1. Java8新特性

**Lambda 表达式和函数式接口**

Lambda 表达式定义：

Lambda: In programming languages such as Lisp, Python and Ruby lambda is an operator used to denote **anonymous functions** or **closures**, following the usage of lambda calculus.

为何需要使用 Lambda 表达式：

* 在 Java 中，我们无法将函数作为一个参数传递给一个方法，也无法声明一个返回一个函数的方法。
* 在 JavaScript 中，函数的参数是一个函数，返回值是另一个函数的情况是非常常见的，JavaScript 是一门典型的函数式语言。

我们通过一个例子来引入：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/11 14:55  \* @Description:  \*/ public class Test1 {  public static void main(String[] args) {  List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);  for (int i = 0; i < list.size(); i++) {  System.out.println(list.get(i));  }  System.out.println("-----------------");  for (int val : list) {  System.out.println(val);  }   System.out.println("-----------------");  list.forEach(new Consumer<Integer>() {  @Override  public void accept(Integer integer) {  System.out.println(integer);  }  });  } } |

这是 3 种遍历集合的方式，第一就是简单的遍历，第二种是我们是常说的增强 for 循环遍历。第三种就是 Java 8 新增的方法，先看看 Consumer 这个接口。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | package java.util.function;  import java.util.Objects;   @FunctionalInterface public interface Consumer<T> {   void accept(T t);   default Consumer<T> andThen(Consumer<? super T> after) {  Objects.requireNonNull(after);  return (T t) -> { accept(t); after.accept(t); };  } } |

注解上是一个函数式接口，我们看看这个接口的作用。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 | package java.lang;  import java.lang.annotation.\*;  /\*\*  \* An informative annotation type used to indicate that an interface  \* type declaration is intended to be a <i>functional interface</i> as  \* defined by the Java Language Specification.  \*  \* Conceptually, a functional interface has exactly one abstract  \* method. Since {@linkplain java.lang.reflect.Method#isDefault()  \* default methods} have an implementation, they are not abstract. If  \* an interface declares an abstract method overriding one of the  \* public methods of {@code java.lang.Object}, that also does  \* <em>not</em> count toward the interface's abstract method count  \* since any implementation of the interface will have an  \* implementation from {@code java.lang.Object} or elsewhere.  \*  \* 有且只有一个抽象方法的接口，如果有重写 Object 中的方法，那也是可以的。  \*  \* <p>Note that instances of functional interfaces can be created with  \* lambda expressions, method references, or constructor references.  \*  \* 函数式接口可以通过 lambda 表达式、方法引用和构造方法引用来创建。  \*  \* <p>If a type is annotated with this annotation type, compilers are  \* required to generate an error message unless:  \*  \* <ul>  \* <li> The type is an interface type and not an annotation type, enum, or class.  \* <li> The annotated type satisfies the requirements of a functional interface.  \* </ul>  \*  \* <p>However, the compiler will treat any interface meeting the  \* definition of a functional interface as a functional interface  \* regardless of whether or not a {@code FunctionalInterface}  \* annotation is present on the interface declaration.  \*  \* 编译器会对满足定义函数式接口的接口当做函数式接口，不管它有没有 @FunctionalInterface 注解声明。  \*  \* @jls 4.3.2. The Class Object  \* @jls 9.8 Functional Interfaces  \* @jls 9.4.3 Interface Method Body  \* @since 1.8  \*/ @Documented @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE) public @interface FunctionalInterface {} |

函数式接口可以通过 lambda 表达式、方法引用和构造方法引用来创建。

* lambda 表达式：() -> System.out.println(i)
* 方法引用：System.out::print
* 构造方法引用：new::ArrayList

用一个例子来说明什么是函数式接口。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | @FunctionalInterface interface Cons {  void print();  String toString(); }  /\*\*  \* @Author: cuzz  \* @Date: 2019/8/11 16:13  \* @Description:  \*/ public class Test2 {   public void test(Cons func) {  func.print();  }   public static void main(String[] args) {  Test2 test2 = new Test2();  test2.test(() -> System.out.println("xxx"));   Cons func = () -> System.out.println("yyy");  test2.test(func);  System.out.println(func.getClass()); // 输出 class com.cuzz.Test2$$Lambda$2/2074407503  System.out.println(func.getClass().getSuperclass()); // 输出 class java.lang.Object  } } |

