# 基础

## 为什么说使用枚举来实现单例是最好的选择

枚举可解决线程安全问题：枚举在经过javac的编译之后，会被转换成形如public final class T extends Enum的定义。

枚举可避免反序列化破坏单例：在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject等方法。普通的Java类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新new出来的，所以这就破坏了单例。但是，枚举的反序列化并不是通过反射实现的。所以，也就不会发生由于反序列化导致的单例破坏问题。

## http1.0和http1.1的区别

http1.0协议使用非持久连接，即在非持久连接下，一个tcp连接只传输一个web对象,http1.1默认使用持久连接，一个连接可以传输多个对象

http1.1对请求和响应增加了请求头和响应头

http1.1支持断点续传，http1.0不支持

http1.1通过请求头部使得在一台web服务器可以同一个IP，端口，使用不同的主机名创建多个web虚拟站点

http1.1有身份认证机制，http提供特殊的状态码和头部来帮助web站点执行身份认证

## TCP“3次握手，4次挥手”

TCP是一种面向连接的单播协议，在发送数据前，通信双方必须在彼此间建立一条连接。所谓的“连接”，其实是客户端和服务器的内存里保存的一份关于对方的信息，如ip地址、端口号等。

**三次握手**

第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

第二次握手：服务端发送网络包，客户端收到了。这样客户端就能得出结论：服务端的发送能力、客户端的接收能力是正常的。

第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。

**四次挥手**

TCP连接是双向传输的对等的模式，就是说双方都可以同时向对方发送或接收数据。当有一方要关闭连接时，会发送指令告知对方，我要关闭连接了。这时对方会回一个ACK，此时一个方向的连接关闭。但是另一个方向仍然可以继续传输数据，等到发送完了所有的数据后，会发送一个FIN段来关闭此方向上的连接。接收方发送ACK确认关闭连接。注意，接收到FIN报文的一方只能回复一个ACK, 它是无法马上返回对方一个FIN报文段的，因为结束数据传输的“指令”是上层应用层给出的，我只是一个“搬运工”，我无法了解“上层的意志”

其实3次握手的目的并不只是让通信双方都了解到一个连接正在建立，还在于利用数据包的选项来传输特殊的信息，交换初始序列号ISN。

3次握手是指发送了3个报文段，4次挥手是指发送了4个报文段。注意，SYN和FIN段都是会利用重传进行可靠传输的。

## 常用的集合主要在java.util包下面：

主要接口有List,Set,Map,Queue,Deque,SortedMap,SortedSet

List:

线程不安全：ArrayList, LinkedList, AttributeList, RoleList

线程安全：CopyOnWriteArraylist ,Stack, Vector

Set:

线程不安全：EnumSet, HashSet, LinkedHashSet, TreeSet

线程安全：ConcurrentSkipListSet, CopyOnWriteArraySet

Map:

线程不安全：EnumMap, HashMap, IdentityHashMap, LinkedHashMap, Properties, TreeMap

线程安全：ConcurrentHashMap, ConcurrentSkipListMap, Hashtable

SortedMap继承于map:

ConcurrentSkipListMap, TreeMap

SortedSet继承于set:

ConcurrentSkipListSet, TreeSet

queue:

PriorityQueue, （优先级队列）

ConcurrentLinkedQueue, （基于链表的并发队列）

DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）

ArrayBlockingQueue, （基于数组的并发阻塞队列）

LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

PriorityBlockingQueue, （带优先级的无界阻塞队列）

SynchronousQueue （并发同步阻塞队列）

Deque:

LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）

ArrayDeque, （数组双端队列）

## 讲讲自己对HashMap的理解？

HashMap是以key-value键值对存储数据的集合，JDK1.7以前，HashMap底层使用的是数组+单向链表的数据结构（在数据结构中，一般称之为“链表散列“），其主要包含put，get方法，put方法用于存储数据，get用于获取数据。当使用put方法时，首先会根据key的hashcode与数组大小长度-1进行逻辑与运算，也就是通过求模计算出该数据在数组中存储的位置。然后遍历该位置下的链表，使用equals判断这个链表是否包含这个key，如果存在则替换该entry对象的值，并将旧的value返回。如果不存在则将新数据放在链表的首位。存储完成以后会判断数组的大小是否超过阀值，超过则对数组进行扩容，并遍历整个hashmap，将数据放入新的数组中。get方法则相对简单，只是通过hashcode找到数组对应位置，遍历数组，使用equals方法查找是否包含对应的值，有则返回，无则返回null。

