计算机程序设计语言

Java8中的Lambda表达式

张 天

软件工程组

计算机科学与技术系

南京大学

本章简介

- 本章以Java8为例介绍Lambda表达式的用法和实现方法
  - 在Java8提出之前，Xtext就已经大量使用Lambda表达式作为它构造其语法树遍历、集合操作等的基本方式
  - Xtext提出了一种JVM语言**Xtend**，用于定义动作语义，其实现Lambda表达式的方法与Java8非常相似

- **注：本章的示例代码使用 Eclipse DSL luna，JDK8，Xtext 2.7**

Java8与函数式编程

- Java8中加入了Lambda表达式，用以支持函数式编程

- 函数式编程（functional programming）
  - 又称泛函编程，是一种编程范型，它将电脑运算视为数学上的函数计算，并且避免使用程序状态以及易变对象
  - 函数编程语言最重要的基础是 λ演算（lambda calculus）
  - λ演算的函数可以接受函数当作输入（引数）和输出（传出值）

Java8 Lambda表达式

- JDK8版本中新增加了Lambda表达式的功能，被认为是最具革命性的的增强之一
  - 如同范型的出现从根本上改变了Java一样，如今Lambda表达式的出现也将再一次从根本上改变Java[Java8BG]

- Lambda表达式的出现使Java成为一种混合式风格的编程语言
  - Java8是**命令式**与**函数式**混合风格的语言
  - 但Java8仍是纯面向对象的

Lambda表达式在Java中的实现

- Lambda表达式在Java中的实现方式有Java语言自身的特点，与其他语言的实现方式会有不同

- 两个重要结构

  （1）Lambda表达式自身：本质上就是一个匿名（即没有命名的）方法

  （2）函数式接口：仅包含一个抽象方法的接口

  注：深刻理解这两个重要结构是使用Java8中lambda表达式的关键所在！

（1）Lambda表达式自身

- Lambda表达式本质上就是一个匿名方法
  - 该方法没有命名，也就不能独立执行
  - 该方法的执行是通过实现了函数式接口定义的另一个方法
  - 因此， lambda表达式的使用会导致产生一个匿名类

（2）函数式接口

- 函数式接口（Functional Interface）是**仅包含一个抽象方法**的接口
  - 关键字：抽象方法、接口
  - 从该接口的定义上看，并没有使用任何特殊的关键字用于标示其为lambda表达式，就是一个普通的接口
  - 该接口不普通的地方就是，它仅仅只定义了一个抽象方法
- 函数式接口常常被称作SAM类型，即**单抽象方法**（Single Abstract Method）

Lambda表达式的基础知识

- Lambda表达式运算符 “->”
  - Java8 新增的运算符
  - Lambda表达式的左部和右部
- Lambda表达式的形式参数
  - 形参的数量没有限制
  - 形参的类型主要依靠推断而尽量避免显式声明

Lambda表达式运算符

- 为了表达lambda表达式，Java8引入了新的语法和运算符，即lambda运算符或箭头运算符“->”

- 它将lambda表达式分成两个部分
  - 左部指定了lambda表达式需要的所有参数
  - 右部是lambda表达式的体，指定了lambda表达式的动作
  - 注：Java8定义了两种lambda表达式体，一种是纯表达式，另一种则包含了代码块。前者是标准风格的lambda形式，后者则是为了和命令式风格兼容的方式

简单示例

- 最简单的例子，计算常量值：
  - （）-> 3.1415926
  - 该lambda表达式返回一个常量值，并且返回值的类型**推断**为**double**类型
  - 该表达式的作用类似于如法方法：

    Double Pi（） { return 3.1415926; }
- 注意
  - 这个lambda表达式没有形参，所以形参列表（左部括号部分）为空

更常见的示例

- 调用Java原生API的示例
  - **( ) -> Math.random()* 100**
  - 该表达式通过调用JDK的 Math.random() 获得一个伪随机数，将其乘以100，然后返回结果
- 带形参的例子
  - **(n) -> 1.0 / n**
  - 这个lambda表达式返回n的倒数
  - 注意，尽管可以显式指定形参的类型，但通常不需要这么做，大部分情况下，形参的类型可以**推断**出来（事实上，lambda表达式习惯于不直接给出，而通过推断获得形参的类型）
  - Lambda表达式支持使用任意数量的形参

