从 JDK7 开始，越来越多的语法糖加入到 Java 语言中，只有搞懂了这些内部实现原理，才能真正理解语法背后的限制和潜在的坑。这篇文章我们将讲解字符串的 switch-case 是如何实现的。

0x01 一个小 demo

前面我们已经知道了，switch-case 依据 case 值的稀疏程度，分别由两个指令 tableswitch 和 lookupswitch 实现，但这两个指令都只支持整型值。那怎么样让 String 类型的值也支持 switch-case 呢?

```
public int test(String name) {
    switch (name) {
        case "Java":
            return 100;
        case "Kotlin":
            return 200;
        default:
            return -1;
    }
}
```

我们直接来看字节码

```
 0: aload_1
 1: astore_2
 2: iconst_m1
 3: istore_3

 4: aload_2
 5: invokevirtual #2                  // Method
java/lang/String.hashCode:()I
```

```
 8: lookupswitch  { // 2
     -2041707231: 50 // 对应 "Kotlin".hashCode()
         2301506: 36 // 对应 "Java".hashCode()
         default: 61
    }

36: aload_2
37: ldc           #3                  // String
Java
39: invokevirtual #4                  // Method
java/lang/String.equals:(Ljava/lang/Object;)Z
42: ifeq          61
45: iconst_0
46: istore_3
47: goto          61

50: aload_2
51: ldc           #5                  // String
Kotlin
53: invokevirtual #4                  // Method
java/lang/String.equals:(Ljava/lang/Object;)Z
56: ifeq          61
59: iconst_1
60: istore_3

61: iload_3
62: lookupswitch  { // 2
               0: 88
               1: 91
         default: 95
    }

// 88 ~ 90
```

```
88: bipush        100
90: ireturn

91: sipush        200
94: ireturn

95: iconst_m1
96: ireturn
```

我们先把局部变量表放出来，可能会好理解一些

0	1	2	3
this	name	tmpName	matchIndex

- 0 ~ 3：做一些初始化操作，把入参 name 赋值给局部变量表下标为 2 的变量，记为tmpName，初始化局部变量表 3 位置的变量为 -1，记为 matchIndex
- 4 ~ 8：对 tmpName 调用了 hashCode 函数，得到一个整型值。因为一般而言 hash 都比较离散，所以没有选用 tableswitch 而是用 lookupswitch 来作为 switch case 的实现。
- 36 ~ 47：如果 hashCode 等于 "Java".hashCode() 会跳转到这部分字节码。首先把字符串进行真正意义上的 equals 比较，看是否相等，是否相等使用的是 ifeq 指令， ifeq 这个指令语义上有点绕，ifeq 的含义是ifeq 0则跳转到对应字节码行处，实际上是等于 false 跳转。这里如果相等则把 matchIndex 赋值为 0
- 61 ~ 96：进行最后的 case 分支执行。这一段比较好理解，不再继续做分析。

结合上面的字节码解读，我们可以推演出对应的 Java 代码实现

```
public int test_translate(String name) {
    String tmpName = name;
    int matchIndex = -1;
    switch (tmpName.hashCode()) {
        case -2041707231:
            if (tmpName.equals("Kotlin")) {
                matchIndex = 1;
            }
            break;
        case 2301506:
            if (tmpName.equals("Java")) {
                matchIndex = 0;
            }
            break;
        default:
            break;
    }
    switch (matchIndex) {
        case 0:
            return 100;
        case 1:
            return 200;
        default:
            return -1;
    }
}
```

0x02 hashCode 冲突如何处理

有人可能会想，hashCode 冲突的时候要怎么样处理，比如 "Aa" 和 "BB" 的 hashCode 都是 2112。

```
public int testSameHash(java.lang.String);
descriptor: (Ljava/lang/String;)I
flags: ACC_PUBLIC
Code:
  stack=2, locals=4, args_size=2
     0: aload_1
     1: astore_2
     2: iconst_m1
     3: istore_3

     4: aload_2
     5: invokevirtual #2                  //
Method java/lang/String.hashCode:()I
     8: lookupswitch  { // 1
                  2112: 28
               default: 53
          }

    28: aload_2
    29: ldc           #3                  //
String BB
    31: invokevirtual #4                  //
Method java/lang/String.equals:
(Ljava/lang/Object;)Z
    34: ifeq          42
    37: iconst_1
    38: istore_3
    39: goto          53

    42: aload_2
    43: ldc           #5                  //
String Aa
    45: invokevirtual #4                  //
```

```
Method java/lang/String.equals:
(Ljava/lang/Object;)Z
    48: ifeq          53
    51: iconst_0
    52: istore_3

    53: iload_3
    54: lookupswitch  { // 2
                  0: 80
                  1: 83
            default: 87
        }
    80: bipush        100
    82: ireturn
    83: sipush        200
    86: ireturn
    87: iconst_m1
    88: ireturn
```

可以看到 34 行 在 hashCode 冲突的情况下，JVM 的处理不过是多一次字符串相等的比较。与 "BB" 不相等的情况，会继续判断是否等于 "Aa"，翻译为 Java 源代码如下：

```
public int testSameHash_translate(String name) {
    String tmpName = name;
    int matchIndex = -1;

    switch (tmpName.hashCode()) {
        case 2112:
            if (tmpName.equals("BB")) {
                matchIndex = 1;
            } else if (tmpName.equals("Aa")) {
                matchIndex = 0;
            }
            break;
        default:
            break;
    }

    switch (matchIndex) {
        case 0:
            return 100;
        case 1:
            return 200;
        default:
            return -1;
    }
}
```

0x03 小结

总结一下，JDK7 引入的 String 的 switch 实现流程分为下面几步：

1. 计算字符串 hashCode
2. 使用 lookupswitch 对整型 hashCode 进行分支

3. 对相同 hashCode 值的字符串进行最后的字符串匹配
4. 执行 case 块代码

0x04 思考

最后，给你留两道思考题

1. Java 的 hashCode 冲突的概率其实是很大的，其底层原因是什么?
2. 你可以随意构造两个 hashCode 相同的字符串吗？它们有什么规律

欢迎你在留言区留言，和我一起讨论。