


转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81672834

介绍JVM类加载机制相关的内容，包括7大阶段，双亲委派模型等内容。

JVM类加载机制(65题)

版本：2018/8/17-1(7:18)

 类加载器-思维导图

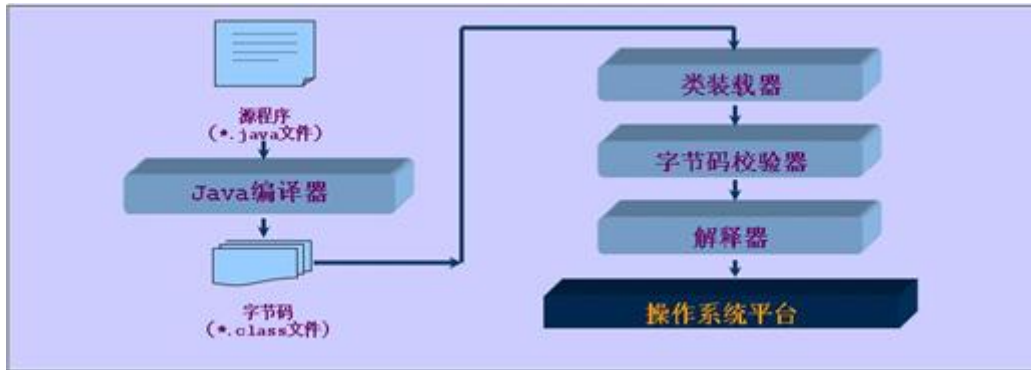
- JVM类加载机制(65题)
 - 基础(13题)
 - 基本类型
 - 引用类型
 - 字节流
 - 类
 - 类加载的阶段(10题)
 - 加载(22题)
 - 加载时机
 - 类加载器
 - 启动类加载器
 - 扩展类加载器
 - 应用类加载器
 - 自定义加载器
 - 双亲委派
 - loadClass源码
 - 链接(12题)
 - 验证
 - 纠错
 - 准备
 - 解析
 - 初始化(8)
 - 单例延迟初始化
 - 总结
 - 参考资料

基础(13题)

1、什么是类加载机制

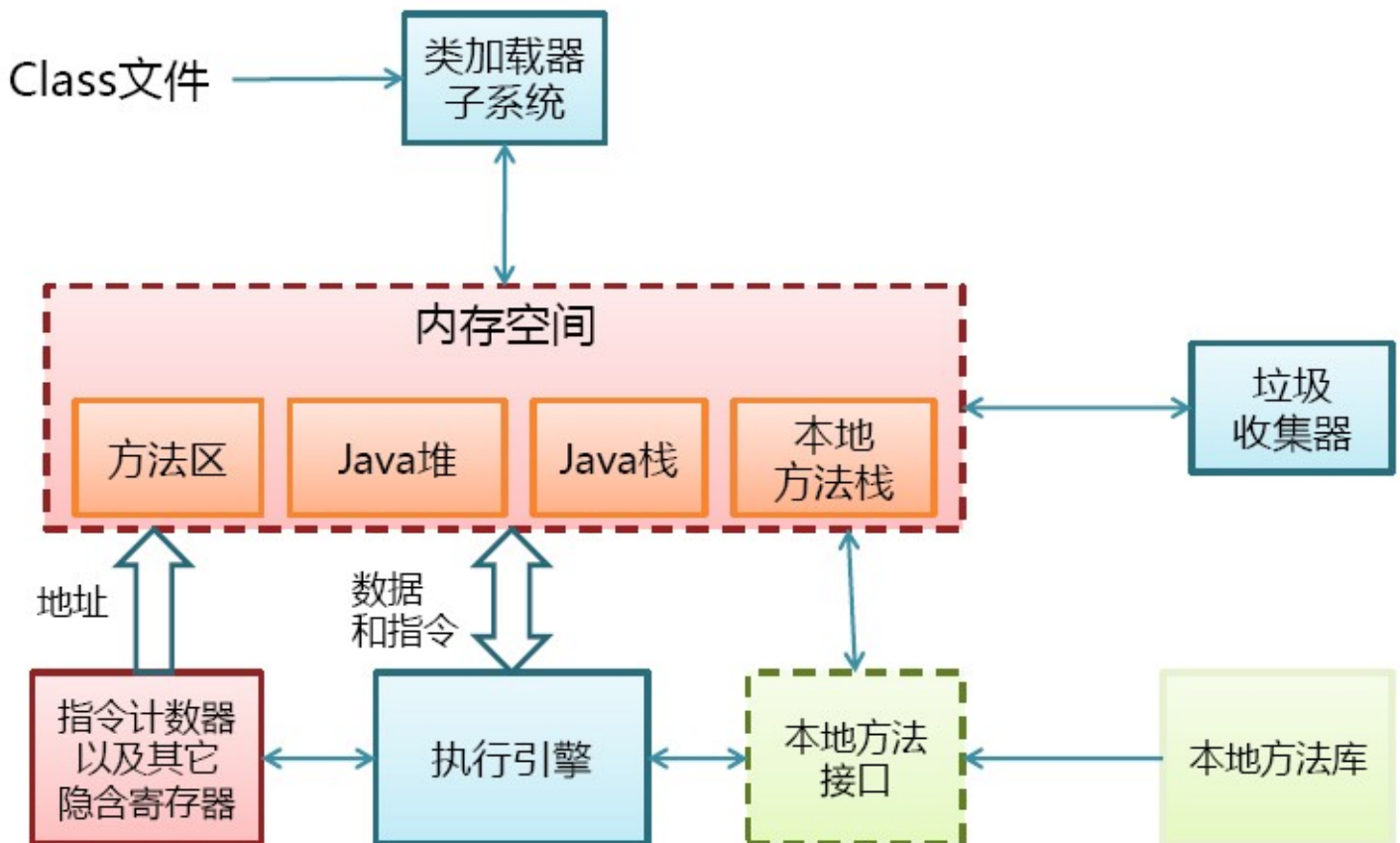
1. 虚拟机把描述类的数据从Class文件加载到内存
2. 并对数据校验、转换解析和初始化
3. 最终形成可以被虚拟机直接使用的Java类型。

