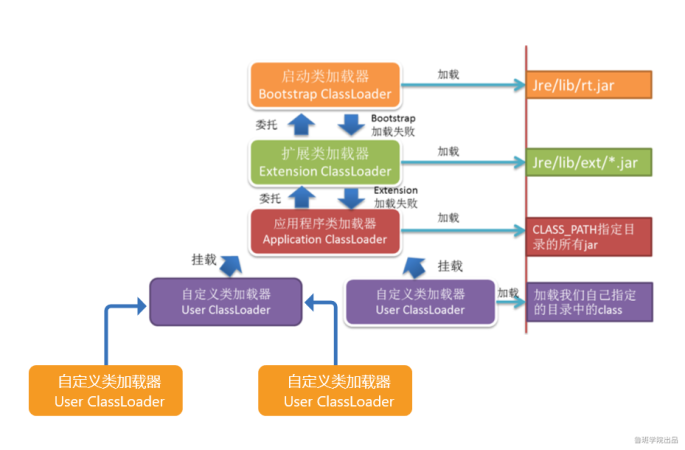
# Jvm虚拟机

HotSpot是基于c++实现，而c++是一门面向对象的语言，本身具备面向对象基本特征，所以Java中的对象表示，最简单的做法是为每个Java类生成一个c++类与之对应。

## 类加载器classLoad

作用：将java文件编译成class文件加载到Java虚拟机中。



### BootStrapClassLoader启动类加载器

1. BootStrapClassLoader隐式存在，他不存在jdk里面，他存在jvm里面，这个对象是看不到的。
2. BootStrapClassLoader启动类加载器是由C++编写的，通过Java程序去查看显示的是null，因此，启动类加载器无法被Java程序调用。
3. BootStrapClassLoader启动类加载器不像其他类加载器有实体，它是没有实体的，JVM将C++处理类加载的一套逻辑定义为启动类加载器。这套逻辑做的事情就是通过启动类加载器加载类sun.launcher.LauncherHelper，执行该类的方法checkAndLoadMain，加载main函数所在的类，启动扩展类加载器、应用类加载器也是在这个时候完成的。

### ExtClassLoader扩展类加载器

ExtClassLoader 这个对象看的到，扩展类加载器，主要加载ext包下面的文件

### ApplicationClassLoader应用类加载器

ApplicationClassLoader系统上下文class文件，项目里面的所有Class

### ContextClassLoader线程上下文类加载器

一种特殊的类加载器，可以通过Thread获取，基于此可实现向下委托加载，ServiceLoader就是ContextClassLoader。

//获取

Thread.currentThread().getContextClassLoader()

// 设置

Thread.currentThread().setContextClassLoader(Test.class.getClassLoader());

### 自定义类加载器

继承类java.lang.ClassLoader

### 双亲委派

（1）当类加载器去加载Class文件的时候，默认情况下并不会由自己去加载，而是由父类加载器去加载（双亲委派，如果父类返回空，才由他的子类处理，也就是BootStrapClassLoader永远是第一个进行处理的），上面就是类加载器的父子关系。

优点：1、保证系统里面同一个Class对象只会存在一份，所以同一个类加载器加载同一个文件多次，实际上只会加载1次。2、保证系统核心的class文件不被篡改

缺点：1、无法做到不委派 2、无法做到向下委派

（2）他们是这样做继承关系的：

Class ApplicationClassLoader{

ExtClassLoader parent;

}

### 打破双亲委派

1. 自定义类加载器去实现

因为在某些情况下父类加载器需要委托子类加载器去加载class文件。受到加载范围的限制，父类加载器无法加载到需要的文件，以Driver接口为例，由于Driver接口定义在jdk当中的，而其实现由各个数据库的服务商来提供，比如mysql的就写了MySQL Connector，那么问题就来了，DriverManager（也由jdk提供）要加载各个实现了Driver接口的实现类，然后进行管理，但是DriverManager由启动类加载器加载，只能加载JAVA\_HOME的lib下文件，而其实现是由服务商提供的，由系统类加载器加载，这个时候就需要启动类加载器来委托子类来加载Driver实现，从而破坏了双亲委派。

类似这样的情况就需要打破双亲委派。

打破双亲委派的意思其实就是不委派、向下委派

1. SPI机制

是一种服务发现机制。它通过在ClassPath路径下的resources/META-INF/services文件夹查找文件，自动加载文件里所定义的类。这一机制为很多框架扩展提供了可能，比如在Dubbo、JDBC中都使用到了SPI机制。

JDBC的驱动使用的SPI：

ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);

Iterator<Driver> driversIterator = loadedDrivers.iterator();

### 全盘委托

用来确认默认使用的类加载器（new关键字用那个类加载器），当有new关键字需要类加载器加载的时候，jvm会判断当前调用new关键字的类加载器是什么，然后用调用new关键字的类加载器加载，简单的就是说，new Test()使用的类加载器是main方法的类加载器

### 沙箱安全

比如我定义了一个类名为String所在包为java.lang，因为这个类本来是属于jdk的，如果没有沙箱安全机制的话，这个类将会污染到我所有的String,但是由于沙箱安全机制，所以就委托顶层的bootstrap加载器查找这个类，如果没有的话就委托extClassLoader, extClassLoader没有就到appClassLoader，但是由于String就是jdk的源代码，所以在bootstrap那里就加载到了，先找到先使用，所以就使用bootstrap里面的String,后面的一概不能使用，这就保证了不被恶意代码污染

### 知识点

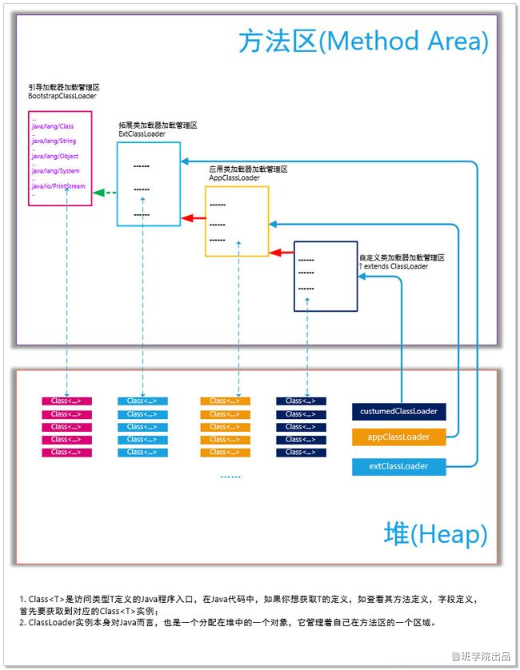
1. 同一个Class被不同的ClassLoader加载，他们的类型是不相同的。比如：

A a = classloaderA.load

A b = classloaderB.load

a = b（报错，类型不相同）

1. ClassLoader.loadClass 是从缓存中获取Class对象
2. ClassLoader.defineClass把Class文件转换成Class对象，并放到缓存中
3. hashCode是内存地址
4. 不同的类加载器加载同一个类，这个class相等吗？不相等，因为每个类加载器在方法区中都有各自独立的内存空间来存放各自的class类，如图：类加载器加载的类如何存储



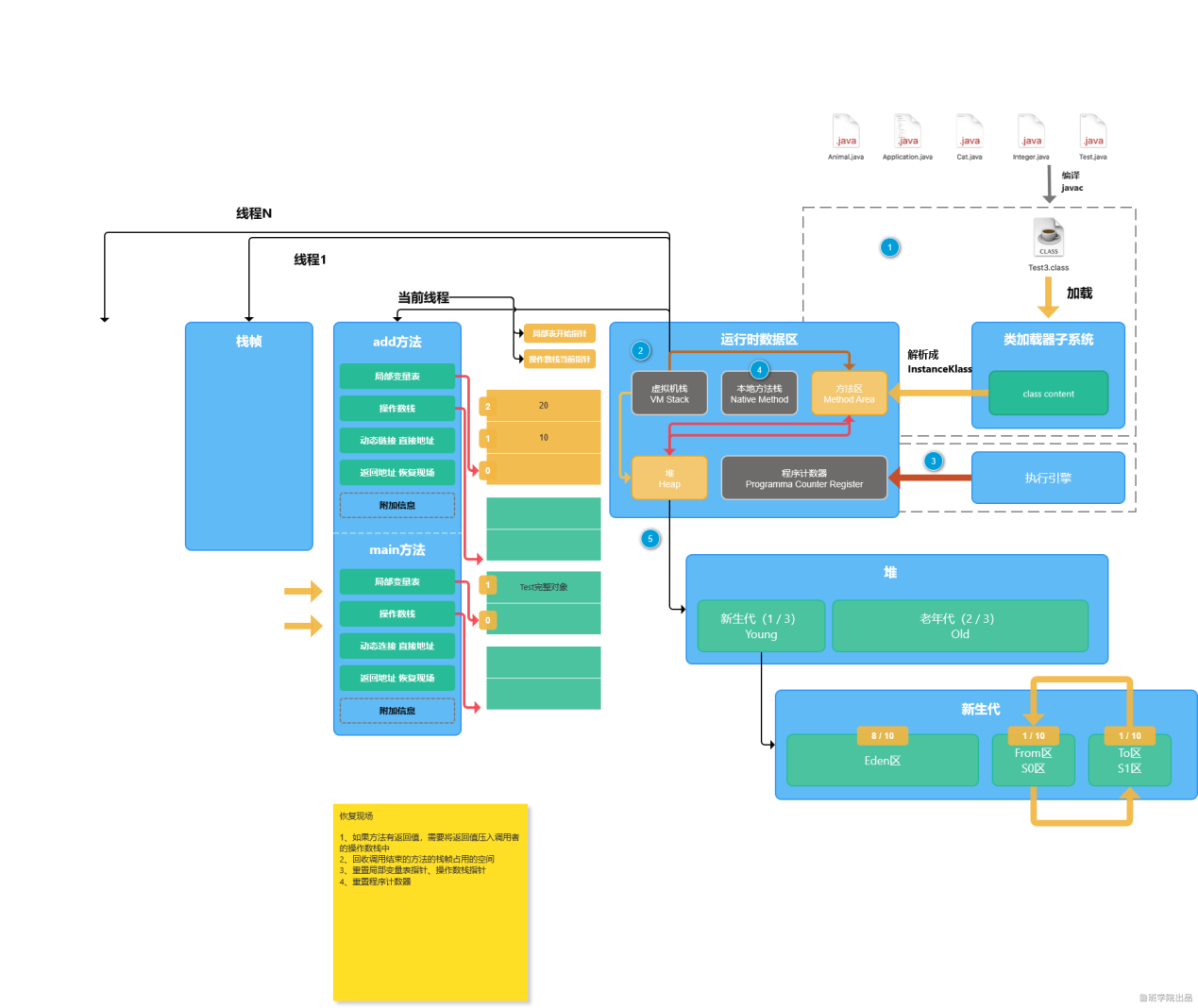
## JIT及时编译

java的编译机制的时候，分为两个部分。

一个是javac指令将java源码变为java字节码的静态编译过程。另一个是java字节码编译为本地机器码的过程，并且因为这个过程是在程序运行时期完成的所以称之为即时编译。

## JVM内存区域（JVM不是java写的，是openJDK写的）

在JVM中CheapObj、ValueObj、AllStatic这3个类的子类组成了Jvm内存模型。



4个名词

（1）Class文件：硬盘上.class文件

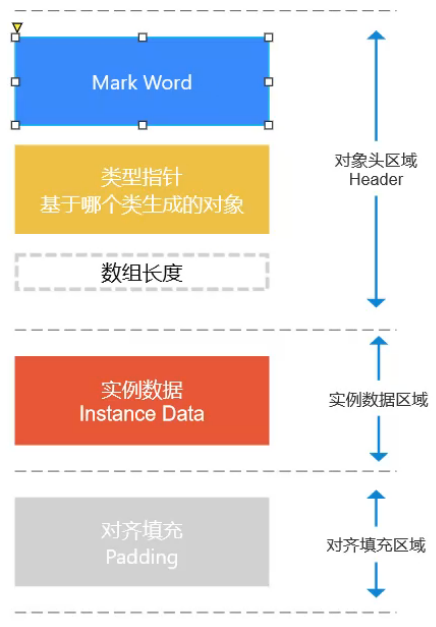
（2）Class Content：类加载器将.class文件加载到内存，存储字节码文件数据的那块区域。类在加载过程中，方法区中的ApplicationClassLoader首先将class解析成InstanceKlass类，用来存放类的元信息，并将信息保存到方法区中，然后在堆区生成InstanceMirrorKlass镜像类，用来存放class对象。

