**深入理解Java对象的创建过程：类的初始化与实例化**

<https://blog.csdn.net/justloveyou_/article/details/72466416>

摘要：

　　在Java中，一个对象在可以被使用之前必须要被正确地初始化，这一点是Java规范规定的。在实例化一个对象时，JVM首先会检查相关类型是否已经加载并初始化，如果没有，则JVM立即进行加载并调用类构造器完成类的初始化。在类初始化过程中或初始化完毕后，根据具体情况才会去对类进行实例化。本文试图对JVM执行类初始化和实例化的过程做一个详细深入地介绍，以便从Java虚拟机的角度清晰解剖一个Java对象的创建过程。

版权声明：

本文原创作者：书呆子Rico

作者博客地址：http://blog.csdn.net/justloveyou\_/

友情提示：

　　一个Java对象的创建过程往往包括 类初始化 和 类实例化 两个阶段。本文的姊妹篇《 JVM类加载机制概述：加载时机与加载过程》主要介绍了类的初始化时机和初始化过程，本文在此基础上，进一步阐述了一个Java对象创建的真实过程。

一、Java对象创建时机

　　我们知道，一个对象在可以被使用之前必须要被正确地实例化。在Java代码中，有很多行为可以引起对象的创建，最为直观的一种就是使用new关键字来调用一个类的构造函数显式地创建对象，这种方式在Java规范中被称为 : 由执行类实例创建表达式而引起的对象创建。除此之外，我们还可以使用反射机制(Class类的newInstance方法、使用Constructor类的newInstance方法)、使用Clone方法、使用反序列化等方式创建对象。下面笔者分别对此进行一一介绍：

1). 使用new关键字创建对象

　　这是我们最常见的也是最简单的创建对象的方式，通过这种方式我们可以调用任意的构造函数（无参的和有参的）去创建对象。比如：

　　Student student = new Student();

1

2). 使用Class类的newInstance方法(反射机制)

　　我们也可以通过Java的反射机制使用Class类的newInstance方法来创建对象，事实上，这个newInstance方法调用无参的构造器创建对象，比如：

　　Student student2 = (Student)Class.forName("Student类全限定名").newInstance();

或者：

　　Student stu = Student.class.newInstance();

1

2

3

3). 使用Constructor类的newInstance方法(反射机制)

　　java.lang.relect.Constructor类里也有一个newInstance方法可以创建对象，该方法和Class类中的newInstance方法很像，但是相比之下，Constructor类的newInstance方法更加强大些，我们可以通过这个newInstance方法调用有参数的和私有的构造函数，比如：

public class Student {

private int id;

public Student(Integer id) {

this.id = id;

}

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

Constructor<Student> constructor = Student.class

.getConstructor(Integer.class);

Student stu3 = constructor.newInstance(123);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

　　使用newInstance方法的这两种方式创建对象使用的就是Java的反射机制，事实上Class的newInstance方法内部调用的也是Constructor的newInstance方法。

4). 使用Clone方法创建对象

　　无论何时我们调用一个对象的clone方法，JVM都会帮我们创建一个新的、一样的对象，特别需要说明的是，用clone方法创建对象的过程中并不会调用任何构造函数。关于如何使用clone方法以及浅克隆/深克隆机制，笔者已经在博文《 Java String 综述(下篇)》做了详细的说明。简单而言，要想使用clone方法，我们就必须先实现Cloneable接口并实现其定义的clone方法，这也是原型模式的应用。比如：

public class Student implements Cloneable{

private int id;

public Student(Integer id) {

this.id = id;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// TODO Auto-generated method stub

return super.clone();

}

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

Constructor<Student> constructor = Student.class

.getConstructor(Integer.class);

Student stu3 = constructor.newInstance(123);

Student stu4 = (Student) stu3.clone();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

5). 使用(反)序列化机制创建对象

　　当我们反序列化一个对象时，JVM会给我们创建一个单独的对象，在此过程中，JVM并不会调用任何构造函数。为了反序列化一个对象，我们需要让我们的类实现Serializable接口，比如：

public class Student implements Cloneable, Serializable {

private int id;

public Student(Integer id) {

this.id = id;

