# **Lambda Expression ( -> 斜着看 ）** http://dl2.iteye.com/upload/attachment/0120/5007/a283f7f2-7eaf-3b2c-b871-614174ffb32c.png  Lambda Expression VS. Anonymous inner class 对于只有一个方法的接口的实现类，写实现类时，只需提供（ 参数 + Lambda + 方法体 ）即可。

**Java代码**

1. **interface** A{
2. **void** show();
3. }
5. **public** **class** LambdaDemo{
6. **public** **static** **void** main(String[] args){
7. A obj;
9. // old
10. obj = **new** A(){
11. **public** **void** show(){
12. System.out.println("Hello");
13. }
14. }
16. // java 8
17. obj = () -> {
18. System.out.println("Multiple");
19. System.out.println("Lines ... ");
20. }
22. // or
23. obj = () -> System.out.println("Single line.");
25. }
26. }

说明：



# **1.函数式接口（Functional Interface）**

所谓的函数式接口，当然首先是一个接口，然后就是在这个接口里面只能有一个抽象方法。这种类型的接口也称为SAM接口，即Single Abstract Method interfaces。**为接口引入了一个新的注解：@FunctionalInterface**

1.1  函数式接口里允许定义默认方法：  
函数式接口里是可以包含默认方法，因为默认方法不是抽象方法，其有一个默认实现，所以是符合函数式接口的定义的；  
1.2  函数式接口里允许定义静态方法：  
函数式接口里是可以包含静态方法，因为静态方法不能是抽象方法，是一个已经实现了的方法，所以是符合函数式接口的定义的；  
1.3  函数式接口里允许定义java.lang.Object里的public方法：  
函数式接口里是可以包含Object里的public方法，这些方法对于函数式接口来说，不被当成是抽象方法（虽然它们是抽象方法）；因为任何一个函数式接口的实现，默认都继承了Object类，包含了来自java.lang.Object里对这些抽象方法的实现；

1.4  函数式接口里允许子接口继承多个父接口，但每个父接口中都只能存在一个抽象方法，且必须的相同的抽象方法。

下面给出几个函数式接口的例子：

@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);

}

default Predicate<T> negate() {

return (t) -> !test(t);

}

default Predicate<T> or(Predicate<? super T> other) {

Objects.*requireNonNull*(other);

return (t) -> test(t) || other.test(t);

}

static <T> Predicate<T> isEqual(Object targetRef) {

return (null == targetRef)

? Objects::*isNull*

: object -> targetRef.equals(object);

}

@FunctionalInterface

public interface Comparator<T> {

int compare(T o1, T o2);

boolean equals(Object obj);

…………

@FunctionalInterface

public interface A<T> extends Predicate, MyPredicate {

*/\*\**

*\* 1.*

*\* 父接口 MyPredicate与 Predicate中存在相同的抽象方法：boolean test(T t);*

*\* 因此子接口中的功能也只有 boolean test(T t);*

*\*/*

*/\*\**

*\* 2.*

*\* 如果多个父接口有各自不同的抽象方法，则子接口也会存在继承多个抽象方法*

*\* 此时，子接口就不是函数式接口*

*\*/*

}

# 2.@FunctionalInterface注解

Java 8为函数式接口引入了一个新注解@FunctionalInterface，主要用于编译级错误检查，加上该注解，当你写的接口不符合函数式接口定义的时候，编译器会报错。加不加@FunctionalInterface对于接口是不是函数式接口没有影响，该注解只是提醒编译器去检查该接口是否仅包含一个抽象方法。

# 3.函数式接口的运用

3.1  函数式接口也是接口，依旧可以通过之前的调用方式，通过实现类来使用接口。

public class FunctionalInterfaceTest<T> implements Predicate<T>{

public static void main(String[] args) throws InterruptedException {

Predicate p = new FunctionalInterfaceTest();

System.*out*.println(p.test(1));

