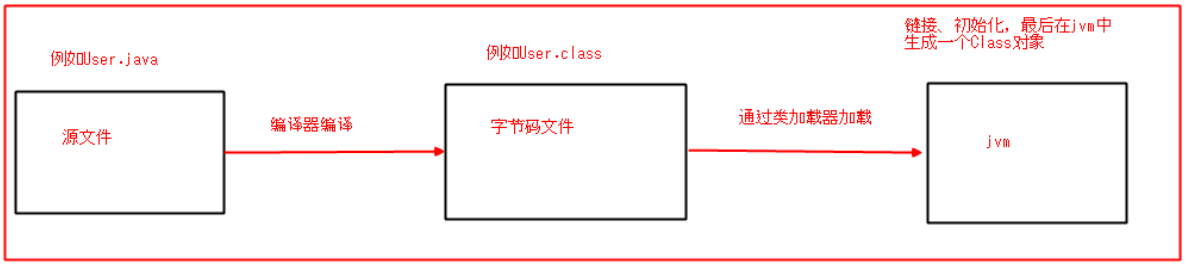
可以参考：

原文链接：https://blog.csdn.net/qq_41701956/article/details/81664921

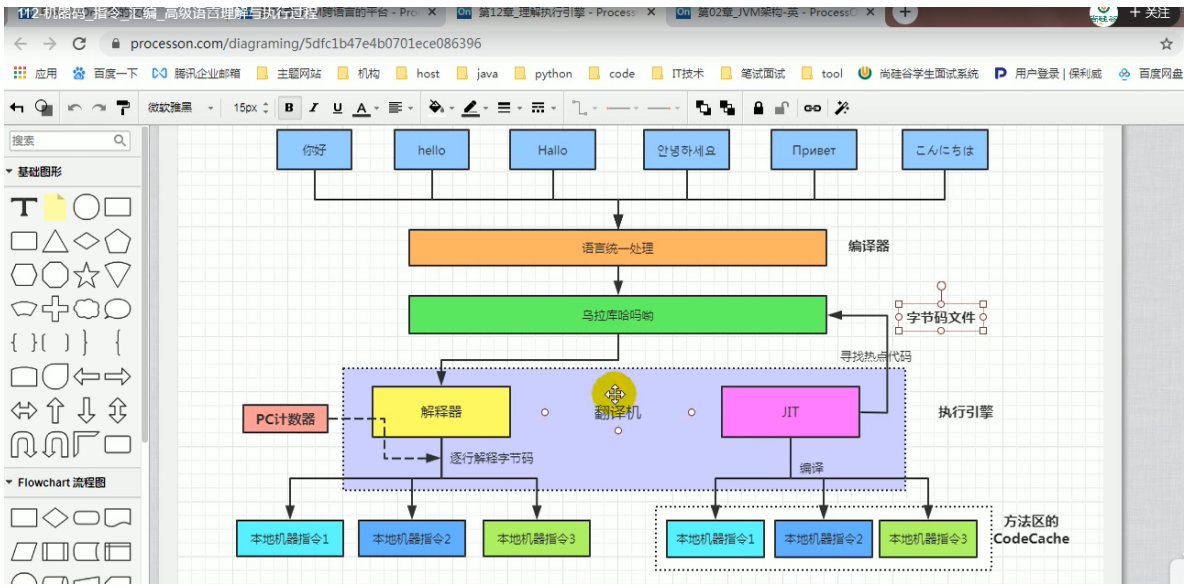
一、类加载过程以及Class文件

大家知道一个类加载到jvm大概是经过了几个步骤的吧！**编译成字节码文件，加载，链接（验证，准备，解析），初始化.....，**我就简单的用下面这个图一起看看；



1 编译器编译

这个没什么好说的，由于java是静态语言，在执行java程序之前会先把我们写的java文件给转化成特殊的二进制码的形式，编译器就是做这个转化的工作的工具，而且在我们写代码的时候，还没运行程序之前，就会报错，在某处代码下面会有红线标识，做这个工作的就是编译器。



2 类加载器的分类和加载顺序

2.1什么是类加载器

什么是类加载器呢？我有一个很生动很形象的例子：假如字节码文件是一个人，而jvm就是地府，你说人死了会怎么进入地府呢？自己肯定找不到地府的位置，于是要让黑白无常请你过去了，类加载器在这里就是黑白无常！

2.2类加载器分类

类加载器分为四种:包括一种自定义的

- **启动类加载器**（Bootstrap ClassLoader）：使用C++实现（仅限于HotSpot），是虚拟机自身的一部分。负责将存放在**\lib目录中的类库**加载到虚拟机中。其无法被Java程序直接引用。
- **扩展类加载器**（Extention ClassLoader）由ExtClassLoader实现，负责加载**\lib\ext目录中的所有类库**，开发者可以直接使用。
- **应用程序类加载器**（Application ClassLoader）：由APPClassLoader实现。负责加载**用户类路径（ClassPath）上所指定的类库。**

启动类加载器（Bootstrap ClassLoader）：最顶级的类加载器，还是用C++写的；在我们编写java程序的时候，编译器会自动的帮我们导入一下常用的jar包，用的就是这个类加载器，比如我们最熟悉的lang包下的Object，String，Integer等都是我们可以直接用的，而不需要我们手动导入；具体的会导入哪些jar包呢，这就需要我们配置环境变量JAVA_HOME，编译器会去环境变量中找%JAVA_HOME%\jre\lib，这下面所有jar包然后进行加载到内存中，注意不是加载在JVM中；而且出于安全考虑，启动类加载器只加载包名为java、javax、sun等开头的类

扩展类加载器（Extension ClassLoader）：父类加载器是启动类加载器，java语言实现，负责加载%JAVA_HOME%\jre\lib\ext 路径下的jar包,这个不会自动加载，只有在需要加载的时候才去加载。

应用类加载器（Application ClassLoader）：父类加载器是扩展类加载器，java语言实现，也可以叫做系统类加载器（System ClassLoader），这个类加载器主要是加载我们在写项目时编写的放在类路径下的类，比如maven项目中src/main/java/所有类

自定义类加载器：需要我们自己实现，当特殊情况下我们需要自定义类加载器，只需要实现ClassLoader接口，然后重写findClass（）方法，我们就能够自己实现一个类加载器，而且自己实现类加载器之后可以去加载任何地方的类。假如我新建一个类放在F盘的随便一个角落里也可以指定类路径去加载，有兴趣的小伙伴可以去试试。

不考虑自定义类加载器，可以看到，启动、扩展、应用这三个加载器就像是爷爷，爸爸，儿子一样的关系，所以要加载一个类的话，选用哪个类加载器呢？肯定是有什么好吃的先让儿子吃呀，然而儿子又很有孝心，会把到手的好吃的给爸爸吃。爸爸又会给爷爷吃，爷爷会尝试着吃，假如一看这东西糖分太高于是就又给爸爸吃，爸爸也尝试着吃，发现这东西不好吃，于是最后还是给儿子吃....这就是类加载器的双亲委托机制，随便找了一幅图看看：

2.3 类加载器的加载顺序： 双亲委派模型

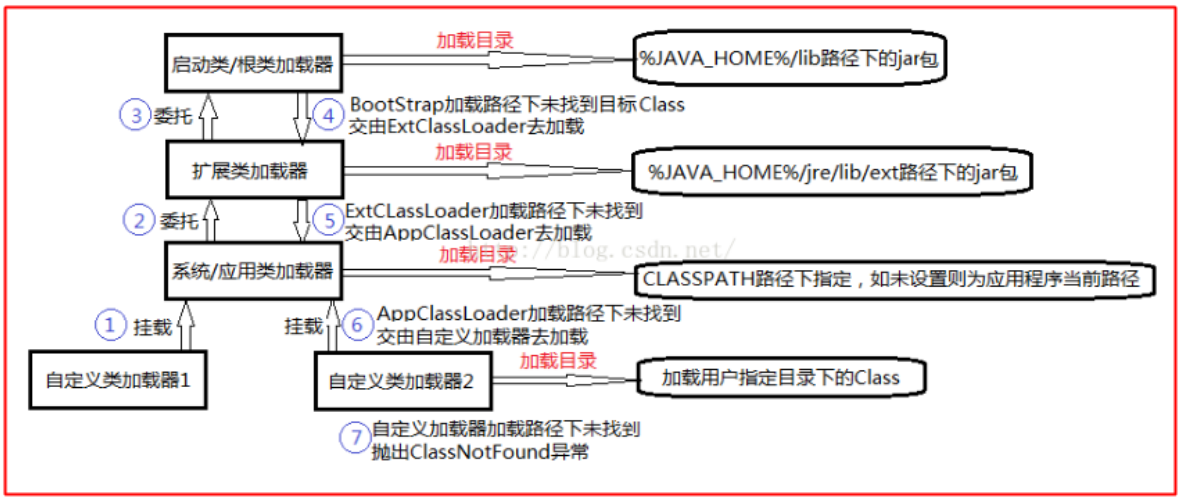
双亲委派模型（Parents Delegation Model）要求除了顶层的启动类加载器外，其余加载器都应当**有自己的父类加载器**。类加载器之间的父子关系，通过组合关系复用。

工作过程：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求**委派给父类加载器**完成。每个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有到父加载器反馈自己无法完成这个加载请求（它的搜索范围没有找到所需的类）时，子加载器才会尝试自己去加载。

为什么要使用双亲委派模型，组织类加载器之间的关系？

Java类随着它的类加载器一起具备了一种带优先级的层次关系。比如java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此Object类在程序的各个类加载器环境中，都是同一个类。

如果没有使用双亲委派模型，让各个类加载器自己去加载，那么Java类型体系中最基础的行为也得不到保障，应用程序会变得一片混乱。



3. Class文件的组成

Class文件是一组以**8位字节**为基础单位的**二进制流**，各个数据项目间没有任何分隔符。当遇到8位字节以上空间的数据项时，则会按照**高位在前**的方式分隔成若干个8位字节进行存储。

3.1 魔数与Class文件的版本

每个Class文件的头4个字节称为**魔数**（Magic Number），它的唯一作用是用于确定这个文件**是否为一个能被虚拟机接受的Class文件**。OxCAFEBABE。

