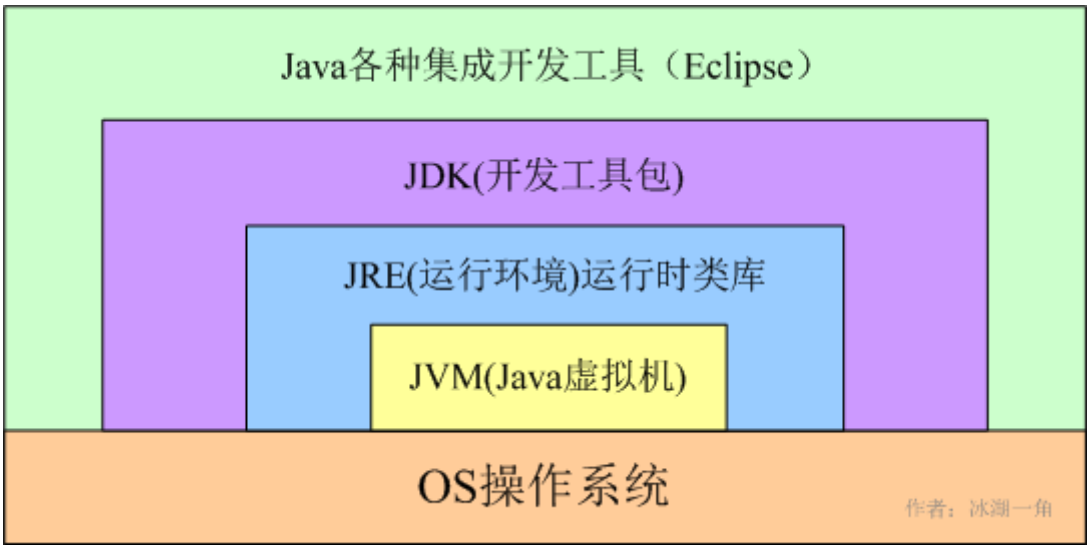
1 JVM前奏篇

1.1 JDK、JRE、JVM

Java程序是运行在JVM(Java虚拟机)上的，在开发程序之前都要配置Java开发环境，其中首先要做的就是JDK的安装和配置，那么JDK、JVM、JRE到底有何联系和区别呢？想必并不是每一个程序员都能说得清楚的，本文接下来将带你了解它们之间的关系。



1.1.1 JDK

JDK(Java SE Development Kit)，Java标准开发包，它提供了编译、运行Java程序所需的各种工具和资源，包括Java编译器、Java运行时环境，以及常用的Java类库等。

下图是JDK的安装目录：

1.1.2 JRE

JRE(Java Runtime Environment) 、Java运行环境，用于解释执行Java的字节码文件。普通用户而只需要安装 JRE（Java Runtime Environment）来运行 Java 程序。而程序开发者必须安装JDK来编译、调试程序。

下图是JRE的安装目录：里面有两个文件夹bin和lib，在这里可以认为bin里的就是jvm，lib中则是jvm工作所需要的类库，而jvm和 lib和起来就称为jre。

1.1.3 JVM

JVM(Java Virtual Machine)，Java虚拟机，是JRE的一部分。它是整个java实现跨平台的最核心的部分，负责解释执行字节码文件，是可运行java字节码文件的虚拟计算机。所有平台的上的JVM向编译器提供相同的接口，而编译器只需要面向虚拟机，生成虚拟机能识别的代码，然后由虚拟机来解释执行。

当使用Java编译器编译Java程序时，生成的是与平台无关的字节码，这些字节码只面向JVM。不同平台的JVM都是不同的，但它们都提供了相同的接口。JVM是Java程序跨平台的关键部分，只要为不同平台实现了相应的虚拟机，编译后的Java字节码就可以在该平台上运行。

1.1.4 区别与联系

1. JDK 用于开发，JRE 用于运行java程序；如果只是运行Java程序，可以只安装JRE，无需安装JDK。
2. JDk包含JRE，JDK 和 JRE 中都包含 JVM。
3. JVM 是 java 编程语言的核心并且具有平台独立性。

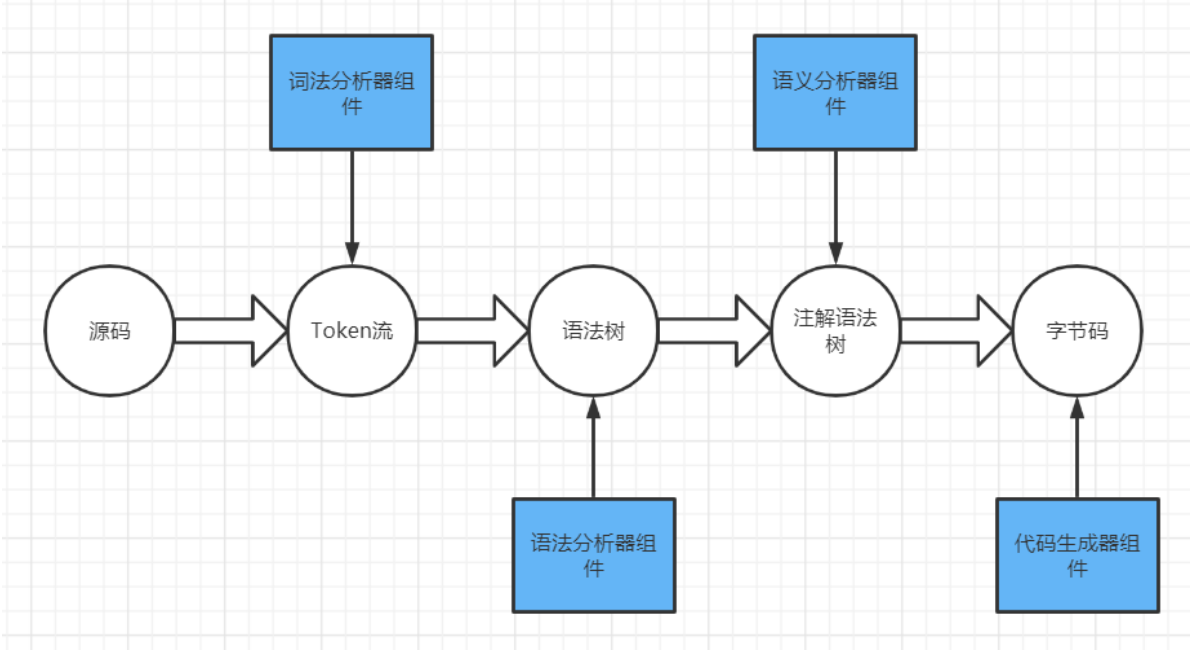
官网链接：https://docs.oracle.com/javase/8/docs/index.html

