反射、枚举以及lambda表达式

【本节目标】

1. **掌握反射**
2. **掌握枚举**
3. **掌握lambda表达式使用**

1 定义

Java的反射（reflection）机制是在**运行**状态中，对于任意一个类，都能够知道这个类的**所有属性和方法**；对于任意一个对象，都能够调用它的任意方法和属性，既然能拿到那么，我们就可以修改部分类型信息；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射（reflection）机制。

2 用途(了解)

1. 在日常的第三方应用开发过程中，经常会遇到**某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放**，这时候就可以利用Java的反射机制通过反射来获取所需的私有成员或是方法 。
2. 反射最重要的用途就是**开发各种通用框架**，比如在spring中，我们将所有的类Bean交给spring容器管理，无论是XML配置Bean还是注解配置，当我们从容器中获取Bean来依赖注入时，容器会读取配置，而配置中给的就是类的信息，spring根据这些信息，需要创建那些Bean，spring就动态的创建这些类。

3 反射基本信息

Java程序中许多对象在运行时会出现两种类型：**运行时类型(RTTI)和编译时类型**，例如Person p = new Student()；这句代码中p在编译时类型为Person，运行时类型为Student。程序需要在运行时发现对象和类的真实信息。而通过使用反射程序就能判断出该对象和类属于哪些类。

4 反射相关的类（重要）

类名	用途
Class类	代表类的实体，在运行的Java应用程序中表示类和接口
Field类	代表类的成员变量/类的属性
Method类	代表类的方法
Constructor类	代表类的构造方法

4.1 Class类(反射机制的起源)

Class帮助文档代表类的实体，在运行的Java应用程序中表示类和接口 .

Java文件被编译后，生成了.class文件，JVM此时就要去解读.class文件 ,被编译后的Java文件.class也被JVM解析为一个对象，这个对象就是`java.lang.Class` .这样当程序在运行时，每个java文件就最终变成了Class类对象的一个实例。我们通过Java的反射机制应用到这个实例，就可以去**获得甚至去添加改变这个类的属性和动作**，使得这个类成为一个动态的类 .

4.1.1 Class类中的相关方法(方法的使用方法在后边的示例当中)

- **(重要)**常用获得类相关的方法

方法	用途
getClassLoader()	获得类的加载器
getDeclaredClasses()	返回一个数组，数组中包含该类中所有类和接口类的对象(包括私有的)
forName(String className)	根据类名返回类的对象
newInstance()	创建类的实例
getName()	获得类的完整路径名字

- **(重要)**常用获得类中属性相关的方法(以下方法返回值为Field相关)

方法	用途
getField(String name)	获得某个公有的属性对象
getFields()	获得所有公有的属性对象
getDeclaredField(String name)	获得某个属性对象
getDeclaredFields()	获得所有属性对象

- (了解)获得类中注解相关的方法

方法	用途
getAnnotation(Class annotationClass)	返回该类中与参数类型匹配的公有注解对象
getAnnotations()	返回该类所有的公有注解对象
getDeclaredAnnotation(Class annotationClass)	返回该类中与参数类型匹配的所有注解对象
getDeclaredAnnotations()	返回该类所有的注解对象

- **(重要)**获得类中构造器相关的方法（以下方法返回值为Constructor相关）

方法	用途
getConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的公有构造方法
getConstructors()	获得该类的所有公有构造方法
getDeclaredConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的构造方法
getDeclaredConstructors()	获得该类所有构造方法

- **(重要)**获得类中方法相关的方法（以下方法返回值为Method相关）

方法	用途
getMethod(String name, Class...<?> parameterTypes)	获得该类某个公有的方法
getMethods()	获得该类所有公有的方法
getDeclaredMethod(String name, Class...<?> parameterTypes)	获得该类某个方法
getDeclaredMethods()	获得该类所有方法

4.2 反射示例

4.2.1 获得Class对象的三种方式

在反射之前，我们需要做的第一步就是先拿到当前需要反射的类的Class对象，然后通过Class对象的核心方法，达到反射的目的，即：在**运行**状态中，对于任意一个类，都能够知道这个类的**所有属性和方法**；对于任意一个对象，都能够调用它的任意方法和属性，既然能拿到那么，我们就可以修改部分类型信息。

