# 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)

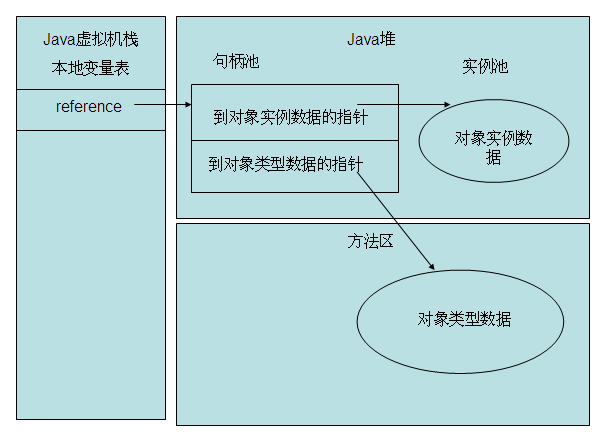
唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

各个线程之间计数器互不影响，独立存储。

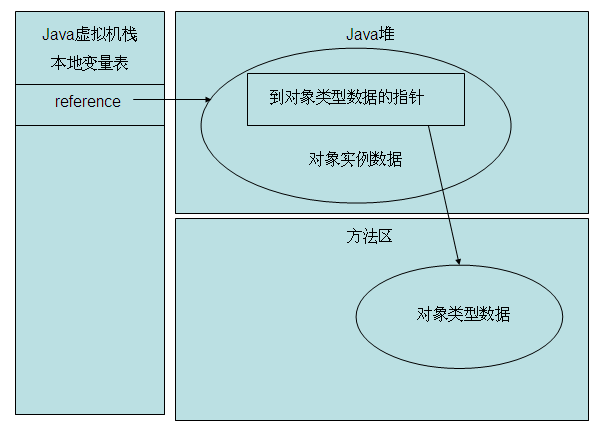
### 对象访问的两种方式

主流的两种访问方式:使用句柄和直接指针。(HotSpot虚拟机就是使用直接指针的访问方式)

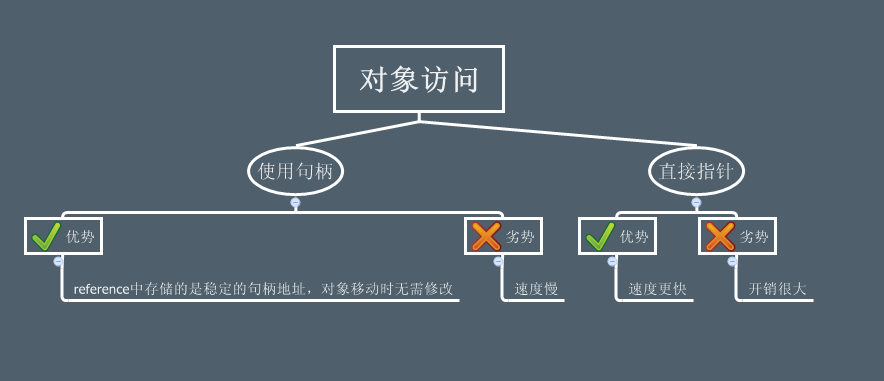
#### 使用句柄访问



#### 使用直接指针访问



优缺点比较



### 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，年轻代使用行发收集，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。

### 内存分析工具

#### jps

jps是JDK提供的一个小工具，上面的命令会把操作系统里面的java应用都展示出来，显示PID，启动类或者JAR，VM参数。可以通过jps -help进一步了解详细信息。当然也可以用操作系统的netsat查询PID。

##### jps -vl

#### jmap

##### jmap -heap pid

jmap查看内存使用情况与生成heapdump

jmap -heap 1234  (1234为进程号)

-heap：打印heap空间的概要，这里可以粗略的检验heap空间的使用情况。

例：

jmap -heap 12345

输出：

Attaching to process ID 2657, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 1.5.0\_16-b02

using thread-local object allocation.

