04 | JVM是如何执行方法调用的？（上）

2018-07-27 郑雨迪

04 | JVM是如何执行方法调用的？（上）
朗读人：郑雨迪　13′37″ | 6.25M

前不久在写代码的时候，我不小心踩到一个可变长参数的坑。你或许已经猜到了，它正是可变长参数方法的重载造成的。（注：官方文档建议避免重载可变长参数方法，见 [1] 的最后一段。）

我把踩坑的过程放在了文稿里，你可以点击查看。

```
void invoke(Object obj, Object... args) { ... }
void invoke(String s, Object obj, Object... args) { ... }

invoke(null, 1);    // 调用第二个 invoke 方法
invoke(null, 1, 2); // 调用第二个 invoke 方法
invoke(null, new Object[]{1}); // 只有手动绕开可变长参数的语法糖，
                               // 才能调用第一个 invoke 方法
```

当时情况是这样子的，某个 API 定义了两个同名的重载方法。其中，第一个接收一个 Object，以及声明为 Object...的变长参数；而第二个则接收一个 String、一个 Object，以及声明为 Object...的变长参数。

这里我想调用第一个方法，传入的参数为 (null, 1)。也就是说，声明为 Object 的形式参数所对应的实际参数为 null，而变长参数则对应 1。

通常来说，之所以不提倡可变长参数方法的重载，是因为 Java 编译器可能无法决定应该调用哪个目标方法。

在这种情况下，编译器会报错，并且提示这个方法调用有二义性。然而，Java 编译器直接将我的方法调用识别为调用第二个方法，这究竟是为什么呢？

带着这个问题，我们来看一看 Java 虚拟机是怎么识别目标方法的。

重载与重写

在 Java 程序里，如果同一个类中出现多个名字相同，并且参数类型相同的方法，那么它无法通过编译。也就是说，在正常情况下，如果我们想要在同一个类中定义名字相同的方法，那么它们的参数类型必须不同。这些方法之间的关系，我们称之为重载。

```
小知识：这个限制可以通过字节码工具绕开。也就是说，在编译完成之后，我们可以再向 class 文件中添加方法名和
```

重载的方法在编译过程中即可完成识别。具体到每一个方法调用，Java 编译器会根据所传入参数的声明类型（注意与实际类型区分）来选取重载方法。选取的过程共分为三个阶段：

1. 在不考虑对基本类型自动装拆箱（auto-boxing，auto-unboxing），以及可变长参数的情况下选取重载方法；
2. 如果在第 1 个阶段中没有找到适配的方法，那么在允许自动装拆箱，但不允许可变长参数的情况下选取重载方法；
3. 如果在第 2 个阶段中没有找到适配的方法，那么在允许自动装拆箱以及可变长参数的情况下选取重载方法。

如果 Java 编译器在同一个阶段中找到了多个适配的方法，那么它会在其中选择一个最为贴切的，而决定贴切程度的一个关键就是形式参数类型的继承关系。

在开头的例子中，当传入 null 时，它既可以匹配第一个方法中声明为 Object 的形式参数，也可以匹配第二个方法中声明为 String 的形式参数。由于 String 是 Object 的子类，因此 Java 编译器会认为第二个方法更为贴切。

除了同一个类中的方法，重载也可以作用于这个类所继承而来的方法。也就是说，如果子类定义了与父类中非私有方法同名的方法，而且这两个方法的参数类型不同，那么在子类中，这两个方法同样构成了重载。

那么，如果子类定义了与父类中非私有方法同名的方法，而且这两个方法的参数类型相同，那么这两个方法之间又是什么关系呢？

如果这两个方法都是静态的，那么子类中的方法隐藏了父类中的方法。如果这两个方法都不是静态的，且都不是私有的，那么子类的方法重写了父类中的方法。

众所周知，Java 是一门面向对象的编程语言，它的一个重要特性便是多态。而方法重写，正是多态最重要的一种体现方式：它允许子类在继承父类部分功能的同时，拥有自己独特的行为。

打个比方，如果你经常漫游，那么你可能知道，拨打 10086 会根据你当前所在地，连接到当地的客服。重写调用也是如此：它会根据调用者的动态类型，来选取实际的目标方法。

JVM 的静态绑定和动态绑定

接下来，我们来看看 Java 虚拟机是怎么识别方法的。

Java 虚拟机识别方法的关键在于类名、方法名以及方法描述符（method descriptor）。前面两个就不做过多的解释了。至于方法描述符，它是由方法的参数类型以及返回类型所构成。在同一个类中，如果同时出现多个名字相同且描述符也相同的方法，那么 Java 虚拟机会在类的验证阶段报错。

