前端：HTML/HTML5、CSS/CSS3、Javascript、jQuery、RequireJS、AngularJS、Vue

后端：Java、Struts2/Spring MVC、JPA/Mybatis、Spring Boot

安全：Shiro、Spring Security

中间件：Dubbo、ActiveMQ/RabbitMQ、Nginx

数据库：MySQL/Oracle、Redis、MongoDB

搜索引擎：Lucene、Elasticsearch

代码管理：svn、git

构建工具：Maven、Jenkins

spring boot :定时任务、批处理、Eureka、Thymeleaf、Jenkins；

Java:垃圾回收机制、类加载机制；

LDAP

数据库：JPA

前端：electron

**工具：**[**Notion数字笔记使用教程**](http://www.baidu.com/link?url=Jxv7rUU2NY-n6FBJedjl8OUg5Q17bt21sFAqe0VLMKuwmw7cjbb5IhxHkTWr_bcdUhXATJlwxu3W3vz9gWVnzOcUynuyqqu8aUhRcn114NG)

# 后端

## （一）Java8新特性

### 1．Lambda 表达式

#### 1.1语法

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）。

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。

#### 1.2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。

### 2.方法引用

方法引用通过方法的名字来指向一个方法。方法引用使用一对冒号 **::** 。

方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

### 3.函数式接口

函数式接口(FunctionalInterface)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。

函数式接口可以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。

### 4.默认方法

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。

在方法名前面加个default关键字即可实现默认方法。

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。

#### 4.1语法

默认方法语法格式如下：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

}

#### 4.2多个默认方法

一个接口有默认方法，考虑这样的情况，一个类实现了多个接口，且这些接口有相同的默认方法，以下实例说明了这种情况的解决方法：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

}

public interface fourWheeler {

    default void print() {

        System.out.println("我是一辆四轮车!");

    }

}

第一个解决方案是创建自己的默认方法，来覆盖重写接口的默认方法：

public class Car implements vehicle, fourWheeler {

    @Override

    public void print() {

        System.out.println("我是一辆四轮汽车!");

    }

}

第二种解决方案可以使用 super 来调用指定接口的默认方法：

public class Car implements vehicle, fourWheeler {

    @Override

    public void print() {

        vehicle.super.print();

    }

}

#### 4.3静态默认方法

Java 8 的另一个特性是接口可以声明（并且可以提供实现）静态方法。例如：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

    // 静态方法

    static void blowHorn() {

        System.out.println("按喇叭!!!");

    }

}

### 5.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

#### 5.1什么是 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。

#### 5.2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。

public static void main(String[] args) {

    List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");

    List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

}

#### 5.3 forEach

Stream 提供了新的方法 'forEach' 来迭代流中的每个数据。以下代码片段使用forEach 输出了10个随机数：

Random random = new Random();

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

#### 5.4 map

map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

// 获取对应的平方数

List<Integer> squaresList = numbers.stream().map(i -> i \* i).distinct().collect(Collectors.toList());

#### 5.5 filter

filter 方法用于通过设置条件过滤出元素。以下代码片段使用filter 方法过滤出空字符串：

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

// 获取空字符串的数量

int count = (int) strings.stream().filter(string -> string.isEmpty()).count();

#### 5.6 limit

limit 方法用于获取指定数量的流。以下代码片段使用 limit 方法打印出 10 条数据：

Random random = new Random();

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

#### 5.7 sorted

sorted 方法用于对流进行排序。以下代码片段使用 sorted 方法对输出的 10 个随机数进行排序：

Random random = new Random();

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

#### 5.8 并行（parallel）程序

parallelStream 是流并行处理程序的代替方法。以下实例我们使用parallelStream 来输出空字符串的数量：

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");

// 获取空字符串的数量

int count = (int) strings.parallelStream().filter(string -> string.isEmpty()).count();

我们可以很容易的在顺序运行和并行直接切换。

#### 5.9 Collectors

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。Collectors可用于返回列表或字符串：

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("筛选列表: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("合并字符串: " + mergedString);

#### 5.10统计

另外，一些产生统计结果的收集器也非常有用。它们主要用于int、double、long等基本类型上，它们可以用来产生类似如下的统计结果。

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();

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());

### 6.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。

#### 6.1类声明

以下是一个 java.util.Optional<T> 类的声明：

public final class Optional<T> extends Object

#### 6.2类方法

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Option 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)

如果存在该值，提供的映射方法，如果返回非null，返回一个Optional描述结果。

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。

### 7.Nashorn JavaScript

Nashorn 一个 javascript 引擎。

从JDK1.8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JSR292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。

#### 7.1 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：

HelloWorld!

