Please visit website: http://cxyroad.com

一文涵盖Lambda,Stream,响应式编程，从此爱上高效率编程
==========

一文涵盖Lambda,Stream,响应式编程，从此爱上高效率编程
==========

前言
--

本文结构为 先是一个例子，带你快速体验，之后再去深究里面的方法。以及一些底层原理是如何实现的。从如何用，到如何用好，如何用精。学习操作，学习思维。

Lambda表达式

初体验

这个是Java8的新特性。来源于数学中的演算。

是一套关于函数定义、输入量、输出量的计算方案。

简单来说，Lambda表达式就是一个简单的函数。

首先来看一个案例。

这里有一个接口。

```

public interface Factory {
    Object getObject();
}
```

```

以及实现类

```

```
public class SubClass implements Factory{
    @Override
    public Object getObject() {
        return new User();

    }
}

```

实体类是：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private String name;
    private Integer age;
}

```

我们来看前俩种写法

```
public static void main(String[] args) {
    //1.子类实现接口
    Factory factory = new SubClass();
    User user1 = (User) factory.getObject();
    System.out.println(user1);

    //2.匿名内部类
    factory = new Factory() {
        @Override
```

```
        public Object getObject() {
            return new User(”xiaou”, 18);
        }
    };
    User user2 = (User) factory.getObject();
    System.out.println(user2);
}

```

之后我们来看一个lambda表达式的写法。

```
//3.Lambda表达式
factory = () -> {
    return new User(”xiaou”, 18);
};

```

可以看出来是非常的简单的

通过对比，我们可以看出这个是非常的简单的。

我们来说一下他的优点：

* 让代码变得更加`简洁、紧凑`
* 函数式编程:

+ 函数是”第一等公民”，可以像一个变量一样去调用
+ 可以赋值给变量
+ 可以作为其他函数的参数就行传递
+ 可以作为其他函数的返回值

首先来看第三条

```
//lambda作为参数进行传递
User user4 = getUserFormFactory(() -> {
```

```
    return new User(”xiaou”, 20);
}, ”User”);
System.out.println(user4);
```

```
public static User getUserFormFactory(Factory factory, String
beanName) {
    User object = (User) factory.getObject();
    if (object != null &&
object.getClass().getSimpleName().equals(beanName)) {
        return object;
    }
    return null;
}
```

来看第四条

```
public static Factory getFactory() {
    return () -> {
        return new User(”xiaou”, 21);
    };
}
```

```
//lambda作为参数返回值
Factory factory2 = getFactory();
User user5 = (User) factory2.getObject();
System.out.println(user5);
```

语法细节

![image-20240412115135397](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/943a08b63d0a40b6be5fe5831b7041a8~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1541&h=671&s=192037&e=png&b=fefe

fe)

了解了语法，我们就可以进行一个改进对我们上面的案例。

```
//3.Lambda表达式
factory = () -> {
    return new User(”xiaou”, 19);
};
//改进版
factory = () -> new User(”xiaou”, 19);
```

如果你不知道什么时候应该简写，idea是可以去给你提醒的

![image-20240412115421836](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/737c656fca834890b0ce37080042d40b~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1396&h=327&s=55618&e=png&b=222427)

之后来看他的别的简写语法：

* 函数体只有一个语句，省略大括号

```
  (String msg) -> System.out.println(”hello” + msg);
```

* 只有一个参数，省略圆括号，同时省略类型

```
msg -> System.out.println(”hello ” + msg);
```

* 函数体只有一个表达式，省略return

```
(int a, int b) -> a + b;
```
* 省略参数类型，编译器可以进行推断

```
(a, b) -> a + b;
```

之后来看使用lambda表达式的前提。

使用前提

1.必须有一个函数式接口

函数式接口是Java中的一个关键概念，它指的是具有以下特征的接口：

* **必须有一个抽象方法**：这是函数式接口的核心 特征，接口中只有一个抽象方法。
* **`@FunctionalInterface`注解**：虽然这个注解不是必须的，但它可以用来明确标识一个接口是函数式接口，同时提供编译时检查。

2.常见的函数式接口

Java API提供了多个预定义的函数式接口，这些接口广泛应用于函数式编程中，它们包括：

* **Runnable / Callable**：用于表示没有返回值的可执行任务，Callable还可以返回一个值并抛出异常。
* **Supplier**：表示一个不带参数但可以产生结果的操作。
* **Consumer**：表示一个接受单个参数并返回无内容的操作，通常用于处理输入的数据。
* **Comparator**：用于定义对象比较逻辑。
* **Predicate**：表示一个参数的谓词（布尔值函数），用于判断一个对象是否满足某个条件。

* **Function**：表示一个参数并返回结果的函数。

下面我们来详细讲解这些函数式接口

常见的函数式接口

Runnable

