第4天-Java虚拟机详解

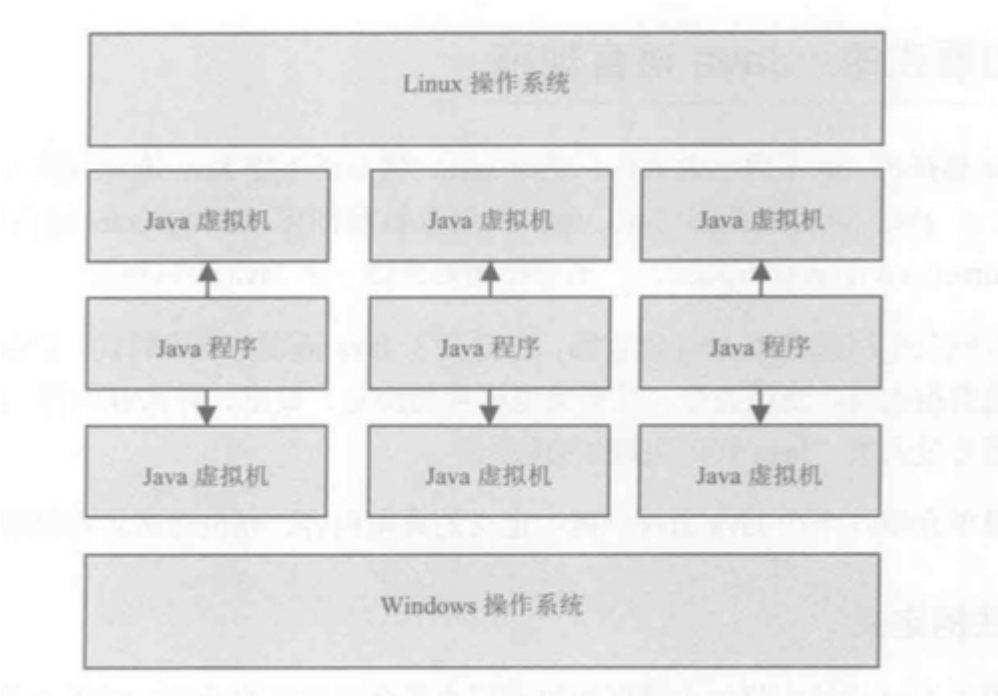
一、JVM 虚拟机常识

1、什么是JAVA虚拟机

所谓虚拟机，就是一台虚拟的计算机。他是一款软件，用来执行一系列虚拟计算机指令。大体上，虚拟机可以分为系统虚拟机和程序虚拟机。大名鼎鼎的 VisualBox、VMware就属于系统虚拟机。他们完全是对物理计算机的仿真。提供了一个可以运行完整操作系统的软件平台。 程序虚拟机的典型代表就是Java虚拟机，它专门为执行单个计算机程序而设计，在Java虚拟机中执行的指令我们称为Java字节码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。

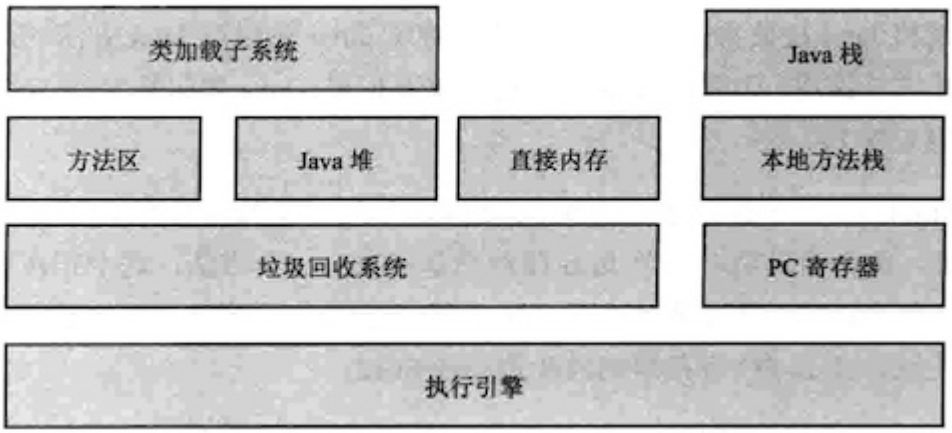
2、JAVA 如何做到跨平台

同一个JAVA程序(JAVA字节码的集合)，通过JAVA虚拟机(JVM)运行于各大主流操作系统平台 比如Windows、CentOS、Ubuntu等。程序以虚拟机为中介，来实现跨平台。



3、虚拟机基本结构

我们要对JVM虚拟机的结构有一个感性的认知。毕竟我们不是编程人员，认知程度达不到那么深入。



1、类加载子系统

负责从文件系统或者网络中加载 Class 信息，加载的类信息存放于一块称为方法区的内存空间。除了类信息外，方法区中可能还会存放运行时常量池信息，包括字符串字面量和数字量。

2、Java堆

在虚拟机启动的时候建立，它是Java程序最主要的内存工作区域。几乎所有的Java对象实例都放Java堆中。堆空间是所有线程共享的，这是一块与Java应用密切相关的内存空间。

3、Java的NIO库(直接内存)

