Java8 新特性

> Author：zhangzhang
>
> Version：1.0.0

一、Java8概述

> Java8 (又称 JKD1.8) 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Java8 。
>
> - 支持Lambda表达式
> - 函数式接口
> - 新的Stream API
> - 新的日期 API
> - 其他特性

二、Lambda表达式

2.1 概念

> - Lambda表达式是特殊的匿名内部类，语法更简洁。
> - Lambda表达式允许把函数作为一个方法的参数（函数作为方法参数传递），将代码像数据一样传递。

2.2 语法

> <函数式接口> <变量名> = (参数1，参数2...) -> {}()
>
> //方法体
>
> {};

2.3 基本使用

> 演示案例：

```
public class Demo1 {
  public static void main(String[] args) {
    //匿名内部类
    Runnable runnable=new Runnable() {

      @Override
      public void run() {
        System.out.println("子线程执行了.....");
      }
    };
```

```
    //Lambda表达式
    Runnable runnable2=()->System.out.println("子线程执行了2.........");

    new Thread(runnable2).start();
    new Thread(()->System.out.println("子线程执行了3.........")).start();

    ////////////////////////////////
    //匿名内部类
    Comparator<String> com=new Comparator<String>() {

      @Override
      public int compare(String o1, String o2) {
        // TODO Auto-generated method stub
        return o1.length()-o2.length();
      }
    };

    //Lambda表达式

    Comparator<String> com2=(String o1, String o2)-> {
      // TODO Auto-generated method stub
      return o1.length()-o2.length();
    };

    Comparator<String> com3=(o1,o2)->o1.length()-o2.length();

    TreeSet<String> treeSet=new TreeSet<>(com3);
  }
}
```

> Lambda引入了新的操作符：->(箭头操作符)，->将表达式分成两部分：
>
> - 左侧：(参数1，参数2...)表示参数列表
> - 右侧：{}内部是方法体

注意事项：

- **形参列表的数据类型会自动推断。**
- **如果形参列表为空，只需保留() 。**
- **如果形参只有1个，()可以省略，只需要参数的名称即可。**
- **如果执行语句只有一句，且无返回值，{}可以省略，若有返回值，则若想省去{}，则必须同时省略return，且执行语句也保证只有一句。**
- **Lambda不会生成一个单独的内部类文件。**

2.4 课堂案例

> Usb接口：

```
@FunctionalInterface
public interface Usb {
  void service();
}
```

> TestUsb类：

```
public class TestUsb {
  public static void main(String[] args) {
    //匿名内部类
    Usb mouse=new Usb() {

      @Override
      public void service() {
        System.out.println("鼠标开始工作了.........");
      }
    };

    Usb fan=()->System.out.println("风扇开始工作了.........");


    run(mouse);
    run(fan);
  }
  public static void run(Usb usb) {
    usb.service();
  }
}
```

三、函数式接口【重点】

3.1 概念

> - 如果一个接口只有一个抽象方法，则该接口称之为函数式接口。
> - 函数式接口可以使用Lambda表达式，Lambda表达式会被匹配到这个抽象方法上 。
>   - @FunctionalInterface 注解检测接口是否符合函数式接口规范。

3.2 常见函数式接口

接口	参数类型	返回类型	说明
Consumer< T > 消费型接口	T	void	void accept(T t);对类型为T的对象应用操作
Supplier< T > 供给型接口	无	T	T get(); 返回类型为T的对象
Function< T,R > 函数型接口	T	R	R apply(T t);对类型为T的对象应用操作，并返回类型为R类型的对象。
Predicate< T > 断言型接口	T	boolean	boolean test(T t);确定类型为T的对象是否满足条件，并返回boolean类型。

