Java8 新特性

> Author：King
>
> Version：9.0.2

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表达式不是Java最早使用的，很多语言就支持Lambda表达式，例如：C++，C#，Python，Scala等。如果有Python或者Javascript的语言基础，对理解Lambda表达式有很大帮助，可以这么说lambda表达式其实就是实现SAM接口的语法糖，使得Java也算是支持函数式编程的语言。Lambda**写的好**可以极大的减少代码冗余，同时可读性也好过冗长的匿名内部类。
- 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。
- Lamdba大大减少代码量，同时也带来了可读性差的问题！
- Lamdba的本质就是一种匿名内部类实例的简化语法

2.2 lambda语法

Lambda表达式语法格式：

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

说明：

- Lambda要求简化的接口只能有一个抽象方法，推荐使用@FunctionalInterface约束
- (形参列表)它就是你要赋值的函数式接口的抽象方法的(形参列表)，照抄
- {Lambda体}就是实现这个抽象方法的方法体
- ->称为Lambda操作符（减号和大于号中间不能有空格，而且必须是英文状态下半角输入方式）

优化：Lambda表达式可以精简

- 当{Lambda体}中只有一句语句时，可以省略{}和{;}

  ```
  () -> 方法体;
  ```

- 当{Lambda体}中只有一句语句时，并且这个语句还是一个return语句，那么return也可以省略，但是如果{;}没有省略的话，return是不能省略的

  ```
  () -> {return 方法体;}

  () -> 方法体;
  ```

- (形参列表)的类型可以省略

  ```
  (int x,int y) -> {return 方法体;}
  (x,y) -> {return 方法体;}
  ```

- 当(形参列表)的形参个数只有一个，那么可以把数据类型和()一起省略，但是形参名不能省略

```
(int x)-> {return 方法体；}
x-> {return 方法体；}
```

- 当(形参列表)是空参时，()不能省略

2.3 Lambda使用

2.3.1 基本Lambda

> 无参数，多行方法体接口

1. 声明接口

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * description:数据解析接口
 * @FunctionalInterface 限制当前方法只能有一个接口
 */
@FunctionalInterface
public interface DataParser {

    /**
     * 处理数据方法
     */
    void  deal();

}
```

2. 简化语法

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * description:
 */
public class UseLambda {

    public static void main(String[] args) {

        /**
         * Java默认语法
         */
        DataParser dataParser = new DataParser() {
            @Override
            public void deal() {
                System.out.println("111");
                System.out.println("222");
            }
        };

        dataParser.deal();
```

```
        /**
         * lambda简化
         * 要求1：  确保接口只有一个方法，方可使用lambda!
         *          可以通过@FunctionalInterface限制接口只有一个方法
         *
         *          思考：为什么必须只能有个抽象方法呢？
         *
         * 简化语法：  () [参数] -> { 方法体;}
         */

        DataParser dataParser1 = () ->{
            System.out.println("111");
            System.out.println("222");
        };

        dataParser1.deal();
    }
}
```

2.3.2 Lambda优化

> 优化型参数列表和返回方法题

1. 接口声明

```
public interface DataParser1 {

    String deal(String k,String p);

}

public interface DataParser2 {

    void deal(String k);

}

public interface DataParser3 {

    String deal(String k);

}

public interface DataParser4 {

    String deal();

}
```

2. lambda使用

```
package com.atguigu.lambda;

import com.atguigu.interfaces.DataParser1;
```

```java
import com.atguigu.interfaces.DataParser2;
import com.atguigu.interfaces.DataParser3;
import com.atguigu.interfaces.DataParser4;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * description:
 */
public class UseLambda1 {

    public static void main(String[] args) {

        /**
         * 场景1：多参数，有返回值
         * java方式
         */

        DataParser1 dataParser1 = new DataParser1() {
            @Override
            public String deal(String k, String p) {
                //方法体
                String ret = k + p;
                System.out.println("ret = " + ret);
                return ret;
            }
        };

        /**
         * lambda使用1: 正常简化
         */

        DataParser1 dataParser11 = (String k,String p) -> {
            String ret = k + p;
            System.out.println("ret = " + ret);
            return ret;
        };

        /**
         * lambda使用1: 简化形参列表！但是因为多行，并且有返回值 {}不能省略！
         */

        DataParser1 dataParser12 = (k,p)->{
            String ret = k + p;
            System.out.println("ret = " + ret);
            return ret;
        };


        //-----------
        //场景2：  单参数，方法体  无返回值
        //-----------
        DataParser2 dataParser2 = k -> System.out.println("heihei");


        //-----------
        //场景3：  单参数，方法体  有返回值
```

```
        //-----------
        DataParser3 dataParser3 = k -> k+"test";


        //-----------
        //场景3： 无参数，方法体  有返回值
        //-----------
        DataParser4 dataParser4 = () -> "test" ;


    }
}
```

