

Java 进阶篇：理解 Class 和 Class 加载过程

1. 了解 .class

1.1 字节码 .class 来源

我们都知道 Java 是跨平台的，一次编写可以在各个操作系统上运行。而其中的一次编写，就是指：

通过 Java 语法编写 *.java 文件，由编译器产生 *.class。而 *.class 可以发布到各种操作系统上，由各个系统上 JVM 的 Java 运行环境来加载进行运行。

随着开源生态的发展，Java 虚拟机规范《Java Virtual Machine Specification》的制定，*.class 不一定只有 Java 语言和 Java 编译器可以生成，目前市场比较有名的还有 JRuby、Groovy、Jython 等。

语言	其源文件	编译器	字节码	JVM
Java 语言	*.java	javac 编译器	字节码 .class	Java 虚拟机
JRuby 语言	*.rb	jruby 编译器	字节码 .class	Java 虚拟机
Groovy 语言	*.groovy	groovy 编译器	字节码 .class	Java 虚拟机
其它符合标准的语言	.	对应的编译器	字节码 .class	Java 虚拟机

1.2 .class 文件结构和内容

.class 文件是一个 8 位字节的二进制流文件。

简单来说，就是一个字节占用 8 位，即一个字节用 8 位的 0、1 二进制流表示。也就是说，Java 世界里面的二进制流，不管是 16-bit、32-bit、64-bit，都将会被分别转成 2、3、8 个 8 位的字节。

可以用 java.io.DataInput 和 java.io.DataOutput 来处理 8 位字节的二进制流，读、存 classes 文件可以用 java.io.DataInputStream 和 java.io.DataOutputStream。

.class 文件中的每个字节都是紧凑地拼接在一起，中间没有任何分割符号，因此每一个字节都有其一定意义。就像一张表一样具有详细的分割，.class 文件可以看做一个由具有固

定排版的 0、1 组成、每个位置都不能错的二进制流文件。所以加载完成后会有验证机制。

严格的类结构如下：

```
ClassFile {
    u4          magic;
    u2          minor_version;
    u2          major_version;
    u2          constant_pool_count;
    cp_info      constant_pool[constant_pool_count-1];
    u2          access_flags;
    u2          this_class;
    u2          super_class;
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info   fields[fields_count];
    u2          methods_count;
    method_info  methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

根据 Java 虚拟机的规定，class 文件采用的这种类似 C 语言的伪结构来储存，而其中包含了两种数据类型：无符号和表。

u1、u2、u3、u4 等就是无符号数据类型，分别代表了 1 个字节、2 个字节、3 个字节、4 个字节等等以此类推。而其中的 ***_info 就代表了一种表的数据类型，其后的结构详情是个表结构。非常细的表结构我们知道就行了，最重要的是上面的 class 结构。

我们来分别看下每个结构代表什么意思，如下表：

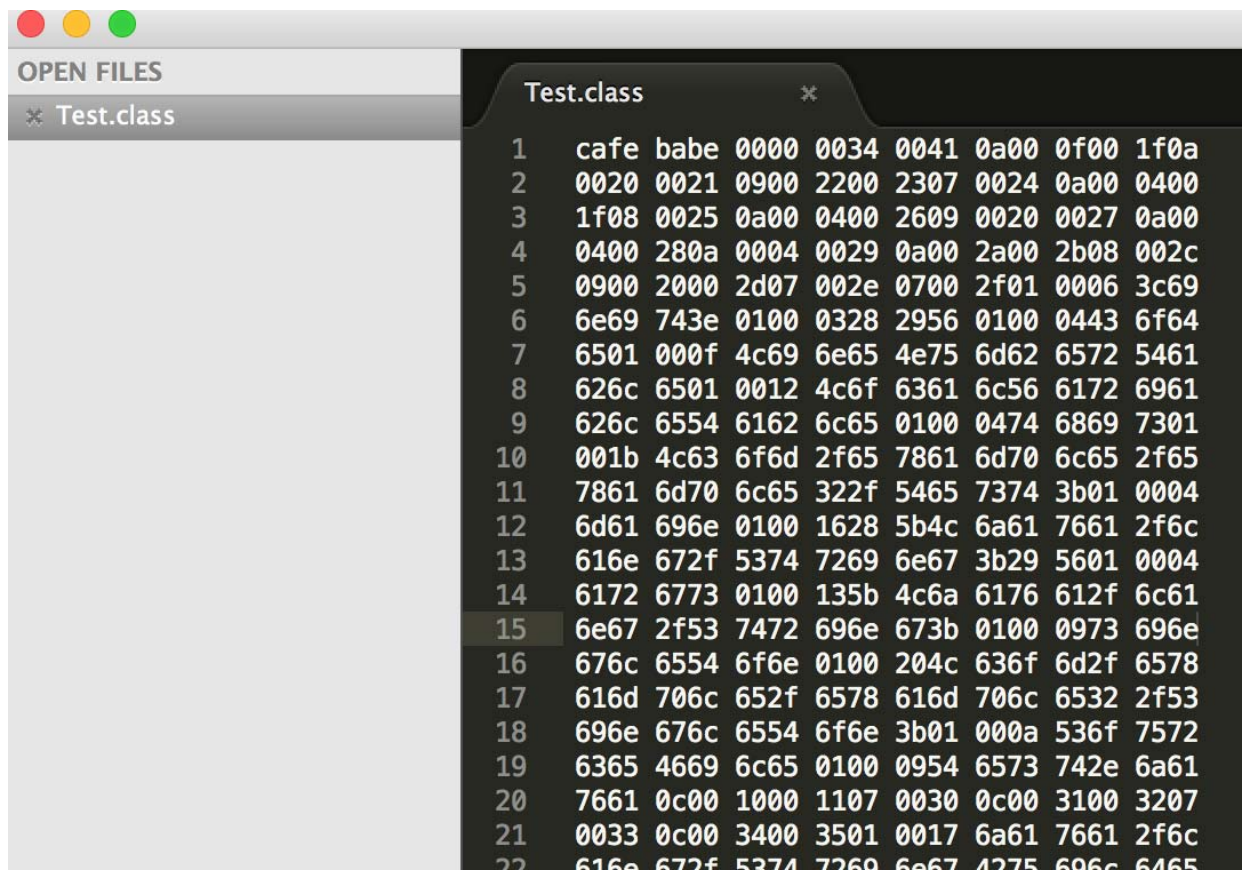
结构体	名字	字节数	描述
magic	魔数	u4 四个字节	其实懂计算机储存原理的都知道，这是用来识别二进制的文件属于什么文件类型。而 .class 文件的魔数用 16 进制来表示正是 CAFEBAFE（咖啡宝贝）
minor_version, major_version	次版本号，主版本号	u2 分别 2 个字节	支持的最低大版本号，小版本号
constant_pool_count	常量池数量	u2 2 个字节长度	

结构体	名字	字节数	描述
constant_pool	常量池	cp_info 常量池表结构	里面包含 String 常量、Class、interface 名称、字段名、常量的引用等
access_flags	访问表示	u2 2 个字节长度	表示这个 class 是 class 还是接口；是否定义了 public 类型；是否定义了 abstract 类型；是否被声明了 final 等
this_class super_class	、 当前类、父类	u2 2 个字节长度	在 constant_pool 里面符合 CONSTANT_Class_info 结构的索引的位置
interfaces_count	接口索引数量	u2 2 个字节长度	接口在 constant_pool 里面符合 CONSTANT_Class_info 结构的索引的数量
interfaces[]	接口索引数组	u2 2 个字节长度	接口在 constant_pool 里面符合 CONSTANT_Class_info 结构的索引的位置
fields_count	字段数量	u2 2 个字节长度	
fields[]	字段的表结构数组	field_info 表结构	符合field_info结构的字段value table
methods_count	方法数量	u2 2 个字节长度	
methods[]	方法对应的表的结构数组	method_info 表结构	符合 method_info 结构的字段 value table
attributes_count	属性数量	u2 2 个字节长度	
attributes[]	属性表结构的数组	attribute_info 表结构	符合 attribute_info 结构的字段 value table

感兴趣的同学可以详细看一下[《JVM 规范》](#)

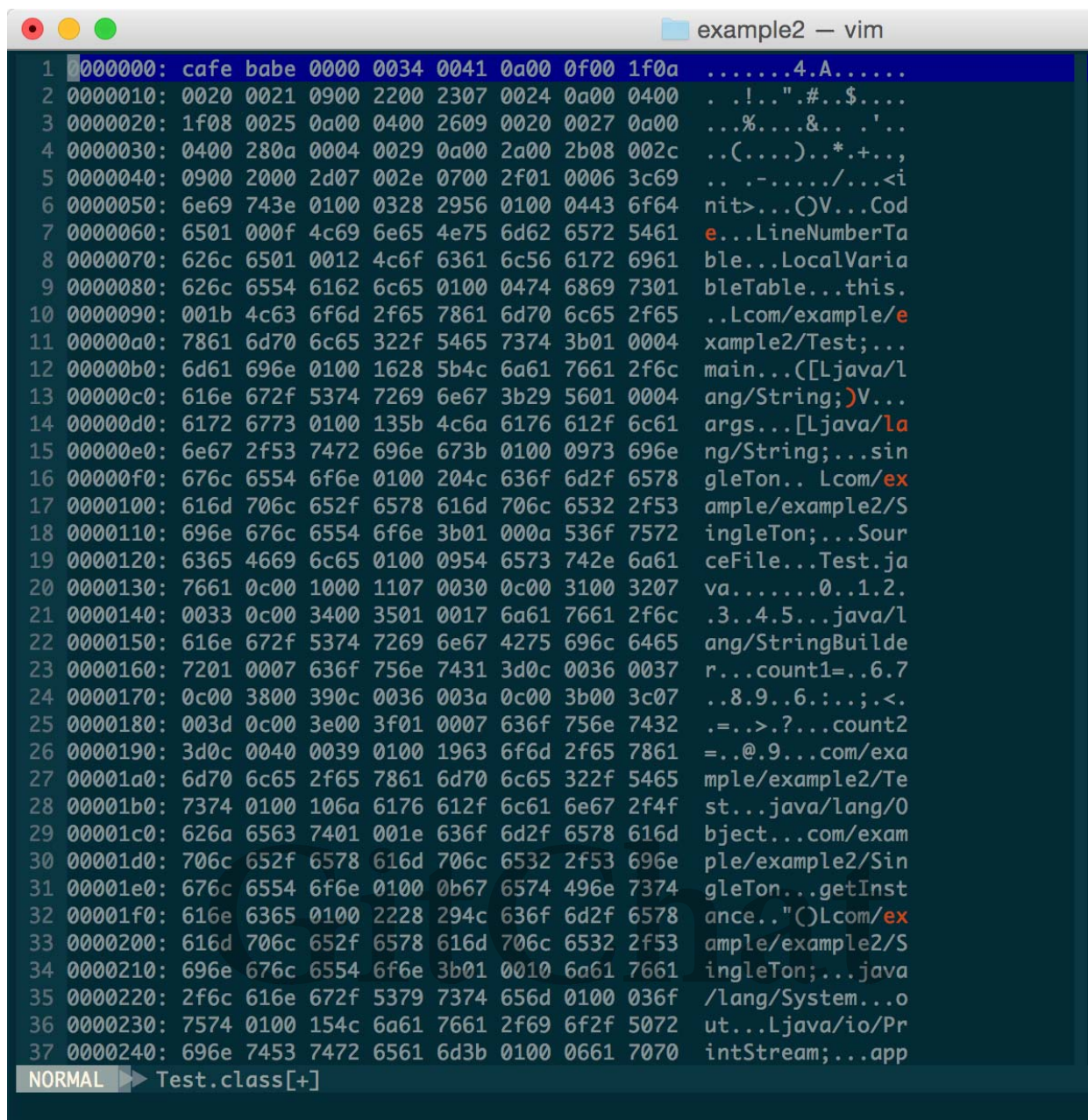
1.3 查看 .class 文件里面的二进制

MAC 情况下，我们可以直接通过 Sublime 打开 .class 文件，此工具默认会以 16 进制的格式来显示字节流。



MAC 中使用 vi 修改二进制文件

1. 首先以二进制方式编辑这个文件 `vi -b datafile`
2. 使用 xxd 转换为 16 进制 `:%!xxd`