JDK1.8之后HashMap数据结构做了改动，在链表数据达到8个时会转换为使用红黑树的数据结构存储，put，get方法也做了相应的改动。put方法存储数据时会先判断当前数据结构是否为红黑树，是则通过红黑树的方式存储数据，而且扩容的时候也有些改变，1.7是遍历整个数组，并将链表的数据一个一个重新计算hashcode到新的数组中，但1.8则是先生成新的两个链表，再放到新的数组中。

HashMap 1.8新增了TREEIFY\_THRESHOLD 用于判断是否需要将链表转换为红黑树的阈值。并且HashEntry 修改为 Node。

## ConcurrentHashMap 是如何实现的？

**1.7、1.8 实现有何不同？为什么这么做？**

base1.7

1.7仍然是数组加链表，相比于HashMap，ConcurrentHashMap数组与entry很多核心数据都使用了volatile修饰.原理上来说：ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理。因为volatile无法保证原子性， put 操作时仍然需要加锁处理。首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用自旋获取锁。如果重试的次数达到了配置（MAX\_SCAN\_RETRIES）则改为阻塞锁获取，保证能获取成功。

由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

base1.8

JDK8中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。1.8抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。也将 1.7 中存放数据的 HashEntry 改为 Node，但作用都是相同的。其中的 val next 都用了 volatile 修饰，保证了可见性。

## JVM内存划分

JDK1.7时JVM内存划分为程序计数器，虚拟机栈，本地方法栈，堆，方法区，JDK1.8将方法区移入到了直接内存中，称为元空间。

**程序计数器**主要有两个作用：在多线程的情况下，程序计数器用于记录当前线程执行的位置。

字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

**虚拟机栈：**用于描述Java方法执行的状态，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java方法有两种返回方式：return 语句，抛出异常

**本地方法栈：**虚拟机栈的作用非常相似，区别是：虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法。

**堆：**在虚拟机启动时创建，唯一目的就是存放对象实例，所有的对象实例以及数组都要在堆上分配。Java堆还可以细分为：新生代和老年代：再细致一点有：eden区、s0区、s1区都属于新生代，tentired 区属于老年代。

**方法区：**是各个线程共享内存区域，用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。HotSpot 虚拟机中方法区也常被称为 “永久代。

**直接内存**

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 异常出现。

## Java内存模型

Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型的规范。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存的变量是主内存中拷贝过来的，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

而JMM就作用于工作内存和主存之间数据同步过程。他规定了如何做数据同步以及什么时候做数据同步。

**Java内存模型的实现**

了解Java多线程的朋友都知道，在Java中提供了一系列和并发处理相关的关键字，比如volatile、synchronized、final、concurren包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字。

## JVM调优

调优工具：Jconsole,jProfile,visualVM

如何调优：

堆信息查看：

可查看堆空间大小分配（年轻代，年老代，持久代）

提供即时的垃圾回收功能

垃圾监控

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

年老代年轻代大小划分是否合理

内存泄漏

垃圾回收算法设置是否合理

查看FullGC的频率

Linux使用jstat命令查看jvm的GC情况，通常运行命令如下：

jstat -gc 12538 5000

即会每5秒一次显示进程号为12538的java进成的GC情况

性能调优建议： jvm调优没有一个固定模板配置说必须如何操作，它需要根据系统的情况不同对待。 但是可以有如下建议：

1、初始化内存和最大内存尽量保持一致，避免内存不够用继续扩充内存。最大内存不要超过物理内存，例如内存8g，你可以设置最大内存4g/6g但是不能超过8g否则加载类的时候没有空间会报错。

2、gc/full gc频率不要太高、每次gc时间不要太长、根据系统应用来定。

## 对象在内存中的初始化过程

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

类的加载指的是将类的.class文件以二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个Class对象，用来封装类在方法区内的数据结构。然后会对文件进行JVM规范校验，代码逻辑校验。校验通过以后，会在堆中为类的静态变量及成员变量分配内存，并将其初始化为默认值，接着执行构造代码块，对对象进行初始化，在堆中为类的对象分配内存。

private static int a = 3;在准备阶段会被a设为默认值0，执行初始化时把3赋予了a。静态代码块从上到下执行

变量覆盖：ParentClass subclass = new SubClass();

