第三节:springboot源码解析(王炸篇)

今天内容

1:spring注解 热身

2:springboot 自动装配原理

3:springboot启动原理(jar 包启动)

4:springboot启动原理[redacted]

5:作业:springboot的自定义启动器

一:spring注解之如何导入Bean的几种方式

1）@Bean注解，**不做讲解**

2）包扫描来加载Bean 比如标识@Controller @Service @Repository @Compent 不做讲解

3) @Import几种取值来注册bean

①：实现ImportSelector接口的类

②：实现ImportBeanDefinitionRegistrar接口來注冊bean

4）实现factoryBean的方式来导入组件 (不做讲解)

1.1)通过@Import注解来导入ImportSelector组件

①:写一个配置类在配置类上标注一个@Import的注解，

```
@Configuration
@Import(value = {TulingSelector.class})
public class TulingConfig {
}
```

②：在@Import注解的value值 写自己需要导入的组件

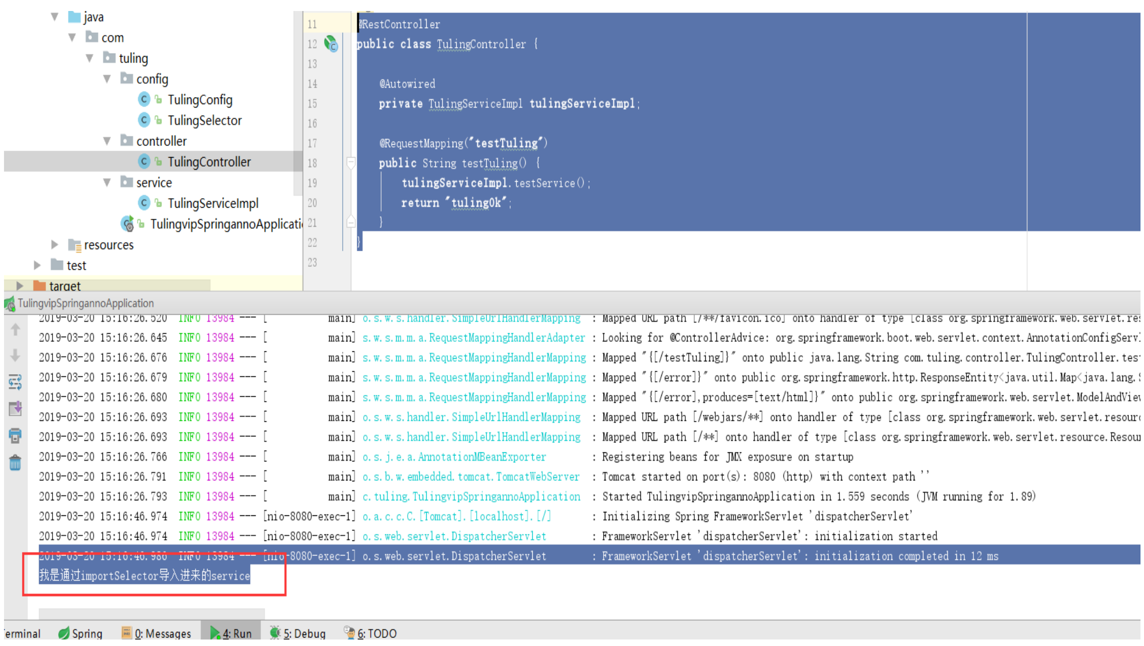
在selectImports方法中 就是你需要导入组件的全类名

```
public class TulingSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]{"com.tuling.service.TulingServiceImpl"};
    }
}
```

核心代码:

```
@RestController
```

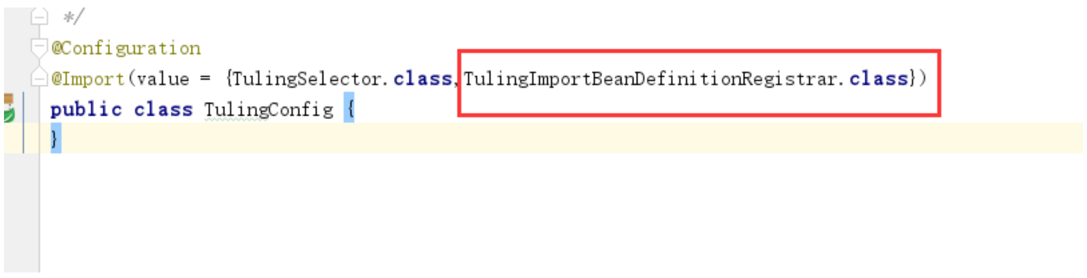
```
public class TulingController {

    //自动注入 tulingServiceImpl
    @Autowired
    private TulingServiceImpl tulingServiceImpl;

    @RequestMapping("testTuling")
    public String testTuling() {
        tulingServiceImpl.testService();
        return "tulingOk";
    }
}

这里是没有标注其他注解提供给spring包扫描的
public class TulingServiceImpl {

    public void testService() {
        System.out.println("我是通过importSelector导入进来的service");
    }
}
```



```
@RestController
public class TulingController {

    @Autowired
    private TulingServiceImpl tulingServiceImpl;

    @RequestMapping("testTuling")
    public String testTuling() {
        tulingServiceImpl.testService();
        return "tulingOk";
    }
}
```

```
2019-03-20 15:16:26.520  INFO 13984 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.res
2019-03-20 15:16:26.645  INFO 13984 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.context.AnnotationConfigServ
2019-03-20 15:16:26.676  INFO 13984 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/testTuling]}" onto public java.lang.String com.tuling.controller.TulingController.tes
2019-03-20 15:16:26.679  INFO 13984 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.S
2019-03-20 15:16:26.680  INFO 13984 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView
2019-03-20 15:16:26.693  INFO 13984 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resour
2019-03-20 15:16:26.693  INFO 13984 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.Resou
2019-03-20 15:16:26.766  INFO 13984 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2019-03-20 15:16:26.791  INFO 13984 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-03-20 15:16:26.793  INFO 13984 --- [           main] c.tuling.TulingvipSpringannoApplication  : Started TulingvipSpringannoApplication in 1.559 seconds (JVM running for 1.89)
2019-03-20 15:16:46.974  INFO 13984 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2019-03-20 15:16:46.974  INFO 13984 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2019-03-20 15:16:46.986  INFO 13984 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 12 ms
我是通过importSelector导入进来的service
```

1.2）通过@Import导入ImportBeanDefinitionRegistrar 从而进来导入组件

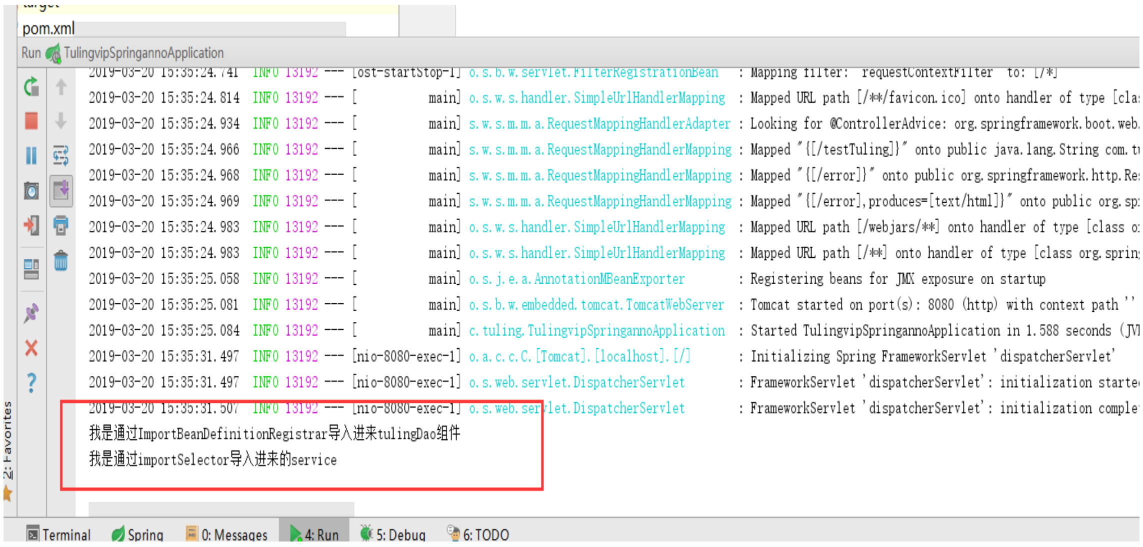


```
*/
@Configuration
@Import(value = {TulingSelector.class, TulingImportBeanDefinitionRegistrar.class})
public class TulingConfig {
}
```

