



# 第16章 Java8的其它 新特性

讲师：宋红康  
新浪微博：尚硅谷-宋红康



# 目录



1

Lambda表达式

2

函数式(Functional)接口

3

方法引用与构造器引用

4

强大的Stream API

5

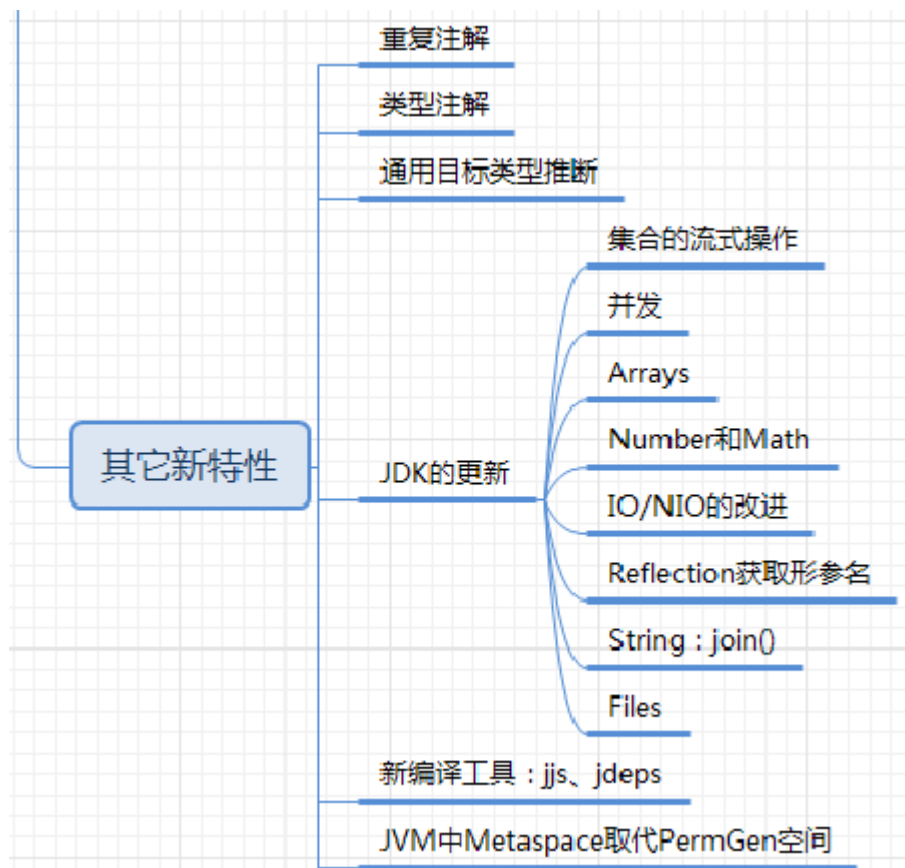
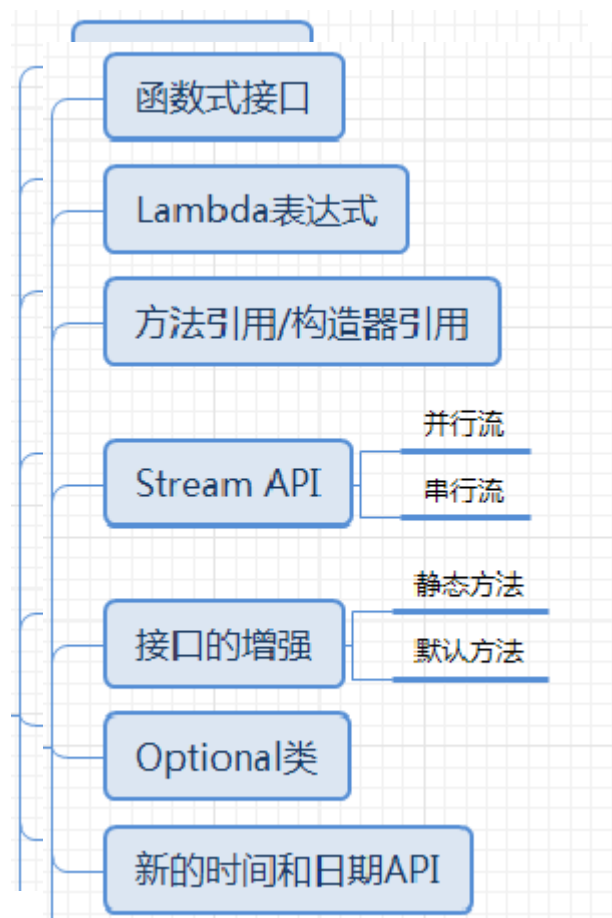
Optional类



