一、Lamda表达式之函数式接口介绍及表达式初识

谜00016 关注

0.586 2018.05.13 10:48:49 字数 1,288 阅读 848

注：部分素材来源Lambda表达式及函数式接口介绍-51CTO学院

最近在看一些三方框架的时候，发现在这些框架中大量的使用了java8的新特性，比如随处可见的lamda表达式以及Stream类的很多骚操作，看的是一脸懵逼！在当今社会，颠覆在时时刻刻的上演，故步自封只能被无情的淘汰！拥抱变化这是你我唯一能做的！

回到正题，如何看懂框架源码中这些“奇奇怪怪”的代码（不明觉厉），因此很有必要学习一下java8的某些重要的新特性，Lamda表达式当仁不让。

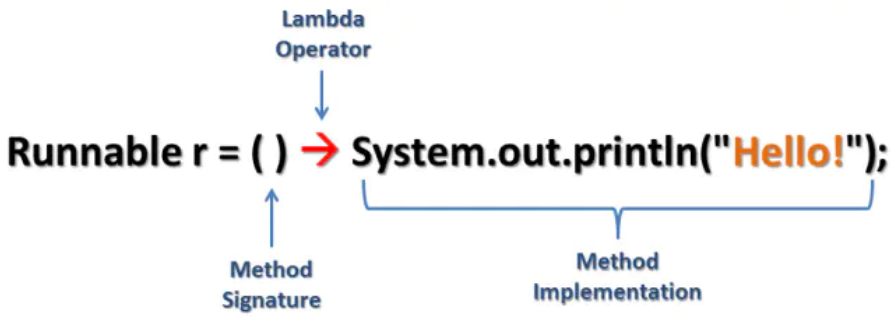
我们先了解一下基本的理论知识

1、什么是Lamda表达式

Lambda表达式是 Java8 中最重要的新功能之一。Lambda表达式是对象，是一个**函数式接口**的实例。使用 Lambda 表达式可以替代只有一个抽象函数的接口实现，告别匿名内部类，代码看起来更简洁易懂。Lambda表达式同时还提升了对集合、框架的迭代、遍历、过滤数据的操作。

2、Lamda表达式的语法格式

args -> expr或者(Object... args) -> {函数式接口抽象方法实现逻辑}，()里面参数的个数，根据函数式接口里面抽象方法的参数个数来决定。当只有一个参数的时候，()可以省略当expr逻辑非常简单的时候，{}和return可以省略



2、Lamda表达式特点

1）：函数式编程

2）：参数类型自动推断

3）：代码量少，简洁

3、Lamda表达式的优点

1）：更简洁的代码

2）：更容易的并行

4、如何学好Lamda表达式

1）：熟悉泛型

2）：多练，多用Stream API

5、Lambda表达式使用场景

任何有函数式接口的地方

6、什么是函数式接口（重点）

只有一个**抽象方法**（Object类中的方法除外）的接口是函数式接口

上面简单介绍了一些基本的理论知识，理解起来可能有些抽象。怎么办呢？下面我们具体的来学习一下，通过实例练习之后，我相信回过头再看这些理论知识会好很多。

我们举个最常用的lamda应用场景：

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


    new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("普通--》线程执行");
        }
    }).start();

    new Thread(()-> System.out.println("lamda--->线程执行")).start();
}
```

之前我们从理论知识中知道，lamda的应用场景是在有函数式接口的地方，那么我们是不是可以理解为Runnable为函数式接口呢?我们

重温一下函数式接口定义： 有且只有一个抽象方法（非Object方法）的接口为函数式接口。

我们来看下Runnable接口源码：

```
 */
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

不出所料，Runable接口正是标准的函数式接口。细心的同学应该发现了Runnable接口上面有个注解：FunctionalInterface，该注解正是标注该接口是函数式接口。如果不是函数式接口，添加这个注解会出错。举个例子：

```
@FunctionalInterface
interface UserMapper {
```

第一个错误原因是不满足只有一个抽象方法的条件，第二个错误的原因是hashCode()方法是object方法，除去之外没有了抽象方法，不满足有一个抽象方法的条件，因此也不是函数式接口。

我们来了解一下jdk中一些非常常用的函数式接口（重点罗列常用接口），熟悉这些接口是非常必要的。（先留个印象，后面以及实际开发中会常用到这些接口）

Consumer 代表一个输入　Supplier 代表一个输出

Function 代表一个输入，一个输出（一般输入和输出是不同类型的）　UnaryOperator 代表一个输入，一个输出（输入和输出是相同类型的）

BiConsumer 代表两个输入

BiFunction 代表两个输入，一个输出（一般输入和输出是不同类型的）BinaryOperator 代表两个输入，一个输出（输入和输出是相同类型的）

我是这么记忆的，分为四组，

第一组单次元的输入Supplier和输出Comsumer，

第二组两个输入的BiConsumer

第三组二次元的输入输出不同类型Function(一般不同类型，但是也是可以同类型的)以及输入输出同类型UnaryOprator

第四组两个输入，一个输出且输入输出不同类型的BiFunction(同时也是可以同类型的）以及输入输出同类型的BinaryOperator 。

下面就上面几个常用的函数式接口进行一些简单的试用。

```
/*
* 一个输入
* */
Consumer<User> consumer = (a) -> {
    System.out.println("一个输入");
    a.setAge(45);
};

User u =new User();
consumer.accept(u);
System.out.println(u.getAge());
```

User是我自定义的一个类，有age。我们使用泛型约束输入类型，输入user的实例，对age进行重新赋值。

```
/*
* 一个输出
* */
Supplier<User> supplier = () -> new User();
/*代替无参的工厂模式*/
User user =  supplier.get();
```

```java
/*
* 两个输入
* */
BiConsumer<String ,User> biConsumer = (a, b) -> {
    //逻辑代码
    System.out.println("两个输入");
    System.out.println(a);
    System.out.println(b.getAge());
};

biConsumer.accept( t: "测试",new User());
```

```java
/*一个输入一个输出（一般输入输出不同类型）*/
Function <User,Integer> function=(a)->{return a.getAge();};
Integer apply = function.apply(new User( age: 5));
System.out.println(apply);
```

```java
/*一个输入一个输出相同类型*/
UnaryOperator<User> unaryOperator=(a)->{return new User( age: a.getAge()+1);};
User apply1 = unaryOperator.apply(new User( age: 2));
System.out.println(apply1.getAge());
```

```java
/*两个输入一个输出，一般输入和输出是不同类型的*/
BiFunction<Integer,String,User> biFunction=(a,b)->{return new User(a,b);};
User jack = biFunction.apply( t: 25,  u: "Jack");
System.out.println(jack.getAge()+jack.getName());
```

```java
/*两个输入，且输入和输出是相同类型*/
BinaryOperator<User> binaryOperator=(a,b)->{return new User(a.getAge(),b.getName());};
User bob = binaryOperator.apply(new User( age: 36), new User( name: "Bob"));
System.out.println(bob.getAge()+bob.getName());
```