# 相关问题

## 堆和栈比较

-1）数据存放

栈中存放的是基本类型的变量or引用类型的变量

堆中存放的是对象or数组

-2）数据操作

当线程执行一个方法时，就会创建一个栈帧，并将该栈帧压栈；方法执行完后就将该栈帧出栈，并释放掉该栈的内存空间。

几乎所有的对象都是在堆中分配的，而该对象的引用却在栈中分配。即在堆中创建了该对象，在栈中创建一个指向该对象的指针（引用）。

-3）数据共享

栈的数据是线程私有的，堆中的数据是线程共享的。

## 数据在内存中存放

实例一：

int a=3; int b=3; 》》说明：编译器先处理“int a = 3；”，首先它会在栈中创建一个变量为a的引用，然后查找栈中是否有3这个值，如果没找到，就将3存放进来，然后将a指向3。接着处理“int b = 3；”，在创建完b的引用变量后，因为在栈中已经有3这个值，便将b 直接指向3。这样，就出现了a 与b 同时均指向3的情况。

实例二：

Object obj = new Object();

-1）执行“Object obj”：它会在Java栈的本地变量表中创建一个引用类型变量obj

-2）执行“new Object()”：它反映到Java堆中，形成一块存储了Object类型所有实例数据值(Instance Data,对象中各个实例字段的数据)的结构化内存，具体根据类型以及虚拟机实现的对象内存布局(Object Memory Layout)的不同，这块内存的长度时不固定的。另外，在Java堆中还必须包括能查找到此对象类型数据(如对象类型、父类、实现的接口、方法等)的地址信息，这些类型数据则存储在方法区中。

实例三：

Integer a=new Integer(3);//(1) Integer b=new Integer(3);//(2) 》》说明：在执行(1)时，首先在栈中创建一个变量a，然后在堆内存中实例化一个对象，并且将变量a指向这个实例化的对象。在执行(2)时，过程类似，此时，在堆内存中，会有两个Integer类型的对象。

实例四：

--1）创建两个类

public class Person {

private String name;

private int age;

//get/set方法(省略)

}

public class PersonTest {

public static void main(String[] args) {

Person person = new Person();

person.setName("testName");

System.out.println(person);

}

}

--2）流程描述：

-1）.java文件通过javac.exe编译成.class文件后，通过java.exe运行PersonTest.class，随后被加载到JVM中，方法区/元空间存储该类的信息（类的名称、方法信息、字段信息）；

-2）JVM找到PersonTest的主函数入口（main），为main函数创建栈帧，开始执行main函数；

-3）main函数的第一条命令Person person = new Person();就是让JVM创建一个Person对象，但是这时候方法区中没有Person类的信息，所以JVM马上加载Person类，把Person类的类型信息放到方法区/元空间中；

-4）加载完Person类之后，Java虚拟机做的第一件事情就是在堆区中为一个新的Person实例分配内存, 然后调用构造函数初始化Person实例，这个Person实例持有着指向方法区的Person类的类型信息（其中包含有方法表，java动态绑定的底层实现）的引用；

-5）当使用person.setName("testName");的时候，JVM根据person引用找到Person对象，然后根据person对象持有的引用定位到方法区中Person类的类型信息的方法表，获得setName()函数的字节码的地址；

-6）为setName()函数创建栈帧，开始运行setName()函数

实例五：

--1）Java代码：

String str1="droid";

String str2="droid";

String str3=new String("droid");

System.out.println(str1==str2); //true

System.out.println(str1==str3); //false

--2）流程描述：

-1）字面量创建形式 String str1="droid"; JVM检测这个字面量，这里我们认为没有内容为droid的对象存在。JVM通过字符串常量池查找不到内容为droid的字符串对象的引用，那么会创建这个字符串对象，然后将刚创建的对象的引用放入到字符串常量池中，并且将引用返回给变量str1；

-2）String str2="droid"; 同样JVM还是要检测这个字面量，JVM通过查找字符串常量池，发现内容为”droid”字符串对象的引用，于是将该引用返回给变量str2。注意这里不会重新创建新的字符串对象；

-3）因此能判断str1和str2是同一对象。

-4）String str3=new String("droid");当我们使用了new来构造字符串对象的时候，不管字符串常量池中有没有相同内容的对象的引用，新的字符串对象都会创建；

-5）因此能判断str1和str3不是同一对象；

注：对于通过new方式创建的String对象，每次都会在Heap上创建一个新的实例，但是对于字符串字面量的形式，只有当字符串常量池中不存在相同对象时才会创建。

http://blog.csdn.net/tianya3530/article/details/53912025 --后面部分没怎么看懂！！

http://blog.csdn.net/catoop/article/details/49903939 --数据存储例子！值得一看

## 数组在内存中存放原理

详情：https://www.2cto.com/kf/201801/715942.html

对于一个原始数据类型的数组，一般需要24字节的头信息[对象开销(16字节)+N(用于保存长度)(4字节)+填充字节(4字节)]再加上保存值所需的内存。

对于一维数组，以int[]型数组为例，假设数组长度为N，则内存占用(24+4N)个字节。即占用内存总量=头信息内存+数组N个int值占用内存。

多维数组实际存储的是一维数组，高维数组不断的降维转化为低维数组。

例如：一个M\*N的double类型的二维数组，其实际占用内存总量为：约等于8MN(8MN+32M+24)。

-1）这里以64位架构的计算机为例，每个对象引用地址为8位

-2）double[][] a = new double[M][N];

-3）一个一维数组存储M个一维数组引用，占用8个字节，总计字节为：(24+8M)

-4）有M个一维对象,每个对象占用内存为24+8N，总计字节为：M\*(24+8N)

-5）即二位数组总计为：24+8M+M\*(24+8N)

# JVM介绍

相关链接：

https://segmentfault.com/a/1190000015605327 ！！！

相关链接：

https://www.cnblogs.com/Qian123/p/5707562.html //1

<http://www.cnblogs.com/sunada2005/p/3577799.html> //2

<http://blog.csdn.net/ns_code/article/details/17565503>

http://www.cnblogs.com/gw811/archive/2012/10/18/2730117.html

http://blog.csdn.net/lizhongkaide/article/details/50701052 --未完待续

http://blog.csdn.net/bluetjs/article/details/52874852 --当时没懂

http://blog.csdn.net/zhaozheng7758/article/details/8623562 --没看懂

## 概述

JVM（Java Virtual Machine，Java虚拟机）：即在计算机上虚拟一个计算机，它存在于内存中。我们写的所有类、常量、变量、方法都在内存中，这就决定了我们程序运行是否健壮、高效。