# Java 8新特性简介

Java 8 (又称为 jdk 1.8) 是 Java 语言开发的一个主要版本。

Java 8 是oracle公司于2014年3月发布，可以看成是自Java 5 以来最具革命性的版本。Java 8为Java语言、编译器、类库、开发工具与JVM带来了大量新特性。





- 速度更快
- 代码更少(增加了新的语法: **Lambda 表达式**)
- 强大的 **Stream API**
- 便于并行
- 最大化减少空指针异常: Optional
- Nashorn引擎, 允许在JVM上运行JS应用



## 并行流与串行流

**并行流**就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。相比较串行的流，**并行的流可以很大程度上提高程序的执行效率。**

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。

Stream API 可以声明性地通过 `parallel()` 与 `sequential()` 在并行流与顺序流之间进行切换。



## 16-1 Lambda表达式





# 为什么使用 Lambda 表达式

Lambda 是一个**匿名函数**，我们可以把 Lambda 表达式理解为是一段**可以传递的代码**（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java的语言表达能力得到了提升。



- 从匿名类到 Lambda 的转换举例1

```
//匿名内部类
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello World!");
    }
};
```



```
//Lambda 表达式
Runnable r1 = () -> System.out.println("Hello Lambda!");
```



- 从匿名类到 Lambda 的转换举例2

```
//原来使用匿名内部类作为参数传递
TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(o1.length(), o2.length());
    }
});
```



```
//Lambda 表达式作为参数传递
TreeSet<String> ts2 = new TreeSet<>(  
    (o1, o2) -> Integer.compare(o1.length(), o2.length())  
);
```



Lambda 表达式：在Java 8 语言中引入的一种新的语法元素和操作符。这个操作符为 “->”， 该操作符被称为 **Lambda 操作符** 或**箭头操作符**。它将 Lambda 分为两个部分：

**左侧：** 指定了 Lambda 表达式需要的**参数列表**

**右侧：** 指定了 **Lambda 体**，是抽象方法的实现逻辑，也即 Lambda 表达式要执行的功能。



语法格式一：无参，无返回值

```
Runnable r1 = () -> {System.out.println("Hello Lambda!");};
```

语法格式二：Lambda 需要一个参数，但是没有返回值。

```
Consumer<String> con = (String str) -> {System.out.println(str);};
```

语法格式三：数据类型可以省略，因为可由编译器推断得出，称为“类型推断”

```
Consumer<String> con = (str) -> {System.out.println(str);};
```



语法格式四：Lambda 若只需要一个参数时，参数的小括号可以省略

```
Consumer<String> con = str -> {System.out.println(str);};
```

语法格式五：Lambda 需要两个或以上的参数，多条执行语句，并且可以有返回值

```
Comparator<Integer> com = (x,y) -> {  
    System.out.println("实现函数式接口方法！");  
    return Integer.compare(x,y);  
};
```

语法格式六：当 Lambda 体只有一条语句时，return 与大括号若有，都可以省略

```
Comparator<Integer> com = (x,y) -> Integer.compare(x, y);
```



# 类型推断

上述 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 `javac` 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。这就是所谓的“类型推断”。

