SpringBoot源码分析

SpringBoot简化了基于Spring的Java应用开发，降低了使用难度。从这个意义上来讲SpringBoot是对Spring狂的进一步封装。这个封装是的很多使用者：“只知其然，而不知其所以然”。在使用过程中出现问题时，不知道如何排错或者不能更好地使用SpringBoot。接下来本章节从几个维度来分析SpringBoot的源码：

- SpringBoot的自动配置原理
- SpringBoot启动流程
- SpringBoot的starter
- SpringBoot的内嵌Web服务器原理
- SpringBoot的扫描包@CompentScan的远离
- SpringBoot的@Value的原理
- SpringBoot和Spring的关系

01、下载SpringBoot的源码

springboot的源代码托管给了github。下载地址是：

> *https://github.com/spring-projects/spring-boot.git*

手动下载：

```
git clone https://github.com/spring-projects/spring-boot.git
```

02、剖析自动配置原理

Springboot是基于零XML配置的。使用：“习惯优先于配置”的策略。如果默认不满足要求，那么大部分情况下只需要在配置文件中配置即可。要完成这些工作，需要自动配置。要搞清楚SpringBoot的自动配置原理。需要了解Spring替代XML配置的注解：

- @Configuration注解：用在某个类上，表名被注解类的是一个配置类。对应Spring的XML配置文件。在该类某个方法上使用@Bean。完成Spring的bean的创建。已替代配置。

- @ComponentScan注解：扫码@Component、@Service、@Controller、@Repository等注解。如果使用该注解的默认值，只扫码被注解类当前包以及子包下的bean类，如果配置了basePackages，则扫描指定包，它替代了XML中的：context:component-scan/
- @EnableAutoConfiguration注解：内部原理是@Import注解，表明导入一个或者多个组件类，通常是配置类，也可以是一个实现selectImport接口的子类。这个功能等价于XML的导入其他的XML配置文件，但是功能增强了。可以导入@Configuration注解的配置类、和ImportSelector和ImportBeanDefinitionRegistrar实现类，以及常规的注解类。

通常SpringBoot启动类和starter不在同一个包下。而@ComponentScan默认只能扫描当前包和子包。在启动类中考@Import去导入starter配置类不算自动配置。

SpringBoot要完成自动配置，需要有新的机制来读取其他包下的配置类，事件监听等，SprignBoot依靠SpringFactoiresLoader类读取META-INF/Spring.factories配置的Spring配置类等，实现了自动配置。

03、@SpringBootApplication注解

对于springBoot项目通过需要添加一个@SpringBootApplication，该注解源代码如图所示，它是一个复合注解：

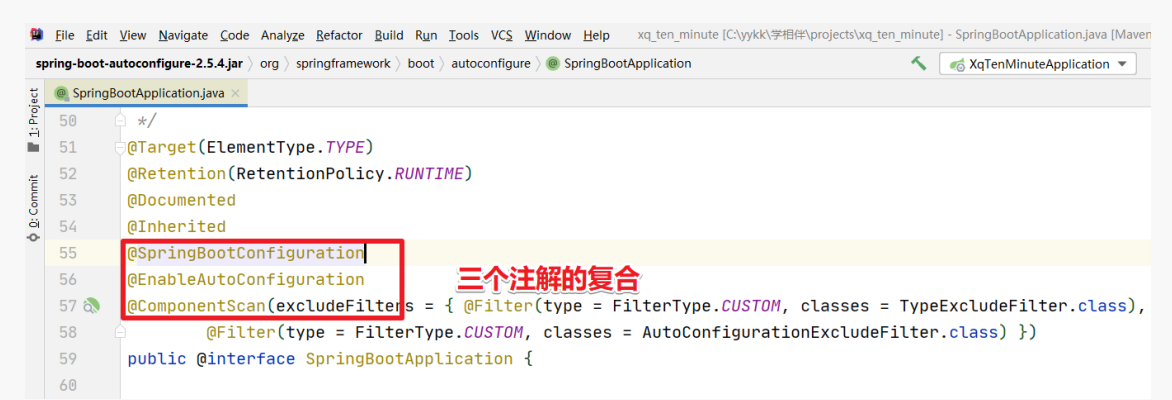
```
package com.xq;

import com.xq.common.jwt.JwtOperatorProperties;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.context.properties.EnableConfigurationPrope
rties;
import org.springframework.stereotype.Repository;


@SpringBootApplication
@MapperScan("com.xq.mapper")
@EnableConfigurationProperties(JwtOperatorProperties.class)
public class XqTenMinuteApplication {


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