2.3.3 利用lambda实现线程

> 当需要启动一个线程去完成任务时，通常会通过`java.lang.Runnable`接口来定义任务内容，并使用`java.lang.Thread`类来启动该线程。代码如下：

```
public class Demo01Runnable {
    public static void main(String[] args) {
        // 匿名内部类
        Runnable task = new Runnable() {
            @Override
            public void run() { // 覆盖重写抽象方法
                System.out.println("多线程任务执行！");
            }
        };
        new Thread(task).start(); // 启动线程
    }
}
```

本着“一切皆对象”的思想，这种做法是无可厚非的：首先创建一个`Runnable`接口的匿名内部类对象来指定任务内容，再将其交给一个线程来启动。

> lambda优化思维

```
public class Demo02LambdaRunnable {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("多线程任务执行！")).start(); // 启动线程
    }
}
```

2.3.4 lambda作为函数参数实践

> 例如：声明一个计算器`Calculator`接口，内含抽象方法`calc`可以对两个int数字进行计算，并返回结果：

```
public interface Calculator {
    int calc(int a, int b);
}
```

在测试类中，声明一个如下方法：

```
public static void invokeCalc(int a, int b, Calculator calculator) {
        int result = calculator.calc(a, b);
        System.out.println("结果是: " + result);
}
```

测试函数运算：

```
public static void main(String[] args) {
        invokeCalc(1, 2, (int a,int b)-> {return a+b;});
        invokeCalc(1, 2, (int a,int b)-> a-b);
        invokeCalc(1, 2, (int a,int b)-> {return a*b;});
        invokeCalc(1, 2, (int a,int b)-> {return a/b;});
        invokeCalc(1, 2, (int a,int b)-> {return a%b;});
        invokeCalc(1, 2, (int a,int b)-> {return a>b?a:b;});
}
```

2.4 Jdk提供lambda接口

> jdk在1.8更新了一部分lambda接口，一般作用在集合数据处理！
>
> 集合新增接口参数方法，配合新增加接口快速完成数据处理！

2.4.1 消费型接口

消费型接口的抽象方法特点：有形参，但是返回值类型是void

接口名	抽象方法	描述
Consumer	void accept(T t)	接收一个对象用于完成功能
BiConsumer<T,U>	void accept(T t, U u)	接收两个对象用于完成功能
DoubleConsumer	void accept(double value)	接收一个double值
IntConsumer	void accept(int value)	接收一个int值
LongConsumer	void accept(long value)	接收一个long值
ObjDoubleConsumer	void accept(T t, double value)	接收一个对象和一个double值
ObjIntConsumer	void accept(T t, int value)	接收一个对象和一个int值
ObjLongConsumer	void accept(T t, long value)	接收一个对象和一个long值

2.4.2 供给型接口

这类接口的抽象方法特点：无参，但是有返回值

接口名	抽象方法	描述
Supplier	T get()	返回一个对象
BooleanSupplier	boolean getAsBoolean()	返回一个boolean值
DoubleSupplier	double getAsDouble()	返回一个double值
IntSupplier	int getAsInt()	返回一个int值
LongSupplier	long getAsLong()	返回一个long值

2.4.3 判断型接口

这里接口的抽象方法特点：有参，但是返回值类型是boolean结果。

接口名	抽象方法	描述
Predicate	boolean test(T t)	接收一个对象
BiPredicate<T,U>	boolean test(T t, U u)	接收两个对象
DoublePredicate	boolean test(double value)	接收一个double值
IntPredicate	boolean test(int value)	接收一个int值
LongPredicate	boolean test(long value)	接收一个long值

