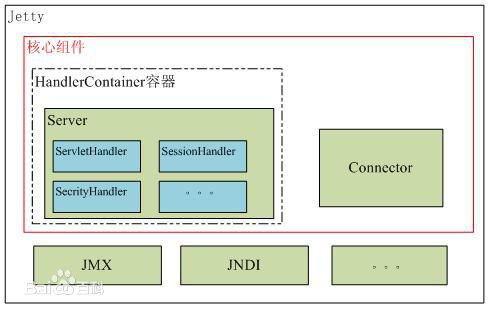
Jetty嵌入式HttpServer

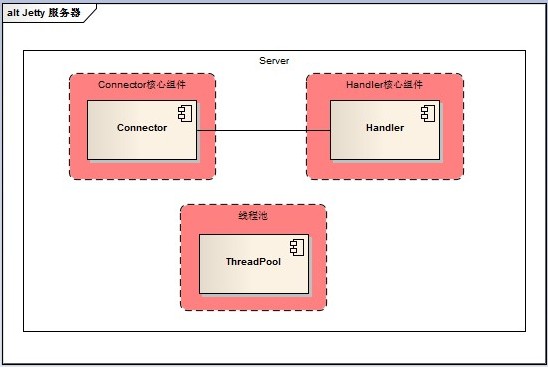
Jetty是一个开源的Servlet容器，基于Java的Web容器，例如Jsp和Servlet提供运行环境。Jetty是使用Java语言编写的，它的API以一组JAR包的形式发布。开发人员可以将Jetty容器实例化一个对象，迅速为一些独立运行的Java应用提供网络和Web连接。Jetty的核心组件如下图所示：



Jetty核心组件

# 1.Server

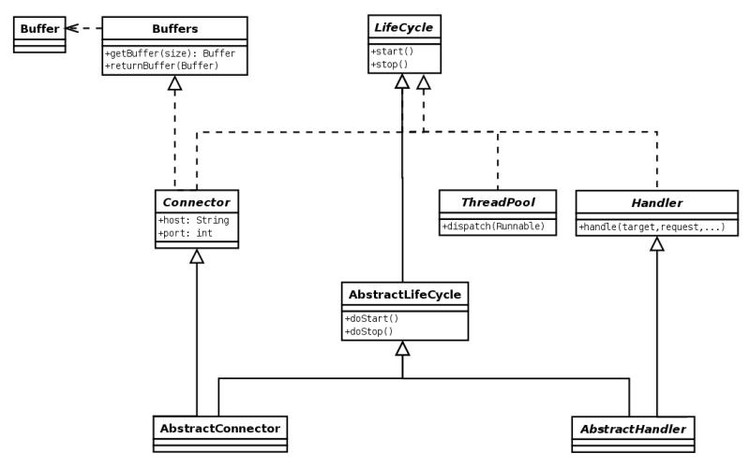
Jetty是由一个或多个connecctor核心组件以及一系列handler组件和线程池组成，结构图如下所示：



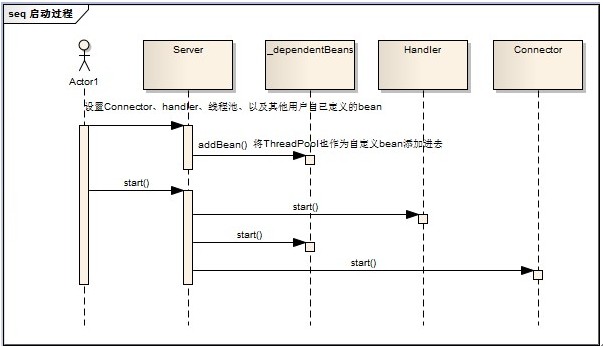
Jetty结构

Connector负责监听及接收客户连接请求，而Handler组件则负责处理请求并给予响应，这两个组件需要的线程资源都直接从线程池ThreadPool中获取。Server负责创建并初始化connector、handler和ThreadPool组件，然后调用start方法启动它们。

Jetty遵循一些比较规范的模式，最抽象的概念是通过接口定义，如连接器、处理器和缓冲区，在抽象类中实现这些接口的通用操作，如AbstractConnector、AbstractHandler和AbstractBuffer，如下图所示：



