java 8 (http://39.106.3.150/tags/#java-8)

Java 8新特性总结

Posted by Palmer on 07-14，2019

Java 8 (又称为 jdk 1.8) 是 Java 语言开发的一个主要版本。 Oracle 公司于 2014 年 3 月 18 日发布 Java 8 ，它支持函数式编程，新的 JavaScript 引擎，新的日期 API，新的Stream API 等。
Java8 新增了非常多的特性，我们主要讨论以下几个：
Lambda 表达式 – 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。
方法引用 – 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
默认方法 – 默认方法就是一个在接口里面有了一个实现的方法。
新工具 – 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。
Stream API –新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。
Date Time API – 加强对日期与时间的处理。
Optional 类 – Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
Nashorn, JavaScript 引擎 – Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。
Java 8 希望有自己的编程风格，并与 Java 7 区别开，以下实例展示了 Java 7 和 Java 8 的编程格式：

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class Java8Tester {
   public static void main(String args[]){

      List<String> names1 = new ArrayList<String>();
      names1.add("Google ");
      names1.add("Runoob ");
      names1.add("Taobao ");
      names1.add("Baidu ");
      names1.add("Sina ");

      List<String> names2 = new ArrayList<String>();
      names2.add("Google ");
      names2.add("Runoob ");
      names2.add("Taobao ");
      names2.add("Baidu ");
      names2.add("Sina ");

      Java8Tester tester = new Java8Tester();
      System.out.println("使用 Java 7 语法: ");

      tester.sortUsingJava7(names1);
      System.out.println(names1);
      System.out.println("使用 Java 8 语法: ");

      tester.sortUsingJava8(names2);
      System.out.println(names2);
   }

   // 使用 java 7 排序
   private void sortUsingJava7(List<String> names){
      Collections.sort(names, new Comparator<String>() {
         @Override
         public int compare(String s1, String s2) {
            return s1.compareTo(s2);
         }
      });
   }

   // 使用 java 8 排序
   private void sortUsingJava8(List<String> names){
      Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
   }
}
```

Lambda 表达式

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）。< br> 使用 Lambda 表达式可以使代码变的更加简洁紧凑。
lambda 表达式的语法格式如下：

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

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。
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。
可选的大括号：如果主体包含了一个语句，就不需要使用大括号。
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，大括号需要指定明表达式返回了一个数值。
Lambda 表达式的简单例子:

```
// 1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值
x -> 2 * x

// 3. 接受2个参数(数字),并返回他们的差值
(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)
(String s) -> System.out.print(s)
```

```
public class Java8Tester {
   public static void main(String args[]){
      Java8Tester tester = new Java8Tester();

      // 类型声明
      MathOperation addition = (int a, int b) -> a + b;

      // 不用类型声明
      MathOperation subtraction = (a, b) -> a - b;

      // 大括号中的返回语句
      MathOperation multiplication = (int a, int b) -> { return a * b; };

      // 没有大括号及返回语句
      MathOperation division = (int a, int b) -> a / b;

      System.out.println("10 + 5 = " + tester.operate(10, 5, addition));
      System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));
      System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));
      System.out.println("10 / 5 = " + tester.operate(10, 5, division));

      // 不用括号
      GreetingService greetService1 = message ->
      System.out.println("Hello " + message);

      // 用括号
      GreetingService greetService2 = (message) ->
      System.out.println("Hello " + message);

      greetService1.sayMessage("Runoob");
      greetService2.sayMessage("Google");
   }

   interface MathOperation {
      int operation(int a, int b);
   }

   interface GreetingService {
      void sayMessage(String message);
   }

   private int operate(int a, int b, MathOperation mathOperation){
      return mathOperation.operation(a, b);
   }
}
```

输出结果：