Lambda表达式语法灵活性

- Java8提供了一些简化写法的语法便利（Syntax Sugar）
  - 当lambda表达式仅有一个形参的时候，不必将左部指定的形参名称用圆括号括起来
  - 例如，**(n) -> (n % 2) == 0** 可以简写成 **n -> (n % 2)== 0**
  - 和形参不一样的是，返回类型是**必须**不显式指定的，因为一定可以**推断**出来
- 尽管Java8提供了很多类似的语法便利，但**个人**强烈建议大家尽量不要使用，而是用**中规中矩**的方式编写代码！

函数式接口

- 函数式接口是指仅指定了一个抽象方法的接口
  - 注意，从JDK8开始，可以为接口声明的方法指定一个或多个**默认方法**，而默认方法是非抽象方法（Java8的重大变化）
  - 函数式接口可以包含默认方法，但在任意情况下，都必须“**有且仅有一个**”抽象方法
  - 特别注意，这里强调的“**有且仅有一个**”只是针对抽象方法而言

简单示例

- 一个简单而典型的函数式接口

```
interface MyValue {
    double getValue() ;
}
```

- 在这个例子中，getValue() 方法隐式的是抽象方法，并且也是MyValue接口的唯一方法
- 因此，MyValue是一个函数式接口，其功能由getValue() 方法定义

使用Lambda表达式

- Lambda表达式不是独立执行的，而是通过对函数式接口中定义的抽象方法的实现来使用的
- **目标类型的上下文**
  - 问题：什么地方能够使用这个lambda表达式呢？或者，所定义的lambda表达式能够使用在什么样的上下文环境中呢？
  - 类比于方法的使用：方法的使用是通过**方法名**、**参数列表**以及**返回类型**来明确的
  - Lambda表达式其实也类似，当把一个lambda表达式赋值给一个函数式接口的引用时，就创建了一个使用该表达式的上下文（即目标类型的上下文）
  - 目标类型的上下文包括：**变量初始化**、**return语句**和**方法实参**等

MyValue的上下文示例

- 首先声明对函数式接口 MyValue 的一个引用：
  - MyValue myVal;
- 然后，将一个lambda表达式赋给该接口引用：
  - myVal = () -> 3.1415926;
- 该lambda表达式与 getValue() 兼容，是因为它们都没有形参，并且都是返回一个double类型的值

注1：lambda表达式的返回值类型是通过**推断**得到的！

注2：函数式接口的抽象方法的类型必须与lambda表达式的类型兼容，否则就会导致编译时错误

函数式接口的初始化

- 函数式接口的初始化发生在其目标类型上下文中出现lambda表达式的时候，如MyValue示例中：
  - **myVal = () -> 3.1415926;**
  - 或者，也可以将声明和赋值两条语句合并到一条语句中，即**MyValue myVal = () -> 3.1415926;**
- 这时候，JDK会**自动隐式地**创建实现了函数式接口的一个具体类的**实例**，其中方法的实现体就是该lambda表达式的内容

对上例的一个思考

- 注意到，MyValue是一个接口（函数式接口也还是接口），而myVal则是一个实例变量，那么该接口的实现体（通过implement关键字）在哪里定义？
- 答案是不需要创建，JDK也不会在代码中显示生成
  - Option1：隐式地替用户创建（字节码中）
  - Option2：不创建类，而直接在实例化的时候创建实例并传给函数式接口的引用（可能性不大）
- 在代码中将lambda表达式赋值给函数式接口的声明位置，JDK就会隐式地创建该实现体的实例

调用lambda表达式

- 当通过**实例引用**调用函数式接口中声明的方法时，就会执行lambda表达式
  - 例如：

    System.out.println(“A constant value: ” + myVal.getValue());
  - 执行结果：

    A constant value: 3.1415926

- 注：可以看出，lambda表达式为Java8提供了一种将代码片段转换为对象的方法

带形参的例子

```
interface MyParamValue {
    double  getValue(double v);
}
MyParamValue  myPval  =  (n) -> 1.0 / n ;
System.out.println("Reciprocal of 4 is " + myPval.getValue(4.0) );
```