```
1 000000: cafe babe 0000 0034 0041 0a00 0f00 1f0a .....4.A.....
2 000010: 0020 0021 0900 2200 2307 0024 0a00 0400 . .!.."#.$....
3 000020: 1f08 0025 0a00 0400 2609 0020 0027 0a00 ...%...&.. '...
4 000030: 0400 280a 0004 0029 0a00 2a00 2b08 002c ..(....)*.+.+,
5 000040: 0900 2000 2d07 002e 0700 2f01 0006 3c69 .. -...../...<i
6 000050: 6e69 743e 0100 0328 2956 0100 0443 6f64 nit>...(C)V...Cod
7 000060: 6501 000f 4c69 6e65 4e75 6d62 6572 5461 e...LineNumberTa
8 000070: 626c 6501 0012 4c6f 6361 6c56 6172 6961 ble...LocalVaria
9 000080: 626c 6554 6162 6c65 0100 0474 6869 7301 bleTable...this.
10 000090: 001b 4c63 6f6d 2f65 7861 6d70 6c65 2f65 ..Lcom/example/e
11 0000a0: 7861 6d70 6c65 322f 5465 7374 3b01 0004 xample2/Test;...
12 0000b0: 6d61 696e 0100 1628 5b4c 6a61 7661 2f6c main...(Ljava/l
13 0000c0: 616e 672f 5374 7269 6e67 3b29 5601 0004 ang/String;)V...
14 0000d0: 6172 6773 0100 135b 4c6a 6176 612f 6c61 args...(Ljava/la
15 0000e0: 6e67 2f53 7472 696e 673b 0100 0973 696e ng/String;...sin
16 0000f0: 676c 6554 6f6e 0100 204c 636f 6d2f 6578 gleTon.. Lcom/ex
17 000100: 616d 706c 652f 6578 616d 706c 6532 2f53 ample/example2/S
18 000110: 696e 676c 6554 6f6e 3b01 000a 536f 7572 ingleTon;...Sour
19 000120: 6365 4669 6c65 0100 0954 6573 742e 6a61 ceFile...Test.ja
20 000130: 7661 0c00 1000 1107 0030 0c00 3100 3207 va.....0..1.2.
21 000140: 0033 0c00 3400 3501 0017 6a61 7661 2f6c .3..4.5...java/l
22 000150: 616e 672f 5374 7269 6e67 4275 696c 6465 ang/StringBuilde
23 000160: 7201 0007 636f 756e 7431 3d0c 0036 0037 r...count1=..6.7
24 000170: 0c00 3800 390c 0036 003a 0c00 3b00 3c07 ..8.9..6...;<.
25 000180: 003d 0c00 3e00 3f01 0007 636f 756e 7432 =..>?...count2
26 000190: 3d0c 0040 0039 0100 1963 6f6d 2f65 7861 =..@.9...com/exa
27 0001a0: 6d70 6c65 2f65 7861 6d70 6c65 322f 5465 mple/example2/Te
28 0001b0: 7374 0100 106a 6176 612f 6c61 6e67 2f4f st...java/lang/0
29 0001c0: 626a 6563 7401 001e 636f 6d2f 6578 616d bject...com/exam
30 0001d0: 706c 652f 6578 616d 706c 6532 2f53 696e ple/example2/Sin
31 0001e0: 676c 6554 6f6e 0100 0b67 6574 496e 7374 gleTon...getInst
32 0001f0: 616e 6365 0100 2228 294c 636f 6d2f 6578 ance.."()Lcom/ex
33 000200: 616d 706c 652f 6578 616d 706c 6532 2f53 ample/example2/S
34 000210: 696e 676c 6554 6f6e 3b01 0010 6a61 7661 ingleTon;...java
35 000220: 2f6c 616e 672f 5379 7374 656d 0100 036f /lang/System...o
36 000230: 7574 0100 154c 6a61 7661 2f69 6f2f 5072 ut...Ljava/io/Pr
37 000240: 696e 7453 7472 6561 6d3b 0100 0661 7070 intStream;...app
```

NORMAL Test.class[+]

1.4 class.java

当我们大致了解了 .class 文件里面的字节流，接下去来了解一下 java.lang.class 类。

认识 class 对象之前，先来了解一个概念，RTTI（Run-Time Type Identification）——运行时类型识别。

这个词一直是 C++ 的概念，Java 中出现 RTTI 的说法是源于《Thinking in Java》一书。其作用是在运行时识别一个对象的类型和类的信息，这里分两种：

- **传统的“RTTI”**，它假定我们在编译期已知道了所有类型（在没有反射机制创建和使用类对象时，一般都是编译期已确定其类型，如 new 对象时该类必须已定义好）
- 另外一种**是反射机制**，它允许我们在运行时发现和使用类型的信息。在 Java 中用来表示运行时类型信息的对应类就是 class 类，class 类也是一个实实在在的类，存在于 JDK 的 java.lang 包中。部分源码内容如下：

