jetty源码之

http请求到servlet的过程

程傑 2016.8.10

前言

springMVC 与 jetty

jetty 与 servlet

复习一下基础的servlet先…

servlet

什么是servlet?

Servlet 是实现特殊接口的 java 类，是运行在服务器端的Java应用程序，具有独立于平台和协议的特性。

servlet的工作模式

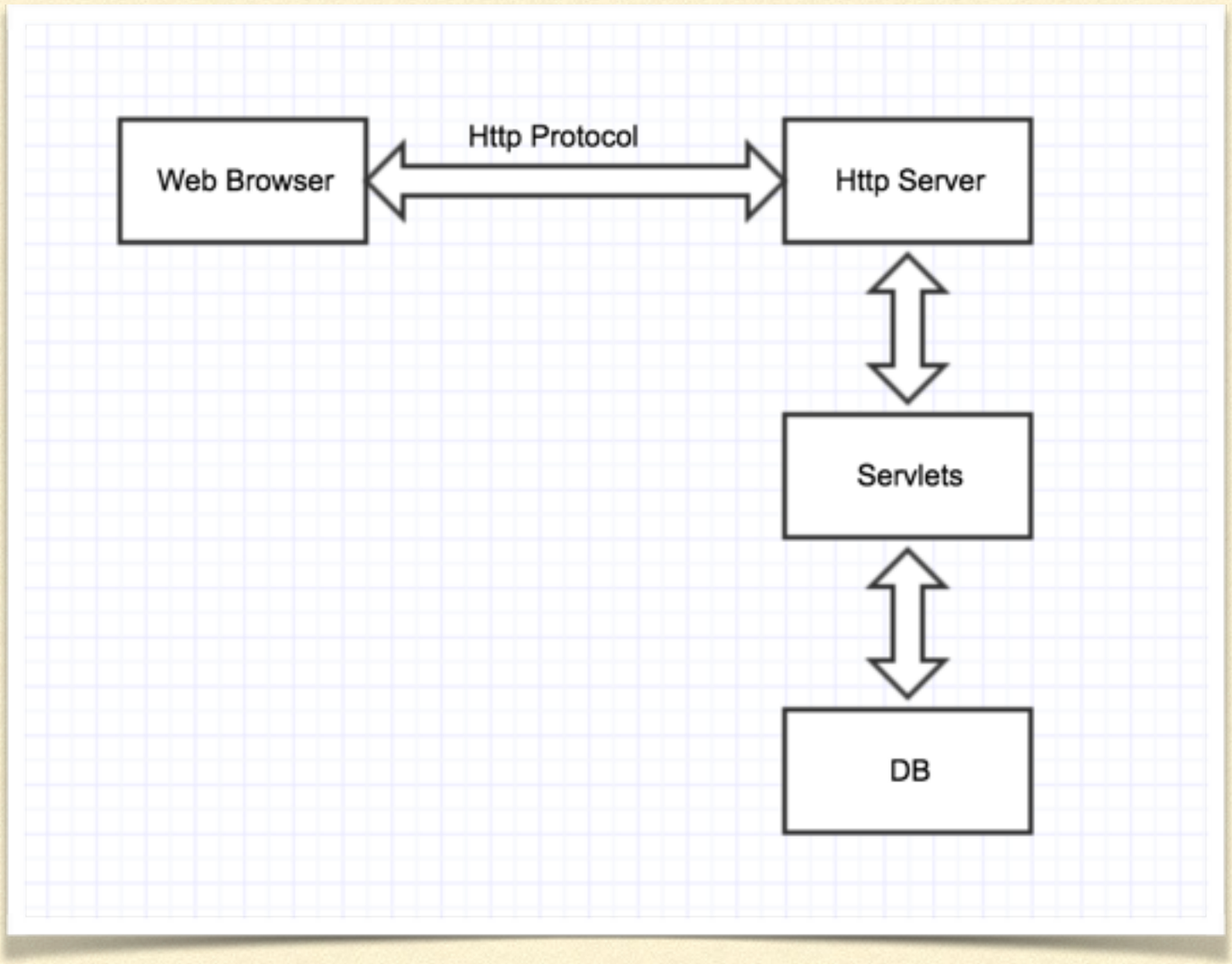
客户端发送请求至服务器，服务器启动并调用Servlet，Servlet根据客户端请求生成响应内容并将其传给服务器，还可以动态创建网页

