**JDK1.8新特性介绍**

[一 前言 3](#_Toc113318701)

[二 Lambda表达式 3](#_Toc113318702)

[2.1 Lambda概念 3](#_Toc113318705)

[2.2 无参数，无返回 5](#_Toc113318706)

[2.3 有一个参数，无返回 5](#_Toc113318707)

[2.4 有多个参数，有返回，有一条lambda体语句 5](#_Toc113318708)

[2.5 有多个参数，有返回，有多条lambda体语句 6](#_Toc113318709)

[三 函数式接口 7](#_Toc113318710)

[3.1 函数式接口： 7](#_Toc113318714)

[3.2 格式 7](#_Toc113318715)

[3.3 @FunctionalInterface注解 7](#_Toc113318716)

[3.4 使用函数式接口作为方法的参数 9](#_Toc113318717)

[3.5 使用函数式接口作为方法的返回值 10](#_Toc113318718)

[3.6 常用函数式接口 11](#_Toc113318719)

[3.6.1 提供类型：Supplier<T>接口 11](#_Toc113318720)

[3.6.2 消费类型：Consumer<T>接口 12](#_Toc113318721)

[3.6.3 断定类型：Predicate<T>接口 15](#_Toc113318722)

[3.6.4 转换类型：Function<T>接口 18](#_Toc113318723)

[3.6.5 其他函数式接口 20](#_Toc113318724)

[四 方法引用 21](#_Toc113318725)

[4.1 方法引用概念 21](#_Toc113318730)

[4.2 通过对象名引用成员方法 22](#_Toc113318731)

[4.3 通过类名称引用静态方法 23](#_Toc113318732)

[4.4 通过super引用成员方法 24](#_Toc113318733)

[4.5 通过this引用成员方法 25](#_Toc113318734)

[4.6 类的构造器引用 25](#_Toc113318735)

[4.7 数组的构造器引用 26](#_Toc113318736)

[五 StreamAPI 27](#_Toc113318737)

[5.1 Stream概念 27](#_Toc113318743)

[5.2 获取Stream流 28](#_Toc113318744)

[5.2.1 根据Collection获取流 28](#_Toc113318745)

[5.2.2 根据Map获取流 29](#_Toc113318746)

[5.2.3 根据数组获取流 30](#_Toc113318747)

[5.2.4 获取无限流 30](#_Toc113318748)

[5.3 Stream流常用方法 30](#_Toc113318749)

[5.3.1 逐一处理：forEach 31](#_Toc113318750)

[5.3.2 过滤：filter 31](#_Toc113318751)

[5.3.3 映射：map 31](#_Toc113318752)

[5.3.4 flatMap 32](#_Toc113318753)

[5.3.5 规约reduce 33](#_Toc113318754)

[5.3.6 组合：concat 34](#_Toc113318755)

[5.3.7 转换：Collect 34](#_Toc113318756)

[5.3.8 其他方法 35](#_Toc113318757)

[5.4 并行流和顺序流 35](#_Toc113318758)

[六 Optional容器 36](#_Toc113318759)

[6.1 Optional概念 37](#_Toc113318766)

[6.2 Optional使用示例 37](#_Toc113318767)

[七 JDK8时间日期API 39](#_Toc113318768)

[7.1 旧版日期时间API存在的问题 39](#_Toc113318776)

[7.2 新日期时间 API 介绍 41](#_Toc113318777)

[7.3 API使用说明 42](#_Toc113318778)

[7.3.1 日期和时间类 42](#_Toc113318779)

[7.3.2 日期时间格式化与解析 42](#_Toc113318780)

[7.3.3 设置日期时间的时区 42](#_Toc113318781)

[八 其他特性 42](#_Toc113318782)

[九 特别推荐参考： 43](#_Toc113318783)

# 前言

本文档主要介绍了JDK1.8版本中的一些重要的新特性，仅供参考，涉及特性知识点内容如下：

* + **Lambda表达式**
  + **函数式接口**
  + **方法引用和构造器调用**
  + **Stream API**
  + **Optional容器**
  + **新时间日期API**
  + **其他特性**

# Lambda表达式

lambda[表达式](https://so.csdn.net/so/search?q=%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1001.2101.3001.7020)本质上是一段匿名内部类，也可以是一段可以传递的代码，lambda表达式为匿名内部类的简写，类似于匿名内部类的语法糖；但又区别于匿名内部类



## Lambda概念

**匿名内部类特点：**

* 基于多态（多数基于接口编程）
* 实现类无需名称
* 允许多个抽象方法

Lambda的语法简洁，没有面向对象复杂的束缚，其特点如下：

1. 使用Lambda必须有接口，并且**接口中有且仅有一个抽象方法**。  
   只有当接口中的抽象方法存在且唯一时，才可以使用Lambda，但排除接口默认方法以及声明中覆盖Object的公开方法。
2. 使用Lambda必须具有**上下文推断**。  
   也就是方法的参数或局部变量类型必须为Lambda对应的接口类型，才能使用Lambda作为该接口的实例。

**标准格式：**

由三部分组成：一些参数、一个箭头、一段代码

|  |
| --- |
| 1. (参数列表)->{一些重要方法的代码}; 2. ():接口中抽象方法的参数列表，没有参数，就空着；有参数就写出参数，多个参数用逗号分隔。 3. ->：传递：把参数传递给方法体{} 4. {}:重写接口的抽象方法的方法体   **注意：**  箭头操作符的左侧对应接口中参数列表(lambda表达式的参数列表)， 箭头右侧为该抽象方法的实现（lambda表达式所需执行的功能）。 |

lambda优化写法：可以推导的，都可以省略(凡是能根据上下文推导出来的内容，都可以省略不写)：

|  |
| --- |
| 1. （参数列表）：括号中参数列表的数据类型，可以省略不写，因为JVM编译器通过上下文推断出数据类型，即**类型推断** 2. （参数列表）：括号中的参数只有一个，那么类型和（）都可以省略 3. {一些代码} ：如果{}中的代码只有一行，无论是否有返回值，都可以省略（{}，return，分号）   **注意：**  要省略{},return,分号 必须一起省略 |

代码例：

|  |
| --- |
| public class MyLambda {  public static void main(String[] args) {  //常规写法  new Thread(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName()+"新线程创建了");  }  }).start();    //使用Lambda  new Thread(()->{  System.out.println(Thread.currentThread().getName()+"新线程创建了");  }).start();    //优化lambda  new Thread(()->System.out.println(Thread.currentThread().getName()+"新线程创建了")).start();  }  } |

## 无参数，无返回

例：()->System.out.println("hello lambda")

|  |
| --- |
| public interface Cook {  public abstract void makeFood();  }  public class Demo01Cook {  public static void main(String[] args) {  invokeCook(new Cook() {  public void makeFood() {  System.out.println("做饭。。。");  }  });  //使用Lambda  invokeCook(()->{  System.out.println("做饭。。。");  });  //优化lambda  invokeCook(()-> System.out.println("做饭。。。"));  }  public static void invokeCook(Cook cook){  cook.makeFood();  }  } |

## 有一个参数，无返回

例：x->System.out.println("hello lambda")

|  |
| --- |
| import java.util.function.Consumer;    public class Demo2 {  public static void main(String[] args) {  Consumer<String> consumer = x-> System.out.println(x);  consumer.accept("有参数无返回");  }  } |

