# Java

## JVM(Java虚拟机)

### 运行时数据区



#### 所有线程共享的数据区（堆空间线程共享）

##### 方法区(Method Area)

存放被虚拟机加载的类信息、常量、静态变量、编译后的代码数据，也称为Permanent Generation。

-XX:MaxPermSize=16m ##设置持久代最大为16m。

##### 堆(Heap)

存放对象实例

-Xmx3500m ##设置堆最大可用内存。

-Xms3500m ##设置堆最小可用内存。

-Xmn2g

##设置年轻代大小（整个堆大小=年轻代大小+年老代大小+持久代大小）。

##持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。

##此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-XX:NewRatio=4

##设置年轻代（包括Eden和两个Survivor区）与年老代的比值（不包括持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5。

-XX:SurvivorRatio=4

##设置年轻代中Eden区与Survivor区的大小比值。

设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6。

-XX:MaxTenuringThreshold=0

##设置垃圾最大年龄。

如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代 。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间 ，增加在年轻代即被回收的概率。

#### 线程隔离的数据区（栈空间线程私有）

生命周期与线程相同。

##### 虚拟机栈(Virtual Machine Stack)

Java方法执行的内存模型。

Stack Frame(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。

每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放编译可知的各种基本数据类型、对象引用类型和returnAddress类型(指向一条字节码指令的地址)。

long和double占用2个局部变量空间(Slot)，其余占用1个。

局部变量表所需内存空间在编译期间完成分配，当进入方法时，这个方法需要在帧中分配多大局部变量空间是完全确定的，方法在运行期间不会改变局部变量表的大小。

如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

-Xss128k

##设置每个线程的堆栈大小。

##根据应用的线程所需内存大小进行调整。

##在相同物理内存下，减小这个值能生成更多的线程。

##但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

##### 本地方法栈(Native Method Stack)

与虚拟机栈类似。

虚拟机栈为虚拟机执行Java方法（字节码）服务。

本地方法栈为虚拟机使用的Native方法服务。

有的虚拟机(例如Sun HotSpot)直接把本地方法栈和虚拟机栈合二为一。

##### 程序计数器(Program Counter Register)

当前线程所执行的字节码的行号指示器，记录正在执行的虚拟机字节码指令的地址，如果正在执行Native方法，这个计数器值为空(Undefined)

唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

各个线程之间计数器互不影响，独立存储。

### 垃圾回收算法

#### 标记-清除算法(Mark-Sweep)

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经介绍过了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

#### 复制算法(Copying)

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

#### 标记-整理算法(Mark-Compact)

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

#### 分代收集算法(Generational Collection)

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

### 垃圾回收器

#### 串行回收器

只适用于小数据量的情况。

#### 并行回收器（吞吐量优先）

并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

XX:+UseParallelGC

##选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20

##配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

-XX:+UseParallelOldGC

##配置年老代垃圾收集方式为并行收集。

-XX:MaxGCPauseMillis=100

##设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

-XX:+UseAdaptiveSizePolicy

##设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

#### 并发回收器(响应时间优先)

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

-XX:+UseConcMarkSweepGC

##CMS(Concurrent Mark-Sweep)

##设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC

##设置年轻代为并行收集。可与CMS收集同时使用。jdk5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

-XX:CMSFullGCsBeforeCompaction=5

##由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection

##打开对年老代的压缩。可能会影响性能，但是可以消除碎片。

### 辅助信息

#### -XX:+PrintGC

##输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs]

                [Full GC 121376K->10414K(130112K), 0.0650971 secs]

#### -XX:+PrintGCDetails

##输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]

                [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

#### -XX:+PrintGCTimeStamps

##可与上面两个混合使用。

##输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

#### -XX:+PrintGCApplicationConcurrentTime

##打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用

##输出形式：Application time: 0.5291524 seconds

#### -XX:+PrintGCApplicationStoppedTime

##打印垃圾回收期间程序暂停的时间。可与上面混合使用

输出形式：Total time for which application threads were stopped: 0.0468229 seconds

#### -XX:PrintHeapAtGC

##打印GC前后的详细堆栈信息

#### -Xloggc:filename

##与上面几个配合使用，把相关日志信息记录到文件以便分析。

### Eclipse Memory Analysis

#### 安装

在Eclipse help -> Eclipse Marketplace下搜索Memory

## 集合

### HashMap的实现原理

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

#### HashMap的数据结构

在Java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。

#### HashMap的存取实现

##### 存储

当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标）， 如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

##### 读取

从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

#### 归纳

简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，  
也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

### 队列(Queue)

## 多线程(Thread)

### 进程与线程的区别

进程：每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销，一个进程包含1--n个线程。（进程是资源分配的最小单位）。

线程：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。（线程是cpu调度的最小单位）。

线程和进程一样分为五个阶段：创建、就绪、运行、阻塞、终止。

多进程是指操作系统能同时运行多个任务（程序）。

多线程是指在同一程序中有多个顺序流在执行。

### Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

总结：

实现Runnable接口比继承Thread类所具有的优势：

适合多个相同的程序代码的线程去处理同一个资源。

可以避免java中的单继承的限制。

增加程序的健壮性，代码可以被多个线程共享，代码和数据独立。

线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类。

### 线程状态转换



**新建状态（New）**：新创建了一个线程对象。

**就绪状态（Runnable）**：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

**运行状态（Running）**：就绪状态的线程获取了CPU，执行程序代码。

**阻塞状态（Blocked）**：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、**等待阻塞**：运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)