Server的启动过程，如下所示：



Jetty的启动是从Server的start方法开始：

1）Server类继承了AbstractLifeCycle类，首先会执行AbstractLifeCycle的start方法

2）创建队列线程池，用于处理每个建立连接产生的任务

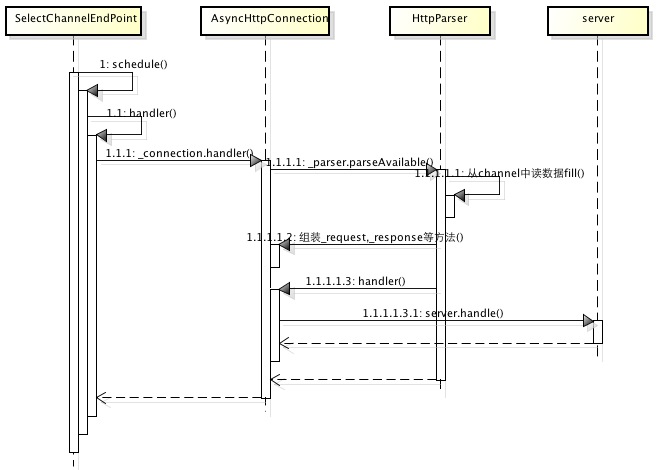
3）启动设置到Server的Handler（构成Handler链，调用方法super.doStart），执行如下：

HandlerWrapper.doStart

—> ContextHandlerCollection#mapContext //path和handler构成对应关系

—> ContexHandler.doStart //handler的初始化工作

4）启动Connector，接收客户端的请求，Connector默认采用NIO的方式接收请求，



Jetty接收到请求，解析Request、Response属性并最终调用Handler中的方法。

# 2.Handler

Handler组件负责处理接收到的请求，核心API是handle方法，如下所示：

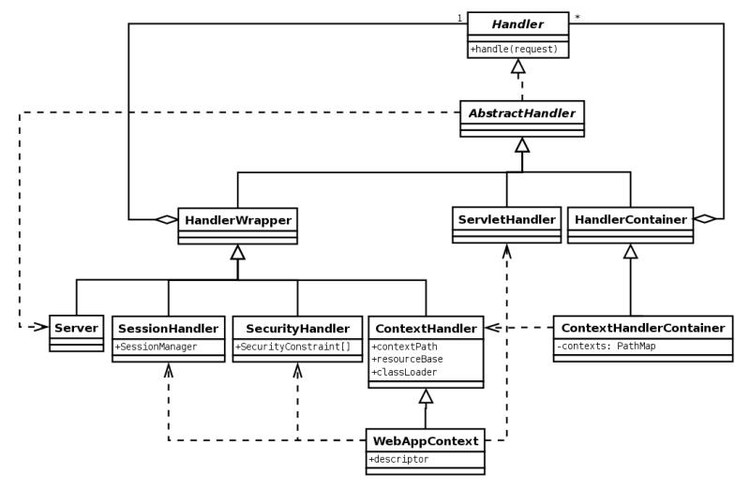
**public** **void** handle(String target, HttpServletRequest request, HttpServletResponse response, **int** dispatch)

**throws** IOException, ServletException;

这个方法可以处理请求，传递请求另外一个处理器（或者servlet）,或者修改并保证请求传递出去。有三种类型的处理器：

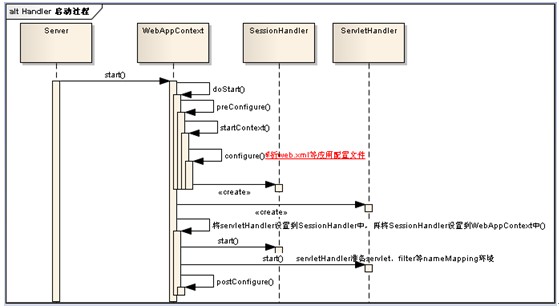
* Coordinating Handlers，负责路由请求到其他处理器的处理器，如HandlerCollection，ContextHandlerCollection
* Filtering Handlers，负责增加请求并传递到其他处理器的处理器，如HandlerWrapper，ContextHandler,SessionHandler
* Generating Handlers，负责产生内容的处理器，如ResourceHandler,ServletHandler