1.2 java变class

也就是我们说的编译过程，由java编译器把java文件变成class的字节码文件

javac ClassB.java

javap -verbose ClassB.class



词法分析器：读取源代码，一个字节一个字节的读取出来，找出这个词法中if,else,while等

结论：就是分析源码哪些是标点符号，哪些是动词，哪些是名词，并转化成Token流形式

语法分析器：分析Token流，检查是否符合Java语言规范

结论：检查Token流是否符合JAVA语言规范，有没有语法错误。

例如：if后面是否跟着boolean表达式

语义分析器：主要工作就是将比较复杂的语法语义转化成简单的语法，

就如难懂的文言文转化为大家都懂的百话文，或者是注释一下一些不懂的成语。

结论：就是将复杂的语法转化为简单语法。例如：foreach转化为for循环的过程

代码生成器：将注解语法树生转化成字节码的过程

结论：代码生成器的结果就是生成符合java虚拟机规范的字节码

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.1

ClassFile结构

```
ClassFile {
    u4             magic;    //魔法数，Class文件的标识。值是固定的，为0xCAFEBABE
    u2             minor_version;    //小版本
    u2             major_version;  //主版本号 34，52对应java 8 ,也就是我们jdk8生成的
class版本
    u2             constant_pool_count;    //38  常量池计算器
    cp_info        constant_pool[constant_pool_count-1];    //常量池
    u2             access_flags;  //访问标志  标志了类或者接口的访问信息，比如是类还是接
口还是注解、枚举，是否是abstract,如果是类，是否被声明成final等等
    u2             this_class;  //类索引
    u2             super_class; //父类索引
    u2             interfaces_count;    //接口计数器  ，这也是之前大家在讨论动态代理的时
候最多是65535个接口的时候讨论过的
    u2             interfaces[interfaces_count];
    u2             fields_count;    //字段个数
    field_info     fields[fields_count];  //字段集合
    u2             methods_count;    //方法计数器
    method_info    methods[methods_count];  //方法集合
    u2             attributes_count;  //附加属性计数器
    attribute_info attributes[attributes_count];    //附加属性集合
}
```

我们的class文件发现有2种数据类型

u几u几，这种就是无符号数，属于基本的数据类型，用来描述数字、索引引用，数量值等，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

_info结尾的，这种数据类型叫做表，都是以 _info结尾

magic

魔数，区分文件类型的一种标志，一般都是文件的前几个字节来表示，比如0XCAFE BABE表示class文件，因为我们java的图标是杯cafe，因为为了蹭当时的热度，当时咖啡最盛产的地方就是爪哇岛，所以名字叫java,图标就是cafe，然后java bean就是咖啡豆

minor_version major_version

次要和主要版本号，一起确定class文件格式的版本

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7 有版本说明！！

所以我们的这个class的版本是52，所以是jdk1.8编译

constant_pool_count & constant_pool[constant_pool_count-1]

数量，会比实际多1

我们写代码时都是从0开始的，但是这里的常量池却是从1开始，因为它把第0项常量空出来了。这是为了在于满足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，这种情况可用索引值0来表示

我们可以通过命令查看常量池信息

```
javap -verbose ClassB.class
```

常量池主要存放2大类常量：字面量和符号引用

字面量：文本字符串，声明为final的常量池

类 型 上面的tag对应这里的标志	标 志	描 述
CONSTANT_utf8_info	1	UTF-8编码的字符串
CONSTANT_Integer_info	3	整形字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MothodType_info	16	标志方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

