Lecture4-4 Lambda

1. 行为参数化 Behavior parameterization

行为参数化是一种软件开发模式，它允许处理频繁的需求更改

简而言之，它意味着获取一个代码块并使其可用而不执行它，这个代码块可以稍后由程序的其他部分调用，可以推迟该代码块的执行

可以将代码块作为参数传递给稍后将执行它的另一个方法。因此，方法的行为是基于该代码块参数化的

处理某个集合

如果你在处理某个集合，你可能会想写个方法来

- 对于集合中的某个元素“做些什么”
- 当完成处理集合后“做些什么”
- 当遇到了一些错误的时候“做些什么”

2. Lambda 表达式介绍

Lambda 有下面的几个特点

- **匿名的 Anonymous**：它不像通常的方法那样有一个显式的名称，更少需要编写和思考
- **函数的 Function**：Lambda 不像方法那样与特定的类相关联，但与方法一样，lambda也有一个参数列表、一个主体、一个返回类型和一个可能被抛出的异常列表
- **传递的 Passed around**：Lambda 表达式可以作为参数传递给方法或存储在变量中
- **简洁的 Concise**：不需要像编写匿名类那样编写大量的样板文件

使用 Lambda 之前

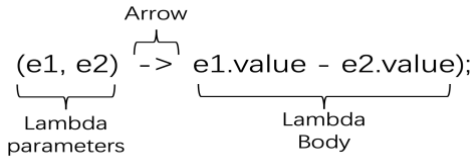
```
Collections.sort(arrayList, new Comparator<Element>() {
    @Override
    public int compare(Element o1, Element o2) {
        return o1.value - o2.value;
    }
});
```

使用 Lambda 之后

```
Collections.sort(arrayList, (e1, e2)->e1.value - e2.value);
```

3. Lambda 的组成

Lambda 由 3 个部分组成



- **参数列表 list of parameters**：在本例中，它反映了 `Comparator` 对象的 `compare()` 方法的参数
- **箭头 arrow**：将参数列表与 lambda 体分离
- **lamba 体**：在本例中位使用两个元素的值比较它们，该表达式被认为是 Lambda 的返回值

一些不同的表达 Lambda 表达式的示例

```
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArgument = event -> System.out.println("botton clicked");

Runnable multiStatement = () -> {
    System.out.print("Hello");
    System.out.println(" World")
}

BinaryOperator<Long> twoArguments = (x, y) -> x + y;

BinaryOperator<Long> explicitArguments = (Long x, Long y) -> x + y;
```

用例	Lambda 表达示例
逻辑表达 return boolean	`(List<String> list) -> list.isEmpty()`
创造一个 object	`() -> new Integer(10)`
调用 object 的方法	`(e) -> {System.out.println(e);}`
选择 / 提取一个对象	`(String s) -> s.length()`
结合两个值	`(int a, int b) -> a * b`
比较两个 object	`(e1, e2) -> e1.value - e2.value`

3. 函数式接口 Functional Interface

引入

```
Collections.sort(arrayList, new Comparator<Element>() {
    @Override
    public int compare(Element o1, Element o2) {
        return o1.value - o2.value;
    }
});
```

上述的代码，将一个 Lambda 赋值给一个 `Comparator<Element>` 类的变量，然后将它传入 `Collections.sort()` 作为第二个参数

事实上，`Comparator<T>` 是一个函数式接口

```
public interface Comparator<T>{
    int compare(T o1, T o2);
}
```

函数式接口是只有一个指定一个抽象方法的接口

一些常见的函数式接口

```
public interface Runnable{
    void run();
}
```

```
public interface ActionListener extends EventListener{
    void actionPerformed(ActionEvent e);
}
```

```
public interface Callable<V>{
    V call();
}
```

```
public interface PrivilegeAction<V>{
    T run();
}
```

- 一个接口也可以有很多**默认方法 default method**（提供方法体的实现）
- 即算一个接口有很多个默认方法，但只要它只有一个抽象方法，那么它也可以被称为函数式接口

4. 重要的函数式接口

接口名称	参数	返回值	示例
`Predicate<T>`	T	boolean	相册时候被公布
`Consumer<T>`	T	void	打印一个值
`Function<T, R>`	T	R	获得 Artist 对象的名称
`Supplier<T>`	void	T	工厂方法
`UnrayOperator<T>`	T	T	逻辑非
`BinaryOperator<T>`	(T, T)	T	将两个数相乘