类图如下所示：



## 2.1 Handler调用流程

Jetty将真正处理请求的动作都抽象成Handler，因此所有处理组件都要实现这个接口，包括下图中的WebAppContext(AbstractLifeCycle里面的start方法会调用WebAppContext中的doStart方法)，启动过程如下：



WebAppContext中doStart方法，会初始化四个类：

* WebInfConfiguration，生成classLoader加载应用的war及目录等，加载WEB-INF类及Lib classpaths等
* WebXmlConfiguration,WebAppContext中会调用ContextHandler的doStart方法，其中startContext中调用WebXmlConfiguration类解析XML文件，并初始化xml里面配置的servlet
* JettyWebXmlConfiguration和TagLibConfiguration的作用大致也和WebXmlConfiguration相同

WebAppContext根据请求，创建对应的SessionHandler和ServletHandler，并将ServletHandler设置到SessionHandler中，同时将SessionHandler设置到WebAppContext中。最后是调用ServeletHandler中的start方法，准备servlet和filter到nameMapping环境。

## 2.2 ContextHandler

ContextHandler通过设置context、servlet路径及context classloader来封装一种调用，其

继承自ScopedHandler，是Jetty中实现对一个Web Application的各种资源进行管理，并串联实现整个Servlet框架的类，比如部分实现了ServletContext接口，并在其doScope方法中为当前Request的执行提供相应的环境，如设置servletPath、pathInfo、设置ServletContext到ThreadLocal中。在Jetty中，Servlet的执行流程和框架由ServletHandler实现，Security框架由SecrityHandler完成，而ContextHandler主要用于实现环境的配置，如Request状态的设置、ClassLoader的配置、ServletContext的实现等。在ContextHandler类中包含了一个Context的内部类，用于实现ServletContext，而ContextHandler中很多字段和方法也是用于实现ServletContext。ContextHandler包括以下部分：

* ServletContextHandler : Servlet的一次请求需要调用到Thread
* ServletPath, Servlet请求路径
* ContextClassLoader

classLoader用于加载类，WebAppClassLoader:用于HttpContext中的类加载

* LifeCycle: 生命周期的管理
* Attributes：
* Container：
* ContextHandlerCollection，一般Server中会部署多个应用，每个应用都对应着一个自己的Context，这就需要Collection去管理，而且需要对Http请求进行路由，将http请求交给相应的应用来处理。ContextHandlerCollection，维护Jetty中部署的应用，并将Http请求交给匹配的context，然后context再转由内部的servlet来处理。

注：基本的调用流程，Connector -> Server -> ContextHandlerCollection -> WebAppContext ->ServletHandler -> Servlet

## 2.3 HelloHandler的执行

HelloHandler的源码如下(功能简单，仅从路径中获取参数)：

public class HelloHandler extends AbstractHandler {

@Override

public void handle(String target,

HttpServletRequest request,

HttpServletResponse response,

int dispatch)

throws IOException, ServletException {

/\*\*

\* 从URL中传递处理的参数

\* http://localhost:60999/search?query=hello

\* \*/

String query = request.getParameter("query");

System.out.println("receive query:"+query);

String result = "Welcome to my Server";

if( null != query && query.equals("hello")){

result = query + "." + result;

}

System.out.println("request:"+request.toString()+

"response:"+response.toString()+

" result:"+result);

}

}

对HelloHandler的调用代码如下：

public static void main(String[] args){

Server server = new Server(8080);

System.out.println("Server start");

ContextHandler context = new ContextHandler();

context.setContextPath("/search");

context.setResourceBase(".");

context.setClassLoader(Thread.currentThread().getContextClassLoader());

server.setHandler(context);

context.setHandler(new HelloHandler());

server.start();

server.join();

}

Jetty的工作方式很简单，接受到请求，并将这个请求交给在Server中注册的代理Handler去执行，如何执行开发1者的Handler，由开发者去规定，Jetty要做的是调用注册的第一个Handler的handle方法，要接受一个Web请求，首先要创建一个ContextHandler，如上面代码所示。

