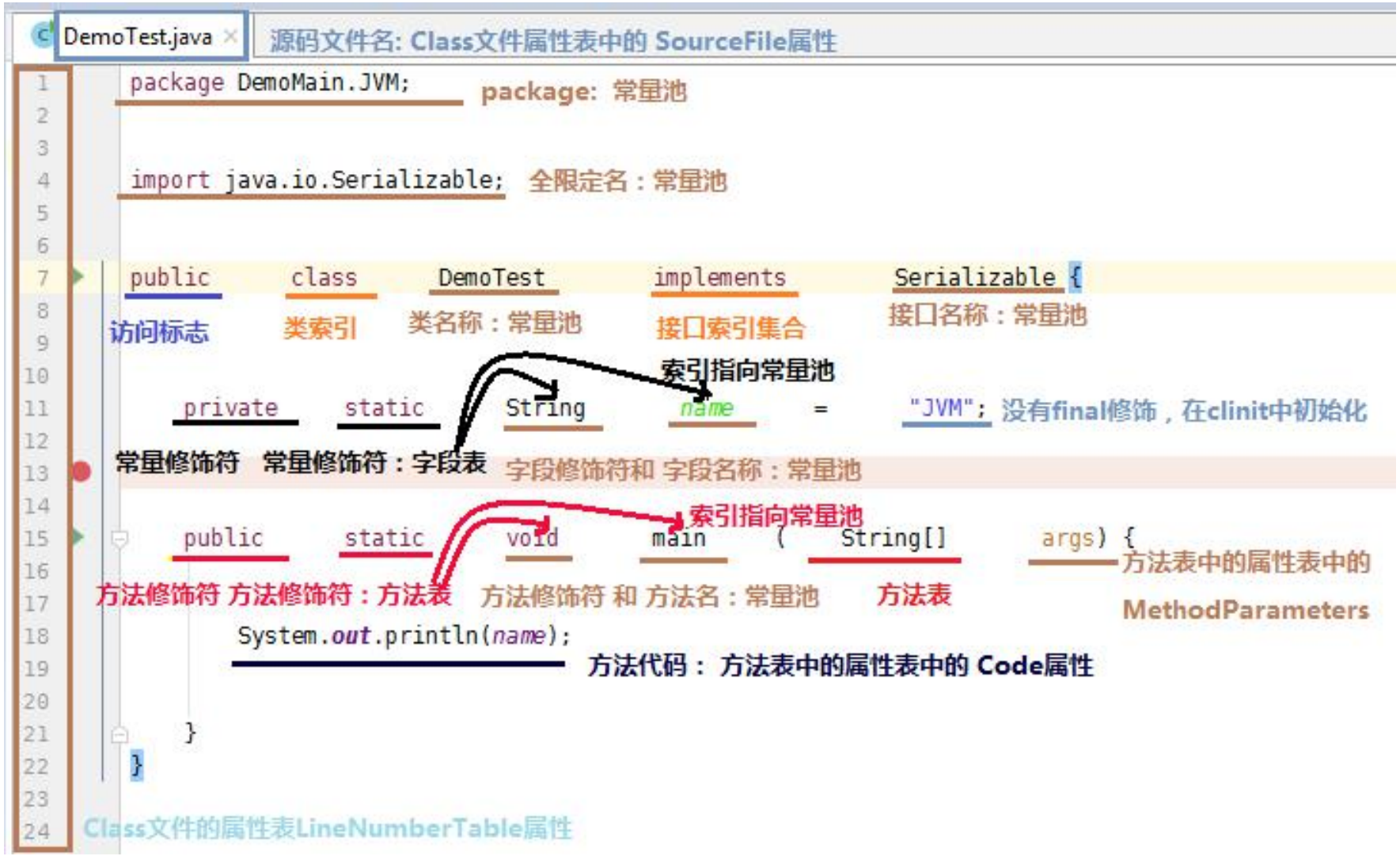
1. 谈谈你对Class文件结构的了解



Class文件是一组以8个字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在文件之中，中间没有添加任何分隔符，这使得整个Class文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。

根据《Java虚拟机规范》的规定，Class文件格式采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：“**无符号数**”和“**表**”。

无符号数属于基本的数据类型，以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成字符串值。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，为了便于区分，所有表的命名都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上也可以视作是一张表，这张表由表6-1所示的数据项按严格顺序排列构成。

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count - 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

2. class文件的魔数和主次版本号

每个Class文件的头4个字节被称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件。

紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是次版本号（Minor Version），第7和第8个字节是主版本号（Major Version）。Java的版本号是从45开始的，JDK 1.1之后的每个JDK大版本发布主版本号向上加1

3. 为什么常量池计数器从1 开始，而不是从0 开始

由于常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项u2类型的数据，代表常量池容量计数值（constant_pool_count）。与Java中语言习惯不同，这个容量计数是从1而不是0开始的，如下图所示，常量池容量（偏移地址：0x00000008）为十六进制数0x0016，即十进制的22，这就代表常量池中有21项常量，索引值范围为1～21。在Class文件格式规范制定之时，设计者将**第0项常量空出来是有特殊考虑的，这样做的目的在于，如果后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义**，可以把索引值设置为0来表示。Class文件结构中只有常量池的容量计数是从1开始，对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从0开始。

举例：匿名内部类本身没有类名称，进行名称引用时会将index指向0；Object类的文件结构的父类索引指向0；

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	32	00	16	07	00	02	01	00	1D
00000010	6F	72	67	2F	66	65	6E	69	78	73	6F	66	74	2F	63	6C
00000020	61	7A	7A	2F	54	65	73	74	43	6C	61	73	73	07	00	04
00000030	01	00	10	6A	61	76	61	2F	6						62	6A
00000040	65	63	74	01	00	01	6D	01	0						3C	69
00000050	6E	69	74	3E	01	00	03	28	2						6F	64

数据解释器

8 Bit (+): 22

4. Class文件常量池中存放的什么内容

常量池中主要存放两大类常量：**字面量**（Literal）和**符号引用**（Symbolic References）。

字面量比较接近于Java语言层面的常量概念，如文本字符串、被声明为final的常量值等。

符号引用则属于编译原理方面的概念，主要包括下面几类常量：

·被模块导出或者开放的包（Package）
·类和接口的全限定名（Fully Qualified Name）
·字段的名称和描述符（Descriptor）
·方法的名称和描述符
·方法句柄和方法类型（Method Handle、Method Type、Invoke Dynamic）
·动态调用点和动态常量（Dynamically-Computed Call Site、Dynamically-Computed Constant）

DemoTest.java

```
1  package DemoMain.JVM;          package: 常量池
2
3
4  import java.io.Serializable;   全限定名：常量池
5
6
7  public class DemoTest implements Serializable {
8                          类名称：常量池          接口名称：常量池
9
10
11     private static String name = "JVM";
12
13                         字段修饰符和 字段名称：常量池
14
15     public static void main ( String[] args) {
16
17                         方法修饰符 和 方法名：常量池
18         System.out.println(name);
19
20
21     }
22 }
```

5. Java 字段名和方法名长度限制是多少，为什么?

首先我们先要明确，方法名称和字段名称都是存储于常量池中的。而存储这两个名称需要用到常量池中的CONSTANT_Utf8_info 类型来存储。我们先来看下CONSTANT_Utf8_info的存储结构。

CONSTANT_Utf8_info型常量的结构

类　　型	名　　称	数　　量
u1	tag	1
u2	length	1
u1	bytes	length

CONSTANT_Utf8_info型常量的最大长度也就是Java中方法、字段名的最大长度。而这里的最大长度就是length的最大值，既u2类型能表达的最大值65535。所以Java程序中如果定义了超过64KB英文字符的变量或方法名，即使规则和全部字符都是合法的，也会无法编译。

6. Class文件结构中的访问标志作用。

在常量池结束之后，紧接着的2个字节代表访问标志（access_flags），这个标志用于识别一些类或者接口层次的访问信息，包括：这个Class是类还是接口；是否定义为public类型；是否定义为abstract类型；如果是类的话，是否被声明为final；等等。

标 志 名 称	标志值	含　　义
ACC_PUBLIC	0x0001	是否为 public 类型
ACC_FINAL	0x0010	是否被声明为 final，只有类可设置
ACC_SUPER	0x0020	是否允许使用 invokespecial 字节码指令的新语义，invokespecial 指令的语义在 JDK 1.0.2 发生过改变，为了区别这条指令使用哪种语义，JDK 1.0.2 之后编译出来的类的这个标志都必须为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口或者抽象类来说，此标志值为真，其他类型值为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举
ACC_MODULE	0x8000	标识这是一个模块

7. 类索引，父类索引和接口索引集合的作用，并举例详细说说类索引过程

DemoTest.java

```
1  package DemoMain.JVM;
2
3
4  import java.io.Serializable;
5
6
7  public class DemoTest implements Serializable {
8
9
10
11     private static String name = "JVM";
12
13
14
15     public static void main ( String[] args) {
16
17
18         System.out.println(name);
19
20
21     }
22 }
```

package: 常量池

全限定名：常量池

访问标志　类索引　类名称：常量池　接口索引集合　接口名称：常量池

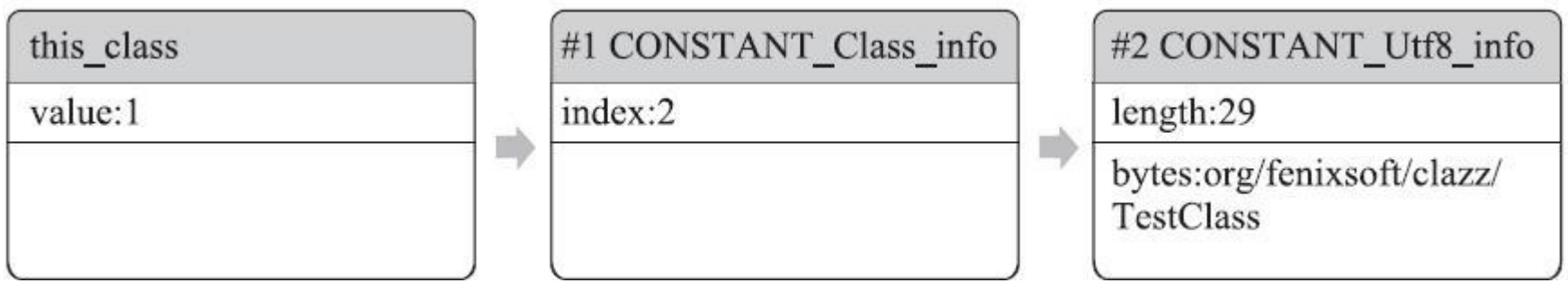
字段修饰符和 字段名称：常量池

方法修饰符 和 方法名：常量池

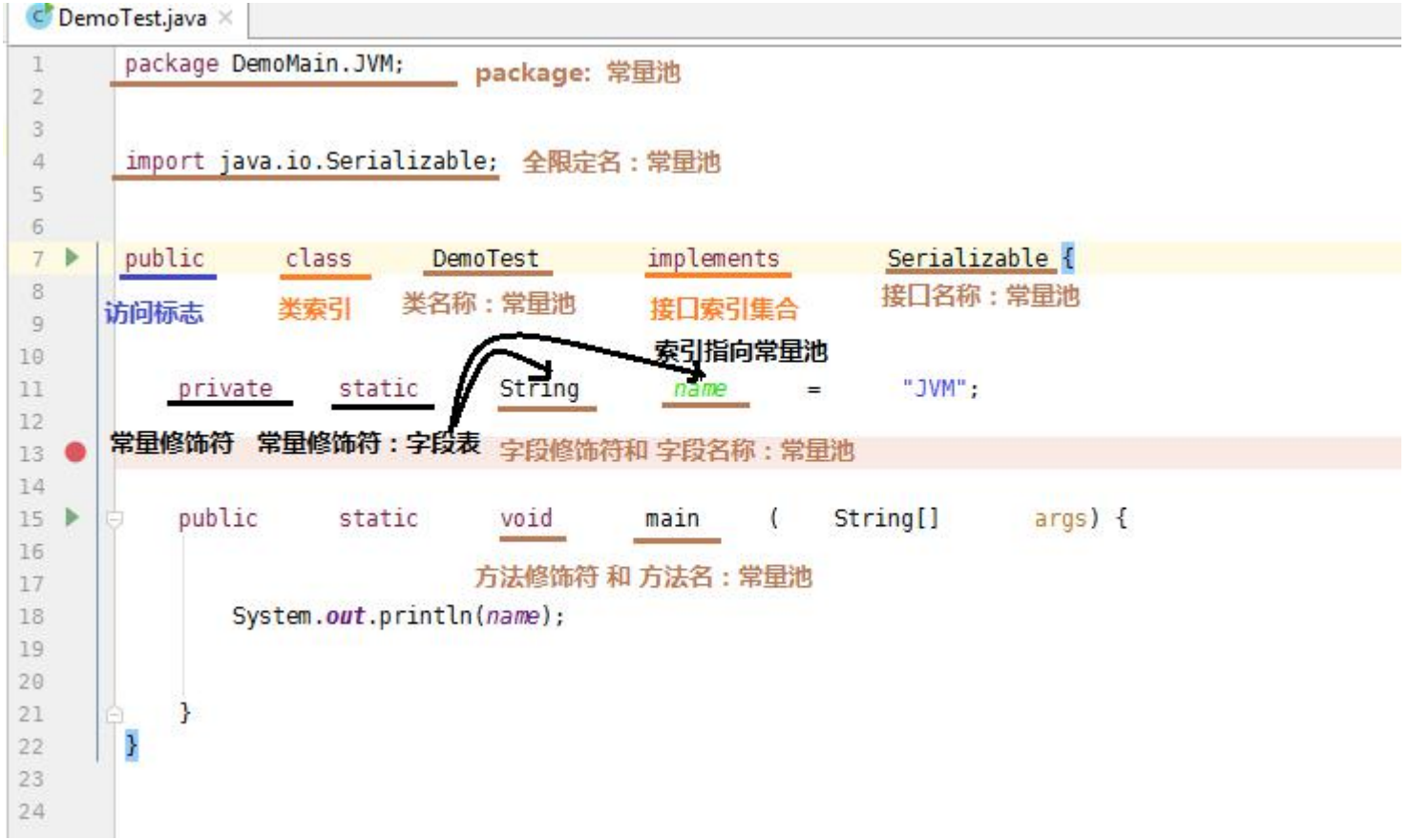
类索引（this_class）和**父类索引**（super_class）都是一个u2类型的数据，而**接口索引集合**（interfaces）是一组u2类型的数据的集合，Class文件中由这三项数据来**确定该类型的继承实现关系**。

类索引用于确定这个**类的全限定名（全限定名称存储于常量池）**，**父类索引**用于确定这个类的**父类的全限定名。（全限定名称存储于常量池）。**这里说的索引，是指向常量池中的constant_class_info类型，constant_class_info 又指向了constanct_utf8_info 从而最终找到全限定名。
由于Java语言不允许多重继承，所以父类索引只有一个，除了java.lang.Object之外，所有的Java类都有父类，因此除了java.lang.Object外，所有Java类的父类索引都不为0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按implements关键字（如果这个Class文件表示的是一个接口，则应当是extends关键字）后的接口顺序从左到右排列在接口索引集合中。

类索引、父类索引和接口索引集合都按顺序排列在访问标志之后，类索引和父类索引用两个u2类型的索引值表示，它们各自指向一个类型为CONSTANT_Class_info的类描述符常量，通过CONSTANT_Class_info类型的常量中的索引值可以找到定义在CONSTANT_Utf8_info类型的常量中的全限定名字符串。



8. class文件的字段表存储什么信息，与常量池有什么联系



字段表（field_info）用于描述接口或者类中声明的变量。Java语言中的“字段”（Field）包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。

字段表存储的其实是变量(非局部变量)的修饰符 + 字段描述符索引（索引指向常量池） + 字段名称索引（索引指向常量池）。

修饰符：字段可以包括的修饰符有字段的作用域（public、private、protected修饰符）、是实例变量还是类变量（static修饰符）、可变性（final）、并发可见性（volatile修饰符，是否强制从主内存读写）、可否被序列化（transient修饰符）等。

描述符：字段类型。

public final static String NUMBER="1" 。pulic final 和static 是方式的修饰符，这些都存放在class文件的字段表中。 String是字段的描述符，存放于常量池中，Number是字段的名称，存放于常量池。这两部分的关联，是通过字段表的name_index指向常量池中的字段名称number 和 descrip_index 指向常量池中的描述符。

类　　型	名　　称	数　　量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1

与access flage类似。存放 public final static

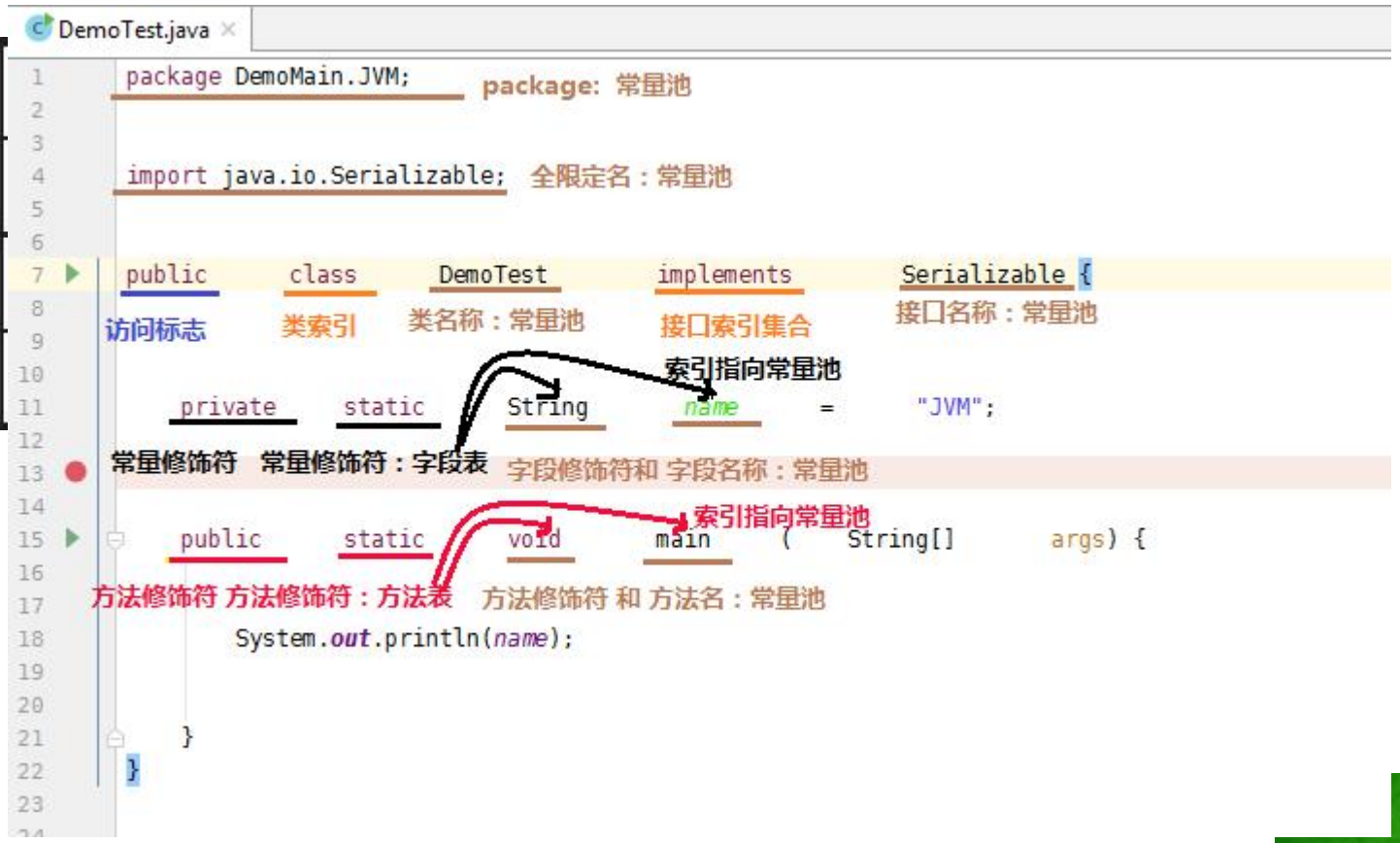
指向常量池中的类型 String

指向常量池中字段名称 name

9.class文件的方法表存储什么信息，与常量池有什么联系

Class文件存储格式中对方法的描述与对字段的描述采用了几乎完全一致的方式，方法表的结构如同字段表一样，依次包括访问标志（access_flags）、名称索引（name_index）、描述符索引（descriptor_index）、属性表集合（attributes）几项。

