1 批处理文件(bat)

简单的说，批处理的作用就是自动的连续执行多条命令 .编写 bat 处理文件可以使用记事本的方式:
常见批处理文件的命令：

echo 表示显示此命令后的字符

tiltle 设置窗口的标题。

echo off 表示在此语句后所有运行的命令都不显示命令行本身

color 设置窗体的字体颜色。

@与 echo off 相象，但它是加在每个命令行的最前面，表示运行时不显示这一行的命令行（只能影响当前行）。

pause 运行此句会暂停批处理的执行并在屏幕上显示 Press any key to continue... 的提示，等待用户按任意键后继续

rem 表示此命令后的字符为解释行（注释），不执行，只是给自己今后参考用的（相当于程序中的注释） 或者%注释的内容%

%[1-9]表示参数，参数是指在运行批处理文件时在文件名后加的以空格（或者 Tab）分隔的字符串

2 对象拷贝

2.1 对象的浅拷贝

浅复制(浅克隆)被复制对象的所有变量都含有与原来对象相同的值，而所有的对其他对象的引用仍然只指向原来的对象，换言之，浅复制仅仅复制锁考虑的对象，而不复制它所引用的对象。

```
public class Student implements Cloneable{
        String name;
        int age;
    Student(String name,int age){
         this.name=name;
         this.age=age;
    }

    public Object clone(){
         Object o =null;
         try{
          o=super.clone();//Object中的clone()识别出你要复制的哪一个对象
         }
         catch(CloneNotSupportedException e){
          System.out.println(e.toString());
         }
         return o;
    }

 public static void main(String[] args){
      Student s1 = new Student("zhang",18);
      Student s2 = (Student)s1.clone();
      s2.name="li";
      s2.age=20;
      System.out.println("name="+s1.name+","+"age="+s1.age);//修改学生2
后不影响学生1的值
   }
}
```

2.2 对象深拷贝

深复制(深克隆)被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量，那些引用其他对象的变量将指向被复制过的新对象，而不再试原有的那些被引用的对象，换言之，深复制把要复制的对象所引用的对象都复制了一遍。

把对象写到流里的过程是串行化（Serilization）过程，但是在 Java 程序师圈子里又非常形象地称为“冷冻”或者“腌咸菜（picking）”过程；而把对象从流中读出来的并行化（Deserialization）过程则叫做“解冻”或者“回鲜(depicking)”过程。应当指出的是，写

在流里的是对象的一个拷贝，而原对象仍然存在于 JVM 里面，因此“腌成咸菜”的只是对象的一个拷贝，Java 咸菜还可以回鲜。

在 Java 语言里深复制一个对象，常常可以先使对象实现 Serializable 接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里（腌成咸菜），再从流里读出来（把咸菜回鲜），便可以重建对象。

```
public Object deepClone()
{
 //将对象写到流里
     ByteArrayOutoutStream bo=new ByteArrayOutputStream();
     ObjectOutputStream oo=new ObjectOutputStream(bo);
     oo.writeObject(this);
     //从流里读出来
  ByteArrayInputStream bi=new ByteArrayInputStream(bo.toByteArray());
     ObjectInputStream oi=new ObjectInputStream(bi);
     return(oi.readObject());
}
```

1.内存溢出

由于Java具备自动的垃圾回收机制,当我们使用完对象之后,它们会被自动回收,是不是我们在 Java 程序中不需要再考虑内存管理了吗?

请看如下程序:

```
class Stack {
    private Object[] elements;
    // 初始化角标
    int index = 0;
    // 默认初始化容量
    private int initialCapacity = 10;

    public Stack() {
        elements = new Object[initialCapacity];
    }

    // 压栈 push
    public void push(Object e) {
        ensureCapacity();
        elements[index++] = e;
        // System.out.println(index);
    }
```

```
    // 弹栈 pop
    public Object pop() {
        if (index == 0) {
            throw new RuntimeException("没有元素");
        }
        return elements[--index];
    }

    private void ensureCapacity() {
        if (index == elements.length) {
            elements = Arrays.copyOf(elements, index * 2 + 1);
        }
    }
}
```

注意:从栈中弹出的对象不会作为垃圾回收,即使程序不再使用这些对象,因为栈内部继续维护着这些对象.最终可能会导致内存占用的不断增加,程序性能降低.这就是内存泄漏.

