lib/ext/\*.jar 中。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。

那些位于网络上静态文件服务器提供的 jar 包和 class文件，jdk 内置了一个 URLClassLoader，ExtensionClassLoader 和 AppClassLoader 都是 URLClassLoader 的子类。

## ClassLoader 传递性

程序在运行过程中，遇到了一个未知的类，它会选择哪个 ClassLoader 来加载它呢？虚拟机的策略是使用调用者 Class 对象的 ClassLoader 来加载当前未知的类。何为调用者 Class 对象？就是在遇到这个未知的类时，虚拟机肯定正在运行一个方法调用（静态方法或者实例方法），这个方法挂在哪个类上面，那这个类就是调用者 Class 对象。前面我们提到每个 Class 对象里面都有一个 classLoader 属性记录了当前的类是由谁来加载的。

因为 ClassLoader 的传递性，所有延迟加载的类都会由初始调用 main 方法的这个 ClassLoader 全全负责，它就是 AppClassLoader。

## 双亲委派

前面我们提到 AppClassLoader 只负责加载 Classpath 下面的类库，如果遇到没有加载的系统类库怎么办，AppClassLoader 必须将系统类库的加载工作交给 BootstrapClassLoader 和 ExtensionClassLoader 来做，这就是我们常说的「双亲委派」。

## Class.forName

当我们在使用 jdbc 驱动时，经常会使用 Class.forName 方法来动态加载驱动类。

其原理是 mysql 驱动的 Driver 类里有一个静态代码块，它会在 Driver 类被加载的时候执行。这个静态代码块会将 mysql 驱动实例注册到全局的 jdbc 驱动管理器里。

class Driver {

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

}

}

...

}

forName 方法同样也是使用调用者 Class 对象的 ClassLoader 来加载目标类。不过 forName 还提供了多参数版本，可以指定使用哪个 ClassLoader 来加载

通过这种形式的 forName 方法可以突破内置加载器的限制，通过使用自定类加载器允许我们自由加载其它任意来源的类库。根据 ClassLoader 的传递性，目标类库传递引用到的其它类库也将会使用自定义加载器加载。

## 自定义加载器

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

## Class.forName vs ClassLoader.loadClass

这两个方法都可以用来加载目标类，它们之间有一个小小的区别，那就是 Class.forName() 方法可以获取原生类型的 Class，而 ClassLoader.loadClass() 则会报错。

URLClassLoader v1 = new URLClassLoader(new URL[]{new URL(v2dir)});

Class<?> depv1Class = v1.loadClass("Dep");

Object depv1 = depv1Class.getConstructor().newInstance();

depv1Class.getMethod("print").invoke(depv1);

即使是同样的字节码用不同的 ClassLoader 加载出来的类都不能算同一个类

# 时间处理

**为什么以下代码无法得到美国时间。（在东八区的计算机上）**

System.out.println(Calendar.getInstance(TimeZone.getTimeZone("America/Los\_Angeles")).getTime());

通过查看Date.toString的源码，发现在输出的过程中该方法只会去获取系统的默认时区，只有修改了默认时区才会显示该时区的时间。

但是，通过阅读Calendar的源码，我们可以发现，getInstance方法虽然有一个参数可以传入时区，但是并没有将默认时区设置成传入的时区。

而在Calendar.getInstance.getTime后得到的时间只是一个时间戳，其中未保留任何和时区有关的信息，所以，在输出时，还是显示的是当前系统默认时区的时间。

**什么是冬令时？什么是夏令时？**

夏令时、冬令时的出现，是为了充分利用夏天的日照，所以时钟要往前拨快一小时，冬天再把表往回拨一小时。其中夏令时从3月第二个周日持续到11月第一个周日。

冬令时：北京和洛杉矶时差16小时，北京和纽约时差13小时。 夏令时：北京和洛杉矶时差15小时，北京和纽约时差15小时。

**CET,UTC,GMT,CST几种常见时间的含义和关系？**

CET，欧洲中部时间（英語：Central European Time，CET）是比世界标准时间（UTC）早一个小时的时区名称之一。

UTC，协调世界时，又称世界标准时间或世界协调时间，简称UTC。

GMT，格林尼治标准时间，是指位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义在通过那里的经线。

CST，北京时间，China Standard Time，又名中国标准时间，是中国的标准时间。

CET=UTC/GMT + 1小时、CST=UTC/GMT +8 小时、CST=CET+9

# 排序算法

那么，O(1), O(n), O(logn), O(nlogn)就可以看作既可表示算法复杂度，也可以表示空间复杂度。

大O加上（）的形式，里面其实包裹的是一个函数f(),O（f()）,指明某个算法的耗时/耗空间与数据增长量之间的关系。其中的n代表输入数据的量。

如果ax=N（a>0，且a≠1），那么数x叫做以a为底N的对数，记作x=logaN，读作以a为底N的对数，其中a叫做对数的底数，N叫做真数。





## 桶排序

第一步，就是创建这些桶，确定每一个桶的区间范围：

第二步，遍历原始数列，把元素对号入座放入各个桶中：

第三步，每个桶内部的元素分别排序

第四步，遍历所有的桶，输出所有元素

## 计数排序

计数排序不是基于元素比较，而是利用数组下标来确定元素的正确位置，计数排序适用一定范围的整数排序，在取值范围不是很大的情况下，它的性能甚至快过那些O(nlogn)的排序。

第一步：初始化一个下标数组

第二步：遍历需要排序的数组，在第一个数组中找对应位置，并在对应下标下加1

第三步：按下标数字输出第一个数组