### 1: Spring Cloud Gateway 启动初始化的过程

引入：<dependency>

<groupId>org.springframework.cloud</groupId>

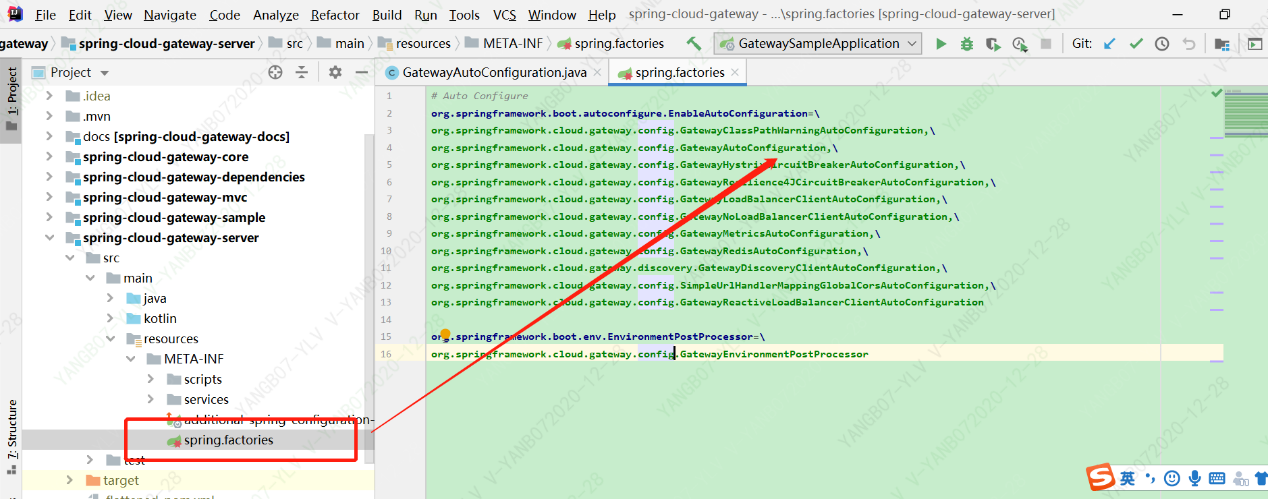
<artifactId>spring-cloud-starter-gateway</artifactId>

