一、传统Servlet容器

1.Eclipse jetty

embedded 嵌入式

jetty可以提供的作用：

- 异步的HTTP Server
- 标准的Servlet容器
- webSocket server
- http/2 server
- 异步的客户端 （java1-java3都是BIO 之后是NIO是非阻塞IO java7之后是AIO异步IO）
- OSGI JNDI JMX JASPI AJP support

2.Apache Tomcat

因为tomcat中也包含了一个HTTP服务器，它也可以被视为一个单独的WEB服务器

标准实现：

- Servlet
- Jsp
- Expression Language
- WebSocket 保持长连接

Tomcat

（1）核心组件（Components）

（2）静态资源处理

（3）欢迎页面（Welcome file list）

（4）JSP处理

（5）类加载（ClassLoading）

（6）连接器（Connectors）

（7）JDBC数据源

（8）JNDI（java Naming and Directory Interfaces）

下面来详细的去看下这些~

3. Tomcat详解

（1）核心组件（Components）

tomcat的核心组件：

Engine

Host

Context

（2）静态资源处理

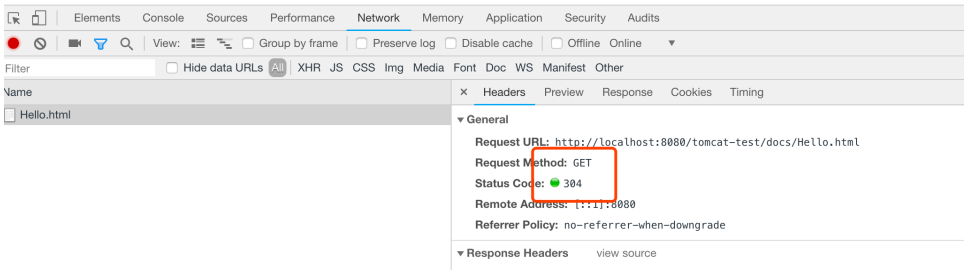
在tomcat中的web.xml文件中 有默认servlet的加载 DefaultServlet

里面有对静态资源的配置，比如是否展示静态资源的列表目录 listing的设置

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
        <!-- 是否开启debug模式 -->
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <!-- 静态资源是否展示目录列表 -->
        <param-name>listings</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

下面就是访问静态资源 docs/hello.html 反映码304 代表no-definend

静态欢迎页面



（3）JSP的处理

Tomcat 的 web.xml中对jsp的配置处理 上面的大段注释就是相关的注释，比如我们可以设置是否进行jsp文件的预编译 通过设置development参数来控制

```
<!-- tomcat中的jsp处理 -->
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>xpoweredBy</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>
```

（4）欢迎页的处理

Tomcat中web.xml配置文件对于欢迎页的配置，但是要注意在project的web.xml也可以配置这个欢迎页，那个会覆盖这里的配置

```
<!-- ==================== Default Welcome File List ==================== -->
<!-- When a request URI refers to a directory, the default servlet looks  -->
<!-- for a "welcome file" within that directory and, if present, to the   -->
<!-- corresponding resource URI for display.                              -->
<!-- If no welcome files are present, the default servlet either serves a -->
<!-- directory listing (see default servlet configuration on how to       -->
<!-- customize) or returns a 404 status, depending on the value of the    -->
<!-- listings setting.                                                    -->
<!--                                                                      -->
<!-- If you define welcome files in your own application's web.xml        -->
<!-- deployment descriptor, that list *replaces* the list configured      -->
<!-- here, so be sure to include any of the default values that you wish  -->
<!-- to use within your application.                                       -->
  <!-- 只要是root目录 会去映射 welcome file list -->
  <welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
      <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
```