```
19      }
20  }
21
```



- @SpringBootConfiguration：内部原理是@Configuration，代表被@SpringBootConfiguration注解的类是一个配置类，在配置类中可以定义一个或者多个@Bean的方法，初始化放入到ioc容器中。
- @EnableAutoConfiguration：用于触发自动配置，原理是@Import机制。
- @ComponentScan：扫描当前包以及子包下的加了注解的bean类。

04、@EnableAutoConfiguration注解

@EnableAutoConfiguration该注解会触发自动导入，源码如下图：

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

其上的@Import注解用于导入其他配置类，配置的value属性值是一个ImportSelector接口的实现类。

- 方法selectImports()方法根据参数AnnotationMetadata值返回所有候选配置类全限定名，
- getExclusionFilter()方法从候选配置类中筛选出满足条件的候选配置类。

ImportSelector接口如下：

```java
public interface ImportSelector {

    String[] selectImports(AnnotationMetadata
importingClassMetadata);

    @Nullable
    default Predicate<String> getExclusionFilter() {
        return null;
    }
}

```

05、AutoConfigurationImportSelector类

AutoConfigurationImportSelector是ImportSelector接口的子类。其中selectImports方法是覆盖的方法如下：

```java
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        }
        AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
        return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
```

```java
// 1: 该方法返回需要导入的配置类
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata)
{
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    // 读取自动配置类.
    AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(annotationMetadata);
    return
StringUtils.toStringArray(autoConfigurationEntry.getConfigurations()
);
}
```

方法:getAutoConfigurationEntry()中调用了getCandidateConfigurations来读取候选配置类：如下：

```
```

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    // 读取候选配置类
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
    // 过滤筛选出满足条件的配置类，把不满足条件的配置类进行提出
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}

```

方法：getCandidateConfigurations方法中调用了 SpringFactoriesLoader的静态方法 loadFactoryNames 如下：

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    // 该方法的作用：用于读取META-INF/spring.factories文件内容，该文件内容是属性文件格式。
        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;
    }
```

方法SpringFactoriesLoader.loadFactoryNames()用于读取META-INF/spring.factories文件内容，该文件内容是属性文件格式。如下：

```
// 用于读取META-INF/spring.factories文件内容，该文件内容是属性文件格式。如下：

```

```
public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
    ClassLoader classLoaderToUse = classLoader;
    if (classLoader == null) {
        classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
    }

    String factoryTypeName = factoryType.getName();
    // 用于读取META-INF/spring.factories文件内容，该文件内容是属性文件格式。如下：
    return (List)loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
}

private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader) {
    // 1：根据类加载器去缓存中读取信息，目的：提升性能和速度
    Map<String, List<String>> result = (Map)cache.get(classLoader);
    if (result != null) {
        return result;
    } else {
        // 用来存储监听器，事件，配置类的容器map
        HashMap result = new HashMap();
        try {
            // 找到类路径下包括资源文件META-INF/spring.factories的所有类资源。

            Enumeration urls = classLoader.getResources("META-INF/spring.factories");

            while(urls.hasMoreElements()) {
                URL url = (URL)urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                Iterator var6 = properties.entrySet().iterator();

                while(var6.hasNext()) {
                    Entry<?, ?> entry = (Entry)var6.next();
                    String factoryTypeName = ((String)entry.getKey()).trim();
                    String[] factoryImplementationNames = StringUtils.commaDelimitedListToStringArray((String)entry.getValue());
                    String[] var10 = factoryImplementationNames;
```

```
                    int var11 = factoryImplementationNames.length;

                    for(int var12 = 0; var12 < var11; ++var12) {
                        String factoryImplementationName =
var10[var12];

((List)result.computeIfAbsent(factoryTypeName, (key) -> {
                            return new ArrayList();
                        })).add(factoryImplementationName.trim());
                    }
                }
            }

            result.replaceAll((factoryType, implementations) -> {
                return
(List)implementations.stream().distinct().collect(Collectors.collect
ingAndThen(Collectors.toList(), Collections::unmodifiableList));
            });
            cache.put(classLoader, result);
            return result;
        } catch (IOException var14) {
            throw new IllegalArgumentException("Unable to load
factories from location [META-INF/spring.factories]", var14);
        }
    }
}

```

上面的代码含义是：

把自定义的配置类，监听器、事件等配置到META-INF/spring.factories中，无论是否在SpringBoot启动类包以及子包下，类都会把SpringBoot自动扫码到。这就是就是的自动配置的基本原理。

比如：spring-boot-autoconfigure-2.6.2.jar 自身的提供的META-INF/spring.factories配置如下：

```
# Initializers 启动应用下下文监听器初始化
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactory
ContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluation
ReportLoggingListener