```

public final class Class<T> implements java.io.Serializable,
                                     GenericDeclaration,
                                     Type,
                                     AnnotatedElement {
    private static final int ANNOTATION= 0x00002000;
    private static final int ENUM      = 0x00004000;
    private static final int SYNTHETIC = 0x00001000;
    private static native void registerNatives();
    static {
        registerNatives();
    }
    /*
     * Private constructor. Only the Java Virtual Machine creates
     Class objects.
     * This constructor is not used and prevents the default
     constructor being
     * generated.
     */
    private Class(ClassLoader loader) {
        // Initialize final field for classLoader. The
        initialization value of non-null
        // prevents future JIT optimizations from assuming this
        final field is null.
        classLoader = loader;
    }
}

```

作为 Java 工程师不一定要知道 .class 里面的文件字节流详细到每个内容是什么意思，但是 class.java 里面的方法都是我们必须掌握的。因为“反射”在实际开发中会经常被用到。

class 类被创建后的对象就是 class 对象，注意，class 对象表示的是自己手动编写类的类型信息。比如创建一个 Shapes 类，那么 JVM 就会创建一个 Shapes 对应 class 类的 class 对象，该 class 对象保存了 Shapes 类相关的类型信息。

实际上在 Java 中每个类都有一个 class 对象，每当我们编写编译一个新创建的类，就会产生一个对应 class 对象并且这个 class 对象会被保存在同名 .class 文件里（编译后的字节码文件保存的就是 class 对象）。

那为什么需要这样一个 class 对象呢？

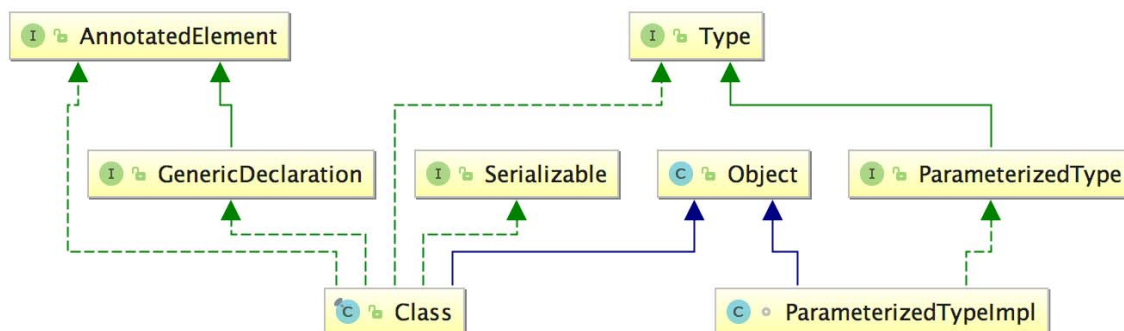
当我们 new 一个新对象或者引用静态成员变量时，Java 虚拟机中的类加载器子系统会将对应 class 对象加载到 JVM 中，然后 JVM 再根据这个类型信息相关的 class 对象创建我们需要的实例对象或者提供静态变量的引用值。

需要特别注意的是，手动编写的每个 class 类，无论创建多少个实例对象，在 JVM 中都只有一个 class 对象，即在内存中每个类有且只有一个相对应的 class 对象（即也就是所

谓的 meta, Java 元数据) 。

- class 类只存私有构造函数，因此对应 class 对象只能由 JVM 创建和加载
- class 类的对象作用是运行时提供或获得某个对象的类型信息，这点对于反射技术很重要。

class 的 UML 图：



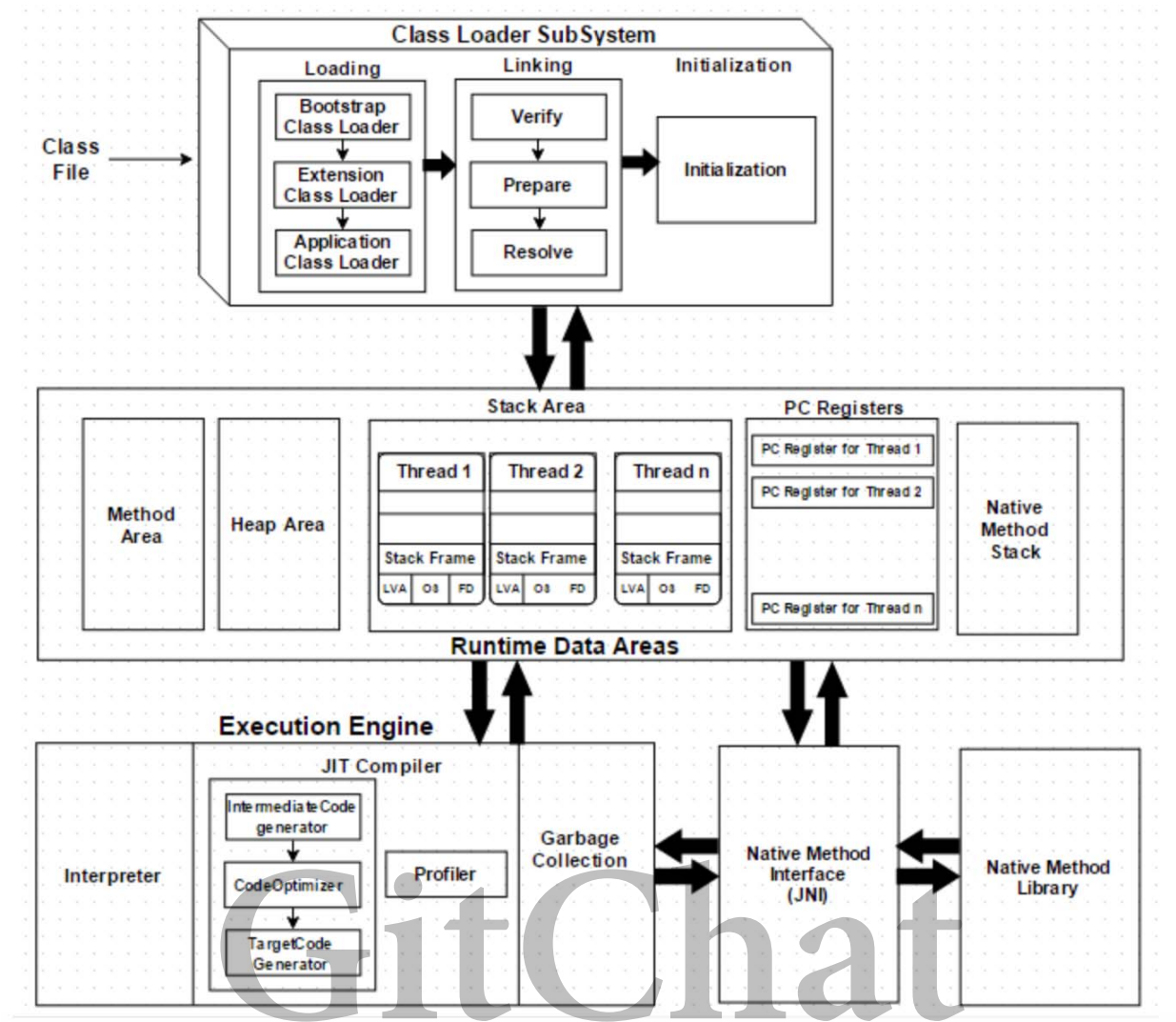
我们通过此图可以发现：

1. Java 里面的反射，涉及到的任何东西都是字节。
2. 而每个反射回来的对象不一定是 class，还有 ParameterizedTypeImpl。

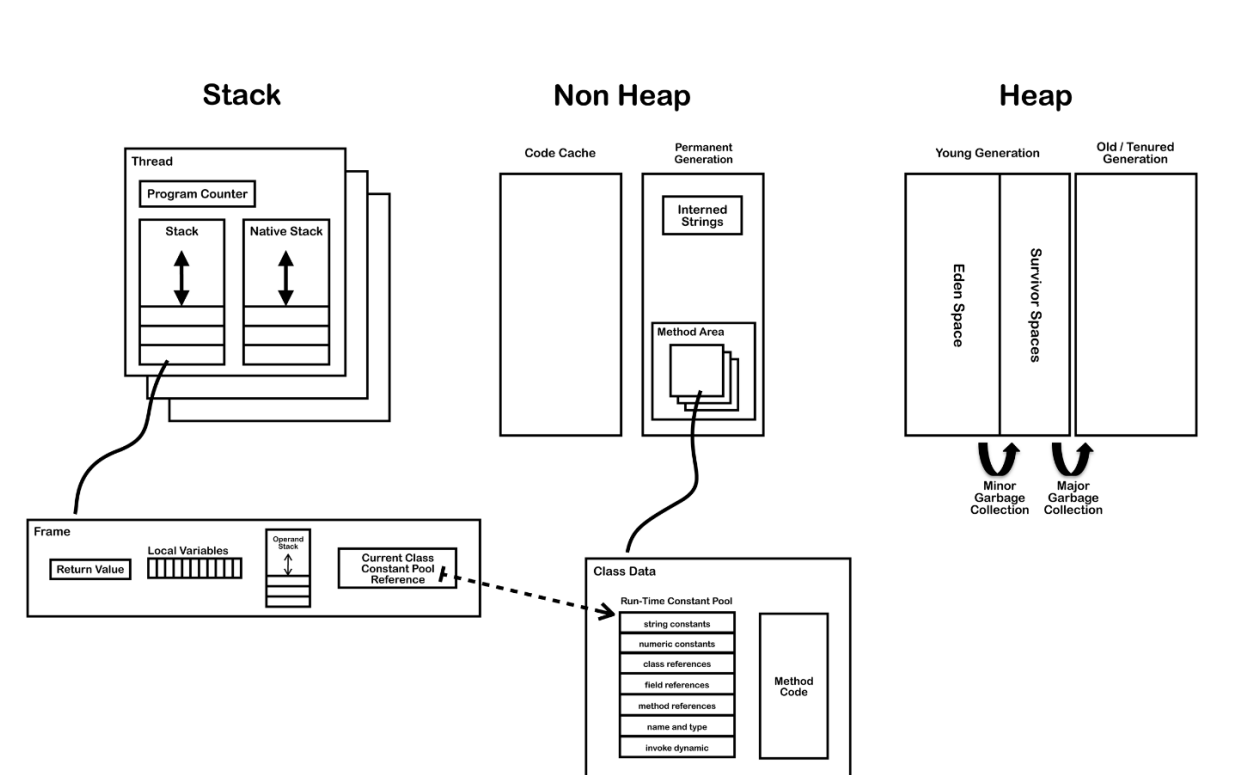
2. class 在 JVM 中

在整个 JVM 的架构设计中，Class Loader 作为一个子系统，是 JVM 运行之前必须执行的一个过程，即在任何 class 初始化之前必须先进行加载，加载完然后放到 MethodArea 里

面，整个过程如下图所示：

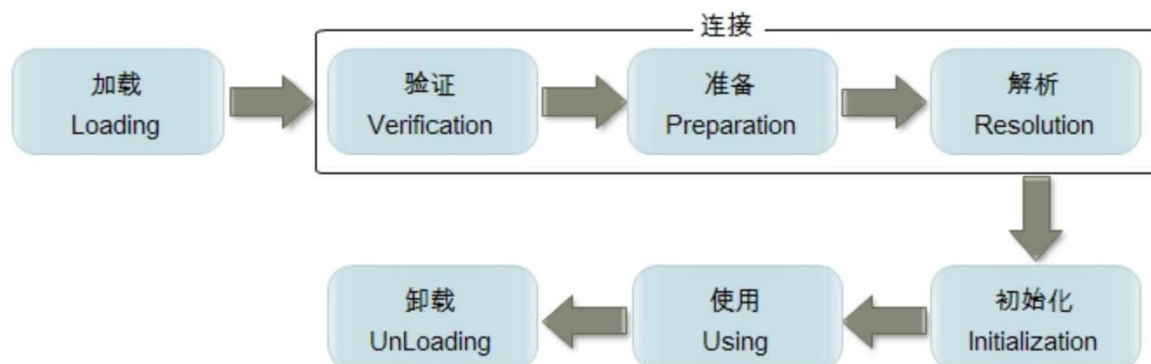


而其中的 Method Area 就是用于存储所得 class 结构的字节流。在运行时会通过 class ref 建立联系如下图：



2.1 class 类加载时机与过程

当我们有了 .class 之后，接下去就是 JVM 通过规范进行 .class 的内存加载。类从加载到虚拟机内存中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用和卸载这 7 个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）。正如下图所示：



其中，加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班的“开始”（仅仅指的是开始，而非执行或者结束，因为这些阶段通常都是互相交叉的混合进行，通常会在一个阶段执行的过程中调用或者激活另一个阶段），而解析阶段则不一定（它在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 语言的运行时绑定）。

