[JAVA帮助文档 1](#_Toc507487678)

[一. java8新特性 1](#_Toc507487679)

[1. Lambda表达式 1](#_Toc507487680)

[1.1 Lambda表达式简介 1](#_Toc507487681)

[1.2 语法 1](#_Toc507487682)

[1.3 Lambda行为参数化 3](#_Toc507487683)

[1.4 函数式接口 5](#_Toc507487684)

[1.5 java Buildin函数式接口 5](#_Toc507487685)

[1.6 交叉类型 10](#_Toc507487686)

[1.7 静态方法引用 11](#_Toc507487687)

[1.8 实例方法调用 12](#_Toc507487688)

[1.9 构造函数引用 13](#_Toc507487689)

[1.10 通用方法引用 14](#_Toc507487690)

[1.11 Lambda表达式作用域 15](#_Toc507487691)

[1.12 Lambda变量捕获 15](#_Toc507487692)

[1.13 Lambda主体语句 17](#_Toc507487693)

[2. Optional类 18](#_Toc507487694)

[3. 接口默认方法 19](#_Toc507487695)

# JAVA笔记

## java8新特性

### Lambda表达式

#### Lambda表达式简介

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。

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）。

使用 Lambda 表达式可以使代码变的更加简洁紧凑。

#### 语法

lambda 表达式的语法格式如下：

(parameters) -> expression 或 (parameters) ->{ statements; }

即：

(参数) -> 表达式 或 (参数) ->{语句块;}

以下是lambda表达式的重要特征:

**可选类型声明：**不需要声明参数类型，编译器可以统一识别参数值。见

**可选的参数圆括号：**一个参数无需定义圆括号，但多个参数需要定义圆括号, 没有参数时要保留括号。见

**可选的大括号：**如果主体只包含一个语句，可以不使用大括号。见, 如果只是一个表达式,则需要加上return组成一个语句,然后才能使用大括号,反之如果返回的是一个表达式的结果, 则可以省略大括号及return关键字.如果1语句中的a + b; 改为{a + b;};则会编译报错，应改为{ return a + b;};才会正确。return关键字和大括号必须同时存在。如语句会编译报错。

**可选的返回关键字：**如果主体只有一个表达式返回值则编译器会自动返回值，省略大括号时可同时省略return关键字。有return关键字时大括号不能省略。

**final修饰符：参数可用final修饰符修饰。**

实例:

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  MathOperation add = (**int** a, **int** b) -> a + b; //  //可选类型声明：可以不声明参数类型，编译器可以统一识别参数值。  MathOperation sub = (a, b) -> a - b; //  MathOperation mult = (a, b) -> { a\*b;}; //  //可选的大括号：如果主体只包含一个语句，就不需要使用大括号。  MathOperation div = (a, b) -> a/b; //  MathOperation div1 = (a, b) -> **return** a/b; //编译报错  System.***out***.println("10 + 5 = " + add.operation(10, 5)); //10 + 5 = 15  System.***out***.println("10 - 5 = " + sub.operation(10, 5)); //10 - 5 = 5  System.***out***.println("10 \* 5 = " + mult.operation(10, 5)); //10 \* 5 = 50  System.***out***.println("10 / 5 = " + div.operation(10, 5)); //10 / 5 = 2    //可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。  Hello hello = message -> System.***out***.println("Hello " + message); //  hello.sayHello("zs"); //Hello zs  //没有参数时仍然要保留括号  Hi hi = () -> System.***out***.println("Hi"); //  hi.sayHi(); //Hi    }    **interface** MathOperation {  **int** operation(**int** a, **int** b);  }    **interface** Hello {  **void** sayHello(String message);  }  **interface** Hi {  **void** sayHi();  }  } |

注意:Lambda表达式接口必须是函数式接口，即只能包含一个抽象方法。一个以上会报错，但可以有default方法。

#### Lambda行为参数化

将lambda表达式作为参数传递给方法。

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  *operate*((x,y) -> x+y);  *operate*((x,y) -> x-y);  *operate*((x,y) -> x\*y);  *operate*((x,y) -> x/y);  }    **private** **static** **void** operate(IntOperation operation) {  **int** a = 4, b = 2;  System.***out***.println(operation.operation(a, b));  }    @FunctionalInterface  **interface** IntOperation {  **int** operation(**int** a, **int** b);  }    } |

注意：如果出现了方法重载，则需要指明参数类型,如果此时不指明参数类型，编译器无法判断你调用的是哪个方法，则会编译报错。

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  *operate*((**int** x, **int** y) -> x+y);  *operate*((**long** x, **long** y) -> x-y);  }    **private** **static** **void** operate(IntOperation operation) {  **int** a = 4, b = 2;  System.***out***.println(operation.operation(a, b));  }    **private** **static** **void** operate(LongOperation operation) {  **long** a = 4, b = 2;  System.***out***.println(operation.operation(a, b));  }    @FunctionalInterface  **interface** IntOperation {  **int** operation(**int** a, **int** b);  }    @FunctionalInterface  **interface** LongOperation {  **long** operation(**long** a, **long** b);  }  } |

#### 函数式接口

函数式接口是具有一个方法的接口，用作lambda表达式的类型。

lambda表达式表示函数式接口的实例。

lambda表达式本身不能用作独立的表达式。如（x，y）-> x + y;不能单独存在。

函数式接口可以重新申明Object中的方法，该方法不会被视为抽象方法。

|  |
| --- |
| @FunctionalInterface  **interface** LongOperation {  **long** operation(**long** a, **long** b);  **boolean** equals(Object obj); //重新申明Object类中的equals方法。此时不会报错，如果将其改为非Object类中的方法名，则会报错。  } |