类　　型	名　　称
u2	access_flags
u2	name_index
u2	descriptor_index



篇

10. 说说你对Class文件结构属性表的理解及接触过的属性

属性表（attribute_info）在前面的讲解之中已经出现过数次，Class文件、字段表、方法表都可以携带自己的属性表集合，以描述某些场景专有的信息。与Class文件中其他的数据项目要求严格的顺序、长度和内容不同，属性表集合的限制稍微宽松一些，不再要求各个属性表具有严格顺序，并且《Java虚拟机规范》允许只要不与已有属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息。

对于每一个属性，它的名称都要从常量池中引用一个CONSTANT_Utf8_info类型的常量来表示，而属性值的结构则是完全自定义的，只需要通过一个u4的长度属性去说明属性值所占用的位数即可。

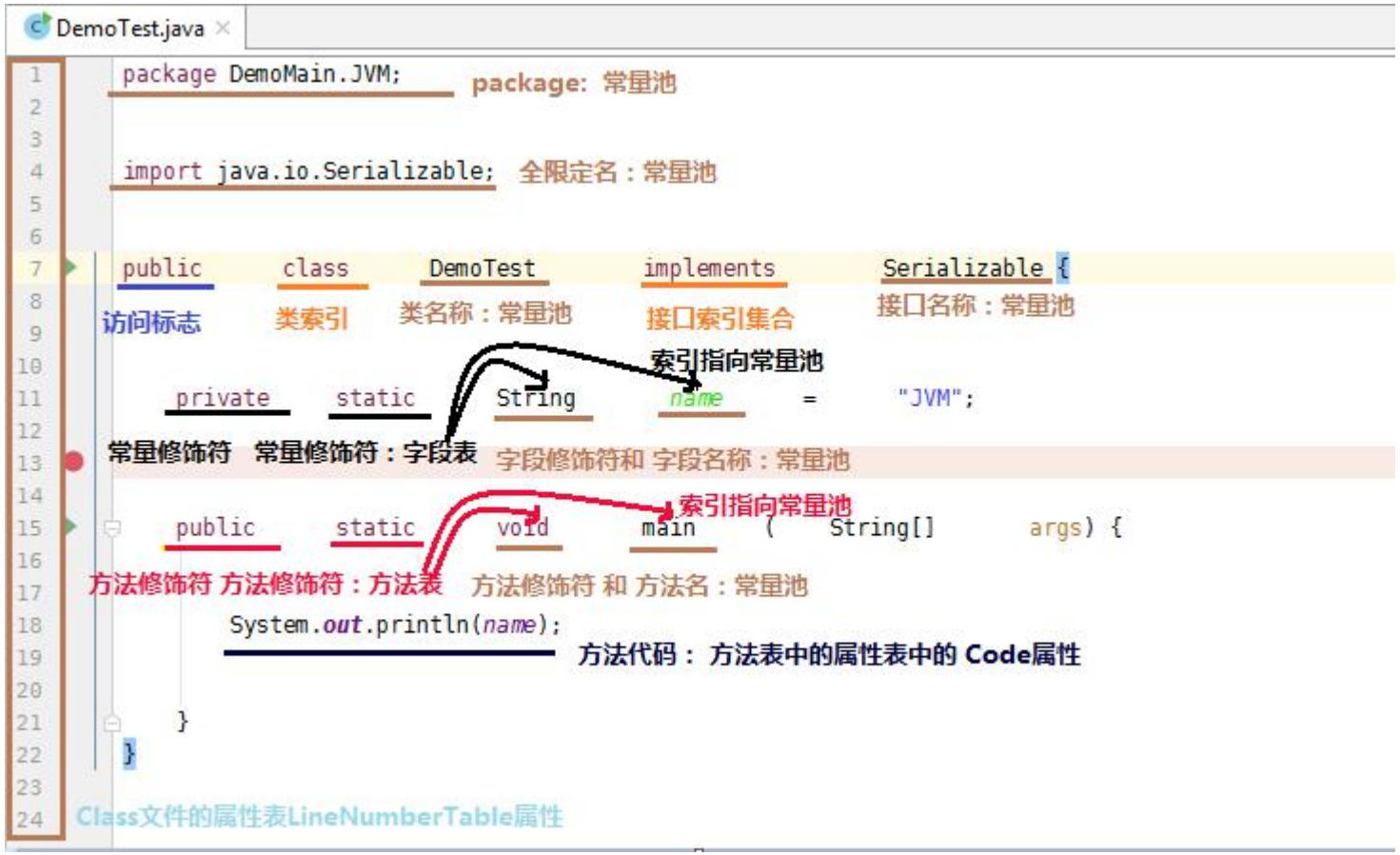
类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

Code 属性： Java程序方法体里面的代码经过Javac编译器处理之后，最终变为字节码指令存储在Code属性内。Code属性出现在方法表的属性集合之中，但并非所有的方法表都必须存在这个属性，譬如接口或者抽象类中的方法就不存在Code属性，如果方法表有Code属性存在，那么它的结构如下图所示：

类　　型	名　　称	数　　量	
u2	attribute_name_index	1	此常量值固定为“Code”
u4	attribute_length	1	
u2	max_stack	1	操作数栈深度的最大值
u2	max_locals	1	局部变量所需最大空间，按Slot计算
u4	code_length	1	方法代码长度
u1	code	code_length	方法源代码
u2	exception_table_length	1	
exception_info	exception_table	exception_table_length	
u2	attributes_count	1	
attribute_info	attributes	attributes_count	StackMapTable属性，后边说

LineNumberTable属性

LineNumberTable属性用于描述Java源码行号与字节码行号（字节码的偏移量）之间的对应关系。它并不是运行时必需的属性，但默认会生成到Class文件之中，可以在Javac中使用-g：none或-g：lines选项来取消或要求生成这项信息。如果选择不生成LineNumberTable属性，对程序运行产生的最主要影响就是当抛出异常时，堆栈中将不会显示出错的行号，并且在调试程序的时候，也无法按照源码行来设置断点。

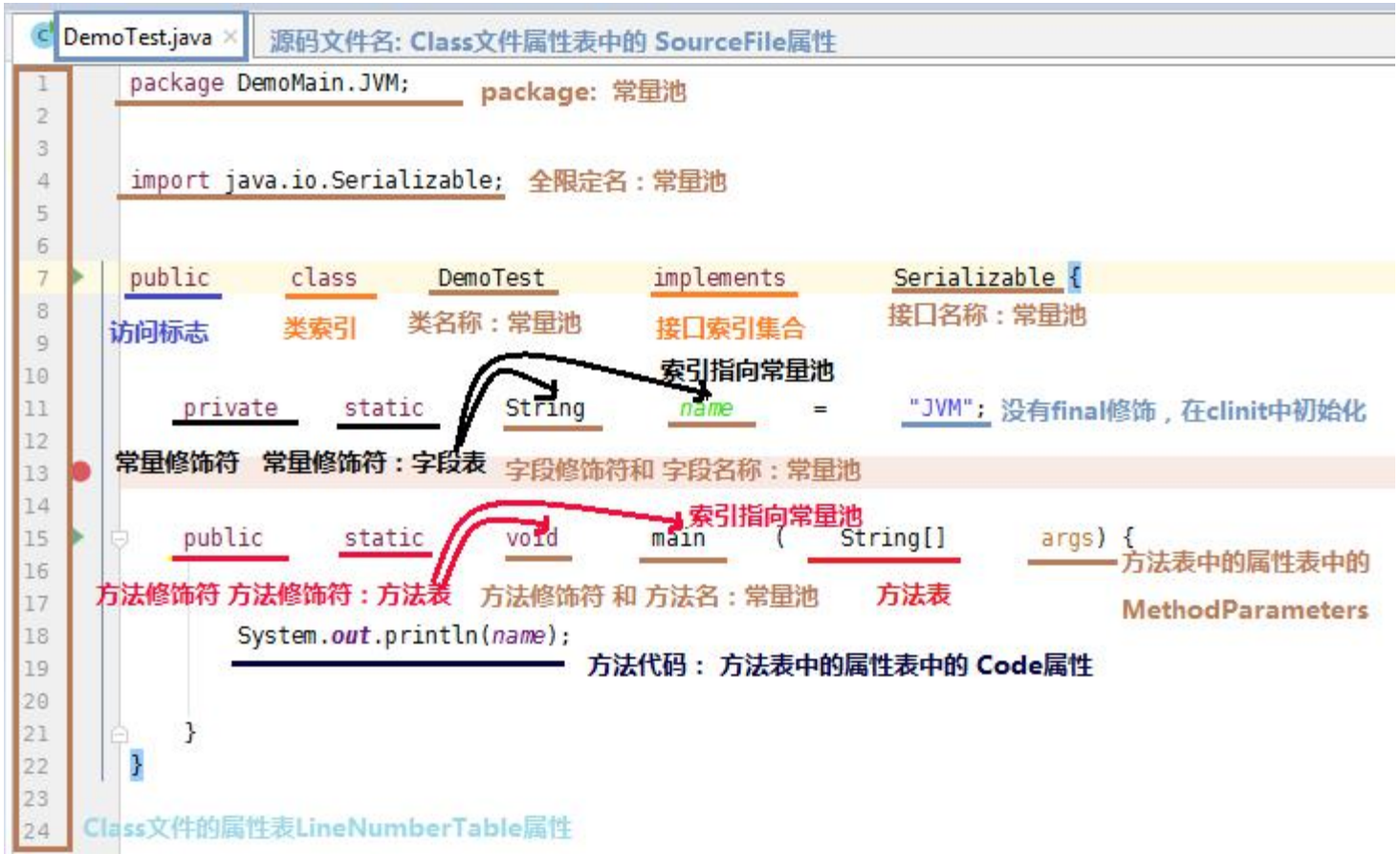


ConstantValue属性

ConstantValue属性的作用是通知虚拟机自动为静态变量赋值。只有被static关键字修饰的变量（类变量）才可以使用这项属性。类似“int x=123”和“static int x=123”这样的变量定义在Java程序里面是非常常见的事情，但虚拟机对这两种变量赋值的方式和时刻都有所不同。对非static类型的变量（也就是实例变量）的赋值是在实例构造器<init>()方法中进行的；而对于类变量，则有两种方式可以选择：在类构造器<clinit>()方法中或者使用ConstantValue属性。目前Oracle公司实现的Javac编译器的选择是，如果同时使用final和static来修饰一个变量（按照习惯，这里称“常量”更贴切），并且这个变量的数据类型是基本类型或者java.lang.String的话，就将会生成ConstantValue属性来进行初始化；如果这个变量没有被final修饰，或者并非基本类型及字符串，则将会选择在<clinit>()方法中进行初始化。

比如说: public static final String PHONE_NUM= '''123400000'; 为什么我们写代码的时候，定义String类型的静态变量的时候要这样写呢? name
还要是大写的。 因为，如果用static和final 同时
修饰string类型的变量，我们在javac编译的过程
中就把这个PHONE_NUM的值进行了初始化，放
到了constantVaule属性里。

否则，需要在类加载过程的最后一步初始化这一步调用clint方法进行初始化。



11. JVM 类加载的整体流程

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。（初次获取Class文件字节流） --加载

2）class文件格式验证 -- 链接-验证-文件格式验证

3）将这个字节流所代表的**静态存储（class文件本身）**结构转化为方法区的运行时数据结构。 -- 加载

4）在内存（Java堆）中生成这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。 -- 加载

5）元数据验证 -- 链接-验证-元数据验证

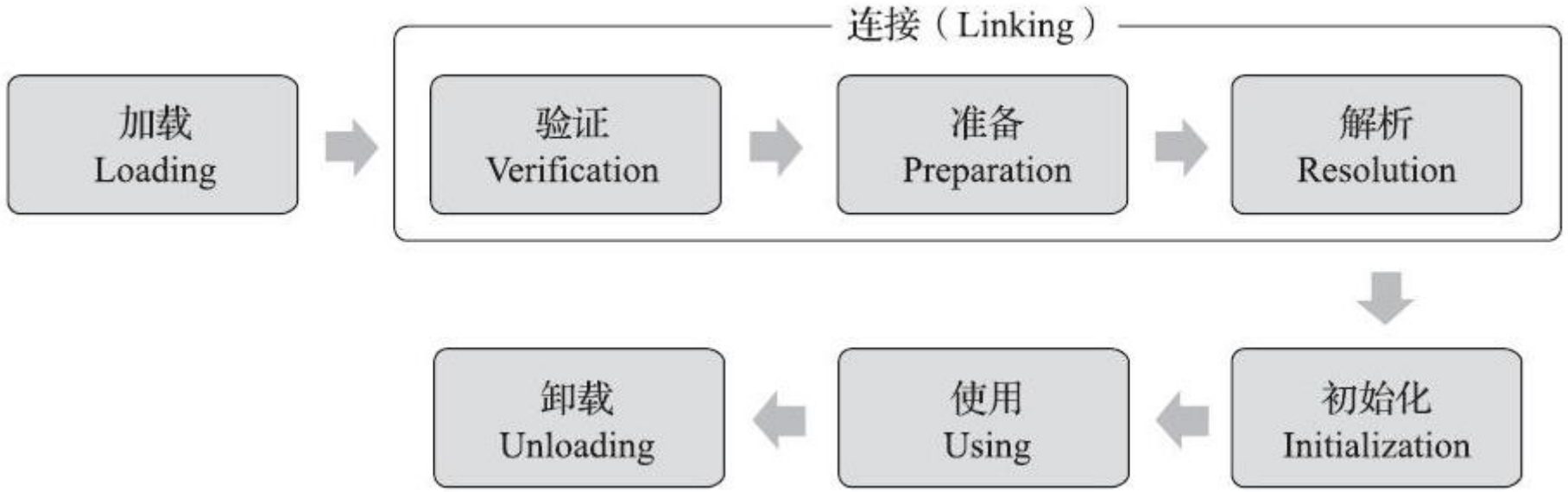
6）字节码验证 -- 链接-验证-字节码验证

7）准备 -- 链接-准备

?）解析 -- 链接-解析

??）符号引用验证 --链接-验证-符号引用验证

8）初始化 -- 初始化



准确的说，类加载是 从加载开始，但是加载和连接可能有交叉进行的部分，解析的时机也难以确定，但大体的类加载流程即是如此

12. 加载阶段JVM 具体进行的什么操作

“加载”（Loading）阶段是整个“类加载”（Class Loading）过程中的一个阶段。在加载阶段，Java虚拟机需要完成以下三件事情：

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。（初次获取Class文件字节流）

2）将这个字节流所代表的**静态存储（class文件本身）**结构转化为方法区的运行时数据结构。（Java虚拟机外部的二进制字节流就按照虚拟机所设定的格式存储在方法区之中）

3）在内存（Java堆）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。

相对于类加载过程的其他阶段，非数组类型的加载阶段（准确地说，是加载阶段中获取类的二进制字节流的动作）是开发人员可控性最强的阶段。加载阶段既可以使用Java虚拟机里内置的引导类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员通过定义自己的类加载器去控制字节流的获取方式（重写一个类加载器的findClass()或loadClass()方法），实现根据自己的想法来赋予应用程序获取运行代码的动态性。

13. JVM 加载数组和加载类有什么区别与联系

对于数组类而言，情况就有所不同，**数组类本身不通过类加载器创建**，它是**由Java虚拟机直接在内存中动态构造出来的。**但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（Element Type，指的是数组去掉所有维度的类型）最终还是要靠类加载器来完成加载，一个数组类（下面简称为C）创建过程遵循以下规则：

·如果数组的组件类型（Component Type，指的是数组去掉一个维度的类型）是引用类型，那就遵循JVM类加载过程（3个JVM 类加载器）去加载这个组件类型，**数组C将被标识在加载该组件类型的类加载器的类名称空间上**（这点很重要，一个类型必须与类加载器一起确定唯一性）。

·如果数组的组件类型不是引用类型（例如int[]数组的组件类型为int），Java虚拟机将会把数组C标记为与引导类加载器（启动类加载器，bootstarp classloader）关联。

·数组类的可访问性与它的组件类型的可访问性一致，如果组件类型不是引用类型，它的数组类的可访问性将默认为public，可被所有的类和接口访问到。

14. 验证阶段JVM主要做了什么

验证是连接阶段的第一步，这一阶段的目的是确保Class文件的字节流中包含的信息符合《Java虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。

验证阶段是非常重要的，这个阶段是否严谨，直接决定了Java虚拟机是否能承受恶意代码的攻击，从代码量和耗费的执行性能的角度上讲，验证阶段的工作量在虚拟机的类加载过程中占了相当大的比重。

从整体上看，验证阶段大致上会完成下面四个阶段的检验动作：**文件格式验证、元数据验证、字节码验证和符号引用验证。**

文件格式验证

要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括下面这些验证点：

·是否以魔数0xCAFEBABE开头。

·主、次版本号是否在当前Java虚拟机接受范围之内。

·常量池的常量中是否有不被支持的常量类型（检查常量tag标志）。

·指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。

·CONSTANT_Utf8_info型的常量中是否有不符合UTF-8编码的数据。

·Class文件中各个部分及文件本身是否有被删除的或附加的其他信息

……

这阶段的验证是基于二进制字节流进行的，**只有通过了这个阶段的验证之后，这段字节流才被允许进入Java虚拟机内存的方法区中进行存储**，所以后面的三个验证阶段全部是基于方法区的存储结构上进行的，不会再直接读取、操作字节流了。

元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合《Java语言规范》的要求，这个阶段包括的验证点如下

·这个类是否有父类（除了java.lang.Object之外，所有的类都应当有父类）。

·这个类的父类是否继承了不允许被继承的类（被final修饰的类）。

·如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。

·类中的字段、方法是否与父类产生矛盾（例如覆盖了父类的final字段，或者出现不符合规则的方法重载，例如方法参数都一致，但返回值类型却不同等）。

....

