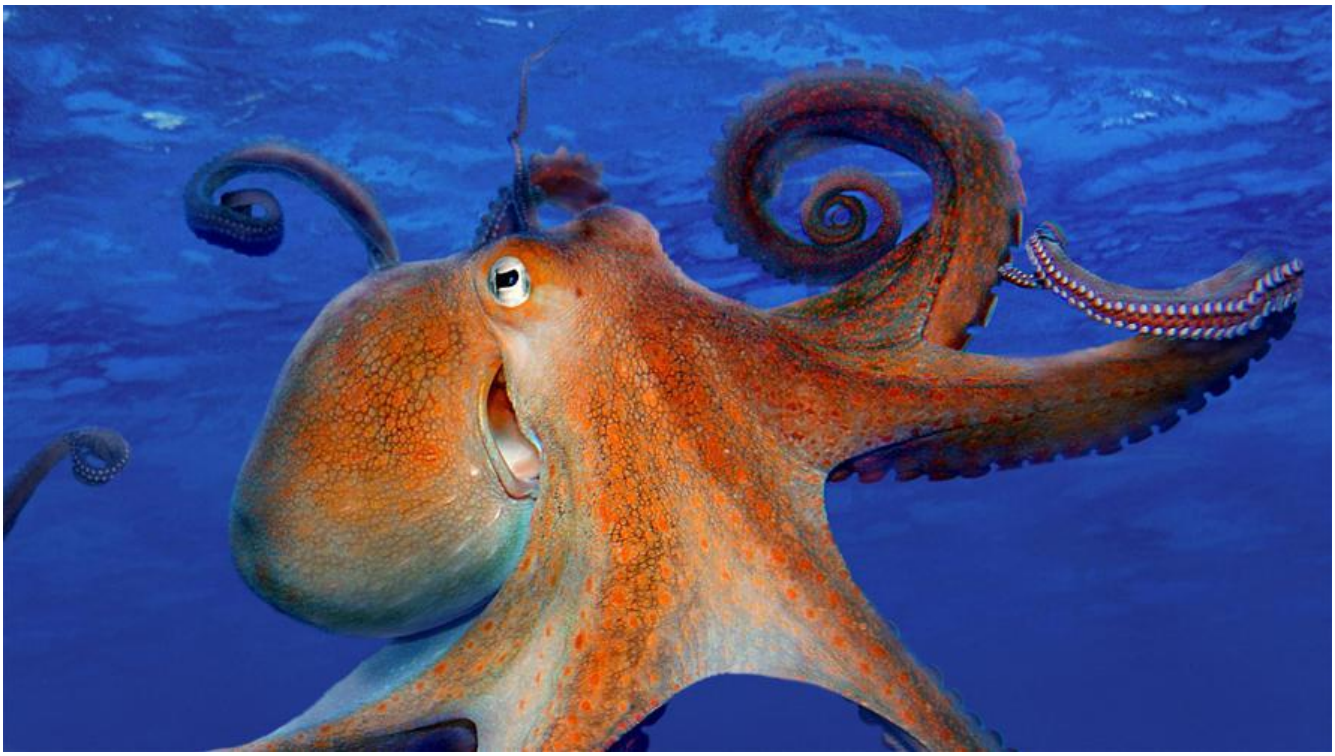
链滴

JDK8 新特性

作者：2457081614

原文链接：https://ld246.com/article/1576682053887

来源网站：链滴



概述

- 语法变化
  - Lamada 表达式
  - Default 方法
  - 重复注解
  - 扩展注解支持
- 集合
  - 新增 Stream 流
  - HashMap 性能提升
- 安全方面
  - 客户端 TLS 1.2 默认启用；
  - Sha-224 消息摘要；
  - 对高熵随机数生成的更好支持；
  - 持 Kerberos 5 协议转换和受限代理。
  - ...
- 工具
  - 提供 jjs 命令来调用 Nashorn 引擎；
  - Jdeps 命令行工具用于分析类文件；
  - JMX 提供远程访问诊断命令。

原文链接：JDK8 新特性

- 国际化
  - Unicode 增强，包括对 Unicode 6.2.0 的支持；
  - 新的日历和语言环境 API；
  - 可安装自定义资源包。
- 新增日期处理包
- IO 和 NIO
  - 减小 <JDK_HOME>/jre/lib/charsets.jar 文大小；
  - java.lang.String(byte[], *)和 java.lang.String.getBytes()性能提升。
- 新增工具类
  - 并行数组排序；
  - Base64 编码解码；
  - 无符号算术支持 。
- 网络
  - 新增 java.net.URLPermission；
  - 在 java. net 类中。 如果安装了安全管理器，则 HttpURLConnection 调用打开连接的请求需权限。
- 并发
  - 添加新的类和接口

