# **Spring Cloud**

**目录**

**[1.特点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_features)**

**[一.云原生应用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_native_applications)**

**[2. Spring Cloud上下文：应用程序上下文服务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_context_application_context_services)**

**[2.1.Bootstrap应用程序上下文](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_the_bootstrap_application_context)**

**[2.2.应用程序上下文层次结构](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_application_context_hierarchies)**

**[2.3.更改引导程序Properties的位置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "customizing-bootstrap-properties)**

**[2.4.覆盖远程Properties的值](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "overriding-bootstrap-properties)**

**[2.5.自定义引导程序配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_the_bootstrap_configuration)**

**[2.6.自定义Bootstrap Property源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "customizing-bootstrap-property-sources)**

**[2.7.记录配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_logging_configuration)**

**[2.8.环境变化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_environment_changes)**

**[2.9.刷新范围](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "refresh-scope)**

**[2.10.加密与解密](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_encryption_and_decryption)**

**[2.11.终点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_endpoints)**

**[3. Spring Cloud Commons：通用抽象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_commons_common_abstractions)**

**[3.1.@EnableDiscoveryClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_enablediscoveryclient)**

[3.1.1.健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_health_indicator)

[3.1.2.订购DiscoveryClient实例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ordering_discoveryclient_instances)

**[3.2.服务注册](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serviceregistry)**

[3.2.1.ServiceRegistry自动注册](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serviceregistry_auto_registration)

[ServiceRegistry自动注册Events](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serviceregistry_auto_registration_events)

[3.2.2.服务注册表执行器端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_registry_actuator_endpoint)

**[3.3.Spring RestTemplate作为负载均衡器客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_resttemplate_as_a_load_balancer_client)**

**[3.4.Spring WebClient作为负载均衡器客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_webclient_as_a_load_balancer_client)**

[3.4.1.重试失败的请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retrying_failed_requests)

**[3.5.多个RestTemplate对象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_resttemplate_objects)**

**[3.6.多个WebClient对象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_webclient_objects)**

**[3.7.Spring WebFlux WebClient作为负载均衡器客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "loadbalanced-webclient)**

[3.7.1.Spring具有响应式负载均衡器的WebFlux WebClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "webflux-with-reactive-loadbalancer)

[3.7.2.Spring WebFlux WebClient，带有非反应式负载均衡器客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_webflux_webclient_with_non_reactive_load_balancer_client)

[3.7.3.传递自己的负载均衡客户端配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_passing_your_own_load_balancer_client_configuration)

**[3.8.忽略网络接口](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "ignore-network-interfaces)**

**[3.9.HTTP客户端工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "http-clients)**

**[3.10.启用的功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "enabled-features)**

[3.10.1.功能类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feature_types)

[3.10.2.声明功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_declaring_features)

**[3.11.Spring Cloud兼容性验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_compatibility_verification)**

**[二.Spring Cloud Config](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_config)**

**[4.快速入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start)**

**[4.1.客户端使用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_side_usage)**

**[5. Spring Cloud Config服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_config_server)**

**[5.1.环境Repository](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_environment_repository)**

[5.1.1.Git后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_git_backend)

[跳过SSL证书验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_skipping_ssl_certificate_validation)

[设置HTTP连接超时](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setting_http_connection_timeout)

[Git URI中的占位符](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_placeholders_in_git_uri)

[模式匹配和多个Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pattern_matching_and_multiple_repositories)

[认证方式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_authentication)

[使用AWS CodeCommit进行身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_authentication_with_aws_codecommit)

[使用属性进行Git SSH配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_git_ssh_configuration_using_properties)

[Git搜索路径中的占位符](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_placeholders_in_git_search_paths)

[强制拉动Git Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_force_pull_in_git_repositories)

[删除Git Repositories中未跟踪的分支](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deleting_untracked_branches_in_git_repositories)

[Git刷新率](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_git_refresh_rate)

[5.1.2.版本控制后端文件系统使用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_version_control_backend_filesystem_use)

[5.1.3.文件系统后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_file_system_backend)

[5.1.4.Vault后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault-backend)

[多个Properties来源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_properties_sources)

[5.1.5.通过代理访问后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_accessing_backends_through_a_proxy)

[5.1.6.与所有应用程序共享配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sharing_configuration_with_all_applications)

[基于文件的Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-config-server-file-based-repositories)

[Vault服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-config-server-vault-server)

[CredHub服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_credhub_server)

[5.1.7.JDBC后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_jdbc_backend)

[5.1.8.CredHub后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_credhub_backend)

[OAuth 2.0](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_oauth_2_0)

[5.1.9.复合环境Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "composite-environment-repositories)

[定制复合环境Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_composite_environment_repositories)

[5.1.10.Property覆盖](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_property_overrides)

**[5.2.健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_health_indicator_2)**

**[5.3.安全](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_security)**

**[5.4.加密与解密](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_encryption_and_decryption_2)**

**[5.5.密钥管理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_key_management)**

**[5.6.创建密钥库进行测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_a_key_store_for_testing)**

**[5.7.使用多个按键和按键旋转](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_multiple_keys_and_key_rotation)**

**[5.8.提供加密的Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serving_encrypted_properties)**

**[6.提供其他格式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serving_alternative_formats)**

**[7.提供纯文本](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serving_plain_text)**

**[8.嵌入配置服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_embedding_the_config_server)**

**[9.推送通知和Spring Cloud Bus](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_push_notifications_and_spring_cloud_bus)**

**[10. Spring Cloud Config客户](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_config_client)**

**[10.1.配置第一个引导程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "config-first-bootstrap)**

**[10.2.发现第一引导](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "discovery-first-bootstrap)**

**[10.3.快速配置客户端失败](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "config-client-fail-fast)**

**[10.4.配置客户端重试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "config-client-retry)**

**[10.5.查找远程配置资源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_locating_remote_configuration_resources)**

**[10.6.为配置服务器指定多个URL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_specifying_multiple_urls_for_the_config_server)**

**[10.7.配置超时](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuring_timeouts)**

**[10.8.安全](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_security_2)**

[10.8.1.健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_health_indicator_3)

[10.8.2.提供自定义的RestTemplate](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "custom-rest-template)

[10.8.3.Vault](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_vault)

**[10.9.Vault中的嵌套键](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_nested_keys_in_vault)**

**[三.Spring Cloud Netflix](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_netflix)**

**[11.服务发现：Eureka个客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_discovery_eureka_clients)**

**[11.1.如何包括Eureka客户](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-eureka-client-starter)**

**[11.2.向Eureka注册](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_with_eureka)**

**[11.3.通过Eureka服务器进行身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_authenticating_with_the_eureka_server)**

**[11.4.状态页和健康指示器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_status_page_and_health_indicator)**

**[11.5.注册安全的应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_a_secure_application)**

**[11.6.Eureka的健康检查](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_eurekas_health_checks)**

**[11.7.Eureka实例和客户端的元数据](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_eureka_metadata_for_instances_and_clients)**

[11.7.1.在Cloud Foundry上使用Eureka](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_eureka_on_cloud_foundry)

[11.7.2.在AWS上使用Eureka](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_eureka_on_aws)

[11.7.3.更改Eureka实例ID](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_changing_the_eureka_instance_id)

**[11.8.使用EurekaClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_eurekaclient)**

[11.8.1.没有Jersey的EurekaClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_eurekaclient_without_jersey)

**[11.9.本地Netflix EurekaClient的替代产品](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_alternatives_to_the_native_netflix_eurekaclient)**

**[11.10.为什么注册服务这么慢？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_why_is_it_so_slow_to_register_a_service)**

**[11.11.区域](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zones)**

**[11.12.刷新Eureka个客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_refreshing_eureka_clients)**

**[12.服务发现：Eureka服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server)**

**[12.1.如何包括Eureka服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-eureka-server-starter)**

**[12.2.如何运行Eureka服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-running-eureka-server)**

**[12.3.高可用性，区域和区域](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server-zones-and-regions)**

**[12.4.独立模式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server-standalone-mode)**

**[12.5.同行意识](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server-peer-awareness)**

**[12.6.何时首选IP地址](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server-prefer-ip-address)**

**[12.7.保护Eureka服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_securing_the_eureka_server)**

**[12.8.JDK 11支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_jdk_11_support)**

**[13.断路器：Hystrix个客户](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_circuit_breaker_hystrix_clients)**

**[13.1.如何包含Hystrix](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_include_hystrix)**

**[13.2.传播安全上下文或使用Spring范围](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-hystrix-starter)**

**[13.3.健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_health_indicator_4)**

**[13.4.Hystrix指标流](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_hystrix_metrics_stream)**

**[14.断路器：Hystrix仪表板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_circuit_breaker_hystrix_dashboard)**

**[15. Hystrix超时和Ribbon客户](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_hystrix_timeouts_and_ribbon_clients)**

**[15.1.如何包括Hystrix信息中心](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-hystrix-dashboard-starter)**

**[15.2.Turbine](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_turbine)**

[15.2.1.集群端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_clusters_endpoint)

**[15.3.Turbine Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_turbine_stream)**

**[16.客户端负载平衡器：Ribbon](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon)**

**[16.1.如何包含Ribbon](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-ribbon-starter)**

**[16.2.自定义Ribbon客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_the_ribbon_client)**

**[16.3.为所有Ribbon客户自定义默认值](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_the_default_for_all_ribbon_clients)**

**[16.4.通过设置Properties自定义Ribbon客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_the_ribbon_client_by_setting_properties)**

**[16.5.将Ribbon与Eureka一起使用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_ribbon_with_eureka)**

**[16.6.示例：如何在没有Eureka的情况下使用Ribbon](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon-without-eureka)**

**[16.7.示例：在Ribbon中禁用Eureka](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example_disable_eureka_use_in_ribbon)**

**[16.8.直接使用Ribbon API](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_ribbon_api_directly)**

**[16.9.缓存Ribbon配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "ribbon-child-context-eager-load)**

**[16.10.如何配置Hystrix线程池](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "how-to-configure-hystrix-thread-pools)**

**[16.11.如何提供Ribbon的IRule的密钥](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "how-to-provdie-a-key-to-ribbon)**

**[17.外部配置：Archaius](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_external_configuration_archaius)**

**[18.路由器和过滤器：Zuul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_router_and_filter_zuul)**

**[18.1.如何包含Zuul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-zuul-starter)**

**[18.2.嵌入式Zuul反向代理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-zuul-reverse-proxy)**

**[18.3.Zuul Http客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul_http_client)**

**[18.4.Cookie和敏感标题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cookies_and_sensitive_headers)**

**[18.5.忽略标题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ignored_headers)**

**[18.6.管理端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_management_endpoints)**

[18.6.1.路线终点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_routes_endpoint)

[18.6.2.过滤端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_filters_endpoint)

**[18.7.勒索模式和局部Forwards](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_strangulation_patterns_and_local_forwards)**

**[18.8.通过Zuul上传文件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_uploading_files_through_zuul)**

**[18.9.查询字符串编码](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_string_encoding)**

**[18.10.请求URI编码](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_request_uri_encoding)**

**[18.11.普通嵌入式Zuul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_plain_embedded_zuul)**

**[18.12.禁用Zuul过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_disable_zuul_filters)**

**[18.13.提供路线的Hystrix后备广告](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "hystrix-fallbacks-for-routes)**

**[18.14.Zuul超时](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul_timeouts)**

**[18.15.重写Location标头](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-redirect-location-rewrite)**

**[18.16.启用跨源请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_enabling_cross_origin_requests)**

**[18.17.指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_metrics)**

**[18.18.Zuul开发人员指南](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide)**

[18.18.1.Zuul Servlet](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_the_zuul_servlet)

[18.18.2.Zuul RequestContext](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul_requestcontext)

[18.18.3.@EnableZuulProxy与@EnableZuulServer](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_enablezuulproxy_vs_enablezuulserver)

[18.18.4.@EnableZuulServer过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-enable-filters)

[18.18.5.@EnableZuulProxy过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_enablezuulproxy_filters)

[18.18.6.自定义Zuul过滤器示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_zuul_filter_examples)

[如何编写前置过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-pre-filter)

[如何编写路由过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-route-filter)

[如何编写帖子过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-post-filter)

[18.18.7.Zuul错误如何工作](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_zuul_errors_work)

[18.18.8.Zuul急切的应用程序上下文加载](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul_eager_application_context_loading)

**[19. Polyglot支持Sidecar](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_polyglot_support_with_sidecar)**

**[20.重试失败的请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "retrying-failed-requests)**

**[20.1.退避政策](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_backoff_policies)**

**[20.2.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration)**

[20.2.1.Zuul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul)

**[21. HTTP客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_clients)**

**[22.维护模式下的模块](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_modules_in_maintenance_mode)**

**[四.Spring Cloud OpenFeign](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_openfeign)**

**[23.声明式REST客户端：Feign](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign)**

**[23.1.如何包含Feign](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "netflix-feign-starter)**

**[23.2.覆盖Feign默认值](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign-overriding-defaults)**

**[23.3.手动创建Feign客户](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_feign_clients_manually)**

**[23.4.Feign Hystrix支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign-hystrix)**

**[23.5.Feign Hystrix后备](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign-hystrix-fallback)**

**[23.6.Feign和@Primary](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feign_and_primary)**

**[23.7.Feign继承支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign-inheritance)**

**[23.8.Feign请求/响应压缩](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feign_requestresponse_compression)**

**[23.9.Feign记录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feign_logging)**

**[23.10.Feign @QueryMap支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feign_querymap_support)**

**[23.11.故障排除](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_troubleshooting)**

[23.11.1.早期初始化错误](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_early_initialization_errors)

**[五. Spring Cloud Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_stream)**

**[24. Spring的数据集成之旅简史](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_a_brief_history_of_springs_data_integration_journey)**

**[25.快速入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start_2)**

**[25.1.使用Spring Initializr创建示例应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-creating-sample-application)**

**[25.2.将项目导入到IDE中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-importing-project)**

**[25.3.添加消息处理程序，构建并运行](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-adding-message-handler)**

**[26. 2.0中有什么新功能？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_whats_new_in_2_0)**

**[26.1.新功能和组件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-new-features)**

**[26.2.显着增强](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-notable-enhancements)**

[26.2.1.执行器和Web依赖性现在都是可选的](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-actuator-web-dependencies)

[26.2.2.内容类型协商的改进](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-content-type-negotiation-improvements)

**[26.3.显着的弃用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-notable-deprecations)**

[26.3.1.Java序列化（Java本机和Kryo​​）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-deprecation-java-serialization)

[26.3.2.不推荐使用的类和方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-deprecation-classes-methods)

**[27.简介Spring Cloud Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-introducing)**

**[28.主要概念](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_main_concepts)**

**[28.1.应用模式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-application-model)**

[28.1.1.胖子](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_fat_jar)

**[28.2.Binder抽象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-binder-abstraction)**

**[28.3.持久的发布-订阅支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-persistent-publish-subscribe-support)**

**[28.4.消费群体](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "consumer-groups)**

**[28.5.消费者类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "consumer-types)**

[28.5.1.耐用性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "durability)

**[28.6.分区支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "partitioning)**

**[29.编程模型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_programming_model)**

**[29.1.目的地Binders](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_destination_binders)**

**[29.2.目标绑定](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_destination_bindings)**

**[29.3.生产和消费信息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-producing-consuming-messages)**

[29.3.1.Spring Integration支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_integration_support)

[29.3.2.使用@StreamListener注释](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_streamlistener_annotation)

[29.3.3.使用@StreamListener进行基于内容的路由](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_streamlistener_for_content_based_routing)

[29.3.4.Spring Cloud功能支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_function)

[功能组成](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_functional_composition)

[29.3.5.使用轮询的消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-streams-overview-using-polled-consumers)

[总览](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_overview)

[处理错误](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "polled-errors)

**[29.4.错误处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-error-handling)**

[29.4.1.应用错误处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_application_error_handling)

[29.4.2.系统错误处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_system_error_handling)

[删除失败的消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_drop_failed_messages)

[DLQ-死信队列](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dlq_dead_letter_queue)

[重新排队失败的消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_re_queue_failed_messages)

[29.4.3.重试模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_template)

**[29.5.反应式编程支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-reactive-programming-support)**

[29.5.1.基于Reactor的处理程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_reactor_based_handlers)

[29.5.2.反应源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_reactive_sources)

**[30. Binders](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-binders)**

**[30.1.生产者和消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producers_and_consumers)**

**[30.2.Binder SPI](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-binder-api)**

**[30.3.Binder检测](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binder_detection)**

[30.3.1.类路径检测](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_classpath_detection)

**[30.4.类路径上的多个Binders](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-binders)**

**[30.5.连接到多个系统](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-systems)**

**[30.6.绑定可视化和控制](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binding_visualization_and_control)**

**[30.7.Binder配置Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binder_configuration_properties)**

**[31.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_options)**

**[31.1.绑定服务Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binding_service_properties)**

**[31.2.绑定Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "binding-properties)**

[31.2.1.通用绑定Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_common_binding_properties)

[31.2.2.消费者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_properties)

[31.2.3.生产者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_properties)

**[31.3.使用动态绑定的目的地](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "dynamicdestination)**

**[32.内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management)**

**[32.1.机械学](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_mechanics)**

[32.1.1.内容类型与参数类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_content_type_versus_argument_type)

[32.1.2.讯息转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_message_converters)

**[32.2.提供的MessageConverters](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_provided_messageconverters)**

**[32.3.用户定义的消息转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-user-defined-message-converters)**

**[33. Schema进化支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "schema-evolution)**

**[33.1.Schema注册客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_schema_registry_client)**

[33.1.1.Schema注册机构客户Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_schema_registry_client_properties)

**[33.2.Avro Schema注册客户端消息转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_avro_schema_registry_client_message_converters)**

[33.2.1.Avro Schema注册表消息转换器Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_avro_schema_registry_message_converter_properties)

**[33.3.Apache Avro消息转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_avro_message_converters)**

**[33.4.支持Schema的转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_converters_with_schema_support)**

**[33.5.Schema注册服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_schema_registry_server)**

[33.5.1.Schema Registry Server API](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_schema_registry_server_api)

[注册新的Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-registering-new-schema)

[通过主题，格式和版本检索现有的Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-subject-format-version)

[通过主题和格式检索现有的Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-subject-format)

[通过ID检索现有的Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-id)

[按主题，格式和版本删除Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-subject-format-version)

[按ID删除Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-id)

[按主题删除Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-subject)

[33.5.2.使用Confluent的Schema注册表](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_confluents_schema_registry)

**[33.6.Schema注册和解决](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_schema_registration_and_resolution)**

[33.6.1.Schema注册流程（序列化）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-schema-registration-process)

[33.6.2.Schema解析过程（反序列化）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-schema-resolution-process)

**[34.应用程序间通信](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_inter_application_communication)**

**[34.1.连接多个应用程序实例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-connecting-multiple-application-instances)**

**[34.2.实例索引和实例计数](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count)**

**[34.3.分区](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-partitioning)**

[34.3.1.配置输出绑定以进行分区](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-configuring-output-bindings-partitioning)

[34.3.2.配置输入绑定以进行分区](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-configuring-input-bindings-partitioning)

**[35.测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_testing)**

**[35.1.禁用测试Binder自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_disabling_the_test_binder_autoconfiguration)**

**[36.健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_health_indicator_5)**

**[37.指标发射器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-metrics-emitter)**

**[38.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_samples)**

**[38.1.在CloudFoundry上部署流应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deploying_stream_applications_on_cloudfoundry)**

**[六.Binder实现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binder_implementations)**

**[39. Apache Kafka Binder](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_kafka_binder)**

**[39.1.用法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_usage)**

**[39.2.Apache Kafka Binder概述](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_kafka_binder_overview)**

**[39.3.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_options_2)**

[39.3.1.Kafka Binder Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kafka_binder_properties)

[39.3.2.Kafka消费者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-consumer-properties)

[39.3.3.Kafka生产者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-producer-properties)

[39.3.4.用法示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_usage_examples)

[示例：将autoCommitOffset设置为false并依靠手动进行](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example_setting_autocommitoffset_to_false_and_relying_on_manual_acking)

[示例：安全配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example_security_configuration)

[示例：暂停和恢复使用方](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pause-resume)

**[39.4.错误通道](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-error-channels)**

**[39.5.Kafka指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-metrics)**

**[39.6.Dead-Letter主题处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-dlq-processing)**

**[39.7.用Kafka Binder进行分区](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partitioning_with_the_kafka_binder)**

**[40. Apache Kafka流Binder](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_kafka_streams_binder)**

**[40.1.用法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_usage_2)**

**[40.2.Kafka流Binder概述](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kafka_streams_binder_overview)**

[40.2.1.流DSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_streams_dsl)

**[40.3.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_options_3)**

[40.3.1.Kafka流Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kafka_streams_properties)

[40.3.2.TimeWindow属性：](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_timewindow_properties)

**[40.4.多个输入绑定](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_input_bindings)**

[40.4.1.多个输入绑定作为接收器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_input_bindings_as_a_sink)

[40.4.2.多个输入绑定作为处理器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_input_bindings_as_a_processor)

**[40.5.多个输出绑定（又名分支）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_output_bindings_aka_branching)**

**[40.6.讯息转换](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_message_conversion)**

[40.6.1.出站序列化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_outbound_serialization)

[40.6.2.入站反序列化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_inbound_deserialization)

**[40.7.错误处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_error_handling)**

[40.7.1.处理反序列化异常](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_handling_deserialization_exceptions)

[40.7.2.处理非反序列化异常](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_handling_non_deserialization_exceptions)

**[40.8.国营商店](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_state_store)**

**[40.9.互动查询](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_interactive_queries)**

**[40.10.访问基础的KafkaStreams对象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_accessing_the_underlying_kafkastreams_object)**

**[40.11.状态清理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_state_cleanup)**

**[41. RabbitMQ Binder](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rabbitmq_binder)**

**[41.1.用法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_usage_3)**

**[41.2.RabbitMQ Binder概述](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rabbitmq_binder_overview)**

**[41.3.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_options_4)**

[41.3.1.RabbitMQ Binder Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-binder-properties)

[41.3.2.RabbitMQ消费者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rabbitmq_consumer_properties)

[41.3.3.高级侦听器容器配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_advanced_listener_container_configuration)

[41.3.4.Rabbit生产者Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rabbit_producer_properties)

**[41.4.重试RabbitMQ Binder](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_with_the_rabbitmq_binder)**

[41.4.1.放在一起](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_putting_it_all_together)

**[41.5.错误通道](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-error-channels)**

**[41.6.Dead-Letter队列处理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-dlq-processing)**

[41.6.1.非分区目的地](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_non_partitioned_destinations)

[41.6.2.分区目的地](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partitioned_destinations)

[republishToDlq=false](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_republishtodlqfalse)

[republishToDlq=true](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_republishtodlqtrue)

**[41.7.用RabbitMQ Binder分区](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partitioning_with_the_rabbitmq_binder)**

**[七.Spring Cloud Bus](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_bus)**

**[42.快速入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start_3)**

**[43. Bus端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_bus_endpoints)**

**[43.1.Bus刷新端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_bus_refresh_endpoint)**

**[43.2.Bus环保端点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_bus_env_endpoint)**

**[44.寻址实例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_addressing_an_instance)**

**[45.解决服务的所有实例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_addressing_all_instances_of_a_service)**

**[46.服务ID必须唯一](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_id_must_be_unique)**

**[47.定制消息代理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_the_message_broker)**

**[48.追踪Bus Events](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tracing_bus_events)**

**[49.广播自己的Events](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_broadcasting_your_own_events)**

**[49.1.在自定义包中注册事件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_events_in_custom_packages)**

**[八.Spring Cloud Sleuth](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_sleuth)**

**[50.引言](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_introduction)**

**[50.1.术语](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_terminology)**

**[50.2.目的](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_purpose)**

[50.2.1.Zipkin的分布式跟踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_distributed_tracing_with_zipkin)

[50.2.2.可视化错误](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_visualizing_errors)

[50.2.3.勇敢的分布式跟踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_distributed_tracing_with_brave)

[50.2.4.现场例子](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_live_examples)

[50.2.5.日志关联](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_log_correlation)

[使用Logstash进行JSON Logback](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_json_logback_with_logstash)

[50.2.6.传播Span上下文](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_propagating_span_context)

[行李与Span标签](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_baggage_versus_span_tags)

**[50.3.将侦探添加到项目中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "sleuth-adding-project)**

[50.3.1.仅侦探（对数相关）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_only_sleuth_log_correlation)

[50.3.2.通过HTTP与Zipkin一起侦听](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sleuth_with_zipkin_via_http)

[50.3.3.侦探，其中Zipkin高于RabbitMQ或Kafka](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sleuth_with_zipkin_over_rabbitmq_or_kafka)

**[50.4.覆盖Zipkin的自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_overriding_the_auto_configuration_of_zipkin)**

**[51.其他资源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_additional_resources)**

**[52.特点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_features_2)**

**[52.1.勇敢简介](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_introduction_to_brave)**

[52.1.1.追踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tracing)

[52.1.2.本地追踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_local_tracing)

[52.1.3.自定义跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_spans)

[52.1.4.隐式查找当前的Span](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_implicitly_looking_up_the_current_span)

[52.1.5.RPC追踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rpc_tracing)

[单向跟踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_one_way_tracing)

**[53.抽样](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sampling)**

**[53.1.声明式抽样](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_declarative_sampling)**

**[53.2.定制采样](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_sampling)**

**[53.3.在Spring Cloud Sleuth中采样](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sampling_in_spring_cloud_sleuth)**

**[54.传播](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_propagation)**

**[54.1.传播额外的字段](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_propagating_extra_fields)**

[54.1.1.前缀字段](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "prefixed-fields)

[54.1.2.提取传播的上下文](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_extracting_a_propagated_context)

[54.1.3.在客户端和服务器之间共享范围ID](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sharing_span_ids_between_client_and_server)

[54.1.4.实施传播](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_implementing_propagation)

**[55.当前跟踪组件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_current_tracing_component)**

**[56.当前Span](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_current_span)**

**[56.1.手动设置范围](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setting_a_span_in_scope_manually)**

**[57.仪器仪表](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_instrumentation)**

**[58. Span生命周期](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_span_lifecycle)**

**[58.1.创建并完成spans](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "creating-and-finishing-spans)**

**[58.2.连续跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "continuing-spans)**

**[58.3.使用显式父级创建Span](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "creating-spans-with-explicit-parent)**

**[59.命名spans](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_naming_spans)**

**[59.1.@SpanName注释](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spanname_annotation)**

**[59.2.toString()方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tostring_method)**

**[60.使用Annotations管理跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_managing_spans_with_annotations)**

**[60.1.基本原理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rationale)**

**[60.2.创建新的跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_new_spans)**

**[60.3.连续跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_continuing_spans)**

**[60.4.进阶标签设定](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_advanced_tag_setting)**

[60.4.1.定制提取器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_extractor)

[60.4.2.为值解析表达式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_resolving_expressions_for_a_value)

[60.4.3.使用toString()方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_tostring_method)

**[61.定制](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizations)**

**[61.1.定制者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizers)**

**[61.2.HTTP](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http)**

**[61.3.TracingFilter](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tracingfilter)**

**[61.4.RPC](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rpc)**

**[61.5.定制服务名称](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_service_name)**

**[61.6.自定义报告的跨度](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customization_of_reported_spans)**

**[61.7.主机定位器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_host_locator)**

**[62.将跨度发送到Zipkin](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sending_spans_to_zipkin)**

**[63. Zipkin流Span消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zipkin_stream_span_consumer)**

**[64.整合](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_integrations)**

**[64.1.开放追踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_opentracing)**

**[64.2.可运行和可调用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_runnable_and_callable)**

**[64.3.Hystrix](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_hystrix)**

[64.3.1.自定义并发策略](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_concurrency_strategy)

[64.3.2.手动指令设定](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_manual_command_setting)

**[64.4.RxJava](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rxjava)**

**[64.5.HTTP整合](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_integration)**

[64.5.1.HTTP过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_filter)

[64.5.2.处理程序拦截器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_handlerinterceptor)

[64.5.3.异步Servlet支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_async_servlet_support)

[64.5.4.WebFlux支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_webflux_support)

[64.5.5.Dubbo RPC支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dubbo_rpc_support)

**[64.6.HTTP客户端集成](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_client_integration)**

[64.6.1.同步休息模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_synchronous_rest_template)

[64.6.2.异步休息模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_asynchronous_rest_template)

[多个异步休息模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_asynchronous_rest_templates)

[64.6.3.WebClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_webclient)

[64.6.4.特拉弗森](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_traverson)

[64.6.5.Apache HttpClientBuilder和HttpAsyncClientBuilder](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_httpclientbuilder_and_httpasyncclientbuilder)

[64.6.6.净值HttpClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_netty_httpclient)

[64.6.7.UserInfoRestTemplateCustomizer](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_userinforesttemplatecustomizer)

**[64.7.Feign](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_feign)**

**[64.8.gRPC](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_grpc)**

[64.8.1.变体1](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_variant_1)

[依存关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dependencies)

[服务器检测](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_server_instrumentation)

[客户端工具](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_instrumentation)

[64.8.2.变体2](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_variant_2)

**[64.9.异步通讯](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_asynchronous_communication)**

[64.9.1.@Async带注释的方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_async_annotated_methods)

[64.9.2.@Scheduled带注释的方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scheduled_annotated_methods)

[64.9.3.Executor，ExecutorService和ScheduledExecutorService](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_executor_executorservice_and_scheduledexecutorservice)

[定制执行者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customization_of_executors)

**[64.10.讯息传递](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_messaging)**

[64.10.1.Spring Integration和Spring Cloud Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_integration_and_spring_cloud_stream)

[64.10.2.Spring RabbitMq](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_rabbitmq)

[64.10.3.Spring Kafka](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_kafka)

[64.10.4.Spring JMS](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_jms)

**[64.11.Zuul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_zuul_2)**

**[64.12.项目Reactor](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_project_reactor)**

**[65.运行示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_running_examples)**

**[九.Spring Cloud Consul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_consul)**

**[66.安装Consul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-install)**

**[67. Consul Agent](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-agent)**

**[68.使用Consul进行服务发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-discovery)**

**[68.1.如何激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_activate)**

**[68.2.向Consul注册](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_with_consul)**

[68.2.1.将管理注册为单独的服务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_management_as_a_separate_service)

**[68.3.HTTP运行状况检查](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_health_check)**

[68.3.1.元数据和Consul标签](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_metadata_and_consul_tags)

[68.3.2.使Consul实例ID唯一](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_making_the_consul_instance_id_unique)

[68.3.3.将标头应用于健康检查请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_applying_headers_to_health_check_requests)

**[68.4.查找服务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_looking_up_services)**

[68.4.1.使用Ribbon](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_ribbon)

[68.4.2.使用DiscoveryClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_discoveryclient)

**[68.5.Consul目录观看](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consul_catalog_watch)**

**[69.使用Consul的分布式配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config)**

**[69.1.如何激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_activate_2)**

**[69.2.客制化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing)**

**[69.3.配置监视](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config-watch)**

**[69.4.YAML或带有配置的Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config-format)**

**[69.5.git2consul与配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config-git2consul)**

**[69.6.快速失败](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-failfast)**

**[70. Consul重试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-retry)**

**[71. Spring Cloud Bus与Consul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-bus)**

**[71.1.如何激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_activate_3)**

**[72.带有Hystrix的断路器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-hystrix)**

**[73. Hystrix通过Turbine和Consul进行指标汇总](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-turbine)**

**[十.Spring Cloud Zookeeper](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_zookeeper)**

**[74.安装Zookeeper](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-install)**

**[75.使用Zookeeper进行服务发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-discovery)**

**[75.1.激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_activating)**

**[75.2.向Zookeeper注册](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_with_zookeeper)**

**[75.3.使用DiscoveryClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_discoveryclient_2)**

**[76.将Spring Cloud Zookeeper与Spring Cloud Netflix组件一起使用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-netflix)**

**[76.1.Ribbon与Zookeeper](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ribbon_with_zookeeper)**

**[77. Spring Cloud Zookeeper和服务注册中心](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-service-registry)**

**[77.1.实例状态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_instance_status)**

**[78. Zookeeper依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies)**

**[78.1.使用Zookeeper依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-using)**

**[78.2.激活Zookeeper依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-activating)**

**[78.3.设置Zookeeper依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-setting-up)**

[78.3.1.别名](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-setting-up-aliases)

[78.3.2.路径](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_path)

[78.3.3.负载均衡器类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_load_balancer_type)

[78.3.4.Content-Type模板和版本](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_content_type_template_and_version)

[78.3.5.默认标题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_default_headers)

[78.3.6.必需的依赖关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_required_dependencies)

[78.3.7.存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stubs)

**[78.4.配置Spring Cloud Zookeeper依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-configuring)**

**[79. Spring Cloud Zookeeper依赖性观察者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependency-watcher)**

**[79.1.激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_activating_2)**

**[79.2.注册听众](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_a_listener)**

**[79.3.使用状态检查器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependency-watcher-presence-checker)**

**[80.使用Zookeeper的分布式配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-config)**

**[80.1.激活](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_activating_3)**

**[80.2.客制化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customizing_2)**

**[80.3.访问控制列表（ACL）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_access_control_lists_acls)**

**[十一.Spring Cloud Security](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_security)**

**[81.快速入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quickstart)**

**[81.1.OAuth2单点登录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_oauth2_single_sign_on)**

**[81.2.OAuth2受保护的资源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_oauth2_protected_resource)**

**[82.更多细节](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_more_detail)**

**[82.1.单点登录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_single_sign_on)**

**[82.2.令牌中继](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_token_relay)**

[82.2.1.Spring Cloud网关中的客户端令牌中继](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_token_relay_in_spring_cloud_gateway)

[82.2.2.客户令牌中继](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_token_relay)

[82.2.3.Zuul代理中的客户端令牌中继](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_token_relay_in_zuul_proxy)

[82.2.4.资源服务器令牌中继](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_resource_server_token_relay)

**[83.配置Zuul代理的下游身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuring_authentication_downstream_of_a_zuul_proxy)**

**[十二.Spring Cloud for Cloud Foundry](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_for_cloud_foundry)**

**[84.发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_discovery)**

**[85.单点登录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_single_sign_on_2)**

**[十三.Spring Cloud Contract](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract)**

**[86. Spring Cloud Contract](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_2)**

**[87. Spring Cloud Contract验证程序简介](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_verifier_introduction)**

**[87.1.历史](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_history)**

**[87.2.为什么要签约验证人？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_why_a_contract_verifier)**

[87.2.1.测试问题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_testing_issues)

**[87.3.目的](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_purposes)**

**[87.4.这个怎么运作](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_it_works)**

[87.4.1.三秒游](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour)

[在生产者方面](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour-producer)

[在消费者方面](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour-consumer)

[87.4.2.三分钟游](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour)

[在生产者方面](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour-producer)

[在消费者方面](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour-consumer)

[87.4.3.定义合同](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_defining_the_contract)

[87.4.4.客户端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_side)

[87.4.5.服务器端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_server_side)

**[87.5.消费者驱动Contracts（CDC）分步指南](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_step_by_step_guide_to_consumer_driven_contracts_cdc)**

[87.5.1.技术说明](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_technical_note)

[87.5.2.消费者方（贷款发行）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_side_loan_issuance)

[87.5.3.生产者方（欺诈检测服务器）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_side_fraud_detection_server)

[87.5.4.消费者方（贷款发行）最后一步](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_side_loan_issuance_final_step)

**[87.6.依存关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dependencies_2)**

**[87.7.附加链接](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_additional_links)**

[87.7.1.Spring Cloud Contract个视频](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_video)

[87.7.2.读物](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_readings)

**[87.8.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_samples_2)**

**[88. Spring Cloud Contract常见问题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_faq)**

**[88.1.为什么使用Spring Cloud Contract验证程序而不使用X？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_why_use_spring_cloud_contract_verifier_and_not_x)**

**[88.2.我不想在Groovy中写合同！](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_i_dont_want_to_write_a_contract_in_groovy)**

**[88.3.这个值是什么（consumer（），producer（））？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_what_is_this_valueconsumer_producer)**

**[88.4.如何进行存根版本控制？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_do_stubs_versioning)**

[88.4.1.API版本控制](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_api_versioning)

[88.4.2.JAR版本控制](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_jar_versioning)

[88.4.3.开发或产品存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dev_or_prod_stubs)

**[88.5.合同共同回购](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_common_repo_with_contracts)**

[88.5.1.回购结构](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_repo_structure)

[88.5.2.工作流程](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_workflow)

[88.5.3.消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer)

[88.5.4.制片人](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer)

[88.5.5.如何为每个主题而不是每个生产者定义消息传递合同？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_can_i_define_messaging_contracts_per_topic_not_per_producer)

[对于Maven项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_for_maven_project)

[对于Gradle项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_for_gradle_project)

**[88.6.我需要二进制存储吗？我不能使用Git吗？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_do_i_need_a_binary_storage_cant_i_use_git)**

[88.6.1.协议约定](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_protocol_convention)

[88.6.2.制片人](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_2)

[88.6.3.生产者，合同存储在本地](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_with_contracts_stored_locally)

[与生产者和存根之间的合同保持一致](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_keeping_contracts_with_the_producer_and_stubs_in_an_external_repository)

[88.6.4.消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_2)

**[88.7.我可以使用契约代理吗？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_can_i_use_the_pact_broker)**

[88.7.1.契约消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pact_consumer)

[88.7.2.制片人](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_3)

[88.7.3.契约消费者（生产者合同法）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pact_consumer_producer_contract_approach)

**[88.8.如何调试由生成的测试客户端发送的请求/响应？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_can_i_debug_the_requestresponse_being_sent_by_the_generated_tests_client)**

[88.8.1.如何调试WireMock发送的映射/请求/响应？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_can_i_debug_the_mappingrequestresponse_being_sent_by_wiremock)

[88.8.2.如何查看在HTTP服务器存根中注册了什么？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_can_i_see_what_got_registered_in_the_http_server_stub)

[88.8.3.我可以引用文件中的文本吗？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_can_i_reference_text_from_file)

**[89. Spring Cloud Contract验证程序设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_verifier_setup)**

**[89.1.Gradle项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-project)**

[89.1.1.先决条件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-prerequisites)

**[90.添加具有依赖性的Gradle插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-add-gradle-plugin)**

**[90.1.Gradle和“保证放心的2.0”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-and-rest-assured)**

**[90.2.Gradle的快照版本](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-snapshot-versions)**

**[90.3.添加存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-add-stubs)**

**[90.4.运行插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-run-plugin)**

**[90.5.默认设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-default-setup)**

**[90.6.配置插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-configure-plugin)**

**[90.7.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-configuration-options)**

**[90.8.所有测试的单一基类](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-single-base-class)**

**[90.9.Contracts的不同基类](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-different-base-classes)**

**[90.10.调用生成的测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-invoking-generated-tests)**

**[90.11.将存根推送到SCM](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-pushing-stubs-to-scm)**

**[90.12.Spring Cloud Contract消费者方面的验证者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-consumer)**

**[90.13.Maven项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-project)**

[90.13.1.添加Maven插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-add-plugin)

[90.13.2.Maven和“保证放心” 2.0](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-rest-assured)

[90.13.3.Maven的快照版本](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-snapshot-versions)

[90.13.4.添加存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-add-stubs)

[90.13.5.运行插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-run-plugin)

[90.13.6.配置插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-configure-plugin)

[90.13.7.配置选项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-configuration-options)

[90.13.8.所有测试的单一基类](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-single-base)

[90.13.9.合同的不同基本类别](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-different-base)

[90.13.10.调用生成的测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-invoking-generated-tests)

[90.13.11.将存根推送到SCM](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-pushing-stubs-to-scm)

[90.13.12.Maven插件和STS](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-sts)

[90.13.13.Maven具有Spock测试的插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_maven_plugin_with_spock_tests)

**[90.14.存根和传递依存关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stubs_and_transitive_dependencies)**

**[90.15.情境](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenarios)**

**[90.16.Docker项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "docker-project)**

[90.16.1.Maven，JAR和二进制存储的简短介绍](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_short_intro_to_maven_jars_and_binary_storage)

[90.16.2.这个怎么运作](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_it_works_2)

[环境变量](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_environment_variables)

[90.16.3.使用例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example_of_usage)

[90.16.4.服务器端（nodejs）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "docker-server-side)

**[91. Spring Cloud Contract验证者消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_verifier_messaging)**

**[91.1.整合方式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_integrations_2)**

**[91.2.手动集成测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_manual_integration_testing)**

**[91.3.发布方测试生成](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_publisher_side_test_generation)**

[91.3.1.方案1：无输入消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_1_no_input_message)

[91.3.2.方案2：由输入触发的输出](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_2_output_triggered_by_input)

[91.3.3.方案3：无输出消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_3_no_output_message)

**[91.4.消费者存根生成](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_stub_generation)**

**[92. Spring Cloud Contract Stub Runner](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_stub_runner)**

**[92.1.快照版本](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_snapshot_versions)**

**[92.2.将存根发布为JAR](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "publishing-stubs-as-jars)**

**[92.3.Stub Runner核心](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_core)**

[92.3.1.检索存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retrieving_stubs)

[存根下载](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_downloading)

[类路径扫描](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_classpath_scanning)

[配置HTTP服务器存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuring_http_server_stubs)

[92.3.2.正在运行的存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_running_stubs)

[使用主应用程序运行](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_running_using_main_app)

[HTTP存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_stubs)

[查看注册的映射](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_viewing_registered_mappings)

[信息存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_messaging_stubs)

**[92.4.Stub Runner JUnit规则和Stub Runner JUnit5扩展](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_junit_rule_and_stub_runner_junit5_extension)**

[92.4.1.Maven设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_maven_settings)

[92.4.2.提供固定端口](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_providing_fixed_ports)

[92.4.3.流利的API](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_fluent_api)

[92.4.4.Stub Runner与Spring](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_with_spring)

**[92.5.Stub Runner Spring Cloud](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_spring_cloud)**

[92.5.1.存根服务发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stubbing_service_discovery)

[测试配置文件和服务发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_test_profiles_and_service_discovery)

[92.5.2.附加配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_additional_configuration)

**[92.6.Stub Runner引导应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_boot_application)**

[92.6.1.如何使用它？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_use_it)

[Stub Runner服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_server)

[Stub Runner服务器胖子](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_server_fat_jar)

[Spring Cloud CLI](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_cli)

[92.6.2.终点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_endpoints_2)

[HTTP](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_2)

[讯息传递](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_messaging_2)

[92.6.3.例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example)

[92.6.4.Stub Runner使用服务发现启动](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_boot_with_service_discovery)

**[92.7.每个消费者存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stubs_per_consumer)**

**[92.8.共同](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_common)**

[92.8.1.JUnit和Spring的通用Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "common-properties-junit-spring)

[92.8.2.Stub Runner存根ID](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stub-runner-stub-ids)

**[92.9.Stub Runner Docker](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stubrunner-docker)**

[92.9.1.如何使用它](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_how_to_use_it_2)

[92.9.2.非JVM项目中的客户端用法示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_example_of_client_side_usage_in_a_non_jvm_project)

**[93. Stub Runner用于消息传递](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stub-runner-for-messaging)**

**[93.1.存根触发](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_triggering)**

[93.1.1.按标签触发](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-label)

[93.1.2.按组和工件ID触发](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-group-artifact-ids)

[93.1.3.由工件ID触发](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-artifact-ids)

[93.1.4.触发所有讯息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-all-messages)

**[93.2.Stub Runner Camel](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_camel)**

[93.2.1.将其添加到项目中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_adding_it_to_the_project)

[93.2.2.禁用功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_disabling_the_functionality)

[93.2.3.例子](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_examples)

[存根结构](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stubs_structure)

[方案1（无输入消息）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_1_no_input_message_2)

[场景2（由输入触发输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_2_output_triggered_by_input_2)

[方案3（输入无输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scenario_3_input_with_no_output)

**[93.3.Stub Runner整合](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_integration)**

[93.3.1.将跑步者添加到项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_adding_the_runner_to_the_project)

[93.3.2.禁用功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_disabling_the_functionality_2)

[方案1（无输入消息）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-1)

[场景2（由输入触发输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-2)

[方案3（输入无输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-3)

**[93.4.Stub Runner Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_stream)**

[93.4.1.将跑步者添加到项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_adding_the_runner_to_the_project_2)

[93.4.2.禁用功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_disabling_the_functionality_3)

[方案1（无输入消息）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-1)

[场景2（由输入触发输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-2)

[方案3（输入无输出）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-3)

**[93.5.Stub Runner Spring AMQP](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stub_runner_spring_amqp)**

[93.5.1.将跑步者添加到项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_adding_the_runner_to_the_project_3)

[触发消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_triggering_the_message)

[Spring AMQP测试配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_amqp_test_configuration)

**[94. Contract DSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl)**

**[94.1.局限性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_limitations)**

**[94.2.常见的顶级元素](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_common_top_level_elements)**

[94.2.1.描述](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-description)

[94.2.2.名称](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-name)

[94.2.3.忽略Contracts](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-ignoring-contracts)

[94.2.4.从文件传递值](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-passing-values-from-files)

[94.2.5.HTTP顶级元素](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-http-top-level-elements)

**[94.3.请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_request)**

**[94.4.响应](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_response)**

**[94.5.动态特性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dynamic_properties)**

[94.5.1.体内的动态特性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dynamic_properties_inside_the_body)

[94.5.2.常用表达](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_regular_expressions)

[94.5.3.传递可选参数](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_passing_optional_parameters)

[94.5.4.在服务器端执行自定义方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_executing_custom_methods_on_the_server_side)

[94.5.5.引用响应中的请求](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_referencing_the_request_from_the_response)

[94.5.6.注册自己的WireMock扩展](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_your_own_wiremock_extension)

[94.5.7.匹配器部分中的动态Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-matchers)

**[94.6.JAX-RS支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_jax_rs_support)**

**[94.7.异步支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_async_support)**

**[94.8.使用上下文路径](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_working_with_context_paths)**

**[94.9.使用WebFlux](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_working_with_webflux)**

[94.9.1.WebFlux与WebTestClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_webflux_with_webtestclient)

[94.9.2.WebFlux具有显式模式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_webflux_with_explicit_mode)

**[94.10.REST的XML支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_xml_support_for_rest)**

**[94.11.消息传递顶级元素](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_messaging_top_level_elements)**

[94.11.1.方法触发的输出](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-output-triggered-method)

[94.11.2.消息触发的输出](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-output-triggered-message)

[94.11.3.消费者/生产者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-consumer-producer)

[94.11.4.共同](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-common)

**[94.12.一个文件中多个Contracts](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_multiple_contracts_in_one_file)**

**[94.13.从合同中生成Spring REST Docs片段](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_generating_spring_rest_docs_snippets_from_the_contracts)**

**[95.定制化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customization)**

**[95.1.扩展DSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_extending_the_dsl)**

[95.1.1.普通JAR](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_common_jar)

[95.1.2.将依赖项添加到项目中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_adding_the_dependency_to_the_project)

[95.1.3.在项目的依赖关系中测试依赖关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_test_the_dependency_in_the_projects_dependencies)

[95.1.4.在插件的依赖项中测试依赖项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_test_a_dependency_in_the_plugins_dependencies)

[95.1.5.引用DSL中的类](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_referencing_classes_in_dsls)

**[96.使用可插拔架构](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_pluggable_architecture)**

**[96.1.定制合同转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_contract_converter)**

[96.1.1.契约转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pact-converter)

[96.1.2.契约合同](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pact_contract)

[96.1.3.生产者契约](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pact_for_producers)

[96.1.4.消费者契约](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pact_for_consumers)

**[96.2.使用自定义测试生成器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_custom_test_generator)**

**[96.3.使用自定义存根生成器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_custom_stub_generator)**

**[96.4.使用自定义Stub Runner](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_custom_stub_runner)**

**[96.5.使用自定义存根下载器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_the_custom_stub_downloader)**

**[96.6.使用SCM存根下载器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "scm-stub-downloader)**

**[96.7.使用契约存根下载器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pact-stub-downloader)**

**[97. Spring Cloud Contract WireMock](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_contract_wiremock)**

**[97.1.自动注册存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_registering_stubs_automatically)**

**[97.2.使用文件指定存根实体](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_using_files_to_specify_the_stub_bodies)**

**[97.3.替代方案：使用JUnit规则](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_alternative_using_junit_rules)**

**[97.4.放松模板的SSL验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_relaxed_ssl_validation_for_rest_template)**

**[97.5.WireMock和Spring MVC模拟](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_wiremock_and_spring_mvc_mocks)**

**[97.6.定制WireMock配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_customization_of_wiremock_configuration)**

**[97.7.使用REST文档生成存根](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_generating_stubs_using_rest_docs)**

**[97.8.使用REST文档生成Contracts](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_generating_contracts_by_using_rest_docs)**

**[98.迁移](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_migrations)**

**[98.1.1.0.x→1.1.x](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "cloud-verifier-1.0-1.1)**

[98.1.1.生成的存根的新结构](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_new_structure_of_generated_stubs)

**[98.2.1.1.x→1.2.x](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "cloud-verifier-1.1-1.2)**

[98.2.1.自定义HttpServerStub](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_httpserverstub)

[98.2.2.用于生成测试的新软件包](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_new_packages_for_generated_tests)

[98.2.3.TemplateProcessor中的新方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_new_methods_in_templateprocessor)

[98.2.4.RestAssured 3.0](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_restassured_3_0)

**[98.3.1.2.x→2.0.x](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "cloud-verifier-1.2-2.0)**

**[99.链接](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_links)**

**[十四.Spring Cloud Vault](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_vault)**

**[100.快速入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start_4)**

**[101.客户端使用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_client_side_usage_2)**

**[101.1.认证方式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_authentication_2)**

**[102.认证方式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication)**

**[102.1.令牌认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.token)**

**[102.2.AppId验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.appid)**

[102.2.1.自定义用户ID](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_userid)

**[102.3.AppRole身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_approle_authentication)**

**[102.4.AWS-EC2身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.awsec2)**

**[102.5.AWS-IAM身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.awsiam)**

**[102.6.Azure MSI身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.azuremsi)**

**[102.7.TLS证书认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.clientcert)**

**[102.8.隔间认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.cubbyhole)**

**[102.9.GCP-GCE认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.gcpgce)**

**[102.10.GCP-IAM认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.gcpiam)**

**[102.11.Kubernetes认证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.kubernetes)**

**[103.秘密后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends)**

**[103.1.通用后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.generic)**

**[103.2.版本化键值后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.kv.versioned)**

**[103.3.Consul](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.consul)**

**[103.4.RabbitMQ](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.rabbitmq)**

**[103.5.AWS](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.aws)**

**[104.数据库后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.database-backends)**

**[104.1.数据库](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.database)**

**[104.2.阿帕奇Cassandra](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.cassandra)**

**[104.3.MongoDB](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.mongodb)**

**[104.4.的MySQL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.mysql)**

**[104.5.PostgreSQL的](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.postgresql)**

**[105.配置PropertySourceLocator行为](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.configurer)**

**[106.服务注册表配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_registry_configuration)**

**[107. Vault客户端快速失败](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.fail-fast)**

**[108. Vault客户端SSL配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.ssl)**

**[109.租赁生命周期管理（续订和撤销）](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault-lease-renewal)**

**[十五.Spring Cloud网关](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_gateway)**

**[110.如何包括Spring Cloud网关](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-starter)**

**[111.词汇表](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_glossary)**

**[112.工作原理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-how-it-works)**

**[113.配置路由谓词工厂和网关过滤工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuring_route_predicate_factories_and_gateway_filter_factories)**

**[113.1.快捷方式配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_shortcut_configuration)**

**[113.2.完全展开的论点](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_fully_expanded_arguments)**

**[114.路由谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-request-predicates-factories)**

**[114.1.后路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_after_route_predicate_factory)**

**[114.2.路线谓词工厂之前](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_before_route_predicate_factory)**

**[114.3.路线谓词工厂之间](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_between_route_predicate_factory)**

**[114.4.Cookie路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cookie_route_predicate_factory)**

**[114.5.标头路由谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_header_route_predicate_factory)**

**[114.6.主机路由谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_host_route_predicate_factory)**

**[114.7.方法路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_method_route_predicate_factory)**

**[114.8.路径路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_path_route_predicate_factory)**

**[114.9.查询路由谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_route_predicate_factory)**

**[114.10.RemoteAddr路由谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_remoteaddr_route_predicate_factory)**

**[114.11.重量路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_weight_route_predicate_factory)**

[114.11.1.修改远程地址的解析方式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_modifying_the_way_remote_addresses_are_resolved)

**[115.网关过滤器工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gatewayfilter_factories)**

**[115.1.AddRequestHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_addrequestheader_gatewayfilter_factory)**

**[115.2.AddRequestParameter GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_addrequestparameter_gatewayfilter_factory)**

**[115.3.AddResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_addresponseheader_gatewayfilter_factory)**

**[115.4.DedupeResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deduperesponseheader_gatewayfilter_factory)**

**[115.5.Hystrix GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "hystrix)**

**[115.6.FallbackHeaders GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "fallback-headers)**

**[115.7.MapRequestHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_maprequestheader_gatewayfilter_factory)**

**[115.8.PrefixPath GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_prefixpath_gatewayfilter_factory)**

**[115.9.PreserveHostHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_preservehostheader_gatewayfilter_factory)**

**[115.10.RequestRateLimiter GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_requestratelimiter_gatewayfilter_factory)**

[115.10.1.Redis RateLimiter](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_redis_ratelimiter)

**[115.11.重定向到GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_redirectto_gatewayfilter_factory)**

**[115.12.RemoveRequestHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_removerequestheader_gatewayfilter_factory)**

**[115.13.RemoveResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_removeresponseheader_gatewayfilter_factory)**

**[115.14.RewritePath GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rewritepath_gatewayfilter_factory)**

**[115.15.RewriteLocationResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rewritelocationresponseheader_gatewayfilter_factory)**

**[115.16.RewriteResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rewriteresponseheader_gatewayfilter_factory)**

**[115.17.SaveSession GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_savesession_gatewayfilter_factory)**

**[115.18.SecureHeaders GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_secureheaders_gatewayfilter_factory)**

**[115.19.SetPath GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setpath_gatewayfilter_factory)**

**[115.20.SetRequestHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setrequestheader_gatewayfilter_factory)**

**[115.21.SetResponseHeader GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setresponseheader_gatewayfilter_factory)**

**[115.22.SetStatus GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setstatus_gatewayfilter_factory)**

**[115.23.StripPrefix GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stripprefix_gatewayfilter_factory)**

**[115.24.重试GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_gatewayfilter_factory)**

**[115.25.RequestSize GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_requestsize_gatewayfilter_factory)**

**[115.26.修改请求正文GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_modify_request_body_gatewayfilter_factory)**

**[115.27.修改响应主体GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_modify_response_body_gatewayfilter_factory)**

**[115.28.默认过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_default_filters)**

**[116.全局过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_global_filters)**

**[116.1.全局过滤器和GatewayFilter的组合订购](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_combined_global_filter_and_gatewayfilter_ordering)**

**[116.2.正向路由过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_forward_routing_filter)**

**[116.3.LoadBalancerClient筛选器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_loadbalancerclient_filter)**

**[116.4.ReactiveLoadBalancerClientFilter](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "reactive-loadbalancer-client-filter)**

**[116.5.网络路由过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_netty_routing_filter)**

**[116.6.净写响应过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_netty_write_response_filter)**

**[116.7.RouteToRequestUrl过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_routetorequesturl_filter)**

**[116.8.Websocket路由过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_websocket_routing_filter)**

**[116.9.网关指标过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gateway_metrics_filter)**

**[116.10.将交换标记为已路由](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_marking_an_exchange_as_routed)**

**[117. HttpHeadersFilters](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_httpheadersfilters)**

**[117.1.转发的标题过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_forwarded_headers_filter)**

**[117.2.RemoveHopByHop标头过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_removehopbyhop_headers_filter)**

**[117.3.XForwarded标头过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_xforwarded_headers_filter)**

**[118. TLS / SSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tls_ssl)**

**[118.1.TLS握手](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tls_handshake)**

**[119.配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_2)**

**[119.1.流利的Java Routes API](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_fluent_java_routes_api)**

**[119.2.DiscoveryClient路由定义定位器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_discoveryclient_route_definition_locator)**

[119.2.1.为DiscoveryClient路由配置谓词和过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuring_predicates_and_filters_for_discoveryclient_routes)

**[120. Reactor Netty访问日志](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_reactor_netty_access_logs)**

**[121. CORS配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cors_configuration)**

**[122.执行器API](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_actuator_api)**

**[122.1.详细执行器格式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_verbose_actuator_format)**

**[122.2.检索路由过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retrieving_route_filters)**

[122.2.1.全局过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_global_filters_2)

[122.2.2.路线过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_route_filters)

**[122.3.刷新路由缓存](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_refreshing_the_route_cache)**

**[122.4.检索网关中定义的路由](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retrieving_the_routes_defined_in_the_gateway)**

**[122.5.检索有关特定路线的信息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retrieving_information_about_a_particular_route)**

**[122.6.创建和删除特定路线](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_and_deleting_a_particular_route)**

**[122.7.回顾：所有端点的列表](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_recap_list_of_all_endpoints)**

**[123.故障排除](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "troubleshooting)**

**[123.1.日志级别](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_log_levels)**

**[123.2.窃听](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_wiretap)**

**[124.开发人员指南](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_developer_guide)**

**[124.1.编写自定义路线谓词工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_writing_custom_route_predicate_factories)**

**[124.2.编写自定义GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_writing_custom_gatewayfilter_factories)**

**[124.3.编写自定义全局过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_writing_custom_global_filters)**

**[125.使用Spring MVC或Webflux构建一个简单的网关](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_building_a_simple_gateway_using_spring_mvc_or_webflux)**

**[十六.Spring Cloud功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_function_2)**

**[126.引言](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_introduction_2)**

**[127.入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_getting_started)**

**[128.构建和运行功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_building_and_running_a_function)**

**[129.功能目录和灵活的功能签名](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_function_catalog_and_flexible_function_signatures)**

**[129.1.Java 8功能支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_java_8_function_support)**

**[129.2.Kotlin Lambda支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kotlin_lambda_support)**

**[130.独立的Web应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_standalone_web_applications)**

**[131.独立流应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_standalone_streaming_applications)**

**[132.部署打包功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deploying_a_packaged_function)**

**[133.功能性Bean定义](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_functional_bean_definitions)**

**[133.1.将功能与传统的Bean定义进行比较](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_comparing_functional_with_traditional_bean_definitions)**

**[133.2.测试功能应用](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_testing_functional_applications)**

**[133.3.功能性Bean声明的局限性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_limitations_of_functional_bean_declaration)**

**[134.动态编译](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dynamic_compilation)**

**[135.无服务器平台适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_serverless_platform_adapters)**

**[135.1.AWS Lambda](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_aws_lambda)**

[135.1.1.介绍](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_introduction_3)

[135.1.2.有关JAR布局的注意事项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_notes_on_jar_layout)

[135.1.3.上载](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_upload)

[135.1.4.Platfom的特定功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_platfom_specific_features)

[HTTP和API网关](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_http_and_api_gateway)

**[135.2.Azure功能](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_azure_functions)**

[135.2.1.有关JAR布局的注意事项](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_notes_on_jar_layout_2)

[135.2.2.建立](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_build)

[135.2.3.运行示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_running_the_sample)

**[135.3.Apache Openwhisk](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_apache_openwhisk)**

[135.3.1.快速开始](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start_5)

**[十七.Spring Cloud Kubernetes](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_kubernetes)**

**[136.为什么需要Spring Cloud Kubernetes？](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_why_do_you_need_spring_cloud_kubernetes)**

**[137. Starters](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_starters)**

**[138. Kubernetes的DiscoveryClient](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_discoveryclient_for_kubernetes)**

**[139. Kubernetes本机服务发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kubernetes_native_service_discovery)**

**[140. Kubernetes PropertySource实现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kubernetes_propertysource_implementations)**

**[140.1.使用ConfigMapPropertySource](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "configmap-propertysource)**

**[140.2.秘密PropertySource](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_secrets_propertysource)**

**[140.3.PropertySource重新加载](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_propertysource_reload)**

**[141. Ribbon在Kubernetes中发现](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ribbon_discovery_in_kubernetes)**

**[142. Kubernetes生态系统意识](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kubernetes_ecosystem_awareness)**

**[142.1.Kubernetes配置文件自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kubernetes_profile_autoconfiguration)**

**[142.2.Istio意识](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_istio_awareness)**

**[143.豆荚健康指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pod_health_indicator)**

**[144. Leader选举](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_leader_election)**

**[145. Kubernetes内部的安全配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_security_configurations_inside_kubernetes)**

**[145.1.命名空间](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_namespace)**

**[145.2.服务帐号](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_account)**

**[146.服务注册中心的实施](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_service_registry_implementation)**

**[147.范例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_examples_2)**

**[148.其他资源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_other_resources)**

**[149.建筑](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_building)**

**[149.1.基本编译和测试](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_basic_compile_and_test)**

**[149.2.文献资料](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_documentation)**

**[149.3.使用代码](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_working_with_the_code)**

[149.3.1.使用m2eclipse导入eclipse](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_importing_into_eclipse_with_m2eclipse)

[149.3.2.不使用m2eclipse导入eclipse](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_importing_into_eclipse_without_m2eclipse)

**[150.贡献](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_contributing)**

**[150.1.签署贡献者许可协议](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sign_the_contributor_license_agreement)**

**[150.2.行为守则](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_code_of_conduct)**

**[150.3.规范和内务守则](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_code_conventions_and_housekeeping)**

**[150.4.Checkstyle](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_checkstyle)**

[150.4.1.Checkstyle配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_checkstyle_configuration)

**[150.5.IDE设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ide_setup)**

[150.5.1.Intellij IDEA](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_intellij_idea)

**[十八.Spring Cloud GCP](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-gcp-reference)**

**[151.引言](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_introduction_4)**

**[152.依赖性管理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dependency_management)**

**[153.入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_getting_started_2)**

**[153.1.Spring Initializr](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_initializr)**

[153.1.1.GCP支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gcp_support)

[153.1.2.GCP讯息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gcp_messaging)

[153.1.3.GCP储存](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gcp_storage)

**[153.2.代码示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_code_samples)**

**[153.3.代码挑战](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_code_challenges)**

**[153.4.入门指南](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_getting_started_guides)**

**[154. Spring Cloud GCP核心](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-gcp-core)**

**[154.1.专案编号](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_project_id)**

**[154.2.证书](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_credentials)**

[154.2.1.范围](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_scopes)

**[154.3.环境](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_environment)**

**[154.4.Spring Initializr](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_initializr_2)**

**[155.Google Cloud Pub / Sub](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_google_cloud_pubsub)**

**[155.1.发布/订阅操作和模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pubsub_operations_template)**

[155.1.1.发布到主题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_publishing_to_a_topic)

[JSON支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_json_support)

[155.1.2.订阅订阅](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_subscribing_to_a_subscription)

[155.1.3.从订阅中提取消息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pulling_messages_from_a_subscription)

**[155.2.发布/订阅管理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_pubsub_management)**

[155.2.1.创建一个话题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_a_topic)

[155.2.2.删除主题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deleting_a_topic)

[155.2.3.列出主题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_listing_topics)

[155.2.4.创建订阅](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_creating_a_subscription)

[155.2.5.删除订阅](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_deleting_a_subscription)

[155.2.6.列表订阅](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_listing_subscriptions)

**[155.3.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pubsub-configuration)**

**[155.4.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample)**

**[156. Spring资源](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_resources)**

**[156.1.谷歌云存储](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_google_cloud_storage)**

[156.1.1.设置内容类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_setting_the_content_type)

**[156.2.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_3)**

**[156.3.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_2)**

**[157. Spring JDBC](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_jdbc)**

**[157.1.先决条件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_prerequisites)**

**[157.2.Spring Boot Google Cloud SQL入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_boot_starter_for_google_cloud_sql)**

[157.2.1.DataSource创建流程](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_datasource_creation_flow)

[157.2.2.故障排除技巧](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_troubleshooting_tips)

[连接问题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "connection-issues)

[像c.g.cloud.sql.core.SslSocketFactory : Re-throwing cached exception due to attempt to refresh instance information too soon after error这样的错误](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_errors_like_c_g_cloud_sql_core_sslsocketfactory_re_throwing_cached_exception_due_to_attempt_to_refresh_instance_information_too_soon_after_error)

[PostgreSQL：java.net.SocketException: already connected问题](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_postgresql_java_net_socketexception_already_connected_issue)

**[157.3.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_samples_3)**

**[158. Spring Integration](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_integration)**

**[158.1.Cloud Pub / Sub的通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_channel_adapters_for_cloud_pubsub)**

[158.1.1.入站通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_inbound_channel_adapter)

[158.1.2.出站通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_outbound_channel_adapter)

[158.1.3.标头映射](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_header_mapping)

**[158.2.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_3)**

**[158.3.Google Cloud Storage的通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_channel_adapters_for_google_cloud_storage)**

[158.3.1.入站通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_inbound_channel_adapter_2)

[158.3.2.入站流媒体通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_inbound_streaming_channel_adapter)

[158.3.3.出站通道适配器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_outbound_channel_adapter_2)

**[158.4.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_4)**

**[159. Spring Cloud Stream](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_stream_2)**

**[159.1.总览](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_overview_2)**

**[159.2.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_4)**

[159.2.1.生产者目标配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_producer_destination_configuration)

[159.2.2.消费者目的地配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_consumer_destination_configuration)

**[159.3.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_5)**

**[160. Spring Cloud Sleuth](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_sleuth_2)**

**[160.1.追踪](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_tracing_2)**

**[160.2.Spring Boot Stackdriver Trace入门](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_boot_starter_for_stackdriver_trace)**

**[160.3.覆盖自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_overriding_the_auto_configuration)**

**[160.4.与记录集成](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_integration_with_logging)**

**[160.5.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_6)**

**[161. Stackdriver记录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stackdriver_logging)**

**[161.1.Web MVC拦截器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_web_mvc_interceptor)**

**[161.2.登录支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_logback_support)**

[161.2.1.通过API记录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_log_via_api)

[161.2.2.通过控制台登录](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_log_via_console)

**[161.3.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_7)**

**[162. Spring Cloud Config](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_cloud_config_2)**

**[162.1.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_5)**

**[162.2.快速开始](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_quick_start_6)**

**[162.3.在运行时刷新配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_refreshing_the_configuration_at_runtime)**

**[162.4.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_8)**

**[163. Spring Data Cloud Spanner](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_data_cloud_spanner)**

**[163.1.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_6)**

[163.1.1.Cloud Spanner设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_spanner_settings)

[163.1.2.Repository设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_repository_settings)

[163.1.3.自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_autoconfiguration)

**[163.2.对象映射](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_object_mapping)**

[163.2.1.建设者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_constructors)

[163.2.2.表](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_table)

[表名的SpEL表达式](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spel_expressions_for_table_names)

[163.2.3.主键](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_primary_keys)

[163.2.4.列](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_columns)

[163.2.5.嵌入式对象](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_embedded_objects)

[163.2.6.人际关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_relationships)

[163.2.7.支持的类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_supported_types)

[163.2.8.清单](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_lists)

[163.2.9.结构清单](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_lists_of_structs)

[163.2.10.自定义类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_types)

[163.2.11.结构数组列的自定义转换器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_converter_for_struct_array_columns)

**[163.3.扳手操作和模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spanner_operations_template)**

[163.3.1.SQL查询](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sql_query)

[163.3.2.读](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read)

[163.3.3.高级阅读](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_advanced_reads)

[过时的阅读](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_stale_read)

[从二级索引读取](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read_from_a_secondary_index)

[读取偏移量和限制](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read_with_offsets_and_limits)

[排序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sorting)

[部分阅读](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partial_read)

[查询与读取的选项摘要](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_summary_of_options_for_query_vs_read)

[163.3.4.写入/更新](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_write_update)

[插入](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_insert)

[更新资料](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_update)

[增补](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_upsert)

[部分更新](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partial_update)

[163.3.5.DML](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dml)

[163.3.6.交易次数](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_transactions)

[读/写事务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_readwrite_transaction)

[只读交易](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read_only_transaction)

[带有@Transactional批注的声明式事务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_declarative_transactions_with_transactional_annotation)

[163.3.7.DML语句](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_dml_statements)

**[163.4.Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_repositories)**

[163.4.1.CRUD Repository](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_crud_repository)

[163.4.2.分页和排序Repository](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_paging_and_sorting_repository)

[163.4.3.扳手Repository](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spanner_repository)

**[163.5.查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods)**

[163.5.1.按约定查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_by_convention)

[163.5.2.自定义SQL / DML查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_sqldml_query_methods)

[具有命名查询属性的查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_with_named_queries_properties)

[带注释的查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_with_annotation)

[163.5.3.投影](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_projections)

[163.5.4.REST Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rest_repositories)

**[163.6.数据库和Schema管理员](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_database_and_schema_admin)**

**[163.7.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_9)**

**[164. Spring Data Cloud Datastore](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_data_cloud_datastore)**

**[164.1.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_7)**

[164.1.1.Cloud Datastore设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_datastore_settings)

[164.1.2.Repository设置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_repository_settings_2)

[164.1.3.自动配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_autoconfiguration_2)

**[164.2.对象映射](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_object_mapping_2)**

[164.2.1.建设者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_constructors_2)

[164.2.2.类](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kind)

[164.2.3.按键](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_keys)

[164.2.4.领域](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_fields)

[164.2.5.支持的类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_supported_types_2)

[164.2.6.自定义类型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_types_2)

[164.2.7.集合和数组](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_collections_and_arrays)

[164.2.8.自定义转换器的集合](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_converter_for_collections)

**[164.3.人际关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_relationships_2)**

[164.3.1.嵌入式实体](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_embedded_entities)

[地图](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_maps)

[164.3.2.祖辈关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_ancestor_descendant_relationships)

[164.3.3.关键参考关系](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_key_reference_relationships)

**[164.4.数据存储操作和模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_datastore_operations_template)**

[164.4.1.GQL查询](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_gql_query)

[164.4.2.按编号查找](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_find_by_ids)

[指标](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_indexes)

[读取偏移量，限制和排序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read_with_offsets_limits_and_sorting)

[部分阅读](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partial_read_2)

[164.4.3.写入/更新](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_write_update_2)

[部分更新](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_partial_update_2)

[164.4.4.交易次数](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_transactions_2)

[带有@Transactional批注的声明式事务](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_declarative_transactions_with_transactional_annotation_2)

[164.4.5.对地图的读写支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_read_write_support_for_maps)

**[164.5.Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_repositories_2)**

[164.5.1.按约定查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_by_convention_2)

[164.5.2.自定义GQL查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_custom_gql_query_methods)

[带注释的查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_with_annotation_2)

[具有命名查询属性的查询方法](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_query_methods_with_named_queries_properties_2)

[164.5.3.交易次数](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_transactions_3)

[164.5.4.投影](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_projections_2)

[164.5.5.REST Repositories](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_rest_repositories_2)

**[164.6.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_10)**

**[165. Redis的Cloud Memorystore](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_memorystore_for_redis)**

**[165.1.Spring缓存](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_spring_caching)**

**[166.云身份识别代理（IAP）身份验证](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_identity_aware_proxy_iap_authentication)**

**[166.1.组态](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_configuration_8)**

**[166.2.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_11)**

**[167.Google Cloud Vision](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_google_cloud_vision)**

**[167.1.云视觉模板](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_vision_template)**

**[167.2.检测图像标签示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_detect_image_labels_example)**

**[167.3.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_12)**

**[168. Cloud Foundry](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_cloud_foundry)**

**[169. Kotlin支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_kotlin_support)**

**[169.1.先决条件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_prerequisites_2)**

**[170.示例](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_sample_13)**

**[十九.附录：配置纲要Properties](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_appendix_compendium_of_configuration_properties)**

# 

Spring Cloud为开发人员提供了用于快速构建分布式系统中某些常见模式的工具（例如，配置管理，服务发现，断路器，智能路由，微代理，控制总线）。分布式系统的协调产生了样板模式，并且使用Spring云开发人员可以快速支持实现这些模式的服务和应用程序。它们可以在任何分布式环境中正常工作，包括开发人员自己的笔记本电脑，裸机数据中心和受管理的平台，例如Cloud Foundry。

版本：Greenwich.SR5

# **1.特点**

Spring Cloud专注于为典型的用例和扩展机制提供良好的开箱即用体验，以涵盖其他情况。

* 分布式/版本化配置
* 服务注册和发现
* 路由
* 服务到服务的呼叫
* 负载均衡
* 断路器
* 分布式消息传递

# **第一部分：云原生应用程序**

[Cloud Native](https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook" \t "https://www.springcloud.cc/_top)是一种应用程序开发风格，可鼓励在持续交付和价值驱动型开发领域轻松采用最佳实践。一个相关的学科是构建[12要素应用程序](https://12factor.net/" \t "https://www.springcloud.cc/_top)，其中开发实践与交付和运营目标保持一致，例如，通过使用声明性编程，管理和监视。Spring Cloud通过多种特定方式促进了这些发展方式。起点是一组功能，分布式系统中的所有组件都需要轻松访问这些功能。

其中Spring Cloud建立在[Spring Boot](https://projects.spring.io/spring-boot" \t "https://www.springcloud.cc/_top)上，涵盖了许多这些功能。Spring Cloud作为两个库提供了更多功能：Spring Cloud上下文和Spring Cloud Commons。Spring Cloud上下文为Spring Cloud应用程序的ApplicationContext提供了实用程序和特殊服务（引导上下文，加密，刷新作用域和环境端点）。Spring Cloud Commons是在不同的Spring Cloud实现中使用的一组抽象和通用类（例如Spring Cloud Netflix和Spring Cloud Consul）。

如果由于“密钥大小非法”而导致异常，并且使用Sun的JDK，则需要安装Java密码术扩展（JCE）无限强度管辖权策略文件。有关更多信息，请参见以下链接：

* [Java 6 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html" \t "https://www.springcloud.cc/_top)
* [Java 7 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html" \t "https://www.springcloud.cc/_top)
* [Java 8 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html" \t "https://www.springcloud.cc/_top)

将文件解压缩到您使用的JRE / JDK x64 / x86版本的JDK / jre / lib / security文件夹中。

|  |
| --- |
| IMG_256 |
| Spring Cloud是根据非限制性Apache 2.0许可证发行的。如果您想为文档的这一部分做出贡献或发现错误，可以在[github上](https://github.com/spring-cloud/spring-cloud-commons/tree/master/docs/src/main/asciidoc" \t "https://www.springcloud.cc/_top)找到源代码和项目跟踪工具。 |

## **2. Spring Cloud上下文：应用程序上下文服务**

Spring Boot对于如何使用Spring来构建应用程序有自己的看法。例如，它具有用于公共配置文件的常规位置，并具有用于公共管理和监视任务的端点。Spring Cloud以此为基础，并添加了一些功能，可能系统中的所有组件都将使用或偶尔需要这些功能。

## **2.1 Bootstrap应用程序上下文**

Spring Cloud应用程序通过创建“ bootstrap ”上下文来运行，该上下文是主应用程序的父上下文。它负责从外部源加载配置属性，并负责解密本地外部配置文件中的属性。这两个上下文共享一个Environment，它是任何Spring应用程序的外部属性的来源。默认情况下，引导程序属性（不是bootstrap.properties，而是引导程序阶段加载的属性）具有较高的优先级，因此它们不能被本地配置覆盖。

引导上下文使用与主应用程序上下文不同的约定来定位外部配置。可以使用bootstrap.yml来代替application.yml（或.properties），而将引导程序和外部环境的外部配置很好地分开。以下清单显示了一个示例：

**bootstrap.yml。**

spring:

application:

name: foo

cloud:

config:

uri: ${SPRING\_CONFIG\_URI:http://localhost:8888}

如果您的应用程序需要来自服务器的任何特定于应用程序的配置，则最好设置spring.application.name（在bootstrap.yml或application.yml中）。为了将属性spring.application.name用作应用程序的上下文ID，必须在bootstrap.[properties | yml]中进行设置。

如果要检索特定的配置文件配置，还应该在bootstrap.[properties | yml]中设置spring.profiles.active。

您可以通过设置spring.cloud.bootstrap.enabled=false来完全禁用引导过程（例如，在系统属性中）。

## **2.2应用程序上下文层次结构**

如果从SpringApplication或SpringApplicationBuilder构建应用程序上下文，那么Bootstrap上下文将作为父级添加到该上下文。Spring的一个功能是子上下文从其父级继承属性源和配置文件，因此与构建没有Spring Cloud Config的相同上下文相比，“ 主 ”应用程序上下文包含其他属性源。其他属性来源是：

* “ bootstrap ”：如果在Bootstrap上下文中找到任何PropertySourceLocators并且具有非空属性，则会以高优先级显示可选的CompositePropertySource。一个示例是Spring Cloud Config服务器中的属性。有关如何自定义此属性源内容的说明，请参见“ [第2.6节“自定义Bootstrap Property源”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "customizing-bootstrap-property-sources" \o "2.6自定义Bootstrap属性源) ”。
* “ applicationConfig：[classpath：bootstrap.yml] ”（以及相关文件，如果Spring配置文件处于活动状态）：如果您拥有bootstrap.yml（或.properties），则这些属性用于配置Bootstrap上下文。然后，当它们的父级被设置时，它们被添加到子级上下文。它们的优先级低于application.yml（或.properties）以及创建Spring Boot应用程序过程中正常添加到子级的任何其他属性源的优先级。有关如何自定义这些属性源内容的说明，请参见“ [第2.3节“更改引导程序Properties”的位置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "customizing-bootstrap-properties" \o "2.3更改引导程序属性的位置) ”。

由于属性源的排序规则，“ bootstrap ”条目优先。但是，请注意，这些不包含来自bootstrap.yml的任何数据，该数据的优先级非常低，但可用于设置默认值。

您可以通过设置创建的任何ApplicationContext的父上下文来扩展上下文层次结构，例如，使用其自己的界面或使用SpringApplicationBuilder便捷方法（parent()，child()和sibling()）。引导上下文是您自己创建的最高级祖先的父级。层次结构中的每个上下文都有其自己的“ bootstrap ”（可能为空）属性源，以避免无意间将价值从父辈提升到子孙后代。如果有配置服务器，则层次结构中的每个上下文原则上也可以具有不同的spring.application.name，因此也具有不同的远程属性源。正常的Spring应用程序上下文行为规则适用于属性解析：子上下文的属性按名称以及属性源名称覆盖父级属性。（如果子项具有与父项同名的属性源，则子项中不包括来自父项的值）。

请注意，SpringApplicationBuilder可让您在整个层次结构中共享Environment，但这不是默认设置。因此，同级上下文尤其不需要具有相同的配置文件或属性源，即使它们可能与其父级共享相同的值。

## **2.3更改引导程序Properties的位置**

可以通过设置spring.cloud.bootstrap.name（默认值：bootstrap），spring.cloud.bootstrap.location（默认值：空）或spring.cloud.bootstrap.additional-location（默认值：空）来指定bootstrap.yml（或.properties）位置。 —例如，在系统属性中。这些属性的行为类似于具有相同名称的spring.config.\*变体。使用spring.cloud.bootstrap.location将替换默认位置，并且仅使用指定的位置。要将位置添加到默认位置列表中，可以使用spring.cloud.bootstrap.additional-location。实际上，它们是通过在引导程序Environment中设置这些属性来设置引导程序ApplicationContext的。如果存在有效的配置文件（通过spring.profiles.active或通过您正在构建的上下文中的Environment API），该配置文件中的属性也会被加载，这与常规Spring Boot应用程序中的加载情况相同-例如，从bootstrap-development.properties中获取development个人资料。

## **2.4覆盖远程Properties的值**

通过引导上下文添加到应用程序中的属性源通常是“ 远程的 ”（例如，来自Spring Cloud Config Server）。默认情况下，不能在本地覆盖它们。如果要让您的应用程序使用其自己的系统属性或配置文件覆盖远程属性，则远程属性源必须通过设置spring.cloud.config.allowOverride=true来授予其权限（在本地设置无效）。设置该标志后，将使用两个更细粒度的设置来控制远程属性相对于系统属性和应用程序本地配置的位置：

* spring.cloud.config.overrideNone=true：从任何本地属性源覆盖。
* spring.cloud.config.overrideSystemProperties=false：只有系统属性，命令行参数和环境变量（而不是本地配置文件）才应覆盖远程设置。

## **2.5自定义Bootstrap配置**

通过将项添加到名为org.springframework.cloud.bootstrap.BootstrapConfiguration的项下的/META-INF/spring.factories中，可以将引导上下文设置为执行您喜欢的任何操作。它包含用于创建上下文的Spring @Configuration类的逗号分隔列表。您可以在此处创建要用于主应用程序上下文进行自动装配的任何beans。@Beans类型为ApplicationContextInitializer的特殊合同。如果要控制启动顺序，则可以用@Order批注标记类（默认顺序为last）。

|  |
| --- |
| IMG_257 |
| 当添加自定义BootstrapConfiguration，小心你添加类不是@ComponentScanned错误地进入你的“ 主 ”应用程序上下文，这里可能并不需要它们。为引导配置类使用单独的程序包名称，并确保@ComponentScan或带注释的配置类@SpringBootApplication尚未包含该名称。 |

引导过程结束时，将初始化程序注入到主要的SpringApplication实例中（这是正常的Spring Boot启动顺序，无论它是作为独立应用程序运行还是部署在应用程序服务器中）。首先，从spring.factories中找到的类创建引导上下文。然后，在启动之前，将类型为ApplicationContextInitializer的所有@Beans添加到主SpringApplication。

## **2.6自定义引导程序Property源**

引导过程添加的外部配置的默认属性来源是Spring Cloud Config服务器，但是您可以通过将类型PropertySourceLocator的beans添加到引导上下文（通过spring.factories）来添加其他来源。例如，您可以从其他服务器或数据库插入其他属性。

例如，请考虑以下定制定位器：

@Configuration**public** **class** CustomPropertySourceLocator **implements** PropertySourceLocator {

@Override

**public** PropertySource<?> locate(Environment environment) {

**return** **new** MapPropertySource("customProperty",

Collections.<String, Object>singletonMap("property.from.sample.custom.source", "worked as intended"));

}

}

传入的Environment是即将创建的ApplicationContext的那个，换句话说，就是我们为其提供其他属性源的那个。它已经有其正常的Spring Boot提供的属性源，因此您可以使用这些属性来定位特定于此Environment的属性源（例如，通过在spring.application.name上键入它，这与默认设置相同）。 Spring Cloud Config服务器属性源定位符）。

如果您创建一个包含此类的jar，然后添加包含以下内容的META-INF/spring.factories，则customProperty PropertySource会出现在任何在其类路径中包含该jar的应用程序中：

org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPropertySourceLocator

## **2.7日志配置**

如果要使用Spring Boot来配置日志设置，则应将此配置放在`bootstrap。[yml | 属性]（如果您希望将其应用于所有事件）。

|  |
| --- |
| IMG_258 |
| 为了使Spring Cloud正确初始化日志记录配置，您不能使用自定义前缀。例如，初始化记录系统时，Spring Cloud无法识别使用custom.loggin.logpath。 |

## **2.8环境变化**

应用程序侦听EnvironmentChangeEvent并以几种标准方式对更改做出反应（用户可以通过常规方式将其他ApplicationListeners作为@Beans添加）。观察到EnvironmentChangeEvent时，它会列出已更改的键值，并且应用程序将这些键值用于：

* 重新绑定上下文中的任何@ConfigurationProperties beans
* 为logging.level.\*中的所有属性设置记录器级别

请注意，默认情况下，Config Client不轮询Environment中的更改。通常，我们不建议您使用这种方法来检测更改（尽管您可以使用@Scheduled注释对其进行设置）。如果您具有横向扩展的客户端应用程序，则最好向所有实例广播EnvironmentChangeEvent，而不是让它们轮询更改（例如，使用[Spring Cloud Bus](https://github.com/spring-cloud/spring-cloud-bus" \t "https://www.springcloud.cc/_top)）。

只要您可以实际更改Environment并发布事件，EnvironmentChangeEvent就涵盖了一大类刷新用例。请注意，这些API是公共的，并且是核心Spring的一部分）。您可以通过访问/configprops端点（正常的Spring Boot Actuator功能）来验证更改是否绑定到@ConfigurationProperties beans。例如，DataSource可以在运行时更改其maxPoolSize（由Spring Boot创建的默认DataSource是@ConfigurationProperties bean）并动态地增加容量。重新绑定@ConfigurationProperties并不涵盖另一类用例，在这种情况下，您需要对刷新有更多的控制，并且需要对整个ApplicationContext进行原子更改。为了解决这些问题，我们有@RefreshScope。

## **2.9刷新范围**

进行配置更改时，标记为@RefreshScope的Spring @Bean将得到特殊处理。此功能解决了状态beans的问题，该状态仅在初始化时才注入配置。例如，如果通过Environment更改数据库URL时DataSource具有打开的连接，则您可能希望这些连接的持有者能够完成他们正在做的事情。然后，下次某物从池中借用一个连接时，它将获得一个具有新URL的连接。

有时，甚至可能必须将@RefreshScope批注应用到只能初始化一次的某些beans上。如果bean是“不可变的”，则必须用@RefreshScope注释bean或在属性键spring.cloud.refresh.extra-refreshable下指定类名。

|  |  |
| --- | --- |
| IMG_259 | **重要** |
| 如果您自己创建一个DataSource bean，而实现是一个HikariDataSource，则返回最特定的类型，在这种情况下为HikariDataSource。否则，您将需要设置spring.cloud.refresh.extra-refreshable=javax.sql.DataSource。 |

刷新作用域beans是惰性代理，它们在使用时（即在调用方法时）进行初始化，并且作用域充当初始化值的缓存。若要强制bean在下一个方法调用上重新初始化，必须使它的缓存条目无效。

RefreshScope在上下文中是bean，并具有公用的refreshAll()方法，可通过清除目标缓存来刷新作用域中的所有beans。/refresh端点公开了此功能（通过HTTP或JMX）。要按名称刷新单个bean，还有一个refresh(String)方法。

要公开/refresh端点，您需要在应用程序中添加以下配置：

management: endpoints: web: exposure: include: refresh

|  |
| --- |
| IMG_260 |
| @RefreshScope在@Configuration类上（在技术上）有效，但是可能会导致令人惊讶的行为。例如，这并不意味着该类中定义的所有@Beans本身都在@RefreshScope中。具体而言，除非刷新本身在@RefreshScope中，否则依赖那些beans的任何内容都不能依赖于刷新启动时对其进行更新。在这种情况下，将在刷新时重建它，并重新注入其依赖项。此时，它们将从刷新的@Configuration重新初始化。 |

## **2.10加密和解密**

Spring Cloud具有Environment预处理器，用于在本地解密属性值。它遵循与Config Server相同的规则，并且通过encrypt.\*具有相同的外部配置。因此，您可以使用{cipher}\*形式的加密值，并且只要存在有效密钥，就可以在主应用程序上下文获得Environment设置之前对它们进行解密。要在应用程序中使用加密功能，您需要在类路径中包含Spring Security RSA（Maven坐标：“ org.springframework.security:spring-security-rsa”），并且还需要JVM中的全功能JCE扩展。

如果由于“密钥大小非法”而导致异常，并且使用Sun的JDK，则需要安装Java密码术扩展（JCE）无限强度管辖权策略文件。有关更多信息，请参见以下链接：

* [Java 6 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html" \t "https://www.springcloud.cc/_top)
* [Java 7 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html" \t "https://www.springcloud.cc/_top)
* [Java 8 JCE](https://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html" \t "https://www.springcloud.cc/_top)

将文件解压缩到您使用的JRE / JDK x64 / x86版本的JDK / jre / lib / security文件夹中。

## **2.11端点**

对于Spring Boot Actuator应用程序，可以使用一些其他管理端点。您可以使用：

* 从POST到/actuator/env以更新Environment并重新绑定@ConfigurationProperties和日志级别。
* /actuator/refresh重新加载引导上下文并刷新@RefreshScope beans。
* /actuator/restart关闭ApplicationContext并重新启动（默认情况下禁用）。
* /actuator/pause和/actuator/resume用于调用Lifecycle方法（ApplicationContext中的stop()和start()）。

|  |
| --- |
| IMG_261 |
| 如果禁用/actuator/restart端点，则/actuator/pause和/actuator/resume端点也将被禁用，因为它们只是/actuator/restart的特例。 |

## **3. Spring Cloud Commons：通用抽象**

服务发现，负载平衡和断路器之类的模式将它们带到一个通用的抽象层，可以由所有Spring Cloud客户端使用，而与实现无关（例如，使用Eureka或Consul进行的发现） ）。

## **3.1 @EnableDiscoveryClient**

Spring Cloud Commons提供了@EnableDiscoveryClient批注。这将寻找META-INF/spring.factories与DiscoveryClient接口的实现。Discovery Client的实现在org.springframework.cloud.client.discovery.EnableDiscoveryClient键下将配置类添加到spring.factories。DiscoveryClient实现的示例包括[Spring Cloud Netflix Eureka](https://cloud.spring.io/spring-cloud-netflix/" \t "https://www.springcloud.cc/_top)，[Spring Cloud Consul发现](https://cloud.spring.io/spring-cloud-consul/" \t "https://www.springcloud.cc/_top)和[Spring Cloud Zookeeper发现](https://cloud.spring.io/spring-cloud-zookeeper/" \t "https://www.springcloud.cc/_top)。

默认情况下，DiscoveryClient的实现会自动将本地Spring Boot服务器注册到远程发现服务器。可以通过在@EnableDiscoveryClient中设置autoRegister=false来禁用此行为。

|  |
| --- |
| IMG_262 |
| 不再需要@EnableDiscoveryClient。您可以在类路径上放置DiscoveryClient实现，以使Spring Boot应用程序向服务发现服务器注册。 |

### **3.1.1健康指标**

公用创建了Spring Boot HealthIndicator，DiscoveryClient实现可以通过实现DiscoveryHealthIndicator来参与。要禁用复合HealthIndicator，请设置spring.cloud.discovery.client.composite-indicator.enabled=false。基于DiscoveryClient的通用HealthIndicator是自动配置的（DiscoveryClientHealthIndicator）。要禁用它，请设置spring.cloud.discovery.client.health-indicator.enabled=false。要禁用DiscoveryClientHealthIndicator的描述字段，请设置spring.cloud.discovery.client.health-indicator.include-description=false。否则，它可能会像已卷起的HealthIndicator中的description一样冒泡。

### **3.1.2订购DiscoveryClient实例**

DiscoveryClient接口扩展了Ordered。当使用多个发现客户端时，这很有用，因为它允许您定义返回的发现客户端的顺序，类似于如何订购由Spring应用程序加载的beans。默认情况下，任何DiscoveryClient的顺序都设置为0。如果要为自定义DiscoveryClient实现设置不同的顺序，则只需覆盖getOrder()方法，以便它返回适合您的设置的值。除此之外，您可以使用属性来设置Spring Cloud提供的DiscoveryClient实现的顺序，其中包括ConsulDiscoveryClient，EurekaDiscoveryClient和ZookeeperDiscoveryClient。为此，您只需要将spring.cloud.{clientIdentifier}.discovery.order（对于Eureka，则为eureka.client.order）属性设置为所需的值。

## **3.2服务注册**

Commons现在提供一个ServiceRegistry接口，该接口提供诸如register(Registration)和deregister(Registration)之类的方法，这些方法使您可以提供自定义的注册服务。Registration是标记界面。

以下示例显示了正在使用的ServiceRegistry：

@Configuration@EnableDiscoveryClient(autoRegister=false)**public** **class** MyConfiguration {

**private** ServiceRegistry registry;

**public** MyConfiguration(ServiceRegistry registry) {

**this**.registry = registry;

}

*// called through some external process, such as an event or a custom actuator endpoint*

**public** **void** register() {

Registration registration = constructRegistration();

**this**.registry.register(registration);

}

}

每个ServiceRegistry实现都有自己的Registry实现。

* ZookeeperRegistration与ZookeeperServiceRegistry一起使用
* EurekaRegistration与EurekaServiceRegistry一起使用
* ConsulRegistration与ConsulServiceRegistry一起使用

如果您使用的是ServiceRegistry接口，则将需要为使用的ServiceRegistry实现传递正确的Registry实现。

### **3.2.1 ServiceRegistry自动注册**

默认情况下，ServiceRegistry实现会自动注册正在运行的服务。要禁用该行为，可以设置：\* @EnableDiscoveryClient(autoRegister=false)以永久禁用自动注册。\* spring.cloud.service-registry.auto-registration.enabled=false通过配置禁用行为。

#### **ServiceRegistry自动注册Events**

服务自动注册时将触发两个事件。注册服务之前会触发名为InstancePreRegisteredEvent的第一个事件。注册服务后，将触发名为InstanceRegisteredEvent的第二个事件。您可以注册ApplicationListener，以收听和响应这些事件。

|  |
| --- |
| IMG_263 |
| 如果将spring.cloud.service-registry.auto-registration.enabled设置为false，则不会触发这些事件。 |

### **3.2.2服务注册表执行器端点**

Spring Cloud Commons提供了一个/service-registry执行器端点。该端点依赖于Spring应用程序上下文中的Registration bean。使用GET调用/service-registry会返回Registration的状态。对具有JSON正文的同一终结点使用POST会将当前Registration的状态更改为新值。JSON正文必须包含带有首选值的status字段。请参阅更新状态时用于允许值的ServiceRegistry实现的文档以及为状态返回的值。例如，Eureka的受支持状态为UP，DOWN，OUT\_OF\_SERVICE和UNKNOWN。

## **3.3 Spring RestTemplate作为负载均衡器客户端**

RestTemplate可以自动配置为在后台使用负载均衡器客户端。要创建负载均衡的RestTemplate，请创建RestTemplate @Bean并使用@LoadBalanced限定符，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@LoadBalanced

@Bean

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {

@Autowired

**private** RestTemplate restTemplate;

**public** String doOtherStuff() {

String results = restTemplate.getForObject("http://stores/stores", String.**class**);

**return** results;

}

}

|  |  |
| --- | --- |
| IMG_264 | **警告** |
| RestTemplate bean不再通过自动配置创建。各个应用程序必须创建它。 |

URI需要使用虚拟主机名（即服务名，而不是主机名）。Ribbon客户端用于创建完整的物理地址。有关如何设置RestTemplate的详细信息，请参见[RibbonAutoConfiguration](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-ribbon/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonAutoConfiguration.java" \t "https://www.springcloud.cc/_top)。

|  |  |
| --- | --- |
| IMG_265 | **重要** |
| 为了使用负载均衡的RestTemplate，您需要在类路径中具有负载均衡器实现。推荐的实现是BlockingLoadBalancerClient-添加org.springframework.cloud:spring-cloud-loadbalancer以便使用它。RibbonLoadBalancerClient也可以使用，但是目前正在维护中，我们不建议将其添加到新项目中。 |

|  |
| --- |
| IMG_266 |
| 如果要使用BlockingLoadBalancerClient，请确保项目类路径中没有RibbonLoadBalancerClient，因为向后兼容的原因，默认情况下将使用它。 |

## **3.4 Spring WebClient作为负载均衡器客户端**

WebClient可以自动配置为使用负载均衡器客户端。要创建负载均衡的WebClient，请创建WebClient.Builder @Bean并使用@LoadBalanced限定符，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@Bean

@LoadBalanced

**public** WebClient.Builder loadBalancedWebClientBuilder() {

**return** WebClient.builder();

}

}

**public** **class** MyClass {

@Autowired

**private** WebClient.Builder webClientBuilder;

**public** Mono<String> doOtherStuff() {

**return** webClientBuilder.build().get().uri("http://stores/stores")

.retrieve().bodyToMono(String.**class**);

}

}

URI需要使用虚拟主机名（即服务名，而不是主机名）。Ribbon客户端用于创建完整的物理地址。

|  |  |
| --- | --- |
| IMG_267 | **重要** |
| 如果要使用@LoadBalanced WebClient.Builder，则需要在类路径中有一个loadbalancer实现。建议您将org.springframework.cloud:spring-cloud-loadbalancer依赖项添加到项目中。然后，将在下面使用ReactiveLoadBalancer。或者，此功能也可以在spring-cloud-starter-netflix-ribbon上使用，但是该请求将由后台的非响应LoadBalancerClient处理。此外，spring-cloud-starter-netflix-ribbon已经处于维护模式，因此我们不建议您将其添加到新项目中。 |

|  |
| --- |
| IMG_268 |
| 在下面使用的ReactorLoadBalancer支持缓存。如果检测到cacheManager，将使用ServiceInstanceSupplier的缓存版本。如果没有，我们将从发现服务中检索实例，而不进行缓存。如果您使用ReactiveLoadBalancer，建议您在项目中[启用缓存](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html" \t "https://www.springcloud.cc/_top)。 |

### **3.4.1重试失败的请求**

可以配置负载均衡的RestTemplate以重试失败的请求。默认情况下，禁用此逻辑。您可以通过在应用程序的类路径中添加[Spring重试](https://github.com/spring-projects/spring-retry" \t "https://www.springcloud.cc/_top)来启用它。负载平衡的RestTemplate遵循与重试失败的请求有关的某些Ribbon配置值。您可以使用client.ribbon.MaxAutoRetries，client.ribbon.MaxAutoRetriesNextServer和client.ribbon.OkToRetryOnAllOperations属性。如果要通过对类路径使用Spring重试来禁用重试逻辑，则可以设置spring.cloud.loadbalancer.retry.enabled=false。有关这些属性的作用的说明，请参见[Ribbon文档](https://github.com/Netflix/ribbon/wiki/Getting-Started" \l "the-properties-file-sample-clientproperties" \t "https://www.springcloud.cc/_top)。

如果要在重试中实现BackOffPolicy，则需要创建LoadBalancedRetryFactory类型的bean并覆盖createBackOffPolicy方法：

@Configuration**public** **class** MyConfiguration {

@Bean

LoadBalancedRetryFactory retryFactory() {

**return** **new** LoadBalancedRetryFactory() {

@Override

**public** BackOffPolicy createBackOffPolicy(String service) {

**return** **new** ExponentialBackOffPolicy();

}

};

}

}

|  |
| --- |
| IMG_269 |
| 前面示例中的client应替换为您的Ribbon客户名称。 |

如果要向重试功能中添加一个或多个RetryListener实现，则需要创建类型为LoadBalancedRetryListenerFactory的bean，并返回要用于给定服务的RetryListener数组，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@Bean

LoadBalancedRetryListenerFactory retryListenerFactory() {

**return** **new** LoadBalancedRetryListenerFactory() {

@Override

**public** RetryListener[] createRetryListeners(String service) {

**return** **new** RetryListener[]{**new** RetryListener() {

@Override

**public** <T, E **extends** Throwable> **boolean** open(RetryContext context, RetryCallback<T, E> callback) {

*//TODO Do you business...*

**return** true;

}

@Override

**public** <T, E **extends** Throwable> **void** close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

@Override

**public** <T, E **extends** Throwable> **void** onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

}};

}

};

}

}

## **3.5多个RestTemplate对象**

如果您想要一个RestTemplate而不是负载均衡的，请创建一个RestTemplate bean并注入它。要访问负载均衡的RestTemplate，请在创建@Bean时使用@LoadBalanced限定符，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@LoadBalanced

@Bean

RestTemplate loadBalanced() {

**return** **new** RestTemplate();

}

@Primary

@Bean

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {@Autowired**private** RestTemplate restTemplate;

@Autowired

@LoadBalanced

**private** RestTemplate loadBalanced;

**public** String doOtherStuff() {

**return** loadBalanced.getForObject("http://stores/stores", String.**class**);

}

**public** String doStuff() {

**return** restTemplate.getForObject("https://example.com", String.**class**);

}

}

|  |  |
| --- | --- |
| IMG_270 | **重要** |
| 注意，在前面的示例中，在普通的RestTemplate声明上使用了@Primary批注，以消除不合格的@Autowired注入的歧义。 |

|  |
| --- |
| IMG_271 |
| 如果看到诸如java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89之类的错误，请尝试注入RestOperations或设置spring.aop.proxyTargetClass=true。 |

## **3.6多个WebClient对象**

如果要使WebClient负载不均衡，请创建一个WebClient bean并注入它。要访问负载均衡的WebClient，请在创建@Bean时使用@LoadBalanced限定符，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@LoadBalanced

@Bean

WebClient.Builder loadBalanced() {

**return** WebClient.builder();

}

@Primary

@Bean

WebClient.Builder webClient() {

**return** WebClient.builder();

}

}

**public** **class** MyClass {

@Autowired

**private** WebClient.Builder webClientBuilder;

@Autowired

@LoadBalanced

**private** WebClient.Builder loadBalanced;

**public** Mono<String> doOtherStuff() {

**return** loadBalanced.build().get().uri("http://stores/stores")

.retrieve().bodyToMono(String.**class**);

}

**public** Mono<String> doStuff() {

**return** webClientBuilder.build().get().uri("http://example.com")

.retrieve().bodyToMono(String.**class**);

}

}

## **3.7 Spring WebFlux WebClient作为负载均衡器客户端**

### **3.7.1 Spring具有响应式负载均衡器的WebFlux WebClient**

可以将WebClient配置为使用ReactiveLoadBalancer。如果将org.springframework.cloud:spring-cloud-loadbalancer添加到项目中，并且spring-webflux在类路径中，则会自动配置ReactorLoadBalancerExchangeFilterFunction。以下示例说明如何配置WebClient以在后台使用无功负载均衡器：

**public** **class** MyClass {

@Autowired

**private** ReactorLoadBalancerExchangeFilterFunction lbFunction;

**public** Mono<String> doOtherStuff() {

**return** WebClient.builder().baseUrl("http://stores")

.filter(lbFunction)

.build()

.get()

.uri("/stores")

.retrieve()

.bodyToMono(String.**class**);

}

}

URI需要使用虚拟主机名（即服务名，而不是主机名）。ReactorLoadBalancerClient用于创建完整的物理地址。

### **3.7.2 Spring WebFlux WebClient，带有非反应式负载均衡器客户端**

如果您的项目中没有org.springframework.cloud:spring-cloud-loadbalancer，但是确实有spring-cloud-starter-netflix-ribbon，则仍可以将WebClient与LoadBalancerClient结合使用。如果spring-webflux在类路径中，将自动配置LoadBalancerExchangeFilterFunction。但是请注意，这是在后台使用非反应性客户端。以下示例显示如何配置WebClient以使用负载均衡器：

**public** **class** MyClass {

@Autowired

**private** LoadBalancerExchangeFilterFunction lbFunction;

**public** Mono<String> doOtherStuff() {

**return** WebClient.builder().baseUrl("http://stores")

.filter(lbFunction)

.build()

.get()

.uri("/stores")

.retrieve()

.bodyToMono(String.**class**);

}

}

URI需要使用虚拟主机名（即服务名，而不是主机名）。LoadBalancerClient用于创建完整的物理地址。

警告：现在不建议使用此方法。我们建议您将[WebFlux与电抗性负载平衡器一起](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "webflux-with-reactive-loadbalancer" \o "3.7.1具有响应式负载均衡器的Spring WebFlux WebClient) 使用。

### **3.7.3传递自己的Load-Balancer客户端配置**

您还可以使用@LoadBalancerClient批注传递您自己的负载平衡器客户端配置，并传递负载平衡器客户端的名称和配置类，如下所示：

@Configuration@LoadBalancerClient(value = "stores", configuration = StoresLoadBalancerClientConfiguration.class)**public** **class** MyConfiguration {

@Bean

@LoadBalanced

**public** WebClient.Builder loadBalancedWebClientBuilder() {

**return** WebClient.builder();

}

}

也可以通过@LoadBalancerClients注释将多个配置（对于一个以上的负载均衡器客户端）一起传递，如下所示：

@Configuration@LoadBalancerClients({@LoadBalancerClient(value = "stores", configuration = StoresLoadBalancerClientConfiguration.class), @LoadBalancerClient(value = "customers", configuration = CustomersLoadBalancerClientConfiguration.class)})**public** **class** MyConfiguration {

@Bean

@LoadBalanced

**public** WebClient.Builder loadBalancedWebClientBuilder() {

**return** WebClient.builder();

}

}

## **3.8忽略网络接口**

有时，忽略某些命名的网络接口很有用，以便可以将它们从服务发现注册中排除（例如，在Docker容器中运行时）。可以设置正则表达式列表，以使所需的网络接口被忽略。以下配置将忽略docker0接口以及所有以veth开头的接口：

**application.yml。**

spring:

cloud:

inetutils:

ignoredInterfaces:

- docker0

- veth.\*

您还可以通过使用正则表达式列表来强制仅使用指定的网络地址，如以下示例所示：

**bootstrap.yml。**

spring:

cloud:

inetutils:

preferredNetworks:

- 192.168

- 10.0

您也可以只使用站点本地地址，如以下示例所示：.application.yml

spring:

cloud:

inetutils:

useOnlySiteLocalInterfaces: true

有关构成站点本地地址的详细信息，请参见[Inet4Address.html.isSiteLocalAddress（）](https://docs.oracle.com/javase/8/docs/api/java/net/Inet4Address.html" \l "isSiteLocalAddress--" \t "https://www.springcloud.cc/_top)。

## **3.9 HTTP客户端工厂**

Spring Cloud Commons提供了beans用于创建Apache HTTP客户端（ApacheHttpClientFactory）和OK HTTP客户端（OkHttpClientFactory）。仅当OK HTTP jar位于类路径上时，才创建OkHttpClientFactory bean。此外，Spring Cloud Commons提供了beans用于创建两个客户端使用的连接管理器：ApacheHttpClientConnectionManagerFactory用于Apache HTTP客户端，OkHttpClientConnectionPoolFactory用于OK HTTP客户端。如果您想自定义在下游项目中如何创建HTTP客户端，则可以提供自己的beans实现。另外，如果您提供类型为HttpClientBuilder或OkHttpClient.Builder的bean，则默认工厂将使用这些构建器作为返回到下游项目的构建器的基础。您还可以通过将spring.cloud.httpclientfactories.apache.enabled或spring.cloud.httpclientfactories.ok.enabled设置为false来禁用这些beans的创建。

## **3.10启用的功能**

Spring Cloud Commons提供了一个/features执行器端点。该端点返回类路径上可用的功能以及是否已启用它们。返回的信息包括功能类型，名称，版本和供应商。

### **3.10.1特征类型**

“功能”有两种类型：抽象和命名。

抽象功能是定义接口或抽象类并创建实现的功能，例如DiscoveryClient，LoadBalancerClient或LockService。抽象类或接口用于在上下文中找到该类型的bean。显示的版本为bean.getClass().getPackage().getImplementationVersion()。

命名功能是没有实现的特定类的功能，例如“ Circuit Breaker”，“ API Gateway”，“ Spring Cloud Bus”等。这些功能需要一个名称和一个bean类型。

### **3.10.2声明功能**

任何模块都可以声明任意数量的HasFeature beans，如以下示例所示：

@Bean**public** HasFeatures commonsFeatures() {

**return** HasFeatures.abstractFeatures(DiscoveryClient.**class**, LoadBalancerClient.**class**);

}

@Bean**public** HasFeatures consulFeatures() {

**return** HasFeatures.namedFeatures(

**new** NamedFeature("Spring Cloud Bus", ConsulBusAutoConfiguration.**class**),

**new** NamedFeature("Circuit Breaker", HystrixCommandAspect.**class**));

}

@Bean

HasFeatures localFeatures() {

**return** HasFeatures.builder()

.abstractFeature(Foo.**class**)

.namedFeature(**new** NamedFeature("Bar Feature", Bar.**class**))

.abstractFeature(Baz.**class**)

.build();

}

这些beans中的每一个都应放入受到适当保护的@Configuration中。

## **3.11 Spring Cloud兼容性验证**

由于某些用户在设置Spring Cloud应用程序时遇到问题，我们决定添加兼容性验证机制。如果您当前的设置与Spring Cloud要求不兼容，则会断开，并附上一份报告，说明出了什么问题。

目前，我们验证哪个版本的Spring Boot已添加到您的类路径中。

报告范例

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

APPLICATION FAILED TO START

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

Your project setup is incompatible with our requirements due to following reasons:

- Spring Boot [2.1.0.RELEASE] is not compatible with this Spring Cloud release train

Action:

Consider applying the following actions:

- Change Spring Boot version to one of the following versions [1.2.x, 1.3.x] .

You can find the latest Spring Boot versions here [https://spring.io/projects/spring-boot#learn].

If you want to learn more about the Spring Cloud Release train compatibility, you can visit this page [https://spring.io/projects/spring-cloud#overview] and check the [Release Trains] section.

为了禁用此功能，请将spring.cloud.compatibility-verifier.enabled设置为false。如果要覆盖兼容的Spring Boot版本，只需用兼容的Spring Boot版本的逗号分隔列表设置spring.cloud.compatibility-verifier.compatible-boot-versions属性。

# **第二部分 Spring Cloud Config**

**Greenwich SR5**

Spring Cloud Config为分布式系统中的外部化配置提供服务器端和客户端支持。使用Config Server，您可以在中心位置管理所有环境中应用程序的外部属性。客户端和服务器上的概念都与Spring Environment和PropertySource抽象映射相同，因此它们非常适合Spring应用程序，但可以与以任何语言运行的任何应用程序一起使用。在应用程序从开发人员到测试人员再到生产人员的整个部署过程中，您可以管理这些环境之间的配置，并确保应用程序具有它们迁移时所需的一切。服务器存储后端的默认实现使用git，因此它轻松支持带标签的配置环境版本，并且可以通过各种工具来访问这些内容来管理内容。添加替代实现并将其插入Spring配置很容易。

## **4.快速入门**

此快速入门介绍了如何使用Spring Cloud Config服务器的服务器和客户端。

首先，启动服务器，如下所示：

$ cd spring-cloud-config-server

$ ../mvnw spring-boot:run

该服务器是Spring Boot应用程序，因此，如果愿意，可以从IDE运行它（主类是ConfigServerApplication）。

接下来尝试一个客户端，如下所示：

$ curl localhost:8888/foo/development

{"name":"foo","label":"master","propertySources":[

{"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}},

{"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}

]}

定位属性源的默认策略是克隆git存储库（位于spring.cloud.config.server.git.uri），并使用它来初始化小型SpringApplication。小型应用程序的Environment用于枚举属性源并将其发布在JSON端点上。

HTTP服务具有以下形式的资源：

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

application在SpringApplication中作为spring.config.name注入（在常规Spring Boot应用中通常是application），profile是有效配置文件（或逗号分隔）属性列表），而label是可选的git标签（默认为master。）

Spring Cloud Config服务器从各种来源获取远程客户端的配置。以下示例从git存储库（必须提供）中获取配置，如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo

其他来源包括任何与JDBC兼容的数据库，Subversion，Hashicorp Vault，Credhub和本地文件系统。

## **4.1客户端使用**

要在应用程序中使用这些功能，您可以将其构建为依赖于spring-cloud-config-client的Spring Boot应用程序（例如，请参阅config-client或示例应用程序的测试用例）。添加依赖项最方便的方法是使用Spring Boot启动器org.springframework.cloud:spring-cloud-starter-config。还有一个Maven用户的父pom和BOM（spring-cloud-starter-parent），以及一个Gradle和Spring CLI用户的Spring IO版本管理属性文件。以下示例显示了典型的Maven配置：

**pom.xml。**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>{spring-boot-docs-version}</version>

<relativePath /> *<!-- lookup parent from repository -->*

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>{spring-cloud-version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency></dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins></build>

*<!-- repositories also needed for snapshots and milestones -->*

现在，您可以创建一个标准的Spring Boot应用程序，例如以下HTTP服务器：

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello World!";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

当此HTTP服务器运行时，它将从端口8888上的默认本地配置服务器（如果正在运行）中获取外部配置。要修改启动行为，可以使用bootstrap.properties（类似于application.properties，但用于应用程序上下文的引导阶段），如以下示例所示：

spring.cloud.config.uri: http://myconfigserver.com

默认情况下，如果未设置应用程序名称，将使用application。要修改名称，可以将以下属性添加到bootstrap.properties文件中：

spring.application.name: myapp

|  |
| --- |
| IMG_272 |
| 设置属性${spring.application.name}时，请勿在您的应用名称前加上保留字application-，以防止解析正确的属性源时出现问题。 |

引导程序属性在/env端点中显示为高优先级属性源，如以下示例所示。

$ curl localhost:8080/env

{

"profiles":[],

"configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},

"servletContextInitParams":{},

"systemProperties":{...},

...

}

名为``configService:<URL of remote repository>/<file name>的属性源包含值为bar的foo属性，并且优先级最高。

|  |
| --- |
| IMG_273 |
| 属性源名称中的URL是git存储库，而不是配置服务器URL。 |

## **5. Spring Cloud Config服务器**

Spring Cloud Config服务器为外部配置（名称/值对或等效的YAML内容）提供了一个基于HTTP资源的API。通过使用@EnableConfigServer批注，服务器可嵌入到Spring Boot应用程序中。因此，以下应用程序是配置服务器：

**ConfigServer.java。**

@SpringBootApplication@EnableConfigServer**public** **class** ConfigServer {

**public** **static** **void** main(String[] args) {

SpringApplication.run(ConfigServer.**class**, args);

}

}

像所有Spring Boot应用程序一样，它默认在端口8080上运行，但是您可以通过各种方式将其切换到更传统的端口8888。最简单的方法也是设置默认配置存储库，方法是使用spring.config.name=configserver（在Config Server jar中有configserver.yml）启动它。另一种方法是使用您自己的application.properties，如以下示例所示：

**application.properties。**

server.port: 8888spring.cloud.config.server.git.uri: file://${user.home}/config-repo

其中${user.home}/config-repo是包含YAML和属性文件的git存储库。

|  |
| --- |
| IMG_274 |
| 在Windows上，如果文件URL带有驱动器前缀（例如，[file:///${user.home}/config-repo](https://www.springcloud.cc/${user.home}/config-repo" \t "https://www.springcloud.cc/_top)）是绝对的，则在文件URL中需要一个额外的“ /”。 |

|  |
| --- |
| IMG_275 |
| 以下清单显示了在前面的示例中创建git存储库的方法：  $ cd $HOME  $ mkdir config-repo  $ cd config-repo  $ git init .  $ echo info.foo: bar > application.properties  $ git add -A .  $ git commit -m "Add application.properties" |

|  |
| --- |
| IMG_276 |
| 将本地文件系统用于git存储库仅用于测试。您应该使用服务器在生产环境中托管配置存储库。 |

|  |
| --- |
| IMG_277 |
| 如果仅将文本文件保留在其中，则配置存储库的初始克隆可以快速有效。如果存储二进制文件（尤其是大文件），则可能会在首次配置请求时遇到延迟，或者在服务器中遇到内存不足错误。 |

## **5.1环境Repository**

您应该在哪里存储配置服务器的配置数据？控制此行为的策略是服务Environment对象的EnvironmentRepository。Environment是Spring Environment的域的浅表副本（包括propertySources作为主要特征）。Environment资源由三个变量参数化：

* {application}，它映射到客户端的spring.application.name。
* {profile}，它映射到客户端上的spring.profiles.active（以逗号分隔的列表）。
* {label}，这是服务器端功能，标记了一组“版本化”的配置文件。

Repository实现通常类似于Spring Boot应用程序，从等于{application}参数的spring.config.name和等于{profiles}参数的spring.profiles.active加载配置文件。配置文件的优先规则也与常规Spring Boot应用程序中的规则相同：活动配置文件的优先级高于默认设置，并且，如果有多个配置文件，则最后一个优先（与向Map添加条目类似）。

以下示例客户端应用程序具有此引导程序配置：

**bootstrap.yml。**

spring: application: name: foo profiles: active: dev,mysql

（与Spring Boot应用程序一样，这些属性也可以由环境变量或命令行参数设置）。

如果存储库基于文件，则服务器从application.yml（在所有客户端之间共享）和foo.yml（以foo.yml优先）创建一个Environment。如果YAML文件中包含指向Spring配置文件的文档，则将以更高的优先级应用这些文件（按列出的配置文件的顺序）。如果存在特定于配置文件的YAML（或属性）文件，这些文件也将以比默认文件更高的优先级应用。较高的优先级会转换为Environment中较早列出的PropertySource。（这些相同的规则适用于独立的Spring Boot应用程序。）

您可以将spring.cloud.config.server.accept-empty设置为false，以便在未找到应用程序的情况下Server返回HTTP 404状态。默认情况下，此标志设置为true。

### **5.1.1 Git后端**

EnvironmentRepository的默认实现使用Git后端，这对于管理升级和物理环境以及审核更改非常方便。要更改存储库的位置，可以在Config Server中设置spring.cloud.config.server.git.uri配置属性（例如，在application.yml中）。如果您使用file:前缀进行设置，则它应在本地存储库中运行，以便无需服务器即可快速轻松地开始使用。但是，在那种情况下，服务器直接在本地存储库上运行而无需克隆它（如果它不是裸露的，这并不重要，因为Config Server从不对“远程”存储库进行更改）。要扩展Config Server并使其高度可用，您需要使服务器的所有实例都指向同一存储库，因此仅共享文件系统可以工作。即使在那种情况下，最好对共享文件系统存储库使用ssh:协议，以便服务器可以克隆它并将本地工作副本用作缓存。

此存储库实现将HTTP资源的{label}参数映射到git标签（提交ID，分支名称或标记）。如果git分支或标记名称包含斜杠（/），则应使用特殊字符串(\_)在HTTP URL中指定标签（以避免与其他URL路径产生歧义）。例如，如果标签为foo/bar，则替换斜杠将产生以下标签：foo(\_)bar。特殊字符串(\_)的包含内容也可以应用于{application}参数。如果您使用命令行客户端（例如curl），请注意URL中的括号-您应使用单引号（''）将其从外壳中移出。

#### **跳过SSL证书验证**

可以通过将git.skipSslValidation属性设置为true（默认值为false）来禁用配置服务器对Git服务器的SSL证书的验证。

spring: cloud: config: server: git: uri: https://example.com/my/repo skipSslValidation: **true**

#### **设置HTTP连接超时**

您可以配置配置服务器将等待获取HTTP连接的时间（以秒为单位）。使用git.timeout属性。

spring: cloud: config: server: git: uri: https://example.com/my/repo timeout: 4

#### **Git URI中的占位符**

Spring Cloud Config服务器支持带有{application}和{profile}（如果需要的话还有{label}）占位符的git存储库URL，但是请记住该标签始终用作git标签。因此，您可以使用类似于以下的结构来支持“ 每个应用程序一个存储库 ”策略：

spring: cloud: config: server: git: uri: https://github.com/myorg/{application**}**

您也可以使用类似的模式（{profile}）来支持“ 每个配置文件一个存储库 ”策略。

此外，在{application}参数中使用特殊字符串“（\_）”可以启用对多个组织的支持，如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/{application**}**

其中在请求时以以下格式提供{application}：organization(\_)application。

#### **模式匹配和多个Repositories**

Spring Cloud Config还通过在应用程序和概要文件名称上进行模式匹配来支持更复杂的需求。模式格式是以逗号分隔的{application}/{profile}名称列表，带有通配符（请注意，以通配符开头的模式可能需要加引号），如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo repos: simple: https://github.com/simple/config-repo special: pattern: special\*/dev\*,\*special\*/dev\* uri: https://github.com/special/config-repo local: pattern: local\* uri: file:/home/configsvc/config-repo

如果{application}/{profile}与任何模式都不匹配，它将使用在spring.cloud.config.server.git.uri下定义的默认URI。在上面的示例中，对于“ 简单 ”存储库，模式为simple/\*（在所有配置文件中仅匹配一个名为simple的应用程序）。在“ 本地 ”库匹配，在所有配置文件（该/\*后缀会自动添加到没有档案资料匹配的任何模式）local开头的所有应用程序名称。

|  |
| --- |
| IMG_278 |
| 的“ 单行 ”中所使用的短切“ 简单 ”的例子可以只用于如果唯一的属性被设置为URI。如果您需要设置其他任何内容（凭证，模式等），则需要使用完整表格。 |

回购中的pattern属性实际上是一个数组，因此您可以使用YAML数组（或属性文件中的[0]，[1]等后缀）绑定到多个模式。如果要运行具有多个配置文件的应用程序，则可能需要这样做，如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo repos: development: pattern:

- '\*/development'

- '\*/staging' uri: https://github.com/development/config-repo staging: pattern:

- '\*/qa'

- '\*/production' uri: https://github.com/staging/config-repo

|  |
| --- |
| IMG_279 |
| Spring Cloud猜测一个包含不以\*结尾的配置文件的模式表示您实际上要匹配以该模式开头的配置文件列表（因此\*/staging是["\*/staging", "\*/staging,\*"]的快捷方式，等等）。例如，这很普遍，例如，您需要在本地的“ 开发 ”配置文件中运行应用程序，而又需要在远程的“ 云 ”配置文件中运行应用程序。 |

每个存储库还可以选择将配置文件存储在子目录中，用于搜索这些目录的模式可以指定为searchPaths。以下示例在顶层显示了一个配置文件：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo searchPaths: foo,bar\*

在前面的示例中，服务器在顶层和foo/子目录中以及名称以bar开头的任何子目录中搜索配置文件。

默认情况下，首次请求配置时，服务器会克隆远程存储库。可以将服务器配置为在启动时克隆存储库，如以下顶级示例所示：

spring: cloud: config: server: git: uri: https://git/common/config-repo.git repos: team-a: pattern: team-a-\* cloneOnStart: **true** uri: https://git/team-a/config-repo.git team-b: pattern: team-b-\* cloneOnStart: **false** uri: https://git/team-b/config-repo.git team-c: pattern: team-c-\* uri: https://git/team-a/config-repo.git

在前面的示例中，服务器在接受任何请求之前会在启动时克隆team-a的config-repo。在请求从存储库进行配置之前，不会克隆所有其他存储库。

|  |
| --- |
| IMG_280 |
| 设置要在Config Server启动时克隆的存储库有助于在Config Server启动时快速识别配置错误的配置源（例如无效的存储库URI）。在未为配置源启用cloneOnStart的情况下，Config Server可能会以配置错误或无效的配置源成功启动，并且直到应用程序从该配置源请求配置时才检测到错误。 |

#### **认证方式**

要在远程存储库上使用HTTP基本认证，请分别添加username和password属性（不在URL中），如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo username: trolley password: strongpassword

如果您不使用HTTPS和用户凭据，则在将密钥存储在默认目录（~/.ssh）中且URI指向SSH位置（例如git@github.com:configuration/cloud-configuration）时，SSH也应立即可用。重要的是，~/.ssh/known\_hosts文件中应包含Git服务器的条目，并且其格式应为ssh-rsa。不支持其他格式（例如ecdsa-sha2-nistp256）。为避免意外，您应确保Git服务器的known\_hosts文件中仅存在一个条目，并且该条目与您提供给配置服务器的URL匹配。如果在URL中使用主机名，则要在known\_hosts文件中完全使用该主机名（而不是IP）。使用JGit访问存储库，因此您找到的任何文档都应该适用。HTTPS代理设置可以在~/.git/config中设置，也可以（使用与其他JVM进程相同的方式）使用系统属性（-Dhttps.proxyHost和-Dhttps.proxyPort）进行设置。

|  |
| --- |
| IMG_281 |
| 如果您不知道~/.git目录在哪里，请使用git config --global来操纵设置（例如，git config --global http.sslVerify false）。 |

#### **使用AWS CodeCommit进行身份验证**

Spring Cloud Config服务器还支持[AWS CodeCommit](https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html" \t "https://www.springcloud.cc/_top)身份验证。从命令行使用Git时，AWS CodeCommit使用身份验证帮助程序。该帮助程序未与JGit库一起使用，因此，如果Git URI与AWS CodeCommit模式匹配，则会为AWS CodeCommit创建一个JGit CredentialProvider。AWS CodeCommit URI遵循以下模式：//git-codecommit.${AWS\_REGION}.amazonaws.com/${repopath}。

如果您提供带有AWS CodeCommit URI的用户名和密码，则它们必须是提供对存储库访问权限的[AWS accessKeyId和secretAccessKey](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSGettingStartedGuide/AWSCredentials.html" \t "https://www.springcloud.cc/_top)。如果您未指定用户名和密码，则使用[AWS Default Credential Provider链](https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html" \t "https://www.springcloud.cc/_top)检索accessKeyId和secretAccessKey 。

如果您的Git URI与CodeCommit URI模式（如前所示）匹配，则必须在用户名和密码或默认凭据提供商链支持的位置之一中提供有效的AWS凭据。AWS EC2实例可以将[IAM角色用于EC2实例](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_282 |
| aws-java-sdk-core jar是可选的依赖项。如果aws-java-sdk-core jar不在您的类路径中，则无论git服务器URI如何，都不会创建AWS Code Commit凭证提供程序。 |

#### **使用属性进行Git SSH配置**

默认情况下，当使用SSH URI连接到Git存储库时，Spring Cloud Config服务器使用的JGit库使用SSH配置文件，例如~/.ssh/known\_hosts和/etc/ssh/ssh\_config。在Cloud Foundry之类的云环境中，本地文件系统可能是临时的，或者不容易访问。在这种情况下，可以使用Java属性设置SSH配置。为了激活基于属性的SSH配置，必须将spring.cloud.config.server.git.ignoreLocalSshSettings属性设置为true，如以下示例所示：

spring: cloud: config: server: git: uri: git@gitserver.com:team/repo1.git ignoreLocalSshSettings: **true** hostKey: someHostKey hostKeyAlgorithm: ssh-rsa privateKey: |

-----BEGIN RSA PRIVATE KEY-----

MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX

IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF

ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud

1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i

oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W

DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd

fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b

BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG

EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj

5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8

+AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe

pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG

ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ

xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW

dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi

PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX

VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z

FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R

gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4

VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV

cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee

KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N

CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs

q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J

69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT

-----END RSA PRIVATE KEY-----

下表描述了SSH配置属性。

**表5.1。SSH配置Properties**

| **Property名称** | **备注** |
| --- | --- |
| **ignoreLocalSshSettings** | 如果为true，请使用基于属性的SSH配置而不是基于文件的SSH配置。必须设置为spring.cloud.config.server.git.ignoreLocalSshSettings，**而不是**在存储库定义中。 |
| **私钥** | 有效的SSH私钥。如果ignoreLocalSshSettings为true并且Git URI为SSH格式，则必须设置。 |
| **hostKey** | 有效的SSH主机密钥。如果还设置了hostKeyAlgorithm，则必须设置。 |
| **hostKeyAlgorithm** | ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521之一。如果还设置了hostKey，则必须设置。 |
| **strictHostKeyChecking** | true或false。如果为false，请忽略主机密钥错误。 |
| **knownHostsFile** | 自定义.known\_hosts文件的位置。 |
| **preferredAuthentications** | 覆盖服务器身份验证方法顺序。如果服务器在publickey方法之前进行了键盘交互式身份验证，则应该可以避免登录提示。 |

#### **Git搜索路径中的占位符**

Spring Cloud Config服务器还支持带有{application}和{profile}（如果需要的话还有{label}）占位符的搜索路径，如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo searchPaths: '{application}'

上面的清单导致在存储库中搜索与目录（以及顶层）同名的文件。通配符在带有占位符的搜索路径中也有效（搜索中包括任何匹配的目录）。

#### **强制拉动Git Repositories**

如前所述，Spring Cloud Config服务器会复制远程git存储库，以防本地副本变脏（例如，操作系统进程更改了文件夹内容），使得Spring Cloud Config服务器无法从远程更新本地副本。资料库。

要解决此问题，有一个force-pull属性，如果本地副本脏了，则可以使Spring Cloud Config服务器从远程存储库强制拉出，如以下示例所示：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo force-pull: **true**

如果您有多个存储库配置，则可以为每个存储库配置force-pull属性，如以下示例所示：

spring: cloud: config: server: git: uri: https://git/common/config-repo.git force-pull: **true** repos: team-a: pattern: team-a-\* uri: https://git/team-a/config-repo.git force-pull: **true** team-b: pattern: team-b-\* uri: https://git/team-b/config-repo.git force-pull: **true** team-c: pattern: team-c-\* uri: https://git/team-a/config-repo.git

|  |
| --- |
| IMG_283 |
| force-pull属性的默认值为false。 |

#### **删除Git Repositories中未跟踪的分支**

由于Spring Cloud Config服务器在检出分支到本地存储库（例如，通过标签获取属性）后具有远程git存储库的克隆，因此它将永久保留该分支，直到下一个服务器重新启动（这将创建新的本地存储库）。因此，可能会删除远程分支，但仍可获取其本地副本。而且，如果Spring Cloud Config服务器客户端服务以--spring.cloud.config.label=deletedRemoteBranch,master开头，它将从deletedRemoteBranch本地分支获取属性，而不是从master获取属性。

为了使本地存储库分支保持整洁并保持远程状态-可以设置deleteUntrackedBranches属性。这将使Spring Cloud Config服务器从本地存储库中**强制**删除未跟踪的分支。例：

spring: cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo deleteUntrackedBranches: **true**

|  |
| --- |
| IMG_284 |
| deleteUntrackedBranches属性的默认值为false。 |

#### **Git刷新率**

您可以使用spring.cloud.config.server.git.refreshRate控制配置服务器多久从Git后端获取更新的配置数据。以秒为单位指定此属性的值。默认情况下，该值为0，这意味着配置服务器将在每次请求时从Git存储库中获取更新的配置。

### **5.1.2版本控制后端文件系统使用**

|  |
| --- |
| IMG_285 |
| 使用基于VCS的后端（git，svn），文件被检出或克隆到本地文件系统。默认情况下，它们以config-repo-为前缀放在系统临时目录中。例如，在Linux上，它可能是/tmp/config-repo-<randomid>。一些操作系统[通常会清除](https://serverfault.com/questions/377348/when-does-tmp-get-cleared/377349" \l "377349" \t "https://www.springcloud.cc/_top)临时目录。这可能导致意外行为，例如缺少属性。为避免此问题，请通过将spring.cloud.config.server.git.basedir或spring.cloud.config.server.svn.basedir设置为不在系统临时结构中的目录来更改Config Server使用的目录。 |

### **5.1.3文件系统后端**

Config Server中还有一个“ 本机 ”配置文件，该配置文件不使用Git，而是从本地类路径或文件系统（您要使用spring.cloud.config.server.native.searchLocations指向的任何静态URL）加载配置文件。要使用本机配置文件，请使用spring.profiles.active=native启动Config Server。

|  |
| --- |
| IMG_286 |
| 请记住对文件资源使用file:前缀（默认情况下，没有前缀的通常是类路径）。与任何Spring Boot配置一样，您可以嵌入${}样式的环境占位符，但是请记住，Windows中的绝对路径需要额外的/（例如，[file:///${user.home}/config-repo](https://www.springcloud.cc/${user.home}/config-repo" \t "https://www.springcloud.cc/_top)）。 |

|  |
| --- |
| IMG_287 |
| searchLocations的默认值与本地Spring Boot应用程序（即[classpath:/, classpath:/config, file:./, file:./config]）相同。这不会将application.properties从服务器公开给所有客户端，因为服务器中存在的所有属性源在被发送到客户端之前都已被删除。 |

|  |
| --- |
| IMG_288 |
| 文件系统后端非常适合快速入门和测试。要在生产环境中使用它，您需要确保文件系统可靠并且可以在Config Server的所有实例之间共享。 |

搜索位置可以包含{application}，{profile}和{label}的占位符。这样，您可以隔离路径中的目录并选择一种对您有意义的策略（例如，每个应用程序的子目录或每个配置文件的子目录）。

如果在搜索位置中不使用占位符，则此存储库还将HTTP资源的{label}参数附加到搜索路径上的后缀，因此将从每个搜索位置**和**与该名称相同的子目录加载属性文件。标签（在Spring环境中，带有标签的属性优先）。因此，没有占位符的默认行为与添加以/{label}/结尾的搜索位置相同。例如，file:/tmp/config与file:/tmp/config,file:/tmp/config/{label}相同。可以通过设置spring.cloud.config.server.native.addLabelLocations=false来禁用此行为。

### **5.1.4 Vault后端**

Spring Cloud Config服务器还支持[Vault](https://www.vaultproject.io/" \t "https://www.springcloud.cc/_top)作为后端。

Vault是用于安全访问机密的工具。秘密是您要严格控制访问权限的所有内容，例如API密钥，密码，证书和其他敏感信息。Vault为所有机密提供了统一的界面，同时提供了严格的访问控制并记录了详细的审核日志。

有关Vault的更多信息，请参见[Vault快速入门指南](https://learn.hashicorp.com/vault/?track=getting-started" \l "getting-started" \t "https://www.springcloud.cc/_top)。

要使配置服务器能够使用Vault后端，您可以使用vault配置文件运行配置服务器。例如，在配置服务器的application.properties中，您可以添加spring.profiles.active=vault。

默认情况下，配置服务器假定您的Vault服务器在[http://127.0.0.1:8200](http://127.0.0.1:8200/" \t "https://www.springcloud.cc/_top)下运行。它还假定后端的名称为secret，密钥为application。所有这些默认值都可以在配置服务器的application.properties中进行配置。下表描述了可配置的Vault属性：

| **名称** | **默认值** |
| --- | --- |
| host | 127.0.0.1 |
| port | 8200 |
| scheme | http |
| backend | secret |
| defaultKey | application |
| profileSeparator | , |
| kvVersion | 1 |
| skipSslValidation | false |
| timeout | 5 |
| namespace | null |

|  |  |
| --- | --- |
| IMG_289 | **重要** |
| 上表中的所有属性必须以spring.cloud.config.server.vault为前缀，或放置在复合配置的正确Vault部分中。 |

所有可配置的属性都可以在org.springframework.cloud.config.server.environment.VaultEnvironmentProperties中找到。

Vault 0.10.0引入了版本化的键值后端（k / v后端版本2），该后端公开了与早期版本不同的API，现在它需要在安装路径和实际上下文路径之间使用data/并包装data对象中的秘密。设置kvVersion=2将考虑到这一点。

（可选）支持Vault企业版X-Vault-Namespace标头。要将其发送到Vault，请设置namespace属性。

在配置服务器运行时，您可以向服务器发出HTTP请求以从Vault后端检索值。为此，您需要Vault服务器的令牌。

首先，将一些数据放入您的Vault中，如以下示例所示：

$ vault kv put secret/application foo=bar baz=bam

$ vault kv put secret/myapp foo=myappsbar

其次，向配置服务器发出HTTP请求以检索值，如以下示例所示：

$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"

您应该看到类似于以下内容的响应：

**{**

"name":"myapp"**,**

"profiles":**[**

"default"

]**,**

"label":null**,**

"version":null**,**

"state":null**,**

"propertySources":**[**

**{**

"name":"vault:myapp"**,**

"source":**{**

"foo":"myappsbar"

**}**

**},**

**{**

"name":"vault:application"**,**

"source":**{**

"baz":"bam"**,**

"foo":"bar"

**}**

**}**

**]}**

#### **多个Properties来源**

使用Vault时，可以为您的应用程序提供多个属性源。例如，假设您已将数据写入Vault中的以下路径：

secret/myApp,dev

secret/myApp

secret/application,dev

secret/application

写入secret/application的Properties对[使用Config Server的所有应用程序](https://www.springcloud.cc/spring-cloud-greenwich.html)均可[用](https://www.springcloud.cc/spring-cloud-greenwich.html)。名称为myApp的应用程序将具有写入secret/myApp和secret/application的所有属性。当myApp启用了dev配置文件时，写入上述所有路径的属性将可用，列表中第一个路径中的属性优先于其他属性。

### **5.1.5通过代理访问后端**

配置服务器可以通过HTTP或HTTPS代理访问Git或Vault后端。通过proxy.http和proxy.https下的设置，可以为Git或Vault控制此行为。这些设置是针对每个存储库的，因此，如果您使用[组合环境存储库](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "composite-environment-repositories" \o "5.1.9复合环境存储库)，则必须分别为组合中的每个后端配置代理设置。如果使用的网络需要HTTP和HTTPS URL分别使用代理服务器，则可以为单个后端配置HTTP和HTTPS代理设置。

下表描述了HTTP和HTTPS代理的代理配置属性。所有这些属性都必须以proxy.http或proxy.https作为前缀。

**表5.2。代理配置Properties**

| **Property名称** | **备注** |
| --- | --- |
| **主办** | 代理的主机。 |
| **港口** | 用于访问代理的端口。 |
| **nonProxyHosts** | 配置服务器应在代理外部访问的所有主机。如果同时为proxy.http.nonProxyHosts和proxy.https.nonProxyHosts提供了值，则将使用proxy.http值。 |
| **用户名** | 用来验证代理的用户名。如果同时为proxy.http.username和proxy.https.username提供了值，则将使用proxy.http值。 |
| **密码** | 用来验证代理的密码。如果同时为proxy.http.password和proxy.https.password提供了值，则将使用proxy.http值。 |

以下配置使用HTTPS代理访问Git存储库。

spring: profiles: active: git cloud: config: server: git: uri: https://github.com/spring-cloud-samples/config-repo proxy: https: host: my-proxy.host.io password: myproxypassword port: '3128' username: myproxyusername nonProxyHosts: example.com

### **5.1.6与所有应用程序共享配置**

所有应用程序之间的共享配置根据您采用的方法而异，如以下主题所述：

* [称为“基于文件的Repositories”部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-config-server-file-based-repositories" \o "基于文件的存储库)
* [称为“ Vault服务器”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-config-server-vault-server" \o "保管箱服务器)

#### **基于文件的Repositories**

使用基于文件（git，svn和本机）的存储库，所有客户端应用程序之间共享文件名称为application\*（application.properties，application.yml，application-\*.properties等）的资源。您可以使用具有这些文件名的资源来配置全局默认值，并在必要时使它们被应用程序特定的文件覆盖。

#\_property\_overrides [属性覆盖]功能也可以用于设置全局默认值，允许使用占位符应用程序在本地覆盖它们。

|  |
| --- |
| IMG_290 |
| 使用“ 本机 ”配置文件（本地文件系统后端），您应该使用不属于服务器自身配置的显式搜索位置。否则，默认搜索位置中的application\*资源将被删除，因为它们是服务器的一部分。 |

#### **Vault服务器**

将Vault用作后端时，可以通过将配置放在secret/application中来与所有应用程序共享配置。例如，如果您运行以下Vault命令，则所有使用配置服务器的应用程序都将具有可用的属性foo和baz：

$ vault write secret/application foo=bar baz=bam

#### **CredHub服务器**

将CredHub用作后端时，可以通过将配置放在/application/中或将其放在应用程序的default配置文件中来与所有应用程序共享配置。例如，如果您运行以下CredHub命令，则使用配置服务器的所有应用程序将具有对它们可用的属性shared.color1和shared.color2：

credhub set --name "/application/profile/master/shared" --type=json

value: {"shared.color1": "blue", "shared.color": "red"}

credhub set --name "/my-app/default/master/more-shared" --type=json

value: {"shared.word1": "hello", "shared.word2": "world"}

### **5.1.7 JDBC后端**

Spring Cloud Config服务器支持JDBC（关系数据库）作为配置属性的后端。您可以通过向类路径中添加spring-jdbc并使用jdbc配置文件或添加类型为JdbcEnvironmentRepository的bean来启用此功能。如果您在类路径上包括正确的依赖项（有关更多详细信息，请参见用户指南），Spring Boot将配置数据源。

数据库需要有一个名为PROPERTIES的表，该表具有名为APPLICATION，PROFILE和LABEL的列（通常具有Environment的含义），以及KEY和VALUE，用于Properties样式的键和值对。Java中所有字段的类型均为String，因此您可以根据需要将它们设置为VARCHAR。Property值的行为与来自名为{application}-{profile}.properties的Spring Boot属性文件的值的行为相同，包括所有加密和解密，它们将用作后处理步骤（也就是说，在存储库中直接执行）。

### **5.1.8 CredHub后端**

Spring Cloud Config服务器支持[CredHub](https://docs.cloudfoundry.org/credhub" \t "https://www.springcloud.cc/_top)作为配置属性的后端。您可以通过向[Spring CredHub](https://spring.io/projects/spring-credhub" \t "https://www.springcloud.cc/_top)添加依赖项来启用此功能。

**pom.xml。**

<dependencies>

<dependency>

<groupId>org.springframework.credhub</groupId>

<artifactId>spring-credhub-starter</artifactId>

</dependency></dependencies>

以下配置使用双向TLS访问CredHub：

spring: profiles: active: credhub cloud: config: server: credhub: url: https://credhub:8844

属性应存储为JSON，例如：

credhub set --name "/demo-app/default/master/toggles" --type=json

value: {"toggle.button": "blue", "toggle.link": "red"}

credhub set --name "/demo-app/default/master/abs" --type=json

value: {"marketing.enabled": true, "external.enabled": false}

名称为spring.cloud.config.name=demo-app的所有客户端应用程序将具有以下属性：

{

toggle.button: "blue",

toggle.link: "red",

marketing.enabled: true,

external.enabled: false

}

|  |
| --- |
| IMG_291 |
| 如果未指定配置文件，将使用default，而未指定标签时，将使用master作为默认值。注意：添加到application的值将由所有应用程序共享。 |

#### **OAuth 2.0**

您可以使用[UAA](https://docs.cloudfoundry.org/concepts/architecture/uaa.html" \t "https://www.springcloud.cc/_top)作为提供程序通过[OAuth 2.0](https://oauth.net/2/" \t "https://www.springcloud.cc/_top)进行身份验证。

**pom.xml。**

<dependencies>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-client</artifactId>

</dependency></dependencies>

以下配置使用OAuth 2.0和UAA访问CredHub：

spring: profiles: active: credhub cloud: config: server: credhub: url: https://credhub:8844 oauth2: registration-id: credhub-client security: oauth2: client: registration: credhub-client: provider: uaa client-id: credhub\_config\_server client-secret: asecret authorization-grant-type: client\_credentials provider: uaa: token-uri: https://uaa:8443/oauth/token

|  |
| --- |
| IMG_292 |
| 使用的UAA客户ID的范围应为credhub.read。 |

### **5.1.9复合环境Repositories**

在某些情况下，您可能希望从多个环境存储库中提取配置数据。为此，您可以在配置服务器的应用程序属性或YAML文件中启用composite配置文件。例如，如果要从Subversion存储库以及两个Git存储库中提取配置数据，则可以为配置服务器设置以下属性：

spring: profiles: active: composite cloud: config: server: composite:

- type: svn uri: file:///path/to/svn/repo

- type: git uri: file:///path/to/rex/git/repo

- type: git uri: file:///path/to/walter/git/repo

使用此配置，优先级由composite键下的存储库列出顺序确定。在上面的示例中，首先列出了Subversion存储库，因此在Subversion存储库中找到的值将覆盖在一个Git存储库中为同一属性找到的值。在rex Git存储库中找到的值将在walter Git存储库中为相同属性找到的值之前使用。

如果只想从每种不同类型的存储库中提取配置数据，则可以在配置服务器的应用程序属性或YAML文件中启用相应的配置文件，而不启用composite配置文件。例如，如果要从单个Git存储库和单个HashiCorp Vault服务器中提取配置数据，则可以为配置服务器设置以下属性：

spring: profiles: active: git**,** vault cloud: config: server: git: uri: file:///path/to/git/repo order: 2 vault: host: 127.0.0.1 port: 8200 order: 1

使用此配置，可以通过order属性确定优先级。您可以使用order属性为所有存储库指定优先级顺序。order属性的数值越低，优先级越高。存储库的优先级顺序有助于解决包含相同属性值的存储库之间的任何潜在冲突。

|  |
| --- |
| IMG_293 |
| 如果您的复合环境包括上一个示例中的Vault服务器，则在对配置服务器的每个请求中都必须包含Vault令牌。请参阅[Vault后端](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault-backend" \o "5.1.4保管库后端)。 |

|  |
| --- |
| IMG_294 |
| 从环境存储库中检索值时，任何类型的故障都会导致整个组合环境的故障。 |

|  |
| --- |
| IMG_295 |
| 使用复合环境时，所有存储库都包含相同的标签很重要。如果您的环境与前面的示例中的环境类似，并且您请求带有master标签的配置数据，但是Subversion存储库不包含名为master的分支，则整个请求将失败。 |

#### **定制复合环境Repositories**

除了使用Spring Cloud中的一个环境存储库之外，您还可以提供自己的EnvironmentRepository bean作为复合环境的一部分。为此，您的bean必须实现EnvironmentRepository接口。如果要在复合环境中控制自定义EnvironmentRepository的优先级，则还应该实现Ordered接口并覆盖getOrdered方法。如果未实现Ordered接口，则EnvironmentRepository的优先级最低。

### **5.1.10 Property覆盖**

Config Server具有“ 替代 ”功能，使操作员可以为所有应用程序提供配置属性。应用程序使用常规的Spring Boot钩子不会意外更改重写的属性。要声明覆盖，请将名称/值对的映射添加到spring.cloud.config.server.overrides，如以下示例所示：

spring: cloud: config: server: overrides: foo: bar

前面的示例使作为配置客户端的所有应用程序读取foo=bar，而与它们自己的配置无关。

|  |
| --- |
| IMG_296 |
| 配置系统不能强制应用程序以任何特定方式使用配置数据。因此，覆盖无法执行。但是，它们确实为Spring Cloud Config客户端提供了有用的默认行为。 |

|  |
| --- |
| IMG_297 |
| 通常，可以使用反斜杠（\）来转义$或{，以逃避（并在客户端上解析）具有${}的Spring环境占位符。例如，除非应用程序提供自己的app.foo，否则\${app.foo:bar}解析为bar。 |

|  |
| --- |
| IMG_298 |
| 在YAML中，您不需要转义反斜杠本身。但是，在属性文件中，在服务器上配置替代时，确实需要转义反斜杠。 |

您可以通过在远程存储库中设置spring.cloud.config.overrideNone=true标志（默认为false），使客户端中所有替代的优先级更像默认值，让应用程序在环境变量或系统属性中提供自己的值。

## **5.2健康指标**

Config Server带有运行状况指示器，用于检查配置的EnvironmentRepository是否正常工作。默认情况下，它会向EnvironmentRepository询问名为app的应用程序，default配置文件以及EnvironmentRepository实现提供的默认标签。

您可以配置运行状况指示器以检查更多应用程序以及自定义配置文件和自定义标签，如以下示例所示：

spring: cloud: config: server: health: repositories: myservice: label: mylabel myservice-dev: name: myservice profiles: development

您可以通过设置spring.cloud.config.server.health.enabled=false禁用运行状况指示器。

## **5.3安全性**

您可以用对您有意义的任何方式来保护Config Server（从物理网络安全到OAuth2承载令牌），因为Spring Security和Spring Boot为许多安全措施提供了支持。

要使用默认的Spring Boot配置的HTTP基本安全性，请在类路径上包含Spring Security（例如，通过spring-boot-starter-security）。缺省值为user用户名和随机生成的密码。随机密码在实践中没有用，因此我们建议您配置密码（通过设置spring.security.user.password）并对其进行加密（有关如何操作的说明，请参见下文）。

## **5.4加密和解密**

|  |  |
| --- | --- |
| IMG_299 | **重要** |
| 要使用加密和解密功能，您需要在JVM中安装完整功能的JCE（默认情况下不包括）。您可以从Oracle 下载“ Java密码学扩展（JCE）无限强度辖区策略文件 ”并按照安装说明进行操作（本质上，您需要用下载的JRE lib / security目录替换这两个策略文件）。 |

如果远程属性源包含加密的内容（值以{cipher}开头），则在通过HTTP发送给客户端之前，将对它们进行解密。此设置的主要优点是，当属性值处于“ 静止 ”状态时（例如，在git存储库中），不需要使用纯文本格式。如果无法解密某个值，则将其从属性源中删除，并使用相同的密钥添加一个附加属性，但附加前缀为invalid和一个表示“ 不适用 ”的值（通常为<n/a>）。这很大程度上是为了防止将密文用作密码并意外泄漏。

如果为配置客户端应用程序设置了远程配置存储库，则它可能包含与以下内容类似的application.yml：

**application.yml。**

spring: datasource: username: dbuser password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'

.properties文件中的加密值不能用引号引起来。否则，该值不会解密。以下示例显示了有效的值：

**application.properties。**

spring.datasource.username: dbuser

spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ

您可以安全地将此纯文本推送到共享的git存储库，并且秘密密码仍然受到保护。

服务器还公开/encrypt和/decrypt端点（假设这些端点是安全的，并且只能由授权代理访问）。如果您编辑远程配置文件，则可以使用Config Server通过POST到/encrypt端点来加密值，如以下示例所示：

$ curl localhost:8888/encrypt -d mysecret

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

|  |
| --- |
| IMG_300 |
| 如果您加密的值中包含需要URL编码的字符，则应对curl使用--data-urlencode选项以确保它们正确编码。 |

|  |
| --- |
| IMG_301 |
| 确保不要在加密值中包含任何curl命令统计信息。将值输出到文件可以帮助避免此问题。 |

也可以通过/decrypt使用反向操作（前提是服务器配置了对称密钥或完整密钥对），如以下示例所示：

$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

|  |
| --- |
| IMG_302 |
| 如果使用curl进行测试，请使用--data-urlencode（而不是-d）或设置显式的Content-Type: text/plain以确保在有特殊字符时，curl可以正确编码数据（“ +”特别棘手）。 |

在将加密的值放入YAML或属性文件之前，以及将其提交并将其推送到远程（可能不安全）存储之前，请获取加密的值并添加{cipher}前缀。

/encrypt和/decrypt端点也都接受/\*/{application}/{profiles}形式的路径，当客户端调用主应用程序时，可用于按应用程序（名称）和配置文件控制密码。环境资源。

|  |
| --- |
| IMG_303 |
| 要以这种精细的方式控制密码，您还必须提供类型为TextEncryptorLocator的@Bean，以按名称和配置文件创建不同的加密器。默认情况下不提供（所有加密使用相同的密钥）。 |

spring命令行客户端（安装了Spring Cloud CLI扩展名）也可以用于加密和解密，如以下示例所示：

$ spring encrypt mysecret --key foo

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

要使用文件中的密钥（例如用于加密的RSA公钥），请在密钥值前添加“ @”并提供文件路径，如以下示例所示：

$ spring encrypt mysecret --key @${HOME}/.ssh/id\_rsa.pub

AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...

|  |
| --- |
| IMG_304 |
| --key参数是强制性的（尽管前缀为--）。 |

## **5.5密钥管理**

Config Server可以使用对称（共享）密钥或非对称密钥（RSA密钥对）。非对称选择在安全性方面优越，但使用对称密钥通常更方便，因为它是在bootstrap.properties中配置的单个属性值。

要配置对称密钥，您需要将encrypt.key设置为秘密字符串（或使用ENCRYPT\_KEY环境变量将其保留在纯文本配置文件之外）。

|  |
| --- |
| IMG_305 |
| 您无法使用encrypt.key配置非对称密钥。 |

要配置非对称密钥，请使用密钥库（例如，由JDK附带的keytool实用程序创建的密钥库）。密钥库属性为encrypt.keyStore.\*，其中\*等于

| **Property** | **描述** |
| --- | --- |
| encrypt.keyStore.location | Contains a Resource location |
| encrypt.keyStore.password | Holds the password that unlocks the keystore |
| encrypt.keyStore.alias | Identifies which key in the store to use |
| encrypt.keyStore.type | The type of KeyStore to create. Defaults to jks. |

加密是使用公钥完成的，解密需要私钥。因此，原则上，如果只想加密（并准备用私钥在本地解密值），则只能在服务器中配置公钥。实际上，您可能不希望在本地进行解密，因为它会将密钥管理过程分布在所有客户端上，而不是将其集中在服务器上。另一方面，如果您的配置服务器相对不安全并且只有少数客户端需要加密的属性，那么它可能是一个有用的选项。

## **5.6创建密钥库进行测试**

要创建用于测试的密钥库，可以使用类似于以下内容的命令：

$ keytool -genkeypair -alias mytestkey -keyalg RSA \

-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \

-keypass changeme -keystore server.jks -storepass letmein

|  |
| --- |
| IMG_306 |
| 使用JDK 11或更高版本时，使用上述命令时可能会收到以下警告。在这种情况下，您可能需要确保keypass和storepass值匹配。 |

Warning: Different store and key passwords not supported for PKCS12 KeyStores. Ignoring user-specified -keypass value.

