

day25-基础加强

学习目标

- 1. 能够通过反射技术获取Class字节码对象
- 2. 能够通过反射技术获取构造方法对象，并创建对象。
- 3. 能够通过反射获取成员方法对象，并且调用方法。
- 4. 能够通过反射获取属性对象，并且能够给对象的属性赋值和取值。
- 5. 能够使用Beanutils常用方法操作JavaBean对象
- 6. 能够说出注解的作用
- 7. 能够自定义注解和使用注解
- 8. 能够说出常用的元注解及其作用
- 9. 能够解析注解并获取注解中的数据
- 10. 能够完成注解的MyTest案例

一,反射【重点】

1.概述

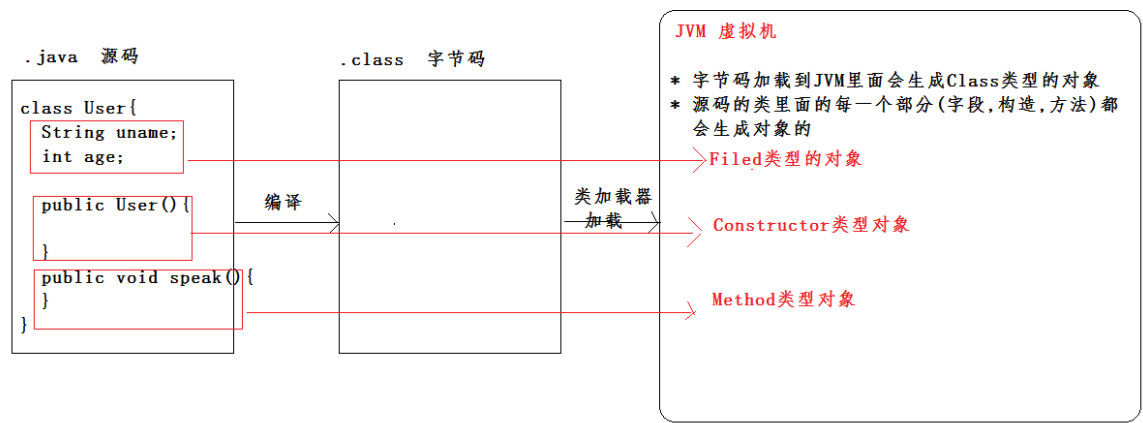
1.1什么是反射

反射是一种机制/功能，利用该机制/功能可以在程序运行过程中对类进行解剖并操作类中的构造方法，成员方法，成员属性。

反射乃框架之灵魂 JavaEE

反射就是把Java的各种成分(字段,方法)映射成相应的Java类.

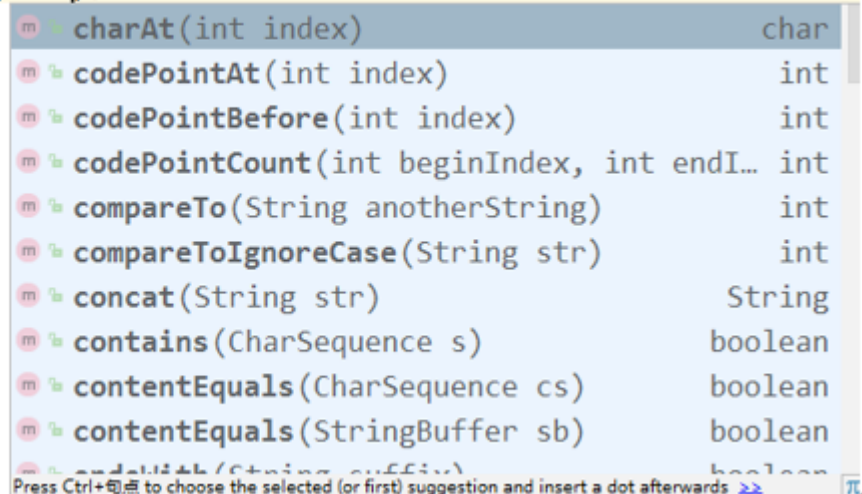
反射就是把Java的各种成分(字段,方法)映射成相应的Java类.