Mark Sweep Compact GC

Heap Configuration:

MinHeapFreeRatio = 40

MaxHeapFreeRatio = 70

MaxHeapSize = 67108864 (64.0MB)

NewSize = 655360 (0.625MB)

MaxNewSize = 4294901760 (4095.9375MB)

OldSize = 1441792 (1.375MB)

NewRatio = 12

SurvivorRatio = 8

PermSize = 8388608 (8.0MB)

MaxPermSize = 67108864 (64.0MB)

Heap Usage:

New Generation (Eden + 1 Survivor Space):----------------------------------------新生代区

capacity = 4521984 (4.3125MB)

used = 1510200 (1.4402389526367188MB)

free = 3011784 (2.8722610473632812MB)

33.39684527853261% used

Eden Space:--------------------------------------------------------------------伊甸园区

capacity = 4063232 (3.875MB)

used = 1495992 (1.4266891479492188MB)

free = 2567240 (2.4483108520507812MB)

36.81778446320565% used

From Space:-------------------------------------------------------------------年轻代（幸存者乐园1）

capacity = 458752 (0.4375MB)

used = 14208 (0.0135498046875MB)

free = 444544 (0.4239501953125MB)

3.0970982142857144% used

To Space:-----------------------------------------------------------------------------年轻代（幸存者乐园2）

capacity = 458752 (0.4375MB)

used = 0 (0.0MB)

free = 458752 (0.4375MB)

0.0% used

concurrent mark-sweep generation:-------------------------------------------------老年代

capacity = 8589934592 (8192.0MB)

used = 0 (0.0MB)

free = 8589934592 (8192.0MB)

0.0% used

Perm Generation:----------------------------------------------------------------------永久代

capacity = 11796480 (11.25MB)

used = 11712040 (11.169471740722656MB)

free = 84440 (0.08052825927734375MB)

99.28419325086806% used

以上的输出很简单，第四行起开始输出此进程我们的JAVA使用的环境。

Heap Configuration，指在我们启动时设置的一些JVM参数。像最大使用内存大小，年老代，年青代，持久代大小等。有这个可以很简单的查看本进程的内存使用情况。也许进程占用的总内存比较多，但我们在这里可以看到真正用到的并没有多少，很多都是"Free"。内存使用的堆积大多在老年代，内存池露始于此，所以要格外关心“tenured generation”。

##### jmap -dump:live,format=b,file=fileName pid

产生一个HeapDump文件

##### jmap -histo [pid]

查看JVM堆中对象详细占用情况

#### jinfo

##### jinfo -flags pid

显示JVM的参数，包括显示设置的和系统默认的。比如所用的垃圾回收器，堆的最大值等。

##### jinfo -sysprops pid

显示System.getProperties()的内容

#### jstat

##### jstat -gc pid

显示JVM的各个内存区使用情况（容量和使用量），GC的次数和耗时。

##### jstat -class pid

查看class的加载情况。

#### jstack [pid]

线程的所有堆栈信息，包括死锁进程

线程状态的分析

死锁，Deadlock（重点关注）

执行中，Runnable

等待资源，Waiting on condition（重点关注）

          如果发现有大量的线程都在处在 Wait on condition，从线程 stack看， 正等待网络读写，这可能是一个网络瓶颈的征兆。因为网络阻塞导致线程无法执行。一种情况是网络非常忙，几 乎消耗了所有的带宽，仍然有大量数据等待网络读 写；另一种情况也可能是网络空闲，但由于路由等问题，导致包无法正常的到达。所以要结合系统的一些性能观察工具来综合分析，比如 netstat统计单位时间的发送包的数目，如果很明显超过了所在网络带宽的限制 ; 观察 cpu的利用率，如果系统态的 CPU时间，相对于用户态的 CPU时间比例较高；

