

# 深入SpringBoot核心注解原理

原创 后端技术精选 2019-05-20 15:01:11 5724 收藏 21

版权

分类专栏: [Springboot](#) 文章标签: [springboot](#) [springboot注解](#) [springboot核心](#)

## 源码分享: Javaweb练手项目下载

今天跟大家来探讨下SpringBoot的核心注解@SpringBootApplication以及run方法, 理解下springBoot为什么不需要XML, 达到零配置

首先我们先来看段代码

```
@SpringBootApplication
public class StartEurekaApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(StartEurekaApplication.class, args);
    }
}
```

我们点进@SpringBootApplication来看

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

}
```

上面的元注解我们在这里不在做解释, 相信大家在开发当中肯定知道, 我们要来说@SpringBootConfiguration @EnableAutoConfiguration 这两个注解, 到这里我们知道 SpringBootApplication注解里除了元注解, 我们可以看到又是@SpringBootConfiguration, @EnableAutoConfiguration, @ComponentScan的组合注解, 官网上也有详细说明, 那我们现在把目光投向这三个注解。

首先我们先来看 @SpringBootConfiguration, 那我们点进来看

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

我们可以看到这个注解除了元注解以外, 就只有一个@Configuration, 那也就是说这个注解相当于@Configuration, 所以这两个注解作用是一样的, 那他是干嘛的呢, 相信很多人都知道, 它是让我们能够去注册一些额外的Bean, 并且导入一些额外的配置。那@Configuration还有一个作用就是把该类变成一个配置类, 不需要额外

的XML进行配置。所以@SpringBootConfiguration就相当于@Configuration。

那我们继续来看下一个@EnableAutoConfiguration，这个注解官网说是 让Spring自动去进行一些配置，那我们点进来

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
}
```

可以看到它是由 @AutoConfigurationPackage，@Import(EnableAutoConfigurationImportSelector.class)这两个而组成的，我们先说@AutoConfigurationPackage，他是说：让包中的类以及子包中的类能够被自动扫描到spring容器中。我们来看@Import(EnableAutoConfigurationImportSelector.class)这个是核心，之前我们说自动配置，那他到底帮我们配置了什么，怎么配置的？

就和@Import(EnableAutoConfigurationImportSelector.class)息息相关，程序中默认使用的类就自动帮我们找到。我们来看EnableAutoConfigurationImportSelector.class

```
public class EnableAutoConfigurationImportSelector
    extends AutoConfigurationImportSelector {

    @Override
    protected boolean isEnabled(AnnotationMetadata metadata) {
        if (getClass().equals(EnableAutoConfigurationImportSelector.class)) {
            return getEnvironment().getProperty(
                EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY, Boolean.class,
                true);
        }
        return true;
    }
}
```

可以看到他继承了AutoConfigurationImportSelector我们继续来看AutoConfigurationImportSelector，这个类有一个方法

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = filter(configurations, autoConfigurationMetadata);
        fireAutoConfigurationImportEvents(configurations, exclusions);
    }
}
```

```

        return configurations.toArray(new String[configurations.size()]);    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}

```

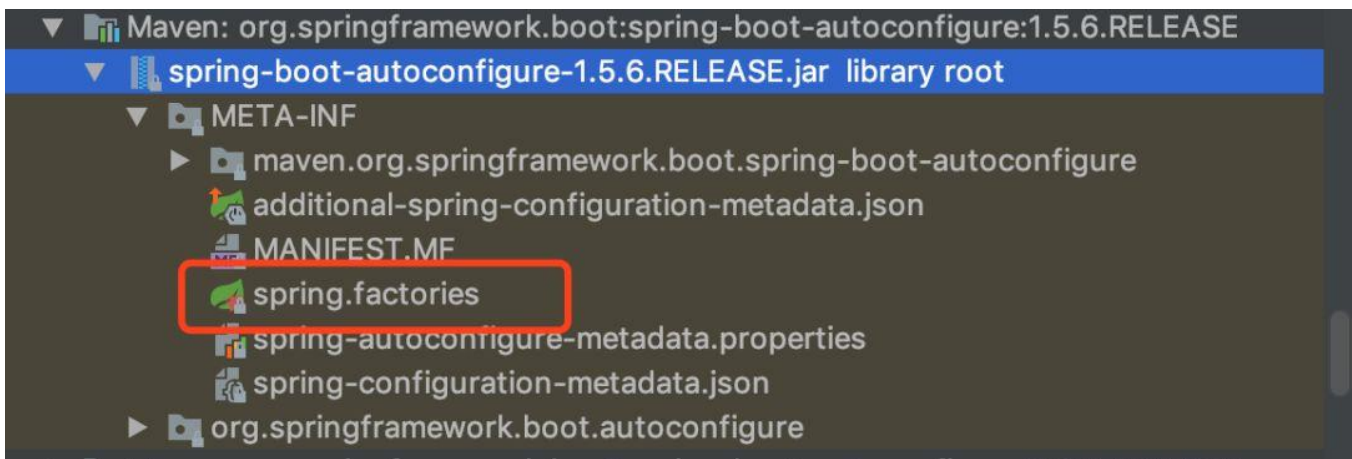
这个类会帮你扫描那些类自动去添加到程序当中。我们可以看到getCandidateConfigurations()这个方法，他的作用就是引入系统已经加载好的一些类，到底是那些类呢，我们点进去看一下