servlet担当客户请求（浏览器或其他http程序）与服务器响应的中间层。

Java 类库的全部功能对 Servlet 来说都是可用的。它可以通过 sockets 和 RMI 机制与 applets、数据库或其他软件进行交互。

servlet架构

Servlet 在 Web 应用程序中的位置

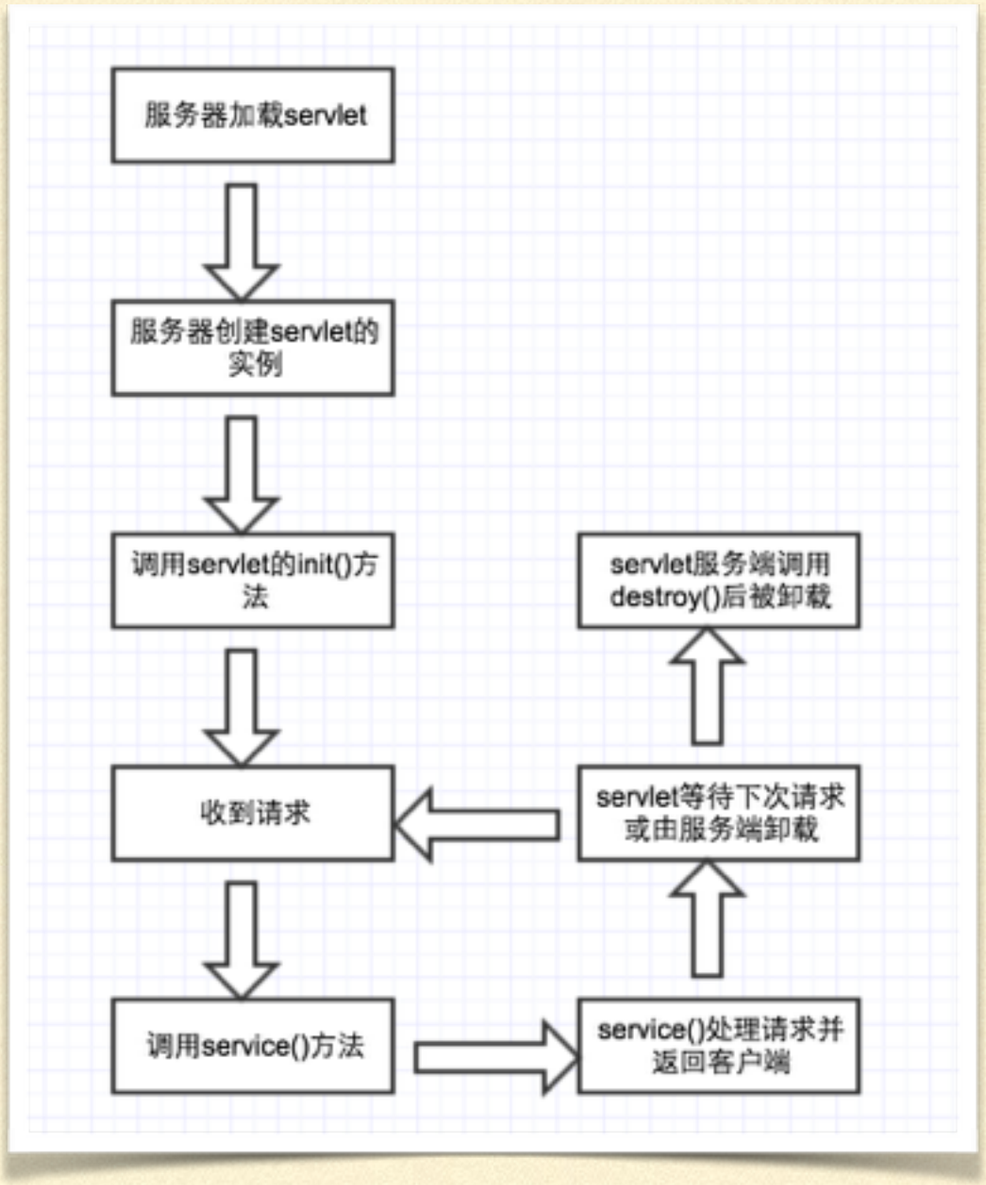


servlet生命周期

Servlet 生命周期可被定义为从创建直到毁灭的整个过程，由部署servlet的容器来控制。

- Servlet 通过调用 init () 方法进行初始化。
- Servlet 调用 service() 方法来处理客户端的请求。
- Servlet 通过调用 destroy() 方法终止。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