可以看到，Java 虚拟机与 Java 语言不同，它并不限制名字与参数类型相同，但返回类型不同的方法出现在同一个类中，对于调用这些方法的字节码来说，由于字节码所附带的方法描述符包含了返回类型，因此 Java 虚拟机能够准确地识别目标方法。

Java 虚拟机中关于方法重写的判定同样基于方法描述符。也就是说，如果子类定义了与父类中非私有、非静态方法同名的方法，那么只有当这两个方法的参数类型以及返回类型一致，Java 虚拟机才会判定为重写。

对于 Java 语言中重写而 Java 虚拟机中非重写的情况，编译器会通过生成桥接方法 [2] 来实现 Java 中的重写语义。

由于对重载方法的区分在编译阶段已经完成，我们可以认为 Java 虚拟机不存在重载这一概念。因此，在某些文章中，重载也被称为静态绑定（static binding），或者编译时多态（compile-time polymorphism）；而重写则被称为动态绑定（dynamic binding）。

这个说法在 Java 虚拟机语境下并非完全正确。这是因为某个类中的重载方法可能被它的子类所重写，因此 Java 编译器会将所有对非私有实例方法的调用编译为需要动态绑定的类型。

确切地说，Java 虚拟机中的静态绑定指的是在解析时便能够直接识别目标方法的情况，而动态绑定则指的是需要在运行过程中根据调用者的动态类型来识别目标方法的情况。

具体来说，Java 字节码中与调用相关的指令共有五种。

1. invokestatic：用于调用静态方法。
2. invokespecial：用于调用私有实例方法、构造器，以及使用 super 关键字调用父类的实例方法或构造器，和所实现接口的默认方法。
3. invokevirtual：用于调用非私有实例方法。
4. invokeinterface：用于调用接口方法。
5. invokedynamic：用于调用动态方法。

由于 invokedynamic 指令较为复杂，我将在后面的篇章中单独介绍。这里我们只讨论前四种。

我在文章中贴了一段代码，展示了编译生成这四种调用指令的情况。

```
interface 客户 {
  boolean isVIP();
}

class 商户 {
  public double 折后价格 (double 原价, 客户 某客户) {
    return 原价 * 0.8d;
  }
}

class 奸商 extends 商户 {
  @Override
  public double 折后价格 (double 原价, 客户 某客户) {
    if (某客户.isVIP()) {                          // invokeinterface
      return 原价 * 价格歧视 ();                     // invokestatic
    } else {
      return super. 折后价格 (原价, 某客户);          // invokespecial
    }
  }
  public static double 价格歧视 () {
    // 咱们的杀熟算法太粗暴了，应该将客户城市作为随机数生成器的种子。
    return new Random()                            // invokespecial
           .nextDouble()                           // invokevirtual
```

```
            + 0.8d;
    }
}
```

在代码中，“商户”类定义了一个成员方法，叫做“折后价格”，它将接收一个 double 类型的参数，以及一个“客户”类型的参数。这里“客户”是一个接口，它定义了一个接口方法，叫“isVIP”。

我们还定义了另一个叫做“奸商”的类，它继承了“商户”类，并且重写了“折后价格”这个方法。如果客户是 VIP，那么它会被给到一个更低的折扣。

在这个方法中，我们首先会调用“客户”接口的”isVIP“方法。该调用会被编译为 invokeinterface 指令。

如果客户是 VIP，那么我们会调用奸商类的一个名叫“价格歧视”的静态方法。该调用会被编译为 invokestatic 指令。如果客户不是 VIP，那么我们会通过 super 关键字调用父类的“折后价格”方法。该调用会被编译为 invokespecial 指令。

在静态方法“价格歧视”中，我们会调用 Random 类的构造器。该调用会被编译为 invokespecial 指令。然后我们会以这个新建的 Random 对象为调用者，调用 Random 类中的 nextDouble 方法。该调用会被编译为 invokevirutal 指令。

对于 invokestatic 以及 invokespecial 而言，Java 虚拟机能够直接识别具体的目标方法。

而对于 invokevirtual 以及 invokeinterface 而言，在绝大部分情况下，虚拟机需要在执行过程中，根据调用者的动态类型，来确定具体的目标方法。

唯一的例外在于，如果虚拟机能够确定目标方法有且仅有一个，比如说目标方法被标记为 final[3][4]，那么它可以不通过动态类型，直接确定目标方法。

调用指令的符号引用

在编译过程中，我们并不知道目标方法的具体内存地址。因此，Java 编译器会暂时用符号引用来表示该目标方法。这一符号引用包括目标方法所在的类或接口的名字，以及目标方法的方法名和方法描述符。