计算机的基本组成：运算器、控制器、存储器、输入和输出设备。因此JVM也是这套元素，运算器是交给硬件CPU处理；为了适应“一次编译、随处运行”的情况，需要做一个翻译动作，于是就有了JVM自己的命令集。JVM的命令集则是可以到处运行的，因为JVM做了翻译，根据不同的CPU，翻译成不同的机器语言。

JVM自己的命令集与汇编的命令集优点类似，每一种汇编命令集针对一个系列的CPU，比如8086系列的汇编也是可以用在8088上的，但是却不能跑在8051上。

## JDK/JRE/JVM描述

-1）JDK（Java Development Kit，Java开发工具包）：

可以让开发者开发、编译、执行Java应用程序。

是完整的Java软件开发包，包括了JRE、编译器(javac.exe)和其它工具（比如，JavaDoc、Java调试器）。

每种操作系统的指令都是不同的，而JDK可以区分操作系统，只要你的本地系统装了JDK，这个JDK就能够和当前系统兼容的。

-2）JRE（Java Runtime Environment，Java运行时环境）：

是JDK的一部分，它包括Java虚拟机、Java核心类库和支持文件。

-3）JVM（Java Virtual Machine，Java虚拟机）：

是JRE的一部分，它是虚拟出来的计算机。

## JVM实现跨平台

实现跨平台原因：（一次编译、随处运行）

-1）通过javac.exe编译器编译我们的.java源文件后，生成.class文件；

-2）这些.class文件不能直接运行的，它运行在JVM之上，需要交给JVM来解析运行！

-3）JVM是运行在操作系统之上的，不同的操作系统有不同的JDK；

-4）因此不用关系class字节码是在哪个操作系统编译的，只要符合JVM规范，那么这个字节码文件就可运行！！

所以Java就做到了跨平台🡪一次编译，到处运行！

## 构成

Java内存模型：（4大部分）

1）类加载器(ClassLoader)

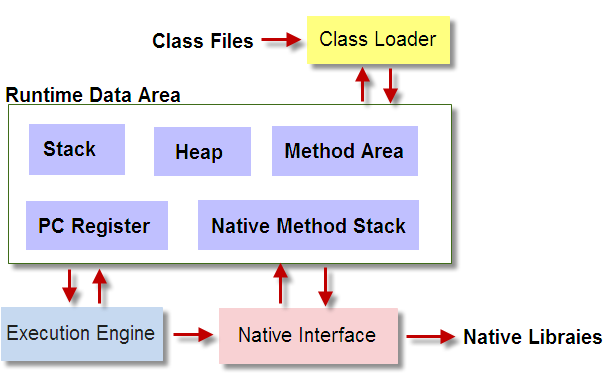
2）运行时数据区(Runtime Data Area)

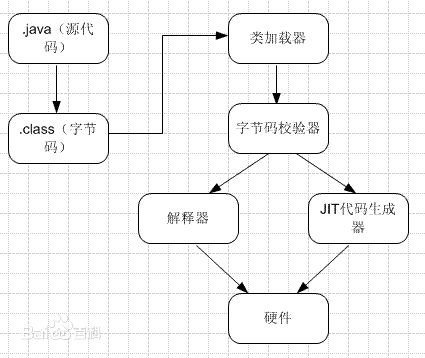
3）执行引擎(Execution Engine)

4）本地库接口(Native Interface)

其中，运行时数据区有5个区：堆(Heap)、方法区(Method Area)、程序计数器(PC Register Counter)、栈(Stack)、本地方法栈(Native Method Stack)

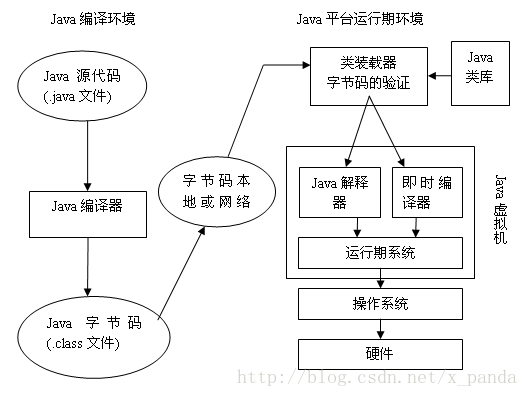
JVM的构成图：

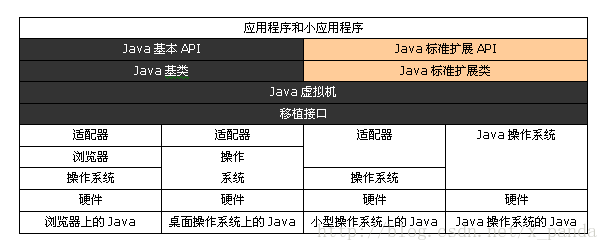




## JVM工作原理

Java代码（.java文件）被编译器(javac.exe)编译成Java字节码（.class文件），字节码被装入内存。一旦字节码进入虚拟机，它就会被Java解释器解释执行，或针对热点代码会被jit即时编译器转换成机器码执行。





Java平台由Java虚拟机和Java应用程序接口搭建，Java语言则是进入这个平台的通道，用Java语言编写并编译的程序可用运行在这个平台上。

# 类加载器(ClassLoader)

功能：

通过类全限定名获取类的二进制字节流

解析二进制字节流，获取Class类实例

加载classpath下的静态资源

获取类加载器：

ClassLoader loader = User.class.getClassLoader(); //User为一个类

ClassLoader parentloader = loader.getParent();

ClassLoader.getSystemClassLoader() //得到Application ClassLoader类加载器

InputStream input = ClassLoader.getSystemResourceAsStream("aa.properties"); //加载配置

## 概述

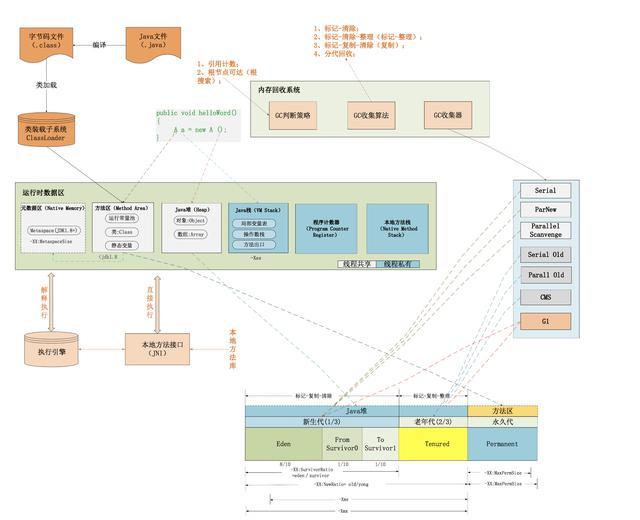
类加载器本身也是一个类，它的作用是把class文件从硬盘读取到内存（方法区/元空间）中。

