Spring boot web 容器实例化

目录

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>
```

spring-boot-starter-web-2.1.0.RELEASE.pom × ServerProperties.java

```
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
    <version>2.1.0.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <version>2.1.0.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
    <scope>compile</scope>
  </dependency>
```

运行截图

```
Mapping filter: 'requestContextFilter' to: [/*]
Initializing ExecutorService 'applicationTaskExecutor'
Tomcat started on port(s): 8081 (http) with context path ''
Started DemoApplication in 3.749 seconds (JVM running for 5.752)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 7 ms
```

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

```xml
<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

```
Mapping filter: 'requestContextFilter' to: [/*]
Initializing ExecutorService 'applicationTaskExecutor'
Undertow started on port(s) 8081 (http) with context path ''
Started DemoApplication in 3.334 seconds (JVM running for 5.304)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 7 ms
```

1.3 Web 服务器的配置：application.properties

server 配置通用属性

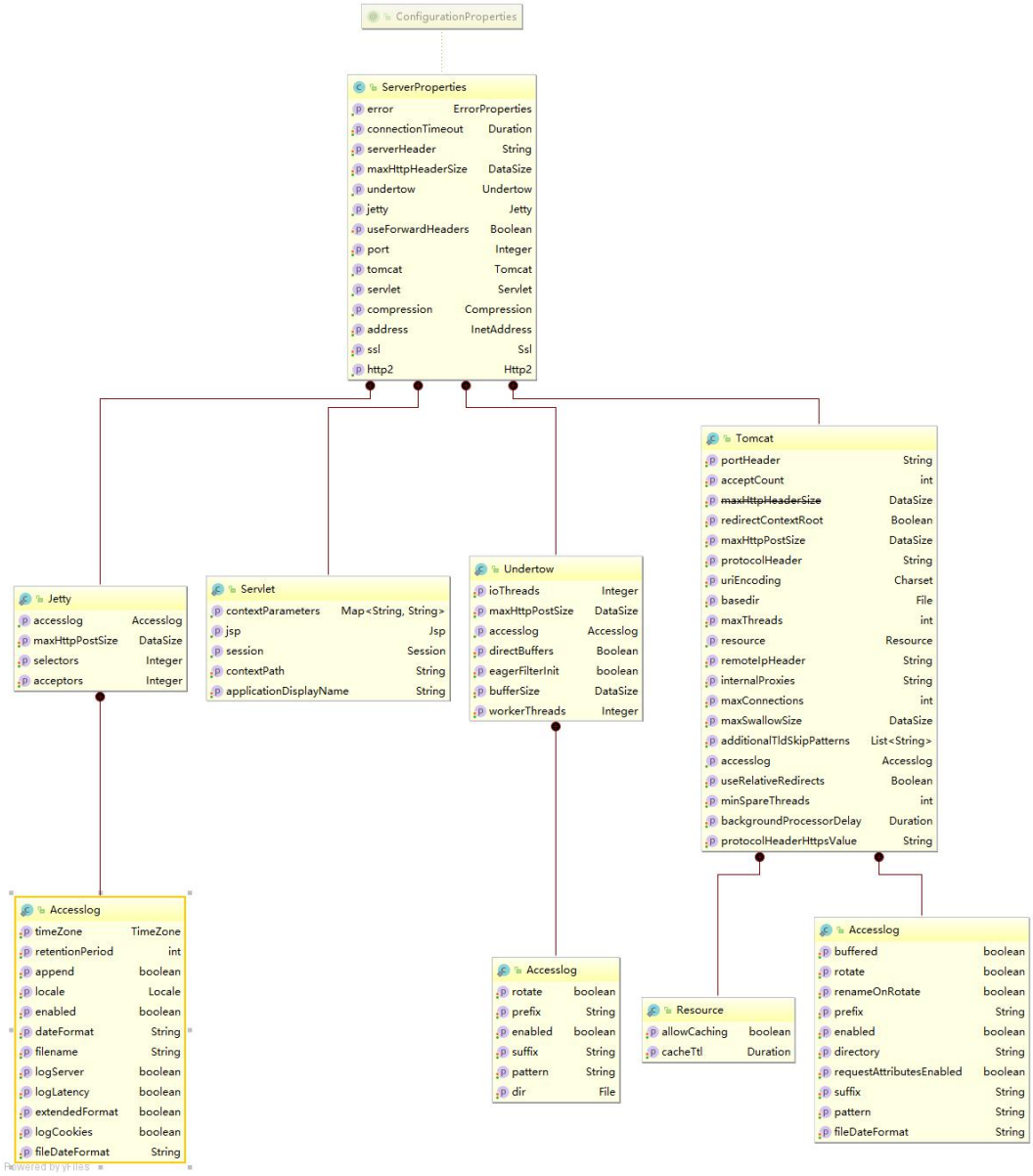
Server.servlet 配置 servlet 项目的属性

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：