等待获取监视器，Waiting on monitor entry（重点关注）

暂停，Suspended

对象等待中，Object.wait() 或 TIMED\_WAITING

阻塞，Blocked（重点关注）

停止，Parked

TIMED\_WAITING (parking)：这里指定了时间，到达指定的时间后自动退出等待状态；parking指线程处于挂起中

#### jstack pid

查看线程运行情况，检测是否有死锁。

#### jconsole

#### jvisualvm

#### 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。

### 常见线程名词解释

#### 主线程

JVM调用程序main()所产生的线程。

#### 当前线程

这个是容易混淆的概念。一般指通过Thread.currentThread()来获取的进程。

#### 后台线程（守护线程）

指为其他线程提供服务的线程，也称为守护线程。JVM的垃圾回收线程就是一个后台线程。**用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束**

#### 前台线程

是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过isDaemon()和setDaemon()方法来判断和设置一个线程是否为后台线程。

### 线程池

#### ThreadPoolExecutor

##### 构造方法参数说明

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；

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；

**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；

**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

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。

threadFactory：线程工厂，主要用来创建线程。

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

### 最简单的SpringBoot Web工程配置

#### pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.6.RELEASE</version>

<relativePath />

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!-- 上边引入 parent，因此 下边无需指定版本 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

#### Controller

@RestController

public class TestController {

@GetMapping("/helloworld")

public String helloworld() {

return "helloworld";

}

}

#### SpringbootApplication

@SpringBootApplication

public class SpringbootApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootApplication.class, args);

}

}

### 配置详解

#### application.properties

##### server.port=8888

指定端口

##### spring.application.name=hello

指定应用名(SpringCloud中会被注册为服务名)

#### application.yml

environments:

dev:

url: http://dev.bar.com

name: Developer Setup

prod:

url: http://foo.bar.com

name: My Cool App

server:

port:8881

---

spring:

profiles:test

server:

port:8882

---

spring:

profiles:prod

server:

port:8883

#### 自定义参数

除了可以在Spring Boot的配置文件中设置各个Starter模块中预定义的配置属性，也可以在配置文件中定义一些我们需要的自定义属性。比如在application.properties中添加：

book.name=SpringCloudInAction

book.author=ZhaiYongchao

然后，在应用中可以通过@Value注解来加载这些自定义的参数，比如：

@Component

public class Book {

@Value("${book.name}")

private String name;

@Value("${book.author}")

private String author;

// 省略getter和setter

}

@Value注解加载属性值的时候可以支持两种表达式来进行配置，如下所示。

·一种是上面介绍的PlaceHolder方式，格式为${...}，大括号内为PlaceHolder。

·另一种是使用SpEL表达式（Spring Expression Language），格式为#{...}，大括号内为SpEL表达式。

#### 参数引用

在application.properties中的各个参数之间可以直接通过使用PlaceHolder的方式来进行引用，就像下面的设置：

book.name=SpringCloud

book.author=ZhaiYongchao

book.desc=${book.author} is writing《${book.name}》

book.desc参数引用了上文中定义的book.name和book.author属性，最后该属性的值就是ZhaiYongchao is writing《SpringCloud》。

#### 使用随机数

在一些特殊情况下，我们希望有些参数每次被加载的时候不是一个固定的值，比如密钥、服务端口等。在Spring Boot的属性配置文件中，可以通过使用${random}配置来产生随机的int值、long值或者string字符串，这样我们就可以容易地通过配置随机生成属性，而不是在程序中通过编码来实现这些逻辑。

${random}的配置方式主要有以下几种，读者可作为参考使用。

# 随机字符串

com.didispace.blog.value=${random.value}

# 随机int

com.didispace.blog.number=${random.int}

# 随机long

com.didispace.blog.bignumber=${random.long}

# 10以内的随机数

com.didispace.blog.test1=${random.int(10)}

# 10~20的随机数

com.didispace.blog.test2=${random.int[10,20]}

该配置方式可以设置应用端口等场景，以避免在本地调试时出现端口冲突的麻烦。

#### 命令行参数

使用命令java -jar来启动的方式。该命令除了启动应用之外，还可以在命令行中指定应用的参数，比如java -jar xxx.jar --server.port=8888，直接以命令行的方式来设置server.port属性，并将启动应用的端口设为8888。

