# Java文件到class文件

## 编译过程

.java—》词法分析器—》tokens流—》语法分析器—》语法树/抽象语法树—》语法分析器—》注解抽象语法树—》字节码生成器—》得到class文件

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

## 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

### 魔数

每个Class文件的头4个字节称为魔数（magic），它的唯一作用是判断该文件是否为一个能被虚拟机接受的Class文件。它的值固定为0xCAFEBABE。紧接着magic的4个字节存储的是Class文件的次版本号和主版本号，高版本的JDK能向下兼容低版本的Class文件，但不能运行更高版本的Class文件。

### 常量池

  major\_version之后是常量池（constant\_pool）的入口，它是Class文件中与其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项目之一。

**常量池中主要存放两大类常量：字面量和符号引用**。字面量比较接近于Java层面的常量概念，如文本字符串、被声明为final的常量值等。而符号引用总结起来则包括了下面三类常量：

类和接口的全限定名（即带有包名的Class名，如：org.lxh.test.TestClass）

字段的名称和描述符（private、static等描述符）

方法的名称和描述符（private、static等描述符）

虚拟机在加载Class文件时才会进行动态连接，也就是说，Class文件中不会保存各个方法和字段的最终内存布局信息，因此，这些字段和方法的符号引用不经过转换是无法直接被虚拟机使用的。**当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类加载过程中的解析阶段将其替换为直接引用，并翻译到具体的内存地址中**。

### 访问标记

  在常量池结束之后，紧接着的2个字节代表访问标志（access\_flag），这个标志用于识别一些类或接口层次的访问信息，包括：这个Class是类还是接口，是否定义为public类型，abstract类型，如果是类的话，是否声明为final，等等。每种访问信息都由一个十六进制的标志值表示，如果同时具有多种访问信息，则得到的标志值为这几种访问信息的标志值的逻辑或。

### 类索引，父类索引，接口索引

类索引（this\_class）和父类索引（super\_class）都是一个u2类型的数据，而接口索引集合（interfaces）则是一组u2类型的数据集合，**Class文件中由这三项数据来确定这个类的继承关系**。类索引、父类索引和接口索引集合都按照顺序排列在访问标志之后，类索引和父类索引两个u2类型的索引值表示，它们各自指向一个类型为COMNSTANT\_Class\_info的类描述符常量，通过该常量中的索引值找到定义在COMNSTANT\_Utf8\_info类型的常量中的全限定名字符串。而接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按implements语句（如果这个类本身是个接口，则应当是extend语句）后的接口顺序从左到右排列在接口的索引集合中。

### 字段表集合

 字段表（field\_info）**用于描述接口或类中声明的变量**。字段包括了类级变量或实例级变量，但不包括在方法内声明的变量。字段的名字、数据类型、修饰符等都是无法固定的，只能引用常量池中的常量来描述

### 方法表集合

方法里的Java代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为“Code”的属性里，关于属性表的项目

### 属性表集合

在Class文件、字段表、方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。，在Class文件、字段表、方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。

   下面简要说明下final、static、static final修饰的字段赋值的区别：

static修饰的字段在类加载过程中的准备阶段被初始化为0或null等默认值，而后在初始化阶段（触发类构造器<clinit>）才会被赋予代码中设定的值，如果没有设定值，那么它的值就为默认值。

final修饰的字段在运行时被初始化（可以直接赋值，也可以在实例构造器中赋值），一旦赋值便不可更改；

static final修饰的字段在Javac时生成ConstantValue属性，在类加载的准备阶段根据ConstantValue的值为该字段赋值，它没有默认值，必须显式地赋值，否则Javac时会报错。可以理解为在编译期即把结果放入了常量池中。

# Class文件加载到虚拟机中

## 装载

查找和导入class文件

1. 通过全类名获取定义此类的二进制字节流
2. 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
3. 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

## 连接

### 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

**为类的静态变量分配内存并设置类静态变量的初始值**（默认值），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。

在Class文件中它以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现；

比如org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language（假设是这个，当然实际中是由类似于CONSTANT\_Class\_info的常量来表示的）来表示Language类的地址。各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