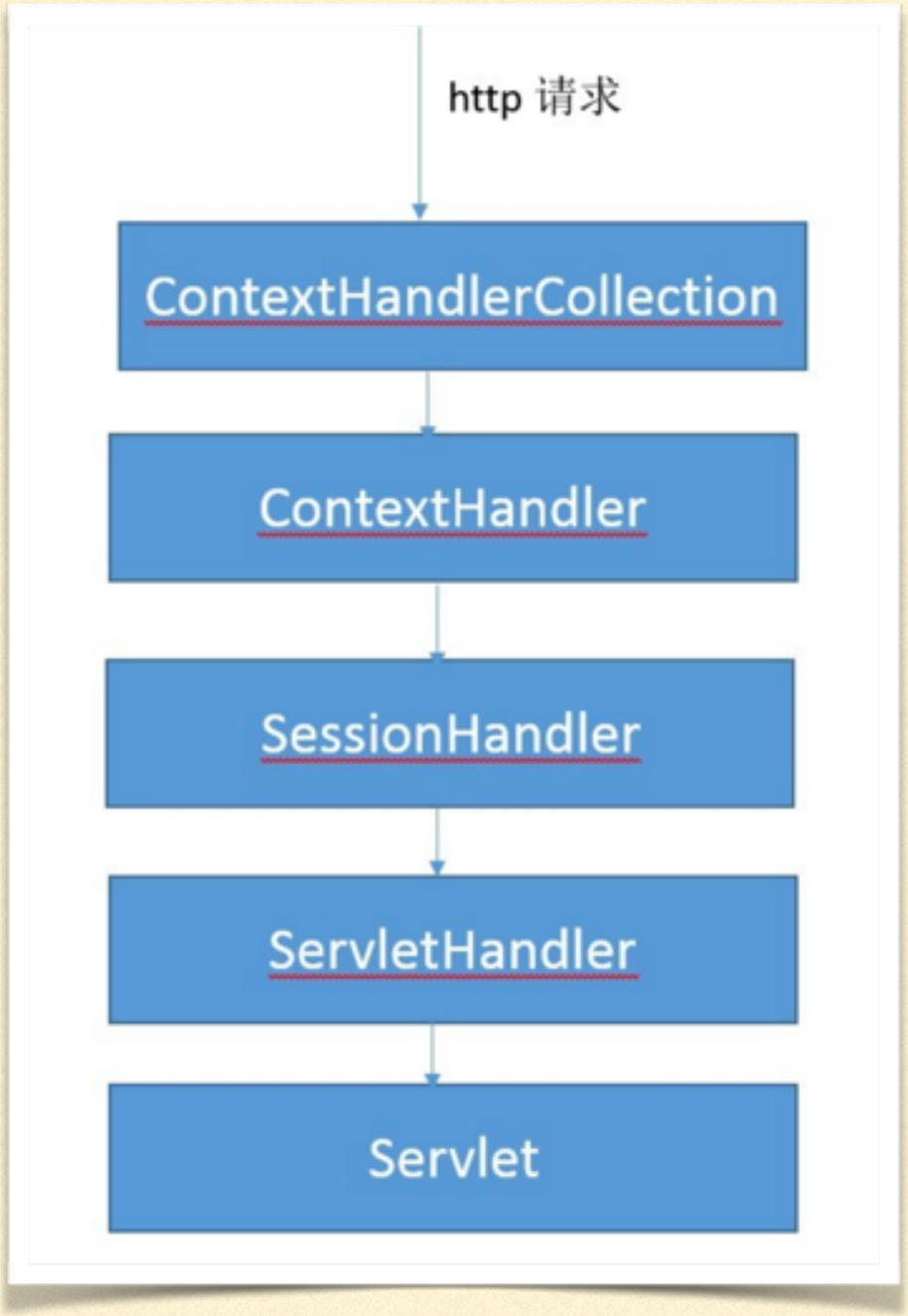
当一个请求映射到一个servlet时



servlet先复习到这了...