2.4.4 功能型接口

这类接口的抽象方法特点：既有参数又有返回值

接口名	抽象方法	描述
Function<T,R>	R apply(T t)	接收一个T类型对象，返回一个R类型对象结果
UnaryOperator	T apply(T t)	接收一个T类型对象，返回一个T类型对象结果
DoubleFunction	R apply(double value)	接收一个double值，返回一个R类型对象
IntFunction	R apply(int value)	接收一个int值，返回一个R类型对象
LongFunction	R apply(long value)	接收一个long值，返回一个R类型对象
ToDoubleFunction	double applyAsDouble(T value)	接收一个T类型对象，返回一个double
ToIntFunction	int applyAsInt(T value)	接收一个T类型对象，返回一个int
ToLongFunction	long applyAsLong(T value)	接收一个T类型对象，返回一个long
DoubleToIntFunction	int applyAsInt(double value)	接收一个double值，返回一个int结果
DoubleToLongFunction	long applyAsLong(double value)	接收一个double值，返回一个long结果
IntToDoubleFunction	double applyAsDouble(int value)	接收一个int值，返回一个double结果
IntToLongFunction	long applyAsLong(int value)	接收一个int值，返回一个long结果
LongToDoubleFunction	double applyAsDouble(long value)	接收一个long值，返回一个double结果
LongToIntFunction	int applyAsInt(long value)	接收一个long值，返回一个int结果
DoubleUnaryOperator	double applyAsDouble(double operand)	接收一个double值，返回一个double
IntUnaryOperator	int applyAsInt(int operand)	接收一个int值，返回一个int结果
LongUnaryOperator	long applyAsLong(long operand)	接收一个long值，返回一个long结果
BiFunction<T,U,R>	R apply(T t, U u)	接收一个T类型和一个U类型对象，返回一个R类型对象结果

接口名	抽象方法	描述
BinaryOperator	T apply(T t, T u)	接收两个T类型对象，返回一个T类型对象结果
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)	接收一个T类型和一个U类型对象，返回一个double
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)	接收一个T类型和一个U类型对象，返回一个int
ToLongBiFunction<T,U>	long applyAsLong(T t, U u)	接收一个T类型和一个U类型对象，返回一个long
DoubleBinaryOperator	double applyAsDouble(double left, double right)	接收两个double值，返回一个double结果
IntBinaryOperator	int applyAsInt(int left, int right)	接收两个int值，返回一个int结果
LongBinaryOperator	long applyAsLong(long left, long right)	接收两个long值，返回一个long结果

2.5 Lambda标准练习

2.5.1 无参无返回值形式

> 假如有自定义函数式接口Call如下：

```
public interface Call {
    void shout();
}
```

> 在测试类中声明一个如下方法：

```
public static void callSomething(Call call){
    call.shout();
  }
```

> 在测试类的main方法中调用callSomething方法，并用Lambda表达式为形参call赋值，可以喊出任意你想说的话。

```
public class TestLambda {
  public static void main(String[] args) {
    callSomething(()->System.out.println("回家吃饭"));
    callSomething(()->System.out.println("我爱你"));
    callSomething(()->System.out.println("滚蛋"));
    callSomething(()->System.out.println("回来"));
  }
  public static void callSomething(Call call){
    call.shout();
  }
}
interface Call {
    void shout();
```

```
}
```

2.5.2 消费型接口

代码示例：Consumer接口

在JDK1.8中Collection集合接口的父接口Iterable接口中增加了一个默认方法：

`public default void forEach(Consumer<? super T> action)` 遍历Collection集合的每个元素，执行“xxx消费型”操作。

在JDK1.8中Map集合接口中增加了一个默认方法：

`public default void forEach(BiConsumer<? super K,? super V> action)` 遍历Map集合的每对映射关系，执行“xxx消费型”操作。

案例：

（1）创建一个Collection系列的集合，添加你知道的编程语言，调用forEach方法遍历查看

（2）创建一个Map系列的集合，添加一些(key,value)键值对，例如，添加编程语言排名和语言名称，调用forEach方法遍历查看

Jun 2019	Jun 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.004%	-0.36%
2	2		C	13.300%	-1.64%
3	4	^	Python	8.530%	+2.77%
4	3	v	C++	7.384%	-0.95%
5	6	^	Visual Basic .NET	4.624%	+0.86%
6	5	v	C#	4.483%	+0.17%

