第4章 职责链模式

提出问题

> 问题描述：假设提交请求的对象并不明确知道谁是最终响应请求的对象，当产生一个请求时，应该如何寻找恰当的处理对象？如果对每个请求都制定一个处理对象，无疑会导致命令发送者与接受者之间的强耦合关系。如下：

```
public class HandlerA{
    public handle(){
        System.out.println("A Task");
    }
}
public class HandlerB{
    public handle(){
        System.out.println("B Task");
    }
}
public class HandlerC{
    public handle(){
        System.out.println("C Task");
    }
}
public class Test{
    public static void main(String[] args){
        HandlerA ha = new HandlerA();
        HandlerB hb = new HandlerB();
        HandlerC hc = new HandlerC();
        String taskType = "A";
        switch(taskType){
            case "A":ha.handle();break;
            case "B":hb.handle();break;
            case "C":hc.handle();break;
            default:throw new RuntimeException("Unsupported");break;
        }
    }
}
```

模式名称

职责链模式：Chain of Responsibility

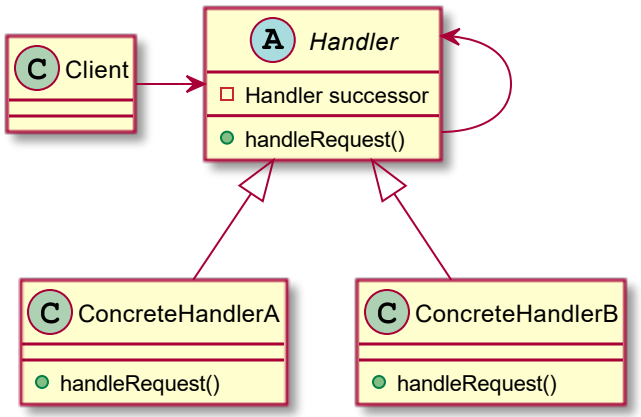
设计意图

职责链模式（Chain of Responsibility）将多个接收对象组织成链式结构，将请求沿着职责链传递直到找到恰当的接收者来处理请求，这使得每个对象都有机会处理请求，降低了请求发送者和接收者之间的耦合强度。

