先贩卖一下焦虑，Java8发于2014年3月18日，距离现在已经快6年了，如果你对Java8的新特性还没有应用，甚至还一无所知，那你真得关注公众号“程序新视界”，好好系列的学习一下Java8的新特性。Lambda表达式已经在新框架中普通使用了，如果你对Lambda还一无所知，真得认真学习一下本篇文章了。

现在进入正题Java8的Lambda，首先看一下发音 (['læmdə])表达式。注意该词的发音，b是不发音的，da发[də]音。

为什么要引入Lambda表达式

简单的来说，引入Lambda就是为了简化代码，允许把函数作为一个方法的参数传递进方法中。如果有JavaScript的编程经验，马上会想到这不就是闭包吗。是的，Lambda表达式也可以称作Java中的闭包。

先回顾一下Java8以前，如果想把某个接口的实现类作为参数传递给一个方法会怎么做？要么创建一个类实现该接口，然后new出一个对象，在调用方法时传递进去，要么使用匿名类，可以精简一些代码。以创建一个线程并打印一行日志为例，使用匿名函数写法如下：

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("欢迎关注公众号：程序新视界");
    }
}).start();
```

在java8以前，使用匿名函数已经算是很简洁的写法了，再来看看使用Lambda表达式，上面的代码会变成什么样子。

```
new Thread(() -> System.out.println("欢迎关注公众号：程序新视
界")).start();
```

是不是简洁到爆！

我们都知道java是面向对象的编程语言，除了部分简单数据类型，万物皆对象。因此，在Java中定义函数或方法都离不开对象，也就意味着很难直接将方法或函数像参数一样传递，而Java8中的Lambda表达式的出现解决了这个问题。

Lambda表达式使得Java拥有了函数式编程的能力，但在Java中Lambda表达式是对象，它必须依附于一类特别的对象类型——函数式接口(functional interface)，后面详细讲解。

Lambda表达式简介

Lambda表达式是一种匿名函数(对Java而言这并不完全准确)，通俗的说，它是没有声明的方法，即没有访问修饰符、返回值声明和名字的方法。使用Lambda表达式的好处很明显就是可以使代码变的更加简洁紧凑。

Lambda表达式的使用场景与匿名类的使用场景几乎一致，都是在某个功能（方法）只使用一次的时候。

Lambda表达式语法结构

Lambda表达式通常使用(param)->(body)语法书写，基本格式如下：

```
//没有参数
() -> body

// 1个参数
(param) -> body
// 或
(param) ->{ body; }

// 多个参数
(param1, param2...) -> { body }
// 或
(type1 param1, type2 param2...) -> { body }
```

常见的Lambda表达式如下：

```
// 无参数,返回值为字符串“公众号：程序新视界”
() -> "公众号：程序新视界";

// 1个String参数，直接打印结果
(System.out::println);
// 或
(String s) -> System.out.print(s)

// 1个参数(数字),返回2倍值
x -> 2 * x;

// 2个参数(数字),返回差值
(x, y) -> x – y

// 2个int型整数,返回和值
(int x, int y) -> x + y
```

对照上面的示例，我们再总结一下Lambda表达式的结构：

- Lambda表达式可以有0~n个参数。
- 参数类型可以显式声明，也可以让编译器从上下文自动推断类型。如(int x)和(x)是等价的。
- 多个参数用小括号括起来，逗号分隔。一个参数可以不用括号。
- 没有参数用空括号表示。
- Lambda表达式的正文可以包含零条，一条或多条语句，如果有返回值则必须包含返回值语句。如果只有一条可省略大括号。如果有一条以上则必须包含在大括号（代码块）中。

函数式接口

函数式接口(Functional Interface)是Java8对一类特殊类型的接口的称呼。这类接口只定义了唯一的**抽象方法**的接口（除了隐含的Object对象的公共方法），因此最开始也就做SAM类型的接口（Single Abstract Method）。

比如上面示例中的java.lang.Runnable就是一种函数式接口，在其内部只定义了一个void run()的抽象方法，同时在该接口上注解了@FunctionalInterface。

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

@FunctionalInterface注解是用来表示该接口要符合函数式接口的规范，除了隐含的Object对象的公共方法以外只可有一个抽象方法。当然，如果某个接口只定义一个抽象方法，不使用该注解也是可以使用Lambda表达式的，但是没有该注解的约束，后期可能会新增其他的抽象方法，导致已经使用Lambda表达式的地方出错。使用@FunctionalInterface从编译层面解决了可能的错误。

比如当注解@FunctionalInterface之后，写两个抽象方法在接口内，会出现以下提示：