核心代码:

```
public class TulingImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata annotationMetadata, BeanDefinitionRegistry beanDefinitionReg
        //定义一个BeanDefinition
        RootBeanDefinition rootBeanDefinition = new RootBeanDefinition(TulingDao.class);
        //把自定义的bean定义导入到容器中
        beanDefinitionRegistry.registerBeanDefinition("tulingDao",rootBeanDefinition);
    }
}

通过ImportSelector功能导入进来的
public class TulingServiceImpl {

    @Autowired
    private TulingDao tulingDao;

    public void testService() {
        tulingDao.testTulingDao();
        System.out.println("我是通过importSelector导入进来的service");
    }
}

通过ImportBeanDefinitionRegistar导入进来的
public class TulingDao {

    public void testTulingDao() {
        System.out.println("我是通过ImportBeanDefinitionRegistrar导入进来tulingDao组件");
    }
}
```

测试结果:



```
2019-03-20 15:35:24.741  INFO 13192 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2019-03-20 15:35:24.814  INFO 13192 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [cla
2019-03-20 15:35:24.934  INFO 13192 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.
2019-03-20 15:35:24.966  INFO 13192 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/testTuling]}" onto public java.lang.String com.t
2019-03-20 15:35:24.968  INFO 13192 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.Re
2019-03-20 15:35:24.969  INFO 13192 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.sp
2019-03-20 15:35:24.983  INFO 13192 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class o
2019-03-20 15:35:24.983  INFO 13192 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.sprin
2019-03-20 15:35:25.058  INFO 13192 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2019-03-20 15:35:25.081  INFO 13192 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-03-20 15:35:25.084  INFO 13192 --- [           main] c.tuling.TulingvipSpringannoApplication  : Started TulingvipSpringannoApplication in 1.588 seconds (JV
2019-03-20 15:35:31.497  INFO 13192 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2019-03-20 15:35:31.497  INFO 13192 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization starte
2019-03-20 15:35:31.507  INFO 13192 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization comple
我是通过ImportBeanDefinitionRegistrar导入进来tulingDao组件
我是通过importSelector导入进来的service
```

1.3)spring底层条件装配的原理@Conditional

应用要求:**比如我有二个组件,一个是**TulingLog **一个是**TulingAspect

而TulingLog **是依赖**TulingAspect**的 只有容器中有**TulingAspect**组件才会加载**TulingLog

```
tulingLog组件  依赖TulingAspect组件
public class TulingLog {
}

tulingAspect组件
public class TulingAspect {
}
```

①:自定义条件组件条件

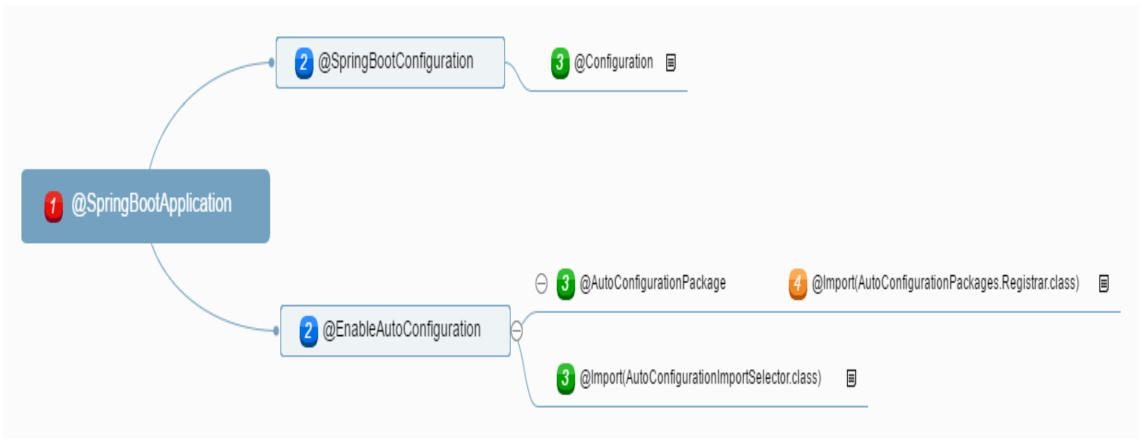
```
public class TulingConditional implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
        //容器中包含tulingAspect组件才返回Ture
        if(conditionContext.getBeanFactory().containsBean("tulingAspect")){
            return true;
        }else{
            return false;
        }

    }
}

    -----------------------------------该情况下会加载二个组件----------------------------------------

    @Bean
    public TulingAspect tulingAspect() {
        System.out.println("TulingAspect组件自动装配到容器中");
        return new TulingAspect();
    }


    @Bean
    @Conditional(value = TulingConditional.class)
    public TulingLog tulingLog() {
        System.out.println("TulingLog组件自动装配到容器中");
        return new TulingLog();
    }

    -----------------------------------二个组件都不会被加载------------------------------------
    /*@Bean**/
    public TulingAspect tulingAspect() {
        System.out.println("TulingAspect组件自动装配到容器中");
        return new TulingAspect();
    }


    @Bean
    @Conditional(value = TulingConditional.class)
    public TulingLog tulingLog() {
        System.out.println("TulingLog组件自动装配到容器中");
        return new TulingLog();
    }
```

==================================**到此结束spring自层注解**

==============================

二:springboot自动装配原理

2.1)@Springboot注解组合图


1 @SpringBootApplication
2 @SpringBootConfiguration
3 @Configuration
2 @EnableAutoConfiguration
3 @AutoConfigurationPackage
4 @Import(AutoConfigurationPackages.Registrar.class)
3 @Import(AutoConfigurationImportSelector.class)

根据上面的@SpringBootApplication注解 我们来着重分析如下二个类

①：AutoConfigurationImportSelector.class

②：AutoConfigurationPackages.Registrar.class

先分析AutoConfigurationImportSelector为我们干了什么活？？

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
	if (!isEnabled(annotationMetadata)) {
		return NO_IMPORTS;
	}
	AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
			.loadMetadata(this.beanClassLoader);
	AnnotationAttributes attributes = getAttributes(annotationMetadata);
	//获取候选的配置类
	List<String> configurations = getCandidateConfigurations(annotationMetadata,
			attributes);
	//移除重复的
	configurations = removeDuplicates(configurations);
	Set<String> exclusions = getExclusions(annotationMetadata, attributes);
	checkExcludedClasses(configurations, exclusions);
	configurations.removeAll(exclusions);
	configurations = filter(configurations, autoConfigurationMetadata);
	fireAutoConfigurationImportEvents(configurations, exclusions);
	//返回出去
	return StringUtils.toStringArray(configurations);
}

//获取候选的配置类
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;
}

//加载配置类
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
	String factoryClassName = factoryClass.getName();
	return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}
```

```
    private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
        MultiValueMap<String, String> result = cache.get(classLoader);
        if (result != null) {
            return result;
        }

        try {

          //"META-INF/spring.factories"  去类路径下该文件中加载 EnableAutoConfiguration.class
            Enumeration<URL> urls = (classLoader != null ?
                    classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                    ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
            result = new LinkedMultiValueMap<>();
            //遍历解析出来的集合
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                UrlResource resource = new UrlResource(url);
                //放在Properties中
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                for (Map.Entry<?, ?> entry : properties.entrySet()) {
                    String factoryClassName = ((String) entry.getKey()).trim();
                    for (String factoryName : StringUtils.commaDelimitedListToStringArray((String) entry.getValue())
                        result.add(factoryClassName, factoryName.trim());
                    }
                }
            }
            cache.put(classLoader, result);
            //返回
            return result;
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load factories from location [" +
                    FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
    }
```

主要是扫描spring-boot-autoconfigure\2.0.8.RELEASE\spring-boot-autoconfigure-2.0.8.RELEASE.jar!\META-INF\spring.factories 中EnableAutoConfiguration对应的全类名

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityRequestMatcherProviderAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
```

上面的一些个 XXXAutoConfiguration**都是一个个自动配置类**

我们就拿二个来分析一下 这些自动配置类是如何工作的???

