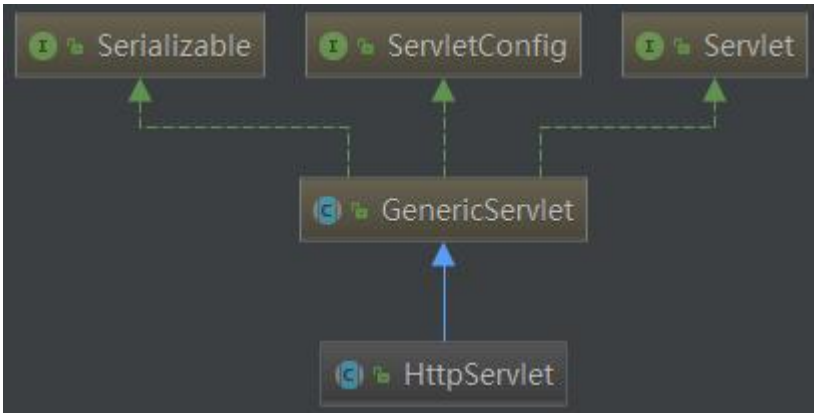
Servlet 和 Tomcat 底层源码分析

一、 Servlet 源码分析

1 Servlet 结构图



Servlet 和 ServletConfig 都是顶层接口,而 GenericServlet 实现了这两个顶层接口,然后 HttpServlet 继承了 GenericServlet 类.所以要实现一个 Servlet 直接就可以继承 HttpServlet

2 Servlet 接口

```
public interface Servlet {
    //负责初始化 Servlet 对象。容器一旦创建好 Servlet 对象后，就调用此方法来初始化 Servlet 对象
    public void init(ServletConfig config) throws ServletException;

    //负责处理客户的请求并返回响应。当容器接收到客户端要求访问特定的 servlet 请求时，就会调用 Servlet 的 service 方法
    public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;

    //Destroy()方法负责释放 Servlet 对象占用的资源，当 servlet 对象结束生命周期时，servlet 容器调用此方法来销毁 servlet 对象.

    public void destroy();


    //说明：Init(),service(),destroy() 这三个方法是 Servlet 生命周期中的最重要的三个方法。

    //返回一个字符串，在该字符串中包含 servlet 的创建者，版本和版权等信息
    public String getServletInfo();
```

```
        //GetServletConfig: 返回一个 ServletConfig 对象，该对象中包含了 Servlet 初始化参数信息

        public ServletConfig getServletConfig();
    }
```

init 方法接收一个 ServletConfig 参数,由容器传入.ServletConfig 就是 Servlet 的配置,在 web.xml 中定义 Servlet 时通过 init-param 标签配置的参数由 ServletConfig 保存

3 ServletConfig 接口

```
    public interface ServletConfig {
        //用于获取 Servlet 名,web.xml 中定义的 servlet-name
        String getServletName();
        //获取 Servlet 上下文对象(非常重要)
        ServletContext getServletContext();
        //获取 init-param 中的配置参数
        String getInitParameter(String var1);
        //获取配置的所有 init-param 名字集合
        Enumeration<String> getInitParameterNames();
    }
```

