今日内容

```
1. Junit单元测试
2. 反射
3. 注解
```

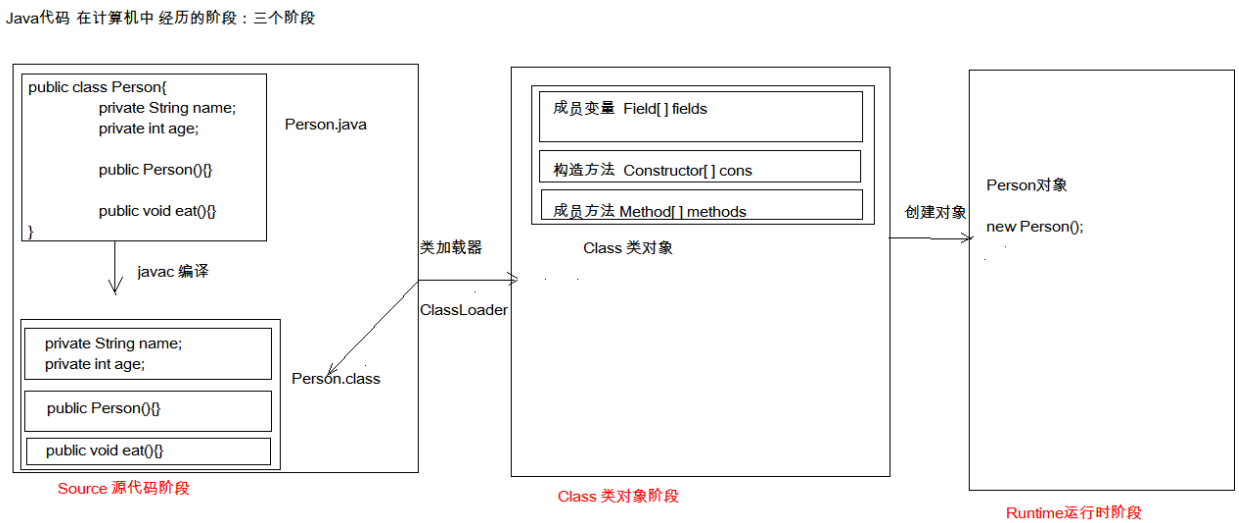
Junit单元测试

- 测试分类：
  1. 黑盒测试：不需要写代码，给输入值，看程序是否能够输出期望的值。
  2. 白盒测试：需要写代码的。关注程序具体的执行流程。
- Junit使用：白盒测试
  - 步骤：
    1. 定义一个测试类(测试用例)
       - 建议：
         - 测试类名：被测试的类名Test CalculatorTest
         - 包名：xxx.xxx.xx.test cn.itcast.test
    2. 定义测试方法：可以独立运行
       - 建议：
         - 方法名：test测试的方法名 testAdd()
         - 返回值：void
         - 参数列表：空参
    3. 给方法加@Test
    4. 导入junit依赖环境
  - 判定结果：
    - 红色：失败
    - 绿色：成功
    - 一般我们会使用断言操作来处理结果
      - Assert.assertEquals(期望的结果,运算的结果);
  - 补充：
    - @Before:
      - 修饰的方法会在测试方法之前被自动执行
    - @After:
      - 修饰的方法会在测试方法执行之后自动被执行s

```
package cn.test;

import org.junit.After;
```

```java
import org.junit.Before;
import org.junit.Test;

public class test {

    @Before
    public void testBefore(){
        System.out.println("Before");    //测试之前执行
    }

    @After
    public void testAfter(){
        System.out.println("After");     //测试之后执行
    }


    @Test
    public void testAdd(){
        System.out.println("Add");   //测试
    }

}
```

反射：框架设计的灵魂



> 其中在编译成对象之前程序需要经过一个Class类对象的阶段，在这个阶段成员变量被封装为Field[]，构造方法被封装为Constructor[]，成员方法被封装为Method[]

- 框架：半成品软件。可以在框架的基础上进行软件开发，简化编码
- 反射：将类的各个组成部分封装为其他对象，这就是反射机制
    - 好处：
        1. 可以在程序运行过程中，操作这些对象。
        2. 可以解耦，提高程序的可扩展性。
- 获取Class对象的方式：
    1. Class.forName("全类名")：将字节码文件加载进内存，返回Class对象
        - 多用于配置文件，将类名定义在配置文件中。读取文件，加载类
    2. 类名.class：通过类名的属性class获取
        - 多用于参数的传递
    3. 对象.getClass()：getClass()方法在Object类中定义着。
        - 多用于对象的获取字节码的方式
    - 结论： 同一个字节码文件(*.class)在一次程序运行过程中，只会被加载一次，不论通过哪一种方式获取的Class对象都是同一个。
- Class对象功能：
    - 获取功能：
        1. 获取成员变量们
            - Field[] getFields() ： 获取所有public修饰的成员变量
            - Field getField(String name) 获取指定名称的 public修饰的成员变量
            - Field[] getDeclaredFields() 获取所有的成员变量，不考虑修饰符
            - Field getDeclaredField(String name)
        2. 获取构造方法们
            - Constructor<?>[] getConstructors()
            - Constructor getConstructor(类<?>... parameterTypes)
            - Constructor getDeclaredConstructor(类<?>... parameterTypes)
            - Constructor<?>[] getDeclaredConstructors()
        3. 获取成员方法们：
            - Method[] getMethods()
            - Method getMethod(String name, 类<?>... parameterTypes)
            - Method[] getDeclaredMethods()
            - Method getDeclaredMethod(String name, 类<?>... parameterTypes)
        4. 获取全类名
            - String getName()
- Field：成员变量
    - 操作：
        1. 设置值
            - void set(Object obj,Object value)
        2. 获取值
            - Object get(Object obj)
        3. 忽略访问权限修饰符的安全检查
            - setAccessible(true):暴力反射