（3）Class对象：Class<?> clazz = Test.class，在JVM中真正获取到的是InstanceMirrorKlass实例。

（4）对象：Test obj = new Test()

虚拟机栈、方法区、堆区关系

1. 虚拟机栈指向方法区。动态链接
2. 虚拟机栈指向堆区。Test obj = new Test()
3. 方法区指向堆区。引用类型的静态属性
4. 堆区指向方法区。对象的内存布局Klass pointer，它会指向该对象的instanceKlass实例。比如obj对象想要找到Test对象的instanceKlass类的元信息，需要通过在对象的对象头区域中类型指针Klass pointer，它永远指向instanceKlass类的元信息（存放在方法区），简单理解就是obj所属的是Test类，Test类在方法区生成的是instanceKlass实例



### 方法区（信息共享，其他区可以随时使用）

存放类信息（.class文件）、常量、static变量（1.8以前在方法区，以后在堆区，因为它存储在InstanceMirrorKlass中）、JIT编译后的代码。

方法区是规范，元空间、永久代是具体实现，方法区可以简单理解为是一个理论。

永久代，1.8以前的方法区实现，存放在堆区；元空间，1.8以后方法区实现，存放在直接内存。

### java堆区（信息共享，其他区可以随时使用）

存放实例对象（同时也会本地缓存一个Class引用，当类加载器去读取的对象的时候，会先去缓存里找引用），GC（垃圾回收器）主要操作这个区域。

最小内存-Xms：物理内存的1/64

最大内存-Xmx：物理内存的1/4，最大和最小最好调成一样大小

什么对象会进入老年代？

1、15次gc仍然存活的对象。这里次数只能<=15，因为分代年龄占4个bit，所以只能0-15



2、大对象。对象大小超过Eden区的一半，Eden区不是固定的，Eden区大小是在运行期动态调整的

3、空间担保。针对Eden区，当gc后Eden区还剩下的对象不足以存放到From/To区，就将剩下的对象全部放到老年代中

4、动态年龄判断。针对Eden区+From区，当gc后Eden区和From区存放不下对象，会将剩下的对象放到To区，但是不足以存放到To区，就将剩下的对象全部放到老年代中

### VM stack（虚拟机栈）

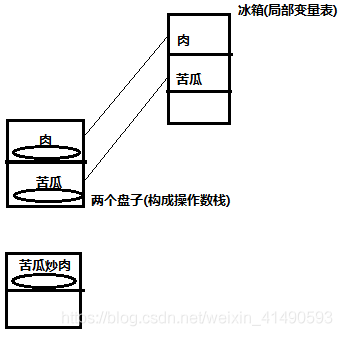
Java方法在运行时的内存模型，用来存放栈贞的一个区域，拟机栈主要使用的是栈帧，每一个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。栈贞里面存放的是局部变量、操作数栈、动态链接/直接地址、返回地址/恢复现场、附加信息。

一个线程一个虚拟机栈；方法被调用几次，虚拟机栈中就有多少个栈帧。

（1）局部变量表：存放局部变量的表，LocalVariableTable。

（2）操作数栈：存储操作数的栈，可以理解等号后面的操作，比如push、pop。

大白话理解局部变量表和操作数栈：首先从冰箱(局部变量表)里取出肉和苦瓜放在两个盘子(两个盘子构成的操作数栈)里,然后经过放在锅里炒后,又放回栈里(其中一个盘子),最后可以把抄好的苦瓜炒肉放到冰箱第三格存起来改天吃。



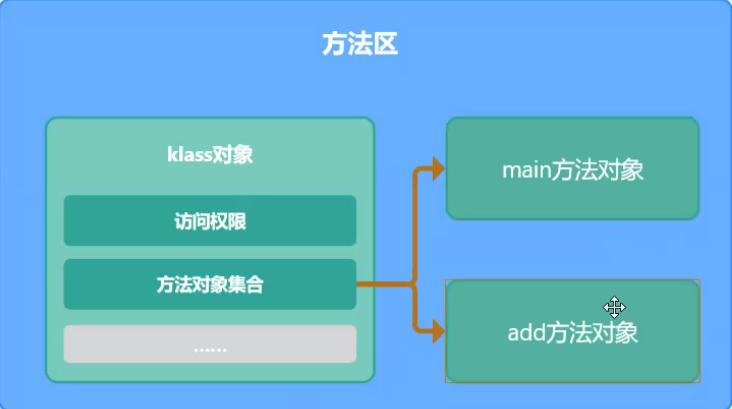
（3）动态链接：main方法对应的jvm对象在元空间中的内存地址。

List<MethodObject \*>方法对象集合

MethodObject \*mainSrc

MethodObject \*addSrc

\*mainSrc这个就是方法的动态链接，它指向的是一个内存地址



（4）返回地址/恢复现场：add方法执行完后，需要回到main方法中来，因此add栈帧中需要存放main方法的许多指针信息，不然这些信息不存放会导致add执行完方法后不知道回到main方法程序的哪一步。

JVM执行main()方法，内部做了那些事情。

1、创建main方法的栈帧

2、将main方法的操作数栈指针赋值给线程的属性：操作数栈指针

3、将main方法的局部变量表指针赋值给线程的属性：局部变量表指针

JVM运行add方法，内部做了那些事情。

1. 创建add方法的栈帧
2. 在add方法的栈帧中保存main方法的字节码下一行程序计数器
3. 线程的局部表开始指针（main的）保存至add方法的栈帧中
4. 线程的操作数栈开始指针（main的）保存至add方法的栈帧中
5. 将add方法的局部表指针赋值给线程的局部表指针
6. 将add方法的操作数栈指针赋值给线程的操作数栈指针
7. Test test = new Test()需要经过如下几个步骤

0 new #2 <com/luban/ziya/jvm/Test>

1、在堆区生成了一个IntanceOopDesc对象（不完全对象，未执行构造方法）

2、将不完全对象的指针压入栈

3 dup

duplicate 复制的简写

1、复制栈顶元素

2、压入栈

为什么要dup？

答：将对象指针作为this进行传参

4 invokespecial #3 <com/luban/ziya/jvm/Test.<init>>

非静态方法，第一个参数就是this指针，dup是将this指针压入栈，然后在这一步完成this指针赋值

1、执行invokespecial字节码指令，完成运行方法的环境构建(在这过程中完成this指针赋值)