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```



## 16-2 函数式(Functional)接口





# 什么是函数式(Functional)接口

- 只包含一个抽象方法的接口，称为函数式接口。
- 你可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda 表达式抛出一个受检异常(即：非运行时异常)，那么该异常需要在目标接口的抽象方法上进行声明）。
- 我们可以在一个接口上使用 **@FunctionalInterface** 注解，这样做可以检查它是否是一个函数式接口。同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。
- 在 **java.util.function** 包下定义了 Java 8 的丰富的函数式接口



### 如何理解函数式接口

- Java从诞生日起就一直倡导“一切皆对象”，在Java里面面向对象(OOP)编程是一切。但是随着python、scala等语言的兴起和新技术的挑战，Java不得不做出调整以便支持更加广泛的技术要求，也即java不但可以支持OOP还可以支持OOF（面向函数编程）
- 在函数式编程语言当中，函数被当做一等公民对待。在将函数作为一等公民的编程语言中，Lambda表达式的类型是函数。但是在Java8中，有所不同。在Java8中，Lambda表达式是对象，而不是函数，它们必须依附于一类特别的对象类型——函数式接口。
- 简单的说，在Java8中，Lambda表达式就是一个函数式接口的实例。这就是Lambda表达式和函数式接口的关系。也就是说，只要一个对象是函数式接口的实例，那么该对象就可以用Lambda表达式来表示。
- 所以以前用匿名实现类表示的现在都可以用Lambda表达式来写。



### 函数式接口举例

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface Runnable is
     * to create a thread, starting the thread causes the object's
     * run method to be called in that separately executi
     * thread.
     *
     * <p>
     * The general contract of the method run is that it
     * take any action whatsoever.
     *
     * @see      java.lang.Thread#run()
     */
    public abstract void run();
}
```



### 自定义函数式接口

```
@FunctionalInterface↵  
public interface MyNumber{↵  
    public double getValue();↵  
}↵
```

函数式接口中使用泛型：

```
@FunctionalInterface↵  
public interface MyFunc<T>{↵  
    public T getValue(T t);↵  
}↵
```



### 作为参数传递 Lambda 表达式

```
public String toUpperString(MyFunc<String> mf, String str){  
    return mf.getValue(str);  
}
```

作为参数传递 **Lambda** 表达式:

```
String newStr = toUpperString(  
    (str) -> str.toUpperCase(), "abcdef");  
System.out.println(newStr);
```

实例化

作为参数传递 **Lambda** 表达式: 为了将 **Lambda** 表达式作为参数传递, 接收 **Lambda** 表达式的参数类型必须是与该 **Lambda** 表达式兼容的函数式接口的类型。



# Java 内置四大核心函数式接口

函数式接口	参数类型	返回类型	用途
<b>Consumer&lt;T&gt;</b> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： <b>void accept(T t)</b>
<b>Supplier&lt;T&gt;</b> 供给型接口	无	T	返回类型为T的对象，包含方法： <b>T get()</b>
<b>Function&lt;T, R&gt;</b> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法： <b>R apply(T t)</b>
<b>Predicate&lt;T&gt;</b> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回boolean 值。包含方法： <b>boolean test(T t)</b>



### 其他接口

函数式接口	参数类型	返回类型	用途
<b>BiFunction&lt;T, U, R&gt;</b>	T, U	R	对类型为 T, U 参数应用操作, 返回 R 类型的结果。包含方法为: <b>R apply(T t, U u);</b>
<b>UnaryOperator&lt;T&gt;</b> (Function子接口)	T	T	对类型为T的对象进行一元运算, 并返回T类型的结果。包含方法为: <b>T apply(T t);</b>
<b>BinaryOperator&lt;T&gt;</b> (BiFunction 子接口)	T, T	T	对类型为T的对象进行二元运算, 并返回T类型的结果。包含方法为: <b>T apply(T t1, T t2);</b>
<b>BiConsumer&lt;T, U&gt;</b>	T, U	void	对类型为T, U 参数应用操作。 包含方法为: <b>void accept(T t, U u)</b>
<b>BiPredicate&lt;T,U&gt;</b>	T,U	boolean	包含方法为: <b>boolean test(T t,U u)</b>
<b>ToIntFunction&lt;T&gt;</b> <b>ToLongFunction&lt;T&gt;</b> <b>ToDoubleFunction&lt;T&gt;</b>	T	int long double	分别计算int、long、double值的函数
<b>IntFunction&lt;R&gt;</b> <b>LongFunction&lt;R&gt;</b> <b>DoubleFunction&lt;R&gt;</b>	int long double	R	参数分别为int、long、double 类型的函数



## 16-3 方法引用与构造器引用





# 方法引用(Method References)

- 当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！
- 方法引用可以看做是Lambda表达式深层次的表达。换句话说，方法引用就是Lambda表达式，也就是函数式接口的一个实例，通过方法的名字来指向一个方法，可以认为是Lambda表达式的一个语法糖。
- 要求：实现接口的抽象方法的参数列表和返回值类型，必须与方法引用的方法的参数列表和返回值类型保持一致！
- 格式：使用操作符 “::” 将类(或对象) 与 方法名分隔开来。
- 如下三种主要使用情况：
  - 对象::实例方法名
  - 类::静态方法名
  - 类::实例方法名



### 方法引用

例如：

