Java面试题-内存+GC+类加载器+JVM调优

一.什么是值传递和引用传递？ Java 中是值传递还是引用传递，还是都有？

值传递 就是在方法调用的时候，实参是将自己的一份拷贝赋给形参，在方法内，对该参数值的修改不影响原来实参。
引用传递 是在方法调用的时候，实参将自己的地址传递给形参，此时方法内对该参数值的改变，就是对该实参的实际操作。
在 Java 中只有一种传递方式，那就是 值传递,
可能比较让人迷惑的就是 Java 中的对象传递时，对形参的改变依然会意向到该对象的内容。

二.Java内存分配

每运行一个Java程序会产生一个Java进程，每个Java进程可能包含一个或者多个线程，每一个Java进程对应唯一一个JVM实例，每一个JVM实例唯一对应一个堆，每一个线程有一个自己私有的栈。
进程所创建的所有类的实例（也就是对象）或数组（指的是数组的本身，不是引用）都放在堆中,并由该进程所有的线程共享。Java中分配堆内存是自动初始化的，即为一个对象分配内存的时候，会初始化这个对象中变量。虽然Java中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在栈中分配,也就是说在建立一个对象时在堆和栈中都分配内存，在堆中分配的内存实际存放这个被创建的对象的本身，而在栈中分配的内存只是存放指向这个堆对象的引用而已。
局部变量 new 出来时，在栈空间和堆空间中分配空间，当局部变量生命周期结束后，栈空间立刻被回收，堆空间区域等待GC回收。

JVM的内存可分为5个区：堆(heap)、虚拟机栈(Vitual Machine Stack),本地方法栈,方法区(method，也叫静态区)和程序计数器：

堆区:

1.存储的全部是对象，每个对象都包含一个与之对应的class的信息(class的目的

是得到操作指令)；
2.jvm只有一个堆区(heap)，且被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身和数组本身；

栈区:

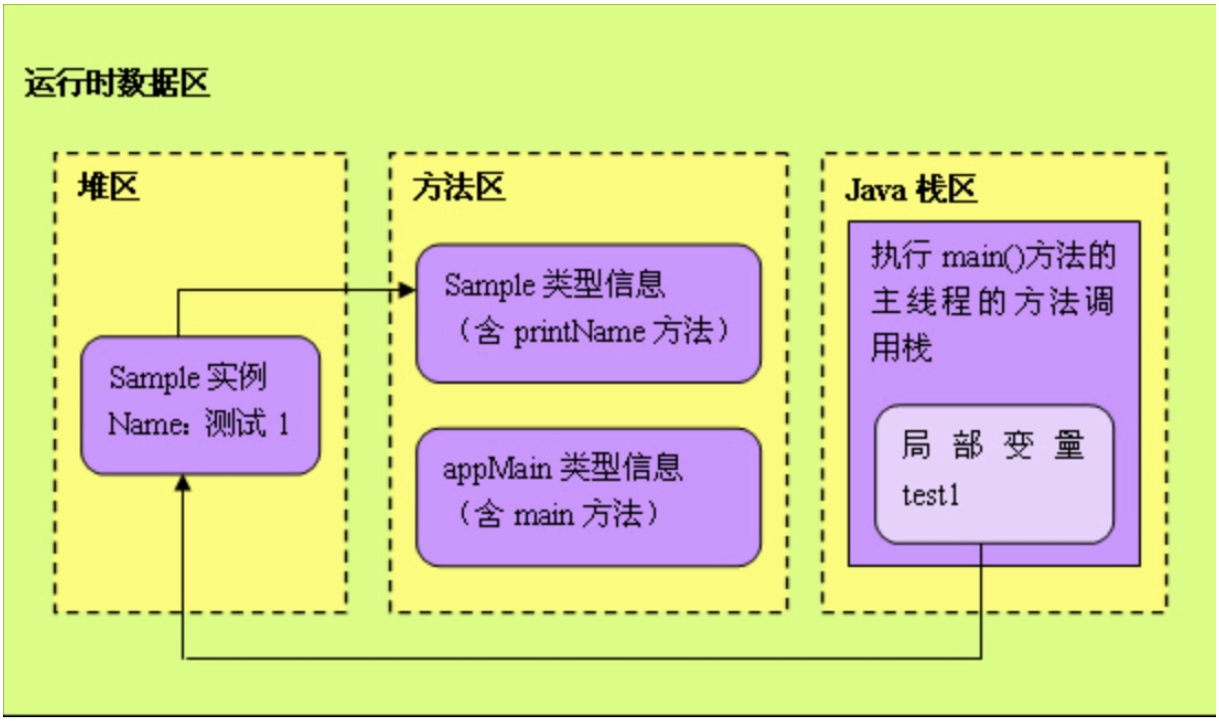
1.每个线程包含一个栈区，栈中只保存基础数据类型本身和自定义对象的引用；
2.每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问；
3.栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)；

