Spring boot web容器实例化

目录

[1. 介绍](#_Toc1967_WPSOffice_Level1) [2](#_Toc1967_WPSOffice_Level1)

[1.1 默认配置：tomcat](#_Toc10092_WPSOffice_Level2) [2](#_Toc10092_WPSOffice_Level2)

[1.2 其他配置：undertow和jetty](#_Toc2885_WPSOffice_Level2) [2](#_Toc2885_WPSOffice_Level2)

[1.3 Web服务器的配置：application.properties](#_Toc22297_WPSOffice_Level2) [3](#_Toc22297_WPSOffice_Level2)

[2. Web 服务器相关的结构信息](#_Toc10092_WPSOffice_Level1) [5](#_Toc10092_WPSOffice_Level1)

[2.1.1 接口定义](#_Toc32178_WPSOffice_Level2) [5](#_Toc32178_WPSOffice_Level2)

[2.1.2 实现类](#_Toc23319_WPSOffice_Level2) [5](#_Toc23319_WPSOffice_Level2)

[2.2.1 接口介绍](#_Toc16647_WPSOffice_Level2) [5](#_Toc16647_WPSOffice_Level2)

[2.2.2 TomcatServletWebServerFactory](#_Toc32717_WPSOffice_Level2) [5](#_Toc32717_WPSOffice_Level2)

[2.3.1 接口介绍](#_Toc28906_WPSOffice_Level2) [6](#_Toc28906_WPSOffice_Level2)

[2.3.2 实现类](#_Toc22456_WPSOffice_Level2) [6](#_Toc22456_WPSOffice_Level2)

[2.4.1 接口介绍](#_Toc28044_WPSOffice_Level2) [6](#_Toc28044_WPSOffice_Level2)

[2.4.2 实现类](#_Toc9034_WPSOffice_Level2) [7](#_Toc9034_WPSOffice_Level2)

[3. 启动WebServer的流程](#_Toc2885_WPSOffice_Level1) [9](#_Toc2885_WPSOffice_Level1)

[3.1 创建WebServer](#_Toc23187_WPSOffice_Level2) [9](#_Toc23187_WPSOffice_Level2)

[3.1.1 调用链](#_Toc10092_WPSOffice_Level3) [9](#_Toc10092_WPSOffice_Level3)

[3.1.2 创建过程解析](#_Toc2885_WPSOffice_Level3) [9](#_Toc2885_WPSOffice_Level3)

[3.2 启动WebServer](#_Toc12446_WPSOffice_Level2) [12](#_Toc12446_WPSOffice_Level2)

[3.2.1 调用链](#_Toc22297_WPSOffice_Level3) [12](#_Toc22297_WPSOffice_Level3)

[3.2.2 启动过程解析](#_Toc32178_WPSOffice_Level3) [12](#_Toc32178_WPSOffice_Level3)

[4. 使用外部web服务器部署](#_Toc22297_WPSOffice_Level1) [14](#_Toc22297_WPSOffice_Level1)

[4.1 必需的配置](#_Toc27252_WPSOffice_Level2) [14](#_Toc27252_WPSOffice_Level2)

[4.2 可选/建议配置](#_Toc14844_WPSOffice_Level2) [14](#_Toc14844_WPSOffice_Level2)

[4.3 注意事项](#_Toc375_WPSOffice_Level2) [15](#_Toc375_WPSOffice_Level2)

1. 介绍

使用spring boot创建web项目不需要手动部署到诸如tomcat的服务容器上并启动，当项目运行起来的时候，spring boot会自动为创建、部署并启动web容器。Spring boot目前内嵌三种web容器，tomcat，jetty和undertow，其中tomcat是默认的web容器。

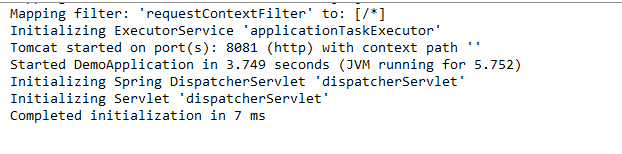
1.1 默认配置：tomcat

Pom.xml文件配置：引入web starter配置即可，starter-web的pom文件中引入了tomcat的starter

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>



运行截图



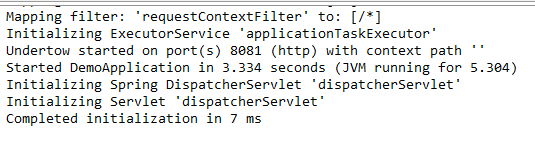
1.2 其他配置：undertow和jetty

Pom.xml文件设置：去掉默认的tomcat的starter配置并引入新容器的starter配置

在多个依赖都被引入的情况下，优先级：tomcat> jetty >undertow

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 <**exclusions**>  
 <**exclusion**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  
 </**exclusion**>  
 </**exclusions**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-undertow</**artifactId**>  
</**dependency**>