可以说明3点：

* 函数式接口只有一个非重写 Object 的抽象方法
* lambda 表达式就是一个匿名类
* 对于一个函数式接口，我们并不关心这个抽象方法的名称。

**从Consumer深入理解函数式接口和方法引用**

我们回到这个例子当中

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | public class Test1 {  public static void main(String[] args) {  list.forEach(new Consumer<Integer>() {  @Override  public void accept(Integer integer) {  System.out.println(integer);  }  });  } } |

先看看 Iterable#forEach 这个方法，是 Iterable 这个接口这的默认方法，在 Java 8 中接口中是允许默认方法。对于 Iterable#forEach 是对每个元素执行给定的动作。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 | public interface Iterable<T> {  /\*\*  \* Returns an iterator over elements of type {@code T}.  \*  \* @return an Iterator.  \*/  Iterator<T> iterator();   /\*\*  \* Performs the given action for each element of the {@code Iterable}  \* until all elements have been processed or the action throws an  \* exception. Unless otherwise specified by the implementing class,  \* actions are performed in the order of iteration (if an iteration order  \* is specified). Exceptions thrown by the action are relayed to the  \* caller.  \*   \* 对每个元素执行给定的动作。  \*  \* @implSpec  \* <p>The default implementation behaves as if:  \* <pre>{@code  \* for (T t : this)  \* action.accept(t);  \* }</pre>  \*  \* @param action The action to be performed for each element  \* @throws NullPointerException if the specified action is null  \* @since 1.8  \*/  default void forEach(Consumer<? super T> action) {  Objects.requireNonNull(action);  for (T t : this) {  action.accept(t);  }  }   default Spliterator<T> spliterator() {  return Spliterators.spliteratorUnknownSize(iterator(), 0);  } } |

看看 Consumer 是什么

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 | package java.util.function;  import java.util.Objects;  /\*\*  \* Represents an operation that accepts a single input argument and returns no  \* result. Unlike most other functional interfaces, {@code Consumer} is expected  \* to operate via side-effects.  \*  \* 表示一个操作接受单一输入参数，无返回结果。  \*  \* <p>This is a <a href="package-summary.html">functional interface</a>  \* whose functional method is {@link #accept(Object)}.  \*  \* @param <T> the type of the input to the operation  \*  \* @since 1.8  \*/ @FunctionalInterface public interface Consumer<T> {   /\*\*  \* Performs this operation on the given argument.  \*  \* @param t the input argument  \*/  void accept(T t);   /\*\*  \* Returns a composed {@code Consumer} that performs, in sequence, this  \* operation followed by the {@code after} operation. If performing either  \* operation throws an exception, it is relayed to the caller of the  \* composed operation. If performing this operation throws an exception,  \* the {@code after} operation will not be performed.  \*  \* @param after the operation to perform after this operation  \* @return a composed {@code Consumer} that performs in sequence this  \* operation followed by the {@code after} operation  \* @throws NullPointerException if {@code after} is null  \*/  default Consumer<T> andThen(Consumer<? super T> after) {  Objects.requireNonNull(after);  return (T t) -> { accept(t); after.accept(t); };  } } |

**lambda 表达式的作用：**

* lambda 表达式为 Java 添加了缺失的函数式编程特性，使我们能将函数当做一等公民看待。
* 在将函数作为一等公民的语言中，lambda 表达式的类型是函数。但在 Java 中，lambda 表达式是对象，它们必须依附于一类特别的对象（函数式接口）；

**Lambda 表达式的深入**

对于 lambda 表达式需要根据上下文来推断，我们并不知道() -> {} 是什么，不知道对应的参数，方法是什么，只用通过前面的 Cons 定义才知道。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | @FunctionalInterface interface Cons1 {  void print1(); }  @FunctionalInterface interface Cons2 {  void print2(); }  /\*\*  \* @Author: cuzz  \* @Date: 2019/8/11 16:13  \* @Description:  \*/ public class Test2 {  public static void main(String[] args) {  Cons1 cons1 = () -> {};  Cons2 cons2 = () -> {};  System.out.println(cons1.getClass().getInterfaces()[0]); // interface com.cuzz.Cons1  System.out.println(cons2.getClass().getInterfaces()[0]); // interface com.cuzz.Cons2  } } |

我们先看一个排序的例子：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/12 23:09  \* @Description: 排序  \*/ public class Test4 {   public static void main(String[] args) {  List<String> list = Arrays.asList("cuzz", "faker", "mlxg");    Collections.sort(list, (String s1, String s2) -> {  return s2.compareTo(s1);  }); // 1    Collections.sort(list, (s1, s2) -> s2.compareTo(s1)); // 2  } } |

从 1 到 2 简化了很多，修饰符 String 和 return 都可以省略。**Java Lambda 表达式是一种匿名函数，它没有声明方法，也没有访问修饰符、返回值和名字。**

