目录

[JVM 3](#_Toc45187489)

[Jvm，jre,jdk的关系 4](#_Toc45187490)

[运行时数据区 5](#_Toc45187491)

[程序计数器 5](#_Toc45187492)

[虚拟机栈 5](#_Toc45187493)

[本地方法栈 6](#_Toc45187494)

[堆 6](#_Toc45187495)

[元空间（方法区） 7](#_Toc45187496)

[直接内存 7](#_Toc45187497)

[Java对象 7](#_Toc45187498)

[对象的创建 7](#_Toc45187499)

[创建对象的方式 7](#_Toc45187500)

[使用new创建对象过程 7](#_Toc45187501)

[对象的内存布局 8](#_Toc45187502)

[对象头 8](#_Toc45187503)

[实例数据 8](#_Toc45187504)

[对齐填充 9](#_Toc45187505)

[OOM异常 9](#_Toc45187506)

[起因 9](#_Toc45187507)

[Java堆溢出 9](#_Toc45187508)

[虚拟机栈和本地方法栈溢出 10](#_Toc45187509)

[方法区和运行时常量池内存溢出 10](#_Toc45187510)

[类加载 10](#_Toc45187511)

[类加载过程 10](#_Toc45187512)

[加载 10](#_Toc45187513)

[验证 11](#_Toc45187514)

[准备 11](#_Toc45187515)

[解析 11](#_Toc45187516)

[初始化 12](#_Toc45187517)

[类加载器 12](#_Toc45187518)

[双亲委派模式 13](#_Toc45187519)

[自定义类加载器 13](#_Toc45187520)

[打破双亲委派模式 14](#_Toc45187521)

[垃圾收集器 14](#_Toc45187522)

[回收对象判断 14](#_Toc45187523)

[引用计数法 14](#_Toc45187524)

[可达性分析算法 14](#_Toc45187525)

[引用 15](#_Toc45187526)

[收集算法 15](#_Toc45187527)

[发生回收的时候 15](#_Toc45187528)

[标记-清除算法 16](#_Toc45187529)

[复制算法 16](#_Toc45187530)

[标记-整理算法 16](#_Toc45187531)

[分代收集算法 16](#_Toc45187532)

[收集器种类 18](#_Toc45187533)

[Serial收集器 18](#_Toc45187534)

[ParNew收集器 18](#_Toc45187535)

[Parallel scavenge收集器 18](#_Toc45187536)

[Serial old收集器 19](#_Toc45187537)

[Parallel old收集器 19](#_Toc45187538)

[CMS收集器 19](#_Toc45187539)

[G1收集器 20](#_Toc45187540)

[Gc配置参数 21](#_Toc45187541)

[Stop the world 21](#_Toc45187542)

[常用的jvm参数 22](#_Toc45187543)

[-x参数 22](#_Toc45187544)

[-XX参数 23](#_Toc45187545)

[虚拟机性能监控 24](#_Toc45187546)

[jdk命令工具 24](#_Toc45187547)

[查看进程jps 24](#_Toc45187548)

[统计jvm运行状态信息jstat 24](#_Toc45187549)

[查看jvm配置参数jinfo 25](#_Toc45187550)

[堆快照生成命令jmap 25](#_Toc45187551)

[Jhat 25](#_Toc45187552)

[查询jvm当前的线程快照信息jstack 25](#_Toc45187553)

[可视化工具 26](#_Toc45187554)

[Jconsole 26](#_Toc45187555)

[Visualvm 27](#_Toc45187556)

[内存分析 27](#_Toc45187557)

[Gc日志分析 27](#_Toc45187558)

[执行引擎 27](#_Toc45187559)

[性能优化 28](#_Toc45187560)

[一些问题 28](#_Toc45187561)

[根据什么来决定jvm堆空间大下 28](#_Toc45187562)

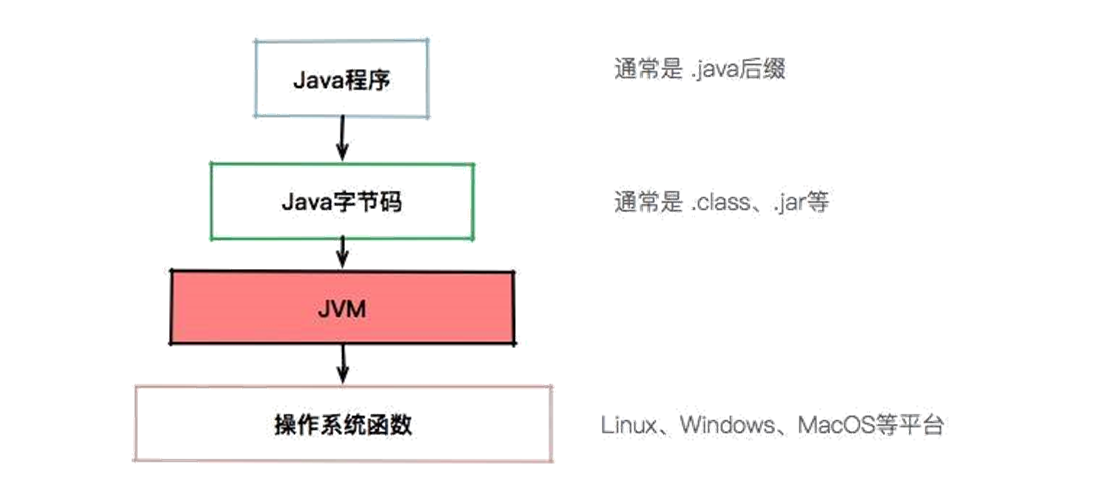
[根据什么来选择垃圾收集器 28](#_Toc45187563)

[JVM 为什么使用元空间替换了永久代 29](#_Toc45187564)

[大并发高流量场景下进行估算和调优 30](#_Toc45187565)

# JVM

JVM全称java virtual machine，java虚拟机，能够识别.class文件，并且能够解析它的指令，最终调用操作系统的函数，完成想要的操作；有了jvm，java就可以实现跨平台，jvm只需要保证能够正确的执行.class文件，就可以允许在诸如linux，windows，macos等平台上, JVM 与操作系统之间的关系：JVM 上承开发语言，下接操作系统，它的中间接口就是字节码。



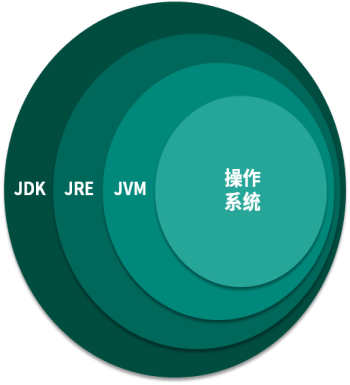
Javac将.java文件编译成class文件；

# Jvm，jre,jdk的关系

Jvm，jre,jdk的关系

Jvm是运行java程序的核心，但是仅仅只有jvm是不能实现一次编译到处运行的；还需要一个基本的类库，比如怎么操作文件，怎么连接网络等，java提供了jvm运行所需要的全部类库；所以jvm加上运行时需要的类库就组成的jre（java Runtime environment），要运行java程序只需要一个jre就可以了

