

Java编程补充

浙江大学城市学院

彭彬

pengb@zucc.edu.cn



Java Annotation

用于标注:类, 方法, 变量的一种语法结构, 从Java5.0开始引入, 比如最常见的@Override。官方对Annotation的说明如下

*An annotation is a form of **metadata**, that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. **Annotations have no direct effect on the operation of the code they annotate.***

注意: Annotation是不会给代码的执行序列或者功能带来直接影响的, 也就是不要设想使用Annotation来进行业务逻辑控制或者实现逻辑功能, Annotation就是用来做辅助系统提供的。主要的作用包括:

- 为编译器提供信息 (Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.)
- 编译时提供动态处理 (Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.)
- 运行时提供信息 (Runtime processing — Some annotations are available to be examined at runtime.)

Java Annotation~语法和初步理解

注解的语法为：

@Token(key = value)

如果只有一个默认属性，属性key可以省略

@SuppressWarnings(value = "unchecked")可以写为@SuppressWarnings(" unchecked")

没有属性可以不写：比如 @Override

我们用Java语言内置的三个注解来感受一下Annotation的用途：

- @Deprecated：可以用来描述一个类、方法或者字段，表示java不赞成使用这些被描述的对象，如果我们使用了这些类、方法或者字段，**编译器**会给我们警告。
- @Override注解是一个**编译时注解**，它主要用在一个子类的方法中，当被注解的子类的方法在父类中找不到与之匹配的方法时，编译器会报错。
- @SuppressWarnings注解的作用是使**编译器忽略掉编译器警告**。比如，如果我们有一个方法调用了一个@Deprecated方法，或者做了一个不安全的类型转换，此时编译器会生成一个警告。如果我们不想看到这些警告，我们就可以使用@SuppressWarnings注解忽略掉这些警告。

Java Annotation~再理解

我们之前学习Spring使用了大量的注解，比如：

```
@RestController
public class QuestionController {

    private final QuestionRepository repository;
    QuestionController(QuestionRepository repository) { this.repository = repository; }

    @GetMapping("/question/json/{id}")
    Question viewJson(@PathVariable Integer id) {
        QuestionEntity entity = repository.getOne(id);
        Question ret = new Question();
        BeanUtils.copyProperties(entity, ret);
        return ret;
    }
}
```

比如通过@GetMapping注解，使得viewJson方法和GET请求/question/json/xx这样的请求进行了关联，该注解不会影响请求到来时的业务处理过程，但是该注解起到了配置的作用。