2.2 何时开始类的初始化

什么情况下需要开始类加载过程的第一个阶段：“加载”。

虚拟机规范中并没强行约束在什么时候开始类加载过程，可以交给虚拟机的具体实现自由把握。但加载一定发生在初始化之前。对于初始化时机，虚拟机严格规定了如下几种情况：

1. 创建类的实例
2. 访问类的静态变量（被 final 修饰的静态变量除外）

被 final 修饰的静态变量除外原因：常量一种特殊的变量，因为编译器把他们当作值 (value) 而不是域 (field) 来对待。如果你的代码中用到了常变量 (constant variable)，编译器并不会生成字节码，而是直接把这个值插入到字节码中。这是一种很有用的优化，但是如果你需要改变 final 域的值那么每一块用到那个域的代码都需要重新编译。

3. 访问类的静态方法
4. 反射如 (Class.forName(“my.xyz.Test”))
5. 当初始化一个类时，发现其父类还未初始化，则先出发父类的初始化
6. 虚拟机启动时，定义了 main() 方法的那个类先初始化

以上情况称为对一个类进行“主动引用”，除此之外均不会触发类的初始化，称为“被动引

用”。接口的加载过程与类的加载过程稍有不同。接口中不能使用 `static{} 块`。当一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有真正在使用到父接口时（例如引用接口中定义的常量）才会初始化。也就是说：

1. 子类调用父类的静态变量，子类不会被初始化。只有父类被初始化。对于静态字段，只有直接定义这个字段的类才会被初始化。
2. 通过数组定义来引用类，不会触发类的初始化
3. 访问类的常量，不会初始化类

2.3 初始化 5 步走

2.3.1 加载

“加载” (Loading) 是“类加载” (Class Loading) 的第一个阶段，在此阶段，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。
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。
3. 在 Java 堆中生成一个代表这个类的 `java.lang.class` 对象，作为方法区这些数据的访问入口。

加载阶段既可以使用系统提供的类加载器完成，也可以由用户自定义的类加载器来完成。加载阶段与连接阶段的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始。

2.3.2 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。

Java 语言本身是相对安全的语言，使用 Java 编码无法做到如访问数组边界以外的数据、将一个对象转型为它并未实现的类型等，如果这样做，编译器将拒绝编译。但是，class 文件并不一定是由 Java 源码编译而来，可以使用任何途径，包括用十六进制编辑器（如 UltraEdit）直接编写。如果直接编写了有害的“代码”，而虚拟机在加载该 class 时不进行检查的话，就有可能危害到虚拟机或程序的安全。

不同的虚拟机，对类验证的实现可能有所不同，但大致都会完成下面四个阶段的验证：文件格式验证、元数据验证、字节码验证和符号引用验证。

1. 文件格式验证，是要验证字节流是否符合 class 文件格式规范，且能被当前版本的虚拟机处理。如验证魔数是否 `0xCAFEBAE`；主、次版本号是否正在当前虚拟机处理范围之内；常量池的常量中是否有不被支持的常量类型……该验证阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区中，经过这个阶段的验证后，字节流才会进入内存的方法区中存储，所以后面的三个验证阶段都是基于方法区的存储结构进行的。
2. 元数据验证，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。可能包括的验证如：这个类是否有父类；这个类的父类是

否继承了不被允许继承的类；如果这个类不是抽象类，是否实现了其父类或接口中要求实现的所有方法.....

3. 字节码验证，主要工作是进行数据流和控制流分析，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。如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。
4. 符号引用验证，发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在“解析阶段”中发生。验证符号引用中通过字符串描述的权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字段的描述符及简单名称所描述的方法和字段；符号引用中的类、字段和方法的访问性（private、protected、public、default）是否可被当前类访问

验证阶段对于虚拟机的类加载机制来说，不一定是必要的阶段。如果所运行的全部代码确认是安全的，可以使用 `-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时间。

2.3.3 准备

准备阶段为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中实例变量的内存，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。

