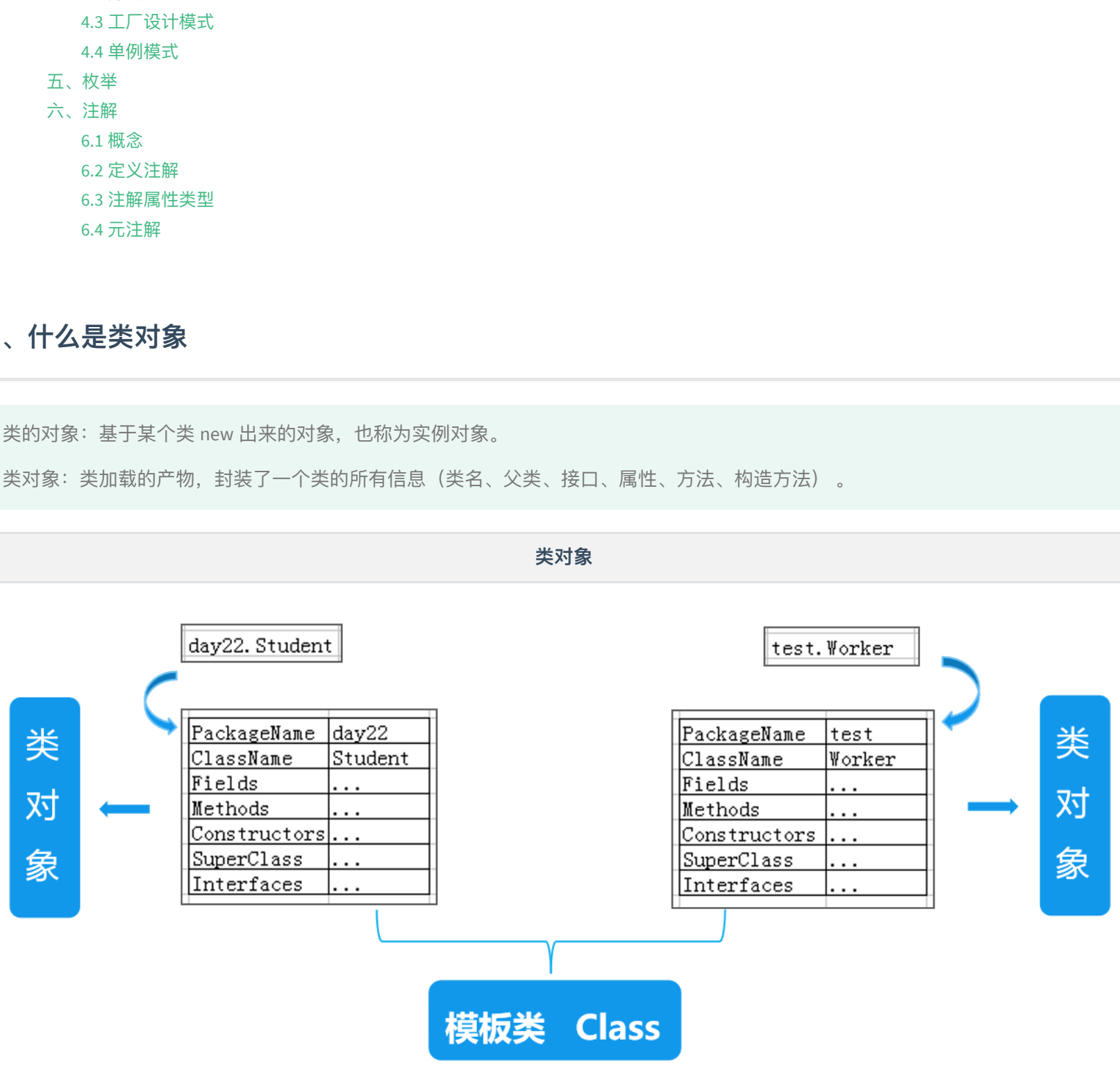
反 射

> Author：zhangzhang
>
> Version：1.0.0

一、什么是类对象

> 类的对象：基于某个类 new 出来的对象，也称为实例对象。
>
> 类对象：类加载的产物，封装了一个类的所有信息（类名、父类、接口、属性、方法、构造方法） 。