允许Java程序使用直接内存。直接内存是在Java堆外的、直接向系统申请的内存空间。通常访问直接内存的速度会优于Java堆。因此出于性能考虑，读写频繁的场合考虑使用直接内存。由于直接内存在Java堆外，因此它的大小不会受限于Xmx指定的最大堆大小。但是系统内存是有限的，Java堆和直接内存的总和依然受限于操作系统能给出的最大内存。

4、垃圾回收系统

垃圾回收系统是Java虚拟机的重要组成部分，垃圾回收器可以对方法区、Java堆和直接内存进行回收。

5、Java栈

每一个Java虚拟机线程都有一个私有的Java栈。一个线程的Java栈在线程创建的时候被创建。Java保存着帧信息，Java栈中保存着局部变量、方法参数，同时和Java方法的调用、返回密切相关。

6、本地方法

与Java栈非常类似，最大的不同在于Java栈用于Java方法的调用，而本地方法栈用于本地方法调用。作为Java虚拟机的重要扩展，Java虚拟机运行Java程序直接调用本地方法（通常使用C编写）。

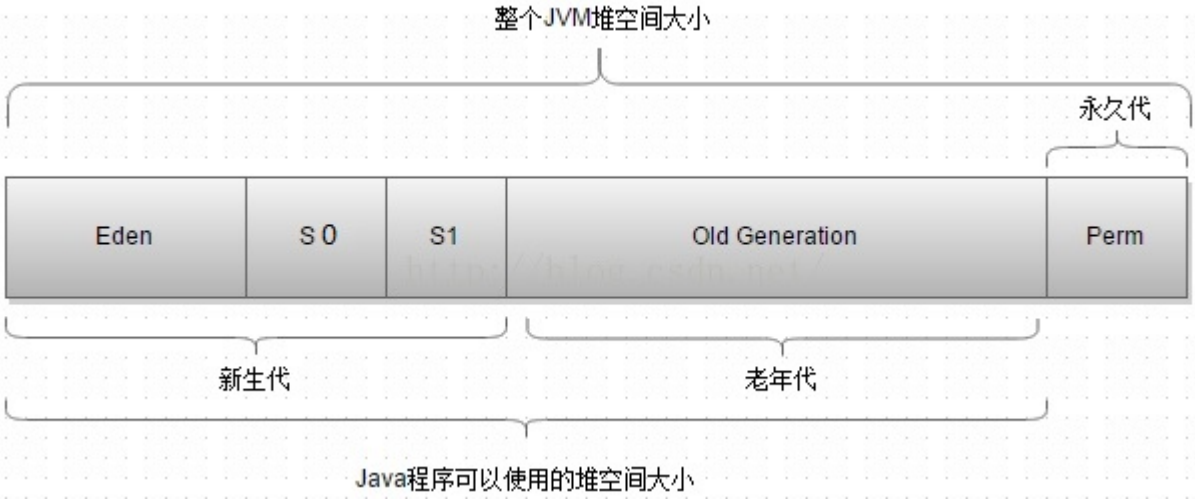
7、PC寄存器

每个线程私有的空间，Java虚拟机会为每一个Java线程创建PC寄存器。在任意时刻，一个Java线程总是在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC寄存器就会指向当前正在被执行的指令。如果当前方法是本地方法，那么PC寄存的值就是undefined.

8、执行引擎

是Java虚拟机最核心组件之一，它负责执行虚拟机的字节码。使用即时编译技术将方法编译成机器码后再执行。

4、虚拟机堆内存结构



JVM中堆空间可以分成三个大区，年轻代、老年代、永久代(方法区)。

1、年轻代

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集那些生命周期短的对象。年轻代分为三个区域：EDEN、Survivor 0(简称S0,也通常称为from区)、Survivor 1(简称S1，也通常称为to区)。其中S0与S1的大小是相同等大的，三者所占年轻代的比例大致为8:1:1，S0与S1就像"孪生兄弟"一样，我们大家不必去纠结此比例(可以通过修改JVM某些动态参数来调整)的大小.只需谨记三点就好： 1、S0与S1相同大小。 2、EDEN区远比S(S0+S1)区大,EDEN占了整个年轻代的大致70%至80%左右。 3、年轻代分为2个区(EDEN区、Survivor区)、3个板块(EDEN、S0、S1)。

2、老年代

在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。 那一个对象到底要经过多少次垃圾回收才能从年轻代进入老年代呢? 我们通常认为在新生代中的对象，每经历过一次GC(垃圾回收),如果它没有被回收，它的年龄就会被加1， 虚拟机提供了一个参数来可控制新生代对象的最大年龄:MaxTenuringThreshold。默认情况下，这个参数是15。 也就是说，在新生代的对象最多经历15次GC，就可以进入老年代。 假如存在一种这样的情况，一个新生代对象，占用新生代空间特别大。在GC时若不回收，新生代空间将不足。但是若要回收，程序还没有使用完。此时就不会依据这个对象的MaxTenuringThreshold 参数。而是直接晋升到老年代。所以说MaxTenuringThreshold 参数是晋升老年代的充分非必要条件。

3、永久代(方法区)

也通常被叫做方法区。是一块所有线程共享的内存区域。用于保存系统的类信息，比如类的字段、方法、常量池。

5、JVM 虚拟机参数类型

1、标准参数

标准参数中包括功能和输出的结果都是很稳定的，基本上不会随着JVM版本的变化而变化。可以用java -help检索出所有标准参数。

2、X 类型参数

非标准化的参数，在将来的版本中可能会改变。所有的这类参数都以 -X 开始，并且可以用 java -X 来检索。 注意，不能保证所有参数都可以被检索出来.

