

# 前言

## 为什么写这本书

现在测试人员越来越“不务正业了”，都在想着去写代码，去做自动化，去做高大上的工具。我始终认为测试人员应该以业务为主，从业务的角度去设计好测试用例，以保证好产品的质量为第一宗旨。但代码也好，工具也罢，只能从效率上去起作用，对产品的质量并没有显著的效果。但是这是个浮躁的时代，只有会自动化，会代码，才能去更好的包装自己，从而获取高薪。为了顺应这种浮躁的现象，所以我也推出了这本书，本书主要是讲开发测试平台方面的书，希望大家能从此书中获取到更多的“利器”，从而去获取到更高的薪水或更好的发展平台。

## 开发测试

在我所经历的测试时代，我认为应该分为三个阶段：功能测试阶段，自动化测试阶段，测试平台开发阶段。在功能测试阶段，在公交车上经常能看到“测试缺口 20 万”这样的广告，那时候外包公司多，全民手工测试，测试流程完善，这也就是大家所说的功能测试或手工测试。在互联网兴起之后，敏捷开发的概念也兴起了，为了应付版本的快速发布，UI 自动化测试就开始爆了，测试人员这时候也开始兴奋了，原来测试人员也可以写代码的！一个新的 title 也出现了：测试开发。很明显的就是在招聘简章中都喜欢招测试开发人员了，所以，测试开发比功能测试人员的工资要高那么一点点。在移动互联网兴起后，接口测试也越来越重要了，做接口测试对工具或代码的要求好像要更高一点，所以，测试开发就越来越吃香了，为了突显测试人员的代码能力，各种轮子开始造的飞起了，各个公司都开始做自动化测试平台了，这点同样的从招聘简章中可以看出，做自动化测试平台就需要有开发的技术，测试的思想了，比以前的测试开发又高端了那么一点点了，我个人称之为开发测试。

## 针对人群及目标

本书主要是针对有一定的 java 基础,同时又想在开发测试领域有一定的发展的人,或者说想顺应新时代的测试领域的人。看完本书,希望大家能够对开发技术有一定的认识,在与开发的“交锋”中不再处于下风,要让大家知道接口原来是这样写的,web 开发原来是这样做的,同时也认识到原来开发也并不难,只是自己以前没有太多的关注。最重要的,是希望大家能树立起信心,不再迷茫,早日进入自己理想的公司或者更大的平台去发展。

## 作者简介

本人没有 BAT 经历,这点很遗憾,混迹于测试界多年,但一直没有放弃一颗写代码的心,一直有记录自己的博客:

<http://www.cnblogs.com/zhangfei/>,

也有在 GITHUB 上开源自己写的一些工具:

<https://github.com/zhangfei19841004/zson>,

前段时间也写了一本测试方面的书<<测试开发 Java21 天>>:

<http://yuedu.baidu.com/ebook/f6dbb2a2f01dc281e53af0f3>

目前在一个小互联网公司做开发,希望能在该书中反应出自己在学习开发技术中的一些路线。

# 环境安装

## 知识体系

- 本书会涉及到的工具有：

工具	描述
jdk	1.7 版本
eclipse	支持 jdk1.7 版本的即可
maven	最新版
tomcat	7.0.X 版本

## Java 环境安装及配置

- 请参考<<测试开发 Java21 天>>第一章内容

## Maven 安装及配置

- Maven 的作用

Maven 是一个项目管理的 java 工具。java 发展到今天，各种类库丰富多彩，我们在一个项目中可能用到的 jar 包很多，需要一个个的下载到本地，然后再推到版本控制服务器上，这样大家一起从版本控制服务器上下载 jar 包并使用，如果 jar 包升级了，那得再下载并推到版本控制服务器，大家再一起更新，相当的麻烦，如果 jar 包之间有依赖，那更加的痛不欲生。为了让开发人员摆脱这些"杂事"，更加专注于代码本身，Maven 就应运而生了，所以 Maven 的第一个重要功能就是 jar 包管理，它可以帮我们自动的下载 jar 包，自动的关联上工程，只需要在一个 pom.xml 文件中配置好要下载 jar 包的包名就行了，那 Maven 会去哪里下载这些配置好的 jar 包？这就是我们所说的 Maven 中央库，Maven 配置里有一个默认链接，指向了这个 Maven 中央库，当然 Maven 中央库有很多个，想换时只需要更改这个链接即可。Maven 中央库提供了下载功能，同样的，也可以把我们写好的 jar 包推向 Maven 中央库，以方便向别人共享使用。于是一个团队在协同做项目时，只需要把

pom.xml 推到版本控制服务器, 大家一起维护这个 pom.xml, 就解决了这些个环境的难题了。Maven 还可以帮助我们编译打包 java 代码, 这也是 Maven 的另一个非常重要的功能。

- Maven 下载地址  
<http://maven.apache.org/download.cgi>  
下载完成后, 解压并配置环境变量即可。

## Tomcat 安装

- 下载地址  
<http://tomcat.apache.org/download-70.cgi>  
下载完成后, 解压即可。

## Eclipse 集成 Tomcat

1. 下载插件: <http://www.eclipse-tomcat.com/tomcatPlugin.html#A3>  
根据我们所装的 Eclipse 及 Tomcat 来选择插件版本, 建议用 3.3 版本的即可。
2. 下载完后, 并解压:



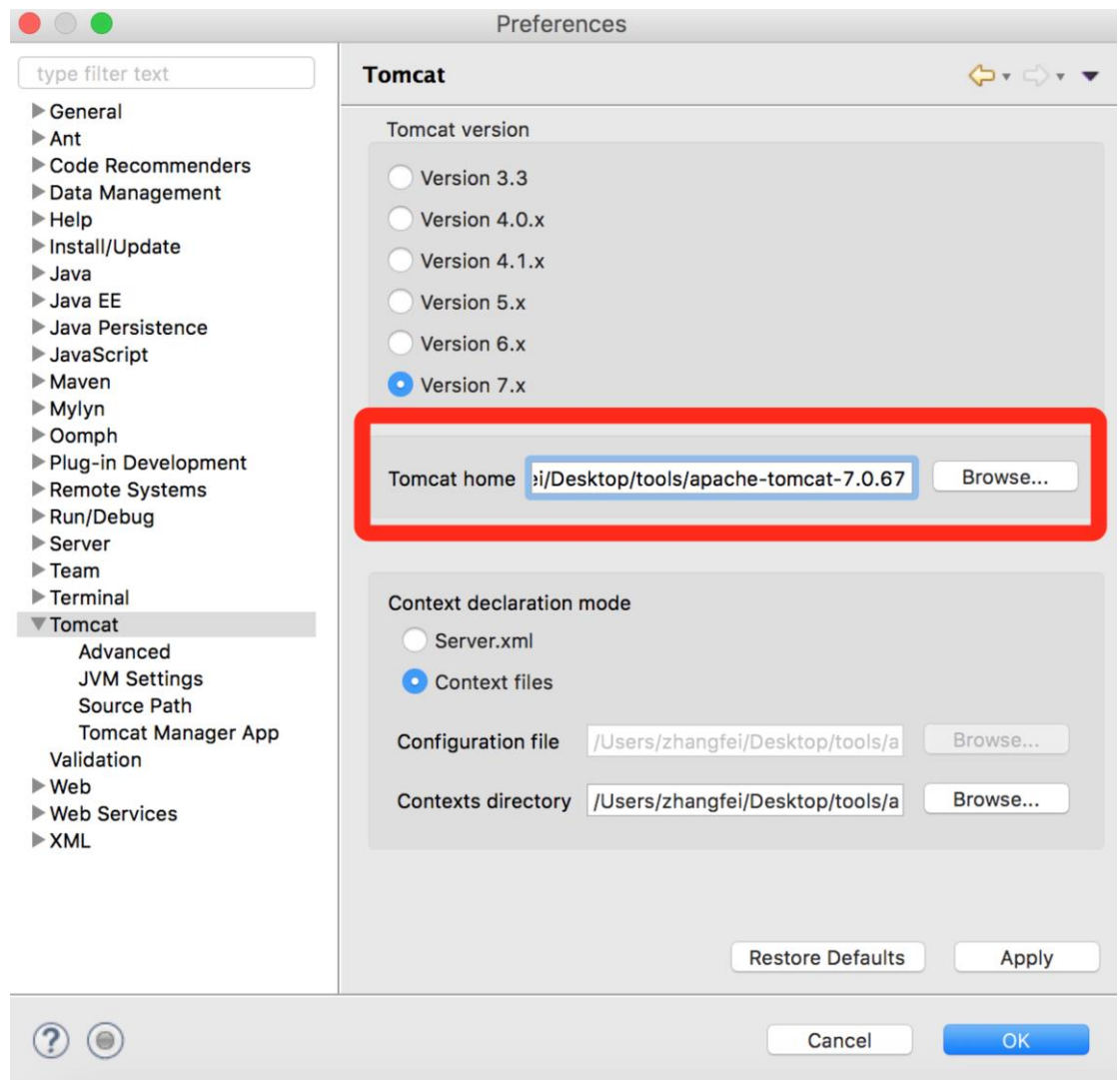
com.sysdeo.eclipse  
.tomcat\_3.3.0

3. 将 com.sysdeo.eclipse.tomcat\_3.3.0 文件拷贝到 eclipse 根目录下的 plugins 目录中。重启 Eclipse, 工具栏里出现图标