Java Annotation~如何自定义注解

我们可以通过Java提供的几个“元注解”来自定义注解。“元注解”就是Java自带的用于定义注解的注解。

@Retention

@Target

@Inherited

@Documented

@interface

比如我们打开之前Spring 的@GetMapping注解的源代码，其定义为：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@GetMapping(method = RequestMethod.GET)
public @interface GetMapping {
```

Java Annotation~如何自定义注解

几个“元注解”的含义和基本用法为：


@interface: 是java中用于声明注解类的关键字. 使用该注解表示将自动继承java.lang.annotation.Annotation类, 该过程由编译器完成.

@Retention: 该注解用于定义注解的作用范围，只能是以下三个值：

1. RetentionPolicy.SOURCE: 注解只存在于源码中，不会存在于.class文件中，在编译时会被忽略掉
2. RetentionPolicy.CLASS: 注解只存在于.class文件中，在编译期有效，但是在运行期会被忽略掉，这也是默认范围
3. RetentionPolicy.RUNTIME: 在运行期有效，JVM在运行期可以通过反射获得注解信息

@Documented: 作用是告诉JavaDoc工具，当前注解本身也要显示在Java Doc中。

@Inherited: 注解表示当前注解会被注解类的子类继承。



Java Annotation~如何自定义注解

@Target: 用于指定注解作用于java的哪些元素，未标注则表示可修饰所有. 有以下这些元素类型可供选择:

`ElementType.ANNOTATION_TYPE` can be applied to an annotation type.

`ElementType.CONSTRUCTOR` can be applied to a constructor.

`ElementType.FIELD` can be applied to a field or property.

`ElementType.LOCAL_VARIABLE` can be applied to a local variable.

`ElementType.METHOD` can be applied to a method-level annotation.

`ElementType.PACKAGE` can be applied to a package declaration.

`ElementType.PARAMETER` can be applied to the parameters of a method.

`ElementType.TYPE` can be applied to any element of a class.

Java Annotation~再来看如何定义一个Annotation

比如Spring中的@GetMapping注解如下:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {

    /** Alias for {@link RequestMapping#name}. ...*/
    @AliasFor(annotation = RequestMapping.class)
    String name() default "";

    /** Alias for {@link RequestMapping#value}. ...*/
    @AliasFor(annotation = RequestMapping.class)
    String[] value() default {};

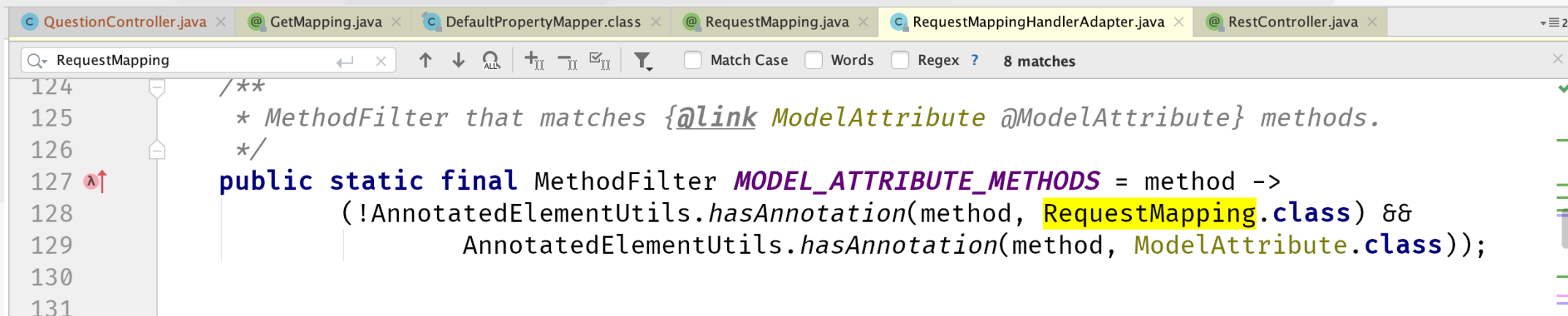
    /** Alias for {@link RequestMapping#path}. ...*/
    @AliasFor(annotation = RequestMapping.class)
    String[] path() default {};
```

其含义为:

- 1) 用于注解“方法”
- 2) 该注解在“运行时”起作用
- 3) 该注解会在生成JavaDoc时也被放入生成的文档中
- 4) 该注解被RequestMapping注解标注, 参数为
(method=RequestMethod.GET)
- 5) 该注解具有name, value等属性值

Java Annotation~再来看自定义Annotation的作用

比如在Spring MVC的源代码RequestMappingHandlerAdapter类中有如下代码



```
124 /**
125  * MethodFilter that matches {@link ModelAttribute @ModelAttribute} methods.
126  */
127 public static final MethodFilter MODEL_ATTRIBUTE_METHODS = method ->
128     (!AnnotatedElementUtils.hasAnnotation(method, RequestMapping.class) &&
129      AnnotatedElementUtils.hasAnnotation(method, ModelAttribute.class));
130
131
```

从上面代码中我们可以看出，此成员变量根据某个方法是否有两个标注而具有不同的值，这个就是典型的在运行时根据Annotation的值来进行动态配置。

Java Annotation~定义一个Annotation来体会

比如我们准备为代码日志扩展编写人信息，那么我们可以定义如下标注

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CodeMemo {
    String name() default "unkown";
    String date() default "";
    String intro() default "";
}
```

```
@CodeMemo(name = "pb", date = "2020/04/22", intro = "Stream使用演示")
public void demo(){

