# 一个很详细的web.xml讲解

1、启动一个WEB项目的时候，WEB容器会去读取它的配置文件web.xml，读取<listener>和<context-param>两个结点。

2、紧急着，容器创建一个ServletContext（servlet上下文），这个web项目的所有部分都将共享这个上下文。

3、容器将<context-param>转换为键值对，并交给servletContext。

4、容器创建<listener>中的类实例，创建监听器。

## 一、Web.xml的作用?

作用是用来初始化配置信息的。

比如：Welcome页面、servlet、servlet-mapping、filter、listener、启动加载级别等。

## 二、Web.xml是每个J2EE工程所必需的吗？

当你的工程不需要上述配置时，就不需要web.xml来配置你的Application。

## 三、节点元素Load-on-startup

Load-on-startup 元素在web应用启动的时候指定了servlet被加载的顺序，它的值必须是一个整数。

如果它的值是一个负整数或是这个元素不存在，那么容器会在该servlet被调用的时候，加载这个servlet 。

如果值是正整数或零，容器在配置的时候就加载并初始化这个servlet，容器必须保证值小的先被加载。如果值相等，容器可以自动选择先加载谁。

在servlet的配置当中，<load-on-startup>5</load-on-startup>的含义是：

标记容器是否在启动的时候就加载这个servlet。

当值为0或者大于0时，表示容器在应用启动时就加载这个servlet；

当是一个负数时或者没有指定时，则指示容器在该servlet被选择时才加载。

正数的值越小，启动该servlet的优先级越高。

## 四、加载顺序

首先可以肯定的是，加载顺序与它们在 web.xml 文件中的先后顺序无关。即不会因为 filter 写在 listener 的前面而会先加载 filter。最终得出的结论是：ServletContext -> listener -> filter -> servlet

同时还存在着这样一种配置节：context-param，它用于向 ServletContext 提供键值对，即应用程序上下文信息。我们的 listener, filter 等在初始化时会用到这些上下文中的信息，那么 context-param 配置节是不是应该写在 listener 配置节前呢？实际上 context-param 配置节可写在任意位置，因此真正的加载顺序为：context-param -> listener -> filter -> servlet

对于某类配置节而言，与它们出现的顺序是有关的。以 filter 为例，web.xml 中当然可以定义多个 filter，与 filter 相关的一个配置节是 filter-mapping，这里一定要注意，对于拥有相同 filter-name 的 filter 和 filter-mapping 配置节而言，filter-mapping 必须出现在 filter 之后，否则当解析到 filter-mapping 时，它所对应的 filter-name 还未定义。web 容器启动时初始化每个 filter 时，是按照 filter 配置节出现的顺序来初始化的，当请求资源匹配多个 filter-mapping 时，filter 拦截资源是按照 filter-mapping 配置节出现的顺序来依次调用 doFilter() 方法的。

servlet 同 filter 类似，此处不再赘述。

由此，可以看出，web.xml 的加载顺序是：ServletContext -> context-param -> listener -> filter -> servlet ，而同个类型之间的实际程序调用的时候的顺序是根据对应的 mapping 的顺序进行调用的。

## 五、常用标签及其功能

## <context-param></context-param>

使用上下文初始化参数

1. *<!--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*上下文初始化参数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-->*
2. <context-param>
3. <param-name>webAppRootKey</param-name>
4. <param-value>business.root</param-value>
5. </context-param>
6. *<!-- spring config -->*
7. <context-param>
8. <param-name>contextConfigLocation</param-name>
9. <param-value>/WEB-INF/spring-configuration/\*.xml</param-value>
10. </context-param>

## ****5.1<context-param>解释：****

<context-param>元素含有一对参数名和参数值，用作应用的Servlet上下文初始化参数，参数名在整个Web应用中必须是惟一的，在web应用的整个生命周期中上下文初始化参数都存在，任意的Servlet和jsp都可以随时随地访问它。<param-name>子元素包含有参数名，而<param-value>子元素包含的是参数值。作为选择，可用<description>子元素来描述参数。