Predicate<T>

```
@FunctionalInterface
public interface Predicate<T>{
    boolean test(T t);
}
```

`java.util.function.Predicate<T>` 接口定义了一个抽象方法 `test()`，该方法接受泛型类型 `T` 的对象并返回一个布尔值，当需要表示使用 `T` 类型对象的布尔表达式时，可能需要使用此接口

可以定义如下的泛型函数，来根据某个条件过滤某个 List 中的对象

```
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for (T s : list) {
        if (p.test(s)) {
            results.add(s);
        }
    }
    return results;
}
```

可以定义一个 lambda 函数接受一个 String 的 list，过滤掉空的字符串

```
public static void main(String[] args) {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("");
    listOfStrings.add("abc");
    listOfStrings.add("\n");
    listOfStrings.add("e");
    List<String> nonEmpty = filter(listOfStrings, (String s) -> !s.isEmpty());
    System.out.println(nonEmpty);
}
```

Consumer<T>

```
@FunctionalInterface
public interface Consumer<T>{
    void accept(T t);
}
```

`java.util.function.Consumer<T>` 接口定义了一个抽象方法 `accept()`，该方法接受泛型类型 `T` 的对象，返回空，当您需要访问 `T` 类型的对象并对其执行一些操作时，可以使用此接口

可以定义如下的泛型函数，来根据某个条件对 list 中的每个对象执行某种操作

```
public static <T> void forEach(List<T> list, Consumer<T> c) {
    for (T s : list) {
        c.accept(s);
    }
}
```

可以定义一个 lambda 函数接受一个 String 的 list，打印 list 中的每个元素

```
public static void main(String[] args) {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("");
    listOfStrings.add("abc");
    listOfStrings.add("\n");
    listOfStrings.add("e");
    forEach(listOfStrings, (s)-> System.out.println(s));
}
```

Function<T, R>

```
public interface Function<T, R> {
    R apply(T t);
}
```

`java.util.function.Function<T,R>` 接口定义了一个抽象方法 `apply()`，该方法接受泛型类型 `T` 的对象，返回一个 `R` 类型的对象，当需要定义将信息从输入对象映射到输出（例如，提取苹果的权重或将字符串映射到其长度）的 Lambda时，可以使用此接口

可以定义如下的函数，它将 list 中的元素逐一映射成了另一个类型

```
public static <T, R> List<R> map(List<T> list, Function<T,R> f){
    List<R> results = new ArrayList<>();
    for(T s : list){
        results.add(f.apply(s));
    }
    return results;
}
```

可以定义一个 lambda 函数接受一个 String 的 list，映射成每个 String 的长度的 list

```
public static void main(String[] args) throws InvalidKeyException {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("");
    listOfStrings.add("abc");
    listOfStrings.add("\n");
    listOfStrings.add("e");
    List<Integer> listStrLen1 = map(listOfStrings, (s)->s.length());
}
```

5. 函数式接口的具体应用

基本数据 Lambda 表达式

上面说明了常见的一些函数式接口，对于 `double` 和 `long` 等基本类型，这些接口也都有相应的版本

```
IntPredicate isEven = i -> i % 2 == 0;

IntConsumer pringInt = i -> System.out.println(Integer.toString(i));

IntFunction<String> intToString = i -> Integer.toString(i);
ToIntFunction<String> perseInt = str -> Integer.valueOf(str);
IntToDoubleFunction intAsDouble = i -> Integer.valueOf(i).doubleValue();
```

```
8   ToIntBiFunction<String, String> maxLength = (left, right) ->
    Math.max(left.length(), right.length());
9
10  IntSupplier randomInt = () -> new Random().nextInt();
11
12  IntUnrayOperator negateInt = i -> -1 * i;
13
14  IntBinaryOperator multiplyInt = (x,y) -> x*y;
```

静态方法 Lambda 表达式

Lambda 表达式可以写成静态方法的格式

```
1   IntFunction<String> intToString = Integer::toString;
2   ToIntFunction<String> perseInt = Integer::valueOf;
```

构造方法 Lambda 表达式

```
1   Function<String, BigInteger> newBigInt = BigInteger::new;
```