JDK8 之 default 关键字

在 jdk1.8 以前接口里面是只能有抽象 方法，不能有任何 方法的实现。jdk1.8里 面打破了这个规定，入了新的关键字 default，使用 default 修饰方法，可以在接口里面定义具体的方法实现。

- 默认方法：接口里面定义一个默认方法，这个接口的实现类实现了这个接口之后，不用管这个 defaut 修饰的方法就可以直接调调用，即接口方法的默认实现
- 静态方法: 接口名.静态方法来访问接口中的静态方法。

```
public interface StudentService {

    /**
     * JAVA8中新增default关键字
     */
    default void test()
    {
        System.out.print("hello world");
    }


    static void test1() {
        System.out.println("这是静态方法");
```

```
    }
}
```

JDK8 之 base64 加解密 API

Jdk1.8 的 java.util 包中，新增了 Base64 的类。相比于传统 sun.misc 和 Apache Commons Codec 效率较高，不用导包。

```
public class Base64Demo {

    /**
     * java 8新增 base64编码解码
     * @param args
     * @throws UnsupportedEncodingException
     */
    public static void main(String[] args) throws UnsupportedEncodingException {
        Base64.Encoder encoder = Base64.getEncoder();
        Base64.Decoder decoder = Base64.getDecoder();

        byte[] src = "hello world".getBytes();
        String encodeData = encoder.encodeToString(src);
        System.out.println("编码数据：  "+encodeData);

        System.out.println("解码数据:"+ new String(decoder.decode(encodeData),"UTF-8"));
    }
}
```

JDK8 之时间日期处理类

JAVA8 新增 LocalDate、LocalTime、LocalDateTime 日期处理类，LocalDate 只精确到天、LocalTime 只包含具体时间（时分秒）、LocalDateTime 包含日期和时间。新增 DateTimeFormatter(线程全)和 Duration 对时间的处理变得极为方便，具体使用如下。

```
class LocalDateDemo {

    public static void main(String[] args) throws InterruptedException {

        /**
         * LocalDate不包含具体时间（小时、分、秒），只有日期
         */
        LocalDate localDate = LocalDate.now();

        System.out.println("当前时间:"+ localDate);
        System.out.println("当前月份"+localDate.getMonthValue());

        //增加
        LocalDate newLocalDate = localDate.plusYears(2);
        System.out.println("增加的时间"+newLocalDate);

        //减小
        LocalDate minLocalDate = localDate.minusYears(66);
```

```
        System.out.println("减小的时间"+minLocalDate);

        //修改月份
        LocalDate localDate1 =localDate.withMonth(5);
        System.out.println(localDate1);

        LocalDateTime localDateTime = LocalDateTime.now();
        Thread.sleep(1111);
        System.out.println(localDateTime.isAfter(LocalDateTime.now()));


        //DateTimeFormatter线程安全
        LocalDateTime ldt = LocalDateTime.now();
        System.out.println(ldt);
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String ldtStr = dtf.format(ldt);
        System.out.println(ldtStr);

        //时间比较
        LocalDateTime today = LocalDateTime.now();
        System.out.println(today);
        LocalDateTime changeDate = LocalDateTime.of(2020,10,1,10,40,30);
        System.out.println(changeDate);
        Duration duration = Duration.between( today,changeDate);//第二个参数减第一个参数

        System.out.println(duration.toDays());//两个时间差的天数
        System.out.println(duration.toHours());//两个时间差的小时数
        System.out.println(duration.toMinutes());//两个时间差的分钟数
        System.out.println(duration.toMillis());//两个时间差的毫秒数
        System.out.println(duration.toNanos());//两个时间差的纳秒数


    }
}
```

JDK8 之 Lambda 表达式

在 JDK8 之前，Java 是不志持函数式编程的，所谓的函数编程，即可理解是将一个函数（也称为“行”）作为 一个参数进 行传递， 面向对象编程是对数据的抽象，而函数式编程则是对 行为的抽象（将为作为 一个参数进行传递）。

lambda 表达式 使 用场景(前提)：一个接 口中只包含 一个方法，则可以使 用 Lambda 表达式，这的接口称之为“函数接口” 语法： (params) -> ex
ression。

```
public class LamadaDemo {

    public static void main(String[] args) {

        /**
         * JDK8之前创建线程
         */
        new Thread(new Runnable() {
            @Override
```

```
            public void run() {
                System.out.println("hello world");
            }
        });

        //JDK8
        new Thread(()->System.out.println("hello world"));


        //JDK8之前排序
        List<String> list = Arrays.asList("aaa","ggg","ffff","ccc");
        Collections.sort(list, new Comparator<String>() {
                    @Override
                    public int compare(String a, String b) {
                        return b.compareTo(a);
                    }
                }
        );
        for (String string : list) {
            System.out.println(string);
        }

        //JDK8排序
        Collections.sort(list, (a,b)->b.compareTo(a)
        );
        for (String string : list) {
            System.out.println(string);
        }


    }
}
```

