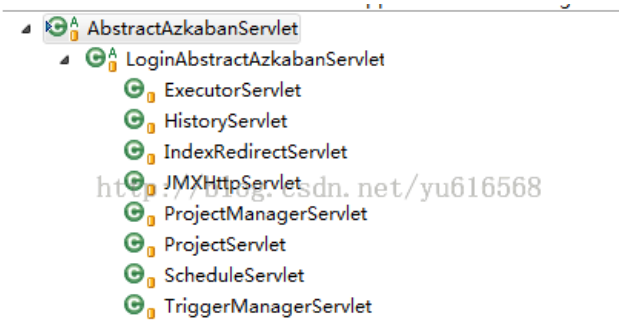
Azkaban使用的两个设计模式

缘由

最近看了看azkaban的代码，发现有两个设计模式比较巧妙，一个是在webServer端处理用户请求的servlet的类结构，一个是在execServer中使用的观察者模式。

webServer端的servlet的层次结构

如下图，下面是webServer提供的所有的servlet信息：



AbstractAzkabanServlet类继承自HttpServlet，我们一般都是实现doGet和doPost方法（还有针对delete请求的方法等）处理GET和POST请求，在azkaban中需要对每一个请求进行用户认证，认证的方式通过发放token的方式，在每一个token保存最后访问时间和创建时间，对访问间隔和一个token存在的最长时间都有限制。

在以上的类结构中，可以使用父类处理登录请求和用户认证的处理，所以在这种类结构中，每一个类提供如下方法：

- AbstractAzkabanServlet类提供了一些基础的页面，还有一些参数的处理方法；
- LoginAbstractAzkabanServlet类提供了doGet和doPost方法，这两个方法拦截所有的用户请求，这是因为其他所有具体的servlet都不提供这两个函数，所以所有的处理请求都会被LoginAbstractAzkabanServlet类拦截处理，然后再将具体的请求分成get请求、post请求和MultiformPost请求（负责处理文件上传的请求），而这三种请求分别再由具体的servlet处理。

具体的实现方式就只在父类（LoginAbstractAzkabanServlet）中实现doGet和doPost函数，拦截所有的登录、认证或者其他的所有请求都需要的操作，然后在该类中定义了多个抽象函数（需要由子类处理的请求），由子类实现不同的逻辑，在doGet或者doPost调用这些抽象函数。

这种方法能够使得代码简洁，子类只需要负责具体的逻辑，不需要被其他的公共操作干扰。

在这部分学习到了一些servlet中使用的方法：

- 在对于每一个请求的servlet创建的时候会调用init方法，参数为ServletConfig对象，我们可以调用setAttribute(key, value)保存一个全局的对象，在servlet对象启动的时候调用init方法使用参数的getServletContext().getAttribute(key)方法获取该对象，这样就可以避免使用全局变量的方式传递变量了。
- doGet和doPost的方法都有两个参数：HttpServletRequest req, HttpServletResponse resp，分别表示请求的参数信息和需要填充的回复信息，可以通过req.getParameter(key)方法获取执行的参数内容，resp.setContentType(str)方法设置http请求回复中的content内容（这里都是使用json格式）。
- 在azkaban中需要一些重定向的请求，例如需要把所有没有设置servlet的请求都转发到"/index"下，重定向的实现可以通过两种方式，一种是直接在服务器内部跳转，另一种方式是回复301/302。response.sendRedirect(目的路径)方法是返回给客户端重定向回复，然后浏览器收到该回复之后会跳转到目的路径；request.getRequestDispatcher().forward(request,response)方法是直接在服务器内部进行跳转，对用户是透明的。

观察者模式

这是一个比较常用的设计模式了，以前在学习linux网络编程的时候就经常接触到这种模式，那时我们的网络编程模型使用的是epoll，并且在同一个连接的所有请求都是异步的，我们为每一个连接注册epoll的读写监听事件，然后由epoll在连接就绪的时候调用，这时候epoll就是一个观察者，它能够观察所有就绪的时间，然后取出保存在epoll中的对象指针调用处理函数来处理。所以在这种模式中涉及到两类对象：观察者和处理者，观察者负责监听所有可能发生的时间，在事件就绪之后通知处理者（一般通过调用处理者的接口），在处理者需要注册到观察者，当然可以在观察这种保存每个处理者感兴趣的事件再根据发生的事件类型判断是否通知，也可以将全部的事件都通知处理者，由处理者决定对那些事件进行处理，epoll使用的是前者，azkaban使用的是后者。