字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型校验完毕以后，这阶段就要对类的方法体**（Class文件中的Code属性）进行校验分析**，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为，例如：

·保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似于“在操作栈放置了一个int类型的数据，使用时却按long类型来加载入本地变量表中”这样的情况。

·保证任何跳转指令都不会跳转到方法体以外的字节码指令上。

·保证方法体中的类型转换总是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的。

由于数据流分析和控制流分析的高度复杂性，Java虚拟机的设计团队为了避免过多的执行时间消耗在字节码验证阶段中，在JDK 6之后的Javac编译器和Java虚拟机里进行了一项联合优化，把尽可能多的校验辅助措施挪到Javac编译器里进行。具体做法是给方法体Code属性的属性表中新增加了一项名为“StackMapTable”的新属性，这项属性描述了方法体所有的基本块（Basic Block，指按照控制流拆分的代码块）开始时本地变量表和操作栈应有的状态，在字节码验证期间，Java虚拟机就不需要根据程序推导这些状态的合法性，只需要检查**StackMapTable属性**中的记录是否合法即可。这样就将字节码验证的类型推导转变为类型检查，从而节省了大量校验时间。

符号引用验证

最后一个阶段的校验行为发生在虚拟机将符号引用转化为直接引用[3]的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。

符号引用验证可以看作是对类自身以外（常量池中的各种符号引用）的各类信息进行匹配性校验，通俗来说就是，该类是否缺少或者被禁止访问它依赖的某些外部类、方法、字段等资源。本阶段通常需要校验下列内容:

·符号引用中通过字符串描述的全限定名是否能找到对应的类。

·在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。p

·符号引用中的类、字段、方法的可访问性（private、protected、public、<package>）是否可被当前类访问。

……

符号引用验证的主要目的是确保解析行为能正常执行，如果无法通过符号引用验证，Java虚拟机将会抛出一个java.lang.IncompatibleClassChangeError的子类异常，典型的如：java.lang.IllegalAccessError、java.lang.NoSuchFieldError、java.lang.NoSuchMethodError等。

15. 类加载器链接的准备阶段做了什么

准备阶段是正式为类中定义的变量（即静态变量，被static修饰的变量）分配内存并设置类变量初始值的阶段，**从概念上讲**，这些变量所使用的内存都应当在方法区中进行分配，但必须注意到方法区本身是一个逻辑上的区域，在JDK 7及之前，HotSpot使用永久代来实现方法区时，实现是完全符合这种逻辑概念的；而在JDK 8及之后，类变量则会随着Class对象一起存放在Java堆中，这时候“类变量在方法区”就完全是一种对逻辑概念的表述了。

关于准备阶段，还有两个容易产生混淆的概念需要着重强调，首先是**这时候进行内存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在Java堆中。其次是这里所说的初始值**“通常情况”**下是数据类型的零值，假设一个类变量的定义为：

```
public static int value = 123;
```

那变量value在**准备阶段**过后的**初始值为0**而不是123，因为这时尚未开始执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把**value赋值为123**的动作**要到类的初始化阶段**才会被执行。

上面提到在**“通常情况”**下初始值是零值，那言外之意是相对的会有某些“特殊情况”：如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量值就会被初始化为ConstantValue属性所指定的初始值。final static String

16. 类加载器链接的解析阶段做了什么

解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程。

·**符号引用**（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。
·**直接引用**（Direct References）：直接引用是可以直接指向目标的指针、相对偏移量或者是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在虚拟机的内存中存在。

《Java虚拟机规范》之中并未规定解析阶段发生的具体时间，虚拟机实现可以根据需要来自行判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。。对同一个符号引用进行多次解析请求是很常见的事情，除invokedynamic指令以外，虚拟机实现可以对第一次解析的结果进行缓存，譬如在运行时直接引用常量池中的记录，并把常量标识为已解析状态，从而避免解析动作重复进行。

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符这7类符号引用进行，分别对应于常量池的CONSTANT_Class_info、CON-STANT_Fieldref_info、CONSTANT_Methodref_info、CONSTANT_InterfaceMethodref_info、CONSTANT_MethodType_info、CONSTANT_MethodHandle_info、CONSTANT_Dyna-mic_info和CONSTANT_InvokeDynamic_info 8种常量类型[2]。

17. 类加载初始化阶段做了什么

类的初始化阶段是类加载过程的最后一个步骤，之前介绍的几个类加载的动作里，除了在加载阶段用户应用程序可以通过自定义类加载器的方式局部参与外，其余动作都完全由Java虚拟机来主导控制。直到初始化阶段，Java虚拟机才真正开始执行类中编写的Java程序代码，将主导权移交给应用程序。

进行准备阶段时，变量已经赋过一次系统要求的初始零值，而在初始化阶段，则会根据程序员通过程序编码制定的主观计划去初始化类变量和其他资源。我们也可以从另外一种更直接的形式来表达：初始化阶段就**是执行类构造器<clinit>()方法的过程**。<clinit>()并不是程序员在Java代码中直接编写的方法，它是Javac编译器的自动生成物。

18. 说说你对clinit方法的理解

·<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{}块）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决定的，静态语句块中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。

```
public class Test {
    static {
        i = 0;  //  给变量复制可以正常编译通过
        System.out.print(i);  // 这句编译器会提示“非法向前引用”
    }
    static int i = 1;
```

·<clinit>()方法与类的构造函数（即在虚拟机视角中的实例构造器<init>()方法）不同，它不需要显式地调用父类构造器，Java虚拟机会保证在子类的<clinit>()方法执行前，父类的<clinit>()方法已经执行完毕。因此在Java虚拟机中第一个被执行的<clinit>()方法的类型肯定是java.lang.Object。

```
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}
static class Sub extends Parent {
    public static int B = A;
}
public static void main(String[] args) {
    System.out.println(Sub.B);
}
```

2

·<clinit>()方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()

·接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法，因为只有当父接口中定义的变量被使用时，父接口才会被初始化。此外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

·Java虚拟机必须保证一个类的<clinit>()方法在多线程环境中被正确地加锁同步，如果多个线程同时去初始化一个类，那么只会有其中一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行完毕<clinit>()方法。**需要注意，其他线程虽然会被阻塞，但如果执行＜clinit＞()方法的那条线程退出＜clinit＞()方法后，其他线程唤醒后则不会再次进入＜clinit＞()方法。同一个类加载器下，一个类型只会被初始化一次。如果在一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞[2]，在实际应用中这种阻塞往往是很隐蔽的。**

19. 什么情况下JVM会立即对类进行初始化操作

对于初始化阶段，《Java虚拟机规范》则是严格规定了有且只有六种情况必须立即对类进行“初始化”。

1）遇到new、getstatic、putstatic或invokestatic这四条**字节码指令**时，如果类型没有进行过初始化，则需要先触发其初始化阶段。能够生成这四条指令的典型Java代码场景有：

·使用new关键字实例化对象的时候。

·读取或设置一个类型的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候。

·调用一个类型的静态方法的时候。

2）使用java.lang.reflect包的方法对类型进行反射调用的时候，如果类型没有进行过初始化，则需要先触发其初始化。

3）当初始化类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

5）当使用JDK 7新加入的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果为REF_getStatic、REF_putStatic、REF_invokeStatic、REF_newInvokeSpecial四种类型的方法句柄，并且这个方法句柄对应的类没有进行过初始化，则需要先触发其初始化。

6）当一个接口中定义了JDK 8新加入的默认方法（被default关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。

```
package org.fenixsoft.classloading;
/**
 * 被动使用类字段演示一：
 * 通过子类引用父类的静态字段，不会导致子类初始化
 **/
public class SuperClass {
    static {
        System.out.println("SuperClass init!");
    }
    public static int value = 123;
}
public class SubClass extends SuperClass {
    static {
        System.out.println("SubClass init!");
    }
}
/**
 * 非主动使用类字段演示
 **/
public class NotInitialization {
    public static void main(String[] args) {
        System.out.println(SubClass.value);
    }
}
```

上述代码运行之后，只会输出“SuperClass init！”对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

```
package org.fenixsoft.classloading;
/**
 * 被动使用类字段演示二：
 * 通过数组定义来引用类，不会触发此类的初始化
 **/
public class NotInitialization {
    public static void main(String[] args) {
        SuperClass[] sca = new SuperClass[10];
    }
}
```

通过数组定义来引用类，不会触发此类的初始化。

```
package org.fenixsoft.classloading;
/**
 * 被动使用类字段演示三：
 * 常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量的
   类的初始化
 **/
public class ConstClass {
    static {
        System.out.println("ConstClass init!");
    }
    public static final String HELLOWORLD = "hello world";
}
/**
 * 非主动使用类字段演示
 **/
public class NotInitialization {
    public static void main(String[] args) {
        System.out.println(ConstClass.HELLOWORLD);
    }
}
```

上述代码运行之后，也没有输出“ConstClass init！”

参看面试题 180中的ConstantValue属性。

HELLOWORLD 在javac的编译阶段就已经存入了字段表中的属性表的ConstantValue属性中。

20. 不同的类加载器对instanceof关键字运算的结果会有影响吗

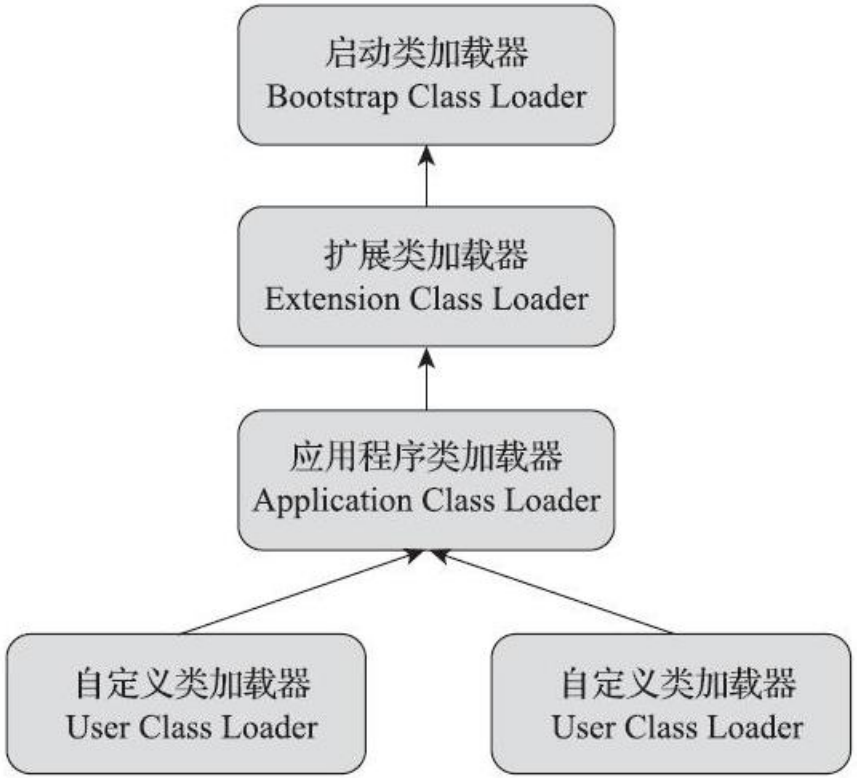
```
public class ClassLoaderTest {
    public static void main(String[] args) throws Exception {
        ClassLoader myLoader = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name) throws ClassNotFoundException {
                try {
                    String fileName = name.substring(name.lastIndexOf(".") + 1)+".class";
                    InputStream is = getClass().getResourceAsStream(fileName);
                    if (is == null) {
                        return super.loadClass(name);
                    }
                    byte[] b = new byte[is.available()];
                    is.read(b);
                    return defineClass(name, b, 0, b.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException(name);
                }
            }
        };
        Object obj = myLoader.loadClass("org.fenixsoft.classloading.ClassLoaderTest").newInstance();
        System.out.println(obj.getClass());
        System.out.println(obj instanceof org.fenixsoft.classloading.ClassLoaderTest);
    }
}
```

运行结果：
class org.fenixsoft.classloading.ClassLoaderTest
false

两行输出结果中，从第一行可以看到这个对象确实是类org.fenixsoft.classloading.ClassLoaderTest实例化出来的，但在第二行的输出中却发现这个对象与类org.fenixsoft.classloading.ClassLoaderTest做所属类型检查的时候返回了false。这是因为Java虚拟机中同时存在了两个ClassLoaderTest类，一个是由虚拟机的应用程序类加载器所加载的，另外一个是由我们自定义的类加载器加载的，虽然它们都来自同一个Class文件，但在Java虚拟机中仍然是两个互相独立的类，做对象所属类型检查时的结果自然为false。

对于任意一个类，都必须由加载它的类加载器和这个类本身一起共同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个Java虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

21. 说下双亲委派模型



·启动类加载器（Bootstrap Class Loader）：这个类加载器负责加载存放在<JAVA_HOME>\lib目录，或者被-Xbootclasspath参数所指定的路径中存放的，而且是Java虚拟机能够识别的（按照文件名识别，如rt.jar、tools.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机的内存中。启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器去处理，那直接使用null代替即可。

·扩展类加载器（Extension Class Loader）：这个类加载器是在类sun.misc.Launcher$ExtClassLoader中以Java代码的形式实现的。它负责加载<JAVA_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量所指定的路径中所有的类库。

·应用程序类加载器（Application Class Loader）：这个类加载器由sun.misc.Launcher$AppClassLoader来实现。由于应用程序类加载器是ClassLoader类中的getSystem-ClassLoader()方法的返回值，所以有些场合中也称它为“系统类加载器”。它负责加载用户类路径（ClassPath）上所有的类库，开发者同样可以直接在代码中使用这个类加载器。如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

双亲委派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有自己的父类加载器。不过这里类加载器之间的父子关系一般不是以继承（Inheritance）的关系来实现的，而是通常使用组合（Composition）关系来复用父加载器的代码。

双亲委派模型的工作过程是：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去完成，每一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最顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去完成加载。

使用双亲委派模型来组织类加载器之间的关系，一个显而易见的好处就是Java中的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，它存放在rt.jar之中，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启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都能够保证是同一个类。（**翻看上一题 190, 想要确定一个加载一个类的唯一性，是需要由类加载器和类共同验证才能生效的。**）

```
protected synchronized Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException
{
    // 首先，检查请求的类是否已经被加载过了
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
        if (parent != null) {
            c = parent.loadClass(name, false);
        } else {
            c = findBootstrapClassOrNull(name);
        }
        } catch (ClassNotFoundException e) {
            // 如果父类加载器抛出ClassNotFoundException
            // 说明父类加载器无法完成加载请求
        }
        if (c == null) {
            // 在父类加载器无法加载时
            // 再调用本身的findClass方法来进行类加载
            c = findClass(name);
        }
    }
    if (resolve) {
        resolveClass(c);
    }
    return c;
}
```

22. 说说运行时栈帧的结构

Java虚拟机以方法作为最基本的执行单元，“栈帧”（Stack Frame）则是用于支持虚拟机进行方法调用和方法执行背后的数据结构，它也是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始至执行结束的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

在编译Java程序源码的时候，栈帧中需要多大的局部变量表，需要多深的操作数栈就已经被分析计算出来，并且写入到方法表的Code属性之中。换言之，一个栈帧需要分配多少内存，并不会受到程序运行期变量数据的影响，而仅仅取决于程序源码和具体的虚拟机实现的栈内存布局形式。

对于执行引擎来讲，在活动线程中，只有位于栈顶的方法才是在运行的，只有位于栈顶的栈帧才是生效的，其被称为“当前栈帧”（Current Stack Frame），与这个栈帧所关联的方法被称为“当前方法”（Current Method）。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。

23. 说说你对栈帧中的局部变量表的理解以及变量的存储

局部变量表（Local Variables Table）是一组变量值的存储空间，用于存放**方法参数**和**方法内部定义的局部变量**。在Java程序被编译为Class文件时，就在方法的Code属性的max_locals数据项中确定了该方法所需分配的局部变量表的最大容量。

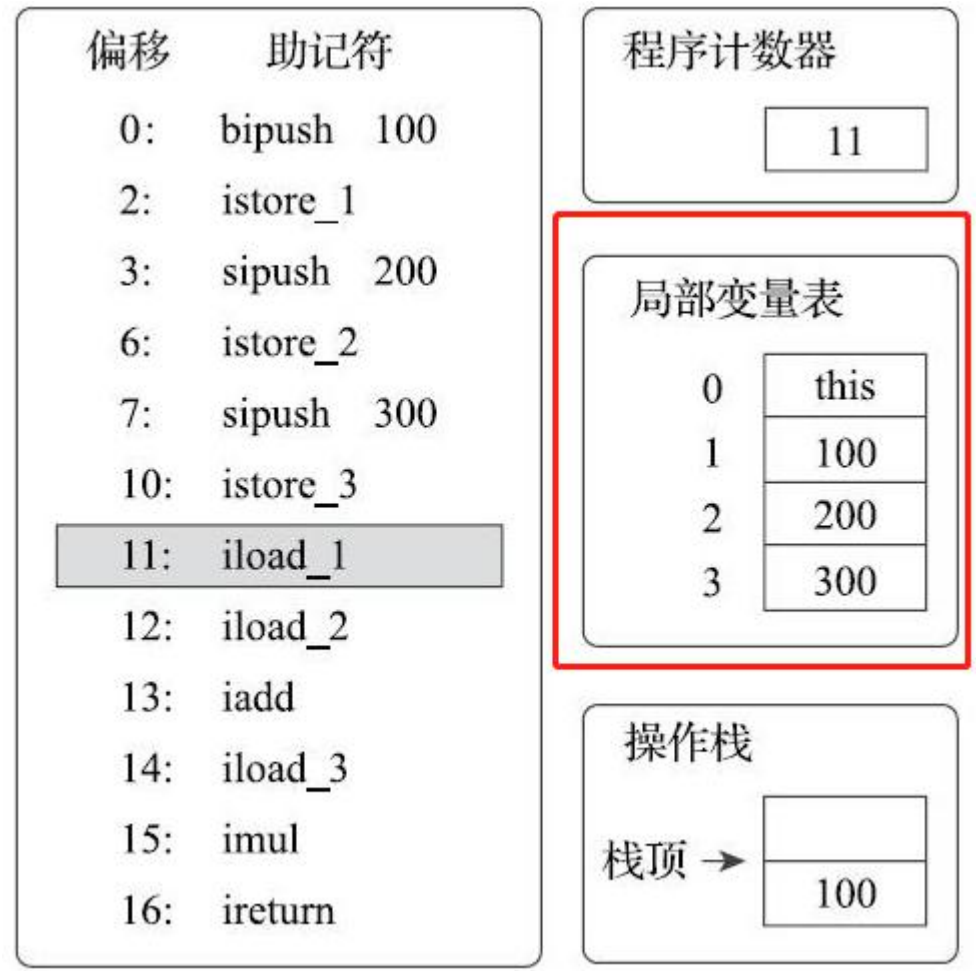
局部变量表的容量以变量槽（Variable Slot）为最小单位，《Java虚拟机规范》中并没有明确指出一个变量槽应占用的内存空间大小，只是很有导向性地说到每个变量槽都应该能存放一个boolean、byte、char、short、int、float、reference或returnAddress类型的数据，这8种数据类型，都可以使用32位或更小的物理内存来存储，但这种描述与明确指出“每个变量槽应占用32位长度的内存空间”是有本质差别的，它允许变量槽的长度可以随着处理器、操作系统或虚拟机实现的不同而发生变化。

对于64位的数据类型，Java虚拟机会以高位对齐的方式为其分配两个连续的变量槽空间。Java语言中明确的64位的数据类型只有long和double两种。由于局部变量表是建立在线程堆栈中的，属于线程私有的数据，无论读写两个连续的变量槽是否为原子操作，都不会引起数据竞争和线程安全问题。

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

使用javap命令看看它的字节码指令

```
public int calc();
  Code:
    Stack=2, Locals=4, Args_size=1
     0:  bipush  100
     2:  istore_1
     3:  sipush  200
     6:  istore_2
     7:  sipush  300
    10:  istore_3
    11:  iload_1
    12:  iload_2
    13:  iadd
    14:  iload_3
    15:  imul
    16:  ireturn
}
```



istore指令，_1代表存入变量槽1.是将局部变量存入局部变量表。

24. 栈帧的局部变量表是如何定位变量的？是如何完成实参到形参传递的?