符号引用总结起来则包括了下面三类常量：

类和接口的全限定名（即带有包名的Class名，如：org.lxh.test.TestClass）

字段的名称和描述符（private、static等描述符）

方法的名称和描述符（private、static等描述符）

**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（也就是对应在内存中的地址）

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

类或接口解析

字段解析

类方法解析

接口方法解析

## 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

在装载阶段中通过类的全限定名获取其定义的二进制字节流，这个过程需要类加载器来完成，类加载器就是用来加载class文件的

### 分类

**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

加载顺序：加载自顶向下，也就是由上层逐层尝试加载类

优势：Java类有了一种优先级的层次划分关系。比如Object类，这个毫无疑问应该交给最上层的加载器进行加载，即使是你覆盖了它，最终也是由系统默认的加载器进行加载的，因此Object类在各种类加载环境中都是同一个类。如果没有双亲委派模型，那么各个类加载器自取加载的话，就会出现很多个不同的 Object 类，应用程序会一片混乱。

打破双亲委派模式

继承classloader类，然后重写其中的loadclass方法

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

# 运行时数据区

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;

类文件被加载后，类中的内容（比如变量，常量，方法，对象等信息）需要存储，存储的就是jvm的运行时数据区

## 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。

## 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

## 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.

### 栈帧结构

栈帧是用于虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区的虚拟机栈的栈元素；栈帧存储了方法的局部变量，操作数栈，动态链接和方法返回地址等信息；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

**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；在Java程序被编译为Class文件时，就在方法的Code属性的max\_locals数据项中确定了方法所需要分配的最大局部变量表的容量。

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；

Java中用32位以内的数据类型有:boolean,byte,char,short,int,float,reference,

returnAddress八种类型

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程。如果是实例方法（非static的方法），那么局部变量表中第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字this来访问这个隐含的参数，其余参数则按照参数表的顺序来排列，暂用从1开始的局部变量Slot，参数表分配完毕后，在根据方法体内部定义的变量顺序和作用域分配其余的Slot。

#### 操作数栈

操作数栈(Operand Stack)也常称为操作栈，它是一个后入先出栈(LIFO)。同局部变量表一样，操作数栈的最大深度也在编译的时候写入到方法的Code属性的max\_stacks数据项中。

操作数栈的每一个元素可以是任意Java数据类型，32位的数据类型占一个栈容量，64位的数据类型占2个栈容量,且在方法执行的任意时刻，操作数栈的深度都不会超过max\_stacks中设置的最大值。

当一个方法刚刚开始执行时，其操作数栈是空的，随着方法执行和字节码指令的执行，会从局部变量表或对象实例的字段中复制常量或变量写入到操作数栈，再随着计算的进行将栈中元素出栈到局部变量表或者返回给方法调用者，也就是出栈/入栈操作。一个完整的方法执行期间往往包含多个这样出栈/入栈的过程。

#### 动态链接

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

在一个class文件中，一个方法要调用其他方法，需要将这些方法的符号引用转化为其在内存地址中的直接引用，而符号引用存在于方法区中的运行时常量池。

Java虚拟机栈中，每个栈帧都包含一个指向运行时常量池中该栈所属方法的符号引用，持有这个引用的目的是为了支持方法调用过程中的动态连接(Dynamic Linking)。

这些符号引用一部分会在类加载阶段或者第一次使用时就直接转化为直接引用，这类转化称为静态解析。另一部分将在每次运行期间转化为直接引用，这类转化称为动态连接。比如使用某个接口变量调用方法的时候，在加载阶段是不会知道接口对应的是哪个实现类实例，就需要在执行的时候来判断，所以就需要动态链接

#### 方法返回地址

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

当一个方法开始执行时，可能有两种方式退出该方法：

正常完成出口  
异常完成出口  
正常完成出口是指方法正常完成并退出，没有抛出任何异常(包括Java虚拟机异常以及执行时通过throw语句显示抛出的异常)。如果当前方法正常完成，则根据当前方法返回的字节码指令，这时有可能会有返回值传递给方法调用者(调用它的方法)，或者无返回值。具体是否有返回值以及返回值的数据类型将根据该方法返回的字节码指令确定。

异常完成出口是指方法执行过程中遇到异常，并且这个异常在方法体内部没有得到处理，导致方法退出。

无论是Java虚拟机抛出的异常还是代码中使用athrow指令产生的异常，只要在本方法的异常表中没有搜索到相应的异常处理器，就会导致方法退出。  
无论方法采用何种方式退出，**在方法退出后都需要返回到方法被调用的位置**，程序才能继续执行，方法返回时可能需要在当前栈帧中保存一些信息，用来帮他恢复它的上层方法执行状态。