2、Java程序的执行流程



1. java文件(源程序)会被Java编译器进行处理。
2. 会生成.class文件(字节码)
3. .class文件通过类装载器进行加载。

3、JVM大致的物理结构图



4、Java语言的类型分类

1. 基本类型
2. 引用类型

基本类型

5、Java的基本类型是由JVM预先定义好的

引用类型

6、Java的引用类型细分为四种

1. 类
2. 接口
3. 数组类
4. 泛型参数

7、数组类是什么？

1. 定义的数组本质上也是 `Object` 类
2. 也包含了`equals`、`toString`等继承自 `Object`类 的方法

```
int[] a;  
a.toString();
```

8、JVM实际上有哪些引用类型

只有三种：

1. 类: 具有对应的字节流
 2. 接口：具有对应的字节流
 3. 数组类： 由JVM直接生成。
- 泛型参数 会在编译的过程中被擦除。

9、数组类不需要被加载，会由JVM直接生成。

10、无论是直接生成的数组类，还是加载的类，都需要JVM对其进行链接和初始化。

字节流

11、字节流的常见形式有哪些？

1. Java编译器生成的class文件
2. 程序内部可以直接生成
3. 可以从网络中获取(例如：网页中内嵌的小程序Java applet)

12、字节流有什么用？

1. 无论是任何形式的字节流，最终都会被加载到JVM中，成为类或者接口。

类

13、类的唯一性是如何确定的？

1. 由 类加载器实例 和 类的全名 两者一同确定的。
2. 即使是同一串字节流，经由不同的类加载器加载，也会得到两个不同的类。
3. 大型项目中可以借助该特性，来运行同一个类的不同版本(同时使用两个版本需要分开编译)
4. 但是应用开发者写代码时，是无法区分的。

类加载的阶段(10题)

1、Class文件经过类装载机加载后，如何让我们能调用Class对象的功能？

1. Class文件经过类装载机加载后，JVM中会生成一份 描述Class结构 的元信息对象。
2. 通过元信息对象可以获知Class的结构信息：构造函数，属性，方法等。
3. Java允许用户借助元信息对象间接调用Class对象的功能。(就是常见的Class类)

2、类从加载到内存，到从内存中卸载所需要经历的生命周期(7个小阶段)



1. 七个阶段：加载、验证、准备、解析、初始化、使用、卸载。
2. 验证、准备、解析这三个阶段统称为链接 Linking

3、类加载的三大步骤

1. 加载
2. 链接
3. 初始化

4、类装载机是什么？

就是寻找类的字节码文件，并构造类在JVM内部表示的对象组件。

5、加载的作用？

查找和导入Class文件：

6、链接的作用？

把类的二进制数据合入到JVM中，且使之能够执行：

1. 校验：检查载入的Class文件数据的完整性缺陷。
2. 准备：给类的静态变量分配存储空间。
3. 解析：将符号引用转成直接引用

7、初始化的作用？

对类的静态变量，静态代码块执行初始化操作。

8、Java为何能够动态扩展？

1. 能够在 运行时期动态加载 和 动态链接
2. 可以在运行时在指定其实际的实现：多态
3. 解析过程有时候还可以在初始化之后执行：动态绑定(多态)