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

设计结构

类图



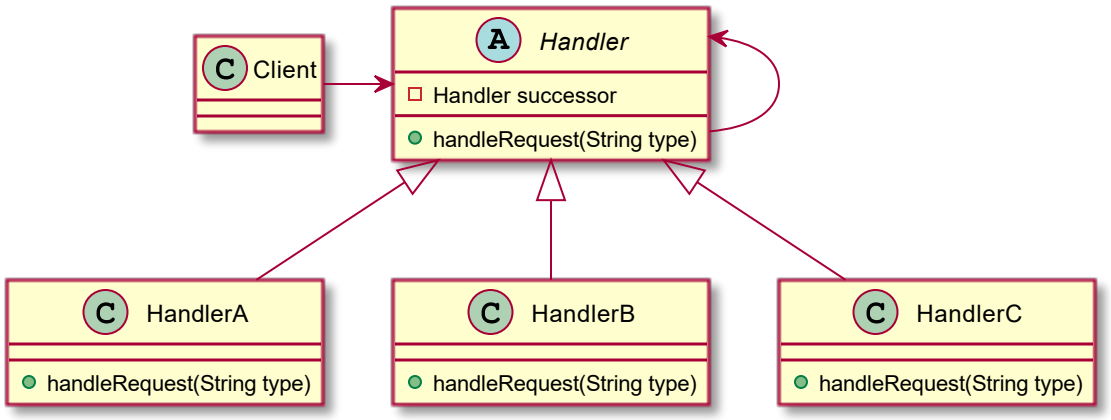
参与者

抽象处理者（Handler）：定义请求处理方法，维护继任者引用。
具体处理者（ConcreteHandler）：对具体请求进行处理，否则将请求转发给继任者。

代码

```
public abstract class Handler{
    private Handler successor;
    public void setSuccessor(Handler successor){
        this.successor = successor;
    }
    public void getSuccessor(){
        return this.successor;
    }
    public abstract void handleRequest(String request);
}
public class ConcreteHandlerA extends Handler{
    public void handleRequest(String request){
        if(request.equals("A")){
            //handle
        }else if(getSuccessor() != null){
            getSuccessor().handleRequest(request);
        }else{
            throw new RuntimeException("Unspported");
        }
    }
}
public class ConcreteHandlerB extends Handler{
    public void handleRequest(String request){
        if(request.equals("B")){
            //handle
        }else if(getSuccessor() != null){
            getSuccessor().handleRequest(request);
        }else{
            throw new RuntimeException("Unspported");
        }
    }
}
public class Test{
    public static void main(String[] args){
        Handler h = new ConcreteHandlerA(new ConcreteHandlerB(null));
        String request = "A";
        h.handleRequest(request);
    }
}
```

解决问题


Client
Handler
Handler successor
handleRequest(String type)
HandlerA
handleRequest(String type)
HandlerB
handleRequest(String type)
HandlerC
handleRequest(String type)

```
public abstract class Handler {
    private Handler successor;
    public Handler(Handler successor) {
        this.successor = successor;
    }
    public void handle(String request) {
        if(getSuccessor() != null) {
            getSuccessor().handle(request);
        }else {
            throw new RuntimeException("Unspported Request!");
        }
    }
    public Handler getSuccessor() {
        return successor;
    }
    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }
}
public class HandlerA extends Handler {
    public HandlerA(Handler successor) {
        super(successor);
    }
    @Override
    public void handle(String request) {
        if(request.equals("A"))
            System.out.println("Handle A");
        else
            super.handle(request);
    }
}
public class HandlerB extends Handler {
    public HandlerB(Handler successor) {
        super(successor);
    }
    @Override
    public void handle(String request) {
        if(request.equals("B"))
            System.out.println("Handle B");
        else
            super.handle(request);
    }
}
public class HandlerC extends Handler {
    public HandlerC(Handler successor) {
        super(successor);
```

```
    }
    @Override
    public void handle(String request) {
        if(request.equals("C"))
            System.out.println("Handle C");
        else
            super.handle(request);
    }
}
public class Test {
    public static void main(String[] args) {
        Handler hanlderChain = new HandlerA(new HandlerB(new HandlerC(null)));
        String request = "A";
        hanlderChain.handle(request);
    }
}
```

