

附录A Java高级技术介绍

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 掌握**Java**枚举类的声明方法和基本用法。
- 掌握**Java**泛型的声明方法和基本用法，理解泛型限定和泛型继承的原理。
- 掌握反射和动态代理的机制。
- 掌握**Java**元注解的用法，掌握注解的自定义方法和反射解析方法。
- 掌握**Lambda**表达式的本质和基本用法。



主要内容

- **A.1 Java枚举和泛型**
- **A.2 Java反射和代理**
- **A.3 Java注解和Lambda表达式**



A.1 Java枚举和泛型

- A.1.1 Java枚举

- A.1.2 Java泛型



A.1.1 Java枚举

- 自Java5推出枚举类型，用**enum**关键字声明一个枚举类，表明该类仅有有限个其所枚举的几个实例对象。
- 枚举类声明中可包含以下内容：
 - ◇ 枚举值、属性、构造函数、方法
 - 构造函数只能被**package-private(default)**或者**private**修饰，用于内部调用



枚举类中的方法

■ 所有的**enum**类型都是**Enum**的子类(但不需写**extends**), 因而继承了相应方法

✧ **ordinal()**: 返回枚举值所在的索引位置, 从**0**开始

✧ **compareTo()**: 比较两个枚举值的索引位置大小

✧ **toString()**: 返回枚举值的字符串表示 (默认行为是输出枚举值声明的名称)

✧ **valueOf()**: 将字符串初始化为枚举对象

✧ **values()**: 返回所有的枚举值 (不是枚举值的字符串表示)



枚举类示例

```
import java.util.Arrays;

public enum SexEnum{

    //枚举值
    MAN(1, "男"), WOMAN(2, "女");

    //属性
    private Integer value;
    private String desc;

    //构造函数
    SexEnum(Integer value, String desc) {
        this.value = value;
        this.desc = desc;
    }

    //方法
    public Integer getValue() { return this.value; }

    @Override
    //返回枚举值的字符串表示
    public String toString() { return this.desc; }
```

枚举类示例

```
public static void main(String[] args) {  
    SexEnum s1 = SexEnum.MAN;  
    //将字符串初始化为枚举对象  
    SexEnum s2 = SexEnum.valueOf("MAN");  
    SexEnum s3 = SexEnum.WOMAN;  
    System.out.println(s1 == s2); //true  
    //返回枚举值所在的索引位置, 从0开始  
    System.out.println(s1.ordinal()); // 0  
    System.out.println(s3.ordinal()); // 1  
    //比较两个枚举值的索引位置大小  
    System.out.println(s1.compareTo(s3)); //-1: MAN<WOMAN  
    //返回所有的枚举值  
    SexEnum[] enums = SexEnum.values();  
    //遍历所有的枚举值  
    for(SexEnum e:enums){ System.out.println(e); } // 男、女  
    Arrays.asList(enums).forEach(System.out::println); //男、女  
}
```


运行结果

```
Run: SexEnum x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
true
0
1
-1
男
女
男
女
```



A.1.2 Java泛型

- 1.泛型入门
- 2.泛型限定
- 3.泛型继承



1.泛型入门

- 泛型（**Generic Programming**）是JDK1.5引进的新特性，其作用是让编写的代码可以被很多不同类型的对象所重用。
- 泛型的本质：类型参数化，从而避免类型转换，实现代码可复用。



泛型的声明和应用

■ 声明泛型类和泛型接口：在类名或接口名后加<T>

✧ 多个字母表示多个类型变量，如<T, U>等

✧ 类型变量可以修饰成员变量/局部变量/参数/返回值

✧ T也可以再是一个泛型类

■ 声明泛型方法：在方法的修饰符后，返回类型前加<T>



泛型类声明示例

```
import java.io.Serializable;

//泛型接口

public interface Calculator<T extends Serializable> {

    public T add(T operand1, T operand2);

}
```

```
import java.io.Serializable;

//泛型类

public class Interval<T> implements Serializable {

    private T lower, upper;

    public Interval(T lower, T upper) {

        this.lower = lower;

        this.upper = upper;

    }

    //泛型方法

    public static <U> U getMiddle(U... a) {

        return a[a.length / 2];

    }

    //省略get、set方法

}
```

泛型类应用示例

```
public class IntervalCalculator implements Calculator<Interval<Integer>> {  
    public Interval<Integer> add(Interval<Integer> operand1, Interval<Integer> operand2) {  
        int lower = operand1.getLower() + operand2.getLower();  
        int upper = operand1.getUpper() + operand2.getUpper();  
        return new Interval<>(lower, upper);  
    }  
  
    public static void main(String[] args) {  
        Calculator<Interval<Integer>> c = new IntervalCalculator();  
        Interval<Integer> i1 = new Interval<>(1, 2);  
        Interval<Integer> i2 = new Interval<>(3, 4);  
        Interval<Integer> i3 = c.add(i1, i2);  
        Integer low = i3.getLower();  
        Integer upper = i3.getUpper();  
        System.out.println "[" + low + "," + upper + "];"  
        System.out.println(Interval.<String>getMiddle("L", "M", null));  
    }  
}
```


2.泛型限定

- 泛型限定是指在声明或应用泛型时限定类型变量只能取特定值。
- 泛型限定形式：
 - ◇ **<T extends Class & Interfaces>**: 声明T必须是给定类**Class**的子类（包括自身）或接口**Interfaces**的实现类。
 - **extends**后可以有多个接口，但只能一个类，且类必须排第一位
 - ◇ **<? extends Class>**: 应用泛型时限定类型变量只能取值为给定类**Class**的子类（包括自身）。
 - ◇ **<? super Class>**: 应用泛型时限定类型变量只能取值为给定类**Class**的超类（包括自身）。
 - ◇ **<?>**: 应用泛型时不限定类型变量取值。