在浏览器中输出http://localhost:8080时的请求会代理到Server类的ContextHandler的handle方法，ContextHandler又会调动HelloHandler的handle方法（业务逻辑相关的Handler）。

# 3.Servlet

Servlet是Server Applet的缩写，即在服务器端运行的小程序，而Servlet框架则是对Http服务器和用户小程序中间层的标准化和抽象，这一层隔离了HTTP服务器的实现细节，而Servlet规范定义了各个类的行为，从而保证了这些服务器端运行的小程序对服务器实现的无关性。在Servlet中，几个核心类：

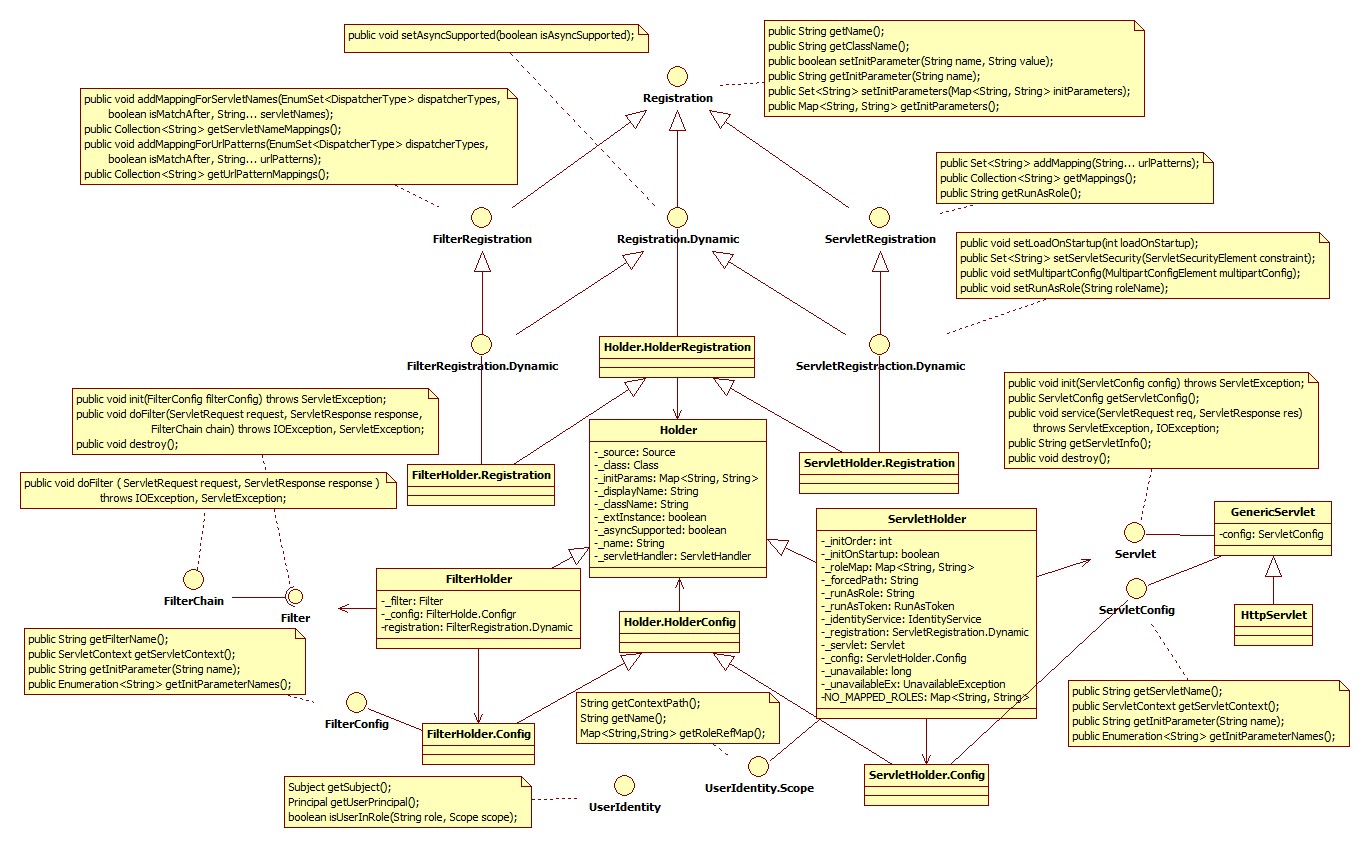
* ServletContext，定义了一些与可以与Servlet Container交互的方法
* Registration，定义Filter和Servlet的动态注册
* ServletRequest(HttpServletRequest)，对Htttp请求消息的封装
* ServletReponse(HttpServletResponse)，对Http相应消息的封装
* RequestDispatcher，将当前请求分发给另一个URL，设置ServletContext以实现进一步的处理
* Servlet(HttpServlet)，所有服务器小程序要实现的接口，重新doGet/doPost等方法以实现响应请求的相关逻辑
* Filter（FilterChain），在进入Servlet之前以及出Servlet以后添加一些用户自定义的逻辑，以实现一些Filter功能，如用户验证、日志打印等
* AsyncContext，实现异步请求处理