```
$ javac Java8Tester.java
$ java Java8Tester
10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
Hello Runoob
Hello Google
```

使用 Lambda 表达式需要注意以下两点：

Lambda 表达式主要用来定义行内执行的方法类型接口，例如，一个简单方法接口。在上面例子中，我们使用各种类型的Lambda表达式来定义MathOperation接口的方法。然后我们定义了sayMessage的执行。

Lambda 表达式免去了使用匿名方法的麻烦，并且给予Java简单但是强大的函数化的编程能力。

变量作用域

lambda 表达式只能引用标记了 final 的外层局部变量，这就是说不能在lambda 内部修改定义在域外的局部变量，否则会编译错误。

在 Java8Tester.java 文件输入以下代码：

```
public class Java8Tester {

   final static String salutation = "Hello! ";

   public static void main(String args[]){
      GreetingService greetService1 = message ->
      System.out.println(salutation + message);
      greetService1.sayMessage("Runoob");
   }

   interface GreetingService {
      void sayMessage(String message);
   }
}
```

我们也可以直接在 lambda 表达式中访问外层的局部变量：

```
public class Java8Tester {
    public static void main(String args[]) {
        final int num = 1;
        Converter<Integer, String> s = (param) -> System.out.println(String.valueOf(p
        s.convert(2);  // 输出结果为 3
    }

    public interface Converter<T1, T2> {
        void convert(int i);
    }
}
```

lambda 表达式的局部变量可以不用声明为 final，但是必须不可被后面的代码修改（即隐性的具有 final 的语义）

```
int num = 1;
Converter<Integer, String> s = (param) -> System.out.println(String.valueOf(param + r
s.convert(2);
num = 5;
//报错信息：Local variable num defined in an enclosing scope must be final or effectiv
 final
```

在 Lambda 表达式当中不允许声明一个与局部变量同名的参数或者局部变量。

```
String first = "";
Comparator<String> comparator = (first, second) -> Integer.compare(first.length(), se
```

方法引用

方法引用通过方法的名字来指向一个方法。
方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
方法引用使用一对冒号 :: 。
下面，我们在 Car 类中定义了 4 个方法作为例子来区分 Java 中 4 种不同方法的引用。

```
package com.runoob.main;

@FunctionalInterface
public interface Supplier<T> {
    T get();
}

class Car {
    //Supplier是jdk1.8的接口，这里和lamda一起使用了
    public static Car create(final Supplier<Car> supplier) {
        return supplier.get();
    }

    public static void collide(final Car car) {
        System.out.println("Collided " + car.toString());
    }

    public void follow(final Car another) {
        System.out.println("Following the " + another.toString());
    }

    public void repair() {
        System.out.println("Repaired " + this.toString());
    }
}
```

构造器引用：它的语法是Class::new，或者更一般的Class< T >::new实例如下：

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );
```

静态方法引用：它的语法是Class::static_method，实例如下：

```
cars.forEach( Car::collide );
```

特定类的任意对象的方法引用：它的语法是Class::method实例如下：

```
cars.forEach( Car::repair );
```

特定对象的方法引用：它的语法是instance::method实例如下：

```
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

方法引用实例 在 Java8Tester.java 文件输入以下代码：

```
import java.util.List;
import java.util.ArrayList;

public class Java8Tester {
   public static void main(String args[]){
      List names = new ArrayList();

      names.add("Google");
      names.add("Runoob");
      names.add("Taobao");
      names.add("Baidu");
      names.add("Sina");

      names.forEach(System.out::println);
   }
}
```

函数式接口

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。函数式接口可以被隐式转换为 lambda 表达式。Lambda 表达式和方法引用（实际上也可认为是Lambda表达式）上。 如定义了一个函数式接口如下：

```
@FunctionalInterface
interface GreetingService
{
    void sayMessage(String message);
}
```

那么就可以使用Lambda表达式来表示该接口的一个实现(注：JAVA 8 之前一般是用匿名类实现的)：

```
GreetingService greetService1 = message -> System.out.println("Hello " + message);
```

函数式接口可以对现有的函数友好地支持 lambda。 JDK 1.8 之前已有的函数式接口:

```
java.lang.Runnable
java.util.concurrent.Callable
java.security.PrivilegedAction
java.util.Comparator
java.io.FileFilter
java.nio.file.PathMatcher
java.lang.reflect.InvocationHandler
java.beans.PropertyChangeListener
java.awt.event.ActionListener
javax.swing.event.ChangeListener
```

JDK 1.8 新增加的函数接口：

```
java.util.function
```

java.util.function 它包含了很多类，用来支持 Java的 函数式编程，该包中的函数式接口有：

序号	接口 & 描述
1	**BiConsumer<T,U>** 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
2	**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
3	**BinaryOperator<T>** 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
4	**BiPredicate<T,U>** 代表了一个两个参数的boolean值方法

5	**BooleanSupplier** 代表了boolean值结果的提供方
6	**Consumer<T>** 代表了接受一个输入参数并且无返回的操作
7	**DoubleBinaryOperator** 代表了作用于两个double值操作符的操作，并且返回了一个double值的结果。
8	**DoubleConsumer** 代表一个接受double值参数的操作，并且不返回结果。
9	**DoubleFunction<R>** 代表接受一个double值参数的方法，并且返回结果
10	**DoublePredicate** 代表一个拥有double值参数的boolean值方法
11	**DoubleSupplier** 代表一个double值结构的提供方

12	**DoubleToIntFunction** 接受一个double类型输入，返回一个int类型结果。

13	**DoubleToLongFunction** 接受一个double类型输入，返回一个long类型结果
14	**DoubleUnaryOperator** 接受一个参数同为类型double,返回值类型也为double 。
15	**Function<T,R>** 接受一个输入参数，返回一个结果。
16	**IntBinaryOperator** 接受两个参数同为类型int,返回值类型也为int 。
17	**IntConsumer** 接受一个int类型的输入参数，无返回值 。
18	**IntFunction<R>** 接受一个int类型输入参数，返回一个结果 。
19	**IntPredicate** ：接受一个int输入参数，返回一个布尔值的结果。

20	**IntSupplier** 无参数，返回一个int类型结果。

21	**IntToDoubleFunction** 接受一个int类型输入，返回一个double类型结果 。
22	**IntToLongFunction** 接受一个int类型输入，返回一个long类型结果。
23	**IntUnaryOperator** 接受一个参数同为类型int,返回值类型也为int 。
24	**LongBinaryOperator** 接受两个参数同为类型long,返回值类型也为long。
25	**LongConsumer** 接受一个long类型的输入参数，无返回值。
26	**LongFunction<R>** 接受一个long类型输入参数，返回一个结果。
27	**LongPredicate** R接受一个long输入参数，返回一个布尔值类型结果。

28	**LongSupplier** 无参数，返回一个结果long类型的值。

29	**LongToDoubleFunction** 接受一个long类型输入，返回一个double类型结果。
30	**LongToIntFunction** 接受一个long类型输入，返回一个int类型结果。
31	**LongUnaryOperator** 接受一个参数同为类型long,返回值类型也为long。
32	**ObjDoubleConsumer<T>** 接受一个object类型和一个double类型的输入参数，无返回值。
33	**ObjIntConsumer<T>** 接受一个object类型和一个int类型的输入参数，无返回值。
34	**ObjLongConsumer<T>** 接受一个object类型和一个long类型的输入参数，无返回值。
35	**Predicate<T>** 接受一个输入参数，返回一个布尔值结果。

36	**Supplier<T>** 无参数，返回一个结果。

37	**ToDoubleBiFunction<T,U>** 接受两个输入参数，返回一个double类型结果
38	**ToDoubleFunction<T>** 接受一个输入参数，返回一个double类型结果
39	**ToIntBiFunction<T,U>** 接受两个输入参数，返回一个int类型结果。
40	**ToIntFunction<T>** 接受一个输入参数，返回一个int类型结果。
41	**ToLongBiFunction<T,U>** 接受两个输入参数，返回一个long类型结果。
42	**ToLongFunction<T>** 接受一个输入参数，返回一个long类型结果。
43	**UnaryOperator<T>** 接受一个参数为类型T,返回值类型也为T。

函数式接口实例 Predicate 接口是一个函数式接口，它接受一个输入参数 T，返回一个布尔值结果。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）。该接口用于测试对象是 true 或 false。我们可以通过以下实例（Java8Tester.java）来了解函数式接口 Predicate 的使用：

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

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

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

      System.out.println("输出所有数据:");

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

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

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

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

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

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

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

关于 **@FunctionalInterface** 注解

Java 8为函数式接口引入了一个新注解@FunctionalInterface，主要用于编译级错误检查，加上该注解，当你写的接口不符合函数式接口定义的时候，编译器会报错。

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。

默认方法语法格式如下：

