Java 基础（6）：Switch 语句的深入解析

switch 语句是非常的基础的知识，掌握起来也不难掌握，语法比较简单。但大部分人基本是知其然，不知其所以然。譬如 早期 JDK 只允许 switch 的表达式的值 int 及 int 类型以下的基本类型，后期的 JDK 却允许匹配比较字符串、枚举类型，这是怎么做到的呢？原理是什么？本文将深入去探索。

switch 语法格式

```
switch (表达式) {
    case 常量表达式或枚举常量:
        语句;
        break;
    case 常量表达式或枚举常量:
        语句;
        break;
    ......
    default: 语句;
        break;
}
```

switch 匹配的表达式可以是：

- byte、short、char、int 类型及这 4 种类型的包装类型；
- 枚举类型；
- String 类型；

case 匹配的表达式可以是：

- 常量表达式；
- 枚举常量；

注意一点： case 提供了 switch 表达式的入口地址，一旦 switch 表达式与某个 case 分支匹配 ,则从该分支的语句开始执行 ,一直执行下去 ,即其后的所有 case 分支的语句也会被执行，直到遇到 break 语句。

```
public static void main(String[] args) {
    String  s = "a";
    switch (s) {
        case "a": //a 分支
             System.out.println("匹配成功 1");
        case "b": //b 分支
               System.out.println("匹配成功 2");
        case "c": //c 分支
                System.out.println("匹配成功 3");
```

```
                break;
        case "d": //d 分支
                System.out.println("匹配成功 4");
                break;
        default:
            break;
    }
}
```

运行结果：

```
匹配成功 1
匹配成功 2
匹配成功 3
```

switch 成功匹配了 a 分支，但 a、b 分支都没有 break 语句，所以一直执行 a 分支后的所有语句，直到遇到 c 分支的 break 语句才终止。

尽管 switch 支持的类型扩充了几个，但**其实在底层中，swtich 只能支持 4 种基本类型，其他几个类型是通过一些方式来间接处理的**，下面便是讲解编译器对扩充类型的处理。

1.对包装类的处理

对包装类的处理是最简单的 —— 拆箱。看下面的例子，switch 比较的是包装类 Byte 。

```
Byte b = 2;
    switch (b) {
    case 1:
        System.out.println("匹配成功");
        break;
    case 2:
        System.out.println("匹配成功");
        break;
    }
```

用 jad 反编译一下这段代码，得到的代码如下：

```
Byte b = Byte.valueOf((byte)2);
switch(b.byteValue()){
    case 1: // '\001'
        System.out.println("\u5339\u914D\u6210\u529F");
        break;
    case 2: // '\002'
        System.out.println("\u5339\u914D\u6210\u529F");
        break;
```

```
}
```

反编译的代码很简单，底层的 switch 比较的是 Byte 通过（拆箱）方法 byteValue()得到的 byte 值。顺便说一下，这段反编译代码不仅揭开了拆箱的解析原理，也展示了装箱的解析原理（第一句代码）；

2.枚举类型

为了简单起见，直接采用 JDK 提供的枚举类型的线程状态类 Thread.state 类。

```
Thread.State state = Thread.State.RUNNABLE;
    switch (state) {
    case NEW:
        System.out.println("线程处于创建状态");
        break;
    case RUNNABLE:
        System.out.println("线程处于可运行状态");
        break;
    case TERMINATED:
        System.out.println("线程结束");
        break;
    default:
        break;
}
```

反编译代码：

```
Sex sex = Sex.MALE;
    switch($SWITCH_TABLE$Test_2018_1_14$Sex()[sex.ordinal()]){
        case 1: // '\001'
            System.out.println("sex:male");
            break;
        case 2: // '\002'
            System.out.println("sex:female");
            break;
    }
```