## 有多个参数，有返回，有一条lambda体语句

例：(Person p1,Person p2)->{ return p1.getAge()-p2.getAge(); }

|  |
| --- |
| package com.hcx.lambda;    import java.util.Arrays;  import java.util.Comparator;  public class MyArrays {  public class Person{  private String name;  private Integer age;  }    public static void main(String[] args) {  Person[] arr = {new Person("陈奕迅",40),  new Person("钟汉良",39),  new Person("杨千嬅",38)};  //对年龄进行排序  Arrays.sort(arr, new Comparator<Person>() {  @Override  public int compare(Person o1, Person o2) {  return o1.getAge()-o2.getAge();  }  });  //使用lambda  Arrays.sort(arr,(Person p1,Person p2)->{  return p1.getAge()-p2.getAge();  });  //优化lambda  Arrays.sort(arr,(p1,p2)->p1.getAge()-p2.getAge());  Arrays.sort(arr,Comparator.comparing(Person::getAge));    for (Person p:arr){  System.out.println(p);  }  }  } |

## 有多个参数，有返回，有多条lambda体语句

例：(x，y) -> {System.out.println(“xxx”);return xxxx;}；

|  |
| --- |
| public static void main(String[] args) {  Person[] arr = {new Person("陈奕迅",40),  new Person("钟汉良",39),  new Person("杨千嬅",38)};  //对年龄进行排序  //优化lambda  Arrays.sort(arr,(p1,p2)->{  System.out.println("比较用户");  return p1.getAge()-p2.getAge());  }  Arrays.sort(arr,Comparator.comparing(Person::getAge));    for (Person p:arr){  System.out.println(p);  }  } |

# 函数式接口



## 函数式接口：

有且仅有一个抽象方法的接口。

函数式接口，即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数式接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导。

## 格式

|  |
| --- |
| 修饰符 interface 接口名称 {  public abstract 返回值类型 方法名称(可选参数信息);  // 其他非抽象方法内容  }  **注意**: 抽象方法的 public abstract 是可以省略的 |

## @FunctionalInterface注解

与 @Override 注解的作用类似，Java 8中专门为函数式接口引入了一个新的注解： @FunctionalInterface 。用于函数式接口类型声明的信息注解类型，这些接口的实例被Lambda表达式、方法引用或构造器引用创建。函数式接口只能有一个抽象方法，但排除接口默认方法以及声明中覆盖Object的公开方法。

**@FunctionalInterface**注解是定义一个lambda表达式的基础， 即是否是函数式接口可以标注也可以不标注。函数式接口必须有一个精确的抽象方法，但排除以下两种：

① java8的default，他们不属于抽象方法。

② 如果该接口声明的一个抽象方法覆盖了任意一个object的方法，也排除掉。

|  |
| --- |
| @FunctionalInterface  public interface MyFunctionInterface {  void myMethod();  //唯一个抽象方法  void method();  //排除用default修饰的方法  default void method1(){  }  //排除Ojbect下的方法  int hashCode();  } |

注意：@FuncationlInterface不能标注在注解、类以及枚举上。一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会报错。需要注意的是，即使不使用该注解，只要满足函数式接口的定义，这仍然是一个函数式接口，使用起来都一样。

例：

|  |
| --- |
| @FunctionalInterface  public interface MyFunctionalInterface {  void method();  }  public class MyFunctionalInterfaceImpl implements MyFunctionalInterface{  @Override  public void method() {  }  }  /\*\*  \* 函数式接口的使用:一般可以作为方法的参数和返回值类型  \*/  public class Demo {  //定义一个方法,参数使用函数式接口MyFunctionalInterface  public static void show(MyFunctionalInterface myInter){  myInter.method();  }    public static void main(String[] args) {  //调用show方法,方法的参数是一个接口,所以可以传递接口的实现类对象  show(new MyFunctionalInterfaceImpl());    //调用show方法,方法的参数是一个接口,所以我们可以传递接口的匿名内部类  show(new MyFunctionalInterface() {  @Override  public void method() {  System.out.println("使用匿名内部类重写接口中的抽象方法");  }  });    //调用show方法,方法的参数是一个函数式接口,所以我们可以Lambda表达式  show(()->{  System.out.println("使用Lambda表达式重写接口中的抽象方法");  });    //简化Lambda表达式  show(()-> System.out.println("使用Lambda表达式重写接口中的抽象方法"));  }  } |

## 使用函数式接口作为方法的参数

|  |
| --- |
| public class Demo01Runnable {  //定义一个方法startThread,方法的参数使用函数式接口Runnable  public static void startThread(Runnable run){  //开启多线程  new Thread(run).start();  }  public static void main(String[] args) {  //调用startThread方法,方法的参数是一个接口,那么我们可以传递这个接口的匿名内部类  startThread(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName()+"-->"+"线程启动了");  }  });  //调用startThread方法,方法的参数是一个函数式接口,所以可以传递Lambda表达式  startThread(()->{  System.out.println(Thread.currentThread().getName()+"-->"+"线程启动了");  });  //优化Lambda表达式  startThread(()->System.out.println(Thread.currentThread().getName()+"-->"+"线程启动了"));  }  } |

## 使用函数式接口作为方法的返回值

|  |
| --- |
| import java.util.Arrays;  import java.util.Comparator;    /\*  如果一个方法的返回值类型是一个函数式接口，那么就可以直接返回一个Lambda表达式。  当需要通过一个方法来获取一个java.util.Comparator接口类型的对象作为排序器时,就可以调该方法获取。  \*/  public class Demo02Comparator {  //定义一个方法,方法的返回值类型使用函数式接口Comparator  public static Comparator<String> getComparator(){  //方法的返回值类型是一个接口,那么我们可以返回这个接口的匿名内部类  /\*return new Comparator<String>() {  @Override  public int compare(String o1, String o2) {  //按照字符串的降序排序  return o2.length()-o1.length();  }  };\*/  //方法的返回值类型是一个函数式接口,所以我们可以返回一个Lambda表达式  /\*return (String o1, String o2)->{  //按照字符串的降序排序  return o2.length()-o1.length();  };\*/  //继续优化Lambda表达式  return (o1, o2)->o2.length()-o1.length();  }  public static void main(String[] args) {  //创建一个字符串数组  String[] arr = {"a","bb","ccc","dddd"};  //输出排序前的数组  System.out.println(Arrays.toString(arr));  //调用Arrays中的sort方法,对字符串数组进行排序  Arrays.sort(arr,getComparator());  //输出排序后的数组  System.out.println(Arrays.toString(arr));  }  } |

每次都要声明一个接口，写一个抽象方法，然后再用这个接口作为参数去用lambda实现。当果不需要！这个新特性就是为了我们使用简单的呀，所以java已经内置了一堆函数式接口了。

**下面是JDK8内置四大核心函数接口：**

| **函数式接口** | **参数类型** | **返回类型** | **用途** |
| --- | --- | --- | --- |
| Supplier<T> 供给型 | 无 | T | 返回类型为T的对象，方法：T get() |
| Consumer<T>消费型 | T | void | 对类型为T的对象应用操作，方法：void accept(T t) |
| Predicate<T> 断定型 | T | boolean | 确定类型为T的对象是否满足某种约束，返回布尔值，方法：boolean test(T t) |
| Function<T,R>函数型 | T | R | 对类型为T的对象应用操作，并返回R类型的对象，方法：R apply(T,t) |