## 5.2 什么情况下使用，为什么使用<context-param>：

比如：定义一个管理员email地址用来从程序发送错误，或者与你整个应用程序有关的其他设置。使用自己定义的设置文件需要额外的代码和管理；直接在你的程序中使用硬编码（Hard-coding）参数值会给你之后修改程序带来麻烦，更困难的是，要根据不同的部署使用不同的设置；通过这种办法，可以让其他开发人员更容易找到相关的参数，因为它是一个用于设置这种参数的标准位置。

配置Spring，必须需要<listener>，而<context-param>可有可无，如果在web.xml中不写<context-param>配置信息，默认的路径是/WEB-INF/applicationontext.xml，在WEB-INF目录下创建的xml文件的名称必须是applicationContext.xml。如果是要自定义文件名可以在web.xml里加入contextConfigLocation这个context参数：在<param-value>

## ****6. <session-config></session-config>****

1. *<!-- Set timeout to 120 minutes -->*
2. <session-config>
3. <session-timeout>120</session-timeout>
4. </session-config>

<session-config> 用于设置容器的session参数，比如：<session-timeout> 用于指定http session的失效时间。默认时间设置在<jakarta>/conf/web.xml (30 minutes)。<session-timeout>用来指定默认的会话超时时间间隔，以分钟为单位。该元素值必须为整数。如果 session-timeout元素的值为零或负数，则表示会话将永远不会超时。

### 7. <listener></listener>

1. *<!--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*监听器配置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-->*
2. *<!-- Spring的log4j监听器 -->*
3. <listener>
4. <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
5. </listener>
6. <listener>
7. <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
8. </listener>
9. *<!-- 与CAS Single Sign Out Filter配合，注销登录信息 -->*
10. <listener>
11. <listener-class>com.yonyou.mcloud.cas.client.session.SingleSignOutHttpSessionListener</listener-class>
12. </listener>

#### 7.1 Listener介绍：

Servlet的监听器Listener，它是实现了javax.servlet.ServletContextListener 接口的服务器端程序，它也是随web应用的启动而启动，**只初始化一次**，随web应用的停止而销毁。**主要作用是： 做一些初始化的内容添加工作、设置一些基本的内容、比如一些参数或者是一些固定的对象等等。**

<listener>为web应用程序定义监听器，监听器用来监听各种事件，比如：application和session事件，所有的监听器按照相同的方式定义，功能取决去它们各自实现的接口，常用的Web事件接口有如下几个：  
(1). ServletContextListener：用于监听Web应用的启动和关闭；  
(2). ServletContextAttributeListener：用于监听ServletContext范围（application）内属性的改变；  
(3). ServletRequestListener：用于监听用户的请求；  
(4). ServletRequestAttributeListener：用于监听ServletRequest范围（request）内属性的改变；  
(5). HttpSessionListener：用于监听用户session的开始和结束；  
(6). HttpSessionAttributeListener：用于监听HttpSession范围（session）内属性的改变。

<listener>主要用于监听Web应用事件，其中有两个比较重要的WEB应用事件：**应用的启动和停止（**starting up or shutting down**）**和**Session的创建和失效（**created or destroyed**）**。应用启动事件发生在应用第一次被Servlet容器装载和启动的时候；停止事件发生在Web应用停止的时候。

Session创建事件发生在每次一个新的session创建的时候，类似地Session失效事件发生在每次一个Session失效的时候。为了使用这些Web应用事件做些有用的事情，我们必须创建和使用一些特殊的“监听类”。它们是实现了以下两个接口中任何一个接口的简单java类：javax.servlet.ServletContextListener或javax.servlet.http.HttpSessionListener，如果想让你的类监听应用的启动和停止事件，你就得实现ServletContextListener接口；想让你的类去监听Session的创建和失效事件，那你就得实现HttpSessionListener接口。