从编译代码中发现，编译器对于枚举类型的处理，是通过创建一个辅助数组来处理，这个数组是通过一个$SWITCH_TABLE$java$lang$Thread$State() 方法创建的，数组是一个 int[]类型数组，数组很简单，在每个枚举常量的序号所对应的数组下标位置的赋一个值，按序号大小赋值，从 1 开始递增。 其代码如下：

```
//int 数组
private static int $SWITCH_TABLE$java$lang$Thread$State[];
```

```
//创建数组的方法
static int[] $SWITCH_TABLE$java$lang$Thread$State()
    {
        $SWITCH_TABLE$java$lang$Thread$State;
        if($SWITCH_TABLE$java$lang$Thread$State == null) goto _L2; else goto _L1
_L1:
        return;
_L2:
        JVM INSTR pop ;
        int ai[] = new int[Thread.State.values().length];
        try
        {
            ai[Thread.State.BLOCKED.ordinal()] = 3;
        }
        catch(NoSuchFieldError _ex) { }
        try
        {
            ai[Thread.State.NEW.ordinal()] = 1;
        }
        catch(NoSuchFieldError _ex) { }
        try
        {
            ai[Thread.State.RUNNABLE.ordinal()] = 2;
        }
        catch(NoSuchFieldError _ex) { }
        try
        {
            ai[Thread.State.TERMINATED.ordinal()] = 6;
        }
        catch(NoSuchFieldError _ex) { }
        try
        {
            ai[Thread.State.TIMED_WAITING.ordinal()] = 5;
        }
        catch(NoSuchFieldError _ex) { }
        try
        {
            ai[Thread.State.WAITING.ordinal()] = 4;
        }
        catch(NoSuchFieldError _ex) { }
```

```
        return $SWITCH_TABLE$java$lang$Thread$State = ai;
    }
}
```

3.对 String 类型的处理

依旧是先看个例子，再查看这个例子反编译代码，了解编译器的是如何解析的。

```
public static void main(String[] args) {
    String  s = "China";
    switch (s) {
        case "America":
                System.out.println("匹配到美国");
                break;
        case "China":
               System.out.println("匹配到中国");
               break;
        case "Japan":
                System.out.println("匹配到日本");
        default:
            break;
    }
}
```

反编译得到的代码：

```
public static void main(String args[]){
    String s = "China";
    String s1;
    switch((s1 = s).hashCode()){
        default:
            break;
        case 65078583:
            if(s1.equals("China"))
                System.out.println("\u5339\u914D\u5230\u4E2D\u56FD");
            break;
        case 71341030:
            if(s1.equals("Japan"))
                System.out.println("\u5339\u914D\u5230\u65E5\u672C");
            break;
        case 775550446:
```

```
            if(s1.equals("America"))
                System.out.println("\u5339\u914D\u5230\u7F8E\u56FD");
            break;
    }
}
```

从反编译的代码可以看出，switch 的 String 变量、case 的 String 常量都变成对应的字符串的 hash 值。也就是说，switch 仍然没有超出它的限制，只是通过使用 String 对象的 hash 值来进行匹配比较，从而支持 String 类型。

- 底层的 switch 只能处理 4 个基本类型的值。其他三种类型需要通过其他方式间接处理，即转成基本类型来处理。
- 编译器对包装类的处理是通过拆箱。
- 对枚举类型的处理，是通过枚举常量的序号及一个数组。
- 对字符串 String 的处理，是通过 String 的 hash 值。

1) 下列哪一种叙述是正确的（D ）

A． abstract 修饰符可修饰字段、方法和类

B． 抽象方法的 body 部分必须用一对大括号{ }包住

C． 声明抽象方法，大括号可有可无

D． 声明抽象方法不可写出大括号

2) JSP 和 Servlet 有哪些相同点和不同点，他们之间的联系是什么？

JSP 是 Servlet 技术的扩展，本质上是 Servlet 的简易方式，更强调应用的外表表达。JSP 编译后是"类 servlet"。Servlet 和 JSP 最主要的不同点在于，Servlet 的应用逻辑是在 Java 文件中，并且完全从表示层中的 HTML 里分离开来。而 JSP 的情况是 Java 和 HTML 可以组合成一个扩展名为.jsp 的文件。JSP 侧重于视图，Servlet 主要用于控制逻辑。