Class文件结构中常量池中11种数据类型的结构总表			
常量	项目	类型	描述
CONSTANT_Utf8_info	tag	U1	值为1
	length	U2	UTF-8编码的字符串长度
	bytes	U1	长度为length的UTF-8编码的字符串
CONSTANT_Integer_info	tag	U1	值为3
	bytes	U4	按照高位在前存储的int值
CONSTANT_Float_info	tag	U1	值为4
	bytes	U4	按照高位在前存储的float值
CONSTANT_Long_info	tag	U1	值为5
	bytes	U8	按照高位在前存储的long值
CONSTANT_Double_info	tag	U1	值为6
	bytes	U8	按照高位在前存储的double值
CONSTANT_Class_info	tag	U1	值为7
	index	U2	指向全限定名常量项的索引
CONSTANT_String_info	tag	U1	值为8
	index	U2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	U1	值为9
	index	U2	指向声明字段的类或者接口描述符CONSTANT_Class_info的索引项
	index	U2	指向字段描述符CONSTANT_NameAndType_info的索引项
CONSTANT_Methodref_info	tag	U1	值为10
	index	U2	指向声明方法的类描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_InterfaceMethodref_info	tag	U1	值为11
	index	U2	指向声明方法的接口描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_NameAndType_info	tag	U1	值为12
	index	U2	指向该字段或方法名称常量项的索引
	index	U2	指向该字段或方法描述符常量项的索引

常量池各个表结构

所以我们调用final的时候，在类不加载的时候也能访问，但是有个坑，常量池的值必须固定

```
public static final String str = "classB str";
public final String str = "classB str";
```

符号引用：类和接口的全限定名、字段的名称和描述符、方法的名称和描述符

访问标志：

```
00000000 00000001：public
00000000 00010000：final
00000000 00100000：super
00000010 00000000：interface
00000100 00000000：abstract
00010000 00000000：synthetic
00100000 00000000：annotation
01000000 00000000：enum
```

1.3 class类的生命周期

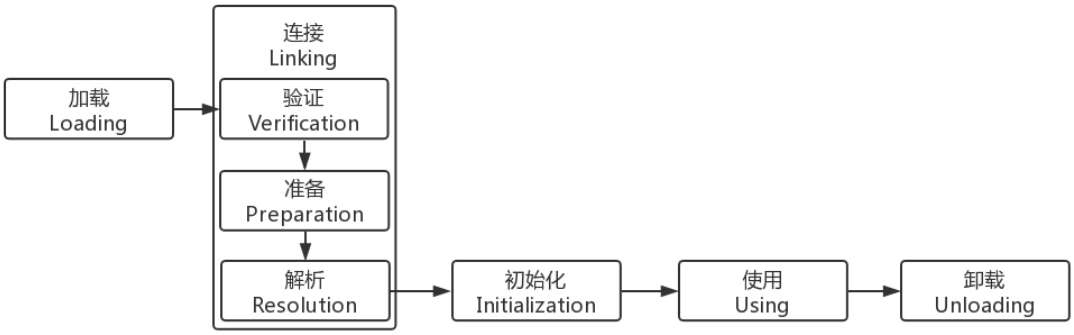
也就是我们经常讲的类加载！！！把class类加载到JVM

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

在Java语言里面，类型的加载、连接和初始化过程都是在程序运行期间完成的

1.3.1 类加载的过程

类的整个生命周期如下图：



为支持运行时绑定，解析过程在某些情况下可在初始化之后再开始，除解析过程外的其他加载过程必须按照如图顺序开始。

1.3.2 加载（load）

1. 通过全限定类名来获取定义此类的二进制字节流。（先找到类文件所在的位置：磁盘的全路径）
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。（类文件的信息交给JVM）
3. 在内存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。（类文件所对应的对象Class ---->JVM）

1.3.2.1 引导类加载器（Bootstrap ClassLoader ）(启动类加载器)

加载$JAVA_HOME中 jre/lib/rt.jar里所有的class或Xbootclassoath选项指定的jar包! 这些类是 Java 运行的基础类，BootstrapClassLoader 比较特殊，它不继承 ClassLoader，而是由 JVM 内部实现；

C++实现，并非java代码实现，所以在JVM虚拟机规范中的支持两种类型的类加载器，分别为引导类加载器（Bootstrap ClassLoader）和自定义类加载器（User-Defin ed ClassLoader），什么拓展类，应用程序类、自动类都可以称为自定义类加载器

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.3.1 官网就2个

1.3.2.2 扩展类加载器（Extension ClassLoader）

加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/ext/*.jar或-Djava.ext.dirs（启动指令属性）指定目录下的jar包，比如

C:\Program Files\Java\jdk1.8.0_152\jre\lib下的所有jar包的类

具体实现为sum.misc.Launcher中的内部类ExtClassLoader

1.3.2.3 应用程序（系统）类加载器（Application ClassLoader）

加载classpath中指定的jar包及 Djava.class.path所指定目录下的类和jar包

具体实现为sum.misc.Launcher中的内部类AppClassLoader

1.3.2.4 自定义类加载器（Custom ClassLoader）

前面3个是java自带的，提供好了的，那么我们也可以自定义！

通过java.lang.ClassLoader的子类自定义加载class，属于应用程序根据自身需要自定义的，ClassLoader，如tomcat、 jboss都会根据j2ee规范自行实现ClassLoader

-verbose:class

```
    //  System.out.println(ClassB.class.getClassLoader());
    //   System.out.println(Class.forName("org.example.ClassB"));
    //  System.out.println(ClassB.str);
//        Class<?> clazz = Class.forName(ClassB.class.getName());
//        ClassLoader classLoader=clazz.getClassLoader();
//        System.out.println("类加载器是"+classLoader.getClass().getSimpleName());

//        Class<?> clazz =
Class.forName(com.sun.nio.zipfs.ZipPath.class.getName());
//        ClassLoader classLoader=clazz.getClassLoader();
//        System.out.println("类加载器是"+classLoader.getClass().getSimpleName());
//
System.out.println(classLoader.getParent().getClass().getSimpleName());


//        String s1 = new String("he") + new String("llo");
//        String s2 = new String("h") + new String("ello");
//        String s3 = s1.intern();
//        String s4 = s2.intern();
//        System.out.println(s1 == s3);
//        System.out.println(s1 == s4);
```