示例代码：

```
@Test
public void test1(){
    List<String> list = Arrays.asList("java","c","python","c++","VB","C#");
    list.forEach(s -> System.out.println(s));
    }
  @Test
  public void test2(){
    HashMap<Integer,String> map = new HashMap<>();
    map.put(1, "java");
    map.put(2, "c");
    map.put(3, "python");
    map.put(4, "c++");
        map.put(5, "VB");
        map.put(6, "C#");
    map.forEach((k,v) -> System.out.println(k+"->"+v));
  }
```

2.5.3 供给型接口

代码示例：Supplier接口

在JDK1.8中增加了StreamAPI，java.util.stream.Stream是一个数据流。这个类型有一个静态方法：

`public static <T> Stream<T> generate(Supplier<T> s)`可以创建Stream的对象。而又包含一个forEach方法可以遍历流中的元素：`public void forEach(Consumer<? super T> action)`。

案例：

现在请调用Stream的generate方法，来产生一个流对象，并调用Math.random()方法来产生数据，为Supplier函数式接口的形参赋值。最后调用forEach方法遍历流中的数据查看结果。

```
@Test
public void test2(){
    Stream.generate(() -> Math.random()).forEach(num ->
System.out.println(num));
}
```

2.5.4 功能型接口

代码示例：Funtion<T,R>接口

在JDK1.8时Map接口增加了很多方法，例如：

`public default void replaceAll(BiFunction<? super K,? super V,? extends V> function)`按照function指定的操作替换map中的value。

`public default void forEach(BiConsumer<? super K,? super V> action)`遍历Map集合的每对映射关系，执行“xxx消费型”操作。

案例：

（1）声明一个Employee员工类型，包含编号、姓名、薪资。

（2）添加n个员工对象到一个HashMap<Integer,Employee>集合中，其中员工编号为key，员工对象为value。

（3）调用Map的forEach遍历集合

（4）调用Map的replaceAll方法，将其中薪资低于10000元的，薪资设置为10000。

（5）再次调用Map的forEach遍历集合查看结果

Employee类：

```
class Employee{
  private int id;
  private String name;
  private double salary;
  public Employee(int id, String name, double salary) {
    super();
    this.id = id;
    this.name = name;
    this.salary = salary;
  }
  public Employee() {
    super();
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
```

```
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public double getSalary() {
    return salary;
  }
  public void setSalary(double salary) {
    this.salary = salary;
  }
  @Override
  public String toString() {
    return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";
  }

}
```

测试类:

```
import java.util.HashMap;

public class TestLambda {
  public static void main(String[] args) {
    HashMap<Integer,Employee> map = new HashMap<>();
    Employee e1 = new Employee(1, "张三", 8000);
    Employee e2 = new Employee(2, "李四", 9000);
    Employee e3 = new Employee(3, "王五", 10000);
    Employee e4 = new Employee(4, "赵六", 11000);
    Employee e5 = new Employee(5, "钱七", 12000);

    map.put(e1.getId(), e1);
    map.put(e2.getId(), e2);
    map.put(e3.getId(), e3);
    map.put(e4.getId(), e4);
    map.put(e5.getId(), e5);

    map.forEach((k,v) -> System.out.println(k+"="+v));
    System.out.println();

    map.replaceAll((k,v)->{
      if(v.getSalary()<10000){
        v.setSalary(10000);
      }
      return v;
    });
    map.forEach((k,v) -> System.out.println(k+"="+v));
  }
}
```

2.5.5 判断型接口

代码示例：Predicate接口

JDK1.8时，Collecton接口增加了一下方法，其中一个如下：

`public default boolean removeIf(Predicate<? super E> filter)` 用于删除集合中满足filter指定的条件判断的。

`public default void forEach(Consumer<? super T> action)` 遍历Collection集合的每个元素，执行“xxx消费型”操作。

案例：

（1）添加一些字符串到一个Collection集合中

（2）调用forEach遍历集合

（3）调用removeIf方法，删除其中字符串的长度<5的

（4）再次调用forEach遍历集合

```
import java.util.ArrayList;

public class TestLambda {
  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("hello");
    list.add("java");
    list.add("atguigu");
    list.add("ok");
    list.add("yes");

    list.forEach(str->System.out.println(str));
    System.out.println();

    list.removeIf(str->str.length()<5);
    list.forEach(str->System.out.println(str));
  }
}
```