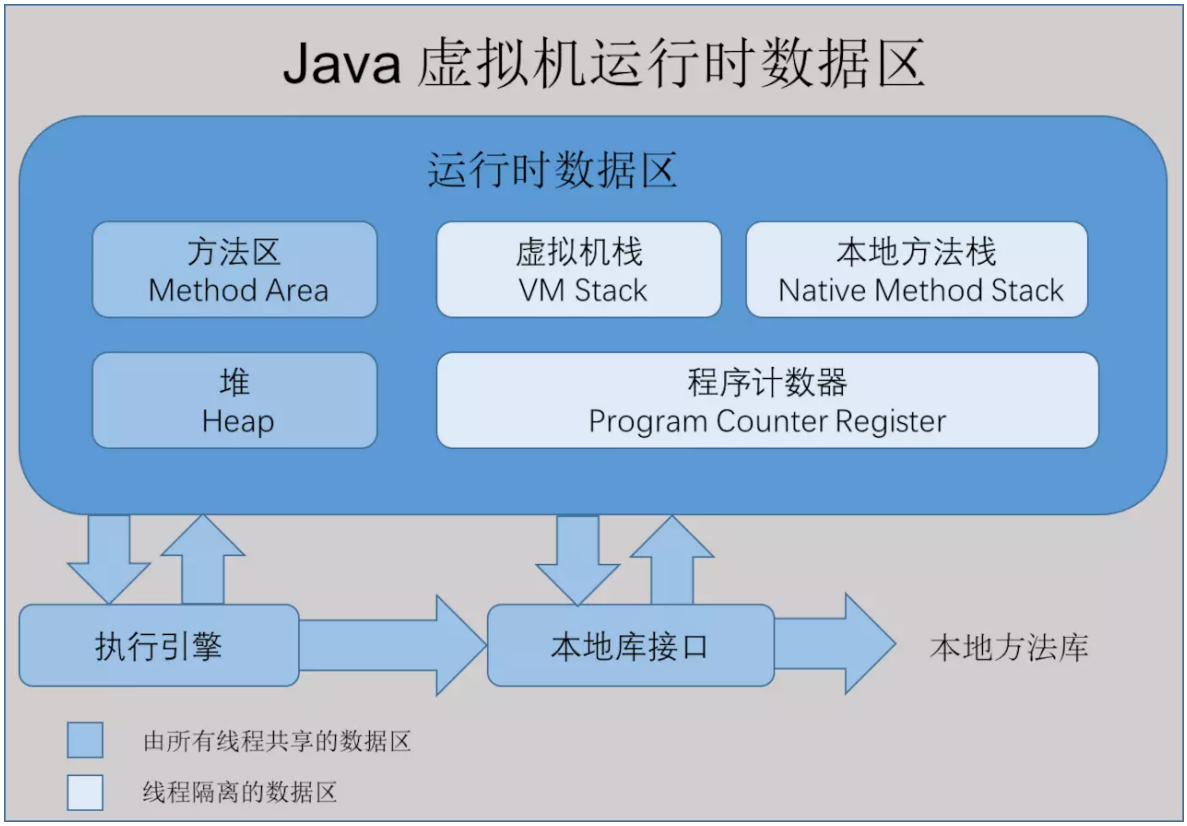
接下来是Class文件的**版本号**：第5,6字节是次版本号（Minor Version），第7,8字节是主版本号（Major Version）。

使用JDK 1.7编译输出Class文件，格式代码为：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	33	00	16	07	00	02	01	00	09
00000010	54	65	73	74	43	6C	61	73	73	07	00	04	01	00	10	6A

前四个字节为魔数，次版本号是0x0000，主版本号是0x0033，说明本文件是**可以被1.7及以上版本的虚拟机执行的文件。**

二、Java虚拟机的内存管理



1.程序计数器

内存空间小，线程私有。

程序计数器是一个记录着当前线程所执行的字节码的***行号指示器。***

如果线程正在执行一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器的值则为 (Undefined)。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

2 Java 虚拟机栈

线程私有，生命周期和线程一致。

描述的是 Java 方法执行的内存模型：每个**方法**在执行时都会创建一个栈帧(Stack Frame)用于存储局部变量表、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结束，就对应着一个栈帧从虚拟机栈中入栈到出栈的过程。***

局部变量表：存放了编译期可知的各种基本类型(boolean、byte、char、short、int、float、long、double)、对象引用(reference 类型)和 returnAddress 类型(指向了一条字节码指令的地址)

```
String s1 = "abc";
```

StackOverflowError：线程请求的栈深度大于虚拟机所允许的深度。 OutOfMemoryError：如果虚拟机栈可以动态扩展，而扩展时无法申请到足够的内存

3 本地方法栈

区别于 Java 虚拟机栈的是，Java 虚拟机栈为虚拟机执行 Java 方法(也就是字节码)服务，

而本地方法栈则为虚拟机使用到的 Native 方法服务。

（Native:简单地讲，一个Native Method就是一个java调用非java代码的接口。）

也会有 StackOverflowError 和 OutOfMemoryError 异常。

4 Java 堆 线程共享

对于绝大多数应用来说，这块区域是 JVM 所管理的内存中最大的一块。**主要是存放对象实例和数组。也是垃圾收集器管理的主要区域**，分为新生代（由Eden 与Survivor Space 组成）和老生代，可能会抛出 OutOfMemoryError异常。

*/

```
/ s3创建在堆内存中*

String s3 = new String("abc");
```

OutOfMemoryError：如果堆中没有内存完成实例分配，并且堆也无法再扩展时，抛出该异常。

5 方法区 共享内存

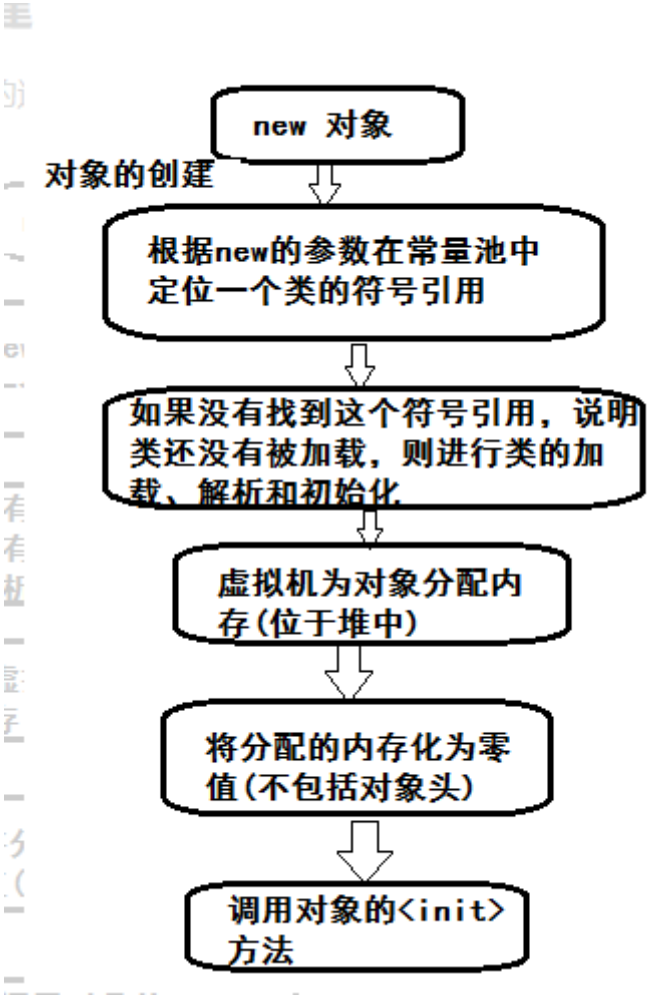
存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

6 运行时常量池

属于方法区一部分，用于**存放编译期生成的各种字面量和符号引用。**编译器和运行期(String 的 intern())都可以将常量放入池中。内存有限，无法申请时抛出 OutOfMemoryError。

```
程序计数器：记录当前线程执行的字节码的行号指示器。

java虚拟机栈：基本类型，以及栈帧（java方法执行到结束，对应一个栈帧）。

本地方法栈：为native方法服务。

java堆：实例化对象和数组。

方法区：静态变量，常量。

运行时常量：字面量和符号引用。
```

三、java对象的创建过程



类的符号引用：

在java中，一个java类将会编译成一个class文件。在**编译时，java类并不知道引用类的实际内存地址，因此只能使用符号引用来代替。**比如org.simple.People类引用org.simple.Tool类，在编译时People类并不知道Tool类的实际内存地址，因此只能使用符号org.simple.Tool(假设)来表示Tool类的地址。**而在类装载器装载People类时，此时可以通过虚拟机获取Tool类 的实际内存地址，**因此便可以既将符号org.simple.Tool替换为Tool类的实际内存地址，及直接引用地址。（

在解析阶段，Java虚拟机会把类的二级制数据中的符号引用替换为直接引用。）

1.0 直接引用：

```
public class StringAndStringBuilder{
  public static void main(String[] args){
     System.out.println ("s=" + "asdfa");
  }
}
```

2.0 符号引用：

```
public class StringAndStringBuilder{
  public static void main(String[] args){
    String s="asdfa";
     System.out.println ("s=" + s);
  }
}
```

四、垃圾回收器与内存分配策略

1 概述

程序计数器、虚拟机栈、本地方法栈 3 个区域随线程生灭(因为是线程私有)，栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期才知道那些对象会创建，这部分内存的分配和回收都是动态的，垃圾回收期所关注的就是这部分内存。

2 对象已死吗?