改进版本

```
class Stack {
    private Object[] elements;
    // 初始化角标
    int index = 0;
    // 默认初始化容量
    private int initialCapacity = 10;

    public Stack() {
        elements = new Object[initialCapacity];
    }

    // 压栈 push
    public void push(Object e) {
        ensureCapacity();
        elements[index++] = e;
        // System.out.println(index);
    }

    // 弹栈 pop
    public Object pop() {
        if (index == 0) {
            throw new RuntimeException("没有元素");
        }
        Object obj = elements[--index];
        elements[index] = null;
        return obj;
```

```
    }

    private void ensureCapacity() {
        if (index == elements.length) {
            elements = Arrays.copyOf(elements, index * 2 + 1);
        }
    }
}
```

2.设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。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。

2.1.观察者模式

有时又被称为
发布-订阅<Publish/Subscribe>模式、
模型-视图<Model/View>模式、
源-收听者<Source/Listener>模式
或从属者<Dependents>模式）
这是软件设计模式的一种。
观察者模式（Observer）完美的将观察者和被观察的对象分离开。
此种模式中，一个目标物件管理所有相依于它的观察者物件，并且在它本身的状态改变时主动发出通知。
这通常透过呼叫各观察者所提供的方法来实现。
此种模式通常被用来实作事件处理系统。
有多个观察者时，不可以依赖特定的通知次序。
Swing 大量使用观察者模式，许多 GUI 框架也是如此。
气象站：

```java
public class WeatherStation {

    private String weather;

    String[] weathers = {"下雨","下雪","下冰雹","出太阳"};

    static List<BookWeather> list = new ArrayList<BookWeather>();

    Random random = new Random();


    public void startWork(){

        new Thread(){

            @Override
            public void run() {
                while(true){
                    updateWeather();
                    try {
                        Thread.sleep(random.nextInt(1000)+500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }
    public void updateWeather(){
        weather = weathers[random.nextInt(4)];
        System.out.println("天气: "+ weather);
    }


    public String getWeather() {
        return weather;
    }
```

人：

```java
public class Person implements BookWeather {

    String name;

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

    private WeatherStation station ;

    public Person(String name,WeatherStation station){
        this(name);
        this.station = station;
    }

    //下雨","下雪 ","下冰雹","出太阳"
    @Override
    public void notifyWeather() {
        String weather = station.getWeather();
        if("下雨".equals(weather)){
            System.out.println(name+"打着雨伞上班");
        }else if("下雪".equals(weather)){
            System.out.println(name+"溜冰 上班");
        }else if("下冰雹".equals(weather)){
            System.out.println(name+"带着头盔 上班");
        }else if("出太阳".equals(weather)){
            System.out.println(name+"晒着太阳 上班");
        }
    }

}
```

测试类：

```
public class Test {

    public static void main(String[] args) throws InterruptedException
{
        WeatherStation station = new WeatherStation();
        station.startWork();

        Person p1 = new Person("小明",station);
        while(true){
            p1.notifyWeather();
            Thread.sleep(2000);
        }
    }
```

问题：天气变化两三次，小明才知道一次。

解决方案 ：

```
package cn.itcast.test;
import java.util.List;
import java.util.ArrayList;
import java.util.Random;
public class WeatherStation {

    private String weather;

    String[] weathers = {"下雨","下雪","下冰雹","出太阳"};

    private static    List<BookWeather> list = new
ArrayList<BookWeather>();

    Random random = new Random();

    public void addListaner(BookWeather e){
        list.add(e);
    }

    public void startWork(){

        new Thread(){

            @Override
            public void run() {
                while(true){
                    updateWeather();
                    try {
```

```
                    Thread.sleep(random.nextInt(1000)+500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }.start();
}
public void updateWeather(){
    weather = weathers[random.nextInt(4)];
    System.out.println("天气："+ weather);
    for(BookWeather item : list){
        item.notifyWeather(weather);
    }
}


public String getWeather() {
    return weather;
}
```

人：

```java
public class Person implements BookWeather {

    String name;

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

    private WeatherStation station ;

    public Person(String name,WeatherStation station){
        this(name);
        this.station = station;
    }

    //下雨","下雪 ","下冰雹","出太阳"
    @Override
    public void notifyWeather(String weather) {
        if("下雨".equals(weather)){
            System.out.println(name+"打着雨伞上班");
        }else if("下雪".equals(weather)){
            System.out.println(name+"溜冰 上班");
        }else if("下冰雹".equals(weather)){
            System.out.println(name+"带着头盔 上班");
        }else if("出太阳".equals(weather)){
            System.out.println(name+"晒着太阳 上班");
        }
    }


}
```

接口:

```java
public interface BookWeather {

    public void notifyWeather(String weather);
}
```

```
public class Test {

    public static void main(String[] args) throws InterruptedException
{
        WeatherStation station = new WeatherStation();
        station.startWork();

        Person p1 = new Person("小明");
        Person p2 = new Person("小红");
        Person p3 = new Person("小青 ");
        station.addListaner(p1);
        station.addListaner(p2);
        station.addListaner(p3);

    }
}
```

2.2. 单例

Singleton

是指只能拥有一个实例的类就是单例类。

私有构造方法。

获取方式

通过公共的静态方法创建单一的实例。

两种模式

懒汉模式 – 通常被称为延迟加载。注意存在线程安全问题.