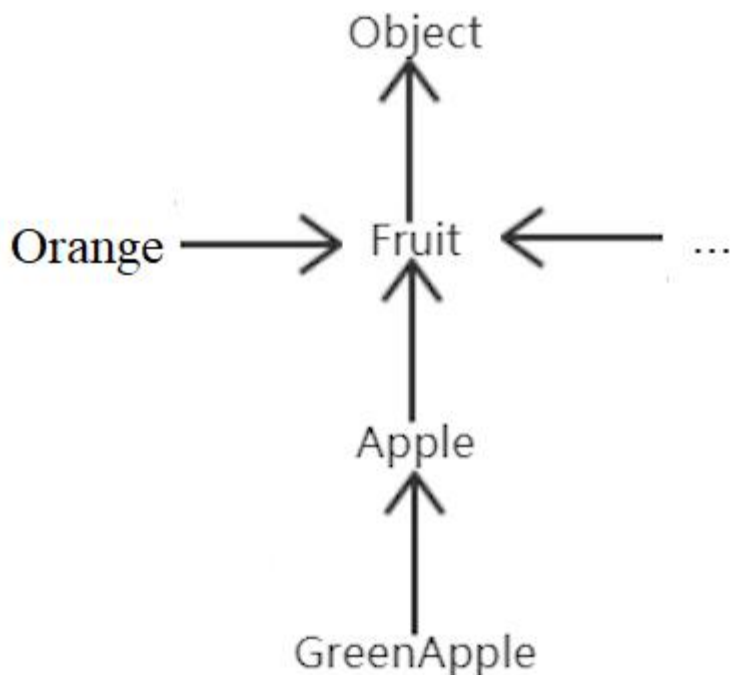
泛型PECS原则

■ 泛型PECS原则：Producer Extends, Consumer Super

- ✧ 对于泛型类 $X<T>$ ，如果只能从该泛型类`get`类型 T 的数据，而不能向该泛型类`set`类型 T 的数据时，则使用`<? extends Class>`限定形式（泛型类是生产者，往外输出东西）
- ✧ 对于泛型类 $X<T>$ ，如果只能向该泛型类`set`类型 T 的数据，而不能从该泛型类`get`类型 T 的数据时，则使用`<? super Class>`限定形式（泛型类是消费者，往内增加东西）
- ✧ 如果既想`set`又想`get`类型 T 的数据，那就不用通配符

示例

- 设Pair<T>为一个泛型类，其有**first**、**second**两个T类型成员变量，并有以下继承关系的类：



示例

```
public static void CS() { //Consumer Super
    Pair<? super Apple> fruitPair = new Pair<Fruit>();
    fruitPair.setFirst(new Apple(5));           //Apple可转型到Apple超类Fruit
    fruitPair.setSecond(new GreenApple(5)); //GreenApple可转型到Apple超类Fruit
    //fruitPair.setSecond(new Object());       //Object无法转型到Apple超类Fruit，故编译报错
    Fruit fruit = (Fruit)fruitPair.getFirst(); //?具有不确定性，出来的对象类型只能是Object，故需强制转换
}

public static void PE() { //Producer Extends
    Pair<? extends Fruit> fruitPair = new Pair<Apple>(new Apple(3), new GreenApple(4));
    Fruit first = fruitPair.getFirst();         //Apple可以转型到Fruit
    Fruit second = fruitPair.getSecond();       //GreenApple可以转型到Fruit
    //fruitPair.setFirst(new Apple(5));         //?具有不确定性，故编译错误
}

public static void unrestrict() { //无限定
    Pair<?> fruitPair = new Pair<Apple>(new Apple(3), new GreenApple(4));
    Fruit first = (Fruit)fruitPair.getFirst(); //?具有不确定性，出来的对象类型只能是Object，故需强制转换
    Fruit second = (Fruit)fruitPair.getSecond(); //?具有不确定性，出来的对象类型只能是Object，故需强制转换
    //fruitPair.setFirst(new Apple(5));         //?具有不确定性，故编译错误
}
```

3.泛型继承

- **Pair<Class1>**和**Pair<Class2>**没有任何关系，无论**Class1**和**Class2**之间是什么关系。
- 泛型类可以扩展或实现其他的类，如**ArrayList<T>**实现**List<T>**。
 - ✧ **List<Orange> oranges = new ArrayList<Orange>()**