```

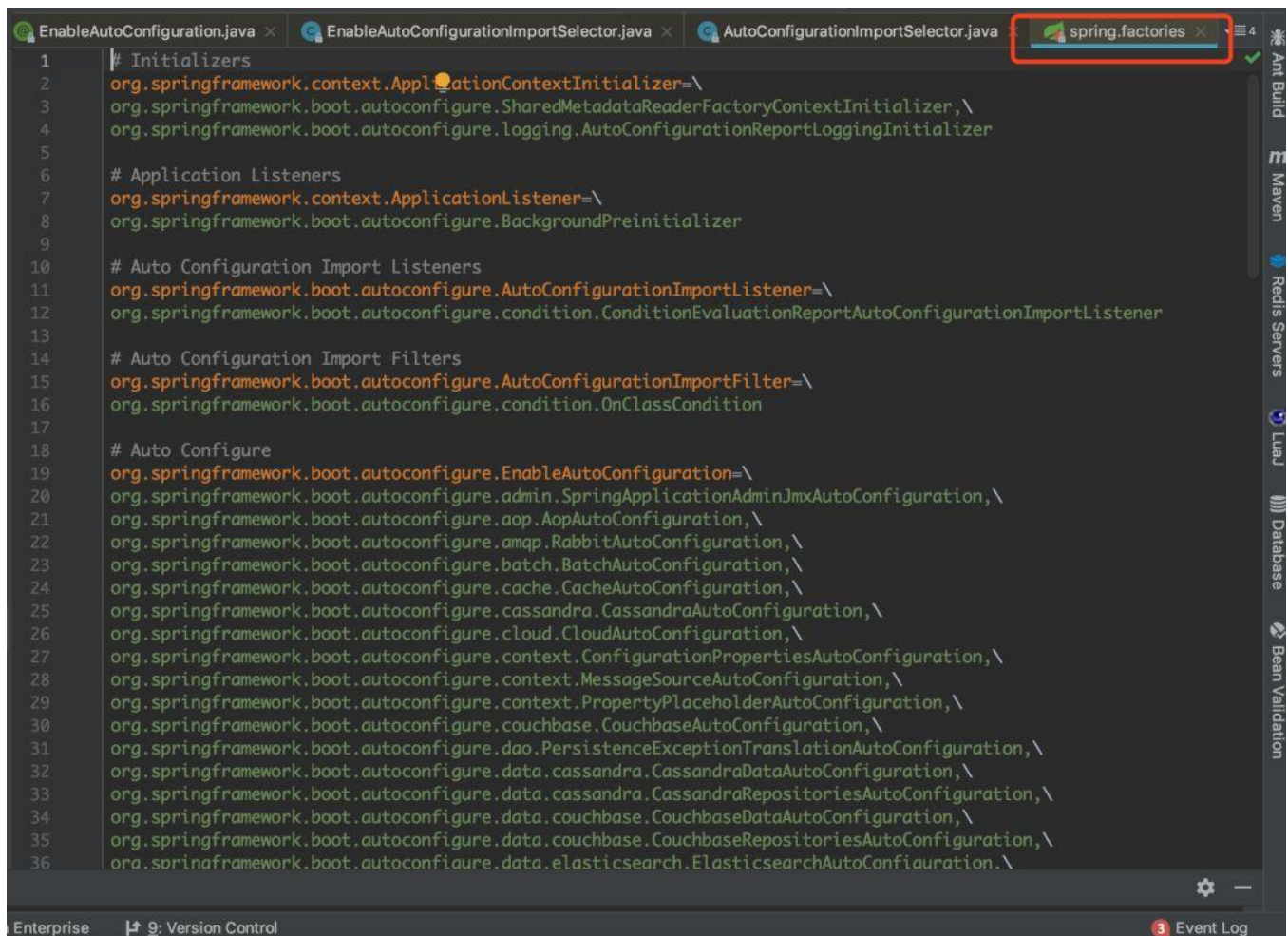
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you "
        + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}

```

这个类回去寻找的一个目录为META-INF/spring.factories，也就是说他帮你加载让你去使用也就是在这个META-INF/spring.factories目录装配的，他在哪里？



我们点进spring.factories来看