在用命令行方式启动Spring Boot应用时，连续的两个减号--就是对application.properties中的属性值进行赋值的标识。所以，java -jar xxx.jar--server.port=8888命令，等价于在application.properties中添加属性server.port=8888。

通过命令行来修改属性值是Spring Boot非常重要的一个特性。通过此特性，理论上已经使得应用的属性在启动前是可变的，所以其中的端口号也好、数据库连接也好，都是可以在应用启动时发生改变的，而不同于以往的Spring应用通过Maven的Profile在编译器中进行不同环境的构建。Spring Boot的这种方式，可以让应用程序的打包内容贯穿开发、测试以及线上部署，而Maven不同Profile的方案为每个环境所构建的包，其内容本质上是不同的。但是，如果每个参数都需要通过命令行来指定，这显然也不是一个好的方案，所以下面我们看看如何在Spring Boot中实现多环境的配置。

#### 多环境配置

我们在开发应用的时候，通常同一套程序会被应用和安装到几个不同的环境中，比如开发、测试、生产等。其中每个环境的数据库地址、服务器端口等配置都不同，如果在为不同环境打包时都要频繁修改配置文件的话，那必将是个非常烦琐且容易发生错误的事。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot也不例外，或者说实现起来更加简单。

在Spring Boot中，多环境配置的文件名需要满足application-{profile}.properties的格式，其中{profile}对应你的环境标识，如下所示。

·application-dev.properties：开发环境。

·application-test.properties：测试环境。

·application-prod.properties：生产环境。至于具体哪个配置文件会被加载，需要在application.properties文件中通过spring.profiles.active属性来设置，其值对应配置文件中的{profile}值。如spring.profiles.active=test就会加载application-test.properties配置文件内容。

下面，以不同环境配置不同的服务端口为例，进行样例实验。

·针对各环境新建不同的配置文件application-dev.properties、application-test.properties、application-prod.properties。

·在这三个文件中均设置不同的server.port属性，例如，dev环境设置为1111，test环境设置为2222，prod环境设置为3333。

·application.properties中设置spring.profiles.active=dev，意为默认以dev环境设置。

·测试不同配置的加载。

·执行java -jar xxx.jar，可以观察到服务端口被设置为1111，也就是默认的开发环境（dev）。·执行java -jar xxx.jar --spring.profiles.active=test，可以观察到服务端口被设置为2222，也就是测试环境的配置（test）。

·执行java -jar xxx.jar --spring.profiles.active=prod，可以观察到服务端口被设置为3333，也就是生产环境的配置（prod）。

按照上面的实验，可以如下总结多环境的配置思路：

·在application.properties中配置通用内容，并设置spring.profiles.active=dev，以开发环境为默认配置。

·在application-{profile}.properties中配置各个环境不同的内容。

·通过命令行方式去激活不同环境的配置。

#### 加载顺序

在上面的例子中，我们将Spring Boot应用需要的配置内容都放在了项目工程中，已经能够通过spring.profiles.active或是通过Maven来实现多环境的支持。但是，当团队逐渐壮大，分工越来越细致之后，往往不需要让开发人员知道测试或是生产环境的细节，而是希望由每个环境各自的负责人（QA或是运维）来集中维护这些信息。那么如果还是以这样的方式存储配置内容，对于不同环境配置的修改就不得不去获取工程内容来修改这些配置内容，当应用非常多的时候就变得非常不方便。同时，配置内容对开发人员都可见，这本身也是一种安全隐患。对此，出现了很多将配置内容外部化的框架和工具，后续将要介绍的Spring Cloud Config就是其中之一，为了后续能更好地理解Spring Cloud Config的加载机制，我们需要对Spring Boot对数据文件的加载机制有一定的了解。