分析源码1：org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration

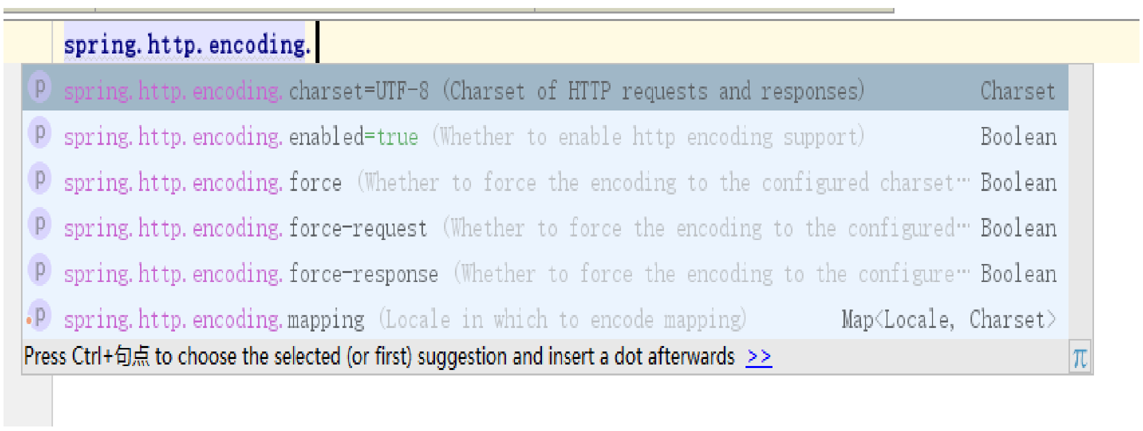
```
@Configuration   //标识是一个自动配置类
@EnableConfigurationProperties(HttpEncodingProperties.class) 启动指定类的配置功能，并且把配置文件中的属性和HttpEncodi
@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET) //spring底层的@Conditional注解的变
@ConditionalOnClass(CharacterEncodingFilter.class) ，判断环境中是否没有这个类

判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的
//即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的
@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true)
public class HttpEncodingAutoConfiguration {

  自动配置类的属性映射
    private final HttpEncodingProperties properties;

    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
        this.properties = properties;
    }

  //配置一个 CharacterEncodingFilter 是springmvc解决乱码的 ，若容器中没有该组件 ，那么就会创建该组件
    @Bean
    @ConditionalOnMissingBean
    public CharacterEncodingFilter characterEncodingFilter() {
        CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
```

```
            filter.setEncoding(this.properties.getCharset().name());
            filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));
            filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
            return filter;
        }

        @Bean
        public LocaleCharsetMappingsCustomizer localeCharsetMappingsCustomizer() {
            return new LocaleCharsetMappingsCustomizer(this.properties);
        }

        private static class LocaleCharsetMappingsCustomizer implements
                WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>, Ordered {

            private final HttpEncodingProperties properties;

            LocaleCharsetMappingsCustomizer(HttpEncodingProperties properties) {
                this.properties = properties;
            }

            @Override
            public void customize(ConfigurableServletWebServerFactory factory) {
                if (this.properties.getMapping() != null) {
                    factory.setLocaleCharsetMappings(this.properties.getMapping());
                }
            }

            @Override
            public int getOrder() {
                return 0;
            }

        }

}
```

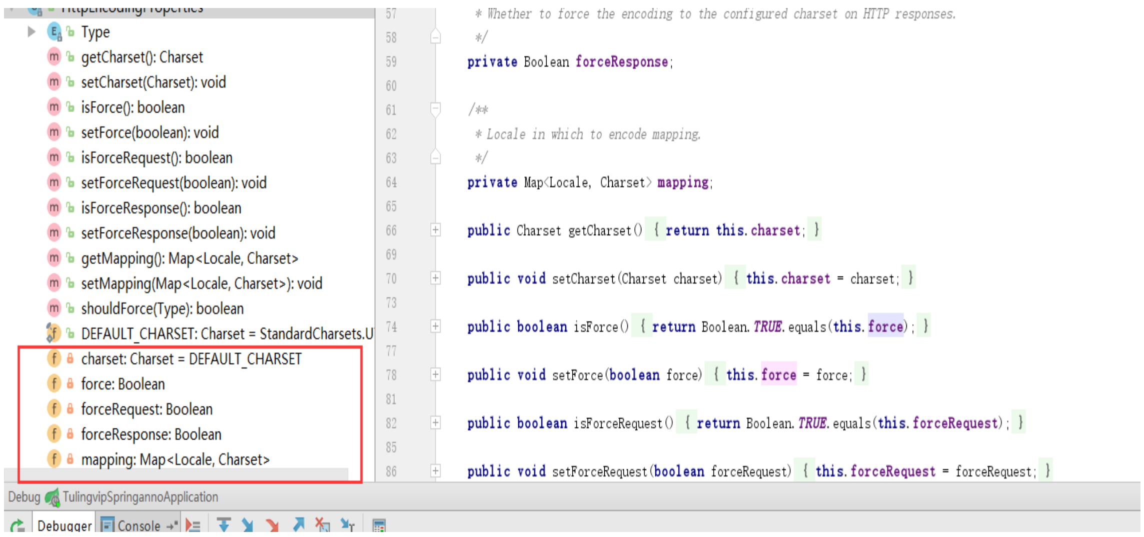
我们来看下 HttpEncodingProperties**，这个类是用来什么的？ 就是我们**yml**中能配置什么类，在这个类中都会有一个属性一一对应**

```
@ConfigurationProperties(prefix = "spring.http.encoding") //从配置文件中获取指定的值和bean的属
性进行绑定
public class HttpEncodingProperties {
public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");
```

我们对应的配置文件(yml)中就会有对应属性来配置



```
spring.http.encoding.
spring.http.encoding.charset=UTF-8 (Charset of HTTP requests and responses)    Charset
spring.http.encoding.enabled=true (Whether to enable http encoding support)    Boolean
spring.http.encoding.force (Whether to force the encoding to the configured charset… Boolean
spring.http.encoding.force-request (Whether to force the encoding to the configured… Boolean
spring.http.encoding.force-response (Whether to force the encoding to the configure… Boolean
spring.http.encoding.mapping (Locale in which to encode mapping)    Map<Locale, Charset>
Press Ctrl+句点 to choose the selected (or first) suggestion and insert a dot afterwards >>
```