3、XX 类型参数

非标准化的参数，它们同样不是标准的，随着JVM版本的变化可能会发生变化。在实际情况中 X 参数和 XX 参数并没有什么不同。X 参数的功能是十分稳定的，然而很多 XX 参数主要用于JVM调优和debug。

所有的 XX 参数都以"-XX:"开始，但是随后的语法不同，取决于参数的类型： 1、对于布尔类型的参数，我们有"+"或"-"，然后才设置 JVM 选项的实际名称。 例如，-XX:+ 用于激活选项，而 -XX:- 用于注销选项。

Example: 开启GC日志的参数: -XX:+PrintGC 2、对于需要非布尔值的参数，如 string 或者 integer，我们先写参数的名称，后面加上"="，最后赋值。 例如: -XX:MaxPermSize=2048m

6、常用的JVM参数

以上介绍完了JVM的三类参数类型，接下来我们主要聊聊常用的JVM参数。

1、跟踪JAVA虚拟机的垃圾回收

JVM 的 GC的日志是以替换的方式(>)写入的，而不是追加(>>)，如果下次写入到同一个文件中的话，以前的GC内容会被清空。这导致我们重启了JAVA服务后，历史的GC日志将会丢失。

```
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Xloggc:filename
```

Example

此种写法，会导致JAVA服务重启后，GC日志丢失

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/data0/logs/gc.log
```

在这里GC 日志支持 %p 和 %t 两个参数:

- %p 将会被替换为对应的进程 PID
- %t 将会被替代为时间字符串，格式为: YYYY-MM-DD_HH-MM-SS

此种写法，不管怎么重启，GC历史日志将不会丢失

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/data0/logs/gc-%t.log"
```

2、配置JAVA虚拟机的堆空间

```
-Xms:初始堆大小
-Xmx:最大堆大小
# 实际生产环境中， 我们通常将初始化堆(-Xms) 和 最大堆(-Xmx) 设置为一样大。以避免程序频繁的申请堆空间。


-Xmn: 设置年轻代大小，至于这个参数则是对 -XX:newSize、-XX:MaxnewSize两个参数的同时配置，

-XX:NewRatio=   #设置年轻代和老年代的比值
-XX:SurvivorRatio=eden/from=eden/to # 年轻代中Eden区与两个Survivor区的比值
```

Example:

```
-Xmn1m -XX:SurvivorRatio=2
# 这里的eden 于from(to) 的比值为2:1 ，因此在新生代为1m的空间里， eden 区为 512KB, from 和 to 分别为
256KB. 而新生代总大小为 512KB + 256KB + 256KB = 1MB
```

```
-Xms20M -Xmx20M -XX:NewRatio=2
# 这里 老年代和新生代的比值为2:1 ， 因此在堆大小为20MB的区间里， 新生代大小为: 20MB * 1/3 = 6MB左右
# 老年代为 13MB 左右。
```

3、配置JAVA虚拟机的永久区(方法区)

```
-XX:PermSize=n    # 设置持久代初始内存分配大小.
-XX:MaxPermSize=n # 设置持久代分配的内存的最大上限.
```

4、配置JAVA虚拟机的栈

```
-Xss128k # 设置每个线程的堆栈大小
```

7、常用垃圾回收算法

1、引用计数法

引用计数法是最经典的一种垃圾回收算法。其实现很简单，对于一个A对象，只要有任何一个对象引用了A，则A的引用计算器就加1，当引用失效时，引用计数器减1.只要A的引用计数器值为0，则对象A就不可能再被使用。但是该算法却存在严重的问题： 1、无法处理循环引用的问题，因此在Java的垃圾回收器中，没有使用该算法。

