设计模式

一　装饰者设计模式

1　使用场景

给某个对象(注意是对象，而不是类)的功能进行扩展时，可以考虑使用装饰者设计模式。

在 IO 流这部分内容中，JDK 的源码使用了大量的装饰者设计模式。

比如 BufferedReader 可以对 FileReader 进行装饰。

2　实例

2.A　在 Servlet 中每个页面都需要解决编码问题，这时可以创建一个 filter 拦截所有的请求然后统一处理编码问题,最后放行，传入 request 和 response。

2.B　但是有一个问题，当请求为 get 时，处理编码问题有点特殊，需用对每个参数编码

```
String v=request.getParameter(name);
byte[] bytes= v.getBytes("iso8859-1");
//encode="utf-8"
return new String(bytes,enCode);
```

我对每个参数编码后放行时如何传递到请求的页面中，显然这是没办法做到的

2.C　那能不能让请求页面的 request 调用 getParameter(name)方法时，先解决编码问题再返回？

答案是可以的，可以自定义类，改变 getParameter()方法的实现方式，放行时，将该类传递过去，而不是传递 request，所以 requets 和自定义类必须具有相同的父类或父接口。

2.D 实现

2.D.a 自定义接口构造方法应传递一个 request 参数并保存

```
}
private class EncodeServletRequest extends HttpServletRequestWrapper {
    private HttpServletRequest request;
    private EncodeServletRequest(HttpServletRequest request){
        super(request);
        this.request=request;
    }
```

注意：HttpServletRequestWrapper 本身已经是 HttpServlet 的装饰类，所以我们这里只需要继承他重写相关方法，而不用实现 HttpServerlet，重写所以有法，简化操作

2.D.b 实现 getParameter(name)方法

```
    @Override
    public String getParameter(String name) {

        try {
            String v=super.getParameter(name);
            if (v==null){
                return null;
            }
            byte[] bytes= v.getBytes( charsetName: "iso8859-1");
           //encode="utf-8"
            System.out.println(new String(bytes,enCode));
            return new String(bytes,enCode);
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return null;
    }
```

2.D.c 放行传入自定义类

```
    //解决 响应乱码 包括字符流和字节流
    //装饰者设计模式
    filterChain.doFilter(new EncodeServletRequest((HttpServletRequest) servletRequest),servletResponse);
}
```

2.D.d 注意：

请求转发后又经过 filter，经过二次编码又会乱码。

解决：该对象在该请求链有效，所以加个 flag=true，第一次运行进入然后 flag=false，后面不再进入。

二　单例设计模式

常见的单例，只实例化一次
ServletContext 、Filter、Listener、Servlet、连接池
单例可以避免实例对象的重复创建，可以减少每次创建的时间开销，也可以节省内存空间

1　饿汉模式

构造函数私有化，保证不被其他类实例化
在类中创建了静态的实例供别人使用，提供一个获取该实例的方法
因为是静态的，所以在类加载时创建一次实例，从此以后供别人使用,在整个程序周期都存在
为什么叫饿汉？等不及了，非常饿，所以类加载时就创建

优点：类加载时创建实例且只有一个，不存在多线程问题

```
3   public class Singleton {
4       private static Singleton singleton=new Singleton();
5       private Singleton(){
6
7       }
8       public static  Singleton getInstance(){
9           return singleton;
10      }
11  }
```

2 懒汉模式

2.A 第一次调用时创建

```
public class Singleton {
    private static Singleton singleton=null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        if (singleton==null){
            singleton=new Singleton();
        }
        return singleton;
    }
}
```

2.B 但是如果有多个线程同时访问，那就不能保证只创建一个实例

假如第一个线程进入，此时 singleton 为 null，正在创建但还未创建，此时另一线程也判断 singleton 为 null，也进入创建，最终创建了二个实例
解决:给方法加同步关键字

```
public class Singleton {
    private static Singleton singleton=null;
    private Singleton(){

    }
    public static synchronized   Singleton getInstance(){

            if (singleton == null) {
                singleton = new Singleton();
            }

        return singleton;
    }
}
```

2.C 但是给整个方法加锁会严重影响效率

可以给同步代码块加上锁

```
public class Singleton {
    private static Singleton singleton=null;
    private Singleton(){

    }
    public static  Singleton getInstance(){
        synchronized (Singleton.class){
            if (singleton == null) {
                singleton = new Singleton();
            }
        }
        return singleton;

    }
}
```

2.D 那么还有一个问题,假如已经创建完成了，singleton 不等于 null，后面每次每次都要等待锁，然后判断，影响效率

其实除了第一次创建外，后面就不需要等待了，直接判断 null 返回即可

```
    public static  Singleton getInstance(){
        if (singleton == null) {
            synchronized (Singleton.class){
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
```