Java虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从0开始至局部变量表最大的变量槽数量。如果访问的是32位数据类型的变量，索引N就代表了使用第N个变量槽，如果访问的是64位数据类型的变量，则说明会同时使用第N和N+1两个变量槽。对于两个相邻的共同存放一个64位数据的两个变量槽，虚拟机不允许采用任何方式单独访问其中的某一个，《Java虚拟机规范》中明确要求了如果遇到进行这种操作的字节码序列，虚拟机就应该在类加载的校验阶段中抛出异常。

当一个方法被调用时，Java虚拟机会使用局部变量表来完成参数值到参数变量列表的传递过程，即实参到形参的传递。如果执行的是实例方法（没有被static修饰的方法），那局部变量表中第0位索引的变量槽默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从1开始的局部变量槽，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的变量槽。

25. 说说栈帧中的操作数栈

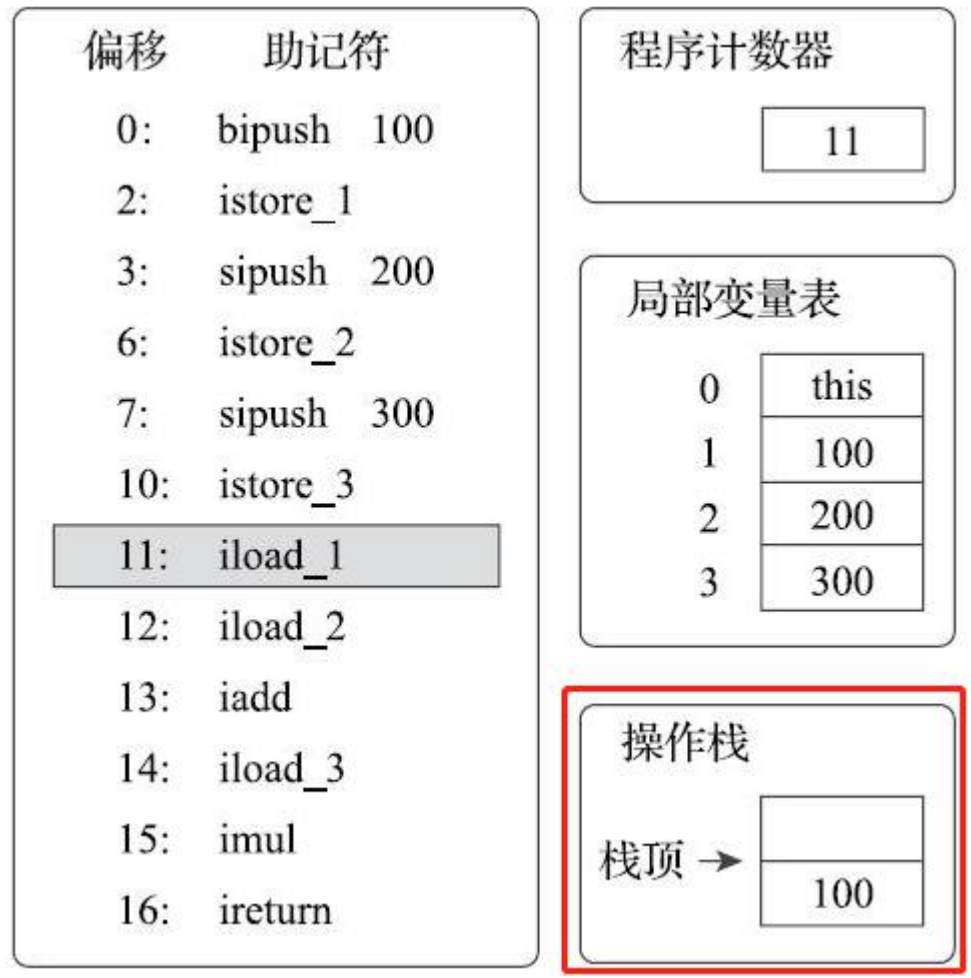
操作数栈（Operand Stack）也常被称为操作栈，它是一个后入先出（Last In First Out，LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候被写入到Code属性的max_stacks数据项之中。操作数栈的每一个元素都可以是包括long和double在内的任意Java数据类型。32位数据类型所占的栈容量为1，64位数据类型所占的栈容量为2。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈和入栈操作。譬如在做算术运算的时候是通过将运算涉及的操作数栈压入栈顶后调用运算指令来进行的，又譬如在调用其他方法的时候是通过操作数栈来进行方法参数的传递。举个例子，例如整数加法的字节码指令iadd，这条指令在运行的时候要求操作数栈中最接近栈顶的两个元素已经存入了两个int型的数值，当执行这个指令时，会把这两个int值出栈并相加，然后将相加的结果重新入栈。

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

使用javap命令看看它的字节码指令

```
public int calc();
  Code:
    Stack=2, Locals=4, Args_size=1
     0:  bipush  100
     2:  istore_1
     3:  sipush  200
     6:  istore_2
     7:  sipush  300
    10:  istore_3
    11:  iload_1
    12:  iload_2
    13:  iadd
    14:  iload_3
    15:  imul
    16:  ireturn
}
```



iload指令，_1代表变量槽1. 是将局部变量从变量表中复制到操作数栈栈顶。

26. 一个线程终两个不同的栈帧是否会有存储空间重叠的部分。

两个不同栈帧作为不同方法的虚拟机栈的元素，是完全相互独立的。但是在大多虚拟机的实现里都会进行一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样做不仅节约了一些空间，更重要的是在进行方法调用时就可以直接共用一部分数据，无须进行额外的参数复制传递了

27. 栈帧中的动态链接和返回地址有什么作用

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（Dynamic Linking）。我们知道Class文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池里指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就被转化为直接引用，这种转化被称为静态解析。另外一部分将在**每一次运行期间都转化为直接引用，这部分就称为动态连接**。

当一个方法开始执行后，只有两种方式退出这个方法。一个是正常退出，另外一个是异常退出。无论采用何种退出方式，在方法退出之后，都必须返回到最初方法被调用时的位置，程序才能继续执行，方法返回时**可能需要**在栈帧中保存一些信息，用来帮助恢复它的上层主调方法的执行状态。一般来说，方法正常退出时，主调方法的PC计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中就一般不会保存这部分信息。方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整PC计数器的值以指向方法调用指令后面的一条指令等。

这里写的“可能”是由于这是基于概念模型的讨论，只有具体到某一款Java虚拟机实现，会执行哪些操作才能确定下来。

28. JVM中 方法调用 的两种形式

一种形式是解析；另外一种形式是分派。

所有方法调用的目标方法在Class文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能够成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译那一刻就已经确定下来。这类方法的调用被称为**解析（Resolution）。**

调用不同类型的方法，字节码指令集里设计了不同的指令。在Java虚拟机支持以下5条方法调用字节码指令：
·invokestatic。用于调用静态方法。
·invokespecial。用于调用实例构造器<init>()方法、私有方法和父类中的方法。
·invokevirtual。用于调用所有的虚方法。
·invokeinterface。用于调用接口方法，会在运行时再确定一个实现该接口的对象。
·invokedynamic。先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段中确定唯一的调用版本，Java语言里符合这个条件的方法共有静态方法、私有方法、实例构造器、父类方法4种，再加上被final修饰的方法（尽管它使用invokevirtual指令调用），这5种方法调用会在类加载的时候就可以把符号引用解析为该方法的直接引用。这些方法统称为“非虚方法”（Non-Virtual Method），与之相反，其他方法就被称为“虚方法”（Virtual Method）。

分派又分为静态分派，动态分派。

所有依赖静态类型来决定方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用表现就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的，这点也是为何一些资料选择把它归入“解析”而不是“分派”的原因。

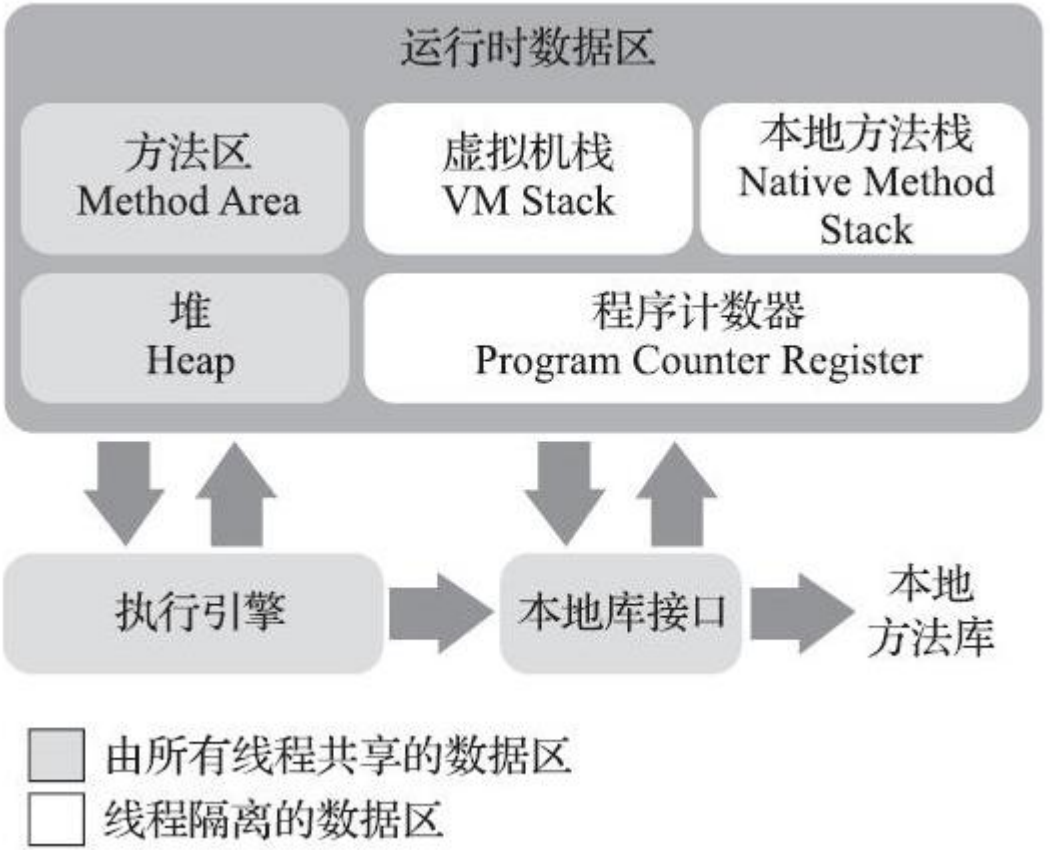
动态分派的实现过程，它与Java语言多态性的另外一个重要体现——重写（Override）有着很密切的关联。
动态分派其实就是动态定位到实现类的方法进行调用。

29. 说说你对 java.lang.invoke包的理解，与反射调用的区别

JDK 7时新加入的java.lang.invoke包，这个包的主要目的是在之前单纯依靠符号引用来确定调用的目标方法这条路之外，提供一种新的动态确定目标方法的机制，称为“方法句柄”（Method Handle）。

·Reflection和MethodHandle机制本质上都是在模拟方法调用，但是Reflection是在模拟Java代码层次的方法调用，而MethodHandle是在模拟字节码层次的方法调用。在MethodHandles.Lookup上的3个方法findStatic()、findVirtual()、findSpecial()正是为了对应于invokestatic、invokevirtual（以及invokeinterface）和invokespecial这几条字节码指令的执行权限校验行为，而这些底层细节在使用Reflection API时是不需要关心的。
·Reflection中的java.lang.reflect.Method对象远比MethodHandle机制中的java.lang.invoke.MethodHandle对象所包含的信息来得多。前者是方法在Java端的全面映像，包含了方法的签名、描述符以及方法属性表中各种属性的Java端表示方式，还包含执行权限等的运行期信息。而后者仅包含执行该方法的相关信息。用开发人员通俗的话来讲，Reflection是重量级，而MethodHandle是轻量级。

30. 运行时数据区包括什么



31. 说说你对程序计数器的理解

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在Java虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换、分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个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**；如果正在执行的是本地（Native）方法，这个计数器值则应为空（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。

32. 方法区主要是存储什么数据的

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

说到方法区，不得不提一下“永久代”这个概念，尤其是在JDK 8以前，许多Java程序员都习惯在HotSpot虚拟机上开发、部署程序，很多人都更愿意把方法区称呼为“永久代”（Permanent Generation），或将两者混为一谈。本质上这两者并不是等价的，因为仅仅是当时的HotSpot虚拟机设计团队选择把收集器的分代设计扩展至方法区，或者说使用永久代来实现方法区而已，这样使得HotSpot的垃圾收集器能够像管理Java堆一样管理这部分内存，省去专门为方法区编写内存管理代码的工作。

到了JDK 8，终于完全废弃了永久代的概念，改用与JRockit、J9一样在本地内存中实现的元空间（Meta-space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

《Java虚拟机规范》对方法区的约束是非常宽松的，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，甚至还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域的确是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这区域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，一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收有时又确实是必要的。

33. 运行时常量池与 Class文件结构的常量池有什么区别和联系

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表（Constant Pool Table），用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是说，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。
既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OOM异常。

34. 什么是直接内存？是运行时数据区的一部分吗

直接内存（Direct Memory）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。

在JDK 1.4中新加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

35. 说说下JVM堆内存中对象的内存布局

在HotSpot虚拟机里，对象在堆内存中的存储布局可以划分为三个部分：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot虚拟机对象的对象头部分包括两类信息。第一类是用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32个比特和64个比特，官方称它为“Mark Word”。对象需要存储的运行时数据很多，其实已经超出了32、64位Bitmap结构所能记录的最大限度，但对象头里的信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个有着动态定义的数据结构，以便在极小的空间内存储尽量多的数据，根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机中，如对象未被同步锁锁定的状态下，Mark Word的32个比特存储空间中的25个比特用于存储对象哈希码，4个比特用于存储对象分代年龄，2个比特用于存储锁标志位，1个比特固定为0。对象头的另外一部分是类型指针，即对象指向它的类型元数据的指针，Java虚拟机通过这个指针来确定该对象是哪个类的实例。

接下来实例数据部分是对象真正存储的有效信息，即我们在程序代码里面所定义的各种类型的字段内容，无论是从父类继承下来的，还是在子类中定义的字段都必须记录起来。这部分的存储顺序会受到虚拟机分配策略参数（-XX：FieldsAllocationStyle参数）和字段在Java源码中定义顺序的影响。HotSpot虚拟机默认的分配顺序为longs/doubles、ints、shorts/chars、bytes/booleans、oops（Ordinary Object Pointers，OOPs），从以上默认的分配策略中可以看到，相同宽度的字段总是被分配到一起存放，在满足这个前提条件的情况下，在父类中定义的变量会出现在子类之前。如果HotSpot虚拟机的+XX：CompactFields参数值为true（默认就为true），那子类之中较窄的变量也允许插入父类变量的空隙之中，以节省出一点点空间。

对象的第三部分是对齐填充，这并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说就是任何对象的大小都必须是8字节的整数倍。对象头部分已经被精心设计成正好是8字节的倍数（1倍或者2倍），因此，如果对象实例数据部分没有对齐的话，就需要通过对齐填充来补全。

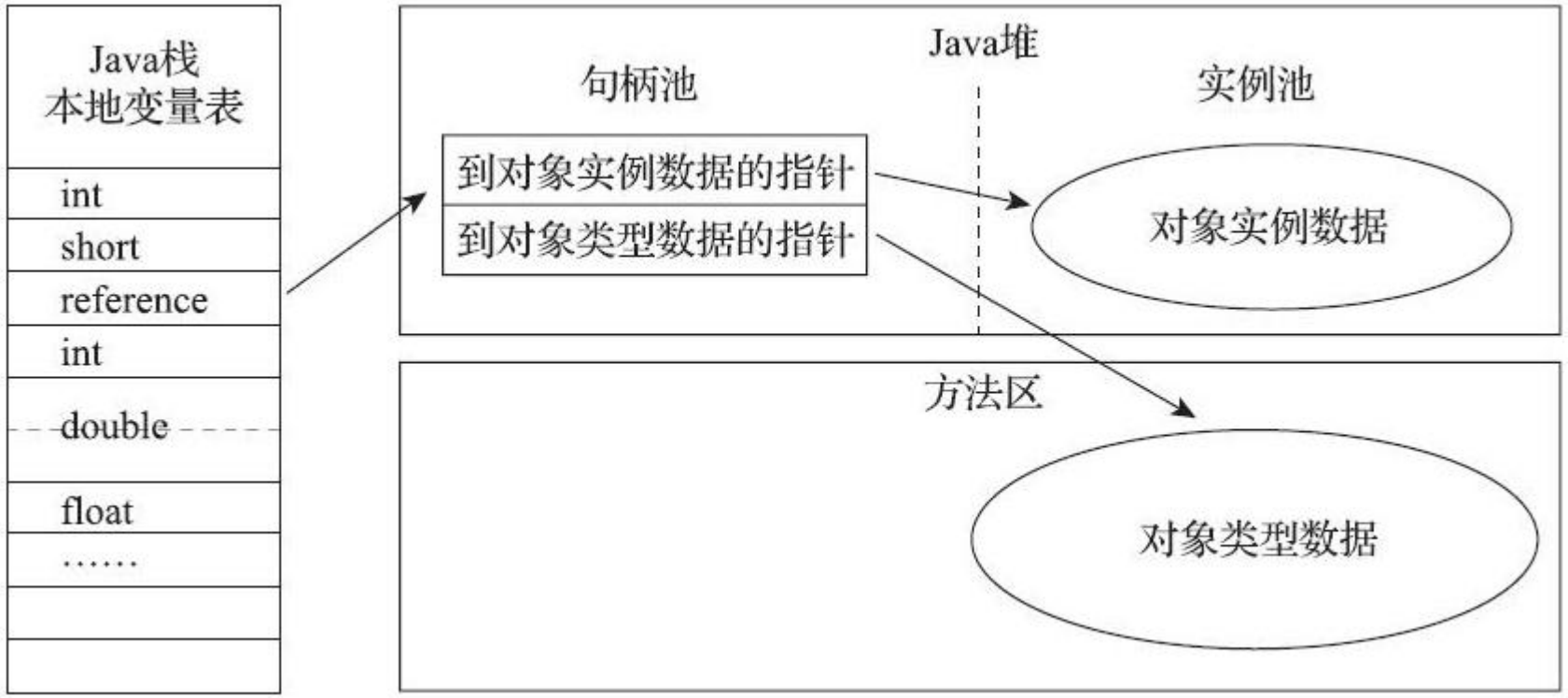
36. 64位Jvm，new Object（）新创建的对象在Java中占用多少内存

markword 8 字节，因为 java 默认使用了 calssPointer 压缩，classpointer 4 字节，对象实例0字节，padding 4 字节 因此是 16 字节 如果没开启 classpointer 默认压缩，markword 8 字节，classpointer 8 字节，对象实例0字节，padding 0 字节 也是 16 字节。

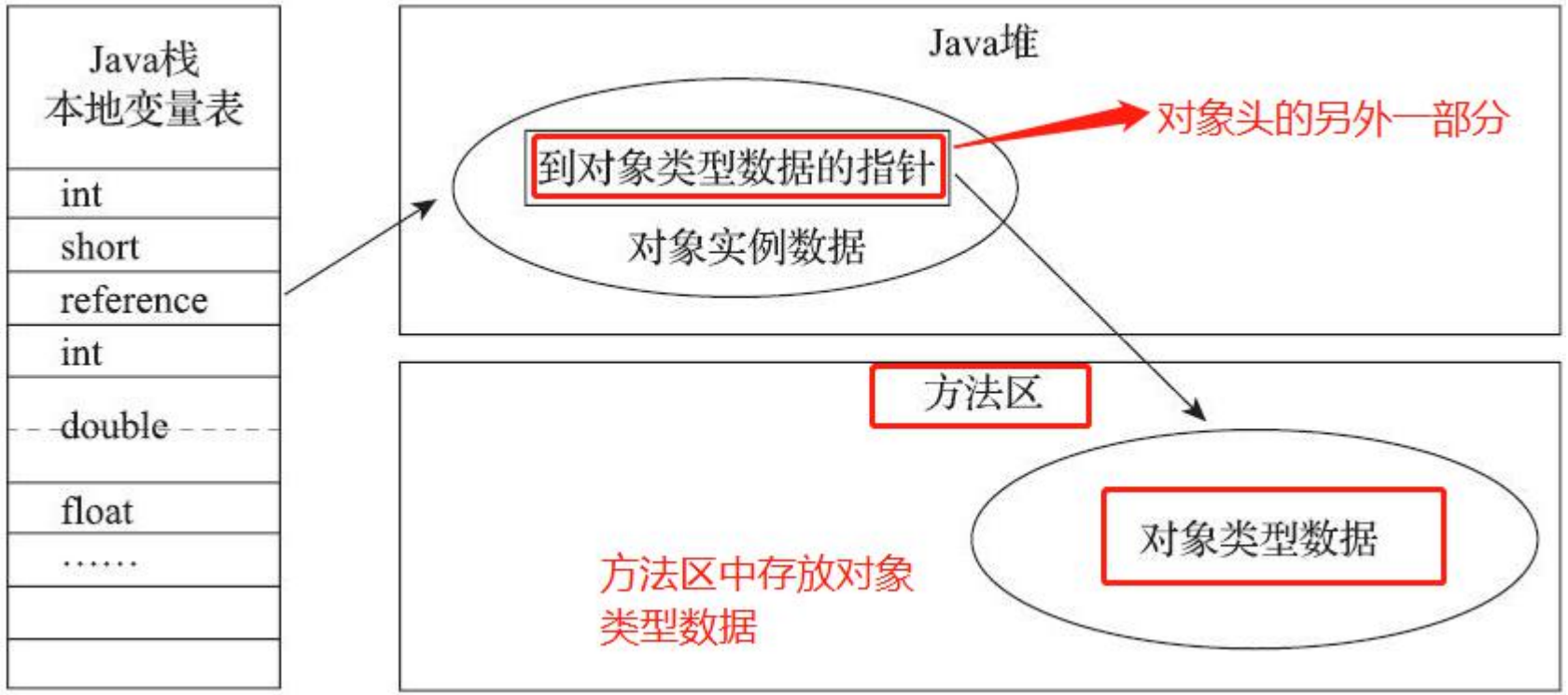
37.对象是如何被线程访问定位的?