（二）、**同步阻塞**：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、**其他阻塞**：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。（注意,sleep是不会释放持有的锁）

**死亡状态（Dead）**：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

### 线程调度

#### 线程优先级

Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

MAX\_PRIORITY

          ##线程可以具有的最高优先级，取值为10。

MIN\_PRIORITY

          ##线程可以具有的最低优先级，取值为1。

NORM\_PRIORITY

          ##分配给线程的默认优先级，取值为5。

#### Thread.sleep

线程睡眠，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态，**不释放锁**。

#### Object.wait

线程等待，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法。**wait释放锁，必须在对象锁代码块里调用,否则会抛出java.lang.IllegalMonitorStateException**。

#### Object.notify(Object.notifyAll)

Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。**notify和notifyAll()必须在对象锁代码块里调用,否则会抛出java.lang.IllegalMonitorStateException**。

#### Thread.yield

线程让步，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

yield应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

#### Thread.join

线程加入，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

join是Thread类的一个方法，启动线程后直接调用，即join()的作用是：“等待该线程终止”，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子线程结束了才能执行。

#### Thread.**interrupt**

不要以为它是中断某个线程！它只是向线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出异常，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

#### **wait和sleep区别**

共同点：

他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

wait()和sleep()都可以通过interrupt()方法 打断线程的暂停状态 ，从而使线程立刻抛出InterruptedException。

如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep/join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出InterruptedException 。

不同点：

Thread类的方法：sleep(),yield()等

Object的方法：wait()和notify()等

每个对象都有一个锁来控制同步访问。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

sleep，wait必须捕获异常，而notify和notifyAll不需要捕获异常。

所以sleep()和wait()方法的最大区别是：

　　　　sleep()睡眠时，保持对象锁，仍然占有该锁；

　　　　而wait()睡眠时，释放对象锁。

　　但是wait()和sleep()都可以通过interrupt()方法打断线程的暂停状态，从而使线程立刻抛出InterruptedException（但不建议使用该方法）。

sleep()方法

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CUP的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会;

　　 sleep()是Thread类的Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个Synchronized块中调用Sleep()方法是，线程虽然休眠了，但是对象的机锁并木有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。

　　在sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级。

wait（）方法

wait()方法是Object类里的方法；当一个线程执行到wait()方法时，它就进入到一个和该对象相关的等待池中，同时失去（释放）了对象的机锁（暂时失去机锁，wait(long timeout)超时时间到后还需要返还对象锁）；其他线程可以访问；

　　wait()使用notify或者notifyAlll或者指定睡眠时间来唤醒当前等待池中的线程。

　　wiat()必须放在synchronized block中，否则会在program runtime时扔出java.lang.IllegalMonitorStateException异常。

### 线程池

#### ThreadPoolExecutor

##### 构造方法参数说明

1. **corePoolSize**：核心池的大小。

这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

1. **maximumPoolSize**：线程池最大线程数。

这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

1. **keepAliveTime**：表示线程没有任务执行时最多保持多久时间会终止。

默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

1. **unit**：参数keepAliveTime的时间单位。

有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

1. workQueue：一个阻塞队列，用来存储等待执行的任务。

这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

PriorityBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和SynchronousQueue。线程池的排队策略与BlockingQueue有关。

1. threadFactory：线程工厂，主要用来创建线程。
2. handler：表示当拒绝处理任务时的策略。

有以下四种取值：

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。

#### ThreadPoolExecutor继承关系

ThreadPoolExecutor(类)<- AbstractExecutorService(抽象类)<- ExecutorService(接口)<- Executor(接口)

Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

#### 线程池方法说明

##### execute()

execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

##### submit()

submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果。

##### shutdown()

关闭线程池

##### shutdownNow()

马上关闭线程池

##### getQueue()

##### getPoolSize()

##### getActiveCount()

##### getCompletedTaskCount()

#### 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

volatile int runState;

static final int **RUNNING** = 0;

static final int **SHUTDOWN** = 1;

static final int **STOP** = 2;

static final int **TERMINATED** = 3;

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于**RUNNING**状态；

如果调用了shutdown()方法，则线程池处于**SHUTDOWN**状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于**STOP**状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于**SHUTDOWN**或**STOP**状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为**TERMINATED**状态。

## Spring Boot

### Spring Boot四个核心

#### 自动配置

#### 起步依赖

#### 命令行界面(Spring Boot CLI)

#### Actuator

Actuator提供运行时检视应用程序内部情况的能力，包括以下细节:

Spring应用程序上下文里配置的Bean。

Spring Boot自动配置做的决策。

应用程序取到的环境变量、系统属性、配置属性和命令行参数。

应用程序里线程的当前状态。

应用程序最近处理过的HTTP请求的追踪情况。

各种和内存用量、垃圾回收、Web请求以及数据源用量相关的指标。

Actuator通过Web端点和Shell界面向外界提供信息。

## 设计模式

## Maven

## Gradle