## 常用函数式接口

## 提供类型：Supplier<T>接口

**特点：只出不进，作为方法/构造参数、方法返回值**

java.util.function.Supplier<T>接口仅包含一个无参的方法：T get()。用来获取一个泛型参数指定类型的对象数据。

Supplier<T>接口被称之为生产型接口,指定接口的泛型是什么类型,那么接口中的get方法就会生产什么类型的数据

|  |
| --- |
| import java.util.function.Supplier;  /\*\*  \* MySupplier.  \*/  public class MySupplier {    public static String getString(Supplier<String> supplier){  return supplier.get();  }    public static void main(String[] args) {  getString(new Supplier<String>() {  @Override  public String get() {  return null;  }  });  String s = getString(()->"Eason");  System.out.println(s);  }  }  import java.util.function.Supplier;    /\*\*  \* Created by hongcaixia on 2019/10/29.  \*/  public class GetMax {    public static int getMaxNum(Supplier<Integer> supplier){  return supplier.get();  }    public static void main(String[] args) {  int[] arr = {-1,0,1,2,3};  int maxValue = getMaxNum(()->{  int max = arr[0];  for(int i=0;i<arr.length;i++){  if(arr[i]>max){  max = arr[i];  }  }  return max;  });  System.out.println("数组元素的最大值是："+maxValue);  }  } |

## 消费类型：Consumer<T>接口

**特点：只进不出，作为方法/构造参数**

java.util.function.Consumer<T>接口则正好与Supplier接口相反，它不是生产一个数据，而是消费一个数据，其数据类型由泛型决定。Consumer接口中包含抽象方法void accept(T t)，意为消费一个指定泛型的数据。

Consumer接口是一个消费型接口,泛型执行什么类型,就可以使用accept方法消费什么类型的数据至于具体怎么消费(使用),需要自定义

* **accept方法**

|  |
| --- |
| import java.util.function.Consumer;    /\*\*  \* Created by hongcaixia on 2019/10/29.  \*/  public class MyConsumer {  public static void method(String name, Consumer<String> consumer){  consumer.accept(name);  }  public static void main(String[] args) {  method("小哇", new Consumer<String>() {  @Override  public void accept(String s) {  System.out.println("s是："+s);  }  });  method("小哇",(name)->{  String s = new StringBuffer(name).reverse().toString();  System.out.println("s是："+s);  });  }  } |

* **andThen:**

Consumer接口的默认方法andThen。

作用：需要两个Consumer接口,可以把两个Consumer接口组合到一起,在对数据进行消费

|  |
| --- |
| import java.util.function.Consumer;  /\*\*  \* AndThen  \*/  public class AndThen {  public static void method(String s, Consumer<String> consumer1,  Consumer<String> consumer2){  //consumer1.accept(s);  //consumer2.accept(s);  //使用andThen方法,把两个Consumer接口连接到一起,在消费数据  //con1连接con2,先执行con1消费数据,在执行con2消费数据  consumer1.andThen(consumer2).accept(s);  }  public static void main(String[] args) {  method("Hello",  (t)-> System.out.println(t.toUpperCase()), //消费方式:把字符串转换为大写输出  (t)-> System.out.println(t.toLowerCase()));//消费方式:把字符串转换为小写输出    method("Hello", new Consumer<String>() {  @Override  public void accept(String s) {  System.out.println(s.toUpperCase());  }},new Consumer<String>() {  @Override  public void accept(String s1) {  System.out.println(s1.toUpperCase());  }  });  }  } |

按照格式“姓名：XX。性别：XX。”的格式将信息打印；要求将打印姓名的动作作为第一个Consumer接口的Lambda实例，将打印性别的动作作为第二个Consumer接口的Lambda实例，将两个Consumer接口按照顺序“拼接”到一起。

|  |
| --- |
| public class DemoTest {  //定义一个方法,参数传递String类型的数组和两个Consumer接口,泛型使用String  public static void printInfo(String[] arr, Consumer<String> con1,  Consumer<String> con2){  //遍历字符串数组  for (String message : arr) {  //使用andThen方法连接两个Consumer接口,消费字符串  con1.andThen(con2).accept(message);  }  }    public static void main(String[] args) {  //定义一个字符串类型的数组  String[] arr = { "陈奕迅,男", "钟汉良,男", "胡歌,男" };  //调用printInfo方法,传递一个字符串数组,和两个Lambda表达式  printInfo(arr,(message)->{  //消费方式:对message进行切割,获取姓名,按照指定的格式输出  String name = message.split(",")[0];  System.out.print("姓名: "+name);  },(message)->{  //消费方式:对message进行切割,获取年龄,按照指定的格式输出  String age = message.split(",")[1];  System.out.println("；年龄: "+age+"。");  });  }  } |

## 断定类型：Predicate<T>接口

**特点：boolean类型判断，作为方法/构造参数**

java.util.function.Predicate<T>接口，

**作用:**对某种数据类型的数据进行判断,结果返回一个boolean值

Predicate接口中包含一个抽象方法：boolean test(T t):用来对指定数据类型数据进行判断的方法

**结果:**  符合条件,返回true；不符合条件,返回false

|  |
| --- |
| import java.util.function.Predicate;  /\*\*  \* MyPredicate1  \*/  public class MyPredicate1 {  public static boolean validateStr(String str, Predicate<String> predicate){  return predicate.test(str);  }    public static void main(String[] args) {  String str = "abcdef";  boolean b = validateStr(str,string->str.length()>5);  System.out.println(b);    boolean b1 = validateStr(str, new Predicate<String>() {  @Override  public boolean test(String s) {  return s.length()>5;  }  });  System.out.println(b1);  }  } |

* **and方法：**

Predicate接口中有一个方法and,表示并且关系,也可以用于连接两个判断条件

|  |
| --- |
| default Predicate<T> and(Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> this.test(t) && other.test(t);  } |

|  |
| --- |
| import java.util.function.Predicate;  /\*\*  \* MyPredicateAnd.  \*/  public class MyPredicateAnd {  public static boolean validateStr(String str, Predicate<String> pre1,  Predicate<String> pre2){  //return pre1.test(str) && pre2.test(str);  return pre1.and(pre2).test(str);  }    public static void main(String[] args) {  String s = "abcdef";  boolean b = validateStr(s,str->str.length()>5, str->str.contains("a"));  System.out.println(b);  }  } |

* **or方法：**

Predicate接口中有一个方法or,表示或者关系,也可以用于连接两个判断条件

|  |
| --- |
| default Predicate<T> or(Predicate<? super T> other) {  Objects.requireNonNull(other);  return (t) -> test(t) || other.test(t);  } |

|  |
| --- |
| package com.hcx.lambda;    import java.util.function.Predicate;  /\*\*  \* MyPredicateOr  \*/  public class MyPredicateOr {  public static boolean validateStr(String s, Predicate<String> pre1,  Predicate<String> pre2){  //return pre1.test(s) || pre2.test(s);  return pre1.or(pre2).test(s);  }    public static void main(String[] args) {  String s = "acdef";  boolean b = validateStr(s,str->str.length()>5,str->str.contains("a"));    validateStr(s, new Predicate<String>() {  @Override  public boolean test(String str) {  return s.length()>5;  }  }, new Predicate<String>() {  @Override  public boolean test(String str) {  return s.contains("a");  }  });  System.out.println(b);  }  } |