**@FunctionalInterface注释**

@FunctionalInterface注释在java.lang包中定义。我们可以选择使用它来标记一个函数式接口。

如果注释@FunctionalInterface在非函数式接口或其他类型（如类）上注释，则会发生编译时错误。

具有一个抽象方法的接口仍然是一个功能接口，即使我们不用@FunctionalInterface注释。

#### java Buildin函数式接口

**Function (函数)**

表示接受类型T的参数并返回类型R的结果的函数。

|  |
| --- |
| API:  public interface Function<T,R>{  ...  public R apply(T t);  ...  }  示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Function<String,Integer> f = (String s) -> s.length();  System.***out***.println(f.apply("string")); //6  }  }  注：还有andThen():创建一个Function，调用当前函数和指定的函数后得到结果。  compose():创建一个Function，该函数调用指定的函数，然后调用当前函数并返回结果。  identity():创建一个返回其参数的函数。  注：对应的专用函数  DoubleFunction<R>  R apply(double value) 传入一个double类型返回R类型。  IntFunction<R> R apply(int value)  DoubleToIntFunction int applyAsInt(double value)  DoubleToLongFunction long applyAsLong(double value)  IntToDoubleFunction double applyAsDouble(int value)  IntToLongFunction long applyAsLong(int value)  LongFunction<R> R apply(long value)  ToDoubleFunction<T> double applyAsDouble(T value)  ToIntFunction<T> int applyAsInt(T value)  ToLongFunction<T> long applyAsLong(T value) |

**BiFunction**

表示一个函数，它接受类型T和U的两个参数，并返回类型R的结果。

|  |
| --- |
| API:  public interface BiFunction<T,U,R>{  ...  public R apply(T t, U u);  ...  }  示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  BiFunction<Integer,Double,String> bf = (Integer a, Double b) -> String.*valueOf*(a \* b);  System.***out***.println(bf.apply(5, 4.0)); //20.0  }  }  注：此函数还有一个andThen()  注：对应的专用函数  ToDoubleBiFunction<T,U> double applyAsDouble(T t, U u) 传入T , U 返回double类型  ToIntBiFunction<T,U> int applyAsInt(T t, U u)  ToLongBiFunction<T,U> long applyAsLong(T t, U u) |

**Predicate (判断)**

表示为指定参数返回true或false的布尔函数。

|  |
| --- |
| API:  public Predicate<T> {  ...  public boolean test(T t);  ...  }  示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Predicate<Character> p = (Character c) -> c >= '0' && c <= '9';  System.***out***.println(p.test('a')); //false  }  }  注：此函数还有and() : 组合两个具有短路逻辑AND的谓词。  isEqual() : 返回一个谓词，根据Objects.equals(Object，Object)测试两个参数是否相等。  negate() : 否定原始谓词的谓词。  or() : 组合了具有短路逻辑或的两个谓词。  注：对应的专用函数  DoublePredicate boolean test(double value)  IntPredicate boolean test(int value)  LongPredicate boolean test(long value) |

**BiPredicate**

表示为两个指定的参数返回true或false的布尔函数。

|  |
| --- |
| API:  public interface BiPredicate<T,U>{  ...  public boolean test(T t, U u);  ...  }  注：此函数还有and() , negate() , or()方法？？？？ |

**Consumer (消费者)**

表示接受参数并且不返回结果的操作。

|  |
| --- |
| API:  public interface Consumer<T>{  ...  public void accept(T t);  ...  }  注：此函数还有andThen()方法？？？  注：对应的专用函数  DoubleConsumer void accept(double value)  IntConsumer void accept(int value)  LongConsumer void accept(long value)  ObjDoubleConsumer void accept(T t, double value)  ObjIntConsumer void accept(T t, int value)  ObjLongConsumer void accept(T t, long value) |

**BiConsumer** **(消费者)**

表示接受两个参数并且不返回结果的操作。

|  |
| --- |
| API:  public interface BiConsumer<T,U>{  ...  public void accept(T t, U u);  ...  }  注：此函数还有andThen()方法？？？ |

**Supplier(供应者)**

表示返回类型T的值的函数。

|  |
| --- |
| API:  public interface Supplier<T>{  ...  public T get();  ...  }  注：对应的专用函数  DoubleSupplier double getAsDouble();  IntSupplier int getAsInt();  LongSupplier long getAsLong(); |

**UnaryOperator (一元操作)**

表示接受参数并返回相同类型的结果的函数。

|  |
| --- |
| API:  public interface UnaryOperator<T>{  ...  public T apply(T t);  ...  }  注：对应的专用函数  DoubleUnaryOperator double applyAsDouble(double value)  IntUnaryOperator int applyAsInt(int value)  LongUnaryOperator long applyAsLong(long value) |

**BinaryOperator**

表示一个函数，它接受两个参数并返回相同类型的结果。

|  |
| --- |
| API:  public interface BinaryOperator<T>{  ...  public T apply(T t1, T t2);  ...  }  注：对应的专用函数  DoubleBinaryOperator double applyAsDouble(double left, double right)  IntBinaryOperator int applyAsInt(int left, int right)  LongBinaryOperator long applyAsLong(long left, long right) |