```
public static int value=123;//在准备阶段value初始值为0。在初始化阶段才会变为123。
```

2.3.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

1. 符号引用（Symbolic Reference）：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。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。
2. 直接引用（Direct Reference）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，如果有了直接引用，那么引用的目标必定已经在内存中存在。

2.3.5 初始化

类初始化是类加载过程的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

初始化阶段是执行类构造器 `<clinit>()` 方法的过程。`<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。

3. ClassLoader 详解:

Java 默认提供的三个 ClassLoader，分别是：Bootstrap ClassLoader、Extension ClassLoader、App ClassLoader。我们执行一个新建的 test.java，打印出来所有的 ClassLoader 来看看加载器都有哪些？

```
public class Test {  
    public static void main(String[] args) {  
        ClassLoader loader = Test.class.getClassLoader();    //获得  
        加载ClassLoaderTest.class这个类的类加载器  
        while(loader != null) {  
            System.out.println(loader);  
            loader = loader.getParent();    //获得父类加载器的引用  
        }  
    }  
}
```

通过上面例子，执行结果如下：

```
sun.misc.Launcher$AppClassLoader@14dad5dc  
sun.misc.Launcher$ExtClassLoader@1d81eb93  
null
```

- 第一行结果说明：ClassLoaderTest 的类加载器是 AppClassLoader。
- 第二行结果说明：AppClassLoader 的类加载器是 ExtClassLoader，即 parent=ExtClassLoader。
- 第三行结果说明：ExtClassLoader 的类加载器是 Bootstrap ClassLoader，因为 Bootstrap ClassLoader 不是一个普通的 Java 类，所以 ExtClassLoader 的 parent=null，所以第三行的打印结果为 null 就是这个原因。

3.1 ClassLoader 的体系架构及其加载原理

ClassLoader 是个抽象父类，看一下它的关键代码：

```
public abstract class ClassLoader {  
    private static native void registerNatives();  
    static {  
        registerNatives();  
    }  
    protected Class<?> loadClass(String name, boolean resolve) throws  
    ClassNotFoundException{  
        synchronized (getClassLoadingLock(name)) {  
            // First, check if the class has already been loaded  
            Class c = findLoadedClass(name);  
            if (c != null) return c;  
            // The class is not loaded. The caller must pass the resolved  
            // class name.  
            if (parent != null) {  
                // First, try the parent class loader to see if they  
                // can load the class. Don't call the parent's loadClass()  
                // method. We've already called it. We're only calling it  
                // because we want to ensure that the parent is responsible  
                // for the class's existence; not us. This means that we  
                // won't fail to load the class if the parent can't load it.  
                c = parent.loadClass(name, false);  
                if (c != null) return c;  
            }  
            // If the parent can't load the class, then we load the  
            // class. We're calling the parent's loadClass() method to  
            // ensure that the parent is responsible for the class's  
            // existence; not us. This means that we won't fail to load  
            // the class if the parent can't load it.  
            c = findClass(name);  
            if (resolve) resolveClass(c);  
            return c;  
        }  
    }  
}
```

