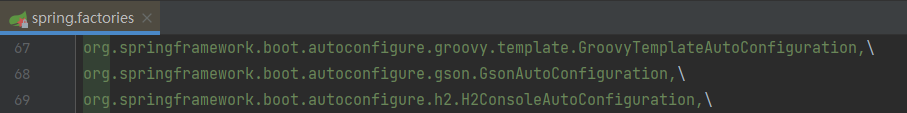
# ***Spring***

1. springBoot 自动装配原理

1.@SpringBootApplication 包含两个重要注解：@EnableAutoConfiguration(自动配置类)，@SpringBootConfiguration(包含@Configuration，表明这也是个配置类)。

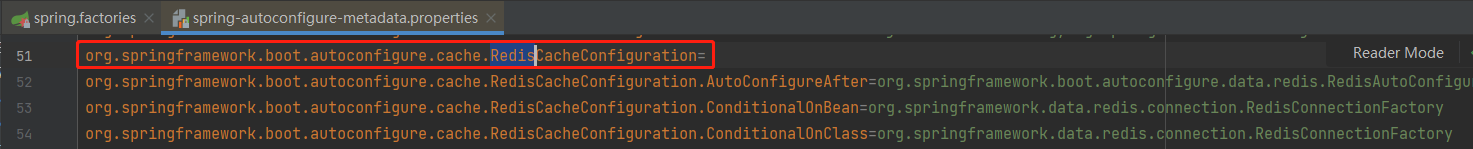
2. @EnableAutoConfiguration中包含@AutoConfigurationPackage(会加载本目录录下所有配置类)，@Import(自动导入资源)。

3. @Import中参数AutoConfigurationImportSelector.class,此类的List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes)会加载autoconfigure.jar/META-INF/spring.factories文件



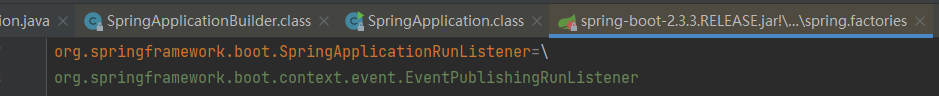
4. spring.factories文件中包含spring提供的所有的第三方组件配置类的全类名，这些类就是自动装配时所使用到的。

5. autoconfigure.jar/META-INF/下还有一个文件：spring-autoconfigure-metadata.properties，该文件中包含了配置类中的@Condition，AutoConfigurationImportFilter.class会在spring启动前通过match()方法将项目没有引用的配置类过滤掉，从而节约spring启动时使用@Condition来过滤的时间。



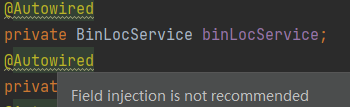
6.最后通过筛选的配置类会在springBoot启动时加入到spring容器中。

1. springBoot启动流程
2. 调用SpringApplication.class.run(String[] args) 。
3. 第一步:生成stopWath.class用于记录时间。
4. 第二步:获取springBoot.jar/META-INF/spring.factories文件，得到所有SpringApplicationRunListener，用于发布SpringApplicationEvent。



1. ConfigurableEnvironment environment = this.prepareEnvironment(listeners, applicationArguments)创建环境对象，输入jvm启动参数。
2. 获取banner信息。
3. 创建ConfigurableApplicationContext，用于判断项目类型：SERVLET,REACTIVE,DEFAULT。
4. 准备应用上下文prepareContext():设置环境对象，获取BeanFactory并注册单例bean(将beanName放入beanDefinitionMap中),发布contextLoaded事件。
5. 刷新应用上下文refreshContext():加载bean到容器中，是实现starter的关键，如果是web项目会创建一个server并启动。
6. 最后进行收尾发布时finish()事件返回配置应用上下文。

1. springBoot starter原理
2. 确定依赖并编写自动配置：@Configuraton:指定配置类 @ConditionOnXXX:指定条件下配置类生效 @AutoConfigureAfter:指定配置类顺序 @Bean添加组件 @ConfiguationPropertie:指定配置项目 @Enable ConfiguationProperties:使配置项生效
3. 在META-INF/中添加spring.factories文件，指定配置类全类名
4. 上传到maven服务器提供下载
5. 字段注入不推荐原因



Spring提供三种类型的注入方式：字段注入，构造器注入，setter()注入，其中字段注入与构造器注入是强依赖，如果没有准备好相应的bean，项目启动会出现异常。