```
Consumer<String> con = (x) -> System.out.println(x);
```

等同于：

```
Consumer<String> con2 = System.out :: println;
```

例如：

```
Comparator<Integer> com = (x,y) -> Integer.compare(x, y);
```

等同于：

```
Comparator<Integer> com1 = Integer::compare;
```

```
int value = com.compare(12, 32);
```



### 方法引用

例如：

```
BiPredicate<String, String> bp = (x,y) -> x.equals(y);
```

等同于：

```
BiPredicate<String,String> bp1 = String::equals;
```

```
boolean flag = bp1.test("hello", "hi");
```

注意：当函数式接口方法的第一个参数是需要引用方法的调用者，并且第二个参数是需要引用方法的参数(或无参数)时：**ClassName::methodName**



### 构造器引用

格式: **ClassName::new**

与函数式接口相结合, 自动与函数式接口中方法兼容。

可以把构造器引用赋值给定义的方法, 要求构造器参数列表要与接口中抽象方法的参数列表一致! 且方法的返回值即为构造器对应类的对象。

例如:

```
Function<Integer, MyClass> fun = (n) -> new MyClass(n);
```

等同于:

```
Function<Integer, MyClass> fun = MyClass::new;
```



### 数组引用

格式: **type[] :: new**

例如:

```
Function<Integer, Integer[]> fun = (n) -> new Integer[n];
```

等同于:

```
Function<Integer, Integer[]> fun = Integer[]::new;
```



## 16-4 强大的Stream API



### Stream API说明

- Java8中有两大最为重要的改变。第一个是 **Lambda 表达式**；另外一个则是 **Stream API**。
- **Stream API ( java.util.stream)** 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为Stream API可以极大提供Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。
- Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用 **Stream API** 对集合数据进行操作，就类似于使用 **SQL** 执行的数据库查询。也可以使用 **Stream API** 来并行执行操作。简言之，Stream API 提供了一种高效且易于使用的处理数据的方式。



# 为什么要使用Stream API

- 实际开发中，项目中多数数据源都来自于Mysql，Oracle等。但现在数据源可以更多了，有MongoDB，Redis等，而这些NoSQL的数据就需要Java层面去处理。
- Stream 和 Collection 集合的区别：Collection 是一种静态的内存数据结构，而 Stream 是有关计算的。前者是主要面向内存，存储在内存中，后者主要是面向 CPU，通过 CPU 实现计算。





# 什么是 Stream

## Stream到底是什么呢？

是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“集合讲的是数据，**Stream**讲的是计算！”

注意：

- ①Stream 自己不会存储元素。
- ②Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- ③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。



# Stream 的操作三个步骤

### ● 1- 创建 Stream

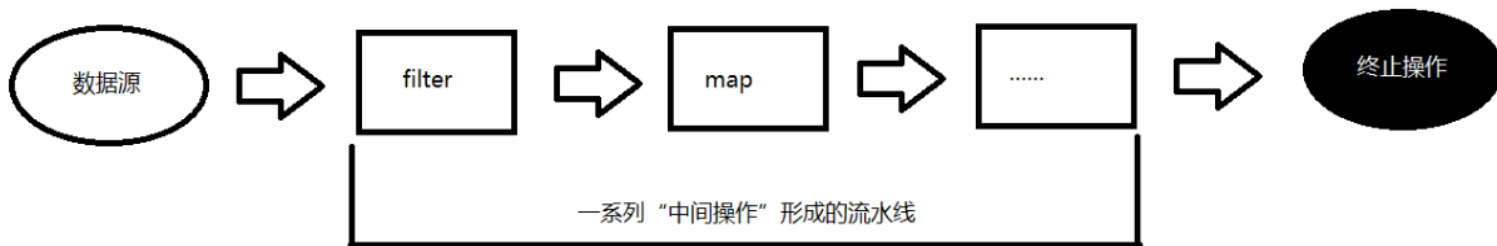
一个数据源（如：集合、数组），获取一个流

### ● 2- 中间操作

一个中间操作链，对数据源的数据进行处理

### ● 3- 终止操作(终端操作)

一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用





### 创建 Stream方式一：通过集合

Java8 中的 Collection 接口被扩展，提供了两个获取流的方法：

- **default Stream<E> stream()** : 返回一个顺序流
- **default Stream<E> parallelStream()** : 返回一个并行流



### 创建 Stream方式二：通过数组

Java8 中的 Arrays 的静态方法 `stream()` 可以获取数组流：

- **`static <T> Stream<T> stream(T[] array)`**: 返回一个流

重载形式，能够处理对应基本类型的数组：

- `public static IntStream stream(int[] array)`
- `public static LongStream stream(long[] array)`
- `public static DoubleStream stream(double[] array)`



### 创建 Stream方式三：通过Stream的of()

可以调用Stream类静态方法 `of()`，通过显示值创建一个流。它可以接收任意数量的参数。

- **`public static<T> Stream<T> of(T... values)` : 返回一个流**



### 创建 Stream 方式四：创建无限流

可以使用静态方法 `Stream.iterate()` 和 `Stream.generate()`，创建无限流。

- 迭代

```
public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

- 生成

```
public static<T> Stream<T> generate(Supplier<T> s)
```



// 方式四：创建无限流

@Test

```
public void test4() {
```

```
    // 迭代
```

```
    // public static<T> Stream<T> iterate(final T seed, final  
    // UnaryOperator<T> f)
```

```
    Stream<Integer> stream = Stream.iterate(0, x -> x + 2);  
    stream.limit(10).forEach(System.out::println);
```

```
    // 生成
```

```
    // public static<T> Stream<T> generate(Supplier<T> s)  
    Stream<Double> stream1 = Stream.generate(Math::random);  
    stream1.limit(10).forEach(System.out::println);
```

```
}
```



### Stream 的中间操作

多个**中间操作**可以连接起来形成一个**流水线**，除非流水线上触发终止操作，否则**中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”。**

#### 1-筛选与切片

方 法	描 述
<b>filter(Predicate p)</b>	接收 Lambda ， 从流中排除某些元素
<b>distinct()</b>	筛选，通过流所生成元素的 <b>hashCode() 和 equals()</b> 去除重复元素
<b>limit(long maxSize)</b>	截断流，使其元素不超过给定数量
<b>skip(long n)</b>	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 <b>limit(n)</b> 互补