```

        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not
found
                // from the non-null parent class loader
            }
            if (c == null) {
                // If still not found, then invoke findClass in
order
                // to find the class.
                long t1 = System.nanoTime();
                c = findClass(name);
                // this is the defining class loader; record the
stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
.....}

```

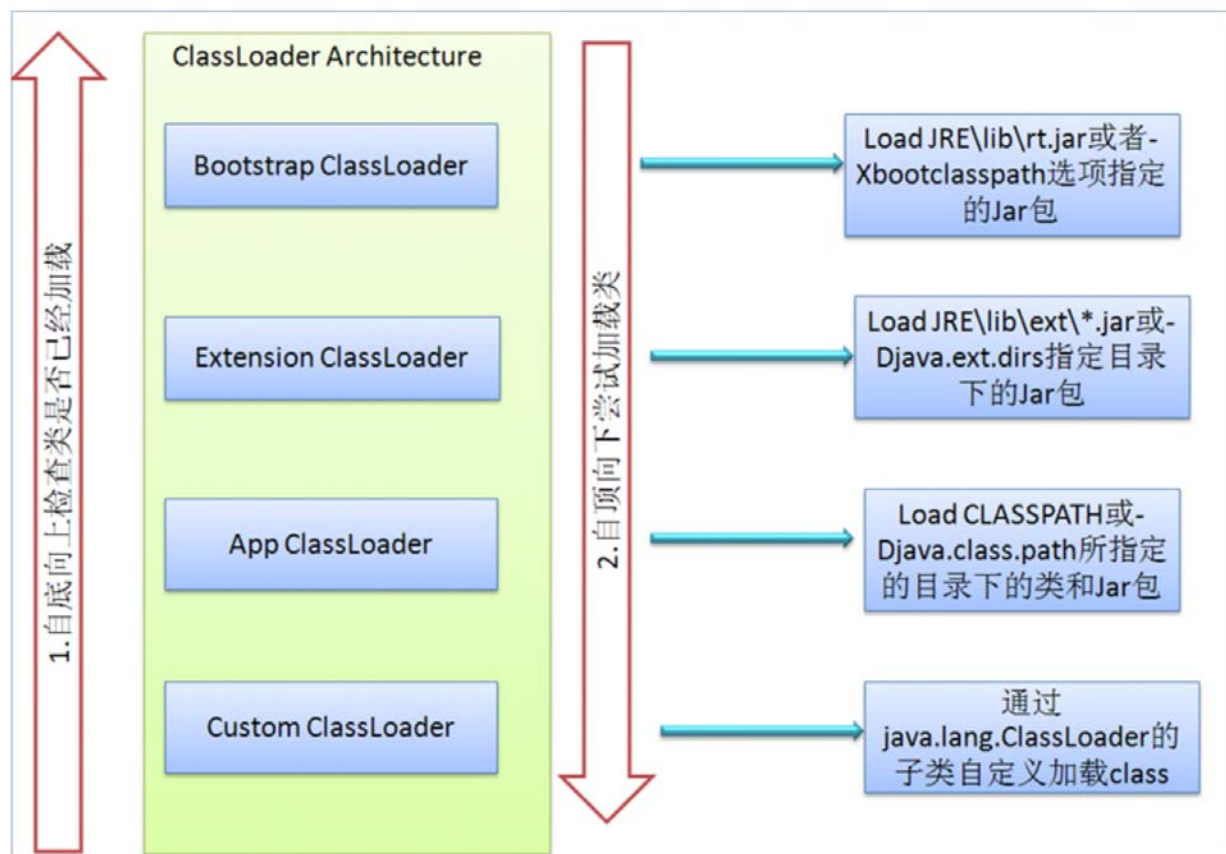
通过源码和 JDK 里面的 classload 注释，这方法的目的是使用指定的二进制名称来加载类，默认实现按照以下顺序查找类：

1. 调用 findLoadedClass(String) 方法检查这个类是否被加载过
2. 使用父加载器调用 loadClass(String) 方法
3. 如果父加载器为 Null，类加载器装载虚拟机内置的加载器调用 findClass(String) 方法装载类

如果按照以上的步骤成功的找到对应的类，并且该方法接收的 resolve 参数的值为 true，那么就调用 resolveClass(Class) 方法来处理类。

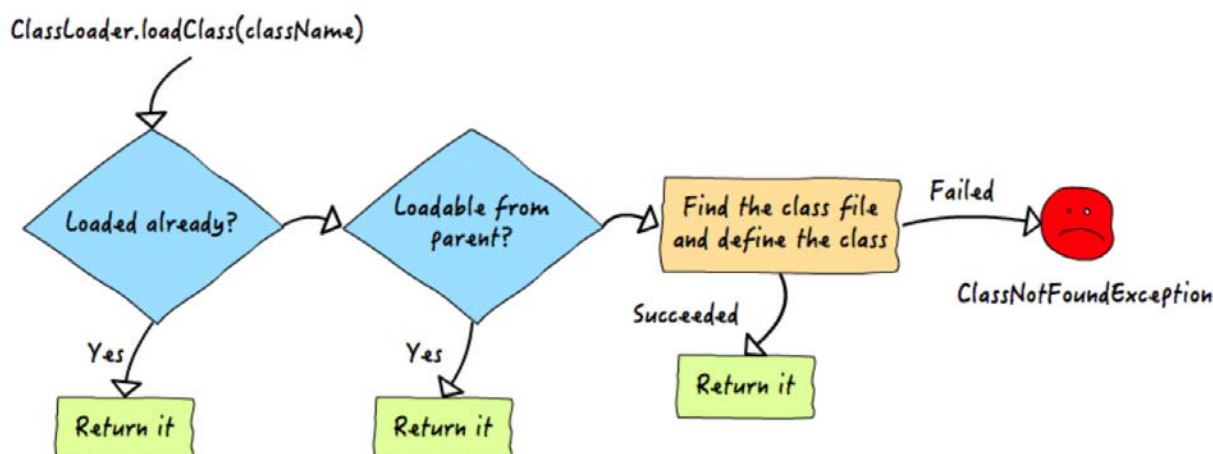
ClassLoader 的子类最好覆盖 findClass(String)，除非被重写，这个方法默认在整个装载过程中都是同步的（线程安全的）。

也就是如下规则：



双亲委托模型：

其实这个规则概括起来就是我们常说的“双亲委托模型”。每个 ClassLoader 实例都有一个父类加载器的引用（不是继承的关系，是一个包含的关系），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但可以用作其它 ClassLoader 实例的父类加载器。当一个 ClassLoader 实例需要加载某个类时，它会试图亲自搜索某个类，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器 Bootstrap ClassLoader 试图加载，如果没加载到，则把任务转交给 Extension ClassLoader 试图加载，如果也没加载到，则转交给 App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等 URL 中加载该类。如果它们都没有加载到这个类时，则抛出 `ClassNotFoundException` 异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的 Class 实例对象。正如下图所示：



为什么要使用双亲委托这种模型？

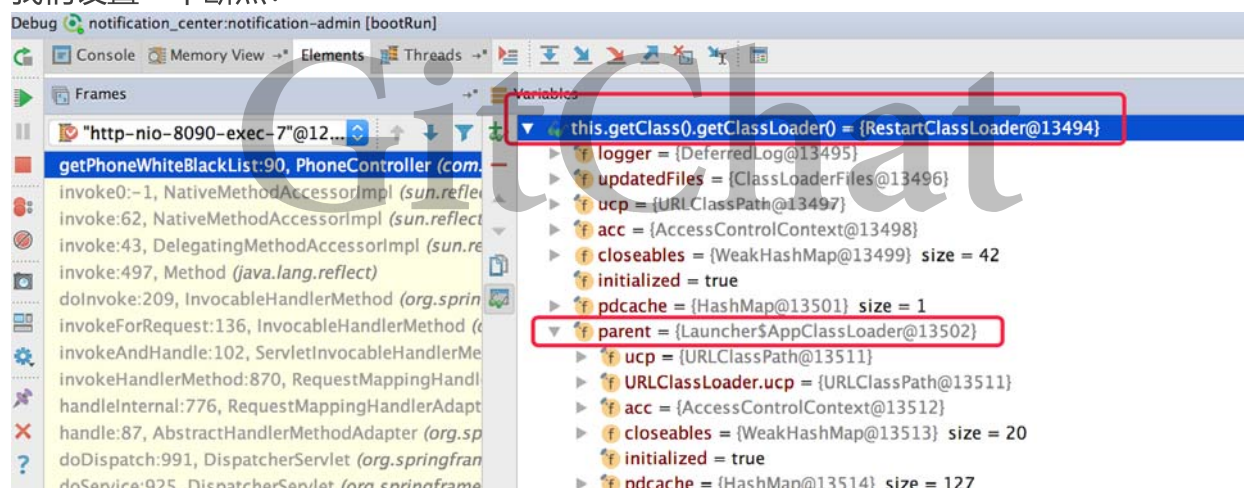
因为这样可以避免重复加载，当父类已经加载了该类的时候，就没有必要子 ClassLoader 再加载一次。考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的 string 来动态替代 Java 核心 API 中定义的类型，这样会存在非常大的安全隐患；而双亲委托的方式，就可以避免这种情况，因为 string 已经在启动时就被引导类加载器（Bootstrap ClassLoader）加载，所以用户自定义的 ClassLoader 永远也无法加载一个自己写的 string，除非你改变 JDK 中 ClassLoader 搜索类的默认算法。

JVM 在搜索类的时候，如何判定两个 Class 是相同的？

JVM 在判定两个 Class 是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实例加载的。只有两者同时满足的情况下，JVM 才认为这两个 Class 是相同的。

3.2 Spring 的 ClassLoader

而当我们使用 Spring Boot 的时候，会用到 spring-boot-devtools 来帮我们实现热部署。而其中热部署的原理就是使用了自定义 class loader 来实现类的重新加载，如下图，我们设置一个断点：

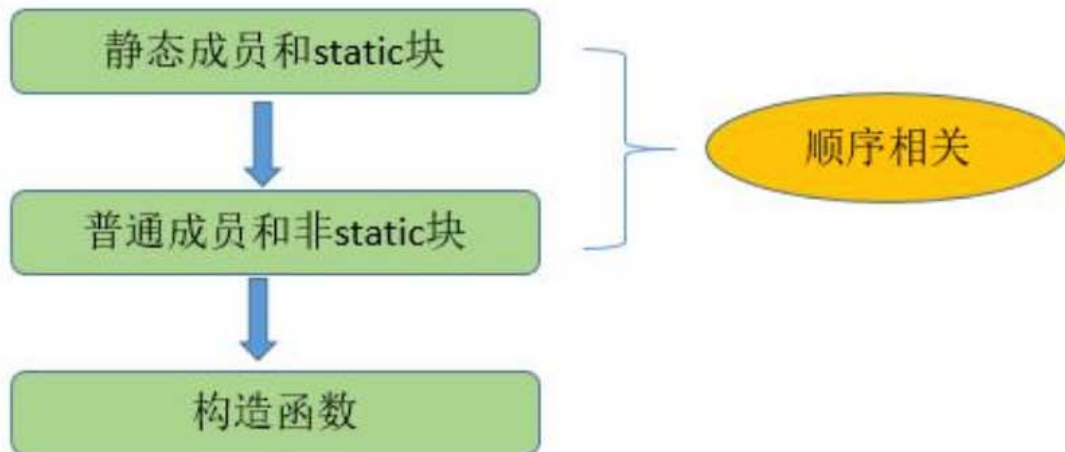


4. Java 初始化顺序

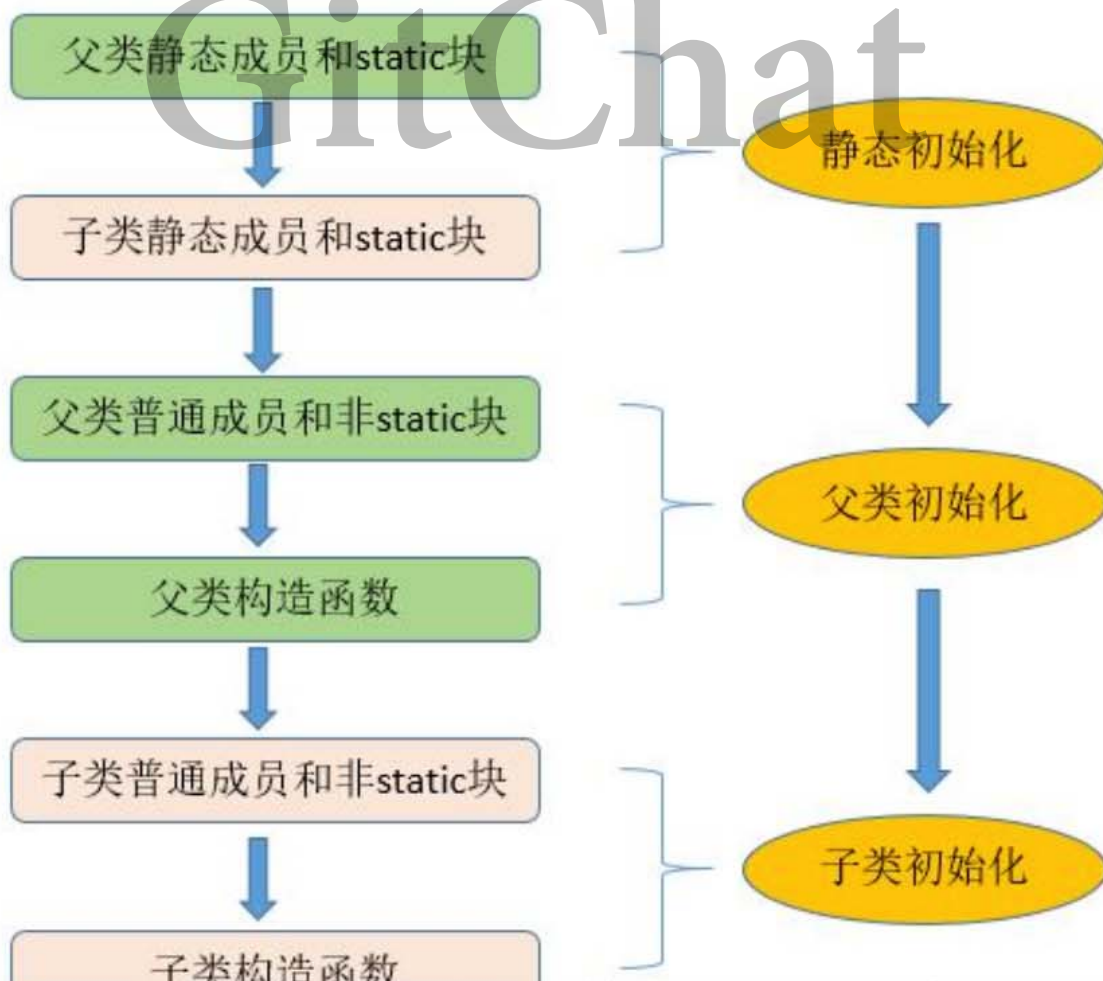
正如下图所示的 Java 单个对象的初始化顺序：

初始化顺序图示

非继承关系



继承关系



非继承加载顺序验证 1：