Java源代码（.java文件）通过编译器(javac.exe)编译成字节码（.class文件）后，类加载器（ClassLoader）才能将字节码装入运行时数据区（<jdk1.8之前是装入方法区，>=jdk1.8之后是装入元数据区），才能被后续的Java运行程序（线程）正常使用（实例化/引用）。

类从被加载到虚拟机中开始，到卸载出内存为止它的生命周期包括7个阶段：加载，验证，准备，解析，初始化，使用，卸载。其中准备、验证、解析3个部分统称连接。

注：ClassLoader只管加载，只要符合文件结构就加载，至于说能不能运行，是由执行引擎(Execution Engine)负责的。

注：Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行。程序运行的基础类完全加载到JVM中，而其它类则在需要时才被加载。这也是为了节省内存开销！



## 加载时机

虚拟机规范是严格规定有且只有5种情况会立即对类进行“初始化”(class文件加载到JVM中)：

-1）创建类的实例(new的方式)。访问某个类或接口的静态变量，或者对该静态变量赋值，调用类的静态方法。

-2）反射

-3）初始化某个类的子类，该类也会被初始化

-4）Java虚拟机启动时被标明为启动类的类，直接使用java.exe命令来运行某个主类（包含main方法的那个类）

-5）当使用JDK1.7的动态语言支持时(…)

## 三种类加载器

-1）启动类加载器（Bootstrap ClassLoader）：负责将$JAVA\_HOME\lib目录中的类库加载到虚拟机中，如rt.jar。

-2）扩展类加载器（Extension ClassLoader）：负责加载$JAVA\_HOME\lib\ext目录、或-Djava.ext.dirs指定目录下的扩展类

-3）应用程序类加载器（Application ClassLoader）：默认，负责加载classpath路径下的类

注：Java在逻辑上并不存在Bootstrap ClassLoader的实体！因为它是用C++编写的，所以打印其内容将会得到null。

问：为什么要自定义类加载器？

我们需要的类不一定都在classPathl路径下，对于自定义路径中的class类文件的加载，我们需要实现自定义类加载器。

## 双亲委托机制

JVM加载类的实现方式，我们称为“双亲委托机制”。

工作原理：当一个类加载器收到类加载请求，它不会立即加载这个类，而是把该请求委托给自己的父加载器；每个类加载器都如此，因此所有的类加载请求最终都会传送到顶层的Bootstrap ClassLoader中；只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

原因：保证基础类的加载，优先使用系统的类。防止自定义的一些跟jdk标准库中冲突的全限定名的类被加载，导致标准库函数不可用。

设计思想：之所以是这种机制，还有一个安全上的考虑：若某人将一个恶意的基础类加载到jvm，委托机制会交给其父加载器去加载，由于父加载器能找到基础类，自然就不会将该恶意的基础类加载进来。

例如：假设有一个开发者自己编写了一个名为Java.lang.Object的类，想借此欺骗JVM。现在他要使用自定义ClassLoader来加载自己编写的java.lang.Object类。然而幸运的是，双亲委托模型不会让他成功。因为JVM会优先在Bootstrap ClassLoader的路径下找到java.lang.Object类，并载入它。

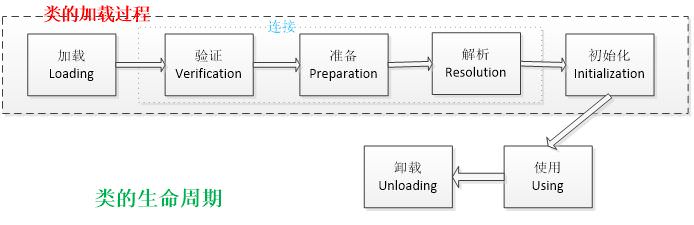
## 类加载方式

-1）隐式加载：通过new等方式生成对象时，隐式调用类加载器加载对应的类到JVM中

-2）显示加载：反射，通过class.forName()等方式，显式加载需要的类。

## 类加载详细过程

类Class从被加载到jvm开始、到卸载出内存的生命周期：



说明：

-1）一个class字节码文件变成加载到内存中变成能够被使用的类，需经过加载、连接（包括验证、准备、解析3个部分）、初始化三大步骤。

-2）以上过程中，加载、验证、准备、初始化、卸载5个阶段的顺序都是确定的。而解析阶段则不一定，为了支持Java语言的运行时绑定，解析过程在某些情况下可以初始化阶段之后再开始。

在Java中是否所有类型在类加载过程中都需要经过这几个步骤？

-1）Java语言的类型可以分为两大类：基本类型、引用类型；

-2）基本类型是由虚拟机预先定义好的，所以不会经历单独的类加载过程；

-3）引用类型又分为四种：类、接口、数组类、泛型参数。由于泛型参数会在编译过程中被擦除，所以Java中只有类、接口、数组类三种类型需要经历JVM对其连接和初始化过程。

附：三种类型中，数组类是由JVM直接生成的，类和接口则有对应的字节流，字节流最常见形式就是由编译器生成的class文件。

### 加载（Loading）

查找字节流并据此创建类的过程。（参考java.lang.ClassLoader的loadClass()方法）

加载阶段完成后，jvm外部的二进制字节流就按照虚拟机所需要的格式存储在方法区（jdk.18之前）或元数据区（jdk1.8之后）的内存中。

实际上，加载阶段虚拟机完成以下三件事：

-1）通过一个类的全限定名来获取定义此类的二进制字节流；

-2）将这个字节流所代表的静态存储结构转换为方法区（jdk.18之前）或元数据区（jdk1.8之后）的运行时数据结构。

-3）在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

附：全限定类名，如java.lang包中定义的Object类的全限定类名为java.lang.Object

### 验证（Verification）

验证是连接阶段的第一步，这一阶段目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。

该阶段完成4个检验工作：文件格式校验、元数据验证、字节码验证、符号引用验证

### 准备（Preparation）

该阶段为类变量（static变量）分配内存并将其初始化为默认值。

这些变量使用的内存都在方法区（jdk1.8之前）或元数据区（jdk1.8之后）中分配。

这时候进行分配的仅包括类变量，而不包括实例变量（实例变量将会在对象实例化时随着对象一起分配到Java堆中）

注：这里的初始值通常情况下是数据类型的零值，如“publicstaticintvalue=123;”那变量value在准备阶段过后的初始值为0而不是123，把value赋值为123的动作将在初始化阶段执行。

