Java的函数式编程(Lambda表达式)

Java平台从**Java 8**开始，支持函数式编程

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，**只要输入是确定的，输出就是确定的**，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

**函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！**

在了解Lambda之前，我们先回顾一下Java的方法。

Java的方法分为实例方法，例如Integer定义的equals()方法：

public final class Integer {

boolean equals(Object o) {

...

}

}

以及静态方法，例如Integer定义的parseInt()方法：

public final class Integer {

public static int parseInt(String s) {

...

}

}

无论是实例方法，还是静态方法，本质上都相当于过程式语言的函数。例如C函数：

char\* strcpy(char\* dest, char\* src)

**只不过Java的实例方法隐含地传入了一个this变量，即实例方法总是有一个隐含参数this。**

**函数式编程（Functional Programming）是把函数作为基本运算单元，函数可以作为变量，可以接收函数，还可以返回函数。历史上研究函数式编程的理论是Lambda演算，所以我们经常把支持函数式编程的编码风格称为Lambda表达式。**

### Lambda表达式

在Java程序中，我们经常遇到一大堆单方法接口，即一个接口只定义了一个方法：

* Comparator
* Runnable
* Callable

以Comparator为例，我们想要调用Arrays.sort()时，可以传入一个Comparator实例，以匿名类方式编写如下：

String[] array = ...

Arrays.sort(array, new Comparator<String>() {

public int compare(String s1, String s2) {

return s1.compareTo(s2);

}

});

上述写法非常繁琐。从Java 8开始，我们可以用Lambda表达式替换单方法接口。改写上述代码如下：

Arrays.sort(array, (s1, s2) -> {

return s1.compareTo(s2);

});

观察Lambda表达式的写法，它只需要写出方法定义：

(s1, s2) -> {

return s1.compareTo(s2);

}

(s1, s2) -> s1.compareTo(s2);

其中，参数是(s1, s2)，**参数类型可以省略**，因为**编译器可以自动推断出String类型**。-> { ... }表示方法体，所有代码写在内部即可。Lambda表达式没有class定义，因此写法非常简洁。

如果只有一行return xxx的代码，完全可以用更简单的写法：

Arrays.sort(array, (s1, s2) -> s1.compareTo(s2));

**返回值的类型也是由编译器自动推断的，这里推断出的返回值是int，因此，只要返回int，编译器就不会报错。**

### FunctionalInterface

**我们把只定义了单方法(一个抽象方法,不包括父级的以及静态,默认实现的方法)的接口称之为函数式接口--FunctionalInterface，用注解@FunctionalInterface标记。**

例如，Callable接口：

@FunctionalInterface

public interface Callable<V> {

V call() throws Exception;

}

再来看Comparator接口：

@FunctionalInterface

public interface Comparator<T> {

int compare(T o1, T o2);

boolean equals(Object obj);

default Comparator<T> reversed() {

return Collections.reverseOrder(this);

}

default Comparator<T> thenComparing(Comparator<? super T> other) {

...

}

...

}

虽然Comparator接口有很多方法，但只有一个抽象方法int compare(T o1, T o2)，其他的方法都是default方法或static方法。另外注意到boolean equals(Object obj)是Object定义的方法，不算在接口方法内。因此，Comparator也是一个FunctionalInterface。

使用Lambda表达式，我们就可以不必编写FunctionalInterface接口的实现类，从而简化代码：

Arrays.sort(array, (s1, s2) -> {

return s1.compareTo(s2);

});

实际上，除了Lambda表达式，我们还可以直接传入方法引用。例如：

public class Main {

public static void main(String[] args) {

String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };

Arrays.sort(array, Main::cmp);

System.out.println(String.join(", ", array));

}

static int cmp(String s1, String s2) {

return s1.compareTo(s2);

}

}

上述代码在Arrays.sort()中直接传入了静态方法cmp的引用，用Main::cmp表示。

因此，所谓方法引用，是指如果某个方法签名和接口恰好一致，就可以直接传入方法引用。

因为Comparator<String>接口定义的方法是int compare(String, String)，和静态方法int cmp(String, String)相比，除了方法名外，方法参数一致，返回类型相同，因此，我们说两者的方法签名一致，可以直接把方法名作为Lambda表达式传入：

Arrays.sort(array, Main::cmp);

**注意：在这里，方法签名只看参数类型和返回类型，不看方法名称，也不看类的继承关系。**

我们再看看如何引用实例方法。如果我们把代码改写如下：

public class Main {

public static void main(String[] args) {

String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };

Arrays.sort(array, String::compareTo);

System.out.println(String.join(", ", array));

}

}

不但可以编译通过，而且运行结果也是一样的，这说明String.compareTo()方法也符合Lambda定义。

观察String.compareTo()的方法定义：

public final class String {

public int compareTo(String o) {

...

}

}

这个方法的签名只有一个参数，为什么和int Comparator<String>.compare(String, String)能匹配呢？

因为实例方法有一个隐含的this参数，String类的compareTo()方法在实际调用的时候，第一个隐含参数总是传入this，相当于静态方法：

public static int compareTo(this, String o);

所以，String.compareTo()方法也可作为方法引用传入。

### 构造方法引用

除了可以引用静态方法和实例方法，我们还可以引用构造方法。

我们来看一个例子：如果要把一个List<String>转换为List<Person>，应该怎么办？

class Person {

String name;

public Person(String name) {

this.name = name;

}

}

List<String> names = List.of("Bob", "Alice", "Tim");

List<Person> persons = ???

传统的做法是先定义一个ArrayList<Person>，然后用for循环填充这个List：

List<String> names = List.of("Bob", "Alice", "Tim");

List<Person> persons = new ArrayList<>();

for (String name : names) {

persons.add(new Person(name));

}

要更简单地实现String到Person的转换，我们可以引用Person的构造方法：

public class Main {

public static void main(String[] args) {

List<String> names = List.of("Bob", "Alice", "Tim");

List<Person> persons = names.stream().map(Person::new).collect(Collectors.toList());

System.out.println(persons);

}

}

class Person {

String name;

public Person(String name) {

this.name = name;

}

public String toString() {

return "Person:" + this.name;

}

}

后面我们会讲到Stream的map()方法。现在我们看到，这里的map()需要传入的FunctionalInterface的定义是：

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

}

把泛型对应上就是方法签名Person apply(String)，即传入参数String，返回类型Person。而Person类的构造方法恰好满足这个条件，因为构造方法的参数是String，而构造方法虽然没有return语句，但它会隐式地返回this实例，类型就是Person，因此，此处可以引用构造方法。构造方法的引用写法是类名::new，因此，此处传入Person::new。