官方文档给出的解释是：

1. 使用字段注入太过于方便，开发者可能会因此破坏单一职责，使用构造器注入则会稍微谨慎一点。
2. 使用字段注入无法使用final修饰符，这样bean的引用可能被认为修改造成异常
3. 使用字段注册则在进行单元测试时需要启动spring容器或者使用反射或者其他mock框架来生成需要的bean造成耦合
4. springMvc 原理

前端控制器（DispatcherServlet）：接收请求，响应结果。

处理器映射器（HandlerMapping）：根据URL去查找处理器，映射url与controller的方法。

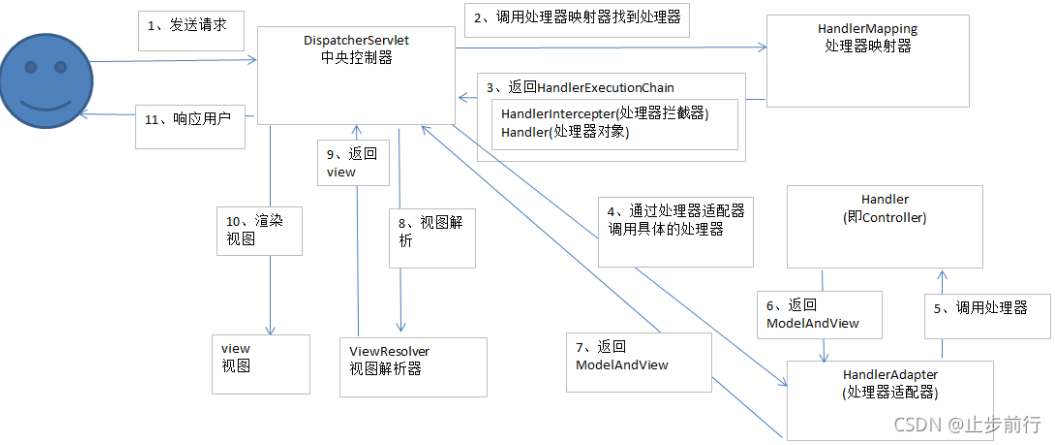
处理器（Handler）：（Controller）

处理器适配器（HandlerAdapter）：会把处理器包装成适配器，这样就可以支持多种类型的处理器（适配器模式的应用）

视图解析器（ViewResovler）：进行视图解析，多返回的字符串，进行处理，可以解析成对应的页面

**为什么需要HandlerAdapter**:

因为springMVC提供了三种形式的Hadler：实现HttpRequestHandler接口，@Controller等注解修饰，实现Controller接口，也正是因为handler的实现方式较多，而springMVC又需要统一处理请求，所以采用了适配器设计模式：HandlerMapping接口里面只有一个getHandler()方法，而且返回类型是HandlerExecutionChain，用HandlerExecutionChain里面定义了一个Object类型的handler属性，并对handler进行了封装，在每个请求里加入了拦截器链。然后将这个HandlerExecutionChain里面的handler传给了HandlerAdapter。



1. AOP原理

切面：拦截器类，其中会定义切点以及通知

切点：具体拦截的某个业务点。

通知：切面当中的方法，声明通知方法在目标业务层的执行位置，通知类型如下：

前置通知：@Before 在目标业务方法执行之前执行

后置通知：@After 在目标业务方法执行之后执行

返回通知：@AfterReturning 在目标业务方法返回结果之后执行

异常通知：@AfterThrowing 在目标业务方法抛出异常之后

环绕通知：@Around 功能强大，可代替以上四种通知，还可以控制目标业务方法是否执行以及何时执行

**JDKProxy还是CGLIBProxy：**

JDK原生动态代理是Java原生支持的，不需要任何外部依赖，但是它只能基于接口进行代理；CGLIB通过继承的方式进行代理，无论目标对象有没有实现接口都可以代理，但是无法处理final的情况。目前springBoot2.x默认使用的cglib，如果偏要springboot 走JDK动态代理，那么需要在application.properties里面配置spring.aop.proxy-target-class=false。

# ***Redis***

1. 数据持久化
2. 集群搭建方式
3. 哨兵模式
4. 数据类型与底层结构
5. 为什么快
6. 体系结构

# ***RocketMq***

1. 体系结构
2. 事务消息
3. 优点与应用场景

# ***MySql***

1. 事务隔离级别
2. 索引原则
3. B+ 树
4. 锁
5. 执行计划
6. 执行顺序