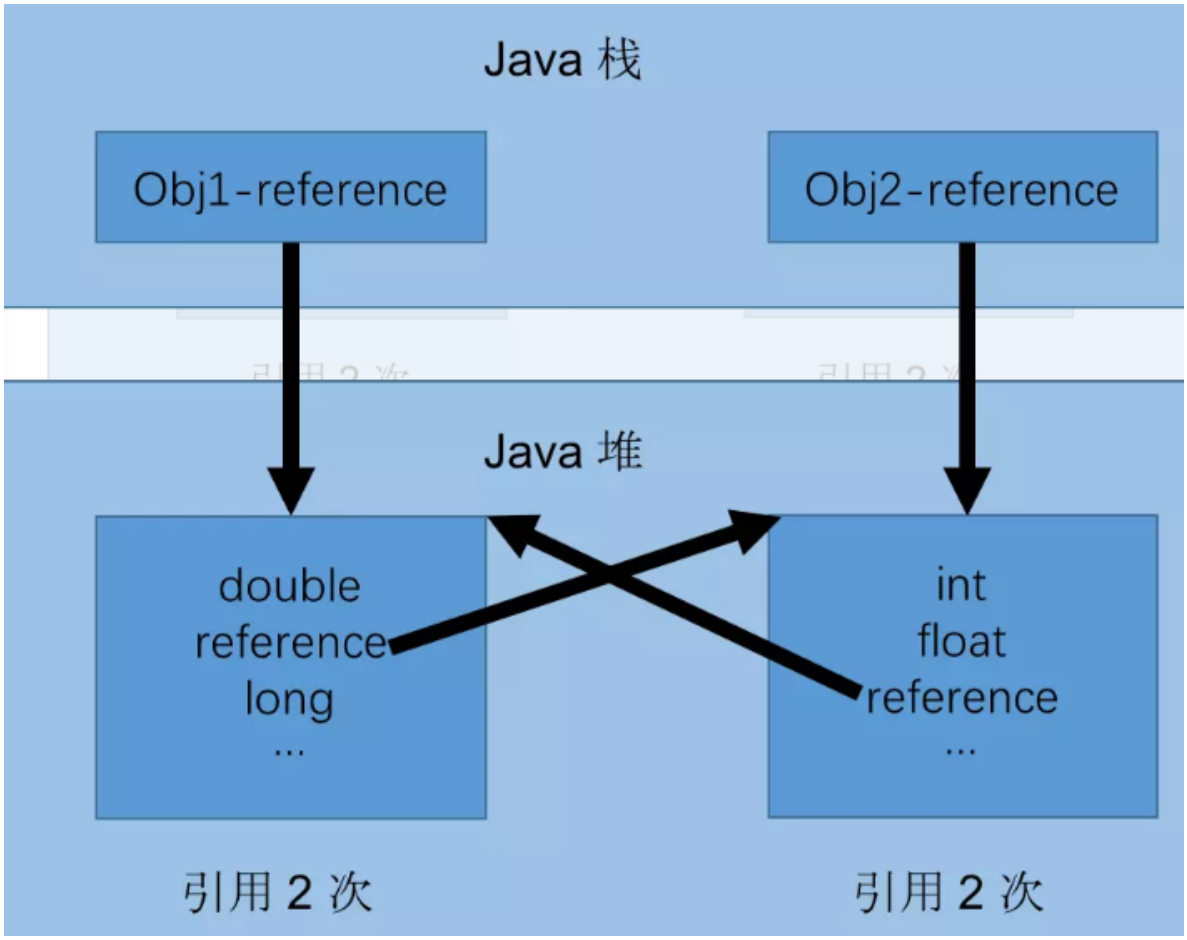
在进行内存回收之前要做的事情就是判断那些对象是'死'的，哪些是'活'的。

2.1 引用计数法

给对象添加一个引用计数器。

> 每当有一个地方引用它，计数器就+1,；当引用失效时，计数器就-1；任何时刻计数器都为0的对象就是不能再被使用的。

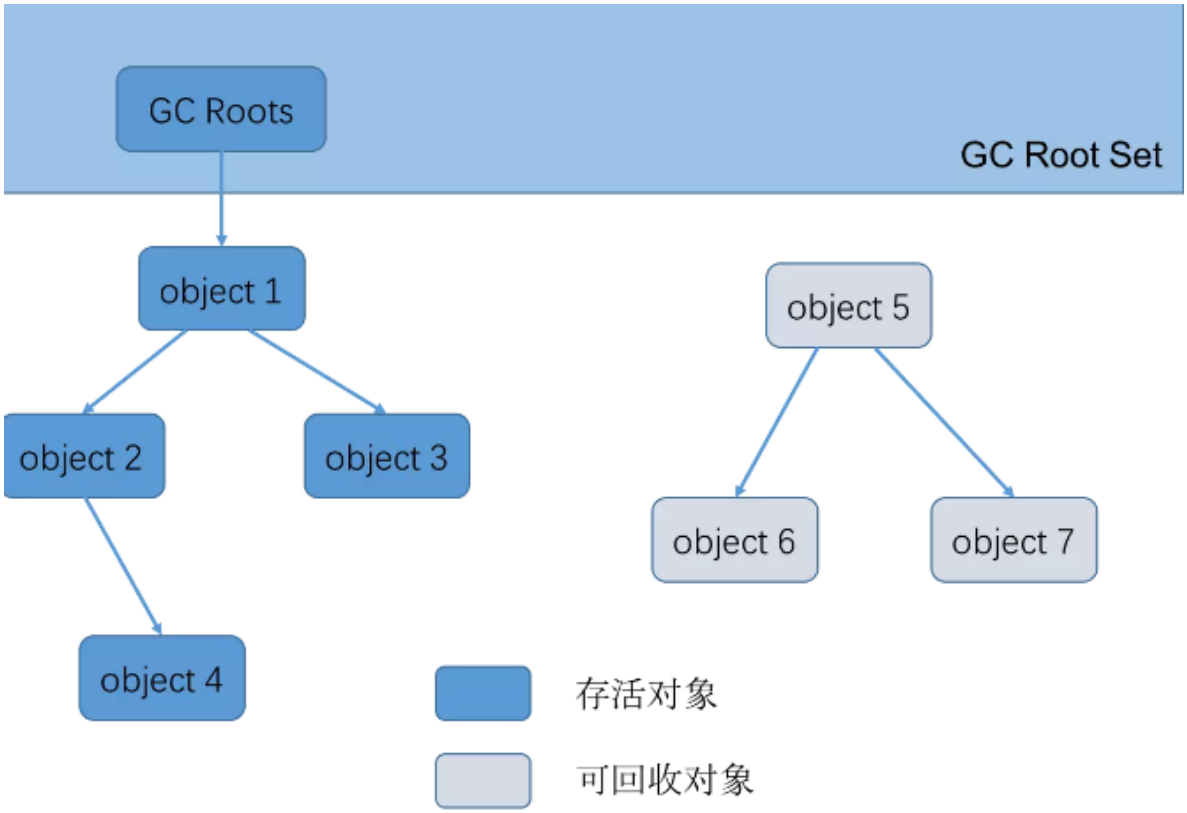
缺点：很难解决对象之间的循环引用问题。



从图中可以看出，如果不下小心直接把 Obj1-reference 和 Obj2-reference 置 null。则在 Java 堆当中的两块内存依然保持着互相引用无法回收。

2.2 可达性分析法

通过一系列的 **'GC Roots' 的对象作为起始点**，从这些节点出发所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连的时候说明对象不可用。



可作为 GC Roots 的对象：(?没有堆的吗)

虚拟机栈(栈帧中的本地变量表)中引用的对象 方法区中类静态属性引用的对象 方法区中常量引用的对象 本地方法栈中 JNI(即一般说的 Native 方法) 引用的对象

2.3 再谈引用

前面的两种方式判断存活时都与'引用'有关。但是 JDK 1.2 之后，引用概念进行了扩充，下面具体介绍。

强引用

类似于 Object obj = new Object(); 创建的，只要强引用在就不回收。

软引用

SoftReference 类实现软引用。在系统要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。

弱引用

WeakReference 类实现弱引用。对象只能生存到下一次垃圾收集之前。在垃圾收集器工作时，无论内存是否足够都会回收掉只被弱引用关联的对象。

虚引用

PhantomReference 类实现虚引用（ˈfæntəm）。无法通过虚引用获取一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

2.4 生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是"facebook"的，这时候它们暂时出于"缓刑"阶段，**一个对象的真正死亡至少要经历两次标记过程**：如果对象在进行中可达性分析后发现没有与 GC Roots 相连接的引用链，那他将会被第一次标记并且进行一次筛选，筛选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，或者 finalize() 方法已经被虚拟机调用过，虚拟机将这两种情况都视为"没有必要执行"。 如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象竟会放置在一个叫做 F-Queue 的队列中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的"执行"是指虚拟机会出发这个方法，并不承诺或等待他运行结束。finalize() 方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize() 中成功拯救自己 —— 只要重新与引用链上的任何一个对象简历关联即可。 finalize() 方法只会被系统自动调用一次。

2.5 回收方法区

在堆中，尤其是在新生代中，一次垃圾回收一般可以回收 70% ~ 95% 的空间，而永久代的垃圾收集效率远低于此。 永久代垃圾回收主要两部分内容：废弃的常量和无用的类。

判断废弃常量：一般是判断没有该常量的引用。 判断无用的类：要以下三个条件都满足

该类所有的实例都已经回收，也就是 Java 堆中不存在该类的任何实例 加载该类的 ClassLoader 已经被回收 该类对应的 java.lang.Class 对象没有任何地方呗引用，无法在任何地方通过反射访问该类的方法

3 垃圾回收算法

仅提供思路

3.1 标记 —— 清除算法

直接标记清除就可。

两个不足：

效率不高 空间会产生大量碎片

3.2 复制算法

把空间分成两块，每次只对其中一块进行 GC（Garbage Collection）。当这块内存使用完时，就将还存活的对象复制到另一块上面。

解决前一种方法的不足，但是会造成空间利用率低下。因为大多数新生代对象都不会熬过第一次 GC。所以没必要 1 : 1 划分空间。可以分一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 空间和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 上，最后清理 Eden 和 Survivor 空间。大小比例一般是 8 : 1 : 1，每次浪费 10% 的 Survivor 空间。但是这里有一个问题就是如果存活的大于 10% 怎么办？这里采用一种分配担保策略：多出来的对象直接进入老年代。

3.3 标记-整理算法

不同于针对新生代的复制算法，针对老年代的特点，创建该算法。主要是把存活对象移到内存的一端。

3.4 分代回收

内存分配有哪些原则?