运行截图



1.3 Web服务器的配置：application.properties

server 配置通用属性

Server.servlet 配置servlet项目的属性

server.tomcat 配置tomcat的属性

server.port=# 配置程序端口，默认8080

server.connection-timeout=# 配置连接超时时间

server.server-header= #设置响应的Header中Server字段的值

server.servlet.session-timeout= #用户会话session过期时间

server.servlet.context-path= #配置访问路径。默认为/

server.tomcat.uri-encoding = #配置Tomcat编码，默认UTF-8

server.tomcat.basedir = #Tomcat的根目录，如果没有指定，那么将使用临时目录

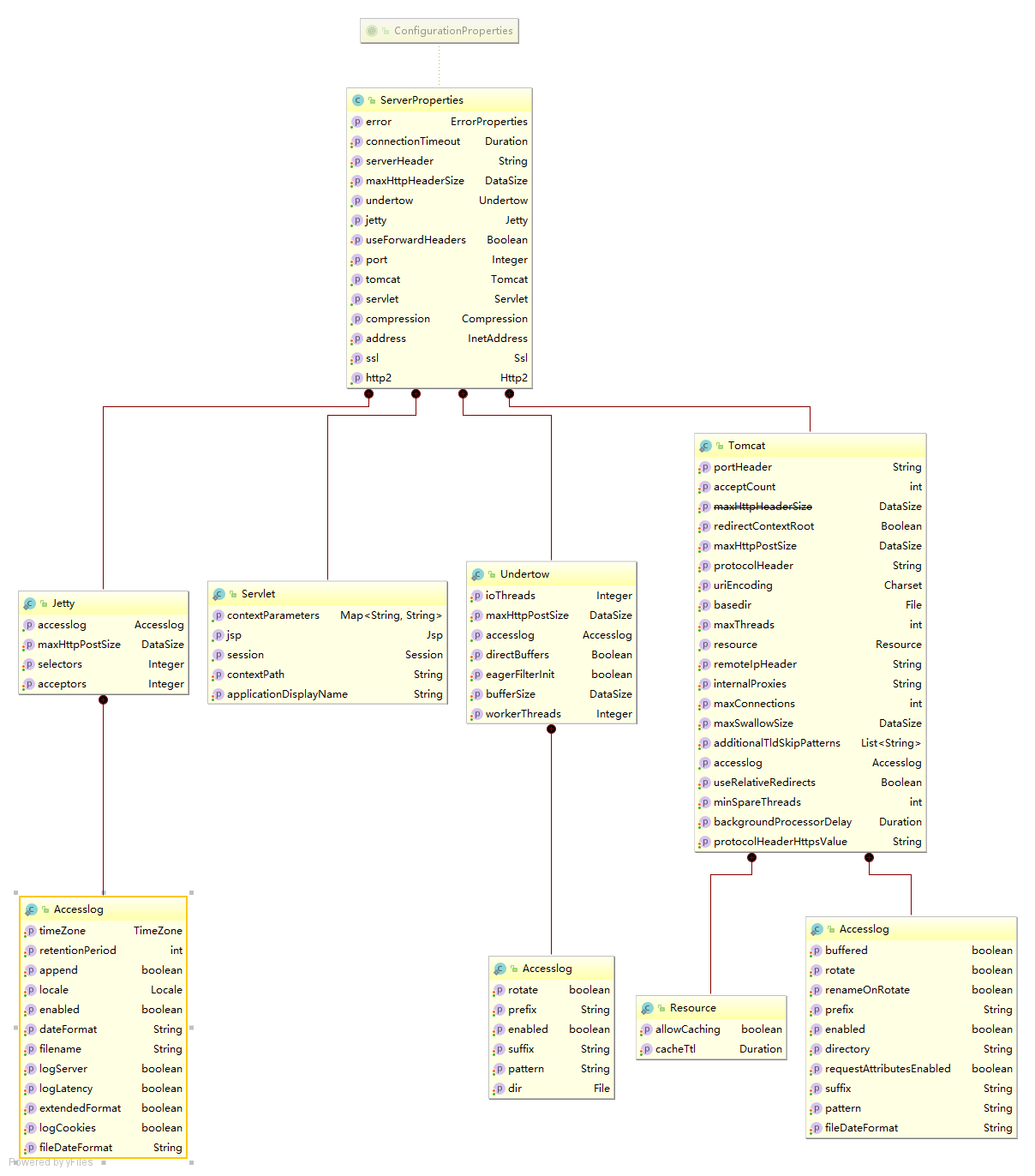
server.tomcat.max-connections= #Tomcat的最大连接数，默认值是10,000