自定义函数编程

定义一个函数式接口：

```
//声明这是一个函数式接口
@FunctionalInterface
public interface MyLamada<R,T> {
    R operator(T t1,T t2);
}
```

测试：

```
public class MyLamadaTest {

    public static void main(String[] args) throws Exception {
        System.out.println(operator(20, 5, (Integer x, Integer y) -> {
            return x * y;
        }));
        System.out.println(operator(20, 5, (x, y) -> x + y));
        System.out.println(operator(20, 5, (x, y) -> x - y));
        System.out.println(operator(20, 5, (x, y) -> x / y));
```

```
    }
    public static Integer operator(Integer x, Integer y,
                                   MyLamada<Integer, Integer> of) {
        return of.operator(x, y);
    }
}
```

JDK8 函数式编程

Lambda 表达式必须先定义接口，创建相关 方法之后才可使用，这样做 十分不便，其实 java8 已经置了许多接口, 例如下面四个功能型接口，所以 一般很少会由用户去定义新的函数式接口。Java8 内的四大核心函数式接口：

1. Consumer <T> : 消费型接口：有入参，无返回值
2. Supplier <T> : 供给型接口：无入参，有返回值
3. Function<T, R> : 函数型接口：有入参，
返回值
4. Predicate <T> : 断言型接口：有入参，有返回值，返回值类型确定是 boolean

```
public class FunctionDemo {

    public static void main(String[] args) {
        //声明function函数
        Function<Integer,Integer> function = p->p*10;
        System.out.println(function.apply(99));

        //自定义function函数，只能穿一个参数
        MyFunction<String,String> myFunction = new MyFunction<>();
        System.out.println(myFunction.apply("xw study"));

        //BiFunction支持传两个参数
        BiFunction<String,String,String> biFunction = (a,b)->a+b;
        System.out.println(biFunction.apply("a","b"));


    }
}


/**
 * @author by xw
 * @Description predicate有入参，有返回值，返回类型是boolean类型
 */
public class PredicateDemo {
    public static void main(String[] args) {
        List<String> list =
                Arrays.asList("awewrwe","vdssdsd","aoooo","psdddsd");
        List<String> results = filter(list,obj->obj.startsWith("a"));
        System.out.println(results);
    }
    public static List<String> filter(List<String> list,
                                      Predicate<String> predicate) {
```

```
        List<String> results = new ArrayList<>();
        for (String str : list) {
            if (predicate.test(str)) {
                results.add(str);
            }
        }
        return results;
    }
}



/**
 * @author by xw
 * @Description supplier只有返回值，没有入参
 */
public class SupplierDemo {
    public static void main(String[] args) {
        //Student student = new Student();
        Student student = newStudent();
        System.out.println(student.getName());
    }
    public static Student newStudent(){
        Supplier<Student> supplier = ()-> {
            Student student = new Student();
            student.setName("默认名称");
            return student;
        };
        return supplier.get();
    }

    static class Student{
        private String name;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
    }
}

public class ConsumerDemo {

    public static void main(String[] args) throws Exception {
        Consumer<String> consumer = obj->{
            System.out.println(obj);
            System.out.println("调用短信接口发送短信，或者打印日志");
        };

        sendMsg("8888888",consumer);
    }
    public static void sendMsg(String phone, Consumer<String> consumer){
```

```
        consumer.accept(phone);
    }
}
```

未完待续。。。。。

项目地址：项目地址

JDK8 之 Stream 流

Stream 中 文称为 “流”，通过将集合转换为这么 一种叫做 “流”的元素队列，通过声明性方式，够对集合中的每个元素进 行一系列并行或串行的流水线操作。

操作过程

1. 数据元素便是原始集合，如 List、Set、Map 等
2. 生成流，可以是串行流 stream() 或者并行流 parallelStream()
3.中间操作，可以是 排序，聚合，过滤，转换等
3. 终端操作，很多流操作本身就会返回一个流，所以多个操作可以直接连接起来，最后统一进行收集

