# 通过javap命令分析java汇编指令

## 一、javap命令简述

javap是jdk自带的反解析工具。它的作用就是根据class字节码文件，反解析出当前类对应的code区（汇编指令）、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。
当然这些信息中，有些信息（如本地变量表、指令和代码行偏移量映射表、常量池中方法的参数名称等等）需要在使用javac编译成class文件时，指定参数才能输出，比如，你直接javac xx.java，就不会在生成对应的局部变量表等信息，如果你使用javac -g xx.java就可以生成所有相关信息了。如果你使用的eclipse，则默认情况下，eclipse在编译时会帮你生成局部变量表、指令和代码行偏移量映射表等信息的。
通过反编译生成的汇编代码，我们可以深入的了解java代码的工作机制。比如我们可以查看i++；这行代码实际运行时是先获取变量i的值，然后将这个值加1，最后再将加1后的值赋值给变量i。
通过局部变量表，我们可以查看局部变量的作用域范围、所在槽位等信息，甚至可以看到槽位复用等信息。

javap的用法格式：

```
javap <options> <classes>
```

其中classes就是你要反编译的class文件。
在命令行中直接输入javap或javap -help可以看到javap的options有如下选项：

```
-help  --help  -?        输出此用法消息
-version                 版本信息，其实是当前javap所在jdk的版本信息，不是class在哪个jdk下生成的。
-v  -verbose             输出附加信息（包括行号、本地变量表，反汇编等详细信息）
-l                         输出行号和本地变量表
-public                    仅显示公共类和成员
-protected               显示受保护的/公共类和成员
-package                 显示程序包/受保护的/公共类 和成员 (默认)
-p  -private             显示所有类和成员
-c                       对代码进行反汇编
-s                       输出内部类型签名
-sysinfo                 显示正在处理的类的系统信息 (路径, 大小, 日期, MD5 散列)
-constants               显示静态最终常量
-classpath <path>        指定查找用户类文件的位置
-bootclasspath <path>    覆盖引导类文件的位置
```

一般常用的是-v -l -c三个选项。
javap -v classxx，不仅会输出行号、本地变量表信息、反编译汇编代码，还会输出当前类用到的常量池等信息。
javap -l 会输出行号和本地变量表信息。
javap -c 会对当前class字节码进行反编译生成汇编代码。
查看汇编代码时，需要知道里面的jvm指令，可以参考官方文档：
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html
另外通过jclasslib工具也可以看到上面这些信息，而且是可视化的，效果更好一些。

## 二、javap测试及内容详解

前面已经介绍过javap输出的内容有哪些，东西比较多，这里主要介绍其中code区(汇编指令)、局部变量表和代码行偏移映射三个部分。
如果需要分析更多的信息，可以使用javap -v进行查看。
另外，为了更方便理解，所有汇编指令不单拎出来讲解，而是在反汇编代码中以注释的方式讲解（吐槽一下，简书的markdown貌似不能改字体颜色，这一点很不爽）。

下面写段代码测试一下：
**例子1：**分析一下下面的代码反汇编之后结果：

```java
public class TestDate {

    private int count = 0;

    public static void main(String[] args) {
        TestDate testDate = new TestDate();
        testDate.test1();
    }

    public void test1(){
        Date date = new Date();
        String name1 = "wangerbei";
        test2(date,name1);
        System.out.println(date+name1);
    }

    public void test2(Date dateP,String name2){
        dateP = null;
        name2 = "zhangsan";
    }

    public void test3(){
        count++;
    }

    public void  test4(){
        int a = 0;
        {
            int b = 0;
            b = a+1;
        }
        int c = a+1;
    }
}
```