```
public class Java8Tester {
   public static void main(String args[]){
      Vehicle vehicle = new Car();
      vehicle.print();
   }
}

interface Vehicle {
   default void print(){
      System.out.println("我是一辆车!");
   }

   static void blowHorn(){
      System.out.println("按喇叭!!!");
   }
}

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

class Car implements Vehicle, FourWheeler {
   public void print(){
      Vehicle.super.print();
      FourWheeler.super.print();
      Vehicle.blowHorn();
      System.out.println("我是一辆汽车!");
   }
}
```

```
$ javac Java8Tester.java
$ java Java8Tester
我是一辆车!
我是一辆四轮车!
按喇叭!!!
我是一辆汽车!
```

Stream

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

Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。

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。

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。

数据源 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。

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。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.Map;

public class Java8Tester {
   public static void main(String args[]){
      System.out.println("使用 Java 7: ");

      // 计算空字符串
      List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
      System.out.println("列表: " +strings);
      long count = getCountEmptyStringUsingJava7(strings);

      System.out.println("空字符数量为: " + count);
      count = getCountLength3UsingJava7(strings);

      System.out.println("字符串长度为 3 的数量为: " + count);

      // 删除空字符串
      List<String> filtered = deleteEmptyStringsUsingJava7(strings);
      System.out.println("筛选后的列表: " + filtered);

      // 删除空字符串，并使用逗号把它们合并起来
      String mergedString = getMergedStringUsingJava7(strings,", ");
      System.out.println("合并字符串: " + mergedString);
      List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

      // 获取列表元素平方数
      List<Integer> squaresList = getSquares(numbers);
      System.out.println("平方数列表: " + squaresList);
      List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);

      System.out.println("列表: " +integers);
      System.out.println("列表中最大的数 : " + getMax(integers));
      System.out.println("列表中最小的数 : " + getMin(integers));
      System.out.println("所有数之和 : " + getSum(integers));
      System.out.println("平均数 : " + getAverage(integers));
      System.out.println("随机数: ");

      // 输出10个随机数
      Random random = new Random();

      for(int i=0; i < 10; i++){
         System.out.println(random.nextInt());
      }

      System.out.println("使用 Java 8: ");
      System.out.println("列表: " +strings);

      count = strings.stream().filter(string->string.isEmpty()).count();
      System.out.println("空字符串数量为: " + 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);

      filtered = strings.stream().filter(string ->!string.isEmpty()).collect(Collectc
      System.out.println("筛选后的列表: " + filtered);

      mergedString = strings.stream().filter(string ->!string.isEmpty()).collect(Coll
      System.out.println("合并字符串: " + mergedString);

      squaresList = numbers.stream().map( i ->i*i).distinct().collect(Collectors.toLi
      System.out.println("Squares List: " + squaresList);
      System.out.println("列表: " +integers);

      IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatist

      System.out.println("列表中最大的数 : " + stats.getMax());
      System.out.println("列表中最小的数 : " + stats.getMin());
      System.out.println("所有数之和 : " + stats.getSum());
      System.out.println("平均数 : " + stats.getAverage());
      System.out.println("随机数: ");

      random.ints().limit(10).sorted().forEach(System.out::println);

      // 并行处理
      count = strings.parallelStream().filter(string -> string.isEmpty()).count();
      System.out.println("空字符串的数量为: " + count);
   }

   private static int getCountEmptyStringUsingJava7(List<String> strings){
      int count = 0;

      for(String string: strings){

         if(string.isEmpty()){
            count++;
         }
      }
      return count;
   }

   private static int getCountLength3UsingJava7(List<String> strings){
      int count = 0;

      for(String string: strings){

         if(string.length() == 3){
            count++;
         }
      }
      return count;
   }

   private static List<String> deleteEmptyStringsUsingJava7(List<String> strings){
      List<String> filteredList = new ArrayList<String>();

      for(String string: strings){

         if(!string.isEmpty()){
             filteredList.add(string);
```

```
        }
    }
    return filteredList;
}

private static String getMergedStringUsingJava7(List<String> strings, String separ
    StringBuilder stringBuilder = new StringBuilder();

    for(String string: strings){

        if(!string.isEmpty()){
            stringBuilder.append(string);
            stringBuilder.append(separator);
        }
    }
    String mergedString = stringBuilder.toString();
    return mergedString.substring(0, mergedString.length()-2);
}

private static List<Integer> getSquares(List<Integer> numbers){
    List<Integer> squaresList = new ArrayList<Integer>();

    for(Integer number: numbers){
        Integer square = new Integer(number.intValue() * number.intValue());

        if(!squaresList.contains(square)){
            squaresList.add(square);
        }
    }
    return squaresList;
}

private static int getMax(List<Integer> numbers){
    int max = numbers.get(0);

    for(int i=1;i < numbers.size();i++){

        Integer number = numbers.get(i);

        if(number.intValue() > max){
            max = number.intValue();
        }
    }
    return max;
}

private static int getMin(List<Integer> numbers){
    int min = numbers.get(0);

    for(int i=1;i < numbers.size();i++){
        Integer number = numbers.get(i);

        if(number.intValue() < min){
            min = number.intValue();
        }
    }
    return min;
}
```

```
    private static int getSum(List numbers){
       int sum = (int)(numbers.get(0));

       for(int i=1;i < numbers.size();i++){
          sum += (int)numbers.get(i);
       }
       return sum;
    }

    private static int getAverage(List<Integer> numbers){
       return getSum(numbers) / numbers.size();
    }
}
```