**Lambda 表达式作用：**

* 传递行为，而不仅仅是值
* 提升抽象层次
* API 重用性好
* 更加灵活

**Lambda 基本语法：**

* Java 中的 Lambda 表达式基本语法
  + 如：(argument) -> {body}
  + 省略类型：(arg1, arg2, ...) -> {body}
  + 有类型：(type1 arg1, type2 arg2, ...) -> {body}
* Lambda 示例说明
  + (int a, int b) -> {return a + b;}
  + () -> System.out.println("hello world")
  + (String s) -> {System.out.println(s);}
  + () -> 42
  + () -> {return "cuzz"};
* Lambda结构
  + 一个 Lambda 表达式可以有零个或多个参数
  + 参数的类型既可以明确声明，也可以根据上下文来推断，如：(int a) 与 (a) 效果相同
  + 所有的参数需包含在圆括号内，参数之间用逗号相隔。如：(a, b) 或 (String a, int b float c)
  + 空圆括号表示参数集为空，如：() -> 42
  + 当只有一个参数，且其类型可推导时，圆括号可以省略，如：a -> return a \* a
  + Lambda 表达式的主题可以包含零条或多条语句
  + 如果 Lambda 表达式的主体只有一条语句，花括号可以省略，匿名函数的返回类型与该主体表达式一致
  + 如果 Lambda 表达式的主体包含一条以上语句，表达式必须使用花括号

**Function**

直接先看源码

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 | /\*\*  \* Represents a function that accepts one argument and produces a result.  \*  \* <p>This is a <a href="package-summary.html">functional interface</a>  \* whose functional method is {@link #apply(Object)}.  \*  \* @param <T> the type of the input to the function  \* @param <R> the type of the result of the function  \*  \* @since 1.8  \*/ @FunctionalInterface public interface Function<T, R> {   /\*\*  \* Applies this function to the given argument.  \*  \* @param t the function argument  \* @return the function result  \*/  R apply(T t);   default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  Objects.requireNonNull(before);  return (V v) -> apply(before.apply(v));  }   default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  Objects.requireNonNull(after);  return (T t) -> after.apply(apply(t));  }   /\*\*  \* Returns a function that always returns its input argument.  \*  \* @param <T> the type of the input and output objects to the function  \* @return a function that always returns its input argument  \*/  static <T> Function<T, T> identity() {  return t -> t;  } } |

可以看出 Function 有一个抽象方法和两个默认方法以及一个静态方法。

**（1） Function#apply**

Stream#map 里就是接受一个 Function，对于 Function 意思就是从一个映射到另一个。下面例子就是把字符串映射到大写。对于 String::toUpperCase 使用的是方法引用。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/11 23:13  \* @Description:  \*/ public class Test3 {  public static void main(String[] args) {  List<String> list = Arrays.asList("cuzz", "faker", "mlxg");   list.stream().map(item -> item.toUpperCase()).forEach(item -> System.out.println(item));  list.stream().map(String::toUpperCase).forEach(System.out::println);  Function<String, String> function = String::toUpperCase;  System.out.println(function.getClass());  } } |

我们看一个例子：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/13 0:08  \* @Description:  \*/ public class FunctionTest {   public static void main(String[] args) {  FunctionTest function= new FunctionTest();  int res1 = function.compute(100, target -> target \* target);  int res2 = function.compute(100, target -> target + 1);  System.out.println(res1); // 10000  System.out.println(res2); // 101   int res3 = function.pow(100);  int res4 = function.addOne(100);  System.out.println(res3); // 10000  System.out.println(res4); // 101   }   public int compute(int a, Function<Integer, Integer> function) {  return function.apply(a);  }   public int pow(int a) {  return a \* a;  }  public int addOne(int a) {  return a + 1;  } } |

**看看 #compute 这个方法，第二个参数传递的是行为，而不是具体的值。** 我们本来要定义两个方法，pow 和 addOne 现在把这种行为传递进来。

**（2）Function#compose 和 Function#andThen**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 | /\*\*  \* Returns a composed function that first applies the {@code before}  \* function to its input, and then applies this function to the result.  \* If evaluation of either function throws an exception, it is relayed to  \* the caller of the composed function.  \*  \* @param <V> the type of input to the {@code before} function, and to the  \* composed function  \* @param before the function to apply before this function is applied  \* @return a composed function that first applies the {@code before}  \* function and then applies this function  \* @throws NullPointerException if before is null  \*  \* @see #andThen(Function)  \*/ default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  Objects.requireNonNull(before);  return (V v) -> apply(before.apply(v)); }  /\*\*  \* Returns a composed function that first applies this function to  \* its input, and then applies the {@code after} function to the result.  \* If evaluation of either function throws an exception, it is relayed to  \* the caller of the composed function.  \*  \* @param <V> the type of output of the {@code after} function, and of the  \* composed function  \* @param after the function to apply after this function is applied  \* @return a composed function that first applies this function and then  \* applies the {@code after} function  \* @throws NullPointerException if after is null  \*  \* @see #compose(Function)  \*/ default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  Objects.requireNonNull(after);  return (T t) -> after.apply(apply(t)); } |