上面代码通过JAVAC -g 生成class文件，然后通过javap命令对字节码进行反汇编：

```
$ javap -c -l TestDate
```

得到下面内容(指令等部分是我参照着官方文档总结的)：

```
Warning: Binary file TestDate contains com.justest.test.TestDate
Compiled from "TestDate.java"
public class com.justest.test.TestDate {
  //默认的构造方法，在构造方法执行时主要完成一些初始化操作，包括一些成员变量的初始化赋值等操作
  public com.justest.test.TestDate();
    Code:
       0: aload_0 //从本地变量表中加载索引为0的变量的值，也即this的引用，压入栈
       1: invokespecial #10  //出栈，调用java/lang/Object."<init>":()V 初始化对象，就是this指
       4: aload_0  // 4到6表示，调用this.count = 0，也即为count复制为0。这里this引用入栈
       5: iconst_0 //将常量0，压入到操作数栈
       6: putfield      //出栈前面压入的两个值（this引用，常量值0），  将0取出，并赋值给count
       9: return
//指令与代码行数的偏移对应关系，每一行第一个数字对应代码行数，第二个数字对应前面code中指令前面的数字
    LineNumberTable:
      line 5: 0
      line 7: 4
      line 5: 9
    //局部变量表，start+length表示这个变量在字节码中的生命周期起始和结束的偏移位置（this生命周期从头0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   Lcom/justest/test/TestDate;

  public static void main(java.lang.String[]);
    Code:
// new指令，创建一个class com/justest/test/TestDate对象，new指令并不能完全创建一个对象，对象只有在
       0: new  //创建对象，并将对象引用压入栈
       3: dup //将操作数栈定的数据复制一份，并压入栈，此时栈中有两个引用值
       4: invokespecial #20  //pop出栈引用值,调用其构造函数，完成对象的初始化
       7: astore_1 //pop出栈引用值，将其（引用）赋值给局部变量表中的变量testDate
       8: aload_1  //将testDate的引用值压入栈，因为testDate.test1();调用了testDate，这里使用alo
       9: invokevirtual #21 // Method test1:()V  引用出栈，调用testDate的test1()方法
      12: return //整个main方法结束返回
    LineNumberTable:
      line 10: 0
      line 11: 8
      line 12: 12
    //局部变量表，testDate只有在创建完成并赋值后，才开始声明周期
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
```

```
        0      13     0  args   [Ljava/lang/String;
        8       5     1 testDate   Lcom/justest/test/TestDate;

  public void test1();
    Code:
       0: new           #27                 // 0到7创建Date对象，并赋值给date变量
       3: dup
       4: invokespecial #29                 // Method java/util/Date."<init>":()V
       7: astore_1
       8: ldc           #30     // String wangerbei，将常量“wangerbei”压入栈
      10: astore_2  //将栈中的“wangerbei”pop出，赋值给name1
      11: aload_0 //11到14，对应test2(date,name1);默认前面加this.
      12: aload_1 //从局部变量表中取出date变量
      13: aload_2 //取出name1变量
      14: invokevirtual #32                 // Method test2: (Ljava/util/Date;Ljava/lang/
  // 17到38对应System.out.println(date+name1);
      17: getstatic     #36                 // Field java/lang/System.out:Ljava/io/PrintS
  //20到35是jvm中的优化手段，多个字符串变量相加，不会两两创建一个字符串对象，而使用StringBuilder来创建
      20: new           #42                 // class java/lang/StringBuilder
      23: dup
      24: invokespecial #44                 // Method java/lang/StringBuilder."<init>":()
      27: aload_1
      28: invokevirtual #45                 // Method java/lang/StringBuilder.append:(Lja
      31: aload_2
      32: invokevirtual #49                 // Method java/lang/StringBuilder.append:(Lja
      35: invokevirtual #52                 // Method java/lang/StringBuilder.toString:()
      38: invokevirtual #56                 // Method java/io/PrintStream.println:(Ljava/
      41: return
    LineNumberTable:
      line 15: 0
      line 16: 8
      line 17: 11
      line 18: 17
      line 19: 41
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      42     0  this   Lcom/justest/test/TestDate;
          8      34     1  date   Ljava/util/Date;
         11      31     2 name1   Ljava/lang/String;

  public void test2(java.util.Date, java.lang.String);
    Code:
       0: aconst_null //将一个null值压入栈
       1: astore_1 //将null赋值给dateP
       2: ldc           #66       // String zhangsan 从常量池中取出字符串“zhangsan”压入栈中
       4: astore_2 //将字符串赋值给name2
       5: return
    LineNumberTable:
      line 22: 0
      line 23: 2
      line 24: 5
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lcom/justest/test/TestDate;
          0       6     1 dateP   Ljava/util/Date;
          0       6     2 name2   Ljava/lang/String;

  public void test3();
    Code:
       0: aload_0 //取出this，压入栈
       1: dup   //复制操作数栈栈顶的值，并压入栈，此时有两个this对象引用值在操作数组栈
       2: getfield #12// Field count:I this出栈，并获取其count字段，然后压入栈，此时栈中有一个th
       5: iconst_1 //取出一个int常量1,压入操作数栈
       6: iadd  // 从栈中取出count和1，将count值和1相加，结果入栈
       7: putfield      #12 // Field count:I  一次弹出两个，第一个弹出的是上一步计算值，第二个弹出
      10: return
    LineNumberTable:
      line 27: 0
      line 28: 10
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/justest/test/TestDate;
public void test4();
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_1
       5: iconst_1
       6: iadd
       7: istore_2
       8: iload_1
```

```
        9: iconst_1
       10: iadd
       11: istore_2
       12: return
     LineNumberTable:
       line 33: 0
       line 35: 2
       line 36: 4
       line 38: 8
       line 39: 12
     //看下面，b和c的槽位slot一样，这是因为b的作用域就在方法块中，方法块结束，局部变量表中的槽位就被释放，
     LocalVariableTable:
       Start  Length  Slot  Name   Signature
           0      13     0  this   Lcom/justest/test/TestDate;
           2      11     1     a   I
           4       4     2     b   I
          12       1     2     c   I
}
```

