链滴

Java开发中，初始化和重写有哪些隐患？

作者：balala
原文链接：https://ld246.com/article/1446107644457
来源网站：链滴

在Java开发中，我们常常都需要用到重写方法和初始化方法，但在程序实现过程中，重写和初始化的患大家知道多少呢？今天小编就这隐患为大家分享一二。

虽然本文是针对Java语言的重写和初始化讲解，但是对于所有的面向对象程序设计语言的初始化都适，废话不多说，直接进入正题。

问题

首先我们通过代码来看一个问题，有一个类SuperClass

```
public class SuperClass {

private int mSuperX;

public SuperClass() {

    setX(99);

}

public void setX(int x) {

   mSuperX = x;

}

}
```

现在我们想随时知道mSuperX是什么值, 不用反射, 因为父类从不直接修改mSuperX的值, 总是通过seX来改, 那么最简单的方法就是继承SuperClass, 重写setX方法, 监听它的改变就可以了。下面就是子类ubClass:

```
public class SubClass extends SuperClass {

private int mSubX = 1;

public SubClass() {}

@Override

public void setX(int x) {

    super.setX(x);

    mSubX = x;

    System.out.println("SubX is assigned " + x);

}

public void printX() {
```

```
    System.out.println("SubX = " + mSubX);
}
}
```

使用mSubX来跟踪mSuperX

因为在ViewGroup中, clipToPadding默认值是true(为了简化问题, 把它当成boolean, 实际并不是), ViewGroup初始化有可能不调用setClipToPadding, 此时是默认值, 为了模拟这种情况, 将mSubX初化为1.

最后在main里调用:

```
public class Main {

public static void main(String[] args) {
    SubClass sc = new SubClass();
    sc.printX();
}
}
```

那么问题来了，终端输出的结果是什么呢？相信很多人,都 认为终端输出的结果应该是:

SubX is assigned 99

SubX = 99

其实，真正运行后输出的是:

SubX is assigned 99

SubX = 1

实际分析

是不是很想知道，到底发生了什么？最简单的方法就是看程序到底是怎么执行的，比如单步调试，者直接一点，看看Java字节码。

下面是Main的字节码

```
Compiled from "Main.java"
public class bugme.Main {
......
public static void main(java.lang.String[]);
```

```
Code:
  0: new           #2                  // class bugme/SubClass
  3: dup
  4: invokespecial #3                  // Method bugme/SubClass."<init>":()V
  ......
}
```

这是直接用javap反编译.class文件得到的。虽说同样是Java语言写的, 用apktool反编译APK文件(其的dex文件)得到的smali代码和Java Bytecode明显长得不一样（字节码，隐含了一个栈和局部变量表。

这段代码首先new一个SubClass实例, 把引用入栈, dup是把栈顶复制一份入栈, invokespecial #3将顶元素出栈并调用它的某个方法, 这个方法具体是什么要看常量池里第3个条目是什么, 但是javap生成字节码直接给我们写在旁边了, 即SubClass.<init>.

接下来看SubClass.<init>,

```
public class bugme.SubClass extends bugme.SuperClass {
public bugme.SubClass();

Code:
  0: aload_0
  1: invokespecial #1                  // Method bugme/SuperClass."<init>":()V
  ......
```

这里面并没有方法叫<init>, 是因为javap为了方便我们阅读, 直接把它改成类名bugme.SubClass, 顺说明一下, bugme是包名。 <init>方法并非通常意义上的构造方法, 这是Java帮我们合成的一个方法, 面的指令会帮我们按顺序进行普通成员变量初始化, 也包括初始化块里的代码, 注意是按顺序执行, 这都执行完了之后才轮到构造方法里代码生成的指令执行。 这里aload_0将局部变量表中下标为0的元入栈, 其实就是Java中的this, 结合invokespecial #1, 是在调用父类的构造函数, 也就是我们常见的supr()。

所以我们再看SuperClass.<init>

```
public class bugme.SuperClass {
public bugme.SuperClass();

Code:
  0: aload_0
  1: invokespecial #1                  // Method java/lang/Object."<init>":()V
  4: aload_0
```

```
  5: bipush        99
  7: invokevirtual #2                  // Method setX:(I)V
 10: return
.....
}
```

同样是先调了父类Object的构造方法, 然后再将this, 99入栈, invokevirtual #2旁边注释了是调用setX, 参数分别是this和99也就是this.setX(99), 然而这个方法被重写了, 调用的是子类的方法, 所以我们再看ubClass.setX:

```
public class bugme.SubClass extends bugme.SuperClass {
.....
public void setX(int);

Code:
  0: aload_0
  1: iload_1
  2: invokespecial #3                  // Method bugme/SuperClass.setX:(I)V
  .....
}
```

这里将局部变量表前两个元素都入栈, 第一个是this, 第二个是括号里的参数, 也就是99, invokespecial #3调用的是父类的setX, 也就是我们代码中写的super.setX(int)

SuperClass.setX就很简单了:

```
public class bugme.SuperClass {
.....
public void setX(int);

Code:
  0: aload_0
  1: iload_1
  2: putfield      #3                  // Field mSuperX:I
  5: return
```