（5）类加载（Classloading）
也符合双亲委派的原则

BootStrap ClassLoader （bootStrap的ClassLoaderd）

System ClassLoader（用户指定的classLoader）

Common ClassLoader

- [ ] Webapp ClassLoader

（6）JDBC数据源
在tomcat配置文件中的context.xml文件中可以配置这种数据库连接的数据源

```
<Resource name="jdbc/TestDB" auth="Container" type="javax.sql.DataSource"
              maxTotal="100" maxIdle="30" maxWaitMillis="10000"
              username="javauser" password="javadude" driverClassName="com.mysql.jdbc.Driver"
              url="jdbc:mysql://localhost:3306/javatest"/>
```

然后可以在project的web.xml中引用这个数据源
之后可以写一个servlet去测试jdbc的连接 servlet中的获取连接是用的JNDI技术
可以根据JNDI获取contex，然后从context中拿到对应的数据库的配置

```
public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException,IOException {
    PrintWriter write = response.getWriter();
    // 因为下边要输出<br>标签 所以设置一下输出格式
    response.setContentType("text/html;charset=UTF-8");


    try {
        // 拿到数据库连接
        Connection connection = dataSource.getConnection();

        // statement
        Statement statement = connection.createStatement();

        ResultSet resultSet = statement.executeQuery("SHOW DATABASES");
        // 打印结果集
        while(resultSet.next()) {
            String dataName = resultSet.getString(1);
            write.write(dataName);
            write.write("</br>");
            write.flush();
        }

        connection.close();

    } catch(Exception e) {
        throw new ServletException(e);
    }
```

在web.xml中配置这个servlet 包括访问这个servlet的url-mapping

```
<!-- 注册测试数据库连接的servlet -->
<servlet>
    <servlet-name>JDBCTestServlet</servlet-name>
    <servlet-class>com.segmentfault.lesson5.servlet.JDBCTestServlet</servlet-class>
</servlet>
<!--测试连接数据库的访问mapping url-->
<servlet-mapping>
    <servlet-name>JDBCTestServlet</servlet-name>
    <url-pattern>/test/jdbc</url-pattern>
</servlet-mapping>
```

一个思考题：为什么在做JDBCTestServlet的时候需要序列化接口?

（7）JNDI技术 可以理解为加载资源，就像spring中的context

基本类型

资源（Resource） 比如上边在project的web.xml文件中配置的数据库连接资源

环境（Environment）可以在web.xml中配置一个entry之类的东西 然后在servlet中通过context.lookup读到这个bean

配置方式

context.xml配置

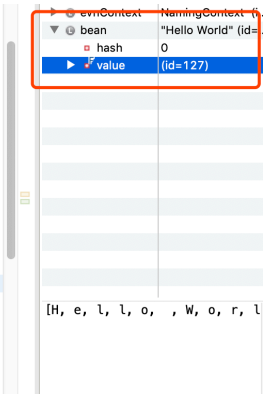
web.xml配置

web.xml中可与配置

```
<!-- JNDI 引入一个bean -->
<env-entry>
    <env-entry-name>Bean</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello World</env-entry-value>
</env-entry>
```

在servlet通过JNDI的context拿到这个值

```java
// 定义数据源
private DataSource dataSource;

public void init(ServletConfig servletConfig) throws ServletException {

    try {
        // 利用JNDI技术获取数据库的连接
        Context context = new InitialContext();
        // 上下文
        Context evnContext = (Context) context.lookup("java:comp/env");
        // 从resource中找到这个dataSource
        dataSource = (DataSource) evnContext.lookup("jdbc/TestDB");

        // 测试获取一个bean
        String bean = (String) evnContext.lookup("Bean");
        System.out.println(bean);
    } catch(Exception e) {
        throw new ServletException(e);
    }
}
```



（8）连接器

Connectors 连接器，通过配置server.xml 里面的Conector节点来配置一些属性去配置

端口 （port）

```
<!-- 连接器的相关配置 -->
<!-- A "Connector" represents an endpoint by which requests are received
     and responses are returned. Documentation at :
     Java HTTP Connector: /docs/config/http.html
     Java AJP  Connector: /docs/config/ajp.html
     APR (HTTP/AJP) Connector: /docs/apr.html
     Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
-->
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
<!-- A "Connector" using the shared thread pool -->
<!--
<Connector executor="tomcatThreadPool"
```