A.2 Java反射和代理

- A.2.1 反射机制

- A.2.2 动态代理



A.2.1 反射机制

■ **Java**反射机制的核心是在程序运行时动态加载类并获取类的详细信息，从而操作类或对象的属性和方法。

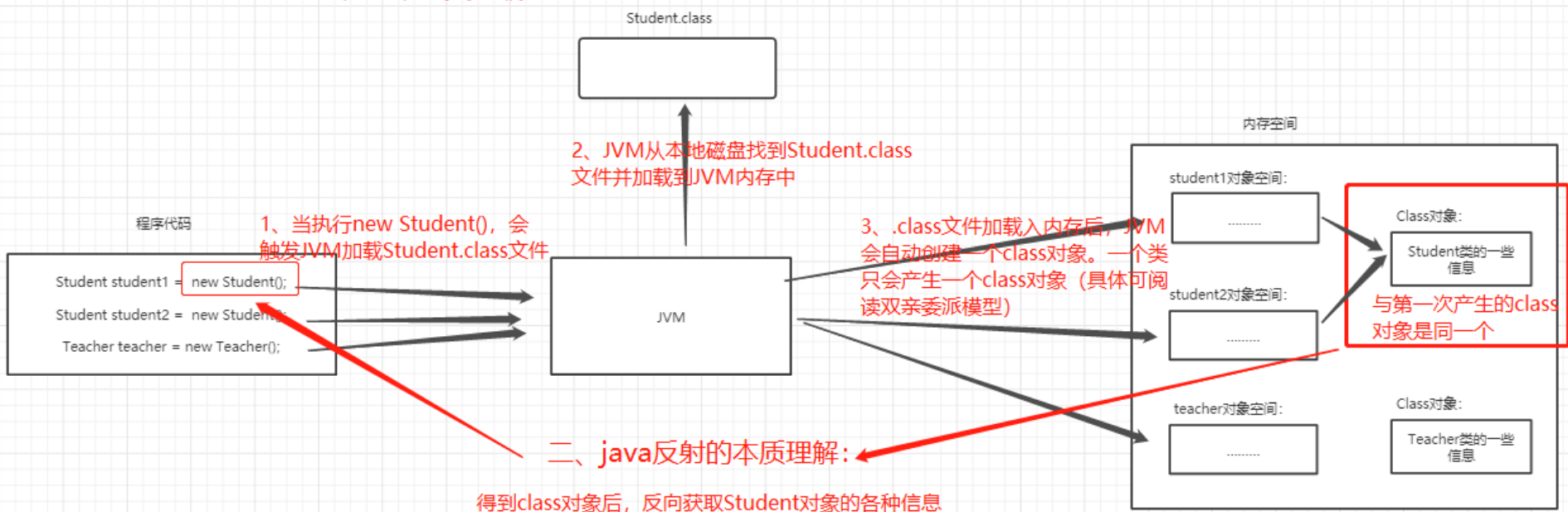
✧本质是JVM得到**class**对象之后，再通过**class**对象进行反编译，从而获取对象的各种信息。

■ **Spring**框架利用**Java**反射机制，使用配置文件把框架的各个组件串联起来，实现代码和配置文件的相互独立。



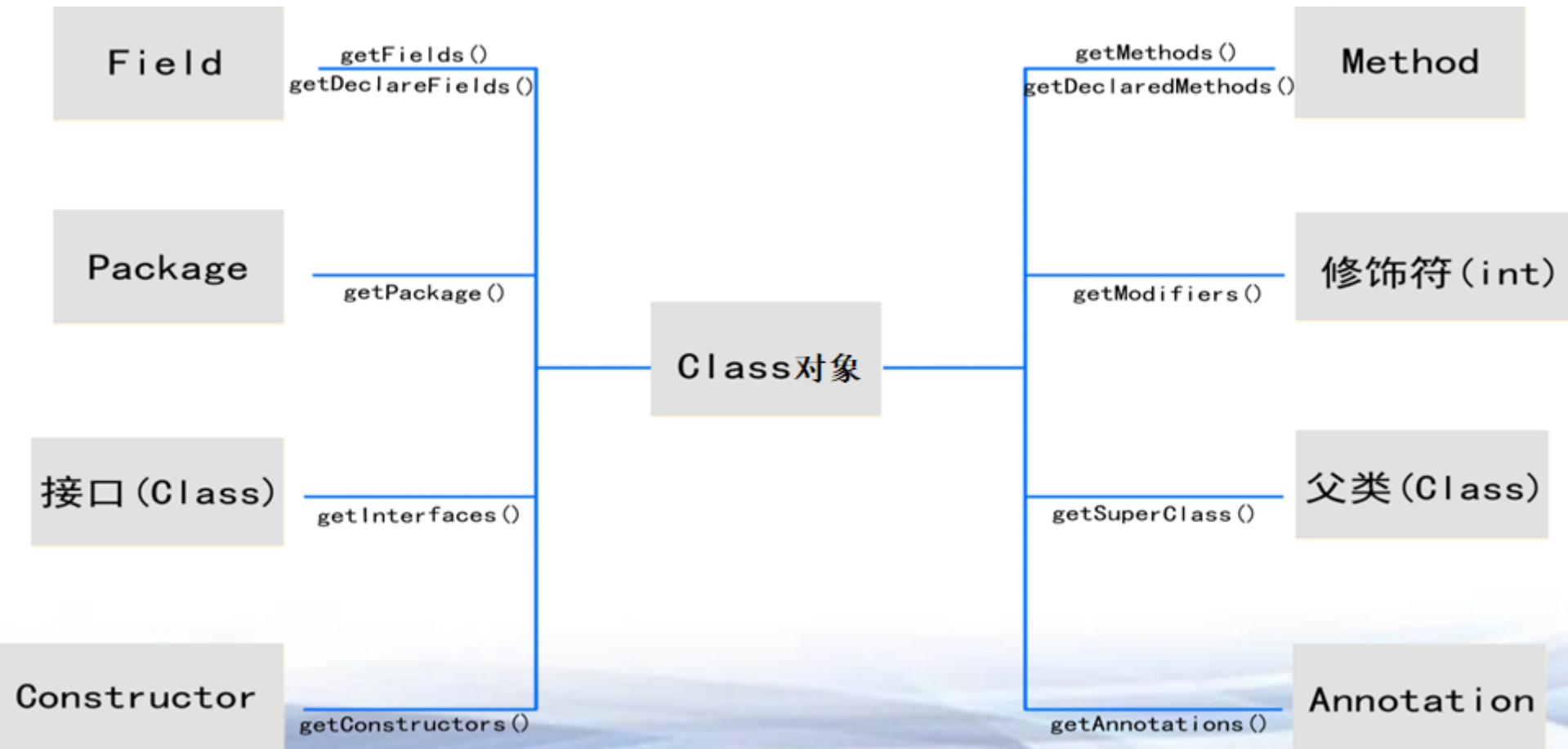
反射的原理

一、正常的类加载过程



JVM会为每个类创建一个**Class对象**（`java.lang.Class`类对象），**Class对象**即该类的类型标识(**Runtime Type Identification**)，通过该**Class对象**就可以获取这个类的信息，然后通过使用**java.lang.reflect**包下的**API**以达到各种动态需求。