*注：访问URL指定路径后，会将请求代理到Server类的Handle方法，然后交给ContextHandler的handle方法，ContextHandler又会调用HelloHandler的handle方法。*

*在Servlet中，ContextHandler调用的是ServletHandler的Handle方法，然后由路径决定Servlet和Filter的执行。（ServletHandler仅是Server中生成响应内容的Handler，也是开发者最常涉及到的Handler）*

## 3.1 Servlet Handler

ServletHandler是一个用于管理Filter、FilterMapping、Servlet、ServletMapping等的容器。在Jetty中，Servlet由ServletHolder封装，Filter由FilterHolder封装。



ServletHandler相关的成员变量：

private FilterHolder[] \_filters=new FilterHolder[0];

private FilterMapping[] \_filterMappings;

private ServletHolder[] \_servlets=new ServletHolder[0];

private ServletMapping[] \_servletMappings;

private final Map<String,FilterHolder> \_filterNameMap= new HashMap<String,FilterHolder>();

private List<FilterMapping> \_filterPathMappings;

private MultiMap<String> \_filterNameMappings;

private final Map<String,ServletHolder> \_servletNameMap=new HashMap<String,ServletHolder>();

 private PathMap \_servletPathMap;

其中FilterHolder和ServletHolder分别用于存储Filter和Servlet实例及配置信息。

## 3.2 Servlet和Filter

在Jetty中，每个Servlet和其相关信息都是由ServletHolder封装，并且将Servlet相关操作代理给ServletHolder，并交由Servlet Container管理其实例，负责其生命周期，以及响应请求到来时调用相关方法。