类对象


> 注意：每个类加载到内存都会生成一个唯一的类对象。

二、获取类对象的方法

> - 通过类的对象，获取类对象。

```
Student s = new Student();
Class c = s.getClass();
```

> - 通过类名获取类对象。

```
Class c = 类名.class;
```

> - 通过静态方法获取类对象。

```
Class c=Class.forName(“包名.类名”);
```

三、反射通用操作【 重点 】

3.1 常见方法

方法名	描述
public String getName()	获取类的完全名称
public Package getPackage()	获取包信息
public Class<? super T> getSuperclass()	获取父类
public Class<?>[] getInterfaces()	获取实现父接口
public Field[] getFields()	获取字段信息
public Method[] getMethods()	获取方法信息
public Constructor<?>[] getConstructors()	获取构造方法
public T newInstance()	反射创建对象

3.2 通用操作

> 反射通用操作：使用反射机制获取类对象，并使用Class对象的方法获取表示类成员的各种对象（比如Constructor、Method、Field等），实现反射各种应用。

> 案例演示：反射操作。

> Person类：

```
public class Person implements Serializable,Cloneable{
  //姓名
  private String name;
  //年龄
  private int age;

  public Person() {
    System.out.println("无参构造执行了...");
  }

  public Person(String name, int age) {
    super();
    this.name = name;
    this.age = age;
    System.out.println("带参构造方法执行了...");
  }


  //吃
  public void eat() {
    System.out.println(name+"正在吃东西......");
  }


  @Override
  public String toString() {
    return "Person [name=" + name + ", age=" + age + "]";
  }
  //带参的方法
  public void eat(String food) {
    System.out.println(name+"开始吃...."+food);
  }

  //私有的方法
  private void privateMethod() {
    System.out.println("这是一个私有方法");
  }

  //静态方法
  public static void staticMethod() {
    System.out.println("这是一个静态方法");
  }
}
```

> TestPerson类：

```
public class TestPerson {
  public static void main(String[] args) throws Exception {
    //调用测试以下方法
        //代码略
  }
  //获取类对象的三种方式
  public static void getClazz() throws Exception {
    //1使用对象获取类对象
    Person zhangsan=new Person();
```

```
    Class<?> class1=zhangsan.getClass();
    System.out.println(class1.hashCode());
    //2使用类名.class属性
    Class<?> class2=Person.class;
    System.out.println(class2.hashCode());
    //3使用Class的静态方法[推荐使用]
    Class<?> class3=Class.forName("com.qf.chap17_1.Person");
    System.out.println(class3.hashCode());
  }

  //1 使用反射获取类的名字、包名、父类、接口
  public static void reflectOpe1() throws Exception {
    //(1)获取类对象 Person
    Class<?> class1=Class.forName("com.qf.chap17_1.Person");
    //getName();
    System.out.println(class1.getName());
    //getPackage();
    System.out.println(class1.getPackage().getName());
    //getSuperClass();
    System.out.println(class1.getSuperclass().getName());
    //getInterfaces();
    Class<?>[] classes=class1.getInterfaces();
    System.out.println(Arrays.toString(classes));

    System.out.println(class1.getSimpleName());
    System.out.println(class1.getTypeName());

  }

  //2使用反射获取类的构造方法，创建对象
  public static void reflectOpe2() throws Exception{
    //(1)获取类的类对象
    Class<?> class1=Class.forName("com.qf.chap17_1.Person");
    //(2)获取类的构造方法 Constructor
    Constructor<?>[] cons=class1.getConstructors();
    for (Constructor<?> con : cons) {
      System.out.println(con.toString());
    }
    //(3)获取类中无参构造
    Constructor<?> con=class1.getConstructor();
    Person zhangsan=(Person)con.newInstance();
    Person lisi=(Person)con.newInstance();
    System.out.println(zhangsan.toString());
    System.out.println(lisi.toString());
    //简便方法:类对象.newInstance();
    Person wangwu=(Person)class1.newInstance();
    System.out.println(wangwu.toString());
    //(4)获取类中带参构造方法
    Constructor<?> con2=class1.getConstructor(String.class,int.class);
    Person xiaoli=(Person)con2.newInstance("晓丽",20);
    System.out.println(xiaoli.toString());

  }

  //3使用反射获取类中的方法，并调用方法
  public static void reflectOpe3() throws Exception{
    // (1) 获取类对象
    Class<?> class1=Class.forName("com.qf.chap17_1.Person");
    // (2) 获取方法  Method对象
    //2.1getMethods() 获取公开的方法，包括从父类继承的方法
    //Method[] methods=class1.getMethods();
    //2.2getDeclaredMethods() 获取类中的所有方法，包括私有、默认、保护的 、不包含继承的方法
    Method[] methods=class1.getDeclaredMethods();
    for (Method method : methods) {
      System.out.println(method.toString());
    }
    // (3) 获取单个方法
    //3.1eat
    Method eatMethod=class1.getMethod("eat");
    //调用方法
    //正常调用方法  Person zhangsan=new Person();  zhangsan.eat();
    Person zhangsan=(Person)class1.newInstance();
    eatMethod.invoke(zhangsan);//zhangsan.eat();
    System.out.println("-------------------");
    //3.2toString
    Method toStringMethod=class1.getMethod("toString");
    Object result=toStringMethod.invoke(zhangsan);
    System.out.println(result);
    System.out.println("-------------------");
    //3.3带参的eat
    Method eatMethod2=class1.getMethod("eat", String.class);
    eatMethod2.invoke(zhangsan, "鸡腿");

    //3.4获取私有方法
    Method privateMethod=class1.getDeclaredMethod("privateMethod");
    //设置访问权限无效
    privateMethod.setAccessible(true);
    privateMethod.invoke(zhangsan);
```

```
    //3.4获取静态方法
    Method staticMethod=class1.getMethod("staticMethod");
    //正常调用 Person.staticMethod
    staticMethod.invoke(null);

  }

  //4使用反射实现一个可以调用任何对象方法的通用方法
  public static Object invokeAny(Object obj,String methodName,Class<?>[] types,Object...args) throws Exception {
    //1获取类对象
    Class<?> class1=obj.getClass();
    //2获取方法
    Method method=class1.getMethod(methodName, types);
    //3调用
    return method.invoke(obj, args);
  }

  //5使用反射获取类中的属性
  public static void reflectOpe4() throws Exception{
    // (1) 获取类对象
    Class<?> class1=Class.forName("com.qf.chap17_1.Person");
    // (2) 获取属性(字段) 公开的字段, 父类继承的字段
    //Field[] fields=class1.getFields();
    //getDeclaredFields()获取所有的属性, 包括私有, 默认 , 包含,
    Field[] fields=class1.getDeclaredFields();
    System.out.println(fields.length);
    for (Field field : fields) {
      System.out.println(field.toString());
    }
    // (3) 获取name属性
    Field namefield=class1.getDeclaredField("name");
    namefield.setAccessible(true);
    // (4) 赋值  正常调用  Person zhangsan=new Person(); zhangsan.name="张三";
    Person zhangsan=(Person)class1.newInstance();
    namefield.set(zhangsan, "张三"); //zhangsan.name="张三";
    // (5)  获取值
    System.out.println(namefield.get(zhangsan));// zhangsan.name
  }
}
```