将server.jks文件放入类路径中（例如），然后在您的bootstrap.yml中，为Config Server创建以下设置：

encrypt: keyStore: location: classpath:/server.jks password: letmein alias: mytestkey secret: changeme

## **5.7使用多个按键和按键旋转**

除了加密属性值中的{cipher}前缀外，Config Server在（Base64编码的）密文开始之前查找零个或多个{name:value}前缀。密钥被传递到TextEncryptorLocator，后者可以执行为密码找到TextEncryptor所需的任何逻辑。如果已配置密钥库（encrypt.keystore.location），则默认定位器将查找具有key前缀提供的别名的密钥，其密文类似于以下内容：

foo: bar: `{cipher}{key:testkey}...`

定位器查找名为“ testkey”的键。也可以通过在前缀中使用{secret:…​}值来提供机密。但是，如果未提供，则默认为使用密钥库密码（这是在构建密钥库且未指定密钥时得到的密码）。如果确实提供了机密，则还应该使用自定义SecretLocator对机密进行加密。

当密钥仅用于加密几个字节的配置数据时（也就是说，它们未在其他地方使用），从密码的角度讲，几乎不需要旋转密钥。但是，您有时可能需要更改密钥（例如，在发生安全漏洞时）。在这种情况下，所有客户端都需要更改其源配置文件（例如，在git中），并在所有密码中使用新的{key:…​}前缀。请注意，客户端需要首先检查Config Server密钥库中的密钥别名是否可用。

|  |
| --- |
| IMG_307 |
| 如果要让Config Server处理所有加密以及解密，则还可以将{name:value}前缀添加为发布到/encrypt端点的纯文本。 |

## **5.8服务加密的Properties**

有时，您希望客户端在本地解密配置，而不是在服务器中进行解密。在这种情况下，如果提供encrypt.\*配置来查找密钥，则仍然可以拥有/encrypt和/decrypt端点，但是需要通过放置{10 /来显式关闭对传出属性的解密。 bootstrap.[yml|properties]中的}。如果您不关心端点，则在不配置键或启用标志的情况下都可以使用。

## **6.提供其他格式**

来自环境端点的默认JSON格式非常适合Spring应用程序使用，因为它直接映射到Environment抽象。如果愿意，可以通过在资源路径中添加后缀（“ .yml”，“。yaml”或“ .properties”）来使用与YAML或Java属性相同的数据。对于不关心JSON终结点的结构或它们提供的额外元数据的应用程序来说，这可能很有用（例如，不使用Spring的应用程序可能会受益于此方法的简单性）。

YAML和属性表示形式还有一个附加标志（提供为名为resolvePlaceholders的布尔查询参数），用于指示应在输出中解析源文档中的占位符（以标准Spring ${…​}格式）在渲染之前，如果可能的话。对于不了解Spring占位符约定的消费者来说，这是一个有用的功能。

|  |
| --- |
| IMG_308 |
| 使用YAML或属性格式存在一些限制，主要是与元数据的丢失有关。例如，JSON被构造为属性源的有序列表，其名称与该源相关。即使值的来源有多个来源，YAML和属性形式也会合并到一个映射中，并且原始来源文件的名称也会丢失。同样，YAML表示也不一定是后备存储库中YAML源的忠实表示。它由一系列平面属性来源构成，并且必须对密钥的形式进行假设。 |

## **7.提供纯文本**

除了使用Environment抽象（或YAML或属性格式的抽象表示之一）之外，您的应用程序可能需要根据其环境量身定制的通用纯文本配置文件。Config Server通过位于/{application}/{profile}/{label}/{path}的附加终结点提供了这些终结点，其中application，profile和label与常规环境终结点具有相同的含义，但是path是指向以下环境的路径文件名（例如log.xml）。该端点的源文件与环境端点的定位方式相同。属性和YAML文件使用相同的搜索路径。但是，不是汇总所有匹配资源，而是仅返回第一个要匹配的资源。

找到资源后，可通过对提供的应用程序名称，配置文件和标签使用有效的Environment来解析常规格式（${…​}）的占位符。通过这种方式，资源端点与环境端点紧密集成在一起。考虑以下用于GIT或SVN存储库的示例：

application.yml

nginx.conf

nginx.conf如下所示：

server {

listen 80;

server\_name ${nginx.server.name};

}

和application.yml像这样：

nginx: server: name: example.com*---*spring: profiles: developmentnginx: server: name: develop.com

/foo/default/master/nginx.conf资源可能如下：

server {

listen 80;

server\_name example.com;

}

和/foo/development/master/nginx.conf像这样：

server {

listen 80;

server\_name develop.com;

}

|  |
| --- |
| IMG_309 |
| 与用于环境配置的源文件一样，profile用于解析文件名。因此，如果要使用特定于配置文件的文件，可以通过名为logback-development.xml（优先于logback.xml）的文件来解析/\*/development/\*/logback.xml。 |

|  |
| --- |
| IMG_310 |
| 如果不想提供label并让服务器使用默认标签，则可以提供useDefaultLabel请求参数。因此，default配置文件的前面的示例可以是/foo/default/nginx.conf?useDefaultLabel。 |

## **8.嵌入配置服务器**

Config Server最好作为独立应用程序运行。但是，如果需要，可以将其嵌入另一个应用程序。为此，请使用@EnableConfigServer批注。在这种情况下，名为spring.cloud.config.server.bootstrap的可选属性会很有用。它是一个标志，用于指示服务器是否应从其自己的远程存储库中进行配置。默认情况下，该标志为关闭状态，因为它会延迟启动。但是，当嵌入到另一个应用程序中时，以与其他任何应用程序相同的方式进行初始化是有意义的。将spring.cloud.config.server.bootstrap设置为true时，还必须使用[复合环境存储库配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "composite-environment-repositories" \o "5.1.9复合环境存储库)。例如

spring: application: name: configserver profiles: active: composite cloud: config: server: composite: - type: native search-locations: ${HOME}/Desktop/config bootstrap: **true**

|  |
| --- |
| IMG_311 |
| 如果使用引导标志，则配置服务器需要在bootstrap.yml中配置其名称和存储库URI。 |

要更改服务器端点的位置，可以（可选）设置spring.cloud.config.server.prefix（例如，/config）以在前缀下提供资源。前缀应以/开头，但不能以/结尾。它应用于Config Server中的@RequestMappings（即，在Spring Boot server.servletPath和server.contextPath前缀之下）。

如果要直接从后端存储库（而不是从配置服务器）读取应用程序的配置，则基本上需要没有端点的嵌入式配置服务器。您可以不使用@EnableConfigServer注释（设置为spring.cloud.config.server.bootstrap=true）来完全关闭端点。

## **9.推送通知和Spring Cloud Bus**

许多源代码存储库提供程序（例如Github，Gitlab，Gitea，Gitee，Gogs或Bitbucket）都通过Webhook通知您存储库中的更改。您可以通过提供者的用户界面将Webhook配置为URL和您感兴趣的一组事件。例如，[Github](https://developer.github.com/v3/activity/events/types/" \l "pushevent" \t "https://www.springcloud.cc/_top)使用POST到Webhook，其JSON主体包含提交列表和设置为push的标头（X-Github-Event）。如果在spring-cloud-config-monitor库上添加依赖项并在Config Server中激活Spring Cloud Bus，则会启用/monitor端点。

激活Webhook后，配置服务器将发送一个针对它认为可能已更改的应用程序的RefreshRemoteApplicationEvent。变化检测可以被策略化。但是，默认情况下，它会查找与应用程序名称匹配的文件中的更改（例如，foo.properties面向foo应用程序，而application.properties面向所有应用程序）。当您要覆盖此行为时，使用的策略是PropertyPathNotificationExtractor，该策略接受请求标头和正文作为参数，并返回已更改文件路径的列表。

默认配置可以与Github，Gitlab，Gitea，Gitee，Gogs或Bitbucket一起使用。除了来自Github，Gitlab，Gitee或Bitbucket的JSON通知之外，您还可以通过使用path={application}模式的形式编码的正文参数POST到/monitor来触发更改通知。这样做会向匹配{application}模式（可以包含通配符）的应用程序广播。

|  |
| --- |
| IMG_312 |
| 仅当在配置服务器和客户端应用程序中都激活了spring-cloud-bus时，才发送RefreshRemoteApplicationEvent。 |

|  |
| --- |
| IMG_313 |
| 默认配置还检测本地git存储库中的文件系统更改。在这种情况下，不使用Webhook。但是，一旦您编辑配置文件，就会广播刷新。 |

## **10. Spring Cloud Config客户**

Spring Boot应用程序可以立即利用Spring Config Server（或应用程序开发人员提供的其他外部属性源）。它还选择了与Environment更改事件相关的一些其他有用功能。

## **10.1配置第一个引导程序**

在类路径上具有Spring Cloud Config客户端的任何应用程序的默认行为如下：配置客户端启动时，它将绑定到配置服务器（通过spring.cloud.config.uri引导程序配置属性）并初始化Spring Environment（带有远程资源来源）。

此行为的最终结果是，所有要使用Config Server的客户端应用程序都需要一个bootstrap.yml（或环境变量），其服务器地址设置为spring.cloud.config.uri（默认为“ http：// localhost” ：8888“）。

## **10.2发现第一引导程序**

如果使用DiscoveryClient实现，例如Spring Cloud Netflix和Eureka Service Discovery或Spring Cloud Consul，则可以让Config Server向Discovery Service注册。但是，在默认的“ Config First ”模式下，客户端无法利用注册。

如果您更喜欢使用DiscoveryClient来查找配置服务器，则可以通过设置spring.cloud.config.discovery.enabled=true（默认值为false）来进行。这样做的最终结果是，所有客户端应用程序都需要具有适当发现配置的bootstrap.yml（或环境变量）。例如，对于Spring Cloud Netflix，您需要定义Eureka服务器地址（例如，在eureka.client.serviceUrl.defaultZone中）。使用此选项的价格是启动时需要进行额外的网络往返，以查找服务注册。好处是，只要发现服务是固定点，配置服务器就可以更改其坐标。默认服务ID是configserver，但是您可以通过设置spring.cloud.config.discovery.serviceId在客户端上（以及在服务器上，以一种通常的服务方式，例如通过设置spring.application.name）来更改该ID。

发现客户端实现均支持某种元数据映射（例如，对于Eureka，我们有eureka.instance.metadataMap）。Config Server的某些其他属性可能需要在其服务注册元数据中进行配置，以便客户端可以正确连接。如果Config Server受HTTP Basic保护，则可以将凭据配置为user和password。另外，如果Config Server具有上下文路径，则可以设置configPath。例如，以下YAML文件适用于作为Eureka客户端的Config Server：

**bootstrap.yml。**

eureka: instance:

... metadataMap: user: osufhalskjrtl password: lviuhlszvaorhvlo5847 configPath: /config

## **10.3配置客户端快速失败**

在某些情况下，如果服务无法连接到Config Server，您可能希望启动失败。如果这是期望的行为，请设置引导程序配置属性spring.cloud.config.fail-fast=true，以使客户端因Exception而停止。

## **10.4配置客户端重试**

如果您希望配置服务器在您的应用程序启动时偶尔会不可用，则可以使其在失败后继续尝试。首先，您需要设置spring.cloud.config.fail-fast=true。然后，您需要将spring-retry和spring-boot-starter-aop添加到类路径中。默认行为是重试六次，初始回退间隔为1000ms，随后的回退的指数乘数为1.1。您可以通过设置spring.cloud.config.retry.\*配置属性来配置这些属性（和其他属性）。

|  |
| --- |
| IMG_314 |
| 要完全控制重试行为，请添加ID为configServerRetryInterceptor的@Bean类型的@Bean。Spring重试有一个RetryInterceptorBuilder，它支持创建一个。 |

## **10.5查找远程配置资源**

Config Service提供来自/{application}/{profile}/{label}的属性源，其中客户端应用程序中的默认绑定如下：

* “名称” = ${spring.application.name}
* “个人资料” = ${spring.profiles.active}（实际上是Environment.getActiveProfiles()）
* “ label” =“大师”

|  |
| --- |
| IMG_315 |
| 设置属性${spring.application.name}时，请勿在您的应用名称前加上保留字application-，以防止解析正确的属性源时出现问题。 |

您可以通过设置spring.cloud.config.\*（其中\*为name，profile或label）来覆盖所有参数。label对于回滚到以前的配置版本很有用。使用默认的Config Server实现，它可以是git标签，分支名称或提交ID。标签也可以以逗号分隔的列表形式提供。在这种情况下，列表中的项目将一一尝试直到成功为止。在要素分支上工作时，此行为可能很有用。例如，您可能想使配置标签与分支对齐，但使其成为可选（在这种情况下，请使用spring.cloud.config.label=myfeature,develop）。

## **10.6为配置服务器指定多个地址**

为确保在部署了Config Server的多个实例时并希望不时有一个或多个实例不可用时的高可用性，可以指定多个URL（作为spring.cloud.config.uri属性下的逗号分隔列表）或您的所有实例都在服务注册表中注册，例如Eureka（如果使用Discovery-First Bootstrap模式）。请注意，只有在未运行Config Server时（即，应用程序退出时）或发生连接超时时，这样做才能确保高可用性。例如，如果Config Server返回500（内部服务器错误）响应，或者Config Client从Config Server收到401（由于凭据错误或其他原因），则Config Client不会尝试从其他URL提取属性。此类错误表示用户问题，而不是可用性问题。

如果您在Config Server上使用HTTP基本安全性，则仅当将凭据嵌入在spring.cloud.config.uri属性下指定的每个URL中时，当前才有可能支持per-Config Server身份验证凭据。如果使用任何其他类型的安全性机制，则（当前）不能支持每台配置服务器的身份验证和授权。

## **10.7配置超时**

如果要配置超时阈值：

* 可以使用属性spring.cloud.config.request-read-timeout配置读取超时。
* 可以使用属性spring.cloud.config.request-connect-timeout配置连接超时。

## **10.8安全性**

如果在服务器上使用HTTP基本安全性，则客户端需要知道密码（如果不是默认用户名，则需要用户名）。您可以通过配置服务器URI或通过单独的用户名和密码属性来指定用户名和密码，如以下示例所示：

**bootstrap.yml。**

spring: cloud: config: uri: https://user:secret@myconfig.mycompany.com

以下示例显示了传递相同信息的另一种方法：

**bootstrap.yml。**

spring: cloud: config: uri: https://myconfig.mycompany.com username: user password: secret

spring.cloud.config.password和spring.cloud.config.username值会覆盖URI中提供的任何内容。

如果您在Cloud Foundry上部署应用程序，则提供密码的最佳方法是通过服务凭据（例如URI中的密码），因为它不需要在配置文件中。以下示例在本地工作，并且适用于名为configserver的Cloud Foundry上的用户提供的服务：

**bootstrap.yml。**

spring: cloud: config: uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888**}**

如果您使用另一种形式的安全性，则可能需要向ConfigServicePropertySourceLocator提供一个RestTemplate（例如，通过在引导上下文中进行抓取并将其注入）。ConfigServicePropertySourceLocator [提供一个{848](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "custom-rest-template" \o "10.8.2提供自定义的RestTemplate) /}（例如，通过在引导上下文中进行抓取并将其注入）。

### **10.8.1健康指标**

Config Client提供Spring Boot运行状况指示器，该指示器尝试从Config Server加载配置。可以通过设置health.config.enabled=false禁用运行状况指示器。由于性能原因，响应也被缓存。默认的生存时间为5分钟。要更改该值，请设置health.config.time-to-live属性（以毫秒为单位）。

### **10.8.2提供自定义的RestTemplate**

在某些情况下，您可能需要自定义来自客户端对配置服务器的请求。通常，这样做涉及传递特殊的Authorization标头以验证对服务器的请求。提供自定义RestTemplate：

1. 创建一个新的配置bean，实现为PropertySourceLocator，如以下示例所示：

**CustomConfigServiceBootstrapConfiguration.java。**

@Configuration**public** **class** CustomConfigServiceBootstrapConfiguration {

@Bean

**public** ConfigServicePropertySourceLocator configServicePropertySourceLocator() {

ConfigClientProperties clientProperties = configClientProperties();

ConfigServicePropertySourceLocator configServicePropertySourceLocator = **new** ConfigServicePropertySourceLocator(clientProperties);

configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));

**return** configServicePropertySourceLocator;

}

}

1. 在resources/META-INF中，创建一个名为spring.factories的文件并指定您的自定义配置，如以下示例所示：

**spring.factories.**

org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapConfiguration

### **10.8.3 Vault**

当使用Vault作为配置服务器的后端时，客户端需要为服务器提供令牌以从Vault中检索值。可以通过在bootstrap.yml中设置spring.cloud.config.token在客户端中提供此令牌，如以下示例所示：

**bootstrap.yml。**

spring: cloud: config: token: YourVaultToken

## **10.9 Vault中的嵌套键**

Vault支持将键嵌套在Vault中存储的值中的功能，如以下示例所示：

echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -

此命令将JSON对象写入您的Vault。要访问Spring中的这些值，您将使用传统的dot（.）批注，如以下示例所示

@Value("${appA.secret}")

String name = "World";

前面的代码会将name变量的值设置为appAsecret。

# **第三部分 Spring Cloud Netflix**

**Greenwich SR5**

该项目通过自动配置并绑定到Spring环境和其他Spring编程模型习惯用法，为Spring Boot应用提供了Netflix OSS集成。使用一些简单的批注，您可以快速启用和配置应用程序内部的通用模式，并使用经过测试的Netflix组件构建大型分布式系统。提供的模式包括服务发现（Eureka），断路器（Hystrix），智能路由（Zuul）和客户端负载平衡（Ribbon）。

## **11.服务发现：Eureka个客户端**

服务发现是基于微服务的体系结构的主要宗旨之一。尝试手动配置每个客户端或某种形式的约定可能很困难并且很脆弱。Eureka是Netflix Service Discovery服务器和客户端。可以将服务器配置和部署为高可用性，每个服务器将有关已注册服务的状态复制到其他服务器。

## **11.1如何包括Eureka客户**

要将Eureka客户端包括在您的项目中，请使用启动器，其组ID为org.springframework.cloud，工件ID为​​spring-cloud-starter-netflix-eureka-client。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

## **11.2向Eureka注册**

当客户端向Eureka注册时，它会提供有关其自身的元数据-例如主机，端口，运行状况指示器URL，主页和其他详细信息。Eureka从属于服务的每个实例接收心跳消息。如果心跳在可配置的时间表上进行故障转移，则通常会将实例从注册表中删除。

以下示例显示了最小的Eureka客户端应用程序：

@SpringBootApplication@RestController**public** **class** Application {

@RequestMapping("/")

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

请注意，前面的示例显示了普通的[Spring Boot](https://projects.spring.io/spring-boot/" \t "https://www.springcloud.cc/_top)应用程序。通过在类路径上使用spring-cloud-starter-netflix-eureka-client，您的应用程序将自动在Eureka服务器中注册。如下例所示，需要进行配置才能找到Eureka服务器：

**application.yml。**

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

在前面的示例中，defaultZone是一个魔术字符串后备值，它为任何不表达首选项的客户端提供服务URL（换句话说，这是一个有用的默认值）。

|  |
| --- |
| IMG_316 |
| defaultZone属性区分大小写，并且需要使用驼峰式大小写，因为serviceUrl属性是Map<String, String>。因此，defaultZone属性不遵循default-zone的常规Spring Boot蛇形惯例。 |

默认应用程序名称（即服务ID），虚拟主机和非安全端口（从Environment获取）分别为${spring.application.name}，${spring.application.name}和${server.port}。

在类路径上具有spring-cloud-starter-netflix-eureka-client可使应用程序同时进入Eureka “ 实例 ”（即，它自己注册）和“ 客户端 ”（它可以查询注册表以定位其他服务）。实例行为由eureka.instance.\*配置键驱动，但是如果确保您的应用程序具有spring.application.name的值（这是Eureka服务ID或VIP的默认值），则默认值很好。

有关[可](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java" \t "https://www.springcloud.cc/_top)配置选项的更多详细信息，请参见[EurekaInstanceConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java" \t "https://www.springcloud.cc/_top)和[EurekaClientConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java" \t "https://www.springcloud.cc/_top)。

要禁用Eureka Discovery Client，可以将eureka.client.enabled设置为false。当spring.cloud.discovery.enabled设置为false时，Eureka Discovery Client也将被禁用。

## **11.3通过Eureka服务器进行身份验证**

如果其中一个eureka.client.serviceUrl.defaultZone URL内嵌了凭据，则HTTP基本身份验证会自动添加到您的eureka客户端（卷曲样式，如下：[http://user:password@localhost:8761/eureka](http://user:password@localhost:8761/eureka" \t "https://www.springcloud.cc/_top)）。对于更复杂的需求，您可以创建类型为DiscoveryClientOptionalArgs的@Bean并将ClientFilter实例注入其中，所有这些实例都应用于从客户端到服务器的调用。

|  |
| --- |
| IMG_317 |
| 由于Eureka中的限制，无法支持每服务器的基本身份验证凭据，因此仅使用找到的第一组凭据。 |

## **11.4状态页和运行状况指示器**

Eureka实例的状态页和运行状况指示器分别默认为/info和/health，这是Spring Boot Actuator应用程序中有用端点的默认位置。即使您使用非默认上下文路径或Servlet路径（例如server.servletPath=/custom），也需要更改这些内容，即使对于Actuator应用程序也是如此。下面的示例显示两个设置的默认值：

**application.yml。**

eureka:

instance:

statusPageUrlPath: ${server.servletPath}/info

healthCheckUrlPath: ${server.servletPath}/health

这些链接显示在客户端使用的元数据中，并在某些情况下用于确定是否将请求发送到您的应用程序，因此，如果请求准确，将很有帮助。

|  |
| --- |
| IMG_318 |
| 在Dalston中，还需要在更改该管理上下文路径时设置状态和运行状况检查URL。从Edgware开始就删除了此要求。 |

## **11.5注册安全的应用程序**

如果您希望通过HTTPS与您的应用进行联系，则可以在EurekaInstanceConfig中设置两个标志：

* eureka.instance.[nonSecurePortEnabled]=[false]
* eureka.instance.[securePortEnabled]=[true]

这样做会使Eureka发布实例信息，该实例信息显示出对安全通信的明确偏好。对于以这种方式配置的服务，Spring Cloud DiscoveryClient始终返回以https开头的URI。同样，以这种方式配置服务时，Eureka（本机）实例信息具有安全的运行状况检查URL。

由于Eureka在内部工作的方式，它仍然会为状态和主页发布非安全URL，除非您也明确地覆盖了它们。您可以使用占位符来配置eureka实例URL，如以下示例所示：

**application.yml。**

eureka:

instance:

statusPageUrl: https://${eureka.hostname}/info

healthCheckUrl: https://${eureka.hostname}/health

homePageUrl: https://${eureka.hostname}/

（请注意，${eureka.hostname}是本机占位符，仅在Eureka的更高版本中可用。您也可以使用Spring占位符来实现相同的目的，例如，使用${eureka.instance.hostName}。）

|  |
| --- |
| IMG_319 |
| 如果您的应用程序在代理后面运行，并且SSL终止在代理中（例如，如果您在Cloud Foundry或其他平台中作为服务运行），则需要确保拦截代理的“ 转发 ”标头并由应用处理。如果嵌入在Spring Boot应用程序中的Tomcat容器具有针对'X-Forwarded-\ \*'标头的显式配置，则此操作自动发生。应用程序提供的指向自身的链接错误（错误的主机，端口或协议）表明此配置错误。 |

## **11.6 Eureka的健康检查**

默认情况下，Eureka使用客户端心跳来确定客户端是否启动。除非另有说明，否则发现客户端不会根据Spring Boot Actuator传播应用程序的当前运行状况检查状态。因此，在成功注册之后，Eureka始终宣布该应用程序处于“启动”状态。可以通过启用Eureka运行状况检查来更改此行为，这将导致应用程序状态传播到Eureka。结果，所有其他应用程序都不会将流量发送到处于“ UP”状态以外的其他状态的应用程序。以下示例显示如何为客户端启用运行状况检查：

**application.yml。**

eureka:

client:

healthcheck:

enabled: true

|  |
| --- |
| IMG_320 |
| eureka.client.healthcheck.enabled=true仅应在application.yml中设置。在bootstrap.yml中设置该值会导致不良的副作用，例如在Eureka中以UNKNOWN状态注册。 |

如果您需要对运行状况检查进行更多控制，请考虑实施自己的com.netflix.appinfo.HealthCheckHandler。

## **11.7 Eureka实例和客户端的元数据**

值得花费一些时间来了解Eureka元数据的工作方式，因此您可以在平台上使用有意义的方式来使用它。有用于信息的标准元数据，例如主机名，IP地址，端口号，状态页和运行状况检查。这些都发布在服务注册表中，并由客户端用于以直接方式联系服务。可以将其他元数据添加到eureka.instance.metadataMap中的实例注册中，并且可以在远程客户端中访问此元数据。通常，除非使客户端知道元数据的含义，否则其他元数据不会更改客户端的行为。有几种特殊情况，在本文档的后面部分进行介绍，其中Spring Cloud已经为元数据映射分配了含义。

### **11.7.1在Cloud Foundry上使用Eureka**

Cloud Foundry具有全局路由器，因此同一应用程序的所有实例都具有相同的主机名（其他具有类似体系结构的PaaS解决方案具有相同的排列）。这不一定是使用Eureka的障碍。但是，如果您使用路由器（建议或什至是强制性的，具体取决于平台的设置方式），则需要显式设置主机名和端口号（安全或不安全），以便它们使用路由器。您可能还希望使用实例元数据，以便可以区分客户端上的实例（例如，在自定义负载平衡器中）。默认情况下，eureka.instance.instanceId为vcap.application.instance\_id，如以下示例所示：

**application.yml。**

eureka:

instance:

hostname: ${vcap.application.uris[0]}

nonSecurePort: 80

根据在Cloud Foundry实例中设置安全规则的方式，您可能可以注册并使用主机VM的IP地址进行直接的服务到服务的调用。Pivotal Web服务（[PWS](https://run.pivotal.io/" \t "https://www.springcloud.cc/_top)）尚不提供此功能。

### **11.7.2在AWS上使用Eureka**

如果计划将应用程序部署到AWS云，则必须将Eureka实例配置为可感知AWS。您可以通过如下自定义[EurekaInstanceConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java" \t "https://www.springcloud.cc/_top)来实现：

@Bean@Profile("!default")**public** EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {

EurekaInstanceConfigBean b = **new** EurekaInstanceConfigBean(inetUtils);

AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");

b.setDataCenterInfo(info);

**return** b;

}

### **11.7.3更改Eureka实例ID**

一个普通的Netflix Eureka实例注册的ID等于其主机名（即，每个主机仅提供一项服务）。Spring Cloud Eureka提供了明智的默认值，其定义如下：

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

一个示例是myhost:myappname:8080。

通过使用Spring Cloud，可以通过在eureka.instance.instanceId中提供唯一标识符来覆盖此值，如以下示例所示：

**application.yml。**

eureka:

instance:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

通过前面示例中显示的元数据和在本地主机上部署的多个服务实例，在其中插入随机值以使实例唯一。在Cloud Foundry中，vcap.application.instance\_id是在Spring Boot应用程序中自动填充的，因此不需要随机值。

## **11.8使用EurekaClient**

一旦拥有作为发现客户端的应用程序，就可以使用它从[Eureka服务器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server" \o "12.服务发现：Eureka服务器)发现服务实例。一种方法是使用本地com.netflix.discovery.EurekaClient（而不是Spring Cloud DiscoveryClient），如以下示例所示：

@Autowired

private EurekaClient discoveryClient;

public String serviceUrl() {

InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);

return instance.getHomePageUrl();

}

|  |
| --- |
| IMG_321 |
| 请勿在@PostConstruct方法或@Scheduled方法（或可能尚未启动ApplicationContext的任何地方）中使用EurekaClient。它是通过SmartLifecycle（带有phase=0）进行初始化的，因此最早可以依靠它的是处于更高阶段的另一个SmartLifecycle。 |

### **11.8.1没有Jersey的EurekaClient**

默认情况下，EurekaClient使用Jersey进行HTTP通信。如果希望避免来自Jersey的依赖关系，可以将其从依赖关系中排除。Spring Cloud基于Spring RestTemplate自动配置传输客户端。以下示例显示Jersey被排除在外：

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

<exclusions>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-client</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-core</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey.contribs</groupId>

<artifactId>jersey-apache-client4</artifactId>

</exclusion>

</exclusions>

</dependency>

## **11.9替代本机Netflix EurekaClient**

您无需使用原始Netflix EurekaClient。而且，通常在某种包装器后面使用它会更方便。Spring Cloud 通过逻辑Eureka服务标识符（VIP）而非物理URL 支持[Feign](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-feign" \o "23.声明式REST客户端：伪装)（REST客户端生成器）和[Spring RestTemplate](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon" \o "16.客户端负载均衡器：功能区)。要使用固定的物理服务器列表配置Ribbon，可以将<client>.ribbon.listOfServers设置为以逗号分隔的物理地址（或主机名）列表，其中<client>是客户端的ID。

您还可以使用org.springframework.cloud.client.discovery.DiscoveryClient，它为发现客户端提供一个简单的API（非Netflix专用），如以下示例所示：

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

}

## **11.10为什么注册服务这么慢？**

成为实例还涉及到注册表的定期心跳（通过客户端的serviceUrl），默认持续时间为30秒。直到实例，服务器和客户端在其本地缓存中都具有相同的元数据后，客户端才能发现该服务（因此可能需要3个心跳）。您可以通过设置eureka.instance.leaseRenewalIntervalInSeconds来更改周期。将其设置为小于30的值可以加快使客户端连接到其他服务的过程。在生产中，最好使用默认值，因为服务器中的内部计算对租约续订期进行了假设。

## **11.11区域**

如果您已将Eureka客户端部署到多个区域，则您可能希望这些客户端在尝试使用其他区域中的服务之前先使用同一区域中的服务。要进行设置，您需要正确配置Eureka客户端。

首先，您需要确保已将Eureka服务器部署到每个区域，并且它们彼此对等。有关 更多信息，请参见[区域和区域](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-eureka-server-zones-and-regions" \o "12.3高可用性，区域和区域)部分。

接下来，您需要告诉Eureka服务位于哪个区域。您可以使用metadataMap属性来做到这一点。例如，如果将service 1部署到zone 1和zone 2上，则需要在service 1中设置以下Eureka属性：

**1区服务1**

eureka.instance.metadataMap.zone = zone1

eureka.client.preferSameZoneEureka = true

**2区服务1**

eureka.instance.metadataMap.zone = zone2

eureka.client.preferSameZoneEureka = true

## **11.12刷新Eureka个客户端**

默认情况下，EurekaClient bean是可刷新的，这意味着Eureka客户端属性可以更改和刷新。发生刷新时，客户端将从Eureka服务器中注销，并且可能会在短暂的时间内不提供给定服务的所有实例。消除这种情况的一种方法是禁用刷新Eureka客户端的功能。为此，请设置eureka.client.refresh.enable=false。

## **12.服务发现：Eureka服务器**

本节介绍如何设置Eureka服务器。

## **12.1如何包括Eureka服务器**

要将Eureka服务器包含在您的项目中，请使用启动器，其组ID为org.springframework.cloud，工件ID为​​spring-cloud-starter-netflix-eureka-server。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_322 |
| 如果您的项目已经使用Thymeleaf作为模板引擎，则Eureka服务器的Freemarker模板可能无法正确加载。在这种情况下，必须手动配置模板加载器： |

**application.yml。**

spring:

freemarker:

template-loader-path: classpath:/templates/

prefer-file-system-access: false

## **12.2如何运行Eureka服务器**

以下示例显示了最小的Eureka服务器：

@SpringBootApplication@EnableEurekaServer**public** **class** Application {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

该服务器具有一个主页，其中包含UI和HTTP API端点，用于/eureka/\*下的常规Eureka功能。

以下链接提供了一些Eureka背景知识：[磁通电容器](https://github.com/cfregly/fluxcapacitor/wiki/NetflixOSS-FAQ" \l "eureka-service-discovery-load-balancer" \t "https://www.springcloud.cc/_top)和[google小组讨论](https://groups.google.com/forum/?fromgroups" \l "!topic/eureka_netflix/g3p2r7gHnN0" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_323 |
| 由于Gradle的依赖性解析规则以及缺少父bom功能，取决于spring-cloud-starter-netflix-eureka-server可能会导致应用程序启动失败。要解决此问题，请添加Spring Boot Gradle插件并按如下所示导入Spring Cloud Starter父Bom：  **build.gradle。**  buildscript {  dependencies {  classpath("org.springframework.boot:spring-boot-gradle-plugin:{spring-boot-docs-version}")  }  }  apply plugin: "spring-boot"  dependencyManagement {  imports {  mavenBom "org.springframework.cloud:spring-cloud-dependencies:{spring-cloud-version}"  }  } |

## **12.3高可用性，区域和区域**

Eureka服务器没有后端存储，但是注册表中的所有服务实例都必须发送心跳信号以使其注册保持最新（因此可以在内存中完成）。客户端还具有Eureka注册的内存缓存（因此，对于每个对服务的请求，它们都不必转到注册表）。

默认情况下，每个Eureka服务器也是Eureka客户端，并且需要（至少一个）服务URL来定位对等方。如果您不提供该服务，则该服务将运行并运行，但是它将使您的日志充满关于无法向对等方注册的噪音。

另请参阅[下面的内容，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon" \o "16.客户端负载均衡器：功能区)了解客户端在[Ribbon中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon" \o "16. Client Side Load Balancer: Ribbon)对区域和区域的支持。

## **12.4独立模式**

只要存在某种监视器或弹性运行时（例如Cloud Foundry），两个缓存（客户端和服务器）和心跳的组合就可以使独立的Eureka服务器对故障具有相当的恢复能力。在独立模式下，您可能希望关闭客户端行为，以使其不会继续尝试并无法到达其对等对象。下面的示例演示如何关闭客户端行为：

**application.yml（独立Eureka服务器）。**

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

请注意，serviceUrl指向与本地实例相同的主机。

## **12.5同行意识**

通过运行多个实例并要求它们彼此注册，可以使Eureka更具弹性和可用性。实际上，这是默认行为，因此要使其正常工作，您需要做的就是向对等体添加有效的serviceUrl，如以下示例所示：

**application.yml（两个对等感知Eureka服务器）。**

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

client:

serviceUrl:

defaultZone: http://peer2/eureka/

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

client:

serviceUrl:

defaultZone: http://peer1/eureka/

在前面的示例中，我们有一个YAML文件，该文件可以通过在不同的Spring配置文件中运行，在两个主机（peer1和peer2）上运行同一服务器。您可以通过操纵/etc/hosts解析主机名来使用此配置来测试单个主机上的对等感知（在生产环境中这样做没有太大价值）。实际上，如果您在知道其主机名的计算机上运行，​​则不需要eureka.instance.hostname（默认情况下，使用java.net.InetAddress进行查找）。

您可以将多个对等方添加到系统，并且只要它们都通过至少一个边缘相互连接，它们就可以在彼此之间同步注册。如果对等方在物理上是分开的（在一个数据中心内部或在多个数据中心之间），则该系统原则上可以解决“ 裂脑 ”型故障。您可以将多个对等方添加到系统中，并且只要它们都直接相互连接，它们就可以在彼此之间同步注册。

**application.yml（三个对等感知Eureka服务器）。**

eureka:

client:

serviceUrl:

defaultZone: http://peer1/eureka/,http://peer2/eureka/,http://peer3/eureka/

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

---

spring:

profiles: peer3

eureka:

instance:

hostname: peer3

## **12.6何时首选IP地址**

在某些情况下，Eureka最好公布服务的IP地址而不是主机名。将eureka.instance.preferIpAddress设置为true，并且当应用程序向eureka注册时，它将使用其IP地址而不是其主机名。

|  |
| --- |
| IMG_324 |
| 如果Java无法确定主机名，则IP地址将发送到Eureka。设置主机名的唯一明确方法是设置eureka.instance.hostname属性。您可以在运行时使用环境变量（例如，eureka.instance.hostname=${HOST\_NAME}）设置主机名。 |

## **12.7保护Eureka服务器**

您只需通过spring-boot-starter-security将Spring Security添加到服务器的类路径中即可保护Eureka服务器。默认情况下，当Spring Security在类路径上时，它将要求在每次向应用程序发送请求时都发送有效的CSRF令牌。Eureka客户通常不会拥有有效的跨站点请求伪造（CSRF）令牌，您需要为/eureka/\*\*端点禁用此要求。例如：

@EnableWebSecurity**class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.csrf().ignoringAntMatchers("/eureka/\*\*");

**super**.configure(http);

}

}

有关CSRF的更多信息，请参见[Spring Security文档](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/" \l "csrf" \t "https://www.springcloud.cc/_top)。

可以在Spring Cloud示例存储[库中](https://github.com/spring-cloud-samples/eureka/tree/Eureka-With-Security" \t "https://www.springcloud.cc/_top)找到Eureka演示服务器。

## **12.8 JDK 11支持**

Eureka服务器所依赖的JAXB模块已在JDK 11中删除。如果要在运行Eureka服务器时使用JDK 11，则必须在POM或Gradle文件中包括这些依赖项。

<dependency>

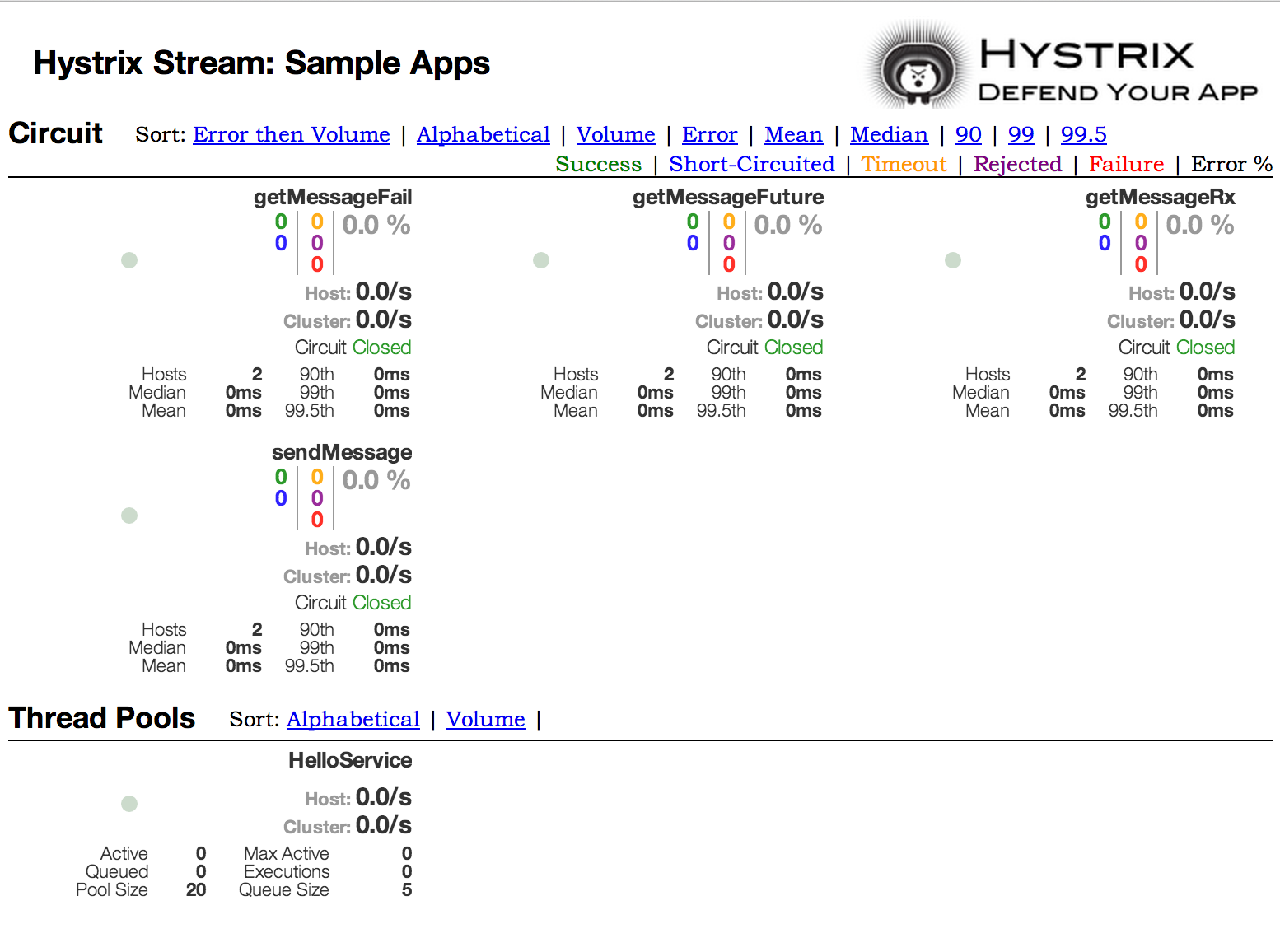
<groupId>org.glassfish.jaxb</groupId>

<artifactId>jaxb-runtime</artifactId></dependency>

## **13.断路器：Hystrix个客户**

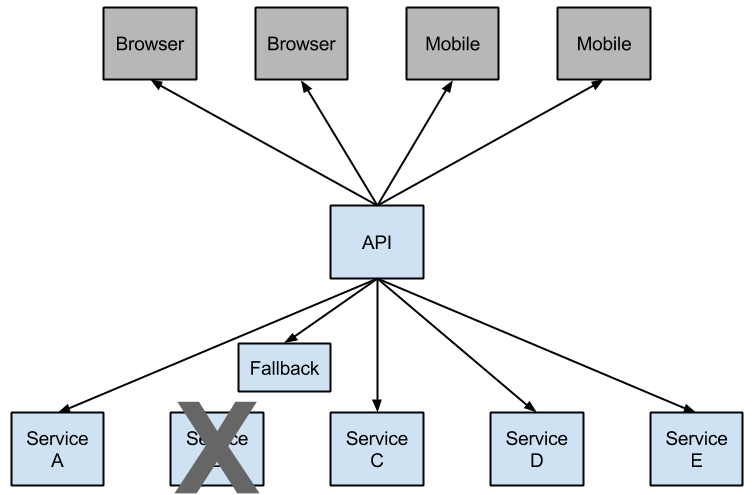
Netflix创建了一个名为[Hystrix](https://github.com/Netflix/Hystrix" \t "https://www.springcloud.cc/_top)的库，该库实现了[断路器模式](https://martinfowler.com/bliki/CircuitBreaker.html" \t "https://www.springcloud.cc/_top)。在微服务架构中，通常有多个服务调用层，如以下示例所示：

**图13.1。微服务图**



较低级别的服务中的服务故障可能会导致级联故障，直至用户。在metrics.rollingStats.timeInMilliseconds定义的滚动窗口中，当对特定服务的调用超过circuitBreaker.requestVolumeThreshold（默认：20个请求）并且失败百分比大于circuitBreaker.errorThresholdPercentage（默认：> 50％）时（默认：10秒） ），则电路断开并且无法进行呼叫。在错误和断路的情况下，开发人员可以提供备用功能。

**图13.2 Hystrix后备可防止级联故障**



开路可停止级联故障，并让不堪重负的服务时间得以恢复。后备可以是另一个受Hystrix保护的呼叫，静态数据或合理的空值。可以将回退链接在一起，以便第一个回退进行其他业务调用，然后回退到静态数据。

## **13.1如何包括Hystrix**

要将Hystrix包含在您的项目中，请使用起始者，其组ID为org.springframework.cloud，工件ID为​​spring-cloud-starter-netflix-hystrix。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

以下示例显示了具有Hystrix断路器的最小Eureka服务器：

@SpringBootApplication

@EnableCircuitBreaker

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

@Component

public class StoreIntegration {

@HystrixCommand(fallbackMethod = "defaultStores")

public Object getStores(Map<String, Object> parameters) {

//do stuff that might fail

}

public Object defaultStores(Map<String, Object> parameters) {

return /\* something useful \*/;

}

}

@HystrixCommand由一个名为[“ javanica ”](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica" \t "https://www.springcloud.cc/_top)的Netflix contrib库提供。Spring Cloud将带有注释的Spring beans自动包装在与Hystrix断路器连接的代理中。断路器计算何时断开和闭合电路，以及在发生故障时应采取的措施。

要配置@HystrixCommand，可以将commandProperties属性与@HystrixProperty批注一起使用。有关 更多详细信息，请参见 [此处](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica" \l "configuration" \t "https://www.springcloud.cc/_top)。有关 可用属性的详细信息，请参见[Hystrix Wiki](https://github.com/Netflix/Hystrix/wiki/Configuration" \t "https://www.springcloud.cc/_top)。

## **13.2传播安全上下文或使用Spring范围**

如果要将某些线程本地上下文传播到@HystrixCommand中，则默认声明无效，因为默认声明在线程池中执行命令（如果超时）。通过要求Hystrix使用不同的“ 隔离策略 ”，可以通过配置或直接在批注中切换Hystrix来使用与调用方相同的线程。下面的示例演示了如何在注释中设置线程：

@HystrixCommand(fallbackMethod = "stubMyService",

commandProperties = {

@HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")

}

)

...

如果使用@SessionScope或@RequestScope，则同样适用。如果遇到运行时异常，提示它找不到范围内的上下文，则需要使用同一线程。

您还可以选择将hystrix.shareSecurityContext属性设置为true。这样做会自动配置一个Hystrix并发策略插件挂钩，以将SecurityContext从您的主线程转移到Hystrix命令所使用的那个线程。Hystrix不允许注册多个Hystrix并发策略，因此可以通过将自己的HystrixConcurrencyStrategy声明为Spring bean来使用扩展机制。Spring Cloud在Spring上下文中寻找您的实现，并将其包装在自己的插件中。

## **13.3健康指标**

连接的断路器的状态也显示在调用应用程序的/health端点中，如以下示例所示：

**{**

"hystrix": **{**

"openCircuitBreakers": **[**

"StoreIntegration::getStoresByLocationLink"

]**,**

"status": "CIRCUIT\_OPEN"

**},**

"status": "UP"**}**

## **13.4 Hystrix指标流**

要启用Hystrix指标流，请包括对spring-boot-starter-actuator的依赖性并设置management.endpoints.web.exposure.include: hystrix.stream。这样做将/actuator/hystrix.stream作为管理端点公开，如以下示例所示：

<dependency>

<groupId>org.springframework.boot</groupId>

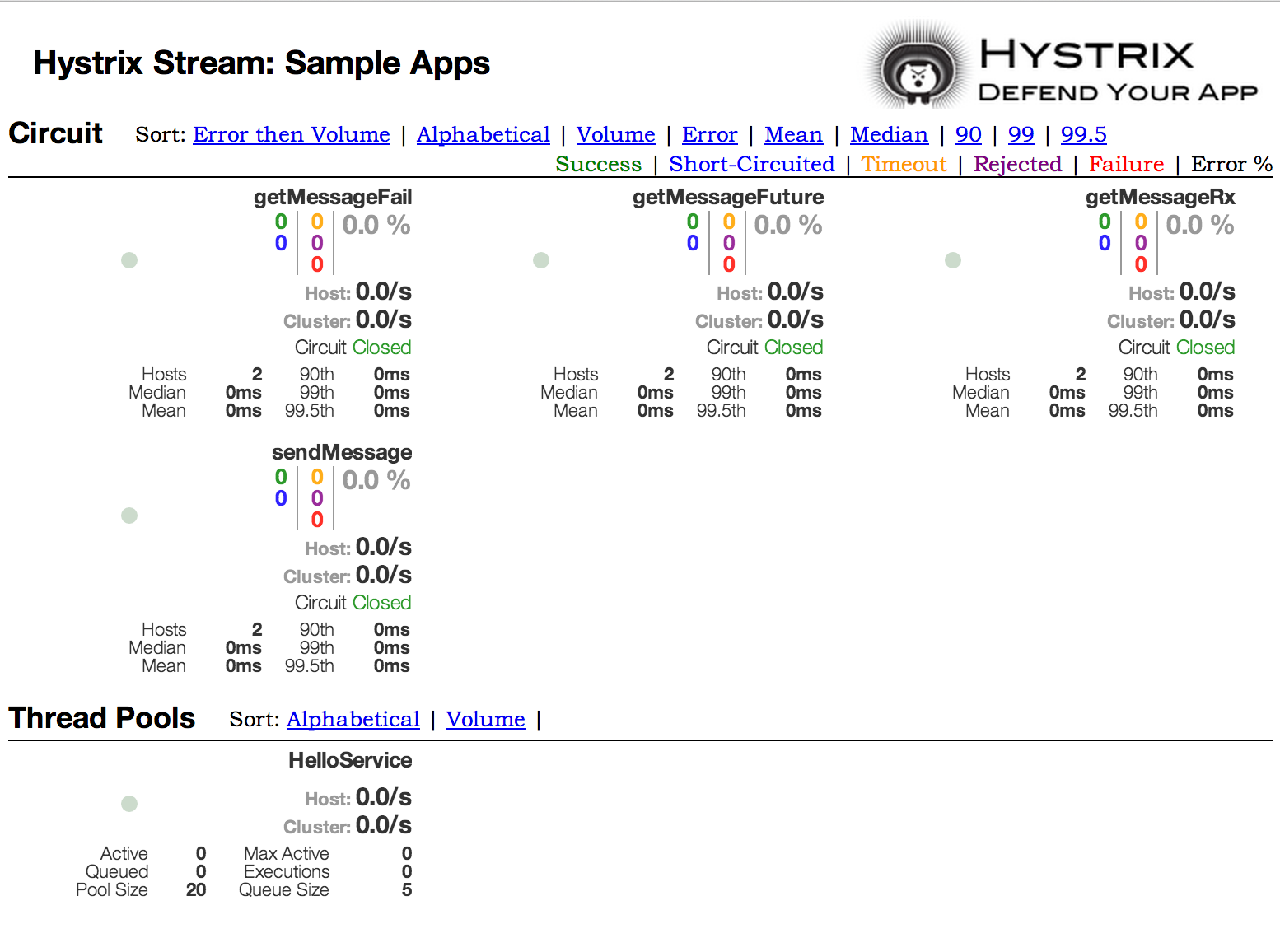
<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

## **14.断路器：Hystrix仪表板**

Hystrix的主要好处之一是它收集的有关每个HystrixCommand的一组度量。Hystrix仪表板以有效的方式显示每个断路器的运行状况。

**图14.1 Hystrix资讯主页**



## **15. Hystrix超时和Ribbon客户**

当使用包裹Ribbon客户端的Hystrix命令时，您要确保Hystrix超时配置为比配置的Ribbon超时更长，包括可能进行的任何重试。例如，如果您的Ribbon连接超时是一秒钟，并且Ribbon客户端可能重试了3次请求，则Hystrix超时应该稍微超过3秒。

## **15.1如何包括Hystrix信息中心**

要将Hystrix仪表板包含在您的项目中，请使用启动器，其组ID为org.springframework.cloud，工件ID为​​spring-cloud-starter-netflix-hystrix-dashboard。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

要运行Hystrix信息中心，请用@EnableHystrixDashboard注释Spring Boot主类。然后访问/hystrix，并将仪表板指向Hystrix客户端应用程序中单个实例的/hystrix.stream端点。

|  |
| --- |
| IMG_328 |
| 连接到使用HTTPS的/hystrix.stream端点时，JVM必须信任服务器使用的证书。如果证书不受信任，则必须将证书导入JVM，以使Hystrix仪表板成功连接到流终结点。 |

## **15.2 Turbine**

从系统的整体运行状况来看，查看单个实例的Hystrix数据不是很有用。[Turbine](https://github.com/Netflix/Turbine" \t "https://www.springcloud.cc/_top)是一个应用程序，它将所有相关的/hystrix.stream端点聚合到一个组合的/turbine.stream中，以便在Hystrix仪表板中使用。个别实例通过Eureka定位。运行Turbine需要使用@EnableTurbine注释对您的主类进行注释（例如，通过使用spring-cloud-starter-netflix-turbine设置类路径）。[Turbine 1 Wiki中](https://github.com/Netflix/Turbine/wiki/Configuration-(1.x)" \t "https://www.springcloud.cc/_top)记录的所有配置属性均适用。唯一的区别是turbine.instanceUrlSuffix不需要预先添加的端口，因为除非turbine.instanceInsertPort=false，否则它将自动处理。

|  |
| --- |
| IMG_329 |
| 默认情况下，Turbine通过在Eureka中查找其hostName和port条目，然后将/hystrix.stream附加到已注册实例上来查找/hystrix.stream端点。如果实例的元数据包含management.port，则使用它代替/hystrix.stream端点的port值。默认情况下，名为management.port的元数据条目等于management.port配置属性。可以使用以下配置覆盖它： |

eureka:

instance:

metadata-map:

management.port: ${management.port:8081}

turbine.appConfig配置密钥是Eureka serviceId的列表，涡轮使用它们来查找实例。然后，在Hystrix仪表板中使用涡轮流，其URL类似于以下内容：

[https://my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME](https://my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME" \t "https://www.springcloud.cc/_top)

如果名称为default，则可以省略cluster参数。cluster参数必须与turbine.aggregator.clusterConfig中的条目匹配。从Eureka返回的值是大写的。因此，如果存在一个向Eureka注册的名为customers的应用程序，则以下示例可用：

turbine:

aggregator:

clusterConfig: CUSTOMERS

appConfig: customers

如果您需要自定义Turbine应该使用哪些集群名称（因为您不想在turbine.aggregator.clusterConfig配置中存储集群名称），请提供类型为TurbineClustersProvider的bean。

clusterName可以通过turbine.clusterNameExpression中的SPEL表达式进行自定义，其中根目录为InstanceInfo的实例。默认值为appName，这意味着Eureka serviceId成为群集密钥（即，客户的InstanceInfo的appName为CUSTOMERS）。一个不同的示例是turbine.clusterNameExpression=aSGName，它从AWS ASG名称获取集群名称。以下清单显示了另一个示例：

turbine:

aggregator:

clusterConfig: SYSTEM,USER

appConfig: customers,stores,ui,admin

clusterNameExpression: metadata['cluster']

在前面的示例中，来自四个服务的群集名称是从它们的元数据映射中拉出的，并且期望其值包括SYSTEM和USER。

要将“ 默认 ”群集用于所有应用程序，您需要一个字符串文字表达式（如果在YAML中，也要使用单引号和双引号进行转义）：

turbine:

appConfig: customers,stores

clusterNameExpression: "'default'"

Spring Cloud提供了spring-cloud-starter-netflix-turbine，它具有运行Turbine服务器所需的所有依赖关系。要添加Turbine，请创建一个Spring Boot应用程序并使用@EnableTurbine对其进行注释。

|  |
| --- |
| IMG_330 |
| 默认情况下，Spring Cloud允许Turbine使用主机和端口以允许每个主机，每个集群多个进程。如果你想建立在本地Netflix的行为Turbine，以不使每台主机的多个进程，每簇（关键实例ID是主机名），集合turbine.combineHostPort=false。 |

### **15.2.1集群端点**

在某些情况下，其他应用程序了解在Turbine中配置了哪些custers可能会很有用。为此，您可以使用/clusters端点，该端点将返回所有已配置集群的JSON数组。

**GET /集群。**

**[**

**{**

"name": "RACES"**,**

"link": "http://localhost:8383/turbine.stream?cluster=RACES"

**},**

**{**

"name": "WEB"**,**

"link": "http://localhost:8383/turbine.stream?cluster=WEB"

**}]**

可以通过将turbine.endpoints.clusters.enabled设置为false来禁用此端点。

## **15.3 Turbine Stream**

在某些环境中（例如在PaaS设置中），从所有分布式Hystrix命令中提取指标的经典Turbine模型不起作用。在这种情况下，您可能想让Hystrix命令将指标推送到Turbine。Spring Cloud通过消息传递实现了这一点。要在客户端上执行此操作，请向spring-cloud-netflix-hystrix-stream和您选择的spring-cloud-starter-stream-\*添加一个依赖项。有关代理以及如何配置客户端凭据的详细信息，请参见[Spring Cloud Stream文档](https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/" \t "https://www.springcloud.cc/_top)。对于本地代理，它应该开箱即用。

在服务器端，创建一个Spring Boot应用程序，并用@EnableTurbineStream对其进行注释。Turbine Stream服务器需要使用Spring Webflux，因此，spring-boot-starter-webflux必须包含在您的项目中。将spring-cloud-starter-netflix-turbine-stream添加到您的应用程序时，默认包含spring-boot-starter-webflux。

然后，您可以将Hystrix仪表板指向Turbine Stream服务器，而不是单独的Hystrix流。如果Turbine Stream在myhost的端口8989上运行，则将[http://myhost:8989](http://myhost:8989/" \t "https://www.springcloud.cc/_top)放在Hystrix仪表板的流输入字段中。电路以其各自的serviceId为前缀，后跟一个点（.），然后是电路名称。

Spring Cloud提供了spring-cloud-starter-netflix-turbine-stream，其中包含使Turbine Stream服务器运行所需的所有依赖项。然后，您可以添加您选择的流绑定程序，例如spring-cloud-starter-stream-rabbit。

Turbine Stream服务器还支持cluster参数。与Turbine服务器不同，Turbine Stream使用eureka serviceIds作为群集名称，并且这些名称不可配置。

如果Turbine Stream服务器在my.turbine.server的端口8989上运行，并且您的环境中有两个eureka serviceId customers和products，则以下URL将在Turbine Stream服务器上可用。default和空群集名称将提供Turbine Stream服务器接收的所有度量。

https://my.turbine.sever:8989/turbine.stream?cluster=customers

https://my.turbine.sever:8989/turbine.stream?cluster=products

https://my.turbine.sever:8989/turbine.stream?cluster=default

https://my.turbine.sever:8989/turbine.stream

因此，您可以将eureka serviceIds用作Turbine仪表板（或任何兼容的仪表板）的群集名称。您无需为Turbine Stream服务器配置任何属性，例如turbine.appConfig，turbine.clusterNameExpression和turbine.aggregator.clusterConfig。

|  |
| --- |
| IMG_331 |
| Turbine Stream服务器使用Spring Cloud Stream从配置的输入通道中收集所有度量。这意味着它不会从每个实例中主动收集Hystrix指标。它仅可以提供每个实例已经收集到输入通道中的度量。 |

## **16.客户端负载平衡器：Ribbon**

Ribbon是一种客户端负载平衡器，可让您对HTTP和TCP客户端的行为进行大量控制。Feign已使用Ribbon，因此，如果使用@FeignClient，则本节也适用。

Ribbon中的中心概念是指定客户的概念。每个负载均衡器都是组件的一部分，这些组件可以一起工作以按需联系远程服务器，并且该组件具有您作为应用程序开发人员提供的名称（例如，使用@FeignClient批注）。根据需要，Spring Cloud通过使用RibbonClientConfiguration为每个命名的客户端创建一个新的集合作为ApplicationContext。其中包含ILoadBalancer，RestClient和ServerListFilter。

## **16.1如何包括Ribbon**

要将Ribbon包含在您的项目中，请使用起始者，其组ID为org.springframework.cloud，工件ID为​​spring-cloud-starter-netflix-ribbon。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

## **16.2自定义Ribbon客户端**

您可以使用<client>.ribbon.\*中的外部属性来配置Ribbon客户端的某些位，这与本地使用Netflix API相似，不同之处在于可以使用Spring Boot配置文件。可以将本机选项检查为[CommonClientConfigKey](https://github.com/Netflix/ribbon/blob/master/ribbon-core/src/main/java/com/netflix/client/config/CommonClientConfigKey.java" \t "https://www.springcloud.cc/_top)（功能区核心的一部分）中的静态字段。

Spring Cloud还允许您通过使用@RibbonClient声明其他配置（在RibbonClientConfiguration之上）来完全控制客户端，如以下示例所示：

@Configuration@RibbonClient(name = "custom", configuration = CustomConfiguration.class)**public** **class** TestConfiguration {

}

在这种情况下，客户端由RibbonClientConfiguration中已有的组件以及CustomConfiguration中的任何组件组成（其中后者通常会覆盖前者）。

|  |
| --- |
| IMG_332 |
| CustomConfiguration类必须是@Configuration类，但请注意，对于主应用程序上下文，它不在@ComponentScan中。否则，它由所有@RibbonClients共享。如果您使用@ComponentScan（或@SpringBootApplication），则需要采取措施避免将其包括在内（例如，可以将其放在单独的，不重叠的程序包中，或指定要在@ComponentScan）。 |

下表显示了Spring Cloud Netflix默认为Ribbon提供的beans：

| **Bean类型** | **Bean名称** | **班级名称** |
| --- | --- | --- |
| IClientConfig | ribbonClientConfig | DefaultClientConfigImpl |
| IRule | ribbonRule | ZoneAvoidanceRule |
| IPing | ribbonPing | DummyPing |
| ServerList<Server> | ribbonServerList | ConfigurationBasedServerList |
| ServerListFilter<Server> | ribbonServerListFilter | ZonePreferenceServerListFilter |
| ILoadBalancer | ribbonLoadBalancer | ZoneAwareLoadBalancer |
| ServerListUpdater | ribbonServerListUpdater | PollingServerListUpdater |

创建其中一种类型的bean并将其放置在@RibbonClient配置中（例如上述FooConfiguration），您可以覆盖所描述的每个beans，如以下示例所示：

@Configuration**protected** **static** **class** FooConfiguration {

@Bean

**public** ZonePreferenceServerListFilter serverListFilter() {

ZonePreferenceServerListFilter filter = **new** ZonePreferenceServerListFilter();

filter.setZone("myTestZone");

**return** filter;

}

@Bean

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

}

上一示例中的include语句将NoOpPing替换为PingUrl，并提供了自定义serverListFilter。

## **16.3为所有Ribbon客户端自定义默认值**

通过使用@RibbonClients批注并注册默认配置，可以为所有Ribbon客户端提供默认配置，如以下示例所示：

@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)**public** **class** RibbonClientDefaultConfigurationTestsConfig {

**public** **static** **class** BazServiceList **extends** ConfigurationBasedServerList {

**public** BazServiceList(IClientConfig config) {

**super**.initWithNiwsConfig(config);

}

}

}

@Configuration**class** DefaultRibbonConfig {

@Bean

**public** IRule ribbonRule() {

**return** **new** BestAvailableRule();

}

@Bean

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

@Bean

**public** ServerList<Server> ribbonServerList(IClientConfig config) {

**return** **new** RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);

}

@Bean

**public** ServerListSubsetFilter serverListFilter() {

ServerListSubsetFilter filter = **new** ServerListSubsetFilter();

**return** filter;

}

}

## **16.4通过设置Properties自定义Ribbon客户端**

从版本1.2.0开始，Spring Cloud Netflix现在支持通过将属性设置为与[Ribbon文档](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers" \l "components-of-load-balancer" \t "https://www.springcloud.cc/_top)兼容来自定义Ribbon客户端。

这使您可以在启动时在不同环境中更改行为。

以下列表显示了受支持的属性>：

* <clientName>.ribbon.NFLoadBalancerClassName：应实施ILoadBalancer
* <clientName>.ribbon.NFLoadBalancerRuleClassName：应实施IRule
* <clientName>.ribbon.NFLoadBalancerPingClassName：应实施IPing
* <clientName>.ribbon.NIWSServerListClassName：应实施ServerList
* <clientName>.ribbon.NIWSServerListFilterClassName：应实施ServerListFilter

|  |
| --- |
| IMG_333 |
| 这些属性中定义的类优先于使用@RibbonClient(configuration=MyRibbonConfig.class)定义的beans和Spring Cloud Netflix提供的默认值。 |

要为名为users的服务名称设置IRule，可以设置以下属性：

**application.yml。**

users:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule

有关[Ribbon](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers" \t "https://www.springcloud.cc/_top)提供的实现，请参见[Ribbon文档](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers" \t "https://www.springcloud.cc/_top)。

## **16.5将Ribbon与Eureka一起使用**

当Eureka与Ribbon结合使用时（也就是说，两者都在类路径上），ribbonServerList被扩展名DiscoveryEnabledNIWSServerList覆盖，这将填充{71中的服务器列表/}。它还用NIWSDiscoveryPing替换了IPing接口，该接口委托给Eureka确定服务器是否启动。默认安装的ServerList是DomainExtractingServerList。其目的是不使用AWS AMI元数据（这就是Netflix所依赖的）使元数据可用于负载均衡器。默认情况下，服务器列表是使用实例元数据中提供的“ zone ”信息构建的（因此，在远程客户端上，设置为eureka.instance.metadataMap.zone）。如果缺少该字段，并且设置了approximateZoneFromHostname标志，则它可以使用服务器主机名中的域名作为该区域的代理。一旦区域信息可用，就可以在ServerListFilter中使用它。默认情况下，它用于在与客户端相同的区域中定位服务器，因为默认值为ZonePreferenceServerListFilter。默认情况下，以与远程实例相同的方式（即通过eureka.instance.metadataMap.zone）确定客户端的区域。

|  |
| --- |
| IMG_334 |
| 设置客户端区域的传统“ archaius ”方法是通过名为“ @zone”的配置属性。如果可用，Spring Cloud优先于所有其他设置使用该设置（请注意，该键必须在YAML配置中用引号引起来）。 |

|  |
| --- |
| IMG_335 |
| 如果没有其他区域数据源，则根据客户端配置（而不是实例配置）进行猜测。我们取eureka.client.availabilityZones（这是从区域名称到区域列表的映射），然后为实例自己的区域拉出第一个区域（即eureka.client.region，其默认值为“ us-east-1” ”，以与本机Netflix兼容）。 |

## **16.6示例：如何在没有Eureka的情况下使用Ribbon**

Eureka是一种抽象发现远程服务器的便捷方法，因此您不必在客户端中对它们的URL进行硬编码。但是，如果您不想使用Eureka，则Ribbon和Feign也可以使用。假设您为“商店”声明了@RibbonClient，并且Eureka未被使用（甚至不在类路径上）。Ribbon客户端默认为配置的服务器列表。您可以提供以下配置：

**application.yml。**

stores:

ribbon:

listOfServers: example.com,google.com

## **16.7示例：在Ribbon中禁用Eureka**

将ribbon.eureka.enabled属性设置为false会显式禁用Ribbon中的Eureka，如以下示例所示：

**application.yml。**

ribbon:

eureka:

enabled: false

## **16.8直接使用Ribbon API**

您也可以直接使用LoadBalancerClient，如以下示例所示：

**public** **class** MyClass {

@Autowired

**private** LoadBalancerClient loadBalancer;

**public** **void** doStuff() {

ServiceInstance instance = loadBalancer.choose("stores");

URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), instance.getPort()));

*// ... do something with the URI*

}

}

## **16.9 Ribbon配置的缓存**

每个Ribbon命名的客户端都有一个相应的子应用程序上下文，Spring Cloud维护该上下文。该应用程序上下文在对命名客户端的第一个请求上延迟加载。通过指定Ribbon客户端的名称，可以更改此延迟加载行为，以代替在启动时急于加载这些子应用程序上下文，如以下示例所示：

**application.yml。**

ribbon:

eager-load:

enabled: true

clients: client1, client2, client3

## **16.10如何配置Hystrix线程池**

如果将zuul.ribbonIsolationStrategy更改为THREAD，则Hystrix的线程隔离策略将用于所有路由。在这种情况下，HystrixThreadPoolKey默认设置为RibbonCommand。这意味着所有路由的HystrixCommands在相同的Hystrix线程池中执行。可以使用以下配置更改此行为：

**application.yml。**

zuul:

threadPool:

useSeparateThreadPools: true

前面的示例导致在每个路由的Hystrix线程池中执行HystrixCommands。

在这种情况下，默认HystrixThreadPoolKey与每个路由的服务ID相同。要将前缀添加到HystrixThreadPoolKey，请将zuul.threadPool.threadPoolKeyPrefix设置为要添加的值，如以下示例所示：

**application.yml。**

zuul:

threadPool:

useSeparateThreadPools: true

threadPoolKeyPrefix: zuulgw

## **16.11如何提供Ribbon的IRule的密钥**

如果您需要提供自己的IRule实现来处理诸如“ canary ”测试之类的特殊路由要求，请将一些信息传递给IRule的choose方法。

**com.netflix.loadbalancer.IRule.java。**

public interface IRule{

public Server choose(Object key);

:

您可以提供一些信息，供您的IRule实现用来选择目标服务器，如以下示例所示：

RequestContext.getCurrentContext()

.set(FilterConstants.LOAD\_BALANCER\_KEY, "canary-test");

如果您使用密钥FilterConstants.LOAD\_BALANCER\_KEY将任何对象放入RequestContext中，则该对象将传递到IRule实现的choose方法中。上例中显示的代码必须在执行RibbonRoutingFilter之前执行。Zuul的前置过滤器是执行此操作的最佳位置。您可以通过预过滤器中的RequestContext访问HTTP标头和查询参数，因此可以用来确定传递到Ribbon的LOAD\_BALANCER\_KEY。如果没有在RequestContext中用LOAD\_BALANCER\_KEY放置任何值，则将空值作为choose方法的参数传递。

## **17.外部配置：Archaius**

[Archaius](https://github.com/Netflix/archaius" \t "https://www.springcloud.cc/_top)是Netflix客户端配置库。它是所有Netflix OSS组件用于配置的库。Archaius是[Apache Commons Configuration](https://commons.apache.org/proper/commons-configuration" \t "https://www.springcloud.cc/_top)项目的扩展。它允许通过轮询源以进行更改或通过将源将更改推送到客户端来更新配置。Archaius使用Dynamic <Type> Property类作为属性的句柄，如以下示例所示：

**Archaius示例。**

**class** ArchaiusTest {

DynamicStringProperty myprop = DynamicPropertyFactory

.getInstance()

.getStringProperty("my.prop");

**void** doSomething() {

OtherClass.someMethod(myprop.get());

}

}

Archaius具有自己的一组配置文件和加载优先级。Spring应用程序通常不应该直接使用Archaius，但是仍然需要本地配置Netflix工具。Spring Cloud具有Spring环境桥，因此Archaius可以从Spring环境读取属性。该桥允许Spring Boot项目使用常规配置工具链，同时允许它们按记录的方式配置Netflix工具（大部分情况下）。

## **18.路由器和过滤器：Zuul**

路由是微服务架构不可或缺的一部分。例如，/可能被映射到您的web应用程序，/api/users被映射到用户服务，/api/shop被映射到商店服务。 [Zuul](https://github.com/Netflix/zuul" \t "https://www.springcloud.cc/_top)是Netflix的基于JVM的路由器和服务器端负载平衡器。

[Netflix将Zuul](https://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27" \t "https://www.springcloud.cc/_top)用于以下[用途](https://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27" \t "https://www.springcloud.cc/_top)：

* 认证方式
* 见解
* 压力测试
* 金丝雀测试
* 动态路由
* 服务迁移
* 减载
* 安全
* 静态响应处理
* 主动/主动流量管理

Zuul的规则引擎可使用几乎所有JVM语言编写规则和过滤器，并内置对Java和Groovy的支持。

|  |
| --- |
| IMG_336 |
| 配置属性zuul.max.host.connections已被两个新属性zuul.host.maxTotalConnections和zuul.host.maxPerRouteConnections取代，它们分别默认为200和20。 |

|  |
| --- |
| IMG_337 |
| 所有路由的默认Hystrix隔离模式（ExecutionIsolationStrategy）为SEMAPHORE。如果首选该隔离模式，则可以将zuul.ribbonIsolationStrategy更改为THREAD。 |

## **18.1如何包含Zuul**

要将Zuul包含在您的项目中，请使用组ID为org.springframework.cloud和工件ID为​​spring-cloud-starter-netflix-zuul的启动程序。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

## **18.2嵌入式Zuul反向代理**

Spring Cloud已创建嵌入式Zuul代理，以简化UI应用程序要对一个或多个后端服务进行代理调用的常见用例的开发。此功能对于用户界面代理所需的后端服务很有用，从而避免了为所有后端独立管理CORS和身份验证问题的需求。

要启用它，请用@EnableZuulProxy注释Spring Boot主类。这样做会导致将本地呼叫转发到适当的服务。按照约定，ID为users的服务从位于/users的代理接收请求（前缀被去除）。代理使用Ribbon来定位要通过发现转发到的实例。所有请求均在[hystrix命令](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "hystrix-fallbacks-for-routes" \o "18.13提供路线的Hystrix后备)中执行，因此失败以Hystrix指标显示。一旦电路断开，代理就不会尝试与服务联系。

|  |
| --- |
| IMG_338 |
| Zuul入门程序不包含发现客户端，因此，对于基于服务ID的路由，您还需要在类路径上提供其中之一（Eureka是一种选择）。 |

要跳过自动添加服务的过程，请将zuul.ignored-services设置为服务ID模式的列表。如果服务与被忽略但已包含在显式配置的路由映射中的模式匹配，则将其忽略，如以下示例所示：

**application.yml。**

zuul: ignoredServices: '\*' routes: users: /myusers/\*\*

在前面的示例中，**除** users **外**，所有服务均被忽略。

要增加或更改代理路由，可以添加外部配置，如下所示：

**application.yml。**

zuul: routes: users: /myusers/\*\*

前面的示例意味着对/myusers的HTTP调用被转发到users服务（例如，/myusers/101被转发到/101）。

要对路由进行更细粒度的控制，可以分别指定路径和serviceId，如下所示：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* serviceId: users\_service

前面的示例意味着对/myusers的HTTP调用将转发到users\_service服务。路由必须具有可以指定为蚂蚁样式模式的path，因此/myusers/\*仅匹配一个级别，而/myusers/\*\*则分层匹配。

后端的位置可以指定为serviceId（用于发现服务）或url（用于物理位置），如以下示例所示：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* url: https://example.com/users\_service

这些简单的url路由不会作为HystrixCommand来执行，也不会使用Ribbon对多个URL进行负载均衡。为了实现这些目标，可以使用静态服务器列表指定一个serviceId，如下所示：

**application.yml。**

zuul: routes: echo: path: /myusers/\*\* serviceId: myusers-service stripPrefix: **true**

hystrix: command: myusers-service: execution: isolation: thread: timeoutInMilliseconds: ...

myusers-service: ribbon: NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList listOfServers: https://example1.com,http://example2.com ConnectTimeout: 1000 ReadTimeout: 3000 MaxTotalHttpConnections: 500 MaxConnectionsPerHost: 100

另一种方法是指定服务路由并为serviceId配置Ribbon客户端（这样做需要在Ribbon中禁用Eureka支持- [有关更多信息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon-without-eureka" \o "16.6示例：如何在没有尤里卡的情况下使用色带)，请参见[上文](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon-without-eureka" \o "16.6 Example: How to Use Ribbon Without Eureka)），如下所示例：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* serviceId: users

ribbon: eureka: enabled: **false**

users: ribbon: listOfServers: example.com,google.com

您可以使用regexmapper在serviceId和路由之间提供约定。它使用正则表达式命名组从serviceId中提取变量，并将其注入到路由模式中，如以下示例所示：

**ApplicationConfiguration.java。**

@Bean**public** PatternServiceRouteMapper serviceRouteMapper() {

**return** **new** PatternServiceRouteMapper(

"(?<name>^.+)-(?<version>v.+$)",

"${version}/${name}");

}

前面的示例意味着myusers-v1中的serviceId被映射到路由/v1/myusers/\*\*。可以接受任何正则表达式，但是所有命名组必须同时存在于servicePattern和routePattern中。如果servicePattern与serviceId不匹配，则使用默认行为。在前面的示例中，myusers中的serviceId被映射到“ / myusers / \*\*”路由（未检测到版本）。默认情况下，此功能是禁用的，仅适用于发现的服务。

要为所有映射添加前缀，请将zuul.prefix设置为一个值，例如/api。默认情况下，在转发请求之前，将从请求中删除代理前缀（您可以使用zuul.stripPrefix=false将此行为关闭）。您还可以关闭从单个路由中剥离特定于服务的前缀，如以下示例所示：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* stripPrefix: **false**

|  |
| --- |
| IMG_339 |
| zuul.stripPrefix仅适用于zuul.prefix中设置的前缀。它对给定路由的path中定义的前缀没有任何影响。 |

在前面的示例中，对/myusers/101的请求被转发到users服务上的/myusers/101。

zuul.routes条目实际上绑定到类型为ZuulProperties的对象。如果查看该对象的属性，则可以看到它也有一个retryable标志。将该标志设置为true，以使Ribbon客户端自动重试失败的请求。当您需要修改使用Ribbon客户端配置的重试操作的参数时，也可以将该标志设置为true。

默认情况下，X-Forwarded-Host标头被添加到转发的请求中。要关闭它，请设置zuul.addProxyHeaders = false。默认情况下，前缀路径被剥离，并且后端请求使用X-Forwarded-Prefix标头（在前面显示的示例中为/myusers）。

如果设置默认路由（/），则带有@EnableZuulProxy的应用程序可以充当独立服务器。例如，zuul.route.home: /会将所有流量（“ / \*\*”）路由到“ home”服务。

如果需要更细粒度的忽略，则可以指定要忽略的特定模式。这些模式在路线定位过程开始时进行评估，这意味着模式中应包含前缀以保证匹配。被忽略的模式跨越所有服务，并取代任何其他路由规范。以下示例显示了如何创建忽略的模式：

**application.yml。**

zuul: ignoredPatterns: /\*\*/admin/\*\* routes: users: /myusers/\*\*

前面的示例意味着所有呼叫（例如/myusers/101）都被转发到users服务上的/101。但是，包括/admin/在内的呼叫无法解决。

|  |
| --- |
| IMG_340 |
| 如果您需要保留路由的顺序，则需要使用YAML文件，因为使用属性文件时顺序会丢失。以下示例显示了这样的YAML文件： |

**application.yml。**

zuul: routes: users: path: /myusers/\*\* legacy: path: /\*\*

如果要使用属性文件，则legacy路径可能最终位于users路径的前面，从而导致users路径不可访问。

## **18.3 Zuul HTTP客户端**

Zuul使用的默认HTTP客户端现在由Apache HTTP客户端支持，而不是已弃用的Ribbon RestClient。要使用RestClient或okhttp3.OkHttpClient，请分别设置ribbon.restclient.enabled=true或ribbon.okhttp.enabled=true。如果要自定义Apache HTTP客户端或OK HTTP客户端，请提供类型为ClosableHttpClient或OkHttpClient的bean。

## **18.4 Cookie和敏感标题**

您可以在同一系统中的服务之间共享标头，但您可能不希望敏感标头泄漏到下游到外部服务器中。您可以在路由配置中指定忽略的标头列表。Cookies发挥着特殊的作用，因为它们在浏览器中具有定义明确的语义，并且始终将它们视为敏感内容。如果代理的使用者是浏览器，那么下游服务的cookie也会给用户带来麻烦，因为它们都混杂在一起（所有下游服务看起来都来自同一位置）。

如果您对服务的设计很谨慎（例如，如果只有一个下游服务设置cookie），则可以让它们从后端一直流到调用者。另外，如果您的代理设置了cookie，并且您的所有后端服务都在同一系统中，则很自然地简单地共享它们（例如，使用Spring Session将它们链接到某些共享状态）。除此之外，由下游服务设置的任何cookie可能对调用者都无用，因此建议您将（至少）Set-Cookie和Cookie设置为敏感的标头，用于那些没有使用的路由您网域的一部分。即使对于属于您网域的路由，在让Cookie在它们和代理之间流动之前，也应仔细考虑其含义。

可以将敏感头配置为每个路由的逗号分隔列表，如以下示例所示：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* sensitiveHeaders: Cookie,Set-Cookie,Authorization url: https://downstream

|  |
| --- |
| IMG_341 |
| 这是sensitiveHeaders的默认值，因此除非您希望其与众不同，否则无需进行设置。这是Spring Cloud Netflix 1.1中的新功能（在1.0中，用户无法控制标题，并且所有cookie都双向流动）。 |

sensitiveHeaders是黑名单，默认值不为空。因此，要使Zuul发送所有标头（ignored除外），必须将其显式设置为空列表。如果要将Cookie或授权标头传递到后端，则必须这样做。以下示例显示如何使用sensitiveHeaders：

**application.yml。**

zuul: routes: users: path: /myusers/\*\* sensitiveHeaders: url: https://downstream

您还可以通过设置zuul.sensitiveHeaders来设置敏感标题。如果在路由上设置了sensitiveHeaders，它将覆盖全局sensitiveHeaders设置。

## **18.5忽略标题**

除了路由敏感的标头之外，您还可以为与下游服务交互期间应丢弃的值（请求和响应）设置一个名为zuul.ignoredHeaders的全局值。默认情况下，如果Spring Security不在类路径中，则它们为空。否则，它们将初始化为Spring Security指定的一组众所周知的“ 安全性 ”标头（例如，涉及缓存）。在这种情况下的假设是，下游服务也可以添加这些标头，但是我们需要来自代理的值。要在类路径上有Spring Security时不丢弃这些众所周知的安全标头，可以将zuul.ignoreSecurityHeaders设置为false。如果您在Spring Security中禁用了HTTP安全响应标头，并希望由下游服务提供值，则这样做很有用。

## **18.6管理端点**

默认情况下，如果将@EnableZuulProxy与Spring Boot Actuator结合使用，则将启用两个附加端点：

* 路线
* 筛选器

### **18.6.1路由端点**

在/routes处的路由端点的GET返回已映射路由的列表：

**GET /路线。**

**{**

/stores/\*\*: "http://localhost:8081"**}**

可以通过将?format=details查询字符串添加到/routes来请求其他路由详细信息。这样做会产生以下输出：

**获取/ routes / details。**

**{**

"/stores/\*\*": **{**

"id": "stores"**,**

"fullPath": "/stores/\*\*"**,**

"location": "http://localhost:8081"**,**

"path": "/\*\*"**,**

"prefix": "/stores"**,**

"retryable": **false,**

"customSensitiveHeaders": **false,**

"prefixStripped": **true**

**}}**

POST至/routes强制刷新现有路由（例如，当服务目录中发生更改时）。您可以通过将endpoints.routes.enabled设置为false来禁用此端点。

|  |
| --- |
| IMG_342 |
| 路由应该自动响应服务目录中的更改，但是从POST到/routes是强制更改立即进行的一种方法。 |

### **18.6.2过滤器端点**

过滤器端点/filters的GET按类型返回Zuul过滤器的映射。对于地图中的每种过滤器类型，您将获得该类型的所有过滤器的列表以及它们的详细信息。

## **18.7扼杀模式和局部Forwards**

迁移现有应用程序或API时，常见的模式是“ 勒死 ”旧的端点，并用不同的实现方式慢慢替换它们。Zuul代理是一个有用的工具，因为您可以使用它来处理来自旧端点的客户端的所有流量，但可以将一些请求重定向到新请求。

以下示例显示“ 扼杀 ”方案的配置详细信息：

**application.yml。**

zuul: routes: first: path: /first/\*\* url: https://first.example.com second: path: /second/\*\* url: forward:/second third: path: /third/\*\* url: forward:/3rd legacy: path: /\*\* url: https://legacy.example.com

在前面的示例中，我们扼杀了“ legacy ”应用程序，该应用程序映射到与其他模式之一不匹配的所有请求。/first/\*\*中的路径已使用外部URL提取到新服务中。/second/\*\*中的路径被转发，以便可以在本地处理（例如，使用普通Spring @RequestMapping）。/third/\*\*中的路径也被转发，但是前缀不同（/third/foo被转发到/3rd/foo）。

|  |
| --- |
| IMG_343 |
| 被忽略的模式不会被完全忽略，它们不会由代理处理（因此它们也可以在本地有效转发）。 |

## **18.8通过Zuul上传文件**

如果使用@EnableZuulProxy，则可以使用代理路径上载文件，只要文件很小，它就可以正常工作。对于大文件，有一个替代路径可以绕过“ / zuul / \*”中的Spring DispatcherServlet（以避免进行多部分处理）。换句话说，如果您拥有zuul.routes.customers=/customers/\*\*，则可以将POST大文件复制到/zuul/customers/\*。Servlet路径通过zuul.servletPath外部化。如果代理路由将您带到Ribbon负载均衡器，则超大文件也需要提高超时设置，如以下示例所示：

**application.yml。**

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000ribbon: ConnectTimeout: 3000 ReadTimeout: 60000

请注意，要使流技术处理大文件，您需要在请求中使用分块编码（某些浏览器默认不这样做），如以下示例所示：

$ curl -v -H "Transfer-Encoding: chunked" \

-F "file=@mylarge.iso" localhost:9999/zuul/simple/file

## **18.9查询字符串编码**

在处理传入请求时，查询参数将被解码，以便可以在Zuul过滤器中进行修改。然后将它们重新编码，在路由过滤器中重建后端请求。例如，如果结果是使用Javascript的encodeURIComponent()方法编码的，则结果可能不同于原始输入。虽然这在大多数情况下不会引起问题，但某些web服务器可能对复杂查询字符串的编码很挑剔。

要强制对查询字符串进行原始编码，可以将特殊标志传递给ZuulProperties，以便使用HttpServletRequest::getQueryString方法按原样使用查询字符串，如以下示例所示：

**application.yml。**

zuul: forceOriginalQueryStringEncoding: **true**

|  |
| --- |
| IMG_344 |
| 该特殊标志仅适用于SimpleHostRoutingFilter。另外，您松开了使用RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)轻松覆盖查询参数的功能，因为现在直接在原始HttpServletRequest上获取了查询字符串。 |

## **18.10请求URI编码**

在处理传入请求时，在将请求URI与路由匹配之前，先对其进行解码。然后在路由过滤器中重建后端请求时，将对请求URI进行重新编码。如果您的URI包含编码的“ /”字符，则可能导致某些意外行为。

要使用原始请求URI，可以将特殊标志传递给'ZuulProperties'，以便使用HttpServletRequest::getRequestURI方法按原样使用URI，如以下示例所示：

**application.yml。**

zuul: decodeUrl: **false**

|  |
| --- |
| IMG_345 |
| 如果使用requestURI RequestContext属性覆盖请求URI，并且此标志设置为false，则将不对在请求上下文中设置的URL进行编码。确保URL已被编码是您的责任。 |

## **18.11普通嵌入式Zuul**

如果使用@EnableZuulServer（而不是@EnableZuulProxy），则也可以运行Zuul服务器而不进行代理或有选择地打开代理平台的某些部分。您添加到类型为ZuulFilter的应用程序中的所有beans都会自动安装（与@EnableZuulProxy一样），但是不会自动添加任何代理过滤器。

在这种情况下，仍然可以通过配置“ zuul.routes。\*”来指定进入Zuul服务器的路由，但是没有服务发现也没有代理。因此，“ serviceId”和“ url”设置将被忽略。以下示例将“ / api / \*\*”中的所有路径映射到Zuul过滤器链：

**application.yml。**

zuul: routes: api: /api/\*\*

## **18.12禁用Zuul过滤器**

Spring Cloud的Zuul带有多个ZuulFilter beans，默认情况下在代理和服务器模式下都启用。有关可以启用的过滤器列表，请参见[Zuul过滤器包](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-zuul/src/main/java/org/springframework/cloud/netflix/zuul/filters" \t "https://www.springcloud.cc/_top)。如果要禁用一个，请设置zuul.<SimpleClassName>.<filterType>.disable=true。按照惯例，filters之后的软件包是Zuul过滤器类型。例如，要禁用org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter，请设置zuul.SendResponseFilter.post.disable=true。

## **18.13为路线提供Hystrix后备**

当Zuul中给定路由的电路跳闸时，可以通过创建类型为FallbackProvider的bean提供回退响应。在此bean中，您需要指定回退的路由ID，并提供一个ClientHttpResponse作为回退的路由。以下示例显示了一个相对简单的FallbackProvider实现：

**class** MyFallbackProvider **implements** FallbackProvider {

@Override

**public** String getRoute() {

**return** "customers";

}

@Override

**public** ClientHttpResponse fallbackResponse(String route, **final** Throwable cause) {

**if** (cause **instanceof** HystrixTimeoutException) {

**return** response(HttpStatus.GATEWAY\_TIMEOUT);

} **else** {

**return** response(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**private** ClientHttpResponse response(**final** HttpStatus status) {

**return** **new** ClientHttpResponse() {

@Override

**public** HttpStatus getStatusCode() **throws** IOException {

**return** status;

}

@Override

**public** **int** getRawStatusCode() **throws** IOException {

**return** status.value();

}

@Override

**public** String getStatusText() **throws** IOException {

**return** status.getReasonPhrase();

}

@Override

**public** **void** close() {

}

@Override

**public** InputStream getBody() **throws** IOException {

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

@Override

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

以下示例显示了上一个示例的路由配置可能如何显示：

zuul: routes: customers: /customers/\*\*

如果您想为所有路由提供默认后备，则可以创建类型为FallbackProvider的bean，并让getRoute方法返回\*或null，如以下示例：

**class** MyFallbackProvider **implements** FallbackProvider {

@Override

**public** String getRoute() {

**return** "\*";

}

@Override

**public** ClientHttpResponse fallbackResponse(String route, Throwable throwable) {

**return** **new** ClientHttpResponse() {

@Override

**public** HttpStatus getStatusCode() **throws** IOException {

**return** HttpStatus.OK;

}

@Override

**public** **int** getRawStatusCode() **throws** IOException {

**return** 200;

}

@Override

**public** String getStatusText() **throws** IOException {

**return** "OK";

}

@Override

**public** **void** close() {

}

@Override

**public** InputStream getBody() **throws** IOException {

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

@Override

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

## **18.14 Zuul超时**

如果要为通过Zuul代理的请求配置套接字超时和读取超时，则根据您的配置，有两个选项：

* 如果Zuul使用服务发现，则需要使用ribbon.ReadTimeout和ribbon.SocketTimeout Ribbon属性配置这些超时。

如果通过指定URL配置了Zuul路由，则需要使用zuul.host.connect-timeout-millis和zuul.host.socket-timeout-millis。

## **18.15重写Location标头**

如果Zuul在web应用程序的前面，则当web应用程序通过HTTP状态代码3XX重定向时，您可能需要重新编写Location标头。否则，浏览器将重定向到web应用程序的URL，而不是Zuul URL。您可以配置LocationRewriteFilter Zuul过滤器，将Location标头重写为Zuul的URL。它还添加回去的全局前缀和特定于路由的前缀。以下示例通过使用Spring配置文件添加过滤器：

**import** org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;

...

@Configuration@EnableZuulProxy**public** **class** ZuulConfig {

@Bean

**public** LocationRewriteFilter locationRewriteFilter() {

**return** **new** LocationRewriteFilter();

}

}

|  |  |
| --- | --- |
| IMG_346 | **警告** |
| 小心使用此过滤器。筛选器作用于所有3XX响应代码的Location头，这可能并不适用于所有情况，例如将用户重定向到外部URL时。 |

## **18.16启用跨源请求**

默认情况下，Zuul将所有跨源请求（CORS）路由到服务。如果您希望Zuul处理这些请求，可以通过提供自定义WebMvcConfigurer bean来完成：

@Bean**public** WebMvcConfigurer corsConfigurer() {

**return** **new** WebMvcConfigurer() {

**public** **void** addCorsMappings(CorsRegistry registry) {

registry.addMapping("/path-1/\*\*")

.allowedOrigins("https://allowed-origin.com")

.allowedMethods("GET", "POST");

}

};

}

在上面的示例中，我们允许[https://allowed-origin.com](https://allowed-origin.com/" \t "https://www.springcloud.cc/_top)中的GET和POST方法将跨域请求发送到以path-1开头的端点。您可以使用/\*\*映射将CORS配置应用于特定的路径模式或整个应用程序的全局路径。您可以通过此配置来自定义属性：allowedOrigins，allowedMethods，allowedHeaders，exposedHeaders，allowCredentials和maxAge。

## **18.17指标**

Zuul将在执行器指标终结点下提供指标，以解决路由请求时可能发生的任何故障。可以通过点击/actuator/metrics来查看这些指标。指标的名称格式为ZUUL::EXCEPTION:errorCause:statusCode。

## **18.18 Zuul开发人员指南**

有关Zuul的工作原理的一般概述，请参见[Zuul Wiki](https://github.com/Netflix/zuul/wiki/How-it-Works" \t "https://www.springcloud.cc/_top)。

### **18.18.1 Zuul Servlet**

Zuul被实现为Servlet。对于一般情况，Zuul已嵌入Spring调度机制中。这使Spring MVC可以控制路由。在这种情况下，Zuul缓冲请求。如果需要通过Zuul而不缓冲请求（例如，用于大文件上传），则Servlet也将安装在Spring Dispatcher之外。缺省情况下，该servlet的地址为/zuul。可以使用zuul.servlet-path属性更改此路径。

### **18.18.2 Zuul RequestContext**

要在过滤器之间传递信息，Zuul使用[RequestContext](https://github.com/Netflix/zuul/blob/1.x/zuul-core/src/main/java/com/netflix/zuul/context/RequestContext.java" \t "https://www.springcloud.cc/_top)。其数据保存在每个请求专用的ThreadLocal中。有关在何处路由请求，错误以及实际的HttpServletRequest和HttpServletResponse的信息存储在此处。RequestContext扩展了ConcurrentHashMap，因此任何内容都可以存储在上下文中。[FilterConstants](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-zuul/src/main/java/org/springframework/cloud/netflix/zuul/filters/support/FilterConstants.java" \t "https://www.springcloud.cc/_top)包含Spring Cloud Netflix安装的过滤器使用的密钥（[稍后会](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-enable-filters" \o "18.18.4 @EnableZuulServer筛选器)详细介绍）。

### **18.18.3 @EnableZuulProxy与@EnableZuulServer**

Spring Cloud Netflix安装了许多过滤器，具体取决于启用了Zuul的注释。@EnableZuulProxy是@EnableZuulServer的超集。换句话说，@EnableZuulProxy包含@EnableZuulServer安装的所有筛选器。“ 代理 ”中的其他过滤器启用路由功能。如果您想使用“ 空白 ” Zuul，则应使用@EnableZuulServer。

### **18.18.4 @EnableZuulServer过滤器**

@EnableZuulServer创建一个SimpleRouteLocator，该文件从Spring Boot配置文件中加载路由定义。

已安装以下过滤器（按常规方式Spring Beans）：

前置过滤器：

* + ServletDetectionFilter：检测请求是否通过Spring分派器进行。设置键为FilterConstants.IS\_DISPATCHER\_SERVLET\_REQUEST\_KEY的布尔值。
  + FormBodyWrapperFilter：解析表单数据并为下游请求重新编码。
  + DebugFilter：如果设置了debug请求参数，则将RequestContext.setDebugRouting()和RequestContext.setDebugRequest()设置为true。\*路由过滤器：
  + SendForwardFilter：使用Servlet RequestDispatcher的Forwards请求。转发位置存储在RequestContext属性FilterConstants.FORWARD\_TO\_KEY中。这对于转发到当前应用程序中的端点很有用。

帖子过滤器：

* + SendResponseFilter：将代理请求的响应写入当前响应。

错误过滤器：

* + SendErrorFilter：如果RequestContext.getThrowable()不为空，则Forwards至/error（默认）。您可以通过设置error.path属性来更改默认转发路径（/error）。

### **18.18.5 @EnableZuulProxy过滤器**

创建一个DiscoveryClientRouteLocator，它从DiscoveryClient（例如Eureka）以及属性中加载路由定义。从DiscoveryClient为每个serviceId创建一条路由。添加新服务后，将刷新路由。

除了前面描述的过滤器之外，还安装了以下过滤器（常规Spring Beans）：

前置过滤器：

* + PreDecorationFilter：根据提供的RouteLocator确定路线和路线。它还为下游请求设置了各种与代理相关的标头。

路线过滤器：

RibbonRoutingFilter：使用Ribbon，Hystrix和可插拔的HTTP客户端发送请求。在RequestContext属性FilterConstants.SERVICE\_ID\_KEY中可以找到服务ID。此过滤器可以使用不同的HTTP客户端：

* + - Apache HttpClient：默认客户端。
    - Squareup OkHttpClient v3：通过在类路径上放置com.squareup.okhttp3:okhttp库并设置ribbon.okhttp.enabled=true来启用。
    - Netflix Ribbon HTTP客户端：通过设置ribbon.restclient.enabled=true启用。该客户端具有局限性，包括不支持PATCH方法，但是还具有内置的重试功能。
  + SimpleHostRoutingFilter：通过Apache HttpClient将请求发送到预定的URL。可在RequestContext.getRouteHost()中找到URL。

### **18.18.6自定义Zuul过滤器示例**

下面的大多数“如何编写”示例都包含在[示例Zuul过滤器](https://github.com/spring-cloud-samples/sample-zuul-filters" \t "https://www.springcloud.cc/_top)项目中。在该存储库中也有一些处理请求或响应正文的示例。

本节包括以下示例：

* [名为“如何编写预过滤器”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-pre-filter" \o "如何编写前置过滤器)
* [名为“如何编写路由过滤器”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-route-filter" \o "如何编写路由过滤器)
* [名为“如何编写后置过滤器”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "zuul-developer-guide-sample-post-filter" \o "如何编写帖子过滤器)

#### **如何编写前置过滤器**

前置过滤器可在RequestContext中设置数据，以便在下游的过滤器中使用。主要用例是设置路由过滤器所需的信息。以下示例显示了Zuul前置过滤器：

**public** **class** QueryParamPreFilter **extends** ZuulFilter {

@Override

**public** **int** filterOrder() {

**return** PRE\_DECORATION\_FILTER\_ORDER - 1; *// run before PreDecoration*

}

@Override

**public** String filterType() {

**return** PRE\_TYPE;

}

@Override

**public** **boolean** shouldFilter() {

RequestContext ctx = RequestContext.getCurrentContext();

**return** !ctx.containsKey(FORWARD\_TO\_KEY) *// a filter has already forwarded*

&& !ctx.containsKey(SERVICE\_ID\_KEY); *// a filter has already determined serviceId*

}

@Override

**public** Object run() {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest request = ctx.getRequest();

**if** (request.getParameter("sample") != null) {

*// put the serviceId in `RequestContext`*

ctx.put(SERVICE\_ID\_KEY, request.getParameter("foo"));

}

**return** null;

}

}

前面的过滤器从sample请求参数中填充SERVICE\_ID\_KEY。实际上，您不应该执行这种直接映射。而是应从sample的值中查找服务ID。

现在已填充SERVICE\_ID\_KEY，PreDecorationFilter将不运行，而RibbonRoutingFilter将运行。

|  |
| --- |
| IMG_347 |
| 如果要路由到完整URL，请致电ctx.setRouteHost(url)。 |

要修改路由过滤器转发到的路径，请设置REQUEST\_URI\_KEY。

#### **如何编写路由过滤器**

路由过滤器在预过滤器之后运行，并向其他服务发出请求。这里的许多工作是将请求和响应数据与客户端所需的模型相互转换。以下示例显示了Zuul路由过滤器：

**public** **class** OkHttpRoutingFilter **extends** ZuulFilter {

@Autowired

**private** ProxyRequestHelper helper;

@Override

**public** String filterType() {

**return** ROUTE\_TYPE;

}

@Override

**public** **int** filterOrder() {

**return** SIMPLE\_HOST\_ROUTING\_FILTER\_ORDER - 1;

}

@Override

**public** **boolean** shouldFilter() {

**return** RequestContext.getCurrentContext().getRouteHost() != null

&& RequestContext.getCurrentContext().sendZuulResponse();

}

@Override

**public** Object run() {

OkHttpClient httpClient = **new** OkHttpClient.Builder()

*// customize*

.build();

RequestContext context = RequestContext.getCurrentContext();

HttpServletRequest request = context.getRequest();

String method = request.getMethod();

String uri = **this**.helper.buildZuulRequestURI(request);

Headers.Builder headers = **new** Headers.Builder();

Enumeration<String> headerNames = request.getHeaderNames();

**while** (headerNames.hasMoreElements()) {

String name = headerNames.nextElement();

Enumeration<String> values = request.getHeaders(name);

**while** (values.hasMoreElements()) {

String value = values.nextElement();

headers.add(name, value);

}

}

InputStream inputStream = request.getInputStream();

RequestBody requestBody = null;

**if** (inputStream != null && HttpMethod.permitsRequestBody(method)) {

MediaType mediaType = null;

**if** (headers.get("Content-Type") != null) {

mediaType = MediaType.parse(headers.get("Content-Type"));

}

requestBody = RequestBody.create(mediaType, StreamUtils.copyToByteArray(inputStream));

}

Request.Builder builder = **new** Request.Builder()

.headers(headers.build())

.url(uri)

.method(method, requestBody);

Response response = httpClient.newCall(builder.build()).execute();

LinkedMultiValueMap<String, String> responseHeaders = **new** LinkedMultiValueMap<>();

**for** (Map.Entry<String, List<String>> entry : response.headers().toMultimap().entrySet()) {

responseHeaders.put(entry.getKey(), entry.getValue());

}

**this**.helper.setResponse(response.code(), response.body().byteStream(),

responseHeaders);

context.setRouteHost(null); *// prevent SimpleHostRoutingFilter from running*

**return** null;

}

}

前面的过滤器将Servlet请求信息转换为OkHttp3请求信息，执行HTTP请求，并将OkHttp3响应信息转换为Servlet响应。

#### **如何编写帖子过滤器**

后置过滤器通常操纵响应。以下过滤器将随机UUID添加为X-Sample标头：

**public** **class** AddResponseHeaderFilter **extends** ZuulFilter {

@Override

**public** String filterType() {

**return** POST\_TYPE;

}

@Override

**public** **int** filterOrder() {

**return** SEND\_RESPONSE\_FILTER\_ORDER - 1;

}

@Override

**public** **boolean** shouldFilter() {

**return** true;

}

@Override

**public** Object run() {

RequestContext context = RequestContext.getCurrentContext();

HttpServletResponse servletResponse = context.getResponse();

servletResponse.addHeader("X-Sample", UUID.randomUUID().toString());

**return** null;

}

}

|  |
| --- |
| IMG_348 |
| 其他操作，例如转换响应主体，则更加复杂且计算量大。 |

### **18.18.7 Zuul错误的工作方式**

如果在Zuul过滤器生命周期的任何部分抛出异常，则将执行错误过滤器。仅当RequestContext.getThrowable()不是null时才运行SendErrorFilter。然后，它在请求中设置特定的javax.servlet.error.\*属性，并将请求转发到Spring Boot错误页面。

### **18.18.8 Zuul急切的应用程序上下文加载**

Zuul内部使用Ribbon来调用远程URL。默认情况下，Ribbon客户端在第一次调用时被Spring Cloud延迟加载。可以使用以下配置为Zuul更改此行为，这会导致在应用程序启动时急于加载与子Ribbon相关的应用程序上下文。以下示例显示了如何启用即时加载：

**application.yml。**

zuul:

ribbon:

eager-load:

enabled: true

## **19. Polyglot支持Sidecar**

您是否要使用非JVM语言来利用Eureka，Ribbon和Config Server？Spring Cloud Netflix Sidecar的灵感来自[Netflix Prana](https://github.com/Netflix/Prana" \t "https://www.springcloud.cc/_top)。它包括一个HTTP API，用于获取给定服务的所有实例（按主机和端口）。您也可以通过嵌入式Zuul代理来代理服务调用，该代理从Eureka获取其路由条目。可以直接通过主机查找或通过Zuul代理访问Spring Cloud Config服务器。非JVM应用程序应实施运行状况检查，以便Sidecar可以向Eureka报告应用程序是启动还是关闭。

要在项目中包含Sidecar，请使用组ID为org.springframework.cloud且工件ID为​​spring-cloud-netflix-sidecar的依赖项。

要启用Sidecar，请使用@EnableSidecar创建一个Spring Boot应用程序。该注释包括@EnableCircuitBreaker，@EnableDiscoveryClient和@EnableZuulProxy。在与非JVM应用程序相同的主机上运行结果应用程序。

要配置侧车，请将sidecar.port和sidecar.health-uri添加到application.yml。sidecar.port属性是非JVM应用程序侦听的端口。这样Sidecar可以正确地向Eureka注册应用程序。sidecar.secure-port-enabled选项提供了一种启用流量安全端口的方法。sidecar.health-uri是在非JVM应用程序上可访问的URI，它模仿Spring Boot运行状况指示器。它应该返回类似于以下内容的JSON文档：

**health-uri-document。**

**{**

"status":"UP"**}**

以下application.yml示例显示了Sidecar应用程序的示例配置：

**application.yml。**

server: port: 5678spring: application: name: sidecar

sidecar: port: 8000 health-uri: http://localhost:8000/health.json

DiscoveryClient.getInstances()方法的API为/hosts/{serviceId}。以下针对/hosts/customers的示例响应在不同的主机上返回两个实例：

**/ hosts / customers。**

**[**

**{**

"host": "myhost"**,**

"port": 9000**,**

"uri": "http://myhost:9000"**,**

"serviceId": "CUSTOMERS"**,**

"secure": **false**

**},**

**{**

"host": "myhost2"**,**

"port": 9000**,**

"uri": "http://myhost2:9000"**,**

"serviceId": "CUSTOMERS"**,**

"secure": **false**

**}]**

非JVM应用程序（如果Sidecar位于端口5678上）可通过[http://localhost:5678/hosts/{serviceId}](http://localhost:5678/hosts/{serviceId}" \t "https://www.springcloud.cc/_top)访问此API。

Zuul代理会自动将Eureka中已知的每个服务的路由添加到/<serviceId>，因此可以在/customers中使用客户服务。非JVM应用程序可以在[http://localhost:5678/customers](http://localhost:5678/customers" \t "https://www.springcloud.cc/_top)上访问客户服务（假设Sidecar正在侦听5678端口）。

如果Config Server已向Eureka注册，则非JVM应用程序可以通过Zuul代理对其进行访问。如果ConfigServer的serviceId为configserver并且Sidecar在端口5678上，则可以在[http：// localhost：5678 / configserver上](http://localhost:5678/configserver" \t "https://www.springcloud.cc/_top)对其进行访问。

非JVM应用程序可以利用Config Server返回YAML文档的功能。例如，调用[https://sidecar.local.spring.io:5678/configserver/default-master.yml](https://sidecar.local.spring.io:5678/configserver/default-master.yml" \t "https://www.springcloud.cc/_top) 可能会导致YAML文档类似于以下内容：

eureka: client: serviceUrl: defaultZone: http://localhost:8761/eureka/ password: passwordinfo: description: Spring Cloud Samples url: https://github.com/spring-cloud-samples

要在使用HTTP时使运行状况检查请求接受所有证书，请将sidecar.accept-all-ssl-certificates设置为`true。

## **20.重试失败的请求**

Spring Cloud Netflix提供了多种发出HTTP请求的方式。您可以使用负载均衡的RestTemplate，Ribbon或Feign。无论您如何选择创建HTTP请求，始终都有一个请求失败的机会。当请求失败时，您可能希望自动重试该请求。为此，在使用Sping Cloud Netflix时，您需要在应用程序的类路径中包含[Spring重试](https://github.com/spring-projects/spring-retry" \t "https://www.springcloud.cc/_top)。如果存在Spring重试，则负载平衡的RestTemplates，Feign和Zuul会自动重试任何失败的请求（假设您的配置允许这样做）。

## **20.1退避政策**

默认情况下，重试请求时不使用任何退避策略。如果要配置退避策略，则需要创建类型为LoadBalancedRetryFactory的bean并为给定服务覆盖createBackOffPolicy方法，如以下示例所示：

@Configuration**public** **class** MyConfiguration {

@Bean

LoadBalancedRetryFactory retryFactory() {

**return** **new** LoadBalancedRetryFactory() {

@Override

**public** BackOffPolicy createBackOffPolicy(String service) {

**return** **new** ExponentialBackOffPolicy();

}

};

}

}

## **20.2配置**

将Ribbon与Spring重试一起使用时，可以通过配置某些Ribbon属性来控制重试功能。为此，请设置client.ribbon.MaxAutoRetries，client.ribbon.MaxAutoRetriesNextServer和client.ribbon.OkToRetryOnAllOperations属性。有关这些属性的作用的说明，请参见[Ribbon文档](https://github.com/Netflix/ribbon/wiki/Getting-Started" \l "the-properties-file-sample-clientproperties" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_349 |
| 启用client.ribbon.OkToRetryOnAllOperations包括重试POST请求，由于请求正文的缓冲，这可能会对服务器资源产生影响。 |

此外，当响应中返回某些状态代码时，您可能想重试请求。您可以通过设置clientName.ribbon.retryableStatusCodes属性来列出希望Ribbon客户端重试的响应代码，如以下示例所示：

clientName: ribbon: retryableStatusCodes: 404,502

您也可以创建类型为LoadBalancedRetryPolicy的bean，并实现retryableStatusCode方法以根据状态码重试请求。

### **20.2.1 Zuul**

您可以通过将zuul.retryable设置为false来关闭Zuul的重试功能。您还可以通过将zuul.routes.routename.retryable设置为false来逐个路由禁用重试功能。

## **21. HTTP客户端**

Spring Cloud Netflix会自动为您创建Ribbon，Feign和Zuul使用的HTTP客户端。但是，您也可以根据需要提供自定义的HTTP客户端。为此，如果使用的是Apache Http Cient，则可以创建类型为ClosableHttpClient的bean，如果使用的是OK HTTP，则可以创建类型为OkHttpClient的bean。

|  |
| --- |
| IMG_350 |
| 创建自己的HTTP客户端时，您还负责为这些客户端实施正确的连接管理策略。这样做不当会导致资源管理问题。 |

## **22.维护模式下的模块**

将模块置于维护模式意味着Spring Cloud团队将不再向模块添加新功能。我们将修复阻止程序错误和安全性问题，还将考虑并审查社区的一些小请求。

自Greenwich 发布列车全面上市以来，我们打算继续为这些模块提供至少一年的支持。

以下Spring Cloud Netflix模块和相应的启动器将进入维护模式：

* spring-cloud-netflix-archaius
* spring-cloud-netflix-hystrix-contract
* spring-cloud-netflix-hystrix-dashboard
* spring-cloud-netflix-hystrix-stream
* spring-cloud-netflix-hystrix
* spring-cloud-netflix-ribbon
* spring-cloud-netflix-turbine-stream
* spring-cloud-netflix-turbine
* spring-cloud-netflix-zuul

|  |
| --- |
| IMG_351 |
| 这不包括Eureka或并发限制模块。 |

# **第四部分 Spring Cloud OpenFeign**

**Greenwich SR5**

该项目通过自动配置并绑定到Spring环境和其他Spring编程模型习惯用法，为Spring Boot应用提供了OpenFeign集成。

## **23.声明式REST客户端：Feign**

[Feign](https://github.com/Netflix/feign" \t "https://www.springcloud.cc/_top)是声明性的web服务客户端。它使编写web服务客户端更加容易。要使用Feign，请创建一个接口并对其进行注释。它具有可插入的注释支持，包括Feign注释和JAX-RS注释。Feign还支持可插拔编码器和解码器。Spring Cloud添加了对Spring MVC注释的支持，并支持使用Spring Web中默认使用的同一HttpMessageConverters。Spring Cloud集成了Ribbon和Eureka以在使用Feign时提供负载平衡的http客户端。

## **23.1如何包括Feign**

要将Feign包含在您的项目中，请将启动器与组org.springframework.cloud和工件ID spring-cloud-starter-openfeign一起使用。有关 使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

示例spring boot应用

@SpringBootApplication@EnableFeignClients**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

**StoreClient.java。**

@FeignClient("stores")**public** **interface** StoreClient {

@RequestMapping(method = RequestMethod.GET, value = "/stores")

List<Store> getStores();

@RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")

Store update(@PathVariable("storeId") Long storeId, Store store);

}

在@FeignClient批注中，字符串值（上面的“ stores”）是一个任意的客户端名称，用于创建Ribbon负载均衡器（请参见[下面的Ribbon support的详细信息](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon" \o "16.客户端负载均衡器：功能区)）。您还可以使用url属性（绝对值或仅是主机名）来指定URL。在应用程序上下文中，bean的名称是接口的标准名称。要指定自己的别名值，可以使用@FeignClient批注的qualifier值。

上面的Ribbon客户端将希望发现“商店”服务的物理地址。如果您的应用程序是Eureka客户端，则它将在Eureka服务注册表中解析该服务。如果您不想使用Eureka，则可以简单地在外部配置中配置服务器列表（[例如，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon-without-eureka" \o "16.6示例：如何在没有尤里卡的情况下使用色带)参见 [上文](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-ribbon-without-eureka" \o "16.6 Example: How to Use Ribbon Without Eureka)）。

## **23.2覆盖Feign默认值**

Spring Cloud的Feign支持中的中心概念是指定客户的概念。每个虚拟客户端都是组件的一部分，这些组件可以一起工作以按需联系远程服务器，并且该组件的名称是您使用@FeignClient批注将其指定为应用程序开发人员的。Spring Cloud根据需要使用FeignClientsConfiguration为每个命名客户端创建一个新的合奏作为ApplicationContext。其中包含feign.Decoder，feign.Encoder和feign.Contract。通过使用@FeignClient批注的contextId属性，可以覆盖该集合的名称。

Spring Cloud使您可以通过使用@FeignClient声明其他配置（在FeignClientsConfiguration之上）来完全控制假客户端。例：

@FeignClient(name = "stores", configuration = FooConfiguration.class)**public** **interface** StoreClient {

*//..*

}

在这种情况下，客户端由FeignClientsConfiguration中已有的组件以及FooConfiguration中的任何组件组成（其中后者将覆盖前者）。

|  |
| --- |
| IMG_352 |
| FooConfiguration不需要用@Configuration进行注释。但是，如果是的话，请注意将其从任何可能包含此配置的@ComponentScan中排除，因为它将成为feign.Decoder，feign.Encoder，feign.Contract等的默认来源，指定时。可以通过将其与任何@ComponentScan或@SpringBootApplication放在单独的，不重叠的包中来避免这种情况，也可以在@ComponentScan中将其明确排除在外。 |

|  |
| --- |
| IMG_353 |
| 现在不推荐使用serviceId属性，而推荐使用name属性。 |

|  |
| --- |
| IMG_354 |
| 除了更改ApplicationContext集合的名称之外，还使用@FeignClient批注的contextId属性，它会覆盖客户端名称的别名，并将其用作配置名称的一部分bean为该客户端创建的。 |

|  |
| --- |
| IMG_355 |
| 以前，使用url属性不需要name属性。现在需要使用name。 |

name和url属性中支持占位符。

@FeignClient(name = "${feign.name}", url = "${feign.url}")**public** **interface** StoreClient {

*//..*

}

Spring Cloud Netflix默认提供以下beans伪装（BeanType beanName：ClassName）：

* Decoder feignDecoder：ResponseEntityDecoder（包装SpringDecoder）
* Encoder feignEncoder：SpringEncoder
* Logger feignLogger：Slf4jLogger
* Contract feignContract：SpringMvcContract
* Feign.Builder feignBuilder：HystrixFeign.Builder
* Client feignClient：如果启用了Ribbon，则它是LoadBalancerFeignClient，否则使用默认的伪装客户端。

可以通过分别将feign.okhttp.enabled或feign.httpclient.enabled设置为true并将其放在类路径中来使用OkHttpClient和ApacheHttpClient虚拟客户端。您可以自定义HTTP客户端，方法是在使用Apache时提供ClosableHttpClient的bean，在使用OK HTTP时提供OkHttpClient。

Spring Cloud Netflix 默认情况下不会为伪装提供以下beans，但仍会从应用程序上下文中查找以下类型的beans以创建伪装客户端：

* Logger.Level
* Retryer
* ErrorDecoder
* Request.Options
* Collection<RequestInterceptor>
* SetterFactory

创建其中一种类型的bean并将其放置在@FeignClient配置中（例如上述FooConfiguration），您可以覆盖上述的每个beans。例：

@Configuration**public** **class** FooConfiguration {

@Bean

**public** Contract feignContract() {

**return** **new** feign.Contract.Default();

}

@Bean

**public** BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

**return** **new** BasicAuthRequestInterceptor("user", "password");

}

}

这将SpringMvcContract替换为feign.Contract.Default，并将RequestInterceptor添加到RequestInterceptor的集合中。

@FeignClient也可以使用配置属性进行配置。

application.yml

feign: client: config: feignName: connectTimeout: 5000 readTimeout: 5000 loggerLevel: full errorDecoder: com.example.SimpleErrorDecoder retryer: com.example.SimpleRetryer requestInterceptors:

- com.example.FooRequestInterceptor

- com.example.BarRequestInterceptor decode404: **false** encoder: com.example.SimpleEncoder decoder: com.example.SimpleDecoder contract: com.example.SimpleContract

可以按照与上述类似的方式在@EnableFeignClients属性defaultConfiguration中指定默认配置。不同之处在于此配置将适用于所有伪客户端。

如果您希望使用配置属性来配置所有@FeignClient，则可以使用default虚拟名称创建配置属性。

application.yml

feign: client: config: default: connectTimeout: 5000 readTimeout: 5000 loggerLevel: basic

如果我们同时创建@Configuration bean和配置属性，则配置属性将获胜。它将覆盖@Configuration值。但是，如果要将优先级更改为@Configuration，可以将feign.client.default-to-properties更改为false。

|  |
| --- |
| IMG_356 |
| 如果您需要在RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE中使用ThreadLocal绑定变量，或者在Feign中禁用Hystrix。 |

application.yml

*# To disable Hystrix in Feign*feign: hystrix: enabled: **false**

*# To set thread isolation to SEMAPHORE*hystrix: command: default: execution: isolation: strategy: SEMAPHORE

如果我们要创建多个具有相同名称或URL的伪装客户端，以便它们指向同一台服务器，但每个客户端使用不同的自定义配置，则必须使用@FeignClient的contextId属性，以避免这些配置beans的名称冲突。

@FeignClient(contextId = "fooClient", name = "stores", configuration = FooConfiguration.class)**public** **interface** FooClient {

*//..*

}

@FeignClient(contextId = "barClient", name = "stores", configuration = BarConfiguration.class)**public** **interface** BarClient {

*//..*

}

## **23.3手动创建Feign客户**

在某些情况下，可能有必要使用上述方法无法实现的方式自定义Feign客户。在这种情况下，您可以使用[Feign Builder API](https://github.com/OpenFeign/feign/" \l "basics" \t "https://www.springcloud.cc/_top)创建客户端 。下面是一个示例，该示例创建两个具有相同接口的Feign客户端，但为每个客户端配置一个单独的请求拦截器。

@Import(FeignClientsConfiguration.class)**class** FooController {

**private** FooClient fooClient;

**private** FooClient adminClient;

@Autowired

**public** FooController(Decoder decoder, Encoder encoder, Client client, Contract contract) {

**this**.fooClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.contract(contract)

.requestInterceptor(**new** BasicAuthRequestInterceptor("user", "user"))

.target(FooClient.**class**, "http://PROD-SVC");

**this**.adminClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.contract(contract)

.requestInterceptor(**new** BasicAuthRequestInterceptor("admin", "admin"))

.target(FooClient.**class**, "http://PROD-SVC");

}

}

|  |
| --- |
| IMG_357 |
| 在上面的示例中，FeignClientsConfiguration.class是Spring Cloud Netflix提供的默认配置。 |

|  |
| --- |
| IMG_358 |
| PROD-SVC是客户将向其请求的服务的名称。 |

|  |
| --- |
| IMG_359 |
| Feign Contract对象定义在接口上有效的注释和值。自动连线的Contract bean提供对SpringMVC注释的支持，而不是默认的Feign本机注释。 |

## **23.4 Feign Hystrix支持**

如果Hystrix在类路径上并且在feign.hystrix.enabled=true上，则Feign将使用断路器包装所有方法。还可以返回com.netflix.hystrix.HystrixCommand。这使您可以使用反应性模式（通过调用.toObservable()或.observe()或异步使用（通过调用.queue()）。

要基于每个客户端禁用Hystrix支持，请创建具有{prototype“范围的普通Feign.Builder，例如：

@Configuration**public** **class** FooConfiguration {

@Bean

@Scope("prototype")

**public** Feign.Builder feignBuilder() {

**return** Feign.builder();

}

}

|  |
| --- |
| IMG_360 |
| 在Spring Cloud Dalston发行版之前，如果Hystrix在类路径Feign上，则默认情况下会将所有方法包装在断路器中。Spring Cloud Dalston中对此默认行为进行了更改，以支持选择加入方法。 |

## **23.5 Feign Hystrix后备**

Hystrix支持回退的概念：当它们的电路断开或出现错误时执行的默认代码路径。要为给定的@FeignClient启用回退，请将fallback属性设置为实现回退的类名称。您还需要将实现声明为Spring bean。

@FeignClient(name = "hello", fallback = HystrixClientFallback.class)**protected** **interface** HystrixClient {

@RequestMapping(method = RequestMethod.GET, value = "/hello")

Hello iFailSometimes();

}

**static** **class** HystrixClientFallback **implements** HystrixClient {

@Override

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback");

}

}

如果需要访问引起后备触发器的原因，则可以使用@FeignClient中的fallbackFactory属性。

@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)**protected** **interface** HystrixClient {

@RequestMapping(method = RequestMethod.GET, value = "/hello")

Hello iFailSometimes();

}

@Component**static** **class** HystrixClientFallbackFactory **implements** FallbackFactory<HystrixClient> {

@Override

**public** HystrixClient create(Throwable cause) {

**return** **new** HystrixClient() {

@Override

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback; reason was: " + cause.getMessage());

}

};

}

}

|  |
| --- |
| IMG_361 |
| Feign中的后备实现以及Hystrix后备如何工作存在局限性。返回com.netflix.hystrix.HystrixCommand和rx.Observable的方法当前不支持后备。 |

## **23.6 Feign和@Primary**

当将Feign与后退Hystrix一起使用时，ApplicationContext中有多个相同类型的beans。这将导致@Autowired无法正常工作，因为没有一个bean或标记为主要的一个。要解决此问题，Spring Cloud Netflix将所有Feign实例标记为@Primary，因此Spring Framework将知道要插入哪个bean。在某些情况下，这可能不是理想的。要关闭此行为，请将@FeignClient的primary属性设置为false。

@FeignClient(name = "hello", primary = false)**public** **interface** HelloClient {

*// methods here*

}

## **23.7 Feign继承支持**

Feign通过单继承接口支持样板API。这允许将常用操作分组为方便的基本接口。

**UserService.java。**

**public** **interface** UserService {

@RequestMapping(method = RequestMethod.GET, value ="/users/{id}")

User getUser(@PathVariable("id") **long** id);

}

**UserResource.java。**

@RestController**public** **class** UserResource **implements** UserService {

}

**UserClient.java。**

**package** project.user;

@FeignClient("users")**public** **interface** UserClient **extends** UserService {

}

|  |
| --- |
| IMG_362 |
| 通常不建议在服务器和客户端之间共享接口。它引入了紧密耦合，并且实际上也不能与当前形式的Spring MVC一起使用（方法参数映射不被继承）。 |

## **23.8 Feign请求/响应压缩**

您可以考虑为Feign请求启用请求或响应GZIP压缩。您可以通过启用以下属性之一来做到这一点：

feign.compression.request.enabled=true

feign.compression.response.enabled=true

Feign请求压缩为您提供的设置类似于您为web服务器设置的设置：

feign.compression.request.enabled=true

feign.compression.request.mime-types=text/xml,application/xml,application/json

feign.compression.request.min-request-size=2048

这些属性使您可以选择压缩媒体类型和最小请求阈值长度。

## **23.9 Feign日志记录**

为每个创建的Feign客户端创建一个记录器。默认情况下，记录器的名称是用于创建Feign客户端的接口的全类名称。Feign日志记录仅响应DEBUG级别。

**application.yml。**

logging.level.project.user.UserClient: DEBUG

您可以为每个客户端配置的Logger.Level对象告诉Feign要记录多少。选择是：

* NONE，无日志记录（**DEFAULT**）。
* BASIC，仅记录请求方法和URL以及响应状态代码和执行时间。
* HEADERS，记录基本信息以及请求和响应头。
* FULL，记录请求和响应的标题，正文和元数据。

例如，以下内容会将Logger.Level设置为FULL：

@Configuration**public** **class** FooConfiguration {

@Bean

Logger.Level feignLoggerLevel() {

**return** Logger.Level.FULL;

}

}

## **23.10 Feign @QueryMap支持**

OpenFeign @QueryMap批注支持将POJO用作GET参数映射。不幸的是，默认的OpenFeign QueryMap注释与Spring不兼容，因为它缺少value属性。

Spring Cloud OpenFeign提供等效的@SpringQueryMap批注，该批注用于将POJO或Map参数注释为查询参数映射。

例如，Params类定义参数param1和param2：

*// Params.java***public** **class** Params {

**private** String param1;

**private** String param2;

*// [Getters and setters omitted for brevity]*

}

以下伪装客户端通过使用@SpringQueryMap批注来使用Params类：

@FeignClient("demo")**public** **class** DemoTemplate {

@GetMapping(path = "/demo")

String demoEndpoint(@SpringQueryMap Params params);

}

## **23.11故障排除**

### **23.11.1早期初始化错误**

根据您使用Feign客户端的方式，启动应用程序时可能会看到初始化错误。要变通解决此问题，您可以在自动接线客户端时使用ObjectProvider。

@Autowired

ObjectProvider<TestFeginClient> testFeginClient;

# **第五部分。Spring Cloud Stream**

## **24. Spring的数据集成之旅简史**

Spring的数据集成之旅始于[Spring Integration](https://projects.spring.io/spring-integration/" \t "https://www.springcloud.cc/_top)。通过其编程模型，它为开发人员提供了一致的开发经验，以构建可以包含[企业集成模式](http://www.enterpriseintegrationpatterns.com/" \t "https://www.springcloud.cc/_top)以与外部系统（例如数据库，消息代理等）连接的应用程序。

快进到云时代，微服务已在企业环境中变得突出。[Spring Boot](https://projects.spring.io/spring-boot/" \t "https://www.springcloud.cc/_top)改变了开发人员构建应用程序的方式。借助Spring的编程模型和Spring Boot处理的运行时职责，无缝开发了基于生产，生产级Spring的独立微服务。

为了将其扩展到数据集成工作负载，Spring Integration和Spring Boot被放到一个新项目中。Spring Cloud Stream出生了。

使用Spring Cloud Stream，开发人员可以：\*隔离地构建，测试，迭代和部署以数据为中心的应用程序。\*应用现代微服务架构模式，包括通过消息传递进行组合。\*以事件为中心的思维将应用程序职责分离。事件可以表示及时发生的事件，下游消费者应用程序可以在不知道事件起源或生产者身份的情况下做出反应。\*将业务逻辑移植到消息代理（例如RabbitMQ，Apache Kafka，Amazon Kinesis）上。\*通过使用项目Reactor的Flux和Kafka Streams API，可以在基于通道的应用程序和基于非通道的应用程序绑定方案之间进行互操作，以支持无状态和有状态的计算。\*依靠框架对常见用例的自动内容类型支持。可以扩展到不同的数据转换类型。

## **25.快速入门**

您可以按照以下三步指南在不到5分钟的时间内尝试Spring Cloud Stream。

我们向您展示如何创建一个Spring Cloud Stream应用程序，该应用程序接收来自您选择的消息传递中间件的消息（稍后会详细介绍），并将接收到的消息记录到控制台。我们称之为LoggingConsumer。尽管不是很实用，但是它很好地介绍了一些主要概念和抽象，使您更容易理解本用户指南的其余部分。

三个步骤如下：

1. [第25.1节“使用Spring Initializr创建示例应用程序”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-creating-sample-application" \o "25.1使用Spring Initializr创建一个示例应用程序)
2. [第25.2节“将项目导入IDE”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-importing-project" \o "25.2将项目导入IDE)
3. [第25.3节“添加，处理和构建消息处理程序”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-adding-message-handler" \o "25.3添加消息处理程序，构建并运行)

## **25.1使用Spring Initializr创建示例应用程序**

要开始使用，请访问[Spring Initializr](https://start.spring.io/" \t "https://www.springcloud.cc/_top)。从那里，您可以生成我们的LoggingConsumer应用程序。为此：

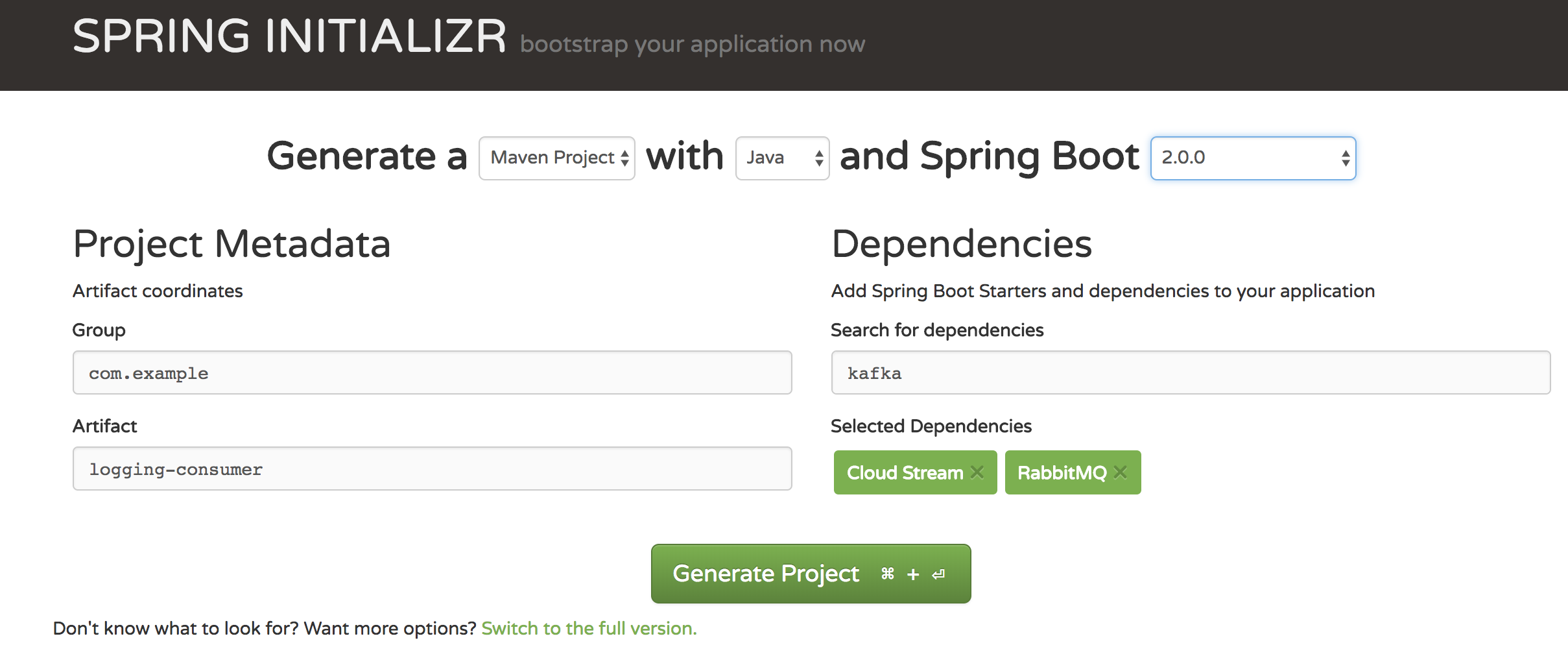
1. 在“ **依赖关系”**部分，开始输入stream。当“ 云流 ”选项出现时，选择它。
2. 开始输入“ kafka”或“兔子”。

选择“ Kafka ”或“ RabbitMQ ”。

基本上，您选择应用程序绑定到的消息传递中间件。我们建议您使用已经安装的那种，或者对安装和运行感到更自在。另外，从“启动程序”屏幕上可以看到，还有一些其他选项可以选择。例如，您可以选择Gradle作为构建工具，而不是Maven（默认设置）。

在**工件**字段中，输入“ logging-consumer”。

**Artifact**字段的值成为应用程序名称。如果您选择RabbitMQ作为中间件，则Spring Initializr现在应该如下所示：



单击**生成项目**按钮。

这样做会将生成的项目的压缩版本下载到硬盘上。

1. 将文件解压缩到要用作项目目录的文件夹中。

|  |
| --- |
| IMG_364 |
| 我们鼓励您探索Spring Initializr中可用的许多可能性。它使您可以创建许多不同种类的Spring应用程序。 |

## **25.2将项目导入IDE**

现在，您可以将项目导入到IDE中。请记住，取决于IDE，您可能需要遵循特定的导入过程。例如，根据项目的生成方式（Maven或Gradle），您可能需要遵循特定的导入过程（例如，在Eclipse或STS中，您需要使用File→Import→Maven→现有的Maven项目）。

导入后，该项目必须没有任何错误。另外，src/main/java应该包含com.example.loggingconsumer.LoggingConsumerApplication。

从技术上讲，此时，您可以运行应用程序的主类。它已经是有效的Spring Boot应用程序。但是，它没有任何作用，因此我们想添加一些代码。

## **25.3添加消息处理程序，构建并运行**

修改com.example.loggingconsumer.LoggingConsumerApplication类，如下所示：

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** LoggingConsumerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(LoggingConsumerApplication.**class**, args);

}

@StreamListener(Sink.INPUT)

**public** **void** handle(Person person) {

System.out.println("Received: " + person);

}

**public** **static** **class** Person {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String toString() {

**return** **this**.name;

}

}

}

从前面的清单中可以看到：

* 我们已经使用@EnableBinding(Sink.class)启用了Sink绑定（输入无输出）。这样做会向框架发出信号，以启动对消息传递中间件的绑定，在该消息传递中间件自动创建绑定到Sink.INPUT通道的目的地（即队列，主题和其他）。
* 我们添加了一个handler方法来接收类型为Person的传入消息。这样做可以使您看到框架的核心功能之一：它尝试自动将传入的消息有效负载转换为类型Person。

您现在有了一个功能齐全的Spring Cloud Stream应用程序，该应用程序确实侦听消息。为了简单起见，我们从这里开始，假设您在[第一步中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-creating-sample-application" \o "25.1使用Spring Initializr创建一个示例应用程序)选择了RabbitMQ 。假设已经安装并运行了RabbitMQ，则可以通过在IDE中运行其main方法来启动应用程序。

您应该看到以下输出：

--- [ main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound: input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg, bound to: input

--- [ main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]

--- [ main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#2a3a299:0/SimpleConnection@66c83fc8. . .

. . .

--- [ main] o.s.i.a.i.AmqpInboundChannelAdapter : started inbound.input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg

. . .