# Application Listeners spring的事件监听
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer
```

```
9
10 # Environment Post Processors
11 org.springframework.boot.env.EnvironmentPostProcessor=\
12 org.springframework.boot.autoconfigure.integration.IntegrationProp
   ertiesEnvironmentPostProcessor
13
14 # Auto Configuration Import Listeners
15 org.springframework.boot.autoconfigure.AutoConfigurationImportList
   ener=\
16 org.springframework.boot.autoconfigure.condition.ConditionEvaluati
   onReportAutoConfigurationImportListener
17
18 # Auto Configuration Import Filters
19 org.springframework.boot.autoconfigure.AutoConfigurationImportFilt
   er=\
20 org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
21 org.springframework.boot.autoconfigure.condition.OnClassCondition,
   \
22 org.springframework.boot.autoconfigure.condition.OnWebApplicationC
   ondition
23
24 # Auto Configure   配置类，多个以逗号分开，如果换行需要加\
25 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
26 org.springframework.boot.autoconfigure.admin.SpringApplicationAdmi
   nJmxAutoConfiguration,\
27 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
28 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguratio
   n,\
29 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguratio
   n,\
30 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguratio
   n,\
31 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConf
   iguration,\
32 org.springframework.boot.autoconfigure.context.ConfigurationProper
   tiesAutoConfiguration,\
33 org.springframework.boot.autoconfigure.context.LifecycleAutoConfig
   uration,\
34 org.springframework.boot.autoconfigure.context.MessageSourceAutoCo
   nfiguration,\
35 org.springframework.boot.autoconfigure.context.PropertyPlaceholder
   AutoConfiguration,\
36 org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConf
   iguration,\
37 org.springframework.boot.autoconfigure.dao.PersistenceExceptionTra
   nslationAutoConfiguration,\
```

```
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.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
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.Neo4jReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfiguration,\
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.ElasticsearchRestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
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.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration,\
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.neo4j.Neo4jAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.netty.NettyAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration,\
org.springframework.boot.autoconfigure.r2dbc.R2dbcTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketRequesterAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketServerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketStrategiesAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
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.security.rsocket.RSocketSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.saml2.Saml2RelyingPartyAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,\
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.ReactiveMultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebSessionIdResolverAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration,\
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,\
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.autoconfigure.data.redis.RedisUrlSyntaxFailureAnalyzer,\
org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnalyzer,\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationScriptMissingFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jooq.NoDslContextBeanFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.MissingR2dbcPoolDependencyFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.MultipleConnectionPoolConfigurationsFailureAnalzyer,\
org.springframework.boot.autoconfigure.r2dbc.NoConnectionFactoryBeanFailureAnalyzer,\
org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer

# Template availability providers
org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider=\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabilityProvider,\
```

```
org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityProvider

# DataSource initializer detectors
org.springframework.boot.sql.init.dependency.DatabaseInitializerDetector=\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationInitializerDatabaseInitializerDetector

# Depends on database initialization detectors
org.springframework.boot.sql.init.dependency.DependsOnDatabaseInitializationDetector=\
org.springframework.boot.autoconfigure.batch.JobRepositoryDependsOnDatabaseInitializationDetector,\
org.springframework.boot.autoconfigure.quartz.SchedulerDependsOnDatabaseInitializationDetector,\
org.springframework.boot.autoconfigure.session.JdbcIndexedSessionRepositoryDependsOnDatabaseInitializationDetector

```

06、@Conditional注解

@Conditional从Spring4开始引入，用于条件性启动或者禁用@Configuration类或者@Bean方法。Starter配置的一些Bean可能需要修改，比如：默认数据源是HikariDataSource换成Druid数据源，那么默认的数据库HikariDataSource对应的Bean就不能在配置了，否则就会存在两个数据源，因而某些Bean是否需要注册到Spring容器是有条件的。SpringBoot使用@Conditional来完成Bean的条件注册。接下来用一些例子来说明：

需求

根据当前操作系统返回列举文件夹的命令：

- Windows -- dir
- Linux -- ls

01、新建一个maven项目 spring-boot-conditional-20

02、定义接口

```
package com.conditional.service;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:19
 */
public class LinuxListService implements ListService {

    @Override
    public String showCommand() {
        return "ls";
    }
}

```

window服务

```
package com.conditional.service;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:19
 */
public class WindowListService implements ListService {

    @Override
    public String showCommand() {
        return "dir";
    }
}

```

linux服务

```
package com.conditional.service;

/**
 * @author 飞哥
```

```
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:19
 */
public class LinuxListService implements ListService {

    @Override
    public String showCommand() {
        return "ls";
    }
}

```

03、定义Controller

```
package com.conditional.controller;

import com.conditional.service.ListService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:21
 */
@RestController
public class ListController {