由于引用计数器算法存在循环引用以及性能的问题，java虚拟机并未使用此算法作为垃圾回收算法。

2、标记清除法

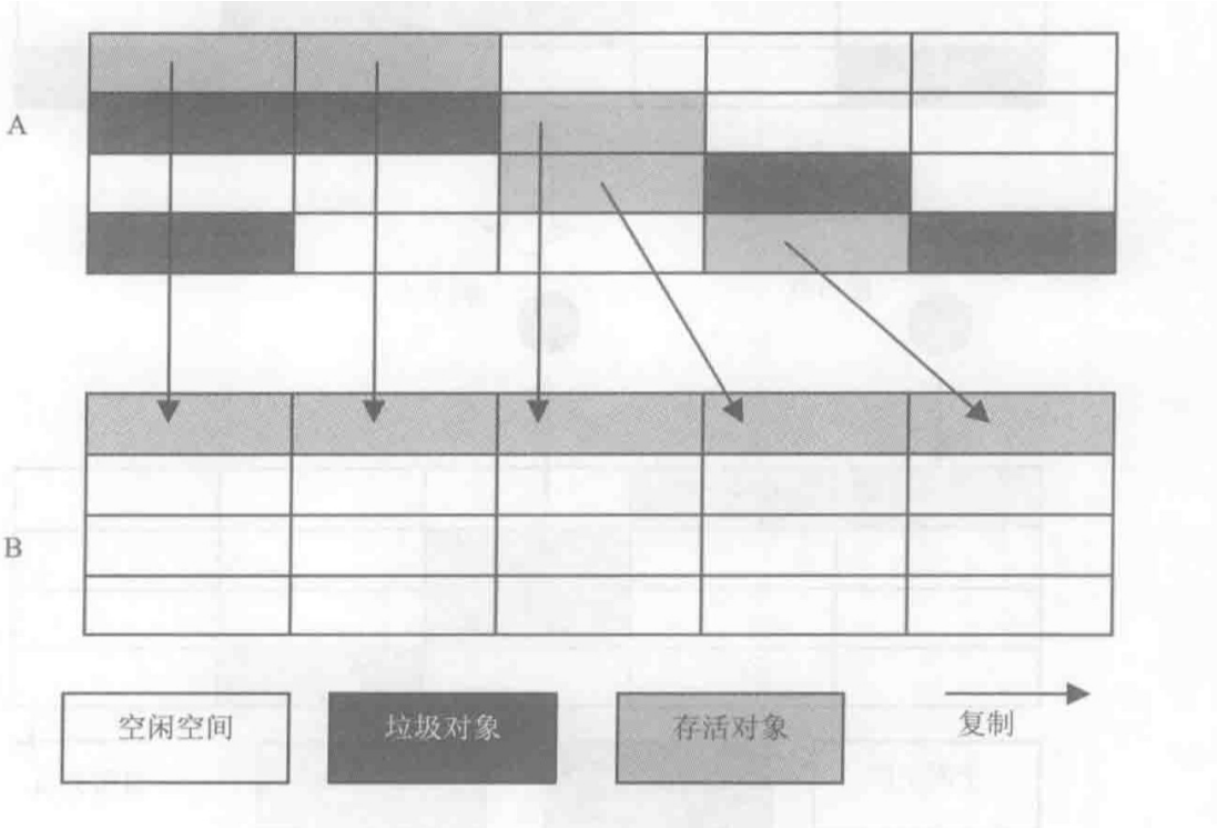
标记-清除算法是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。 缺陷: ①.效率问题：标记清除过程效率都不高。 ②.空间问题：标记清除之后会产生大量的不连续的内存碎片

3、 标记压缩法

标记-压缩算法适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。 标记-压缩算法的最终效果等同于标记-清除算法执行完成之后，再进行一次内存碎片的整理。基于此，这种算法也解决了内存碎片问题。

4、复制算法

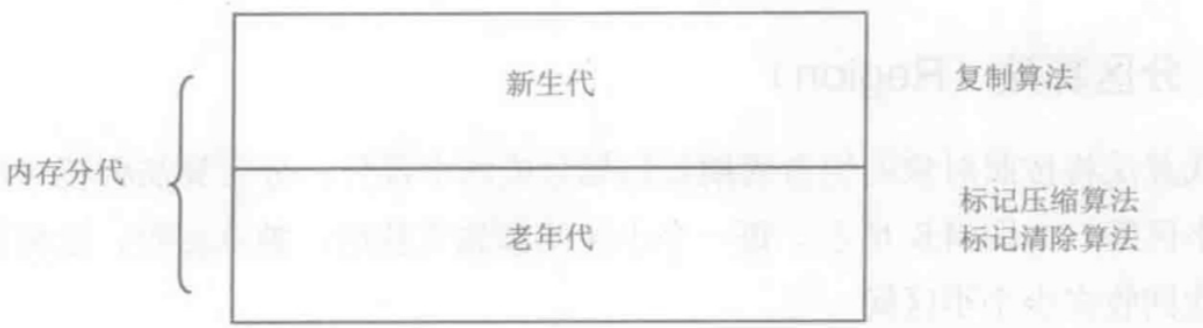
与标记-清除算法相比，复制算法是一种相对高效的回收方法。但不适用于存活对象较多的场合，如老年代。它将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收 缺陷： 空间浪费，浪费了50%的内存空间。



我们前文介绍的JVM 的新生代，分为Eden 及 S0 和 S1 ， 其中S0 和 S1 是两个容量相等的区域。其实在JVM 的垃圾回收中, S0 和 S1 就使用了复制算法作为它们的垃圾回收算法。

5、分代算法

介绍了复制、标记清除、标记压缩等垃圾回收算法。在所有的算法中，并没有一种算法可以完全取代其他算法，它们都具有自己独特的优势和特点。因此，根据垃圾回收对象的特性，使用合适的算法回收，才是明智的选择。分代算法就是基于这种思想，它将内存区间根据对象的特点分成几块，根据每块内存区间的特点，使用不同的回收算法，以提高垃圾回收的效率。 一般来说，Java虚拟机会将所有的新建对象都放到称为新生代的区域中，大约90%的新建对象会被回收，因此新生代比较适合使用复制算法。当一个对象经过几次回收后依然存活，对象就会被放到称为老年代的内存空间。在老年代中，几乎所有对象都经过几次垃圾回收后依然得以存活的。因此可以认为对象在一段时期内，甚至在应用程序的整个生命周期中，将是常驻内存的。 在极端情况下，老年代对象的存活率可以达到100%。如果依然使用复制算法回收老年代，将需要复制大量对象。根据分代的思想，可以对老年代的回收使用与新生代不通的标记压缩或者标记清除算法，以提高垃圾回收效率。