#### ****7.2 Listener配置：****

配置Listener只要向Web应用注册Listener实现类即可，无序配置参数之类的东西，因为Listener获取的是Web应用ServletContext（application）的配置参数。为Web应用配置Listener的两种方式：

(1). 使用@WebListener修饰Listener实现类即可。

(2). 在web.xml文档中使用<listener>进行配置。

我们选择web.xml这种配置方式，只有一个元素<listener-class>指定Listener的实现类，如下所示：

1. <listener>
2. <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
3. </listener>

这里的<listener>用于Spring的加载，Spring加载可以利用**ServletContextListener**实现，也可以采用**load-on-startup Servlet**实现，但是当<filter>需要用到bean时，但加载顺序是：先加载<filter>后加载<servlet>，则<filter>中初始化操作中的bean为null；所以，如果过滤器中要使用到bean，此时就可以根据加载顺序<listener> -> <filter> -> <servlet>，将spring的加载改成Listener的方式。

(1). 利用ServletContextListener实现：

1. <servlet>
2. <servlet-name>context</servlet-narne>
3. <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
4. <load-on-startup>1</load-on-startup>
5. </servlet>

(2).采用load-on-startup Servlet 实现：

1. <listener>
2. <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
3. </listener>

我们选择了第二种方式，在J2EE工程中web服务器启动的时候最先调用web.xml，上面这段配置的意思是加载spring的监听器，其中ContextLoaderListener的作用就是启动Web容器时，自动装配applicationContext.xml的配置信息，执行它所实现的方法。

## 8. <filter></filter>

<!--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*过滤器配置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-->  
<!-- 字符集过滤器 -->  
<**filter**>  
 <**filter-name**>CharacterEncodingFilter</**filter-name**>  
 <**filter-class**>org.springframework.web.filter.CharacterEncodingFilter</**filter-class**>  
 <**init-param**>  
 <**param-name**>encoding</**param-name**>  
 <**param-value**>UTF-8</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**param-name**>forceEncoding</**param-name**>  
 <**param-value**>true</**param-value**>  
 </**init-param**>  
</**filter**>  
 <!-- 单点登出过滤器 -->  
<**filter**>  
<**filter-name**>CAS Single Sign Out Filter</**filter-name**>  
<**filter-class**>com.yonyou.mcloud.cas.client.session.SingleSignOutFilter</**filter-class**>  
</**filter**>  
 <!-- 认证过滤器 -->  
<**filter**>  
<**filter-name**>CAS Authentication Filter</**filter-name**>  
<**filter-class**>com.yonyou.mcloud.cas.client.authentication.ExpandAuthenticationFilter</**filter-class**>  
<**init-param**>  
 <**param-name**>casServerLoginUrl</**param-name**>  
 <**param-value**>https://dev.yonyou.com:443/sso-server/login</**param-value**>  
</**init-param**>  
<**init-param**>  
 <!--这里的server是服务端的IP -->  
 <**param-name**>serverName</**param-name**>  
 <**param-value**>http://10.1.215.40:80</**param-value**>  
</**init-param**>  
</**filter**>  
 <!-- 验证ST/PT过滤器 -->  
<**filter**>  
 <**filter-name**>CAS Validation Filter</**filter-name**>  
 <**filter-class**>org.jasig.cas.client.validation.Cas20ProxyReceivingTicketValidationFilter</**filter-class**>  
 <**init-param**>  
 <**param-name**>casServerUrlPrefix</**param-name**>  
 <**param-value**>https://dev.yonyou.com:443/sso-server</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**param-name**>serverName</**param-name**>  
 <**param-value**>http://10.1.215.40:80</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**param-name**>proxyCallbackUrl</**param-name**>  
 <**param-value**>https://dev.yonyou.com:443/business/proxyCallback</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**param-name**>proxyReceptorUrl</**param-name**>  
 <**param-value**>/proxyCallback</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**param-name**>proxyGrantingTicketStorageClass</**param-name**>  
 <**param-value**>com.yonyou.mcloud.cas.client.proxy.MemcachedBackedProxyGrantingTicketStorageImpl</**param-value**>  
 </**init-param**>  
 <!-- 解决中文问题 -->  
 <**init-param**>  
 <**param-name**>encoding</**param-name**>  
 <**param-value**>UTF-8</**param-value**>  
 </**init-param**>  
</**filter**>  
<**filter**>  
 <**filter-name**>CAS HttpServletRequest Wrapper Filter</**filter-name**>  
 <**filter-class**>org.jasig.cas.client.util.HttpServletRequestWrapperFilter</**filter-class**>  
</**filter**>  
<**filter**>  
 <**filter-name**>CAS Assertion Thread Local Filter</**filter-name**>  
 <**filter-class**>org.jasig.cas.client.util.AssertionThreadLocalFilter</**filter-class**>  
</**filter**>  
<**filter**>  
 <**filter-name**>NoCache Filter</**filter-name**>  
 <**filter-class**>com.yonyou.mcloud.cas.client.authentication.NoCacheFilter</**filter-class**>  
</**filter**>  
 <!--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*映射关系配置\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-->  
<**filter-mapping**>  
 <**filter-name**>CharacterEncodingFilter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>NoCache Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS Single Sign Out Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS Validation Filter</**filter-name**>  
 <**url-pattern**>/proxyCallback</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS Authentication Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS Validation Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS HttpServletRequest Wrapper Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>  
<**filter-mapping**>  
 <**filter-name**>CAS Assertion Thread Local Filter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>