```
1 # Initializers
2 org.springframework.context.ApplicationContextInitializer=\
3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
4 org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer
5
6 # Application Listeners
7 org.springframework.context.ApplicationListener=\
8 org.springframework.boot.autoconfigure.BackgroundPreinitializer
9
10 # Auto Configuration Import Listeners
11 org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
12 org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener
13
14 # Auto Configuration Import Filters
15 org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
16 org.springframework.boot.autoconfigure.condition.OnClassCondition
17
18 # Auto Configure
19 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
20 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
21 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
22 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
23 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
24 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
25 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
26 org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
27 org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
28 org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
29 org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
30 org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
31 org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
32 org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
33 org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
34 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
35 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
36 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
```

我们可以发现帮我们配置了很多类的全路径，比如你想整合activemq，或者说Servlet



```

9  org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
10 org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
11 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
12 org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
13 org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
14 org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
15 org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
16 org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
17 org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
18 org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
19 org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
20 org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
21 org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
22 org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
23 org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
24 org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
25 org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
26 org.springframework.boot.autoconfigure ldap.embedded.EmbeddedLdapAutoConfiguration,\
27 org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
28 org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
29 org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
30 org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
31 org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\
32 org.springframework.boot.autoconfigure.mobile.DeviceDelegatingViewResolverAutoConfiguration,\
33 org.springframework.boot.autoconfigure.mobile.SitePreferenceAutoConfiguration,\
34 org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
35 org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
36 org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
37 org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
38 org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\
39 org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

```

可以看到他都已经帮我们引入了进来，我看随便拿几个来看

```

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,
org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,
org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,
org.springframework.boot.autoconfigure.security.oauth2.OAuth2AutoConfiguration,

```

比如我们经常用的security，可以看到已经帮你配置好，所以我们的EnableAutoConfiguration主要作用就是让你自动去配置，但并不是所有都是创建好的，是根据你程序去进行决定。那我们继续来看

```

@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM,
classes = TypeExcludeFilter.class), @Filter(type = FilterType.CUSTOM,
classes = AutoConfigurationExcludeFilter.class) })

```

这个注解大家应该都不陌生，扫描包，放入spring容器，那他在springboot当中做了什么策略呢？我们可以点跟烟去思考，帮我们做了一个排除策略，他在这里结合SpringBootConfiguration去使用，为什么是排除，因为不可能一上来全部加载，因为内存有限。

那么我们来总结下@SpringbootApplication：就是说，他已经把很多东西准备好，具体是否使用取决于我们的程序或者说配置，那我们到底用不用？那我们继续来看一行代码

```

public static void main(String[] args)
{
    SpringApplication.run(StartEurekaApplication.class, args);
}

```

那们来看下在执行run方法到底有没有用到哪些自动配置的东西，比如说内置的Tomcat，那我们来找找内置Tomcat，我们点进run

```

public static ConfigurableApplicationContext run(Object[] sources, String[] args) {
    return new SpringApplication(sources).run(args);
}

```

然后他调用又一个run方法，我们点进来看

```

public ConfigurableApplicationContext run(String... args) {
    //计时器
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
    //监听器
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(
            args);
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
            applicationArguments);
        Banner printedBanner = printBanner(environment);
        //准备上下文
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        //预刷新context
        prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);
        //刷新context
        refreshContext(context);
        //刷新之后的context
        afterRefresh(context, applicationArguments);
        listeners.finished(context, null);
        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopWatch);
        }
        return context;
    }
    catch (Throwable ex) {
        handleRunFailure(context, listeners, analyzers, ex);
        throw new IllegalStateException(ex);
    }
}

```

那我们关注的就是 refreshContext(context); 刷新context，我们点进来看

```

private void refreshContext(ConfigurableApplicationContext context) {
    refresh(context);
    if (this.registerShutdownHook) {
        try {
            context.registerShutdownHook();
        }
        catch (AccessControlException ex) {
            // Not allowed in some environments.
        }
    }
}

```

```
}
```

我们继续点进refresh(context);