* **negate方法：**

Predicate接口中有一个方法negate,也表示取反的意思

|  |
| --- |
| default Predicate<T> negate() {  return (t) -> !test(t);  } |

|  |
| --- |
| import java.util.function.Predicate;  /\*\*  \* MyPredicateNegate  \*/  public class MyPredicateNegate {  public static boolean validateStr(String s, Predicate<String> pre){  // return !pre.test(s);  return pre.negate().test(s);  }    public static void main(String[] args) {  String s = "acde";  boolean b = validateStr(s,str->str.length()>5);  System.out.println(b);  }  } |

**Predicate测试代码例：**

|  |
| --- |
| package com.hcx.lambda;    import java.util.ArrayList;  import java.util.function.Predicate;    /\*\*  \* PredicateTest  \*/  public class PredicateTest {    public static ArrayList<String> filter(String[] arr, Predicate<String> pre1,  Predicate<String> pre2) {  ArrayList<String> list = new ArrayList<>();  for (String s : arr) {  boolean b = pre1.and(pre2).test(s);  if (b) {  list.add(s);  }  }  return list;  }    public static void main(String[] args) {  String[] array = {"迪丽热巴,女", "古力娜扎,女", "佟丽娅,女", "赵丽颖,女"};  ArrayList<String> list = filter(array,  s -> s.split(",")[1].equals("女"),  s -> s.split(",")[0].length() == 4);  for(String s : list){  System.out.println(s);  }  }  } |

## 转换类型：Function<T>接口

**特点：有输入，有输出**

java.util.function.Function<T,R>接口，用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。

* **apply方法**

Function接口中最主要的抽象方法为：R apply(T t)，根据类型T的参数获取类型R的结果。使用的场景例如：将String类型转换为Integer类型。

|  |
| --- |
| package com.hcx.lambda;    import java.util.function.Function;  /\*\*  \* MyFunction  \*/  public class MyFunction {    public static void change(String str, Function<String,Integer> function){  //Integer i = function.apply(str);  //自动拆箱 Integer自动转为int  int i = function.apply(str);  System.out.println(i);  }    public static void main(String[] args) {  String s = "1234";  change(s,str->Integer.parseInt(str));    int i = Integer.parseInt(s);  System.out.println(i);  }  } |

* **andThen方法：**

|  |
| --- |
| package com.hcx.lambda;    import java.util.function.Function;    /\*\*  \* 1.String转Integer，加10  \* Function<String,Integer> fun1 :Integer i = fun1.apply("123")+10;  \* 2.Interger转String  \* Function<Integer,String> fun2 :String s = fun2.apply(i);  \*/  public class MyFunctionTest {  public static void change(String str, Function<String,Integer> fun1,  Function<Integer,String> fun2){  String string = fun1.andThen(fun2).apply(str);  System.out.println(string);  }    public static void main(String[] args) {  change("123",str->Integer.parseInt(str)+10,i->i+"");  }  } |

**测试代码例**

|  |
| --- |
| package com.hcx.lambda;  import java.util.function.Function;    /\*\*  \* String str = "赵丽颖,20";  \* 1.将字符串截取数字年龄部分，得到字符串；  \* 2.将上一步的字符串转换成为int类型的数字；  \* 3.将上一步的int数字累加100，得到结果int数字。  \*/  public class MyFunctionTest2 {  public static int change(String str, Function<String,String> fun1,  Function<String,Integer> fun2, Function<Integer,Integer> fun3){  return fun1.andThen(fun2).andThen(fun3).apply(str);  }    public static void main(String[] args) {  int num = change("赵丽颖,32",str->str.split(",")[1],  str->Integer.parseInt(str),  i->i+100);  System.out.println(num);  }  } |

**注意：使用匿名内部类会编译后会多产生一个类，而使用lambda，底层是invokedynamic指令,不会有多余的类**

## 其他函数式接口

* **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
* **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
* **BinaryOperator：**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
* **BiPredicate<T,U>：**代表了一个两个参数的boolean值方法
* **BooleanSupplier：**代表了boolean值结果的提供方

# 方法引用



## 方法引用概念

若lambda体中的内容，有方法已经实现了，则可以使用方法引用。方法引用是对lambda的简化

|  |
| --- |
| public interface MyPrintable {  void print(String str);  }  public class DemoPrint {  private static void printString(MyPrintable data){  data.print("Hello,World");  }    public static void main(String[] args) {  printString(s->System.out.println(s));  printString(new MyPrintable() {  @Override  public void print(String str) {  System.out.println(str);  }  });  }  }  //优化后的代码：  public class DemoPrint {  private static void printString(MyPrintable data){  data.print("Hello,World");  }    public static void main(String[] args) {  //printString(s->System.out.println(s));  printString(System.out::println);  }  } |

* **方法引用**

双冒号::为引用运算符，而它所在的表达式被称为**方法引用**。

如果Lambda要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者。

* **三种格式：**
* **对象::实例方法名**
* **类::静态方法名**
* **类::实例方法名**
* **分析**
* **Lambda表达式写法： s -> System.out.println(s);**
* **方法引用写法： System.out::println**

以上两种写法完全等效：

第一种：拿到参数之后通过Lambda传递给 System.out.println 方法去处理。

第二种：直接让 System.out 中的 println 方法来取代Lambda。

**注意:**  lambda体中调用的方法的参数列表和返回值类型要与函数式接口的抽象方法的参数列表与返回值类型一致。Lambda 中 传递的参数一定是方法引用中的那个方法可以接收的类型,否则会抛出异常

## 通过对象名引用成员方法

|  |
| --- |
| public interface MyPrintable {  void print(String str);  }  public class MethodReadObj {  public void printUpperCaseString(String str){  System.out.println(str.toUpperCase());  }  }  public class MethodReference1 {    public static void printString(MyPrintable p){  p.print("hello");  }    public static void main(String[] args) {  printString(str-> {  MethodReadObj methodReadObj = new MethodReadObj();  methodReadObj.printUpperCaseString(str);  });    /\*\*  \* 使用方法引用：  \* 1.MethodReadObj对象已经存在  \* 2.成员方法printUpperCaseString已经存在  \* 所以可以使用对象名引用成员方法  \*/  MethodReadObj methodReadObj = new MethodReadObj();  printString(methodReadObj::printUpperCaseString);    }  } |

## 通过类名称引用静态方法

类已经存在，静态方法已经存在，则可以通过类名直接引用静态成员方法

|  |
| --- |
| public interface MyCalc {  int calc(int num);  }  public class MethodRerference2 {    public static int method(int num,MyCalc c){  return c.calc(num);  }    public static void main(String[] args) {  //通过类名直接引用静态成员方法  int number = method(-10, num -> Math.abs(num));  //通过类名直接引用静态成员方法  int number1 = method(-10, Math::abs);  System.out.println(number);  System.out.println(number1);  }  } |

通过类名引用实例方法

|  |
| --- |
| @Test  public void test2(){  BiPredicate<String,String> biPredicate = (x,y) -> x.equals(y);  **//通过类名引用实例方法**  BiPredicate<String,String> biPredicate1 = String::equals;  } |