正片

一次http请求在jetty中的流程

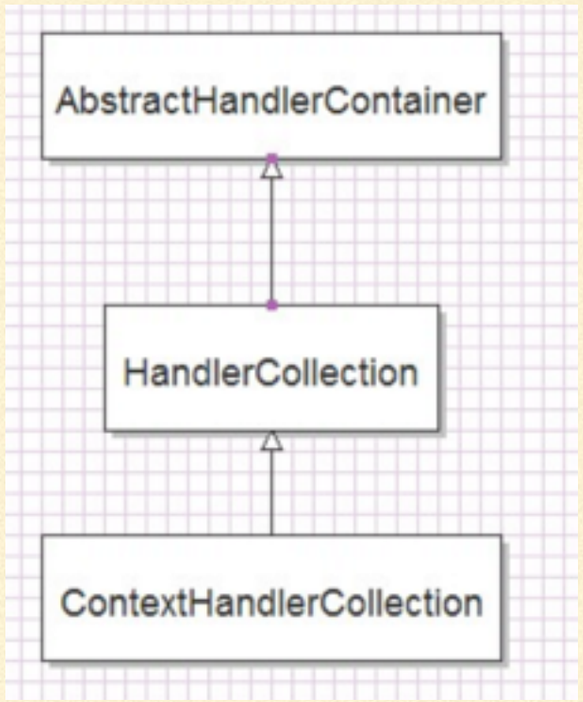


3-1 ContextHanlerCollection

一般会在web容器中部署多个应用，每个应用都对应着一个自己的context（上下文），需要一个集中管理者，维护部署的应用，对http请求进行路由。jetty中就是ContextHanlerCollection干的事情。

它将http请求交给匹配的context，然后context再转由内部的servlet来处理。

继承体系：



先看下ContextHandlerCollection中两个比较重要的属性

```
public class ContextHandlerCollection extends HandlerCollection
{
    private static final Logger LOG = Log.getLogger(ContextHandlerCollection.class);

    private volatile PathMap _contextMap;
    private Class<? extends ContextHandler> _contextClass = ContextHandler.class;
```

- contextMap：用于维护contextPath与contextHandler的对应关系（所有的http请求都要经由它来匹配出相应的contextHandler）

  PathMap是jetty自己定义的类型，它继承自HashMap，但是对其进行了一些扩展，加上了前缀匹配，后缀匹配，lazy匹配什么的。

- contextClass：handler的类型

doStart( )

```
/* ----------------------------------------------------------- */
@Override
protected void doStart() throws Exception
{
    mapContexts();
    super.doStart();
}
```

doStart()用于创建pathMap，创建当前所有的handler与path之间的对应关系。
然后执行父类的doStart方法（主要是启动所包含的所有的handler）

mapContexts( )

```
for (int i=0;i<handlers.length;i++)
{
    ContextHandler handler=(ContextHandler)handlers[i];

    String contextPath=handler.getContextPath();

    if (contextPath==null || contextPath.indexOf(',')>=0 || contextPath.startsWith("*"))
        throw new IllegalArgumentException ("Illegal context spec:"+contextPath);

    if(!contextPath.startsWith("/"))
        contextPath='/'+contextPath;

    if (contextPath.length()>1)
    {
        if (contextPath.endsWith("/"))
            contextPath+="*";
        else if (!contextPath.endsWith("/*"))
            contextPath+="/*";
    }
```

（1）遍历当前所有的handler，也就是contextHandler；

（2）预处理他们的path，例如加入的工程为manager，部署在jetty默认的path将是/manager，预处理是将其变成/manager/*这个样子；

（3）将其放到pathMap里面去。

handle( )