### 解析（Resolution）

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

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7类符号引用进行。

### 初始化（initialization）

该阶段是为初始化静态变量、静态代码块。

这样的过程在程序调用类的静态成员的时候开始执行，所以静态方法main()才会成为一般程序的入口方法。类的构造器也会引发该动作。

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正执行类中定义的Java程序代码。在准备阶段，变量已经经过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序指定的主观计划去初始化类变量和其它资源，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{}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，但是不能访问。

## 测试实例

--使用类加载器加载property文件

Properties property = new Properties();

InputStream inputStream = ClassLoader.getSystemResourceAsStream("db.properties");

property.load(inputStream);

Set<String> propertyNames = property.stringPropertyNames(); //遍历一

for(String name : propertyNames) {

property.getProperty(name);

}

Map<String, String> map = new HashMap<String, String>((Map)property); //遍历二

for(Map.Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

# 执行引擎与JIT即时编译器

## 执行引擎(Execution Engine)

执行引擎，也叫解释器(Interpreter)，负责解释命令，提交操作系统执行。

Java解释器工作时就是通过改变当前线程的程序计数器的值选取下一条字节码指令，任何分支、循环、方法调用、判断、异常处理、线程等待以及恢复线程、递归等等都是通过这个计数器来完成的。

执行字节码，或者执行本地方法。

## JIT即时编译器

我们通常会想：JVM在加载了这些class文件以后，针对这些字节码，逐条取出，逐条执行🡪解析器（Interpreper）解析。但这样会太慢了！！

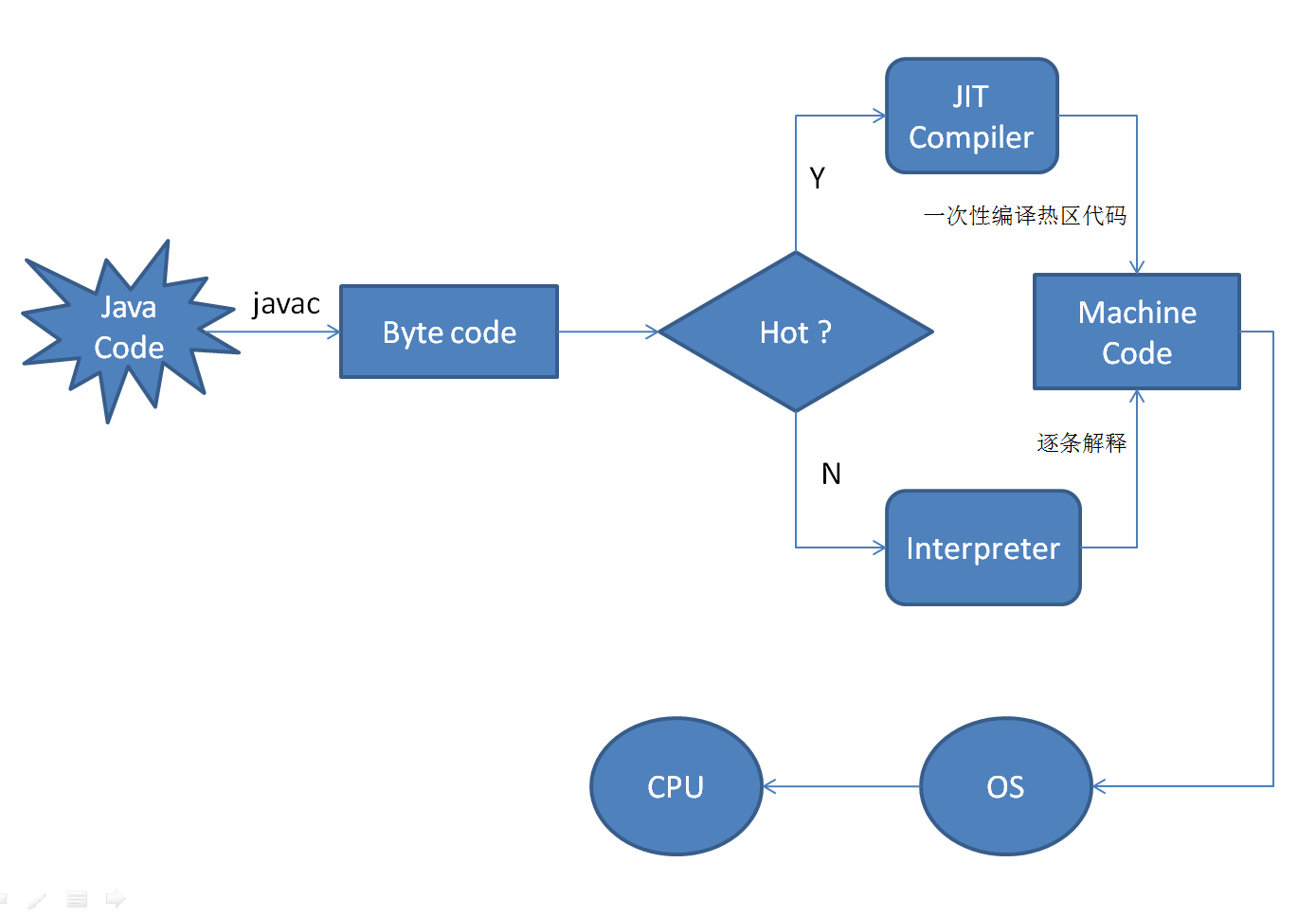
JVM的实现：

-1）将这些Java字节码重新编译优化，生成机器码，让CPU直接执行。这样编出来的代码效率会更高；

-2）编译也是要花费时间的，我们一般对热点代码做编译，非热点代码直接解析就好了。

热点代码解释：一、多次调用的方法。二、多次执行的循环体

使用热点探测来检测是否为热点代码：方式：1.采样 2.计数器



# 本地接口(Native Interface)

本地接口的作用是融合不同的编译语言为Java所用，它的初衷是融合C/C++程序，Java诞生的时候是C/C++横行的时候，要想立足，必须有一个聪明的、睿智的调用C/C++ 程序，于是就在内存中专门开辟了一块区域处理标记为native的代码，它的具体做法是Native Method Stack 中登记native方法，在Execution Engine执行时加载native libraies。