可获取的类信息



反射机制案例

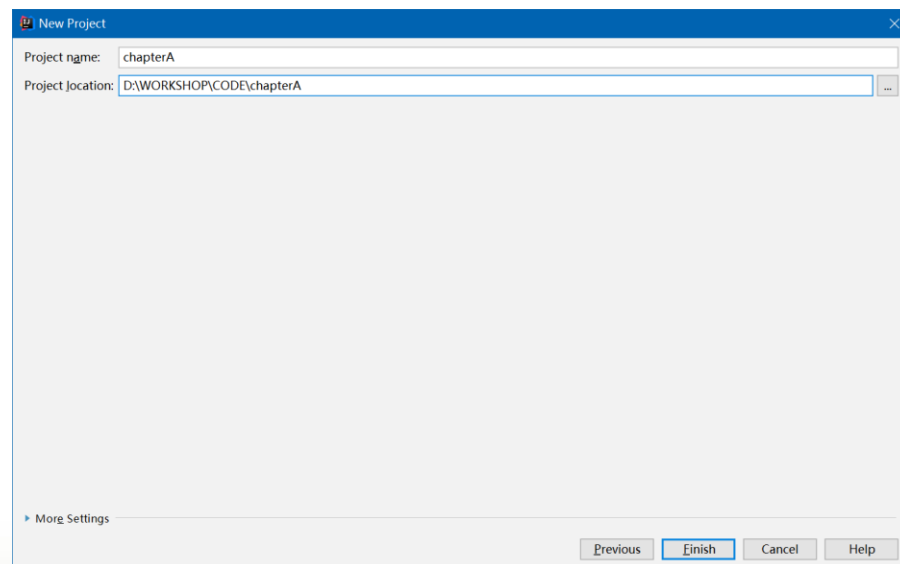
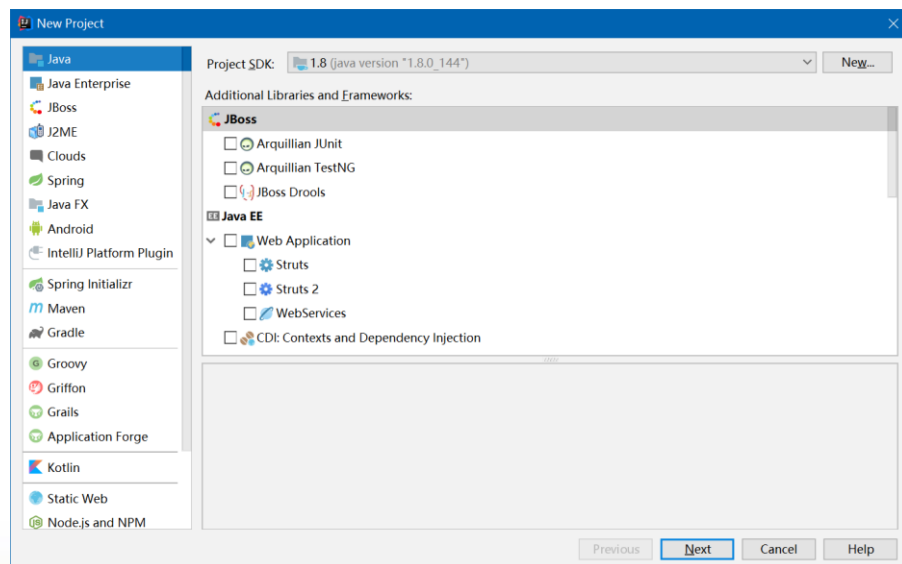
■ 搭建步骤:

- ✧①创建**Java**工程
- ✧②创建目标类
- ✧③创建反射处理类
- ✧④效果测试



①创建Java工程

■在IDEA中，创建一个名为chapterA的普通Java项目。



②创建目标类

- 在chapterA项目的src目录下，创建一个com.scst.service包，并在包中创建接口UserService及其实现类UserServiceImpl



UserService接口类

```
public interface UserService {  
    void sayHi();  
    String sayHello();  
}
```



UserServiceImpl实现类

//目标类

```
public class UserServiceImpl implements UserService {  
    private String hello = "hello";  
    public void sayHi() {  
        System.out.println("hi");  
    }  
    public String sayHello() {  
        System.out.println(this.hello);  
        return this.hello;  
    }  
}
```

③创建反射处理类

- 在chapterA项目的src目录下，创建一个**com.scst.reflect**包，并在包中创建一个反射处理器类**ReflectHandler**，通过反射机制实现对目标类信息（方法和属性）的获取以及动态执行和修改。



ReflectHandler类

```
import com.scst.service.UserService;  
import com.scst.service.UserServiceImpl;  
import java.lang.reflect.Field;  
import java.lang.reflect.Method;  
  
//反射处理器类  
public class ReflectHandler {
```



ReflectHandler类

//运行时获取目标对象的类信息（方法和属性）

```
public void getMethodsandFields(Object target) throws Exception{
```

```
    //1.获取目标类（Class对象）
```

```
    Class<?> clazz =target.getClass();
```

```
    //2.获取目标类声明的所有方法并调用
```

```
    Method[] methods = clazz.getDeclaredMethods();
```

```
    for (Method m:methods){
```

```
        System.out.println(m);
```

```
        m.invoke(target,null);
```

```
    }
```

```
    //3.获取目标类声明的所有属性
```

```
    Field[] fields = clazz.getDeclaredFields();
```

```
    for (Field f:fields){ System.out.println(f); }
```

```
}
```