1.3.2.5 类加载机制

1. 全盘负责：所谓全盘负责，就是当一个类加载器负责加载某个Class时，该Class所依赖和引用其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入。
2. 双亲委派：所谓的双亲委派，则是先让父类加载器试图加载该Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父加载器，依次递归，如果父加载器可以完成类加载任务，就成功返回；只有父加载器无法完成此加载任务时，才自己去加载。
3. 缓存机制。缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区中搜寻该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓冲区中。这就是为很么修改了Class后，必须重新启动JVM，程序所做的修改才会生效的原因。

1.3.2.5 双亲委派

类加载的时候，会优先递归给父类加载器去加载，父类加载器没有加载的时候才会往下的加载器去进行加载

目的：为了安全，保证核心类库的安全性，防止被篡改，以及保证类的唯一

ClassNotFoundException（加载时找不到类）； NotClassFoundException（编译时找不到类）

怎么打破双亲委派 重写classLoad findClass接口

1.3.3 链接(link)

1.3.3.1 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

验证文件格式、元数据、字节码等等

文件格式验证：此阶段保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个 Java 类型信息的要求。

（1）是否以魔数 0xCAFEBABE开头

（2）主、次版本号是否在当前虚拟机处理范围之内 。

（3）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)。

（4）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装型的常量 。

（5）CONSTANT_Utf8_info型的常量中是否有不符合 UTF8编码的数据

（6）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

元数据验证：保证不存在不符合 Java 语言规范的元数据信息。

（1）这个类是否有父类(除了 java.lang.0bject之外,所有的类都应当有父类)

（2）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)

（3）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

（4）类中的字段、 方法是否与父类产生了矛盾(例如覆盖了父类的final字段, 或者出現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：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。例如保证跳转指令不会跳转到方法体以外的字节码指令上。

符号引用验证：在解析阶段中发生，保证可以将符号引用转化为直接引用。

可以考虑使用 `-Xverify:none` 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

1.3.3.2 准备

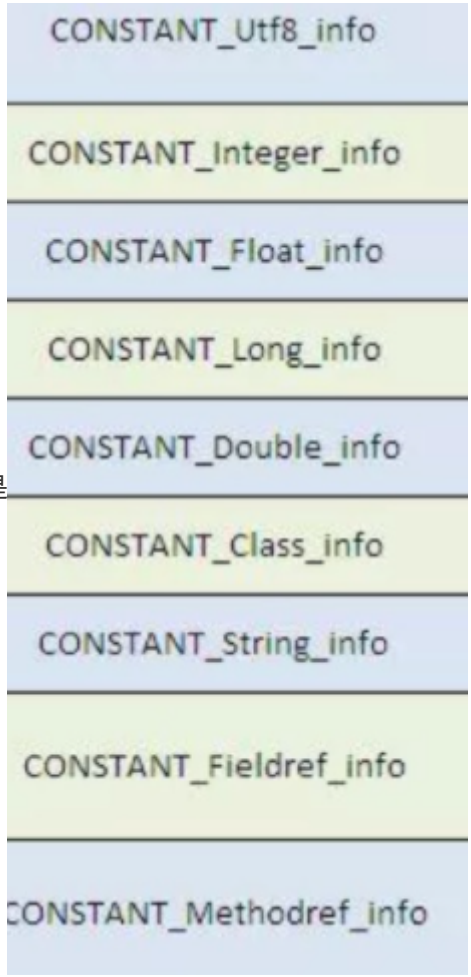
为静态变量分配内存，赋值初始值

1.3.3.3 解析

将常量池中的符号引用转为直接引用

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 7 类符号引用进行。

符号引用：就是



直接引用就是真实地址

1.3.4 初始化（Initialize）

为静态变量赋真正的值。

到初始化阶段，才真正开始执行类中定义的 Java 程序代码，此阶段是执行 `<clinit>()` 方法的过程。

`<init>()` 方法是由编译器按语句在源文件中出现的顺序，依次自动收集类中的所有**类变量**的赋值动作和静态代码块中的语句合并产生的。（不包括构造器中的语句。构造器是初始化对象的，类加载完成后，创建对象时候将调用的 `<init>()` 方法来初始化对象）

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，如下程序：

```
public class Test {
    static {
        // 给变量赋值可以正常编译通过
        i = 0;
        // 这句编译器会提示"非法向前引用"
        System.out.println(i);
    }

    static int i = 1;
}
```

`<clinit>()` 不需要显式调用父类（接口除外，接口不需要调用父接口的初始化方法，只有使用到父接口中的静态变量时才需要调用）的初始化方法 `<clinit>()`，虚拟机会保证在子类的 `<clinit>()` 方法执行之前，父类的 `<clinit>()` 方法已经执行完毕，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。

`<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>()` 方法。

虚拟机会保证一个类的 `<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 `<clinit>()` 方法完毕。

5种类必须类初始化情况：

- 在遇到 new、putstatic、getstatic、invokestatic 字节码指令时，如果类尚未初始化，则需要先触发初始化。
- 对类进行反射调用时，如果类还没有初始化，则需要先触发初始化。
- 初始化一个类时，如果其父类还没有初始化，则需要先初始化父类。
- 虚拟机启动时，用于需要指定一个包含 main() 方法的主类，虚拟机会先初始化这个主类。
- 当使用 JDK 1.7 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 的方法句柄，并且这个方法句柄所对应的类还没初始化，则需要先触发初始化。