    @Autowired
    private ListService listService;
}

```

04、定义配置类

```
package com.conditional.config;

import com.conditional.service.LinuxListService;
```

```java
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
public class ApplicationConfiguration {

    @Bean
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    public ListService linuxListService() {
        return new LinuxListService();
    }
}

```

05、定义启动类

```java
package com.conditional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import java.util.Arrays;

@SpringBootApplication
public class SpringBootConditional20Application {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
```

```
14
     SpringApplication.run(SpringBootConditional20Application.class,
    args);
15
16          // 打印所有的bean，以便于测试
17          String[] beanDefinitionNames =
    applicationContext.getBeanDefinitionNames();
18
     Arrays.stream(beanDefinitionNames).forEach(System.out::println);
19      }
20
21  }
22
```

06、测试

点击运行启动类，控制台输出，如下所示，以为你有两个实现类满足条件，Spring无法判断注入那个实现类对象给接口。

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" -
XX:TieredStopAtLevel=1 -noverify -Dspring.output.ansi.enabled=always
-Dcom.sun.management.jmxremote -Dspring.jmx.enabled=true -
Dspring.liveBeansView.mbeanDomain -
Dspring.application.admin.enabled=true "-javaagent:C:\Program
Files\JetBrains\IntelliJ IDEA
2020.2.1\lib\idea_rt.jar=13809:C:\Program Files\JetBrains\IntelliJ
IDEA 2020.2.1\bin" -Dfile.encoding=UTF-8 -classpath "C:\Program
Files\Java\jdk1.8.0_221\jre\lib\charsets.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\deploy.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\access-bridge-64.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\cldrdata.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\dnsns.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\jaccess.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\jfxrt.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\localedata.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\nashorn.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunec.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunjce_provider.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunmscapi.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunpkcs11.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\zipfs.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\javaws.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jce.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jfr.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jfxswt.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jsse.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\management-agent.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\plugin.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\resources.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\rt.jar;C:\yykk\旅游项目实战开发\学相伴旅
游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析
\spring-boot-conditional-
20\target\classes;C:\yykk\respository\org\springframework\boot\sprin
g-boot-starter-web\2.6.2\spring-boot-starter-web-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter\2.6.2\spring-boot-starter-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-
boot\2.6.2\spring-boot-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
autoconfigure\2.6.2\spring-boot-autoconfigure-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-logging\2.6.2\spring-boot-starter-logging-
2.6.2.jar;C:\yykk\respository\ch\qos\logback\logback-
classic\1.2.9\logback-classic-
1.2.9.jar;C:\yykk\respository\ch\qos\logback\logback-
core\1.2.9\logback-core-
```

```
1.2.9.jar;C:\yykk\respository\org\apache\logging\log4j\log4j-to-
slf4j\2.17.0\log4j-to-slf4j-
2.17.0.jar;C:\yykk\respository\org\apache\logging\log4j\log4j-
api\2.17.0\log4j-api-2.17.0.jar;C:\yykk\respository\org\slf4j\jul-
to-slf4j\1.7.32\jul-to-slf4j-
1.7.32.jar;C:\yykk\respository\jakarta\annotation\jakarta.annotation
-api\1.3.5\jakarta.annotation-api-
1.3.5.jar;C:\yykk\respository\org\yaml\snakeyaml\1.29\snakeyaml-
1.29.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-json\2.6.2\spring-boot-starter-json-
2.6.2.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
databind\2.13.1\jackson-databind-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
annotations\2.13.1\jackson-annotations-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
core\2.13.1\jackson-core-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\datatype\jackso
n-datatype-jdk8\2.13.1\jackson-datatype-jdk8-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\datatype\jackso
n-datatype-jsr310\2.13.1\jackson-datatype-jsr310-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\module\jackson-
module-parameter-names\2.13.1\jackson-module-parameter-names-
2.13.1.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-tomcat\2.6.2\spring-boot-starter-tomcat-
2.6.2.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
core\9.0.56\tomcat-embed-core-
9.0.56.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
el\9.0.56\tomcat-embed-el-
9.0.56.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
websocket\9.0.56\tomcat-embed-websocket-
9.0.56.jar;C:\yykk\respository\org\springframework\spring-
web\5.3.14\spring-web-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
beans\5.3.14\spring-beans-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
webmvc\5.3.14\spring-webmvc-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
aop\5.3.14\spring-aop-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
context\5.3.14\spring-context-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
expression\5.3.14\spring-expression-
5.3.14.jar;C:\yykk\respository\org\slf4j\slf4j-api\1.7.32\slf4j-api-
1.7.32.jar;C:\yykk\respository\org\springframework\spring-
core\5.3.14\spring-core-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
```

```
jcl\5.3.14\spring-jcl-5.3.14.jar"
com.conditional.SpringBootConditional20Application

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.2)

2021-12-28 17:24:56.917  INFO 9128 --- [           main] c.c.SpringBootConditional20Application   : Starting SpringBootConditional20Application using Java 1.8.0_221 on DESKTOP-27SNMQ8 with PID 9128 (C:\yykk\旅游项目实战开发\学相伴旅游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析\spring-boot-conditional-20\target\classes started by 86150 in C:\yykk\旅游项目实战开发\学相伴旅游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析\spring-boot-conditional-20)
2021-12-28 17:24:56.921  INFO 9128 --- [           main] c.c.SpringBootConditional20Application   : No active profile set, falling back to default profiles: default
2021-12-28 17:24:58.001  INFO 9128 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-12-28 17:24:58.014  INFO 9128 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-12-28 17:24:58.014  INFO 9128 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.56]
2021-12-28 17:24:58.144  INFO 9128 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-12-28 17:24:58.145  INFO 9128 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1170 ms
2021-12-28 17:24:58.208  WARN 9128 --- [           main] ConfigServletWebServerApplicationContext : Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'listController': Unsatisfied dependency expressed through field 'listService'; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.conditional.service.ListService' available: expected single matching bean but found 2: windowListService,linuxListService
```

```
2021-12-28 17:24:58.211  INFO 9128 --- [           main] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2021-12-28 17:24:58.227  INFO 9128 --- [           main] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-12-28 17:24:58.257 ERROR 9128 --- [           main] o.s.b.d.LoggingFailureAnalysisReporter   :

***************************
APPLICATION FAILED TO START
***************************

Description:

Field listService in com.conditional.controller.ListController required a single bean, but 2 were found:
    - windowListService: defined by method 'windowListService' in class path resource [com/conditional/config/ApplicationConfiguration.class]
    - linuxListService: defined by method 'linuxListService' in class path resource [com/conditional/config/ApplicationConfiguration.class]


Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed


Process finished with exit code 1