# Stream 的中间操作

## 2-映 射

方法	描述
<b>map(Function f)</b>	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。
<b>mapToDouble(ToDoubleFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
<b>mapToInt(ToIntFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
<b>mapToLong(ToLongFunction f)</b>	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 LongStream。
<b>flatMap(Function f)</b>	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



# Stream 的中间操作

## 3-排序

方法	描述
<b>sorted()</b>	产生一个新流，其中按自然顺序排序
<b>sorted(Comparator com)</b>	产生一个新流，其中按比较器顺序排序



### Stream 的终止操作

- 终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至是 void。
- 流进行了终止操作后，不能再次使用。

#### 1-匹配与查找

方法	描述
<b>allMatch(Predicate p)</b>	检查是否匹配所有元素
<b>anyMatch(Predicate p)</b>	检查是否至少匹配一个元素
<b>noneMatch(Predicate p)</b>	检查是否没有匹配所有元素
<b>findFirst()</b>	返回第一个元素
<b>findAny()</b>	返回当前流中的任意元素



### Stream 的终止操作

方法	描述
<b>count()</b>	返回流中元素总数
<b>max(Comparator c)</b>	返回流中最大值
<b>min(Comparator c)</b>	返回流中最小值
<b>forEach(Consumer c)</b>	<b>内部迭代</b> (使用 Collection 接口需要用户去做迭代, 称为 <b>外部迭代</b> 。相反, Stream API 使用内部迭代——它帮你把迭代做了)



# Stream 的终止操作

## 2-归约

方法	描述
<b>reduce(T iden, BinaryOperator b)</b>	可以将流中元素反复结合起来，得到一个值。返回 T
<b>reduce(BinaryOperator b)</b>	可以将流中元素反复结合起来，得到一个值。返回 Optional<T>

备注：map 和 reduce 的连接通常称为 map-reduce 模式，因 Google 用它来进行网络搜索而出名。



# Stream 的终止操作

## 3-收集

方 法	描 述
<code>collect(Collector c)</code>	将流转换为其他形式。接收一个 <code>Collector</code> 接口的实现，用于给 <code>Stream</code> 中元素做汇总的方法

`Collector` 接口中方法的实现决定了如何对流执行收集的操作(如收集到 `List`、`Set`、`Map`)。

另外，`Collectors` 实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表：



## 16.4 强大的Stream API: Collectors

方法	返回类型	作用
<b>toList</b>	List<T>	把流中元素收集到List
<code>List&lt;Employee&gt; emps= list.stream().collect(Collectors.toList());</code>		
<b>toSet</b>	Set<T>	把流中元素收集到Set
<code>Set&lt;Employee&gt; emps= list.stream().collect(Collectors.toSet());</code>		
<b>toCollection</b>	Collection<T>	把流中元素收集到创建的集合
<code>Collection&lt;Employee&gt; emps =list.stream().collect(Collectors.toCollection(ArrayList::new));</code>		
<b>counting</b>	Long	计算流中元素的个数
<code>long count = list.stream().collect(Collectors.counting());</code>		
<b>summingInt</b>	Integer	对流中元素的整数属性求和
<code>int total=list.stream().collect(Collectors.summingInt(Employee::getSalary));</code>		
<b>averagingInt</b>	Double	计算流中元素Integer属性的平均值
<code>double avg = list.stream().collect(Collectors.averagingInt(Employee::getSalary));</code>		
<b>summarizingInt</b>	IntSummaryStatistics	收集流中Integer属性的统计值。如：平均值
<code>int SummaryStatisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));</code>		