server.tomcat.max-threads= #Tomcat的最大线程数，默认值是200

server.tomcat.resource.allow-caching= #Tomcat中静态资源是否使用缓存，默认开启

server.tomcat.resource.cache-ttl= #Tomcat中静态资源的缓存时间

可配置属性一览，配置相关类org.springframework.boot.autoconfigure.web.ServerProperties:



1. Web 服务器相关的结构信息
   1. WebServer 结构

2.1.1 接口定义

**public interface** WebServer {**void** start() **throws** WebServerException;**void** stop() **throws** WebServerException;**int** getPort();  
}

2.1.2 实现类

以tomcat的实现为例。

**public class** TomcatWebServer **implements** WebServer {

用来存储服务和与之关联的连接器

**private final** Map<Service, Connector[]> **serviceConnectors**;  
tomcat实例  
**private final** Tomcat **tomcat**;

构造函数，传入一个tomcat实例和一个是否自动启动的标志。

**public** TomcatWebServer(Tomcat tomcat, **boolean** autoStart);

初始化方法，在构造函数内被调用

**private void** initialize() **throws** WebServerException;

...

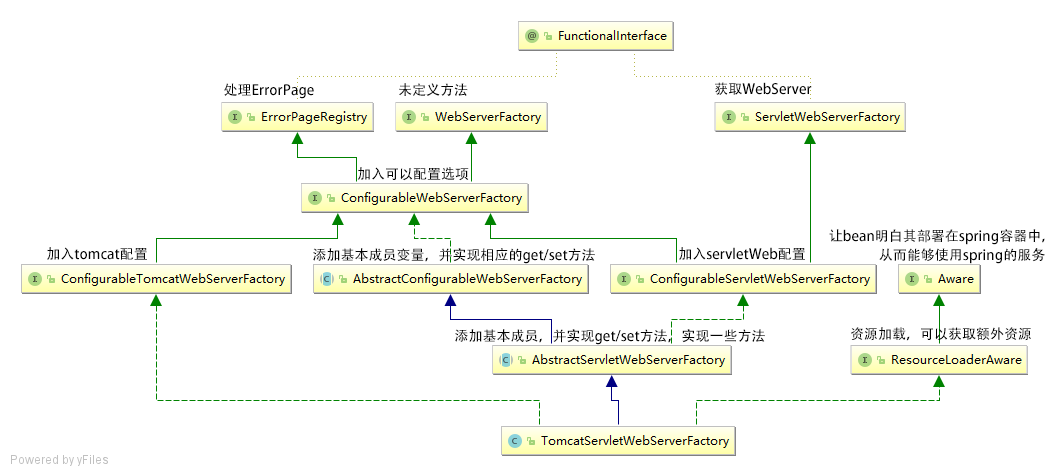
}

* 1. WebServerFactory 结构

2.2.1 接口介绍

依然以tomcat服务器为例。虽然TomcatWebServer的构造函数是公开的，但是在spring boot中并不推荐直接创建一个TomcatWebServer的实例，而是推荐使用相应的Factory来创建WebServer的实例。Springboot里实现了两个关于tomcat服务器的Factory，分别是TomcatReactiveWebServerFactory（对应于reactive web项目） 和TomcatServletWebServerFactory（对应于servlet web项目）。

2.2.2 TomcatServletWebServerFactory



接口ServletWebServerFactory定义了获取WebServer的方法，getWebServer

**public interface** ServletWebServerFactory {WebServer getWebServer(ServletContextInitializer... initializers);

}

类TomcatServletWebServerFactory实现了getWebServer方法并返回一个TomcatWebServer对象。

* 1. WebServerFactoryCustomizer 结构
     1. 接口介绍

在创建WebServerFactory时对其进行自定义操作

**public interface** WebServerFactoryCustomizer<T **extends** WebServerFactory> {**void** customize(T factory);  
}

* + 1. 实现类

以tomcat实现类为例:通过ServerProperties 来获application.properties中设置的属性，并应用到WebServerFactory中。