9、什么是动态绑定(多态)

1. 例如对象具有两个方法，方法名一致，根据参数是String还是int去判断调用哪个方法。

10、类加载的7个阶段的顺序。

1. 加载、验证、准备、初始化、使用、卸载顺序是一致的。
2. 解析 的顺序并不固定，某些情况下可以再初始化阶段后再开始。
3. 类的生命周期的每一个阶段通常都是交叉混合进行的，会在一个阶段执行过程中，就激活了另一个阶段。

加载(22题)

1、加载的作用？

查找和导入Class文件：

1. 查找字节流
2. 根据字节流来创建类(类和接口)。

2、数组类和一般类的区别？

1. 数组类是JVM直接生成，因此不需要 加载
2. 其他类，JVM需要借助 类加载器 来完成查找字节流的工作。

3、类加载的过程？

1. 将类的class文件中的二进制数据读取到内存中
2. 会在 堆区 创建一个java.lang.Class对象。
3. 类的加载最终生成的是位于堆区中的Class对象。Class对象封装了类在方法区内的数据结构。
4. 该Class对象向程序员提供了访问方法区内的数据结构的接口。

加载时机

4、类加载的时机？

1. 类加载器并不是在某个类被首次主动使用时才加载，会在预料到要被使用时进行预先加载。
2. 如果预先加载时，遇到了错误。如果用户之后没有用到该类就不会报错，如果用到了在首次使用时才进行报错。

5、加载.class文件的方式有？(5)

1. 从本地系统中直接加载。
2. 从网络下载.class文件
3. 从zip,jar等文件中加载.class文件
4. 从专有数据库中加载.class文件
5. 将java源文件动态编译为.class文件(动态代理)

类加载器

6、类加载器的作用

1. 加载类：通过内置和自定义的类加载器，将class文件加载到内存中(二进制字节流)，并依次创建类。
2. 提供命名空间的作用。

7、JVM提供的类加载器

加载动作被放到了JVM外部实现，以便让app决定如何获取所需的类。

1. 启动类加载器(Bootstrap ClassLoader)：负责加载 `JAVA_HOME\lib` 目录中的，或通过 `-Xbootclasspath` 参数指定路径中的，且被虚拟机认可（按文件名识别，如 `rt.jar`）的类。
2. 扩展类加载器(Extension ClassLoader)：负责加载 `JAVA_HOME\lib\ext` 目录中的，或通过 `java.ext.dirs` 系统变量指定路径中的类库。
3. 应用程序类加载器(Application ClassLoader)：负责加载用户路径（`classpath`）上的类库。



类加载器

启动类加载器

8、启动类加载器的实现？

1. 是通过 `C++` 实现的
2. 因此没有对应的Java对象(Java中均用 `null` 来代替、)

9、启动类加载器可以访问吗？

不可以：

1. 没有对应的联系方式(没有Java对象)

10、除了启动类加载器之外的类加载器的特点？

1. 都是 `java.lang.ClassLoader` 的子类
2. 因此都具有对应的 Java 对象

11、类加载器是如何被加载的？

1. 除了 启动类加载器 之外，其他的类加载器都需要先由一个类加载器(如：启动类加载器)加载到 JVM 中
2. 加载完成后，才能执行加载类的任务。

12、启动类加载器会做些什么？(Java 9 以前)

1. 负责加载最为基础、最为重要的类。(2种路径)
2. 加载如：JRE lib 目录中 jar 包里面所有的类。
3. 加载如：虚拟机参数 `-Xbootclasspath` 指定的类
4. 扩展类加载器、应用类加载器都是由 Java 核心类库 提供。

13、启动类加载器和扩展类加载器在Java 9 之后的变化

1. Java 9 引入模块系统。
2. Java 9 中扩展类加载器被改名为平台类加载器。
3. Java SE 中少数几个关键模块(`java.base`)是由 启动类加载器 进行加载
4. Java SE 中其余的模块交给平台类加载器进行加载。

扩展类加载器

14、扩展类加载器会做些什么？

1. 负责加载相对次要、但是又通用的类(2种路径)
2. 加载比如：存放在 JRE 的 lib/ext 目录下 jar 包中的类
3. 加载比如：系统变量 `java.ext.dirs` 指定的类
4. 父加载器是 启动类加载器

应用类加载器

15、应用里加载器会做什么？

1. 负责加载应用程序路径下的类(3种路径)
2. 如：虚拟机参数 `-cp`、`-classpath` 指定的路径
3. 如：系统变量 `java.class.path` 路径
4. 如：环境变量 `CLASSPATH` 所指定的路径