## 16.4 强大的Stream API: Collectors



<b>joining</b>	String	连接流中每个字符串
<b>String str= list.stream().map(Employee::getName).collect(Collectors.joining());</b>		
<b>maxBy</b>	Optional<T>	根据比较器选择最大值
<b>Optional&lt;Emp&gt;max= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));</b>		
<b>minBy</b>	Optional<T>	根据比较器选择最小值
<b>Optional&lt;Emp&gt; min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));</b>		
<b>reducing</b>	归约产生的类型	从一个作为累加器的初始值开始，利用BinaryOperator与流中元素逐个结合，从而归约成单个值
<b>int total=list.stream().collect(Collectors.reducing(0, Employee::getSalar, Integer::sum));</b>		
<b>collectingAndThen</b>	转换函数返回的类型	包裹另一个收集器，对其结果转换函数
<b>int how= list.stream().collect(Collectors.collectingAndThen(Collectors.toList(), List::size));</b>		
<b>groupingBy</b>	Map<K, List<T>>	根据某属性值对流分组，属性为K，结果为V
<b>Map&lt;Emp.Status, List&lt;Emp&gt;&gt; map= list.stream().collect(Collectors.groupingBy(Employee::getStatus));</b>		
<b>partitioningBy</b>	Map<Boolean, List<T>>	根据true或false进行分区
<b>Map&lt;Boolean,List&lt;Emp&gt;&gt; vd = list.stream().collect(Collectors.partitioningBy(Employee::getManage));</b>		





## 16-5 Optional类



- 到目前为止，臭名昭著的空指针异常是导致Java应用程序失败的最常见原因。以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。
- Optional<T> 类(`java.util.Optional`) 是一个容器类，它可以保存类型T的值，代表这个值存在。或者仅仅保存null，表示这个值不存在。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。
- Optional类的Javadoc描述如下：这是一个可以为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提供很多有用的方法，这样我们就不用显式进行空值检测。
- 创建Optional类对象的方法：
  - **Optional.of(T t)** : 创建一个 Optional 实例，**t必须非空**；
  - **Optional.empty()** : 创建一个空的 Optional 实例
  - **Optional.ofNullable(T t)**: **t可以为null**
- 判断Optional容器中是否包含对象：
  - **boolean isPresent()** : 判断是否包含对象
  - **void ifPresent(Consumer<? super T> consumer)** : 如果有值，就执行Consumer接口的实现代码，并且该值会作为参数传给它。
- 获取Optional容器的对象：
  - **T get()**: 如果调用对象包含值，返回该值，否则抛异常
  - **T orElse(T other)** : 如果有值则将其返回，否则返回指定的other对象。
  - **T orElseGet(Supplier<? extends T> other)** : 如果有值则将其返回，否则返回由Supplier接口实现提供的对象。
  - **T orElseThrow(Supplier<? extends X> exceptionSupplier)** : 如果有值则将其返回，否则抛出由Supplier接口实现提供的异常。



## 16.5 Optional 类

```
@Test
public void test1() {
    Boy b = new Boy("张三");
    Optional<Girl> opt = Optional.ofNullable(b.getGrilFriend());
    // 如果女朋友存在就打印女朋友的信息
    opt.ifPresent(System.out::println);
}
```

```
@Test
public void test2() {
    Boy b = new Boy("张三");
    Optional<Girl> opt = Optional.ofNullable(b.getGrilFriend());
    // 如果有女朋友就返回他的女朋友，否则只能欣赏“嫦娥”了
    Girl girl = opt.orElse(new Girl("嫦娥"));
    System.out.println("他的女朋友是：" + girl.getName());
}
```



## 16.5 Optional 类

```
@Test
public void test3(){
    Optional<Employee> opt = Optional.of(new Employee("张三", 8888));
    //判断opt中员工对象是否满足条件, 如果满足就保留, 否则返回空
    Optional<Employee> emp = opt.filter(e -> e.getSalary()>10000);
    System.out.println(emp);
}

@Test
public void test4(){
    Optional<Employee> opt = Optional.of(new Employee("张三", 8888));
    //如果opt中员工对象不为空, 就涨薪10%
    Optional<Employee> emp = opt.map(e ->
    {e.setSalary(e.getSalary()%1.1);return e;});
    System.out.println(emp);
}
```

让天下没有难学的技术



尚硅谷