JVM类加载相关笔记

此文档主要记录以下部分相关笔记：

1. java.exe开始类加载运行过程
2. JVM类加载器
3. 双亲委派机制
4. 手写类加载器打破双亲委派
5. Tomcat类加载机制
6. 类加载过程

当我们用java命令运行某个类的main函数启动程序时，首先需要通过类加载器把主类加载到JVM。

package com.test;

import com.test.User;

public class Math {

public static final int initData = 666;

public static User user = new User();

public int compute() { //一个方法对应一块栈帧内存区域

int a = 1;

int b = 2;

int c = (a + b) \* 10;

return c;

}

public static void main(String[] args) {

Math math = new Math();

math.compute();

}

}

通过Java命令执行代码的大体流程如下：



其中 类加载的过程如下



其中loadClass的类加载过程有如下几步：

加载 >> 验证 >> 准备 >> 解析 >> 初始化 >> 使用 >> 卸载

1. 加载：在硬盘上查找并通过IO读入字节码文件，使用到类时才会加载，例如调用类的main()方法，new对象等等，在加载阶段会在内存中生成一个代表这个类的
2. 验证：校验字节码文件的正确性
3. 准备：给类的静态变量分配内存，并赋予默认值（静态变量赋值为默认值，比如Integer是0，Boolean是false，复杂数据类型为null，final修饰的直接赋值）
4. 解析：将符号引用替换为直接引用，该阶段会把一些静态方法(符号引用，比如main()方法)替换为指向数据所存内存的指针或句柄等(直接引用)，这是所谓的静态链接过程(类加载期间完成)，动态链接是在程序运行期间完成的将符号引用替换为直接引用，比如对象的方法链接指向了另一个符号引用，在调用这个对象方法时会调用被指向的符号引用
5. 初始化：对类的静态变量初始化为指定的值，执行静态代码块

使用javap -v Math.class 命令查看class字节码如下：

Classfile /D:/笔记/notes/code/target/classes/com/test/Math.class

Last modified 2021-2-28; size 744 bytes

MD5 checksum c235e43e071a43ac30c22cc28b383dc1

Compiled from "Math.java"

public class com.test.Math

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #8.#35 // java/lang/Object."<init>":()V

#2 = Class #36 // com/test/Math

#3 = Methodref #2.#35 // com/test/Math."<init>":()V

#4 = Methodref #2.#37 // com/test/Math.compute:()I

#5 = Class #38 // com/test/User

#6 = Methodref #5.#35 // com/test/User."<init>":()V

#7 = Fieldref #2.#39 // com/test/Math.user:Lcom/test/User;

#8 = Class #40 // java/lang/Object

#9 = Utf8 initData

#10 = Utf8 I

#11 = Utf8 ConstantValue

#12 = Integer 666

#13 = Utf8 user

#14 = Utf8 Lcom/test/User;

#15 = Utf8 <init>

#16 = Utf8 ()V

#17 = Utf8 Code

#18 = Utf8 LineNumberTable

#19 = Utf8 LocalVariableTable

#20 = Utf8 this

#21 = Utf8 Lcom/test/Math;

#22 = Utf8 compute

#23 = Utf8 ()I

#24 = Utf8 a

#25 = Utf8 b

#26 = Utf8 c

#27 = Utf8 main

#28 = Utf8 ([Ljava/lang/String;)V

#29 = Utf8 args

#30 = Utf8 [Ljava/lang/String;

#31 = Utf8 math

#32 = Utf8 <clinit>

#33 = Utf8 SourceFile

#34 = Utf8 Math.java

#35 = NameAndType #15:#16 // "<init>":()V

#36 = Utf8 com/test/Math

#37 = NameAndType #22:#23 // compute:()I

#38 = Utf8 com/test/User

#39 = NameAndType #13:#14 // user:Lcom/test/User;