```

07、Window下的条件类

定义类WindowConditional实现接口Conditional，如果是在window下就返回true，否则返回false.

```java
package com.conditional.conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
```

```
import org.springframework.core.type.AnnotatedTypeMetadata;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:29
 */
public class WindowsConditional implements Condition {

    @Override
    public boolean matches(ConditionContext context,
AnnotatedTypeMetadata metadata) {
        return
context.getEnvironment().getProperty("os.name").toLowerCase().contai
ns("windows");
    }
}

```

定义类LinuxConditional实现接口Conditional，如果是在linux下就返回true，否则返回false.

```
package com.conditional.conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:29
 */
public class LinuxConditional implements Condition {

    @Override
    public boolean matches(ConditionContext context,
AnnotatedTypeMetadata metadata) {
        return
context.getEnvironment().getProperty("os.name").toLowerCase().contai
ns("linux");
```

```
    }
}

```

08、修改配置类

```
package com.conditional.config;

import com.conditional.conditional.LinuxConditional;
import com.conditional.conditional.WindowsConditional;
import com.conditional.service.LinuxListService;
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
public class ApplicationConfiguration {

    @Bean
    @Conditional(WindowsConditional.class)
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    @Conditional(LinuxConditional.class)
    public ListService linuxListService() {
        return new LinuxListService();
    }
}

```

09测试

运行启动类，项目正常启动，由于项目是在windows系统下。所以WindowsConditional条件满足，所以会把WindowListService注册到spring的ioc容器中，自然结果就是：dir

10、测试添加在类上

修改ApplicationConfiguration，把注解条件增加在配置类上，为了验证@Conditional的方法和类哪个优先，让类返回false.

```
package com.conditional.config;

import com.conditional.conditional.LinuxConditional;
import com.conditional.conditional.WindowsConditional;
import com.conditional.service.LinuxListService;
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
@Conditional(LinuxConditional.class)
public class ApplicationConfiguration {

    @Bean
    @Conditional(WindowsConditional.class)
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    @Conditional(LinuxConditional.class)
    public ListService linuxListService() {
        return new LinuxListService();
```

```
35      }
36  }
37
```