2、执行完成构造方法，这个对象就是完全对象

7 astore\_1

1、将栈顶元素pop出栈

2、将完全对象的地址赋值给局部变量表index=1的位置，也就是给Test完成赋值

### PC（程序计数器）

Java线程的私有数据，这个数据就是执行下一条指令的地址，告诉我们程序执行到哪一步，其实也就是字节码索引。PC的主要功能是用于存放指令的地址。

### Nativemethodstack（本地方法栈）

Java线程的私有数据，与JVM虚拟机的native方法有关。

Java调用c、c++的动态链接库，运行里面的栈

### 元空间如何调优

查看jvm内存：java –XX:+PrintFlagsFinal –version | grep Metaspace

MetaspaceSize最小内存空间 MaxMetaspaceSize最大内存空间

1. 堆区最大和最小调成一样大，防止内存抖动
2. 调成物理内存的1/32
3. 使用调优工具：visualVM、arthas

## Java内存模型 JavaMemoryModel JMM（规范，抽象的模型）：

(1)主内存：共享信息

(2)工作内存：私有信息

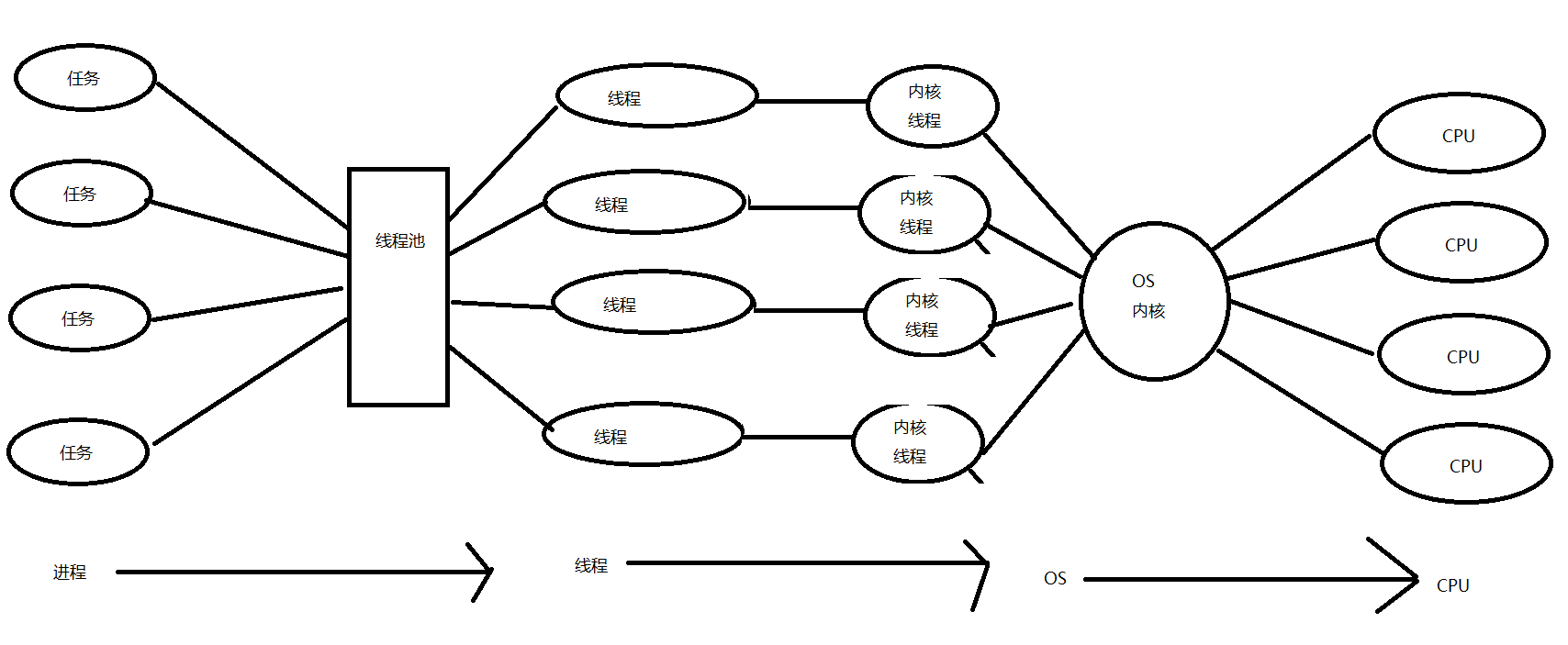
(3)工作方式：

现场修改私有数据，直接在工作空间修改

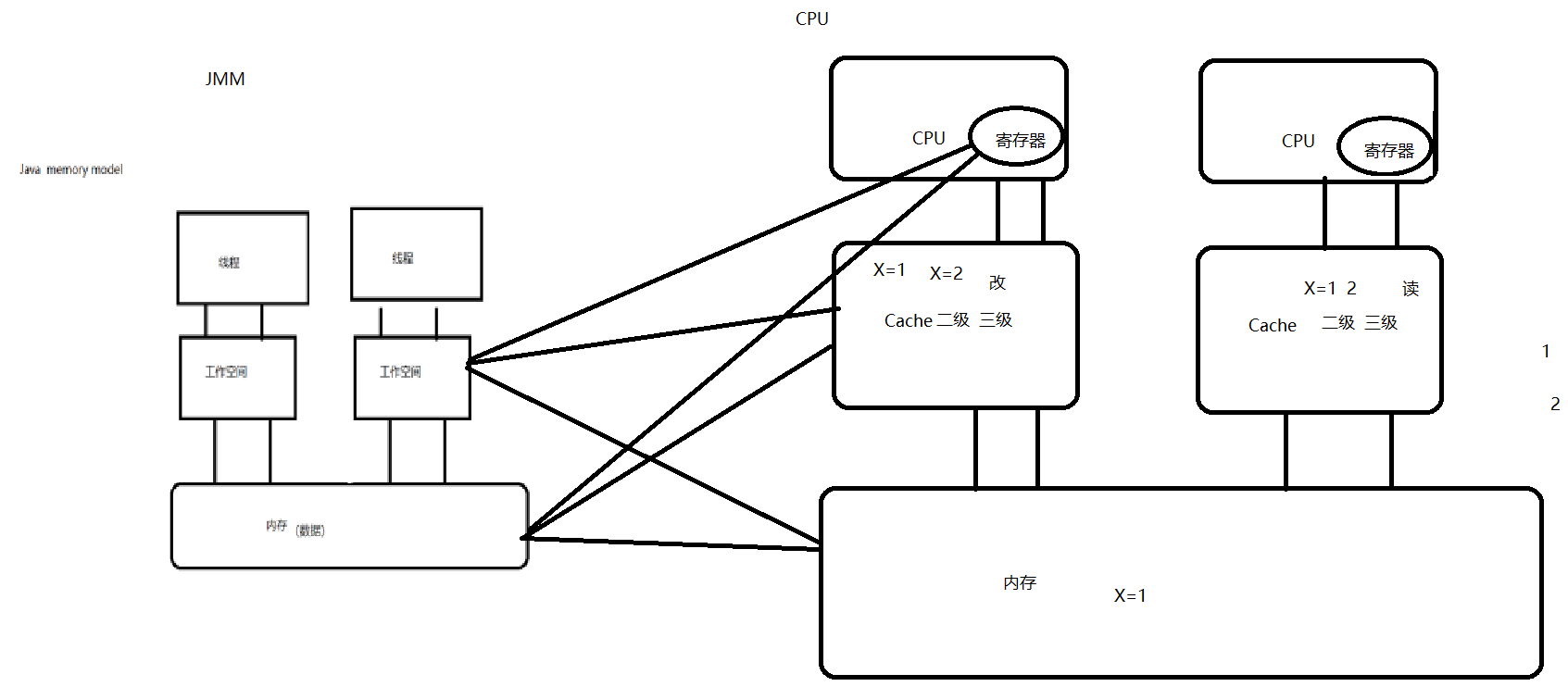
现场修改共享数据，把数据复制到工作空间中，在工作空间中修改，修改完成以后，刷新内存中的数据

Java内存和Java虚拟机内存的关系：Java内存是物理内存，Java虚拟机内存是逻辑内存。物理上的内存划分出Java虚拟机逻辑上的内存，然后在不同的逻辑内存里面完成不同的事情

## Java线程与硬件处理器交互流程



## Java内存模型与硬件内存架构的关系



# JVM-klass

Java类的元信息在JVM中的存放形式

## klass模型类的继承结构

简单理解：java文件用Class进行存储的，而JVM是用Klass进行存储。

Klass：可以理解为在C++中用来存储class信息。

InstanceKlass：在jvm中用来存放java类，也就是用来存储java的元信息（非数组），它主要是将java类的实例存放到方法区中。

InstanceMirrorKlass（镜像类）：在jvm中用来存放class对象，它存放在JVM堆区

InstanceRefKlass：引用，JVM通过引用来找到java元信息

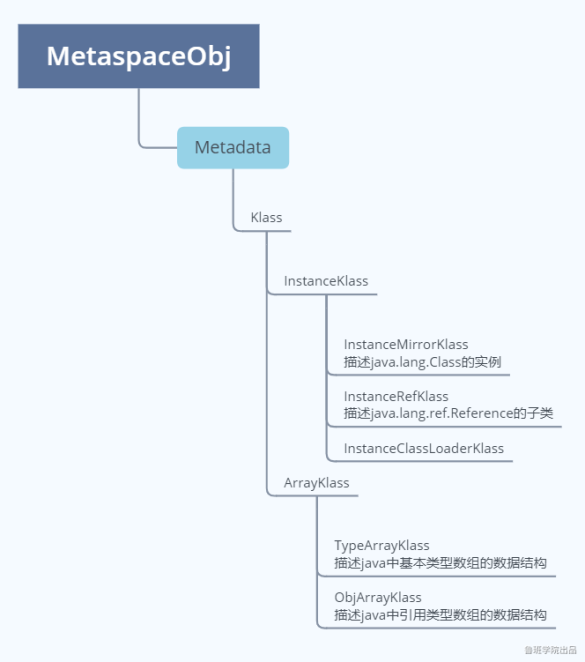
ArrayKlass：存储数组类的元信息，Java中的数组不是静态数据类型，是动态数据类型，即是运行期生成的，Java数组的元信息用ArrayKlass的子类来表示

TypeArrayKlass：java中基本类型的数组在jvm中存在的形式，jvm通过newarray字节码指令来找到对应的Klass。例： int[] arr = new int[1] --> newarray --> TypeArrayKlass

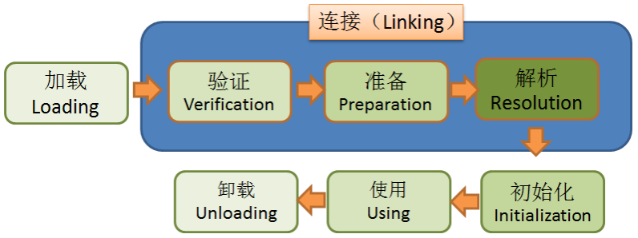
ObjArrayKlass：java中引用类型的数组在jvm中存在的形式，jvm通过anewarray字节码指令来找到对应的Klass。例： Test[] arr = new Test[1] --> anewarray--> ObjArrayKlass

newarray：这个是jvm字节码指令，创建一个指定原始类型（如int, float, char…）的数组，并将其引用值压入栈顶

anewarray：这个是jvm字节码指令，创建一个引用型（如类，接口，数组）的数组，并将其引用值压入栈顶

小结：类加载器将class文件加载到系统，并将class文件解析成元信息（类名、属性名、属性签名、方法名等），这些元信息在jvm中用对应的Klass进行存储。

## 类的加载过程



### 加载

加载主要干了哪些事情？

1、通过类的全限定名获取存储该类的class文件（没有指明必须从哪获取）

2、解析成运行时数据，即instanceKlass实例，存放在方法区

3、在堆区生成该类的Class对象，即instanceMirrorKlass实例

注：jvm是用c++写的，其实可以用任何语言去写，只要能做到这3件事情，就能实现类加载。

何时加载？

Jvm采用懒加载模式，根类加载器BootStrapClassLoader加载类的时候，会先将常用的类进行预加载（String、Thread），其他的类只会在使用的时候才会加载。

（1）主动使用时

1、new、getstatic、putstatic、invokestatic

2、反射

3、初始化一个类的子类会去加载其父类

4、启动类（main函数所在类）

5、当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getstatic,REF\_putstatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化

（2）预加载： String、Thread、Integer

案例：

1. 子类调用父类静态变量，此时会自动调用父类静态代码块，不会调用子类静态代码块。
2. 子类调用父类变量，则会调用子类静态代码块，因为想要使用父类变量，必须实例化子类才行（主动使用子类，间接在使用父类）。
3. 调用被final修饰过的静态常量，不会执行静态代码块，因为此时jvm会将这个常量直接写入到调用类的常量池中，供调用类可以直接使用，不执行被调用类的静态代码块代码。

静态字段如何存储？

JDK8静态属性是存储在镜像类中instanceMirrorKlass，不存在InstanceKlass中，在字节码中它在oop Klass下（oop Klass类的内存地址）。

父类的静态是如何被子类调用的？

1. 逐级往上找
2. 借助另外的数据结构实现，使用K-V的格式存储，Hotspot就是使用的第二种方式，借助另外的数据结构ConstantPoolCache，常量池类ConstantPool中有个属性\_cache指向了这个结构。每一条数据对应一个类ConstantPoolCacheEntry。

内部类在加载阶段会被加载到jvm中，但内部类的静态变量在加载阶段不会被实例化，只有在被主动调用的时候才会实例化。

因为没有指明必须从哪获取class文件，脑洞大开的工程师们开发了这些

1、从压缩包中读取，如jar、war

2、从网络中获取，如Web Applet

3、动态生成，如动态代理、CGLIB

4、由其他文件生成，如JSP

5、从数据库读取

6、从加密文件中读取

### 验证

1、文件格式验证

2、元数据验证

3、字节码验证

4、符号引用验证

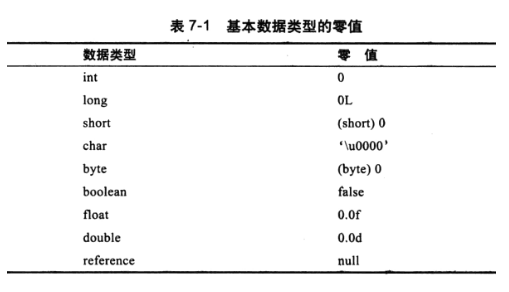
### 准备

为静态变量分配内存、赋初值

注：

1. 不过实例变量是在创建对象的时候完成赋值的，没有赋初值一说。
2. 如果被final修饰，在编译的时候会给属性添加ConstantValue属性，准备阶段直接完成赋值，即没有赋初值这一步。

例：public static final int a = 10和public static int b = 10在准备阶段a=10，b=0



long、double：64 bit

int、float：32 bit

char、short：16 bit

byte：8 bit

boolean：1 bit

### 解析

将编码过程中的间接引用（符号引用，指向运行时常量池引用）转为直接引用（内存地址）

例：将#2 = Class #32 //com/xiaobi/Test\_1中的#32改为@0x0000007c0060880

解析后的信息存储在ConstantPoolCache类实例中

1、类或接口的解析

2、字段解析

3、方法解析

4、接口方法解析

何时解析

思路：

1、加载阶段解析常量池时

2、用的时候

openjdk是第二种思路，在执行特定的字节码指令之前进行解析：

anewarray、checkcast、getfield、getstatic、instanceof、invokedynamic、invokeinterface、invokespecial、invokestatic、invokevirtual、ldc、ldc\_w、ldc2\_w、multianewarray、new、putfield

### 初始化

执行静态代码块，完成静态变量的赋值

1、静态字段、静态代码段，字节码层面会生成clinit方法

2、方法中语句的先后顺序与代码的编写顺序相关

执行结果：a = 1，b = 2

Public staic int a;

Public static int b = 1;

Static{

a++;

b++;

}

执行结果：a = 1，b = 1 因为a在初始化阶段被赋值为0，然后加加为1，而b在执行静态代码块代码之后，被b=1覆盖了

Public staic int a;

Static{

a++;

b++;

}

Public static int b = 1;