1. 对象优先分配在 Eden
2. 大对象直接进入老年代
3. 长期存活的对象将进入老年代
4. 动态对象年龄判定
5. 空间分配担保

新生代采用复制算法

主要用来**存储新创建的对象**，内存较小，垃圾回收频繁。这个区又分为三个区域：**一个 Eden Space 和两个 Survivor Space。**

老年代采用标记-整理算法

主要用来**存储长时间被引用的对象。**它里面存放的是经过**几次在 Young Generation Space 进行扫描判断过仍存活的对象**，内存较大，垃圾回收频率较小。

永生代

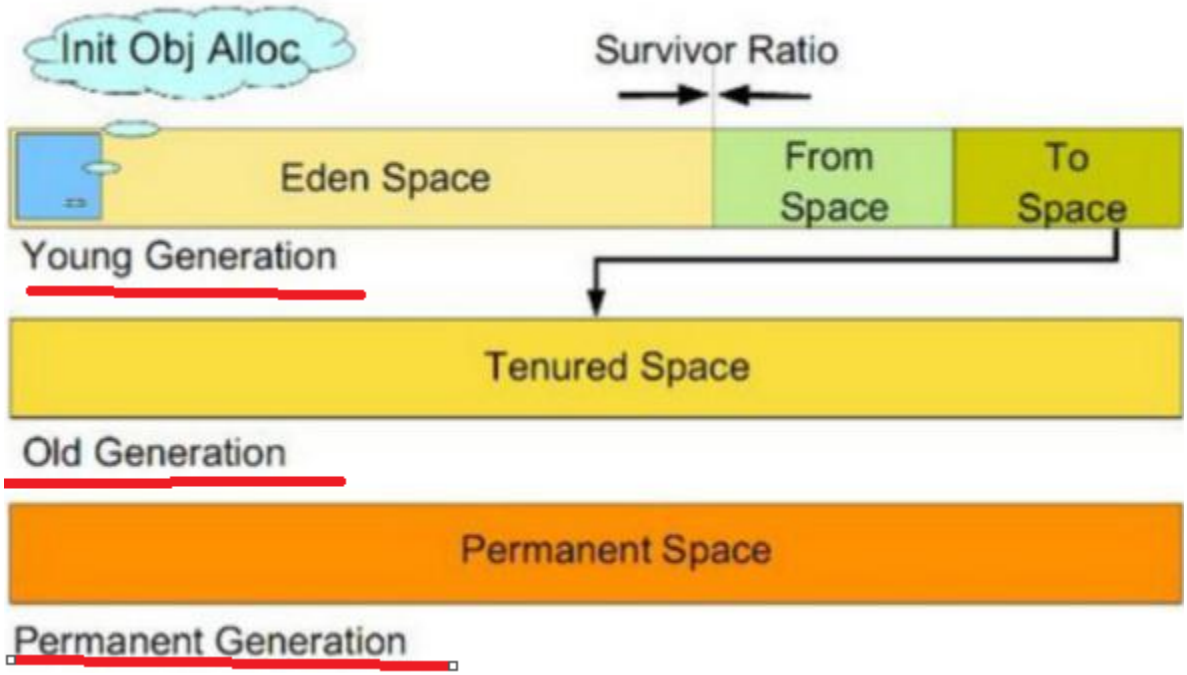
存储不变的类定义、字节码和常量等。

五、gc(Garbage Collection)的定义

GC，即就是Java垃圾回收机制。目前主流的JVM（HotSpot）采用的是**分代收集算**法。与C++不同的是，Java采用的是**类似于树形结构的可达性分析法**来判断对象是否还存在引用。即：从gcroot开始，把所有可以搜索得到的对象标记为存活对象。

1.heap区介绍

通常我们所说的gc主要是针对java heap这块区域的。下面来了解一下heap区。



从图中我们可以看出jvm heap区域是分代的，分为年轻代，老年代和持久代。在垃圾收集过程中，可能会将对象移动到不同区域： - 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。 - 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。 - 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。***当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC）***，这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

2.GC的基本原理:

对于程序员来说，用new关键字即在堆中分配了内存，我们称之为“可达”。

对于GC来说，只要所有被引用的对象为null时，我们称之为“不可达”，就将进行内存的回收。

当***一个对象被创建时，GC开始监控这个对象的大小、内存地址及使用情况。GC采用有向图的方式记录和管理堆(heap)中的所有对象***，通过这种方式可以明确哪些对象是可达的，哪些不是。当确定为不可达时，则对其进行回收。

3.Java GC机制

3.1 Minor GC(ˈmaɪnər 较小的次要的)

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。 当 JVM 无法**为一个新的对象分配空间时**会触发 Minor GC，比如当 Eden 区满了。Eden 和 Survivor 进行标记和复制操作执行 Minor GC 操作时，不会影响到永久代。**从永久代到年轻代的引用被当成 GC roots**，从年轻代到永久代的引用在标记阶段被直接忽略掉。

3.2.Major GC &Full GC(一般full gc也叫major gc)

Major GC 是清理永久代。Full GC 是清理整个堆空间:**包括年轻代和永久代**。 许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的。

4.full gc的触发条件

充分了解了jvm的内存结构之后，下面我们就来说说什么情况下会触发gc。触发full gc的情况主要有这几种：

（1）System.gc()方法的调用。此方法的调用是建议JVM进行Full GC,虽然只是建议而非一定，但很多情况下它会触发 Full GC,从而增加Full GC的频率，也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存.

（2）旧生代空间不足。旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出错误：java.lang.OutOfMemoryError: Java heap space 。为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

（3）Permanet Generation空间满了。**Permanet Generation中存放的为一些class的信息等**，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出错误信息：java.lang.OutOfMemoryError: PermGen space 。为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存。

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

5.gc回收的内容

知道了gc触发的条件之后，我们就能知道gc主要回收什么了？gc的主要作用是回收堆中的对象。通过可达性分析一个对象的引用是否存在，如果不存在，就可以被回收了。

6.gc的具体过程

那么，gc的是如何实现的，这个主要看是用的哪一种回收算法以及用的什么垃圾回收集了。回收算法主要有：

标记-清除复制算法。标记-整理(Mark-Compat)算法。分代收集(Generational Collection)算法。

六、HotSpot 垃圾收集器:7个

HotSpot 虚拟机提供了多种垃圾收集器，每种收集器都有各自的特点，虽然我们要对各个收集器进行比较，但并非为了挑选出一个最好的收集器。我们选择的只是对具体应用最合适的收集器。

1 新生代垃圾收集器

1.1 Serial 垃圾收集器（单线程）

只开启**一条** GC 线程进行垃圾回收，并且在垃圾收集过程中停止一切用户线程(Stop The World)。

一般客户端应用所需内存较小，不会创建太多对象，而且堆内存不大，因此垃圾收集器回收时间短，即使在这段时间停止一切用户线程，也不会感觉明显卡顿。因此 Serial 垃圾收集器**适合客户端**使用。

由于 Serial 收集器只使用一条 GC 线程，避免了线程切换的开销，从而简单高效。

1.2 ParNew 垃圾收集器（多线程）

ParNew 是 Serial 的多线程版本。由多条 GC 线程并行地进行垃圾清理。但清理过程依然需要 Stop The World。

ParNew 追求“**低停顿时间**”,与 Serial 唯一区别就是使用了多线程进行垃圾收集，在多 CPU 环境下性能比 Serial 会有一定程度的提升；但**线程切换需要额外的开销**，因此在单 CPU 环境中表现不如 Serial。

1.3 Parallel Scavenge 垃圾收集器（多线程） (/ˈskævɪndʒ/ 捡破烂)

Parallel Scavenge 和 ParNew 一样，都是多线程、新生代垃圾收集器。但是两者有巨大的不同点：

- Parallel Scavenge：**追求 CPU 吞吐量**，能够在较短时间内完成指定任务，因此**适合没有交互的后台计算。**
- ParNew：追求**降低用户停顿时间**，适合**交互式应用。**

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)

追求高吞吐量，可以通过减少 GC 执行实际工作的时间，然而，仅仅偶尔运行 GC 意味着每当 GC 运行时将有许多工作要做，因为在此期间积累在堆中的对象数量很高。单个 GC 需要花更多的时间来完成，从而导致更高的暂停时间。而考虑到低暂停时间，最好频繁运行 GC 以便更快速完成，反过来又导致吞吐量下降。

- 通过参数 -XX:GCTimeRadio 设置垃圾回收时间占总 CPU 时间的百分比。
- 通过参数 -XX:MaxGCPauseMillis 设置垃圾处理过程最久停顿时间。
- 通过命令 -XX:+UseAdaptiveSizePolicy 开启自适应策略。我们只要设置好堆的大小和 MaxGCPauseMillis 或 GCTimeRadio，收集器会自动调整新生代的大小、Eden 和 Survivor 的比例、对象进入老年代的年龄，以最大程度上接近我们设置的 MaxGCPauseMillis 或 GCTimeRadio。

2 老年代垃圾收集器

2.1 Serial Old 垃圾收集器（单线程）

Serial Old 收集器是 Serial 的老年代版本，都是单线程收集器，只启用一条 GC 线程，都适合客户端应用。它们唯一的区别就是：**Serial Old 工作在老年代，使用“标记-整理”算法；Serial 工作在新生代，使用“复制”算法。**

2.2 Parallel Old 垃圾收集器（多线程）

Parallel Old 收集器是 Parallel Scavenge 的老年代版本，追求 CPU 吞吐量。

2.3 CMS 垃圾收集器（目的：回收停顿时间最短）

CMS(Concurrent Mark Sweep，并发标记清除/concurrent并发的/)收集器是**以获取最短回收停顿时间为目标**的收集器（追求低停顿），它在垃圾收集时使得用户线程和 GC 线程并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿。

运作步骤:

- **初始标记**：Stop The World，仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记。
- **并发标记**：使用**多条**标记线程，与用户线程并发执行。此过程进行可达性分析，标记出所有废弃对象。速度很慢。
- **重新标记**：Stop The World，使用多条标记线程并发执行，将刚才并发标记过程中新出现的废弃对象标记出来。
- **并发清除**：只使用一条 GC 线程，与用户线程并发执行，清除刚才标记的对象。这个过程非常耗时。

并发标记与并发清除过程耗时最长，且可以与用户线程一起工作，因此，**总体上说**，CMS 收集器的内存回收过程是与用户线程**一起并发执行**的。

CMS 的缺点：

- 吞吐量低
- 无法处理浮动垃圾，导致频繁 Full GC
- 使用“标记-清除”算法产生碎片空间

对于产生碎片空间的问题，可以通过开启 -XX:+UseCMSCompactAtFullCollection，在每次 Full GC 完成后都会进行一次内存压缩整理，将零散在各处的对象整理到一块。设置参数 -XX:CMSFullGCsBeforeCompaction告诉 CMS，经过了 N 次 Full GC 之后再进行一次内存整理。