```
protected void refresh(ApplicationContext applicationContext) {  
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);  
    ((AbstractApplicationContext) applicationContext).refresh();  
}
```

他会调用 ((AbstractApplicationContext) applicationContext).refresh();方法，我们点进来

```
public void refresh() throws BeansException, IllegalStateException {  
    synchronized (this.startupShutdownMonitor) {  
        // Prepare this context for refreshing.  
        prepareRefresh();  
  
        // Tell the subclass to refresh the internal bean factory.  
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();  
  
        // Prepare the bean factory for use in this context.  
        prepareBeanFactory(beanFactory);  
  
        try {  
            // Allows post-processing of the bean factory in context subclasses.  
            postProcessBeanFactory(beanFactory);  
  
            // Invoke factory processors registered as beans in the context.  
            invokeBeanFactoryPostProcessors(beanFactory);  
  
            // Register bean processors that intercept bean creation.  
            registerBeanPostProcessors(beanFactory);  
  
            // Initialize message source for this context.  
            initMessageSource();  
  
            // Initialize event multicaster for this context.  
            initApplicationEventMulticaster();  
  
            // Initialize other special beans in specific context subclasses.  
            onRefresh();  
  
            // Check for listener beans and register them.  
            registerListeners();  
  
            // Instantiate all remaining (non-lazy-init) singletons.  
            finishBeanFactoryInitialization(beanFactory);  
  
            // Last step: publish corresponding event.  
            finishRefresh();  
        }  
  
        catch (BeansException ex) {  
            if (logger.isWarnEnabled()) {  
                logger.warn("Exception encountered during context initialization - " +  
                    "cancelling refresh attempt: " + ex);  
            }  
  
            // Destroy already created singletons to avoid dangling resources.  
            destroyBeans();  
        }  
    }  
}
```

```

        // Reset 'active' flag.
        cancelRefresh(ex);

        // Propagate exception to caller.
        throw ex;
    }

    finally {
        // Reset common introspection caches in Spring's core, since we
        // might not ever need metadata for singleton beans anymore...
        resetCommonCaches();
    }
}
}

```

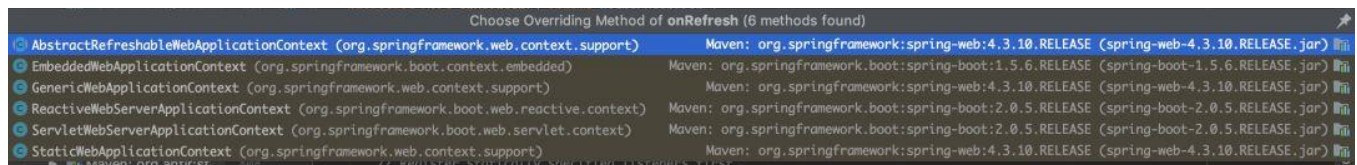
这点代码似曾相识啊 没错，就是一个spring的bean的加载过程我在，解析springIOC加载过程的时候介绍过这里面的方法，如果你看过Spring源码的话，应该知道这些方法都是做什么的。现在我们不关心其他的，我们来看一个方法叫做 onRefresh();方法

```

protected void onRefresh() throws BeansException {
    // For subclasses: do nothing by default.
}

```

他在这里并没有实现，但是我们找他的其他实现，我们来找



我们既然要找Tomcat那就肯定跟web有关，我们可以看到有个ServletWebServerApplicationContext

```

@Override
protected void onRefresh() {
    super.onRefresh();
    try {
        createWebServer();
    }
    catch (Throwable ex) {
        throw new ApplicationContextException("Unable to start web server", ex);
    }
}

```

我们可以看到有一个createWebServer();方法他是创建web容器的，而Tomcat不就是web容器，那他是怎么创建的呢，我们继续看

```

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();
}

```

factory.getWebServer(getSelfInitializer());他通过工厂的方式创建的

```

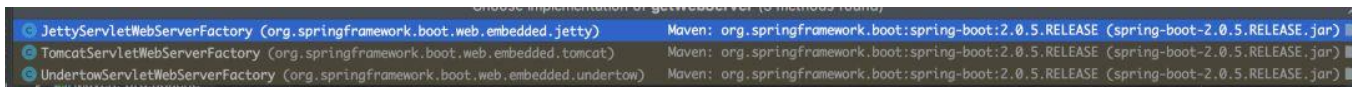
public interface ServletWebServerFactory {

    WebServer getWebServer(ServletContextInitializer... initializers);

}

```

可以看到 它是一个接口，为什么会是接口。因为我们不止是Tomcat一种web容器。



我们看到还有Jetty，那我们来看TomcatServletWebServerFactory