#40 = Utf8 java/lang/Object

{

public static final int initData;

descriptor: I

flags: ACC\_PUBLIC, ACC\_STATIC, ACC\_FINAL

ConstantValue: int 666

public static com.test.User user;

descriptor: Lcom/test/User;

flags: ACC\_PUBLIC, ACC\_STATIC

public com.test.Math();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 2: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/test/Math;

public int compute();

descriptor: ()I

flags: ACC\_PUBLIC

Code:

stack=2, locals=4, args\_size=1

0: iconst\_1

1: istore\_1

2: iconst\_2

3: istore\_2

4: iload\_1

5: iload\_2

6: iadd

7: bipush 10

9: imul

10: istore\_3

11: iload\_3

12: ireturn

LineNumberTable:

line 6: 0

line 7: 2

line 8: 4

line 9: 11

LocalVariableTable:

Start Length Slot Name Signature

0 13 0 this Lcom/test/Math;

2 11 1 a I

4 9 2 b I

11 2 3 c I

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=2, args\_size=1

0: new #2 // class com/test/Math

3: dup

4: invokespecial #3 // Method "<init>":()V

7: astore\_1

8: aload\_1

9: invokevirtual #4 // Method compute:()I

12: pop

13: return

LineNumberTable:

line 12: 0

line 13: 8

line 14: 13

LocalVariableTable:

Start Length Slot Name Signature

0 14 0 args [Ljava/lang/String;

8 6 1 math Lcom/test/Math;

static {};

descriptor: ()V

flags: ACC\_STATIC

Code:

stack=2, locals=0, args\_size=0

0: new #5 // class com/test/User

3: dup

4: invokespecial #6 // Method com/test/User."<init>":()V

7: putstatic #7 // Field user:Lcom/test/User;

10: return

LineNumberTable:

line 4: 0

}

SourceFile: "Math.java"

类被加载到方法区中后主要包含 运行时常量池、类型信息、字段信息、方法信息、类加载器的引用、对应class实例的引用等信息。

类加载器的引用：这个类到类加载器实例的引用

对应class实例的引用：类加载器在加载类信息放到方法区中后，会创建一个对应的Class 类型的对象实例放到堆(Heap)中, 作为开发人员访问方法区中类定义的入口和切入点。

注意，主类在运行过程中如果使用到其它类，会逐步加载这些类。

jar包或war包里的类不是一次性全部加载的，是使用到时才加载。

package com.test;

public class TestDynamicLoad {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load TestDynamicLoad\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static void main(String[] args) {

new A();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load test\*\*\*\*\*\*\*\*\*\*\*\*");

B b = null; //B不会加载，除非这里执行 new B()

}

}

class A {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load A\*\*\*\*\*\*\*\*\*\*\*\*");

}

public A() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*initial A\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

class B {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load B\*\*\*\*\*\*\*\*\*\*\*\*");

}

public B() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*initial B\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

运行结果：  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load TestDynamicLoad\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load A\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*initial A\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load test\*\*\*\*\*\*\*\*\*\*\*\*

类加载器和双亲委派机制

上面的类加载过程主要是通过类加载器来实现的，Java里有如下几种类加载器

1. 引导类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
2. 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
3. 应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
4. 自定义加载器：负责加载用户自定义路径下的类包

package com.test;

import sun.misc.Launcher;

import java.net.URL;

public class TestJDKClassLoader {

public static void main(String[] args) {

System.out.println(String.class.getClassLoader());

System.out.println(com.sun.crypto.provider.DESKeyFactory.class.getClassLoader().getClass().getName());

System.out.println(TestJDKClassLoader.class.getClassLoader().getClass().getName());

System.out.println();

ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();

ClassLoader extClassloader = appClassLoader.getParent();

ClassLoader bootstrapLoader = extClassloader.getParent();

System.out.println("the bootstrapLoader : " + bootstrapLoader);

System.out.println("the extClassloader : " + extClassloader);

System.out.println("the appClassLoader : " + appClassLoader);

System.out.println();

System.out.println("bootstrapLoader加载以下文件：");