在azkaban中定义了EventListener 接口，处理者需要实现该接口，根据参数的event对象执行处理程序：

```
public interface EventListener {
  public void handleEvent(Event event);
}
```

观察者继承自EventHandler类，该类实现了观察者模式需要的函数：

```
public class EventHandler {
  private HashSet<EventListener> listeners = new HashSet<EventListener>();

  public EventHandler() {
  }

  //向观察者注册
  public void addListener(EventListener listener) {
    listeners.add(listener);
  }

  //事件触发，通知所有注册的处理者
  public void fireEventListeners(Event event) {
    ArrayList<EventListener> listeners =
        new ArrayList<EventListener>(this.listeners);
    for (EventListener listener : listeners) {
      listener.handleEvent(event);
    }
  }

  //取消注册
  public void removeListener(EventListener listener) {
    listeners.remove(listener);
  }
}
```

这样就实现了观察者模式的框架，在azkaban中事件的类型有一下几类：

```
  public enum Type {
    FLOW_STARTED,                   //一个流启动执行
    FLOW_FINISHED,                   //一个流执行完成
    JOB_STARTED,                      //一个任务开始执行
    JOB_FINISHED,                      //一个任务执行完成
    JOB_STATUS_CHANGED,        //任务状态变化
    EXTERNAL_FLOW_UPDATED,    //内部流更新
    EXTERNAL_JOB_UPDATED        //内部任务更新
  }
```

在azkaban中可以设置两个flow执行的pipeline，所以在将某一个flow提交到执行服务器执行的时候可以设置他所以来的前一个flow的id，在执行服务器中需要观察前一个flow执行的状态已决定是否执行当前flow的某一个job（典型的pipeline是两个相同的job不能同时执行），在这种场景下需要为当前的flow创建一个FlowWatcher对象，该对象会在初始化的时候创建一个Listener并调用FlowRunner（每一个flow的某次执行的控制对象，这里是前一个flow的执行对象，作为观察者）的addListener方法，在每一个flow的每一个job执行完成之后调用所有fireEventListeners函数。在FlowWatcher的Listener对象的handleEvent函数中在处理事件：

```
 public class LocalFlowWatcherListener implements EventListener {
    @Override
    public void handleEvent(Event event) {
      if (event.getType() == Type.JOB_FINISHED) {
        if (event.getRunner() instanceof FlowRunner) {
          // 这里是在job没有执行（被取消或者跳过）的情况下由FlowRunner通知的
          Object data = event.getData();
          if (data instanceof ExecutableNode) {
            ExecutableNode node = (ExecutableNode) data;
            handleJobStatusChange(node.getNestedId(), node.getStatus());
          }
        } else if (event.getRunner() instanceof JobRunner) {
```

```
        // 这里是在一个job执行完成由jobRunner通知flowRunner，在由FlowRunner通知的。
        JobRunner runner = (JobRunner) event.getRunner();
        ExecutableNode node = runner.getNode();
        System.out.println(node + " looks like " + node.getStatus());
        handleJobStatusChange(node.getNestedId(), node.getStatus());
      }
    } else if (event.getType() == Type.FLOW_FINISHED) {
      stopWatcher();
    }
  }
}
```

在azkaban中每一个flow的一次执行由一个FlowRunner对象负责，这个对象会创建一个线程池用于提交每个job的执行，每一个job由一个JobRunner负责，这两类对象都是一个Runable对象，都是在单独的线程中执行，他们也都是观察者，flowRunner观察每一个flow的执行情况，JobRunner观察他负责的这个job的执行情况，同时flowRunner还会创建一个Listener，它作为JobRunner的处理者，在每一个job的事件发生之后会调用flowRunner的处理者处理，再由该Listener通知flowRunner的所有处理者，也就是LocalFlowWatcherListener。这样的使用方式可以说是观察者模式的一种升级版。

总结

主要还是自己面向对象编程的水平比较差，不能够很好地利用接口、继承等实现代码复用和方便的扩展，这方面以后需要重点学习一下。