JDK8新特性

1.接口相关

> 在JDK8中接口中可以书写普通方法 和 静态方法
>
> 1.普通方法使用default来修饰
>
> 2.静态方法直接书写

```
public interface A {

    void m1();

    public default void m2(){
        System.out.println("接口中的普通方法");
    }

    public static void m3(){
        System.out.println("接口中的静态方法");
    }

}

class A1 implements  A{

    @Override
    public void m1() {
        System.out.println("重写m1方法");
    }


    @Override
    public void m2() {
        A.super.m2();
        System.out.println("对接口中的方法扩展");
    }


    public static void main(String[] args) {
        A.m3();
    }
}

class B{
    public static void m1(){
```

```
    }
}

class B1 extends B{
    public static void main(String[] args) {
        B1.m1();;
    }
}
```

2.集合类相关的

> 1.ArrayList加载数组时机发生了改变，由原来的执行无参构造初始化长度为10的数组改变为了第一次添加元素初始化长度为10的数组
>
> 2.HashMap由原来的数组 + 单向链表 变为了 数组 + 单向链表 + 红黑树

3.函数式接口

> 函数式接口： 函数式接口是指接口中只有一个抽象方法的接口 即可称之为函数式接口 SAM 接口
>
> SAM Single Abstract Method 通常使用@FunctionalInterface注解修饰
>
> 函数式编程： 函数式编程是一种编程思想 就像面向对象 、面向过程编程一样 属于一种编程思想
>
> 函数式编程在很多语言中都有 比较具有代表性 最早使用函数式编程思想的 Haskell
>
> 函数式编程特点： 强调使用哪个函数执行了什么操作 不关心是谁 调用这个函数
>
> 注意：代码越简洁 前期我们理解起来越难 后续用起来越舒服

4.lambda表达式

> 使用lambda表达式 其实属于对函数式接口语法的一种优化 即书写起来更为简洁 优雅 属于JDK1.8新增的内容
>
> 使用格式 参数列表，可以不加类型，直接写参数名称 (参数列表)->{方法体};

```
package com.atguigu.test1;

/**
 *  lambda表达式的前提 ： 必须为函数式接口 (抽象类无法使用lambda表达式 )
 */

@FunctionalInterface
public interface C {
    void m1();
```

```
}



class C1 implements  C{
    @Override
    public void m1() {

    }
}

class TestC{
    public static void main(String[] args) {
        C c1 = new C() {
            @Override
            public void m1() {
                System.out.println("匿名内部类的方式重写m1方法");
            }
        };

        c1.m1();

        // 没有参数 没有返回值 方法体只有一条语句
        C c2 = ()-> System.out.println("lambda表达式的方式重写m1方法");

        c2.m1();

        System.out.println(c2);


        D d1 = (int a,String b)-> System.out.println("hello world " + a  + b);

        // 有参数 没有返回值 方法体只有一条语句
        d1.m1(10, "abc");

        D d2 = (a,b)-> System.out.println("hello world" + a + b );

        // 有参数 有返回值 方法体只有一条语句 return必须省略 直接写返回值
        E e1 = (a)->  a.toString();

        System.out.println(e1.m1(100).length());

        // 有参数 有返回值  方法体多条语句
        E e2 = (a)->{
            System.out.println("方法体1");
            System.out.println("方法体2");
            System.out.println("方法体3");
            return a.toString().toUpperCase();
        };

        System.out.println(e2.m1(100));
```

```
    }
}
interface E{
    String m1(Integer a);
}



interface D{
    void m1(int a,String b);
}
```

5.方法引用

> 方法引用： 将一个方法的方法体(功能)用作当前函数式接口中的抽象方法的方法体(功能) 引用
>
> 构造方法引用 类名 :: new 根据函数式接口中的抽象方法 来匹配一个构造器 用于本方法的方法体执行
>
> 普通方法引用 对象名 :: 方法名 根据函数式接口中的抽象方法 来匹配一个普通方法 用于本方法的方法体执行
>
> 静态方法引用 类名 :: 方法名 根据函数式接口中的抽象方法 来匹配一个静态方法 用于本方法的方法体执行

```
package com.atguigu.test2;


import java.awt.print.Book;

/**
 * @author WHD
 * @description TODO
 * @date 2023/4/28 10:18
 *  方法引用 ： 将一个方法的方法体(功能)用作当前函数式接口中的抽象方法的方法体(功能)  引用
 *
 *  构造方法引用 类名 :: new 根据函数式接口中的抽象方法 来匹配一个构造器  用于本方法的方法体执行
 *  普通方法引用 对象名 :: 方法名 根据函数式接口中的抽象方法 来匹配一个普通方法  用于本方法的方法体执行
 *  静态方法引用 类名 :: 方法名 根据函数式接口中的抽象方法 来匹配一个静态方法  用于本方法的方法体执行
 */
public class Note {
    public static void main(String[] args) {
        A a1 = ()-> System.out.println("lambda表达式的方式实现m1方法");

        A a2 = Student :: new;
        a2.m1();

        B b1 = Student :: new;
        b1.m1("abc", 22);

        System.out.println(b1);

        System.out.println("------------------------------------------------------");

        String str1 = "abc hello world";
```

```
        System.out.println(str1.startsWith("abc"));
        System.out.println(str1.endsWith("abc"));


        C<String,Boolean> c1 = str1 :: startsWith;

        System.out.println(c1.m1("abc"));

        C<Integer,Character> c2 = str1 :: charAt;

        System.out.println(c2.m1(1));

        C<String,String> c3 = str1 :: concat;

        System.out.println(c3.m1("666"));

        C<String,Boolean> c4 = str1 :: contains;

        System.out.println(c4.m1("a"));

        System.out.println("------------------------------------------------------");


        C<Integer,String> c5 = String :: valueOf;

        System.out.println(c5.m1(100).length());

        // Math.random()

        D<Double> d1 = Math :: random;

        System.out.println(d1.m1());

        C<Double,Double> c6 = Math :: ceil;

        System.out.println(c6.m1(5.3));

        // System.out.println();
        E<String> e1 = System.out :: println;
        e1.m1("hello world");

    }
}

interface E<P>{
    void m1(P p);
}

interface D<R>{
    R m1();
}

interface C<P,R>{
```

```
    R m1(P p);
}

interface A{
    void m1();
}
interface B{
    void m1(String a,int b);
}

class Student{
    private String name;
    private int age;

    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 Student(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Student有参构造方法");
    }

    public Student() {
        System.out.println("Student无参构造方法");
    }
}
```

###