ServletConfig 是 Servlet 级别,而 ServletContext 是全局的

4 GenericServlet 抽象类

GenericServlet 是 Servlet 的默认实现,是与具体协议无关的

```
  //抽象类 GenericServlet 实现了 Servlet 接口的同时，也实现了 ServletConfig 接口和 Serializable 这两个接口
    public abstract class GenericServlet
        implements Servlet, ServletConfig, java.io.Serializable
    {
        //私有变量，保存 init()传入的 ServletConfig 对象的引用
        private transient ServletConfig config;

        //无参的构造方法
        public GenericServlet() { }
        ------------------------------------
        以下方法实现了 servlet 接口中的 5 个方法
        实现 Servlet 接口方法开始
        ------------------------------------
        /*
        实现接口 Servlet 中的带参数的 init(ServletConfig Config)方法，将传递的 ServletConfig 对象的引用保存到私有成员变量中，
        使得 GenericServlet 对象和一个 ServletConfig 对象关联.
        同时它也调用了自身的不带参数的 init()方法
```

```
        **/

        public void init(ServletConfig config) throws ServletException {
            this.config = config;
            this.init();     //调用了无参的  init()方法
        }

        //无参的 init()方法
        public void init() throws ServletException {

        }
        //空实现了 destroy 方法
        public void destroy() { }
        //实现了接口中的 getServletConfig 方法，返回 ServletConfig 对象
        public ServletConfig getServletConfig()
        {
            return config;
        }
        //该方法实现接口<Servlet>中的 ServletInfo，默认返回空字符串
        public String getServletInfo() {
            return "";
        }

        //唯一没有实现的抽象方法 service()，仅仅在此声明。交由子类去实现具体的应用
        //在后来的 HttpServlet 抽象类中，针对当前基于 Http 协议的 Web 开发，HttpServlet 抽象类具体实现了这个方法
        //若有其他的协议，直接继承本类后实现相关协议即可，具有很强的扩展性
        public abstract void service(ServletRequest req, ServletResponse res)
                                                                    throws ServletException, IOException;
        ----------------------------------
        实现 Servlet 接口方法结束
        ----------------------------------


    ------------------------------------------
        以下四个方法实现了接口 ServletConfig 中的方法
        实现 ServletConfig 接口开始
    ------------------------------------------
    //该方法实现了接口<ServletConfig>中的 getServletContext 方法，用于返回 servleConfig 对象中所包含的 servletContext 方法
        public ServletContext getServletContext() {
            return getServletConfig().getServletContext();
        }
```

```
        //获取初始化参数
        public String getInitParameter(String name) {
          return getServletConfig().getInitParameter(name);
        }

        //实现了接口<ServletConfig>中的方法，用于返回在 web.xml 文件中为 servlet 所配置的全部的初始化参数的值
        public Enumeration getInitParameterNames() {
            return getServletConfig().getInitParameterNames();

       //获取在 web.xml 文件中注册的当前的这个 servlet 名称。没有在 web.xml 中注册的servlet，该方法直接放回该 servlet 的类名。
       //法实现了接口<ServleConfig>中的 getServletName 方法
        public String getServletName() {
             return config.getServletName();
        }
      ---------------------------------------------
       实现 ServletConfig 接口结束
      ---------------------------------------------

        public void log(String msg) {
            getServletContext().log(getServletName() + ": "+ msg);
        }

        public void log(String message, Throwable t) {
            getServletContext().log(getServletName() + ": " + message, t);
        }
    }
```

5 基于协议的 HttpServlet

```
    public abstract class HttpServlet extends GenericServlet
        implements java.io.Serializable
    {

        private static final String METHOD_GET = "GET";
        private static final String METHOD_POST = "POST";
        ......


        /**
```

```
     * Does nothing, because this is an abstract class.
     *  抽象类 HttpServlet 有一个构造函数，但是空的，什么都没有
     */
    public HttpServlet() { }



    /*分别执行 doGet,doPost,doOpitions,doHead,doPut,doTrace 方法
    在请求响应服务方法 service()中，根据请求类型，分贝调用这些 doXXXX 方法
    所以自己写的 Servlet 只需要根据请求类型覆盖响应的 doXXX 方法即可。
    */

    //doXXXX 方法开始
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
    {
        String protocol = req.getProtocol();
        String msg = lStrings.getString("http.method_get_not_supported");
        if (protocol.endsWith("1.1")) {
            resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED,
msg);
        } else {
            resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
        }
    }


    protected void doHead(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
    {
        .......
    }
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
    {
        String protocol = req.getProtocol();
        String msg = lStrings.getString("http.method_post_not_supported");
        if (protocol.endsWith("1.1")) {
            resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED,
msg);
        } else {
            resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
        }
    }
```

```
    protected void doPut(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException    {
            //todo
    }


    protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        //todo
    }

    protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException     {
       //todo
    }

    protected void doDelete(HttpServletRequest req,
                                    HttpServletResponse resp)
            throws ServletException, IOException     {
            //todo
    }
    //doXXXX 方法结束

    //重载的 service(args0,args1)方法
    protected void service(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
    {
            String method = req.getMethod();

            if (method.equals(METHOD_GET)) {
                    long lastModified = getLastModified(req);
                    if (lastModified == -1) {
                            // servlet doesn't support if-modified-since, no reason
                            // to go through further expensive logic
                            doGet(req, resp);
                    } else {
                            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
                            if (ifModifiedSince < (lastModified / 1000 * 1000)) {
                                    // If the servlet mod time is later, call doGet()
                                    // Round down to the nearest second for a proper compare
                                    // A ifModifiedSince of -1 will always be less
                                    maybeSetLastModified(resp, lastModified);
                                    doGet(req, resp);
                            } else {
```