第一种，使用 Class.forName("类的全路径名"); 静态方法。

前提：已明确类的全路径名。

第二种，使用 .class 方法。

说明：仅适合在编译前就已经明确要操作的 Class

第三种，使用类对象的 getClass() 方法

示例：

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-20
 * Time: 15:24
 */
class Student{
    //私有属性name
    private String name = "bit";
```

```
    //公有属性age
    public int age = 18;
    //不带参数的构造方法
    public Student(){
        System.out.println("Student()");
    }

    private Student(String name,int age) {
        this.name = name;
        this.age = age;
        System.out.println("Student(String,name)");
    }

    private void eat(){
        System.out.println("i am eat");
    }

    public void sleep(){
        System.out.println("i am pig");
    }

    private void function(String str) {
        System.out.println(str);
    }

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

public class TestDemo {
    public static void main(String[] args) {
        /*
        1.通过getClass获取Class对象
         */
        Student s1 = new Student();
        Class c1 = s1.getClass();
        /*
        2.直接通过 类名.class 的方式得到,该方法最为安全可靠，程序性能更高
        这说明任何一个类都有一个隐含的静态成员变量 class
         */
        Class c2 = Student.class;
        /*
        3、通过 Class 对象的 forName() 静态方法来获取，用的最多，
        但可能抛出 ClassNotFoundException 异常
         */
        Class c3 = null;
        try {

            //注意这里是类的全路径，如果有包需要加包的路径
```

```
            c3 = Class.forName("Student");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        //一个类在 JVM 中只会有一个 Class 实例,即我们对上面获取的
        //c1,c2,c3进行 equals 比较，发现都是true
        System.out.println(c1.equals(c2));
        System.out.println(c1.equals(c3));
        System.out.println(c2.equals(c3));
    }
}
```

4.2.2 反射的使用

接下来我们开始使用反射，我们依旧反射上面的Student类，把反射的逻辑写到另外的类当中进行理解

注意：所有和反射相关的包都在`import java.lang.reflect`包下面。

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-20
 * Time: 16:31
 */
public class ReflectClassDemo {
    // 创建对象
    public static void reflectNewInstance() {
        try {
            Class<?> classStudent = Class.forName("Student");
            Object objectStudent = classStudent.newInstance();
            Student student = (Student) objectStudent;
            System.out.println("获得学生对象："+student);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    // 反射私有的构造方法 屏蔽内容为获得公有的构造方法
    public static void reflectPrivateConstructor() {
        try {
            Class<?> classStudent = Class.forName("Student");
            //注意传入对应的参数
            Constructor<?> declaredConstructorStudent = classStudent.getDeclaredConstructor(String.class,int.class);
            //Constructor<?> declaredConstructorStudent = classStudent.getConstructor();
            //设置为true后可修改访问权限
            declaredConstructorStudent.setAccessible(true);
            Object objectStudent = declaredConstructorStudent.newInstance("高博",15);
            //Object objectStudent = declaredConstructorStudent.newInstance();

            Student student = (Student) objectStudent;
```

```
            System.out.println("获得私有构造哈数且修改姓名和年龄："+student);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 反射私有属性
    public static void reflectPrivateField() {
        try {
            Class<?> classStudent = Class.forName("Student");

            Field field  = classStudent.getDeclaredField("name");
            field.setAccessible(true);
            //可以修改该属性的值
            Object objectStudent = classStudent.newInstance();
            Student student = (Student) objectStudent;

            field.set(student,"小明");

            String name = (String) field.get(student);
            System.out.println("反射私有属性修改了name："+ name);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 反射私有方法
    public static void reflectPrivateMethod() {
        try {
            Class<?> classStudent = Class.forName("Student");
            Method methodStudent = classStudent.getDeclaredMethod("function",String.class);
            System.out.println("私有方法的方法名为："+methodStudent.getName());
            //私有的一般都要加
            methodStudent.setAccessible(true);
            Object objectStudent = classStudent.newInstance();
            Student student = (Student) objectStudent;
            methodStudent.invoke(student,"我是给私有的function函数传的参数");

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        //reflectNewInstance();
        //reflectPrivateConstructor();
        //reflectPrivateField();
        reflectPrivateMethod();
    }
}
```