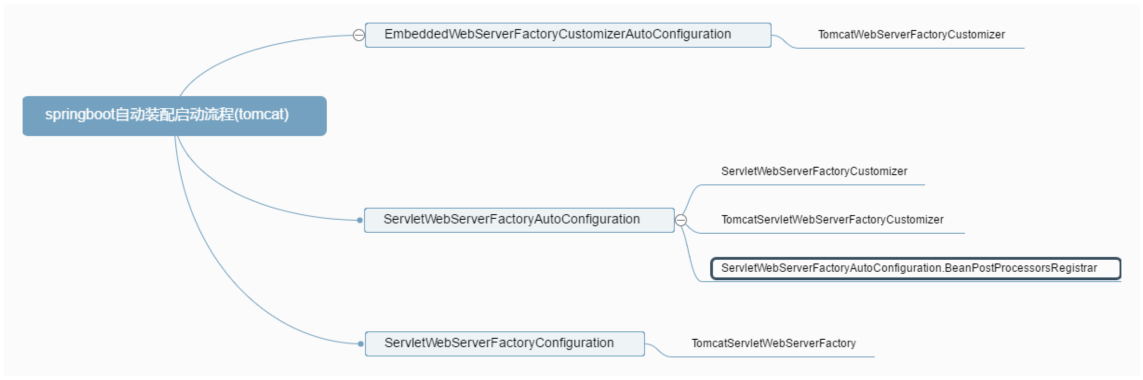
```
Type
getCharset(): Charset
setCharset(Charset): void
isForce(): boolean
setForce(boolean): void
isForceRequest(): boolean
setForceRequest(boolean): void
isForceResponse(): boolean
setForceResponse(boolean): void
getMapping(): Map<Locale, Charset>
setMapping(Map<Locale, Charset>): void
shouldForce(Type): boolean
DEFAULT_CHARSET: Charset = StandardCharsets.U
charset: Charset = DEFAULT_CHARSET
force: Boolean
forceRequest: Boolean
forceResponse: Boolean
mapping: Map<Locale, Charset>
```

```java
 * Whether to force the encoding to the configured charset on HTTP responses.
 */
private Boolean forceResponse;

/**
 * Locale in which to encode mapping.
 */
private Map<Locale, Charset> mapping;

public Charset getCharset() { return this.charset; }

public void setCharset(Charset charset) { this.charset = charset; }

public boolean isForce() { return Boolean.TRUE.equals(this.force); }

public void setForce(boolean force) { this.force = force; }

public boolean isForceRequest() { return Boolean.TRUE.equals(this.forceRequest); }

public void setForceRequest(boolean forceRequest) { this.forceRequest = forceRequest; }
```

以上 就是 AutoConfigurationImportSelector**为我们容器中注册了那些组件，然后根据**maven**依赖导入的**jar**包，根据条件装配来指定哪些组件**

起作用 哪些组件不起作用。

三:上面我们分析了springboot 自动装配原理，接下来我们依靠 自动装配原理来分析出spring Boot的jar包的启动流程.


springboot自动装配启动流程(tomcat)
EmbeddedWebServerFactoryCustomizerAutoConfiguration
TomcatWebServerFactoryCustomizer
ServletWebServerFactoryAutoConfiguration
ServletWebServerFactoryCustomizer
TomcatServletWebServerFactoryCustomizer
ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar
ServletWebServerFactoryConfiguration
TomcatServletWebServerFactory

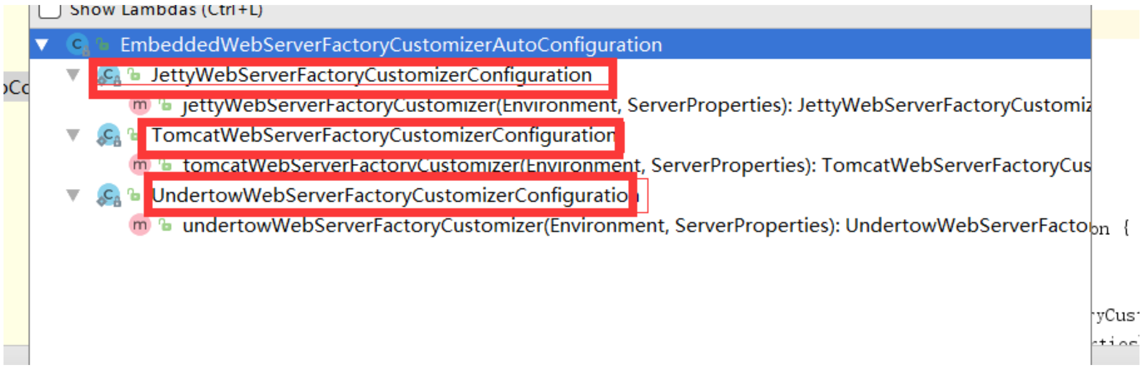
3.1) **我们先来看**springboot **怎么来自动装配**tomcat **相关的组件**

EmbeddedWebServerFactoryCustomizerAutoConfiguration(**内嵌**web**容器工厂自定义定制器装配类**)?

疑问1？： 定制器是用来干什么的?

疑问2？：定制器何时工作?

类的继承关系


Show Lambdas (Ctrl+L)

EmbeddedWebServerFactoryCustomizerAutoConfiguration
JettyWebServerFactoryCustomizerConfiguration
jettyWebServerFactoryCustomizer(Environment, ServerProperties): JettyWebServerFactoryCustomiz
TomcatWebServerFactoryCustomizerConfiguration
tomcatWebServerFactoryCustomizer(Environment, ServerProperties): TomcatWebServerFactoryCus
UndertowWebServerFactoryCustomizerConfiguration
undertowWebServerFactoryCustomizer(Environment, ServerProperties): UndertowWebServerFacto

我们就以tomcat 作为内嵌容器来分析

```
@Configuration
@ConditionalOnWebApplication
@EnableConfigurationProperties(ServerProperties.class)
public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {

    //配置tomcat的
        @Configuration
        @ConditionalOnClass({ Tomcat.class, UpgradeProtocol.class })
        public static class TomcatWebServerFactoryCustomizerConfiguration {

                @Bean
                public TomcatWebServerFactoryCustomizer tomcatWebServerFactoryCustomizer(
                                Environment environment, ServerProperties serverProperties) {
                        return new TomcatWebServerFactoryCustomizer(environment, serverProperties);
                }

        }

    //配置jetty
        @Configuration
        @ConditionalOnClass({ Server.class, Loader.class, WebAppContext.class })
        public static class JettyWebServerFactoryCustomizerConfiguration {

                @Bean
                public JettyWebServerFactoryCustomizer jettyWebServerFactoryCustomizer(
                                Environment environment, ServerProperties serverProperties) {
                        return new JettyWebServerFactoryCustomizer(environment, serverProperties);
                }

        }

    配置undertow的
        @Configuration
        @ConditionalOnClass({ Undertow.class, SslClientAuthMode.class })
        public static class UndertowWebServerFactoryCustomizerConfiguration {

                @Bean
                public UndertowWebServerFactoryCustomizer undertowWebServerFactoryCustomizer(
                                Environment environment, ServerProperties serverProperties) {
                        return new UndertowWebServerFactoryCustomizer(environment, serverProperties);
                }

        }

}
```

