---

title: （一）见鬼！看不懂的Lambda！
date: 2019-11-28
categories:

- JavaSE
- Java面向对象语法
- Lambda表达式系列
  author: 胡亦可

当有一天，有人把这段代码拿到你面前，并一脸真诚地告诉你这是段正确无误的Java代码。🤔告诉我，你是什么感觉？完全看不懂呀！是不是除了👻，就是该怀疑人生了？

```
List<String> strLst = Arrays.asList("Hello", "", "Java", "Hi", "Test");
List<String> strings filterStrings = strLst.stream()
                                    .filter(string ->!string.isEmpty())
                                    .collect(Collectors.joining(", "));
filterStrings.forEach(System.out::println);
```

好不容易熟悉了面向对象的语法，天天在new对象和“.”操作的海洋中遨游的你，是不是有一种被抛弃的感觉？
不用怀疑了。首先，这确实是你每天都在玩儿的Java；其次，这是正儿八经的JDK8中的新语法---Lambda表达式，不是什么歪门邪道；最后，你必须要搞定它，否则真有可能被Java抛弃掉.....

初探Lambda表达式

第一个问题，这个单词咋念？

> ***Lambda：***
> ***λ**是希腊字母表中排序第十一位的字母；*
> *读作:['læ:mdə]；英语名称为Lambda；汉字读音:拉姆达。*

这个希腊符号应该有同学在以前的求学之路中见过，只不过忘记了而已。它在高中物理中出现过，代表波长；它在线性代数中出现过，代表特征值；它在放射学中也出现过，代表衰变常数；如今，它只不过是再次出现在了编程语言当中，但是它代表了什么呢？🤔

作为一个有经验的程序员，面向搜索引擎编程是我们的基本技能，也是我们屡试不爽的编程利器。这次也不例外，百度一下呗.....

- “Lambda表达式是一种匿名函数”；
- “Lambda表达式就是闭包”；
- “这是函数式编程的语法”；
- “Lambda表达式带来了行为参数化”；
- “C++、Java、C#、Python都支持lambda表达式”。

😏，咋样？感觉有没有好一点？好吧，我知道你现在更晕乎了。这些奇奇怪怪的概念和专业术语，似乎想为你展现一个之前你完全没有接触过的编程一角，但是却让你除了朦胧不清就是一头雾水。

咋办呢？你看这么语言都已经支持lambda了，说明这是大势所趋呀，我们根本无法放弃它。但是别忘了，我们的身份。我们是程序员，我们的目标是能够快速地使用它，至少暂时不是分析和研究它的编程思想和涉及地编程理论。所以，让我们抛开那些高大上不接地气的东西，从我们能够了解和掌握的知识点入手，先从语法、从代码、从能做事情开始。这才是学习的王道！！！当我们对它有了一定的掌握层度之后，再来研究归纳总结理论吧。

对于概念的理解，我们首先找一个比较容易的切入点。既然刚才在百度中都提到了一个词汇：**函数**。这就好办了。所谓函数，那是我们再熟悉不过的东西了，也就是方法嘛，对应到面向对象中就是“行为”。那么，至少我们现在可以了解到，Lambda表达式操作的是**方法**，而不是属性数据，对吧?

接下来，我们就从基本语法开始入手吧。

Lambda表达式的基本语法

第一个Lambda表达式

Swing 是一个与平台无关的 Java 类库，用来编写图形用户界面(GUI)。该类库有一个常见的用法:为了响应用户操作，需要注册一个事件监听器。用户一触发事件，监听器就会执行相应定制的操作：

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event) {
        System.out.println("按钮被点击一次");
    }
});
```

在这个例子中，button 对象的 addActionListener（）方法需要传入一个 ActionListener 对象。可是 ActionListener 是一个接口类型，该接口只有一个叫 actionPerformed()的方法，当用户点击屏幕上的按钮时，button 就会调用这个方法。
于是，在上例中采用了传统的做法，即使用匿名内部类的方式。通过 new 一个 ActionListener 对象的同时，重写抽象的actionPerformed（）方法，然后把这个匿名对象传递到button的addActionListener（）方法当中。

匿名内部类是Java语法当中的一个难点，对于初学者并不友好。其实在上面的代码中，对于程序真正有意义的事情只有一件，那就是把System.out.println(("按钮被点击一次");这句代码告知给button按钮，让它知道被点击了以后要做什么事情。而为了达到这个效果，我们不得不去说明传入一个ActionListener对象，被重写的方法名是actionPerformed，还要关注各种括号的匹配。

如果把这段代码换成Lambda表达式呢？它就成了这个样子：

```
button.addActionListener(event -> System.out.println("按钮被点击一次"));
```

是不是一下子就舒心了很多，😏？从整体上看，Lambda表达式确实抛弃使用匿名内部类的繁琐，直接把最关键的actionPerformed代码段直接当成了一个参数传递给了addActionListener方法。
你看这是不是就是我们之前在百度上查到的**行为参数化**，即把一个方法当作参数传入到另一个方法当中去。

当然，深受Java标准面向对象的我们肯定还有一肚子问题，比如：

- 没有写new ActionListener()，那么咋知道是哪种类型的对象呢?
- 这个“->”符号，到底是什么意思呢?
- 如果方法里面不是一句打印代码，而是多句呢?
- event又是个啥?
- ……等等

别急，我们接下来一一说明。

类型推断

分析上面的匿名内部类实现方式和Lambda表达式，我们第一个问题就是button的addAction（）方法，凭什么知道我们传入的是一个ActionListener对象呢？我们在Lambda的实现方式中，明明提都没有提传入的对象类型。

其实这就是Java中提供的**类型推断**。其实类型推断早在JDK7就已经有了，比如：

```
Map<String, Integer> oldWordCounts = new HashMap<String, Integer>();
Map<String, Integer> diamondWordCounts = new HashMap<>();
```

早期的JDK在用泛型的时候必须使用第一种写法；而到了JDK7引入了菱形语法，才有了第二种写法。在这种菱形语法中，我们只需要为变量 diamondWordCounts声明了类型，而对真正的容器对象并没有告知键值对的数据类型，但是JDK7的编译器就可以自己推断出来，从而保证正确的数据类型匹配。
JDK8则更进一步，程序员可省略 Lambda 表达 式中的所有参数类型。编译器可以根据 Lambda 表

达式上下文信息 就能推断出参数的正确类型。程序依然要经过类型检查来保证运行的安全性，但不用再显 式声明类型罢了。这就是所谓的类型推断。

也就是说我们可以这么理解：由于类型推断的存在，JDK8的编译器自己通过button的addActionListener（）的参数要求，帮我们产生了一个ActionListener对象。

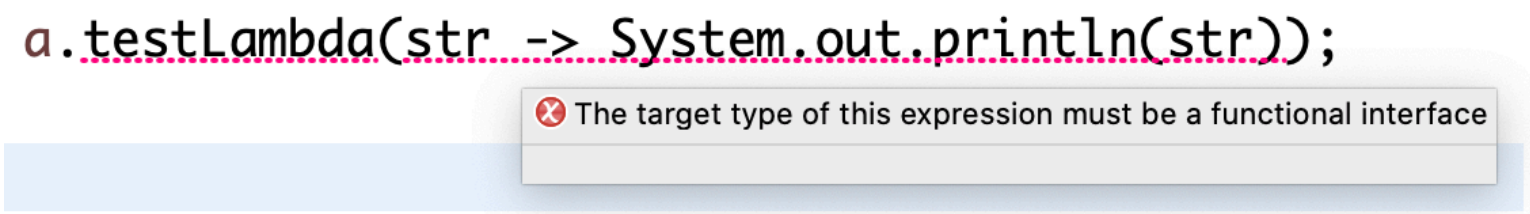
那么，是不是所有的抽象类型，在产生对象的时候都可以这么干呢？我们自己模仿写一个例子：