创建对象自然是为了后续使用该对象，我们的Java程序会通过栈上的reference数据来操作堆上的具体对象。由于reference类型在《Java虚拟机规范》里面只规定了它是一个指向对象的引用，并没有定义这个引用应该通过什么方式去定位、访问到堆中对象的具体位置，所以对象访问方式也是由虚拟机实现而定的，主流的访问方式主要有使用句柄和直接指针两种。

如果使用**句柄访问**的话，Java堆中将可能会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自具体的地址信息



如果使用直接指针访问的话，Java堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference中存储的直接就是对象地址，如果只是访问对象本身的话，就不需要多一次间接访问的开销



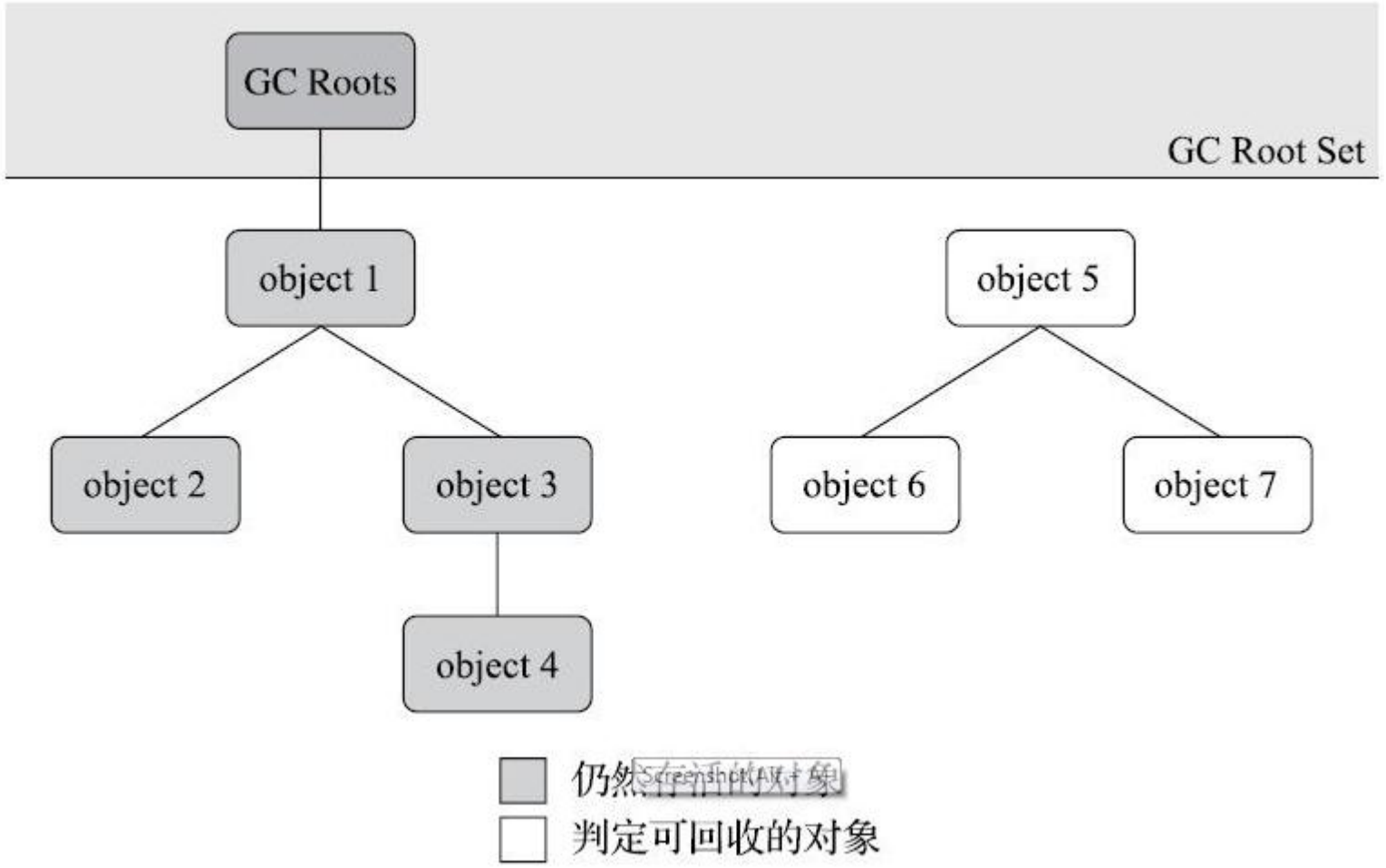
这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要被修改。

使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问在Java中非常频繁，因此这类开销积少成多也是一项极为可观的执行成本，就虚拟机HotSpot而言，它主要使用第二种方式进行对象访问。

38. JVM 是如何判断对象是否存活的

可达性分析算法

这个算法的基本思路就是通过一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”（Reference Chain），如果某个对象到GC Roots间没有任何引用链相连，或者用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的。



39. GCRoots 是什么?

在Java技术体系里面，固定可作为GC Roots的对象包括以下几种:
·在虚拟机**栈**（栈帧中的本地变量表）**中引用的对象**，譬如各个线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等。

·在**方法区中类静态属性引用的对象**，譬如Java类的引用类型静态变量。

·在**方法区中常量引用的对象**，譬如字符串常量池（String Table）里的引用。

·在**本地方法栈中JNI**（即通常所说的Native方法）**引用的对象**。

·**Java虚拟机内部的引用**，如基本数据类型对应的Class对象，一些常驻的异常对象（比如NullPointExcepiton、OutOfMemoryError）等，还有系统类加载器。

·所有被同步**锁**（synchronized关键字）**持有的对象**。

·反映Java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等

40. Java中对象的引用类型有什么？垃圾回收下的表现是什么样的

在JDK 1.2版之后，Java对引用的概念进行了扩充，将引用分为**强引用**（Strongly Re-ference）、**软引用**（Soft Reference）、**弱引用**（Weak Reference）和**虚引用**（Phantom Reference）4种，这4种引用强度依次逐渐减弱。

·**强引用**是最传统的“引用”的定义，是指在程序代码之中普遍存在的引用赋值，即类似“Object obj=new Object()”这种引用关系。无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。
·**软引用**是用来描述一些还有用，但非必须的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK 1.2版之后提供了SoftReference类来实现软引用。
·**弱引用**也是用来描述那些非必须对象，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。当垃圾收集器开始工作，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK 1.2版之后提供了WeakReference类来实现弱引用。
·**虚引用**也称为“幽灵引用”或者“幻影引用”，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的只是为了能在这个对象被收集器回收时收到一个系统通知。在JDK 1.2版之后提供了PhantomReference类来实现虚引用。

41. 如果对象不可达，Jvm会立即回收这个对象吗

即使在可达性分析算法中判定为不可达的对象，也不是“非死不可”的，这时候它们暂时还处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记，随后进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。假如对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，那么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确有必要执行finalize()方法，那么该对象将会被放置在一个名为F-Queue的队列之中，并在稍后由一条由虚拟机自动建立的、低调度优先级的Finalizer线程去执行它们的finalize()方法。

42. 方法区有垃圾回收吗

《Java虚拟机规范》中提到过可以不要求虚拟机在方法区中实现垃圾收集，方法区垃圾收集的“性价比”通常也是比较低的：在Java堆中，尤其是在新生代中，对常规应用进行一次垃圾收集通常可以回收70%至99%的内存空间，相比之下，方法区回收囿于苛刻的判定条件，其区域垃圾收集的回收成果往往远低于此。

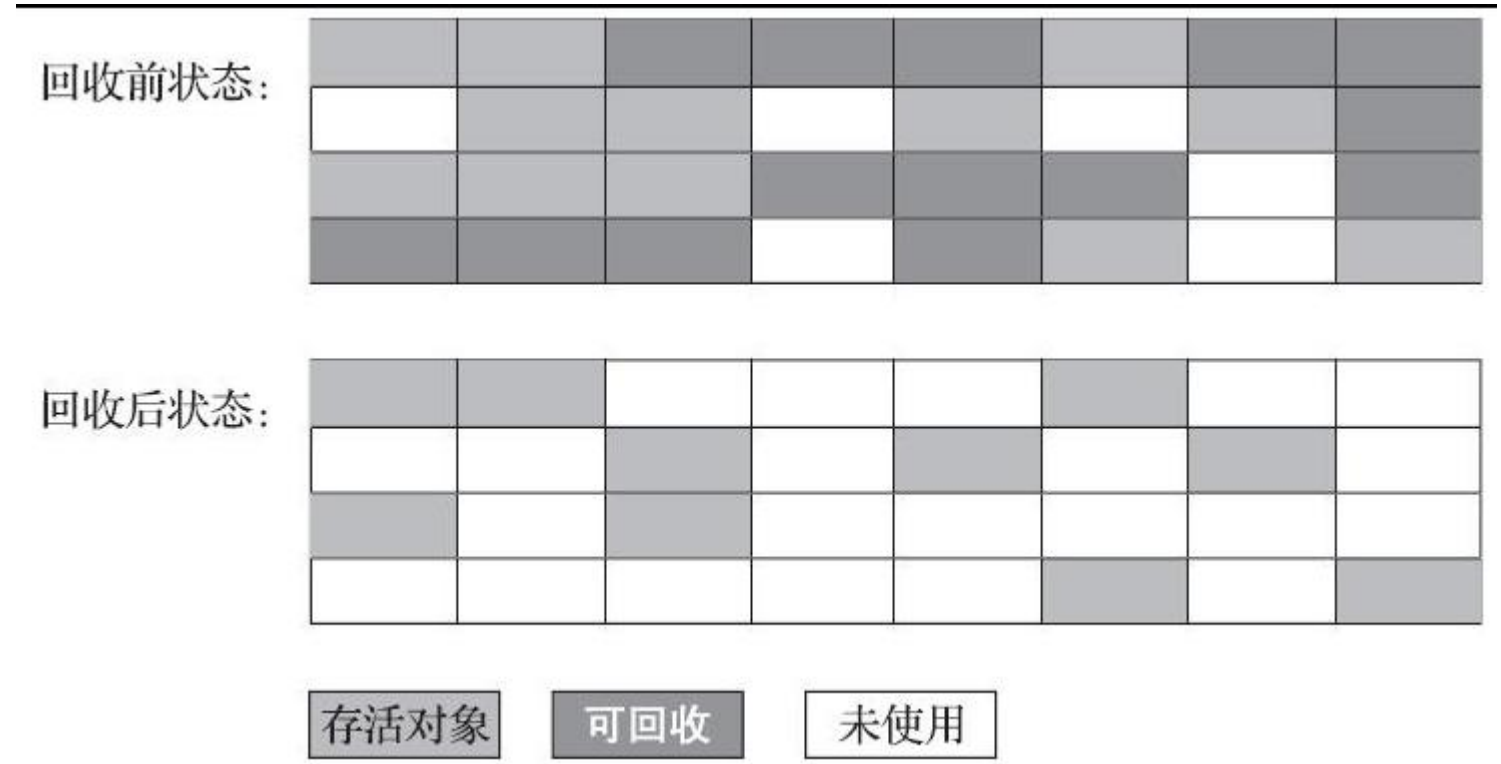
方法区的垃圾收集主要回收两部分内容：废弃的常量和不再使用的类型。

43. 垃圾回收的算法有几种，分别说说利弊

3种。

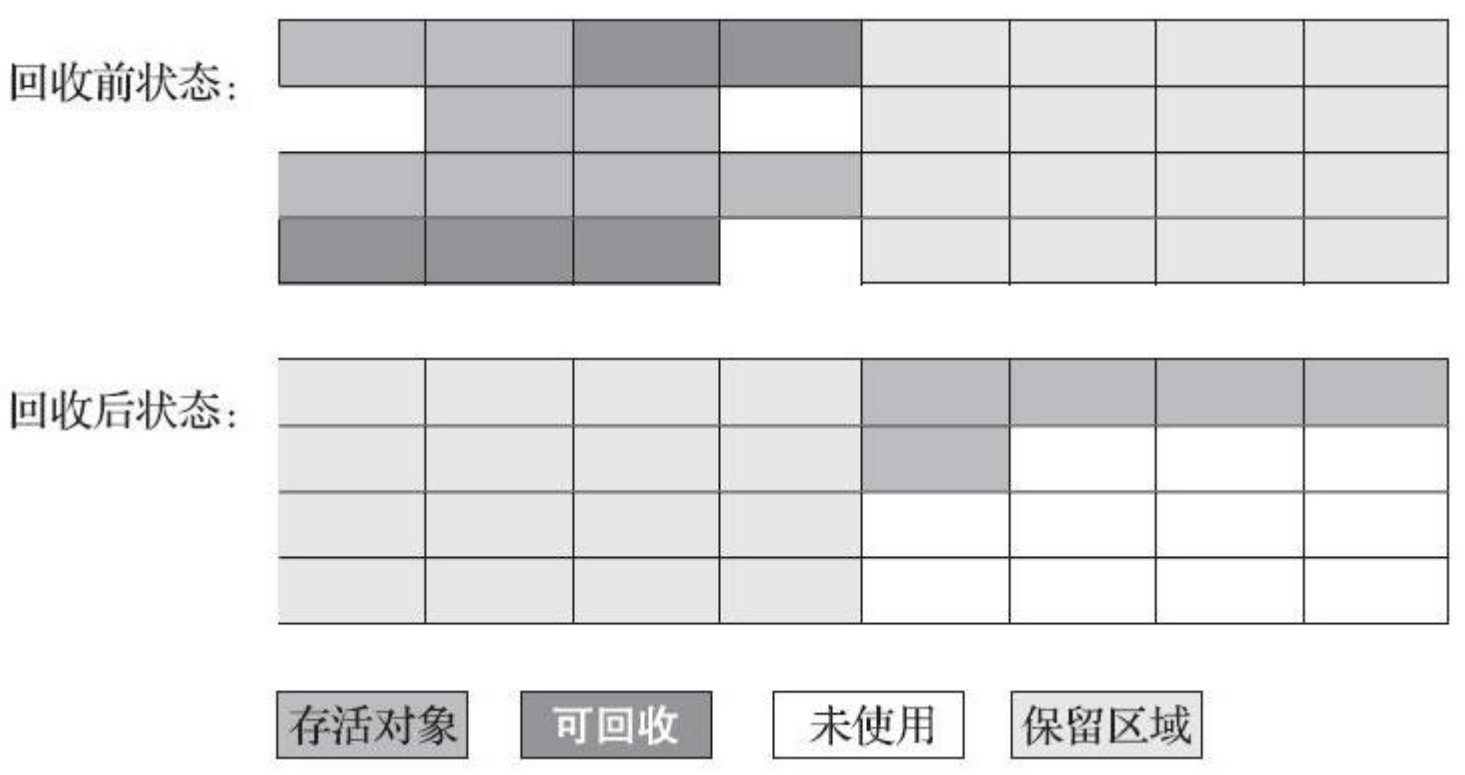
标记-清除算法

算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象，也可以反过来，标记存活的对象，统一回收所有未被标记的对象。
最基础的收集算法，是因为后续的收集算法大多都是以标记-清除算法为基础，对其缺点进行改进而得到的。它的主要缺点有两个：第一个是执行效率不稳定，如果Java堆中包含大量对象，而且其中大部分是需要被回收的，这时必须进行大量标记和清除的动作，导致标记和清除两个过程的执行效率都随对象数量增长而降低；第二个是内存空间的碎片化问题，标记、清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致当以后在程序运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作



标记-复制算法

常被简称为复制算法。为了解决标记-清除算法面对大量可回收对象时执行效率低的问题，1969年Fenichel提出了一种称为“半区复制”（Semispace Copying）的垃圾收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。如果内存中多数对象都是存活的，这种算法将会产生大量的内存间复制的开销，但对于多数对象都是可回收的情况，算法需要复制的就是占少数的存活对象，而且每次都是针对整个半区进行内存回收，分配内存时也就不用考虑有空间碎片的复杂情况，只要移动堆顶指针，按顺序分配即可。这样实现简单，运行高效，不过其缺陷也显而易见，这种复制回收算法的代价是将可用内存缩小为了原来的一半，空间浪费未免太多了一点。



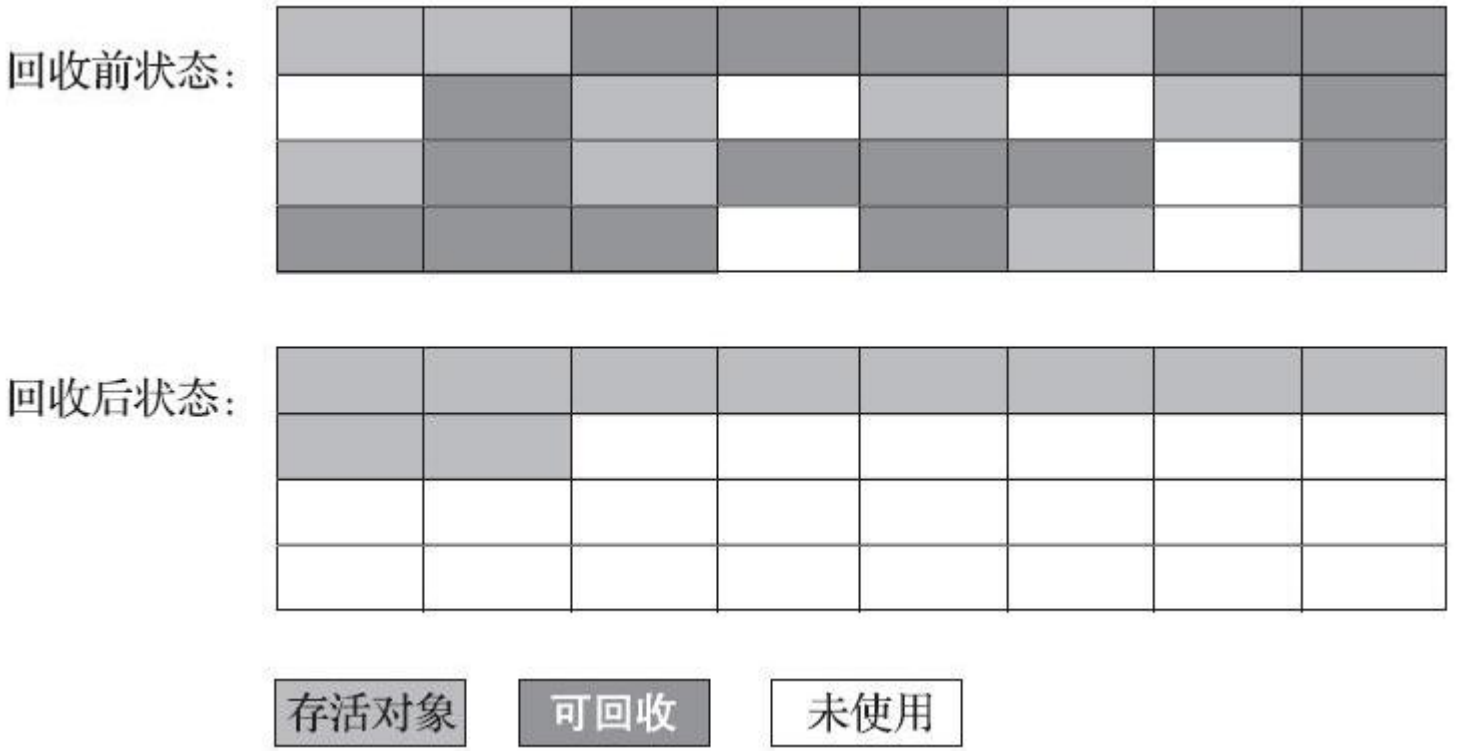
现在的商用Java虚拟机大多都优先采用了这种收集算法去回收新生代，IBM公司曾有一项专门研究对新生代“朝生夕灭”的特点做了更量化的诠释——新生代中的对象有98%熬不过第一轮收集。因此并不需要按照1：1的比例来划分新生代的内存空间。

在1989年，Andrew Appel针对具备“朝生夕灭”特点的对象，提出了一种更优化的半区复制分代策略，现在称为“Appel式回收”。HotSpot虚拟机的Serial、ParNew等新生代收集器均采用了这种策略来设计新生代的内存布局。Appel式回收的具体做法是把新生代分为一块较大的Eden空间和两块较小的Survivor空间，每次分配内存只使用Eden和其中一块Survivor。发生垃圾搜集时，将Eden和Survivor中仍然存活的对象一次性复制到另外一块Survivor空间上，然后直接清理掉Eden和已用过的那块Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8：1

标记-整理算法

标记-复制算法在对象存活率较高时就要进行较多的复制操作，效率将会降低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

针对老年代对象的存亡特征，1974年Edward Lueders提出了另外一种有针对性的“标记-整理”（Mark-Compact）算法，其中的标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存



标记-清除算法与标记-整理算法的本质差异在于前者是一种非移动式的回收算法，而后者是移动式的。是否移动对象都存在弊端，移动则内存回收时会更复杂，不移动则内存分配时会更复杂。从垃圾收集的停顿时间来看，不移动对象停顿时间会更短，甚至可以不需要停顿，但是从整个程序的吞吐量来看，移动对象会更划算。

44. 根节点枚举有了解过吗?

