关于作者

2018年01月10日 阅读 2158

Java 8 Lambda表达式一看就会

匿名内部类的一个问题是：当一个匿名内部类的实现非常简单，比如说接口**只有一个抽象函数**，那么匿名内部类的语法有点笨拙且不清晰。我们经常会有传递一个函数作为参数给另一个函数的实际需求，比如当点击一个按钮时，我们需要给按钮对象设置按钮响应函数。lambda表达式就可以把函数当做函数的参数，代码（函数）当做数据（形参），这种特性满足上述需求。当要实现只有一个抽象函数的接口时，使用lambda表达式能够更灵活。

使用Lambda表达式的一个用例

假设你正在创建一个社交网络应用。你现在要开发一个可以让管理员对用户做各种操作的功能，比如搜索、打印、获取邮件等操作。假设社交网络应用的用户都通过 `Person` 类表示：

java 复制代码

```java
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    private String name;

    private LocalDate birthday;

    private Sex gender;
```

```
    private String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

假设社交网络应用的所有用户都保存在一个 `List<Person>` 的实例中。

我们先使用一个简单的方法来实现这个用例，再通过使用本地类、匿名内部类实现，最终通过lambda表达式做一个高效且简洁的实现。

方法1：创建一个根据某一特性查询匹配用户的方法

最简单的方式是创建几个函数，每个函数搜索指定的用户特征，比如 `searchByAge()` 这种方法，下面的方法打印了年龄大于某特定值的所有用户：

java 复制代码

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

这个方法是有潜在的问题的，如果引入一些变动（比如新的数据类型）这个程序会出错。假设更新了应用且变化了 `Person` 类，比如使用出生年月代替了年龄；也有可能搜索年龄的算法不同。这样你将不到不再写许多API来适应这些变化。

方法2：创建一个更加通用的搜索方法

这个方法比起 printPersonsOlderThan 更加通用;它提供了可以打印某个年龄区间的用户：

java 复制代码

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

如果想打印特定的性别或者打印同时满足特定性别和某年龄区间的用户呢？如果要改动Person类，添加其他属性，比如恋爱状态、地理位置呢？尽管这个方法比 printPersonsOlderThan 方法更加通用，但是每个查询都创建特定的函数都是有可以导致程序不够健壮。你可以使用接口将特定的搜索转交给需要搜索的特定类中（面向接口编程的思想——简单工厂模式）。

方法3：在本地类中设定特定的搜索条件

下面的方法可以打印出符合搜索条件的所有用户信息

java 复制代码

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

这个方法通过调用 `tester.test` 方法检测每个 `roster` 列表中的元素是否满足搜索条件。如果 `tester.test` 返回true，则打印符合条件的 `Person` 实例。

通过实现 `CheckPerson` 接口实现搜索。

java 复制代码

```
interface CheckPerson {
    boolean test(Person p);
}
```

下面的类实现了 `CheckPerson` 接口的 `test` 方法。如果 `Person` 的属性是男性并且年龄在18到25岁之间将会返回true

java 复制代码

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

当要使用这个类的时候，只需要实例化一个实例，并将实例以参数的形式传递给 `printPersons` 方法。

java 复制代码

```
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

尽管这个方式不那么脆弱——当 `Person` 发生变化时你不需要重新更多方法，但是你仍然需要在添加一些代码：要为每个搜索标准创建一个本地类来实现接口。 `CheckPersonEligibleForSelectiveService` 类实现了一个接口，你可以使用一个匿内部类替代本地类，通过声明一个新的内部类来满足不同的搜索。

方法4：在匿名内部类中指定搜索条件

下面的 `printPersons` 函数调用的第二个参数是一个匿名内部类，这个匿名内部类过滤满足性别为男性并且年龄在18到25岁之间的用户：

java 复制代码

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

这个方法减少了很多代码量，因为你不必为每个搜索标准创建一个新类。但是，考虑到 `CheckPerson` 接口只有一个函数，匿名内部类的语法有显得有点笨重。在这种情况下，可以考虑使用lambda表达式替换匿名内部类，像下面介绍的这种。