}

@Override

public boolean test(T t) {

return 1 % 2 == 0;

}

}

3.2  通过Lambda表达式实现接口抽象方法

public static void main(String[] args) throws InterruptedException {

Predicate<Integer> p = a -> a % 2 == 1;

System.*out*.println(p.test(5));

System.*out*.println(p.test(2));

}

3.3  函数式接口与之前编码的比较

*/\*\**

*\* java 8 之前通常使用匿名内部类完成*

*\*/*

Collections.*sort*(dtoList, new Comparator<PlatformCouponOrderDTO>() {

@Override

public int compare(PlatformCouponOrderDTO a, PlatformCouponOrderDTO b) {

return DateUtils.*getMinutesBetween*(a.getTime(), b.getTime());

}

});

*/\*\**

*\* java 8 之后使用Lambda表达式实现函数式接口，使代码量明显减少许多*

*\*/*

Collections.*sort*(dtoList, (a, b) -> DateUtils.*getMinutesBetween*(a.getTime(), b.getTime()));

# 4.[方法引用详解](https://segmentfault.com/a/1190000012269548)

<https://www.cnblogs.com/xiaoxi/p/7099667.html>

实际上方法引用是lambda表达式的一种语法糖。  
在介绍方法引用使用方式之前，先将方法引用分下类  
方法引用共分为四类：  
1.类名::静态方法名  
2.对象::实例方法名  
3.类名::实例方法名   
4.类名::new

首先来看下第一种 类名::静态方法名 为了演示我们自定义了一个Student类

**public** **class** **Student** {

**private** String name;

**private** **int** score;

**public** **Student**(){

}

**public** **Student**(String name,**int** score){

**this**.name = name;

**this**.score = score;

}

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

**public** **int** **getScore**() {

**return** score;

}

**public** **void** **setScore**(**int** score) {

**this**.score = score;

}

**public** **static** **int** **compareStudentByScore**(Student student1,Student student2){

**return** student1.getScore() - student2.getScore();

}

}

Student类有两个属性name和score并提供了初始化name和score的构造方法，并且在最下方提供了两个静态方法分别按score和name进行比较先后顺序。  
接下来的需求是，按着分数由小到大排列并输出，在使用方法引用前，我们先使用lambda表达式的方式进行处理

Student student1 = **new** Student("zhangsan",60);

Student student2 = **new** Student("lisi",70);

Student student3 = **new** Student("wangwu",80);

Student student4 = **new** Student("zhaoliu",90);

List<Student> students = Arrays.asList(student1,student2,student3,student4);

students.sort((o1, o2) -> o1.getScore() - o2.getScore());

**students**.**forEach**(student -> System.out.println(student.getScore()));

sort方法接收一个Comparator函数式接口，接口中唯一的抽象方法compare接收两个参数返回一个int类型值，下方是Comparator接口定义

**@FunctionalInterface**

**public** **interface** **Comparator**<**T**> {

**int** **compare**(T o1, T o2);

}

我们再看下Student类中定义的compareStudentByScore静态方法

**public** **static** **int** **compareStudentByScore**(Student student1,Student student2){

**return** student1.getScore() - student2.getScore();

}

同样是接收两个参数返回一个int类型值，而且是对Student对象的分数进行比较，所以我们这里就可以 使用类名::静态方法名 方法引用替换lambda表达式

**students**.sort(**Student**::compareStudentByScore);

**students**.forEach(**student** **-**> **System**.out.println(**student**.getScore()));

第二种 对象::实例方法名  
我们再自定义一个用于比较Student元素的类

**public** **class** **StudentComparator** {

**public** **int** **compareStudentByScore**(Student student1,Student student2){

**return** student2.getScore() - student1.getScore();

}

}

StudentComparator中定义了一个非静态的，实例方法compareStudentByScore，同样该方法的定义满足Comparator接口的compare方法定义，所以这里可以直接使用 对象::实例方法名 的方式使用方法引用来替换lambda表达式