#### 7.2 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()

>>

#### 7.3 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>

#### 7.4 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

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：

Runoob

12

#### 7.5 JavaScript 中调用 Java

以下实例演示了如何在 JavaScript 中引用 Java 类：

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

function calculate(amount, percentage){

   var result =newBigDecimal(amount).multiply(

   newBigDecimal(percentage)).divide(newBigDecimal("100"),2,BigDecimal.ROUND\_HALF\_EVEN);

   return result.toPlainString();

}

var result = calculate(568000000000000000023,13.9);

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

### 8.Java 8 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。

#### 8.1 本地化日期时间 API

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

Java8Tester.java文件

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：

当前时间: 2018-06-08T15:19:16.910

date1:2018-06-08

月: JUNE, 日: 8, 秒: 16

date2:2012-06-10T15:19:16.910

date3:2014-12-12

date4:22:15

date5:20:15:30

#### 8.2 使用时区的日期时间API

如果我们需要考虑到时区，就可以使用时区的日期时间API：

Java8Tester.java文件

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/Shanghai]");

        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);

    }

}

执行以上脚本，输出结果为：

date1:2015-12-03T10:15:30+08:00[Asia/Shanghai]

ZoneId:Europe/Paris

当期时区: Asia/Shanghai

### 9. Base64

在Java8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。

#### 9.1 内嵌类

序号 内嵌类 & 描述

1.static class Base64.Decoder

该类实现一个解码器用于，使用 Base64 编码来解码字节数据。

2.static class Base64.Encoder

该类实现一个编码器，使用 Base64 编码来编码字节数据。

#### 9.2 方法

序号 方法名 & 描述

1.static Base64.Decoder getDecoder()

返回一个 Base64.Decoder ，解码使用基本型 base64 编码方案。

2.static Base64.Encoder getEncoder()

返回一个 Base64.Encoder ，编码使用基本型 base64 编码方案。

3.static Base64.Decoder getMimeDecoder()

返回一个 Base64.Decoder ，解码使用 MIME 型 base64 编码方案。

4.static Base64.Encoder getMimeEncoder()

返回一个 Base64.Encoder ，编码使用 MIME 型 base64 编码方案。

5.static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)

返回一个 Base64.Encoder ，编码使用 MIME 型 base64 编码方案，可以通过参数指定每行的长度及行的分隔符。

6.static Base64.Decoder getUrlDecoder()

返回一个 Base64.Decoder ，解码使用 URL 和文件名安全型 base64 编码方案。

7.static Base64.Encoder getUrlEncoder()

返回一个 Base64.Encoder ，编码使用 URL 和文件名安全型 base64 编码方案。

注意：Base64 类的很多方法从 java.lang.Object 类继承。

#### 9.3 Base64 实例

以下实例演示了Base64 的使用:

Java8Tester.java文件

public class Java8Tester {

    public static void main(String args[]) {

        try {

            // 使用基本编码

            String base64encodedString = Base64.getEncoder().encodeToString("runoob?java8".getBytes("utf-8"));

            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?java8".getBytes("utf-8"));

            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：

Base64 编码字符串 (基本) :cnVub29iP2phdmE4

原始字符串: runoob?java8

Base64编码字符串(URL):VHV0b3JpYWxzUG9pbnQ\_amF2YTg=

Base64编码字符串(MIME):MjY5OGRlYmEtZDU0ZS00MjY0LWE3NmUtNzFiNTYwY2E4YjM1NmFmMDFlNzQtZDE2NC00MDk3LTlh

ZjItYzNkNGJjNmQwOWE2OWM0NDJiN2YtOGM4Ny00MjhkLWJkMzgtMGVlZjFkZjkyYjJhZDUwYzk0

ZWMtNDE5ZC00MTliLWEyMTAtZGMyMjVkYjZiOTE3ZTkxMjljMTgtNjJiZC00YTFiLTg3MzAtOTA0

YzdjYjgxYjQ0YTUxOWNkMTAtNjgxZi00YjQ0LWFkZGMtMzk1YzRkZjIwMjcyMzA0MTQzN2ItYzBk

My00MmQyLWJiZTUtOGM0MTlmMWIxM2MxYTY4NmNiOGEtNTkxZS00NDk1LThlN2EtM2RjMTZjMWJk

ZWQyZTdhNmZiNDgtNjdiYy00ZmFlLThjNTYtMjcyNDNhMTRhZTkyYjNiNWY2MmEtNTZhYS00ZDhk

LWEwZDYtY2I5ZTUwNzJhNGE1

# 数据库

# 前端

# WEB服务器

# 框架

# 构建工具

# 代码管理

# 中间件