将http请求分发给对应的contextHandler来处理

```
public void handle(String target, Request baseRequest, HttpServletRequ
{
    Handler[] handlers = getHandlers();
    if (handlers==null || handlers.length==0)
    return;
```

```
PathMap map = _contextMap;
if (map!=null && target!=null && target.startsWith("/"))
{
    // first, get all contexts matched by context path
    Object contexts = map.getLazyMatches(target);

        for (int i=0; i<LazyList.size(contexts); i++)
        {
```

```
// no virtualhosts defined for the context, least specific
// will handle any request that does not match to a specific virtual host above
list=hosts.get("*");
for (int j=0; j<LazyList.size(list); j++)
{
    Handler handler = (Handler)LazyList.get(list,j);
    handler.handle(target,baseRequest, request, response);
    if (baseRequest.isHandled())
        return;
}
```

主要做了两件事：

（1）调用当前contextMap的getLazyMatches方法，找到所有与当前匹配的contextHandler

（2）遍历这些handler，然后依次调用他们的handle方法来处理这个请求，直到这次的请求被处理了为止。。

3-2 ContextHandler

每一个web app都对应相应一个context（上下文）,也就对应一个contextHandler，当servlet容器收到外部的http请求后，会根据path来找到对应的contextHandler来处理，然后将请求交由内部对应的servlet来处理。

```
public class ContextHandler extends ScopedHandler implements Attributes, Server.Graceful
{
    private static final Logger LOG = Log.getLogger(ContextHandler.class);

    private static final ThreadLocal<Context> __context = new ThreadLocal<Context>();
```

当前的线程变量，用于保存到当前的servletContext

handle( )

用于处理远程的http请求，判断当前请求的path，如果属于当前的webapp，则处理 。

```
public void handle(String target, HttpServletRequest request, HttpServletResponse response, int dispatch)
        throws IOException, ServletException  {
    boolean new_context=false;
    SContext old_context=null;
    String old_context_path=null;
    String old_servlet_path=null;
    String old_path_info=null;
    ClassLoader old_classloader=null;
    Thread current_thread=null;
    Request base_request=(request instanceof Request)?(Request)request:HttpConnection.getCurrentConnection().getRequest();
```

```
} else if (target.startsWith(_contextPath) && (_contextPath.length()==1 || target.charAt(_contextPath.length())=='/')){
   //这里看http的请求path是否以当前的contextPaht开始
    if (_contextPath.length()>1)
        target=target.substring(_contextPath.length());
} else  {
    //不是这个context该处理的，也就是不是这个web app该处理的
    return;
}
```

see in doScope( )

```
if (dispatch==REQUEST && isProtectedTarget(target))  //判断是否访
    throw new HttpException(HttpServletResponse.SC_NOT_FOUND);
//获取内部的handler来处理
Handler handler = getHandler();
//处理当前的http请求，这里其实是jetty.servlet.SessionHandler，不过最终还是调用ServletHandler
if (handler!=null)
    handler.handle(target, request, response, dispatch);
```

```
finally
{
    if (old_context!=_scontext)
    {
        // reset the classloader
        if (_classLoader!=null)
        {
            current_thread.setContextClassLoader(old_classloader);
        }

        // reset the context and servlet path.
        base_request.setContext(old_context);
        base_request.setContextPath(old_context_path);
        base_request.setServletPath(old_servlet_path);
        base_request.setPathInfo(old_path_info);
```

这个handle()方法有点长..最终会将http请求交给对应的servlet来处理。

处理流程总结：
（1）判断http请求的path与当前web程序的contextPath是否对应，若不对应的话则说明这个http请求不该由它来处理；
（2）如果对应的话，那么设置当前这个request的servletContext，当前线程的classLoader等；
（3）调用内部的handler来处理这个request，这里一般情况下都是SessionHandler。不过SessionHandler最终也会调用ServletHandler，交给servlet来处理；