URL[] urls = Launcher.getBootstrapClassPath().getURLs();

for (int i = 0; i < urls.length; i++) {

System.out.println(urls[i]);

}

System.out.println();

System.out.println("extClassloader加载以下文件：");

System.out.println(System.getProperty("java.ext.dirs"));

System.out.println();

System.out.println("appClassLoader加载以下文件：");

System.out.println(System.getProperty("java.class.path"));

}

}

输出：

null

sun.misc.Launcher$ExtClassLoader

sun.misc.Launcher$AppClassLoader

the bootstrapLoader : null

the extClassloader : sun.misc.Launcher$ExtClassLoader@4b67cf4d

the appClassLoader : sun.misc.Launcher$AppClassLoader@18b4aac2

bootstrapLoader加载以下文件：

file:/C:/JAVA/JDK8/jre/lib/resources.jar

file:/C:/JAVA/JDK8/jre/lib/rt.jar

file:/C:/JAVA/JDK8/jre/lib/sunrsasign.jar

file:/C:/JAVA/JDK8/jre/lib/jsse.jar

file:/C:/JAVA/JDK8/jre/lib/jce.jar

file:/C:/JAVA/JDK8/jre/lib/charsets.jar

file:/C:/JAVA/JDK8/jre/lib/jfr.jar

file:/C:/JAVA/JDK8/jre/classes

extClassloader加载以下文件：

C:\JAVA\JDK8\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext

appClassLoader加载以下文件：

C:\JAVA\JDK8\jre\lib\charsets.jar;C:\JAVA\JDK8\jre\lib\deploy.jar;C:\JAVA\JDK8\jre\lib\ext\access-bridge-64.jar;C:\JAVA\JDK8\jre\lib\ext\cldrdata.jar;C:\JAVA\JDK8\jre\lib\ext\dnsns.jar;C:\JAVA\JDK8\jre\lib\ext\jaccess.jar;C:\JAVA\JDK8\jre\lib\ext\jfxrt.jar;C:\JAVA\JDK8\jre\lib\ext\localedata.jar;C:\JAVA\JDK8\jre\lib\ext\nashorn.jar;C:\JAVA\JDK8\jre\lib\ext\sunec.jar;C:\JAVA\JDK8\jre\lib\ext\sunjce\_provider.jar;C:\JAVA\JDK8\jre\lib\ext\sunmscapi.jar;C:\JAVA\JDK8\jre\lib\ext\sunpkcs11.jar;C:\JAVA\JDK8\jre\lib\ext\zipfs.jar;C:\JAVA\JDK8\jre\lib\javaws.jar;C:\JAVA\JDK8\jre\lib\jce.jar;C:\JAVA\JDK8\jre\lib\jfr.jar;C:\JAVA\JDK8\jre\lib\jfxswt.jar;C:\JAVA\JDK8\jre\lib\jsse.jar;C:\JAVA\JDK8\jre\lib\management-agent.jar;C:\JAVA\JDK8\jre\lib\plugin.jar;C:\JAVA\JDK8\jre\lib\resources.jar;C:\JAVA\JDK8\jre\lib\rt.jar;D:\新的学习空间\笔记\notes\code\target\classes;C:\Program Files\JetBrains\IntelliJ IDEA 2020.2.3\lib\idea\_rt.jar

类加载器初始化过程：

参见类运行加载全过程图可知其中会创建JVM启动器实例sun.misc.Launcher。

sun.misc.Launcher初始化使用了单例模式设计，保证一个JVM虚拟机内只有一个sun.misc.Launcher实例。

在Launcher构造方法内部，其创建了两个类加载器，分别是

sun.misc.Launcher.ExtClassLoader(扩展类加载器)和sun.misc.Launcher.AppClassLoader(应用类加载器)。

JVM默认使用Launcher的getClassLoader()方法返回的类加载器AppClassLoader的实例加载我们的应用程序。

Launcher类的构造方法

public Launcher() {

Launcher.ExtClassLoader var1;

try { // /构造扩展类加载器，在构造的过程中将其父加载器设置为null

var1 = Launcher.ExtClassLoader.getExtClassLoader();