}

这里先把this入栈, 再把参数入栈, putfield #3使得前两个入栈的元素全部出栈, 而成员mSuperX被赋值
这四条指令只对应代码里的一句this.mSuperX = x;

接下来控制流回到子类的setX:

```
public class bugme.SubClass extends bugme.SuperClass {

.....

public void setX(int);


Code:

   0: aload_0

   1: iload_1

   2: invokespecial #3                  // Method bugme/SuperClass.setX:(I)V

 ->5: aload_0                           // 即将执行这句

   6: iload_1

   7: putfield      #2                  // Field mSubX:I

  10: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;

  13: new           #5                  // class java/lang/StringBuilder

  16: dup

  17: invokespecial #6                  // Method java/lang/StringBuilder."<init>":()V

  20: ldc           #7                  // String SubX is assigned

  22: invokevirtual #8                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
java/lang/StringBuilder;

  25: iload_1

  26: invokevirtual #9                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/Strin
Builder;

  29: invokevirtual #10                 // Method java/lang/StringBuilder.toString:()Ljava/lang/String


  32: invokevirtual #11                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

  35: return

}
```

从5处开始继续分析, 5,6,7将参数的值赋给mSubX, 此时mSubX是99了, 下面那一堆则是在执行System out.println("SubX is assigned " + x);并返回, 还可以看到Java自动帮我们使用StringBuilder优化字串拼接, 就不分析了.

说了这么多, 我们的代码才刚把下面箭头指着的这句执行完:

```
public class bugme.SubClass extends bugme.SuperClass {

public bugme.SubClass();


Code:

  0: aload_0

 ->1: invokespecial #1                  // Method bugme/SuperClass."<init>":()V

  4: aload_0

  5: iconst_1

  6: putfield      #2                  // Field mSubX:I

  9: return

.....

}
```

此时mSubX已经是99了, 再执行下面的4,5,6, 这一部分是SubClass的初始化, 代码将把1赋给mSubX, 9被1覆盖了。

方法返回后, 相当于我们执行完了箭头指的这一句代码:

```
public class Main {


public static void main(String[] args) {

  ->SubClass sc = new SubClass();

    sc.printX();

}

}
```

接下来执行的代码将打印mSubX的值, 自然就是1了.

以前就听说过JVM是基于栈的, Dalvik是基于寄存器的, 现在看了Java字节码, 回想一下smali, 自然就明白。 我在Android无需权限显示悬浮窗, 兼谈逆向分析app中有分析smali代码, smali里面经常看到似v0, v1这类东西, 是在操作寄存器, 而刚才分析的bytecode, 指令常常伴随着入栈出栈.

理论解释

Java是面向对象的语言, 面向对象三大特性之一多态性。假如父类构造方法中调用了某个方法, 这个方

恰好被子类重写了, 会发生什么?

根据多态性, 实际被调用的是子类的方法, 这个没错。 再考虑有继承时, 初始化的顺序， 如果是new一子类, 那么初始化顺序是:

父类static成员 -> 子类static成员 -> 父类普通成员初始化和初始化块 -> 父类构造方法 -> 子类普通员初始化和初始化块 -> 子类构造方法

父类构造方法中调用了一次setX, 此时mSubX中已经是我们要跟踪的值, 但之后子类普通成员初始化mSubX又初始化了一遍, 覆盖了前面我们跟踪的值, 自然得到的值就是错的。

Java中, 在构造方法中唯一能安全调用的是基类中的final方法, 自己的final方法(自己的private方法自动inal), 如果类本身是final的, 自然就能安全调用自己所有的方法。

完全遵守这个准则, 可以保证不会出这个bug. 实际上我们常常不能遵守, 所以要时刻小心这个问题.

题外话

关于默认初始化, 比如这样写:

```
public class SubClass extends SuperClass {

private int mSubX;

public SubClass() {}

.....

}
```

如果父类保证一定会在初始化时调用setX, 程序是不会出现上面说的bug的, 因为默认初始化并不是靠成下面这样的代码默认初始化.

```
4: aload_0

5: iconst_1

6: putfield      #2                  // Field mSubX:I
```

所谓的默认初始化, 其实是我们要实例化一个对象之前, 需要一块内存放我们的数据, 这块内存被全部为0, 这就是默认初始化了.

下面这两句话, 虽然效果一样, 但实际是有区别的.

```
private int mSubX;

private int mSubX = 0;
```

一般情况下, 这两句代码对程序没有任何影响(除非你遇到这个bug), 上面一句和下面一句的区别在于, 面一句会导致<init>方法里面生成3条指令, 分别是aload_0, iconst_0, putfield #**, 而上面一句则不。所以如果你的成员变量使用默认值初始化, 就没必要自己赋那个默认值, 而且还能省3条指令。

以上就是Java这类面向对象语言在重写和初始化过程中， 常常容易出现错误理解的地方，分享出来，望对后续Java新人的学习理解有所帮助。

相关文章：《搜索量最大的10个Java问题》 http://www.maiziedu.com/group/article/6937/