我们来看下tomat 工厂定制器 是用来修改设置容器的内容的(把serverProperties的属性设置到tomcat的创建工厂中)

```
public class TomcatWebServerFactoryCustomizer implements WebServerFactoryCustomizer<ConfigurableTomcatWebServ

  ……………………….其他代码省略。。。。。。。。。。。。。。

    @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);
        propertyMapper.from(tomcatProperties::getBackgroundProcessorDelay).whenNonNull()
                .as(Duration::getSeconds).as(Long::intValue)
                .to(factory::setBackgroundProcessorDelay);
        customizeRemoteIpValve(factory);
        propertyMapper.from(tomcatProperties::getMaxThreads).when(this::isPositive)
                .to((maxThreads) -> customizeMaxThreads(factory,
                        tomcatProperties.getMaxThreads()));
        propertyMapper.from(tomcatProperties::getMinSpareThreads).when(this::isPositive)
                .to((minSpareThreads) -> customizeMinThreads(factory, minSpareThreads));
        propertyMapper.from(() -> determineMaxHttpHeaderSize()).when(this::isPositive)
                .to((maxHttpHeaderSize) -> customizeMaxHttpHeaderSize(factory,
                        maxHttpHeaderSize));
        propertyMapper.from(tomcatProperties::getMaxHttpPostSize)
                .when((maxHttpPostSize) -> maxHttpPostSize != 0)
                .to((maxHttpPostSize) -> customizeMaxHttpPostSize(factory,
                        maxHttpPostSize));
        propertyMapper.from(tomcatProperties::getAccesslog)
                .when(ServerProperties.Tomcat.Accesslog::isEnabled)
                .to((enabled) -> customizeAccessLog(factory));
        propertyMapper.from(tomcatProperties::getUriEncoding).whenNonNull()
                .to(factory::setUriEncoding);
        propertyMapper.from(properties::getConnectionTimeout).whenNonNull()
                .to((connectionTimeout) -> customizeConnectionTimeout(factory,
                        connectionTimeout));
        propertyMapper.from(tomcatProperties::getMaxConnections).when(this::isPositive)
                .to((maxConnections) -> customizeMaxConnections(factory, maxConnections));
        propertyMapper.from(tomcatProperties::getAcceptCount).when(this::isPositive)
                .to((acceptCount) -> customizeAcceptCount(factory, acceptCount));
        customizeStaticResources(factory);
        customizeErrorReportValve(properties.getError(), factory);
    }
```

ServletWebServerFactoryAutoConfiguration Servletweb**工厂自动配置类**

很重要********************:@Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class}******************

```
@Configuration
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(ServerProperties.class)
@Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
        ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
        ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
        ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
        @Bean
        public ServletWebServerFactoryCustomizer servletWebServerFactoryCustomizer(
                        ServerProperties serverProperties) {
                return new ServletWebServerFactoryCustomizer(serverProperties);
        }

        @Bean
        @ConditionalOnClass(name = "org.apache.catalina.startup.Tomcat")
        public TomcatServletWebServerFactoryCustomizer tomcatServletWebServerFactoryCustomizer(
                        ServerProperties serverProperties) {
                return new TomcatServletWebServerFactoryCustomizer(serverProperties);
        }

}



.....................................................ServletWebServerFactoryCustomizer核心代码..........................................
        public void customize(ConfigurableServletWebServerFactory factory) {
                PropertyMapper map = PropertyMapper.get().alwaysApplyingWhenNonNull();
                map.from(this.serverProperties::getPort).to(factory::setPort);
                map.from(this.serverProperties::getAddress).to(factory::setAddress);
                map.from(this.serverProperties.getServlet()::getContextPath)
                                .to(factory::setContextPath);
                map.from(this.serverProperties.getServlet()::getApplicationDisplayName)
                                .to(factory::setDisplayName);
                map.from(this.serverProperties.getServlet()::getSession).to(factory::setSession);
                map.from(this.serverProperties::getSsl).to(factory::setSsl);
                map.from(this.serverProperties.getServlet()::getJsp).to(factory::setJsp);
                map.from(this.serverProperties::getCompression).to(factory::setCompression);
                map.from(this.serverProperties::getHttp2).to(factory::setHttp2);
                map.from(this.serverProperties::getServerHeader).to(factory::setServerHeader);
                map.from(this.serverProperties.getServlet()::getContextParameters)
                                .to(factory::setInitParameters);
        }



--------------------------------------------------TomcatServletWebServerFactoryCustomizer核心定制代码-----------
public void customize(TomcatServletWebServerFactory factory) {
                ServerProperties.Tomcat tomcatProperties = this.serverProperties.getTomcat();
                if (!ObjectUtils.isEmpty(tomcatProperties.getAdditionalTldSkipPatterns())) {
                        factory.getTldSkipPatterns()
                                        .addAll(tomcatProperties.getAdditionalTldSkipPatterns());
                }
                if (tomcatProperties.getRedirectContextRoot() != null) {
                        customizeRedirectContextRoot(factory,
                                        tomcatProperties.getRedirectContextRoot());
                }
                if (tomcatProperties.getUseRelativeRedirects() != null) {
                        customizeUseRelativeRedirects(factory,
                                        tomcatProperties.getUseRelativeRedirects());
                }
        }
```

ServletWebServerFactoryConfiguration **容器工厂配置类**

```
@Configuration
class ServletWebServerFactoryConfiguration {

        @Configuration
```

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

//配置tomcat 容器工厂
    @Bean
    public TomcatServletWebServerFactory tomcatServletWebServerFactory() {
        return new TomcatServletWebServerFactory();
    }

}
```

现在我们来分析一下启动流程。。。。。。。。。。。。。。。。。。。。。

1)com.tuling.TulingvipSpringbootAutoconfigPrincipleApplication#main **运行main方法**

2)org.springframework.boot.SpringApplication#run(java.lang.Class<?>, java.lang.String...)

2.1）**传入主配置类，以及命令行参数**

2.2) **创建**SpringApplication**对象**

①:**保存主配置类**

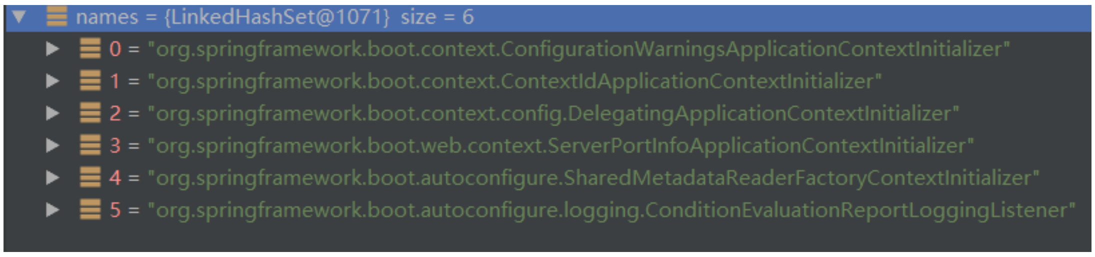
②：**保存**web**应用的配置类型**

③：**去**mate-info/spring.factories**文件中获取** ApplicationContextInitializer(**容器初始化器**) **保存到**springapplication**对象中**

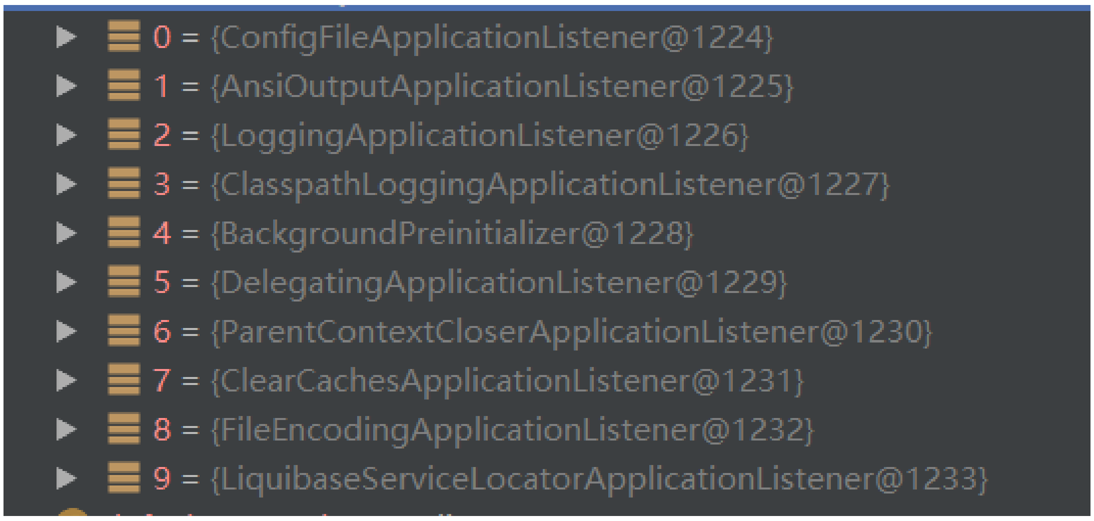
④：**去**mate-info/spring.factories**文件中获取** ApplicationListener(**容器监听器**) **保存到**springapplication**对象中**

⑤：**保存选取 主配置类**

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //保存主配置类
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    //保存web应用的类型
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    //保存 容器初始化器(ApplicationContextInitializer类型的)
    setInitializers((Collection) getSpringFactoriesInstances(
            ApplicationContextInitializer.class));
    //把监听器保存到 SpringApplication中[ApplicationListener]
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    //保存主配置类
    this.mainApplicationClass = deduceMainApplicationClass();
}

//还是去META-INFO/spring.factories 中获取ApplicationContextInitializer 类型，用于初始化容器
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
        Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>(
            SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
            classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}

//查找主配置类 查询的依据就是看哪个方法是否有main方法
private Class<?> deduceMainApplicationClass() {
    try {
        StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
        for (StackTraceElement stackTraceElement : stackTrace) {
            if ("main".equals(stackTraceElement.getMethodName())) {
                return Class.forName(stackTraceElement.getClassName());
            }
        }
    }
    catch (ClassNotFoundException ex) {
        // Swallow and continue
    }
    return null;
}
```



```
names = {LinkedHashSet@1071}  size = 6
    0 = "org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer"
    1 = "org.springframework.boot.context.ContextIdApplicationContextInitializer"
    2 = "org.springframework.boot.context.config.DelegatingApplicationContextInitializer"
    3 = "org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer"
    4 = "org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer"
    5 = "org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener"
```



```
    0 = {ConfigFileApplicationListener@1224}
    1 = {AnsiOutputApplicationListener@1225}
    2 = {LoggingApplicationListener@1226}
    3 = {ClasspathLoggingApplicationListener@1227}
    4 = {BackgroundPreinitializer@1228}
    5 = {DelegatingApplicationListener@1229}
    6 = {ParentContextCloserApplicationListener@1230}
    7 = {ClearCachesApplicationListener@1231}
    8 = {FileEncodingApplicationListener@1232}
    9 = {LiquibaseServiceLocatorApplicationListener@1233}
```