```
public static void main(String[] args) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            System.out.println(name + ”正在运行1”);
        }
    }).start();


    //lambda简化
    new Thread(() -> {
        String name = Thread.currentThread().getName();
        System.out.println(name + ”正在运行2”);
    }).start();
}

```

supplier

```

/**
 * 求数组最大值
 */
public class SupplierLambda {
    public static void main(String[] args) {
        int arr[] = {1, 2, 3, 4, 5};
        int max = getMax(() -> {
            int temp = arr[0];
            for (int i : arr) {
```

```
            if (i > temp) {
                temp = i;
            }
        }
        return temp;
    });
    System.out.println(max);
}

public static int getMax(Supplier<Integer> supplier) {
    return supplier.get();
}
}

```

Consumer

```

```

```
public class ConsumerLambda {
    public static void main(String[] args) {
        consumerString(str -> System.out.println(str));
        consumerString(
                s -> System.out.println(s.toUpperCase()),
                s -> System.out.println(s.toLowerCase())
        );
    }

    static void consumerString(Consumer<String> function) {
        function.accept("Hello");
    }

    static void consumerString(Consumer<String> first,
Consumer<String> sec) {
        first.andThen(sec).accept("Hello World");
    }

}

```

Comparator

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

    String[] arr = {"abc", "def", "ghi"};

    //匿名类方法
    Comparator<String> comparator = new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    };

    //lambda表达式
    Arrays.sort(arr, (o1, o2) -> o1.length() - o2.length());
    System.out.println(Arrays.toString(arr));

}
```

Predicate

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

        // 传统写法
        Predicate<Integer> predicate = new Predicate<Integer>() {
            @Override
            public boolean test(Integer t) {
                return t > 10;
            }
        };
        System.out.println(predicate.test(15));

        // lambda写法
        Predicate<Integer> predicate1 = (t) -> t > 10;
        System.out.println(predicate1.test(15));
    }
```

Function

```
public class FunctionLambda {
    public static void main(String[] args) {
        method((str) -> Integer.parseInt(str) + 10, (num) -> num * 2);
        String str = "xiaou,18";
        int ageNum = getAgeNum(str,
                s -> s.split(",")[1],
                s -> Integer.parseInt(s),
                s -> s + 10);
        System.out.println(ageNum);
    }

    static void method(Function<String, Integer> one, Function<Integer, Integer> two) {
        Integer num = one.andThen(two).apply("10");
        System.out.println(num);
    }

    static int getAgeNum(String str, Function<String, String> one, Function<String, Integer> two, Function<Integer, Integer> three) {
        return one.andThen(two).andThen(three).apply(str);
    }
```

Lambda底层实现

Lambda的本质：

函数式接口的匿名子类的匿名`对象`

我们来看下面的代码

```
    public static void main(String[] args) {
        List<String> stringList = Arrays.asList("掘", "金");
        stringList.forEach(s -> System.out.println(s));
    }
}
```

之后进行一个反编译。

```
public static void main(String[] args) {
    List<String> stringList = Arrays.asList(”\u6398”, ”\u91d1”);
stringList.forEach((Consumer<String>)LambdaMetafactory.metafactory(null, null, null, (Ljava/lang/Object;)V, lambda$main$0(java.lang.String ), (Ljava/lang/String;)V)());
}