3 G1 通用垃圾收集器

G1 是一款面向**服务端应用**的垃圾收集器，它没有新生代和老年代的概念，而是将堆划分为一块块独立的 Region。当要进行垃圾收集时，首先估计每个 Region 中垃圾的数量，每次都从垃圾回收价值最大的 Region 开始回收，因此可以获得最大的回收效率。

从整体上看， G1 是基于“**标记-整理**”算法实现的收集器，从局部（两个 Region 之间）上看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。

这里抛个问题👇
一个对象和它内部所引用的对象可能不在同一个 Region 中，那么当垃圾回收时，是否需要扫描整个堆内存才能完整地进行一次可达性分析？

并不！每个 Region 都有一个 **Remembered Set**，用于记录本区域中所有对象引用的对象所在的区域，进行可达性分析时，只要在 GC Roots 中再加上 Remembered Set 即可防止对整个堆内存进行遍历。

如果不计算维护 Remembered Set 的操作，G1 收集器的工作过程分为以下几个步骤：

- **初始标记**：Stop The World，仅使用一条初始标记线程对所有与 GC Roots 直接关联的对象进行标记。
- **并发标记**：使用**一条**标记线程与用户线程并发执行。此过程进行可达性分析，速度很慢。
- **最终标记**：Stop The World，使用多条标记线程并发执行。
- **筛选回收**：回收废弃对象，此时也要 Stop The World，并使用多条筛选回收线程并发执行。

七、虚拟机和物理机的区别是什么？

这两种机器都有代码执行的能力，但是：

- **物理机**的执行引擎是**直接建立在处理器、硬件、指令集和操作系统层面**的。

- **虚拟机**的执行引擎是自己实现的，因此可以**自行制定**指令集和执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式。

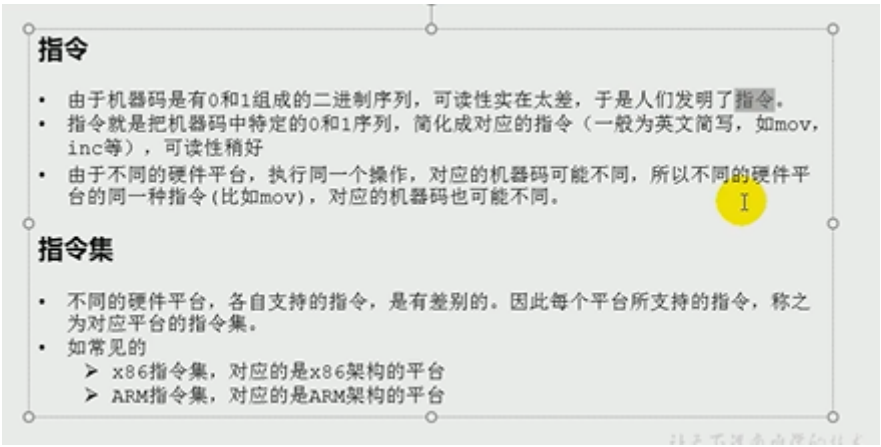
八、运行时栈帧结构

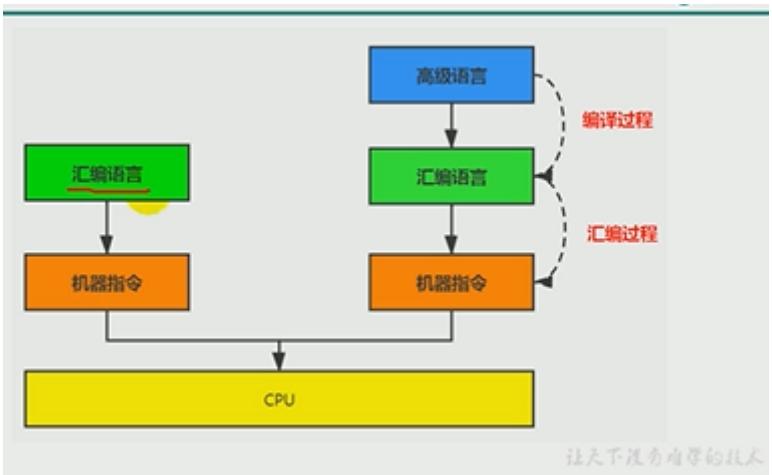
栈帧是用于支持虚拟机进行**方法调用和方法执行**的数据结构， 存储了方法的

- 局部变量表
- 操作数栈
- 动态连接
- 方法返回地址

每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

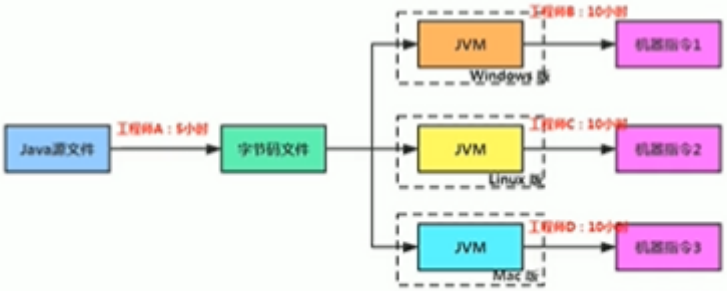
九、字节码指令





解释器

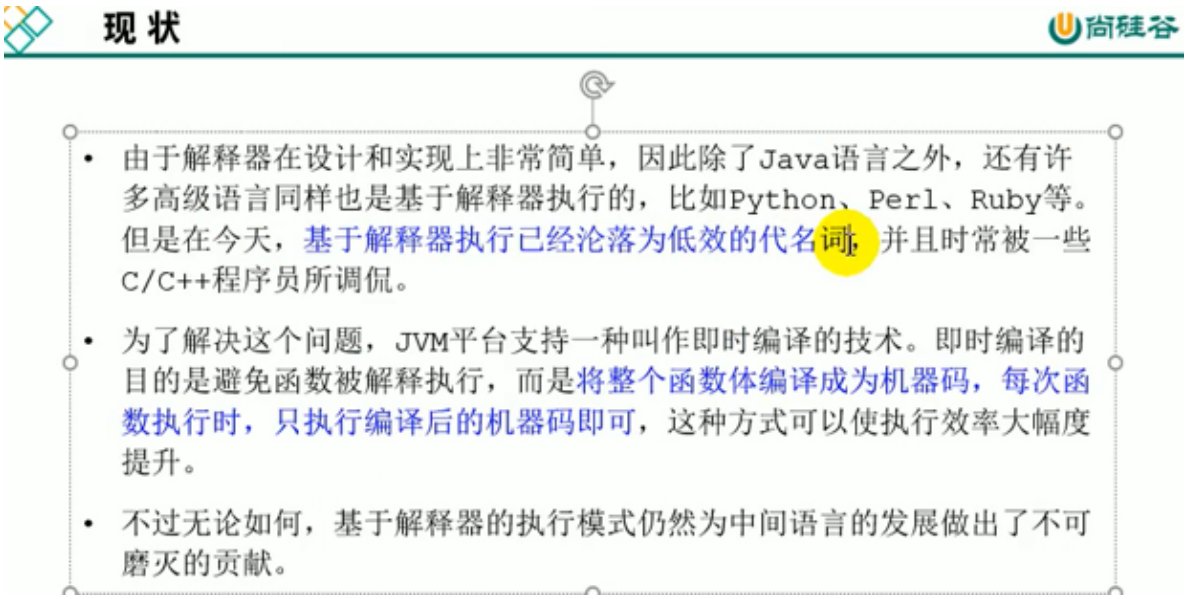
JVM设计者们的初衷仅仅只是单纯地为了满足Java程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。

