Angrily Learn Java 8

(恕学 Java 8)

by bjpengpeng

Mail:bjpengpeng@corp.netease.com

Github:https://github.com/zwjlpeng/Angrily_Learn_Java_8

第一章　Lambda

1.1 引言

课本上说编程有两种模式，面向过程的编程以及面向对象的编程，其实在面向对象编程之前还出现了面向函数的编程(函数式编程) ，以前一直被忽略、不被重视，现在从学术界已经走向了商业界，对函数编程语言的支持目前有 Scala、Erlang、F#、Python、Php、Java、Javascript 等，有人说他将会是编程语言中的下一个主流...

1.2 Lambda 表达式

为什么需要 Lambda 表达式?

1.使用 Lambda 表达式可以使代码变的更加紧凑，例如在 Java 中实现一个线程，只输出一个字符串 Hello World!，我们的代码如下所示：

```
public static void main(String[] args) throws Exception {
    new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("Hello World!");
        }
    }).start();
    TimeUnit.SECONDS.sleep(1000);
}
```

使用 Lambda 表达式之后代码变成如下形式:

```
public static void main(String[] args) throws Exception {
    new Thread(() -> System.out.println("Hello World!")).start();
    TimeUnit.SECONDS.sleep(1000);
}
```

是不是代码变的更紧凑了~，其他的例如各种监听器，以及事件处理器等都可以用这种方式进行简化。

2.修改方法的能力，其实说白了，就是函数中可以接受以函数为单元的参数，在 C/C++中就是函数指针，在 Java 中就是 Lambda 表达式，例如在 Java 中使用集合类对一个字符串按字典序列进行排序，代码如下所示：

```
public static void main(String[] args) {
    String []datas = new String[] {"peng","zhao","li"};
    Arrays.sort(datas);
    Stream.of(datas).forEach(param ->    System.out.println(param));
}
```

在上面代码中用了 Arrays 里的 sort 方法，现在我们不需要按字典排序，而是按字符串的长度进行排序，代码如下所示：

```
public static void main(String[] args) {
    String []datas = new String[] {"peng","zhao","li"};
    Arrays.sort(datas,(v1 , v2) -> Integer.compare(v1.length(),
v2.length()));
    Stream.of(datas).forEach(param -> System.out.println(param));
```

}
是不是很方便，我们不需要实现 Comparable 接口，使用一个 Lambda 表达式就可以改变一个函数的形为~

1.3 Syntax

1.Lambda 表达式的形式化表示如下所示

```
Parameters -> an expression
```

2.如果 Lambda 表达式中要执行多个语句块,需要将多个语句块以{}进行包装，如果有返回值，需要显示指定 return 语句，如下所示:

```
Parameters -> {expressions;};
```

3.如果 Lambda 表达式不需要参数，可以使用一个空括号表示，如下示例所示

```
() -> {for (int i = 0; i < 1000; i++) doSomething();};
```

4.Java 是一个强类型的语言，因此参数必须要有类型，如果编译器能够推测出 Lambda 表达式的参数类型，则不需要我们显示的进行指定，如下所示，在 Java 中推测 Lambda 表达式的参数类型与推测泛型类型的方法基本类似，至于 Java 是如何处理泛型的，此处略去

```
String []datas = new String[] {"peng","zhao","li"};
Arrays.sort(datas,(String v1, String v2) -> Integer.compare(v1.length(),
v2.length()));
```

上述代码中 显示指定了参数类型 Stirng，其实不指定，如下代码所示，也是可以的，因为编译器会根据 Lambda 表达式对应的函数式接口 Comparator<String>进行自动推断

```
String []datas = new String[] {"peng","zhao","li"};;
Arrays.sort(datas,(v1, v2) -> Integer.compare(v1.length(), v2.length()));
```

5.如果 Lambda 表达式只有一个参数，并且参数的类型是可以由编译器推断出来的，则可以如下所示使用 Lambda 表达式,即可以省略参数的类型及括号

```
Stream.of(datas).forEach(param -> {System.out.println(param.length());});
```

6.Lambda 表达式的返回类型，无需指定，编译器会自行推断，说是自行推断

7.Lambda 表达式的参数可以使用修饰符及注解，如 final、@NonNull 等

1.4 函数式接口

函数式接口是 Java 8 为支持 Lambda 表达式新发明的，在上面讲述的 Lambda Syntax 时提到的 sort 排序方法就是一个样例，在这个排序方法中就使用了一个函数式接口，函数的原型声明如下所示

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

上面代码中 Comparator<? Super T>就是一个函数式接口，? Super T or ? entends T 从 Java 5 支持泛型时开始引入，得理解清楚，在此忽略讲述

什么是函数式接口

1.函数式接口具有两个主要特征，是一个接口，这个接口具有唯一的一个抽像方法，我们将满足这两个特性的接口称为函数式接口，说到这，就不得不说一下接口中是有具体实现这个问题啦~

2.Lambda 表达式不能脱离目标类型存在，这个目录类型就是函数式接口，所下所示是一个样例：

```
String []datas = new String[] {"peng","zhao","li"};
Comparator<String> comp = (v1,v2) -> Integer.compare(v1.length(),
v2.length());
```

```
Arrays.sort(datas,comp);
Stream.of(datas).forEach(param -> {System.out.println(param);});
```