- 注意，并没有指定**n**的类型，但可以从上下文推断它的类型（显式声明也允许，但不建议使用）。
- 为了在目标类型的上下文中使用lambda表达式，抽象方法的形参数量和类型必须和lambda表达式的形参的数量和类型**兼容**

关于类型兼容的理解

- 函数式接口中抽象方法的形参数量和类型必须和lambda表达式的形参的数量和类型兼容
- 这里所谓的兼容是从Java语言类型之间的兼容性来说的
  - 强类型语言会在表达式中使用隐式类型转换，但必须遵守一定的类型匹配原则
  - 例如，double类型可以接收int类型的数据，但不能接收boolean类型的数据

两个类型兼容性的示例

- 例如上例可改为：
  - MyParamValue myPval = (n) -> 1 + n ;
  - 类型匹配原则：double类型可以接收int类型的数据

- 但不可以改为：
  - MyParamValue myPval = (n) -> **ture** ;
  - 类型匹配原则：double类型不能接收boolean类型的数据
  - 注意，Java编译环境会提示Error如下：
    - **Type mismatch: cannot convert from boolean to double**

关于形参类型的显式声明

- 虽然没有必要，但Java8也允许对lambda表达式中的形参进行显式声明

- 如果显式声明，列表中的所有形参都要声明
  - 合法的声明：

    （int n, int d） -> （n % d）== 0
  - 非法的声明方式：

    （int n, d） -> （n % d）== 0

    （n, int d） -> （n % d）== 0

块lambda表达式

- 以上示例中的lambda表达式体都只包含单个表达式，这样的体被称为**表达式体**，这样的lambda表达式也被称为**表达式lambda**
- 但有些情况下，会需要用到多个表达式，这时就需要用到块lambda表达式
  - 右部由一个代码块构成，其中可以包含多条语句
  - 这样的表达式体称为**块体（block body）**，具有块体的lambda表达式也成为**块lambda**

块体的构成

- 在块lambda中，可以声明变量、使用循环、指定if和switch语句、创建嵌套代码块等
- 创建块lambda只需要用花括号将lambda体括起来即可
  - 例如：

```
NumericFunc smallestF = (n) -> {
    int result = 1;

    // Get absolute value of n.
    n = n < 0 ? -n : n;

    for (int i = 2; i <= n / i; i++)
        if ((n % i) == 0) {
            result = i;
            break;
        }
    System.out.println("inside block");
    return result;
};
```

不可缺少的return

- 在块lambda中**必须**显式地使用return语句来返回值
  - 之所以必须用return来指定返回值，是因为块lambda体代表的不再是一个单个的表达式
  - 单个表达式的返回值是明确的，而且类型也完全可以推断出来
- 注意
  - 可以看出，块lambda表达式已经比较接近Java中普通方法的定义，而和经典的lambda表达式（从数学中函数的概念来看）不太一样了，这是lambda表达式在命令式语言上实现时的权衡

块lambda示例

```
interface NumericFunc {
    int func(int n);
}

class BlockLambdaDemo {
    public static void main(String args[]) {

        // This block lambda returns the smallest positive factor of a value.
        NumericFunc smallestF = (n) -> {
            int result = 1;

            // Get absolute value of n.
            n = n < 0 ? -n : n;

            for (int i = 2; i <= n / i; i++)
                if ((n % i) == 0) {
                    result = i;
                    break;
                }
            System.out.println("inside block");
            return result;
        };

        System.out.println("Smallest factor of 12 is " + smallestF.func(12));
        System.out.println("Smallest factor of 11 is " + smallestF.func(11));
    }
}
```

运行结果

- 上例是一个求最小因子的程序
- 可以看到在块lambda的体内，可以像普通方法体内部一样使用各种控制结构和函数调用
  - 例如打印语句，System.***out.println("inside block");***
- 运行结果，控制台打印如下：

inside block
Smallest factor of 12 is 2
inside block
Smallest factor of 11 is 1

范型函数式接口

- Lambda表达式自身不直接是范型的，但与lambda表达式关联的函数式接口可以是范型的
  - 这时，lambda表达式的目标类型部分由声明函数式接口引用时指定的**实参类型**来决定
  - 理解范型函数式接口和理解传统Java中的范型编程相似

