JDK8新特性

1，Lambda表达式&函数式接口

1.1，什么是Lambda表达式?

lambda表达式可以理解为一种匿名函数的代替，lambda允许将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，目的是简化代码的编写。

1.2，什么是函数式接口?

lambda表达式需要函数式接口的支持

所谓函数式接口，是指只有一个抽象方法

另外JDK8也提供了一个注解，帮助我们编译时检查语法是否符合

@FunctionInterface

1.3，Lambda表达式使用案例

lambda表达式的基本语法：

```
函数式接口  变量名  = (参数1，参数2...)->{

          //方法体

};
```

案例1：

```
public static void main(String[] args){
    //匿名内部类
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            System.out.println("hello,lambda");
        }
    };
    new Thread(runnable).start();

    //lambda表达式
    Runnable runnable2 = ()->System.out.println("hello,lambda!");
    new Thread(runnable2).start();

    //lambda表达式
    new Thread(()->System.out.print("hello,lambda!!!")).start();
}
```

案例2：

```
//匿名内部类
Comparator<String> comparator = new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length()-o2.length();
    }
};
TreeSet<String> treeSet = new TreeSet<>(comparator);

//lambda表达式
TreeSet<String> treeSet2 = new TreeSet<>((o1,o2)->o1.length()-o2.length());
```

1.4，lambda表达式注意事项

```
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

1、形参列表的数据类型会自动推断;
2、如果形参列表为空，只需保留();
3、如果形参只有1个，()可以省略，只需要参数的名称即可;
4、如果执行语句只有1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1句;
5、lambda不会生成一个单独的内部类文件;
6、lambda表达式若访问了局部变量，则局部变量必须是final的，若是局部变量没有加final关键字，系统会自动添加，此 后在修改该局部变量，会报错。
```

2，流式编程-StreamAPI

2.1，什么是Stream?

Stream是Java8中处理数组、集合的抽象概念，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。

一个Stream表面上与一个集合很类似，集合中保存的是数据，而流设置的是对数据的操作。

Stream的特点：

```
1，Stream 自己不会存储元素。
2，Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
3，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。

Stream遵循“做什么，而不是怎么去做”的原则。只需要描述需要做什么，而不用考虑程序是怎样实现的。
```

2.2，快速体验StreamAPI的特点

```
public static void main(String[] args){
    List<String> data = new ArrayList<>();
```

```
        data.add("hello");
        data.add("stream");
        data.add("good night!");

        //传统做法
        long count = 0;
        for (String s : data) {
            if (s.length()>3) {
                count++;
            }
        }
        System.out.println(count);

        //StreamAPI结合lambda表达式
        //链式编程
        long count2 = data.stream().filter(s->s.length()>3).count();
        System.out.println(count2);
    }
```

2.3，使用StreamAPI的步骤

```
1. 创建一个Stream。 （创建）
2. 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。 （中间操作）
3. 使用一个终止操作来产生一个结果。该操作会强制他之前的延迟操作立即执行。在这之后，该Stream就不会再被使用了。（终止操作）
```

2.4，创建Stream的方式

上面已经演示了集合创建stream的方式

这里面再补充一种数组的方式

```
int[] array = new int[]{1,2,3};
IntStream stream = Arrays.stream(array);
long result = stream.filter(i -> i > 1).count();
System.out.println(result);
```

2.5，Stream的中间操作

2.5.1，筛选和切片(filter,limit,skip(n),distinct)

```
package com.hgz.jdk8;

/**
 * @author huangguizhao
 */
public class Employee{
    private String name;
    private int age;
    private int salary;
```

```
public Employee(String name, int age, int salary) {
    this.name = name;
    this.age = age;
    this.salary = salary;
}

public Employee() {
}

@Override
public String toString() {
    return "Employee{" +
            "name='" + name + '\'' +
            ", age=" + age +
            ", salary=" + salary +
            '}';
}

@Override
public boolean equals(Object obj) {
    return true;
}

@Override
public int hashCode() {
    return 1;
}



public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public int getSalary() {
    return salary;
}

public void setSalary(int salary) {
    this.salary = salary;
```

```
    }
}
```

```
public static void main(String[] args){
    List<Employee> employees = new ArrayList<>();
    employees.add(new Employee("zhangsan",20,8000));
    employees.add(new Employee("lisi",30,18000));
    employees.add(new Employee("wangwu",40,28000));
    employees.add(new Employee("wangwu",40,28000));

    //筛选
    Stream<Employee> employeeStream = employees.stream().filter(e -> e.getAge() > 30);
    employeeStream.forEach(t->System.out.print(t.getName()));

    //获取第一个
    Stream<Employee> limitStream = employees.stream().limit(1);
    limitStream.forEach(System.out::println);

    //跳过两个
    Stream<Employee> skipStream = employees.stream().skip(2);
    skipStream.forEach(System.out::println);

    System.out.println("------------------");
    //自动调用equals和hashcode
    Stream<Employee> distinctStream = employees.stream().distinct();
    distinctStream.forEach(System.out::println);

}
```

2.5.2，映射（map）

对象转换

```
// map--接收 Lambda ，
// 将元素转换成其他形式或提取信息。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。
Stream<String> stringStream = employees.stream().map(e -> e.getName());
stringStream.forEach(System.out::println);

//小写转换为大写
List<String> list = Arrays.asList("a","b","c");
Stream<String> upperCaseStream = list.stream().map(String::toUpperCase);
upperCaseStream.forEach(System.out::println);
```

2.5.3，排序（sorted）

```
System.out.println("-------------按年龄自然排序--------------");
Stream<Integer> sortedStream = employees.stream().map(Employee::getAge).sorted();
sortedStream.forEach(System.out::println);

System.out.println("-------------定制排序-----------------------");
```

```
Stream<Employee> sortedStream2 = employees.stream().sorted((x, y) -> {
    if (x.getAge() == y.getAge()) {
        return x.getName().compareTo(y.getName());
    } else {
        return x.getAge() - y.getAge();
    }
});

sortedStream2.forEach(System.out::println);
```

2.6，终止操作

2.6.1，遍历操作

forEach

2.6.2，查找和匹配

```
allMatch--检查是否匹配所有元素
anyMatch--检查是否至少匹配一个元素
noneMatch--检查是否没有匹配的元素
findFirst--返回第一个元素
findAny--返回当前流中的任意元素
count--返回流中元素的总个数
max--返回流中大值
min--返回流中小值
```

```
boolean b1 = employees.stream().allMatch(e -> e.getAge() > 30);
boolean b2 = employees.stream().anyMatch(e -> e.getAge() > 30);
Optional<Employee> max = employees.stream().max((x, y) -> x.getAge() - y.getAge());
System.out.println(b1);
System.out.println(b2);
System.out.println(max.get());
```

2.7，Stream的串行和并行

Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行

串行：

```
public static void main(String[] args){
    //初始化數據
    int max = 1000000;
    List<String> values = new ArrayList<>(max);
    for (int i = 0; i < max; i++) {
        UUID uuid = UUID.randomUUID();
        values.add(uuid.toString());
    }
    //串行計算
    long start = System.currentTimeMillis();
```

```
    long count = values.stream().sorted().count();
    System.out.println(count);
    long end = System.currentTimeMillis();
    long millis = end-start;
    System.out.println(millis);


}
```

并行

```
//并行計算
long start = System.currentTimeMillis();
long count = values.parallelStream().sorted().count();
System.out.println(count);
long end = System.currentTimeMillis();
long millis = end-start;
System.out.println(millis);
```