#### 交叉类型

Java 8引入了一种称为交集类型的新类型。

交叉类型是多种类型的交叉。

在两种类型之间使用Type1 & Type2，以表示Type1，Type2的交集的新类型。它是所有类型的子类型。

|  |
| --- |
| 示例1：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  MyInterface mi = (MyInterface & Sum)(x,y) -> x \* y;  System.***out***.println(mi); //此时mi对象不能调用Sum接口的sum方法。  }    **interface** MyInterface{ //这是一个非功能接口  }    @FunctionalInterface  **interface** Sum { //这是一个功能接口  **long** sum(**long** x, **long** y);  }  }  示例2：以下这种方式可以使lambda表达式可序列化。  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Serializable ser = (Serializable & Sum)(x,y) -> x \* y;  System.***out***.println(ser);  }    @FunctionalInterface  **interface** Sum {  **long** sum(**long** x, **long** y);  }  } |

#### 静态方法引用

**语法：**

Qualifier::MethodName

::两个连续的冒号充当分隔符。

MethodName是方法的名称。

Qualifier告诉在哪里找到方法引用。

|  |
| --- |
| 示例1：  String::length; 从String类引用length方法。这里String是限定符，length是方法名。  示例2：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  //注toBinaryString()方法为Integer类的一个静态方法。  //用常用的lambda表达式  Function<Integer,String> fun1 = x -> Integer.*toBinaryString*(x);  System.***out***.println(fun1.apply(10));  //使用静态方法引用  Function<Integer,String> fun2 = Integer::*toBinaryString*;  System.***out***.println(fun2.apply(10));  }    }  注：在调用重载静态方法时要注意参数类型的对应。 |

#### 实例方法调用

语法：

实例对象：：方法名

类名：：方法名

|  |
| --- |
| 示例1：实例对象：：方法名; 此处“hello”为String类的一个实例对象。  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  //使用常用的lambda表达式  Supplier<Integer> sup = () -> "hello".length();  System.***out***.println(sup.get()); //5    //使用实例方法引用  Supplier<Integer> sup1 = "hello"::length;  System.***out***.println(sup1.get()); //5  }  }  示例2：类名：：方法名；  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  //使用实例方法引用  Function<String, Integer> fun1 = String::length;  String name = "java";  **int** len = fun1.apply(name);  System.***out***.println(len);  }  } |

#### 构造函数引用

语法：ClassName::new

|  |
| --- |
| 示例1：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Supplier<String> sup = () -> **new** String();  String s = sup.get();    Supplier<String> sup1 = String::**new**;  String s1 = sup1.get();    Function<String,String> fun = str -> **new** String(str);  String s2 = fun.apply("java");    Function<String, String> fun1 = String::**new**;  String s3 = fun1.apply("java");  }  }  示例2：数组构造函数引用  ArrayTypeName::new  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  IntFunction<**int**[]> inf = size -> **new** **int**[size]; //一般Lambda表达式  **int**[] arr = inf.apply(5);  System.***out***.println(Arrays.*toString*(arr));    IntFunction<**int**[]> inf1 = **int**[]::**new**; //数组构造函数引用  **int**[] arr1 = inf1.apply(5);  System.***out***.println(Arrays.*toString*(arr1));  }  }  示例3：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Function<Integer,**int**[]> fun = size -> **new** **int**[size];  **int**[] arr = fun.apply(5);  System.***out***.println(Arrays.*toString*(arr));    Function<Integer,**int**[]> fun1 = **int**[]::**new**;  **int**[] arr1 = fun1.apply(5);  System.***out***.println(Arrays.*toString*(arr1));  }  } |

#### 通用方法引用

通过指定实际的类型参数来在泛型方法引用中使用通用方法。

语法：ClassName::<TypeName>methodName

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Function<String[], List<String>> fun = Arrays::<String>*asList*;  System.***out***.println(fun.apply(**new** String[] {"a","b"}));  }  } |

泛型构造函数引用的语法：ClassName<TypeName>::new

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Supplier<ArrayList<String>> sup = ArrayList<String> :: **new**;  ArrayList<String> list = sup.get();  System.***out***.println(list);  }  } |

#### 1.11 Lambda表达式作用域

lambda表达式与其外部方法具有相同的范围。 lambda表达式不会创建自己的作用域。

如果我们在lambda中使用关键字this和super表达式在方法中，它们的行为与我们在该方法中使用它们一样。

|  |
| --- |
| 示例1：  以下代码从lambda表达式输出this。this在lambda表达式中是指外部类不是lambda表达式本身。  **public** **class** Java8Tester {    **public** Java8Tester() {  Function<String, String> fun = x -> {  System.***out***.println(**this**.getClass().getName());  **return** x;  };  fun.apply("");  }    **public** **static** **void** main(String[] args) {  **new** Java8Tester(); // com.xzz.api.Java8Tester  }  } |

#### 1.12 Lambda变量捕获

可以访问局部变量, 局部变量用final定义或只初始化了一次，即首次初始化后没有被再次赋值。

|  |
| --- |
| 示例：  **public** **class** Java8Tester {    **public** **static** **void** main(String[] args) {  **final** String x = "hello"; //String x = "hello";  //x = "hi"; //将上名的final去掉后再将此注释去掉会报错，说明lambda表达式只能访问初始化一次的非final局部变量。  Function<String, String> fun = y -> {**return** x + " " + y;};  System.***out***.println(fun.apply("Java")); //hello Java  }  } |