很清晰的看到项目启动失败了。因为你当前配置类必须在Linux环境下才会加载，而当前环境是window很明显匹配不上，因此整个配置类都不再处理。

07、常见的Conditional注解

只用一个注解就好，不要自己再来实现Condtion接口，Spring框架提供了一系列相关的注解，如下表

注解	说明
`@ConditionalOnSingleCandidate`	当给定类型的bean存在并且指定为Primary的给定类型存在时,返回true
`@ConditionalOnMissingBean`	当给定的类型、类名、注解、昵称在beanFactory中不存在时返回true.各类型间是or的关系
`@ConditionalOnBean`	与上面相反，要求bean存在
`@ConditionalOnMissingClass`	当给定的类名在类路径上不存在时返回true,各类型间是and的关系
`@ConditionalOnClass`	与上面相反，要求类存在
`@ConditionalOnCloudPlatform`	当所配置的CloudPlatform为激活时返回true
`@ConditionalOnExpression`	spel表达式执行为true
`@ConditionalOnJava`	运行时的java版本号是否包含给定的版本号.如果包含,返回匹配,否则,返回不匹配
`@ConditionalOnProperty`	要求配置属性匹配条件
`@ConditionalOnJndi`	给定的jndi的Location 必须存在一个.否则,返回不匹配
`@ConditionalOnNotWebApplication`	web环境不存在时
`@ConditionalOnWebApplication`	web环境存在时
`@ConditionalOnResource`	要求制定的资源存在

例子	说明

例子	说明
@ConditionalOnBean(javax.sql.DataSource.class)	Spring容器或者所有父容器中要存在至少一个javax.sql.DataSource类的实
@ConditionalOnClass({ Configuration.class,FreeMarkerConfigurationFactory.class })	类加载器中必须存在Configuration和FreeMarkerConfigurationFa这两个类
@ConditionalOnExpression(“’${server.host}’==’localhost’”)	server.host配置项的值需要是localhost
ConditionalOnJava(JavaVersion.EIGHT)	Java版本至少是8
@ConditionalOnMissingBean(value = ErrorController.class, search = SearchStrategy.CURRENT)	Spring当前容器中不存在ErrorController类型的bean
@ConditionalOnMissingClass(“GenericObjectPool”)	类加载器中不能存在GenericObjectPool这个类
@ConditionalOnNotWebApplication	必须在非Web应用下才会生效
@ConditionalOnProperty(prefix = “spring.aop”, name = “auto”, havingValue = “true”, matchIfMissing = true)	应用程序的环境中必须有spring.aop.auto这项配置，且的值是true或者环境中不存在spring.aop.auto配置(matchIfMissing为true)
@ConditionalOnResource(resources=”mybatis.xml”)	类加载路径中必须存在mybatis.xml文件
@ConditionalOnSingleCandidate(PlatformTransactionManager.class)	Spring当前或父容器中必须有PlatformTransactionManager个类型的实例，且只有一个实
@ConditionalOnWebApplication	必须在Web应用下才会生效

08、SpringBoot启动流程

01、SpringApplication初始化方法

我们在SpringBoot启动类中调用SpringApplication的静态方法run。如下代码所示：

```
package com.conditional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
```

```
6
7  import java.util.Arrays;
8
9  @SpringBootApplication
10 public class SpringBootConditional20Application {
11
12     public static void main(String[] args) {
13
     SpringApplication.run(SpringBootConditional20Application.class,
   args);
14     }
15 }
16
```

run方法如下：

```
1   public static ConfigurableApplicationContext run(Class<?>
   primarySource, String... args) {
2           return run(new Class[]{primarySource}, args);
3       }
```

它又调用了另外一个重载run方法，首先创建一个SpringApplication对象，然后调用非静态run方法

```
1  public static ConfigurableApplicationContext run(Class<?>[]
   primarySources, String[] args) {
2          return (new SpringApplication(primarySources)).run(args);
3      }
```

上面是一个run方法的重载。注意这个时候查看的时候要分为两段来分析：

- 构造函数部分
- run方法部分

构造函数部分

```
1  public SpringApplication(Class<?>... primarySources) {
2          this((ResourceLoader)null, primarySources);
3      }
```

```
1   public SpringApplication(ResourceLoader resourceLoader, Class<?>...
   primarySources) {
2           this.sources = new LinkedHashSet();
3           this.bannerMode = Mode.CONSOLE;
4           this.logStartupInfo = true;
```

```
        this.addCommandLineProperties = true;
        this.addConversionService = true;
        this.headless = true;
        this.registerShutdownHook = true;
        this.additionalProfiles = Collections.emptySet();
        this.isCustomEnvironment = false;
        this.lazyInitialization = false;
        this.applicationContextFactory = ApplicationContextFactory.DEFAULT;
        this.applicationStartup = ApplicationStartup.DEFAULT;
        this.resourceLoader = resourceLoader;
        Assert.notNull(primarySources, "PrimarySources must not be null");
        this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
        this.webApplicationType = WebApplicationType.deduceFromClasspath();
        this.bootstrapRegistryInitializers = new ArrayList(this.getSpringFactoriesInstances(BootstrapRegistryInitializer.class));

         this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));

         this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));
        this.mainApplicationClass = this.deduceMainApplicationClass();
    }
```