方法退出过程实际上就等同于把当前栈帧出栈，因此退出可以执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值(如果有的话)压如调用者的操作数栈中，调整PC计数器的值以指向方法调用指令后的下一条指令。  
一般来说，方法正常退出时，调用者的PC计数值可以作为返回地址，栈帧中可能保存此计数值。而方法异常退出时，返回地址是通过异常处理器表确定的，栈帧中一般不会保存此部分信息。

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

## 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

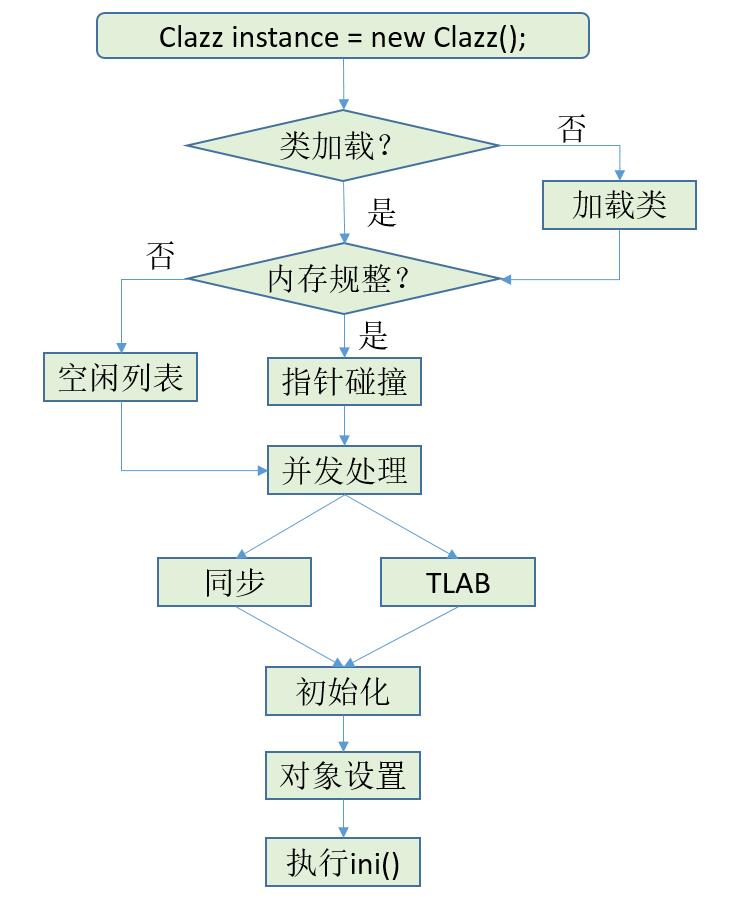
# Java中的对象

## 创建java对象

### 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创建对象的过程



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

1. 类加载检查

JVM遇到一条new指令时，首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类的加载过程。

1. 对象内存分配

对象所需内存的大小在类加载完成后便完全确定（对象内存布局），为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

根据Java堆中是否规整有两种内存的分配方式：（Java堆是否规整由所采用的垃圾收集器是否带有压缩整理功能决定）

指针碰撞(Bump the pointer)  
Java堆中的内存是规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，分配内存也就是把指针向空闲空间那边移动一段与内存大小相等的距离。例如：Serial、ParNew等收集器。

空闲列表(Free List)  
Java堆中的内存不是规整的，已使用的内存和空闲的内存相互交错，就没有办法简单的进行指针碰撞了。虚拟机必须维护一张列表，记录哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录。例如：CMS这种基于Mark-Sweep算法的收集器。

1. 并发处理

对象创建在虚拟机中时非常频繁的行为，即使是仅仅修改一个指针指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

同步  
虚拟机采用CAS配上失败重试的方式保证更新操作的原子性

本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）  
把内存分配的动作按照线程划分为在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存（TLAB）。哪个线程要分配内存，就在哪个线程的TLAB上分配。只有TLAB用完并分配新的TLAB时，才需要同步锁定。