范型函数式接口示例

传统Java范型编程的类型形参<T>

```
interface SomeTest<T> {
        boolean test(T n, T m);
}
```

- 范型函数式接口的定义部分
  - **SomeTest<T>中的<T>是传统范型编程的基本写法，没有任何特别之处**

```
class GenericFunctionalInterfaceDemo {
    public static void main(String args[]) {
        // This lambda expression determines if one integer is
        // a factor of another.
        SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;

        if (isFactor.test(10, 2))
            System.out.println("2 is a factor of 10");
        System.out.println();

        // The next lambda expression determines if one Double is
        // a factor of another.
        SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;

        if (isFactorD.test(212.0, 4.0))
            System.out.println("4.0 is a factor of 212.0");
        System.out.println();

        // This lambda expression determines if one string is
        // part of another.
        SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;

        String str = "Generic Functional Interface";

        System.out.println("Testing string: " + str);

        if (isIn.test(str, "face"))
            System.out.println("'face' is found.");
        else
            System.out.println("'face' not found.");
    }
}
```

SomeTest<Integer>中的Integer是形参的实例化

作为实参传递lambda表达式

- Lambda表达式可以用在任何提供了目标类型的上下文中
- 前面示例的目标上下文是赋值和初始化，还可以将lambda表达式作为实参传递
- 事实上，这是lambda表达式的一种常见且强大的用途，因为可以将可执行代码作为实参传递给方法

实参传递lambda表达式示例

- 这个示例用lambda表达式创建三个字符串函数
  - 颠倒字符串
  - 用连字符替代空格
  - 颠倒字符串中字母的大小写

```
interface StringFunc {
    String func(String str);
}

static String changeStr( StringFunc sf , String s) {
    return sf.func(s);
}
```

以函数式接口作为第一参数形参类型，实际上是用于接收**赋值过lambda表达式的实例**，第二个参数是被处理的字符串

1. 颠倒字符串实现与调用

```
// Define a lambda expression that reverses the contents
// of a string and assign it to a StringFunc reference variable.
StringFunc reverse = (str) -> {
    String result = "";

    for (int i = str.length() - 1; i >= 0; i--)
        result += str.charAt(i);

    return result;
};

// Pass reverse to the first argument to changeStr().
// Pass the input string as the second argument.
outStr = changeStr(reverse, inStr);
System.out.println("The string reversed: " + outStr);
```

用lambda表达式初始化一个reverse实例引用，这个块lambda实现了颠倒字符串的功能

将reverse实例引用作为实参传递给changeStr函数，实现对lambda表达式的调用

2.用连字符替代空格

注意这里是直接将lambda表达式写在传递实参的位置！

```
// This lambda expression replaces spaces with hyphens.
// It is embedded directly in the call to changeStr().
outStr = changeStr((str) -> str.replace(' ', '-'), inStr);
System.out.println("The string with spaces replaced: " + outStr);
```

- 这有点像Java中的匿名内部类，可以带来结构清晰的好处
- 注意，这跟前面提到的语法便利（Syntax Sugar）还不太一样，这样做的好处是可以避免单独创建一个实例引用（这在匿名内部类的应用上好处尤为明显）

3. 颠倒字符串中字母的大小写

```
// This block lambda inverts the case of the characters in the
// string. It is also embedded directly in the call to changeStr().
outStr = changeStr( (str) -> {
    String result = "";
    char ch;

    for (int i = 0; i < str.length(); i++) {
        ch = str.charAt(i);
        if (Character.isUpperCase(ch))
            result += Character.toLowerCase(ch);
        else
            result += Character.toUpperCase(ch);
    }
    return result;
}, inStr);

System.out.println("The string in reversed case: " + outStr);
```