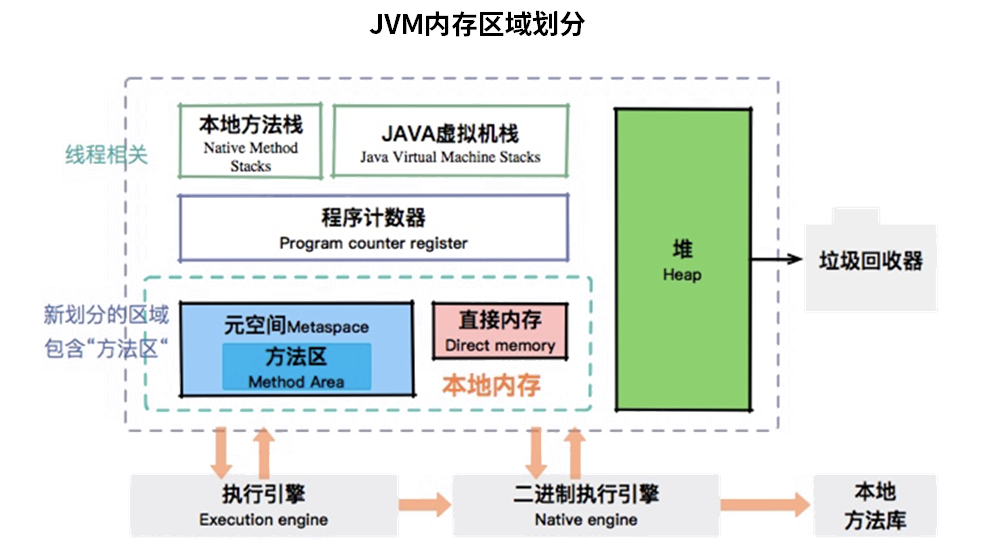
对于jdk（java development kit）,更加庞大些，他还包括了开发中一些工具，比较javac，java，jar;它们都是java开发的核心;这三者之间的关系，jdk > jre > jvm



# 运行时数据区

程序运行时数据存放的区域

Jvm在执行java程序时会把它所管理的区域划分为若干不同的数据区，运行时数据区也就是java虚拟机所管理的内存区域，下图为jdk1.8的内存区域



## 程序计数器

是一块较小的内存空间，作用是当前线程运行的字节码的行号指示器（存放执行字节码的行号）；字节码解释器就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支，循环，跳转，异常处理，线程恢复等功能都要依赖这个计数器来完成

Java多线程时通过通过cpu的分片机制来执行，在任意时刻，一个处理器只会执行一个线程，因此为了线程切换后能够恢复到正确的执行位置，每条线程都要有一个独立的程序计数器，各个线程之间互相不影响，独立存在，所以程序计数器是线程私有的；

当执行java方法时，计数器记录的是正在执行的字节码指令，如果执行的是native方法，这个计数器值为空，这个区域不会出现oom

如果我们的程序在线程之间进行切换，凭什么能够知道这个线程已经执行到什么地方？既然是线程，就代表它在获取CPU时间片上，是不可预知的，需要有一个地方，对线程正在运行的点位进行缓冲记录，以便在获取 CPU 时间片时能够快速恢复

## 虚拟机栈

虚拟机栈是线程私有的，它的生命周期和线程相同；

它描述了java方法执行的内存模型：每个方法执行的同时都会创建一个栈帧，用于存放局部变量表，操作数栈，动态链接，方法出口等信息；每个方法从调用到执行完成的过程，就对应着一个栈帧从入栈到出栈的过程

局部变量表：是一组变量值存储空间，用于存放方法参数和方法内定义的局部变量；局部变量表的容量以变量槽(Variable Slot)为最小单位，Java虚拟机规范并没有定义一个槽所应该占用内存空间的大小，但是规定了一个槽应该可以存放一个32位以内的数据类型。当存放double或者long类型时，会占用2个变量槽

在执行方法时。虚拟机使用局部变量表完成参数值到参数变量列表的传递过程，执行实例方法时，那么局部变量表中第0个索引的slot默认是用于传递方法所属对象的实例，在方法中可以使用this来访问这个隐式参数

操作数栈又称为操作栈，是一个后入先出的栈，在方法执行过程中各种字节码指令在操作数栈中写入和提取

动态链接：每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了方法调用过程中的动态链接

方法返回地址：方法执行后退出这个方法，有2中停止方式，1：执行引擎遇到任何一个方法返回的字节码指令（正常结束），2：在方法执行中遇到异常，并且这个异常没有在方法中处理

当线程请求的栈深度大于虚拟机所允许的深度，抛出stackoverflowerror异常

当栈动态扩展无法申请到足够的内存时，就会抛出outofmemoryerror异常

-Xss可设置栈的大小，-Xss规定了每个线程堆栈的大小。

## 本地方法栈

和java虚拟机栈相似，不过这是为执行本地方法服务的，java虚拟机栈是为java方法服务的

## 堆

堆是java虚拟机管理的一块最大的内存区域，堆使用来存放对象实例的，几乎所有的对象都是存放在这个区域的，堆是线程共享的

由于收集器基本上采用分代回收算法，所以堆还可以细分为：新生代和老年代；新生代又可以分为Eden区和from survivor区和to survivor区

-Xmx是指设定程序运行期间最大可占用的内存大小

-Xms 是指设定程序启动时占用内存大小

outofmemoryError：堆中没有内存空间分配对象，就会抛出这个异常

## 元空间（方法区）

Jdk1.8之前，是方法区，1.8之后元空间代替了方法区

方法区是线程共享的，用来存储被加载的类信息，常量，静态变量，即时编译后的代码等数据，方法区又称为“永久代；

运行时常量池是方法区的一部分，用于存放编译期生成的各种字面量和符号引用

方法区在jdk1.8中已经被废弃，取而代之的就是元空间（metaspace）,元空间使用操作系统的内存，jvm不会再出现方法区的内存溢出，但是无限使用会造成操作系统的崩溃，所以，一般也会使用参数 -XX:MetaspaceSize 和-XX:MaxMetaspaceSize 来控制大小。

## 直接内存

直接内存不是虚拟机运行时数据区的一部分，也不是java虚拟机规范定义的内存区域；

在jdk1.4中加入nio类，引入了一种基于通道和缓冲区的i/o方式，可以使用native方法直接分配堆外内存，然后通过一个存储在java堆中的directByteBuffer对象作为这块内存的引用进行操作

# Java对象

## 对象的创建

### 创建对象的方式

1. 使用new关键字创建
2. 使用反射创建
3. 使用克隆
4. 使用反序列化

## 使用new创建对象过程

首先检查这个指定的参数是否能在常量池中定位到一个类的符号引用，检查这个符号引用对应的类是否被加载，验证，准备，解析初始化过，，如果没有，那么先进行相应的类加载过程

在类加载完成后，然后虚拟机为新生的对象分配内存空间；对象所需要的内存大小在类加载后就已经确定了；分配内存空间就是将堆内存中划分一块对应大下的内存空间；

当java堆空间规整时，所有使用过的内存都放在一边，空闲的在另一边，中间放着一个指针作为分界点的指示器，把指针向空闲的内存区域移动一段和对象大小相等的距离，这种方式成为**指针碰撞**；当堆空间不规整时，虚拟机需要维护一个列表，记录那些内存空间可以使用，分配时从列表中选择一块相应大小的空间给对象，并且更新列表的记录，这种方式成为**空闲列表**

划分空间后，将分配到的内存空间都初始化为零值，保证对象实例字段在java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值

接下来就是对对象进行必要的设置，比如对象是哪个类的实例，如何才能找到类的元数据信息，对象的哈希码，对象的gc分代年龄等信息

执行<init>方法(执行构造方法)，把对象按程序员的意愿进行初始化，这样一个真正的对象就生成了

<init>方法

Init是对象的构造器方法，也就是说在程序进行new对象时调用该类的构造方法时才会执行init方法

<clinit>方法

初始化阶段是执行类构造器＜clinit＞（）方法的过程。

**＜clinit＞（）**方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值

public class Test{

static{

i=0；//给变量赋值可以正常编译通过

System.out.print（i）；//这句编译器会提示"非法向前引用"