private static /* synthetic */ void lambda$main$0(String s) {
    System.out.println(s);
}

```

这个就是反编译的内容

可以看到，他其实就是对一个方法返回的结果进行了一个强转

所以说，他本身就是一个`实例`

这个方法的核心也就是去写了一个字节码文件

![image-20240412143018577](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/d2f53ba5cf9c42a389ae144c93520ef8~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1698&h=994&s=232665&e=png&b=1f2023)

下面是一个过程，比较枯燥，如果想看的可以来阅读一下

> * 为lambda表达式生成一个类，类名类似于：`$$Lambda$xxx$yyy`，其中`xxx`为lambda源码实现所在的类名，`yyy`为一串类似md5字符串的值。具体生成算法暂未深入研究。该类会继承于函数式接口，并且有一个`public static`的名为`INSTANCE`的字段，该字段的类型为`$$Lambda$xxx$yyy`，在类`cinit`中（经典的单例实现）。接口的抽象方法在实现很简单，就是调用一个名为`xxx.lambda$zzz$N`方法，`xxx`为lambda源码实现所在的类名，`zzz`为lambda源码实现所在的类的方法名，`N`为阿拉伯数字。`xxx.lambda$zzz$N`其实就是lambda表达式的具体逻辑的藏身之处。

> * `xxx.lambda$zzz$N`做为lambda实现的类，它是一个位于lambda实现类中的静态方法。如前文所述，它就是lambda表达式的具体实现。
> * 在lambda表达式调用的地方，会将lambda表达式替换为`$$Lambda$xxx$yyy.INSTANCE`。在后续调用接口方法，就直接调用`$$Lambda$xxx$yyy.INSTANCE`的接口实现，接着转调到实现类中的`xxx.lambda$zzz$N`方法，执行lambda真正的逻辑。

你可以把Lambda理解为一个**语法糖**。他用到了ASM技术，有兴趣的也可以去了解一下。下面是一个图来进行一个总结

![image-20240412143405923](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/79661adc141c4f2b8556052c03131871~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1764&h=708&s=325460&e=png&b=fefdfd)

方法引用

初体验

我们也直接来看一个事例。

```
public static void main(String[] args) {
    //调用printString方法，参数是一个Printable类型的对象 也就是函数式接口
    printString(s -> {
        System.out.println(s);
    });
}

public static void printString(Printable p) {
    p.print("Xiaou");
}

```

我们来进行一个简化

```
//调用printString方法，参数是个Printable类型的对象 也就是函数式接口
printString(s -> {
    System.out.println(s);
});


//使用lambda表达式简化
printString(System.out::println);
```

这个如果你看不懂，我们可以先看一下源码：

我们可以看到out就是一个PrintStream类型的常量

![image-20240412144421658](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/4ec64faf184442ddaaa6cf8f7d548c8d~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1465&h=609&s=64614&e=png&b=212225)

之后我们来看println

![image-20240412144652618](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/aa9c55e5d8874b4095641b86cd2d07fe~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1234&h=623&s=60747&e=png&b=1f2023)

也就是说，这个代码也可以这样写

```
PrintStream out = System.out;
//使用lambda表达式简化
printString(out::println);
```

这样看起来就很直观了。

为什么需要这个方法引用呢?

当lambda表达式所要完成的业务逻辑已经存在 --已经有某个函数实现了

我们可以直接引用对于的方法

底层实现

我们还是进行一个反编译：

![image-20240412144945298](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/e646be18f39a4416ba85f5f30b75ebaa~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1755&h=741&s=449698&e=png&b=fefefe)

我们可以看出来，lambda和方法引用实际上是一回事。

不过lambda是需要我们自己去写方法。

而方法引用是用系统里面自带的方法。

语法格式

方法引用运算符

双冒号 ::

哪些方法可以引用?

* 类方法
* 构造方法
* 实例方法

被引用方法与函数式接口抽象方法需要满足以下条件：

* 参数列表相同

但是这种也是兼容的，属于是特殊情况

![image-20240412150159113](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/e0ad643aead740a39d5cd9d92968a877~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=929&h=299&s=101520&e=png&b=fdfdfd)

* 返回值类型兼容

格式	范例
类名::静态方法	`Integer::parseInt`
类名::new	`Student::new`
对象::成员方法	`”Hello”::toUpperCase` `this::方法名/super::方法名`

方法引用举例

静态方法引用

```
public static void main(String[] args) {
    int number = method(-10, Math::abs);
    System.out.println(number);
}

public static int method(int number, Calcable calcable) {
    return calcable.calsAbs(number);
}
```

为什么这个可以使用呢?

我们来看我们的接口