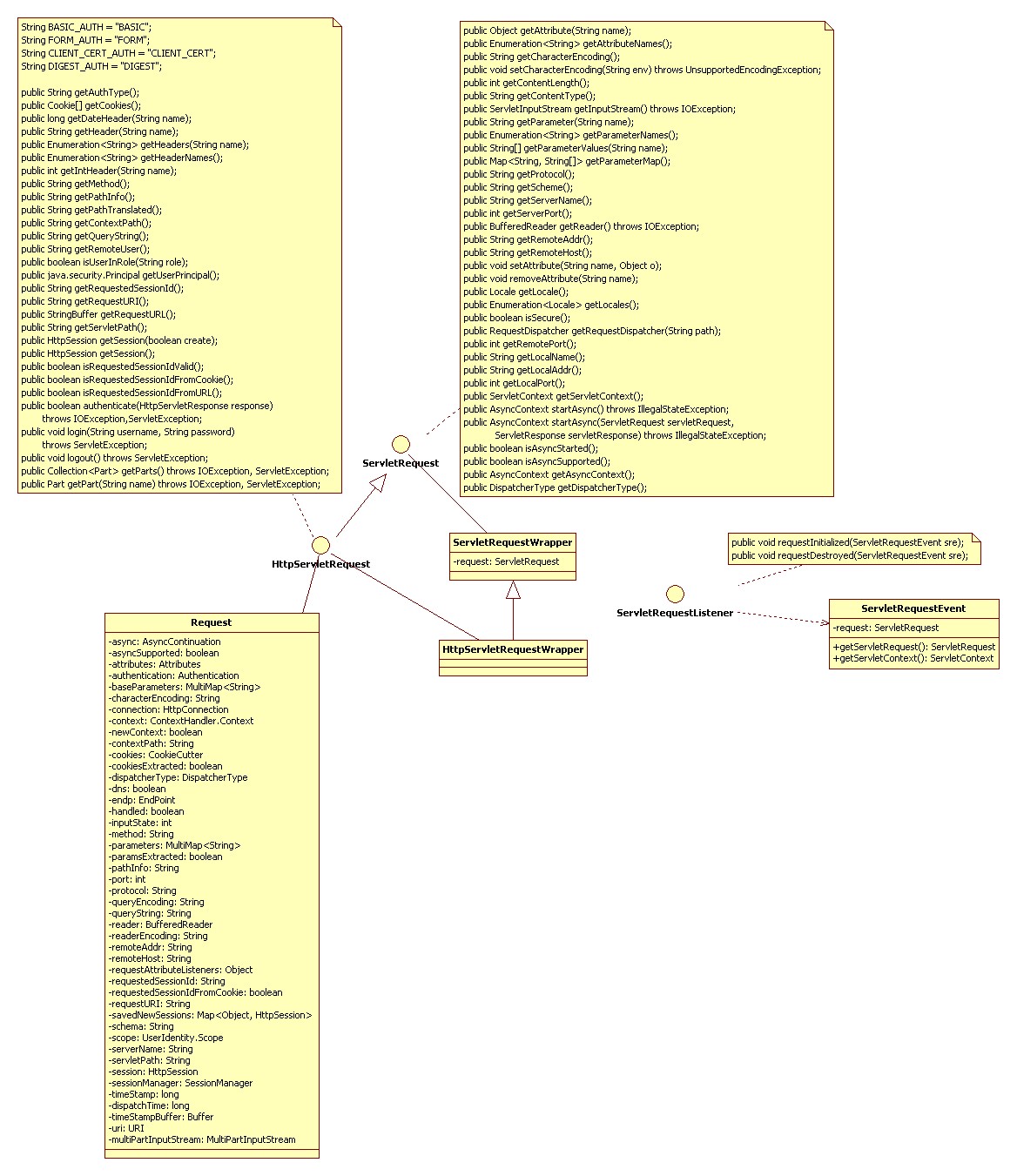
Filter，在Servlet的service方法执行前后添加的一些公共的Filter逻辑，引入Filter机制，为面向Filter的编程提供很大的便利，例如，记录一个request从进入Servlet Container到输出所花费的时间，为每个Request添加一些额外的信息以帮助之后的处理，对所有或者特定的Request添加用户验证功能等。

Filter可以在web.xml文件中定义，由servlet container负责其实例化、初始化及doFilter

## 3.3 ServletRequest和ServletResponse

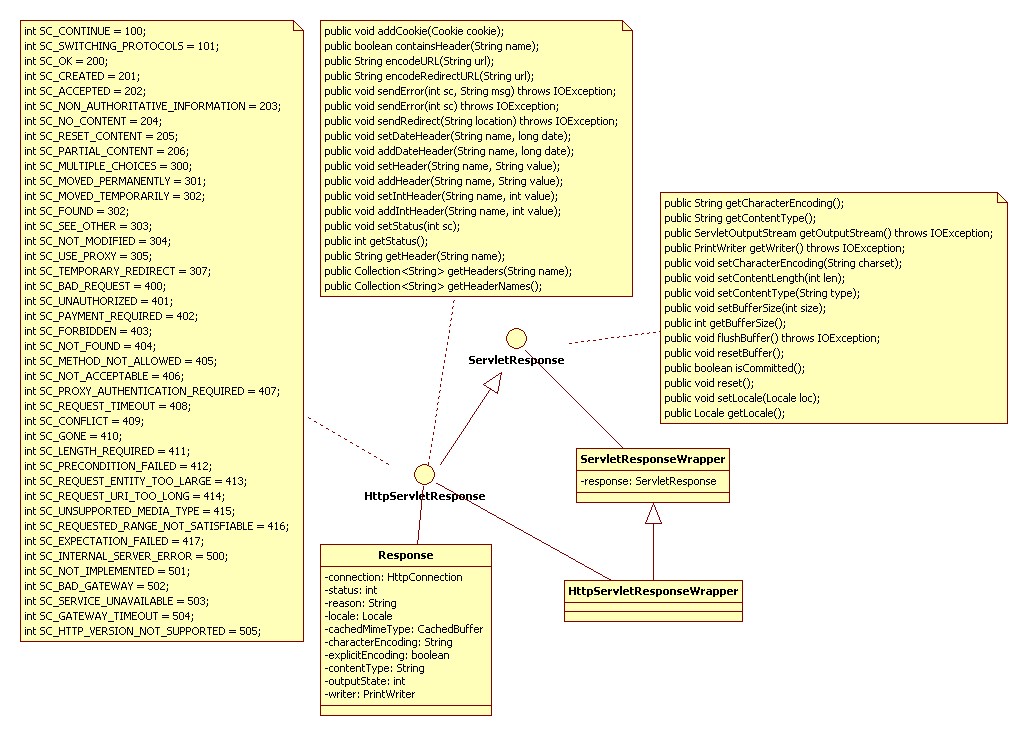
ServletRequest是对Servlet请求消息的封装，其子接口HttpServletRequest则是对HTTP请求消息的封装，在Servlet框架中默认实现了ServletRequestWrapper和