# 运行时数据区(Runtime Data Area)

运行数据区是整个JVM 的重点。我们所有写的程序都被加载到这里，之后才开始运行，Java生态系统如此的繁荣，得益于该区域的优良自治。

整个JVM框架由加载器加载文件，然后执行器在内存中处理数据，需要与异构系统交互是可以通过本地接口进行，瞧，一个完整的系统诞生了！

## 分类

### 5大区域

JVM将运行时数据区划分为5个区域：

-1）方法区(Method Area)：

线程共享，JVM启动后立即创建。

用于存储已被虚拟机加载的类信息，包括：静态变量（static）、常量（final static）、字节码（解释器解释所得）、机器码（jit即时编译器编译所得）

-2）堆(Heap)：

线程共享，JVM启动后立即创建。

用来存放new出来的对象实例和数组，是垃圾回收的主要场所

3）程序计数器(Program Counter Register)：

线程私有，用于存放下一条指令所在单元的地址。jvm就是通过读取程序计数器的值来决定下一条需要执行的字节码指令，进而进行选择语句、循环、异常处理等。

4）虚拟机栈/Java方法栈(VM Stack)：

线程私有，用来描述Java方法的执行。

存放局部变量、基本类型数据、及堆内存中某对象的引用。

5）本地方法栈(Native Method Stack)：

线程私有，用于提供native方法的使用。

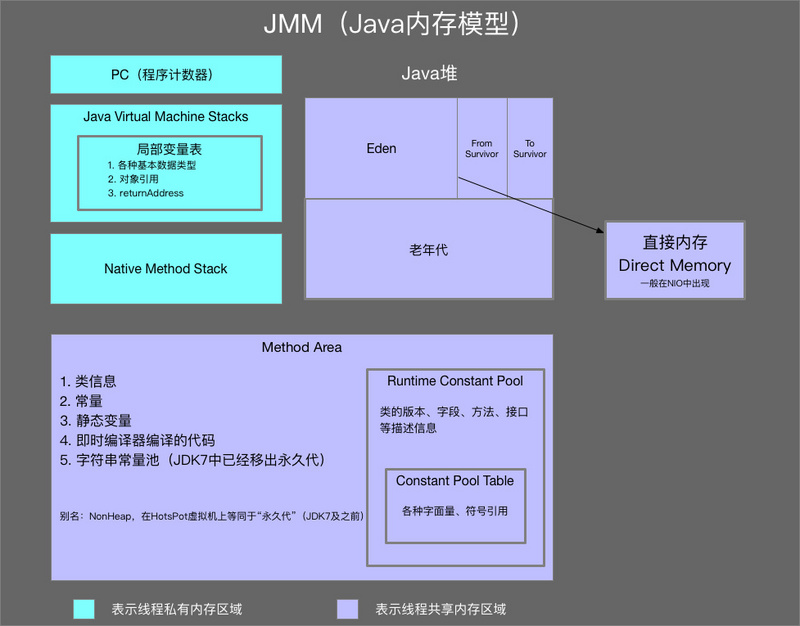
### 线程共享和私有的原因：

方法区和堆是线程共享，其它三个是线程私有的。

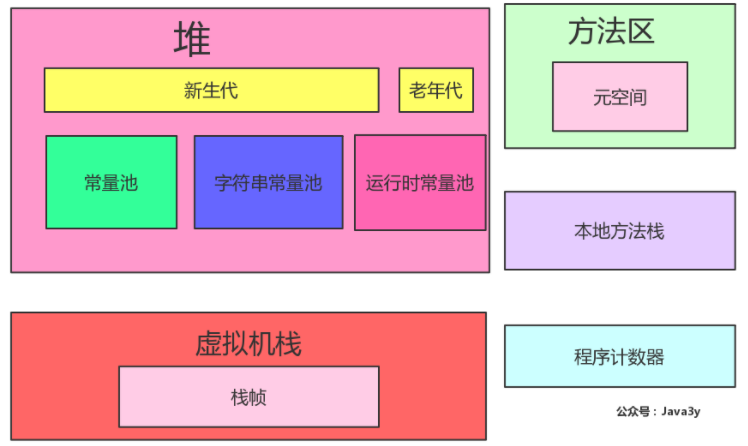
一个Java源程序文件会被编译成class字节码文件，每个Java程序都需运行在自己的JVM上，然后告知JVM程序的运行入口，再被JVM通过java解释器加载运行。

概括起来，JVM初始运行时都会分配好Method Area（方法区）和Heap（堆），而JVM每创建一个线程，就为其分配一个Program Counter Register（程序计数器）、VM Stack（虚拟机栈）、Native Method Stack（本地方法栈），当线程终止时这三者（程序计数器、虚拟机栈、本地方法栈）所占用的内存空间都会被释放掉。

JMM（Java内存模型）：



基于JDK1.8的Java内存模型：



## PC(Program Counter Register，程序计数器)

线程私有，用于存放下一条指令所在单元的地址，如果执行Native方法则为空（Undefined）。

线程私有原因：由于Java多线程通过交替线程轮流切换并分配处理时间的方式来实现，而同一时间内处理器的一个内核只会执行一条线程中的指令，因此需要每个线程都有一个独立的程序计数器来记录当前指令的行号。

Java解释器工作时就是通过改变当前线程的程序计数器的值选取下一条字节码指令，任何分支、循环、方法调用、判断、异常处理、线程等待以及恢复线程、递归等等都是通过这个计数器来完成的。

该区域唯一不存在OutOfMemoryError情况。

## 虚拟机栈/Java方法栈(VM Stack)

### 概述

虚拟机栈描述：

线程私有，用于描述Java方法的执行，每个方法在执行的同时都会创建一个栈帧。

栈的内存分配是在程序运行时进行的，但其所需的内存空间是在编译期完成的，大小完全确定。

栈帧描述：

栈由许多帧组成。

每个方法被执行时都会创建一个栈帧(Stack Frame)，用于存储该方法的局部变量表、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。

JVM栈只进行两种操作：以帧为单位的压栈和出栈操作。

-1）当线程执行一个方法时，就会创建一个栈帧，并将该栈帧压栈；

-2）方法执行完后就将该栈帧出栈，并释放掉该栈的内存空间。

局部变量表描述：

虚拟机栈中有一个局部变量表，存放了编译期可知的各种基本数据类型、对象引用和returnAddress类型。

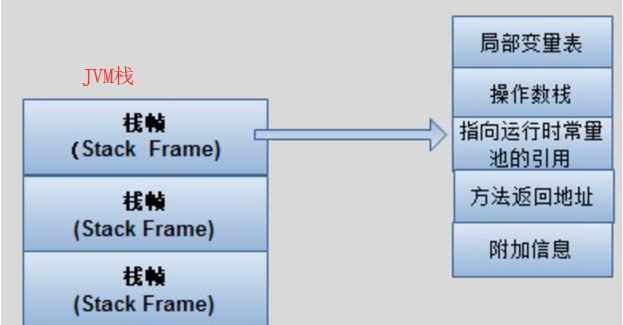
8个基本数据类型：byte、char、int、short、long、float、double、boolean；