```
public interface Calcable {
    int calsAbs(int num);
}
```

之后看这个abs

![image-20240412150726020](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/3fdae1e1d9ea48edba1666725a0082e5~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1008&h=272&s=18994&e=png&b=1f2124)

可以发现俩个的返回值是一样的。并且参数也是一样的。所以我们可以使用。

如果我们改为别的类型，例如string

![image-20240412150817862](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/6df6d300669c49b6a8afcb2d0212b631~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1831&h=834&s=104243&e=png&b=202124)

可以发现他就报错了

构造方法引用

```
public interface PersonBuilder {
    Person builderPerson(String name);
}
```

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

```
    printName(”xiaou”, Person::new);
}

public static void printName(String name, PersonBuilder personBuilder) {
    Person person = personBuilder.builderPerson(name);
    System.out.println(person.getName());
}
```

普通方法引用

这个是没有优化的

```
package com.xiaou.demo7;

public class Test {
    public static void main(String[] args) {
        printString(s -> {
            Method method = new Method();
            method.printUpperCassString(s);
        });
    }

    public static void printString(Printable printable) {
        printable.print(”Hello”);
    }
}
```

```
//优化版本 引用某个普通成员的方法 对象名::方法名
public static void main(String[] args) {
    Method method = new Method();
    printString(method::printUpperCassString);
}

public static void printString(Printable printable) {
    printable.print(”Hello”);
}
```

super和this

```
public class Human {
    public void say(){
        System.out.println("xiaou");
    }
}
```

```
public class Man extends Human {
    @Override
    public void say() {
        System.out.println("xiaou say");
    }

    public void method(Greetable g) {
        g.greet();
    }

    public void show() {
        method(() -> {
            Human h = new Human();
            h.say();
        });
    }
}
```

我们需要来简化这个方法。

我们来看一下如何简化

```
public void show() {
    method(super::say);
}
```

```

之后来看this的。

```
public class Husband {
    public void buyHouse() {
        System.out.println("buyhouse");
    }

    public void marry(Richable richable) {
        richable.buy();
    }

    public void soHappy() {
        marry(() -> this.buyHouse());
    }
}

```

可以简化为

```
public void soHappy() {
    marry(this::buyHouse);
}

```

#### 数组的方法引用

```
public static void main(String[] args) {
    int[] arr1 = createArray(10, int[]::new);
}

public static int[] createArray(int size, ArrayBuilder arrayBuilder) {
    return arrayBuilder.builderArray(size);
}

```
```

Stream流

初体验

我们直接来看需求：

查询集合中复合条件的人员

> 查询集合中姓张、并且长度为3的人
>
>
> 并打印出来

我们来看传统的做法。

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("张三");
    list.add("张三丰");
    list.add("张学友");
    list.add("张伟");

    // 遍历List集合
    List<String> listA = new ArrayList<>();
    for (String s : list) {
        if (s.startsWith("张")) {
            listA.add(s);
        }
    }

    //对listA进行处理
    List<String> listB = new ArrayList<>();
    for (String s : listA) {
        if (s.length() == 3) {
            listB.add(s);
        }
    }
```

```
    for (String s : listB) {
        System.out.println(s);
    }
}
```