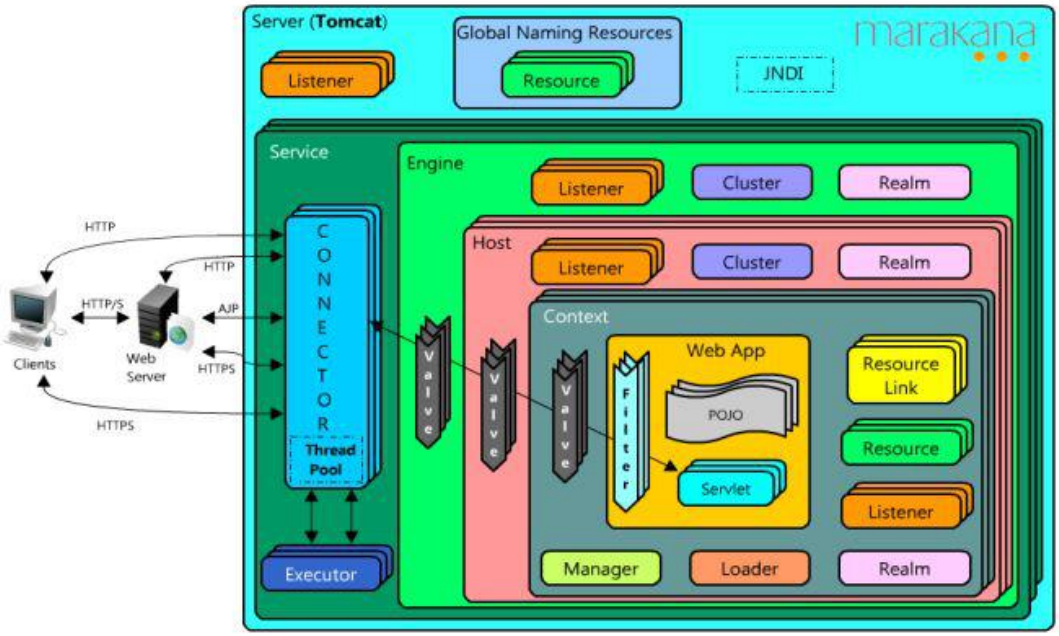
```
                        resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
                    }
                }

            } else if (method.equals(METHOD_HEAD)) {
                long lastModified = getLastModified(req);
                maybeSetLastModified(resp, lastModified);
                doHead(req, resp);

            } else if (method.equals(METHOD_POST)) {
                doPost(req, resp);

            } else if (method.equals(METHOD_PUT)) {
                doPut(req, resp);

            } else if (method.equals(METHOD_DELETE)) {
                doDelete(req, resp);

            } else if (method.equals(METHOD_OPTIONS)) {
                doOptions(req,resp);

            } else if (method.equals(METHOD_TRACE)) {
                doTrace(req,resp);

            } else {
                //
                // Note that this means NO servlet supports whatever
                // method was requested, anywhere on this server.
                //

                String errMsg = lStrings.getString("http.method_not_implemented");
                Object[] errArgs = new Object[1];
                errArgs[0] = method;
                errMsg = MessageFormat.format(errMsg, errArgs);

                resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
            }
        }


    //实现父类的 service(ServletRequest req,ServletResponse res)方法
    // 通过参数的向下转型，然后调用重载的
service(HttpservletRequest,HttpServletResponse)方法
```

```
    public void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException
    {
            HttpServletRequest              request;
            HttpServletResponse              response;

            try {
                request = (HttpServletRequest) req;    //向下转型
                response = (HttpServletResponse) res; //参数向下转型
            } catch (ClassCastException e) {
                throw new ServletException("non-HTTP request or response");
            }
            service(request, response);    //调用重载的 service()方法
    }

    ......//其他方法
}
```