不能改变lambda表达式之外的变量的值。

|  |
| --- |
| 示例：  **public** **class** Java8Tester {    **public** **static** **void** main(String[] args) {  String x = "hello";  Function<String, String> fun = y -> {  //x = "hi"; //打开此注释会报错，说明在lambda表达式中不能改其外变量的值。  **return** x + " " + y;  };  System.***out***.println(fun.apply("Java")); //hello Java  }  } |

lambda表达式中可以更改全局变量的值

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **static** String *x* = "hello";    **public** **static** **void** main(String[] args) {  Function<String, String> fun = y -> {  *x* = "hi"; //可以更改全局变量的值  **return** *x* + " " + y;  };  System.***out***.println(fun.apply("Java")); //hi Java  }  } |

#### 1.13 Lambda主体语句

可以使用break，continue，return和throw在lambda表达式主体中的语句。

不能使用跳转语句做非局部跳转。

使用break语句退出lambda表达式中的for循环，但是不能lambda表达式外的for循环。

|  |
| --- |
| 示例：  **public** **class** Java8Tester {    **public** **static** **void** main(String[] args) {  Function<String, String> fun = y -> {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(i);  **if** (i == 4) {  **break**;  }  }  **return** y + " lambda";  };  System.***out***.println(fun.apply("hi"));  }  } |
| 0  1  2  3  4  hi lambda |

### Optional类

Optional 类是一个可以为null的容器对象。

如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  Integer v1 = **null**;  Integer v2 = 10;  Optional<Integer> a = Optional.*ofNullable*(v1);  Optional<Integer> b = Optional.*of*(v2);  System.***out***.println(*sum*(a,b));  }    **public** **static** Integer sum(Optional<Integer> a, Optional<Integer> b) {    //isPresent() 判断值是否存在，存在返回true  System.***out***.println("第一个参数是否存在：" + a.isPresent());  System.***out***.println("第二个参数是否存在：" + b.isPresent());    //orElse(T other)如果存在返回存在的值，否则返回other  Integer v1 = a.orElse(0); //0  Integer v2 = b.orElse(0); //10    //get()返回存在的值  v2 = b.get(); //10  //v1 = a.get();//不存在会抛出NoSuchElementException    **return** v1 + v2;  }  } |

### 接口默认方法

Java 8 新增了接口的默认方法。

简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。

我们只需在方法名前面加个default关键字即可实现默认方法。

**为什么要有这个特性？**

首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题。

语法：

|  |
| --- |
| 示例：  **public** **interface** vehicle {  **default** **void** print() {  System.***out***.println("我是一辆车！");  }  } |

一个类实现了多个接口，但这些接口都有相同的默认方法，此时如何解决：

|  |
| --- |
| 现有两个接口：  **interface** vehicle {  **default** **void** print() {  System.***out***.println("我是一辆车！");  }  }  **interface** fourWheeler {  **default** **void** print() {  System.***out***.println("我是一辆四轮车！");  }  }   1. 重写默认方法   示例：  **class** car **implements** vehicle,fourWheeler {  **public** **void** print() {    }  }   1. 使用super指定实现类的默认接口方法   **class** car **implements** Vehicle,FourWheeler {  @Override  **public** **void** print() {  FourWheeler.**super**.print();  }    } |

静态默认方法：

|  |
| --- |
| **interface** FourWheeler {  **static** **void** print() {  System.***out***.println("我是一辆四轮车！");  }  } |

### Stream

#### 4.1 简介

Java 8 API添加了一个新的接口称为流Stream，可以让你以一种声明的方式处理数据。

Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流， 流在管道中传输， 并且可以在管道的节点上进行处理， 比如筛选， 排序，聚合等。

元素流在管道中经过中间操作（intermediate operation）的处理，最后由最终操作(terminal operation)得到前面处理的结果。

#### 4.2 什么是Stream

Stream（流）是一个来自数据源的元素队列并支持聚合操作。

元素是特定类型的对象，形成一个队列。 Java中的Stream并不会存储元素，而是按需计算。

流的来源。 可以是集合，数组，I/O channel， 产生器generator 等。

聚合操作: 类似SQL语句一样的操作， 比如filter, map, reduce, find, match, sorted等。

和以前的Collection操作不同， Stream操作还有两个基础的特征：

Pipelining: 中间操作都会返回流对象本身。 这样多个操作可以串联成一个管道， 如同流式风格（fluent style）。 这样做可以对操作进行优化， 比如延迟执行(laziness)和短路( short-circuiting)。

内部迭代： 以前对集合遍历都是通过Iterator或者For-Each的方式, 显式的在集合外部进行迭代， 这叫做外部迭代。 Stream提供了内部迭代的方式， 通过访问者模式(Visitor)实现。