#### 8.1 Filter介绍：

Filter可认为是Servle的一种“加强版”，主要用于对用户请求request进行预处理，也可以对Response进行后处理，是个典型的处理链。使用Filter的完整流程是：Filter对用户请求进行预处理，接着将请求HttpServletRequest交给Servlet进行处理并生成响应，最后Filter再对服务器响应HttpServletResponse进行后处理。Filter与Servlet具有完全相同的生命周期，且Filter也可以通过<init-param>来配置初始化参数，获取Filter的初始化参数则使用FilterConfig的getInitParameter()。

换种说法，Servlet里有request和response两个对象，Filter能够在一个request到达Servlet之前预处理request，也可以在离开Servlet时处理response，Filter其实是一个Servlet链。以下是Filter的一些常见应用场合，

(1)认证Filter

(2)日志和审核Filter

(3)图片转换Filter

(4)数据压缩Filter

(5)密码Filter

(6)令牌Filter

(7)触发资源访问事件的Filter

(8)XSLT Filter

(9)媒体类型链Filter

Filter可负责拦截多个请求或响应；一个请求或响应也可被多个Filter拦截。创建一个Filter只需两步:  
(1) 创建Filter处理类  
(2) Web.xml文件中配置Filter  
Filter必须实现javax.servlet.Filter接口，在该接口中定义了三个方法：  
(1) void init(FilterConfig config)：用于完成Filter的初始化。FilteConfig用于访问Filter的配置信息。  
(2) void destroy()：用于Filter销毁前，完成某些资源的回收。  
(3) void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)：实现过滤功能的核心方法，该方法就是对每个请求及响应增加额外的处理。该方法实现对用户请求request进行预处理，也可以实现对服务器响应response进行后处理---它们的分界线为是否调用了chain.doFilter(request，response)，执行该方法之前，即对用户请求request进行预处理，执行该方法之后，即对服务器响应response进行后处理。

8.2 Filter配置：

Filter可认为是Servlet的“增强版”，因此Filter配置与Servlet的配置非常相似，需要配置两部分：配置Filter名称和Filter拦截器URL模式。区别在于Servlet通常只配置一个URL，而Filter可以同时配置多个请求的URL。配置Filter有两种方式：  
(1). 在Filter类中通过Annotation进行配置。

