---

servlet规范

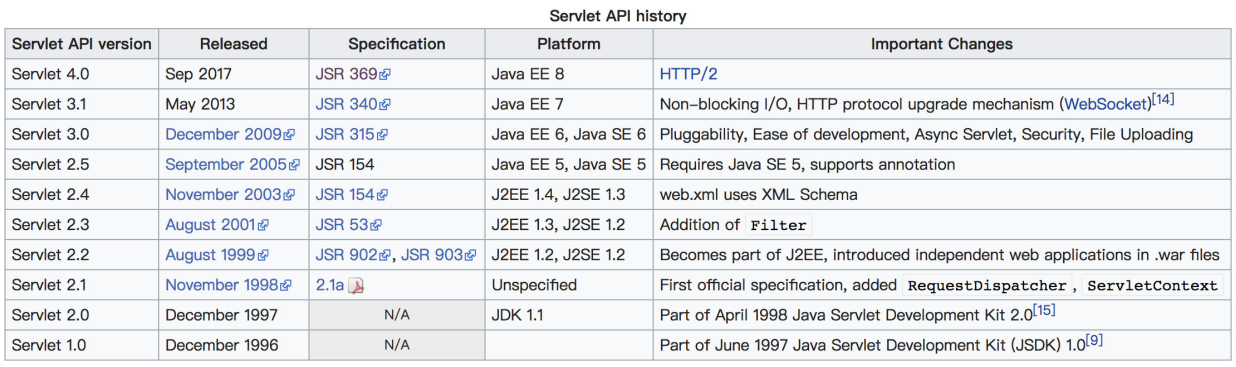
1 servlet 3.1规范

1.1 What is servlet

```
A servlet is a JavaTM technology-based Web component, managed by a
container, that generates dynamic content.
```

> From servlet 3.1

1.2 History


Servlet API history

Servlet API version	Released	Specification	Platform	Important Changes
Servlet 4.0	Sep 2017	JSR 369	Java EE 8	HTTP/2
Servlet 3.1	May 2013	JSR 340	Java EE 7	Non-blocking I/O, HTTP protocol upgrade mechanism (WebSocket)[14]
Servlet 3.0	December 2009	JSR 315	Java EE 6, Java SE 6	Pluggability, Ease of development, Async Servlet, Security, File Uploading
Servlet 2.5	September 2005	JSR 154	Java EE 5, Java SE 5	Requires Java SE 5, supports annotation
Servlet 2.4	November 2003	JSR 154	J2EE 1.4, J2SE 1.3	web.xml uses XML Schema
Servlet 2.3	August 2001	JSR 53	J2EE 1.3, J2SE 1.2	Addition of `Filter`
Servlet 2.2	August 1999	JSR 902, JSR 903	J2EE 1.2, J2SE 1.2	Becomes part of J2EE, introduced independent web applications in .war files
Servlet 2.1	November 1998	2.1a	Unspecified	First official specification, added `RequestDispatcher`, `ServletContext`
Servlet 2.0	December 1997	N/A	JDK 1.1	Part of April 1998 Java Servlet Development Kit 2.0[15]
Servlet 1.0	December 1996	N/A		Part of June 1997 Java Servlet Development Kit (JSDK) 1.0[9]

> From wiki

1.3 Servlet Life Cycle

- Loading and Instantiation

  When the servlet engine is started, needed servlet classes must be located by the servlet container(WEB-INF/lib)

- Initialization

  ```
  The container initializes the servlet instance by calling the init
  method of the Servlet interface with a unique (per servlet
  declaration) object implementing the ServletConfig interface
  ```