#### 4.3 生成流

在 Java 8 中, 集合接口有两个方法来生成流：

stream() − 为集合创建串行流。

parallelStream() − 为集合创建并行流。

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  System.***out***.println("使用java7：");  //计算空字符串  List<String> strings = Arrays.*asList*("abc","","bc","efg","abcd","","jkl","abc","bc");  System.***out***.println("列表：" + strings);  **long** count = *getCountStringUsingJava7*(strings);  System.***out***.println("空字串数量为：" + count);    count = *getCountLength3UsingJava7*(strings);  System.***out***.println("字符串长度为3的数量为： " + count);    List<String> filterList = *deleteEmptyStringUsingJava7*(strings);  System.***out***.println("删除空字符串后的结果：" + filterList);    System.***out***.println("========================");  System.***out***.println("使用java8：");  System.***out***.println("列表：" + strings);  //filter 过滤，参数lambda表达式返回结果为false的过滤掉，相当于if(){}  //count() 返回符合条件的元素个数,这是一个终端操作  count = strings.stream().filter(string -> string.isEmpty()).count();  System.***out***.println("空字串数量为：" + count);    count = strings.stream().filter(string -> string.length() == 3).count();  System.***out***.println("字符串长度为3的数量为： " + count);  //collect对流进行可变的还原操作，Collectors 类实现了很多归约操作，例如将流转换成集合和聚合元素。  filterList = strings.stream().filter(str -> str.length() != 0).collect(Collectors.*toList*());  System.***out***.println("删除空字符串后的结果：" + filterList);    List<String> strings1 = Arrays.*asList*("a");  //allMatch() 流中的所有元素是否都匹配给出的条件，全部匹配则返回true  System.***out***.println("strings中是否全部为空串：" + strings.stream().allMatch(str -> str.isEmpty()));//当strings为[]时为会返回true  //anyMatch() 流中的是否有元素匹配给出的条件，有则返回true  System.***out***.println("strings中是否包含有空串：" + strings.stream().anyMatch(str -> str.isEmpty()));  //noneMatch() 指流中的任何一个元素都不匹配给出的条件，则返回true  System.***out***.println("strings中是否不含有空串：" + strings.stream().noneMatch(str -> str.isEmpty()));    //创建一个连接的流，其元素是第一个流的所有元素，其次是第二个流的所有元素。  System.***out***.println("concat连接操作：" + Stream.*concat*(strings.stream(), strings1.stream()).collect(Collectors.*toList*()));    //distinct 去重操作  System.***out***.println("distinct去重操作：" + strings.stream().distinct().collect(Collectors.*toList*()));    //empty 返回一个空的顺序流  System.***out***.println("空的顺序流：" + Stream.*empty*().collect(Collectors.*toList*()));    //返回流的第一个元素，如果流为空，则返回一个空的Optional，如果流没有确定的顺序，则可以返回任何元素。  System.***out***.println("返回一个空的流：" + strings.stream().findFirst().get());    //循环遍历流的每一个元素,forEach在并行中不保证按顺序输出。  strings.stream().forEach(System.***out***::println);  //循环遍历流的每一个元素,forEachOrdered一次只处理一个元素，按顺序输出。  strings.stream().forEachOrdered(System.***out***::println);    //返回由流元素组成的子流，子流的长度不超过limit参数  System.***out***.println(strings.stream().limit(3).collect(Collectors.*toList*()));    //map 返回流中的元素进行函数运算后的结果所组成的流  System.***out***.println("map: " + strings.stream().map(x -> x+1).collect(Collectors.*toList*()));    //Optional<T> max(Comparator<? super T> comparator)  //Optional<T> min(Comparator<? super T> comparator)    //返回一个顺序的给定元素的流  System.***out***.println("of: " + Stream.*of*(1,2).collect(Collectors.*toList*()));  //返回只有一个元素的流  System.***out***.println("of: " + Stream.*of*(1).collect(Collectors.*toList*()));    strings.stream().peek(e -> System.***out***.println("before:" + e)).filter(x -> x.length() > 2).peek(e -> System.***out***.println("after:"+e));    }    **private** **static** **int** getCountStringUsingJava7(List<String> strs) {  **int** count = 0;  **for** (String s : strs) {  **if** (s.isEmpty())  count ++;  }  **return** count;  }    **private** **static** **int** getCountLength3UsingJava7(List<String> strs) {  **int** count = 0;  **for** (String s : strs) {  **if** (s.length() == 3) {  count++;  }  }  **return** count;  }    **private** **static** List<String> deleteEmptyStringUsingJava7(List<String> strs) {  List<String> list = **new** ArrayList<String>();  **for** (String s : strs) {  **if**(s.length() != 0) {  list.add(s);  }  }  **return** list;  }  } |

### 新的日期API

Java 8 在 java.time 包下提供了很多新的 API。以下为两个比较重要的 API：

Local(本地) − 简化了日期时间的处理，没有时区的问题。

Zoned(时区) − 通过制定的时区处理日期时间。

新的java.time包涵盖了所有处理日期，时间，日期/时间，时区，时刻（instants），过程（during）与时钟（clock）的操作。

#### 获取当前时间及年月日时分秒星期 now() , getxxxx()

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  //now() 获取当前的系统时间  LocalDateTime localdt = LocalDateTime.*now*();  System.***out***.println("当前时间：" + localdt); //当前时间：2018-02-27T10:28:23.239    //获取时期  LocalDate localDate = localdt.toLocalDate();  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27  LocalDate localDate1 = LocalDate.*now*();  System.***out***.println("localDate1: " + localDate1); //localDate1: 2018-02-27    //获取时间  LocalTime localTime = localdt.toLocalTime();  System.***out***.println("localTime: " + localTime); //localTime: 10:28:23.239  LocalTime localTime1 = LocalTime.*now*();  System.***out***.println("localTime1: " + localTime1); //localTime1: 10:28:23.239    ZonedDateTime zoneddatetime = ZonedDateTime.*now*();  System.***out***.println("ZonedDateTime: " + zoneddatetime); //ZonedDateTime: 2018-02-27T10:30:26.686+08:00[Asia/Shanghai]    //获取年月日时分秒  //年返回的是int型的值  System.***out***.println("年： " + localdt.getYear());  System.***out***.println("年： " + localDate.getYear());  System.***out***.println("年： " + zoneddatetime.getYear());    //月份返回的是一个Month对象  System.***out***.println("月： " + localdt.getMonth());  System.***out***.println("月： " + localDate.getMonthValue());  System.***out***.println("月： " + zoneddatetime.getMonth().getValue());    //月中的天  System.***out***.println("日： " + localdt.getDayOfMonth());  System.***out***.println("日： " + localDate.getDayOfMonth());  System.***out***.println("日： " + zoneddatetime.getDayOfMonth());  //年中的天  System.***out***.println("日： " + localdt.getDayOfYear());  System.***out***.println("日： " + localDate.getDayOfYear());  System.***out***.println("日： " + zoneddatetime.getDayOfYear());    //星期 返回的是week对象  System.***out***.println("星期： " + localdt.getDayOfWeek());  System.***out***.println("星期： " + localDate.getDayOfWeek().getValue());  System.***out***.println("星期： " + zoneddatetime.getDayOfWeek().getValue());    //时  System.***out***.println("时： " + localdt.getHour());  System.***out***.println("时： " + localTime.getHour());  System.***out***.println("时： " + zoneddatetime.getHour());    //分  System.***out***.println("分： " + localdt.getMinute());  System.***out***.println("分： " + localTime.getMinute());  System.***out***.println("分： " + zoneddatetime.getMinute());    //秒  System.***out***.println("秒： " + localdt.getSecond());  System.***out***.println("秒： " + localTime.getSecond());  System.***out***.println("秒： " + zoneddatetime.getSecond());  //获取时间毫秒值  **long** time = LocalDateTime.*now*().atZone(ZoneId.*systemDefault*()).toInstant().toEpochMilli();  System.***out***.println("time: " + time);  }  } |

