**Java部分**

## 谈谈你对java的理解

♦平台无关性

♦面向对象

♦GC

♦类库

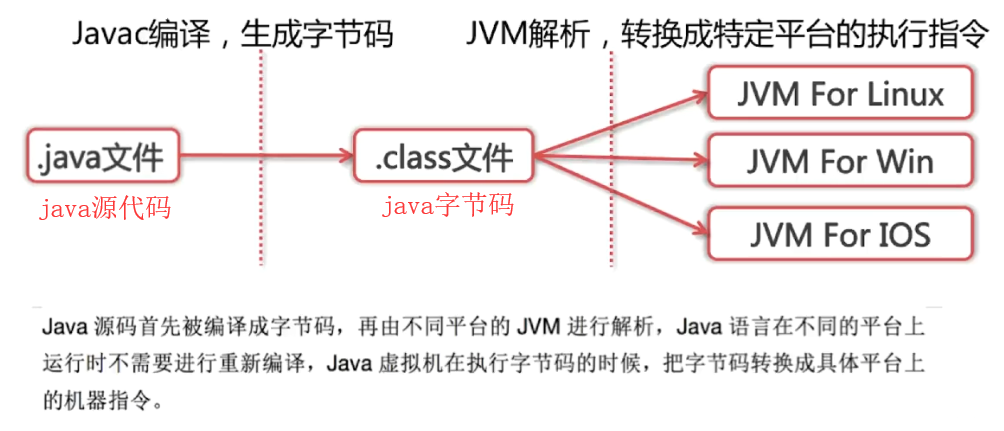
♦语言特性

♦异常处理

## Java平台无关性的实现

### 1.实现过程

|  |
| --- |
| **Compile once，Run Anywhere**  编译一次，到处运行；指的是Java语言的跨平台能力，而这种能力主要由Java虚拟机提供的。Java源代码经过javac编译后生成屏蔽操作系统和硬件的字节码，再经由JVM编译、解释执行。  拓展:http://www.bubuko.com/infodetail-2588169.html |