HttpServlet 是基于 Http 协议实现的 Servlet 基类,我们在写 Servlet 的时候直接继承它就行了.SpringMVC 中的 DispatchServlet 就是继承了 HttpServlet.HttpServlet 重新了 service 方法,而 service 方法首先将 ServletRequest 和 ServletResponse 转成 HttpServletRequest 和 HttpServletResponse,然后根据 Http 不同类型的请求,再路由到不同的处理方法进行处理

二、 Tomcat 源码分析

1 Tomcat 架构图



1.1 Server

Server 服务器的意思，代表整个 tomcat 服务器，一个 tomcat 只有一个 Server

Server 中包含至少一个 Service 组件，用于提供具体服务。这个在配置文件中也得到很好的体现（port=“8005” shutdown="SHUTDOWN"是在 8005 端口监听到"SHUTDOWN"命令，服务器就会停止）

1.2 Service

Service 中的一个逻辑功能层， 一个 Server 可以包含多个 Service

Service 接收客户端的请求，然后解析请求，完成相应的业务逻辑，然后把处理后的结果返回给客户端，一般会提供两个方法，一个 start 打开服务 Socket 连接，监听服务端口，一个 stop 停止服务释放网络资源。

1.3 Connector

称作连接器，是 Service 的核心组件之一，一个 Service 可以有多个 Connector，主要是连接客户端请求,用于接受请求并将请求封装成 Request 和 Response，然后交给 Container 进行处理，Container 处理完之后在交给 Connector 返回给客户端。

1.4 Container

Service 的另一个核心组件，按照层级有 Engine，Host，Context，Wrapper 四种，一个 Service 只有一个 Engine，其主要作用是执行业务逻辑

1.5 Engine

一个 Service 中有多个 Connector 和一个 Engine，Engine 表示整个 Servlet 引擎，一个 Engine 下面可以包含一个或者多个 Host，即一个 Tomcat 实例可以配置多个虚拟主机，默认的情况下 conf/server.xml 配置文件中<Engine name="Catalina" defaultHost="localhost"> 定义了一个名为 Catalina 的 Engine。

一个 Engine 包含多个 Host 的设计，使得一个服务器实例可以承担多个域名的服务

1.6 Host

代表一个站点，也可以叫虚拟主机，一个 Host 可以配置多个 Context，在 server.xml 文件中的默认配置为 <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">, 其中 appBase=webapps， 也就是<CATALINA_HOME>\webapps 目录，unpackingWARS=true 属性指定在 appBase 指定的目录中的 war 包都自动的解压，autoDeploy=true 属性指定对加入到 appBase 目录的 war 包进行自动的部署。