5、反射优点和缺点

优点：

1. 对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法
2. 增加程序的灵活性和扩展性，降低耦合性，提高自适应能力
3. 反射已经运用在了很多流行框架如：Struts、Hibernate、Spring 等等。

缺点：

1. 使用反射会有效率问题。会导致程序效率降低。具体参考这里：http://www.imooc.com/article/293679
2. 反射技术绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂 。

6 重点总结

1. 反射的意义
2. 反射重要的几个类： Class类 、Field类、 Method类、 Constructor类
3. 学会合理利用反射，一定要在安全环境下使用。

枚举的使用

1 背景及定义

枚举是在JDK1.5以后引入的。主要用途是：将一组常量组织起来，在这之前表示一组常量通常使用定义常量的方式：

```
public static  final int RED = 1;
public static  final int GREEN = 2;
public static  final int BLACK = 3;
```

但是常量举例有不好的地方，例如：可能碰巧有个数字1，但是他有可能误会为是RED，现在我们可以直接用枚举来进行组织，这样一来，就拥有了类型，枚举类型。而不是普通的整形1.

```
public enum TestEnum {
   RED,BLACK,GREEN;
}
```

优点：将常量组织起来统一进行管理

场景：错误状态码，消息类型，颜色的划分，状态机等等....

本质：是`java.lang.Enum`的子类，也就是说，自己写的枚举类，就算没有显示的继承`Enum`，但是其默认继承了这个类。

2 使用

1、switch语句

```
public enum TestEnum {
   RED,BLACK,GREEN,WHITE;
   public static void main(String[] args) {

      TestEnum testEnum2 = TestEnum.BLACK;
```

```
        switch (testEnum2) {
            case RED:
                System.out.println("red");
                break;
            case BLACK:
                System.out.println("black");
                break;
            case WHITE:
                System.out.println("WHITE");
                break;
            case GREEN:
                System.out.println("black");
                break;
            default:
                break;
        }
    }
}
```

2、常用方法

`Enum`类的常用方法

方法名称	描述
values()	以数组形式返回枚举类型的所有成员
ordinal()	获取枚举成员的索引位置
valueOf()	将普通字符串转换为枚举实例
compareTo()	比较两个枚举成员在定义时的顺序

示例一：

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-06
 * Time: 16:19
 */
public enum TestEnum {
    RED,BLACK,GREEN,WHITE;
    public static void main(String[] args) {
        TestEnum[] testEnum2 =  TestEnum.values();
        for (int i = 0; i < testEnum2.length; i++) {
            System.out.println(testEnum2[i] + " " + testEnum2[i].ordinal());
        }
        System.out.println("==========================");
        System.out.println(TestEnum.valueOf("GREEN"));
    }
```

```
}
```

示例二：

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-06
 * Time: 16:19
 */
public enum TestEnum {
    RED,BLACK,GREEN,WHITE;
    public static void main(String[] args) {
        //拿到枚举实例BLACK
        TestEnum testEnum = TestEnum.BLACK;
        //拿到枚举实例RED
        TestEnum testEnum21 = TestEnum.RED;
        System.out.println(testEnum.compareTo(testEnum21));
        System.out.println(BLACK.compareTo(RED));
        System.out.println(RED.compareTo(BLACK));
    }
}
```

刚刚说过，在Java当中枚举实际上就是一个类。所以我们在定义枚举的时候，还可以这样定义和使用枚举：

重要：枚举的构造方法默认是私有的

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-06
 * Time: 16:19
 */
public enum TestEnum {
    RED("red",1),BLACK("black",2),WHITE("white",3),GREEN("green",4);
    private  String name;
    private  int key;