HttpServletRequestWrapper以便使用户对请求进行Wrap，类图如下：



在Jetty中，使用Request类实现了HttpServletRequest接口，Request中包含了HttpConnection包，HttpConnection包含了HttpParser解析Http请求后的所有信息，如请求行，请求头以及请求内容。

ServletResponse是对Servlet响应消息的封装，其子接口HttpServletResponse则是对HTTP响应消息的封装。



## 3.4 Servlet Example

定义SearchServlet，类如下：

public class SearchServlet extends HttpServlet {

private static final long serialVersionUID = -4012838481920999524L;

@Override

public void doPost(HttpServletRequest request,HttpServletResponse response) throws IOException {

String query = request.getParameter("query");

String result = "Welcome to My Server,It's a POST request.";

if( null != query && !query.trim().equals("")){

result = query + "."+result;

}

Tools.printToJson(result,response);

}

@Override

public void doGet(HttpServletRequest request,HttpServletResponse response) throws IOException {

String query = request.getParameter("query");

String result = "Welcome to my Server,It's a GET request";

if(null != query && !query.trim().equals("")){

result = query +"."+result;

}

Tools.printToJson(result,response);

}

}

在浏览器中输入：http://192.168.146.130:8099/search?query=test

输出如下：

test.Welcome to my Server,It's a GET request

Jetty Server如下：

public class JettyServletExampleServer {

public static void main(String[] args) throws Exception {

Server server = new Server(8099);

ServletContextHandler context = new ServletContextHandler(

ServletContextHandler.SESSIONS

);

context.setContextPath("/");

server.setHandler(context);

context.addServlet(new ServletHolder(new SearchServlet()),"/search");

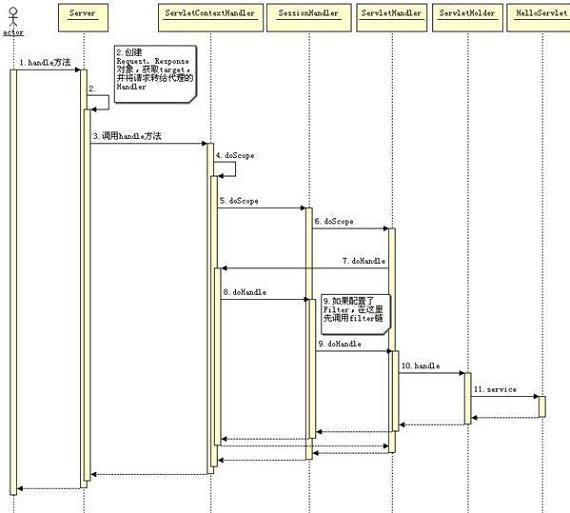
server.start();

server.join();

}

}

创建一个ServletContextHandler并给这个Handler添加一个Servlet，这里的ServletHolder是Servlet的一个装饰类，类似于Tomcat中的StandWrapper，下面是调用Servelet的时序图：