为了能够更合理地重写各属性的值，Spring Boot使用了下面这种较为特别的属性加载顺序：

1. 在命令行中传入的参数。

2. SPRING\_APPLICATION\_JSON中的属性。SPRING\_APPLICATION\_JSON是以JSON格式配置在系统环境变量中的内容。

3. java:comp/env中的JNDI属性。

4. Java的系统属性，可以通过System.getProperties()获得的内容。5. 操作系统的环境变量。

6. 通过random.\*配置的随机属性。

7. 位于当前应用jar包之外，针对不同{profile}环境的配置文件内容，例如application-{profile}.properties或是YAML定义的配置文件。

8. 位于当前应用jar包之内，针对不同{profile}环境的配置文件内容，例如application-{profile}.properties或是YAML定义的配置文件。

9. 位于当前应用jar包之外的application.properties和YAML配置内容。

10. 位于当前应用jar包之内的application.properties和YAML配置内容。

11. 在@Configuration注解修改的类中，通过@PropertySource注解定义的属性。

12. 应用默认属性，使用SpringApplication.setDefaultProperties定义的内容。

优先级按上面的顺序由高到低，数字越小优先级越高。

可以看到，其中第7项和第9项都是从应用jar包之外读取配置文件，所以，实现外部化配置的原理就是从此切入，为其指定外部配置文件的加载位置来取代jar包之内的配置内容。通过这样的实现，我们的工程在配置中就变得非常干净，只需在本地放置开发需要的配置即可，而不用关心其他环境的配置，由其对应环境的负责人去维护即可。

### 监控与管理

#### 开启HTTP security认证

<dependency>

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-security**</artifactId>**

</dependency>

application.properties

security.user.name=admin

security.user.password=123456

management.security.enabled=true

management.security.role=ADMIN

#### spring-boot-starter-actuator

##### ****关闭验证****

默认情况下很多端点是不允许访问的，会返回401:Unauthorized。

application.properties

management.security.enabled=false

##### /health

{

"status" : "UP",

"diskSpace" : {

"status" : "UP",

"total" : 107374178304,

"free" : 19108171776,

"threshold" : 10485760

}

}

##### /autoconfig

## SpringCloud

spring cloud 为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等。

### 服务治理

服务治理可以说是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的自动化注册与发现。为什么我们在微服务架构中那么需要服务治理模块呢？微服务系统没有它会有什么不好的地方吗？

在最初开始构建微服务系统的时候可能服务并不多，我们可以通过做一些静态配置来完成服务的调用。比如，有两个服务A和B，其中服务A需要调用服务B来完成一个业务操作时，为了实现服务B的高可用，不论采用服务端负载均衡还是客户端负载均衡，都需要手工维护服务B的具体实例清单。但是随着业务的发展，系统功能越来越复杂，相应的微服务应用也不断增加，我们的静态配置就会变得越来越难以维护。并且面对不断发展的业务，我们的集群规模、服务的位置、服务的命名等都有可能发生变化，如果还是通过手工维护的方式，那么极易发生错误或是命名冲突等问题。同时，对于这类静态内容的维护也必将消耗大量的人力。

**服务治理**可以说是微服务架构中最为核心和基础的模块，它主要用来实现**各个微服务实例的自动化注册与发现**。为什么我们在微服务架构中那么需要服务治理模块呢？微服务系统没有它会有什么不好的地方吗？

在最初开始构建微服务系统的时候可能服务并不多，我们可以通过做一些静态配置来完成服务的调用。比如，有两个服务A和B，其中服务A需要调用服务B来完成一个业务操作时，为了实现服务B的高可用，不论采用服务端负载均衡还是客户端负载均衡，都需要手工维护服务B的具体实例清单。但是随着业务的发展，系统功能越来越复杂，相应的微服务应用也不断增加，我们的静态配置就会变得越来越难以维护。并且面对不断发展的业务，我们的集群规模、服务的位置、服务的命名等都有可能发生变化，如果还是通过手工维护的方式，那么极易发生错误或是命名冲突等问题。同时，对于这类静态内容的维护也必将消耗大量的人力。