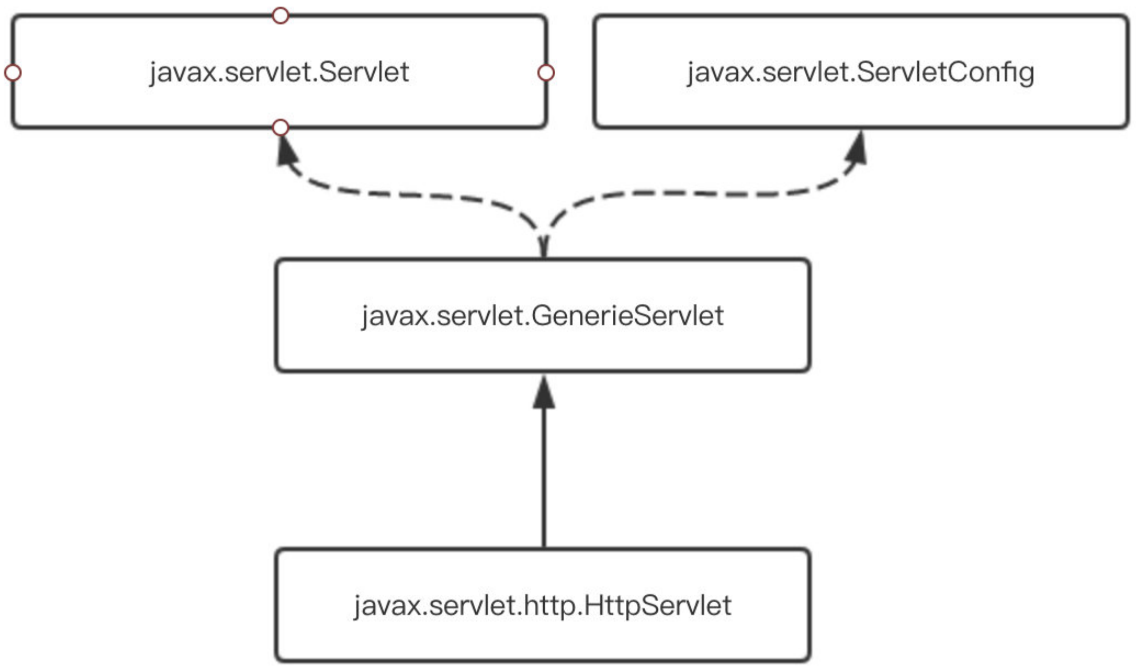
(ServletConfig used by Servlet Container)

- Request Handling

  After a servlet is properly initialized, the servlet container may use it to handle client requests.

- End of Service

1.4 Servlet 继承结构


javax.servlet.Servlet
javax.servlet.ServletConfig
javax.servlet.GenerieServlet
javax.servlet.http.HttpServlet

1.5 ServletContext

```
    The ServletContext interface defines a servlet’s view of the Web
application within which the servlet is running.  (web.xml) The Container
Provider is responsible for providing an implementation of the
ServletContext interface in the servlet container.
```

```
InitParameter
config
    -Filter
    -Listenr
    -Servlet
Attribute
Resource
...
```

> see : ApplicationContext、ApplicationContextFacade (tomcat)

1.6 Request

HttpServletRequest

- HTTP Protocol Parameters

  > - getParameter
  > - getParameterNames
  > - getParameterValues
  > - getParameterMap