```
Multiple non-overriding abstract methods found in interface
com.secbro2.lambda.NoParamInterface
```

通过函数式接口我们也可以得出一个简单的结论：可使用Lambda表达式的接口，只能有一个抽象方法（除了隐含的Object对象的公共方法）。

注意此处的方法限制为抽象方法，如果接口内有其他静态方法则不会受限制。

方法引用，双冒号操作

[方法引用]的格式是，类名::方法名。

像如ClassName::methodName或者objectName::methodName的表达式，我们把它叫做方法引用（Method Reference）,通常用在Lambda表达中。

看一下示例：

```
// 无参数情况
NoParamInterface paramInterface2 = ()-> new HashMap<>();
// 可替换为
NoParamInterface paramInterface1 = HashMap::new;

// 一个参数情况
OneParamInterface oneParamInterface1 = (String string) ->
System.out.print(string);
// 可替换为
OneParamInterface oneParamInterface2 = (System.out::println);

// 两个参数情况
Comparator c = (Computer c1, Computer c2) ->
c1.getAge().compareTo(c2.getAge());
// 可替换为
Comparator c = (c1, c2) -> c1.getAge().compareTo(c2.getAge());
// 进一步可替换为
Comparator c = Comparator.comparing(Computer::getAge);
```

再比如我们用函数式接口java.util.function.Function来实现一个String转Integer的功能，可以如下写法：

```
Function<String, Integer> function = Integer::parseInt;
Integer num = function.apply("1");
```

根据Function接口的定义Function<T,R>，其中T表示传入类型，R表示返回类型。具体就是实现了Function的apply方法，在其方法内调用了Integer.parseInt方法。

通过上面的讲解，基本的语法已经完成，以下内容通过实例来逐一演示在不同的场景下如何使用。

Runnable线程初始化示例

Runnable线程初始化是比较典型的应用场景。

```
// 匿名函类写法
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("欢迎关注公众号：程序新视界");
    }
}).start();

// lambda表达式写法
new Thread(() -> System.out.println("欢迎关注公众号：程序新视
界")).start();

// lambda表达式 如果方法体内有多行代码需要带大括号
new Thread(() -> {
    System.out.println("欢迎关注公众号");
    System.out.println("程序新视界");
}).start();
```

通常都会把lambda表达式内部变量的名字起得短一些，这样能使代码更简短。

事件处理示例

Swing API编程中经常会用到的事件监听。

```
// 匿名函类写法
JButton follow =  new JButton("关注");
follow.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("已关注公众号：程序新视界");
    }
});

// lambda表达式写法
follow.addActionListener((e) -> System.out.println("已关注公众号：程序新视界"));

// lambda表达式写法
follow.addActionListener((e) -> {
    System.out.println("已关注公众号");
    System.out.println("程序新视界");
});
```

列表遍历输出示例

传统遍历一个List，基本上都使用for循环来遍历，Java8之后List拥有了forEach方法，可配合lambda表达式写出更加简洁的方法。

```
List<String> list = Arrays.asList("欢迎","关注","程序新视界");