(2). 在web.xml文件中通过配置文件进行配置。  
我们使用的是web.xml这种配置方式，下面重点介绍<filter>内包含的一些元素。  
<filter>用于指定Web容器中的过滤器，可包含<filter-name>、<filter-class>、<init-param>、<icon>、<display-name>、<description>。

(1).<filter-name>用来定义过滤器的名称，该名称在整个程序中都必须唯一。

(2).<filter-class>元素指定过滤器类的完全限定的名称，即Filter的实现类。

(3).<init-param>为Filter配置参数，与<context-param>具有相同的元素描述符<param-name>和<param-value>。

(4).<filter-mapping>元素用来声明Web应用中的过滤器映射，过滤器被映射到一个servlet或一个URL 模式。这个过滤器的<filter>和<filter-mapping>必须具有相同的<filter-name>，指定该Filter所拦截的URL。过滤是按照部署描述符的<filter-mapping>出现的顺序执行的。

## ****9.1 Servlet介绍：****

Servlet通常称为服务器端小程序，是运行在服务器端的程序，用于处理及响应客户的请求。Servlet是个特殊的java类，继承于**HttpServlet**。客户端通常只有GET和POST两种请求方式，Servlet为了响应则两种请求，必须重写doGet()和doPost()方法。大部分时候，Servlet对于所有的请求响应都是完全一样的，此时只需要重写service()方法即可响应客户端的所有请求。

另外，HttpServlet有两个方法：

(1). init(ServletConfig config)：创建Servlet实例时，调用该方法的初始化Servlet资源。

(2). destroy()：销毁Servlet实例时，自动调用该方法的回收资源。

通常无需重写init()和destroy()两个方法，除非需要在初始化Servlet时，完成某些资源初始化的方法，才考虑重写init()方法，如果重写了init()方法，应在重写该方法的第一行调用super.init(config)，该方法将调用HttpServlet的init()方法。如果需要在销毁Servlet之前，先完成某些资源的回收，比如关闭数据库连接，才需要重写destory方法()。

**Servlet的生命周期：**

创建Servlet实例有两个时机：

(1). 客户端第一次请求某个Servlet时，系统创建该Servlet的实例，大部分Servlet都是这种Servlet。

(2). Web应用启动时立即创建Servlet实例，即load-on-start Servlet。

**每个Servlet的运行都遵循如下生命周期：**

(1). 创建Servlet实例。

(2). Web容器调用Servlet的init()方法，对Servlet进行初始化。

(3). Servlet初始化后，将一直存在于容器中，用于响应客户端请求，如果客户端发送GET请求，容器调用Servlet的doGet()方法处理并响应请求；如果客户端发送POST请求，容器调用Servlet的doPost()方法处理并响应请求。或者统一使用service()方法处理来响应用户请求。

(4). Web容器决定销毁Servlet时，先调用Servlet的destory()方法，通常在关闭Web应用时销毁Servlet实例。

## 9.2 Servlet配置：

为了让Servlet能响应用户请求，还必须将Servlet配置在web应用中，配置Servlet需要修改web.xml文件。从Servlet3.0开始，配置Servlet有两种方式：

(1). 在Servlet类中使用@WebServlet Annotation进行配置。

(2). 在web.xml文件中进行配置。

我们用web.xml文件来配置Servlet，需要配置<servlet>和<servlet-mapping>。<servlet>用来声明一个Servlet。<icon>、<display-name>和<description>元素的用法和<filter>的用法相同。<init-param>元素与<context-param>元素具有相同的元素描述符，可以使用<init-param>子元素将初始化参数名和参数值传递给Servlet，访问Servlet配置参数通过ServletConfig对象来完成，ServletConfig提供如下方法：

java.lang.String.getInitParameter(java.lang.String name)：用于获取初始化参数

ServletConfig获取配置参数的方法和ServletContext获取配置参数的方法完全一样，只是ServletConfig是取得当前Servlet的配置参数，而ServletContext是获取整个Web应用的配置参数。