方法5：通过Lambda表达式实搜索接口

`CheckPerson` 接口是一个**函数式接口**。接口中只有一个抽象方法的接口属于函数式接口（一个函数式接口也可能包换一个活多个默认方法或者静态方法）。由于函数式接口只包含一个抽象方法，你可以在实现该方法的时候省略方法的名字。因此你可以使用**lambda表达式**取代匿名内部类表达式，像下面这样调用:

java 复制代码

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

lambda表达式的语法后面会做详细介绍。你还可以使用**标准的函数式接口**取代 `CheckPerson` 接口，这样会进一步减少代码量。

方法6：使用标准的函数式接口和Lambda表达式

`CheckPerson` 接口是一个非常简单的接口：

java 复制代码

```
interface CheckPerson {
    boolean test(Person p);
}
```

它只有一个抽象方法，因此它是一个函数式接口。这个函数有个一个参数和一个返回值。它太过简单以至于没有必要在你应用中定义它。因此JDK中定义了一些标准的函数式接口，可以在 `java.util.function` 包中找到。比如，你可以使用 `Predicate<T>` 取代 `CheckPerson` 。这个接口中只包含 `boolean test(T t)` 方法。

java 复制代码

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

`Predicate<T>` 是一个泛型接口，泛型需要在尖括号（<>）指定一个或者多个参数。这个接口中只包换一个参数T。当你声明或者通过一个真实的类型参数实例化泛型后，你将得到一个参数化的类型。比如，参数化后的类型 `Predicate<Person>` 像下面代码所示：

java 复制代码

```
interface Predicate<Person> {
    boolean test(Person t);
}
```

参数化后的的接口包含一个接口，这和 `CheckPerson.boolean test(Person p)` 完全一样。因此，你可以像下面的代码一样使用 `Predicate<T>` 取代 `CheckPerson` ：

java 复制代码

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

那么，可以这样调用这个函数：

java 复制代码

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

这个不是使用lamdba表达式的唯一的方式。建议使用下面的其他方式使用lambda表达。

方法7：在应用中全都使用Lambda表达式

再来看看方法 `printPersonsWithPredicate` 哪里还可以使用lambda表达式：

java 复制代码

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

这个方法检测 `roster` 中的每个 `Person` 实例是否满足 `tester` 的标准。如果Person实例满足 `tester` 中设定的标准，那么 `Person` 实例的信息将会被打印出来。

你可以指定一个不同的动作来执行打印满足 `tester` 中定义的搜索条件的 `Person` 实例。你可以指定这个动作是一个lambda表达式。假设你想要一个功能和 `printPerson` 一样的lambda表示式（一个参数、返回void），你需要实现一个函数式接口。在这种情况下，你需要一个包含一个只有一个Person类型参数和返回void的函数式接口。 `Consumer<T>` 接口包换一个 `void accept(T t)` 函数，它符合上述需求。下面的函数使用 `Consumer<Person>` 调用 `accept()` 从而取代了 `p.printPerson()` 的调用。

java 复制代码

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }
}
```

那么可以这样调用 `processPersons` 函数：

java 复制代码

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

如果你想对用户的信息进行更多处理而不止打印出来，那该怎么办呢？假设你想验证成员的个人信息或者获取他们的联系人的信息呢？在这种情况下，你需要一个有返回值的抽象函数的函数式

接口。 `Function<T,R>` 接口包含了 `R apply(T t)` 方法，有一个参数和一个返回值。下面的方法获取参数匹配到的数据，然后根据lambda表达式代码块做相应的处理：

java 复制代码

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

下面的函数从 `roster` 中获取符合搜索条件的用户的邮箱地址，并将地址打印出来。

java 复制代码

```
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

方法8：使用泛型使之更加通用

再处理 `processPersonsWithFunction` 函数，下面的函数可以接受包含任何数据类型的集合：

java 复制代码

```
public static <X, Y> void processElements(
    Iterable<X> source,
```

```
        Predicate<X> tester,
        Function <X, Y> mapper,
        Consumer<Y> block) {
        for (X p : source) {
            if (tester.test(p)) {
                Y data = mapper.apply(p);
                block.accept(data);
            }
        }
    }
```