为了解决微服务架构中的服务实例维护问题，产生了大量的服务治理框架和产品。这些框架和产品的实现都围绕着服务注册与服务发现机制来完成对微服务应用实例的自动化管理。

#### 服务注册

在服务治理框架中，通常都会构建一个注册中心，每个服务单元向注册中心登记自己提供的服务，将主机与端口号、版本号、通信协议等一些附加信息告知注册中心，注册中心按服务名分类组织服务清单。比如，我们有两个提供服务A的进程分别运行于192.168.0.100:8000和192.168.0.101:8000位置上，另外还有三个提供服务B的进程分别运行于192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000位置上。当这些进程均启动，并向注册中心注册自己的服务之后，注册中心就会维护类似下面的一个服务清单。另外，服务注册中心还需要以心跳的方式去监测清单中的服务是否可用，若不可用需要从服务清单中剔除，达到排除故障服务的效果。

#### 服务发现

由于在服务治理框架下运作，服务间的调用不再通过指定具体的实例地址来实现，而是通过向服务名发起请求调用实现。所以，服务调用方在调用服务提供方接口的时候，并不知道具体的服务实例位置。因此，调用方需要向服务注册中心咨询服务，并获取所有服务的实例清单，以实现对具体服务实例的访问。

比如，现有服务C希望调用服务A，服务C就需要向注册中心发起咨询服务请求，服务注册中心就会将服务A的位置清单返回给服务C，如按上例服务A的情况，C便获得了服务A的两个可用位置192.168.0.100:8000和192.168.0.101:8000。当服务C要发起调用的时候，便从该清单中以某种轮询策略取出一个位置来进行服务调用，这就是后续我们将会介绍的客户端负载均衡。这里我们只是列举了一种简单的服务治理逻辑，以方便理解服务治理框架的基本运行思路。

实际的框架为了性能等因素，不会采用每次都向服务注册中心获取服务的方式，并且不同的应用场景在缓存和服务剔除等机制上也会有一些不同的实现策略。

#### Spring Cloud Eureka

Spring Cloud Eureka是Spring Cloud Netflix微服务套件中的一部分，它基于Netflix Eureka做了二次封装，主要负责完成微服务架构中的服务治理功能。Spring Cloud通过为Eureka增加了Spring Boot风格的自动化配置，我们只需通过简单引入依赖和注解配置就能让Spring Boot构建的微服务应用轻松地与Eureka服务治理体系进行整合。

Spring Cloud Eureka，使用Netflix Eureka来实现服务注册与发现，它既包含了服务端组件，也包含了客户端组件，并且服务端与客户端均采用Java编写，所以Eureka主要适用于通过Java实现的分布式系统，或是与JVM兼容语言构建的系统。但是，由于Eureka服务端的服务治理机制提供了完备的RESTful API，所以它也支持将非Java语言构建的微服务应用纳入Eureka的服务治理体系中来。只是在使用其他语言平台的时候，需要自己来实现Eureka的客户端程序。不过庆幸的是，在目前几个较为流行的开发平台上，都已经有了一些针对Eureka注册中心的客户端实现框架，比如.NET平台的Steeltoe、Node.js的eureka-js-client等。

##### Eureka服务端

我们也称为服务注册中心。它同其他服务注册中心一样，支持高可用配置。它依托于强一致性提供良好的服务实例可用性，可以应对多种不同的故障场景。如果Eureka以集群模式部署，当集群中有分片出现故障时，那么Eureka就转入自我保护模式。它允许在分片故障期间继续提供服务的发现和注册，当故障分片恢复运行时，集群中的其他分片会把它们的状态再次同步回来。以在AWS上的实践为例，Netflix推荐每个可用的区域运行一个Eureka服务端，通过它来形成集群。不同可用区域的服务注册中心通过异步模式互相复制各自的状态，这意味着在任意给定的时间点每个实例关于所有服务的状态是有细微差别的。

##### Eureka客户端

主要处理服务的注册与发现。客户端服务通过注解和参数配置的方式，嵌入在客户端应用程序的代码中，在应用程序运行时，Eureka客户端向注册中心注册自身提供的服务并周期性地发送心跳来更新它的服务租约。同时，它也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性地刷新服务状态。