所以，我们发现，JVM的加载肯定是lazy loading，也就是我们经常讲的懒加载，用到谁加载谁，用不到就不加载

1.3.5 使用

实例化

1.3.6 卸载

Class被回收要满足以下三个条件:

a) No Instance：该类所有的实例都已经被GC;

b) No ClassLoader：加载该类的ClassLoader实例已经被GC;

c) No Reference：该类的java.lang.Class对象没有被引用。(XXX.class, 静态变量/方法)

1.4 JVM内存模型

JVM内存模型就是我刚刚加载的class文件数据我要加载到JVM，那么这些数据我放在JVM哪里

1.4.1 方法区(Method Area)

方法区在JDK8中的实现是MeatSpace元空间 之前就是Perm Space 永久代

方法区是各个线程共享的内存区域，在虚拟机启动时创建。

Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译时期生成的各种字面量和符号引用，这部分内容将在 类加载后进入方法区的运行时常量池中存放。

但是有很多人说字符串常量池已经放到了heap里面，但是官网文档还是没有介绍的，我们先过

1.4.2 Heap(堆)

Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。Java对象实例以及数组都在堆上分配，被所有线程共享.JAVA对象实例以及数组都在堆上分配.

1.4.3 Java Virtual Machine Stacks(虚拟机栈)

虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出

1.4.4 The pc Register（程序计数器）

程序计数器占用的内存空间很小，由于Java虚拟机的多线程是通过线程轮流切换，并分配处理器执行时间的方式来实现的，在任意时刻，一个处理器只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程需要有一个独立的程序计数器(线程私有)。如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址；

1.4.5 Native Method Stacks(本地方法栈)

如果当前线程执行的方法是Native类型的，这些方法就会在本地方法栈中执行。

1.5 面试题

详解JVM内存模型？有哪些是线程共享，哪些是线程私有

程序计数器：当前线程所执行的字节码的行号指示器，用于记录正在执行的虚拟机字节指令地址，线程私有。

Java虚拟栈：存放基本数据类型、对象的引用、方法出口等，线程私有。

Native方法栈：和虚拟栈相似，只不过它服务于Native方法，线程私有。

Java堆：java内存最大的一块，所有对象实例、数组都存放在java堆，GC回收的地方，线程共享。

方法区：存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码数据等。（即永久代），回收目标主要是常量池的回收和类型的卸载，各线程共享

简单说下类加载器，什么是双亲委派，怎么打破

引导类加载器、拓展类加载器、应用程序（系统）类加载器这个是java实现的3个加载器，其中引导类由c++语言实现，负责 jre/lib/rt.jar，扩展类跟应用程序由java语言实现，继承自抽象类ClassLoader

展类加载器（Extension ClassLoader）：负责加载<JAVA_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。
应用程序类加载器（Application ClassLoader）。负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派就是每个类加载器收到类加载请求的时候，首先不会尝试加载这个类，而是给到父类加载器去加载，当父类加载器没有然后递归给子类加载！！

目的是为了保证类的唯一性，以及系统类的安全，不被篡改

打破双亲委派机制则继承ClassLoader类，重写loadClass和findClass方法。

描述下JAVA class类的生命周期

1.6 补充

1.6.1 invoke

具体来说,Java字节码中与调用相关的指令共有五种:

1.invokestatic:用来调用静态方法，即使用 static 关键字修饰的方法。 它要调用的方法在编译期间确定，运行期不会修改，属于静态绑定。它也是所有方法调用指令里面最快的。

例如：

```
String.valueOf(123)
```

2.invokespecial:它是特殊的方法调用，包括实例构造方法、私有方法（private 修饰的方法）和父类方法（即 super 关键字调用的方法）。很明显明显，这些特殊的方法可以直接确定实际执行的方法的实现，与 invokestatic 一样，也属于静态绑定。

3.invokevirtual:用来调用 public、protected、package 访问级别的方法。方法要根据对象类型不同动态选择不同的类，在编译期不确定，属于动态绑定。

例如：

```
public class Car {
    public void printCarName()  {
        System.out.println("Car name from parent");
    }
}
public class Audi extends Car {
    @Override
    public void printCarName() {
        System.out.println("Car name is Audi");
    }
}
public class BMW extends Car {
    @Override
    public void printCarName() {
```

```
        System.out.println("Car name is BMW");
    }
}
public class InvokeVirtualTest {
    private static Car audiCar = new Audi ();
    private static Car bmwCar= new BMW ();
    public static void main(String[] args) {
        audiCar.printCarName();
        bmwCar.printCarName();
    }
}

0: getstatic     #2                  // Field audiCar :LCar;
3: invokevirtual #3                  // Method Car.printCarName:()V
6: getstatic     #4                  // Field bmwCar:LCar;
9: invokevirtual #3                  // Method Car.printCarName:()V
```

invokevirtual会根据对象的实际类型进行分派（虚方法分派），在编译期间不能确定最终会调用子类还是父类的方法。

4.invokeinterface:用于调用接口方法，在运行时再确定一个实现此接口的对象，也属于动态绑定。

5.invokedynamic:**出现于JDK8中的lambda表达式中，是一种调用方法的新方式，它用来告诉 JVM 可以延迟确认最终要调用的哪个方法。一开始 invokedynamic 并不知道要调用什么目标方法。第一次调用时引导方法（Bootstrap Method）会被调用，由这个引导方法决定哪个目标方法进行调用。