- File upload

  content-type : multipart/form-data

- Attributes
- Headers
- Request Path Elements

  > requestURI = contextPath + servletPath + pathInfo

- Path Translation Methods

  > - ServletContext.getRealPath
  > - HttpServletRequest.getPathTranslated

- Non Blocking IO

  Non-blocking IO only works with async request processing in Servlets and Filters

- Cookies
- SSL
- Internationalization

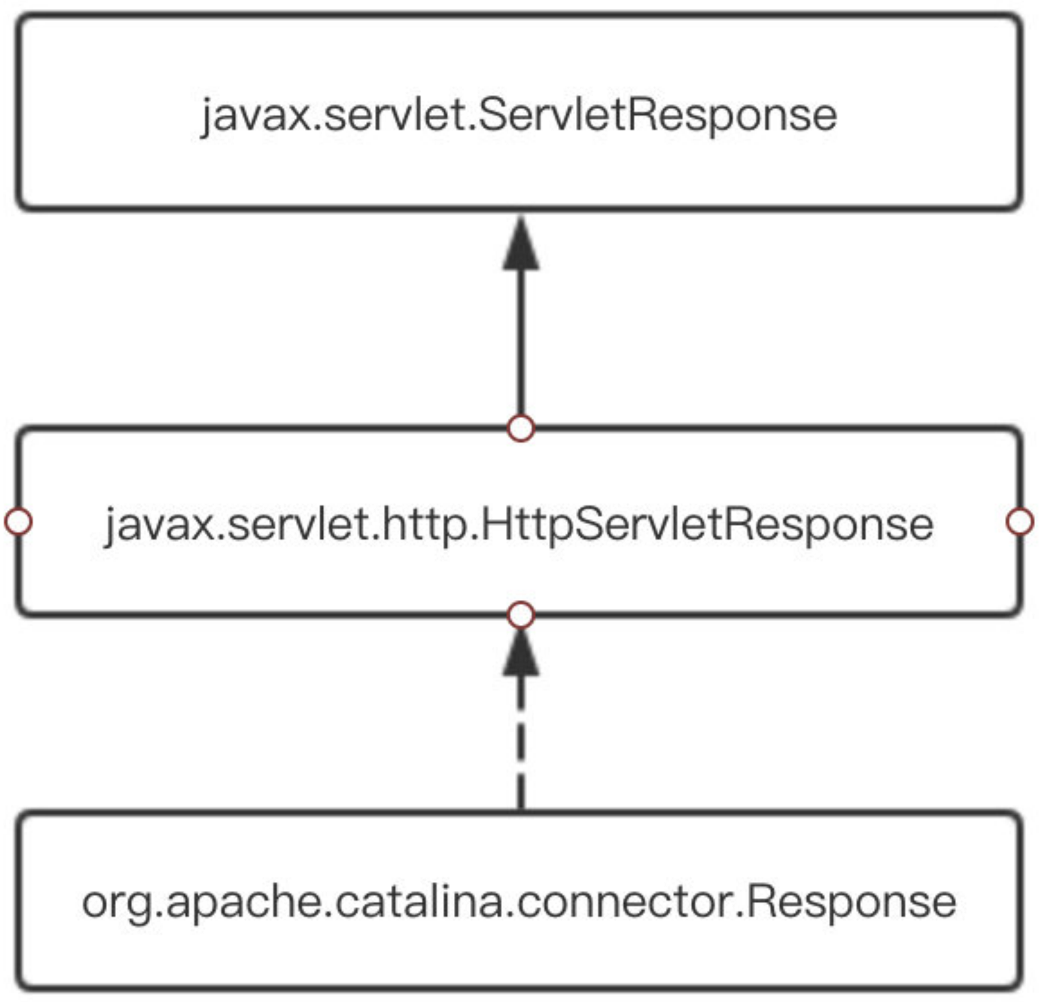
Accept-Language : zh-cn

- > ■ getLocale ■ getLocales
- Request data encoding

  The default encoding of a request the container uses to create the request reader and parse POST data must be “ISO-8859-1” if none has been specified by the client request.
- Lifetime of the Request Object

  Each request object is valid only within the scope of a servlet’s service method, or within the scope of a filter’s doFilter method, unless the asynchronous processing is enabled for the component and the startAsync method is invoked on the request object.

1.7 Response


javax.servlet.ServletResponse
javax.servlet.http.HttpServletResponse
org.apache.catalina.connector.Response

1.8 Filter

what is Filter?

```
A filter is a reusable piece of code that can transform the content of HTTP
requests,
```

responses, and header information.

> `org.springframework.web.servlet.HandlerInterceptor`

1.9 Lifecycle Events

```
Event
    -Servlet
    -Session
    -Request
EventListener
    -Servlet
    -Session
    -Request
```

Example : `ServletContextListener`

```java
public class ContextLoaderListener implements ServletContextListener {
    private ContextLoader contextLoader;

    public ContextLoaderListener() {
    }

    public void contextInitialized(ServletContextEvent event) {
        this.contextLoader = this.createContextLoader();
 this.contextLoader.initWebApplicationContext(event.getServletContext());
    }

    protected ContextLoader createContextLoader() {
        return new ContextLoader();
    }

    public ContextLoader getContextLoader() {
        return this.contextLoader;
    }

    public void contextDestroyed(ServletContextEvent event) {
        if(this.contextLoader != null) {
 this.contextLoader.closeWebApplicationContext(event.getServletContext());
        }

    }
}
```