} catch (IOException var10) {

throw new InternalError("Could not create extension class loader", var10);

}

try {//构造应用类加载器，在构造的过程中将其父加载器设置为ExtClassLoader，

//Launcher的loader属性值是AppClassLoader，我们一般都是用这个类加载器来加载我们自己写的应用程序

this.loader = Launcher.AppClassLoader.getAppClassLoader(var1);

} catch (IOException var9) {

throw new InternalError("Could not create application class loader", var9);

}

Thread.currentThread().setContextClassLoader(this.loader);

String var2 = System.getProperty("java.security.manager");

if (var2 != null) {

SecurityManager var3 = null;

if (!"".equals(var2) && !"default".equals(var2)) {

try {

var3 = (SecurityManager)this.loader.loadClass(var2).newInstance();

} catch (IllegalAccessException var5) {

} catch (InstantiationException var6) {

} catch (ClassNotFoundException var7) {

} catch (ClassCastException var8) {

}

} else {

var3 = new SecurityManager();

}

if (var3 == null) {

throw new InternalError("Could not create SecurityManager: " + var2);

}

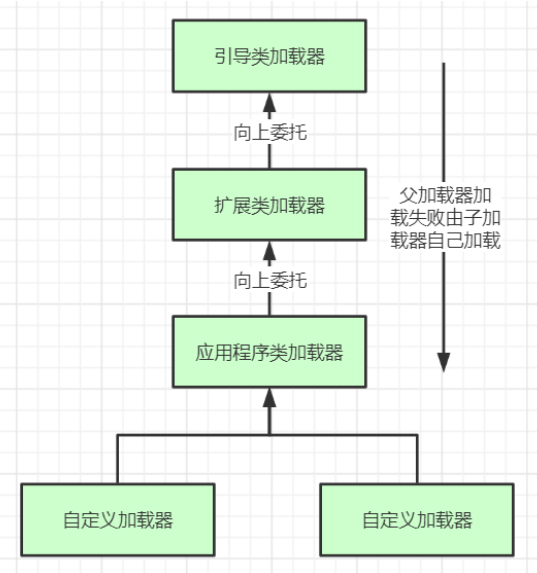
System.setSecurityManager(var3);

}

}

双亲委派机制

JVM类加载器是有亲子层级结构的，如下图



这里类加载其实就有一个双亲委派机制，加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。

比如我们的Math类，最先会找应用程序类加载器加载，应用程序类加载器会先委托扩展类加载器加载，扩展类加载器再委托引导类加载器，顶层引导类加载器在自己的类加载路径里找了半天没找到Math类，则向下退回加载Math类的请求，扩展类加载器收到回复就自己加载，在自己的类加载路径里找了半天也没找到Math类，又向下退回Math类的加载请求给应用程序类加载器，应用程序类加载器于是在自己的类加载路径里找Math类，结果找到了就自己加载了

双亲委派机制说简单点就是，先找父亲加载，不行再由儿子自己加载

我们来看下应用程序类加载器AppClassLoader加载类的双亲委派机制源码，AppClassLoader的loadClass方法最终会调用其父类ClassLoader的loadClass方法，该方法的大体逻辑如下：

1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。
2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）.或者是调用bootstrap类加载器来加载。
3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载。

为什么要设计双亲委派机制？

1. 沙箱安全机制：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改
2. 避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，保证被加载类的唯一性

全盘负责委托机制

“全盘负责”是指当一个ClassLoder装载一个类时，除非显示的使用另外一个ClassLoder，该类所依赖及引用的类也由这个ClassLoder载入。

自定义类加载器示例：

自定义类加载器只需要继承java.lang.ClassLoader类，该类有两个核心方法，一个是loadClass(String,boolean)，实现了双亲委派机制，还有一个方法是findClass，默认实现是空方法，所以我们自定义类加载器主要是重写findClass方法

package com.test;

import java.io.FileInputStream;

import java.lang.reflect.Method;

public class MyClassLoaderTest {

static class MyClassLoader extends ClassLoader {

private String classPath;

public MyClassLoader(String classPath) {

this.classPath = classPath;