#### 构造时间: of()

|  |
| --- |
| **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {  LocalDate localDate = LocalDate.*of*(2018, 2, 27);  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27    LocalDateTime localDateTime = LocalDateTime.*of*(2018, 2, 27, 11, 7);  System.***out***.println("localDateTime： " + localDateTime); //localDateTime： 2018-02-27T11:07    LocalTime localTime = LocalTime.*of*(11, 8, 59);  System.***out***.println("localTime: " + localTime); //localTime: 11:08:59  }  } |

#### 5.4 从指定参数中派生datetime对象 from()

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {    LocalDateTime localDateTime = LocalDateTime.*of*(2018, 2, 27, 11, 7);  System.***out***.println("localDateTime： " + localDateTime); //localDateTime： 2018-02-27T11:07  LocalDate localDate = LocalDate.*from*(localDateTime);  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27    LocalTime localTime = LocalTime.*from*(localDateTime);  System.***out***.println("localTime: " + localTime); //localTime: 11:07  }  }  注： 如果参数对象中不包含有派生子对象，则会抛出异常。 |

#### 5.5 修改指定字段的值 withXXX()

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {    LocalDate localDate = LocalDate.*now*();  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27    localDate = localDate.withYear(2017);  System.***out***.println("localDate: " + localDate); //localDate: 2017-02-27    localDate = localDate.withMonth(10);  System.***out***.println("localDate: " + localDate); //localDate: 2017-10-27    localDate = localDate.withDayOfMonth(10);  System.***out***.println("localDate: " + localDate); //localDate: 2017-10-10  }  } |

#### 5.6 给日期补上时间，或给时间补上日期 atXXX()

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {    LocalDate localDate = LocalDate.*now*();  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27    LocalDateTime localDateTime = localDate.atStartOfDay();  System.***out***.println("localDateTime: " + localDateTime); //localDateTime: 2018-02-27T00:00    localDateTime = localDate.atTime(8, 8, 8);  System.***out***.println("localDateTime: " + localDateTime); //localDateTime: 2018-02-27T08:08:08    LocalTime localTime = LocalTime.*now*();  localDateTime = localTime.atDate(localDate);  System.***out***.println("localDateTime: " + localDateTime); //localDateTime: 2018-02-27T11:44:40.366  }  } |

#### 5.7 日期计算 plusXXX()相加 minusXXXX()相减

|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {    LocalDate localDate = LocalDate.*now*();  System.***out***.println("localDate: " + localDate); //localDate: 2018-02-27    localDate = localDate.minusYears(3);  System.***out***.println("localDate: " + localDate); //localDate: 2015-02-24    localDate = localDate.minusDays(3);  System.***out***.println("localDate: " + localDate); //localDate: 2015-02-24    localDate = localDate.plusYears(4);  System.***out***.println("localDate: " + localDate); //localDate: 2019-02-24    localDate = localDate.plusDays(6);  System.***out***.println("localDate: " + localDate); //localDate: 2019-03-02    }  } |

#### 5.8 字符串解析为日期 parse()

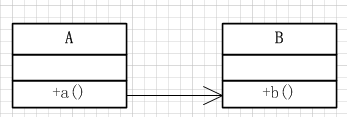
|  |
| --- |
| 示例：  **public** **class** Java8Tester {  **public** **static** **void** main(String[] args) {    //此时月日要由两位数组成  LocalDate localDate = LocalDate.*parse*("2017-08-01");  System.***out***.println("localDate: " + localDate);    DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("yyyy-M-d");  localDate = LocalDate.*parse*("2017-8-1", formatter);  System.***out***.println("localDate: " + localDate);    //注意日和时之间要加上一个T，所有字段都要是两位数  LocalDateTime localDateTime = LocalDateTime.*parse*("2017-08-10T10:10:10");  System.***out***.println("localDateTime" + localDateTime);    formatter = DateTimeFormatter.*ofPattern*("yyyy-M-d H:m:s");  localDateTime = LocalDateTime.*parse*("2017-1-18 18:1:10", formatter);  System.***out***.println("localDateTime" + localDateTime);  //将一个long值转换为日期字符串  DateTimeFormatter df= DateTimeFormatter.*ofPattern*("YYYY-MM-dd HH:mm:ss.SSS");  String localDateTime1 = df.format(LocalDateTime.*ofInstant*(Instant.*ofEpochMilli*(1519713299470L),ZoneId.*of*("Asia/Shanghai")));  System.***out***.println("localDateTime1: " + localDateTime1);  //将日期转换为时间毫秒值  **long** time = LocalDateTime.*now*().atZone(ZoneId.*systemDefault*()).toInstant().toEpochMilli();  System.***out***.println("time: " + time);  }  } |