1.10 Session

2 Servlet

```
Server + Applet 的缩写，表示一个服务器应用
```

2.1 Servlet接口

```java
package javax.servlet;
import java.io.IOException;
public interface Servlet {
    public void init(ServletConfig config) throws ServletException;

    public ServletConfig getServletConfig();

    public void service(ServletRequest req, ServletResponse res)
   throws ServletException, IOException;

    public String getServletInfo();

    public void destroy();

}
```

> Load-on-startup 为负的话不会在容器启动调用

2.2 ServletConfig接口

```java
package javax.servlet;

import java.util.Enumeration;

/**
 * A servlet configuration object used by a servlet container
 * to pass information to a servlet during initialization.
 */
 public interface ServletConfig {

    public String getServletName();

    public ServletContext getServletContext();

    public String getInitParameter(String name);

    public Enumeration<String> getInitParameterNames();

}
```

如下配置

```
<!--web.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>application-context.xml</param-value>
    </context-param>
<servlet>
    <servlet-name>demoDispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.Dispatcher</servlet-
class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>demo-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
</web-app>
```

在Servlet中 可以分别通过它们的getInitParameter方法获取，比如：

```
String contextLocation =
getServletConfig().getServletContext().getInitParameter(
    "contextConfigLocation");
String servletLocation =
getServletConfig().getInitParameter("contextConfigLocation");
```

2.3 GenerieServlet

```
`Servlet`的默认实现，同时实现了`ServletConfig`接口、`Serializable`接口，所以可以直
接调用`ServletConfig`里面的方法。详细可参考如下类注释。
```

```
package javax.servlet;

import java.io.IOException;
import java.util.Enumeration;

/**
 *
 * Defines a generic, protocol-independent
 * servlet. To write an HTTP servlet for use on the
 * Web, extend {@link javax.servlet.http.HttpServlet} instead.
 *
 * <p><code>GenericServlet</code> implements the <code>Servlet</code>
 * and <code>ServletConfig</code> interfaces. <code>GenericServlet</code>
 * may be directly extended by a servlet, although it's more common to
extend
```

```
 * a protocol-specific subclass such as <code>HttpServlet</code>.
 *
 * <p><code>GenericServlet</code> makes writing servlets
 * easier. It provides simple versions of the lifecycle methods
 * <code>init</code> and <code>destroy</code> and of the methods
 * in the <code>ServletConfig</code> interface. <code>GenericServlet</code>
 * also implements the <code>log</code> method, declared in the
 * <code>ServletContext</code> interface.
 *
 * <p>To write a generic servlet, you need only
 * override the abstract <code>service</code> method.
 *
 */
public abstract class GenericServlet
    implements Servlet, ServletConfig, java.io.Serializable
{
    private transient ServletConfig config;

    public GenericServlet() {}

    public void destroy() {}

    public String getInitParameter(String name) {
        return getServletConfig().getInitParameter(name);
    }

    public Enumeration getInitParameterNames() {
        return getServletConfig().getInitParameterNames();
    }

    public ServletConfig getServletConfig() {
        return config;
    }

    public ServletContext getServletContext() {
        return getServletConfig().getServletContext();
    }

    public String getServletInfo() {
        return "";
    }

    public void init(ServletConfig config) throws ServletException {
        this.config = config;
        this.init();
    }

    public void init() throws ServletException {
    }

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

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

    public abstract void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;

    public String getServletName() {
        return config.getServletName();
    }
}
```

附：为什么需要实现 `java.io.Serializable` 接口?

答：在 Servlet 2.4 规范的 7.7.2 Distributed Environments 章节中，有一句这样的描述：

```
The distributed servlet container must support the mechanism necessary for

migrating objects that implement Serializable.
```