对象引用（reference类型）：它不等同于对象本身（对象存放在堆区中），可能是一个指向对象起始地址的指针，也可能是指向一个代表对象的句柄；

returnAddress类型：指向了一条字节码指令的地址；

因此java中基本类型的变量是值传递，而非基本类型的变量是引用传递。

### 存放的类型：



详细描述：

1）局部变量表：

是一个以字为单位的数组，每个数组元素对应一个局部变量的值。调用方法时将方法的局部变量组成一个数组，通过索引来访问。若为非静态方法，则加入一个隐含的引用参数this，该参数指向调用这个方法的对象。

32位变量槽，存放各种基本数据类型、方法的形式参数、引用对象的地址（reference类型）和returnAddress类型（指向了一条字节码指令的地址）。

64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1个。

局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量是完成确定的，在方法运行期间不会修改局部变量表的大小。

2）操作数栈：

也是一个数组，但是通过栈操作来访问。虚拟机将操作数栈为工作区。

基于栈的执行引擎，虚拟机把操作数栈作为它的工作区，大多数指令都要从这里弹出数据、执行运算，然后把结果压回操作数栈。

操作数即是那些被指令操作的数据。如a=b+c时，就将即将被操作的参数压栈，将b和c压栈，然后由操作指令将它们弹出，并执行操作。

3）帧数据区：

处理常量池解析，异常处理等

4）动态连接：

每个栈帧都包含一个指向运行时常量池（方法区的一部分）中该栈帧所属方法的引用。持有这个引用是为了支持方法调用过程中的动态连接。Class文件的常量池中有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另一部分将在每一次的运行期间转化为直接应用，这部分称为动态连接。

5）方法出口：

返回方法被调用的位置，恢复上层方法的局部变量和操作数栈，如果无返回值，则把它压入调用者的操作数栈。

### JVM栈性能调优参数：

--当JVM栈的空间不足时，会抛出StackOverflowError的错误。

1）-Xss 设置线程栈的大小，堆的内存越大，线程栈的内存越大，能创建的线程数就越小。

java虚拟机栈,规定了两种异常状况：

如果线程请求的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。

如果虚拟机栈动态扩展，而扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

## 本地方法栈(Native Method Stack)

Java通过Java本地接口JNI（Java Native Interface）来调用其它语言编写的程序，在Java里面用native修饰符来描述一个方法是本地方法。

JVM采用本地方法栈来支持native方法的执行，区域用于存储每个native方法调此用的状态。

性能调优参数：可由-Xss设置。

虚拟机栈为虚拟机执行Java方法（也就是字节码）服务。本地方法栈则是为虚拟机使用到的Native方法服务。

## 堆(Heap)

### 概述

线程共享，JVM启动时立即创建，用来存放new出来的对象实例和数组。

在堆中声明的对象，是不能直接访问的，必须通过在栈中声明该对象的引用变量来访问。引用变量相当于该对象的一个名称，使用栈中的引用变量来访问堆中的对象或数组。

注：创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆中，但是它们共享该类的方法。对象使用方法的时候方法才被压入栈，方法不使用则不占用内存。

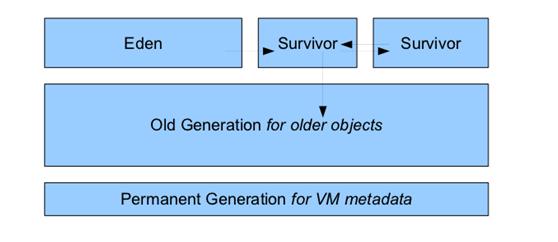
### 分类：

堆由新生代，老年代，持久代(即方法区)组成。

新生代内存由Eden空间、From Survivor空间、To Survivor空间组成，默认按照8:1:1的比例分配。

### JDK1.7中堆内存划分

JDK1.7版本中，堆内存被分为三块区域：新生代（young generation）、老年代（old generation）、永久代（Permanent Generation for VM matedata）



-1）新生代中，一个对象被创建之后首先会被放到新生代中，假设存活期超两个Survivor之后就会被转移到老年代中。

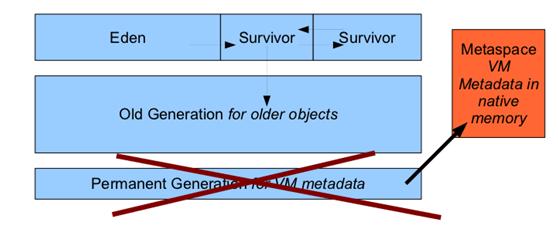
-2）永久代内存中存放着对象的方法、变量等元数据信息。若永久代内存不够会发生错误：java.lang.OutOfMenoryError:PermGen。

### JDK1.8中堆内存划分

JDK1.8将存放元数据中的永久代从堆内存中移到了本地内存（native memory）中，这样永久代不再占用堆内存，它能够通过自己主动增长来避免JDK7以及前期版本号中常见的永久代内存错误(java.lang.OutOfMemoryError: PermGen)。

JDK1.8提供了一个新的设置Metaspace内存大小的参数。

例如：-XX:MaxMetaspaceSize=128m设置最大的元内存为128m。若你设置的元内存空间太小会发生错误：java.lang.OutOfMemoryError:Metadata space。



## 堆-新生代

几乎所有新生成的对象在新生代的Eden区生成（大对象直接在老年代生成）。新生代的目标就是尽可能快速收集掉那些生命周期短的对象。

新生代又被划分成三个区域：Eden、From Survivor、To Survivor。其比例为Eden：From：To=8:1:1（可通过-XX:SurvivorRatio指定）。

当Eden区满后触发新生代的MinorGC，将Eden区和非空闲Survivor区存活的对象移动到另外一个空闲的Survivor区中；再清空Eden区和From Survivor区；最后互换From区和To区。

对象总是在Eden区出生，From Survivor保存当前的幸存对象，To Survivor为空。保证一个Survivor区是空的，两个Survivor区是对称的，没有先后关系。

在特殊情况下，Survivor区可以配置多个，这样可以增加对象在新生代中的存活时间，减少被放到年老代的可能。

### 为什么新生代有两个Survivor区？

如果只有一个survivor区，那么这个survivor区里面的dead object在清理掉后就会产生内存碎片，为了避免内存碎片那么必须将live object移动来移动去，但这样就会影响性能。

如果有两个survivor区，在发生垃圾收集时，将Eden区和from survivor区还存活的对象直接移动到to survivor区，再清空这两个区，这样就不会出现内存碎片的问题。