}

@Override

public String toString() {

return "Student [id=" + id + "]";

}

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

Constructor<Student> constructor = Student.class

.getConstructor(Integer.class);

Student stu3 = constructor.newInstance(123);

// 写对象

ObjectOutputStream output = new ObjectOutputStream(

new FileOutputStream("student.bin"));

output.writeObject(stu3);

output.close();

// 读对象

ObjectInputStream input = new ObjectInputStream(new FileInputStream(

"student.bin"));

Student stu5 = (Student) input.readObject();

System.out.println(stu5);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

6). 完整实例

public class Student implements Cloneable, Serializable {

private int id;

public Student() {

}

public Student(Integer id) {

this.id = id;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// TODO Auto-generated method stub

return super.clone();

}

@Override

public String toString() {

return "Student [id=" + id + "]";

}

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

System.out.println("使用new关键字创建对象：");

Student stu1 = new Student(123);

System.out.println(stu1);

System.out.println("\n---------------------------\n");

System.out.println("使用Class类的newInstance方法创建对象：");

Student stu2 = Student.class.newInstance(); //对应类必须具有无参构造方法，且只有这一种创建方式

System.out.println(stu2);

System.out.println("\n---------------------------\n");

System.out.println("使用Constructor类的newInstance方法创建对象：");

Constructor<Student> constructor = Student.class

.getConstructor(Integer.class); // 调用有参构造方法

Student stu3 = constructor.newInstance(123);

System.out.println(stu3);

System.out.println("\n---------------------------\n");

System.out.println("使用Clone方法创建对象：");

Student stu4 = (Student) stu3.clone();

System.out.println(stu4);

System.out.println("\n---------------------------\n");

System.out.println("使用(反)序列化机制创建对象：");

// 写对象

ObjectOutputStream output = new ObjectOutputStream(

new FileOutputStream("student.bin"));

output.writeObject(stu4);

output.close();

// 读取对象

ObjectInputStream input = new ObjectInputStream(new FileInputStream(

"student.bin"));

Student stu5 = (Student) input.readObject();

System.out.println(stu5);

}

}/\* Output:

使用new关键字创建对象：

Student [id=123]

---------------------------

使用Class类的newInstance方法创建对象：

Student [id=0]

---------------------------

使用Constructor类的newInstance方法创建对象：

Student [id=123]

---------------------------

使用Clone方法创建对象：

Student [id=123]

---------------------------

使用(反)序列化机制创建对象：

Student [id=123]

\*///:~

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

　　从Java虚拟机层面看，除了使用new关键字创建对象的方式外，其他方式全部都是通过转变为invokevirtual指令直接创建对象的。

二. Java 对象的创建过程

　　当一个对象被创建时，虚拟机就会为其分配内存来存放对象自己的实例变量及其从父类继承过来的实例变量(即使这些从超类继承过来的实例变量有可能被隐藏也会被分配空间)。在为这些实例变量分配内存的同时，这些实例变量也会被赋予默认值(零值)。在内存分配完成之后，Java虚拟机就会开始对新创建的对象按照程序猿的意志进行初始化。在Java对象初始化过程中，主要涉及三种执行对象初始化的结构，分别是 实例变量初始化、实例代码块初始化 以及 构造函数初始化。

1、实例变量初始化与实例代码块初始化

　　我们在定义（声明）实例变量的同时，还可以直接对实例变量进行赋值或者使用实例代码块对其进行赋值。如果我们以这两种方式为实例变量进行初始化，那么它们将在构造函数执行之前完成这些初始化操作。实际上，如果我们对实例变量直接赋值或者使用实例代码块赋值，那么编译器会将其中的代码放到类的构造函数中去，并且这些代码会被放在对超类构造函数的调用语句之后(还记得吗？Java要求构造函数的第一条语句必须是超类构造函数的调用语句)，构造函数本身的代码之前。例如：

public class InstanceVariableInitializer {

private int i = 1;

private int j = i + 1;

public InstanceVariableInitializer(int var){

System.out.println(i);