1.2反射的应用场景

- 1. 开发工具中写代码时的提示

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



开发工具之所以能够把该对象的方法和属性展示出来就使用利用了反射机制对该对象所有类进行了解剖获取到了类中的所有方法和属性信息，这是反射在IDE中的一个使用场景。

2. 各种框架的设计



以上三个图标上面的名字就是Java的三大框架，简称SSH。

这三大框架的内部实现也大量使用到了反射机制，所以要想学好这些框架，则必须要求对反射机制熟练了

2. 获得Class字节码对象的三种方法

- 对象.getClass()
- 使用Class类的forName("类的全限定名") 最常见的方式
- 使用“类名.class”

3. 获取Class对象的信息 通过字节码创建对象(要求: 依赖的是无参构造)

- 获取简单类名

```
String getSimpleName(); 获取简单类名，只是类名，没有包
```

- 获取完整类名

```
String getName(); 获取完整类名，包含包名 + 类名
```

- 创建对象(依赖无参构造)

```
T newInstance(); 创建此 Class对象所表示的类的一个新实例。
```

示例代码

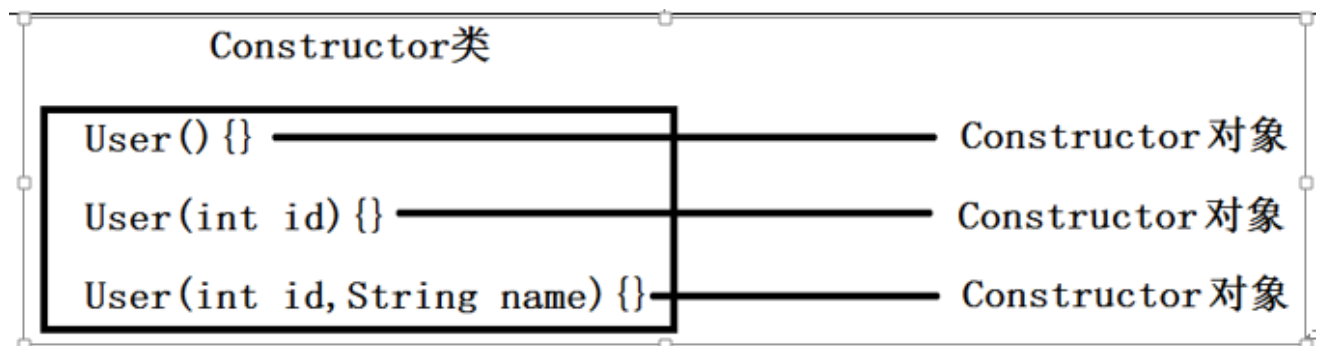
```
//b. 字节码对象里面常见的方法
public void fun02() throws Exception {
    //1. Class.forName("类的全限定名");
    //Class clazz = Class.forName("com.itheima.bean.Student");
    Class clazz = Student.class;
    //2. 获得类名(简单类名) Student
    System.out.println(clazz.getSimpleName());
    //3. 获得完整的类名 com.itheima.bean.Student
    System.out.println(clazz.getName());

    //4. 通过字节码创建对象(要求: 依赖的是无参构造)-- new Student();
    Student student = (Student) clazz.newInstance();
    student.speak();
}
```

4.构造函数的反射

4.1概述

Constructor是构造方法类，类中的每一个构造方法都是Constructor的对象，通过Constructor对象可以实例化对象。



4.2 Class类中与Constructor相关方法 仅仅只能获得构造方法

```
clazz.getDeclaredConstructors(); //获得所有的构造方法(包含私有的)
clazz.getConstructor(Class... paramType 传入参数类型的字节码对象数组); //获得特定的构造方法
constructor.newInstance(Object...params 传入参数值);// 根据构造方法创建对象
```

示例代码

```
//c. 反射构造方法
public void fun03() throws Exception {
    //1. Class.forName("类的全限定名");
    Class clazz = Class.forName("com.itheima.bean.Student");

    //2. 获得所有的构造方法(包含私有的)
    Constructor[] constructors = clazz.getDeclaredConstructors();

    System.out.println(constructors.length);
}
```

```
//3. 反射获得某一个特定的构造方法
//eg: public Student(String name, int age, String sex)
Constructor constructor = clazz.getConstructor(String.class, int.class, String.class);
Student student = (Student) constructor.newInstance("张三", 18, "男");//根据构造方法创建一个
对象
System.out.println(student.getAge());

}
```

5.属性的反射

5.1概述

Field是属性类，类中的每一个属性都是Field的对象，通过Field对象可以给对应的属性赋值和取值。

5.2 Class类中与Field相关方法

1. Field[] `getFields()`
获取所有的public修饰的属性对象，返回数组
2. Field[] `getDeclaredFields()`
获取所有的属性对象，包括private修饰的，返回数组
3. Field `getField(String name)`
根据属性名获得属性对象，只能获取public修饰的
4. Field `getDeclaredField(String name)`
根据属性名获得属性对象，包括private修饰的