ReflectHandler类

//运行时改变目标对象私有变量的值

```
public void setField(Object target, String fieldName ,Object value) throws Exception{
```

```
    //1.获取目标类（Class对象）
```

```
    Class<?> clazz =target.getClass();
```

```
    //2.获取目标类的指定字段并修改其值
```

```
    Field field = clazz.getDeclaredField(fieldName);
```

```
    field.setAccessible(true);
```

```
    field.set(target,value);
```

```
}
```



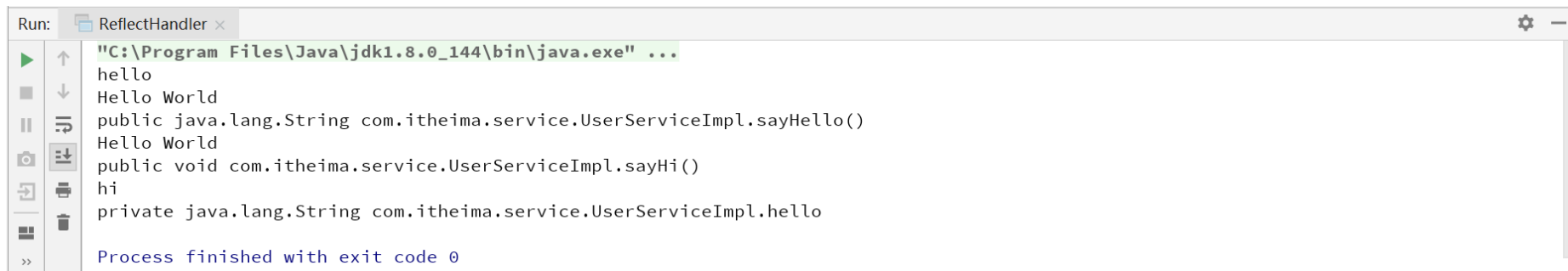
ReflectHandler类

```
public static void main(String[] args) throws Exception{  
    UserService userService = new UserServiceImpl();  
    userService.sayHello();  
    ReflectHandler handler = new ReflectHandler();  
    handler.setField(userService,"hello","Hello World");  
    userService.sayHello();  
    handler.getMethodsandFields(userService);  
}  
}
```



④效果测试

■ 执行ReflectHandler类中的main()方法，观察效果。



```
Run: ReflectHandler x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
hello
Hello World
public java.lang.String com.itheima.service.UserServiceImpl.sayHello()
Hello World
public void com.itheima.service.UserServiceImpl.sayHi()
hi
private java.lang.String com.itheima.service.UserServiceImpl.hello
Process finished with exit code 0
```



A.2.2 动态代理

- 动态代理就是在程序运行期，创建目标对象的代理对象，并对目标对象中的方法进行功能性增强的一种技术。
- 有了动态代理的技术，那么就可以在不修改方法源码的情况下，增强被代理对象的方法的功能，在方法执行前后做任何你想做的事情。
 - ✧ 从而可以对同一类型**Java**对象进行统一配置和管理，提升开发效率。



动态代理实现

■ 有两类动态代理实现方法：

✧ ① **JDK** 动态代理

✧ ② **CGLIB** 代理



①JDK动态代理实现

- 在chapterA项目的src目录下，创建一个com.scst.proxy包，并在包中创建代理处理器类JdkProxyHandler，通过该代理处理器类创建目标对象的代理对象，并增强目标方法的功能。



JdkProxyHandler类

```
import com.scst.service.UserService;  
import com.scst.service.UserServiceImpl;  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
  
//代理处理器  
public class JdkProxyHandler implements InvocationHandler {  
    //声明目标对象  
    private Object target;
```



JdkProxyHandler类

//创建目标对象的代理对象

```
public Object createProxy(Object target) {  
    this.target = target;  
    //获取目标类（Class对象）  
    Class<?> clazz =target.getClass();  
    //获取目标类的类加载器  
    ClassLoader classLoader = clazz.getClassLoader();  
    //获取目标类实现的所有接口  
    Class<?>[] interfaces = clazz.getInterfaces();  
    //创建代理对象并返回  
    return Proxy.newProxyInstance(classLoader,interfaces,this);  
}
```

JdkProxyHandler类

//程序执行目标方法时被调用，在此增强目标方法功能

//proxy JDK生成的代理对象

//method 被反射的方法

//args 被反射方法的参数数组

@Override

public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {

 System.out.println("在此实现目标方法执行前增强");

 //执行目标方法

 Object obj = method.invoke(target, args);

 System.out.println("在此实现目标方法执行后增强");

 //返回目标方法返回的结果

 return obj;

}

JdkProxyHandler类

```
public static void main(String[] args){  
    JdkProxyHandler handler = new JdkProxyHandler();  
    UserService userService = (UserService)handler.createProxy(new UserServiceImpl());  
    System.out.println(userService.sayHello());  
    System.out.println("-----");  
    userService.sayHi();  
}  
}
```