在Java中，普通的方法调用的类名、方法名、入参类型、返回类型都是确定的：

```
audiCar.printCarName("audi car");
```

但Java还可以通过方法句柄 MethodHandle 来调用方法：

```
public class Foo {
    public void printCarName(String s) {
        System.out.println(s);
    }
    public static void main(String[] args) throws Throwable {
        Foo foo = new Foo();
        MethodType methodType = MethodType.methodType(void.class, String.class);
        MethodHandle methodHandle =
MethodHandles.lookup().findVirtual(Foo.class, "printCarName", methodType);
        methodHandle.invokeExact(foo, "audi car");
    }
}

运行输出
audi car
```

1.6.2 字节码指令

```
指令码 助记符     说明
0x00 nop         无操作
0x01 aconst_null 将null推送至栈顶
0x02 iconst_m1    将int型-1推送至栈顶
```

```
0x03 iconst_0    将int型0推送至栈顶
0x04 iconst_1    将int型1推送至栈顶
0x05 iconst_2    将int型2推送至栈顶
0x06 iconst_3    将int型3推送至栈顶
0x07 iconst_4    将int型4推送至栈顶
0x08 iconst_5    将int型5推送至栈顶
0x09 lconst_0    将long型0推送至栈顶
0x0a lconst_1    将long型1推送至栈顶
0x0b fconst_0    将float型0推送至栈顶
0x0c fconst_1    将float型1推送至栈顶
0x0d fconst_2    将float型2推送至栈顶
0x0e dconst_0    将double型0推送至栈顶
0x0f dconst_1    将double型1推送至栈顶
0x10 bipush     将单字节的常量值(-128~127)推送至栈顶
0x11 sipush     将一个短整型常量值(-32768~32767)推送至栈顶
0x12 ldc    将int, float或String型常量值从常量池中推送至栈顶
0x13 ldc_w    将int, float或String型常量值从常量池中推送至栈顶（宽索引）
0x14 ldc2_w     将long或double型常量值从常量池中推送至栈顶（宽索引）
0x15 iload    将指定的int型本地变量推送至栈顶
0x16 lload    将指定的long型本地变量推送至栈顶
0x17 fload    将指定的float型本地变量推送至栈顶
0x18 dload    将指定的double型本地变量推送至栈顶
0x19 aload    将指定的引用类型本地变量推送至栈顶
0x1a iload_0    将第一个int型本地变量推送至栈顶
0x1b iload_1    将第二个int型本地变量推送至栈顶
0x1c iload_2    将第三个int型本地变量推送至栈顶
0x1d iload_3    将第四个int型本地变量推送至栈顶
0x1e lload_0    将第一个long型本地变量推送至栈顶
0x1f lload_1    将第二个long型本地变量推送至栈顶
0x20 lload_2    将第三个long型本地变量推送至栈顶
0x21 lload_3    将第四个long型本地变量推送至栈顶
0x22 fload_0    将第一个float型本地变量推送至栈顶
0x23 fload_1    将第二个float型本地变量推送至栈顶
0x24 fload_2    将第三个float型本地变量推送至栈顶
0x25 fload_3    将第四个float型本地变量推送至栈顶
0x26 dload_0    将第一个double型本地变量推送至栈顶
0x27 dload_1    将第二个double型本地变量推送至栈顶
0x28 dload_2    将第三个double型本地变量推送至栈顶
0x29 dload_3    将第四个double型本地变量推送至栈顶
0x2a aload_0    将第一个引用类型本地变量推送至栈顶
0x2b aload_1    将第二个引用类型本地变量推送至栈顶
0x2c aload_2    将第三个引用类型本地变量推送至栈顶
0x2d aload_3    将第四个引用类型本地变量推送至栈顶
0x2e iaload    将int型数组指定索引的值推送至栈顶
0x2f laload    将long型数组指定索引的值推送至栈顶
0x30 faload    将float型数组指定索引的值推送至栈顶
0x31 daload    将double型数组指定索引的值推送至栈顶
0x32 aaload    将引用型数组指定索引的值推送至栈顶
0x33 baload    将boolean或byte型数组指定索引的值推送至栈顶
0x34 caload    将char型数组指定索引的值推送至栈顶
0x35 saload    将short型数组指定索引的值推送至栈顶
0x36 istore    将栈顶int型数值存入指定本地变量
0x37 lstore    将栈顶long型数值存入指定本地变量
0x38 fstore    将栈顶float型数值存入指定本地变量
0x39 dstore    将栈顶double型数值存入指定本地变量
0x3a astore    将栈顶引用型数值存入指定本地变量
0x3b istore_0    将栈顶int型数值存入第一个本地变量
0x3c istore_1    将栈顶int型数值存入第二个本地变量
```