> 案例演示：

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

    //Lambda表达式
    Consumer<Double> consumer= t->System.out.println("聚餐消费:"+t);

    happy(t->System.out.println("聚餐消费:"+t), 1000);
    happy(t->System.out.println("唱歌消费:"+t), 2000);

    int[] arr=getNums(()->new Random().nextInt(100), 5);
    System.out.println(Arrays.toString(arr));
    int[] arr2=getNums(()->new Random().nextInt(1000), 10);
    System.out.println(Arrays.toString(arr2));

    String result=handlerString(s->s.toUpperCase(), "hello");
    System.out.println(result);
    String result2=handlerString(s->s.trim(), "   zhangsan        ");
    System.out.println(result2);

    List<String> list=new ArrayList<>();
    list.add("zhangsan");
    list.add("zhangwuji");
    list.add("lisi");
    list.add("wangwu");
    list.add("zhaoliu");
    List<String> result=filterNames(s->s.startsWith("zhang"), list);
    System.out.println(result.toString());

    List<String> result2=filterNames(s->s.length()>5, list);
    System.out.println(result2);
  }
  //Consumer 消费型接口
  public static void happy(Consumer<Double> consumer,double money) {
    consumer.accept(money);
  }

  //Supplier 供给型接口
  public static int[] getNums(Supplier<Integer> supplier,int count) {
    int[] arr=new int[count];
    for(int i=0;i<count;i++) {
      arr[i]=supplier.get();
    }
    return arr;
  }

  //Function函数型接口
  public static String handlerString(Function<String, String> function,String str) {
    return function.apply(str);
  }

  //Predicate 断言型接口

  public static List<String> filterNames(Predicate<String> predicate,List<String> list){
    List<String> resultList=new ArrayList<String>();
    for (String string : list) {
      if(predicate.test(string)) {
        resultList.add(string);
      }
    }
    return resultList;
  }

}
```

四、方法引用

4.1 概念

> - 方法引用是Lambda表达式的一种简写形式。
> - 如果Lambda表达式方法体中只是调用一个特定的已经存在的方法，则可以使用方法引用。

> 常见形式：
>
> - 对象::实例方法
> - 类::静态方法
> - 类::实例方法
> - 类::new

4.2 基本使用

> Employee类：

```
public class Employee {
  private String name;
  private double money;
  public Employee() {
    // TODO Auto-generated constructor stub
  }

  public Employee(String name, double money) {
    super();
    this.name = name;
    this.money = money;
  }

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public double getMoney() {
    return money;
  }
  public void setMoney(double money) {
    this.money = money;
  }
  @Override
  public String toString() {
    return "Employee [name=" + name + ", money=" + money + "]";
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    long temp;
    temp = Double.doubleToLongBits(money);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Employee other = (Employee) obj;
    if (Double.doubleToLongBits(money) != Double.doubleToLongBits(other.money))
      return false;
    if (name == null) {
      if (other.name != null)
        return false;
    } else if (!name.equals(other.name))
      return false;
    return true;
  }