## 通过super引用成员方法

如果存在继承关系，当Lambda中需要出现super调用时，也可以使用方法引用进行替代。

|  |
| --- |
| @FunctionalInterface  public interface MyMeet {  void meet();  }  public class Parent {  public void hello(){  System.out.println("hello，I'm Parent");  }  }  public class Child extends Parent{  @Override  public void hello() {  System.out.println("hello，I'm Child");  }    public void method(MyMeet myMeet){  myMeet.meet();  }    public void show(){  method(()->{  Parent parent = new Parent();  parent.hello();  });    //使用super关键字调用父类  method(()->super.hello());    /\*\*  \* 使用方法引用：使用super引用父类的成员方法：  \* 1.super已经存在  \* 2.父类的成员方法hello已经存在  \* 可以直接使用super引用父类的成员方法  \*/  method(super::hello);  }    public static void main(String[] args) {  new Child().show();  }  } |

## 通过this引用成员方法

this代表当前对象，如果需要引用的方法就是当前类中的成员方法，那么可以使用 this::成员方法的格式来使用方法引用

|  |
| --- |
| @FunctionalInterface  public interface MyWallet {  void buy();  }  public class BuyThing {  public void buyCar(){  System.out.println("买了一辆别摸我");  }    public void getSalary(MyWallet myWallet){  myWallet.buy();  }    public void method(){  getSalary(()->this.buyCar());  /\*\*  \* 1.this已经存在  \* 2.本类的成员方法buyCar已经存在  \* 所以可以直接使用this引用本类的成员方法buyCar  \*/  getSalary(this::buyCar);  }    public static void main(String[] args) {  new BuyThing().method();  }  } |

## 类的构造器引用

由于构造器的名称与类名完全一样，并不固定。所以构造器引用使用 类名称::new 的格式表示。

|  |
| --- |
| @Data //Lambada注解  @NoArgsConstructor  @AllArgsConstructor  public class Person {  private String name;  }  @FunctionalInterface  public interface PersonBuilder {  //根据传递的姓名，创建Perosn对象  Person builderPerson(String name);  }  public class Demo {  //传递姓名和PersonBuilder接口，通过姓名创建Person对象  public static void printName(String name,PersonBuilder personBuilder){  Person person = personBuilder.builderPerson(name);  System.out.println(person.getName());  }    public static void main(String[] args) {  printName("hongcaixia",str->new Person(str));  /\*\*  \* 使用方法引用：  \* 1.构造方法new Person(String name)已知  \* 2.创建对象已知  \* 可以使用Person引用new创建对象  \*/  printName("hongcaixia",Person::new);  }  } |

## 数组的构造器引用

数组也是 Object 的子类对象，所以同样具有构造器，只是语法稍有不同。

**格式：Type[]::new**

|  |
| --- |
| @FunctionalInterface  public interface ArrayBuilder {  //创建int类型数组的方法，参数传递数组的长度，返回创建好的int类型数组  int[] builderArray(int length);  }  public class DemoArrayBuilder {  public static int[] createArray(int length,ArrayBuilder arrayBuilder){  return arrayBuilder.builderArray(length);  }  public static void main(String[] args) {  int[] arr1 = createArray(5, length -> new int[length]);  System.out.println(arr1.length);  /\*\*  \* 1.已知创建的是int[]数组  \* 2.创建的数组长度也是已知的  \* 使用方法引用，int[]引用new，根据参数传递的长度创建数组  \*/  int[] arr2 = createArray(10,int[]::new);  System.out.println(Arrays.toString(arr2));  System.out.println(arr2.length);  }  } |

# StreamAPI



## Stream概念

流是数据渠道，用于操作数据源（集合、数组等）生成的元素序列。

* 元素是特定类型的对象，形成一个队列。
* 数据源流的来源。 可以是集合，数组等。
* Stream自己不会存储元素，而是按需计算。
* Stream不会改变源对象，并且能返回一个持有结果的新流
* Stream操作是延迟操作，意味着他们会等到需要结果的时候才执行
* **和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或者增强for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，流可以直接调用遍历方法。
* **Stream操作的三个步骤：**
* 创建Stream：一个数据源（如集合、数组），获取一个流
* 中间操作：一个操作链，对数据源的数据进行处理
* 终止操作：一个终止操作，执行中间操作链并产生结果

**注意：“Stream流”其实是一个集合元素的函数模型，它并不是集合，也不是数据结构，其本身并不存储任何元素（或其地址值）。**

|  |
| --- |
| import java.util.\*;  import java.util.function.Predicate;    /\*\*  \* Created by hongcaixia on 2019/10/31.  \*/  public class MyPredicate {  public static void main(String[] args) {  List<Integer> nums = Arrays.asList(10,20,3,-5,-8);  Collection<Integer> positiveNum = filter(nums,num->num>0);  Collection<Integer> negativeNum = filter(nums,num->num<0);  System.out.println(positiveNum);  System.out.println(negativeNum);  }    private static <E> Collection<E> filter(Collection<E> source, Predicate<E> predicate){  List<E> list = new ArrayList<>(source);  Iterator<E> iterator = list.iterator();  while (iterator.hasNext()){  E element = iterator.next();  if(!predicate.test(element)){  iterator.remove();  }  }  return Collections.unmodifiableList(list);  }  } |

## 获取Stream流

java.util.stream.Stream<T>是Java8新加入的常用的流接口。（不是函数式接口）

* **所有的 Collection 集合都可以通过stream默认方法获取流**
* **Stream 接口的静态方法of可以获取数组对应的流**

**获取一个流有以下几种常用的方式：**

## 根据Collection获取流

java.util.Collection接口中加入了default方法 stream 用来获取流，所以其所有实现类均可获取流。

* stream()：获取串行流
* parallelStream()：获取并行流

|  |
| --- |
| package com.hcx.stream;    import java.util.\*;  import java.util.stream.Stream;    /\*\*  \* Created by hongcaixia on 2019/10/31.  \*/  public class GetStreamFromCollection {  public static void main(String[] args) {    List<String> list = new ArrayList<>();  Stream<String> stream1 = list.stream();    Set<String> set = new HashSet<>();  Stream<String> stream2 = set.stream();    Vector<String> vector = new Vector<>();  Stream<String> stream3 = vector.stream();  }  } |

## 根据Map获取流

java.util.Map 接口不是Collection的子接口，且其K-V数据结构不符合流元素的单一特征，所以获取对应的流需要分key、value或entry等情况：

|  |
| --- |
| package com.hcx.stream;    import java.util.HashMap;  import java.util.Map;  import java.util.stream.Stream;    /\*\*  \* Created by hongcaixia on 2019/10/31.  \*/  public class GetStreamFromMap {    public static void main(String[] args) {  Map<String,String> map = new HashMap<>();    Stream<Map.Entry<String, String>> stream1 = map.entrySet().stream();    Stream<String> stream2 = map.keySet().stream();  Stream<String> stream3 = map.values().stream();    }  } |

## 根据数组获取流

如果使用的不是集合或映射而是数组，由于数组对象不可能添加默认方法，所以 Stream 接口中提供了静态方法of （**of 方法的参数是一个可变参数，所以支持数组。**）

|  |
| --- |
| package com.hcx.stream;    import java.util.stream.Stream;  /\*\*  \* GetStreamFromArray  \*/  public class GetStreamFromArray {    public static void main(String[] args) {  String[] array = {"陈奕迅","钟汉良","杨千嬅"};  Stream<String> stream = Stream.of(array);  }  } |

## 获取无限流

获取一个长度不受限制的流数据，下面的为了测试用limit 方法对流进行截取

|  |
| --- |
| @Test public void test6() {  Stream<Integer> stream = Stream.iterate(0, x -> x + 2).limit(10);  stream.forEach(x -> System.out.println(x));  System.out.println("------------------------------------");  stream = Stream.iterate(0, x -> x +1).limit(10);  stream.forEach(x -> System.out.println(x));  } |