3)运行SpringbootApplication的run方法

```
public ConfigurableApplicationContext run(String... args) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        //创建一个 容器对象
        ConfigurableApplicationContext context = null;
        Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
        configureHeadlessProperty();
        //去meta-info/spring.factories中获取SpringApplicationRunListener 监听器(事件发布监听器)
        SpringApplicationRunListeners listeners = getRunListeners(args);
        //发布容器 starting事件(通过spring的事件多播器)
        listeners.starting();
        try {
          //封装命令行参数
            ApplicationArguments applicationArguments = new DefaultApplicationArguments(
                    args);
            //准备容器环境
            1:获取或者创建环境
            2：把命令行参数设置到环境中
            3：通过监听器发布环境准备事件
                ConfigurableEnvironment environment = prepareEnvironment(listeners,
                        applicationArguments);
                configureIgnoreBeanInfo(environment);
            //打印springboot的图标
                Banner printedBanner = printBanner(environment);
                //创建容器 根据webApplicationType 来创建容器 通过反射创建
                context = createApplicationContext();
                //去meta-info类中 获取异常报告
                exceptionReporters = getSpringFactoriesInstances(
                        SpringBootExceptionReporter.class,
                        new Class[] { ConfigurableApplicationContext.class }, context);
                //准备环境
                1：把环境设置到容器中
                2: 循环调用AppplicationInitnazlier 进行容器初始化工作
                3:发布容器上下文准备完成事件
                4:注册关于springboot特性的相关单例Bean
                5:发布容器上下文加载完毕事件
                prepareContext(context, environment, listeners, applicationArguments,printedBanner);
                refreshContext(context);
                //运行 ApplicationRunner  和CommandLineRunner
                afterRefresh(context, applicationArguments);
                stopWatch.stop();
                if (this.logStartupInfo) {
                    new StartupInfoLogger(this.mainApplicationClass)
                            .logStarted(getApplicationLog(), stopWatch);
                }
                //发布容器启动事件
                listeners.started(context);
                //运行 ApplicationRunner  和CommandLineRunner
                callRunners(context, applicationArguments);
        }
        catch (Throwable ex) {
            //出现异常；调用异常分析保护类进行分析
                handleRunFailure(context, ex, exceptionReporters, listeners);
                throw new IllegalStateException(ex);
        }

        try {
            //发布容器运行事件
                listeners.running(context);
        }
        catch (Throwable ex) {
                handleRunFailure(context, ex, exceptionReporters, null);
                throw new IllegalStateException(ex);
        }
        return context;
    }
```

5)org.springframework.boot.SpringApplication#refreshContext

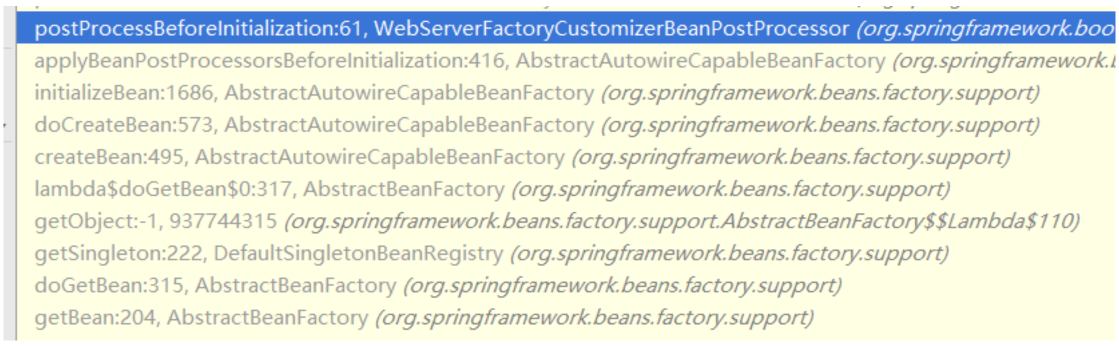
6)org.springframework.context.support.AbstractApplicationContext#refresh

7)org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext#onRefresh

8)org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext#createWebServer

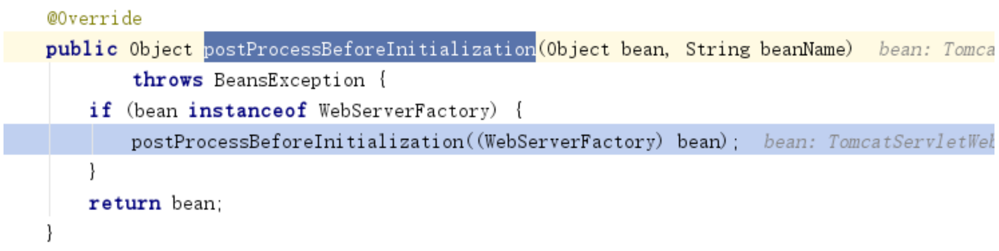
8.1)org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext#getWebServer 获取web 服务器工厂

以下是springioc容器启动的核心流程，在这里不做详细解释,大概步骤为如下:

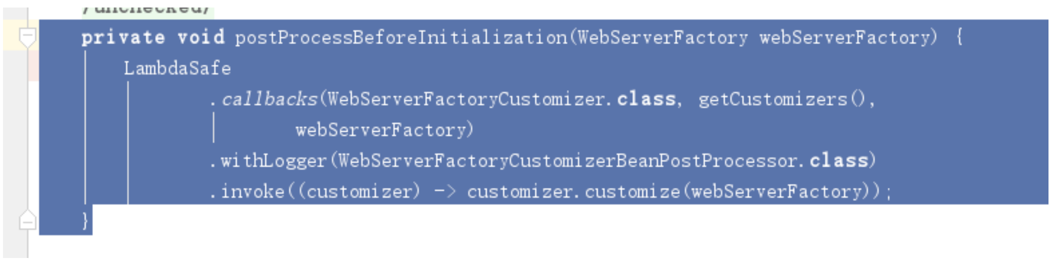

```
postProcessBeforeInitialization:61, WebServerFactoryCustomizerBeanPostProcessor (org.springframework.boo
applyBeanPostProcessorsBeforeInitialization:416, AbstractAutowireCapableBeanFactory (org.springframework.
initializeBean:1686, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
doCreateBean:573, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
createBean:495, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
lambda$doGetBean$0:317, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1, 937744315 (org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$110)
getSingleton:222, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:315, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:204, AbstractBeanFactory (org.springframework.beans.factory.support)
```

....