迄今为止，所有收集器在根节点枚举这一步骤时都是必须暂停用户线程的。现在可达性分析算法耗时最长的查找引用链的过程已经可以做到与用户线程一起并发，但根节点枚举始终还是必须在一个能**保障一致性的快照**中才得以进行——这里“一致性”的意思是整个枚举期间执行子系统看起来就像被冻结在某个时间点上，不会出现分析过程中，根节点集合的对象引用关系还在不断变化的情况，若这点不能满足的话，分析结果准确性也就无法保证。这是导致垃圾收集过程必须停顿所有用户线程的其中一个重要原因，即使是号称停顿时间可控，或者（几乎）不会发生停顿的CMS、G1、ZGC等收集器，枚举根节点时也是必须要停顿的。

由于目前主流Java虚拟机使用的都是**准确式垃圾收集**，所以当用户线程停顿下来之后，其实并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机是有办法直接得到哪些地方存放着对象引用的。在HotSpot的解决方案里，是使用一组称为**OopMap**的数据结构来达到这个目的。一旦类加载动作完成的时候，**HotSpot就会把对象内什么偏移量上是什么类型的数据计算出来，在即时编译过程中，也会在特定的位置记录下栈里和寄存器里哪些位置是引用**。这样收集器在扫描时就可以直接得知这些信息了，并不需要真正一个不漏地从方法区等GC Roots开始查找。

45. 什么是Jvm的安全点

在OopMap的协助下，HotSpot可以快速准确地完成GC Roots枚举，但一个很现实的问题随之而来：可能导致引用关系变化，或者说导致OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外存储空间，这样垃圾收集伴随而来的空间成本就会变得无法忍受的高昂。

实际上HotSpot也的确没有为每条指令都生成OopMap，只是在“特定的位置”记录了这些信息，这些位置被称为安全点（Safepoint）。有了安全点的设定，也就决定了用户程序执行时并非在代码指令流的任意位置都能够停顿下来开始垃圾收集，而是强制要求必须执行到达安全点后才能够暂停。因此，安全点的选定既不能太少以至于让收集器等待时间过长，也不能太过频繁以至于过分增大运行时的内存负荷。**安全点位置的选取基本上是以“是否具有让程序长时间执行的特征”为标准进行选定的**，因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这样的原因而长时间执行，**“长时间执行”的最明显特征就是指令序列的复用，例如方法调用、循环跳转、异常跳转等都属于指令序列复用，所以只有具有这些功能的指令才会产生安全点。**

46. 如何在垃圾收集发生时让所有线程都跑到最近的安全点

有两种方案可供选择：**抢先式中断（Preemptive Suspension）**和**主动式中断（Voluntary Suspension）**，**抢先式中断**不需要线程的执行代码主动去配合，在垃圾收集发生时，系统首先把所有用户线程全部中断，如果发现有用户线程中断的地方不在安全点上，就恢复这条线程执行，让它一会再重新中断，直到跑到安全点上。现在**几乎没有虚拟机**实现**采用抢先式中断来**暂停线程响应GC事件。

而**主动式中断**的思想是当垃圾收集需要中断线程的时候，**不直接对线程操作，仅仅简单地设置一个标志位**，各个**线程**执行过程时会不停地主动去轮询这个标志，一旦**发现中断标志为真时就自己在最近的安全点上主动中断挂起**。**轮询标志的地方和安全点是重合的**，另外还要加上所有创建对象和其他需要在Java堆上分配内存的地方，这是为了检查是否即将要发生垃圾收集，避免没有足够内存分配新对象。

47. 程序“不执行”的时候线程如何达到安全点?

安全点机制保证了程序执行时，在不太长的时间内就会遇到可进入垃圾收集过程的安全点。但是，程序“不执行”的时候呢？所谓的程序不执行就是没有分配处理器时间，典型的场景便是用户线程处于Sleep状态或者Blocked状态，这时候线程无法响应虚拟机的中断请求，不能再走到安全的地方去中断挂起自己，虚拟机也显然不可能持续等待线程重新被激活分配处理器时间。对于这种情况，就必须**引入安全区域（Safe Region）来解决**。

安全区域是指能够确保在某一段代码片段之中，引用关系不会发生变化，因此，在这个区域中任意地方开始垃圾收集都是安全的。我们也可以把安全区域看作被扩展拉伸了的安全点。

当用户线程执行到安全区域里面的代码时，首先会标识自己已经进入了安全区域，那样当这段时间里虚拟机要发起垃圾收集时就不必去管这些已声明自己在安全区域内的线程了。当线程要离开安全区域时，它要检查虚拟机是否已经完成了根节点枚举（或者垃圾收集过程中其他需要暂停用户线程的阶段），如果完成了，那线程就当作没事发生过，继续执行；否则它就必须一直等待，直到收到可以离开安全区域的信号为止。

48. 垃圾回收是如何处理跨代引用问题的?

跨代引用举例：假如要现在进行一次只局限于新生代区域内的收集（Minor GC），但新生代中的对象是完全有可能被老年代所引用的，为了找出该区域中的存活对象，不得不在固定的GC Roots之外，再额外遍历整个老年代中所有对象来确保可达性分析结果的正确性，反过来也是一样。

并不只是新生代、老年代之间才有跨代引用的问题，所有涉及部分区域收集（Partial GC）行为的垃圾收集器，典型的如G1、ZGC和Shenandoah收集器，都会面临相同的问题，JVM 为了用尽量少的资源消耗解决跨代引用下的垃圾回收问题，引入了**记忆集**。记忆集是一种用于记录从非收集区域指向收集区域的指针集合的抽象数据结构。

在垃圾收集的场景中，收集器只需要通过记忆集判断出某一块非收集区域是否存在有指向了收集区域的指针就可以了，并不需要了解这些跨代指针的全部细节。目前最常用的一种记忆集实现形式种称为“卡表”,卡表中的每个记录精确到一块内存区域（每块内存区域称之为卡页），该区域内有对象含有跨代指针。

一个卡页的内存中通常包含不止一个对象，只要卡页内有一个（或更多）对象的字段存在着跨代指针，那就将对应卡表的数组元素的值标识为1，称为这个元素变脏（Dirty），没有则标识为0。在垃圾收集发生时，只要筛选出卡表中变脏的元素，就能轻易得出哪些卡页内存块中包含跨代指针，把它们加入GC Roots中一并扫描。

49. 说说垃圾回收器的三色标记理论

了当前主流编程语言的垃圾收集器基本上都是依靠可达性分析算法来判定对象是否存活的，可达性分析算法理论上要求全过程都基于一个能保障一致性的快照中才能够进行分析，这意味着必须全程冻结用户线程的运行。在根节点枚举这个步骤中，由于GC Roots相比起整个Java堆中全部的对象毕竟还算是极少数，且在各种优化技巧（如OopMap）的加持下，它带来的停顿已经是非常短暂且相对固定（不随堆容量而增长）的了。

“标记”阶段是所有追踪式垃圾收集算法的共同特征，如果这个阶段会随着堆变大而等比例增加停顿时间，其影响就会波及几乎所有的垃圾收集器。

·**白色**：表示对象尚未被垃圾收集器访问过。显然在可达性分析刚刚开始的阶段，所有的对象都是白色的，若在分析结束的阶段，仍然是白色的对象，即代表不可达。

·**黑色**：表示对象已经被垃圾收集器访问过，且这个对象的所有引用都已经扫描过。黑色的对象代表已经扫描过，它是安全存活的，如果有其他对象引用指向了黑色对象，无须重新扫描一遍。黑色对象不可能直接（不经过灰色对象）指向某个白色对象。

·**灰色**：表示对象已经被垃圾收集器访问过，但这个对象上至少存在一个引用还没有被扫描过。

50. 说说什么是“对象消失”，垃圾回收器如何避免对象消失

“对象消失”的问题，即原本应该是黑色的对象被误标为白色。

当且仅当以下两个条件同时满足时，会产生“对象消失”的问题:

·赋值器插入了一条或多条从黑色对象到白色对象的新引用;

·赋值器删除了全部从灰色对象到该白色对象的直接或间接引用。

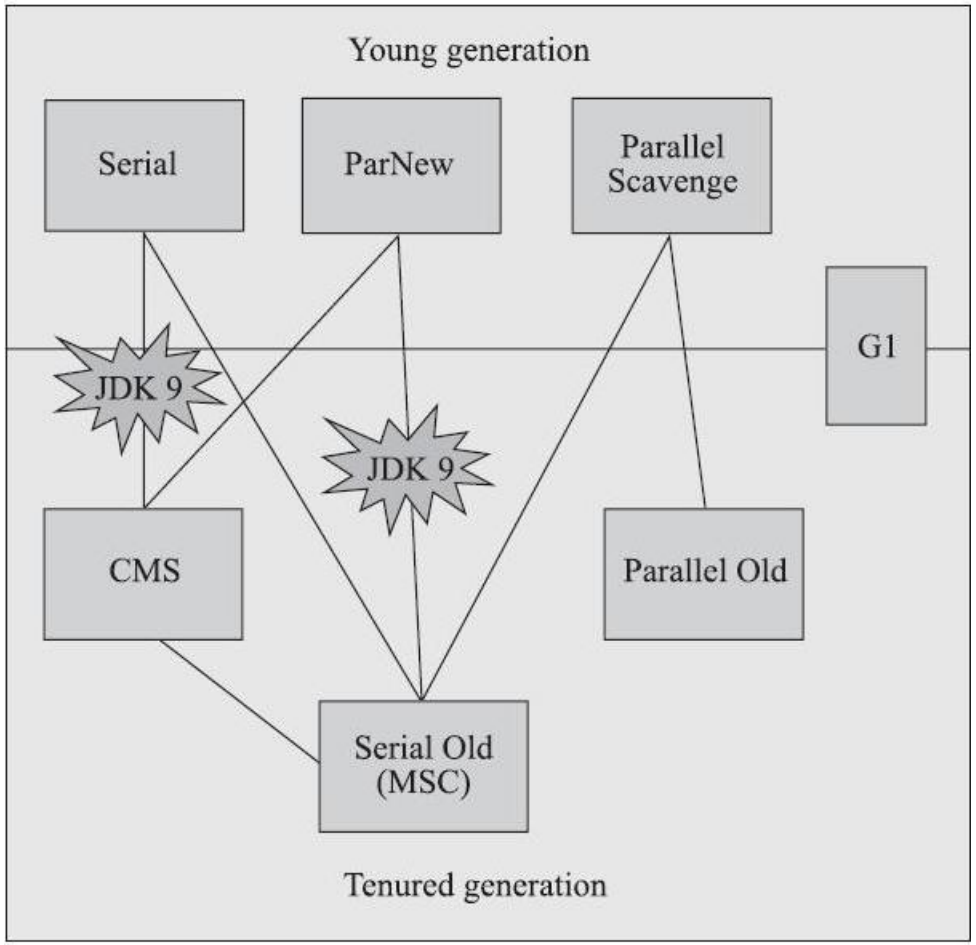
因此，我们要解决并发扫描时的对象消失问题，只需破坏这两个条件的任意一个即可。由此分别产生了两种解决方案：增量更新（Incremental Update）和原始快照（Snapshot At The Beginning，SATB）。

增量更新要破坏的是第一个条件，当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。这可以简化理解为，黑色对象一旦新插入了指向白色对象的引用之后，它就变回灰色对象了(因为新加入的白色节点未被扫描过)。

原始快照要破坏的是第二个条件，当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次。这也可以简化理解为，无论引用关系删除与否，都会按照刚刚开始扫描那一刻的对象图快照来进行搜索。

以上无论是对引用关系记录的插入还是删除，虚拟机的记录操作都是通过写屏障实现的。在HotSpot虚拟机中，增量更新和原始快照这两种解决方案都有实际应用，譬如，CMS是基于增量更新来做并发标记的，G1、Shenandoah则是用原始快照来实现。

51. 说说你所知道的垃圾回收器的种类



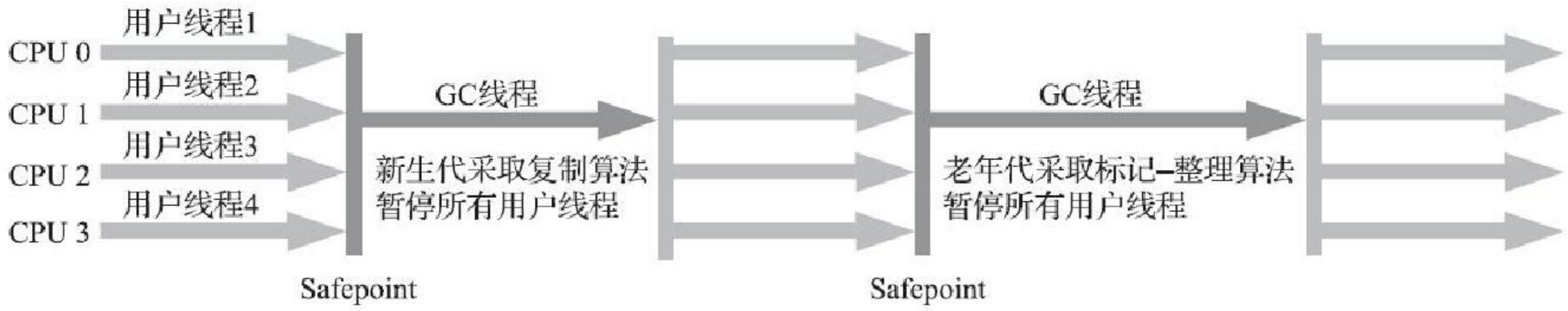
图中展示了七种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用，图中收集器所处的区域，则表示它是属于新生代收集器抑或是老年代收集器。

52. 说说Serial收集器的特点及使用场景

该收集器是一个**单线程工作的收集器**，但它的“单线程”的意义并不仅仅是说明它只会使用一个处理器或一条收集线程去完成垃圾收集工作，更重要的是强调在它进行垃圾收集时，必**须暂停其他所有工作线程，直到它收集结束。**

迄今为止，它依然是HotSpot虚拟机运行在**客户端模式**下的**默认新生代收集器**，有着优于其他收集器的地方，那就是简单而高效（与其他收集器的单线程相比），对于内存资源受限的环境，它是所有收集器里**额外内存消耗（Memory Footprint）最小**的；对于单核处理器或处理器核心数较少的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户桌面的应用场景以及近年来流行的部分微服务应用中，分配给虚拟机管理的内存一般来说并不会特别大，收集几十兆甚至一两百兆的新生代（仅仅是指新生代使用的内存，桌面应用甚少超过这个容量），垃圾收集的停顿时间完全可以控制在十几、几十毫秒，最多一百多毫秒以内，只要不是频繁发生收集，这点停顿时间对许多用户来说是完全可以接受的。所以，Serial收集器对于运行在客户端模式下的虚拟机来说是一个很好的选择。

Serial/Serial Old收集器运行示意图



53. ParNew收集器是如何进行垃圾回收的

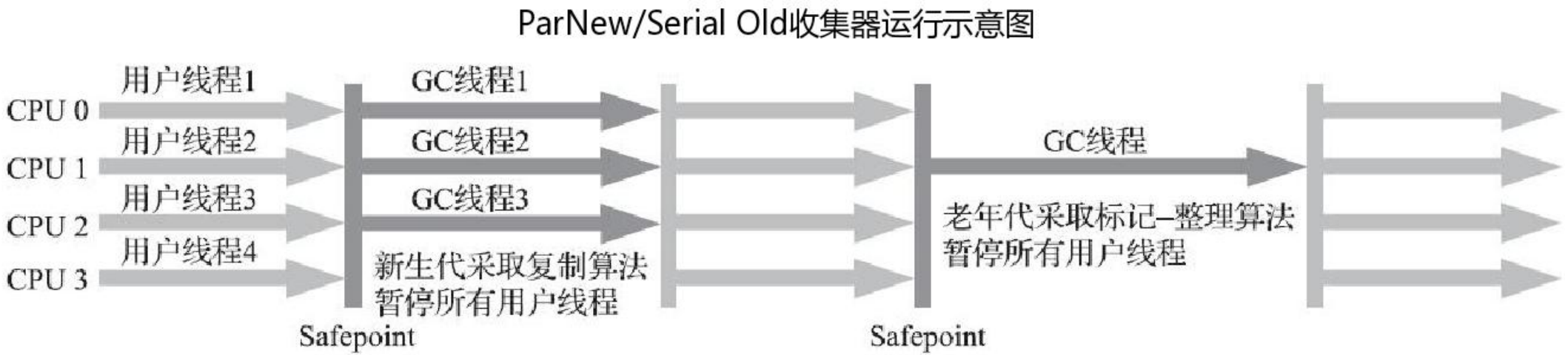
ParNew收集器**实质上是Serial收集器的多线程并行版本**，除了同时使用多条线程进行垃圾收集之外，其余的行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一致，在实现上这两种收集器也共用了相当多的代码。

ParNew收集器除了支持多线程并行收集之外，其他与Serial收集器相比并没有太多创新之处，但它却是不少运行在服务端模式下的HotSpot虚拟机，尤其是JDK 7之前的遗留系统中首选的新生代收集器，其中有一个与功能、性能无关但其实很重要的原因是：除了Serial收集器外，目前只有它能与CMS收集器配合工作。

在JDK 5发布时，HotSpot推出了一款在强交互应用中几乎可称为具有划时代意义的垃圾收集器——CMS收集器。这款收集器是HotSpot虚拟机中第一款真正意义上支持并发的垃圾收集器，它首次实现了让垃圾收集线程与用户线程（基本上）同时工作。

遗憾的是，CMS作为老年代的收集器，却无法与JDK 1.4.0中已经存在的新生代收集器Parallel Scavenge配合工作[1]，所以在JDK 5中使用CMS来收集老年代的时候，新生代只能选择ParNew或者Serial收集器中的一个。ParNew收集器是激活CMS后（使用-XX：+UseConcMarkSweepGC选项）的默认新生代收集器，也可以使用-XX：+/-UseParNewGC选项来强制指定或者禁用它。

ParNew收集器在单核心处理器的环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程（Hyper-Threading）技术实现的伪双核处理器环境中都不能百分之百保证超越Serial收集器。当然，随着可以被使用的处理器核心数量的增加，ParNew对于垃圾收集时系统资源的高效利用还是很有好处的。它默认开启的收集线程数与处理器核心数量相同，在处理器核心非常多（譬如32个，现在CPU都是多核加超线程设计，服务器达到或超过32个逻辑核心的情况非常普遍）的环境中，可以使用-XX：ParallelGCThreads参数来限制垃圾收集的线程数。



54. Parallel Scavenge收集器如何工作的，有什么特点

Parallel Scavenge收集器也是一款新生代收集器，它同样是基于标记-复制算法实现的收集器，也是能够**并行收集**的多线程收集器……Parallel Scavenge的诸多特性从表面上看和ParNew非常相似，那它有什么特别之处呢？

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是处理器用于运行用户代码的时间与处理器总消耗时间的比值，即：

$$吞吐量=\frac{运行用户代码时间}{运行用户代码时间+运行垃圾收集时间}$$

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX: MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX：GCTimeRatio参数。

-XX：MaxGCPauseMillis参数允许的值是一个大于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用户设定值。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更小一点就能使得系统的垃圾收集速度变得更快，垃圾收集停顿时间缩短是以牺牲吞吐量和新生代空间为代价换取的：系统把新生代调得小一些，收集300MB新生代肯定比收集500MB快，但这也直接导致垃圾收集发生得更频繁，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参数的值则应当是一个大于0小于100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。譬如把此参数设置为**19**，那允许的最大垃圾收集时间就占总时间的5%（**即1/(1+19)**），默认值为**99**，即允许最大1%（**即1/(1+99)**）的垃圾收集时间。

由于与吞吐量关系密切，Parallel Scavenge收集器也经常被称作“吞吐量优先收集器”。除上述两个参数之外，Parallel Scavenge收集器还有一个参数-XX：+**UseAdaptiveSizePolicy**值得我们关注。这是一个开关参数，当这个参数被激活之后，就不需要人工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX：SurvivorRatio）、晋升老年代对象大小（-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者最大的吞吐量。这种调节方式称为垃圾收集的自适应的调节策略（GC Ergonomics）

55. 说说Serial Old收集器的特点及使用场景

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法。这个收集器的主要意义也是供客户端模式下的HotSpot虚拟机使用。如果在服务端模式下，它也可能有两种用途：一种是在JDK 5以及之前的版本中与Parallel Scavenge收集器搭配使用，另外一种就是作为CMS收集器发生失败时的后备预案，在并发收集发生Concurrent Mode Failure时使用。