# 4.Google Injector

Guice是Google开发的轻量级IOC(控制反转，依赖倒置原理)容器，面向对象编程来减少计算机程序的耦合，一般分为依赖注入和依赖查找两种，其在对象被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它，本质上说是一个对象如果获取他所依赖的对象的引用（绑定，将一个接口绑定到具体的类中，客户端不需关心具体的实现，只需要获取相应的接口完成任务即可）。（当前IOC框架还有Spring）

IOC框架，有构造、属性以及函数注入三种方式，Guice的实现只需要在构造函数、字段或者注入函数上标注@Inject字段.

## 4.1 Hello Guice举例

类依赖注入，接口：

public interface HelloGuice {

public String sayHello();

}

具体实现类：

public class HelloGuiceImpl implements HelloGuice {

@Override

public String sayHello() {

return "Hello , Guice!";

}}

注入模块，实现Module接口

public class HelloGuiceModule implements Module {

@Override

public void configure(Binder binder) {

binder.bind(HelloGuice.class).to(HelloGuiceImpl.class);

}

}

测试类：

public class TestGuice {

public static void main(String[] args){

Injector injector = Guice.createInjector(new HelloGuiceModule());

HelloGuice hello = injector.getInstance(HelloGuice.class);

System.out.println(hello.sayHello());

}

}

输出结果：Hello , Guice

解析,Guice Injector在获取HelloGuice.class实例时根据HelloGuiceModule中#configure定义返回HelloGuiceIml实例。

## 4.2 @Injector注释的使用

在接口HelloGuice类定义上添加@ImplementedBy(HelloGuiceImpl.class)，不用再定义GuiceModule来显示指定绑定的实现类。（@Singleton，单例）

### 4.2.1基本属性注入

例子如下：

public class FieldInjectDemo {

@Inject

private HelloGuice helloGuice;

public HelloGuice getHelloGuice(){

return helloGuice;

}

public static void main(String[] args){

FieldInjectDemo demo= Guice.createInjector().getInstance(FieldInjectDemo.class);

demo.getHelloGuice().sayHello();

}}

接口HellGuice通过@Inject注解注入到FieldInjectDemo类中，然后在FieldInjectDemo类中使用此服务。HelloGuice通过@ImpletedBy注解关联到HelloGuiceImpl类中。

### 4.2.2 构造函数注入

public class ConstructorInjectDemo {

private HelloGuice helloGuice;

@Inject

public ConstructorInjectDemo(HelloGuice helloGuice){

this.helloGuice = helloGuice;

}

public HelloGuice getHelloGuice(){

return helloGuice;

}

public static void main(String[] args){

ConstructorInjectDemo cid = Guice.createInjector().getInstance(ConstructorInjectDemo.class);

cid.getHelloGuice().sayHello();

}

}

在构造函数上添加@Inject来达到自动注入的目的。

### 4.2.3 函数注入

public class SetterInjectDemo {

private HelloGuice helloGuice;

@Inject

public void setHelloGuice(HelloGuice helloGuice){

this.helloGuice = helloGuice;

}

public HelloGuice getHelloGuice(){

return helloGuice;

}

public static void main(String[] args){

SetterInjectDemo sid = Guice.createInjector().getInstance(SetterInjectDemo.class);

sid.getHelloGuice().sayHello();

}

}

# 5.Jetty 使用经验

## 5.1 Ajax跨域访问

## 5.2标注

* @Context
* @Provider , javax.ws.rs.ext.Provider

JAX-RS 提供程序是一些应用程序组件，允许在三个关键领域进行运行时行为的定制：数据绑定、异常映射以及上下文解析（比如，向运行时提供 JAXBContext 实例）。每个 JAX-RS 提供程序类必须由 @Provider 注释

## 5.3 Filter

InsecureAmFilter

InsecureAmFilterInitializer

## 5.4 WebApps#withAttributes

## 5.5 ServletContainer

参考文献：

http://kanpiaoxue.iteye.com/blog/2110496

http://liudeh-009.iteye.com/blog/1663224

http://www.ibm.com/developerworks/cn/java/j-lo-jetty/

附录：

Server启动详述：