自定义加载器

16、自定义加载器有什么用？

1. 能实现特殊的加载方式。
2. 比如：可以对class文件进行加密，加载时再利用自定义类加载器进行解密。

双亲委派

17、双亲委派模型是什么？

1. JVM提供了三种类加载器，还有用户自定义的类加载器。这些加载器之间的层次关系被称为 类加载器的双亲委派模型。
2. 该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过 组合（Composition）关系 来实现，而不是通过继承（Inheritance）。

18、双亲委派模型的实现方法？

1. 是通过组合关系来实现的，而不是通过继承。

19、双亲委派模型的处理流程

1. 某个类加载器在接收到 加载类的请求 时，首先将加载任务委托给父类加载器，依次递归，直到启动类加载器。
2. 如果父类加载器可以完成类加载任务，就成功返回
3. 只有父类加载器无法完成此加载任务时，才自己去加载。

20、双亲委派模型的优点？

1. Java类随着它的类加载器一起具备了一种带有优先级的层次关系。
2. 例如rt.jar中的类java.lang.Object，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在任何情况下都是同一个类。
3. 反例：如果没有双亲委派模型，而是各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱。

loadClass源码

21、ClassLoader的loadClass源码解析：


```

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {
    // 1、判断该类是否加载过
    Class<?> clazz = findLoadedClass(className);
    // 2、没有加载过，进行加载。
    if (clazz == null) {
        ClassNotFoundException suppressed = null;
        // 3、交给父类加载器进行加载
        try {
            clazz = parent.loadClass(className, false);
        } catch (ClassNotFoundException e) {
            suppressed = e;
        }
        // 4、父类加载器无法加载，则自身进行加载。
        if (clazz == null) {
            try {
                clazz = findClass(className);
            } catch (ClassNotFoundException e) {
                e.addSuppressed(suppressed);
                throw e;
            }
        }
    }

    return clazz;
}

```

22、ClassLoader的findClass源码解析：

```

protected Class<?> findClass(String className) throws ClassNotFoundException {
    throw new ClassNotFoundException(className);
}

```

是一个空实现，因此留给开发者去实现自己的类加载器。

链接(12题)

1、链接的作用？

1. 链接，是指将创建的类合并到JVM中，并使之能够执行的过程。
2. 分为：验证、准备、解析三个阶段。

验证

2、验证的作用

1. 目的在于，确保被加载类能够满足JVM的约束条件
2. 通常，Java编译器生成的类文件必然满足JVM的约束条件。

3. (这部分涉及到字节码注入等内容)

纠错

3、验证的作用：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？

不准确！

1. 进入到验证阶段时，类加载器已经将class文件中的字节流加载为对应的类，应该是确保被加载类中的信息。但是因为类也是由Class文件字节流加载而成，所以说不够准确。且类加载器也会影响生成类的内容。

准备

4、准备阶段的作用

1. 为类的静态变量分配内存，并且设置类静态变量的初始值。
2. JVM会构造其他跟类层次相关的数据结构：比如用来实现虚方法的动态绑定的方法表。

5、准备阶段static和static final修饰的变量初始化的区别？

```
public static int v = 8080;
```

2-仅仅是 `static int v` 在准备阶段只会将初始值设置为 0，而进行 8080赋值的 `putstatic` 指令是在程序被编译后，存放于类构造器方法之中。

```
public static final int v = 8080;
```

3- `static final int v` 会在编译阶段给V生成ConstantValue属性，并且在准备阶段根据ConstantValue属性 赋值为8080

解析

6、解析的作用

1. 将常量池中的符号引用替换为直接引用。
2. 如果目标类、目标字段所在类、目标方法所在类还未加载过，解析还会触发这个类的加载(但是未必会触发这个类的链接和初始化)

7、符号引用的由来？

1. class文件被加载到虚拟机之前，该类无法知道其他类以及其方法、字段所对应的具体地址
2. 甚至不知道自己方法、字段的地址。
3. 因此每当需要引用这些内容时，java编译器会生成一个符号引用来代替。

8、符号引用的特点

1. 当一个类被加载时，该类所用到的别的类的符号引用都会保存在常量池中。
2. 符号应用的目标不一定已经在内存中。
3. 运行阶段，该符号引用一定能够武器以德定位到具体目标上。

```
//例如下列常量
CONSTANT_Class_info
CONSTANT_Field_info
CONSTANT_Method_info
```