```
//抽象类
public abstract class A{
    public abstract void doSomething(String str);
}

//测试类
public class TestMain{
    public static void main(String[] args) {
    //匿名内部类实现
    A a = new A(){
       public void doSomething(String str) {
          System.out.println(str);
       }
    });
    a.doSomething("Hello");

    //Lambda表达式实现
    A aa = str -> System.out.println(str);
    aa.doSomething("Hello");
    }
}
```

写完代码一经编译，我们立马发现此处的Lambda表达式的实现方式给我们报编译时错误了。
{% img /img/20191128/1.png 500 50 %}



报错信息翻译过来时：这个表达式的目标类型必须是一个函数（functional）接口。

函数接口

接口，我们当然熟悉得不得了，但是函数接口，好像真的是第一次听说呢。这是个什么东西呢？难道是一种特殊的接口？管他的，先把B改为接口试一试。(就是这么粗暴👎)

```
//修改A的类型为接口
public interface A{
    public abstract void doSomething(String str);
}

//测试类
public class TestMain{
    public static void main(String[] args) {
    //匿名内部类实现
    A a = new A(){
        public void doSomething(String str) {
            System.out.println(str);
        }
    });
    a.doSomething("Hello");
    //Lambda表达式实现
    A aa = str -> System.out.println(str);
    aa.doSomething("Hello");
    }
}
```

😱，哎呀妈呀，对啦！！！编译通过，执行效果是：

```
Hello
Hello
```

那也就是说，所谓的函数接口就是一种接口，Lambda表达式在Java中本质上就是函数接口类型的。只有满足这种函数接口的语法的，才支持使用Lambda表达式。那是不是所有的接口都是函数接口呢？

这个我们就不一一探寻了，直接给出结论吧。
函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口(参见：JDK8接口的默认方法和静态方法语法)。函数式接口可以被隐式转换为 lambda 表达式。
另外，JDK8还有专门定义了一个注解 @FunctionalInterface 用于限定修饰这种接口。
@FunctionalInterface不是必须的，可以省略。

Lambda表达式语法揭秘

通过上面的学习，我们知道Lambda表达式就是对应了只有一个抽象方法的函数接口。它本身只需要关注的是这个抽象方法的实现部分，其它的都交给了Java编译期去进行推断和生成。所以，在语法上，我们需要学会Lambda表达式的各种书写，以满足抽象方法的变化。比如：

- 抽象方法没有参数 或 有多个参数呢?
- 抽象方法有返回值呢?
- 抽象方法的实现代码不仅仅是打印这一行代码而是多行呢?

我们首先给出Lambda表达式的标准语法：
（形参列表）->{ 代码段 }

示例：

```
public interface A{
    public int doSomething(String str,int num);
}

public class TestMain {

    public static void main(String[] args) {
        A a = (String str,int num) -> {
            for(int i = 0; i < num; i++) {
                System.out.println(str);
            }
            return "打印" + num + "次" + str;
        };

        String result = a.doSomething("OK,Done!", 5);
        System.out.println(result);
    }
}
```

编译执行后，效果如下：

```
OK,Done!
OK,Done!
OK,Done!
OK,Done!
OK,Done!
打印5次OK,Done!
```

这就是一个标准的、全面的Lambda表达式语法。语法讲解，先说关键的：

1. (String str,int num)是对应的抽象方法的形参数列表；
2. "->"是固定分隔符，左边是参数列表，右边是抽象方法的实现代码块；
3. {}当中遵循我们以往的一贯写法，可以有多条语句，逻辑控制语句，产生对象，调用方法，return返回等等均可；

有同学说，为啥这个和我们前几节的例子有点不一样呢？这是因为Lambda表达式在特殊情况可以变形。

1、如果函数接口的抽象方法无参，那么左边只需打上空括号：

```
() -> {......}
```

2、如果抽象方法的实现代码只有一句，那么右边可以省略“{}”、return关键字 和 这条语句的“;”号：

```
() -> 5 + 3
```

3、如果抽象方法带参，那么左边可以省略参数类型，形参名也无需保持一致：

```
(x,y) -> "打印" + x + "次" + y
```

4、如果抽象方法只有一个形参，那么左边可以省略“()”括号：

```
str -> System.out.println(str);
```

一个容易犯错的坑

其实这个坑是个老坑，一个从匿名内部类中带过来的问题。如果你熟悉匿名内部类的使用，你一定遇到过这样的问题：

```
public interface A{
    public abstract void doSomething(String str);
}

//测试类
public class TestMain{
    public static void main(String[] args) {
        int initNum = 8;
    //匿名内部类实现
```

```
    A a = new A(){
        public void doSomething(String str) {
            System.out.println(str + initNum);//编译不报错
            initNum = 10;//编译报错
            System.out.println(initNum);
        }
    });
    }
}
```

这个时候，我们会发现这段代码编译不通过。可是报错的代码不是第12行，而是第13行。这说明匿名内部类只能把它所在方法的局部变量当成一个常量使用。即：只能访问，不能修改！（这个问题的底层原因，这里就不讨论了，详细请看内部类的文章）。
其实在老版本的JDK中，这里的语法要求更加的严格，那就是initNum必须在声明的时候严格规范使用“final”关键字修饰。在JDK8当中，这个要求被省略了，但你仍然只能在匿名内部类中把它当作常量操作，否则编译不通过。
lambda表达式作为匿名内部类的简化语法，对这个问题同样保持相同的处理方案，所以请各位一定要注意！
lambda表达式中，如果需要操作该表达式所在方法的局部变量，那么只能将其作为一个final的常量进行操作。

Lambda的学习建议

至此，我们学到了Lambda表达式的基本语法和基本用途。基本语法就不再多说了，参看本文的红色部分即可。这里想再聊聊基本用途。

本文中，我在举例时都使用的是替换匿名内部类的场景。这当然不是Lambda表达式使用的唯一场景，但之所以这样选择，是因为这个场景是过渡知识点最自然的场景。后面，我们还会继续深入学习下去，自然会逐步接触更多的变化。

Lambda表达式的语法，对于只熟悉Java面向对象语法的程序员来说确实很难；而无论是“行为参数化”还是“函数式编程思想”的认知，更是需要我们一步一步理解。在这里给出两条建议：

1. 多练，熟能生巧；在日常编码中，凡是能够用Lambda表达式的地方就请有意识地主动使用；
2. 每次使用后，请思考一下你使用的这个场景是不是能够显示出“行为参数化”或“把一个函数传入另一个函数”这样的效果。逐渐地把感官上的体会上升到经验性的认知当中。

title: （二）再次简洁！Lambda之方法引用
date: 2019-11-29
categories:

- JavaSE
- Java面向对象语法
- Lambda表达式系列
  author: 胡亦可

大家已经学习了Lambda表达式的基本概念，那么让我们回到第一章在开篇就给看到的那一段代码，你现在是什么感觉呢?