### 新生代对象晋升到老年代情况

--相关链接：https://blog.csdn.net/qq\_27327855/article/details/80586510

1、Eden区满后，进行Minor GC，当Eden区和from Surviror区依然存活的对象无法放入到to Survivor区中，则通过分配担保机制提前转移到老年代中

2、若对象体积太大（超过-XX:pretenureSizeThreshold值），新生代无法容纳这个对象，就会绕过新生代，直接在老年代分配，此参数只对Serial及ParNew两款收集器有效。

3、长期存活的对象将进入老年代。

如果对象在Eden出生并在第一次发生MinorGC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；以后每熬过一轮MinorGC，年龄Age+1，当年龄到一定程度时（超过-XX:MaxTenuringThreshold值）会移动到老年代。

4、动态对象年龄判定。

虚拟机并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

## 直接内存(Direct Memory)

在 JDK1.4 中引入了NIO，可以通过 Native 方法直接在堆外分配内存，然后通过在堆中存储的引用来对这块内存区域做操作。

注：这块区域并不会在 -Xmx和 -Xms 设置的大小之内，因此在设置 JVM 参数的时候要注意考虑这块内存区域，避免设置的内存区域总额大于物理内存。

直接内存并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现。

可通过-XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆的最大值（-Xmx指定）一样。

## 方法区(Method Area)

### 概述

线程共享，随着JVM启动而创建。

用于存储已被虚拟机加载的类信息，包括：静态变量（static）、常量（final static）、字节码（解释器解释所得）、机器码（jit即时编译器编译所得）。

JDK1.7将方法区放到堆区了，在堆的持久代里面。Jdk1.8中永久区被移除，被元数据区所取代。

持久代就是经常说的方法区里面存放类信息，常量池，方法等。例如 static String str0="0123456789";是放在方法区里。

通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域。

方法区的内存回收目标主要是针对常量池的回收和对类型的卸载。当方法区使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。如果你项目中使用代理模式或者CGLIB的话可能在运行时生成大量的类，使得永久区内存溢出。

### 方法区存放的信息包括：

1）类的基本信息：

每个类的全限定名

每个类的直接超类的全限定名（可约束类型转换）

直接超接口的全限定名的有序列表

是类还是接口

该类的访问修饰符

2）以装载类的详细信息：

包括运行时常量池、字段信息、方法信息、静态变量、到类classLoader的引用、到类class的引用。

--1）运行时常量池：在方法区中，每个类都对应一个常量池，存放该类所用到的所有常量，常量池中存储了诸如文字字符串、final变量值、类名和方法名常量。它们以数组形式通过索引被访问，是外部调用与类联系及类型对象化的桥梁。

--2）字段信息：存放类中声明的每一字段的信息，包括字段的名、类型、修饰符。字段名称指的是类或接口的实例变量或类变量，字段的描述符是一个指示字段的类型的字符串，如private A a=null; 则a为字段名，A为描述符，private为修饰符。

--3）方法信息：存放类中声明的每一个方法的信息，包括方法名、返回值类型、参数类型、修饰符、异常、方法的字节码。（在编译的时候，就已经将方法的局部变量、操作数栈大小等确定并存放在字节码中，在装载的时候，随着类一起装入方法区。）

在运行时，JVM从常量池中获得符号引用，然后在运行时解析成引用项的实际地址，最后通过常量池中的全限定名、方法和字段描述符，把当前类或接口中的代码与其它类或接口中的代码联系起来。

### 方法区卸载Class的条件

1）该类所有的实例已经被回收

2）加载该类的ClassLoader已经被回收

3）该类对应的java.lang.Class对象没有任何地方被引用

附：方法区除了回收无用class，也回收废弃常量，即没有被引用常量

## 各个常量池的情况

目的：JVM为了减少字符串对象的重复创建。

字符串常量池只存储引用，不存储内容！！

JDK1.8之前字符串常量池位于方法区；JDK1.8之后字符串常量池已经被挪到堆中。

每一个类或者接口，在其编译后生成的.class文件中，有一个部分叫做常量表(constant\_pool\_table)，JVM将class文件加载以后，就解析常量表的内容到运行时常量池。

### 分类

分3类：常量池、运行时常量池、字符串常量池。

1）常量池（Constant Pool Table）

常量池存储的是编译期生成的两大类常量，这部分内容将在类加载后放入方法区的运行时常量池或字符串字面量池中。这两大类常量如下：

-1）字面量（Literal）：文本字符串、声明为final的常量值等---->用双引号引起来的字符串字面量都会放进里面

-2）符号引用（Symbolic Refarences）:类和接口的全限定名、字段的名称和描述符、方法的名称和描述符

2）运行时常量池（Runtime Constant Pool）：

字符串池/字符串常量池(String Pool/String Constant Pool)：是常量池中的一部分，存在编译期间类中产生的字符串类型数据。

### 字符串常量池的工作原理

工作原理：！！

当代码中出现字面量形式创建字符串对象时，JVM首先会对这个字面量检查。若字符串常量池中存在相同内容的字符串对象的引用，则将这个引用返回，否则新的字符串对象被创建，然后将这个引用放入字符串常量池，并返回该引用。

实现前提：

前提条件是Java中String对象是不可变的，这样可以安全保证多个变量共享同一个对象。若String对象可变，某个引用修改了该对象的值，那么其他的变量也会受到影响，显然这样是不合理的。

详情：http://droidyue.com/blog/2014/12/21/string-literal-pool-in-java/

# 对象在jvm内存的分配规则

详情：https://blog.csdn.net/hyg0811/article/details/101444643

1、对象优先在eden分配；

2、大对象直接进入老年代；

大对象指需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组。

虚拟机提供一个-XX:PretenureSizeThreshold参数，大于该值的对象直接在老年代分配。其目的是避免在Eden区及两个Survivor区之间发生大量的内存复制

3、长期存活的对象将进入老年代

虚拟机给每个对象定义了一个对象年龄（Age）计数器。每经过一次MinorGC则对象年龄+1，当年龄达到15岁（默认值），就会晋升到老年代。

年龄阈值设置：-XX:MaxTenuringThreshold

4、动态对象年龄

虚拟机并不是永远地要求兑现过的年龄必须达到了MaxTenuringThreshold才能晋升老年代。

问：jvm的分代年龄为什么是15？

在32位HotSpot虚拟机中，若对象处于未被锁定的状态下，那么Mark Word的32bit空间中25bit用于存储对象哈希码，4bit用于存储对象分代年龄，2bit用于存储锁标志位，1bit固定为0。因此，对象的分代年龄占4位，也就是0000，最大值为1111也就是最大为15，而不可能为16，20之类的了。