> 按照规范的设计，Servlet 有一个钝化的特性，类似于 Servlet 持久化到文件，然后当容器 Crash 回复后，可以重新恢复保存前的状态。

2.4 HttpServlet

```
package javax.servlet.http;

import ....;

/**
 *
 * Provides an abstract class to be subclassed to create
 * an HTTP servlet suitable for a Web site. A subclass of
 * <code>HttpServlet</code> must override at least
 * one method, usually one of these:
 *
 * <ul>
 * <li> <code>doGet</code>, if the servlet supports HTTP GET requests
```

```java
 * <li> <code>doPost</code>, for HTTP POST requests
 * <li> <code>doPut</code>, for HTTP PUT requests
 * <li> <code>doDelete</code>, for HTTP DELETE requests
 * <li> <code>init</code> and <code>destroy</code>,
 * to manage resources that are held for the life of the servlet
 * <li> <code>getServletInfo</code>, which the servlet uses to
 * provide information about itself
 * </ul>
 *
 * <p>There's almost no reason to override the <code>service</code>
 * method. <code>service</code> handles standard HTTP
 * requests by dispatching them to the handler methods
 * for each HTTP request type (the <code>do</code><i>XXX</i>
 * methods listed above).
 *
 * <p>Likewise, there's almost no reason to override the
 * <code>doOptions</code> and <code>doTrace</code> methods.
 *
 * <p>Servlets typically run on multithreaded servers,
 * so be aware that a servlet must handle concurrent
 * requests and be careful to synchronize access to shared resources.
 * Shared resources include in-memory data such as
 * instance or class variables and external objects
 * such as files, database connections, and network
 * connections.
 * See the
 * <a
href="http://java.sun.com/Series/Tutorial/java/threads/multithreaded.html">
 * Java Tutorial on Multithreaded Programming</a> for more
 * information on handling multiple threads in a Java program.
 *
 * @author  Various
 * @version $Version$
 *
 */
public abstract class HttpServlet extends GenericServlet
    implements java.io.Serializable
{
    private static final String METHOD_DELETE = "DELETE";
    private static final String METHOD_HEAD = "HEAD";
    private static final String METHOD_GET = "GET";
    private static final String METHOD_OPTIONS = "OPTIONS";
    private static final String METHOD_POST = "POST";
    private static final String METHOD_PUT = "PUT";
    private static final String METHOD_TRACE = "TRACE";

    private static final String HEADER_IFMODSINCE = "If-Modified-Since";
    private static final String HEADER_LASTMOD = "Last-Modified";
```

```
/**
* Resource bundles contain locale-specific objects.
*/
private static final String LSTRING_FILE =
"javax.servlet.http.LocalStrings";
private static ResourceBundle lStrings =
ResourceBundle.getBundle(LSTRING_FILE);

public HttpServlet() { }

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 long getLastModified(HttpServletRequest req) {
    return -1;
}

protected void doHead(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException{
    NoBodyResponse response = new NoBodyResponse(resp);

    doGet(req, response);
    response.setContentLength();
}

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{
   //略,类似doPost
}
```

```
    protected void doDelete(HttpServletRequest req,
                HttpServletResponse resp)
    throws ServletException, IOException{
       //略,类似doPost
    }

    protected void doOptions(HttpServletRequest req, HttpServletResponse
resp)
    throws ServletException, IOException{
      //略,主要用于调试,输出允许类型
    }

    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);
        }
    }

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

        try {
            request = (HttpServletRequest) req;
            response = (HttpServletResponse) res;
        } catch (ClassCastException e) {
            throw new ServletException("non-HTTP request or response");
        }
        service(request, response);
    }
}

/*
 * A response that includes no body, for use in (dumb) "HEAD" support.
 * This just swallows that body, counting the bytes in order to set
 * the content length appropriately.  All other methods delegate directly
 * to the HTTP Servlet Response object used to construct this one.
 */
// file private
class NoBodyResponse extends HttpServletResponseWrapper {
    private NoBodyOutputStream      noBody;
    private PrintWriter         writer;
    private boolean         didSetContentLength;

    // file private
    NoBodyResponse(HttpServletResponse r) {
        super(r);
    noBody = new NoBodyOutputStream();
    }
    // ....
}

/*
 * Servlet output stream that gobbles up all its data.
 */

// file private
class NoBodyOutputStream extends ServletOutputStream {
    //...
}
```

```
doXXX 都是模板方法，如果子类没有实现将抛出异常

doGet 方法前还会对是否过期做检查，如果没有过期，则直接返回304状态码做缓存。

doHead调用了doGet的请求，然后返回空body的response

doOptions和doTrace 主要是用来做一些调试工作
```