--- [ main] c.e.l.LoggingConsumerApplication : Started LoggingConsumerApplication in 2.531 seconds (JVM running for 2.897)

转到RabbitMQ管理控制台或任何其他RabbitMQ客户端，然后向input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg发送消息。anonymous.CbMIwdkJSBO1ZoPDOtHtCg部分代表组名并已生成，因此在您的环境中它一定是不同的。对于更可预测的内容，可以通过设置spring.cloud.stream.bindings.input.group=hello（或您喜欢的任何名称）来使用显式组名。

消息的内容应为Person类的JSON表示形式，如下所示：

{"name":"Sam Spade"}

然后，在控制台中，您应该看到：

Received: Sam Spade

您还可以将应用程序生成并打包到引导jar中（使用./mvnw clean install），并使用java -jar命令运行生成的JAR。

现在，您有了一个正在运行的（尽管非常基础的）Spring Cloud Stream应用程序。

## **26. 2.0中有什么新功能？**

Spring Cloud Stream引入了许多新功能，增强功能和更改。以下各节概述了最值得注意的部分：

* [第26.1节“新功能和组件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-new-features" \o "26.1新功能和组件)
* [第26.2节“显着增强”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-notable-enhancements" \o "26.2显着增强)

## **26.1新功能和组件**

* **轮询使用者**：引入轮询使用者，使应用程序可以控制消息处理速率。请参见 “ [第29.3.5，‘使用轮询消费者](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-streams-overview-using-polled-consumers" \o "29.3.5使用轮询的使用者) ’ ”的更多细节。您也可以阅读[此博客文章](https://spring.io/blog/2018/02/27/spring-cloud-stream-2-0-polled-consumers" \t "https://www.springcloud.cc/_top)以获取更多详细信息。
* **千分尺支持**：度量标准已切换为使用[千分尺](https://micrometer.io/" \t "https://www.springcloud.cc/_top)。MeterRegistry也以bean的形式提供，以便自定义应用程序可以将其自动连线以捕获自定义指标。有关更多详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-metrics-emitter" \o "37. Metrics Emitter)请参见 “ [第37章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-metrics-emitter" \o "37.指标发射器)*[度量标准发射器](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-metrics-emitter" \o "37.指标发射器)* ”。
* **新的执行器绑定控件**：新的执行器绑定控件使您可以可视化并控制绑定的生命周期。有关更多详细信息，请参见[第30.6节“绑定可视化和控件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_binding_visualization_and_control" \o "30.6绑定可视化和控制)。
* **可配置的RetryTemplate**：除了提供配置RetryTemplate的属性外，我们现在还允许您提供自己的模板，有效地覆盖了框架提供的模板。要使用它，请在您的应用程序中将其配置为@Bean。

## **26.2显着增强**

此版本包括以下显着增强：

* [第26.2.1节“致动器和Web依赖关系现在都是可选的”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-actuator-web-dependencies" \o "26.2.1现在，执行器和Web依赖项都是可选的)
* [第26.2.2节“内容类型协商的改进”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-content-type-negotiation-improvements" \o "26.2.2内容类型协商的改进)
* [第26.3节“显着的弃用”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-notable-deprecations" \o "26.3显着弃用)

### **26.2.1执行器和Web依赖项现在都是可选的**

如果既不需要执行器也不需要web依赖项，那么此更改将减少已部署应用程序的占用空间。通过手动添加以下依赖项之一，它还使您可以在反应式和常规web范式之间切换。

以下清单显示了如何添加常规的web框架：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId></dependency>

以下清单显示了如何添加反应式web框架：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId></dependency>

下表显示了如何添加执行器依赖性：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId></dependency>

### **26.2.2内容类型协商的改进**

Verion 2.0的核心主题之一是围绕内容类型协商和消息转换的改进（在一致性和性能方面）。以下摘要概述了该领域的显着变化和改进。有关更多详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32. Content Type Negotiation)请参见“ [第32章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)* ”部分。此外，[此博客文章还](https://spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation" \t "https://www.springcloud.cc/_top)包含更多详细信息。

* 现在，所有消息转换**仅**由MessageConverter对象处理。
* 我们引入了@StreamMessageConverter批注以提供自定义MessageConverter对象。
* 我们引入了默认的Content Type作为application/json，在迁移1.3应用程序或以混合模式（即1.3生产者→2.0消费者）进行操作时，需要考虑该默认值。
* 在无法确定所提供的MessageHandler的参数类型的情况下，带有文本有效载荷且contentType为text/…​或…​/json的消息不再转换为Message<String>。 public void handle(Message<?> message)或public void handle(Object payload)）。此外，强参数类型可能不足以正确地转换消息，因此contentType标头可能被某些MessageConverters用作补充。

## **26.3显着弃用**

从2.0版开始，不推荐使用以下项目：

* [第26.3.1节“ Java序列化（Java Native和Kryo​​）”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-deprecation-java-serialization" \o "26.3.1 Java序列化（Java本机和Kryo​​）)
* [第26.3.2节“不推荐使用的类和方法”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-preface-deprecation-classes-methods" \o "26.3.2不推荐使用的类和方法)

### **26.3.1 Java序列化（Java本机和Kryo​​）**

JavaSerializationMessageConverter和KryoMessageConverter暂时保留。但是，我们计划将来将它们移出核心软件包和支持。弃用此文件的主要原因是要标记基于类型，特定于语言的序列化可能在分布式环境中引起的问题，在该环境中，生产者和使用者可能依赖于不同的JVM版本或具有不同版本的支持库（即Kryo）。我们还想提请注意这样一个事实，即消费者和生产者甚至可能都不是基于Java的，因此多语言风格的序列化（即JSON）更适合。

### **26.3.2不推荐使用的类和方法**

以下是显着弃用的快速摘要。有关更多详细信息，请参见相应的{spring-cloud-stream-javadoc-current} [javadoc]。

* SharedChannelRegistry.使用SharedBindingTargetRegistry。
* Bindings.符合条件的Beans已通过其类型唯一标识，例如，提供了Source，Processor或自定义绑定：

public interface Sample {

String OUTPUT = "sampleOutput";

@Output(Sample.OUTPUT)

MessageChannel output();

}

* HeaderMode.raw.使用none，headers或embeddedHeaders
* ProducerProperties.partitionKeyExtractorClass赞成partitionKeyExtractorName，而ProducerProperties.partitionSelectorClass赞成partitionSelectorName。此更改可确保Spring配置和管理两个组件，并以Spring友好的方式对其进行引用。
* BinderAwareRouterBeanPostProcessor.在保留该组件的同时，它不再是BeanPostProcessor，并且将来会重命名。
* BinderProperties.setEnvironment(Properties environment).使用BinderProperties.setEnvironment(Map<String, Object> environment)。

本节将详细介绍如何使用Spring Cloud Stream。它涵盖了诸如创建和运行流应用程序之类的主题。

## **27.简介Spring Cloud Stream**

Spring Cloud Stream是用于构建消息驱动的微服务应用程序的框架。Spring Cloud Stream在Spring Boot的基础上创建了独立的生产级Spring应用程序，并使用Spring Integration提供了到消息代理的连接。它提供了来自多家供应商的中间件的合理配置，并介绍了持久性发布-订阅语义，使用者组和分区的概念。

您可以在应用程序中添加@EnableBinding批注，以立即连接到消息代理，还可以在方法中添加@StreamListener，以使其接收流处理的事件。以下示例显示了接收外部消息的接收器应用程序：

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** VoteRecordingSinkApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(VoteRecordingSinkApplication.**class**, args);

}

@StreamListener(Sink.INPUT)

**public** **void** processVote(Vote vote) {

votingService.recordVote(vote);

}

}

@EnableBinding批注将一个或多个接口作为参数（在这种情况下，该参数是单个Sink接口）。接口声明输入和输出通道。Spring Cloud Stream提供了Source，Sink和Processor接口。您也可以定义自己的接口。

以下清单显示了Sink接口的定义：

**public** **interface** Sink {

String INPUT = "input";

@Input(Sink.INPUT)

SubscribableChannel input();

}

@Input注释标识一个输入通道，接收到的消息通过该输入通道进入应用程序。@Output注释标识一个输出通道，已发布的消息通过该输出通道离开应用程序。@Input和@Output批注可以使用频道名称作为参数。如果未提供名称，则使用带注释的方法的名称。

Spring Cloud Stream为您创建接口的实现。您可以通过自动装配在应用程序中使用它，如以下示例所示（来自测试用例）：

@RunWith(SpringJUnit4ClassRunner.class)@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)@WebAppConfiguration@DirtiesContext**public** **class** StreamApplicationTests {

@Autowired

**private** Sink sink;

@Test

**public** **void** contextLoads() {

assertNotNull(**this**.sink.input());

}

}

## **28.主要概念**

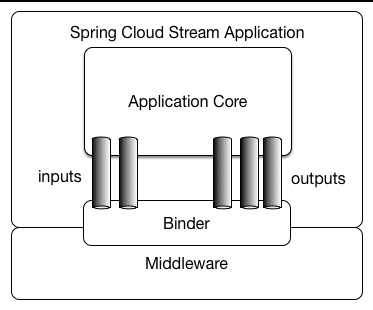
Spring Cloud Stream提供了许多抽象和原语，简化了消息驱动的微服务应用程序的编写。本节概述以下内容：

* [Spring Cloud Stream的应用程序模型](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-application-model" \o "28.1应用模型)
* [第28.2节“ Binder抽象”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-binder-abstraction" \o "28.2活页夹抽象)
* [持久的发布-订阅支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-persistent-publish-subscribe-support" \o "28.3持久的发布-订阅支持)
* [消费者群体支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "consumer-groups" \o "28.4消费群体)
* [分区支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "partitioning" \o "28.6分区支持)
* [可插拔的Binder SPI](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-binder-api" \o "30.2粘合剂SPI)

## **28.1应用模型**

Spring Cloud Stream应用程序由与中间件无关的内核组成。该应用程序通过Spring Cloud Stream注入到其中的输入和输出通道与外界进行通信。通道通过特定于中间件的Binder实现与外部代理连接。

**图28.1 Spring Cloud Stream申请**



### **28.1.1 Fat JAR**

可以从IDE以独立模式运行Spring Cloud Stream应用程序以进行测试。要在生产环境中运行Spring Cloud Stream应用程序，您可以使用为Maven或Gradle提供的标准Spring Boot工具来创建可执行（或“ fat ”）JAR。有关更多详细信息，请参见[Spring Boot参考指南](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-build.html" \l "howto-create-an-executable-jar-with-maven" \t "https://www.springcloud.cc/_top)。

## **28.2 Binder抽象**

Spring Cloud Stream为[Kafka](https://github.com/spring-cloud/spring-cloud-stream-binder-kafka" \t "https://www.springcloud.cc/_top)和[Rabbit MQ](https://github.com/spring-cloud/spring-cloud-stream-binder-rabbit" \t "https://www.springcloud.cc/_top)提供了Binder实现。Spring Cloud Stream还包括一个[TestSupportBinder](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-test-support/src/main/java/org/springframework/cloud/stream/test/binder/TestSupportBinder.java" \t "https://www.springcloud.cc/_top)，它使通道保持[不变](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-test-support/src/main/java/org/springframework/cloud/stream/test/binder/TestSupportBinder.java" \t "https://www.springcloud.cc/_top)，因此测试可以与通道直接交互并可靠地断言所接收的内容。您也可以使用可扩展的API编写自己的Binder。

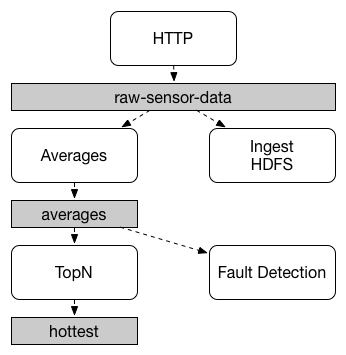
Spring Cloud Stream使用Spring Boot进行配置，而Binder抽象使Spring Cloud Stream应用程序可以灵活地连接中间件。例如，部署者可以在运行时动态选择通道连接到的目的地（例如Kafka主题或RabbitMQ交换）。可以通过外部配置属性以及Spring Boot支持的任何形式（包括应用程序参数，环境变量以及application.yml或application.properties文件）提供这种配置。在[第27章“](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-introducing" \o "27.介绍Spring Cloud Stream)*[介绍Spring Cloud Stream”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-introducing" \o "27.介绍Spring Cloud Stream)*的接收器示例中，将spring.cloud.stream.bindings.input.destination应用程序属性设置为raw-sensor-data会使其从raw-sensor-data Kafka主题或绑定到该队列的队列中读取raw-sensor-data RabbitMQ交换。

Spring Cloud Stream自动检测并使用在类路径上找到的活页夹。您可以使用具有相同代码的不同类型的中间件。为此，在构建时包括一个不同的活页夹。对于更复杂的用例，您还可以在应用程序中打包多个活页夹，并在运行时选择活页夹（甚至为不同的通道使用不同的活页夹）。

## **28.3持久的发布-订阅支持**

应用程序之间的通信遵循发布-订阅模型，其中数据通过共享主题进行广播。在下图中可以看到，该图显示了一组交互的Spring Cloud Stream应用程序的典型部署。

**图28.2 Spring Cloud Stream发布-订阅**



传感器报告给HTTP端点的数据将发送到名为raw-sensor-data的公共目标。从目的地开始，它由计算时间窗平均值的微服务应用程序和另一个将原始数据提取到HDFS（Hadoop分布式文件系统）的微服务应用程序独立处理。为了处理数据，两个应用程序都在运行时将主题声明为其输入。

发布-订阅通信模型降低了生产者和使用者的复杂性，并允许在不中断现有流程的情况下将新应用添加到拓扑中。例如，在平均计算应用程序的下游，您可以添加一个应用程序，该应用程序计算用于显示和监视的最高温度值。然后，您可以添加另一个解释相同平均值流以进行故障检测的应用程序。通过共享主题而不是点对点队列进行所有通信可以减少微服务之间的耦合。

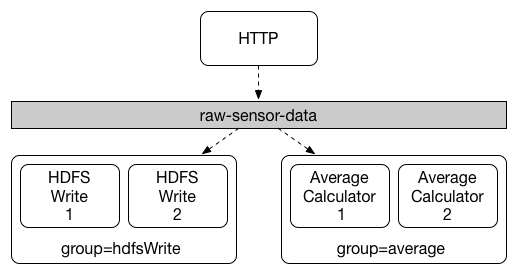
尽管发布-订阅消息传递的概念并不是新概念，但是Spring Cloud Stream采取了额外的步骤，使其成为其应用程序模型的明智选择。通过使用本机中间件支持，Spring Cloud Stream还简化了跨不同平台的发布-订阅模型的使用。

## **28.4消费群体**

尽管发布-订阅模型使通过共享主题轻松连接应用程序变得很重要，但是通过创建给定应用程序的多个实例进行扩展的能力同样重要。这样做时，会将应用程序的不同实例置于竞争的消费者关系中，在该消费者关系中，仅其中一个实例可以处理给定消息。

Spring Cloud Stream通过消费者群体的概念对这种行为进行建模。（Spring Cloud Stream消费者组类似于Kafka消费者组并受其启发。）每个消费者绑定都可以使用spring.cloud.stream.bindings.<channelName>.group属性来指定组名。对于下图所示的消费者，此属性将设置为spring.cloud.stream.bindings.<channelName>.group=hdfsWrite或spring.cloud.stream.bindings.<channelName>.group=average。

**图28.3 Spring Cloud Stream消费群体**



订阅给定目标的所有组都将收到已发布数据的副本，但是每个组中只有一个成员从该目标接收给定消息。默认情况下，未指定组时，Spring Cloud Stream会将应用程序分配给与所有其他使用者组具有发布-订阅关系的匿名且独立的单成员使用者组。

## **28.5消费者类型**

支持两种类型的使用者：

* 消息驱动（有时称为异步）
* 轮询（有时称为同步）

在2.0版之前，仅支持异步使用者。消息一旦可用，就会被传递，并且有线程可以处理它。

当您希望控制消息的处理速率时，可能需要使用同步使用者。

### **28.5.1耐久性**

与公认的Spring Cloud Stream应用程序模型一致，消费者组订阅是持久的。也就是说，活页夹实现可确保组订阅是持久的，并且一旦创建了至少一个组订阅，该组将接收消息，即使在组中所有应用程序停止时发送消息也是如此。

|  |
| --- |
| IMG_368 |
| 匿名订阅本质上是非持久的。对于某些活页夹实现（例如RabbitMQ），可能具有非持久的组订阅。 |

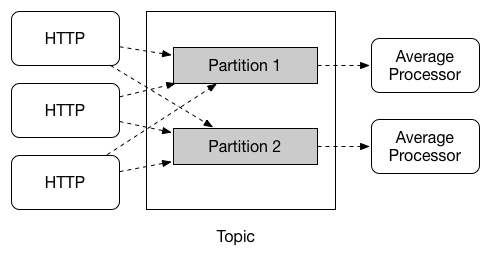
通常，在将应用程序绑定到给定目标时，最好始终指定使用者组。扩展Spring Cloud Stream应用程序时，必须为其每个输入绑定指定使用者组。这样做可以防止应用程序的实例接收重复的消息（除非需要这种行为，这是不寻常的）。

## **28.6分区支持**

Spring Cloud Stream支持在给定应用程序的多个实例之间分区数据。在分区方案中，物理通信介质（例如代理主题）被视为结构化为多个分区。一个或多个生产者应用程序实例将数据发送到多个消费者应用程序实例，并确保由共同特征标识的数据由同一消费者实例处理。

Spring Cloud Stream提供了用于以统一方式实现分区处理用例的通用抽象。因此，无论代理本身是否自然地被分区（例如，Kafka）（例如，RabbitMQ），都可以使用分区。

**图28.4。Spring Cloud Stream分区**



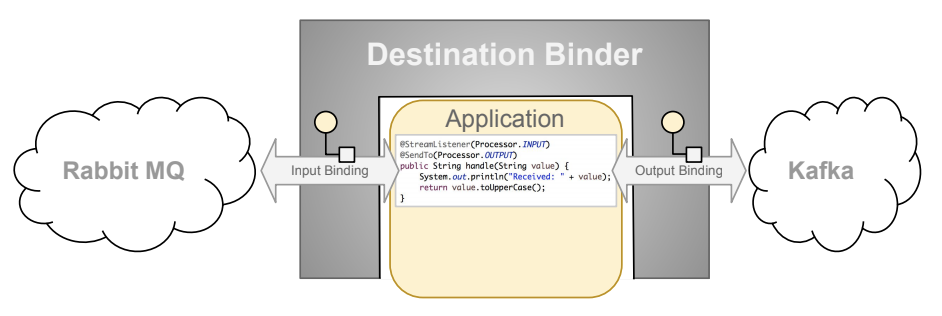
分区是有状态处理中的关键概念，对于确保所有相关数据都一起处理，分区是至关重要的（出于性能或一致性方面的考虑）。例如，在带时间窗的平均计算示例中，重要的是，来自任何给定传感器的所有测量都应由同一应用实例处理。

|  |
| --- |
| IMG_370 |
| 要设置分区处理方案，必须同时配置数据产生端和数据消耗端。 |

## **29.编程模型**

要了解编程模型，您应该熟悉以下核心概念：

* **目标Binders：**负责与外部消息传递系统集成的组件。
* **目标绑定：**在外部消息传递系统和应用程序之间提供的消息的生产者和使用者（由目标Binders创建）之间的桥梁。
* **消息：**生产者和消费者用来与目的地Binders（以及因此通过外部消息系统进行的其他应用程序）通信的规范数据结构。



## **29.1目的地Binders**

目标Binders是Spring Cloud Stream的扩展组件，负责提供必要的配置和实现以促进与外部消息传递系统的集成。这种集成负责连接，委派和与生产者和消费者之间的消息路由，数据类型转换，用户代码调用等等。

Binders承担了许多样板工作，否则这些工作就落在了您的肩上。但是，要实现这一点，活页夹仍然需要用户提供的一些简单但需要的指令集形式的帮助，通常以某种类型的配置形式出现。

尽管讨论所有可用的绑定器和绑定配置选项（本手册的其余部分都涉及它们）不在本节的讨论范围之内，但 目标绑定确实需要特别注意。下一节将详细讨论。

## **29.2目标绑定**

如前所述，目标绑定提供了外部消息传递系统与应用程序提供的生产者和消费者之间的桥梁。

将@EnableBinding批注应用于应用程序的配置类之一可定义目标绑定。@EnableBinding注释本身使用@Configuration进行元注释，并触发Spring Cloud Stream基础结构的配置。

下面的示例显示了一个功能完整且运行正常的Spring Cloud Stream应用程序，该应用程序从INPUT目标接收的消息净荷为String类型（请参见[第32章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*部分），并将其记录到控制台，并将其转换为大写字母后将其发送到OUTPUT目标。

@SpringBootApplication@EnableBinding(Processor.class)**public** **class** MyApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(MyApplication.**class**, args);

}

@StreamListener(Processor.INPUT)

@SendTo(Processor.OUTPUT)

**public** String handle(String value) {

System.out.println("Received: " + value);

**return** value.toUpperCase();

}

}

如您所见，@EnableBinding批注可以将一个或多个接口类作为参数。这些参数称为绑定，它们包含表示可绑定组件的方法。这些组件通常是基于通道的活页夹（例如Rabbit，Kafka等）的消息通道（请参见[Spring消息传递](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-messaging.html" \t "https://www.springcloud.cc/_top)）。但是，其他类型的绑定可以为相应技术的本机功能提供支持。例如，Kafka Streams绑定器（以前称为KStream）允许直接绑定到Kafka Streams（有关更多详细信息，请参见[Kafka Streams](https://docs.spring.io/autorepo/docs/spring-cloud-stream-binder-kafka-docs/1.1.0.M1/reference/htmlsingle/" \t "https://www.springcloud.cc/_top)）。

Spring Cloud Stream已经为典型的消息交换合同提供了绑定接口，其中包括：

* **接收器：**通过提供消费消息的目的地来标识消息消费者的合同。
* **源：**通过提供将生成的消息发送到的目的地，来标识消息生产者的合同。
* **处理器：**通过公开两个允许使用和产生消息的目的地，封装了接收器和源协定。

**public** **interface** Sink {

String INPUT = "input";

@Input(Sink.INPUT)

SubscribableChannel input();

}

**public** **interface** Source {

String OUTPUT = "output";

@Output(Source.OUTPUT)

MessageChannel output();

}

**public** **interface** Processor **extends** Source, Sink {}

尽管前面的示例满足了大多数情况，但是您也可以通过定义自己的绑定接口并使用@Input和@Output批注来标识实际的可绑定组件，从而定义自己的合同。

例如：

**public** **interface** Barista {

@Input

SubscribableChannel orders();

@Output

MessageChannel hotDrinks();

@Output

MessageChannel coldDrinks();

}

将上一个示例中显示的接口用作@EnableBinding的参数将分别触发三个绑定通道的创建，分别命名为orders，hotDrinks和coldDrinks。

您可以根据需要提供任意数量的绑定接口，作为@EnableBinding批注的参数，如以下示例所示：

@EnableBinding(value = { Orders.**class**, Payment.**class** })

在Spring Cloud Stream中，可绑定的MessageChannel组件是Spring消息传递MessageChannel（用于出站）及其扩展名SubscribableChannel（用于入站）。

**可轮询的目标绑定**

尽管前面描述的绑定支持基于事件的消息使用，但是有时您需要更多控制，例如使用率。

从2.0版开始，您现在可以绑定可轮询的使用者：

以下示例显示了如何绑定可轮询的使用者：

**public** **interface** PolledBarista {

@Input

PollableMessageSource orders();

. . .

}

在这种情况下，PollableMessageSource的实现绑定到orders“通道”。有关更多详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-streams-overview-using-polled-consumers" \o "29.3.5使用轮询的使用者)请参见[第29.3.5节“使用轮询的使用者”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-streams-overview-using-polled-consumers" \o "29.3.5 Using Polled Consumers)。

**自定义频道名称**

通过使用@Input和@Output批注，可以为该通道指定自定义的通道名称，如以下示例所示：

**public** **interface** Barista {

@Input("inboundOrders")

SubscribableChannel orders();

}

在前面的示例中，创建的绑定通道被命名为inboundOrders。

通常，您不需要直接访问各个通道或绑定（除非通过@EnableBinding注释对其进行配置）。但是，您有时可能会遇到诸如测试或其他极端情况的情况。

除了为每个绑定生成通道并将其注册为Spring beans外，对于每个绑定接口，Spring Cloud Stream还会生成一个实现该接口的bean。这意味着您可以通过在应用程序中自动接线来访问表示绑定或各个通道的接口，如以下两个示例所示：

自动接线绑定界面

@Autowire**private** Source source

**public** **void** sayHello(String name) {

source.output().send(MessageBuilder.withPayload(name).build());

}

自动连线个别频道

@Autowire**private** MessageChannel output;

**public** **void** sayHello(String name) {

output.send(MessageBuilder.withPayload(name).build());

}

对于自定义通道名称或在需要特别命名通道的多通道方案中，您也可以使用标准Spring的@Qualifier批注。

下面的示例演示如何以这种方式使用@Qualifier批注：

@Autowire@Qualifier("myChannel")**private** MessageChannel output;

## **29.3产生和消耗消息**

您可以使用Spring Integration注释或Spring Cloud Stream本机注释编写Spring Cloud Stream应用程序。

### **29.3.1 Spring Integration支持**

Spring Cloud Stream建立在[Enterprise Integration Patterns](http://www.enterpriseintegrationpatterns.com/" \t "https://www.springcloud.cc/_top)定义的概念和模式的基础之上，并依靠其内部实现依赖于Spring项目组合[Spring Integration](https://projects.spring.io/spring-integration/" \t "https://www.springcloud.cc/_top)框架中已经建立且流行的Enterprise Integration Patterns实现 。

因此，它支持Spring Integration已经建立的基础，语义和配置选项是很自然的。

例如，您可以将Source的输出通道附加到MessageSource并使用熟悉的@InboundChannelAdapter注释，如下所示：

@EnableBinding(Source.class)**public** **class** TimerSource {

@Bean

@InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10", maxMessagesPerPoll = "1"))

**public** MessageSource<String> timerMessageSource() {

**return** () -> **new** GenericMessage<>("Hello Spring Cloud Stream");

}

}

同样，可以在提供处理器绑定合同的消息处理程序方法的实现时使用@Transformer或@ServiceActivator ，如以下示例所示：

@EnableBinding(Processor.class)**public** **class** TransformProcessor {

@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)

**public** Object transform(String message) {

**return** message.toUpperCase();

}

}

|  |
| --- |
| IMG_372 |
| 尽管这可能会略过一些，但重要的是要了解，当您使用@StreamListener注释从同一绑定进行消费时，将使用pub-sub模型。用@StreamListener注释的每种方法都收到其自己的消息副本，并且每种方法都有其自己的使用者组。但是，如果通过使用Spring Integration批注之一（例如@Aggregator，@Transformer或@ServiceActivator）从同一绑定中消费，则这些消费在竞争模型中。没有为每个订阅创建单独的消费者组。 |

### **29.3.2使用@StreamListener注释**

作为对Spring Integration支持的补充，Spring Cloud Stream提供了自己的@StreamListener注释，其模仿其他Spring消息注释（@MessageMapping，@JmsListener，@RabbitListener等）并提供便利，例如基于内容的路由等。

@EnableBinding(Sink.class)**public** **class** VoteHandler {

@Autowired

VotingService votingService;

@StreamListener(Sink.INPUT)

**public** **void** handle(Vote vote) {

votingService.record(vote);

}

}

与其他Spring消息传递方法一样，方法参数可以用@Payload，@Headers和@Header进行注释。

对于返回数据的方法，必须使用@SendTo批注为该方法返回的数据指定输出绑定目标，如以下示例所示：

@EnableBinding(Processor.class)**public** **class** TransformProcessor {

@Autowired

VotingService votingService;

@StreamListener(Processor.INPUT)

@SendTo(Processor.OUTPUT)

**public** VoteResult handle(Vote vote) {

**return** votingService.record(vote);

}

}

### **29.3.3使用@StreamListener进行基于内容的路由**

Spring Cloud Stream支持根据条件将消息调度到用@StreamListener注释的多个处理程序方法。

为了有资格支持条件分派，一种方法必须满足以下条件：

* 它不能返回值。
* 它必须是单独的消息处理方法（不支持反应性API方法）。

该条件由注释的condition参数中的SpEL表达式指定，并针对每条消息进行评估。所有与条件匹配的处理程序都在同一线程中调用，并且不必假设调用的顺序。

在具有分配条件的@StreamListener的以下示例中，所有带有标头type且具有值bogey的消息都被分配到receiveBogey方法，所有带有标头{11的消息值bacall的/}发送到receiveBacall方法。

@EnableBinding(Sink.class)@EnableAutoConfiguration**public** **static** **class** TestPojoWithAnnotatedArguments {

@StreamListener(target = Sink.INPUT, condition = "headers['type']=='bogey'")

**public** **void** receiveBogey(@Payload BogeyPojo bogeyPojo) {

*// handle the message*

}

@StreamListener(target = Sink.INPUT, condition = "headers['type']=='bacall'")

**public** **void** receiveBacall(@Payload BacallPojo bacallPojo) {

*// handle the message*

}

}

condition**上下文中的内容类型协商**

了解使用@StreamListener参数condition的基于内容的路由背后的一些机制很重要，尤其是在整个消息类型的上下文中。如果在继续之前熟悉[第32章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32. Content Type Negotiation)这也可能会有所帮助。

请考虑以下情形：

@EnableBinding(Sink.class)@EnableAutoConfiguration**public** **static** **class** CatsAndDogs {

@StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Dog'")

**public** **void** bark(Dog dog) {

*// handle the message*

}

@StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Cat'")

**public** **void** purr(Cat cat) {

*// handle the message*

}

}

前面的代码是完全有效的。它可以毫无问题地进行编译和部署，但是永远不会产生您期望的结果。

这是因为您正在测试的东西在您期望的状态下尚不存在。这是因为消息的有效负载尚未从有线格式（byte[]）转换为所需的类型。换句话说，它尚未经过[第32章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*中描述的类型转换过程。

因此，除非使用SPeL表达式评估原始数据（例如，字节数组中第一个字节的值），否则请使用基于消息标头的表达式（例如condition = "headers['type']=='dog'"）。

|  |
| --- |
| IMG_373 |
| 目前，仅基于通道的绑定程序（不支持响应编程）支持通过@StreamListener条件进行分派。 |

### **29.3.4 Spring Cloud功能支持**

从Spring Cloud Stream v2.1开始，定义流处理程序和源的另一种方法是使用对[Spring Cloud函数的](https://cloud.spring.io/spring-cloud-function/" \t "https://www.springcloud.cc/_top)内置支持，其中可以将它们表示为java.util.function.[Supplier/Function/Consumer]类型的beans。

若要指定要绑定到绑定公开的外部目标的功能bean，必须提供spring.cloud.stream.function.definition属性。

这是Processor应用程序将消息处理程序公开为java.util.function.Function的示例

@SpringBootApplication@EnableBinding(Processor.class)**public** **class** MyFunctionBootApp {

**public** **static** **void** main(String[] args) {

SpringApplication.run(MyFunctionBootApp.**class**, "--spring.cloud.stream.function.definition=toUpperCase");

}

@Bean

**public** Function<String, String> toUpperCase() {

**return** s -> s.toUpperCase();

}

}

在上面的代码中，我们仅定义了类型为java.util.function.Function的bean（称为toUpperCase）并将其标识为bean，用作消息处理程序，其“输入”和“输出”必须绑定到外部目标由处理器绑定公开。

以下是支持源，处理器和接收器的简单功能应用程序的示例。

这是定义为java.util.function.Supplier的Source应用程序的示例

@SpringBootApplication@EnableBinding(Source.class)**public** **static** **class** SourceFromSupplier {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SourceFromSupplier.**class**, "--spring.cloud.stream.function.definition=date");

}

@Bean

**public** Supplier<Date> date() {

**return** () -> **new** Date(12345L);

}

}

这是定义为java.util.function.Function的Processor应用程序的示例

@SpringBootApplication@EnableBinding(Processor.class)**public** **static** **class** ProcessorFromFunction {

**public** **static** **void** main(String[] args) {

SpringApplication.run(ProcessorFromFunction.**class**, "--spring.cloud.stream.function.definition=toUpperCase");

}

@Bean

**public** Function<String, String> toUpperCase() {

**return** s -> s.toUpperCase();

}

}

这是一个定义为java.util.function.Consumer的接收器应用程序的示例

@EnableAutoConfiguration@EnableBinding(Sink.class)**public** **static** **class** SinkFromConsumer {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SinkFromConsumer.**class**, "--spring.cloud.stream.function.definition=sink");

}

@Bean

**public** Consumer<String> sink() {

**return** System.out::println;

}

}

#### **功能组成**

使用此编程模型，您还可以从功能组合中受益，在该功能组合中，您可以从一组简单的函数中动态组成复杂的处理程序。作为示例，我们将以下函数bean添加到上面定义的应用程序中

@Bean**public** Function<String, String> wrapInQuotes() {

**return** s -> "\"" + s + "\"";

}

并修改spring.cloud.stream.function.definition属性以反映您打算从'toUpperCase'和'wrapInQuotes'编写新函数的意图。为此，可以使用Spring Cloud函数使用|（管道）符号。因此，完成我们的示例，我们的属性现在将如下所示：

—spring.cloud.stream.function.definition=toUpperCase|wrapInQuotes

### **29.3.5使用轮询的使用者**

#### **总览**

使用轮询的使用者时，您可以按需轮询PollableMessageSource。考虑以下受调查消费者的示例：

**public** **interface** PolledConsumer {

@Input

PollableMessageSource destIn();

@Output

MessageChannel destOut();

}

给定上一个示例中的受调查消费者，您可以按以下方式使用它：

@Bean**public** ApplicationRunner poller(PollableMessageSource destIn, MessageChannel destOut) {

**return** args -> {

**while** (someCondition()) {

**try** {

**if** (!destIn.poll(m -> {

String newPayload = ((String) m.getPayload()).toUpperCase();

destOut.send(**new** GenericMessage<>(newPayload));

})) {

Thread.sleep(1000);

}

}

**catch** (Exception e) {

*// handle failure*

}

}

};

}

PollableMessageSource.poll()方法采用一个MessageHandler参数（通常为lambda表达式，如此处所示）。如果收到并成功处理了消息，它将返回true。

与消息驱动的使用者一样，如果MessageHandler引发异常，消息将发布到错误通道，如“ [???”中所述。](https://www.springcloud.cc/spring-cloud-greenwich.html)”。

通常，poll()方法会在MessageHandler退出时确认该消息。如果该方法异常退出，则该消息将被拒绝（不重新排队），但请参阅[“处理错误”一节](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "polled-errors" \o "处理错误)。您可以通过对确认负责来覆盖该行为，如以下示例所示：

@Bean**public** ApplicationRunner poller(PollableMessageSource dest1In, MessageChannel dest2Out) {

**return** args -> {

**while** (someCondition()) {

**if** (!dest1In.poll(m -> {

StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).noAutoAck();

*// e.g. hand off to another thread which can perform the ack*

*// or acknowledge(Status.REQUEUE)*

})) {

Thread.sleep(1000);

}

}

};

}

|  |  |
| --- | --- |
| IMG_374 | **重要** |
| 您必须在某一时刻ack（或nack）消息，以避免资源泄漏。 |

|  |  |
| --- | --- |
| IMG_375 | **重要** |
| 某些消息传递系统（例如Apache Kafka）在日志中维护简单的偏移量。如果传递失败，并用StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUEUE);重新排队，则重新传递任何以后成功确认的消息。 |

还有一个重载的poll方法，其定义如下：

poll(MessageHandler handler, ParameterizedTypeReference<?> type)

type是一个转换提示，它允许转换传入的消息有效负载，如以下示例所示：

**boolean** result = pollableSource.poll(received -> {

Map<String, Foo> payload = (Map<String, Foo>) received.getPayload();

...

}, **new** ParameterizedTypeReference<Map<String, Foo>>() {});

#### **处理错误**

默认情况下，为可轮询源配置了一个错误通道。如果回调引发异常，则将ErrorMessage发送到错误通道（<destination>.<group>.errors）；此错误通道也桥接到全局Spring Integration errorChannel。

您可以使用@ServiceActivator订阅任何一个错误通道来处理错误。如果没有订阅，则将仅记录错误并确认消息成功。如果错误通道服务激活器引发异常，则该消息将被拒绝（默认情况下），并且不会重新发送。如果服务激活器抛出RequeueCurrentMessageException，则该消息将在代理处重新排队，并在随后的轮询中再次检索。

如果侦听器直接抛出RequeueCurrentMessageException，则如上所述，该消息将重新排队，并且不会发送到错误通道。

## **29.4错误处理**

错误会发生，Spring Cloud Stream提供了几种灵活的机制来处理它们。错误处理有两种形式：

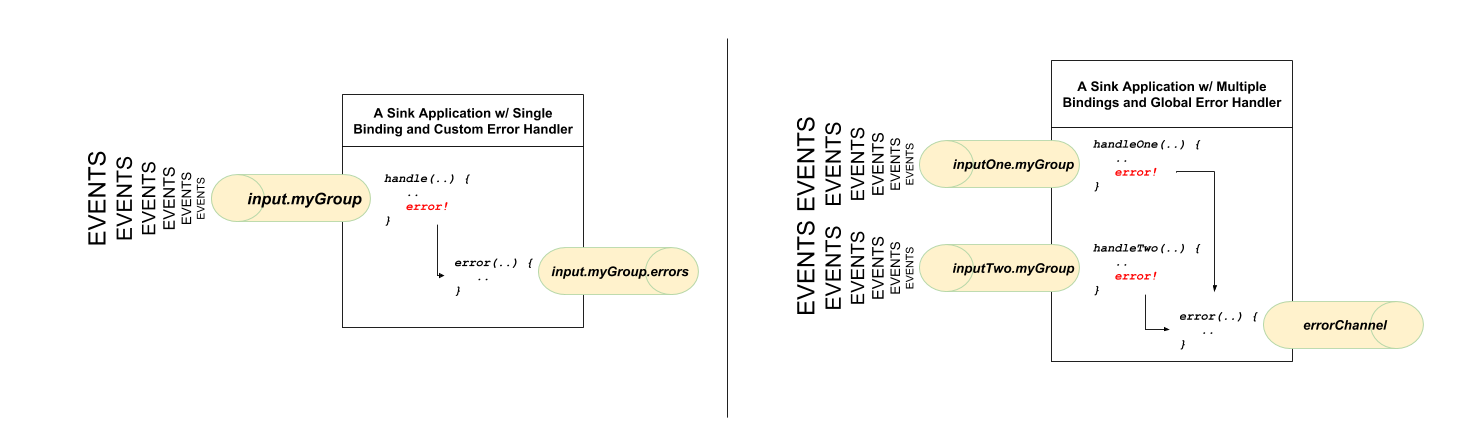
* **应用程序：**错误处理是在应用程序（自定义错误处理程序）中完成的。
* **系统：**将错误处理委托给绑定程序（重新排队，DL和其他）。注意，这些技术取决于绑定程序的实现和底层消息传递中间件的功能。

Spring Cloud Stream使用[Spring重试](https://github.com/spring-projects/spring-retry" \t "https://www.springcloud.cc/_top)库来促进成功的消息处理。有关更多详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_template" \o "29.4.3重试模板)请参见[第29.4.3节“重试模板”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_template" \o "29.4.3 Retry Template)。但是，当所有方法均失败时，消息处理程序引发的异常将传播回绑定程序。那时，活页夹调用自定义错误处理程序或将错误传达回消息传递系统（重新排队，DLQ等）。

### **29.4.1应用程序错误处理**

有两种类型的应用程序级错误处理。可以在每个绑定订阅中处理错误，或者全局处理程序可以处理所有绑定订阅错误。让我们查看详细信息。

**图29.1 具有自定义和全局错误处理程序的Spring Cloud Stream接收器应用程序**



对于每个输入绑定，Spring Cloud Stream创建具有以下语义<destinationName>.errors的专用错误通道。

|  |
| --- |
| IMG_377 |
| <destinationName>由绑定的名称（例如input）和组的名称（例如myGroup）组成。 |

考虑以下：

spring.cloud.stream.bindings.input.group=myGroup

@StreamListener(Sink.INPUT) *// destination name 'input.myGroup'***public** **void** handle(Person value) {

**throw** **new** RuntimeException("BOOM!");

}

@ServiceActivator(inputChannel = Processor.INPUT + ".myGroup.errors") *//channel name 'input.myGroup.errors'***public** **void** error(Message<?> message) {

System.out.println("Handling ERROR: " + message);

}

在前面的示例中，目标名称为input.myGroup，专用错误通道名称为input.myGroup.errors。

|  |
| --- |
| IMG_378 |
| @StreamListener批注的使用专门用于定义桥接内部通道和外部目标的绑定。假设目标特定错误通道没有关联的外部目标，则该通道是Spring Integration（SI）的特权。这意味着必须使用SI处理程序注释之一（即@ ServiceActivator，@ Transformer等）定义用于此类目标的处理程序。 |

|  |
| --- |
| IMG_379 |
| 如果未指定group，则使用匿名组（类似于input.anonymous.2K37rb06Q6m2r51-SPIDDQ），这不适用于错误处理方案，因为在创建目标之前，您不知道它将是什么。 |

另外，如果您绑定到现有目的地，例如：

spring.cloud.stream.bindings.input.destination=myFooDestination

spring.cloud.stream.bindings.input.group=myGroup

完整的目标名称为myFooDestination.myGroup，然后专用错误通道名称为myFooDestination.myGroup.errors。

回到例子...

预订名为input的通道的handle(..)方法会引发异常。给定错误通道input.myGroup.errors的订阅者，所有错误消息均由该订阅者处理。

如果您有多个绑定，则可能需要一个错误处理程序。Spring Cloud Stream 通过将每个单独的错误通道桥接到名为errorChannel的通道来自动提供对全局错误通道的支持，从而允许单个订阅者处理所有错误，如以下示例所示：

@StreamListener("errorChannel")**public** **void** error(Message<?> message) {

System.out.println("Handling ERROR: " + message);

}

如果错误处理逻辑相同，则与哪个处理程序产生错误无关，这可能是一个方便的选择。

### **29.4.2系统错误处理**

系统级错误处理意味着将错误传递回消息传递系统，并且鉴于并非每个消息传递系统都相同，因此各个粘合剂的功能可能有所不同。

也就是说，在本节中，我们解释了系统级错误处理背后的一般思想，并以Rabbit活页夹为例。注意：Kafka活页夹提供了类似的支持，尽管某些配置属性确实有所不同。另外，有关更多详细信息和配置选项，请参见各个活页夹的文档。

如果未配置内部错误处理程序，则错误将传播到绑定程序，而绑定程序随后会将这些错误传播回消息传递系统。根据消息传递系统的功能，此类系统可能会丢弃该消息，重新排队该消息以进行重新处理或将失败的消息发送给DLQ。Rabbit和Kafka都支持这些概念。但是，其他联编程序可能没有，因此请参阅您单独的联编程序的文档，以获取有关受支持的系统级错误处理选项的详细信息。

#### **删除失败的消息**

默认情况下，如果未提供其他系统级配置，则消息传递系统将丢弃失败的消息。尽管在某些情况下可以接受，但在大多数情况下是不可接受的，我们需要一些恢复机制来避免消息丢失。

#### **DLQ-死信队列**

DLQ允许将失败的消息发送到特殊目标：-Dead Letter Queue。

配置后，失败的消息将发送到此目标，以进行后续的重新处理或审核与对帐。

例如，继续前面的示例，并使用Rabbit活页夹设置DLQ，您需要设置以下属性：

spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true

请记住，在以上属性中，input对应于输入目标绑定的名称。consumer指示它是消费者属性，auto-bind-dlq指示绑定程序为input目标配置DLQ，这将导致名为input.myGroup.dlq的附加Rabbit队列。

配置完成后，所有失败的消息都会通过错误消息路由到此队列，类似于以下内容：

delivery\_mode: 1

headers:

x-death:

count: 1

reason: rejected

queue: input.hello

time: 1522328151

exchange:

routing-keys: input.myGroup

Payload {"name”:"Bob"}

从上面可以看到，原始消息会保留下来以供进一步操作。

但是，您可能已经注意到的一件事是，有关消息处理的原始问题的信息有限。例如，您看不到与原始错误相对应的堆栈跟踪。要获取有关原始错误的更多相关信息，您必须设置一个附加属性：

spring.cloud.stream.rabbit.bindings.input.consumer.republish-to-dlq=true

这样做会强制内部错误处理程序在将错误消息发布到DLQ之前拦截该错误消息并向其添加其他信息。配置完成后，您会看到错误消息包含与原始错误有关的更多信息，如下所示：

delivery\_mode: 2

headers:

x-original-exchange:

x-exception-message: has an error

x-original-routingKey: input.myGroup

x-exception-stacktrace: org.springframework.messaging.MessageHandlingException: nested exception is

org.springframework.messaging.MessagingException: has an error, failedMessage=GenericMessage [payload=byte[15],

headers={amqp\_receivedDeliveryMode=NON\_PERSISTENT, amqp\_receivedRoutingKey=input.hello, amqp\_deliveryTag=1,

deliveryAttempt=3, amqp\_consumerQueue=input.hello, amqp\_redelivered=false, id=a15231e6-3f80-677b-5ad7-d4b1e61e486e,

amqp\_consumerTag=amq.ctag-skBFapilvtZhDsn0k3ZmQg, contentType=application/json, timestamp=1522327846136}]

at org.spring...integ...han...MethodInvokingMessageProcessor.processMessage(MethodInvokingMessageProcessor.java:107)

at. . . . .

Payload {"name”:"Bob"}

这有效地结合了应用程序级和系统级的错误处理，以进一步协助下游故障排除机制。

#### **重新排队失败的消息**

如前所述，当前支持的活页夹（Rabbit和Kafka）依靠RetryTemplate来促进成功的消息处理。有关详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_template" \o "29.4.3重试模板)请参见[第29.4.3节“重试模板”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_retry_template" \o "29.4.3 Retry Template)。但是，对于max-attempts属性设置为1的情况，将禁用消息的内部重新处理。此时，您可以通过指示消息传递系统重新排队失败的消息来促进消息的重新处理（重试）。重新排队后，失败的消息将被发送回原始处理程序，从而创建一个重试循环。

如果错误的性质与某些资源的偶发性但短期不可用有关，则此选项可能是可行的。

为此，必须设置以下属性：

spring.cloud.stream.bindings.input.consumer.max-attempts=1

spring.cloud.stream.rabbit.bindings.input.consumer.requeue-rejected=true

在前面的示例中，max-attempts设置为1，实际上禁用了内部重试，而requeue-rejected（重新排队拒绝消息的缩写）被设置为true。设置后，失败的消息将重新提交给同一处理程序并连续循环，直到处理程序抛出AmqpRejectAndDontRequeueException为止，从本质上讲，您可以在处理程序本身内构建自己的重试逻辑。

### **29.4.3重试模板**

RetryTemplate是[Spring重试](https://github.com/spring-projects/spring-retry" \t "https://www.springcloud.cc/_top)库的一部分。尽管涵盖RetryTemplate的所有功能超出了本文档的范围，但我们将提及以下与RetryTemplate特别相关的使用者属性：

maxAttempts

处理消息的尝试次数。

默认值：3。

backOffInitialInterval

重试时的退避初始间隔。

默认值1000毫秒。

backOffMaxInterval

最大退避间隔。

默认值10000毫秒。

backOffMultiplier

退避乘数。

默认为2.0。

defaultRetryable

retryableExceptions中未列出的由侦听器引发的异常是否可以重试。

默认值：true。

retryableExceptions

键中Throwable类名称的映射，值中布尔值的映射。指定将要重试的那些异常（和子类）。另请参见defaultRetriable。示例：spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false。

默认值：空。

尽管上述设置足以满足大多数自定义要求，但它们可能无法满足某些复杂的要求，此时，您可能希望提供自己的RetryTemplate实例。为此，在应用程序配置中将其配置为bean。应用程序提供的实例将覆盖框架提供的实例。另外，为避免冲突，必须将绑定程序要使用的RetryTemplate实例限定为@StreamRetryTemplate。例如，

@StreamRetryTemplate**public** RetryTemplate myRetryTemplate() {

**return** **new** RetryTemplate();

}

从上面的示例中可以看到，由于@StreamRetryTemplate是合格的@Bean，因此无需使用@Bean对其进行注释。

## **29.5反应式编程支持**

Spring Cloud Stream还支持使用反应式API，将传入和传出的数据作为连续的数据流进行处理。可通过spring-cloud-stream-reactive获得对反应式API的支持，需要将其显式添加到您的项目中。

具有响应式API的编程模型是声明性的。您可以使用描述从入站数据流到出站数据流的功能转换的运算符，而不是指定每个消息的处理方式。

目前Spring Cloud Stream仅支持[Reactor API](https://projectreactor.io/" \t "https://www.springcloud.cc/_top)。将来，我们打算支持基于反应式流的更通用的模型。

反应式编程模型还使用@StreamListener注释来设置反应式处理程序。区别在于：

* @StreamListener批注不能指定输入或输出，因为它们作为参数提供并从方法返回值。
* 该方法的参数必须用@Input和@Output注释，分别指示传入和传出数据流连接到哪个输入或输出。
* 该方法的返回值（如果有）用@Output注释，指示应该将数据发送到的输入。

|  |
| --- |
| IMG_380 |
| 响应式编程支持需要Java 1.8。 |

|  |
| --- |
| IMG_381 |
| 从Spring Cloud Stream 1.1.1起（从发行版Brooklyn.SR2开始），反应式编程支持要求使用Reactor 3.0.4.RELEASE及更高版本。不支持更早的Reactor版本（包括3.0.1.RELEASE，3.0.2.RELEASE和3.0.3.RELEASE）。spring-cloud-stream-reactive可传递地检索正确的版本，但是项目结构可以将io.projectreactor:reactor-core的版本管理为早期版本，尤其是在使用Maven时。使用Spring Initializr和Spring Boot 1.x生成的项目就是这种情况，该项目将Reactor版本覆盖为2.0.8.RELEASE。在这种情况下，必须确保发布了正确版本的工件。您可以通过在项目中添加对版本为3.0.4.RELEASE或更高版本的io.projectreactor:reactor-core的直接依赖来实现。 |

|  |
| --- |
| IMG_382 |
| 当前，术语“ 反应式 ”的使用是指正在使用的反应式API，而不是指执行模型是反应式的（也就是说，绑定的端点仍然使用“推”式而非“拉式”模型）。尽管通过使用Reactor提供了一些反压支持，但在将来的发行版中，我们确实打算通过对连接的中间件使用本机反应性客户端来完全支持反应性管道。 |

### **29.5.1基于Reactor的处理程序**

基于Reactor的处理程序可以具有以下参数类型：

* 对于带有@Input注释的参数，它支持Reactor Flux类型。入站Flux的参数化遵循与处理单个消息时相同的规则：可以是整个Message，可以是Message有效负载的POJO或由于以下原因而产生的POJO：基于Message内容类型标头的转换。提供了多个输入。
* 对于带有Output注释的参数，它支持FluxSender类型，该类型将方法生成的Flux与输出连接起来。一般而言，仅在该方法可以具有多个输出时才建议将输出指定为参数。

基于Reactor的处理程序支持Flux的返回类型。在这种情况下，必须用@Output进行注释。当单个输出Flux可用时，建议使用该方法的返回值。

以下示例显示了基于Reactor的Processor：

@EnableBinding(Processor.class)@EnableAutoConfiguration**public** **static** **class** UppercaseTransformer {

@StreamListener

@Output(Processor.OUTPUT)

**public** Flux<String> receive(@Input(Processor.INPUT) Flux<String> input) {

**return** input.map(s -> s.toUpperCase());

}

}

使用输出参数的同一处理器看起来像以下示例：

@EnableBinding(Processor.class)@EnableAutoConfiguration**public** **static** **class** UppercaseTransformer {

@StreamListener

**public** **void** receive(@Input(Processor.INPUT) Flux<String> input,

@Output(Processor.OUTPUT) FluxSender output) {

output.send(input.map(s -> s.toUpperCase()));

}

}

### **29.5.2反应源**

Spring Cloud Stream反应性支持还提供了通过@StreamEmitter注释创建反应性源的功能。通过使用@StreamEmitter批注，可以将常规源转换为被动源。@StreamEmitter是方法级别的注释，用于将方法标记为用@EnableBinding声明的输出的发射器。您不能将@Input批注与@StreamEmitter一起使用，因为标有该批注的方法不会监听任何输入。而是用标记为@StreamEmitter的方法生成输出。遵循@StreamListener中使用的相同编程模型，@StreamEmitter还允许灵活地使用@Output批注，具体取决于方法是否具有任何参数，返回类型和其他考虑因素。

本节的其余部分包含使用各种样式的@StreamEmitter批注的示例。

以下示例每毫秒发出一次Hello, World消息，并发布到Reactor Flux中：

@EnableBinding(Source.class)@EnableAutoConfiguration**public** **static** **class** HelloWorldEmitter {

@StreamEmitter

@Output(Source.OUTPUT)

**public** Flux<String> emit() {

**return** Flux.intervalMillis(1)

.map(l -> "Hello World");

}

}

在前面的示例中，Flux中的结果消息被发送到Source的输出通道。

下一个示例是@StreamEmmitter的另一种形式，它发送Reactor Flux。以下方法代替返回Flux，而是使用FluxSender从源代码中以编程方式发送Flux：

@EnableBinding(Source.class)@EnableAutoConfiguration**public** **static** **class** HelloWorldEmitter {

@StreamEmitter

@Output(Source.OUTPUT)

**public** **void** emit(FluxSender output) {

output.send(Flux.intervalMillis(1)

.map(l -> "Hello World"));

}

}

下一个示例在功能和样式上与上述代码段完全相同。但是，它没有在方法上使用显式的@Output注释，而是在方法参数上使用了注释。

@EnableBinding(Source.class)@EnableAutoConfiguration**public** **static** **class** HelloWorldEmitter {

@StreamEmitter

**public** **void** emit(@Output(Source.OUTPUT) FluxSender output) {

output.send(Flux.intervalMillis(1)

.map(l -> "Hello World"));

}

}

本节的最后一个示例是使用Reactive Streams Publisher API并利用[Spring Integration Java DSL中](https://github.com/spring-projects/spring-integration-java-dsl/wiki/Spring-Integration-Java-DSL-Reference" \t "https://www.springcloud.cc/_top)对它的支持来编写反应源的另一种方式。以下示例中的Publisher仍在幕后使用Reactor Flux，但是，从应用程序角度看，这对用户是透明的，并且对于Spring Integration仅需要响应流和Java DSL：

@EnableBinding(Source.class)@EnableAutoConfiguration**public** **static** **class** HelloWorldEmitter {

@StreamEmitter

@Output(Source.OUTPUT)

@Bean

**public** Publisher<Message<String>> emit() {

**return** IntegrationFlows.from(() ->

**new** GenericMessage<>("Hello World"),

e -> e.poller(p -> p.fixedDelay(1)))

.toReactivePublisher();

}

}

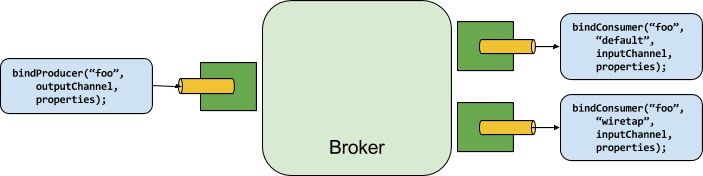
## **30. Binders**

Spring Cloud Stream提供了Binder抽象，用于连接到外部中间件上的物理目标。本节提供有关Binder SPI背后的主要概念，其主要组件以及特定于实现的详细信息。

## **30.1生产者和消费者**

下图显示了生产者和消费者的一般关系：

**图30.1。生产者和消费者**



生产者是将消息发送到通道的任何组件。可以将该通道绑定到具有该代理的Binder实现的外部消息代理。调用bindProducer()方法时，第一个参数是代理内目标的名称，第二个参数是生产者向其发送消息的本地通道实例，第三个参数包含属性（例如分区键表达式） ），以在为该通道创建的适配器中使用。

使用者是从通道接收消息的任何组件。与生产者一样，消费者的渠道可以绑定到外部消息代理。调用bindConsumer()方法时，第一个参数是目标名称，第二个参数提供逻辑消费者组的名称。由给定目标的使用者绑定表示的每个组都接收生产者发送到该目标的每个消息的副本（也就是说，它遵循常规的发布-订阅语义）。如果有多个使用相同组名绑定的使用者实例，那么消息将在这些使用者实例之间进行负载平衡，以便由生产者发送的每条消息仅在每个组内的单个使用者实例中被使用（也就是说，它遵循常规排队语义）。

## **30.2 Binder SPI**

Binder SPI由许多接口，现成的实用程序类和发现策略组成，这些策略提供了用于连接到外部中间件的可插拔机制。

SPI的关键是Binder接口，这是将输入和输出连接到外部中间件的策略。以下清单显示了Binder接口的定义：

**public** **interface** Binder<T, C **extends** ConsumerProperties, P **extends** ProducerProperties> {

Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);

}

该接口已参数化，提供了许多扩展点：

* 输入和输出绑定目标。从1.0版开始，仅支持MessageChannel，但将来打算将其用作扩展点。
* 扩展的使用者和生产者属性，允许特定的Binder实现添加可以以类型安全的方式支持的补充属性。

典型的活页夹实现包括以下内容：

* 实现Binder接口的类；
* 一个Spring @Configuration类，它创建类型为Binder的bean和中间件连接基础结构。

在类路径上找到一个META-INF/spring.binders文件，其中包含一个或多个绑定程序定义，如以下示例所示：

kafka:\

org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration

## **30.3 Binder检测**

Spring Cloud Stream依赖于Binder SPI的实现来执行将通道连接到消息代理的任务。每个Binder实现通常都连接到一种消息传递系统。

### **30.3.1类路径检测**

默认情况下，Spring Cloud Stream依靠Spring Boot的自动配置来配置绑定过程。如果在类路径上找到单个Binder实现，则Spring Cloud Stream将自动使用它。例如，旨在仅绑定到RabbitMQ的Spring Cloud Stream项目可以添加以下依赖项：

<dependency>

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

<artifactId>spring-cloud-stream-binder-rabbit</artifactId></dependency>

有关其他绑定程序依赖项的特定Maven坐标，请参阅该绑定程序实现的文档。

## **类路径上的30.4倍数Binders**

当类路径上存在多个绑定程序时，应用程序必须指示将哪个绑定程序用于每个通道绑定。每个活页夹配置都包含一个META-INF/spring.binders文件，它是一个简单的属性文件，如以下示例所示：

rabbit:\

org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration

其他提供的活页夹实现（例如Kafka）也存在类似的文件，并且期望自定义活页夹实现也将提供它们。关键字表示绑定程序实现的标识名，而该值是逗号分隔的配置类列表，每个配置类都包含一个且仅一个bean类型为org.springframework.cloud.stream.binder.Binder的定义。

可以使用spring.cloud.stream.defaultBinder属性（例如，spring.cloud.stream.defaultBinder=rabbit）在全局上执行Binder选择，也可以通过在每个通道绑定上配置活页夹来分别进行Binder选择。例如，从Kafka读取并写入RabbitMQ的处理器应用程序（具有分别名为input和output的通道用于读取和写入）可以指定以下配置：

spring.cloud.stream.bindings.input.binder=kafka

spring.cloud.stream.bindings.output.binder=rabbit

## **30.5连接到多个系统**

默认情况下，活页夹共享应用程序的Spring Boot自动配置，以便创建在类路径上找到的每个活页夹的一个实例。如果您的应用程序应连接到多个相同类型的代理，则可以指定多个绑定程序配置，每个配置具有不同的环境设置。

|  |
| --- |
| IMG_384 |
| 启用显式绑定程序配置将完全禁用默认的绑定程序配置过程。如果这样做，则配置中必须包括所有正在使用的活页夹。打算透明使用Spring Cloud Stream的框架可以创建可以按名称引用的活页夹配置，但它们不会影响默认的活页夹配置。为此，活页夹配置可以将其defaultCandidate标志设置为false（例如spring.cloud.stream.binders.<configurationName>.defaultCandidate=false）。这表示独立于默认绑定程序配置过程而存在的配置。 |

以下示例显示了连接到两个RabbitMQ代理实例的处理器应用程序的典型配置：

spring:

cloud:

stream:

bindings:

input:

destination: thing1

binder: rabbit1

output:

destination: thing2

binder: rabbit2

binders:

rabbit1:

type: rabbit

environment:

spring:

rabbitmq:

host: <host1>

rabbit2:

type: rabbit

environment:

spring:

rabbitmq:

host: <host2>

## **30.6绑定可视化和控制**

从2.0版开始，Spring Cloud Stream支持通过Actuator端点进行绑定的可视化和控制。

从2.0版执行器开始，并且web是可选的，您必须首先添加web依赖项之一，然后手动添加执行器依赖项。以下示例说明如何为Web框架添加依赖项：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId></dependency>

以下示例显示如何为WebFlux框架添加依赖项：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId></dependency>

您可以添加执行器依赖项，如下所示：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId></dependency>

|  |
| --- |
| IMG_385 |
| 要在Cloud Foundry中运行Spring Cloud Stream 2.0应用程序，必须将spring-boot-starter-web和spring-boot-starter-actuator添加到类路径中。否则，由于运行状况检查失败，该应用程序将无法启动。 |

您还必须通过设置以下属性来启用bindings执行器端点：--management.endpoints.web.exposure.include=bindings。

一旦满足这些先决条件。应用程序启动时，您应该在日志中看到以下内容：

: Mapped "{[/actuator/bindings/{name}],methods=[POST]. . .

: Mapped "{[/actuator/bindings],methods=[GET]. . .

: Mapped "{[/actuator/bindings/{name}],methods=[GET]. . .

要显示当前绑定，请访问以下URL：[http://<host>:<port>/actuator/bindings](http://<host>:<port>/actuator/bindings" \t "https://www.springcloud.cc/_top)

或者，要查看单个绑定，请访问类似于以下内容的URL之一：[http://<host>:<port>/actuator/bindings/myBindingName](http://<host>:<port>/actuator/bindings/myBindingName" \t "https://www.springcloud.cc/_top)

您还可以通过发布到相同的URL来停止，开始，暂停和恢复单个绑定，同时提供一个state作为JSON的参数，如以下示例所示：

curl -d'{“ state”：“ STOPPED”}'-H“内容类型：应用程序/ json” -X POST [http：// <主机>：<port> / actuator / bindings / myBindingName](http://<host>:<port>/actuator/bindings/myBindingName" \t "https://www.springcloud.cc/_top) curl -d'{ “ state”：“ STARTED”}'-H“内容类型：application / json” -X POST [http：// <host>：<port> / actuator / bindings / myBindingName](http://<host>:<port>/actuator/bindings/myBindingName" \t "https://www.springcloud.cc/_top) curl -d'{“ state”：“ PAUSED“}'-H”内容类型：application / json“ -X POST [http：// <host>：<port> / actuator / bindings / myBindingName](http://<host>:<port>/actuator/bindings/myBindingName" \t "https://www.springcloud.cc/_top) curl -d'{” state“：” RESUMED“}'- H“内容类型：应用程序/ json” -X POST [http：// <主机>：<端口> / actuator / bindings / myBindingName](http://<host>:<port>/actuator/bindings/myBindingName" \t "https://www.springcloud.cc/_top)

|  |
| --- |
| IMG_386 |
| PAUSED和RESUMED仅在相应的活页夹及其基础技术支持时起作用。否则，您会在日志中看到警告消息。当前，只有Kafka活页夹支持PAUSED和RESUMED状态。 |

## **30.7 Binder配置Properties**

定制活页夹配置时，以下属性可用。这些属性通过org.springframework.cloud.stream.config.BinderProperties公开

它们必须以spring.cloud.stream.binders.<configurationName>为前缀。

类型

资料夹类型。它通常引用在类路径上找到的绑定器之一-特别是META-INF/spring.binders文件中的键。

默认情况下，它具有与配置名称相同的值。

继承环境

配置是否继承应用程序本身的环境。

默认值：true。

环境

根可用于定制活页夹环境的一组属性。设置此属性后，在其中创建活页夹的上下文不是应用程序上下文的子级。该设置允许在粘合剂组分和应用组分之间完全分离。

默认值：empty。

defaultCandidate

活页夹配置是被视为默认活页夹的候选者还是仅在明确引用时才可以使用。此设置允许添加活页夹配置，而不会干扰默认处理。

默认值：true。

## **31.配置选项**

Spring Cloud Stream支持常规配置选项以及绑定和活页夹的配置。一些活页夹使附加的绑定属性支持特定于中间件的功能。

可以通过Spring Boot支持的任何机制向Spring Cloud Stream应用程序提供配置选项。这包括应用程序参数，环境变量以及YAML或.properties文件。

## **31.1绑定服务Properties**

这些属性通过org.springframework.cloud.stream.config.BindingServiceProperties公开

spring.cloud.stream.instanceCount

应用程序已部署实例的数量。必须在生产者端进行分区设置。使用RabbitMQ时必须在用户端设置，如果使用autoRebalanceEnabled=false，则必须在Kafka时设置。

默认值：1。

spring.cloud.stream.instanceIndex

应用程序的实例索引：从0到instanceCount - 1的数字。用于通过RabbitMQ和Kafka（如果是autoRebalanceEnabled=false）进行分区。在Cloud Foundry中自动设置以匹配应用程序的实例索引。

spring.cloud.stream.dynamic目的地

可以动态绑定的目的地列表（例如，在动态路由方案中）。如果设置，则只能绑定列出的目的地。

默认值：空（将任何目的地绑定）。

spring.cloud.stream.defaultBinder

如果配置了多个联编程序，则使用的默认联编程序。请参见[Classpath上的多个Binders](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-binders" \o "30.4类路径上的多个绑定器)。

默认值：空。

spring.cloud.stream.overrideCloudConnectors

仅当cloud配置文件处于活动状态并且应用程序提供了Spring Cloud Connectors时，此属性才适用。如果属性为false（默认值），则绑定器检测到合适的绑定服务（例如，RabbitMQ绑定器在Cloud Foundry中绑定的RabbitMQ服务）并将其用于创建连接（通常通过Spring Cloud Connectors）。设置为true时，此属性指示绑定程序完全忽略绑定的服务，并依赖Spring Boot属性（例如，依赖于环境中为RabbitMQ绑定程序提供的spring.rabbitmq.\*属性） 。[连接到多个系统时](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-systems" \o "30.5连接到多个系统)，此属性的典型用法是嵌套在自定义环境[中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-systems" \o "30.5 Connecting to Multiple Systems)。

默认值：false。

spring.cloud.stream.bindingRetryInterval

例如，活页夹不支持后期绑定和代理（例如Apache Kafka）关闭时，重试绑定创建之间的间隔（以秒为单位）。将该值设置为零可将此类情况视为致命情况，从而阻止应用程序启动。

默认值：30

## **31.2绑定Properties**

绑定属性是使用spring.cloud.stream.bindings.<channelName>.<property>=<value>格式提供的。<channelName>代表正在配置的通道的名称（例如，对于Source，为output）。

为避免重复，Spring Cloud Stream支持所有通道的设置值，格式为spring.cloud.stream.default.<property>=<value>。

在避免重复使用扩展绑定属性时，应使用此格式-spring.cloud.stream.<binder-type>.default.<producer|consumer>.<property>=<value>。

在下面的内容中，我们指出了省略了spring.cloud.stream.bindings.<channelName>.前缀的位置，仅着眼于属性名称，但要了解前缀是在运行时包含的。

### **31.2.1通用绑定Properties**

这些属性通过org.springframework.cloud.stream.config.BindingProperties公开

以下绑定属性可用于输入和输出绑定，并且必须以spring.cloud.stream.bindings.<channelName>.为前缀（例如，spring.cloud.stream.bindings.input.destination=ticktock）。

可以使用前缀spring.cloud.stream.default设置默认值（例如“ spring.cloud.stream.default.contentType = application / json”）。

目的地

绑定的中间件上的通道的目标目的地（例如，RabbitMQ交换或Kafka主题）。如果将通道绑定为使用者，则可以将其绑定到多个目标，并且目标名称可以指定为逗号分隔的String值。如果未设置，则使用通道名称。此属性的默认值不能被覆盖。

组

渠道的消费群体。仅适用于入站绑定。请参阅[消费者组](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "consumer-groups" \o "28.4消费群体)。

默认值：null（指示匿名使用者）。

内容类型

频道的内容类型。请参见“ [第32章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)*[内容类型协商](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "content-type-management" \o "32.内容类型协商)* ”。

默认值：application/json。

黏合剂

此绑定使用的粘合剂。有关详细信息，请参见“ [第30.4节“类路径上的多个Binders](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "multiple-binders" \o "30.4类路径上的多个绑定器) ”）。

默认值：null（如果存在默认活页夹，则使用默认活页夹）。

### **31.2.2消费者Properties**

这些属性通过org.springframework.cloud.stream.binder.ConsumerProperties公开

以下绑定属性仅可用于输入绑定，并且必须以spring.cloud.stream.bindings.<channelName>.consumer.为前缀（例如，spring.cloud.stream.bindings.input.consumer.concurrency=3）。

可以使用spring.cloud.stream.default.consumer前缀（例如，spring.cloud.stream.default.consumer.headerMode=none）设置默认值。

并发

入站使用者的并发。

默认值：1。

分区的

消费者是否从分区生产者那里接收数据。

默认值：false。

headerMode

设置为none时，禁用输入的标头解析。仅对本身不支持消息头并且需要消息头嵌入的消息中间件有效。当不支持本机头时，使用来自非Spring Cloud Stream应用程序的数据时，此选项很有用。设置为headers时，它使用中间件的本机头机制。设置为embeddedHeaders时，它将标头嵌入到消息有效负载中。

默认值：取决于活页夹的实现。

maxAttempts

如果处理失败，则尝试处理消息的次数（包括第一次）。设置为1以禁用重试。

默认值：3。

backOffInitialInterval

重试时的退避初始间隔。

默认值：1000。

backOffMaxInterval

最大退避间隔。

默认值：10000。

backOffMultiplier

退避乘数。

默认值：2.0。

defaultRetryable

retryableExceptions中未列出的由侦听器引发的异常是否可以重试。

默认值：true。

instanceIndex

设置为大于零的值时，它允许自定义此使用者的实例索引（如果与spring.cloud.stream.instanceIndex不同）。设置为负值时，默认为spring.cloud.stream.instanceIndex。有关更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count" \o "34.2 Instance Index and Instance Count)请参见“ [第34.2节“实例索引和实例计数”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count" \o "34.2实例索引和实例计数) ”。

默认值：-1。

instanceCount

设置为大于零的值时，它允许自定义此使用者的实例计数（如果与spring.cloud.stream.instanceCount不同）。设置为负值时，默认为spring.cloud.stream.instanceCount。有关更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count" \o "34.2 Instance Index and Instance Count)请参见“ [第34.2节“实例索引和实例计数”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count" \o "34.2实例索引和实例计数) ”。

默认值：-1。

retryableExceptions

键中Throwable类名称的映射，值中布尔值的映射。指定将要重试的那些异常（和子类）。另请参见defaultRetriable。示例：spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false。

默认值：空。

useNativeDecoding

设置为true时，入站消息将直接由客户端库反序列化，该库必须进行相应配置（例如，设置适当的Kafka生产者值反序列化器）。使用此配置时，入站消息解组不是基于绑定的contentType。使用本机解码时，生产者负责使用适当的编码器（例如，Kafka生产者值序列化程序）对出站消息进行序列化。同样，当使用本机编码和解码时，headerMode=embeddedHeaders属性将被忽略，并且标头不会嵌入消息中。请参见生产者属性useNativeEncoding。

默认值：false。

### **31.2.3生产者Properties**

这些属性通过org.springframework.cloud.stream.binder.ProducerProperties公开

以下绑定属性仅可用于输出绑定，并且必须以spring.cloud.stream.bindings.<channelName>.producer.为前缀（例如，spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id）。

可以使用前缀spring.cloud.stream.default.producer（例如，spring.cloud.stream.default.producer.partitionKeyExpression=payload.id）设置默认值。

partitionKeyExpression

一个SpEL表达式，用于确定如何对出站数据进行分区。如果已设置或设置了partitionKeyExtractorClass，则会对该通道上的出站数据进行分区。partitionCount必须设置为大于1的值才能生效。与partitionKeyExtractorClass互斥。请参见“ [第28.6节“分区支持”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "partitioning" \o "28.6分区支持) ”。

默认值：null。

partitionKeyExtractorClass

PartitionKeyExtractorStrategy实现。如果已设置，或者已设置partitionKeyExpression，则会对该通道上的出站数据进行分区。partitionCount必须设置为大于1的值才能生效。与partitionKeyExpression互斥。请参见“ [第28.6节“分区支持”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "partitioning" \o "28.6分区支持) ”。

默认值：null。

partitionSelectorClass

PartitionSelectorStrategy实现。与partitionSelectorExpression互斥。如果两者均未设置，则将该分区选择为hashCode(key) % partitionCount，其中key通过partitionKeyExpression或partitionKeyExtractorClass计算。

默认值：null。

partitionSelectorExpression

用于自定义分区选择的SpEL表达式。与partitionSelectorClass互斥。如果两者均未设置，则将分区选择为hashCode(key) % partitionCount，其中key通过partitionKeyExpression或partitionKeyExtractorClass计算。

默认值：null。

partitionCount

数据的目标分区数（如果启用了分区）。如果生产者已分区，则必须将其设置为大于1的值。在Kafka上，它被解释为提示。取其较大者，并使用目标主题的分区数。

默认值：1。

requiredGroups

生产者必须确保将消息传递到的组的逗号分隔列表，即使它们是在创建消息之后开始的（例如，通过在RabbitMQ中预先创建持久队列）。

headerMode

设置为none时，它将禁用在输出中嵌入标头。它仅对本身不支持消息头并且需要消息头嵌入的消息中间件有效。当不支持本机头时，为非Spring Cloud Stream应用程序生成数据时，此选项很有用。设置为headers时，它使用中间件的本机头机制。设置为embeddedHeaders时，它将标头嵌入到消息有效负载中。

默认值：取决于活页夹的实现。

useNativeEncoding

设置为true时，出站消息将直接由客户端库进行序列化，该库必须进行相应配置（例如，设置适当的Kafka生产者值序列化程序）。使用此配置时，出站消息编组不是基于绑定的contentType。使用本机编码时，使用方负责使用适当的解码器（例如，Kafka使用方值反序列化器）对入站消息进行反序列化。此外，当使用本机编码和解码时，headerMode=embeddedHeaders属性将被忽略，并且标头不会嵌入消息中。请参阅消费者属性useNativeDecoding。

默认值：false。

errorChannelEnabled

设置为true时，如果活页夹支持异步发送结果，则发送失败将发送到目标的错误通道。参见“ [???](https://www.springcloud.cc/spring-cloud-greenwich.html)”以获取更多信息。

默认值：false。

## **31.3使用动态绑定的目的地**

除了使用@EnableBinding定义的通道外，Spring Cloud Stream还允许应用程序将消息发送到动态绑定的目的地。例如，当需要在运行时确定目标目的地时，这很有用。应用程序可以通过使用@EnableBinding注释自动注册的BinderAwareChannelResolver bean来实现。

“ spring.cloud.stream.dynamicDestinations”属性可用于将动态目标名称限制为已知集合（白名单）。如果未设置此属性，则可以动态绑定任何目标。

BinderAwareChannelResolver可以直接使用，如以下使用路径变量来确定目标通道的REST控制器示例所示：

@EnableBinding@Controller**public** **class** SourceWithDynamicDestination {

@Autowired

**private** BinderAwareChannelResolver resolver;

@RequestMapping(path = "/{target}", method = POST, consumes = "\*/\*")

@ResponseStatus(HttpStatus.ACCEPTED)

**public** **void** handleRequest(@RequestBody String body, @PathVariable("target") target,

@RequestHeader(HttpHeaders.CONTENT\_TYPE) Object contentType) {

sendMessage(body, target, contentType);

}

**private** **void** sendMessage(String body, String target, Object contentType) {

resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,

**new** MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT\_TYPE, contentType))));

}

}

现在考虑当我们在默认端口（8080）上启动应用程序并使用CURL发出以下请求时会发生什么：

curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders

在经纪人中创建目的地“客户”和“订单”（交换为Rabbit或在主题为“ Kafka”中），名称为“客户”和“订单”，数据为发布到适当的目的地。

BinderAwareChannelResolver是通用的Spring Integration DestinationResolver，并且可以注入到其他组件中，例如，在路由器中使用基于传入的target字段的SpEL表达式的路由器JSON消息。以下示例包含一个读取SpEL表达式的路由器：

@EnableBinding@Controller**public** **class** SourceWithDynamicDestination {

@Autowired

**private** BinderAwareChannelResolver resolver;

@RequestMapping(path = "/", method = POST, consumes = "application/json")

@ResponseStatus(HttpStatus.ACCEPTED)

**public** **void** handleRequest(@RequestBody String body, @RequestHeader(HttpHeaders.CONTENT\_TYPE) Object contentType) {

sendMessage(body, contentType);

}

**private** **void** sendMessage(Object body, Object contentType) {

routerChannel().send(MessageBuilder.createMessage(body,

**new** MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT\_TYPE, contentType))));

}

@Bean(name = "routerChannel")

**public** MessageChannel routerChannel() {

**return** **new** DirectChannel();

}

@Bean

@ServiceActivator(inputChannel = "routerChannel")

**public** ExpressionEvaluatingRouter router() {

ExpressionEvaluatingRouter router =

**new** ExpressionEvaluatingRouter(**new** SpelExpressionParser().parseExpression("payload.target"));

router.setDefaultOutputChannelName("default-output");

router.setChannelResolver(resolver);

**return** router;

}

}

该[路由器接收器应用程序](https://github.com/spring-cloud-stream-app-starters/router" \t "https://www.springcloud.cc/_top)使用此技术的按需创建的目的地。

如果预先知道通道名称，则可以像其他任何目的地一样配置生产者属性。或者，如果您注册NewBindingCallback<> bean，则会在创建绑定之前调用它。回调采用绑定程序使用的扩展生产者属性的通用类型。它有一种方法：

**void** configure(String channelName, MessageChannel channel, ProducerProperties producerProperties,

T extendedProducerProperties);

下面的示例显示如何使用RabbitMQ活页夹：

@Bean**public** NewBindingCallback<RabbitProducerProperties> dynamicConfigurer() {

**return** (name, channel, props, extended) -> {

props.setRequiredGroups("bindThisQueue");

extended.setQueueNameGroupOnly(true);

extended.setAutoBindDlq(true);

extended.setDeadLetterQueueName("myDLQ");

};

}

|  |
| --- |
| IMG_387 |
| 如果需要支持具有多个活页夹类型的动态目标，请对通用类型使用Object，并根据需要强制转换extended参数。 |

## **32.内容类型协商**

数据转换是任何消息驱动的微服务体系结构的核心功能之一。假设在Spring Cloud Stream中，此类数据表示为Spring Message，则在到达消息之前，可能必须将消息转换为所需的形状或大小。这是必需的，原因有两个：

1. 转换传入消息的内容以匹配应用程序提供的处理程序的签名。
2. 将外发邮件的内容转换为有线格式。

有线格式通常为byte[]（对于Kafka和Rabbit活页夹而言是正确的），但是它由活页夹实现方式控制。

在Spring Cloud Stream中，消息转换是通过org.springframework.messaging.converter.MessageConverter完成的。

|  |
| --- |
| IMG_388 |
| 作为后续细节的补充，您可能还需要阅读以下[博客文章](https://spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation" \t "https://www.springcloud.cc/_top)。 |

## **32.1力学**

为了更好地理解内容类型协商的机制和必要性，我们以下面的消息处理程序为例，看一个非常简单的用例：

@StreamListener(Processor.INPUT)@SendTo(Processor.OUTPUT)**public** String handle(Person person) {..}

|  |
| --- |
| IMG_389 |
| 为简单起见，我们假设这是应用程序中唯一的处理程序（我们假设没有内部管道）。 |

前面示例中所示的处理程序期望将Person对象作为参数，并产生String类型作为输出。为了使框架成功将传入的Message作为参数传递给此处理程序，它必须以某种方式将Message类型的有效负载从有线格式转换为Person类型。换句话说，框架必须找到并应用适当的MessageConverter。为此，该框架需要用户提供一些指导。处理程序方法本身的签名（Person类型）已经提供了这些指令之一。因此，从理论上讲，这应该是（并且在某些情况下是足够的）。但是，对于大多数用例而言，为了选择适当的MessageConverter，框架需要额外的信息。缺少的部分是contentType。

Spring Cloud Stream提供了三种机制来定义contentType（按优先顺序）：

1. **标题**：contentType可以通过消息本身进行通信。通过提供contentType标头，您可以声明用于查找和应用适当的MessageConverter的内容类型。

**绑定**：可以通过设置spring.cloud.stream.bindings.input.content-type属性为每个目标绑定设置contentType。

|  |
| --- |
| IMG_390 |
| 属性名称中的input段与目的地的实际名称相对应（在本例中为“输入”）。通过这种方法，您可以在每个绑定的基础上声明用于查找和应用适当的MessageConverter的内容类型。 |

1. **默认值**：如果Message标头或绑定中不存在contentType，则使用默认的application/json内容类型来查找和应用适当的MessageConverter。

如前所述，前面的列表还演示了平局时的优先顺序。例如，标头提供的内容类型优先于任何其他内容类型。对于按绑定设置的内容类型也是如此，这实际上使您可以覆盖默认内容类型。但是，它也提供了明智的默认值（由社区反馈确定）。

将application/json设置为默认值的另一个原因是由分布式微服务体系结构驱动的互操作性要求，在该体系结构中，生产者和使用者不仅可以在不同的JVM中运行，而且还可以在不同的非JVM平台上运行。

当非无效处理程序方法返回时，如果返回值已经是Message，则该Message成为有效负载。但是，当返回值不是Message时，将使用返回值作为有效负载构造新的Message，同时从输入Message继承标头减去由SpringIntegrationProperties.messageHandlerNotPropagatedHeaders定义或过滤的标头。默认情况下，仅设置一个标头：contentType。这意味着新的Message没有设置contentType头，从而确保contentType可以演进。您始终可以选择不从处理程序方法返回Message的位置，在该方法中可以注入所需的任何标头。

如果存在内部管道，则通过相同的转换过程将Message发送到下一个处理程序。但是，如果没有内部管道或您已经到达内部管道的末尾，则Message将发送回输出目的地。

### **32.1.1内容类型与参数类型**

如前所述，为了使框架选择适当的MessageConverter，它需要参数类型以及（可选）内容类型信息。选择适当的MessageConverter的逻辑驻留在参数解析器（HandlerMethodArgumentResolvers）中，该解析器在调用用户定义的处理程序方法之前（即当框架知道实际的参数类型时）触发。如果参数类型与当前有效负载的类型不匹配，则框架将委派给预先配置的MessageConverters的堆栈，以查看其中是否有一个可以转换有效负载。如您所见，MessageConverter的Object fromMessage(Message<?> message, Class<?> targetClass);操作将targetClass作为其参数之一。该框架还确保提供的Message始终包含一个contentType头。当没有contentType标头时，它会插入按绑定的contentType标头或默认的contentType标头。contentType参数类型的组合是框架确定消息是否可以转换为目标类型的机制。如果找不到合适的MessageConverter，则会引发异常，您可以通过添加自定义MessageConverter来处理该异常（请参见“ [第32.3节，“用户定义的消息转换器”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-user-defined-message-converters" \o "32.3用户定义的消息转换器) ”）。

但是，如果有效载荷类型与处理程序方法声明的目标类型匹配，该怎么办？在这种情况下，没有任何要转换的内容，并且有效载荷未经修改地传递。尽管这听起来很简单且合乎逻辑，但请记住以Message<?>或Object作为参数的处理程序方法。通过将目标类型声明为Object（在Java中为instanceof，是所有内容），实际上就放弃了转换过程。

|  |
| --- |
| IMG_391 |
| 不要期望仅根据contentType将Message转换为其他类型。请记住，contentType是目标类型的补充。如果需要，您可以提供一个提示，MessageConverter可能会也可能不会考虑。 |

### **32.1.2消息转换器**

MessageConverters定义两种方法：

Object fromMessage(Message<?> message, Class<?> targetClass);

Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);

重要的是要了解这些方法的约定及其用法，尤其是在Spring Cloud Stream的上下文中。

fromMessage方法将传入的Message转换为参数类型。Message的有效载荷可以是任何类型，而MessageConverter的实际实现要支持多种类型。例如，某些JSON转换器可能支持有效负载类型为byte[]，String等。当应用程序包含内部管道（即输入→handler1→handler2→....→输出）并且上游处理程序的输出结果为Message时，这可能不是初始连接格式，这一点很重要。

但是，toMessage方法的合同更为严格，必须始终将Message转换为有线格式：byte[]。

因此，出于所有意图和目的（尤其是在实现自己的转换器时），您将这两种方法视为具有以下签名：

Object fromMessage(Message<?> message, Class<?> targetClass);

Message<**byte**[]> toMessage(Object payload, @Nullable MessageHeaders headers);

## **32.2提供的MessageConverters**

如前所述，该框架已经提供了MessageConverters堆栈来处理最常见的用例。以下列表按优先级描述了提供的MessageConverters（使用了第一个有效的MessageConverter）：

1. ApplicationJsonMessageMarshallingConverter：org.springframework.messaging.converter.MappingJackson2MessageConverter的变体。对于contentType为application/json（默认）的情况，支持将Message的有效负载转换为POJO或从POJO转换为PO195。
2. TupleJsonMessageConverter：已**弃用**支持将Message的有效负载转换为org.springframework.tuple.Tuple或从org.springframework.tuple.Tuple转换。
3. ByteArrayMessageConverter：在contentType为application/octet-stream的情况下，支持将Message的有效载荷从byte[]转换为byte[]。它本质上是一个传递，主要是为了向后兼容而存在。
4. ObjectStringMessageConverter：当contentType为text/plain时，支持将任何类型转换为String。它调用Object的toString()方法，或者，如果有效载荷为byte[]，则调用新的String(byte[])。
5. JavaSerializationMessageConverter：已**弃用**当contentType为application/x-java-serialized-object时，支持基于Java序列化的转换。
6. KryoMessageConverter：已**弃用**当contentType为application/x-java-object时，支持基于Kryo序列化的转换。
7. JsonUnmarshallingConverter：类似于ApplicationJsonMessageMarshallingConverter。当contentType为application/x-java-object时，它支持任何类型的转换。它期望将实际类型信息作为属性嵌入在contentType中（例如，application/x-java-object;type=foo.bar.Cat）。

当找不到合适的转换器时，框架将引发异常。发生这种情况时，应检查代码和配置，并确保您没有错过任何内容（即，确保使用绑定或标头提供了contentType）。但是，很可能您发现了一些不常见的情况（例如自定义contentType），并且提供的MessageConverters的当前堆栈不知道如何进行转换。在这种情况下，您可以添加自定义MessageConverter。请参见[第32.3节“用户定义的消息转换器”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-user-defined-message-converters" \o "32.3用户定义的消息转换器)。

## **32.3用户定义的消息转换器**

Spring Cloud Stream公开了定义和注册其他MessageConverters的机制。要使用它，请实现org.springframework.messaging.converter.MessageConverter，将其配置为@Bean，并用@StreamMessageConverter进行注释。然后将其附加到MessageConverter的现有堆栈中。

|  |
| --- |
| IMG_392 |
| 了解自定义MessageConverter实现已添加到现有堆栈的开头很重要。因此，自定义MessageConverter实现优先于现有实现，您可以覆盖它们并添加到现有转换器中。 |

下面的示例说明如何创建消息转换器bean以支持称为application/bar的新内容类型：

@EnableBinding(Sink.class)@SpringBootApplication**public** **static** **class** SinkApplication {

...

@Bean

@StreamMessageConverter

**public** MessageConverter customMessageConverter() {

**return** **new** MyCustomMessageConverter();

}

}

**public** **class** MyCustomMessageConverter **extends** AbstractMessageConverter {

**public** MyCustomMessageConverter() {

**super**(**new** MimeType("application", "bar"));

}

@Override

**protected** **boolean** supports(Class<?> clazz) {

**return** (Bar.**class**.equals(clazz));

}

@Override

**protected** Object convertFromInternal(Message<?> message, Class<?> targetClass, Object conversionHint) {

Object payload = message.getPayload();

**return** (payload **instanceof** Bar ? payload : **new** Bar((**byte**[]) payload));

}

}

Spring Cloud Stream还为基于Avro的转换器和模式演变提供支持。有关详细信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "schema-evolution" \o "33. Schema Evolution Support)请参见“ [第33章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "schema-evolution" \o "33.模式演化支持)*[Schema Evolution支持](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "schema-evolution" \o "33.模式演化支持)* ”。

## **33. Schema进化支持**

Spring Cloud Stream为模式演化提供了支持，因此数据可以随着时间的推移而演化，并且仍然可以与较新的生产者和消费者以及反之亦然。大多数序列化模型，尤其是旨在跨不同平台和语言进行移植的模型，都依赖于一种描述如何在二进制有效负载中序列化数据的模式。为了序列化数据然后解释它，发送方和接收方都必须有权访问描述二进制格式的模式。在某些情况下，可以从序列化时的有效负载类型或反序列化时的目标类型推断模式。但是，许多应用程序可以从访问描述二进制数据格式的显式架构中受益。通过模式注册表，您可以以文本格式（通常为JSON）存储模式信息，并使该信息可用于需要它以二进制格式接收和发送数据的各种应用程序。模式可引用为一个元组，该元组包括：

* 主题，是架构的逻辑名称
* 模式版本
* 模式格式，描述数据的二进制格式

以下各节详细介绍了架构演变过程中涉及的各种组件。

## **33.1 Schema注册客户端**

与模式注册表服务器进行交互的客户端抽象是SchemaRegistryClient接口，该接口具有以下结构：

**public** **interface** SchemaRegistryClient {

SchemaRegistrationResponse register(String subject, String format, String schema);

String fetch(SchemaReference schemaReference);

String fetch(Integer id);

}

Spring Cloud Stream提供了开箱即用的实现，可以与其自己的模式服务器进行交互，也可以与Confluent Schema注册中心进行交互。

可以使用@EnableSchemaRegistryClient来配置Spring Cloud Stream模式注册表的客户端，如下所示：

@EnableBinding(Sink.class)

@SpringBootApplication

@EnableSchemaRegistryClient

**public** **static** **class** AvroSinkApplication {

...

}

|  |
| --- |
| IMG_393 |
| 默认转换器经过优化，不仅可以缓存来自远程服务器的模式，还可以缓存parse()和toString()方法，这是非常昂贵的。因此，它使用了不缓存响应的DefaultSchemaRegistryClient。如果要更改默认行为，则可以直接在代码上使用客户端，并将其覆盖为所需的结果。为此，您必须将属性spring.cloud.stream.schemaRegistryClient.cached=true添加到应用程序属性中。 |

### **33.1.1 Schema注册客户端Properties**

Schema Registry Client支持以下属性：

spring.cloud.stream.schemaRegistryClient.endpoint

模式服务器的位置。进行设置时，请使用完整的URL，包括协议（http或https），端口和上下文路径。

默认

[http://localhost:8990/](http://localhost:8990/" \t "https://www.springcloud.cc/_top)

spring.cloud.stream.schemaRegistryClient.cached

客户端是否应缓存架构服务器响应。由于缓存发生在消息转换器中，因此通常设置为false。使用架构注册表客户端的客户端应将此设置为true。

默认

true

## **33.2 Avro Schema注册客户端消息转换器**

对于已在应用程序上下文中注册了SchemaRegistryClient bean的应用程序，Spring Cloud Stream自动为架构管理配置Apache Avro消息转换器。由于接收消息的应用程序可以轻松访问可以与自己的读取器模式进行协调的写入器模式，因此这简化了模式的演变。

对于出站消息，如果通道的内容类型设置为application/\*+avro，则激活MessageConverter，如下例所示：

spring.cloud.stream.bindings.output.contentType=application/\*+avro

在出站转换期间，消息转换器尝试使用SchemaRegistryClient推断每个出站消息的模式（基于其类型）并将其注册到主题（基于有效负载类型）。如果已经找到相同的模式，则将检索对其的引用。如果不是，则注册架构，并提供新的版本号。通过使用以下方案，将消息与contentType头一起发送：application/[prefix].[subject].v[version]+avro，其中prefix是可配置的，而subject是从有效负载类型推导出来的。

例如，类型为User的消息可能作为二进制有效载荷发送，其内容类型为application/vnd.user.v2+avro，其中user是主题，2是版本号。

接收消息时，转换器从传入消息的标头中推断模式引用，并尝试检索它。该模式在反序列化过程中用作编写器模式。

### **33.2.1 Avro Schema注册表消息转换器Properties**

如果通过设置spring.cloud.stream.bindings.output.contentType=application/\*+avro启用了基于Avro的架构注册表客户端，则可以通过设置以下属性来自定义注册行为。

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled

如果您希望转换器使用反射来从POJO推断Schema，请启用。

默认值：false

spring.cloud.stream.schema.avro.readerSchema

Avro通过查看写入器模式（原始有效负载）和读取器模式（您的应用程序有效负载）来比较模式版本。有关更多信息，请参见[Avro文档](https://avro.apache.org/docs/1.7.6/spec.html" \t "https://www.springcloud.cc/_top)。如果设置，它将覆盖在模式服务器上的所有查找，并将本地模式用作读取器模式。默认值：null

spring.cloud.stream.schema.avro.schema位置

在Schema服务器中注册此属性中列出的所有.avsc文件。

默认值：empty

spring.cloud.stream.schema.avro.prefix

Content-Type标头上要使用的前缀。

默认值：vnd

## **33.3 Apache Avro消息转换器**

Spring Cloud Stream通过其spring-cloud-stream-schema模块为基于模式的消息转换器提供支持。当前，基于模式的消息转换器开箱即用的唯一序列化格式是Apache Avro，将来的版本中将添加更多格式。

spring-cloud-stream-schema模块包含可用于Apache Avro序列化的两种消息转换器：

* 转换器使用序列化或反序列化对象的类信息或启动时具有已知位置的模式。
* 使用架构注册表的转换器。它们在运行时定位架构，并随着域对象的发展动态注册新架构。

## **33.4支持Schema的转换器**

AvroSchemaMessageConverter支持通过使用预定义的架构或通过使用类中可用的架构信息（反射地或包含在SpecificRecord中）来对消息进行序列化和反序列化。如果提供自定义转换器，则不会创建默认的AvroSchemaMessageConverter bean。以下示例显示了一个自定义转换器：

要使用自定义转换器，您只需将其添加到应用程序上下文中，就可以选择指定一个或多个与其关联的MimeTypes。默认MimeType为application/avro。

如果转换的目标类型是GenericRecord，则必须设置架构。

以下示例显示如何通过在没有预定义架构的情况下注册Apache Avro MessageConverter在接收器应用程序中配置转换器。在此示例中，请注意，哑剧类型值为avro/bytes，而不是默认的application/avro。

@EnableBinding(Sink.class)@SpringBootApplication**public** **static** **class** SinkApplication {

...

@Bean

**public** MessageConverter userMessageConverter() {

**return** **new** AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));

}

}

相反，以下应用程序使用预定义的架构（在类路径上找到）注册一个转换器：

@EnableBinding(Sink.class)@SpringBootApplication**public** **static** **class** SinkApplication {

...

@Bean

**public** MessageConverter userMessageConverter() {

AvroSchemaMessageConverter converter = **new** AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));

converter.setSchemaLocation(**new** ClassPathResource("schemas/User.avro"));

**return** converter;

}

}

## **33.5 Schema注册表服务器**

Spring Cloud Stream提供了模式注册表服务器实现。要使用它，可以将spring-cloud-stream-schema-server工件添加到项目中，并使用@EnableSchemaRegistryServer批注，该批注将架构注册表服务器REST控制器添加到您的应用程序。该注释旨在与Spring Boot web应用程序一起使用，并且服务器的侦听端口由server.port属性控制。spring.cloud.stream.schema.server.path属性可用于控制模式服务器的根路径（尤其是当它嵌入在其他应用程序中时）。spring.cloud.stream.schema.server.allowSchemaDeletion布尔属性可以删除模式。默认情况下，这是禁用的。

架构注册表服务器使用关系数据库来存储架构。默认情况下，它使用嵌入式数据库。您可以使用[Spring Boot SQL数据库和JDBC配置选项来自](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/" \l "boot-features-sql" \t "https://www.springcloud.cc/_top)定义模式存储。

以下示例显示了一个启用架构注册表的Spring Boot应用程序：

@SpringBootApplication@EnableSchemaRegistryServer**public** **class** SchemaRegistryServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SchemaRegistryServerApplication.**class**, args);

}

}

### **33.5.1 Schema Registry Server API**

Schema Registry Server API包含以下操作：

* POST / —参见“ [称为“注册新的Schema”部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-registering-new-schema" \o "注册新架构) ”
* 'GET / {subject} / {format} / {version}'-参见“ [称为“按主题，格式和版本检索现有Schema的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-subject-format-version" \o "通过主题，格式和版本检索现有架构) ”
* GET /{subject}/{format} —参见“ [称为“按主题和格式检索现有Schema的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-subject-format" \o "通过主题和格式检索现有模式) ”
* GET /schemas/{id}-参见“ [称为“通过ID检索现有Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-id" \o "通过ID检索现有架构) ” [的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-retrieve-schema-id" \o "Retrieving an Existing Schema by ID) ”
* DELETE /{subject}/{format}/{version} —请参阅“ [称为“按主题，格式和版本删除Schema的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-subject-format-version" \o "按主题，格式和版本删除架构) ”
* DELETE /schemas/{id} —参见“ [称为“通过ID删除Schema](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-id" \o "通过ID删除架构) ” [的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-id" \o "Deleting a Schema by ID) ”
* DELETE /{subject}-参见“ [称为“按主题删除Schema的部分”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-deleting-schema-subject" \o "按主题删除架构) ”

#### **注册新的Schema**

要注册新模式，请向/端点发送一个POST请求。

/接受具有以下字段的JSON有效负载：

* subject：架构主题
* format：架构格式
* definition：架构定义

它的响应是JSON中的架构对象，具有以下字段：

* id：架构ID
* subject：架构主题
* format：架构格式
* version：架构版本
* definition：模式定义

#### **通过主题，格式和版本检索现有的Schema**

要按主题，格式和版本检索现有架构，请向/{subject}/{format}/{version}端点发送GET请求。

它的响应是JSON中的架构对象，具有以下字段：

* id：架构ID
* subject：架构主题
* format：架构格式
* version：架构版本
* definition：模式定义

#### **通过主题和格式检索现有的Schema**

要按主题和格式检索现有架构，请向/subject/format端点发送一个GET请求。

它的响应是JSON中每个模式对象的模式列表，其中包含以下字段：

* id：模式ID
* subject：架构主题
* format：架构格式
* version：架构版本
* definition：架构定义

#### **通过ID检索现有的Schema**

要通过其ID检索架构，请向/schemas/{id}端点发送一个GET请求。

它的响应是JSON中的架构对象，具有以下字段：

* id：架构ID
* subject：架构主题
* format：架构格式
* version：架构版本
* definition：模式定义

#### **按主题，格式和版本删除Schema**

要删除由其主题，格式和版本标识的模式，请向/{subject}/{format}/{version}端点发送一个DELETE请求。

#### **按ID删除Schema**

要通过其ID删除模式，请向/schemas/{id}端点发送一个DELETE请求。

#### **按主题删除Schema**

DELETE /{subject}

按主题删除现有架构。

|  |
| --- |
| IMG_394 |
| 本说明仅适用于Spring Cloud Stream 1.1.0.RELEASE的用户。Spring Cloud Stream 1.1.0.RELEASE使用表名schema来存储Schema对象。Schema是许多数据库实现中的关键字。为了避免将来发生任何冲突，从1.1.1.RELEASE开始，我们为存储表选择了名称SCHEMA\_REPOSITORY。任何升级的Spring Cloud Stream 1.1.0.RELEASE用户都应在升级之前将其现有模式迁移到新表中。 |

### **33.5.2使用Confluent的Schema注册表**

默认配置将创建DefaultSchemaRegistryClient bean。如果要使用Confluent模式注册表，则需要创建类型为ConfluentSchemaRegistryClient的bean，该类型将替代框架默认配置的类型。下面的示例说明如何创建这样的bean：

@Bean**public** SchemaRegistryClient schemaRegistryClient(@Value("${spring.cloud.stream.schemaRegistryClient.endpoint}") String endpoint){

ConfluentSchemaRegistryClient client = **new** ConfluentSchemaRegistryClient();

client.setEndpoint(endpoint);

**return** client;

}

|  |
| --- |
| IMG_395 |
| ConfluentSchemaRegistryClient已针对Confluent平台4.0.0版进行了测试。 |

## **33.6 Schema注册和解决**

为了更好地了解Spring Cloud Stream如何注册和解析新模式及其对Avro模式比较功能的使用，我们提供了两个单独的小节：

* “ [第33.6.1节“ Schema注册过程（序列化）”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-schema-registration-process" \o "33.6.1模式注册过程（序列化）) ”
* “ [第33.6.2节“ Schema解析过程（反序列化）”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-schema-resolution-process" \o "33.6.2模式解析过程（反序列化）) ”

### **33.6.1 Schema注册流程（序列化）**

注册过程的第一部分是从通过通道发送的有效负载中提取模式。诸如SpecificRecord或GenericRecord之类的Avro类型已经包含一个架构，可以从实例中立即检索该架构。对于POJO，如果将spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled属性设置为true（默认值），则将推断模式。

**图33.1 Schema作家解析过程**

IMG_396

获得一个模式，转换器从远程服务器加载其元数据（版本）。首先，它查询本地缓存。如果未找到结果，它将把数据提交给服务器，服务器将提供版本信息。转换器始终缓存结果，以避免为每个需要序列化的新消息查询Schema服务器的开销。

**图33.2 Schema注册流程**

IMG_397

使用架构版本信息，转换器将消息的contentType标头设置为携带版本信息，例如：application/vnd.user.v1+avro。

### **33.6.2 Schema解析过程（反序列化）**

当读取包含版本信息的消息（即contentType标头，其格式类似于“ [第33.6.1节“ Schema注册过程（序列化）”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-schema-registration-process" \o "33.6.1模式注册过程（序列化）) ”）所述）时，转换器将查询Schema服务器以获取消息的编写器架构。一旦找到了传入消息的正确架构，它将检索阅读器架构，并使用Avro的架构解析支持将其读入阅读器定义（设置默认值和所有缺少的属性）。

**图33.3 Schema阅读解决流程**

IMG_398

|  |
| --- |
| IMG_399 |
| 您应该了解编写者架构（编写消息的应用程序）和阅读者架构（接收应用程序）之间的区别。我们建议[花点时间](https://avro.apache.org/docs/1.7.6/spec.html" \t "https://www.springcloud.cc/_top)阅读[Avro术语](https://avro.apache.org/docs/1.7.6/spec.html" \t "https://www.springcloud.cc/_top)并了解其过程。Spring Cloud Stream始终获取编写程序架构以确定如何读取消息。如果要使Avro的模式演变支持正常工作，则需要确保为应用程序正确设置了readerSchema。 |

## **34.应用程序间通信**

Spring Cloud Stream启用应用程序之间的通信。应用程序间通信是一个涉及多个问题的复杂问题，如以下主题所述：

* “ [第34.1节“连接多个应用程序实例”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-connecting-multiple-application-instances" \o "34.1连接多个应用程序实例) ”
* “ [第34.2节“实例索引和实例计数”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-instance-index-instance-count" \o "34.2实例索引和实例计数) ”
* “ [第34.3节“分区”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-partitioning" \o "34.3分区) ”

## **34.1连接多个应用程序实例**

尽管Spring Cloud Stream使单个Spring Boot应用程序易于连接到消息传递系统，但Spring Cloud Stream的典型方案是创建多应用程序管道，微服务应用程序在该管道中相互发送数据。您可以通过关联“ 相邻 ”应用程序的输入和输出目标来实现此方案。

假设设计要求Time Source应用程序将数据发送到Log Sink应用程序。您可以将名为ticktock的公共目标用于两个应用程序中的绑定。

时间源（通道名称为output）将设置以下属性：

spring.cloud.stream.bindings.output.destination=ticktock

日志接收器（通道名称为input）将设置以下属性：

spring.cloud.stream.bindings.input.destination=ticktock

## **34.2实例索引和实例计数**

在扩展Spring Cloud Stream应用程序时，每个实例可以接收有关同一应用程序还存在多少其他实例以及它自己的实例索引是什么的信息。Spring Cloud Stream通过spring.cloud.stream.instanceCount和spring.cloud.stream.instanceIndex属性进行此操作。例如，如果存在HDFS接收器应用程序的三个实例，则所有三个实例的spring.cloud.stream.instanceCount都设置为3，并且各个应用程序的spring.cloud.stream.instanceIndex都设置为0，1，和2。

通过Spring Cloud Data Flow部署Spring Cloud Stream应用程序时，将自动配置这些属性；否则，将自动配置这些属性。分别启动Spring Cloud Stream应用程序时，必须正确设置这些属性。默认情况下，spring.cloud.stream.instanceCount是1，而spring.cloud.stream.instanceIndex是0。

在按比例放大的方案中，这两个属性的正确配置通常对于解决分区行为很重要（请参见下文），并且某些绑定程序（例如，Kafka绑定程序）始终需要这两个属性，以便确保在多个使用者实例之间正确分割数据。

## **34.3分区**

Spring Cloud Stream中的分区包括两个任务：

* “ [第34.3.1节“配置用于分区的输出绑定”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-configuring-output-bindings-partitioning" \o "34.3.1配置用于分区的输出绑定) ”
* “ [第34.3.2节“配置用于分区的输入绑定”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-configuring-input-bindings-partitioning" \o "34.3.2配置用于分区的输入绑定) ”

### **34.3.1配置用于分区的输出绑定**

您可以通过设置其partitionKeyExpression或partitionKeyExtractorName属性及其partitionCount属性中的一个或仅一个，来配置输出绑定以发送分区数据。

例如，以下是有效的典型配置：

spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id

spring.cloud.stream.bindings.output.producer.partitionCount=5

基于该示例配置，通过使用以下逻辑将数据发送到目标分区。

根据partitionKeyExpression为发送到分区输出通道的每条消息计算分区键的值。partitionKeyExpression是一个SpEL表达式，该表达式根据出站消息进行评估以提取分区键。

如果SpEL表达式不足以满足您的需要，则可以通过提供org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy的实现并将其配置为bean（通过使用@Bean注释）来计算分区键值。 。如果在应用程序上下文中有多个org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy类型的bean，则可以通过使用partitionKeyExtractorName属性指定其名称来进一步过滤它，如以下示例所示：

--spring.cloud.stream.bindings.output.producer.partitionKeyExtractorName=customPartitionKeyExtractor

--spring.cloud.stream.bindings.output.producer.partitionCount=5

. . .

@Bean

public CustomPartitionKeyExtractorClass customPartitionKeyExtractor() {

return new CustomPartitionKeyExtractorClass();

}

|  |
| --- |
| IMG_400 |
| 在Spring Cloud Stream的早期版本中，您可以通过设置spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass属性来指定org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy的实现。从2.0版开始，不推荐使用此属性，并且在将来的版本中将不再支持该属性。 |

一旦计算出消息密钥，分区选择过程就会将目标分区确定为0与partitionCount - 1之间的值。适用于大多数情况的默认计算基于以下公式：key.hashCode() % partitionCount。这可以在绑定上进行自定义，方法是将SpEL表达式设置为针对'key'进行评估（通过partitionSelectorExpression属性），也可以将org.springframework.cloud.stream.binder.PartitionSelectorStrategy的实现配置为bean （通过使用@ Bean批注）。与PartitionKeyExtractorStrategy类似，当应用程序上下文中有多个这种类型的bean可用时，您可以使用spring.cloud.stream.bindings.output.producer.partitionSelectorName属性进一步过滤它，如以下示例所示：

--spring.cloud.stream.bindings.output.producer.partitionSelectorName=customPartitionSelector

. . .

@Bean

public CustomPartitionSelectorClass customPartitionSelector() {

return new CustomPartitionSelectorClass();

}

|  |
| --- |
| IMG_401 |
| 在Spring Cloud Stream的早期版本中，您可以通过设置spring.cloud.stream.bindings.output.producer.partitionSelectorClass属性来指定org.springframework.cloud.stream.binder.PartitionSelectorStrategy的实现。从2.0版开始，不推荐使用此属性，并且在将来的版本中将不再支持该属性。 |

### **34.3.2配置用于分区的输入绑定**

通过设置应用程序本身的partitioned属性以及应用程序本身的instanceIndex和instanceCount属性，将输入绑定（通道名称为input）配置为接收分区数据。下面的例子：

spring.cloud.stream.bindings.input.consumer.partitioned=true

spring.cloud.stream.instanceIndex=3

spring.cloud.stream.instanceCount=5

instanceCount值表示应在其之间分区数据的应用程序实例的总数。instanceIndex在多个实例中必须是唯一的值，其值应介于0和instanceCount - 1之间。实例索引可帮助每个应用程序实例识别从中接收数据的唯一分区。活页夹要求使用不支持本地分区的技术。例如，对于RabbitMQ，每个分区都有一个队列，该队列名称包含实例索引。对于Kafka，如果autoRebalanceEnabled为true（默认值），则Kafka负责在实例之间分配分区，并且不需要这些属性。如果autoRebalanceEnabled设置为false，则绑定器将使用instanceCount和instanceIndex来确定实例所预订的分区（您必须拥有与实例数量一样多的分区） 。活页夹分配分区而不是Kafka。如果您希望特定分区的消息始终发送到同一实例，这可能很有用。当活页夹配置需要它们时，重要的是正确设置两个值，以确保使用所有数据，并且应用程序实例接收互斥的数据集。

尽管在单独情况下使用多个实例进行分区数据处理可能会很复杂，但Spring Cloud Dataflow可以通过正确填充输入和输出值并让您依赖运行时来显着简化流程基础架构，以提供有关实例索引和实例计数的信息。

## **35.测试**

Spring Cloud Stream提供了在不连接消息传递系统的情况下测试您的微服务应用程序的支持。您可以使用spring-cloud-stream-test-support库提供的TestSupportBinder库来完成此操作，该库可以作为测试依赖项添加到应用程序中，如以下示例所示：

<dependency>

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

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency>

|  |
| --- |
| IMG_402 |
| TestSupportBinder使用Spring Boot自动配置机制来取代在类路径上找到的其他绑定器。因此，在添加活页夹作为依赖项时，必须确保使用了test范围。 |

TestSupportBinder使您可以与绑定的频道进行交互，并检查应用程序发送和接收的所有消息。

对于出站消息通道，TestSupportBinder注册一个订户，并将应用程序发出的消息保留在MessageCollector中。在测试期间可以检索它们，并针对它们进行断言。

您还可以将消息发送到入站消息通道，以便使用者应用程序可以使用消息。以下示例显示了如何在处理器上测试输入和输出通道：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM\_PORT)**public** **class** ExampleTest {

@Autowired

**private** Processor processor;

@Autowired

**private** MessageCollector messageCollector;

@Test

@SuppressWarnings("unchecked")

**public** **void** testWiring() {

Message<String> message = **new** GenericMessage<>("hello");

processor.input().send(message);

Message<String> received = (Message<String>) messageCollector.forChannel(processor.output()).poll();

assertThat(received.getPayload(), equalTo("hello world"));

}

@SpringBootApplication

@EnableBinding(Processor.class)

**public** **static** **class** MyProcessor {

@Autowired

**private** Processor channels;

@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)

**public** String transform(String in) {

**return** in + " world";

}

}

}

在前面的示例中，我们创建了一个具有输入通道和输出通道的应用程序，两者均通过Processor接口绑定。绑定的接口被注入到测试中，以便我们可以访问两个通道。我们在输入通道上发送一条消息，然后使用Spring Cloud Stream的测试支持提供的MessageCollector捕获消息已作为结果发送到输出通道。收到消息后，我们可以验证组件是否正常运行。

## **35.1禁用测试Binder自动配置**

测试绑定程序背后的目的是取代类路径上的所有其他绑定程序，以使其易于测试您的应用程序而无需更改生产依赖性。在某些情况下（例如，集成测试），使用实际的生产绑定程序是有用的，并且这需要禁用测试绑定程序自动配置。为此，可以使用Spring Boot自动配置排除机制之一来排除org.springframework.cloud.stream.test.binder.TestSupportBinderAutoConfiguration类，如以下示例所示：

@SpringBootApplication(exclude = TestSupportBinderAutoConfiguration.class)

@EnableBinding(Processor.class)

**public** **static** **class** MyProcessor {

@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)

**public** String transform(String in) {

**return** in + " world";

}

}

禁用自动配置后，测试绑定程序将在类路径上可用，并且其defaultCandidate属性设置为false，以使其不会干扰常规用户配置。可以使用名称test来引用它，如以下示例所示：

spring.cloud.stream.defaultBinder=test

## **36.健康指标**

Spring Cloud Stream为活页夹提供了健康指标。它以名称binders注册，可以通过设置management.health.binders.enabled属性来启用或禁用。

默认情况下，management.health.binders.enabled设置为false。将management.health.binders.enabled设置为true会启用运行状况指示器，使您可以访问/health端点以检索绑定程序运行状况指示器。

健康指标是特定于活页夹的，某些活页夹实现不一定提供健康指标。

## **37.指标发射器**

Spring Boot Actuator为[Micrometer](https://micrometer.io/" \t "https://www.springcloud.cc/_top)提供依赖项管理和自动配置，[Micrometer](https://micrometer.io/" \t "https://www.springcloud.cc/_top)是一种支持众多[监视系统](https://docs.spring.io/spring-boot/docs/2.0.0.RELEASE/reference/htmlsingle/" \l "production-ready-metrics" \t "https://www.springcloud.cc/_top)的应用程序度量外观。

Spring Cloud Stream提供了将任何可用的基于千分尺的度量标准发送到绑定目标的支持，从而允许从流应用程序定期收集度量标准数据，而无需依赖于轮询各个端点。

通过定义spring.cloud.stream.bindings.applicationMetrics.destination属性来激活“度量标准发射器”，该属性指定当前绑定程序用于发布度量标准消息的绑定目标的名称。

例如：

spring.cloud.stream.bindings.applicationMetrics.destination=myMetricDestination

前面的示例指示绑定程序绑定到myMetricDestination（即，Rabbit交换，Kafka主题等）。

以下属性可用于自定义指标的发射：

spring.cloud.stream.metrics.key

发出的度量标准的名称。每个应用程序的唯一值。

默认值：${spring.application.name:${vcap.application.name:${spring.config.name:application}}}

spring.cloud.stream.metrics.properties

允许白名单应用程序属性添加到度量有效负载

默认值：null。

spring.cloud.stream.metrics.meter-filter

控制要捕获的“仪表”的模式。例如，指定spring.integration.\*将捕获名称以spring.integration.开头的仪表的度量标准信息

默认值：捕获所有“仪表”。

spring.cloud.stream.metrics.schedule-interval

控制发布度量标准数据的速率的时间间隔。

默认值：1分钟

考虑以下：

java -jar time-source.jar \

--spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \

--spring.cloud.stream.metrics.properties=spring.application\*\* \

--spring.cloud.stream.metrics.meter-filter=spring.integration.\*

下面的示例显示由于上述命令而发布到绑定目标的数据的有效负载：

{

"name": "application",

"createdTime": "2018-03-23T14:48:12.700Z",

"properties": {

},

"metrics": [

{

"id": {

"name": "spring.integration.send",

"tags": [

{

"key": "exception",

"value": "none"

},

{

"key": "name",

"value": "input"

},

{

"key": "result",

"value": "success"

},

{

"key": "type",

"value": "channel"

}

],

"type": "TIMER",

"description": "Send processing time",

"baseUnit": "milliseconds"

},

"timestamp": "2018-03-23T14:48:12.697Z",

"sum": 130.340546,

"count": 6,

"mean": 21.72342433333333,

"upper": 116.176299,

"total": 130.340546

}

]

}

|  |
| --- |
| IMG_403 |
| 鉴于在迁移到Micrometer后Metric消息的格式略有变化，发布的消息还将STREAM\_CLOUD\_STREAM\_VERSION标头设置为2.x，以帮助区分Metric消息和旧版本的Spring Cloud Stream 。 |

## **38.示例**

有关Spring Cloud Stream示例，请参见GitHub上的[spring-cloud-stream-samples](https://github.com/spring-cloud/spring-cloud-stream-samples" \t "https://www.springcloud.cc/_top)存储库。

## **38.1在CloudFoundry上部署流应用程序**

在CloudFoundry上，通常通过称为[VCAP\_SERVICES](https://docs.cloudfoundry.org/devguide/deploy-apps/environment-variable.html" \l "VCAP-SERVICES" \t "https://www.springcloud.cc/_top)的特殊环境变量来公开服务。

配置资料夹连接时，可以使用环境变量中的值，如[数据流Cloud Foundry服务器](https://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/current-SNAPSHOT/reference/htmlsingle/" \l "getting-started-ups" \t "https://www.springcloud.cc/_top)文档中所述。

# **第六部分 Binder实现**

## **39. Apache Kafka Binder**

## **39.1使用**

要使用Apache Kafka活页夹，您需要将spring-cloud-stream-binder-kafka作为依赖项添加到Spring Cloud Stream应用程序中，如以下Maven的示例所示：

<dependency>

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

<artifactId>spring-cloud-stream-binder-kafka</artifactId></dependency>

另外，您也可以使用Spring Cloud Stream Kafka入门程序，如以下Maven示例所示：

<dependency>

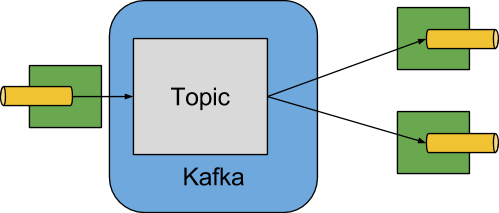
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-kafka</artifactId></dependency>

## **39.2 Apache Kafka Binder概述**

下图显示了Apache Kafka活页夹的工作方式的简化图：

**图39.1。Kafka Binder**



Apache Kafka Binder实现将每个目的地映射到Apache Kafka主题。消费者组直接映射到相同的Apache Kafka概念。分区也直接映射到Apache Kafka分区。

活页夹当前使用Apache Kafka kafka-clients 1.0.0 jar，并且设计为与至少该版本的代理一起使用。该客户端可以与较早的代理进行通信（请参见Kafka文档），但是某些功能可能不可用。例如，对于低于0.11.xx的版本，不支持本机头。另外，0.11.xx不支持autoAddPartitions属性。

## **39.3配置选项**

本节包含Apache Kafka活页夹使用的配置选项。

有关与活页夹有关的常见配置选项和属性，请参阅[核心文档](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "binding-properties" \o "31.2绑定属性)。

### **39.3.1 Kafka Binder Properties**

spring.cloud.stream.kafka.binder.brokers

Kafka活页夹所连接的代理列表。

默认值：localhost。

spring.cloud.stream.kafka.binder.defaultBrokerPort

brokers允许指定带有或不带有端口信息的主机（例如，host1,host2:port2）。当代理列表中未配置任何端口时，这将设置默认端口。

默认值：9092。

spring.cloud.stream.kafka.binder.configuration

客户端属性（生产者和消费者）的键/值映射传递给绑定程序创建的所有客户端。由于生产者和消费者都使用了这些属性，因此应将使用限制为通用属性，例如安全性设置。Properties在这里取代引导中设置的所有属性。

默认值：空地图。

spring.cloud.stream.kafka.binder.consumerProperties

任意Kafka客户端使用者属性的键/值映射。这里的Properties取代了启动时和上面的configuration属性中设置的所有属性。

默认值：空地图。

spring.cloud.stream.kafka.binder.headers

活页夹传输的自定义标头列表。仅当与kafka-clients版本<0.11.0.0的旧版应用程序（⇐1.3.x）通信时才需要。较新的版本本机支持标头。

默认值：空。

spring.cloud.stream.kafka.binder.healthTimeout

等待获取分区信息的时间，以秒为单位。如果此计时器到期，运行状况将报告为已关闭。

默认值：10

spring.cloud.stream.kafka.binder.requiredAcks

代理程序上所需的确认数。有关生产者acks属性的信息，请参见Kafka文档。

默认值：1。

spring.cloud.stream.kafka.binder.minPartitionCount

仅在设置了autoCreateTopics或autoAddPartitions时有效。活页夹在生成或使用数据的主题上配置的全局最小分区数。可以通过生产者的partitionCount设置或生产者的instanceCount \* concurrency设置的值（如果任一个较大）来代替它。

默认值：1。

spring.cloud.stream.kafka.binder.producer属性

任意Kafka客户端生产者属性的键/值映射。这里的Properties取代了启动时和上面的configuration属性中设置的所有属性。

默认值：空地图。

spring.cloud.stream.kafka.binder.replicationFactor

如果autoCreateTopics有效，则自动创建的主题的复制因子。可以在每个绑定上覆盖。

默认值：1。

spring.cloud.stream.kafka.binder.autoCreateTopics

如果设置为true，则活页夹将自动创建新主题。如果设置为false，则活页夹依赖于已配置的主题。在后一种情况下，如果主题不存在，则活页夹无法启动。

|  |
| --- |
| IMG_405 |
| 此设置与代理的auto.topic.create.enable设置无关，并且不影响它。如果服务器设置为自动创建主题，则可以使用默认代理设置将它们作为元数据检索请求的一部分进行创建。 |

默认值：true。

spring.cloud.stream.kafka.binder.autoAddPartitions

如果设置为true，则活页夹将根据需要创建新分区。如果设置为false，则活页夹依赖于已配置的主题的分区大小。如果目标主题的分区数小于预期值，则活页夹无法启动。

默认值：false。

spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix

在活页夹中启用事务。请参阅Kafka文档中的transaction.id和spring-kafka文档中的[Transactions](https://docs.spring.io/spring-kafka/reference/html/_reference.html" \l "transactions" \t "https://www.springcloud.cc/_top)。启用事务后，将忽略各个producer属性，并且所有生产者都将使用spring.cloud.stream.kafka.binder.transaction.producer.\*属性。

默认值null（无交易）

spring.cloud.stream.kafka.binder.transaction.producer。\*

交易绑定中生产者的全球生产者属性。请参见spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix和[第39.3.3节“ Kafka生产者Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-producer-properties" \o "39.3.3 Kafka生产者属性)以及所有[活页夹](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-producer-properties" \o "39.3.3 Kafka Producer Properties)支持的常规生产者属性。

默认值：请参见各个生产者属性。

spring.cloud.stream.kafka.binder.headerMapperBeanName

KafkaHeaderMapper的bean名称，用于将spring-messaging标头映射到Kafka标头和从Kafka标头映射。例如，如果您希望自定义在标头中使用JSON反序列化的DefaultKafkaHeaderMapper中的受信任软件包，请使用此方法。

默认值：无。

### **39.3.2 Kafka消费者Properties**

以下属性仅适用于Kafka使用者，并且必须以spring.cloud.stream.kafka.bindings.<channelName>.consumer.为前缀。

管理员配置

供应主题时使用的Kafka主题属性中的Map（例如，spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0

默认值：无。

管理员副本分配

副本分配的Map <Integer，List <Integer >>，键为分区，值为分配。在配置新主题时使用。请参阅kafka-clients jar中的NewTopic Javadocs。

默认值：无。

管理员复制因子

设置主题时要使用的复制因子。覆盖活页夹范围的设置。忽略是否​​存在replicas-assignments。

默认值：无（使用资料夹范围的默认值1）。

autoRebalanceEnabled

当为true时，主题分区将在使用者组的成员之间自动重新平衡。当false时，将为每个使用者分配基于spring.cloud.stream.instanceCount和spring.cloud.stream.instanceIndex的固定分区集合。这要求在每个启动的实例上同时设置spring.cloud.stream.instanceCount和spring.cloud.stream.instanceIndex属性。在这种情况下，spring.cloud.stream.instanceCount属性的值通常必须大于1。

默认值：true。

ackEachRecord

当autoCommitOffset为true时，此设置指示在处理每条记录后是否提交偏移量。默认情况下，在处理consumer.poll()返回的记录批次中的所有记录之后，将提交偏移量。可以使用max.poll.records Kafka属性控制轮询返回的记录数，该属性是通过使用者configuration属性设置的。将此设置为true可能会导致性能下降，但是这样做会减少发生故障时重新传送记录的可能性。另外，请参见活页夹requiredAcks属性，该属性还影响落实偏移量的性能。

默认值：false。

autoCommitOffset

处理消息后是否自动提交偏移量。如果设置为false，则入站消息中将出现带有类型为org.springframework.kafka.support.Acknowledgment头的键kafka\_acknowledgment的头。应用程序可以使用此标头来确认消息。有关详细信息，请参见示例部分。当此属性设置为false时，Kafka活页夹将ack模式设置为org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL，应用程序负责确认记录。另请参阅ackEachRecord。

默认值：true。

autoCommitOnError

仅在autoCommitOffset设置为true时有效。如果设置为false，它将抑制导致错误的消息的自动提交，仅对成功的消息进行提交。如果持续出现故障，它允许流从上次成功处理的消息自动重播。如果设置为true，它将始终自动提交（如果启用了自动提交）。如果未设置（默认值），则其有效值与enableDlq相同，如果将错误消息发送到DLQ，则自动提交错误消息，否则不提交。

默认值：未设置。

resetOffsets

是否将使用者的偏移量重置为startOffset提供的值。

默认值：false。

startOffset

新组的起始偏移量。允许的值：earliest和latest。如果为消费者“绑定”显式设置了消费者组（通过spring.cloud.stream.bindings.<channelName>.group），则“ startOffset”设置为earliest。否则，将anonymous消费者组的值设置为latest。另请参见resetOffsets（在此列表的前面）。

默认值：null（等于earliest）。

enableDlq

设置为true时，它将为使用者启用DLQ行为。默认情况下，导致错误的消息将转发到名为error.<destination>.<group>的主题。可以通过设置dlqName属性来配置DLQ主题名称。对于错误数量相对较小并且重放整个原始主题可能太麻烦的情况，这为更常见的Kafka重播方案提供了一个替代选项。有关更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-dlq-processing" \o "39.6死信主题处理)请参见[第39.6节“ Dead-Letter主题处理”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-dlq-processing" \o "39.6 Dead-Letter Topic Processing)处理。从2.0版开始，发送到DLQ主题的消息将通过以下标头得到增强：x-original-topic，x-exception-message和x-exception-stacktrace为byte[]。 **当**destinationIsPattern**为**true**时不允许使用。**

默认值：false。

组态

使用包含通用Kafka使用者属性的键/值对进行映射。

默认值：空地图。

dlqName

接收错误消息的DLQ主题的名称。

默认值：null（如果未指定，则导致错误的消息将转发到名为error.<destination>.<group>的主题）。

dlqProducerProperties

使用此功能，可以设置特定于DLQ的生产者属性。通过kafka生产者属性可用的所有属性都可以通过此属性设置。

默认：默认Kafka生产者属性。

标头

指示入站通道适配器填充哪些标准头。允许的值：none，id，timestamp或both。如果使用本机反序列化并且第一个组件接收消息需要id（例如配置为使用JDBC消息存储的聚合器），则很有用。

默认值：none

converterBeanName

实现RecordMessageConverter的bean的名称。在入站通道适配器中用于替换默认的MessagingMessageConverter。

默认值：null

idleEventInterval

事件之间的间隔（以毫秒为单位），指示最近未接收到任何消息。使用ApplicationListener<ListenerContainerIdleEvent>接收这些事件。有关用法示例，请参见[“示例：暂停和恢复使用者”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pause-resume" \o "示例：暂停和恢复使用方)一节。

默认值：30000

destinationIsPattern

如果为true，则将目的地视为正则表达式Pattern，用于由代理匹配主题名称。设置为true时，不设置主题，并且不允许enableDlq，因为绑定者在设置阶段不知道主题名称。请注意，检测与模式匹配的新主题所花费的时间由消费者属性metadata.max.age.ms控制，该属性（在撰写本文时）默认为300,000ms（5分钟）。可以使用上面的configuration属性进行配置。

默认值：false

### **39.3.3 Kafka生产者Properties**

以下属性仅适用于Kafka生产者，并且必须以spring.cloud.stream.kafka.bindings.<channelName>.producer.为前缀。

管理员配置

预置新主题时使用的Kafka主题属性中的Map（例如，spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0

默认值：无。

管理员副本分配

副本分配的Map <Integer，List <Integer >>，键为分区，值为分配。在配置新主题时使用。请参见kafka-clients jar中的NewTopic javadocs。

默认值：无。

管理员复制因子

设置新主题时要使用的复制因子。覆盖活页夹范围的设置。如果存在replicas-assignments，则忽略。

默认值：无（使用资料夹范围的默认值1）。

缓冲区大小

Kafka生产者在发送之前尝试分批处理的数据量的上限（以字节为单位）。

默认值：16384。

同步

生产者是否同步。

默认值：false。

batchTimeout

生产者在发送消息之前等待允许更多消息在同一批中累积的时间。（通常，生产者根本不等待，仅发送在上一次发送过程中累积的所有消息。）非零值可能会增加吞吐量，但会增加延迟。

默认值：0。

messageKeyExpression

根据用于填充产生的Kafka消息的密钥的传出消息（例如，headers['myKey']）评估的SpEL表达式。有效负载无法使用，因为在评估此表达式时，有效负载已经采用byte[]的形式。

默认值：none。

headerPatterns

以逗号分隔的简单模式列表，以匹配要映射到ProducerRecord中的Kafka Headers的Spring消息头。模式可以以通配符（星号）开头或结尾。可以使用前缀!来否定模式。比赛在第一个比赛（正数或负数）之后停止。例如，!ask,as\*将传递ash，但不会传递ask。id和timestamp从未映射。

默认值：\*（所有标头-id和timestamp除外）

组态

使用包含通用Kafka生产者属性的键/值对进行映射。

默认值：空地图。

|  |
| --- |
| IMG_406 |
| Kafka活页夹使用生产者的partitionCount设置作为提示来创建具有给定分区数的主题（与minPartitionCount结合使用，两者中的最大值是所使用的值）。同时为活页夹配置minPartitionCount和为应用程序配置partitionCount时要小心，因为使用了较大的值。如果已经存在一个分区数较小的主题，并且禁用了autoAddPartitions（默认设置），则绑定器无法启动。如果已经存在一个分区数较小的主题，并且启用了autoAddPartitions，则会添加新的分区。如果已经存在的主题的分区数量大于最大数量（minPartitionCount或partitionCount），则使用现有分区计数。 |

### **39.3.4使用示例**

在本节中，我们将说明针对特定方案使用前面的属性。

#### **示例：将autoCommitOffset设置为false并依靠手动进行**

此示例说明了如何在用户应用程序中手动确认偏移。

本示例要求将spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset设置为false。在您的示例中使用相应的输入通道名称。

@SpringBootApplication

@EnableBinding(Sink.class)

public class ManuallyAcknowdledgingConsumer {

public static void main(String[] args) {

SpringApplication.run(ManuallyAcknowdledgingConsumer.class, args);

}

@StreamListener(Sink.INPUT)

public void process(Message<?> message) {

Acknowledgment acknowledgment = message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT, Acknowledgment.class);

if (acknowledgment != null) {

System.out.println("Acknowledgment provided");

acknowledgment.acknowledge();

}

}

}

#### **示例：安全配置**

Apache Kafka 0.9支持客户端和代理之间的安全连接。要利用此功能，请遵循[Apache Kafka文档](https://kafka.apache.org/090/documentation.html" \l "security_configclients" \t "https://www.springcloud.cc/_top)中的[准则以及Confluent文档中](http://docs.confluent.io/2.0.0/kafka/security.html" \t "https://www.springcloud.cc/_top)的Kafka 0.9 [安全准则](http://docs.confluent.io/2.0.0/kafka/security.html" \t "https://www.springcloud.cc/_top)。使用spring.cloud.stream.kafka.binder.configuration选项为活页夹创建的所有客户端设置安全性属性。

例如，要将security.protocol设置为SASL\_SSL，请设置以下属性：

spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_SSL

可以以类似方式设置所有其他安全属性。

使用Kerberos时，请遵循[参考文档](https://kafka.apache.org/090/documentation.html" \l "security_sasl_clientconfig" \t "https://www.springcloud.cc/_top)中的[说明](https://kafka.apache.org/090/documentation.html" \l "security_sasl_clientconfig" \t "https://www.springcloud.cc/_top)来创建和引用JAAS配置。

Spring Cloud Stream支持通过使用JAAS配置文件并使用Spring Boot属性将JAAS配置信息传递到应用程序。

##### **使用JAAS配置文件**

可以使用系统属性为Spring Cloud Stream应用程序设置JAAS和（可选）krb5文件位置。以下示例显示如何通过使用JAAS配置文件使用SASL和Kerberos启动Spring Cloud Stream应用程序：

java -Djava.security.auth.login.config=/path.to/kafka\_client\_jaas.conf -jar log.jar \

--spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \

--spring.cloud.stream.bindings.input.destination=stream.ticktock \

--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_PLAINTEXT

##### **使用Spring Boot Properties**

作为使用JAAS配置文件的替代方法，Spring Cloud Stream提供了一种通过使用Spring Boot属性为Spring Cloud Stream应用程序设置JAAS配置的机制。

以下属性可用于配置Kafka客户端的登录上下文：

spring.cloud.stream.kafka.binder.jaas.loginModule

登录模块名称。正常情况下无需设置。

默认值：com.sun.security.auth.module.Krb5LoginModule。

spring.cloud.stream.kafka.binder.jaas.controlFlag

登录模块的控制标志。

默认值：required。

spring.cloud.stream.kafka.binder.jaas.options

使用包含登录模块选项的键/值对进行映射。

默认值：空地图。

以下示例显示如何使用Spring Boot配置属性使用SASL和Kerberos启动Spring Cloud Stream应用程序：

java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \

--spring.cloud.stream.bindings.input.destination=stream.ticktock \

--spring.cloud.stream.kafka.binder.autoCreateTopics=false \

--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_PLAINTEXT \

--spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \

--spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \

--spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka\_client.keytab \

--spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client-1@EXAMPLE.COM

前面的示例表示以下JAAS文件的等效项：

KafkaClient {

com.sun.security.auth.module.Krb5LoginModule required

useKeyTab=true

storeKey=true

keyTab="/etc/security/keytabs/kafka\_client.keytab"

principal="kafka-client-1@EXAMPLE.COM";

};

如果所需的主题已经存在于代理上或将由管理员创建，则可以关闭自动创建，仅需要发送客户端JAAS属性。

|  |
| --- |
| IMG_407 |
| 请勿在同一应用程序中混合使用JAAS配置文件和Spring Boot属性。如果-Djava.security.auth.login.config系统属性已经存在，则Spring Cloud Stream将忽略Spring Boot属性。 |

|  |
| --- |
| IMG_408 |
| 将autoCreateTopics和autoAddPartitions与Kerberos一起使用时要小心。通常，应用程序可能使用在Kafka和Zookeeper中没有管理权限的主体。因此，依靠Spring Cloud Stream创建/修改主题可能会失败。在安全的环境中，强烈建议您使用Kafka工具创建主题并以管理方式管理ACL。 |

#### **示例：暂停和恢复使用方**

如果希望暂停使用但不引起分区重新平衡，则可以暂停并恢复使用方。通过将Consumer作为参数添加到@StreamListener中，可以简化此操作。要恢复，需要为ListenerContainerIdleEvent实例使用ApplicationListener。事件的发布频率由idleEventInterval属性控制。由于使用者不是线程安全的，因此必须在调用线程上调用这些方法。

以下简单的应用程序显示了如何暂停和恢复：

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

@StreamListener(Sink.INPUT)

**public** **void** in(String in, @Header(KafkaHeaders.CONSUMER) Consumer<?, ?> consumer) {

System.out.println(in);

consumer.pause(Collections.singleton(**new** TopicPartition("myTopic", 0)));

}

@Bean

**public** ApplicationListener<ListenerContainerIdleEvent> idleListener() {

**return** event -> {

System.out.println(event);

**if** (event.getConsumer().paused().size() > 0) {

event.getConsumer().resume(event.getConsumer().paused());

}

};

}

}

## **39.4错误通道**

从版本1.3开始，绑定程序无条件地将异常发送到每个使用者目标的错误通道，并且还可以配置为将异步生产者发送失败消息发送到错误通道。有关更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-error-handling" \o "29.4错误处理)请参见[第29.4节“错误处理”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-overview-error-handling" \o "29.4 Error Handling)。

发送失败的ErrorMessage的有效载荷是具有以下属性的KafkaSendFailureException：

* failedMessage：发送失败的Spring消息Message<?>。
* record：从failedMessage创建的原始ProducerRecord

没有对生产者异常的自动处理（例如发送到[Dead-Letter队列](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "kafka-dlq-processing" \o "39.6死信主题处理)）。您可以使用自己的Spring Integration流使用这些异常。

## **39.5 Kafka指标**

Kafka活页夹模块公开以下指标：

spring.cloud.stream.binder.kafka.offset：此度量标准指示给定的消费者组尚未从给定的活页夹主题中消费多少消息。提供的指标基于Mircometer指标库。度量标准包含消费者组信息，主题以及与主题上的最新偏移量有关的承诺偏移量的实际滞后时间。该指标对于向PaaS平台提供自动缩放反馈特别有用。

## **39.6 Dead-Letter主题处理**

因为您无法预期用户将如何处置死信，所以该框架没有提供任何标准机制来处理它们。如果死信的原因是暂时的，则您可能希望将消息路由回原始主题。但是，如果问题是永久性问题，则可能导致无限循环。本主题中的示例Spring Boot应用程序是如何将这些消息路由回原始主题的示例，但是在尝试了三遍之后，将其移至“ 停车场 ”主题。该应用程序是另一个从死信主题读取的spring-cloud-stream应用程序。5秒钟未收到任何消息时，它将终止。

这些示例假定原始目的地为so8400out，而使用者组为so8400。

有两种策略可供考虑：

* 考虑仅在主应用程序未运行时才运行重新路由。否则，瞬态错误的重试会很快用完。
* 或者，使用两阶段方法：使用此应用程序将路由到第三个主题，将另一个应用程序从那里路由回到主主题。

以下代码清单显示了示例应用程序：

**application.properties。**

spring.cloud.stream.bindings.input.group=so8400replay

spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out

spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot

spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries

**应用。**

@SpringBootApplication@EnableBinding(TwoOutputProcessor.class)**public** **class** ReRouteDlqKApplication **implements** CommandLineRunner {

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) {

SpringApplication.run(ReRouteDlqKApplication.**class**, args).close();

}

**private** **final** AtomicInteger processed = **new** AtomicInteger();

@Autowired

**private** MessageChannel parkingLot;

@StreamListener(Processor.INPUT)

@SendTo(Processor.OUTPUT)

**public** Message<?> reRoute(Message<?> failed) {

processed.incrementAndGet();

Integer retries = failed.getHeaders().get(X\_RETRIES\_HEADER, Integer.**class**);

**if** (retries == null) {

System.out.println("First retry for " + failed);

**return** MessageBuilder.fromMessage(failed)

.setHeader(X\_RETRIES\_HEADER, **new** Integer(1))

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build();

}

**else** **if** (retries.intValue() < 3) {

System.out.println("Another retry for " + failed);

**return** MessageBuilder.fromMessage(failed)

.setHeader(X\_RETRIES\_HEADER, **new** Integer(retries.intValue() + 1))

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build();

}

**else** {

System.out.println("Retries exhausted for " + failed);

parkingLot.send(MessageBuilder.fromMessage(failed)

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build());

}

**return** null;

}

@Override

**public** **void** run(String... args) **throws** Exception {

**while** (true) {

**int** count = **this**.processed.get();

Thread.sleep(5000);

**if** (count == **this**.processed.get()) {

System.out.println("Idle, terminating");

**return**;

}

}

}

**public** **interface** TwoOutputProcessor **extends** Processor {

@Output("parkingLot")

MessageChannel parkingLot();

}

}

## **39.7使用Kafka Binder进行分区**

Apache Kafka本机支持主题分区。

有时，将数据发送到特定的分区是有好处的-例如，当您要严格订购消息处理时（特定客户的所有消息应转到同一分区）。

以下示例显示了如何配置生产方和消费者方：

@SpringBootApplication@EnableBinding(Source.class)**public** **class** KafkaPartitionProducerApplication {

**private** **static** **final** Random RANDOM = **new** Random(System.currentTimeMillis());

**private** **static** **final** String[] data = **new** String[] {

"foo1", "bar1", "qux1",

"foo2", "bar2", "qux2",

"foo3", "bar3", "qux3",

"foo4", "bar4", "qux4",

};

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(KafkaPartitionProducerApplication.**class**)

.web(false)

.run(args);

}

@InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))

**public** Message<?> generate() {

String value = data[RANDOM.nextInt(data.length)];

System.out.println("Sending: " + value);

**return** MessageBuilder.withPayload(value)

.setHeader("partitionKey", value)

.build();

}

}

**application.yml。**

spring: cloud: stream: bindings: output: destination: partitioned.topic producer: partitioned: **true** partition-key-expression: headers['partitionKey'**]** partition-count: 12

|  |  |
| --- | --- |
| IMG_409 | **重要** |
| 必须为该主题提供足够的分区，以实现所有消费者组所需的并发性。上面的配置最多支持12个使用者实例（如果concurrency为2，则为6；如果并发值为3，则为4，依此类推）。通常最好“ 过量供应 ”分区，以使将来的使用者或并发性增加。 |

|  |
| --- |
| IMG_410 |
| 前面的配置使用默认分区（key.hashCode() % partitionCount）。根据键值，这可能会或可能不会提供适当的平衡算法。您可以使用partitionSelectorExpression或partitionSelectorClass属性覆盖此默认设置。 |

由于分区是由Kafka本地处理的，因此在使用者端不需要特殊配置。Kafka在实例之间分配分区。

以下Spring Boot应用程序侦听Kafka流，并打印（到控制台）每条消息去往的分区ID：

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** KafkaPartitionConsumerApplication {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(KafkaPartitionConsumerApplication.**class**)

.web(false)

.run(args);

}

@StreamListener(Sink.INPUT)

**public** **void** listen(@Payload String in, @Header(KafkaHeaders.RECEIVED\_PARTITION\_ID) **int** partition) {

System.out.println(in + " received from partition " + partition);

}

}

**application.yml。**

spring: cloud: stream: bindings: input: destination: partitioned.topic group: myGroup

您可以根据需要添加实例。Kafka重新平衡分区分配。如果实例计数（或instance count \* concurrency）超过了分区数，则某些使用者处于空闲状态。

## **40. Apache Kafka流Binder**

## **40.1使用**

要使用Kafka Streams活页夹，只需使用以下Maven坐标将其添加到Spring Cloud Stream应用程序中：

<dependency>

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

<artifactId>spring-cloud-stream-binder-kafka-streams</artifactId></dependency>

## **40.2 Kafka流Binder概述**

Spring Cloud Stream的Apache Kafka支持还包括为Apache Kafka流绑定显式设计的绑定器实现。通过这种本机集成，Spring Cloud Stream“处理器”应用程序可以直接在核心业务逻辑中使用 [Apache Kafka Streams](https://kafka.apache.org/documentation/streams/developer-guide" \t "https://www.springcloud.cc/_top) API。

Kafka Streams活页夹实现基于[Spring Kafka](https://docs.spring.io/spring-kafka/reference/html/_reference.html" \l "kafka-streams" \t "https://www.springcloud.cc/_top) 项目中的[Kafka Streams](https://docs.spring.io/spring-kafka/reference/html/_reference.html" \l "kafka-streams" \t "https://www.springcloud.cc/_top)提供的基础。

Kafka Streams绑定器为Kafka Streams中的三种主要类型（KStream，KTable和GlobalKTable）提供了绑定功能。

作为本机集成的一部分， Kafka Streams API提供的高级[Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html" \t "https://www.springcloud.cc/_top)可用于业务逻辑。

还提供[处理器API](https://docs.confluent.io/current/streams/developer-guide/processor-api.html" \t "https://www.springcloud.cc/_top)支持的早期版本。

如早期所述，Spring Cloud Stream中的Kafka流支持仅在处理器模型中严格可用。可以应用一种模型，在该模型中，可以从入站主题读取消息，进行业务处理，然后可以将转换后的消息写入出站主题。它也可以用于无出站目的地的处理器应用程序中。

### **40.2.1流DSL**

此应用程序消耗来自Kafka主题（例如words）的数据，在5秒的时间窗口内为每个唯一单词计算单词计数，并将计算结果发送到下游主题（例如counts）进行进一步处理。

@SpringBootApplication

@EnableBinding(KStreamProcessor.class)

public class WordCountProcessorApplication {

@StreamListener("input")

@SendTo("output")

public KStream<?, WordCount> process(KStream<?, String> input) {

return input

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

.groupBy((key, value) -> value)

.windowedBy(TimeWindows.of(5000))

.count(Materialized.as("WordCounts-multi"))

.toStream()

.map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()))));

}

public static void main(String[] args) {

SpringApplication.run(WordCountProcessorApplication.class, args);

}

一旦构建为uber-jar（例如wordcount-processor.jar），您就可以像下面一样运行上面的示例。

java -jar wordcount-processor.jar --spring.cloud.stream.bindings.input.destination=words --spring.cloud.stream.bindings.output.destination=counts

此应用程序将使用来自Kafka主题words的消息，并将计算的结果发布到输出主题counts。

Spring Cloud Stream将确保来自传入和传出主题的消息都自动绑定为KStream对象。作为开发人员，您可以专注于代码的业务方面，即编写处理器中所需的逻辑。框架自动处理Kafka Streams基础结构所需的Streams DSL特定配置的设置。

## **40.3配置选项**

本节包含Kafka Streams绑定程序使用的配置选项。

有关与活页夹有关的常见配置选项和属性，请参阅[核心文档](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "binding-properties" \o "31.2绑定属性)。

### **40.3.1 Kafka流Properties**

以下属性在活页夹级别可用，并且必须以spring.cloud.stream.kafka.streams.binder.文字作为前缀。

组态

使用包含与Apache Kafka Streams API有关的属性的键/值对进行映射。此属性必须以spring.cloud.stream.kafka.streams.binder.为前缀。以下是使用此属性的一些示例。

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde=org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde=org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms=1000

有关可能用于流配置的所有属性的更多信息，请参见Apache Kafka Streams文档中的StreamsConfig JavaDocs。

经纪人

经纪人网址

默认值：localhost

zkNodes

Zookeeper网址

默认值：localhost

serdeError

反序列化错误处理程序类型。可能的值为-logAndContinue，logAndFail或sendToDlq

默认值：logAndFail

applicationId

在绑定程序级别全局设置Kafka Streams应用程序的application.id的简便方法。如果应用程序包含多个StreamListener方法，则应在每个输入绑定的绑定级别上设置application.id。

默认值：none

以下属性仅适用于Kafka Streams生产者，并且必须以spring.cloud.stream.kafka.streams.bindings.<binding name>.producer.字面量为前缀。为方便起见，如果存在多个输出绑定并且它们都需要一个公共值，则可以使用前缀spring.cloud.stream.kafka.streams.default.producer.进行配置。

钥匙串

要使用的密钥序列

默认值：none。

valueSerde

使用价值服务

默认值：none。

useNativeEncoding

标志以启用本机编码

默认值：false。

以下属性仅适用于Kafka Streams使用者，并且必须以spring.cloud.stream.kafka.streams.bindings.<binding name>.consumer.`literal. For convenience, if there multiple input bindings and they all require a common value, that can be configured by using the prefix `spring.cloud.stream.kafka.streams.default.consumer.为前缀。

applicationId

设置每个输入绑定的application.id。

默认值：none

钥匙串

要使用的密钥序列

默认值：none。

valueSerde

使用价值服务

默认值：none。

物化为

使用传入的KTable类型实现状态存储

默认值：none。

useNativeDecoding

标志以启用本机解码

默认值：false。

dlqName

DLQ主题名称。

默认值：none。

### **40.3.2 TimeWindow属性：**

窗口化是流处理应用程序中的重要概念。以下属性可用于配置时间窗口计算。

spring.cloud.stream.kafka.streams.timeWindow.length

赋予此属性后，您可以将TimeWindows bean自动连接到应用程序中。该值以毫秒为单位。

默认值：none。

spring.cloud.stream.kafka.streams.timeWindow.advanceBy

值以毫秒为单位。

默认值：none。

## **40.4多个输入绑定**

对于需要多个传入KStream对象或KStream和KTable对象的组合的用例，Kafka Streams绑定程序提供了多个绑定支持。

让我们来看看它的作用。

### **40.4.1多个输入绑定作为接收器**

@EnableBinding(KStreamKTableBinding.class)

.....