协议（protocol）

这里可以通过tomcat自带的NIO2 去配置

线程池（ThreadPool）

可以通过executor属性去配置对应处理请求的线程池

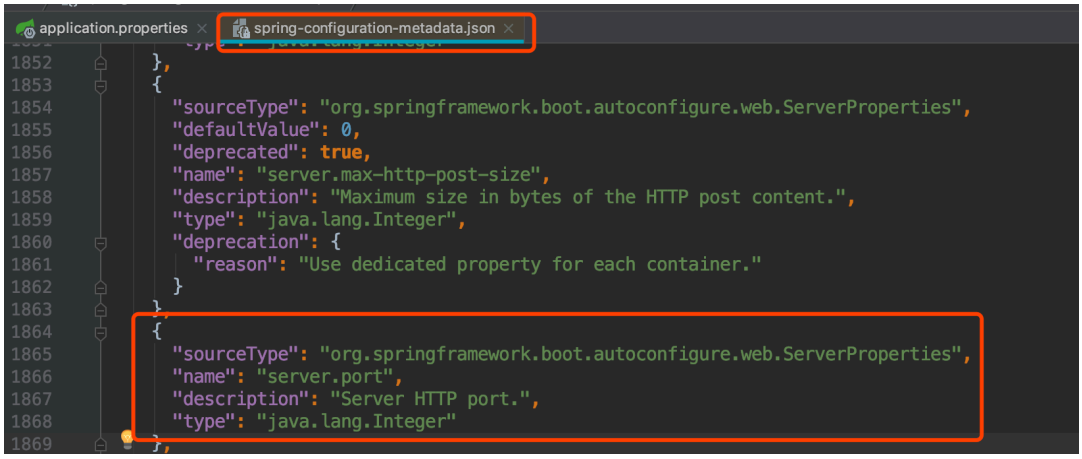
```
<!--The connectors can use a shared executor, you can define one or more named thread pools-->
<!--<Executor name="tomcatThreadPool" namePrefix="tomcat-test-exec-"
    maxThreads="150" minSpareThreads="4"/> -->

<!-- http浏览器请求的连接器的相关配置 -->
<!-- A "Connector" represents an endpoint by which requests are received
     and responses are returned. Documentation at :
     Java HTTP Connector: /docs/config/http.html
     Java AJP  Connector: /docs/config/ajp.html
     APR (HTTP/AJP) Connector: /docs/apr.html
     Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
-->
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
<!-- A "Connector" using the shared thread pool -->
<!--
```

超时时间（TimeOut）

同样也是去设置属性

二、spring boot嵌入式容器

可以根据已知的server.port的设置去找到对应的配置metadata，里面会有对应的source_type，这里就是配置类，在里面会发现Tomcat是其中的内置类。



可以通过修改上下文和连接点的一些配置（但是真实在使用的过程中是使用配置文件去做的）

三、问题

内置tomcat的集群怎么搞?

微服务是无状态的，嵌入式的容器（比如通过一致性hash或者轮询、nginx来轮询去访问集群） load balance去做

而传统的J2EE是有状态的（网段之间的广播）

spring cloud不一定所有组件都好 dubbo是二进制的协议 grpc是文本protostuff序列化协议

spring boot嵌入式容器的Tomcat是包含在spring中一个bean中，而传统的是用tomcat去加载spring的context（通过事件监听）

ClassLoader
双亲委派的体现：
（1）classLoader抽象类去定义的时候里面有parent
（2）构造函数可以传入parent去初始化
（3）loadClass时 首先调用JNI的方法找到加载类、parent不为空时然后递归去调用loadClass方法、为空时则去找BootStrapClassLoader

为什么还会用Mybatis
Mybatis
是面向sql的，自由度更高
Hibernate JPA
是面向对象的 所以自由度没那么高