System.out.println(j);

this.i = var;

System.out.println(i);

System.out.println(j);

}

{ // 实例代码块

j += 3;

}

public static void main(String[] args) {

new InstanceVariableInitializer(8);

}

}/\* Output:

1

5

8

5

\*///:~

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

　　上面的例子正好印证了上面的结论。特别需要注意的是，Java是按照编程顺序来执行实例变量初始化器和实例初始化器中的代码的，并且不允许顺序靠前的实例代码块初始化在其后面定义的实例变量，比如：

public class InstanceInitializer {

{

j = i;

}

private int i = 1;

private int j;

}

public class InstanceInitializer {

private int j = i;

private int i = 1;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

　　上面的这些代码都是无法通过编译的，编译器会抱怨说我们使用了一个未经定义的变量。之所以要这么做是为了保证一个变量在被使用之前已经被正确地初始化。但是我们仍然有办法绕过这种检查，比如：

public class InstanceInitializer {

private int j = getI();

private int i = 1;

public InstanceInitializer() {

i = 2;

}

private int getI() {

return i;

}

public static void main(String[] args) {

InstanceInitializer ii = new InstanceInitializer();

System.out.println(ii.j);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

　　如果我们执行上面这段代码，那么会发现打印的结果是0。因此我们可以确信，变量j被赋予了i的默认值0，这一动作发生在实例变量i初始化之前和构造函数调用之前。

2、构造函数初始化

　　我们可以从上文知道，实例变量初始化与实例代码块初始化总是发生在构造函数初始化之前，那么我们下面着重看看构造函数初始化过程。众所周知，每一个Java中的对象都至少会有一个构造函数，如果我们没有显式定义构造函数，那么它将会有一个默认无参的构造函数。在编译生成的字节码中，这些构造函数会被命名成<init>()方法，参数列表与Java语言书写的构造函数的参数列表相同。

　　我们知道，Java要求在实例化类之前，必须先实例化其超类，以保证所创建实例的完整性。事实上，这一点是在构造函数中保证的：Java强制要求Object对象(Object是Java的顶层对象，没有超类)之外的所有对象构造函数的第一条语句必须是超类构造函数的调用语句或者是类中定义的其他的构造函数，如果我们既没有调用其他的构造函数，也没有显式调用超类的构造函数，那么编译器会为我们自动生成一个对超类构造函数的调用，比如：

public class ConstructorExample {

}

1

2

3

　　对于上面代码中定义的类，我们观察编译之后的字节码，我们会发现编译器为我们生成一个构造函数，如下，

aload\_0

invokespecial #8; //Method java/lang/Object."<init>":()V

return

1

2

3

　　上面代码的第二行就是调用Object类的默认构造函数的指令。也就是说，如果我们显式调用超类的构造函数，那么该调用必须放在构造函数所有代码的最前面，也就是必须是构造函数的第一条指令。正因为如此，Java才可以使得一个对象在初始化之前其所有的超类都被初始化完成，并保证创建一个完整的对象出来。

　　特别地，如果我们在一个构造函数中调用另外一个构造函数，如下所示，

public class ConstructorExample {

private int i;

ConstructorExample() {

this(1);

....

}

ConstructorExample(int i) {

....

this.i = i;

....

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

　　对于这种情况，Java只允许在ConstructorExample(int i)内调用超类的构造函数，也就是说，下面两种情形的代码编译是无法通过的：

public class ConstructorExample {

private int i;

ConstructorExample() {

super();

this(1); // Error:Constructor call must be the first statement in a constructor

....

}

ConstructorExample(int i) {

....

this.i = i;

....

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

或者，

public class ConstructorExample {

private int i;

ConstructorExample() {

this(1);

super(); //Error: Constructor call must be the first statement in a constructor

....