.....

@StreamListener

public void process(@Input("inputStream") KStream<String, PlayEvent> playEvents,

@Input("inputTable") KTable<Long, Song> songTable) {

....

....

}

interface KStreamKTableBinding {

@Input("inputStream")

KStream<?, ?> inputStream();

@Input("inputTable")

KTable<?, ?> inputTable();

}

在上面的示例中，应用程序被编写为接收器，即没有输出绑定，并且应用程序必须决定有关下游处理的内容。当您以这种方式编写应用程序时，您可能希望向下游发送信息或将其存储在状态存储中（有关可查询状态存储，请参见下文）。

对于传入的KTable，如果要将计算具体化为状态存储，则必须通过以下属性将其表示。

spring.cloud.stream.kafka.streams.bindings.inputTable.consumer.materializedAs: all-songs

上面的示例显示了将KTable用作输入绑定。绑定器还支持GlobalKTable的输入绑定。当您必须确保应用程序的所有实例都可以访问主题中的数据更新时，GlobalKTable绑定非常有用。KTable和GlobalKTable绑定仅在输入上可用。Binder支持KStream的输入和输出绑定。

### **40.4.2作为处理器的多个输入绑定**

@EnableBinding(KStreamKTableBinding.class)

....

....

@StreamListener

@SendTo("output")

public KStream<String, Long> process(@Input("input") KStream<String, Long> userClicksStream,

@Input("inputTable") KTable<String, String> userRegionsTable) {

....

....

}

interface KStreamKTableBinding extends KafkaStreamsProcessor {

@Input("inputX")

KTable<?, ?> inputTable();

}

## **40.5多个输出绑定（又名分支）**

Kafka流允许根据某些谓词将出站数据分为多个主题。Kafka Streams绑定程序提供对此功能的支持，而不会损害最终用户应用程序中通过StreamListener公开的编程模型。

您可以按照上面在字数示例中展示的常用方法编写应用程序。但是，使用分支功能时，您需要做一些事情。首先，您需要确保您的返回类型为KStream[]，而不是常规的KStream。其次，您需要使用SendTo批注，该批注按顺序包含输出绑定（请参见下面的示例）。对于这些输出绑定中的每一个，您都需要配置目标，内容类型等，并符合标准Spring Cloud Stream的要求。

这是一个例子：

@EnableBinding(KStreamProcessorWithBranches.class)

@EnableAutoConfiguration

public static class WordCountProcessorApplication {

@Autowired

private TimeWindows timeWindows;

@StreamListener("input")

@SendTo({"output1","output2","output3})

public KStream<?, WordCount>[] process(KStream<Object, String> input) {

Predicate<Object, WordCount> isEnglish = (k, v) -> v.word.equals("english");

Predicate<Object, WordCount> isFrench = (k, v) -> v.word.equals("french");

Predicate<Object, WordCount> isSpanish = (k, v) -> v.word.equals("spanish");

return input

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

.groupBy((key, value) -> value)

.windowedBy(timeWindows)

.count(Materialized.as("WordCounts-1"))

.toStream()

.map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()))))

.branch(isEnglish, isFrench, isSpanish);

}

interface KStreamProcessorWithBranches {

@Input("input")

KStream<?, ?> input();

@Output("output1")

KStream<?, ?> output1();

@Output("output2")

KStream<?, ?> output2();

@Output("output3")

KStream<?, ?> output3();

}

}

Properties:

spring.cloud.stream.bindings.output1.contentType: application/json

spring.cloud.stream.bindings.output2.contentType: application/json

spring.cloud.stream.bindings.output3.contentType: application/json

spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms: 1000

spring.cloud.stream.kafka.streams.binder.configuration:

default.key.serde: org.apache.kafka.common.serialization.Serdes$StringSerde

default.value.serde: org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.bindings.output1:

destination: foo

producer:

headerMode: raw

spring.cloud.stream.bindings.output2:

destination: bar

producer:

headerMode: raw

spring.cloud.stream.bindings.output3:

destination: fox

producer:

headerMode: raw

spring.cloud.stream.bindings.input:

destination: words

consumer:

headerMode: raw

## **40.6消息转换**

与基于消息通道的活页夹应用程序类似，Kafka Streams活页夹可适应现成的内容类型转换，而不会做出任何妥协。

Kafka Streams操作通常会知道用于正确转换键和值的SerDe类型。因此，在入站和出站转换时依靠Apache Kafka Streams库本身提供的SerDe工具可能比使用框架提供的内容类型转换更为自然。另一方面，您可能已经熟悉框架提供的内容类型转换模式，并且您希望继续用于入站和出站转换。

Kafka Streams活页夹实现中都支持这两个选项。

### **40.6.1出站序列化**

如果禁用本机编码（这是默认设置），则框架将使用用户设置的contentType转换消息（否则，将应用默认的application/json）。在这种情况下，对于出站序列化，它将忽略出站上设置的任何SerDe。

这是在出站上设置contentType的属性。

spring.cloud.stream.bindings.output.contentType: application/json

这是启用本机编码的属性。

spring.cloud.stream.bindings.output.nativeEncoding: true

如果在输出绑定上启用了本机编码（用户必须如上所述明确地启用本机编码），则框架将在出站上跳过任何形式的自动消息转换。在这种情况下，它将切换到用户设置的Serde。将使用在实际输出绑定上设置的valueSerde属性。这是一个例子。

spring.cloud.stream.kafka.streams.bindings.output.producer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde

如果未设置此属性，则它将使用“默认” SerDe：spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde。

值得一提的是，Kafka Streams绑定程序不会在出站上序列化密钥-它仅依赖于Kafka本身。因此，您必须在绑定上指定keySerde属性，否则它将默认为应用程序范围的公用keySerde。

绑定级别密钥serde：

spring.cloud.stream.kafka.streams.bindings.output.producer.keySerde

公用密钥序列：

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde

如果使用分支，则需要使用多个输出绑定。例如，

interface KStreamProcessorWithBranches {

@Input("input")

KStream<?, ?> input();

@Output("output1")

KStream<?, ?> output1();

@Output("output2")

KStream<?, ?> output2();

@Output("output3")

KStream<?, ?> output3();

}

如果设置了nativeEncoding，则可以如下对各个输出绑定设置不同的SerDe。

spring.cloud.stream.kafka.streams.bindings.output1.producer.valueSerde=IntegerSerde

spring.cloud.stream.kafka.streams.bindings.output2.producer.valueSerde=StringSerde

spring.cloud.stream.kafka.streams.bindings.output3.producer.valueSerde=JsonSerde

然后，如果您具有SendTo这样的@SendTo（{“ output1”，“ output2”，“ output3”}），则分支中的KStream[]将应用上面定义的正确的SerDe对象。如果未启用nativeEncoding，则可以如下在输出绑定上设置不同的contentType值。在这种情况下，框架将使用适当的消息转换器在将消息发送到Kafka之前转换消息。

spring.cloud.stream.bindings.output1.contentType: application/json

spring.cloud.stream.bindings.output2.contentType: application/java-serialzied-object

spring.cloud.stream.bindings.output3.contentType: application/octet-stream

### **40.6.2入站反序列化**

类似的规则适用于入站数据反序列化。

如果禁用本机解码（这是默认设置），则框架将使用用户设置的contentType转换消息（否则，将应用默认的application/json）。在这种情况下，它将针对入站反序列化而忽略入站上设置的任何SerDe。

这是在入站上设置contentType的属性。

spring.cloud.stream.bindings.input.contentType: application/json

这是启用本机解码的属性。

spring.cloud.stream.bindings.input.nativeDecoding: true

如果在输入绑定上启用了本机解码（用户必须如上所述明确启用它），则框架将跳过对入站进行的任何消息转换。在这种情况下，它将切换到用户设置的SerDe。将使用在实际输出绑定上设置的valueSerde属性。这是一个例子。

spring.cloud.stream.kafka.streams.bindings.input.consumer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde

如果未设置此属性，它将使用默认的SerDe：spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde。

值得一提的是，Kafka Streams绑定程序不会反序列化入站的密钥-它仅依赖于Kafka本身。因此，您必须在绑定上指定keySerde属性，否则它将默认为应用程序范围的公用keySerde。

绑定级别密钥serde：

spring.cloud.stream.kafka.streams.bindings.input.consumer.keySerde

公用密钥序列：

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde

与在出站上进行KStream分支的情况一样，为每个绑定设置值SerDe的好处是，如果您有多个输入绑定（多个KStreams对象），并且它们都需要单独的值SerDe，则可以分别配置它们。如果使用通用配置方法，则此功能将不适用。

## **40.7错误处理**

Apache Kafka流提供了本机处理反序列化错误引起的异常的功能。有关该支持的详细信息，请参阅[本](https://cwiki.apache.org/confluence/display/KAFKA/KIP-161:+streams+deserialization+exception+handlers" \t "https://www.springcloud.cc/_top) 开箱，Apache Kafka流提供2种反序列化异常处理的- logAndContinue和logAndFail。顾名思义，前者将记录错误并继续处理下一条记录，而后者将记录错误并失败。LogAndFail是默认的反序列化异常处理程序。

### **40.7.1处理反序列化异常**

Kafka Streams活页夹通过以下属性支持选择异常处理程序。

spring.cloud.stream.kafka.streams.binder.serdeError: logAndContinue

除了上述两个反序列化异常处理程序之外，绑定程序还提供了第三个用于将错误记录（毒丸）发送到DLQ主题的代理。这是启用此DLQ异常处理程序的方法。

spring.cloud.stream.kafka.streams.binder.serdeError: sendToDlq

设置以上属性后，所有反序列化错误记录都会自动发送到DLQ主题。

spring.cloud.stream.kafka.streams.bindings.input.consumer.dlqName: foo-dlq

如果已设置，则错误记录将发送到主题foo-dlq。如果未设置，则它将创建名称为error.<input-topic-name>.<group-name>的DLQ主题。

在Kafka Streams活页夹中使用异常处理功能时，需要记住两件事。

* 属性spring.cloud.stream.kafka.streams.binder.serdeError适用于整个应用程序。这意味着如果同一应用程序中有多个StreamListener方法，则此属性将应用于所有这些方法。
* 反序列化的异常处理与本机反序列化和框架提供的消息转换一致。

### **40.7.2处理非反序列化异常**

对于Kafka Streams活页夹中的常规错误处理，最终用户应用程序要处理应用程序级错误。作为为反序列化异常处理程序提供DLQ的副作用，Kafka Streams绑定器提供了一种访问直接从应用程序发送bean的DLQ的方法。一旦可以访问该bean，就可以以编程方式将任何异常记录从应用程序发送到DLQ。

使用高级DSL仍然难以进行鲁棒的错误处理。Kafka Streams本身还不支持错误处理。

但是，当您在应用程序中使用低级处理器API时，有一些选项可以控制此行为。见下文。

@Autowired

private SendToDlqAndContinue dlqHandler;

@StreamListener("input")

@SendTo("output")

public KStream<?, WordCount> process(KStream<Object, String> input) {

input.process(() -> new Processor() {

ProcessorContext context;

@Override

public void init(ProcessorContext context) {

this.context = context;

}

@Override

public void process(Object o, Object o2) {

try {

.....

.....

}

catch(Exception e) {

//explicitly provide the kafka topic corresponding to the input binding as the first argument.

//DLQ handler will correctly map to the dlq topic from the actual incoming destination.

dlqHandler.sendToDlq("topic-name", (byte[]) o1, (byte[]) o2, context.partition());

}

}

.....

.....

});

}

## **40.8国立商店**

使用DSL时，Kafka流会自动创建状态存储。使用处理器API时，您需要手动注册状态存储。为此，您可以使用KafkaStreamsStateStore批注。您可以指定存储的名称和类型，用于控制日志和禁用缓存的标志等。一旦在引导阶段由绑定程序创建了存储，就可以通过处理器API访问此状态存储。下面是一些执行此操作的原语。

创建状态存储：

@KafkaStreamsStateStore(name="mystate", type= KafkaStreamsStateStoreProperties.StoreType.WINDOW, lengthMs=300000)

public void process(KStream<Object, Product> input) {

...

}

访问状态存储：

Processor<Object, Product>() {

WindowStore<Object, String> state;

@Override

public void init(ProcessorContext processorContext) {

state = (WindowStore)processorContext.getStateStore("mystate");

}

...

}

## **40.9交互式查询**

作为公共Kafka Streams绑定程序API的一部分，我们公开了一个名为InteractiveQueryService的类。您可以在应用程序中以Spring bean的身份进行访问。从您的应用程序访问此bean的一种简单方法是“自动装配” bean。

@Autowired

private InteractiveQueryService interactiveQueryService;

一旦获得对此bean的访问权限，就可以查询您感兴趣的特定状态存储。见下文。

ReadOnlyKeyValueStore<Object, Object> keyValueStore =

interactiveQueryService.getQueryableStoreType("my-store", QueryableStoreTypes.keyValueStore());

如果有多个Kafka Streams应用程序实例正在运行，则在以交互方式查询它们之前，您需要确定哪个应用程序实例承载密钥。InteractiveQueryService API提供了用于标识主机信息的方法。

为了使它起作用，必须按如下所示配置属性application.server：

spring.cloud.stream.kafka.streams.binder.configuration.application.server: <server>:<port>

以下是一些代码片段：

org.apache.kafka.streams.state.HostInfo hostInfo = interactiveQueryService.getHostInfo("store-name",

key, keySerializer);

if (interactiveQueryService.getCurrentHostInfo().equals(hostInfo)) {

//query from the store that is locally available

}

else {

//query from the remote host

}

## **40.10访问基础的KafkaStreams对象**

可以通过编程方式访问spring-kafka中负责构造KafkaStreams对象的StreamBuilderFactoryBean。每个StreamBuilderFactoryBean被注册为stream-builder，并附加了StreamListener方法名称。例如，如果您的StreamListener方法被命名为process，则流生成器bean被命名为stream-builder-process。由于这是工厂bean，因此在以编程方式访问它时，应在前面加上一个＆符号（&）进行访问。下面是一个示例，并假设StreamListener方法被命名为process

StreamsBuilderFactoryBean streamsBuilderFactoryBean = context.getBean("&stream-builder-process", StreamsBuilderFactoryBean.class);

KafkaStreams kafkaStreams = streamsBuilderFactoryBean.getKafkaStreams();

## **40.11状态清理**

默认情况下，绑定停止时将调用Kafkastreams.cleanup()方法。请参阅[Spring Kafka文档](https://docs.spring.io/spring-kafka/reference/html/_reference.html" \l "_configuration" \t "https://www.springcloud.cc/_top)。要修改此行为，只需向应用程序上下文中添加一个CleanupConfig @Bean（配置为在启动，停止或不清除时进行清除）即可；bean将被检测并连接到工厂bean。

## **41. RabbitMQ Binder**

## **41.1使用**

要使用RabbitMQ活页夹，您可以使用以下Maven坐标将其添加到Spring Cloud Stream应用程序中：

<dependency>

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

<artifactId>spring-cloud-stream-binder-rabbit</artifactId></dependency>

或者，您可以使用Spring Cloud Stream RabbitMQ入门程序，如下所示：

<dependency>

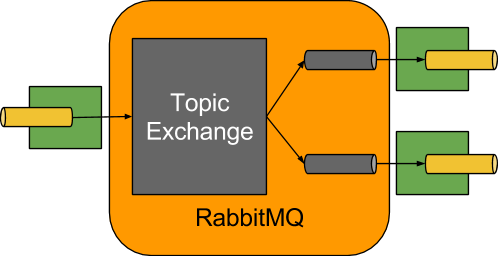
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId></dependency>

## **41.2 RabbitMQ Binder概述**

以下简化图显示了RabbitMQ绑定程序的工作方式：

**图41.1。RabbitMQ Binder**



默认情况下，RabbitMQ Binder实现将每个目的地映射到TopicExchange。对于每个消费者组，Queue绑定到该TopicExchange。每个使用者实例为其组的Queue有一个对应的RabbitMQ Consumer实例。对于分区的生产者和使用者，队列带有分区索引后缀，并将分区索引用作路由键。对于匿名使用者（没有group属性的使用者），将使用自动删除队列（具有随机唯一名称）。

通过使用可选的autoBindDlq选项，您可以配置活页夹以创建和配置死信队列（DLQ）（以及死信交换机DLX，以及路由基础结构）。默认情况下，死信队列具有目标名称，后跟.dlq。如果启用了重试（maxAttempts > 1），则在重试用尽后，失败的消息将传递到DLQ。如果禁用了重试（maxAttempts = 1），则应将requeueRejected设置为false（默认值），以便将失败的消息路由到DLQ，而不是重新排队。此外，republishToDlq使绑定程序将失败的消息发布到DLQ（而不是拒绝它）。通过此功能，可以将其他信息（例如x-exception-stacktrace标头中的堆栈跟踪）添加到标头中的消息中。此选项不需要启用重试。您只需尝试一次即可重新发布失败的消息。从1.2版开始，您可以配置重新发布邮件的传递模式。请参阅[republishDeliveryMode属性](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-stream-rabbit-republish-delivery-mode)。

|  |  |
| --- | --- |
| IMG_412 | **重要** |
| 将requeueRejected设置为true（与republishToDlq=false一起使用）会导致消息被重新排队并连续重新发送，除非失败的原因是短暂的，否则这可能不是您想要的。通常，应通过将maxAttempts设置为大于1或将republishToDlq设置为true在活页夹中启用重试。 |

有关这些属性的更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-binder-properties" \o "41.3.1 RabbitMQ活页夹属性)请参见[第41.3.1节“ RabbitMQ Binder Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-binder-properties" \o "41.3.1 RabbitMQ Binder Properties)。

该框架没有提供任何标准机制来使用死信消息（或将其重新路由回主队列）。[第41.6节“ Dead-Letter队列处理”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-dlq-processing" \o "41.6死信队列处理)中介绍了一些选项。

|  |
| --- |
| IMG_413 |
| 在Spring Cloud Stream应用程序中使用多个RabbitMQ活页夹时，重要的是禁用'RabbitAutoConfiguration'以避免将来自RabbitAutoConfiguration的相同配置应用于两个活页夹。您可以使用@SpringBootApplication批注来排除该类。 |

从版本2.0开始，RabbitMessageChannelBinder将RabbitTemplate.userPublisherConnection属性设置为true，以便非事务生成器避免对使用者的死锁，如果由于代理上的[内存警报](https://www.rabbitmq.com/memory.html" \t "https://www.springcloud.cc/_top)而阻止了缓存的连接，则可能发生死锁。

|  |
| --- |
| IMG_414 |
| 当前，multiplex使用者（一个使用者正在监听多个队列）仅受消息驱动的使用者支持；被轮询的使用者只能从单个队列中检索消息。 |

## **41.3配置选项**

本节包含特定于RabbitMQ Binder和绑定通道的设置。

有关常规绑定配置选项和属性的信息，请参见[Spring Cloud Stream核心文档](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-core-docs/src/main/asciidoc/spring-cloud-stream-overview.adoc" \l "configuration-options" \t "https://www.springcloud.cc/_top)。

### **41.3.1 RabbitMQ Binder Properties**

默认情况下，RabbitMQ活页夹使用Spring Boot的ConnectionFactory。一致地，它支持RabbitMQ的所有Spring Boot配置选项。（有关参考，请参见[Spring Boot文档](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \l "common-application-properties" \t "https://www.springcloud.cc/_top)）。RabbitMQ配置选项使用spring.rabbitmq前缀。

除Spring Boot选项外，RabbitMQ活页夹还支持以下属性：

spring.cloud.stream.rabbit.binder.adminAddresses

以逗号分隔的RabbitMQ管理插件URL列表。仅在nodes包含多个条目时使用。该列表中的每个条目都必须在spring.rabbitmq.addresses中具有一个对应的条目。仅当您使用RabbitMQ群集并希望从托管队列的节点使用时才需要。有关更多信息，请参见[队列亲和力和LocalizedQueueConnectionFactory](https://docs.spring.io/spring-amqp/reference/html/_reference.html" \l "queue-affinity" \t "https://www.springcloud.cc/_top)。

默认值：空。

spring.cloud.stream.rabbit.binder.nodes

RabbitMQ节点名称的逗号分隔列表。多个条目时，用于定位队列所在的服务器地址。此列表中的每个条目都必须在spring.rabbitmq.addresses中具有一个对应的条目。仅当您使用RabbitMQ群集并希望从托管队列的节点使用时才需要。有关更多信息，请参见[队列亲和力和LocalizedQueueConnectionFactory](https://docs.spring.io/spring-amqp/reference/html/_reference.html" \l "queue-affinity" \t "https://www.springcloud.cc/_top)。

默认值：空。

spring.cloud.stream.rabbit.binder.compressionLevel

压缩绑定的压缩级别。参见java.util.zip.Deflater。

默认值：1（BEST\_LEVEL）。

spring.cloud.stream.binder.connection-name-prefix

连接名称前缀，用于命名此绑定程序创建的连接。名称是此前缀，后跟#n，其中，每次打开新连接时，n都会递增。

默认值：无（Spring AMQP默认值）。

### **41.3.2 RabbitMQ消费者Properties**

以下属性仅适用于Rabbit使用者，并且必须以spring.cloud.stream.rabbit.bindings.<channelName>.consumer.为前缀。

确认模式

确认模式。

默认值：AUTO。

autoBindDlq

是否自动声明DLQ并将其绑定到绑定器DLX。

默认值：false。

bindingRoutingKey

用于将队列绑定到交换机的路由密钥（如果bindQueue为true）。对于分区的目的地，附加了-<instanceIndex>。

默认值：#。

bindQueue

是否将队列绑定到目标交换机。如果您已经设置了自己的基础结构并且先前已创建并绑定了队列，则将其设置为false。

默认值：true。

ConsumerTagPrefix

用于创建消费者标签；将被附加在#n之后，其中对于每个创建的使用者，n递增。例如：${spring.application.name}-${spring.cloud.stream.bindings.input.group}-${spring.cloud.stream.instance-index}。

默认值：无-经纪人将生成随机的消费者标签。

deadLetterQueueName

DLQ的名称

默认值：prefix+destination.dlq

deadLetterExchange

分配给队列的DLX。仅当autoBindDlq为true时才相关。

默认值：“ prefix + DLX”

deadLetterExchangeType

分配给队列的DLX的类型。仅当autoBindDlq为true时才相关。

默认值：“直接”

deadLetterRoutingKey

分配给队列的死信路由键。仅当autoBindDlq为true时才相关。

默认值：destination

声明Dlx

是否声明目的地交换死信。仅当autoBindDlq为true时才相关。如果您有预配置的DLX，请设置为false。

默认值：true。

声明交换

是否声明目的地交换。

默认值：true。

延迟交易

是否将交换声明为Delayed Message Exchange。需要代理上的延迟消息交换插件。x-delayed-type参数设置为exchangeType。

默认值：false。

dlqDeadLetterExchange

如果声明了DLQ，则分配给该队列的DLX。

默认值：none

dlqDeadLetterRoutingKey

如果声明了DLQ，则为该队列分配一个死信路由密钥。

默认值：none

dlqExpires

删除未使用的死信队列的时间（以毫秒为单位）。

默认值：no expiration

dlqLazy

用x-queue-mode=lazy参数声明死信队列。请参阅[“ 惰性队列 ”](https://www.rabbitmq.com/lazy-queues.html" \t "https://www.springcloud.cc/_top)。考虑使用策略而不是此设置，因为使用策略允许更改设置而不删除队列。

默认值：false。

dlqMaxLength

死信队列中的最大消息数。

默认值：no limit

dlqMaxLengthBytes

所有消息中的死信队列中的最大总字节数。

默认值：no limit

dlqMaxPriority

死信队列中消息的最大优先级（0-255）。

默认值：none

dlqOverflow行为

超过dlqMaxLength或dlqMaxLengthBytes时应采取的行动；当前为drop-head或reject-publish，但请参考RabbitMQ文档。

默认值：none

dlqTtl

声明时应用于死信队列的默认生存时间（以毫秒为单位）。

默认值：no limit

持久订阅

订阅是否应持久。仅当还设置了group时才有效。

默认值：true。

exchangeAutoDelete

如果declareExchange为true，则是否应自动删除交换（即在删除最后一个队列之后将其删除）。

默认值：true。

耐用

如果declareExchange为真，则该交换是否应该持久（即，在代理重新启动后仍然存在）。

默认值：true。

exchangeType

交换类型：非分区目的地为direct，fanout或topic，分区目的地为direct或topic。

默认值：topic。

独家

是否创建独家消费者。当为true时，并发应为1。通常在需要严格订购但使热备用实例在发生故障后接管时使用。请参阅recoveryInterval，该文件控制备用实例尝试使用的频率。

默认值：false。

过期

删除未使用的队列的时间（以毫秒为单位）。

默认值：no expiration

failedDeclarationRetryInterval

缺少队列的尝试之间的间隔（以毫秒为单位）。

默认值：5000

headerPatterns

从入站邮件映射标头的模式。

默认值：['\*']（所有标题）。

懒

使用x-queue-mode=lazy参数声明队列。请参阅[“ 惰性队列 ”](https://www.rabbitmq.com/lazy-queues.html" \t "https://www.springcloud.cc/_top)。考虑使用策略而不是此设置，因为使用策略允许更改设置而不删除队列。

默认值：false。

maxConcurrency

最大消费者数。

默认值：1。

最长长度

队列中的最大消息数。

默认值：no limit

maxLengthBytes

来自所有消息的队列中的最大总字节数。

默认值：no limit

maxPriority

队列中消息的最大优先级（0-255）。

默认值：none

missingQueuesFatal

当找不到队列时，是否将条件视为致命并停止侦听器容器。默认值为false，以便容器继续尝试从队列中使用数据，例如，在使用群集且承载非HA队列的节点关闭时。

默认值：false

溢出行为

超过maxLength或maxLengthBytes时应采取的行动；当前为drop-head或reject-publish，但请参考RabbitMQ文档。

默认值：none

预取

预取计数。

默认值：1。

字首

要添加到destination和队列名称的前缀。

默认值：“”。

queueDeclarationRetries

缺少队列时重试消耗的次数。仅当missingQueuesFatal为true时相关。否则，容器将无限期地重试。

默认值：3

queueNameGroupOnly

为true时，从名称等于group的队列中使用。否则，队列名称为destination.group。例如，在使用Spring Cloud Stream从现有RabbitMQ队列中消费时，这很有用。

默认值：false。

recoveryInterval

连接恢复尝试之间的时间间隔（以毫秒为单位）。

默认值：5000。

重新排队

禁用重试或republishToDlq为false时是否应该重新排定传送失败。

默认值：false。

republishDeliveryMode

当republishToDlq为true时，指定重新发布邮件的传递方式。

默认值：DeliveryMode.PERSISTENT

republishToDlq

默认情况下，拒绝重试后失败的消息将被拒绝。如果配置了死信队列（DLQ），则RabbitMQ将失败的消息（未更改）路由到DLQ。如果设置为true，则绑定程序将使用其他标头将失败的消息重新发布到DLQ，包括异常消息和来自最终失败原因的堆栈跟踪。

默认值：false

交易的

是否使用交易渠道。

默认值：false。

ttl

声明时应用于队列的默认生存时间（以毫秒为单位）。

默认值：no limit

txSize

两次之间的分娩次数。

默认值：1。

### **41.3.3高级侦听器容器配置**

要设置未公开为绑定程序或绑定属性的侦听器容器属性，请向应用程序上下文中添加类型为ListenerContainerCustomizer的单个bean。将设置活页夹和绑定属性，然后将调用定制程序。定制程序（configure()方法）提供了队列名称以及使用者组作为参数。

### **41.3.4 Rabbit生产者Properties**

以下属性仅适用于Rabbit生产者，并且必须以spring.cloud.stream.rabbit.bindings.<channelName>.producer.为前缀。

autoBindDlq

是否自动声明DLQ并将其绑定到绑定器DLX。

默认值：false。

batchingEnabled

是否启用生产者的邮件批处理。根据以下属性（在此列表的后三个条目中进行了描述），将消息批处理为一条消息：'batchSize'，batchBufferLimit和batchTimeout。有关更多信息，请参见[批处理](https://docs.spring.io/spring-amqp//reference/html/_reference.html" \l "template-batching" \t "https://www.springcloud.cc/_top)。

默认值：false。

batchSize

启用批处理时要缓冲的消息数。

默认值：100。

batchBufferLimit

启用批处理时的最大缓冲区大小。

默认值：10000。

batchTimeout

启用批处理时的批处理超时。

默认值：5000。

bindingRoutingKey

用于将队列绑定到交换机的路由密钥（如果bindQueue为true）。仅适用于未分区的目的地。仅在提供requiredGroups的情况下适用，然后仅对那些组有效。

默认值：#。

bindQueue

是否将队列绑定到目标交换机。如果您已经设置了自己的基础结构并且先前已创建并绑定了队列，则将其设置为false。仅在提供requiredGroups的情况下适用，然后仅对那些组有效。

默认值：true。

压缩

发送时是否应压缩数据。

默认值：false。

deadLetterQueueName

DLQ的名称仅在提供了requiredGroups之后才适用，然后仅适用于那些组。

默认值：prefix+destination.dlq

deadLetterExchange

分配给队列的DLX。仅当autoBindDlq为true时相关。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：“ prefix + DLX”

deadLetterExchangeType

分配给队列的DLX的类型。仅当autoBindDlq为true时才相关。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：“直接”

deadLetterRoutingKey

分配给队列的死信路由键。仅当autoBindDlq为true时相关。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：destination

声明Dlx

是否声明目的地交换死信。仅当autoBindDlq为true时才相关。如果您有预配置的DLX，请设置为false。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：true。

声明交换

是否声明目的地交换。

默认值：true。

delayExpression

一个SpEL表达式，用于评估应用于消息的延迟（x-delay标头）。如果交换不是延迟的消息交换，则无效。

默认值：未设置x-delay头。

延迟交易

是否将交换声明为Delayed Message Exchange。需要代理上的延迟消息交换插件。x-delayed-type参数设置为exchangeType。

默认值：false。

deliveryMode

交付方式。

默认值：PERSISTENT。

dlqDeadLetterExchange

声明DLQ后，将分配给该队列的DLX。仅在提供了requiredGroups之后才适用，然后仅对那些组适用。

默认值：none

dlqDeadLetterRoutingKey

声明DLQ后，将分配一个死信路由密钥给该队列。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：none

dlqExpires

删除未使用的死信队列之前的时间（以毫秒为单位）。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no expiration

dlqLazy

用x-queue-mode=lazy参数声明死信队列。请参阅[“ 惰性队列 ”](https://www.rabbitmq.com/lazy-queues.html" \t "https://www.springcloud.cc/_top)。考虑使用策略而不是此设置，因为使用策略允许更改设置而不删除队列。仅在提供requiredGroups时适用，然后仅适用于那些组。

dlqMaxLength

死信队列中的最大消息数。仅在提供了requiredGroups之后才适用，然后仅对那些组适用。

默认值：no limit

dlqMaxLengthBytes

所有消息中的死信队列中的最大总字节数。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no limit

dlqMaxPriority

死信队列中邮件的最大优先级（0-255）仅在提供requiredGroups时才适用，然后仅适用于那些组。

默认值：none

dlqTtl

声明时应用于死信队列的默认生存时间（以毫秒为单位）。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no limit

exchangeAutoDelete

如果declareExchange为true，则是否应自动删除交换（在删除最后一个队列之后将其删除）。

默认值：true。

耐用

如果declareExchange为true，则交换是否应该持久（在代理重新启动后生存）。

默认值：true。

exchangeType

交换类型：direct，fanout或topic用于未分区的目的地，direct或topic用于分区的目的地。

默认值：topic。

过期

删除未使用的队列之前的时间（以毫秒为单位）。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no expiration

headerPatterns

标头要映射到出站邮件的模式。

默认值：['\*']（所有标题）。

懒

使用x-queue-mode=lazy参数声明队列。请参阅[“ 惰性队列 ”](https://www.rabbitmq.com/lazy-queues.html" \t "https://www.springcloud.cc/_top)。考虑使用策略而不是此设置，因为使用策略允许更改设置而不删除队列。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：false。

最长长度

队列中的最大消息数。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no limit

maxLengthBytes

来自所有消息的队列中的最大总字节数。仅在提供requiredGroups的情况下适用，然后仅对那些组有效。

默认值：no limit

maxPriority

队列中消息的最大优先级（0-255）。仅在提供requiredGroups的情况下适用，然后仅对那些组有效。

默认值：none

字首

要添加到destination交换名称的前缀。

默认值：“”。

queueNameGroupOnly

当true时，从名称等于group的队列中使用。否则，队列名称为destination.group。例如，在使用Spring Cloud Stream从现有RabbitMQ队列中消费时，这很有用。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：false。

routingKeyExpression

一个SpEL表达式，用于确定发布消息时要使用的路由密钥。对于固定的路由键，请使用文字表达式，例如属性文件中的routingKeyExpression='my.routingKey'或YAML文件中的routingKeyExpression: '''my.routingKey'''。

默认值：destination或destination-<partition>（用于分区目标）。

交易的

是否使用交易渠道。

默认值：false。

ttl

声明时适用于队列的默认生存时间（以毫秒为单位）。仅在提供requiredGroups时适用，然后仅适用于那些组。

默认值：no limit

|  |
| --- |
| IMG_415 |
| 对于RabbitMQ，可以由外部应用程序设置内容类型标头。Spring Cloud Stream支持它们作为用于任何类型的传输的扩展内部协议的一部分—包括Kafka（0.11之前的版本）之类的传输，其本身不支持标头。 |

## **41.4重试RabbitMQ Binder**

当在活页夹中启用重试后，侦听器容器线程将在配置的任何退避期间暂停。当需要单个消费者严格订购时，这可能很重要。但是，对于其他用例，它阻止在该线程上处理其他消息。使用活页夹重试的另一种方法是设置带有时间的无效字母，以保留在无效字母队列（DLQ）中，并在DLQ本身上进行无效字母配置。有关此处讨论的属性的更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-binder-properties" \o "41.3.1 RabbitMQ Binder Properties)请参见“ [第41.3.1节“ RabbitMQ Binder Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-binder-properties" \o "41.3.1 RabbitMQ活页夹属性) ”。您可以使用以下示例配置来启用此功能：

* 将autoBindDlq设置为true。活页夹创建一个DLQ。（可选）您可以在deadLetterQueueName中指定一个名称。
* 将dlqTtl设置为您要在两次重新交付之间等待的退避时间。
* 将dlqDeadLetterExchange设置为默认交换。来自DLQ的过期消息被路由到原始队列，因为默认的deadLetterRoutingKey是队列名称（destination.group）。设置为默认交换是通过将属性设置为无值来实现的，如下例所示。

要强制对消息进行死信处理，请抛出AmqpRejectAndDontRequeueException或将requeueRejected设置为true（默认值）并抛出任何异常。

循环无休止地继续进行，这对于瞬态问题很好，但是您可能需要在尝试几次后放弃。幸运的是，RabbitMQ提供了x-death标头，可让您确定发生了多少个循环。

要在放弃后确认消息，请抛出ImmediateAcknowledgeAmqpException。

### **41.4.1整合在一起**

以下配置创建一个交换myDestination，其中队列myDestination.consumerGroup与通配符路由键#绑定到主题交换：

---

spring.cloud.stream.bindings.input.destination=myDestination

spring.cloud.stream.bindings.input.group=consumerGroup

#disable binder retries

spring.cloud.stream.bindings.input.consumer.max-attempts=1

#dlx/dlq setup

spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true

spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000

spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=

---

此配置将创建一个绑定到直接交换（DLX）的DLQ，其路由密钥为myDestination.consumerGroup。当邮件被拒绝时，它们将被路由到DLQ。5秒后，该消息到期，并通过使用队列名称作为路由键将其路由到原始队列，如以下示例所示：

**Spring Boot应用程序。**

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** XDeathApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(XDeathApplication.**class**, args);

}

@StreamListener(Sink.INPUT)

**public** **void** listen(String in, @Header(name = "x-death", required = false) Map<?,?> death) {

**if** (death != null && death.get("count").equals(3L)) {

*// giving up - don't send to DLX*

**throw** **new** ImmediateAcknowledgeAmqpException("Failed after 4 attempts");

}

**throw** **new** AmqpRejectAndDontRequeueException("failed");

}

}

请注意，x-death标头中的count属性是Long。

## **41.5错误通道**

从版本1.3开始，绑定程序无条件地将异常发送到每个使用者目标的错误通道，并且还可以配置为将异步生产者发送失败消息发送到错误通道。参见“ [???](https://www.springcloud.cc/spring-cloud-greenwich.html)”以获取更多信息。

RabbitMQ有两种类型的发送失败：

* 返回的消息，
* 负面认可的[出版商确认](https://www.rabbitmq.com/confirms.html" \t "https://www.springcloud.cc/_top)。

后者很少见。根据RabbitMQ文档，“只有在负责队列的Erlang进程中发生内部错误时，才会传递[nack]。”

除了启用生产者错误通道（如“ [???](https://www.springcloud.cc/spring-cloud-greenwich.html) ”中所述）之外，RabbitMQ绑定程序仅在正确配置连接工厂的情况下，才将消息发送到通道，如下所示：

* ccf.setPublisherConfirms(true);
* ccf.setPublisherReturns(true);

将Spring Boot配置用于连接工厂时，请设置以下属性：

* spring.rabbitmq.publisher-confirms
* spring.rabbitmq.publisher-returns

用于返回消息的ErrorMessage的有效负载是具有以下属性的ReturnedAmqpMessageException：

* failedMessage：发送失败的Spring消息Message<?>。
* amqpMessage：原始spring-amqp Message。
* replyCode：指示失败原因的整数值（例如312-无路由）。
* replyText：指示失败原因的文本值（例如，NO\_ROUTE）。
* exchange：发布消息的交易所。
* routingKey：发布消息时使用的路由密钥。

对于否定确认的确认，有效负载为NackedAmqpMessageException，具有以下属性：

* failedMessage：发送失败Spring消息Message<?>。
* nackReason：原因（如果有，您可能需要检查代理日志以获取更多信息）。

没有对这些异常的自动处理（例如发送到[死信队列](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "rabbit-dlq-processing" \o "41.6死信队列处理)）。您可以使用自己的Spring Integration流使用这些异常。

## **41.6 Dead-Letter队列处理**

因为您无法预期用户将如何处置死信，所以该框架没有提供任何标准机制来处理它们。如果死信的原因是暂时的，则您可能希望将消息路由回原始队列。但是，如果问题是永久性问题，则可能导致无限循环。以下Spring Boot应用程序显示了一个示例，该示例说明了如何将这些消息路由回原始队列，但在尝试三次后将其移至第三个“ 停车场 ”队列。第二个示例使用[RabbitMQ延迟消息交换](https://www.rabbitmq.com/blog/2015/04/16/scheduling-messages-with-rabbitmq/" \t "https://www.springcloud.cc/_top)为重新排队的消息引入延迟。在此示例中，每次尝试的延迟都会增加。这些示例使用@RabbitListener从DLQ接收消息。您也可以在批处理过程中使用RabbitTemplate.receive()。

这些示例假定原始目的地为so8400in，而消费者组为so8400。

### **41.6.1非分区目标**

前两个示例适用于目标**未**分区的情况：

@SpringBootApplication**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

@Autowired

**private** RabbitTemplate rabbitTemplate;

@RabbitListener(queues = DLQ)

**public** **void** rePublish(Message failedMessage) {

Integer retriesHeader = (Integer) failedMessage.getMessageProperties().getHeaders().get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

failedMessage.getMessageProperties().getHeaders().put(X\_RETRIES\_HEADER, retriesHeader + 1);

**this**.rabbitTemplate.send(ORIGINAL\_QUEUE, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

@Bean

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

@SpringBootApplication**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**private** **static** **final** String DELAY\_EXCHANGE = "dlqReRouter";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

@Autowired

**private** RabbitTemplate rabbitTemplate;

@RabbitListener(queues = DLQ)

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

headers.put("x-delay", 5000 \* retriesHeader);

**this**.rabbitTemplate.send(DELAY\_EXCHANGE, ORIGINAL\_QUEUE, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

@Bean

**public** DirectExchange delayExchange() {

DirectExchange exchange = **new** DirectExchange(DELAY\_EXCHANGE);

exchange.setDelayed(true);

**return** exchange;

}

@Bean

**public** Binding bindOriginalToDelay() {

**return** BindingBuilder.bind(**new** Queue(ORIGINAL\_QUEUE)).to(delayExchange()).with(ORIGINAL\_QUEUE);

}

@Bean

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

### **41.6.2分区目标**

对于分区目标，所有分区都有一个DLQ。我们根据标题确定原始队列。

#### **republishToDlq=false**

当republishToDlq为false时，RabbitMQ使用x-death标头将消息发布到DLX / DLQ，标头包含有关原始目的地的信息，如以下示例所示：

@SpringBootApplication**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_DEATH\_HEADER = "x-death";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

@Autowired

**private** RabbitTemplate rabbitTemplate;

@SuppressWarnings("unchecked")

@RabbitListener(queues = DLQ)

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X\_DEATH\_HEADER);

String exchange = (String) xDeath.get(0).get("exchange");

List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");

**this**.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

@Bean

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

#### **republishToDlq=true**

当republishToDlq为true时，重新发布的恢复程序将原始交换和路由密钥添加到标头中，如以下示例所示：

@SpringBootApplication**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**private** **static** **final** String X\_ORIGINAL\_EXCHANGE\_HEADER = RepublishMessageRecoverer.X\_ORIGINAL\_EXCHANGE;

**private** **static** **final** String X\_ORIGINAL\_ROUTING\_KEY\_HEADER = RepublishMessageRecoverer.X\_ORIGINAL\_ROUTING\_KEY;

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

@Autowired

**private** RabbitTemplate rabbitTemplate;

@RabbitListener(queues = DLQ)

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

String exchange = (String) headers.get(X\_ORIGINAL\_EXCHANGE\_HEADER);

String originalRoutingKey = (String) headers.get(X\_ORIGINAL\_ROUTING\_KEY\_HEADER);

**this**.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

@Bean

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

## **41.7使用RabbitMQ Binder进行分区**

RabbitMQ不支持本地分区。

有时，将数据发送到特定分区是有利的-例如，当您要严格订购消息处理时，特定客户的所有消息都应转到同一分区。

RabbitMessageChannelBinder通过将每个分区的队列绑定到目标交换机来提供分区。

以下Java和YAML示例显示了如何配置生产者：

**制片人。**

@SpringBootApplication@EnableBinding(Source.class)**public** **class** RabbitPartitionProducerApplication {

**private** **static** **final** Random RANDOM = **new** Random(System.currentTimeMillis());

**private** **static** **final** String[] data = **new** String[] {

"abc1", "def1", "qux1",

"abc2", "def2", "qux2",

"abc3", "def3", "qux3",

"abc4", "def4", "qux4",

};

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(RabbitPartitionProducerApplication.**class**)

.web(false)

.run(args);

}

@InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))

**public** Message<?> generate() {

String value = data[RANDOM.nextInt(data.length)];

System.out.println("Sending: " + value);

**return** MessageBuilder.withPayload(value)

.setHeader("partitionKey", value)

.build();

}

}

**application.yml。**

spring: cloud: stream: bindings: output: destination: partitioned.destination producer: partitioned: **true** partition-key-expression: headers['partitionKey'**]** partition-count: 2 required-groups:

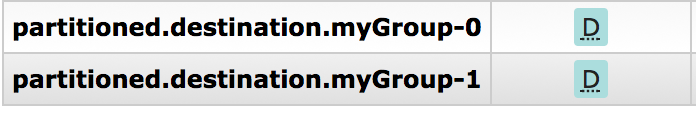
- myGroup

|  |
| --- |
| IMG_416 |
| 前例中的配置使用默认分区（key.hashCode() % partitionCount）。根据键值，这可能会或可能不会提供适当的平衡算法。您可以使用partitionSelectorExpression或partitionSelectorClass属性覆盖此默认设置。  仅当在部署生产者时需要提供消费者队列时，才需要required-groups属性。否则，发送到分区的所有消息都将丢失，直到部署了相应的使用者为止。 |

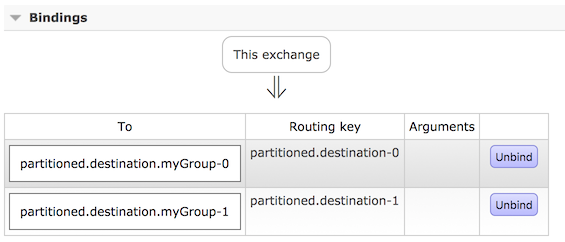
以下配置提供了主题交换：

IMG_417

该交换绑定了以下队列：



以下绑定将队列与交换关联：



以下Java和YAML示例继续了前面的示例，并显示了如何配置使用者：

**消费者。**

@SpringBootApplication@EnableBinding(Sink.class)**public** **class** RabbitPartitionConsumerApplication {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(RabbitPartitionConsumerApplication.**class**)

.web(false)

.run(args);

}

@StreamListener(Sink.INPUT)

**public** **void** listen(@Payload String in, @Header(AmqpHeaders.CONSUMER\_QUEUE) String queue) {

System.out.println(in + " received from queue " + queue);

}

}

**application.yml。**

spring: cloud: stream: bindings: input: destination: partitioned.destination group: myGroup consumer: partitioned: **true** instance-index: 0

|  |  |
| --- | --- |
| IMG_420 | **重要** |
| RabbitMessageChannelBinder不支持动态缩放。每个分区至少必须有一个使用方。使用者的instanceIndex用于指示使用了哪个分区。诸如Cloud Foundry之类的平台只能具有一个带有instanceIndex的实例。 |

# **第七部分 Spring Cloud Bus**

Spring Cloud Bus用轻量级消息代理链接分布式系统的节点。然后可以使用此代理来广播状态更改（例如配置更改）或其他管理指令。一个关键的想法是，该总线就像用于Spring Boot应用的分布式致动器，可以横向扩展。但是，它也可以用作应用之间的通信渠道。该项目为AMQP经纪人或Kafka作为传输者提供了入门者。

|  |
| --- |
| IMG_421 |
| Spring Cloud是根据非限制性Apache 2.0许可证发行的。如果您想为文档的这一部分做出贡献或发现错误，请在[github](https://github.com/spring-cloud/spring-cloud-config/tree/master/docs/src/main/asciidoc" \t "https://www.springcloud.cc/_top)的项目中找到源代码和问题跟踪程序。 |

## **42.快速入门**

如果Spring Cloud Bus在类路径中检测到自身，则通过添加Spring Boot autconfiguration来工作。要启用总线，请将spring-cloud-starter-bus-amqp或spring-cloud-starter-bus-kafka添加到依赖管理中。Spring Cloud负责其余的工作。确保代理（RabbitMQ或Kafka）可用且已配置。在本地主机上运行时，您无需执行任何操作。如果是远程运行，请使用Spring Cloud连接器或Spring Boot约定来定义代理凭据，如以下Rabbit的示例所示：

**application.yml。**

spring:

rabbitmq:

host: mybroker.com

port: 5672

username: user

password: secret

总线当前支持将消息发送到侦听的所有节点或特定服务的所有节点（由Eureka定义）。/bus/\*执行器名称空间具有一些HTTP端点。当前，有两个已实现。第一个/bus/env发送键/值对以更新每个节点的Spring环境。第二个/bus/refresh重新加载每个应用程序的配置，就好像它们都已在其/refresh端点上被ping一样。

|  |
| --- |
| IMG_422 |
| Spring Cloud Bus入门者介绍了Rabbit和Kafka，因为这是两个最常见的实现。但是，Spring Cloud Stream非常灵活，并且活页夹可与spring-cloud-bus一起使用。 |

## **43. Bus端点**

Spring Cloud Bus提供两个端点/actuator/bus-refresh和/actuator/bus-env，分别对应于Spring Cloud Commons，/actuator/refresh和/actuator/env中的各个执行器端点。

## **43.1 Bus刷新端点**

/actuator/bus-refresh端点清除RefreshScope缓存并重新绑定@ConfigurationProperties。有关更多信息，请参见[刷新作用域](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "refresh-scope" \o "2.9刷新范围)文档。

要公开/actuator/bus-refresh端点，您需要在应用程序中添加以下配置：

management.endpoints.web.exposure.include=bus-refresh

## **43.2 Bus环保端点**

/actuator/bus-env端点使用跨多个实例的指定键/值对更新每个实例环境。

要公开/actuator/bus-env端点，您需要在应用程序中添加以下配置：

management.endpoints.web.exposure.include=bus-env

/actuator/bus-env端点接受具有以下形状的POST请求：

**{**

"name": "key1"**,**

"value": "value1"**}**

## **44.寻址实例**

应用程序的每个实例都有一个服务ID，该服务ID的值可以用spring.cloud.bus.id设置，并且其值应按冒号分隔的标识符列表（从最小到最具体）排列。默认值是根据环境构造的，它是spring.application.name和server.port（或spring.application.index，如果已设置）的组合。ID的默认值以app:index:id的形式构造，其中：

* app是vcap.application.name（如果存在），或者是spring.application.name
* index是vcap.application.instance\_index（如果存在），依次为spring.application.index，local.server.port，server.port或0。
* id是vcap.application.instance\_id（如果存在）或随机值。

HTTP端点接受“ 目的地 ”路径参数，例如/bus-refresh/customers:9000，其中destination是服务ID。如果该ID由总线上的一个实例拥有，它将处理该消息，而所有其他实例将忽略它。

## **45.解决服务的所有实例**

的“ 目的地 ”参数中使用的Spring PathMatcher（与路径分隔作为结肠- :），以确定是否一个实例处理该消息。使用前面的示例，/bus-env/customers:\*\*定位“ 客户 ”服务的所有实例， 而与其余服务ID无关。

## **46.服务ID必须唯一**

总线尝试两次消除处理事件，一次来自原始ApplicationEvent，一次来自队列。为此，它将对照当前服务ID检查发送服务ID。如果服务的多个实例具有相同的ID，则不会处理事件。在本地计算机上运行时，每个服务都在不同的端口上，并且该端口是ID的一部分。Cloud Foundry提供了区分索引。为确保ID在Cloud Foundry之外是唯一的，请为每个服务实例将spring.application.index设置为唯一。

## **47.定制消息代理**

Spring Cloud Bus使用[Spring Cloud Stream](https://cloud.spring.io/spring-cloud-stream" \t "https://www.springcloud.cc/_top)广播消息。因此，要使消息流动，您只需要在类路径中包括您选择的活页夹实现即可。对于AMQP（RabbitMQ）和Kafka（spring-cloud-starter-bus-[amqp|kafka]）的公交车来说，起动器很方便。一般来说，Spring Cloud Stream依赖于Spring Boot自动配置约定来配置中间件。例如，可以使用spring.rabbitmq.\*配置属性来更改AMQP代理地址。Spring Cloud Bus在spring.cloud.bus.\*中具有一些本机配置属性（例如，spring.cloud.bus.destination是用作外部中间件的主题的名称）。通常，默认值就足够了。

要了解有关如何自定义消息代理设置的更多信息，请参阅Spring Cloud流文档。

## **48.追踪Bus Events**

可以通过设置spring.cloud.bus.trace.enabled=true来跟踪Bus事件（RemoteApplicationEvent的子类）。如果这样做，Spring Boot TraceRepository（如果存在）将显示发送的每个事件以及每个服务实例的所有确认。以下示例来自/trace端点：

**{**

"timestamp": "2015-11-26T10:24:44.411+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.ack"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "stores:8081"**,**

"destination": "\*:\*\*"

**}**

**},**

**{**

"timestamp": "2015-11-26T10:24:41.864+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.sent"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "customers:9000"**,**

"destination": "\*:\*\*"

**}**

**},**

**{**

"timestamp": "2015-11-26T10:24:41.862+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.ack"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "customers:9000"**,**

"destination": "\*:\*\*"

**}}**

前面的迹线显示RefreshRemoteApplicationEvent从customers:9000发送，广播到所有服务，并由customers:9000和stores:8081接收（确认）。

要自己处理确认信号，您可以为应用程序中的AckRemoteApplicationEvent和SentApplicationEvent类型添加一个@EventListener（并启用跟踪）。或者，您可以点击TraceRepository并从那里挖掘数据。

|  |
| --- |
| IMG_423 |
| 任何Bus应用程序都可以跟踪acks。但是，有时，在中央服务中执行此操作很有用，该服务可以对数据进行更复杂的查询，或将其转发给专门的跟踪服务。 |

## **49.广播自己的Events**

Bus可以携带RemoteApplicationEvent类型的任何事件。默认传输是JSON，解串器需要提前知道将要使用哪些类型。要注册新类型，必须将其放在org.springframework.cloud.bus.event的子包中。

要自定义事件名称，可以在自定义类上使用@JsonTypeName或依赖默认策略，即使用类的简单名称。

|  |
| --- |
| IMG_424 |
| 生产者和消费者都需要访问类定义。 |

## **49.1在自定义程序包中注册事件**

如果您不能或不想将org.springframework.cloud.bus.event的子包用于自定义事件，则必须使用@RemoteApplicationEventScan批注指定要扫描哪些包来扫描类型为RemoteApplicationEvent的事件。用@RemoteApplicationEventScan指定的软件包包括子软件包。

例如，考虑以下自定义事件，称为MyEvent：

**package** com.acme;

**public** **class** MyEvent **extends** RemoteApplicationEvent {

...

}

您可以通过以下方式在反序列化器中注册该事件：

**package** com.acme;

@Configuration@RemoteApplicationEventScan**public** **class** BusConfiguration {

...

}

不指定值，将注册使用@RemoteApplicationEventScan的类的包。在本示例中，使用包BusConfiguration注册了com.acme。

您还可以通过使用@RemoteApplicationEventScan上的value，basePackages或basePackageClasses属性来明确指定要扫描的软件包，如以下示例所示：

**package** com.acme;

@Configuration*//@RemoteApplicationEventScan({"com.acme", "foo.bar"})//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})*@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)**public** **class** BusConfiguration {

...

}

@RemoteApplicationEventScan的所有上述示例都是等效的，因为com.acme软件包是通过在@RemoteApplicationEventScan上显式指定软件包来注册的。

|  |
| --- |
| IMG_425 |
| 您可以指定要扫描的多个基本软件包。 |

# **第八部分 Spring Cloud Sleuth**

阿德里安·科尔（Adrian Cole），斯宾塞·吉布（Spencer Gibb），马辛·格热兹扎克（Marcin Grzejszczak），戴夫·瑟（Dave Syer），杰伊·布莱恩特（Jay Bryant）

**Greenwich SR5**

## **50.引言**

Spring Cloud Sleuth为[Spring Cloud](https://cloud.spring.io/" \t "https://www.springcloud.cc/_top)实现了分布式跟踪解决方案。

## **50.1术语**

Spring Cloud Sleuth借鉴了[Dapper的](https://research.google.com/pubs/pub36356.html" \t "https://www.springcloud.cc/_top)术语。

**Span**：基本工作单元。例如，发送RPC是一个新的跨度，就像发送响应到RPC一样。跨度由跨度的唯一64位ID和跨度所属的跟踪的另一个64位ID标识。跨区还具有其他数据，例如描述，带有时间戳的事件，键值注释（标签），引起跨度的跨区ID和进程ID（通常为IP地址）。

跨度可以启动和停止，并且可以跟踪其时序信息。创建跨度后，您必须在将来的某个时间点将其停止。

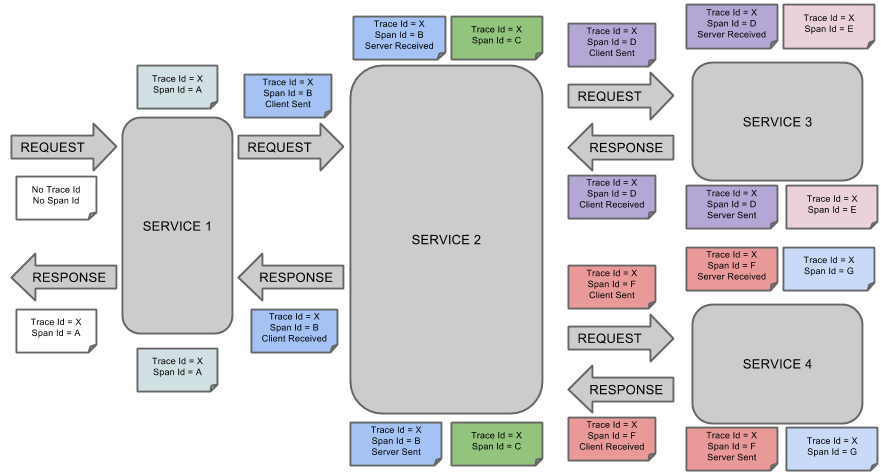
|  |
| --- |
| IMG_426 |
| 开始跟踪的初始跨度称为root span。该跨度的ID的值等于跟踪ID。 |

**迹线：**一组spans，形成树状结构。例如，如果您运行分布式大数据存储，则跟踪可能由PUT请求形成。

**注释：**用于及时记录事件的存在。使用 [Brave](https://github.com/openzipkin/brave" \t "https://www.springcloud.cc/_top)工具，我们不再需要为[Zipkin](https://zipkin.io/" \t "https://www.springcloud.cc/_top)设置特殊事件来了解客户端和服务器是谁，请求在哪里开始以及在哪里结束。但是，出于学习目的，我们标记这些事件以突出显示发生了哪种操作。

* **cs**：客户端已发送。客户提出了要求。此注释指示跨度的开始。
* **sr**：接收到服务器：服务器端收到了请求并开始处理它。从此时间戳中减去cs时间戳可显示网络延迟。
* **ss**：服务器已发送。在请求处理完成时进行注释（当响应被发送回客户端时）。从此时间戳中减去sr时间戳将显示服务器端处理请求所需的时间。
* **cr**：收到客户。表示跨度结束。客户端已成功收到服务器端的响应。从此时间戳中减去cs时间戳将显示客户端从服务器接收响应所需的整个时间。

下图显示了**Span**和**Trace**在系统中的外观以及Zipkin批注：



音符的每种颜色都表示一个跨度（从**A**到**G**共有七个spans- ）。请考虑以下注意事项：

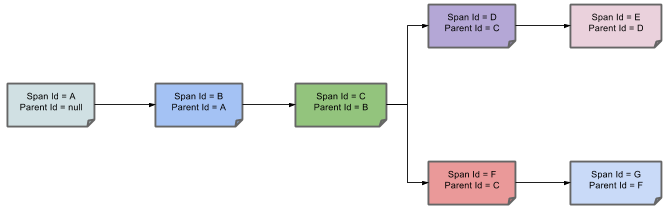
Trace Id = X

Span Id = D

Client Sent

该说明指出，当前跨距**跟踪编号**设定为**X**和**Span标识**设置为**d**。同样，发生了Client Sent事件。

下图显示了spans的父子关系：

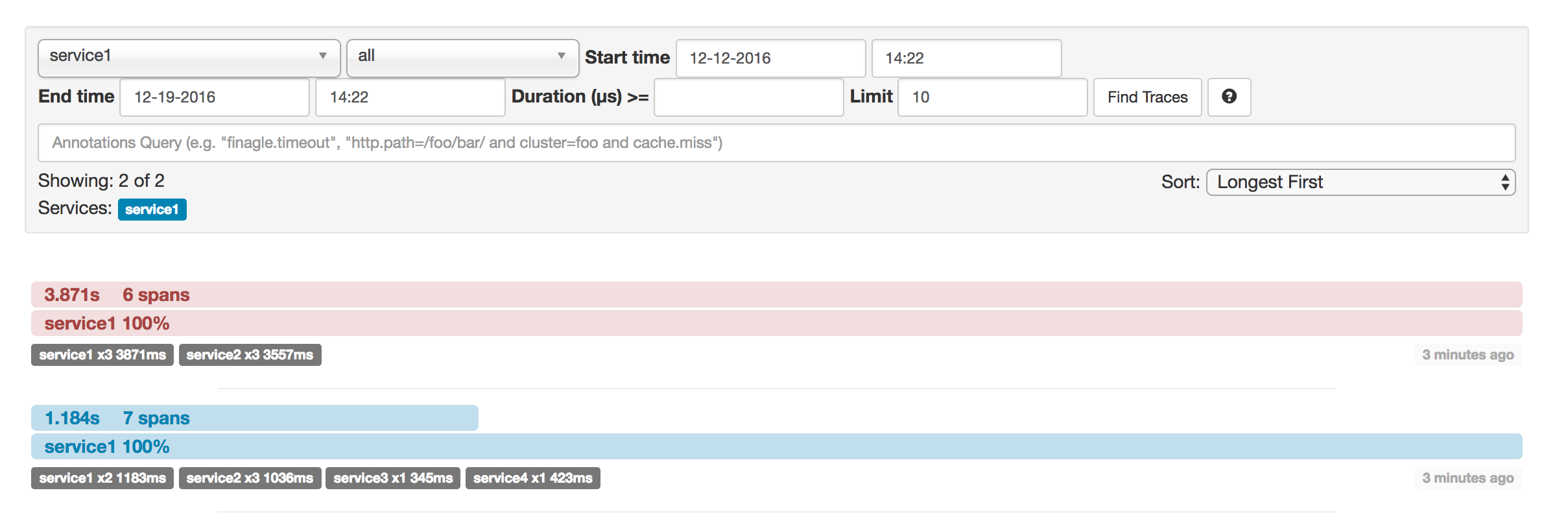


## **50.2目的**

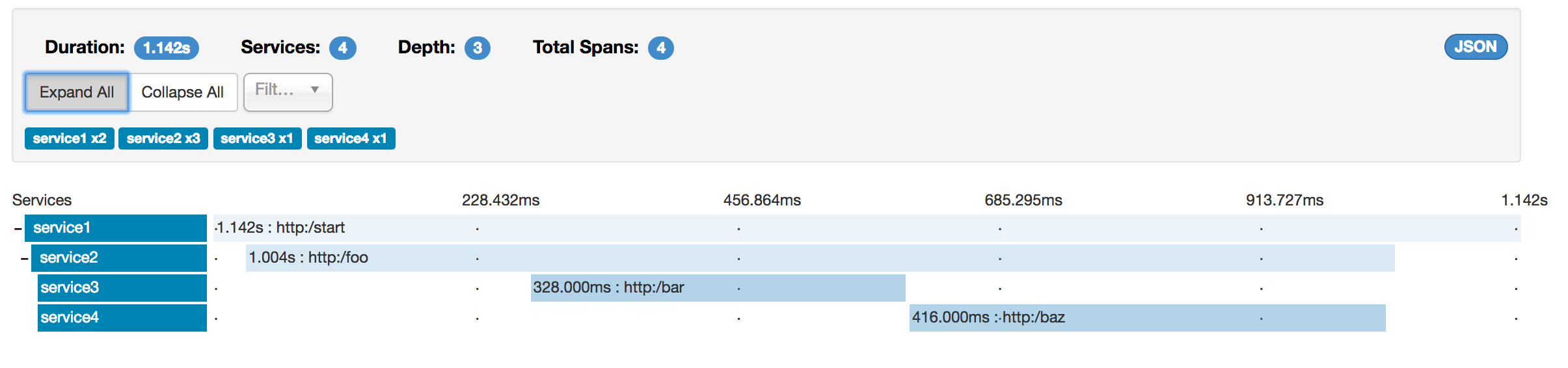
以下各节引用上图中显示的示例。

### **50.2.1使用Zipkin进行分布式跟踪**

本示例有七个spans。如果转到Zipkin中的跟踪，则可以在第二个跟踪中看到此数字，如下图所示：



但是，如果选择特定的跟踪，则可以看到四个spans，如下图所示：



|  |
| --- |
| IMG_431 |
| 选择特定跟踪时，您会看到合并的spans。这意味着，如果有两个spans发送到Zipkin，并且带有“服务器已接收和服务器已发送”或“客户端已接收和客户端已发送”注释，它们将显示为单个跨度。 |

在这种情况下，为什么七个spans和四个spans之间有区别？

* 一个跨度来自http:/start跨度。它具有服务器已接收（sr）和服务器已发送（ss）批注。
* 从service1到service2到http:/foo端点的RPC调用中有两个spans。客户发送（cs）和客户接收（cr）事件在service1端发生。服务器已接收（sr）和服务器已发送（ss）事件在service2端发生。这两个spans构成一个与RPC调用相关的逻辑范围。
* 从service2到service3到http:/bar端点的RPC调用中有两个spans。客户发送（cs）和客户接收（cr）事件在service2端发生。服务器已接收（sr）和服务器已发送（ss）事件在service3端发生。这两个spans构成一个与RPC调用相关的逻辑范围。
* 从service2到service4到http:/baz端点的RPC调用中有两个spans。客户发送（cs）和客户接收（cr）事件在service2端发生。服务器已收到（sr）和服务器已发送（ss）事件在service4端发生。这两个spans构成一个与RPC调用相关的逻辑范围。

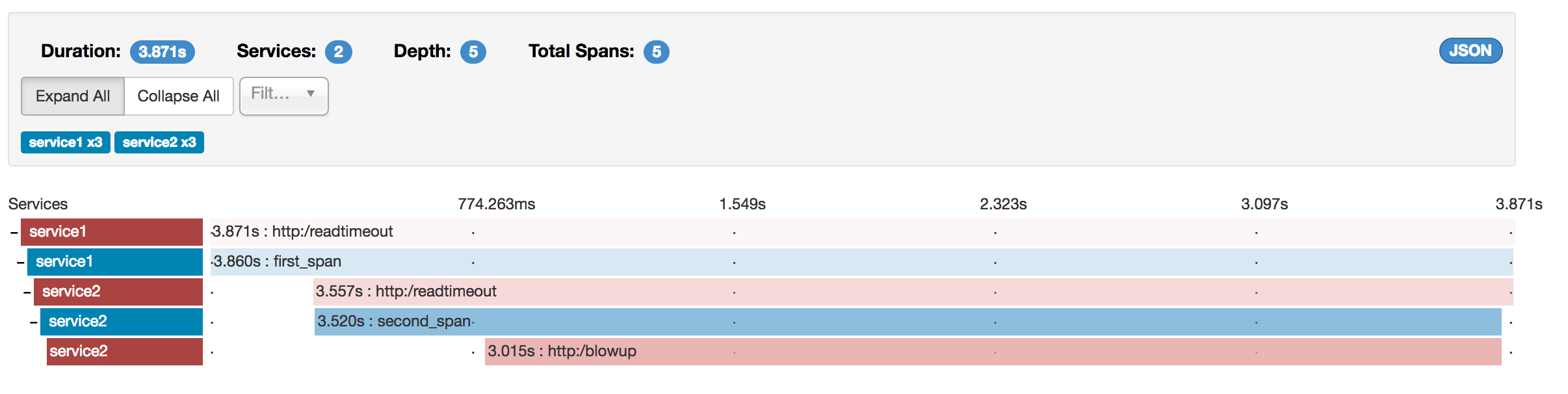
因此，如果我们计算物理量spans，则有一个来自http:/start的实体，两个来自service1的实体，调用service2，两个来自service2的实体，调用service3，还有两个来自service2致电service4。总而言之，我们总共有七个spans。

从逻辑上讲，我们看到四个跨度的信息，因为我们有一个跨度与到service1的传入请求有关，而有三个spans与RPC调用有关。

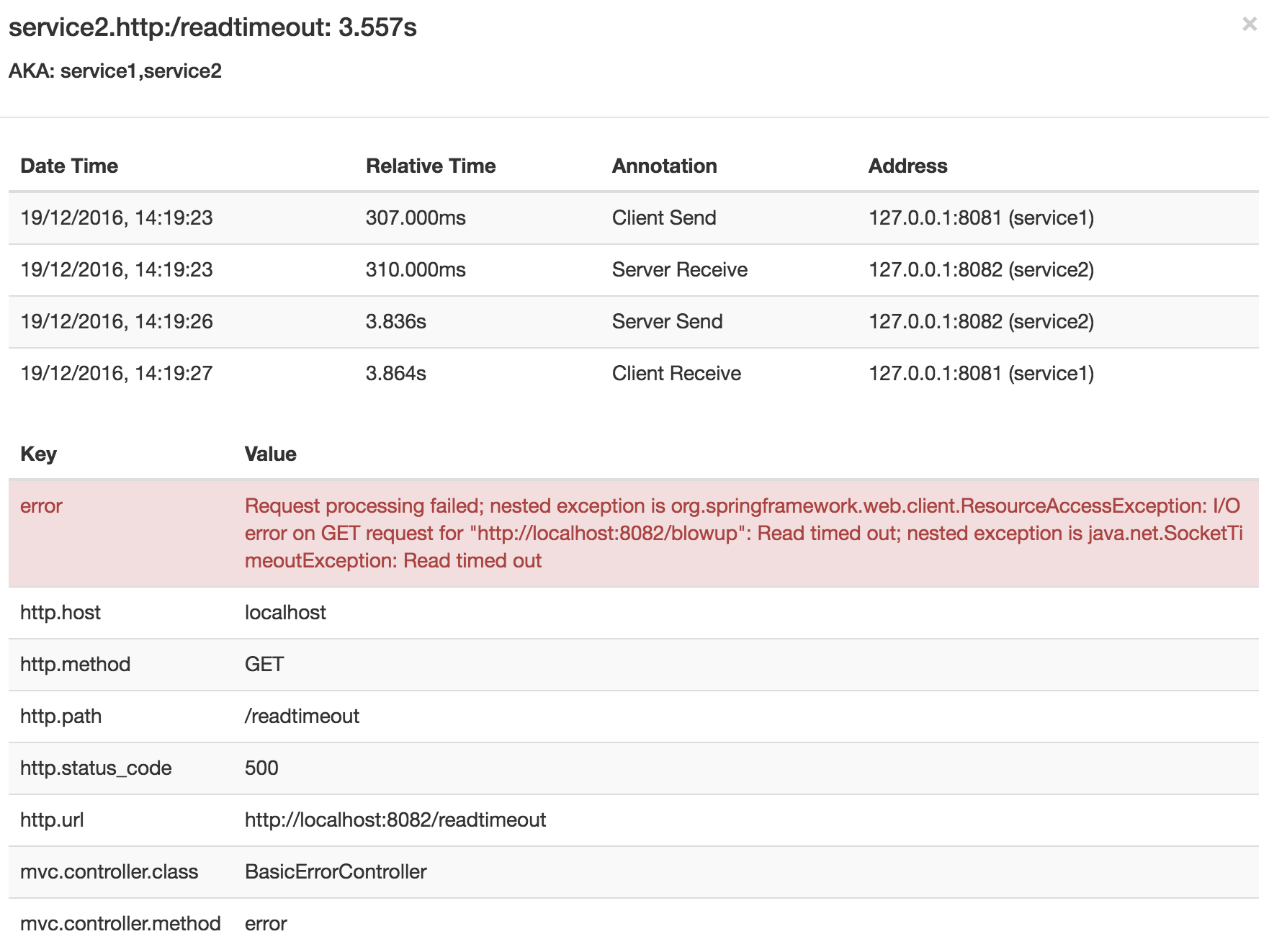
### **50.2.2可视化错误**

Zipkin使您可以可视化跟踪中的错误。当引发异常但未捕获到异常时，我们在跨度上设置了适当的标记，然后Zipkin即可正确着色。您可以在迹线列表中看到一条红色的迹线。之所以出现，是因为引发了异常。

如果单击该跟踪，将看到类似的图片，如下所示：



如果然后单击spans之一，则会看到以下内容



跨度显示了错误的原因以及与之相关的整个堆栈跟踪。

### **50.2.3勇敢的分布式跟踪**

从版本2.0.0开始，Spring Cloud Sleuth使用[Brave](https://github.com/openzipkin/brave" \t "https://www.springcloud.cc/_top)作为跟踪库。因此，Sleuth不再负责存储上下文，而是将工作委托给Brave。

由于Sleuth与Brave具有不同的命名和标记约定，因此我们决定从现在开始遵循Brave的约定。但是，如果要使用传统的Sleuth方法，可以将spring.sleuth.http.legacy.enabled属性设置为true。

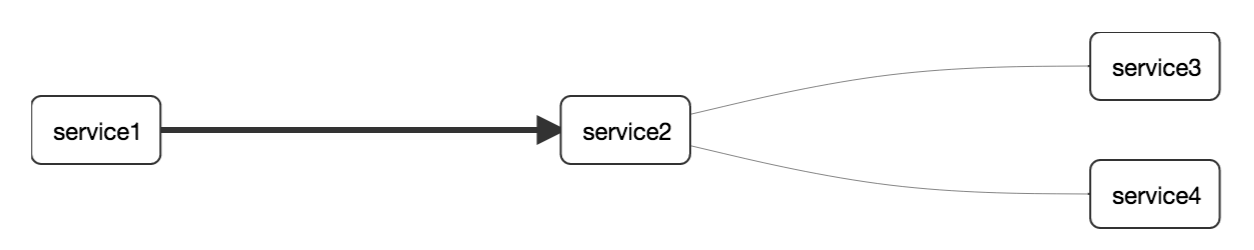
### **50.2.4现场示例**

**图50.1。点击“ Pivotal Web服务”图标即可观看！**



[点击此处观看直播！](https://docssleuth-zipkin-server.cfapps.io/" \t "https://www.springcloud.cc/_top)

Zipkin中的依赖关系图应类似于下图：



**图50.2。点击“ Pivotal Web服务”图标即可观看！**



[点击此处观看直播！](https://docssleuth-zipkin-server.cfapps.io/dependency" \t "https://www.springcloud.cc/_top)

### **50.2.5日志关联**

当使用grep通过扫描等于（例如）2485ec27856c56f4的跟踪ID来读取这四个应用程序的日志时，将获得类似于以下内容的输出：

service1.log:2016-02-26 11:15:47.561 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Hello from service1. Calling service2

service2.log:2016-02-26 11:15:47.710 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Hello from service2. Calling service3 and then service4

service3.log:2016-02-26 11:15:47.895 INFO [service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1] i.s.c.sleuth.docs.service3.Application : Hello from service3

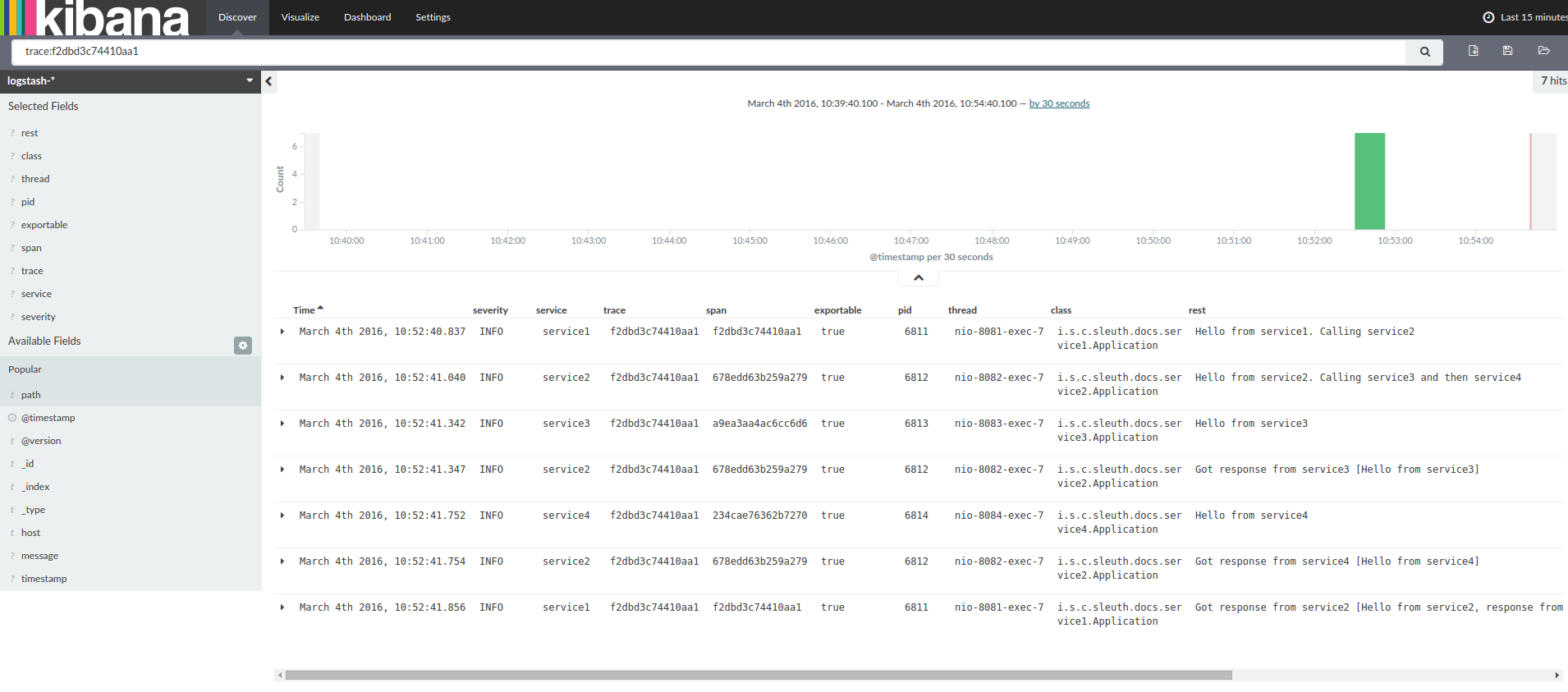
service2.log:2016-02-26 11:15:47.924 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service3 [Hello from service3]

service4.log:2016-02-26 11:15:48.134 INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1] i.s.c.sleuth.docs.service4.Application : Hello from service4

service2.log:2016-02-26 11:15:48.156 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service4 [Hello from service4]

service1.log:2016-02-26 11:15:48.182 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Got response from service2 [Hello from service2, response from service3 [Hello from service3] and from service4 [Hello from service4]]

如果您使用日志汇总工具（例如[Kibana](https://www.elastic.co/products/kibana" \t "https://www.springcloud.cc/_top)，[Splunk](https://www.splunk.com/" \t "https://www.springcloud.cc/_top)和其他工具），则可以对发生的事件进行排序。来自Kibana的示例类似于下图：



如果要使用[Logstash](https://www.elastic.co/guide/en/logstash/current/index.html" \t "https://www.springcloud.cc/_top)，以下清单显示了Logstash的Grok模式：

filter {

# pattern matching logback pattern

grok {

match => { "message" => "%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

|  |
| --- |
| IMG_438 |
| 如果要将Grok与Cloud Foundry中的日志一起使用，则必须使用以下模式： |

filter {

# pattern matching logback pattern

grok {

match => { "message" => "(?m)OUT\s+%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

#### **使用Logstash进行JSON Logback**

通常，您不想将日志存储在文本文件中，而是存储在Logstash可以立即选择的JSON文件中。为此，您必须执行以下操作（出于可读性考虑，我们以groupId:artifactId:version表示法传递依赖项）。

**依赖关系设置**

1. 确保Logback位于类路径（ch.qos.logback:logback-core）上。
2. 添加Logstash Logback编码。例如，要使用版本4.6，请添加net.logstash.logback:logstash-logback-encoder:4.6。

**登录设置**

考虑以下Logback配置文件示例（名为[logback-spring.xml](https://github.com/spring-cloud-samples/sleuth-documentation-apps/blob/master/service1/src/main/resources/logback-spring.xml" \t "https://www.springcloud.cc/_top)）。

<?xml version="1.0" encoding="UTF-8"?><configuration>

<include resource="org/springframework/boot/logging/logback/defaults.xml"/>

​

<springProperty scope="context" name="springAppName" source="spring.application.name"/>

*<!-- Example for logging into the build folder of your project -->*

<property name="LOG\_FILE" value="${BUILD\_FOLDER:-build}/${springAppName}"/>​

*<!-- You can override this to have a custom pattern -->*

<property name="CONSOLE\_LOG\_PATTERN"

value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"/>

*<!-- Appender to log to console -->*

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

*<!-- Minimum logging level to be presented in the console logs-->*

<level>DEBUG</level>

</filter>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

*<!-- Appender to log to file -->*​

<appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

​

*<!-- Appender to log to file in a JSON format -->*

<appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}.json</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">

<providers>

<timestamp>

<timeZone>UTC</timeZone>

</timestamp>

<pattern>

<pattern>

{

"severity": "%level",

"service": "${springAppName:-}",

"trace": "%X{X-B3-TraceId:-}",

"span": "%X{X-B3-SpanId:-}",

"parent": "%X{X-B3-ParentSpanId:-}",

"exportable": "%X{X-Span-Export:-}",

"pid": "${PID:-}",

"thread": "%thread",

"class": "%logger{40}",

"rest": "%message"

}

</pattern>

</pattern>

</providers>

</encoder>

</appender>

​

<root level="INFO">

<appender-ref ref="console"/>

*<!-- uncomment this to have also JSON logs -->*

*<!--<appender-ref ref="logstash"/>-->*

*<!--<appender-ref ref="flatfile"/>-->*

</root></configuration>

该Logback配置文件：

* 将应用程序中的信息以JSON格式记录到build/${spring.application.name}.json文件中。
* 注释了两个附加的附加程序：控制台和标准日志文件。
* 具有与上一部分相同的日志记录模式。

|  |
| --- |
| IMG_439 |
| 如果使用自定义logback-spring.xml，则必须在bootstrap中传递spring.application.name，而不是在application属性文件中传递。否则，您的自定义登录文件将无法正确读取该属性。 |

### **50.2.6传播Span上下文**

跨度上下文是必须跨进程边界传播到任何子项spans的状态。Span上下文的一部分是行李。跟踪和跨度ID是跨度上下文的必需部分。行李是可选部件。

行李是存储在span上下文中的一组key：value对。行李与踪迹一起旅行，并附着在每个跨度上。Spring Cloud Sleuth理解，如果HTTP标头以baggage-为前缀，则标头与行李有关，对于消息传递，标头以baggage\_开头。

|  |  |
| --- | --- |
| IMG_440 | **重要** |
| 当前对行李物品的数量或大小没有限制。但是，请记住，太多会降低系统吞吐量或增加RPC延迟。在极端情况下，由于超出传输级消息或标头容量，过多的行李可能会使应用程序崩溃。 |

以下示例显示跨度设置行李：

Span initialSpan = **this**.tracer.nextSpan().name("span").start();

ExtraFieldPropagation.set(initialSpan.context(), "foo", "bar");

ExtraFieldPropagation.set(initialSpan.context(), "UPPER\_CASE", "someValue");

#### **行李与Span标签**

行李随身携带（每个孩子跨度都包含其父母的行李）。Zipkin不了解行李并且不接收该信息。

|  |  |
| --- | --- |
| IMG_441 | **重要** |
| 从Sleuth 2.0.0开始，您必须在项目配置中显式传递行李密钥名称。[在此处](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "prefixed-fields" \o "54.1.1 Prefixed fields)阅读有关该设置的更多信息 |

标签被附加到特定范围。换句话说，它们仅针对该特定跨度显示。但是，您可以按标签搜索以找到轨迹，前提是存在一个具有所搜索标签值的跨度。

如果您希望能够基于行李查找跨度，则应在根跨度中添加相应的条目作为标签。

|  |  |
| --- | --- |
| IMG_442 | **重要** |
| 范围必须在范围内。 |

以下清单显示了使用行李的集成测试：

**设置。**

spring.sleuth:

baggage-keys:

- baz

- bizarrecase

propagation-keys:

- foo

- upper\_case

**编码。**

initialSpan.tag("foo",

ExtraFieldPropagation.get(initialSpan.context(), "foo"));

initialSpan.tag("UPPER\_CASE",

ExtraFieldPropagation.get(initialSpan.context(), "UPPER\_CASE"));

## **50.3在项目中添加侦探**

本节介绍如何使用Maven或Gradle将Sleuth添加到项目中。

|  |  |
| --- | --- |
| IMG_443 | **重要** |
| 为确保您的应用程序名称正确显示在Zipkin中，请在bootstrap.yml中设置spring.application.name属性。 |

### **50.3.1仅侦探（对数关联）**

如果您只想使用Spring Cloud Sleuth而没有Zipkin集成，则将spring-cloud-starter-sleuth模块添加到您的项目中。

下面的示例演示如何使用Maven添加Sleuth：

**Maven.**

<dependencyManagement> IMG_444

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

<dependency> IMG_445

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

<artifactId>spring-cloud-starter-sleuth</artifactId></dependency>

|  |  |
| --- | --- |
| [IMG_446](https://www.springcloud.cc/spring-cloud-greenwich.html#CO1-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_447](https://www.springcloud.cc/spring-cloud-greenwich.html#CO1-2) | 将依赖项添加到spring-cloud-starter-sleuth。 |

下面的示例演示如何使用Gradle添加Sleuth：

**Gradle.**

dependencyManagement { IMG_448

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies { IMG_449

compile "org.springframework.cloud:spring-cloud-starter-sleuth"

}

|  |  |
| --- | --- |
| [IMG_450](https://www.springcloud.cc/spring-cloud-greenwich.html#CO2-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_451](https://www.springcloud.cc/spring-cloud-greenwich.html#CO2-2) | 将依赖项添加到spring-cloud-starter-sleuth。 |

### **50.3.2通过HTTP与Zipkin一起侦听**

如果您同时需要Sleuth和Zipkin，请添加spring-cloud-starter-zipkin依赖项。

以下示例显示如何针对Maven执行此操作：

**Maven.**

<dependencyManagement> IMG_452

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

<dependency> IMG_453

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

<artifactId>spring-cloud-starter-zipkin</artifactId></dependency>

|  |  |
| --- | --- |
| [IMG_454](https://www.springcloud.cc/spring-cloud-greenwich.html#CO3-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_455](https://www.springcloud.cc/spring-cloud-greenwich.html#CO3-2) | 将依赖项添加到spring-cloud-starter-zipkin。 |

以下示例显示了如何对Gradle执行此操作：

**Gradle.**

dependencyManagement { IMG_456

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies { IMG_457

compile "org.springframework.cloud:spring-cloud-starter-zipkin"

}

|  |  |
| --- | --- |
| [IMG_458](https://www.springcloud.cc/spring-cloud-greenwich.html#CO4-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_459](https://www.springcloud.cc/spring-cloud-greenwich.html#CO4-2) | 将依赖项添加到spring-cloud-starter-zipkin。 |

### **50.3.3侦探，其中Zipkin高于RabbitMQ或Kafka**

如果要使用RabbitMQ或Kafka而不是HTTP，请添加spring-rabbit或spring-kafka依赖项。默认目的地名称为zipkin。

如果使用Kafka，则必须相应地设置属性spring.zipkin.sender.type：

spring.zipkin.sender.type: kafka

|  |  |
| --- | --- |
| IMG_460 | **警告** |
| spring-cloud-sleuth-stream已过时，并且与这些目的地不兼容。 |

如果要让Sleuth超过RabbitMQ，请添加spring-cloud-starter-zipkin和spring-rabbit依赖项。

以下示例显示了如何对Gradle执行此操作：

**Maven.**

<dependencyManagement> IMG_461

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

<dependency> IMG_462

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

<artifactId>spring-cloud-starter-zipkin</artifactId></dependency><dependency> IMG_463

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId></dependency>

|  |  |
| --- | --- |
| [IMG_464](https://www.springcloud.cc/spring-cloud-greenwich.html#CO5-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_465](https://www.springcloud.cc/spring-cloud-greenwich.html#CO5-2) | 将依赖项添加到spring-cloud-starter-zipkin。这样，所有嵌套的依赖项都将被下载。 |
| [IMG_466](https://www.springcloud.cc/spring-cloud-greenwich.html#CO5-3) | 要自动配置RabbitMQ，请添加spring-rabbit依赖项。 |

**Gradle.**

dependencyManagement { IMG_467

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies {

compile "org.springframework.cloud:spring-cloud-starter-zipkin" IMG_468

compile "org.springframework.amqp:spring-rabbit" IMG_469

}

|  |  |
| --- | --- |
| [IMG_470](https://www.springcloud.cc/spring-cloud-greenwich.html#CO6-1) | 我们建议您通过Spring BOM添加依赖项管理，这样就不必自己管理版本。 |
| [IMG_471](https://www.springcloud.cc/spring-cloud-greenwich.html#CO6-2) | 将依赖项添加到spring-cloud-starter-zipkin。这样，所有嵌套的依赖项都将被下载。 |
| [IMG_472](https://www.springcloud.cc/spring-cloud-greenwich.html#CO6-3) | 要自动配置RabbitMQ，请添加spring-rabbit依赖项。 |

## **50.4覆盖Zipkin的自动配置**

Spring Cloud Sleuth支持从2.1.0版开始将跟踪发送到多个跟踪系统。为了使它起作用，每个跟踪系统都需要具有Reporter<Span>和Sender。如果要覆盖提供的beans，则需要给它们指定一个特定的名称。为此，您可以分别使用ZipkinAutoConfiguration.REPORTER\_BEAN\_NAME和ZipkinAutoConfiguration.SENDER\_BEAN\_NAME。

@Configuration**protected** **static** **class** MyConfig {

@Bean(ZipkinAutoConfiguration.REPORTER\_BEAN\_NAME)

Reporter<zipkin2.Span> myReporter() {

**return** AsyncReporter.create(mySender());

}

@Bean(ZipkinAutoConfiguration.SENDER\_BEAN\_NAME)

MySender mySender() {

**return** **new** MySender();

}

**static** **class** MySender **extends** Sender {

**private** **boolean** spanSent = false;

**boolean** isSpanSent() {

**return** **this**.spanSent;

}

@Override

**public** Encoding encoding() {

**return** Encoding.JSON;

}

@Override

**public** **int** messageMaxBytes() {

**return** Integer.MAX\_VALUE;

}

@Override

**public** **int** messageSizeInBytes(List<**byte**[]> encodedSpans) {

**return** encoding().listSizeInBytes(encodedSpans);

}

@Override

**public** Call<Void> sendSpans(List<**byte**[]> encodedSpans) {

**this**.spanSent = true;

**return** Call.create(null);

}

}

}

## **51.其他资源**

[点击此处，](https://content.pivotal.io/springone-platform-2017/distributed-tracing-latency-analysis-for-your-microservices-grzejszczak-krishna" \t "https://www.springcloud.cc/_top)您可以观看[Reshmi Krishna](https://twitter.com/reshmi9k" \t "https://www.springcloud.cc/_top)和[Marcin Grzejszczak](https://twitter.com/mgrzejszczak" \t "https://www.springcloud.cc/_top)谈论Spring Cloud Sleuth和Zipkin [的视频](https://content.pivotal.io/springone-platform-2017/distributed-tracing-latency-analysis-for-your-microservices-grzejszczak-krishna" \t "https://www.springcloud.cc/_top)。

您可以[在openzipkin / sleuth-webmvc-example系统信息库中](https://github.com/openzipkin/sleuth-webmvc-example" \t "https://www.springcloud.cc/_top)检查Sleuth和Brave的不同设置。

## **52.特点**

将跟踪和跨度ID添加到Slf4J MDC，因此您可以在日志聚合器中从给定的跟踪或跨度提取所有日志，如以下示例日志所示：

2016-02-02 15:30:57.902 INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...

2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...

2016-02-02 15:31:01.936 INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...

请注意来自MDC的[appname,traceId,spanId,exportable]条目：

* + spanId：发生的特定操作的ID。
  + appname：记录跨度的应用程序的名称。
  + traceId：包含跨度的延迟图的ID。
  + exportable：是否应将日志导出到Zipkin。您何时希望跨度不可导出？当您要将某些操作包装在Span中并且仅将其写入日志时。
* 提供对常见的分布式跟踪数据模型的抽象：跟踪，spans（形成DAG），注释和键值注释。Spring Cloud Sleuth大致基于HTrace，但与Zipkin（Dapper）兼容。
* Sleuth记录计时信息以帮助进行延迟分析。通过使用侦探，您可以查明应用程序中延迟的原因。

编写Sleuth时不要过多记录日志，也不会导致生产应用程序崩溃。为此，Sleuth：

* + 在带内传播有关调用图的结构数据，并在带外传播其余数据。
  + 包括对诸如HTTP之类的层的自觉检测。
  + 包括用于管理数量的采样策略。
  + 可以向Zipkin系统报告以进行查询和可视化。
* 从Spring应用程序（servlet过滤器，异步端点，休息模板，计划的操作，消息通道，Zuul过滤器和Feign客户端）中检测常见的入口和出口点。
* Sleuth包含默认逻辑以跨HTTP或消息传递边界加入跟踪。例如，HTTP传播适用于Zipkin兼容的请求标头。
* 侦查可以在进程之间传播上下文（也称为行李）。因此，如果您在Span上设置了行李元素，则该元素将通过HTTP或消息传递被下游发送到其他进程。
* 提供一种创建或继续spans以及通过注释添加标签和日志的方法。

如果spring-cloud-sleuth-zipkin在类路径上，则该应用生成并收集Zipkin兼容的跟踪。默认情况下，它通过HTTP将它们发送到本地主机（端口9411）上的Zipkin服务器。您可以通过设置spring.zipkin.baseUrl来配置服务的位置。

* + 如果您依赖spring-rabbit，则您的应用会将跟踪发送到RabbitMQ代理，而不是HTTP。
  + 如果您依赖spring-kafka并设置为spring.zipkin.sender.type: kafka，则您的应用会将跟踪发送到Kafka代理而不是HTTP。

|  |  |
| --- | --- |
| IMG_473 | **警告** |
| spring-cloud-sleuth-stream已过时，不应再使用。 |

* Spring Cloud Sleuth与[OpenTracing](https://opentracing.io/" \t "https://www.springcloud.cc/_top)兼容。

|  |  |
| --- | --- |
| IMG_474 | **重要** |
| 如果使用Zipkin，请通过设置spring.sleuth.sampler.probability（默认值：0.1，即10％）来配置spans导出的概率。否则，您可能会认为Sleuth无法正常工作，因为它忽略了一些spans。 |

|  |
| --- |
| IMG_475 |
| 始终设置SLF4J MDC，并且按先前显示的示例，登录用户可以立即在日志中看到跟踪和跨度ID。其他日志记录系统必须配置自己的格式化程序才能获得相同的结果。默认值如下：logging.pattern.level设置为%5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]（这是Logback用户的Spring Boot功能）。如果您不使用SLF4J，则不会自动应用此模式。 |

## **52.1.Brave简介**

|  |  |
| --- | --- |
| IMG_476 | **重要** |
| 从版本2.0.0开始，Spring Cloud Sleuth使用 [Brave](https://github.com/openzipkin/brave" \t "https://www.springcloud.cc/_top)作为跟踪库。为方便起见，我们在此处嵌入了Brave文档的一部分。 |

|  |  |
| --- | --- |
| IMG_477 | **重要** |
| 在大多数情况下，您只需要使用Sleuth提供的Brave提供的Tracer或SpanCustomizer beans。以下文档概述了Brave是什么以及它如何工作。 |

勇敢是一个库，用于捕获有关分布式操作的延迟信息并将其报告给Zipkin。大多数用户不直接使用Brave。他们使用库或框架，而不是代表他们使用Brave。

该模块包括一个跟踪器，该跟踪器创建并加入spans，以对潜在的分布式工作的延迟进行建模。它还包括用于在网络边界上传播跟踪上下文的库（例如，使用HTTP标头）。

### **52.1.1跟踪**

最重要的是，您需要一个brave.Tracer，配置为[向Zipkin报告](https://github.com/openzipkin/zipkin-reporter-java" \t "https://www.springcloud.cc/_top)。

以下示例设置通过HTTP（而非Kafka）将跟踪数据（spans）发送到Zipkin：

**class** MyClass {

**private** **final** Tracer tracer;

*// Tracer will be autowired*

MyClass(Tracer tracer) {

**this**.tracer = tracer;

}

**void** doSth() {

Span span = tracer.newTrace().name("encode").start();

*// ...*

}

}

|  |  |
| --- | --- |
| IMG_478 | **重要** |
| 如果您的跨度包含的名称长于50个字符，则该名称将被截断为50个字符。您的名字必须明确明确。知名人士会导致延迟问题，有时甚至会引发异常。 |

跟踪器创建并加入spans，以对潜在分布式工作的延迟进行建模。它可以采用采样来减少处理过程中的开销，减少发送到Zipkin的数据量，或同时减少两者。

跟踪程序返回的跨度在完成时将数据报告到Zipkin，如果未采样则不执行任何操作。开始跨度后，您可以注释感兴趣的事件或添加包含详细信息或查找键的标签。

跨度具有包含跟踪标识符的上下文，该标识符将跨度放置在代表分布式操作的树中的正确位置。

### **52.1.2本地跟踪**

跟踪永远不会离开进程的代码时，请在范围范围内运行它。

@Autowired Tracer tracer;

*// Start a new trace or a span within an existing trace representing an operation*

ScopedSpan span = tracer.startScopedSpan("encode");**try** {

*// The span is in "scope" meaning downstream code such as loggers can see trace IDs*

**return** encoder.encode();

} **catch** (RuntimeException | Error e) {

span.error(e); *// Unless you handle exceptions, you might not know the operation failed!*

**throw** e;

} **finally** {

span.finish(); *// always finish the span*

}

当您需要更多功能或更好的控制时，请使用Span类型：

@Autowired Tracer tracer;

*// Start a new trace or a span within an existing trace representing an operation*

Span span = tracer.nextSpan().name("encode").start();*// Put the span in "scope" so that downstream code such as loggers can see trace IDs***try** (SpanInScope ws = tracer.withSpanInScope(span)) {

**return** encoder.encode();

} **catch** (RuntimeException | Error e) {

span.error(e); *// Unless you handle exceptions, you might not know the operation failed!*

**throw** e;

} **finally** {

span.finish(); *// note the scope is independent of the span. Always finish a span.*

}

上面的两个示例都报告了完全相同的跨度！

在上面的示例中，范围将是新的根范围或现有跟踪中的下一个子级。

### **52.1.3自定义跨度**

一旦具有跨度，就可以向其添加标签。标签可以用作查找关键字或详细信息。例如，您可以在运行时版本中添加标签，如以下示例所示：

span.tag("clnt/finagle.version", "6.36.0");

向第三方公开自定义spans的功能时，最好使用brave.SpanCustomizer而不是brave.Span。前者更易于理解和测试，不会用跨度生命周期挂钩吸引用户。

**interface** MyTraceCallback {

**void** request(Request request, SpanCustomizer customizer);

}

由于brave.Span实现了brave.SpanCustomizer，因此可以将其传递给用户，如以下示例所示：

**for** (MyTraceCallback callback : userCallbacks) {

callback.request(request, span);

}

### **52.1.4隐式查找电流Span**

有时，您不知道跟踪是否正在进行，并且您不希望用户执行空检查。brave.CurrentSpanCustomizer通过将数据添加到正在进行或删除的任何跨度中来解决此问题，如以下示例所示：

例如

*// The user code can then inject this without a chance of it being null.*@Autowired SpanCustomizer span;

**void** userCode() {

span.annotate("tx.started");

...

}

### **52.1.5 RPC跟踪**

|  |
| --- |
| IMG_479 |
| 在滚动自己的RPC工具之前，请检查[此处编写](https://github.com/openzipkin/brave/tree/master/instrumentation" \t "https://www.springcloud.cc/_top)的[工具](https://github.com/openzipkin/brave/tree/master/instrumentation" \t "https://www.springcloud.cc/_top)和[Zipkin的列表](https://zipkin.io/pages/existing_instrumentations.html" \t "https://www.springcloud.cc/_top)。 |

RPC跟踪通常由拦截器自动完成。它们在幕后添加了与其在RPC操作中的角色相关的标签和事件。

以下示例显示如何添加客户端范围：

@Autowired Tracing tracing;@Autowired Tracer tracer;

*// before you send a request, add metadata that describes the operation*

span = tracer.nextSpan().name(service + "/" + method).kind(CLIENT);

span.tag("myrpc.version", "1.0.0");

span.remoteServiceName("backend");

span.remoteIpAndPort("172.3.4.1", 8108);

*// Add the trace context to the request, so it can be propagated in-band*

tracing.propagation().injector(Request::addHeader)

.inject(span.context(), request);

*// when the request is scheduled, start the span*

span.start();

*// if there is an error, tag the span*

span.tag("error", error.getCode());*// or if there is an exception*

span.error(exception);

*// when the response is complete, finish the span*

span.finish();

#### **单向跟踪**

有时，您需要对有请求但无响应的异步操作进行建模。在常规的RPC跟踪中，您使用span.finish()表示已收到响应。在单向跟踪中，由于不希望响应，因此改用span.flush()。

下面的示例显示客户端如何建模单向操作：

@Autowired Tracing tracing;@Autowired Tracer tracer;

*// start a new span representing a client request*

oneWaySend = tracer.nextSpan().name(service + "/" + method).kind(CLIENT);

*// Add the trace context to the request, so it can be propagated in-band*

tracing.propagation().injector(Request::addHeader)

.inject(oneWaySend.context(), request);

*// fire off the request asynchronously, totally dropping any response*

request.execute();

*// start the client side and flush instead of finish*

oneWaySend.start().flush();

下面的示例显示服务器如何处理单向操作：

@Autowired Tracing tracing;@Autowired Tracer tracer;

*// pull the context out of the incoming request*

extractor = tracing.propagation().extractor(Request::getHeader);

*// convert that context to a span which you can name and add tags to*

oneWayReceive = nextSpan(tracer, extractor.extract(request))

.name("process-request")

.kind(SERVER)

... add tags etc.

*// start the server side and flush instead of finish*

oneWayReceive.start().flush();

*// you should not modify this span anymore as it is complete. However,// you can create children to represent follow-up work.*

next = tracer.newSpan(oneWayReceive.context()).name("step2").start();

## **53.抽样**

可以采用采样来减少收集和报告的过程外数据。如果未对跨度进行采样，则不会增加开销（无操作）。

采样是一项前期决策，这意味着报告数据的决策是在跟踪的第一个操作中做出的，并且该决策会向下游传播。

默认情况下，全局采样器将单个速率应用于所有跟踪的操作。Tracer.Builder.sampler控制此设置，默认为跟踪每个请求。

## **53.1声明式采样**

一些应用程序需要根据java方法的类型或注释进行采样。

大多数用户使用框架拦截器来自动化这种策略。以下示例显示了它可能在内部如何工作：

@Autowired Tracer tracer;

*// derives a sample rate from an annotation on a java method*

DeclarativeSampler<Traced> sampler = DeclarativeSampler.create(Traced::sampleRate);

@Around("@annotation(traced)")**public** Object traceThing(ProceedingJoinPoint pjp, Traced traced) **throws** Throwable {

*// When there is no trace in progress, this decides using an annotation*

Sampler decideUsingAnnotation = declarativeSampler.toSampler(traced);

Tracer tracer = tracer.withSampler(decideUsingAnnotation);

*// This code looks the same as if there was no declarative override*

ScopedSpan span = tracer.startScopedSpan(spanName(pjp));

**try** {

**return** pjp.proceed();

} **catch** (RuntimeException | Error e) {

span.error(e);

**throw** e;

} **finally** {

span.finish();

}

}

## **53.2自定义采样**

根据操作的不同，您可能需要应用不同的策略。例如，您可能不想跟踪对静态资源（例如图像）的请求，或者您想跟踪所有对新API的请求。

大多数用户使用框架拦截器来自动化这种策略。以下示例显示了它可能在内部如何工作：

@Autowired Tracer tracer;@Autowired Sampler fallback;

Span nextSpan(**final** Request input) {

Sampler requestBased = Sampler() {

@Override **public** **boolean** isSampled(**long** traceId) {

**if** (input.url().startsWith("/experimental")) {

**return** true;

} **else** **if** (input.url().startsWith("/static")) {

**return** false;

}

**return** fallback.isSampled(traceId);

}

};

**return** tracer.withSampler(requestBased).nextSpan();

}

## **53.3 Spring Cloud Sleuth中的采样**

默认情况下，Spring Cloud Sleuth将所有spans设置为不可导出。这意味着跟踪将显示在日志中，而不显示在任何远程存储中。测试默认值通常就足够了，如果仅使用日志（例如，使用ELK聚合器），则可能只需要它即可。如果将跨度数据导出到Zipkin，则还有一个Sampler.ALWAYS\_SAMPLE设置可以导出所有内容，还有一个ProbabilityBasedSampler设置可以对spans的固定分数进行采样。

|  |
| --- |
| IMG_480 |
| 如果使用spring-cloud-sleuth-zipkin，则默认为ProbabilityBasedSampler。您可以通过设置spring.sleuth.sampler.probability来配置导出。传递的值必须是0.0到1.0的两倍。 |

可以通过创建bean定义来安装采样器，如以下示例所示：

@Bean**public** Sampler defaultSampler() {

**return** Sampler.ALWAYS\_SAMPLE;

}

|  |
| --- |
| IMG_481 |
| 您可以将HTTP标头X-B3-Flags设置为1，或者在进行消息传递时，可以将spanFlags标头设置为1。这样做将强制电流跨度可导出，而不管采样决定如何。 |

为了使用速率受限的采样器，请设置spring.sleuth.sampler.rate属性，以选择每秒钟间隔要接受的跟踪量。最小数量为0，最大数量为2,147,483,647（最大整数）。

## **54.传播**

需要进行传播以确保源自同一根的活动被收集到同一条迹线中。最常见的传播方法是通过将RPC请求发送到接收它的服务器来从客户端复制跟踪上下文。

例如，进行下游HTTP调用时，其跟踪上下文被编码为请求标头，并与之一起发送，如下图所示：

Client Span Server Span

┌──────────────────┐ ┌──────────────────┐

│ │ │ │

│ TraceContext │ Http Request Headers │ TraceContext │

│ ┌──────────────┐ │ ┌───────────────────┐ │ ┌──────────────┐ │

│ │ TraceId │ │ │ X─B3─TraceId │ │ │ TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ │ ParentSpanId │ │ Extract │ X─B3─ParentSpanId │ Inject │ │ ParentSpanId │ │

│ │ ├─┼─────────>│ ├────────┼>│ │ │

│ │ SpanId │ │ │ X─B3─SpanId │ │ │ SpanId │ │

│ │ │ │ │ │ │ │ │ │

│ │ Sampled │ │ │ X─B3─Sampled │ │ │ Sampled │ │

│ └──────────────┘ │ └───────────────────┘ │ └──────────────┘ │

│ │ │ │

└──────────────────┘ └──────────────────┘

上面的名称来自[B3 Propagation](https://github.com/openzipkin/b3-propagation" \t "https://www.springcloud.cc/_top)，它内置于Brave，并具有许多语言和框架的实现。

大多数用户使用框架拦截器来自动化传播。接下来的两个示例显示了这对于客户端和服务器的工作方式。

以下示例显示了客户端传播如何工作：

@Autowired Tracing tracing;

*// configure a function that injects a trace context into a request*

injector = tracing.propagation().injector(Request.Builder::addHeader);

*// before a request is sent, add the current span's context to it*

injector.inject(span.context(), request);

以下示例显示了服务器端传播的工作方式：

@Autowired Tracing tracing;@Autowired Tracer tracer;

*// configure a function that extracts the trace context from a request*

extractor = tracing.propagation().extractor(Request::getHeader);

*// when a server receives a request, it joins or starts a new trace*

span = tracer.nextSpan(extractor.extract(request));

## **54.1传播额外的字段**

有时您需要传播额外的字段，例如请求ID或备用跟踪上下文。例如，如果您处于Cloud Foundry环境中，则可能要传递请求ID，如以下示例所示：

*// when you initialize the builder, define the extra field you want to propagate*

Tracing.newBuilder().propagationFactory(

ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-vcap-request-id")

);

*// later, you can tag that request ID or use it in log correlation*

requestId = ExtraFieldPropagation.get("x-vcap-request-id");

您可能还需要传播未使用的跟踪上下文。例如，您可能处于Amazon Web服务环境中，但没有向X-Ray报告数据。为了确保X射线可以正确共存，请传递其跟踪标头，如以下示例所示：

tracingBuilder.propagationFactory(

ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-amzn-trace-id")

);

|  |
| --- |
| IMG_482 |
| 在Spring Cloud Sleuth中，跟踪构建器Tracing.newBuilder()的所有元素都定义为beans。因此，如果您要传递自定义PropagationFactory，则足以创建该类型的bean，我们将在Tracing bean中进行设置。 |

### **54.1.1前缀字段**

如果它们遵循通用模式，则还可以在字段前面加上前缀。以下示例显示了如何按原样传播x-vcap-request-id字段，但如何分别以x-baggage-country-code和x-baggage-user-id的形式发送country-code和user-id字段：

Tracing.newBuilder().propagationFactory(

ExtraFieldPropagation.newFactoryBuilder(B3Propagation.FACTORY)

.addField("x-vcap-request-id")

.addPrefixedFields("x-baggage-", Arrays.asList("country-code", "user-id"))

.build()

);

以后，您可以调用以下代码来影响当前跟踪上下文的国家/地区代码：

ExtraFieldPropagation.set("x-country-code", "FO");

String countryCode = ExtraFieldPropagation.get("x-country-code");

或者，如果您有对跟踪上下文的引用，则可以显式使用它，如以下示例所示：

ExtraFieldPropagation.set(span.context(), "x-country-code", "FO");

String countryCode = ExtraFieldPropagation.get(span.context(), "x-country-code");

|  |  |
| --- | --- |
| IMG_483 | **重要** |
| 与以前版本的Sleuth的不同之处在于，使用Brave，您必须传递行李钥匙列表。有两个属性可以实现此目的。使用spring.sleuth.baggage-keys，您可以为HTTP呼叫设置前缀为baggage-的密钥，而为消息传递则设置前缀baggage\_的密钥。您还可以使用spring.sleuth.propagation-keys属性来传递已列入白名单且没有任何前缀的前缀键列表。注意，标题键前面没有x-。 |

为了自动将行李值设置为Slf4j的MDC，您必须使用白名单中的行李和传播键列表来设置spring.sleuth.log.slf4j.whitelisted-mdc-keys属性。例如，spring.sleuth.log.slf4j.whitelisted-mdc-keys=foo会将foo行李的价值设置为MDC。

|  |  |
| --- | --- |
| IMG_484 | **重要** |
| 请记住，将条目添加到MDC可能会大大降低应用程序的性能！ |

如果要将行李条目添加为标签，以使可以通过行李条目搜索spans，则可以将白名单中的行李钥匙列表设置为spring.sleuth.propagation.tag.whitelisted-keys。要禁用此功能，您必须传递spring.sleuth.propagation.tag.enabled=false属性。

### **54.1.2提取传播的上下文**

TraceContext.Extractor<C>从传入的请求或消息中读取跟踪标识符和采样状态。载体通常是一个请求对象或标头。

此实用程序用于标准工具（例如HttpServerHandler），但也可以用于自定义RPC或消息传递代码。

TraceContextOrSamplingFlags通常仅与Tracer.nextSpan(extracted)一起使用，除非您要在客户端和服务器之间共享范围ID。

### **54.1.3在客户端和服务器之间共享范围ID**

正常的检测模式是创建一个跨度，该跨度代表RPC的服务器端。Extractor.extract在应用于传入的客户端请求时可能返回完整的跟踪上下文。Tracer.joinSpan尝试使用相同的跨度ID（如果支持）或不创建子跨度来继续此跟踪。当跨度ID被共享时，报告的数据包括这样的标志。

下图显示了B3传播的示例：

┌───────────────────┐ ┌───────────────────┐

Incoming Headers │ TraceContext │ │ TraceContext │

┌───────────────────┐(extract)│ ┌───────────────┐ │(join)│ ┌───────────────┐ │

│ X─B3-TraceId │─────────┼─┼> TraceId │ │──────┼─┼> TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ X─B3-ParentSpanId │─────────┼─┼> ParentSpanId │ │──────┼─┼> ParentSpanId │ │

│ │ │ │ │ │ │ │ │ │

│ X─B3-SpanId │─────────┼─┼> SpanId │ │──────┼─┼> SpanId │ │

└───────────────────┘ │ │ │ │ │ │ │ │

│ │ │ │ │ │ Shared: true │ │

│ └───────────────┘ │ │ └───────────────┘ │

└───────────────────┘ └───────────────────┘

某些传播系统仅转发在Propagation.Factory.supportsJoin() == false时检测到的父范围ID。在这种情况下，始终会提供一个新的跨度ID，而传入的上下文将确定父ID。

下图显示了AWS传播的示例：

┌───────────────────┐ ┌───────────────────┐

x-amzn-trace-id │ TraceContext │ │ TraceContext │

┌───────────────────┐(extract)│ ┌───────────────┐ │(join)│ ┌───────────────┐ │

│ Root │─────────┼─┼> TraceId │ │──────┼─┼> TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ Parent │─────────┼─┼> SpanId │ │──────┼─┼> ParentSpanId │ │

└───────────────────┘ │ └───────────────┘ │ │ │ │ │

└───────────────────┘ │ │ SpanId: New │ │

│ └───────────────┘ │

└───────────────────┘

注意：某些跨度报告程序不支持共享跨度ID。例如，如果设置了Tracing.Builder.spanReporter(amazonXrayOrGoogleStackdrive)，则应通过设置Tracing.Builder.supportsJoin(false)来禁用加入。这样做会在Tracer.joinSpan()上强制一个新的子跨度。

### **54.1.4实施传播**

TraceContext.Extractor<C>由Propagation.Factory插件实现。在内部，此代码使用以下之一创建联合类型TraceContextOrSamplingFlags：\* TraceContext（如果存在跟踪ID和跨度ID）。\* TraceIdContext，如果存在跟踪ID但不存在跨度ID。\* SamplingFlags，如果不存在标识符。

某些Propagation实现从提取（例如，读取传入的标头）到注入（例如，写入输出的标头）的角度携带额外的数据。例如，它可能带有请求ID。当实现中有额外数据时，它们将按以下方式处理：\*如果提取了TraceContext，则将额外数据添加为TraceContext.extra()。\*否则，将其添加为Tracer.nextSpan处理的TraceContextOrSamplingFlags.extra()。

## **55.当前跟踪组件**

Brave支持“ 当前跟踪组件 ”概念，仅在您无其他方法获得参考时才应使用。这样做是针对JDBC连接的，因为它们通常在跟踪组件之前进行初始化。

可通过Tracing.current()获得实例化的最新跟踪组件。您也可以使用Tracing.currentTracer()仅获取跟踪器。如果您使用这些方法之一，请不要缓存结果。而是在每次需要它们时查找它们。

## **56.当前Span**

Brave支持代表飞行中操作的“ 当前跨度 ”概念。您可以使用Tracer.currentSpan()将自定义标签添加到跨度，并使用Tracer.nextSpan()创建正在运行的子项。

|  |  |
| --- | --- |
| IMG_485 | **重要** |
| 在Sleuth中，您可以通过tracer.currentSpan()方法自动连接Tracer bean以检索当前跨度。要检索当前上下文，只需调用tracer.currentSpan().context()。要以字符串形式获取当前跟踪ID，可以使用traceIdString()方法，例如：tracer.currentSpan().context().traceIdString()。 |

## **56.1手动设置范围**

编写新工具时，将您创建的跨度作为当前跨度放置在示波器中很重要。这样做不仅使用户可以使用Tracer.currentSpan()访问它，而且还允许自定义文件（例如SLF4J MDC）查看当前的跟踪ID。

Tracer.withSpanInScope(Span)促进了这一点，并且通过使用try-with-resources惯用法最方便地使用。每当可能调用外部代码（例如进行拦截器或其他操作）时，请将范围放在范围内，如以下示例所示：

@Autowired Tracer tracer;

**try** (SpanInScope ws = tracer.withSpanInScope(span)) {

**return** inboundRequest.invoke();

} **finally** { *// note the scope is independent of the span*

span.finish();

}

在极端情况下，您可能需要临时清除当前跨度（例如，启动不应与当前请求关联的任务）。为此，请将null传递给withSpanInScope，如以下示例所示：

@Autowired Tracer tracer;

**try** (SpanInScope cleared = tracer.withSpanInScope(null)) {

startBackgroundThread();

}

## **57.仪器仪表**

Spring Cloud Sleuth将自动检测所有Spring应用程序，因此您无需执行任何操作即可激活它。通过根据可用的堆栈使用多种技术来添加检测。例如，对于servlet web应用程序，我们使用Filter，对于Spring Integration，我们使用ChannelInterceptors。

您可以自定义跨度标签中使用的键。为了限制范围数据的数量，默认情况下，HTTP请求仅使用少量元数据（例如状态码，主机和URL）进行标记。您可以通过配置spring.sleuth.keys.http.headers（标题名称列表）来添加请求标题。

|  |
| --- |
| IMG_486 |
| 仅当有Sampler允许标签时，才会收集和导出标签。默认情况下，没有这样的Sampler，以确保不存在不进行任何配置而意外收集太多数据的危险）。 |

## **58. Span生命周期**

您可以通过brave.Tracer在Span上执行以下操作：

* [start](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "creating-and-finishing-spans" \o "58.1创建和整理跨度)：开始跨度时，将分配其名称并记录开始时间戳。
* [close](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "creating-and-finishing-spans" \o "58.1创建和整理跨度)：跨度完成（记录跨度的结束时间），并且，如果对跨度进行了采样，则有资格进行收集（例如，到Zipkin）。
* [继续](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "continuing-spans" \o "58.2连续跨度)：创建一个新的span实例。它是继续的副本。
* [detach](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "continuing-spans" \o "58.2连续跨度)：跨度不会停止或关闭。它只会从当前线程中删除。
* [使用显式父级创建](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "creating-spans-with-explicit-parent" \o "58.3使用显式父级创建跨度)：您可以创建一个新的跨度并为其设置一个显式父级。

|  |
| --- |
| IMG_487 |
| Spring Cloud Sleuth为您创建Tracer的实例。为了使用它，您可以对其进行自动接线。 |

## **58.1创建并完成spans**

您可以使用Tracer手动创建spans，如以下示例所示：

*// Start a span. If there was a span present in this thread it will become// the `newSpan`'s parent.*

Span newSpan = **this**.tracer.nextSpan().name("calculateTax");**try** (Tracer.SpanInScope ws = **this**.tracer.withSpanInScope(newSpan.start())) {

*// ...*

*// You can tag a span*

newSpan.tag("taxValue", taxValue);

*// ...*

*// You can log an event on a span*

newSpan.annotate("taxCalculated");

}**finally** {

*// Once done remember to finish the span. This will allow collecting*

*// the span to send it to Zipkin*

newSpan.finish();

}

在前面的示例中，我们可以看到如何创建跨度的新实例。如果此线程中已经有一个跨度，它将成为新跨度的父级。

|  |  |
| --- | --- |
| IMG_488 | **重要** |
| 创建跨度后，请始终保持清洁。另外，请务必完成要发送到Zipkin的所有跨度。 |

|  |  |
| --- | --- |
| IMG_489 | **重要** |
| 如果您的跨度包含的名称大于50个字符，则该名称将被截断为50个字符。您的名字必须明确明确。知名人士会导致延迟问题，有时甚至会引发例外情况。 |

## **58.2连续跨度**

有时，您不想创建一个新跨度，但想继续一个跨度。这种情况的示例如下：

* **AOP**：如果在到达方面之前已经创建了一个跨度，则您可能不想创建一个新的跨度。
* **Hystrix**：执行Hystrix命令很可能是当前处理的逻辑部分。实际上，它仅仅是技术实现细节，您不一定要在跟踪中将其反映为一个单独的实体。

要继续跨度，可以使用brave.Tracer，如以下示例所示：

*// let's assume that we're in a thread Y and we've received// the `initialSpan` from thread X*

Span continuedSpan = **this**.tracer.toSpan(newSpan.context());**try** {

*// ...*

*// You can tag a span*

continuedSpan.tag("taxValue", taxValue);

*// ...*

*// You can log an event on a span*

continuedSpan.annotate("taxCalculated");

}**finally** {

*// Once done remember to flush the span. That means that*

*// it will get reported but the span itself is not yet finished*

continuedSpan.flush();

}

## **58.3使用显式父级创建Span**

您可能要开始一个新的跨度并提供该跨度的显式父项。假定范围的父级在一个线程中，而您想在另一个线程中开始一个新的范围。在Brave中，每当您调用nextSpan()时，它都会参照当前范围的跨度创建一个跨度。您可以将范围放入范围中，然后调用nextSpan()，如以下示例所示：

*// let's assume that we're in a thread Y and we've received// the `initialSpan` from thread X. `initialSpan` will be the parent// of the `newSpan`*

Span newSpan = null;**try** (Tracer.SpanInScope ws = **this**.tracer.withSpanInScope(initialSpan)) {

newSpan = **this**.tracer.nextSpan().name("calculateCommission");

*// ...*

*// You can tag a span*

newSpan.tag("commissionValue", commissionValue);

*// ...*

*// You can log an event on a span*

newSpan.annotate("commissionCalculated");

}**finally** {

*// Once done remember to finish the span. This will allow collecting*

*// the span to send it to Zipkin. The tags and events set on the*

*// newSpan will not be present on the parent*

**if** (newSpan != null) {

newSpan.finish();

}

}

|  |  |
| --- | --- |
| IMG_490 | **重要** |
| 创建这样的跨度后，必须完成它。否则，不会报告该错误（例如，报告到Zipkin）。 |

## **59.命名spans**

选择一个跨度名称不是一件容易的事。跨度名称应描述一个操作名称。该名称应为低基数，因此不应包含标识符。

由于正在进行很多检测，因此一些跨度名称是人为的：

* 当控制器名称为controllerMethodName的控制器接收到controller-method-name时
* async用于使用包装的Callable和Runnable接口完成的异步操作。
* 用@Scheduled注释的方法返回类的简单名称。

幸运的是，对于异步处理，您可以提供显式命名。

## **59.1 @SpanName注释**

您可以使用@SpanName注释来明确命名跨度，如以下示例所示：

@SpanName("calculateTax")

**class** TaxCountingRunnable **implements** Runnable {

@Override

**public** **void** run() {

*// perform logic*

}

}

}

在这种情况下，按以下方式处理时，跨度名为calculateTax：

Runnable runnable = **new** TraceRunnable(**this**.tracing, spanNamer,

**new** TaxCountingRunnable());

Future<?> future = executorService.submit(runnable);*// ... some additional logic ...*

future.get();

## **59.2 toString()方法**

很少为Runnable或Callable创建类。通常，创建一个匿名类的实例。您不能注释此类。为了克服该限制，如果不存在@SpanName批注，我们将检查该类是否具有toString()方法的自定义实现。

运行这样的代码将导致创建一个名为calculateTax的跨度，如以下示例所示：

Runnable runnable = **new** TraceRunnable(**this**.tracing, spanNamer, **new** Runnable() {

@Override

**public** **void** run() {

*// perform logic*

}

@Override

**public** String toString() {

**return** "calculateTax";

}

});

Future<?> future = executorService.submit(runnable);*// ... some additional logic ...*

future.get();

## **60.使用Annotations管理跨度**

您可以使用各种注释来管理spans。

## **60.1基础**

使用注释管理spans的原因很多，其中包括：

* 与API无关的方法可以与跨度进行协作。使用注释使用户可以添加到跨度，而跨度api不依赖库。这样，Sleuth可以更改其核心API，以减少对用户代码的影响。
* 减少了基本跨距操作的表面积。如果没有此功能，则必须使用span api，该api的生命周期命令可能无法正确使用。通过仅公开作用域，标记和日志功能，您可以进行协作而不会意外中断跨度生命周期。
* 与运行时生成的代码协作。使用诸如Spring Data和Feign之类的库，可以在运行时生成接口的实现。因此，对象的跨度包裹是乏味的。现在，您可以在接口和这些接口的参数上提供注释。

## **60.2创建新的跨度**

如果您不想手动创建本地spans，则可以使用@NewSpan批注。另外，我们提供了@SpanTag批注以自动方式添加标签。

现在我们可以考虑一些用法示例。

@NewSpan**void** testMethod();

在不带任何参数的情况下对方法进行注释会导致创建一个新的跨度，其名称等于带注释的方法名称。

@NewSpan("customNameOnTestMethod4")**void** testMethod4();

如果您在批注中提供值（直接或通过设置name参数），则创建的跨度将提供的值作为名称。

*// method declaration*@NewSpan(name = "customNameOnTestMethod5")**void** testMethod5(@SpanTag("testTag") String param);

*// and method execution***this**.testBean.testMethod5("test");

您可以将名称和标签结合在一起。让我们专注于后者。在这种情况下，带注释的方法的参数运行时值的值将成为标记的值。在我们的示例中，标记键为testTag，标记值为test。

@NewSpan(name = "customNameOnTestMethod3")@Override**public** **void** testMethod3() {

}

您可以在类和接口上都放置@NewSpan批注。如果您重写接口的方法并为@NewSpan批注提供一个不同的值，则最具体的将获胜（在这种情况下，将设置customNameOnTestMethod3）。

## **60.3连续跨度**

如果要将标记和注释添加到现有范围，则可以使用@ContinueSpan注释，如以下示例所示：

*// method declaration*@ContinueSpan(log = "testMethod11")**void** testMethod11(@SpanTag("testTag11") String param);

*// method execution***this**.testBean.testMethod11("test");**this**.testBean.testMethod13();

（请注意，与@NewSpan注释相反，您还可以使用log参数添加日志。）

这样，跨度将继续，并且：

* 创建名为testMethod11.before和testMethod11.after的日志条目。
* 如果引发异常，还将创建名为testMethod11.afterFailure的日志条目。
* 将创建一个标签，标签为testTag11，值为test。

## **60.4高级标签设置**

有3种不同的方法可以将标签添加到跨度。它们全部由SpanTag注释控制。优先级如下：

1. 尝试使用TagValueResolver类型的bean和提供的名称。
2. 如果未提供bean名称，请尝试计算一个表达式。我们搜索TagValueExpressionResolver bean。默认实现使用SPEL表达式解析。 **重要事项**您只能从SPEL表达式中引用属性。由于安全限制，不允许执行方法。
3. 如果找不到任何要求值的表达式，请返回参数的toString()值。

### **60.4.1定制提取器**

用于以下方法的标记的值是通过TagValueResolver接口的实现来计算的。必须将其类名作为resolver属性的值传递。

考虑以下带注释的方法：

@NewSpan**public** **void** getAnnotationForTagValueResolver(

@SpanTag(key = "test", resolver = TagValueResolver.class) String test) {

}

现在进一步考虑以下TagValueResolver bean的实现：

@Bean(name = "myCustomTagValueResolver")**public** TagValueResolver tagValueResolver() {

**return** parameter -> "Value from myCustomTagValueResolver";

}

前面的两个示例导致将标签值设置为等于Value from myCustomTagValueResolver。

### **60.4.2解析一个值的表达式**

考虑以下带注释的方法：

@NewSpan**public** **void** getAnnotationForTagValueExpression(

@SpanTag(key = "test", expression = "'hello' + ' characters'") String test) {

}

TagValueExpressionResolver的自定义实现不会导致SPEL表达式的求值，并且在跨度上设置了值为4 characters的标签。如果要使用其他表达式解析机制，则可以创建自己的bean实现。

### **60.4.3使用toString()方法**

考虑以下带注释的方法：

@NewSpan**public** **void** getAnnotationForArgumentToString(@SpanTag("test") Long param) {

}

以值15运行前面的方法会导致将标签的字符串值设置为"15"。

## **61.定制**

## **61.1定制器**

使用Brave 5.7，您可以使用多种选项为项目提供定制程序。勇敢与

* TracingCustomizer-允许配置插件协作构建Tracing的实例。
* CurrentTraceContextCustomizer-允许配置插件协作构建CurrentTraceContext的实例。
* ExtraFieldCustomizer-允许配置插件协作构建ExtraFieldPropagation.Factory的实例。

Sleuth将搜索这些类型的beans，并自动应用自定义。

## **61.2 HTTP**

如果需要定制与HTTP相关的spans的客户端/服务器解析，只需注册类型为brave.http.HttpClientParser或brave.http.HttpServerParser的bean。如果需要客户端/服务器采样，只需注册类型为brave.sampler.SamplerFunction<HttpRequest>的bean，并将bean sleuthHttpClientSampler命名为客户端采样器，将sleuthHttpServerSampler命名为服务器采样器。

为了方便起见，可以使用@HttpClientSampler和@HttpServerSampler批注注入适当的beans或通过其静态字符串NAME字段引用bean名称。

查看Brave的代码，以查看有关如何制作基于路径的采样器的示例 [https://github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy](https://github.com/openzipkin/brave/tree/master/instrumentation/http" \l "sampling-policy" \t "https://www.springcloud.cc/_top)

如果您想完全重写HttpTracing bean，则可以使用SkipPatternProvider接口检索spans不应采样的URL Pattern。在下面，您可以看到在服务器端Sampler<HttpRequest>中使用SkipPatternProvider的示例。

@Configuration**class** Config {

@Bean(name = HttpServerSampler.NAME)

SamplerFunction<HttpRequest> myHttpSampler(SkipPatternProvider provider) {

Pattern pattern = provider.skipPattern();

**return** request -> {

String url = request.path();

**boolean** shouldSkip = pattern.matcher(url).matches();

**if** (shouldSkip) {

**return** false;

}

**return** null;

};

}

}

## **61.3 TracingFilter**

您还可以修改TracingFilter的行为，该行为负责处理输入的HTTP请求并基于HTTP响应添加标签。您可以通过注册自己的TracingFilter bean实例来自定义标签或修改响应头。

在下面的示例中，我们注册TracingFilter bean，添加包含当前Span的跟踪ID的ZIPKIN-TRACE-ID响应标头，并添加带有键custom和一个值的标签tag到跨度。

@Component@Order(TraceWebServletAutoConfiguration.TRACING\_FILTER\_ORDER + 1)**class** MyFilter **extends** GenericFilterBean {

**private** **final** Tracer tracer;

MyFilter(Tracer tracer) {

**this**.tracer = tracer;

}

@Override

**public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

Span currentSpan = **this**.tracer.currentSpan();

**if** (currentSpan == null) {

chain.doFilter(request, response);

**return**;

}

*// for readability we're returning trace id in a hex form*

((HttpServletResponse) response).addHeader("ZIPKIN-TRACE-ID",

currentSpan.context().traceIdString());

*// we can also add some custom tags*

currentSpan.tag("custom", "tag");

chain.doFilter(request, response);

}

}

## **61.4 RPC**

Sleuth会自动配置RpcTracing bean，它是RPC工具（例如gRPC或Dubbo）的基础。

如果需要自定义RPC跟踪的客户端/服务器采样，只需注册类型为brave.sampler.SamplerFunction<RpcRequest>的bean，并将bean sleuthRpcClientSampler命名为客户端采样器，将sleuthRpcServerSampler命名为服务器采样器。

为了方便起见，可以使用@RpcClientSampler和@RpcServerSampler批注来注入正确的beans或通过其静态字符串NAME字段引用bean名称。

例如 这是一个每秒跟踪100个“ GetUserToken”服务器请求的采样器。这不会启动对运行状况检查服务的请求的新跟踪。其他请求将使用全局采样配置。

@Configuration**class** Config {

@Bean(name = RpcServerSampler.NAME)

SamplerFunction<RpcRequest> myRpcSampler() {

Matcher<RpcRequest> userAuth = and(serviceEquals("users.UserService"),

methodEquals("GetUserToken"));

**return** RpcRuleSampler.newBuilder()

.putRule(serviceEquals("grpc.health.v1.Health"), Sampler.NEVER\_SAMPLE)

.putRule(userAuth, RateLimitingSampler.create(100)).build();

}

}

有关更多信息，请参见[https://github.com/openzipkin/brave/tree/master/instrumentation/rpc#sampling-policy](https://github.com/openzipkin/brave/tree/master/instrumentation/rpc" \l "sampling-policy" \t "https://www.springcloud.cc/_top)

## **61.5定制服务名称**

默认情况下，Sleuth假定在将跨度发送到Zipkin时，您希望跨度的服务名称等于spring.application.name属性的值。但是，并非总是如此。在某些情况下，您想为来自应用程序的所有spans显式提供一个不同的服务名称。为此，可以将以下属性传递给应用程序以覆盖该值（该示例适用于名为myService的服务）：

spring.zipkin.service.name: myService

## **61.6报告跨度的定制**

在报告spans（例如，向Zipkin发送）之前，您可能需要以某种方式修改该范围。您可以使用FinishedSpanHandler界面执行此操作。

在Sleuth中，我们生成具有固定名称的spans。一些用户希望根据标签的值来修改名称。您可以实现FinishedSpanHandler接口来更改该名称。

以下示例显示如何注册两个实现FinishedSpanHandler的beans：

@Bean

FinishedSpanHandler handlerOne() {

**return** **new** FinishedSpanHandler() {

@Override

**public** **boolean** handle(TraceContext traceContext, MutableSpan span) {

span.name("foo");

**return** true; *// keep this span*

}

};

}

@Bean

FinishedSpanHandler handlerTwo() {

**return** **new** FinishedSpanHandler() {

@Override

**public** **boolean** handle(TraceContext traceContext, MutableSpan span) {

span.name(span.name() + " bar");

**return** true; *// keep this span*

}

};

}

前面的示例导致报告的跨度的名称刚好在报告之前更改为foo bar（例如，更改为Zipkin）。

## **61.7主机定位器**

|  |  |
| --- | --- |
| IMG_491 | **重要** |
| 本节是关于通过服务发现定义**主机**。这是**不是**要找到Zipkin通过服务发现。 |

要定义与特定跨度相对应的主机，我们需要解析主机名和端口。默认方法是从服务器属性中获取这些值。如果未设置，则尝试从网络接口检索主机名。

如果您启用了发现客户端，并且希望从服务注册表中的已注册实例中检索主机地址，则必须设置spring.zipkin.locator.discovery.enabled属性（该属性适用于基于HTTP和基于流的跨度报告），如下：

spring.zipkin.locator.discovery.enabled: **true**

## **62.将跨度发送到Zipkin**

默认情况下，如果将spring-cloud-starter-zipkin作为依赖项添加到项目，则关闭跨度后，跨度将通过HTTP发送到Zipkin。通信是异步的。您可以通过设置spring.zipkin.baseUrl属性来配置URL，如下所示：

spring.zipkin.baseUrl: https://192.168.99.100:9411/

如果您想通过服务发现来找到Zipkin，则可以在URL内传递Zipkin的服务ID，如以下zipkinserver服务ID的示例所示：

spring.zipkin.baseUrl: http://zipkinserver/

要禁用此功能，只需将spring.zipkin.discoveryClientEnabled设置为`false。

启用发现客户端功能后，Sleuth使用LoadBalancerClient查找Zipkin服务器的URL。这意味着您可以设置负载平衡配置，例如通过Ribbon。

zipkinserver: ribbon: ListOfServers: host1,host2

如果在类路径上一起有web，rabbit或kafka，则可能需要选择将spans发送到zipkin的方式。为此，请将web，rabbit或kafka设置为spring.zipkin.sender.type属性。以下示例显示了为web设置发件人类型：

spring.zipkin.sender.type: web

要自定义通过HTTP发送spans到Zipkin的RestTemplate，可以注册ZipkinRestTemplateCustomizer bean。

@Configuration**class** MyConfig {

@Bean ZipkinRestTemplateCustomizer myCustomizer() {

**return** **new** ZipkinRestTemplateCustomizer() {

@Override

**void** customize(RestTemplate restTemplate) {

*// customize the RestTemplate*

}

};

}

}

但是，如果您想控制创建RestTemplate对象的整个过程，则必须创建zipkin2.reporter.Sender类型的bean。

@Bean Sender myRestTemplateSender(ZipkinProperties zipkin,

ZipkinRestTemplateCustomizer zipkinRestTemplateCustomizer) {

RestTemplate restTemplate = mySuperCustomRestTemplate();

zipkinRestTemplateCustomizer.customize(restTemplate);

**return** myCustomSender(zipkin, restTemplate);

}

## **63. Zipkin流Span消费者**

|  |  |
| --- | --- |
| IMG_492 | **重要** |
| 我们建议对基于消息的跨度发送使用Zipkin的本机支持。从Edgware版本开始，不推荐使用Zipkin Stream服务器。在Finchley版本中，将其删除。 |

如果出于某种原因需要创建不赞成使用的Stream Zipkin服务器，请参阅[Dalston文档](https://cloud.spring.io/spring-cloud-static/Dalston.SR4/multi/multi__span_data_as_messages.html" \l "_zipkin_consumer" \t "https://www.springcloud.cc/_top)。

## **64.整合**

## **64.1 OpenTracing**

Spring Cloud Sleuth与[OpenTracing](https://opentracing.io/" \t "https://www.springcloud.cc/_top)兼容。如果您在类路径上具有OpenTracing，我们将自动注册OpenTracing Tracer bean。如果您要禁用此功能，请将spring.sleuth.opentracing.enabled设置为false

## **64.2可运行和可调用**

如果将逻辑包装在Runnable或Callable中，则可以将这些类包装在其Sleuth代表中，如以下Runnable的示例所示：

Runnable runnable = **new** Runnable() {

@Override

**public** **void** run() {

*// do some work*

}

@Override

**public** String toString() {

**return** "spanNameFromToStringMethod";

}

};*// Manual `TraceRunnable` creation with explicit "calculateTax" Span name*

Runnable traceRunnable = **new** TraceRunnable(**this**.tracing, spanNamer, runnable,

"calculateTax");*// Wrapping `Runnable` with `Tracing`. That way the current span will be available// in the thread of `Runnable`*

Runnable traceRunnableFromTracer = **this**.tracing.currentTraceContext()

.wrap(runnable);

以下示例显示了如何对Callable执行此操作：

Callable<String> callable = **new** Callable<String>() {

@Override

**public** String call() **throws** Exception {

**return** someLogic();

}

@Override

**public** String toString() {

**return** "spanNameFromToStringMethod";

}

};*// Manual `TraceCallable` creation with explicit "calculateTax" Span name*

Callable<String> traceCallable = **new** TraceCallable<>(**this**.tracing, spanNamer,

callable, "calculateTax");*// Wrapping `Callable` with `Tracing`. That way the current span will be available// in the thread of `Callable`*

Callable<String> traceCallableFromTracer = **this**.tracing.currentTraceContext()

.wrap(callable);

这样，您可以确保为每个执行创建并关闭新的跨度。

## **64.3 Hystrix**

### **64.3.1自定义并发策略**

我们注册了一个[HystrixConcurrencyStrategy](https://github.com/Netflix/Hystrix/wiki/Plugins" \l "concurrencystrategy" \t "https://www.springcloud.cc/_top)名为TraceCallable 的自定义，该自定义将所有Callable实例包装在其Sleuth代表中。根据调用Hystrix命令之前是否已经进行了跟踪，该策略将开始还是继续跨度。要禁用自定义Hystrix并发策略，请将spring.sleuth.hystrix.strategy.enabled设置为false。

### **64.3.2手动命令设置**

假定您具有以下HystrixCommand：

HystrixCommand<String> hystrixCommand = **new** HystrixCommand<String>(setter) {

@Override

**protected** String run() **throws** Exception {

**return** someLogic();

}

};

要传递跟踪信息，必须在HystrixCommand的Sleuth版本中包装相同的逻辑，称为TraceCommand，如以下示例所示：

TraceCommand<String> traceCommand = **new** TraceCommand<String>(tracer, setter) {

@Override

**public** String doRun() **throws** Exception {

**return** someLogic();

}

};

## **64.4 RxJava**

我们注册了一个自定义[RxJavaSchedulersHook](https://github.com/ReactiveX/RxJava/wiki/Plugins" \l "rxjavaschedulershook" \t "https://www.springcloud.cc/_top)，该自定义将所有Action0实例包装在其Sleuth代表中，称为TraceAction。挂钩将开始或继续跨度，具体取决于在计划操作之前是否已经进行了跟踪。要禁用自定义RxJavaSchedulersHook，请将spring.sleuth.rxjava.schedulers.hook.enabled设置为false。