子类跟父类有相同普通变量的情况下，subClass.protectedField输出的是父类定义的变量

**类的初始化顺序**

1. 父类--静态变量

2. 父类--静态初始化块

3. 子类--静态变量

4. 子类--静态初始化块

5. 父类--变量

6. 父类--初始化块

7. 父类--构造器

8. 子类--变量

9. 子类--初始化块

10.子类--构造器

## 什么时候一个对象会被GC？

被GC的对象首先会被标记为是否可回收，一般采用有向图的方式记录和管理堆(heap)中的所有对象，从根节点向下检索，不能检索到的则视为这个对象不可用，并标记为可回收。标记后的对象并不会立即回收，对象没有覆盖finalize()方法,或者finalize()方法已经被虚拟机调用过此对象会被直接回收。否则这个对象将会放置在一个队列之中,并在稍后由一个由虚拟机线程去执行其finalize()方法，做一些清除前的工作，例如资源释放等；直至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空间。

任何一个对象的finalize()方法都只会被系统自动调用一次,如果对象面临下一次回收,它的finalize()方法不会被再次执行

**垃圾收集器**

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收

答： 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

新生代的垃圾回收器包括Serial、ParNew、Parallel Scavenge，老年代的垃圾回收器包括CMS、Serial Old、Parallel Old。其中新生代的三种垃圾回收器都采用了复制算法，并且会触发STOP THE WORLD。CMS使用标记清除算法，Serial Old、Parallel Old使用标记整理算法。

**Serial收集器**是一个**单线程收集器**（-XX：+UseSerialGC），这个“单线程”不只是说它只会使用一个CPU或者一条线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它垃圾收集结束。

**ParNew收集器**是Serial**收集器的多线程**版本（-XX:+UseParNewGC -XX:ParallelGCThreads），它能够与CMS收集器配合工作，因此，在运行在Server模式下的虚拟机中，ParNew收集器是首选的新生代收集器。

**Parallel Scavenge**这也是一个并行的新生代垃圾收集器，不同于其他收集器（以尽可能缩短垃圾收集时用户线程的停顿时间为目的），它是唯一一个以达到一个可控制的吞吐量为目标的垃圾收集器。

**CMS收集器**的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。

重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。

应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。

CMS收集器，我们会面临两个主要的挑战，可能需要进行调优：堆碎片;对象分配率高

CMS收集器并没有任何碎片整理的机制。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。

G1算法：

G1把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；

增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时 把之前用掉的部分中的存活对象再放到后面没有用的空间中

回收步骤：

初始标记（Initial Marking） stop the world

并发标记（Concurrent Marking）

最终标记暂停（Final Marking Pause） stop the world

存活对象计算及清除（Live Data Counting and Cleanup）

## 自定义加载器

自定义类加载器分为以下几个步骤：

1、定义类。继承ClassLoader

2、重写ClassLoader的 findClass方法

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」。

## jvm中有以下三种锁(由上到下越来越“重量级”)：偏向锁,轻量级锁,重量级锁

**偏向锁**：适用于只有一个线程访问同步块场景。加锁和解锁不需要额外的消耗。

**轻量级锁**： 可以有多个线程交替进入临界区，在竞争不激烈的时候，稍微自旋等待一下就能获得锁。这也是CAS的实现方式。竞争的线程不会阻塞，提高了程序的响应速度。但是会消耗CPU

**重量级锁**：比如Synchronized它会利用java对象的监视器锁（monitor）实现，同一临界区内只允许一个线程操作。线程竞争不使用自旋，不会消耗CPU。

## CAS原理及缺点

CAS是通过unsafe类的compareAndSwap方法实现的， 它将内存位置的内容与给定值进行比较，只有在相同的情况下，将该内存位置的内容修改为新的给定值。

compareAndSwap方法的第一个参数是要修改的对象，第二个参数是对象中要修改变量的偏移量，第三个参数是修改之前的值，第四个参数是预想修改后的值。通过将偏移量与修改之前的值做比较，则将将偏移量修改为预想修改的值，并返回true.