二、JVM 运维实用排障工具

准备实验环境

1.准备一台服务器

```
#安装 Tomcat & JDK
安装时候选择 tomcat 软件版本要与程序开发使用的版本一致。jdk 版本要进行与 tomcat 保持一致。

#系统环境说明
[root@qfedu.com ~]# getenforce
Disabled
[root@qfedu.com ~]# systemctl status firewalld.service
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled; vendor preset: enabled)
   Active: inactive (dead)
     Docs: man:firewalld(1)
安装 JDK
1.上传jdk到服务器中，安装jdk
[root@qfedu.com ~]# mkdir /application   #创建工作目录
[root@qfedu.com ~]# tar xzf jdk-8u60-linux-x64.tar.gz -C /application/
[root@qfedu.com ~]# mv /application/jdk1.8.0_60 /application/jdk
# 设置环境变量
[root@qfedu.com ~]# vim /etc/profile
export JAVA_HOME=/application/jdk   #指定java安装目录
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH    #用于指定java系统查找命令的路径
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib:$JAVA_HOME/lib/tools.jar  #类的路径，在编译运行java程序时，如果有调用到其他类的时候，在classpath中寻找需要的类。
[root@qfedu.com ~]# source /etc/profile  #让环境变量生效

#测试jdk是否安装成功
[root@qfedu.com ~]# java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)

#安装Tomcat
将tomcat安装包上传到服务器中:
[root@qfedu.com ~]# tar xzf apache-tomcat-8.0.27.tar.gz -C /application/
[root@qfedu.com ~]# mv /application/apache-tomcat-8.0.27 /application/tomcat
# 设置环境变量
[root@qfedu.com ~]# echo 'export TOMCAT_HOME=/application/tomcat'>>/etc/profile
[root@qfedu.com ~]# source /etc/profile
启动tomcat
[root@qfedu.com ~]# /application/tomcat/bin/startup.sh
```

1、jps

用来查看Java程序进程的具体状态, 包括进程ID，进程启动的路径及启动参数等等，与unix上的ps类似，只不过jps是用来显示java进程。 常用参数如下:

```
-q：忽略输出的类名、Jar名以及传递给main方法的参数，只输出pid
-m：输出传递给main方法的参数，如果是内嵌的JVM则输出为null
-l：输出完全的包名，应用主类名，jar的完全路径名
-v：输出传给jvm的参数
```

注意: 使用jps 时的运行账户要和JVM 虚拟机启动的账户一致。若启动JVM虚拟机是运行的账户为www，那使用jps指令时，也要使用www 用户去指定。 sudo -u www jps

Example

查看已经运行的 JVM 进程的实际启动参数

```
[root@qfedu.com bin]# jps  -v
38372 Jps -Dapplication.home=/usr/local/jdk -Xms8m
38360 Bootstrap -Djava.util.logging.config.file=/data0/tomcat/conf/logging.properties -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Xms4096m -Xmx4096m -
XX:PermSize=1024m -XX:MaxPermSize=2048m -Djdk.tls.ephemeralDHKeySize=2048 -
Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Dignore.endorsed.dirs= -
Dcatalina.base=/data0/tomcat -Dcatalina.home=/data0/tomcat -Djava.io.tmpdir=/data0/tomcat/temp
```

2、jstack

jstack 用于打印出给定的java进程ID或远程调试服务的Java堆栈信息。jstack是非常有用的。打印出来的信息通常在运维的过程中被保存起来(保存故障现场)，以供RD们去分析故障。

常用参数如下:

```
jstack <pid>
jstack [-l] <pid> //长列表. 打印关于锁的附加信息
jstack [-F] <pid> //当’jstack [-l] pid’没有响应的时候强制打印栈信息
```

Example

打印JVM 的堆栈信息，以供问题排查

```
[root@qfedu.com ~]# jstack -F 38360 > /tmp/jstack.log
```

3、jinfo

可以查看或修改运行时的JVM进程的参数。 常用参数:

```
jinfo [option] pid

where <option> is one of:
    -flag <name>         to print the value of the named VM flag
    -flag [+|-]<name>    to enable or disable the named VM flag
    -flag <name>=<value> to set the named VM flag to the given value
    -flags               to print VM flags
```