5.3Field类中常用方法

`set(obj, value)`;通用方法都是给对象obj的属性设置使用
`get(obj)`;通用方法是获取对象obj对应的属性值的
`void setAccessible(true)`;暴力反射，设置为可以直接访问私有类型的属性

示例代码

```
//d. 反射字段
public void fun04() throws Exception {
    Student student = new Student("张三", 18, "男");

    //1. Class.forName("类的全限定名");
    Class clazz = Class.forName("com.itheima.bean.Student");

    //2. 获得所有的字段(包含私有的)
    Field[] fields = clazz.getDeclaredFields();

    //System.out.println(fields.length);
}
```

```

    for (Field field : fields) {
        //取值
        //field.get(传入对象)
        //设置值
        //field.set(对象,值);
    }
    //3. 获得某一个特定的字段    String name;
    Field nameFiled = clazz.getDeclaredField("name");
    nameFiled.setAccessible(true); //暴力破解(可以访问私有的字段或者方法)
    //取值
    Object nameValue = nameFiled.get(student);
    System.out.println(nameValue);

    //赋值
    nameFiled.set(student, "李四");

    System.out.println(student.getName());

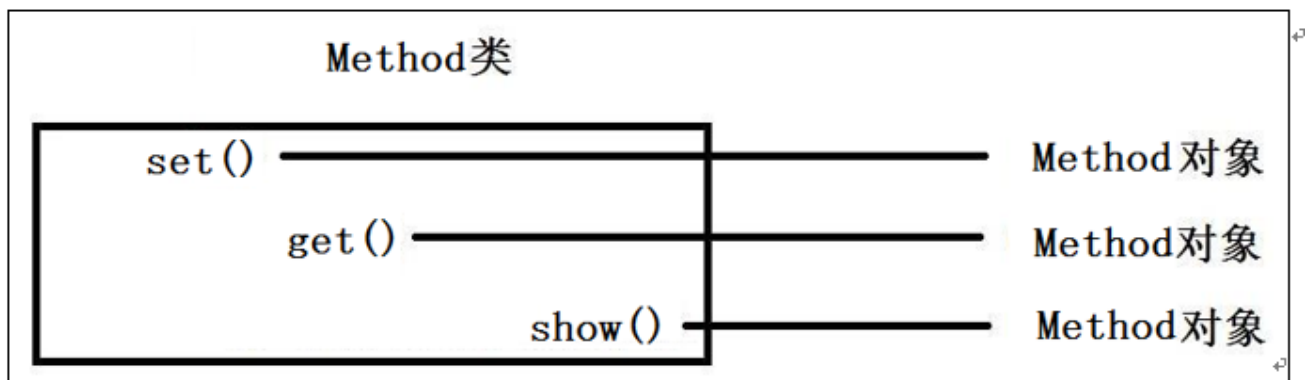
}

```

6.方法的反射

6.1概述

Method是方法类，类中的每一个方法都是Method的对象，通过Method对象可以调用方法。



6.2Class类中与Method相关方法

1. Method[] getMethods()
获取所有的public修饰的成员方法，包括父类中 Object中所有方法
2. Method[] getDeclaredMethods()
获取当前类中所有的方法，包含私有的，不包括父类中
3. Method getMethod("方法名", 方法的参数类型的字节码对象数组) 如果没有参数就不写
根据方法名和参数类型获得一个方法对象，只能是获取public修饰的
4. Method getDeclaredMethod("方法名", 方法的参数类型的字节码对象数组)
根据方法名和参数类型获得一个方法对象，包括private修饰的

6.3 Method类中常用方法

1. Object invoke(Object obj调用这个方法的对象, Object... args方法参数值的数组)

根据参数args调用对象obj的该成员方法

如果obj=null, 则表示该方法是静态方法

2. void setAccessible(true)

暴力反射, 设置为可以直接调用私有修饰的成员方法

示例代码

```
import java.lang.reflect.Method;
public class ConstructorDemo {
    public static void main(String[] args) throws Exception {
        Class clazz=Student.class;
        Method[] fields = clazz.getDeclaredMethods();
        for (Method field : fields) {
            System.out.println(field.getName());
        }
        Method method=clazz.getDeclaredMethod("speak1",String.class);
        method.setAccessible(true);
        System.out.println(method.getName());
        method.invoke(clazz.newInstance(),"李四");    李四为传入方法的参数
    }
}
```