    /**
     * 1、当枚举对象有参数后，需要提供相应的构造函数
     * 2、枚举的构造函数默认是私有的 这个一定要记住
     * @param name
     * @param key
     */
    private TestEnum (String name,int key) {
        this.name = name;
        this.key = key;
    }

    public static TestEnum getEnumKey (int key) {
```

```
        for (TestEnum t: TestEnum.values()) {
            if(t.key == key) {
                return t;
            }
        }
        return null;
    }

    public static void main(String[] args) {
        System.out.println(getEnumKey(2));
    }

}
```

3 枚举优点缺点

优点:

1. 枚举常量更简单安全 。
2. 枚举具有内置方法 ， 代码更优雅

缺点:

1. 不可继承，无法扩展

4 枚举和反射

4.1 枚举是否可以通过反射，拿到实例对象呢?

我们刚刚在反射里边看到了，任何一个类，哪怕其构造方法是私有的，我们也可以通过反射拿到他的实例对象，那么枚举的构造方法也是私有的，我们是否可以拿到呢？接下来，我们来实验一下：

同样利用上述提供的枚举类来进行举例：

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-02-24
 * Time: 16:13
 */
public enum TestEnum {

    RED("red",1),BLACK("black",2),WHITE("white",3),GREEN("green",4);
    private  String name;
    private  int key;

    /**
     * 1、当枚举对象有参数后，需要提供相应的构造函数
     * 2、枚举的构造函数默认是私有的 这个一定要记住
     * @param name
     * @param key

     */
```

```
    private TestEnum (String name,int key) {
        this.name = name;
        this.key = key;
    }

    public static TestEnum getEnumKey (int key) {
        for (TestEnum t: TestEnum.values()) {
            if(t.key == key) {
                return t;
            }
        }
        return null;
    }

    public static void reflectPrivateConstructor() {
        try {
            Class<?> classStudent = Class.forName("TestEnum");
            //注意传入对应的参数,获得对应的构造方法来构造对象,当前枚举类是提供了两个参数分别是String和int。
            Constructor<?> declaredConstructorStudent = classStudent.getDeclaredConstructor(String.class,int.class);
            //设置为true后可修改访问权限
            declaredConstructorStudent.setAccessible(true);
            Object objectStudent = declaredConstructorStudent.newInstance("绿色",666);
            TestEnum testEnum = (TestEnum) objectStudent;
            System.out.println("获得枚举的私有构造函数："+testEnum);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        reflectPrivateConstructor();
    }

}
```

输出结果：

```
java.lang.NoSuchMethodException: TestEnum.<init>(java.lang.String, int)
    at java.lang.Class.getConstructor0(Class.java:3082)
    at java.lang.Class.getDeclaredConstructor(Class.java:2178)
    at TestEnum.reflectPrivateConstructor(TestEnum.java:40)
    at TestEnum.main(TestEnum.java:54)
```

老铁们啊，看到没有哇！异常信息是：`java.lang.NoSuchMethodException: TestEnum.<init>(java.lang.String, int)`,什么意思是：就是没有对应的构造方法，我的天呐！我们提供的枚举的构造方法就是两个参数分别是`String`和`int`啊！！！！问题出现在哪里呢？还记不记得我们说过的，我们所有的枚举类，都是默认继承与`java.lang.Enum`,说到继承，继承了什么？继承了父类除构造函数外的所有东西，并且子类要帮助父类进行构造！而我们写的类，并没有帮助父类构造！那意思是，我们要在自己的枚举类里面，提供super吗？不是的，枚举比较特殊，虽然我们写的是两个，但是默认他还添加了两个参数，哪两个参数呢？我们看一下Enum类的源码：

```
protected Enum(String name, int ordinal) {
    this.name = name;
    this.ordinal = ordinal;
}
```