**public class** TomcatWebServerFactoryCustomizer **implements** WebServerFactoryCustomizer<ConfigurableTomcatWebServerFactory>, Ordered {  
 **private final** Environment **environment**;  
 **private final** ServerProperties **serverProperties**;  
 **public** TomcatWebServerFactoryCustomizer(Environment environment,  
 ServerProperties serverProperties);  
 @Override  
 **public void** customize(ConfigurableTomcatWebServerFactory factory) {  
 ServerProperties properties = **this**.**serverProperties**;  
 ServerProperties.Tomcat tomcatProperties = properties.getTomcat();  
 PropertyMapper propertyMapper = PropertyMapper.*get*();  
 propertyMapper.from(tomcatProperties::getBasedir).whenNonNull()  
 .to(factory::setBaseDirectory);  
 ...  
 }

...

}

2.4 WebServerFactoryCustomizerBeanPostProcessor

实现了BeanPostProcessor接口，在创建WebServerFactory时应用所有与之相关的WebServerFactoryCustomizer的，并返回自定义之后的Factory。

2.4.1 接口介绍

BeanPostProcessor接口：

定义了两个方法，分别在bean被初始化之前被调用的和在bean被初始化之后被调用。

**public interface** BeanPostProcessor {@Nullable  
 **default** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  
 **return** bean;  
 }@Nullable  
 **default** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  
 **return** bean;  
 }  
  
}

2.4.2 实现类

在方法postProcessBeforeInitialization中获取与传入WebServerFactory相关的Customizer并回调它们的customize方法。

**public class** WebServerFactoryCustomizerBeanPostProcessor  
 **implements** BeanPostProcessor, BeanFactoryAware {  
 **private** ListableBeanFactory **beanFactory**;  
 **private** List<WebServerFactoryCustomizer<?>> **customizers**;  
 @Override  
 **public void** setBeanFactory(BeanFactory beanFactory);  
 @Override  
 **public** Object postProcessBeforeInitialization(Object bean, String beanName);  
 @Override  
 **public** Object postProcessAfterInitialization(Object bean, String beanName);

@SuppressWarnings(**"unchecked"**)  
 **private void** postProcessBeforeInitialization(WebServerFactory webServerFactory) {  
 LambdaSafe  
 .*callbacks*(WebServerFactoryCustomizer.**class**, getCustomizers(), webServerFactory)  
 .withLogger(WebServerFactoryCustomizerBeanPostProcessor.**class**)  
 .invoke((customizer) -> customizer.customize(webServerFactory));  
 }  
  
}

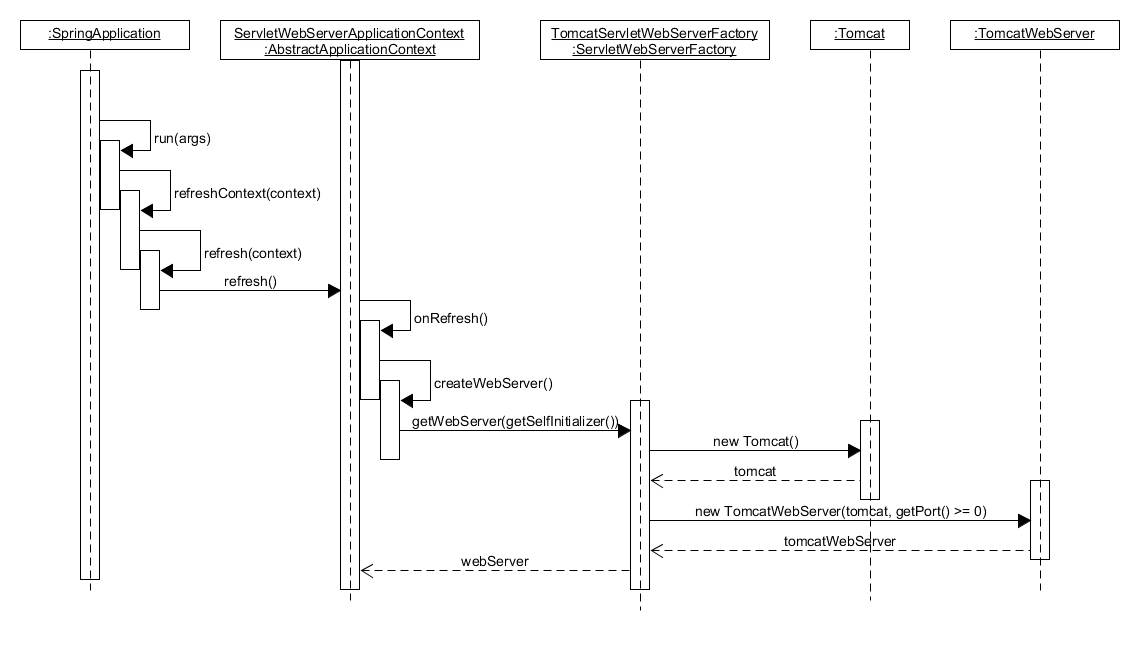
2.5 ServletWebServerFactoryConfiguration