方法区（静态区）：

1.被所有的线程共享，方法区包含所有的class（class是指类的原始代码，要创建一个类的对象，首先要把该类的代码加载到方法区中，并且初始化）和static变量。
2.方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。

```
    AppMain.java

    public   class  AppMain //运行时，jvm 把appmain的代码全部都放入方法区
    {
    public   static   void  main(String[] args)  //main 方法本身放入方法区。
    {
    Sample test1 = new  Sample( " 测试1 " );   //test1是引用，所以放到栈区里，  Sample是自定义对象应该放到堆里面
    Sample test2 = new  Sample( " 测试2 " );

    test1.printName();
    test2.printName();
    }
    }

    public   class  Sample //运行时，jvm 把appmain的信息都放入方法区

    {
    /** 范例名称 */
    private String name; //new Sample实例后，  name 引用放入栈区里，name 对应的 String 对象放入堆里

    /** 构造方法 */
```

```
    public  Sample(String name)
    {
    this .name = name;
    }

    /** 输出 */
    public   void  printName() //在没有对象的时候，print方法跟随sample类被放入方法区里。
    {
    System.out.println(name);
    }
    }
```



三.JVM 内存分哪几个区，每个区的作用是什么？

Java虚拟机主要分为以下几个区：

1.方法区：

1.有时候也成为 永久代 ，在该区内很少发生垃圾回收，但是并不代表不发生 GC ，在这里进行的 GC 主要是对方法区里的常量池和对类型的卸载

2.方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译

器编译后的代码等数据。

3.该区域是被线程共享的。

4.方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

2.虚拟机栈：

1.虚拟机栈也就是我们平常所称的 栈内存, 它为 Java 方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

2.虚拟机栈是线程私有的，它的生命周期与线程相同。

3.局部变量表里存储的是基本数据类型、 returnAddress 类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定

4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式

5.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支持方法调用过程中的动态连接 . 动态链接就是将常量池中的符号引用在运行期转化为直接引用。