9、符号引用的实例

1. 对于一个方法调用，编译器会生成一个符号引用：包含了目标方法所在类的名字、目标方法的名字、接收参数类型、返回值类型
2. 用该符号引用就能代指所要调用的方法

10、直接引用

1. 直接引用就是直接指向目标的指针。
2. 在类加载器解析的时候，会通过符号引用去找到那个引用的类的地址，这个类的地址就是 直接引用
3. 引用的目标必定已经存在在内存中。

11、不同内容的解析流程

1. 类或接口的解析：判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。
2. 字段解析：对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束。
3. 类方法解析：对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。
4. 接口方法解析：与类方法解析步骤类似，只是接口不会有父类，因此，只递归向上搜索父接口就行了。

12、JVM规范对于解析的要求

1. 规范并未要求在 链接 过程中要完成 解析
2. 规范仅仅规定了：某些字节码使用了符号引用，那么在执行之前，需要完成对这些符号引用的解析。

初始化(8)

1、Java中如何初始化一个静态字段？

1. 声明时直接赋值
2. 静态代码块中对其赋值

2、直接复制的静态字段用final修饰是如何初始化的？

1. 在类型为 基本类型 或者 字符串 是，会被Java编译器标记为常量值 `ConstantValue`
2. 静态字段会在 准备阶段 根据`ConstantValue`直接进行赋值。

3、其他

1. 除了 `static final int/String = xxx` 以外的静态变量的直接赋值操作，以及静态代码块中的代码，都会被Java编译器放置到同一方法中：`clinit`

4、初始化阶段的作用

1. 为标记为常量值的字段赋值
2. 执行 `clinit` 方法：JVM会通过锁来保证 `clinit` 仅仅执行一次。
3. 初始化完成后，类才正式成为 可执行的状态

5、clinit的注意事项

1. JVM会保证`clinit`方法执行前，父类的`clinit`方法已经执行完毕。
2. 如果一个类中没有静态变量赋值也没有静态代码块，则编译器不会给该类生成`clinit`方法

6、JVM规范枚举了类的初始化会触发的场景

1. JVM启动时，初始化用户指定的主类。
2. new 创建实例时，会初始化目标类。
3. 访问静态字段时，会初始化静态字段所在的类
4. 访问静态方法时，会初始化静态方法所在的类。
5. 子类的初始化会触发父类的初始化。
6. 如果一个接口定义了default方法，那么直接实现或者间接实现该接口的类的初始化，会触发该接口的初始化。
7. 使用反射API对某个类进行反射调用时，初始化这个类
8. 除此调用`MethodHandle`实例是，初始化该`MethodHandle`指向的方法所在的类。

7、哪些场景下不会执行类的初始化阶段？

1. 通过子类引用父类的静态字段：只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译时期会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量的类的初始化。
4. 通过类名获取Class对象，不会触发类的初始化。
5. 通过`Class.forName`加载指定类时，如果指定参数`initialize`为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过ClassLoader默认的`loadClass`方法，也不会触发初始化动作。

单例延迟初始化

8、初始化时机在单例上的运用。

```
public class Singleton {  
    public static Singleton getInstance(){  
        // 1. 调用该方法时，才会访问 LazyHolder.INSTANCE这个静态类的静态变量  
        return LazyHolder.INSTANCE;  
    }  
    private static class LazyHolder{  
        // 2. 才会触发Singleton的初始化  
        static final Singleton INSTANCE = new Singleton();  
    }  
    private Singleton(){}  
}
```

1. 参考自规范的：访问静态字段时，会初始化静态字段所在的类
2. 因为 类初始化 是线程安全的，并且只会执行一次。因此在多线程环境下，依然能保证只有一个Singleton实例。
3. 解释：getInstance()调用了LazyHolder的静态字段INSTANCE，所以会触发 LazyHolder 的加载和初始化，而静态字段INSTANCE的赋值是 新建Singleton实例，而初始化阶段是线程安全且只执行一次，因此就算是多线程，也只会创建一个 Singleton对象 (new Singleton是在LazyHolder的初始化阶段赋值的)

总结

1、类加载的阶段有哪几个是用户可以自定义的

1. 只有加载阶段可以自定义类加载器
2. 其他阶段都是由JVM主导

2、新建数组会初始化元素的类吗？

不会：

1. 新建数组只会加载元素的类
2. 新建数组不会链接、初始化元素的类

3、JVM不会直接使用class文件

4、类加载链接的本质目的？

1. 在JVM中创建相对应的类结构，会保存在元空间(方法)，并且标记为已链接，可以使用。

参考资料

1. 简述类加载机制
2. 深入理解类加载机制
3. JVM 类加载机制详解
4. 【深入理解JVM】：类加载器与双亲委派模型