## 二．java回调机制

**模块间调用**

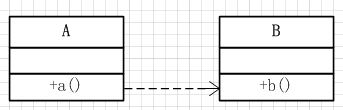
在一个应用系统中，无论使用何种语言开发，必然存在模块之间的调用，调用的方式分为几种：

（1）同步调用



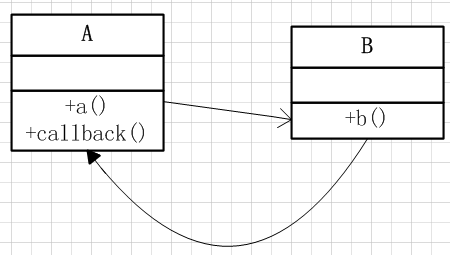
同步调用是最基本并且最简单的一种调用方式，类A的方法a()调用类B的方法b()，一直等待b()方法执行完毕，a()方法继续往下走。**这种调用方式适用于方法b()执行时间不长的情况**，因为b()方法执行时间一长或者直接阻塞的话，a()方法的余下代码是无法执行下去的，这样会造成整个流程的阻塞。

（2）异步调用



异步调用是为了解决同步调用可能出现阻塞，导致整个流程卡住而产生的一种调用方式。**类A的方法方法a()通过新起线程的方式调用类B的方法b()，代码接着直接往下执行**，这样无论方法b()执行时间多久，都不会阻塞住方法a()的执行。但是这种方式，由于方法a()不等待方法b()的执行完成，在方法a()需要方法b()执行结果的情况下（视具体业务而定，有些业务比如启异步线程发个微信通知、刷新一个缓存这种就没必要），必须通过一定的方式对方法b()的执行结果进行监听。在Java中，可以使用Future+Callable的方式做到这一点，具体做法可以参见我的这篇文章[Java多线程21：多线程下其他组件之CyclicBarrier、Callable、Future和FutureTask](http://www.cnblogs.com/xrq730/p/4872722.html)。

（3）回调



最后是回调，回调的思想是：

* **类A的a()方法调用类B的b()方法**
* **类B的b()方法执行完毕主动调用类A的callback()方法**

这样一种调用方式组成了上图，也就是一种**双向**的调用方式。

**编程思路：**

|  |
| --- |
| 场景：老师问学生一个问题，学生思考后回答老师，老师公布答案并判断学生是否回答正确？   1. 定义回调接口Callback, 并定义callback(String answer)方法 2. 定义一个Teacher接口，接口中定义一个ask()方法，表示老师要问学生问题, 同时继承Callback接口。 3. 定义一个Student接口，接口中定义一个回答问题的方法，方法中要有Callback接口类型参数，以便于调用回调方法。 4. 定义一个TeacherB类，即具体的老师，定义一个构造方法Teacher(Student student)，即老师要指定一位学生。实现回调方法，实现问题的方法（在此方法中调用学生的回答问题方法，并将其**自身对象作为参数**传入） 5. 定义一个StudentA类，即具体的学生，实现回答问题的方法并调用回调方法。   以上将老师和学生抽象出来的目的是不管是哪个老师提问还是哪个学生回答问题，都不会影响回调方法的定义。 |

**代码示例**

接下来看一下回调的代码示例，代码模拟的是这样一种场景：老师问学生问题，学生思考完毕回答老师。

首先定义一个回调接口，只有一个方法tellAnswer(int answer)，即学生思考完毕告诉老师答案：

/\*\*

\* 回调接口

\*/

public interface Callback {

public void tellAnswer(int answer);

}

定义一个老师对象，实现Callback接口：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 老师对象，原文出处http://www.cnblogs.com/xrq730/p/6424471.html

3 \*/

4 public class Teacher implements Callback {

5

6 private Student student;

7

8 public Teacher(Student student) {

9 this.student = student;

10 }

11

12 public void askQuestion() {

13 student.resolveQuestion(**this**);

14 }

15

16 @Override

17 public void tellAnswer(int answer) {

18 System.out.println("知道了，你的答案是" + answer);

19 }

20

21 }

[复制代码](javascript:void(0);)

老师对象有两个public方法：

（1）回调接口tellAnswer(int answer)，即学生回答完毕问题之后，老师要做的事情

（2）问问题方法askQuestion()，即向学生问问题

接着定义一个学生接口，学生当然是解决问题，但是接收一个Callback参数，这样学生就知道解决完毕问题向谁报告：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 学生接口，原文出处http://www.cnblogs.com/xrq730/p/6424471.html

3 \*/

4 public interface Student {

5

6 public void resolveQuestion(Callback callback);

7

8 }

[复制代码](javascript:void(0);)

最后定义一个具体的学生叫Ricky：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 一个名叫Ricky的同学解决老师提出的问题，原文出处http://www.cnblogs.com/xrq730/p/6424471.html