3.本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。

4.堆

Java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

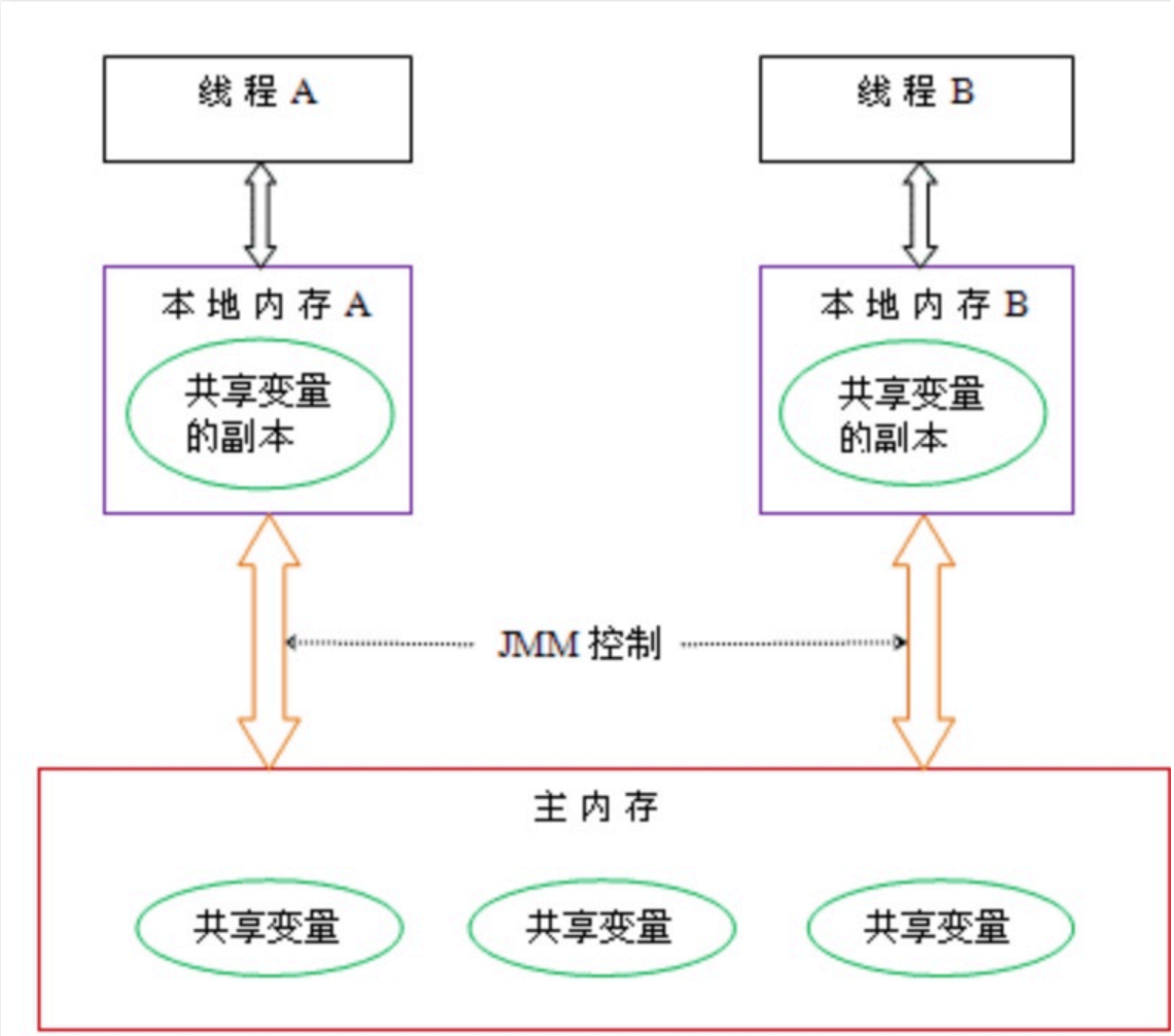
5.程序计数器

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行

的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个 Java 虚拟机规范没有规定任何 OOM 情况的区域。

四.Java内存模型

Java 内存模型 (JMM) 是线程间通信的控制机制 .JMM 定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读 / 写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java 内存模型的抽象示意图如下：



从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1. 首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。
2. 然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

写的很好 :http://www.infoq.com/cn/articles/java-memory-model-1

五.简述Java垃圾回收机制？

在 Java 中，程序员是不需要显式的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

六.如何判断一个对象是否存活 ?(或者 GC 对象的判定方法)

判断一个对象是否存活有两种方法：

1.引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“ 死对象 ”, 将会被垃圾回收 .
引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象 B ，对象 B 又引用者对象 A ，那么此时 A,B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2.可达性算法 (引用链法)

该算法的思想是：从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。
在 Java 中可以作为 GC Roots 的对象有以下几种：

- **虚拟机栈中引用的对象**
- **方法区类静态属性引用的对象**
- **方法区常量池引用的对象**
- **本地方法栈 JNI 引用的对象**

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象也不一定会被回收 。当一个对象不可达 GC Root 时，这个对象并 不会立马被

回收 ， 而是处于一个死缓的阶段， 若要被真正的回收需要经历两次标记 .

如果对象在可达性分析中没有与 GC Root 的引用链， 那么此时就会被第一次标记并且进行一次筛选， 筛选的条件是是否有必要执行 finalize() 方法。 当对象没有覆盖 finalize() 方法或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行 finalize() 方法，那么这个对象将会放在一个称为 F-Queue 的对队列中， 虚拟机会触发一个 Finalize() 线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize() 执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。 GC 对处于 F-Queue 中的对象进行第二次被标记，这时，该对象将被移除 "即将回收" 集合，等待回收。

七.Java中垃圾收集的方法有哪些？

1.标记 - 清除：

这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题： 1. 效率不高，标记和清除的效率都很低； 2. 会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC动作。

2.复制算法：

为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清除完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一半的内存。
于是将该算法进行了改进， 内存区域不再是按照 1 ： 1 去划分，而是将内存划分为 8:1:1 三部分，较大那份内存叫 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。 (Java 堆又分为新生代和老年代)

3.标记 - 整理

该算法主要是为了解决标记 - 清除算法 产生大量内存碎片的问题.当对象存活率

较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候先将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

4.分代收集

现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用 复制 算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用 标记 - 整理 或者 标记 - 清除。

七.Java 类加载机制及其过程？

类加载机制:

虚拟机把描述类的数据从 Class 文件加载到内存中，并对数据进行校验，准备,解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型的过程,被称为Java的类加载机制。

具体加载过程:

1.加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情:
1.通过一个类的全限定名获取该类的二进制流。
2.将该二进制流中的静态存储结构转化为方法区运行时的数据结构。
3.在内存中生成该类的 Class 对象，作为该类的数据访问入口。

2.验证

验证的目的是为了确保 Class 文件的字节流中的信息不会危害到虚拟机 . 在该阶段主要完成以下四钟验证:
1.文件格式验证：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型 .
2.元数据验证 : 对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3.字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流

的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4.符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

3.准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

```
public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。
```

4.解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

5.初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

八.类加载器的双亲委派模型机制?