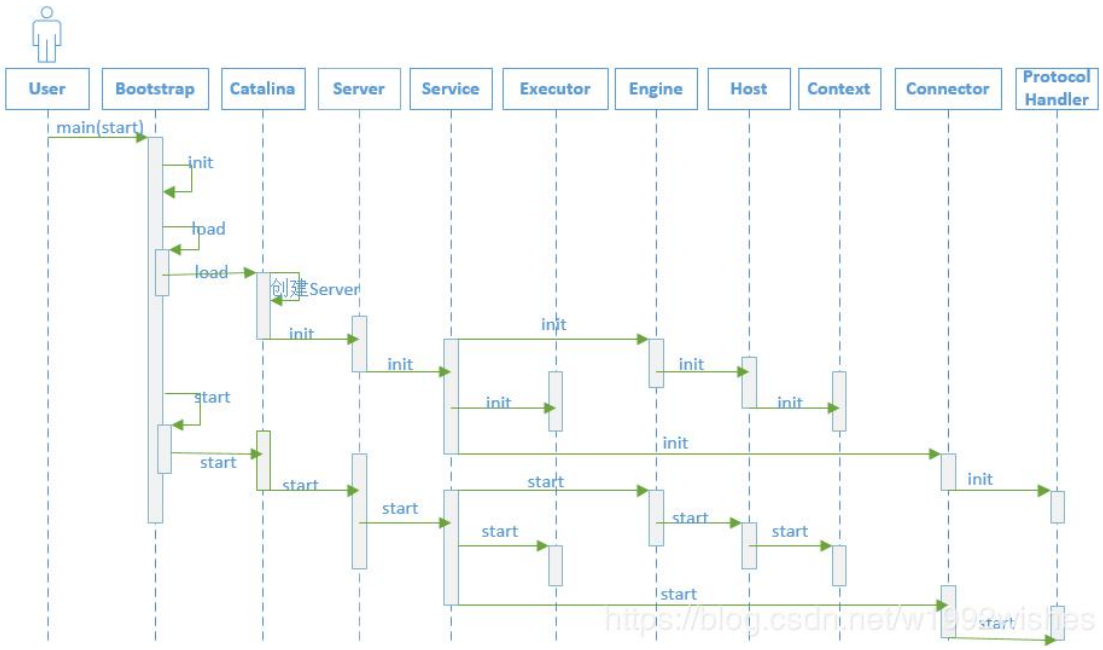
1.7 Context

Context，代表一个应用程序，就是日常开发中的 web 程序，或者一个 WEB-INF 目录以及下面的 web.xml 文件，换句话说每一个运行的 webapp 最终都是以 Context 的形式存在，每个 Context 都有一个根路径和请求路径；与 Host 的区别是 Context 代表一个应用，如，默认配置下 webapps 下的每个目录都是一个应用，其中 ROOT 目录中存放主应用，其他目录存放别的子应用，而整个 webapps 是一个站点。

2 Tomcat 启动源码分析

2.1 启动流程

tomcat 的启动流程很标准化，入口是 BootStrap，统一按照生命周期管理接口 Lifecycle 的定义进行启动。首先，调用 init()方法逐级初始化，接着调用 start()方法进行启动，同时，每次调用伴随着生命周期状态变更事件的触发。



2.2 启动文件分析

2.2.1 Startup.bat

2.2.2 catalina.bat

```
set "CLASSPATH=%CLASSPATH%%CATALINA_HOME%\bin\bootstrap.jar"
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
```

3 Bootstrap

main 方法是整个 tomcat 启动时的入口。在 main 方法中，使用 bootstrap.init()来初始化类加载器和创建 Catalina 实例，然后再启动 Catalina 线程

3.1 bootstrap.init()方法

用于初始化容器相关，首先创建类加载器，然后通过反射创建 org.apache.catalina.startup.Catalina 实例。

4 Catalina

4.1 Lifecycle 接口

Lifecycle 提供一种统一的管理对象生命周期的接口。通过 Lifecycle、LifecycleListener、

LifecycleEvent，Catalina 实现了对 tomcat 各种组件、容器统一的启动和停止的方式。

在 Tomcat 服务开启过程中启动的一些列组件、容器，都实现了 org.apache.catalina.Lifecycle 这个接口，其中的 init()、start() 方法、stop() 方法，为其子类实现了统一的 start 和 stop 管理

4.2 load 方法解析 server.xml 配置文件

load 方法解析 server.xml 配置文件，并加载 Server、Service、Connector、Container、Engine、Host、Context、Wrapper 一系列的容器。加载完成后，调用 initialize()来开启一个新的 Server

4.3 Digester 类解析 server.xml 文件

利用 Digester 类解析 server.xml 文件，得到容器的配置。

4.4 demon.start()

demon.start()方法会调用 Catalina 的 start 方法

Catalina 实例执行 start 方法。这里有两个点，一个是 load()加载 server.xml 配置、初始化 Server 的过程，一个是 getServer().start()开启服务、初始化并开启一系列组件、子容器的过程

5 StandardServer

5.1.1 service.initialize()

然后拿到 StandardServer 实例调用 initialize()方法初始化 Tomcat 容器的一系列组件。一些容器初始化的的时候，都会调用其子容器的 initialize()方法，初始化它的子容器。顺序是 StandardServer、StandardService、StandardEngine、Connector。每个容器都在初始化自身相关设置的同时，将子容器初始化。