Example

```
# 根据 PID 查看目前分配的最大堆栈
[root@qfedu.com ~]# jinfo -flag MaxHeapSize 38360
-XX:MaxHeapSize=4294967296
# 动态更改 JVM 的最大堆栈值
[root@qfedu.com ~]# jinfo -flag MaxHeapSize=4294967296  38360
Exception in thread "main" com.sun.tools.attach.AttachOperationFailedException: flag
'MaxHeapSize' cannot be changed

    at sun.tools.attach.LinuxVirtualMachine.execute(LinuxVirtualMachine.java:229)
    at sun.tools.attach.HotSpotVirtualMachine.executeCommand(HotSpotVirtualMachine.java:261)
    at sun.tools.attach.HotSpotVirtualMachine.setFlag(HotSpotVirtualMachine.java:234)
    at sun.tools.jinfo.JInfo.flag(JInfo.java:134)
    at sun.tools.jinfo.JInfo.main(JInfo.java:81)

# jinfo 并不能动态的改变所有的JVM 参数。 那到底有哪些参数能够被动态的改变呢?
# java -XX:+PrintFlagsFinal -version  查看JVM 的所有参数
# java -XX:+PrintFlagsFinal -version | grep manageable

#查看可以动态修改的参数
[root@qfedu.com ~]# java -XX:+PrintFlagsFinal -version | grep manageable
     intx CMSAbortablePrecleanWaitMillis            = 100
{manageable}
     intx CMSTriggerInterval                        = -1
{manageable}
     intx CMSWaitDuration                           = 2000
{manageable}
     bool HeapDumpAfterFullGC                       = false
{manageable}
     bool HeapDumpBeforeFullGC                      = false
{manageable}
     bool HeapDumpOnOutOfMemoryError                = false
{manageable}
    ccstr HeapDumpPath                              =
{manageable}
    uintx MaxHeapFreeRatio                          = 70
{manageable}
    uintx MinHeapFreeRatio                          = 40
{manageable}
     bool PrintClassHistogram                       = false
{manageable}
     bool PrintClassHistogramAfterFullGC            = false
{manageable}
     bool PrintClassHistogramBeforeFullGC           = false
{manageable}
     bool PrintConcurrentLocks                      = false
{manageable}
     bool PrintGC                                   = false
{manageable}
     bool PrintGCDateStamps                         = false
{manageable}
```

```
     bool PrintGCDetails                           = false
{manageable}
     bool PrintGCID                                = false
{manageable}
     bool PrintGCTimeStamps                        = false
{manageable}

# 也只有以上这些值才能够动态的被改变
[root@qfedu.com ~]# jinfo -flag CMSWaitDuration=1900  38360
# 查看， jinfo -flags 查看 JVM 的 flags
[root@qfedu.com ~]# jinfo -flags 38360
Attaching to process ID 38360, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.91-b14
Non-default VM flags: -XX:CICompilerCount=2 -XX:CMSWaitDuration=1900 -
XX:InitialHeapSize=4294967296 -XX:MaxHeapSize=4294967296 -XX:MaxNewSize=1431633920 -
XX:MinHeapDeltaBytes=196608 -XX:NewSize=1431633920 -XX:OldSize=2863333376 -
XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps
Command line:  -Djava.util.logging.config.file=/data0/tomcat/conf/logging.properties -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Xms4096m -Xmx4096m -
XX:PermSize=1024m -XX:MaxPermSize=2048m -Djdk.tls.ephemeralDHKeySize=2048 -
Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Dignore.endorsed.dirs= -
Dcatalina.base=/data0/tomcat -Dcatalina.home=/data0/tomcat -Djava.io.tmpdir=/data0/tomcat/temp
```

4、jstat

主要利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对Heap size和垃圾回收状况的监控。常用指令:

```
[root@qfedu.com ~]# jstat -gc 113059 1000 10 // 打印PID 为 113059 JVM 状态，一共打印10次，每次间隔时间为1s(1000ms)

-gc 用于查看JVM中堆的垃圾收集情况的统计
```

注: jstat 的用法超级强大， 我们这里只是列举出列其中一个简单的应用。

Example

```
[root@qfedu.com ~]# jstat -gc 113059 1000 10
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT
   FGC    FGCT     GCT
195904.0 195904.0  0.0   21610.3 1567680.0 1516721.9 8526272.0  3557507.8  1048576.0 163148.4
2577   92.033   0      0.000   92.033
195904.0 195904.0 23600.9  0.0   1567680.0 142541.6 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 266338.1 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 413941.8 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 642390.6 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 813957.3 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 984223.2 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 1155472.7 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0 23600.9  0.0   1567680.0 1399228.5 8526272.0  3558435.8  1048576.0 163148.4
2578   92.060   0      0.000   92.060
195904.0 195904.0  0.0   23866.6 1567680.0 38005.6  8526272.0  3559196.7  1048576.0 163148.4
2579   92.092   0      0.000   92.092
```

字段意义如下

列名	说明
S0C	新生代中Survivor space中S0当前容量的大小（KB）
S1C	新生代中Survivor space中S1当前容量的大小（KB）
S0U	新生代中Survivor space中S0容量使用的大小（KB）
S1U	新生代中Survivor space中S1容量使用的大小（KB）
EC	Eden space当前容量的大小（KB）
EU	Eden space容量使用的大小（KB）
OC	Old space当前容量的大小（KB）
OU	Old space使用容量的大小（KB）
PC	Permanent space当前容量的大小（KB）
PU	Permanent space使用容量的大小（KB）
YGC	从应用程序启动到采样时发生 Young GC 的次数
YGCT	从应用程序启动到采样时 Young GC 所用的时间(秒)
FGC	从应用程序启动到采样时发生 Full GC 的次数
FGCT	从应用程序启动到采样时 Full GC 所用的时间(秒)
GCT	T从应用程序启动到采样时用于垃圾回收的总时间(单位秒)，它的值等于YGC+FGC