}

private byte[] loadByte(String name) throws Exception {

name = name.replaceAll("\\.", "/");

FileInputStream fis = new FileInputStream(classPath + "/" + name + ".class");

int len = fis.available();

byte[] data = new byte[len];

fis.read(data);

fis.close();

return data;

}

protected Class<?> findClass(String name) throws ClassNotFoundException {

try {

byte[] data = loadByte(name);

//defineClass将一个字节数组转为Class对象，这个字节数组是class文件读取后最终的字节数组。

return defineClass(name, data, 0, data.length);

} catch (Exception e) {

e.printStackTrace();

throw new ClassNotFoundException();

}

}

}

public static void main(String args[]) throws Exception {

//初始化自定义类加载器，会先初始化父类ClassLoader，其中会把自定义类加载器的父加载器设置为应用程序类加载器AppClassLoader

MyClassLoader classLoader = new MyClassLoader("D:/test");

//D盘创建 test/com/tuling/jvm 几级目录，将User类的复制类User1.class丢入该目录

Class clazz = classLoader.loadClass("com.test.User1");

Object obj = clazz.newInstance();

Method method = clazz.getDeclaredMethod("sout", null);

method.invoke(obj, null);

System.out.println(clazz.getClassLoader().getClass().getName());

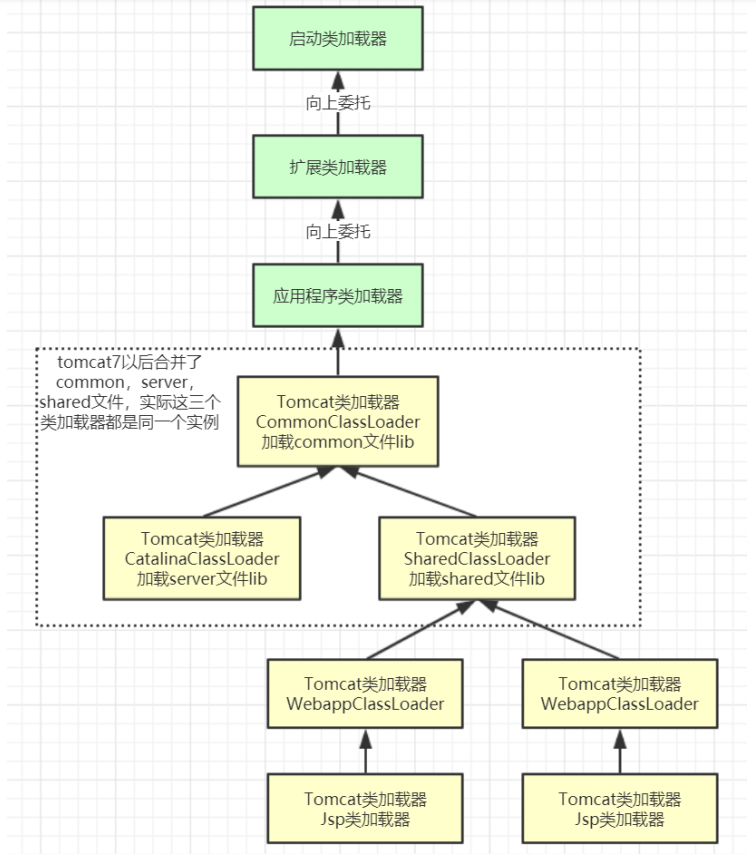
}

}

运行结果：

=======自己的加载器加载类调用方法=======

com.test.MyClassLoaderTest$MyClassLoader



这里说一下Java中静态代码块、构造方法、代码块、父类与子类之间执行顺序及父类子类实例化对象。

先说一下相关的几个概念：

## 1】几个概念

### ① 静态代码块

在java中使用static关键字声明的代码块。每个静态代码块只会执行一次。JVM在加载类时会执行静态代码块，静态代码块先于主方法执行。

static{

System.out.println("这是静态代码块");