### 为什么要先编译成字节码再解析成机器码（为什么jvm不直接将源码解析成机器码去执行）

1. 准备工作：直接编译每次执行都需要各种检查，先编译再解析可以可以节省大量的准备工作,提升效率。
2. 兼容性：可以将别的语言解析成字节码。

### 如何查看字节码

1. **使用命令行：**

命令行有两种方式

java -c

在编译后，进入字节码存放的文件夹，例如：FileA\Hello.class，使用命令行窗口,执行命令：

java -c FileA\Hello.class，即可在窗口浏览字节码文件

javap -verbose HellloWorld.class

javap -verbose Hello.class ，同样可以浏览字节码文件

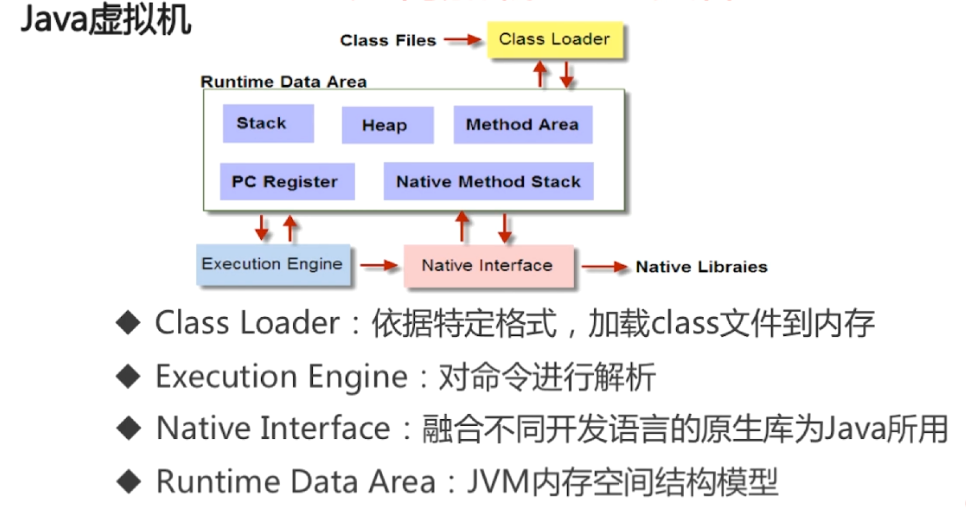
**（2）外接工具的方式**

Java Bytecode Editor等工具

1. **编译器内置插件**

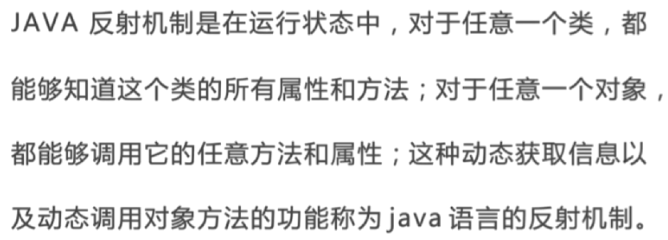
## JVM如何加载.class文件

### Jvm架构



## Java反射

### 谈谈反射



### 写一个反射的例子



## ClassLoader类加载器

### 类从编译到执行的过程

1. 编译器将Robot.Java源文件编译成Robot.class字节码文件
2. ClassLoader将字节码转换成JVM中class<Robot>对象

### 2.谈谈classLoader

ClassLoader在Java中有着非常重要的作用,它主要工作在Class装载的加载阶段,其主要作用是从系统外部获得Class二进制数据流.它是Java的核心组件,所有的Class都是由ClassLoader进行加载的,ClassLoader负责通过将Class文件里的二进制数据流装载进系统,然后交给Java虚拟机进行连接，初始化等操作.

### 3.classLoader的种类

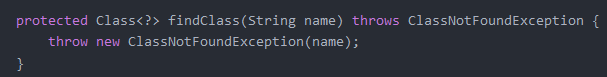
|  |
| --- |
| **1.BootStrapClassLoader : C++编写, 加载核心库Java.\*;**  **2.ExtClassLoader : Java编写,加载扩展库javax.\*;**  **3.AppClassLoader : Java编写,加载程序所在目录;**  **4.自定义ClassLoader : Java编写,定制化加载;** |

### 自定义classLoader的实现过程

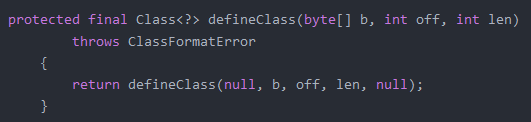
**应用场景：**

|  |
| --- |
| 假如，你开发的java程序都需要从E:\classloader1目录下的类文件中加载class，而不是系统指定的系统目录或者classpath目录下加载，则如何解决？ 需要自定义classloader。 |

1. 重写findClass() : 这个函数是用来寻找Class文件的;



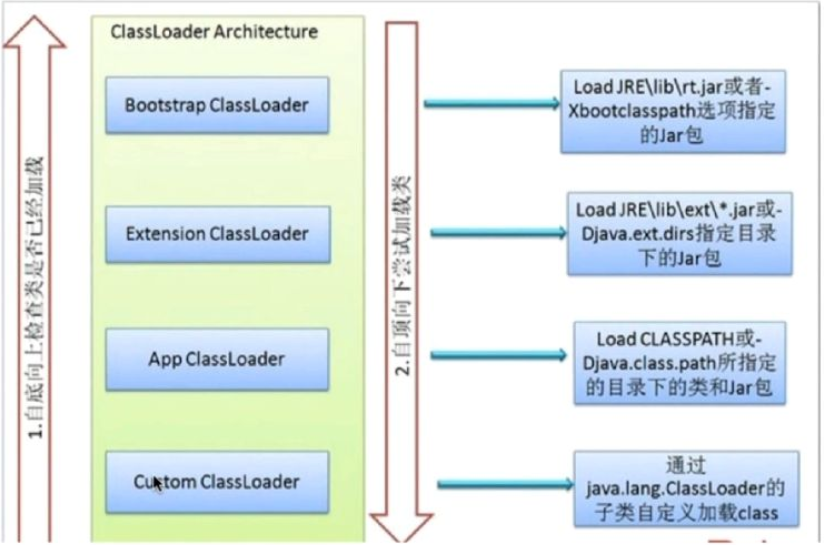
1. 重写defineClass() : 解析字节码返回对象;