注意这里是直接将lambda表达式写在传递实参的位置！

对外层作用域的访问

- 在lambda表达式中，可以访问其外层作用域中定义的变量，如外层类的**实例**和**静态变量**
- Lambda表达式也可以显式或隐式地访问**this** 变量
  - 该this引用指向的是该lambda表达式外层类的实例
  - 因此，lambda表达式可以获取后设置其外层类的实例或静态变量的值，以及调用其外层类定义的方法
- 当lambda表达式使用其外层作用域内定义的局部变量时，会产生一种特殊的情况，称为**变量捕获（variable capture）**

关于获得外层类实例的this引用

- 能够获得外层类实例的this引用是非常重要的一种能力，通过这个this，就可以直接访问该外层类实例的所有成员，包括私有成员
- Java内部类（inner class）的实例就是通过获得外部类的this引用来访问其所有成员的（同样包括私有）
  - Java内部类实例化时，**必须** 隐式地获得一个外部类的this引用，因此也强制了必须现有外部类的实例，才能有内部类实例

变量捕获

- 变量捕获是指lambda表达式使用了其外层作用域定义的局部变量
- 在这种情况下，lambda表达式只能使用实质上为**final**的局部变量
  - 从本质上看，final是指在**第一次赋值**以后，**值不发生变化**的变量
  - 只要本质上属于final的即可，并不一定要显示地声明为final
- **说白一点，lambda表达式就是不能修改外层作用域内的局部变量！**
- 这也是对“函数式编程”无**函数副作用**的支持

变量捕获的反作用

```
interface MyFunc {
    int func(int n);
}

class VarCapture {
    public static void main(String args[]) {
        // A local variable that can be captured.
        int num = 10;
//        num = 9;
        MyFunc myLambda = (n) -> {
            // This use of num is OK. It does not modify num.
            int v = num + n;

            // However, the following is illegal because it attempts
            // to modify the value of num.
            // num++;

            return v;
        };
        // Use the lambda. This will display 18.
        System.out.println(myLambda.func(8));

        // The following line would also cause an error, because
        // it would remove the effectively final status from num.
//        num = 9;
    }
}
```

特别注意这两个对局部变量的修改都是非法的！

会引起编译出错，提示:
Local variable num defined in an enclosing scope must be final or effectively final

可见，一旦**lambda**表达式使用了外层的局部变量，就好像“捕获”该变量一样

从lambda表达式抛出异常

- 可以从lambda表达式中抛出异常，但必须在函数式接口中抽象方法的throws子句列出的异常兼容

```
interface MyIOAction {
  boolean ioAction(Reader rdr) throws IOException;
}

class LambdaExceptionDemo {

  public static void main(String args[])
  {
    double[] values  = { 1.0, 2.0, 3.0, 4.0 };

    // This block lambda could throw an IOException.
    // Thus, IOException must be specified in a throws
    // clause of ioAction() in MyIOAction.
    MyIOAction myIO = (rdr) ->  {
      int ch = rdr.read(); // could throw IOException
      // ...
      return true;
    };
  }
}
```

方法引用

- 方法引用（method reference）也是Java8引入的一个重要特性，用于提供一种引用方法而不执行方法的方式
- 该特性与lambda表达式相关，也需要有兼容的函数式接口构成目标类型上下文
- 在使用时，方法引用也会创建函数式接口的一个实例

- 注1：方法引用的主要用途其实就是为函数式接口中的抽象方法赋予具体实现的方式
- 注2：从本质上看，方法引用和lambda表达式起到对函数式接口同样的作用（即方法实现体）

方法引用分类

- 方法引用的类型有很多种，主要可以分为两大类
  - 静态方法引用
  - 动态方法引用

- 注，在体会方法引用的用法时，要注意从**如何获得一个方法的实现体**的角度去把握

静态方法的方法引用

- 静态方法的方法引用语法：
  - **ClassName::methodName**
  - 类名后面跟上方法名，中间用双冒号隔开
  - 注，“ :: ”是JDK8新增的分隔符，专门用于此目的
- 下面通过程序示例展示
  - 先定义一个函数式接口IntPredicate，包含一个test()方法
  - 再创建MyIntPredicates类，其中包含三个**静态方法**：isPrime、isEven和isPositive(int)，用于引用给函数式接口的抽象方法，作为其实现体
  - 定义一个测试方法，并在main函数中调用