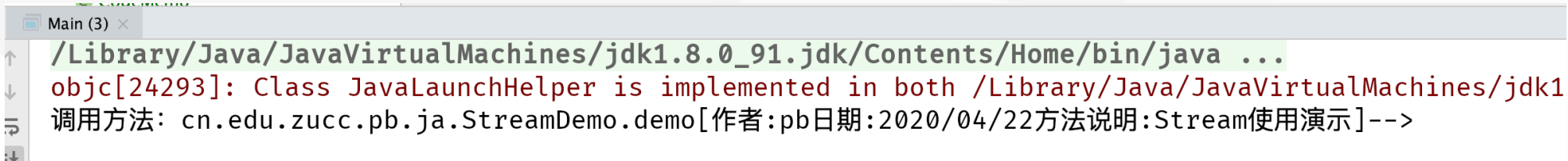
}
```

Java Annotation~定义一个Annotation来体会

可以使用反射获取对应的Annotation

```
private static void logCodeMemo(Class clazz, String methodName){
    Method method = null;
    try {
        method = clazz.getMethod(methodName);
        CodeMemo codeMemo = method.getAnnotation(CodeMemo.class);
        if(codeMemo!=null){
            System.out.println("调用方法: "
                + clazz.getName() + "." + method.getName()
                + "[作者:" + codeMemo.name()
                + "日期:" + codeMemo.date()
                + "方法说明:" + codeMemo.intro()
                + "]-->");
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    }
}
```

```
logCodeMemo(StreamDemo.class, methodName: "demo");
streamDemo.demo();
```



The screenshot shows a terminal window titled 'Main (3)' with the following output:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java ...
objc[24293]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1
调用方法: cn.edu.zucc.pb.ja.StreamDemo.demo[作者:pb日期:2020/04/22方法说明:Stream使用演示]-->
```

Java Stream（流）

从Java8开始，Java提供了Stream对象，可以用一种更直观的方式来操作集合运算，并且可以进行更高阶的抽象。比如下面的例子：

```
+-----+      +-----+  +-----+  +---+  +-----+
| stream of elements +----> |filter+> |sorted+> |map+> |collect|
+-----+      +-----+  +-----+  +---+  +-----+
```

以上的流程转换为 Java 代码为：

```
List<Integer> transactionsIds =
widgets.stream()
    .filter(b -> b.getColor() == RED)
    .sorted((x,y) -> x.getWeight() - y.getWeight())
    .mapToInt(Widget::getWeight)
    .sum();
```