当然CAS一定要volatile变量配合

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

compareAndSet这个方法主要调用unsafe.compareAndSwapInt这个方法，这个方法有四个参数，其中第一个参数为需要改变的对象，第二个为偏移量(即之前求出来的valueOffset的值)，第三个参数为期待的值，第四个为更新后的值。整个方法的作用即为若调用该方法时，value的值与expect这个值相等，那么则将value修改为update这个值，并返回一个true，如果调用该方法时，value的值与expect这个值不相等，那么不做任何操作，并范围一个false。

**CAS的缺点：**

1.CPU开销较大：在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

3.ABA问题：这是CAS机制最大的问题所在。

如线程1从内存X中取出A，这时候另一个线程2也从内存X中取出A，并且线程2进行了一些操作将内存X中的值变成了B，然后线程2又将内存X中的数据变成A，这时候线程1进行CAS操作发现内存X中仍然是A，然后线程1操作成功。虽然线程1的CAS操作成功，但是整个过程就是有问题的。比如链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。

所以JAVA中提供了AtomicStampedReference/AtomicMarkableReference来处理会发生ABA问题的场景，主要是在对象中额外再增加一个标记来标识对象是否有过变更。

## 死锁产生的四个条件：

1）互斥。共享资源同时只能被一个线程访问。

2）占有且等待。线程T1在取得共享资源A的时候，请求等待资源B的时候并不释放资源A。

3）不可抢占。其他线程不能强行抢占线程的资源。

4）循环等待条件。线程T1在持有资源A1，同时在请求等待获取资源B，线程T2在持有资源B,然后在请求等待线程T1的持有资源，形成了交叉闭环申请。

处理死锁的方法可有以下4种：

1）死锁预防。由于互斥这个条件无法避免。操作系统本身所具有的特点。可对死锁产生的其他三个条件进行破坏。首先条件2，占有且等待，可以一次性申请所有的资源，可以破坏掉占有且等待。条件三不可抢占。当线程去请求其他资源时，如果获取不到锁，可以主动释放自己的锁，这样不可抢占的条件也被破坏掉了。条件四循环等待条件。可以对申请的资源进行编号，按序访问，这样线性的去申请资源，则不会造成交叉循环。

2）死锁避免。就不用去破坏形成死锁的其他条件。在资源动态分配的过程中。用某种方法判断防止进入不安全状态。从而避免发生死锁。可以使用银行家算法。死锁避免的算法会导致系统开销的增加。

3）死锁检测。死锁预防和死锁检测都是死锁发生之前的预防策略。死锁检测是通过系统设置的检测机构及时的判断当前系统是否处于死锁状态，并精确的确定当前死锁相关的进程和资源，执行死锁解除策略

4）死锁解除。这是与死锁检测结合使用的。它使用的方式是剥夺。就是将进程所占有的资源强行收回，分配给其他进程。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**死锁排查**

1 、 使用top命令查看cpu占用资源较高的PID

2、 通过jps 找到当前用户下的java程序PID：执行 jps -l 能够打印出所有的应用的PID，找到有一个PID和这个cpu使用100%一样的ID！！就知道是哪一个服务了。

3、 使用 pidstat -p 1 3 -u -t

4 、 找到cpu占用较高的线程TID

5、 将TID转换为十六进制的表示方式

6、 通过jstack -l 输出当前进程的线程信息

7、 查找 TID对应的线程(输出的线程id为十六进制)，找到对应的代码

## Lock与synchronized的区别

1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的

2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的条件下提供一种退出的机制。

3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以    及设置等待时限等方式退出条件队列。

4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

## synchronized和ReentrantLock的区别

synchronized是关键字，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。

## 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。核心线程数不会被关闭，会一直等待任务到来。

## 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）

## 线程间有哪些通信方式？

线程间的通信模式有多种,可以是等待通知模式，共享内存模式，管道通信模式

等待通知模式可以使用wait() 和通知 notify() 方法来进行线程间通讯，也可以使用join()方法，在 join 线程完成后会调用 notifyAll() 方法，是在 JVM 实现中调用。。

共享内存模式可以使用volatile共享内存，也可以使用类似于CountDownLatch，CyclicBarrier，Semaphore这样的并发工具进行通信。

管道通信模式可以使用PipedWriter，PipedReader，writer.connect(reader)

也可以使用阻塞队列BlockingQueue来共享数据

# 框架及服务