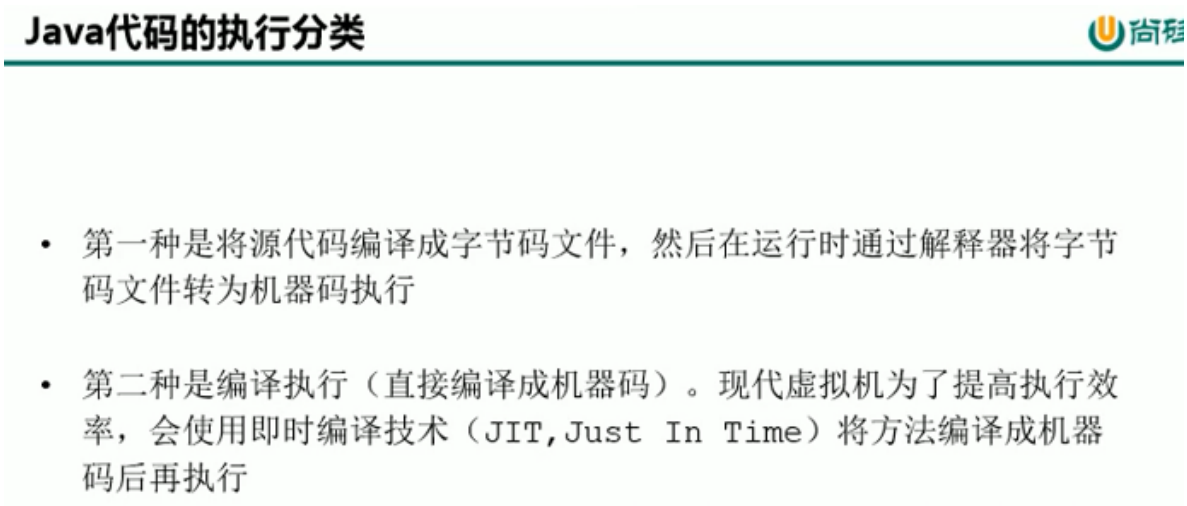


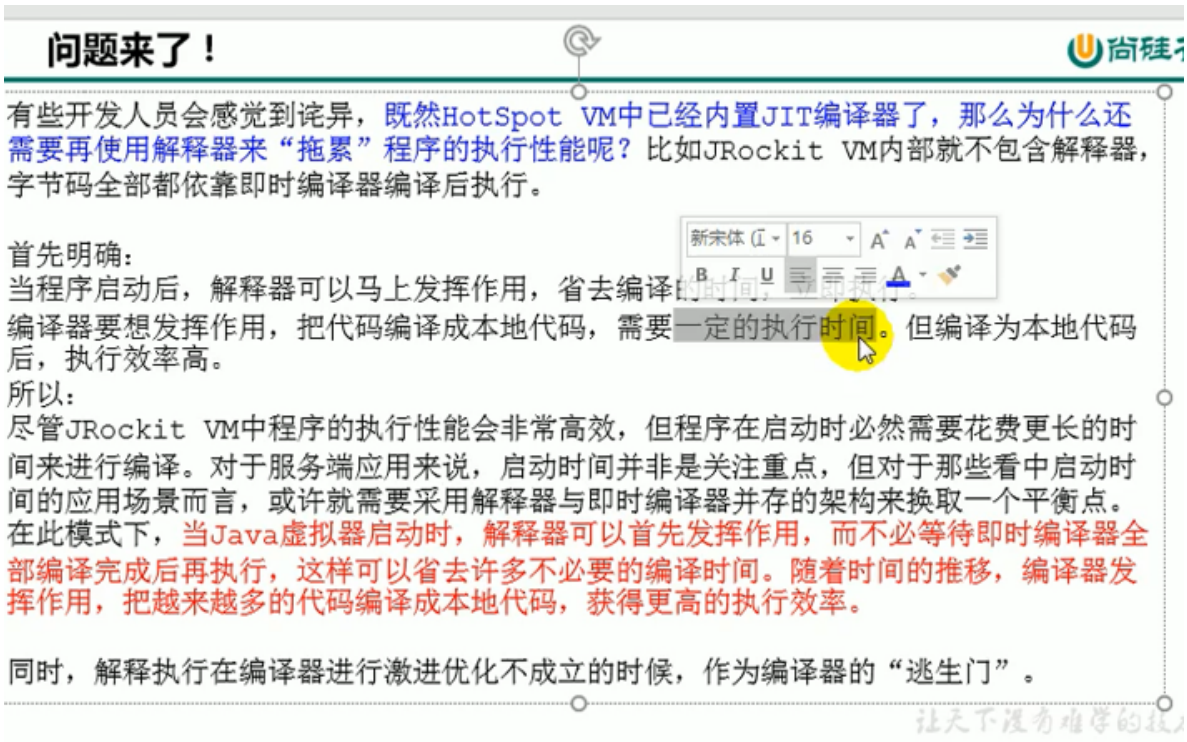
解释器工作机制（或工作任务）

- 解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为对应平台的本地机器指令执行。
- 当一条字节码指令被解释执行完成后，接着再根据PC寄存器中记录的下一条需要被执行的字节码指令执行解释操作。

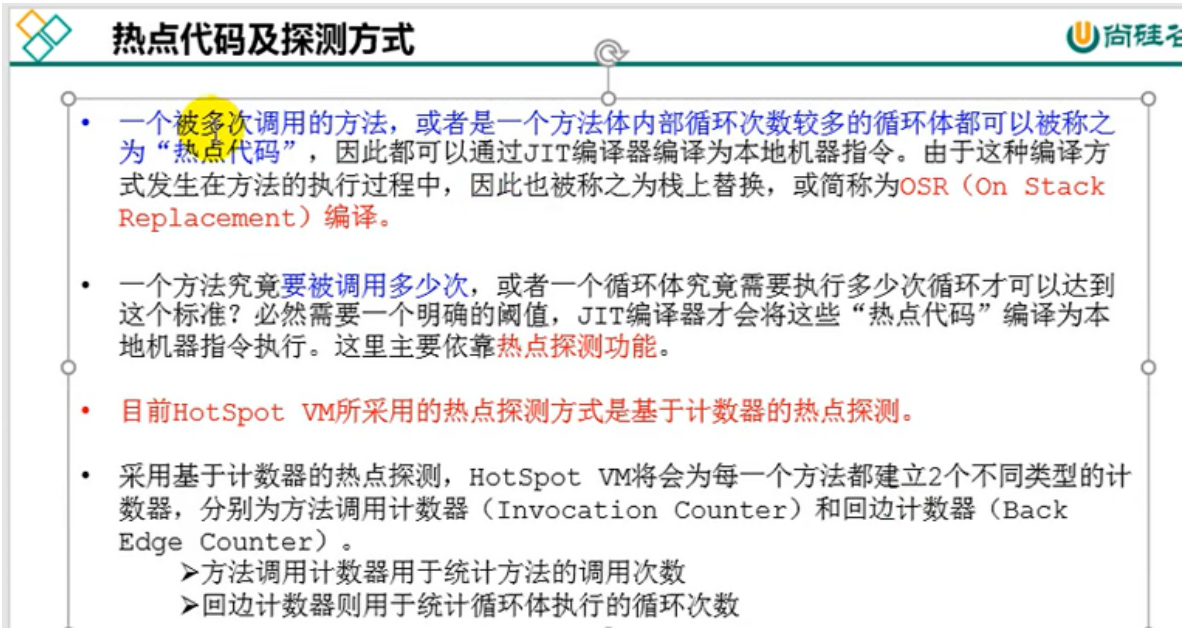
- 字节码解释器在执行时通过纯软件代码模拟字节码的执行，效率非常低下。
- 而模板解释器将每一条字节码和一个模板函数相关联，模板函数中直接产生这条字节码执行时的机器码，从而很大程度上提高了解释器的性能。
  - 在HotSpot VM中，解释器主要由Interpreter模块和Code模块构成。
    - ✓ Interpreter模块：实现了解释器的核心功能
    - ✓ Code模块：用于管理HotSpot VM在运行时生成的本地机器指令







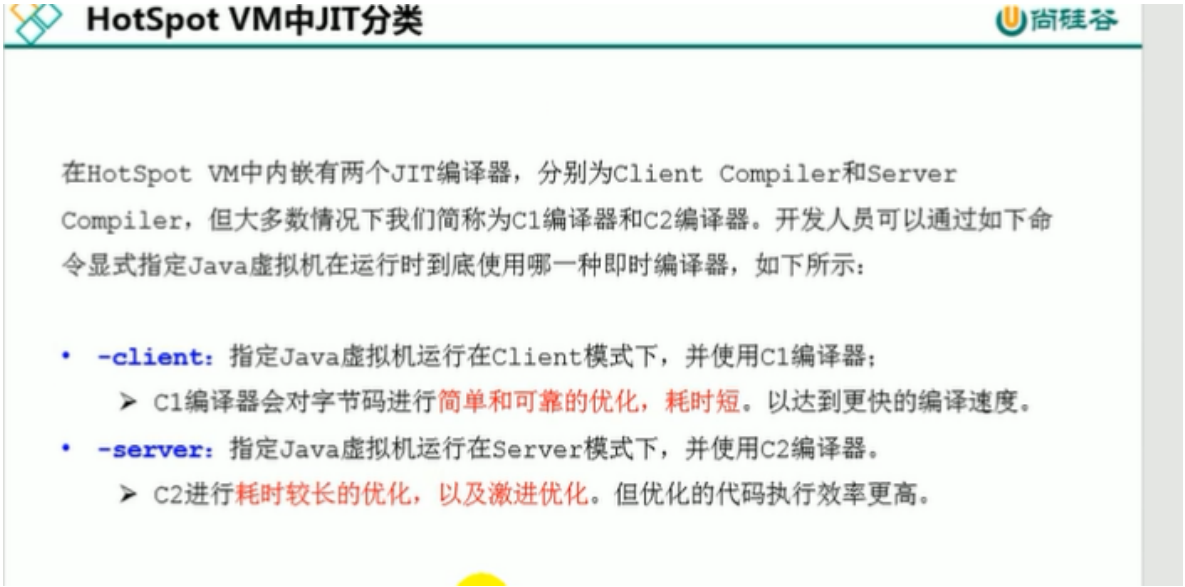
Java虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字(称为**操作码**，Opcode)以及跟随其后的零至多个代表此操作所需参数(称为**操作数**，Operands)而构成。操作码的长度为1个字节，因此**最大只有256条**，是基于**栈**的指令集架构。

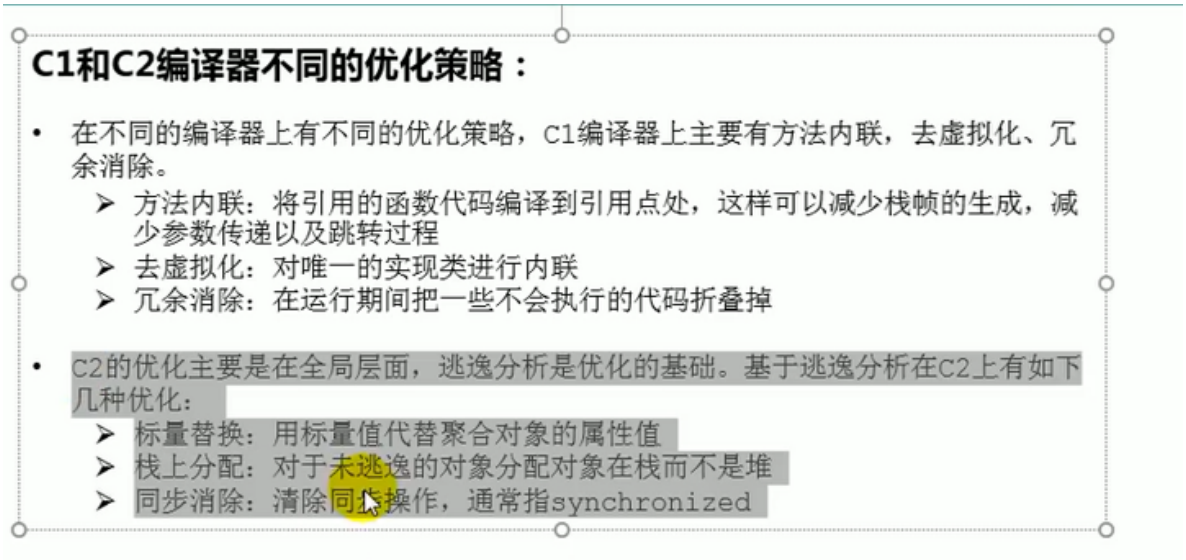


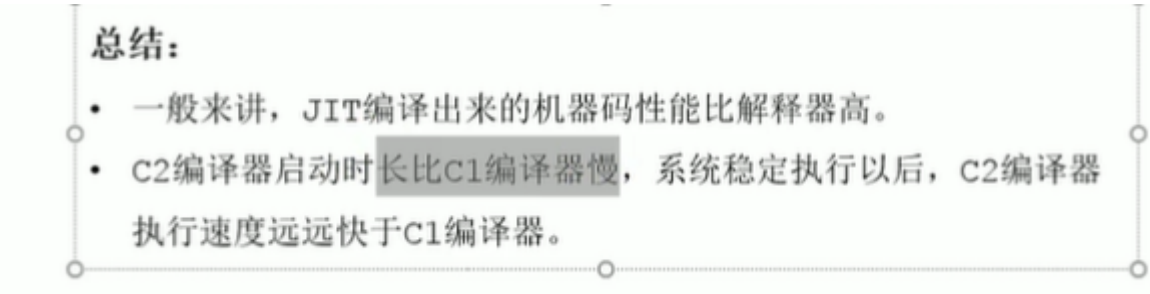
热度衰减

- 如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过**一定的时间限度**，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被**减少一半**，这个过程称为方法调用计数器热度的**衰减**（Counter Decay），而这段时间就称为此方法统计的**半衰周期**（Counter Half Life Time）。
- 进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 **-XX:-UseCounterDecay** 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。
- 另外，可以使用 **-XX:CounterHalfLifeTime** 参数设置半衰周期的时间，单位是秒。