1. 自定义ClassLoader

主体部分实现

### ClassLoader的双亲委派机制



#### 双亲委派机制：

·当AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

·当ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。

·如果BootStrapClassLoader加载失败（例如在$JAVA\_HOME/jre/lib里未查找到该class），会使用ExtClassLoader来尝试加载；

·若ExtClassLoader也加载失败，则会使用AppClassLoader来加载，如果AppClassLoader也加载失败，则会报出异常ClassNotFoundException。

#### （2）为何使用双亲委派机制去加载类

1. 避免重复字节码的加载;
2. 提供了JDK核心类加载的沙箱环境;

#### （3）通过openJDK查看java源码

**原因**：由于jdk部分代码闭源，不对外公布，需要查看openJdk（也是不完全开源）

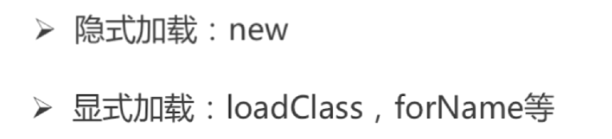
**场景**：查看classLoader下的native方法源码

### 拓展

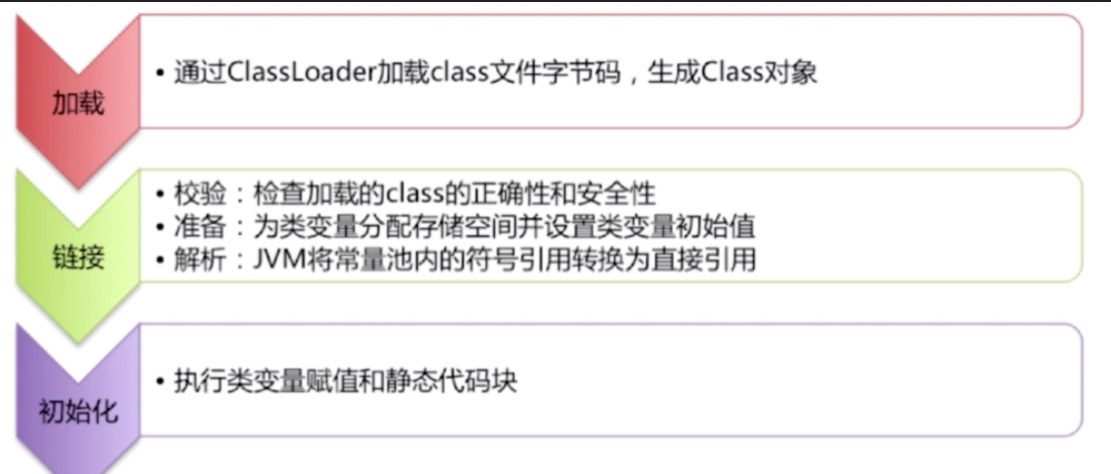
|  |
| --- |
| **类的加载过程分为加载阶段、连接阶段、初始化。**   1. 加载阶段：寻找class文件。 2. 连接阶段：验证class文件的正确性；为类的静态变量准备内存，并为其初始化默认值；把类中符号引用转换为直接引用。 3. 初始化阶段：为累的静态变量赋予正确的初始值，也就是在代码的值。 |
| JVM目前对类的初始化时一次lazy。所以何时初始化就会根据不同情况而不同，如果主动使用，则会进行类的初始化。 |
| **类的6种主动使用情况：**   1. 通过调用静态变量会导致类的初始化。 但是若是静态常量，则不会。 2. 调用静态方法，会导致类的初始化过程。 3. 调用class.forName。 4. 调用子类会导致父类初始化，但是如果只是通过子类调用父内的静态方法后者变量，则不会导致子类初始化。 5. 调用main方法的类也会被初始化。 6. 调用 new关键字，必然会进行类的初始化，但是数组除外。   **类的被动使用：除了主动的以外，都是被动的。** |