7.反射练习

1.创建一个Student类

2.用单元测试来获取Student字节码,通过字节码创建出对象,构造函数的反射.成员变量的反射,成员方法的反射,反射main函数

Student类:

```
public class Student {
    private String name;
    private int age;
    private String sex;
    private Date birth;
    private String grade;
    构造方法 空参,满参,有参(部分参数)
    speak方法无参无返回,有参无返回
    get和set方法
}
```

7.1.获取Student字节码

@Test

```

public void fun01(){
    //1.对象.getClass();
    Student student = new Student();
    Class clazz = student.getClass();

    //2.类.class
    Class<Student> clazz1 = Student.class;

    //3.Class.forName("全限定名")
    try {
        Class clazz2 = Class.forName("com.itheima.demo4.Student");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

```

7.2通过字节码创建出对象【重点】

```

Student s1 = (Student) clazz.newInstance();

```

7.3.构造函数的反射

```

@Test
public void fun03() throws Exception{
    //1.获得字节码
    Class clazz = Class.forName("com.itheima.demo4.Student");
    //2.得到Constructor对象
    Constructor constructor =
clazz.getConstructor(String.class,int.class,String.class,Date.class);
    Student student = (Student) constructor.newInstance("张三",18,"李四",new Date());
    //student.speak();
    Date birth = student.getBirth();
    System.out.println(birth.toString());
}

```

7.4.成员变量字段的反射

```

Field[] fields = clazz.getDeclaredFields();
for (Field field : fields) {
    //System.out.println(field.getName());
}
//给成员变量赋值
Student student = new Student("张三", 18);
Field field = clazz.getDeclaredField("name");
field.setAccessible(true);//暴力破解
field.set(s1, "李四");
System.out.println(s1.getName());
}

```

7.5.成员方法的反射【重点】

```
public class ConstructorDemo {
    public static void main(String[] args) throws Exception {
        Class clazz=Student.class;
        Method m = clazz.getDeclaredMethod("speak1", String.class);
        m.setAccessible(true);
        m.invoke(clazz.newInstance(),"sasg");
    }
}
```

7.6.反射main方法【了解】首先要调用的main方法中最好有args参数的相关操作，不然调用无意义

```
package reflect;
public class Student {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}

package reflect;
import java.lang.reflect.Method;
public class ConstructorDemo {
    public static void main(String[] args) throws Exception {
        Class clazz = Student.class;
        Method m=clazz.getMethod("main",String[].class);
        // m.invoke(null,(Object)new String[]{"aa","bb"}); 由于主方法是静态的也可以传null
        m.invoke(clazz.newInstance(),(Object)new String[]{"aa","bb"});
    }
}
```

二,注解

1.案例需求

在一个类(测试类,TestDemo)中有三个方法,其中两个方法上有@MyTest,另一个没有.还有一个主测试类(MainTest)中有一个main方法. 在main方法中,让TestDemo类中含有@MyTest方法执行. 自定义@MyTest, 模拟单元测试.

```
class TestDemo{
    @MyTest
    public void fun01(){}
    @MyTest
    public void fun02(){}
}
```



```

    public void fun03(){}
}

class MainTest{
    public static void main(String[] args){
        //1. 获得TestDemo这个里面的所有的方法（反射）
        //2. 让有@MyTest注解的方法执行
    }
}

```

2,技术分析

2.1注解概述

annotation,是一种代码级别的说明,和类 接口平级关系.

注解 (Annotation) 相当于一种标记, 在程序中加入注解就等于为程序打上某种标记, 以后, javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有没有标记, 看你的程序有什么标记, 就去干相应的事, 标记可以加在包、类, 属性、方法, 方法的参数以及局部变量上定义

2.2.注解的作用

执行编译期的检查 例如:@Override

分析代码(主要用途:替代配置文件); 用在框架里面, 注解开发

2.3.JDK提供的三个基本的注解

@Override:描述方法的重写.

@SuppressWarnings:压制警告.

