反射

Class类的使用
方法的反射
成员变量的反射
构造函数的反射
java类加载机制

为什么要用反射呢？

反射，就是对类的分析解剖，可以让程序自己去分析类。class之后，用字符串就可以动态调用与处理类的属性与方法。如果没有反射，你只能通过api调用。就变得非常不灵活。
比如你要做一个通用的批量导入，传什么类，就导入什么字段的数据。这时你只能通过反射动态获取字段。并对字段做相应的处理。没有反射，你只能写死为一个类，做不了通用。

使用反射要注意的地方：耗费资源，要在关键的地方使用，至于为什么不是使用new呢，是因为，对于程序的开发模式之前一直强调：尽量减少耦合，而减少耦合的最好做法是使用接口，但是就算使用了接口也逃不出关键字**new**，所以实际上**new**是造成耦合的关键元凶。

如果现在接口的子类增加了，那么工厂类肯定需要修改，这是它所面临的最大问题，而这个最大问题造成的关键性的病因是**new**，那么如果说现在不使用关键字**new**了，变为了反射机制呢？

反射机制实例化对象的时候实际上只需要“包.类”就可以，

这是具体代码

```
package java2;

public class FactoryDemo {
public static void main(String[] args) {
Fruit f = Factory.getInstance("java2.Orange") ;
f.eat() ;
 }
}

interface Fruit { public void eat() ;
}

class Apple implements Fruit {
public void eat() {
System.out.println("吃苹果。");
}
}


class Orange implements Fruit {
public void eat() {
System.out.println("吃橘子。");
 }
 }


class Factory {
public static Fruit getInstance(String className) {
Fruit f = null;
try{
f = (Fruit) Class.forName(className).newInstance() ;
            } catch(Exception e) {
e.printStackTrace();
            }
return f ;
 }
 }
```

发现，这个时候即使增加了接口的子类，工厂类照样可以完成对象的实例化操作，这个才是真正的工厂类，可以应对于所有的变化。如果单独从开发角度而言，与开发者关系不大，但是对于日后学习的一些框架技术这个就是它实现的命脉，在日后的程序开发上，如果发现操作的过程之中需要传递了一个完整的“包.类”名称的时候几乎都是反射机制作用。

使用反射机制也可以取得类之中的构造方法
调用无参构造方法实例化对象要比调用有参构造的更加简单、方便，所以在日后的所有开发之中，凡是有简单**Java**类出现的地方，都一定要提供无参构造。

1 **在面向对象的世界里，万事万物都是对象**
类是对象，类是java.lang.Class的对象

任何一个类都是Class的实例对象，这个实例对象有三种表现方式
Class c1=Test.class

Clss.forName('类的全称')
1 不仅表示了类的类类型,还代表了动态加载类
2 请大家区分编译，运行
3 编译时刻加载类是静态加载类。运行时刻加载类是动态加载类

```
Foo foo=new Foo();
//Class的表示方法
//第一种表示
Class c1=Foo.class;
//第二种表示 已知该类的对象通过getClass方法
Class c2=foo.getClass();
//c1,c2都表示了Foo类的类类型
System.out.println(c1==c2);
//官网c1,c2表现了Foo类的类类型（class type ）
//第三种方式
try {
Class c3=Class.forName("com.imooc.reflect.Foo");
}
```

类类型
静态加载类：编译时加载的类（用new关键字创建的类）
动态加载类：运行时加载的类（用类类型创建的类）
如 Class c = Class.forName("类全名");就是动态加载。
接口的引用 = c.newInstance();//创建实例对象，需要有无参构造方法。

编译时加载类是静态加载类，运行时加载类是动态加载类。
Class.forName("类的全称")不仅表示了类的类类型，还代表了动态加载类。

```
class OfficeBetter
{

   public static void main(String[] args)

   {
      try{
         //动态加载类，在运行时加载
         Class c= Class.forName(args[0]);
                  //通过类类型，创建该类的对象
        (OfficeAble) oa= (OfficeAble)c.newInstance();
                     oa.start();
         }catch(Exception e)
      {
          e.printStackTrace();
      }

   }

}
```