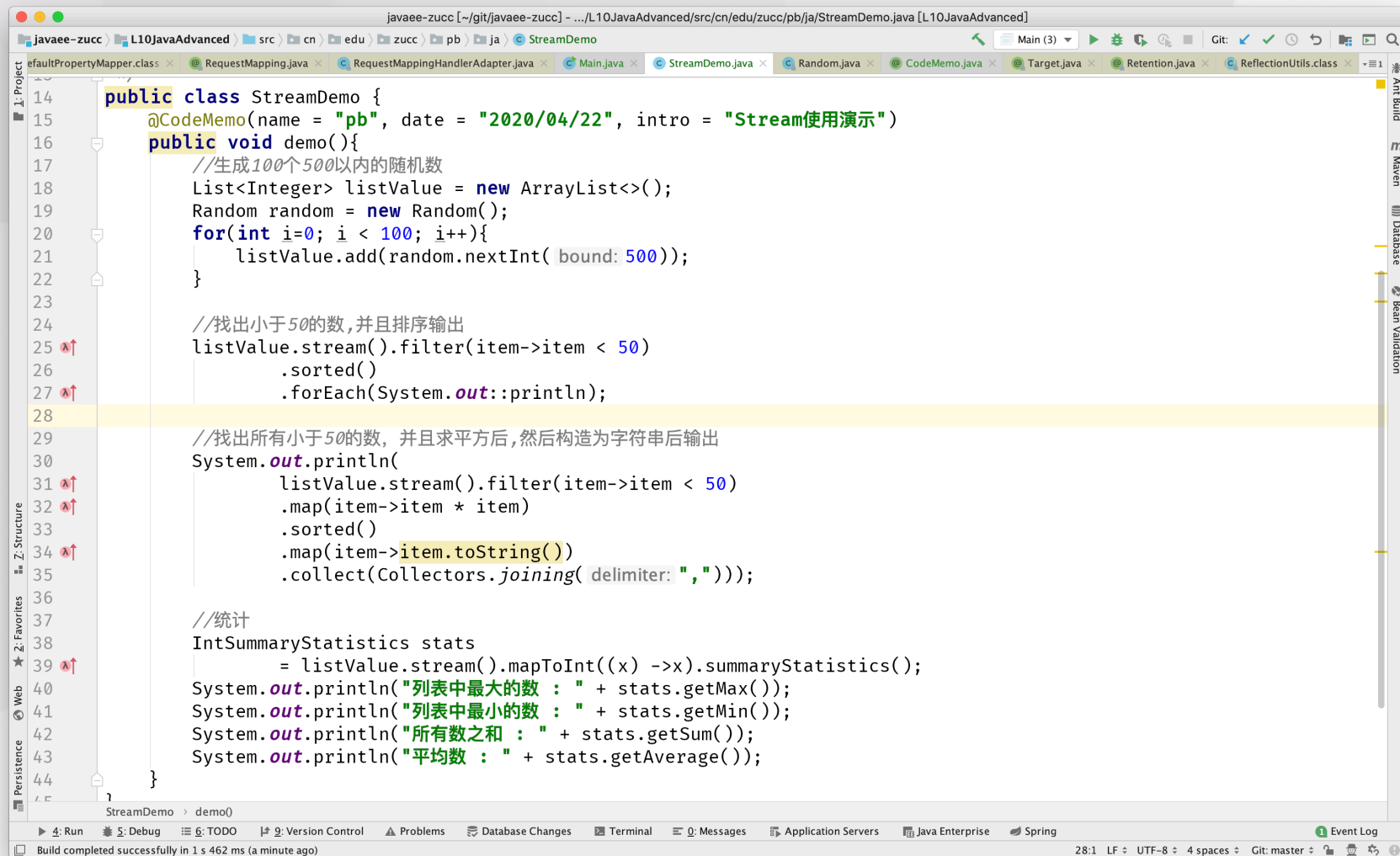
Java Stream

Stream是一个来自数据源的元素队列，并支持聚合操作


- 元素：是特定类型的对象，形成一个队列。Java中的Stream并不会存储元素，而是按需计算。
- 数据源：流的来源。可以是集合，数组，I/O channel，产生器generator 等。
- 聚合操作：类似SQL语句一样的操作，比如filter, map, sum等。
- Pipelining：中间操作都会返回流对象本身。这样多个操作可以串联成一个管道，如同流式风格（fluent style）。
- 内部迭代：以前对集合遍历都是通过Iterator或者For-Each的方式，显式的在集合外部进行迭代，这叫做外部迭代。Stream提供了内部迭代的方式。

Java Stream

产生集合最常用的方式就是从Collection获取，Java 8的Collection类有一个stream方法，返回集合元素组成的流。我们对着下面的例子来进行说明：



```
14 public class StreamDemo {
15     @CodeMemo(name = "pb", date = "2020/04/22", intro = "Stream使用演示")
16     public void demo(){
17         //生成100个500以内的随机数
18         List<Integer> listValue = new ArrayList<>();
19         Random random = new Random();
20         for(int i=0; i < 100; i++){
21             listValue.add(random.nextInt( bound: 500));
22         }
23
24         //找出小于50的数,并且排序输出
25         listValue.stream().filter(item->item < 50)
26             .sorted()
27             .forEach(System.out::println);
28
29         //找出所有小于50的数, 并且求平方后,然后构造为字符串后输出
30         System.out.println(
31             listValue.stream().filter(item->item < 50)
32                 .map(item->item * item)
33                 .sorted()
34                 .map(item->item.toString())
35                 .collect(Collectors.joining( delimiter: ", ")));
36
37         //统计
38         IntSummaryStatistics stats
39             = listValue.stream().mapToInt((x) ->x).summaryStatistics();
40         System.out.println("列表中最大的数 : " + stats.getMax());
41         System.out.println("列表中最小的数 : " + stats.getMin());
42         System.out.println("所有数之和 : " + stats.getSum());
43         System.out.println("平均数 : " + stats.getAverage());
44     }
}
```