C1(CLIENT),C2(SERVER)







字节码与数据类型

在Java虚拟机的指令集中，大多数的指令都包含了其操作所对应的数据类型信息。iload中的i表示的是int。i代表对int类型的数据操作，l代表long，s代表short，b代表byte，c代表char，f代表float，d代表double，a代表reference。也有不包含类型信息的：goto与类型无关；Arraylength操作数组类型。

加载与存储指令

加载和存储指令用于将数据在栈帧中的局部变量表和操作数栈之间来回传输，这类指令包括：

将一个局部变量加载到操作栈：iload

将一个数值从操作数栈存储到局部变量表：istore 将一个常量加载到操作数栈：bipush。 扩充局部变量表的访问索引的指令：wide

运算指令

运算或算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。大体上算术指令可以分为两种：对整型数据进行运算的指令与对浮点型数据进行运算的指令，无论是哪种算术指令，都使用Java虚拟机的数据类型，由于没有直接支持byte、short、char和boolean类型的算术指令，对于这类数据的运算，应使用操作int类型的指令代替。 注：e = a + b + c + d +e，操作数栈的深度依然是2。

类型转换指令

类型转换指令可以将两种不同的数值类型进行相互转换，这些转换操作一般用于实现用户代码中的显示类型转换操作，或者用来处理字节码指令中数据类型相关指令无法与数据类型一一对应的问题。 宽化类型处理：int i= 1；long l = i； 窄化类型处理：User user = new User(); Object obj = user; 处理窄化类型转换时，必须显式地使用转换指令来完成，这些转换指令包括：i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l和d2f。

对象创建与访问指令

虽然类实例和数组都是对象，但Java虚拟机对类实例和数组的创建与操作使用了不同的字节码指令。对象创建后，就可以通过对象访问指令获取对象实例或者数组实例中的字段或者数组元素，这些指令如下：

创建类实例的指令：new； 创建数组的指令：newarray、anewarray、multianewarray； 访问类字段(static字段，或者称为类变量)和实例字段(非static字段，或者称为实例变量)的指令：getstatic、putstatic、getfield、putfield； 把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload； 将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore； 取数组长度的指令：arraylength； 检查类实例类型的指令：instanceof、checkcast。

```
class Demo
{
    public static void main(String[ ] args){
        User user = new User();
        User[] users = new User[10];
        int[] is = new int[10];

        user.name = "hello";
        String username = user.name;
    }

}

class User{
    String name;
    static int age;
}
```

1234567891011121314151617 字节码指令为：

```
Code:
      stack=2, locals=5, args_size=1
         0: new           #2                  // class User
         3: dup
         4: invokespecial #3                  // Method User."<init>":()V
         7: astore_1
         8: bipush        10
        10: anewarray     #2                  // class User
        13: astore_2
        14: bipush        10
        16: newarray       int
        18: astore_3
        19: aload_1
        20: ldc           #4                  // String hello
        22: putfield      #5                  // Field User.name:Ljava/lang/String;
        25: aload_1
        26: getfield      #5                  // Field User.name:Ljava/lang/String;
        29: astore        4
        31: return
```

操作数栈管理指令 如同操作一个普通数据结构中的堆栈那样，Java虚拟机提供了一些用于直接操作操作数栈的指令，包括：

将操作数栈的栈顶一个或两个元素出栈：pop、pop2；（不常用） 复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、dup_x2、dup2_x2； 将栈最顶端的两个数值互换:swap。

控制转移指令 控制转移指令可以让Java虚拟机有条件或无条件地从指定的位置指令而不是控制转移指令的下一条指令继续执行程序，从概念模型上理解，可以认为控制转移指令就是在有条件或无条件地修改PC寄存器的值。如：goto等。 方法调用

invokevirtual:用于调用对象的实例方法，根据对象的实际类型进行分派(虚方法分派)，这也是Java语言中最常见的方法分派方式。 invokeinterface:用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。 invokespecial：用于调用一些需要特殊处理的实例方法，包括实例初始化方法、私有方法和父类方法。 invokestatic:用于调用类方法(static 方法)

异常处理指令 在Java程序中显示抛出异常的操作（throw语句）都由athrow指令来实现，除了用throw语句显示抛出异常情况之外，Java虚拟机规范还规定了许多运行时异常会在其他Java虚拟机指令检测到异常状况时自动抛出。而在Java虚拟机中，处理异常(catch语句)不是由字节码指令来实现的，而是采用异常表来完成的。

1. 什么是方法调用?

方法调用唯一的任务是**确定被调用方法的版本**（调用哪个方法），暂时还不涉及方法内部的具体运行过程。

2. Java的方法调用，有什么特殊之处?

Class文件的编译过程**不包含**传统编译的**连接步骤**，一切方法调用在Class文件里面存储的都只是**符号引用**，而不是方法在实际运行时内存布局中的入口地址。这使得Java有强大的动态扩展能力，但使Java方法的调用过程变得相对复杂，需要在类加载期间甚至到运行时才能确定目标方法的直接引用。

3. Java虚拟机调用字节码指令有哪些?

- invokestatic：调用静态方法
- invokespecial：调用实例构造器方法、私有方法和父类方法
- invokevirtual：调用所有的虚方法
- invokeinterface：调用接口方法

4. 虚拟机是如何执行方法里面的字节码指令的?

- 解释执行（通过解释器执行）
- 编译执行（通过即时编译器产生本地代码）

当主流的虚拟机中都包含了即时编译器后，Class文件中的代码到底会被解释执行还是编译执行，只有虚拟机自己才能准确判断。

Javac编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一动作是在Java虚拟机之外进行的，而解释器在虚拟机的内部，所以Java程序的编译是半独立的实现。

十、基于栈的指令集和基于寄存器的指令集

1. 什么是基于栈的指令集?

Java编译器输出的指令流，里面的指令大部分都是零地址指令，它们依赖**操作数栈**进行工作。

计算“1+1=2”，基于栈的指令集是这样的：

```
iconst_1
iconst_1
iadd
istore_0
```

两条iconst_1指令连续地把两个常量1压入栈中，iadd指令把栈顶的两个值出栈相加，把结果放回栈顶，最后istore_0把栈顶的值放到局部变量表的第0个Slot中。

2. 什么是基于寄存器的指令集?

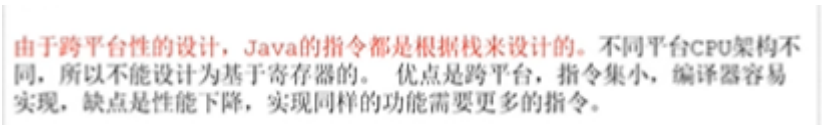
最典型的是x86的地址指令集，依赖寄存器工作。

计算“1+1=2”，基于寄存器的指令集是这样的：

```
mov eax, 1
add eax, 1
```

mov指令把EAX寄存器的值设为1，然后add指令再把这个值加1，结果就保存在EAX寄存器里。

3. 基于栈的指令集的优缺点?



优点：

- 可移植性好：用户程序不会直接用到这些寄存器，由虚拟机自行决定把一些访问最频繁的数据（程序计数器、栈顶缓存）放到寄存器以获取更好的性能。

- 代码相对紧凑：字节码中每个字节就对应一条指令
- 编译器实现简单：不需要考虑空间分配问题，所需空间都在栈上操作

缺点：

- 执行速度稍慢
- 完成相同功能所需的指令数多

频繁的访问栈，意味着频繁的访问内存，相对于处理器，内存才是执行速度的瓶颈。

十一、Javac编译过程分为哪些步骤?

1. 解析与填充符号表
2. 插入式注解处理器的注解处理
3. 分析与字节码生成

图 10-4　Javac 的编译过程[①]

十二、什么是即时编译器?