您可以为不想为其创建spans的线程名称定义一个正则表达式列表。为此，请在spring.sleuth.rxjava.schedulers.ignoredthreads属性中提供用逗号分隔的正则表达式列表。

|  |  |
| --- | --- |
| IMG_493 | **重要** |
| 建议使用反应式编程和Sleuth的方法是使用Reactor支持。 |

## **64.5 HTTP集成**

可以通过将spring.sleuth.web.enabled属性设置为等于false的值来禁用此部分中的功能。

### **64.5.1 HTTP筛选器**

通过TracingFilter，所有采样的传入请求都将导致创建Span。Span的名称为http: +请求发送到的路径。例如，如果请求已发送到/this/that，则名称将为http:/this/that。您可以通过设置spring.sleuth.web.skipPattern属性来配置要跳过的URI。如果类路径上有ManagementServerProperties，则其值contextPath将附加到提供的跳过模式中。如果要重用Sleuth的默认跳过模式并仅添加自己的模式，请使用spring.sleuth.web.additionalSkipPattern传递这些模式。

默认情况下，所有spring boot执行器端点都会自动添加到跳过模式中。如果要禁用此行为，请将spring.sleuth.web.ignore-auto-configured-skip-patterns设置为true。

要更改跟踪过滤器注册的顺序，请设置spring.sleuth.web.filter-order属性。

要禁用记录未捕获异常的过滤器，可以禁用spring.sleuth.web.exception-throwing-filter-enabled属性。

### **64.5.2 HandlerInterceptor**

由于我们希望跨度名称精确，因此我们使用TraceHandlerInterceptor来包装现有的HandlerInterceptor或将其直接添加到现有的HandlerInterceptors的列表中。TraceHandlerInterceptor向给定的HttpServletRequest添加一个特殊的请求属性。如果TracingFilter没有看到此属性，它将创建一个“ fallback ”跨度，这是在服务器端创建的另一个跨度，以便在UI中正确显示跟踪。如果发生这种情况，可能是缺少仪器。在这种情况下，请在Spring Cloud Sleuth中提出问题。

### **64.5.3异步Servlet支持**

如果您的控制器返回Callable或WebAsyncTask，则Spring Cloud Sleuth会继续使用现有跨度而不是创建新跨度。

### **64.5.4 WebFlux支持**

通过TraceWebFilter，所有采样的传入请求都将导致创建Span。Span的名称为http: +请求发送到的路径。例如，如果请求已发送到/this/that，则名称为http:/this/that。您可以使用spring.sleuth.web.skipPattern属性来配置要跳过的URI。如果类路径上有ManagementServerProperties，则其值contextPath将附加到提供的跳过模式中。如果要重用Sleuth的默认跳过模式并追加自己的跳过模式，请使用spring.sleuth.web.additionalSkipPattern传递这些模式。

要更改跟踪过滤器注册的顺序，请设置spring.sleuth.web.filter-order属性。

### **64.5.5 Dubbo RPC支持**

通过与Brave的集成，Spring Cloud Sleuth支持[Dubbo](https://dubbo.apache.org/" \t "https://www.springcloud.cc/_top)。添加brave-instrumentation-dubbo依赖项就足够了：

<dependency>

<groupId>io.zipkin.brave</groupId>

<artifactId>brave-instrumentation-dubbo</artifactId></dependency>

您还需要设置一个包含以下内容的dubbo.properties文件：

dubbo.provider.filter=tracingdubbo.consumer.filter=tracing

您可以[在此处](https://github.com/openzipkin/brave/tree/master/instrumentation/dubbo-rpc" \t "https://www.springcloud.cc/_top)阅读有关Brave-Dubbo集成的更多信息。Spring Cloud Sleuth和Dubbo的示例可以在[此处](https://github.com/openzipkin/sleuth-webmvc-example/compare/add-dubbo-tracing" \t "https://www.springcloud.cc/_top)找到。

## **64.6 HTTP客户端集成**

### **64.6.1同步休息模板**

我们注入了RestTemplate拦截器，以确保所有跟踪信息都传递给请求。每次拨打电话时，都会创建一个新的Span。收到响应后关闭。要阻止同步RestTemplate功能，请将spring.sleuth.web.client.enabled设置为false。

|  |  |
| --- | --- |
| IMG_494 | **重要** |
| 您必须将RestTemplate注册为bean，以便注入拦截器。如果使用关键字new创建一个RestTemplate实例，则检测不起作用。 |

### **64.6.2异步休息模板**

|  |  |
| --- | --- |
| IMG_495 | **重要** |
| 从Sleuth 2.0.0开始，我们不再注册AsyncRestTemplate类型的bean。创建这样的bean取决于您。然后我们对其进行检测。 |

要阻止AsyncRestTemplate功能，请将spring.sleuth.web.async.client.enabled设置为false。要禁用默认TraceAsyncClientHttpRequestFactoryWrapper的创建，请将spring.sleuth.web.async.client.factory.enabled设置为false。如果根本不想创建AsyncRestClient，请将spring.sleuth.web.async.client.template.enabled设置为false。

#### **多个异步休息模板**

有时您需要使用异步实现模板的多个实现。在以下代码段中，您可以看到一个如何设置这样的自定义AsyncRestTemplate的示例：

@Configuration@EnableAutoConfiguration**static** **class** Config {

@Bean(name = "customAsyncRestTemplate")

**public** AsyncRestTemplate traceAsyncRestTemplate() {

**return** **new** AsyncRestTemplate(asyncClientFactory(),

clientHttpRequestFactory());

}

**private** ClientHttpRequestFactory clientHttpRequestFactory() {

ClientHttpRequestFactory clientHttpRequestFactory = **new** CustomClientHttpRequestFactory();

*// CUSTOMIZE HERE*

**return** clientHttpRequestFactory;

}

**private** AsyncClientHttpRequestFactory asyncClientFactory() {

AsyncClientHttpRequestFactory factory = **new** CustomAsyncClientHttpRequestFactory();

*// CUSTOMIZE HERE*

**return** factory;

}

}

### **64.6.3 WebClient**

我们注入了一个ExchangeFilterFunction实现，该实现创建了一个范围，并通过成功和错误时回调，负责关闭客户端spans。

要阻止此功能，请将spring.sleuth.web.client.enabled设置为false。

|  |  |
| --- | --- |
| IMG_496 | **重要** |
| 您必须将WebClient注册为bean，以便应用跟踪工具。如果您使用关键字new创建一个WebClient实例，则检测无效。 |

### **64.6.4特拉弗森**

如果您使用[Traverson](https://docs.spring.io/spring-hateoas/docs/current/reference/html/" \l "client.traverson" \t "https://www.springcloud.cc/_top)库，则可以将RestTemplate作为bean注入Traverson对象。由于RestTemplate已被拦截，因此您将获得对客户端中跟踪的完全支持。以下伪代码显示了如何执行此操作：

@Autowired RestTemplate restTemplate;

Traverson traverson = **new** Traverson(URI.create("http://some/address"),

MediaType.APPLICATION\_JSON, MediaType.APPLICATION\_JSON\_UTF8).setRestOperations(restTemplate);*// use Traverson*

### **64.6.5 Apache HttpClientBuilder和HttpAsyncClientBuilder**

我们对HttpClientBuilder和HttpAsyncClientBuilder进行检测，以便将跟踪上下文注入已发送的请求中。

要阻止这些功能，请将spring.sleuth.web.client.enabled设置为false。

### **64.6.6净值HttpClient**

我们对Netty的HttpClient进行检测。

要阻止此功能，请将spring.sleuth.web.client.enabled设置为false。

|  |  |
| --- | --- |
| IMG_497 | **重要** |
| 您必须将HttpClient注册为bean，以便进行检测。如果您使用new关键字创建HttpClient实例，则检测无效。 |

### **64.6.7 UserInfoRestTemplateCustomizer**

我们检测Spring Security的UserInfoRestTemplateCustomizer。

要阻止此功能，请将spring.sleuth.web.client.enabled设置为false。

## **64.7 Feign**

默认情况下，Spring Cloud Sleuth与Feign到TraceFeignClientAutoConfiguration集成。您可以通过将spring.sleuth.feign.enabled设置为false来完全禁用它。如果这样做，则不会发生与Feign相关的检测。

Feign工具的一部分是通过FeignBeanPostProcessor完成的。您可以通过将spring.sleuth.feign.processor.enabled设置为false来禁用它。如果将其设置为false，则Spring Cloud Sleuth不会检测任何自定义的Feign组件。但是，所有默认工具仍然存在。

## **64.8 gRPC**

Spring Cloud Sleuth 通过TraceGrpcAutoConfiguration 为[gRPC](https://grpc.io/" \t "https://www.springcloud.cc/_top)提供了工具。您可以通过将spring.sleuth.grpc.enabled设置为false来完全禁用它。

### **64.8.1变体1**

#### **依存关系**

|  |  |
| --- | --- |
| IMG_498 | **重要** |
| gRPC集成依赖于两个外部库来检测客户端和服务器，并且这两个库都必须位于类路径中才能启用检测。 |

Maven:

<dependency>

<groupId>io.github.lognet</groupId>

<artifactId>grpc-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.brave</groupId>

<artifactId>brave-instrumentation-grpc</artifactId>

</dependency>

Gradle:

compile("io.github.lognet:grpc-spring-boot-starter")

compile("io.zipkin.brave:brave-instrumentation-grpc")

#### **服务器检测**

Spring Cloud Sleuth利用grpc-spring-boot-starter向带有@GRpcService注释的所有服务注册Brave的gRPC服务器拦截器。

#### **客户端工具**

gRPC客户端利用ManagedChannelBuilder来构造用于与gRPC服务器通信的ManagedChannel。本机ManagedChannelBuilder提供静态方法作为构建ManagedChannel实例的入口点，但是，此机制不受Spring应用程序上下文的影响。

|  |  |
| --- | --- |
| IMG_499 | **重要** |
| Spring Cloud Sleuth提供了一个SpringAwareManagedChannelBuilder，可以通过Spring应用程序上下文对其进行自定义，并由gRPC客户端注入。**创建**ManagedChannel**实例时必须使用此构建器。** |

侦探创建了一个TracingManagedChannelBuilderCustomizer，将Brave的客户端拦截器注入到SpringAwareManagedChannelBuilder中。

### **64.8.2变体2**

[Grpc Spring Boot Starter](https://github.com/yidongnan/grpc-spring-boot-starter" \t "https://www.springcloud.cc/_top)自动检测到Spring Cloud Sleuth的存在以及brave为gRPC提供的工具，并注册了必要的客户端和/或服务器工具。

## **64.9异步通信**

### **64.9.1 @Async带注释的方法**

在Spring Cloud Sleuth中，我们检测与异步相关的组件，以便在线程之间传递跟踪信息。您可以通过将spring.sleuth.async.enabled的值设置为false来禁用此行为。

如果您使用@Async注释方法，我们将自动创建具有以下特征的新Span：

* 如果该方法用@SpanName注释，则注释的值为Span的名称。
* 如果该方法未使用@SpanName进行注释，则Span名称是带有注释的方法名称。
* 该范围用方法的类名和方法名标记。

### **64.9.2 @Scheduled带注释的方法**

在Spring Cloud Sleuth中，我们对调度的方法执行进行检测，以便在线程之间传递跟踪信息。您可以通过将spring.sleuth.scheduled.enabled的值设置为false来禁用此行为。

如果您使用@Scheduled注释方法，我们将自动创建具有以下特征的新跨度：

* 跨度名称是带注释的方法名称。
* 该范围用方法的类名和方法名标记。

如果要跳过某些带有@Scheduled注释的类的跨度创建，则可以使用与@Scheduled带注释的类的标准名称匹配的正则表达式来设置spring.sleuth.scheduled.skipPattern。如果同时使用spring-cloud-sleuth-stream和spring-cloud-netflix-hystrix-stream，则会为每个Hystrix指标创建一个范围，并将其发送到Zipkin。这种行为可能很烦人。这就是默认情况下spring.sleuth.scheduled.skipPattern=org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask的原因。

### **64.9.3 Executor，ExecutorService和ScheduledExecutorService**

我们提供LazyTraceExecutor，TraceableExecutorService和TraceableScheduledExecutorService。每次提交，调用或计划新任务时，这些实现都会创建spans。

以下示例显示了使用CompletableFuture时如何将跟踪信息传递给TraceableExecutorService：

CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {

*// perform some logic*

**return** 1\_000\_000L;

}, **new** TraceableExecutorService(beanFactory, executorService,

*// 'calculateTax' explicitly names the span - this param is optional*

"calculateTax"));

|  |  |
| --- | --- |
| IMG_500 | **重要** |
| Sleuth不适用于parallelStream()。如果要使跟踪信息通过流传播，则必须使用supplyAsync(…​)的方法，如前所示。 |

如果有beans实现了您想从跨度创建中排除的Executor接口，则可以使用spring.sleuth.async.ignored-beans属性，在其中可以提供bean名称的列表。

#### **定制执行者**

有时，您需要设置AsyncExecutor的自定义实例。以下示例显示如何设置这样的自定义Executor：

@Configuration@EnableAutoConfiguration@EnableAsync*// add the infrastructure role to ensure that the bean gets auto-proxied*@Role(BeanDefinition.ROLE\_INFRASTRUCTURE)**static** **class** CustomExecutorConfig **extends** AsyncConfigurerSupport {

@Autowired

BeanFactory beanFactory;

@Override

**public** Executor getAsyncExecutor() {

ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();

*// CUSTOMIZE HERE*

executor.setCorePoolSize(7);

executor.setMaxPoolSize(42);

executor.setQueueCapacity(11);

executor.setThreadNamePrefix("MyExecutor-");

*// DON'T FORGET TO INITIALIZE*

executor.initialize();

**return** **new** LazyTraceExecutor(**this**.beanFactory, executor);

}

}

|  |
| --- |
| IMG_501 |
| 为确保对配置进行后期处理，请记住在@Configuration类上添加@Role(BeanDefinition.ROLE\_INFRASTRUCTURE) |

## **64.10消息传递**

通过将spring.sleuth.messaging.enabled属性设置为等于false的值，可以禁用本节中的功能。

### **64.10.1 Spring Integration和Spring Cloud Stream**

Spring Cloud Sleuth与[Spring Integration](https://projects.spring.io/spring-integration/" \t "https://www.springcloud.cc/_top)集成。它为发布和订阅事件创建spans。要禁用Spring Integration检测，请将​​spring.sleuth.integration.enabled设置为false。

您可以提供spring.sleuth.integration.patterns模式来显式提供要包括以进行跟踪的通道的名称。默认情况下，除hystrixStreamOutput通道以外的所有通道都包括在内。

|  |  |
| --- | --- |
| IMG_502 | **重要** |
| 使用Executor构建Spring Integration IntegrationFlow时，必须使用Executor的未跟踪版本。用TraceableExecutorService装饰Spring Integration执行器通道会导致spans被不正确地关闭。 |

如果要自定义从消息头读取和向消息头写入跟踪上下文的方式，就足以注册类型的beans：

* Propagation.Setter<MessageHeaderAccessor, String>-用于将标头写入消息
* Propagation.Getter<MessageHeaderAccessor, String>-用于从邮件中读取标题

### **64.10.2 Spring RabbitMq**

我们对RabbitTemplate进行检测，以便将跟踪标头注入到消息中。

要阻止此功能，请将spring.sleuth.messaging.rabbit.enabled设置为false。

### **64.10.3 Spring Kafka**

我们检测Spring Kafka的ProducerFactory和ConsumerFactory，以便将跟踪标头注入到创建的Spring Kafka的Producer和Consumer中。

要阻止此功能，请将spring.sleuth.messaging.kafka.enabled设置为false。

### **64.10.4 Spring JMS**

我们对JmsTemplate进行检测，以便将跟踪标头注入到消息中。我们还在消费者方面支持@JmsListener带注释的方法。

要阻止此功能，请将spring.sleuth.messaging.jms.enabled设置为false。

|  |  |
| --- | --- |
| IMG_503 | **重要** |
| 我们不支持JMS的行李运送 |

## **64.11 Zuul**

我们通过跟踪信息丰富Ribbon请求来检测Zuul Ribbon集成。要禁用Zuul支持，请将spring.sleuth.zuul.enabled属性设置为false。

## **64.12项目Reactor**

对于取决于项目Reactor的项目，例如Spring Cloud网关，建议将spring.sleuth.reactor.decorate-on-each选项设置为false。这样，与标准仪器机制相比，应观察到性能提高。此选项的作用是它将包装修饰符onLast而不是onEach，这将导致创建的对象少得多。缺点是，当项目Reactor将更改线程时，跟踪传播将继续进行而不会出现问题，但是依赖于ThreadLocal的任何内容（例如MDC条目）都可能出现故障。

## **65.运行示例**

您可以看到在[Pivotal Web服务中](https://run.pivotal.io/" \t "https://www.springcloud.cc/_top)部署的正在运行的示例。在以下链接中查看它们：

* [Zipkin用于示例顶部的应用程序](https://docssleuth-zipkin-server.cfapps.io/" \t "https://www.springcloud.cc/_top)。首先向[服务1](https://docssleuth-service1.cfapps.io/start" \t "https://www.springcloud.cc/_top)发出请求，然后检查Zipkin中的跟踪。
* [Zipkin用于PWS上的Brewery](https://docsbrewing-zipkin-server.cfapps.io/" \t "https://www.springcloud.cc/_top)，其[Github代码](https://github.com/spring-cloud-samples/brewery" \t "https://www.springcloud.cc/_top)。确保选择了7天的回溯期。如果没有任何痕迹，请转到“ [呈现应用程序”](https://docsbrewing-presenting.cfapps.io/" \t "https://www.springcloud.cc/_top) 并订购一些啤酒。然后检查Zipkin是否有痕迹。

# **第九部分 Spring Cloud Consul**

**Greenwich SR5**

该项目通过自动配置并绑定到Spring环境和其他Spring编程模型习惯用法，为Spring Boot应用提供了Consul集成。通过一些简单的注释，您可以快速启用和配置应用程序内部的通用模式，并使用基于Consul的组件构建大型分布式系统。提供的模式包括服务发现，控制Bus和配置。通过与Spring Cloud Netflix集成，可以提供智能路由（Zuul）和客户端负载平衡（Ribbon），断路器（Hystrix）。

## **66.安装Consul**

请参阅[安装文档](https://www.consul.io/intro/getting-started/install.html" \t "https://www.springcloud.cc/_top)以获取有关如何安装Consul的说明。

## **67. Consul Agent**

Consul Agent客户端必须可用于所有Spring Cloud Consul应用程序。默认情况下，代理客户端应位于localhost:8500。有关如何启动代理客户端以及如何连接到Consul Agent服务器集群的详细信息，请参阅[代理文档](https://consul.io/docs/agent/basics.html" \t "https://www.springcloud.cc/_top)。为了进行开发，安装consul后，可以使用以下命令启动Consul Agent：

./src/main/bash/local\_run\_consul.sh

这将在服务器模式下的端口8500上启动代理，并且ui可从[http：// localhost：8500获得。](http://localhost:8500/" \t "https://www.springcloud.cc/_top)

## **68.使用Consul进行服务发现**

服务发现是基于微服务的体系结构的关键原则之一。尝试手动配置每个客户端或某种形式的约定可能非常困难并且非常脆弱。Consul通过[HTTP API](https://www.consul.io/docs/agent/http.html" \t "https://www.springcloud.cc/_top)和[DNS](https://www.consul.io/docs/agent/dns.html" \t "https://www.springcloud.cc/_top)提供服务发现服务。Spring Cloud Consul利用HTTP API进行服务注册和发现。这不会阻止非Spring Cloud应用程序利用DNS接口。Consul代理服务器在[群集中](https://www.consul.io/docs/internals/architecture.html" \t "https://www.springcloud.cc/_top)运行，该[群集](https://www.consul.io/docs/internals/architecture.html" \t "https://www.springcloud.cc/_top)通过[八卦协议进行通信](https://www.consul.io/docs/internals/gossip.html" \t "https://www.springcloud.cc/_top)并使用[Raft共识协议](https://www.consul.io/docs/internals/consensus.html" \t "https://www.springcloud.cc/_top)。

## **68.1如何激活**

要激活Consul服务发现，请将启动器与组org.springframework.cloud和工件ID spring-cloud-starter-consul-discovery一起使用。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

## **68.2向Consul注册**

当客户端向Consul注册时，它将提供有关其自身的元数据，例如主机和端口，id，名称和标签。默认情况下，会创建一个HTTP [检查，](https://www.consul.io/docs/agent/checks.html" \t "https://www.springcloud.cc/_top)该[检查](https://www.consul.io/docs/agent/checks.html" \t "https://www.springcloud.cc/_top)每10秒Consul命中/health端点。如果运行状况检查失败，则将该服务实例标记为关键。

示例Consul客户：

@SpringBootApplication@RestController**public** **class** Application {

@RequestMapping("/")

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

（即完全正常的Spring Boot应用）。如果Consul客户端位于localhost:8500之外的其他位置，则需要进行配置才能找到该客户端。例：

**application.yml。**

spring:

cloud:

consul:

host: localhost

port: 8500

|  |  |
| --- | --- |
| IMG_504 | **警告** |
| 如果使用[Spring Cloud Consul Config](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config" \o "69.使用Consul进行分布式配置)，则需要将以上值放置在bootstrap.yml中，而不是application.yml中。 |

来自Environment的默认服务名称，实例ID和端口分别为${spring.application.name}，Spring上下文ID和${server.port}。

要禁用Consul发现客户端，可以将spring.cloud.consul.discovery.enabled设置为false。当spring.cloud.discovery.enabled设置为false时，Consul Discovery Client也将被禁用。

要禁用服务注册，可以将spring.cloud.consul.discovery.register设置为false。

### **68.2.1将管理注册为单独的服务**

如果将管理服务器端口设置为与应用程序端口不同的端口，则通过设置management.server.port属性，管理服务将被注册为与应用程序服务不同的服务。例如：

**application.yml。**

spring:

application:

name: myApp

management:

server:

port: 4452

以上配置将注册以下两项服务：

* 申请服务：

ID: myApp

Name: myApp

* 管理服务：

ID: myApp-management

Name: myApp-management

管理服务将从应用程序服务继承其instanceId和serviceName。例如：

**application.yml。**

spring:

application:

name: myApp

management:

server:

port: 4452

spring:

cloud:

consul:

discovery:

instance-id: custom-service-id

serviceName: myprefix-${spring.application.name}

以上配置将注册以下两项服务：

* 申请服务：

ID: custom-service-id

Name: myprefix-myApp

* 管理服务：

ID: custom-service-id-management

Name: myprefix-myApp-management

通过以下属性可以进行进一步的自定义：

/\*\* Port to register the management service under (defaults to management port) \*/

spring.cloud.consul.discovery.management-port

/\*\* Suffix to use when registering management service (defaults to "management" \*/

spring.cloud.consul.discovery.management-suffix

/\*\* Tags to use when registering management service (defaults to "management" \*/

spring.cloud.consul.discovery.management-tags

## **68.3 HTTP运行状况检查**

Consul实例的运行状况检查默认为“ / health”，这是Spring Boot Actuator应用程序中有用端点的默认位置。如果您使用非默认上下文路径或Servlet路径（例如server.servletPath=/foo）或管理端点路径（例如management.server.servlet.context-path=/admin），则即使对于Actuator应用程序，也需要更改它们。也可以配置Consul用于检查运行状况端点的间隔。“ 10s”和“ 1m”分别代表10秒和1分钟。例：

**application.yml。**

spring:

cloud:

consul:

discovery:

healthCheckPath: ${management.server.servlet.context-path}/health

healthCheckInterval: 15s

您可以通过设置management.health.consul.enabled=false来禁用运行状况检查。

### **68.3.1元数据和Consul标签**

Consul尚不支持有关服务的元数据。Spring Cloud的ServiceInstance有一个Map<String, String> metadata字段。Spring Cloud Consul使用Consul标签来近似元数据，直到Consul正式支持元数据。格式为key=value的标签将被拆分并分别用作Map键和值。没有等号=的标记将用作键和值。

**application.yml。**

spring:

cloud:

consul:

discovery:

tags: foo=bar, baz

上面的配置将生成带有foo→bar和baz→baz的映射。

### **68.3.2使Consul实例ID唯一**

默认情况下，consul实例注册的ID与其Spring应用程序上下文ID相同。默认情况下，Spring应用程序上下文ID为${spring.application.name}:comma,separated,profiles:${server.port}。在大多数情况下，这将允许一项服务的多个实例在一台计算机上运行。如果需要进一步的唯一性，则可以使用Spring Cloud在spring.cloud.consul.discovery.instanceId中提供唯一的标识符来覆盖它。例如：

**application.yml。**

spring:

cloud:

consul:

discovery:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

有了此元数据，并在本地主机上部署了多个服务实例，随机值将在其中加入以使实例唯一。在Cloudfoundry中，vcap.application.instance\_id将自动在Spring Boot应用程序中填充，因此将不需要随机值。

### **68.3.3将标头应用于健康检查请求**

标头可以应用于健康检查请求。例如，如果您尝试注册使用[Vault Backend](https://github.com/spring-cloud/spring-cloud-config/blob/master/docs/src/main/asciidoc/spring-cloud-config.adoc" \l "vault-backend" \t "https://www.springcloud.cc/_top)的[Spring Cloud Config](https://cloud.spring.io/spring-cloud-config/" \t "https://www.springcloud.cc/_top)服务器：

**application.yml。**

spring:

cloud:

consul:

discovery:

health-check-headers:

X-Config-Token: 6442e58b-d1ea-182e-cfa5-cf9cddef0722

根据HTTP标准，每个标头可以有多个值，在这种情况下，可以提供一个数组：

**application.yml。**

spring:

cloud:

consul:

discovery:

health-check-headers:

X-Config-Token:

- "6442e58b-d1ea-182e-cfa5-cf9cddef0722"

- "Some other value"

## **68.4查找服务**

### **68.4.1使用Ribbon**

Spring Cloud支持[Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-feign" \t "https://www.springcloud.cc/_top)（REST客户端构建器），还支持[Spring RestTemplate，](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-ribbon" \t "https://www.springcloud.cc/_top) 以使用逻辑服务名称/标识而不是物理URL查找服务。Feign和发现感知的RestTemplate都使用[Ribbon](https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html" \l "spring-cloud-ribbon" \t "https://www.springcloud.cc/_top)进行客户端负载平衡。

如果要使用RestTemplate访问服务STORES，只需声明：

@LoadBalanced

@Bean

public RestTemplate loadbalancedRestTemplate() {

new RestTemplate();

}

并以这种方式使用它（注意我们如何使用Consul中的STORES服务名称/ id而不是完全限定的域名）：

@Autowired

RestTemplate restTemplate;

public String getFirstProduct() {

return this.restTemplate.getForObject("https://STORES/products/1", String.class);

}

如果您在多个数据中心中有Consul个群集，并且要访问另一个数据中心中的服务，则仅靠服务名称/ id是不够的。在这种情况下，请使用属性spring.cloud.consul.discovery.datacenters.STORES=dc-west，其中STORES是服务名称/ id，而dc-west是STORES服务所在的数据中心。

### **68.4.2使用DiscoveryClient**

您还可以使用org.springframework.cloud.client.discovery.DiscoveryClient，它为发现客户端提供了一个简单的API，它不是特定于Netflix的，例如

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

}

## **68.5 Consul目录手表**

Consul目录监视利用consul [监视服务的能力](https://www.consul.io/docs/agent/watches.html" \l "services" \t "https://www.springcloud.cc/_top)。Catalog Watch进行阻塞Consul HTTP API调用，以确定是否有任何服务已更改。如果有新的服务数据，则会发布心跳事件。

要更改称为“配置监视”的频率，请更改spring.cloud.consul.config.discovery.catalog-services-watch-delay。默认值为1000，以毫秒为单位。延迟是上一次调用结束与下一次调用开始之间的时间量。

要禁用目录监视集spring.cloud.consul.discovery.catalogServicesWatch.enabled=false。

手表使用Spring TaskScheduler将通话安排到consul。默认情况下，它是ThreadPoolTaskScheduler，其poolSize为1。要更改TaskScheduler，请创建一个类型为TaskScheduler的bean，名称为ConsulDiscoveryClientConfiguration.CATALOG\_WATCH\_TASK\_SCHEDULER\_NAME常量。

## **69.使用Consul的分布式配置**

Consul提供了用于存储配置和其他元数据的[键/值存储](https://consul.io/docs/agent/http/kv.html" \t "https://www.springcloud.cc/_top)。Spring Cloud Consul Config是[Config Server和Client](https://github.com/spring-cloud/spring-cloud-config" \t "https://www.springcloud.cc/_top)的替代方法。在特殊的“引导”阶段，配置被加载到Spring环境中。默认情况下，配置存储在/config文件夹中。基于应用程序的名称和模拟解析属性的Spring Cloud Config顺序的活动配置文件，将创建多个PropertySource实例。例如，名称为“ testApp”且配置文件为“ dev”的应用程序将创建以下属性源：

config/testApp,dev/

config/testApp/

config/application,dev/

config/application/

最具体的属性来源在顶部，最不具体的属性在底部。config/application文件夹中的Properties适用于使用consul进行配置的所有应用程序。config/testApp文件夹中的Properties仅可用于名为“ testApp”的服务的实例。

当前在启动应用程序时读取配置。向/refresh发送HTTP POST将导致重新加载配置。[第69.3节“ Config Watch”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-consul-config-watch" \o "69.3配置观察)还将自动检测更改并重新加载应用程序上下文。

## **69.1如何激活**

要开始使用Consul配置，请使用组org.springframework.cloud和工件ID为​​spring-cloud-starter-consul-config的启动器。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

这将启用将配置Spring Cloud Consul Config的自动配置。

## **69.2定制**

Consul Config可以使用以下属性来自定义：

**bootstrap.yml。**

spring:

cloud:

consul:

config:

enabled: true

prefix: configuration

defaultContext: apps

profileSeparator: '::'

* enabled将此值设置为“ false”将禁用Consul Config
* prefix设置配置值的基本文件夹
* defaultContext设置所有应用程序使用的文件夹名称
* profileSeparator设置分隔符的值，该分隔符用于在带有配置文件的属性源中分隔配置文件名称

## **69.3配置观察**

Consul Config Watch利用consul的功能来[监视键前缀](https://www.consul.io/docs/agent/watches.html" \l "keyprefix" \t "https://www.springcloud.cc/_top)。Config Watch进行阻塞Consul HTTP API调用，以确定当前应用程序的任何相关配置数据是否已更改。如果有新的配置数据，则会发布刷新事件。这等效于调用/refresh执行器端点。

要更改称为“配置监视”的频率，请更改spring.cloud.consul.config.watch.delay。默认值为1000，以毫秒为单位。延迟是上一次调用结束与下一次调用开始之间的时间量。

要禁用配置监视，请设置spring.cloud.consul.config.watch.enabled=false。

手表使用Spring TaskScheduler将通话安排到consul。默认情况下，它是ThreadPoolTaskScheduler，其poolSize为1。要更改TaskScheduler，请创建一个类型为TaskScheduler的bean，名称为ConsulConfigAutoConfiguration.CONFIG\_WATCH\_TASK\_SCHEDULER\_NAME常量。

## **69.4 YAML或带有配置的Properties**

与单个键/值对相反，以YAML或Properties格式存储属性的对象可能更方便。将spring.cloud.consul.config.format属性设置为YAML或PROPERTIES。例如使用YAML：

**bootstrap.yml。**

spring:

cloud:

consul:

config:

format: YAML

必须在consul中的相应data键中设置YAML。使用上面的默认值，键看起来像：

config/testApp,dev/data

config/testApp/data

config/application,dev/data

config/application/data

您可以将YAML文档存储在上面列出的任何键中。

您可以使用spring.cloud.consul.config.data-key更改数据密钥。

## **69.5 git2consul与配置**

git2consul是一个Consul社区项目，它从git存储库中将文件加载到Consul中的各个键中。默认情况下，键的名称是文件的名称。YAML和Properties文件分别受文件扩展名.yml和.properties的支持。将spring.cloud.consul.config.format属性设置为FILES。例如：

**bootstrap.yml。**

spring:

cloud:

consul:

config:

format: FILES

给定/config中的以下键，development配置文件和应用程序名称foo：

.gitignore

application.yml

bar.properties

foo-development.properties

foo-production.yml

foo.properties

master.ref

将创建以下属性源：

config/foo-development.properties

config/foo.properties

config/application.yml

每个密钥的值必须是格式正确的YAML或Properties文件。

## **69.6快速失败**

如果无法使用consul进行配置，则在某些情况下（例如本地开发或某些测试场景）可能会很方便而不会失败。在bootstrap.yml中设置spring.cloud.consul.config.failFast=false将使配置模块记录警告而不是引发异常。这将允许应用程序继续正常启动。

## **70. Consul重试**

如果您希望应用启动时consul代理有时不可用，则可以要求它在失败后继续尝试。您需要将spring-retry和spring-boot-starter-aop添加到类路径中。默认行为是重试6次，初始回退间隔为1000ms，随后的回退的指数乘数为1.1。您可以使用spring.cloud.consul.retry.\*配置属性来配置这些属性（和其他属性）。这适用于Spring Cloud Consul Config和Discovery注册。

|  |
| --- |
| IMG_505 |
| 要完全控制重试，请添加ID为“ consulRetryInterceptor”的类型为RetryOperationsInterceptor的@Bean。Spring重试有一个RetryInterceptorBuilder，可以轻松创建一个。 |

## **71. Spring Cloud Bus与Consul**

## **71.1如何激活**

要开始使用Consul Bus，请将启动器与组org.springframework.cloud和工件ID spring-cloud-starter-consul-bus一起使用。有关使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

有关可用的执行器端点以及如何发送自定义消息的信息，请参见[Spring Cloud Bus](https://cloud.spring.io/spring-cloud-bus/" \t "https://www.springcloud.cc/_top)文档。

## **72.带有Hystrix的断路器**

通过在项目pom.xml：spring-cloud-starter-hystrix中包含此启动程序，应用程序可以使用Spring Cloud Netflix项目提供的Hystrix断路器。Hystrix不依赖Netflix Discovery Client。@EnableHystrix批注应放在配置类（通常是主类）上。然后可以用@HystrixCommand注释方法，以使其受到断路器的保护。有关更多详细信息，请参见[文档](https://projects.spring.io/spring-cloud/spring-cloud.html" \l "_circuit_breaker_hystrix_clients" \t "https://www.springcloud.cc/_top)。

## **73. Hystrix通过Turbine和Consul进行指标汇总**

Turbine（由Spring Cloud Netflix项目提供）汇总了多个实例Hystrix指标流，因此仪表板可以显示汇总视图。Turbine使用DiscoveryClient接口查找相关实例。要将Turbine与Spring Cloud Consul结合使用，请以类似于以下示例的方式配置Turbine应用程序：

**pom.xml。**

<dependency>

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

<artifactId>spring-cloud-netflix-turbine</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-consul-discovery</artifactId>

</dependency>

请注意，Turbine依赖项不是启动器。涡轮启动器包括对Netflix Eureka的支持。

**application.yml。**

spring.application.name: turbine

applications: consulhystrixclient

turbine:

aggregator:

clusterConfig: ${applications}

appConfig: ${applications}

clusterConfig和appConfig部分必须匹配，因此将以逗号分隔的服务ID列表放入单独的配置属性中很有用。

**Turbine。java。**

@EnableTurbine

@SpringBootApplication

public class Turbine {

public static void main(String[] args) {

SpringApplication.run(DemoturbinecommonsApplication.class, args);

}

}

# **第十部分。Spring Cloud Zookeeper**

该项目通过自动配置并绑定到Spring环境和其他Spring编程模型习惯用法，为Spring Boot应用程序提供了Zookeeper集成。通过一些注释，您可以快速启用和配置应用程序内部的通用模式，并使用基于Zookeeper的组件构建大型分布式系统。提供的模式包括服务发现和配置。与Spring Cloud Netflix集成可提供智能路由（Zuul），客户端负载平衡（Ribbon）和断路器（Hystrix）。

## **74.安装Zookeeper**

有关如何安装Zookeeper的说明，请参阅[安装文档](https://zookeeper.apache.org/doc/current/zookeeperStarted.html" \t "https://www.springcloud.cc/_top)。

Spring Cloud Zookeeper在后台使用Apache Curator。尽管Zookeeper开发团队仍将Zookeeper 3.5.x视为“测试版”，但实际情况是，许多用户在生产中使用了它。但是，Zookeeper 3.4.x也用于生产中。在Apache Curator 4.0之前，两个版本的Apache Curator支持Zookeeper的两个版本。从Curator 4.0开始，两个Zookeeper版本都通过相同的Curator库支持。

如果要与版本3.4集成，则需要更改curator附带的Zookeeper依赖项，并因此更改spring-cloud-zookeeper。为此，只需排除该依赖性并添加如下所示的3.4.x版本。

**专家。**

<dependency>

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

<artifactId>spring-cloud-starter-zookeeper-all</artifactId>

<exclusions>

<exclusion>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

</exclusion>

</exclusions></dependency><dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.12</version>

<exclusions>

<exclusion>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</exclusion>

</exclusions></dependency>

**摇篮。**

compile('org.springframework.cloud:spring-cloud-starter-zookeeper-all') {

exclude group: 'org.apache.zookeeper', module: 'zookeeper'

}

compile('org.apache.zookeeper:zookeeper:3.4.12') {

exclude group: 'org.slf4j', module: 'slf4j-log4j12'

}

## **75.使用Zookeeper进行服务发现**

服务发现是基于微服务的体系结构的关键原则之一。尝试手动配置每个客户端或某种形式的约定可能很困难并且很脆弱。[Curator](https://curator.apache.org/" \t "https://www.springcloud.cc/_top)（用于Zookeeper的Java库）通过[Service Discovery Extension](https://curator.apache.org/curator-x-discovery/" \t "https://www.springcloud.cc/_top)提供服务发现。Spring Cloud Zookeeper使用此扩展名进行服务注册和发现。

## **75.1激活**

包括对org.springframework.cloud:spring-cloud-starter-zookeeper-discovery的依赖项将启用设置Spring Cloud Zookeeper发现的自动配置。

|  |
| --- |
| IMG_506 |
| 对于web功能，您仍然需要包含org.springframework.boot:spring-boot-starter-web。 |

|  |  |
| --- | --- |
| IMG_507 | **警告** |
| 当使用Zookeeper的3.4版本时，您需要按[此处](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-install" \o "74.安装Zookeeper)所述更改包含依赖项的方式。 |

## **75.2向Zookeeper注册**

客户端向Zookeeper注册时，它将提供有关其自身的元数据（例如主机和端口，ID和名称）。

以下示例显示了一个Zookeeper客户端：

@SpringBootApplication@RestController**public** **class** Application {

@RequestMapping("/")

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

|  |
| --- |
| IMG_508 |
| 前面的示例是普通的Spring Boot应用程序。 |

如果Zookeeper位于localhost:2181之外的其他位置，则配置必须提供服务器的位置，如以下示例所示：

**application.yml。**

spring:

cloud:

zookeeper:

connect-string: localhost:2181

|  |  |
| --- | --- |
| IMG_509 | **警告** |
| 如果使用[Spring Cloud Zookeeper Config](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-config" \o "80. Zookeeper的分布式配置)，则上一示例中显示的值必须位于bootstrap.yml中，而不是application.yml中。 |

默认服务名称，实例ID和端口（从Environment获取）分别为${spring.application.name}，Spring上下文ID和${server.port}。

在类路径上具有spring-cloud-starter-zookeeper-discovery可使该应用同时进入Zookeeper “ 服务 ”（即，它自己注册）和“ 客户端 ”（即，它可以查询Zookeeper以定位其他服务） 。

如果要禁用Zookeeper Discovery Client，可以将spring.cloud.zookeeper.discovery.enabled设置为false。

## **75.3使用DiscoveryClient**

Spring Cloud 使用逻辑服务名称而不是物理URL 支持 [Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-feign" \t "https://www.springcloud.cc/_top) （REST客户端生成器）和 [Spring RestTemplate](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-ribbon" \t "https://www.springcloud.cc/_top)。

您还可以使用org.springframework.cloud.client.discovery.DiscoveryClient，它为发现客户端提供了一个不特定于Netflix的简单API，如以下示例所示：

@Autowired**private** DiscoveryClient discoveryClient;

**public** String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

**if** (list != null && list.size() > 0 ) {

**return** list.get(0).getUri().toString();

}

**return** null;

}

## **76.将Spring Cloud Zookeeper与Spring Cloud Netflix组件一起使用**

Spring Cloud Netflix提供了有用的工具，无论您使用哪种DiscoveryClient实现方式，它们都可以工作。Feign，Turbine，Ribbon和Zuul与Spring Cloud Zookeeper一起使用。

## **76.1 Ribbon与Zookeeper**

Spring Cloud Zookeeper提供了Ribbon的ServerList的实现。使用spring-cloud-starter-zookeeper-discovery时，默认情况下Ribbon被自动配置为使用ZookeeperServerList。

## **77. Spring Cloud Zookeeper和服务注册中心**

Spring Cloud Zookeeper实现了ServiceRegistry接口，允许开发人员以编程方式注册任意服务。

ServiceInstanceRegistration类提供了一种builder()方法来创建Registration可以使用的Registration对象，如以下示例所示：

@Autowired**private** ZookeeperServiceRegistry serviceRegistry;

**public** **void** registerThings() {

ZookeeperRegistration registration = ServiceInstanceRegistration.builder()

.defaultUriSpec()

.address("anyUrl")

.port(10)

.name("/a/b/c/d/anotherservice")

.build();

**this**.serviceRegistry.register(registration);

}

## **77.1实例状态**

Netflix Eureka支持向服务器注册OUT\_OF\_SERVICE实例。这些实例不作为活动服务实例返回。这对于诸如蓝色/绿色部署之类的行为很有用。（请注意，Curator Service Discovery配方不支持此行为。）利用灵活的有效负载，Spring云Zookeeper通过更新某些特定的元数据，然后在过滤器中的该元数据上进行过滤，来实现OUT\_OF\_SERVICE。 Ribbon ZookeeperServerList。ZookeeperServerList过滤掉所有不等于UP的非空实例状态。如果实例状态字段为空，则为了向后兼容，它被视为UP。要更改实例的状态，请用OUT\_OF\_SERVICE和POST设置为ServiceRegistry实例状态执行器端点，如以下示例所示：

$ http POST http://localhost:8081/service-registry status=OUT\_OF\_SERVICE

|  |
| --- |
| IMG_510 |
| 前面的示例使用来自[https://httpie.org](https://httpie.org/" \t "https://www.springcloud.cc/_top)的http命令。 |

## **78. Zookeeper依赖项**

以下主题介绍如何使用Spring Cloud Zookeeper依赖项：

* [第78.1节“使用Zookeeper依赖项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-using" \o "78.1使用Zookeeper依赖关系)
* [第78.2节“激活Zookeeper依赖项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-activating" \o "78.2激活Zookeeper依赖关系)
* [第78.3节“设置Zookeeper依赖项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-setting-up" \o "78.3设置Zookeeper依赖关系)
* [第78.4节“配置Spring Cloud Zookeeper依赖项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-configuring" \o "78.4配置Spring Cloud Zookeeper依赖关系)

## **78.1使用Zookeeper依赖项**

Spring Cloud Zookeeper使您可以将应用程序的依赖项作为属性提供。作为依赖项，您可以了解在Zookeeper中注册的其他应用程序，并希望通过[Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-feign" \t "https://www.springcloud.cc/_top) （REST客户端构建器）和[Spring RestTemplate](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc" \l "spring-cloud-ribbon" \t "https://www.springcloud.cc/_top)调用它们 。

您还可以使用Zookeeper依赖关系观察程序功能来控制和监视依赖关系的状态。

## **78.2激活Zookeeper依赖项**

包括对org.springframework.cloud:spring-cloud-starter-zookeeper-discovery的依赖关系将启用自动配置，以建立Spring Cloud Zookeeper依赖关系。即使您在属性中提供了依赖关系，也可以关闭依赖关系。为此，请将spring.cloud.zookeeper.dependency.enabled属性设置为false（默认为true）。

## **78.3设置Zookeeper依赖项**

考虑下面的依赖关系表示示例：

**application.yml。**

spring.application.name: yourServiceName

spring.cloud.zookeeper:

dependencies:

newsletter:

path: /path/where/newsletter/has/registered/in/zookeeper

loadBalancerType: ROUND\_ROBIN

contentTypeTemplate: application/vnd.newsletter.$version+json

version: v1

headers:

header1:

- value1

header2:

- value2

required: false

stubs: org.springframework:foo:stubs

mailing:

path: /path/where/mailing/has/registered/in/zookeeper

loadBalancerType: ROUND\_ROBIN

contentTypeTemplate: application/vnd.mailing.$version+json

version: v1

required: true

接下来的几节将逐一遍历依赖项的每个部分。根属性名称为spring.cloud.zookeeper.dependencies。

### **78.3.1别名**

在root属性下，您必须将每个依赖项表示为别名。这是由于Ribbon的限制所致，它要求将应用程序ID放置在URL中。因此，您无法通过任何复杂的路径，例如/myApp/myRoute/name。别名是您使用的名称，而不是DiscoveryClient，Feign或RestTemplate的serviceId。

在前面的示例中，别名为newsletter和mailing。以下示例显示了别名为newsletter的Feign的用法：

@FeignClient("newsletter")**public** **interface** NewsletterService {

@RequestMapping(method = RequestMethod.GET, value = "/newsletter")

String getNewsletters();

}

### **78.3.2路径**

该路径由path YAML属性表示，并且是在Zookeeper下注册依赖项的路径。如上 [一节所述](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependencies-setting-up-aliases" \o "78.3.1别名)，Ribbon对URL进行操作。结果，该路径不符合其要求。这就是Spring Cloud Zookeeper将别名映射到正确路径的原因。

### **78.3.3负载均衡器类型**

负载均衡器类型由loadBalancerType YAML属性表示。

如果您知道在调用此特定依赖项时必须应用哪种负载平衡策略，则可以在YAML文件中提供它，并自动应用它。您可以选择以下负载平衡策略之一：

* STICKY：选择后，将始终调用该实例。
* 随机：随机选择一个实例。
* ROUND\_ROBIN：反复遍历实例。

### **78.3.4 Content-Type模板和版本**

Content-Type模板和版本由contentTypeTemplate和version YAML属性表示。

如果您在Content-Type标头中对API进行版本控制，则不想将此标头添加到每个请求中。另外，如果您要调用API的新版本，则不想在代码中漫游以提高API版本。因此，您可以为contentTypeTemplate提供特殊的$version占位符。该占位符将由version YAML属性的值填充。考虑以下contentTypeTemplate的示例：

application/vnd.newsletter.$version+json

进一步考虑以下version：

v1

contentTypeTemplate和版本的组合会为每个请求创建一个Content-Type标头，如下所示：

application/vnd.newsletter.v1+json

### **78.3.5默认标题**

默认标头由YAML中的headers映射表示。

有时，每次对依赖项的调用都需要设置一些默认头。要在代码中不要这样做，可以在YAML文件中进行设置，如以下示例headers部分所示：

headers:

Accept:

- text/html

- application/xhtml+xml

Cache-Control:

- no-cache

该headers部分将在您的HTTP请求中添加带有适当值列表的Accept和Cache-Control标头。

### **78.3.6所需的依赖关系**

所需的依存关系由YAML中的required属性表示。

如果在应用程序启动时需要建立依赖关系之一，则可以在YAML文件中设置required: true属性。

如果您的应用程序在引导期间无法本地化所需的依赖关系，则会引发异常，并且Spring上下文无法设置。换句话说，如果所需的依赖项未在Zookeeper中注册，则您的应用程序将无法启动。

您可以[在本文档后面的内容中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-dependency-watcher-presence-checker" \o "79.3使用状态检查器)详细了解Spring Cloud Zookeeper Presence Checker 。

### **78.3.7存根**

您可以提供一个冒号分隔的指向JAR的路径，其中包含依赖项的存根，如以下示例所示：

stubs: org.springframework:myApp:stubs

哪里：

* org.springframework是groupId。
* myApp是artifactId。
* stubs是分类器。（请注意，stubs是默认值。）

因为stubs是默认分类器，所以前面的示例等于以下示例：

stubs: org.springframework:myApp

## **78.4配置Spring Cloud Zookeeper依赖项**

您可以设置以下属性来启用或禁用部分Zookeeper依赖关系功能：

* spring.cloud.zookeeper.dependencies：如果不设置此属性，则不能使用Zookeeper依赖关系。
* spring.cloud.zookeeper.dependency.ribbon.enabled（默认情况下启用）：Ribbon需要显式全局配置或特定的依赖项配置。通过启用此属性，可以实现运行时负载平衡策略解析，并且可以使用Zookeeper依赖项的loadBalancerType部分。需要此属性的配置具有LoadBalancerClient的实现，该实现委托给下一个项目符号中介绍的ILoadBalancer。
* spring.cloud.zookeeper.dependency.ribbon.loadbalancer（默认情况下启用）：由于使用此属性，自定义ILoadBalancer知道传递给Ribbon的URI部分实际上可能是别名，必须将其解析为Zookeeper。没有此属性，您将无法在嵌套路径下注册应用程序。
* spring.cloud.zookeeper.dependency.headers.enabled（默认情况下启用）：此属性注册一个RibbonClient，该文件会自动将适当的标头和内容类型及其版本附加在Dependency配置中。没有此设置，这两个参数将不起作用。
* spring.cloud.zookeeper.dependency.resttemplate.enabled（默认情况下启用）：启用后，此属性会修改带有@LoadBalanced注释的RestTemplate的请求标头，以使其传递标头和内容类型以及在依赖项配置中设置的版本。没有此设置，这两个参数将不起作用。

## **79. Spring Cloud Zookeeper依赖性观察者**

Dependency Watcher机制使您可以将侦听器注册到您的依赖项。实际上，该功能是Observator模式的实现。当依赖项改变时，其状态（变为UP或DOWN）可以应用一些自定义逻辑。

## **79.1激活**

Spring Cloud Zookeeper需要启用依赖功能，才能使用Dependency Watcher机制。

## **79.2注册侦听器**

要注册侦听器，您必须实现一个名为org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener的接口并将其注册为bean。该接口为您提供了一种方法：

**void** stateChanged(String dependencyName, DependencyState newState);

如果要注册特定依赖项的侦听器，则dependencyName将是您具体实现的区分符。newState为您提供有关您的依存关系已更改为CONNECTED还是DISCONNECTED的信息。

## **79.3使用状态检查器**

依赖关系监视程序绑定的是称为状态检查器的功能。它使您可以在应用程序启动时提供自定义行为，以根据依赖项的状态做出反应。

抽象org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier类的默认实现是org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier，它的工作方式如下。

1. 如果依赖项标记为required，而不是Zookeeper，则在应用程序启动时，它将引发异常并关闭。
2. 如果依赖性不是required，则org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker记录在WARN级别缺少依赖性。

因为仅当没有DependencyPresenceOnStartupVerifier类型的bean时才注册DefaultDependencyPresenceOnStartupVerifier，所以可以覆盖此功能。

## **80.使用Zookeeper的分布式配置**

Zookeeper提供了一个 [分层的名称空间](https://zookeeper.apache.org/doc/current/zookeeperOver.html" \l "sc_dataModelNameSpace" \t "https://www.springcloud.cc/_top) ，该[名称空间](https://zookeeper.apache.org/doc/current/zookeeperOver.html" \l "sc_dataModelNameSpace" \t "https://www.springcloud.cc/_top)使客户端可以存储任意数据，例如配置数据。Spring Cloud Zookeeper Config是[Config Server和Client](https://github.com/spring-cloud/spring-cloud-config" \t "https://www.springcloud.cc/_top)的替代方法 。在特殊的“ 引导 ” 阶段将配置加载到Spring环境中。默认情况下，配置存储在/config名称空间中。根据应用程序名称和活动配置文件创建多个PropertySource实例，以模拟解析属性的Spring Cloud Config顺序。例如，名称为testApp且配置文件为dev的应用程序为其创建了以下属性源：

* config/testApp,dev
* config/testApp
* config/application,dev
* config/application

最具体的属性来源在顶部，最不具体的属性在底部。config/application名称空间中的Properties适用于所有使用zookeeper进行配置的应用程序。config/testApp名称空间中的Properties仅可用于名为testApp的服务的实例。

当前在启动应用程序时读取配置。向/refresh发送HTTP POST请求会导致重新加载配置。当前未实现监视配置名称空间（Zookeeper支持）。

## **80.1激活**

包括对org.springframework.cloud:spring-cloud-starter-zookeeper-config的依赖项将启用设置Spring Cloud Zookeeper Config的自动配置。

|  |  |
| --- | --- |
| IMG_511 | **警告** |
| 当使用Zookeeper的3.4版本时，您需要按[此处](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-zookeeper-install" \o "74.安装Zookeeper)所述更改包含依赖项的方式。 |

## **80.2定制**

Zookeeper Config可以通过设置以下属性来自定义：

**bootstrap.yml。**

spring:

cloud:

zookeeper:

config:

enabled: true

root: configuration

defaultContext: apps

profileSeparator: '::'

* enabled：将此值设置为false会禁用Zookeeper Config。
* root：设置配置值的基本名称空间。
* defaultContext：设置所有应用程序使用的名称。
* profileSeparator：设置分隔符的值，该分隔符用于在带有配置文件的属性源中分隔配置文件名称。

## **80.3访问控制列表（ACL）**

您可以通过调用CuratorFramework bean的addAuthInfo方法来添加Zookeeper ACL的身份验证信息。实现此目的的一种方法是提供自己的CuratorFramework bean，如以下示例所示：

@BoostrapConfiguration**public** **class** CustomCuratorFrameworkConfig {

@Bean

**public** CuratorFramework curatorFramework() {

CuratorFramework curator = **new** CuratorFramework();

curator.addAuthInfo("digest", "user:password".getBytes());

**return** curator;

}

}

请查阅 [ZookeeperAutoConfiguration类，](https://github.com/spring-cloud/spring-cloud-zookeeper/blob/master/spring-cloud-zookeeper-core/src/main/java/org/springframework/cloud/zookeeper/ZookeeperAutoConfiguration.java" \t "https://www.springcloud.cc/_top) 以了解CuratorFramework bean的默认配置。

另外，您可以从依赖现有CuratorFramework bean的类中添加凭据，如以下示例所示：

@BoostrapConfiguration**public** **class** DefaultCuratorFrameworkConfig {

**public** ZookeeperConfig(CuratorFramework curator) {

curator.addAuthInfo("digest", "user:password".getBytes());

}

}

bean的创建必须在升压阶段进行。您可以注册配置类以在此阶段运行，方法是使用@BootstrapConfiguration进行注释，并将它们包含在以逗号分隔的列表中，该列表设置为resources/META-INF/spring.factories文件中org.springframework.cloud.bootstrap.BootstrapConfiguration属性的值，如图所示在以下示例中：

**资源/META-INF/spring.factories。**

org.springframework.cloud.bootstrap.BootstrapConfiguration=\

my.project.CustomCuratorFrameworkConfig,\

my.project.DefaultCuratorFrameworkConfig

# **第十一部分。Spring Cloud Security**

Spring Cloud Security提供了一组原语，用于以最小的代价构建安全的应用程序和服务。可以在外部（或中央）进行大量配置的声明性模型，通常可以通过中央身份管理服务来实现大型的，相互协作的远程组件系统。在Cloud Foundry之类的服务平台中使用它也非常容易。在Spring Boot和Spring Security OAuth2的基础上，我们可以快速创建实现通用模式（例如单点登录，令牌中继和令牌交换）的系统。

|  |
| --- |
| IMG_512 |
| Spring Cloud是根据非限制性Apache 2.0许可证发行的。如果您想为文档的这一部分做出贡献或发现错误，请在[github](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/asciidoc" \t "https://www.springcloud.cc/_top)的项目中找到源代码和问题跟踪程序。 |

## **81.快速入门**

## **81.1 OAuth2单点登录**

这是一个具有HTTP Basic身份验证和单个用户帐户的Spring Cloud“ Hello World”应用程序：

**app.groovy。**

@Grab('spring-boot-starter-security')@Controller**class** Application {

@RequestMapping('/')

String home() {

'Hello World'

}

}

您可以使用spring run app.groovy运行它，并在日志中查看密码（用户名是“ user”）。到目前为止，这只是Spring Boot应用的默认设置。

这是带有OAuth2 SSO的Spring Cloud应用：

**app.groovy。**

@Controller@EnableOAuth2Sso**class** Application {

@RequestMapping('/')

String home() {

'Hello World'

}

}

指出不同？该应用程序实际上将与上一个应用程序完全相同，因为它尚不知道它是OAuth2凭证。

您可以很容易地在github中注册一个应用程序，因此，如果要在自己的域上使用生产应用程序，请尝试。如果您愿意在localhost：8080上进行测试，请在应用程序配置中设置以下属性：

**application.yml。**

security: oauth2: client: clientId: bd1c0a783ccdd1c9b9e4 clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1 accessTokenUri: https://github.com/login/oauth/access\_token userAuthorizationUri: https://github.com/login/oauth/authorize clientAuthenticationScheme: form resource: userInfoUri: https://api.github.com/user preferTokenInfo: **false**

运行上面的应用程序，它将重定向到github进行授权。如果您已经登录github，您甚至不会注意到它已通过身份验证。仅当您的应用程序在端口8080上运行时，这些凭据才有效。

要限制客户端获得访问令牌时要求的范围，可以设置security.oauth2.client.scope（逗号分隔或YAML中的数组）。默认情况下，作用域为空，并且由授权服务器决定默认值是什么，通常取决于它所拥有的客户端注册中的设置。

|  |
| --- |
| IMG_513 |
| 上面的示例都是Groovy脚本。如果要用Java（或Groovy）编写相同的代码，则需要在类路径中添加Spring Security OAuth2（例如，在[此处](https://github.com/spring-cloud-samples/sso" \t "https://www.springcloud.cc/_top)查看 [示例](https://github.com/spring-cloud-samples/sso" \t "https://www.springcloud.cc/_top)）。 |

## **81.2 OAuth2受保护的资源**

您想使用OAuth2令牌保护API资源吗？这是一个简单的示例（与上面的客户端配对）：

**app.groovy。**

@Grab('spring-cloud-starter-security')@RestController@EnableResourceServer**class** Application {

@RequestMapping('/')

def home() {

[message: 'Hello World']

}

}

和

**application.yml。**

security: oauth2: resource: userInfoUri: https://api.github.com/user preferTokenInfo: **false**

## **82.更多细节**

## **82.1单点登录**

|  |
| --- |
| IMG_514 |
| 所有OAuth2 SSO和资源服务器功能已在版本1.3中移至Spring Boot。您可以在[Spring Boot用户指南中](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \t "https://www.springcloud.cc/_top)找到文档 。 |

## **82.2令牌中继**

令牌中继是OAuth2使用者充当客户端并将传入令牌转发到传出资源请求的地方。使用者可以是纯客户端（如SSO应用程序）或资源服务器。

### **Spring Cloud网关中的客户端令牌中继82.2.1**

如果您的应用程序还具有 [Spring Cloud Gateway](https://cloud.spring.io/spring-cloud-static/current/single/spring-cloud.html" \l "_spring_cloud_gateway" \t "https://www.springcloud.cc/_top)嵌入式反向代理，则可以要求它向下游转发OAuth2访问令牌到它正在代理的服务。因此，可以像下面这样简单地增强上面的SSO应用程序：

**App.java。**

@Autowired**private** TokenRelayGatewayFilterFactory filterFactory;

@Bean**public** RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

**return** builder.routes()

.route("resource", r -> r.path("/resource")

.filters(f -> f.filter(filterFactory.apply()))

.uri("http://localhost:9000"))

.build();

}

或这个

**application.yaml。**

spring: cloud: gateway: routes: - id: resource uri: http://localhost:9000 predicates: - Path=/resource filters: - TokenRelay=

它将（除了登录用户并获取令牌之外）将身份验证令牌传递到服务下游（在这种情况下为/resource）。

要为Spring Cloud网关启用此功能，请添加以下依赖项

* org.springframework.boot:spring-boot-starter-oauth2-client
* org.springframework.cloud:spring-cloud-starter-security

它是如何工作的？该 [滤波器](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/gateway/TokenRelayGatewayFilterFactory.java" \t "https://www.springcloud.cc/_top) 提取用于下游请求从当前认证的用户的访问令牌，并把它在请求报头。

有关完整的工作示例，请参见此[项目](https://github.com/spring-cloud-samples/sample-gateway-oauth2login" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_515 |
| TokenRelayGatewayFilterFactory使用的ReactiveOAuth2AuthorizedClientService的默认实现使用内存中的数据存储。如果需要更强大的解决方案，则需要提供自己的实现ReactiveOAuth2AuthorizedClientService。 |

### **82.2.2客户端令牌中继**

如果您的应用是面向OAuth2客户端的用户（即已声明@EnableOAuth2Sso或@EnableOAuth2Client），则它的请求范围为Spring Boot中的OAuth2ClientContext。您可以从此上下文中创建自己的OAuth2RestTemplate，并自动装配OAuth2ProtectedResourceDetails，然后该上下文将始终向下游转发访问令牌，如果过期则自动刷新访问令牌。（这些是Spring安全和Spring Boot的功能。）

|  |
| --- |
| IMG_516 |
| 如果您使用client\_credentials令牌，则Spring Boot（1.4.1）不会自动创建OAuth2ProtectedResourceDetails。在这种情况下，您需要创建自己的ClientCredentialsResourceDetails并使用@ConfigurationProperties("security.oauth2.client")对其进行配置。 |

### **Zuul代理中的82.2.3客户令牌中继**

如果您的应用程序还具有 [Spring云Zuul](https://cloud.spring.io/spring-cloud.html" \l "netflix-zuul-reverse-proxy" \t "https://www.springcloud.cc/_top)嵌入式反向代理（使用@EnableZuulProxy），则可以要求它向下游转发OAuth2访问令牌到它正在代理的服务。因此，可以像下面这样简单地增强上面的SSO应用程序：

**app.groovy。**

@Controller@EnableOAuth2Sso@EnableZuulProxy**class** Application {

}

并且它将（除了登录用户并获取令牌之外）还将身份验证令牌传递到/proxy/\*服务的下游。如果这些服务是通过@EnableResourceServer实现的，则它们将在正确的标头中获得有效的令牌。

它是如何工作的？@EnableOAuth2Sso注释会插入spring-cloud-starter-security（您可以在传统应用中手动完成此操作），并依次触发ZuulFilter的一些自动配置，该激活本身是因为Zuul位于类路径（通过@EnableZuulProxy）。该 [滤波器](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/OAuth2TokenRelayFilter.java" \t "https://www.springcloud.cc/_top) 只提取用于下游请求从当前认证的用户的访问令牌，并把它在请求报头。

|  |
| --- |
| IMG_517 |
| Spring Boot不会自动创建refresh\_token所需的OAuth2RestOperations。在这种情况下，您需要创建自己的OAuth2RestOperations，以便OAuth2TokenRelayFilter可以根据需要刷新令牌。 |

### **82.2.4资源服务器令牌中继**

如果您的应用具有@EnableResourceServer，则您可能希望将传入令牌下游中继到其他服务。如果您使用RestTemplate与下游服务联系，那么这只是如何在正确的上下文中创建模板的问题。

如果您的服务使用UserInfoTokenServices对传入令牌进行身份验证（即它使用的是security.oauth2.user-info-uri配置），那么您可以使用自动连接的OAuth2ClientContext简单地创建一个OAuth2RestTemplate（它将由身份验证填充）在到达后端代码之前进行处理）。等效地（对于Spring Boot 1.4），您可以注入UserInfoRestTemplateFactory，并在您的配置中获取其OAuth2RestTemplate。例如：

**MyConfiguration.java。**

@Bean**public** OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {

**return** factory.getUserInfoRestTemplate();

}

然后，该其余模板将具有与身份验证过滤器使用的相同的OAuth2ClientContext（请求范围），因此您可以使用它来发送具有相同访问令牌的请求。

如果您的应用未使用UserInfoTokenServices，但仍是客户端（即它声明了@EnableOAuth2Client或@EnableOAuth2Sso），则使用Spring Security覆盖用户从{12创建的任何OAuth2RestOperations /} OAuth2Context也将转发令牌。默认情况下，此功能作为MVC处理程序拦截器实现，因此仅在Spring MVC中有效。如果您不使用MVC，则可以使用包装AccessTokenContextRelay的自定义过滤器或AOP拦截器来提供相同的功能。

这是一个基本示例，展示了如何使用在其他位置创建的自动连接的休息模板（“ foo.com”是接受与周围应用程序相同的令牌的资源服务器）：

**MyController.java。**

@Autowired**private** OAuth2RestOperations restTemplate;

@RequestMapping("/relay")**public** String relay() {

ResponseEntity<String> response =

restTemplate.getForEntity("https://foo.com/bar", String.**class**);

**return** "Success! (" + response.getBody() + ")";

}

如果您不希望转发令牌（这是一个有效的选择，因为您可能想扮演自己的角色，而不是发送令牌的客户端），那么您只需要创建自己的OAuth2Context自动装配默认值。

Feign客户端还将选择使用OAuth2ClientContext的拦截器（如果可用），因此它们还应在RestTemplate可以使用的任何地方进行令牌中继。

## **83.配置Zuul代理的下游身份验证**

您可以通过proxy.auth.\*设置来控制@EnableZuulProxy下游的授权行为。例：

**application.yml。**

proxy: auth: routes: customers: oauth2 stores: passthru recommendations: none

在此示例中，“客户”服务获得一个OAuth2令牌中继，“商店”服务获得一个传递（授权标头仅向下游传递），“推荐”服务将其授权标头删除。默认行为是在有令牌可用的情况下进行令牌中继，否则通过。

有关完整的详细信息，请参见 [ProxyAuthenticationProperties](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/ProxyAuthenticationProperties" \t "https://www.springcloud.cc/_top)。

# **第十二部分。Spring Cloud for Cloud Foundry**

使用Cloudfoundry的Spring Cloud，可以轻松在[Cloud Foundry](https://github.com/cloudfoundry" \t "https://www.springcloud.cc/_top)（平台即服务）中运行 [Spring Cloud](https://github.com/spring-cloud" \t "https://www.springcloud.cc/_top)应用 。Cloud Foundry具有“服务”的概念，这是您“绑定”到应用程序的中间软件，本质上为它提供了一个包含凭证的环境变量（例如，用于服务的位置和用户名）。

spring-cloud-cloudfoundry-commons模块配置基于Reactor的Cloud Foundry Java客户端v 3.0，并且可以独立使用。

spring-cloud-cloudfoundry-web项目为Cloud Foundry中的Web应用程序的某些增强功能提供了基本支持：自动绑定到单点登录服务，并可以选择启用粘性路由进行发现。

spring-cloud-cloudfoundry-discovery项目提供了Spring Cloud Commons DiscoveryClient的实现，因此您可以@EnableDiscoveryClient并以spring.cloud.cloudfoundry.discovery.[username,password]的身份提供凭据（如果您没有连接到Pivotal Web服务），然后您可以直接或通过LoadBalancerClient使用DiscoveryClient。[Pivotal Web服务](https://run.pivotal.io/" \t "https://www.springcloud.cc/_top)），然后您可以直接或通过LoadBalancerClient使用DiscoveryClient。

首次使用它时，发现客户端可能会变慢，原因是它必须从Cloud Foundry获取访问令牌。

## **84.发现**

这是一个具有Cloud Foundry发现功能的Spring Cloud应用：

**app.groovy。**

@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')@RestController@EnableDiscoveryClient**class** Application {

@Autowired

DiscoveryClient client

@RequestMapping('/')

String home() {

'Hello from ' + client.getLocalServiceInstance()

}

}

如果运行时没有任何服务绑定：

$ spring jar app.jar app.groovy

$ cf push -p app.jar

它将在首页中显示其应用名称。

DiscoveryClient可以根据进行身份验证的凭据列出一个空间中的所有应用程序，其中该空间默认为客户端正在其中运行的应用程序（如果有）。如果未配置org和space，则根据Cloud Foundry中用户的配置文件默认设置。

## **85.单点登录**

|  |
| --- |
| IMG_518 |
| 所有OAuth2 SSO和资源服务器功能已在版本1.3中移至Spring Boot。您可以在[Spring Boot用户指南中](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \t "https://www.springcloud.cc/_top)找到文档 。 |

该项目提供了从CloudFoundry服务凭据到Spring Boot功能的自动绑定。例如，如果您有一个名为“ sso”的CloudFoundry服务，其凭证包含“ client\_id”，“ client\_secret”和“ auth\_domain”，它将自动绑定到您通过{9启用的Spring OAuth2客户端/}（来自Spring Boot）。可以使用spring.oauth2.sso.serviceId来参数化服务的名称。

# **第十三部分。Spring Cloud Contract**

文档作者：Adam Dudczak，MathiasDüsterhöft，Marcin Grzejszczak，Dennis Kieselhorst，JakubKubryński，Karol Lassak，Olga Maciaszek-Sharma，MariuszSmykuła，Dave Syer，Jay Bryant

Greenwich SR5

## **86. Spring Cloud Contract**

在将新功能推到分布式系统中的新应用程序或服务时，您需要信心。该项目为Spring应用程序中的消费者驱动Contracts和服务模式提供支持（用于HTTP和基于消息的交互），涵盖了编写测试，将其作为资产发布以及声明合同的一系列选项。由生产者和消费者保存。

## **87. Spring Cloud Contract验证程序简介**

Spring Cloud Contract Verifier支持基于JVM的应用程序的消费者驱动合同（CDC）开发。它将TDD移至软件体系结构级别。

Spring Cloud Contract验证程序随附合同定义语言（CDL）。合同定义用于产生以下资源：

* 在客户端代码上进行集成测试（客户端测试）时，WireMock将使用JSON存根定义。测试代码仍然必须是手工编写的，并且测试数据由Spring Cloud Contract Verifier产生。
* 消息传递路由（如果您正在使用消息传递服务）。我们与Spring Integration，Spring Cloud Stream，Spring AMQP和Apache Camel集成。您还可以设置自己的集成。
* 验收测试（在JUnit 4，JUnit 5或Spock中）用于验证API的服务器端实现是否符合合同（服务器测试）。Spring Cloud Contract验证程序将生成完整的测试。

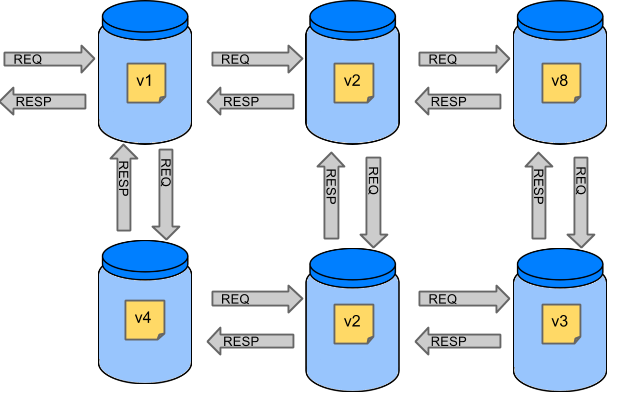
## **87.1历史**

在成为Spring Cloud Contract之前，此项目称为[Accurest](https://github.com/Codearte/accurest" \t "https://www.springcloud.cc/_top)。它是由 （[Codearte）的](https://github.com/Codearte" \t "https://www.springcloud.cc/_top)[Marcin Grzejszczak](https://twitter.com/mgrzejszczak" \t "https://www.springcloud.cc/_top)和[Jakub Kubrynski](https://twitter.com/jkubrynski" \t "https://www.springcloud.cc/_top)创建的。

0.1.0版本于2015年1月26日发布，并随着1.0.0版本于2016年2月29日发布而变得稳定。

## **87.2为什么要使用合同验证者？**

假设我们有一个包含多个微服务的系统：



### **87.2.1测试问题**

如果我们想在左上角测试该应用程序以确定它是否可以与其他服务通信，则可以执行以下两项操作之一：

* 部署所有微服务并执行端到端测试。
* 在单元/集成测试中模拟其他微服务。

两者都有优点，也有很多缺点。

**部署所有微服务并执行端到端测试**

好处：

* 模拟生产。
* 测试服务之间的真实通信。

缺点：

* 要测试一个微服务，我们必须部署6个微服务，几个数据库等。
* 测试运行的环境被锁定为单个测试套件（在此期间，其他任何人都无法运行测试）。
* 他们需要很长时间才能运行。
* 反馈在此过程中非常晚。
* 他们很难调试。

**在单元/集成测试中模拟其他微服务**

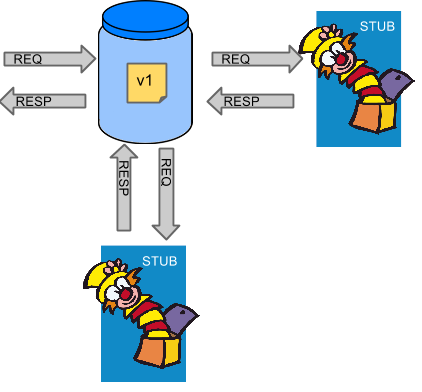
好处：

* 他们提供了非常快速的反馈。
* 他们没有基础架构要求。

缺点：

* 服务的实现者创建的存根可能与现实无关。
* 您可以通过测试并通过失败的生产。

为了解决上述问题，创建了带有Stub Runner的Spring Cloud Contract验证程序。主要思想是为您提供非常快速的反馈，而无需建立整个微服务世界。如果您使用存根，则仅需要应用程序直接使用的应用程序。



Spring Cloud Contract验证程序可确保您使用的存根是由您正在调用的服务创建的。另外，如果可以使用它们，则表示它们已经在生产者方面进行了测试。简而言之，您可以信任这些存根。

## **87.3目的**

Spring Cloud Contract Verifier Stub Runner的主要目的是：

* 为了确保WireMock / Messaging存根（在开发客户端时使用）完全执行实际的服务器端实现。
* 推广ATDD方法和微服务架构风格。
* 提供一种发布合同更改的方法，该更改在双方立即可见。
* 生成要在服务器端使用的样板测试代码。

|  |  |
| --- | --- |
| IMG_521 | **重要** |
| Spring Cloud Contract验证程序的目的不是开始在合同中编写业务功能。假设我们有一个欺诈检查的业务用例。如果某个用户可能出于100种不同的原因而成为欺诈行为，那么我们假设您将创建2个合同，一个用于肯定案件，一个用于否定案件。合同测试用于测试应用程序之间的合同，而不是模拟完整的行为。 |

## **87.4工作原理**

本节探讨Spring Cloud Contract带有Stub Runner的验证程序的工作原理。

### **87.4.1三秒游**

这个非常简短的导览使用Spring Cloud Contract来完成：

* [名为“在生产者端”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour-producer" \o "在生产者方面)
* [“消费者方面”一节](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour-consumer" \o "在消费者方面)

您可以[在这里](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour" \o "87.4.2三分钟游)找到更长的行程 。

#### **在生产者方面**

要开始使用Spring Cloud Contract，请将具有REST/消息合同（以Groovy DSL或YAML表示）的文件添加到由contractsDslDir属性设置的合同目录中。默认情况下为$rootDir/src/test/resources/contracts。

然后将Spring Cloud Contract Verifier依赖项和插件添加到您的构建文件中，如以下示例所示：

<dependency>

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

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope></dependency>

以下清单显示了如何添加插件，该插件应放在文件的build / plugins部分中：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions></plugin>

运行./mvnw clean install会自动生成测试，以验证应用程序是否符合添加的合同。默认情况下，测试在org.springframework.cloud.contract.verifier.tests.下生成。

由于尚不存在合同描述的功能的实现，因此测试失败。

要使它们通过，您必须添加处理HTTP请求或消息的正确实现。另外，您必须为自动生成的测试添加正确的基础测试类。该类由所有自动生成的测试扩展，并且应包含运行它们所需的所有设置（例如RestAssuredMockMvc控制器设置或消息传递测试设置）。

一旦实现和测试基类就位，测试就会通过，并且将应用程序和存根构件都构建并安装在本地Maven存储库中。现在可以合并更改，并且可以在在线存储库中发布应用程序和存根工件。

#### **在消费者方面**

Spring Cloud Contract Stub Runner可以用于集成测试中，以获取模拟实际服务的运行中WireMock实例或消息传递路由。

为此，请将依赖项添加到Spring Cloud Contract Stub Runner中，如以下示例所示：

<dependency>

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

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope></dependency>

您可以通过以下两种方式之一在Maven存储库中安装生产者端存根：

通过检出生产者端存储库并添加合同并通过运行以下命令来生成存根：

$ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

|  |
| --- |
| IMG_522 |
| 由于生产者方合同实施尚未到位，因此跳过了测试，因此自动生成的合同测试失败。 |

通过从远程存储库获取已经存在的生产者服务存根。为此，请将存根工件ID和工件存储库URL作为Spring Cloud Contract Stub Runner属性传递，如以下示例所示：

stubrunner: ids: 'com.example:http-server-dsl:+:stubs:8080' repositoryRoot: https://repo.spring.io/libs-snapshot

现在，您可以使用@AutoConfigureStubRunner注释测试类。在注释中，为Spring Cloud Contract Stub Runner提供group-id和artifact-id值，以为您运行协作者的存根，如以下示例所示：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment=WebEnvironment.NONE)@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},

stubsMode = StubRunnerProperties.StubsMode.LOCAL)**public** **class** LoanApplicationServiceTests {

|  |
| --- |
| IMG_523 |
| 从在线存储库下载存根时，请使用REMOTE stubsMode，而对于脱机工作，请使用LOCAL。 |

现在，在集成测试中，您可以接收预期由协作服务发出的HTTP响应或消息的存根版本。

### **87.4.2三分钟游**

此简短的导览使用Spring Cloud Contract来完成：

* [名为“在生产者端”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour-producer" \o "在生产者方面)
* [“消费者方面”一节](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-minute-tour-consumer" \o "在消费者方面)

您可以[在这里](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "spring-cloud-contract-verifier-intro-three-second-tour" \o "87.4.1三秒游)找到更简短的导览 。

#### **在生产者方面**

要开始使用Spring Cloud Contract，请将具有REST/消息传递合同（以Groovy DSL或YAML表示）的文件添加到由contractsDslDir属性设置的合同目录中。默认情况下，它是$rootDir/src/test/resources/contracts。

对于HTTP存根，合同定义了应针对给定请求返回的响应类型（考虑到HTTP方法，URL，标头，状态码等）。以下示例显示了Groovy DSL中的HTTP存根如何收缩：

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url '/fraudcheck'

body([

"client.id": $(regex('[0-9]{10}')),

loanAmount: 99999

])

headers {

contentType('application/json')

}

}

response {

status OK()

body([

fraudCheckStatus: "FRAUD",

"rejection.reason": "Amount too high"

])

headers {

contentType('application/json')

}

}

}

YAML中表示的同一合同应类似于以下示例：

request: method: PUT url: /fraudcheck body: "client.id": 1234567890 loanAmount: 99999 headers: Content-Type: application/json matchers: body: - path: $.['client.id'**]** type: by\_regex value: "[0-9]{10}"response: status: 200 body: fraudCheckStatus: "FRAUD" "rejection.reason": "Amount too high" headers: Content-Type: application/json;charset=UTF-8

对于消息传递，可以定义：

* 可以定义输入和输出消息（考虑发送消息的位置和位置，消息正文和标头）。
* 收到消息后应调用的方法。
* 调用时应触发消息的方法。

以下示例显示了以Groovy DSL表示的骆驼消息传递协定：

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:delete')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

assertThat('bookWasDeleted()')

}

}

以下示例显示了用YAML表示的同一合同：

label: some\_label

input:

messageFrom: jms:delete

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

assertThat: bookWasDeleted()

然后，您可以将Spring Cloud Contract Verifier依赖项和插件添加到您的构建文件中，如以下示例所示：

<dependency>

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

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope></dependency>

以下清单显示了如何添加插件，该插件应放在文件的build / plugins部分中：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions></plugin>

运行./mvnw clean install会自动生成测试，以验证应用程序是否符合添加的合同。默认情况下，生成的测试在org.springframework.cloud.contract.verifier.tests.下。

以下示例显示了自动生成的HTTP合同测试示例：

@Test**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

前面的示例使用Spring的MockMvc运行测试。这是HTTP合同的默认测试模式。但是，也可以使用JAX-RS客户端和显式HTTP调用。（为此，请将插件的testMode属性分别更改为JAX-RS或EXPLICIT。）

从2.1.0版开始，也可以使用RestAssuredWebTestClient`with Spring’s reactive `WebTestClient在后台运行。在使用基于Web-Flux的响应式应用程序时，特别推荐使用此方法。为了使用WebTestClient，请将testMode设置为WEBTESTCLIENT。

这是在WEBTESTCLIENT测试模式下生成的测试的示例：

[source,java,indent=0]

@Test

public void validate\_shouldRejectABeerIfTooYoung() throws Exception {

// given:

WebTestClientRequestSpecification request = given()

.header("Content-Type", "application/json")

.body("{\"age\":10}");

// when:

WebTestClientResponse response = given().spec(request)

.post("/check");

// then:

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/json.\*");

// and:

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['status']").isEqualTo("NOT\_OK");

}

除了默认的JUnit 4，您可以通过将插件的testFramework属性设置为JUNIT5或Spock来使用JUnit 5或Spock测试。

|  |
| --- |
| IMG_524 |
| 现在，您还可以基于合同生成WireMock方案，方法是在合同文件名的开头添加订单号，后跟下划线。 |

以下示例显示了在Spock中为消息存根合约自动生成的测试：

[source,groovy,indent=0]

given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

\'\'\'{"bookName":"foo"}\'\'\',

['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:

noExceptionThrown()

bookWasDeleted()

由于尚不存在合同描述的功能的实现，因此测试失败。

要使它们通过，您必须添加处理HTTP请求或消息的正确实现。另外，您必须为自动生成的测试添加正确的基础测试类。该类由所有自动生成的测试扩展，并且应包含运行它们所需的所有设置（例如，RestAssuredMockMvc控制器设置或消息传递测试设置）。

一旦实现和测试基类就位，测试就会通过，并且将应用程序和存根构件都构建并安装在本地Maven存储库中。有关将存根jar安装到本地存储库的信息显示在日志中，如以下示例所示：

[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar

[INFO]

[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---

[INFO]

[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar

[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

现在，您可以合并更改，并在在线存储库中发布应用程序和存根工件。

**Docker项目**

为了在使用非JVM技术创建应用程序时启用合同，已经创建了springcloud/spring-cloud-contract Docker映像。它包含一个项目，该项目会自动为HTTP合同生成测试并以EXPLICIT测试模式执行它们。然后，如果测试通过，它将生成Wiremock存根并将其发布到工件管理器（可选）。为了使用该映像，您可以将合同挂载到/contracts目录中并设置一些环境变量。

#### **在消费者方面**

Spring Cloud Contract Stub Runner可用于集成测试中，以获取模拟实际服务的正在运行的WireMock实例或消息传递路由。

首先，将依赖项添加到Spring Cloud Contract Stub Runner：

<dependency>

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

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope></dependency>

您可以通过以下两种方式之一在Maven存储库中安装生产者端存根：

通过检出生产者端存储库并添加合同并通过运行以下命令来生成存根：

$ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

|  |
| --- |
| IMG_525 |
| 由于生产者方合同实施尚未到位，因此跳过了测试，因此自动生成的合同测试失败。 |

从远程存储库获取已经存在的生产者服务存根。为此，请将存根工件标识和工件存储库UR1作为Spring Cloud Contract Stub Runner属性传递，如以下示例所示：

stubrunner: ids: 'com.example:http-server-dsl:+:stubs:8080' repositoryRoot: https://repo.spring.io/libs-snapshot

现在，您可以使用@AutoConfigureStubRunner注释测试类。在注释中，为Spring Cloud Contract Stub Runner提供group-id和artifact-id以便为您运行协作者的存根，如以下示例所示：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment=WebEnvironment.NONE)@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},

stubsMode = StubRunnerProperties.StubsMode.LOCAL)**public** **class** LoanApplicationServiceTests {

|  |
| --- |
| IMG_526 |
| 从在线存储库下载存根时，请使用REMOTE stubsMode，而对于脱机工作，请使用LOCAL。 |

在集成测试中，您可以接收HTTP响应的残存版本或预期由协作服务发出的消息。您可以在构建日志中看到类似于以下内容的条目：

2016-07-19 14:22:25.403 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to resolve the latest version2016-07-19 14:22:25.438 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT2016-07-19 14:22:25.439 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []2016-07-19 14:22:25.451 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar2016-07-19 14:22:25.465 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI: file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar]2016-07-19 14:22:25.475 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to [/var/folders/0p/xwq47sq106x1\_g3dtv6qfm940000gq/T/contracts100276532569594265]2016-07-19 14:22:27.737 INFO 41050 --- [ main] o.s.c.c.stubrunner.StubRunnerExecutor : All stubs are now running RunningStubs [namesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

### **87.4.3定义合同**

作为服务的使用者，我们需要定义要实现的目标。我们需要制定我们的期望。这就是为什么我们签订合同的原因。

假设您要发送一个包含客户公司ID以及它要向我们借款的金额的请求。您还希望通过PUT方法将其发送到/ fraudcheck URL。

**Groovy DSL。**

*/\**

*\* Copyright 2013-2019 the original author or authors.*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* https://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* limitations under the License.*

*\*/*

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request { *// (1)*

method 'PUT' *// (2)*

url '/fraudcheck' *// (3)*

body([ *// (4)*

"client.id": $(regex('[0-9]{10}')),

loanAmount : 99999

])

headers { *// (5)*

contentType('application/json')

}

}

response { *// (6)*

status OK() *// (7)*

body([ *// (8)*

fraudCheckStatus : "FRAUD",

"rejection.reason": "Amount too high"

])

headers { *// (9)*

contentType('application/json')

}

}

}

*/\**

*From the Consumer perspective, when shooting a request in the integration test:*

*(1) - If the consumer sends a request*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the response will be sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` equal to `application/json`*

*From the Producer perspective, in the autogenerated producer-side test:*

*(1) - A request will be sent to the producer*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the test will assert if the response has been sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` matching `application/json.\*`*

*\*/*

**YAML。**

request: # (1)

method: PUT # (2)

url: /fraudcheck # (3)

body: # (4)

"client.id": 1234567890

loanAmount: 99999

headers: # (5)

Content-Type: application/json

matchers:

body:

- path: $.['client.id'] # (6)

type: by\_regex

value: "[0-9]{10}"

response: # (7)

status: 200 # (8)

body: # (9)

fraudCheckStatus: "FRAUD"

"rejection.reason": "Amount too high"

headers: # (10)

Content-Type: application/json;charset=UTF-8

#From the Consumer perspective, when shooting a request in the integration test:

#

#(1) - If the consumer sends a request

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`

#(7) - then the response will be sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json`

#

#From the Producer perspective, in the autogenerated producer-side test:

#

#(1) - A request will be sent to the producer

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id` `1234567890`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(7) - then the test will assert if the response has been sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`

### **87.4.4客户端**

Spring Cloud Contract生成存根，可在客户端测试期间使用。您将获得一个正在运行的WireMock实例/消息传递路由，以模拟该服务。您想使用适当的存根定义来提供该实例。

在某个时间点，您需要向欺诈检测服务发送请求。

ResponseEntity<FraudServiceResponse> response = restTemplate.exchange(

"http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,

**new** HttpEntity<>(request, httpHeaders), FraudServiceResponse.**class**);

用@AutoConfigureStubRunner注释测试类。在批注中，提供Stub Runner的组ID和工件ID，以下载协作者的存根。

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.NONE)@AutoConfigureStubRunner(ids = {

"com.example:http-server-dsl:+:stubs:6565" }, stubsMode = StubRunnerProperties.StubsMode.LOCAL)**public** **class** LoanApplicationServiceTests {

之后，在测试期间，Spring Cloud Contract在Maven存储库中自动找到存根（模拟真实服务），并将其暴露在已配置（或随机）的端口上。

### **87.4.5服务器端**

由于您正在开发存根，因此需要确保它实际上类似于您的具体实现。您不能存在存根以一种方式运行而应用程序以不同方式运行的情况，尤其是在生产环境中。

为了确保您的应用程序符合您在存根中定义的方式，将从提供的存根中生成测试。

自动生成的测试或多或少看起来像这样：

@Test**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

## **87.5消费者驱动Contracts（CDC）循序渐进指南**

考虑欺诈检测和贷款发行过程的示例。业务场景是这样的，我们希望向人们发放贷款，但又不想他们从我们那里窃取资金。我们系统的当前实施情况向所有人提供贷款。

假设Loan Issuance是Fraud Detection服务器的客户端。在当前的Sprint中，我们必须开发一个新功能：如果客户想要借太多钱，那么我们会将客户标记为欺诈。

技术说明-欺诈检测的artifact-id为http-server，而贷款发行的人工ID为http-client，两者的group-id为com.example。

社交评论-客户和服务器开发团队都需要在整个过程中直接沟通并讨论更改。CDC完全是关于沟通的。

在[服务器端代码可以在这里](https://github.com/spring-cloud/spring-cloud-contract/tree/2.1.x/samples/standalone/dsl/http-server" \t "https://www.springcloud.cc/_top)和[这里的客户端代码](https://github.com/spring-cloud/spring-cloud-contract/tree/2.1.x/samples/standalone/dsl/http-client" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_527 |
| 在这种情况下，生产者拥有合同。实际上，所有合同都在生产者的资料库中。 |

### **87.5.1技术说明**

如果使用**SNAPSHOT** / **Milestone** / **Release Candidate**版本，请在您的版本中添加以下部分：

**Maven.**

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository></repositories><pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository></pluginRepositories>

**Gradle.**

repositories {

mavenCentral()

mavenLocal()

maven { url "https://repo.spring.io/snapshot" }

maven { url "https://repo.spring.io/milestone" }

maven { url "https://repo.spring.io/release" }

}

### **87.5.2消费者方（发放贷款）**

作为贷款发行服务的开发人员（欺诈检测服务器的使用者），您可以执行以下步骤：

1. 通过为您的功能编写测试来开始进行TDD。
2. 编写缺少的实现。
3. 在本地克隆欺诈检测服务存储库。
4. 在欺诈检测服务的仓库中本地定义合同。
5. 添加Spring Cloud Contract验证程序插件。
6. 运行集成测试。
7. 提出拉取请求。
8. 创建一个初始实现。
9. 接管请求请求。
10. 编写缺少的实现。
11. 部署您的应用程序。
12. 在线工作。

**通过为您的功能编写测试来开始进行TDD。**

@Test**public** **void** shouldBeRejectedDueToAbnormalLoanAmount() {

*// given:*

LoanApplication application = **new** LoanApplication(**new** Client("1234567890"),

99999);

*// when:*

LoanApplicationResult loanApplication = service.loanApplication(application);

*// then:*

assertThat(loanApplication.getLoanApplicationStatus())

.isEqualTo(LoanApplicationStatus.LOAN\_APPLICATION\_REJECTED);

assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");

}

假设您已经编写了新功能的测试。如果收到大量贷款申请，则系统应拒绝该贷款申请并提供一些说明。

**编写缺少的实现。**

在某个时间点，您需要向欺诈检测服务发送请求。假设您需要发送包含客户ID和客户希望借入的金额的请求。您想通过PUT方法将其发送到/fraudcheck网址。

ResponseEntity<FraudServiceResponse> response = restTemplate.exchange(

"http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,

**new** HttpEntity<>(request, httpHeaders), FraudServiceResponse.**class**);

为简单起见，欺诈检测服务的端口设置为8080，应用程序在8090上运行。

如果此时开始测试，则会中断测试，因为当前没有服务在端口8080上运行。

**在本地克隆欺诈检测服务存储库。**

您可以从服务器端合同开始。为此，您必须首先克隆它。

$ git clone https://your-git-server.com/server-side.git local-http-server-repo

**在欺诈检测服务的仓库中本地定义合同。**

作为消费者，您需要定义要实现的目标。您需要制定自己的期望。为此，请编写以下合同：

|  |  |
| --- | --- |
| IMG_528 | **重要** |
| 将合同放在src/test/resources/contracts/fraud文件夹下。fraud文件夹很重要，因为生产者的测试基类名称引用了该文件夹。 |

**Groovy DSL。**

*/\**

*\* Copyright 2013-2019 the original author or authors.*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* https://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* limitations under the License.*

*\*/*

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request { *// (1)*

method 'PUT' *// (2)*

url '/fraudcheck' *// (3)*

body([ *// (4)*

"client.id": $(regex('[0-9]{10}')),

loanAmount : 99999

])

headers { *// (5)*

contentType('application/json')

}

}

response { *// (6)*

status OK() *// (7)*

body([ *// (8)*

fraudCheckStatus : "FRAUD",

"rejection.reason": "Amount too high"

])

headers { *// (9)*

contentType('application/json')

}

}

}

*/\**

*From the Consumer perspective, when shooting a request in the integration test:*

*(1) - If the consumer sends a request*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the response will be sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` equal to `application/json`*

*From the Producer perspective, in the autogenerated producer-side test:*

*(1) - A request will be sent to the producer*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the test will assert if the response has been sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` matching `application/json.\*`*

*\*/*

**YAML。**

request: # (1)

method: PUT # (2)

url: /fraudcheck # (3)

body: # (4)

"client.id": 1234567890

loanAmount: 99999

headers: # (5)

Content-Type: application/json

matchers:

body:

- path: $.['client.id'] # (6)

type: by\_regex

value: "[0-9]{10}"

response: # (7)

status: 200 # (8)

body: # (9)

fraudCheckStatus: "FRAUD"

"rejection.reason": "Amount too high"

headers: # (10)

Content-Type: application/json;charset=UTF-8

#From the Consumer perspective, when shooting a request in the integration test:

#

#(1) - If the consumer sends a request

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`

#(7) - then the response will be sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json`

#

#From the Producer perspective, in the autogenerated producer-side test:

#

#(1) - A request will be sent to the producer

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id` `1234567890`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(7) - then the test will assert if the response has been sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`

YML合同很简单。但是，当您查看使用静态类型的Groovy DSL编写的合同时-您可能会怀疑value(client(…​), server(…​))部分是什么。通过使用此表示法，Spring Cloud Contract使您可以定义JSON块，URL等动态的部分。如果是标识符或时间戳，则无需对值进行硬编码。您要允许一些不同的值范围。要启用值范围，可以为使用者方设置与这些值匹配的正则表达式。您可以通过地图符号或带插值的字符串来提供主体。有关更多信息[，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl" \o "94.合同DSL)请参见[第94章](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl" \o "94. Contract DSL)*[Contract DSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl" \o "94. Contract DSL)*部分。我们强烈建议您使用地图符号！

|  |
| --- |
| IMG_529 |
| 您必须了解地图符号才能设置合同。请阅读[有关JSON](https://groovy-lang.org/json.html" \t "https://www.springcloud.cc/_top)的 [Groovy文档](https://groovy-lang.org/json.html" \t "https://www.springcloud.cc/_top)。 |

前面显示的合同是双方之间的协议，其中：

如果HTTP请求与所有

* + /fraudcheck端点上的PUT方法，
  + 一个具有client.id且与正则表达式[0-9]{10}和loanAmount等于99999匹配的JSON正文，
  + 和值为application/vnd.fraud.v1+json的Content-Type标头，

然后将HTTP响应发送给使用者

* + 状态为200，
  + 包含JSON正文，其fraudCheckStatus字段包含值FRAUD，而rejectionReason字段包含值Amount too high，
  + 还有一个Content-Type标头，其值为application/vnd.fraud.v1+json。

一旦准备好在集成测试中实际检查API，就需要在本地安装存根。

**添加Spring Cloud Contract验证程序插件。**

我们可以添加Maven或Gradle插件。在此示例中，您将了解如何添加Maven。首先，添加Spring Cloud Contract BOM。

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-release.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

接下来，添加Spring Cloud Contract Verifier Maven插件

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

<convertToYaml>true</convertToYaml>

</configuration></plugin>

自从添加了插件以来，您将获得Spring Cloud Contract Verifier功能，这些功能来自提供的合同：

* 生成并运行测试
* 制作并安装存根

您不想生成测试，因为作为消费者，您只想玩存根。您需要跳过测试的生成和执行。执行时：

$ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

在日志中，您会看到以下内容：

[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar

[INFO]

[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---

[INFO]

[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar

[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

以下行非常重要：

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

它确认http-server的存根已安装在本地存储库中。

**运行集成测试。**

为了从自动存根下载的Spring Cloud Contract Stub Runner功能中受益，您必须在用户端项目（Loan Application service）中执行以下操作：

添加Spring Cloud Contract BOM：

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-release-train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

将依赖项添加到Spring Cloud Contract Stub Runner：

<dependency>

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

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope></dependency>

用@AutoConfigureStubRunner注释测试类。在注释中，为Stub Runner提供group-id和artifact-id，以下载合作者的存根。（可选步骤）由于您是与离线协作者一起玩，因此您还可以提供离线工作切换（StubRunnerProperties.StubsMode.LOCAL）。

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.NONE)@AutoConfigureStubRunner(ids = {

"com.example:http-server-dsl:+:stubs:6565" }, stubsMode = StubRunnerProperties.StubsMode.LOCAL)**public** **class** LoanApplicationServiceTests {

现在，当您运行测试时，您将看到类似以下的内容：

2016-07-19 14:22:25.403 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to resolve the latest version2016-07-19 14:22:25.438 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT2016-07-19 14:22:25.439 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []2016-07-19 14:22:25.451 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar2016-07-19 14:22:25.465 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI: file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar]2016-07-19 14:22:25.475 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to [/var/folders/0p/xwq47sq106x1\_g3dtv6qfm940000gq/T/contracts100276532569594265]2016-07-19 14:22:27.737 INFO 41050 --- [ main] o.s.c.c.stubrunner.StubRunnerExecutor : All stubs are now running RunningStubs [namesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

此输出意味着Stub Runner找到了您的存根，并为您的应用启动了服务器，其组ID为com.example，工件ID为​​http-server，存根的版本为0.0.1-SNAPSHOT，且分类器为stubs端口8080。

**提出拉取请求。**

到目前为止，您所做的是一个迭代过程。您可以试用合同，将其安装在本地，然后在用户端工作，直到合同按您的意愿运行。

对结果满意并通过测试后，将拉取请求发布到服务器端。目前，消费者方面的工作已经完成。

### **87.5.3生产者端（欺诈检测服务器）**

作为欺诈检测服务器（贷款发放服务的服务器）的开发人员：

**创建一个初始实现。**

提醒一下，您可以在此处看到初始实现：

@RequestMapping(value = "/fraudcheck", method = PUT)**public** FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {**return** **new** FraudCheckResult(FraudCheckStatus.OK, NO\_REASON);

}

**接管请求请求。**

$ git checkout -b contract-change-pr master

$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr

您必须添加自动生成的测试所需的依赖项：

<dependency>

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

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope></dependency>

在Maven插件的配置中，传递packageWithBaseClasses属性

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

<convertToYaml>true</convertToYaml>

</configuration></plugin>

|  |  |
| --- | --- |
| IMG_530 | **重要** |
| 本示例通过设置packageWithBaseClasses属性使用“基于约定”的命名。这样做意味着最后两个软件包组合在一起以成为基础测试类的名称。在我们的案例中，合同位于src/test/resources/contracts/fraud下。由于从contracts文件夹开始没有两个软件包，因此仅选择一个，应该为fraud。添加后缀Base，并大写fraud。这将为您提供FraudBase测试类名称。 |

所有生成的测试都扩展了该类。在那边，您可以设置Spring上下文或任何必需的内容。在这种情况下，请使用[Rest Assured MVC](https://github.com/rest-assured/rest-assured" \t "https://www.springcloud.cc/_top)启动服务器端FraudDetectionController。

*/\**

*\* Copyright 2013-2019 the original author or authors.*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* https://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* limitations under the License.*

*\*/*

**package** com.example.fraud;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;**import** org.junit.Before;

**public** **class** FraudBase {

@Before

**public** **void** setup() {

RestAssuredMockMvc.standaloneSetup(**new** FraudDetectionController(),

**new** FraudStatsController(stubbedStatsProvider()));

}

**private** StatsProvider stubbedStatsProvider() {

**return** fraudType -> {

**switch** (fraudType) {

**case** DRUNKS:

**return** 100;

**case** ALL:

**return** 200;

}

**return** 0;

};

}

**public** **void** assertThatRejectionReasonIsNull(Object rejectionReason) {

assert rejectionReason == null;

}

}

现在，如果您运行./mvnw clean install，则会得到以下内容：

Results :

Tests in error:

ContractVerifierTest.validate\_shouldMarkClientAsFraud:32 » IllegalState Parsed...

发生此错误的原因是您有一个新合同，从中生成了一个测试，但由于未实现该功能而失败了。自动生成的测试如下所示：

@Test**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

如果您使用了Groovy DSL，则可以看到value(consumer(…​), producer(…​))块中存在的合同的所有producer()部分都已注入到测试中。如果使用YAML，则同样适用于response的matchers部分。

请注意，在生产者方面，您也在执行TDD。期望以测试的形式表达。此测试使用合同中定义的URL，标头和正文向我们自己的应用程序发送请求。它还期望响应中精确定义的值。换句话说，您拥有red，green和refactor的red部分。现在是将red转换为green的时候了。

**编写缺少的实现。**

因为您知道预期的输入和预期的输出，所以可以编写缺少的实现：

@RequestMapping(value = "/fraudcheck", method = PUT)**public** FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {**if** (amountGreaterThanThreshold(fraudCheck)) {

**return** **new** FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT\_TOO\_HIGH);

}**return** **new** FraudCheckResult(FraudCheckStatus.OK, NO\_REASON);

}

再次执行./mvnw clean install时，测试通过。由于Spring Cloud Contract Verifier插件将测试添加到generated-test-sources中，因此您实际上可以从IDE中运行这些测试。

**部署您的应用程序。**

完成工作后，即可部署更改。首先，合并分支：

$ git checkout master

$ git merge --no-ff contract-change-pr

$ git push origin master

您的CI可能会运行类似./mvnw clean deploy之类的东西，它将同时发布应用程序和存根工件。

### **87.5.4消费者方（贷款发行）最后一步**

作为贷款发行服务的开发人员（欺诈检测服务器的使用者）：

**合并分支以掌握。**

$ git checkout master

$ git merge --no-ff contract-change-pr

**在线工作。**

现在，您可以禁用Spring Cloud Contract Stub Runner的脱机工作，并指定包含存根的存储库所在的位置。此时，服务器端的存根会自动从Nexus / Artifactory下载。您可以将stubsMode的值设置为REMOTE。以下代码显示了通过更改属性来实现相同目的的示例。

stubrunner: ids: 'com.example:http-server-dsl:+:stubs:8080' repositoryRoot: https://repo.spring.io/libs-snapshot

而已！

## **87.6依赖项**

添加依赖项的最佳方法是使用适当的starter依赖项。

对于stub-runner，请使用spring-cloud-starter-stub-runner。使用插件时，添加spring-cloud-starter-contract-verifier。

## **87.7其他链接**

以下是与Spring Cloud Contract验证程序和Stub Runner有关的一些资源。请注意，有些可能已经过时，因为Spring Cloud Contract Verifier项目正在不断开发中。

### **87.7.1 Spring Cloud Contract视频**

您可以从华沙水罐观看有关Spring Cloud Contract的视频：

### **87.7.2阅读**

* [Marcin Grzejszczak关于Accurest的演讲的幻灯片](https://www.slideshare.net/MarcinGrzejszczak/stick-to-the-rules-consumer-driven-contracts-201507-confitura" \t "https://www.springcloud.cc/_top)
* [Marcin Grzejszczak博客中与Accurest相关的文章](https://toomuchcoding.com/blog/categories/accurest/" \t "https://www.springcloud.cc/_top)
* [Spring Cloud Contract Marcin Grzejszczak博客中的相关文章](https://toomuchcoding.com/blog/categories/spring-cloud-contract/" \t "https://www.springcloud.cc/_top)
* [Groovy关于JSON的文档](https://groovy-lang.org/json.html" \t "https://www.springcloud.cc/_top)

## **87.8个示例**

您可以在[示例中](https://github.com/spring-cloud-samples/spring-cloud-contract-samples" \t "https://www.springcloud.cc/_top)找到一些 [示例](https://github.com/spring-cloud-samples/spring-cloud-contract-samples" \t "https://www.springcloud.cc/_top)。

## **88. Spring Cloud Contract常见问题**

## **88.1为什么使用Spring Cloud Contract验证程序而不使用X？**

目前，Spring Cloud Contract是基于JVM的工具。因此，当您已经为JVM创建软件时，它可能是您的首选。该项目具有许多非常有趣的功能，但尤其是其中许多确实使Spring Cloud Contract验证程序在消费者驱动合同（CDC）工具的“市场”上脱颖而出。最有趣的是：

* 通过消息进行CDC的可能性
* 清晰易用的静态类型DSL
* 可以将您当前的JSON文件复制粘贴到合同中，并仅编辑其元素
* 根据定义的合同自动生成测试
* Stub Runner功能-存根会在运行时从Nexus / Artifactory自动下载
* Spring Cloud集成-集成测试不需要发现服务
* Spring Cloud Contract与Pact开箱即用地集成在一起，并提供简单的钩子来扩展其功能
* 通过Docker添加对使用的任何语言和框架的支持

## **88.2我不想在Groovy中写合同！**

没问题。您可以在YAML中写合同！

## **88.3这个值是什么（consumer（），producer（））？**

与存根相关的最大挑战之一是它们的可重用性。只有将它们广泛使用，它们才能达到目的。通常使困难的是请求/响应元素的硬编码值。例如日期或ID。想象以下JSON请求

**{**

"time" : "2016-10-10 20:10:15"**,**

"id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a"**,**

"body" : "foo"**}**

和JSON响应

**{**

"time" : "2016-10-10 21:10:15"**,**

"id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051"**,**

"body" : "bar"**}**

想象一下通过更改系统中的时钟或提供数据提供者的存根实现来设置time字段的适当值（让我们假定此内容是由数据库生成）所需的痛苦。与名为id的字段相同。您将创建UUID生成器的存根实现吗？毫无意义...

因此，作为消费者，您希望发送与任何时间形式或任何UUID匹配的请求。这样，您的系统将像往常一样工作-会生成数据，而您无需存根任何东西。假设在上述JSON的情况下，最重要的部分是body字段。您可以专注于此并为其他字段提供匹配。换句话说，您希望存根像这样工作：

**{**

"time" : "SOMETHING THAT MATCHES TIME"**,**

"id" : "SOMETHING THAT MATCHES UUID"**,**

"body" : "foo"**}**

就响应作为消费者而言，您需要可以操作的具体价值。所以这样的JSON是有效的

**{**

"time" : "2016-10-10 21:10:15"**,**

"id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051"**,**

"body" : "bar"**}**

如您在前几节中所看到的，我们根据合同生成测试。因此，从生产者的角度来看，情况似乎大不相同。我们正在解析提供的合同，并且在测试中我们想向您的端点发送真实请求。因此，对于请求的生产者而言，我们无法进行任何形式的匹配。我们需要生产者后端可以使用的具体价值。这样的JSON是有效的：

**{**

"time" : "2016-10-10 20:10:15"**,**

"id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a"**,**

"body" : "foo"**}**

另一方面，从合同有效性的角度来看，响应不一定包含time或id的具体值。假设您是在生产者端生成的-再次，您必须进行大量的存根操作以确保始终返回相同的值。因此，从生产者的角度来看，您可能想要以下响应：

**{**

"time" : "SOMETHING THAT MATCHES TIME"**,**

"id" : "SOMETHING THAT MATCHES UUID"**,**

"body" : "bar"**}**

那么，您如何才能一次为消费者提供匹配者，为生产者提供具体价值，反之亦然？在Spring Cloud Contract中，我们允许您提供**动态值**。这意味着通信的双方可能会有所不同。您可以传递值：

通过value方法

value(consumer(...), producer(...))

value(stub(...), test(...))

value(client(...), server(...))

或使用$()方法

$(consumer(...), producer(...))

$(stub(...), test(...))

$(client(...), server(...))

您可以在[第94章](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl" \o "94.合同DSL)*[Contract DSL](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl" \o "94.合同DSL)*部分中了解有关此内容的更多信息。

调用value()或$()会告诉Spring Cloud Contract您将传递动态值。在consumer()方法内部，传递应该在使用者方（在生成的存根中）使用的值。在producer()方法内部，传递应该在生产方（在生成的测试中）使用的值。

|  |
| --- |
| IMG_531 |
| 如果一侧传递了正则表达式，而另一侧则没有传递，则另一侧将自动生成。 |

通常，您会将该方法与regex帮助方法一起使用。例如consumer(regex('[0-9]{10}'))。

概括起来，上述情况的合同看起来或多或少像这样（时间和UUID的正则表达式已简化，很可能是无效的，但在此示例中，我们希望保持非常简单）：

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

url '/someUrl'

body([

time : value(consumer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),

id: value(consumer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))

body: "foo"

])

}

response {

status OK()

body([

time : value(producer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),

id: value([producer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))

body: "bar"

])

}

}

|  |  |
| --- | --- |
| IMG_532 | **重要** |
| 请阅读[与JSON相关](https://groovy-lang.org/json.html" \t "https://www.springcloud.cc/_top)的[Groovy文档，](https://groovy-lang.org/json.html" \t "https://www.springcloud.cc/_top)以了解如何正确构造请求/响应主体。 |

## **88.4如何进行存根版本控制？**

### **88.4.1 API版本控制**

让我们尝试回答一个问题，即版本控制的真正含义。如果您指的是API版本，则有不同的方法。

* 使用超媒体，链接，并且不以任何方式对您的API进行版本控制
* 通过标题/网址传递版本

我不会尝试回答哪种方法更好的问题。应该选择适合您需求并允许您产生业务价值的任何东西。

假设您对API进行了版本控制。在这种情况下，您应提供与所支持版本一样多的合同。您可以为每个版本创建一个子文件夹，也可以将其附加到合同名称之后-更加适合您。

### **88.4.2 JAR版本控制**

如果用版本控制来表示包含存根的JAR版本，则实质上有两种主要方法。

假设您正在执行持续交付/部署，这意味着您每次通过管道都将生成一个新版本的jar，并且该jar可以随时投入生产。例如，您的jar版本如下所示（它建立于20.10.2016 at 20:15:21）：

1.0.0.20161020-201521-RELEASE

在这种情况下，您生成的存根罐将如下所示。

1.0.0.20161020-201521-RELEASE-stubs.jar

在这种情况下，引用存根时应在application.yml或@AutoConfigureStubRunner中提供最新版本的存根。您可以通过传递+符号来实现。例

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})

但是，如果版本是固定的（例如1.0.4.RELEASE或2.1.1），则必须设置jar版本的具体值。2.1.1的示例。

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})

### **88.4.3开发或生产存根**

您可以操纵分类器，以针对其他服务或已部署到生产中的服务的存根的当前开发版本运行测试。如果您更改构建以使用prod-stubs分类器部署存根，则在进行生产部署后，可以在一种情况下使用开发存根运行测试，在一种情况下使用产品存根运行测试。

使用存根开发版本的测试示例

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})

使用生产版本的存根进行测试的示例

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})

您还可以通过部署管道中的属性传递这些值。

## **88.5合同通用回购**

除了与生产者签订合同之外，存储合同的另一种方法是将合同放在一个共同的地方。这可能与安全问题有关，在这些安全问题中，消费者无法克隆生产者的代码。同样，如果您将合同放在一个地方，那么作为生产者，您将知道您有多少个消费者，以及将因本地变更而中断的消费者。

### **88.5.1回购结构**

假设我们有一个生产者，其坐标为com.example:server，并且有3个使用者：client1，client2，client3。然后，在具有通用合同的存储库中，您将具有以下设置（可以[在此处](https://github.com/spring-cloud/spring-cloud-contract/tree/2.1.x/samples/standalone/contracts" \t "https://www.springcloud.cc/_top)签出）：

├── com

│   └── example

│   └── server

│   ├── client1

│   │   └── expectation.groovy

│   ├── client2

│   │   └── expectation.groovy

│   ├── client3

│   │   └── expectation.groovy

│   └── pom.xml

├── mvnw

├── mvnw.cmd

├── pom.xml

└── src

└── assembly

└── contracts.xml

如您所见，在以斜杠分隔的groupid /工件ID文件夹（com/example/server）下，您对3个使用者（client1，client2和client3）有期望。期望是本文档中所述的标准Groovy DSL合同文件。该存储库必须产生一个JAR文件，该文件将仓库内容一一对应。

server文件夹中的pom.xml的示例。

<?xml version="1.0" encoding="UTF-8"?><project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Server Stubs</name>

<description>POM used to install locally stubs for consumer side</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.10.RELEASE</version>

<relativePath/>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

<spring-cloud-contract.version>2.1.6.BUILD-SNAPSHOT</spring-cloud-contract.version>

<spring-cloud-release.version>Greenwich.BUILD-SNAPSHOT

</spring-cloud-release.version>

<excludeBuildFolders>true</excludeBuildFolders>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-release.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- By default it would search under src/test/resources/ -->*

<contractsDirectory>${project.basedir}</contractsDirectory>

</configuration>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

</project>

如您所见，除了Spring Cloud Contract Maven插件以外，没有其他依赖项。消费者方必须运行这些pom才能运行mvn clean install -DskipTests在本地安装生产者项目的存根。

根文件夹中的pom.xml如下所示：

<?xml version="1.0" encoding="UTF-8"?><project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example.standalone</groupId>

<artifactId>contracts</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Contracts</name>

<description>Contains all the Spring Cloud Contracts, well, contracts. JAR used by the

producers to generate tests and stubs

</description>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>contracts</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<configuration>

<attach>true</attach>

<descriptor>${basedir}/src/assembly/contracts.xml</descriptor>

*<!-- If you want an explicit classifier remove the following line -->*

<appendAssemblyId>false</appendAssemblyId>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

它使用Assembly插件来构建包含所有合同的JAR。这种设置的示例在这里：

<assembly xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>project</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>${project.basedir}</directory>

<outputDirectory>/</outputDirectory>

<useDefaultExcludes>true</useDefaultExcludes>

<excludes>

<exclude>\*\*/${project.build.directory}/\*\*</exclude>

<exclude>mvnw</exclude>

<exclude>mvnw.cmd</exclude>

<exclude>.mvn/\*\*</exclude>

<exclude>src/\*\*</exclude>

</excludes>

</fileSet>

</fileSets></assembly>

### **88.5.2工作流程**

该工作流程看上去与Step by step guide to CDC中介绍的工作流程相似。唯一的区别是，生产者不再拥有合同。因此，消费者和生产者必须在公共存储库中处理公共合同。

### **88.5.3消费者**

当**消费者**希望脱机处理合同时，而不是克隆生产者代码，消费者团队将克隆公共存储库，转到所需的生产者文件夹（例如com/example/server）并运行mvn clean install -DskipTests以在本地安装存根。从合同转换。

|  |
| --- |
| IMG_533 |
| 您需要在[本地安装Maven](https://maven.apache.org/download.cgi" \t "https://www.springcloud.cc/_top) |

### **88.5.4生产者**

作为**生产者**，足以更改Spring Cloud Contract验证程序以提供URL和包含合同的JAR依赖项：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<contractsMode>REMOTE</contractsMode>

<contractsRepositoryUrl>

https://link/to/your/nexus/or/artifactory/or/sth

</contractsRepositoryUrl>

<contractDependency>

<groupId>com.example.standalone</groupId>

<artifactId>contracts</artifactId>

</contractDependency>

</configuration></plugin>

通过此设置，将从[http://link/to/your/nexus/or/artifactory/or/sth](http://link/to/your/nexus/or/artifactory/or/sth" \t "https://www.springcloud.cc/_top)下载组ID为com.example.standalone和工件为contracts的JAR。然后将其解压缩到本地临时文件夹中，并选择com/example/server下的合同作为生成测试和存根的合同。根据该约定，当完成一些不兼容的更改时，生产者团队将知道哪些消费者团队将被破坏。

其余流程看起来相同。

### **88.5.5如何按主题而不是按生产者定义消息传递合同？**

为了避免通用仓库中的消息合同重复，当很少有生产者将消息写到一个主题时，我们可以创建一个结构，将其余合同放置在每个生产者的文件夹中，并将消息合同放置在每个主题的文件夹中。

#### **对于Maven项目**

为了能够在生产者端进行工作，我们应该指定一个包含模式，以通过我们感兴趣的消息传递主题过滤通用存储库jar。Maven Spring Cloud Contract plugin的includedFiles属性允许我们执行此操作。还需要指定contractsPath，因为默认路径将是公用存储库groupid/artifactid。

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<configuration>

<contractsMode>REMOTE</contractsMode>

<contractsRepositoryUrl>http://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>

<contractDependency>

<groupId>com.example</groupId>

<artifactId>common-repo-with-contracts</artifactId>

<version>+</version>

</contractDependency>

<contractsPath>/</contractsPath>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*messaging.\*</contractPackageRegex>

<baseClassFQN>com.example.services.MessagingBase</baseClassFQN>

</baseClassMapping>

<baseClassMapping>

<contractPackageRegex>.\*rest.\*</contractPackageRegex>

<baseClassFQN>com.example.services.TestBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

<includedFiles>

<includedFile>\*\*/${project.artifactId}/\*\*</includedFile>

<includedFile>\*\*/${first-topic}/\*\*</includedFile>

<includedFile>\*\*/${second-topic}/\*\*</includedFile>

</includedFiles>

</configuration></plugin>

#### **对于Gradle项目**

* 为common-repo依赖项添加定制配置：

ext {

conractsGroupId = "com.example"

contractsArtifactId = "common-repo"

contractsVersion = "1.2.3"

}

configurations {

contracts {

transitive = false

}

}

* 将common-repo依赖项添加到您的类路径中：

dependencies {

contracts "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"

testCompile "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"

}

* 将依赖项下载到适当的文件夹：

task getContracts(type: Copy) {

from configurations.contracts

into **new** File(project.buildDir, "downloadedContracts")

}

* 解压缩JAR：

task unzipContracts(type: Copy) {

def zipFile = **new** File(project.buildDir, "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")

def outputDir = file("${buildDir}/unpackedContracts")

from zipTree(zipFile)

into outputDir

}

* 清理未使用的合同：

task deleteUnwantedContracts(type: Delete) {

delete fileTree(dir: "${buildDir}/unpackedContracts",

include: "\*\*/\*",

excludes: [

"\*\*/${project.name}/\*\*"", "\*\*/${first-topic}/\*\*",

"\*\*/${second-topic}/\*\*"])

}

* 创建任务依赖项：

unzipContracts.dependsOn("getContracts")

deleteUnwantedContracts.dependsOn("unzipContracts")

build.dependsOn("deleteUnwantedContracts")

* 通过使用contractsDslDir属性指定包含合同的目录来配置插件

contracts {

contractsDslDir = **new** File("${buildDir}/unpackedContracts")

}

## **88.6我需要二进制存储吗？我不能使用Git吗？**

在多语言的世界中，有些语言不使用二进制存储，例如Artifactory或Nexus。从Spring Cloud Contract版本2.0.0开始，我们提供了在SCM存储库中存储合同和存根的机制。当前唯一支持的SCM是Git。

存储库必须进行以下设置（您可以[在此处](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/contracts_git/" \t "https://www.springcloud.cc/_top)检出）：

.

└── META-INF

└── com.example

└── beer-api-producer-git

└── 0.0.1-SNAPSHOT

├── contracts

│   └── beer-api-consumer

│   ├── messaging

│   │   ├── shouldSendAcceptedVerification.groovy

│   │   └── shouldSendRejectedVerification.groovy

│   └── rest

│   ├── shouldGrantABeerIfOldEnough.groovy

│   └── shouldRejectABeerIfTooYoung.groovy

└── mappings

└── beer-api-consumer

└── rest

├── shouldGrantABeerIfOldEnough.json

└── shouldRejectABeerIfTooYoung.json

在META-INF文件夹下：

* 我们通过groupId（例如，com.example）对应用程序进行分组
* 那么每个应用程序都通过artifactId（例如beer-api-producer-git）表示

接下来，是应用程序的版本（例如0.0.1-SNAPSHOT）。从Spring Cloud Contract版本2.1.0开始，您可以指定以下版本（假设您的版本遵循语义版本）

+或latest-查找存根的最新版本（假设快照始终是给定修订版本的最新工件）。这意味着：

* + - 如果您有1.0.0.RELEASE，2.0.0.BUILD-SNAPSHOT和2.0.0.RELEASE版本，我们将假定最新版本为2.0.0.BUILD-SNAPSHOT
    - 如果您使用的版本为1.0.0.RELEASE和2.0.0.RELEASE，我们将假定最新版本为2.0.0.RELEASE
    - 如果您有一个名为latest或+的版本，我们将选择该文件夹

release-查找存根的最新版本。这意味着：

* + - 如果您使用的版本为1.0.0.RELEASE，2.0.0.BUILD-SNAPSHOT和2.0.0.RELEASE，我们将假定最新版本为2.0.0.RELEASE
    - 如果您有一个名为release的版本，我们将选择该文件夹

最后，有两个文件夹：

* + contracts-优良作法是将每个消费者所需的合同与消费者名称一起存储在文件夹中（例如beer-api-consumer）。这样，您可以使用stubs-per-consumer功能。进一步的目录结构是任意的。
  + mappings-在该文件夹中，Maven / Gradle Spring Cloud Contract插件将推送存根服务器映射。在使用者方面，Stub Runner将扫描此文件夹以使用存根定义启动存根服务器。文件夹结构将是在contracts子文件夹中创建的文件夹的副本。

### **88.6.1协议约定**

为了控制合同来源的类型和位置（无论是二进制存储还是SCM存储库），可以在存储库URL中使用协议。Spring Cloud Contract遍历已注册的协议解析器，并尝试获取合同（通过插件）或存根（通过Stub Runner）。

目前，对于SCM功能，我们支持Git存储库。要使用它，在属性中需要放置存储库URL的位置，您只需在连接URL前面加上git://。在这里您可以找到几个示例：

git://file:///foo/bar

git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git

git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git

### **88.6.2生产者**

对于生产者，要使用SCM方法，我们可以重用与外部合同相同的机制。我们通过包含git://协议的URL路由Spring Cloud Contract以使用SCM实现。

|  |  |
| --- | --- |
| IMG_534 | **重要** |
| 您必须在Maven中手动添加pushStubsToScm目标，或者在Gradle中执行（绑定）pushStubsToScm任务。我们不会开箱即用将存根推送到您的git存储库的origin。 |

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

*<!-- By default we will not push the stubs back to SCM,*

*you have to explicitly add it as a goal -->*

<goal>pushStubsToScm</goal>

</goals>

</execution>

</executions></plugin>

**Gradle.**

contracts {

// We want to pick contracts from a Git repository

contractDependency {

stringNotation = "${project.group}:${project.name}:${project.version}"

}

/\*

We reuse the contract dependency section to set up the path

to the folder that contains the contract definitions. In our case the

path will be /groupId/artifactId/version/contracts

\*/

contractRepository {

repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git"

}

// The mode can't be classpath

contractsMode = "REMOTE"

// Base class mappings etc.

}

/\*

In this scenario we want to publish stubs to SCM whenever

the `publish` task is executed

\*/

publish.dependsOn("publishStubsToScm")

通过这样的设置：

* Git项目将被克隆到一个临时目录
* SCM存根下载器将转到META-INF/groupId/artifactId/version/contracts文件夹以查找合同。例如，对于com.example:foo:1.0.0，路径为META-INF/com.example/foo/1.0.0/contracts
* 将根据合同生成测试
* 将根据合同创建存根
* 测试通过后，存根将在克隆的存储库中提交
* 最后，将推动该存储库的origin

### **88.6.3生产者，合同存储在本地**

使用SCM作为存根和合同目的地的另一种选择是与生产者一起在本地存储合同，并且仅将合同和存根推送到SCM。在下面，您可以找到使用Maven和Gradle完成此操作所需的设置。

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

*<!-- In the default configuration, we want to use the contracts stored locally -->*

<configuration>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*messaging.\*</contractPackageRegex>

<baseClassFQN>com.example.BeerMessagingBase</baseClassFQN>

</baseClassMapping>

<baseClassMapping>

<contractPackageRegex>.\*rest.\*</contractPackageRegex>

<baseClassFQN>com.example.BeerRestBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

<basePackageForTests>com.example</basePackageForTests>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

*<!-- By default we will not push the stubs back to SCM,*

*you have to explicitly add it as a goal -->*

<goal>pushStubsToScm</goal>

</goals>

<configuration>

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>git://file://${env.ROOT}/target/contract\_empty\_git/

</contractsRepositoryUrl>

*<!-- Example of URL via git protocol -->*

*<!--<contractsRepositoryUrl>git://git@github.com:spring-cloud-samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->*

*<!-- Example of URL via http protocol -->*

*<!--<contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->*

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</contractDependency>

*<!-- The mode can't be classpath -->*

<contractsMode>LOCAL</contractsMode>

</configuration>

</execution>

</executions></plugin>

**Gradle.**

contracts {

// Base package for generated tests

basePackageForTests = "com.example"

baseClassMappings {

baseClassMapping(".\*messaging.\*", "com.example.BeerMessagingBase")

baseClassMapping(".\*rest.\*", "com.example.BeerRestBase")

}

}

/\*

In this scenario we want to publish stubs to SCM whenever

the `publish` task is executed

\*/

publishStubsToScm {

// We want to modify the default set up of the plugin when publish stubs to scm is called

customize {

// We want to pick contracts from a Git repository

contractDependency {

stringNotation = "${project.group}:${project.name}:${project.version}"

}

/\*

We reuse the contract dependency section to set up the path

to the folder that contains the contract definitions. In our case the

path will be /groupId/artifactId/version/contracts

\*/

contractRepository {

repositoryUrl = "git://file://${System.getenv("ROOT")}/target/contract\_empty\_git/"

}

// The mode can't be classpath

contractsMode = "LOCAL"

}

}

publish.dependsOn("publishStubsToScm")

publishToMavenLocal.dependsOn("publishStubsToScm")

通过这样的设置：

* 从默认的src/test/resources/contracts目录中选择Contracts
* 将根据合同生成测试
* 将根据合同创建存根

一旦测试通过

* + Git项目将被克隆到一个临时目录
  + 存根和合同将在克隆的存储库中提交
* 最后，将推动该存储库的origin

#### **与生产者和存根之间的合同保持一致**

也可以将合同保留在生产者存储库中，但将存根保留在外部git repo中。当您想使用基本的消费者-生产者协作流程，但又无法使用工件存储库来存储存根时，这是最有用的。

为此，请使用通常的生产者设置，然后添加pushStubsToScm目标并在要保留存根的存储库中设置contractsRepositoryUrl。

### **88.6.4消费者**

在使用者方面，通过@AutoConfigureStubRunner批注，JUnit规则，JUnit 5扩展名或属性传递repositoryRoot参数时，足以传递带有协议前缀的SCM存储库的URL。例如

@AutoConfigureStubRunner(

stubsMode="REMOTE",

repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git",

ids="com.example:bookstore:0.0.1.RELEASE"

)

通过这样的设置：

* Git项目将被克隆到一个临时目录
* SCM存根下载器将转到META-INF/groupId/artifactId/version/文件夹以查找存根定义和合同。例如，对于com.example:foo:1.0.0，路径为META-INF/com.example/foo/1.0.0/
* 存根服务器将启动并提供映射
* 将在消息传递测试中读取和使用消息传递定义

## **88.7我可以使用契约代理吗？**

使用[Pact时](https://pact.io/" \t "https://www.springcloud.cc/_top)，可以使用[Pact Broker](https://github.com/pact-foundation/pact_broker" \t "https://www.springcloud.cc/_top) 来存储和共享Pact定义。从Spring Cloud Contract 2.0.0开始，可以从Pact Broker中获取Pact文件以生成测试和存根。

作为前提条件，需要使用Pact Converter和Pact Stub Downloader。您必须通过spring-cloud-contract-pact依赖项添加它们。您可以在[第96.1.1节“协议转换器”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pact-converter" \o "96.1.1 Pact转换器)部分中了解更多信息。

|  |  |
| --- | --- |
| IMG_535 | **重要** |
| 条约遵循消费者合同约定。这意味着消费者首先创建契约约定，然后与生产者共享文件。这些期望是由消费者的代码产生的，如果不满足期望，则可能破坏生产者。 |

### **88.7.1契约使用者**

使用者使用Pact框架生成Pact文件。该契约文件将发送到契约代理。可以在[此处](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/consumer_pact" \t "https://www.springcloud.cc/_top)找到此类设置的示例。

### **88.7.2生产者**

对于生产者，要使用Pact Broker中的Pact文件，我们可以重复使用与外部合同相同的机制。我们通过包含pact://协议的URL路由Spring Cloud Contract以使用Pact实现。只需将URL传递给Pact Broker。可以在[此处](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/producer_pact" \t "https://www.springcloud.cc/_top)找到此类设置的示例。

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>pact://http://localhost:8085</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

*<!-- When + is passed, a latest tag will be applied when fetching pacts -->*

<version>+</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

*<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->*

<dependencies>

<dependency>

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

<artifactId>spring-cloud-contract-pact</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

</dependencies></plugin>

**Gradle.**

buildscript {

repositories {

//...

}

dependencies {

// ...

// Don't forget to add spring-cloud-contract-pact to the classpath!

classpath "org.springframework.cloud:spring-cloud-contract-pact:${contractVersion}"

}

}

contracts {

// When + is passed, a latest tag will be applied when fetching pacts

contractDependency {

stringNotation = "${project.group}:${project.name}:+"

}

contractRepository {

repositoryUrl = "pact://http://localhost:8085"

}

// The mode can't be classpath

contractsMode = "REMOTE"

// Base class mappings etc.