}
```

> TestEmployee类：

```
public class Demo4 {
```

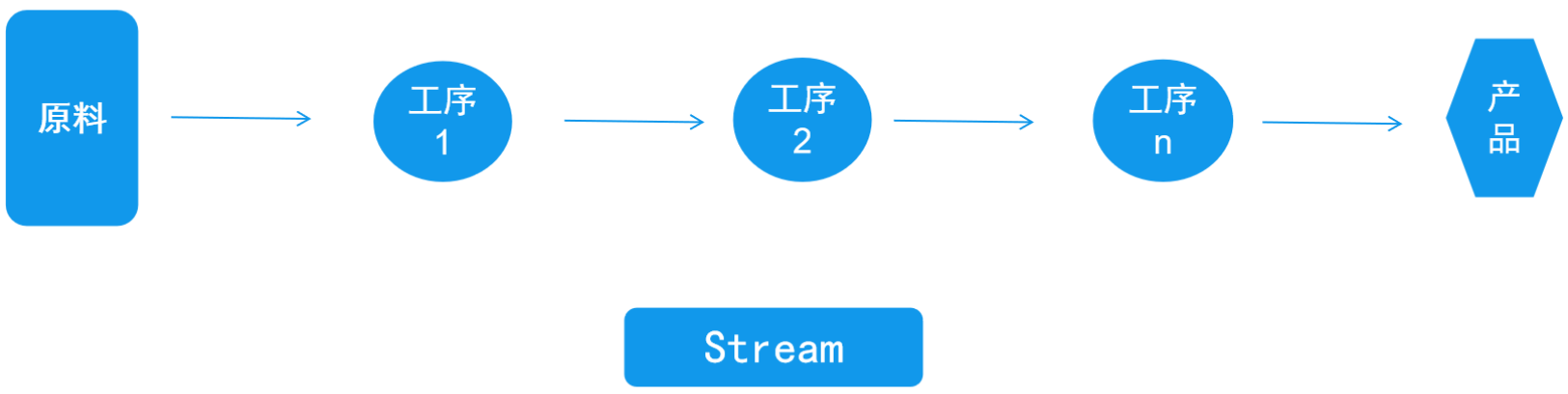
```
    public static void main(String[] args) {
      //1 对象::实例方法
      Consumer<String> consumer=s->System.out.println(s);
      consumer.accept("hello");
      Consumer<String> consumer2=System.out::println;
      consumer.accept("world");

      //2类::静态方法
      Comparator<Integer> com=(o1,o2)->Integer.compare(o1, o2);
      Comparator<Integer> com2=Integer::compare;

      //3类::实例方法
      Function<Employee, String> function=e->e.getName();
      Function<Employee, String> function2=Employee::getName;

      System.out.println(function2.apply(new Employee("小明", 50000)));

      //4类::new
      Supplier<Employee> supplier=()->new Employee();
      Supplier<Employee> supplier2=Employee::new;

      Employee employee=supplier.get();
      System.out.println(employee.toString());

    }
}
```

五、什么是Stream【 重点 】

5.1 概念

> 流（Stream）与集合类似，但集合中保存的是数据，而Stream中保存对集合或数组数据的操作。

Stream



5.2 Stream特点

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

5.3 Stream使用步骤

> 创建：
>
> - 新建一个流。
>
> 中间操作：
>
> - 在一个或多个步骤中，将初始Stream转化到另一个Stream的中间操作。
>
> 终止操作：
>
> - 使用一个终止操作来产生一个结果。该操作会强制之前的延迟操作立即执行，在此之后，该Stream就不能使用了。

5.4 创建Stream

> - 通过Collection对象的stream()或parallelStream()方法。
> - 通过Arrays类的stream()方法。
> - 通过Stream接口的of()、iterate()、generate()方法。
> - 通过IntStream、LongStream、DoubleStream接口中的of、range、rangeClosed方法。

> 案例演示：

```
public class Demo5 {
  public static void main(String[] args) {
    //(1)Collection对象中的stream()和parallelStream()方法
    ArrayList<String> arrayList=new ArrayList<>();
```

```
        arrayList.add("apple");
        arrayList.add("huawei");
        arrayList.add("xiaomi");
        Stream<String> stream = arrayList.parallelStream();
        //遍历
//        stream.forEach(s->System.out.println(s));
        stream.forEach(System.out::println);
        //(2)Arrays工具类的stream方法
        String[] arr= {"aaa","bbb","ccc"};
        Stream<String> stream2=Arrays.stream(arr);
        stream2.forEach(System.out::println);

        //(3)Stream接口中的of iterate 、generate 方法

        Stream<Integer> stream3 = Stream.of(10,20,30,40,50);
        stream3.forEach(System.out::println);
        //迭代流
        System.out.println("-----迭代流------");
        Stream<Integer> iterate = Stream.iterate(0, x->x+2);
        iterate.limit(5).forEach(System.out::println);
        System.out.println("--------生成流----------");
        //生成流
        Stream<Integer> generate = Stream.generate(()->new Random().nextInt(100));
        generate.limit(10).forEach(System.out::println);