效果与适用性

优点

- 降低了发出请求的对象和处理请求的对象之间的耦合
- 增强了给对象指派职责的灵活性

缺点

- 不能保证请求一定被接收
- 系统性能将受到一定影响，而且在进行代码调试时不太方便，可能会造成循环调用
- 可能不容易观察运行时的特征，有碍于除错

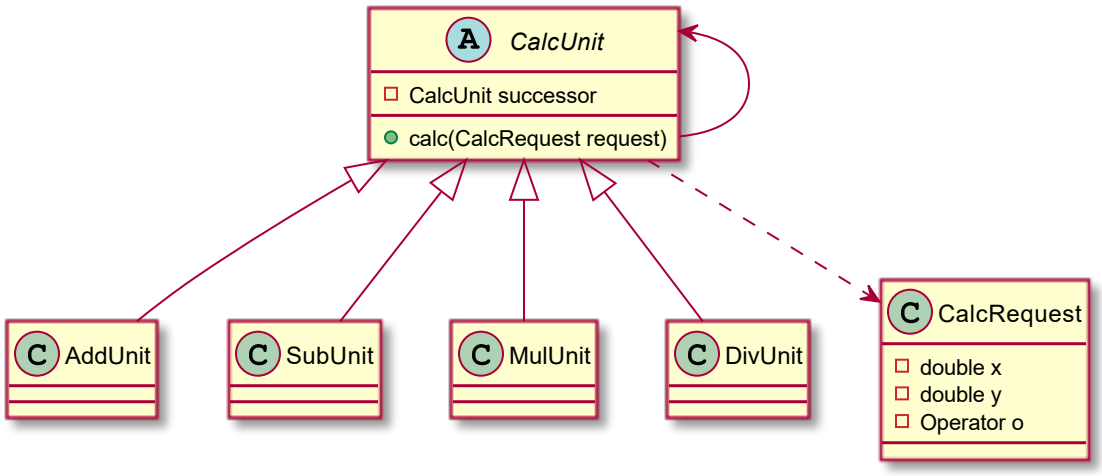
扩展案例

计算器

> 设计一个能进行四则运算的计算器，能根据输入的字符串请求返回计算结果，效果如下：

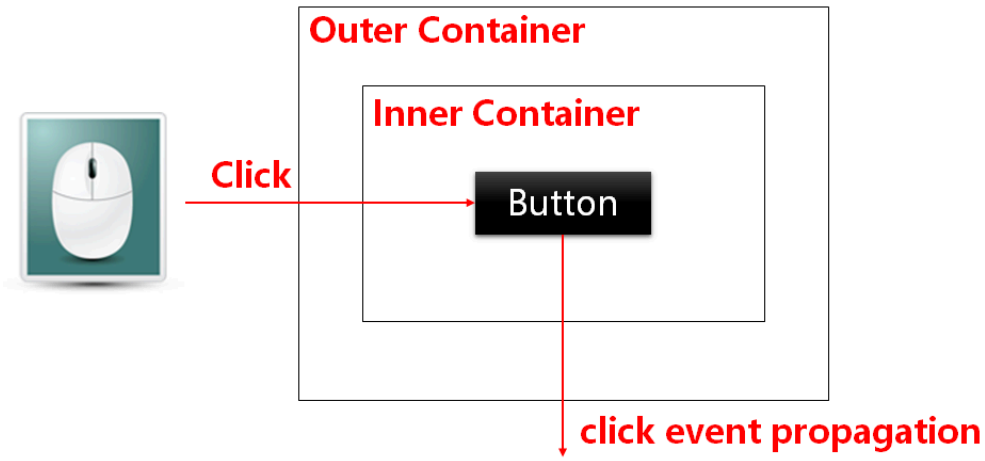
```
输入计算式: 1 + 1
结果:2.0
输入计算式: 2 - 1
结果:1.0
输入计算式: 45 / 9
结果:5.0
输入计算式: 77 * 12
结果:924.0
输入计算式: 2 $ 2
Exception in thread "main" java.lang.RuntimeException: 错误的计算符
```