饿汉模式

懒汉式的单例模式线程安全问题的解决方案：

```
    class Single{

    //声明本类的一个私有的成员变量
    private static Single single;

    //第一步 ：私有化构造方法
    private Single(){

    }
     //  第三步：提供一个公共的方法获取该类的实例对象
    public static Single getInstance(){
        if(single==null){
        synchronized (single) {
                if(single==null){
                     single = new Single();
                }
            }
        }
        return single;
    }
}
```

3.反射

类字节码文件是在硬盘上存储的，是一个个的.class 文件。我们在 new 一个对象时，JVM 会先把字节码文件的信息读出来放到内存中，第二次用时，就不用在加载了，而是直接使用之前缓存的这个字节码信息。

字节码的信息包括：类名、声明的方法、声明的字段等信息。在 Java 中“万物皆对象”，这些信息当然也需要封装一个对象，这就是 Class 类、Method 类、Field 类。

通过 Class 类、Method 类、Field 类等等类可以得到这个类型的一些信息，甚至可以不用 new 关键字就创建一个实例，可以执行一个对象中的方法，设置或获取字段的值，**这就是反射技术**。

3.1. Class 类

1.1.1. 获取 Class 对象的三种方式

Java 中有一个 Class 类用于代表某一个类的字节码。

Java 提供了三种方式获取类的字节码

forName()。forName 方法用于加载某个类的字节码到内存中，并使用 class 对象进行封装

类名.class

对象.getClass()

```
/**
     * 加载类的字节码的3种方式
     * @throws Exception
     * */
    public void test1() throws Exception {
        // 方式一
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        // 方式二
        Class clazz2 = Person.class;
        // 方式三
        Person p1 = new Person();
        Class clazz3 = p1.getClass();
    }
```

1.1.2. 通过 Class 类获取类型的一些信息

1. getName()类的名称（全名，全限定名）
2 getSimpleName()类的的简单名称（不带包名）
3. getModifiers(); 类的的修饰符
4.创建对象

无参数构造创建对象

newInstance()

5. 获取指定参数的构造器对象，并可以使用Constructor对象创建一个实例

Constructor<T> getConstructor(Class<?>... parameterTypes)

```
/**
     * 通过Class对象获取类的一些信息
     *
     * @throws Exception
     * */
    private static void test2() throws Exception {

        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        // 获取类的名称
        String name = clazz1.getName();
        System.out.println(name); //
```

```
cn.itcast.gz.reflect.Person
        // 获取类的简单名称
        System.out.println(clazz1.getSimpleName()); // Person
        // 获取类的修饰符
        int modifiers = clazz1.getModifiers();
        System.out.println(modifiers);
        // 构建对象(默认调用无参数构造.)
        Object ins = clazz1.newInstance();
        Person p = (Person) ins;
        System.out.println(p); //
cn.itcast.gz.reflect.Person@c17164
        // 获取指定参数的构造函数
        Constructor<?> con = clazz1.getConstructor(String.class,
int.class);
        // 使用Constructor创建对象.
        Object p1 = con.newInstance("jack", 28);
        System.out.println(((Person) p1).getName());
    }
```

1.1.3. 通过 Class 类获取类型中的方法的信息

1.获取公共方法包括继承的父类的方法

getMethods()返回一个数组,元素类型是 Method

2.获取指定参数的公共方法

getMethod("setName", String.class);

3.获得所有的方法,包括私有

Method[] getDeclaredMethods()