        //(4)IntStream,LongStream,DoubleStream  的of  、range、rangeClosed
        IntStream stream4 = IntStream.of(100,200,300);
        stream4.forEach(System.out::println);
        IntStream range = IntStream.rangeClosed(0, 50);
        range.forEach(System.out::println);
    }
}
```

5.5 中间操作

> 常见中间操作：
>
> - filter、limit、skip、distinct、sorted
> - map
> - parallel

> 案例演示：

```
public class Demo6 {
    public static void main(String[] args) {
        ArrayList<Employee> list=new ArrayList<>();
        list.add(new Employee("小王", 15000));
        list.add(new Employee("小张", 12000));
        list.add(new Employee("小李", 18000));
        list.add(new Employee("小孙", 20000));
        list.add(new Employee("小刘", 25000));
        //list.add(new Employee("小刘", 25000));
        //中间操作1
            //filter过滤、limit 限制、skip 跳过、distinct 去掉重复、sorted排序
        //(1) filter过滤
        System.out.println("------filter-------");
        list.stream()
          .filter(e->e.getMoney()>15000)
          .forEach(System.out::println);
        //(2) limit限制
        System.out.println("----limit------");
        list.stream()
          .limit(2)
          .forEach(System.out::println);
        //(3) skip跳过
        System.out.println("-----skip------");
        list.stream()
          .skip(2)
          .forEach(System.out::println);
        System.out.println("------distinct--------");
        //(4) distinct去重复
        list.stream()
          .distinct()
          .forEach(System.out::println);

        System.out.println("---------sorted---------");
        //(5) sorted排序
        list.stream()
          .sorted((e1,e2)->Double.compare(e1.getMoney(), e2.getMoney()))
          .forEach(System.out::println);

        //中间操作2 map
        System.out.println("---------map--------");
        list.stream()
          .map(e->e.getName())
          .forEach(System.out::println);
        //中间操作3 parallel 采用多线程 效率高
```

```
    System.out.println("---------map--------");
    list.parallelStream()
      .forEach(System.out::println);
  }
}
```

> 串行流和并行流：
>
> - 串行流使用单线程。
> - 并行流使用多线程，效率更高。

```
public class Demo7 {
  public static void main(String[] args) {
    //串行流和并行流的区别
    ArrayList<String> list=new ArrayList<>();
    for(int i=0;i<5000000;i++) {
      list.add(UUID.randomUUID().toString());
    }
    //串行：10秒  并行：7秒
    long start=System.currentTimeMillis();
        long count=list.Stream().sorted().count();
    //long count=list.parallelStream().sorted().count();
    System.out.println(count);
    long end=System.currentTimeMillis();
    System.out.println("用时:"+(end-start));
  }
}
```

5.6 终止操作

> 常见终止操作：
>
> - forEach、min、max、count
> - reduce、collect

> 案例演示：

```
public class Demo8 {
  public static void main(String[] args) {
    ArrayList<Employee> list = new ArrayList<>();
    list.add(new Employee("小王", 15000));
    list.add(new Employee("小张", 12000));
    list.add(new Employee("小李", 18000));
    list.add(new Employee("小孙", 20000));
    list.add(new Employee("小刘", 25000));
    //1 终止操作 foreach
    list.stream()
      .filter(e->{
          System.out.println("过滤了....");
          return e.getMoney()>15000;
       })
      .forEach(System.out::println);
    //2 终止操作 min max count
    System.out.println("-----min-----");
    Optional<Employee> min = list.stream()
      .min((e1,e2)->Double.compare(e1.getMoney(), e2.getMoney()));
    System.out.println(min.get());
    System.out.println("-----max-----");
    Optional<Employee> max = list.stream()
      .max((e1,e2)->Double.compare(e1.getMoney(), e2.getMoney()));
    System.out.println(max.get());

    long count = list.stream().count();
    System.out.println("员工个数:"+count);

