原来 Lamda 表达式是这样写的

程序员小灰 今天

以下文章来源于低并发编程 ， 作者闪客sun

低并发编程

战略上藐视技术，战术上重视技术

Lamda 表达式非常方便，在项目中一般在 stream 编程中用的比较多。

```
List<Student> studentList = gen();
Map<String, Student> map = studentList .stream()
        .collect(Collectors.toMap(Student::getId, a -> a, (a, b) -> a));
```

理解一个 Lamda 表达式就三步：

1. 确认 Lamda 表达式的类型
2. 找到要实现的方法
3. 实现这个方法

就这三步，没其他的了。而每一步，都非常非常简单，以至于我分别展开讲一下，你就懂了。

确认 Lamda 表达式的类型

能用 Lamda 表达式来表示的类型，必须是一个**函数式接口**，而函数式接口，就是只有一个抽象方法的接口。

我们看下非常熟悉的 Runnable 接口在 JDK 中的样子就明白了。

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

这就是一个标准的函数式接口。

因为只有一个抽象方法。而且这个接口上有个注解

@FunctionalInterface

这个仅仅是在**编译期**帮你检查你这个接口是否符合函数式接口的条件，比如你没有任何抽象方法，或者有多个抽象方法，编译是无法通过的。

```
// 没有实现任何抽象方法的接口
@FunctionalInterface
public interface MyRunnable {}

// 编译后控制台显示如下信息
Error:(3, 1) java:
  意外的 @FunctionalInterface 注释
  MyRunnable 不是函数接口
    在 接口 MyRunnable 中找不到抽象方法
```

再稍稍复杂一点，Java 8 之后接口中是允许使用**默认方法**和**静态方法**的，而这些都不算抽象方法，所以也可以加在函数式接口里。

看看你可能不太熟悉又有点似曾相识的一个接口。

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {...}
}
```

看，只有一个抽象方法，还有一个默认方法（方法体的代码省略了），这个也不影响它是个函数式接口。再看一个更复杂的，多了静态方法，这同样也是个函数式接口，因为它仍然只有一个抽象方法。自行体会吧。

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

    default Predicate<T> and(Predicate<? super T> other) {...}
    default Predicate<T> negate() {...}
    default Predicate<T> or(Predicate<? super T> other) {...}

    static <T> Predicate<T> isEqual(Object targetRef) {...}
    static <T> Predicate<T> not(Predicate<? super T> target) {...}
}
```

先不用管这些方法都是干嘛的，这些类在 Stream 设计的方法中比比皆是，我们就先记住这么一句话，**Lamda 表达式需要的类型为函数式接口，函数式接口里只有一个抽象方法**，就够了，以上三个例子都属于函数式接口。

恭喜你，已经学会了 Lamda 表达式最难的部分，就是认识函数式接口。

找到要实现的方法

Lamda 表达式就是实现一个方法，什么方法呢？就是刚刚那些函数式接口中的**抽象方法。**

那就太简单了，因为函数式接口有且只有一个抽象方法，找到它就行了。我们尝试把刚刚那几个函数式接口的抽象方法找到。

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {...}
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) {...}
    default Predicate<T> negate() {...}
    default Predicate<T> or(Predicate<? super T> other) {...}
    static <T> Predicate<T> isEqual(Object targetRef) {...}
    static <T> Predicate<T> not(Predicate<? super T> target) {...}
}
```

好了，这就找到了，简单吧！

实现这个方法

Lamda 表达式就是要实现这个抽象方法，如果不用 Lamda 表达式，你一定知道用**匿名类**如何去实现吧？比如我们实现刚刚 Predicate 接口的匿名类。

```
Predicate<String> predicate = new Predicate<String>() {
    @Override
    public boolean test(String s) {
        return s.length() != 0;
```

```
    }
};
```

那如果换成 Lamda 表达式呢？就像这样。

```
Predicate<String> predicate =
    (String s) -> {
        return s.length() ! = 0;
    };
```

看出来了么？这个 Lamda 语法由三部分组成：

参数块：就是前面的 (String s)，就是简单地把要实现的抽象方法的参数原封不动写在这。

小箭头：就是 -> 这个符号。

代码块：就是要实现的方法原封不动写在这。

不过这样的写法你一定不熟悉，连 idea 都不会帮我们简化成这个奇奇怪怪的样子，别急，我要变形了！其实是对其进行格式上的简化。

首先看参数快部分，(String s) 里面的类型信息是多余的，因为完全可以由编译器推导，去掉它。

```
Predicate<String> predicate =
    (s) -> {
        return s.length() ! = 0;
    };
```

当只有一个参数时，括号也可以去掉。