```
List<String> strLst = Arrays.asList("Hello", "", "Java", "Hi", "Test");
List<String> strings filterStrings = strLst.stream()
                                    .filter(string ->!string.isEmpty())
                                    .collect(Collectors.joining(", "));
filterStrings.forEach(System.out::println);
```

是不是还是感觉有看不懂的代码，😏。

```
System.out::println
```

这又是什么鬼？哈哈哈，来，接着学！

概念

方法引用也是JDK8提出的新特性，而且是与Lambda表达式息息相关的一个特性。
不卖关子，直接给概念：方法引用就是Lambda表达式，它是对Lambda表达式的一种简化语法。

方法引用与Lambda关系

我们先看下面这个示例：

```
@FunctionalInterface
interface MyFuncInter {
    public void func(String str);
}
public class TestMain {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyFuncInter a = s -> System.out.println(s);
        a.func("Hello");
    }
}
```

这是一个我们很熟悉的例子，采用的也是我们已经学习过的标准的Lambda表达式。执行后的效果是：

```
Hello
```

然后我们用方法引用来修改它：

```
@FunctionalInterface
interface MyFuncInter {
    public void func(String str);
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //MyFuncInter a = s -> System.out.println(s);
        MyFuncInter a = System.out::println;
        a.func("Hello");
    }
}
```

我们发现无论是编译还是运行都是没有问题的，运行的结果也是打印Hello。这无非证明了我们一开始给出的概念：方法引用与Lambda表达式是一致的，它只是另一种写法而已。

但是，再细想想，这种简化会不会导致信息的丢失呢?
分析一下：
首先，对于一个Lambda表达式来说，它的代码块中其实是可以执行多条语句的，甚至包括所有的逻辑操作、new对象、调用方法、返回结果等等，只需要写到“{}”当中；

```
@FunctionalInterface
interface MyFuncInter {
    public void func(String str);
}
public class TestMain {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyFuncInter a = s -> {System.out.println(s);
                              System.out.println("OK!")};
        a.func("Hello");
    }
}
```

这是没有任何问题的，结果显示：

```
Hello
OK!
```

那么方法引用可以吗？可以一次引用多个方法吗？试验：

```
@FunctionalInterface
interface MyFuncInter {
    public void func(String str);
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyFuncInter a = {System.out::println};
    }
}
```

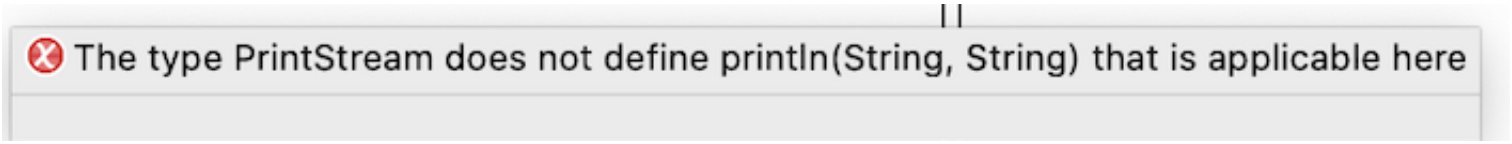
这个时候我们发现，还不用写第二条代码，光打上一对“{}”在方法引用的前后就已经不能通过编译了。也即是说，方法引用只能针对与单独一句代码的，且该代码是调用方法的Lambda表达式。

其次：标准的Lambda表达式，指定了参数s；而方法引用压根连s都没有了，也就是说这个信息被省略了。那假如函数接口定义了两个参数，那么方法引用中的输出语句到底输出谁呢？测试一下：

```
@FunctionalInterface
interface MyFuncInter {
    //函数接口指定两个参数
    public void func(String str1, String str2);
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //Lambda表达式指定打印第一个参数
```

```
        MyFuncInter a =  (s1,s2)  -> System.out.println(s1);
        //编译报错
        MyFuncInter b = System.out::println;
        a.func("Hello","Lambda");
        b.func("Hello","方法引用");
    }
}
```

这个时候你马上看到方法引用的语句报错了，信息如下：
{% img /img/20191129/1.png 500 50%}



这个报错很有意思，它说的不是不知道该传递哪个参数到println()方法当中，而是说println()没有定义可以用在这儿的带两个String参数的重载方法。这说明在方法引用的语法下，编译期压根儿就没有考虑过对两个参数str1和str2要分别干什么，而是一股脑儿的直接丢给方法引用中“::”号后面的println()方法了。
那这是否以为着，如果我们的方法引用不是引用的System.out.println()方法，而是一个可以直接接收两个参数的方法，就可以了呢？马上，再次修改代码：

```
@FunctionalInterface
interface MyFuncInter {
    //函数接口指定两个参数
    public void func(String str1, String str2);
}

class ClassA{
    public void operate(String str1, String str2){
        System.out.println(str1 + "," + str2);
    }
}

public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyFuncInter a = new ClassA()::operate;
        a.("Hello","方法引用");
    }
}
```

果然，这样就编译通过，并且运行也是正确的：

```
Hello,方法引用
```

这说明方法引用要求函数接口与调用方法的参数列表保持一致。

接下来，大家还可以按上面的思路去测测返回类型要求。我在这里就省略了，最后给出核心知识点。

1. 方法引用确实就是一种Lambda表达式的简化缩写，但不针对所有的Lambda表达式；
2. 方法引用只能对只有一条语句块，且该语句块是调用一个已有方法的Lambda表达式进行简写；
3. 方法引用还要求简写对应的函数接口方法的参数列表和返回类型与被调用方法保持一致。

方法引用语法分类

刚才上面的例子，我们都是使用的System.out.println()方法做的例子。其实，这只是一种情况。方法引用的语法一共分为了4种情况：

通过对象调用该对象的普通行为

语法 --- 对象名::实例方法名

```
@FunctionalInterface
public interface MyFuncInter {
    public int func();
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //引用字符串对象“Hello”的length()方法
        MyFuncInter a = "Hello"::length;
        int length = a.func();
        System.out.println(length);
    }
}
```

通过类调用该类的静态行为

语法 --- 类名::静态方法名