```

public class ExecuteDemo {
    public static void main(String[ ] args) {
        DemonstrateOrder DO = new
DemonstrateOrder ( );    }
}
class DemonstrateOrder{
    static String vstatic = verify(" 1a - Static class
variable vstatic initialized first." );
    String vnonstatic = verify(" 2 - Non-static class
variable vnonstatic initialized." );
    static {System.out.println(" 1b - Static initialization
block ran." ); }
    static String verify(String s) { System.out.println(s);
return s; }
    DemonstrateOrder ( ) {System.out.println(" 3 -
Constructor ran." ); }
}

```

我们执行一下打印结果如下:

```

1a - Static class variable vstatic initialized first.
1b - Static initialization block ran.
2 - Non-static class variable vnonstatic initialized.
3 - Constructor ran.

```

继承加载顺序验证 2:

```

class Parent {
String p = verify("4 - Parent's non-static class variable p
initialized.");
static String pstatic = verify("1a - Parent's static class
variable pstatic initialized.");
static {System.out.println("1b - Parent's static initialization
block ran.");}
static String verify(String s) {System.out.println(s); return
s;}
Parent( ) {System.out.println("5 - Parent constructor ran."); }
}
// Child
public class Child extends Parent {
static {System.out.println("2a - Child's static initialization
block ran.");}
String c = verify("6 - Child's non-static class variable c
initialized.");
static String cstatic = verify("2b - Child's static class
variable cstatic initialized.");
Child( ) {
//super( );    //调不调用super执行结果一样

```

```
System.out.println ("7 - Child constructor ran."); }  
public static void main(String[ ] args) {  
System.out.println("3 - Child's main(..) method ran. Invoking  
Child with new.");  
Child C = new Child( );  
System.out.println("8 - The rest of main(..) was run.");  
}  
}
```

执行结果如下：