执行以上脚本，输出结果为：

```
$ javac Java8Tester.java
$ java Java8Tester
使用 Java 7:
列表: [abc, , bc, efg, abcd, , jkl]
空字符数量为: 2
字符串长度为 3 的数量为: 3
筛选后的列表: [abc, bc, efg, abcd, jkl]
合并字符串: abc, bc, efg, abcd, jkl
平方数列表: [9, 4, 49, 25]
列表: [1, 2, 13, 4, 15, 6, 17, 8, 19]
列表中最大的数 : 19
列表中最小的数 : 1
所有数之和 : 85
平均数 : 9
随机数:
-393170844
-963842252
447036679
-1043163142
-881079698
221586850
-1101570113
576190039
-1045184578
1647841045
使用 Java 8:
列表: [abc, , bc, efg, abcd, , jkl]
空字符串数量为: 2
字符串长度为 3 的数量为: 3
筛选后的列表: [abc, bc, efg, abcd, jkl]
合并字符串: abc, bc, efg, abcd, jkl
Squares List: [9, 4, 49, 25]
列表: [1, 2, 13, 4, 15, 6, 17, 8, 19]
列表中最大的数 : 19
列表中最小的数 : 1
所有数之和 : 85
平均数 : 9.444444444444445
随机数:
-1743813696
-1301974944
-1299484995
-779981186
136544902
555792023
1243315896
1264920849
1472077135
1706423674
空字符串的数量为: 2
```

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。
类声明,以下是一个 java.util.Optional 类的声明：

```
public final class Optional<T>
extends Object
```