将多个计算单元按链式结构组织，将封装好的计算请求依次传递，匹配成功的计算单元对请求进行处理并返回结果，类图如下：

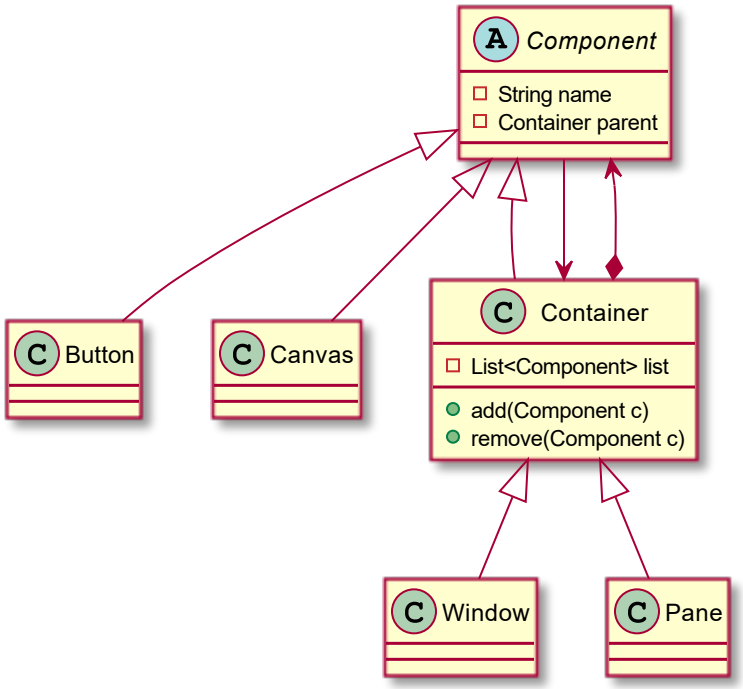


事件冒泡

> 图形界面开发中的事件冒泡是指元素上发生事件后，会将事件以冒泡形式向父节点传播直到祖先节点。如下图，鼠标在按钮上的单击事件会一直传播到最外层容器。

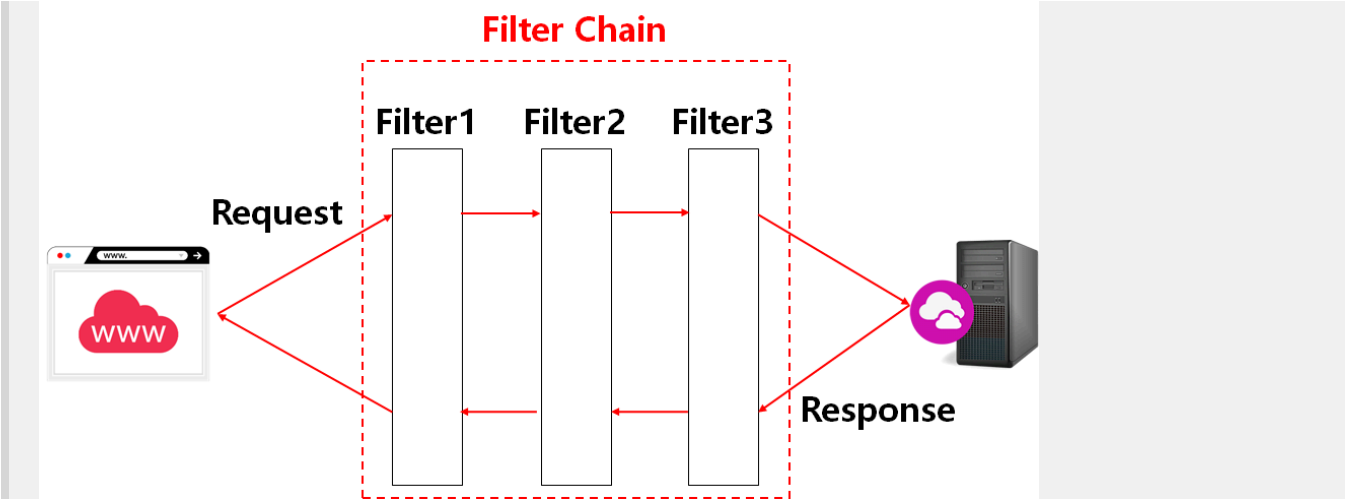


图形界面元素（或组件）采用组合模式组织，同时又需要通过职责链模式设计事件传播。参考类图如下：

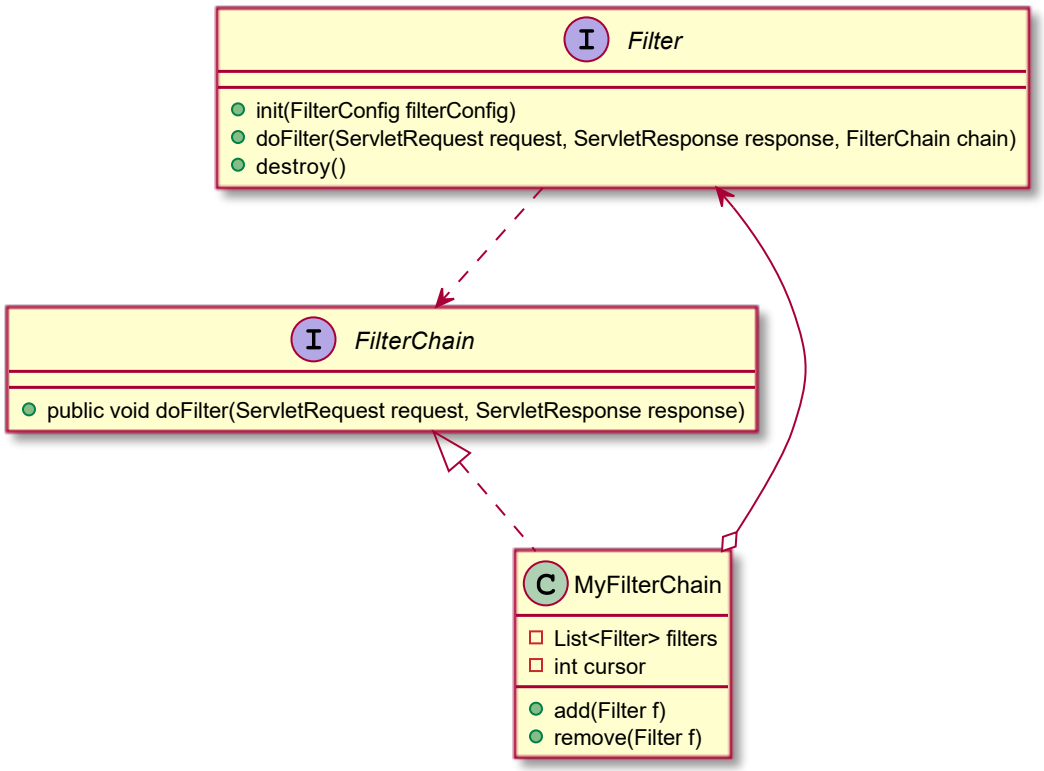


过滤器与过滤链

> 在Web应用程序中可以注册多个过滤器（Filter），每个过滤器都可以针对某一个URL进行拦截并做一些处理。多个过滤器可以组成滤器链对同一个URL进行拦截处理。如下图所示：



J2EE-API中只有Filter和FilterChain两个接口，实现Filter接口的类中处理具体过滤业务，过滤链则由实现FilterChain接口的类管理，这里我们设计了一个MyFilterChain类。类图如下：



过滤器链的内部实现如下：

```
public class MyFilterChain implements FilterChain{
    private List<Filter> filters;
    private ServletRequest request;
    private ServletResponse response;
    private int cursor;

    public MyFilterChain() {
        filters = new ArrayList<Filter>();
        cursor = -1;
    }

    public void doFilter(ServletRequest request,
        ServletResponse response) throws IOException, ServletException{
        // 如果存在下一个过滤器则将指针指向下一个过滤器，调用过滤功能
        if(hasNext()) {
            next();
            filters.get(cursor).doFilter(request, response, this);
        }
        this.request = request;
        this.response = response;
    }

    private boolean hasNext() {
        return this.cursor < filters.size() - 1;
    }
    private void next() {
        this.cursor++;
    }
    public void add(Filter f) {
        filters.add(f);
    }
    public void remove(Filter f) {
        filters.remove(f);
    }
    public void remove(int i) {
        filters.remove(i);
    }
}
```

思考题

> 网络上的各种资源（如HTML文档、图像、视频、片段和程序等）均可由一个URI（Universal Resource Identifier，通用资源标识符）进行定位，URI一般由三部分组成：命名机制（http://，ftp://）、资源主机名和资源自身名称。基于责任链模式设计程序

> 访问http://、ftp://、mailto://等不同类型资源，并对资源进行处理。要求给出类图以及实现思路。（提示：假设抽象处理类为 `URIHandler` ，处理方法定义为 `request(URI rui)` ）