3 \*/

4 public class Ricky implements Student {

5

6 @Override

7 public void resolveQuestion(Callback callback) {

8 // 模拟解决问题

9 try {

10 Thread.sleep(3000);

11 } catch (InterruptedException e) {

12

13 }

14

15 // 回调，告诉老师作业写了多久

16 callback.tellAnswer(3);

17 }

18

19 }

[复制代码](javascript:void(0);)

在解决完毕问题之后，第16行向老师报告答案。

写一个测试类，比较简单：

[复制代码](javascript:void(0);)

1 /\*\*

2 \* 回调测试，原文出处http://www.cnblogs.com/xrq730/p/6424471.html

3 \*/

4 public class CallbackTest {

5

6 @Test

7 public void testCallback() {

8 Student student = new Ricky();

9 Teacher teacher = new Teacher(student);

10

11 teacher.askQuestion();

12

13 }

14

15 }

[复制代码](javascript:void(0);)

代码运行结果就一行：

知道了，你的答案是3

简单总结、分析一下这个例子就是：

（1）老师调用学生接口的方法resolveQuestion，向学生提问

（2）学生解决完毕问题之后调用老师的回调方法tellAnswer

这样一套流程，构成了一种双向调用的关系。

**代码分析**

分析一下上面的代码，上面的代码我这里做了两层的抽象：

（1）将老师进行抽象

* 将老师进行抽象之后，对于学生来说，就不需要关心到底是哪位老师询问我问题，只要我根据询问的问题，得出答案，然后告诉提问的老师就可以了，即使老师换了一茬又一茬，对我学生而言都是没有任何影响的

（2）将学生进行抽象

* 将学生进行抽象之后，对于老师这边来说就非常灵活，因为老师未必对一个学生进行提问，可能同时对Ricky、Jack、Lucy三个学生进行提问，这样就可以将成员变量Student改为List<Student>，这样在提问的时候遍历Student列表进行提问，然后得到每个学生的回答即可

这个例子是一个典型的体现接口作用的例子，之所以这么说是因为我想到有些朋友可能不太明白接口的好处，不太明白接口好处的朋友可以重点看一下这个例子，多多理解。

总结起来，**回调的核心就是回调方将本身即this传递给调用方**，这样调用方就可以在调用完毕之后告诉回调方它想要知道的信息。回调是一种思想、是一种机制，至于具体如何实现，如何通过代码将回调实现得优雅、实现得可扩展性比较高，一看开发者的个人水平，二看开发者对业务的理解程度。

**同步回调与异步回调**

上面的例子，可能有人会提出这样的疑问：

这个例子需要用什么回调啊，使用同步调用的方式，学生对象回答完毕问题之后直接把回答的答案返回给老师对象不就好了？

这个问题的提出没有任何问题，可以从两个角度去理解这个问题。

首先，老师不仅仅想要得到学生的答案怎么办？可能这个老师是个更喜欢听学生解题思路的老师，在得到学生的答案之前，老师更想先知道学生姓名和学生的解题思路，当然有些人可以说，那我可以定义一个对象，里面加上学生的姓名和解题思路不就好了。这个说法在我看来有两个问题：

（1）如果老师想要的数据越来越多，那么返回的对象得越来越大，而使用回调则可以进行数据分离，将一批数据放在回调方法中进行处理，至于哪些数据依具体业务而定，如果需要增加返回参数，直接在回调方法中增加即可

（2）无法解决老师希望得到学生姓名、学生解题思路先于学生回答的答案的问题

因此我认为简单的返回某个结果确实没有必要使用回调而可以直接使用同步调用，但是如果**有多种数据需要处理且数据有主次之分**，使用回调会是一种更加合适的选择，优先处理的数据放在回调方法中先处理掉。

另外一个理解的角度则更加重要，就是标题说的同步回调和**异步回调**了。例子是一个同步回调的例子，意思是老师向Ricky问问题，Ricky给出答案，老师问下一个同学，得到答案之后继续问下一个同学，这是一种正常的场景，但是如果我把场景改一下：

老师并不想One-By-One这样提问，而是同时向Ricky、Mike、Lucy、Bruce、Kate五位同学提问，让同学们自己思考，哪位同学思考好了就直接告诉老师答案即可。

这种场景相当于是说，同学思考完毕完毕问题要有一个办法告诉老师，有两个解决方案：

（1）使用Future+Callable的方式，等待异步线程执行结果，这相当于就是同步调用的一种变种，因为其本质还是方法返回一个结果，即学生的回答

（2）使用异步回调，同学回答完毕问题，调用回调接口方法告诉老师答案即可。由于老师对象被抽象成了Callback接口，因此这种做法的扩展性非常好，就像之前说的，即使老师换了换了一茬又一茬，对于同学来说，只关心的是调用Callback接口回传必要的信息即可

## 三．JAVA I/O

### 1. 分类：

a. 基于字节操作的IO接口：InputStream 和 OutputStream

b. 基于字符操作的IO接口：Writer 和 Reader

c. 基于磁盘操作的IO接口：File

d. 基于网络操作的IO接口：Socket

### 2. 接口的类层次图

ObjectInputStream

|  |
| --- |
| InputStream类层次图  InputStream  PipedInputStream  ByteArrayInputStream  FileInputStream  FilterInputStream  SocketInputStream  DataInputStream  BufferedInputStream  InflaterInputStream  ZipInputStream |

|  |
| --- |
| OutputStream类层次图  OutputStream  FileOutputStream  ByteArrayOutputStream  PipedOutputStream  FilterOutputStream  ZipOutputStream  PrintStream  SocketOutputStream  DataOutputStream |

|  |
| --- |
| Writer  Writer |