## loadClass和forName的区别

### 1.类的加载方式



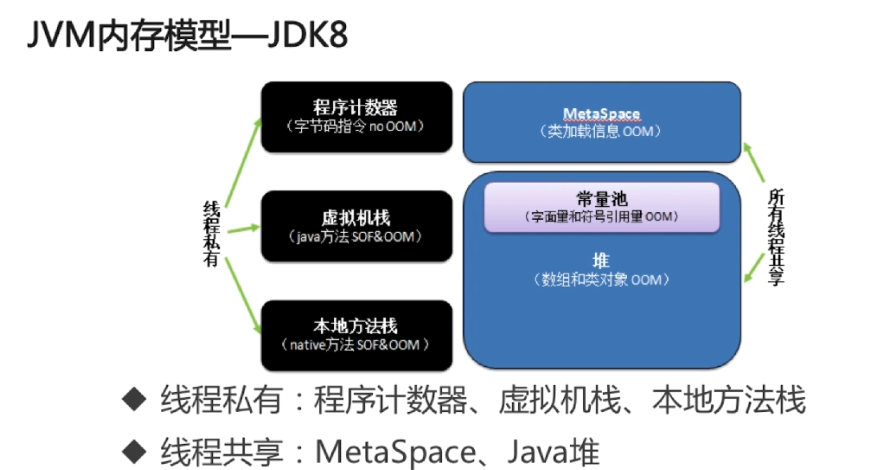
### 2.类的装载过程



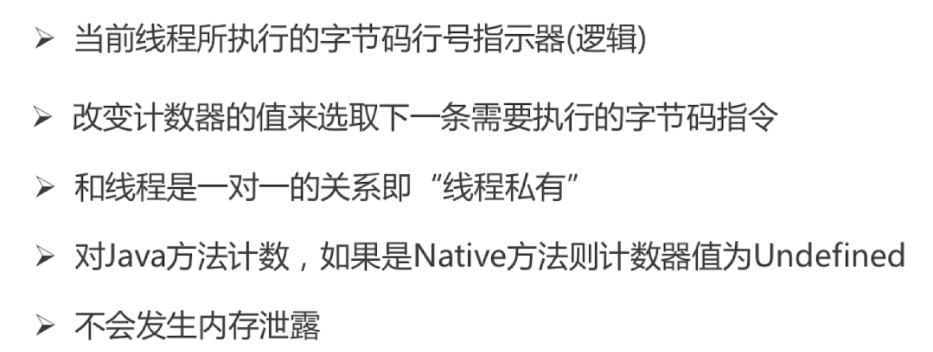
### 3.loadClass和forName的区别

 （1） Class.forName得到的class是已经初始化完成的;  
 （2） Classloder.loadClass得到的class是还没有链接的;