**例子2：**下面一个例子

先有一个User类：

```
public class User {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

然后写一个操作User对象的测试类：

```
public class TestUser {

    private int count;

    public void test(int a){
        count = count + a;
    }

    public User initUser(int age,String name){
        User user = new User();
        user.setAge(age);
        user.setName(name);
        return user;
    }

    public void changeUser(User user,String newName){
        user.setName(newName);
    }
}
```

先javac -g 编译成class文件。

然后对TestUser类进行反汇编：

```
$ javap -c -l TestUser
```

得到反汇编结果如下：

```
Warning: Binary file TestUser contains com.justest.test.TestUser
Compiled from "TestUser.java"
```

```
public class com.justest.test.TestUser {

//默认的构造函数
  public com.justest.test.TestUser();

    Code:
       0: aload_0
       1: invokespecial #10                 // Method java/lang/Object."<init>":()V
       4: return

    LineNumberTable:
      line 3: 0

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   Lcom/justest/test/TestUser;

  public void test(int);

    Code:
       0: aload_0 //取this对应的对应引用值，压入操作数栈
       1: dup //复制栈顶的数据，压入栈，此时栈中有两个值，都是this对象引用
       2: getfield      #18 // 引用出栈，通过引用获得对应count的值，并压入栈
       5: iload_1 //从局部变量表中取得a的值，压入栈中
       6: iadd //弹出栈中的count值和a的值，进行加操作，并将结果压入栈
       7: putfield      #18 // 经过上一步操作后，栈中有两个值，栈顶为上一步操作结果，栈顶下面是this
      10: return //return void

    LineNumberTable:
      line 8: 0
      line 9: 10

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/justest/test/TestUser;
          0      11     1     a   I

  public com.justest.test.User initUser(int, java.lang.String);