### SystemDictionary

保存系统已加载的class信息，反射获取到的类信息是从这里得到的。

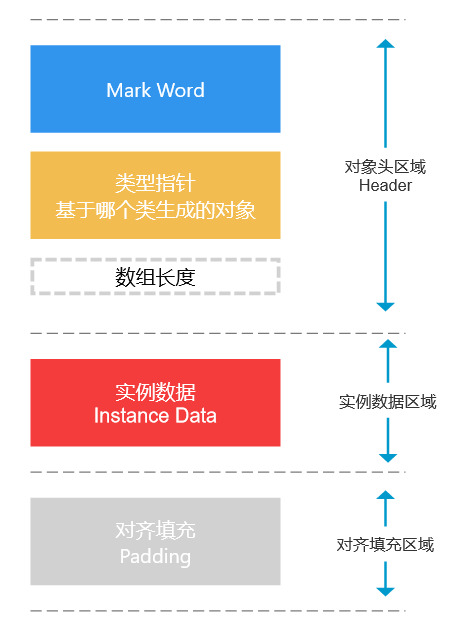
this.findLoadedClass从SystemDictionary这里获取到class的缓存信息。

# JVM-OOP

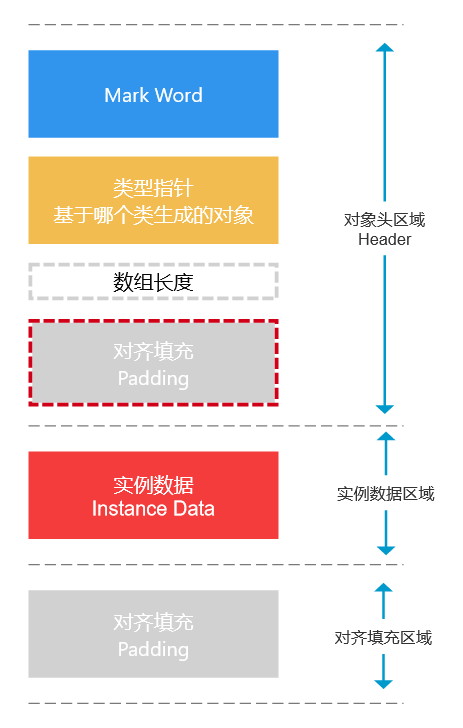
Java中的对象在JVM中的存在形式

## 对象的内存结构（对象的内存布局）

### 内存结构一



### 内存结构二



### 对象头

1. Mark Word：32位4字节,64位8字节
2. 类型指针Klass pointer：对象所属类的元信息的实例指针，instanceKlass在方法区的地址。指针压缩开启占4字节，关闭8字节。
3. 数组长度，对象是数组占4字节，不是数组，占0字节

### 实例数据

类的非静态属性，生成对象时就是实例数据。比如：public class Test{ int a = 10}

|  |  |
| --- | --- |
| 对象属性类型 | 字节 |
| boolean | 1 B |
| byte | 1 B |
| char | 2 B |
| short | 2 B |
| int | 4 B |
| float | 4 B |
| double | 8 B |
| long | 8 B |
| 引用类型 | 开启指针压缩4 B  关闭指针压缩 8 B |

### 对齐填充

Java中所有对象大小都是8字节对齐（8的整数倍）。

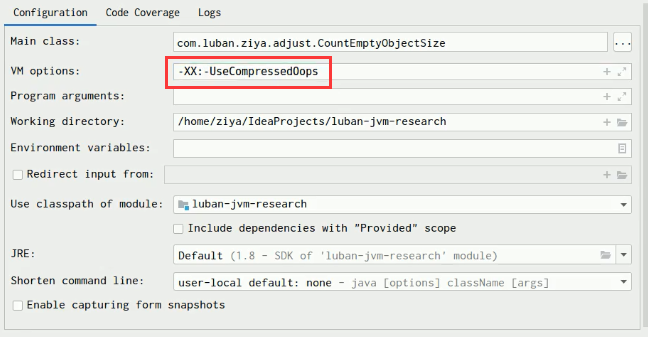
如果一个对象占30字节，JVM底层会补2字节（对齐填充），凑成32字节，达到8字节对齐。

## 指针压缩

JVM开启指针压缩，会将一个指针由8字节压缩成4字节，此时指针表示的不是引用对象的实际内存地址，是一个相对的地址，压缩的是这个指针。

开启：-XX:+UseCompressedOops JDK6以后默认开启

关闭：-XX:-UseCompressedOops



指针压缩为了节省空间，提高寻址效率。它有2个特点：存储时，后三位0抹除；使用时，后三位补0，因为java中所有的对象都是8字节对齐的。

注意：二进制是从右到左进行计算。

（1）一个oop能表示的最大堆空间是多少？ 答：32G

32位操作系统可以寻址到最大内存是4G，2的32次方。

32G = 2的35次方，因为使用的时候，尾部补了三个0，它的指针最大只有35位，所以是32 +3次方。

111111......111 000 –> 最左边是2的35次方

（2）32G堆瓶颈，32G不够用了，如何扩容？

8字节对齐改成16字节对齐（将java中所有的对象改为16字节对齐的），也就是32 \* 2 = 64G，因为使用时，尾部会补4个0

（3）JDK底层为什么没用16字节对齐

GC算法处理32G的堆已经是极限了，因为CPU运算能力有限。在清理内存时，会大量使用CPU性能。

## 对象计算

### 对象-没有属性

开启指针压缩：8 + 4 + 0 + 0 + 对齐填充

关闭指针压缩：8 + 8 + 0 + 0 + 对齐填充

### 对象-基本属性

开启指针压缩：8 + 4 + 0 + 属性类型字节\*个数 + 对齐填充

关闭指针压缩：8 + 8 + 0 + 属性类型字节\*个数 + 对齐填充

### 对象-数组属性

开启指针压缩：8 + 4 + 4 + 数组属性类型字节\*数组长度 + 对齐填充

关闭指针压缩：8 + 8 + 4 + 对齐填充 + 数组属性类型字节\*数组长度 + 对齐填充

注意：数组对象，在关闭指针压缩的情况下会出现两段填充

# 字节码

|  |  |  |
| --- | --- | --- |
| 二进制 | 十进制（从右往左算） | 十六进制 |
| 1101 | 1+4+8 = 13 | d（10-15：a-f，逢十六进一） |

## 大端与小端

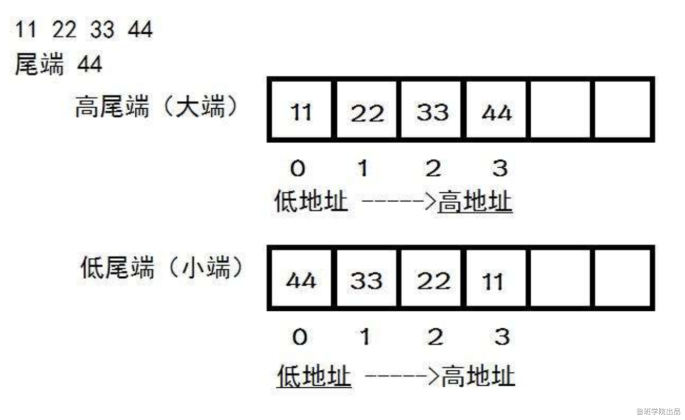
计算机的大端和小端模式存储没区别，区别在于读取的方式不同，小端是从高地址往低地址读取，大端模式是从低地址往高地址读取。简单理解为大端是从左往右读，小端是从右往左读。例如：0000010 : ca fe ba be 小端读取：be ba fe ca 大端读取：ca fe ba be

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ca fe ba be | | 大端  写：低地址存高位,高地址存低位  读：从低地址往高地址读 | | 小端  写：低地址存低位,高地址存高位  读：从高地址往低地址读 | |
|  |  | 存 | 读 | 存 | 读 |
| 低地址 | 0x01 | ca | ca fe ba be | be | be ba fe ca |
|  | 0x02 | Fe |  | ba |  |
|  | 0x03 | ba |  | fe |  |
| 高地址 | 0x04 | be |  | ca |  |

大端：数据的高字节（高位）保存在内存的低地址中，数据的低字节（低位）保存在内存的高地址中。网络传输是大端模式存储

小端：数据的低字节（低位）保存在内存的低地址中，数据的高字节保存在内存的高地址（高位）中。内存和主机存储是小端模式存储的。

11是高字节，44是低字节



## 字节码文件组成

ca fe ba be 00 00 00 34 00 19 0a 00 04 00 15 09

00 03 00 16 07 00 17 07 00 18 01 00 01 61 01 00

U1：占1个字节

U2：占2个字节

U4：占4个字节



### 魔数

判断一个文件是不是一个合格的class文件，如果不是，就不执行，通过ca fe ba be来进行判断，占4个字节

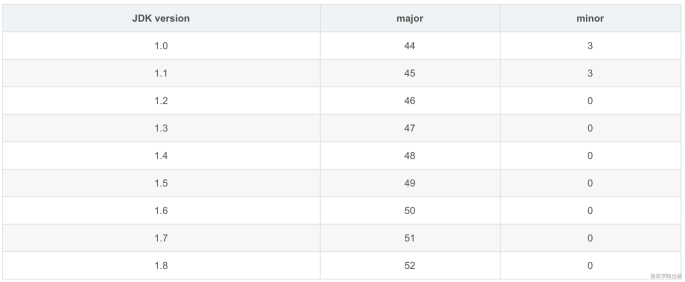
次版本号

16进制，占2个字节，00 00

主版本号（JDK 1.8）

16进制，占2个字节，00 34

转成10进制：3\*16 + 4 = 52



### 常量池

常量池有3种

1. class中的常量池 静态
2. 运行时常量池 动态
3. 字符串常量池 StringTable

常量池个数：16进制，占2个字节，00 19转成10进制：1\*16 + 9 = 25

常量池最小的index是1,所以真实的常量池个数=字节码文件中的常量池个数-1

解析常量池（对照这下面的图来解析常量池）

这里其实是读一个字节去解析，转换成10进制之后，找到对应的含义是什么，然后存放到对应的常量中，比如：第一个元素中的0a，会被存放到CONSTANT\_METHODREF\_INFO这个常量池类型中去，00 04指向第4个元素

第一个元素/第一个常量池

tag：0a 转成10进制：10 method

class\_index：00 04 转成10进制：4 字节码文件在解析的时候，并不知道这里是什么，所以这里放的是符号引用，后面要转直接引用，这里含义是指向第4个元素

name\_and\_type\_index：00 15 转成10进制：21

第二个元素

tag：09 转成10进制：9 filed

class\_index：00 03 转成10进制：3 符号引用，这里的3指向的是第三个元素

name\_and\_type\_index：00 16 转成10进制：22

第三个元素

tag：07 转成10进制：7 class

class\_index：00 17 转成10进制：23

第四个元素

tag：07 转成10进制：7 class

class\_index：00 18 转成10进制：24

第五个元素

tag：01 转成10进制：1 utf8

length：00 01 转成10进制：1 占1个字节

bytes：61 转成10进制：6\*16 +1 = 97 ascii的97为a



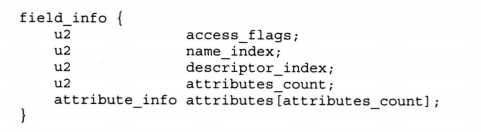
### access\_flag



### 接口

一个类最多实现65535个接口，如果在interfaces\_count区域中的接口数量为0，则interfaces[]这块区域不会在字节码文件中出现。

### 解析属性



第一个属性

access\_flags：00 08 static

name\_index：00 05 转成10进制：5 符号引用，这里的5指向的是第五个元素a

descriptor\_index：00 06 描述符是I代表int

attributes\_count：00 00 属性的属性数量，如果被final修饰，在编译的时候会给属性添加ConstantValue属性

attributes：属性内容，如果属性的数量为0，这块区域就不会出现在字节码文件中

（1）数据类型的描述符

byte描述符是B

void描述符是V

char描述符是C

double描述符是D

引用类型描述符是L

String描述符是Ljava/lang/String;

数据类型描述符是[

byte[]描述符是[B

String[]描述符是[Ljava/lang/String;

（2）方法的描述符

（数据类型的描述符）返回值的描述符

比如：public static void mian(String[] args)

描述符：([L java/lang/String;)V

public static String a(String[][] a,int b,Ziya c)