2.5.6 判断型接口

案例：

（1）声明一个Employee员工类型，包含编号、姓名、性别，年龄，薪资。

（2）声明一个EmployeeSerice员工管理类，包含一个ArrayList集合的属性all，在EmployeeSerice的构造器中，创建一些员工对象，为all集合初始化。

（3）在EmployeeSerice员工管理类中，声明一个方法：ArrayList get(Predicate p)，即将满足p指定的条件的员工，添加到一个新的ArrayList 集合中返回。

（4）在测试类中创建EmployeeSerice员工管理类的对象，并调用get方法，分别获取：

- 所有员工对象
- 所有年龄超过35的员工
- 所有薪资高于15000的女员工
- 所有编号是偶数的员工

- 名字是“张三”的员工
- 年龄超过25，薪资低于10000的男员工

示例代码：

Employee类：

```
public class Employee{
  private int id;
  private String name;
  private char gender;
  private int age;
  private double salary;

  public Employee(int id, String name, char gender, int age, double salary) {
    super();
    this.id = id;
    this.name = name;
    this.gender = gender;
    this.age = age;
    this.salary = salary;
  }
  public Employee() {
    super();
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public double getSalary() {
    return salary;
  }
  public void setSalary(double salary) {
    this.salary = salary;
  }
  @Override
  public String toString() {
    return "Employee [id=" + id + ", name=" + name + ", gender=" + gender + ",
age=" + age + ", salary=" + salary
        + "]";
  }
}
```

员工管理类：

```
class EmployeeService{
  private ArrayList<Employee> all;
  public EmployeeService(){
    all = new ArrayList<Employee>();
```

```
    all.add(new Employee(1, "张三", '男', 33, 8000));
    all.add(new Employee(2, "翠花", '女', 23, 18000));
    all.add(new Employee(3, "无能", '男', 46, 8000));
    all.add(new Employee(4, "李四", '女', 23, 9000));
    all.add(new Employee(5, "老王", '男', 23, 15000));
    all.add(new Employee(6, "大嘴", '男', 23, 11000));
  }
  public ArrayList<Employee> get(Predicate<Employee> p){
    ArrayList<Employee> result = new ArrayList<Employee>();
    for (Employee emp : all) {
      if(p.test(emp)){
        result.add(emp);
      }
    }
    return result;
  }
}
```

测试类：

```
public class TestLambda {
  public static void main(String[] args) {
    EmployeeService es = new EmployeeService();

    es.get(e -> true).forEach(e->System.out.println(e));
    System.out.println();
    es.get(e -> e.getAge()>35).forEach(e->System.out.println(e));
    System.out.println();
    es.get(e -> e.getSalary()>15000 && e.getGender()=='女').forEach(e-
>System.out.println(e));
    System.out.println();
    es.get(e -> e.getId()%2==0).forEach(e->System.out.println(e));
    System.out.println();
    es.get(e -> "张三".equals(e.getName())).forEach(e->System.out.println(e));
    System.out.println();
    es.get(e -> e.getAge()>25 && e.getSalary()<10000 &&
e.getGender()=='男').forEach(e->System.out.println(e));
  }
}
```

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:

```
package com.atguigu.bean;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/7 23:25
 * description:
 */
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:

```
package com.atguigu.method;

import com.atguigu.bean.Employee;

import java.util.Comparator;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * projectName: demo
 *
 * @author: 赵伟风
 * description:
 */
public class TestEmp {

    public static void main(String[] args) {

        //todo:简化实例方法
        Consumer<String> consumer = s -> System.out.println(s);

        consumer.accept("test1");

        //简化！省略参数，直接简化输出
        Consumer<String> consumer1 = System.out::println;
        consumer.accept("哈哈");


