

1 包机制

问题：当定义了多个类的时候，可能会发生类名的重复问题。

在 java 中采用包机制处理开发者定义类名冲突问题。

怎么使用 java 的包机制呢？

1. 使用 package 关键字。
2. package 包名。

```
package pack;
class PackageDemo1
{
    public static void main(String[] args)
    {
        System.out.println("Hello package!");
    }
}
```

问题：

1. javac PackDemo1.java 编译没有问题。
2. java PackDemo1 运行出错。

```
Exception in thread "main" java.lang.NoClassDefFoundError: PackageDemo1 (wrong name: pack/PackageDemo1)
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$000(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClassInternal(Unknown Source)
Could not find the main class: PackageDemo1. Program will exit.
```

错误原因分析：

在当前目录下找不到有 pack 目录，更加找不到 pack 目录下面的 PackageDemo1.java 文件。

解决办法：

1. 自己在当前目录下新建一个 pack 目录。
2. 执行 Java pack.PackageDemo1 命令。(包其实就是文件夹)。

存在的问题：使用包机制的话，我们是否每次都要自己创建一个文件夹呢？

解决：

在编译的时候则可以指定类文件存放的文件夹了。

javac -d . PackageDemo1.java -d 后面跟着就是包名，指定包存放的路径。

包的优点

1. 防止类文件冲突。

2. 使源文件与类文件分离，便于软件最终发布。

注意细节

1. 一个 java 类只能定义在一个包中。
2. 包语句肯定是描述类的第一条语句。

包机制引发的问题

有了包之后访问类每次都需要把包名和类名写全。

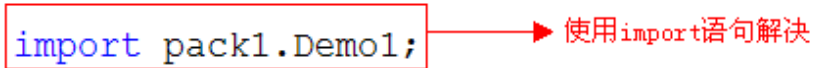
解决:使用 import 语句。

格式: import 包名.类名;

```
package pack2;

import pack1.Demo1;

public class Demo2
{
    public static void main(String[] args)
    {
        Demo1 demo = new Demo1();
        demo.test();
    }
}
```



注意细节:

1. 如果想使用一个包中的许多类时，这时不需要多条的导入语句，使用 “*” 号通配符代表所有的类。
2. 使用*时不能导入包中的子类包的 class 文件。
3. import 语句可以是多条。

2 访问修饰符

访问修饰符是用来控制类、属性、方法的可见性的关键字称之为访问修饰符。

	public	protected	default	private
同一类中	√	√	√	√
同一包中	√	√	√	
子类	√	√		
不同包中	√			

1. public 一个类中，同一包中，子类中，不同包中
2. protected 一个类中，同一包中，子类中
3. default 一个类中，同一包中
4. private 一个类中

1. (修饰类成员) 类成员

1. 成员使用 private 修饰只在本类中使用。
2. 如果一个成员没有使用任何修饰符，就是 default，该成员可以被包中的其他类访问。
3. protected 成员被 protected 修饰可以被包中其他类访问，并且位于不同包中的子类也可以访问。
4. public 修饰的成员可以被所有类访问。

2. (修饰类) 类

1. 类只有两种 public 和默认 (成员内部类可以使用 private)
2. 父类不可以是 private 和 protected，子类无法继承
3. public 类可以被所有类访问
4. 默认类只能被同一个包中的类访问

3 Jar 包

1: **jar** 就是打包文件

jar 文件是一种打包文件 java archive File, 与 zip 兼容，称之为 jar 包。开发了很多类，需要将类提供给别人使用，通常以 jar 包形式提供。当项目写完之后，需要及将 class 字节码文件打包部署给客户。如何打包？可以使用 jar 命令。

2: jar 命令

- 1: jar 工具存放于 jdk 的 bin 目录中 (jar.exe)
- 2: jar 工具：主要用于对 class 文件进行打包 (压缩)
- 3: dos 中输入 jar 查看帮助

3: 案例使用 jar 命令

将 day10 中的 cn 文件打包为名字为 test.jar 文件 (cn 文件是使用 javac -d 编译带包的 class 文件夹)

jar cvf test.jar cn

详细命令:

- 1: jar cf test.jar cn 在当前目录生成 test.jar 文件, 没有显示执行过程
- 2: jar cvf test.jar cn 显示打包中的详细信息
- 3: jar tf test.jar 显示 jar 文件中包含的所有目录和文件名
- 4: jar tvf test.jar 显示 jar 文件中包含的所有目录和文件名大小, 创建时间详细信息
- 5: jar xf test.jar 解压 test.jar 到当前目录, 不显示信息
- 6: jar xvf test.jar 解压 test.jar 到当前目录, 显示详细信息
- 7: 可以使用 WinRAR 进行 jar 解压
- 8: 将两个类文件归档到一个名为 test2.jar 的归档文件中:
jar cvf test2.jar Demo3.class Demo4.class
- 9: 重定向
 - 1: tvf 可以查看 jar 文件内容, jar 文件大, 包含内容多, dos 看不全。
 - 2: 查看 jdk 中的 rt.jar 文件 jar tvf rt.jar
 - 3: jar tvf rt.jar>d:\rt.txt

4 模板设计.

设计模式就是为了解决某类事情提出的解决方法。

案例: 计算一段程序的执行时间

```
模板设计模式
场景: 计算一段代码的运行时间
*/

class Demo5
{
    //计算一段代码的运行时间
    public static void runTime(){
        long startTime = System.currentTimeMillis();
        for(int i = 0 ; i< 10000; i++){
            System.out.println("i : "+i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("该程序运行的时间为: "+(endTime-startTime));
    }

    public static void main(String[] args)
    {
        runTime();
    }
}
```

这就是模板

存在问题:

1. 计算的程序的可变的。
2. 把会改变的程序抽取出来单独做一个方法。
3. 但是该方法不能确定运行的代码, 声明为抽象的方法。
4. 创建实现类继承并实现父类的未实现的函数。

5. 为了避免子类重写父类的模版代码，需要将模版代码修饰为 final

```
abstract class RunCode
{
    // 计算一段代码的运行时间？
    public final void getRuntime(){
        // 获取系统的当前时间 毫秒 一秒 = 1000毫秒
        long start = System.currentTimeMillis();
        // 测试代码
        code();
        // 获取系统的当前时间 毫秒 一秒 = 1000毫秒
        long end = System.currentTimeMillis();
        System.out.println("运行时间: " + ( end - start ) );
    }
    public abstract void code();
}
```

案例二：炒菜做饭

```
abstract class Cook
{
    // 1. 形成完成这项功能的模版代码 5. 将模版代码声明为final避免子类重写
    public final void doCook(){
        // 做饭
        // 3. 在模版代码中调用可变部分的代码
        System.out.println("买" + food() + ".....");
        System.out.println("洗" + food() + ".....");
        System.out.println("炒" + food() + ".....");
        System.out.println("做米饭.....");
        System.out.println("吃饭.....");
        System.out.println("刷碗.....");
    }
    // 2. 将模版代码中可变的数据抽取为一个函数,并修饰为抽象的
    public abstract String food();
}
// 4. 编写子类实现父类的未实现的功能
class MyCook extends Cook
{
    public String food(){
        return "鲑鱼";
    }
}
```