1. 内存空间初始化

虚拟机将分配到的内存空间都初始化为零值（不包括对象头）,如果使用了TLAB，这一工作过程也可以提前至TLAB分配时进行。

内存空间初始化保证了对象的实例字段在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。

**注意：类的成员变量可以不显示地初始化（Java虚拟机都会先自动给它初始化为默认值）。方法中的局部变量如果只负责接收一个表达式的值，可以不初始化，但是参与运算和直接输出等其它情况的局部变量需要初始化。**

1. 对象设置

虚拟机对对象进行必要的设置，例如这个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。这些信息存放在对象的对象头之中。

1. 执行init()

在上面的工作都完成之后，从虚拟机的角度看，一个新的对象已经产生了。但是从Java程序的角度看，对象的创建才刚刚开始init()方法还没有执行，所有的字段都还是零。

所以，一般来说（由字节码中是否跟随invokespecial指令所决定），执行new指令之后会接着执行init()方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算产生出来。

<clinit>方法

**在类加载的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

一个java对象在内存中包括3个部分：对象头，实例数据和对齐填充



### 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

实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类中继承下来的，还是在子类中定义的，都需要记录下来。HotSpot虚拟机默认的分配策略为longs/doubles、ints、shorts/chars、bytes/booleans、oop，从分配策略中可以看出，相同宽度的字段总是分配到一起。

### 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

# GC

## 确定一个对象是垃圾

要进行垃圾回收，得先知道什么样的对象是垃圾

### 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

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

## 收集算法

### 标记-清除算法

算法分为标记和清除2个阶段：

标记：找出内存中需要回收的对象，并把他们标记出来，这个过程堆中所有的对象都会被扫一遍，从而才能确定需要回收的对象，比较耗时

清除：清除被标记需要回收的对象，释放内存空间

缺点

1. 标记和清除两个过程都比较耗时，效率都不高
2. 会产生大量的不连续的内存碎片，内存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进行另一次垃圾收集动作

### 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（young区）和老年代(old区)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并且存活较久，没有必要进行对象赋值，就使用标记整理算法或者标记清除算法

新生代分为：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

## 垃圾收集器

### Serial收集器

单线程收集器，使用一个cpu或者一个线程去收集线程去进行收集工作时，在收集的时候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

优点：简单高效，拥有很高的单线程收集线程

缺点：收集过程需要暂停所有线程

算法：复制算法

使用范围：新生代

应用：client模式下的默认新生代收集器

### 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，

优点：在多cpu时，比serial效率高

缺点：收集过程暂停所有的应用程序线程，单cpu时比serial效率差

算法：复制算法

使用范围：新生代

应用：运行在sever模式下的虚拟机中首选的新生代收集器

### 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。

### 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 old收集器是parallel scavenger收集器的老年代版本，使用多线程和标记整理算法

吞吐量优先

### 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

并发清除和并发标记，收集器线程可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行

优点：并发收集，低停顿

缺点：产生大量的内存碎片，并发阶段会降低吞吐量

### 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

在g1收集器时，java堆的内存布局与其他收集器有很大的差别，G1收集器将java堆划分为多个大小相同的独立区域。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，

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；

### 收集器分类

串行收集器：serial和serial old

并行收集器：（吞吐量优先）parallel scavenge 和parallel old，多条垃圾收集器并行工作，但此时用户线程仍处于等待状态。**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 cms和g1,用户线程和垃圾收集线程同时执行（ 但并不一定是并行的，可能是交替执行的），垃圾收集线程在执行的时候不会停顿用户线程的运行，**适用于相对时间有要求的场景，比如web**

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，停顿时间越短越适合需要和用户交互的程序，良好的响应速度能提升用户体验

吞吐量：运行用户代码时间/（运行用户代码时间+垃圾收集时间）；高吞吐量则可以高效的利用cpu时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互任务