2. Web 服务器相关的结构信息

2.1 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。

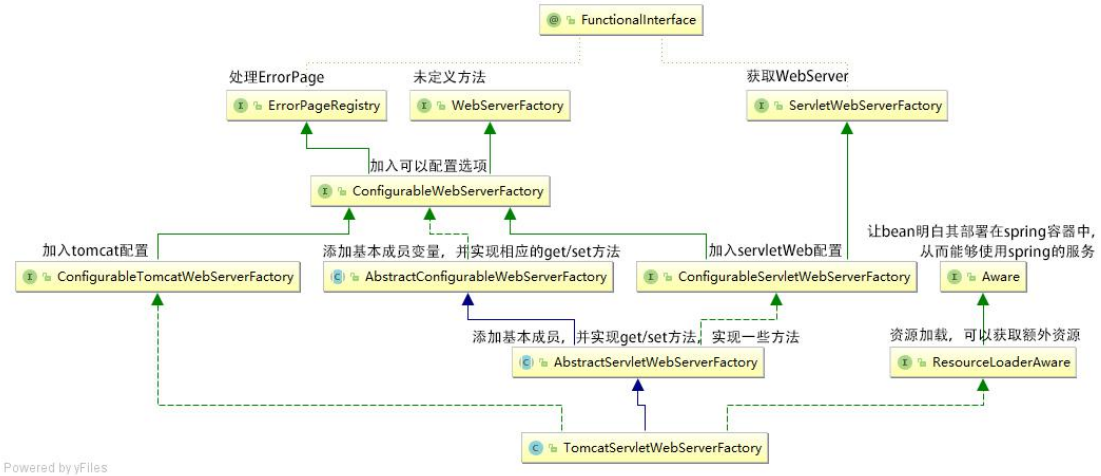
```
public class TomcatWebServer implements WebServer {
用来存储服务和与之关联的连接器
private final Map<Service, Connector[]> serviceConnectors;
tomcat 实例
private final Tomcat tomcat;
构造函数，传入一个 tomcat 实例和一个是否自动启动的标志。
public TomcatWebServer(Tomcat tomcat, boolean autoStart);
初始化方法，在构造函数内被调用
private void initialize() throws WebServerException;
...
}
```

2.2 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。

2.3 WebServerFactoryCustomizer 结构

2.3.1 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);
}
```

2.3.2 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。

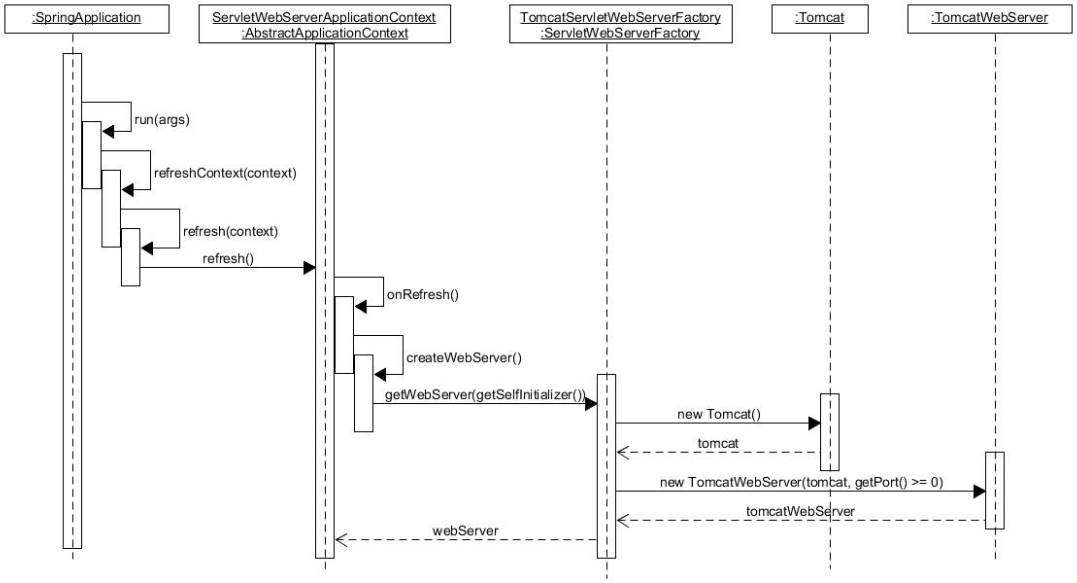
```
@Configuration
class ServletWebServerFactoryConfiguration {
   @Configuration
   @ConditionalOnClass({ Servlet.class, Tomcat.class,
UpgradeProtocol.class })
   @ConditionalOnMissingBean(value = ServletWebServerFactory.class,
search = SearchStrategy.CURRENT)
   public static class EmbeddedTomcat {
      @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();
      }
   }
}
```

3. 启动 WebServer 的流程

3.1 创建 WebServer

3.1.1 调用链

只保留了最直接关联的方法调用：



3.1.2 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);
}
```