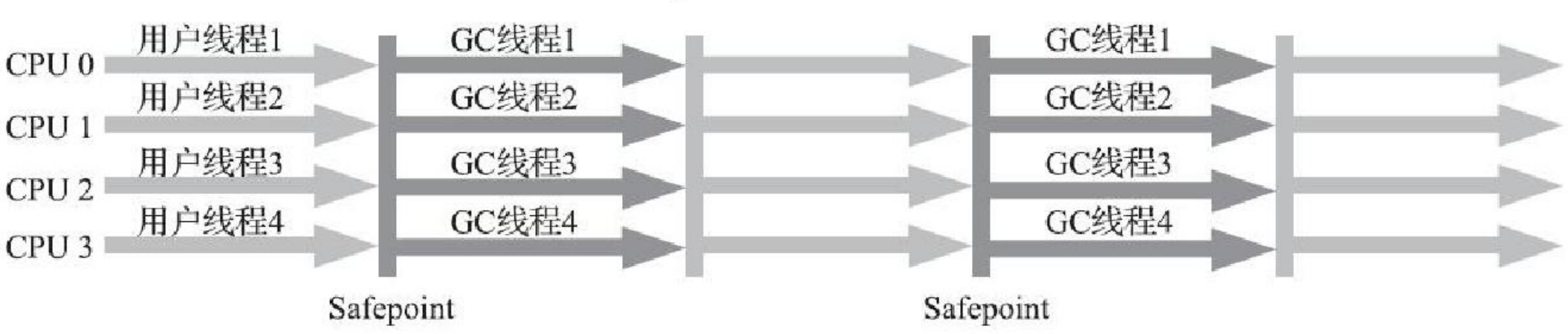
需要说明一下，Parallel Scavenge收集器架构中本身有PS MarkSweep收集器来进行老年代收集，并非直接调用Serial Old收集器，但是这个PS MarkSweep收集器与Serial Old的实现几乎是一样的，所以在官方的许多资料中都是直接以Serial Old代替PS MarkSweep进行讲解

56. 说说Parallel Old收集器的特点及使用场景

Parallel Old是Parallel Scavenge收集器的老年代版本，支持多线程并发收集，基于标记-整理算法实现。这个收集器是直到JDK 6时才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于相当尴尬的状态，原因是如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old（PS MarkSweep）收集器以外别无选择，其他表现良好的老年代收集器，如CMS无法与它配合工作。由于老年代Serial Old收集器在服务端应用性能上的“拖累”，使用Parallel Scavenge收集器也未必能在整体上获得吞吐量最大化的效果。同样，由于单线程的老年代收集中无法充分利用服务器多处理器的并行处理能力，在老年代内存空间很大而且硬件规格比较高级的运行环境中，这种组合的总吞吐量甚至不一定比ParNew加CMS的组合来得优秀。

直到Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的搭配组合，在注重吞吐量或者处理器资源较为稀缺的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器这个组合。

Parallel Scavenge/Parallel Old收集器运行示意图



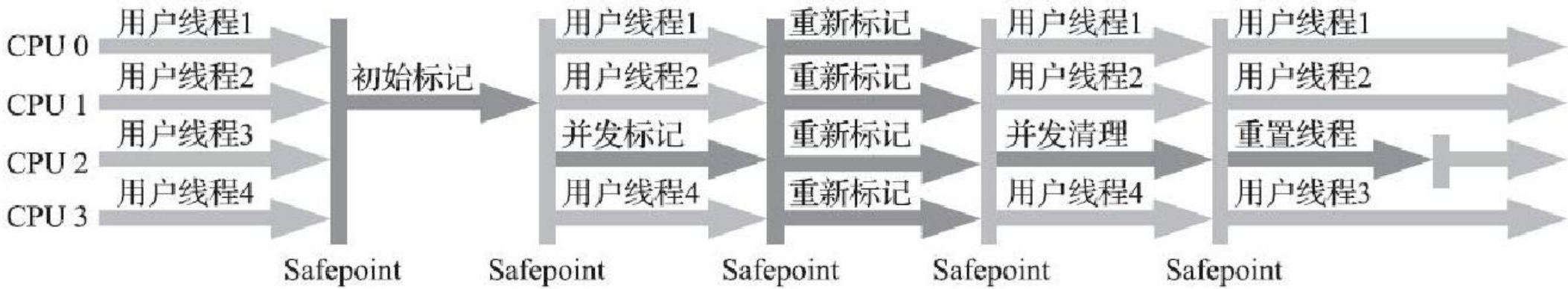
57. 说说CMS收集器的特点

CMS（**Concurrent Mark Sweep**）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网网站或者基于浏览器的B/S系统的服务端上，这类应用通常都会较为关注服务的响应速度，希望系统停顿时间尽可能短，以给用户带来良好的交互体验。CMS收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于标记-清除算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为**四个步骤**，包括：

1）初始标记（CMS initial mark）
2）并发标记（CMS concurrent mark）
3）重新标记（CMS remark）
4）并发清除（CMS concurrent sweep）

其中**初始标记、重新标记这两个步骤仍然需要“Stop The World”**。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快；并发标记阶段就是从GC Roots的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行；而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录（详见220题中关于增量更新的讲解），这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短；最后是并发清除阶段，清理删除掉标记阶段判断的已经死亡的对象，由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的。


CPU 0
CPU 1
CPU 2
CPU 3
用户线程1
用户线程2
用户线程3
用户线程4
初始标记
并发标记
重新标记
并发清理
重置线程
Safepoint
Safepoint
Safepoint
Safepoint
Safepoint

58. CMS收集器有什么缺点吗

CMS是一款优秀的收集器，它最主要的优点在名字上已经体现出来：并发收集、低停顿，一些官方公开文档里面也称之为“并发低停顿收集器”（Concurrent Low Pause Collector）。CMS收集器是HotSpot虚拟机追求低停顿的第一次成功尝试，但是它还远达不到完美的程度，至少有以下**三个明显的缺点**：

首先，CMS收集器对处理器资源非常敏感。事实上，面向并发设计的程序都对处理器资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但却会因为占用了一部分线程（或者说处理器的计算能力）而导致应用程序变慢，降低总吞吐量。**CMS默认启动的回收线程数是（处理器核心数量+3）/4**，也就是说，如果处理器核心数在四个或以上，并发回收时垃圾收集线程只占用不超过25%的处理器运算资源，并且会随着处理器核心数量的增加而下降。但是当处理器核心数量不足四个时，CMS对用户程序的影响就可能变得很大。如果应用本来的处理器负载就很高，还要分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然大幅降低。为了缓解这种情况，虚拟机提供了一种称为“**增量式并发收集器**”（Incremental Concurrent Mark Sweep/i-CMS）的CMS收集器变种，所做的事情和以前单核处理器年代PC机操作系统靠抢占式多任务来模拟多核并行多任务的思想一样，是在并发标记、清理的时候让收集器线程、用户线程交替运行，尽量减少垃圾收集线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得较少一些，直观感受是速度变慢的时间更多了，但速度下降幅度就没有那么明显。实践证明增量式的CMS收集器效果很一般，从JDK 7开始，i-CMS模式已经被声明为“deprecated”，即已过时不再提倡用户使用，**到JDK 9发布后i-CMS模式被完全废弃**。

然后，由于CMS收集器无法处理“浮动垃圾”（Floating Garbage），有可能出现“Con-current Mode Failure”失败进而导致另一次完全“Stop The World”的Full GC的产生。在CMS的并发标记和并发清理阶段，用户线程是还在继续运行的，程序在运行自然就还会伴随有新的垃圾对象不断产生，但这一部分垃圾对象是出现在标记过程结束以后，CMS无法在当次收集中处理掉它们，只好留待下一次垃圾收集时再清理掉。这一部分垃圾就称为“浮动垃圾”。同样也是由于在垃圾收集阶段用户线程还需要持续运行，那就还需要预留足够内存空间提供给用户线程使用，因此CMS收集器不能像其他收集器那样等待到老年代几乎完全被填满了再进行收集，必须预留一部分空间供并发收集时的程序运作使用。在JDK 5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在实际应用中老年代增长并不是太快，可以适当调高参数-XX：CMSInitiatingOccu-pancyFraction的值来提高CMS的触发百分比，降低内存回收频率，获取更好的性能。到了JDK 6时，CMS收集器的启动阈值就已经默认提升至92%。但这又会更容易面临另一种风险：要是CMS运行期间预留的内存无法满足程序分配新对象的需要，就会出现一次“并发失败”（Concurrent Mode Failure），这时候虚拟机将不得不启动后备预案：冻结用户线程的执行，临时启用Serial Old收集器来重新进行老年代的垃圾收集，但这样停顿时间就很长了。所以参数-XX：CMSInitiatingOccupancyFraction设置得太高将会很容易导致大量的并发失败产生，性能反而降低，用户应在生产环境中根据实际应用情况来权衡设置。

还有最后一个缺点，在本节的开头曾提到，CMS是一款基于“标记-清除”算法实现的收集器，如果对前面对垃圾回收器算法讲解还有印象的话，就可能想到这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很多剩余空间，但就是无法找到足够大的连续空间来分配当前对象，而不得不提前触发一次Full GC的情况。为了解决这个问题，CMS收集器提供了一个-XX：+UseCMS-CompactAtFullCollection开关参数（默认是开启的，此参数从JDK 9开始废弃），用于在CMS收集器不得不进行Full GC时开启内存碎片的合并整理过程，由于这个内存整理必须移动存活对象，（在Shenandoah和ZGC出现前）是无法并发的。这样空间碎片问题是解决了，但停顿时间又会变长，因此虚拟机设计者们还提供了另外一个参数-XX：CMSFullGCsBefore-Compaction（此参数从JDK 9开始废弃），这个参数的作用是要求CMS收集器在执行过若干次（数量由参数值决定）不整理空间的Full GC之后，下一次进入Full GC前会先进行碎片整理（默认值为0，表示每次进入Full GC时都进行碎片整理）。**只能预防，不能根治。**

59. 说说你对 G1 垃圾回收器的了解

Garbage First（简称G1）收集器是垃圾收集器技术发展历史上的里程碑式的成果，它开创了收集器**面向局部收集**的设计思路和**基于Region的内存布局**形式。

G1是一款主要面向服务端应用的垃圾收集器。HotSpot开发团队最初赋予它的期望是（在比较长期的）未来可以替换掉JDK 5中发布的CMS收集器。现在这个期望目标已经实现过半了，JDK 9发布之日，G1宣告取代Parallel Scavenge加Parallel Old组合，成为服务端模式下的默认垃圾收集器，而CMS则沦落至被声明为不推荐使用（Deprecate）的收集器。如果对JDK 9及以上版本的HotSpot虚拟机使用参数-XX：+UseConcMarkSweepGC来开启CMS收集器的话，用户会收到一个警告信息，提示CMS未来将会被废弃。

在G1收集器出现之前的所有其他收集器，包括CMS在内，垃圾收集的目标范围要么是整个新生代（Minor GC），要么就是整个老年代（Major GC），再要么就是整个Java堆（FullGC）。而G1跳出了这个樊笼，它可以面向堆内存任何部分来组成回收集（Collection Set，一般简称CSet）进行回收，衡量标准不再是它属于哪个分代，而是哪块内存中存放的垃圾数量最多，回收收益最大，这就是G1收集器的**Mixed GC模式**。

60. G1 收集器的 Region 堆内存布局是什么原理

虽然G1也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1不再坚持固定大小以及固定数量的分代区域划分，而是把连续的Java堆划分为多个大小相等的独立区域（Region），每一个Region都可以根据需要，扮演新生代的Eden空间、Survivor空间，或者老年代空间。收集器能够对扮演不同角色的Region采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。

Region中还有一类特殊的Humongous区域，专门用来存储大对象。G1认为只要大小超过了一个Region容量一半的对象即可判定为大对象。每个Region的大小可以通过参数-XX：G1HeapRegionSize设定，取值范围为1MB ~ 32MB，且应为2的N次幂。而对于那些超过了整个Region容量的超级大对象，将会被存放在N个连续的Humongous Region之中，G1的大多数行为都把Humongous Region作为老年代的一部分来进行看待。

虽然G1仍然保留新生代和老年代的概念，但新生代和老年代不再是固定的了，它们都是一系列区域（不需要连续）的动态集合。

61. G1收集器为什么能够建立可预测的停顿时间模型

G1收集器之所以能建立可预测的停顿时间模型，是因为它将Region作为单次回收的最小单元，即每次收集到的内存空间都是Region大小的整数倍，这样可以有计划地避免在整个Java堆中进行全区域的垃圾收集。

更具体的处理思路是让G1收集器去跟踪各个Region里面的垃圾堆积的“价值”大小，价值即回收所获得的空间大小以及回收所需时间的经验值，然后在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间（使用参数-XX：MaxGCPauseMillis指定，默认值是200毫秒），优先处理回收价值收益最大的那些Region，这也就是“Garbage First”名字的由来。这种使用Region划分内存空间，以及具有优先级的区域回收方式，保证了G1收集器在有限的时间内获取尽可能高的收集效率。

62. 多个独立Region里面存在的跨Region引用对象如何解决

解决的思路我们已经知道（之前提到的记忆集 218题）：使用记忆集避免全堆作为GC Roots扫描，但在G1收集器上记忆集的应用其实要复杂很多，它的每个Region都维护有自己的记忆集，这些记忆集会记录下别的Region指向自己的指针，并标记这些指针分别在哪些卡页的范围之内。G1的记忆集在存储结构的本质上是一种哈希表，Key是别的Region的起始地址，Value是一个集合，里面存储的元素是卡表的索引号。这种“双向”的卡表结构（卡表是“我指向谁”，这种结构还记录了“谁指向我”）比原来的卡表实现起来更复杂，同时由于Region数量比传统收集器的分代数量明显要多得多，因此G1收集器要比其他的传统垃圾收集器有着更高的内存占用负担。根据经验，G1至少要耗费大约相当于Java堆容量10%至20%的额外内存来维持收集器工作。

63. 在并发标记阶段如何保证收集线程与用户线程互不干扰地运行

CMS收集器采用增量更新算法实现，而G1收集器则是通过原始快照（SATB）算法来实现的。此外，垃圾收集对用户线程的影响还体现在回收过程中新创建对象的内存分配上，程序要继续运行就肯定会持续有新对象被创建，G1为每一个Region设计了两个名为TAMS（Top at Mark Start）的指针，把Region中的一部分空间划分出来用于并发回收过程中的新对象分配，并发回收时新分配的对象地址都必须要在这两个指针位置以上。G1收集器默认在这个地址以上的对象是被隐式标记过的，即默认它们是存活的，不纳入回收范围。与CMS中的“Concurrent Mode Failure”失败会导致Full GC类似，如果内存回收的速度赶不上内存分配的速度，G1收集器也要被迫冻结用户线程执行，导致Full GC而产生长时间“Stop The World”。

64.G1收集器垃圾回收的步骤

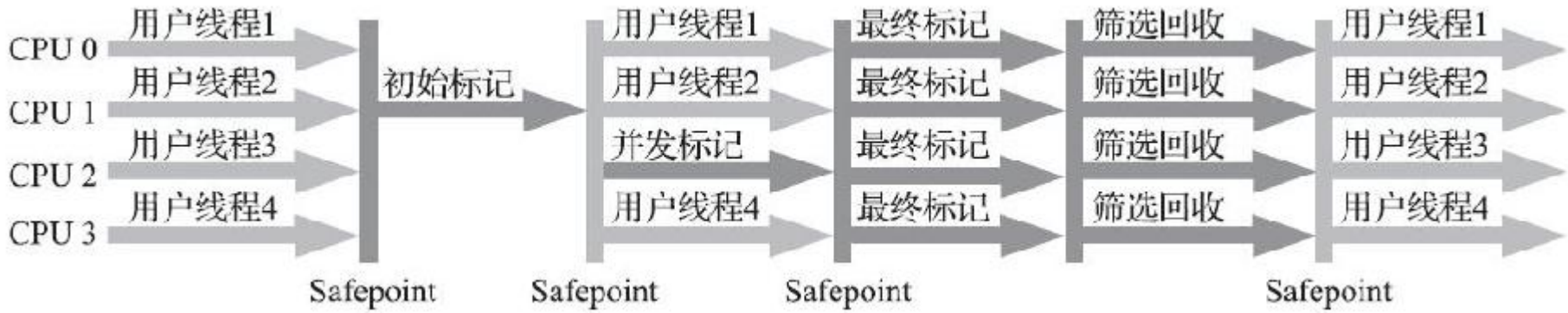
·**初始标记**（Initial Marking）：仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS指针的值，让下一阶段用户线程并发运行时，能正确地在可用的Region中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行Minor GC的时候同步完成的，所以G1收集器在这个阶段实际并没有额外的停顿。

·**并发标记**（Concurrent Marking）：从GC Root开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫描完成以后，还要重新处理SATB记录下的在并发时有引用变动的对象。

·**最终标记**（Final Marking）：对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的SATB记录。

·**筛选回收**（Live Data Counting and Evacuation）：负责更新Region的统计数据，对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个Region构成回收集，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。

从上述阶段的描述可以看出，G1收集器除了并发标记外，其余阶段也是要完全暂停用户线程的。


CPU 0
CPU 1
CPU 2
CPU 3
用户线程1
用户线程2
用户线程3
用户线程4
初始标记
并发标记
最终标记
筛选回收
Safepoint
Safepoint
Safepoint
Safepoint

65. Jvm 堆内存结构

把新生代分为一块较大的Eden空间和两块较小的Survivor空间，每次分配内存只使用Eden和其中一块Survivor。发生垃圾搜集时，将Eden和Survivor中仍然存活的对象一次性复制到另外一块Survivor空间上，然后直接清理掉Eden和已用过的那块Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8：1。

除新生代外，还有一个老年代。

关于永久代的问题，翻看面试题202.

