Lambda表达式（也称为闭包）是整个Java 8发行版中最受期待的在Java语言层面上的改变，Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中），或者把代码看成数据：函数式程序员对这一概念非常熟悉。在JVM平台上的很多语言（Groovy，Scala，……）从一开始就有Lambda，但是Java程序员不得不使用毫无新意的匿名类来代替lambda。
关于Lambda设计的讨论占用了大量的时间与社区的努力。可喜的是，最终找到了一个平衡点，使得可以使用一种即简洁又紧凑的新方式来构造Lambdas。在最简单的形式中，一个lambda可以由用逗号分隔的参数列表、–>符号与函数体三部分表示。例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) )
```

# 函数式接口

@FuncationalInterface
可以通过 `@FunctionalInterface` 注解来显式指定一个接口是函数式接口（以避免无意声明了一个符合函数式标准的接口），加上这个注解之后，编译器就会验证该接口是否满足函数式接口的要求

```
@FunctionalInterface

public
interface
Functional {


void
method();


}
```

实现函数式类型的另一种方式是引入一个全新的*结构化*函数类型，我们也称其为"箭头"类型。例如，一个接收`String`和`Object`并返回`int`的函数类型可以被表示为`(String, Object) -> int`。我们仔细考虑了这个方式，但出于下面的原因，最终将其否定：

- 它会为Java类型系统引入额外的复杂度，并带来结构类型（Structural Type）和指名类型（Nominal Type）的混用。（Java几乎全部使用指名类型）
- 它会导致类库风格的分歧——一些类库会继续使用回调接口，而另一些类库会使用结构化函数类型

- 它的语法会变得十分笨拙，尤其在包含受检异常（checked exception）之后
- 每个函数类型很难拥有其运行时表示，这意味着开发者会受到类型擦除（erasure）的困扰和局限。比如说，我们无法对方法`m(T->U)`和`m(X->Y)`进行重载（Overload）

所以我们选择了"使用已知类型"这条路——因为现有的类库大量使用了函数式接口，通过沿用这种模式，我们使得现有类库能够直接使用lambda表达式。例如下面是Java SE 7中已经存在的函数式接口：

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.beans.PropertyChangeListener

除此之外，Java SE 8中增加了一个新的包：`java.util.function`，它里面包含了常用的函数式接口，例如：

- `Predicate<T>`——接收`T`对象并返回`boolean`
- `Consumer<T>`——接收`T`对象，不返回值
- `Function<T, R>`——接收`T`对象，返回`R`对象
- `Supplier<T>`——提供`T`对象（例如工厂），不接收值
- `UnaryOperator<T>`——接收`T`对象，返回`T`对象
- `BinaryOperator<T>`——接收两个`T`对象，返回`T`对象

除了上面的这些基本的函数式接口，我们还提供了一些针对原始类型（Primitive type）的特化（Specialization）函数式接口，例如`IntSupplier`和`LongBinaryOperator`。（我们只为`int`、`long`和`double`提供了特化函数式接口，如果需要使用其它原始类型则需要进行类型转换）同样的我们也提供了一些针对多个参数的函数式接口，例如`BiFunction<T, U, R>`，它接收`T`对象和`U`对象，返回`R`对象。

函数式接口(Functional Interface)就是一个具有一个方法的普通接口。

函数式接口可以被隐式转换为lambda表达式。

函数式接口可以现有的函数友好地支持 lambda。

JDK 1.8之前已有的函数式接口:

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction

- java.util.Comparator
- java.io.FileFilter
- java.nio.file.PathMatcher
- java.lang.reflect.InvocationHandler
- java.beans.PropertyChangeListener
- java.awt.event.ActionListener
- javax.swing.event.ChangeListener

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>**<br>代表了一个接受两个输入 |
| 2 | **BiFunction<T,U,R>**<br>代表了一个接受两个输入 |
| 3 | **BinaryOperator<T>**<br>代表了一个作用于于两个<br>作符同类型的结果 |
| 4 | **BiPredicate<T,U>**<br>代表了一个两个参数的b |
| 5 | **BooleanSupplier**<br>代表了boolean值结果的 |
| 6 | **Consumer<T>**<br>代表了接受一个输入参数 |
| 7 | **DoubleBinaryOperat**<br>代表了作用于两个doub<br>double值的结果。 |
| 8 | **DoubleConsumer**<br>代表一个接受double值 |
| 9 | **DoubleFunction<R>**<br>代表接受一个double值 |
| 10 | **DoublePredicate**<br>代表一个拥有double值 |
| 11 | **DoubleSupplier**<br>代表一个double值结构 |
| 12 | **DoubleToIntFunctio**<br>接受一个double类型输 |
| 13 | **DoubleToLongFunct**<br>接受一个double类型输 |
| 14 | **DoubleUnaryOperat** |

| | |