}

static int i=1；

}

虚拟机会保证在子类的＜clinit＞（）方法执行之前，父类的＜clinit＞（）方法已经执行完毕。 因此在虚拟机中第一个被执行的＜clinit＞（）方法的类肯定是java.lang.Object。由于父类的＜clinit＞（）方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作

## 对象的内存布局

### 对象头

对象头包含3部分信息，第一部分存储对象自身的运行时数据，比如哈希码，gc分代年龄，锁状态标志，线程持有的锁，偏向线程id，偏向时间戳等，这部分称为mark word；另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针确定实例属于哪个类

还有一部分时，如果当前对象是数组，那么就存放数组的长度（对象是数组才会存在这个部分）

### 实例数据

对象真正存储的有效信息，也是在程序代码中定义的各种类型的字段内容，无论是从父类继承下来，还是子类中定义的，都记录下来

### 对齐填充

没有什么特殊意义，也没有特别的含义，只是起到了占位符的作用;对象大小的必须是8字节的整数倍，不够就需要对齐填充来补全

## OOM异常

### 起因

所以OOM到底是什么引起的呢？有几个原因：

内存的容量太小了，需要扩容，或者需要调整堆的空间。（-xms启动时占用的内存大小，-xmx最大的堆内存空间）

错误的引用方式，发生了内存泄漏。没有及时的切断与GCRoots的关系。比如线程池里的线程，在复用的情况下忘记清理 ThreadLocal 的内容。

接口没有进行范围校验，外部传参超出范围。比如数据库查询时的每页条数等。

对堆外内存无限制的使用。这种情况一旦发生更加严重，会造成操作系统内存耗尽。

排查oom

Jvm参数

-XX:+HeapDumpOnOutOfMemoryError dump出hprof文件，用eclipse memory analyzer查看分析，查找到那里的对象导致oom

-XX:HeapDumpPath=/tmp/logs 指定路径

Tomcat默认可以使用的内存为128MB，在较大型的应用项目中，这点内存是不够的，需要调大。

Windows下，在文件/bin/catalina.bat，Unix下，在文件/bin/catalina.sh的前面，增加如下设置：JAVA\_OPTS='-Xms【初始化内存大小】 -Xmx【可以使用的最大内存】'

### Java堆溢出

将-Xms和-Xmx设置相同可以避免堆自动扩展

-XX:+HeapDumpOnOutOfMemoryError dump出hprof文件，用eclipse memory analyzer查看分析，查找到那里的对象导致oom

### 虚拟机栈和本地方法栈溢出

-Xss参数设置栈容量，

栈中异常有2种

1. 如果线程请求的栈深度大于了虚拟机所允许的最大栈深度将会抛出stackoverflowerror异常
2. 如果虚拟机在扩展栈时无法申请到足够的内存空间，将会抛出OutOfMemoryError异常

情况1：使用-Xss参数减少栈内存容量，结果抛出stackoverflowerror，异常出现时输出的栈深度相应缩小

情况2：在方法中定义大量的本地变量，增大栈帧变量表的长度，结果抛出stackoverflowerror，异常出现时输出的栈深度相应缩小

**没有实验出抛出outofmemoryError**

### 方法区和运行时常量池内存溢出

-XX:PermSize

-XX:MaxPermSIze

-XX:PermSize=10M

-XX:MaxPermSIze=10M

String.intern()方法：如果字符串常量池中已经包含一个等于池string对象的字符串，则返回代表池中这个String的对象；否则将String对象包含的字符串添加的常量池中

（运行时产生大量的类来填满方法区，就会内存溢出）

动态生成class的应用需要注意类的回收状况：应用的场景：CGLib字节码增强，动态语言，大量jsp或者动态生成jsp文件的引用

# 类加载

把类的数据从class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成被虚拟机直接使用的java类型，这就是类加载机制

## 类加载过程

### 加载

通过全类名获取定义此类的二进制字节流

将这个字节流代表的静态存储结构转换为方法区的运行时数据结构

生成这个类的class对象，作为方法区这个类的各种数据的方位入口

Class对象封装了类在方法区内的数据结构。提供访问方法区的数据结构的接口，在java堆中生成一个代表这个类的java.lang.class的对象，作为方法区中这些数据的访问接口

### 验证

目的是确保加载的class文件的字节流中包含的信息符合当前虚拟机的要求，并不会危害虚拟机自身安全；

验证动作：

**文件格式验证**：是否以魔数开始，主次版本是否是当前虚拟机支持的版本

**元数据验证**：当前类是否有父类（除了Object类之外都要父类），这个类的父类是否继承了不被允许继承的类（被final修饰的类）;当前类不是抽象类，是否实现了其父类或者接口之中都要去实现的方法

**字节码验证**：通过数据流和控制流分析，确定程序语义是否合法，符号逻辑

保证跳转指令不会转移到方法体以外的字节码指令上

保证方法中的类转换是有效的，例如，可以把一个子类类型赋值个一个父类类型，不可用把父类类型赋值给子类数据类型，甚至对象赋值给一个毫不相关的类型

**符号引用验证:字符串全限定名是否能够找到对应的类，符号引用中的类，字段，方法的访问性是否可以被当前类访问**

**符号引用验证的目的是确保解析的动作能够正常进行**

### 准备

为类静态变量分配内存并设置类静态变量的初始值（也就是零值或者默认值）的阶段，这里只是初始化静态变量（被static修饰的变量），不包括实例变量；这里的初始值是指数据类型的零值，比如static int a=123;在准备阶段执行后a的值为0，不是123；把value赋值为123是在类构造器<clienct>()方法中，也就是说真正赋值是在初始化阶段，

### 解析

将常量池中的符号引用转换为直接引用的过程

**符号引用**：用一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用的时候无歧义的定位到目标即可

**直接引用**：直接指向目标的指针，相对偏移量或者一个能间接定位到目标的句柄

**解析阶段负责把整个类激活，串成一个可以找到彼此的网**，过程不可谓不重要。那这个阶段都做了哪些工作呢？大体可以分为：

类或接口的解析

类方法解析

接口方法解析

字段解析

### 初始化

初始化阶段根据程序员通常程序制定的主观计划去初始化列变量和其他资源，或者说初始化阶段是执行类构造器<clinit>()方法的过程

**＜clinit＞（）**方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值

public class Test{

static{

i=0；//给变量赋值可以正常编译通过

System.out.print（i）；//这句编译器会提示"非法向前引用"

}

static int i=1；

}

虚拟机会保证在子类的＜clinit＞（）方法执行之前，父类的＜clinit＞（）方法已经执行完毕。 因此在虚拟机中第一个被执行的＜clinit＞（）方法的类肯定是java.lang.Object。由于父类的＜clinit＞（）方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作

接口中不能使用静态语句块，但任然有变量初始化的赋值操作，所以接口中仍然有<clinit>()方法，和类不相同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>方法，只有在使用父接口的类变量时才执行，另外，接口的实现类在初始化是也不执行接口的<clinit>方法

在多线程同时去初始化一个类时只有一个线程能够执行类的clinit方法，其他线程阻塞知道执行完毕

## 类加载器

通过类的权限定名来获取描述类的二进制字节流，实现这个动作的代码称为“类加载器”

比较2个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义，如果两个类来源于同一个class文件，同一个虚拟机加载，但是是被不同的类加载器加载额，那么那他们就不相等