描述符：([[Ljava/lang/String;,I,Ljava/luban/Ziya;) L java/lang/String;

### 解析方法

结构如下：

Method

Code

LineNumberTable：能检测代码在第几行错误

LocalVariableTable：存储参数、局部变量

Exception

方法解析过程

（1）方法数量methods\_count

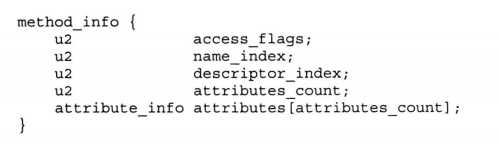
00 03 转换成10进制为3，代表有3个方法，分别是clinit、init、main这3个方法

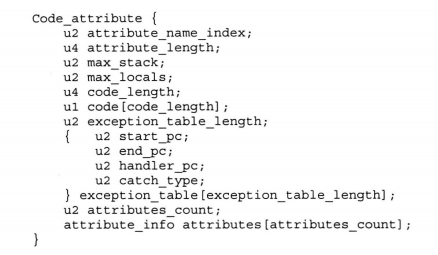
clinit：类中有静态属性或者静态代码块，编译时会自动生成这个方法，只会有1个

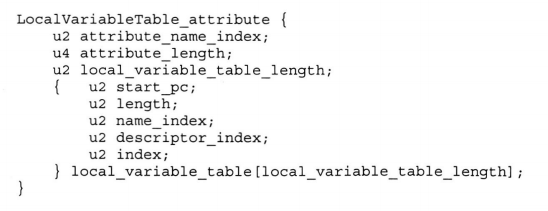
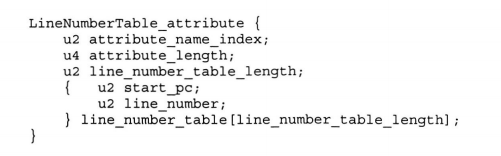
init：默认构造方法

main

（2）方法信息methods\_info







第一个方法

Access\_flags：00 01

Name\_index：00 07

Descriptor\_index：00 08

Attributes\_count（属性数量，Code、Exception）：00 01

Code\_Attribute：00 09，如果属性为0，字节码文件中就没有这个区域

Attribute\_name\_index：00 09（code）

Attribute\_length：00 00 00 2f 方法总共占47个字节

Max\_stack（操作数栈大小）：00 01

Max\_locals（局部表大小）：00 01

Code\_length：00 00 00 05

Code（方法体、字节码指令）：2a b7 00 01 b1 b1代表return

Excetion\_table\_length：00 00

Exccetion\_table：由于异常数量为0，所以这块区域没有数据

Attributes\_count（Code属性的属性）：00 02

LineNumberTable\_attribute{

Attribute\_name\_index

Attribute\_length

Line\_number\_table\_length

Start\_pc

Line\_number

}

LocalVariableTable\_attribute{

Attribute\_name\_index

Attribute\_length

local\_variable\_table\_length

Start\_pc

Length

Name\_index

Descriptor\_index

index

}

# 字符串

Klass体系是java中类在JVM中存在的形式，oop体系是java中对象在JVM中存在的形式。其实都是C++的数据结构。

## 常量池

### Class文件中的常量池

汇编指令里的Constant pool，它存放在硬盘上。

Javap –verbose：javap是jdk自带的反解析工具。它的作用就是根据class字节码文件，反解析出当前类对应的code区（汇编指令）、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。

### 运行时常量池

我们常说的常量池指的就是运行时常量池。

它是C++中InstanceKlass的一个属性ConstantPool\* \_constants ，它存放在方法区

### 字符串常量池

即String Pool，但是JVM中对应的类是StringTable，底层实现是一个hashtable

存放在堆区

String有时候会存放到StringTable，有时候不会

（1）Hashtable是以什么形式存储字符串的？

字符串name = “xiaobi”

Hashtable以key/value形式存储数据key = name value = xiaobi

（2）Hashtable结构

数组+链表

数组存放key通过hash算法计算出的hashValue值

链表存放key/value数据

（3）Hashtable是如何查找字符串的？

1、将key通过hash算法计算出的hashValue，比如name的hashValue = 11

2、在hashtable的数组中，根据index=11去匹配元素，只有1个则返回

3、如果是多个，根据链表进行遍历，比对key，返回匹配的元素

1. StringTable的key和value的生成方式

Key：

hashValue = hash\_string(name,len)

index = hash\_to\_index(hashValue)

1. 根据字符串已经字符串长度计算出hashValue
2. 根据hashValue计算出index，这个index就是key

Value：

HashtableEntry<oop, mtSymbol>\* entry = new\_entry(hashValue, string());

将Java的String类的实例instanceOopDesc封装成HashtableEntry。instanceOopDesc就是java中的String对象在JVM中存在的形式（oop体系是java中对象在JVM中存在的形式）。

## 字符串数组

### 字符串数组在JVM中是怎样存储的？

字符串在元信息存储在TypeArrayKlass中，生成的对象存储在TypeArrayOopDesc中。

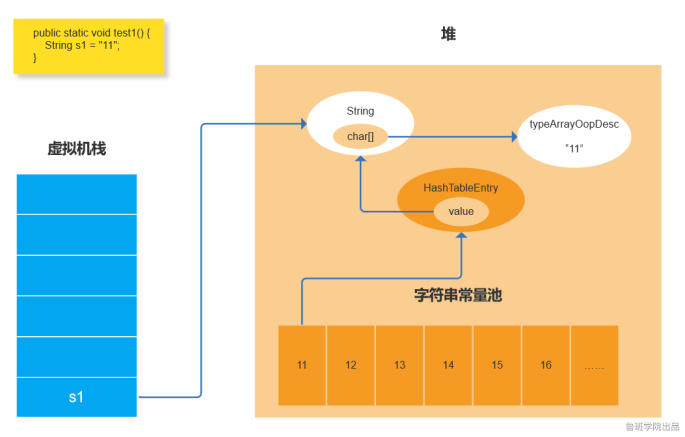
## HashCode生成方式

|  |
| --- |
| String的hashcode与String的内容有关，相同的内容，他的hashcode相同  public int hashCode() {  int h = this.hash;  if (h == 0 && this.value.length > 0) {  char[] val = this.value;  for(int i = 0; i < this.value.length; ++i) {  h = 31 \* h + val[i];  }  this.hash = h;  }  return h;  } |

## 不同方式创建字符串在JVM中的存在形式

### （1）String s1 = “1” 生成了几个oop，分别是什么？

生成了2个oop，分别是TypeArrayOopDesc（char数组）和instanceOopDesc（String对象），图片白色的部分才是oop。HashTableEntry是C++对象不是oop

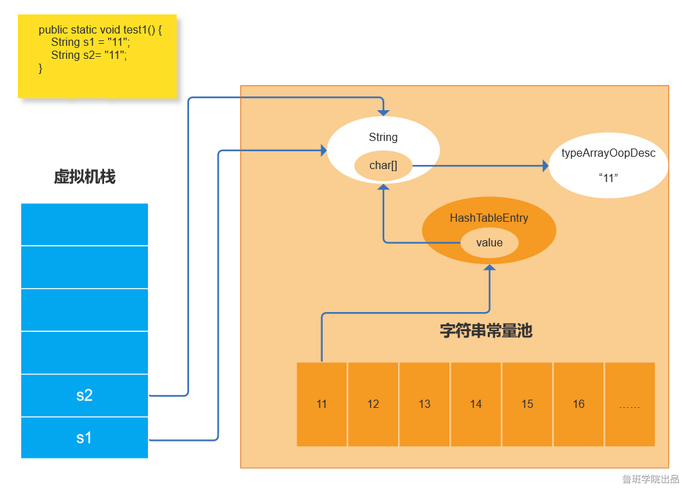


### （2）String s1 = “1”和String s2 = “1”一起生成了几个oop对象，几个String对象？

生成了2个oop对象，因为s1和s2指向同一个String对象对应的instanceOopDesc

生成了1个String对象

注：如果内容不同，则生成的oop和String互不干扰，不会重复指向。

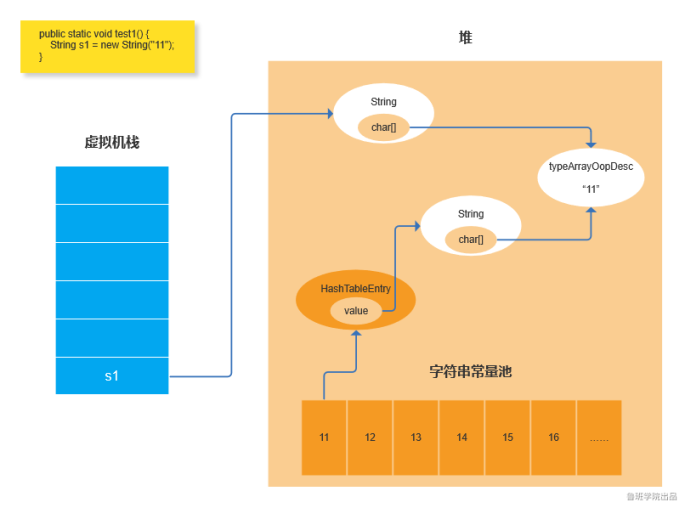


### （3）String s3 = new String(“2”) 生成了几个oop，分别是什么？

生成2个String和3个oop， 2个instanceOopDesc和1个TypeArrayOopDesc，因为new String也要生成一个instanceOopDesc。

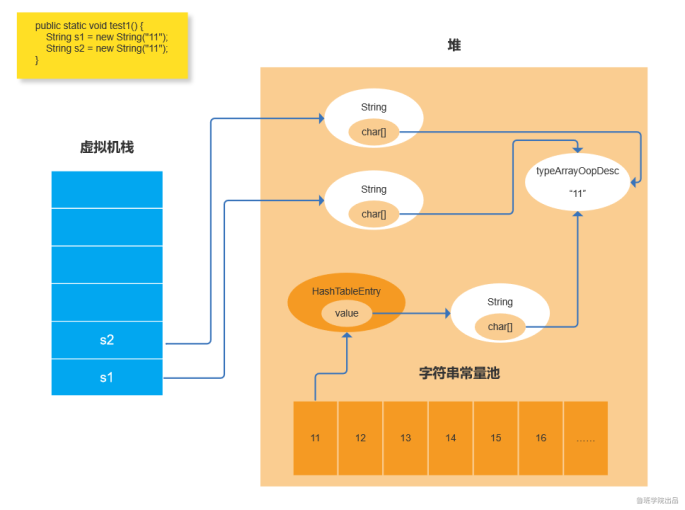
里面做了些什么动作？

1. 去字符串常量池中查，有则直接返回对应的String对象，没有则创建String对象和char数组对象
2. 将String对象对应的instanceOopDesc封装成HashTableEntry，作为StringTable的value进行存储
3. new String又在堆区创建一个对象，char数组指向已经创建好的typeArrayOopDesc对象



### （4）String s3 = new String(“2”)和String s4 = new String(“2”) 生成了几个oop？

生成4个oop，3个instanceOopDesc和1个TypeArrayOopDesc，因为有2个new String



### （5）String s1 = “1” String s2 = “1” String s3 = s1+s2创建了几个oop对象？几个String对象？

4个oop对象，2个instanceOopDesc（String对象）、2个TypeArrayOopDesc（char数组对象）。

String s1 = “1”：1个char，1个String

String s2 = “1”：0个char，0个String

String s3 = s1+s2：1个char，1个String

重点：上面的代码相当于new StringBuilder().append(“1”).append(“1”).toString()，因为toString()方法调用new String(this.value,0,this.count) -> new String(new char[]{‘1’,’1’},0,2) 函数，他会生成1个char、1个String，所以是4个oop对象。