##### 搭建注册服务中心

首先，创建一个基础的Spring Boot工程，命名为eureka-server，并在pom.xml中引入必要的依赖内容，代码如下：<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.7.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.SR5</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常简单，只需在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：

@EnableEurekaServer

@SpringBootApplication

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需在application.properties中增加如下配置：

server.port=1111

eureka.instance.hostname=localhost

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${serve

r.port}/eureka/

由于后续内容也都会在本地运行，为了与后续要进行注册的服务区分，这里将服务注册中心的端口通过server.port属性设置为1111。

eureka.client.register-with-eureka：由于该应用为注册中心，所以设置为false，代表不向注册中心注册自己。

eureka.client.fetch-registry：由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以也设置为false。

在完成了上面的配置后，启动应用并访问http://localhost:1111/。可以看到如下图所示的Eureka信息面板，其中Instances currently registered with Eureka栏是空的，说明该注册中心还没有注册任何服务。

##### 注册服务提供者

在完成了服务注册中心的搭建之后，接下来我们尝试将一个既有的Spring Boot应用加入Eureka的服务治理体系中去。

可以使用上一章中实现的快速入门工程来进行改造，将其作为一个微服务应用向服务注册中心发布自己。首先，修改pom.xml，增加Spring Cloud Eureka模块的依赖，具体代码如下所示：

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.SR5</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

接着，改造/hello请求处理接口，通过注入DiscoveryClient对象，在日志中打印出服务的相关内容。

@RestController

public class HelloController {

private final Logger logger = Logger.getLogger(getClass());

@Autowired

private DiscoveryClient client;

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String index() {

ServiceInstance instance = client.getLocalServiceInstance();

logger.info("/hello, host:" + instance.getHost() + ", service\_id:" +

instance.getServiceId());

return "Hello World";

}

}

然后，在主类中通过加上@EnableDiscoveryClient注解，激活Eureka中的DiscoveryClient实现（自动化配置，创建DiscoveryClient接口针对Eureka客户端的EurekaDiscoveryClient实例），才能实现上述Controller中对服务信息的输出。

@EnableDiscoveryClient

@SpringBootApplication

public class HelloApplication {

public static void main(String[] args) {

SpringApplication.run(HelloApplication.class, args);

}

}

最后，我们需要在application.properties配置文件中，通过spring.application.name属性来为服务命名，比如命名为hello-service。再通过eureka.client.serviceUrl.defaultZone属性来指定服务注册中心的地址，这里我们指定为之前构建的服务注册中心地址，完整配置如下所示：

spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

下面我们分别启动服务注册中心以及这里改造后的hello-service服务。在hello-service服务控制台中，Tomcat启动之后，com.netflix.discovery.DiscoveryClient对象打印了该服务的注册信息，表示服务注册成功。

s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)

c.n.e.EurekaDiscoveryClientConfiguration : Updating port to 8080

com.didispace.HelloApplication : Started HelloApplication in 7.218

seconds (JVM running for 11.646)

com.netflix.discovery.DiscoveryClient : DiscoveryClient\_HELLO-SERVICE/PC-

201602152056:hello-service-registration status: 204

而此时在服务注册中心的控制台中，可以看到类似下面的输出，名为hello-service的服务被注册成功了。

c.n.e.registry.AbstractInstanceRegistry : Registered instance

HELLO-SERVICE/PC-201602152056:hello-service with status UP (replication=true)

我们也可以通过访问Eureka的信息面板，在Instances currently registered with Eureka一栏中看到服务的注册信息。

通过访问http://localhost:8080/hello，直接向该服务发起请求，在控制台中可以看到如下输出：

com.didispace.web.HelloController : /hello, host:PC-201602152056,

service\_id:hello-service

这些输出内容就是之前我们在HelloController中注入的DiscoveryClient接口对象，从服务注册中心获取的服务相关信息。

## 设计模式

## Maven

## Gradle