}

ConstructorExample(int i) {

this.i = i;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

　　Java通过对构造函数作出这种限制以便保证一个类的实例能够在被使用之前正确地初始化。

3、 小结

　　总而言之，实例化一个类的对象的过程是一个典型的递归过程，如下图所示。进一步地说，在实例化一个类的对象时，具体过程是这样的：

　　在准备实例化一个类的对象前，首先准备实例化该类的父类，如果该类的父类还有父类，那么准备实例化该类的父类的父类，依次递归直到递归到Object类。此时，首先实例化Object类，再依次对以下各类进行实例化，直到完成对目标类的实例化。具体而言，在实例化每个类时，都遵循如下顺序：先依次执行实例变量初始化和实例代码块初始化，再执行构造函数初始化。也就是说，编译器会将实例变量初始化和实例代码块初始化相关代码放到类的构造函数中去，并且这些代码会被放在对超类构造函数的调用语句之后，构造函数本身的代码之前。

　　Ps: 关于递归的思想与内涵的介绍，请参见我的博文《 算法设计方法：递归的内涵与经典应用》。

4、实例变量初始化、实例代码块初始化以及构造函数初始化综合实例

　　笔者在《 JVM类加载机制概述：加载时机与加载过程》一文中详细阐述了类初始化时机和初始化过程，并在文章的最后留了一个悬念给各位，这里来揭开这个悬念。建议读者先看完《 JVM类加载机制概述：加载时机与加载过程》这篇再来看这个，印象会比较深刻，如若不然，也没什么关系~~

//父类

class Foo {

int i = 1;

Foo() {

System.out.println(i); -----------(1)

int x = getValue();

System.out.println(x); -----------(2)

}

{

i = 2;

}

protected int getValue() {

return i;

}

}

//子类

class Bar extends Foo {

int j = 1;

Bar() {

j = 2;

}

{

j = 3;

}

@Override

protected int getValue() {

return j;

}

}

public class ConstructorExample {

public static void main(String... args) {

Bar bar = new Bar();

System.out.println(bar.getValue()); -----------(3)

}

}/\* Output:

2

0

2

\*///:~

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

　　根据上文所述的类实例化过程，我们可以将Foo类的构造函数和Bar类的构造函数等价地分别变为如下形式：

//Foo类构造函数的等价变换：

Foo() {

i = 1;

i = 2;

System.out.println(i);

int x = getValue();

System.out.println(x);

}

1

2

3

4

5

6

7

8

//Bar类构造函数的等价变换

Bar() {

Foo();

j = 1;

j = 3;

j = 2

}

1

2

3

4

5

6

7

　　这样程序就好看多了，我们一眼就可以观察出程序的输出结果。在通过使用Bar类的构造方法new一个Bar类的实例时，首先会调用Foo类构造函数，因此(1)处输出是2，这从Foo类构造函数的等价变换中可以直接看出。(2)处输出是0，为什么呢？因为在执行Foo的构造函数的过程中，由于Bar重载了Foo中的getValue方法，所以根据Java的多态特性可以知道，其调用的getValue方法是被Bar重载的那个getValue方法。但由于这时Bar的构造函数还没有被执行，因此此时j的值还是默认值0，因此(2)处输出是0。最后，在执行(3)处的代码时，由于bar对象已经创建完成，所以此时再访问j的值时，就得到了其初始化后的值2，这一点可以从Bar类构造函数的等价变换中直接看出。

三. 类的初始化时机与过程

　　关于类的初始化时机，笔者在博文《 JVM类加载机制概述：加载时机与加载过程》已经介绍的很清楚了，此处不再赘述。简单地说，在类加载过程中，准备阶段是正式为类变量(static 成员变量)分配内存并设置类变量初始值（零值）的阶段，而初始化阶段是真正开始执行类中定义的java程序代码(字节码)并按程序猿的意图去初始化类变量的过程。更直接地说，初始化阶段就是执行类构造器<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代码块static{}中的语句合并产生的，其中编译器收集的顺序是由语句在源文件中出现的顺序所决定。