* compose方法是一个默认方法，这个方法接收一个 function 作为参数，将参数 function 执行的结果作为参数给调用的 function，以此来实现两个function组合的功能。
* andThen 方法也是接收一个 function 作为参数，与 compse 不同的是，先执行本身的 apply 方法，将执行的结果作为参数给参数中的 function。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/20 23:59  \* @Description: #compose and #andThen test  \*/ public class FunctionTest2 {   public static void main(String[] args) {  FunctionTest2 test = new FunctionTest2();  System.out.println(test.compute1(2, value -> value \* 2, value -> value \* value)); // 8  System.out.println(test.compute2(2, value -> value \* 2, value -> value \* value)); // 16  }    public int compute1(int a, Function<Integer, Integer> function1, Function<Integer, Integer> function2) {  return function1.compose(function2).apply(a);  }   public int compute2(int a, Function<Integer, Integer> function1, Function<Integer, Integer> function2) {  return function1.andThen(function2).apply(a);  } } |

发现 compute1 是先执行第二个 Function 再执行第一，compute2 相反。

**BiFunction**

先看源码

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 | /\*\*  \* Represents a function that accepts two arguments and produces a result.  \* This is the two-arity specialization of {@link Function}.  \*  \* <p>This is a <a href="package-summary.html">functional interface</a>  \* whose functional method is {@link #apply(Object, Object)}.  \*  \* @param <T> the type of the first argument to the function  \* @param <U> the type of the second argument to the function  \* @param <R> the type of the result of the function  \*  \* @see Function  \* @since 1.8  \*/ @FunctionalInterface public interface BiFunction<T, U, R> {   /\*\*  \* Applies this function to the given arguments.  \*  \* @param t the first function argument  \* @param u the second function argument  \* @return the function result  \*/  R apply(T t, U u);   /\*\*  \* Returns a composed function that first applies this function to  \* its input, and then applies the {@code after} function to the result.  \* If evaluation of either function throws an exception, it is relayed to  \* the caller of the composed function.  \*  \* @param <V> the type of output of the {@code after} function, and of the  \* composed function  \* @param after the function to apply after this function is applied  \* @return a composed function that first applies this function and then  \* applies the {@code after} function  \* @throws NullPointerException if after is null  \*/  default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after) {  Objects.requireNonNull(after);  return (T t, U u) -> after.apply(apply(t, u));  } } |

我看一个例子

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/21 7:36  \* @Description:  \*/ public class BiFunctionTest {   public static void main(String[] args) {  BiFunctionTest test = new BiFunctionTest();  // 加法  System.out.println(test.add(1, 2));  System.out.println(test.compute(1, 2, (a, b) -> a + b));    // 减法  System.out.println(test.subtract(1, 2));  System.out.println(test.compute(1, 2, (a, b) -> a - b));  }    public int compute(int a, int b, BiFunction<Integer, Integer, Integer> biFunction) {  return biFunction.apply(a, b);  }   public int add(int a, int b) {  return a + b;  }   public int subtract(int a, int b) {  return a - b;  } } |

