JDK1.8新特性

1、Date Time API

1-1、 Date Time Formatter

```
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH:mm:ss");
        LocalDateTime ld = LocalDateTime.now();

        System.out.println("DateTimeFormatter:"+ld);

        String formt = dtf.format(ld);
        System.out.println("格式化DateTimeFormatter：:"+formt);

        TemporalAccessor d = dtf.parse(formt);
        System.out.println("TemporalAccessor:"+d);
        TemporalAccessor ta = dtf.parse("2020年08月25日 12:53:51");
        System.out.println("TemporalAccessor 字符串转换：:"+ta);
```

效果

```
DateTimeFormatter:2022-04-18T09:33:10.855403300
格式化DateTimeFormatter：2022年04月18日 09:33:10
TemporalAccessor:{},ISO resolved to 2022-04-18T09:33:10
TemporalAccessor 字符串转换：{},ISO resolved to 2020-08-25T12:53:51
```

1-2、Duration和Period日期比较

```
        Instant in1 = Instant.now();
        System.out.println("Instant:"+in1);
        Thread.sleep(2000);
        Instant in2 = Instant.now();

        //比较时间
        Duration duration = Duration.between(in1,in2);
        System.out.println("日期比较：:"+duration);

        LocalDate l1 = LocalDate.now();
        //设置日期
        LocalDate l2 = LocalDate.of(2020,7,20);
        Period p1 = Period.between(l2,l1);
        System.out.println("整体相差时间：:"+p1);
        System.out.println("相差的年份：:"+p1.getYears());
        System.out.println("相差的月份：:"+p1.getMonths());

        System.out.println("相差的天数：:"+p1.getDays());
```

效果

```
Instant:2022-04-18T01:34:49.212166400Z
日期比较：PT2.1960435S
整体相差时间：P1Y8M29D
相差的年份：1
相差的月份：8
相差的天数：29
```

1-3、Local Date Time

```
        LocalDateTime ldt = LocalDateTime.now();
        System.out.println("local Date time:"+ldt);

        LocalDate ld = LocalDate.now();
        System.out.println("local Date:"+ld);

        LocalTime lt = LocalTime.now();
        System.out.println("local Time:"+lt);

        //设置某一个时间
        LocalDateTime ldt2 = LocalDateTime.of(2020,8,25,9,21,0);
        System.out.println("local date time:"+ldt2);
```

效果

```
local Date time:2022-04-18T09:39:53.581719200
local Date:2022-04-18
local Time:09:39:53.784094
local date time:2020-08-25T09:21
```

2、Lambda表达式

2-1、循环遍历

List集合

```
        List<String> list = new ArrayList<>();

        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        for(String s : list)
        {
            System.out.println(s);
        }
```

Lambda

```
list.forEach(  (x) -> System.out.println(x)   );
```

`list.forEach`相当于了for循环

`x`相当于循环中的变量`s`

只有一条语句，可以省略`{}`，直接输出就行了。

Map集合

```
       Map<String,Object> map = new HashMap<>();
        map.put("1",1);
        map.put("2",1);
        map.put("3",1);
        for(Map.Entry<String,Object> entry : map.entrySet())
        {
            System.out.println(entry.getKey()+">>"+entry.getValue());
        }
```

Lambda

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

2-2、接口

无参无返回方法

```
public interface Pet
{
    public void eat();

}
```

```
//匿名内部类写法
       Pet p = new Pet()
       {
           @Override
           public void eat()
           {
               System.out.println("匿名内部类");
           }


       };
```

Lambda

```
  Pet p1 = () ->
        {
            System.out.println("Lambda");
            System.out.println("Lambda2");
        };
        p1.eat();
```

有参有返回

```
public interface Pet2
{
    int number(int x,String name);
}
```

Lambda

```
        Pet2 p2 = (x, y) ->
        {
            System.out.println(x+","+y);

            return x*2;
        };
        p2.number(19,"wangyang");
```

3、Stream流

3-1、流的创建

```
 //集合  Collection的子类集合（单值集合）
       List<String> list = new ArrayList<String>();
       Stream<String> s1 = list.stream();
       Stream<String> s2 = list.parallelStream();

       //map不行
//        Map<String,Object> map = new HashMap<>();
//        System.out.println(map.stream());
```

