

二

## 15 Java语法糖与Java编译器

在前面的篇章中，我们多次提到了 Java 语法和 Java 字节码的差异之处。这些差异之处都是通过 Java 编译器来协调的。今天我们便来列举一下 Java 编译器的协调工作。

### 自动装箱与自动拆箱

首先要提到的便是 Java 的自动装箱（auto-boxing）和自动拆箱（auto-unboxing）。

我们知道，Java 语言拥有 8 个基本类型，每个基本类型都有对应的包装（wrapper）类型。

之所以需要包装类型，是因为许多 Java 核心类库的 API 都是面向对象的。举个例子，Java 核心类库中的容器类，就只支持引用类型。

当需要一个能够存储数值的容器类时，我们往往定义一个存储包装类对象的容器。

对于基本类型的数值来说，我们需要先将其转换为对应的包装类，再存入容器之中。在 Java 程序中，这个转换可以是显式，也可以是隐式的，后者正是 Java 中的自动装箱。

```
public int foo() {  
    ArrayList<Integer> list = new ArrayList<>();  
    list.add(0);  
    int result = list.get(0);  
    return result;  
}
```

以上图中的 Java 代码为例。我构造了一个 Integer 类型的 ArrayList，并且向其中添加一个 int 值 0。然后，我会获取该 ArrayList 的第 0 个元素，并作为 int 值返回给调用者。这段代码对应的 Java 字节码如下所示：

```
public int foo();  
Code:  
  0: new java/util/ArrayList  
  3: dup  
  4: invokespecial java/util/ArrayList."<init>":()V
```

```
7: astore_1
8: aload_1
9: iconst_0
10: invokestatic java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
13: invokevirtual java/util/ArrayList.add:(Ljava/lang/Object;)Z
16: pop
17: aload_1
18: iconst_0
19: invokevirtual java/util/ArrayList.get:(I)Ljava/lang/Object;
22: checkcast java/lang/Integer
25: invokevirtual java/lang/Integer.intValue:()I
28: istore_2
29: iload_2
30: ireturn
```

当向泛型参数为 `Integer` 的 `ArrayList` 添加 `int` 值时，便需要用到自动装箱了。在上面字节码偏移量为 10 的指令中，我们调用了 `Integer.valueOf` 方法，将 `int` 类型的值转换为 `Integer` 类型，再存储至容器类中。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

这是 `Integer.valueOf` 的源代码。可以看到，当请求的 `int` 值在某个范围内时，我们会返回缓存了的 `Integer` 对象；而当所请求的 `int` 值在范围之外时，我们则会新建一个 `Integer` 对象。

在介绍反射的那一篇中，我曾经提到参数 `java.lang.Integer.IntegerCache.high`。这个参数将影响这里面的 `IntegerCache.high`。

也就是说，我们可以通过配置该参数，扩大 `Integer` 缓存的范围。Java 虚拟机参数 `-XX:+AggressiveOpts` 也会将 `IntegerCache.high` 调整至 20000。

奇怪的是，Java 并不支持对 `IntegerCache.low` 的更改，也就是说，对于小于 -128 的整数，我们无法直接使用由 Java 核心类库所缓存的 `Integer` 对象。

```
25: invokevirtual java/lang/Integer.intValue:()I
```

当从泛型参数为 `Integer` 的 `ArrayList` 取出元素时，我们得到的实际上也是 `Integer` 对象。如果应用程序期待的是一个 `int` 值，那么就会发生自动拆箱。

在我们的例子中，自动拆箱对应的是字节码偏移量为 25 的指令。该指令将调用 `Integer.intValue` 方法。这是一个实例方法，直接返回 `Integer` 对象所存储的 `int` 值。

## 泛型与类型擦除

你可能已经留意到了，在前面例子生成的字节码中，往 ArrayList 中添加元素的 add 方法，所接受的参数类型是 Object；而从 ArrayList 中获取元素的 get 方法，其返回类型同样也是 Object。

前者还好，但是对于后者，在字节码中我们需要进行向下转换，将所返回的 Object 强制转换为 Integer，方能进行接下来的自动拆箱。

```
13: invokevirtual java/util/ArrayList.add:(Ljava/lang/Object;)Z
...
19: invokevirtual java/util/ArrayList.get:(I)Ljava/lang/Object;
22: checkcast java/lang/Integer
```

之所以会出现这种情况，是因为 Java 泛型的类型擦除。这是个什么概念呢？简单地说，那便是 Java 程序里的泛型信息，在 Java 虚拟机里全部都丢失了。这么做主要是为了兼容引入泛型之前的代码。