ServletWebServerFactory被实例化的地方，满足条件的bean才会被加载，有优先性。

@Configuration  
**class** ServletWebServerFactoryConfiguration {  
 @Configuration  
 @ConditionalOnClass({ Servlet.**class**, Tomcat.**class**, UpgradeProtocol.**class** })  
 @ConditionalOnMissingBean(value = ServletWebServerFactory.**class**, search = SearchStrategy.***CURRENT***)  
 **public static class** EmbeddedTomcat {  
 @Bean  
 **public** TomcatServletWebServerFactory tomcatServletWebServerFactory() {  
 **return new** TomcatServletWebServerFactory();  
 }  
 }  
@Configuration  
 @ConditionalOnClass({ Servlet.**class**, Server.**class**, Loader.**class**,  
 WebAppContext.**class** })  
 @ConditionalOnMissingBean(value = ServletWebServerFactory.**class**, search = SearchStrategy.***CURRENT***)  
 **public static class** EmbeddedJetty {  
  
 @Bean  
 **public** JettyServletWebServerFactory JettyServletWebServerFactory() {  
 **return new** JettyServletWebServerFactory();  
 }  
 }  
@Configuration  
 @ConditionalOnClass({ Servlet.**class**, Undertow.**class**, SslClientAuthMode.**class** })  
 @ConditionalOnMissingBean(value = ServletWebServerFactory.**class**, search = SearchStrategy.***CURRENT***)  
 **public static class** EmbeddedUndertow {  
 @Bean  
 **public** UndertowServletWebServerFactory undertowServletWebServerFactory() {  
 **return new** UndertowServletWebServerFactory();  
 }  
 }  
}

1. 启动WebServer的流程
   1. 创建WebServer
      1. 调用链

只保留了最直接关联的方法调用：



* + 1. 创建过程解析

（1）ServletWebServerApplicationContext.createWebServer方法：

先判断是否存在WebServer和servletContext的实例：第一次启动时是不存在这两个实例的，所以先获取WebServerFactory，然后通过Factory来创建WebServer的实例。

**private void** createWebServer() {  
 WebServer webServer = **this**.**webServer**;  
 ServletContext servletContext = getServletContext();  
 **if** (webServer == **null** && servletContext == **null**) {  
 ServletWebServerFactory factory = getWebServerFactory();   
 **this**.**webServer** = factory.getWebServer(getSelfInitializer());  
 }  
 **else if** (servletContext != **null**) {  
 **try** {  
 getSelfInitializer().onStartup(servletContext);  
 }  
 **catch** (ServletException ex) {  
 **throw new** ApplicationContextException(**"Cannot initialize servlet context"**,ex);  
 }  
 }  
 initPropertySources();  
}

（2）ServletWebServerApplicationContext.getWebServerFactory方法：

通过BeanFactory来获取TomcatServletWebServerFactory

**protected** ServletWebServerFactory getWebServerFactory() {String[] beanNames = getBeanFactory()  
 .getBeanNamesForType(ServletWebServerFactory.**class**);  
 **if** (beanNames.**length** == 0) {  
 **throw new** ApplicationContextException(  
 **"Unable to start ServletWebServerApplicationContext due to missing "**+ **"ServletWebServerFactory bean."**);  
 }  
 **if** (beanNames.**length** > 1) {  
 **throw new** ApplicationContextException(  
 **"Unable to start ServletWebServerApplicationContext due to multiple "** + **"ServletWebServerFactory beans : "** + StringUtils.*arrayToCommaDelimitedString*(beanNames));  
 }  
 **return** getBeanFactory().getBean(beanNames[0], ServletWebServerFactory.**class**);  
}