}

通过这样的设置：

* 契约文件将从契约代理下载
* Spring Cloud Contract将把Pact文件转换为测试和存根
* 与存根一样的JAR会像往常一样自动创建

### **88.7.3契约消费者（生产者合同法）**

在您不想执行“消费者合同”方法（为每个消费者定义期望）但您更愿意使用“生产者Contracts”（生产者提供合同并发​​布存根）的情况下，足够使用Spring Cloud Contract和Stub Runner选项。可以在[此处](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/consumer_pact_stubrunner" \t "https://www.springcloud.cc/_top)找到此类设置的示例。

首先，请记住添加Stub Runner和Spring Cloud Contract Pact模块作为测试依赖项。

**Maven.**

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

*<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->*<dependencies>

*<!-- ... -->*

<dependency>

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

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>spring-cloud-contract-pact</artifactId>

<scope>test</scope>

</dependency></dependencies>

**Gradle.**

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"

}

}

dependencies {

//...

testCompile("org.springframework.cloud:spring-cloud-starter-contract-stub-runner")

// Don't forget to add spring-cloud-contract-pact to the classpath!

testCompile("org.springframework.cloud:spring-cloud-contract-pact")

}

接下来，只需将Pact Broker的URL传递到以pact://协议为前缀的repositoryRoot。例如pact://http://localhost:8085

@RunWith(SpringRunner.class)@SpringBootTest@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.REMOTE,

ids = "com.example:beer-api-producer-pact",

repositoryRoot = "pact://http://localhost:8085")**public** **class** BeerControllerTest {

*//Inject the port of the running stub*

@StubRunnerPort("beer-api-producer-pact") **int** producerPort;

*//...*

}

通过这样的设置：

* 契约文件将从契约代理下载
* Spring Cloud Contract将把Pact文件转换为存根定义
* 存根服务器将启动并被存入存根

有关Pact支持的更多信息，请转至[第96.7节“使用Pact存根下载器”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "pact-stub-downloader" \o "96.7使用契约存根下载器)部分。

## **88.8如何调试由生成的测试客户端发送的请求/响应？**

生成的测试全部以依赖[Apache HttpClient的](https://hc.apache.org/httpcomponents-client-ga/" \t "https://www.springcloud.cc/_top)某种形式或方式归结为RestAssured 。HttpClient具有一种称为“ [有线记录”的功能](https://hc.apache.org/httpcomponents-client-ga/logging.html" \l "Wire_Logging" \t "https://www.springcloud.cc/_top)，该功能会将整个请求和响应记录到HttpClient中。Spring Boot具有用于执行此类操作的日志记录[通用应用程序属性](https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html" \t "https://www.springcloud.cc/_top)，只需将其添加到您的应用程序属性中

logging.level.org.apache.http.wire=DEBUG

### **88.8.1如何调试WireMock发送的映射/请求/响应？**

从版本1.2.0开始，我们将WireMock日志记录打开到info，而将WireMock通知程序打开为冗长。现在，您将完全知道WireMock服务器收到了什么请求，以及选择了哪个匹配的响应定义。

要关闭此功能，只需将WireMock日志记录更改为ERROR

logging.level.com.github.tomakehurst.wiremock=ERROR

### **88.8.2如何查看在HTTP服务器存根中注册了什么？**

您可以使用@AutoConfigureStubRunner，StubRunnerRule或`StubRunnerExtension`上的mappingsOutputFolder属性来按工件ID转储所有映射。同样，将启动给定存根服务器的启动端口。

### **88.8.3我可以引用文件中的文本吗？**

是! 在1.2.0版中，我们添加了这种可能性。在DSL中调用file(…​)方法并提供相对于合同所在位置的路径就足够了。如果您使用的是YAML，则只需使用bodyFromFile属性。

## **89. Spring Cloud Contract验证程序设置**

您可以通过以下方式设置Spring Cloud Contract验证程序：

* [作为Gradle项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-project" \o "89.1 Gradle项目)
* [作为Maven项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-project" \o "90.13 Maven项目)
* [作为Docker项目](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "docker-project" \o "90.16 Docker项目)

## **89.1 Gradle项目**

要了解如何为Spring Cloud Contract验证程序设置Gradle项目，请阅读以下部分：

* [第89.1.1节“前提条件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-prerequisites" \o "89.1.1前提条件)
* [第90章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-add-gradle-plugin" \o "90.添加具有依赖性的Gradle插件)*[添加具有依赖性的Gradle插件](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-add-gradle-plugin" \o "90.添加具有依赖性的Gradle插件)*
* [第90.1节“ Gradle和“保证放心的2.0””](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-and-rest-assured" \o "90.1摇摇欲坠和保证放心2.0)
* [第90.2节“ Gradle的快照版本”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-snapshot-versions" \o "90.2 Gradle的快照版本)
* [第90.3节“添加存根”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-add-stubs" \o "90.3添加存根)
* [第90.5节“默认设置”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-default-setup" \o "90.5默认设置)
* [第90.6节“配置插件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-configure-plugin" \o "90.6配置插件)
* [第90.7节“配置选项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-configuration-options" \o "90.7配置选项)
* [第90.8节“所有测试的单一基类”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-single-base-class" \o "90.8所有测试的单一基类)
* [第90.9节“ Contracts的不同基类”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-different-base-classes" \o "90.9合同的不同基本类别)
* [第90.10节“调用生成的测试”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-invoking-generated-tests" \o "90.10调用生成的测试)
* [第90.11节“将存根推送到SCM”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-pushing-stubs-to-scm" \o "90.11将存根推送到SCM)
* [第90.12节“消费者方的Spring Cloud Contract验证程序”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gradle-consumer" \o "90.12消费者方的Spring Cloud Contract Verifier)

### **89.1.1前提条件**

为了将Spring Cloud Contract验证程序与WireMock一起使用，您必须使用Gradle或Maven插件。

|  |
| --- |
| IMG_536 |
| 如果要在项目中使用Spock，则必须分别添加spock-core和spock-spring模块。查看[Spock文档以获取更多信息](https://spockframework.github.io/" \t "https://www.springcloud.cc/_top) |

## **90.添加具有依赖性的Gradle插件**

要添加具有依赖性的Gradle插件，可以使用类似于以下代码：

**插件DSL GA版本。**

*// build.gradle*

plugins {

id "groovy"

*// this will work only for GA versions of Spring Cloud Contract*

id "org.springframework.cloud.contract" version "${GAVerifierVersion}"

}

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${GAVerifierVersion}"

}

}

dependencies {

testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"

*// example with adding Spock core and Spock Spring*

testCompile "org.spockframework:spock-core:${spockVersion}"

testCompile "org.spockframework:spock-spring:${spockVersion}"

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

}

**插件DSL非GA版本。**

*// settings.gradle*

pluginManagement {

plugins {

id "org.springframework.cloud.contract" version "${verifierVersion}"

}

repositories {

*// to pick from local .m2*

mavenLocal()

*// for snapshots*

maven { url "https://repo.spring.io/snapshot" }

*// for milestones*

maven { url "https://repo.spring.io/milestone" }

*// for GA versions*

gradlePluginPortal()

}

}

*// build.gradle*

plugins {

id "groovy"

id "org.springframework.cloud.contract"

}

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifierVersion}"

}

}

dependencies {

testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"

*// example with adding Spock core and Spock Spring*

testCompile "org.spockframework:spock-core:${spockVersion}"

testCompile "org.spockframework:spock-spring:${spockVersion}"

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

}

**旧版插件应用程序。**

*// build.gradle*

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot\_version}"

classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier\_version}"

*// here you can also pass additional dependencies such as Pact or Kotlin spec e.g.:*

*// classpath "org.springframework.cloud:spring-cloud-contract-spec-kotlin:${verifier\_version}"*

}

}

apply plugin: 'groovy'

apply plugin: 'spring-cloud-contract'

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifier\_version}"

}

}

dependencies {

testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"

*// example with adding Spock core and Spock Spring*

testCompile "org.spockframework:spock-core:${spockVersion}"

testCompile "org.spockframework:spock-spring:${spockVersion}"

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

}

## **90.1 Gradle和“保证放心” 2.0**

默认情况下，Rest Assured 3.x被添加到类路径中。但是，要使用Rest Assured 2.x，可以将其添加到插件的classpath中，如下所示：

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot\_version}"

classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier\_version}"

classpath "com.jayway.restassured:rest-assured:2.5.0"

classpath "com.jayway.restassured:spring-mock-mvc:2.5.0"

}

}

depenendencies {

*// all dependencies*

*// you can exclude rest-assured from spring-cloud-contract-verifier*

testCompile "com.jayway.restassured:rest-assured:2.5.0"

testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"

}

这样，插件会自动看到类路径中存在Rest Assured 2.x，并相应地修改了导入。

## **Gradle的90.2快照版本**

将其他快照存储库添加到build.gradle以使用快照版本，快照版本在每次成功构建后都会自动上传，如下所示：

*/\**

*We need to use the [buildscript {}] section when we have to modify*

*the classpath for the plugins. If that's not the case this section*

*can be skipped.*

*If you don't need to modify the classpath (e.g. add a Pact dependency),*

*then you can just set the [pluginManagement {}] section in [settings.gradle] file.*

*// settings.gradle*

*pluginManagement {*

*repositories {*

*// for snapshots*

*maven {url "https://repo.spring.io/snapshot"}*

*// for milestones*

*maven {url "https://repo.spring.io/milestone"}*

*// for GA versions*

*gradlePluginPortal()*

*}*

*}*

*\*/*

buildscript {

repositories {

mavenCentral()

mavenLocal()

maven { url "https://repo.spring.io/snapshot" }

maven { url "https://repo.spring.io/milestone" }

maven { url "https://repo.spring.io/release" }

}

}

## **90.3添加存根**

默认情况下，Spring Cloud Contract验证程序在src/test/resources/contracts目录中寻找存根。

包含存根定义的目录被视为类名，每个存根定义均被视为单个测试。Spring Cloud Contract验证程序假定它包含至少一层要用作测试类名称的目录。如果存在多个嵌套目录，则使用除最后一个嵌套目录以外的所有目录作为包名。例如，具有以下结构：

src/test/resources/contracts/myservice/shouldCreateUser.groovy

src/test/resources/contracts/myservice/shouldReturnUser.groovy

Spring Cloud Contract验证程序使用两种方法创建名为defaultBasePackage.MyService的测试类：

* shouldCreateUser()
* shouldReturnUser()

## **90.4运行插件**

该插件注册自己以在check任务之前被调用。如果您希望它成为构建过程的一部分，则无需执行其他任何操作。如果只想生成测试，请调用generateContractTests任务。

## **90.5默认设置**

默认的Gradle插件设置会创建该版本的以下Gradle部分（以伪代码）：

contracts {

testFramework ='JUNIT'

testMode = 'MockMvc'

generatedTestSourcesDir = project.file("${project.buildDir}/generated-test-sources/contracts")

generatedTestResourcesDir = project.file("${project.buildDir}/generated-test-resources/contracts")

contractsDslDir = file("${project.rootDir}/src/test/resources/contracts")

basePackageForTests = 'org.springframework.cloud.verifier.tests'

stubsOutputDir = project.file("${project.buildDir}/stubs")

*// the following properties are used when you want to provide where the JAR with contract lays*

contractDependency {

stringNotation = ''

}

contractsPath = ''

contractsWorkOffline = false

contractRepository {

cacheDownloadedContracts(true)

}

}

tasks.create(type: Jar, name: 'verifierStubsJar', dependsOn: 'generateClientStubs') {

baseName = project.name

classifier = contracts.stubsSuffix

from contractVerifier.stubsOutputDir

}

project.artifacts {

archives task

}

tasks.create(type: Copy, name: 'copyContracts') {

from contracts.contractsDslDir

into contracts.stubsOutputDir

}

verifierStubsJar.dependsOn 'copyContracts'

publishing {

publications {

stubs(MavenPublication) {

artifactId project.name

artifact verifierStubsJar

}

}

}

## **90.6配置插件**

要更改默认配置，请在Gradle配置中添加一个contracts代码段，如下所示：

contracts {

testMode = 'MockMvc'

baseClassForTests = 'org.mycompany.tests'

generatedTestSourcesDir = project.file('src/generatedContract')

}

## **90.7配置选项**

* **testMode**：定义验收测试的模式。默认情况下，模式是MockMvc，它基于Spring的MockMvc。对于真实的HTTP调用，也可以将其更改为**WebTestClient**，**JaxRsClient**或 **Explicit**。
* **import**：创建一个带有导入的数组，该数组应包含在生成的测试中（例如['org.myorg.Matchers']）。默认情况下，它将创建一个空数组。
* **staticImports**：创建一个包含静态导入的数组，该数组应包含在生成的测试中（例如['org.myorg.Matchers。\*']）。默认情况下，它将创建一个空数组。
* **basePackageForTests**：指定所有生成的测试的基本软件包。如果未设置，则从baseClassForTests’s package and from `packageWithBaseClasses中选择值。如果这些值均未设置，则该值设置为org.springframework.cloud.contract.verifier.tests。
* **baseClassForTests**：为所有生成的测试创建基类。默认情况下，如果您使用Spock类，则该类为spock.lang.Specification。
* **packageWithBaseClasses**：定义所有基类所在的包。此设置优先于**baseClassForTests**。
* **baseClassMappings**：明确地将合同包映射到基类的FQN。此设置优先于**packageWithBaseClasses**和**baseClassForTests**。
* **ruleClassForTests**：指定应添加到生成的测试类的规则。
* **ignoreFiles**：使用Antmatcher来定义应跳过处理的存根文件。默认情况下，它是一个空数组。
* **ContractsDslDir**：指定包含使用GroovyDSL编写的合同的目录。默认情况下，其值为$rootDir/src/test/resources/contracts。
* **createdTestSourcesDir**：指定应放置从Groovy DSL生成的测试的测试源目录。默认情况下，其值为$buildDir/generated-test-sources/contracts。
* **createdTestResourcesDir**：指定测试资源目录，应该放置Groovy DSL生成的测试所使用的资源。默认情况下，其值为$buildDir/generated-test-resources/contracts。
* **stubsOutputDir**：指定应放置从Groovy DSL生成的WireMock存根的目录。
* **testFramework**：指定要使用的目标测试框架。当前，Spock，JUnit 4（TestFramework.JUNIT）和JUnit 5受支持，而JUnit 4是默认框架。
* **contractProperties**：包含要传递给Spring Cloud Contract组件的属性的映射。这些属性可能由内置或自定义存根下载器使用。

当您要指定包含合同的JAR的位置时，使用以下属性：

* **contractDependency**：指定提供groupid:artifactid:version:classifier坐标的依赖关系。您可以使用contractDependency闭包进行设置。
* **ContractsPath**：指定jar的路径。如果下载了合同依存关系，则路径默认为groupid/artifactid，其中groupid以斜杠分隔。否则，它将在提供的目录下扫描合同。
* **ContractsMode**：指定下载合同的方式（JAR是否可以脱机使用，远程使用等）。
* **deleteStubsAfterTest**：如果设置为false，则不会从临时目录中删除任何下载的合同

您可以在下面找到通过插件打开的实验功能列表：

* **convertToYaml**：将所有DSL转换为声明性的YAML格式。当您在Groovy DSL中使用外部库时，这可能非常有用。通过启用此功能（将其设置为true），您将不需要在使用者方面添加库依赖项。
* **assertJsonSize**：您可以在生成的测试中检查JSON数组的大小。默认情况下禁用此功能。

## **90.8所有测试的单一基类**

在默认的MockMvc中使用Spring Cloud Contract验证程序时，您需要为所有生成的验收测试创建基本规范。在此类中，您需要指向一个端点，该端点应进行验证。

**abstract** **class** BaseMockMvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** PairIdController())

}

**void** isProperCorrelationId(Integer correlationId) {

assert correlationId == 123456

}

**void** isEmpty(String value) {

assert value == null

}

}

如果使用Explicit模式，则可以使用基类来初始化整个测试的应用程序，就像在常规集成测试中看到的那样。如果使用JAXRSCLIENT模式，则此基类还应包含一个protected WebTarget webTarget字段。目前，测试JAX-RS API的唯一选项是启动web服务器。

## **90.9 Contracts的不同基类**

如果合同之间的基类不同，则可以告诉Spring Cloud Contract插件自动生成的测试应扩展哪个类。您有两种选择：

* 遵循约定，提供packageWithBaseClasses
* 通过baseClassMappings提供显式映射

**按照惯例**

约定是这样的：如果您在src/test/resources/contract/foo/bar/baz/下拥有合同，并且将packageWithBaseClasses属性的值设置为com.example.base，则Spring Cloud Contract验证程序会假设存在一个BarBazBase com.example.base包下的类。换句话说，系统将获取包的最后两个部分（如果存在），并形成一个后缀为Base的类。此规则优先于**baseClassForTests**。这是一个在contracts闭包中如何工作的示例：

packageWithBaseClasses = 'com.example.base'

**通过映射**

您可以将合同包的正则表达式手动映射到匹配合同的基类的完全限定名称。您必须提供一个名为baseClassMappings的列表，该列表由baseClassMapping对象组成，这些对象采用从contractPackageRegex到baseClassFQN的映射。考虑以下示例：

baseClassForTests = "com.example.FooBase"

baseClassMappings {

baseClassMapping('.\*/com/.\*', 'com.example.ComBase')

baseClassMapping('.\*/bar/.\*': 'com.example.BarBase')

}

假设您的合同是-src/test/resources/contract/com/-src/test/resources/contract/foo/

通过提供baseClassForTests，我们可以在没有成功映射的情况下进行回退。（您也可以提供packageWithBaseClasses作为后备。）这样，从src/test/resources/contract/com/合约生成的测试扩展了com.example.ComBase，而其余测试扩展了com.example.FooBase。

## **90.10调用生成的测试**

为了确保提供方符合已定义的合同，您需要调用：

./gradlew generateContractTests **test**

## **90.11将存根推送到SCM**

如果您使用SCM存储库保留合同和存根，则可能需要自动化将存根推入存储库的步骤。为此，只需调用pushStubsToScm任务即可。例：

$ ./gradlew pushStubsToScm

在[第96.6节“使用SCM存根下载器”下，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "scm-stub-downloader" \o "96.6使用SCM存根下载器)您可以找到可以通过contractsProperties字段（例如contracts { contractsProperties = [foo:"bar"] }），通过contractsProperties方法（例如contracts { contractsProperties([foo:"bar"]) }）传递的所有可能的配置选项属性或环境变量。

## **90.12 Spring Cloud Contract使用者方面的验证者**

在消费服务中，您需要以与提供者完全相同的方式配置Spring Cloud Contract Verifier插件。如果您不想使用Stub Runner，则需要复制存储在src/test/resources/contracts中的合同，并使用以下方法生成WireMock JSON存根：

./gradlew generateClientStubs

|  |
| --- |
| IMG_537 |
| 必须设置stubsOutputDir选项，才能生成存根。 |

如果存在，JSON存根可以用于使用服务的自动化测试中。

@ContextConfiguration(loader == SpringApplicationContextLoader, classes == Application)**class** LoanApplicationServiceSpec **extends** Specification {

@ClassRule

@Shared

WireMockClassRule wireMockRule == **new** WireMockClassRule()

@Autowired

LoanApplicationService sut

def 'should successfully apply for loan'() {

given:

LoanApplication application =

**new** LoanApplication(client: **new** Client(clientPesel: '12345678901'), amount: 123.123)

when:

LoanApplicationResult loanApplication == sut.loanApplication(application)

then:

loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN\_APPLIED

loanApplication.rejectionReason == null

}

}

LoanApplication致电FraudDetection服务。该请求由配置有Spring Cloud Contract验证程序生成的存根的WireMock服务器处理。

## **90.13 Maven项目**

要了解如何为Spring Cloud Contract验证程序设置Maven项目，请阅读以下部分：

* [第90.13.1节“添加maven插件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-add-plugin" \o "90.13.1添加Maven插件)
* [第90.13.2节“ Maven和“保证放心2.0””](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-rest-assured" \o "90.13.2 Maven和安心保障2.0)
* [第90.13.3节“ Maven的快照版本”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-snapshot-versions" \o "90.13.3 Maven的快照版本)
* [第90.13.4节“添加存根”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-add-stubs" \o "90.13.4添加存根)
* [第90.13.5节“运行插件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-run-plugin" \o "90.13.5运行插件)
* [第90.13.6节“配置插件”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-configure-plugin" \o "90.13.6配置插件)
* [第90.13.7节“配置选项”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-configuration-options" \o "90.13.7配置选项)
* [第90.13.8节“所有测试的单一基类”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-single-base" \o "90.13.8所有测试的单一基类)
* [第90.13.9节“合同的不同基类”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-different-base" \o "90.13.9合同的不同基类)
* [第90.13.10节，“调用生成的测试”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-invoking-generated-tests" \o "90.13.10调用生成的测试)
* [第90.13.11节，“将存根推送到SCM”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-pushing-stubs-to-scm" \o "90.13.11将存根推送到SCM)
* [第90.13.12节，“ Maven插件和STS”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "maven-sts" \o "90.13.12 Maven插件和STS)

### **90.13.1添加Maven插件**

以类似以下方式添加Spring Cloud Contract BOM：

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-release.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

接下来，添加Spring Cloud Contract Verifier Maven插件：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

<convertToYaml>true</convertToYaml>

</configuration></plugin>

您可以在 [Spring Cloud Contract Maven插件文档（2.0.0.RELEASE版本的示例）中了解更多信息](https://cloud.spring.io/spring-cloud-static/spring-cloud-contract/2.0.0.RELEASE/spring-cloud-contract-maven-plugin/" \t "https://www.springcloud.cc/_top)。

### **90.13.2 Maven和“保证放心” 2.0**

默认情况下，Rest Assured 3.x被添加到类路径中。但是，可以通过将Rest Assured 2.x添加到插件类路径中来使用它，如下所示：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example</packageWithBaseClasses>

</configuration>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-contract-verifier</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>rest-assured</artifactId>

<version>2.5.0</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>spring-mock-mvc</artifactId>

<version>2.5.0</version>

<scope>compile</scope>

</dependency>

</dependencies>

</plugin>

<dependencies>

<!-- all dependencies -->

<!-- you can exclude rest-assured from spring-cloud-contract-verifier -->

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>rest-assured</artifactId>

<version>2.5.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>spring-mock-mvc</artifactId>

<version>2.5.0</version>

<scope>test</scope>

</dependency>

</dependencies>

这样，插件会自动看到classpath中存在Rest Assured 3.x，并相应地修改了导入。

### **Maven的90.13.3快照版本**

对于Snapshot和Milestone版本，必须将以下部分添加到pom.xml中，如下所示：

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository></repositories><pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository></pluginRepositories>

### **90.13.4添加存根**

默认情况下，Spring Cloud Contract验证程序正在src/test/resources/contracts目录中寻找存根。包含存根定义的目录被视为类名，每个存根定义均被视为单个测试。我们假定它至少包含一个目录用作测试类名称。如果嵌套目录有多个级别，则使用除最后一个嵌套目录之外的所有目录作为包名。例如，具有以下结构：

src/test/resources/contracts/myservice/shouldCreateUser.groovy

src/test/resources/contracts/myservice/shouldReturnUser.groovy

Spring Cloud Contract验证程序使用两种方法创建名为defaultBasePackage.MyService的测试类

* shouldCreateUser()
* shouldReturnUser()

### **90.13.5运行插件**

插件目标generateTests被分配为在称为generate-test-sources的阶段中被调用。如果您希望它成为构建过程的一部分，则无需执行任何操作。如果只想生成测试，请调用generateTests目标。

### **90.13.6配置插件**

要更改默认配置，只需在插件定义或execution定义中添加configuration部分，如下所示：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>convert</goal>

<goal>generateStubs</goal>

<goal>generateTests</goal>

</goals>

</execution>

</executions>

<configuration>

<basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackageForTests>

<baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpec</baseClassForTests>

</configuration></plugin>

### **90.13.7配置选项**

* **testMode**：定义验收测试的模式。默认情况下，模式是MockMvc，它基于Spring的MockMvc。对于真实的HTTP调用，也可以将其更改为**WebTestClient**，**JaxRsClient**或 **Explicit**。
* **basePackageForTests**：指定所有生成的测试的基本软件包。如果未设置，则从baseClassForTests’s package and from `packageWithBaseClasses中选取值。如果这些值均未设置，则该值设置为org.springframework.cloud.contract.verifier.tests。
* **ruleClassForTests**：指定应添加到生成的测试类的规则。
* **baseClassForTests**：为所有生成的测试创建基类。默认情况下，如果使用Spock类，则该类为spock.lang.Specification。
* **contractDirectory**：指定一个目录，其中包含用GroovyDSL编写的合同。默认目录为/src/test/resources/contracts。
* **createdTestSourcesDir**：指定应放置从Groovy DSL生成的测试的测试源目录。默认情况下，其值为$buildDir/generated-test-sources/contracts。
* **createdTestResourcesDir**：指定测试资源目录，测试所使用的资源在该目录中生成
* **testFramework**：指定要使用的目标测试框架。当前，Spock，JUnit 4（TestFramework.JUNIT）和JUnit 5受支持，而JUnit 4是默认框架。
* **packageWithBaseClasses**：定义所有基类所在的包。此设置优先于**baseClassForTests**。约定是这样的：如果您在（例如）src/test/resources/contract/foo/bar/baz/下拥有合同，并将packageWithBaseClasses属性的值设置为com.example.base，则Spring Cloud Contract Verifier假定存在一个com.example.base包下的BarBazBase类。换句话说，系统将获取包的最后两个部分（如果存在的话），并形成一个带有Base后缀的类。
* **baseClassMappings**：指定提供contractPackageRegex的基类映射的列表，该列表将根据合同所在的包进行检查，而baseClassFQN则映射到匹配的合同的基类的标准名称。例如，如果您在src/test/resources/contract/foo/bar/baz/下有一个合同并映射了属性.\* → com.example.base.BaseClass，则从这些合同生成的测试类将扩展com.example.base.BaseClass。此设置优先于 **packageWithBaseClasses**和**baseClassForTests**。
* **contractProperties**：包含要传递给Spring Cloud Contract组件的属性的映射。这些属性可能由内置或自定义存根下载器使用。

如果要从Maven存储库下载合同定义，则可以使用以下选项：

* **contractDependency**：包含所有打包合同的合同依赖关系。
* **ContractsPath**：具有打包合同的JAR中具体合同的路径。默认值为groupid/artifactid，其中gropuid以斜杠分隔。
* **ContractsMode**：选择将要找到并注册存根的模式
* **deleteStubsAfterTest**：如果设置为false，则不会从临时目录中删除任何下载的合同
* **contractRepositoryUrl**：包含合同的工件的仓库的URL。如果未提供，请使用当前的Maven。
* **contractRepositoryUsername**：用于通过合同连接到仓库的用户名。
* **contractRepositoryPassword**：用于通过合同连接到仓库的密码。
* **contractRepositoryProxyHost**：用于通过合同连接到仓库的代理主机。
* **ContractsRepositoryProxyPort**：用于通过合同连接到仓库的代理端口。

我们仅缓存非快照的显式提供的版本（例如，不会缓存+或1.0.0.BUILD-SNAPSHOT）。默认情况下，此功能处于打开状态。

您可以在下面找到通过插件打开的实验功能列表：

* **convertToYaml**：将所有DSL转换为声明性的YAML格式。当您在Groovy DSL中使用外部库时，这可能非常有用。通过启用此功能（将其设置为true），您将不需要在使用者端添加库依赖项。
* **assertJsonSize**：您可以在生成的测试中检查JSON数组的大小。默认情况下禁用此功能。

### **90.13.8所有测试的单一基类**

在默认的MockMvc中使用Spring Cloud Contract验证程序时，您需要为所有生成的验收测试创建基本规范。在此类中，您需要指向一个端点，该端点应进行验证。

**package** org.mycompany.tests

**import** org.mycompany.ExampleSpringController**import** com.jayway.restassured.module.mockmvc.RestAssuredMockMvc**import** spock.lang.Specification

**class** MvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** ExampleSpringController())

}

}

如果需要，您还可以设置整个上下文。

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;**import** org.junit.Before;**import** org.junit.runner.RunWith;**import** org.springframework.beans.factory.annotation.Autowired;**import** org.springframework.boot.test.context.SpringBootTest;**import** org.springframework.test.context.junit4.SpringRunner;**import** org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT, classes = SomeConfig.class, properties="some=property")**public** **abstract** **class** BaseTestClass {

@Autowired

WebApplicationContext context;

@Before

**public** **void** setup() {

RestAssuredMockMvc.webAppContextSetup(**this**.context);

}

}

如果使用EXPLICIT模式，则可以使用基类类似地初始化整个测试的应用程序，就像在常规集成测试中可能会发现的那样。

**import** io.restassured.RestAssured;**import** org.junit.Before;**import** org.junit.runner.RunWith;**import** org.springframework.beans.factory.annotation.Autowired;**import** org.springframework.boot.test.context.SpringBootTest;**import** org.springframework.boot.web.server.LocalServerPort**import** org.springframework.test.context.junit4.SpringRunner;**import** org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT, classes = SomeConfig.class, properties="some=property")**public** **abstract** **class** BaseTestClass {

@LocalServerPort

**int** port;

@Before

**public** **void** setup() {

RestAssured.baseURI = "http://localhost:" + **this**.port;

}

}

如果使用JAXRSCLIENT模式，则此基类还应包含一个protected WebTarget webTarget字段。目前，测试JAX-RS API的唯一选项是启动web服务器。

### **90.13.9合同的不同基类**

如果合同之间的基类不同，则可以告诉Spring Cloud Contract插件自动生成的测试应扩展哪个类。您有两种选择：

* 遵循约定，提供packageWithBaseClasses
* 通过baseClassMappings提供显式映射

**按照惯例**

约定是这样的：如果您在src/test/resources/contract/foo/bar/baz/下拥有合同，并且将packageWithBaseClasses属性的值设置为com.example.base，则Spring Cloud Contract验证者会假设存在一个BarBazBase类（位于com.example.base包中）。换句话说，系统将获取包的最后两个部分（如果存在的话），并形成一个带有Base后缀的类。此规则优先于**baseClassForTests**。这是一个在contracts闭包中如何工作的示例：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<packageWithBaseClasses>hello</packageWithBaseClasses>

</configuration></plugin>

**通过映射**

您可以将合同包的正则表达式手动映射到匹配合同的基类的完全限定名称。您必须提供一个名为baseClassMappings的列表，该列表由baseClassMapping对象组成，这些对象采用从contractPackageRegex到baseClassFQN的映射。考虑以下示例：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<baseClassForTests>com.example.FooBase</baseClassForTests>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*com.\*</contractPackageRegex>

<baseClassFQN>com.example.TestBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

</configuration></plugin>

假设您在以下两个位置拥有合同：\* src/test/resources/contract/com/ \* src/test/resources/contract/foo/

通过提供baseClassForTests，我们可以进行后备，以防映射未成功。（您也可以提供packageWithBaseClasses作为后备。）这样，从src/test/resources/contract/com/合约生成的测试扩展了com.example.ComBase，而其余测试扩展了com.example.FooBase。

### **90.13.10调用生成的测试**

Spring Cloud Contract Maven插件在名为/generated-test-sources/contractVerifier的目录中生成验证代码，并将该目录附加到testCompile目标。

对于Groovy Spock代码，请使用以下代码：

<plugin>

<groupId>org.codehaus.gmavenplus</groupId>

<artifactId>gmavenplus-plugin</artifactId>

<version>1.5</version>

<executions>

<execution>

<goals>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

<configuration>

<testSources>

<testSource>

<directory>${project.basedir}/src/test/groovy</directory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</testSource>

<testSource>

<directory>${project.build.directory}/generated-test-sources/contractVerifier</directory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</testSource>

</testSources>

</configuration></plugin>

为确保提供方符合已定义的合同，您需要调用mvn generateTest test。

### **90.13.11将存根推送到SCM**

如果您使用SCM存储库保留合同和存根，则可能需要自动化将存根推入存储库的步骤。为此，添加pushStubsToScm目标就足够了。例：

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

*<!-- By default we will not push the stubs back to SCM,*

*you have to explicitly add it as a goal -->*

<goal>pushStubsToScm</goal>

</goals>

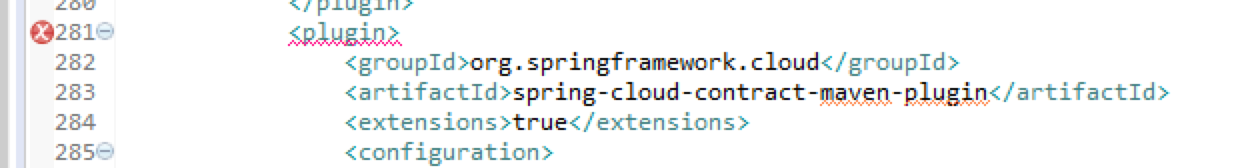
</execution>

</executions></plugin>

在[第96.6节“使用SCM存根下载器”下，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "scm-stub-downloader" \o "96.6使用SCM存根下载器)您可以找到可以通过<configuration><contractProperties>映射，系统属性或环境变量传递的所有可能的配置选项。

### **90.13.12 Maven插件和STS**

如果在使用STS时看到以下异常：



当您单击错误标记时，您应该看到类似以下内容：

plugin:1.1.0.M1:convert:default-convert:process-**test**-resources) org.apache.maven.plugin.PluginExecutionException: Execution default-convert of goal org.springframework.cloud:spring-

cloud-contract-maven-plugin:1.1.0.M1:convert failed. at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginManager.java:145) at

org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at

...

org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by: java.lang.NullPointerException at

org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildContext.hasDelta(EclipseIncrementalBuildContext.java:53) at

org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildContext.java:59) at

为了解决此问题，请在pom.xml中提供以下部分：

<build>

<pluginManagement>

<plugins>

*<!--This plugin's configuration is used to store Eclipse m2e settings*

*only. It has no influence on the Maven build itself. -->*

<plugin>

<groupId>org.eclipse.m2e</groupId>

<artifactId>lifecycle-mapping</artifactId>

<version>1.0.0</version>

<configuration>

<lifecycleMappingMetadata>

<pluginExecutions>

<pluginExecution>

<pluginExecutionFilter>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<versionRange>[1.0,)</versionRange>

<goals>

<goal>convert</goal>

</goals>

</pluginExecutionFilter>

<action>

<execute />

</action>

</pluginExecution>

</pluginExecutions>

</lifecycleMappingMetadata>

</configuration>

</plugin>

</plugins>

</pluginManagement></build>

### **90.13.13 Maven具有Spock测试的插件**

您可以选择[Spock框架](http://spockframework.org/" \t "https://www.springcloud.cc/_top)来使用Maven和Gradle插件来创建和执行自动生成的合同验证测试。但是，尽管Gradle确实很简单，但是在Maven中，您将需要一些附加设置才能使测试正确编译和执行。

首先，您将必须使用诸如[GMavenPlus](https://github.com/groovy/GMavenPlus" \t "https://www.springcloud.cc/_top)插件之类的插件将Groovy添加到您的项目中。在GMavenPlus插件中，您将需要显式设置测试源，包括定义基本测试类的路径和添加了生成的合同测试的路径。请参考以下示例：

<plugin>

<groupId>org.codehaus.gmavenplus</groupId>

<artifactId>gmavenplus-plugin</artifactId>

<version>1.6.1</version>

<executions>

<execution>

<goals>

<goal>compileTests</goal>

<goal>addTestSources</goal>

</goals>

</execution>

</executions>

<configuration>

<testSources>

<testSource>

<directory>${project.basedir}/src/test/groovy</directory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</testSource>

<testSource>

<directory>

${project.basedir}/target/generated-test-sources/contracts/com/example/beer

</directory>

<includes>

<include>\*\*/\*.groovy</include>

<include>\*\*/\*.gvy</include>

</includes>

</testSource>

</testSources>

</configuration>

<dependencies>

<dependency>

<groupId>org.codehaus.groovy</groupId>

<artifactId>groovy-all</artifactId>

<version>2.4.15</version>

<scope>runtime</scope>

<type>pom</type>

</dependency>

</dependencies>

如果您坚持以Spec结尾测试类名称的Spock约定，则还需要调整Maven Surefire插件设置，如以下示例所示：

## **90.14存根和传递依赖项**

Maven和Gradle插件添加了为您创建存根jar的任务。出现的一个问题是，当重用存根时，您可能会错误地导入该存根的所有依赖项。构建Maven工件时，即使您有几个罐子，它们也共享一个pom：

├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar

├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1

├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar

├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1

├── github-webhook-0.0.1.BUILD-SNAPSHOT.jar

├── github-webhook-0.0.1.BUILD-SNAPSHOT.pom

├── github-webhook-0.0.1.BUILD-SNAPSHOT-stubs.jar

├── ...

└── ...

使用这些依赖关系有三种可能性，以使传递依赖关系没有任何问题：

* 将所有应用程序依赖项标记为可选
* 为存根创建一个单独的artifactid
* 排除消费者方面的依赖

**将所有应用程序依赖项标记为可选**

如果在github-webhook应用程序中将所有依赖项标记为可选，则在另一个应用程序中包含github-webhook存根时（或当Stub Runner下载了该依赖项时），则因为所有依赖项是可选的，它们将不会下载。

**为存根创建单独的**artifactid

如果您创建单独的artifactid，则可以按照您希望的任何方式进行设置。例如，您可能决定完全没有依赖项。

**排除消费者方面的依赖**

作为使用者，如果将存根依赖项添加到类路径中，则可以显式排除不需要的依赖项。

## **90.15场景**

您可以使用Spring Cloud Contract验证程序处理方案。您需要做的就是在创建合同时遵守正确的命名约定。约定要求在订货号后加上下划线。无论您使用的是YAML还是Groovy，这都会起作用。例：

my\_contracts\_dir\

scenario1\

1\_login.groovy

2\_showCart.groovy

3\_logout.groovy

这样的树使Spring Cloud Contract验证程序生成名称为scenario1的WireMock场景，并执行以下三个步骤：

1. 登录为Started的登录指向...
2. 标记为Step1的showCart指向...
3. 标记为Step2的注销将关闭方案。

有关WireMock方案的更多详细信息，[请](https://wiremock.org/docs/stateful-behaviour/" \t "https://www.springcloud.cc/_top)参见 [https://wiremock.org/docs/stateful-behaviour/](https://wiremock.org/docs/stateful-behaviour/" \t "https://www.springcloud.cc/_top)

Spring Cloud Contract验证程序还会生成具有保证执行顺序的测试。

## **90.16 Docker项目**

我们正在发布一个springcloud/spring-cloud-contract Docker映像，其中包含一个项目，该项目将生成测试并针对运行中的应用程序以EXPLICIT模式执行测试。

|  |
| --- |
| IMG_539 |
| EXPLICIT模式意味着从合同生成的测试将发送真实请求，而不是模拟请求。 |

### **90.16.1 Maven，JAR和二进制存储的简短介绍**

由于Docker映像可由非JVM项目使用，因此最好解释Spring Cloud Contract打包默认值背后的基本术语。

以下定义的一部分来自[Maven词汇表](https://maven.apache.org/glossary.html" \t "https://www.springcloud.cc/_top)

* Project：Maven根据项目进行思考。您将构建的所有内容都是项目。这些项目遵循定义明确的“项目对象模型”。项目可以依赖于其他项目，在这种情况下，后者称为“依赖项”。一个项目可能与几个子项目一致，但是这些子项目仍被视为项目。
* Artifact：工件是项目产生或使用的东西。Maven为项目产生的工件示例包括：JAR，源和二进制发行版。每个工件都由组ID和组内唯一的工件ID唯一标识。
* JAR：JAR代表Java ARchive。这是一种基于ZIP文件格式的格式。Spring Cloud Contract将合同和生成的存根打包到JAR文件中。
* GroupId：组ID是项目的通用唯一标识符。尽管这通常只是项目名称（例如commons-collections），但使用完全合格的软件包名称将其与具有类似名称的其他项目（例如org.apache.maven）区分开来会很有帮助。通常，GroupId发布到工件管理器时，将使用斜杠分隔并构成URL的一部分。例如，组ID com.example和工件ID application为/com/example/application/。
* Classifier：Maven依赖性表示法如下所示：groupId:artifactId:version:classifier。分类器是传递给依赖项的附加后缀。例如stubs，sources。相同的依存关系，例如com.example:application可能会产生多个因分类器而彼此不同的工件。
* Artifact manager：生成二进制文件/源代码/软件包时，希望它们可供其他人下载/引用或重用。在JVM的世界中，这些工件将是JAR，对于Ruby而言，它们是宝石，对于Docker，则是Docker映像。您可以将这些工件存储在管理器中。此类管理器的示例可以是[Artifactory](https://jfrog.com/artifactory/" \t "https://www.springcloud.cc/_top) 或[Nexus](https://www.sonatype.org/nexus/" \t "https://www.springcloud.cc/_top)。

### **90.16.2工作原理**

该图像在/contracts文件夹下搜索合同。运行测试的输出将在/spring-cloud-contract/build文件夹下可用（对于调试目的很有用）。

您安装合同，传递环境变量就足够了，该映像将：

* 生成合同测试
* 针对提供的URL执行测试
* 生成[WireMock](https://github.com/tomakehurst/wiremock" \t "https://www.springcloud.cc/_top)存根
* （可选-默认情况下处于启用状态）将存根发布到Artifact Manager

#### **环境变量**

Docker映像需要一些环境变量以指向您正在运行的应用程序，工件管理器实例等。

* PROJECT\_GROUP-您的项目的组ID。默认为com.example
* PROJECT\_VERSION-您项目的版本。默认为0.0.1-SNAPSHOT
* PROJECT\_NAME-工件ID。默认为example
* REPO\_WITH\_BINARIES\_URL-工件管理器的URL。默认值为[http://localhost:8081/artifactory/libs-release-local](http://localhost:8081/artifactory/libs-release-local" \t "https://www.springcloud.cc/_top)，这是本地运行的[Artifactory](https://jfrog.com/artifactory/" \t "https://www.springcloud.cc/_top)的默认URL
* REPO\_WITH\_BINARIES\_USERNAME-伪影管理器受保护时（可选）的用户名
* REPO\_WITH\_BINARIES\_PASSWORD-安全工件管理器时的密码（可选）
* PUBLISH\_ARTIFACTS-如果设置为true，则会将工件发布到二进制存储。默认为true。

当合同位于外部存储库中时，将使用这些环境变量。要启用此功能，必须设置EXTERNAL\_CONTRACTS\_ARTIFACT\_ID环境变量。

* EXTERNAL\_CONTRACTS\_GROUP\_ID-带有合同的项目的组ID。默认为com.example
* EXTERNAL\_CONTRACTS\_ARTIFACT\_ID-带有合同的项目的工件ID。
* EXTERNAL\_CONTRACTS\_CLASSIFIER-带有合同的项目分类。默认为空
* EXTERNAL\_CONTRACTS\_VERSION-带有合同的项目版本。默认值为+，相当于选择最新的
* EXTERNAL\_CONTRACTS\_REPO\_WITH\_BINARIES\_URL-工件管理器的URL。默认值为REPO\_WITH\_BINARIES\_URL env var。如果未设置，则默认为[http://localhost:8081/artifactory/libs-release-local](http://localhost:8081/artifactory/libs-release-local" \t "https://www.springcloud.cc/_top)，这是在本地运行的[Artifactory](https://jfrog.com/artifactory/" \t "https://www.springcloud.cc/_top)的默认URL
* EXTERNAL\_CONTRACTS\_PATH-包含合同的项目内给定项目的合同路径。默认为斜线分隔的EXTERNAL\_CONTRACTS\_GROUP\_ID与/和EXTERNAL\_CONTRACTS\_ARTIFACT\_ID串联在一起。例如，对于组ID foo.bar和工件ID baz，将导致foo/bar/baz合同路径。
* EXTERNAL\_CONTRACTS\_WORK\_OFFLINE-如果设置为true，则将从容器的.m2中检索带有合同的工件。将本地.m2挂载为容器的/root/.m2路径上可用的卷。您不能同时设置EXTERNAL\_CONTRACTS\_WORK\_OFFLINE和EXTERNAL\_CONTRACTS\_REPO\_WITH\_BINARIES\_URL。

执行测试时使用以下环境变量：

* APPLICATION\_BASE\_URL-应该对其执行测试的URL。请记住，必须可以从Docker容器访问它（例如localhost将不起作用）
* APPLICATION\_USERNAME-（可选）用于对应用程序进行基本身份验证的用户名
* APPLICATION\_PASSWORD-（可选）用于对应用程序进行基本身份验证的密码

### **90.16.3使用示例**

让我们看一个简单的MVC应用程序

$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs

$ **cd** bookstore

合同位于/contracts文件夹下。

### **90.16.4服务器端（nodejs）**

因为我们要运行测试，所以我们可以执行：

$ npm **test**

但是，出于学习目的，让我们将其分为几部分：

*# Stop docker infra (nodejs, artifactory)*

$ ./stop\_infra.sh*# Start docker infra (nodejs, artifactory)*

$ ./setup\_infra.sh

*# Kill & Run app*

$ pkill -f "node app"

$ nohup node app &

*# Prepare environment variables*

$ SC\_CONTRACT\_DOCKER\_VERSION="..."

$ APP\_IP="192.168.0.100"

$ APP\_PORT="3000"

$ ARTIFACTORY\_PORT="8081"

$ APPLICATION\_BASE\_URL="http://${APP\_IP}:${APP\_PORT}"

$ ARTIFACTORY\_URL="http://${APP\_IP}:${ARTIFACTORY\_PORT}/artifactory/libs-release-local"

$ CURRENT\_DIR="$( pwd )"

$ CURRENT\_FOLDER\_NAME=${PWD*##\*/}*

$ PROJECT\_VERSION="0.0.1.RELEASE"

*# Execute contract tests*

$ docker run --rm -e "APPLICATION\_BASE\_URL=${APPLICATION\_BASE\_URL}" -e "PUBLISH\_ARTIFACTS=true" -e "PROJECT\_NAME=${CURRENT\_FOLDER\_NAME}" -e "REPO\_WITH\_BINARIES\_URL=${ARTIFACTORY\_URL}" -e "PROJECT\_VERSION=${PROJECT\_VERSION}" -v "${CURRENT\_DIR}/contracts/:/contracts:ro" -v "${CURRENT\_DIR}/node\_modules/spring-cloud-contract/output:/spring-cloud-contract-output/" springcloud/spring-cloud-contract:"${SC\_CONTRACT\_DOCKER\_VERSION}"

*# Kill app*

$ pkill -f "node app"

将会发生的是通过bash脚本：

* 基础设施将被建立（MongoDb，Artifactory）。在现实生活中，您只需运行带有模拟数据库的NodeJS应用程序。在此示例中，我们想展示如何立即受益于Spring Cloud Contract。

由于这些限制，合同也代表了有状态的情况

* + 第一个请求是POST，它导致数据被插入到数据库中
  + 第二个请求是GET，它返回带有一个先前插入的元素的数据列表
* NodeJS应用程序将启动（在端口3000上）

合同测试将通过Docker生成，并且测试将针对正在运行的应用程序执行

* + 合同将从/contracts文件夹中提取。
  + 测试执行的输出在node\_modules/spring-cloud-contract/output下可用。
* 存根将被上传到Artifactory。您可以在[http：// localhost：8081 / artifactory / libs-release-local / com / example / bookstore / 0.0.1.RELEASE /中检出它们](http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/" \t "https://www.springcloud.cc/_top)。存根将在这里[http：// localhost：8081 / artifactory / libs-release-local / com / example / bookstore / 0.0.1.RELEASE / bookstore-0.0.1.RELEASE-stubs.jar](http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar" \t "https://www.springcloud.cc/_top)。

要查看客户端的外观，请查看[第92.9节“ Stub Runner Docker”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stubrunner-docker" \o "92.9 Stub Runner Docker)部分。

## **91. Spring Cloud Contract验证者消息**

Spring Cloud Contract验证程序使您可以验证使用消息传递作为通信手段的应用程序。本文档中显示的所有集成都可以与Spring一起使用，但是您也可以创建自己的一个并使用它。

## **91.1集成**

您可以使用以下四种集成配置之一：

* 阿帕奇骆驼
* Spring Integration
* Spring Cloud Stream
* Spring AMQP

由于我们使用Spring Boot，因此，如果您已将这些库之一添加到类路径中，则会自动设置所有消息传递配置。

|  |  |
| --- | --- |
| IMG_540 | **重要** |
| 请记住将@AutoConfigureMessageVerifier放在生成的测试的基类上。否则，Spring Cloud Contract验证程序的消息传递部分将不起作用。 |

|  |  |
| --- | --- |
| IMG_541 | **重要** |
| 如果要使用Spring Cloud Stream，请记住对org.springframework.cloud:spring-cloud-stream-test-support添加依赖项，如下所示： |

**Maven.**

<dependency>

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

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope></dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-stream-test-support"

## **91.2手动集成测试**

测试使用的主界面为org.springframework.cloud.contract.verifier.messaging.MessageVerifier。它定义了如何发送和接收消息。您可以创建自己的实现以实现相同的目标。

在测试中，您可以插入ContractVerifierMessageExchange以发送和接收遵循合同的消息。然后将@AutoConfigureMessageVerifier添加到测试中。这是一个例子：

@RunWith(SpringTestRunner.class)@SpringBootTest@AutoConfigureMessageVerifier**public** **static** **class** MessagingContractTests {

@Autowired

**private** MessageVerifier verifier;

...

}

|  |
| --- |
| IMG_542 |
| 如果您的测试也需要存根，则@AutoConfigureStubRunner包括消息传递配置，因此您只需要一个注释。 |

## **91.3发布者方测试生成**

DSL中包含input或outputMessage部分会导致在发布者方面创建测试。默认情况下，将创建JUnit 4测试。但是，也可以创建JUnit 5或Spock测试。

我们应考虑3种主要情况：

* 方案1：没有输入消息会生成输出消息。输出消息由应用程序内部的组件（例如，调度程序）触发。
* 方案2：输入消息触发输出消息。
* 方案3：输入消息已被使用，没有输出消息。

|  |  |
| --- | --- |
| IMG_543 | **重要** |
| 对于不同的消息传递实现，传递给messageFrom或sentTo的目的地可能具有不同的含义。对于**流**和**集成**，首先**将**其解析为频道的destination。然后，如果没有这样的destination，则将其解析为频道名称。对于**Camel来说**，这是一个确定的组成部分（例如jms）。 |

### **91.3.1方案1：无输入消息**

对于给定的合同：

**Groovy DSL。**

def contractDsl = Contract.make {

label 'some\_label'

input {

triggeredBy('bookReturnedTriggered()')

}

outputMessage {

sentTo('activemq:output')

body('''{ "bookName" : "foo" }''')

headers {

header('BOOK-NAME', 'foo')

messagingContentType(applicationJson())

}

}

}

**YAML。**

label: some\_label

input:

triggeredBy: bookReturnedTriggered

outputMessage:

sentTo: activemq:output

body:

bookName: foo

headers:

BOOK-NAME: foo

contentType: application/json

创建了以下JUnit测试：

''' *// when:*

bookReturnedTriggered();

*// then:*

ContractVerifierMessage response = contractVerifierMessaging.receive("activemq:output");

assertThat(response).isNotNull();

assertThat(response.getHeader("BOOK-NAME")).isNotNull();

assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");

assertThat(response.getHeader("contentType")).isNotNull();

assertThat(response.getHeader("contentType").toString()).isEqualTo("application/json");

*// and:*

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));

assertThatJson(parsedJson).field("bookName").isEqualTo("foo");'''

然后将创建以下Spock测试：

''' when:

bookReturnedTriggered()

then:

ContractVerifierMessage response = contractVerifierMessaging.receive('activemq:output')

assert response != null

response.getHeader('BOOK-NAME')?.toString() == 'foo'

response.getHeader('contentType')?.toString() == 'application/json'

and:

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("foo")

'''

### **91.3.2场景2：由输入触发的输出**

对于给定的合同：

**Groovy DSL。**

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:input')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('jms:output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML。**

label: some\_label

input:

messageFrom: jms:input

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

outputMessage:

sentTo: jms:output

body:

bookName: foo

headers:

BOOK-NAME: foo

创建了以下JUnit测试：

'''*// given:*

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

"{\\"bookName\\":\\"foo\\"}"

, headers()

.header("sample", "header"));

*// when:*

contractVerifierMessaging.send(inputMessage, "jms:input");

*// then:*

ContractVerifierMessage response = contractVerifierMessaging.receive("jms:output");

assertThat(response).isNotNull();

assertThat(response.getHeader("BOOK-NAME")).isNotNull();

assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");*// and:*

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));

assertThatJson(parsedJson).field("bookName").isEqualTo("foo");'''

然后将创建以下Spock测试：

"""\given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

'''{"bookName":"foo"}''',

['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:input')

then:

ContractVerifierMessage response = contractVerifierMessaging.receive('jms:output')

assert response !- null

response.getHeader('BOOK-NAME')?.toString() == 'foo'

and:

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("foo")"""

### **91.3.3方案3：无输出消息**

对于给定的合同：

**Groovy DSL。**

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:delete')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

assertThat('bookWasDeleted()')

}

}

**YAML。**

label: some\_label

input:

messageFrom: jms:delete

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

assertThat: bookWasDeleted()

创建了以下JUnit测试：

'''*// given:*

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

"{\\"bookName\\":\\"foo\\"}"

, headers()

.header("sample", "header"));

*// when:*

contractVerifierMessaging.send(inputMessage, "jms:delete");

*// then:*

bookWasDeleted();'''

然后将创建以下Spock测试：

'''given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

\'\'\'{"bookName":"foo"}\'\'\', ['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:

noExceptionThrown()

bookWasDeleted()'''

## **91.4消费者存根生成**

与HTTP部分不同，在消息传递中，我们需要使用存根在{{JAR}}中发布Groovy DSL。然后在用户端对其进行解析，并创建正确的存根路由。

有关更多信息，请参见[第93章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stub-runner-for-messaging" \o "93.消息传递存根运行器)*[消息传递的Stub Runner](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stub-runner-for-messaging" \o "93.消息传递存根运行器)*部分。

**Maven.**

<dependencies>

<dependency>

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

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency></dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Greenwich.BUILD-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

**Gradle.**

ext {

contractsDir = file("mappings")

stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")

}

*// Automatically added by plugin:// copyContracts - copies contracts to the output folder from which JAR will be created// verifierStubsJar - JAR with a provided stub suffix// the presented publication is also added by the plugin but you can modify it as you wish*

publishing {

publications {

stubs(MavenPublication) {

artifactId "${project.name}-stubs"

artifact verifierStubsJar

}

}

}

## **92. Spring Cloud Contract Stub Runner**

使用Spring Cloud Contract验证程序时，您可能会遇到的问题之一是将生成的WireMock JSON存根从服务器端传递到客户端（或传递到各种客户端）。在客户端的消息传递方面也发生了同样的事情。

复制JSON文件并设置客户端手动进行消息传递是不可能的。这就是为什么我们引入了Spring Cloud Contract Stub Runner。它可以自动为您下载并运行存根。

## **92.1快照版本**

将其他快照存储库添加到您的build.gradle文件中以使用快照版本，快照版本在每次成功构建后都会自动上载：

**Maven.**

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository></repositories><pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository></pluginRepositories>

**Gradle.**

*/\**

*We need to use the [buildscript {}] section when we have to modify*

*the classpath for the plugins. If that's not the case this section*

*can be skipped.*

*If you don't need to modify the classpath (e.g. add a Pact dependency),*

*then you can just set the [pluginManagement {}] section in [settings.gradle] file.*

*// settings.gradle*

*pluginManagement {*

*repositories {*

*// for snapshots*

*maven {url "https://repo.spring.io/snapshot"}*

*// for milestones*

*maven {url "https://repo.spring.io/milestone"}*

*// for GA versions*

*gradlePluginPortal()*

*}*

*}*

*\*/*

buildscript {

repositories {

mavenCentral()

mavenLocal()

maven { url "https://repo.spring.io/snapshot" }

maven { url "https://repo.spring.io/milestone" }

maven { url "https://repo.spring.io/release" }

}

## **92.2将存根发布为JAR**

最简单的方法是集中存根的保存方式。例如，您可以将它们作为罐子保存在Maven存储库中。

|  |
| --- |
| IMG_544 |
| 对于Maven和Gradle而言，该设置均可使用。但是，您可以根据需要自定义它。 |

**Maven.**

*<!-- First disable the default jar setup in the properties section --><!-- we don't want the verifier to do a jar for us -->*<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>

*<!-- Next add the assembly plugin to your build --><!-- we want the assembly plugin to generate the JAR -->*<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>stub</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<inherited>false</inherited>

<configuration>

<attach>true</attach>

<descriptors>

$../../../../src/assembly/stub.xml

</descriptors>

</configuration>

</execution>

</executions></plugin>

*<!-- Finally setup your assembly. Below you can find the contents of src/main/assembly/stub.xml -->*<assembly

xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>stubs</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>src/main/java</directory>

<outputDirectory>/</outputDirectory>

<includes>

<include>\*\*com/example/model/\*.\*</include>

</includes>

</fileSet>

<fileSet>

<directory>${project.build.directory}/classes</directory>

<outputDirectory>/</outputDirectory>

<includes>

<include>\*\*com/example/model/\*.\*</include>

</includes>

</fileSet>

<fileSet>

<directory>${project.build.directory}/snippets/stubs</directory>

<outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDirectory>

<includes>

<include>\*\*/\*</include>

</includes>

</fileSet>

<fileSet>

<directory>$../../../../src/test/resources/contracts</directory>

<outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputDirectory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</fileSet>

</fileSets></assembly>

**Gradle.**

ext {

contractsDir = file("mappings")

stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")

}

*// Automatically added by plugin:// copyContracts - copies contracts to the output folder from which JAR will be created// verifierStubsJar - JAR with a provided stub suffix// the presented publication is also added by the plugin but you can modify it as you wish*

publishing {

publications {

stubs(MavenPublication) {

artifactId "${project.name}-stubs"

artifact verifierStubsJar

}

}

}

## **92.3 Stub Runner核心**

为服务合作者运行存根。将存根视为服务合同可将存根运行器用作 [消费者驱动Contracts的实现](https://martinfowler.com/articles/consumerDrivenContracts.html" \t "https://www.springcloud.cc/_top)。

Stub Runner允许您自动下载提供的依赖项的存根（或从类路径中选择它们），为它们启动WireMock服务器，并使用正确的存根定义来提供它们。对于消息传递，定义了特殊的存根路由。

### **92.3.1检索存根**

您可以选择以下获取存根的选项

* 基于醚的解决方案，可从Artifactory / Nexus下载带有存根的JAR
* 类路径扫描解决方案，可通过模式搜索类路径以检索存根
* 编写自己的org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder实现以进行完全自定义

后一个示例在“ [自定义Stub Runner”](https://www.springcloud.cc/spring-cloud-greenwich.html)部分中进行了描述。

#### **存根下载**

您可以通过stubsMode开关控制存根下载。它从StubRunnerProperties.StubsMode枚举中选择值。您可以使用以下选项

* StubRunnerProperties.StubsMode.CLASSPATH（默认值）-将从类路径中选择存根
* StubRunnerProperties.StubsMode.LOCAL-将从本地存储区中选择存根（例如.m2）
* StubRunnerProperties.StubsMode.REMOTE-将从远程位置选择存根

例：

@AutoConfigureStubRunner(repositoryRoot="https://foo.bar", ids = "com.example:beer-api-producer:+:stubs:8095", stubsMode = StubRunnerProperties.StubsMode.LOCAL)

#### **类路径扫描**

如果将stubsMode属性设置为StubRunnerProperties.StubsMode.CLASSPATH（或由于默认值CLASSPATH而未设置任何内容），则将扫描类路径。让我们看下面的例子：

@AutoConfigureStubRunner(ids = {

"com.example:beer-api-producer:+:stubs:8095",

"com.example.foo:bar:1.0.0:superstubs:8096"

})

如果您已将依赖项添加到类路径中

**Maven.**

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-api-producer-restdocs</artifactId>

<classifier>stubs</classifier>

<version>0.0.1-SNAPSHOT</version>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>\*</groupId>

<artifactId>\*</artifactId>

</exclusion>

</exclusions></dependency><dependency>

<groupId>com.example.foo</groupId>

<artifactId>bar</artifactId>

<classifier>superstubs</classifier>

<version>1.0.0</version>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>\*</groupId>

<artifactId>\*</artifactId>

</exclusion>

</exclusions></dependency>

**Gradle.**

testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {

transitive = false

}

testCompile("com.example.foo:bar:1.0.0:superstubs") {

transitive = false

}

然后，将扫描您的类路径上的以下位置。对于com.example:beer-api-producer-restdocs

* /META-INF/com.example/beer-api-producer-restdocs/ **\* /** \*。
* /contracts/com.example/beer-api-producer-restdocs/ **\* /** \*。
* /mappings/com.example/beer-api-producer-restdocs/ **\* /** \*。

和com.example.foo:bar

* /META-INF/com.example.foo/bar/ **\* /** \*。
* /contracts/com.example.foo/bar/ **\* /** \*。
* /mappings/com.example.foo/bar/ **\* /** \*。

|  |
| --- |
| IMG_545 |
| 如您所见，打包生产者存根时必须显式提供组和工件ID。 |

生产者将像这样设置合同：

└── src

└── **test**

└── resources

└── contracts

   └── com.example

      └── beer-api-producer-restdocs

      └── nested

      └── contract3.groovy

要实现正确的存根包装。

或使用[Maven assembly插件](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/2.1.x/producer_with_restdocs/pom.xml" \t "https://www.springcloud.cc/_top)或 [Gradle Jar](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/2.1.x/producer_with_restdocs/build.gradle" \t "https://www.springcloud.cc/_top)任务，您必须在存根jar中创建以下结构。

└── META-INF

└── com.example

└── beer-api-producer-restdocs

└── 2.0.0

├── contracts

│   └── nested

   │ └── contract2.groovy

   └── mappings

   └── mapping.json

通过维护这种结构，可以扫描类路径，而无需下载工件即可从消息传递/ HTTP存根中受益。

#### **配置HTTP服务器存根**

Stub Runner具有HttpServerStub的概念，该概念抽象了HTTP服务器的底层具体实现（例如，WireMock是实现之一）。有时，您需要对存根服务器执行一些其他调整，这对于给定的实现而言是具体的。为此，Stub Runner为您提供了httpServerStubConfigurer属性，该属性在批注JUnit规则中可用，并且可以通过系统属性进行访问，您可以在其中提供org.springframework.cloud.contract.stubrunner.HttpServerStubConfigurer接口的实现。这些实现可以更改给定HTTP服务器存根的配置文件。

Spring Cloud Contract Stub Runner带有一个可以扩展的实现，适用于WireMock-org.springframework.cloud.contract.stubrunner.provider.wiremock.WireMockHttpServerStubConfigurer。在configure方法中，您可以为给定的存根提供自己的自定义配置。用例可能是在HTTPs端口上为给定的工件ID启动WireMock。例：

**WireMockHttpServerStubConfigurer实现。**

@CompileStatic**static** **class** HttpsForFraudDetection **extends** WireMockHttpServerStubConfigurer {

**private** **static** **final** Log log = LogFactory.getLog(HttpsForFraudDetection)

@Override

WireMockConfiguration configure(WireMockConfiguration httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {

**if** (httpServerStubConfiguration.stubConfiguration.artifactId == "fraudDetectionServer") {

**int** httpsPort = SocketUtils.findAvailableTcpPort()

log.info("Will set HTTPs port [" + httpsPort + "] for fraud detection server")

**return** httpStubConfiguration

.httpsPort(httpsPort)

}

**return** httpStubConfiguration

}

}

然后，您可以通过注释重用它

@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",

httpServerStubConfigurer = HttpsForFraudDetection)

只要找到一个https端口，它将优先于http端口。

### **92.3.2运行存根**

#### **使用主应用程序运行**

您可以为主类设置以下选项：

-c, --classifier Suffix **for** the jar containing stubs (e.

g. 'stubs' **if** the stub jar would

have a 'stubs' classifier **for** stubs:

foobar-stubs ). Defaults to 'stubs'

(**default**: stubs)

--maxPort, --maxp <Integer> Maximum port value to be assigned to

the WireMock instance. Defaults to

15000 (**default**: 15000)

--minPort, --minp <Integer> Minimum port value to be assigned to

the WireMock instance. Defaults to

10000 (**default**: 10000)

-p, --password Password to user when connecting to

repository

--phost, --proxyHost Proxy host to use **for** repository

requests

--pport, --proxyPort [Integer] Proxy port to use **for** repository

requests

-r, --root Location of a Jar containing server

where you keep your stubs (e.g. http:

*//nexus.*

net/content/repositories/repository)

-s, --stubs Comma separated list of Ivy

representation of jars with stubs.

Eg. groupid:artifactid1,groupid2:

artifactid2:classifier

--sm, --stubsMode Stubs mode to be used. Acceptable values

[CLASSPATH, LOCAL, REMOTE]

-u, --username Username to user when connecting to

repository

#### **HTTP存根**

存根在JSON文档中定义，其语法在[WireMock文档中](http://wiremock.org/stubbing.html" \t "https://www.springcloud.cc/_top)定义

例：

{

"request": {

"method": "GET",

"url": "/ping"

},

"response": {

"status": 200,

"body": "pong",

"headers": {

"Content-Type": "text/plain"

}

}

}

#### **查看注册的映射**

每个存根协作者都会在\_\_/admin/端点下公开已定义映射的列表。

您还可以使用mappingsOutputFolder属性将映射转储到文件。对于基于注释的方法，它看起来像这样

@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",

mappingsOutputFolder = "target/outputmappings/")

对于这样的JUnit方法：

@ClassRule @Shared StubRunnerRule rule = **new** StubRunnerRule()

.repoRoot("http://some\_url")

.downloadStub("a.b.c", "loanIssuance")

.downloadStub("a.b.c:fraudDetectionServer")

.withMappingsOutputFolder("target/outputmappings")

然后，如果您检出文件夹target/outputmappings，则会看到以下结构

.

├── fraudDetectionServer\_13705

└── loanIssuance\_12255

这意味着注册了两个存根。fraudDetectionServer在端口13705上注册，loanIssuance在端口12255上注册。如果我们看一下其中一个文件，我们将看到（对于WireMock）可用于给定服务器的映射：

[**{**

"id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"**,**

"request" : **{**

"url" : "/name"**,**

"method" : "GET"

**},**

"response" : **{**

"status" : 200**,**

"body" : "fraudDetectionServer"

**},**

"uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"**},**

...**]**

#### **信息存根**

根据提供的Stub Runner依赖性和DSL，将自动设置消息传递路由。

## **92.4 Stub Runner JUnit规则和Stub Runner JUnit5扩展**

Stub Runner带有JUnit规则，因此您可以很容易地下载和运行给定组和工件ID的存根：

@ClassRule**public** **static** StubRunnerRule rule = **new** StubRunnerRule().repoRoot(repoRoot())

.stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.downloadStub("org.springframework.cloud.contract.verifier.stubs",

"loanIssuance")

.downloadStub(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer");

@BeforeClass@AfterClass**public** **static** **void** setupProps() {

System.clearProperty("stubrunner.repository.root");

System.clearProperty("stubrunner.classifier");

}

JUnit 5也有一个StubRunnerExtension。StubRunnerRule和StubRunnerExtension的工作方式非常相似。执行完规则/扩展名后，Stub Runner将连接到Maven存储库，并针对给定的依赖项列表尝试执行以下操作：

* 下载它们
* 本地缓存它们
* 将它们解压缩到一个临时文件夹
* 从提供的端口范围/提供的端口中为随机端口上的每个Maven依赖项启动WireMock服务器
* 向WireMock服务器提供有效的WireMock定义的所有JSON文件
* 也可以发送消息（记住要通过MessageVerifier接口的实现）

Stub Runner使用[Eclipse Aether](https://wiki.eclipse.org/Aether" \t "https://www.springcloud.cc/_top)机制下载Maven依赖项。查看他们的[文档](https://wiki.eclipse.org/Aether" \t "https://www.springcloud.cc/_top)以获取更多信息。

由于StubRunnerRule和StubRunnerExtension实现了StubFinder，因此它们使您可以找到已启动的存根：

**package** org.springframework.cloud.contract.stubrunner;

**import** java.net.URL;**import** java.util.Collection;**import** java.util.Map;

**import** org.springframework.cloud.contract.spec.Contract;

**/\*\***

**\* Contract for finding registered stubs.**

**\***

**\* @author Marcin Grzejszczak**

**\*/public** **interface** StubFinder **extends** StubTrigger {

**/\*\***

**\* For the given groupId and artifactId tries to find the matching URL of the running**

**\* stub.**

**\* @param groupId - might be null. In that case a search only via artifactId takes**

**\* place**

**\* @param artifactId - artifact id of the stub**

**\* @return URL of a running stub or throws exception if not found**

**\* @throws StubNotFoundException in case of not finding a stub**

**\*/**

URL findStubUrl(String groupId, String artifactId) **throws** StubNotFoundException;

**/\*\***

**\* For the given Ivy notation {@code [groupId]:artifactId:[version]:[classifier]}**

**\* tries to find the matching URL of the running stub. You can also pass only**

**\* {@code artifactId}.**

**\* @param ivyNotation - Ivy representation of the Maven artifact**

**\* @return URL of a running stub or throws exception if not found**

**\* @throws StubNotFoundException in case of not finding a stub**

**\*/**

URL findStubUrl(String ivyNotation) **throws** StubNotFoundException;

**/\*\***

**\* @return all running stubs**

**\*/**

RunningStubs findAllRunningStubs();

**/\*\***

**\* @return the list of Contracts**

**\*/**

Map<StubConfiguration, Collection<Contract>> getContracts();

}

Spock测试中的用法示例：

@ClassRule@Shared

StubRunnerRule rule = **new** StubRunnerRule()

.stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())

.downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")

.withMappingsOutputFolder("target/outputmappingsforrule")

def 'should start WireMock servers'() {

expect: 'WireMocks are running'

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null

rule.findStubUrl('loanIssuance') != null

rule.findStubUrl('loanIssuance') == rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null

and:

rule.findAllRunningStubs().isPresent('loanIssuance')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')

and: 'Stubs were registered'

"${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

}

def 'should output mappings to output folder'() {

when:

def url = rule.findStubUrl('fraudDetectionServer')

then:

**new** File("target/outputmappingsforrule", "fraudDetectionServer\_${url.port}").exists()

}

JUnit测试中的用法示例：

@Test

**public** **void** should\_start\_wiremock\_servers() **throws** Exception {

*// expect: 'WireMocks are running'*

then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",

"loanIssuance")).isNotNull();

then(rule.findStubUrl("loanIssuance")).isNotNull();

then(rule.findStubUrl("loanIssuance")).isEqualTo(rule.findStubUrl(

"org.springframework.cloud.contract.verifier.stubs", "loanIssuance"));

then(rule.findStubUrl(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer"))

.isNotNull();

*// and:*

then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();

then(rule.findAllRunningStubs().isPresent(

"org.springframework.cloud.contract.verifier.stubs",

"fraudDetectionServer")).isTrue();

then(rule.findAllRunningStubs().isPresent(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer"))

.isTrue();

*// and: 'Stubs were registered'*

then(httpGet(rule.findStubUrl("loanIssuance").toString() + "/name"))

.isEqualTo("loanIssuance");

then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() + "/name"))

.isEqualTo("fraudDetectionServer");

}

**private** String httpGet(String url) **throws** Exception {

**try** (InputStream stream = URI.create(url).toURL().openStream()) {

**return** StreamUtils.copyToString(stream, Charset.forName("UTF-8"));

}

}

}

JUnit 5扩展示例：

*// Visible for Junit*@RegisterExtension**static** StubRunnerExtension stubRunnerExtension = **new** StubRunnerExtension()

.repoRoot(repoRoot()).stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.downloadStub("org.springframework.cloud.contract.verifier.stubs",

"loanIssuance")

.downloadStub(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")

.withMappingsOutputFolder("target/outputmappingsforrule");

@BeforeAll@AfterAll**static** **void** setupProps() {

System.clearProperty("stubrunner.repository.root");

System.clearProperty("stubrunner.classifier");

}

**private** **static** String repoRoot() {

**try** {

**return** StubRunnerRuleJUnitTest.**class**.getResource("/m2repo/repository/")

.toURI().toString();

}

**catch** (Exception e) {

**return** "";

}

}

检查**JUnit和Spring**的**Common属性，**以获取有关如何应用Stub Runner全局配置的更多信息。

|  |  |
| --- | --- |
| IMG_546 | **重要** |
| 要将JUnit规则或JUnit 5扩展与消息传递一起使用，您必须向规则构建器（例如rule.messageVerifier(new MyMessageVerifier())）提供MessageVerifier接口的实现。如果不这样做，则每当您尝试发送消息时，都会引发异常。 |

### **92.4.1 Maven设置**

存根下载器接受其他本地存储库文件夹的Maven设置。当前不考虑存储库和配置文件的身份验证详细信息，因此您需要使用上述属性进行指定。

### **92.4.2提供固定端口**

您还可以在固定端口上运行存根。您可以通过两种不同的方式来实现。一种是在属性中传递它，另一种是通过JUnit规则的流畅API。

### **92.4.3 Fluent API**

使用StubRunnerRule或StubRunnerExtension时，可以添加一个存根进行下载，然后将最后一个已下载存根的端口传递给该端口。

@ClassRule**public** **static** StubRunnerRule rule = **new** StubRunnerRule().repoRoot(repoRoot())

.stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.downloadStub("org.springframework.cloud.contract.verifier.stubs",

"loanIssuance")

.withPort(12345).downloadStub(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:12346");

@BeforeClass@AfterClass**public** **static** **void** setupProps() {

System.clearProperty("stubrunner.repository.root");

System.clearProperty("stubrunner.classifier");

}

您可以看到对于此示例，以下测试有效：

then(rule.findStubUrl("loanIssuance"))

.isEqualTo(URI.create("http://localhost:12345").toURL());

then(rule.findStubUrl("fraudDetectionServer"))

.isEqualTo(URI.create("http://localhost:12346").toURL());

### **92.4.4 Stub Runner与Spring**

设置Stub Runner项目的Spring配置。

通过在配置文件中提供存根列表，Stub Runner会自动下载并在WireMock中注册所选存根。

如果要查找存根依赖项的URL，可以自动装配StubFinder接口，并使用如下所示的方法：

@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)@SpringBootTest(properties = [" stubrunner.cloud.enabled=false",

'foo=${stubrunner.runningstubs.fraudDetectionServer.port}',

'fooWithGroup=${stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port}'])@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",

httpServerStubConfigurer = HttpsForFraudDetection)@ActiveProfiles("test")**class** StubRunnerConfigurationSpec **extends** Specification {

@Autowired

StubFinder stubFinder

@Autowired

Environment environment

@StubRunnerPort("fraudDetectionServer")

**int** fraudDetectionServerPort

@StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")

**int** fraudDetectionServerPortWithGroupId

@Value('${foo}')

Integer foo

@BeforeClass

@AfterClass

**void** setupProps() {

System.clearProperty("stubrunner.repository.root")

System.clearProperty("stubrunner.classifier")

WireMockHttpServerStubAccessor.clear()

}

def 'should mark all ports as random'() {

expect:

WireMockHttpServerStubAccessor.everyPortRandom()

}

def 'should start WireMock servers'() {

expect: 'WireMocks are running'

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null

stubFinder.findStubUrl('loanIssuance') != null

stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')

stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null

and:

stubFinder.findAllRunningStubs().isPresent('loanIssuance')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')

and: 'Stubs were registered'

"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

and: 'Fraud Detection is an HTTPS endpoint'

stubFinder.findStubUrl('fraudDetectionServer').toString().startsWith("https")

}

def 'should throw an exception when stub is not found'() {

when:

stubFinder.findStubUrl('nonExistingService')

then:

thrown(StubNotFoundException)

when:

stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId')

then:

thrown(StubNotFoundException)

}

def 'should register started servers as environment variables'() {

expect:

environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null

stubFinder.findAllRunningStubs().getPort("loanIssuance") == (environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)

and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null

stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as Integer)

and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null

stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port") as Integer)

}

def 'should be able to interpolate a running stub in the passed test property'() {

given:

**int** fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")

expect:

fraudPort > 0

environment.getProperty("foo", Integer) == fraudPort

environment.getProperty("fooWithGroup", Integer) == fraudPort

foo == fraudPort

}

@Issue("#573")

def 'should be able to retrieve the port of a running stub via an annotation'() {

given:

**int** fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")

expect:

fraudPort > 0

fraudDetectionServerPort == fraudPort

fraudDetectionServerPortWithGroupId == fraudPort

}

def 'should dump all mappings to a file'() {

when:

def url = stubFinder.findStubUrl("fraudDetectionServer")

then:

**new** File("target/outputmappings/", "fraudDetectionServer\_${url.port}").exists()

}

@Configuration

@EnableAutoConfiguration

**static** **class** Config {}

@CompileStatic

**static** **class** HttpsForFraudDetection **extends** WireMockHttpServerStubConfigurer {

**private** **static** **final** Log log = LogFactory.getLog(HttpsForFraudDetection)

@Override

WireMockConfiguration configure(WireMockConfiguration httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {

**if** (httpServerStubConfiguration.stubConfiguration.artifactId == "fraudDetectionServer") {

**int** httpsPort = SocketUtils.findAvailableTcpPort()

log.info("Will set HTTPs port [" + httpsPort + "] for fraud detection server")

**return** httpStubConfiguration

.httpsPort(httpsPort)

}

**return** httpStubConfiguration

}

}

}

对于以下配置文件：

stubrunner:

repositoryRoot: classpath:m2repo/repository/

ids:

- org.springframework.cloud.contract.verifier.stubs:loanIssuance

- org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer

- org.springframework.cloud.contract.verifier.stubs:bootService

stubs-mode: remote

除了使用属性外，还可以使用@AutoConfigureStubRunner内部的属性。您可以在下面找到通过在注释上设置值来获得相同结果的示例。

@AutoConfigureStubRunner(

ids = ["org.springframework.cloud.contract.verifier.stubs:loanIssuance",

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",

"org.springframework.cloud.contract.verifier.stubs:bootService"],

stubsMode = StubRunnerProperties.StubsMode.REMOTE,

repositoryRoot = "classpath:m2repo/repository/")

Stub Runner Spring以下列方式为每个已注册的WireMock服务器注册环境变量。Stub Runner ID com.example:foo，com.example:bar的示例。

* stubrunner.runningstubs.foo.port
* stubrunner.runningstubs.com.example.foo.port
* stubrunner.runningstubs.bar.port
* stubrunner.runningstubs.com.example.bar.port

您可以在代码中引用该代码。

您也可以使用@StubRunnerPort批注注入正在运行的存根的端口。注释的值可以是groupid:artifactid，也可以只是artifactid。Stub Runner ID com.example:foo，com.example:bar的示例。

@StubRunnerPort("foo")**int** fooPort;@StubRunnerPort("com.example:bar")**int** barPort;

## **92.5 Stub Runner Spring Cloud**

Stub Runner可以与Spring Cloud集成。

对于现实生活中的示例，您可以查看

* [生产者应用示例](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/producer" \t "https://www.springcloud.cc/_top)
* [消费者应用示例](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.1.x/consumer_with_discovery" \t "https://www.springcloud.cc/_top)

### **92.5.1存根服务发现**

Stub Runner Spring Cloud的最重要特征是它正在存根

* DiscoveryClient
* Ribbon ServerList

这意味着无论您使用的是Zookeeper，Consul，Eureka还是其他任何东西，您都不需要在测试中使用它。我们正在启动依赖项的WireMock实例，并且在您每次使用Feign，直接负载均衡RestTemplate或DiscoveryClient来调用那些存根服务器而不是调用真实服务时，都告诉您的应用程序发现工具。

例如，该测试将通过

def 'should make service discovery work'() {

expect: 'WireMocks are running'

"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

and: 'Stubs can be reached via load service discovery'

restTemplate.getForObject('http://loanIssuance/name', String) == 'loanIssuance'

restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name', String) == 'fraudDetectionServer'

}

对于以下配置文件

stubrunner:

idsToServiceIds:

ivyNotation: someValueInsideYourCode

fraudDetectionServer: someNameThatShouldMapFraudDetectionServer

#### **测试配置文件和服务发现**

在集成测试中，您通常既不想调用发现服务（例如Eureka）也不能调用Config Server。这就是为什么您要创建其他测试配置以禁用这些功能的原因。

由于[spring-cloud-commons](https://github.com/spring-cloud/spring-cloud-commons/issues/156" \t "https://www.springcloud.cc/_top)达到此目的的某些限制，您已通过如下所示的静态块（Eureka的示例）禁用了这些属性

*//Hack to work around https://github.com/spring-cloud/spring-cloud-commons/issues/156*

**static** {

System.setProperty("eureka.client.enabled", "false");

System.setProperty("spring.cloud.config.failFast", "false");

}

### **92.5.2附加配置**

您可以使用stubrunner.idsToServiceIds:映射将存根的artifactId与您的应用程序名称匹配。您可以通过提供：stubrunner.cloud.ribbon.enabled等于false来禁用Stub Runner Ribbon支持，您可以通过提供：stubrunner.cloud.enabled等于false来禁用Stub Runner支持

|  |
| --- |
| IMG_547 |
| 默认情况下，将对所有服务发现进行存根。这意味着无论您是否拥有DiscoveryClient，都将忽略其结果。但是，如果要重用它，只需将stubrunner.cloud.delegate.enabled设置为true，然后现有的DiscoveryClient结果将与存根的结果合并。 |

Stub Runner使用的默认Maven配置可以通过以下系统属性或环境变量进行调整

* maven.repo.local-自定义Maven本地存储库位置的路径
* org.apache.maven.user-settings-自定义Maven用户设置位置的路径
* org.apache.maven.global-settings-Maven全局设置位置的路径

## **92.6 Stub Runner引导应用程序**

Spring Cloud Contract Stub Runner引导是一个Spring Boot应用程序，它公开REST端点以触发消息传递标签并访问启动的WireMock服务器。

用例之一是在已部署的应用程序上运行一些冒烟（端到端）测试。您可以查看[Spring Cloud Pipelines](https://github.com/spring-cloud/spring-cloud-pipelines" \t "https://www.springcloud.cc/_top) 项目以获取更多信息。

### **92.6.1如何使用？**

#### **Stub Runner服务器**

只需添加

compile "org.springframework.cloud:spring-cloud-starter-stub-runner"

用@EnableStubRunnerServer注释课程，建立一个胖子罐，就可以开始了！

对于属性，请检查**Stub Runner Spring**部分。

#### **Stub Runner服务器胖子**

您可以从Maven下载独立的JAR（例如对于版本2.0.1.RELEASE），如下所示：

$ wget -O stub-runner.jar 'https://search.maven.org/remotecontent?filepath=org/springframework/cloud/spring-cloud-contract-stub-runner-boot/2.0.1.RELEASE/spring-cloud-contract-stub-runner-boot-2.0.1.RELEASE.jar'

$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...

#### **Spring Cloud CLI**

从[Spring Cloud CLI](https://cloud.spring.io/spring-cloud-cli" \t "https://www.springcloud.cc/_top) 项目的1.4.0.RELEASE版本开始，您可以通过执行spring cloud stubrunner启动Stub Runner引导。

为了通过配置，只需在当前工作目录或名为config的子目录中或在~/.spring-cloud中创建一个stubrunner.yml文件。该文件可能如下所示（运行本地安装的存根示例）

**stubrunner.yml。**

stubrunner:

stubsMode: LOCAL

ids:

- com.example:beer-api-producer:+:9876

然后只需从终端窗口调用spring cloud stubrunner即可启动Stub Runner服务器。它将在端口8750上可用。

### **92.6.2端点**

#### **HTTP**

* GET /stubs-以ivy:integer表示法返回所有正在运行的存根的列表
* GET /stubs/{ivy}-返回给定ivy表示法的端口（调用端点ivy时也只能是artifactId）

#### **讯息传递**

对于消息

* GET /triggers-以ivy : [ label1, label2 …​]表示法返回所有运行标签的列表
* POST /triggers/{label}-使用label执行触发器
* POST /triggers/{ivy}/{label}-为给定的ivy表示法使用label执行触发器（调用端点ivy时也只能是artifactId）

### **92.6.3例子**

@ContextConfiguration(classes = StubRunnerBoot, loader = SpringBootContextLoader)@SpringBootTest(properties = "spring.cloud.zookeeper.enabled=false")@ActiveProfiles("test")**class** StubRunnerBootSpec **extends** Specification {

@Autowired

StubRunning stubRunning

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning),

**new** TriggerController(stubRunning))

}

def 'should return a list of running stub servers in "full ivy:port" notation'() {

when:

String response = RestAssuredMockMvc.get('/stubs').body.asString()

then:

def root = **new** JsonSlurper().parseText(response)

root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs' **instanceof** Integer

}

def 'should return a port on which a [#stubId] stub is running'() {

when:

def response = RestAssuredMockMvc.get("/stubs/${stubId}")

then:

response.statusCode == 200

Integer.valueOf(response.body.asString()) > 0

where:

stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:+:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootService:+',

'org.springframework.cloud.contract.verifier.stubs:bootService',

'bootService']

}

def 'should return 404 when missing stub was called'() {

when:

def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")

then:

response.statusCode == 404

}

def 'should return a list of messaging labels that can be triggered when version and classifier are passed'() {

when:

String response = RestAssuredMockMvc.get('/triggers').body.asString()

then:

def root = **new** JsonSlurper().parseText(response)

root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs'?.containsAll(["delete\_book", "return\_book\_1", "return\_book\_2"])

}

def 'should trigger a messaging label'() {

given:

StubRunning stubRunning = Mock()

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning), **new** TriggerController(stubRunning))

when:

def response = RestAssuredMockMvc.post("/triggers/delete\_book")

then:

response.statusCode == 200

and:

1 \* stubRunning.trigger('delete\_book')

}

def 'should trigger a messaging label for a stub with [#stubId] ivy notation'() {

given:

StubRunning stubRunning = Mock()

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning), **new** TriggerController(stubRunning))

when:

def response = RestAssuredMockMvc.post("/triggers/$stubId/delete\_book")

then:

response.statusCode == 200

and:

1 \* stubRunning.trigger(stubId, 'delete\_book')

where:

stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:stubs', 'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']

}

def 'should throw exception when trigger is missing'() {

when:

RestAssuredMockMvc.post("/triggers/missing\_label")

then:

Exception e = thrown(Exception)

e.message.contains("Exception occurred while trying to return [missing\_label] label.")

e.message.contains("Available labels are")

e.message.contains("org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs=[]")

e.message.contains("org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs=")

}

}

### **92.6.4 Stub Runner使用服务发现启动**

使用Stub Runner引导程序的一种可能性是将其用作“烟雾测试”的存根的提要。这是什么意思？假设您不想将50个微服务部署到测试环境中以检查您的应用程序是否运行正常。在构建过程中，您已经执行了一组测试，但是您还想确保应用程序的包装是正确的。您可以做的是将应用程序部署到环境中，启动该应用程序并在其上运行一些测试，以查看其是否正常运行。我们可以称这些测试为冒烟测试，因为它们的想法是仅检查少数几个测试场景。

这种方法的问题在于，如果您正在执行微服务，则很可能正在使用服务发现工具。Stub Runner引导程序允许您通过启动所需的存根并将其注册到服务发现工具中来解决此问题。让我们看一下使用Eureka进行这种设置的示例。假设Eureka已经在运行。

@SpringBootApplication@EnableStubRunnerServer@EnableEurekaClient@AutoConfigureStubRunner**public** **class** StubRunnerBootEurekaExample {

**public** **static** **void** main(String[] args) {

SpringApplication.run(StubRunnerBootEurekaExample.**class**, args);

}

}

如您所见，我们想启动Stub Runner引导服务器@EnableStubRunnerServer，启用Eureka客户端@EnableEurekaClient，并且我们想打开桩头运行程序功能@AutoConfigureStubRunner。

现在假设我们要启动此应用程序，以便存根自动注册。我们可以通过运行应用程序java -jar ${SYSTEM\_PROPS} stub-runner-boot-eureka-example.jar来做到这一点，其中${SYSTEM\_PROPS}将包含以下属性列表

\* -Dstubrunner.repositoryRoot=https://repo.spring.io/snapshot (1)

\* -Dstubrunner.cloud.stubbed.discovery.enabled=false (2)

\* -Dstubrunner.ids=org.springframework.cloud.contract.verifier.stubs:loanIssuance,org.

\* springframework.cloud.contract.verifier.stubs:fraudDetectionServer,org.springframework.

\* cloud.contract.verifier.stubs:bootService (3)

\* -Dstubrunner.idsToServiceIds.fraudDetectionServer=

\* someNameThatShouldMapFraudDetectionServer (4)

\*

\* (1) - we tell Stub Runner where all the stubs reside (2) - we don't want the default

\* behaviour where the discovery service is stubbed. That's why the stub registration will

\* be picked (3) - we provide a list of stubs to download (4) - we provide a list of

这样，您部署的应用程序可以通过服务发现将请求发送到启动的WireMock服务器。默认情况下，极有可能在application.yml中设置了1-3点，因为它们不太可能改变。这样，每次启动Stub Runner引导时，您只能提供要下载的存根列表。

## **每个消费者92.7个存根**

在某些情况下，同一端点的2个使用者希望有2个不同的响应。

|  |
| --- |
| IMG_548 |
| 这种方法还使您可以立即知道哪个使用者正在使用API​​的哪一部分。您可以删除API产生的部分响应，并且可以查看哪些自动生成的测试失败。如果没有失败，那么您可以安全地删除响应的那部分，因为没有人使用它。 |

让我们看下面的示例，该示例为生产者定义的合同称为producer。有2个使用者：foo-consumer和bar-consumer。

**消费者**foo-service

request {

url '/foo'

method GET()

}

response {

status OK()

body(

foo: "foo"

}

}

**消费者**bar-service

request {

url '/foo'

method GET()

}

response {

status OK()

body(

bar: "bar"

}

}

您不能为同一请求产生2个不同的响应。因此，您可以正确打包合同，然后从stubsPerConsumer功能中受益。

在生产者方面，消费者可以拥有一个文件夹，其中仅包含与他们相关的合同。通过将stubrunner.stubs-per-consumer标志设置为true，我们不再注册所有存根，而是仅注册与使用者应用程序名称相对应的存根。换句话说，我们将扫描每个存根的路径，如果它在路径中包含带有使用者名称的子文件夹，则它将被注册。

在foo生产商一方，合同看起来像这样

.

└── contracts

├── bar-consumer

│   ├── bookReturnedForBar.groovy

│   └── shouldCallBar.groovy

└── foo-consumer

├── bookReturnedForFoo.groovy

└── shouldCallFoo.groovy

作为bar-consumer使用者，您可以将spring.application.name或stubrunner.consumer-name设置为bar-consumer，也可以按以下方式设置测试：

@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)@SpringBootTest(properties = ["spring.application.name=bar-consumer"])@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",

repositoryRoot = "classpath:m2repo/repository/",

stubsMode = StubRunnerProperties.StubsMode.REMOTE,

stubsPerConsumer = true)**class** StubRunnerStubsPerConsumerSpec **extends** Specification {

...

}

然后，仅允许引用在名称中包含bar-consumer的路径下注册的存根（即来自src/test/resources/contracts/bar-consumer/some/contracts/…​文件夹的存根）。

或明确设置消费者名称

@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)@SpringBootTest@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",

repositoryRoot = "classpath:m2repo/repository/",

consumerName = "foo-consumer",

stubsMode = StubRunnerProperties.StubsMode.REMOTE,

stubsPerConsumer = true)**class** StubRunnerStubsPerConsumerWithConsumerNameSpec **extends** Specification {

...

}

然后，仅允许引用在名称中包含foo-consumer的路径下注册的存根（即来自src/test/resources/contracts/foo-consumer/some/contracts/…​文件夹的存根）。

您可以查看[问题224](https://github.com/spring-cloud/spring-cloud-contract/issues/224" \t "https://www.springcloud.cc/_top)，以了解有关此更改背后原因的更多信息。

## **92.8共同的**

本节简要介绍了常用属性，包括：

* [第92.8.1节“ JUnit和Spring的公用Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "common-properties-junit-spring" \o "92.8.1 JUnit和Spring的通用属性)
* [第92.8.2节“ Stub Runner存根ID”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stub-runner-stub-ids" \o "92.8.2存根运行器存根ID)

### **92.8.1 JUnit和Spring的通用Properties**

您可以使用系统属性或Spring配置属性来设置重复属性。这是它们的名称及其默认值：

| **Property名称** | **默认值** | **描述** |
| --- | --- | --- |
| stubrunner.minPort | 10000 | Minimum value of a port for a started WireMock with stubs. |
| stubrunner.maxPort | 15000 | Maximum value of a port for a started WireMock with stubs. |
| stubrunner.repositoryRoot |  | Maven repo URL. If blank, then call the local maven repo. |
| stubrunner.classifier | stubs | Default classifier for the stub artifacts. |
| stubrunner.stubsMode | CLASSPATH | The way you want to fetch and register the stubs |
| stubrunner.ids |  | Array of Ivy notation stubs to download. |
| stubrunner.username |  | Optional username to access the tool that stores the JARs with stubs. |
| stubrunner.password |  | Optional password to access the tool that stores the JARs with stubs. |
| stubrunner.stubsPerConsumer | false | Set to true if you want to use different stubs for each consumer instead of registering all stubs for every consumer. |
| stubrunner.consumerName |  | If you want to use a stub for each consumer and want to override the consumer name just change this value. |

### **92.8.2 Stub Runner存根ID**

您可以提供存根以通过stubrunner.ids系统属性下载。他们遵循以下模式：

groupId:artifactId:version:classifier:port

请注意，version，classifier和port是可选的。

* 如果您未提供port，则将随机选择一个。
* 如果不提供classifier，则使用默认值。（请注意，您可以通过以下方式传递空的分类器：groupId:artifactId:version:）。
* 如果您未提供version，则将传递+并下载最新的+。

port表示WireMock服务器的端口。

|  |  |
| --- | --- |
| IMG_549 | **重要** |
| 从1.0.4版开始，您可以提供Stub Runner要考虑的一系列版本。您可以[在此处](https://wiki.eclipse.org/Aether/New_and_Noteworthy" \l "Version_Ranges" \t "https://www.springcloud.cc/_top)阅读有关[Aether版本控制范围的](https://wiki.eclipse.org/Aether/New_and_Noteworthy" \l "Version_Ranges" \t "https://www.springcloud.cc/_top)更多信息 。 |

## **92.9 Stub Runner Docker**

我们正在发布一个spring-cloud/spring-cloud-contract-stub-runner Docker映像，它将启动Stub Runner的独立版本。

如果您想了解更多关于Maven的基础知识，工件ID，组ID，分类器和工件管理器，请单击此处[第90.16节“ Docker项目”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "docker-project" \o "90.16 Docker项目)。

### **92.9.1使用方法**

只需执行docker镜像即可。您可以将[第92.8.1节“ JUnit的公用Properties和Spring”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "common-properties-junit-spring" \o "92.8.1 JUnit和Spring的通用属性) 作为环境变量进行传递。惯例是所有字母都应大写。驼峰符号和点（.）应该通过下划线（\_）分隔。例如，stubrunner.repositoryRoot属性应表示为STUBRUNNER\_REPOSITORY\_ROOT环境变量。

### **92.9.2非JVM项目中的客户端用法示例**

我们想使用在[第90.16.4节“服务器端（nodejs）”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "docker-server-side" \o "90.16.4服务器端（nodejs）)步骤中创建的存根。假设我们要在端口9876上运行存根。NodeJS代码在这里可用：

$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs

$ **cd** bookstore

让我们使用存根运行Stub Runner引导应用程序。

*# Provide the Spring Cloud Contract Docker version*

$ SC\_CONTRACT\_DOCKER\_VERSION="..."*# The IP at which the app is running and Docker container can reach it*

$ APP\_IP="192.168.0.100"*# Spring Cloud Contract Stub Runner properties*

$ STUBRUNNER\_PORT="8083"*# Stub coordinates 'groupId:artifactId:version:classifier:port'*

$ STUBRUNNER\_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"

$ STUBRUNNER\_REPOSITORY\_ROOT="http://${APP\_IP}:8081/artifactory/libs-release-local"*# Run the docker with Stub Runner Boot*

$ docker run --rm -e "STUBRUNNER\_IDS=${STUBRUNNER\_IDS}" -e "STUBRUNNER\_REPOSITORY\_ROOT=${STUBRUNNER\_REPOSITORY\_ROOT}" -e "STUBRUNNER\_STUBS\_MODE=REMOTE" -p "${STUBRUNNER\_PORT}:${STUBRUNNER\_PORT}" -p "9876:9876" springcloud/spring-cloud-contract-stub-runner:"${SC\_CONTRACT\_DOCKER\_VERSION}"

这是怎么回事

* 一个独立的Stub Runner应用程序已启动
* 它在端口9876上下载了坐标为com.example:bookstore:0.0.1.RELEASE:stubs的存根
* 它是从Artifactory下载的，运行速度为[http://192.168.0.100:8081/artifactory/libs-release-local](http://192.168.0.100:8081/artifactory/libs-release-local" \t "https://www.springcloud.cc/_top)
* 过一会儿Stub Runner将在端口8083上运行
* 并且存根将在端口9876上运行

在服务器端，我们构建了一个有状态的存根。让我们使用curl声明存根已正确设置。

*# let's execute the first request (no response is returned)*

$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title", "genre" : "Genre", "description" : "Description", "author" : "Author", "publisher" : "Publisher", "pages" : 100, "image\_url" : "https://d213dhlpdb53mu.cloudfront.net/assets/pivotal-square-logo-41418bd391196c3022f3cd9f3959b3f6d7764c47873d858583384e759c7db435.svg", "buy\_url" : "https://pivotal.io" }' http://localhost:9876/api/books*# Now time for the second request*

$ curl -X GET http://localhost:9876/api/books*# You will receive contents of the JSON*

|  |  |
| --- | --- |
| IMG_550 | **重要** |
| 如果要使用在主机上本地构建的存根，则应传递环境变量-e STUBRUNNER\_STUBS\_MODE=LOCAL并安装本地m2的卷-v "${HOME}/.m2/:/root/.m2:ro" |

## **93. Stub Runner用于消息传递**

Stub Runner可以在内存中运行已发布的存根。它可以与以下框架集成：

* Spring Integration
* Spring Cloud Stream
* 阿帕奇骆驼
* Spring AMQP

它还提供了与市场上任何其他解决方案集成的切入点。

|  |  |
| --- | --- |
| IMG_551 | **重要** |
| 如果您在类路径Stub Runner上有多个框架，则需要定义应使用的框架。假设您在类路径上同时具有AMQP，Spring Cloud Stream和Spring Integration。然后，您需要设置stubrunner.stream.enabled=false和stubrunner.integration.enabled=false。这样，剩下的唯一框架就是Spring AMQP。 |

## **93.1存根触发**

要触发消息，请使用StubTrigger界面：

**package** org.springframework.cloud.contract.stubrunner;

**import** java.util.Collection;**import** java.util.Map;

**/\*\***

**\* Contract for triggering stub messages.**

**\***

**\* @author Marcin Grzejszczak**

**\*/public** **interface** StubTrigger {

**/\*\***

**\* Triggers an event by a given label for a given {@code groupid:artifactid} notation.**

**\* You can use only {@code artifactId} too.**

**\***

**\* Feature related to messaging.**

**\* @param ivyNotation ivy notation of a stub**

**\* @param labelName name of the label to trigger**

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger(String ivyNotation, String labelName);

**/\*\***

**\* Triggers an event by a given label.**

**\***

**\* Feature related to messaging.**

**\* @param labelName name of the label to trigger**

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger(String labelName);

**/\*\***

**\* Triggers all possible events.**

**\***

**\* Feature related to messaging.**

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger();

**/\*\***

**\* Feature related to messaging.**

**\* @return a mapping of ivy notation of a dependency to all the labels it has.**

**\*/**

Map<String, Collection<String>> labels();

}

为方便起见，StubFinder接口扩展了StubTrigger，因此您在测试中只需要一个即可。

StubTrigger提供了以下触发消息的选项：

* [第93.1.1节“按标签触发”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-label" \o "93.1.1按标签触发)
* [第93.1.2节“按组和工件ID触发”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-group-artifact-ids" \o "93.1.2按组和工件ID触发)
* [第93.1.3节“由工件ID触发”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-artifact-ids" \o "93.1.3由工件ID触发)
* [第93.1.4节“触发所有消息”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "trigger-all-messages" \o "93.1.4触发所有消息)

### **93.1.1按标签触发**

stubFinder.trigger('return\_book\_1')

### **93.1.2按组和工件ID触发**

stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService', 'return\_book\_1')

### **93.1.3由工件ID触发**

stubFinder.trigger('streamService', 'return\_book\_1')

### **93.1.4触发所有消息**

stubFinder.trigger()

## **93.2 Stub Runner Camel**

Spring Cloud Contract验证程序Stub Runner的消息传递模块为您提供了一种与Apache Camel集成的简便方法。对于提供的工件，它将自动下载存根并注册所需的路由。

### **93.2.1将其添加到项目中**

在类路径上同时包含Apache Camel和Spring Cloud Contract Stub Runner就足够了。请记住用@AutoConfigureStubRunner注释测试类。

### **93.2.2禁用功能**

如果需要禁用此功能，只需传递stubrunner.camel.enabled=false属性。

### **93.2.3例子**

#### **存根结构**

让我们假设我们有以下Maven存储库，其中包含用于camelService应用程序的已部署存根。

└── .m2

└── repository

└── io

└── codearte

└── accurest

└── stubs

└── camelService

├── 0.0.1-SNAPSHOT

│   ├── camelService-0.0.1-SNAPSHOT.pom

│   ├── camelService-0.0.1-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

存根包含以下结构：

├── META-INF

│   └── MANIFEST.MF

└── repository

├── accurest

│   ├── bookDeleted.groovy

│   ├── bookReturned1.groovy

│   └── bookReturned2.groovy

└── mappings

让我们考虑以下合同（用**1**编号）：

Contract.make {

label 'return\_book\_1'

input {

triggeredBy('bookReturnedTriggered()')

}

outputMessage {

sentTo('jms:output')

body('''{ "bookName" : "foo" }''')

headers {

header('BOOK-NAME', 'foo')

}

}

}

和数字**2**

Contract.make {

label 'return\_book\_2'

input {

messageFrom('jms:input')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('jms:output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

#### **方案1（无输入消息）**

为了通过return\_book\_1标签触发消息，我们将使用StubTigger接口，如下所示

stubFinder.trigger('return\_book\_1')

接下来，我们要收听发送到jms:output的消息的输出

Exchange receivedMessage = consumerTemplate.receive('jms:output', 5000)

并且收到的消息将通过以下断言

receivedMessage != null

assertThatBodyContainsBookNameFoo(receivedMessage.in.body)

receivedMessage.in.headers.get('BOOK-NAME') == 'foo'

#### **场景2（由输入触发输出）**

由于已为您设置了路由，仅向jms:output目标发送一条消息就足够了。

producerTemplate.

sendBodyAndHeaders('jms:input', **new** BookReturned('foo'), [sample: 'header'])

接下来，我们要监听发送到jms:output的消息的输出

Exchange receivedMessage = consumerTemplate.receive('jms:output', 5000)

并且收到的消息将通过以下断言

receivedMessage != null

assertThatBodyContainsBookNameFoo(receivedMessage.in.body)

receivedMessage.in.headers.get('BOOK-NAME') == 'foo'

#### **方案3（输入无输出）**

由于已为您设置了路由，仅向jms:output目标发送一条消息就足够了。

producerTemplate.

sendBodyAndHeaders('jms:delete', **new** BookReturned('foo'), [sample: 'header'])

## **93.3 Stub Runner集成**

Spring Cloud Contract验证程序Stub Runner的消息传递模块为您提供了一种与Spring Integration集成的简便方法。对于提供的工件，它会自动下载存根并注册所需的路由。

### **93.3.1将运行器添加到项目**

您可以在类路径上同时使用Spring Integration和Spring Cloud Contract Stub Runner。请记住用@AutoConfigureStubRunner注释测试类。

### **93.3.2禁用功能**

如果需要禁用此功能，请设置stubrunner.integration.enabled=false属性。

假设您具有以下Maven存储库，其中包含integrationService应用程序的已部署存根：

└── .m2

└── repository

└── io

└── codearte

└── accurest

└── stubs

└── integrationService

├── 0.0.1-SNAPSHOT

│   ├── integrationService-0.0.1-SNAPSHOT.pom

│   ├── integrationService-0.0.1-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

进一步假设存根包含以下结构：

├── META-INF

│   └── MANIFEST.MF

└── repository

├── accurest

│   ├── bookDeleted.groovy

│   ├── bookReturned1.groovy

│   └── bookReturned2.groovy

└── mappings

考虑以下合同（编号**1**）：

Contract.make {

label 'return\_book\_1'

input {

triggeredBy('bookReturnedTriggered()')

}

outputMessage {

sentTo('output')

body('''{ "bookName" : "foo" }''')

headers {

header('BOOK-NAME', 'foo')

}

}

}

现在考虑**2**：

Contract.make {

label 'return\_book\_2'

input {

messageFrom('input')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

以及以下Spring Integration路线：

<?xml version="1.0" encoding="UTF-8"?><beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:beans="http://www.springframework.org/schema/beans"

xmlns="http://www.springframework.org/schema/integration"

xsi:schemaLocation="http://www.springframework.org/schema/beans

https://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/integration

http://www.springframework.org/schema/integration/spring-integration.xsd">

*<!-- REQUIRED FOR TESTING -->*

<bridge input-channel="output"

output-channel="outputTest"/>

<channel id="outputTest">

<queue/>

</channel>

</beans:beans>

这些示例适用于三种情况：

* [称为“方案1（无输入消息）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-1" \o "方案1（无输入消息）)
* [称为“方案2（由输入触发的输出）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-2" \o "场景2（由输入触发输出）)
* [称为“方案3（输入无输出）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "integration-scenario-3" \o "方案3（输入无输出）)

#### **方案1（无输入消息）**

要通过return\_book\_1标签触发消息，请使用StubTigger接口，如下所示：

stubFinder.trigger('return\_book\_1')

要监听发送到output的消息的输出，请执行以下操作：

Message<?> receivedMessage = messaging.receive('outputTest')

收到的消息将通过以下断言：

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

#### **场景2（由输入触发输出）**

由于已为您设置了路由，因此您可以向output目标发送消息：

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'input')

要监听发送到output的消息的输出，请执行以下操作：

Message<?> receivedMessage = messaging.receive('outputTest')

收到的消息传递以下断言：

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

#### **方案3（输入无输出）**

由于已为您设置了路由，因此您可以向input目标发送消息：

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'delete')

## **93.4 Stub Runner Stream**

Spring Cloud Contract验证程序Stub Runner的消息传递模块为您提供了一种与Spring Stream集成的简便方法。对于提供的工件，它会自动下载存根并注册所需的路由。

|  |
| --- |
| IMG_552 |
| 如果Stub Runner与Stream集成，则首先将messageFrom或sentTo字符串解析为频道的destination，并且不存在这样的destination，则将目的地解析为频道名称。 |

|  |  |
| --- | --- |
| IMG_553 | **重要** |
| 如果要使用Spring Cloud Stream，请记住要添加对org.springframework.cloud:spring-cloud-stream-test-support的依赖。 |

**Maven.**

<dependency>

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

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope></dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-stream-test-support"

### **93.4.1将运行器添加到项目**

您可以在类路径上同时使用Spring Cloud Stream和Spring Cloud Contract Stub Runner。请记住用@AutoConfigureStubRunner注释测试类。

### **93.4.2禁用功能**

如果需要禁用此功能，请设置stubrunner.stream.enabled=false属性。

假设您具有以下Maven存储库，其中包含streamService应用程序的已部署存根：

└── .m2

└── repository

└── io

└── codearte

└── accurest

└── stubs

└── streamService

├── 0.0.1-SNAPSHOT

│   ├── streamService-0.0.1-SNAPSHOT.pom

│   ├── streamService-0.0.1-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

进一步假设存根包含以下结构：

├── META-INF

│   └── MANIFEST.MF

└── repository

├── accurest

│   ├── bookDeleted.groovy

│   ├── bookReturned1.groovy

│   └── bookReturned2.groovy

└── mappings

考虑以下合同（编号**1**）：

Contract.make {

label 'return\_book\_1'

input { triggeredBy('bookReturnedTriggered()') }

outputMessage {

sentTo('returnBook')

body('''{ "bookName" : "foo" }''')

headers { header('BOOK-NAME', 'foo') }

}

}

现在考虑**2**：

Contract.make {

label 'return\_book\_2'

input {

messageFrom('bookStorage')

messageBody([

bookName: 'foo'

])

messageHeaders { header('sample', 'header') }

}

outputMessage {

sentTo('returnBook')

body([

bookName: 'foo'

])

headers { header('BOOK-NAME', 'foo') }

}

}

现在考虑以下Spring配置：

stubrunner.repositoryRoot: classpath:m2repo/repository/stubrunner.ids: org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-SNAPSHOT:stubsstubrunner.stubs-mode: remotespring: cloud: stream: bindings: output: destination: returnBook input: destination: bookStorage

server: port: 0

debug: **true**

这些示例适用于三种情况：

* [称为“方案1（无输入消息）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-1" \o "方案1（无输入消息）)
* [称为“方案2（由输入触发的输出）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-2" \o "场景2（由输入触发输出）)
* [称为“方案3（输入无输出）”的部分](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "stream-scenario-3" \o "方案3（输入无输出）)

#### **方案1（无输入消息）**

要通过return\_book\_1标签触发消息，请使用StubTrigger接口，如下所示：

stubFinder.trigger('return\_book\_1')

要收听发送到destination为returnBook的频道的消息的输出，请执行以下操作：

Message<?> receivedMessage = messaging.receive('returnBook')

收到的消息传递以下断言：

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

#### **场景2（由输入触发输出）**

由于已为您设置了路线，因此您可以向bookStorage destination发送消息：

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'bookStorage')

要监听发送到returnBook的消息的输出，请执行以下操作：

Message<?> receivedMessage = messaging.receive('returnBook')

收到的消息传递以下断言：

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

#### **方案3（输入无输出）**

由于已为您设置了路由，因此您可以向output目标发送消息：

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'delete')

## **93.5 Stub Runner Spring AMQP**

Spring Cloud Contract验证程序Stub Runner的消息传递模块提供了一种与Spring AMQP的Rabbit模板集成的简便方法。对于提供的工件，它会自动下载存根并注册所需的路由。

集成尝试独立工作（即，不与正在运行的RabbitMQ消息代理进行交互）。它期望在应用程序上下文中使用RabbitTemplate并将其用作名为@SpyBean的spring boot测试。结果，它可以使用模仿间谍功能来验证和检查应用程序发送的消息。

在消息使用者方面，存根运行器考虑应用程序上下文中的所有@RabbitListener带注释的终结点和所有SimpleMessageListenerContainer对象。

由于消息通常以AMQP的形式发送到交易所，因此消息合同包含交易所名称作为目的地。另一端的消息侦听器绑定到队列。绑定将交换连接到队列。如果触发了消息合同，则Spring AMQP存根运行器集成会在应用程序上下文中查找与此交换匹配的绑定。然后，它从Spring交换收集队列，并尝试查找绑定到这些队列的消息侦听器。将为所有匹配的消息侦听器触发该消息。

如果您需要使用路由键，则足以通过amqp\_receivedRoutingKey消息传递标头传递它们。

### **93.5.1将运行器添加到项目**

您可以在类路径上同时使用Spring AMQP和Spring Cloud Contract Stub Runner，并设置属性stubrunner.amqp.enabled=true。请记住用@AutoConfigureStubRunner注释测试类。

|  |  |
| --- | --- |
| IMG_554 | **重要** |
| 如果您已经在类路径上具有Stream and Integration，则需要通过设置stubrunner.stream.enabled=false和stubrunner.integration.enabled=false属性来显式禁用它们。 |

假设您具有以下Maven存储库，其中包含spring-cloud-contract-amqp-test应用程序的已部署存根。

└── .m2

└── repository

└── com

└── example

└── spring-cloud-contract-amqp-**test**

├── 0.4.0-SNAPSHOT

│   ├── spring-cloud-contract-amqp-**test**-0.4.0-SNAPSHOT.pom

│   ├── spring-cloud-contract-amqp-**test**-0.4.0-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

进一步假设存根包含以下结构：

├── META-INF

│   └── MANIFEST.MF

└── contracts

└── shouldProduceValidPersonData.groovy

考虑以下合同：

Contract.make {

*// Human readable description*

description 'Should produce valid person data'

*// Label by means of which the output message can be triggered*

label 'contract-test.person.created.event'

*// input to the contract*

input {

*// the contract will be triggered by a method*

triggeredBy('createPerson()')

}

*// output message of the contract*

outputMessage {

*// destination to which the output message will be sent*

sentTo 'contract-test.exchange'

headers {

header('contentType': 'application/json')

header('\_\_TypeId\_\_': 'org.springframework.cloud.contract.stubrunner.messaging.amqp.Person')

}

*// the body of the output message*

body([

id : $(consumer(9), producer(regex("[0-9]+"))),

name: "me"

])

}

}

现在考虑以下Spring配置：

stubrunner: repositoryRoot: classpath:m2repo/repository/ ids: org.springframework.cloud.contract.verifier.stubs.amqp:spring-cloud-contract-amqp-test:0.4.0-SNAPSHOT:stubs stubs-mode: remote amqp: enabled: **true**server: port: 0

#### **触发消息**

要使用上述合同触发消息，请使用StubTrigger界面，如下所示：

stubTrigger.trigger("contract-test.person.created.event")

该消息的目的地为contract-test.exchange，因此Spring AMQP存根运行器集成查找与该交换有关的绑定。

@Bean**public** Binding binding() {

**return** BindingBuilder.bind(**new** Queue("test.queue"))

.to(**new** DirectExchange("contract-test.exchange")).with("#");

}

绑定定义绑定队列test.queue。结果，以下侦听器定义将与合同消息匹配并调用。

@Bean**public** SimpleMessageListenerContainer simpleMessageListenerContainer(

ConnectionFactory connectionFactory,

MessageListenerAdapter listenerAdapter) {

SimpleMessageListenerContainer container = **new** SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames("test.queue");

container.setMessageListener(listenerAdapter);

**return** container;

}

此外，以下带注释的侦听器将匹配并被调用：

@RabbitListener(bindings = @QueueBinding(value = @Queue("test.queue"), exchange = @Exchange(value = "contract-test.exchange", ignoreDeclarationExceptions = "true")))**public** **void** handlePerson(Person person) {

**this**.person = person;

}

|  |
| --- |
| IMG_555 |
| 该消息被直接移交给与匹配的SimpleMessageListenerContainer相关联的MessageListener的onMessage方法。 |

#### **Spring AMQP测试配置**

为了避免Spring AMQP在我们的测试期间尝试连接到正在运行的代理，请配置模拟ConnectionFactory。

要禁用模拟的ConnectionFactory，请设置以下属性：stubrunner.amqp.mockConnection=false

stubrunner: amqp: mockConnection: **false**

## **94. Contract DSL**

Spring Cloud Contract支持2种类型的DSL。一种写在Groovy中，另一种写在YAML中。

如果您决定将合同写在Groovy中，那么如果您以前没有使用过Groovy，请不要惊慌。确实不需要语言知识，因为Contract DSL仅使用它的一小部分（仅文字，方法调用和闭包）。同样，DSL是静态类型的，以使其在不了解DSL本身的情况下就可以被程序员读取。

|  |  |
| --- | --- |
| IMG_556 | **重要** |
| 请记住，在Groovy合同文件中，必须为Contract类和make静态导入（例如org.springframework.cloud.spec.Contract.make { …​ }）提供完全限定名称。您还可以导入Contract类：import org.springframework.cloud.spec.Contract，然后调用Contract.make { …​ }。 |

|  |
| --- |
| IMG_557 |
| Spring Cloud Contract支持在单个文件中定义多个合同。 |

以下是Groovy合同定义的完整示例：

以下是YAML合同定义的完整示例：

description: Some description

name: some name

priority: 8

ignored: true

request:

url: /foo

queryParameters:

a: b

b: c

method: PUT

headers:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

|  |
| --- |
| IMG_558 |
| 您可以使用独立的maven命令将合同编译为存根映射：mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert |

## **94.1局限性**

|  |
| --- |
| IMG_559 |
| Spring Cloud Contract验证程序未正确支持XML。请使用JSON或帮助我们实现此功能。 |

|  |
| --- |
| IMG_560 |
| 验证JSON数组大小的支持是实验性的。如果要打开它，请将以下系统属性的值设置为true：spring.cloud.contract.verifier.assert.size。默认情况下，此功能设置为false。您还可以在插件配置中提供assertJsonSize属性。 |

|  |
| --- |
| IMG_561 |
| 由于JSON结构可以具有任何形式，因此在使用Groovy DSL和GString中的value(consumer(…​), producer(…​))表示法时，可能无法正确解析它。这就是为什么您应该使用Groovy地图符号的原因。 |

## **94.2通用顶级元素**

以下各节描述了最常见的顶级元素：

* [第94.2.1节“描述”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-description" \o "94.2.1说明)
* [第94.2.2节“名称”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-name" \o "94.2.2名称)
* [第94.2.3节“忽略Contracts”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-ignoring-contracts" \o "94.2.3忽略合同)
* [第94.2.4节“从文件传递值”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-passing-values-from-files" \o "94.2.4从文件传递值)
* [第94.2.5节“ HTTP顶级元素”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-http-top-level-elements" \o "94.2.5 HTTP顶级元素)

### **94.2.1说明**

您可以在合同中添加description。描述是任意文本。以下代码显示了一个示例：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

description('''given:

An input

when:

Sth happens

then:

Output''') }

**YAML。**

description: Some description

name: some name

priority: 8

ignored: true

request:

url: /foo

queryParameters:

a: b

b: c

method: PUT

headers:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

### **94.2.2名称**

您可以为合同提供名称。假设您提供了以下名称：should register a user。如果这样做，自动生成的测试的名称为validate\_should\_register\_a\_user。另外，WireMock存根中的存根的名称为should\_register\_a\_user.json。

|  |  |
| --- | --- |
| IMG_562 | **重要** |
| 您必须确保该名称不包含任何使生成的测试无法编译的字符。另外，请记住，如果为多个合同提供相同的名称，则自动生成的测试将无法编译，并且生成的存根会相互覆盖。 |

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

name("some\_special\_name")

}

**YAML。**

name: some name

### **94.2.3忽略Contracts**

如果要忽略合同，则可以在插件配置中设置忽略合同的值，也可以在合同本身上设置ignored属性：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

ignored()

}

**YAML。**

ignored: true

### **94.2.4从文件传递值**

从版本1.2.0开始，您可以传递文件中的值。假设您在我们的项目中拥有以下资源。

└── src

    └── **test**

       └── resources

          └── contracts

    ├── readFromFile.groovy

    ├── request.json

    └── response.json

进一步假设您的合同如下：

**Groovy DSL。**

*/\**

*\* Copyright 2013-2019 the original author or authors.*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* https://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* limitations under the License.*

*\*/*

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

request {

method('PUT')

headers {

contentType(applicationJson())

}

body(file("request.json"))

url("/1")

}

response {

status OK()

body(file("response.json"))

headers {

contentType(applicationJson())

}

}

}

**YAML。**

request:

method: GET

url: /foo

bodyFromFile: request.json

response:

status: 200

bodyFromFile: response.json

进一步假设JSON文件如下：

**request.json**

**{**

"status": "REQUEST"**}**

**response.json**

**{**

"status": "RESPONSE"**}**

当进行测试或存根生成时，文件的内容将传递到请求或响应的主体。文件名必须是相对于合同所在文件夹的位置的文件。

如果您需要以二进制格式传递文件的内容，则足以使用Groovy DSL中的fileAsBytes方法或YAML中的bodyFromFileAsBytes字段。

**Groovy DSL。**

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

request {

url("/1")

method(PUT())

headers {

contentType(applicationOctetStream())

}

body(fileAsBytes("request.pdf"))

}

response {

status 200

body(fileAsBytes("response.pdf"))

headers {

contentType(applicationOctetStream())

}

}

}

**YAML。**

request:

url: /1

method: PUT

headers:

Content-Type: application/octet-stream

bodyFromFileAsBytes: request.pdf

response:

status: 200

bodyFromFileAsBytes: response.pdf

headers:

Content-Type: application/octet-stream

|  |  |
| --- | --- |
| IMG_563 | **重要** |
| 每当要使用HTTP和消息传递的二进制有效负载时，都应使用此方法。 |

### **94.2.5 HTTP顶级元素**

在合同定义的顶级闭合中可以调用以下方法。request和response是必需的。priority是可选的。

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

*// Definition of HTTP request part of the contract*

*// (this can be a valid request or invalid depending*

*// on type of contract being specified).*

request {

method GET()

url "/foo"

*//...*

}

*// Definition of HTTP response part of the contract*

*// (a service implementing this contract should respond*

*// with following response after receiving request*

*// specified in "request" part above).*

response {

status 200

*//...*

}

*// Contract priority, which can be used for overriding*

*// contracts (1 is highest). Priority is optional.*

priority 1

}

**YAML。**

priority: 8

request:

...

response:

...

|  |  |
| --- | --- |
| IMG_564 | **重要** |
| 如果要使合同具有**较高**的优先级，则需要将**较低的**数字传递给priority标签/方法。例如，值5的priority的优先级高于值10的priority。**更高**比priority与{值4141 /}优先级。 |

## **94.3要求**

HTTP协议只需要在请求中指定**方法和URL**。合同的请求定义中必须包含相同的信息。

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*// HTTP request method (GET/POST/PUT/DELETE).*

method 'GET'

*// Path component of request URL is specified as follows.*

urlPath('/users')

}

response {

*//...*

status 200

}

}

**YAML。**

method: PUT

url: /foo

可以指定一个绝对值而不是相对值url，但是建议使用urlPath，因为这样做会使测试**独立**于**主机**。

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

*// Specifying `url` and `urlPath` in one contract is illegal.*

url('http://localhost:8888/users')

}

response {

*//...*

status 200

}

}

**YAML。**

request:

method: PUT

urlPath: /foo

request可能包含**查询参数**。

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

method GET()

urlPath('/users') {

*// Each parameter is specified in form*

*// `'paramName' : paramValue` where parameter value*

*// may be a simple literal or one of matcher functions,*

*// all of which are used in this example.*

queryParameters {

*// If a simple literal is used as value*

*// default matcher function is used (equalTo)*

parameter 'limit': 100

*// `equalTo` function simply compares passed value*

*// using identity operator (==).*

parameter 'filter': equalTo("email")

*// `containing` function matches strings*

*// that contains passed substring.*

parameter 'gender': value(consumer(containing("[mf]")), producer('mf'))

*// `matching` function tests parameter*

*// against passed regular expression.*

parameter 'offset': value(consumer(matching("[0-9]+")), producer(123))

*// `notMatching` functions tests if parameter*

*// does not match passed regular expression.*

parameter 'loginStartsWith': value(consumer(notMatching(".{0,2}")), producer(3))

}

}

*//...*

}

response {

*//...*

status 200

}

}

**YAML。**

request:

...

queryParameters:

a: b

b: c

headers:

foo: bar

fooReq: baz

cookies:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

fixedDelayMilliseconds: 1000

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

cookies:

- key: foo2

regex: bar

- key: foo3

predefined:

request可能包含其他**请求标头**，如以下示例所示：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

method GET()

url "/foo"

*// Each header is added in form `'Header-Name' : 'Header-Value'`.*

*// there are also some helper methods*

headers {

header 'key': 'value'

contentType(applicationJson())

}

*//...*

}

response {

*//...*

status 200

}

}

**YAML。**

request:

...

headers:

foo: bar

fooReq: baz

request可能包含其他**请求cookie**，如以下示例所示：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

method GET()

url "/foo"

*// Each Cookies is added in form `'Cookie-Key' : 'Cookie-Value'`.*

*// there are also some helper methods*

cookies {

cookie 'key': 'value'

cookie('another\_key', 'another\_value')

}

*//...*

}

response {

*//...*

status 200

}

}

**YAML。**

request:

...

cookies:

foo: bar

fooReq: baz

request可能包含一个**请求正文**：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

method GET()

url "/foo"

*// Currently only JSON format of request body is supported.*

*// Format will be determined from a header or body's content.*

body '''{ "login" : "john", "name": "John The Contract" }'''

}

response {

*//...*

status 200

}

}

**YAML。**

request:

...

body:

foo: bar

request可能包含**多部分**元素。要包含多部分元素，请使用multipart方法/部分，如以下示例所示

**Groovy DSL。**

**YAML。**

request:

method: PUT

url: /multipart

headers:

Content-Type: multipart/form-data;boundary=AaB03x

multipart:

params:

# key (parameter name), value (parameter value) pair

formParameter: '"formParameterValue"'

someBooleanParameter: true

named:

- paramName: file

fileName: filename.csv

fileContent: file content

matchers:

multipart:

params:

- key: formParameter

regex: ".+"

- key: someBooleanParameter

predefined: any\_boolean

named:

- paramName: file

fileName:

predefined: non\_empty

fileContent:

predefined: non\_empty

response:

status: 200

在前面的示例中，我们以两种方式之一定义参数：

**Groovy DSL**

* 直接使用映射符号，其中值可以是动态属性（例如formParameter: $(consumer(…​), producer(…​))）。
* 通过使用允许您设置命名参数的named(…​)方法。命名参数可以设置name和content。您可以通过带有两个参数的方法（例如，named("fileName", "fileContent")）或通过映射表示法（例如，named(name: "fileName", content: "fileContent")）来调用它。

**YAML**

* 通过multipart.params部分设置多部分参数
* 可以通过multipart.named部分设置命名参数（给定参数名称的fileName和fileContent）。该部分包含paramName（参数名称），fileName（文件名称），fileContent（文件内容）字段

可以通过matchers.multipart部分设置动态位

* + 对于参数，请使用params部分，该部分可以接受regex或predefined正则表达式
  + 对于命名参数，请使用named部分，其中首先通过paramName定义参数名称，然后可以通过regex或{12 /传递fileName或fileContent的参数化} 正则表达式

根据该合同，生成的测试如下：

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "multipart/form-data;boundary=AaB03x")

.param("formParameter", "\"formParameterValue\"")

.param("someBooleanParameter", "true")

.multiPart("file", "filename.csv", "file content".getBytes());

*// when:*

ResponseOptions response = given().spec(request)

.put("/multipart");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

WireMock存根如下：

'''**{**

"request" : **{**

"url" : "/multipart"**,**

"method" : "PUT"**,**

"headers" : **{**

"Content-Type" : **{**

"matches" : "multipart/form-data;boundary=AaB03x.\*"

**}**

**},**

"bodyPatterns" : **[** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"formParameter\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n\\".+\\"\\r\\n--\\\\1.\*"

**},** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"someBooleanParameter\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n(true|false)\\r\\n--\\\\1.\*"

**},** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"file\\"; filename=\\"[\\\\S\\\\s]+\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n[\\\\S\\\\s]+\\r\\n--\\\\1.\*"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"transformers" : **[** "response-template"**,** "foo-transformer" **]**

**}}**

'''

## **94.4回应**

响应必须包含**HTTP状态代码，**并且可能包含其他信息。以下代码显示了一个示例：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

method GET()

url "/foo"

}

response {

*// Status code sent by the server*

*// in response to request specified above.*

status OK()

}

}

**YAML。**

response:

...

status: 200

除了status之外，响应还可以包含**header**，**cookie**和**body**，它们的指定方式与请求中的指定方式相同（请参见上一段）。

|  |
| --- |
| IMG_565 |
| 通过Groovy DSL，您可以引用org.springframework.cloud.contract.spec.internal.HttpStatus方法来提供有意义的状态而不是数字。例如，您可以呼叫OK()来获取状态200或致电BAD\_REQUEST()来获取400。 |

## **94.5动态特性**

合同可以包含一些动态属性：时间戳记，ID等。您不想强迫使用者将其时钟存根始终返回相同的时间值，以使它与存根匹配。

对于Groovy DSL，您可以通过两种方式在合同中提供动态部分：将它们直接传递到正文中，或在称为bodyMatchers的单独部分中进行设置。

|  |
| --- |
| IMG_566 |
| 在2.0.0之前，这些版本是使用testMatchers和stubMatchers设置的，请查看[迁移指南](https://github.com/spring-cloud/spring-cloud-contract/wiki/Spring-Cloud-Contract-2.0-Migration-Guide" \t "https://www.springcloud.cc/_top)以获取更多信息。 |

对于YAML，您只能使用matchers部分。

### **94.5.1体内的动态特性**

|  |  |
| --- | --- |
| IMG_567 | **重要** |
| 本部分仅对Groovy DSL有效。请查看 [第94.5.7节“匹配器节中的动态Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-matchers" \o "94.5.7匹配器部分中的动态属性)一节，以获取类似功能的YAML示例。 |

您可以使用value方法设置主体内部的属性，或者如果使用Groovy映射符号，则可以使用$()设置主体内部的属性。下面的示例演示如何使用value方法设置动态属性：

value(consumer(...), producer(...))

value(c(...), p(...))

value(stub(...), test(...))

value(client(...), server(...))

以下示例说明如何使用$()设置动态属性：

$(consumer(...), producer(...))

$(c(...), p(...))

$(stub(...), test(...))

$(client(...), server(...))

两种方法都同样有效。stub和client方法是consumer方法的别名。接下来的部分将详细介绍您可以使用这些值做什么。

### **94.5.2正则表达式**

|  |  |
| --- | --- |
| IMG_568 | **重要** |
| 本部分仅对Groovy DSL有效。请查看 [第94.5.7节“匹配器节中的动态Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-matchers" \o "94.5.7匹配器部分中的动态属性)一节，以获取类似功能的YAML示例。 |

您可以使用正则表达式在Contract DSL中编写请求。当您想要指示应遵循给定模式的请求提供给定响应时，这样做特别有用。另外，在测试和服务器端测试都需要使用模式而不是精确值时，可以使用正则表达式。

下面的示例演示如何使用正则表达式编写请求：

org.springframework.cloud.contract.spec.Contract.make {

request {

method('GET')

url $(consumer(~/\/[0-9]{2}/), producer('/12'))

}

response {

status OK()

body(

id: $(anyNumber()),

surname: $(

consumer('Kowalsky'),

producer(regex('[a-zA-Z]+'))

),

name: 'Jan',

created: $(consumer('2014-02-02 12:23:43'), producer(execute('currentDate(it)'))),

correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-72d2220dd827'),

producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))

)

)

headers {

header 'Content-Type': 'text/plain'

}

}

}

您也只能在通信的一侧提供正则表达式。如果这样做，那么合同引擎将自动提供与提供的正则表达式匹配的生成的字符串。以下代码显示了一个示例：

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url value(consumer(regex('/foo/[0-9]{5}')))

body([

requestElement: $(consumer(regex('[0-9]{5}')))

])

headers {

header('header', $(consumer(regex('application\\/vnd\\.fraud\\.v1\\+json;.\*'))))

}

}

response {

status OK()

body([

responseElement: $(producer(regex('[0-9]{7}')))

])

headers {

contentType("application/vnd.fraud.v1+json")

}

}

}

在前面的示例中，通信的另一端具有为请求和响应而生成的相应数据。

Spring Cloud Contract带有一系列可在合同中使用的预定义正则表达式，如以下示例所示：

**protected** **static** **final** Pattern TRUE\_OR\_FALSE = Pattern.compile(/(true|false)/)**protected** **static** **final** Pattern ALPHA\_NUMERIC = Pattern.compile('[a-zA-Z0-9]+')**protected** **static** **final** Pattern ONLY\_ALPHA\_UNICODE = Pattern.compile(/[\p{L}]\*/)**protected** **static** **final** Pattern NUMBER = Pattern.compile('-?(\\d\*\\.\\d+|\\d+)')**protected** **static** **final** Pattern INTEGER = Pattern.compile('-?(\\d+)')**protected** **static** **final** Pattern POSITIVE\_INT = Pattern.compile('([1-9]\\d\*)')**protected** **static** **final** Pattern DOUBLE = Pattern.compile('-?(\\d\*\\.\\d+)')**protected** **static** **final** Pattern HEX = Pattern.compile('[a-fA-F0-9]+')**protected** **static** **final** Pattern IP\_ADDRESS = Pattern.

compile('([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])')**protected** **static** **final** Pattern HOSTNAME\_PATTERN = Pattern.

compile('((http[s]?|ftp):/)/?([^:/\\s]+)(:[0-9]{1,5})?')**protected** **static** **final** Pattern EMAIL = Pattern.

compile('[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,6}')**protected** **static** **final** Pattern URL = UrlHelper.URL**protected** **static** **final** Pattern HTTPS\_URL = UrlHelper.HTTPS\_URL**protected** **static** **final** Pattern UUID = Pattern.

compile('[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}')**protected** **static** **final** Pattern ANY\_DATE = Pattern.

compile('(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])')**protected** **static** **final** Pattern ANY\_DATE\_TIME = Pattern.

compile('([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')**protected** **static** **final** Pattern ANY\_TIME = Pattern.

compile('(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')**protected** **static** **final** Pattern NON\_EMPTY = Pattern.compile(/[\S\s]+/)**protected** **static** **final** Pattern NON\_BLANK = Pattern.compile(/^\s\*\S[\S\s]\*/)**protected** **static** **final** Pattern ISO8601\_WITH\_OFFSET = Pattern.

compile(/([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.\d{1,6})?(Z|[+-][01]\d:[0-5]\d)/)

**protected** **static** Pattern anyOf(String... values) {

**return** Pattern.compile(values.collect({ "^$it\$" }).join("|"))

}

RegexProperty onlyAlphaUnicode() {

**return** **new** RegexProperty(ONLY\_ALPHA\_UNICODE).asString()

}

RegexProperty alphaNumeric() {

**return** **new** RegexProperty(ALPHA\_NUMERIC).asString()

}

RegexProperty number() {

**return** **new** RegexProperty(NUMBER).asDouble()

}

RegexProperty positiveInt() {

**return** **new** RegexProperty(POSITIVE\_INT).asInteger()

}

RegexProperty anyBoolean() {

**return** **new** RegexProperty(TRUE\_OR\_FALSE).asBooleanType()

}

RegexProperty anInteger() {

**return** **new** RegexProperty(INTEGER).asInteger()

}

RegexProperty aDouble() {

**return** **new** RegexProperty(DOUBLE).asDouble()

}

RegexProperty ipAddress() {

**return** **new** RegexProperty(IP\_ADDRESS).asString()

}

RegexProperty hostname() {

**return** **new** RegexProperty(HOSTNAME\_PATTERN).asString()

}

RegexProperty email() {

**return** **new** RegexProperty(EMAIL).asString()

}

RegexProperty url() {

**return** **new** RegexProperty(URL).asString()

}

RegexProperty httpsUrl() {

**return** **new** RegexProperty(HTTPS\_URL).asString()

}

RegexProperty uuid() {

**return** **new** RegexProperty(UUID).asString()

}

RegexProperty isoDate() {

**return** **new** RegexProperty(ANY\_DATE).asString()

}

RegexProperty isoDateTime() {

**return** **new** RegexProperty(ANY\_DATE\_TIME).asString()

}

RegexProperty isoTime() {

**return** **new** RegexProperty(ANY\_TIME).asString()

}

RegexProperty iso8601WithOffset() {

**return** **new** RegexProperty(ISO8601\_WITH\_OFFSET).asString()

}

RegexProperty nonEmpty() {

**return** **new** RegexProperty(NON\_EMPTY).asString()

}

RegexProperty nonBlank() {

**return** **new** RegexProperty(NON\_BLANK).asString()

}

在您的合同中，可以按以下示例所示使用它：

Contract dslWithOptionalsInString = Contract.make {

priority 1

request {

method POST()

url '/users/password'

headers {

contentType(applicationJson())

}

body(

email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),

callback\_url: $(consumer(regex(hostname())), producer('http://partners.com'))

)

}

response {

status 404

headers {

contentType(applicationJson())

}

body(

code: value(consumer("123123"), producer(optional("123123"))),

message: "User not found by email = [${value(producer(regex(email())), consumer('not.existing@user.com'))}]"

)

}

}

为了使事情变得更加简单，您可以使用一组预定义的对象，这些对象将自动假定您要传递正则表达式。所有这些方法均以any前缀开头：

T anyAlphaUnicode()

T anyAlphaNumeric()

T anyNumber()

T anyInteger()

T anyPositiveInt()

T anyDouble()

T anyHex()

T aBoolean()

T anyIpAddress()

T anyHostname()

T anyEmail()

T anyUrl()

T anyHttpsUrl()

T anyUuid()

T anyDate()

T anyDateTime()

T anyTime()

T anyIso8601WithOffset()

T anyNonBlankString()

T anyNonEmptyString()

T anyOf(String... values)

这是如何引用这些方法的示例：

Contract contractDsl = Contract.make {

label 'trigger\_event'

input {

triggeredBy('toString()')

}

outputMessage {

sentTo 'topic.rateablequote'

body([

alpha : $(anyAlphaUnicode()),

number : $(anyNumber()),

anInteger : $(anyInteger()),

positiveInt : $(anyPositiveInt()),

aDouble : $(anyDouble()),

aBoolean : $(aBoolean()),

ip : $(anyIpAddress()),

hostname : $(anyHostname()),

email : $(anyEmail()),

url : $(anyUrl()),

httpsUrl : $(anyHttpsUrl()),

uuid : $(anyUuid()),

date : $(anyDate()),

dateTime : $(anyDateTime()),

time : $(anyTime()),

iso8601WithOffset: $(anyIso8601WithOffset()),

nonBlankString : $(anyNonBlankString()),

nonEmptyString : $(anyNonEmptyString()),

anyOf : $(anyOf('foo', 'bar'))

])

}

}

### **94.5.3传递可选参数**

|  |  |
| --- | --- |
| IMG_569 | **重要** |
| 本部分仅对Groovy DSL有效。请查看 [第94.5.7节“匹配器节中的动态Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-matchers" \o "94.5.7匹配器部分中的动态属性)一节，以获取类似功能的YAML示例。 |

可以在合同中提供可选参数。但是，只能为以下项提供可选参数：

* 请求的 STUB端
* 回应的测试方

以下示例显示了如何提供可选参数：

org.springframework.cloud.contract.spec.Contract.make {

priority 1

request {

method 'POST'

url '/users/password'

headers {

contentType(applicationJson())

}

body(

email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),

callback\_url: $(consumer(regex(hostname())), producer('https://partners.com'))

)

}

response {

status 404

headers {

header 'Content-Type': 'application/json'

}

body(

code: value(consumer("123123"), producer(optional("123123")))

)

}

}

通过使用optional()方法包装正文的一部分，可以创建必须存在0次或多次的正则表达式。

如果您将Spock用于其中，则将从上一个示例生成以下测试：

""" given:

def request = given()

.header("Content-Type", "application/json")

.body('''{"email":"abc@abc.com","callback\_url":"https://partners.com"}''')

when:

def response = given().spec(request)

.post("/users/password")

then:

response.statusCode == 404

response.header('Content-Type') == 'application/json'

and:

DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).field("['code']").matches("(123123)?")"""