## Stream流常用方法

**方法可以被分成两种：**

* **延迟方法**：返回值类型仍然是 Stream 接口自身类型的方法，因此支持链式调用。（除了终结方法外，其余方法均为延迟方法。）
* **终结方法**：返回值类型不再是 Stream接口自身类型的方法，因此不再支持类似 StringBuilder 那样的链式调用。终结方法包括 count 和 forEach 方法。

## 逐一处理：forEach

**方法签名：**void forEach(Consumer<? super T> action);

该方法接收一个Consumer接口函数，会将每一个流元素交给该函数进行处理。Consumer是一个消费型的函数式接口，可传递lambda表达式，消费数据

|  |
| --- |
| package com.hcx.stream;    import java.util.stream.Stream;  public class StreamForEach {  public static void main(String[] args) {  Stream<String> stream = Stream.of("张三", "李四", "王五");  stream.forEach(str-> System.out.println(str));  }  } |

## 过滤：filter

**方法签名：**void forEach(Consumer<? super T> action);

可以通过 filter 方法将一个流转换成另一个子集流。该接口接收一个Predicate 函数式接口参数（可以是一个Lambda或方法引用）作为筛选条件。

java.util.stream.Predicate函数式接口唯一的抽象方法为: boolean test(T t);

该方法将会产生一个boolean值结果，代表指定的条件是否满足：

如果结果为true，那么Stream流的filter方法将会留用元素；

如果结果为false，那么filter 方法将会舍弃元素。

|  |
| --- |
| public class StreamFilter {  public static void main(String[] args) {  Stream<String> stream = Stream.of("陈奕迅", "陈伟霆", "陈七", "钟汉良");  Stream<String> stream1 = stream.filter(str -> str.startsWith("陈"));  stream1.forEach(str-> System.out.println(str));  System.out.println("------------------------------------");  // java.lang.IllegalStateException  stream1.forEach(str -> System.out.println(str));  }  } |

注意：**Stream属于管道流，只能被消费一次**

Stream流调用完毕方法，数据就回流到下一个Steam上，而这时第一个Stream流已经使用完毕，就会关闭了，所以第一个Stream流就不能再调用方法了，否则将发生异常（java.lang.IllegalStateException: stream has already been operated upon or closed）。

## 映射：map

**方法签名：**<R> Stream<R> map(Function<? super T, ? extends R> mapper);

接收lambda，将元素转换成其他形式或提取信息，接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。即将流中的元素映射到另一个流中。

该接口需要一个 Function 函数式接口参数，可以将当前流中的T类型数据转换为另一种R类型的流。java.util.stream.Function 函数式接口，其中唯一的抽象方法为： R apply(T t);这可以将一种T类型转换成为R类型，而这种转换的动作，就称为“映射”。

|  |
| --- |
| public class StreamMap {  public static void main(String[] args) {  Person person = new Person("hcx",24);  Person person1 = new Person("hcx2",24);  List<Person> list = new ArrayList<>();  list.add(person);  list.add(person1);  //list.stream().map(p -> p.getName()).forEach( str -> System.out.println(str));  list.stream().map(Person::getName).forEach(System.out::println);  }  public class Person{  public Person(String name, int age){  this.name=name;  this.age=age;  }  private String name;  private int age;  public String getName(){return this.name;}  }  } |

## flatMap

接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。即对流扁平化处理,浅显一点解释就是把几个小的list转换到一个大的list

如：[['a','b'],['c','d']] - > ['a','b','c','d']  
如果我们使用常用的map()方法获取的lowercaseWords数据结构为：[['a','b','c'],['m','d','w'],['k','e','t']]。如果我们要得到如：['a','b','c','m','d','w','k','e','t']这样数据结构的数据，就需要使用flatMap()方法。

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.stream.Stream;  public class StreamFlatMapTest {  @Test  public void runTest() {  testFlatMap();  }  public void testFlatMap() {  List<String> list = Arrays.asList("a", "b", "c");  Stream<Stream<Character>> streamStream = list.stream().map(TempTest::filterCharacter);  streamStream.forEach((stream) -> stream.forEach(System.out::println));  //使用flatMap  Stream<Character> characterStream = list.stream().flatMap(TempTest::filterCharacter);  characterStream.forEach(System.out::println);  }  public static Stream<Character> filterCharacter(String str) {  List<Character> list = new ArrayList<>();  for (Character c : str.toCharArray()) list.add(c);  return list.stream();  }  } |

## 规约reduce

将流中元素反复结合起来，得到一个值

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.util.Arrays;  import java.util.List;  public class TempTest {  @Test  public void runTest() {  List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);  nums.stream().reduce((x, y) -> Integer.sum(x, y)).ifPresent(System.out::println);  nums.stream().reduce((x, y) -> x + y).ifPresent(System.out::println);  nums.stream().reduce(Integer::sum).ifPresent(System.out::println);  Integer sum = nums.stream().reduce(100, (x, y) -> x + y);  System.out.println(sum);  }  } |

## 组合：concat

**方法签名**：static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)

**这是一个静态方法，与 java.lang.String 当中的 concat 方法不同**

如果有两个流，希望合并成为一个流，那么可以使用 Stream 接口的静态方法 concat

|  |
| --- |
| public class StreamConcat {  public static void main(String[] args) {  Stream<String> stream1 = Stream.of("陈奕迅", "陈伟霆", "陈七", "钟汉良");  String[] arr = {"1","2","3"};  Stream<String> stream2 = Stream.of(arr);  Stream<String> concatStream = Stream.concat(stream1, stream2);  concatStream.forEach(str-> System.out.println(str));  }  } |

## 转换：Collect

将流转换为其他形式。接收一个Collector的实现，用于给stream中元素做汇总的方法

|  |
| --- |
| import java.util.LinkedList; import java.util.List; import java.util.stream.Collectors; import java.util.stream.Stream;  public class TempTest {  @Test  public void runTest() {  List<Integer> list = Stream.of(1, 2, 3, 4, 5).collect(Collectors.toList());  List<Integer> list1 = Stream.of(1, 2, 3, 4, 5).collect(LinkedList::new, List::add, List::addAll);  System.out.println(list.getClass());//class java.util.ArrayList  System.out.println(list1.getClass());//class java.util.LinkedList  }  } |

下面是可以多级分组(groupingBy)的使用方式：

|  |
| --- |
| //The following will classify Person objects by city:  Map<String, List<Person>> peopleByCity = personStream.collect(  Collectors.groupingBy(Person::getCity));  //The following will classify Person objects by state and city, cascading two Collectors together:  Map<String, Map<String, List<Person>>> peopleByStateAndCity = personStream.collect(  Collectors.groupingBy(Person::getState, Collectors.groupingBy(Person::getCity))); |

下面是可以数据分区(partitioningBy)的使用方式：

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;  import java.util.stream.Stream;  public class TempTest {  @Test  public void runTest() {  List<Integer> list = Stream.of(1, 2, 3, 4, 5).collect(Collectors.toList());  Map<Boolean, List<Integer>> map = list.stream().collect(Collectors.partitioningBy((m) -> m > 3));  for (Map.Entry<Boolean, List<Integer>> entry : map.entrySet()) {  System.out.println(entry.getKey() + ":" + entry.getValue());  }  }  }  **输入结果：**  false:[1, 2, 3]  true:[4, 5] |