四、设计模式【 重点 】

4.1 概念

> - 一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。
> - 可以简单理解为特定问题的固定解决方法。
> - 在Gof的《设计模式》书中描述了23 种设计模式。

4.2 好处

> 使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、重用性。

4.3 工厂设计模式

> 开发中有一个非常重要的原则“开闭原则”，对拓展开放、对修改关闭。
>
> 工厂模式主要负责对象创建的问题。
>
> 可通过反射进行工厂模式的设计，完成动态的对象创建。

> 案例演示：

> Usb类：

```
/**
 * 父类产品
 * @author wgy
 *
 */
public interface Usb {
  void service();
}
```

> Mouse类：

```
public class Mouse implements Usb{

  @Override
  public void service() {
    System.out.println("鼠标开始工作了.....");
  }

}
```

Fan类：

```
public class Fan implements Usb{

  @Override
  public void service() {
    System.out.println("风扇开始工作了...");
  }

}
```

KeyBoard类：

```
public class KeyBoard implements Usb{

  @Override
  public void service() {
    System.out.println("键盘开始工作了...");
  }

}
```

UsbFactory类：

```
public class UsbFactory {
  public static Usb createUsb(String type) {//类型的全名称 com.qf.
    Usb usb=null;
    Class<?> class1=null;
    try {
      class1 = Class.forName(type);
      usb=(Usb)class1.newInstance();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }

    return usb;
  }
}
```

TestUsb测试类：

```
public class TestUsb {
  public static void main(String[] args) throws Exception{
    System.out.println("==========请选择 1 鼠标  2风扇 3 u盘===========");
    Scanner input=new Scanner(System.in);
    String choice=input.next();
    //1 = com.qf.chap17_2.Mouse
    //2 = com.qf.chap17_2.Fan
    //3 = com.qf.chap17_2.Upan
    //4 = com.qf.chap17_2.KeyBoard
    Properties properties=new Properties();
    FileInputStream fis=new FileInputStream("src\\usb.properties");
    properties.load(fis);
    fis.close();

    Usb usb=UsbFactory.createUsb(properties.getProperty(choice));
    if(usb!=null) {
      System.out.println("购买成功");
      usb.service();
    }else {
      System.out.println("购买失败，您要购买的产品不存在");
    }
  }
}
```

usb.properties配置文件。

```
1=com.qf.chap17_2.Mouse
2=com.qf.chap17_2.Fan
3=com.qf.chap17_2.Upan
4=com.qf.chap17_2.KeyBoard
```