**(1). <description>、<display-name>和<icon>**

1). <description>：为Servlet指定一个文本描述。

2). <display-name>：为Servlet提供一个简短的名字被某些工具显示。

3). <icon>：为Servlet指定一个图标，在图形管理工具中表示该Servlet。

**(2). <servlet-name>、<servlet-class>和<jsp-file>元素**

<servlet>必须含有<servlet-name>和<servlet-class>，或者<servlet-name>和<jsp-file>。 描述如下：

1). <servlet-name>用来定义servlet的名称，该名称在整个应用中必须是惟一的。

2). <servlet-class>用来指定servlet的完全限定的名称。

3). <jsp-file>用来指定应用中JSP文件的完整路径。这个完整路径必须由/开始。

**(3). <load-on-startup>**

如果load-on-startup元素存在，而且也指定了jsp-file元素，则JSP文件会被重新编译成Servlet，同时产生的Servlet也被载入内存。<load-on-startup>的内容可以为空，或者是一个整数。这个值表示由Web容器载入内存的顺序。

举个例子：如果有两个Servlet元素都含有<load-on-startup>子元素，则<load-on-startup>子元素值较小的Servlet将先被加载。如果<load-on-startup>子元素值为空或负值，则由Web容器决定什么时候加载Servlet。如果两个Servlet的<load-on-startup>子元素值相同，则由Web容器决定先加载哪一个Servlet。<load-on-startup>1</load-on-startup>表示启动容器时，初始化Servlet。

**(4). <servlet-mapping>**

<servlet-mapping>含有<servlet-name>和<url-pattern>

1). <servlet-name>：Servlet的名字，唯一性和一致性，与<servlet>元素中声明的名字一致。

2). <url-pattern>：指定相对于Servlet的URL的路径。该路径相对于web应用程序上下文的根路径。<servlet-mapping>将URL模式映射到某个Servlet，即该Servlet处理的URL。

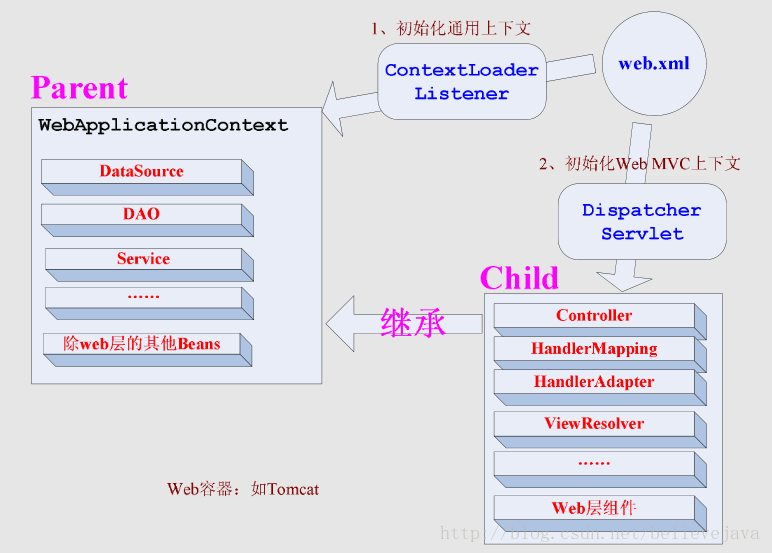
**(5). 加载Servlet的过程**

容器的Context对象对请求路径(URL)做出处理，去掉请求URL的上下文路径后，按路径映射规则和Servlet映射路径（<url- pattern>）做匹配，如果匹配成功，则调用这个Servlet处理请求。