66. 新创建的对象可能直接分配到老年代吗

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC。

对象就是指需要大量连续内存空间的Java对象，最典型的大对象便是那种很长的字符串，或者元素数量很庞大的数组。大对象对虚拟机的内存分配来说就是一个不折不扣的坏消息，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，我们写程序的时候应注意避免。

在Java虚拟机中要避免大对象的原因是，在分配空间时，它容易导致内存明明还有不少空间时就**提前触发垃圾收集，以获取足够的连续空间才能安置好它们**，而当复制对象时，大对象就意味着高额的内存复制开销。**HotSpot虚拟机提供了-XX：PretenureSizeThreshold参数，指定大于该设置值的对象直接在老年代分配**，这样做的目的就是避免在Eden区及两个Survivor区之间来回复制，产生大量的内存复制操作。

67. 老年代存放的都是什么对象

HotSpot虚拟机中多数收集器都采用了分代收集来管理堆内存，那内存回收时就必须能决策哪些存活对象应当放在新生代，哪些存活对象放在老年代中。为做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器，存储在对象头中。

对象通常在Eden区里诞生，如果经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，该对象会被移动到Survivor空间中，并且将其对象年龄设为1岁。对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（**默认为15**），就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数**-XX：MaxTenuringThreshold**设置。

68. Jvm中对象的年龄必须达到-XX：MaxTenuringThreshold才能晋升老年代吗？绝对吗?

为了能更好地适应不同程序的内存状况，HotSpot虚拟机并不是永远要求对象的年龄必须达到-XX:MaxTenuringThreshold才能晋升老年代，如果在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-XX:MaxTenuringThreshold中要求的年龄。

69. 什么是空间分配担保

在发生Minor GC之前，虚拟机必须先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那这一次Minor GC可以确保是安全的。如果不成立，则虚拟机会先查看-XX：HandlePromotionFailure参数的设置值是否允许担保失败（Handle Promotion Failure）；如果允许，那会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者-XX：HandlePromotionFailure设置不允许冒险，那这时就要改为进行一次Full GC。

70. Jvm查看进程状况工具jps有使用过吗

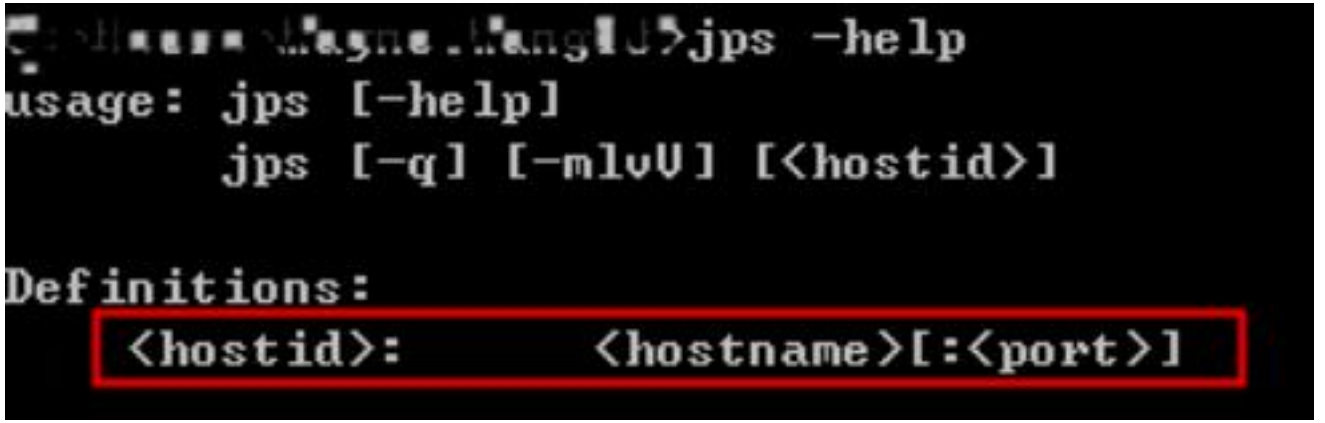
可以列出正在运行的**虚拟机进程**，并显示虚拟机**执行主类**（Main Class，main()函数所在的类）名称以及这些进程的本地虚拟机唯一ID（**LVMID**，Local Virtual Machine Identifier）。虽然功能比较单一，但它绝对是使用频率最高的JDK命令行工具，因为其他的JDK工具大多需要输入它查询到的LVMID来确定要监控的是哪一个虚拟机进程。对于**本地虚拟机**进程来说，**LVMID与操作系统的进程ID**（PID，Process Identifier）是**一致**的，使用Windows的任务管理器或者UNIX的ps命令也可以查询到虚拟机进程的LVMID，但**如果同时启动了多个虚拟机进程，无法根据进程名称定位时，那就必须依赖jps命令显示主类的功能才能区分了。**

命令格式： jps [options] [hostid]

选　　项	作　　用
-q	只输出 LVMID，省略主类的名称
-m	输出虚拟机进程启动时传递给主类 main() 函数的参数
-l	输出主类的全名，如果进程执行的是 JAR 包，则输出 JAR 路径
-v	输出虚拟机进程启动时的 JVM 参数

71. jps工具能查看远程虚拟机的运行状态吗?

jps可以通过RMI协议查询开启了RMI服务的远程虚拟机进程状态，参数hostid为RMI注册表中注册的主机名。



72. 使用过虚拟机 jstat 监控工具吗?

jstat（JVM Statistics Monitoring Tool）是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的**类加载、内存、垃圾收集、即时编译等运行时数据**，在没有GUI图形界面、只提供了纯文本控制台环境的服务器上，它将是**运行期定位虚拟机性能问题的常用工具**。

jstat命令格式为:

jstat [option vmid [interval[s|ms] [count]]]

参数**interval**和**count**代表**查询间隔和次数**，如果省略这2个参数，说明只查询一次。假设需要每250**毫秒**查询一次进程2764垃圾收集状况，一共查询20次，那命令应当是:

jstat -gc 2764 250 20

选项**option**代表用户希望查询的虚拟机信息，主要分为三类：**类加载**、**垃圾收集**、**运行期编译状况**。

选　　项	作　　用
-class	监视类加载、卸载数量、总空间以及类装载所耗费的时间
-gc	监视 Java 堆状况，包括 Eden 区、2 个 Survivor 区、老年代、永久代等的容量，已用空间，垃圾收集时间合计等信息
-gccapacity	监视内容与 -gc 基本相同，但输出主要关注 Java 堆各个区域使用到的最大、最小空间

-gcutil	监视内容与 -gc 基本相同，但输出主要关注已使用空间占总空间的百分比
-gccause	与 -gcutil 功能一样，但是会额外输出导致上一次垃圾收集产生的原因
-gcnew	监视新生代垃圾收集状况
-gcnewcapacity	监视内容与 -gcnew 基本相同，输出主要关注使用到的最大、最小空间
-gcold	监视老年代垃圾收集状况
-gcoldcapacity	监视内容与 -gcold 基本相同，输出主要关注使用到的最大、最小空间
-gcpermcapacity	输出永久代使用到的最大、最小空间
-compiler	输出即时编译器编译过的方法、耗时等信息
-printcompilation	输出已经被即时编译的方法

对于命令格式中的**VMID与LVMID**需要特别说明一下：如果是本地虚拟机进程，VMID与LVMID是一致的；如果是**远程虚拟机进程，那VMID的格式应当是**：

[protocol:][//]lvmid[@hostname[:port]/servername]

jstat执行样例

```
jstat -gcutil 2764
S0     S1     E      O      P      YGC    YGCT    FGC    FGCT    GCT
0.00   0.00   6.20   41.42  47.20  16     0.105   3      0.472   0.577
```

查询结果表明：这台服务器的新生代Eden区（E，表示Eden）使用了6.2%的空间，2个Survivor区（S0、S1，表示Survivor0、Survivor1）里面都是空的，老年代（O，表示Old）和永久代（P，表示Permanent）则分别使用了41.42%和47.20%的空间。程序运行以来共发生Minor GC（YGC，表示Young GC）16次，总耗时0.105秒；发生Full GC（FGC，表示Full GC）3次，总耗时（FGCT，表示Full GC Time）为0.472秒；所有GC总耗时（GCT，表示GC Time）为0.577秒。

73.jinfo 工具是做什么的

jinfo（Configuration Info for Java）的作用是实时查看和调整虚拟机各项参数。使用jps命令的-v参数可以查看虚拟机启动时显式指定的参数列表，但如果想知道**未被显式指定的参数的系统默认值**，j可以使用jinfo的-flag选项进行查询了。jinfo还可以使用-sysprops选项把虚拟机进程的System.getProperties()的内容打印出来。这个命令在JDK 5时期已经随着Linux版的JDK发布，当时只提供了信息查询的功能，JDK 6之后，jinfo在Windows和Linux平台都有提供，并且**加入了在运行期修改部分参数值的能力（可以使用-flag[+|-]name或者-flag name=value在运行期修改一部分运行期可写的虚拟机参数值）**

jinfo命令格式：

jinfo [option] pid

jinfo -flag CMSInitiatingOccupancyFraction 1444

-XX:CMSInitiatingOccupancyFraction=85

74. jmap 工具是做什么的?

jmap（Memory Map for Java）命令**用于生成堆转储快照**（一般称为heapdump或dump文件）。如果不使用jmap命令，要想获取Java堆转储快照也还有一些比较“暴力”的手段：譬如在第2章中用过的-XX:+HeapDumpOnOutOfMemoryError参数，可以让虚拟机在内存溢出异常出现之后自动生成堆转储快照文件，通过-XX：+HeapDumpOnCtrlBreak参数则可以使用[Ctrl]+[Break]键让虚拟机生成堆转储快照文件，又或者在Linux系统下通过Kill-3命令发送进程退出信号“恐吓”一下虚拟机，也能顺利拿到堆转储快照。

jmap的作用并不仅仅是为了获取堆转储快照，它还可以**查询finalize执行队列**、**Java堆**和**方法区的详细信息**，如**空间使用率**、**当前用的是哪种收集器**等。

jmap命令格式:
jmap [option] vmid

使用jmap生成dump文件

```
jmap -dump:format=b,file=eclipse.bin 3500
Dumping heap to C:\Users\IcyFenix\eclipse.bin ...
Heap dump file created
```

命令中的 format=b，代表的是格式化为 binary 格式

option 选项中的命令如下图所示：

选　　项	作　　用
-dump	生成 Java 堆转储快照。格式为 -dump:[live,]format=b,file=<filename>，其中 live 子参数说明是否只 dump 出存活的对象
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux/Solaris 平台下有效
-heap	显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在 Linux/Solaris 平台下有效
-histo	显示堆中对象统计信息，包括类、实例数量、合计容量
-permstat	以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux/Solaris 平台下有效
-F	当虚拟机进程对 -dump 选项没有响应时，可使用这个选项强制生成 dump 快照。只在 Linux/Solaris 平台下有效

75. 有试用过 jhat 分析过转存储快照吗

JDK提供jhat（JVM Heap Analysis Tool）命令与jmap搭配使用，来分析jmap生成的堆转储快照。jhat内置了一个微型的HTTP/Web服务器，生成堆转储快照的分析结果后，可以在浏览器中查看。不过实事求是地说，在实际工作中，除非手上真的没有别的工具可用，否则多数人是不会直接使用jhat命令来分析堆转储快照文件的，主要原因有两个方面。一是一般不会在部署应用程序的服务器上直接分析堆转储快照，即使可以这样做，也会尽量将堆转储快照文件复制到其他机器上进行分析，因为分析工作是一个耗时而且极为耗费硬件资源的过程，既然都要在其他机器上进行，就没有必要再受命令行工具的限制了。另外一个原因是jhat的分析功能相对来说比较简陋，后面题目将会介绍到的VisualVM，以及专业用于分析堆转储快照文件的Eclipse Memory Analyzer、IBM HeapAnalyzer等工具，都能实现比jhat更强大专业的分析功能。

使用jhat分析dump文件

```
jhat eclipse.bin
Reading from eclipse.bin...
Dump file created Fri Nov 19 22:07:21 CST 2010
Snapshot read, resolving...
Resolving 1225951 objects...
Chasing references, expect 245 dots....
Eliminating duplicate references...
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

屏幕显示“Server is ready.”的提示后，用户在浏览器中输入http://localhost:7000/可以看到分析结果


http://localhost:7000/
All Classes (excluding platf...
Package org.osgi.service.url
class org.osgi.service.url.AbstractURLStreamHandlerService [0x5fde8a8]
class org.osgi.service.url.URLStreamHandlerService [0x5fdd868]
class org.osgi.service.url.URLStreamHandlerSetter [0x716f858]
Package org.osgi.util.tracker
class org.osgi.util.tracker.AbstractTracked [0x5e73888]
class org.osgi.util.tracker.ServiceTracker [0x5e702a0]
class org.osgi.util.tracker.ServiceTracker$1 [0x5fb3a50]
class org.osgi.util.tracker.ServiceTracker$AllTracked [0x5e74090]
class org.osgi.util.tracker.ServiceTracker$Tracked [0x5e73c58]
class org.osgi.util.tracker.ServiceTrackerCustomizer [0x5d2a428]
Other Queries
All classes including platform
Show all members of the rootset
Show instance counts for all classes (including platform)
Show instance counts for all classes (excluding platform)
Show heap histogram
Show finalizer summary
Execute Object Query Language (OQL) query
分析结果默认以包为单位进行分组显示，分析内存泄漏问题主要会使用到其中的“Heap Histogram”
402/1412
Internet | 保护模式: 禁用
100%

76. jstack 工具是做什么的

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照（一般称为threaddump或者javacore文件）。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间挂起等，都是导致线程长时间停顿的常见原因。线程出现停顿时通过jstack来查看各个线程的调用堆栈，就可以获知没有响应的线程到底在后台做些什么事情，或者等待着什么资源。

jstack命令格式：
jstack [option] vmid

option选项的合法值与具体含义下图所示。

选　　项	作　　用
-F	当正常输出的请求不被响应时，强制输出线程堆栈
-l	除堆栈外，显示关于锁的附加信息
-m	如果调用到本地方法的话，可以显示 C/C++ 的堆栈

使用jstack查看线程堆栈（部分结果）

```
jstack -l 3500
2010-11-19 23:11:26
Full thread dump Java HotSpot(TM) 64-Bit Server VM (17.1-b03 mixed mode):
"[ThreadPool Manager] - Idle Thread" daemon prio=6 tid=0x0000000039dd4000 nid= 0xf50 in Object.wait()
[0x000000003c96f000]
   java.lang.Thread.State: WAITING (on object monitor)
      at java.lang.Object.wait(Native Method)
      - waiting on <0x0000000016bdcc60> (a org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor)
      at java.lang.Object.wait(Object.java:485)
      at org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor.run (Executor. java:106)
      - locked <0x0000000016bdcc60> (a org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor)
   Locked ownable synchronizers:
      - None
```

从JDK 5起，java.lang.Thread类新增了一个getAllStackTraces()方法用于获取虚拟机中所有线程的StackTraceElement对象。使用这个方法可以通过简单的几行代码完成jstack的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈

77. 什么是内存泄漏，与内存溢出有什么关系

内存泄漏：是指创建的对象已经没有用处，正常情况下应该会被垃圾收集器回收，但是由于该对象仍然被其他对象进行了无效引用，导致不能够被垃圾收集器及时清理，这种现象称之为内存泄漏。

内存泄漏会导致内存堆积，最终发生内存溢出，导致OOM。

发生内存泄漏大部分是由于程序代码导致的，排查方法一般是使用 visualVM 进行heap dump，查看占用空间比较多的 class 对象，然后检查该对象的instances 以及 reference引用，最终定位到程序代码。

如果堆内存比较大，进行head dump 产生的资源消耗不可接受，可以尝试使用轻量级的jmap生成堆转储快照分析，思路与使用可视化工具一样。

78. 什么是对象逃逸，对象逃逸分析的优化有几种

逃逸分析的基本原理是：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，例如作为调用参数传递到其他方法中，这种称为方法逃逸；甚至还有可能被外部线程访问到，譬如赋值给可以在其他线程中访问的实例变量，这种称为线程逃逸；从不逃逸、方法逃逸到线程逃逸，称为对象由低到高的不同逃逸程度。

优化有三种：栈上分配；标量替换；锁消除（或称同步消除）。

·**栈上分配**（Stack Allocations）：在Java虚拟机中，Java堆上分配创建对象的内存空间几乎是Java程序员都知道的常识，Java堆中的对象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，就可以访问到堆中存储的对象数据。虚拟机的垃圾收集子系统会回收堆中不再使用的对象，但回收动作无论是标记筛选出可回收对象，还是回收和整理内存，都需要耗费大量资源。如果确定一个对象不会逃逸出线程之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，完全不会逃逸的局部对象和不会逃逸出线程的对象所占的比例是很大的，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集子系统的压力将会下降很多。栈上分配可以支持方法逃逸，但不能支持线程逃逸。

·**标量替换**（Scalar Replacement）：若一个数据已经无法再分解成更小的数据来表示了，Java虚拟机中的原始数据类型（int、long等数值类型及reference类型等）都不能再进一步分解了，那么这些数据就可以被称为标量。相对的，如果一个数据可以继续分解，那它就被称为聚合量（Aggregate），Java中的对象就是典型的聚合量。如果把一个Java对象拆散，根据程序访问的情况，将其用到的成员变量恢复为原始类型来访问，这个过程就称为标量替换。假如逃逸分析能够证明一个对象不会被方法外部访问，并且这个对象可以被拆散，那么程序真正执行的时候将可能不去创建这个对象，而改为直接创建它的若干个被这个方法使用的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上（栈上存储的数据，很大机会被虚拟机分配至物理机器的高速寄存器中存储）分配和读写之外，还可以为后续进一步的优化手段创建条件。标量替换可以视作栈上分配的一种特例，实现更简单（不用考虑整个对象完整结构的分配），但对逃逸程度的要求更高，它不允许对象逃逸出方法范围内。

·**同步消除**（Synchronization Elimination）：线程同步本身是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那么这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以安全地消除掉。

```
public String concatString(String s1, String s2, String s3) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}
```

每个StringBuffer.append()方法中都有一个同步块，锁就是sb对象。虚拟机观察变量sb，经过逃逸分析后会发现它的动态作用域被限制在concatString()方法内部。也就是sb的所有引用都永远不会逃逸到concatString()方法之外，其他线程无法访问到它，所以这里虽然有锁，但是可以被安全地消除掉。在解释执行时这里仍然会加锁，但在经过服务端编译器的即时编译之后，这段代码就会忽略所有的同步措施而直接执行。

79. 什么是锁粗化

```
for（；；）{
        加锁操作;
}
```

如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

```
加锁操作：{
        for（；；）{

        }
}
```

80. 有过JVM调优经验吗?

JVM调优情况十分复杂，各种情况都可能导致垃圾回收不能够达到预想的效果。对于场景问题，可以从如下几个大方向进行设计：

1. 在大访问压力下，MinorGC 频繁，MinorGC 是针对新生代进行回收的，每次在MGC 存活下来的对象，会移动到Survivor1区。先到这里为止，大访问压力下，MGC 频繁一些是正常的，只要MGC 延迟不导致停顿时间过长或者引发FGC，那可以适当的增大Eden 空间大小，降低频繁程度，同时要保证，空间增大对垃圾回收时间产生的停顿时间增长也是可以接受的。

继续说，如果MinorGC 频繁，且容易引发 Full GC。需要从如下几个角度进行分析。a：每次MGC存活的对象的大小，是否能够全部移动到 S1区，如果S1 区大小 < MGC 存活的对象大小，这批对象会直接进入老年代。注意了，这批对象的年龄才1岁，很有可能再多等1次MGC 就能被回收了，可是却进入了老年代，只能等到Full GC 进行回收，很可怕。这种情况下，应该在系统压测的情况下，实时监控MGC存活的对象大小，并合理调整eden和s区的大小以及比例。还有一种情况会导致对象在未达到15岁之前，直接进入老年代，就是S1区的对象，相同年龄的对象所占总空间大小>s1区空间大小的一半，所以为了应对这种情况，对于S区的大小的调整就要考虑：尽量保证峰值状态下，S1区的对象所占空间能够在MGC的过程中，相同对象年龄所占空间不大于S1区空间的一半，因此对于S1空间大小的调整，也是十分重要的。

2. 由于大对象创建频繁，导致Full GC 频繁。对于大对象，JVM专门有参数进行控制，-XX: PretenureSizeThreshold。超过这个参数值的对象，会直接进入老年代，只能等到full GC 进行回收，所以在系统压测过程中，要重点监测大对象的产生。如果能够优化对象大小，则进行代码层面的优化，优化如：根据业务需求看是否可以将该大对象设置为单例模式下的对象，或者该大对象是否可以进行拆分使用，或者如果大对象确定使用完成后，将该对象赋值为null，方便垃圾回收。如果代码层面无法优化，则需要考虑：a:调高-XX: PretenureSizeThreshold参数的大小，使对象有机会在eden区创建，有机会经历MGC以被回收。但是这个参数的调整要结合MGC过程中Eden区的大小是否能够承载，包括S1区的大小承载问题。b：这是最不希望发生的情况，如果必须要进入老年代，也要尽量保证，该对象确实是长时间使用的对象，放入老年代的总对象创建量不会造成老年代的内存空间迅速长满发生Full GC，在这种情况下，可以通过定时脚本，在业务系统不繁忙情况下，主动触发full gc。

3.MGC 与 FGC 停顿时间长导致影响用户体验。其实对于停顿时间长的问题无非就两种情况：a：gc 真实回收过程时间长，即real time时间长。这种时间长大部分是因为内存过大导致，导致从标记到清理的过程中需要对很大的空间进行操作，导致停顿时间长。b：gc真实回收时间 real time 并不长，但是user time(用户态执行时间) 和 sys time（核心态执行时间）时间长，导致从客户角度来看，停顿时间过长。对于a情况，要考虑减少堆内存大小，包括新生代和老年代，比如之前使用16G的堆内存，可以考虑将16G 内存拆分为4个4G的内存区域，可以单台机器部署JVM逻辑集群，也可以为了降低GC回收时间进行4节点的分布式部署，这里的分布式部署是为了降低GC垃圾回收时间。对于b情况，要考虑线程是否及时达到了安全点，通过-XX：+PrintSafepointStatistics和-XX: PrintSafepointStatisticsCount=1去查看安全点日志，如果有长时间未达到安全点的线程，再通过参数-XX: +SafepointTimeout和-XX：SafepointTimeoutDelay=2000两个参数来找到大于2000ms到达安全点的线程，这里的2000ms可以根据情况自己设置，然后对代码进行针对的调整。除了安全点问题，也有可能是操作系统本身负载比较高，导致处理速度过慢，线程达到安全点时间长，因此需要同时检测操作系统自身的运行情况。

4.内存泄漏导致的MGC和FGC频繁，最终引发oom。内存泄漏的排查参见面试题247.

5. 纯代码级别导致的MGC和FGC频繁。如果是这种情况，那就只能对代码进行大范围的调整，这种情况就非常多了，而且会很糟糕。如大循环体中的new 对象，未使用合理容器进行对象托管导致对象创建频繁，不合理的数据结构使用等等。

总之，JVM的调优无非就一个目的，在系统可接受的情况下达到一个合理的MGC和FGC的频率以及可接受的回收时间。