以前我们定义一个四则运算需要需要先定义方法，现在通过 BiFunction 可以把这种行为传递进来。

**Predicate**

**（1）源码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 | /\*\*  \* Represents a predicate (boolean-valued function) of one argument.  \*  \* <p>This is a <a href="package-summary.html">functional interface</a>  \* whose functional method is {@link #test(Object)}.  \*  \* @param <T> the type of the input to the predicate  \*  \* @since 1.8  \*/ @FunctionalInterface public interface Predicate<T> {   /\*\*  \* Evaluates this predicate on the given argument.  \*  \* @param t the input argument  \* @return {@code true} if the input argument matches the predicate,  \* otherwise {@code false}  \*/  boolean test(T t);   /\*\*  \* Returns a composed predicate that represents a short-circuiting logical  \* AND of this predicate and another. When evaluating the composed  \* predicate, if this predicate is {@code false}, then the {@code other}  \* predicate is not evaluated.  \*  \* <p>Any exceptions thrown during evaluation of either predicate are relayed  \* to the caller; if evaluation of this predicate throws an exception, the  \* {@code other} predicate will not be evaluated.  \*  \* @param other a predicate that will be logically-ANDed with this  \* predicate  \* @return a composed predicate that represents the short-circuiting logical  \* AND of this predicate and the {@code other} predicate  \* @throws NullPointerException if other is null  \*/  default Predicate<T> and(Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> test(t) && other.test(t);  }   /\*\*  \* Returns a predicate that represents the logical negation of this  \* predicate.  \*  \* @return a predicate that represents the logical negation of this  \* predicate  \*/  default Predicate<T> negate() {  return (t) -> !test(t);  }   /\*\*  \* Returns a composed predicate that represents a short-circuiting logical  \* OR of this predicate and another. When evaluating the composed  \* predicate, if this predicate is {@code true}, then the {@code other}  \* predicate is not evaluated.  \*  \* <p>Any exceptions thrown during evaluation of either predicate are relayed  \* to the caller; if evaluation of this predicate throws an exception, the  \* {@code other} predicate will not be evaluated.  \*  \* @param other a predicate that will be logically-ORed with this  \* predicate  \* @return a composed predicate that represents the short-circuiting logical  \* OR of this predicate and the {@code other} predicate  \* @throws NullPointerException if other is null  \*/  default Predicate<T> or(Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> test(t) || other.test(t);  }   /\*\*  \* Returns a predicate that tests if two arguments are equal according  \* to {@link Objects#equals(Object, Object)}.  \*  \* @param <T> the type of arguments to the predicate  \* @param targetRef the object reference with which to compare for equality,  \* which may be {@code null}  \* @return a predicate that tests if two arguments are equal according  \* to {@link Objects#equals(Object, Object)}  \*/  static <T> Predicate<T> isEqual(Object targetRef) {  return (null == targetRef)  ? Objects::isNull  : object -> targetRef.equals(object);  } } |

**（2）例子**

以前我们根据不同的条件筛选数据需要些多个方法，现在只要先定义一个这种接受行为的方法。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/21 23:35  \* @Description: Predicate test  \*/ public class PredicateTest {  public static void main(String[] args) {  List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);  PredicateTest test = new PredicateTest();   // 查找奇数  test.findOdd(list);  test.conditionFilter(list, i -> i % 2 != 0);   // 查找偶数  test.findEven(list);  test.conditionFilter(list, i -> i % 2 == 0);  }   public void conditionFilter(List<Integer> list, Predicate<Integer> predicate) {  for (int i : list) {  if (predicate.test(i)) {  System.out.print(i + " ");  }  }  System.out.println();  }   public void findOdd(List<Integer> list) {  for (int i : list) {  if (i % 2 != 0) {  System.out.print(i + " ");  }  }  System.out.println();  }   public void findEven(List<Integer> list) {  for (int i : list) {  if (i % 2 == 0) {  System.out.print(i + " ");  }  }  System.out.println();  } } |

**（3）Predicate#and 和 Predicate#or**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | public class PredicateTest {  public static void main(String[] args) {  List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);  PredicateTest test = new PredicateTest();   // 查找 大于 3 的奇数  test.conditionFilter2(list, i -> i > 3, i -> i % 2 != 0);  }   public void conditionFilter2(List<Integer> list, Predicate<Integer> predicate1, Predicate<Integer> predicate2) {  for (int i : list) {  if (predicate1.and(predicate2).test(i)) {  System.out.print(i + " ");  }  }  System.out.println();  } } |

**Supplier**

**（1）不接受参数，返回一个值。**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | /\*\*  \* Represents a supplier of results.  \*  \* <p>There is no requirement that a new or distinct result be returned each  \* time the supplier is invoked.  \*  \* <p>This is a <a href="package-summary.html">functional interface</a>  \* whose functional method is {@link #get()}.  \*  \* @param <T> the type of results supplied by this supplier  \*  \* @since 1.8  \*/ @FunctionalInterface public interface Supplier<T> {   /\*\*  \* Gets a result.  \*  \* @return a result  \*/  T get(); } |

**（2）例子**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | /\*\*  \* @Author: cuzz  \* @Date: 2019/8/22 23:32  \* @Description:  \*/ public class SupplierTest {  public static void main(String[] args) {  Supplier<Student> supplier1 = () -> new Student();  Supplier<Student> supplier2 = Student::new;  } }  @Data class Student {  private String name = "cuzz";  private int age = 20; } |