3 servlet容器 tomcat

3.1 Tomcat的顶层结构

```
Catalina 管理整个Tomcat的管理类

Server 最顶层容器，代表整个服务器

    Service 提供具体服务 （多个）

        Connector 负责网络连接、request/response的创建（可以有多个连接，从servet.xml的配置也可以看出，同时提供http和https，也可以提供相同协议不同端口的连接）

        Container 具体处理Servlet
```

3.2 Bootstrap

```
`org.apache.catalina.startup.Bootstrap` 是Tomcat的入口，作用类似一个
`CatalinaAdptor`，具体处理还是Catalina来完成，这样做的好处是可以把启动的入口和具体的
管理类分开，从而可以很方便地创建出多种启动方式。
```

> BootStrap不在Tomcat依赖包下， 而是在bin目录 通过反射 完全松耦合

```
package org.apache.catalina.startup;

import ...;

public final class Bootstrap {

    private static final Log log = LogFactory.getLog(Bootstrap.class);

    /**
     * Daemon object used by main.
     */
    private static Bootstrap daemon = null;
    /**
     * Daemon reference.
     */
    private Object catalinaDaemon = null;

    ClassLoader commonLoader = null;
    ClassLoader catalinaLoader = null;
    ClassLoader sharedLoader = null;


    private void initClassLoaders() {
        try {
            commonLoader = createClassLoader("common", null);
            if( commonLoader == null ) {
                // no config file, default to this loader - we might be in
a 'single' env.
                commonLoader=this.getClass().getClassLoader();
            }
            catalinaLoader = createClassLoader("server", commonLoader);
            sharedLoader = createClassLoader("shared", commonLoader);
        } catch (Throwable t) {
            handleThrowable(t);
            log.error("Class loader creation threw exception", t);
            System.exit(1);
        }
    }

    private ClassLoader createClassLoader(String name, ClassLoader parent)
        throws Exception {
```

```java
        String value = CatalinaProperties.getProperty(name + ".loader");
        if ((value == null) || (value.equals("")))
            return parent;

        value = replace(value);

        List<Repository> repositories = new ArrayList<>();

        String[] repositoryPaths = getPaths(value);

        for (String repository : repositoryPaths) {
            // Check for a JAR URL repository
            try {
                @SuppressWarnings("unused")
                URL url = new URL(repository);
                repositories.add(
                        new Repository(repository, RepositoryType.URL));
                continue;
            } catch (MalformedURLException e) {
                // Ignore
            }

            // Local repository
            if (repository.endsWith("*.jar")) {
                repository = repository.substring
                    (0, repository.length() - "*.jar".length());
                repositories.add(
                        new Repository(repository, RepositoryType.GLOB));
            } else if (repository.endsWith(".jar")) {
                repositories.add(
                        new Repository(repository, RepositoryType.JAR));
            } else {
                repositories.add(
                        new Repository(repository, RepositoryType.DIR));
            }
        }

        return ClassLoaderFactory.createClassLoader(repositories, parent);
    }

    /**
     * Initialize daemon.
     * @throws Exception Fatal initialization error
     */
    public void init() throws Exception {

        initClassLoaders();

        Thread.currentThread().setContextClassLoader(catalinaLoader);
        SecurityClassLoad.securityClassLoad(catalinaLoader);

        // Load our startup class and call its process() method
        if (log.isDebugEnabled())
            log.debug("Loading startup class");
        Class<?> startupClass =
            catalinaLoader.loadClass
            ("org.apache.catalina.startup.Catalina");
        Object startupInstance = startupClass.newInstance();

        // Set the shared extensions class loader
        if (log.isDebugEnabled())
            log.debug("Setting startup class properties");
        String methodName = "setParentClassLoader";
        Class<?> paramTypes[] = new Class[1];
        paramTypes[0] = Class.forName("java.lang.ClassLoader");
        Object paramValues[] = new Object[1];
        paramValues[0] = sharedLoader;
        Method method =
            startupInstance.getClass().getMethod(methodName, paramTypes);
        method.invoke(startupInstance, paramValues);

        catalinaDaemon = startupInstance;

    }

    /**
     * Load daemon.
     */
    private void load(String[] arguments)
        throws Exception {

        // Call the load() method
        String methodName = "load";