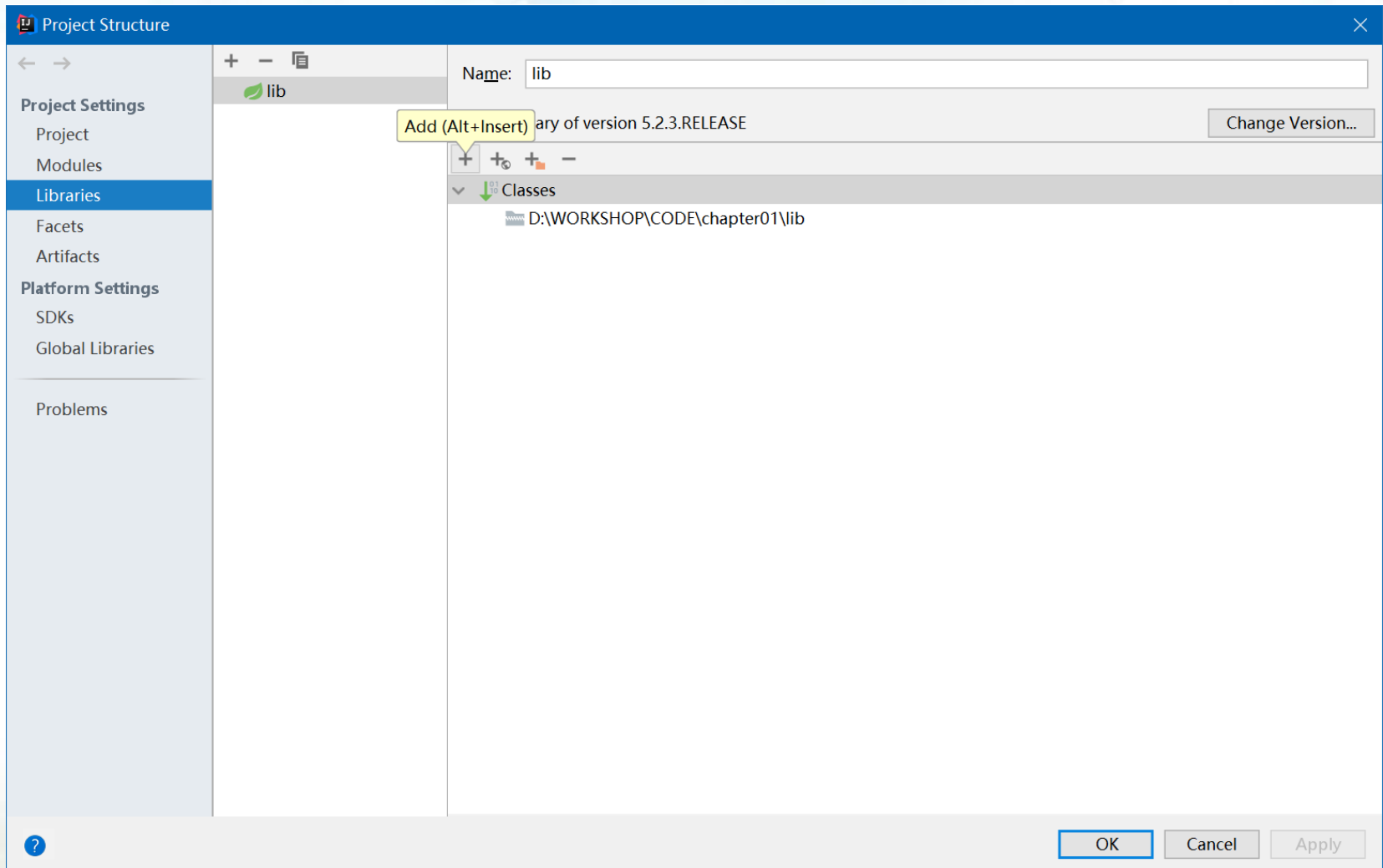


②CGLIB代理实现

- 在chapterA项目的com.scst.proxy包中创建一个代理处理器类CglibProxyHandler，通过该代理处理器类创建目标对象的代理对象，并增强目标方法的功能。
- 实现该代理处理器类需事先引入以下类库：
 - ✧ spring-core-5.2.3.RELEASE.jar



引入类库



CglibProxyHandler类

```
import java.lang.reflect.Method;
import com.scst.service.UserService;
import com.scst.service.UserServiceImpl;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

//代理处理器
public class CglibProxyHandler implements MethodInterceptor {
```



CglibProxyHandler类

//创建目标对象的代理对象

```
public Object createProxy(Object target) {
```

```
    //创建CGLIB核心类实例
```

```
    Enhancer enhancer = new Enhancer();
```

```
    //设置需要增强的目标类（Class对象），目标类为代理对象的父类
```

```
    enhancer.setSuperclass(target.getClass());
```

```
    //设置回调对象，this代表本对象，程序执行目标方法时将会回调本对象的
```

```
    intercept()方法
```

```
    enhancer.setCallback(this);
```

```
    //创建代理对象并返回
```

```
    return enhancer.create();
```

```
}
```

CglibProxyHandler类

//程序执行目标方法时被调用，在此增强目标方法功能

//proxy CGLIB生成的代理对象

//method 被拦截的方法

//args 被拦截方法的参数数组

//methodProxy 方法的代理对象，用于执行父类的方法

@Override

```
public Object intercept(Object proxy, Method method, Object[] args, MethodProxy  
methodProxy) throws Throwable {
```

```
    System.out.println("在此实现目标方法执行前增强");
```

```
    //通过方法代理对象执行目标方法（proxy的父类方法）
```

```
    Object obj = methodProxy.invokeSuper(proxy, args);
```

```
    System.out.println("在此实现目标方法执行后增强");
```

```
    //返回目标方法返回的结果
```

```
    return obj;
```

```
}
```

CglibProxyHandler类

```
public static void main(String[] args){  
    CglibProxyHandler handler = new CglibProxyHandler();  
    UserService userService = (UserService)handler.createProxy(new UserServiceImpl());  
    System.out.println(userService.sayHello());  
    System.out.println("-----");  
    userService.sayHi();  
}  
}
```