**9.3DispatcherServlet在web.xml中的配置：**

1. *<!-- Spring view分发器 对所有的请求都由business对应的类来控制转发 -->*
2. <servlet>
3. <servlet-name>business</servlet-name>
4. <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
5. <init-param>
6. <param-name>publishContext</param-name>
7. <param-value>false</param-value>
8. </init-param>
9. <load-on-startup>1</load-on-startup>
10. </servlet>

#### ****ContextLoaderListener和DispatcherServlet初始化上下文关系和区别：****



## spring的核心配置文件的名字 叫做 applicationContext.xml，后期也可以通过配置文件中的配置修改名称，在web.xml中进行如下配置：

web.xml

1. <context-param>
2. <param-name>contextConfigLocation</param-name>
3. <param-value>classpath:spring/applicationContext\*.xml</param-value>
4. </context-param>

## 2：核心配置文件中关于dao层的配置。

(1)：首先准备db.properties 配置文件，最简单的配置如下。

db.properties

1. jdbc.driver=com.mysql.jdbc.Driver
2. jdbc.url=jdbc:mysql://127.0.0.1:3306/demo?characterEncoding=utf-8
3. jdbc.username=root
4. jdbc.password=123456

(2)：然后加载在核心配置文件中加载数据库文件.

<context:property-placeholder location="classpath:resource/db.properties" />

(3)：配置数据库连接池，配置类可以用BasicDatasource，也可以用阿里巴巴的配置核心类 DruidDataSource。

spring-mybatis.xml

1. <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
2. destroy-method="close">
3. <property name="url" value="${jdbc.url}" />
4. <property name="username" value="${jdbc.username}" />
5. <property name="password" value="${jdbc.password}" />
6. <property name="driverClassName" value="${jdbc.driver}" />
7. <property name="maxActive" value="10" />
8. <property name="minIdle" value="5" />
9. </bean>

后期需要可以在其中添加多个属性配置。

(4)：spring和hibernate，和mybatis的整合主要是整合sessionFactory.

     和hibernate的一个整合。

1. <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
2. *<!-- 与dataSource -->*
3. <property name="dataSource">
4. <ref bean="dataSource"/>
5. </property>
6. </bean>

和mybatis的一个整合.

1. *<!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的 -->*
2. <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
3. *<!-- 数据库连接池 -->*
4. <property name="dataSource" ref="dataSource" />
5. *<!-- 加载mybatis的全局配置文件 -->*
6. <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml" />
7. </bean>

(5)：配置文件中关于事务的配置。

1. <!-- 事务管理器 -->
2. <bean id="transactionManager"
3. class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4. *<!-- 数据源 -->*
5. <property name="dataSource" ref="dataSource" />
6. </bean>

配置通知。

1. *<!-- 通知 -->*
2. <tx:advice id="txAdvice" transaction-manager="transactionManager">
3. <tx:attributes>
4. *<!-- 传播行为 -->*
5. <tx:method name="save\*" propagation="REQUIRED" />
6. <tx:method name="insert\*" propagation="REQUIRED" />
7. <tx:method name="add\*" propagation="REQUIRED" />
8. <tx:method name="create\*" propagation="REQUIRED" />
9. <tx:method name="delete\*" propagation="REQUIRED" />
10. <tx:method name="update\*" propagation="REQUIRED" />
11. <tx:method name="find\*" propagation="SUPPORTS" read-only="true" />
12. <tx:method name="select\*" propagation="SUPPORTS" read-only="true" />
13. <tx:method name="get\*" propagation="SUPPORTS" read-only="true" />
14. </tx:attributes>
15. </tx:advice>

关于切面的配置。

1. *<!-- 切面 -->*
2. <aop:config>
3. <aop:advisor advice-ref="txAdvice"
4. pointcut="execution(\* com.store.service.\*.\*(..))" />
5. </aop:config>

关于配置文件中的service层的配置。     扫描包下面所有的service层。

<context:component-scan base-package="com.xiaoao.service"/>

关于注解注入的配置

<mvc:annotation-driven />