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

使用双亲委派模型的好处在于 **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**。例如类 `java.lang.Object` ，它存在在 **rt.jar** 中，无论哪一个

类加载器要加载这个类，最终都是委派给处于模型最顶端的 **Bootstrap ClassLoader** 进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个 `java.lang.Object` 的同名类并放在 ClassPath 中，那系统中将会出现多个不同的 Object 类，程序将混乱。因此，如果开发者尝试编写一个与 **rt.jar** 类库中重名的 Java 类，可以正常编译，但是永远无法被加载运行。

双亲委派模型的系统实现

在 `java.lang.ClassLoader` 的 `loadClass()` 方法中，先检查是否已经被加载过，若没有加载则调用父类加载器的 `loadClass()` 方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父加载失败，则抛出 `ClassNotFoundException` 异常后，再调用自己的 `findClass()` 方法进行加载。

```
protected synchronized Class<?> loadClass(String name,boolean resolve
)throws ClassNotFoundException{
    //check the class has been loaded or not
    Class c = findLoadedClass(name);
    if(c == null){
        try{
            if(parent != null){
                c = parent.loadClass(name,false);
            }else{
                c = findBootstrapClassOrNull(name);
            }
        }catch(ClassNotFoundException e){
            //if throws the exception ,the father can not complete th
e load
        }
        if(c == null){
            c = findClass(name);
        }
    }
    if(resolve){
        resolveClass(c);
    }
    return c;
}
```

注意，双亲委派模型是 Java 设计者推荐给开发者的类加载器的实现方式，并不是强制规定的。大多数的类加载器都遵循这个模型，但是 JDK 中也有较大规模破坏双亲模型的情况，例如线程上下文类加载器（**Thread Context**

ClassLoader）的出现

九.什么是类加载器，类加载器有哪些？

1.类加载器的概念:

类加载器（ClassLoader）是 Java 语言的一项创新，也是 Java 流行的一个重要原因。在类加载的第一阶段“加载”过程中，需要通过一个类的全限定名来获取定义此类的二进制字节流，完成这个动作的代码块就是 **类加载器**。这一动作是放在 Java 虚拟机外部去实现的，以便让应用程序自己决定如何获取所需的类。

2.类加载器的分类:

从 Java 虚拟机的角度来说,只存在两种不同的类加载器：

一种是启动类加载器（**Bootstrap ClassLoader**），这个类加载器使用 C++ 语言实现（HotSpot 虚拟机中），是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都有 Java 语言实现，独立于虚拟机外部，并且全部继承自 `java.lang.ClassLoader`。

从开发者的角度，类加载器可以细分为：

- 启动（Bootstrap）类加载器：负责将 Java_Home/lib 下面的类库加载到内存中（比如 **rt.jar**）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
- 标准扩展（Extension）类加载器：是由 Sun 的 **ExtClassLoader（sun.misc.Launcher$ExtClassLoader）** 实现的。它负责将 Java_Home /lib/ext 或者由系统变量 java.ext.dir 指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
- 应用程序(系统)（Application）类加载器：是由 Sun 的 **AppClassLoader（sun.misc.Launcher$AppClassLoader）** 实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。由于这个类加载器是 ClassLoader 中的 `getSystemClassLoader()` 方法的返回值，因此一般称为系统（System）加载器。

- 除此之外，还有自定义的类加载器.
  通过继承 java.lang.ClassLoader 类的方式实现。
  它们之间的层次关系被称为类加载器的 **双亲委派模型**。该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过组合（Composition）关系来实现，而不是通过继承（Inheritance）。

类加载器的特性:

1.加载类的开放性

虚拟机规范并没有指明二进制字节流要从一个 Class 文件获取，或者说根本没有指明从哪里获取、怎样获取。这种开放使得 Java 在很多领域得到充分运用，例如：

- 从 ZIP 包中读取，这很常见，成为 JAR，EAR，WAR 格式的基础
- 从网络中获取，最典型的应用就是 Applet
- 运行时计算生成，最典型的是动态代理技术，在 `java.lang.reflect.Proxy` 中，就是用了 `ProxyGenerator.generateProxyClass` 来为特定接口生成形式为“*$Proxy”的代理类的二进制字节流
- 有其他文件生成，最典型的 JSP 应用，由 JSP 文件生成对应的 Class 类

2.类加载器与类的唯一性

类加载器虽然只用于实现类的加载动作，但是对于任意一个类，都需要由加载它的类加载器和这个类本身共同确立其在 Java 虚拟机中的 **唯一性**。通俗的说，JVM 中两个类是否“相等”，首先就必须是同一个类加载器加载的，否则，即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要类加载器不同，那么这两个类必定是不相等的。

这里的“相等”，包括代表类的 Class 对象的 `equals()` 方法、`isAssignableFrom()` 方法、`isInstance()` 方法的返回结果，也包括使用 `instanceof` 关键字做对象所属关系判定等情况。

下代码说明了不同的类加载器对 `instanceof` 关键字运算的结果的影响。

```
package com.jvm.classloading;
```

```
import java.io.IOException;
import java.io.InputStream;

/**
 * 类加载器在类相等判断中的影响
 *
 * instanceof关键字
 *
 */

public class ClassLoaderTest {
    public static void main(String[] args) throws Exception {
        // 自定义类加载器
        ClassLoader myLoader = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name) throws ClassNotFou
ndException {
                try {
                    String fileName = name.substring(name.lastIndexOf
(".") + 1) + ".class";
                    InputStream is = getClass().getResourceAsStream(f
ileName);
                    if (is == null) {
                        return super.loadClass(fileName);
                    }
                    byte[] b = new byte[is.available()];
                    is.read(b);
                    return defineClass(name, b, 0, b.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException();
                }
            }
        };

        // 使用 ClassLoaderTest 的类加载器加载本类
        Object obj1 = ClassLoaderTest.class.getClassLoader().loadClas
s("com.jvm.classloading.ClassLoaderTest").newInstance();
        System.out.println(obj1.getClass());
        System.out.println(obj1 instanceof com.jvm.classloading.Class
LoaderTest);

        // 使用自定义类加载器加载本类
        Object obj2 = myLoader.loadClass("com.jvm.classloading.ClassL
oaderTest").newInstance();
        System.out.println(obj2.getClass());
        System.out.println(obj2 instanceof com.jvm.classloading.Class
LoaderTest);
```

```
    }
}
```

输出结果：

```
class com.jvm.classloading.ClassLoaderTest
true
class com.jvm.classloading.ClassLoaderTest
false
```

myLoader 是自定义的类加载器，可以用来加载与自己在同一路径下的 Class 文件。main 函数的第一部分使用系统加载主类 ClassLoaderTest 的类加载器加载 ClassLoaderTest，输出显示，obj1 的所属类型检查正确，这是虚拟机中有 2 个 ClassLoaderTest 类，一个是主类，另一个是 main() 方法中加载的类，由于这两个类使用同一个类加载器加载并且来源于同一个 Class 文件，因此这两个类是完全相同的。

第二部分使用自定义的类加载器加载 ClassLoaderTest，`class com.jvm.classloading.ClassLoaderTest` 显示，obj2 确实是类 `com.jvm.classloading.ClassLoaderTest` 实例化出来的对象，但是第二句输出 false。此时虚拟机中有 3 个 ClassLoaderTest 类，由于第 3 个类的类加载器与前面 2 个类加载器不同，虽然来源于同一个 Class 文件，但它是一个独立的类，所属类型检查是返回结果自然是 false。

十.简述 Java 内存分配与回收策率以及 Minor(新生代) GC 和 Major(老年代) GC

1.内存分配策略:

- 1.对象优先在堆的 Eden 区分配。
- 2.大对象直接进入老年代 .
- 3.长期存活的对象将直接进入老年代 .

2.回收策略:

当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC. Minor GC 通常发生在新生代的Eden 区，在这个区的对象生存期短，往往发生 GC 的频率

较高，回收速度比较快 .

Full GC/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC, 但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

十一.关于JVM调优

请参考博客
https://www.cnblogs.com/andy-zhou/p/5327288.html

https://www.cnblogs.com/xingzc/p/5756119.html

https://blog.csdn.net/wolf_love666/article/details/79787735