（5）TomcatWebServer 构造函数：

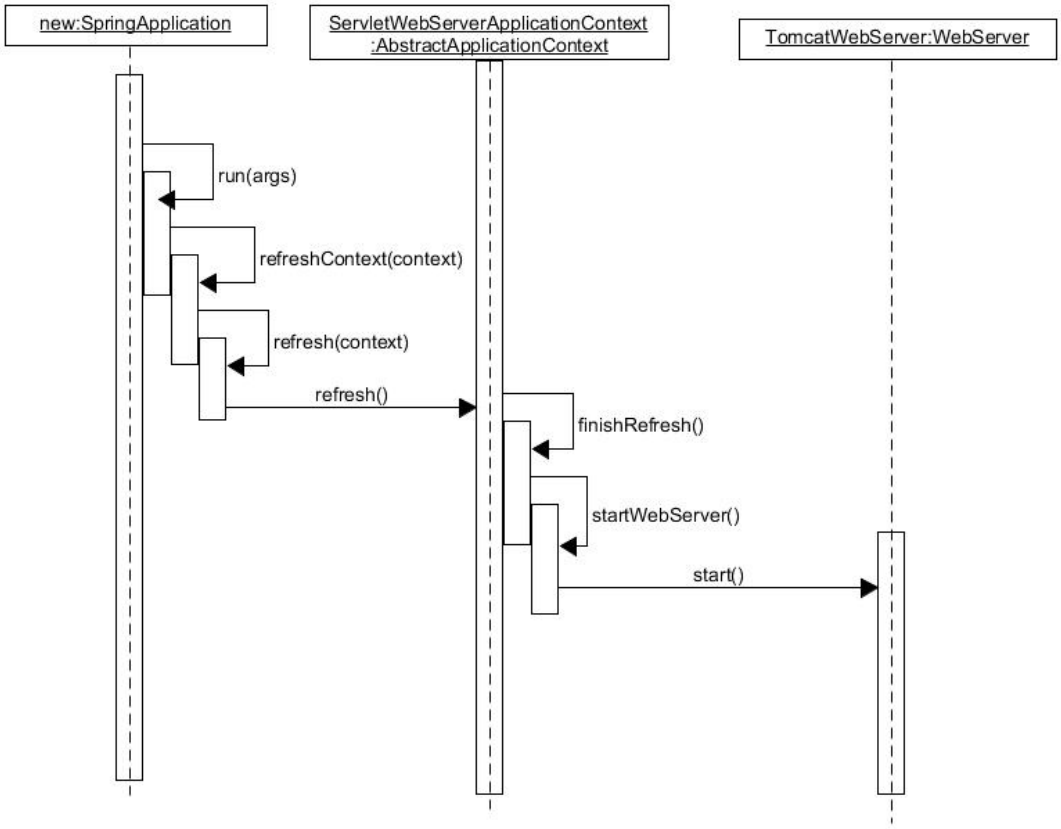
```
public TomcatWebServer(Tomcat tomcat, boolean autoStart) {
   Assert.notNull(tomcat, "Tomcat Server must not be null");
   this.tomcat = tomcat;
   this.autoStart = autoStart;
   initialize();
}
```

（6）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();
```

3.2 启动 WebServer

3.2.1 调用链



3.2.2 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;
}
```

（2）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());
      }
   }
}
```