也就是说，我们自己的构造函数有两个参数一个是String一个是int，同时他默认后边还会给两个参数，一个是String一个是int。也就是说，这里我们正确给的是4个参数：

```
 public static void reflectPrivateConstructor() {
      try {
         Class<?> classStudent = Class.forName("TestEnum");
         //注意传入对应的参数,获得对应的构造方法来构造对象,当前枚举类是提供了两个参数分别是String和int。
         Constructor<?> declaredConstructorStudent =
classStudent.getDeclaredConstructor(String.class,int.class,String.class,int.class);
         //设置为true后可修改访问权限
         declaredConstructorStudent.setAccessible(true);
        //后两个为子类参数，大家可以将当前枚举类的key类型改为double验证
         Object objectStudent = declaredConstructorStudent.newInstance("父类参数",666,"子类参数",888);
         TestEnum testEnum = (TestEnum) objectStudent;
         System.out.println("获得枚举的私有构造函数："+testEnum);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
```

此时运行程序结果是：

```
java.lang.IllegalArgumentException: Cannot reflectively create enum objects
    at java.lang.reflect.Constructor.newInstance(Constructor.java:416)
    at TestEnum.reflectPrivateConstructor(TestEnum.java:46)
    at TestEnum.main(TestEnum.java:55)
```

嗯！没错，他还报错了，不过这次就是我想要的结果！此时的异常信息显示，是我的一个方法这个方法是：`newInstance()`报错了！没错，问题就是这里，我们来看一下这个方法的源码，为什么会抛出`java.lang.IllegalArgumentException:`异常呢？

源码显示：

```
@NotNull @CallerSensitive
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
           IllegalArgumentException, InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz,  obj: null, modifiers);
        }
    }
    if ((clazz.getModifiers() & Modifier.ENUM) != 0)
        throw new IllegalArgumentException("Cannot reflectively create enum objects");
    ConstructorAccessor ca = constructorAccessor;   // read volatile
    if (ca == null) {
        ca = acquireConstructorAccessor();
    }
    /unchecked/
    T inst = (T) ca.newInstance(initargs);
    return inst;
}
```

是的，枚举在这里被过滤了，你不能通过反射获取枚举类的实例！这道题是2017年阿里巴巴曾经问到的一个问题，不看不知道，一看吓一跳！同学们记住这个坑。**原版问题是：为什么枚举实现单例模式是安全的？**希望同学们记住这个问题！

5 总结

1、枚举本身就是一个类，其构造方法默认为私有的，且都是默认继承与`java.lang.Enum`

2、枚举可以避免反射和序列化问题

3、枚举的优点和缺点

面试问题(单例模式学完后可以回顾)：

1、写一个单例模式。

```
public class Singleton {
   private volatile static Singleton uniqueInstance;
   private Singleton() {}
   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         synchronized (Singleton.class){
            if(uniqueInstance == null){//进入区域后，再检查一次，如果仍是null,才创建实例
               uniqueInstance = new Singleton();
            }

         }
```

```
        }
        return uniqueInstance;
    }
}
```

2、用静态内部类实现一个单例模式

```
class Singleton {
    /** 私有化构造器 */
    private Singleton() {

    }
    /** 对外提供公共的访问方法 */
    public static Singleton getInstance() {
        return UserSingletonHolder.INSTANCE;
    }

    /** 写一个静态内部类，里面实例化外部类 */
    private static class UserSingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton u1 = Singleton.getInstance();
        Singleton u2 = Singleton.getInstance();
        System.out.println("两个实例是否相同："+ (u1==u2));
    }
}
```

3、用枚举实现一个单例模式

```
public enum TestEnum {
    INSTANCE;
    public TestEnum getInstance(){
        return INSTANCE;
    }
    public static void main(String[] args) {
        TestEnum singleton1=TestEnum.INSTANCE;
        TestEnum singleton2=TestEnum.INSTANCE;
        System.out.println("两个实例是否相同："+(singleton1==singleton2));
    }
}
```

Lambda表达式

1 背景

Lambda表达式是Java SE 8中一个重要的新特性。lambda表达式允许你通过表达式来代替功能接口。 lambda表达式就和方法一样,它提供了一个正常的参数列表和一个使用这些参数的主体(body,可以是一个表达式或一个代码块)。 **Lambda 表达式（Lambda expression）**，基于数学中的λ演算得名，也可称为闭包（Closure）。