证明插件已经安装成功。该插件主要是开启或停止 Tomcat。

4. 插件安装成功后, 还需要将插件与已安装的 Tomcat 关联起来, 以便开启或停止已安装的 Tomcat。如下图所示:

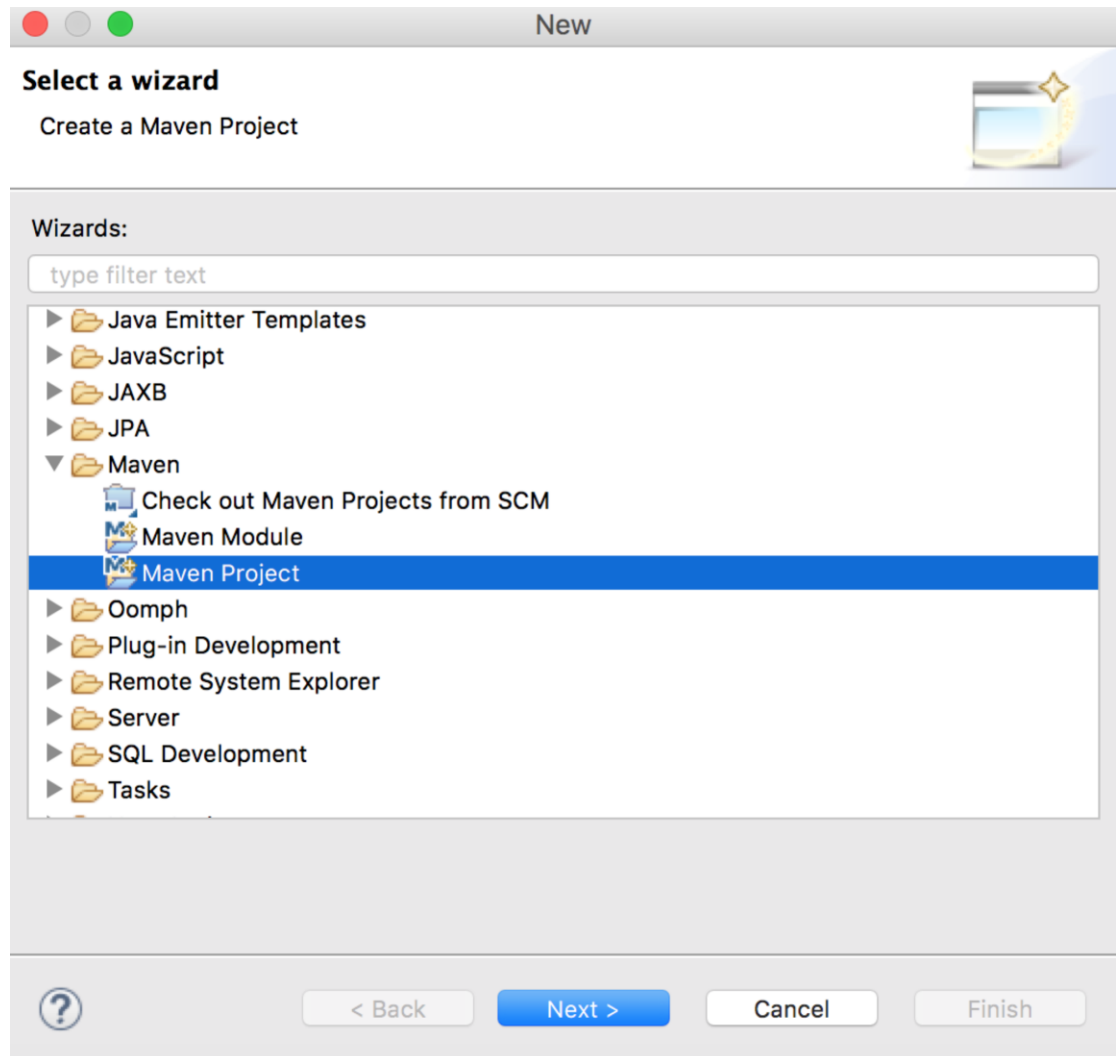


关联后，大家可以在 Eclipse 中试着去启动一下 Tomcat，在控制台中会有启动信息。

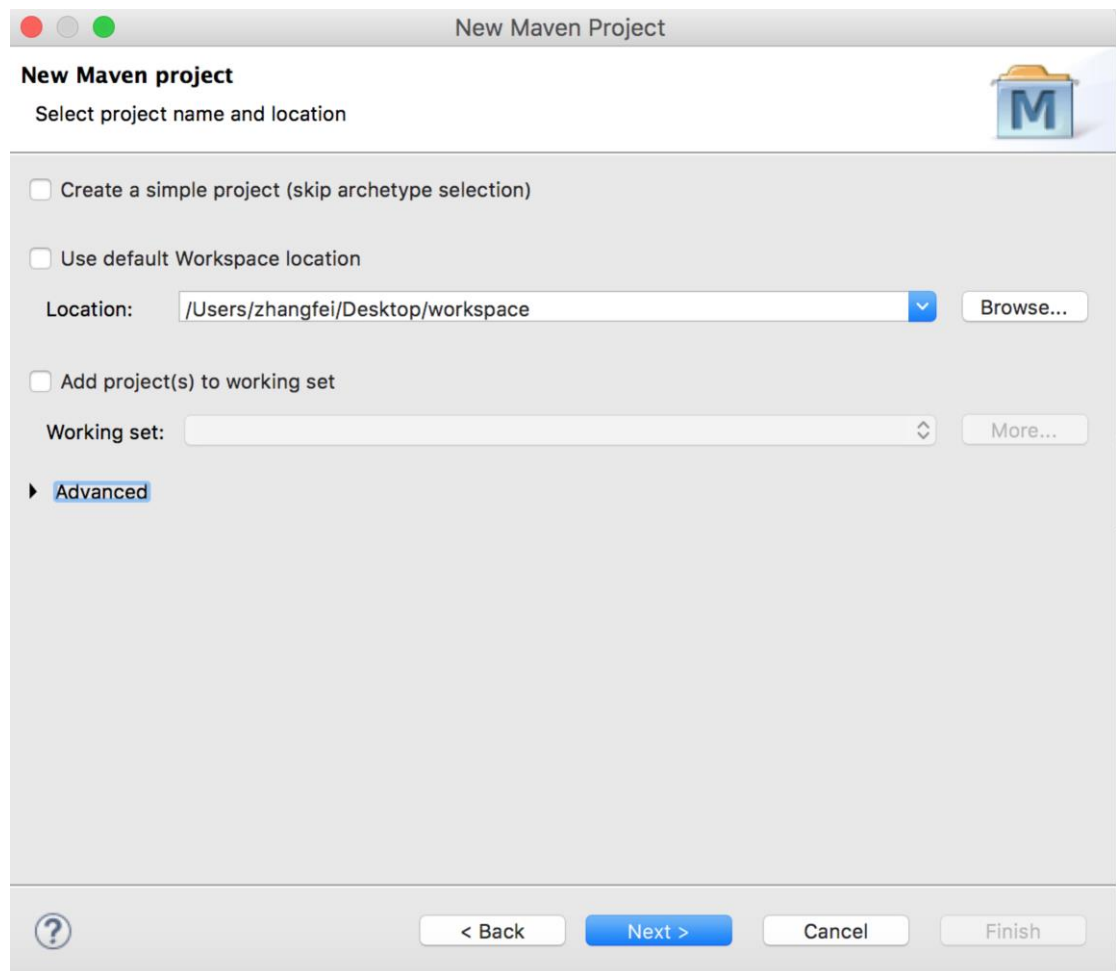
# 初建工程

## 创建 Maven 工程

1. 在 Eclipse 中, New->Maven->Maven Project



2. Next 后, 通过”Browse...”按钮选择工程目录:



3. Next 后, 再 Next, 进入如下页面:

New Maven Project

New Maven project

✗ Enter a group id for the artifact.

Group Id:

Artifact Id:

Version: 0.0.1-SNAPSHOT

Package:

Properties available from archetype:

Name	Value

Add...