可以这样调用上述函数来实现打印符合搜索条件的用户的邮箱：

java 复制代码

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

该方法的调用只要执行了下面动作：

1. 从集合中获取对象，在这个例子中它是包换 `Person` 实例的 `roster` 集合。 `roster` 是一个List类型，同时也是一个Iterable类型。
2. 过滤符合 `Predicate` 数据类型的 `tester` 的对象。在这个例子中，Predicate对象是一个指定了符合搜索条件的lambda表达式。
3. 使用 `Function` 类型的mapper映射每个符合过滤条件的对象。在这个例子中，Function对象时要给返回用户的邮箱地址。
4. 对每个映射到的对象执行一个在 `Consumer` 对象块中定义的的动作。在这个例子中，Consumer对象时一个打印Function对象返回的电子邮箱的lamdba表达式。

你可以通过一个聚合操作取代上述操作。

方法9：使用lambda表达式作为参数的合并操作

下面的例子使用了聚合操作，打印出了符合搜索条件的用户的电子邮箱：

java 复制代码

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

下面的表映射了 `processElements` 函数执行操作和与之对应的聚合操作

processElements动作	**聚合操作**
获取对象源	Stream stream()
过滤符合**Predicate对象**（lambda表达式）的实例	Stream filter(Predicate<? super T> predicate)
使用**Function对象**映射符合过滤标准的对象到一个值	Stream map(Function<? super T,? extends R> mapper)
执行**Consumer对象**（lambda表达式）设定的动作	void forEach(Consumer<? super T> action)

`filter` , `map` 和 `forEach` 是聚合操作。聚合操作是从 `stream` 中处理各个元素的，而不是直接从集合中（这就是为什么第一个调用的函数是 `stream()` ）。steam是对各个元素进行序列化操作。和集合不同，它不是一个储存数据的数据结构。相反地，stream加载了源中的值，比如集合通过 `pipeline` 将数据加载到stream中。 `pipeline` 是stream的一种序列化操作，这个例子中的就是 `filter- map-forEach` 。还有，聚合操作通常可以接收一个lambda表达式作为参数，这样你可自定义需要的动作。

在GUI程序中使用lambda表达式

为了处理一个图形用户界面（GUI）应用中的事件，比如键盘输入事件，鼠标移动事件，滚动事件，你通常是实现一个特定的接口来创建一个事件处理。通常，时间处理接口就是一个函数式接口，它们通常只有一个函数。

之前使用匿名内部类实现的时间相应：

java 复制代码

```
btn.setOnAction(new EventHandler<ActionEvent>() {
           @Override
           public void handle(ActionEvent event) {
               System.out.println("Hello World!");
           }
       });
```

可以使用如下代码替代：

java 复制代码

```
btn.setOnAction(
         event -> System.out.println("Hello World!")
       );
```

Lambda表达式语法

一个lambda表达式由一下结构组成：

- `()` 括起来参数，如果有多个参数就使用逗号分开。 `CheckPerson.test` 函数有一个参数p,代表Person的一个实例。

  注意： 你可以省略lambda表达式中的参数类型。另外，如果只有一个参数也可以省略括号。比如下面的lambda表达式也是合法的：

java 复制代码

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

- 箭头符号：->
- 主体：有一个表达式或者一个声明块组成。例子中使用这样的表达式：

java 复制代码

```
p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

如果设定的是一个表达式，java运行时将会计算表达式并最终返回结果。同时，你可以使用一个返回声明：

java 复制代码

```
p -> {
    return p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
}
```

在lambda表达式中返回的不是一个表达式，那么就必须使用 {} 将代码块括起来。但是，当返回的是一个 void 类型时则不需要括号。比如，下面的也是一个合法的lambda表达式：

java 复制代码

```
email -> System.out.println(email)
```

lambda表达式看起来有点像声明函数，可以把lambda表达式看做是一个匿名函数（没有名称的函数）。

下面是一个有多个形参的lambda表达式的例子：

java 复制代码