        //todo 简化静态方法
        Comparator<Integer> comparator1 = new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
```

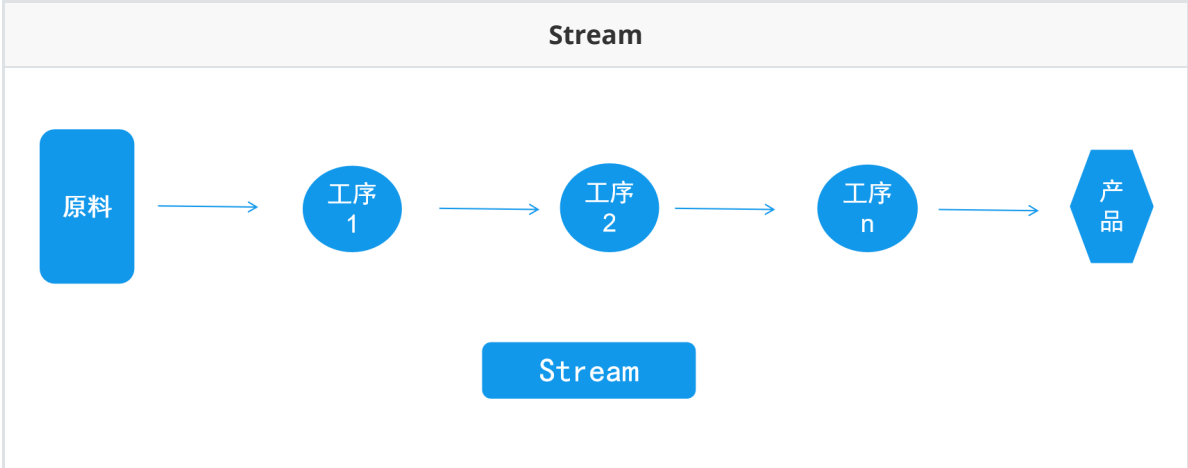
```
                return Integer.compare(o1, o2);
            }
        };

        Comparator<Integer> comparator = (o1,o2) -> Integer.compare(o1, o2);

        int compare = comparator.compare(2, 1);
        System.out.println("compare = " + compare);

        //方法引用
        Comparator<Integer> comparator2 = Integer::compareTo;
        int compare1 = comparator2.compare(2, 1);
        System.out.println("compare1 = " + compare1);

        //todo 简化实例方法
        Function<Employee,String> function = new Function<Employee, String>() {
            @Override
            public String apply(Employee employee) {

                return employee.getName();
            }
        };


        String ret = function.apply(new Employee("哈哈", 100));
        System.out.println("ret = " + ret);

        //lambda
        Function<Employee,String> function1 = employee -> employee.getName();

        //lambda + 方法引用
        Function<Employee,String> function2 = Employee::getName;

        System.out.println(function2.apply(new Employee("呵呵", 18)));

        //TODO new简化
        //基本写法
        Supplier<Employee> supplier = new Supplier<Employee>() {
            @Override
            public Employee get() {

                return new Employee();
            }
        };

        //lambda
        Supplier<Employee> supplier1 = ()-> new Employee();

        //lambda+方法引用
        Supplier<Employee> supplier2 = Employee::new;



    }
```

```
}
```

五、什么是Stream【重点】

5.1 概念

> 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。
>
> 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。
>
> Stream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，负责存储数据，Stream流讲的是计算，负责处理数据！”

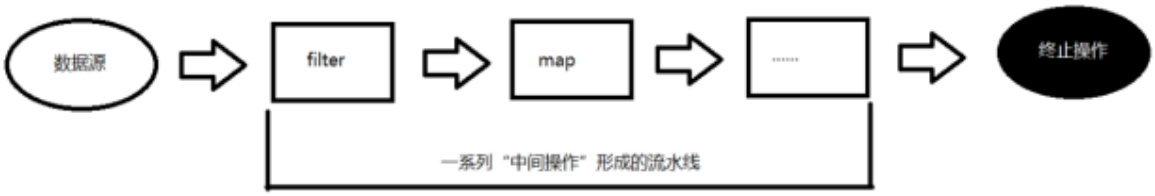


5.2 Stream特点

- Stream 自己不会存储元素。
- Stream 不会改变源对象。每次处理都会返回一个持有结果的新Stream。
- Stream 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。

5.3 Stream使用步骤

> Stream 的操作三个步骤：
>
> 1. 创建 Stream：通过一个数据源（如：集合、数组），获取一个流
> 2. 中间操作：中间操作是个操作链，对数据源的数据进行n次处理，但是在终结操作前，并不会真正执行。
> 3. 终止操作：一旦执行终止操作，就执行中间操作链，最终产生结果并结束Stream。



5.4 创建Stream

1. 创建方式一：基于集合

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

   - public default Stream stream() : 返回一个顺序流
   - public default Stream parallelStream() : 返回一个并行流

2. 创建方式二：基于数组

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

   - public static Stream stream(T[] array): 返回一个流

3. 创建方式三：基于Stream.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 Stream of(T... values) : 返回一个顺序流

> 代码演示：