还将生成以下存根：

'''{

"request" : {

"url" : "/users/password",

"method" : "POST",

"bodyPatterns" : [ {

"matchesJsonPath" : "$[?(@.['email'] =~ /([a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,6})?/)]"

}, {

"matchesJsonPath" : "$[?(@.['callback\_url'] =~ /((http[s]?|ftp):\\\\/)\\\\/?([^:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"

} ],

"headers" : {

"Content-Type" : {

"equalTo" : "application/json"

}

}

},

"response" : {

"status" : 404,

"body" : "{\\"code\\":\\"123123\\",\\"message\\":\\"User not found by email == [not.existing@user.com]\\"}",

"headers" : {

"Content-Type" : "application/json"

}

},

"priority" : 1

}'''

### **94.5.4在服务器端执行自定义方法**

|  |  |
| --- | --- |
| IMG_570 | **重要** |
| 本部分仅对Groovy DSL有效。请查看 [第94.5.7节“匹配器节中的动态Properties”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-matchers" \o "94.5.7匹配器部分中的动态属性)一节，以获取类似功能的YAML示例。 |

您可以定义在测试期间在服务器端执行的方法调用。可以将这种方法添加到配置中定义为“ baseClassForTests”的类中。以下代码显示了测试用例的合同部分的示例：

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url $(consumer(regex('^/api/[0-9]{2}$')), producer('/api/12'))

headers {

header 'Content-Type': 'application/json'

}

body '''\ [{

"text": "Gonna see you at Warsaw"

}]

''' }

response {

body(

path: $(consumer('/api/12'), producer(regex('^/api/[0-9]{2}$'))),

correlationId: $(consumer('1223456'), producer(execute('isProperCorrelationId($it)')))

)

status OK()

}

}

以下代码显示了测试用例的基类部分：

**abstract** **class** BaseMockMvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** PairIdController())

}

**void** isProperCorrelationId(Integer correlationId) {

assert correlationId == 123456

}

**void** isEmpty(String value) {

assert value == null

}

}

|  |  |
| --- | --- |
| IMG_571 | **重要** |
| 您不能同时使用String和execute来执行串联。例如，呼叫header('Authorization', 'Bearer ' + execute('authToken()'))会导致不正确的结果。而是调用header('Authorization', execute('authToken()'))并确保authToken()方法返回您需要的所有内容。 |

从JSON读取的对象的类型可以是以下之一，具体取决于JSON路径：

* String：如果您指向JSON中的String值。
* JSONArray：如果您指向JSON中的List。
* Map：如果您指向JSON中的Map。
* Number：如果您指向JSON中的Integer，Double等。
* Boolean：如果您指向JSON中的Boolean。

在合同的请求部分，您可以指定body应该从方法中获取。

|  |  |
| --- | --- |
| IMG_572 | **重要** |
| 您必须同时提供消费者和生产方。execute部分适用于整个身体-不适用于部分身体。 |

以下示例显示如何从JSON读取对象：

Contract contractDsl = Contract.make {

request {

method 'GET'

url '/something'

body(

$(c('foo'), p(execute('hashCode()')))

)

}

response {

status OK()

}

}

前面的示例导致在请求正文中调用hashCode()方法。它应类似于以下代码：

*// given:*

MockMvcRequestSpecification request = given()

.body(hashCode());

*// when:*

ResponseOptions response = given().spec(request)

.get("/something");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

### **94.5.5引用响应中的请求**

最好的情况是提供固定值，但是有时您需要在响应中引用一个请求。

如果您使用Groovy DSL编写合同，则可以使用fromRequest()方法，该方法使您可以从HTTP请求中引用一堆元素。您可以使用以下选项：

* fromRequest().url()：返回请求URL和查询参数。
* fromRequest().query(String key)：返回具有给定名称的第一个查询参数。
* fromRequest().query(String key, int index)：返回具有给定名称的第n个查询参数。
* fromRequest().path()：返回完整路径。
* fromRequest().path(int index)：返回第n个路径元素。
* fromRequest().header(String key)：返回具有给定名称的第一个标头。
* fromRequest().header(String key, int index)：返回具有给定名称的第n个标题。
* fromRequest().body()：返回完整的请求正文。
* fromRequest().body(String jsonPath)：从请求中返回与JSON路径匹配的元素。

如果您使用的是YAML合同定义，则必须使用带有自定义Spring Cloud Contract函数的 [Handlebars](https://handlebarsjs.com/" \t "https://www.springcloud.cc/_top) {{{ }}}符号来实现此目的。

* {{{ request.url }}}：返回请求URL和查询参数。
* {{{ request.query.key.[index] }}}：返回具有给定名称的第n个查询参数。例如，键foo的第一个条目{{{ request.query.foo.[0] }}}
* {{{ request.path }}}：返回完整路径。
* {{{ request.path.[index] }}}：返回第n个路径元素。例如，首次输入` {{{request.path。[0] }}}
* {{{ request.headers.key }}}：返回具有给定名称的第一个标头。
* {{{ request.headers.key.[index] }}}：返回具有给定名称的第n个标头。
* {{{ request.body }}}：返回完整的请求正文。
* {{{ jsonpath this 'your.json.path' }}}：从请求中返回与JSON路径匹配的元素。例如json路径$.foo-{{{ jsonpath this '$.foo' }}}

考虑以下合同：

**Groovy DSL。**

**YAML。**

request:

method: GET

url: /api/v1/xxxx

queryParameters:

foo:

- bar

- bar2

headers:

Authorization:

- secret

- secret2

body:

foo: bar

baz: 5

response:

status: 200

headers:

Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"

body:

url: "{{{ request.url }}}"

path: "{{{ request.path }}}"

pathIndex: "{{{ request.path.1 }}}"

param: "{{{ request.query.foo }}}"

paramIndex: "{{{ request.query.foo.1 }}}"

authorization: "{{{ request.headers.Authorization.0 }}}"

authorization2: "{{{ request.headers.Authorization.1 }}"

fullBody: "{{{ request.body }}}"

responseFoo: "{{{ jsonpath this '$.foo' }}}"

responseBaz: "{{{ jsonpath this '$.baz' }}}"

responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"

运行JUnit测试生成将导致类似于以下示例的测试：

*// given:*

MockMvcRequestSpecification request = given()

.header("Authorization", "secret")

.header("Authorization", "secret2")

.body("{\"foo\":\"bar\",\"baz\":5}");

*// when:*

ResponseOptions response = given().spec(request)

.queryParam("foo","bar")

.queryParam("foo","bar2")

.get("/api/v1/xxxx");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Authorization")).isEqualTo("foo secret bar");*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fullBody']").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");

assertThatJson(parsedJson).field("['authorization']").isEqualTo("secret");

assertThatJson(parsedJson).field("['authorization2']").isEqualTo("secret2");

assertThatJson(parsedJson).field("['path']").isEqualTo("/api/v1/xxxx");

assertThatJson(parsedJson).field("['param']").isEqualTo("bar");

assertThatJson(parsedJson).field("['paramIndex']").isEqualTo("bar2");

assertThatJson(parsedJson).field("['pathIndex']").isEqualTo("v1");

assertThatJson(parsedJson).field("['responseBaz']").isEqualTo(5);

assertThatJson(parsedJson).field("['responseFoo']").isEqualTo("bar");

assertThatJson(parsedJson).field("['url']").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2");

assertThatJson(parsedJson).field("['responseBaz2']").isEqualTo("Bla bla bar bla bla");

如您所见，响应中已正确引用了请求中的元素。

生成的WireMock存根应类似于以下示例：

**{**

"request" : **{**

"urlPath" : "/api/v1/xxxx"**,**

"method" : "POST"**,**

"headers" : **{**

"Authorization" : **{**

"equalTo" : "secret2"

**}**

**},**

"queryParameters" : **{**

"foo" : **{**

"equalTo" : "bar2"

**}**

**},**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$[?(@.['baz'] == 5)]"

**},** **{**

"matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "{\"authorization\":\"{{{request.headers.Authorization.[0]}}}\",\"path\":\"{{{request.path}}}\",\"responseBaz\":{{{jsonpath this '$.baz'}}} ,\"param\":\"{{{request.query.foo.[0]}}}\",\"pathIndex\":\"{{{request.path.[1]}}}\",\"responseBaz2\":\"Bla bla {{{jsonpath this '$.foo'}}} bla bla\",\"responseFoo\":\"{{{jsonpath this '$.foo'}}}\",\"authorization2\":\"{{{request.headers.Authorization.[1]}}}\",\"fullBody\":\"{{{escapejsonbody}}}\",\"url\":\"{{{request.url}}}\",\"paramIndex\":\"{{{request.query.foo.[1]}}}\"}"**,**

"headers" : **{**

"Authorization" : "{{{request.headers.Authorization.[0]}}};foo"

**},**

"transformers" : **[** "response-template" **]**

**}}**

发送请求（例如合同的request部分中提出的请求）会导致发送以下响应正文：

**{**

"url" : "/api/v1/xxxx?foo=bar&foo=bar2"**,**

"path" : "/api/v1/xxxx"**,**

"pathIndex" : "v1"**,**

"param" : "bar"**,**

"paramIndex" : "bar2"**,**

"authorization" : "secret"**,**

"authorization2" : "secret2"**,**

"fullBody" : "{\"foo\":\"bar\",\"baz\":5}"**,**

"responseFoo" : "bar"**,**

"responseBaz" : 5**,**

"responseBaz2" : "Bla bla bar bla bla"**}**

|  |  |
| --- | --- |
| IMG_573 | **重要** |
| 此功能仅适用于版本大于或等于2.5.1的WireMock。Spring Cloud Contract验证程序使用WireMock的response-template响应转换器。它使用把手将“ Mustache {{{ }}}”模板转换为适当的值。此外，它注册了两个帮助程序功能： |

* escapejsonbody：以可以嵌入JSON的格式转义请求正文。
* jsonpath：对于给定的参数，在请求正文中找到一个对象。

### **94.5.6注册自己的WireMock扩展**

WireMock允许您注册自定义扩展。默认情况下，Spring Cloud Contract注册该转换器，使您可以引用响应中的请求。如果要提供自己的扩展，则可以注册org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions接口的实现。由于我们使用spring.factories扩展方法，因此可以在META-INF/spring.factories文件中创建一个类似于以下内容的条目：

org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\

org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtensions

org.springframework.cloud.contract.spec.ContractConverter=\

org.springframework.cloud.contract.stubrunner.TestCustomYamlContractConverter

以下是自定义扩展的示例：

**TestWireMockExtensions.groovy。**

*/\**

*\* Copyright 2013-2019 the original author or authors.*

*\**

*\* Licensed under the Apache License, Version 2.0 (the "License");*

*\* you may not use this file except in compliance with the License.*

*\* You may obtain a copy of the License at*

*\**

*\* https://www.apache.org/licenses/LICENSE-2.0*

*\**

*\* Unless required by applicable law or agreed to in writing, software*

*\* distributed under the License is distributed on an "AS IS" BASIS,*

*\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.*

*\* See the License for the specific language governing permissions and*

*\* limitations under the License.*

*\*/*

**package** org.springframework.cloud.contract.verifier.dsl.wiremock

**import** com.github.tomakehurst.wiremock.extension.Extension

**/\*\***

**\* Extension that registers the default transformer and the custom one**

**\*/class** TestWireMockExtensions **implements** WireMockExtensions {

@Override

List<Extension> extensions() {

**return** [

**new** DefaultResponseTransformer(),

**new** CustomExtension()

]

}

}

**class** CustomExtension **implements** Extension {

@Override

String getName() {

**return** "foo-transformer"

}

}

|  |  |
| --- | --- |
| IMG_574 | **重要** |
| 如果要将转换仅应用于明确需要它的映射，请记住重写applyGlobally()方法并将其设置为false。 |

### **94.5.7“匹配器”部分中的动态Properties**

如果您使用[Pact](https://docs.pact.io/" \t "https://www.springcloud.cc/_top)，那么以下讨论可能看起来很熟悉。很少有用户习惯于在主体之间进行分隔并设置合同的动态部分。

您可以使用bodyMatchers部分有两个原因：

* 定义应该以存根结尾的动态值。您可以在合同的request或inputMessage部分中进行设置。
* 验证测试结果。本部分位于合同的response或outputMessage中。

当前，Spring Cloud Contract验证程序仅支持具有以下匹配可能性的基于JSON路径的匹配器：

**Groovy DSL**

对于存根（在消费者方面的测试中）：

* + byEquality()：通过提供的JSON路径从消费者请求中获取的值必须等于合同中提供的值。
  + byRegex(…​)：通过提供的JSON路径从消费者请求中获取的值必须与正则表达式匹配。您还可以传递期望的匹配值的类型（例如asString()，asLong()等）
  + byDate()：通过提供的JSON路径从消费者请求中获取的值必须与ISO日期值的正则表达式匹配。
  + byTimestamp()：通过提供的JSON路径从消费者请求中获取的值必须与ISO DateTime值的正则表达式匹配。
  + byTime()：通过提供的JSON路径从消费者请求中获取的值必须与ISO时间值的正则表达式匹配。

进行验证（在生产者方生成的测试中）：

* + byEquality()：通过提供的JSON路径从生产者的响应中获取的值必须等于合同中提供的值。
  + byRegex(…​)：通过提供的JSON路径从生产者的响应中获取的值必须与正则表达式匹配。
  + byDate()：通过提供的JSON路径从生产者的响应中获取的值必须与ISO日期值的正则表达式匹配。
  + byTimestamp()：通过提供的JSON路径从生产者的响应中获取的值必须与ISO DateTime值的正则表达式匹配。
  + byTime()：通过提供的JSON路径从生产者的响应中获取的值必须与ISO时间值的正则表达式匹配。
  + byType()：通过提供的JSON路径从生产者的响应中获取的值必须与合同中的响应主体中定义的类型相同。byType可以关闭，您可以在其中设置minOccurrence和maxOccurrence。对于请求端，应该使用闭包声明集合的大小。这样，您可以声明展平集合的大小。要检查未展平的集合的大小，请对byCommand(…​) testMatcher使用自定义方法。

byCommand(…​)：通过提供的JSON路径从生产者的响应中获取的值作为输入传递到您提供的自定义方法。例如，byCommand('foo($it)')导致调用foo方法，与JSON路径匹配的值将传递到该方法。从JSON读取的对象的类型可以是以下之一，具体取决于JSON路径：

* + - String：如果您指向String值。
    - JSONArray：如果指向List。
    - Map：如果指向Map。
    - Number：如果指向Integer，Double或其他类型的数字。
    - Boolean：如果指向Boolean。
  + byNull()：通过提供的JSON路径从响应中获取的值必须为null

**YAML。**请阅读Groovy部分，详细了解类型的含义

对于YAML，匹配器的结构如下所示

- path: $.foo

type: by\_regex

value: bar

regexType: as\_string

或者，如果您要使用预定义的正则表达式之一[only\_alpha\_unicode, number, any\_boolean, ip\_address, hostname, email, url, uuid, iso\_date, iso\_date\_time, iso\_time, iso\_8601\_with\_offset, non\_empty, non\_blank]：

- path: $.foo

type: by\_regex

predefined: only\_alpha\_unicode

在下面，您可以找到允许的“类型”列表。

对于stubMatchers：

* + by\_equality
  + by\_regex
  + by\_date
  + by\_timestamp
  + by\_time

by\_type

* + - 还有2个其他字段被接受：minOccurrence和maxOccurrence。

对于testMatchers：

* + by\_equality
  + by\_regex
  + by\_date
  + by\_timestamp
  + by\_time

by\_type

* + - 还有2个其他字段：minOccurrence和maxOccurrence。
  + by\_command
  + by\_null

您还可以通过regexType字段定义正则表达式对应的类型。在下面，您可以找到允许的正则表达式类型列表：

* as\_integer
* as\_double
* as\_float，
* 只要
* as\_short
* as\_boolean
* as\_string

考虑以下示例：

**Groovy DSL。**

Contract contractDsl = Contract.make {

request {

method 'GET'

urlPath '/get'

body([

duck : 123,

alpha : 'abc',

number : 123,

aBoolean : true,

date : '2017-01-01',

dateTime : '2017-01-01T01:23:45',

time : '01:02:34',

valueWithoutAMatcher: 'foo',

valueWithTypeMatch : 'string',

key : [

'complex.key': 'foo'

]

])

bodyMatchers {

jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())

jsonPath('$.duck', byEquality())

jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())

jsonPath('$.alpha', byEquality())

jsonPath('$.number', byRegex(number()).asInteger())

jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())

jsonPath('$.date', byDate())

jsonPath('$.dateTime', byTimestamp())

jsonPath('$.time', byTime())

jsonPath("\$.['key'].['complex.key']", byEquality())

}

headers {

contentType(applicationJson())

}

}

response {

status OK()

body([

duck : 123,

alpha : 'abc',

number : 123,

positiveInteger : 1234567890,

negativeInteger : -1234567890,

positiveDecimalNumber: 123.4567890,

negativeDecimalNumber: -123.4567890,

aBoolean : true,

date : '2017-01-01',

dateTime : '2017-01-01T01:23:45',

time : "01:02:34",

valueWithoutAMatcher : 'foo',

valueWithTypeMatch : 'string',

valueWithMin : [

1, 2, 3

],

valueWithMax : [

1, 2, 3

],

valueWithMinMax : [

1, 2, 3

],

valueWithMinEmpty : [],

valueWithMaxEmpty : [],

key : [

'complex.key': 'foo'

],

nullValue : null

])

bodyMatchers {

*// asserts the jsonpath value against manual regex*

jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())

*// asserts the jsonpath value against the provided value*

jsonPath('$.duck', byEquality())

*// asserts the jsonpath value against some default regex*

jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())

jsonPath('$.alpha', byEquality())

jsonPath('$.number', byRegex(number()).asInteger())

jsonPath('$.positiveInteger', byRegex(anInteger()).asInteger())

jsonPath('$.negativeInteger', byRegex(anInteger()).asInteger())

jsonPath('$.positiveDecimalNumber', byRegex(aDouble()).asDouble())

jsonPath('$.negativeDecimalNumber', byRegex(aDouble()).asDouble())

jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())

*// asserts vs inbuilt time related regex*

jsonPath('$.date', byDate())

jsonPath('$.dateTime', byTimestamp())

jsonPath('$.time', byTime())

*// asserts that the resulting type is the same as in response body*

jsonPath('$.valueWithTypeMatch', byType())

jsonPath('$.valueWithMin', byType {

*// results in verification of size of array (min 1)*

minOccurrence(1)

})

jsonPath('$.valueWithMax', byType {

*// results in verification of size of array (max 3)*

maxOccurrence(3)

})

jsonPath('$.valueWithMinMax', byType {

*// results in verification of size of array (min 1 & max 3)*

minOccurrence(1)

maxOccurrence(3)

})

jsonPath('$.valueWithMinEmpty', byType {

*// results in verification of size of array (min 0)*

minOccurrence(0)

})

jsonPath('$.valueWithMaxEmpty', byType {

*// results in verification of size of array (max 0)*

maxOccurrence(0)

})

*// will execute a method `assertThatValueIsANumber`*

jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))

jsonPath("\$.['key'].['complex.key']", byEquality())

jsonPath('$.nullValue', byNull())

}

headers {

contentType(applicationJson())

header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))

}

}

}

**YAML。**

request:

method: GET

urlPath: /get/1

headers:

Content-Type: application/json

cookies:

foo: 2

bar: 3

queryParameters:

limit: 10

offset: 20

filter: 'email'

sort: name

search: 55

age: 99

name: John.Doe

email: 'bob@email.com'

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

key:

"complex.key": 'foo'

nullValue: null

valueWithMin:

- 1

- 2

- 3

valueWithMax:

- 1

- 2

- 3

valueWithMinMax:

- 1

- 2

- 3

valueWithMinEmpty: []

valueWithMaxEmpty: []

matchers:

url:

regex: /get/[0-9]

# predefined:

# execute a method

#command: 'equals($it)'

queryParameters:

- key: limit

type: equal\_to

value: 20

- key: offset

type: containing

value: 20

- key: sort

type: equal\_to

value: name

- key: search

type: not\_matching

value: '^[0-9]{2}$'

- key: age

type: not\_matching

value: '^\\w\*$'

- key: name

type: matching

value: 'John.\*'

- key: hello

type: absent

cookies:

- key: foo

regex: '[0-9]'

- key: bar

command: 'equals($it)'

headers:

- key: Content-Type

regex: "application/json.\*"

body:

- path: $.duck

type: by\_regex

value: "[0-9]{3}"

- path: $.duck

type: by\_equality

- path: $.alpha

type: by\_regex

predefined: only\_alpha\_unicode

- path: $.alpha

type: by\_equality

- path: $.number

type: by\_regex

predefined: number

- path: $.aBoolean

type: by\_regex

predefined: any\_boolean

- path: $.date

type: by\_date

- path: $.dateTime

type: by\_timestamp

- path: $.time

type: by\_time

- path: "$.['key'].['complex.key']"

type: by\_equality

- path: $.nullvalue

type: by\_null

- path: $.valueWithMin

type: by\_type

minOccurrence: 1

- path: $.valueWithMax

type: by\_type

maxOccurrence: 3

- path: $.valueWithMinMax

type: by\_type

minOccurrence: 1

maxOccurrence: 3

response:

status: 200

cookies:

foo: 1

bar: 2

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

valueWithMin:

- 1

- 2

- 3

valueWithMax:

- 1

- 2

- 3

valueWithMinMax:

- 1

- 2

- 3

valueWithMinEmpty: []

valueWithMaxEmpty: []

key:

'complex.key': 'foo'

nulValue: null

matchers:

headers:

- key: Content-Type

regex: "application/json.\*"

cookies:

- key: foo

regex: '[0-9]'

- key: bar

command: 'equals($it)'

body:

- path: $.duck

type: by\_regex

value: "[0-9]{3}"

- path: $.duck

type: by\_equality

- path: $.alpha

type: by\_regex

predefined: only\_alpha\_unicode

- path: $.alpha

type: by\_equality

- path: $.number

type: by\_regex

predefined: number

- path: $.aBoolean

type: by\_regex

predefined: any\_boolean

- path: $.date

type: by\_date

- path: $.dateTime

type: by\_timestamp

- path: $.time

type: by\_time

- path: $.valueWithTypeMatch

type: by\_type

- path: $.valueWithMin

type: by\_type

minOccurrence: 1

- path: $.valueWithMax

type: by\_type

maxOccurrence: 3

- path: $.valueWithMinMax

type: by\_type

minOccurrence: 1

maxOccurrence: 3

- path: $.valueWithMinEmpty

type: by\_type

minOccurrence: 0

- path: $.valueWithMaxEmpty

type: by\_type

maxOccurrence: 0

- path: $.duck

type: by\_command

value: assertThatValueIsANumber($it)

- path: $.nullValue

type: by\_null

value: null

headers:

Content-Type: application/json

在前面的示例中，您可以在matchers部分中查看合同的动态部分。对于请求部分，您可以看到，对于valueWithoutAMatcher以外的所有字段，存根都应包含的正则表达式的值已明确设置。对于valueWithoutAMatcher，验证的方式与不使用匹配器的方式相同。在这种情况下，测试将执行相等性检查。

对于bodyMatchers部分中的响应端，我们以类似的方式定义动态部分。唯一的区别是byType匹配器也存在。验证程序引擎检查四个字段，以验证来自测试的响应是否具有与JSON路径匹配给定字段的值，与响应主体中定义的类型相同的类型，并通过以下检查（基于方法被调用）：

* 对于$.valueWithTypeMatch，引擎检查类型是否相同。
* 对于$.valueWithMin，引擎检查类型并断言大小是否大于或等于最小出现次数。
* 对于$.valueWithMax，引擎检查类型并断言大小是否小于或等于最大出现次数。
* 对于$.valueWithMinMax，引擎检查类型并断言大小是否在最小和最大出现之间。

生成的测试类似于以下示例（请注意，and部分将自动生成的断言和匹配器的断言分开）：

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/json")

.body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\",\"key\":{\"complex.key\":\"foo\"}}");

*// when:*

ResponseOptions response = given().spec(request)

.get("/get");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/json.\*");*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['valueWithoutAMatcher']").isEqualTo("foo");*// and:*

assertThat(parsedJson.read("$.duck", String.**class**)).matches("[0-9]{3}");

assertThat(parsedJson.read("$.duck", Integer.**class**)).isEqualTo(123);

assertThat(parsedJson.read("$.alpha", String.**class**)).matches("[\\p{L}]\*");

assertThat(parsedJson.read("$.alpha", String.**class**)).isEqualTo("abc");

assertThat(parsedJson.read("$.number", String.**class**)).matches("-?(\\d\*\\.\\d+|\\d+)");

assertThat(parsedJson.read("$.aBoolean", String.**class**)).matches("(true|false)");

assertThat(parsedJson.read("$.date", String.**class**)).matches("(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");

assertThat(parsedJson.read("$.dateTime", String.**class**)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");

assertThat(parsedJson.read("$.time", String.**class**)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");

assertThat((Object) parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.**class**);

assertThat((Object) parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin", java.util.Collection.**class**)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);

assertThat((Object) parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax", java.util.Collection.**class**)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);

assertThat((Object) parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax", java.util.Collection.**class**)).as("$.valueWithMinMax").hasSizeBetween(1, 3);

assertThat((Object) parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty", java.util.Collection.**class**)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEqualTo(0);

assertThat((Object) parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty", java.util.Collection.**class**)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqualTo(0);

assertThatValueIsANumber(parsedJson.read("$.duck"));

assertThat(parsedJson.read("$.['key'].['complex.key']", String.**class**)).isEqualTo("foo");

|  |  |
| --- | --- |
| IMG_575 | **重要** |
| 请注意，对于byCommand方法，该示例调用assertThatValueIsANumber。此方法必须在测试基类中定义或静态导入到测试中。请注意，byCommand调用已转换为assertThatValueIsANumber(parsedJson.read("$.duck"));。这意味着引擎采用了方法名称，并将正确的JSON路径作为参数传递给它。 |

在下面的示例中，生成的WireMock存根为：

'''**{**

"request" : **{**

"urlPath" : "/get"**,**

"method" : "POST"**,**

"headers" : **{**

"Content-Type" : **{**

"matches" : "application/json.\*"

**}**

**},**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"

**},** **{**

"matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"

**},** **{**

"matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"

**},** **{**

"matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with value')]"

**},** **{**

"matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] == 4)]"

**},** **{**

"matchesJsonPath" : "$[?(@.duck =~ /([0-9]{3})/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.duck == 123)]"

**},** **{**

"matchesJsonPath" : "$[?(@.alpha =~ /([\\\\p{L}]\*)/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.alpha == 'abc')]"

**},** **{**

"matchesJsonPath" : "$[?(@.number =~ /(-?(\\\\d\*\\\\.\\\\d+|\\\\d+))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.date =~ /((\\\\d\\\\d\\\\d\\\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01]))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"

**},** **{**

"matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.\*)/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.valueWithMin.size() >= 1)]"

**},** **{**

"matchesJsonPath" : "$[?(@.valueWithMax.size() <= 3)]"

**},** **{**

"matchesJsonPath" : "$[?(@.valueWithMinMax.size() >= 1 && @.valueWithMinMax.size() <= 3)]"

**},** **{**

"matchesJsonPath" : "$[?(@.valueWithOccurrence.size() >= 4 && @.valueWithOccurrence.size() <= 4)]"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "{\\"duck\\":123,\\"alpha\\":\\"abc\\",\\"number\\":123,\\"aBoolean\\":true,\\"date\\":\\"2017-01-01\\",\\"dateTime\\":\\"2017-01-01T01:23:45\\",\\"time\\":\\"01:02:34\\",\\"valueWithoutAMatcher\\":\\"foo\\",\\"valueWithTypeMatch\\":\\"string\\",\\"valueWithMin\\":[1,2,3],\\"valueWithMax\\":[1,2,3],\\"valueWithMinMax\\":[1,2,3],\\"valueWithOccurrence\\":[1,2,3,4]}"**,**

"headers" : **{**

"Content-Type" : "application/json"

**},**

"transformers" : **[** "response-template" **]**

**}}**'''

|  |  |
| --- | --- |
| IMG_576 | **重要** |
| 如果您使用matcher，则matcher用JSON路径寻址的请求和响应部分将从声明中删除。在验证集合的情况下，必须为集合的**所有**元素创建匹配器。 |

考虑以下示例：

Contract.make {

request {

method 'GET'

url("/foo")

}

response {

status OK()

body(events: [[

operation : 'EXPORT',

eventId : '16f1ed75-0bcc-4f0d-a04d-3121798faf99',

status : 'OK'

], [

operation : 'INPUT\_PROCESSING',

eventId : '3bb4ac82-6652-462f-b6d1-75e424a0024a',

status : 'OK'

]

]

)

bodyMatchers {

jsonPath('$.events[0].operation', byRegex('.+'))

jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))

jsonPath('$.events[0].status', byRegex('.+'))

}

}

}

前面的代码导致创建以下测试（代码块仅显示断言部分）：

and:

DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("16f1ed75-0bcc-4f0d-a04d-3121798faf99")

assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("EXPORT")

assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("INPUT\_PROCESSING")

assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("3bb4ac82-6652-462f-b6d1-75e424a0024a")

assertThatJson(parsedJson).array("['events']").contains("['status']").isEqualTo("OK")

and:

assertThat(parsedJson.read("\$.events[0].operation", String.**class**)).matches(".+")

assertThat(parsedJson.read("\$.events[0].eventId", String.**class**)).matches("^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})\$")

assertThat(parsedJson.read("\$.events[0].status", String.**class**)).matches(".+")

如您所见，断言的格式不正确。仅声明数组的第一个元素。为了解决此问题，您应该将断言应用于整个$.events集合，并使用byCommand(…​)方法进行断言。

## **94.6 JAX-RS支持**

Spring Cloud Contract验证程序支持JAX-RS 2客户端API。基类需要定义protected WebTarget webTarget和服务器初始化。测试JAX-RS API的唯一选项是启动web服务器。同样，带有主体的请求需要设置内容类型。否则，将使用默认值application/octet-stream。

为了使用JAX-RS模式，请使用以下设置：

testMode == 'JAXRSCLIENT'

以下示例显示了生成的测试API：

''' *// when:*

Response response = webTarget

.path("/users")

.queryParam("limit", "10")

.queryParam("offset", "20")

.queryParam("filter", "email")

.queryParam("sort", "name")

.queryParam("search", "55")

.queryParam("age", "99")

.queryParam("name", "Denis.Stepanov")

.queryParam("email", "bob@email.com")

.request()

.method("GET");

String responseAsString = response.readEntity(String.**class**);

*// then:*

assertThat(response.getStatus()).isEqualTo(200);

*// and:*

DocumentContext parsedJson = JsonPath.parse(responseAsString);

assertThatJson(parsedJson).field("['property1']").isEqualTo("a");'''

## **94.7异步支持**

如果您在服务器端使用异步通信（您的控制器返回Callable，DeferredResult，依此类推），那么在合同中，您必须在{10中提供一个async()方法/} 部分。以下代码显示了一个示例：

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

method GET()

url '/get'

}

response {

status OK()

body 'Passed'

async()

}

}

**YAML。**

response:

async: true

您还可以使用fixedDelayMilliseconds方法/属性来向存根添加延迟。

**Groovy DSL。**

org.springframework.cloud.contract.spec.Contract.make {

request {

method GET()

url '/get'

}

response {

status 200

body 'Passed'

fixedDelayMilliseconds 1000

}

}

**YAML。**

response:

fixedDelayMilliseconds: 1000

## **94.8使用上下文路径**

Spring Cloud Contract支持上下文路径。

|  |  |
| --- | --- |
| IMG_577 | **重要** |
| 完全支持上下文路径所需的唯一更改是**PRODUCER**端的开关 。另外，自动生成的测试必须使用**EXPLICIT**模式。消费者方面保持不变。为了使生成的测试通过，您必须使用**EXPLICIT** 模式。 |

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<testMode>EXPLICIT</testMode>

</configuration></plugin>

**Gradle.**

contracts {

testMode = 'EXPLICIT'

}

这样，您生成的测试**不**使用MockMvc。这意味着您要生成真实的请求，并且需要设置生成的测试的基类以在真实套接字上工作。

考虑以下合同：

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

url '/my-context-path/url'

}

response {

status OK()

}

}

下面的示例显示如何设置基类和“确保放心”：

**import** io.restassured.RestAssured;**import** org.junit.Before;**import** org.springframework.boot.web.server.LocalServerPort;**import** org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)**class** ContextPathTestingBaseClass {

@LocalServerPort **int** port;

@Before

**public** **void** setup() {

RestAssured.baseURI = "http://localhost";

RestAssured.port = **this**.port;

}

}

如果您这样做：

* 自动生成的测试中的所有请求都将发送到包含您的上下文路径的真实端点（例如/my-context-path/url）。
* 您的合同反映出您具有上下文路径。您生成的存根还具有该信息（例如，在存根中，您必须调用/my-context-path/url）。

## **94.9使用WebFlux**

Spring Cloud Contract提供了两种使用WebFlux的方法。

### **94.9.1使用WebTestClient的WebFlux**

其中之一是通过WebTestClient模式。

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<testMode>WEBTESTCLIENT</testMode>

</configuration></plugin>

**Gradle.**

contracts {

testMode = 'WEBTESTCLIENT'

}

以下示例显示如何为WebFlux设置WebTestClient基类和RestAssured：

**import** io.restassured.module.webtestclient.RestAssuredWebTestClient;**import** org.junit.Before;

**public** **abstract** **class** BeerRestBase {

@Before

**public** **void** setup() {

RestAssuredWebTestClient.standaloneSetup(

**new** ProducerController(personToCheck -> personToCheck.age >= 20));

}

}

}

### **94.9.2具有显式模式的WebFlux**

另一种方法是在生成的测试中使用EXPLICIT模式来使用WebFlux。

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<testMode>EXPLICIT</testMode>

</configuration></plugin>

**Gradle.**

contracts {

testMode = 'EXPLICIT'

}

以下示例显示如何为Web通量设置基类和“确保休息”：

@RunWith(SpringRunner.class)@SpringBootTest(classes = BeerRestBase.Config.class,

webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT,

properties = "server.port=0")**public** **abstract** **class** BeerRestBase {

*// your tests go here*

*// in this config class you define all controllers and mocked services*@Configuration@EnableAutoConfiguration**static** **class** Config {

@Bean

PersonCheckingService personCheckingService() {

**return** personToCheck -> personToCheck.age >= 20;

}

@Bean

ProducerController producerController() {

**return** **new** ProducerController(personCheckingService());

}

}

}

## **94.10对REST的XML支持**

对于REST合同，我们还支持XML请求和响应主体。XML主体必须作为String或GString在body元素内传递。还可以为请求和响应提供身体匹配器。代替jsonPath(…​)方法，应使用org.springframework.cloud.contract.spec.internal.BodyMatchers.xPath方法，并以所需的xPath作为第一个参数，并以适当的MatchingType作为第二个参数。支持byType()以外的所有主体匹配器。

这是带有XML响应主体的Groovy DSL合同的示例：

Contract.make {

request {

method GET()

urlPath '/get'

headers {

contentType(applicationXml())

}

}

response {

status(OK())

headers {

contentType(applicationXml())

}

body """<test>

<duck type='xtype'>123</duck>

<alpha>abc</alpha>

<list>

<elem>abc</elem>

<elem>def</elem>

<elem>ghi</elem>

</list>

<number>123</number>

<aBoolean>true</aBoolean>

<date>2017-01-01</date>

<dateTime>2017-01-01T01:23:45</dateTime>

<time>01:02:34</time>

<valueWithoutAMatcher>foo</valueWithoutAMatcher>

<key><complex>foo</complex></key>

</test>""" bodyMatchers {

xPath('/test/duck/text()', byRegex("[0-9]{3}"))

xPath('/test/duck/text()', byCommand('test($it)'))

xPath('/test/duck/xxx', byNull())

xPath('/test/duck/text()', byEquality())

xPath('/test/alpha/text()', byRegex(onlyAlphaUnicode()))

xPath('/test/alpha/text()', byEquality())

xPath('/test/number/text()', byRegex(number()))

xPath('/test/date/text()', byDate())

xPath('/test/dateTime/text()', byTimestamp())

xPath('/test/time/text()', byTime())

xPath('/test/\*/complex/text()', byEquality())

xPath('/test/duck/@type', byEquality())

}

}

}

以下是带有XML请求和响应主体的YAML合同的示例：

include::{verifier\_core\_path}/src/test/resources/yml/contract\_rest\_xml.yml

这是自动生成的XML响应正文测试的示例：

@Test**public** **void** validate\_xmlMatches() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/xml");

*// when:*

ResponseOptions response = given().spec(request).get("/get");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

*// and:*

DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance()

.newDocumentBuilder();

Document parsedXml = documentBuilder.parse(**new** InputSource(

**new** StringReader(response.getBody().asString())));

*// and:*

assertThat(valueFromXPath(parsedXml, "/test/list/elem/text()")).isEqualTo("abc");

assertThat(valueFromXPath(parsedXml,"/test/list/elem[2]/text()")).isEqualTo("def");

assertThat(valueFromXPath(parsedXml, "/test/duck/text()")).matches("[0-9]{3}");

assertThat(nodeFromXPath(parsedXml, "/test/duck/xxx")).isNull();

assertThat(valueFromXPath(parsedXml, "/test/alpha/text()")).matches("[\\p{L}]\*");

assertThat(valueFromXPath(parsedXml, "/test/\*/complex/text()")).isEqualTo("foo");

assertThat(valueFromXPath(parsedXml, "/test/duck/@type")).isEqualTo("xtype");

}

## **94.11消息传递顶级元素**

用于消息传递的DSL与专注于HTTP的DSL看起来有些不同。以下各节说明了差异：

* [第94.11.1节“由方法触发的输出”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-output-triggered-method" \o "94.11.1方法触发的输出)
* [第94.11.2节“消息触发的输出”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-output-triggered-message" \o "94.11.2由消息触发的输出)
* [第94.11.3节“消费者/生产者”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-consumer-producer" \o "94.11.3消费者/生产者)
* [第94.11.4节“公用”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "contract-dsl-common" \o "94.11.4共同的)

### **94.11.1方法触发的输出**

可以通过调用方法（例如启动a并发送消息时调用Scheduler）来触发输出消息，如以下示例所示：

**Groovy DSL。**

def dsl = Contract.make {

*// Human readable description*

description 'Some description'

*// Label by means of which the output message can be triggered*

label 'some\_label'

*// input to the contract*

input {

*// the contract will be triggered by a method*

triggeredBy('bookReturnedTriggered()')

}

*// output message of the contract*

outputMessage {

*// destination to which the output message will be sent*

sentTo('output')

*// the body of the output message*

body('''{ "bookName" : "foo" }''')

*// the headers of the output message*

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML。**

# Human readable description

description: Some description

# Label by means of which the output message can be triggered

label: some\_label

input:

# the contract will be triggered by a method

triggeredBy: bookReturnedTriggered()

# output message of the contract

outputMessage:

# destination to which the output message will be sent

sentTo: output

# the body of the output message

body:

bookName: foo

# the headers of the output message

headers:

BOOK-NAME: foo

在前面的示例情况下，如果执行了称为bookReturnedTriggered的方法，则输出消息将发送到output。在消息**发布者**方面，我们生成了一个测试，该测试调用该方法来触发消息。在**使用者**方面，您可以使用some\_label来触发消息。

### **94.11.2由消息触发的输出**

可以通过接收消息来触发输出消息，如以下示例所示：

**Groovy DSL。**

def dsl = Contract.make {

description 'Some Description'

label 'some\_label'

*// input is a message*

input {

*// the message was received from this destination*

messageFrom('input')

*// has the following body*

messageBody([

bookName: 'foo'

])

*// and the following headers*

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML。**

# Human readable description

description: Some description

# Label by means of which the output message can be triggered

label: some\_label

# input is a message

input:

messageFrom: input

# has the following body

messageBody:

bookName: 'foo'

# and the following headers

messageHeaders:

sample: 'header'

# output message of the contract

outputMessage:

# destination to which the output message will be sent

sentTo: output

# the body of the output message

body:

bookName: foo

# the headers of the output message

headers:

BOOK-NAME: foo

在前面的示例中，如果在input目标上收到了适当的消息，则将输出消息发送到output。在消息**发布者**侧，引擎生成一个测试，将该输入消息发送到定义的目的地。在 **使用者**方面，您可以将消息发送到输入目标，也可以使用标签（示例中为some\_label）来触发消息。

### **94.11.3消费者/生产者**

|  |  |
| --- | --- |
| IMG_578 | **重要** |
| 本部分仅对Groovy DSL有效。 |

在HTTP中，您使用的符号是client / stub and `server / test。您也可以在消息传递中使用这些范例。此外，Spring Cloud Contract验证程序还提供了consumer和producer方法，如以下示例所示（请注意，您可以使用$或value方法来提供consumer和producer部分）：

Contract.make {

label 'some\_label'

input {

messageFrom value(consumer('jms:output'), producer('jms:input'))

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo $(consumer('jms:input'), producer('jms:output'))

body([

bookName: 'foo'

])

}

}

### **94.11.4共同的**

在input或outputMessage部分中，可以使用在基类或静态导入中定义的method（例如assertThatMessageIsOnTheQueue()）的名称调用assertThat。Spring Cloud Contract将在生成的测试中执行该方法。

## **94.12在一个文件中多个Contracts**

您可以在一个文件中定义多个合同。这样的合同可能类似于以下示例：

**Groovy DSL。**

**import** org.springframework.cloud.contract.spec.Contract

[

Contract.make {

name("should post a user")

request {

method 'POST'

url('/users/1')

}

response {

status OK()

}

},

Contract.make {

request {

method 'POST'

url('/users/2')

}

response {

status OK()

}

}

]

**YAML。**

---

name: should post a user

request:

method: POST

url: /users/1

response:

status: 200

---

request:

method: POST

url: /users/2

response:

status: 200

---

request:

method: POST

url: /users/3

response:

status: 200

在前面的示例中，一个合同具有name字段，而另一个则没有。这导致生成两个看起来或多或少像这样的测试：

**package** org.springframework.cloud.contract.verifier.tests.com.hello;

**import** com.example.TestBase;**import** com.jayway.jsonpath.DocumentContext;**import** com.jayway.jsonpath.JsonPath;**import** com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;**import** com.jayway.restassured.response.ResponseOptions;**import** org.junit.Test;

**import** **static** com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.\*;**import** **static** com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;**import** **static** org.assertj.core.api.Assertions.assertThat;

**public** **class** V1Test **extends** TestBase {

@Test

**public** **void** validate\_should\_post\_a\_user() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given();

*// when:*

ResponseOptions response = given().spec(request)

.post("/users/1");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

}

@Test

**public** **void** validate\_withList\_1() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given();

*// when:*

ResponseOptions response = given().spec(request)

.post("/users/2");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

}

}

请注意，对于具有name字段的合同，生成的测试方法名为validate\_should\_post\_a\_user。对于一个没有名称的名称，它称为validate\_withList\_1。它对应于文件WithList.groovy的名称以及列表中合同的索引。

下例显示了生成的存根：

should post a user.json

1\_WithList.json

如您所见，第一个文件从合同中获取了name参数。第二个名称带有合同文件名称（WithList.groovy），并带有索引前缀（在这种情况下，合同在文件中合同列表中的索引为1）。

|  |
| --- |
| IMG_579 |
| 如您所见，命名合同会更好，因为这样做会使您的测试更有意义。 |

## **94.13从合同中生成Spring REST文档片段**

当您想使用Spring REST Docs包含API的请求和响应时，如果使用MockMvc和RestAssuredMockMvc，则只需对设置进行一些细微更改。如果还没有，只需包括以下依赖项。

**Maven.**

<dependency>

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

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope></dependency><dependency>

<groupId>org.springframework.restdocs</groupId>

<artifactId>spring-restdocs-mockmvc</artifactId>

<optional>true</optional></dependency>

**Gradle.**

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

testCompile 'org.springframework.restdocs:spring-restdocs-mockmvc'

接下来，您需要对基类进行一些更改，例如以下示例。

**package** com.example.fraud;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;**import** org.junit.Before;**import** org.junit.Rule;**import** org.junit.rules.TestName;**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;**import** org.springframework.boot.test.context.SpringBootTest;**import** org.springframework.restdocs.JUnitRestDocumentation;**import** org.springframework.test.context.junit4.SpringRunner;**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;**import** org.springframework.web.context.WebApplicationContext;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

@RunWith(SpringRunner.class)@SpringBootTest(classes = Application.class)**public** **abstract** **class** FraudBaseWithWebAppSetup {

**private** **static** **final** String OUTPUT = "target/generated-snippets";

@Rule

**public** JUnitRestDocumentation restDocumentation = **new** JUnitRestDocumentation(OUTPUT);

@Rule

**public** TestName testName = **new** TestName();

@Autowired

**private** WebApplicationContext context;

@Before

**public** **void** setup() {

RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(**this**.context)

.apply(documentationConfiguration(**this**.restDocumentation))

.alwaysDo(document(

getClass().getSimpleName() + "\_" + testName.getMethodName()))

.build());

}

**protected** **void** assertThatRejectionReasonIsNull(Object rejectionReason) {

assert rejectionReason == null;

}

}

如果您使用独立安装程序，则可以这样设置RestAssuredMockMvc：

**package** com.example.fraud;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;**import** org.junit.Before;**import** org.junit.Rule;**import** org.junit.rules.TestName;

**import** org.springframework.restdocs.JUnitRestDocumentation;**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

**public** **abstract** **class** FraudBaseWithStandaloneSetup {

**private** **static** **final** String OUTPUT = "target/generated-snippets";

@Rule

**public** JUnitRestDocumentation restDocumentation = **new** JUnitRestDocumentation(OUTPUT);

@Rule

**public** TestName testName = **new** TestName();

@Before

**public** **void** setup() {

RestAssuredMockMvc.standaloneSetup(MockMvcBuilders

.standaloneSetup(**new** FraudDetectionController())

.apply(documentationConfiguration(**this**.restDocumentation))

.alwaysDo(document(

getClass().getSimpleName() + "\_" + testName.getMethodName())));

}

}

|  |
| --- |
| IMG_580 |
| 从Spring REST Docs的1.2.0.RELEASE版本开始，您无需为生成的代码片段指定输出目录。 |

## **95.定制化**

|  |  |
| --- | --- |
| IMG_581 | **重要** |
| 本部分仅对Groovy DSL有效 |

您可以通过扩展DSL来定制Spring Cloud Contract验证程序，如本节其余部分所示。

## **95.1扩展DSL**

您可以为DSL提供自己的功能。此功能的关键要求是保持静态兼容性。在本文档的后面，您可以看到以下示例：

* 使用可重用的类创建一个JAR。
* 在DSL中引用这些类。

您可以在[此处](https://github.com/spring-cloud-samples/spring-cloud-contract-samples" \t "https://www.springcloud.cc/_top)找到完整的示例 。

### **95.1.1通用JAR**

以下示例显示了可以在DSL中重用的三个类。

**PatternUtils**包含**消费者**和**生产者**都使用的函数。

**package** com.example;

**import** java.util.regex.Pattern;

**/\*\***

**\* If you want to use {@link Pattern} directly in your tests**

**\* then you can create a class resembling this one. It can**

**\* contain all the {@link Pattern} you want to use in the DSL.**

**\***

**\* <pre>**

**\* {@code**

**\* request {**

**\* body(**

**\* [ age: $(c(PatternUtils.oldEnough()))]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* Notice that we're using both {@code $()} for dynamic values**

**\* and {@code c()} for the consumer side.**

**\***

**\* @author Marcin Grzejszczak**

**\*/***//tag::impl[]***public** **class** PatternUtils {

**public** **static** String tooYoung() {

*//remove::start[]*

**return** "[0-1][0-9]";

*//remove::end[return]*

}

**public** **static** Pattern oldEnough() {

*//remove::start[]*

**return** Pattern.compile("[2-9][0-9]");

*//remove::end[return]*

}

**/\*\***

**\* Makes little sense but it's just an example ;)**

**\*/**

**public** **static** Pattern ok() {

*//remove::start[]*

**return** Pattern.compile("OK");

*//remove::end[return]*

}

}*//end::impl[]*

**ConsumerUtils**包含由使用功能的**消费者**。

**package** com.example;

**import** org.springframework.cloud.contract.spec.internal.ClientDslProperty;

**/\*\***

**\* DSL Properties passed to the DSL from the consumer's perspective.**

**\* That means that on the input side {@code Request} for HTTP**

**\* or {@code Input} for messaging you can have a regular expression.**

**\* On the {@code Response} for HTTP or {@code Output} for messaging**

**\* you have to have a concrete value.**

**\***

**\* @author Marcin Grzejszczak**

**\*/***//tag::impl[]***public** **class** ConsumerUtils {

**/\*\***

**\* Consumer side property. By using the {@link ClientDslProperty}**

**\* you can omit most of boilerplate code from the perspective**

**\* of dynamic values. Example**

**\***

**\* <pre>**

**\* {@code**

**\* request {**

**\* body(**

**\* [ age: $(ConsumerUtils.oldEnough())]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* That way it's in the implementation that we decide what value we will pass to the consumer**

**\* and which one to the producer.**

**\***

**\* @author Marcin Grzejszczak**

**\*/**

**public** **static** ClientDslProperty oldEnough() {

*//remove::start[]*

*// this example is not the best one and*

*// theoretically you could just pass the regex instead of `ServerDslProperty` but*

*// it's just to show some new tricks :)*

**return** **new** ClientDslProperty(PatternUtils.oldEnough(), 40);

*//remove::end[return]*

}

}*//end::impl[]*

**ProducerUtils**包含由使用的功能**制片人**。

**package** com.example;

**import** org.springframework.cloud.contract.spec.internal.ServerDslProperty;

**/\*\***

**\* DSL Properties passed to the DSL from the producer's perspective.**

**\* That means that on the input side {@code Request} for HTTP**

**\* or {@code Input} for messaging you have to have a concrete value.**

**\* On the {@code Response} for HTTP or {@code Output} for messaging**

**\* you can have a regular expression.**

**\***

**\* @author Marcin Grzejszczak**

**\*/***//tag::impl[]***public** **class** ProducerUtils {

**/\*\***

**\* Producer side property. By using the {@link ProducerUtils}**

**\* you can omit most of boilerplate code from the perspective**

**\* of dynamic values. Example**

**\***

**\* <pre>**

**\* {@code**

**\* response {**

**\* body(**

**\* [ status: $(ProducerUtils.ok())]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* That way it's in the implementation that we decide what value we will pass to the consumer**

**\* and which one to the producer.**

**\*/**

**public** **static** ServerDslProperty ok() {

*// this example is not the best one and*

*// theoretically you could just pass the regex instead of `ServerDslProperty` but*

*// it's just to show some new tricks :)*

**return** **new** ServerDslProperty( PatternUtils.ok(), "OK");

}

}*//end::impl[]*

### **95.1.2在项目中添加依赖项**

为了使插件和IDE能够引用常见的JAR类，您需要将依赖项传递给您的项目。

### **95.1.3在项目的依存关系中测试依存关系**

首先，添加通用jar依赖项作为测试依赖项。因为您的合同文件在测试资源路径上可用，所以常见的jar类在您的Groovy文件中自动变为可见。以下示例显示如何测试依赖关系：

**Maven.**

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-common</artifactId>

<version>${project.version}</version>

<scope>test</scope></dependency>

**Gradle.**

testCompile("com.example:beer-common:0.0.1.BUILD-SNAPSHOT")

### **95.1.4在插件的依赖关系中测试依赖关系**

现在，您必须添加插件的依赖关系，以便在运行时重用，如以下示例所示：

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example</packageWithBaseClasses>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*intoxication.\*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

</configuration>

<dependencies>

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-common</artifactId>

<version>${project.version}</version>

<scope>compile</scope>

</dependency>

</dependencies></plugin>

**Gradle.**

classpath "com.example:beer-common:0.0.1.BUILD-SNAPSHOT"

### **95.1.5 DSL中的引用类**

现在，您可以在DSL中引用您的类，如以下示例所示：

**package** contracts.beer.rest

**import** com.example.ConsumerUtils**import** com.example.ProducerUtils**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