**Bootstrap classloader：启动类加载器**

这是加载器中的大 Boss，任何类的加载行为，都要经它过问。它的作用是加载核心类库，也就是 rt.jar、resources.jar、charsets.jar 等。

Java\_home中jre/lib/rt.jar中的class

当然这些 jar 包的路径是可以指定的，-Xbootclasspath 参数可以完成指定操作。

这个加载器是 C++ 编写的，随着 JVM 启动。

**Extention classloader：扩展类加载器**

扩展类加载器，主要用于加载 lib/ext 目录下的 jar 包和 .class 文件。同样的，通过系统变量 java.ext.dirs 可以指定这个目录. 这个加载器是个 Java 类，继承自 URLClassLoader。

Java\_home中的jre/lib/\*jar的class，

-Djava.ext.dirs指定目录下的jar包

**App classloader ： 应用程序加载器**

这是我们写的 Java 类的默认加载器，有时候也叫作 System ClassLoader。一般用来加载 classpath 下的其他所有 jar 包和 .class 文件

**Consum classloader：自定义加载器**

自定义类加载器，支持一个自定义的扩展功能

### 双亲委派模式

一个类加载器收到加载类的请求，首先不会自己去加载这个类，而是把这个请求交给父加载器去加载，每层加载器都是这样的，知道没有父类加载器；当父类加载器无法完成加载时才由自己来加载

一个类在同一个类加载器中具有唯一性，而不同的类加载器中是允许同名类存在的，这里是指全类名，但是在整个jvm中，即使全类名相同，类加载器不同，仍然不能算作是同一个类，无法通过instanceof，或者equals等方法校验

双亲委派模式优势

采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关**可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次**。其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类或者自定义Integer类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。

在遵循了双亲委派模型的规则之下，是不允许出现核心类库被替换或取代的可能，即不能在自己的 classpath 定义 java.lang.\*之类的Class去替换JRE 中的Class。

### 自定义类加载器

1.自定义加载器，重写findClass方法外还重写了loadClass方法，加载自定路径下的类

沿用双亲委派机制自定义类加载器很简单，只需继承ClassLoader类并重写findClass方法即可（也就是指定加载类的路径）。

### 打破双亲委派模式

通过findClass方法发现指定的类，但它除了重写findClass方法外还重写了loadClass方法，

默认的loadClass方法是实现了双亲委派机制的逻辑，即会先让父类加载器加载，当无法加载时才由自己加载。

这里为了破坏双亲委派机制必须重写loadClass方法，即这里先尝试交由System类加载器加载，加载失败才会由自己加载。

它并没有优先交给父类加载器，这就打破了双亲委派机制

使用spi机制

# 垃圾收集器

## 回收对象判断

### 引用计数法

引用计数法:给对象一个引用计数器，当有一个地方使用了对象时，计数器值就+1，当引用失效时，引用减一，当计数器值为0时对象就不可能在被使用，这个对象就可以被回收

这种方式存在一个难以解决的问题，就是对象之间的循环引用；

比如：一个类的中存在一个当前类的成员变量，初始化了2个对象，对象的这个变量指定对方，然后让这2个对象无任何引用（也就是将变量赋值为null），这2个对象不可能在被访问，但是它们互相引用者对方，导致它们的引用计数器不为0，所以无法被回收

Class Reference{

Reference r;

}

Reference a=new Reference();

Reference b=new Reference();

a.r=b

b.r=a;

a=null

b=null

### 可达性分析算法

可达性分析算法

以gcroot为起点，从这些节点开始向下搜索，搜索走过的路径称为引用链，当一个对象从到gcroot之间没有引用链时，这个对象是不可用的

Gc root对象：

虚拟机栈中引用的对象，

方法区中的静态属性引用的对象，

方法区中的常量引用的对象

### 引用

强引用

代码中普遍存在的形式：Object obj=new Object();

强引用Strongreferences当内存空间不足，系统撑不住了，JVM就会抛出OutOfMemoryError错误。即使程序会异常终止，这种对象也不会被回收。这种引用属于最普通最强硬的一种存在，只有在和 GC Roots 断绝关系时，才会被消灭掉。

软引用

用来描述一些还有用但是非必需的对象，对于软引用关联的对象，在系统将要发生内存溢出溢出之前，就会把这些对象进行回收，如果还没有足够的空间，那么才会出现内存溢出

将创建好的对象传给softreference

Object object=new Object();

SoftReference<Object>softRef=new SoftReference(object);通过get方法获取对象

弱引用

也是用来描述一些有用但是非必需的对象，只是他的强大比软引用更加弱些，当系统发送垃圾回收时，无论内存是否充足都会被回收

Object object=new Object();

WeakReference<Object>softRef=new WeakReference(object);

虚引用

一个对象是否有序引用存在，完全不会对其生存时间构成影响，也无法通过虚引用来获取实例对象。为对象设虚引用的目的是能在对象被收集器回收时收到一个系统通知

## 收集算法

### 发生回收的时候

GC是jvm自动完成的，根据jvm的系统环境而定的，所以时机是不确定的

可以使用system.gc进行手动回收，但是什么时候进行是由jvm来确定的，具体什么时候也是无法控制的，也就是说system.gc只是通知要回收，什么时候回收是由jvm来确定的，不建议使用这种方式进行gc，因为gc会消耗的资源比较大

当Eden区或者s区不够用，发送minor gc

当老年代不够用是发生major gc

当方法区不够用时

System.gc

### 标记-清除算法

算法分为标记和清除2个阶段：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

不足：一个是效率问题，标记和清除的效率都不高，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，碎片太多会造成之后分配大对象空间时无法找到足够大的连续空间而不得不触发一次gc

### 复制算法

将内存分为2块相同的区域，每次只使用其中的一块，当一块使用完成后就将还存活的对象移动到另一块，完后清理已使用过的内存空间，这样每次都回收一半的内存空间，就不会造成内存碎片的问题，实现简单高效；但是只使用一半的内存，太浪费资源了。

Hotspot虚拟机默认Eden区和survivor区的大小比例是8:1:1，也就是每次新生代可用空间为90%，只有10%浪费；当对象的存活率较高时就要进行较多的复制，效率将会变低

内存分配担保

因为无法保证每次回收时存活对象不超过10%，当survivor空间不够时，需要依赖其他的内存进行分配担保，这里的其他内存就是指老年代；如果另一个survivor区内存空间不足以存放回收时存活的对象，那么这些对象直接通过分配担保机制进入老年代

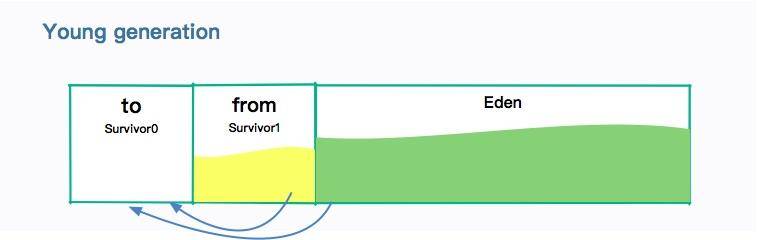
### 标记-整理算法

只要是用来**回收老年代**；标记出存活的对象，将存活的对象移动到一端，然后直接清理掉端边界另一边的对象

老年代存活率较高使用复制算法效率低

### 分代收集算法

根据对象存活的周期，将java堆分为新生代和老年代根据年代的特点选择不同的收集算法，在新生代中，每次gc时都会回收大批的对象，只有少量存活，使用复制算法；老年代对象存活率较高，没有额外的空间进行分配担保，就使用标记整理算法或者标记清除算法