```
package com.atguigu.stream;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

/**
 * projectName: demos
 * @author: 赵伟风
 * description:创建stream
 */
public class CreateStream {

    public static void main(String[] args) {
        //1.基于可变参数
        Stream<Integer> integerStream = Stream.of(1, 2, 2, 3, 4, 5);
        //2.基于数组创建
        String [] names = {"小井老师","小利亚老师","小魔仙老师"};
        Stream<String> stream = Arrays.stream(names);
        //3.基于集合创建
        List<String> arrs = new ArrayList<>();
        //单线程 串行执行
        Stream<String> stream1 = arrs.stream();
        //多线程 并发执行
        Stream<String> stringStream = arrs.parallelStream();
    }
}
```

> 性能对比

```java
package com.atguigu.stream;

import java.util.ArrayList;
import java.util.UUID;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/8 0:04
 * description:
 */
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));
    }
}
```

5.5 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”。

方法	描述
filter(Predicate p)	接收 Lambda ， 从流中排除某些元素，保留符合条件的元素
distinct()	筛选，通过流所生成元素的equals() 去除重复元素
limit(long maxSize)	截断流，使其元素不超过给定数量
skip(long n)	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 limit(n) 互补
peek(Consumer action)	接收Lambda，对流中的每个数据执行Lambda体操作
sorted()	产生一个新流，其中按自然顺序排序
sorted(Comparator com)	产生一个新流，其中按比较器顺序排序
map(Function f)	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。
mapToDouble(ToDoubleFunction f)	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。
mapToInt(ToIntFunction f)	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。
mapToLong(ToLongFunction f)	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。

> 案例演示：演示中间

```
package com.atguigu.stream;

import org.junit.Test;

import java.util.Arrays;
import java.util.stream.Stream;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * description:
 */
public class Test08StreamMiddle {

    @Test
    public void test11() {
        String[] arr = {"hello", "world", "java"};

        Arrays.stream(arr)
                .map(t -> t.toUpperCase())
                .forEach(System.out::println);
    }

    @Test
    public void test10() {
```

```
        Stream.of(1, 2, 3, 4, 5)
                .map(t -> t += 1)//Function<T,R>接口抽象方法 R apply(T t)
                .forEach(System.out::println);
    }

    @Test
    public void test09() {
        //希望能够找出前三个最大值，前三名最大的，不重复
        Stream.of(11, 2, 39, 4, 54, 6, 2, 22, 3, 3, 4, 54, 54)
                .distinct()
                .sorted((t1, t2) -> -Integer.compare(t1, t2))//Comparator接口
int compare(T t1, T t2)
                .limit(3)
                .forEach(System.out::println);
    }

    @Test
    public void test08() {
        long count = Stream.of(1, 2, 3, 4, 5, 6, 2, 2, 3, 3, 4, 4, 5)
                .distinct()
                .peek(System.out::println)  //Consumer接口的抽象方法  void accept(T
t)
                .count();
        System.out.println("count=" + count);
    }


    @Test
    public void test07() {
        Stream.of(1, 2, 3, 4, 5, 6, 2, 2, 3, 3, 4, 4, 5)
                .skip(5)
                .distinct()
                .filter(t -> t % 3 == 0)
                .forEach(System.out::println);
    }

    @Test
    public void test06() {
        Stream.of(1, 2, 3, 4, 5, 6, 2, 2, 3, 3, 4, 4, 5)
                .skip(5)
                .forEach(System.out::println);
    }

    @Test
    public void test05() {
        Stream.of(1, 2, 2, 3, 3, 4, 4, 5, 2, 3, 4, 5, 6, 7)
                .distinct()  //(1,2,3,4,5,6,7)
                .filter(t -> t % 2 != 0) //(1,3,5,7)
                .limit(3)
                .forEach(System.out::println);
    }