> 对public的成员变量进行操作

```
package cn.test;

public class Person {
    public int age;
    public String name;

    public boolean sex;
    private int id;

    public Person(int age, String name, boolean sex, int id) {
        this.age = age;
        this.name = name;
        this.sex = sex;
        this.id = id;
    }

    public Person() {
    }

    public boolean isSex() {
        return sex;
    }

    public void setSex(boolean sex) {
        this.sex = sex;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Person{" +
                "age=" + age +
                ", name='" + name + '\'' +
                ", sex=" + sex +
                ", id=" + id +
                '}';
    }
}
```

```
package cn.test;

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

```
public class PersonImpl {

    public static void main(String[] args) throws NoSuchFieldException,
IllegalAccessException {

        Class<Person> personClass = Person.class;   //获取字节码文件

        System.out.println("下面遍历getFields[]");  //获取组

        Field[] fields = personClass.getFields();

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

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

        System.out.println("下面对特定的属性进行操作");

        Field name = personClass.getField("name");//获取值

        Person person = new Person();
        Object value = name.get(person);    //这里面传一个Object传进去，返回一个Object

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

        System.out.println("value-->"+value);

        System.out.println("person-->"+person);

        System.out.println("设置后");

        name.set(person,"张三");

        value = name.get(person);
        System.out.println("name-->"+value);

        System.out.println("person-->"+person);

    }
}
```

```
//输出:
下面遍历getFields[]
public int cn.test.Person.age
public java.lang.String cn.test.Person.name
public boolean cn.test.Person.sex
---------------------
下面对特定的属性进行操作
----------
value-->null
person-->Person{age=0, name='null', sex=false, id=0}
设置后
name-->张三
person-->Person{age=0, name='张三', sex=false, id=0}
```

对private修饰符进行操作

```
package cn.test;

public class Person {

    private String name;
    private int age;

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

```
package cn.test;

import java.lang.reflect.Field;

public class PersonImpl {

    public static void main(String[] args) throws NoSuchFieldException,
IllegalAccessException {

        Class<Person> personClass = Person.class;          // 获取Person字节码文件

        System.out.println("下面是获取所有的成员变量");

        Field[] declaredFields = personClass.getDeclaredFields();    //获取每一个成员变量

        for (Field declaredField : declaredFields) {
            System.out.println(declaredField);  //打印输出
        }
```

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

        System.out.println("下面是操作指定的成员变量");

        Field name = personClass.getDeclaredField("name");  //获取指定的成员变量

        System.out.println("name-->"+name);   //打印输出

        /*
        * 由于成员变量不可以直接进行访问，否则会警告，所以在这里忽略访问修饰符的安全检查，进行暴力反射
        * */
        name.setAccessible(true);   //暴力反射

        Person person = new Person();   //获取新的person对象

        Object value = name.get(person);    //获取新的person对象中的name

        System.out.println("name的值-->"+value);

        name.set(person,"王小二");

        System.out.println("设置之后");

        value = name.get(person);

        System.out.println("name的值-->"+value);
    }

}
```

```
//输出
下面是获取所有的成员变量
private java.lang.String cn.test.Person.name
private int cn.test.Person.age
------------------
下面是操作指定的成员变量
name-->private java.lang.String cn.test.Person.name
name-->null
设置之后
name-->王小二
```

- Constructor:构造方法
  - 创建对象：
    - T newInstance(Object... initargs)
- 如果使用空参数构造方法创建对象，操作可以简化：Class对象的newInstance方法

```
package cn.test;

public class Person {
```

```
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

```
package cn.test;


import java.lang.reflect.Constructor;

public class PersonImpl {

    public static void main(String[] args) throws Exception {
        /*
        * 使用构造方法
        * */
        Class<Person> personClass = Person.class;

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

            System.out.println("指定构造方法,使用有参的构造方法来创建对象");

        Constructor<Person> constructor1 = personClass.getConstructor(String.class,
int.class); //有参的构造方法

            Person person1 = constructor1.newInstance("王小二", 22);   //创建有参的对象

            System.out.println(person1);

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

            System.out.println("制定构造方法，使用无参的构造方法来创建对象");

        Constructor<Person> constructor2 = personClass.getConstructor();    //无参的构造方法

            Person person2 = constructor2.newInstance();        //创建无参的对象
```

```
            System.out.println(person2);

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

            Person person3 = personClass.newInstance();  //直接创建对象，不必调用方法，其实最后调用的就是刚才写的那些东西

            System.out.println(person3);
    }

}
//其实如果要想使用构造方法设置私有，那么也需要暴力反射，比如constructor1.setAccessible(true)
```

```
-----------------
指定构造方法,使用有参的构造方法来创建对象
Person{name='王小二', age=22}
-----------------
制定构造方法，使用无参的构造方法来创建对象
Person{name='null', age=0}
----------------
Person{name='null', age=0}
```