当然，并不是每一个泛型参数被擦除类型后都会变成 Object 类。对于限定了继承类的泛型参数，经过类型擦除后，所有的泛型参数都将变成所限定的继承类。也就是说，Java 编译器将选取该泛型所能指代的所有类中层次最高的那个，作为替换泛型的类。

```
class GenericTest<T extends Number> {
    T foo(T t) {
        return t;
    }
}
```

举个例子，在上面这段 Java 代码中，我定义了一个 T extends Number 的泛型参数。它对应的字节码如下所示。可以看到，foo 方法的方法描述符所接收参数的类型以及返回类型都为 Number。方法描述符是 Java 虚拟机识别方法调用的目标方法的关键。

```
T foo(T);
descriptor: (Ljava/lang/Number;)Ljava/lang/Number;
flags: (0x0000)
Code:
    stack=1, locals=2, args_size=2
        0: aload_1
        1: areturn
Signature: (TT;)TT;
```

不过，字节码中仍存在泛型参数的信息，如方法声明里的 T foo(T)，以及方法签名 (Signature) 中的“(TT;)TT;”。这类信息主要由 Java 编译器在编译他类时使用。

既然泛型会被类型擦除，那么我们还有必要用它吗？

我认为是有必要的。Java 编译器可以根据泛型参数判断程序中的语法是否正确。举例来说，尽管经过类型擦除后，`ArrayList.add` 方法所接收的参数是 `Object` 类型，但是往泛型参数为 `Integer` 类型的 `ArrayList` 中添加字符串对象，Java 编译器是会报错的。

```
ArrayList<Integer> list = new ArrayList<>();
list.add("0"); // 编译出错
```

## 桥接方法

泛型的类型擦除带来了不少问题。其中一个便是方法重写。在第四篇的课后实践中，我留了这么一段代码：

```
class Merchant<T extends Customer> {
    public double actionPrice(T customer) {
        return 0.0d;
    }
}

class VIPOnlyMerchant extends Merchant<VIP> {
    @Override
    public double actionPrice(VIP customer) {
        return 0.0d;
    }
}
```

`VIPOnlyMerchant` 中的 `actionPrice` 方法是符合 Java 语言的方法重写的，毕竟都使用 `@Override` 来注解了。然而，经过类型擦除后，父类的方法描述符为 `(LjavaCustomer;)D`，而子类的方法描述符为 `(LVIP;)D`。这显然不符合 Java 虚拟机关于方法重写的定义。

为了保证编译而成的 Java 字节码能够保留重写的语义，Java 编译器额外添加了一个桥接方法。该桥接方法在字节码层面重写了父类的方法，并将调用子类的方法。

```
class VIPOnlyMerchant extends Merchant<VIP>
...
    public double actionPrice(VIP);
        descriptor: (LVIP;)D
        flags: (0x0001) ACC_PUBLIC
        Code:
            0: dconst_0
            1: dreturn

    public double actionPrice(Customer);
        descriptor: (LjavaCustomer;)D
```

```

flags: (0x1041) ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
Code:
  0: aload_0
  1: aload_1
  2: checkcast class VIP
  5: invokevirtual actionPrice:(LVIP;)D
  8: dreturn

```

```

// 这个桥接方法等同于
public double actionPrice(Customer customer) {
    return actionPrice((VIP) customer);
}

```

在我们的例子中，VIPOnlyMerchant 类将包含一个桥接方法 actionPrice(Customer)，它重写了父类的同名同方法描述符的方法。该桥接方法将传入的 Customer 参数强制转换为 VIP 类型，再调用原本的 actionPrice(VIP) 方法。

当一个声明类型为 Merchant，实际类型为 VIPOnlyMerchant 的对象，调用 actionPrice 方法时，字节码里的符号引用指向的是 Merchant.actionPrice(Customer) 方法。Java 虚拟机将动态绑定至 VIPOnlyMerchant 类的桥接方法之中，并且调用其 actionPrice(VIP) 方法。

需要注意的是，在 javap 的输出中，该桥接方法的访问标识符除了代表桥接方法的 ACC\_BRIDGE 之外，还有 ACC\_SYNTHETIC。它表示该方法对于 Java 源代码来说是不可见的。当你尝试通过传入一个声明类型为 Customer 的对象作为参数，调用 VIPOnlyMerchant 类的 actionPrice 方法时，Java 编译器会报错，并且提示参数类型不匹配。

```

Customer customer = new VIP();
new VIPOnlyMerchant().actionPrice(customer); // 编译出错