8.2)org.springframework.boot.web.server.WebServerFactoryCustomizerBeanPostProcessor#postProces


```java
@Override
public Object postProcessBeforeInitialization(Object bean, String beanName)   bean: Tomca
        throws BeansException {
    if (bean instanceof WebServerFactory) {
        postProcessBeforeInitialization((WebServerFactory) bean);   bean: TomcatServletWel
    }
    return bean;
}
```

8.3)org.springframework.boot.web.server.WebServerFactoryCustomizerBeanPostProcessor#postProcessBeforeInitializati


```java
private void postProcessBeforeInitialization(WebServerFactory webServerFactory) {
    LambdaSafe
            .callbacks(WebServerFactoryCustomizer.class, getCustomizers(),
                    webServerFactory)
            .withLogger(WebServerFactoryCustomizerBeanPostProcessor.class)
            .invoke((customizer) -> customizer.customize(webServerFactory));
}
```

8.3.1)WebServerFactoryCustomizerBeanPostProcessor **是一个什么东西? 在哪里注册到容器中的???**

我们往容器中导入了 BeanPostProcessorsRegistrar **他实现了** ImportBeanDefinitionRegistrar

在他的 registerBeanDefinitions**注册**Bean**定义的时候 注册了** webServerFactoryCustomizerBeanPostProcessor

想知道 webServerFactoryCustomizerBeanPostProcessor**何时在容器中注册的么? ? ? ? ?**

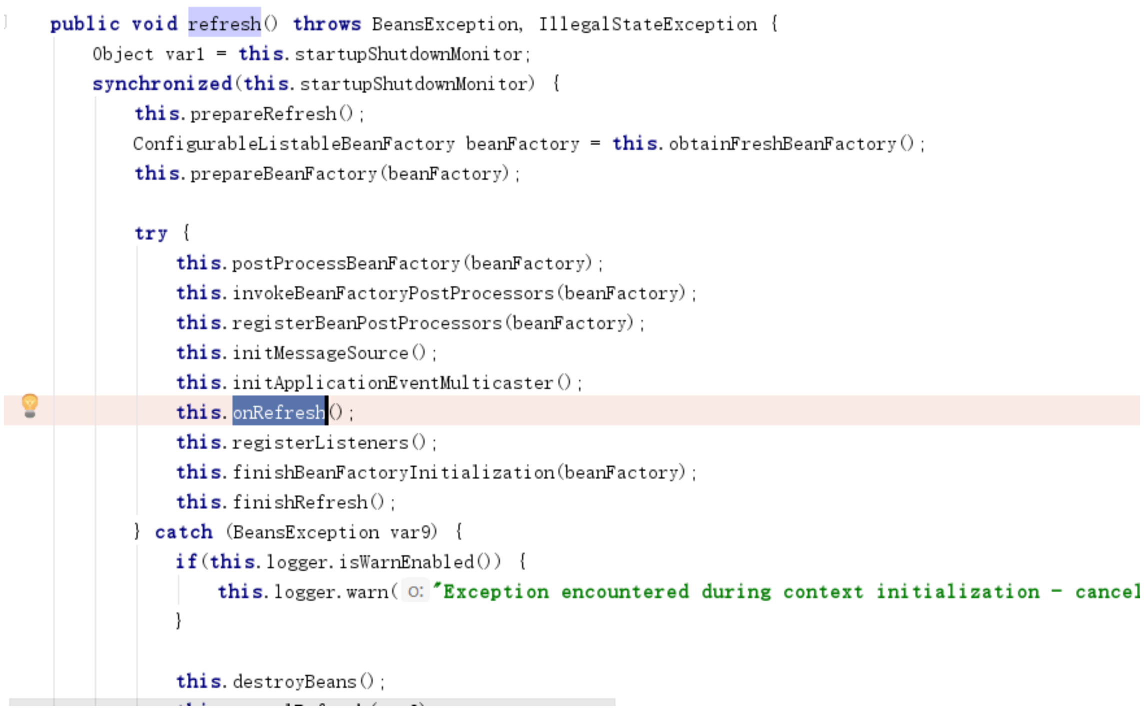
```java
public static class BeanPostProcessorsRegistrar implements ImportBeanDefinitionRegistrar, BeanFactoryAware {
        private ConfigurableListableBeanFactory beanFactory;

        @Override
        public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
                if (beanFactory instanceof ConfigurableListableBeanFactory) {
                        this.beanFactory = (ConfigurableListableBeanFactory) beanFactory;
                }
        }

        @Override
        public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
                        BeanDefinitionRegistry registry) {
                if (this.beanFactory == null) {
                        return;
                }
                registerSyntheticBeanIfMissing(registry,
                                "webServerFactoryCustomizerBeanPostProcessor",
                                WebServerFactoryCustomizerBeanPostProcessor.class);
                registerSyntheticBeanIfMissing(registry,
                                "errorPageRegistrarBeanPostProcessor",
                                ErrorPageRegistrarBeanPostProcessor.class);
        }
```

9: org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory#getWebServer **创建**tomcat **并且容器启动**

```
    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);
    }

    protected TomcatWebServer getTomcatWebServer(Tomcat tomcat) {
        //端口大于0启动启动
            return new TomcatWebServer(tomcat, getPort() >= 0);
    }

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

    tomcat启动流程
    private void initialize() throws WebServerException {
            TomcatWebServer.logger
                            .info("Tomcat initialized with port(s): " + getPortsDescription(false));
            synchronized (this.monitor) {
                    try {
                            addInstanceIdToEngineName();

                            Context context = findContext();
                            context.addLifecycleListener((event) -> {
                                if (context.equals(event.getSource())
                                        && Lifecycle.START_EVENT.equals(event.getType())) {
                                    // Remove service connectors so that protocol binding doesn't
                                    // happen when the service is started.
                                    removeServiceConnectors();
                                }
                            });

                            // Start the server to trigger initialization listeners
                            this.tomcat.start();

                            // We can re-throw failure exception directly in the main thread
                            rethrowDeferredStartupExceptions();

                            try {
                                ContextBindings.bindClassLoader(context, context.getNamingToken(),
                                        getClass().getClassLoader());
                            }
                            catch (NamingException ex) {
                                // Naming is not enabled. Continue
                            }

                            // Unlike Jetty, all Tomcat threads are daemon threads. We create a
                            // blocking non-daemon to stop immediate shutdown
                            startDaemonAwaitThread();
                    }
                    catch (Exception ex) {
                            stopSilently();
                            throw new WebServerException("Unable to start embedded Tomcat", ex);
                    }
            }
        }
```

10）在IOC 容器中的 org.springframework.context.support.AbstractApplicationContext#refresh 的 onReFresh（）带动tomcat启动