Java Stream~中间操作

Java Stream常用操作

中间操作符

对于数据流来说，中间操作符在执行制定处理程序后，数据流依然可以传递给下一级的操作符。

- `map`: 转换操作符，把比如A→B。
- `flatMap` : 展开操作，比如把 `int[] {2, 3, 4}` 展开为2, 3, 4。
- `limit`: 个数限定操作，比如数据流只取前三个，`stream.limit(3)`。
- `skip` : 略去若干元素操作，略去前三个，`stream.skip(3)`。
- `distinct` : 去重操作，对重复元素去重，底层使用了`equals`方法。
- `filter` : 过滤操作，把不符合条件的元素剔除。
- `peek`: 对每个元素执行操作并返回一个新的 `Stream`, 用于修改流中的元素
- `sorted`: 排序操作，对元素排序，前提是实现`Comparable`接口，也可以自定义比较器。

Java Stream~终止操作

Java Stream常用操作

终止操作符

终止操作符就是用来对数据进行收集或者消费的，数据到了终止操作这里就不会向下流动了，终止操作符只能使用一次。

- `collect` 收集操作，将所有数据收集起来，这个操作非常重要，官方的提供的Collectors 提供了非常多收集器，可以说Stream的核心在于Collectors。
- `count` 统计操作，统计最终的数据个数。
- `findFirst`、`findAny` 查找操作，查找第一个、查找任何一个 返回的类型为Optional。
- `noneMatch`、`allMatch`、`anyMatch` 匹配操作，数据流中是否存在符合条件的元素 返回值为bool 值。
- `min`、`max` 最值操作，需要自定义比较器，返回数据流中最大最小的值。
- `reduce` 归纳操作，将整个数据流的值规约为一个值，`count`、`min`、`max`底层就是使用`reduce`。
// 字符串连接, concat = "ABCD"
String concat = Stream.of("A", "B", "C", "D").reduce("", String::concat);
- `forEach`: 对每个元素进行遍历操作。
- `toArray` 数组操作，将数据流的元素转换成数组。

Lambda表达式

Lambda 表达式：也称为闭包，一种函数的定义表达式，并且可以作为参数进行传递，比如上面Stream中用到的

```
listValue.stream().filter(item->item < 50)
    .sorted()
    .forEach(item->System.out.print(item.toString() + ", "));
```

下面这个就是典型的Lambda表达式，相当于是一个匿名方法，但是其非常方便的看做表达式，可以作为参数传递

item->System.out.print(item.toString() + ", ")

语法

lambda 表达式的语法格式如下：

```
(parameters) -> expression
或
(parameters) ->{ statements; }
```

以下是lambda表达式的重要特征：

- **可选类型声明**：不需要声明参数类型，编译器可以统一识别参数值。
- **可选的参数圆括号**：一个参数无需定义圆括号，但多个参数需要定义圆括号。
- **可选的大括号**：如果主体包含了一个语句，就不需要使用大括号。
- **可选的返回关键字**：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

Lambda表达式~几个简单的例子

下面是几个简单的例子

第二个Lambda表达式有一个Apple类型的参数并返回一个boolean(苹果的重量是否超过150克)

```
(String s) -> s.length()  
-> (Apple a) -> a.getWeight() > 150  
(int x, int y) -> {  
    System.out.println("Result:");  
    System.out.println(x+y);  
}
```

```
() -> 42
```

```
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

第五个Lambda表达式具有两个Apple类型的参数, 返回一个int: 比较两个Apple的重量

第一个Lambda表达式具有一个String类型的参数并返回一个int。Lambda没有return语句, 因为已经隐含了return

第三个Lambda表达式具有两个int类型的参数而没有返回值(void返回)。注意Lambda表达式可以包含多行语句, 这里是两行

第四个Lambda表达式没有参数, 返回一个int

Lambda表达式~我们哪里可以使用Lambda表达式

我们可以在任何“函数式接口上使用Lambda表达式”，比如我们来看看之前使用的Stream中可以使用Lambda表达式地方的定义
我们之前使用了Stream的filter方法，传递了一个Lambda表达式：