- Method：方法对象
  - 执行方法：
    - Object invoke(Object obj, Object... args)
  - 获取方法名称：
    - String getName:获取方法名

> 使用Method

```
package cn.test;

public class Person {

    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = 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 void eat(){
        System.out.println("eat......");
    }

    public void eat(String food){
        System.out.println("eat"+food+".....");
    }


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

```
package cn.test;


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

public class PersonImpl {

    public static void main(String[] args) throws Exception {
        Class<Person> personClass = Person.class;      //获取字节码文件

        Person person = new Person();      //获取Person对象

        System.out.println("下面是使用单个方法");

        Method eat1 = personClass.getMethod("eat"); //获取指定的方法，使用无参方法
        eat1.invoke(person);    //执行方法

        Method eat2 = personClass.getMethod("eat", String.class);   //获取有参方法，传进去需要
的字节码文件
        eat2.invoke(person,"food"); //执行方法
```

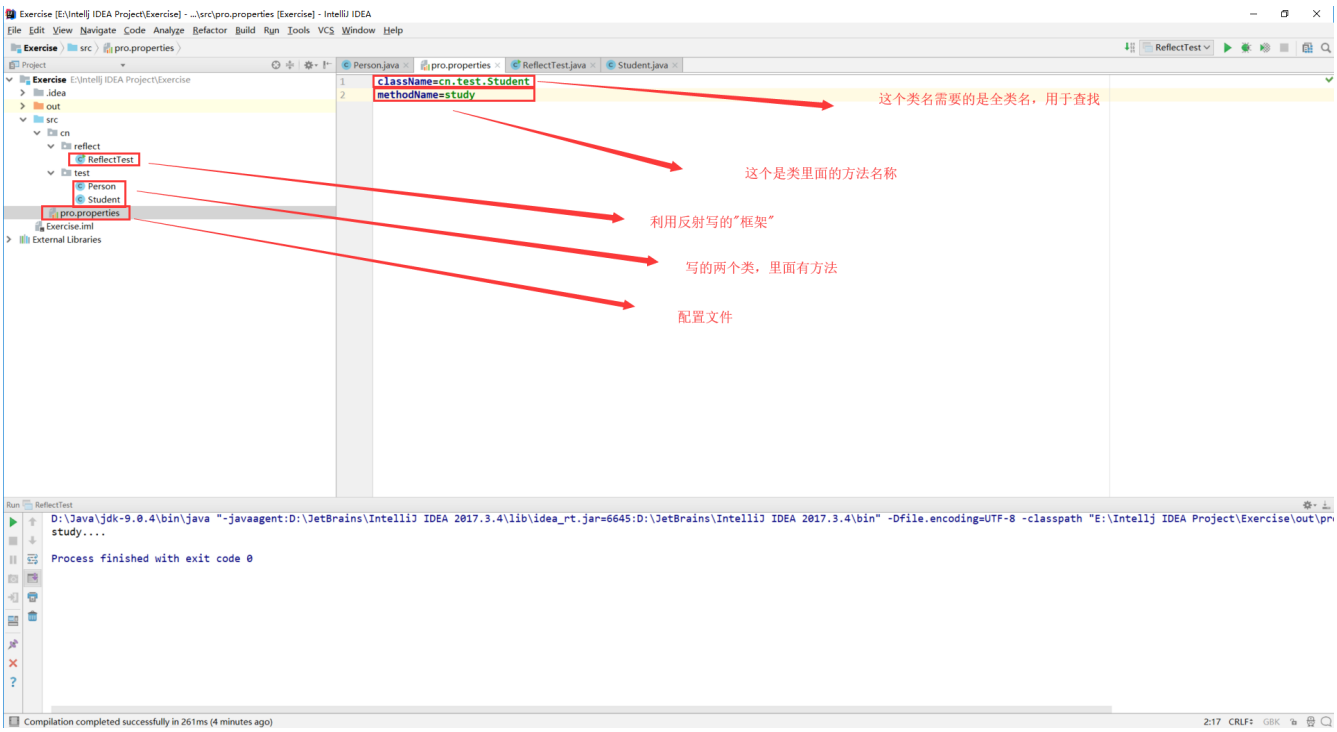
```
        System.out.println("----------------------");

        System.out.println("下面是获取public修饰的方法组，注意，隐藏的方法也会被显示出来");

        Method[] methods = personClass.getMethods();

        for (Method method : methods) {
            System.out.println(method); //打印输出
        }

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

        System.out.println("获取方法名称");

            System.out.println(eat1.getName());

        System.out.println("获取类名称");

            System.out.println(personClass.getName());
    }

}
/*
* 注意，假如要获取私有方法的时候，那么也要使用暴力反射来进行访问，比如
* eat1.setAccessible(true);
* */
```

```
下面是使用单个方法
eat......
eatfood.....
----------------------
下面是获取public修饰的方法组，注意，隐藏的方法也会被显示出来
public java.lang.String cn.test.Person.toString()
public java.lang.String cn.test.Person.getName()
public void cn.test.Person.setName(java.lang.String)
public int cn.test.Person.getAge()
public void cn.test.Person.setAge(int)
public void cn.test.Person.eat(java.lang.String)
public void cn.test.Person.eat()
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
------------------
获取方法名称
eat
获取类名称
```