Remove

Advanced

< Back Next > Cancel Finish

说明:

- Group Id, 组织 Id, 比如我们常看到或听到的 org.apache, 也可以理解为一个工程的姓!
- Artifact Id, 工程名称。在一个中央仓库, 理论上来说, Group Id 与 Artifact Id 的组合是唯一的, 即一个工程的姓名是唯一的, 这样方便我们去查找与下载所需要的 jar 包, 如果有两个 jar 包的姓与名是完全一样的, 那我们在下载的时候, 就不知所然了, 不知道下哪一个了, 所以我们在新建工程时, 要尽可能的确保我们的 Group Id 与 Artifact Id 是唯一的, 以免在我们把 jar 包推到中央仓库时出现问题。
- Version, 版本名称, 因为工程是不断的迭代的, 每一个迭代版本, 我们都要赋予一个版本号。
- Package, 包名, 工程的 Group Id 与 Artifact Id 的组合, 即一个工程的姓名。该栏会自动的填充, 不需要我们手动输入。

填完后:



New Maven Project

Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

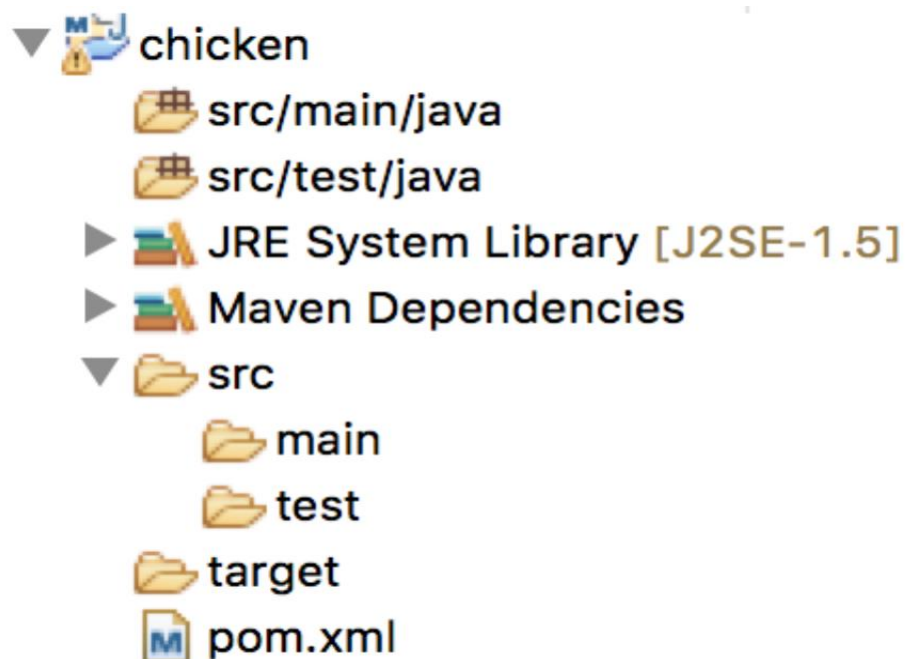
Properties available from archetype:

Name	Value

Advanced

< Back Next > Cancel Finish

4. 点击 Finish。工程创建完毕。



## 认识 pom.xml

因为工程所有的 jar 包管理与工程编译等都是通过配置 pom.xml 来完成的, 所以理解 pom.xml 就变得尤为重要了, 在 Maven 工程创建完成后, 会自动的帮我们生成一个 pom.xml 文件, 且里面的内容如下:

```
chicken/pom.xml 23
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <groupId>com.zf</groupId>
6   <artifactId>chicken</artifactId>
7   <version>0.0.1-SNAPSHOT</version>
8   <packaging>jar</packaging>
9   <name>chicken</name>
10
11   <properties>
12     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13   </properties>
14
15 </project>
16
```

说明:

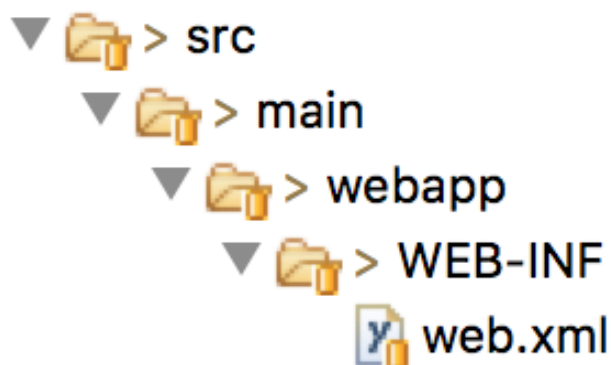
- 根节点 project 中属性 xmlns, xmlns:xsi, xsi:schemaLocation 共同定义了 pom.xml 文件的命名空间, 所谓的命名空间就是给这个 xml 文件规定了节点的名称和属性, 即在这个 pom.xml 文件中的所有节点, 必须是在这个命名空间中的, 否则就会报错, 大家可以试着在 pom.xml 文件中随便加入一个节点, 看是否会报错。
- modelVersion, pom 版本。为了确保稳定的使用, 这个元素是强制性的, 且固定为 4.0.0, 除非 maven 开发者升级模板, 否则不需要修改。
- groupId, 工程的姓。
- artifactId, 工程的名。
- version, 工程版本号
- packaging, 工程打包编译的类型, 比如打成 jar 包, 或许 web 工程的 war 包。
- name, 工程的描述名称, 可要可不要的节点。
- properties, 定义变量。可定义自定义变量或初始化 Maven 内置变量。具体用法后面再介绍。
- project.build.sourceEncoding, 这是 Maven 的一个内置变量, 这里相当于给这个变量一个初始值。该变量是定义工程编译时的编码格式。

## pom.xml 的其它节点

```
chicken/pom.xml 23
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <groupId>com.zf</groupId>
7   <artifactId>chicken</artifactId>
8   <version>1.0</version>
9   <name>chicken</name>
10  <description>测试项目</description>
11  <packaging>war</packaging>
12
13  <properties>
14    <java-version>1.7</java-version>
15    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16    <webapp-folder>src/main/webapp</webapp-folder>
17    <springframework-version>4.2.0.RELEASE</springframework-version>
18    <slf4j-version>1.7.5</slf4j-version>
19  </properties>
20
21  <build>
22    <finalName>${project.artifactId}</finalName>
23  </build>
24
25 </project>
```

说明:

- description, 描述, 与 name 一样, 可要可不要的节点。
- java-version, 自定义变量。定义 JDK 版本
- webapp-folder, Maven 内置变量, 定义项目 webapp 路径。webapp 是指 web application, 是整个 web 工程的根目录, 也用来存放与 web 相关的一些文件, 比如 js, jsp, web.xml 等。



- springframework-version, 定义 spring 的版本。
- slf4j-version, 定义 log 的版本。
- build, 编译工程。
- finalName, 指编译后的 war 包名称。\${} 表示变量的使用,

`project.artifactId` 表示根节点 `project` 下的 `artifactId` 节点的值, 即编译后的 `war` 包名称为 `chicken.war`。根节点 `project` 下的其它子节点的值也可以这样获取及引用。

## pom.xml 的 jar 包配置

```
<dependencies>
  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${springframework-version}</version>
  </dependency>
  <!-- web相关 -->
  <dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.servlet</artifactId>
    <version>3.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.2</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
  <!-- log -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
  </dependency>
  <!-- fastjson -->
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.12</version>
  </dependency>
  <!-- 其它jar包 -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.1</version>
  </dependency>
</dependencies>
```

将以上的内容加至 project 根节点下，即加入并关联了以上的 jar 包。



















节点说明:

- dependencies, 依赖, 表示工程依赖的 jar 包。
- dependency, 依赖的某一个 jar 包。
- groupId, 依赖的 jar 包的姓。
- artifactId, 依赖的 jar 包的名。
- version, 依赖的 jar 包的版本号。
- 上面说过, 通过 groupId 与 artifactId 可以唯一确定一个 jar 包, 加了一个 dependency 的配置后, Maven 会自动的去中央库里下载 dependency 的 jar 包, 并关联起来。换句话说, 如果工程需要一个 jar 包, 加上一个 dependency 节点, 并在 dependency 节点下加上 groupId, artifactId 与 version 就可以了。

Jar 包说明:

- spring-webmac, 使用 spring 框架所需要的 jar 包, 当前只需要加入这一个 jar 包即可, 其它的目前阶段不需要, 后面需要时再加。里面的 version 用到了自定义变量: `${springframework-version}`, 这样的好处是如果有很多个 spring 的 jar 包, 在统一维护升级时, 只需要更改自定义变量的值即可。
- javax.servlet, jsp-api, jstl, 分别是 servlet 相关的 jar 包, jsp 相关的 jar 包, EL 表达式的 jar 包。
- slf4j-log4j12, fastjson, commons-lang3, 分别是 log 相关的 jar 包, json 相关的 jar 包, 常用方法相关的 jar 包。

将 jar 包给配置好后, 在工程结构中会多出一个 Maven Dependencies, 展开后可以看到所有的 jar 包都已下载好并关联到 Eclipse 中去了:

- ▼  Maven Dependencies
- ▶  spring-webmvc-4.2.0.RELEASE.jar -
  - ▶  spring-beans-4.2.0.RELEASE.jar - /l
  - ▶  spring-context-4.2.0.RELEASE.jar -
  - ▶  spring-aop-4.2.0.RELEASE.jar - /Us
  - ▶  aopalliance-1.0.jar - /Users/zhangfe
  - ▶  spring-core-4.2.0.RELEASE.jar - /Us
  - ▶  commons-logging-1.2.jar - /Users/z
  - ▶  spring-expression-4.2.0.RELEASE.ja
  - ▶  spring-web-4.2.0.RELEASE.jar - /Us
  - ▶  javax.servlet-3.0.jar - /Users/zhangf
  - ▶  jsp-api-2.2.jar - /Users/zhangfei/.m
  - ▶  jstl-1.2.jar - /Users/zhangfei/.m2/re
  - ▶  slf4j-log4j12-1.7.5.jar - /Users/zha
  - ▶  slf4j-api-1.7.5.jar - /Users/zhangfei
  - ▶  log4j-1.2.17.jar - /Users/zhangfei/.r
  - ▶  fastjson-1.2.12.jar - /Users/zhangfe
  - ▶  commons-lang3-3.1.jar - /Users/zh