序号	方法 & 描述
1	**static <T> Optional<T> empty()** 返回空的 Optional 实例。
2	**boolean equals(Object obj)** 判断其他对象是否等于 Optional。
3	**Optional<T> filter(Predicate<? super <T> predicate)** 如果值存在，并且这个值匹配给定的 predicate，返回一个Optional用以描述这个值，否则返回一个空的Optional。
4	**<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)** 如果值存在，返回基于Optional包含的映射方法的值，否则返回一个空的Optional
5	**T get()** 如果在这个Optional中包含这个值，返回值，否则抛出异常：NoSuchElementException

6	**int hashCode()** 返回存在值的哈希码，如果值不存在 返回 0。
7	**void ifPresent(Consumer<? super T> consumer)** 如果值存在则使用该值调用 consumer , 否则不做任何事情。
8	**boolean isPresent()** 如果值存在则方法会返回true，否则返回 false。
9	**<U>Optional<U> map(Function<? super T,? extends U> mapper)** 如果有值，则对其执行调用映射函数得到返回值。如果返回值不为 null，则创建包含映射返回值的Optional作为map方法返回值，否则返回空Optional。
10	**static <T> Optional<T> of(T value)** 返回一个指定非null值的Optional。
11	**static <T> Optional<T> ofNullable(T value)** 如果为非空，返回 Optional 描述的指定值，否则返回空的 Optional。
12	**T orElse(T other)** 如果存在该值，返回值， 否则返回 other。
13	**T orElseGet(Supplier<? extends T> other)** 如果存在该值，返回值， 否则触发 other，并返回 other 调用的结果。

14	**<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)** 如果存在该值，返回包含的值，否则抛出由 Supplier 继承的异常
15	**String toString()** 返回一个Optional的非空字符串，用来调试

注意： 这些方法是从 java.lang.Object 类继承来的。
我们可以通过以下实例来更好的了解 Optional 类的使用：

```
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      // Optional.ofNullable - 允许传递为 null 参数
      Optional<Integer> a = Optional.ofNullable(value1);

      // Optional.of - 如果传递的参数是 null，抛出异常 NullPointerException
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      // Optional.isPresent - 判断值是否存在

      System.out.println("第一个参数值存在: " + a.isPresent());
      System.out.println("第二个参数值存在: " + b.isPresent());

      // Optional.orElse - 如果值存在，返回它，否则返回默认值
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - 获取值，值需要存在
      Integer value2 = b.get();
      return value1 + value2;
   }
}


$ javac Java8Tester.java
$ java Java8Tester
第一个参数值存在: false
第二个参数值存在: true
10
```

Nashorn JavaScript

Nashorn 一个 javascript 引擎。
从JDK 1.8开始，Nashorn取代Rhino(JDK 1.6, JDK1.7)成为Java的嵌入式JavaScript引擎。Nashorn完全支持ECMAScript 5.1规范以及一些扩展。它使用基于JSR 292的新语言特性，其中包含在JDK 7中引入的invokedynamic，将JavaScript编译成Java字节码。与先前的Rhino实现相比，这带来了2到10倍的性能提升。

jjs
jjs是个基于Nashorn引擎的命令行工具。它接受一些JavaScript源代码为参数，并且执行这些源代码。

例如，我们创建一个具有如下内容的sample.js文件：

```
print('Hello World!');
```

打开控制台，输入以下命令：

```
$ jjs sample.js
```

以上程序输出结果为：

```
Hello World!
```

jjs 交互式编程
打开控制台，输入以下命令：

```
$ jjs
jjs> print("Hello, World!")
Hello, World!
jjs> quit()
>>
```

传递参数
打开控制台，输入以下命令：

```
$ jjs -- a b c
jjs> print('字母: ' +arguments.join(", "))
字母: a, b, c
jjs>
```

Java 中调用 JavaScript
使用 ScriptEngineManager, JavaScript 代码可以在 Java 中执行，实例如下：

Java8Tester.java 文件

```
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;

public class Java8Tester {
   public static void main(String args[]){

      ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
      ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");

      String name = "Runoob";
      Integer result = null;

      try {
         nashorn.eval("print('" + name + "')");
         result = (Integer) nashorn.eval("10 + 2");

      }catch(ScriptException e){
         System.out.println("执行脚本错误: "+ e.getMessage());
      }

      System.out.println(result.toString());
   }
}
```