③效果测试

- 分别执行JdkProxyHandler类和CglibProxyHandler类中的main()方法，观察效果。



```
Run: JdkProxyHandler x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
在此实现目标方法执行前增强
hello
在此实现目标方法执行后增强
hello
-----
在此实现目标方法执行前增强
hi
在此实现目标方法执行后增强

Run: CglibProxyHandler x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
在此实现目标方法执行前增强
hello
在此实现目标方法执行后增强
hello
-----
在此实现目标方法执行前增强
hi
在此实现目标方法执行后增强
```

A.3 Java注解和Lambda表达式

■ A.3.1 Java注解

■ A.3.2 Lambda表达式



A.3.1 Java注解

- 1.Java注解入门
- 2.Java元注解
- 3.注解的解析



1.Java注解入门

■ 注解（**Annotation**）自**JDK 1.5**引入，是以**@**开头的表达式，是位于源码(代码+注释+注解)，自带一定功能的，使用其他工具进行处理的标签。

✧它提供了额外的程序，增加了自由度，迅速被广大框架所接受。

■ 例如：

✧ **@Override**标记该方法覆盖父类的方法

✧ **@Deprecated**标记该元素是属于过时的

✧ **@SuppressWarnings**用来压制各种不同类型的警告信息



注解的声明和使用

■ 声明一个注解是通过**注解接口**来实现的。

✧ 注解接口是指用 **@interface** 定义的接口。

✧ 注解接口中的方法对应于注解使用中括号里的属性及其取值。

✧ 注解接口不需要实现。



示例

```
public @interface myAnnotation {  
    String value() default "My annotation test";  
    String user();  
}
```

```
@myAnnotation(value = "Another test", user = "admin")  
public class TestAnnotation {  
}
```



2.Java元注解

- 所谓元注解是指修饰注解的注解，用于限定或指明注解的属性。
- 在Java预定义的注解中，包括5个元注解：
 - ✧ ① @Target
 - ✧ ② @Retention
 - ✧ ③ @Inherited
 - ✧ ④ @Documented
 - ✧ ⑤ @Repeatable



① @Target

- 用于限定目标注解作用于什么位置，如：
@Target({ElementType.METHOD})
- **ElementType**可取值如下：
 - ✧ **ElementType.ANNOTATION_TYPE**: 注解
 - ✧ **ElementType.CONSTRUCTOR**: 构造方法
 - ✧ **ElementType.FIELD**: 域
 - ✧ **ElementType.LOCAL_VARIABLE**: 局部变量
 - ✧ **ElementType.METHOD**: 方法
 - ✧ **ElementType.PACKAGE**: 包
 - ✧ **ElementType.PARAMETER**: 方法或构造方法的参数
 - ✧ **ElementType.TYPE**: 类、接口



② @Retention

■ 用来指明目标注解的保留策略，如：
@Retention(RetentionPolicy.CLASS)

■ **RetentionPolicy**可取值如下：

✧ **RetentionPolicy.SOURCE**：注解仅存在于源码，不保留到class文件

✧ **RetentionPolicy.CLASS**：这是默认的注解保留策略。注解存在于.class文件，但是不能被JVM加载。

✧ **RetentionPolicy.RUNTIME**：注解存在于.class文件，且可以被JVM运行时访问到。



③ @Inherited

- 用于让一个类和它的子类都包含某个注解。

```
import java.lang.annotation.Inherited
```

```
@Inherited  
public @interface MyAnnotation {  
  
}
```

```
@MyAnnotation  
public class MySuperClass { ... }
```

```
public class MySubClass extends MySuperClass { ... }
```

④ @Documented

■ 该注解用于让目标注解显示在归档中。

✧ 当某个目标注解使用**Documented**注解后，则使用该目标注解的类在使用**Javadoc**归档后，就会显示该条目标注解。



⑤ @Repeatable

- 该注解自**JDK1.8**引入，表示目标注解在同一位置可以重复被应用，但目标注解和容器注解需同时被定义。



示例

```
@Repeatable(RepeatableAnnotations.class)
```

```
public @interface RepeatableAnnotation {  
    int a() default 0;  
    int b() default 0;  
    int c() default 0;  
}
```

```
@Target({ElementType.METHOD})
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
```

```
public @interface RepeatableAnnotations {  
    RepeatableAnnotation[] value();  
}
```

```
package com.scst.annotation;
```

```
public class Student {  
    @RepeatableAnnotation(a=1,b=2,c=3)  
    @RepeatableAnnotation(a=1,b=2,c=4)  
    public static void add(int x, int y, int z){...}  
}
```

3.注解的解析

■ 根据RetentionPolicy的不同，注解分3种类型：

✧ **SOURCE**：注解在源码，不在class文件

- 注解只有在源码级别进行注解处理
- Java提供**注解处理器**来解析带注解的源码，产生新的最终不带注解的源码文件

✧ **CLASS**：在源码和class文件中，但JVM不加载

- 只能采用**字节码工具**进行特殊处理，如ASM工具（<https://asm.ow2.io>）

✧ **RUNTIME**：注解在源码和class文件中，JVM加载

- 可用**反射**解析注解
- 通过反射找出加在类、方法、域、参数等上的注解，便知道需要对这些类、方法等做什么具体工作



RUNTIME注解解析API

- **Class.getAnnotations()**
- **Class.isAnnotation()**
- **Class.isAnnotationPresent(Class annotationClass)**
- **Method.getAnnotations()**
- **Method.isAnnotationPresent(Class annotationClass)**
- **Field.getAnnotations()**
- **Field.isAnnotationPresent(Class annotationClass)**
- **Constructor.getAnnotations()**
- **Constructor.isAnnotationPresent(Class annotationClass)**



