类与接口（4）：方法重载解析

一、方法重载简介

方法重载： 当两个（或多个）方法的名称相同，而参数的对应类型或个数不同时，我们就说方法重载了。当然，编译器也能识别出来。

编译器是如何识别调用了哪个方法？

在往下讲前，我们先来了解一下：编译器是怎么才能识别出程序调用了那个方法。其实，这个问题就是在问：在调用方法处，编译器能得到调用方法的什么信息，从而能找到对应的方法？我们一般的方法调用是这样的：

```
method( vars );
```

也就是说，方法调用处，一共为编译器提供两个信息：方法名、参数列表。

所以，**编译器只能通过方法名和参数列表来识别调用方法。**

有一道面试题问：**为什么不能通过返回类型来重载方法？**

就是上面所说的，方法调用处并没有提供返回类型的信息，所以当多个方法只有返回类型不一样时，编译器就不知道调用了那个方法了。

我们已经知道了编译器是怎么识别方法的了，**而对于方法重载，其要求方法名是一样的，那么我们只需要关注参数列表便可以了。参数列表区分，或者说重载方法的区分：**

- 参数的个数
- 参数的类型
- 参数的顺序

二、方法重载的匹配选择

方法重载后，方法调用处可能会遇到应该选择哪个重载方法的问题，如果只有唯一个重载方法可以匹配，那么就没问题。然而，大部分情况却是有多个重载方法是可以匹配的，那么这时候就应该选择最合适的重载方法.

匹配最合适、最明确的重载方法，其实就是实参列表去匹配当前重载方法中形参列表，寻找与实参列表最相近的形参列表。

1. 基本类型之间的重载

对于基本类型来说，从“短类型”扩展成“长类型”是默认允许、自动进行的，这就可能造成了实参可能匹配到多个“长类型”的形参，看个简单例子：

```
public static void main(String[] args) {
```

```
    short s = 4;
    m(s);
}

public static void m(int x){//方法一
    System.out.println("重载方法一");
}

public static void m(float x){//方法二
    System.out.println("重载方法二");
}
```

运行结果：

```
重载方法一
```

short 类型可以默认自动转换成 int、'float'类型。但 m(s)真正匹配选择的是 m(int x)方法，而不是形参长度更长的 m(float x)。所以可以看出，**基本类型的形参匹配规则是**： 如果没有匹配到精确类型的形参，则优先匹配存储长度(范围)大于且是最接近实参的存储长度的形参，从而确定调用哪个重载方法。

2. 引用类型间的重载

对于引用类型来说，可以匹配到多个重载方法的原因是：引用类型的对象进行类型上转也是 JVM 默认自动进行的,那么就可能匹配多个祖先类型的形参看下面的例子：

```
public class Test_3 {
   public static void main(String[] args)  {
       Children children = new Children();
       someMethod(children);
   }

   public static void someMethod(Ancestor an) {//重载方法 1
       System.out.println("this is Ancestor Method!");
   }

   public static void someMethod(Parent an) {//重载方法 2
       System.out.println("this is Parent Method!");
   }
}
//3 个具有继承关系的类
class Ancestor{//祖先类
}
```

```
class Parent extends Ancestor{//父类，继承于 Ancestor
}
class Children extends Parent{//子类，继承于 Parent
}
```

运行结果：

```
this is Parent Method!
```

可以看出，**引用类型与基本类型一样，都是选择”最明确的方法“， 引用类型间选择最明确的重载方法的规则是： 如果找不到重载方法的形参的引用类型与实参一致，则实参优先匹配 在继承树结构上，离实参类型最近的形参**，则此形参所在的重载方法便是最明确的重载方法。

3. 自动装箱拆箱、可变参数类型

装箱拆箱、以及可变参数列表的处理都是由编译器自动处理，也就是说是默认自动进行的，这同样会让实参列表可以匹配多个形参列表 ,可以匹配多个重载方法。

此小节将会涉及到基本类型、引用类型、自动装箱拆箱可变参数的重载方法匹配的优先级。

看下面的例子，这个例子包括很多情况：

```
public class Test_3 {
   public static void main(String[] args)  {
       short s = 5;
       overloadMethod(s);// test1
       Integer i = 10;
       overloadMethod(i);//test2
       overloadMethod(s,s);//test3
   }

   public static void overloadMethod(int a) { //m1
       System.out.println("调用  overloadMethod(int)");
   }

   public static void overloadMethod(Short in) {//m2
       System.out.println("调用  overloadMethod(short)");
   }

   public static void overloadMethod(int a,int b) {//m3
       System.out.println("调用  overloadMethod(int,int)");
   }
```

```
    public static void overloadMethod(short... s) { //m4
     System.out.println("调用  overloadMethod(short...)");
    }

    public static void overloadMethod(Integer... i) {//m5
        System.out.println("调用  overloadMethod(Integer...)");
    }
}
```