1.1 Lambda表达式的语法

基本语法: **(parameters) -> expression** 或 **(parameters) ->{ statements; }**

Lambda表达式由三部分组成:

1. **paramaters**：类似方法中的形参列表，这里的参数是函数式接口里的参数。这里的参数类型可以明确的声明也可不声明而由JVM隐含的推断。另外当只有一个推断类型时可以省略掉圆括号。
2. **->**：可理解为“被用于”的意思
3. **方法体**：可以是表达式也可以代码块，是函数式接口里方法的实现。代码块可返回一个值或者什么都不反回，这里的代码块块等同于方法的方法体。如果是表达式，也可以返回一个值或者什么都不反回。

```
// 1. 不需要参数,返回值为 2
() -> 2

// 2. 接收一个参数(数字类型),返回其2倍的值
x -> 2 * x

// 3. 接受2个参数(数字),并返回他们的和
(x, y) -> x + y

// 4. 接收2个int型整数,返回他们的乘积
(int x, int y) -> x * y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)
(String s) -> System.out.print(s)
```

1.2 函数式接口

要了解Lambda表达式,首先需要了解什么是函数式接口，函数式接口定义：一个接口有且只有一个抽象方法 。

注意:

1. 如果一个接口只有一个抽象方法，那么该接口就是一个函数式接口
2. 如果我们在某个接口上声明了 `@FunctionalInterface` 注解，那么编译器就会按照函数式接口的定义来要求该接口，这样如果有两个抽象方法，程序编译就会报错的。所以，从某种意义上来说，只要你保证你的接口中只有一个抽象方法，你可以不加这个注解。加上就会自动进行检测的。

定义方式:

```
@FunctionalInterface
interface NoParameterNoReturn {
   //注意：只能有一个方法
   void test();
}
```

但是这种方式也是可以的:

```
@FunctionalInterface
interface NoParameterNoReturn {
    void test();
    default void test2() {
        System.out.println("JDK1.8新特性，default默认方法可以有具体的实现");
    }
}
```

2 Lambda表达式的基本使用

首先，我们实现准备好几个接口：

```
//无返回值无参数
@FunctionalInterface
interface NoParameterNoReturn {
    void test();
}
//无返回值一个参数
@FunctionalInterface
interface OneParameterNoReturn {
    void test(int a);
}
//无返回值多个参数
@FunctionalInterface
interface MoreParameterNoReturn {
    void test(int a,int b);
}
//有返回值无参数
@FunctionalInterface
interface NoParameterReturn {
    int test();
}

//有返回值一个参数
@FunctionalInterface
interface OneParameterReturn {
    int test(int a);
}
//有返回值多参数
@FunctionalInterface
interface MoreParameterReturn {
    int test(int a,int b);
}
```

我们在上面提到过，Lambda可以理解为：Lambda就是匿名内部类的简化，实际上是创建了一个类，实现了接口，重写了接口的方法 。

没有使用lambda表达式的时候的调用方式：

```
NoParameterNoReturn noParameterNoReturn = new NoParameterNoReturn(){
    @Override
    public void test() {
        System.out.println("hello");
    }
};

noParameterNoReturn.test();
```

具体使用见以下示例代码：

```
public class TestDemo {
    public static void main(String[] args) {
        NoParameterNoReturn noParameterNoReturn = ()->{
            System.out.println("无参数无返回值");
        };
        noParameterNoReturn.test();
        OneParameterNoReturn oneParameterNoReturn = (int a)->{
            System.out.println("一个参数无返回值："+ a);
        };
        oneParameterNoReturn.test(10);

        MoreParameterNoReturn moreParameterNoReturn = (int a,int b)->{
            System.out.println("多个参数无返回值："+a+" "+b);
        };
        moreParameterNoReturn.test(20,30);

        NoParameterReturn noParameterReturn = ()->{
            System.out.println("有返回值无参数！");
            return 40;
        };
        //接收函数的返回值
        int ret = noParameterReturn.test();
        System.out.println(ret);
        OneParameterReturn oneParameterReturn = (int a)->{
            System.out.println("有返回值有一个参数！");
            return a;
        };

        ret = oneParameterReturn.test(50);
        System.out.println(ret);

        MoreParameterReturn moreParameterReturn = (int a,int b)->{
            System.out.println("有返回值多个参数！");
            return a+b;
        };
        ret = moreParameterReturn.test(60,70);
        System.out.println(ret);
    }
}
```