问：对象能晋升老年代条件？

-1）虚拟机并不是永远地要求兑现过的年龄必须达到了MaxTenuringThreshold才能晋升老年代；

-2）若Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

# jvm调优

https://blog.51cto.com/13672983/2378314?source=dra --看不懂！！

## 调优原则

gc调优步骤：

1.打印gc日志

> java -jar -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Xloggc:./gc.log eureka.jar

2.通过gc日志分析得到关键性指标

调优原则：

-1）将进入老年代的对象数量降到最低

-2）减少FullGC的执行时间

目标：避免fullGC和gc频率，还有每次gc时间。达到最终gc时间小，程序停顿小。

1.堆大小的调优：

一般来说调整参数越大越好，1、降低gc频率但也增加单次gc时间；2、对象更有可能成为垃圾。

平衡调整new和old的比例。可以xms/xmx调整为一样大，初始化一般为内存的1/64，1/4。避免频繁调整，每次调整都会触发一次FullGC.当然也不总是，1、堆调整代价大但是如果死机了呢。还不如一次fullgc

2.新生代调优：

1、Eden区大小调整，xmn，降低gc频率

2、晋升,(1)尽可能让对象待在survival中，使之在新生代被收回，不然进入old区，降低旧生代的对象和gc。(2)调整比例，平衡点。New 里面的Eden和survival0. survival01  8:1:1。(3)新生代长时间没被回收的，设置阀值，进行放入Old，避免频繁copy

3.老年代的调优：

尽可能对新生代调优，在不紧要的时候受用fullgc。加Cpu，硬件调整。

程序优化

JVM参数：

堆内存大小：初始化堆内存(-Xms)、最大堆内存(-Xmx)

新生代空间大小：新生代和老年代的内存比(-XX:NewRatio)、新生代内存大小(-XX:NewSize)、Eden和Survivor区的内存比(-XX:SurvorRatio)

永久代内存大小：-XX:PermSize、-XX:MaxPermSize

GC优化最常见的参数是-Xms、-Xmx和-XX:NewRatio。

只有当出现OutOfMemoryError错误时你才需要区设置永久代内存。

## 内存溢出&内存泄漏

如果内存没有被及时回收造成内存占用失控主要有以下两种情况：内存溢出&内存泄漏

内存溢出是创建太多对象导致内存空间不足；内存泄漏是无用对象没有回收

内存溢出（Memory Overflow）：

程序在运行过程中无法申请到足够内存而出现OOM。

内存泄露（Memory Leak）：

指对象占用内存后，对象没有被GC回收，它始终占用内存。内存泄漏的堆积回造成内存溢出。

内存泄漏一般是因为对象被引用无法回收，比如一个集合中存着很多对象，可能你在外部代码把对象的引用置空了，但是由于对象还被集合给引用着，所以无法被回收，导致内存泄漏。测试也很简单，就在集合里添加对象，添加完以后把引用置空，循环操作，一会就会出现oom异常，原因是内存泄漏太多了，导致没有空间分配新的对象。

OOM分三种:

1）堆内存溢出：主要原因是创建了太多对象，比如一个集合类死循环添加一个数，此时设置jvm参数使堆内存最大值为10m，一会就会报oom异常。

2）栈内存溢出：主要与栈空间和线程有关，因为栈是线程私有的，如果创建太多线程，内存值超过栈空间上限，也会报oom。

3）方法区内存溢出：主要是由于动态加载类的数量太多，或不断创建一个动态代理，jdk1.7之前会报permGem OOM，jdk1.8则会报meta space oom，因为jdk1.8中使用元数据区替换了堆中的永久代。

Java存在两种内存异常：

1）如果线程请求的栈深度大于虚拟机允许的深度，将抛出StackOverflowError错误

2）如果虚拟机栈可以动态扩展，在扩展时无法申请到足够的内存，就会抛出OutOfMemoryError错误

## 三种溢出介绍

### 堆内存溢出

Heap的大小是Young Generation和Tenured Generation之和。在jvm中如果98%的时间是用于GC，且可用的Heap size不足2%时将抛出该异常信息。

堆内存越大越不容易Full GC；越小，则垃圾回收越频繁，增加垃圾回收时间，减低吞吐量。

堆溢出会抛出异常：java.lang.OutOfMemoryError: Java heap space

设置堆内存：

-1）-Xms1024m：初始化堆内存。

-2）-Xmx1024m：最大堆内存。经验：设值Xms大小等于Xmx大小。

-3）-XX:NewSize=512m：初始化新生代内存，需小于-Xms的值。

-4）-XX:MaxNewSize=1024m：最大新生代内存，应该小于-Xmx的值。

-5）-XX:NewRatio：设置新生代和老年代的比例。

-6）-XX:SurviorRatio：新生代中Eden区和Survivor区的比例，默认8:1:1

-7）-XX:TargetSurvivorRatio：设置survivor区的可使用率，达到此值后被送到老年代

### 永久代溢出

出现原因：当程序加载很多class情况下，超出PermSize情况下会抛出异常。

PermGen space：全称是Permanent Generation space，是指内存的永久保存区域。

主要存放Class和Meta信息，Class在load时被放入PermGen space区域，且JVM在运行期间不会清除该区域。

jdk1.7版：

-XX:PermSize=1024M #初始化持久区大小

-XX:MaxPermSize=1024M #最大持久区大小，默认为64MB

经验: 设置PermSize大小等于MaxPermSize大小

溢出会抛出：java.lang.OutOfMemoryError: PermGen space

jdk1.8版：

-XX:MaxMetaspaceSize=1024M #设置元数据区大小

溢出会抛出：java.lang.OutOfMemoryError Metaspace

### 栈内存溢出

溢出会抛出：java.lang.StackOverflowError

出现原因：调用函数的“层”太多了。

解决方法: 代码中递归也不要递归的层次过多。通常栈的大小是1-2MB

## JVM性能调优

-1）JDK1.7版

-Xms1024m -Xmx1024m

-XX:NewSize=512m

-XX:MaxNewSize=1024m

-XX:PermSize=1024m

-XX:MaxPermSize=1024m

-XX:+DisableExplicitGC

-2）JDK1.8版

-Xms1024m -Xmx1024m

-XX:NewSize=512m

-XX:MaxNewSize=1024m

-XX:+DisableExplicitGC

-XX:MetaspaceSize1024m #指定元空间的初始大小，若不指定时元空间会根据应用程序运行时的需要动态调整大小

-XX:MaxMetaspaceSize={unlimited} #元空间大小受限于你机器的内存

附：jdk1.8中已移除永久代，因此已经没有PermSize、MaxPermSize

## 思考题

线上应用系统出现问题，怎么快速定位系统哪块资源问题？