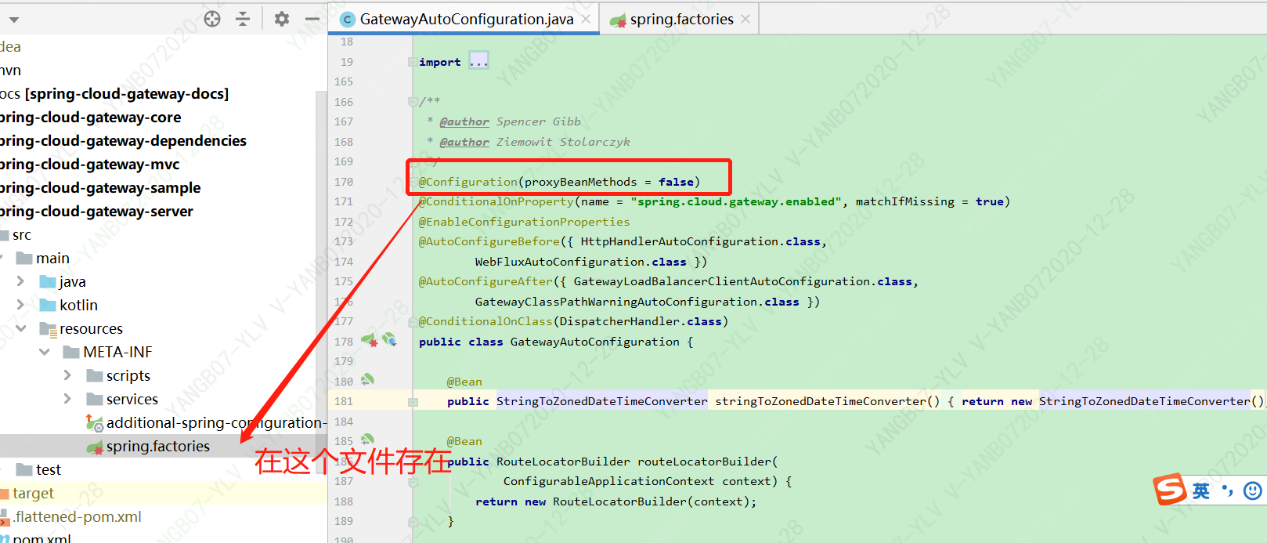
</dependency>

#### 怎么从源头加载起来的

springboot的启动注解@SpringBootApplication，源码里面有包含三个注解

* @SpringBootConfiguration    #用于spring的xml配置信息加入到springIOC容器的配置类
* @EnableAutoConfiguration    #最重要：从classpath中找所有spring.factories文件，并将其中的配置项信息通过反射实例化有类上标注了@Configuration的Java文件，然后汇总并加载到IOC容器中，如下截图：





* @ComponentScan：自动扫描并加载符合条件的组件，如@Comonent @Resitory

,@Bean注解到IOC容器中

gateway启动的过程其实就是构建spring.factories文件的对象然后将各个模块拼装到一起交给RoutePredicateHandlerMapping，