Lambda 表达式被赋值给了 comp 函数接口变量

3.函数式接口可以使用@FunctionalInterface 进行标注，使用这个标注后，主要有两个优势，编译器知道这是一个函数式接口，符合函数式的要求，另一个就是生成 Java Doc 时会进行显式标注

4.异常，如果 Lambda 表达式会抛出非运行时异常，则函数式接口也需要抛出异常，说白了，还是一句话，函数式接口是 Lambda 表达式的目标类型

5.函数式接口中可以定义 public static 方法，想想在 Java 中我们提供了 Collection 接口，同时还提供了一个 Collections 工具类等等，在 Java 中将这种 Collections 的实现转移到了接口里面，但是为了保证向后兼容性，以前的这种 Collection/Collections 等逻辑均未改变

6.函数式接口可以提供多个抽像方法，纳尼！上面不是说只能有一个嘛?是的，在函数式接口中可以提供多个抽像方法，但这些抽像方法限制了范围，只能是 Object 类型里的已有方法，为什么要这样做呢?此处忽略，大家可以自己研究

7.函数式接口里面可以定义方法的默认实现，如下所示是 Predicate 类的代码,不仅可以提供一个 default 实现，而且可以提供多个 default 实现呢，Java 8 以前可以嘛?我和我的小伙伴们都惊呆了，这也就导致出现了多继承下的问题，想知道 Java 8 是如何对其进行处理的嘛，其实很 Easy，后面我会再讲~

8.为什么要提供 default 接口的实现？如下就是一个默认实现

```
default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}
```

Java 8 中在接口中增加了默认实现这种函数，其实在很大程序上违背了接口具有抽象这种特征的，增加 default 实现主要原因是因为考虑兼容及代码的更改成本，例如，在 Java 8 中向 iterator 这种接口增加一个方法，那么实现这个接口的所有类都要需实现一遍这个方法，那么 Java 8 需要更改的类就太多的，因此在 Iterator 接口里增加一个 default 实现，那么实现这个接口的所有类就都具有了这种实现，说白了，就是一个模板设计模式吧

1.5 方法引用

有时，我们需要执行的代码在某些类中已经存在，这时我们没必要再去写 Lambda 表达式，可以直接使用该方法，这种情况我们称之为方法引用，如下所示，未采用方法引用前的代码

如下所示

```
Stream.of(datas).forEach(param -> {System.out.println(param);});
```

使用方法引用后的代码如下所示

```
Stream.of(datas).forEach(System.out::println);
```

以上示例使用的是 out 对象，下面示例使用的是类的静态方法引用对字符串数组里的元素忽略大小写进行排序

```
String []datas = new String[] {"peng","Zhao","li"};
Arrays.sort(datas,String::compareToIgnoreCase);
Stream.of(datas).forEach(System.out::println);
```

上面就是方法引用的一些典型示例

方法引用的具体分类

- `Object:instanceMethod`
- `Class:staticMethod`
- `Class:instanceMethod`

上面分类中前两种在 Lambda 表达式的意义上等同，都是将参数传递给方法，如上示例

```
System.out::println == x -> System.out.println(x)
```

最后一种分类，第一个参数是方法执行的目标，如下示例

```
String::compareToIgnoreCase == (x,y) -> x.compareToIgnoreCase(y)
```

还有类似于 super::instanceMethod 这种方法引用本质上与 Object::instanceMethod 类似

1.6 构造方法引用

构造方法引用与方法引用类似，除了一点，就是构造方法引用的方法是 new!以下是两个示例

示例一：

```
String str = "test";
Stream.of(str).map(String::new).peek(System.out::println).findFirst();
```

示例二：

```
String []copyDatas = Stream.of(datas).toArray(String[]::new);
Stream.of(copyDatas).forEach(x -> System.out.println(x));
```

总结一下，构造方法引用有两种形式

- `Class::new`
- `Class[]::new`

1.7 Lambda 表达式作用域

总体来说，Lambda 表达式的变量作用域与内部类非常相似，只是条件相对来说，放宽了些以前内部类要想引用外部类的变量，必须像下面这样

```
final String[] datas = new String[] { "peng", "Zhao", "li" };
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println(datas);
    }
}).start();
```

将变量声明为 final 类型的，现在在 Java 8 中可以这样写代码

```
String []datas = new String[] {"peng","Zhao","li"};
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println(datas);
    }
}).start();
```

也可以这样写

```
new Thread(() -> System.out.println(datas)).start();
```

总之你爱怎么写，就怎么写吧,I don’t Care it!

看了上面的两段代码，能够发现一个显著的不同，就是 Java 8 中内部类或者 Lambda 表达式对外部类变量的引用条件放松了，不要求强制的加上 final 关键字了，但是 Java 8 中要求这个变量是 effectively final

What is effectively final?

Effectively final 就是有效只读变量，意思是这个变量可以不加 final 关键字，但是这个变量必须是只读变量，即一旦定义后，在后面就不能再随意修改，如下代码会编译出错

```
String []datas = new String[] {"peng","Zhao","li"};
datas = null;
new Thread(() -> System.out.println(datas)).start();
```

Java 中内部类以及 Lambda 表达式中也不允许修改外部类中的变量，这是为了避免多线程情况下的 race condition

Lambda 中变量以及 this 关键字

Lambda 中定义的变量与外部类中的变量作用域相同，即外部类中定义了，Lambda 就不能再重复定义了，同时在 Lambda 表达式使用的 this 关键字，指向的是外部类,大家可以自行实践下，此处略

°