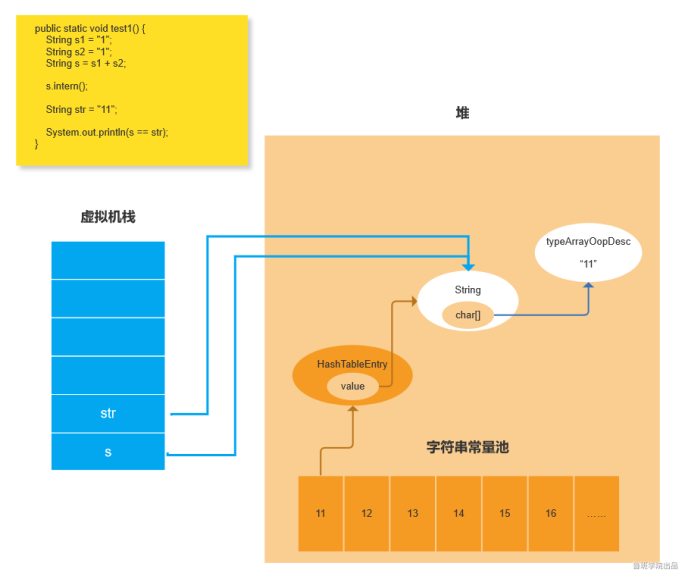
### （6）String s1 = new String(new char[]{‘1’,’1’},0,2)几个oop？几个String？

1个String，2个oop(1个instanceOopDesc，1个TypeArrayOopDesc)，它没有在常量池中生成记录。

### （7）intern做了什么

常量池中有就直接返回，常量池中没有就创建一个，然后返回。

不加intern结果为false，加了结果为true



### （8）final影响字符串比较

加了final的基本类型变量相加，会固定相加后的值，并在常量池中新建常量。

加了final的引用类型变量相加，固定的是对象的指针。

final String s1 = “1”; final String s2 = “1”;String str = “11”;

str == s1+s2 -> true

final String s1 = new String(“1”); final String s2 = new String(“1”);String str = “11”;

str == s1+s2 -> false

# 执行引擎

## 什么是执行引擎？

JVM运行Java程序的一套子系统

## JDK的两种解释器

### 字节码解释器

Java字节码 -> C++代码 -> 硬编码

字节码解释器就是一步步执行字节码，然后找到对应的C++代码，最后执行字节码对应的硬编码。

（1）Java代码：public static void main(String args)

（2）Java字节码：使用javac将java文件编译成class文件，然后使用Javap反汇编器将class文件，翻译成可读懂的字节码指令文件，里面包含了各种各样的指令，而这才是纯正的java字节码。字节码解释器就是在这一步将字节码翻译成C++代码，最后在执行硬编码。

比如：

public static void main(java.lang.String[]);

Code:

0: ldc #2 // int 100000

2: anewarray #3 // class java/lang/ref/SoftReference

（3）C++代码：ConstantPool\* constants = istate->method()->constants();

（4）硬编码：Java字节码文件（16进制的字节）ca fe ba be 00 00 00 34

### 模板解释器

Java字节码 -> 硬编码

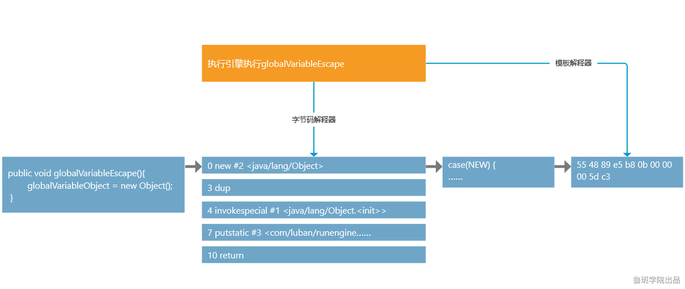
模板解释器具体过程

1. 申请一块内存：可读可写可执行（系统API申请）

JIT在Mac是无法运行的，Mac无法申请可执行的内存块

1. 将处理字节码的硬编码拿过来（硬编码怎么拿到？）
2. 将处理字节码的硬编码写入申请的内存
3. 申请一个函数指针，用这个函数指针执行这块内存
4. 调用的时候，直接通过这个函数指针调用就可以了

简单理解：前面已经触发了及时编译，不需要硬编码以前代码解释的过程，直接执行硬编码。

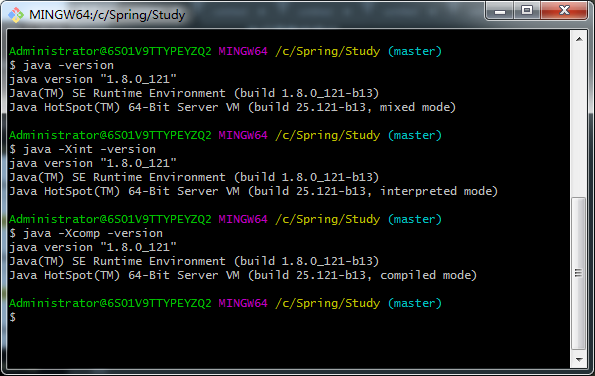


### 三种运行模式

-Xint 纯字节码解释器，性能9894ms

-Xcomp 纯模板解释器，性能2411ms，如果程序比较小，该性能最优

-Xmixed 字节码解释器 + 模板解释器，性能2638ms，如果程序比较大，该性能最优



## 即时编译器

即时编译器（JIT compiler，just-in-time compiler）是一个把Java的字节码（包括需要被解释的指令的程序）转换成可以直接发送给处理器的指令的程序。通常情况下，Java程序最初都是被编译为字节码，通过解释器进行解释执行，解释执行能够获得更好的启动时间。某些被频繁执行的方法或者代码块，会被JVM认定为“热点代码”。在运行时JVM会把这些热点代码编译成与本地平台相关的机器码，并且进行各种层次的优化，以提高执行效率。完成这个任务的编译器称为即时编译器（JIT编译器）。

简单的理解：即时编译器（异步的）生成的代码就是给模板解释器用的。模板解释器执行的硬编码就是即时编译器给编译的，他和字节码解释器没有关系。

C1：client模式下的即时编译器

1. 触发的条件相对C2比较宽松，需要收集的数据较少。
2. 编译的优化比较浅：基本运算在编译的时候运算掉了，比如final，final后面内容是固定的
3. C1编译器编译生成的代码执行效率较C2低

C2：server模式下的即时编译器

1. 触发的条件比较严格，一般来说，程序运行了一段时间以后才会触发
2. 优化比较深
3. C2编译器编译生成的代码执行效率较C1高

混合编译：C1+C2

程序运行初期触发C1编译器，程序运行一段时间后触发C2编译器

GraalVM：JDK14才有的

即时编译器触发条件：热点代码达到最小单位时触发，即时编译的最小单位不是一个函数，而是代码块（比如for、while）。如果已经触发过即时编译就不会再次触发。

（1）Client编译器模式下，N默认值1500

（2）Server编译器模式下，N默认值10000

默认值查看指令：java -client -XX:+PrintFlagsFinal -version | grep CompileThreshold

热度衰减：一段时间没有执行某个方法，就会以2倍速度往下掉。

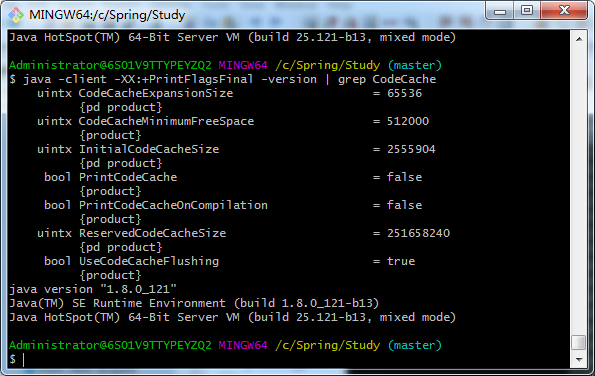
### 热点代码

编译好的硬编码，热点代码缓存区，它存放在方法区中。会自动清理，LRU

热点代码缓存区查看命令：java -client -XX:+PrintFlagsFinal -version | grep CodeCache

（1）erver编译器模式下代码缓存大小起始于2496K

（2）Client编译器模式下代码缓存大小起始于160K



调优参数：1、InitialCodeCacheSize 2、ReservedCodeCacheSize 一般2个会调成一样大

### 即时编译器是如何运行的

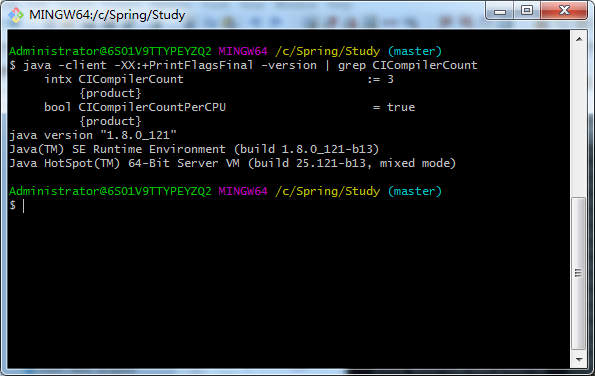
通过VM\_THREAD（VM的系统线程）触发，system.gc也是通过VM\_THREAD触发运行的。

1. 将即时编译任务写入到队列QUEUE中
2. VM\_THREAD从队列中读取任务并运行

### 执行即时编译的线程有多少，以及如何调优

java -client -XX:+PrintFlagsFinal -version | grep CICompilerCount

3个线程，可以通过-XX:CICompilerCount=N来完成线程调优



## 热机切冷机故障

场景：为程序增加新的服务器节点，将热机压力平摊到冷机，主要是更改负载均衡器，但是流量切到冷机，冷机立马挂掉。

原因：

1、热机有热点代码缓存，抗的并发更大，但冷机不是

2、热机切过去冷机时，程序一边在运行，一边在触发即时编译，CPU负载压力大，导致热机切冷机出现故障。

解决方案：流量一点点切换到冷机上。

## 如何理解Java是半编译半解释语言

1. javac编译，java运行
2. 字节码解释器解释执行，模板解释器编译执行

## 逃逸分析

逃逸：是一种现象。对象的作用域不是局部的（非局部变量），逃到方法外、线程外。比如：共享变量、私有变量、返回值、参数等。方法里的参数就不是。

分析：是一种技术手段。基于逃逸分析，JVM开发了三种优化技术。因为如果对象发生了逃逸，该对象可能在程序中被访问到的地方无法确定，导致对象被传进了不确定的代码中去运行。

总结：不逃逸才需要优化。

### 栈上分配

对象在虚拟机栈上分配，逃逸分析如果是开启的（默认开启），栈上分配就是存在的。如果关闭逃逸分析，创建的对象在堆区。

在启动java项目中添加启动参数，VM options中加入如下代码：

-XX:+DoEscapeAnalysis 开启逃逸分析

-XX:-DoEscapeAnalysis 关闭逃逸分析

### 标量替换

标量：不可再分，java中的基本数据类型就是标量。

聚合量：可再分，对象。

比如：

Public void Test(){

Position position = new Position(1,2,3)

System.out.println(position.x); // JVM在做逃逸分析时，会被替换成1，这就是标量替换

System.out.println(position.y); // JVM在做逃逸分析时，会被替换成2，这就是标量替换

System.out.println(position.z); // JVM在做逃逸分析时，会被替换成3，这就是标量替换

}

### 锁消除

JVM在做逃逸分析时，发现锁对象是局部变量，不会发生逃逸，JVM会消除锁

public void noEscape(){

synchronized (new Object()){ //仅创建线程可见,对象无逃逸

System.out.println(“hello”);

}

}

锁消除后

public void noEscape(){

System.out.println(“hello”);