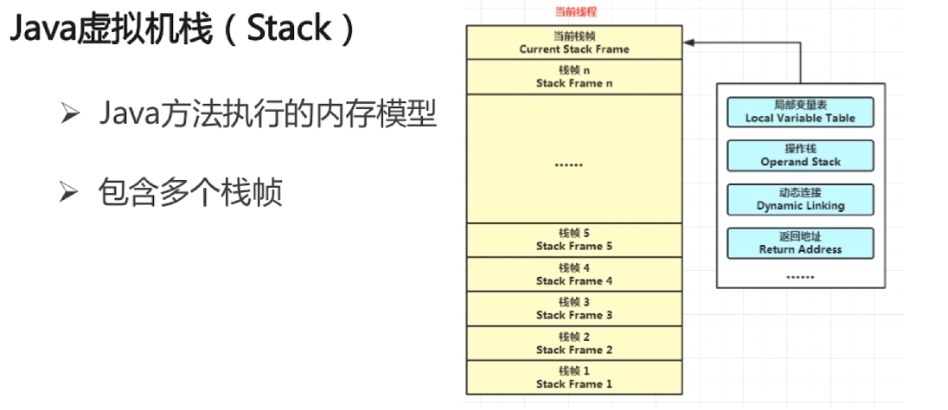
## Java的内存模型

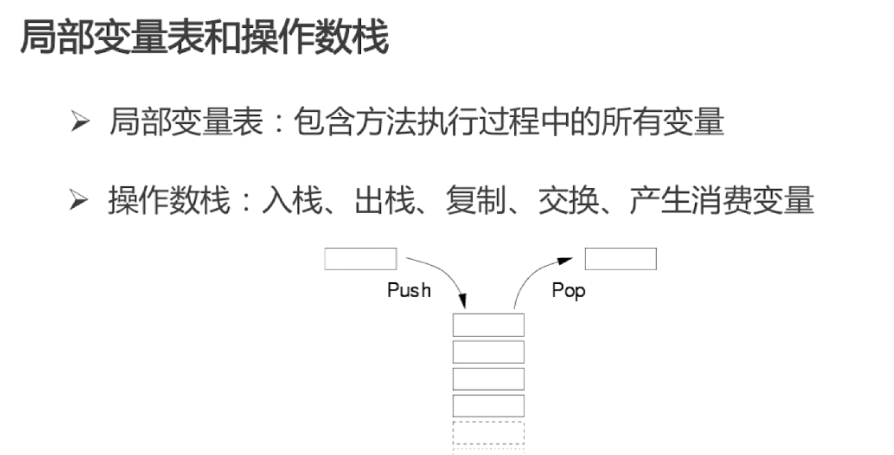


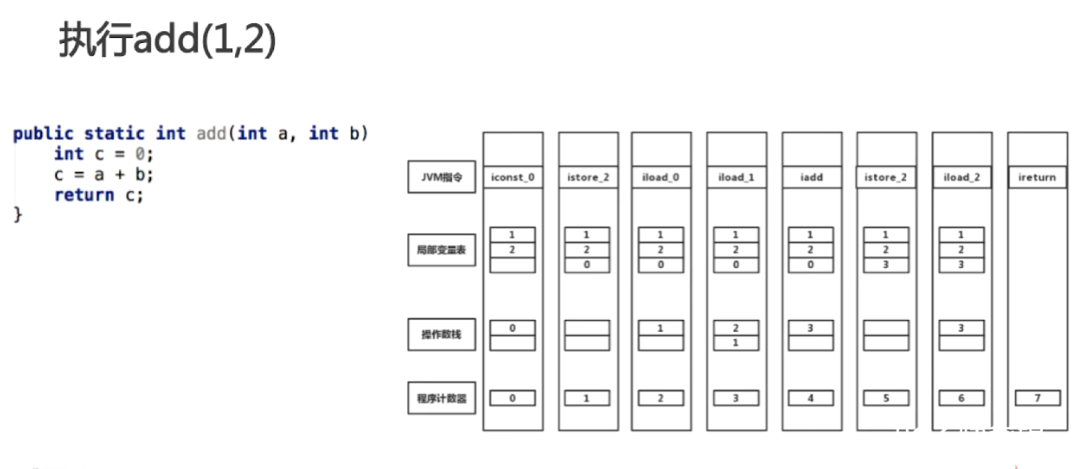
**程序计数器：**

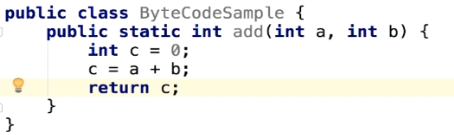


程序计数器是逻辑计算器，而非物理计算器









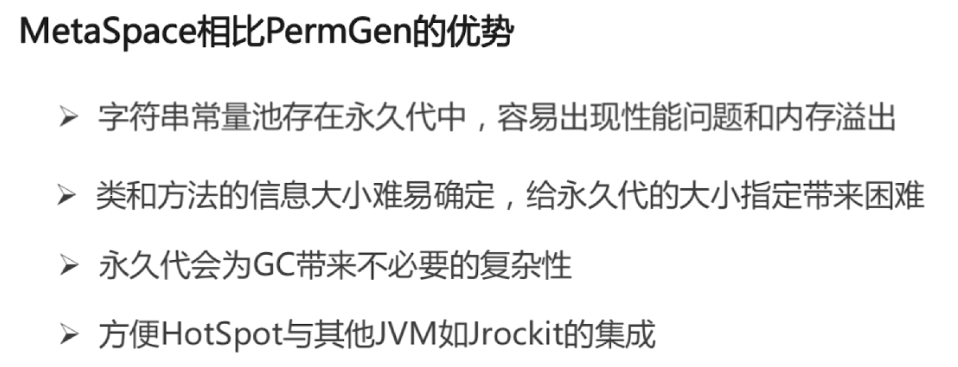
递归为什么会引发java.lang.StackOverFlowRrror异常？

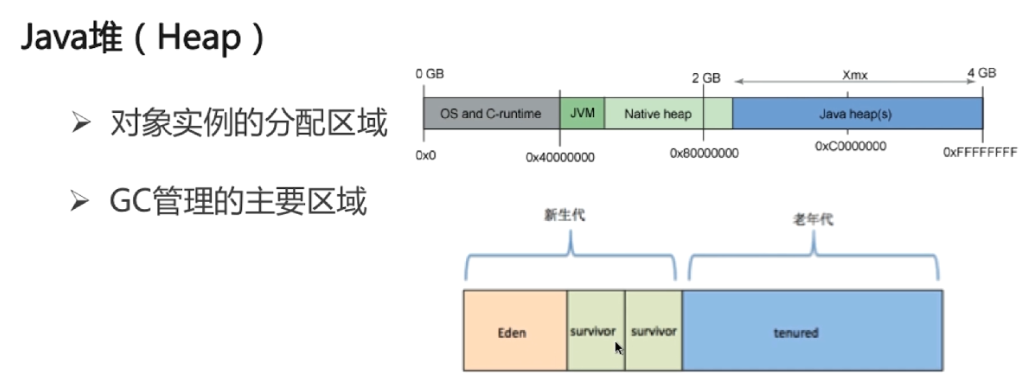
虚拟机栈过多会多引发 java.lang.OutOfMemoryError异常

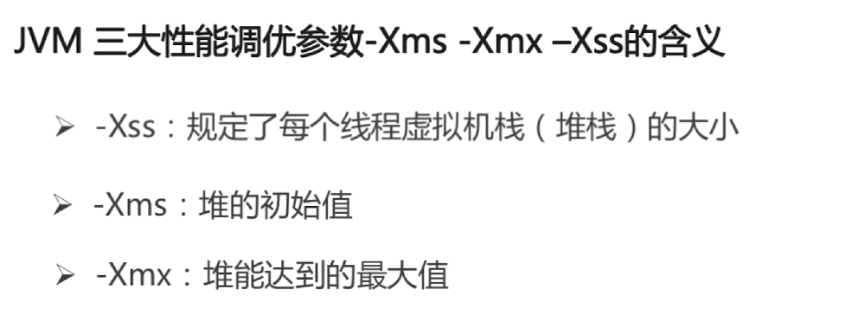