尽量使用动态加载类

4 基本的数据类型
void关键字 都存在类类型
5 Class类的基本API操作
Method类，是方法对象 一个成员方法就是一个Method对象
getMethods()这个方法获取的是所有的public的函数，包括父类继承而来
getDeclaredMethods()获取的是所有该类自己声明的方法 ，不问访问权限
成员变量也是对象 java.lang.reflect.Field Field类封装了关于成员变量的操作
getFields();获取的是所有的public的成员变量信息 getDeclaredFields获取的是该类自己声明的成员变量的信息
构造方法也是对象 java.lang.Constuctor getConstuctor获取的是所有的public的构造函数
getDeclaredConstructors获取的是所有的构造函数
getType()是获得类型

getSimpleName()获得的是包名
getName()返回的实体的名称

eclipse中有重构这个功能，可以抽取出来单独作为方法，参数，一些列功能

静态方法是使用公共内存空间的，就是说所有对象都可以直接引用，不需要创建对象再使用该方法。

因此静态方法中不能用this和super关键字，不能直接访问所属类的实例变量和实例方法(

就是不带static的成员变量和成员成员方法)

只能访问所属类的静态成员变量和成员方法。

static final用来修饰成员变量和成员方法，可简单理解为“全局常量”

静态的属性不依赖于类的存在而存在

静态变量是跟类相关联的，类的所有实例共同拥有一个静态变量。

方法的反射

1 方法的名称与方法的参数才能唯一决定一个方法

方法反射的操作

2 method.invode（对象，参数列表）

通过反射了解集合泛型的本质
通过Class,Method来认知泛型的本质

：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。
自定义泛型方法

```
package generic;

/**
 * 泛型方法测试
 *
 * @author caiyu
 *
 */
public class GenMethod {

    public static <T> void display(T t) {
        System.out.println(t.getClass());
    }
}
```

首先，泛型的声明，必须在方法的修饰符（public,static,final,abstract等）之后，返回值声明之前。

然后，和泛型类一样，可以声明多个泛型，用逗号隔开。

先看看效果，调用display

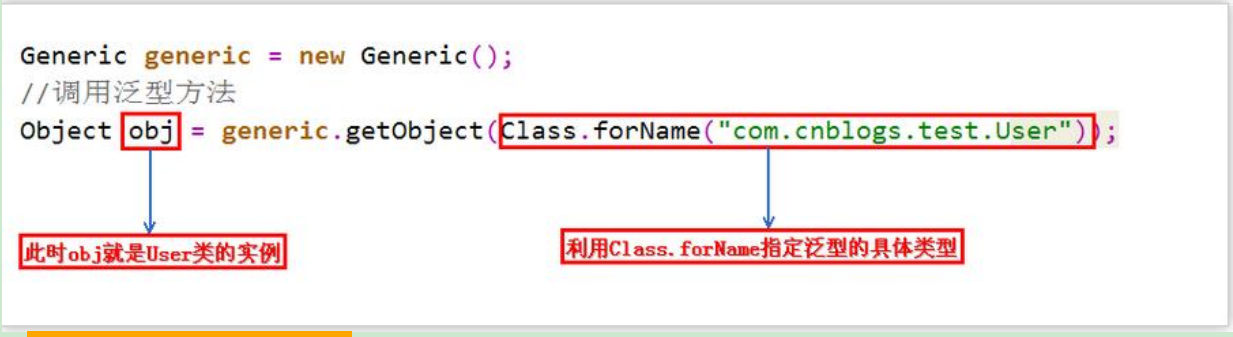
```
/**
* 泛型方法测试
*/

GenMethod.display(123);
GenMethod.display("");
GenMethod.display(123f);
```

结果为：

```
class java.lang.Integer
class java.lang.String
class java.lang.Float
```

自定义泛型方法

调用泛型方法语法格式如下：




```
public class GenericGoods<T>
{

   private String information;
   private   T   t;
   public GenericGoods(T oT)
   {
      this.t = oT;
   }


   public void setData(String sBrand, String sName, String sPrice)
   {
      this.information = "This " + sName + " of " + sBrand + " costs "
            + sPrice + "!";
   }
   public String getClassType()
   {
      return t.getClass().getName();
   }
//省略了set/get方法
}
```

Java集合的泛型，是防止错误输入的，只在编译阶段有效，绕过编译就无效了