该构造函数的作用是：

- 对primarySources初始化
- 根据jar包推断webApplicationType的类型，进而创建对应类型的ApplicationContext
- 初始化ApplicationContextInitializer列表
- 初始化ApplicationListener列表
- 推断包含main方法的主类。

对**primarySources**初始化

spring现在提倡了使用java配置来替代XML配置信息可以来自多个类，这里指定一个主配置类。也就是当前启动类

webApplicationType类型

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.springframework.boot;

import org.springframework.util.ClassUtils;

public enum WebApplicationType {
    NONE,
    SERVLET,
    REACTIVE;

    private static final String[] SERVLET_INDICATOR_CLASSES = new String[]{"javax.servlet.Servlet", "org.springframework.web.context.ConfigurableWebApplicationContext"};
    private static final String WEBMVC_INDICATOR_CLASS = "org.springframework.web.servlet.DispatcherServlet";
    private static final String WEBFLUX_INDICATOR_CLASS = "org.springframework.web.reactive.DispatcherHandler";
    private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.ServletContainer";
    private static final String SERVLET_APPLICATION_CONTEXT_CLASS = "org.springframework.web.context.WebApplicationContext";
    private static final String REACTIVE_APPLICATION_CONTEXT_CLASS = "org.springframework.boot.web.reactive.context.ReactiveWebApplicationContext";

    private WebApplicationType() {
    }

    static WebApplicationType deduceFromClasspath() {
        if (ClassUtils.isPresent("org.springframework.web.reactive.DispatcherHandler", (ClassLoader)null) && !ClassUtils.isPresent("org.springframework.web.servlet.DispatcherServlet", (ClassLoader)null) && !ClassUtils.isPresent("org.glassfish.jersey.servlet.ServletContainer", (ClassLoader)null)) {
            return REACTIVE;
        } else {
            String[] var0 = SERVLET_INDICATOR_CLASSES;
            int var1 = var0.length;
```

```
31
32              for(int var2 = 0; var2 < var1; ++var2) {
33                  String className = var0[var2];
34                  if (!ClassUtils.isPresent(className,
    (ClassLoader)null)) {
35                      return NONE;
36                  }
37              }
38
39              return SERVLET;
40          }
41      }
42
43      static WebApplicationType deduceFromApplicationContext(Class<?>
    applicationContextClass) {
44          if
    (isAssignable("org.springframework.web.context.WebApplicationContext
    ", applicationContextClass)) {
45              return SERVLET;
46          } else {
47              return
    isAssignable("org.springframework.boot.web.reactive.context.Reactive
    WebApplicationContext", applicationContextClass) ? REACTIVE : NONE;
48          }
49      }
50
51      private static boolean isAssignable(String target, Class<?>
    type) {
52          try {
53              return ClassUtils.resolveClassName(target,
    (ClassLoader)null).isAssignableFrom(type);
54          } catch (Throwable var3) {
55              return false;
56          }
57      }
58  }
59
```

方法deduceFromClasspath主要根据几个常量指定类是否在类路径上返回webApplicationType的类型：

- NONE, 不需要内嵌Web容器
- SERVLET,：一个基于Servlet的Web应用，应该启动内嵌的Servlet容器
- REACTIVE;：一个基于Reactive的Web应用，应该启动内嵌的Reactive容器

初始化ApplicationContextInitializer和ApplicationListener

这两个初始化是读取自动配置类的原理一样，都是到jar的META-INF/spring.factories中读取它。它们分别读取的key不同如下：

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

```

推断包含main方法的主类。

```
private Class<?> deduceMainApplicationClass() {
        try {
            StackTraceElement[] stackTrace = (new RuntimeException()).getStackTrace();
            StackTraceElement[] var2 = stackTrace;
            int var3 = stackTrace.length;

            for(int var4 = 0; var4 < var3; ++var4) {
                StackTraceElement stackTraceElement = var2[var4];
                if ("main".equals(stackTraceElement.getMethodName())) {
                    return Class.forName(stackTraceElement.getClassName());
                }
            }
        } catch (ClassNotFoundException var6) {
        }

        return null;
    }