```
@FunctionalInterface
```

```
public interface MyFuncInter {
    public long func();
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //引用System类的currentTimeMillis方法
        MyFuncInter a = System::currentTimeMillis;
        long millis = a.func();
        System.out.println(millis);
    }
}
```

通过类调用该类的构造方法

语法 --- 类名::new

```
@FunctionalInterface
public interface MyFuncInter {
    public String func(char[] c);
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //引用String类的带char[]参数的构造
        MyFuncInter a = String::new;
        char[] c = {'J','A','V','A'}
        String str = a.func(c);
        System.out.println(str);
    }
}
```

这种用法有特殊要求：
函数接口方法的返回类型必须是方法引用语句中的类型。

通过类调用该类的普通行为

语法 --- 类名::实例方法名

```
@FunctionalInterface
public interface MyFuncInter {
```

```
    public boolean func(String str1,String str2);
}
public class TestMain {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //引用String类的带char[]参数的构造
        MyFuncInter a = String::equals;
        boolean flag = a.func("Hello","hello");
        System.out.println(flag);
    }
}
```

这种语法更特殊，甚至与我们上一节的总结有矛盾。String的equals方法只能接受一个参数，而函数接口MyFuncInter的func方法要求接收两个参数。这就不符合上面的第3点总结的参数一致的要求。

这是因为equals()方法不是静态方法，所以用类名String是不可能调用到的。所以这种情况下，它会自动把第一个参数当作实例对象，相当于调用的都是"Hello".equals("hello")。

方法引用的学习建议

方法引用使用运算符::连接类(或对象)与方法名称(或new)实现在特定场景下对lambda表达式的简化表示，使用时要注意方法引用的使用场景及各种方法引用的特性。使用方法引用的好处是能够更进一步简化代码编写，使代码更简洁。
不过，方法引用代替Lambda表达式对代码的简化程度，远没有Lambda表达式代替匿名类的简化程度大。甚至有时反而增加了代码的理解难度，且使用场景的局限性不利于增加或修改代码。因此，个人建议没有必要**刻意**使用方法引用.....

---

title: （三）更多的应用场景！Lambda之java.util.funciton包
date: 2019-11-30
categories:

- JavaSE
- Java面向对象语法
- Lambda表达式系列
  author: 胡亦可

在之前的学习当中，我们学会了Lambda表达式的基本语法和基本场景。我们发现，Lambda表达式

其实就是匿名内部类的简化写法。它关注的是把函数接口中的单一抽象方法，最简洁的方式将这个方法的内部实现告知给调用者。整个语法结构其实在本质上可以跟接口、抽象方法的定义一一对应上。

既然如此，那么对我们而言，凡是之前用到单一方法接口的地方都可以使用Lambda表达式了。编程经验丰富的你能不能说出有哪些应用场景呢?

JDK8之前的传统应用场景

首先在这里罗列出JDK8之前可以使用的传统函数接口：

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.nio.file.PathMatcher
- java.lang.reflect.InvocationHandler
- java.beans.PropertyChangeListener
- java.awt.event.ActionListener
- javax.swing.event.ChangeListener

当然，不是上面所有的函数接口每个程序员都熟悉，这还是与你的知识点范围和工作范围有关系。不过，Runnable、Callable、Comparator、FileFilter、InvocationHandler以及ActionListener和ChangeListener相信大家还是经常操作的。希望大家在以后的开发中，遇到这些应用场景尽量使用Lambda表达式。

但是，分析以上这些函数接口你可以发现，它们都紧紧地对应到某一个单一的应用。如果不遇上这种应用，那么也就没有使用Lambda表达式的可能性了。如果仅就如此，Lambda在Java中的使用范围就非常的狭窄，并不能算做JDK8中的一个重要升级，最多是一个小小的优化。

为了提供更广泛的的应用，JDK8新给我们提供了很多新的函数接口（@FunctionalInterface）。

JDK8之中的更广泛应用场景

在JDK8当中，为了让Lambda表达式能够有更广泛的应用场景，它设计了更多的函数接口放在java.util.function包当中，数目高达40多个。这些函数接口不再像之前的Runnable等针对了某个具体的技术应用，更多的体现的是通用性。接下来，我们模拟一个，描述一下它的设计思路。

设计思路

比如：在现实开发中，我们经常会执行到对两个整型进行各种具体的业务操作，最终得到一个整型结果。你可以对他们做加减乘除，也可以比较大小返回最大值或最小值，甚至是更复杂的业务要求。

OK，JDK8就先帮你定义这样的一个函数接口，参数是两个整型，返回值也是整型。下面是模拟代码：

```
@FunctionalInterface
public interface MyFuncInter {
    public int operate(int numA, int numB);
}
```

那么接下来，凡是在开发当中你遇到了这样的问题域，都可以采用如下的Lambda表达式实现了：

```
public static void main(String[] args) {
    int numA = 17, numB = 23;
    MyFuncInter myFunc = (x,y) -> x > y ? x : y;
    System.out.println("两个数中较大的是：" + myFunc.operate(numA, numB));
    myFunc = (x,y) -> x+y;//根据业务，可以实现更复杂的业务操作
    System.out.println("他们之和是：" + myFunc.operate(numA, numB));
}
```

当然，Java的设计人员肯定考虑的更加周详，以便于满足更多的变化。比如：他们自然而然地不会把参数和返回类型写为固定的int，而会选择更为灵活的泛型设计。