2.1 语法精简

1. 参数类型可以省略，如果需要省略，每个参数的类型都要省略。
2. 参数的小括号里面只有一个参数，那么小括号可以省略
3. 如果方法体当中只有一句代码，那么大括号可以省略
4. 如果方法体中只有一条语句，且是return语句，那么大括号可以省略，且去掉return关键字。

示例代码：

```
public static void main(String[] args) {
    MoreParameterNoReturn moreParameterNoReturn = ( a, b)->{
        System.out.println("无返回值多个参数，省略参数类型："+a+" "+b);
    };
    moreParameterNoReturn.test(20,30);

    OneParameterNoReturn oneParameterNoReturn = a ->{
        System.out.println("无参数一个返回值,小括号可以胜率："+ a);
    };
    oneParameterNoReturn.test(10);

    NoParameterNoReturn noParameterNoReturn = ()->System.out.println("无参数无返回值，方法体中只有一行代码");
    noParameterNoReturn.test();

    //方法体中只有一条语句，且是return语句
    NoParameterReturn noParameterReturn = ()-> 40;
    int ret = noParameterReturn.test();
    System.out.println(ret);

}
```

3 变量捕获

Lambda 表达式中存在变量捕获 ， 了解了变量捕获之后，我们才能更好的理解Lambda 表达式的作用域 。Java当中的匿名类中，会存在变量捕获。

3.1 匿名内部类

匿名内部类就是没有名字的内部类 。我们这里只是为了说明变量捕获，所以，匿名内部类只要会使用就好，那么下面我们来，简单的看看匿名内部类的使用就好了。

具体想详细了解的同学戳这里：https://www.cnblogs.com/SQP51312/p/6100314.html

我们通过简单的代码来学习一下：

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAOBO
 * Date: 2020-04-15
 * Time: 16:16
 */

class Test {
```

```
    public void func(){
        System.out.println("func()");
    }
}
public class TestDemo {
    public static void main(String[] args) {
        new Test(){
            @Override
            public void func() {
                System.out.println("我是内部类，且重写了func这个方法！");
            }
        };
    }
}
```

在上述代码当中的main函数当中，我们看到的就是一个匿名内部类的简单的使用。

3.2 匿名内部类的变量捕获

```
class Test {
    public void func(){
        System.out.println("func()");
    }
}
public class TestDemo {
    public static void main(String[] args) {
        int a = 100;
        new Test(){
            @Override
            public void func() {
                System.out.println("我是内部类，且重写了func这个方法！");
                System.out.println("我是捕获到变量 a == "+a
                        +" 我是一个常量，或者是一个没有改变过值的变量！");
            }
        };
    }
}
```

在上述代码当中的变量a就是，捕获的变量。这个变量要么是被final修饰，如果不是被final修饰的 你要保证在使用之前，没有修改。如下代码就是错误的代码。

```
public class TestDemo {
    public static void main(String[] args) {
        int a = 100;
        new Test(){
            @Override
            public void func() {
                a = 99;
                System.out.println("我是内部类，且重写了func这个方法！");
                System.out.println("我是捕获到变量 a == "+a

                        +" 我是一个常量，或者是一个没有改变过值的变量！");
```

```
        }
      };
    }
}
```

该代码直接编译报错。

3.3 Lambda的变量捕获

在Lambda当中也可以进行变量的捕获，具体我们看一下代码。