函数详解

map 函数

- 作用：将流中的每 一个元素 T 映射为 R（类似类型转换）
- 应用场景: 开发中 Do 转 Dto

```
private void mapTest()
    {
        List<User> list = Arrays.asList(new User(1,"小东","123"),new
                    User(21,"jack","rawer"),
                new User(155,"tom","sadfsdfsdfsd"),
                new User(231,"marry","234324"),new User(100,"2222","122223"));
        List<UserDTO> userDTOList = list.stream().map(obj->{
            UserDTO userDTO = new UserDTO(obj.getId(),obj.getName());
            return userDTO;
        }).collect(Collectors.toList());
        System.out.println(userDTOList);

    }
```

filter 函数

- 作用：用于通过设置的条件过滤出元素

```
// 过滤长度大于5的
 private void filterTest() {
        List<String> list = Arrays.asList("springboot", "springcloud",
                "redis", "git", "netty", "java", "html", "docker");
        List<String> resultList = list.stream().filter(obj -> obj.length() >
```

```
            5).collect(Collectors.toList());
        System.out.println(resultList);
    }
```

sorted 函数

作用：sorted() 对流进 行排序, 其中的元素必须实现 Comparable 接口

```
private void sortdTest() {
        //自然排序
        List<String> list = Arrays.asList("springboot", "springcloud",
                "redis", "git", "netty", "java", "html", "docker");
        List<String> resultList =
                list.stream().sorted().collect(Collectors.toList());

        //根据长度顺序进行排序
        resultList =
                list.stream().sorted(Comparator.comparing(obj ->
                        obj.length())).collect(Collectors.toList());
        //逆序排序
        resultList =
                list.stream().sorted(Comparator.comparing(obj ->
                        obj.length(), Comparator.reverseOrder())).collect(Collectors.toList()
                );
        // 逆序排序，根据effective java规范一般来说在stream表达式中，引用优先于lamada表达式
用。
        resultList =
                list.stream().sorted(Comparator.comparing(String::length).reversed()
                ).collect(Collectors.toList());
        System.out.println(resultList);
    }
```

limit 函数

作用：截断流使其最多只包含指定数量的元素。

```
private static void limitTest() {
        List<String> list = Arrays.asList("springboot", "springcloud",
                "redis", "git", "netty", "java", "html", "docker");
        //limit截取
        List<String> resultList =
                list.stream().sorted(Comparator.comparing(String::length).reversed()
                ).limit(3).collect(Collectors.toList());

        // result  [springcloud, springboot, docker]
        System.out.println(resultList);
    }
```

allMatch 和 anyMatch

allMatch 检查是否匹配所有元素，只有全部符合才返回 true。anyMatch 检查是否至少匹配一个元
。

```
private static void anyMatchAndAllMatchTest()
    {
        List<String> list = Arrays.asList("springboot", "springcloud", "redis",
                "git", "netty", "java", "html", "docker");
        boolean flag = list.stream().allMatch(obj->obj.length()>1);
        System.out.println(flag);

        flag = list.stream().anyMatch(obj->obj.length()>18);
        System.out.println(flag);
    }
```

min 和 max 函数

```
private static void minAndMaxTest() {
        List<Student> list = Arrays.asList(new Student(32), new
                Student(33), new Student(21), new Student(29), new Student(18));

        //求最大值
        Optional<Student> optional = list.stream().max(Comparator.comparingInt(Student::get
ge));
        System.out.println(optional.get().getAge());

        //求最小值
        optional = list.stream().min(Comparator.comparingInt(Student::getAge));
        System.out.println(optional.get().getAge());
    }
```

JDK8 之 Optional 类

- 作用
  - 解决空指针异常
- 创建
  - of()方法，null 值作为参数传递进去,则会抛异常

```
Optional opt = Optional.of(user);
```

  - ofNullable()，如果对象即可能是 null 也可能是 非 null，应该使 用 ofNullable() 方法

```
Optional opt = Optional.ofNullable(user);
```

  - orElse()如果有值则返回该值，否则返回传递给它的参数值

```
package jdk8.optional;


import java.util.Optional;

public class Demo {

    public static void main(String[] args) {
```

```
        User user = null;

        // 如果user是null的话，下面这句话会报错。
        //Optional<User> opt = Optional.of(user);

        //  orElse()如果有值则返回该值，否则返回传递给它的参数值
        User user1 = new User("1");
        User user2 = new User("2");
        /*
        如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象一般使用get之前需要
        先验证是否有值，不然还会报错*/
        Optional<User> opt = Optional.ofNullable(user);
        Optional<User> opt1 = Optional.ofNullable(user1);
        System.out.println("opt " + opt.isPresent());          //false
        System.out.println("opt1 " + opt1.isPresent());       //true
        System.out.println(Optional.ofNullable(user1).orElse(user2));  //user1
    }
}
```

reduce 函数

作用：聚合函数，根据 一定的规则将 Stream 中的元素进行计算后返回 一个唯一的值。

```
int value = Stream.of(1, 2, 3, 4, 5).reduce((item1, item2) -> item1 + item2).get();
```

foreach 函数

作用：循环遍历集合

```
List results = Arrays.asList(new Student(32),new Student(33),new Student(21),new Student(29)
new Student(18));
results.forEach(obj->{ System.out.println(obj.toString()); });
```