```
listValue.stream().filter(item->item < 50)  
    .sorted()  
    .forEach(item->System.out.print(item.toString() + ", "));
```

```
*/  
Stream<T> filter(Predicate<? super T> predicate);
```

```
@FunctionalInterface  
public interface Predicate<T> {
```

```
/** Evaluates this predicate on the given argument. ...*/  
boolean test(T t);
```

Lambda表达式~我们哪里可以使用Lambda表达式

再来一个，我们之前使用了Stream的forEach方法，传递了一个Lambda表达式：

```
listValue.stream().filter(item->item < 50)
    .sorted()
    .forEach(item->System.out.print(item.toString() + ", "));
```

↓

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

↓

```
@FunctionalInterface
public interface Consumer<T> {
```

↓

```
/** Performs this operation on the given argument. ...*/
void accept(T t);
```

Lambda表达式~函数式接口

我们先看一下刚才的例子中的接口定义，比如forEach方法中需要的Consumer<T>接口：

```
@FunctionalInterface
public interface Consumer<T> {

    /** Performs this operation on the given argument. ...*/
    void accept(T t);
}
```

这种有且仅有一个抽象方法的接口称为函数式接口，可以被隐式和Lambda表达式互相转换。所有这种需要“函数式接口”的地方也都可以使用Lambda表达式。比如JDK中最有名的Runnable接口，在Java 8中就可以被视为函数式接口

```
@FunctionalInterface
public interface Runnable {
    /** When an object implementing interface Runnable is used ...*/
    public abstract void run();
}
```



```
Runnable r1 = () -> System.out.println("Hello World 1"); ← 使用Lambda

Runnable r2 = new Runnable(){
    public void run(){
        System.out.println("Hello World 2");
    }
}; ← 使用匿名类
```

Lambda表达式~函数式接口

所以我们可以这样认为：函数式接口就是作为使用Lambda表达式的一种“语法载体”，当你设计一个期待接收Lambda表达式的地方，就可以在语法定义上使用“函数式接口”。在Java这种强类型语言中，任何变量都需要有类型，但是又想拥有python这样函数式语言的灵活性的时候，语言通过引入“函数式接口”来完成“函数作为参数”的设计“函数式语言风格”的任务。所以，为了简化开发，在Java8中定义了大量的函数式接口，供开发者使用，简化日常定义“函数式接口”的任务，举例如下：

java.util.function 它包含了很多类，用来支持 Java的 函数式编程，该包中的函数式接口有：

序号	接口 & 描述
1	BiConsumer<T,U> 代表了一个接受两个输入参数的操作，并且不返回任何结果
2	BiFunction<T,U,R> 代表了一个接受两个输入参数的方法，并且返回一个结果
3	BinaryOperator<T> 代表了一个作用于两个同类型操作符的操作，并且返回了操作符同类型的结果
4	BiPredicate<T,U> 代表了一个两个参数的boolean值方法
5	BooleanSupplier 代表了boolean值结果的提供方
6	Consumer<T> 代表了接受一个输入参数并且无返回的操作

Lambda表达式~进一步使用Lambda表达式

我们自己来完成一个Lambda表达式的例子，我们来编写一个可以将任何一个类型的List转换为另外一个List类型的方法。我们编写如下方法：

```
/**
 * 映射转换任何类型列表
 * @param sourceList
 * @param converter
 * @param <T>
 * @param <R>
 * @return
 */
public <T,R> List<R> convertList(List<T> sourceList, Function<T, R> converter){
    List<R> retList = new ArrayList<>();

    for(T t : sourceList){
        retList.add(converter.apply(t));
    }

    return retList;
}
```

Lambda表达式~进一步使用Lambda表达式

然后我们来使用上面编写的转换方法，使用Lambda表达式作为参数来运行时进行类型转换

```
//使用Function的函数式接口
//映射String到length
List<String> listString = Arrays.asList("eews", "", "bc", "zzzd", "badfsdfsfs", "w", "liyghjk");
List<Integer> listLength = convertList(listString, (String s)->s.length());
listLength.stream().forEach(s->System.out.print(s.toString() + ","));