接口和静态方法定义

```
interface IntPredicate {
    boolean test(int n);
}
// This class defines three static methods that check an integer
// against some condition.
class MyIntPredicates {
    // A static method that returns true if a number is prime.
    static boolean isPrime(int n) {
        if (n < 2)
            return false;
        for (int i = 2; i <= n / i; i++) {
            if ((n % i) == 0)
                return false;
        }
        return true;
    }

    // A static method that returns true if a number is even.
    static boolean isEven(int n) {
        return (n % 2) == 0;
    }

    // A static method that returns true if a number is positive.
    static boolean isPositive(int n) {
        return n > 0;
    }
}
```

重点关注函数式接口中抽象方法的**参数列表**和**返回类型**，看它是否与静态方法的**参数列表**和**返回类型**相匹配

测试方法和main

```
// This method has a functional interface as the type of its
// first parameter. Thus, it can be passed a reference to any
// instance of that interface, including one created by a
// method reference.
static boolean numTest(IntPredicate p, int v) {
    return p.test(v);
}

public static void main(String args[]) {
    boolean result;

    // Here, a method reference to isPrime is passed to numTest().
    result = numTest(MyIntPredicates::isPrime, 17);
    if (result)
        System.out.println("17 is prime.");

    // Next, a method reference to isEven is used.
    result = numTest(MyIntPredicates::isEven, 12);
    if (result)
        System.out.println("12 is even.");

    // Now, a method reference to isPositive is passed.
    result = numTest(MyIntPredicates::isPositive, 11);
    if (result)
        System.out.println("11 is positive.");
}
```

用于方法引用并对所引用的方法进行调用测试

分别引用三个静态方法：
MyIntPredicates::*isPrime*
MyIntPredicates::*isEven*
MyIntPredicates::*isPositive*
注意，跟作为实参lambda表达式一样！

实例方法的引用

- 要引用具体对象（实例）的某个方法，基本语法：
  - **objRef::methodName**
  - 与静态方法引用的语法类似，只不过改为了对象引用 objRef
- 下面通过程序示例展示
  - 先定义一个函数式接口IntPredicate，包含一个test()方法
  - 再创建MyIntNum类，其中包含isFactor**方法**，用于引用给函数式接口的抽象方法，作为其实现体
  - 在main函数中创建**两个**NyIntNum实例，分别引用和测试

接口和方法定义

```
//A functional interface for numeric predicates that operate
//on integer values.
interface IntPredicate2 {
    boolean test(int n);
}

// This class stores an int value and defines the instance
// method isFactor(), which returns true if its argument
// is a factor of the stored value.
class MyIntNum {
    private int v;
    MyIntNum(int x) {
        v = x;
    }
    int getNum() {
        return v;
    }

    // Return true if n is a factor of v.
    boolean isFactor(int n) {
        return (v % n) == 0;
    }
}
```

重点关注函数式接口中抽象方法**test()**的参数列表和返回类型，看它是否与**isFactor()**方法的参数列表和返回类型相匹配

在main中测试

思考：这个例子中最主要的是体会用对象的方法和静态方法有何区别！

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

    MyIntNum myNum = new MyIntNum(12);
    MyIntNum myNum2 = new MyIntNum(16);

    // Here, a method reference to isFactor on myNum is created.
    IntPredicate2 ip = myNum::isFactor;

    // Now, it is used to call isFactor() via test().
    result = ip.test(3);
    if (result)
        System.out.println("3 is a factor of " + myNum.getNum());