### 如何选择合适的收集器

优先调整堆的大小让服务器自己来选择

如果内存小于100M，使用串行收集器

如果是单核，并且没有停顿时间要求，使用串行或jvm自己选

如果允许停顿时间超过1秒，选择并行或者jvm自己选

如果响应时间最重要，并且不能超过1秒，使用并行收集器

G1

G1 jdk1.7开始使用，jdk8成熟，jdk9默认垃圾收集器，适用于新老生代

判断使用使用g1收集器

1. 50%以上的堆被存活对象占用
2. 对象分配和晋级的速度变化非常大
3. 垃圾回收时间比较长

## Java对象引用类型

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

# JVM参数

## 参数类型

### 标准参数

-version

-help

-server

-cp

### -X参数

-Xint 解释执行

-Xcomp 第一次使用就编译成本本地代码

-Xmixed 混合模型，jvm自己来决定

### -XX参数

使用最多的参数类型，主要用于jvm调优和debug

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

### 其他参数

-Xms1000等价于-XX:InitialHeapsize=1000

-Xmx1000等于-XX:MaxHeapSize=1000

-Xss100等于-XX:ThreadStackSize=100

这块就相当于-XX类型的参数

## 查看参数

Java -XX:PrintFlagsFinal -version > flags.txt

## 设置参数

开发工具中设置，比如idea，eclipse

运行jar包的时候，java -XX:+UseG1GC xxx.jar

Web容器比如tomcat，可以在脚本中的进行设置

通过jinfo实时调整某个java进程的参数（参数只有被标记为manageable的flags可以实时修改）

## 参数含义

参数 含义 说明

-XX:CICompilerCount=3 最大并行编译数 如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加JVM崩溃的可能

-XX:InitialHeapSize=100M 初始化堆大小 简写-Xms100M

-XX:MaxHeapSize=100M 最大堆大小 简写-Xmx100M

-XX:NewSize=20M 设置年轻代的大小

-XX:MaxNewSize=50M 年轻代最大大小

-XX:OldSize=50M 设置老年代大小

-XX:MetaspaceSize=50M 设置方法区大小

-XX:MaxMetaspaceSize=50M 方法区最大大小

-XX:+UseParallelGC 使用UseParallelGC 新生代，吞吐量优先

-XX:+UseParallelOldGC 使用UseParallelOldGC 老年代，吞吐量优先

-XX:+UseConcMarkSweepGC 使用CMS 老年代，停顿时间优先

-XX:+UseG1GC 使用G1GC 新生代，老年代，停顿时间优先

-XX:NewRatio 新老生代的比值 比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5

-XX:SurvivorRatio 两个S区和Eden区的比值 比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8， 也就是一个S占整个新生代的1/10

-XX:+HeapDumpOnOutOfMemoryError 启动堆内存溢出打印 当JVM堆内存发生溢出时，也就是OOM，自动生成dump文件

-XX:HeapDumpPath=heap.hprof 指定堆内存溢出打印目录 表示在当前目录生成一个heap.hprof文件

XX:+PrintGCDetails - XX:+PrintGCTimeStamps - XX:+PrintGCDateStamps Xloggc:$CATALINA\_HOME/logs/gc.log 打印出GC日志 可以使用不同的垃圾收集器，对比查看GC情况

-Xss128k 设置每个线程的堆栈大小 经验值是3000-5000最佳

-XX:MaxTenuringThreshold=6 提升年老代的最大临界值 默认值为 15

G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表

-XX:InitiatingHeapOccupancyPercent 启动并发GC周期时堆内存使用占比 示”一直执行GC循环”. 默认值为 45.

-XX:G1HeapWastePercent 允许的浪费堆空间的占比 默认是10%，如果并发标记可回收的空间小于10%,则不会触发MixedGC。

-XX:MaxGCPauseMillis=200ms G1最大停顿时间 暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。 所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。

-XX:ConcGCThreads=n 并发垃圾收集器使用的线程数量 默认值随JVM运行的平台不同而不同

-XX:G1MixedGCLiveThresholdPercent=65 混合垃圾回收周期中要包括的旧区域设置占用率阈值 默认占用率为 65%

-XX:G1MixedGCCountTarget=8 设置标记周期完成后，对存活数据上限为G1MixedGCLIveThresholdPercent 的旧区域执行混合垃圾回收的目标次数 默认8次混合垃圾回收，混合回收的目标是要控制在

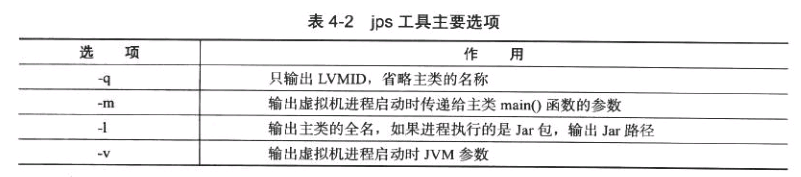
# 常用的命令

## jsp

Jps:查看java进程

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]