年轻代分为：一个伊甸园空间（Eden ），两个幸存者空间（Survivor ）

当年轻代中的 Eden 区分配满的时候，就会触发年轻代的 GC（Minor GC）

在Eden区执行了第一次GC之后，存活的对象会被移动到其中一个Survivor分区（以下简称from）；

Eden区再次GC，这时会采用复制算法，将 Eden 和 from 区一起清理。存活的对象会被复制到 to 区；

接下来，只需要清空Eden和 from 区就可以了，然后将survivor的from和to进行切换

所以在这个过程中，总会有一个 Survivor 分区是空置的。Eden、from、to 的默认比例是 8:1:1，所以只会造成 10% 的空间浪费。

老年代一般使用 “标记-整理”算法，

因为老年代的对象存活率一般是比较高的，空间又比较大，拷贝起来并不划算，还不如采取就地收集的方式

对象进入老年代的途径

1. 升级

对象经过一次monitor gc，存活下来的对象年龄+1，当年龄到达设定的阈值后，就会升级为老年代

1. 分配担保

看一下年轻代的图，每次存活的对象，都会放入其中一个幸存区，这个区域默认的比例是 10%。但是我们无法保证每次存活的对象都小于 10%，当 Survivor 空间不够，就需要依赖其他内存（指老年代）进行分配担保。这个时候，对象也会直接在老年代上分配。

1. 大对象直接在老年代分配

超出某个大小的对象将直接在老年代分配。这个值是通过参数 -XX:PretenureSizeThreshold 进行配置的。默认为 0，意思是全部首选 Eden 区进行分配。

1. 动态对象年龄判断

有的垃圾回收算法，并不要求age必须达到15才能晋升到老年代，它会使用一些动态的计算方法。比如，如果幸存区中相同年龄对象大小的和，大于幸存区的一半，大于或等于age的对象将会直接进入老年代

Minor GC：发生在年轻代的 GC。

触发条件  
1、Eden区域满

2、新创建的对象大小 > Eden所剩空间

Major GC：发生在老年代的 GC。

Full GC：新生代和老年代，全堆垃圾回收。比如 Metaspace 区引起年轻代和老年代的回收。System.gc()

为什么需要survivor区，只有Eden区不行

新生代使用的是复制算法，需要一块存放存活对象的区域,如果没有survivor，那么在进行minor gc时，存活的对象只有进入老年代，这样老年代很快就会被填满，触发major gc,major gc一般会伴随着minor gc，也可以看作是一次full gc

老年代的内存空间远大于新生代，进行一次full gc消耗的时间比minor gc长得多，那么就会影响程序的运行效率

为什么需要2个survivor区域

解决磁盘碎片问题，首次minor gc时，使用一个survivor区，再次minor gc时，需要另一个survivor区来存放Eden区和当前survivor区存活的对象

为什么Eden，s1,s2的比例是8:1:1

统计出来的，

## 收集器种类

年轻代

### Serial收集器

单线程收集器，进行收集工作时，必须停止其他所有的工作线程，直到结束；stop the world, 这就会影响应用的响应时间

### ParNew收集器

ParNew 是 Serial 的多线程版本。由多条 GC 线程并行地进行垃圾清理。清理过程依然要停止用户线程。它能够和CMS收集器配合使用，在单线程环境中收集效果没有serial收集器好，因为多线程之间的存在交互的开销；

使用-XX:ParNewGC选项来指定ParNew为当前收集器，

### Parallel scavenge收集器

这是一个新生代收集器，它使用的是复制算法，也是并行的多线程收集器，看上去和parNew一样，但是parallel scavenger收集器更加关注系统的吞吐量

吞吐量：运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)

另一个多线程版本的收集器，它与ParNew收集器的主要区别：

Parallel Scavenge：追求 CPU 吞吐量，能够在较短时间内完成指定任务，适合没有交互的后台计算。弱交互强计算。

ParNew：追求降低用户停顿时间，适合交互式应用。强交互弱计算。

Parallel 收集器提供了2个参数来控制吞吐量：控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数和直接设置吞吐量大小的-XX:GCTimeRatio参数；

MaxPauseMillis运行的值是一个大于0的毫秒值，收集器尽可能的保证收集时间在设置值之内完成；并不是将这个值设置的越小收集速度就越快，GC停顿时间是牺牲吞吐量和新生代空间换取的；系统把新生代调小一些，收集300M新生代比收集500M快，这也导致了收集的频繁发生，原来10s收集一次，停顿100毫秒，线性变成5秒收集一次，停顿79毫秒，停顿时间是在减少，但是吞吐量也下降了；

### Serial old收集器

serial收集器的老年代版本，也是一个单线程收集器，使用标记-整理算法；这个收集器的意义主要是在client模式下给虚拟机使用；

### Parallel old收集器

parallel收集器的老年代版本，使用多线程和标记整理算法（也是关注的吞吐量）

### CMS收集器

是一个获取最短回收停顿时间为目标的收集器（响应速度快，系统停顿时间短），它在垃圾收集时使得用户线程和 GC 线程能够并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿；**使用标记清除算法**

**运作过程：**

1初始标记，仅仅是标记GC roots直接关联到的对象，速度很快（会暂停用户线程）

2并发标记，进行GC Roots 追踪，用于标记所有可达的对象

3重新标记，为并发标记期间因用户程序继续运行而导致标记发送变动的那部分对象进行标记，标记时间比初始时间长，但是比并发标记时间短（会暂停用户线程）

4并发清除

不建议在使用cms收集器的时候把堆空间设置过大，使用标记清除算法，效率较低（标记和清除）

初始标记和重新标记需要stop the world（也就是停止用户线程），耗时最长的并发标记和并发清除可以和用户线程一起工作，从总体上来看，CMS收集器回收过程与用户线程是并发执行的；

这个收集器的优点：并发收集，低停顿

缺点

1. 对cpu资源比较敏感:并发阶段，它虽然不会导致用户线程停顿，但是会占用一部分线程而导致应用程序变慢，总吞吐量降低
2. 无法处理浮动垃圾：由于CMS清理阶段用户线程正在运行，所有不断的产生新的垃圾对象，这部分出现在标记过后，就不会在本次垃圾收集被回收，只有在下次被回收，这部分垃圾对象称为浮动垃圾
3. 使用标记清除法：那么清理完成后就会出现不连续空间，不利于分配大对象空间；在没有足够大的连续空间来分配大对象是会触发full gc

### G1收集器

G1收集器已在JDK 1.7 u4版本正式投入使用。

特点

并行并发：利用多cpu，多核的环境优势，利用多cpu来缩短stop-the-world的时间；g1收集器可用在收集的同时让执行java程序继续执行

分代收集：能够处理新对象，已存活一段时间的对象，经过多次gc存活下来的对象

空间整合：整体采用标记整理算法，局部采用复制算法，保证内存的规则，不会有不连续的空间

可预测停顿：能够让使用者在明确的M毫秒内消耗在回收的时间不会超过N毫秒

G1收集器将java堆划分为多个大小相同的区域。虽然保留了老年代和新生代，但是新生代和老年代不在是隔离的，他们都是一块区域的集合；

g1收集器之所以能够可预测停顿时间，是因为它可以有计划的避免在整个java堆中进行全区域的收集；**g1收集器跟踪各个区域垃圾堆的大小，在后台维护一张优先级列表，每次根据允许的时间，优先回收价值最大的区域**；这种使用region划分内存空间及优先级的区域回收方式，保证了G1收集器在短时间内获取更高的收集效率