@Deprecated:标记过时

2.4.自定义注解 （尚无功能）

2.4.1注解格式

定义一个类:class

定义一个接口:interface

定义一个枚举:enum

定义一个注解:@interface

本质上就是一个接口,接口中可以定义变量(常量)和方法(抽象),注解中的方法叫注解属性

语法: @interface 注解名{}

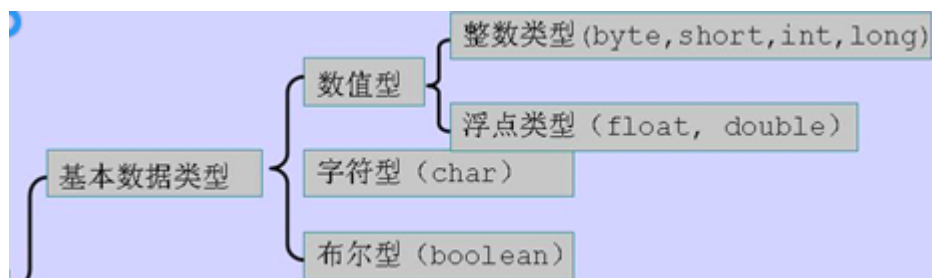
- 示例代码

```
public @interface Annotation01 {  
}
```

2.4.2 注解属性

2.4.2.1 属性类型

1. 基本类型



2. String

3. 枚举类型

4. 注解类型

5. Class 类型

6. 以上类型的一维数组类型

注意:

一旦注解有属性了,使用注解的时候,属性必须有值

```
public @interface MyTest {  
    int a();  
    String b();  
}  
  
public class TextDemo {  
    @MyTest(a = 1, b = "sa")  
    public void fun01(){}  
}
```

- 示例代码

```
/**  
 * 注解的属性; 格式和接口的方法很类似  
 * 1. 基本类型  
 * 2. String  
 * 3. 枚举类型  
 * 4. 注解类型  
 * 5. Class 类型
```

6. 以上类型的一维数组类型

```
*/  
public @interface Annotation02 {  
    int a(); //基本类型  
    String b(); //String  
    Color c(); //枚举类型  
    Annotation01 d(); //注解类型  
    Class e(); //Class类型  
    String[] f(); //一维数组类型  
}
```

2.4.2.2 注解属性赋值

注:

- 若属性类型为一维数组的时候,当数组的值只有一个的时候,大括号可以不写

```
@MyAnnotation4(ss = { "a" })  
@MyAnnotation4(ss = "a")
```

- 若属性名为value的时候,且只有这一个属性需要赋值的时候可以省略value【重点】

```
public @interface MyTest {  
    int value();  
}  
public class TextDemo {  
    @MyTest(1)  
    public void fun01(){}  
}
```

- 注解属性可以有默认值

属性类型 属性名() default 默认值;

```
public @interface MyTest {  
    int value() default 0;  
}  
public class TextDemo {  
    @MyTest(3232)  
    public void fun01(){}  
}
```

2.5. 元注解

2.5.1 概述

定义在注解上的注解

2.5.2类型

实现 [Annotation](#) 的 [java.lang.annotation](#) 中的类

| | | |
|-----------|----------------------------|-------------------------------------|
| interface | Documented | 指示某一类型的注释将通过 javadoc 和类似的默认工具进行文档化。 |
| interface | Inherited | 指示注释类型被自动继承。 |
| interface | Retention | 指示注释类型的注释要保留多久。 |
| interface | Target | 指示注释类型所适用的程序元素的种类。 |

@Target:注解定义在什么上面(位置),

多个位置用逗号隔开,默认注解可以在任何位置. 值为:ElementType的枚举值

METHOD:方法

TYPE:类 接口

FIELD:字段

CONSTRUCTOR:构造方法声明

```
@Target(ElementType.METHOD)
public @interface MyTest {
}

public class TextDemo {
    @MyTest
    public void fun(){}
}
```

@Retention:定义该注解保留到那个代码阶段, 值为:RetentionPolicy类型

SOURCE:只在源码上保留(默认)

CLASS:在源码和字节码上保留

RUNTIME:在所有的阶段都保留

```
@Retention(RetentionPolicy.SOURCE)
public @interface MyTest {
}

public class TextDemo {
    @MyTest
    public void fun(){}
}
```

.java (源码阶段) ----编译--> .class(字节码阶段) ----加载内存--> 运行(RUNTIME)

2.6.注解解析 自定义注解最好保留到运行阶段

java.lang.reflect.AnnotatedElement

- **T getAnnotation(ClassannotationType):**
得到指定类型的注解引用。没有返回null。
- **boolean isAnnotationPresent(Class<?extends Annotation> annotationType) :**
判断指定的注解有没有。
Class、Method、Field、Constructor等实现了AnnotatedElement接口。
- **Annotation[] getAnnotations() :** 得到所有的注解，包含从父类继承下来的。
- **Annotation[] getDeclaredAnnotations() :** 得到自己身上的注解。