得到WebServerFactory对象后，调用WebServerFactory的getWebServer方法来获取WebServer。

（3）TomcatServletWebServerFactory.getWebServer方法：

**public** WebServer getWebServer(ServletContextInitializer... initializers) {  
 Tomcat tomcat = **new** Tomcat();  
 File baseDir = (**this**.**baseDirectory** != **null**) ? **this**.**baseDirectory** : createTempDir(**"tomcat"**);  
 tomcat.setBaseDir(baseDir.getAbsolutePath());  
 Connector connector = **new** Connector(**this**.**protocol**);  
 tomcat.getService().addConnector(connector);  
 customizeConnector(connector);  
 tomcat.setConnector(connector);  
 tomcat.getHost().setAutoDeploy(**false**);  
 configureEngine(tomcat.getEngine());  
 **for** (Connector additionalConnector : **this**.**additionalTomcatConnectors**) {  
 tomcat.getService().addConnector(additionalConnector);  
 }  
 prepareContext(tomcat.getHost(), initializers);  
 **return** getTomcatWebServer(tomcat);  
}

新实例化了一个tomcat对象，并对其进行配置，然后把利用这个tomcat对象实例一个tomcatWebServer并返回。

（4）TomcatServletWebServerFactory.getTomcatWebServer：

实例化一个tomcatWebServer并返回。

**protected** TomcatWebServer getTomcatWebServer(Tomcat tomcat) {  
 **return new** TomcatWebServer(tomcat, getPort() >= 0);  
}

1. TomcatWebServer构造函数：

**public** TomcatWebServer(Tomcat tomcat, **boolean** autoStart) {  
 Assert.*notNull*(tomcat, **"Tomcat Server must not be null"**);  
 **this**.**tomcat** = tomcat;  
 **this**.**autoStart** = autoStart;  
 initialize();  
}

1. TomcatWebServer.initialize方法

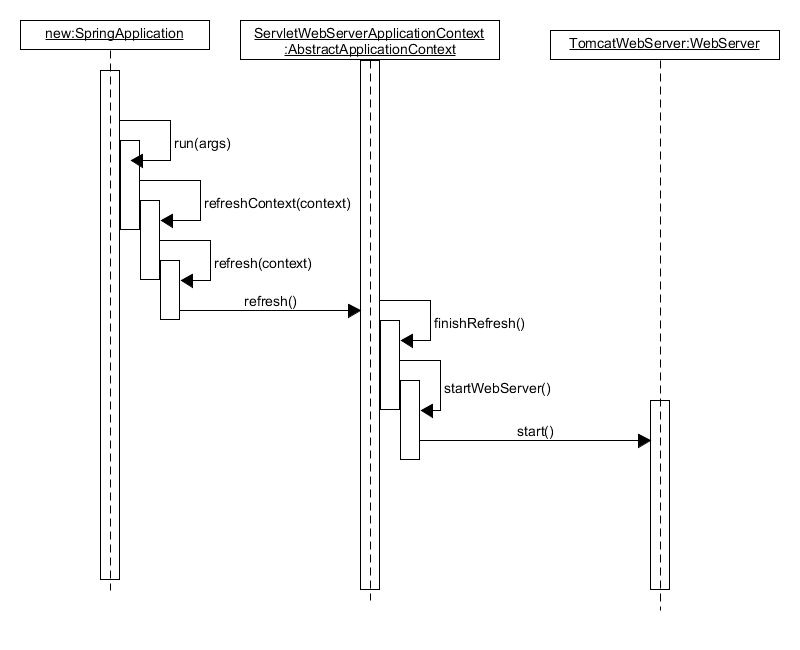
**private void** initialize() **throws** WebServerException {  
 **synchronized** (**this**.**monitor**) {  
 **try** {  
// 将实例ID加到引擎名上：engine.setName(engine.getName() + **"-"** + instanceId)  
 addInstanceIdToEngineName();