1. create route definition and collect

2. collect resource and compose RouteDefinitionRouteLocator

3. init filter handler

4. compose handler mapping

5. add refresh event listener

6. add HttpHeaderFilter beans

7. init GlobalFilter beans

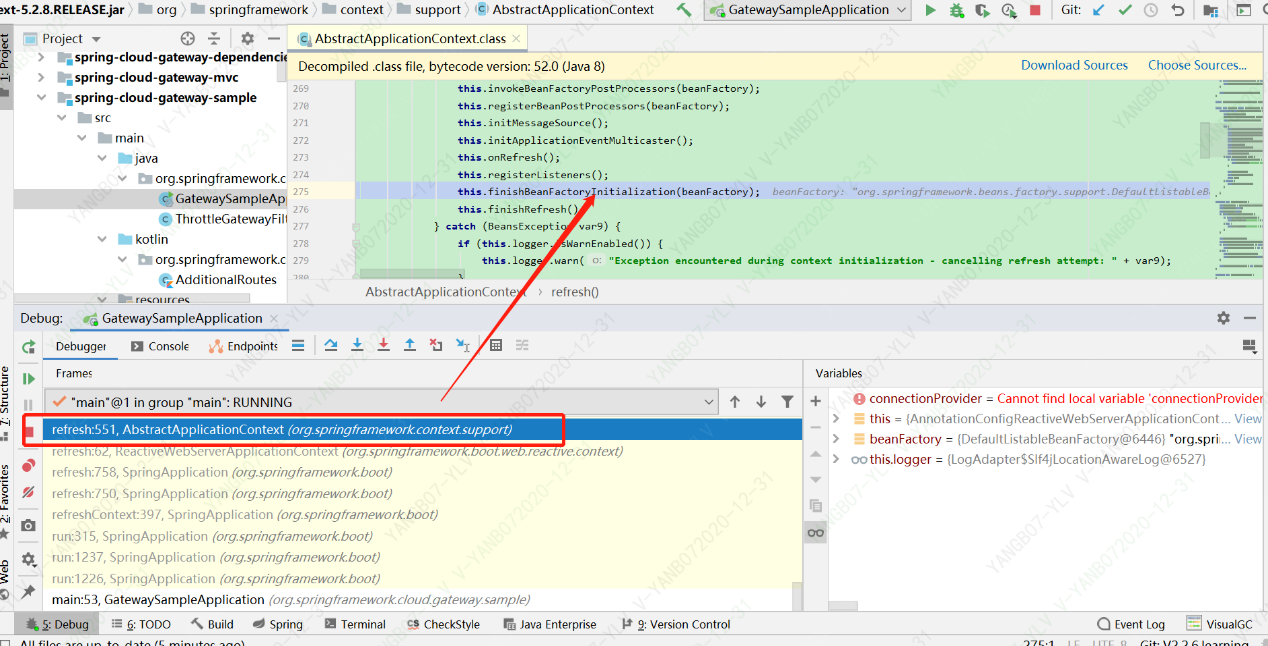
8. init Predicate Factory beans

9. init GatewayFilter Factory beans

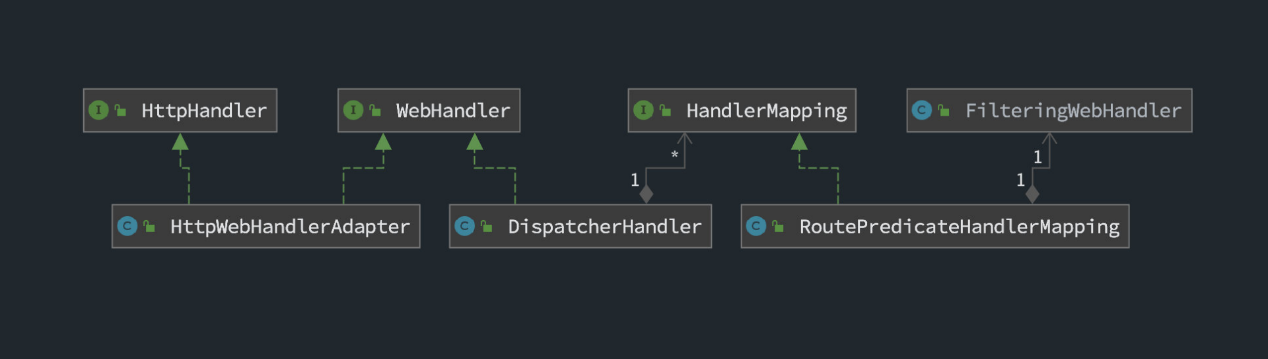
各个组件初始化核心代码：

查看debug调用栈信息，从scg启动main函数到AbstractApplicationContext的refresh，

finishBeanFactoryInitialization(beanFactory)是初始化所有剩下的单实例，将spring.factories文件的对象加入到spring 容器的核心处理逻辑