**怎么判断各个Region中的对象是否可达**

回收新生代是补不得不扫描老年代，Minor gc的效率将会减少；新生代和老年代之间的对象引用时，虚拟机使用remembered set来避免全堆扫描，这样就提示了gc的效率；

g1收集器也是使用remembered set来全region的扫描，每个region都有一个与之对应的Remembered Set，虚拟机发现程序对reference类型数据进行操作时，会产生一个write barrier暂时中断写操作，判断reference类型是否处于不同region之中，如果在不同的region中，就通过cardtable版相关的信息记录到Remember set中，在gc根节点的枚举范围中加入Remembered set即可保证不对全堆扫描也不会有遗漏；

如果不计算维护Remembered set的操作，g1收集器包含以下几个步骤：

初始标记：标记GC roots能够直接关联的对象，并且修改TAMS(Next top at Mark start)的值,让下一阶段用户程序并发运行时，能在正确可用的region中创建对象，这阶段需要停顿线程，但是时间很短；

并发标记：冲GC roots开始对堆中的对象进行可达性分析，找到存活的对象，耗时较长，可与用户程序并发执行

最终标记：修正并发标记期间因程序继续执行而产生标记发送变动的那部分标记记录；虚拟机将这段时间对象变化记录在线程remenber set logs里面；最终标记会把Remember set log中的数据合并到Remembered set中，这阶段需要停顿线程，但是可以并发执行，

筛选回收：首先对各个region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划，

### Gc配置参数

-XX:+UseSerialGC年轻代和老年代都用串行收集器

-XX:+UseParNewGC年轻代使用ParNew，老年代使用SerialOld

-XX:+UseParallelGC 年轻代使用 ParallerGC，老年代使用 Serial Old

-XX:+UseParallelOldGC 新生代和老年代都使用并行收集器

-XX:+UseConcMarkSweepGC，表示年轻代使用 ParNew，老年代的用 CMS

-XX:+UseG1GC 使用 G1垃圾回收器

-XX:+UseZGC 使用 ZGC 垃圾回收器

线上使用最多的垃圾回收器，就有 CMS 和 G1，以及 Java8 默认的 Parallel Scavenge

### Stop the world

为了保证程序不会乱套，最好的办法就是暂停用户的一切线程。也就是在这段时间，你是不能 new 对象的，只能等待。表现在 JVM 上就是短暂的卡顿，什么都干不了。这个头疼的现象，就叫作 Stop the world。简称 STW。

标记阶段，大多数是要 STW 的。如果不暂停用户进程，在标记对象的时候，有可能有其他用户线程会产生一些新的对象和引用，造成混乱

现在的垃圾回收器，都会尽量去减少这个过程。但即使是最先进的 ZGC，也会有短暂的 STW 过程。我们要做的就是在现有基础设施上，尽量减少 GC 停顿

JAVA\_OPTS="$JAVA\_OPTS

-XX:NewRatio=2

-XX:G1HeapRegionSize=8m

-XX:MetaspaceSize=256m

-XX:MaxMetaspaceSize=256m

-XX:MaxTenuringThreshold=10

-XX:+UseG1GC

 -XX:InitiatingHeapOccupancyPercent=45

-XX:MaxGCPauseMillis=200

-verbose:gc

 -XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-XX:+PrintReferenceGC

 -XX:+PrintAdaptiveSizePolicy

-XX:+UseGCLogFileRotation

-XX:NumberOfGCLogFiles=6

-XX:GCLogFileSize=32m

-Xloggc:./var/run/gc.log.$(date +%Y%m%d%H%M)

-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath=./var/run/heap-dump.hprof

-Dfile.encoding=UTF-8

-Dcom.sun.management.jmxremote

-Dcom.sun.management. jmxremote.port=${JMX\_PORT:-0}

-Dcom.sun.management.jmxremote.ssl=false

-Dcom.sun.management.jmxremote.authenticate=false"

如何确保新生代对象被老年代引用的时候不被gc？（查询老年代对象来确认对新生代对象的引用避免误回收）

机制：当老年代存活对象多时，每次minor gc查询老年代所有对象影响gc效率（因为gc stop-the-world），所以在老年代有一个write barrier（写屏障）来管理的card table（卡表），card table存放了所有老年代对象对新生代对象的引用。所以每次minor gc通过查询card table来避免查询整个老年代，以此来提高gc性能。

吞吐量和停顿时间

吞吐量：运行代码的时间/(运行用户代码时间+垃圾收集时间)

停顿时间：垃圾收集器进行垃圾回收终端应用执行响应的时间

停顿时间越短就越适合需要和用户交互的程序，也就是给用户的体验感越好

高吞吐量则可以高效的利用cpu时间，尽快完成程序计算任务主要适合后台运算而不需要太多的交互

如何选择合适的收集器

如果内存小于100m，使用串行收集器

如果是单核，并且没有对停顿时间要求，使用串行或jvm自己选

如果停顿时间超过1秒，选择并行或jvm自己选

如果响应时间最重要，并且超过1秒，使用并行收集器

G1收集器

Jdk7开始使用，jdk8非常成熟，jdk9默认收集器，适用于新老生代

G1中的remembered set

用来记录维护Region中对象的引用关系，类似hash结构，key记录region的地址，value表示引用该对象的集合，这样就可以知道该对象被那些对象引用了，从而不能回收

# 常用的jvm参数

Java -version

Java -help

Java -server

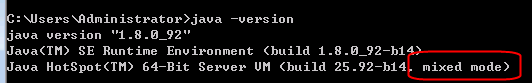
Java -cp

## -x参数

设置jvm的编译模式

默认mixed混合模式，jvm自己来决定

使用-Xmixed来设置（java -Xmixed -version）



解释执行-Xint

Java -Xint -version



-Xcomp 第一次使用就编译成本地代码

## -XX参数

Boolean类型

个数-XX:[+|-]<name> +或者-表示启用或者禁用name属性

比如-XX:UseG1GC 表示启用g1类型的垃圾处理器

非Boolean类型

个数-XX<name>=<value> 表示name属性的value值

比如-XX:MaxGCPauseMillis=500

其他类型

-Xmx (-Xmx100M等于-XX:MaxHeapSize=1000M)

-Xms （相当于-Xms1000M等于-XX:InitialHeapSize=1000M）

-Xmn

-XX:SurvivorRadio

查看参数

设置参数的方式

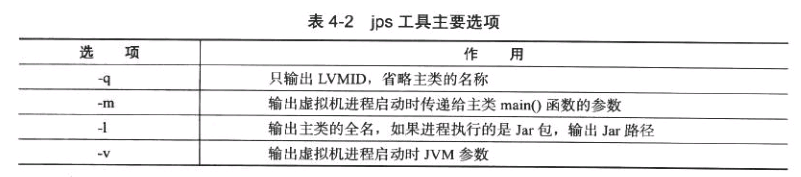
# 虚拟机性能监控

## jdk命令工具

### 查看进程jps

虚拟机进程状况工具；可以列出正在运行的虚拟机进程，并显示虚拟机执行的主类（main()方法所在的类）以及这些进程的**本地虚拟机唯一id**(和操作系统的进程id是一致的)；当启动多个虚拟机进程（也即是执行了多个类的main方法）无法根据进程名称定为（进程名都是java），这时就需要jps来显示主类来区分