执行以上脚本，输出结果为：

```
$ javac Java8Tester.java
$ java Java8Tester
Runoob
12
JavaScript 中调用 Java
以下实例演示了如何在 JavaScript 中引用 Java 类：

var BigDecimal = Java.type('java.math.BigDecimal');

function calculate(amount, percentage) {

   var result = new BigDecimal(amount).multiply(
   new BigDecimal(percentage)).divide(new BigDecimal("100"), 2, BigDecimal.ROUND_HALF

   return result.toPlainString();
}

var result = calculate(5680000000000000000023,13.9);
print(result);
我们使用 jjs 命令执行以上脚本，输出结果如下：

$ jjs sample.js
78952000000000002017.94
```

日期时间 API

Java 8通过发布新的Date-Time API (JSR 310)来进一步加强对日期与时间的处理。在旧版的 Java 中，日期时间 API 存在诸多问题，其中有：

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。

设计很差 – Java的日期/时间类的定义并不一致，在java.util和java.sql的包中都有日期类，此外用于格式化和解析的类在java.text包中定义。java.util.Date同时包含日期和时间，而java.sql.Date仅包含日期，将其纳入java.sql包并不合理。另外这两个类都有相同的名字，这本身就是一个非常糟糕的设计。

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。

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。

LocalDate/LocalTime 和 LocalDateTime 类可以在处理时区不是必须的情况。代码如下：

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;

public class Java8Tester {
   public static void main(String args[]){
      Java8Tester java8tester = new Java8Tester();
      java8tester.testLocalDateTime();
   }

   public void testLocalDateTime(){

      // 获取当前的日期时间
      LocalDateTime currentTime = LocalDateTime.now();
      System.out.println("当前时间: " + currentTime);

      LocalDate date1 = currentTime.toLocalDate();
      System.out.println("date1: " + date1);

      Month month = currentTime.getMonth();
      int day = currentTime.getDayOfMonth();
      int seconds = currentTime.getSecond();

      System.out.println("月: " + month +", 日: " + day +", 秒: " + seconds);

      LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);
      System.out.println("date2: " + date2);

      // 12 december 2014
      LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
      System.out.println("date3: " + date3);

      // 22 小时 15 分钟
      LocalTime date4 = LocalTime.of(22, 15);
      System.out.println("date4: " + date4);

      // 解析字符串
      LocalTime date5 = LocalTime.parse("20:15:30");
      System.out.println("date5: " + date5);
   }
}
```

执行以上脚本，输出结果为：

```
$ javac Java8Tester.java
$ java Java8Tester
当前时间: 2016-04-15T16:55:48.668
date1: 2016-04-15
月: APRIL, 日: 15, 秒: 48
date2: 2012-04-10T16:55:48.668
date3: 2014-12-12
date4: 22:15
date5: 20:15:30
```

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8Tester {
   public static void main(String args[]){
      Java8Tester java8tester = new Java8Tester();
      java8tester.testZonedDateTime();
   }

   public void testZonedDateTime(){

      // 获取当前时间日期
      ZonedDateTime date1 = ZonedDateTime.parse("2015-12-03T10:15:30+05:30[Asia/Shanç
      System.out.println("date1: " + date1);

      ZoneId id = ZoneId.of("Europe/Paris");
      System.out.println("ZoneId: " + id);

      ZoneId currentZone = ZoneId.systemDefault();
      System.out.println("当期时区: " + currentZone);
   }
}

$ javac Java8Tester.java
$ java Java8Tester
date1: 2015-12-03T10:15:30+08:00[Asia/Shanghai]
ZoneId: Europe/Paris
当期时区: Asia/Shanghai
```