}

注意： 静态代码块不能存在于任何方法体内。

### ② 构造代码块(实例初始化块)：

直接在类中定义且没有加static关键字的代码块称为{}构造代码，在创建实例对象的时候先于构造函数被调用执行。

class Test{

int id;

String name;

// JVM加载class时执行

static {

System.out.println("这是静态代码块");

}

// 新建对象的时候执行，构造代码块会先于构造函数被调用时执行

{

this.id = 5;

this.name = "测试";

System.out.println("这是构造代码块");

}

Test(int id) {

this.id = id;

}

public String toString() {

return "name: " + this.name + " , " + "id: " + this.id;

}

}

### ③ 普通代码块

在方法或语句内出现的{}就称为普通代码块。普通代码块和一般的语句执行顺序由他们在代码中出现的次序决定–“先出现先执行”。

public class GeneralCodeBlock01{

public static void main(String[] args){

//如下为普通代码块

{

int x=3;

System.out.println("1,普通代码块内的变量x="+x);

}

int x=1;

System.out.println("主方法内的变量x="+x);

{

int y=7;

System.out.println("2,普通代码块内的变量y="+y);

}

}

}

### 属性、方法、构造方法和代码块都是类中的成员，在创建对象时，各成员的执行顺序如下：

* 父类静态成员和静态初始化块，按在代码中出现的顺序依次执行；
* 子类静态成员和静态初始化块，按在代码中出现的顺序依次执行；
* 父类实例成员和实例初始化块，按在代码中出现的顺序依次执行；
* 执行父类构造方法；
* 子类实例成员和实例初始化块，按在代码中出现的顺序依次执行；
* 执行子类构造方法。

## 【2】父类、子类之间代码块与构造方法

示例代码如下：

package com.web.test2;

public class HelloA {

static{

System.out.println("static A");

}

{System.out.println("I'm A class");}

public HelloA(){

System.out.println("HelloA");

}

public HelloA(String s){

System.out.println(s+"HelloA");

}

public static void main(String[] args) {

new HelloB();

}

}

class HelloB extends HelloA{

public HelloB () {

//只能调用一个父类的构造方法

// super();

super("parent");

System.out.println("HelloB");

}

{System.out.println("I'm B class");}

static{

System.out.println("static B");

}

}

执行结果：

static A

static B

I'm A class

parentHelloA

I'm B class

HelloB

**总结如下：**

① 代码块于构造方法之前执行，静态于非静态之前；

② 在类第一次被调用时，加载该类，静态代码块只执行一次；

③ 项目启动时，只加载需要加载的类(比如xml中显示配置的bean，或者web.xml中的listener等)，并不会将所有的类都加载(这是很可怕的事情)；

④ 静态代码块只能调用静态变量；静态方法只能调用静态变量；

⑤ 非静态代码块或非静态方法可以调用任何(静态+非静态)变量。

⑥ 非静态代码块在实例化对象时，于构造方法之前执行。

## 【3】父类、子类与super()

示例代码如下：

public class People {

String name;

public People() {

System.out.println(1);

}

public People(String name) {

System.out.println(2);

this.name = name;

}

}

class Child extends People{

People father;

public Child () {

//super()系统会默认添加的

System.out.println(4);

}

public Child (String name) {

//super()系统会默认添加的

System.out.println(3);

this.name = name;

father = new People(name+":F");

}

public static void main(String[] args) {

new Child("mike");

}

}

故执行结果：132

## 【4】类中添加静态变量

静态变量与静态代码块一样，都是在类被加载的时候赋值/被执行，而且静态变量与静态代码块执行顺序是按照代码上次次序进行执行的。

示例代码如下：

public class ExA {

//静态成员

private static ExA a = new ExA();

//静态代码块

static {

System.out.println("父类--静态代码块");