三、JVM 运维实用监控工具

1、VirtualVM 介绍

VisualVM 是一款免费的性能分析工具。它通过 jvmstat、JMX、SA（Serviceability Agent）服务性代理以及 Attach API 等多种方式从程序运行时获得实时数据，从而进行动态的性能分析。同时，它能自动选择更快更轻量级的技术尽量减少性能分析对应用程序造成的影响，提高性能分析的精度。

1、安装 VisualVM

先在电脑上面安装jdk

到官网下载相应操作系统对应的软件. 官网地址: http://visualvm.github.io/

2、安装 VisualVM 插件

1.VisualVM 插件中心提供很多插件以供安装

可以通过 VisualVM 应用程序安装，或者从 VisualVM 插件中心手动下载插件，然后离线安装。

从 VisualVM 插件中心安装插件步骤:

- 从主菜单中选择“工具”>“插件”。
- 在“可用插件”标签中，选中该插件的“安装”复选框。单击“安装“。

- 逐步完成插件安装程序。

2、离线安装插件

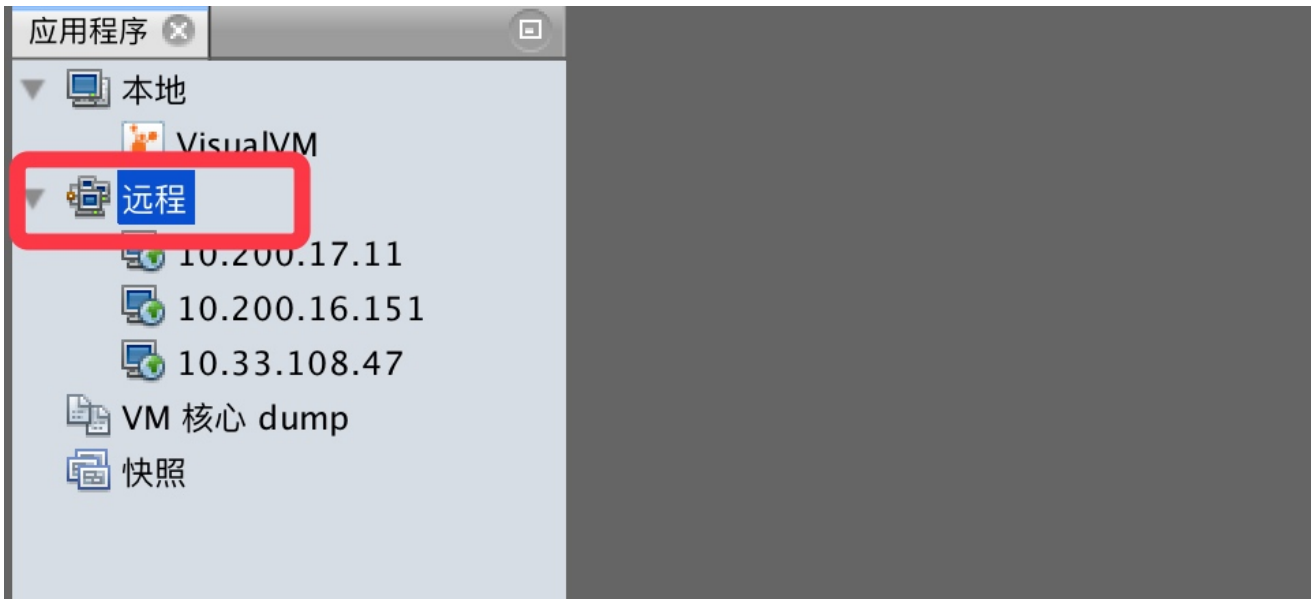
- 到插件中心官网: http://visualvm.github.io/pluginscenters.html 下载对应的VisualVM版本的插件
- 从主菜单中选择“工具”>“插件”。
- 在“已下载”标签中,点击"添加插件"按钮,选择已下载的插件文件 (以.nbm结尾) 打开。
- 选中要打开的插件文件，并单击"安装"按钮，逐步完成插件安装程序。