示例

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

public class Main {

    public static void main(String[] a) throws Exception {

        String className = "com.scst.annotation.Student";

        for (Method m : Class.forName(className).getMethods()) {

            if (m.isAnnotationPresent(RepeatableAnnotations.class)) {

                RepeatableAnnotation[] annos = m.getAnnotationsByType(RepeatableAnnotation.class);

                for (RepeatableAnnotation anno : annos) {

                    System.out.println(anno.a() + "," + anno.b() + "," + anno.c());

                    try {

                        m.invoke(null, anno.a(), anno.b(), anno.c());

                    } catch (Throwable ex) {

                        System.out.printf("Test %s failed: %s %n", m, ex.getCause());

                    }

                }

            }

        }

    }

}
```

RUNTIME注解调用路线

■ **Java**自动为注解产生一个代理类。

✧ 这个代理类包括一个

AnnotationInvocationHandler成员变量。

✧ **AnnotationInvocationHandler**有一个**Map**的成员变量，用来存储注解的所有属性赋值。

■ 在程序中，调用注解接口的方法，将会被代理类接管，然后根据方法名字，到**Map**里面拿相应的**Value**并返回。



A.3.2 Lambda表达式

- 1.基本语法
- 2.Lambda表达式作用域
- 3.函数式接口
- 4.方法引用



1.基本语法

■ lambda表达式的基本语法如下：

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

- 意思是：接受parameters参数，执行expression / statements并返回，也可以不接受任何参数。

■ lambda表达式是函数式接口抽象方法的一个匿名实现。

◇被赋值后，可以看作是一个函数式接口的实例(对象)

- 但Lambda表达式没有存储目标类型(target type)的信息（即一个Lambda表达式可推理适配不同函数式接口类型）。
- Lambda表达式可以作为实参传递给其他方法的形参。

示例

```
public interface Adder {  
    public int selfAdd(int x) ;  
}
```

```
Adder c = x -> {  
    x++;  
    return x;  
};  
System.out.println(c.selfAdd(23));
```

2.Lambda表达式作用域

■ **Lambda表达式和匿名内部类/局部内部类一样，可以捕获变量(capture variables)，即访问外部嵌套块的变量。**

✧但是该变量要求是**final**或者是**effectively final**的

✧在**Lambda表达式**中，也不可声明与（外部嵌套块）局部变量同名的参数或者局部变量。

✧**Lambda表达式中的this**，就是创建这个表达式的方法的**this**参数，即执行该方法的**this**对象。



示例

```
import java.util.function.Consumer;

public class LambdaScopeTest {
    public int x = 0;
    public static void main(String... args) {
        LambdaScopeTest st = new LambdaScopeTest();
        FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```



示例

```
class FirstLevel {  
    public int x = 1;  
    void methodInFirstLevel(int x) {  
        int z = 99; // z not x,y  
        Consumer<Integer> myConsumer = (y) -> {  
            System.out.println("x = " + x); // 23, x must be final or effectively final  
            System.out.println("y = " + y); //23  
            System.out.println("this.x = " + this.x); // 1  
            System.out.println("LambdaScopeTest.this.x = " + LambdaScopeTest.this.x); // 0  
        };  
        myConsumer.accept(23);  
        System.out.println("z = " + z); // 99  
    }  
}
```

3.函数式接口

- 函数式接口是只包含一个抽象（未实现）方法的接口。
 - ✧ 当然从JDK8开始还可以包括default方法、static方法、private方法
- 函数式接口可用@FunctionalInterface注解来标注（非必须），用于编译器检查。
- 系统自带一些函数式接口，涵盖了大部分常用的功能，可以重复使用。
 - ✧ 位于java.util.function包中



常用的系统自带函数式接口

函数式接口	抽象方法	作用
Predicate<T>	boolean test(T t)	接收一个参数，返回一个布尔值
Consumer<T>	void accept(T t)	接收一个参数，做操作，无返回
Supplier<T>	T get()	无输入参数，返回一个数据
Function<T,R>	R apply(T t)	接收一个参数，返回一个数据



4.方法引用

■ 方法引用（**Method Reference**）是访问某方法的**Lambda**表达式的一种简洁形式，其通过“**::**”操作符访问类的方法。

✧ 如：**Math::pow**相当于**(x,y)->Math.pow(x,y)**



几种方法引用形式

■ object::instanceMethod

✧ 如: `System.out::println` 相当于 `(x)->System.out.println(x)`

✧ 如: `this::instanceMethod`、`super::instanceMethod`

■ Class::staticMethod

✧ 如: `Math::abs` 相当于 `(x)->Math.abs(x)`

■ Class::instanceMethod

✧ 如: `String::compareToIgnoreCase` 相当于 `(x,y)->x.equalsIgnoreCase(y)`

■ Class::new

✧ 调用某类构造函数, 支持单个对象构建

■ Class[]::new

✧ 调用某类构造函数, 支持数组对象构建



示例

```
public interface Iterable<T> {  
    /**  
     */  
    Iterator<T> iterator();  
  
    /**  
     */  
    default void forEach(Consumer<? super T> action) {  
        Objects.requireNonNull(action);  
        for (T t : this) {  
            action.accept(t);  
        }  
    }  
}
```

```
SexEnum[] enums = SexEnum.values();
```

```
Arrays.asList(enums).forEach(System.out::println);
```

本章小结

■ 本章具体讲解了：

✧ **A.1 Java枚举和泛型**

✧ **A.2 Java反射和代理**

✧ **A.3 Java注解和Lambda表达式**