}

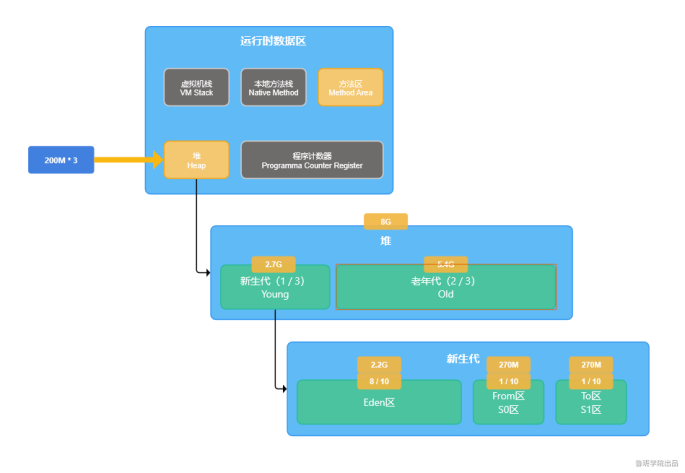
## JVM调优

1. 项目上线前做预估调优
2. 项目上线初期，基于日志做一些基础调优
3. 发生OOM、频繁full gc。做彻底调优

### 调优方向

1. jvm内存模型调优
2. 热点代码缓存区调优

### 样例



这里以亿级流量秒杀电商系统为例：

1、如果每个用户平均访问20个商品详情页，那访客数约等于500w（一亿 / 20）

2、如果按转化率10%来算，那日均订单约等于50w（500w \* 10%）

3、如果40%的订单是在秒杀前两分钟完成的，那么每秒产生1200笔订单（50w \* 40% / 120s）

4、订单支付又涉及到发起支付流程、物流、优惠券、推荐、积分等环节，导致产生大量对象，这里我们假设整个支付流程生成的对象约等于20K，那每秒在Eden区生成的对象约等于20M（1200笔 \* 20K）

5、在生产环境中，订单模块还涉及到百万商家查询订单、改价、包邮、发货等其他操作，又会产生大量对象，我们放大10倍，即每秒在Eden区生成的对象约等于200M（其实这里就是在大并发时刻可以考虑服务降级的地方，架构其实就是取舍）

### 环境

32G内存，1/4放到堆区8G（堆区最小是物理内存的1/64，最大是1/4），堆区又分为1/3新生代2700M，2/3老年代5400M，新生代分为8/10的Eden区2200M，1/10的From区270M，1/10的To区270M

每秒产生200M对象，完成一次操作大概是3秒（保守估计，一般都是5秒往上），所以一次操作大概产生600M对象。

新生代大概在14秒（2700 / 200）中发生young gc，young gc后会触发垃圾回收，有600M对象无法被回收，因为from区和to区分别只有270M，无法存放600M对象，会触发空间担保，会直接进入老年代。

老年代大概在126秒（14 \* 5400 / 600）触发一次full gc

### 如何调优

1. 加机器，使这些对象尽量在young gc阶段被回收，少触发full gc
2. 调方法区
3. 虚拟机栈
4. 堆区
5. 热点代码缓冲区

### 问题

（1）为什么有的系统频繁触发full gc

本质是有对象在young gc时未被清理干净，触发了空间担保/动态年龄判断/度过了15次gc等，进去了老年代

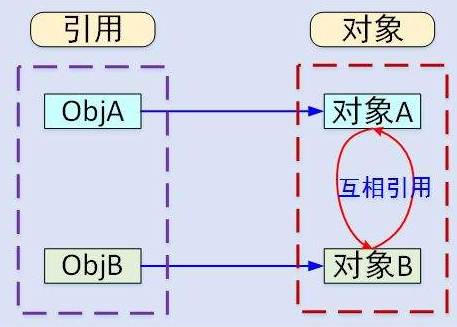
# 垃圾回收

即判断JVM中的所有对象，哪些对象是存活的，哪些对象可回收的算法

## 垃圾判断算法

### 引用计数算法

在对象中添加一个属性用于标记对象被引用的次数，每多一个其他对象引用，计数+1，当引用失效时，计数-1，如果计数=0，表示没有其他对象引用，就可以被回收。这个算法无法解决循环依赖的问题。

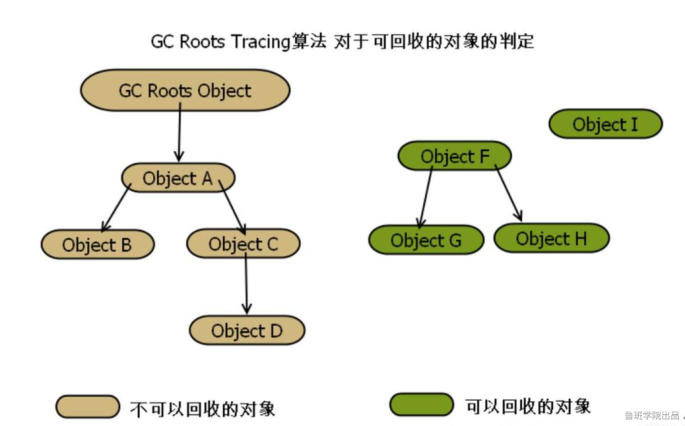


### 可达性分析算法

通过一系列被称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系链向下搜索，如果某个对象无法被搜索到，则说明该对象无引用执行，可回收。相反，则对象处于存活状态，不可回收。JVM中的实现是找到存活对象，未打标记的就是无用对象，GC时会回收。

哪些对象可以作为GC Root呢：

* 所有Java线程当前活跃的栈帧里指向GC堆里的对象的引用；换句话说，当前所有正在被调用的方法的引用类型的参数/局部变量/临时值。
* VM的一些静态数据结构里指向GC堆里的对象的引用，例如说HotSpot VM里的Universe里有很多这样的引用。
* JNI handles，包括global handles和local handles
* （看情况）所有当前被加载的Java类
* （看情况）Java类的引用类型静态变量
* （看情况）Java类的运行时常量池里的引用类型常量（String或Class类型）
* （看情况）String常量池（StringTable）里的引用



#### 三色标记

把遍历对象过程中遇到的对象，按照“是否访问过”这个条件标记成三种颜色：

（1）白色：尚未访问过。未被标记过的对象，GC会回收这种对象

（2）灰色：本对象已访问过，但是本对象 引用到 的其他对象 尚未全部访问完。全部访问后，会转换为黑色。对象节点被标记，但引用的对象未被标记完。

（3）黑色：本对象已访问过，而且本对象 引用到 的其他对象 也全部访问过了。对象节点以及对象节点引用其他对象已被标记完

标记顺序：黑、灰、白

#### 标记程序

1、初始标记：会STW。只标记GC Roots直接关联的对象。

2、并发标记：不会STW。GC线程与用户线程并发运行，会沿着GC Roots直接关联的对象链遍历整个对象图。可想而知需要的时间较长，但因为是与用户线程并发运行的，除了能感知到CPU飙升，不会出现卡顿现象。

3、重新标记：会STW。CMS垃圾收集器通过写屏障+增量更新记录了并发标记阶段新建立的引用关系，重新标记就是去遍历这个记录。

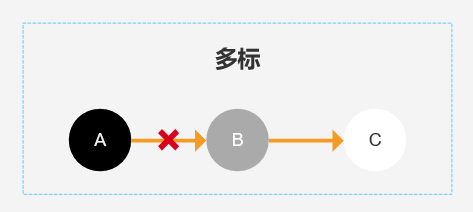
4、并发清除：GC线程与用户线程并发运行，清理未被标记到的对象

默认启动的回收线程数 = (处理器核心数 + 3) / 4

并发标记带来的问题

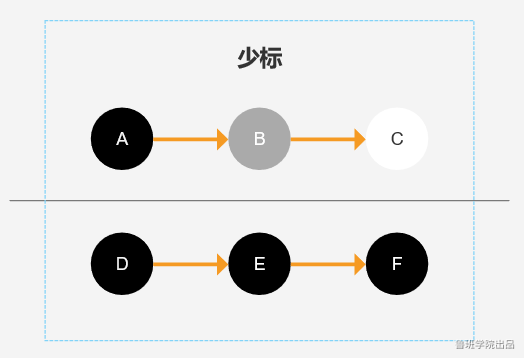
1、多标：GC线程已经标记了B，此时用户代码中A断开了对B的引用（因为GC线程和用户线程是并发执行的，会存在对象的引用被替换成其他对象的情况发送），但此时B已经被标记成了灰色，本轮GC不会被回收，这就是所谓的多标，多标的对象即成为浮动垃圾，躲过了本次GC。

影响：多标对程序逻辑是没有影响的，唯一的影响是该回收的对象躲过了一次GC，造成了些许的内存浪费。



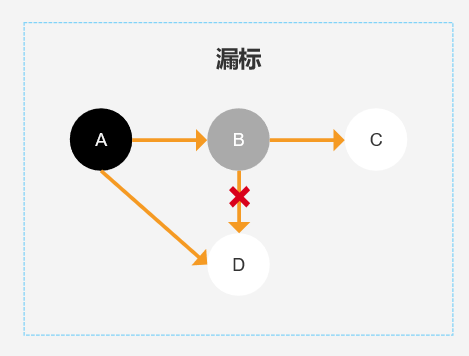
2、少标：标记程序在运行的过程中，用户线程依然会创建对象，此时新创建的对象会被视为黑色，本轮GC不清除，下次GC有可能会被释放。

这里面有的新创建对象被用完，就变成垃圾了，本可以被销毁，但未被销毁，这部分对象即少标环境中的浮动垃圾。



3、漏标：标记程序运行过程中引用链发生改变

漏标是如何产生的呢？GC把B标记完，准备标记B引用的对象，这时用户线程执行代码，代码中断开了B对D的引用，改为A对D的引用。但是A已经被标记成黑色，不会再次扫描A，而D还是白色，执行垃圾回收逻辑的时候D会被回收，程序就会出错了。



漏标问题是如何产生的：

条件一：灰色对象 断开了 白色对象的引用；即灰色对象 原来成员变量的引用 发生了变化。

条件二：黑色对象 重新引用了 该白色对象；即黑色对象 成员变量增加了 新的引用。

如何解决漏标问题？

1. 读屏障：读内存前，加入读前屏障，读内存后加入读后屏障，屏障前后功能可能是记录信息
2. 写屏障：写内存前，加入写前屏障，写内存后加入写后屏障，屏障前后功能可能是记录信息
3. 增量更新incremental update：标记程序运行过程中引用链发生改变（对象A对D建立引用关系），通过写屏障将变动记录下来，并将白色对象加入到待扫描的集合中等待扫描，用于重新标记阶段重新扫描处理。
4. 原始快照SATB：将原先灰色对象对白色对象的引用关系记录下来，放到待扫描集合中等待扫描，用于重新标记阶段重新扫描处理。

（1）、读屏障 + 重新标记

在建立A对D的引用时将D作为白色或灰色对象记录下来，并发标记结束后STW，然后重新标记由D类似的对象组成的集合。

重新标记环节一定要STW，不然标记就没完没了了。

（2）、写屏障 + 增量更新（IU）

这种方式解决的是条件二，即通过写屏障记录下更新，具体做法如下：

对象A对D的引用关系建立时，将D加入带扫描的集合中等待扫描

（3）、写屏障 + 原始快照（SATB）

这种方式解决的是条件一，带来的结果是依然能够标记到D，具体做法如下：

对象B的引用关系变动的时候，即给B对象中的某个属性赋值时，将之前的引用关系记录下来。

标记的时候，扫描旧的对象图，这个旧的对象图即原始快照。

## 内存池

1. OS堆，操作系统堆
2. 操作系统也有内存模型（堆、栈、静态区域、代码区域）
3. Memory Pool 内存池，管理内存块
4. 向操作系统要内存malloc、calloc
5. 释放内存，没有垃圾回收器需要手动释放
6. 其他，打印chunk信息
7. Memory Chunk 内存块
8. 直接持有内存
9. 分配的内存按8字节对齐
10. 将分配到的内存分块处理（Memory Cell内存细胞）
11. Memory Cell 内存细胞
12. 真实在使用的内存
13. 每个Cell占8字节