```
cn.test.Person
```

---

* 案例：
    * 需求：写一个"框架"，不能改变该类的任何代码的前提下，可以帮我们创建任意类的对象，并且执行其中任意方法
        * 实现：
            1. 配置文件
            2. 反射
        * 步骤：
            1. 将需要创建的对象的全类名和需要执行的方法定义在配置文件中
            2. 在程序中加载读取配置文件
            3. 使用反射技术来加载类文件进内存
            4. 创建对象
            5. 执行方法



```java
package cn.test;

public class Person {

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

}
```

```java
package cn.test;

public class Student {

    public void study(){
        System.out.println("study....");
    }

}
```

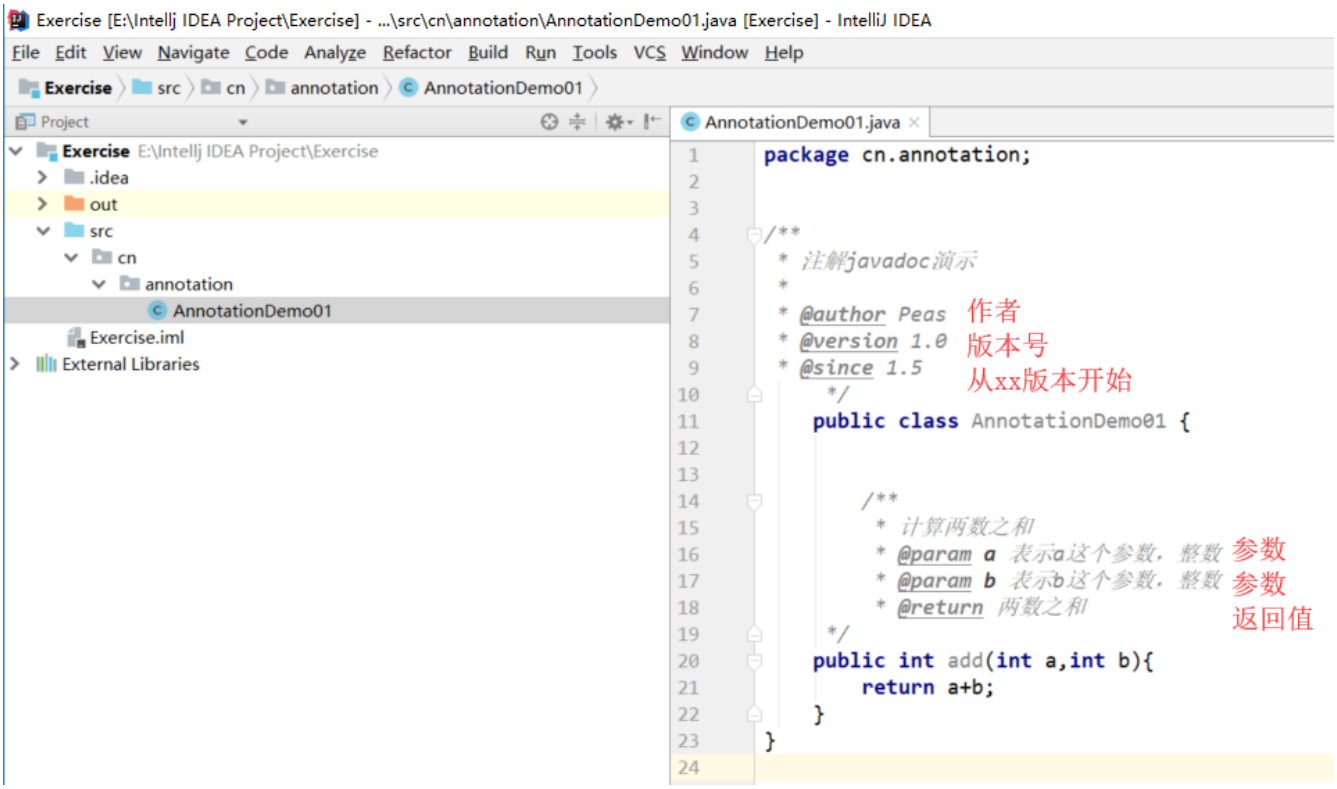
```java
package cn.reflect;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Method;
import java.util.Properties;

public class ReflectTest {

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

        //0.创建Properties对象
        Properties properties = new Properties();

        //1.加载配置文件，转换为一个集合
            //1.1获取配置文件
        /*
        * getClassLoader：类加载器，使用字节码文件获取getClassLoader对象以用来获取类加载器
        * 使用类加载器可以获取任意的文件，自然也可以获取Properties文件
        * */
        ClassLoader classLoader = ReflectTest.class.getClassLoader();

            //1.2
        /*
        *使用类加载器获取Properties文件
        * 使用 getResourceAsStream方法可以将指定的文件转换为InputStream流
        * */
        InputStream resourceAsStream = classLoader.getResourceAsStream("pro.properties");

        /*
        *加载配置文件
        * new Properties().load(new InputStream())
        * */
        properties.load(resourceAsStream);


