

1.理解Java I/O系统

2.熟练使用java.io包

3.掌握I/O设计原则和使用的设计模式

流是数据的抽象，载体

输入/输出流，流入程序输入流，从程序流出的是输出流

字节流/字符流

InputStream/OutputStream

Reader/Writer

基本流程：

```
1  try(  
2    InputStream is = new FileInputStream(new File("path"));  
3  ){  
4    byte[] arr = new byte[1024];  
5    int len;  
6    while((len=is.read(arr))!=-1){  
7      String str = new String(arr,0,len);  
8      System.out.println(str);  
9    }  
10 }catch(Exception e){  
11   e.printStackTrace();  
12 }
```

节点流//过滤流

过滤流包装节点流（装饰模式）

I/O流的链接

Input Stream Chain



Output Stream Chain



装饰模式

以透明的方式包装被装饰对象，以扩展对象功能，不用继承实现子类

• 装饰模式的角色：

- 抽象构件角色 (Component)：给出一个抽象接口，以规范准备接收附加责任的对象。
- 具体构件角色 (Concrete Component)：定义一个将要接收附加责任的类。
- 装饰角色 (Decorator)：持有一个构件 (Component) 对象的引用，并定义一个与抽象构件接口一致的接口。
- 具体装饰角色 (Concrete Decorator)：负责给构件对象“贴上”附加的责任。

抽象构件角色：InputStream

具体构件角色：FileInputStream

装饰角色：FilterInputStream

具体装饰角色：bufferedInputStream

- 装饰模式的特点：

- 装饰对象和真实对象有相同的接口。这样客户端对象就可以以和真实对象相同的方式和装饰对象交互。
- 装饰对象包含一个真实对象的引用（reference）
- 装饰对象接收所有来自客户端的请求。它把这些请求转发给真实的对象。
- 装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部分增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。

编码实现装饰模式

抽象构件角色（Componnet）

```
1 /**
2  * 抽象构件角色（Componnet）
3  * 比如InputStream
4  */
5 public interface Componnet {
6     void doSomething();
7 }
```

具体构件角色（Concrete Component）

```
1 /**
2  * 具体构件角色（Concrete Component）
3  * 例如FileInputStream
4  */
5 public class ConcreteComponnet implements Componnet {
6     @Override
7     public void doSomething() {
8         System.out.println("功能A");
9     }
10 }
```

装饰角色（Decorator）

```
1 /**
2  * 装饰角色（Decorator）
3  * 比如FilterInputStream
4  */
5 public class Decorator implements Componnet{
```

```

6
7  Componnet componnet;
8
9  public Decorator(Componnet componnet) {
10     this.componnet = componnet;
11 }
12 @Override
13 public void doSomething() {
14     componnet.doSomething();
15 }
16 }

```

具体装饰角色_1 (Concrete Decorator)

```

1  /**
2   * 具体装饰角色_1 (Concrete Decorator)
3   * 比如 BufferedInputStream
4   */
5  public class ConcreteDecorator_1 extends Decorator {
6
7      public ConcreteDecorator_1(Componnet componnet) {
8          super(componnet);
9      }
10
11     @Override
12     public void doSomething() {
13
14         super.doSomething();
15         this.doOtherThing();
16     }
17
18     private void doOtherThing(){
19         System.out.println("功能B");
20     }
21 }

```

具体装饰角色_2 (Concrete Decorator)

```

1  package com.zhangtianyi.nio.Decorator;
2
3  /**
4   * 具体装饰角色_2 (Concrete Decorator)
5   * 比如 DataInputStream
6   */

```

```

7 public class ConcreteDecorator_2 extends Decorator {
8
9 public ConcreteDecorator_2(Componnet componnet) {
10     super(componnet);
11 }
12
13 @Override
14 public void doSomething() {
15     super.doSomething();
16     this.doOtherThing();
17 }
18
19 private void doOtherThing() {
20     System.out.println("功能C");
21 }
22 }

```

测试类

```

1 /**
2  * 测试类
3  */
4 public class Client {
5
6
7 public static void main(String[] args) {
8     //未装饰
9     new ConcreteComponnet().doSomething();
10    System.out.println("-----");
11    //装饰一次
12    Componnet componnet = new ConcreteDecorator_1(new ConcreteComponnet());
13    componnet.doSomething();
14    System.out.println("-----");
15    //装饰2次
16    Componnet componnet2 = new ConcreteDecorator_2(new
ConcreteDecorator_1(new ConcreteComponnet()));
17    componnet2.doSomething();
18    System.out.println("-----");
19 }
20
21 }

```

装饰模式适用性：

透明的增加给对象增加功能，不影响其他对象

给对象增加的功能未来可能改变

给子类扩展功能不实际的情况下