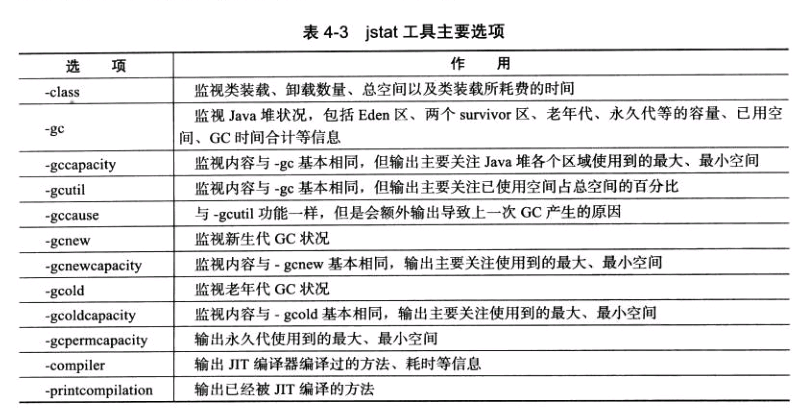
语法：jps [options] [hostid]



### 统计jvm运行状态信息jstat

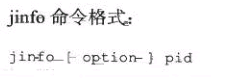
监控虚拟机各种运行状态信息的命令行工具；可以显示虚拟机进程中的类装载，内存，垃圾收集，jit编译等运行数据，

Jstat option vmid



### 查看jvm配置参数jinfo

实时查看和调用虚拟机各项参数，



### 堆快照生成命令jmap

生成堆转储快照，获取dump文件，查询finalize执行队列，查看堆和永久代的信息

### Jhat

和jmap配合使用，来分析jmap生成的堆转储快照，



当然还有其他跟专业的工具：比较eclipsememory analyzer

### 查询jvm当前的线程快照信息jstack

生成虚拟机当前时刻的线程快照，线程快照就是当前虚拟机内每一个正在执行的方法堆栈的集合

Java线程消耗过大的cpu

1使用top命令，展示出cpu消耗高的线程

2使用 shift+H切换到线程模式 找到执行cpu高的线程号pid

3使用Jstack pid > p.txt 用jstack dump 线程的日志

把线程号转换为16进制

根据线程id在dump出的文件中找到对应的线程日志

jstack Dump 日志文件中的线程状态

**dump 文件里，值得关注的线程状态有**：

死锁，Deadlock（重点关注）

执行中，Runnable

等待资源，Waiting on condition（重点关注）

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

暂停，Suspended

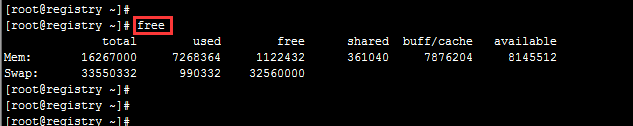
对象等待中，Object.wait() 或 TIMED\_WAITING

阻塞，Blocked（重点关注）

停止，Parked

Linux使用free命令查看实际内存占用

 默认情况下，即在没有选项的情况下，"free"命令显示内存的使用信息。默认按照k（b）的计数单位统计。



total：表示 总计物理内存的大小。

used：表示 已使用多少。

free：表示 可用内存多少。

Shared：表示多个进程共享的内存总额。

Buffers/cached：表示 磁盘缓存的大小。

## 可视化工具

### Jconsole

jconsole.exe

jdk自带的可视化监控工具，查看java程序的运行概况，监控堆信息，永久区使用情况，类加载信息

命令行中输入jconsole,弹出操作窗口

### Visualvm

jvisualvm.exe

目前jdk最强大的运行监控和故障处理程序

监控本地java进程

监控本地java进程的cpu，类，线程等

监控远端java进程

## 内存分析

使用工具Mat(也就是MemoryAnalyz er),分析内存泄露后的dump文件（.hprof）

获取dump文件

1. 手动

使用jmap命令，jmap -dump：format=b,file=head.hprof pid(java进程id)

1. 自动

配置jvm参数

-XX:+heapDumpOnoutofmemoryError -XX:HeapDumpPath=heap.hprof

Dump文件的信息

All objects

All class

Gc root

Thread stacks and local variables

## Gc日志分析

不同收集器的日志分析

## 执行引擎

Javac编译器将java文件编译成class文件，交给jvm运行，因为jvm只能识别class字节码文件，不同的操作系统安装对应版本的jdk,里面包含了各自屏蔽操作系统底层实现细节的jvm，同一份class文件就能在不同的操作系统上运行

Jvm最终需要将字节码转换为机器码，这样才能在机器上运行，字节码转换为机器码是谁来实现的：执行引擎

解释执行：解释器逐条把字节码翻译成机器码并执行，跨平台的保证；当程序需要快速启动和执行的时候，解释器首先发挥作用，省去了编译的时间，立即执行；使用解释器节约内存

即时编译器：当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“热点代码”，为提高热点代码的执行效率，虚拟机将字节码编译成对应平台的可执行机器码，并进行各层次的优化；程序运行后，把越来越多的代码编译成本地代码之后，可以更高效的执行；使用编译器提升效率；

Jvm中内置2个即时编译器，client和server（分别简称为c1编译器和c2编译器）；无论使用哪种编译器，在jvm中编译器和解释器都是混合使用的，也可以使用-Xint来强制使用解释执行模式，使用-Xcomp来取强制使用编译模式

编译器编译本地代码要占用程序的运行时间，要编译出优化程度高的代码，所有要花费更多的时间，想要编译出优化程度更高的代码，解释器还要题编译器收集性能监控信息，者对解释器的运行速度也有影响，为了在程序启动响应速度和运行效率之间到达最佳平衡，jvm启用了分层编译策略