Java程序最初是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“**热点代码**”（Hot Spot Code）。

为了提高热点代码的执行效率，在**运行时**，虚拟机将会把这些代码**编译成与本地平台相关的机器码**，并进行**各种层次的优化**，完成这个任务的编译器成为**即时编译器**（Just In Time Compiler，JIT编译器）。

十三、解释器和编译器

许多主流的商用虚拟机，都同时包含解释器和编译器。

- 当程序需要**快速启动和执行**时，**解释器**首先发挥作用，省去编译的时间，立即执行。
- 当程序运行后，随着时间的推移，**编译器**逐渐发挥作用，把越来越多的代码编译成本地代码，可以**提高执行效率。**

如果内存资源限制较大（部分嵌入式系统），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。同时编译器的代码还能退回成解释器的代码。

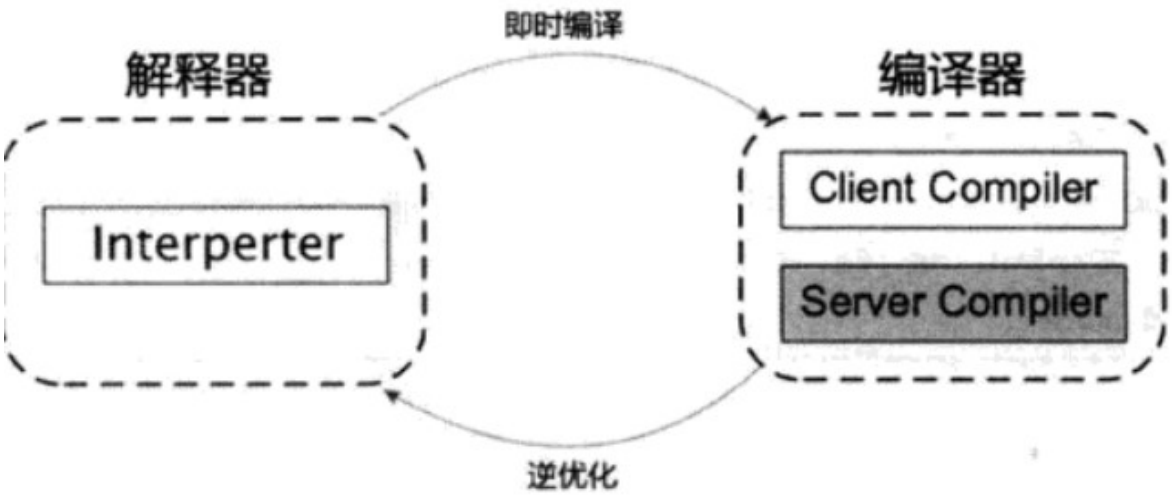


图 11-1　解释器与编译器的交互

1. 为什么要采用分层编译?

因为即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码，所花费的时间越长。

2. 分层编译器有哪些层次?

分层编译根据编译器编译、优化的规模和耗时，划分不同的编译层次，包括：

- **第0层**：程序解释执行，解释器不开启性能监控功能，可触发第1层编译。
- **第1层**：也称为C1编译，将字节码编译为本地代码，进行简单可靠的优化，如有必要加入性能监控的逻辑。
- **第2层**：也称为C2编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

用Client Compiler和Server Compiler将会同时工作。用Client Compiler获取更高的编译速度，用Server Compiler获取更好的编译质量。

十四、编译对象与触发条件

1. 热点代码有哪些?

- 被多次调用的方法
- 被多次执行的循环体

2. 如何判断一段代码是不是热点代码?

要知道一段代码是不是热点代码，是不是需要触发即时编译，这个行为称为**热点探测**。主要有两种方法：

- **基于采样的热点探测**，虚拟机周期性检查各个线程的栈顶，如果发现某个方法经常出现在栈顶，那这个方法就是“热点方法”。实现简单高效，但是很难精确确认一个方法的热度。
- **基于计数器的热点探测**，虚拟机会为每个方法建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是热点方法。

HotSpot虚拟机使用第二种，有两个计数器：

- 方法调用计数器
- 回边计数器（判断循环代码）

3. 方法调用计数器统计方法

统计的是一个相对的执行频率，即一段时间内方法被调用的次数。当超过**一定的时间限度**，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的**热度衰减**，这个时间就被称为**半衰周期**。

十五、有哪些经典的优化技术（即时编译器）?

- 语言无关的经典优化技术之一：公共子表达式消除
- 语言相关的经典优化技术之一：数组范围检查消除
- 最重要的优化技术之一：方法内联
- 最前沿的优化技术之一：逃逸分析

1. 公共子表达式消除

普遍应用于各种编译器的经典优化技术，它的含义是：

如果一个表达式 E 已经被计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成了公共子表达式。没有必要重新计算，直接用结果代替 E 就可以了。

2. 数组边界检查消除

因为Java会自动检查数组越界，每次数组元素的读写都带有一次隐含的条件判定操作，对于拥有大量数组访问的程序代码，这无疑是一种性能负担。

如果数组访问发生在循环之中，并且使用循环变量来进行数组访问，如果编译器只要通过数据流分析就可以判定循环变量的取值范围永远在数组区间内，那么整个循环中就可以把数组的上下界检查消除掉，可以节省很多次的条件判断操作。

3. 方法内联

内联消除了方法调用的成本，还为其他优化手段建立良好的基础。

编译器在进行内联时，如果是非虚方法，那么直接内联。如果遇到虚方法，则会查询当前程序下是否有多个目标版本可供选择，如果查询结果只有一个版本，那么也可以内联，不过这种内联属于**激进优化**，需要预留一个**逃生门**（Guard条件不成立时的Slow Path），称为**守护内联**。

如果程序的后续执行过程中，虚拟机一直没有加载到会令这个方法的接受者的继承关系发现变化的类，那么内联优化的代码可以一直使用。否则需要抛弃掉已经编译的代码，退回到解释状态执行，或者重新进行编译。

4. 逃逸分析

逃逸分析的基本行为就是**分析对象动态作用域**：当一个对象在方法里面被定义后，它可能被外部方法所引用，这种行为被称为**方法逃逸**。被外部线程访问到，被称为**线程逃逸**。

十六、如果对象不会逃逸到方法或线程外，可以做什么优化?

- **栈上分配**：一般对象都是分配在Java堆中的，对于各个线程都是共享和可见的，只要持有这个对象的引用，就可以访问堆中存储的对象数据。但是垃圾回收和整理都会耗时，如果一个对象不会逃逸出方法，可以让这个对象在栈上分配内存，对象所占用的内存空间就可以随着栈帧出栈而销毁。如果能使用栈上分配，那大量的对象会随着方法的结束而自动销毁，垃圾回收的压力会小很多。
- **同步消除**：线程同步本身就是很耗时的过程。如果逃逸分析能确定一个变量不会逃逸出线程，那这个变量的读写肯定就不会有竞争，同步措施就可以消除掉。
- **标量替换**：不创建这个对象，直接创建它的若干个被这个方法使用到的成员变量来替换。

十七、Java与C/C++的编译器对比

1. 即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力。
2. Java语言虽然没有virtual关键字，但是使用虚方法的频率远大于C++，所以即时编译器进行优化时难度要远远大于C++的静态优化编译器。
3. Java语言是可以动态扩展的语言，运行时加载新的类可能改变程序类型的继承关系，使得全局的优化难以进行，因为编译器无法看见程序的全貌，编译器不得不时刻注意并随着类型的变化，而在运行时撤销或重新进行一些优化。
4. Java语言对象的内存分配是在堆上，只有方法的局部变量才能在栈上分配。C++的对象有多种内存分配方式。

十八、物理机如何处理并发问题?

运算任务，除了需要**处理器计算**之外，还需要与**内存交互**，如读取运算数据、存储运算结果等（不能仅靠寄存器来解决）。

计算机的存储设备和处理器的运算速度差了几个数量级，所以不得不加入一层读写速度尽可能接近处理器运算速度的**高速缓存**（Cache），作为内存与处理器之间的缓冲：将运算需要的数据复制到缓存中，让运算快速运行。当运算结束后再从缓存同步回内存，这样处理器就无需等待缓慢的内存读写了。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是引入了一个新的问题：**缓存一致性。**在多处理器系统中，每个处理器都有自己的高速缓存，它们又共享同一主内存。当多个处理器的运算任务都涉及同一块主内存时，可能导致各自的缓存数据不一致。

为了解决一致性的问题，需要各个处理器访问缓存时遵循**缓存一致性协议。**同时为了使得处理器充分被利用，处理器可能会对输出代码进行**乱序执行优化。**Java虚拟机的即时编译器也有类似的**指令重排序优化。**

十九、讲讲什么情况下回出现内存溢出，内存泄漏?

内存泄漏的原因：对象是可达的(一直被引用)，但是对象不会被使用。

内存溢出的原因：

- 内存泄露导致堆栈内存不断增大，从而引发内存溢出。
- 大量的jar，class文件加载，装载类的空间不够，溢出
- 操作大量的对象导致堆内存空间已经用满了，溢出
- nio直接操作内存，内存过大导致溢出

解决：

- 查看程序是否存在内存泄漏的问题
- 设置参数加大空间
- 代码中是否存在死循环或循环产生过多重复的对象实体、
- 查看是否使用了nio直接操作内存。

二十、JVM 年轻代到年老代的晋升过程的判断条件是什么呢?

1. 部分对象会在From和To区域中复制来复制去,**如此交换15次**(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代。
2. 如果**对象的大小大于Eden的二分之一会直接分配在old**，如果old也分配不下，会做一次majorGC，如果小于eden的一半但是没有足够的空间，就进行minorgc也就是新生代GC。
3. minor gc后，survivor仍然放不下，则放到老年代
4. 动态年龄判断，大于等于某个年龄的对象超过了survivor空间一半，大于等于某个年龄的对象直接进入老年代

二十一、JVM 出现 fullGC 很频繁，怎么去线上排查问题

这题就依据full GC的触发条件来做：

- 如果有perm gen的话(jdk1.8就没了)，**要给perm gen分配空间，但没有足够的空间时**，会触发full gc。

  所以看看是不是perm gen区的值设置得太小了。

- `System.gc()`方法的调用

  这个一般没人去调用吧~~~

- 当**统计**得到的Minor GC晋升到旧生代的平均大小**大于老年代的剩余空间**，则会触发full gc(这就可以从多个角度上看了)

  是不是**频繁创建了大对象(也有可能eden区设置过小)**(大对象直接分配在老年代中，导致老年代空间不足--->从而频繁gc) 是不是**老年代的空间设置过小了**(Minor GC几个对象就大于老年代的剩余空间了)

二十二、类的实例化顺序

1. 父类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
2. 子类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
3. 父类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
4. 父类构造方法
5. 子类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
6. 子类构造方法

二十三、JVM 中一次完整的 GC 流程（从 ygc 到 fgc）是怎样的

- YGC： **对新生代堆进行gc。**频率比较高，因为大部分对象的存活寿命较短，在新生代里被回收。性能耗费较小。
- FGC： **全堆范围的gc。**默认堆空间使用到达80%(可调整)的时候会触发fgc。以我们生产环境为例，一般比较少会触发fgc，有时10天或一周左右会有一次。

什么时候执行YGC和FGC

- eden空间不足,执行 young gc
- old空间不足，perm空间不足，调用方法`System.gc()`，ygc时的悲观策略, dump live的内存信息时(jmap –dump:live)，都会执行full gc