|---|---|
| | 接受一个参数同为类型c |
| 15 | **Function<T,R>**<br>接受一个输入参数，返回 |
| 16 | **IntBinaryOperator**<br>接受两个参数同为类型i |
| 17 | **IntConsumer**<br>接受一个int类型的输入 |
| 18 | **IntFunction<R>**<br>接受一个int类型输入参 |
| 19 | **IntPredicate**<br>：接受一个int输入参数 |
| 20 | **IntSupplier**<br>无参数，返回一个int类 |
| 21 | **IntToDoubleFunctio**<br>接受一个int类型输入， |
| 22 | **IntToLongFunction**<br>接受一个int类型输入， |
| 23 | **IntUnaryOperator**<br>接受一个参数同为类型i |
| 24 | **LongBinaryOperato**<br>接受两个参数同为类型l |
| 25 | **LongConsumer**<br>接受一个long类型的输 |
| 26 | **LongFunction<R>**<br>接受一个long类型输入 |
| 27 | **LongPredicate**<br>R接受一个long输入参数 |
| 28 | **LongSupplier**<br>无参数，返回一个结果l |
| 29 | **LongToDoubleFunct**<br>接受一个long类型输入， |
| 30 | **LongToIntFunction**<br>接受一个long类型输入， |
| 31 | **LongUnaryOperator**<br>接受一个参数同为类型l |
| 32 | **ObjDoubleConsume**<br>接受一个object类型和-<br>值。 |
| 33 | **ObjIntConsumer<T>**<br>接受一个object类型和- |
| 34 | **ObjLongConsumer<**<br>接受一个object类型和- |

| | |
|---|---|
| 35 | **Predicate<T>**<br>接受一个输入参数，返 |
| 36 | **Supplier<T>**<br>无参数，返回一个结果。 |
| 37 | **ToDoubleBiFunction**<br>接受两个输入参数，返 |
| 38 | **ToDoubleFunction<**<br>接受一个输入参数，返 |
| 39 | **ToIntBiFunction<T,**<br>接受两个输入参数，返 |
| 40 | **ToIntFunction<T>**<br>接受一个输入参数，返 |
| 41 | **ToLongBiFunction<**<br>接受两个输入参数，返 |
| 42 | **ToLongFunction<T>**<br>接受一个输入参数，返 |
| 43 | **UnaryOperator<T>**<br>接受一个参数为类型T, |

# 函数式接口实例

Predicate <T> 接口是一个函数式接口，它接受一个输入参数 T，返回一个布尔值结果。

该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）。

该接口用于测试对象是 true 或 false。

我们可以通过以下实例（Java8Tester.java）来了解函数式接口 Predicate <T> 的使用：

**Java8Tester.java 文件**

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Java8Tester {
   public static void main(String args[]){
      List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9);
```

```
        // Predicate<Integer> predicate = n -> true
        // n 是一个参数传递到 Predicate 接口的 test 方法
        // n 如果存在则 test 方法返回 true

        System.out.println("输出所有数据:");

        // 传递参数 n
        eval(list, n->true);

        // Predicate<Integer> predicate1 = n -> n%2 == 0
        // n 是一个参数传递到 Predicate 接口的 test 方法
        // 如果 n%2 为 0 test 方法返回 true

        System.out.println("输出所有偶数:");
        eval(list, n-> n%2 == 0 );

        // Predicate<Integer> predicate2 = n -> n > 3
        // n 是一个参数传递到 Predicate 接口的 test 方法
        // 如果 n 大于 3 test 方法返回 true

        System.out.println("输出大于 3 的所有数字:");
        eval(list, n-> n > 3 );
    }

    public static void eval(List<Integer> list,
Predicate<Integer> predicate) {
        for(Integer n: list) {

            if(predicate.test(n)) {
                System.out.println(n + " ");
            }
        }
    }
}
```

执行以上脚本，输出结果为：

```
$ javac Java8Tester.java
$ java Java8Tester
```

输出所有数据：

```
1
2
3
4
5
6
7
8
9
```

输出所有偶数：

```
2
4
6
8
```

输出大于 `3` 的所有数字：

```
4
5
6
7
8
9
```