    Code:
       0: new           #23   // class com/justest/test/User 创建User对象，并将引用压入栈
       3: dup //复制栈顶值，再次压入栈，栈中有两个User对象的地址引用
       4: invokespecial #25   // Method com/justest/test/User."<init>":()V 调用user对象初始
       7: astore_3 //从栈中pop出User对象的引用值，并赋值给局部变量表中user变量
       8: aload_3 //从局部变量表中获得user的值，也就是User对象的地址引用，压入栈中
       9: iload_1 //从局部变量表中获得a的值，并压入栈中，注意aload和iload的区别，一个取值是对象引用，
      10: invokevirtual #26  // Method com/justest/test/User.setAge:(I)V 操作数栈pop出两个值
      13: aload_3 //同7，压入栈
      14: aload_2 //从局部变量表取出name，压入栈
      15: invokevirtual #29  // MethodUser.setName:(Ljava/lang/String;)V 操作数栈pop出两个值
      18: aload_3 //从局部变量取出User引用，压入栈
      19: areturn //areturn指令用于返回一个对象的引用，也就是上一步中User的引用,这个返回值将会被压入

    LineNumberTable:
      line 12: 0
      line 13: 8
      line 14: 13
      line 15: 18

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      20     0  this   Lcom/justest/test/TestUser;
          0      20     1   age   I
          0      20     2  name   Ljava/lang/String;
          8      12     3  user   Lcom/justest/test/User;

  public void changeUser(com.justest.test.User, java.lang.String);

    Code:
       0: aload_1 //局部变量表中取出user，也即User对象引用，压入栈
       1: aload_2 //局部变量表中取出newName，压入栈
       2: invokevirtual #29 // Method User.setName:(Ljava/lang/String;)V pop出栈newName值和
       5: return

    LineNumberTable:
      line 19: 0
      line 20: 5

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lcom/justest/test/TestUser;
          0       6     1  user   Lcom/justest/test/User;
```

```
            0       6     2 newName   Ljava/lang/String;

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

    Code:
       0: new      #1 // class com/justest/test/TestUser 创建TestUser对象，将引用压入栈
       3: dup //复制引用，压入栈
       4: invokespecial #43   // Method "<init>":()V 引用值出栈，调用构造方法，对象初始化
       7: astore_1 //引用值出栈，赋值给局部变量表中变量tu
       8: aload_1 //取出tu值，压入栈
       9: bipush    10 //将int值10压入栈
      11: ldc           #44   // String wangerbei 从常量池中取出“wangerbei” 压入栈
      13: invokevirtual #46    // Method initUser(ILjava/lang/String;)Lcom/justest/test/l
      16: astore_2 //User引用出栈，赋值给user变量
      17: aload_1 //取出tu值，压入栈
      18: aload_2 //取出user值，压入栈
      19: ldc           #48     // String lisi 从常量池中取出“lisi”压入栈
      21: invokevirtual #50     // Method changeUser:(Lcom/justest/test/User;Ljava/lang/S
      24: return //return void

 LineNumberTable:
      line 23: 0
      line 24: 8
      line 25: 17
      line 26: 24

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      25     0  args   [Ljava/lang/String;
          8      17     1    tu   Lcom/justest/test/TestUser;
         17       8     2  user   Lcom/justest/test/User;

}
```

## 三、总结

1、通过javap命令可以查看一个java类反汇编、常量池、变量表、指令代码行号表等等信息。

2、平常，我们比较关注的是java类中每个方法的反汇编中的指令操作过程，这些指令都是顺序执行的，可以参考官方文档查看每个指令的含义，很简单：

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.areturn

3、通过对前面两个例子代码反汇编中各个指令操作的分析，可以发现，一个方法的执行通常会涉及下面几块内存的操作：

（1）java栈中：局部变量表、操作数栈。这些操作基本上都值操作。
（2）java堆。通过对象的地址引用去操作。
（3）常量池。
（4）其他如帧数据区、方法区（jdk1.8之前，常量池也在方法区）等部分，测试中没有显示出来，这里说明一下。

在做值相关操作时：
一个指令，可以从局部变量表、常量池、堆中对象、方法调用、系统调用中等取得数据，这些数据（可能是指，可能是对象的引用）被压入操作数栈。
一个指令，也可以从操作数数栈中取出一到多个值（pop多次），完成赋值、加减乘除、方法传参、系统调用等等操作。