```
@FunctionalInterface
public interface MyFuncInter<T> {
    public T judge(T t1, T t2);
}
```

相应地，我们的使用也就更广泛了，这一个函数接口对应了所有的两个同类型参数返回同类型结果的应用场景。

以上是我们模拟的情况，接下来就请看看真正的来自于java.util.function包中的函数接口吧。

function包中的常用函数接口

function包中的函数接口相当的多，下面只列出了可能最常用的几个：
{%img /img/20191130/1.png 500%}

函数接口名	抽象方法声明
Predicate<T>	boolean test(T t)
Consumer<T>	void accept(T t)
Function<T,R>	R apply(T t)
Supplier<T>	T get()
UnaryOperator<T>	R apply(T t)
BinaryOperator<T>	R apply(T t, U u)

Predicate

别名：断言型接口
作用：接受一个输入参数，返回一个布尔值结果。

默认方法：
1、Predicate and(Predicate<? super T> other) 返回一个Predicate对象，该返回对象是当前Predicate对象与传入方法的Predicate对象的短路逻辑与。

2、Predicate isEqual(Object targetRef) 返回一个Predicate对象，该返回对象是根据Object.equals()方法判断当前Predicate对象与传入的目标对象是否相等的结果。

3、Predicate negate() 返回一个Predicate对象，该返回对象是当前Predicate对象的取反结果。

4、Predicate or(Predicate<? super T> other) 返回一个Predicate对象，该返回对象是当前Pedicate对象与传入方法的Predicate对象的短路逻辑或。

Consumer

别名：消费型接口
作用：接受一个输入参数，执行操作并且无返回的操作

默认方法：
Consumer andThen(Consumer<? super T> after) 返回一个Consumer对象，该返回对象的accept（）方法会先执行当前Consumer对象的accpet()方法，再执行传入的Consumer对象的accpet()方法。

Function

别名：函数型接口
作用：接受一个输入参数，返回一个结果

默认方法：
1、 Function andThen(Function<? super R,? extends V> after) 返回一个Function对象，该返回对象是用当前Function对象的结果作为输入，传入到参数Function对象after中，用after的结果做为返回Fuction对象的结果。

2、 Function compose(Function<? super V,? extends T> before) 返回一个Function对象，该返回对象是用传入的Function对象before的结果作为当前Function对象的输入，从而所得到的最终结果。

静态方法：
Function identity() 始终返回输入参数

Supplier

别名：供给型接口
作用：无参数，返回一个结果。

UnaryOperator

作用：接受一个参数为类型T,返回值类型也为T。

静态方法：
UnaryOperator identity() 始终返回输入参数

继承方法：继承了Function的所有方法默认方法

BinaryOperator

作用：接受两个同类型参数的操作，并且返回了同类型结果

静态方法
1、 BinaryOperator maxBy(Comparator<? super T> comparator) 返回BinaryOperator对象，该对象的结果是根据指定的比较器返回两个参数中的较大者。

2、 BinaryOperator minBy(Comparator<? super T> comparator) 返回BinaryOperator对象，该对象的结果是根据指定的比较器返回两个参数中的较小者。

function包的学习建议

function包中的函数接口是为了扩大Lambda表达式的应用场景而专门设计的。对于Java程序员来说，由于长时间习惯于在面向对象的语法和思维中，恐怕对于这种方式很不适应。

在不熟悉或暂时没有时间去熟悉的情况下，建议不用太过于强求，按你熟悉的方式来开发即可。但是function包作为一个JDK8中的新内容，大家还是要有所了解。因为，JDK8后面提供的一些与Lambda相关的API操作中，在底层默默地使用了它们。如果有一天，你需要学习和了解这些API底层源码的时候，请不要懵逼。

使用随缘，了解即可。

---

title: （四）造福一方！Lambda之Stream流
date: 2019-12-1
categories:

- JavaSE
- Java面向对象语法
- Lambda表达式系列
  author: 胡亦可

到目前为止，我们已经学习了Lambda表达式的本质（函数接口）、语法（参数列表 -> {代码块}）；也接触到了Lambda表达式的基本应用场景（替代匿名内部类）、简化语法（方法引用）。但我相信，这个时候你还是没有体会到Lambda表达式的强大之处。这个还真不怪你🤷，因为我们确实使用匿名内部类的时机并不多见。虽然提供了这么多新语法，但是苦于英雄无用武之地，Lambda,你让我们如何对你爱得起来？😭
JDK8中新增的特性是旨在帮助程序员写出更好的代码，除了新语法的提出，还有就是对核心类库的改进也是很关键的一部分。而在对核心类库的改进中，主要包括集合类的 API 和新引入的流(Stream)，它让程序员得以站在更高的抽象层次上对集合进行操作。
集合操作简直就是我们日常开发中最最常见的应用场景了，专门配合Lambda而修改的集合类API和Stream，直接让Lambda从小众走向了开发舞台的中央。

从外部迭代到内部迭代

集合的应用中最常见的一种就是迭代，即把集合中的元素从头到尾遍历一次。
我们先以数组为例（数组虽然不是JCF中的类型，但它也是一种集合结构，而且是最简单最原始的结

构。）

```
int[] nums = {1,3,5,7,9};
for(int i = 0; i < nums.length; i++){
    System.out.println(nums[i]);
}
```

这是最传统的数组迭代（遍历）。然后到了JDK5，为了简化书写操作，Java又提供了for-eac语法。

```
int[] nums = {1,3,5,7,9};
for(int n : nums){
    System.out.println(n)
}
```

那么到了JDK8，我们现在这么写：

```
int[] nums = {1,3,5,7,9};
Arrays.stream(nums).forEach(System.out:println);
```

就问你爽不爽？！！！😂😂😂
在上面三个列子中，第一个和第二个被称之为"外部迭代"；第三个就是"内部迭代"。这里的“外部”和“内部”之分指的是：迭代是在API外部还是内部实现。这就很明显了，外部迭代是让程序员自己完成迭代的循环控制，以及每次迭代要执行什么代码；而内部迭代的循环控制是在API提供的forEach()方法内部完成，我们只需要告诉它每次迭代做什么事情。很明显，在循环控制没有特殊要求的情况下，使用内部迭代可以让我们的代码更加的简洁。

接下来，我们再到DOC文档中看看forEac()方法。它的声明形式如下：

```
void forEach(Consumer<? super T> action)
```

该方法接受一个Consumer类型的参数，它是.....想起来没有.....在上一篇中我们曾经介绍过它。没错，它就是JDK8中在fuction包中新提供的一个函数接口，被称之为“消费型接口”，作用是接受一个输入参数，执行操作并且无返回的操作。（之前我们说直接使用这些function包的函数接口似乎并不方便，其实它们是在底层默默工作，今天我们的主角Stream的强大功能其实很大层度都是建立在它们之上）。
以上分析说明，forEach方法接收的是一个叫Consumer的函数接口，该接口把接收一个参数（这个参数就是集合中的每个元素），我们只需要告诉传入代码块表明如何操作这个参数即可。（想一想，

这是不是行为参数化的表现。）由此，我们可以写出如下代码：

```
int[] nums = {1,3,5,7,9};
Arrays.stream(nums).forEach(n -> {
        n += 1;
        if(n > 5){
            System.out.println(n)
        }
);
```

这告诉我们就算更复杂的操作，我们也可以用内部迭代完成。那么，这个内部迭代的forEach()方法是谁提供的呢？我们可以清楚地从代码中看到它是由一个叫stream()方法提供的，这就涉及到了我们的本文的主人公。

强大的Stream

首先，让我们清楚一个概念，这里的Stream跟I/O流中的InputStream/OutputStream是完全不同的两个东东。它是被放在java.util.stream包当中的一个接口，其作用是提供一种声明式的方式来处理数据。
那么如何理解Stream到底是干什么的呢？Stream相当于提供了一种流水线的方式去操作数据源的数据，这个流水线上的不同工位我们可以让它去完成不同的工作，最终产生一个执行的结果。我们的基本操作方式是：