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTest {
}

@MyTest
public class Demo {
    private int a;

    @MyTest
    public void show(){
    }

    public static void main(String[] args) throws NoSuchMethodException {
        Class clazz=Demo.class;
        Annotation annotation =clazz.getAnnotation(MyTest.class);
        System.out.println(annotation);
        Method method = clazz.getMethod("show");
        System.out.println(method.isAnnotationPresent(MyTest.class));
    }
}
```

3.思路分析

- 定义两个类(TestDemo, MainDemo)和一个注解(@MyTest)
- 在TestDemo里面定义三个方法, 其中两个方法上面添加@MyTest
- 在MainDemo:
 - //1. 获得TestDemo的字节码
 - //2. 获得TestDemo类里面的所有的方法对象(数组)
 - //3. 遍历方法对象(数组), 判断当前的方法上面是否有@MyTest, 有的话就执行

4.代码实现

- MyTest.java

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTest {
}
```

- TestDemo.java

```
class TestDemo{
    @MyTest
    public void fun01(){
        System.out.println("1");
    }
    @MyTest
    public void fun02(){
        System.out.println("2");
    }
    public void fun03(){
        System.out.println("3");
    }
}
```

- MainDemo.java

```
class MainTest{
    public static void main(String[] args) throws Exception {
        Class clazz=TestDemo.class;
        Method[] m=clazz.getDeclaredMethods();
        for (Method method : m) {
            if(method.isAnnotationPresent(MyTest.class)){
                method.invoke(clazz.newInstance());
            }
        }
    }
}
```

三,BeanUtils

1.BeanUtils相关的知识点

1.1什么是BeanUtils



BeanUtils是Apache Commons组件的成员之一，主要用于简化JavaBean封装数据的操作。常用的操作有以下三个：

1. 对JavaBean的属性进行赋值和取值。
2. 将一个JavaBean所有属性赋值给另一个JavaBean对象中。
3. 将一个Map集合的数据封装到一个JavaBean对象中。

1.2JavaBean复习

JavaBean就是一个类，但该类需要满足以下四个条件：

1. 类必须使用public修饰。
2. 提供无参数的构造器。
3. 提供getter和setter方法访问属性。
4. 实现序列化接口(一般没写)