    // This time, a method reference to isFactor on myNum2 is created.
    // and used to call isFactor() via test().
    ip = myNum2::isFactor;
    result = ip.test(3);
    if (!result)
        System.out.println("3 is not a factor of " + myNum2.getNum());
}
```

注意：通过初始化，myNm和myNum2具有了不同的初始值（int v）：
myNum.v = 12
myNum2.v = 16

运行结果

- 通过运行Run As | Java Application，控制台显示：

  3 is a factor of 12

  3 is not a factor of 16

- 可以看出，两个不同对象的方法引用，使抽象方法绑定了不同的方法实现体，注意区别就是其中的成员变量**int v**具有不同的值（变量**v**绑定了不同的值）
- 特别注意体会，lambda表达式也叫**闭包（closure）**，就是这个原因（似乎拥有一个独立的空间）

对方法引用的动态绑定

- 前面两个例子都可以理解为对方法的**静态绑定**（即**编译期**绑定），也可以通过面向对象动态绑定的特性，将**方法引用的绑定**推迟到动态运行时
- 这种用法也成为对**类方法的引用**，语法：
  - **ClassName::instanceMethodName**
- 面向对象中的动态绑定
  - 父类可以接收子类对象，因此对父类方法的调用可以实际发生在对子类对象方法的调用上
  - 这样就可以设计出多态行为，具体方法的执行由运行时所绑定对象的方法来决定

类方法引用的形式

- 方法引用部分的语法
  - **ClassName::instanceMethodName**
  - **类名**（不是**对象**，也不是**静态方法**）
- 函数式接口中的抽象方法也必须以该类做形参
  - 第一个形参是该类
  - 从第二个开始才是类中该方法的参数列表

类方法引用的示例

```
//A functional interface for numeric predicates that operate
//on an object of type MyIntNum2 and an integer value.
interface MyIntNumPredicate {
    boolean test(MyIntNum2 mv, int n);
}

// This class stores an int value and defines the instance
// method isFactor(), which returns true if its argument
// is a factor of the stored value.
class MyIntNum2 {
    private int v;
    MyIntNum2(int x) {
        v = x;
    }
    int getNum() {
        return v;
    }

    // Return true if n is a factor of v.
    boolean isFactor(int n) {
        return (v % n) == 0;
    }
}
```

特别注意**test**方法的第一个形参是**MyIntNum2**，第二个形参才和**isFactor(int n)**匹配！

类方法引用的调用

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

    MyIntNum2 myNum = new MyIntNum2(12);
    MyIntNum2 myNum2 = new MyIntNum2(16);

    // This makes inp refer to the instance method isFactor().
    MyIntNumPredicate inp = MyIntNum2::isFactor;

    // The following calls isFactor() on myNum.
    result = inp.test(myNum, 3);
    if (result)
        System.out.println("3 is a factor of " + myNum.getNum());

    // The following calls isFactor() on myNum2.
    result = inp.test(myNum2, 3);
    if (!result)
        System.out.println("3 is a not a factor of " + myNum2.getNum());
}
```

这里只引用了类的方法

注意这个地方就具有典型的面向对象**动态绑定**的特点

构造函数引用

- 与创建方法引用类似，也可以创建构造函数的引用
  - 语法：classname::new
  - 可以将该引用赋值给函数式接口，只要构造函数与其抽象方法在参数列表和返回类型上都一致

函数式接口和类的定义

```
interface MyFunc2 {
    MyClass func(String s);
}

class MyClass {
    private String str;

    // This constructor takes an argument.
    MyClass(String s) {
        str = s;
    }
    // This is the default constructor.
    MyClass() {
        str = "";
    }

    String getStr() {
        return str;
    }
}
```

注意该函数式接口中抽象方法**func**的**参数列表**和**返回类型**

注意：MyClass有两个构造函数，**参数列表**不同。

思考：哪一个构造函数可以与函数接口匹配？

main运行示例

```
class ConstructorRefDemo {
    public static void main(String args[]) {
        // Create a reference to the MyClass constructor.
        // Because func() in MyFunc takes an argument, new
        // refers to the parameterized constructor in MyClass,
        // not the default constructor.
        MyFunc2 myClassCons = MyClass::new;

        // Create an instance of MyClass via that constructor reference.
        MyClass mc = myClassCons.func("Testing");

        // Use the instance of MyClass just created.
        System.out.println("str in mc is " + mc.getStr());
    }
```

注意这里是函数接口的**初始化**部分，**隐式创建**该接口的实例，方法也已经被绑定了

具体的函数调用发生在这个位置，具体参数也需要在这里进行传递

参考文献

- [Java8CR]Java: The Complete Reference, Ninth Edition
- [Java8BG]Java: A Beginner's Guide, Sixth Edition
- [TIJ4] Think in Java 4th