```

以标准Java程序启动，从main方法开始执行，目前正在执行的方法通过调用栈可以找到main方法所在类，

02、run方法部分

当SpringApplication创建完毕后，就开始执行run方法了。如下所示：

```
public ConfigurableApplicationContext run(String... args) {
        long startTime = System.nanoTime();
        DefaultBootstrapContext bootstrapContext = this.createBootstrapContext();
        ConfigurableApplicationContext context = null;
        this.configureHeadlessProperty();
        SpringApplicationRunListeners listeners = this.getRunListeners(args);
        listeners.starting(bootstrapContext, this.mainApplicationClass);

        try {
            ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
            ConfigurableEnvironment environment = this.prepareEnvironment(listeners, bootstrapContext, applicationArguments);
            this.configureIgnoreBeanInfo(environment);
            Banner printedBanner = this.printBanner(environment);
            context = this.createApplicationContext();
            context.setApplicationStartup(this.applicationStartup);
            this.prepareContext(bootstrapContext, context, environment, listeners, applicationArguments, printedBanner);
            this.refreshContext(context);
            this.afterRefresh(context, applicationArguments);
            Duration timeTakenToStartup = Duration.ofNanos(System.nanoTime() - startTime);
            if (this.logStartupInfo) {
                (new StartupInfoLogger(this.mainApplicationClass)).logStarted(this.getApplicationLog(), timeTakenToStartup);
            }

            listeners.started(context, timeTakenToStartup);
            this.callRunners(context, applicationArguments);
        } catch (Throwable var12) {
            this.handleRunFailure(context, var12, listeners);
```

```
            throw new IllegalStateException(var12);
        }

        try {
            Duration timeTakenToReady =
Duration.ofNanos(System.nanoTime() - startTime);
            listeners.ready(context, timeTakenToReady);
            return context;
        } catch (Throwable var11) {
            this.handleRunFailure(context, var11,
(SpringApplicationRunListeners)null);
            throw new IllegalStateException(var11);
        }
    }
```

该方法完成的工作如下：

- 启动一个秒表（StopWatch）来统计启动时间
- 通过SpringFactoriesLoader.loadFactoryName获取jar目录下的META-INF/spring.factories下配置的SpringApplicationRunListeners。该接口对SpringApplicaiton的run方法不同阶段进行监听。
- listeners.starting(bootstrapContext, this.mainApplicationClass); 调用了所有SpringApplicationRunListeners的starting()方法。
- ConfigurableEnvironment environment = this.prepareEnvironment(listeners, bootstrapContext, applicationArguments); 根据WebApplicationType类型准备对应类型的类型ConfigurableEnvironment，同时调用listeners.environmentPrepared(bootstrapContext, (ConfigurableEnvironment)environment);通知所有的SpringApplicationRunListener环境准备完毕。
- 打印Banner，如果spring.main.banner-mode=off。就不打印，如果值是console就打印banner到控制台。如果是log。就输出到日志，我们可以在resoruces目录下现金一个banner.txt来修改默认的banner.
- 根据WebApplicationType类型，创建一个类型的ApplicationContext对象。
- 准备上下文

```
private void prepareContext(DefaultBootstrapContext
bootstrapContext, ConfigurableApplicationContext context,
ConfigurableEnvironment environment, SpringApplicationRunListeners
listeners, ApplicationArguments applicationArguments, Banner
printedBanner) {
        context.setEnvironment(environment);// 设置运行环境
        this.postProcessApplicationContext(context);
//applicationContext进行后置处理
        this.applyInitializers(context);//调用所有的
ApplicationContextInitializer的initialize方法
        listeners.contextPrepared(context); // 通知所有监听器上下文准备
完毕
```

```
6           bootstrapContext.close(context);
7           if (this.logStartupInfo) {
8               this.logStartupInfo(context.getParent() == null);
9               this.logStartupProfileInfo(context);
10          }
11
12          ConfigurableListableBeanFactory beanFactory =
    context.getBeanFactory();
13          beanFactory.registerSingleton("springApplicationArguments",
    applicationArguments);
14          if (printedBanner != null) {
15              beanFactory.registerSingleton("springBootBanner",
    printedBanner);
16          }
17
18          if (beanFactory instanceof
    AbstractAutowireCapableBeanFactory) {
19
    ((AbstractAutowireCapableBeanFactory)beanFactory).setAllowCircularRe
    ferences(this.allowCircularReferences);
20              if (beanFactory instanceof DefaultListableBeanFactory) {
21
    ((DefaultListableBeanFactory)beanFactory).setAllowBeanDefinitionOver
    riding(this.allowBeanDefinitionOverriding);
22              }
23          }
24
25          if (this.lazyInitialization) {
26              context.addBeanFactoryPostProcessor(new
    LazyInitializationBeanFactoryPostProcessor());
27          }
28
29          // 加载所有资源
30          Set<Object> sources = this.getAllSources();
31          Assert.notEmpty(sources, "Sources must not be empty");
32          // 注册所有bean到springioc容器
33          this.load(context, sources.toArray(new Object[0]));
34          // 通知监听器上下文加载完毕。
35          listeners.contextLoaded(context);
36      }
37
```

09、SpringBoot的starter机制

10、SpringBoot的内置web容器原理

11、SpringBoot背后的扫包@ComponentScan原理