　　类构造器<clinit>()与实例构造器<init>()不同，它不需要程序员进行显式调用，虚拟机会保证在子类类构造器<clinit>()执行之前，父类的类构造<clinit>()执行完毕。由于父类的构造器<clinit>()先执行，也就意味着父类中定义的静态代码块/静态变量的初始化要优先于子类的静态代码块/静态变量的初始化执行。特别地，类构造器<clinit>()对于类或者接口来说并不是必需的，如果一个类中没有静态代码块，也没有对类变量的赋值操作，那么编译器可以不为这个类生产类构造器<clinit>()。此外，在同一个类加载器下，一个类只会被初始化一次，但是一个类可以任意地实例化对象。也就是说，在一个类的生命周期中，类构造器<clinit>()最多会被虚拟机调用一次，而实例构造器<init>()则会被虚拟机调用多次，只要程序员还在创建对象。

　　注意，这里所谓的实例构造器<init>()是指收集类中的所有实例变量的赋值动作、实例代码块和构造函数合并产生的，类似于上文对Foo类的构造函数和Bar类的构造函数做的等价变换。

四. 总结

　　1、一个实例变量在对象初始化的过程中会被赋值几次？

　　我们知道，JVM在为一个对象分配完内存之后，会给每一个实例变量赋予默认值，这个时候实例变量被第一次赋值，这个赋值过程是没有办法避免的。如果我们在声明实例变量x的同时对其进行了赋值操作，那么这个时候，这个实例变量就被第二次赋值了。如果我们在实例代码块中，又对变量x做了初始化操作，那么这个时候，这个实例变量就被第三次赋值了。如果我们在构造函数中，也对变量x做了初始化操作，那么这个时候，变量x就被第四次赋值。也就是说，在Java的对象初始化过程中，一个实例变量最多可以被初始化4次。

　　2、类的初始化过程与类的实例化过程的异同？

　　类的初始化是指类加载过程中的初始化阶段对类变量按照程序猿的意图进行赋值的过程；而类的实例化是指在类完全加载到内存中后创建对象的过程。

　　3、假如一个类还未加载到内存中，那么在创建一个该类的实例时，具体过程是怎样的？

　　我们知道，要想创建一个类的实例，必须先将该类加载到内存并进行初始化，也就是说，类初始化操作是在类实例化操作之前进行的，但并不意味着：只有类初始化操作结束后才能进行类实例化操作。例如，笔者在博文《 JVM类加载机制概述：加载时机与加载过程》中所提到的下面这个经典案例：

public class StaticTest {

public static void main(String[] args) {

staticFunction();

}

static StaticTest st = new StaticTest();

static { //静态代码块

System.out.println("1");

}

{ // 实例代码块

System.out.println("2");

}

StaticTest() { // 实例构造器

System.out.println("3");

System.out.println("a=" + a + ",b=" + b);

}

public static void staticFunction() { // 静态方法

System.out.println("4");

}

int a = 110; // 实例变量

static int b = 112; // 静态变量

}/\* Output:

2

3

a=110,b=0

1

4

\*///:~

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

　　大家能得到正确答案吗？笔者已经在博文《 JVM类加载机制概述：加载时机与加载过程》中解释过这个问题了，此不赘述。

　　总的来说，类实例化的一般过程是：父类的类构造器<clinit>() -> 子类的类构造器<clinit>() -> 父类的成员变量和实例代码块 -> 父类的构造函数 -> 子类的成员变量和实例代码块 -> 子类的构造函数。

五. 更多

　　更多关于类初始化时机和初始化过程的介绍，请参见我的博文《 JVM类加载机制概述：加载时机与加载过程》。

　　更多关于类加载器等方面的内容，包括JVM预定义的类加载器、双亲委派模型等知识点，请参见我的转载博文《深入理解Java类加载器(一)：Java类加载原理解析》。

　　关于递归的思想与内涵的介绍，请参见我的博文《 算法设计方法：递归的内涵与经典应用》。

---------------------

作者：书呆子Rico

来源：CSDN

原文：https://blog.csdn.net/justloveyou\_/article/details/72466416

版权声明：本文为博主原创文章，转载请附上博文链接！