```
数据源 --> 上流水线 --> 操作1 --> 操作2 --> 操作3.....
```

理解要点：

1. Stream相当于流水线，它的作用是操作而不是存储或管理数据，数据还是存放在数据源的；
2. 上流水线则获取到一个Stream对象，后面每一个操作都是返回一个新的Stream对象（相当于下一个流水），直到最后一个操作。中间操作都是返回的Stream对象，最后操作是返回一个非Stream对象或者甚至没有返回；
3. 为了执行优化，Stream采用“惰性求值”和“及早求值”两种方式执行操作。Stream当中的所有中间操作都不是严格意义的立即执行，而是全部会到最后一个操作中融合起来，这样就可以一次遍历搞定，而非每个操作都做一次遍历。所以中间岗位都是“惰性”--Lazy的，只有最后岗位才是“及早求值”---立即执行的。
4. Stream的操作都是提供的数据映射，不会改变数据源中的数据存储内容。
5. Stream的操作都可采用Lambda表达式的书写方式。

数据源获取Stream的方式

1、上例中，我们看到的是数组获取Stream对象的方式，这里不再赘诉了。就是通过集合框架的工具类Arrays的stream()方法，传入数组即可。

2、 JDK8在集合框架的根接口Collection中增加了一个stream（）方法，通过该方法可以获取集合类的Stram流。既然List、Set是Collection的直接子接口，所以该方式它们均可行。Map则可以通过keySet()方法得到Set类型的键集合，也可以通过values()方法得到Collection类型的值集合，也都可以使用stream()方法。另外Map还可以通过entrySet()方法得到形如“键=值”的Set集合，再通过stream()方法就能得到既有键又有值的Stream流。
演示：

```
List<String> lst = new ArrayList<>();
Set<String> set = new HashSet<>();
Map<Integer,String> map = new HashMap<>();
......//省略放入元素的代码

Stream streamLst = lst.stream();
Stream streamSet = set.stream();
Stream streamKey = map.keySet().stream();
Stream streamValue = map.values().stream();
Stream streamEntrySet = map.entrySet().stream();
```

3、 还可以直接用Stream的of()方法，指定元素获取流。
演示：

```
Stream stream = Stream.of("Hello","World","Java","Lambda","");
```

Stream中常用的中间操作（惰性）

以下这些Stream的方法都是惰性的，也就是这些方法的返回值仍然是Stream类型的新对象，只能充当中间操作。为了让我们的代码可以运行看到结果，我们都会追加一个forEach循环打印在最后，这样才能运行起来看到结果。原因请参见上面蓝色字体的要点3。

map() --- 映射元素

map()方法是对每个元素进行数据值的操作。
演示：

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
```

```
    .forEach(System.out::println);
```

这里首先创建了有5个字符串元素的流，然后利用map()方法对每个元素进行全大写转换形成新的Stream流，然后对流中的每个元素进行打印。最后得到的执行结果是：

```
HELLO
WORLD
JAVA
LAMBDA
STREAM
```

如果我们要执行更复杂的操作，比如：首先把元素转为全大写，然后取每个元素的最后1个字符。那么，理论上我们可以这么写：

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> {e = e.toUpperCase();return e.charAt(e.length() - 1);})
    .forEach(System.out::println);
```

能写出这种代码，首先恭喜你，说明你对Lambda表达式的基本语法很熟悉。但是这种写法虽然可行，却反应了你对流水线的理解不透彻。更好的写法如下：

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .forEach(System.out::println);
```

你看这样写，代码逻辑是不是更清晰了？！你只需要在换为大写之后，在流水线上再加一道元素映射工序获取首字母就可以了。😊
执行结果如下：

```
O
D
A
A
M
```

flatMap() --- 扁平化的map()方法

map()方法将Stream中的元素对象进行一对一的映射，这是很常规的。但在一些特殊情况下，它可能就无能为力了。比如，当我们想每一个元素转换成多个对象，所有元素的所有转换后对象放在一起处理呢？这种情况下，map()方法是做不到的了，因为它始终保持原有流水线的元素个数。那么flatMap()则刚好弥补了这个功能。
flatMap()方法的作用是把stream的每个元素转换到其他对象的Stream。因此，每个对象将被转换为零个、一个或多个基于Stream的不同对象。这些stream的内容将被统一放置到flatMap()方法的返回的新的Stream中。

假如我们要从这5个字符串中去除重复的字符："hello","world","java","lambda","stream"。

用map()

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.split(""))
    .distinct() //去重复
    .forEach(System.out::println);
```

执行结果

```
[Ljava.lang.String;@2a84aee7
[Ljava.lang.String;@a09ee92
[Ljava.lang.String;@30f39991
[Ljava.lang.String;@452b3a41
[Ljava.lang.String;@4a574795
```

这个结果根本不是我们想要的，居然是5个字符串数组。这是因为map()方法只是对与每个元素执行一次split()，每个元素被从一个String拆成了一个String[]，执行后的Stream流当中就装的是5个字符串数组。

其实我们想要的是把这5个字符串的所有字符拆开，形成一个新的流，装的全是单个字符（虽然还是String类型），然后去重。那么，很明显这个新的Stream的元素个数不可能还是5个了，所以map()方法做不到。这个时候就是flatMap()方法出手的时候了。

```
Stream.of("hello","world","java","lambda","stream")
                .flatMap(e -> Stream.of(e.split("")))
            .distinct()
            .forEach(System.out::print);
//为了显示直观，没有打印换行
```

这里我们看到flatMap把每个元素e拆分了一个后，生成了一个Stream对象，然后把这5个Stream对

象又组合成一个完整的返回出去。所以，我们在做distinct()方法或forEach()方法的时候，都不是操作的元素个数为5的Stream，而是flatMap合并后的新Stream。

所以，flatMap()的作用就是把多个Stream合并成一个新的Stream，所以它内部的函数行为应该是一个Stream对象。

filter() --- 条件过滤

filter()是书写过滤条件，满足条件的元素被保留在流水线往下走，不满足的被排除掉。
演示：

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .forEach(System.out::println);
```

这样在流水线中就过滤掉了‘O’字符，只剩下了后面四个：

```
D
A
A
M
```

distinct() --- 去重

“去重”，顾名思义，当然就是去掉重复元素。
演示：

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .forEach(System.out::println);
```

这样在流水线中就过滤掉了‘O’字符，只剩下了后面四个：

```
D
```

```
A
M
```

sorted() --- 排序

sorted()方法用于对元素进行排序。该功能由两个重载功能组成，一个无参，即按元素自带的内部比较器（元素类型实现的Comparable接口）的方式排序；另一个则可以指定一个外部比较器（Comparator接口），结果就会按外部比较器规则进行排序。
演示1:

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted()
    .forEach(System.out::println);
```

执行结果为:

```
A
D
M
```

演示2:

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .forEach(System.out::println);
```

执行结果为:

```
M
D
A
```

limit() --- 限制元素个数

limit()方法用于限制流中元素的个数，即把多出来的元素从流中去掉。
演示:

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .limit(2)
    .forEach(System.out::println);
```