```
public class Calculator {
```

```
    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

方法 operateBinary 执行两个数的数学操作。操作本身是对 IntegerMath 类的实例化。实例中通过lambda表达式定义了两种操作，加法和减法。例子输出结果如下：

java 复制代码

```
40 + 2 = 42
20 - 10 = 10
```

获取闭包中的本地变量

像本地类和匿名类一样，lambda表达式也可以访问本地变量；它们有访问本地变量的权限。lambda表达式也是属于当前作用域的，也就是说它**不从父级作用域中继承任何命名名称，或者引入新一级的作用域**。lambda表达式的作用域就是声明它所在的作用域。下面的这个例子说明了这一点：

java 复制代码

```
import java.util.function.Consumer;
```

```
public class LambdaScopeTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {

            Consumer<Integer> myConsumer = (y) ->
            {
                System.out.println("x = " + x);
                System.out.println("y = " + y);
                System.out.println("this.x = " + this.x);
                System.out.println("LambdaScopeTest.this.x = " +
                    LambdaScopeTest.this.x);
            };
            myConsumer.accept(x);
        }
    }

    public static void main(String... args) {
        LambdaScopeTest st = new LambdaScopeTest();
        LambdaScopeTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

将会输出如下信息：

java 复制代码

```
x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0
```

如果像下面这样在lambda表达式 `myConsumer` 中使用x取代参数y，那么编译将会出错。

java 复制代码

```
Consumer<Integer> myConsumer = (x) -> {
}
```

编译会出现"variable x is already defined in method methodInFirstLevel(int)",因为lambda表达式不引入新的作用域（lambda表达式所在作用域已经有x被定义了）。因此，可以直接访问lambda表达式所在的闭包的成员变量、函数和闭包中的本地变量。比如，lambda表达式可以直接访问方法methodInFirstLevel的参数x。可以使用this关键字访问类级别的作用域。在这个例子中this.x对成员变量FirstLevel.x的值。

然而，像本地和匿名类一样，lambda表达式值可以访问被修饰成final或者effectively final的本地变量和形参。比如，假设在 `methodInFirstLevel` 中添加定义声明如下：

> Effectively Final:一个变量或者参数的值在初始化后就不在发生变化，那么这个变量或者参数就是effectively final类型的。

java 复制代码

```
void methodInFirstLevel(int x) {
    x = 99;
}
```

由于 `x =99` 的声明使 `methodInFirstLevel` 的形参x不再是effectively final类型。结果java编译器就会报类似"local variables referenced from a lambda expression must be final or effectively final"的错误。

目标类型

在运行时java是怎么判断lambda表达式的数据类型的？再看一下那个要选择性别是男性，年龄在18到25岁之间的lambda表达式：

```java
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

这个lambda表达式已参数的形式传递到如下两个函数：

- public static void printPersons(List roster, CheckPerson tester)
- public void printPersonsWithPredicate(List roster, Predicate tester)

当java运行时调用方法 `printPersons` 时，它期望一个 `CheckPerson` 类型的数据，因此lambda表达式就是这种类型。当java运行时调用方法 `printPersonsWithPredicate` 时，它期望一个 `Predicate<Person>` 类型的数据，因此lambda表达式就是这样一个类型。这些方法期望的数据类型就叫目标类型。为了确定lambda表达式的类型，java编译器会在lambda表达式的的上下文中判断它的目标类型。只有java编译器可推测出来了目标类型，lambda表达式才可以被执行。

目标类型和函数参数

对于函数参数，java编译器可以确定目标类型通过两种其他语言特性：重载解析和类型参数推断。看下面两个函数式接口(java.lang.Runnable and java.util.concurrent.Callable):

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

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

方法 `Runnable.run` 不返回任何值，但是 `Callable<V>.call` 有返回值。假设你像下面一样重载了方法 `invoke` ：

```java
void invoke(Runnable r) {
```

```
        r.run();
    }

    <T> T invoke(Callable<T> c) {
        return c.call();
    }
```

那么执行下面程序哪个方法将会被调用呢?

java 复制代码

```
String s = invoke(() -> "done");
```

方法 `invoke(Callable<T>)` 会被调用，因为这个方法有返回一个值；方法 `invoke(Runnable)` 没有返回值。在这种情况下，lambda表达式 `() -> "done"` 的类型是 `Callable<T>` 。

最后

感谢阅读，有兴趣可以关注微信公众账号获取最新推送文章。

> **关注下面的标签，发现更多相似文章**
>
> Java Android
>
> 发布了 篇专栏 · 获得点赞 次 · 文章被阅读 次
>
> 安装掘金浏览器插件
> 打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧!

评论

输入评论...