```
Predicate<String> predicate =
    s -> {
        return s.length() ! = 0;
    };
```

再看代码块部分，方法体中只有一行代码，可以把花括号和 return 关键字都去掉。

```
Predicate<String> p = s -> s.length() != 0;
```

这样看是不是就熟悉点了？

来，再让我们实现一个 Runnable 接口。

```
@FunctionalInterface
```

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

Runnable r = () -> System.out.println("I am running");
```

你看，这个方法没有入参，所以前面括号里的参数就没有了，这种情况下括号就不能省略。

通常我们快速新建一个线程并启动时，是不是像如下的写法，熟悉吧？

```
new Thread(() -> System.out.println("I am running")).start();
```

多个入参

之前我们只尝试了一个入参，接下来我们看看多个入参的。

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
    // default methods removed
}
```

然后看看一个用法，是不是一目了然。

```
BiConsumer<Random, Integer> randomNumberPrinter =
        (random, number) -> {
            for (int i = 0; i < number; i++) {
                System.out.println("next random = " + random.nextInt());
            }
        };

randomNumberPrinter.accept(new Random(314L), 5));
```

刚刚只是多个入参，那我们再加个返回值。

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    // default methods removed
}
```

```
// 看个例子
BiFunction<String, String, Integer> findWordInSentence =
    (word, sentence) -> sentence.indexOf(word);
```

发现规律了没

OK，看了这么多例子，不知道你发现规律了没？

其实函数式接口里那个抽象方法，无非就是**入参的个数**，以及**返回值的类型**。入参的个数可以是一个或者两个，返回值可以是 void，或者 boolean，或者一个类型。那这些种情况的排列组合，就是 JDK 给我们提供的**java.util.function**包下的类。

BiConsumer
BiFunction
BinaryOperator
BiPredicate
BooleanSupplier
Consumer
DoubleBinaryOperator
DoubleConsumer
DoubleFunction
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function
IntBinaryOperator
IntConsumer
IntFunction
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction
LongPredicate
LongSupplier
LongToDoubleFunction

LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer
ObjIntConsumer
ObjLongConsumer
Predicate
Supplier
ToDoubleBiFunction
ToDoubleFunction
ToIntBiFunction
ToIntFunction
ToLongBiFunction
ToLongFunction
UnaryOperator

别看晕了，我们分分类就好了。可以注意到很多类前缀是 Int，Long，Double 之类的，这其实是指定了入参的特定类型，而不再是一个可以由用户自定义的泛型，比如说 DoubleFunction。

```
@FunctionalInterface
public interface DoubleFunction<R> {
    R apply(double value);
}
```

这完全可以由更自由的函数式接口 Function 来实现。

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

那我们不妨先把这些特定类型的函数式接口去掉（我还偷偷去掉了 XXXOperator 的几个类，因为它们都是继承了别的函数式接口），然后再排排序，看看还剩点啥。

Consumer
Function
Predicate

BiConsumer
BiFunction
BiPredicate

Supplier

哇塞，几乎全没了，接下来就重点看看这些。这里我就只把类和对应的抽象方法列举出来

Consumer void accept(T t)
Function R apply(T t)
Predicate boolean test(T t)

BiConsumer void accept(T t, U u)
BiFunction R apply(T t, U u)
BiPredicate boolean test(T t, U u)

Supplier T get()

看出规律了没？上面那几个简单分类就是：

supplier：没有入参，有返回值。
consumer：有入参，无返回值。
predicate：有入参，返回 boolean 值
function：有入参，有返回值

然后带 Bi 前缀的，就是有两个入参，不带的就只有一个如参。OK，这些已经被我们分的一清二楚了，其实就是给我们提供了一个函数的模板，区别仅仅是入参返参个数的排列组合。

用我们常见的 Stream 编程熟悉一下

下面这段代码如果你项目中有用 stream 编程那肯定很熟悉，有一个 Student 的 list，你想把它转换成一个 map，key 是 student 对象的 id，value 就是 student 对象本身。

```
List<Student> studentList = gen();
Map<String, Student> map = studentList .stream()
        .collect(Collectors.toMap(Student::getId, a -> a, (a, b) -> a));
```

把 Lamda 表达式的部分提取出来。

```
Collectors.toMap(Student::getId, a -> a, (a, b) -> a)
```

由于我们还没见过 :: 这种形式，先打回原样，这里只是让你预热一下。

```
Collectors.toMap(a -> a.getId(), a -> a, (a, b) -> a)
```