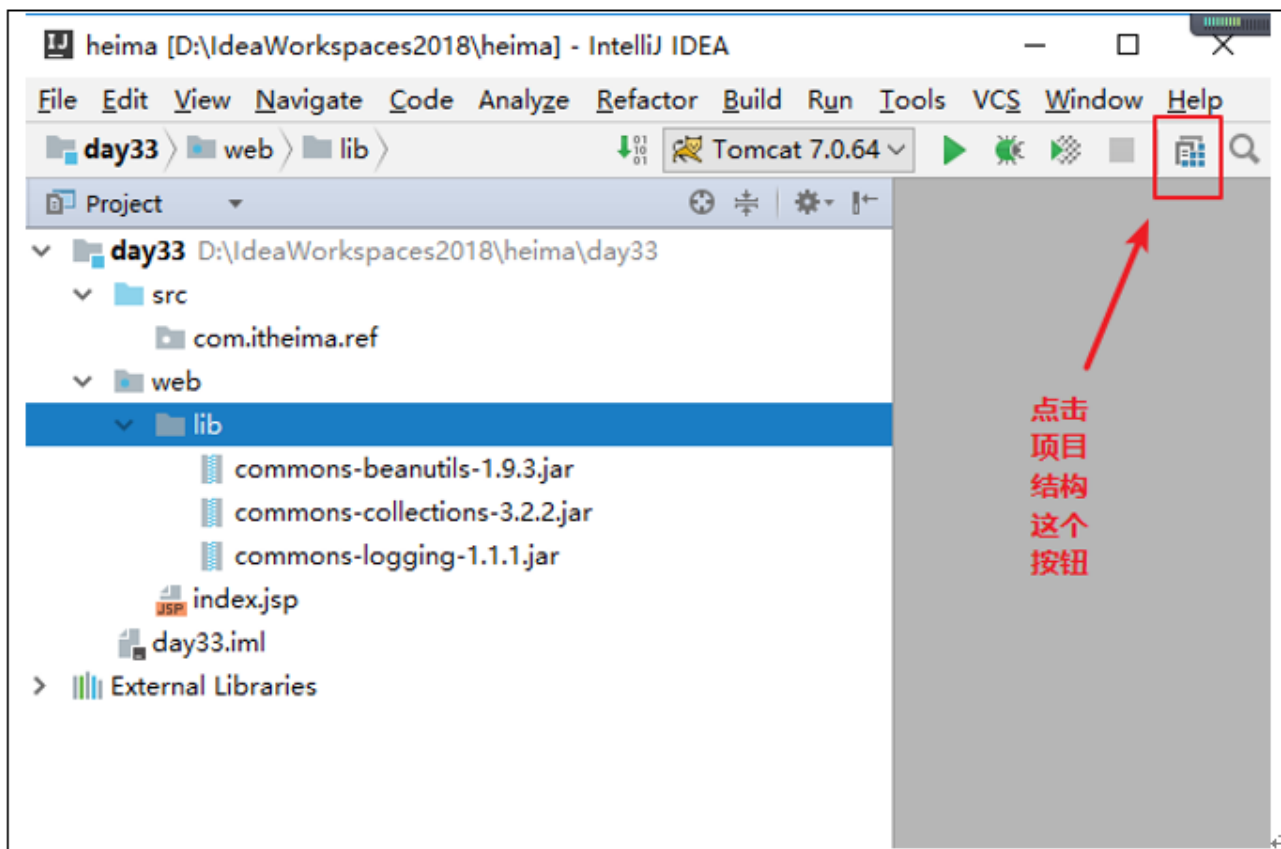
JavaBean的两个重要概念

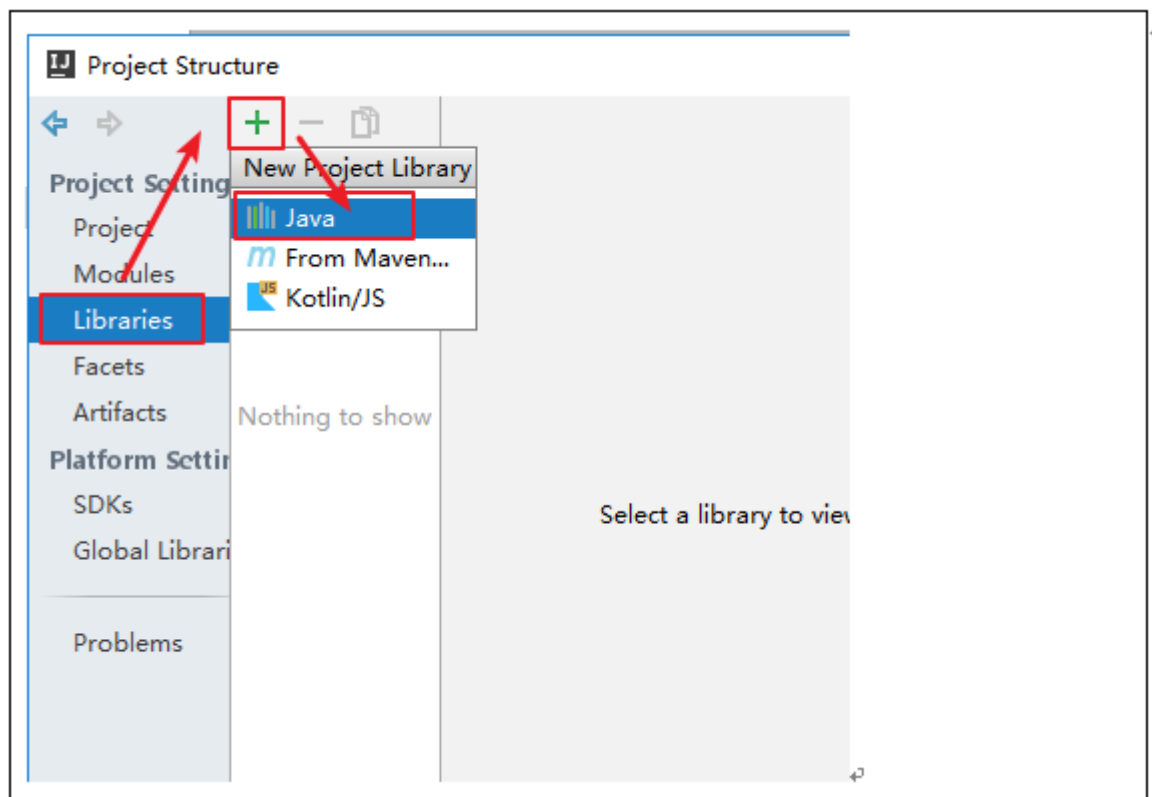
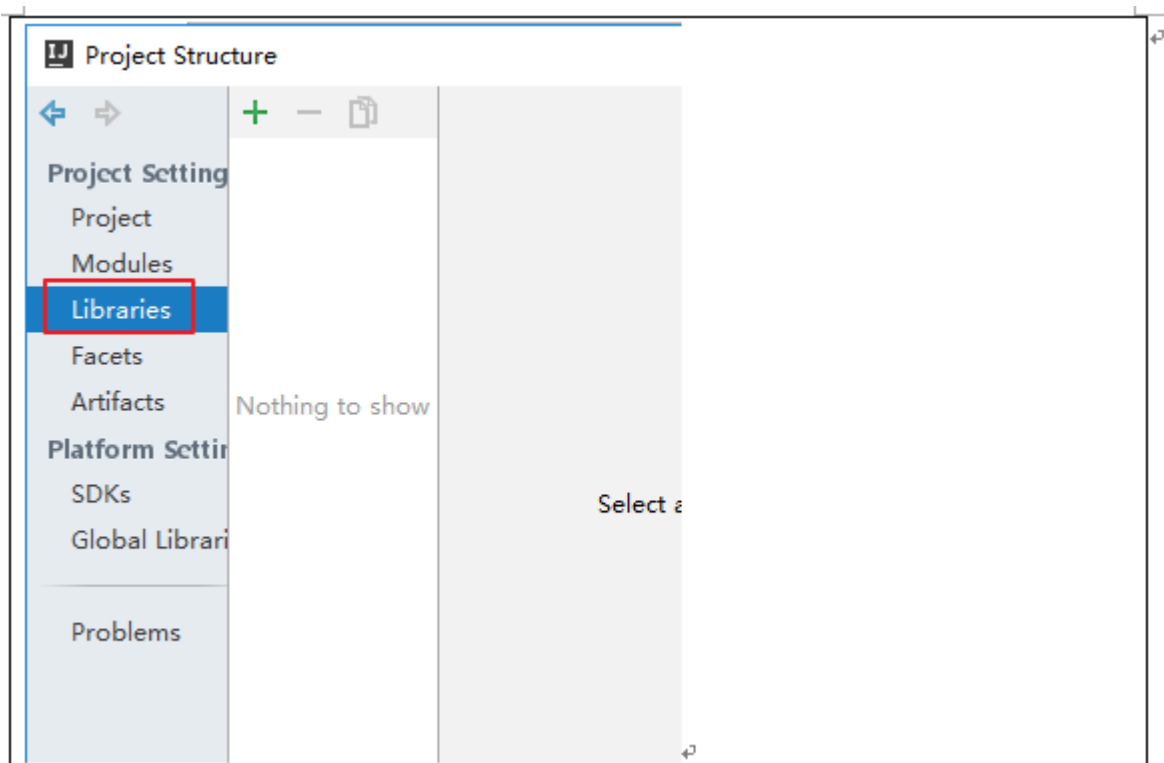
字段：就是成员变量，字段名就是成员变量名。

属性：属性名通过setter/getter方法去掉set/get前缀，首字母小写获得。

2.BeanUtils的使用 直接BeanUtils.populate(user对象,map对象)

2.1导入jar包





2.2 BeanUtils常见的方法

- `public static void populate(Object bean, Map<String, ? extends Object> properties);` 将一个Map集合中的数据封装到指定对象bean中(注意：对象bean的属性名和Map集合中键要相同)
- 键为User属性名，值为对应的值

2.3 BeanUtils的使用

把map的数据封装到JavaBean里面

- Student.java

```
public class User {  
    private String name;  
    private String gender;  
    private String address;  
    private String age;  
    空参构造  
    set/get方法  
    toString方法  
}
```

- BeanUtilsDemo.java

```
public class BeanUtilsTest {  
    public static void main(String[] args) throws InvocationTargetException,  
    IllegalAccessException {  
        User user=new User();  
        Map<String,String> map = new HashMap<String,String>();  
        map.put("name", "林青霞");  
        map.put("gender", "女");  
        map.put("address", "北京");  
        map.put("age", "23");  
        BeanUtils.populate(user,map);  
        System.out.println(user);  
    }  
}
```