运行结果

```
调用 overloadMethod(int)
调用 overloadMethod(int)
调用 overloadMethod(int,int)
```

我们来分析一下上面的例子中，方法调用处可以匹配到的方法：

- test1 处的方法调用可以匹配的重载方法有：m1（基本类型的短类型自动转为长类型）、m2（自动装箱）、m4（可变参数列表）
- test2 处的方法调用可以匹配的重载方法有：m1(自动拆箱)、m5（可变参数列表）;
- test3 处的方法调用可以匹配的重载方法有：m3（基本类型的短类型自动转换成长类型）、m4（可变参数列表）

查看输出结果，发现：test1 处选择了 m1、test2 选择了 m1，test3 选择了 m3。

根据这样的结果，也就是这几种形参匹配规则还是有个匹配的顺序的。对重载方法的选择作以下总结：

- 先按照实参的类型（基本类型或引用类型）对应匹配规则，进行查找最相近的形参列表，从而找到最明确的重载方法；找不到，则执行第二步；
- 对实参进行装箱或拆箱转换（前提是实参是基本类型或者是包装类），再安按照转换得到的类型进行匹配形参的类型（形参类型与转换类型要一致，特别注意基本类型）；找不到，则执行第三步；
- 匹配形参是可变参数的重载方法，此时，形参的类型可以是 实参的类型以及通过 基本类型的短转长、自动装箱拆箱、祖先类型 得到的转换类型。

将上面的总结再简化一下，可以简化成 **重载方法的形参匹配规则的优先级**：

当前类型（基本类型或引用类型）的匹配规则 > 自动装箱拆箱 > 可变参数列表

再看一个例子：

```
public class MyTest {
    public static void main(String[] args) {
      int a = 5;
      short s = 8;
      m(a,s);
    }
```

```
public static void m(int a,Short b) {//m1
    System.out.println("调用了 m(int , Short)");
}

public static void m(float f,short s) {//m2
    System.out.println("调用了 m(float,short)");
}
}
```

运行结果：

```
调用了 m(float,short)
```

分析： 实参都是基本类型，优先考虑形参列表都是基本类型的重载方法，找不到才考虑自动装箱拆箱。

4. 泛型方法的重载

泛型方法的重载规则： 将泛型方法的类型变量擦除，然后与非泛型方法一样，按照上面所说的三种规则一一匹配

```
public static void main(String[] args)  {
   //创建 Runnable 对象
   Runnable r = new Runnable() { public void run(){} };
   //调用泛型方法
   m(r);
}

public static <T> void m(T t) {//m1
    System.out.println("调用了<T> void m(T)");
}

public static <T extends Runnable> void m(T t) {//m2
    System.out.println("调用了<T extends Runnable> void m(T t)");
}
```

运行结果：

```
调用了  void m(T t)
```

上面的两个泛型方法 m(T t)进行类型擦除后是：

```
public static void m(Object t);
public static void m(Runnable t);
```

显然，调用方法应该是 m2,与运行结果相符；

5. 没法确定的重载方法调用

尽管编译器会按照上面所说的三种优先级别去让实参匹配形参，然而匹配的结果却不一定是唯一的，也就是说会匹配到多个方法，从而无法确定调用那个方法，编译失败。

情况一： 实参列表的所有最佳匹配的形参不在同一个方法中

```
public class MyTest {
    public static void main(String[] args) {
      int aa = 5;
      short ss = 8;
      m(aa,ss);//编译不通过，无法确定调用了那个重载方法
    }

   public static void m(int a,double b) {//m1
       System.out.println("调用了 m(int，Short)");
   }

   public static void m(float f,int c) {//m2
        System.out.println("调用了 m(float,short)");
   }
}
```

分析：

m(aa,ss)的调用编译失败，因为实参 aa 的最佳匹配 m(int，double)的第一个形参，而实参 ss 的最佳匹配则是 m(float，short)的第二个形参。

因此，实参列表的(aa,ss)的最佳形参类型匹配分开在了两个重载方法中。

注意一下，即使某个重载方法的形参列表包含最多的最相近的形参类型，只要不是全部，那么依旧无法确定调用了哪个重载方法。

情况二： 可变参数列表的特殊性 -- 无法根据可变参数的类型来重载方法

```
public static void m(short... s) {}
public static void m(Short... s) {}
public static void m(int... s) {}
```

调用测试例子：

```
short s = 8;
Short sl = 10;
m(s,s);//编译不通过
m(s,sl);//编译不通过
m(sl,sl);//编译不通过
```

重写与重载的区别

- 重写是针对父类与子类间的方法，即必须先得继承父类的方法。而重载则没有这种限

制。

- 重写要求方法名字和参数列表都相同，而方法重载则只需要，方法名相同，参数列表不同就行了。
- 方法重载时，方法的调用是在编译时期就已经确定了调用那个方法；方法重写，则要在运行时，才能确定调用的是子类还是父类的方法。