        //2.获取配置文件中定义的数据
            /*
            * 使用 getProperty()来使用key来获取value
            * */
        String className = properties.getProperty("className");
        String methodName = properties.getProperty("methodName");
```

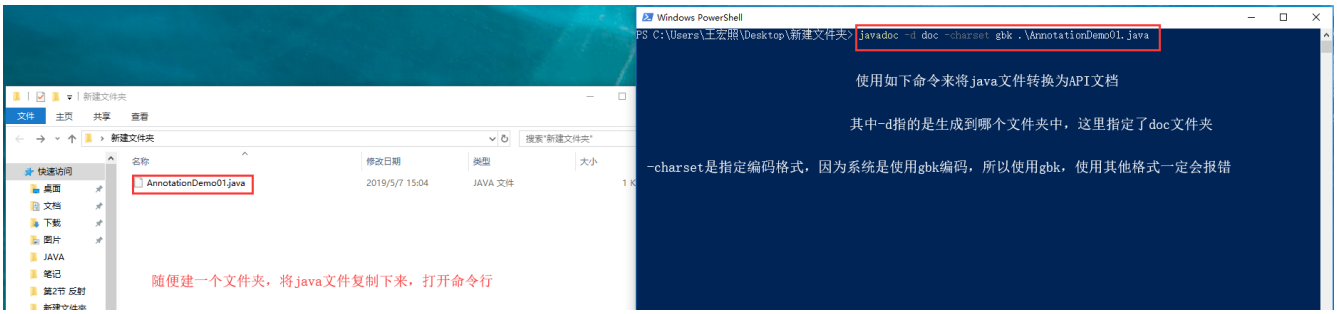
```
        //3.加载类进内存
            /*
            * 使用 Class.forName()加载类进内存
            * */
        Class cls = Class.forName(className);

        //4.创建对象
        Object obj = cls.newInstance();

        //5.使用方法
            //5.1获取方法
        Method method = cls.getMethod(methodName);
            //5.2执行
        method.invoke(obj);