Context context = findContext(); // 添加启动时的监听事件  
 context.addLifecycleListener((event) -> {  
 **if** (context.equals(event.getSource())  
 && Lifecycle.***START\_EVENT***.equals(event.getType())) {  
// 在启动时移除服务连接：service.removeConnector(connector)，但是保存在Map

removeServiceConnectors(); // **serviceConnectors** 里  
 }  
 });  
**this**.**tomcat**.start(); // 启动tomcat来触发初始化事件  
rethrowDeferredStartupExceptions(); // 重新抛出异常  
 **try** {  
 ContextBindings.*bindClassLoader*(context, context.getNamingToken(), getClass().getClassLoader());  
 }  
 **catch** (NamingException ex) {  
}

// 因为tomcat的线程都是守护线程，所以创建一个非守护线程来务startDaemonAwaitThread(); // 防止关闭tomcat时服器立刻停止  
 }  
 **catch** (Exception ex) {  
 stopSilently();  
 **throw new** WebServerException(**"Unable to start embedded Tomcat"**, ex);  
 }  
 }  
}

* 1. 启动WebServer
     1. 调用链



* + 1. 启动过程解析

1. ServletWebServerApplicationContext.startWebServer

如果webServer不为空，则调用WebServer的start方法。

**private** WebServer startWebServer() {  
 WebServer webServer = **this**.**webServer**;  
 **if** (webServer != **null**) {  
 webServer.start();   
 }  
 **return** webServer;  
}

1. TomcatWebServer.start

@Override  
**public void** start() **throws** WebServerException {  
 **synchronized** (**this**.**monitor**) {  
 **if** (**this**.**started**) {  
 **return**;  
 }  
 **try** {

// 从Map **serviceConnectors**里读取service对应的connector并重新添加：

// service.addConnector(connector)  
 addPreviouslyRemovedConnectors();  
 Connector connector = **this**.**tomcat**.getConnector();  
 **if** (connector != **null** && **this**.**autoStart**) {  
 performDeferredLoadOnStartup();  
 }

// 检查与tomcat绑定的所有连接是否都已经启动  
 checkThatConnectorsHaveStarted();

// 将状态设为已启动  
 **this**.**started** = **true**;  
 ***logger***.info(**"Tomcat started on port(s): "** + getPortsDescription(**true**)  
 + **" with context path '"** + getContextPath() + **"'"**);  
 }  
 **catch** (ConnectorStartFailedException ex) {  
 stopSilently();  
 **throw** ex;  
 }  
 **catch** (Exception ex) {  
 **throw new** WebServerException(**"Unable to start embedded Tomcat server"**, ex);  
 }  
 **finally** {  
 Context context = findContext();  
 ContextBindings.*unbindClassLoader*(context, context.getNamingToken(), getClass().getClassLoader());  
 }  
 }  
}

1. 使用外部web服务器部署

4.1 必需的配置

如果不想使用spring boot的内置web服务器，而是使用外部web服务器来部署，那么需要额外的配置：

1. 修改pom文件：把打包方式改为war；starter-tomcat的依赖的作用范围（scope属性）改为provided（在打包时排除），将依赖改为provided而不是直接移除依赖的的好处是依然可以在测试时使用内置tomcat，这样便于测试。

<**packaging**>war</**packaging**>

...

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  
 <**scope**>provided</**scope**>  
</**dependency**>

1. 继承抽象类SpringBootServletInitializer并实现configure方法

抽象类SpringBootServletInitializer实现了接口WebApplicationInitializer，是为了运行传统war项目而设计出来的。为了实现在spring boot项目使用外部服务器部署，需要继承抽象类SpringBootServletInitializer并实现configure方法

**public class** ServletInitializer **extends** SpringBootServletInitializer {  
  
 @Override  
 **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
 **return** application.sources(DemoApplication.**class**);  
 }  
  
}

4.2 可选/建议配置

（1）在pom文件build属性中添加finalName的属性，finalName属性可以设置最后打包出来的war包的名字，如果不添加这个属性也没有问题，只是打包出来的war包的名字会含有版本信息。

<**build**>  
 <**finalName**>demo</**finalName**>  
...  
</**build**>

（2）在application.properties文件中把server相关选项设置为外部服务器的设置：

比如服务器的端口和ContextPath属性，这样便于检查测试时和部署时的行为是否一致。

**server.port**=**8081  
server.servlet.context-path**=**/demo**

4.3 注意事项

application.properties文件中关于server的所有配置对于外部服务器都是无效的，properties中的设置全都是针对内置服务器的，所以不会对外部服务器产生任何影响。