// 传统遍历
for(String str : list){
    System.out.println(str);
}

// lambda表达式写法
list.forEach(str -> System.out.println(str));
// lambda表达式写法
list.forEach(System.out::println);
```

函数式接口示例

在上面的例子中已经看到函数式接口java.util.function.Function的使用，在java.util.function包下中还有其他的类，用来支持Java的函数式编程。比如通过Predicate函数式接口以及lambda表达式，可以向API方法添加逻辑，用更少的代码支持更多的动态行为。

```
@Test
public void testPredicate() {

    List<String> list = Arrays.asList("欢迎", "关注", "程序新视界");

    filter(list, (str) -> ("程序新视界".equals(str)));

    filter(list, (str) -> (((String) str).length() == 5));

}

public static void filter(List<String> list, Predicate condition) {
    for (String content : list) {
        if (condition.test(content)) {
            System.out.println("符合条件的内容：" + content);
        }
    }
}
```

其中filter方法中的写法还可以进一步简化：

```
list.stream().filter((content) ->
condition.test(content)).forEach((content) ->System.out.println("符
合条件的内容：" + content));

list.stream().filter(condition::test).forEach((content) -
>System.out.println("符合条件的内容：" + content));

list.stream().filter(condition).forEach((content) -
>System.out.println("符合条件的内容：" + content));
```

如果不需要“符合条件的内容:”字符串的拼接，还能够进一步简化：

```
list.stream().filter(condition).forEach(System.out::println);
```

如果将调用filter方法的判断条件也写在一起，test方法中的内容可以通过一行代码来实现：

```
list.stream().filter((str) -> ("程序新视界".equals(str))).forEach(System.out::println);
```

如果需要同时满足两个条件或满足其中一个即可，Predicate可以将这样的多个条件合并成一个。

```
Predicate start = (str) -> (((String) str).startsWith("程序"));
Predicate len = (str) -> (((String) str).length() == 5);

list.stream().filter(start.and(len)).forEach(System.out::println);
```

Stream相关示例

在《JAVA8 STREAM新特性详解及实战》一文中已经讲解了Stream的使用。你是否发现Stream的使用都离不开Lambda表达式。是的，所有Stream的操作必须以Lambda表达式为参数。

以Stream的map方法为例：

```
Stream.of("a","b","c").map(item -> item.toUpperCase()).forEach(System.out::println);
Stream.of("a","b","c").map(String::toUpperCase).forEach(System.out::println);
```

更多的使用实例可参看Stream的《JAVA8 STREAM新特性详解及实战》一文。

Lambda表达式与匿名类的区别

- 关键词的区别：对于匿名类，关键词this指向匿名类，而对于Lambda表达式，关键词this指向包围Lambda表达式的类的外部类，也就是说跟表达式外面使用this表达的意思是一样。
- 编译方式：Java编译器编译Lambda表达式时，会将其转换为类的私有方法，再进行动态绑定，通过invokedynamic指令进行调用。而匿名内部类仍然是一个类，编译时编译器会自动为该类取名并生成class文件。

其中第一条，以Spring Boot中ServletWebServerApplicationContext类的一段源码作为示例：

```
private
org.springframework.boot.web.servlet.ServletContextInitializer
getSelfInitializer() {
   return this::selfInitialize;
}

private void selfInitialize(ServletContext servletContext) throws
ServletException {
   prepareWebApplicationContext(servletContext);
   registerApplicationScope(servletContext);

WebApplicationContextUtils.registerEnvironmentBeans(getBeanFactory(
),servletContext);
   for (ServletContextInitializer beans :
getServletContextInitializerBeans()) {
      beans.onStartup(servletContext);
   }
}
```

其中，这里的this指向的就是getSelfInitializer方法所在的类。

小结

至此，Java8 Lambda表达式的基本使用已经讲解完毕，最关键的还是要勤加练习，达到熟能生巧的使用。当然，刚开始可能需要一个适应期，在此期间可以把本篇文章收藏当做一个手册拿来参考。

原文链接：《Java8 Lambda表达式详解手册及实例》