# web.xml 配置

## web.xml 作用

在 web 工程里, web.xml 是非常重要的一个配置文件, 在 web 工程被启动时, 会事先解析 web.xml 并根据 web.xml 文件里的配置进行工程的一些初始化工作。于是 web.xml 的作用就是用来配置静态页面, 配置过滤器, 配置监听器, 配置 servlet 等等, 配置完后, 在启动时, 这些在 web.xml 中配置好的类或页面都会被实例化或初始化。



## web.xml 示例

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
version="2.5">
<display-name>test</display-name>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
</param-value>
</context-param>
<filter>
  <filter-name>SpringEncodingFilter</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>
<filter-mapping>
  <filter-name>SpringEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<listener>
  <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>
</listener>
<servlet>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value></param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
</web-app>
```

## web.xml 说明

- display-name, 显示名称, 无实际意义, 可填可不填。
- context-param, 全局变量, param-value 的值可以被整个 web 工程使用, 但这里只是定义, param-value 并没有被初始化。
- filter, filter-mapping, 过滤器的配置, filter 与 filter-mapping 是成对出现的, filter 里配置过滤类, filter-mapping 里配置要过滤的链接。示例中的过

滤波器的意思是在 request 与 response 时数据的编码格式为 UTF-8。

- listener，监听。是对整个 web 工程在运行期间的监听。示例中的 ContextLoaderListener 监听器是必须的，因为它会初始化 context-param，而 IntrospectorCleanupListener 是防止内存泄漏，这个监听不是必须的。
- servlet，servlet-mapping，配置一个 servlet，也可以理解为配置一个要实例化的类。servlet 与 servlet-mapping 是成对出现的，servlet 里面配置要实例化的类，servlet-mapping 里面配置作用于哪个链接上。对于用 spring，这个 servlet 是必须配置的，且只需要配置这一个 servlet 就可以了，该 servlet 表示所有的请求都会通过 DispatcherServlet 这个类来进行处理。
- session-config，session 有效时间，这里设置的是 30 分钟。

特别说明：在没有对 web 工程有一个全面了解之前，web.xml 还是采用上面的示例即可。

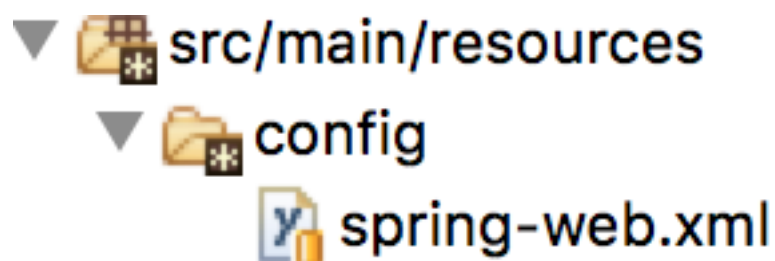
## web.xml 加载说明

web.xml 中有全局变量，有过滤，有监听，有 servlet，那他们的加载顺序是什么样的呢？毫无疑问，context-param 应该被最先加载，因为是全局变量嘛，context-param 加载后，param-value 的值应该要被初始化且使用起来，于是 ContextLoaderListener 监听就该起作用了，监听起来后，对于请求就要进行过滤了，哪些请求该被如何处理，filter 就是干这个的，所有的这些完了后，请求就该进入到 servlet 里进行相应的逻辑处理了。综上所述，web.xml 的加载顺序是：context-param>listener>filter>servlet。

# Spring 之 web 相关配置

## 前言

在新建工程时，我们会看到有个 `src/main/resources` 的目录结构，这个目录约定是用来专门存放配置文件的，也称为资源文件。`spring` 的配置文件当然也会被当成资源文件给存放到 `src/main/resources` 目录下面。为了使 `web` 项目中的众多资源文件方便管理，我们把资源文件也进行分类，`spring` 的配置文件全部放在一个 `config` 目录下面：



## 配置文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <bean id="fastJsonConfig" class="com.alibaba.fastjson.support.config.FastJsonConfig">
        <!-- Default charset -->
        <property name="charset" value="UTF-8"/>
        <!-- Default dateFormat -->
        <property name="dateFormat" value="yyyy-MM-dd HH:mm:ss"/>
        <!-- Feature -->
        <property name="features">
            <list>
                <value>SupportArrayToBean</value>
                <value>UseBigDecimal</value>
            </list>
        </property>
    </bean>
    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="true">
            <!-- 将StringHttpMessageConverter的默认编码设为UTF-8 -->
            <bean class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter4">
                <property name="fastJsonConfig" ref="fastJsonConfig"/>
                <property name="supportedMediaTypes">
                    <list>
                        <value>application/json;charset=UTF-8</value>
                        <value>text/html;charset=UTF-8</value>
                    </list>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>
</beans>
```

## 配置文件说明

可能我们会经常听到 java bean，实际上就是指的一个 java 实例化对象，上图中的<bean>标签就是指去实例化一个类对象。

上面示例中的配置很简单，就是配置了一个 fastjson，主要作用是 response 时的数据序列化。先按照上面的示例进行配置即可。

## 加载配置文件

```
<servlet>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:config/spring-web.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

将 web.xml 中的 DispatcherServlet 改为如上的写法。这样在 DispatcherServlet 进行初始化的时候，就会加载 spring-web.xml 文件并对 spring-web.xml 进行初始化。

# 第一个页面输出

## JSP 说明

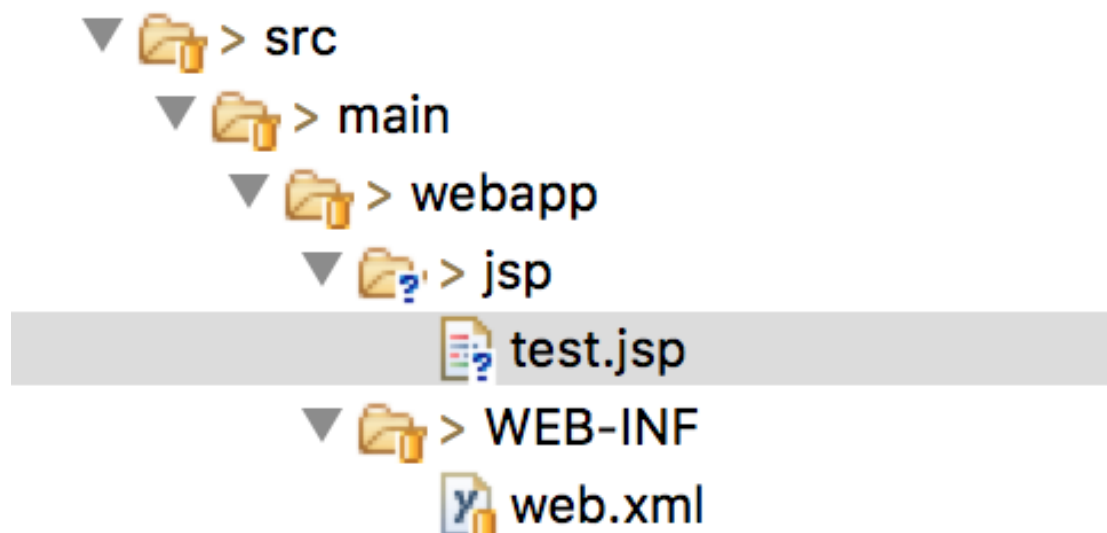
环境配置与 web 工程的相关配置已经讲了五章了,还没有涉及到真正的 web 开发技术,从这一章开始,将会逐渐的由浅入深的讲解 web 开发。首先就让我们了解一下什么是 jsp。

JSP (全称 Java Server Pages) 是由 Sun Microsystems 公司倡导和许多公司参与共同创建的一种使软件开发者可以响应客户端请求,而动态生成 HTML、XML 或其他格式文档的 Web 网页的技术标准。JSP 文件后缀名为(\*.jsp), 它使用 JSP 标签在 HTML 网页中插入 Java 代码。标签通常以<%开头,以%>结束。因为 JSP 中可以直接写 java 代码,所以 JSP 为我们提供了九大内置对象,也就是 9 个全局对象,可以直接在 JSP 中任何地方使用,但我们不需要记住全部的 9 个内置对象,先了解下常用的四个内置对象:

JSP 内置对象	功能
request	客户端的请求信息被封装在 request 对象中
response	response 对象包含了响应客户端请求的有关信息
pageContext	pageContext对象提供了对JSP页面内所有的对象及名字空间的访问,也就是说它可以访问到本页所在的session,也可以获取request的某一属性值,它相当于页面中所有功能的集大成者,它的本类名也叫pageContext。
session	session对象指的是客户端与服务器的 一次会话,从客户连到服务器的一个 WebApplication 开始,直到客户端与服务 器断开连接为止。

## 新建 JSP 页面

我们先在工程中建一个 JSP 文件:



test.jsp 中的内容为:

```
test.jsp
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8" isELIgnored="false"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4   "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6   <head>
7     <title>测试系统</title>
8   </head>
9   <body>
10    Hello world!
11  </body>
12 </html>
13
14
```

之前也讲过,所有的请求都要通过 DispatcherServlet,所以要将 test.jsp 在 DispatcherServlet 中进行“注册”,以便在请求时服务器不认识请求路径而出现 404 错误。“注册”是在 spring-web.xml 中加入:

```
<mvc:resources mapping="/jsp/**" location="/jsp/" />
```

意思是指请求链接中有 jsp 时,会在工程的 jsp 目录下去找对应的文件。

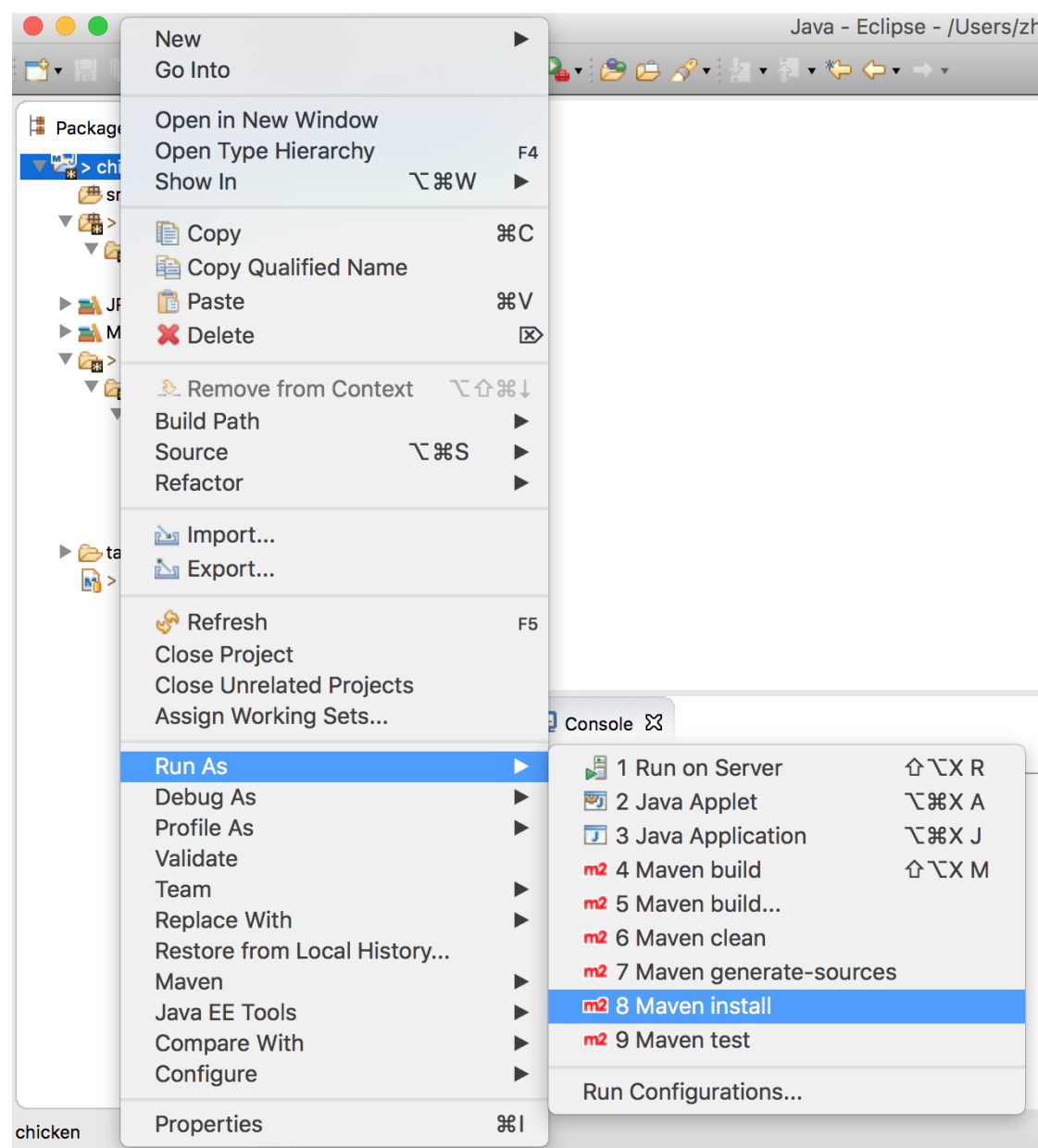
## 编译工程

之前说过, Maven 除了管理 JAR 包外, 另一个重要的功能就是编译, 我们先在 pom.xml 中加入编译时所需要的一个插件:

```
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>${java-version}</source>
        <target>${java-version}</target>
      </configuration>
    </plugin>
  </plugins>
</build> |
```

这个插件的作用就是用来指定 JDK 的版本。如果你使用的 JDK 版本比指定的 JDK 版本低, 则不会被编译通过。加上插件后, 选中工程, 右键:





选中 Maven install 后，就开始编译了，编译完后如下：

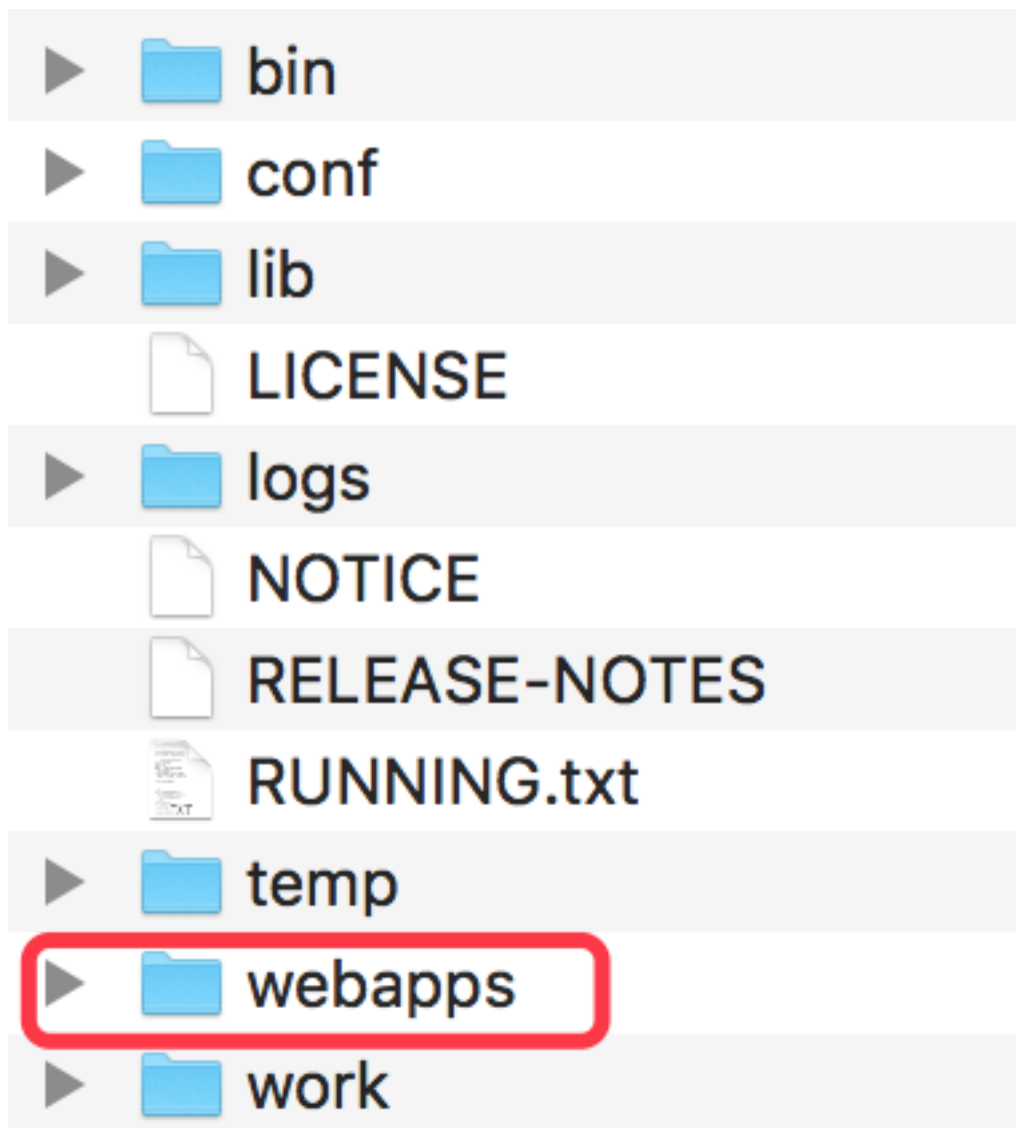
```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ chicken ---
[INFO] Installing /Users/zhangfei/Desktop/workspace/chicken/target/chicken.war to /Use
[INFO] Installing /Users/zhangfei/Desktop/workspace/chicken/pom.xml to /Users/zhangfei
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.476 s
[INFO] Finished at: 2017-02-26T15:28:05+08:00
[INFO] Final Memory: 12M/222M
[INFO] -----
```

上图显示了编译成功，且在工程的 target 目录下与 Maven 本地仓库中各生成了一个 war 包。

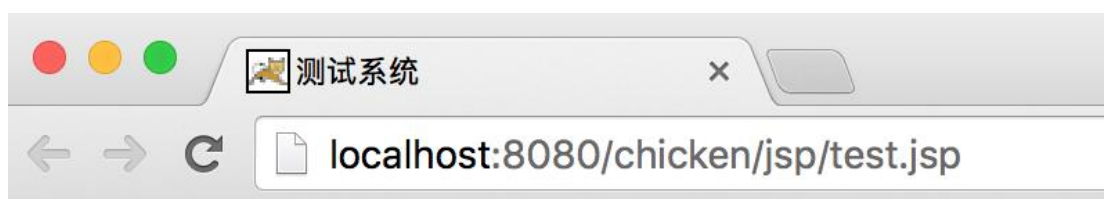
## Tomcat 启动 web 工程

Tomcat 启动 web 工程一般有两种方式, 分别介绍一下。

- 将工程的 target 目录下的 chicken.war 包放在 Tomcat 的 webapps 目录下面, 直接启动 Tomcat。



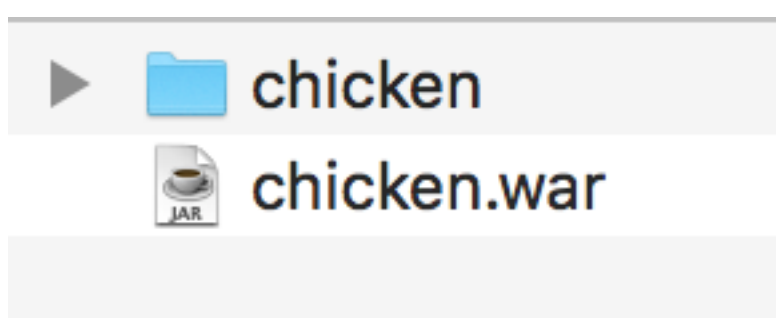
启动后, 在浏览器中输入: <http://localhost:8080/chicken/jsp/test.jsp>  
回车后, 看看我们输出:



Hello world!

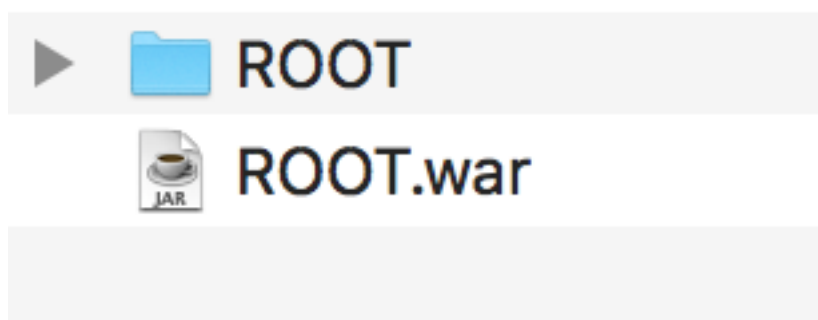
第一个页面正式开启成功了!

我们再回头来看看 tomcat 的 webapps 目录下面:



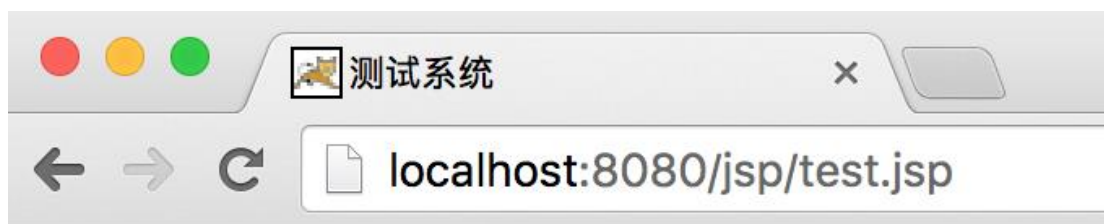
由此可以看出,在 Tomcat 启动时,会将 chicken.war 进行解压,多出了一个 chicken 目录,所以请求的链接地址组成就是域名+端口+解压后的文件夹名+jsp 文件在解压后的文件夹中的路径。

Tomcat 在 webapps 里是有一个默认的目录 ROOT 的,请求 ROOT 目录下的工程时,请求的链接地址就为域名+端口+jsp 文件在 ROOT 文件夹中的路径,所以我们将 chicken.war 改名为 ROOT.war 后放入 webapps 目录下面,并启动 TOMCAT



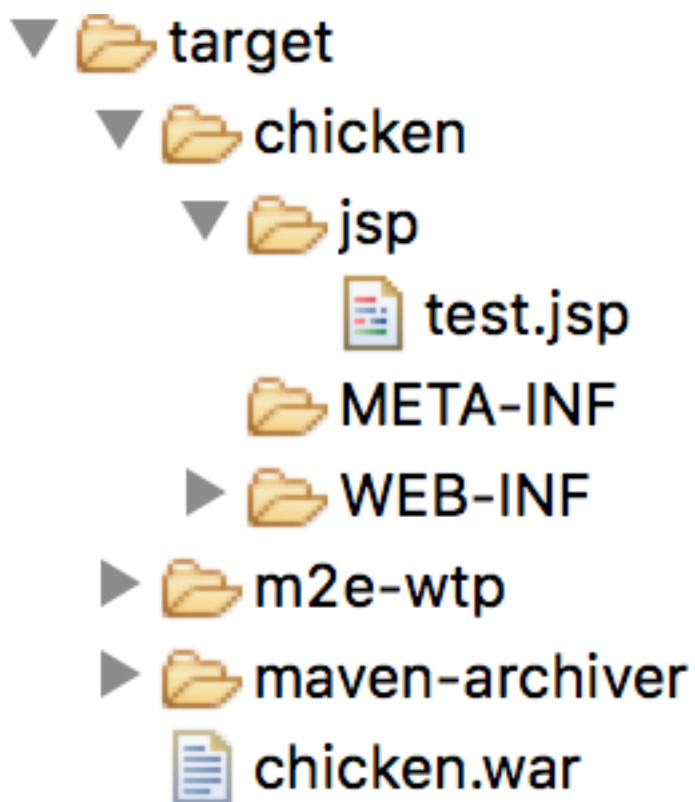
在浏览器中输入: <http://localhost:8080/jsp/test.jsp>

回车后我们看输出:



Hello world!

- 上面的方式是将工程的 `target` 目录下的 `chicken.war` 拷贝至 `tomcat` 下, `tomcat` 同样的提供了一种方式支持将解压后的 `war` 包放在 `tomcat` 外部, 通过某种方式将两者关联起来, 我们来看一下 `target` 目录:



发现编译完成后, 会在 `target` 目录下生成一个 `chicken.war`, 并自动的解压在 `target/chicken` 目录下了, 所以我们只需要将 `tomcat` 与 `target/chicken` 目录关联起来即可。打开 `tomcat` 根目录下的 `conf` 文件夹, 编辑 `server.xml`, 增加如下节点:

```
<Host name="localhost" appBase="webapps"
      unpackWARs="true" autoDeploy="true">

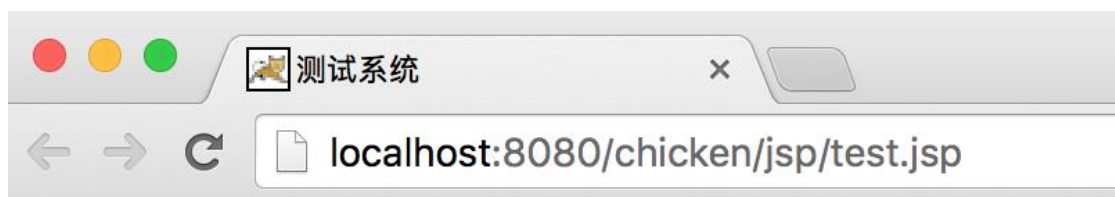
  <!-- SingleSignOn valve, share authentication between web applications
       Documentation at: /docs/config/valve.html -->
  <!--
  <Valve className="org.apache.catalina.authenticator.SingleSignOn" />
  -->