3、如何监控 JVM

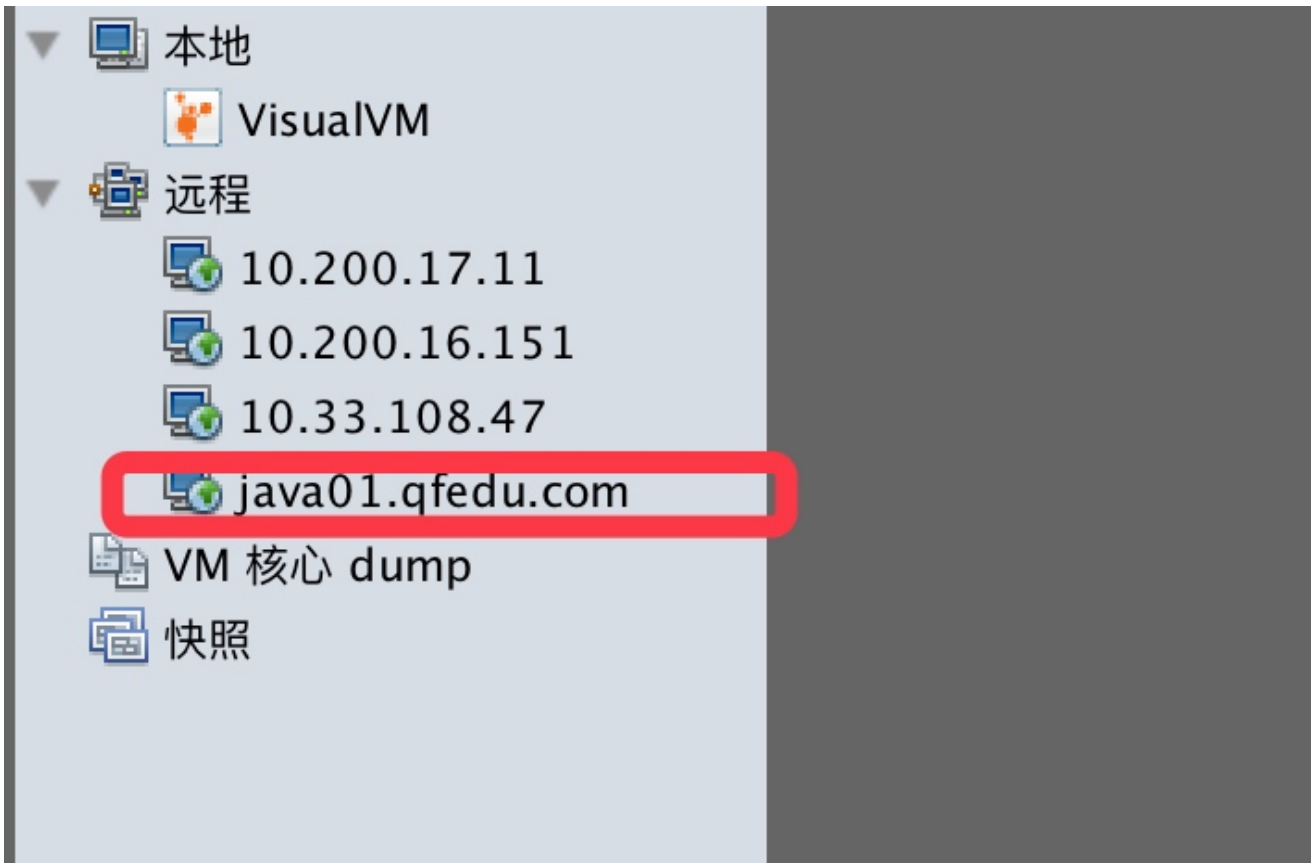
那如何通过VisualVM 去分析远程的JVM虚拟机里的信息呢? 首先要在JVM中开启相关配置，以供VisualVM能够从中获取JVM的信息，如何配置JVM呢？ 这里以Tomcat为例:

```
在Tomcat的 catalina.sh 或者是setenv.sh 脚本中加上如下参数
 CATALINA_OPTS="$CATALINA_OPTS -Dcom.sun.management.jmxremote
 -Dcom.sun.management.jmxremote.port=一个监听端口
 -Dcom.sun.management.jmxremote.authenticate=false
 -Dcom.sun.management.jmxremote.ssl=false
 -Djava.rmi.server.hostname=可解析的主机名称"

具体测试参数如下:
 CATALINA_OPTS="$CATALINA_OPTS -Dcom.sun.management.jmxremote
 -Dcom.sun.management.jmxremote.port=11412
 -Dcom.sun.management.jmxremote.authenticate=false
 -Dcom.sun.management.jmxremote.ssl=false
 -Djava.rmi.server.hostname=java01.qfedu.com"

 注意: 这里的java01.qfedu.com 必须是DNS可解析的主机名，11412 为监听端口

 其次在VisualVM 中配置连接到上面配置的服务器加端口的地址上去:
 * 在VisualVM中选择"远程"
 * 在弹出的窗口中选择"添加远程主机"
 * 此时会在远程中，多出一台主机。此时点击多出来的主机
 * 添加连接JVM相关的信息
```

4、VisualVM 如何监控 JVM具体操作


应用程序
本地
VisualVM
远程
10.200.17.11
10.200.16.151
10.33.108.47
VM 核心 dump
快照


添加远程主机
主机名(H):
java01.qfedu.com
显示名称(D):
java01.qfedu.com
高级设置(A)
取消
确定


本地
VisualVM
远程
10.200.17.11
10.200.16.151
10.33.108.47
java01.qfedu.com
VM 核心 dump
快照


添加 JMX 连接
连接(C):
java01.qfedu.com:11412
用法: <主机名>:<端口> 或 service:jmx:<协议>:<sap
显示名称(D):
java01.qfedu.com:11412
使用安全凭证(E)
用户名(U):
口令(P):
保存安全凭证(S)
不要求SSL连接(N)
取消
确定


VisualVM 1.3.8
应用程序
本地
VisualVM
远程
10.200.17.11
10.200.16.151
10.33.108.47
java01.qfedu.com:11412 (pid 15532)
VM 核心 dump
快照
java01 qfedu com:11412 (pid 15532)
概述
监视
线程
抽样器
MBeans
java01.qfedu.com:11412 (pid 15532)
概述
保存的数据
详细信息
PID: 15532
主机: java01.qfedu.com
主类: <未知>
参数: <无>
JVM: Java HotSpot(TM) 64-Bit Server VM (25.111-b14, mixed mode)
Java: 版本 1.8.0_111, 供应商 Oracle Corporation
Java Home 目录: /usr/local/matrix/jre