description("""Represents a successful scenario of getting a beer

```

given:

client is old enough

when:

he applies **for** a beer

then:

we'll grant him the beer```

""") request {

method 'POST'

url '/check'

body(

age: $(ConsumerUtils.oldEnough())

)

headers {

contentType(applicationJson())

}

}

response {

status 200

body(""" {

"status": "${value(ProducerUtils.ok())}"

}

""") headers {

contentType(applicationJson())

}

}

}

|  |  |
| --- | --- |
| IMG_582 | **重要** |
| 您可以通过将convertToYaml设置为true来设置Spring Cloud Contract插件。这样，您将不必将具有扩展功能的依赖项添加到使用者方，因为使用者方将使用YAML合同而不是Groovy合同。 |

## **96.使用可插拔架构**

您可能会遇到以其他格式（例如YAML，RAML或PACT）定义合同的情况。在那些情况下，您仍然想从自动生成测试和存根中受益。您可以添加自己的实现以生成测试和存根。另外，您可以自定义测试的生成方式（例如，可以生成其他语言的测试）和存根的生成方式（例如，可以为其他HTTP服务器实现生成存根）。

## **96.1定制合同转换器**

ContractConverter接口使您可以注册自己的合同结构转换器的实现。以下代码清单显示了ContractConverter接口：

**package** org.springframework.cloud.contract.spec

**/\*\***

**\* Converter to be used to convert FROM {@link File} TO {@link Contract}**

**\* and from {@link Contract} to {@code T}**

**\***

**\* @param <T > - type to which we want to convert the contract**

**\***

**\* @author Marcin Grzejszczak**

**\* @since 1.1.0**

**\*/interface** ContractConverter<T> **extends** ContractStorer<T> {

**/\*\***

**\* Should this file be accepted by the converter. Can use the file extension**

**\* to check if the conversion is possible.**

**\***

**\* @param file - file to be considered for conversion**

**\* @return - {@code true} if the given implementation can convert the file**

**\*/**

**boolean** isAccepted(File file)

**/\*\***

**\* Converts the given {@link File} to its {@link Contract} representation**

**\***

**\* @param file - file to convert**

**\* @return - {@link Contract} representation of the file**

**\*/**

Collection<Contract> convertFrom(File file)

**/\*\***

**\* Converts the given {@link Contract} to a {@link T} representation**

**\***

**\* @param contract - the parsed contract**

**\* @return - {@link T} the type to which we do the conversion**

**\*/**

T convertTo(Collection<Contract> contract)

}

您的实现必须定义启动转换的条件。另外，您必须定义如何在两个方向上执行该转换。

|  |  |
| --- | --- |
| IMG_583 | **重要** |
| 创建实施后，必须创建一个/META-INF/spring.factories文件，在其中提供实施的完全限定名称。 |

以下示例显示了典型的spring.factories文件：

org.springframework.cloud.contract.spec.ContractConverter=\

org.springframework.cloud.contract.verifier.converter.YamlContractConverter

### **96.1.1 Pact转换器**

Spring Cloud Contract包括对直到第4版的[契约的契约](https://docs.pact.io/" \t "https://www.springcloud.cc/_top)表示的支持。您可以使用Pact文件来代替使用Groovy DSL。在本节中，我们介绍如何为您的项目添加Pact支持。但是请注意，并非所有功能都受支持。从v3开始，您可以为同一个元素组合多个匹配器。您可以将匹配器用于正文，标头，请求和路径；您可以使用价值生成器。Spring Cloud Contract当前仅支持使用AND规则逻辑组合的多个匹配器。除此之外，在转换过程中将跳过请求和路径匹配器。当使用具有给定格式的日期，时间或日期时间值生成器时，将跳过给定格式并使用ISO格式。

为了正确支持使用Pact进行消息传递的Spring Cloud Contract方法，您必须提供一些其他元数据条目。您可以在下面找到此类条目的列表：

* 要定义消息发送到的目的地，必须在Pact文件中设置一个metaData项，键sentTo等于消息发送到的目的地。例如"metaData": { "sentTo": "activemq:output" }

### **96.1.2契约合同**

考虑以下契约合同的示例，该契约是src/test/resources/contracts文件夹下的文件。

{

"provider": {

"name": "Provider"

},

"consumer": {

"name": "Consumer"

},

"interactions": [

{

"description": "",

"request": {

"method": "PUT",

"path": "/fraudcheck",

"headers": {

"Content-Type": "application/vnd.fraud.v1+json"

},

"body": {

"clientId": "1234567890",

"loanAmount": 99999

},

"generators": {

"body": {

"$.clientId": {

"type": "Regex",

"regex": "[0-9]{10}"

}

}

},

"matchingRules": {

"header": {

"Content-Type": {

"matchers": [

{

"match": "regex",

"regex": "application/vnd\\.fraud\\.v1\\+json.\*"

}

],

"combine": "AND"

}

},

"body": {

"$.clientId": {

"matchers": [

{

"match": "regex",

"regex": "[0-9]{10}"

}

],

"combine": "AND"

}

}

}

},

"response": {

"status": 200,

"headers": {

"Content-Type": "application/vnd.fraud.v1+json;charset=UTF-8"

},

"body": {

"fraudCheckStatus": "FRAUD",

"rejectionReason": "Amount too high"

},

"matchingRules": {

"header": {

"Content-Type": {

"matchers": [

{

"match": "regex",

"regex": "application/vnd\\.fraud\\.v1\\+json.\*"

}

],

"combine": "AND"

}

},

"body": {

"$.fraudCheckStatus": {

"matchers": [

{

"match": "regex",

"regex": "FRAUD"

}

],

"combine": "AND"

}

}

}

}

}

],

"metadata": {

"pact-specification": {

"version": "3.0.0"

},

"pact-jvm": {

"version": "3.5.13"

}

}

}

本部分有关使用Pact的其余部分参考前面的文件。

### **96.1.3生产者公约**

在生产者端，您必须在插件配置中添加两个其他依赖项。一个是Spring Cloud Contract Pact支持，另一个是您使用的当前Pact版本。

**Maven.**

<plugin>

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

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

</configuration>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-contract-pact</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

</dependencies></plugin>

**Gradle.**

classpath "org.springframework.cloud:spring-cloud-contract-pact:${findProperty('verifierVersion') ?: verifierVersion}"

当您执行应用程序的构建时，将生成一个测试。生成的测试可能如下：

@Test**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd\\.fraud\\.v1\\+json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['rejectionReason']").isEqualTo("Amount too high");

*// and:*

assertThat(parsedJson.read("$.fraudCheckStatus", String.**class**)).matches("FRAUD");

}

相应的生成的存根可能如下：

{

"id" : "996ae5ae-6834-4db6-8fac-358ca187ab62",

"uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",

"request" : {

"url" : "/fraudcheck",

"method" : "PUT",

"headers" : {

"Content-Type" : {

"matches" : "application/vnd\\.fraud\\.v1\\+json.\*"

}

},

"bodyPatterns" : [ {

"matchesJsonPath" : "$[?(@.['loanAmount'] == 99999)]"

}, {

"matchesJsonPath" : "$[?(@.clientId =~ /([0-9]{10})/)]"

} ]

},

"response" : {

"status" : 200,

"body" : "{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Amount too high\"}",

"headers" : {

"Content-Type" : "application/vnd.fraud.v1+json;charset=UTF-8"

},

"transformers" : [ "response-template" ]

},

}

### **96.1.4消费者契约**

在使用者方面，必须将两个其他依赖项添加到项目依赖项中。一个是Spring Cloud Contract Pact支持，另一个是您使用的当前Pact版本。

**Maven.**

<dependency>

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

<artifactId>spring-cloud-contract-pact</artifactId>

<scope>test</scope></dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-contract-pact"

## **96.2使用自定义测试生成器**

如果要针对Java以外的语言生成测试，或者对验证程序构建Java测试的方式不满意，则可以注册自己的实现。

SingleTestGenerator接口使您可以注册自己的实现。以下代码清单显示了SingleTestGenerator界面：

**package** org.springframework.cloud.contract.verifier.builder

**import** org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties**import** org.springframework.cloud.contract.verifier.file.ContractMetadata

**/\*\***

**\* Builds a single test.**

**\***

**\* @since 1.1.0**

**\*/**

trait SingleTestGenerator {

**/\*\***

**\* Creates contents of a single test class in which all test scenarios from**

**\* the contract metadata should be placed.**

**\***

**\* @param properties - properties passed to the plugin**

**\* @param listOfFiles - list of parsed contracts with additional metadata**

**\* @param className - the name of the generated test class**

**\* @param classPackage - the name of the package in which the test class should be stored**

**\* @param includedDirectoryRelativePath - relative path to the included directory**

**\* @return contents of a single test class**

**\* @deprecated use{@link SingleTestGenerator#buildClass(ContractVerifierConfigProperties, Collection, String, GeneratedClassData)}**

**\*/**

@Deprecated

**abstract** String buildClass(ContractVerifierConfigProperties properties,

Collection<ContractMetadata> listOfFiles, String className, String classPackage, String includedDirectoryRelativePath)

**/\*\***

**\* Creates contents of a single test class in which all test scenarios from**

**\* the contract metadata should be placed.**

**\***

**\* @param properties - properties passed to the plugin**

**\* @param listOfFiles - list of parsed contracts with additional metadata**

**\* @param generatedClassData - information about the generated class**

**\* @param includedDirectoryRelativePath - relative path to the included directory**

**\* @return contents of a single test class**

**\*/**

String buildClass(ContractVerifierConfigProperties properties,

Collection<ContractMetadata> listOfFiles, String includedDirectoryRelativePath, GeneratedClassData generatedClassData) {

String className = generatedClassData.className

String classPackage = generatedClassData.classPackage

String path = includedDirectoryRelativePath

**return** buildClass(properties, listOfFiles, className, classPackage, path)

}

**/\*\***

**\* Extension that should be appended to the generated test class. E.g. {@code .java} or {@code .php}**

**\***

**\* @param properties - properties passed to the plugin**

**\*/**

**abstract** String fileExtension(ContractVerifierConfigProperties properties)

**static** **class** GeneratedClassData {

**public** **final** String className

**public** **final** String classPackage

**public** **final** java.nio.file.Path testClassPath

GeneratedClassData(String className, String classPackage,

java.nio.file.Path testClassPath) {

**this**.className = className

**this**.classPackage = classPackage

**this**.testClassPath = testClassPath

}

}

}

同样，您必须提供一个spring.factories文件，例如以下示例中所示的文件：

org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/

com.example.MyGenerator

## **96.3使用自定义存根生成器**

如果要为WireMock以外的存根服务器生成存根，则可以插入自己的StubGenerator接口实现。以下代码清单显示了StubGenerator接口：

**package** org.springframework.cloud.contract.verifier.converter

**import** groovy.transform.CompileStatic

**import** org.springframework.cloud.contract.spec.Contract**import** org.springframework.cloud.contract.verifier.file.ContractMetadata

**/\*\***

**\* Converts contracts into their stub representation.**

**\***

**\* @since 1.1.0**

**\*/**@CompileStatic**interface** StubGenerator {

**/\*\***

**\* @return {@code true} if the converter can handle the file to convert it into a stub.**

**\*/**

**boolean** canHandleFileName(String fileName)

**/\*\***

**\* @return the collection of converted contracts into stubs. One contract can**

**\* result in multiple stubs.**

**\*/**

Map<Contract, String> convertContents(String rootName, ContractMetadata content)

**/\*\***

**\* @return the name of the converted stub file. If you have multiple contracts**

**\* in a single file then a prefix will be added to the generated file. If you**

**\* provide the {@link Contract#name} field then that field will override the**

**\* generated file name.**

**\***

**\* Example: name of file with 2 contracts is {@code foo.groovy}, it will be**

**\* converted by the implementation to {@code foo.json}. The recursive file**

**\* converter will create two files {@code 0\_foo.json} and {@code 1\_foo.json}**

**\*/**

String generateOutputFileNameForInput(String inputFileName)

}

同样，您必须提供一个spring.factories文件，例如以下示例中所示的文件：

# Stub converters

org.springframework.cloud.contract.verifier.converter.StubGenerator=\

org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter

默认实现是WireMock存根生成。

|  |
| --- |
| IMG_584 |
| 您可以提供多个存根生成器实现。例如，从单个DSL，您可以同时生成WireMock存根和Pact文件。 |

## **96.4使用自定义Stub Runner**

如果决定使用自定义存根生成，则还需要使用自定义方式与其他存根提供程序一起运行存根。

假设您使用[Moco](https://github.com/dreamhead/moco" \t "https://www.springcloud.cc/_top)来构建存根，并且已经编写了存根生成器并将存根放置在JAR文件中。

为了使Stub Runner知道如何运行存根，您必须定义一个自定义HTTP Stub服务器实现，该实现可能类似于以下示例：

**package** org.springframework.cloud.contract.stubrunner.provider.moco

**import** com.github.dreamhead.moco.bootstrap.arg.HttpArgs**import** com.github.dreamhead.moco.runner.JsonRunner**import** com.github.dreamhead.moco.runner.RunnerSetting**import** groovy.util.logging.Commons

**import** org.springframework.cloud.contract.stubrunner.HttpServerStub**import** org.springframework.util.SocketUtils

@Commons**class** MocoHttpServerStub **implements** HttpServerStub {

**private** **boolean** started

**private** JsonRunner runner

**private** **int** port

@Override

**int** port() {

**if** (!isRunning()) {

**return** -1

}

**return** port

}

@Override

**boolean** isRunning() {

**return** started

}

@Override

HttpServerStub start() {

**return** start(SocketUtils.findAvailableTcpPort())

}

@Override

HttpServerStub start(**int** port) {

**this**.port = port

**return** **this**

}

@Override

HttpServerStub stop() {

**if** (!isRunning()) {

**return** **this**

}

**this**.runner.stop()

**return** **this**

}

@Override

HttpServerStub registerMappings(Collection<File> stubFiles) {

List<RunnerSetting> settings = stubFiles.findAll { it.name.endsWith("json") }

.collect {

log.info("Trying to parse [${it.name}]")

**try** {

**return** RunnerSetting.aRunnerSetting().withStream(it.newInputStream()).

build()

}

**catch** (Exception e) {

log.warn("Exception occurred while trying to parse file [${it.name}]", e)

**return** null

}

}.findAll { it }

**this**.runner = JsonRunner.newJsonRunnerWithSetting(settings,

HttpArgs.httpArgs().withPort(**this**.port).build())

**this**.runner.run()

**this**.started = true

**return** **this**

}

@Override

String registeredMappings() {

**return** ""

}

@Override

**boolean** isAccepted(File file) {

**return** file.name.endsWith(".json")

}

}

然后，可以将其注册到spring.factories文件中，如以下示例所示：

org.springframework.cloud.contract.stubrunner.HttpServerStub=\

org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub

现在，您可以使用Moco运行存根。

|  |  |
| --- | --- |
| IMG_585 | **重要** |
| 如果不提供任何实现，则使用默认（WireMock）实现。如果提供多个​​，则使用列表中的第一个。 |

## **96.5使用自定义存根下载器**

您可以通过创建StubDownloaderBuilder接口的实现来自定义存根的下载方式，如以下示例所示：

**package** com.example;

**class** CustomStubDownloaderBuilder **implements** StubDownloaderBuilder {

@Override

**public** StubDownloader build(**final** StubRunnerOptions stubRunnerOptions) {

**return** **new** StubDownloader() {

@Override

**public** Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(

StubConfiguration config) {

File unpackedStubs = retrieveStubs();

**return** **new** AbstractMap.SimpleEntry<>(

**new** StubConfiguration(config.getGroupId(), config.getArtifactId(), version,

config.getClassifier()), unpackedStubs);

}

File retrieveStubs() {

*// here goes your custom logic to provide a folder where all the stubs reside*

}

}

然后，可以将其注册到spring.factories文件中，如以下示例所示：

# Example of a custom Stub Downloader Provider

org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\

com.example.CustomStubDownloaderBuilder

现在，您可以选择一个包含存根源的文件夹。

|  |  |
| --- | --- |
| IMG_586 | **重要** |
| 如果不提供任何实现，则使用默认设置（扫描类路径）。如果提供stubsMode = StubRunnerProperties.StubsMode.LOCAL或, stubsMode = StubRunnerProperties.StubsMode.REMOTE，则将使用Aether实现。如果提供多个​​，则将使用列表中的第一个。 |

## **96.6使用SCM存根下载器**

每当repositoryRoot以SCM协议开头（当前我们仅支持git://）时，存根下载器都会尝试克隆存储库，并将其用作生成测试或存根的合同来源。

通过环境变量，系统属性，插件内部设置的属性或合同存储库配置，您可以调整下载程序的行为。您可以在下面找到属性列表

**表96.1 SCM存根下载器属性**

|  |  |  |
| --- | --- | --- |
| 物业类型 | 物业名称 | 描述 |
| \* git.branch（插件支持）  \* stubrunner.properties.git.branch（系统道具）  \* STUBRUNNER\_PROPERTIES\_GIT\_BRANCH（环境道具） | 主 | 结帐哪个分支 |
| \* git.username（插件支持）  \* stubrunner.properties.git.username（系统道具）  \* STUBRUNNER\_PROPERTIES\_GIT\_USERNAME（env prop） |  | Git克隆用户名 |
| \* git.password（插件支持）  \* stubrunner.properties.git.password（系统道具）  \* STUBRUNNER\_PROPERTIES\_GIT\_PASSWORD（环保道具） |  | Git克隆密码 |
| \* git.no-of-attempts（插件支持）  \* stubrunner.properties.git.no-of-attempts（系统道具）  \* STUBRUNNER\_PROPERTIES\_GIT\_NO\_OF\_ATTEMPTS（环境道具） | 10 | 将提交推送到origin的尝试次数 |
| \* git.wait-between-attempts（插件支持）  \* stubrunner.properties.git.wait-between-attempts（系统道具）  \* STUBRUNNER\_PROPERTIES\_GIT\_WAIT\_BETWEEN\_ATTEMPTS（env prop） | 1000 | 两次尝试将提交推送到origin时要等待的毫秒数 |

## **96.7使用契约存根下载器**

每当repositoryRoot以Pact协议开头（以pact://开头）时，存根下载器都将尝试从Pact Broker中获取Pact合同定义。pact://之后设置的任何内容都将解析为Pact Broker URL。

通过环境变量，系统属性，插件内部设置的属性或合同存储库配置，您可以调整下载程序的行为。您可以在下面找到属性列表

**表96.2。SCM存根下载器属性**

|  |  |  |
| --- | --- | --- |
| 物业名称 | 默认 | 描述 |
| \* pactbroker.host（插件支持）  \* stubrunner.properties.pactbroker.host（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_HOST（环境道具） | 通过URL传递给repositoryRoot的主机 | Pact Broker的URL是什么 |
| \* pactbroker.port（插件支持）  \* stubrunner.properties.pactbroker.port（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PORT（环境道具） | URL的端口已传递到repositoryRoot | Pact Broker的端口是什么 |
| \* pactbroker.protocol（插件支持）  \* stubrunner.properties.pactbroker.protocol（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PROTOCOL（环境道具） | 来自URL的协议传递到repositoryRoot | Pact Broker的协议是什么 |
| \* pactbroker.tags（插件支持）  \* stubrunner.properties.pactbroker.tags（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_TAGS（环境道具） | 存根的版本；如果版本为+，则为latest | 应该使用什么标签来获取存根 |
| \* pactbroker.auth.scheme（插件支持）  \* stubrunner.properties.pactbroker.auth.scheme（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_SCHEME（环境道具） | Basic | 应该使用哪种身份验证来连接到Pact Broker |
| \* pactbroker.auth.username（插件支持）  \* stubrunner.properties.pactbroker.auth.username（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_USERNAME（环境道具） | 用户名传递给contractsRepositoryUsername（Maven）或contractRepository.username（gradle） | 用于连接到Pact Broker的用户名 |
| \* pactbroker.auth.password（插件支持）  \* stubrunner.properties.pactbroker.auth.password（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_PASSWORD（环境道具） | 密码已传递给contractsRepositoryPassword（专家）或contractRepository.password（等级） | 用于连接Pact Broker的密码 |
| \* pactbroker.provider-name-with-group-id（插件支持）  \* stubrunner.properties.pactbroker.provider-name-with-group-id（系统道具）  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PROVIDER\_NAME\_WITH\_GROUP\_ID（env prop） | 假 | 当为true时，提供者名称将为groupId:artifactId的组合。如果为false，则仅使用artifactId |

## **97. Spring Cloud Contract WireMock**

使用Spring Cloud Contract WireMock模块，您可以在Spring Boot应用程序中使用[WireMock](https://github.com/tomakehurst/wiremock" \t "https://www.springcloud.cc/_top)。查看 [示例](https://github.com/spring-cloud/spring-cloud-contract/tree/2.1.x/samples" \t "https://www.springcloud.cc/_top) 以获取更多详细信息。

如果您有一个Spring Boot应用程序，该应用程序使用Tomcat作为嵌入式服务器（这是spring-boot-starter-web的默认设置），则可以将spring-cloud-starter-contract-stub-runner添加到您的类路径中，并添加@AutoConfigureWireMock，以便能够在测试中使用Wiremock。Wiremock作为存根服务器运行，您可以在测试中使用Java API或通过静态JSON声明来注册存根行为。以下代码显示了一个示例：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)@AutoConfigureWireMock(port = 0)**public** **class** WiremockForDocsTests {

*// A service that calls out over HTTP*

@Autowired

**private** Service service;

@Before

**public** **void** setup() {

**this**.service.setBase("http://localhost:"

+ **this**.environment.getProperty("wiremock.server.port"));

}

*// Using the WireMock APIs in the normal way:*

@Test

**public** **void** contextLoads() **throws** Exception {

*// Stubbing WireMock*

stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()

.withHeader("Content-Type", "text/plain").withBody("Hello World!")));

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World!");

}

}

要在其他端口上启动存根服务器，请使用@AutoConfigureWireMock(port=9999)。对于随机端口，请使用值0。可以在测试应用程序上下文中使用“ wiremock.server.port”属性绑定存根服务器端口。使用@AutoConfigureWireMock将类型为WiremockConfiguration的bean添加到测试应用程序上下文中，该变量将被缓存在具有相同上下文的方法和类之间，与Spring集成测试相同。您也可以将WireMockServer类型的bean注入测试中。

## **97.1自动注册存根**

如果使用@AutoConfigureWireMock，它将从文件系统或类路径（默认情况下，从file:src/test/resources/mappings）注册WireMock JSON存根。您可以使用注释中的stubs属性来自定义位置，该属性可以是Ant样式的资源模式或目录。对于目录，将附加**\*/**.json。以下代码显示了一个示例：

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureWireMock(stubs="classpath:/stubs")

public class WiremockImportApplicationTests {

@Autowired

private Service service;

@Test

public void contextLoads() throws Exception {

assertThat(this.service.go()).isEqualTo("Hello World!");

}

}

|  |
| --- |
| IMG_587 |
| 实际上，WireMock总是从src/test/resources/mappings **以及** stubs属性中的自定义位置加载映射。要更改此行为，还可以按照本文档下一节中的说明指定文件根。 |

如果您使用的是Spring Cloud Contract的默认存根jar，则您的存根将存储在/META-INF/group-id/artifact-id/versions/mappings/文件夹下。如果要从该位置，所有嵌入式JAR中注册所有存根，那么使用以下语法就​​足够了。

@AutoConfigureWireMock(port = 0, stubs = "classpath\*:/META-INF/\*\*/mappings/\*\*/\*.json")

## **97.2使用文件指定存根实体**

WireMock可以从类路径或文件系统上的文件中读取响应正文。在这种情况下，您可以在JSON DSL中看到响应具有bodyFileName而不是（文字）body。相对于根目录（默认为src/test/resources/\_\_files）来解析文件。要自定义此位置，可以将@AutoConfigureWireMock批注中的files属性设置为父目录的位置（换句话说，\_\_files是子目录）。您可以使用Spring资源表示法来引用file:…​或classpath:…​位置。不支持通用网址。可以给出一个值列表，在这种情况下，WireMock会在需要查找响应正文时解析存在的第一个文件。

|  |
| --- |
| IMG_588 |
| 当您配置files根目录时，它也会影响存根的自动加载，因为它们来自子目录“映射”中的根目录。files的值对从stubs属性显式加载的存根没有影响。 |

## **97.3替代：使用JUnit规则**

要获得更常规的WireMock体验，可以使用JUnit @Rules启动和停止服务器。为此，请使用方便类WireMockSpring获得一个Options实例，如以下示例所示：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)**public** **class** WiremockForDocsClassRuleTests {

*// Start WireMock on some dynamic port*

*// for some reason `dynamicPort()` is not working properly*

@ClassRule

**public** **static** WireMockClassRule wiremock = **new** WireMockClassRule(

WireMockSpring.options().dynamicPort());

*// A service that calls out over HTTP to wiremock's port*

@Autowired

**private** Service service;

@Before

**public** **void** setup() {

**this**.service.setBase("http://localhost:" + wiremock.port());

}

*// Using the WireMock APIs in the normal way:*

@Test

**public** **void** contextLoads() **throws** Exception {

*// Stubbing WireMock*

wiremock.stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()

.withHeader("Content-Type", "text/plain").withBody("Hello World!")));

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World!");

}

}

@ClassRule表示在运行了此类中的所有方法之后，服务器将关闭。

## **97.4放松模板的SSL验证**

WireMock允许您使用“ https” URL协议对“安全”服务器进行存根。如果您的应用程序希望在集成测试中联系该存根服务器，它将发现SSL证书无效（自安装证书的常见问题）。最好的选择通常是将客户端重新配置为使用“ http”。如果这不是一种选择，则可以要求Spring配置忽略SSL验证错误的HTTP客户端（当然，仅对测试而言如此）。

为了使此工作最小，您需要在应用中使用Spring Boot RestTemplateBuilder，如以下示例所示：

@Bean**public** RestTemplate restTemplate(RestTemplateBuilder builder) {

**return** builder.build();

}

您需要RestTemplateBuilder，因为构建器是通过回调传递的，以对其进行初始化，因此此时可以在客户端中设置SSL验证。如果您使用的是@AutoConfigureWireMock批注或存根运行程序，则这会在测试中自动发生。如果使用JUnit @Rule方法，则还需要添加@AutoConfigureHttpClient批注，如以下示例所示：

@RunWith(SpringRunner.class)@SpringBootTest("app.baseUrl=https://localhost:6443")@AutoConfigureHttpClient**public** **class** WiremockHttpsServerApplicationTests {

@ClassRule

**public** **static** WireMockClassRule wiremock = **new** WireMockClassRule(

WireMockSpring.options().httpsPort(6443));

...

}

如果您使用的是spring-boot-starter-test，则将Apache HTTP客户端放在类路径上，并由RestTemplateBuilder选择它，并将其配置为忽略SSL错误。如果使用默认的java.net客户端，则不需要注释（但不会造成任何危害）。当前不支持其他客户端，但可能会在将来的版本中添加。

要禁用自定义RestTemplateBuilder，请将wiremock.rest-template-ssl-enabled属性设置为false。

## **97.5 WireMock和Spring MVC模拟**

Spring Cloud Contract提供了一个便利类，可以将JSON WireMock存根加载到Spring MockRestServiceServer中。以下代码显示了一个示例：

@RunWith(SpringRunner.class)@SpringBootTest(webEnvironment = WebEnvironment.NONE)**public** **class** WiremockForDocsMockServerApplicationTests {

@Autowired

**private** RestTemplate restTemplate;

@Autowired

**private** Service service;

@Test

**public** **void** contextLoads() **throws** Exception {

*// will read stubs classpath*

MockRestServiceServer server = WireMockRestServiceServer.with(**this**.restTemplate)

.baseUrl("https://example.org").stubs("classpath:/stubs/resource.json")

.build();

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World");

server.verify();

}

}

baseUrl值被附加到所有模拟调用中，并且stubs()方法采用存根路径资源模式作为参数。在前面的示例中，在/stubs/resource.json定义的存根被加载到模拟服务器中。如果要求RestTemplate访问[https://example.org/](https://example.org/" \t "https://www.springcloud.cc/_top)，它将获得在该URL声明的响应。可以指定多个存根模式，每个存根模式都可以是目录（用于所有“ .json”的递归列表），固定文件名（如上例所示）或Ant样式的模式。JSON格式是标准的WireMock格式，您可以在[WireMock网站上](https://wiremock.org/docs/stubbing/" \t "https://www.springcloud.cc/_top)阅读该 [格式](https://wiremock.org/docs/stubbing/" \t "https://www.springcloud.cc/_top)。

当前，Spring Cloud Contract验证程序支持Tomcat，Jetty和Undertow作为Spring Boot嵌入式服务器，而Wiremock本身对特定版本的Jetty（当前为9.2）具有“本机”支持。要使用本机Jetty，您需要添加本机Wiremock依赖项并排除Spring Boot容器（如果有）。

## **97.6自定义WireMock配置**

您可以注册org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer类型的bean以自定义WireMock配置（例如，添加自定义转换器）。例：

@Bean

WireMockConfigurationCustomizer optionsCustomizer() {

**return** **new** WireMockConfigurationCustomizer() {

@Override

**public** **void** customize(WireMockConfiguration options) {*// perform your customization here*

}

};

}

## **97.7使用REST文档生成存根**

[Spring REST Docs](https://projects.spring.io/spring-restdocs" \t "https://www.springcloud.cc/_top)可用于为具有Spring MockMvc或WebTestClient或Rest Assured的HTTP API生成文档（例如，Asciidoctor格式）。在为API生成文档的同时，还可以使用Spring Cloud Contract WireMock生成WireMock存根。为此，编写您的常规REST Docs测试用例，并使用@AutoConfigureRestDocs在REST Docs输出目录中自动生成存根。以下代码显示了使用MockMvc的示例：

@RunWith(SpringRunner.class)@SpringBootTest@AutoConfigureRestDocs(outputDir = "target/snippets")@AutoConfigureMockMvc**public** **class** ApplicationTests {

@Autowired

**private** MockMvc mockMvc;

@Test

**public** **void** contextLoads() **throws** Exception {

mockMvc.perform(get("/resource"))

.andExpect(content().string("Hello World"))

.andDo(document("resource"));

}

}

此测试在“ target / snippets / stubs / resource.json”处生成WireMock存根。它将所有GET请求与“ / resource”路径匹配。与WebTestClient相同的示例（用于测试Spring WebFlux应用程序）看起来像这样：

@RunWith(SpringRunner.class)@SpringBootTest@AutoConfigureRestDocs(outputDir = "target/snippets")@AutoConfigureWebTestClient**public** **class** ApplicationTests {

@Autowired

**private** WebTestClient client;

@Test

**public** **void** contextLoads() **throws** Exception {

client.get().uri("/resource").exchange()

.expectBody(String.**class**).isEqualTo("Hello World")

.consumeWith(document("resource"));

}

}

在没有任何其他配置的情况下，这些测试将为HTTP方法创建一个带有请求匹配器的存根，以及除“主机”和“内容长度”之外的所有标头。为了更精确地匹配请求（例如，匹配POST或PUT的正文），我们需要显式创建一个请求匹配器。这样做有两个效果：

* 创建仅以您指定的方式匹配的存根。
* 断言测试用例中的请求也匹配相同的条件。

此功能的主要入口点是WireMockRestDocs.verify()，它可以代替document()便捷方法，如以下示例所示：

**import** **static** org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureRestDocs(outputDir = "target/snippets")

@AutoConfigureMockMvc

public class ApplicationTests {

@Autowired

private MockMvc mockMvc;

@Test

public void contextLoads() throws Exception {

mockMvc.perform(post("/resource")

.content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))

.andExpect(status().isOk())

.andDo(verify().jsonPath("$.id"))

.andDo(document("resource"));

}

}

该合同规定，任何带有“ id”字段的有效POST都会收到此测试中定义的响应。您可以将对.jsonPath()的呼叫链接在一起以添加其他匹配器。如果不熟悉JSON Path，[JayWay文档](https://github.com/jayway/JsonPath" \t "https://www.springcloud.cc/_top)可以帮助您快速[入门](https://github.com/jayway/JsonPath" \t "https://www.springcloud.cc/_top)。此测试的WebTestClient版本具有您插入相同位置的类似的verify()静态助手。

除了jsonPath和contentType便捷方法之外，您还可以使用WireMock API来验证请求是否与创建的存根匹配，如以下示例所示：

@Test**public** **void** contextLoads() **throws** Exception {

mockMvc.perform(post("/resource")

.content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))

.andExpect(status().isOk())

.andDo(verify()

.wiremock(WireMock.post(

urlPathEquals("/resource"))

.withRequestBody(matchingJsonPath("$.id")))

.andDo(document("post-resource"));

}

WireMock API丰富。您可以通过正则表达式以及JSON路径匹配标头，查询参数和请求正文。这些功能可用于创建具有更广泛参数范围的存根。上面的示例生成一个类似于以下示例的存根：

**post-resource.json。**

**{**

"request" : **{**

"url" : "/resource"**,**

"method" : "POST"**,**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$.id"

**}]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "Hello World"**,**

"headers" : **{**

"X-Application-Context" : "application:-1"**,**

"Content-Type" : "text/plain"

**}**

**}}**

|  |
| --- |
| IMG_589 |
| 您可以使用wiremock()方法或jsonPath()和contentType()方法来创建请求匹配器，但是不能同时使用两种方法。 |

在使用者方面，可以使本节前面部分生成的resource.json在类路径上可用（例如，通过<< publishing-stubs-as-jars）。之后，可以使用WireMock以多种不同方式创建存根，包括使用@AutoConfigureWireMock(stubs="classpath:resource.json")，如本文档前面所述。

## **97.8通过使用REST文档生成Contracts**

您还可以使用Spring REST Docs生成Spring Cloud Contract DSL文件和文档。如果与Spring Cloud WireMock结合使用，则会同时获得合同和存根。

您为什么要使用此功能？社区中的一些人询问有关他们希望转向基于DSL的合同定义的情况的问题，但是他们已经进行了许多Spring MVC测试。使用此功能，您可以生成合同文件，以后可以修改合同文件并将其移动到文件夹（在配置中定义），以便插件找到它们。

|  |
| --- |
| IMG_590 |
| 您可能想知道为什么WireMock模块中有此功能。之所以具有此功能是因为生成合同和存根都是有意义的。 |

考虑以下测试：

**this**.mockMvc

.perform(post("/foo").accept(MediaType.APPLICATION\_PDF)

.accept(MediaType.APPLICATION\_JSON)

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"foo\": 23, \"bar\" : \"baz\" }"))

.andExpect(status().isOk()).andExpect(content().string("bar"))

*// first WireMock*

.andDo(WireMockRestDocs.verify().jsonPath("$[?(@.foo >= 20)]")

.jsonPath("$[?(@.bar in ['baz','bazz','bazzz'])]")

.contentType(MediaType.valueOf("application/json")))

*// then Contract DSL documentation*

.andDo(document("index", SpringCloudContractRestDocs.dslContract()));

前面的测试将创建上一部分中介绍的存根，同时生成合同和文档文件。

该合同称为index.groovy，可能类似于以下示例：

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

request {

method 'POST'

url '/foo'

body(''' {"foo": 23 }

''') headers {

header('''Accept''', '''application/json''')

header('''Content-Type''', '''application/json''')

}

}

response {

status OK()

body(''' bar

''') headers {

header('''Content-Type''', '''application/json;charset=UTF-8''')

header('''Content-Length''', '''3''')

}

testMatchers {

jsonPath('$[?(@.foo >= 20)]', byType())

}

}

}

生成的文档（在这种情况下为Asciidoc格式）包含格式化的合同。该文件的位置为index/dsl-contract.adoc。

## **98.迁移**

|  |
| --- |
| IMG_591 |
| 有关最新的迁移指南，请访问项目的[Wiki页面](https://github.com/spring-cloud/spring-cloud-contract/wiki/" \t "https://www.springcloud.cc/_top)。 |

本节介绍从Spring Cloud Contract验证程序的一个版本迁移到下一版本。它涵盖以下版本升级路径：

## **98.1 1.0.x→1.1.x**

本节介绍从1.0版升级到1.1版的过程。

### **98.1.1生成的存根的新结构**

在1.1.x中，我们对生成的存根的结构进行了更改。如果您一直使用@AutoConfigureWireMock表示法来使用类路径中的存根，那么它将不再起作用。以下示例显示了@AutoConfigureWireMock表示法的工作原理：

@AutoConfigureWireMock(stubs = "classpath:/customer-stubs/mappings", port = 8084)

您必须将存根的位置更改为：classpath:…​/META-INF/groupId/artifactId/version/mappings或使用新的基于类路径的@AutoConfigureStubRunner，如以下示例所示：

@AutoConfigureWireMock(stubs = "classpath:customer-stubs/META-INF/travel.components/customer-contract/1.0.2-SNAPSHOT/mappings/", port = 8084)

如果您不想使用@AutoConfigureStubRunner，并且希望保留原来的结构，请相应地设置插件任务。以下示例适用于前一片段中介绍的结构。

**Maven.**

*<!-- start of pom.xml -->*

<properties>

*<!-- we don't want the verifier to do a jar for us -->*

<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip></properties>

*<!-- ... -->*

*<!-- You need to set up the assembly plugin -->*<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>stub</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<inherited>false</inherited>

<configuration>

<attach>true</attach>

<descriptor>$../../../../src/assembly/stub.xml</descriptor>

</configuration>

</execution>

</executions>

</plugin>

</plugins></build>*<!-- end of pom.xml -->*

*<!-- start of stub.xml-->*

<assembly

xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>stubs</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>${project.build.directory}/snippets/stubs</directory>

<outputDirectory>customer-stubs/mappings</outputDirectory>

<includes>

<include>\*\*/\*</include>

</includes>

</fileSet>

<fileSet>

<directory>$../../../../src/test/resources/contracts</directory>

<outputDirectory>customer-stubs/contracts</outputDirectory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</fileSet>

</fileSets></assembly>

*<!-- end of stub.xml-->*

**Gradle.**

task copyStubs(type: Copy, dependsOn: 'generateWireMockClientStubs') {*// Preserve directory structure from 1.0.X of spring-cloud-contract*

from "${project.buildDir}/resources/main/customer-stubs/META-INF/${project.group}/${project.name}/${project.version}"

into "${project.buildDir}/resources/main/customer-stubs"

}

## **98.2 1.1.x→1.2.x**

本节介绍从版本1.1升级到版本1.2。

### **98.2.1自定义HttpServerStub**

HttpServerStub包含版本1.1以外的方法。方法是String registeredMappings()如果您有实现HttpServerStub的类，则现在必须实现registeredMappings()方法。它应返回一个String，代表单个HttpServerStub中所有可用的映射。

有关更多详细信息，请参见[问题355](https://github.com/spring-cloud/spring-cloud-contract/issues/355" \t "https://www.springcloud.cc/_top)。

### **98.2.2用于生成测试的新软件包**

设置生成的测试程序包名称的流程如下所示：

* 设置basePackageForTests
* 如果未设置basePackageForTests，请从baseClassForTests中选择包装
* 如果未设置baseClassForTests，则选择packageWithBaseClasses
* 如果未设置任何内容，请选择默认值：org.springframework.cloud.contract.verifier.tests

有关更多详细信息，请参见[问题260](https://github.com/spring-cloud/spring-cloud-contract/issues/260" \t "https://www.springcloud.cc/_top)。

### **98.2.3 TemplateProcessor中的新方法**

为了增加对fromRequest.path的支持，必须在TemplateProcessor接口中添加以下方法：

* path()
* path(int index)

有关更多详细信息，请参见[问题388](https://github.com/spring-cloud/spring-cloud-contract/issues/388" \t "https://www.springcloud.cc/_top)。

### **98.2.4 RestAssured 3.0**

在生成的测试类中使用的“放心的保证”被提高到3.0。如果您手动设置Spring Cloud Contract的版本和发行版，可能会看到以下异常：

Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile (default-testCompile) on project some-project: Compilation failure: Compilation failure:

[ERROR] /some/path/SomeClass.java:[4,39] package com.jayway.restassured.response does not exist

由于使用旧版本的插件生成了测试，并且在测试执行时您具有发行版的不兼容版本（反之亦然），因此会发生此异常。

通过[问题267](https://github.com/spring-cloud/spring-cloud-contract/issues/267" \t "https://www.springcloud.cc/_top)完成

## **98.3 1.2.x→2.0.x**

## **99.链接**

以下链接在使用Spring Cloud Contract时可能会有所帮助：

* [Spring Cloud Contract Github Repository](https://github.com/spring-cloud/spring-cloud-contract/" \t "https://www.springcloud.cc/_top)
* [Spring Cloud合同示例](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/" \t "https://www.springcloud.cc/_top)
* [Spring Cloud Contract Gitter](https://gitter.im/spring-cloud/spring-cloud-contract" \t "https://www.springcloud.cc/_top)
* [Spring Cloud Contract WJUG演讲，作者：Marcin Grzejszczak](https://www.youtube.com/watch?v=sAAklvxmPmk" \t "https://www.springcloud.cc/_top)

# **第十四部分。Spring Cloud Vault**

©2016-2019原作者。

|  |
| --- |
| IMG_592 |
| 本文档的副本可以供您自己使用，也可以分发给其他人，但前提是您不对此类副本收取任何费用，并且还应确保每份副本均包含本版权声明（无论是印刷版本还是电子版本）。 |

Spring Cloud Vault Config为分布式系统中的外部化配置提供客户端支持。使用[HashiCorp的Vault，](https://www.vaultproject.io/" \t "https://www.springcloud.cc/_top)您可以在中心位置管理所有环境中应用程序的外部秘密属性。Vault可以管理静态和动态机密，例如远程应用程序/资源的用户名/密码，并为外部服务（例如MySQL，PostgreSQL，Apache Cassandra，MongoDB，Consul，AWS等）提供凭据。

## **100.快速入门**

**先决条件**

要开始使用Vault和本指南，您需要一个类似\* NIX的操作系统，该操作系统提供：

* wget，openssl和unzip
* 至少Java 7和正确配置的JAVA\_HOME环境变量

**安装Vault**

$ src/**test**/bash/install\_vault.sh

**为Vault创建SSL证书**

$ src/**test**/bash/create\_certificates.sh

|  |
| --- |
| IMG_593 |
| create\_certificates.sh在work/ca和JKS信任库work/keystore.jks中创建证书。如果要使用此快速入门指南运行Spring Cloud Vault，则需要将信任库的spring.cloud.vault.ssl.trust-store属性配置为file:work/keystore.jks。 |

**启动Vault服务器**

$ src/**test**/bash/local\_run\_vault.sh

Vault开始使用inmem存储和https在0.0.0.0:8200上侦听。Vault被密封并且在启动时未初始化。

|  |
| --- |
| IMG_594 |
| 如果要运行测试，请保留Vault未初始化。测试将初始化Vault并创建根令牌00000000-0000-0000-0000-000000000000。 |

如果要对应用程序使用Vault或尝试使用它，则需要首先对其进行初始化。

$ **export** VAULT\_ADDR="https://localhost:8200"

$ **export** VAULT\_SKIP\_VERIFY=true *# Don't do this for production*

$ vault init

您应该看到类似以下内容：

Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701

Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02

Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03

Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04

Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805

Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

Vault initialized with 5 keys and a key threshold of 3. Please

securely distribute the above keys. When the Vault is re-sealed,

restarted, or stopped, you must provide at least 3 of these keys

to unseal it again.

Vault does not store the master key. Without at least 3 keys,

your Vault will remain permanently sealed.

Vault将初始化并返回一组启封密钥和根令牌。选择3个键并解开Vault。将Vault令牌存储在VAULT\_TOKEN环境变量中。

$ vault unseal (Key 1)

$ vault unseal (Key 2)

$ vault unseal (Key 3)

$ **export** VAULT\_TOKEN=(Root token)*# Required to run Spring Cloud Vault tests after manual initialization*

$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"

Spring Cloud Vault访问不同的资源。默认情况下，启用秘密后端，该后端通过JSON端点访问秘密配置设置。

HTTP服务具有以下形式的资源：

/secret/{application}/{profile}

/secret/{application}

/secret/{defaultContext}/{profile}

/secret/{defaultContext}

如果将“应用程序”作为spring.application.name插入SpringApplication中（即常规Spring Boot应用程序中通常是“应用程序”），则“配置文件”是有效的配置文件（或逗号分隔的列表）属性）。从Vault中检索到的Properties将按原样使用，而无需进一步添加属性名称的前缀。

## **101.客户端使用**

要在应用程序中使用这些功能，只需将其构建为依赖于spring-cloud-vault-config的Spring Boot应用程序即可（例如，查看测试用例）。Maven示例配置：

**示例101.1 pom.xml**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.0.RELEASE</version>

<relativePath /> *<!-- lookup parent from repository -->*</parent>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-starter-vault-config</artifactId>

<version>{project-version}</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency></dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins></build>

*<!-- repositories also needed for snapshots and milestones -->*

然后，您可以创建一个标准的Spring Boot应用程序，例如以下简单的HTTP服务器：

@SpringBootApplication@RestController**public** **class** Application {

@RequestMapping("/")

**public** String home() {

**return** "Hello World!";

}

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

运行时，它将从端口8200上的默认本地Vault服务器获取外部配置（如果正在运行）。要修改启动行为，您可以使用bootstrap.properties来更改Vault服务器的位置（例如application.properties，但用于应用程序上下文的引导阶段），例如

**示例101.2 bootstrap.yml**

spring.cloud.vault: host: localhost port: 8200 scheme: https uri: https://localhost:8200 connection-timeout: 5000 read-timeout: 15000 config: order: -10

* host设置Vault主机的主机名。主机名将用于SSL证书验证
* port设置Vault端口
* scheme将方案设置为http将使用纯HTTP。支持的方案是http和https。
* uri使用URI配置Vault端点。优先于主机/端口/方案配置
* connection-timeout设置连接超时（以毫秒为单位）
* read-timeout设置读取超时（以毫秒为单位）
* config.order设置属性来源的顺序

启用进一步的集成需要附加的依赖关系和配置。根据您设置Vault的方式，可能需要其他配置，例如 [SSL](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html" \l "vault.config.ssl" \t "https://www.springcloud.cc/_top)和 [身份验证](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html" \l "vault.config.authentication" \t "https://www.springcloud.cc/_top)。

如果应用程序导入了spring-boot-starter-actuator项目，则可以通过/health端点获得保管库服务器的状态。

可以通过属性management.health.vault.enabled（默认值为true）启用或禁用Vault运行状况指示器。

## **101.1身份验证**

Vault需要[身份验证机制](https://www.vaultproject.io/docs/concepts/auth.html" \t "https://www.springcloud.cc/_top)来[授权客户端请求](https://www.vaultproject.io/docs/concepts/tokens.html" \t "https://www.springcloud.cc/_top)。

Spring Cloud Vault支持多种[身份验证机制，](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html" \l "vault.config.authentication" \t "https://www.springcloud.cc/_top)以通过Vault对应用程序进行身份验证。

要快速入门，请使用[Vault初始化](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "quickstart.vault.start)打印的根令牌。

**示例101.3 bootstrap.yml**

spring.cloud.vault: token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

|  |
| --- |
| IMG_595 |
| 请仔细考虑您的安全要求。如果您想快速开始使用Vault，则可以使用静态令牌认证，但是静态令牌不再受到任何保护。向非预期方的任何公开都允许Vault与关联的令牌角色一起使用。 |

## **102.认证方式**

不同的组织对安全性和身份验证有不同的要求。Vault通过提供多种身份验证方法来反映这种需求。Spring Cloud Vault支持令牌和AppId身份验证。

## **102.1令牌认证**

令牌是Vault中进行身份验证的核心方法。令牌认证要求使用[Bootstrap应用程序上下文](https://github.com/spring-cloud/spring-cloud-commons/blob/master/docs/src/main/asciidoc/spring-cloud-commons.adoc" \l "the-bootstrap-application-context" \t "https://www.springcloud.cc/_top)提供静态令牌 。

|  |
| --- |
| IMG_596 |
| 令牌认证是默认的认证方法。如果公开了令牌，则意外的一方将获得对Vault的访问权，并且可以访问目标客户的机密。 |

**示例102.1 bootstrap.yml**

spring.cloud.vault: authentication: TOKEN token: 00000000-0000-0000-0000-000000000000

* authentication将此值设置为TOKEN会选择令牌身份验证方法
* token设置要使用的静态令牌

另请参阅：[Vault文档：令牌](https://www.vaultproject.io/docs/concepts/tokens.html" \t "https://www.springcloud.cc/_top)

## **102.2 AppId身份验证**

Vault支持 由两个难以猜测的令牌组成的[AppId](https://www.vaultproject.io/docs/auth/app-id.html" \t "https://www.springcloud.cc/_top)身份验证。AppId默认为静态配置的spring.application.name。第二个令牌是UserId，它是应用程序确定的一部分，通常与运行时环境有关。IP地址，Mac地址或Docker容器名称就是很好的例子。Spring Cloud Vault配置支持IP地址，Mac地址和静态UserId（例如，通过系统属性提供）。IP和Mac地址表示为十六进制编码的SHA256哈希。

基于IP地址的UserId使用本地主机的IP地址。

**示例10.2 bootstrap.yml使用SHA256 IP地址用户ID**

spring.cloud.vault: authentication: APPID app-id: user-id: IP\_ADDRESS

* authentication将此值设置为APPID会选择AppId身份验证方法
* app-id-path设置要使用的AppId安装的路径
* user-id设置UserId方法。可能的值为IP\_ADDRESS，MAC\_ADDRESS或实现自定义AppIdUserIdMechanism的类名

从命令行生成IP地址UserId的相应命令是：

$ echo -n 192.168.99.1 | sha256sum

|  |
| --- |
| IMG_597 |
| 包含echo的换行符会导致不同的哈希值，因此请确保包含-n标志。 |

基于Mac地址的UserId从本地主机绑定的设备获取其网络设备。该配置还允许指定network-interface提示以选择正确的设备。network-interface的值是可选的，可以是接口名称或接口索引（从0开始）。

**示例102.3 bootstrap.yml使用SHA256 Mac-Address用户ID**

spring.cloud.vault: authentication: APPID app-id: user-id: MAC\_ADDRESS network-interface: eth0

* network-interface设置网络接口以获取物理地址

从命令行生成IP地址UserId的相应命令是：

$ echo -n 0AFEDE1234AC | sha256sum

|  |
| --- |
| IMG_598 |
| Mac地址指定为大写且没有冒号。包括echo的换行符会导致不同的哈希值，因此请确保包含-n标志。 |

### **102.2.1自定义用户ID**

UserId生成是一种开放机制。您可以将spring.cloud.vault.app-id.user-id设置为任何字符串，并且配置的值将用作静态UserId。

使用更高级的方法，可以将spring.cloud.vault.app-id.user-id设置为类名。此类必须在您的类路径上，并且必须实现org.springframework.cloud.vault.AppIdUserIdMechanism接口和createUserId方法。Spring Cloud Vault将在每次使用AppId进行身份验证以获取令牌时通过调用createUserId来获取UserId。

**示例102.4 bootstrap.yml**

spring.cloud.vault: authentication: APPID app-id: user-id: com.examlple.MyUserIdMechanism

**示例102.5 MyUserIdMechanism.java**

public class MyUserIdMechanism implements AppIdUserIdMechanism **{**

@Override

public String createUserId() **{** String userId = ...

return userId;

**}}**

另请参阅：[Vault文档：使用App ID auth后端](https://www.vaultproject.io/docs/auth/app-id.html" \t "https://www.springcloud.cc/_top)

## **102.3 AppRole身份验证**

[AppRole](https://www.vaultproject.io/docs/auth/app-id.html" \t "https://www.springcloud.cc/_top)用于机器身份验证，例如已弃用的（自Vault 0.6.1起）[第102.2节“ AppId身份验证”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.authentication.appid" \o "102.2 AppId身份验证)。AppRole身份验证包含两个很难猜测（秘密）的令牌：RoleId和SecretId。

Spring Vault支持各种AppRole方案（推/拉模式和包装）。

RoleId和可选的SecretId必须由配置提供，Spring Vault将不会查找它们或创建自定义SecretId。

**示例102.6 具有AppRole身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: APPROLE app-role: role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52

支持以下方案以及必需的配置详细信息：

**表102.1。组态**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **方法** | **角色编号** | **SecretId** | **角色名** | **代币** |
| 提供的RoleId / SecretId | 提供 | 提供 |  |  |
| 提供的RoleId不含SecretId | 提供 |  |  |  |
| 提供的RoleId，Pull SecretId | 提供 | 提供 | 提供 | 提供 |
| 拉出RoleId，提供SecretId |  | 提供 | 提供 | 提供 |
| 全拉模式 |  |  | 提供 | 提供 |
| 包裹 |  |  |  | 提供 |
| 包装好的RoleId，提供SecretId | 提供 |  |  | 提供 |
| 提供的RoleId，包装的SecretId |  | 提供 |  | 提供 |

**表102.2。拉/推/包裹矩阵**

|  |  |  |
| --- | --- | --- |
| **角色编号** | **SecretId** | **支持的** |
| 提供 | 提供 | ✅ |
| 提供 | 拉 | ✅ |
| 提供 | 包裹 | ✅ |
| 提供 | 缺席 | ✅ |
| 拉 | 提供 | ✅ |
| 拉 | 拉 | ✅ |
| 拉 | 包裹 | ❌ |
| 拉 | 缺席 | ❌ |
| 包裹 | 提供 | ✅ |
| 包裹 | 拉 | ❌ |
| 包裹 | 包裹 | ✅ |
| 包裹 | 缺席 | ❌ |

|  |
| --- |
| IMG_599 |
| 通过在引导上下文中提供已配置的AppRoleAuthentication bean，您仍然可以使用推/拉/包模式的所有组合。Spring Cloud Vault无法从配置属性中导出所有可能的AppRole组合。 |

|  |  |
| --- | --- |
| IMG_600 | **重要** |
| AppRole身份验证仅限于使用反应式基础结构的简单拉模式。尚不支持全拉模式。将Spring Cloud Vault与Spring WebFlux堆栈一起使用将启用Vault的反应式自动配置，可以通过设置spring.cloud.vault.reactive.enabled=false来禁用它。 |

**示例102.7 具有所有AppRole身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: APPROLE app-role: role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52 secret-id: 1696536f-1976-73b1-b241-0b4213908d39 role: my-role app-role-path: approle

* role-id设置RoleId。
* secret-id设置SecretId。如果在不要求SecretId的情况下配置了AppRole，则可以忽略SecretId（请参见bind\_secret\_id）。
* role：设置拉模式的AppRole名称。
* app-role-path设置要使用的方法认证安装的路径。

另请参阅：[Vault文档：使用AppRole身份验证后端](https://www.vaultproject.io/docs/auth/approle.html" \t "https://www.springcloud.cc/_top)

## **102.4 AWS-EC2身份验证**

的[AWS-EC2](https://www.vaultproject.io/docs/auth/aws-ec2.html" \t "https://www.springcloud.cc/_top) AUTH后端提供了一个安全导入机构为AWS EC2实例，允许的Vault令牌自动检索。与大多数Vault身份验证后端不同，此后端不需要先部署或设置安全敏感的凭据（令牌，用户名/密码，客户端证书等）。而是将AWS视为受信任的第三方，并使用经过加密签名的动态元数据信息来唯一表示每个EC2实例。

**示例102.8 使用AWS-EC2身份验证的bootstrap.yml**

spring.cloud.vault: authentication: AWS\_EC2

AWS-EC2身份验证默认使随机数遵循首次使用信任（TOFU）原则。任何可以访问PKCS＃7身份元数据的意外用户都可以针对Vault进行身份验证。

在首次登录期间，Spring Cloud Vault生成一个随机数，该随机数存储在auth后端中，与实例ID无关。重新认证要求发送相同的随机数。任何其他方都没有随机数，可以在Vault中发出警报以进行进一步调查。

随机数保留在内存中，在应用程序重新启动期间丢失。您可以使用spring.cloud.vault.aws-ec2.nonce配置静态随机数。

AWS-EC2身份验证角色是可选的，默认为AMI。您可以通过设置spring.cloud.vault.aws-ec2.role属性来配置身份验证角色。

**示例102.9 具有配置角色的bo​​otstrap.yml**

spring.cloud.vault: authentication: AWS\_EC2 aws-ec2: role: application-server

**示例102.10 具有所有AWS EC2身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: AWS\_EC2 aws-ec2: role: application-server aws-ec2-path: aws-ec2 identity-document: http://... nonce: my-static-nonce

* authentication将此值设置为AWS\_EC2会选择AWS EC2身份验证方法
* role设置尝试进行登录的角色的名称。
* aws-ec2-path设置要使用的AWS EC2安装的路径
* identity-document设置PKCS＃7 AWS EC2身份文档的URL
* nonce用于AWS-EC2身份验证。空随机数默认为随机数生成

另请参阅：[Vault文档：使用aws auth后端](https://www.vaultproject.io/docs/auth/aws.html" \t "https://www.springcloud.cc/_top)

## **102.5 AWS-IAM身份验证**

在[AWS](https://www.vaultproject.io/docs/auth/aws-ec2.html" \t "https://www.springcloud.cc/_top)后端提供了AWS IAM角色的安全认证机制，允许基础上运行的应用程序的当前IAM角色具有拱顶的自动认证。与大多数Vault身份验证后端不同，此后端不需要先部署或设置安全敏感的凭据（令牌，用户名/密码，客户端证书等）。相反，它将AWS视为受信任的第三方，并使用呼叫者使用其IAM凭据签名的4条信息来验证呼叫者确实在使用该IAM角色。

应用程序正在其中运行的当前IAM角色是自动计算的。如果您在AWS ECS上运行应用程序，则该应用程序将使用分配给正在运行的容器的ECS任务的IAM角色。如果您在EC2实例上裸身运行应用程序，则使用的IAM角色将是分配给EC2实例的角色。

使用AWS-IAM身份验证时，您必须在Vault中创建一个角色并将其分配给您的IAM角色。空的role默认为当前IAM角色的友好名称。

**示例102.11 bootstrap.yml具有必需的AWS-IAM身份验证属性**

spring.cloud.vault: authentication: AWS\_IAM

**示例102.12 具有所有AWS-IAM身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: AWS\_IAM aws-iam: role: my-dev-role aws-path: aws server-id: some.server.name

* role设置尝试进行登录的角色的名称。这应该与您的IAM角色绑定。如果未提供，则当前IAM用户的友好名称将用作保管库角色。
* aws-path设置要使用的AWS装载的路径
* server-id设置用于X-Vault-AWS-IAM-Server-ID标头的值，以防止某些类型的重放攻击。

AWS-IAM需要AWS Java SDK依赖项（com.amazonaws:aws-java-sdk-core），因为身份验证实现将AWS开发工具包类型用于凭证和请求签名。

另请参阅：[Vault文档：使用aws auth后端](https://www.vaultproject.io/docs/auth/aws.html" \t "https://www.springcloud.cc/_top)

## **102.6 Azure MSI身份验证**

所述[天青](https://www.vaultproject.io/docs/auth/azure.html" \t "https://www.springcloud.cc/_top) AUTH后端提供了一个安全导入机构为天青VM实例，允许的Vault令牌自动检索。与大多数Vault身份验证后端不同，此后端不需要先部署或设置安全敏感的凭据（令牌，用户名/密码，客户端证书等）。而是将Azure视为受信任的第三方，并使用可以绑定到VM实例的托管服务身份和实例元数据信息。

**示例102.13 bootstrap.yml具有必需的Azure身份验证属性**

spring.cloud.vault: authentication: AZURE\_MSI azure-msi: role: my-dev-role

**示例102.14 具有所有Azure身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: AZURE\_MSI azure-msi: role: my-dev-role azure-path: azure

* role设置尝试进行登录的角色的名称。
* azure-path设置要使用的Azure装载的路径

Azure MSI身份验证从实例元数据服务中获取有关虚拟机的环境详细信息（订阅ID，资源组，VM名称）。

另请参阅：[Vault文档：使用azure auth后端](https://www.vaultproject.io/docs/auth/azure.html" \t "https://www.springcloud.cc/_top)

## **102.7 TLS证书认证**

cert身份验证后端允许使用由CA签名或自签名的SSL / TLS客户端证书进行身份验证。

要启用cert身份验证，您需要：

1. 使用SSL，请参见[第108章，](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.ssl" \o "108. Vault Client SSL配置)*[Vault客户端SSL配置](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.ssl" \o "108. Vault Client SSL配置)*
2. 配置包含客户端证书和私钥的Java Keystore
3. 将spring.cloud.vault.authentication设置为CERT

**示例102.15 bootstrap.yml**

spring.cloud.vault: authentication: CERT ssl: key-store: classpath:keystore.jks key-store-password: changeit cert-auth-path: cert

另请参阅：[Vault文档：使用Cert auth后端](https://www.vaultproject.io/docs/auth/cert.html" \t "https://www.springcloud.cc/_top)

## **102.8 Cubbyhole身份验证**

Cubbyhole身份验证使用Vault原语来提供安全的身份验证工作流。Cubbyhole身份验证使用令牌作为主要登录方法。临时令牌用于从Vault的Cubbyhole秘密后端获取第二个登录VaultToken。登录令牌通常寿命更长，并且可以与Vault进行交互。登录令牌将从存储在/cubbyhole/response的包装响应中检索。

**创建包装的令牌**

|  |
| --- |
| IMG_601 |
| 用于令牌创建的响应包装要求Vault 0.6.0或更高。 |

**示例102.16 创建和存储令牌**

$ vault token-create -wrap-ttl="10m"

Key Value

--- -----

wrapping\_token: 397ccb93-ff6c-b17b-9389-380b01ca2645

wrapping\_token\_ttl: 0h10m0s

wrapping\_token\_creation\_time: 2016-09-18 20:29:48.652957077 +0200 CEST

wrapped\_accessor: 46b6aebb-187f-932a-26d7-4f3d86a68319

**示例102.17 bootstrap.yml**

spring.cloud.vault: authentication: CUBBYHOLE token: 397ccb93-ff6c-b17b-9389-380b01ca2645

也可以看看：

* [Vault文档：令牌](https://www.vaultproject.io/docs/concepts/tokens.html" \t "https://www.springcloud.cc/_top)
* [Vault文档：Cubbyhole秘密后端](https://www.vaultproject.io/docs/secrets/cubbyhole/index.html" \t "https://www.springcloud.cc/_top)
* [Vault文档：响应包装](https://www.vaultproject.io/docs/concepts/response-wrapping.html" \t "https://www.springcloud.cc/_top)

## **102.9 GCP-GCE认证**

该[GCP](https://www.vaultproject.io/docs/auth/gcp.html" \t "https://www.springcloud.cc/_top) AUTH后端允许Vault通过使用现有的GCP（谷歌云端平台）IAM和GCE凭证登录。

GCP GCE（Google Compute引擎）身份验证为服务帐户创建JSON Web令牌（JWT）形式的签名。使用[实例标识](https://cloud.google.com/compute/docs/instances/verifying-instance-identity" \t "https://www.springcloud.cc/_top)从GCE元数据服务获得Compute Engine实例的JWT 。该API创建一个JSON Web令牌，可用于确认实例身份。

与大多数Vault身份验证后端不同，此后端不需要先部署或设置安全敏感的凭据（令牌，用户名/密码，客户端证书等）。相反，它将GCP视为受信任的第三方，并使用经过加密签名的动态元数据信息来唯一表示每个GCP服务帐户。

**示例102.18 bootstrap.yml具有必需的GCP-GCE身份验证属性**

spring.cloud.vault: authentication: GCP\_GCE gcp-gce: role: my-dev-role

**示例102.19 具有所有GCP-GCE身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: GCP\_GCE gcp-gce: gcp-path: gcp role: my-dev-role service-account: my-service@projectid.iam.gserviceaccount.com

* role设置尝试进行登录的角色的名称。
* gcp-path设置要使用的GCP安装架的路径
* service-account允许将服务帐户ID覆盖为特定值。默认为default服务帐户。

也可以看看：

* [Vault文档：使用GCP身份验证后端](https://www.vaultproject.io/docs/auth/gcp.html" \t "https://www.springcloud.cc/_top)
* [GCP文档：验证实例的身份](https://cloud.google.com/compute/docs/instances/verifying-instance-identity" \t "https://www.springcloud.cc/_top)

## **102.10 GCP-IAM身份验证**

该[GCP](https://www.vaultproject.io/docs/auth/gcp.html" \t "https://www.springcloud.cc/_top) AUTH后端允许Vault通过使用现有的GCP（谷歌云端平台）IAM和GCE凭证登录。

GCP IAM身份验证为服务帐户创建JSON Web令牌（JWT）形式的签名。通过调用GCP IAM的[projects.serviceAccounts.signJwt](https://cloud.google.com/iam/reference/rest/v1/projects.serviceAccounts/signJwt" \t "https://www.springcloud.cc/_top)API 获得服务帐户的JWT 。呼叫者针对GCP IAM进行身份验证，从而证明其身份。此Vault后端将GCP视为受信任的第三方。

IAM凭证可以从运行时环境（特别是[GOOGLE\_APPLICATION\_CREDENTIALS](https://cloud.google.com/docs/authentication/production" \t "https://www.springcloud.cc/_top) 环境变量），Google Compute元数据服务获得，也可以从外部以JSON或base64编码的形式提供。JSON是首选格式，因为它带有调用projects.serviceAccounts.signJwt所需的项目ID和服务帐户标识符。

**示例102.20 bootstrap.yml具有必需的GCP-IAM身份验证属性**

spring.cloud.vault: authentication: GCP\_IAM gcp-iam: role: my-dev-role

**示例102.21。具有所有GCP-IAM身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: GCP\_IAM gcp-iam: credentials: location: classpath:credentials.json encoded-key: e+KApn0= gcp-path: gcp jwt-validity: 15m project-id: my-project-id role: my-dev-role service-account-id: my-service@projectid.iam.gserviceaccount.com

* role设置尝试进行登录的角色的名称。
* credentials.location包含JSON格式的Google凭据的凭据资源的路径。
* credentials.encoded-key JSON格式的OAuth2帐户私钥的base64编码内容。
* gcp-path设置要使用的GCP安装架的路径
* jwt-validity配置JWT令牌有效性。默认为15分钟。
* project-id允许将项目ID覆盖为特定值。从获得的凭据中默认为项目ID。
* service-account允许将服务帐户ID覆盖为特定值。默认为获取的凭证中的服务帐户。

GCP IAM身份验证需要Google Cloud Java SDK依赖项（com.google.apis:google-api-services-iam和com.google.auth:google-auth-library-oauth2-http），因为身份验证实现使用Google API进行凭据和JWT签名。

|  |
| --- |
| IMG_602 |
| Google凭据需要OAuth 2令牌来维护令牌的生命周期。所有API都是同步的，因此GcpIamAuthentication不支持AuthenticationSteps，这是无功使用所必需的。 |

也可以看看：

* [Vault文档：使用GCP身份验证后端](https://www.vaultproject.io/docs/auth/gcp.html" \t "https://www.springcloud.cc/_top)
* [GCP文档：projects.serviceAccounts.signJwt](https://cloud.google.com/iam/reference/rest/v1/projects.serviceAccounts/signJwt" \t "https://www.springcloud.cc/_top)

## **102.11 Kubernetes身份验证**

Kubernetes身份验证机制（自Vault 0.8.3起）允许使用Kubernetes服务帐户令牌向Vault进行身份验证。身份验证基于角色，并且角色绑定到服务帐户名称和名称空间。

包含Pod服务帐户的JWT令牌的文件会自动安装在/var/run/secrets/kubernetes.io/serviceaccount/token中。

**示例102.22 具有所有Kubernetes身份验证属性的bootstrap.yml**

spring.cloud.vault: authentication: KUBERNETES kubernetes: role: my-dev-role kubernetes-path: kubernetes service-account-token-file: /var/run/secrets/kubernetes.io/serviceaccount/token

* role设置角色。
* kubernetes-path设置要使用的Kubernetes安装路径。
* service-account-token-file设置包含Kubernetes服务帐户令牌的文件的位置。默认为/var/run/secrets/kubernetes.io/serviceaccount/token。

也可以看看：

* [Vault文档：Kubernetes](https://www.vaultproject.io/docs/auth/kubernetes.html" \t "https://www.springcloud.cc/_top)
* [Kubernetes文档：为Pod配置服务帐户](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/" \t "https://www.springcloud.cc/_top)

## **103.秘密后端**

## **103.1通用后端**

Spring Cloud Vault在基本级别上支持通用秘密后端。通用机密后端允许将任意值存储为键值存储。单个上下文可以存储一个或多个键值元组。上下文可以按层次进行组织。Spring Cloud Vault允许将应用程序名称和默认上下文名称（application）与活动配置文件结合使用。

/secret/{application}/{profile}

/secret/{application}

/secret/{default-context}/{profile}

/secret/{default-context}

应用程序名称由以下属性确定：

* spring.cloud.vault.generic.application-name
* spring.cloud.vault.application-name
* spring.application.name

可以通过在通用后端中的其他上下文中获取秘密，方法是将其路径添加到应用程序名称中，并用逗号分隔。例如，给定应用程序名称usefulapp,mysql1,projectx/aws，将使用以下每个文件夹：

* /secret/usefulapp
* /secret/mysql1
* /secret/projectx/aws

Spring Cloud Vault将所有活动配置文件添加到可能的上下文路径列表中。没有活动的配置文件将跳过使用配置文件名称的访问上下文。

Properties就像存储时一样暴露（即没有其他前缀）。

spring.cloud.vault: generic: enabled: **true** backend: secret profile-separator: '/' default-context: application application-name: my-app

* enabled将此值设置为false会禁用秘密后端配置使用
* backend设置要使用的秘密装载的路径
* default-context设置所有应用程序使用的上下文名称
* application-name覆盖在通用后端中使用的应用程序名称
* profile-separator在带有配置文件的属性源中将配置文件名称与上下文分开

|  |
| --- |
| IMG_603 |
| 键值秘密后端可以在版本控制（v2）和非版本控制（v1）模式下运行。根据操作模式，需要不同的API来访问机密。确保为非版本化的键值后端启用generic秘密后端使用，并为版本化的键值后端启用kv秘密后端使用。 |

另请参阅：[Vault文档：使用KV秘密引擎-版本1（通用秘密后端）](https://www.vaultproject.io/docs/secrets/kv/kv-v1.html" \t "https://www.springcloud.cc/_top)

## **103.2版本化键值后端**

Spring Cloud Vault支持版本化的键值机密后端。键值后端允许存储任意值作为键值存储。单个上下文可以存储一个或多个键值元组。上下文可以按层次进行组织。Spring Cloud Vault允许将应用程序名称和默认上下文名称（application）与活动配置文件结合使用。

/secret/{application}/{profile}

/secret/{application}

/secret/{default-context}/{profile}

/secret/{default-context}

应用程序名称由以下属性确定：

* spring.cloud.vault.kv.application-name
* spring.cloud.vault.application-name
* spring.application.name

可以通过在键值后端的其他上下文中获取秘密，方法是将其路径添加到应用程序名称中，并以逗号分隔。例如，给定应用程序名称usefulapp,mysql1,projectx/aws，将使用以下每个文件夹：

* /secret/usefulapp
* /secret/mysql1
* /secret/projectx/aws

Spring Cloud Vault将所有活动配置文件添加到可能的上下文路径列表中。没有活动的配置文件将跳过使用配置文件名称的访问上下文。

Properties就像存储时一样暴露（即没有其他前缀）。

|  |
| --- |
| IMG_604 |
| Spring Cloud Vault在安装路径和实际上下文路径之间添加data/上下文。 |

spring.cloud.vault: kv: enabled: **true** backend: secret profile-separator: '/' default-context: application application-name: my-app

* enabled将此值设置为false会禁用秘密后端配置使用
* backend设置要使用的秘密装载的路径
* default-context设置所有应用程序使用的上下文名称
* application-name覆盖在通用后端中使用的应用程序名称
* profile-separator在带有配置文件的属性源中将配置文件名称与上下文分开

|  |
| --- |
| IMG_605 |
| 键值秘密后端可以在版本控制（v2）和非版本控制（v1）模式下运行。根据操作模式，需要不同的API来访问机密。确保为非版本化的键值后端启用generic秘密后端使用，并为版本化的键值后端启用kv秘密后端使用。 |

另请参阅：[Vault文档：使用KV Secrets Engine-版本2（版本化的键值后端）](https://www.vaultproject.io/docs/secrets/kv/kv-v2.html" \t "https://www.springcloud.cc/_top)

## **103.3 Consul**

Spring Cloud Vault可以获取HashiCorp Consul的凭据。Consul集成需要spring-cloud-vault-config-consul依赖性。

**示例103.1 pom.xml**

<dependencies>

<dependency>

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

<artifactId>spring-cloud-vault-config-consul</artifactId>

<version>{project-version}</version>

</dependency></dependencies>

可以通过设置spring.cloud.vault.consul.enabled=true（默认值为false）并为角色名称提供spring.cloud.vault.consul.role=…来启用集成。

获得的令牌存储在spring.cloud.consul.token中，因此使用Spring Cloud Consul可以拾取生成的凭据，而无需进一步配置。您可以通过设置spring.cloud.vault.consul.token-property来配置属性名称。

spring.cloud.vault: consul: enabled: **true** role: readonly backend: consul token-property: spring.cloud.consul.token

* enabled将此值设置为true会启用Consul后端配置用法
* role设置Consul角色定义的角色名称
* backend设置要使用的Consul安装的路径
* token-property设置存储Consul ACL令牌的属性名称

另请参阅：[Vault文档：使用Vault设置Consul](https://www.vaultproject.io/docs/secrets/consul/index.html" \t "https://www.springcloud.cc/_top)

## **103.4 RabbitMQ**

Spring Cloud Vault可以获取RabbitMQ的凭据。

RabbitMQ集成需要spring-cloud-vault-config-rabbitmq依赖性。

**示例103.2 pom.xml**

<dependencies>

<dependency>

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

<artifactId>spring-cloud-vault-config-rabbitmq</artifactId>

<version>{project-version}</version>

</dependency></dependencies>

可以通过设置spring.cloud.vault.rabbitmq.enabled=true（默认为false）并为角色名称提供spring.cloud.vault.rabbitmq.role=…来启用集成。

用户名和密码存储在spring.rabbitmq.username和spring.rabbitmq.password中，因此使用Spring Boot将无需进一步配置即可获取生成的凭据。您可以通过设置spring.cloud.vault.rabbitmq.username-property和spring.cloud.vault.rabbitmq.password-property来配置属性名称。

spring.cloud.vault: rabbitmq: enabled: **true** role: readonly backend: rabbitmq username-property: spring.rabbitmq.username password-property: spring.rabbitmq.password

* enabled将此值设置为true可启用RabbitMQ后端配置用法
* role设置RabbitMQ角色定义的角色名称
* backend设置要使用的RabbitMQ支架的路径
* username-property设置存储RabbitMQ用户名的属性名称
* password-property设置存储RabbitMQ密码的属性名称

另请参阅：[Vault文档：使用Vault设置RabbitMQ](https://www.vaultproject.io/docs/secrets/rabbitmq/index.html" \t "https://www.springcloud.cc/_top)

## **103.5 AWS**

Spring Cloud Vault可以获取AWS的凭证。

AWS集成需要spring-cloud-vault-config-aws依赖性。

**示例103.3 pom.xml**

<dependencies>

<dependency>

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

<artifactId>spring-cloud-vault-config-aws</artifactId>

<version>{project-version}</version>

</dependency></dependencies>

可以通过设置spring.cloud.vault.aws=true（默认值为false）并为角色名称提供spring.cloud.vault.aws.role=…来启用集成。

访问密钥和秘密密钥存储在cloud.aws.credentials.accessKey和cloud.aws.credentials.secretKey中，因此使用Spring Cloud AWS将无需进一步配置即可获取生成的凭证。您可以通过设置spring.cloud.vault.aws.access-key-property和spring.cloud.vault.aws.secret-key-property来配置属性名称。

spring.cloud.vault: aws: enabled: **true** role: readonly backend: aws access-key-property: cloud.aws.credentials.accessKey secret-key-property: cloud.aws.credentials.secretKey

* enabled将此值设置为true可启用AWS后端配置
* role设置AWS角色定义的角色名称
* backend设置要使用的AWS装载的路径
* access-key-property设置存储AWS访问密钥的属性名称
* secret-key-property设置存储AWS密钥的属性名称

另请参阅：[Vault文档：通过Vault设置AWS](https://www.vaultproject.io/docs/secrets/aws/index.html" \t "https://www.springcloud.cc/_top)

## **104.数据库后端**

Vault支持多个数据库机密后端，以根据配置的角色动态生成数据库凭证。这意味着需要访问数据库的服务不再需要配置凭据：它们可以从Vault请求它们，并使用Vault的租赁机制更轻松地滚动密钥。

Spring Cloud Vault与以下后端集成：

* [第104.1节“数据库”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.database" \o "104.1数据库)
* [第104.2节“ Apache Cassandra”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.cassandra" \o "104.2 Apache卡桑德拉)
* [第104.3节“ MongoDB”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.mongodb" \o "104.3 MongoDB)
* [第104.4节“ MySQL”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.mysql" \o "104.4 MySQL)
* [第104.5节“ PostgreSQL”](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "vault.config.backends.postgresql" \o "104.5 PostgreSQL)

使用数据库秘密后端需要启用配置中的后端和spring-cloud-vault-config-databases依赖性。

Vault自0.7.1起发布，带有专用的database秘密后端，该后端允许通过插件集成数据库。您可以通过使用通用数据库后端来使用该特定后端。确保指定适当的后端路径，例如spring.cloud.vault.mysql.role.backend=database。

**示例104.1 pom.xml**

<dependencies>

<dependency>

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

<artifactId>spring-cloud-vault-config-databases</artifactId>

<version>{project-version}</version>

</dependency></dependencies>

|  |
| --- |
| IMG_606 |
| 启用多个JDBC兼容数据库将默认生成凭据并将其存储在相同的属性密钥中，因此JDBC机密的属性名称需要单独配置。 |

## **104.1数据库**

Spring Cloud Vault可以获取[https://www.vaultproject.io/api/secret/databases/index.html上](https://www.vaultproject.io/api/secret/databases/index.html" \t "https://www.springcloud.cc/_top)列出的任何数据库的凭据 。可以通过设置spring.cloud.vault.database.enabled=true（默认为false）并为角色名称提供spring.cloud.vault.database.role=…来启用集成。

虽然数据库后端是通用后端，但spring.cloud.vault.database专门针对JDBC数据库。用户名和密码存储在spring.datasource.username和spring.datasource.password中，因此使用Spring Boot将为DataSource获取生成的凭据，而无需进一步配置。您可以通过设置spring.cloud.vault.database.username-property和spring.cloud.vault.database.password-property来配置属性名称。

spring.cloud.vault: database: enabled: **true** role: readonly backend: database username-property: spring.datasource.username password-property: spring.datasource.password

* enabled将此值设置为true可启用数据库后端配置使用
* role设置数据库角色定义的角色名称
* backend设置要使用的数据库安装路径
* username-property设置存储数据库用户名的属性名称
* password-property设置存储数据库密码的属性名称

另请参阅：[Vault文档：数据库秘密后端](https://www.vaultproject.io/docs/secrets/databases/index.html" \t "https://www.springcloud.cc/_top)

|  |
| --- |
| IMG_607 |
| 当达到最大租用时间时，Spring Cloud Vault不支持获取新凭据并使用它们配置DataSource。也就是说，如果将Vault中“数据库”角色的max\_ttl设置为24h，则意味着在应用程序启动后24小时，它将无法再通过数据库进行身份验证。 |

## **104.2 Apache Cassandra**

|  |
| --- |
| IMG_608 |
| cassandra后端在Vault 0.7.1中已弃用，建议使用database后端并将其安装为cassandra。 |

Spring Cloud Vault可以获取Apache Cassandra的凭据。可以通过设置spring.cloud.vault.cassandra.enabled=true（默认为false）并为角色名称提供spring.cloud.vault.cassandra.role=…来启用集成。

用户名和密码存储在spring.data.cassandra.username和spring.data.cassandra.password中，因此使用Spring Boot将无需进一步配置即可获取生成的凭据。您可以通过设置spring.cloud.vault.cassandra.username-property和spring.cloud.vault.cassandra.password-property来配置属性名称。

spring.cloud.vault: cassandra: enabled: **true** role: readonly backend: cassandra username-property: spring.data.cassandra.username password-property: spring.data.cassandra.password

* enabled将此值设置为true可启用Cassandra后端配置用法
* role设置Cassandra角色定义的角色名称
* backend设置要使用的Cassandra支架的路径
* username-property设置存储Cassandra用户名的属性名称
* password-property设置存储Cassandra密码的属性名称

另请参阅：[Vault文档：使用Vault设置Apache Cassandra](https://www.vaultproject.io/docs/secrets/cassandra/index.html" \t "https://www.springcloud.cc/_top)

## **104.3 MongoDB**

|  |
| --- |
| IMG_609 |
| mongodb后端在Vault 0.7.1中已弃用，建议使用database后端并将其安装为mongodb。 |

Spring Cloud Vault可以获取MongoDB的凭据。可以通过设置spring.cloud.vault.mongodb.enabled=true（默认值为false）并为角色名称提供spring.cloud.vault.mongodb.role=…来启用集成。

用户名和密码存储在spring.data.mongodb.username和spring.data.mongodb.password中，因此使用Spring Boot将无需进一步配置即可获取生成的凭据。您可以通过设置spring.cloud.vault.mongodb.username-property和spring.cloud.vault.mongodb.password-property来配置属性名称。

spring.cloud.vault: mongodb: enabled: **true** role: readonly backend: mongodb username-property: spring.data.mongodb.username password-property: spring.data.mongodb.password

* enabled将此值设置为true启用MongodB后端配置使用
* role设置MongoDB角色定义的角色名称
* backend设置要使用的MongoDB安装的路径
* username-property设置存储MongoDB用户名的属性名称
* password-property设置存储MongoDB密码的属性名称

另请参阅：[Vault文档：使用Vault设置MongoDB](https://www.vaultproject.io/docs/secrets/mongodb/index.html" \t "https://www.springcloud.cc/_top)

## **104.4 MySQL**

|  |
| --- |
| IMG_610 |
| mysql后端在Vault 0.7.1中已弃用，建议使用database后端并将其安装为mysql。spring.cloud.vault.mysql的配置将在以后的版本中删除。 |

Spring Cloud Vault可以获取MySQL的凭据。可以通过设置spring.cloud.vault.mysql.enabled=true（默认值为false）并为角色名称提供spring.cloud.vault.mysql.role=…来启用集成。

用户名和密码存储在spring.datasource.username和spring.datasource.password中，因此使用Spring Boot将无需进一步配置即可获取生成的凭据。您可以通过设置spring.cloud.vault.mysql.username-property和spring.cloud.vault.mysql.password-property来配置属性名称。

spring.cloud.vault: mysql: enabled: **true** role: readonly backend: mysql username-property: spring.datasource.username password-property: spring.datasource.password

* enabled将此值设置为true可启用MySQL后端配置
* role设置MySQL角色定义的角色名称
* backend设置要使用的MySQL挂载路径
* username-property设置存储MySQL用户名的属性名称
* password-property设置存储MySQL密码的属性名称

另请参阅：[Vault文档：使用Vault设置MySQL](https://www.vaultproject.io/docs/secrets/mysql/index.html" \t "https://www.springcloud.cc/_top)

## **104.5 PostgreSQL**

|  |
| --- |
| IMG_611 |
| postgresql后端在Vault 0.7.1中已弃用，建议使用database后端并将其安装为postgresql。spring.cloud.vault.postgresql的配置将在以后的版本中删除。 |

Spring Cloud Vault可以获取PostgreSQL的凭据。可以通过设置spring.cloud.vault.postgresql.enabled=true（默认值为false）并为角色名称提供spring.cloud.vault.postgresql.role=…来启用集成。

用户名和密码存储在spring.datasource.username和spring.datasource.password中，因此使用Spring Boot将无需进一步配置即可获取生成的凭据。您可以通过设置spring.cloud.vault.postgresql.username-property和spring.cloud.vault.postgresql.password-property来配置属性名称。

spring.cloud.vault: postgresql: enabled: **true** role: readonly backend: postgresql username-property: spring.datasource.username password-property: spring.datasource.password

* enabled将此值设置为true可以启用PostgreSQL后端配置
* role设置PostgreSQL角色定义的角色名称
* backend设置要使用的PostgreSQL安装路径
* username-property设置存储PostgreSQL用户名的属性名称
* password-property设置存储PostgreSQL密码的属性名称

另请参阅：[Vault文档：使用Vault设置PostgreSQL](https://www.vaultproject.io/docs/secrets/postgresql/index.html" \t "https://www.springcloud.cc/_top)

## **105.配置PropertySourceLocator行为**

Spring Cloud Vault使用基于属性的配置为通用和发现的秘密后端创建PropertySource。

发现的后端提供VaultSecretBackendDescriptor beans来描述将机密后端用作PropertySource的配置状态。要创建包含路径，名称和属性转换配置的SecretBackendMetadata对象，需要使用SecretBackendMetadataFactory。

SecretBackendMetadata用于支持特定的PropertySource。

您可以注册任意数量的beans实现VaultConfigurer进行自定义。如果Spring Cloud Vault发现至少一个VaultConfigurer bean，则会禁用默认的通用和发现的后端注册。但是，您可以使用SecretBackendConfigurer.registerDefaultGenericSecretBackends()和SecretBackendConfigurer.registerDefaultDiscoveredSecretBackends()启用默认注册。

**public** **class** CustomizationBean **implements** VaultConfigurer {

@Override

**public** **void** addSecretBackends(SecretBackendConfigurer configurer) {

configurer.add("secret/my-application");

configurer.registerDefaultGenericSecretBackends(false);

configurer.registerDefaultDiscoveredSecretBackends(true);

}

}

|  |
| --- |
| IMG_612 |
| 所有定制都必须在引导上下文中进行。将配置类添加到应用程序中org.springframework.cloud.bootstrap.BootstrapConfiguration的META-INF/spring.factories中。 |

## **106.服务注册表配置**

您可以通过设置spring.cloud.vault.discovery.enabled = true（默认值为false），使用DiscoveryClient（例如来自Spring Cloud Consul的服务器）来定位Vault服务器。最终结果是您的应用程序需要带有适当发现配置的bootstrap.yml（或环境变量）。好处是Vault可以更改其坐标，只要发现服务是固定点即可。默认服务ID为vault，但是您可以使用spring.cloud.vault.discovery.serviceId在客户端上更改它。

发现客户端实现均支持某种元数据映射（例如，对于Eureka，我们拥有eureka.instance.metadataMap）。服务的某些其他属性可能需要在其服务注册元数据中进行配置，以便客户端可以正确连接。不提供有关传输层安全性详细信息的服务注册中心需要提供scheme元数据条目，以将其设置为https或http。如果未配置任何方案，并且该服务未作为安全服务公开，则配置默认为spring.cloud.vault.scheme，而未设置时为https。

spring.cloud.vault.discovery: enabled: **true** service-id: my-vault-service

## **107. Vault客户端快速失败**

在某些情况下，如果服务无法连接到Vault服务器，则可能无法启动服务。如果这是所需的行为，请设置引导程序配置属性spring.cloud.vault.fail-fast=true，客户端将因异常而停止。

spring.cloud.vault: fail-fast: **true**

## **108. Vault客户端SSL配置**

可以通过设置各种属性来声明性地配置SSL。您可以设置javax.net.ssl.trustStore来配置JVM范围的SSL设置，或者设置spring.cloud.vault.ssl.trust-store来仅为Spring Cloud Vault Config设置SSL设置。

spring.cloud.vault: ssl: trust-store: classpath:keystore.jks trust-store-password: changeit

* trust-store设置信任库的资源。受SSL保护的Vault通信将使用指定的信任库验证Vault SSL证书。
* trust-store-password设置信任库密码

请注意，仅当Apache Http Components或OkHttp客户端位于类路径上时，才能应用配置spring.cloud.vault.ssl.\*。

## **109.租赁生命周期管理（续订和撤销）**

对于每个秘密，Vault都会创建一个租约：元数据，其中包含诸如持续时间，可更新性等信息。

Vault承诺数据将在给定的持续时间内或生存时间（TTL）下有效。租约到期后，Vault可以撤消数据，并且秘密使用者无法再确定其是否有效。

Spring Cloud Vault除了创建登录令牌和机密外，还保持租赁生命周期。就是说，与租约关联的登录令牌和机密计划在租约到期之前直到终端到期之前进行更新。应用程序关闭会撤消获得的登录令牌和可更新的租约。

秘密服务和数据库后端（例如MongoDB或MySQL）通常会生成可更新的租约，因此在应用程序关闭时将禁用生成的凭据。

|  |
| --- |
| IMG_613 |
| 静态令牌不会更新或吊销。 |

默认情况下，租约续订和吊销是启用的，可以通过将spring.cloud.vault.config.lifecycle.enabled设置为false来禁用。不建议使用此方法，因为租约可能到期，并且Spring Cloud Vault无法再访问Vault或使用生成的凭据的服务，并且在应用程序关闭后有效凭据仍处于活动状态。

spring.cloud.vault: config.lifecycle.enabled: **true**

另请参阅：[Vault文档：租赁，续订和吊销](https://www.vaultproject.io/docs/concepts/lease.html" \t "https://www.springcloud.cc/_top)

# **第十五部分。Spring Cloud网关**

**Greenwich SR5**

该项目提供了一个基于Spring生态系统的API网关，其中包括：Spring 5，Spring Boot 2和项目Reactor。Spring Cloud网关的目的是提供一种简单而有效的方法来路由到API，并向它们提供跨领域的关注，例如：安全性，监视/度量和弹性。

## **110.如何包括Spring Cloud网关**

要将Spring Cloud网关包含在项目中，请将该启动器与组org.springframework.cloud和工件ID spring-cloud-starter-gateway一起使用。有关 使用当前Spring Cloud版本Train设置构建系统的详细信息，请参见[Spring Cloud项目页面](https://projects.spring.io/spring-cloud/" \t "https://www.springcloud.cc/_top)。

如果包括启动器，但由于某种原因，您不希望启用网关，请设置spring.cloud.gateway.enabled=false。

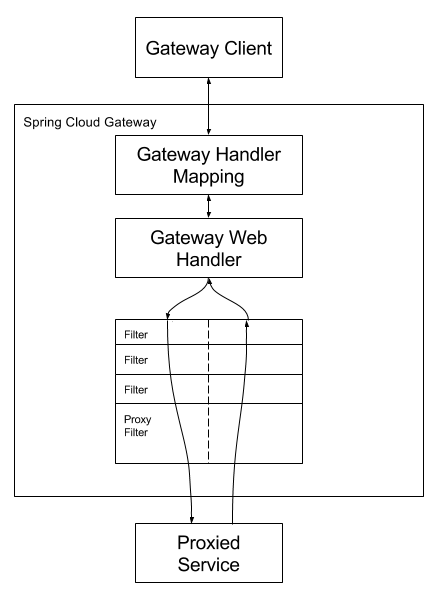
|  |  |
| --- | --- |
| IMG_614 | **重要** |
| Spring Cloud网关基于[Spring Boot 2.x](https://spring.io/projects/spring-boot" \l "learn" \t "https://www.springcloud.cc/_top)， [Spring WebFlux](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html" \t "https://www.springcloud.cc/_top)和[项目Reactor](https://projectreactor.io/docs" \t "https://www.springcloud.cc/_top)[构建](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html" \t "https://www.springcloud.cc/_top)。因此，使用Spring Cloud网关时，许多熟悉的同步库（例如，Spring Data和Spring Security）和模式可能不适用。如果您不熟悉这些项目，建议您在使用Spring Cloud Gateway之前，先阅读它们的文档以熟悉一些新概念。 |

|  |  |
| --- | --- |
| IMG_615 | **重要** |
| Spring Cloud网关需要Spring Boot和Spring Webflux提供的Netty运行时。它不能在传统的Servlet容器中或作为WAR构建。 |

## **111.词汇表**

* **路由**：路由网关的基本构建块。它由ID，目标URI，谓词集合和过滤器集合定义。如果聚合谓词为true，则匹配路由。
* **谓词**：这是[Java 8 Function谓词](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html" \t "https://www.springcloud.cc/_top)。输入类型为[Spring Framework ServerWebExchange](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/ServerWebExchange.html" \t "https://www.springcloud.cc/_top)。这使开发人员可以匹配HTTP请求中的任何内容，例如标头或参数。
* **过滤器**：这些是使用特定工厂构造的实例[Spring Framework GatewayFilter](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/GatewayFilter.html" \t "https://www.springcloud.cc/_top)。在此，可以在发送下游请求之前或之后修改请求和响应。

## **112.工作原理**



客户端向Spring Cloud网关发出请求。如果网关处理程序映射确定请求与路由匹配，则将其发送到网关Web处理程序。该处理程序运行通过特定于请求的筛选器链发送请求。筛选器由虚线分隔的原因是，筛选器可以在发送代理请求之前或之后执行逻辑。执行所有“前置”过滤器逻辑，然后发出代理请求。发出代理请求后，将执行“后”过滤器逻辑。

|  |
| --- |
| IMG_617 |
| 在没有端口的路由中定义的URI将分别将HTTP和HTTPS URI的默认端口分别设置为80和443。 |

## **113.配置路由谓词工厂和网关过滤工厂**

有两种配置谓词和过滤器的方法：快捷方式和完全扩展的参数。下面的大多数示例都使用快捷方式。

名称和自变量名称将在第一部分或每部分的两个部分中以code的形式列出。参数通常按快捷方式配置所需的顺序列出。

## **113.1快捷方式配置**

快捷方式配置由过滤器名称识别，后跟等号（=），后跟由逗号分隔的参数值（,）。

**application.yml。**

spring: cloud: gateway: routes: - id: after\_route uri: https://example.org predicates: - Cookie=mycookie,mycookievalue

先前的示例使用两个参数定义了Cookie Route Predicate Factory，即cookie名称mycookie和与mycookievalue相匹配的值。

## **113.2完全展开的参数**

完全扩展的参数看起来更像带有名称/值对的标准Yaml配置。通常，将有一个name键和一个args键。args键是用于配置谓词或过滤器的键值对的映射。

**application.yml。**

spring: cloud: gateway: routes: - id: after\_route uri: https://example.org predicates: - name: Cookie args: name: mycookie regexp: mycookievalue

这是上面显示的Cookie谓词的快捷方式配置的完整配置。

## **114.路由谓词工厂**

Spring Cloud网关将路由匹配为Spring WebFlux HandlerMapping基础结构的一部分。Spring Cloud网关包括许多内置的Route Predicate工厂。所有这些谓词都与HTTP请求的不同属性匹配。多个路由谓词工厂可以合并，也可以通过逻辑and合并。

## **114.1路由谓词工厂之后**

After路由谓词工厂采用一个参数，即datetime（这是Java ZonedDateTime）。该谓词匹配在当前日期时间之后发生的请求。

**application.yml。**

spring: cloud: gateway: routes: - id: after\_route uri: https://example.org predicates: - After=2017-01-20T17:42:47.789-07:00[America/Denver**]**

该路线与2017年1月20日17:42山区时间（丹佛）之后的所有请求匹配。

## **114.2路线谓词工厂之前**

Before路由谓词工厂采用一个参数datetime（它是Java ZonedDateTime）。该谓词匹配当前日期时间之前发生的请求。

**application.yml。**

spring: cloud: gateway: routes: - id: before\_route uri: https://example.org predicates: - Before=2017-01-20T17:42:47.789-07:00[America/Denver**]**

该路线与2017年1月20日17:42山区时间（丹佛）之前的所有请求匹配。

## **114.3路由谓词工厂之间**

Between路由谓词工厂采用两个参数datetime1和datetime2，它们是Java ZonedDateTime对象。该谓词匹配在datetime1之后和datetime2之前发生的请求。datetime2参数必须在datetime1之后。

**application.yml。**

spring: cloud: gateway: routes: - id: between\_route uri: https://example.org predicates: - Between=2017-01-20T17:42:47.789-07:00[America/Denver]**,** 2017-01-21T17:42:47.789-07:00[America/Denver**]**

该路线与2017年1月20日山区时间（丹佛）之后和2017年1月21日17:42山区时间（丹佛）之后的所有请求匹配。这对于维护时段可能很有用。

## **114.4 Cookie路线谓词工厂**

Cookie Route Predicate Factory采用两个参数，即cookie name和regexp（这是Java正则表达式）。该谓词匹配具有给定名称的cookie，并且值匹配正则表达式。

**application.yml。**

spring: cloud: gateway: routes: - id: cookie\_route uri: https://example.org predicates: - Cookie=chocolate**,** ch.p

此路由与请求匹配，具有一个名为chocolate的cookie，该cookie的值与ch.p正则表达式匹配。

## **114.5标头路由谓词工厂**

Header Route Predicate Factory具有两个参数，标头name和regexp（这是Java正则表达式）。该谓词与具有给定名称的标头匹配，并且值与正则表达式匹配。

**application.yml。**

spring: cloud: gateway: routes: - id: header\_route uri: https://example.org predicates: - Header=X-Request-Id**,** \d+

如果请求具有名为X-Request-Id的标头，且其值与\d+正则表达式匹配（具有一个或多个数字的值），则此路由匹配。

## **114.6主机路由谓词工厂**

Host Route Predicate Factory采用一个参数：主机名patterns的列表。模式是Ant样式的模式，以.作为分隔符。该谓词与匹配模式的Host头匹配。

**application.yml。**

spring: cloud: gateway: routes: - id: host\_route uri: https://example.org predicates: - Host=\*\*.somehost.org,\*\*.anotherhost.org

还支持URI模板变量，例如{sub}.myhost.org。

如果请求的Host标头的值为www.somehost.org或beta.somehost.org或www.anotherhost.org，则此路由将匹配。

该谓词提取URI模板变量（如上例中定义的sub）作为名称和值的映射，并使用在ServerWebExchangeUtils.URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE中定义的键将其放置在ServerWebExchange.getAttributes()中。这些值可供[GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-route-filters)使用。

## **114.7方法路线谓词工厂**

Method路由谓词工厂采用一个methods参数，该参数是一个或多个要匹配的HTTP方法。

**application.yml。**

spring: cloud: gateway: routes: - id: method\_route uri: https://example.org predicates: - Method=GET,POST

如果请求方法是GET或POST，则此路由将匹配。

## **114.8路径路线谓词工厂**

Path路由谓词工厂采用两个参数：Spring PathMatcher patterns的列表和matchOptionalTrailingSeparator的可选标志。

**application.yml。**

spring: cloud: gateway: routes: - id: host\_route uri: https://example.org predicates: - Path=/foo/{segment},/bar/{segment**}**

如果请求路径为例如/foo/1或/foo/bar或/bar/baz，则此路由将匹配。

该谓词提取URI模板变量（如以上示例中定义的segment）作为名称和值的映射，并使用在ServerWebExchangeUtils.URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE中定义的键将其放置在ServerWebExchange.getAttributes()中。这些值可供[GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-route-filters)使用。

可以使用实用程序方法来简化对这些变量的访问。

Map<String, String> uriVariables = ServerWebExchangeUtils.getPathPredicateVariables(exchange);

String segment = uriVariables.get("segment");

## **114.9查询路由谓词工厂**

Query Route Predicate Factory采用两个参数：必需的param和可选的regexp（这是Java正则表达式）。

**application.yml。**

spring: cloud: gateway: routes: - id: query\_route uri: https://example.org predicates: - Query=baz

如果请求包含baz查询参数，则此路由将匹配。

**application.yml。**

spring: cloud: gateway: routes: - id: query\_route uri: https://example.org predicates: - Query=foo**,** ba.

如果请求包含一个foo查询参数，其值与ba.正则表达式匹配，则此路由将匹配，因此bar和baz将匹配。

## **114.10 RemoteAddr路由谓词工厂**

RemoteAddr路由谓词工厂采用sources的列表（最小大小1），它是CIDR表示法（IPv4或IPv6）字符串，例如192.168.0.1/16（其中192.168.0.1是IP地址， 16是子网掩码）。

**application.yml。**

spring: cloud: gateway: routes: - id: remoteaddr\_route uri: https://example.org predicates: - RemoteAddr=192.168.1.1/24

如果请求的远程地址为192.168.1.10，则此路由将匹配。

## **114.11重量路线谓词工厂**

Weight Route Predicate Factory接受两个参数group和weight（一个int）。权重是按组计算的。

**application.yml。**

spring: cloud: gateway: routes: - id: weight\_high uri: https://weighthigh.org predicates: - Weight=group1**,** 8 - id: weight\_low uri: https://weightlow.org predicates: - Weight=group1**,** 2

此路由会将约80％的流量转发到[https://weighthigh.org，](https://weighthigh.org/" \t "https://www.springcloud.cc/_top)并将约20％的流量转发到[https://weighlow.org](https://weighlow.org/" \t "https://www.springcloud.cc/_top)

### **114.11.1修改解析远程地址的方式**

默认情况下，RemoteAddr路由谓词工厂使用传入请求中的远程地址。如果Spring Cloud网关位于代理层后面，则此地址可能与实际的客户端IP地址不匹配。

您可以通过设置自定义RemoteAddressResolver来自定义解析远程地址的方式。Spring Cloud网关带有一个基于[X-Forwarded-For标头](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For" \t "https://www.springcloud.cc/_top) XForwardedRemoteAddressResolver的非默认远程地址解析器。

XForwardedRemoteAddressResolver有两个静态构造方法，它们采用不同的安全性方法：

XForwardedRemoteAddressResolver::trustAll返回一个RemoteAddressResolver，该地址始终使用在X-Forwarded-For标头中找到的第一个IP地址。这种方法容易受到欺骗，因为恶意客户端可能会为X-Forwarded-For设置一个初始值，该初始值将被解析程序接受。

XForwardedRemoteAddressResolver::maxTrustedIndex获取一个索引，该索引与在Spring Cloud网关前面运行的受信任基础结构的数量相关。例如，如果Spring Cloud网关只能通过HAProxy访问，则应使用值1。如果在访问Spring Cloud网关之前需要两跳可信基础结构，则应使用值2。

给定以下标头值：

X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3

下面的maxTrustedIndex值将产生以下远程地址。

| **maxTrustedIndex** | **结果** |
| --- | --- |
| [Integer.MIN\_VALUE,0] | (invalid, IllegalArgumentException during initialization) |
| 1 | 0.0.0.3 |
| 2 | 0.0.0.2 |
| 3 | 0.0.0.1 |
| [4, Integer.MAX\_VALUE] | 0.0.0.1 |

使用Java配置：

GatewayConfig.java

RemoteAddressResolver resolver = XForwardedRemoteAddressResolver

.maxTrustedIndex(1);

...

.route("direct-route",

r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")

.uri("https://downstream1")

.route("proxied-route",

r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")

.uri("https://downstream2")

)

## **115.网关过滤器工厂**

路由过滤器允许以某种方式修改传入的HTTP请求或传出的HTTP响应。路由过滤器适用于特定路由。Spring Cloud网关包括许多内置的GatewayFilter工厂。

注意有关如何使用以下任何过滤器的更多详细示例，请查看[单元测试](https://github.com/spring-cloud/spring-cloud-gateway/tree/master/spring-cloud-gateway-core/src/test/java/org/springframework/cloud/gateway/filter/factory" \t "https://www.springcloud.cc/_top)。

## **115.1 AddRequestHeader GatewayFilter工厂**

AddRequestHeader GatewayFilter工厂采用name和value参数。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_request\_header\_route uri: https://example.org filters: - AddRequestHeader=X-Request-Foo**,** Bar

这会将X-Request-Foo:Bar标头添加到所有匹配请求的下游请求标头中。

AddRequestHeader知道用于匹配路径或主机的URI变量。URI变量可用于该值，并将在运行时扩展。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_request\_header\_route uri: https://example.org predicates: - Path=/foo/{segment**}** filters: - AddRequestHeader=X-Request-Foo**,** Bar-{segment**}**

## **115.2 AddRequestParameter GatewayFilter工厂**

AddRequestParameter GatewayFilter工厂采用name和value参数。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_request\_parameter\_route uri: https://example.org filters: - AddRequestParameter=foo**,** bar

这会将foo=bar添加到所有匹配请求的下游请求的查询字符串中。

AddRequestParameter知道用于匹配路径或主机的URI变量。URI变量可用于该值，并将在运行时扩展。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_request\_parameter\_route uri: https://example.org predicates: - Host: {segment}.myhost.org filters: - AddRequestParameter=foo**,** bar-{segment**}**

## **115.3 AddResponseHeader GatewayFilter工厂**

AddResponseHeader GatewayFilter工厂采用name和value参数。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_response\_header\_route uri: https://example.org filters: - AddResponseHeader=X-Response-Foo**,** Bar

这会将X-Response-Foo:Bar标头添加到所有匹配请求的下游响应的标头中。

AddResponseHeader知道用于匹配路径或主机的URI变量。URI变量可用于该值，并将在运行时扩展。

**application.yml。**

spring: cloud: gateway: routes: - id: add\_response\_header\_route uri: https://example.org predicates: - Host: {segment}.myhost.org filters: - AddResponseHeader=foo**,** bar-{segment**}**

## **115.4 DedupeResponseHeader GatewayFilter工厂**

DedupeResponseHeader GatewayFilter工厂采用name参数和可选的strategy参数。name可以包含标题名称列表，以空格分隔。

**application.yml。**

spring: cloud: gateway: routes: - id: dedupe\_response\_header\_route uri: https://example.org filters: - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin

在网关CORS逻辑和下游逻辑都将它们添加的情况下，这将删除Access-Control-Allow-Credentials和Access-Control-Allow-Origin响应头的重复值。

DedupeResponseHeader过滤器还接受可选的strategy参数。可接受的值为RETAIN\_FIRST（默认值），RETAIN\_LAST和RETAIN\_UNIQUE。

## **115.5 Hystrix GatewayFilter工厂**

[Hystrix](https://github.com/Netflix/Hystrix" \t "https://www.springcloud.cc/_top)是Netflix的一个库，它实现了[断路器模式](https://martinfowler.com/bliki/CircuitBreaker.html" \t "https://www.springcloud.cc/_top)。Hystrix GatewayFilter允许您将断路器引入网关路由，保护服务免受级联故障的影响，并允许您在下游故障的情况下提供后备响应。

要在您的项目中启用Hystrix GatewayFilters，请添加对[Spring Cloud Netflix中的](https://cloud.spring.io/spring-cloud-netflix/" \t "https://www.springcloud.cc/_top) spring-cloud-starter-netflix-hystrix的依赖。

Hystrix GatewayFilter工厂需要一个name参数，它是HystrixCommand的名称。

**application.yml。**

spring: cloud: gateway: routes: - id: hystrix\_route uri: https://example.org filters: - Hystrix=myCommandName

这会将其余过滤器包装在命令名称为myCommandName的HystrixCommand中。

Hystrix过滤器还可以接受可选的fallbackUri参数。当前，仅支持forward:计划的URI。如果调用了后备，则请求将被转发到与URI相匹配的控制器。

**application.yml。**

spring: cloud: gateway: routes: - id: hystrix\_route uri: lb://backing-service:8088 predicates: - Path=/consumingserviceendpoint filters: - name: Hystrix args: name: fallbackcmd fallbackUri: forward:/incaseoffailureusethis - RewritePath=/consumingserviceendpoint**,** /backingserviceendpoint

调用Hystrix后备广告时，它将转发到/incaseoffailureusethis URI。请注意，此示例还通过目标URI上的lb前缀演示了（可选）Spring Cloud Netflix Ribbon负载均衡。

主要方案是将fallbackUri用于网关应用程序中的内部控制器或处理程序。但是，也可以将请求重新路由到外部应用程序中的控制器或处理程序，如下所示：

**application.yml。**

spring: cloud: gateway: routes: - id: ingredients uri: lb://ingredients predicates: - Path=//ingredients/\*\* filters: - name: Hystrix args: name: fetchIngredients fallbackUri: forward:/fallback - id: ingredients-fallback uri: http://localhost:9994 predicates: - Path=/fallback

在此示例中，网关应用程序中没有fallback端点或处理程序，但是，另一个应用程序中没有fallback端点或处理程序，已在[http://localhost:9994](http://localhost:9994/" \t "https://www.springcloud.cc/_top)下注册。

如果将请求转发到后备，则Hystrix网关过滤器还会提供引起请求的Throwable。它作为ServerWebExchangeUtils.HYSTRIX\_EXECUTION\_EXCEPTION\_ATTR属性添加到ServerWebExchange中，可以在网关应用程序中处理后备时使用。

对于外部控制器/处理程序方案，可以添加带有异常详细信息的标头。您可以在[FallbackHeaders GatewayFilter Factory部分中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "fallback-headers" \o "115.6 FallbackHeaders GatewayFilter工厂)找到有关它的更多信息。

Hystrix设置（例如超时）可以使用全局默认值进行配置，也可以使用[Hystrix Wiki中](https://github.com/Netflix/Hystrix/wiki/Configuration" \t "https://www.springcloud.cc/_top)所述的应用程序属性在逐条路由的基础上进行配置。

要为上述示例路由设置5秒超时，将使用以下配置：

**application.yml。**

hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseconds: 5000

## **115.6 FallbackHeaders GatewayFilter工厂**

FallbackHeaders工厂允许您在转发到外部应用程序中的fallbackUri的请求的标头中添加Hystrix执行异常详细信息，例如以下情况：

**application.yml。**

spring: cloud: gateway: routes: - id: ingredients uri: lb://ingredients predicates: - Path=//ingredients/\*\* filters: - name: Hystrix args: name: fetchIngredients fallbackUri: forward:/fallback - id: ingredients-fallback uri: http://localhost:9994 predicates: - Path=/fallback filters: - name: FallbackHeaders args: executionExceptionTypeHeaderName: Test-Header

在此示例中，在运行HystrixCommand时发生执行异常之后，该请求将转发到fallback端点或运行在localhost:9994上的应用程序中的处理程序。具有异常类型，消息和-if available-根本原因异常类型和消息的标头将由FallbackHeaders过滤器添加到该请求。

通过设置下面列出的参数的值及其默认值，可以在配置中覆盖标头的名称：

* executionExceptionTypeHeaderName ("Execution-Exception-Type")
* executionExceptionMessageHeaderName ("Execution-Exception-Message")
* rootCauseExceptionTypeHeaderName ("Root-Cause-Exception-Type")
* rootCauseExceptionMessageHeaderName ("Root-Cause-Exception-Message")

您可以在[Hystrix GatewayFilter工厂部分中](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "hystrix" \o "115.5 Hystrix GatewayFilter工厂)找到有关Hystrix与Gateway一起工作的更多信息。

## **115.7 MapRequestHeader GatewayFilter工厂**

MapRequestHeader GatewayFilter要素采用'fromHeader'和'toHeader'参数。它创建一个新的命名标头（toHeader），并从传入的HTTP请求中从现有的命名标头（fromHeader）中提取值。如果输入标头不存在，则过滤器不起作用。如果新的命名标头已经存在，则将使用新值扩充其值。

**application.yml。**

spring: cloud: gateway: routes: - id: map\_request\_header\_route uri: https://example.org filters: - MapRequestHeader=Bar**,** X-Request-Foo

这会将X-Request-Foo:<values>标头添加到下游请求中，并带有来自传入的HTTP请求Bar标头的更新值。

## **115.8 PrefixPath GatewayFilter工厂**

PrefixPath GatewayFilter工厂采用单个prefix参数。

**application.yml。**

spring: cloud: gateway: routes: - id: prefixpath\_route uri: https://example.org filters: - PrefixPath=/mypath

这会将/mypath作为所有匹配请求的路径的前缀。因此，对/hello的请求将被发送到/mypath/hello。

## **115.9 PreserveHostHeader GatewayFilter工厂**

PreserveHostHeader GatewayFilter工厂没有参数。该过滤器设置请求属性，路由过滤器将检查该请求属性以确定是否应发送原始主机头，而不是由HTTP客户端确定的主机头。

**application.yml。**

spring: cloud: gateway: routes: - id: preserve\_host\_route uri: https://example.org filters:

- PreserveHostHeader

## **115.10 RequestRateLimiter GatewayFilter工厂**

RequestRateLimiter GatewayFilter工厂使用RateLimiter实现来确定是否允许继续当前请求。如果不是，则返回状态HTTP 429 - Too Many Requests（默认）。

该过滤器采用一个可选的keyResolver参数和特定于速率限制器的参数（请参见下文）。

keyResolver是实现KeyResolver接口的bean。在配置中，使用SpEL通过名称引用bean。#{@myKeyResolver}是引用名称为myKeyResolver的bean的SpEL表达式。

**KeyResolver.java。**

**public** **interface** KeyResolver {

Mono<String> resolve(ServerWebExchange exchange);

}

KeyResolver接口允许可插拔策略派生用于限制请求的密钥。在未来的里程碑中，将有一些KeyResolver实现。

KeyResolver的默认实现是PrincipalNameKeyResolver，它从ServerWebExchange检索Principal并调用Principal.getName()。

默认情况下，如果KeyResolver未找到密钥，则请求将被拒绝。可以使用spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key（对或错）和spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code属性来调整此行为。

|  |
| --- |
| IMG_618 |
| 无法通过“快捷方式”符号配置RequestRateLimiter。以下示例无效 |

**application.properties。**

# INVALID SHORTCUT CONFIGURATION

spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2, #{@userkeyresolver}

### **115.10.1 Redis RateLimiter**

redis实现基于[Stripe](https://stripe.com/blog/rate-limiters" \t "https://www.springcloud.cc/_top)所做的工作。它需要使用spring-boot-starter-data-redis-reactive Spring Boot起动器。

使用的算法是[令牌桶算法](https://en.wikipedia.org/wiki/Token_bucket" \t "https://www.springcloud.cc/_top)。

redis-rate-limiter.replenishRate是您希望用户每秒允许多少个请求，而没有任何丢弃的请求。这是令牌桶被填充的速率。

redis-rate-limiter.burstCapacity是允许用户在一秒钟内执行的最大请求数。这是令牌桶可以容纳的令牌数。将此值设置为零将阻止所有请求。

通过在replenishRate和burstCapacity中设置相同的值可以达到稳定的速率。通过将burstCapacity设置为高于replenishRate，可以允许临时突发。在这种情况下，速率限制器需要在突发之间间隔一段时间（根据replenishRate），因为2个连续的突发将导致请求丢失（HTTP 429 - Too Many Requests）。

**application.yml。**

spring: cloud: gateway: routes: - id: requestratelimiter\_route uri: https://example.org filters: - name: RequestRateLimiter args: redis-rate-limiter.replenishRate: 10 redis-rate-limiter.burstCapacity: 20

**Config.java。**

@Bean

KeyResolver userKeyResolver() {

**return** exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));

}

这定义了每个用户10的请求速率限制。允许20个突发，但是下一秒只有10个请求可用。KeyResolver是一个简单的参数，它获取user请求参数（注意：不建议在生产中使用）。

速率限制器也可以定义为实现RateLimiter接口的bean。在配置中，使用SpEL通过名称引用bean。#{@myRateLimiter}是一个SpEL表达式，引用名称为myRateLimiter的bean。

**application.yml。**

spring: cloud: gateway: routes: - id: requestratelimiter\_route uri: https://example.org filters: - name: RequestRateLimiter args: rate-limiter: "#{@myRateLimiter}" key-resolver: "#{@userKeyResolver}"

## **115.11重定向到GatewayFilter工厂**

RedirectTo GatewayFilter工厂采用一个status和一个url参数。状态应该是300系列重定向http代码，例如301。URL应该是有效的URL。这将是Location标头的值。

**application.yml。**

spring: cloud: gateway: routes: - id: prefixpath\_route uri: https://example.org filters: - RedirectTo=302**,** https://acme.org

这将发送带有Location:https://acme.org标头的状态302以执行重定向。

## **115.12 RemoveRequestHeader GatewayFilter工厂**

RemoveRequestHeader GatewayFilter工厂采用一个name参数。它是要删除的标题的名称。

**application.yml。**

spring: cloud: gateway: routes: - id: removerequestheader\_route uri: https://example.org filters: - RemoveRequestHeader=X-Request-Foo

这将删除X-Request-Foo标头，然后将其发送到下游。

## **115.13 RemoveResponseHeader GatewayFilter工厂**

RemoveResponseHeader GatewayFilter工厂采用一个name参数。它是要删除的标题的名称。

**application.yml。**

spring: cloud: gateway: routes: - id: removeresponseheader\_route uri: https://example.org filters: - RemoveResponseHeader=X-Response-Foo

这会将X-Response-Foo标头从响应中删除，然后将其返回到网关客户端。

要删除任何类型的敏感标头，应为可能需要的任何路由配置此过滤器。此外，您可以使用spring.cloud.gateway.default-filters一次配置此过滤器，并将其应用于所有路由。

## **115.14 RewritePath GatewayFilter工厂**

RewritePath GatewayFilter工厂采用路径regexp参数和replacement参数。这使用Java正则表达式提供了一种灵活的方式来重写请求路径。

**application.yml。**

spring: cloud: gateway: routes: - id: rewritepath\_route uri: https://example.org predicates: - Path=/foo/\*\* filters: - RewritePath=/foo(?<segment>/?.\*)**,** $\{segment**}**

对于/foo/bar的请求路径，这将在发出下游请求之前将路径设置为/bar。请注意，由于YAML规范，$\被$所取代。

## **115.15 RewriteLocationResponseHeader GatewayFilter工厂**

RewriteLocationResponseHeader GatewayFilter工厂通常会修改Location响应标头的值，以摆脱后端特定的详细信息。它需要stripVersionMode，locationHeaderName，hostValue和protocolsRegex参数。

**application.yml。**

spring: cloud: gateway: routes: - id: rewritelocationresponseheader\_route uri: http://example.org filters: - RewriteLocationResponseHeader=AS\_IN\_REQUEST**,** Location**,** **,**

例如，对于请求POST [https://api.example.com/some/object/name](https://api.example.com/some/object/name" \t "https://www.springcloud.cc/_top)，Location响应标头值[https://object-service.prod.example.net/v2/some/object/id](https://object-service.prod.example.net/v2/some/object/id" \t "https://www.springcloud.cc/_top)将被重写为[https://api.example.com/some/object/id](https://api.example.com/some/object/id" \t "https://www.springcloud.cc/_top)。

参数stripVersionMode具有以下可能的值：NEVER\_STRIP，AS\_IN\_REQUEST（默认），ALWAYS\_STRIP。

* NEVER\_STRIP-即使原始请求路径不包含版本，也不会剥离版本
* AS\_IN\_REQUEST-仅当原始请求路径不包含版本时，版本才会被剥离
* ALWAYS\_STRIP-即使原始请求路径包含版本，也会剥离版本

参数hostValue（如果提供）将用于替换响应Location标头中的host:port部分。如果未提供，将使用Host请求标头的值。

参数protocolsRegex必须是有效的正则表达式String，协议名称将与之匹配。如果不匹配，过滤器将不执行任何操作。默认值为http|https|ftp|ftps。

## **115.16 RewriteResponseHeader GatewayFilter工厂**

RewriteResponseHeader GatewayFilter工厂采用name，regexp和replacement参数。它使用Java正则表达式以灵活的方式重写响应标头值。

**application.yml。**

spring: cloud: gateway: routes: - id: rewriteresponseheader\_route uri: https://example.org filters: - RewriteResponseHeader=X-Response-Foo**,** **,** password=[^&]+**,** password=\*\*\*

对于标头值为/42?user=ford&password=omg!what&flag=true，在发出下游请求后它将被设置为/42?user=ford&password=\*\*\*&flag=true。由于YAML规范，请使用$\来表示$。

## **115.17 SaveSession GatewayFilter工厂**

在向下游转发呼叫之前，SaveSession GatewayFilter工厂强制执行WebSession::save操作。这在将[Spring Session之类](https://projects.spring.io/spring-session/" \t "https://www.springcloud.cc/_top)的内容用于惰性数据存储并且需要确保在进行转接呼叫之前已保存会话状态时特别有用。

**application.yml。**

spring: cloud: gateway: routes: - id: save\_session uri: https://example.org predicates: - Path=/foo/\*\* filters:

- SaveSession

如果您将[Spring Security](https://projects.spring.io/spring-security/" \t "https://www.springcloud.cc/_top)与Spring Session 集成在一起，并且想要确保安全性详细信息已转发到远程进程，则至关重要。

## **115.18 SecureHeaders GatewayFilter工厂**

SecureHeaders GatewayFilter Factory根据[此博客文章](https://blog.appcanary.com/2017/http-security-headers.html" \t "https://www.springcloud.cc/_top)的建议向响应中添加了许多标头。

**添加了以下标头（以及默认值）：**

* X-Xss-Protection:1; mode=block
* Strict-Transport-Security:max-age=631138519
* X-Frame-Options:DENY
* X-Content-Type-Options:nosniff
* Referrer-Policy:no-referrer
* Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline'
* X-Download-Options:noopen
* X-Permitted-Cross-Domain-Policies:none

要更改默认值，请在spring.cloud.gateway.filter.secure-headers名称空间中设置适当的属性：

**Property进行更改：**

* xss-protection-header
* strict-transport-security
* frame-options
* content-type-options
* referrer-policy
* content-security-policy
* download-options
* permitted-cross-domain-policies

要禁用默认值，请使用逗号分隔值设置属性spring.cloud.gateway.filter.secure-headers.disable。

|  |
| --- |
| IMG_619 |
| 需要使用小写和安全标头的全名。 |

**可以使用以下值：**

* x-xss-protection
* strict-transport-security
* x-frame-options
* x-content-type-options
* referrer-policy
* content-security-policy
* x-download-options
* x-permitted-cross-domain-policies

**例：**spring.cloud.gateway.filter.secure-headers.disable=x-frame-options,strict-transport-security

## **115.19 SetPath GatewayFilter工厂**

SetPath GatewayFilter工厂采用路径template参数。通过允许路径的模板段，它提供了一种操作请求路径的简单方法。这将使用Spring Framework中的uri模板。允许多个匹配段。

**application.yml。**

spring: cloud: gateway: routes: - id: setpath\_route uri: https://example.org predicates: - Path=/foo/{segment**}** filters: - SetPath=/{segment**}**

对于/foo/bar的请求路径，这将在发出下游请求之前将路径设置为/bar。

## **115.20 SetRequestHeader GatewayFilter工厂**

SetRequestHeader GatewayFilter工厂采用name和value参数。

**application.yml。**

spring: cloud: gateway: routes: - id: setrequestheader\_route uri: https://example.org filters: - SetRequestHeader=X-Request-Foo**,** Bar

该GatewayFilter用给定的名称替换所有标头，而不是添加。因此，如果下游服务器响应X-Request-Foo:1234，则将其替换为X-Request-Foo:Bar，下游服务将收到此信息。

SetRequestHeader知道用于匹配路径或主机的URI变量。URI变量可用于该值，并将在运行时扩展。

**application.yml。**

spring: cloud: gateway: routes: - id: setrequestheader\_route uri: https://example.org predicates: - Host: {segment}.myhost.org filters: - SetRequestHeader=foo**,** bar-{segment**}**

## **115.21 SetResponseHeader GatewayFilter工厂**

SetResponseHeader GatewayFilter工厂采用name和value参数。

**application.yml。**

spring: cloud: gateway: routes: - id: setresponseheader\_route uri: https://example.org filters: - SetResponseHeader=X-Response-Foo**,** Bar

该GatewayFilter用给定的名称替换所有标头，而不是添加。因此，如果下游服务器以X-Response-Foo:1234响应，则将其替换为X-Response-Foo:Bar，这是网关客户端将收到的内容。

SetResponseHeader知道用于匹配路径或主机的URI变量。URI变量可用于该值，并将在运行时扩展。

**application.yml。**

spring: cloud: gateway: routes: - id: setresponseheader\_route uri: https://example.org predicates: - Host: {segment}.myhost.org filters: - SetResponseHeader=foo**,** bar-{segment**}**

## **115.22 SetStatus GatewayFilter工厂**

SetStatus GatewayFilter工厂采用单个status参数。它必须是有效的Spring HttpStatus。它可以是整数值404，也可以是枚举NOT\_FOUND的字符串表示形式。

**application.yml。**

spring: cloud: gateway: routes: - id: setstatusstring\_route uri: https://example.org filters: - SetStatus=BAD\_REQUEST - id: setstatusint\_route uri: https://example.org filters: - SetStatus=401

无论哪种情况，响应的HTTP状态都将设置为401。

## **115.23 StripPrefix GatewayFilter工厂**

StripPrefix GatewayFilter工厂采用一个参数parts。parts参数指示在向下游发送请求之前，要从请求中剥离的路径中的零件数。

**application.yml。**

spring: cloud: gateway: routes: - id: nameRoot uri: http://nameservice predicates: - Path=/name/\*\* filters: - StripPrefix=2

通过网关发送到/name/bar/foo的请求时，对nameservice的请求将类似于[http://nameservice/foo](http://nameservice/foo" \t "https://www.springcloud.cc/_top)。

## **115.24重试GatewayFilter工厂**

Retry GatewayFilter Factory支持以下参数集：

* retries：应尝试重试的次数
* statuses：应重试的HTTP状态代码，用org.springframework.http.HttpStatus表示
* methods：应重试的HTTP方法，使用org.springframework.http.HttpMethod表示
* series：要重试的一系列状态代码，使用org.springframework.http.HttpStatus.Series表示
* exceptions：应重试引发的异常列表
* backoff：为重试配置了指数补偿。重试在退避间隔firstBackoff \* (factor ^ n)之后执行，其中n是迭代。如果配置了maxBackoff，则应用的最大退避将被限制为maxBackoff。如果basedOnPreviousValue为true，将使用prevBackoff \* factor计算退避。

如果启用了Retry过滤器，则会配置以下默认值：

* retries-3次
* series — 5XX系列
* methods — GET方法
* exceptions-IOException和TimeoutException
* backoff-已禁用

**application.yml。**

spring: cloud: gateway: routes: - id: retry\_test uri: http://localhost:8080/flakey predicates: - Host=\*.retry.com filters: - name: Retry args: retries: 3 statuses: BAD\_GATEWAY methods: GET,POST backoff: firstBackoff: 10ms maxBackoff: 50ms factor: 2 basedOnPreviousValue: **false**

|  |
| --- |
| IMG_620 |
| 当使用带有forward:前缀URL的重试过滤器时，应仔细编写目标端点，以便在发生错误的情况下不会执行任何可能导致响应发送到客户端并提交的操作。例如，如果目标端点是带注释的控制器，则目标控制器方法不应返回带有错误状态代码的ResponseEntity。相反，它应该抛出一个Exception，或者例如通过一个Mono.error(ex)返回值来发出错误信号，可以将重试过滤器配置为通过重试来处理。 |

|  |
| --- |
| IMG_621 |
| 当将重试过滤器与任何具有主体的HTTP方法一起使用时，主体将被缓存，并且网关将受到内存的限制。正文被缓存在ServerWebExchangeUtils.CACHED\_REQUEST\_BODY\_ATTR定义的请求属性中。对象的类型是org.springframework.core.io.buffer.DataBuffer。 |

## **115.25 RequestSize GatewayFilter工厂**

当请求大小大于允许的限制时，RequestSize GatewayFilter工厂可以限制请求到达下游服务。过滤器采用maxSize参数，该参数是请求的允许大小限制。maxSize is a `DataSize类型，因此值可以定义为数字，后跟可选的DataUnit后缀，例如'KB'或'MB'。字节的默认值为“ B”。

**application.yml。**

spring: cloud: gateway: routes: - id: request\_size\_route uri: http://localhost:8080/upload predicates: - Path=/upload filters: - name: RequestSize args: maxSize: 5000000

当请求因大小而被拒绝时，RequestSize GatewayFilter Factory将响应状态设置为413 Payload Too Large，并带有一个附加报头errorMessage。以下是此类errorMessage的示例。

errorMessage : Request size is larger than permissible limit. Request size is 6.0 MB where permissible limit is 5.0 MB

|  |
| --- |
| IMG_622 |
| 如果未在路由定义中作为过滤器参数提供，则默认请求大小将设置为5 MB。 |

## **115.26修改请求正文GatewayFilter工厂**

**该过滤器被认为是BETA，API将来可能会更改**

ModifyRequestBody过滤器可用于在网关向下游发送请求主体之前修改请求主体。

|  |
| --- |
| IMG_623 |
| 只能使用Java DSL配置此过滤器 |

@Bean**public** RouteLocator routes(RouteLocatorBuilder builder) {

**return** builder.routes()

.route("rewrite\_request\_obj", r -> r.host("\*.rewriterequestobj.org")

.filters(f -> f.prefixPath("/httpbin")

.modifyRequestBody(String.**class**, Hello.**class**, MediaType.APPLICATION\_JSON\_VALUE,

(exchange, s) -> **return** Mono.just(**new** Hello(s.toUpperCase())))).uri(uri))

.build();

}

**static** **class** Hello {

String message;

**public** Hello() { }

**public** Hello(String message) {

**this**.message = message;

}

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

## **115.27修改响应主体GatewayFilter工厂**

**该过滤器被认为是BETA，API将来可能会更改**

ModifyResponseBody过滤器可用于在将响应正文发送回客户端之前对其进行修改。

|  |
| --- |
| IMG_624 |
| 只能使用Java DSL配置此过滤器 |

@Bean**public** RouteLocator routes(RouteLocatorBuilder builder) {

**return** builder.routes()

.route("rewrite\_response\_upper", r -> r.host("\*.rewriteresponseupper.org")

.filters(f -> f.prefixPath("/httpbin")

.modifyResponseBody(String.**class**, String.**class**,

(exchange, s) -> Mono.just(s.toUpperCase()))).uri(uri)

.build();

}

## **115.28默认过滤器**

如果您要添加过滤器并将其应用于所有路由，则可以使用spring.cloud.gateway.default-filters。该属性采用过滤器列表

**application.yml。**

spring: cloud: gateway: default-filters: - AddResponseHeader=X-Response-Default-Foo**,** Default-Bar - PrefixPath=/httpbin

## **116.全局过滤器**

GlobalFilter接口具有与GatewayFilter相同的签名。这些是特殊过滤器，有条件地应用于所有路由。（此接口和用法可能会在将来的里程碑中更改）。

## **116.1组合的全局过滤器和GatewayFilter排序**

当有请求进入（并与路由匹配）时，过滤Web处理程序会将GlobalFilter的所有实例和GatewayFilter的所有特定于路由的实例添加到过滤器链中。该组合的过滤器链通过org.springframework.core.Ordered接口排序，可以通过实现getOrder()方法进行设置。

由于Spring Cloud网关区分执行过滤器逻辑的“前”阶段和“后”阶段（请参阅：[工作原理](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "gateway-how-it-works" \o "112.工作原理)），因此，具有最高优先级的过滤器将在“前”阶段中处于第一个阶段，在“阶段”中处于最后一个阶段。 “后期”阶段。

**ExampleConfiguration.java。**

@Bean**public** GlobalFilter customFilter() {

**return** **new** CustomGlobalFilter();

}

**public** **class** CustomGlobalFilter **implements** GlobalFilter, Ordered {

@Override

**public** Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

log.info("custom global filter");

**return** chain.filter(exchange);

}

@Override

**public** **int** getOrder() {

**return** -1;

}

}

## **116.2转发路由过滤器**

ForwardRoutingFilter在交换属性ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR中寻找URI。如果该网址具有forward方案（即forward:///localendpoint），它将使用Spring DispatcherHandler处理请求。请求URL的路径部分将被转发URL中的路径覆盖。未经修改的原始URL会附加到ServerWebExchangeUtils.GATEWAY\_ORIGINAL\_REQUEST\_URL\_ATTR属性中的列表中。

## **116.3 LoadBalancerClient筛选器**

LoadBalancerClientFilter在交换属性ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR中寻找URI。如果该网址具有lb方案（即lb://myservice），它将使用Spring Cloud LoadBalancerClient将名称（上例中为myservice）解析为实际的主机和端口并替换相同属性中的URI。未经修改的原始URL会附加到ServerWebExchangeUtils.GATEWAY\_ORIGINAL\_REQUEST\_URL\_ATTR属性中的列表中。过滤器还将查看ServerWebExchangeUtils.GATEWAY\_SCHEME\_PREFIX\_ATTR属性，以查看其是否等于lb，然后应用相同的规则。

**application.yml。**

spring: cloud: gateway: routes: - id: myRoute uri: lb://service predicates: - Path=/service/\*\*

|  |
| --- |
| IMG_625 |
| 默认情况下，当在LoadBalancer中找不到服务实例时，将返回503。您可以通过设置spring.cloud.gateway.loadbalancer.use404=true来配置网关以返回404。 |

|  |
| --- |
| IMG_626 |
| 从LoadBalancer返回的ServiceInstance的isSecure值将覆盖对网关的请求中指定的方案。例如，如果请求通过HTTPS进入网关，但ServiceInstance表示它不安全，则下游请求将通过HTTP发出。相反的情况也可以适用。但是，如果在网关配置中为路由指定了GATEWAY\_SCHEME\_PREFIX\_ATTR，则前缀将被删除，并且来自路由URL的结果方案将覆盖ServiceInstance配置。 |

|  |
| --- |
| IMG_627 |
| LoadBalancerClientFilter在引擎盖下使用阻挡物Ribbon LoadBalancerClient。我们建议您改用[ReactiveLoadBalancerClientFilter](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "reactive-loadbalancer-client-filter" \o "116.4 ReactiveLoadBalancerClientFilter)。您可以通过向项目添加org.springframework.cloud:spring-cloud-loadbalancer依赖项并将spring.cloud.loadbalancer.ribbon.enabled的值设置为false来切换为使用它。 |

## **116.4 ReactiveLoadBalancerClientFilter**

ReactiveLoadBalancerClientFilter在交换属性ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR中寻找URI。如果该网址具有lb方案（即lb://myservice），它将使用Spring Cloud ReactorLoadBalancer将名称（上例中为myservice）解析为实际的主机和端口并替换相同属性中的URI。未经修改的原始URL会附加到ServerWebExchangeUtils.GATEWAY\_ORIGINAL\_REQUEST\_URL\_ATTR属性中的列表中。过滤器还将查看ServerWebExchangeUtils.GATEWAY\_SCHEME\_PREFIX\_ATTR属性，以查看其是否等于lb，然后应用相同的规则。

**application.yml。**

spring: cloud: gateway: routes: - id: myRoute uri: lb://service predicates: - Path=/service/\*\*

|  |
| --- |
| IMG_628 |
| 默认情况下，当ReactorLoadBalancer无法找到服务实例时，将返回503。您可以通过设置spring.cloud.gateway.loadbalancer.use404=true将网关配置为返回404。 |

|  |
| --- |
| IMG_629 |
| 从ReactiveLoadBalancerClientFilter返回的ServiceInstance的isSecure值将覆盖对网关的请求中指定的方案。例如，如果请求通过HTTPS进入网关，但ServiceInstance表示它不安全，则下游请求将通过HTTP发出。相反的情况也可以适用。但是，如果在网关配置中为路由指定了GATEWAY\_SCHEME\_PREFIX\_ATTR，则前缀将被删除，并且从路由URL生成的方案将覆盖ServiceInstance配置。 |

## **116.5网络路由过滤器**

如果位于ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR交换属性中的URL具有http或https方案，则将运行Netty路由筛选器。它使用Netty HttpClient发出下游代理请求。响应将放在ServerWebExchangeUtils.CLIENT\_RESPONSE\_ATTR交换属性中，以供以后的过滤器使用。（有一个实验性WebClientHttpRoutingFilter，它执行相同的功能，但不需要净值）

## **116.6 Netty写响应过滤器**

如果ServerWebExchangeUtils.CLIENT\_RESPONSE\_ATTR交换属性中有净值HttpClientResponse，则NettyWriteResponseFilter将运行。它在所有其他筛选器完成后运行，并将代理响应写回到网关客户端响应。（有一个实验性WebClientWriteResponseFilter，它执行相同的功能，但不需要净值）

## **116.7 RouteToRequestUrl过滤器**

如果ServerWebExchangeUtils.GATEWAY\_ROUTE\_ATTR交换属性中存在Route对象，则RouteToRequestUrlFilter将运行。它基于请求URI创建一个新URI，但使用Route对象的URI属性进行了更新。新的URI放置在ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR交换属性中。

如果URI具有方案前缀（例如lb:ws://serviceid），则将从URI中剥离lb方案，并将其放在ServerWebExchangeUtils.GATEWAY\_SCHEME\_PREFIX\_ATTR中，以供以后在过滤器链中使用。

## **116.8 Websocket路由过滤器**

如果位于ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR交换属性中的URL具有ws或wss方案，则Websocket路由筛选器将运行。它使用Spring Web套接字基础结构向下游转发Websocket请求。

通过在URI前面加上lb，例如lb:ws://serviceid，可以实现Websocket的负载均衡。

|  |
| --- |
| IMG_630 |
| 如果您将[SockJS](https://github.com/sockjs" \t "https://www.springcloud.cc/_top)用作常规http的后​​备，则应配置常规HTTP路由以及Websocket路由。 |

**application.yml。**

spring: cloud: gateway: routes:

*# SockJS route* - id: websocket\_sockjs\_route uri: http://localhost:3001 predicates: - Path=/websocket/info/\*\*

*# Normal Websocket route* - id: websocket\_route uri: ws://localhost:3001 predicates: - Path=/websocket/\*\*

## **116.9网关指标过滤器**

要启用网关度量标准，请添加spring-boot-starter-actuator作为项目依赖项。然后，默认情况下，只要属性spring.cloud.gateway.metrics.enabled未设置为false，网关度量过滤器就会运行。该过滤器添加了一个带有以下标记的名为“ gateway.requests”的计时器指标：

* routeId：路线ID
* routeUri：API将被路由到的URI
* outcome：根据[HttpStatus.Series](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.Series.html" \t "https://www.springcloud.cc/_top)分类的结果
* status：返回给客户端的请求的Http状态
* httpStatusCode：返回给客户端的请求的Http状态
* httpMethod：用于请求的Http方法

然后可以从/actuator/metrics/gateway.requests [抓取](https://www.springcloud.cc/images/gateway-grafana-dashboard.jpeg" \t "https://www.springcloud.cc/_top) 这些指标，并且可以轻松地将其与Prometheus集成以创建[Grafana](https://www.springcloud.cc/images/gateway-grafana-dashboard.jpeg" \t "https://www.springcloud.cc/_top)[仪表板](https://www.springcloud.cc/gateway-grafana-dashboard.json" \t "https://www.springcloud.cc/_top)。

|  |
| --- |
| IMG_631 |
| 要启用Prometheus端点，请添加micrometer-registry-prometheus作为项目依赖项。 |

## **116.10将交换标记为已路由**

网关路由ServerWebExchange后，它将通过在交换属性中添加gatewayAlreadyRouted将该交换标记为“已路由”。将请求标记为已路由后，其他路由筛选器将不会再次路由请求，实质上会跳过该筛选器。您可以使用多种便捷方法将交换标记为已路由，或者检查交换是否已路由。

* ServerWebExchangeUtils.isAlreadyRouted接受ServerWebExchange对象，并检查其是否已“路由”
* ServerWebExchangeUtils.setAlreadyRouted接受ServerWebExchange对象，并将其标记为“已路由”

## **117. HttpHeadersFilters**

HttpHeadersFilters在向下游发送请求之前（例如，在NettyRoutingFilter中）已应用于请求。

## **117.1转发的标题过滤器**

Forwarded标头过滤器创建一个Forwarded标头，以发送到下游服务。它将当前请求的Host标头，方案和端口添加到任何现有的Forwarded标头中。

## **117.2 RemoveHopByHop标头过滤器**

RemoveHopByHop标头过滤器从转发的请求中删除标头。被删除的头的默认列表来自[IETF](https://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-14" \l "section-7.1.3" \t "https://www.springcloud.cc/_top)。

**默认删除的标题为：**

* 连接
* 活着
* 代理验证
* 代理授权
* TE
* 预告片
* 传输编码
* 升级

要更改此设置，请将spring.cloud.gateway.filter.remove-non-proxy-headers.headers属性设置为要删除的标头名称列表。

## **117.3 XForwarded标头过滤器**

XForwarded标头过滤器创建各种X-Forwarded-\*标头，以发送到下游服务。它使用Host头，当前请求的方案，端口和路径来创建各种头。

可以通过以下布尔属性（默认为true）控制单个标题的创建：

* spring.cloud.gateway.x-forwarded.for.enabled
* spring.cloud.gateway.x-forwarded.host.enabled
* spring.cloud.gateway.x-forwarded.port.enabled
* spring.cloud.gateway.x-forwarded.proto.enabled
* spring.cloud.gateway.x-forwarded.prefix.enabled

可以通过以下布尔属性（默认为true）控制追加多个标头：

* spring.cloud.gateway.x-forwarded.for.append
* spring.cloud.gateway.x-forwarded.host.append
* spring.cloud.gateway.x-forwarded.port.append
* spring.cloud.gateway.x-forwarded.proto.append
* spring.cloud.gateway.x-forwarded.prefix.append

## **118. TLS / SSL**

网关可以通过遵循常规的Spring服务器配置来侦听https上的请求。例：

**application.yml。**

server: ssl: enabled: **true** key-alias: scg key-store-password: scg1234 key-store: classpath:scg-keystore.p12 key-store-type: PKCS12

网关路由可以同时路由到http和https后端。如果路由到https后端，则可以使用以下配置将网关配置为信任所有下游证书：

**application.yml。**

spring: cloud: gateway: httpclient: ssl: useInsecureTrustManager: **true**

使用不安全的信任管理器不适用于生产。对于生产部署，可以为网关配置一组可以通过以下配置信任的已知证书：

**application.yml。**

spring: cloud: gateway: httpclient: ssl: trustedX509Certificates:

- cert1.pem

- cert2.pem

如果Spring Cloud网关未配置受信任的证书，则使用默认的信任存储（可以使用系统属性javax.net.ssl.trustStore覆盖）。

## **118.1 TLS握手**

网关维护一个客户端池，该客户端池用于路由到后端。通过https进行通信时，客户端会启动TLS握手。许多超时与此握手相关联。可以配置以下超时（显示默认值）：

**application.yml。**

spring: cloud: gateway: httpclient: ssl: handshake-timeout-millis: 10000 close-notify-flush-timeout-millis: 3000 close-notify-read-timeout-millis: 0

## **119.配置**

Spring Cloud网关的配置由RouteDefinitionLocator的集合驱动。

**RouteDefinitionLocator.java。**

**public** **interface** RouteDefinitionLocator {

Flux<RouteDefinition> getRouteDefinitions();

}

默认情况下，PropertiesRouteDefinitionLocator使用Spring Boot的@ConfigurationProperties机制加载属性。

上面的所有配置示例都使用一种快捷方式符号，该快捷方式符号使用位置参数而不是命名参数。以下两个示例是等效的：

**application.yml。**

spring: cloud: gateway: routes: - id: setstatus\_route uri: https://example.org filters: - name: SetStatus args: status: 401 - id: setstatusshortcut\_route uri: https://example.org filters: - SetStatus=401

对于网关的某些用法，属性将是足够的，但某些生产用例将受益于从外部源（例如数据库）加载配置。未来的里程碑版本将基于Spring Data Repositories实现RouteDefinitionLocator实现，例如：Redis，MongoDB和Cassandra。

## **119.1 Fluent Java Routes API**

为了在Java中进行简单的配置，在RouteLocatorBuilder bean中定义了一个流畅的API。

**GatewaySampleApplication.java。**

*// static imports from GatewayFilters and RoutePredicates*@Bean**public** RouteLocator customRouteLocator(RouteLocatorBuilder builder, ThrottleGatewayFilterFactory throttle) {

**return** builder.routes()

.route(r -> r.host("\*\*.abc.org").and().path("/image/png")

.filters(f ->

f.addResponseHeader("X-TestHeader", "foobar"))

.uri("http://httpbin.org:80")

)

.route(r -> r.path("/image/webp")

.filters(f ->

f.addResponseHeader("X-AnotherHeader", "baz"))

.uri("http://httpbin.org:80")

)

.route(r -> r.order(-1)

.host("\*\*.throttle.org").and().path("/get")

.filters(f -> f.filter(throttle.apply(1,

1,

10,

TimeUnit.SECONDS)))

.uri("http://httpbin.org:80")

)

.build();

}

此样式还允许更多自定义谓词断言。RouteDefinitionLocator beans定义的谓词使用逻辑and进行组合。通过使用流畅的Java API，您可以在Predicate类上使用and()，or()和negate()运算符。

## **119.2 DiscoveryClient路由定义定位器**

可以将网关配置为基于在DiscoveryClient兼容服务注册表中注册的服务来创建路由。

要启用此功能，请设置spring.cloud.gateway.discovery.locator.enabled=true并确保在类路径上启用了DiscoveryClient实现（例如Netflix Eureka，Consul或Zookeeper）。

### **119.2.1为DiscoveryClient路由配置谓词和过滤器**

默认情况下，网关为通过DiscoveryClient创建的路由定义单个谓词和过滤器。

默认谓词是使用模式/serviceId/\*\*定义的路径谓词，其中serviceId是DiscoveryClient中服务的ID。

缺省过滤器是带有正则表达式/serviceId/(?<remaining>.\*)和替换文本/${remaining}的重写路径过滤器。这只是在将请求发送到下游之前从路径中剥离服务ID。

如果要自定义DiscoveryClient路由使用的谓词和/或过滤器，可以通过设置spring.cloud.gateway.discovery.locator.predicates[x]和spring.cloud.gateway.discovery.locator.filters[y]来实现。这样做时，如果要保留该功能，则需要确保在上面包含默认谓词和过滤器。以下是此示例的示例。

**application.properties。**

spring.cloud.gateway.discovery.locator.predicates[0].name: Path

spring.cloud.gateway.discovery.locator.predicates[0].args[pattern]: "'/'+serviceId+'/\*\*'"

spring.cloud.gateway.discovery.locator.predicates[1].name: Host

spring.cloud.gateway.discovery.locator.predicates[1].args[pattern]: "'\*\*.foo.com'"

spring.cloud.gateway.discovery.locator.filters[0].name: Hystrix

spring.cloud.gateway.discovery.locator.filters[0].args[name]: serviceId

spring.cloud.gateway.discovery.locator.filters[1].name: RewritePath

spring.cloud.gateway.discovery.locator.filters[1].args[regexp]: "'/' + serviceId + '/(?<remaining>.\*)'"

spring.cloud.gateway.discovery.locator.filters[1].args[replacement]: "'/${remaining}'"

## **120. Reactor Netty访问日志**

要启用Reactor Netty访问日志，请设置-Dreactor.netty.http.server.accessLogEnabled=true。（它必须是Java系统Property，而不是Spring Boot属性）。

日志系统可以配置为具有单独的访问日志文件。以下是示例登录配置：

**logback.xml。**

<appender name="accessLog" class="ch.qos.logback.core.FileAppender">

<file>access\_log.log</file>

<encoder>

<pattern>%msg%n</pattern>

</encoder>

</appender>

<appender name="async" class="ch.qos.logback.classic.AsyncAppender">

<appender-ref ref="accessLog" />

</appender>

<logger name="reactor.netty.http.server.AccessLog" level="INFO" additivity="false">

<appender-ref ref="async"/>

</logger>

## **121. CORS配置**

可以将网关配置为控制CORS行为。“全局” CORS配置是URL模式到[Spring Framework CorsConfiguration](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/cors/CorsConfiguration.html" \t "https://www.springcloud.cc/_top)的映射。

**application.yml。**

spring: cloud: gateway: globalcors: corsConfigurations: '[/\*\*]': allowedOrigins: "https://docs.spring.io" allowedMethods:

- GET

在上面的示例中，对于所有GET请求的路径，来自docs.spring.io的请求都将允许CORS请求。

要为未被某些网关路由谓词处理的请求提供相同的CORS配置，请将属性spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping设置为true。当尝试支持CORS预检请求并且您的路由谓词未评估为true时，这很有用，因为http方法为options。

## **122.执行器API**

/gateway执行器端点允许监视Spring Cloud Gateway应用程序并与之交互。为了可远程访问，必须在应用程序属性中[通过HTTP或JMX](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html" \l "production-ready-endpoints-exposing-endpoints" \t "https://www.springcloud.cc/_top)[启用](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html" \l "production-ready-endpoints-enabling-endpoints" \t "https://www.springcloud.cc/_top)和[公开](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html" \l "production-ready-endpoints-exposing-endpoints" \t "https://www.springcloud.cc/_top)端点。

**application.properties。**

management.endpoint.gateway.enabled=true *# default value*management.endpoints.web.exposure.include=gateway

## **122.1详细执行器格式**

一种新的，更详细的格式已添加到网关。这为每个路由增加了更多细节，从而允许查看与每个路由关联的谓词和过滤器以及任何可用的配置。

/actuator/gateway/routes

**[**

**{**

"predicate": "(Hosts: [\*\*.addrequestheader.org] && Paths: [/headers], match trailing slash: true)"**,**

"route\_id": "add\_request\_header\_test"**,**

"filters": **[**

"[[AddResponseHeader X-Response-Default-Foo = 'Default-Bar'], order = 1]"**,**

"[[AddRequestHeader X-Request-Foo = 'Bar'], order = 1]"**,**

"[[PrefixPath prefix = '/httpbin'], order = 2]"

]**,**

"uri": "lb://testservice"**,**

"order": 0

**}]**

要启用此功能，请设置以下属性：

**application.properties。**

spring.cloud.gateway.actuator.verbose.enabled=true

在将来的版本中，该默认值为true。

## **122.2检索路由过滤器**

### **122.2.1全局过滤器**

要检索应用于所有路由的[全局过滤器](https://www.springcloud.cc/spring-cloud-greenwich.html)，请向/actuator/gateway/globalfilters发出GET请求。产生的响应类似于以下内容：

{

"org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5": 10100,

"org.springframework.cloud.gateway.filter.RouteToRequestUrlFilter@4f6fd101": 10000,

"org.springframework.cloud.gateway.filter.NettyWriteResponseFilter@32d22650": -1,

"org.springframework.cloud.gateway.filter.ForwardRoutingFilter@106459d9": 2147483647,

"org.springframework.cloud.gateway.filter.NettyRoutingFilter@1fbd5e0": 2147483647,

"org.springframework.cloud.gateway.filter.ForwardPathFilter@33a71d23": 0,

"org.springframework.cloud.gateway.filter.AdaptCachedBodyGlobalFilter@135064ea": 2147483637,

"org.springframework.cloud.gateway.filter.WebsocketRoutingFilter@23c05889": 2147483646

}

该响应包含适当的全局过滤器的详细信息。为每个全局过滤器提供过滤器对象的字符串表示形式（例如org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5）和过滤器链中的相应[顺序](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_combined_global_filter_and_gatewayfilter_ordering" \o "116.1组合的全局过滤器和GatewayFilter排序)。

### **122.2.2路由过滤器**

要检索应用于路由的[GatewayFilter工厂](https://www.springcloud.cc/spring-cloud-greenwich.html)，请向/actuator/gateway/routefilters发出GET请求。产生的响应类似于以下内容：

{

"[AddRequestHeaderGatewayFilterFactory@570ed9c configClass = AbstractNameValueGatewayFilterFactory.NameValueConfig]": null,

"[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]": null,

"[SaveSessionGatewayFilterFactory@4449b273 configClass = Object]": null

}

该响应包含应用于任何特定路由的GatewayFilter工厂的详细信息。为每个工厂提供相应对象的字符串表示形式（例如[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]）。请注意，null值是由于端点控制器的实现不完整而导致的，因为它试图设置对象在过滤器链中的顺序，该顺序不适用于GatewayFilter工厂对象。

## **122.3刷新路由缓存**

要清除路由缓存，请向/actuator/gateway/refresh发出POST请求。该请求返回200，但没有响应主体。

## **122.4检索网关中定义的路由**

要检索网关中定义的路由，请向/actuator/gateway/routes发出GET请求。产生的响应类似于以下内容：

[{

"route\_id": "first\_route",

"route\_object": {

"predicate": "org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$Lambda$432/1736826640@1e9d7e7d",

"filters": [

"OrderedGatewayFilter{delegate=org.springframework.cloud.gateway.filter.factory.PreserveHostHeaderGatewayFilterFactory$$Lambda$436/674480275@6631ef72, order=0}"

]

},

"order": 0

},

{

"route\_id": "second\_route",

"route\_object": {

"predicate": "org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$Lambda$432/1736826640@cd8d298",

"filters": []

},

"order": 0

}]

该响应包含网关中定义的所有路由的详细信息。下表描述了响应的每个元素（即路线）的结构。

| **路径** | **类型** | **描述** |
| --- | --- | --- |
| route\_id | String | The route id. |
| route\_object.predicate | Object | The route predicate. |
| route\_object.filters | Array | The [GatewayFilter factories](https://www.springcloud.cc/spring-cloud-greenwich.html) applied to the route. |
| order | Number | The route order. |

## **122.5检索有关特定路线的信息**

要检索有关一条路线的信息，请向/actuator/gateway/routes/{id}发送一个GET请求（例如/actuator/gateway/routes/first\_route）。产生的响应类似于以下内容：

{

"id": "first\_route",

"predicates": [{

"name": "Path",

"args": {"\_genkey\_0":"/first"}

}],

"filters": [],

"uri": "https://www.uri-destination.org",

"order": 0

}]

下表描述了响应的结构。

| **路径** | **类型** | **描述** |
| --- | --- | --- |
| id | String | The route id. |
| predicates | Array | The collection of route predicates. Each item defines the name and the arguments of a given predicate. |
| filters | Array | The collection of filters applied to the route. |
| uri | String | The destination URI of the route. |
| order | Number | The route order. |

## **122.6创建和删除特定路线**

要创建路由，请使用指定路由字段的JSON正文向/gateway/routes/{id\_route\_to\_create}发出POST请求（请参见上一小节）。

要删除路由，请向/gateway/routes/{id\_route\_to\_delete}发出DELETE请求。

## **122.7概述：所有端点的列表**

下表总结了Spring Cloud网关执行器端点。请注意，每个端点都有/actuator/gateway作为基本路径。

| **ID** | **HTTP方法** | **描述** |
| --- | --- | --- |
| globalfilters | GET | Displays the list of global filters applied to the routes. |
| routefilters | GET | Displays the list of GatewayFilter factories applied to a particular route. |
| refresh | POST | Clears the routes cache. |
| routes | GET | Displays the list of routes defined in the gateway. |
| routes/{id} | GET | Displays information about a particular route. |
| routes/{id} | POST | Add a new route to the gateway. |
| routes/{id} | DELETE | Remove an existing route from the gateway. |

## **123.故障排除**

## **123.1日志级别**

以下是一些有用的记录器，它们包含DEBUG和TRACE级别的有价值的故障排除信息。

* org.springframework.cloud.gateway
* org.springframework.http.server.reactive
* org.springframework.web.reactive
* org.springframework.boot.autoconfigure.web
* reactor.netty
* redisratelimiter

## **123.2窃听**

Reactor Netty HttpClient和HttpServer可以启用窃听功能。与将reactor.netty日志级别设置为DEBUG或TRACE结合使用时，将允许记录信息，例如通过网络发送和接收的标头和正文。要启用此功能，请分别为HttpServer和HttpClient设置spring.cloud.gateway.httpserver.wiretap=true和/或spring.cloud.gateway.httpclient.wiretap=true。

## **124.开发人员指南**

这些是编写网关的某些自定义组件的基本指南。

## **124.1编写自定义路由谓词工厂**

为了编写路由谓词，您将需要实现RoutePredicateFactory。您可以扩展名为AbstractRoutePredicateFactory的抽象类。

**MyRoutePredicateFactory.java。**

**public** **class** MyRoutePredicateFactory **extends** AbstractRoutePredicateFactory<HeaderRoutePredicateFactory.Config> {

**public** MyRoutePredicateFactory() {

**super**(Config.**class**);

}

@Override

**public** Predicate<ServerWebExchange> apply(Config config) {

*// grab configuration from Config object*

**return** exchange -> {

*//grab the request*

ServerHttpRequest request = exchange.getRequest();

*//take information from the request to see if it*

*//matches configuration.*

**return** matches(config, request);

};

}

**public** **static** **class** Config {

*//Put the configuration properties for your filter here*

}

}

## **124.2编写自定义GatewayFilter工厂**

为了编写GatewayFilter，您将需要实现GatewayFilterFactory。您可以扩展名为AbstractGatewayFilterFactory的抽象类。

**PreGatewayFilterFactory.java。**

**public** **class** PreGatewayFilterFactory **extends** AbstractGatewayFilterFactory<PreGatewayFilterFactory.Config> {

**public** PreGatewayFilterFactory() {

**super**(Config.**class**);

}

@Override

**public** GatewayFilter apply(Config config) {

*// grab configuration from Config object*

**return** (exchange, chain) -> {

*//If you want to build a "pre" filter you need to manipulate the*

*//request before calling chain.filter*

ServerHttpRequest.Builder builder = exchange.getRequest().mutate();

*//use builder to manipulate the request*

**return** chain.filter(exchange.mutate().request(request).build());

};

}

**public** **static** **class** Config {

*//Put the configuration properties for your filter here*

}

}

**PostGatewayFilterFactory.java。**

**public** **class** PostGatewayFilterFactory **extends** AbstractGatewayFilterFactory<PostGatewayFilterFactory.Config> {

**public** PostGatewayFilterFactory() {

**super**(Config.**class**);

}

@Override

**public** GatewayFilter apply(Config config) {

*// grab configuration from Config object*

**return** (exchange, chain) -> {

**return** chain.filter(exchange).then(Mono.fromRunnable(() -> {

ServerHttpResponse response = exchange.getResponse();

*//Manipulate the response in some way*

}));

};

}

**public** **static** **class** Config {

*//Put the configuration properties for your filter here*

}

}

## **124.3编写自定义全局过滤器**

为了编写自定义全局过滤器，您将需要实现GlobalFilter接口。这会将过滤器应用于所有请求。

如何分别设置全局前置和后置过滤器的示例

@Bean**public** GlobalFilter customGlobalFilter() {

**return** (exchange, chain) -> exchange.getPrincipal()

.map(Principal::getName)

.defaultIfEmpty("Default User")

.map(userName -> {

*//adds header to proxied request*

exchange.getRequest().mutate().header("CUSTOM-REQUEST-HEADER", userName).build();

**return** exchange;

})

.flatMap(chain::filter);

}

@Bean**public** GlobalFilter customGlobalPostFilter() {

**return** (exchange, chain) -> chain.filter(exchange)

.then(Mono.just(exchange))

.map(serverWebExchange -> {

*//adds header to response*

serverWebExchange.getResponse().getHeaders().set("CUSTOM-RESPONSE-HEADER",

HttpStatus.OK.equals(serverWebExchange.getResponse().getStatusCode()) ? "It worked": "It did not work");

**return** serverWebExchange;

})

.then();

}

## **125.使用Spring MVC或Webflux构建一个简单的网关**

|  |
| --- |
| IMG_632 |
| 以下描述了替代样式的网关。先前文档的None适用于以下内容。 |

Spring Cloud Gateway提供了一个名为ProxyExchange的实用程序对象，您可以在常规的Spring web处理程序中将其用作方法参数。它通过镜像HTTP动词的方法支持基本的下游HTTP交换。使用MVC，它还支持通过forward()方法转发到本地处理程序。要使用ProxyExchange，只需在类路径中包含正确的模块（spring-cloud-gateway-mvc或spring-cloud-gateway-webflux）。

MVC示例（代理对远程服务器下游“ /测试”的请求）：

@RestController@SpringBootApplication**public** **class** GatewaySampleApplication {

@Value("${remote.home}")

**private** URI home;

@GetMapping("/test")

**public** ResponseEntity<?> proxy(ProxyExchange<**byte**[]> proxy) **throws** Exception {

**return** proxy.uri(home.toString() + "/image/png").get();

}

}

与Webflux相同：

@RestController@SpringBootApplication**public** **class** GatewaySampleApplication {

@Value("${remote.home}")

**private** URI home;

@GetMapping("/test")

**public** Mono<ResponseEntity<?>> proxy(ProxyExchange<**byte**[]> proxy) **throws** Exception {

**return** proxy.uri(home.toString() + "/image/png").get();

}

}

ProxyExchange上有一些便利的方法可以使处理程序方法发现并增强传入请求的URI路径。例如，您可能希望提取路径的尾随元素以将它们传递到下游：

@GetMapping("/proxy/path/\*\*")**public** ResponseEntity<?> proxyPath(ProxyExchange<**byte**[]> proxy) **throws** Exception {

String path = proxy.path("/proxy/path/");

**return** proxy.uri(home.toString() + "/foos/" + path).get();

}

网关处理程序方法可以使用Spring MVC或Webflux的所有功能。因此，例如，您可以注入请求标头和查询参数，并且可以使用映射批注中的声明来约束传入的请求。有关这些功能的更多详细信息，请参见Spring MVC中的@RequestMapping文档。

可以使用ProxyExchange上的header()方法将标头添加到下游响应中。

您还可以通过将映射器添加到get()等方法来操纵响应头（以及响应中您喜欢的任何其他内容）。映射器是Function，它接收传入的ResponseEntity并将其转换为传出的ResponseEntity。

为不传递到下游的“敏感”标头（默认情况下为“ cookie”和“授权”）以及“代理”标头（x-forwarded-\*）提供了一流的支持。

# **第十六部分。Spring Cloud功能**

马克·费舍尔，戴夫·瑟尔，奥列格·朱拉库斯基

## **126.引言**

Spring Cloud功能是一个具有以下高级目标的项目：

* 通过功能促进业务逻辑的实现。
* 将业务逻辑的开发生命周期与任何特定的运行时目标脱钩，以便相同的代码可以作为web端点，流处理器或任务来运行。
* 支持跨无服务器提供程序的统一编程模型，以及独立运行（本地或在PaaS中）的能力。
* 在无服务器提供程序上启用Spring Boot功能（自动配置，依赖项注入，指标）。

它抽象出了所有传输细节和基础结构，使开发人员可以保留所有熟悉的工具和流程，并专注于业务逻辑。

这是一个完整的，可执行的，可测试的Spring Boot应用程序（实现简单的字符串操作）：

@SpringBootApplication**public** **class** Application {

@Bean

**public** Function<Flux<String>, Flux<String>> uppercase() {

**return** flux -> flux.map(value -> value.toUpperCase());

}

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

它只是一个Spring Boot应用程序，因此可以像其他任何Spring Boot应用程序一样在本地以CI生成，运行和测试它。Function来自java.util，而Flux是来自项目Reactor的反应性流Publisher。来自[项目Reactor](https://projectreactor.io/" \t "https://www.springcloud.cc/_top)的 [反应性流](https://www.reactive-streams.org/" \t "https://www.springcloud.cc/_top) {5297 [/}](https://projectreactor.io/" \t "https://www.springcloud.cc/_top)。可以通过HTTP或消息传递来访问该功能。

Spring Cloud功能具有4个主要功能：

1. 类型为Function，Consumer和Supplier的@Beans的包装程序，将它们作为HTTP端点和/或消息流侦听器/发布程序使用RabbitMQ，Kafka等
2. 将作为Java函数体的字符串编译为字节码，然后将其转换为@Beans，可以像上面那样进行包装。
3. 使用隔离的类加载器部署包含此类应用程序上下文的JAR文件，以便可以将它们打包在一起在单个JVM中。
4. 适用于[AWS Lambda](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-aws" \t "https://www.springcloud.cc/_top)，[Azure](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-azure" \t "https://www.springcloud.cc/_top)，[Apache OpenWhisk](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk" \t "https://www.springcloud.cc/_top)以及其他“无服务器”服务提供商的适配器。

|  |
| --- |
| IMG_633 |
| Spring Cloud是根据非限制性Apache 2.0许可证发行的。如果您想为文档的这一部分做出贡献或发现错误，请在[github](https://github.com/spring-cloud/spring-cloud-function/tree/master/docs/src/main/asciidoc" \t "https://www.springcloud.cc/_top)的项目中找到源代码和问题跟踪程序。 |

## **127.入门**

从命令行构建（并“安装”示例）：

$ ./mvnw clean install

（如果您想使用YOLO，请添加-DskipTests。）

运行其中一个示例，例如

$ java -jar spring-cloud-function-samples/function-sample/target/\*.jar

这将运行该应用程序并通过HTTP公开其功能，因此您可以将字符串转换为大写，如下所示：

$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d Hello

HELLO

您可以通过用新行分隔多个字符串（Flux<String>）来进行转换

$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'Hello

> World'

HELLOWORLD

（您可以在终端中使用QJ在这样的文字字符串中插入新行。）

## **128.构建和运行功能**

上面的示例@SpringBootApplication具有可以在运行时由Spring Cloud函数修饰为HTTP端点或流处理器（例如，使用RabbitMQ，Apache Kafka或JMS）的功能。

@Beans可以是Function，Consumer或Supplier（均来自java.util），其参数类型可以是String或POJO。

函数也可以是Flux<String>或Flux<Pojo>和Spring的云函数，它负责将数据与所需类型之间来回转换，只要它们以纯文本格式出现（或POJO）JSON。还支持Message<Pojo>，在此消息头是从传入事件复制而来的，具体取决于适配器。web适配器还支持从表单编码数据到Map的转换，如果您将函数与Spring Cloud Stream一起使用，则消息有效负载的所有转换和强制功能也将适用。

可以将功能组合在单个应用程序中，也可以每个jar部署一个。由开发人员选择。具有多种功能的应用程序可以以不同的“个性”多次部署，从而在不同的物理传输方式上暴露出不同的功能。

## **129.功能目录和灵活的功能签名**

Spring Cloud函数的主要功能之一是为用户定义的函数适应和支持一系列类型签名，同时提供一致的执行模型。这就是为什么使用[项目Reactor](https://projectreactor.io/" \t "https://www.springcloud.cc/_top)（即Flux<T>和Mono<T>）定义的原语，FunctionCatalog将所有用户定义函数转换为规范表示的原因。例如，用户可以提供类型为Function<String,String>的bean，而FunctionCatalog会将其包装到Function<Flux<String>,Flux<String>>中。

使用基于Reactor的原语不仅有助于用户定义函数的规范表示，而且还有助于建立更健壮和灵活的（反应式）执行模型。

尽管用户通常根本不需要关心FunctionCatalog，但是了解用户代码支持哪些功能很有用。

## **129.1 Java 8功能支持**

一般而言，用户可以期望，如果他们为普通的旧Java类型（或原始包装器）编写函数，则函数目录会将其包装为相同类型的Flux。如果用户使用Message（通过spring-messaging）编写函数，它将从支持键值元数据的任何适配器接收和传输头（例如HTTP头）。这是详细信息。

| **用户功能** | **目录注册** |  |
| --- | --- | --- |
| Function<S,T> | Function<Flux<S>, Flux<T>> |  |
| Function<Message<S>,Message<T>> | Function<Flux<Message<S>>, Flux<Message<T>>> |  |
| Function<Flux<S>, Flux<T>> | Function<Flux<S>, Flux<T>> (pass through) |  |
| Supplier<T> | Supplier<Flux<T>> |  |
| Supplier<Flux<T>> | Supplier<Flux<T>> |  |
| Consumer<T> | Function<Flux<T>, Mono<Void>> |  |
| Consumer<Message<T>> | Function<Flux<Message<T>>, Mono<Void>> |  |
| Consumer<Flux<T>> | Consumer<Flux<T>> |  |

消费者有点特殊，因为它有一个void返回类型，这意味着至少有可能阻塞。很可能您不需要编写Consumer<Flux<?>>，但是如果需要这样做，请记住订阅输入流量。如果声明了非发布者类型的Consumer（正常），它将被转换为返回发布者的函数，以便可以通过受控方式进行订阅。

## **129.2 Kotlin Lambda支持**

我们还为Kotlin lambdas（自v2.0起）提供支持。考虑以下：

@Bean

open fun kotlinSupplier(): () -> String {

**return** { "Hello from Kotlin" }

}

@Bean

open fun kotlinFunction(): (String) -> String {

**return** { it.toUpperCase() }

}

@Bean

open fun kotlinConsumer(): (String) -> Unit {

**return** { println(it) }

}

上面的内容代表配置为Spring beans的Kotlin lambda。每个签名都映射到Java等效的Supplier，Function和Consumer，因此框架支持/识别了签名。尽管Kotlin到Java的映射机制不在本文档的讨论范围之内，但重要的是要理解，此处也适用“ Java 8函数支持”部分中概述的相同的签名转换规则。

要启用Kotlin支持，您需要在类路径中添加spring-cloud-function-kotlin模块，其中包含适当的自动配置和支持类。

## **130.独立的Web应用程序**

spring-cloud-function-web模块具有自动配置，当其包含在Spring Boot web应用程序中（具有MVC支持）时，将激活该配置。还有一个spring-cloud-starter-function-web来收集所有可选的依赖项，以防您只需要简单的入门经验。

激活web配置后，您的应用程序将具有一个MVC端点（默认情况下在“ /”上，但可以使用spring.cloud.function.web.path进行配置），该端点可用于访问应用程序上下文中的功能。支持的内容类型是纯文本和JSON。

| **方法** | **路径** | **请求** | **响应** | **状态** |
| --- | --- | --- | --- | --- |
| GET | /{supplier} | - | Items from the named supplier | 200 OK |
| POST | /{consumer} | JSON object or text | Mirrors input and pushes request body into consumer | 202 Accepted |
| POST | /{consumer} | JSON array or text with new lines | Mirrors input and pushes body into consumer one by one | 202 Accepted |
| POST | /{function} | JSON object or text | The result of applying the named function | 200 OK |
| POST | /{function} | JSON array or text with new lines | The result of applying the named function | 200 OK |
| GET | /{function}/{item} | - | Convert the item into an object and return the result of applying the function | 200 OK |

如上表所示，端点的行为取决于方法以及传入请求数据的类型。当传入的数据是单值的并且目标函数被声明为显然是单值的（即不返回集合或Flux）时，响应也将包含一个单值。对于多值响应，客户端可以通过发送“接受：文本/事件流”来请求服务器发送的事件流。

如果目录中只有一个功能（消费者等），则路径中的名称是可选的。可以使用管道或逗号分隔功能名称来解决复合函数（管道在URL路径中是合法的，但在命令行上键入会有点尴尬）。

如果目录中只有一个功能，而您又想将一个特定功能映射到根路径（例如“ /”），或者想要组合多个功能然后映射到根路径，则可以这样做通过提供spring.cloud.function.definition属性，该属性实际上由spring- = cloud-function- web模块使用，以为存在某种类型的冲突（例如，多个功能可用等）的情况提供默认映射。

例如，

--spring.cloud.function.definition=foo|bar

上面的属性将组成'foo'和'bar'函数，并将组成的函数映射到“ /”路径。

在Message<?>中用输入和输出声明的函数和使用者将在输入消息上看到请求标头，并且输出消息标头将转换为HTTP标头。

在发布文本时，Spring Boot 2.0和更早版本的响应格式可能会有所不同，具体取决于内容协商（提供内容类型和acpt标头以获得最佳效果）。

## **131.独立流应用程序**

要发送或接收来自代理（例如RabbitMQ或Kafka）的消息，您可以利用spring-cloud-stream项目并将其与Spring Cloud功能集成。有关更多详细信息和示例，请参阅Spring Cloud Stream参考手册的[Spring Cloud功能](https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/" \l "_spring_cloud_function" \t "https://www.springcloud.cc/_top)部分。

## **132.部署打包功能**

Spring Cloud函数提供了一个“部署程序”库，通过该库，您可以使用隔离的类加载器启动jar文件（或爆炸档案或jar文件集），并公开其中定义的函数。这是一个非常强大的工具，例如，您可以在不更改目标jar文件的情况下，使功能适应各种不同的输入输出适配器。无服务器平台通常具有内置的这种功能，因此您可以将其视为此类平台中函数调用程序的构建块（实际上，[Riff](https://projectriff.io/" \t "https://www.springcloud.cc/_top) Java函数调用程序使用此库）。

API的标准入口点是Spring配置注释@EnableFunctionDeployer。如果在Spring Boot应用程序中使用了该功能，则部署程序将启动并寻找某种配置以告知其在何处找到功能jar。至少，用户必须提供function.location，它是包含功能的存档的URL或资源位置。它可以选择使用maven:前缀通过依赖关系查找来定位工件（有关完整详细信息，请参见FunctionProperties）。从jar文件引导Spring Boot应用程序，并使用MANIFEST.MF查找起始类，例如，使标准Spring Boot胖子jar可以很好地工作。如果目标jar可以成功启动，则结果是在主应用程序的FunctionCatalog中注册了一个函数。已注册的函数可以通过主应用程序中的代码来应用，即使它是在隔离的类加载器中创建的（通过deault实现）。

## **133.功能性Bean定义**

对于需要快速启动的小型应用程序，Spring Cloud函数支持bean声明的“函数式”样式。bean声明的功能样式是Spring Framework 5.0的一项功能，在5.1中进行了重大改进。

## **133.1将功能与传统的Bean定义进行比较**

这是一种普通的Spring Cloud函数应用程序，具有相似的@Configuration和@Bean声明样式：

@SpringBootApplication**public** **class** DemoApplication {

@Bean

**public** Function<String, String> uppercase() {

**return** value -> value.toUpperCase();

}

**public** **static** **void** main(String[] args) {

SpringApplication.run(DemoApplication.**class**, args);

}

}

您可以在无服务器平台（如AWS Lambda或Azure Functions）中运行以上命令，也可以仅在类路径中包含spring-cloud-function-starter-web，即可在其自己的HTTP服务器中运行上述命令。运行main方法将公开一个端点，您可以使用该端点ping uppercase函数：

$ curl localhost:8080 -d foo

FOO

spring-cloud-function-starter-web中的web适配器使用Spring MVC，因此您需要一个Servlet容器。您也可以在默认服务器为netty的地方使用Webflux（即使您仍然愿意使用Servlet容器），也可以使用spring-cloud-starter-function-webflux依赖项。功能相同，并且两者都可以使用用户应用程序代码。

现在，对于功能beans：用户应用程序代码可以重铸为“功能”形式，如下所示：

@SpringBootConfiguration**public** **class** DemoApplication **implements** ApplicationContextInitializer<GenericApplicationContext> {

**public** **static** **void** main(String[] args) {

FunctionalSpringApplication.run(DemoApplication.**class**, args);

}

**public** Function<String, String> uppercase() {

**return** value -> value.toUpperCase();

}

@Override

**public** **void** initialize(GenericApplicationContext context) {

context.registerBean("demo", FunctionRegistration.**class**,

() -> **new** FunctionRegistration<>(uppercase())

.type(FunctionType.from(String.**class**).to(String.**class**)));

}

}

主要区别在于：

* 主要类是ApplicationContextInitializer。
* @Bean方法已转换为对context.registerBean()的调用
* @SpringBootApplication已替换为@SpringBootConfiguration，以表示我们未启用Spring引导自动配置，但仍将该类标记为“入口点”。
* Spring Boot中的SpringApplication已被Spring Cloud函数中的FunctionalSpringApplication取代（它是一个子类）。

您在Spring Cloud Function应用程序中注册的业务逻辑beans的类型为FunctionRegistration。这是一个包装，其中包含函数以及有关输入和输出类型的信息。在本应用程序的@Bean形式中，信息可以反射性地导出，但是在功能性bean注册中，除非我们使用FunctionRegistration，否则其中的一些信息会丢失。

使用ApplicationContextInitializer和FunctionRegistration的替代方法是使应用程序本身实现Function（或Consumer或Supplier）。示例（与上述等效）：

@SpringBootConfiguration**public** **class** DemoApplication **implements** Function<String, String> {

**public** **static** **void** main(String[] args) {

FunctionalSpringApplication.run(DemoApplication.**class**, args);

}

@Override

**public** String uppercase(String value) {

**return** value.toUpperCase();

}

}

如果您添加类型为Function的独立类，并使用run()方法的另一种形式向SpringApplication注册，它也将起作用。最主要的是，泛型类型信息可在运行时通过类声明获得。

如果您添加spring-cloud-starter-function-webflux，则该应用程序将在其自己的HTTP服务器上运行（由于尚未实现嵌入式Servlet容器的功能形式，因此它目前无法与MVC启动器一起使用）。该应用程序还可以在AWS Lambda或Azure Functions中正常运行，并且启动时间的改善是巨大的。

|  |
| --- |
| IMG_634 |
| “精简型” web服务器对Function签名的范围有一些限制-特别是它（目前）还不支持Message输入和输出，但是POJO和任何类型的Publisher应该没事。 |

## **133.2测试功能应用程序**

Spring Cloud函数还具有一些集成测试实用程序，这些实用程序对于Spring Boot用户而言非常熟悉。例如，这是包装以上应用程序的HTTP服务器的集成测试：

@RunWith(SpringRunner.class)@FunctionalSpringBootTest@AutoConfigureWebTestClient**public** **class** FunctionalTests {

@Autowired

**private** WebTestClient client;

@Test

**public** **void** words() **throws** Exception {

client.post().uri("/").body(Mono.just("foo"), String.**class**).exchange()

.expectStatus().isOk().expectBody(String.**class**).isEqualTo("FOO");

}

}

该测试几乎与您为同一应用程序的@Bean版本编写的测试相同-唯一的区别是@FunctionalSpringBootTest注释，而不是常规的@SpringBootTest。所有其他部件，例如@Autowired WebTestClient，都是标准的Spring Boot功能。

或者，您可以仅使用FunctionCatalog为非HTTP应用编写测试。例如：

@RunWith(SpringRunner.class)@FunctionalSpringBootTest**public** **class** FunctionalTests {

@Autowired

**private** FunctionCatalog catalog;

@Test

**public** **void** words() **throws** Exception {

Function<Flux<String>, Flux<String>> function = catalog.lookup(Function.**class**,

"function");

assertThat(function.apply(Flux.just("foo")).blockFirst()).isEqualTo("FOO");

}

}

（FunctionCatalog始终将函数从Flux返回到Flux，即使用户使用更简单的签名声明它们也是如此。）

## **133.3功能性Bean声明的局限性**

与整个Spring Boot相比，大多数Spring Cloud Function应用程序的范围相对较小，因此我们能够轻松地使其适应这些功能bean的定义。如果您超出了有限的范围，则可以通过切换回@Bean样式配置或使用混合方法来扩展Spring Cloud Function应用。例如，如果您想利用Spring Boot自动配置来与外部数据存储区集成，则需要使用@EnableAutoConfiguration。如果需要，仍可以使用函数声明来定义函数（即“混合”样式），但是在那种情况下，您将需要使用spring.functional.enabled=false显式关闭“全功能模式”，以便Spring Boot可以收回控制权。

## **134.动态编译**

有一个示例应用程序，该应用程序使用函数编译器从配置属性中创建函数。原始的“功能示例”也具有该功能。您可以运行一些脚本来查看编译在运行时发生的情况。要运行这些示例，请切换到scripts目录：

cd scripts

另外，在本地启动RabbitMQ服务器（例如，执行rabbitmq-server）。

启动功能注册表服务：

./function-registry.sh

注册功能：

./registerFunction.sh -n uppercase -f "f->f.map(s->s.toString().toUpperCase())"

使用该功能运行REST微服务：

./web.sh -f uppercase -p 9000

curl -H "Content-Type: text/plain" -H "Accept: text/plain" localhost:9000/uppercase -d foo

注册供应商：

./registerSupplier.sh -n words -f "()->Flux.just(\"foo\",\"bar\")"

使用该供应商运行REST微服务：

./web.sh -s words -p 9001

curl -H "Accept: application/json" localhost:9001/words

注册消费者：

./registerConsumer.sh -n print -t String -f "System.out::println"

使用该使用者运行REST微服务：

./web.sh -c print -p 9002

curl -X POST -H "Content-Type: text/plain" -d foo localhost:9002/print

运行流处理微服务：

首先注册流字供应商：

./registerSupplier.sh -n wordstream -f "()->Flux.interval(Duration.ofMillis(1000)).map(i->\"message-\"+i)"

然后启动源（供应商），处理器（功能）和宿（消费者）应用程序（以相反的顺序）：

./stream.sh -p 9103 -i uppercaseWords -c print

./stream.sh -p 9102 -i words -f uppercase -o uppercaseWords

./stream.sh -p 9101 -s wordstream -o words

输出将显示在接收器应用程序的控制台中（每秒一条消息，转换为大写字母）：

MESSAGE-0

MESSAGE-1

MESSAGE-2

MESSAGE-3

MESSAGE-4

MESSAGE-5

MESSAGE-6

MESSAGE-7

MESSAGE-8

MESSAGE-9

...

## **135.无服务器平台适配器**

除了能够作为独立进程运行之外，Spring Cloud Function应用程序还可以运行现有的无服务器平台之一。在项目中，有适用于 [AWS Lambda](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-aws" \t "https://www.springcloud.cc/_top)， [Azure](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-azure" \t "https://www.springcloud.cc/_top)和 [Apache OpenWhisk的适配器](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk" \t "https://www.springcloud.cc/_top)。在[甲骨文FN平台](https://github.com/fnproject/fn" \t "https://www.springcloud.cc/_top) 都有自己的Spring Cloud功能适配器。和 [里夫](https://projectriff.io/" \t "https://www.springcloud.cc/_top)支持Java的功能和它的 [Java函数调用器](https://github.com/projectriff/java-function-invoker" \t "https://www.springcloud.cc/_top)作用本身为Spring Cloud功能罐子的适配器。

## **135.1 AWS Lambda**

的[AWS](https://aws.amazon.com/" \t "https://www.springcloud.cc/_top)适配器取Spring Cloud功能的应用程序，并将其转换为可以在AWS LAMBDA运行的形式。

### **135.1.1简介**

适配器具有几个可以使用的通用请求处理程序。最通用的是SpringBootStreamHandler，它使用Spring Boot提供的Jackson ObjectMapper对函数中的对象进行序列化和反序列化。您还可以扩展SpringBootRequestHandler并将其输入和输出类型作为类型参数（使AWS能够检查类并自己进行JSON转换）。

如果您的应用程序具有多个Function等类型的@Bean等，则可以通过配置function.name（例如，在AWS中作为FUNCTION\_NAME环境变量）来选择要使用的一个。从Spring Cloud FunctionCatalog中提取函数（首先搜索Function，然后搜索Consumer，最后搜索Supplier）。

### **135.1.2有关JAR布局的注意事项**

Lambda在运行时不需要Spring Cloud函数Web或流适配器，因此在创建发送到AWS的JAR之前，可能需要排除它们。Lambda应用程序必须着色，而Spring Boot独立应用程序则不必着色，因此您可以使用2个单独的jar（根据示例）运行同一应用程序。该示例应用程序将创建2个jar文件，其中一个带有aws分类器以在Lambda中进行部署，而一个可执行（瘦）jar在运行时包括spring-cloud-function-web。Spring Cloud函数将使用Start-Class属性（如果使用入门级父级，将由Spring Boot工具为您添加），从JAR文件清单中尝试为您找到“主类”。 。如果清单中没有Start-Class，则在将功能部署到AWS时可以使用环境变量MAIN\_CLASS。

### **135.1.3上传**

在spring-cloud-function-samples/function-sample-aws下构建示例，并将-aws jar文件上传到Lambda。处理程序可以为example.Handler或org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler（该类的FQN，而不是方法引用，尽管Lambda确实接受方法引用）。

./mvnw -U clean package

使用AWS命令行工具，如下所示：

aws lambda create-function --function-name Uppercase --role arn:aws:iam::[USERID]:role/service-role/[ROLE] --zip-file fileb://function-sample-aws/target/function-sample-aws-2.0.0.BUILD-SNAPSHOT-aws.jar --handler org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler --description "Spring Cloud Function Adapter Example" --runtime java8 --region us-east-1 --timeout 30 --memory-size 1024 --publish

AWS示例中函数的输入类型是Foo，它具有一个称为“ value”的单个属性。因此，您需要使用它进行测试：

{

"value": "test"

}

|  |
| --- |
| IMG_635 |
| AWS示例应用程序以“功能性”风格编写（作为ApplicationContextInitializer）。在Lambda中启动时，这比传统的@Bean风格要快得多，因此，如果不需要@Beans（或@EnableAutoConfiguration），这是一个不错的选择。暖启动不受影响。 |

### **135.1.4 Platfom的特定功能**

#### **HTTP和API网关**

AWS具有某些特定于平台的数据类型，包括消息批处理，这比单独处理每个数据集要高效得多。要使用这些类型，您可以编写依赖于这些类型的函数。或者，您可以依靠Spring从AWS类型中提取数据并将其转换为Spring Message。为此，您要告诉AWS函数具有特定的通用处理程序类型（取决于AWS服务），并提供类型为Function<Message<S>,Message<T>>的bean，其中S和T是您的业务数据类型。如果类型Function的bean不止一个，则可能还需要将Spring Boot属性function.name配置为目标bean的名称（例如，使用FUNCTION\_NAME作为环境变量）。

支持的AWS服务和通用处理程序类型如下所示：

| **服务** | **AWS类型** | **通用处理程序** |  |
| --- | --- | --- | --- |
| API Gateway | APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent | org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler |  |
| Kinesis | KinesisEvent | org.springframework.cloud.function.adapter.aws.SpringBootKinesisEventHandler |  |

例如，要在API网关后面进行部署，请在您的AWS命令行中使用--handler org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler（通过UI）并定义类型为Function<Message<Foo>,Message<Bar>>的@Bean，其中Foo和Bar是POJO类型（数据将由AWS使用Jackson进行编组和解组）。

## **135.2 Azure功能**

所述[天青](https://azure.microsoft.com/" \t "https://www.springcloud.cc/_top)适配器自举一个Spring Cloud功能上下文和通道功能从天青框架呼叫到用户的功能，使用Spring Boot结构，其中必要的。Azure Functions具有一个非常独特但具有侵入性的编程模型，涉及特定于平台的用户代码中的注释。与Spring Cloud一起使用的最简单方法是扩展基类，并在其中编写带有@FunctionName批注的方法，该批注委派给基类方法。

该项目为Spring Cloud Function应用程序提供了到Azure的适配器层。您可以编写一个类型为Function的单个@Bean的应用程序，如果正确放置了JAR文件，则可以将其部署在Azure中。

必须扩展AzureSpringBootRequestHandler，并提供输入和输出类型作为带注释的方法参数（使Azure能够检查类并创建JSON绑定）。基类有两个有用的方法（handleRequest和handleOutput），您可以将实际的函数调用委派给该方法，因此大多数情况下，该函数只会有一行。

例：

**public** **class** FooHandler **extends** AzureSpringBootRequestHandler<Foo, Bar> {

@FunctionName("uppercase")

**public** Bar execute(

@HttpTrigger(name = "req", methods = { HttpMethod.GET,

HttpMethod.POST }, authLevel = AuthorizationLevel.ANONYMOUS)

Foo foo,

ExecutionContext context) {

**return** handleRequest(foo, context);

}

}

此Azure处理程序将委派给Function<Foo,Bar> bean（或Function<Publisher<Foo>,Publisher<Bar>>）。某些Azure触发器（例如@CosmosDBTrigger）会导致输入类型为List，在这种情况下，您可以绑定到Azure处理程序中的List或String（原始JSON）。List输入委托给输入类型为Map<String,Object>或Publisher或相同类型的List的Function。Function的输出可以是List（一对一）或单个值（聚合），并且Azure声明中的输出绑定应该匹配。

如果您的应用具有多个Function等类型的@Bean等，那么您可以通过配置function.name选择一个。或者，如果使Azure处理程序方法中的@FunctionName与函数名称匹配，则它应以这种方式工作（也适用于具有多个函数的函数应用程序）。这些功能是从Spring Cloud FunctionCatalog中提取的，因此默认功能名称与bean名称相同。

### **135.2.1有关JAR布局的注意事项**

在Azure的运行时中不需要Spring Cloud函数Web，因此可以在创建要部署到Azure的JAR之前将其排除在外，但是如果包含它，则不会使用它，因此不需要。保留它不会很麻烦。Azure上的功能应用程序是由Maven插件生成的存档。该函数位于此项目生成的JAR文件中。该示例使用精简版式将其创建为可执行jar，以便Azure可以找到处理程序类。如果您愿意，可以只使用常规的平面JAR文件。依赖性**不**应该包括在内。

### **135.2.2建立**

./mvnw -U clean package

### **135.2.3运行示例**

您可以在本地运行该示例，就像其他Spring Cloud函数示例一样：

和curl -H "Content-Type: text/plain" localhost:8080/function -d '{"value": "hello foobar"}'。

您将需要az CLI应用程序（有关更多详细信息，请参见[https://docs.microsoft.com/zh-cn/azure/azure-functions/functions-create-first-java-maven](https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven" \t "https://www.springcloud.cc/_top)）。要将功能部署在Azure运行时上：

$ az login

$ mvn azure-functions:deploy

在另一个终端上，尝试以下操作：curl [https://<azure-function-url-from-the-log>/api/uppercase](https://<azure-function-url-from-the-log>/api/uppercase" \t "https://www.springcloud.cc/_top) -d '{"value": "hello foobar!"}'。请确保为上述功能使用正确的URL。或者，您可以在Azure仪表板UI中测试该功能（单击功能名称，转到右侧，然后单击“测试”，然后单击右下角的“运行”）。

Azure示例中函数的输入类型是具有单个属性“ Foo”的Foo。因此，您需要使用以下代码进行测试：

{

"value": "foobar"

}

|  |
| --- |
| IMG_636 |
| Azure示例应用程序以“非功能性”样式编写（使用@Bean）。在Azure中启动时，功能样式（仅带有Function或ApplicationContextInitializer）比传统的@Bean样式要快得多，因此，如果不需要@Beans（或@EnableAutoConfiguration），这是一个不错的选择。暖启动不受影响。 |

## **135.3 Apache Openwhisk**

所述[OpenWhisk](https://openwhisk.apache.org/" \t "https://www.springcloud.cc/_top)适配器是在可以在AA搬运工图像被用于部署到Openwhisk一个可执行的jar的形式。该平台以请求-响应模式工作，侦听特定端点上的端口8080，因此适配器是一个简单的Spring MVC应用程序。

### **135.3.1快速入门**

实施POF（确保使用functions软件包）：

package functions;

import java.util.function.Function;

public class Uppercase implements Function<String, String> {

public String apply(String input) {

return input.toUpperCase();

}

}

将其安装到本地Maven存储库中：

./mvnw clean install

创建一个提供其Maven坐标的function.properties文件。例如：

dependencies.function: com.example:pof:0.0.1-SNAPSHOT

将openwhisk运行程序JAR复制到工作目录（与属性文件相同的目录）：

cp spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk/target/spring-cloud-function-adapter-openwhisk-2.0.0.BUILD-SNAPSHOT.jar runner.jar

使用上述属性文件从运行器JAR的--thin.dryrun生成一个m2回购：

java -jar -Dthin.root=m2 runner.jar --thin.name=function --thin.dryrun

使用以下Dockerfile：

FROM openjdk:8-jdk-alpine

VOLUME /tmp

COPY m2 /m2

ADD runner.jar .

ADD function.properties .

ENV JAVA\_OPTS=""

ENTRYPOINT [ "java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "runner.jar", "--thin.root=/m2", "--thin.name=function", "--function.name=uppercase"]

EXPOSE 8080

|  |
| --- |
| IMG_637 |
| 您可以使用Spring Cloud Function应用程序，而不是仅使用带有POF的jar，在这种情况下，您必须更改应用程序在容器中的运行方式，以便它将主类用作源文件。例如，您可以更改上面的ENTRYPOINT并添加--spring.main.sources=com.example.SampleApplication。 |

构建Docker映像：

docker build -t [username/appname] .

推送Docker映像：

docker push [username/appname]

使用OpenWhisk CLI（例如，在vagrant ssh之后）创建操作：

wsk action create example --docker [username/appname]

调用动作：

wsk action invoke example --result --param payload foo

{

"result": "FOO"

}

# **第十七部分。Spring Cloud Kubernetes**

本参考指南介绍了如何使用Spring Cloud Kubernetes。

## **136.为什么需要Spring Cloud Kubernetes？**

Spring Cloud Kubernetes提供了使用Kubernetes本机服务的Spring Cloud通用接口实现。该存储库中提供的项目的主要目的是促进Kubernetes中运行的Spring Cloud和Spring Boot应用程序的集成。

## **137. Starters**

Starters是方便的依赖项描述符，您可以在应用程序中包含它们。包括启动器以获取依赖关系和功能集的Spring Boot自动配置。

| **起动机** | **特征** |
| --- | --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-kubernetes</artifactId></dependency> | [Discovery Client](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_discoveryclient_for_kubernetes" \o "138. DiscoveryClient for Kubernetes) implementation that resolves service names to Kubernetes Services. |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-kubernetes-config</artifactId></dependency> | Load application properties from Kubernetes [ConfigMaps](https://www.springcloud.cc/spring-cloud-greenwich.html) and [Secrets](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "_secrets_propertysource" \o "140.2 Secrets PropertySource). [Reload](https://www.springcloud.cc/spring-cloud-greenwich.html) application properties when a ConfigMap or Secret changes. |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-kubernetes-ribbon</artifactId></dependency> | [Ribbon](https://www.springcloud.cc/spring-cloud-greenwich.html) client-side load balancer with server list obtained from Kubernetes Endpoints. |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-kubernetes-all</artifactId></dependency> | All Spring Cloud Kubernetes features. |

## **138. Kubernetes的DiscoveryClient**

该项目提供了[Kubernetes](https://kubernetes.io/" \t "https://www.springcloud.cc/_top)的[Discovery Client](https://github.com/spring-cloud/spring-cloud-commons/blob/master/spring-cloud-commons/src/main/java/org/springframework/cloud/client/discovery/DiscoveryClient.java" \t "https://www.springcloud.cc/_top) 的[实现](https://kubernetes.io/" \t "https://www.springcloud.cc/_top)。通过此客户端，您可以按名称查询Kubernetes端点（请参阅[服务](https://kubernetes.io/docs/user-guide/services/" \t "https://www.springcloud.cc/_top)）。Kubernetes API服务器通常将服务公开为代表http和https地址的端点的集合，并且客户端可以从作为Pod运行的Spring Boot应用程序进行访问。Spring Cloud Kubernetes Ribbon项目也使用此发现功能来获取为要进行负载平衡的应用程序定义的端点列表。

您可以通过在项目内部添加以下依赖项来免费获得这些东西：

<dependency>

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

<artifactId>spring-cloud-starter-kubernetes</artifactId></dependency>

要启用DiscoveryClient的加载，请将@EnableDiscoveryClient添加到相应的配置或应用程序类中，如以下示例所示：

@SpringBootApplication@EnableDiscoveryClient**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

然后，您可以简单地通过自动装配将客户端注入代码中，如以下示例所示：

@Autowired**private** DiscoveryClient discoveryClient;

您可以通过在application.properties中设置以下属性来选择从所有命名空间启用DiscoveryClient：

spring.cloud.kubernetes.discovery.all-namespaces=true

如果出于任何原因需要禁用DiscoveryClient，则可以在application.properties中设置以下属性：

spring.cloud.kubernetes.discovery.enabled=false

某些Spring Cloud组件使用DiscoveryClient来获取有关本地服务实例的信息。为此，您需要将Kubernetes服务名称与spring.application.name属性对齐。

|  |
| --- |
| IMG_638 |
| spring.application.name对在Kubernetes中为该应用程序注册的名称无效 |

Spring Cloud Kubernetes也可以监视Kubernetes服务目录中的更改并相应地更新DiscoveryClient实现。为了启用此功能，您需要在应用程序的配置类上添加@EnableScheduling。

## **139. Kubernetes本机服务发现**

Kubernetes本身具有（服务器端）服务发现的能力（请参阅：[https](https://kubernetes.io/docs/concepts/services-networking/service/" \l "discovering-services" \t "https://www.springcloud.cc/_top) ://kubernetes.io/docs/concepts/services-networking/service/#discovering-services ）。使用本机kubernetes服务发现可确保与其他工具的兼容性，例如Istio（[https://istio.io](https://istio.io/" \t "https://www.springcloud.cc/_top)），该服务网格可实现负载平衡，功能区，断路器，故障转移等。

然后，调用者服务仅需要引用特定Kubernetes群集中可解析的名称。一个简单的实现可以使用SpringRestTemplate来引用完全限定域名（FQDN），例如[https://{service-name}.{namespace}.svc.{cluster}.local:{service-port}](https://{service-name}.{namespace}.svc.{cluster}.local:{service-port}/" \t "https://www.springcloud.cc/_top)。

此外，您可以将Hystrix用于：

* 通过用@EnableCircuitBreaker注释spring boot应用程序类，在调用方实现断路器
* 后备功能，通过用@HystrixCommand(fallbackMethod=注释相应的方法

## **140. Kubernetes PropertySource实现**

配置Spring Boot应用程序的最常用方法是创建一个application.properties或applicaiton.yaml或application-profile.properties或application-profile.yaml文件，其中包含为应用程序提供自定义值的键值对或Spring Boot首发。您可以通过指定系统属性或环境变量来覆盖这些属性。

## **140.1使用ConfigMap PropertySource**

Kubernetes提供了一个资源[ConfigMap](https://kubernetes.io/docs/user-guide/configmap/" \t "https://www.springcloud.cc/_top)，用于以键值对形式或嵌入的application.properties或application.yaml文件的形式来外部化要传递给您的应用程序的参数。在[Spring Cloud Kubernetes配置](https://www.springcloud.cc/spring-cloud-kubernetes-config" \t "https://www.springcloud.cc/_top)项目使得Kubernetes ConfigMap实例中应用自举可用和触发器时观察到ConfigMap实例中检测到的变化热重装beans或Spring上下文。

默认行为是根据Kubernetes ConfigMap创建一个ConfigMapPropertySource，其值是Spring应用程序名称（由其spring.application.name属性定义）的metadata.name值，或者在bootstrap.properties文件中的以下关键字下定义的自定义名称：spring.cloud.kubernetes.config.name。

但是，可以在其中使用多个ConfigMap实例的情况下进行更高级的配置。spring.cloud.kubernetes.config.sources列表使之成为可能。例如，您可以定义以下ConfigMap实例：

spring: application: name: cloud-k8s-app cloud: kubernetes: config: name: default-name namespace: default-namespace sources:

*# Spring Cloud Kubernetes looks up a ConfigMap named c1 in namespace default-namespace* - name: c1

*# Spring Cloud Kubernetes looks up a ConfigMap named default-name in whatever namespace n2* - namespace: n2

*# Spring Cloud Kubernetes looks up a ConfigMap named c3 in namespace n3* - namespace: n3 name: c3

在前面的示例中，如果未设置spring.cloud.kubernetes.config.namespace，则将在应用程序运行的名称空间中查找名为c1的ConfigMap。

找到的所有匹配ConfigMap都将按以下方式处理：

* 应用单个配置属性。
* 将名为application.yaml的任何属性的内容用作yaml。
* 将名为application.properties的任何属性的内容用作属性文件。

上述流程的唯一例外是ConfigMap包含**单个**密钥，该密钥指示文件是YAML或属性文件。在这种情况下，键的名称不必是application.yaml或application.properties（可以是任何东西），并且属性值可以正确处理。此功能有助于使用以下情况创建ConfigMap的用例：

kubectl create configmap game-config --from-file=/path/to/app-config.yaml

假设我们有一个名为demo的Spring Boot应用程序，它使用以下属性读取其线程池配置。

* pool.size.core
* pool.size.maximum

可以将其外部化为yaml格式的配置映射，如下所示：

kind: ConfigMapapiVersion: v1metadata: name: demodata: pool.size.core: 1 pool.size.max: 16

在大多数情况下，单个属性都可以正常工作。但是，有时嵌入yaml会更方便。在这种情况下，我们使用名为application.yaml的单个属性来嵌入yaml，如下所示：

kind: ConfigMapapiVersion: v1metadata: name: demodata: application.yaml: |- pool: size: core: 1 max:16

以下示例也适用：

kind: ConfigMapapiVersion: v1metadata: name: demodata: custom-name.yaml: |- pool: size: core: 1 max:16

您还可以根据读取ConfigMap时合并在一起的活动配置文件来不同地配置Spring Boot应用程序。您可以通过使用application.properties或application.yaml属性，为特定的配置文件提供不同的属性值，并在他们自己的文档中指定特定于配置文件的值（由---序列指示），如下所示：

kind: ConfigMapapiVersion: v1metadata: name: demodata: application.yml: |- greeting: message: Say Hello to the World farewell: message: Say Goodbye

--- spring: profiles: development greeting: message: Say Hello to the Developers farewell: message: Say Goodbye to the Developers

--- spring: profiles: production greeting: message: Say Hello to the Ops

在上述情况下，使用development配置文件加载到Spring应用程序中的配置如下：

greeting: message: Say Hello to the Developers farewell: message: Say Goodbye to the Developers

但是，如果production配置文件处于活动状态，则配置将变为：

greeting: message: Say Hello to the Ops farewell: message: Say Goodbye

如果两个配置文件均处于活动状态，则在ConfigMap中最后出现的属性将覆盖之前的所有值。

另一个选择是为每个配置文件创建一个不同的配置映射，spring boot将根据活动的配置文件自动获取它

kind: ConfigMapapiVersion: v1metadata: name: demodata: application.yml: |- greeting: message: Say Hello to the World farewell: message: Say Goodbye

kind: ConfigMapapiVersion: v1metadata: name: demo-developmentdata: application.yml: |- spring: profiles: development greeting: message: Say Hello to the Developers farewell: message: Say Goodbye to the Developers

kind: ConfigMapapiVersion: v1metadata: name: demo-productiondata: application.yml: |- spring: profiles: production greeting: message: Say Hello to the Ops farewell: message: Say Goodbye

要告诉Spring Boot应该在引导程序中启用哪个profile，可以将系统属性传递给Java命令。为此，您可以使用环境变量来启动Spring Boot应用程序，该环境变量可以通过OpenShift DeploymentConfig或Kubernetes ReplicationConfig资源文件进行定义，如下所示：

apiVersion: v1kind: DeploymentConfigspec: replicas: 1

... spec: containers: - env: - name: JAVA\_APP\_DIR value: /deployments - name: JAVA\_OPTIONS value: -Dspring.profiles.active=developer

|  |
| --- |
| IMG_639 |
| 您应该检查安全性配置部分。要从Pod内部访问配置映射，您需要具有正确的Kubernetes服务帐户，角色和角色绑定。 |

使用ConfigMap实例的另一种选择是通过运行Spring Cloud Kubernetes应用程序并使Spring Cloud Kubernetes从文件系统读取它们来将它们装入Pod。此行为由spring.cloud.kubernetes.config.paths属性控制。您可以使用它作为上述机制的补充或替代。您可以使用,定界符在spring.cloud.kubernetes.config.paths中指定多个（精确）文件路径。

|  |
| --- |
| IMG_640 |
| 您必须提供每个属性文件的完整确切路径，因为不会递归解析目录。 |

**表140.1。Properties:**

| **名称** | **类型** | **默认** | **描述** |
| --- | --- | --- | --- |
| **spring.cloud.kubernetes.config.enableApi** | **Boolean** | **true** | **通过API启用或禁用使用ConfigMap实例** |
| spring.cloud.kubernetes.config.enabled | Boolean | true | 启用ConfigMaps PropertySource |
| spring.cloud.kubernetes.config.name | String | ${spring.application.name} | 设置ConfigMap的名称以查找 |
| spring.cloud.kubernetes.config.namespace | String | 客户端名称空间 | 设置Kubernetes命名空间在哪里查找 |
| spring.cloud.kubernetes.config.paths | List | null | 设置安装ConfigMap实例的路径 |

## **140.2秘密PropertySource**

Kubernetes具有用于存储敏感数据（例如密码，OAuth令牌等）的[秘密](https://kubernetes.io/docs/concepts/configuration/secret/" \t "https://www.springcloud.cc/_top)的概念。该项目提供了与Secrets的集成，以使Spring Boot应用程序可以访问机密。您可以通过设置spring.cloud.kubernetes.secrets.enabled属性来显式启用或禁用此功能。

启用后，SecretsPropertySource将从以下来源中为Secrets查找Kubernetes：

1. 从秘密坐骑递归读取
2. 以应用程序命名（由spring.application.name定义）
3. 匹配一些标签

**注意：**

默认情况下，出于安全原因，**未启用**通过API消费机密（以上第2点和第3点）。机密上的权限“列表”允许客户端检查指定名称空间中的机密值。此外，我们建议容器通过安装的卷共享机密。

如果您通过API启用使用机密，我们建议您使用授权策略（例如RBAC）限制对机密的访问。有关通过API使用“机密”时的风险和最佳做法的更多信息，请参阅[此文档](https://kubernetes.io/docs/concepts/configuration/secret/" \l "best-practices" \t "https://www.springcloud.cc/_top)。

如果找到了机密，则其数据可供应用程序使用。

假设我们有一个名为demo的spring boot应用程序，该应用程序使用属性读取其数据库配置。我们可以使用以下命令创建Kubernetes机密：

oc create secret generic db-secret --from-literal=username=user --from-literal=password=p455w0rd

前面的命令将创建以下秘密（您可以使用oc get secrets db-secret -o yaml来查看）：

apiVersion: v1data: password: cDQ1NXcwcmQ= username: dXNlcg==kind: Secretmetadata: creationTimestamp: 2017-07-04T09:15:57Z name: db-secret namespace: default resourceVersion: "357496" selfLink: /api/v1/namespaces/default/secrets/db-secret uid: 63c89263-6099-11e7-b3da-76d6186905a8type: Opaque

请注意，数据包含create命令提供的文字的Base64编码版本。

然后，您的应用程序可以使用此秘密-例如，通过将秘密的值导出为环境变量：

apiVersion: v1kind: Deploymentmetadata: name: ${project.artifactId**}**spec: template: spec: containers: - env: - name: DB\_USERNAME valueFrom: secretKeyRef: name: db-secret key: username - name: DB\_PASSWORD valueFrom: secretKeyRef: name: db-secret key: password

您可以通过多种方式选择要使用的秘密：

通过列出映射机密的目录：

-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/db-secret,etc/secrets/postgresql

如果您已将所有机密映射到公共根，则可以将它们设置为：

-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets

通过设置命名机密：

-Dspring.cloud.kubernetes.secrets.name=db-secret

通过定义标签列表：

-Dspring.cloud.kubernetes.secrets.labels.broker=activemq

-Dspring.cloud.kubernetes.secrets.labels.db=postgresql

**表140.2。Properties:**

| **名称** | **类型** | **默认** | **描述** |
| --- | --- | --- | --- |
| **spring.cloud.kubernetes.secrets.enableApi** | **Boolean** | **false** | **通过API启用或禁用使用机密（示例2和3）** |
| spring.cloud.kubernetes.secrets.enabled | Boolean | true | 启用机密PropertySource |
| spring.cloud.kubernetes.secrets.name | String | ${spring.application.name} | 设置要查找的机密名称 |
| spring.cloud.kubernetes.secrets.namespace | String | 客户端名称空间 | 设置Kubernetes命名空间的查找位置 |
| spring.cloud.kubernetes.secrets.labels | Map | null | 设置用于查找机密的标签 |
| spring.cloud.kubernetes.secrets.paths | List | null | 设置安装机密的路径（示例1） |

笔记：

* spring.cloud.kubernetes.secrets.labels属性的行为如[基于Map的绑定](https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-Configuration-Binding" \l "map-based-binding" \t "https://www.springcloud.cc/_top)所定义 。
* spring.cloud.kubernetes.secrets.paths属性的行为与[基于Collection的binding](https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-Configuration-Binding" \l "collection-based-binding" \t "https://www.springcloud.cc/_top)定义的行为相同 。
* 出于安全原因，可能会限制通过API访问机密。首选方法是将机密安装到Pod。

您可以在[spring-boot-camel-config中](https://github.com/fabric8-quickstarts/spring-boot-camel-config" \t "https://www.springcloud.cc/_top)找到使用机密的应用程序示例（尽管尚未更新以使用新的spring-cloud-kubernetes项目）。

## **140.3 PropertySource重新加载**

某些应用程序可能需要检测外部属性源上的更改并更新其内部状态以反映新配置。当相关的ConfigMap或Secret发生更改时，Spring Cloud Kubernetes的重载功能能够触发应用程序重载。

默认情况下，此功能处于禁用状态。您可以使用spring.cloud.kubernetes.reload.enabled=true配置属性（例如，在application.properties文件中）启用它。

支持以下级别的重载（通过设置spring.cloud.kubernetes.reload.strategy属性）：\* refresh（默认）：仅重载用@ConfigurationProperties或@RefreshScope注释的配置beans。此重新加载级别利用了Spring Cloud上下文的刷新功能。\* restart\_context：整个Spring ApplicationContext已正常重启。用新配置重新创建Beans。\* shutdown：Spring ApplicationContext已关闭，以激活容器的重启。使用此级别时，请确保所有非守护程序线程的生命周期都绑定到ApplicationContext，并且已将复制控制器或副本集配置为重新启动Pod。

假设使用默认设置（refresh模式）启用了重新加载功能，则当配置映射更改时，将刷新以下bean：

@Configuration

@ConfigurationProperties(prefix = "bean")

public class MyConfig {

private String message = "a message that can be changed live";

// getter and setters

}

要查看更改是否有效发生，您可以创建另一个bean来定期打印消息，如下所示

@Component**public** **class** MyBean {

@Autowired

**private** MyConfig config;

@Scheduled(fixedDelay = 5000)

**public** **void** hello() {

System.out.println("The message is: " + config.getMessage());

}

}

您可以使用ConfigMap来更改应用程序打印的消息，如下所示：

apiVersion: v1kind: ConfigMapmetadata: name: reload-exampledata: application.properties: |- bean.message=Hello World!

与容器关联的ConfigMap中名为bean.message的属性的任何更改都会反映在输出中。更一般而言，将检测与属性相关联的更改，这些更改的前缀为@ConfigurationProperties注释的prefix字段定义的值。 本章前面已经说明[了将ConfigMap与pod关联](https://www.springcloud.cc/spring-cloud-greenwich.html" \l "configmap-propertysource" \o "140.1使用ConfigMap PropertySource)。

完整的示例可在中找到[spring-cloud-kubernetes-reload-example](https://github.com/fabric8io/spring-cloud-kubernetes/tree/master/spring-cloud-kubernetes-examples/kubernetes-reload-example" \t "https://www.springcloud.cc/_top)。

重新加载功能支持两种操作模式：\*事件（默认）：使用Kubernetes API（web套接字）监视配置映射或机密的更改。任何事件都会对配置进行重新检查，并在发生更改的情况下重新加载。服务帐户上的view角色是必需的，以便侦听配置映射更改。秘密需要更高级别的角色（例如edit）（默认情况下，不监视秘密）。\*轮询：从配置上通过配置映射和机密重新创建配置，以查看配置是否已更改。您可以使用spring.cloud.kubernetes.reload.period属性来配置轮询时间，默认为15秒。它需要与受监视属性源相同的角色。例如，这意味着对文件挂载的秘密源使用轮询不需要特定的特权。

**表140.3。Properties:**

| **名称** | **类型** | **默认** | **描述** |
| --- | --- | --- | --- |
| **spring.cloud.kubernetes.reload.period** | **Duration** | **15s** | **使用polling策略时验证更改的期限** |
| spring.cloud.kubernetes.reload.enabled | Boolean | false | 启用监视属性源和配置重载 |
| spring.cloud.kubernetes.reload.monitoring-config-maps | Boolean | true | 允许监视配置映射中的更改 |
| spring.cloud.kubernetes.reload.monitoring-secrets | Boolean | false | 允许监视机密更改 |
| spring.cloud.kubernetes.reload.strategy` | Enum | refresh | 触发重新加载时使用的策略（refresh，restart\_context或shutdown） |
| spring.cloud.kubernetes.reload.mode | Enum | event | 指定如何侦听属性源（event或polling）中的更改 |

注意：\*请勿在配置映射或机密中使用spring.cloud.kubernetes.reload下的属性。在运行时更改此类属性可能会导致意外结果。\*使用refresh级别时，删除属性或整个配置图不会恢复beans的原始状态。

## **141. Ribbon在Kubernetes中发现**

Spring Cloud调用微服务的客户端应用程序应该对依靠客户端负载平衡功能感兴趣，以便自动发现它可以在哪个端点到达给定服务。该机制已在[spring-cloud-kubernetes-ribbon](https://github.com/spring-cloud/spring-cloud-kubernetes/tree/master/spring-cloud-kubernetes-ribbon" \t "https://www.springcloud.cc/_top)项目中实现，其中Kubernetes客户端填充[Ribbon](https://github.com/Netflix/ribbon" \t "https://www.springcloud.cc/_top) ServerList，其中包含有关此类端点的信息。

该实现是以下启动器的一部分，可以通过将其依赖项添加到pom文件中来使用该实现：

<dependency>

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

<artifactId>spring-cloud-starter-kubernetes-ribbon</artifactId>

<version>${latest.version}</version></dependency>

填充端点列表后，Kubernetes客户端通过匹配Ribbon Client批注中定义的服务名称来搜索位于当前名称空间或项目中的已注册端点，如下所示：

@RibbonClient(name = "name-service")

您可以使用以下格式，在application.properties中（通过应用程序专用的ConfigMap）提供属性，以配置Ribbon的行为：<name of your service>.ribbon.<Ribbon configuration key>，其中：

* <name of your service>对应于您在Ribbon上访问的服务名称，该名称是使用@RibbonClient批注配置的（例如上例中的name-service）。
* <Ribbon configuration key>是[Ribbon的CommonClientConfigKey类](https://github.com/Netflix/ribbon/blob/master/ribbon-core/src/main/java/com/netflix/client/config/CommonClientConfigKey.java" \t "https://www.springcloud.cc/_top)定义的Ribbon配置键之一 。

此外，spring-cloud-kubernetes-ribbon项目定义了两个附加的配置键，以进一步控制Ribbon与Kubernetes的交互方式。特别是，如果端点定义了多个端口，则默认行为是使用找到的第一个端口。要更具体地选择在多端口服务中使用哪个端口，可以使用【7 /】键。如果您想指定应在哪个Kubernetes命名空间中查找目标服务，则可以使用KubernetesNamespace键，在这两个实例中都记住要为这些键加上您的服务名和ribbon前缀（如前所述）。

以下示例使用此模块进行功能区发现：

* [Spring Cloud断路器和Ribbon](https://www.springcloud.cc/spring-cloud-kubernetes-examples/kubernetes-circuitbreaker-ribbon-example" \t "https://www.springcloud.cc/_top)
* [fabric8-quickstarts-Spring Boot-Ribbon](https://github.com/fabric8-quickstarts/spring-boot-ribbon" \t "https://www.springcloud.cc/_top)
* [Kubeflix-贷款经纪人-银行](https://github.com/fabric8io/kubeflix/tree/master/examples/loanbroker/bank" \t "https://www.springcloud.cc/_top)

|  |
| --- |
| IMG_641 |
| 您可以通过在应用程序属性文件中设置spring.cloud.kubernetes.ribbon.enabled=false键来禁用Ribbon发现客户端。 |

## **142. Kubernetes生态系统意识**

无论您的应用程序是否在Kubernetes中运行，本指南前面介绍的所有功能都可以很好地工作。这对于开发和故障排除确实很有帮助。从开发角度来看，这使您可以启动Spring Boot应用程序并调试属于该项目的模块之一。您无需在Kubernetes中部署它，因为该项目的代码依赖于 [Fabric8 Kubernetes Java客户端](https://github.com/fabric8io/kubernetes-client" \t "https://www.springcloud.cc/_top)，该[客户端](https://github.com/fabric8io/kubernetes-client" \t "https://www.springcloud.cc/_top)是流利的DSL，可以使用http协议与Kubernetes Server的REST API进行通信。

要禁用与Kubernetes的集成，可以将spring.cloud.kubernetes.enabled设置为false。请注意，当spring-cloud-kubernetes-config在类路径上时，应在bootstrap.{properties|yml}（或特定于配置文件的文件）中设置spring.cloud.kubernetes.enabled，否则应在application.{properties|yml}（或特定于配置文件的文件）中进行设置。另请注意，以下属性：spring.cloud.kubernetes.config.enabled和spring.cloud.kubernetes.secrets.enabled仅在bootstrap.{properties|yml}中设置时才生效

## **142.1 Kubernetes配置文件自动配置**

当应用程序在Kubernetes中作为pod运行时，名为kubernetes的Spring配置文件将自动被激活。这使您可以自定义配置，以定义在Kubernetes平台中部署Spring Boot应用程序时要应用的beans（例如，不同的开发和生产配置）。

## **142.2 Istio意识**

当您在应用程序类路径的spring-cloud-kubernetes-istio模块，新的配置文件被添加到应用程序，提供的应用运行在Kubernetes集群里面[Istio](https://istio.io/" \t "https://www.springcloud.cc/_top)安装。然后，您可以在Beans和@Configuration类中使用spring @Profile("istio")注释。

Istio感知模块使用me.snowdrop:istio-client与Istio API进行交互，让我们发现流量规则，断路​​器等，从而使我们的Spring Boot应用程序更容易使用此数据来根据环境动态配置自身。

## **143.豆荚健康指标**

Spring Boot用于[HealthIndicator](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthEndpoint.java" \t "https://www.springcloud.cc/_top)公开有关应用程序运行状况的信息。这对于将与健康相关的信息提供给用户非常有用，并且非常适合用作[就绪探针](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/" \t "https://www.springcloud.cc/_top)。

Kubernetes运行状况指示器（是核心模块的一部分）公开以下信息：

* 窗格名称，IP地址，名称空间，服务帐户，节点名称及其IP地址
* 一个标志，指示Spring Boot应用程序是Kubernetes的内部应用程序还是外部应用程序

## **144. Leader选举**

<待定>

## **145. Kubernetes内部的安全配置**

## **145.1命名空间**

该项目中提供的大多数组件都需要知道名称空间。对于Kubernetes（1.3+），名称空间作为服务帐户密码的一部分可供Pod使用，并由客户端自动检测到。对于早期版本，需要将其指定为Pod的环境变量。一种快速的方法如下：

env:

- name: "KUBERNETES\_NAMESPACE"

valueFrom:

fieldRef:

fieldPath: "metadata.namespace"

## **145.2服务帐号**

对于支持集群内基于角色的更细粒度访问的Kubernetes发行版，您需要确保以spring-cloud-kubernetes运行的Pod有权访问Kubernetes API。对于您分配给部署或Pod的任何服务帐户，您需要确保它们具有正确的角色。例如，您可以根据您所在的项目向您的default服务帐户添加cluster-reader权限。

## **146.服务注册中心的实施**

在Kubernetes服务注册由平台控制的情况下，应用程序本身不像其他平台那样控制注册。因此，在Spring Cloud Kubernetes中使用spring.cloud.service-registry.auto-registration.enabled或设置@EnableDiscoveryClient(autoRegister=false)无效。

## **147.范例**

Spring Cloud Kubernetes尝试通过遵循Spring Cloud接口，使应用程序使用Kubernetes Native Services透明化。

在您的应用程序中，您需要向类路径中添加spring-cloud-kubernetes-discovery依赖项，并删除包含DiscoveryClient实现的任何其他依赖项（即，一个Eureka发现客户端）。PropertySourceLocator的情况与此相同，您需要在其中将spring-cloud-kubernetes-config添加到类路径，并删除包含PropertySourceLocator实现的其他任何依赖项（即配置服务器客户端）。

以下项目重点介绍了这些依赖项的用法，并演示了如何从任何Spring Boot应用程序中使用这些库：

* [Spring Cloud Kubernetes示例](https://github.com/spring-cloud/spring-cloud-kubernetes/tree/master/spring-cloud-kubernetes-examples" \t "https://www.springcloud.cc/_top)：位于此存储库中的[示例](https://github.com/spring-cloud/spring-cloud-kubernetes/tree/master/spring-cloud-kubernetes-examples" \t "https://www.springcloud.cc/_top)。

Spring Cloud Kubernetes完整示例：奴才和老板

* + [奴才](https://github.com/salaboy/spring-cloud-k8s-minion" \t "https://www.springcloud.cc/_top)
  + [老板](https://github.com/salaboy/spring-cloud-k8s-boss" \t "https://www.springcloud.cc/_top)
* Spring Cloud Kubernetes完整示例：[SpringOne Platform Tickets Service](https://github.com/salaboy/s1p_docs" \t "https://www.springcloud.cc/_top)
* [具有Spring Cloud Kubernetes发现和配置的Spring Cloud网关](https://github.com/salaboy/s1p_gateway" \t "https://www.springcloud.cc/_top)
* [Spring Boot使用Spring Cloud进行Kubernetes发现和配置的管理员](https://github.com/salaboy/showcase-admin-tool" \t "https://www.springcloud.cc/_top)

## **148.其他资源**

本节列出了其他资源，例如有关Spring Cloud Kubernetes的演示（幻灯片）和视频。

* [PKS上的S1P Spring Cloud](https://salaboy.com/2018/09/27/the-s1p-experience/" \t "https://www.springcloud.cc/_top)
* [Spring Cloud，Docker，Kubernetes→伦敦Java社区，2018年7月](https://salaboy.com/2018/07/18/ljc-july-18-spring-cloud-docker-k8s/" \t "https://www.springcloud.cc/_top)

请随时通过拉取请求向[此存储库](https://github.com/spring-cloud/spring-cloud-kubernetes" \t "https://www.springcloud.cc/_top)提交其他资源。

## **149.建筑**

## **149.1基本编译和测试**

要构建源代码，您将需要安装JDK 1.7。

Spring Cloud将Maven用于大多数与构建相关的活动，并且应该可以通过克隆您感兴趣的项目并键入来快速开始工作

$ ./mvnw install

|  |
| --- |
| IMG_642 |
| 您还可以自己安装Maven（> = 3.3.3），并在下面的示例中运行mvn命令代替./mvnw。如果这样做，则您的本地Maven设置不包含Spring预发行项目的存储库声明，则可能还需要添加-P spring。 |

|  |
| --- |
| IMG_643 |
| 请注意，您可能需要通过将MAVEN\_OPTS环境变量设置为类似-Xmx512m -XX:MaxPermSize=128m的值来增加Maven的可用内存量。我们尝试在.mvn配置中进行介绍，因此，如果您必须执行此操作才能使构建成功，请提出票证以将设置添加到源代码管理中。 |

有关如何构建项目的提示，请查看.travis.yml（如果有）。应该有一个“脚本”，也许还有“安装”命令。另外，请查看“服务”部分，以查看是否需要在本地运行任何服务（例如mongo或Rabbit）。忽略您可能在“ before\_install”中找到的与git相关的位，因为它们与设置git凭据有关，并且您已经拥有了这些凭据。

需要中间件的项目通常包括docker-compose.yml，因此请考虑使用 [Docker Compose](https://docs.docker.com/compose/" \t "https://www.springcloud.cc/_top)在Docker容器中运行中间件服务器。有关mongo，rabbit和redis常见情况的特定说明，请参见[脚本演示存储库中](https://github.com/spring-cloud-samples/scripts" \t "https://www.springcloud.cc/_top)的README 。

|  |
| --- |
| IMG_644 |
| 如果所有其他方法均失败，请使用.travis.yml（通常为./mvnw install）中的命令进行构建。 |

## **149.2文档**

spring-cloud-build模块有一个“ docs”配置文件，如果打开它，将尝试从src/main/asciidoc构建asciidoc源。作为该过程的一部分，它将寻找README.adoc并通过加载所有包含文件来对其进行处理，但不对其进行解析或渲染，只需将其复制到${main.basedir}（默认为$../../../..，即根目录）即可。该项目）。如果自述文件有任何更改，它将在Maven构建后显示为正确位置的修改文件。只需提交并推动更改即可。

## **149.3使用代码**

如果您没有IDE偏好设置，我们建议您在使用代码时使用 [Spring Tools Suite](https://www.springsource.com/developer/sts" \t "https://www.springcloud.cc/_top)或 [Eclipse](https://eclipse.org/" \t "https://www.springcloud.cc/_top)。我们使用 [m2eclipse](https://eclipse.org/m2e/" \t "https://www.springcloud.cc/_top) eclipse插件来获得maven支持。只要其他IDE和工具使用Maven 3.3.3或更高版本，它们也应该可以正常工作。

### **149.3.1使用m2eclipse导入eclipse**

当使用eclipse时，我们建议使用[m2eclipse](https://eclipse.org/m2e/" \t "https://www.springcloud.cc/_top) eclipse插件。如果尚未安装m2eclipse，则可以从“ eclipse市场”中获得。

|  |
| --- |
| IMG_645 |
| 较旧的m2e版本不支持Maven 3.3，因此一旦将项目导入Eclipse，您还需要告诉m2eclipse为项目使用正确的配置文件。如果您在项目中看到许多与POM相关的错误，请检查您是否具有最新的安装。如果您无法升级m2e，请将“ spring”配置文件添加到settings.xml中。或者，您可以将存储库设置从父pom的“ spring”配置文件复制到settings.xml中。 |

### **149.3.2不带m2eclipse导入eclipse**

如果您不想使用m2eclipse，则可以使用以下命令生成eclipse项目元数据：

$ ./mvnw eclipse:eclipse

可以通过从file菜单中选择import existing projects来导入生成的日食项目。

## **150.贡献**

Spring Cloud是在非限制性Apache 2.0许可下发布的，遵循非常标准的Github开发流程，使用Github跟踪程序解决问题并将合并请求合并到master中。如果您想贡献些微不足道的东西，请不要犹豫，但请遵循以下准则。

## **150.1签署贡献者许可协议**

在我们接受不重要的补丁或请求请求之前，我们将需要您签署“ [贡献者许可协议”](https://cla.pivotal.io/sign/spring" \t "https://www.springcloud.cc/_top)。签署贡献者协议并不会授予任何人对主存储库的提交权，但这确实意味着我们可以接受您的贡献，如果这样做，您将获得作者的荣誉。可能会要求活跃的贡献者加入核心团队，并具有合并合并请求的能力。

## **150.2行为准则**

该项目遵守《贡献者公约》[行为守则](https://github.com/spring-cloud/spring-cloud-build/blob/master/docs/src/main/asciidoc/code-of-conduct.adoc" \t "https://www.springcloud.cc/_top)。通过参与，您将遵守此代码。请向[spring-code-of-conduct@pivotal.io](mailto:spring-code-of-conduct@pivotal.io" \t "https://www.springcloud.cc/_top)报告不可接受的行为。

## **150.3编码约定和客房整理**

其中的None对于请求请求至关重要，但它们都会有所帮助。也可以在原始请求请求之后但在合并之前添加它们。

* 使用Spring Framework代码格式约定。如果您使用Eclipse，则可以使用[Spring Cloud Build](https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-dependencies-parent/eclipse-code-formatter.xml" \t "https://www.springcloud.cc/_top)项目中的eclipse-code-formatter.xml文件导入格式化程序设置 。如果使用IntelliJ，则可以使用 [Eclipse Code Formatter插件](https://plugins.jetbrains.com/plugin/6546" \t "https://www.springcloud.cc/_top)来导入相同的文件。
* 确保所有新的.java文件都具有简单的Javadoc类注释，并至少带有一个@author标签来标识您，并且最好至少包含有关该类用途的段落。
* 将ASF许可证标头注释添加到所有新的.java文件（从项目中的现有文件复制）
* 将您自己作为@author添加到您进行了实质性修改（不仅仅是外观更改）的.java文件中。
* 添加一些Javadocs，如果更改名称空间，则添加一些XSD doc元素。
* 进行一些单元测试也有很大帮助-有人必须这样做。
* 如果没有其他人在使用您的分支，请根据当前的主节点（或主项目中的其他目标分支）对其进行重新设置。
* 编写提交消息时，请遵循[以下约定](https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html" \t "https://www.springcloud.cc/_top)，如果要解决现有问题，请在提交消息的末尾添加Fixes gh-XXXX（其中XXXX是问题编号）。

## **150.4 Checkstyle**

Spring Cloud Build带有一组Checkstyle规则。您可以在spring-cloud-build-tools模块中找到它们。该模块下最值得注意的文件是：

**spring-cloud-build-tools /。**

└── src

   ├── checkstyle

   │   └── checkstyle-suppressions.xml IMG_646

   └── main

   └── resources

   ├── checkstyle-header.txt IMG_647

   └── checkstyle.xml IMG_648

|  |  |
| --- | --- |
| [IMG_649](https://www.springcloud.cc/spring-cloud-greenwich.html#CO7-3) | 默认Checkstyle规则 |
| [IMG_650](https://www.springcloud.cc/spring-cloud-greenwich.html#CO7-2) | 文件头设置 |
| [IMG_651](https://www.springcloud.cc/spring-cloud-greenwich.html#CO7-1) | 默认抑制规则 |

### **150.4.1 Checkstyle配置**

Checkstyle规则**默认**为**禁用**。要将checkstyle添加到项目中，只需定义以下属性和插件。

**pom.xml。**

<properties>

<maven-checkstyle-plugin.failsOnError>true</maven-checkstyle-plugin.failsOnError> IMG_652

<maven-checkstyle-plugin.failsOnViolation>true

</maven-checkstyle-plugin.failsOnViolation> IMG_653

<maven-checkstyle-plugin.includeTestSourceDirectory>true

</maven-checkstyle-plugin.includeTestSourceDirectory> IMG_654

</properties>

<build>

<plugins>

<plugin> IMG_655

<groupId>io.spring.javaformat</groupId>

<artifactId>spring-javaformat-maven-plugin</artifactId>

</plugin>

<plugin> IMG_656

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-checkstyle-plugin</artifactId>

</plugin>

</plugins>

<reporting>

<plugins>

<plugin> IMG_657

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-checkstyle-plugin</artifactId>

</plugin>

</plugins>

</reporting>

</build>

|  |  |
| --- | --- |
| [IMG_658](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-1) | 因Checkstyle错误而无法构建 |
| [IMG_659](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-2) | 因Checkstyle违规而无法构建 |
| [IMG_660](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-3) | Checkstyle还分析测试源 |
| [IMG_661](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-4) | 添加Spring Java格式插件，该插件将重新格式化您的代码以通过大多数Checkstyle格式设置规则 |
| [IMG_662](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-5) [IMG_663](https://www.springcloud.cc/spring-cloud-greenwich.html#CO8-6) | 将Checkstyle插件添加到您的构建和报告阶段 |

如果您需要取消某些规则（例如，行长需要更长），那么只需在${project.root}/src/checkstyle/checkstyle-suppressions.xml下定义一个文件就可以了。例：

**projectRoot / src / checkstyle / checkstyle-suppresions.xml。**

<?xml version="1.0"?>

<!DOCTYPE suppressions PUBLIC

"-//Puppy Crawl//DTD Suppressions 1.1//EN"

"https://www.puppycrawl.com/dtds/suppressions\_1\_1.dtd">

<suppressions>

<suppress files=".\*ConfigServerApplication\.java" checks="HideUtilityClassConstructor"/>

<suppress files=".\*ConfigClientWatch\.java" checks="LineLengthCheck"/>

</suppressions>

建议将${spring-cloud-build.rootFolder}/.editorconfig和${spring-cloud-build.rootFolder}/.springformat复制到您的项目中。这样，将应用一些默认格式设置规则。您可以通过运行以下脚本来这样做：

$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/.editorconfig -o .editorconfig

$ touch .springformat

## **150.5 IDE设置**

### **150.5.1 Intellij IDEA**

为了设置Intellij，您应该导入我们的编码约定，检查配置文件并设置checkstyle插件。在[Spring Cloud Build](https://github.com/spring-cloud/spring-cloud-build/tree/master/spring-cloud-build-tools" \t "https://www.springcloud.cc/_top)项目中可以找到以下文件。

**spring-cloud-build-tools /。**

└── src

   ├── checkstyle

   │   └── checkstyle-suppressions.xml IMG_664

   └── main

   └── resources

   ├── checkstyle-header.txt IMG_665

   ├── checkstyle.xml IMG_666

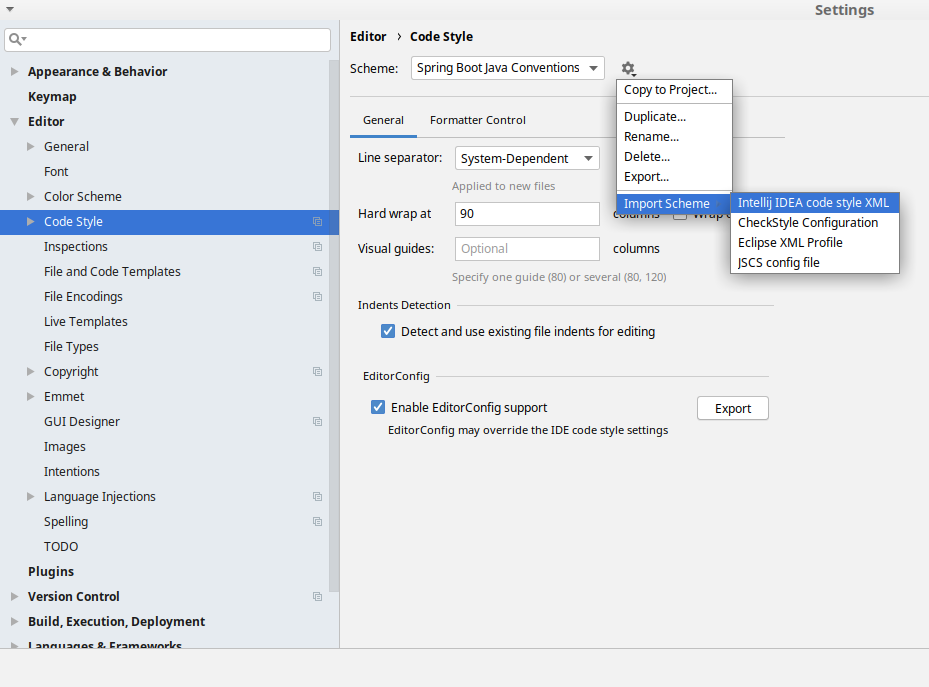
   └── intellij

      ├── Intellij\_Project\_Defaults.xml IMG_667

      └── Intellij\_Spring\_Boot\_Java\_Conventions.xml IMG_668

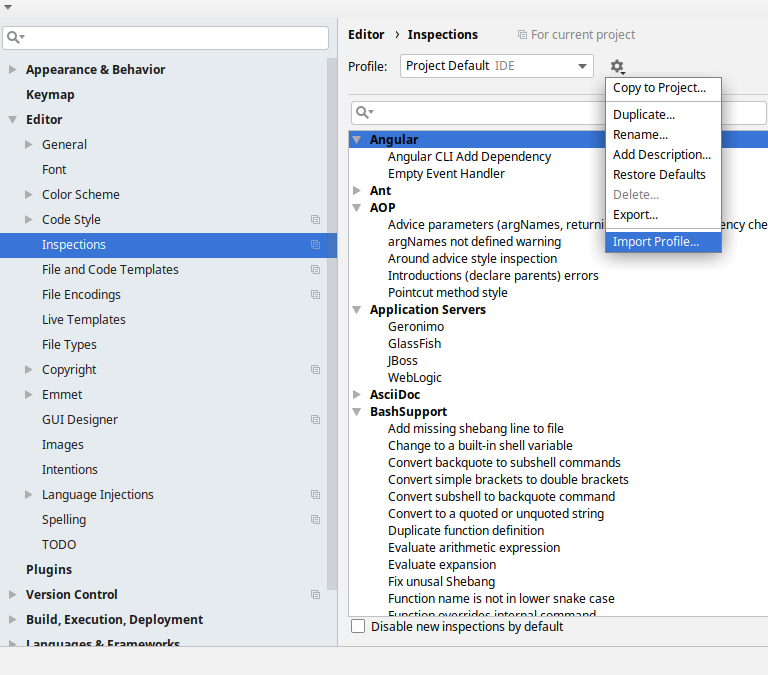
|  |  |
| --- | --- |
| [IMG_669](https://www.springcloud.cc/spring-cloud-greenwich.html#CO9-3) | 默认Checkstyle规则 |
| [IMG_670](https://www.springcloud.cc/spring-cloud-greenwich.html#CO9-2) | 文件头设置 |
| [IMG_671](https://www.springcloud.cc/spring-cloud-greenwich.html#CO9-1) | 默认抑制规则 |
| [IMG_672](https://www.springcloud.cc/spring-cloud-greenwich.html#CO9-4) | 适用于大多数Checkstyle规则的Intellij项目默认值 |
| [IMG_673](https://www.springcloud.cc/spring-cloud-greenwich.html#CO9-5) | 适用于大多数Checkstyle规则的Intellij的项目样式约定 |

**图150.1 代码风格**



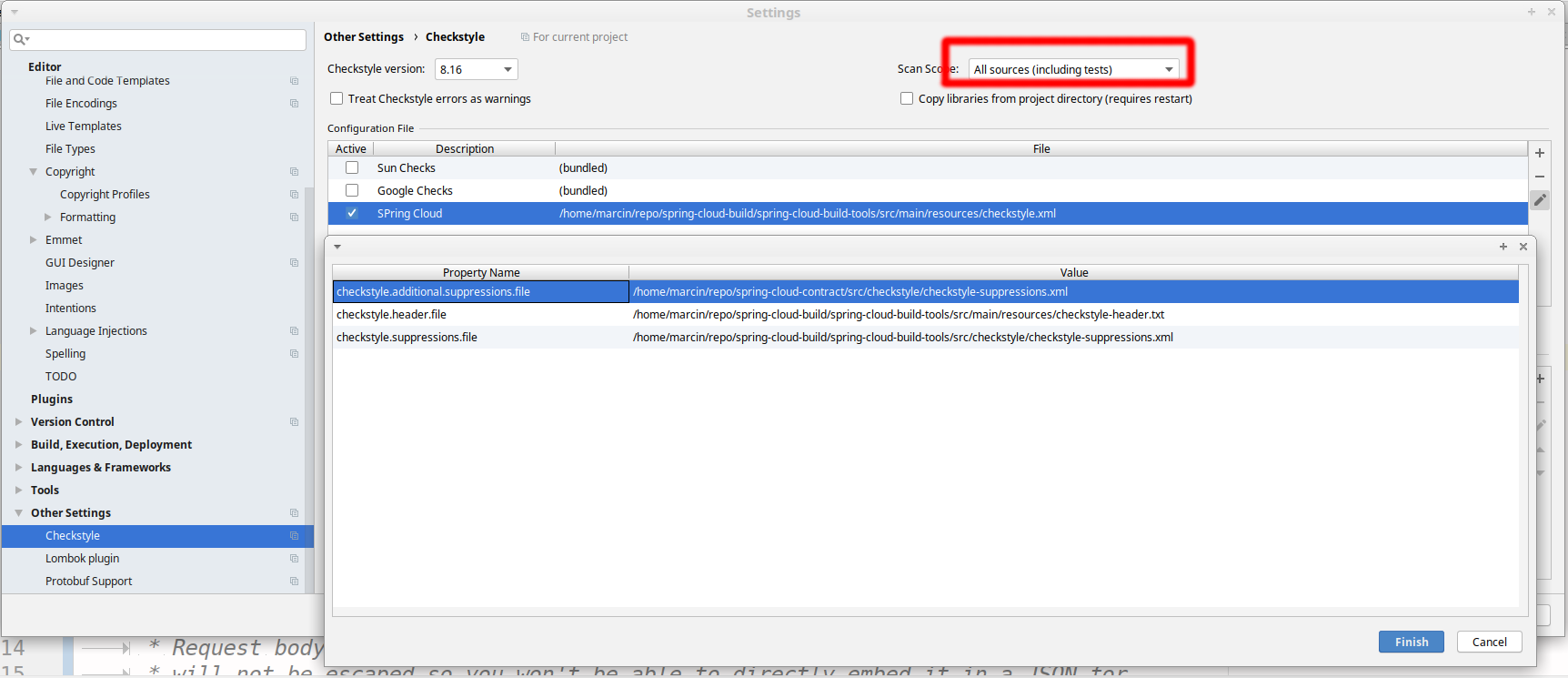
转到File→Settings→Editor→Code style。单击Scheme部分旁边的图标。在那里，单击Import Scheme值，然后选择Intellij IDEA code style XML选项。导入spring-cloud-build-tools/src/main/resources/intellij/Intellij\_Spring\_Boot\_Java\_Conventions.xml文件。

**图150.2 检验概况**



转到File→Settings→Editor→Inspections。单击Profile部分旁边的图标。在此处单击Import Profile，然后导入spring-cloud-build-tools/src/main/resources/intellij/Intellij\_Project\_Defaults.xml文件。

**Checkstyle。**要使Intellij与Checkstyle一起使用，您必须安装Checkstyle插件。建议还安装Assertions2Assertj以自动转换JUnit断言



转到File→Settings→Other settings→Checkstyle。在Configuration file部分中单击+图标。在这里，您必须定义应从何处选择checkstyle规则。在上图中，我们从克隆的Spring Cloud构建库中选择了规则。但是，您可以指向Spring Cloud构建的GitHub存储库（例如，对于checkstyle.xml：[https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml](https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml" \t "https://www.springcloud.cc/_top)）。我们需要提供以下变量：

* checkstyle.header.file-请在克隆的存储库中或通过[https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt](https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt" \t "https://www.springcloud.cc/_top) URL将其指向Spring Cloud构建的spring-cloud-build-tools/src/main/resources/checkstyle-header.txt文件。
* checkstyle.suppressions.file-默认禁止。请在克隆的存储库中或通过[https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml](https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml" \t "https://www.springcloud.cc/_top) URL将其指向Spring Cloud构建的spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml文件。
* checkstyle.additional.suppressions.file-此变量对应于本地项目中的取消显示。例如，您正在研究spring-cloud-contract。然后指向project-root/src/checkstyle/checkstyle-suppressions.xml文件夹。spring-cloud-contract的示例为：/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml。

|  |  |
| --- | --- |
| IMG_677 | **重要** |
| 请记住将Scan Scope设置为All sources，因为我们将checkstyle规则应用于生产和测试源。 |

# **第十八部分。Spring Cloud GCP**

JoãoAndréMartins；Jisha Abubaker；曾荫权 Mike Eltsufin；Artem Bilan; 安德烈亚斯·伯格（Andreas Berger）Balint Pato; 赵成元; 德米特里·索洛玛卡（Dmitry Solomakha）; 艾琳娜·费尔德（Elena Felder）邹丹妮

## **151.引言**

Spring Cloud GCP项目使Spring Framework成为Google Cloud Platform（GCP）的一等公民。

Spring Cloud GCP使您可以利用Spring Framework的强大功能和简单性来：

1. 使用Google Cloud Vision分析图像中的文本，对象和其他内容
2. 通过Google Cloud IAP使用Spring Security
3. 使用Spring Data Cloud Spanner和Spring Data Cloud Datastore映射对象，关系和集合
4. 发布和订阅Google Cloud Pub / Sub主题
5. 使用一些属性配置Spring JDBC以使用Google Cloud SQL
6. 写入和读取Spring由Google Cloud Storage备份的资源
7. 在后台使用Google Cloud Pub / Sub与Spring Integration交换消息
8. 使用Spring Cloud Sleuth和Google Stackdriver Trace跟踪应用的执行情况
9. 使用Spring Cloud Config配置您的应用，并通过Google Runtime Configuration API进行备份
10. 通过Spring Integration GCS通道适配器消费和产生Google Cloud Storage数据

## **152.依赖性管理**

Spring Cloud GCP物料清单（BOM）包含其使用的所有依赖项的版本。

如果您是Maven用户，则将以下内容添加到pom.xml文件中将使您可以不指定任何Spring Cloud GCP依赖版本。相反，您使用的BOM的版本将确定所使用依赖项的版本。

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-gcp-dependencies</artifactId>

<version>{project-version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies></dependencyManagement>

在以下各节中，将假定您正在使用Spring Cloud GCP BOM，并且相关代码段将不包含版本。

Gradle用户可以使用Spring的[依赖项管理插件](https://github.com/spring-gradle-plugins/dependency-management-plugin" \t "https://www.springcloud.cc/_top) Gradle插件来获得相同的BOM体验。为简单起见，本文档其余部分中的Gradle依赖项片段也将省略其版本。

## **153.入门**

有许多可用资源可帮助您尽快了解我们的库。

## **153.1 Spring初始化**

[Spring Initializr](https://start.spring.io/" \t "https://www.springcloud.cc/_top)中有Spring Cloud GCP的三个条目。

### **153.1.1 GCP支持**

GCP支持条目包含对每个Spring Cloud GCP集成的自动配置支持。仅当将其他依赖项添加到类路径时，才启用大多数自动配置代码。

| **Spring Cloud GCP入门** | **所需的依赖项** |
| --- | --- |
| Config | org.springframework.cloud:spring-cloud-gcp-starter-config |
| Cloud Spanner | org.springframework.cloud:spring-cloud-gcp-starter-data-spanner |
| Cloud Datastore | org.springframework.cloud:spring-cloud-gcp-starter-data-datastore |
| Logging | org.springframework.cloud:spring-cloud-gcp-starter-logging |
| SQL - MySql | org.springframework.cloud:spring-cloud-gcp-starter-sql-mysql |
| SQL - PostgreSQL | org.springframework.cloud:spring-cloud-gcp-starter-sql-postgres |
| Trace | org.springframework.cloud:spring-cloud-gcp-starter-trace |
| Vision | org.springframework.cloud:spring-cloud-gcp-starter-vision |
| Security - IAP | org.springframework.cloud:spring-cloud-gcp-starter-security-iap |

### **153.1.2 GCP消息传递**

“ GCP消息传递”条目添加了“ GCP支持”条目和所有必需的依赖项，因此Google Cloud Pub / Sub集成即开即用。

### **153.1.3 GCP存储**

GCP存储条目会添加GCP支持条目和所有必需的依赖项，以便Google Cloud Storage集成可以立即使用。

## **153.2代码示例**

有可用的[代码示例](https://github.com/spring-cloud/spring-cloud-gcp/tree/master/spring-cloud-gcp-samples" \t "https://www.springcloud.cc/_top)演示了我们所有集成的用法。

例如，[Vision API示例](https://github.com/spring-cloud/spring-cloud-gcp/tree/master/spring-cloud-gcp-samples/spring-cloud-gcp-vision-api-sample" \t "https://www.springcloud.cc/_top)显示了如何使用spring-cloud-gcp-starter-vision自动配置Vision API客户端。

## **153.3代码挑战**

在代码挑战中，您将使用一个集成逐步执行任务。[Google Developers Codelabs](https://codelabs.developers.google.com/spring" \t "https://www.springcloud.cc/_top)页面中存在许多挑战。

## **153.4入门指南**

A Spring入门与Spring Integration通道适配器对于谷歌云发布/订阅消息指南可以在[Spring指南](https://spring.io/guides/gs/messaging-gcp-pubsub/" \t "https://www.springcloud.cc/_top)。

## **154. Spring Cloud GCP核心**

每个Spring Cloud GCP模块都使用GcpProjectIdProvider和CredentialsProvider获取GCP项目ID和访问凭证。

Spring Cloud GCP提供了Spring Boot入门程序来自动配置核心组件。

Maven坐标，使用Spring Cloud GCP BOM：

<dependency>

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

<artifactId>spring-cloud-gcp-starter</artifactId></dependency>

Gradle坐标：

dependencies {

compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter'

}

## **154.1项目ID**

GcpProjectIdProvider是返回GCP项目ID字符串的功能接口。

**public** **interface** GcpProjectIdProvider {

String getProjectId();

}

Spring Cloud GCP启动器会自动配置GcpProjectIdProvider。如果指定了spring.cloud.gcp.project-id属性，则提供的GcpProjectIdProvider返回该属性值。

spring.cloud.gcp.project-id=my-gcp-project-id

否则，将根据[规则](https://googlecloudplatform.github.io/google-cloud-java/google-cloud-clients/apidocs/com/google/cloud/ServiceOptions.html" \l "getDefaultProjectId--" \t "https://www.springcloud.cc/_top)的[有序列表](https://googlecloudplatform.github.io/google-cloud-java/google-cloud-clients/apidocs/com/google/cloud/ServiceOptions.html" \l "getDefaultProjectId--" \t "https://www.springcloud.cc/_top)来发现项目ID ：

1. GOOGLE\_CLOUD\_PROJECT环境变量指定的项目ID
2. Google App Engine项目ID
3. 由GOOGLE\_APPLICATION\_CREDENTIALS环境变量指向的JSON凭证文件中指定的项目ID
4. Google Cloud SDK项目ID
5. 来自Google Compute Engine元数据服务器的Google Compute Engine项目ID

## **154.2凭证**

CredentialsProvider是一个功能接口，可返回凭据以认证和授权对Google Cloud Client库的调用。

**public** **interface** CredentialsProvider {

Credentials getCredentials() **throws** IOException;

}

Spring Cloud GCP启动器会自动配置CredentialsProvider。它使用spring.cloud.gcp.credentials.location属性找到Google服务帐户的OAuth2私钥。请记住，此属性是Spring资源，因此可以从许多[不同的位置（](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/resources.html" \l "resources-implementations" \t "https://www.springcloud.cc/_top)例如文件系统，类路径，URL等）获取凭证文件。下一个示例在文件系统中指定凭证位置属性。

spring.cloud.gcp.credentials.location=file:/usr/local/key.json

或者，您可以通过直接指定spring.cloud.gcp.credentials.encoded-key属性来设置凭据。该值应为JSON格式的base64编码的帐户私钥。

如果未通过属性指定凭据，则启动程序将尝试从[许多地方](https://github.com/GoogleCloudPlatform/google-cloud-java" \l "authentication" \t "https://www.springcloud.cc/_top)发现凭据：

1. GOOGLE\_APPLICATION\_CREDENTIALS环境变量指向的凭据文件
2. Google Cloud SDK gcloud auth application-default login命令提供的凭据
3. Google App Engine内置凭据
4. Google Cloud Shell内置凭据
5. Google Compute Engine内置凭据

如果您的应用程序在Google App Engine或Google Compute Engine上运行，则在大多数情况下，您应该省略spring.cloud.gcp.credentials.location属性，而应让Spring Cloud GCP Starter获得那些环境的正确凭据。在App Engine Standard上，使用[App Identity服务帐户凭据](https://cloud.google.com/appengine/docs/standard/java/appidentity/" \t "https://www.springcloud.cc/_top)；在App Engine Flexible上，使用[Flexible服务帐户凭据](https://cloud.google.com/appengine/docs/flexible/java/service-account" \t "https://www.springcloud.cc/_top)；在Google Compute Engine上，使用[Compute Engine默认服务帐户](https://cloud.google.com/compute/docs/access/create-enable-service-accounts-for-instances" \l "using_the_compute_engine_default_service_account" \t "https://www.springcloud.cc/_top)。

### **154.2.1范围**

默认情况下，Spring Cloud GCP Starter提供的凭据包含Spring Cloud GCP支持的每个服务的范围。

|  |  |
| --- | --- |
| Service | Scope |
| Spanner | [https://www.googleapis.com/auth/spanner.admin](https://www.googleapis.com/auth/spanner.admin" \t "https://www.springcloud.cc/_top), [https://www.googleapis.com/auth/spanner.data](https://www.googleapis.com/auth/spanner.data" \t "https://www.springcloud.cc/_top) |
| Datastore | [https://www.googleapis.com/auth/datastore](https://www.googleapis.com/auth/datastore" \t "https://www.springcloud.cc/_top) |
| Pub/Sub | [https://www.googleapis.com/auth/pubsub](https://www.googleapis.com/auth/pubsub" \t "https://www.springcloud.cc/_top) |
| Storage (Read Only) | [https://www.googleapis.com/auth/devstorage.read\_only](https://www.googleapis.com/auth/devstorage.read_only" \t "https://www.springcloud.cc/_top) |
| Storage (Write/Write) | [https://www.googleapis.com/auth/devstorage.read\_write](https://www.googleapis.com/auth/devstorage.read_write" \t "https://www.springcloud.cc/_top) |
| Runtime Config | [https://www.googleapis.com/auth/cloudruntimeconfig](https://www.googleapis.com/auth/cloudruntimeconfig" \t "https://www.springcloud.cc/_top) |
| Trace (Append) | [https://www.googleapis.com/auth/trace.append](https://www.googleapis.com/auth/trace.append" \t "https://www.springcloud.cc/_top) |
| Cloud Platform | [https://www.googleapis.com/auth/cloud-platform](https://www.googleapis.com/auth/cloud-platform" \t "https://www.springcloud.cc/_top) |
| Vision | [https://www.googleapis.com/auth/cloud-vision](https://www.googleapis.com/auth/cloud-vision" \t "https://www.springcloud.cc/_top) |

通过Spring Cloud GCP启动程序，您可以为提供的凭据配置自定义范围列表。为此，请在spring.cloud.gcp.credentials.scopes属性中指定逗号分隔的[Google OAuth2作用域](https://developers.google.com/identity/protocols/googlescopes" \t "https://www.springcloud.cc/_top)列表。

spring.cloud.gcp.credentials.scopes是Google Cloud Platform服务的[Google OAuth2范围](https://developers.google.com/identity/protocols/googlescopes" \t "https://www.springcloud.cc/_top)的逗号分隔列表，提供的CredentialsProvider支持返回的凭据。

spring.cloud.gcp.credentials.scopes=https://www.googleapis.com/auth/pubsub,https://www.googleapis.com/auth/sqlservice.admin

您还可以使用DEFAULT\_SCOPES占位符作为范围来表示启动程序的默认范围，并附加需要添加的其他范围。

spring.cloud.gcp.credentials.scopes=DEFAULT\_SCOPES,https://www.googleapis.com/auth/cloud-vision

## **154.3环境**

GcpEnvironmentProvider是由Spring Cloud GCP启动程序自动配置的功能接口，它返回GcpEnvironment枚举。提供者可以通过编程帮助确定在哪个GCP环境（App Engine Flexible，App Engine Standard，Kubernetes Engine或Compute Engine）中部署应用程序。

**public** **interface** GcpEnvironmentProvider {

GcpEnvironment getCurrentEnvironment();

}

## **154.4 Spring初始化**

可从[Spring Initializr](https://start.spring.io/" \t "https://www.springcloud.cc/_top)到GCP Support条目使用此启动器。

## **155.Google Cloud Pub / Sub**

Spring Cloud GCP提供了一个抽象层，用于发布到Google Cloud Pub / Sub主题和从中订阅，以及创建，列出或删除Google Cloud Pub / Sub主题和订阅。

提供了Spring Boot入门程序来自动配置各种必需的Pub / Sub组件。

Maven坐标，使用Spring Cloud GCP BOM：

<dependency>

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

<artifactId>spring-cloud-gcp-starter-pubsub</artifactId>