  <!-- Access log processes all example.
       Documentation at: /docs/config/valve.html
       Note: The pattern used is equivalent to using pattern="common" -->
  <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
        prefix="localhost_access_log." suffix=".txt"
        pattern="%h %l %u %t &quot;%r&quot; %s %b" />
  <Context path="chicken" reloadable="false"
        docBase="/Users/zhangfei/Desktop/workspace/chicken/target/chicken" />
</Host>
```

path 是指工程名称, docBase 是指工程的 target/chicken 目录的全路径。保存后启动 Tomcat, 并在浏览器里输入:

<http://localhost:8080/chicken/jsp/test.jsp>

同样的可以得到输出:



Hello world!

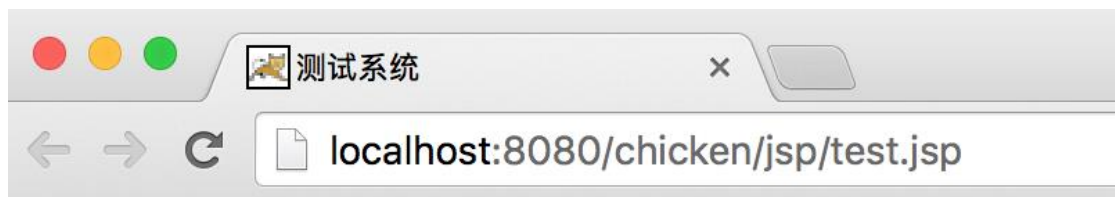
这种分离方式需要修改 server.xml 文件, Tomcat 又提供了一种方式不用修改 server.xml, 将这个节点从 server.xml 里分离出来。打开 Tomcat 根目录下的 conf/Catalina/localhost 目录, 在该目录下新建一个 chicken.xml 文件, 并在 chicken.xml 文件中输入:

```
chicken.xml
1 <Context path="chicken" reloadable="false" docBase="/Users/zhangfei/Desktop/workspace/chicken/target/chicken" />
```

需要注意的是这个 xml 文件的名称与 Context 节点的 path 属性的值要一样, 否则会出现问题。启动 Tomcat 后, 在浏览器中输入:

<http://localhost:8080/chicken/jsp/test.jsp>

同样的可以得到输出:

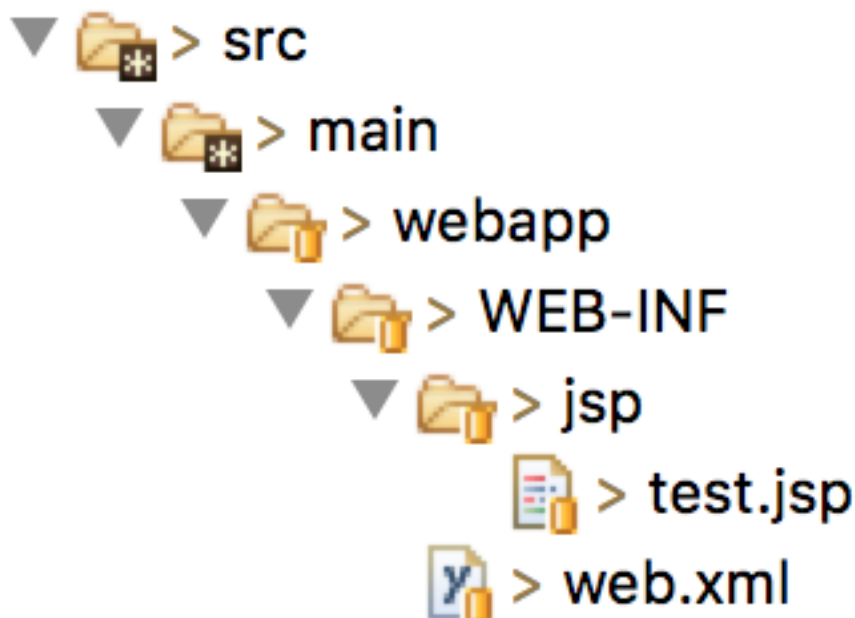


Hello world!

总结: 虽然 Tomcat 有两种启动 war 包的方式, 但细分起来却有 4 种, 我们可以根据习惯, 根据工具, 根据工具的插件等来选择最适合我们的启动方式, 我个人习惯于最后一种启动方式, 即在 localhost 目录下加 xml 文件的启动方式, 在以后的章节中的示例工程也会以该方式来启动 war 包。

## Controller 示例

以上方式是通过对 jsp 文件的直接访问, 但为了系统的安全, 一般情况下, 是不允许直接对 jsp 文件进行直接访问的, web 工程的 webapp/WEB-INF 文件夹里的所有文件, 约定是不允许通过 url 链接进行直接访问的。所以我们将 jsp 文件夹移至 WEB-INF 下面:



并且将 spring-web.xml 里的 mvc:resources 标签给改为:

```
<mvc:resources mapping="/jsp/**" location="/WEB-INF/jsp/" />
```

将工程给 clean 一下: 选中工程->Run as->Maven clean, clean 完成后, 再选中工程->Run as->Maven inatall 进行编译并启动 Tomcat, 并在浏览器中输入:

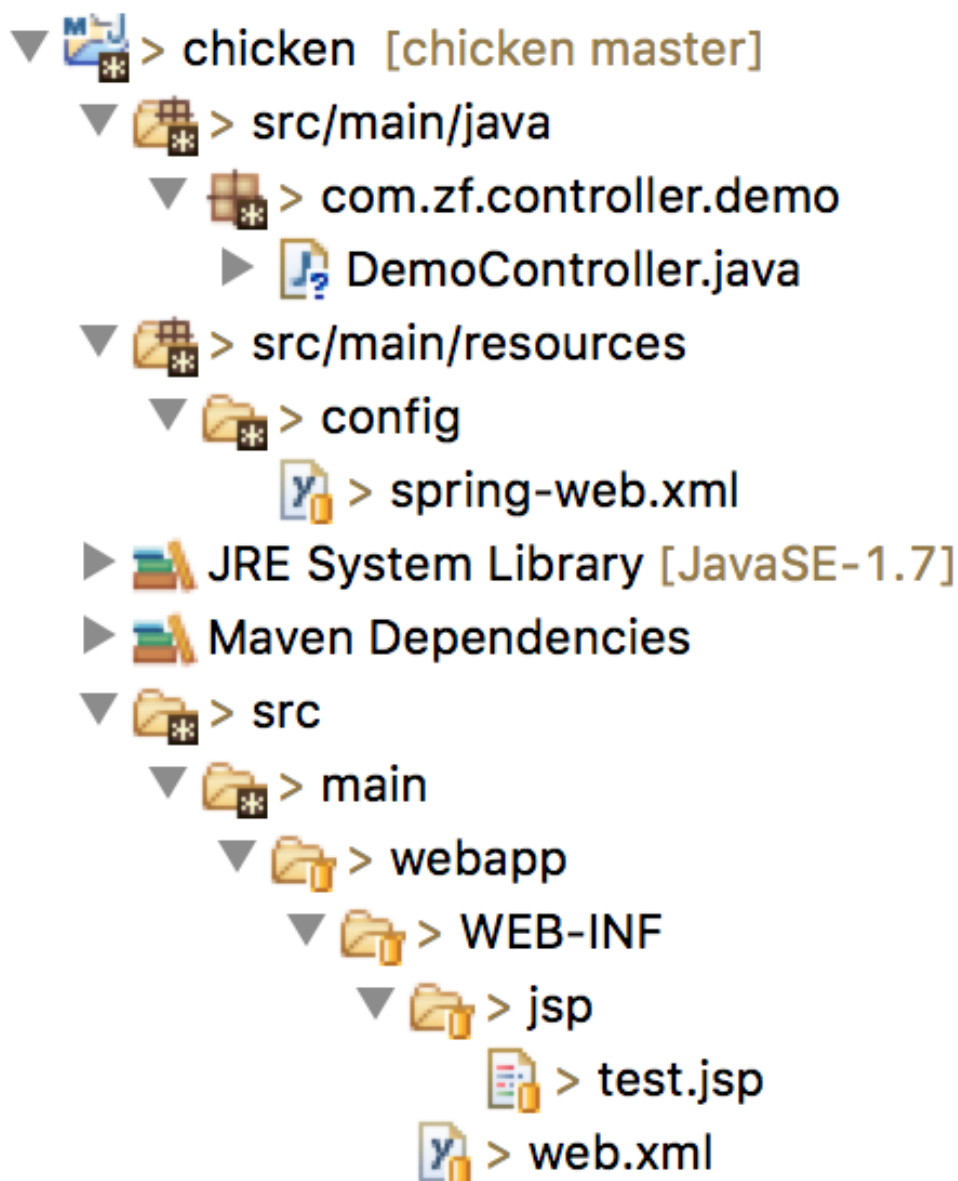
<http://localhost:8080/chicken/jsp/test.jsp>

发现是 404。即表示通过 URL 无法直接访问 WEB-INF 文件夹下的文件, 这时候就需要用工程内部的代码来进行访问了。

首先我们将 spring-web.xml 里的 mvc:resources 标签给注释:

```
<!-- <mvc:resources mapping="/jsp/**" location="/WEB-INF/jsp/" /> -->
```

再在 src/main/java 下建如下的目录与文件:



打开 DemoController.java, 并在里面输入如下的内容:

本文档是由再见理想(QQ408129370)个人编写, 未经授权, 严禁转载。