符号引用存储在 class 文件的常量池之中。根据目标方法是否为接口方法，这些引用可分为接口符号引用和非接口符号引用。我在文章中贴了一个例子，利用“javap -v”打印某个类的常量池，如果你感兴趣的话可以到文章中查看。

```
// 在奸商.class 的常量池中，#16 为接口符号引用，指向接口方法 " 客户.isVIP()"。而 #22 为非接口符号引
$ javap -v 奸商.class ...
Constant pool:
...
  #16 = InterfaceMethodref #27.#29        // 客户.isVIP:()Z
...
  #22 = Methodref          #1.#33         // 奸商. 价格歧视:()D
...
```

上一篇中我曾提到过，在执行使用了符号引用的字节码前，Java 虚拟机需要解析这些符号引用，并替换为实际引用。

对于非接口符号引用，假定该符号引用所指向的类为 C，则 Java 虚拟机会按照如下步骤进行查找。

1. 在 C 中查找符合名字及描述符的方法。
2. 如果没有找到，在 C 的父类中继续搜索，直至 Object 类。
3. 如果没有找到，在 C 所直接实现或间接实现的接口中搜索，这一步搜索得到的目标方法必须是非私有、非静态的。并且，如果目标方法在间接实现的接口中，则需满足 C 与该接口之间没有其他符合条件的目标方法。如果有多个符合条件的目标方法，则任意返回其中一个。

从这个解析算法可以看出，静态方法也可以通过子类来调用。此外，子类的静态方法会隐藏（注意与重写区分）父类中的同名、同描述符的静态方法。

对于接口符号引用，假定该符号引用所指向的接口为 I，则 Java 虚拟机会按照如下步骤进行查找。

1. 在 I 中查找符合名字及描述符的方法。
2. 如果没有找到，在 Object 类中的公有实例方法中搜索。
3. 如果没有找到，则在 I 的超接口中搜索。这一步的搜索结果的要求与非接口符号引用步骤 3 的要求一致。

经过上述的解析步骤之后，符号引用会被解析成实际引用。对于可以静态绑定的方法调用而言，实际引用是一个指向方法的指针。对于需要动态绑定的方法调用而言，实际引用则是一个方法表的索引。具体什么是方法表，我会在下一篇中做出解答。

总结与实践

今天我介绍了 Java 以及 Java 虚拟机是如何识别目标方法的。

在 Java 中，方法存在重载以及重写的概念，重载指的是方法名相同而参数类型不相同的方法之间的关系，重写指的是方法名相同并且参数类型也相同的方法之间的关系。

Java 虚拟机识别方法的方式略有不同，除了方法名和参数类型之外，它还会考虑返回类型。

在 Java 虚拟机中，静态绑定指的是在解析时便能够直接识别目标方法的情况，而动态绑定则指的是需要在运行过程中根据调用者的动态类型来识别目标方法的情况。由于 Java 编译器已经区分了重载的方法，因此可以认为 Java 虚拟机中不存在重载。

在 class 文件中，Java 编译器会用符号引用指代目标方法。在执行调用指令前，它所附带的符号引用需要被解析成实际引用。对于可以静态绑定的方法调用而言，实际引用为目标方法的指针。对于需要动态绑定的方法调用而言，实际引用为辅助动态绑定的信息。

在文中我曾提到，Java 的重写与 Java 虚拟机中的重写并不一致，但是编译器会通过生成桥接方法来弥补。今天的实践环节，我们来看一下两个生成桥接方法的例子。你可以通过“javap -v”来查看 class 文件所包含的方法。

```
interface Customer {
  boolean isVIP();
}

class Merchant {
  public Number actionPrice(double price, Customer customer) {
    ...
  }
}

class NaiveMerchant extends Merchant {
  @Override
  public Double actionPrice(double price, Customer customer) {
    ...
  }
}

class Merchant<T extends Customer> {
  public double actionPrice(double price, T customer) {
    ...
```

```
  }
}


class VIPOnlyMerchant extends Merchant<VIP> {
  @Override
  public double actionPrice(double price, VIP customer) {
    ...
  }
}
```

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html
[2] https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html
[3] https://wiki.openjdk.java.net/display/HotSpot/VirtualCalls
[4] https://wiki.openjdk.java.net/display/HotSpot/InterfaceCalls

通过留言可与作者互动