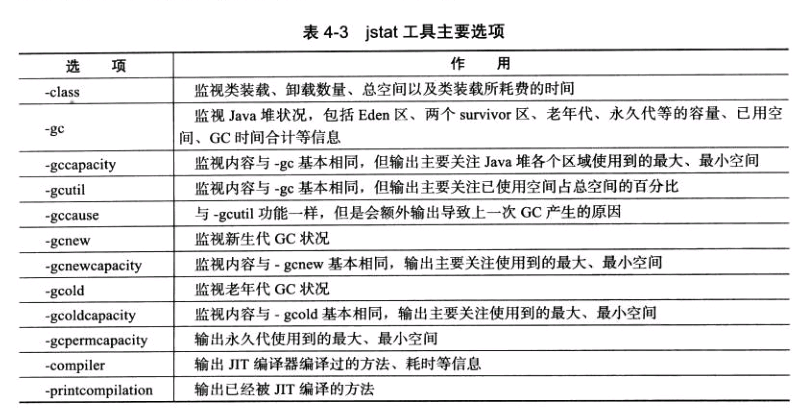


## jinfo

1. 实时查看和调整jvm配置参数
2. 查看 jinfo -flag name PID 查看某个java进程的name属性的值
3. 修改：参数只有被标记为manageable的flags可以被实时修改
4. 查看曾经赋值过值的一些参数：jinfo -flags PID

## jstat

1. 查看虚拟机性能统计信息
2. 查看类装载信息
3. 查看垃圾收集信息



## jstack

查看线程堆栈信息；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

用法 jstack PID

jstack Dump 日志文件中的线程状态

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

排查死锁案例

## jmap

生成堆转存储快照，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

1. 打印出堆内存相关信息 jmap -heap PID
2. Dump出堆内存相关信息 jmap -dump:format=b,file=heap.hprof PID
3. Dump发生堆内存溢出时候的日志

Jvm参数可以加上下面的参数，那当内存溢出的时候，会自动dump出该文件

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hpfrof

# 常用工具

## Jconsole

Jdk自带的可视化监控工具，查看java应用程序的运行状况，监控堆信息，永久区使用情况，类加载信息

在命令行中输入：jconsole

## Jvisualvm

可以监控本地java进程的cpu,类，线程等，直接启动即可

### 监控远端的java进程

监控远端tomcat过程

1. 在jvisualvm中选中远程，右击添加
2. 主机名上写服务器的ip地址，比如31.100.39.63，然后点击确定
3. 右击该主机31.100.39.63，添加JMX
4. 要想让服务器上的tomcat被连接，需要改一下bin/Catalina.sh这个文件
5. 在../conf文件中添加两个文件jmxremote.access和jmxremote.password

Jmxremote.access文件

Guest readonly

Manager readwrite

Jmxremote.password文件

Guest guest

Manager manager

授予权限：chmod 600 \*jmx remote

1. 启动tomcat，来bin目录，使用./startup.sh
2. 查看tomcat启动日志 tail -f ../logs/Catalina.out
3. 查看端口 lsof -I tcp:8080
4. 在刚才的jmx中输入8998端口，并且输入用户名和密码则登陆成功

## Arthas

Arthas支持JDK 6+，支持Linux/Mac/Windows，采用命令行交互模式，同时提供丰富的 Tab 自动补全功能，进一步方便进行问题的定位和诊断。是排查jvm相关问题的工具

当你遇到以下类似问题而束手无策时，Arthas可以帮助你解决：

1. 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception？
2. 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了？
3. 遇到问题无法在线上 debug，难道只能通过加日志再重新发布吗？
4. 线上遇到某个用户的数据处理有问题，但线上同样无法 debug，线下无法重现！
5. 是否有一个全局视角来查看系统的运行状况？
6. 有什么办法可以监控到JVM的实时运行状态？
7. 怎么快速定位应用的热点，生成火焰图？