```
@Controller
@RequestMapping(value="demo")
public class DemoController {

    @RequestMapping(value = "/v1", method = RequestMethod.GET)
    public String demo() {
        return "/WEB-INF/jsp/test.jsp";
    }

}
```

说明: @Controller 注解是 spring 提供的, 表示这个是 mvc 中的 c 层, @RequestMapping 是指访问的路径, @RequestMapping 注解到了类与方法上面, 表示路径是类与方法上的 value 进行连接起来的, 当有如下的请求链接过来后:

<http://localhost:8080/chicken/demo/v1>

demo 方法就会被执行, 相应的响应的 jsp 页面的内容就可以返回给浏览器了。

这个过程有两个需要特别注意的地方:

1. 一个类与方法上的 @RequestMapping 的值加起来应该在整个 web 工程中唯一, 否则当请求过来时, 就不知道执行哪个方法了。
2. 既然方法能够被执行, 那么这个 DemoController 类应该要在 Tomcat 启动时就要被实例化。

问题来了: Tomcat 在启动时, 如何知道哪个类需要实例化? 这就是 @Controller 注解的作用了, 即加了 @Controller 注解的类在 Tomcat 启动时是会被实例化的, 问题又来了, 那如何知道哪些类被加了 @Controller 注解? 解决这个问题的办法是对代码进行扫描, 但如果代码文件过多, 扫描的时间过长, 那么 Tomcat 启动的时间就会变长, 所以, 我们可以用一些配置来规定扫描哪些文件夹, 这样可以减少一些不必要的扫描。之前讲过, Tomcat 启动时, 会加载并执行 web.xml 文件, 我们也分析过那个 web.xml 文件, ContextLoaderListener 监听会负责 spring 配置文件的加载并执行, 于是在 spring 的配置文件 spring-web.xml 文件里加上配置:



```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/spring-mvc.xsd">
```

并加上以下的节点配置:

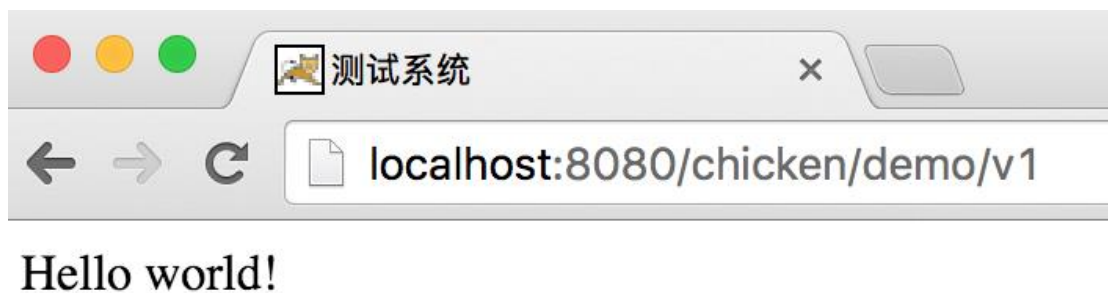
```
<context:component-scan base-package="com.zf" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

以上表示扫描的路径是 com.zf 目录及子目录下的所有的文件，  
context:include-filter 标签是指只扫描@Controller 类。

编译并启动 Tomcat，在浏览器中输入：

<http://localhost:8080/chicken/demo/v1>

输出为：



也并不是类与方法上都必须加上@RequestMapping 注解，例如：

```
@Controller
//@RequestMapping(value="demo")
public class DemoController {

    @RequestMapping(value = "/demo", method = RequestMethod.GET)
    public String demo() {
        return "/WEB-INF/jsp/test.jsp";
    }

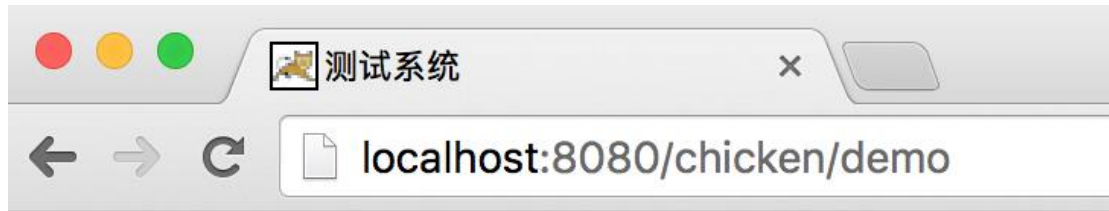
}
```

编译并启动 Tomcat 后，在浏览器中输入：

本文档是由再见理想(QQ408129370)个人编写，未经授权，严禁转载。

<http://localhost:8080/chicken/demo>

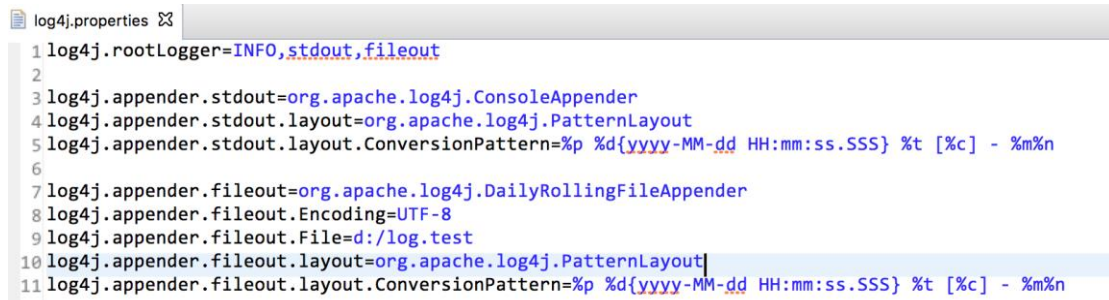
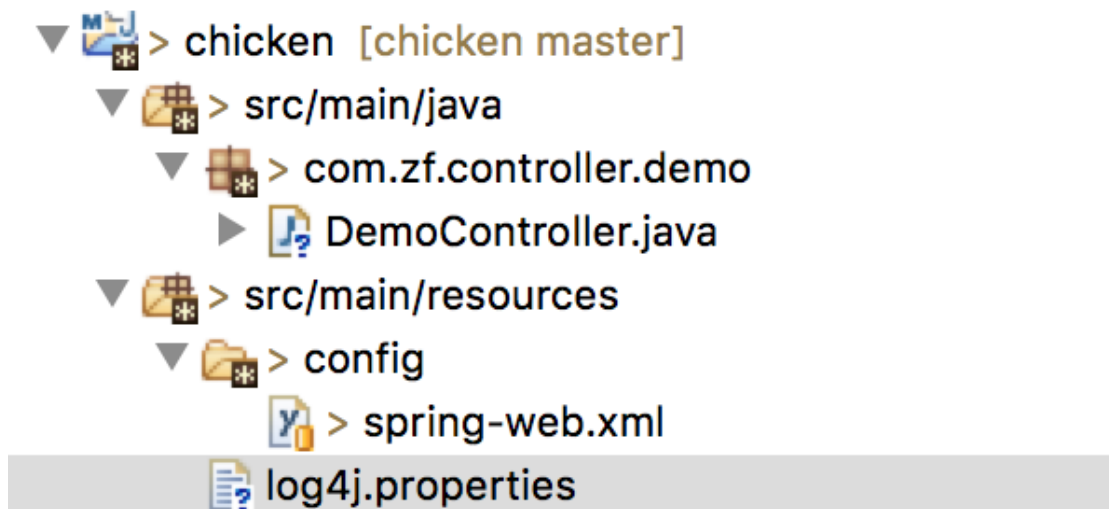
输出为:



Hello world!

## log 配置

log 配置在《测试开发 Java21 天》中讲过，这里不再赘述。需要注意的是 log4j.properties 文件一定要放在 src/main/resources 目录下:



使用方法: 在 pom.xml 文件中, 我们看到用的是 slf4j-log4j12, slf4j 是对 log4j 的封装版本, 在代码中的使用如下:

本文档是由再见理想(QQ408129370)个人编写, 未经授权, 严禁转载。

```
@Controller
@RequestMapping(value="demo")
public class DemoController {

    private final Logger logger = LoggerFactory.getLogger(DemoController.class);

    @RequestMapping(value = "/demo", method = RequestMethod.GET)
    public String demo() {
        logger.info("this is demo.");
        return "/WEB-INF/jsp/test.jsp";
    }
}
```

## 总结

跟着上面的配置一步步的来,你会发现,那些令开发者都头疼的 spring 配置问题,正在被我们一点点的化解,第一个页面也输出成功了,希望大家在开发测试这条路上坚持下去,并最终跟上互联网测试的发展潮流。

## 号外

该文档的后续部分会陆续推出,但不会开放出去,个人正在尝试推出有偿服务,服务内容包括 UI 自动化测试,接口测试,测试平台开发等方面,服务收费为 1000 元/年,在服务的这一年里,会以发放文档的形式给大家,组织大家自己学习,收集大家在自学中的问题,以视频直播的形式进行集中讲解,并且会时不时的请一些测试大牛给大家排忧解难!如果有兴趣的,可以私我!

个人 QQ: 408129370