## 其他方法

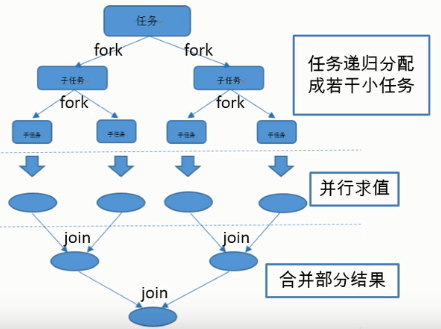
* sorted()：自然排序
* sorted(Comparator com) ：定制排序
* allMatch： 检查是否匹配所有元素
* anyMatch： 检查是否至少匹配一个元素
* noneMatch： 检查是否没有匹配所有元素
* findFirst： 返回第一个元素
* findAny： 返回当前流中的任意元素
* count： 返回流中元素的总个数
* max： 返回流中最大值
* min： 返回流中最小值
* skip：跳过前几个元素，可以使用skip方法获取一个截取之后的新流
* limit：对流进行截取，只取用前n个
* count：返回一个long值代表元素个数
* **分区**

## 并行流和顺序流

并行流是把一个内容拆分(Fork)成多个数据块，并用不同的线程分别处理每个数据块的流。通过parallel()与sequential()可以实现并行流和顺序流之间的切换。

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.util.stream.LongStream;  public class TempTest {  @Test  public void runTest() {  //顺序流（默认）  LongStream.rangeClosed(0, 100).sequential().forEach(System.out::println);  System.out.println("--------------------");  //并行流  LongStream.rangeClosed(0, 100).parallel().forEach(System.out::println);  System.out.println("--------------------");  //并行流转换成穿行流  LongStream.rangeClosed(0, 100).parallel().sequential().forEach(System.out::println);  System.out.println("--------------------");  //并行流计算处理  long reduce = LongStream.rangeClosed(0, 100).parallel().reduce(0, Long::sum);  //5050  System.out.println(reduce);  }  } |

Fork/Join框架：在必要的情况下，将一个大任务进行拆分（fork）成若干小任务（拆到不可再拆时），再将一个个小任务运算的结果进行join汇总。



# Optional容器



## Optional概念

Optional<T>类(java.util.Optional)是一个容器类，代表一个值存在或不存在，原来用null表示一个值不存在，现在Optional可以更好的表达这个概念。并且可以避免空指针异常。使用Optional容器可以快速的定位NPE，并且在一定程度上可以减少对参数非空检验的代码量。

* Optional.of：创建一个容器实例
* Optional.ofNullable：创建一个容器实例（推荐）
* orElse：获取容器内部元素，没有元素返回默认值
* orElseGet：获取容器内部元素，没有元素返回默认值（推荐）
* map：如果有值对其处理，并返回处理后的Optional，否则返回Optional.empty();
* flatMap：与map类似，返回的对象必须是Optional

## Optional使用示例

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.util.Optional;  public class TempTest {  @Test  public void runTest(){  //常规带码逻辑  Boy boy = new Boy();  System.out.println("女朋友是："+getGirlName(boy));  Boy boy2 = new Boy(new Girl("玛利亚"));  System.out.println("女朋友是："+getGirlName(boy2));  //使用Optional的优化逻辑  BoyNew boyNew = new BoyNew();  System.out.println("女朋友是："+getGirlName2(boyNew));  BoyNew boyNew2 = new BoyNew(Optional.ofNullable(new Girl("松岛")));  System.out.println("女朋友是："+getGirlName2(boyNew2));  }  public String getGirlName2(BoyNew boy){  return Optional.ofNullable(boy)  .orElse(new BoyNew())  .getGirl()  .orElse(new Girl("波多"))  .getName();  }  public String getGirlName(Boy boy){  if (boy !=null){  Girl girl = boy.getGirl();  if(girl!=null){  return girl.getName();  }  }  return "波多";  }  public class Girl {  private String name;  @Override  public String toString() {  return "Girl{" + "name='" + name + '\'' + '}';  }  public Girl(String name) {  this.name = name;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  }  public class Boy {  private Girl girl;  public Girl getGirl() {  return girl;  }  public void setGirl(Girl girl) {  this.girl = girl;  }  public Boy() {  }  public Boy(Girl girl) {  this.girl = girl;  }  }  public class BoyNew {  private Optional<Girl> girl = Optional.empty();  public Optional<Girl> getGirl() {  return girl;  }  public void setGirl(Optional<Girl> girl) {  this.girl = girl;  }  public BoyNew() {  }  public BoyNew(Optional<Girl> girl) {  this.girl = girl;  }  }  } |

# JDK8时间日期API

在 JDK8 之前，我们经常使用到的时间API包括(Date、Calendar)，Date 与字符串之间的转换使用 SimpleDateFormat 进行转换（parse()、format() 方法），然而 SimpleDateFormat 不是线程安全的。在设计上也是存在一些缺陷的，比如有两个 Date 类，一个在 java.util 包中，一个在 java.sql 包中。

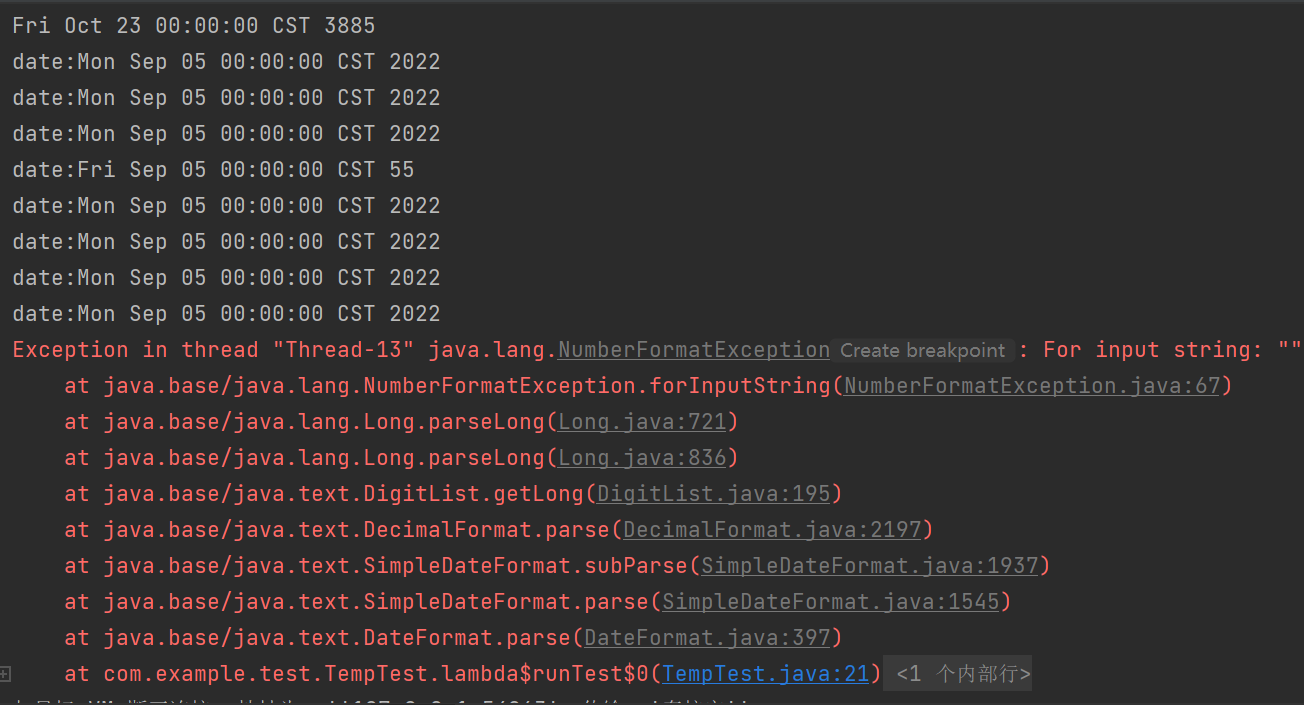
在JDK8 中，引入了一套全新的时间日期API，这套 API 在设计上比较合理，使用时间操作也变得更加方便。并且支持多线程安全操作。