可以看出来是非常的长的

我们来看stream实现

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("张三");
    list.add("张三丰");
    list.add("张学友");
    list.add("张伟");
    list.stream()
            .filter(name -> name.startsWith("张"))
            .filter(name -> name.length() == 3)
            .forEach(System.out::println);
}
```

stream只**做什么**，不怎么做。

filter也就是过滤。

特点

专注于对容器对象的`聚合操作`

提供串行/并行两种模式 使用Fork/Join框架拆分任务

![image-20240412190954643](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/6f8ebfd32977454987554 58c94658931~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=723&h=489&s=77231&e=png&b=ffffff)

提高编程效率、可读性

使用步骤：

获得流->中间操作->终结操作

![image-20240412190804942](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/b6a61de5c1ea47f3b6c2ea18ff02bf64~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=930&h=254&s=43577&e=png&b=fffefe)

API

中间操作 (Intermediate Operations)

中间操作会返回一个新的流，可以无限制地在一个流上串联起来。这些操作通常不会执行流的计算，而是构建一个流的处理管道。

* `map`: 对流中的每个元素应用一个函数，并将结果作为新流的元素。
* `flatMap`: 类似于`map`，但是可以创建一个流的流，然后将这些流连接起来。
* `filter`: 过滤掉不满足给定谓词的元素。
* `distinct`: 去除流中重复的元素。
* `sorted`: 根据自然顺序或者提供的比较器对流中的元素进行排序。
* `peek`: 查看流中的元素，通常用于调试。
* `limit`: 限制流中元素的数量。
* `skip`: 跳过流中前面的元素，数量由参数指定。
* `parallel`: 将流的并行性设置为并行。
* `sequential`: 将流的并行性设置为顺序。
* `unordered`: 表示流的元素没有特定的顺序。
* `concat`: 将两个流连接起来，形成一个有序的流。

终结操作 (Terminal Operations)

终结操作是流操作的最后步骤，它会生成一个结果或者副作用，并且会消耗流中的元素。

* `forEach`: 对流中的每个元素执行一个操作。

* `forEachOrdered`: 与`forEach`类似，但是保证按照流中元素的遇到顺序执行操作。
* `toArray`: 将流中的元素收集到一个数组中。
* `reduce`: 对流中的元素进行累积操作，得到一个结果。
* `collect`: 收集流中的元素到一个收集器中。
* `min`: 返回流中最小的元素。
* `max`: 返回流中最大的元素。
* `count`: 返回流中元素的数量。
* `iterator`: 返回流中元素的迭代器。
* `anyMatch`: 检查流中是否至少有一个元素满足给定的谓词。
* `allMatch`: 检查流中的所有元素是否都满足给定的谓词。
* `noneMatch`: 检查流中是否没有元素满足给定的谓词。
* `findFirst`: 返回流中的第一个元素。
* `findAny`: 返回流中的任意一个元素。

需要注意的是，我们不能再终结方法之后再去添加中间操作。

因为在执行终结方法后，流就已经执行完了。并且关闭了。

常用API

我们来看一些常用的API

获得流

首先是获取流

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    Stream<String> stream1 = list.stream();

    Set<String> set = new HashSet<>();
    Stream<String> stream2 = set.stream();

    Map<String, String> map = new HashMap<>();
    Set<String> keySet = map.keySet();
    Stream<String> stream3 = keySet.stream();

    Collection<String> values = map.values();
```

```
    Stream<String> stream4 = values.stream();

    Set<Map.Entry<String, String>> entries = map.entrySet();
    Stream<Map.Entry<String, String>> stream5 = entries.stream();


    //把数组转换为Stream流
    Integer[] arr = {1, 2, 3, 4, 5};
    String[] arr2 = {"a", "b", "c"};

    Stream<Integer> stream6 = Stream.of(arr);

    Stream<String> stream7 = Stream.of(arr2);

}
```

foreach

```
public static void main(String[] args) {
    Stream<String> stream = Stream.of("a1", "a2", "b1", "c2", "c1");
    stream.forEach(System.out::println);
}
```

filter

这个就和我们上一个例子是一样的。

map

```
public static void main(String[] args) {
    Stream<String> stream = Stream.of("1", "2", "3", "4", "5");
    Stream<Integer> stream2 = stream.map(Integer::parseInt);
    stream2.forEach(System.out::println);
}
```

count

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);

    long count = list.stream().count();
    System.out.println(count);
}
```

collect

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("张三");
    list.add("李四");
    list.add("王五五");

    Stream<String> stream = list.stream().filter(s -> s.length() > 2);

    List<String> collect = stream.collect(Collectors.toList());
    System.out.println(collect);

    Set<Integer> set = new HashSet<>();
    set.add(10);
    set.add(20);
    set.add(30);
    set.add(50);
    Stream<Integer> integerStream = set.stream().filter(s -> s > 20);
    Set<Integer> collect1 = integerStream.collect(Collectors.toSet());
    System.out.println(collect1);

    String[] strArray = {"张三,30", "李四,35", "王五,20"};
    Stream<String> stringStream = Stream.of(strArray).filter(s ->
Integer.parseInt(s.split(",")[1]) > 28);
```

```
    Map<String, Integer> collect2 =
stringStream.collect(Collectors.toMap(
            s -> s.split(",")[0],
            s -> Integer.parseInt(s.split(",")[1])
    ));
    System.out.println(collect2);
}

```

skip

```
public static void main(String[] args) {
    String[] arr = {"a", "b", "c", "d", "e", "f", "g", "h"};
    Stream<String> arr1 = Stream.of(arr);
    //跳过前三个元素
    arr1.skip(3).forEach(System.out::println);
}

```

原文链接: https://juejin.cn/post/7356618290469109814