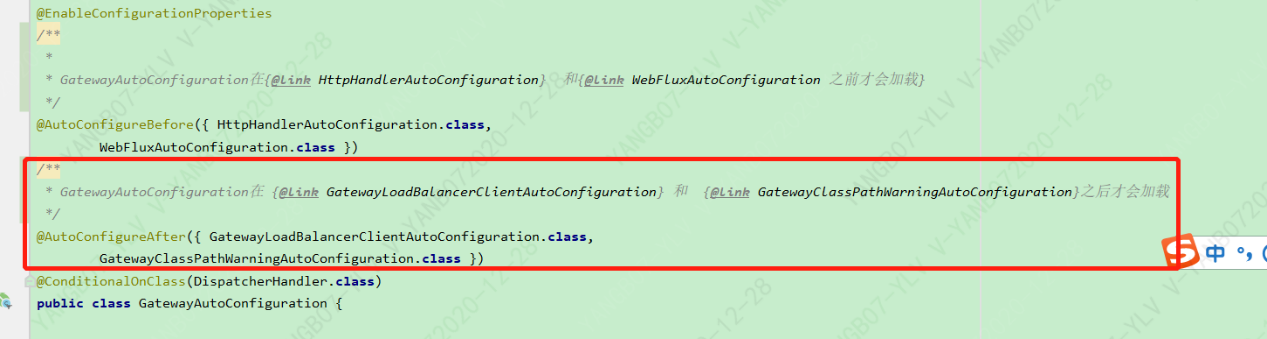
gateway的定义, 作为SpringMVC中的一个模块存在



图很好地定义了HttpHandler、DispatcherHandler和RoutePredicateHandlerMapping的关系

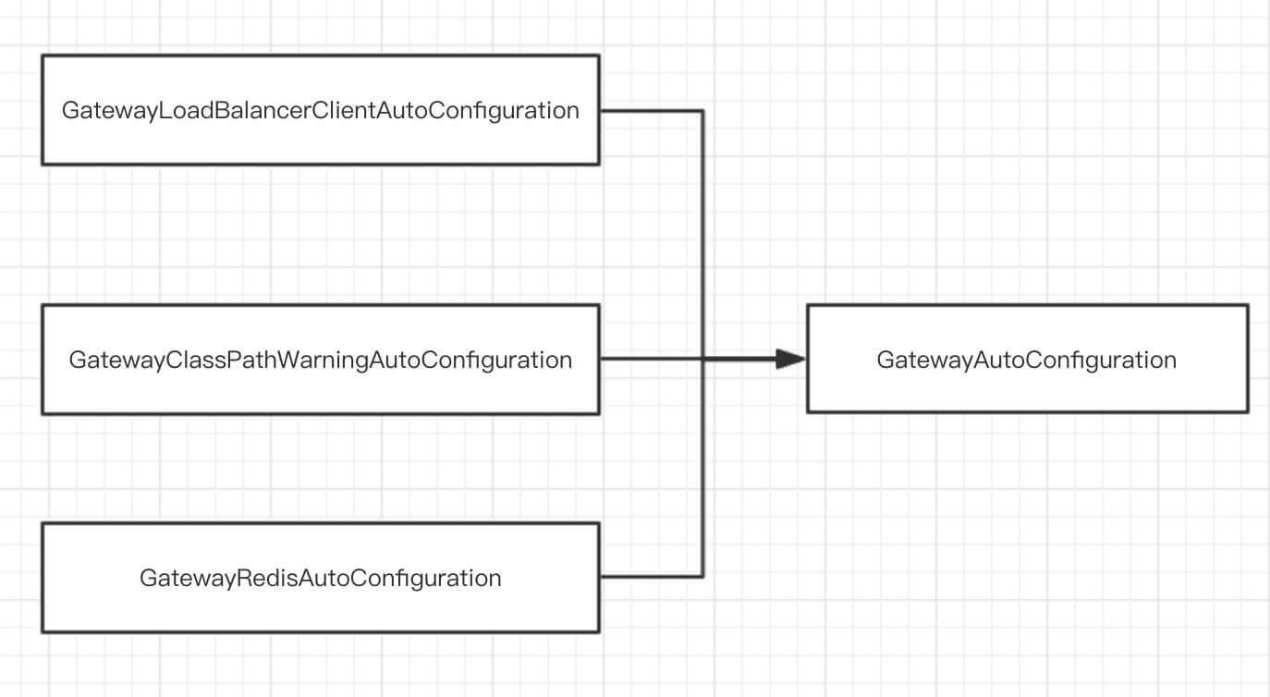
这时整个gateway的模块已经安装到了DispatcherHandler中, gateway的初始化算是完成了

#### 2.四个核心配置类

初始化代码：



初始化图：



* GatewayAutoConfiguration （核心配置类）

@ConditionalOnProperty(name = "spring.cloud.gateway.enabled", matchIfMissing = true)

yml配置是否开始网关，matchIfMissing默认值：true开启

该类初始化GlobalFilter（全局过滤器），NettyConfiguration， GatewayProperties（外部化配置类，配置路由信息）等等

* GatewayClassPathWarningAutoConfiguration
* GatewayLoadBalancerClientAutoConfiguration
* GatewayRedisAutoConfiguration

GatewayAutoConfiguration 的类FilteringWebHandler初始化代码块

@Bean

public FilteringWebHandler filteringWebHandler(List<GlobalFilter> globalFilters) {

return new FilteringWebHandler(globalFilters);

}

public FilteringWebHandler(List<GlobalFilter> globalFilters) {

this.globalFilters = loadFilters(globalFilters);

}

/\*\*

这是FilteringWebHandler类的代码块

这个方法把上面这段代码收集了所有GlobalFilter其中包含框架默认提供的也包含我们自定义的, 进入到new方法可以看到在初始化的时候会根据order的值来进行排序

\*/

private static List<GatewayFilter> loadFilters(List<GlobalFilter> filters) {

return filters.stream().map(filter -> {

GatewayFilterAdapter gatewayFilter = new GatewayFilterAdapter(filter);

if (filter instanceof Ordered) {

int order = ((Ordered) filter).getOrder();

return new OrderedGatewayFilter(gatewayFilter, order);

}

return gatewayFilter;

}).collect(Collectors.toList());

备注：类的加载

@ConditionalOnBean //当给定的在bean存在时,则实例化当前Bean

@ConditionalOnMissingBean // 当给定的在bean不存在时,则实例化当前Bean

@ConditionalOnClass // 当给定的类名在类路径上存在，则实例化当前Bean

@ConditionalOnMissingClass// 当给定的类名在类路径上不存在，则实例化当前Bean

#### 3.Route

* Route 构建的原理

核心成员属性

private final String id;

private final URI uri; //即客户端请求最终被转发的目的地

private final int order;//多个Route之间排序，数值越小排序越靠前,匹配优先级越高

private final AsyncPredicate<ServerWebExchange> predicate;

//Route 的前置条件，即满足相应的条件才会被路由到目的地 uri

private final List<GatewayFilter> gatewayFilters;//过滤器用于处理切面逻辑，如路由转发前修改请求头等。

private final Map<String, Object> metad

后续这里细节还需再次补充