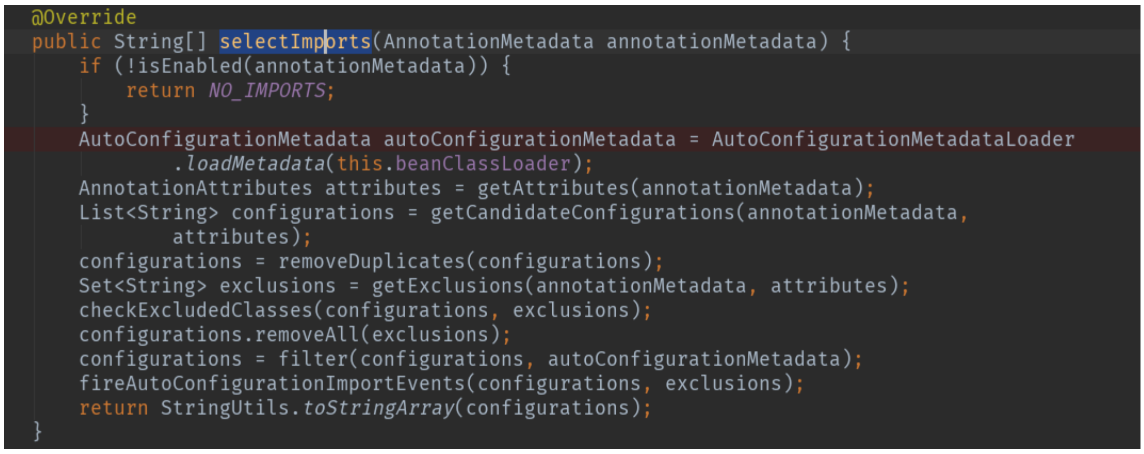
然后在接着执行 ioc容器的其他步骤。



```
public void refresh() throws BeansException, IllegalStateException {
    Object var1 = this.startupShutdownMonitor;
    synchronized(this.startupShutdownMonitor) {
        this.prepareRefresh();
        ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();
        this.prepareBeanFactory(beanFactory);

        try {
            this.postProcessBeanFactory(beanFactory);
            this.invokeBeanFactoryPostProcessors(beanFactory);
            this.registerBeanPostProcessors(beanFactory);
            this.initMessageSource();
            this.initApplicationEventMulticaster();
            this.onRefresh();
            this.registerListeners();
            this.finishBeanFactoryInitialization(beanFactory);
            this.finishRefresh();
        } catch (BeansException var9) {
            if(this.logger.isWarnEnabled()) {
                this.logger.warn( o: "Exception encountered during context initialization - cancel
            }

            this.destroyBeans();
```

疑问？？？？？

1）AutoConfigurationImportSelector#selectImports 的方法是怎么触发的？



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

2）我们自己定义的一些@Controller @Service 怎么到容器中去的？？？

接下来 我们就一一解答你们的疑问?还是以debug的方式来为大家解答

1>AbstractApplicationContext#refresh(**容器的刷新**)

2>AbstractApplicationContext#invokeBeanFactoryPostProcessors **调用bean工厂的后置处理器**

3>PostProcessorRegistrationDelegate#invokeBeanDefinitionRegistryPostProcessors

4>ConfigurationClassPostProcessor#postProcessBeanDefinitionRegistry **配置类的后置处理器**

5>ConfigurationClassPostProcessor#processConfigBeanDefinitions **处理配置的bean定义**

5.1）**找到候选的配置类**(tulingvipSpringbootAutoconfigPrincipleApplication)**我们自己项目中的配置类**

5.2 ）**创建配置类解析器**

6>ConfigurationClassParser#parse **解析我们自己的配置类**(tulingvipSpringbootAutoconfigPrincipleApplication)

7>ConfigurationClassParser#processConfigurationClass**处理配置类**

7.1)**处理配置类上的**@PropertySource**注解** ConfigurationClassParser#processPropertySource

7.2）**处理**@ComponentScan**注解的** ComponentScanAnnotationParser#parse

①:**创建 类路径下的**bean**定义扫描器** ClassPathBeanDefinitionScanner

..**多个步骤 解析**@ComponentScan **注解的属性**

②:ClassPathBeanDefinitionScanner#doScan **真正的扫描**(tulingvipSpringbootAutoconfigPrincipleApplication**所在的包**)

③:**返回我们标志了**@Controller @Service @Response @compent**注解的**bean**定义**

7.3）**处理**@Import**注解** ConfigurationClassParser#processImports

7.4）**处理**@ImportSource**注解**

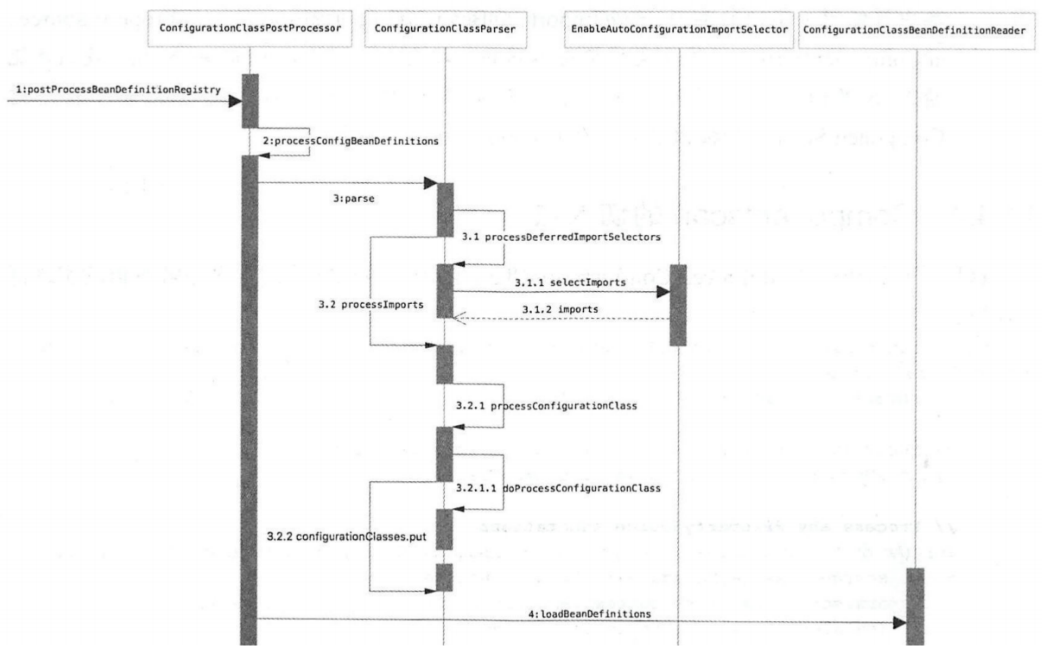
7.5）**处理**@Bean**注解的**

8>ConfigurationClassParser#processDeferredImportSelectors **处理实现了**ImportSelectors**接口的**

9>AutoConfigurationGroup#process (**获取 容器中的所有**ImportSelector **包含了** AutoConfigurationImportSelector)

10>ImportSelector#selectImports **回调** AutoConfigurationImportSelector.selectImports

11: ConfigurationClassBeanDefinitionReader#loadBeanDefinitions **把解析出来的类的**bean**定义 注册到容器中**


ConfigurationClassPostProcessor
ConfigurationClassParser
EnableAutoConfigurationImportSelector
ConfigurationClassBeanDefinitionReader
1:postProcessBeanDefinitionRegistry
2:processConfigBeanDefinitions
3:parse
3.1 processDeferredImportSelectors
3.1.1 selectImports
3.2 processImports
3.1.2 imports
3.2.1 processConfigurationClass
3.2.1.1 doProcessConfigurationClass
3.2.2 configurationClasses.put
4:loadBeanDefinitions