4.获得指定参数的方法,包括私有

Method getDeclaredMethod(String name, Class<?>... parameterTypes)

```
    /**
     * 获取公有方法.
     * @throws Exception
     * */
    private static void test3() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        // 1.获取非私用方法(包括父类继承的方法)
        Method[] methods = clazz1.getMethods();
        System.out.println(methods.length);
        for (Method m : methods) {
            // System.out.println(m.getName());
            if ("eat".equals(m.getName())) {
```

```
                m.invoke(clazz1.newInstance(), null);
            }
        }

    }
```

```
/**
     * 获取指定方法签名的方法
     *
     * @throws Exception
     * */
    private static void test4() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        // 获取指定名称的函数
        Method method1 = clazz1.getMethod("eat", null);
        method1.invoke(new Person(), null);
    }
```

```
/**
     * 获取指定方法名且有参数的方法
     *
     * @throws Exception
     * */
    private static void test5() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        Method method = clazz1.getMethod("eat", String.class);
        method.invoke(new Person(), "包子");
    }

    /**
     * 获取指定方法名,参数列表为空的方法.
     *
     * @throws Exception
     * */
    private static void test4() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        // 获取指定名称的函数
        Method method1 = clazz1.getMethod("eat", null);
        method1.invoke(new Person(), null);
    }
```

```
/**
     * 反射静态方法
     * @throws Exception
     * */
    private static void test7() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        Method method = clazz1.getMethod("play", null);
        method.invoke(null, null);
    }

    /**
     * 访问私有方法 暴力反射
     * @throws Exception
     * */
    private static void test6() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        Method method = clazz1.getDeclaredMethod("movie",
String.class);
        method.setAccessible(true);
        method.invoke(new Person(), "苍老师");
    }
```

1.1.4. 通过 Class 类获取类型中的字段的信息

1.获取公共字段

```
Field[] getFields()
```

2.获取指定参数的公共字段

```
Field getField(String name)
```

3.获取所有的字段

```
Field[] getDeclaredFields()
```

4.获取指定参数的字段,包括私用

```
Field getDeclaredField(String name)
```

```
/**
     * 获取公有的字段
     * */
    private static void test8() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        Field[] fields = clazz1.getFields();
```

```
        Person p = new Person();
        System.out.println(fields.length);
        for (Field f : fields) {
            System.out.println(f.getName());
            if ("name".equals(f.getName())) {
                System.out.println(f.getType().getName());
                f.set(p, "jack");
            }
        }
        System.out.println(p.getName());

    }
```

```
/**
     * 获取私有的字段
     * @throws Exception
     * */
    private static void test9() throws Exception {
        Class clazz1 =
Class.forName("cn.itcast.gz.reflect.Person");
        Field field = clazz1.getDeclaredField("age");
        System.out.println(field.getName());
        field.setAccessible(true);
        Person p = new Person();
        field.set(p, 100);
        System.out.println(p.getAge());
    }
```

3.2. 工厂模式

Factory
例如:汽车销售商场

该模式将创建对象的过程放在了一个静态方法中来实现.在实际编程中,如果需要大量的创建对象,该模式是比较理想的.

```
public class Demo1 {
    public static void main(String[] args) {
        System.out.println("买宝马");
        Car bmw = CarFactory("BMW");
        bmw.run();
```

```
        System.out.println("买大奔");
        Car benz = CarFactory("Benz");
        benz.run();
    }

    public static Car CarFactory(String carName) {
        if ("BMW".equals(carName)) {
            return new BMW();
        } else if ("Benz".equals(carName)) {
            return new Benz();
        } else {
            throw new RuntimeException("车型有误");
        }
    }
}

abstract class Car {

    public abstract void run();
}

class BMW extends Car {

    @Override
    public void run() {
        System.out.println("BMW跑跑");
    }
}

class Benz extends Car {

    @Override
    public void run() {
        System.out.println("Benz跑跑");
    }
}
```

模拟 spring 工厂：

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
class Student{

    private int id;

    private  String name;

    public Student(int id , String name){
        this.id = id;
        this.name = name;
    }

    public Student(){

    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return this.id +"-"+this.name;
    }

}
class Person{

    private int age;

    public Person(){
```

```
    }



    @Override
    public String toString() {
        return this.age+"";
    }
}
public class Demo1 {

    public static void main(String[] args) throws Exception {
        Object o = getInstance();
        System.out.println(o);

    }

    public static Object getInstance() throws Exception{
        FileReader fileReader = new FileReader("src/info.txt");
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        String line = bufferedReader.readLine();
        Class clazz  = Class.forName(line);
        Constructor c =  clazz.getConstructor(null);
        Object c1 = c.newInstance(null);
        while((line=bufferedReader.readLine())!=null){
            String[] datas =  line.split("=");
            Field f = clazz.getDeclaredField(datas[0]);
            f.setAccessible(true);
            if(f.getType()==int.class){
                f.set(c1, Integer.parseInt(datas[1]));
            }else{
            //f.setAccessible(true);
                f.set(c1,datas[1]);
            }
        }
        return c1;
    }



}
```