执行结果为:

```
M
D
```

skip()方法

skip()方法用于跳过流中的前几个元素，作用和limit()相似。
演示:
演示:

```
Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .skip(2)
    .forEach(System.out::println);
```

执行结果:

```
A
```

Stream中常用的最后操作（及早）

以下这些方法只能作为流水线的最后工序，方法执行之后返回的不是Stream对象，或者没有返回值（void）。

forEach() --- 遍历

forEach()方法我们在前面用得很多了，只不过当时我们只执行了打印操作。 通过给forEach()方法，传入带"{}"的Lambda表达式，我们可以让它做更多更复杂的功能。
⚠️注意：这个方法是没有返回类型的。

reduce() --- 万能的带返回方法

reduce()方法是Stream流中的底层方法，它的作用是根据一定的规则将Stream中的元素进行计算后返回一个唯一的值。

reduce()方法有三个重载，分别带一个，两个和三个参数。
带一个参数的：

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

简单说明：
参数：BinaryOperator是二元操作的函数接口，接收两个同类型的参数，其中第一个是返回的结果，第二个是流中的元素。
返回类型：Optional，这是JDK8中专门为了解决Object可能发生空指针异常所设计的一个类，我们会在下一章节讲到它。大家现在可以简单理解为不会报空指针的Object。

带两个参数的：

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

简单说明：
参数1: identity是给在操作开始前给返回变量赋的初始值；
参数2: BinaryOperator与上面的reduce中的保持一致；
返回类型：只能是Stream流中的元素类型。

带三个参数的：

```
<U> U reduce(U identity,
```

```
                BiFunction<U, ? super T, U> accumulator,
                BinaryOperator<U> combiner)
```

简单说明：
参数1:同样是返回值的初始化，值其类型是泛型U，与Reduce方法返回的类型一致；注意此时Stream中元素的类型是T，与U可以不一样也可以一样，这样的话操作空间就大了；
参数2: 其类型是BiFunction，输入是U与T两个类型的数据，而返回的是U类型；也就是说返回的类型与输入的第一个参数类型是一样的，而输入的第二个参数类型与Stream中元素类型是一样的。
参数3: 其类型是BinaryOperator，支持的是对U类型的对象进行操作；

⚠注意：reduce()方法可以完成所有的带返回值的及早操作，后面的count(),max(),min()都是在它的基础上进一步完成封装。只是这个方法使用起来虽然灵活，但过于不方便，所以在实际当中，我们往往时候后面的封装方法，除非遇到极为特殊的情况。

count()

得到Stream流中元素的个数。
演示:

```
long num = Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < '0')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .limit(2)
    .count();
System.out.println(num)
```

执行结果：

```
2
```

⚠注意：该方法返回的是long类型。

max()和min()

这两个方法不用多说，分别是得到最大元素和最小元素。

演示:

```
Optional<Character> c = Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .limit(2)
    .max((t1,t2) -> t1 - t2);

System.out.println(c)
```

执行结果：

```
Optional[M]
```

⚠️注意：这两个方法必须要指定外部比较器；；另外返回的同样是Optioanl对象。

Collect()

Collect()方法利用Collectors 类实现了很多归约操作，例如将流转换成集合和聚合元素。

1、 比如我们要将Stream流中的元素要保存在一个新的集合中；
演示：

```
List<Character> lst = Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .limit(2)
    .collect(Collectors.toList());
```

⚠️注意：此时只能用List接口来接，因为collect()方法并不一定返回我们熟悉的List子类，它会自己根据情况来进行判断选择。如果我们编码人员一定要指定，请使用如下代码：

```
ArrayList<Character> lst = Stream.of("hello","world","java","lambda","stream")
    .map(e -> e.toUpperCase())
    .map(e -> e.charAt(e.length() - 1))
    .filter(e -> e < 'O')
```

```
    .distinct()
    .sorted((t1,t2) -> t2 - t1)
    .limit(2)
    .collect(Collectors..toCollection(ArrayList<Character>::new));
```

2、 比如我们要将Stream流中的元素拼接成一个字符串；

```
String str =  Stream.of("hello","world","java","lambda","stream")
        .map(e -> e.toUpperCase())
        .collect(Collectors.joining(","));
System.out.println(str);
```

执行结果：

```
HELLO,WORLD,JAVA,LAMBDA,STREAM
```

⚠️注意：此时Stream流中的元素只能是String字符串。

统计SummaryStatistics

上面的max()方法,min()方法，使用起来并不是很方便，而且除了最大最小常常还有求和，取平均值等需求。所以，SummaryStatistics（概要统计）被设计出来，它一共分为三种：IntSummaryStatistics，LongSummaryStatistics，DoubleSummaryStatistics。区别当然就是对应的分别是int、long和double类型的结果。

其使用方式为：

```
IntSummaryStatistics iss = Stream.of(106,13,42,56,21).mapToInt(x -> x).summaryStat
istics();
System.out.println(iss.getCount());//元素总个数
System.out.println(iss.getSum());//元素累加和
System.out.println(iss.getMax());//最大元素
System.out.println(iss.getMin());//最小元素
System.out.println(iss.getAverage());//平均值
```

后话与建议

OK，本文向大家展示了Stream流的基本原理和常用用法。当然，大家可以感受到Stream作为JDK8

中的重点新内容，其范围是庞大的，细节也是繁琐的，甚至还有不少内容在本文当中没有涉及。
作为初次接触的同学，我们认为就以本文内容已经算是非常详尽和丰富的了，掌握起来需要花费一定的时间和精力。因此，我们建议大家先使用目前学到的内容，至于更多的内容在以后的章节中再一一为各位呈现，（比如：并行流和串行流）。
希望大家努力💪！

---

title: （五）又一利器！Lambda之Optional
date: 2019-12-2
categories:

- JavaSE
- Java面向对象语法
- Lambda表达式系列
  author: 胡亦可

Stream流的出现，大大提高了Lambda表达式的应用场景，提高了程序员的接受度和代码的简洁度。而流式数据处理方式的出现，也同样立马深入人心，它采用“分而治之”的思想，从而在处理较大数据的，即整理了编程人员的思路，又极大的提高代码的性能。

利用Lambda表达式和流处理思想，JDK8又为我们提供了一个叫Optinal的类，它是专门针对解决我们另一个常见且烦人的问题 --- 空指针异常。

恼人的NullPointerException

空指针异常（NullPointerException）是每个Java程序员都绕不过去的一个话题。虽然Java语言从设计之初就力图让程序员脱离指针的苦海，但是指针确实是实际存在的。Java设计者也只能是让指针在Java语言中变得更加简单易用，而不能完全将其剔除，所以才有了我们日常所见到的关键字null。

空指针异常其实并不是很难理解的异常，也不是不好处理的异常，通常一个简单的if判断就可以提前规避掉它。麻烦之处在于，很多时候程序员根本没有意识到它的存在，又或者它的存在实在是太频繁了。特别是当我们处理比较复杂的业务逻辑的时候，精力的分散导致程序员根本没有意识到空指针的存在。比如：

```
public String getStudentCityCode(StudentBeanr stu) {
    String cityCode = student.getAddress().getCity().getCityCode();
    return cityCode;
```

```
}
```