3-3 ServletHandler

从整个http的处理过程来看，ServletHandler应该算得上是最接近用户定义的servlet的了；

servletHandler也需要负责对http请求的path进行处理，才能将这个请求交给正确的servlet；

属性定义：

```
private ContextHandler _contextHandler;    //这个servlet所属的contextHandler
private ContextHandler.SContext _servletContext;  //当前这个context的servletcontext
private FilterHolder[] _filters;    //filter数组
private FilterMapping[] _filterMappings;  //xml中的filter的map信息
private boolean _filterChainsCached=true;
private int _maxFilterChainsCacheSize=1000;
private boolean _startWithUnavailable=true;

private ServletHolder[] _servlets;    //servletholder数组，一般情况下用户定义的servlet都会被servletholder包装一下
private ServletMapping[] _servletMappings;    //用于保存从xml中读取出来的servlet的map信息

private transient Map _filterNameMap= new HashMap();     //filter的name对应
private transient List _filterPathMappings;    //将filter与特定的path对应起来
private transient MultiMap _filterNameMappings;  //servlet的name与filter的对应，有的filter可能会指定特定的servlet

private transient Map _servletNameMap=new HashMap();    //servlet与名字进行对应
private transient PathMap _servletPathMap;    //pathmap，这个很重要，当请求来了之后，就会通过它来匹配出合适的servlet来处理

protected transient HashMap _chainCache[];    //请求访问path与fitlerchain的缓存，防止每次都要创建
```

web.xml文件中会定义许多servlet，最后都会通过 <servlet-mapping> 元素将某个servlet与一个或者多个path对应起来，最终的对应关系都会在servletPathMap里。

doStart( )

```
@Override
protected synchronized void doStart()
    throws Exception
{
    _servletContext=ContextHandler.getCurrentContext();
    _contextHandler=(ServletContextHandler)(_servletContext==null?null:_servletContext.getContextHandler());

    if (_contextHandler!=null)
    {
        SecurityHandler security_handler = (SecurityHandler)_contextHandler.getChildHandlerByClass(SecurityHandler.class);
        if (security_handler!=null)
            _identityService=security_handler.getIdentityService();
    }

    updateNameMappings();
    updateMappings();
```

－updateNameMappings：名字对应map的更新，也就是在xml文件中定义的servlet名字与servletholder对象的对应，当然还有filter;

－updateMappings()：处理xml文件中定义的mapping信息，这里最重要的事就是更新pathMap。

handle( )

```
// find the servlet
if (target.startsWith("/")) {
    PathMap.Entry entry=getHolderEntry(target);    //通过targer的匹配，找出合适的servletholder
    if (entry!=null) { //如果可以找到合适的servlet
        servlet_holder = (ServletHolder)entry.getValue();    //获取相应的servletHodler
        base_request.setServletName(servlet_holder.getName());  //设置servlet的一些基本信息
```

```
} else { //如果targer不是以"/"开始的,那么表示不是以path来匹配的，所以要用名字来找servlet
    servlet_holder=(ServletHolder)_servletNameMap.get(target);  //通过名字来寻找servletholder
    if (servlet_holder!=null && _filterMappings!=null && _filterMappings.length>0)
    {
        base_request.setServletName(servlet_holder.getName());
        chain=getFilterChain(type, null,servlet_holder);  //构成链
    }
}
```

```
// Do the filter/handling thang
//先进行filter的操作，然后处理http请求
if (servlet_holder!=null)
{
    base_request.setHandled(true);  //这里设置一个标志位，表示这个http请求已经处理过了
    if (chain!=null) {    //如果有filter链，那么还需要先从filter来开始执行
        chain.doFilter(request, response);
    } else {  //如果没有的话，那么直接调用servletholder的handle方法来处理就好了，这期其实也就是调用实际的servlet的service方

        servlet_holder.handle(request,response);
    }
} else {
    notFound(request, response);  //这里表示找不到处理
}
```

handle()方法总结：

—根据target来获取相应的servletHolder，还要获取filter链，在filter执行完了以后再经由servlet来处理这次http请求；

—通过target来获取相应的servletholder调用的是getHolderEntry方法，其实实现就是直接从pathMap里面获得相应的servletHolder。

最后一步！

```
servlet_holder.handle(baseRequest,req,res);
```

open implementation

```
public void handle(Request baseRequest,
                   ServletRequest request,
                   ServletResponse response)
    throws ServletException,
           UnavailableException,
           IOException
{
    if (_class==null)
        throw new UnavailableException("Servlet Not Initialized");

    Servlet servlet=_servlet;
```

```
servlet.service(request,response);
servlet_error=false;
```

完