4.4 单例模式

> 单例（Singleton）：只允许创建一个该类的对象。

> 方式一：饿汉式（类加载时创建，天生线程安全）。

```
public class SingleTon {
  private static final SingleTon instance=new SingleTon();
  private SingleTon() {}
  public static SingleTon getInstance() {
    return instance;
  }
}
```

> 方式二：懒汉式（使用时创建，线程不安全，加同步）。

```
public class SingleTon2 {
  // 创建对象
  private static SingleTon2 instance = null;
  // 私有化构造方法
  private SingleTon2() {
  }

  // 静态方法
  public static  SingleTon2 getInstance() {
    if(instance==null) {//提高执行效率
      synchronized (SingleTon2.class) {
        if (instance == null) {
          instance = new SingleTon2();
        }
      }
    }
    return instance;
  }
}
```

> 方式三：懒汉式（静态内部类写法）

```
public class SingleTon3 {
  private SingleTon3() {}

  private static class Holder{
    static SingleTon3 instance=new SingleTon3();
  }

  public static SingleTon3 getInstance() {
    return Holder.instance;
  }
}
```

五、枚举

> 枚举是一个引用类型，枚举是一个规定了取值范围的数据类型。

> - 枚举变量不能使用其他的数据，只能使用枚举中常量赋值，提高程序安全性。
> - 定义枚举使用enum关键字。
> - 枚举的本质：
>   - 枚举是一个终止类，并继承Enum抽象类。
>   - 枚举中常量是当前类型的静态常量。

> 案例演示：

```
/**
 * 性别枚举
 */
public enum Gender {
  MALE,FEMALE;
}
```

注意：

- **枚举中必须要包含枚举常量,也可以包含属性、方法、私有构造方法。**
- **枚举常量必须在前面，多个常量之间使用逗号隔开，最后分java号可写可不写。**

六、注解

6.1 概念

> 注解(Annotation)：是代码里的特殊标记, 程序可以读取注解，一般用于替代配置文件。
>
> 开发人员可以通过注解告诉类如何运行。
>
> - 在Java技术里注解的典型应用是：可以通过反射技术去得到类里面的注解，以决定怎么去运行类。

6.2 定义注解

> 定义注解使用@interface关键字，注解中只能包含属性。
>
> 常见注解：@Override、@Deprecated

> 案例演示：

```
public @interface MyAnnotation {
  //属性(类似方法)
  String name() default "张三";
  int age() default 20;

}
```

6.3 注解属性类型

> - String类型
> - 基本数据类型
> - Class类型
> - 枚举类型
> - 注解类型
> - 以上类型的一维数组

6.4 元注解

> 元注解：用来描述注解的注解。

> @Retention:用于指定注解可以保留的域。
>
> - RetentionPolicy.CLASS：
>   注解记录在class文件中，运行Java程序时, JVM不会保留，此为默认值。
> - RetentionPolicy.RUNTIME：
>   注解记录在 class文件中，运行Java程序时，JVM会保留，程序可以通过反射获取该注释
> - RetentionPolicy.SOURCE:
>   编译时直接丢弃这种策略的注释。

> @Target：
>
> - 指定注解用于修饰类的哪个成员。

> 案例演示：

> PersonInfo注解类:

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value= {ElementType.METHOD})
public @interface PersonInfo {
  String name();
  int age();
  String sex();
}
```

> Person类：

```
public class Person {

  @MyAnnotation()
  public void show() {

  }

  //@MyAnnotation2(value="大肉",num=25)
  public void eat() {

  }

  @PersonInfo(name="小岳岳",age=30,sex="男")
  public void show(String name,int age,String sex) {
    System.out.println(name+"==="+age+"====="+sex);
```

```
  }

}
```

TestAnnotation类：

```
public class Demo {
  public static void main(String[] args) throws Exception{
    //(1)获取类对象
    Class<?> class1=Class.forName("com.qf.chap17_5.Person");
    //(2)获取方法
    Method method=class1.getMethod("show", String.class,int.class,String.class);
    //(3)获取方法上面的注解信息 personInfo=null
    PersonInfo personInfo=method.getAnnotation(PersonInfo.class);
    //(4)打印注解信息
    System.out.println(personInfo.name());
    System.out.println(personInfo.age());
    System.out.println(personInfo.sex());
    //(5)调用方法
    Person yueyue=(Person)class1.newInstance();
    method.invoke(yueyue, personInfo.name(),personInfo.age(),personInfo.sex());

  }
}
```