```
@FunctionalInterface
interface NoParameterNoReturn {
    void test();
}

public static void main(String[] args) {
        int a = 10;
        NoParameterNoReturn noParameterNoReturn = ()->{
            // a = 99; error
            System.out.println("捕获变量："+a);
        };
        noParameterNoReturn.test();
}
```

4 Lambda在集合当中的使用

为了能够让Lambda和Java的集合类集更好的一起使用，集合当中，也新增了部分接口，以便与Lambda表达式对接。

对应的接口	新增的方法
Collection	removeIf() spliterator() stream() parallelStream() forEach()
List	replaceAll() sort()
Map	getOrDefault() forEach() replaceAll() putIfAbsent() remove() replace() computeIfAbsent() computeIfPresent() compute() merge()

以上方法的作用可自行查看我们发的帮助手册。我们这里会示例一些方法的使用。注意：Collection的forEach()方法是从接口 java.lang.Iterable 拿过来的。

4.1 Collection接口

forEach() 方法演示

该方法在接口 Iterable 当中，原型如下：

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

该方法表示：对容器中的每个元素执行action指定的动作 。

```
public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("bit");
        list.add("hello");
        list.add("lambda");
        list.forEach(new Consumer<String>(){
            @Override
            public void accept(String str){
                //简单遍历集合中的元素。
                System.out.print(str+" ");
            }
        });
    }
```

输出结果：Hello bit hello lambda

我们可以修改为如下代码：

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("Hello");
    list.add("bit");
    list.add("hello");
    list.add("lambda");
    //表示调用一个，不带有参数的方法，其执行花括号内的语句，为原来的函数体内容。
    list.forEach(s -> {
        System.out.println(s);
    });
}
```

输出结果：Hello bit hello lambda

4.2 List接口

sort()方法的演示

sort方法源码：该方法根据c指定的比较规则对容器元素进行排序。

```
public void sort(Comparator<? super E> c) {
    final int expectedModCount = modCount;
    Arrays.sort((E[]) elementData, 0, size, c);
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}
```

使用示例：

```
public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("bit");
        list.add("hello");
        list.add("lambda");
        list.sort(new Comparator<String>() {
            @Override
            public int compare(String str1, String str2){
                //注意这里比较长度
                return str1.length()-str2.length();
            }
        });
        System.out.println(list);
}
```

输出结果：bit, Hello, hello, lambda

修改为lambda表达式：

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("Hello");
    list.add("bit");
    list.add("hello");
    list.add("lambda");
    //调用带有2个参数的方法，且返回长度的差值
    list.sort((str1,str2)-> str1.length()-str2.length());
    System.out.println(list);
}
```

输出结果：bit, Hello, hello, lambda

4.3 Map接口

`HashMap`的`forEach()`

该方法原型如下：

```
default void forEach(BiConsumer<? super K, ? super V> action) {
```

```
    Objects.requireNonNull(action);
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch(IllegalStateException ise) {
            // this usually means the entry is no longer in the map.
            throw new ConcurrentModificationException(ise);
        }
        action.accept(k, v);
    }
}
```

作用是对Map中的每个映射执行action指定的操作。

代码示例：

```
public static void main(String[] args) {
    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "hello");
    map.put(2, "bit");
    map.put(3, "hello");
    map.put(4, "lambda");
    map.forEach(new BiConsumer<Integer, String>(){
        @Override
        public void accept(Integer k, String v){
            System.out.println(k + "=" + v);
        }
    });
}
```

输出结果：

1=hello 2=bit 3=hello 4=lambda

使用lambda表达式后的代码：

```
public static void main(String[] args) {
    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "hello");
    map.put(2, "bit");
    map.put(3, "hello");
    map.put(4, "lambda");
    map.forEach((k,v)->  System.out.println(k + "=" + v));
}
```

输出结果：

1=hello 2=bit 3=hello 4=lambda

5 总结

Lambda表达式的优点很明显，在代码层次上来说，使代码变得非常的简洁。缺点也很明显，代码不易读。

优点：

1. 代码简洁，开发迅速
2. 方便函数式编程
3. 非常容易进行并行计算
4. Java 引入 Lambda，改善了集合操作

缺点：

1. 代码可读性变差
2. 在非并行计算中，很多计算未必有传统的 for 性能要高
3. 不容易进行调试