    }

}
```

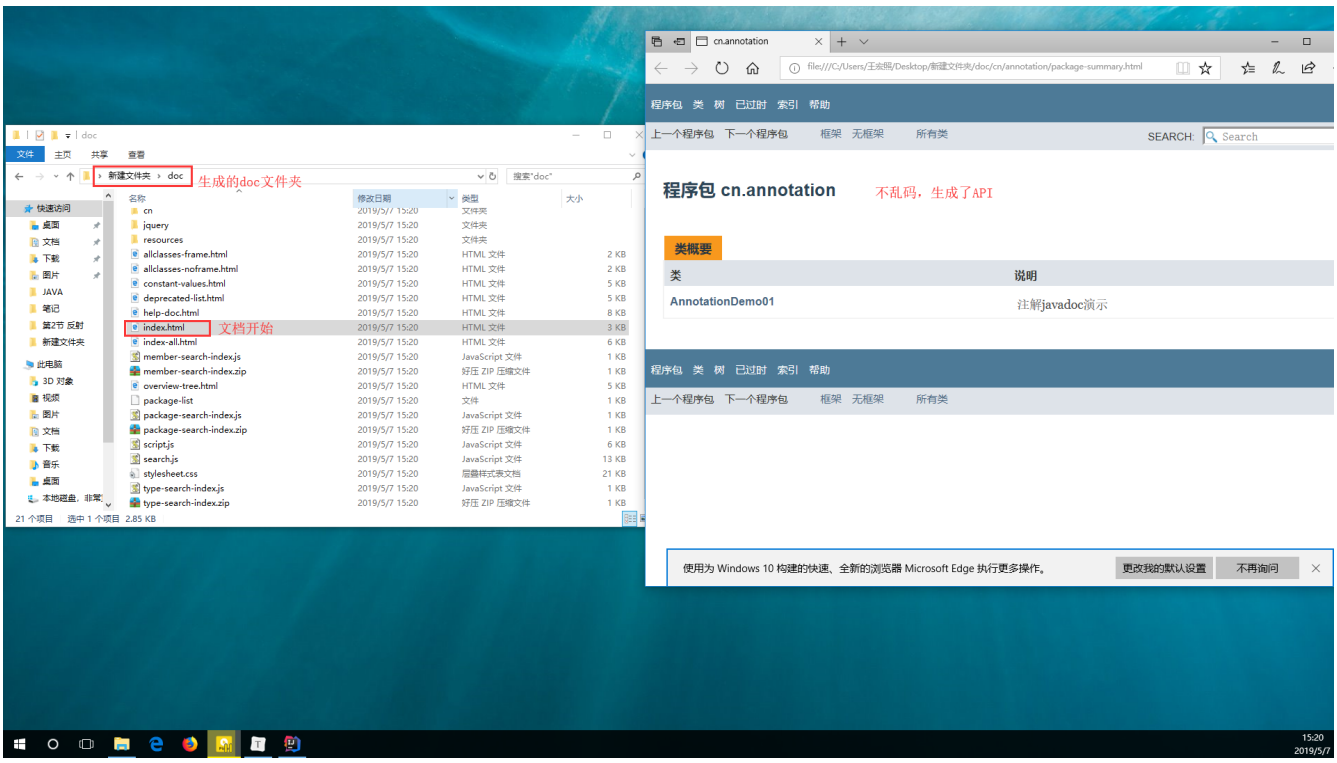
> 只要改动配置文件，无需改动代码就可以执行不同的方法

注解：

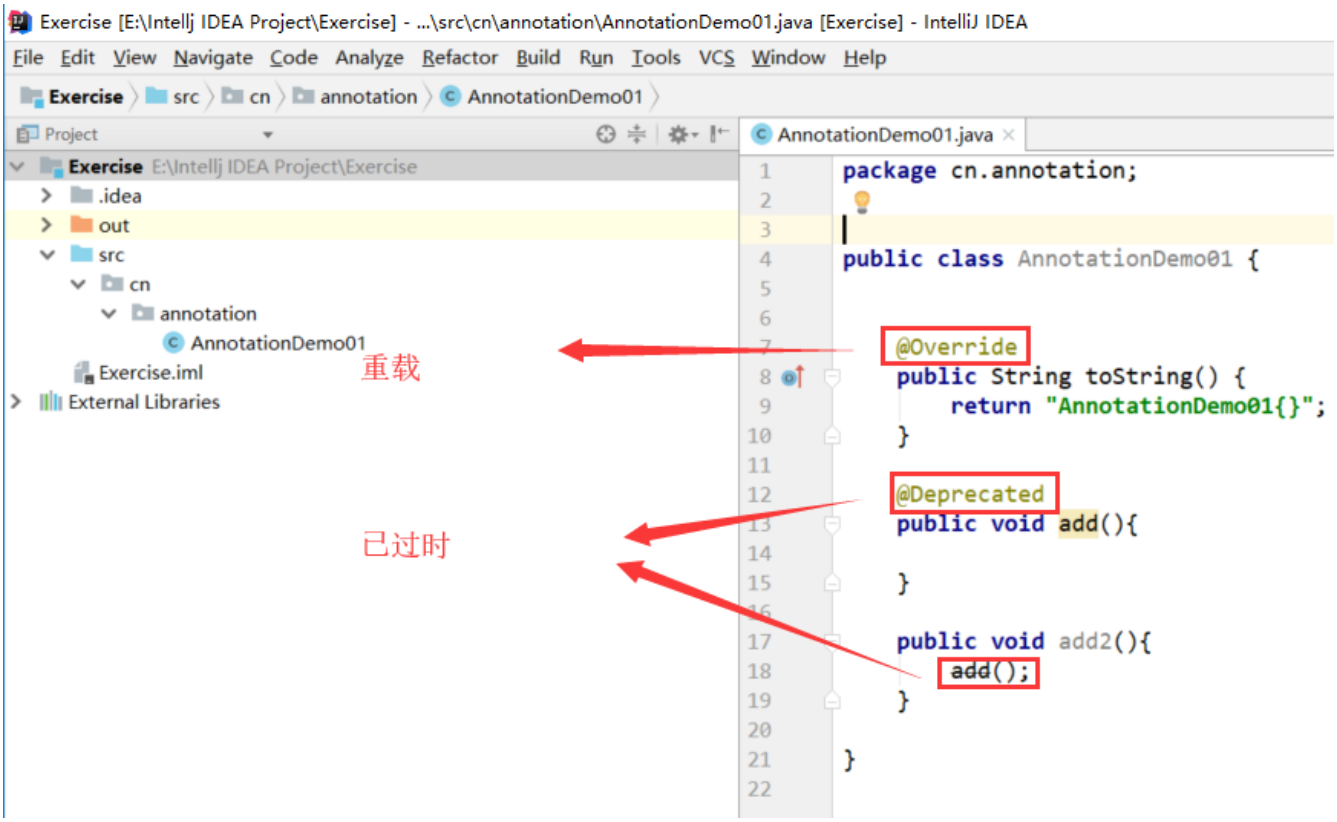
- 概念：说明程序的。给计算机看的
- 注释：用文字描述程序的。给程序员看的
- 定义：注解（Annotation），也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。
- 概念描述：
    - JDK1.5之后的新特性
    - 说明程序的
    - 使用注解：@注解名称

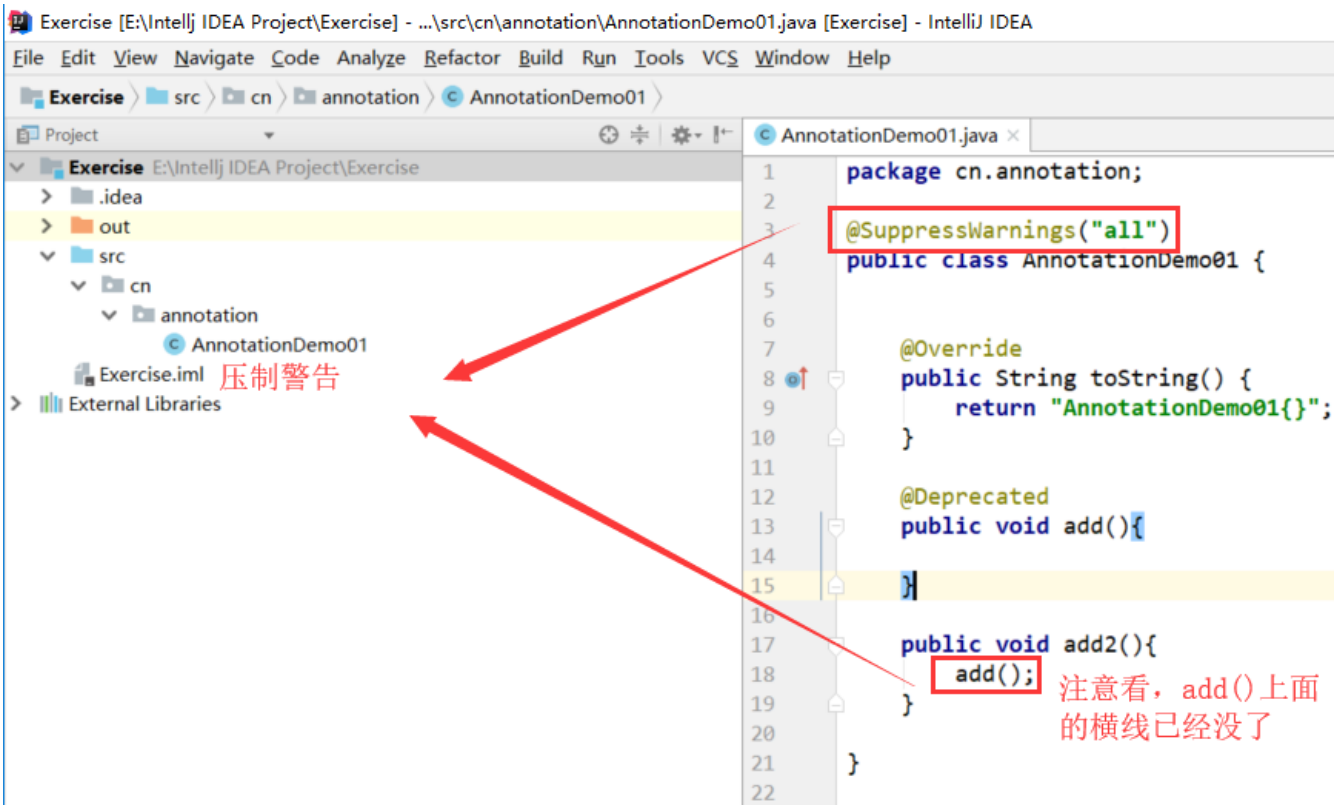

Exercise [E:\Intellj IDEA Project\Exercise] - ...\src\cn\annotation\AnnotationDemo01.java [Exercise] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Exercise src cn annotation AnnotationDemo01
Project
Exercise E:\Intellj IDEA Project\Exercise
.idea
out
src
cn
annotation
AnnotationDemo01
Exercise.iml
External Libraries
AnnotationDemo01.java
package cn.annotation;
/**
* 注解javadoc演示
*
* @author Peas
* @version 1.0
* @since 1.5
*/
public class AnnotationDemo01 {
/**
* 计算两数之和
* @param a 表示a这个参数，整数
* @param b 表示b这个参数，整数
* @return 两数之和
*/
public int add(int a,int b){
return a+b;
}
}
作者
版本号
从xx版本开始
参数
参数
返回值


Windows PowerShell
PS C:\Users\王宏照\Desktop\新建文件夹> javadoc -d doc -charset gbk .\AnnotationDemo01.java
使用如下命令来将java文件转换为API文档
其中-d指的是生成到哪个文件夹中，这里指定了doc文件夹
-charset是指定编码格式，因为系统是使用gbk编码，所以使用gbk，使用其他格式一定会报错