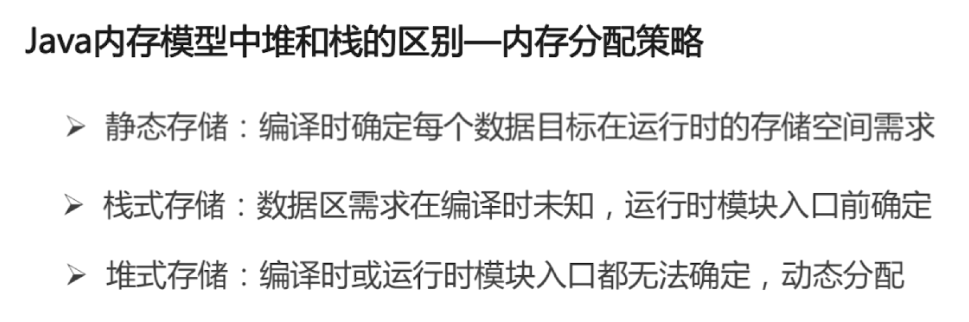
**元空间（MetaSpace）与永久代（PermGen）的区别**

**元空间使用本地内存，而永久代使用的事jvm的内存。**

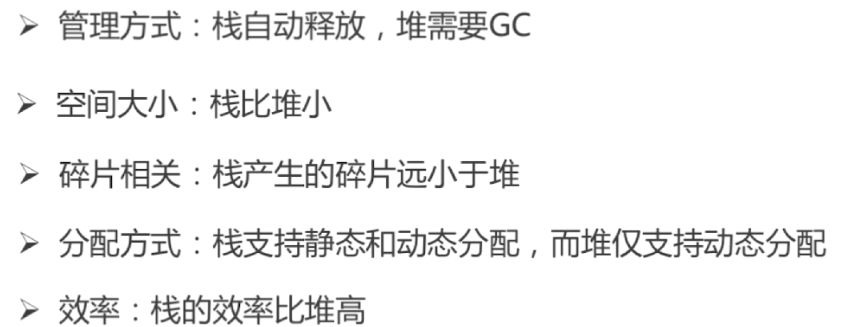








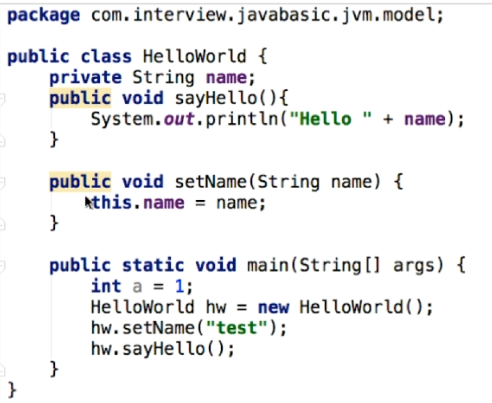
**Java内存模型中堆和栈的区别**

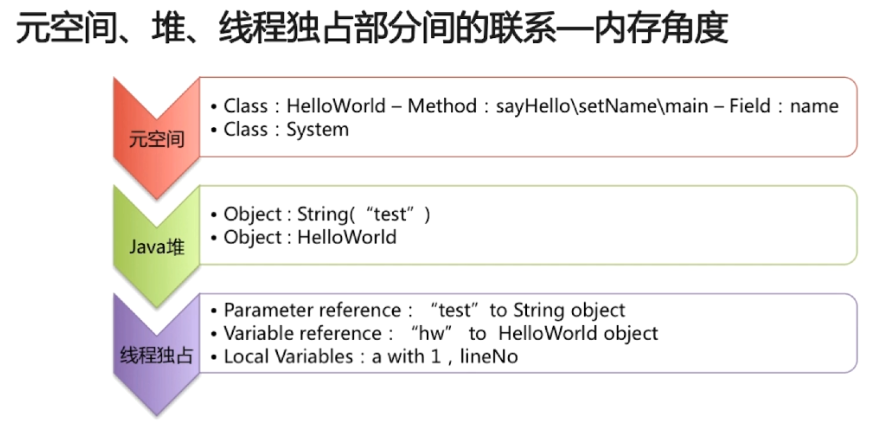


|  |
| --- |
| **堆和栈的概念：**  **堆：**主要用于存储实例化的对象，数组。由JVM动态分配内存空间。一个JVM只有一个堆内存，线程是可以共享数据的。  **栈：**主要用于存储局部变量和对象的引用变量，每个线程都会有一个独立的栈空间，所以线程之间是不共享数据的。 |

|  |
| --- |
| 拓展：  1.栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。  5.栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短 |







|  |
| --- |
| 线程独占：即程序运行阶段。  Main线程分配对应的虚拟机栈， 本地栈和程序计数器 |



## Java常用指令

### Javac

Javac是用来**编译.java**文件的，常用格式如下：

javac destdir srcFile

Eg：javac com/springboot/lizhi/entity/User.java

### Java

使用 java 命令运行.class文件

### Javap

javap是 Java class文件分解器，可以反编译（即对javac编译的文件进行反编译），也可以查看java编译器生成的字节码。用于分解class文件。