## 旧版日期时间API存在的问题

* 设计很差：在 java.util 和 java.sql 的包中都有日期类。java.util.Date 同时包含日期和时间，而java.sql.Date仅包含日期，此外用于格式化和解析的类又在 java.text 包中定义；
* 非线程安全：java.util.Date是非线程安全的,所有的日期类都是可变的，这是 java 日期类最大的问题之一；（时间格式化和解析是线程不安全的）
* 时区处理麻烦：日期类并不提供国际化，没有时区支持。因此 java 引入了 java.util.Calendar 和 java.util.TimeZone 类,但他们同样存在上述所有的问题

|  |
| --- |
| package com.example.test;  import org.junit.Test;  import java.text.ParseException;  import java.text.SimpleDateFormat;  import java.util.Date;  public class TempTest {  @Test  public void runTest() {  //旧版日期时间 API 存在的问题  //1.设计不合理：下面日期不是预期的结果  Date now = new Date(1985, 9, 23);  System.out.println(now);  //2.时间格式化和解析是线程不安全的  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  for (int i = 0; i < 20; i++) {  new Thread(() -> {  try {  Date date = sdf.parse("2022-09-05");  System.out.println("date:" + date);  } catch (ParseException e) {  e.printStackTrace();  }  }).start();  }  }  }} |

**多线程测试结果：(会有日期格式化错误的，还有直接报错的，说明线程是不安全)**

## 新日期时间 API 介绍

JDK8 中增加了一套全新的日期时间 API，这套 API 设计合理，线程安全。新的日期及时间提供了丰富的 API， 位于 java.time 包下，如下是一些该包下的关键类：

* LocalDate：表示日期，包含：年月日。格式为：2020-01-13
* LocalTime：表示时间，包含：时分秒。格式为：16:39:09.307
* LocalDateTime：表示日期时间，包含：年月日 时分秒。格式为：2020-01-13T16:40:59.138
* DateTimeFormatter：日期时间格式化类
* Instant：时间戳类
* Duration：用于计算 2 个时间(LocalTime，时分秒)之间的差距
* Period：用于计算 2 个日期(LocalDate，年月日)之间的差距
* ZonedTime：包含时区的时间
* ZonedDateTime：包含时区的时间
* ZonedDateTime：包含时区的时间

**说明：**其中每个时区都有对应着的 ID，ID的格式为"区域/城市"，例如：Asia/Shanghai 等。

Java 中使用的历法是 ISO-8601 日历系统，他是世界民用历法，也就是我们所说的公历。平年有365天，闰年是366天。此外 Java8 还提供了 4套其他历法，分别是：

* ThaiBuddhistDate：泰国佛教历
* MinguoDate：中华民国历
* JapaneseDate：日本历
* HijrahDate：伊斯兰历

## API使用说明

## 日期和时间类

LocalDate、LocalTime、LocalDateTime类的实例是不可变的对象，分别表示使用 ISO-8601 日历系统的日期、时间、日期和时间。他们提供了简单的日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。

对日期时间的修改，就是对已经存在的 LocalDate对象，根据需求创建它的修改版，最简单的方式是使用 withxxx() 、plusxxx()、minusxxx()方法。这些方法会创建对象的一个副本，并按照需要修改它的属性。

**另外jdbc时间类型和 java8 时间类型对应关系：**

* Date ---> LocalDate
* Time ---> LocalTime
* Timestamp ---> LocalDateTime

## 日期时间格式化与解析

|  |
| --- |
| //使用JDK自带的时间格式:ISO\_LOCAL\_DATE  DateTimeFormatter dtf = DateTimeFormatter.ISO\_LOCAL\_DATE;  String format = LocalDate.now().format(dtf);  System.out.println("format="+format);  //指定时间格式(ofPattern()方法)  DateTimeFormatter dtf1 = DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH时mm分ss秒");  String format1 = LocalDateTime.now().format(dtf1);  System.out.println("format="+format1); |

## 设置日期时间的时区

JDK8 中加入了对时区的支持。LocalDate、LocalTime、LocalDateTime 是不带时区的，带时区的日期时间类分别为：ZonedDate、ZonedTime、ZonedDateTime类。

其中每个时区都有对应着的 ID，ID的格式为"区域/城市"，例如：Asia/Shanghai 等。

ZoneId类：该类中包含了所有的时区信息。

# 其他特性

* + **并发性**

基于新增的lambda表达式和steam特性，为Java 8中为java.util.concurrent.ConcurrentHashMap类添加了新的方法来支持聚焦操作；另外，也为java.util.concurrentForkJoinPool类添加了新的方法来支持通用线程池操作

Java 8还添加了新的java.util.concurrent.locks.StampedLock类，用于支持基于容量的锁——该锁有三个模型用于支持读写操作（可以把这个锁当做是java.util.concurrent.locks.ReadWriteLock的替代者）。

在java.util.concurrent.atomic包中也新增了不少工具类，列举如下：

* DoubleAccumulator
* DoubleAdder
* LongAccumulator
* LongAdder
  + **Base64**

[对Base64编码的支持](https://link.zhihu.com/?target=http%3A//www.javacodegeeks.com/2014/04/base64-in-java-8-its-not-too-late-to-join-in-the-fun.html)已经被加入到Java 8官方库中，这样不需要使用第三方库就可以进行Base64编码

* + **新Java工具**
* 类依赖分析器：jdeps命令

jdeps是一个相当棒的命令行工具，它可以展示包层级和类层级的Java类依赖关系，它以.class文件、目录或者Jar文件为输入，然后会把依赖关系输出到控制台。

如：jedps org.springframework.core-3.0.5.RELEASE.jar

* Nashorn JavaScript引擎：

jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行。

例如，我们写一个func.js的JavaScript文件后，可执行命令：jjs func.js

* + **JVM的新特性**

使用[Metaspace](https://link.zhihu.com/?target=http%3A//www.javacodegeeks.com/2013/02/java-8-from-permgen-to-metaspace.html)（[JEP 122](https://link.zhihu.com/?target=http%3A//openjdk.java.net/jeps/122)）代替持久代（PermGen space）。在JVM参数方面，使用-XX:MetaSpaceSize和-XX:MaxMetaspaceSize代替原来的-XX:PermSize和-XX:MaxPermSize。

# 特别推荐参考：

What's New in JDK 8：<https://www.oracle.com/java/technologies/javase/8-whats-new.html>

Java8学习指南：<https://mkyong.com/tutorials/java-8-tutorials>