}

//构造方法

public ExA() {

System.out.println("父类--构造函数");

}

// 实例化代码块

{

System.out.println("父类--非静态代码块");

}

public static void main(String[] args) {

new ExB();

}

}

class ExB extends ExA {

private static ExB b = new ExB();

static {

System.out.println("子类--静态代码块");

}

{

System.out.println("子类--非静态代码块");

}

public ExB() {

System.out.println("子类--构造函数");

}

}

* result as follows ：

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

父类--非静态代码块

父类--构造函数

子类--非静态代码块

子类--构造函数

子类--静态代码块

父类--非静态代码块

父类--构造函数

子类--非静态代码块

子类--构造函数

**分析如下：**

**① 首先加载父类静态成员和静态代码块**

private static ExA a = new ExA();

static {

System.out.println("父类--静态代码块");

}

这里静态成员赋值为实例化A对象，故而需要先执行A的实例化代码块和构造方法：

public ExA() {

System.out.println("父类--构造函数");

}

// 实例化代码块

{

System.out.println("父类--非静态代码块");

}

此时输入结果为：

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

**② 其次加载子类静态成员和静态代码块**

private static ExB b = new ExB();

static {

System.out.println("子类--静态代码块");

}

子类静态成员赋值为实例化B对象，按照博文最上面子类实例化对象过程可知，此时该加载父类实例化块和构造方法，此时结果更新为：

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

父类--非静态代码块

父类--构造函数

静态变量赋值还没有执行完！！该执行子类实例化代码块和构造方法，此时结果更新为：

# 1. 父类静态成员赋值和静态代码块

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

# 2. 子类静态成员赋值和静态代码块

// 每次调用父类构造方法前都会调用父类实例化代码块

父类--非静态代码块

父类--构造函数这里写代码片

子类--非静态代码块

子类--构造函数

子类B静态变量赋值完，该执行子类静态代码块，此时结果更新如下：

# 1. 父类静态成员赋值和静态代码块

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

# 2. 子类静态成员赋值和静态代码块

// 每次调用父类构造方法前都会调用父类实例化代码块

父类--非静态代码块

父类--构造函数这里写代码片

子类--非静态代码块

子类--构造函数

# 3. 子类静态代码块

子类--静态代码块

**③ 静态完了，该非静态了**

按照如下步骤，该执行第3/4/5/6：

父类静态成员和静态初始化块，按在代码中出现的顺序依次执行；

子类静态成员和静态初始化块，按在代码中出现的顺序依次执行；

父类实例成员和实例初始化块，按在diam中出现的顺序依次执行；

执行父类构造方法；

子类实例成员和实例初始化块，按在代码中出现的顺序依次执行；

执行子类构造方法。

无需多考虑，结果更新如下：

# 1. 父类静态成员赋值和静态代码块

// 父类静态成员

父类--非静态代码块

父类--构造函数

//父类静态代码块

父类--静态代码块

# 2. 子类静态成员赋值和静态代码块

// 每次调用父类构造方法前都会调用父类实例化代码块

父类--非静态代码块

父类--构造函数这里写代码片

子类--非静态代码块

子类--构造函数

# 3. 子类静态代码块

子类--静态代码块

# 4. 父类非静态

父类--非静态代码块

父类--构造函数

# 5. 子类非静态

子类--非静态代码块

子类--构造函数

确实比较难理解，需要特别注意父类和子类静态变量赋值的时候取值为实例化对象，非常量或者null值！！

## 【5】不仅仅父子，还有祖孙

示例如下：

package com.web.test2;

public class Creature {

public Creature(){

System.out.println("空的");

}

public static void main(String[] args) {

new wolf();

}

}

class Animal extends Creature{

public Animal(String name) {

super();

System.out.println("一个参数"+name);

}

public Animal(String name,int age){

//super();错误

this(name);

System.out.println("这个动物带了两个属性"+age);

}

}

class wolf extends Animal{

public wolf(){

super("灰太狼",3);