0x3d istore_2 将栈顶int型数值存入第三个本地变量
0x3e istore_3 将栈顶int型数值存入第四个本地变量
0x3f lstore_0 将栈顶long型数值存入第一个本地变量
0x40 lstore_1 将栈顶long型数值存入第二个本地变量
0x41 lstore_2 将栈顶long型数值存入第三个本地变量
0x42 lstore_3 将栈顶long型数值存入第四个本地变量
0x43 fstore_0 将栈顶float型数值存入第一个本地变量
0x44 fstore_1 将栈顶float型数值存入第二个本地变量
0x45 fstore_2 将栈顶float型数值存入第三个本地变量
0x46 fstore_3 将栈顶float型数值存入第四个本地变量
0x47 dstore_0 将栈顶double型数值存入第一个本地变量
0x48 dstore_1 将栈顶double型数值存入第二个本地变量
0x49 dstore_2 将栈顶double型数值存入第三个本地变量
0x4a dstore_3 将栈顶double型数值存入第四个本地变量
0x4b astore_0 将栈顶引用型数值存入第一个本地变量
0x4c astore_1 将栈顶引用型数值存入第二个本地变量
0x4d astore_2 将栈顶引用型数值存入第三个本地变量
0x4e astore_3 将栈顶引用型数值存入第四个本地变量
0x4f iastore 将栈顶int型数值存入指定数组的指定索引位置
0x50 lastore 将栈顶long型数值存入指定数组的指定索引位置
0x51 fastore 将栈顶float型数值存入指定数组的指定索引位置
0x52 dastore 将栈顶double型数值存入指定数组的指定索引位置
0x53 aastore 将栈顶引用型数值存入指定数组的指定索引位置
0x54 bastore 将栈顶boolean或byte型数值存入指定数组的指定索引位置
0x55 castore 将栈顶char型数值存入指定数组的指定索引位置
0x56 sastore 将栈顶short型数值存入指定数组的指定索引位置
0x57 pop 将栈顶数值弹出 (数值不能是long或double类型的)
0x58 pop2 将栈顶的一个（long或double类型的)或两个数值弹出（其它）
0x59 dup 复制栈顶数值并将复制值压入栈顶
0x5a dup_x1 复制栈顶数值并将两个复制值压入栈顶
0x5b dup_x2 复制栈顶数值并将三个（或两个）复制值压入栈顶
0x5c dup2 复制栈顶一个（long或double类型的)或两个（其它）数值并将复制值压入栈顶
0x5d dup2_x1 复制栈顶的一个或两个值，将其插入栈顶那两个或三个值的下面
0x5e dup2_x2 复制栈顶的一个或两个值，将其插入栈顶那两个、三个或四个值的下面
0x5f swap 将栈最顶端的两个数值互换(数值不能是long或double类型的)
0x60 iadd 将栈顶两int型数值相加并将结果压入栈顶
0x61 ladd 将栈顶两long型数值相加并将结果压入栈顶
0x62 fadd 将栈顶两float型数值相加并将结果压入栈顶
0x63 dadd 将栈顶两double型数值相加并将结果压入栈顶
0x64 isub 将栈顶两int型数值相减并将结果压入栈顶
0x65 lsub 将栈顶两long型数值相减并将结果压入栈顶
0x66 fsub 将栈顶两float型数值相减并将结果压入栈顶
0x67 dsub 将栈顶两double型数值相减并将结果压入栈顶
0x68 imul 将栈顶两int型数值相乘并将结果压入栈顶
0x69 lmul 将栈顶两long型数值相乘并将结果压入栈顶
0x6a fmul 将栈顶两float型数值相乘并将结果压入栈顶
0x6b dmul 将栈顶两double型数值相乘并将结果压入栈顶
0x6c idiv 将栈顶两int型数值相除并将结果压入栈顶
0x6d ldiv 将栈顶两long型数值相除并将结果压入栈顶
0x6e fdiv 将栈顶两float型数值相除并将结果压入栈顶
0x6f ddiv 将栈顶两double型数值相除并将结果压入栈顶
0x70 irem 将栈顶两int型数值作取模运算并将结果压入栈顶
0x71 lrem 将栈顶两long型数值作取模运算并将结果压入栈顶
0x72 frem 将栈顶两float型数值作取模运算并将结果压入栈顶
0x73 drem 将栈顶两double型数值作取模运算并将结果压入栈顶
0x74 ineg 将栈顶int型数值取负并将结果压入栈顶
0x75 lneg 将栈顶long型数值取负并将结果压入栈顶
0x76 fneg 将栈顶float型数值取负并将结果压入栈顶