    //3 终止操作 reduce 规约
    //计算所有员工的工资和
    System.out.println("--------reduce---------");
    Optional<Double> sum = list.stream()
      .map(e->e.getMoney())
      .reduce((x,y)->x+y);
    System.out.println(sum.get());

    //4 终止方法 collect收集
    //获取所有的员工姓名，封装成一个list集合
    System.out.println("------collect------");
    List<String> names = list.stream()
      .map(e->e.getName())
      .collect(Collectors.toList());
    for (String string : names) {
      System.out.println(string);
    }
  }
}
```

六、新时间API

6.1 概述

> 之前时间API存在问题：线程安全问题、设计混乱。
>
> 本地化日期时间 API：
>
> - LocalDate
> - LocalTime
> - LocalDateTime
>
> Instant：时间戳。
>
> ZoneId：时区。
>
> Date、Instant、LocalDateTime的转换。
>
> DateTimeFormatter：格式化类。

6.2 LocalDateTime类

> 表示本地日期时间，没有时区信息

```
public class Demo2 {
  public static void main(String[] args) {
    //1创建本地时间
    LocalDateTime localDateTime=LocalDateTime.now();
    //LocalDateTime localDateTime2=LocalDateTime.of(year, month, dayOfMonth, hour, minute)
    System.out.println(localDateTime);
    System.out.println(localDateTime.getYear());
    System.out.println(localDateTime.getMonthValue());
    System.out.println(localDateTime.getDayOfMonth());

    //2添加两天
    LocalDateTime localDateTime2 = localDateTime.plusDays(2);
    System.out.println(localDateTime2);

    //3减少一个月
    LocalDateTime localDateTime3 = localDateTime.minusMonths(1);
    System.out.println(localDateTime3);
  }
}
```

6.3 Instant、ZoneId类

> Instant表示瞬间；和前面Date类似。
>
> ZoneId表示时区信息。

```
public class Demo3 {
  public static void main(String[] args) {
    //1 创建Instant：时间戳
    Instant instant=Instant.now();
    System.out.println(instant.toString());
    System.out.println(instant.toEpochMilli());
    System.out.println(System.currentTimeMillis());
    //2 添加减少时间

    Instant instant2 = instant.plusSeconds(10);

    System.out.println(Duration.between(instant, instant2).toMillis());

    //3ZoneId
    Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();
    for (String string : availableZoneIds) {
      System.out.println(string);
    }

    System.out.println(ZoneId.systemDefault().toString());

    //1 Date --->Instant--->LocalDateTime
    System.out.println("-------------Date --->Instant---->LocalDateTime-----------");
    Date date=new Date();
    Instant instant3 = date.toInstant();
    System.out.println(instant3);

    LocalDateTime localDateTime = LocalDateTime.ofInstant(instant3, ZoneId.systemDefault());
    System.out.println(localDateTime);

    //2 LocalDateTime --->Instant--->Date
    System.out.println("-------------LocalDateTime --->Instant---->Date-----------");

    Instant instant4 = localDateTime.atZone(ZoneId.systemDefault()).toInstant();
```

```
        System.out.println(instant4);
        Date from = Date.from(instant4);
        System.out.println(from);

    }
}
```

6.4 DateTimeFormatter类

> DateTimeFormatter是时间格式化类。

```
public class Demo4 {
    public static void main(String[] args) {
        //创建DateTimeFormatter
        DateTimeFormatter dtf=DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        //1 把时间格式化成字符串
        String format = dtf.format(LocalDateTime.now());
        System.out.println(format);
        //2 把字符串解析成时间
        LocalDateTime localDateTime = LocalDateTime.parse("2020/03/10 10:20:35", dtf);
        System.out.println(localDateTime);
    }
}
```