### 常用命令

## Mat

Java堆分析器，用来查找内存泄露

Head dump称为堆转储文件，是java进程在某个时间内的快照

Dump文件包含的信息

All object

All classes

Garbage collection roots

Thread stacks and local variables

### 获取dump文件

手动

Jmap -dump：format=b,file=head.hprof 44808

自动

-xx:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof

Histogram

Histogram可以列出内存中的对象，对象的个数及其大小

Leak Suspects

查找并分析内存泄露的可能原因

Top Consume

列出大对象

# Gc优化

## Gc发生的时机

Gc是由jvm自动完成的，根据jvm环境而定的，所以时机不确定。也可以手动进行垃圾回收，比如调用System.gc()方法通知jvm进行一次垃圾回收，但是具体什么时刻运行也无法控制，也就是说System.ge()只是通知要回收，什么时候回收由jvm决定，不建议手动调用这个方法，因为消耗的资源比较大

垃圾回收的时机

1. Eden区或者s区不够用
2. 老年代空间不够用
3. 方法区空间不够用
4. System.gc()

## Gc日志文件分析

获取gc日志文件的参数配置

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Xloggc:gc.log

启动项目后，默认使用ParallelGC

### ParallelGC日志

吞吐量优先

[GC (Allocation Failure) [PSYoungGen: 127520K->20064K(149504K)] 127520K->121449K(491008K), 0.0136101 secs] [Times: user=0.06 sys=0.06, real=0.01 secs]

# GC / Full GC 表示 是 YGC 还是 Full GC；

# (Allocation Failure) 表示 触发GC的原因；

# PSYoungGen GC 发生的地方，指年轻代； 127520K->20064K(149504K) 分别指 GC回收前年轻代占用的空间，GC 回收后年轻代占用的空间；(149504K) 是 JVM分配年轻代总的空间；

# 127520K->121449K(491008K) 描述的是整个 heap 堆（新生代+老年代）， GC回收前占用的空间，GC 回收后占用的空间；(491008K) 是 JVM分配总的空间；

# 0.0136101 secs 指本次GC所消耗的时间；

# [Times: user=0.06 sys=0.06, real=0.01 secs] --- Linux 中 time ls 的结果；

### CMS日志

停顿时间优先

参数设置：-XX:+UseConcMarkSweepGC -Xloggc:cms-gc.log

### G1日志

参数设置：-XX:+UseG1GC -Xloggc:g1-gc.log

## G1调优和最佳指南

### 调优

是否选用g1收集器的判断依据：

1．50%以上的堆被存活对象占用

2．对象分配和晋升的速度变化非常大

3.垃圾回收的时间比较长

主要是通过吞吐量，垃圾回收停顿时间，回收次数来比较

1. G1收集器的参数配置：-XX:+UseG1GC

没有任何的参数特殊设置:只是设置了一下G1垃圾收集器，并通过下面的时间格式输出到文件中

 -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+UseG1GC -Xloggc:g1-gc.log

1. 调整内存大小 在获取日志分析

在步骤一的基础上加上-XX:MetaspaceSize=100M -Xms300M -Xmx300M参数



1. 调整最大停顿时间

-XX:MaxGCPauseMillis=20 设置最大停顿时间指标

把步骤二新增的参数去掉，加上-XX:MaxGCPauseMillis=20



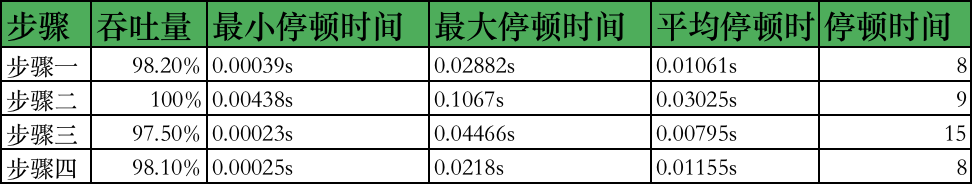
1. 启动并发gc时堆内存占用百分比

调整并发GC内存占用百分比 -XX:InitiatingHeapOccupancyPercent=45

XX:InitiatingHeapOccupancyPercent这个属性的意思是：当内存使用超过该比例时，进行清理,默认值是45;