        Object param[];
        Class<?> paramTypes[];
        if (arguments==null || arguments.length==0) {
            paramTypes = null;
            param = null;
        } else {
            paramTypes = new Class[1];
            paramTypes[0] = arguments.getClass();
            param = new Object[1];
            param[0] = arguments;
        }
        Method method =
            catalinaDaemon.getClass().getMethod(methodName, paramTypes);
        if (log.isDebugEnabled())
```

```
            log.debug("Calling startup class " + method);
        method.invoke(catalinaDaemon, param);
    }

    // ------------------------------------------------------------ Main
Program

    /**
     * Load the Catalina daemon.
     * @param arguments Initialization arguments
     * @throws Exception Fatal initialization error
     */
    public void init(String[] arguments)
        throws Exception {
        init();
        load(arguments);

    }

    /**
     * Start the Catalina daemon.
     * @throws Exception Fatal start error
     */
    public void start()
        throws Exception {
        if( catalinaDaemon==null ) init();

        Method method = catalinaDaemon.getClass().getMethod("start", (Class
[] )null);
        method.invoke(catalinaDaemon, (Object [])null);
    }

    /**
     * Stop the Catalina Daemon.
     * @throws Exception Fatal stop error
     */
    public void stop()
        throws Exception {
        //实现略,主要通过反射调用了catalina的stop
    }

    /**
     * Stop the standalone server.
     * @throws Exception Fatal stop error
     */
    public void stopServer()
        throws Exception {
        //实现略,主要通过反射调用了catalina的stopServer
    }
```

```java
    /**
     * Set flag.
     * @param await <code>true</code> if the daemon should block
     * @throws Exception Reflection error
     */
    public void setAwait(boolean await)
        throws Exception {
        //实现略 ,主要通过反射调用了catalina的setAwait
    }

    public boolean getAwait()
        throws Exception{
        //实现略 ,主要通过反射调用了catalina的getAwait
    }



    /**
     * Destroy the Catalina Daemon.
     */
    public void destroy() {
        // FIXME
    }
    /**
     * Main method and entry point when starting Tomcat via the provided
     * scripts.
     *
     * @param args Command line arguments to be processed
     */
    public static void main(String args[]) {

        if (daemon == null) {
            // Don't set daemon until init() has completed
            Bootstrap bootstrap = new Bootstrap();
            try {
                bootstrap.init();
            } catch (Throwable t) {
                handleThrowable(t);
                t.printStackTrace();
                return;
            }
            daemon = bootstrap;
        } else {
            // When running as a service the call to stop will be on a new
            // thread so make sure the correct class loader is used to
prevent
            // a range of class not found exceptions.

 Thread.currentThread().setContextClassLoader(daemon.catalinaLoader);
```

```
        }

        try {
            String command = "start";
            if (args.length > 0) {
                command = args[args.length - 1];
            }

            if (command.equals("startd")) {
                args[args.length - 1] = "start";
                daemon.load(args);
                daemon.start();
            } else if (command.equals("stopd")) {
                args[args.length - 1] = "stop";
                daemon.stop();
            } else if (command.equals("start")) {
                daemon.setAwait(true);
                daemon.load(args);
                daemon.start();
            } else if (command.equals("stop")) {
                daemon.stopServer(args);
            } else if (command.equals("configtest")) {
                daemon.load(args);
                if (null==daemon.getServer()) {
                    System.exit(1);
                }
                System.exit(0);
            } else {
                log.warn("Bootstrap: command \"" + command + "\" does not
exist.");
            }
        } catch (Throwable t) {
            // Unwrap the Exception for clearer error reporting
            if (t instanceof InvocationTargetException &&
                    t.getCause() != null) {
                t = t.getCause();
            }
            handleThrowable(t);
            t.printStackTrace();
            System.exit(1);
        }

    }
}
```

```
Tomcat 启动脚本 startup.bat 是从main方法中开始的。其中主要做了:
```