## 垃圾回收算法

### 标记-清除算法

面向整个堆，会产生内存碎片。

带来的问题：如果分配大对象时，此时需要连续的空间，但此时的内存是碎片化的，会分配不到内存

### 标记-整理算法

内存碎片合并算法，新生代和老年代基于这个算法来实现

带来的问题：消耗CPU

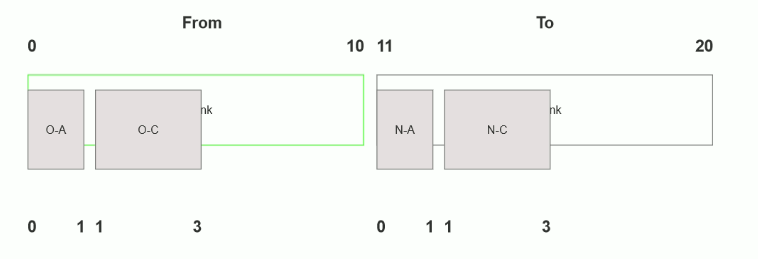
1. 新生代的Eden区，在第一次GC后就会释放内存，生命周期很短，碎片很多，合并碎片的时候需要STW（暂停所有用户线程）
2. 合并碎片同时面临两种空间
3. 空间对象已被释放
4. 空间对象未被释放，需要进行对象搬家（合并内存、数据移动、指针移动）

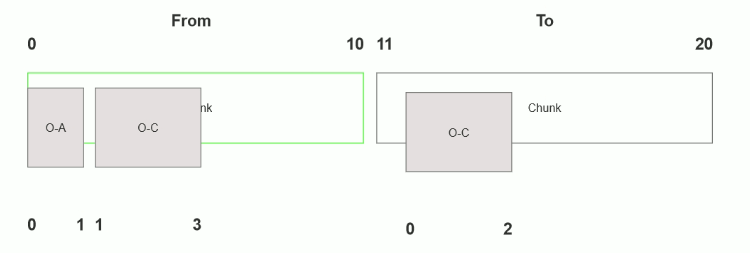
### 分代+复制算法

这个算法存在的目的：解决““标记-整理算法”合并碎片消耗性能过高，gc停止用户线程过长的问题

1. 将内存五五分，一半用From区，一半不用To区
2. 发生gc，From区标记对象
3. From和To区角色切换，From区为To区，To区为From区
4. To区内存处理
5. 清理标记的对象
6. 存活的对象移动到From区，从头部开始分配内存，此时指针也重新计算
7. 数据整理，清空To区内存memset
8. 指针整理

情况一：没有对象被清理



情况二：有对象被清理

这里的11应该是10才对

内存地址计算公式（其实也就是对象的起始位置）：

From区起始地址 + Cell的起始地址 \* 对齐字节数

O-A 0 + 0 \*8

O-C 0 + 1 \*8

N-A 10 + 0 \*8

N-C 10 + 1 \*8

### 问题

数据发生了移动，为什么依然能够访问到

## 内存分配算法

### 指针碰撞算法

内存的分配是并发的，他跟自旋锁类似，会一直CAS自旋，直到分配完内存。中间CAS失败，就代表该段内存已被使用

### 空闲列表

空闲状态的内存

## 串行、并行、并发

串行：用户线程STW，一个GC线程运行

并行：用户线程STW，多个GC线程运行

并发：不需要STW，用户线程、GC线程并发运行

STW：Stop The World暂停所有用户线程

# 垃圾回收器

## 1、Serial收集器

串行垃圾收集器，即GC线程与用户线程先后运行，即GC时需要STW（暂停所有用户线程），直至GC结束才恢复用户线程的运行

专注于收集年轻代，底层是复制算法

相关参数：-XX:+UseSerialGC

## 2、ParNew收集器

Serial收集器的多线程版本。唯一能与CMS收集器搭配使用的新生代收集器。

相关参数：

-XX:+UseConcMarkSweepGC：指定使用CMS后，会默认使用ParNew作为新生代收集器

-XX:+UseParNewGC：强制指定使用ParNew

-XX:ParallelGCThreads：指定垃圾收集的线程数量，ParNew默认开启的收集线程与CPU的数量相同

## 3、Parallel收集器

关注吞吐量的收集器

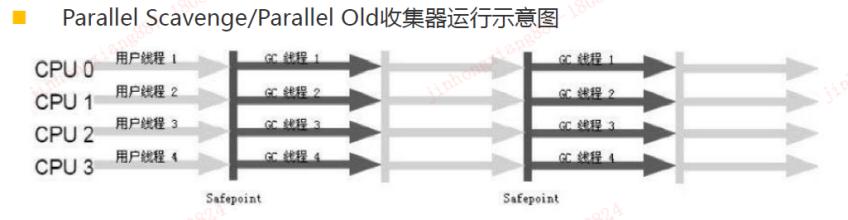
吞吐量 = 运行用户代码时间 / (运行用户代码时间+垃圾收集时间)

相关参数：

-XX:MaxGCPauseMillis：是一个大于0的毫秒数，收集器将尽力保证内存回收花费的时间不超过设定值。不过大家不要异想天开地认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

-XX:GCTimeRatio：一个大于0小于100的整数，也就是垃圾收集时间占总时间的比率。如果把此参数设置为19，那允许的最大GC时间就占总时间的5%（即1 /（1+19）），默认值为99，就是允许最大1%（即1 /（1+99））的垃圾收集时间。

-XX:+UseAdaptiveSizePolicy：一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）



## 4、Serial Old收集器

Serial收集器的老年代版本。基于标记-整理算法实现，

有两个用途：

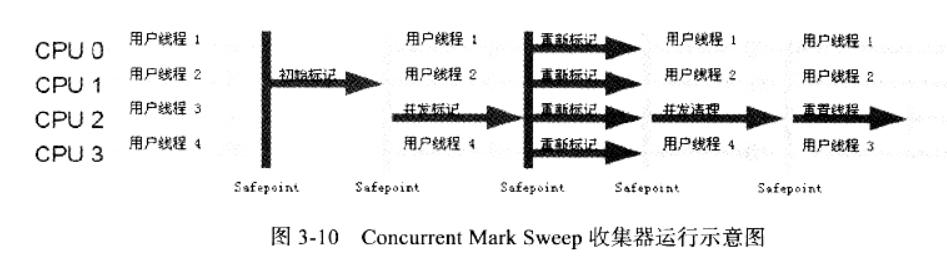
1、与Serial收集器、Parallel收集器搭配使用

2、作为CMS收集器的后备方案

## 5、Parallel Old收集器

Parallel收集器的老年代版本。基于标记-整理算法实现。

## 6、CMS收集器



聚焦低延迟。基于标记-清除算法实现。

由于CMS收集器是并发收集器，即在运行阶段用户线程依然在运行，会产生对象，所以CMS收集器不能等到老年代满了才触发，而是要提前触发，这个阈值是92%。这个阈值可以通过参数-XX:CMSInitiatingOccupancyFraction设置

相关参数：

-XX：+UseConcMarkSweepGC：手动开启CMS收集器

-XX:+CMSIncrementalMode：设置为增量模式

-XX:CMSFullGCsBeforeCompaction：设定进行多少次CMS垃圾回收后，进行一次内存压缩

-XX:+CMSClassUnloadingEnabled：允许对类元数据进行回收

-XX:UseCMSInitiatingOccupancyOnly：表示只在到达阀值的时候，才进行CMS回收

-XX:CMSInitiatingOccupancyFraction：设置CMS收集器在老年代空间被使用多少后触发

-XX:+UseCMSCompactAtFullCollection：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片的整理

CMS收集器工作分四个步骤：

1、初始标记

会STW。只标记GC Roots直接关联的对象。

2、并发标记

不会STW。GC线程与用户线程并发运行。会沿着GC Roots直接关联的对象链遍历整个对象图。可想而知需要的时间较长，但因为是与用户线程并发运行的，除了能感知到CPU飙升，不会出现卡顿现象。

3、重新标记

会STW。CMS垃圾收集器通过写屏障+增量更新记录了并发标记阶段新建立的引用关系，重新标记就是去遍历这个记录。

4、并发清除

GC线程与用户线程并发运行，清理未被标记到的对象

默认启动的回收线程数 = (处理器核心数 + 3) / 4

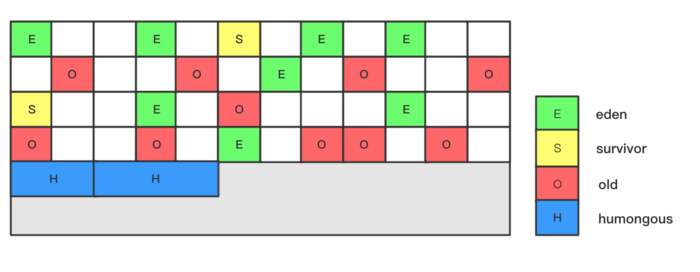
显然CMS收集器依然不是完美的，不然后面就不会出现G1、ZGC等。那有哪些缺点呢？

1、运行期间会与用户线程抢夺CPU资源。当然，这是所有并发收集器的缺点

2、无法处理浮动垃圾（标记结束后创建的对象）

3、内存碎片

## 7、G1收集器Garbage First



（1）G1是所有垃圾回收算法的综合体

（2）G1收集器与之前的所有收集器都不一样，它将堆分成了一个一个Region，这些Region用的时候才被赋予角色：Eden、from、to、humongous。一个region只能是一个角色，不存在一个region既是Eden又是from，一个region默认是2M，有2048个region。

（3）每个region的大小可通过参数-XX:G1HeapRegionSize设置，取值范围是2-32M（2的倍数），最大堆空间2 \* 2048 – 32 \* 2048 = 4G – 64G。

（4）一个对象的大小超过region的一半则被认定为大对象，会用N个连续的region来存储humongous。

### G1名字的由来

回收某个region的价值大小 = 回收获得的空间大小 + 回收所需时间

G1收集器会维护一个优先级列表，也就是G1会算出回收一个region获得的空间大小已经回收时间，每个region按价值大小排序存放在这个优先级列表中。收集时优先收集价值更大的region，这就是G1名字的由来。

### 四个步骤

1、初始标记

会STW。

做了两件事：

1、修改TAMS的值，TAMS以上的值为新创建的对象，默认标记为存活对象，即多标

2、标记GC Roots能直接关联到的对象

2、并发标记

耗时较长。GC线程与用户线程并发运行。

从GC roots能直接关联到的对象开始遍历整个对象图

3、最终标记

遍历写屏障+SATB记录下的旧的引用对象图

4、筛选回收

更新region的统计数据，对各个region的回收价值进行计算并排序，然后根据用户设置的期望暂停时间的期望值生成回收集。

然后开始执行清除操作。将旧的region中的存活对象移动到新的Region中，清理这个旧的region。这个阶段需要STW。

相关参数：

-XX:G1HeapRegionSize：设置region的大小

-XX:MaxGCPauseMillis：设置GC回收时允许的最大停顿时间（默认200ms）

-XX:+UseG1GC：开启g1

-XX:ConcGCThreads：设置并发标记、并发整理的gc线程数

-XX:ParallelGCThreads：STW期间并行执行的gc线程数

缺点：

1、需要10%-20%的内存来存储G1收集器运行需要的数据，如不cset、rset、卡表等

2、运行期间会与用户线程抢夺CPU资源。当然，这是所有并发收集器的缺点

## 查看默认收集器

java -XX:+PrintFlagsFinal -version | grep GC

## GC日志

相关参数：

-XX:+PrintGC 输出GC日志

-XX:+PrintGCDetails 输出GC的详细日志

-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:../logs/gc.log 日志文件的输出路径

日志内容：

1、gc类型：GC、Full GC

2、gc原因：Metadata GC Threshold、Last ditch collection……

3、gc前内存数据

4、gc后内存数据

5、花费的时间：用户态、内核态、实际用时