```
1a - Parent's static class variable pstatic initialized.  
1b - Parent's static initialization block ran.  
2a - Child's static initialization block ran.  
2b - Child's static class variable cstatic initialized.  
3 - Child's main(..) method ran. Invoking Child with new.  
4 - Parent's non-static class variable p initialized.  
5 - Parent constructor ran.  
6 - Child's non-static class variable c initialized.  
7 - Child constructor ran.  
8 - The rest of main(..) was run.
```

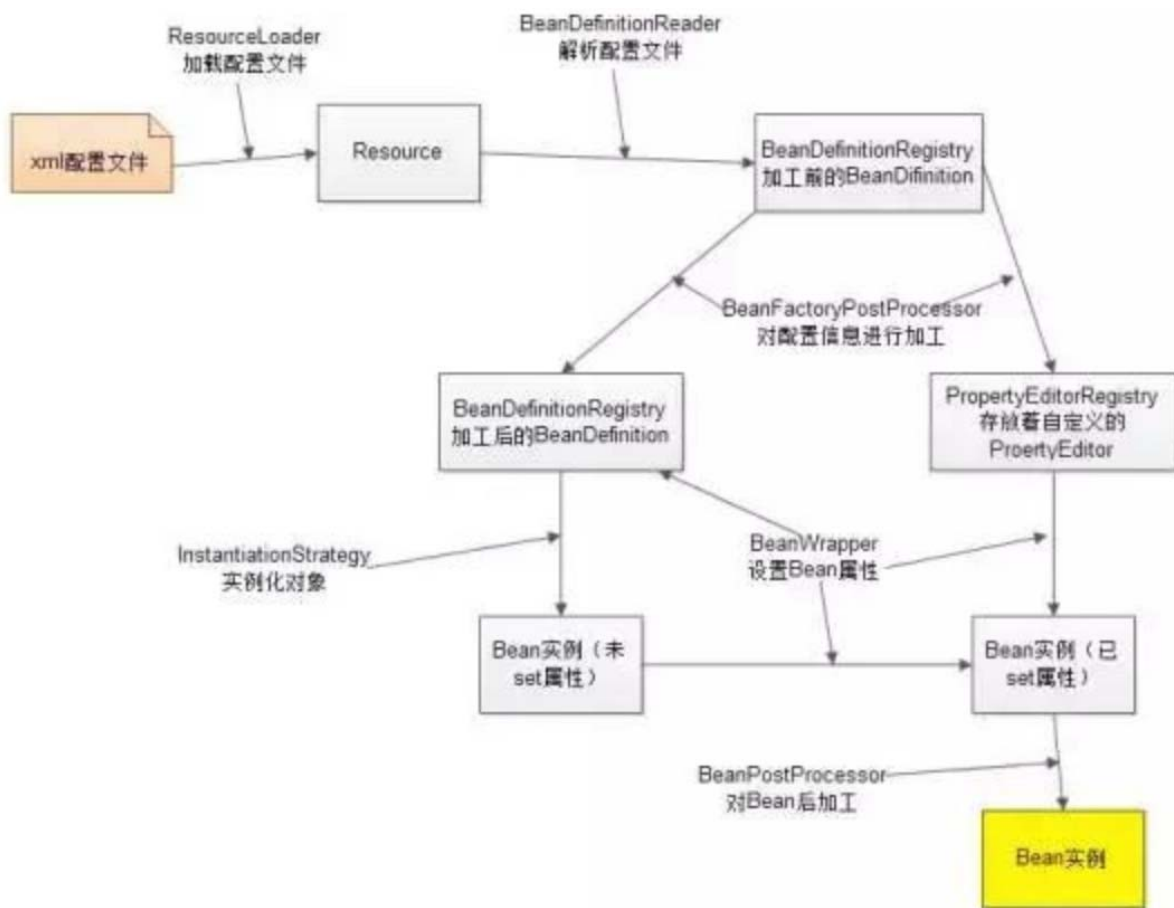
static 关键字

而其中 static 修饰的变量，初始化实际上分成两步：

1. 静态变量随着类加载完成初始化和默认值
2. 当 loading 完，在 initialization 这步完成了初始化

5. Spring Bean 加载原理

我们提到了 JVM 的 Class 的加载原理，那我们也顺便看一下 Spring Bean 的加载原理：一个是应用级别的，一个是 JVM 级别的。我们来简单了解一下：



1. ResourceLoader 从存储介质中加载 Spring 配置信息，并使用 Resource 表示这个配置文件的资源；
2. BeanDefinitionReader 读取 Resource 所指向的配置文件资源，然后解析配置文件。配置文件中每一个解析成一个 BeanDefinition 对象，并保存到 BeanDefinitionRegistry 中；
3. 容器扫描 BeanDefinitionRegistry 中的 BeanDefinition，使用 Java 的反射机制自动识别出 Bean 工厂后处理器（实现 BeanFactoryPostProcessor 接口）的 Bean，然后调用这些 Bean 工厂后处理器对 BeanDefinitionRegistry 中的 BeanDefinition 进行加工处理。主要完成以下两项工作：
 - 1) 对使用到占位符的元素标签进行解析，得到最终的配置值，这意味对一些半成品式的 BeanDefinition 对象进行加工处理并得到成品的 BeanDefinition 对象；
 - 2) 对 BeanDefinitionRegistry 中的 BeanDefinition 进行扫描，通过 Java 反射机制找出所有属性编辑器的 Bean（实现 java.beans.PropertyEditor 接口的 Bean），并自动将它们注册到 Spring 容器的属性编辑器注册表中（PropertyEditorRegistry）；
4. Spring 容器从 BeanDefinitionRegistry 中取出加工后的 BeanDefinition，并调用 InstantiationStrategy 着手进行 Bean 实例化的工作；
5. 在实例化 Bean 时，Spring 容器使用 BeanWrapper 对 Bean 进行封装，BeanWrapper 提供了很多以 Java 反射机制操作 Bean 的方法，它将结合该 Bean 的 BeanDefinition 以及容器中属性编辑器，完成 Bean 属性的设置工作；
6. 利用容器中注册的 Bean 后处理器（实现 BeanPostProcessor 接口的 Bean）对已经完成属性设置工作的 Bean 进行后续加工，直接装配出一个准备就绪的 Bean。

总之一句话：Spring Bean 的 loader 是基于 bean 的实例化配置，然后加载 Bean 进行初始化，然后根据 scope 在适当的实际分配实例化对象。

在面试和工作中起到什么作用：

1. 当我们面试的时候，像阿里、腾讯、百度等一些顶级的互联网公司，会经常考这方面的面试题用来判断你对技术的追求程度。可以说如果问到这样的面试题，基本上就能判断出来你的级别是什么样的。在此老师也希望大家多多留言，老师也不一定想的很全面，希望和大家一起交流。
2. 工作中，热部署、自定义框架的时候，可能必须了解 class 的 loader 机制了，否则你写出来的框架就可能有 bug。
3. 当我们做动态加载的时候，以上的知识都是必备，甚至需要掌握的比老师的这些知识还要多。

老师留下三道面试题，在我们微信 chat 的时候大家来讨论：

1. final static 怎么理解？
2. 如何利用 byte 二进制流来加载 Class？
3. 看下这段代码的最终执行结果是什么？

```
class Singleton {
    private static Singleton singleton = new Singleton();
    public static int count1;
    public static int count2 = 0;
    private Singleton() {
        count1++;
        count2++;
    }
    public static Singleton getInstance() {
        return singleton;
    }
}

public class Test {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        System.out.println("count1=" + singleton.count1);
        System.out.println("count2=" + singleton.count2);
    }
}
```