- 准备容器环境, `init()`初始化类加载器,

- 初始化容器，调用`load()`实际是调用catalina里的`init()`
- 启动容器，通过引用`catalinaDaemon`反射射调用`start()`方法（实际还是通过catalina操作容器）

  关于类加载，我们都知道 JSEE 默认的类加载机制是双亲委派原则（详细查看如下🔎https://www.cnblogs.com/miduos/p/9250565.html）

  通过debug可以发现 commonLoader、catalinaLoader 、sharedLoader 其实三个是同一个，原因是因为`catalina.properties`的配置中默认是空的。

  另外在`init()`中`Thread.currentThread().setContextClassLoader(catalinaLoader);`

3.3 Catalina的启动过程

```
Catalina的启动主要是调用`setAwait()`、`load()`和`start()`方法来完成。
```

- `setAwait()`方法用于设置Server启动完成后是否进入等待状态的标记
- `load()`方法主要是用来加载配置文件`conf/server.xml`创建Server对象 （解析是通过Digester），然后调用Server的`init()`
- `start()`主要是调用Server的`start()`

3.4 Server的启动过程

```
Server的默认实现`org.apache.catalina.core.StandardServer` ，在其父类中
`org.apache.catalina.util.LifecycleBase` 中的`init() ` 实现如下
```

```java
@Override
public final synchronized void init() throws LifecycleException {
    if (!state.equals(LifecycleState.NEW)) {
        invalidTransition(Lifecycle.BEFORE_INIT_EVENT);
    }

    try {
        setStateInternal(LifecycleState.INITIALIZING, null, false);
        initInternal();
        setStateInternal(LifecycleState.INITIALIZED, null, false);
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        setStateInternal(LifecycleState.FAILED, null, false);
        throw new LifecycleException(
                sm.getString("lifecycleBase.initFail",toString()), t);
    }
}
```

`start()`实现如下

```java
/**
 * {@inheritDoc}
 */
@Override
public final synchronized void start() throws LifecycleException {

    if (LifecycleState.STARTING_PREP.equals(state) ||
LifecycleState.STARTING.equals(state) ||
            LifecycleState.STARTED.equals(state)) {

        if (log.isDebugEnabled()) {
            Exception e = new LifecycleException();
            log.debug(sm.getString("lifecycleBase.alreadyStarted",
toString()), e);
        } else if (log.isInfoEnabled()) {
            log.info(sm.getString("lifecycleBase.alreadyStarted",
toString()));
        }

        return;
    }

    if (state.equals(LifecycleState.NEW)) {
        init();
    } else if (state.equals(LifecycleState.FAILED)) {
        stop();
    } else if (!state.equals(LifecycleState.INITIALIZED) &&
            !state.equals(LifecycleState.STOPPED)) {
        invalidTransition(Lifecycle.BEFORE_START_EVENT);
    }

    try {
        setStateInternal(LifecycleState.STARTING_PREP, null, false);
        startInternal();
        if (state.equals(LifecycleState.FAILED)) {
            // This is a 'controlled' failure. The component put itself
into the
            // FAILED state so call stop() to complete the clean-up.
            stop();
        } else if (!state.equals(LifecycleState.STARTING)) {
            // Shouldn't be necessary but acts as a check that sub-classes
are
            // doing what they are supposed to.
            invalidTransition(Lifecycle.AFTER_START_EVENT);
        } else {
            setStateInternal(LifecycleState.STARTED, null, false);
        }
    } catch (Throwable t) {
        // This is an 'uncontrolled' failure so put the component into the
```

```
            // FAILED state and throw an exception.
            ExceptionUtils.handleThrowable(t);
            setStateInternal(LifecycleState.FAILED, null, false);
            throw new
LifecycleException(sm.getString("lifecycleBase.startFail", toString()), t);
        }
    }
```

```
其中 `startInternal() 和 initInternal()` 为模版方法 , 查看其实现类 可以发现是循环
调用了每个`service`的`start()`和`init()`
```

3.5 Service的启动过程

```
    类似于Server , `StandardService`的`initInternal()`和 `startInternal()`的
方法主要调用`container`、`executors`、`mapperListener`、`connectors`的`init()`
和`start()`方法。
```

4 servlet标准实现 springmvc dispatcherServlet