3-2、流的使用

```
  //1.通过Collections中stream方法创建
       List<String> list = Arrays.asList("张三丰","张无忌","张翠山","谢逊","灭绝师太","赵敏","周芷若","小昭");
       //获得流
        Stream<String> s = list.stream();
        System.out.println(s);
//        s.forEach(x -> System.out.println(x));
        //根据条件过滤
//       Stream<String> filterStream =  s.filter( name -> name.startsWith("张"));
        //循环会造成流关闭（流就跟水龙头，开一个，少一个）
//       filterStream.forEach(x -> System.out.println(x));

        //显示前几个数据
//        Stream<String> limitStream = s.limit(5);
//        limitStream.forEach(x -> System.out.println(x));

        //跳过几个数据，显示后面
//        Stream<String> skipStream = s.skip(2);
//        skipStream.forEach(x -> System.out.println(x));

        //一共有多少个数据,会造成流关闭
        System.out.println(s.count());
//       s.forEach(x -> System.out.println(x));

        s.close();
```

4、方法引用

构建普通类

```
public class Car
{
    String name;
    public void createCar()
    {
        System.out.println("创建汽车："+name);
    }
```

```
8
9       public static void driver(Car c)
10      {
11          System.out.println(c.name+"开车");
12      }
13
14      public Car(String name)
15      {
16          this.name = name;
17      }
18  }
19
```

方法引入

```
1           Car c1 = new Car("大众");
2           Car c2 = new Car("红旗");
3           Car c3 = new Car("长城");
4
5           //new ArrayList<>();  list.add(c1);...
6           List<Car> list = Arrays.asList(c1,c2,c3);//等价于了下面4行代码
7   //        List<Car> list2 = new ArrayList<>();
8   //        list2.add(c1);
9   //        list2.add(c2);
10  //        list2.add(c3);
11          //在每次循环中调用了createCar方法
12          list.forEach( car -> car.createCar());
13          //方法引入
14          list.forEach(Car :: createCar);
15          //静态方法引入  ，需要将对象作为参数传递使用
16          list.forEach(Car :: driver);   //Car c1;Car c2 Car c3;作为参数传递进入方法了
    Car.driver();
17          //方法引入 系统对象
18          list.forEach(System.out :: println);
19
20          List<Integer> listInteger =Arrays.asList(1,23,4,5,6,7,8,3);
21
22          listInteger.forEach(System.out::println);
```

5、注解

5-1、系统注解

```
1   //注解：用来描述代码有什么作用
2   public class Test01
3   {
4       public static void main(String[] args)
5       {
```

```
        Student student = new Student();

        student.hello();
    }
}

@Deprecated
@SuppressWarnings("hello")
class Student
{
    @Deprecated
    @SuppressWarnings("")
    String name;

    @SuppressWarnings("警告：该方法很容出错误")
    @Deprecated
    public void hello()
    {

    }

}
```

5-2、自定义注解

```
/**
 *      ElementType.TYPE：类
 *      ElementType.CONSTRUCTOR:构造器
 *      ElementType.METHOD  :方法
 *       ElementType.FIELD:属性
 */
@Target( {ElementType.TYPE  , ElementType.CONSTRUCTOR  , ElementType.METHOD ,
ElementType.FIELD} )
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnncontation
{
    static final String name = "王杨";

    //value可以省略 value=""
    String value() default "863软件";
    int age() default  18;


}
```

5-3、元注解

```
@Retention - 标识这个注解怎么保存，是只在代码中，还是编入class文件中，或者是在运行时可以通过反射访问。
@Documented - 标记这些注解是否包含在用户文档中。
@Target - 标记这个注解应该是哪种 Java 成员。
@Inherited - 标记这个注解是继承于哪个注解类(默认 注解并没有继承于任何子类)
```

5-4、反射调用注解

1、获取类上的注解

```
Class<Teacher> clazz = Teacher.class;

        //获取Teacher类上的MyAnncontation注解
        MyAnncontation an = clazz.getAnnotation(MyAnncontation.class);

        System.out.println(an.age());
        System.out.println(an.value());
        System.out.println(an.name);
```

2、获取方法上的注解

```
       Class<Teacher> clazz = Teacher.class;

        Method m = clazz.getMethod("hello");

        MyAnncontation my = m.getAnnotation(MyAnncontation.class);
        System.out.println(my.value());
        System.out.println(my.age());
```

3、获取属性上的注解

```
       Class<Teacher> clazz = Teacher.class;

        Field f =  clazz.getField("name");

        MyAnncontation my = f.getAnnotation(MyAnncontation.class);

        System.out.println(my.age());
        System.out.println(my.value());
```