    @Test
    public void test04() {
        Stream.of(1, 2, 3, 4, 5, 6, 2, 2, 3, 3, 4, 4, 5)
                .skip(2)
                .limit(3)
```

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


    @Test
    public void test03() {
        Stream.of(1, 2, 3, 4, 5, 6, 2, 2, 3, 3, 4, 4, 5)
                .distinct()
                .sorted(Integer::compareTo)
                .forEach(System.out::println);
    }


    @Test
    public void test02() {
        Stream.of(1, 2, 3, 4, 5, 6)
                .filter(t -> t % 2 == 0)
                .forEach(System.out::println);
    }

    @Test
    public void test01() {
        //1、创建Stream
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);

        //2、加工处理
        //过滤：filter(Predicate p)
        //把里面的偶数拿出来
        /*
         * filter(Predicate p)
         * Predicate是函数式接口，抽象方法：boolean test(T t)
         */
//        Stream stream1 = stream.filter(new Predicate<Integer>() {
//            @Override
//            public boolean test(Integer integer) {
//                return integer%2==0;
//            }
//        });
//
//        stream1.forEach(System.out::print);


        stream = stream.filter(t -> t % 2 == 0);

        //3、终结操作：例如：遍历
        stream.forEach(System.out::println);
    }
}
```

5.6 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。

方法	描述
boolean allMatch(Predicate p)	检查是否匹配所有元素
boolean anyMatch(Predicate p)	检查是否至少匹配一个元素
boolean noneMatch(Predicate p)	检查是否没有匹配所有元素
Optional findFirst()	返回第一个元素
Optional findAny()	返回当前流中的任意元素
long count()	返回流中元素总数
Optional max(Comparator c)	返回流中最大值
Optional min(Comparator c)	返回流中最小值
void forEach(Consumer c)	迭代
T reduce(T iden, BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 T
U reduce(BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 Optional
R collect(Collector c)	将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream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。

> 案例演示：

```
package com.atguigu.test06;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import org.junit.Test;

public class Test09StreamEnding {

```

```
    @Test
    public void test14(){
        List<Integer> list = Stream.of(1,2,4,5,7,8)
                .filter(t -> t%2==0)
                .collect(Collectors.toList());

        System.out.println(list);
    }


    @Test
    public void test13(){
        Optional<Integer> max = Stream.of(1,2,4,5,7,8)
             .reduce((t1,t2) -> t1>t2?t1:t2);//BinaryOperator接口   T apply(T t1,
T t2)
        System.out.println(max);
    }

    @Test
    public void test12(){
        Integer reduce = Stream.of(1,2,4,5,7,8)
             .reduce(0, (t1,t2) -> t1+t2);//BinaryOperator接口   T apply(T t1, T
t2)
        System.out.println(reduce);
    }

    @Test
    public void test11(){
        Optional<Integer> max = Stream.of(1,2,4,5,7,8)
                .max((t1,t2) -> Integer.compare(t1, t2));
        System.out.println(max);
    }

    @Test
    public void test10(){
        Optional<Integer> opt = Stream.of(1,2,4,5,7,8)
                .filter(t -> t%3==0)
                .findFirst();
        System.out.println(opt);
    }

    @Test
    public void test09(){
        Optional<Integer> opt = Stream.of(1,2,3,4,5,7,9)
                .filter(t -> t%3==0)
                .findFirst();
        System.out.println(opt);
    }

    @Test
    public void test08(){
        Optional<Integer> opt = Stream.of(1,3,5,7,9).findFirst();
        System.out.println(opt);
    }

    @Test
    public void test04(){
        boolean result = Stream.of(1,3,5,7,9)
```

```
            .anyMatch(t -> t%2==0);
        System.out.println(result);
    }


    @Test
    public void test03(){
        boolean result = Stream.of(1,3,5,7,9)
            .allMatch(t -> t%2!=0);
        System.out.println(result);
    }

    @Test
    public void test02(){
        long count = Stream.of(1,2,3,4,5)
                .count();
        System.out.println("count = " + count);
    }

    @Test
    public void test01(){
        Stream.of(1,2,3,4,5)
                .forEach(System.out::println);
    }
}
```

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实际上是个容器：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。

如何从Optional容器中取出所包装的对象呢?

（1）T get() ：要求Optional容器必须非空

T get()与of(T value)使用是安全的

（2）T orElse(T other) ：

orElse(T other) 与ofNullable(T value)配合使用，

如果Optional容器中非空，就返回所包装值，如果为空，就用orElse(T other)other指定的默认值（备胎）代替！！

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

> 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 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);
    }
}
```