为什么它被写成这个样子呢？我们看下 Collectors.toMap 这个方法的定义就明白了。

```
public static <T, K, U> Collector<T, ?, Map<K,U>> toMap(
        Function<? super T, ? extends K> keyMapper,
```

```
        Function<? super T, ? extends U> valueMapper,
        BinaryOperator<U> mergeFunction)
{
    return toMap(keyMapper, valueMapper, mergeFunction, HashMap::new);
}
```

看，入参有三个，分别是**Function**，Function，**BinaryOperator**，其中 BinaryOperator 只是继承了 BiFunction 并扩展了几个方法，我们没有用到，所以不妨就把它当做**BiFunction**。

还记得 Function 和 BiFunction 吧?

Function R apply(T t)
BiFunction R apply(T t, U u)

那就很容易理解了。

第一个参数**a -> a.getId()**就是 R apply(T t) 的实现，入参是 Student 类型的对象 a，返回 a.getId()

第二个参数**a -> a**也是 R apply(T t) 的实现，入参是 Student 类型的 a，返回 a 本身

第三个参数**(a, b) -> a**是 R apply(T t, U u) 的实现，入参是Student 类型的 a 和 b，返回是第一个入参 a，Stream 里把它用作当两个对象 a 和 b 的 key 相同时，value 就取第一个元素 a

其中第二个参数 a -> a 在 Stream 里表示从 list 转为 map 时的 value 值，就用原来的对象自己，你肯定还见过这样的写法。

```
Collectors.toMap(a -> a.getId(), Function.identity(), (a, b) -> a)
```

为什么可以这样写，给你看 Function 类的全貌你就明白了。

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ...
    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

看到了吧，identity 这个方法，就是帮我们把表达式给实现了，就不用我们自己写了，其实就是包了个方法。这回知道一个函数式接口，为什么有好多还要包含一堆默认方法和静态方法了吧？就是干这个事用的。

我们再来试一个，Predicate 里面有这样一个默认方法。

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
}
```

它能干嘛用呢？我来告诉你，如果没有这个方法，有一段代码你可能会这样写。

```
Predicate<String> p =
    s -> (s != null) &&
    !s.isEmpty() &&
    s.length() < 5;
```

如果利用上这个方法，就可以变成如下这种优雅形式。

```
Predicate<String> nonNull = s -> s != null;
Predicate<String> nonEmpty = s -> s.isEmpty();
Predicate<String> shorterThan5 = s -> s.length() < 5;

Predicate<String> p = nonNull.and(nonEmpty).and(shorterThan5);
```

自行体会吧。

方法引用

那我们回过头再看刚刚的 Student::getId 这种写法。当方法体中只有一个方法调用时，就可以作这样的简化。

比如这个 a -> a.getId() 就只是对 Student 对象上 getId() 这个方法的调用，那么就可以写成 Student::getId 这种形式。

再看几个例子

```
Function<String, Integer> toLength = s -> s.length();
Function<String, Integer> toLength = String::length;

Function<User, String> getName = user -> user.getName();
Function<String, Integer> toLength = User::getName;

Consumer<String> printer = s -> System.out.println(s);
Consumer<String> printer = System.out::println;
```

如果是构造方法的话，也可以简化。

```
Supplier<List<String>> newListOfStrings = () -> new ArrayList<>();
Supplier<List<String>> newListOfStrings = ArrayList::new;
```

总结

学会理解和写 Lamda 表达式，别忘了最开始的三步。

1. **确认 Lamda 表达式的类型**
2. **找到要实现的方法**
3. **实现这个方法**

Lamda 表达式的类型就是**函数式接口**，要实现的方法就是函数式接口里那个唯一的**抽象方法**，实现这个方法的方式就是**参数块 + 小箭头 + 方法体**，其中参数块和方法体都可以一定程度上简化它的写法。

是不是很简单了！

以上代码例子，都来源于官方的教程，英语好的同学可以看看，是最科学的 Lamda 表达式教程了。

https://dev.java/learn/tutorial/getting-to-know-the-language/lambda-expressions/lambdas.html

的今天的文章主要就是讲怎么写出 Lamda 表达式，至于原理，之后再说。这里提个引子，**你觉得 Lamda 表达式是匿名类的简化么**？按照官方的说法，Lamda 表达式在某些情况下就是匿名类的一种更简单的写法，但是从字节码层面看，完全不同，这又是怎么回事呢？

带着这个问题，给自己埋个坑，我们下讲说说 Lamda 表达式背后的实现原理。

程序员小灰

一群喜爱编程技术和算法的小仓鼠。

397篇原创内容