```

当然，如果你实在想要调用这个桥接方法，那么你可以选择使用反射机制。

```

class Merchant {
    public Number actionPrice(Customer customer) {
        return 0;
    }
}

class NaiveMerchant extends Merchant {
    @Override
    public Double actionPrice(Customer customer) {
        return 0.0D;
    }
}

```

除了前面介绍的泛型重写会生成桥接方法之外，如果子类定义了一个与父类参数类型相同的

方法，其返回类型为父类方法返回类型的子类，那么 Java 编译器也会为其生成桥接方法。

```
class NaiveMerchant extends Merchant
  public java.lang.Double actionPrice(Customer);
    descriptor: (Ljava/lang/Double;)Ljava/lang/Double;
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
      0: dconst_0
      1: invokestatic Double.valueOf:(D)Ljava/lang/Double;
      4: areturn

  public java.lang.Number actionPrice(Customer);
    descriptor: (Ljava/lang/Number;)Ljava/lang/Number;
    flags: (0x1041) ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=2, locals=2, args_size=2
      0: aload_0
      1: aload_1
      2: invokevirtual actionPrice:(Ljava/lang/Double;)Ljava/lang/Double;
      5: areturn
```

我之前曾提到过，class 文件里允许出现两个同名、同参数类型但是不同返回类型的方法。这里的原方法和桥接方法便是其中一个例子。由于该桥接方法同样标注了 ACC\_SYNTHETIC，因此，当在 Java 程序中调用 NaiveMerchant.actionPrice 时，我们只会调用到原方法。

## 其他语法糖

在前面的篇章中，我已经介绍过了变长参数、try-with-resources 以及在同一 catch 代码块中捕获多种异常等语法糖。下面我将列举另外两个常见的语法糖。

foreach 循环允许 Java 程序在 for 循环里遍历数组或者 Iterable 对象。对于数组来说，foreach 循环将从 0 开始逐一访问数组中的元素，直至数组的末尾。其等价的代码如下面所示：

```
public void foo(int[] array) {
    for (int item : array) {
    }
}
// 等同于
public void bar(int[] array) {
    int[] myArray = array;
    int length = myArray.length;
    for (int i = 0; i < length; i++) {
        int item = myArray[i];
    }
}
```

```
}
```

对于 Iterable 对象来说，foreach 循环将调用其 iterator 方法，并且用它的 hasNext 以及 next 方法来遍历该 Iterable 对象中的元素。其等价的代码如下面所示：

```
public void foo(ArrayList<Integer> list) {  
    for (Integer item : list) {  
    }  
}  
// 等同于  
public void bar(ArrayList<Integer> list) {  
    Iterator<Integer> iterator = list.iterator();  
    while (iterator.hasNext()) {  
        Integer item = iterator.next();  
    }  
}
```

字符串 switch 编译而成的字节码看起来非常复杂，但实际上就是一个哈希桶。由于每个 case 所截获的字符串都是常量值，因此，Java 编译器会将原来的字符串 switch 转换为 int 值 switch，比较所输入的字符串的哈希值。

由于字符串哈希值很容易发生碰撞，因此，我们还需要用 String.equals 逐个比较相同哈希值的字符串。

如果你感兴趣的话，可以自己利用 javap 分析字符串 switch 编译而成的字节码。

## 总结与实践

---

今天我主要介绍了 Java 编译器对几个语法糖的处理。

基本类型和其包装类型之间的自动转换，也就是自动装箱、自动拆箱，是通过加入 [Wrapper].valueOf（如 Integer.valueOf）以及 [Wrapper].[primitive]Value（如 Integer.intValue）方法调用来实现的。

Java 程序中的泛型信息会被擦除。具体来说，Java 编译器将选取该泛型所能指代的所有类中层次最高的那个，作为替换泛型的具体类。

由于 Java 语义与 Java 字节码中关于重写的定义并不一致，因此 Java 编译器会生成桥接方法作为适配器。此外，我还介绍了 foreach 循环以及字符串 switch 的编译。

今天的实践环节，你可以探索一下 Java 10 的 var 关键字，是否保存了泛型信息？是否支持自动装拆箱？

```
public void foo() {  
    var value = 1;  
    var list = new ArrayList<Integer>();  
    list.add(value);  
    // list.add("1"); 这一句能够编译吗?  
}
```

[上一页](#)[下一页](#)