AnnotationDemo01.java
2019/5/7 15:04
JAVA 文件
随便建一个文件夹，将java文件复制下来，打开命令行


新建文件夹 doc
生成的doc文件夹
index.html
文档开始
cn.annotation
程序包 cn.annotation
不乱码，生成了API
类概要
类
说明
AnnotationDemo01
注解javadoc演示
SEARCH:
Search
使用为 Windows 10 构建的快速、全新的浏览器 Microsoft Edge 执行更多操作。
更改我的默认设置
不再询问

- 作用分类： ①编写文档：通过代码里标识的注解生成文档【生成文档doc文档】 ②代码分析：通过代码里标识的注解对代码进行分析【使用反射】 ③编译检查：通过代码里标识的注解让编译器能够实现基本的编译检查【Override】
- JDK中预定义的一些注解
  - @Override ： 检测被该注解标注的方法是否是继承自父类(接口)的
  - @Deprecated：该注解标注的内容，表示已过时
  - @SuppressWarnings：压制警告
    - 这个需要传参，一般传递参数all @SuppressWarnings("all")
    - 一般这个注解写到类上，这样所有的警告都没了
    - 这个压制警告里面传all是让所有的警告都不会出现





- 自定义注解
  - 格式：分为两部分
    1. 元注解
    2. public @interface 注解名称{ 属性列表; }
  - 本质：注解本质上就是一个接口，该接口默认继承Annotation接口
    - public interface MyAnno extends java.lang.annotation.Annotation {}
  - 属性：接口中的抽象方法
    - 要求：
      1. 属性的返回值类型有下列取值
         - 基本数据类型
         - String
         - 枚举
         - 注解
         - 以上类型的数组
      2. 定义了属性，在使用时需要给属性赋值
         1. 如果定义属性时，使用default关键字给属性默认初始化值，则使用注解时，可以不进行属性的赋值。
         2. 如果只有一个属性需要赋值，并且属性的名称是value，则value可以省略，直接定义值即可。
         3. 数组赋值时，值使用{}包裹。如果数组中只有一个值，则{}可以省略