System.out.println("狼带了三个属性");

}

}

执行结果如下：

空的

一个参数灰太狼

这个动物带了两个属性3

狼带了三个属性

默认会调用父类无参构造方法，如果父类还有parent，则继续向上搜寻，直到Creature。

this()与super()在一个方法中只能存在一个。

## 【6】如果父类没有空参构造器呢？

父类如下所示：

public class Parent {

private Integer id;

private Integer age;

public Parent(Integer id,Integer age){

this.id=id;

this.age=age;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

}

子类代码如下所示：

public class Child extends Parent {

private Integer id;

private String name;

@Override

public void setId(Integer id) {

this.id = id;

}

public void setName(String name) {

this.name = name;

}

@Override

public Integer getId() {

return id;

}

public String getName() {

return name;

}

}

此时，直接报编译错误，提示如下：

There is no default constructor available in 'com.jane.model.Parent'

即，父类没有默认构造器，你需要重写父类构造器。修改代码如下：

public class Child extends Parent {

private Integer id;

private String name;

public Child(Integer id, Integer age) {

super(id, age);

}

public Child(Integer id, Integer age, Integer id1, String name) {

super(id, age);

this.id = id1;

this.name = name;

}

@Override

public void setId(Integer id) {

this.id = id;

}

public void setName(String name) {

this.name = name;

}

@Override

public Integer getId() {

return id;

}

public String getName() {

return name;

}

}

此时子类同样不能存在空参构造器！！

## 【7】子类对象实例化是否会实例化父类对象

答案当然是否定的！！

**首先举反例证明**

如果父类是个抽象类呢？实例化子类的时候实例化父类，很显然矛盾！故而，实例化子类的时候不会实例化父类对象！！

**其次相关理论**

① 构造器只是负责对java对象实例变量执行初始化（也就是赋初始值），在执行构造器代码之前，该对象所占的内存已经被分配下来，这些内存里值都默认是空值——对于基本类型的变量，默认的空值就是0或false，对于引用类型的变量默认的空值就是null。

② 在处理java类中的成员变量时，并不是采用运行时绑定，而是一般意义上的静态绑定，即成员变量被关联到了类上。必须明确，运行时（动态）绑定针对的范畴只是对象的方法。（方法的话只有static和final(所有private默认是final的)是静态绑定的.）

如果虚拟机在子类对象中找不到某个属性的时候，就到包含该构造方法中的类去找静态绑定相应的属性。

③ 当创建任何java对象时，程序总会先一次调用每个类父类非静态初始化块、父类构造器（总是从Object开始）执行初始化，最后才调用本类的非静态初始化块、构造器执行初始化。

具体请参考博文上面讲述过程。

此时当程序执行到①时，系统会先为父类中的私有属性在堆内存开辟空间

注意，这里并不会实例化父类对象，仅仅是为父类中的属性在堆中开辟了一段内存空间。

然后再为子类在堆内存空间，此时调用父类的构造方法（不写super()的话会隐式的调用），此时通过准备知识1中我们已经知道构造方法仅仅只是负责对java对象实例变量执行初始化，而不会实例化父类。

参考博文：[JVM创建对象的奥秘](https://link.zhihu.com/?target=https%3A//blog.csdn.net/J080624/article/details/82116500)

## 【8】子类继承父类什么？

常见的一句话，子类对象只能继承父类非私有的属性及方法；还有另外一句话–子类继承父类，子类拥有父类所有的属性和方法。

是否矛盾？该怎样理解？先看看第【7】部分，然后继续往下。

使用程序去验证，发现父类的私有属性和私有方法，子类是不能访问的，当然一些父类的私有属性可能可以通过相应的方法访问到，但是私有的方法似乎不能简单的访问，这里暂不考虑Java反射机制。

子类不能通过object.field的方式获取父类私有属性，这说明不属于子类成员变量！但是子类可以使用父类的public方法为父类的私有属性赋值并获取该私有属性值！这可以理解为子类拥有父类私有属性。