**StudentComparator** studentComparator = **new** **StudentComparator**();

**students**.**sort**(studentComparator::compareStudentByScore);

**students**.**forEach**(student -> **System**.out.println(student.getScore()));

第三种 类名::实例方法名   
这种方法引用的方式较之前两种稍微有一些不好理解，因为无论是通过类名调用静态方法还是通过对象调用实例方法这都是符合Java的语法，使用起来也比较清晰明了。那我们带着这个疑问来了解一下这个比较特殊的方法引用。  
现在再看一下Student类中静态方法的定义

**public** **static** **int** **compareStudentByScore**(Student student1,Student student2){

**return** student1.getScore() - student2.getScore();

}

虽然这个方法在语法上没有任何问题，可以作为一个工具正常使用，但是有没有觉得其在设计上是不合适的或者是错误的。这样的方法定义放在任何一个类中都可以正常使用，而不只是从属于Student这个类，那如果要定义一个只能从属于Student类的比较方法下面这个实例方法更合适一些

**public** **int** **compareByScore**(Student student){

**return** **this**.getScore() - student.getScore();

}

接收一个Student对象和当前调用该方法的Student对象的分数进行比较即可。现在我们就可以使用 类名::实例方法名 这种方式的方法引用替换lambda表达式了

**students**.sort(**Student**::compareByScore);

**students**.forEach(**student** **-**> **System**.out.println(**student**.getScore()));

这里非常奇怪，sort方法接收的lambda表达式不应该是两个参数么，为什么这个实例方法只有一个参数也满足了lambda表达式的定义（想想这个方法是谁来调用的）。这就是 类名::实例方法名 这种方法引用的特殊之处：当使用 类名::实例方法名 方法引用时，一定是lambda表达式所接收的第一个参数来调用实例方法，如果lambda表达式接收多个参数，其余的参数作为方法的参数传递进去。  
结合本例来看，最初的lambda表达式是这样的

students.sort((o1, o2) -> o1.getScore() - o2.getScore());

那使用 类名::实例方法名 方法引用时，一定是o1来调用了compareByScore实例方法，并将o2作为参数传递进来进行比较。是不是就符合了compareByScore的方法定义。

第四种 类名::new  
也称构造方法引用，和前两种类似只要符合lambda表达式的定义即可，回想下Supplier函数式接口的get方法，不接收参数有返回值，正好符合无参构造方法的定义  
@FunctionalInterface  
public interface Supplier<T> {

/\*\*

\* Gets a result.

\* @return a result

\*/

T **get**();

}

Supplier<Student> supplier = Student::new;

上面就是使用了Student类构造方法引用创建了supplier实例，以后通过supplier.get()就可以获取一个Student类型的对象，前提是Student类中存在无参构造方法。

小结：本篇全面介绍了方法引用的四种使用方式，且每种方式都有对应一个示例来帮助大家理解。当我们使用lambda表达式进行函数式编程时，如果某个方法正好满足lambda的定义，也满足实际需求的逻辑，就可以使用方法引用的方式来替换lambda表达式。接下来我们将真正开始学习stream api，并结合前面学习的内容体验stream api的强大之处。