System.out.println();
//映射user到学号
List<User> listUser = Arrays.asList(
    new User( name: "张三", sno: "S0001"),
    new User( name: "李四", sno: "S0002"),
    new User( name: "王五", sno: "S0003"),
    new User( name: "钱六", sno: "S0004")
);
List<String> listSno = convertList(listUser, (User u)->u.getSno());
listSno.stream().forEach(s->System.out.print(s + ","));
```


别忘记我们引入Lambda表达式的核心就是希望能够把函数参数化，也就是把我们想执行的操作在运行的时候“传递来传递去”，以提升代码的灵活性和可复用性，Lambda表达式的实质就是把一种“算法”作为参数，使得各种“算法”可以随时根据需要组合为“想要的算法”。

Lambda表达式~方法引用

如果我们已经具有的大量方法也想用的Lambda表达式中，我们该如何处理呢，最基本的方法就是使用下面的Lambda表达式包装一下，比如下面的map的参数，但是有了方法引用可以简化这个写法，然后让编译器帮你完成，这个没带来任何功能上的强大，却使得代码的可读性更好，当然打字也会少一些；

```
System.out.println(  
    listString.stream()  
        .map(item->item.toUpperCase())  
        .collect(Collectors.joining(delimiter: ","))  
);
```

```
System.out.println(  
    listString.stream()  
        .map(String::toUpperCase)  
        .collect(Collectors.joining(delimiter: ","))  
);
```



**你也需会发现，其实使用stream的map方法就可以起到我们之前lambda编写的convertList的作用，的确是！*

Lambda表达式~引用作用域内的变量


Lambda表达式可以引用表达式所在的环境内的变量，但是不能修改变量的值。

```
//引用作用域内的变量
```

```
User teacher = new User( name: "老师", sno: "T0001");  
listUser.stream().forEach(u->System.out.println(u.getName() + "'s teacher is " + teacher.getName()));  
int index = 0;  
listUser.stream().forEach(u->  
    System.out.println((index++) + u.getName() + "'s teacher is " + teacher.getName()));
```

Variable used in lambda expression should be final or effectively final

- 1) 可以引用teacher变量;
- 2) 尝试在Lambda表达式中修改作用域中的变量index会出现错误;



最后一个小例子~做什么

我们写一个小程序来完成下面的功能

- 1) 一个单位有若干员工，员工对象中有姓名，身份证和12个月的工资
- 2) 我们设计一个计算程序对所有员工进行计税计算（12个月工资总额大于起征额才纳税，纳税额为超出部分的20%）
- 3) 我们用Annotation控制一个类是否需要日志明细信息；

最后一个小例子~实现

我们用Stream来生成测试数据

```
new TaxPayer( name: "张三"  
    , random.doubles( streamSize: 12, randomNumberOrigin: 3000, randomNumberBound: 6000)  
    .boxed().toArray(Double[]::new)),
```

用Stream计算总收入

```
public Double totalIncome(){  
    return this.getIncomeOfMonth().stream()  
        .reduce( identity: 0.0, (m1,m2)->m1 + m2);  
}
```


用Stream执行纳税人筛选并计算总纳税金额

```
Double taxTotal = companyEmployee.stream()  
    .filter(item->item.totalIncome() > TAX_START)  
    .map(item->logAndMap(item))  
    .reduce( identity: 0.0, (t1, t2)->t1 + t2 * 0.2);
```

Annotation控制是否需要日志对象详细信息

```
public boolean isNeedLog(Object obj){  
    return obj.getClass().getAnnotation(EntityLog.class) != null;  
}
```

**这里用Annotation控制
是否输出日志不是一个通常做法，仅仅是编写例子
演示*



参考文章

1. <https://docs.oracle.com/javase/tutorial/java/annotations/basics.html>
2. <https://www.runoob.com/w3cnote/java-annotation.html>
3. <https://blog.csdn.net/u013703461/article/details/66259013>
4. <https://www.runoob.com/java/java8-streams.html>
5. <https://www.jianshu.com/p/11c925cdba50>
6. <https://www.ibm.com/developerworks/cn/java/j-lo-java8streamapi/>
7. <https://www.jianshu.com/p/c204e3721733>



END

Pb&Lois