分层编译：

第0层，程序解释执行时，解释器不开启性能监控，

第1层，c1编译，将字节码编译为本地代码，进行简单，可靠的优化如果有必要加入性能监控功能

第2层，c2编译，也是将字节码编译为本地代码，但会启用一些编译耗时较长的优化，甚至会根据性能监控进行一些较为激进的优化

# 类文件结构

class文件是一组8位字节为基础的二进制流，各项数据严格按照属性紧凑排列在class文件中，中间没有任何分隔符

class文件中的数据类型：无符号数和表

无符号数：u1 u2 u3 u4,它们分别表示1个字节，2个字节，4个字节，8个字节的无符号数

表：由无符号数或者其他表作为数据项构成的复合型数据类型，一般以\_info结尾

class文件的头4个字节称为魔数，cofebabe,作用是确定当前文件是否是能被虚拟机接收的class文件

紧接着魔数的是class五年的版本号，5,6字节为此版本号，7,8为主版本号

紧接着版本好之后的是常量池，常量池中的数量是不固定的，所以在常量池的入口需要放置一个u2类型的数据，

表示，常量池的容量的值（constant\_pool\_count）,容量计数是从1开始的，获取常量池容量后，要将值减1就是实际的常量

常量池存放两类常量：字面量，符号引用

字面量：文本字符串，final的常量

符号引用：类和接口的全限定名，字段的名称和描述符，方法名称和描述符

常量池每一项常量都是一个表，表名类似CONSTANT\_xx\_info

反编译后，根据常量类型，找到对应的常量类型表，根据表结构可以找到对应的

16进制数据，根据这个16十六进制找到对应的ascll码反推出java文件中的内容

访问标志

常量池结束后，紧接着的两个字节代表访问标志，用于识别类或者接口的访问信息，

比如：这个class是类还是接口，是否定义为public类型，是否定义为abstract类型，是否什么为final

字段表集合

用来描述接口或者类中声明的变量，字段包括类变量，实例变量，但不包括方法内部定义的局部变量

一个字段包含的信息：作用域（public，private，protected） 实例变量或者类变量修饰符（static）

可变性（final），并发可见性（volatile），是否序列化transient,字段数据类型，字段名称

方法表集合

方法的描述

### 编译器优化

公共子表达式消除

数组边界检查消除

方法内联

逃逸分析

## javac编译过程

### 解析和填充符号表过程

词法分析：将源代码的字符流转换为标记（Token）集合，标记是编译过程中最小的元素，关键字，变量名，字面量，运算符等都可以称为标记

语法分析：将token序列构造抽象语法树的过程，语法树是描述代码语法结构的树形表达式

填充符号表：符号表是一组符号地址和符号信息构成的表格，，符号表中记录的信息在编译的不同阶段都要用到，内容将用于语义检查和产生中间代码，

### 插入式注解处理器的注解处理过程

注解处理

### 语义分析与字节码生成过程

语义分析：获取抽象语法树之后无法保证获取到的源程序是符号逻辑的，需要对结构上正确的源程序进行上下文有关性质的审查，如进行类型审查（比如判断+操作的类型）

标注检查

数据以及控制流分析

解语法糖

生成字节码

## Java语法糖

语法糖不提供实质性的功能，但它们会提高效率，提升语法的严谨性，减少错误代码的机会

泛型与类型的擦除

泛型本质上是参数化类型的应用，也就是说操作的数据类型就被指定为一个参数，这种参数类型可以用在类，方法，接口上

在java语言中泛型只存在源码中，在编译为字节码后就已经替换为原来的原生类型，并在相应的地方插入了强制转型代码，因此对于运行期ArryList<Integer>和ArrayList<String>就是相同的类

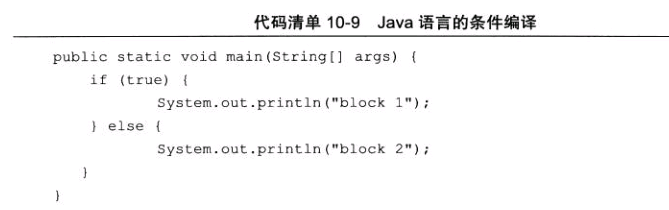
泛型技术就是java语法的语法糖，泛型实现类型称为类型搽除，基于这种方法实现的泛型称为伪泛型；定义了方向的代码编译成字节码文件后，再利用反编译工具进行反编译，泛型不见了；

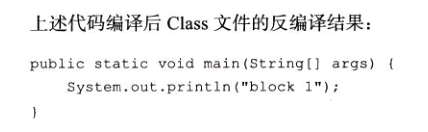
搽除只是对方法的code属性的字节码进行擦除，云数据信息中还是保留了泛型信息，因此可以通过反射获取参数化类型

自动装箱，拆箱，循环遍历

条件编译

针对一些条件判断，比如if





其他语法糖，比如内部类，枚举类，断言语句

# 性能优化

代码层面

非代码层面（内存，gc，cpu）

# 一些问题

## 根据什么来决定jvm堆空间大下

## 根据什么来选择垃圾收集器

Jdk 11版本开始使用g1收集器

1.7或者1.8的版本一般在cms和parallel scavenger之间选择

低延迟选择cms

高吞吐量选择parallel scavenge

Cms收集器的优点

缺点

1. 无法清除浮动垃圾，并发清理阶段用户线程还在运行，程序自然会有新的垃圾产生，那么CMS将无法在这次收集中处理掉它们，只有等到下次
2. 对CPU资源敏感。在并发阶段虽然不会导致用户线程停顿，但是会因为占用一部分线程（或者说CPU资源）而导致应用程序变慢，吞吐量降低，默认是启动（CPU数量＋3）／4的线程数。
3. CMS是基于标记－清除算法实现的，会产生大量的空间碎片

## JVM 为什么使用元空间替换了永久代

追求更低的停顿时间

判断一个垃圾收集器豪华的两个标准

吞吐量：运行用户代码时间/（运行用户代码时间+垃圾回收时间）

停顿时间：垃圾收集器进行垃圾回收中断应用执行响应的时间

Java程序小于100M,使用Serial收集器

应用程序运行在单核cpu中，使用Serial收集器

对停顿时间没有要求;parallel

串行收集器

适用于内存较小的嵌入式设备

并行收集器（吞吐量优先）

适用于科学计算，后台处理等交互场景

并发收集器（停顿时间优先）

用户线程和垃圾收集线程同时执行（但并不一定并行的），

Jdk标准参数：不会随着版本变化而变化

-x：非标准参数

修改jvm的运行方式

Java -Xint -version :解释执行 interpreted mode

Java -comp -version:编译执行 compiled mode

Mixed(解释+编译)

-XX非标准

-XX:[+\-]value

Java进程有多少参数

-XX:PrintFlagsFinal

参数在哪里设置

开发工具

## 大并发高流量场景下进行估算和调优

优化代码获得的性能提升远大于参数调整所获得的性能提升

在gc优化时考虑的三点：系统容量，吞吐量，延迟，

系统容量：系统的资源

吞吐量：在一段时间内完成了多少个事务操作，在一个时间内存完成了多少批量操作

延迟：响应能力；比如按钮触发时间有多快，需要多长时间返回一个网页，查询sql需要多长时间

选择垃圾收集器

1. 当堆大下不是很大是，选择串行收集器一般效率最高，-XX:+UseSerialGC
2. 单核cpu机器上，选择串行收集器比较合适
3. 如果是吞吐量优先，并且对响应时间没有什么特别的要求，选择串行收集器比较好， -XX:UseParallelGC
4. 如果对响应时间要求较高，想要较少的停顿时间，那么可以选择G1,ZGC,CMS都合理；这些收集器的gc停顿时间都比较短，它需要一些额外的资源去处理这些工作，通常吞吐量会低一些；-XX:UseConcMarkSweepGC,-XX:UseG1GC,-XX:UseZGC
5. 平常在web服务器，都是对响应时间要求非常高，所有在web应用中选择性其实就集中在g1,zgc,cms上



## 内存泄露和内存溢出

内存泄露：在申请内存后，无法释放申请到的内存空间，溢出内存泄露似乎不会有大的影响，但是内存泄露堆积后的后果就是内存溢出

内存溢出：在申请内存的时候，没有足够的内存提供申请者使用，或者锁给一个存储int类型数据的存储空间，但却用来存储long类型的数据，那么内存不够用，此时就会报错OOM，也就是内存溢出

### 二者的关系

1. 内存泄漏过多最终会导致内存溢出
2. 内存溢出就是你要的内存空间超过了系统实际剩余的空间
3. 内存泄漏是指向系统申请分配内存进行使用，但是用完了之后不归还，结果就是申请到的那块内存你自己也不能访问（也许是把地址弄丢了），系统就不能将它分配给其他的程序
4. 内存溢出：分配的内存不能够存放当前的数据

内存溢出的原因以及解决方法

产生原因

1. 内存汇总加载的数据量过大，比如一次从数据库取出过多数据
2. 集合类中有对象的引用，使用完之后为清理，使得jvm不能回收
3. 代码中存在死循环或循环产生过多重复的对象实体
4. 启动参数内存值设置过小

解决方法

1. 修改jvm参数，增加内存
2. 检查错误日志，查看oom前是否有其他的异常或错误
3. 对代码进行走查分析，找出可能发生内存溢出的位置

走查重点

1. 检查数据库查询时，是否一次查询大量的数据
2. 检查代码是否出现死循环或递归调用
3. 检查是否在大循环中生成新对象
4. 检查list,map等集合对象是否在使用完成后及时清除集合中的数据
5. 使用内存查看工具产看内存的使用情况