为了获取学生家庭所在地的邮政编号，我们通过学生对象获取家庭地址对象，通过家庭地址对象获取城市对象，再通过城市对象获取邮政编号。在这一连串的操作中，任何一个对象为空，都会导致我们的程序报异常，无法正常执行下去。为了解决这个问题，我们只能修改：

```
public String getStudentCityCode(StudentBean stu) {
    String result = "未找到学生的城市邮编";
    if (stu == null) {
        return result;
    }

    AddressBean address = stu.getAddress();
    if (address == null) {
        return result;
    }

    City city = stu.getCity();
    if (city == null) {
        return result;
    }

    String code = city.getCityCode();
    if(code == null){
        return result;
    }else{
        return code;
    }
}
```

好好一段代码变成了这个鬼样子，你说气人不气人?
如果我们使用Optianl呢?

```
public static String getCityCode(StudentBean stu) {
    return Optional.ofNullable(stu)
           .map(e -> e.getAddress())
           .map(e -> e.getCity())
           .map(e -> e.getCityCode())
           .orElse("未找到学生的城市邮编");
}
```

高下之分，一目了然了吧😊。

Optional的引入

Optinal是一个java.util包的类，它是一个容器，可以包含空值或非空值。

属性

Optional包含两个属性：一个静态属性EMPTY，用来存放一个值为null的Optional对象；另一个是非静态属性value，用来存放非null对象。

```
private static final Optional<?> EMPTY = new Optional<>();
private final T value;
```

构造方法

Optional构造也是两个，不过两个都是private的。这说明Optional不支持new出它的对象。

```
private Optional() {
    this.value = null;
}
//带参构造不允许传null
private Optional(T value) {
    this.value = Objects.requireNonNull(value);
}
```

静态方法

由于Optional构造私有，所以只有通过静态方法产生对象。共三个：

```
Optional<T> empty()
```

生成一个空的Optional对象，其value属性值为null。

```
Optional<T> of(T value)
```

调用Optional(T)构造方法，其value属性值不允许为null。

⚠️注意：如果of()方法传入了null，那么会报空指针异常。

```
Optional<T> ofNullable(T value)
```

与Optional of(T value)的差别是其传入的value允许为null

常用实例方法

Optional拥有以下常见方法：

方法	参数类型	返回类型	说明
get	无	T	返回value的值。如果value为null，则会抛出NoSuchElementException异常
isPresent	无	boolean	value非null返回true，否则false
ifPresent	Consumer<? super T>	void	如果Optional实例有值则为其调用consumer函数接口，否则不做处理
filter	Predicate<? super T>	Optional	如果值存在并且，满足参数Predicate函数接口提供的条件，就返回包括该值的Optional对象；否则返回一个空的Optional对象
map	Function<? super T, ? extends U>	Optional	如果值存在，就对该值执行参数Function函数接口所提供的代码块，返回Optional对象
flatMap	Function<? super T, Optional>	Optional	如果值存在，就对该值执行参数Function函数接口所提供的代码块，返回非null Optional对象
orElse	T	T	如果有值则将其返回，否则返回一个默认值（即参数T）
orElseGet	Supplier<? extends T>	T	如果有值则将其返回，否则返回一个由参数Supplier接口函数执行后指定的生成的值
orElseThrow	Supplier<? extends X>		如果有值则将其返回，否则抛出一个由参数Supplier接口函数执行后指定的生成的异常

以上十个方法，分为两大类：

1. filter()、map()、flatMap()三个方法均返回的是Optional对象。那么根据上一章的知识，我们知道这三个方法在Optional中充当流的中间操作；
2. 剩余的方法都是返回到的非Optional值，所以都只能充当流水线的最终操作。

集中示例：

```
public class OptionalDemo {
  public static void main(String[] args) {
    //创建Optional实例，也可以通过方法返回值得到。
    Optional<String> name = Optional.of("Sanaulla");

    //创建没有值的Optional实例，例如值为'null'
    Optional empty = Optional.ofNullable(null);

    //isPresent方法用来检查Optional实例是否有值。
    if (name.isPresent()) {
      //调用get()返回Optional值。
      System.out.println(name.get());
    }

    try {
      //在Optional实例上调用get()抛出NoSuchElementException。
      System.out.println(empty.get());
    } catch (NoSuchElementException ex) {
      System.out.println(ex.getMessage());
    }

    //ifPresent方法接受lambda表达式参数。
    //如果Optional值不为空，lambda表达式会处理并在其上执行操作。
    name.ifPresent((value) -> {
      System.out.println("The length of the value is: " + value.length());
    });

    //如果有值orElse方法会返回Optional实例，否则返回传入的错误信息。
    System.out.println(empty.orElse("There is no value present!"));
    System.out.println(name.orElse("There is some value!"));

    //orElseGet与orElse类似，区别在于传入的默认值。
    //orElseGet接受lambda表达式生成默认值。
    System.out.println(empty.orElseGet(() -> "Default Value"));
    System.out.println(name.orElseGet(() -> "Default Value"));

    try {
```

```
      //orElseThrow与orElse方法类似，区别在于返回值。
      //orElseThrow抛出由传入的lambda表达式/方法生成异常。
      empty.orElseThrow(ValueAbsentException::new);
    } catch (Throwable ex) {
      System.out.println(ex.getMessage());
    }

    //map方法通过传入的lambda表达式修改Optonal实例默认值。
    //lambda表达式返回值会包装为Optional实例。
    Optional<String> upperName = name.map((value) -> value.toUpperCase());
    System.out.println(upperName.orElse("No value found"));

    //flatMap与map（Funtion）非常相似，区别在于lambda表达式的返回值。
    //map方法的lambda表达式返回值可以是任何类型，但是返回值会包装成Optional实例。
    //但是flatMap方法的lambda返回值总是Optional类型。
    upperName = name.flatMap((value) -> Optional.of(value.toUpperCase()));
    System.out.println(upperName.orElse("No value found"));

    //filter方法检查Optiona值是否满足给定条件。
    //如果满足返回Optional实例值，否则返回空Optional。
    Optional<String> longName = name.filter((value) -> value.length() > 6);
    System.out.println(longName.orElse("The name is less than 6 characters"));

    //另一个示例，Optional值不满足给定条件。
    Optional<String> anotherName = Optional.of("Sana");
    Optional<String> shortName = anotherName.filter((value) -> value.length() > 6)
;
    System.out.println(shortName.orElse("The name is less than 6 characters"));
  }
}
```

执行结果：

```
Sanaulla
No value present
The length of the value is: 8
There is no value present!
Sanaulla
Default Value
Sanaulla
No value present in the Optional instance
SANAULLA
SANAULLA
```

```
Sanaulla
The name is less than 6 characters
```