```

@Override
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);
}

```

那这块代码，就是我们要寻找的内置Tomcat，在这个过程当中，我们可以看到创建Tomcat的一个流程。因为run方法里面加载的东西很多，所以今天就浅谈到这里。如果不明白的话，我们在用另一种方式来理解下，

大家要应该都知道stater举点例子

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>

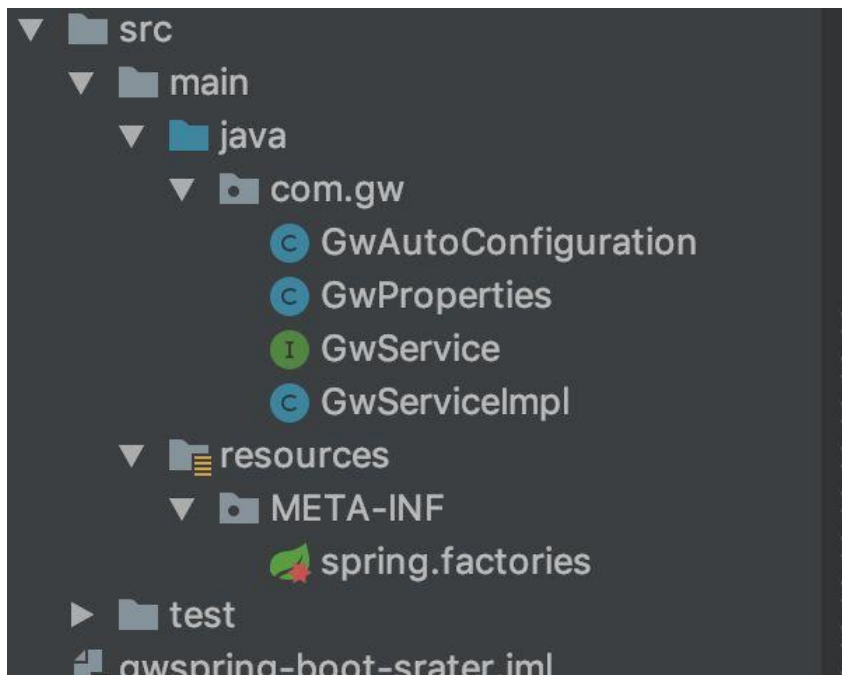
```

所以我们不妨定义一个starter来理解下，我们做一个需求，就是定制化不同的人跟大家说你们好，我们来看

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.4.RELEASE</version>
  <relativePath/>
</parent>
<groupId>com.zgw</groupId>
<artifactId>gw-spring-boot-srater</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
  </dependency>
</dependencies>
```

我们先来看maven配置写入版本号，如果自定义一个starter的话必须依赖spring-boot-autoconfigure这个包,我们先看下项目目录



```
public class GwServiceImpl implements GwService{
    @Autowired
    GwProperties properties;

    @Override
    public void Hello()
    {
        String name=properties.getName();
        System.out.println(name+"说:你们好啊");
    }
}
```

我们做的就是通过配置文件来定制name这个具体实现

```

@Component
@ConfigurationProperties(prefix = "spring.gwname")
public class GwProperties {

    String name="zgw";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

这个类可以通过@ConfigurationProperties读取配置文件

```

@Configuration
@ConditionalOnClass(GwService.class) //扫描类
@EnableConfigurationProperties(GwProperties.class) //让配置类生效
public class GwAutoConfiguration {

    /**
     * 功能描述 托管给spring
     * @author zgw
     * @return
     */
    @Bean
    @ConditionalOnMissingBean
    public GwService gwService()
    {
        return new GwServiceImpl();
    }
}

```

这个为配置类，为什么这么写因为，spring-boot的stater都是这么写的，我们可以参照他仿写stater，以达到自动配置的目的，然后我们在通过spring.factories也来进行配置

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.gw.GwAutoConfiguration
```

然后这样一个简单的stater就完成了，然后可以进行maven的打包，在其他项目引入就可以使用，在这里列出代码地址

<https://github.com/zgw1469039806/gwspringbootstater>

扫码关注后端技术精选，回复“学习资料”，领取100套小程序源码+小程序开发视频和基本Java经典书籍电子版



csdn.net/weixin\_66886616



后端技术精选



Spring

MySQL

Spring Boot

欢迎微信搜索【后端技术精选】关注我的公众号，号内回复“后端面试”，送你一份精心准备的Java面试题（提纲+解析），后端技术精选每天定时推送优质Java技术博客，可以琐碎时间学点儿东西