如果值设置为0，那么G1收集器将一直执行GC操作；

还是一样，去调用步骤三的参数，设置-XX:InitiatingHeapOccupancyPercent=60



### 最佳指南

1. 不要手动设置新生代和老年代的大小，只要设置整个堆的大小

G1收集器在运行过程中，会自己调整新生代和老年代的大小 其实是通过adapt代的大小来调整对象晋升的速度和年龄，从而达到为收集器设置的暂停时间目标 如果手动设置了大小就意味着放弃了G1的自动调优

1. 不断调优暂停时间目标

一般情况下这个值设置到100ms或者200ms都是可以的(不同情况下会不一样)，但如果设置成50ms就不太合理。 暂停时间设置的太短，就会导致出现G1跟不上垃圾产生的速度。终退化成Full GC。所以对这个参数的调优是 一个持续的过程，逐步调整到佳状态。暂停时间只是一个目标，并不能总是得到满足。

1. 使用-XX:ConsGCThreads=n来增加标记线程的数量

IHOP如果阀值设置过高，可能会遇到转移失败的风险，比如对象进行转移时空间不足。如果阀值设置过低，就会 使标记周期运行过于频繁，并且有可能混合收集期回收不到空间。 IHOP值如果设置合理，但是在并发周期时间过长时，可以尝试增加并发线程数，调高ConcGCThreads。

1. MixedGC调优

-XX:InitiatingHeapOccupancyPercent

-XX:G1MixedGCLiveThresholdPercent

-XX:G1MixedGCCountTarger

-XX:G1OldCSetRegionThresholdPercent

Mix GC 混合回收，针对正常的新生代垃圾收集和部分老年代region进行回收。

调整IHOP阈值

参数调小，会提前触发mix gc，频繁进行并发收集会浪费CPU资源（gc没有垃圾可回收导致cpu做无用功）。

参数太高，导致转移空间不足，频繁发生Full gc。

// 设置并发GC时堆内存占用百分比阈值，默认45，当heap中占用超过45%，触发mix gc

-XX:InitiatingHeapOccupancyPercent=45

调整并发线程数

当mix gc周期过长时，可增加标记线程的数量提高效率，注意线程数量不是越多越好。

IHOP如果阀值设置过高，可能会遇到转移失败的风险，比如对象进行转移时空间不足。如果阀值设置过低，就会使标记周期运行过于频繁，并且有可能混合收集期回收不到空间。IHOP值如果设置合理，但是在并发周期时间过长时，可以尝试增加并发线程数，调高ConcGCThreads。

-XX:ConcGCThreads=n

调整混合收集次数

参数调小，会增加每次混合收集的region数量，导致stop-the-world时间增加 。

// 并发运行期间，最多经历几次混合收集周期

-XX:G1MixedGCCountTarger=8

调整存活对象对分区Region占比

// 设置存活对象对Region占比，默认85%

-XX:G1MixedGCLiveThresholdPercent=85

// 设置CSet堆的总大小占比，默认10%

1. 适当增加堆内存大小

# 高并发场景

# Jvm性能优化

## 发现问题

GC频繁

死锁

OOM

线程池不够用

CPU负载过高

## 排查问题

打印GC日志，查看minor gc、major gc,结合gc viewer/gceasy.io等工具分析gc情况

Jstack查看线程堆栈信息

Dump堆文件，使用mat或其他工具分析

使用jdk自带工具jconsole或者jvisualvm，或者其他工具分析

## 解决方案

适当增加堆内存大小，选择合适的垃圾收集器

# 常见问题

## 内存泄漏和内存溢出

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

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

## Young gc会有stw么（stop the world）

不管什么gc都会有stop-the-world,只是发生时间的长短

## Major gc和full gc的区别

Major gc指的是老年代的gc，full gc等于young+old+metaspace的gc

## G1和CMS的区别

CMS用于老年代的回收，gc用于老年代和新生代的回收

G1使用region方式对堆内存进行划分，且基于标记整理算法实现，整体减少了垃圾碎片的产生

## 什么是直接内存

直接内存是在java堆外内存，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，因此出于性能的考虑，读写频繁的场景可能会考虑使用直接内存

不可达的对象一定要被回收

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

## 方法区中的无用类回收

回收的类必须满足下面三个条件才能算是“无用的类”

1、该类所有的实例都已经被回收，也就是说Java堆中不存在该类的任何实例；

2、加载该类的ClassLoader已经被回收；

3、该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

## 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。

## 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

# 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