```
package cn.annotation;
```

```java
public @interface MyAnno {//自定义的注解

    /*
    * 在抽象方法中定义的是属性
    * */
    public String name();

    /*
    * 在抽象方法中定义了一个属性，但是这个属性有默认值，就是22
    * */
    public int age() default 22;


    /*
    * 字符数组
    * */
    int[] i();

}
```

```java
package cn.annotation;

@SuppressWarnings("all")
public class AnnotationDemo01 {


    /*使用注解时，属性需要赋值，当只有一个属性的时候，可以不写 名称=值 ，直接写值即可
    * 这里的注解是自定义的Myanno，需要传值，但是因为age有了默认的22，所以age可以不传
    * 值与值之间可以使用逗号隔开
    * 但是数组赋值是比较特殊的，使用大括号赋值，但是如果只有一个值，大括号可以省略不写
    * */
    @MyAnno(name = "zhangsan",i={1,2,3,4})//使用注解
    public void add(){
    }

}
```

- 元注解：用于描述注解的注解
  - @Target：描述注解能够作用的位置
    - ElementType取值：
      - TYPE：可以作用于类上
      - METHOD：可以作用于方法上
      - FIELD：可以作用于成员变量上
  - @Retention：描述注解被保留的阶段
    - @Retention(RetentionPolicy.RUNTIME)：当前被描述的注解，会保留到class字节码文件中，并被JVM读取到
  - @Documented：描述注解是否可以被抽取到api文档中

- @Inherited：描述注解是否被子类继承

```
package cn.annotation;


import java.lang.annotation.*;


/*
* 表示只能作用于方法上
* */
@Target(value={ElementType.METHOD})

/*
* 注解一般作用于RunTime阶段,一般就用RunTime
* */
@Retention(value= RetentionPolicy.RUNTIME)

/*
* 表示这个注解会被抽取到javadoc文档中
* */
@Documented

/*
* 表示了这个注解会自动被子类继承，只要有一个父类加上了这个注解，那么这个父类的子类都会自动继承这个注解
* */
@Inherited

public @interface MyAnno {//自定义的注解
    public String name();
}
```

- 在程序使用(解析)注解：获取注解中定义的属性值
  1. 获取注解定义的位置的对象 （Class，Method,Field)
  2. 获取指定的注解
     - getAnnotation(Class) //其实就是在内存中生成了一个该注解接口的子类实现对象

```
public class ProImpl implements Pro{
    public String className(){
        return "cn.itcast.annotation.Demo1";
    }
    public String methodName(){
        return "show";
    }
}
```

  3. 调用注解中的抽象方法获取配置的属性值

- 注解

```
package cn.annotation;


import javax.swing.text.Document;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


/*
* 确认这个注解是作用在类上的
* */
@Target(ElementType.TYPE)

/*
* 确认注解的运行时间是在程序运行时
* */
@Retention(RetentionPolicy.RUNTIME)

public @interface Anno {
   String className();
   String methodName();
}
```

```
package cn.annotation;

public class reflect {

    public void show(){
        System.out.println("show...");
    }
}
```

- 第一个类

```
package cn.annotation;

public class reflect {

    public void show(){
        System.out.println("show...");
    }
}
```

- 第二个类

```
package cn.annotation;

public class reflect2 {

    public void show2(){
        System.out.println("show2...");
    }
}
```

- 使用注解来进行方法的调用

```
package cn.annotation;


import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/*
* 使用注解
* 尤其注意methodName="show"的时候，不要写"show()"
* */
@Anno(className = "cn.annotation.reflect",methodName = "show")
public class AnnotationDemo01 {

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

        //1：使用字节码文件获取指定的注解，进而获取注解中的内容

        /*
        * 1.1
        * 获取该类的字节码文件
        * */
        Class<AnnotationDemo01> annotationDemo01Class = AnnotationDemo01.class;

        /*
        * 1.2
        * 使用该类的字节码文件获取指定的注解，就是相当于这个接口有了一个实现类
        * */
        Anno annotation = annotationDemo01Class.getAnnotation(Anno.class);

        /*
        * 1.3
        * 使用该注解对象中定义的抽象方法货的返回值
        * 其实就是相当于这个实现类调用方法
        * */
        String s = annotation.className();
        String s1 = annotation.methodName();

        //2：使用数据执行方法
        /*
        * 2.1
        * 加载类进内存
```

```
        * */
        Class cls = Class.forName(s);

        /*
        * 2.2
        * 创建对象
        * */
        Object obj = cls.newInstance();

        /*
        * 2.3
        * 获取方法
        * */
        Method method = cls.getMethod(s1);
        /*
        * 2.4
        * 使用方法
        * */
        method.invoke(obj);

    }
}
```

```
* 案例：简单的测试框架
* 小结：
    1. 以后大多数时候，我们会使用注解，而不是自定义注解
    2. 注解给谁用？
        1. 编译器
        2. 给解析程序用
    3. 注解不是程序的一部分，可以理解为注解就是一个标签
```