注意点：

- 不能修改包含外部的变量的值
- 不能 用 break 或者 return 或者 continue 等关键词结束或者跳过循环

collector 收集器

作用：一个终端操作, 用于对流中的数据进 行归集操作。

```
 private static  void collectTest()
    {
        List<Integer> a =Arrays.asList(1,2,3);
        List<Integer> b = a.stream().collect(Collectors.toList());
        Set<Integer> c = a.stream().collect(Collectors.toSet());

    }
```

joining 函数

作用：拼接函数

```
/**
    * 该方法可以将Stream得到一个字符串， joining函数接受三个参数，分别表示 元素之间的连
    * 接符、前缀和后缀。
    */
   private static void joiningTest()
   {

       //3种重载方法
      /* Collectors.joining()
       Collectors.joining("param")
       Collectors.joining("param1", "param2", "param3")*/

       String result = Stream.of("springboot", "mysql", "html5",
               "css3").collect(Collectors.joining(",", "[", "]"));
       System.out.println(result); //[springboot,mysql,html5,css3]

   }
```

partitioningBy 分组

Collectors.partitioningBy 分组，key 是 boolean 类型。

```
private static void parttitionByTest()
   {
       List<String> list = Arrays.asList("java", "springboot",
               "HTML5","nodejs","CSS3");
       Map<Boolean, List<String>> result =
               list.stream().collect(partitioningBy(obj -> obj.length() > 4));
       System.out.println(result);

       //key是 false 和true
       //result
      //{false=[java, CSS3], true=[springboot, HTML5, nodejs]}

   }
```

group by 分组

```
private static void groupbyingTest()
   {
       List<Student> students = Arrays.asList(new Student("广东", 23), new
               Student("广东", 24), new Student("广东", 23),new Student("北京", 22), new
               Student("北京", 20), new Student("北京", 20),new Student("海南", 25));
       Map<String, Long> listMap =
               students.stream().collect(Collectors.groupingBy(Student::getProvince,
                       Collectors.counting()));
       listMap.forEach((key, value) -> {System.out.println(key+"省人数有"+value);});
   }
```

summarizing 集合统计

summarizing 统计相关。

```
private static void summarizingTest() {
        List<Student> students = Arrays.asList(new Student("广东", 23), new
                Student("广东", 24), new Student("广东", 23), new Student("北京", 22), new
                Student("北京", 20), new Student("北京", 20), new Student("海南", 25));
        IntSummaryStatistics summaryStatistics =
                students.stream().collect(Collectors.summarizingInt(Student::getAge));
        System.out.println("平均值：" + summaryStatistics.getAverage());
        System.out.println("人数：" + summaryStatistics.getCount());
        System.out.println("最大值：" + summaryStatistics.getMax());
        System.out.println("最小值：" + summaryStatistics.getMin());
        System.out.println("总和：" + summaryStatistics.getSum());
    }
```

JDK8 之新的内存空间

元空间（Metaspace）

在 JDK8 及版本，有个区域叫做“永久代(permanent generation)， 通过 在命令 行设置参数-XX:MaPermSize 来设定永久代最 大可分配的内存空间。该块内存主要是被 JVM用来存放 class 和 mate 信的，当 class 被加载 loader 的时候就会 被存储到该内存区中，如 方法的编译信息及字节码、常量池符号解析、类的层级信息，字段，名字等

在 JDK8 使用本地内存来存储类元数据信息，叫做元空间，另外将常量及静态变量移到堆中，元空间不在虚拟机中，存储在本地内存里面，在默认的情况下 Metaspace 的大小只与本地内存有关。默认设置元空间大小的话会自动扩张，设置带下命令

-XX: MaxPermSize

优点：

- 原来的字符串存在永久代中，容易出现性能问题和内存溢出。
- 类及方法的信息难以确定，指定永久代的大小比较困难，太小导致出现永久代溢出，太大容易导致年代溢出。
- 永久代 GC 效率低

参考文档：

- open jdk8 updates wiki