# **5.JDK 1.8 新增加的函数接口**

* java.util.function

java.util.function 它包含了很多类，用来支持 Java的 函数式编程，该包中的函数式接口有：

|  |  |
| --- | --- |
| **序号** | **接口 & 描述** |
| 1 | **BiConsumer<T,U>**  代表了一个接受两个输入参数的操作，并且不返回任何结果 |
| 2 | **BiFunction<T,U,R>**  代表了一个接受两个输入参数的方法，并且返回一个结果 |
| 3 | **BinaryOperator<T>**  代表了一个作用于于两个同类型操作符的操作，并且返回了操作符同类型的结果 |
| 4 | **BiPredicate<T,U>**  代表了一个两个参数的boolean值方法 |
| 5 | **BooleanSupplier**  代表了boolean值结果的提供方 |
| 6 | **Consumer<T>**  代表了接受一个输入参数并且无返回的操作 |
| 7 | **DoubleBinaryOperator**  代表了作用于两个double值操作符的操作，并且返回了一个double值的结果。 |
| 8 | **DoubleConsumer**  代表一个接受double值参数的操作，并且不返回结果。 |
| 9 | **DoubleFunction<R>**  代表接受一个double值参数的方法，并且返回结果 |
| 10 | **DoublePredicate**  代表一个拥有double值参数的boolean值方法 |
| 11 | **DoubleSupplier**  代表一个double值结构的提供方 |
| 12 | **DoubleToIntFunction**  接受一个double类型输入，返回一个int类型结果。 |
| 13 | **DoubleToLongFunction**  接受一个double类型输入，返回一个long类型结果 |
| 14 | **DoubleUnaryOperator**  接受一个参数同为类型double,返回值类型也为double 。 |
| 15 | **Function<T,R>**  接受一个输入参数，返回一个结果。 |
| 16 | **IntBinaryOperator**  接受两个参数同为类型int,返回值类型也为int 。 |
| 17 | **IntConsumer**  接受一个int类型的输入参数，无返回值 。 |
| 18 | **IntFunction<R>**  接受一个int类型输入参数，返回一个结果 。 |
| 19 | **IntPredicate**  ：接受一个int输入参数，返回一个布尔值的结果。 |
| 20 | **IntSupplier**  无参数，返回一个int类型结果。 |
| 21 | **IntToDoubleFunction**  接受一个int类型输入，返回一个double类型结果 。 |
| 22 | **IntToLongFunction**  接受一个int类型输入，返回一个long类型结果。 |
| 23 | **IntUnaryOperator**  接受一个参数同为类型int,返回值类型也为int 。 |
| 24 | **LongBinaryOperator**  接受两个参数同为类型long,返回值类型也为long。 |
| 25 | **LongConsumer**  接受一个long类型的输入参数，无返回值。 |
| 26 | **LongFunction<R>**  接受一个long类型输入参数，返回一个结果。 |
| 27 | **LongPredicate**  R接受一个long输入参数，返回一个布尔值类型结果。 |
| 28 | **LongSupplier**  无参数，返回一个结果long类型的值。 |
| 29 | **LongToDoubleFunction**  接受一个long类型输入，返回一个double类型结果。 |
| 30 | **LongToIntFunction**  接受一个long类型输入，返回一个int类型结果。 |
| 31 | **LongUnaryOperator**  接受一个参数同为类型long,返回值类型也为long。 |
| 32 | **ObjDoubleConsumer<T>**  接受一个object类型和一个double类型的输入参数，无返回值。 |
| 33 | **ObjIntConsumer<T>**  接受一个object类型和一个int类型的输入参数，无返回值。 |
| 34 | **ObjLongConsumer<T>**  接受一个object类型和一个long类型的输入参数，无返回值。 |
| 35 | **Predicate<T>**  接受一个输入参数，返回一个布尔值结果。 |
| 36 | **Supplier<T>**  无参数，返回一个结果。 |
| 37 | **ToDoubleBiFunction<T,U>**  接受两个输入参数，返回一个double类型结果 |
| 38 | **ToDoubleFunction<T>**  接受一个输入参数，返回一个double类型结果 |
| 39 | **ToIntBiFunction<T,U>**  接受两个输入参数，返回一个int类型结果。 |
| 40 | **ToIntFunction<T>**  接受一个输入参数，返回一个int类型结果。 |
| 41 | **ToLongBiFunction<T,U>**  接受两个输入参数，返回一个long类型结果。 |
| 42 | **ToLongFunction<T>**  接受一个输入参数，返回一个long类型结果。 |
| 43 | **UnaryOperator<T>**  接受一个参数为类型T,返回值类型也为T。 |