**1 java类加载时机**

**引起类加载的场景**

* 1 使用new创建对象时
* 2 读取或设置类的静态变量时(编译期常量除外)
* 3 使用java.lang.reflect包中方法对类进行反射调用时
* 4 初始化一个类时，会先初始化其父类，接口例外
* 5 虚拟机启动的主类，也就是定义main()方法的那个类，会在虚拟机启动就初始化

**不会引起类加载的场景**

public static void main(String[] args) {

Child[] children = new Child[250];

}

public static class Child extends Base {

static {

System.out.println("Child init");

}

}

1 **对于静态字段，只有直接定义这个静态字段的类才会被初始化，通过子类引用不会导致子类被初始化**，比如下面代码

public static void main(String[] args) {

System.out.println(Child.NAME);

}

public static class Base {

public static String NAME = "NAME";

static {

System.out.println("Base init");

}

}

public static class Child extends Base {

static {

System.out.println("Child init");

}

}

打印结果为:

Base init

NAME

* 2 **通过使用数组定义来引用类，不会触发类的初始化**，比如:

public static void main(String[] args) {

Child[] children = new Child[250];

}

public static class Child extends Base {

static {

System.out.println("Child init");

}

}

结果毛都没打印

* **3 引用类的编译期常量不会触发类的初始化**

先来解释什么叫类的编译期常量:

* 第一: 类的编译期常量必须用static修饰，因为非static的在编译期都不能访问，必须要new出来对象才行，而new对象就出发了类的初始化，所以static对应了“编译期常量”中“编译期”这三个字
* 第二: 编译期常量必须用final修饰，这对应了“编译期常量”中“常量”这两个字

那么以static final 修饰的就一定是编译期常量吗？错！比如:

public static final long time = 74110; //这是个编译期常量

public static final long time = System.currentTimeMillis(); //不是编译期常量，因为系统时间只有在运行时才知道，编译期知道个毛啊

我们用代码来验证:

public static void main(String[] args) {

System.out.println(Init.time);

}

public static class Init {

public static final long time = 74110;

static {

System.out.println("Init被初始化!");

}

}

运行结果:

74110

可以看到，并没有引起类的初始化! 这是正常的，因为编译期常量在编译期就被放入常量池，后面访问这个变量都会在常量池找，跟类半毛钱关系都没有，所以不会引起初始化。 接着来看第二个例子:

public static void main(String[] args) {

System.out.println(Init.time);

}

public static class Init {

public static final long time = System.currentTimeMillis();

static {

System.out.println("Init被初始化!");

}

}

运行结果:

Init被初始化!

1596355938901

可以看到，类会先被初始化! **所以编译期常量的第三个要素变量的值需要在编译期就知道**，那么，可以总结一下编译期常量的定义:**static final 同时修饰的并且编译期就知道的才是编译期常量**

Tips: 要知道一个类的变量是不是编译期常量，可以先用javac得到.class文件，然后通过javap -verbose xxx.class来直接查看jvm字节码，如果被ConstantValue修饰，就是编译期常量，比如:

public static class Init {

public static final long time = 74110;

}

对应的jvm指令码，这里只贴出一部分:

{

public static final long time; //表示有个变量叫time

descriptor: I

flags: (0x0019) ACC\_PUBLIC, ACC\_STATIC, ACC\_FINAL //分别对应3个标记:public, static, final

ConstantValue: int 74110 // 表示是编译期常量

public Init();

descriptor: ()V

flags: (0x0001) 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 3: 0

}

如果将time的值改成当前系统时间呢:

public static class Init {

public static final long time = System.currentTimeMillis();

}

对应的jvm字节码如下，这里只贴出部分:

{

public static final long time; //表示有个变量叫time

descriptor: J

flags: (0x0019) ACC\_PUBLIC, ACC\_STATIC, ACC\_FINAL //对应public static final

//发现没有ConstantValue修饰符了

public Hello();

descriptor: ()V

flags: (0x0001) 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 3: 0

static {};

descriptor: ()V

flags: (0x0008) ACC\_STATIC

Code:

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

0: invokestatic #2 // Method java/lang/System.currentTimeMillis:()J

3: putstatic #3 // Field time:J

6: return

LineNumberTable:

line 4: 0

}

**例如 public final int a = 10;这样的a，也是编译期常量，但不是类的，是对象的，需要先创建对象才能使用，所以会引起类加载。**

* **4 接口初始化之前，不会引起父接口的初始化，除非真正用到了父接口的成员(比如成员类)**

因为接口不能定义static块，所以没办法从api层验证，后续我们会从jvm字节码指令来分析验证。

**2 java类加载机制**

满足了类加载时机的条件后就会触发类加载机制

java类加载分为5个步骤: 加载、连接(验证、准备、解析)、初始化、使用、卸载，接下来我们来详细讲解加载、连接和初始化，至于使用，卸载就不废话了

* 1 加载

**加载阶段完成的事情**:

* (1) 通过一个类的全限定名获取定义这个类的二进制字节流；
* (2) 将二进制流转化为方法区的运行时数据结构
* (3) 使用这个结构在内存中生成一个java.lang.Class对象用来作为这个类的访问入口

**可以简单理解为:通过一个类的全限定名在方法区生成一个java.lang.Class对象**

 2 连接(连接阶段拆分为3个阶段)

* 验证: 验证加载阶段Class文件是否合法，比如是否以魔数开头，版本号是否在当前虚拟机的处理范围之内等。
* 准备: 为类变量分配内存并设置初始值，注意是“类变量”，也就是static变量，所以都在方法区分配，这些初始值一般都是“零值”，比如对象的零值是null，int的零值是0，boolean的零值是false等，但是如果是“编译期常量”，则直接就是定义的初始值。
* 解析: 将符号引用转化为直接引用的过程，会确定部分方法的版本

3 初始化: 执行<clinit>()方法的过程。

<clinit>方法是由jvm收集类中所以类变量的“赋值语句”和“static块”得到的，也就是说，如果没有类变量的赋值语句和static块，就不会有<clinit>块，看例子:

public class Hello {

public static final int a = 100;

}

然后用javap -verbose Hello.class查看字节码:

Classfile /Users/lloydfinch/venn/workspace/java/test/Hello.class //路径

Last modified 2020年8月2日; size 229 bytes //修改时间和大小

MD5 checksum 415f32c281d3178ff83100e89e1d092d //校验码

Compiled from "Hello.java" //源文件

public class Hello

minor version: 0 //支持的最低版本号，45对应jdk1.0，之后每次版本号升高就加1

major version: 55 //支持的最高版本号，55-45 = 10,所以对应jdk 11

flags: (0x0021) ACC\_PUBLIC, ACC\_SUPER

this\_class: #2 // Hello

super\_class: #3 // java/lang/Object

interfaces: 0, fields: 1, methods: 1, attributes: 1

Constant pool:

#1 = Methodref #3.#14 // java/lang/Object."<init>":()V

#2 = Class #15 // Hello

#3 = Class #16 // java/lang/Object

#4 = Utf8 a

#5 = Utf8 I

#6 = Utf8 ConstantValue

#7 = Integer 10

#8 = Utf8 <init> //实例构造器

#9 = Utf8 ()V

#10 = Utf8 Code

#11 = Utf8 LineNumberTable

#12 = Utf8 SourceFile

#13 = Utf8 Hello.java

#14 = NameAndType #8:#9 // "<init>":()V

#15 = Utf8 Hello

#16 = Utf8 java/lang/Object

{

public static final int a;

descriptor: I

flags: (0x0019) ACC\_PUBLIC, ACC\_STATIC, ACC\_FINAL

ConstantValue: int 10

public Hello();

descriptor: ()V

flags: (0x0001) 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 3: 0

}

SourceFile: "Hello.java"

我们发现上面并没有<clinit>方法，因为a是个编译期常量，所以并没有，然后我们改成:

public class Hello {

public static int a = 100; //去掉final，那么就等价于赋值语句，因为有final的话，不是赋值语句，而是“初始化语句”

}

对应的字节码指令:

Classfile /Users/lloydfinch/venn/workspace/java/test/Hello.class

Last modified 2020年8月2日; size 265 bytes

MD5 checksum d45f4426aa6b31b54300f59966e46049

Compiled from "Hello.java"

public class Hello

minor version: 0

major version: 55

flags: (0x0021) ACC\_PUBLIC, ACC\_SUPER

this\_class: #3 // Hello

super\_class: #4 // java/lang/Object

interfaces: 0, fields: 1, methods: 2, attributes: 1

Constant pool:

#1 = Methodref #4.#14 // java/lang/Object."<init>":()V

#2 = Fieldref #3.#15 // Hello.a:I

#3 = Class #16 // Hello

#4 = Class #17 // java/lang/Object

#5 = Utf8 a

#6 = Utf8 I

#7 = Utf8 <init>

#8 = Utf8 ()V

#9 = Utf8 Code

#10 = Utf8 LineNumberTable

#11 = Utf8 <clinit> //注意这里，多出来了<clinit>方法

#12 = Utf8 SourceFile

#13 = Utf8 Hello.java

#14 = NameAndType #7:#8 // "<init>":()V

#15 = NameAndType #5:#6 // a:I

#16 = Utf8 Hello

#17 = Utf8 java/lang/Object

{

public static int a;

descriptor: I

flags: (0x0009) ACC\_PUBLIC, ACC\_STATIC

public Hello();

descriptor: ()V

flags: (0x0001) 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 3: 0

static {};

descriptor: ()V

flags: (0x0008) ACC\_STATIC

Code:

stack=1, locals=0, args\_size=0

0: bipush 10

2: putstatic #2 // Field a:I

5: return

LineNumberTable:

line 4: 0

}

SourceFile: "Hello.java"

**<clinit>()方法是根据语句在源文件中出现的顺序生成的，静态语句块只能访问定义在它之前的变量，定义在它之后的，只能赋值不能访问!**

public static class Init {

public static int a = 10;

static {

a = 20; //对

System.out.println(a);//对

b = 10;//对

System.out.println(b);//错，静态语句块不能访问定义在它之后的变量

}

public static int b = 20;

}

jvm会在子类的<clinit>()执行之前自动调用父类的<clinit>()方法，这就意味着父类的静态语句优先于子类赋值变量语句执行，所以java.lang.Object的<clinit>()方法总是第一个被调用

public static void main(String[] args) {

System.out.println(Child.b);

}

public static class Child extends Base {

public static int b = a;

static {

System.out.println("Child Init");

}

}

public static class Base {

public static int a = 10;

static {

System.out.println("赋值为10");

a = 20;

}

}

运行结果:

赋值为10

Child Init

20

<clinit>()方法对类或接口不是必须的，如果类中没有类变量的赋值语句或静态块，就不会有，接口的<clinit>()方法调用前不会先调用父接口的<clinit>()方法，除非父接口定义的变量使用时，才会初始化

jvm会保证<clinit>()方法在多线程中被正确的加锁、同步，可以使用这个特性来实现单例模式，也就是静态内部类单例。比如:

public class SingleInstance {

private static SingleInstance instance;

private SingleInstance() {

}

public static SingleInstance getInstance() {

return Inner.instance;

}

private static class Inner {

//因为这是个静态变量的赋值语句，所以在<clinit>()中，而jvm保护了<clinit>()被正确的加锁、同步，所以是线程安全的

private static SingleInstance instance = new SingleInstance();

}

}

### 方法调用

在“连接”阶段的“解析阶段”，我们会确定一部分方法的版本，比如重载的版本，来看例子:

public static void main(String[] args) {

TestClass testClass = new TestClass();

Base base = new Child();

testClass.info(base);

}

public static class Child extends Base {

}

public static class Base {

}

public void info(Base base) {

System.out.println("info base");

}

public void info(Child child) {

System.out.println("info child");

}

运行结果:

info base

也就是说，**函数的重载是在编译期就确定的，在jvm里面叫“静态分派”**

看另一个例子:

public static void main(String[] args) {

Base base = new Child();

base.info();

}

public static class Child extends Base {

@Override

public void info() {

System.out.println("Child");

}

}

public static class Base {

public void info() {

System.out.println("Base");

}

}

运行结果:

child

**函数的重写在jvm里是“动态分派”**

**总结**

* 1 编译期常量是static final修饰的在编译期就能确定其值的变量，会在jvm指令中ConstantValue标记
* 2 准备阶段就会为类变量分配内存并赋初值，如果是编译期常量，则直接就是指定的值，否则就是零值
* 3 <clinit>()方法会保证父类先执行，并且保证线程安全，可以用来实现静态内部类单例
* 4 方法的重载是静态分配的，方法的重写是动态分配的
* 5 类变量有两个赋值阶段，一次是准备阶段，一次是初始化阶段，编译期常量准备阶段就被正确的赋值，非编辑期常量在初始化阶段才会被正确赋值