0x77 dneg 将栈顶double型数值取负并将结果压入栈顶
0x78 ishl 将int型数值左移位指定位数并将结果压入栈顶
0x79 lshl 将long型数值左移位指定位数并将结果压入栈顶
0x7a ishr 将int型数值右（符号）移位指定位数并将结果压入栈顶
0x7b lshr 将long型数值右（符号）移位指定位数并将结果压入栈顶
0x7c iushr 将int型数值右（无符号）移位指定位数并将结果压入栈顶
0x7d lushr 将long型数值右（无符号）移位指定位数并将结果压入栈顶
0x7e iand 将栈顶两int型数值作“按位与”并将结果压入栈顶
0x7f land 将栈顶两long型数值作“按位与”并将结果压入栈顶
0x80 ior 将栈顶两int型数值作“按位或”并将结果压入栈顶
0x81 lor 将栈顶两long型数值作“按位或”并将结果压入栈顶
0x82 ixor 将栈顶两int型数值作“按位异或”并将结果压入栈顶
0x83 lxor 将栈顶两long型数值作“按位异或”并将结果压入栈顶
0x84 iinc 将指定int型变量增加指定值（i++, i--, i+=2）
0x85 i2l 将栈顶int型数值强制转换成long型数值并将结果压入栈顶
0x86 i2f 将栈顶int型数值强制转换成float型数值并将结果压入栈顶
0x87 i2d 将栈顶int型数值强制转换成double型数值并将结果压入栈顶
0x88 l2i 将栈顶long型数值强制转换成int型数值并将结果压入栈顶
0x89 l2f 将栈顶long型数值强制转换成float型数值并将结果压入栈顶
0x8a l2d 将栈顶long型数值强制转换成double型数值并将结果压入栈顶
0x8b f2i 将栈顶float型数值强制转换成int型数值并将结果压入栈顶
0x8c f2l 将栈顶float型数值强制转换成long型数值并将结果压入栈顶
0x8d f2d 将栈顶float型数值强制转换成double型数值并将结果压入栈顶
0x8e d2i 将栈顶double型数值强制转换成int型数值并将结果压入栈顶
0x8f d2l 将栈顶double型数值强制转换成long型数值并将结果压入栈顶
0x90 d2f 将栈顶double型数值强制转换成float型数值并将结果压入栈顶
0x91 i2b 将栈顶int型数值强制转换成byte型数值并将结果压入栈顶
0x92 i2c 将栈顶int型数值强制转换成char型数值并将结果压入栈顶
0x93 i2s 将栈顶int型数值强制转换成short型数值并将结果压入栈顶
0x94 lcmp 比较栈顶两long型数值大小，并将结果（1，0，-1）压入栈顶
0x95 fcmpl 比较栈顶两float型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为NaN时，将-1压入栈顶
0x96 fcmpg 比较栈顶两float型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为NaN时，将1压入栈顶
0x97 dcmpl 比较栈顶两double型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为NaN时，将-1压入栈顶
0x98 dcmpg 比较栈顶两double型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为NaN时，将1压入栈顶
0x99 ifeq 当栈顶int型数值等于0时跳转
0x9a ifne 当栈顶int型数值不等于0时跳转
0x9b iflt 当栈顶int型数值小于0时跳转
0x9c ifge 当栈顶int型数值大于等于0时跳转
0x9d ifgt 当栈顶int型数值大于0时跳转
0x9e ifle 当栈顶int型数值小于等于0时跳转
0x9f if_icmpeq 比较栈顶两int型数值大小，当结果等于0时跳转
0xa0 if_icmpne 比较栈顶两int型数值大小，当结果不等于0时跳转
0xa1 if_icmplt 比较栈顶两int型数值大小，当结果小于0时跳转
0xa2 if_icmpge 比较栈顶两int型数值大小，当结果大于等于0时跳转
0xa3 if_icmpgt 比较栈顶两int型数值大小，当结果大于0时跳转
0xa4 if_icmple 比较栈顶两int型数值大小，当结果小于等于0时跳转
0xa5 if_acmpeq 比较栈顶两引用型数值，当结果相等时跳转
0xa6 if_acmpne 比较栈顶两引用型数值，当结果不相等时跳转
0xa7 goto 无条件跳转
0xa8 jsr 跳转至指定16位offset位置，并将jsr下一条指令地址压入栈顶
0xa9 ret 返回至本地变量指定的index的指令位置（一般与jsr，jsr_w联合使用）
0xaa tableswitch 用于switch条件跳转，case值连续（可变长度指令）
0xab lookupswitch 用于switch条件跳转，case值不连续（可变长度指令）
0xac ireturn 从当前方法返回int

0xad lreturn 从当前方法返回long
0xae freturn 从当前方法返回float
0xaf dreturn 从当前方法返回double
0xb0 areturn 从当前方法返回对象引用
0xb1 return 从当前方法返回void
0xb2 getstatic 获取指定类的静态域，并将其值压入栈顶
0xb3 putstatic 为指定的类的静态域赋值
0xb4 getfield 获取指定类的实例域，并将其值压入栈顶
0xb5 putfield 为指定的类的实例域赋值
0xb6 invokevirtual 调用实例方法
0xb7 invokespecial 调用超类构造方法，实例初始化方法，私有方法
0xb8 invokestatic 调用静态方法
0xb9 invokeinterface 调用接口方法
0xba invokedynamic 调用动态链接方法
0xbb new 创建一个对象，并将其引用值压入栈顶
0xbc newarray 创建一个指定原始类型（如int, float, char…）的数组，并将其引用值压入栈顶
0xbd anewarray 创建一个引用型（如类，接口，数组）的数组，并将其引用值压入栈顶
0xbe arraylength 获得数组的长度值并压入栈顶
0xbf athrow 将栈顶的异常抛出
0xc0 checkcast 检验类型转换，检验未通过将抛出ClassCastException
0xc1 instanceof 检验对象是否是指定的类的实例，如果是将1压入栈顶，否则将0压入栈顶
0xc2 monitorenter 获得对象的锁，用于同步方法或同步块
0xc3 monitorexit 释放对象的锁，用于同步方法或同步块
0xc4 wide 扩大本地变量索引的宽度
0xc5 multianewarray 创建指定类型和指定维度的多维数组（执行该指令时，操作栈中必须包含各维度的长度值），并将其引用值压入栈顶
0xc6 ifnull 为null时跳转
0xc7 ifnonnull 不为null时跳转
0xc8 goto_w 无条件跳转
0xc9 jsr_w 跳转至指定32位offset位置，并将jsr_w下一条指令地址压入栈顶
==========
0xca breakpoint 调试时的断点标记
0xfe impdep1 为特定软件而预留的语言后门
0xff impdep2 为特定硬件而预留的语言后门
最后三个为保留指令