Base64

在Java 8中，Base64编码已经成为Java类库的标准。
Java 8 内置了 Base64 编码的编码器和解码器。

Base64工具类提供了一套静态方法获取下面三种BASE64编解码器：
基本：输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。
URL：输出映射到一组字符A-Za-z0-9+_，输出是URL和文件。
MIME：输出隐射到MIME友好格式。输出每行不超过76字符，并且使用'\r'并跟随'\n'作为分割。编码输出最后没有行分割。

以下实例演示了 Base64 的使用:

```
import java.util.Base64;
import java.util.UUID;
import java.io.UnsupportedEncodingException;

public class Java8Tester {
   public static void main(String args[]){
      try {

         // 使用基本编码
         String base64encodedString = Base64.getEncoder().encodeToString("runoob?java
         System.out.println("Base64 编码字符串 (基本) :" + base64encodedString);

         // 解码
         byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);

         System.out.println("原始字符串: " + new String(base64decodedBytes, "utf-8"));
         base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?
         System.out.println("Base64 编码字符串 (URL) :" + base64encodedString);

         StringBuilder stringBuilder = new StringBuilder();

         for (int i = 0; i < 10; ++i) {
            stringBuilder.append(UUID.randomUUID().toString());
         }

         byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");
         String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes)
         System.out.println("Base64 编码字符串 (MIME) :" + mimeEncodedString);

      }catch(UnsupportedEncodingException e){
         System.out.println("Error :" + e.getMessage());
      }
   }
}
```

执行以上脚本，输出结果为：

```
$ javac Java8Tester.java
$ java Java8Tester
原始字符串: runoob?java8
Base64 编码字符串 (URL) :VHV0b3JpYWxzUG9pbnQ_amF2YTg=
Base64 编码字符串 (MIME) :M2Q4YmUxMTEtYWRkZi00NzBlLTgyZDgtN2MwNjgzOGY2NGFlOTQ3NDYyMWEt
OTYtY2ZjMzZiMzFhNmZmOGJmOGI2OTYtMzkxZi00OTJiLWEyMTQtMjgwN2RjOGI0MTBmZWUwMGNk
NTktY2ZiZS00MTMxLTgzODctNDRjMjFkYmZmNGM4Njg1NDc3OGItNzNlMC00ZWM4LTgxNzAtNjY3
NTgyMGY3YzVhZWQyMmNiZGItOTIwZi00NGUzLTlkMjAtOTkzZTI1MjUwMDU5ZjdkYjg2M2UtZTJm
YS00Y2Y2LWIwNDYtNWQ2MGRiOWQyZjFiMzJhMzYxOWQtNDE0ZS00MmRiLTk3NDgtNmM4NTczYjMx
ZDIzNGRhOWU4NDAtNTBiMi00ZmE2LWE0M2ItZjU3MWFiNTI2NmQ2NTlmMTFmZjctYjg1NC00NmE1
LWEzMWItYjk3MmEwZTYyNTdk
```

← PREVIOUS POST (HTTP://39.106.3.150/ARCHIVES/20190726)

NEXT POST → (HTTP://39.106.3.150/ARCHIVES/REDIS)

...

撰写评论...

上一页 下一页

FEATURED TAGS **(http://39.106.3.150/tags/)**

java 8 (http://39.106.3.150/tags/#java-8) java8 (http://39.106.3.150/tags/#java8)
redis (http://39.106.3.150/tags/#redis) 监控 (http://39.106.3.150/tags/#1561876610222)
全链路 (http://39.106.3.150/tags/#1561876610220) 容器 (http://39.106.3.150/tags/#1560852708518)
开源框架 (http://39.106.3.150/tags/#1560569459781) Spring (http://39.106.3.150/tags/#spring)
设计模式 (http://39.106.3.150/tags/#1559888728999) linux (http://39.106.3.150/tags/#linux)
SpringBoot (http://39.106.3.150/tags/#springboot) 大数据 (http://39.106.3.150/tags/#1559363598973)
区块链 (http://39.106.3.150/tags/#1559363594390) Java (http://39.106.3.150/tags/#java)

FRIENDS

 (https://github.com/PowehiEdge)