Spring Cloud Netflix

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

Spring cloud 项目为Spring Boot应用提供Netflix OSS集成，并能够自动配置spring环境以及提供其他spring编程特性。只需进行一些简单的注解，你就可以利用久经考验的Netflix组件快速在应用中启用通用的模式，构建大型分布式系统。这些模式包括：服务发现（Eureka），断路器（Hystrix），智能路由（Zuul）和客户端侧的负载均衡（Ribbon）

Service Discovery: Eureka Clients

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand configure each client or some form of convention can be very difficult to do and can be very brittle. Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

服务发现：Eureka Clients

服务发现是微服务架构的关键宗旨之一。手动配置每个客户端或者某种形式的约定是非常困难的，也导致系统非常的脆弱。Eureka是Netflix的服务发现组件，通过配置部署Eureka集群，服务端之间的相互备份提供高可用性。

Registering with Eureka

When a client registers with Eureka, it provides meta-data about itself such as host and port, health indicator URL, home page etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

通过Eureka注册服务

Eureka客户端（服务提供者和服务消费者都是客户端，服务提供者向Eureka注册服务，而消费者向Eureka获取服务）向Eueka注册服务时，提供关于自身的一些元数据信息，比如主机名和端口、健康检查url、主页等。Eureka接收来自服务实例的心跳消息，若在超过指定的时间间隔未收到心跳消息，Eureka将该服务从注册表中移除。

Example eureka client:

Eureka客户端的小栗子

@Configuration

@ComponentScan

@EnableAutoConfiguration

@EnableEurekaClient

@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);

}

}

(i.e. utterly normal Spring Boot app). （真的就是一个绝对普通的Spring Boot应用，ps：@Configuration @ComponentScan @EnableAutoConfiguration 也可以用@SpringBootApplication 一个注解来代替）

In this example we use @EnableEurekaClient explicitly, but with only Eureka available you could also use @EnableDiscoveryClient. Configuration is required to locate the Eureka server.

在这个栗子中，我们明确的使用 @EnableEurekaClient，但也可以使用@EnableDiscoveryClient（@EnableEurekaClient注解包含@EnableDiscoveryClient，但其实基本一样，建议使用@EnableDiscoveryClient）。为了能够定位到Eueka服务端，需要进一些配置，如下：

Example:（栗子栗子栗子）

*application.yml*

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

where "defaultZone" is a magic string fallback value that provides the service URL for any client that doesn’t express a preference (i.e. it’s a useful default).

"defaultZone"配置指定Eureka服务端地址（使用逗号分隔多个，集群配置方式）

The default application name (service ID), virtual host and non-secure port, taken from the Environment, are${spring.application.name}, ${spring.application.name} and ${server.port} respectively.

可以通过使用${spring.application.name}, ${spring.application.name} and ${server.port} ，设置默认的应用名（服务id，基本上应用=服务）、虚拟机和端口

@EnableEurekaClient makes the app into both a Eureka "instance" (i.e. it registers itself) and a "client" (i.e. it can query the registry to locate other services). The instance behaviour is driven by eureka.instance.\* configuration keys, but the defaults will be fine if you ensure that your application has a spring.application.name (this is the default for the Eureka service ID, or VIP).

See [EurekaInstanceConfigBean](http://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) and [EurekaClientConfigBean](http://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java) for more details of the configurable options.

使用@EnableEurekaClient让成为Eureka客户端实例（注册自身作为服务，并能够通过Eureka服务端查询定位其他服务）。可以在配置文件中通过配置eureka.instance.\*属性来指定实例名，但不设置也是可以使用默认的（默认使用spring.application.name中的值）。更多关于配置的信息请参考 [EurekaInstanceConfigBean](http://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) and [EurekaClientConfigBean](http://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java)

Authenticating with the Eureka Server

HTTP basic authentication will be automatically added to your eureka client if one of theeureka.client.serviceUrl.defaultZone URLs has credentials embedded in it (curl style, like<http://user:password@localhost:8761/eureka>). For more complex needs you can create a @Bean of type DiscoveryClientOptionalArgs and inject ClientFilter instances into it, all of which will be applied to the calls from the client to the server.

|  |  |
| --- | --- |
| **NOTE** | Because of a limitation in Eureka it isn’t possible to support per-server basic auth credentials, so only the first set that are found will be used. |

Eureka服务端校验

按照如下方式设置HTTP的基本认证会自动添加到Eureka客户端：eureka.client.serviceUrl.defaultZone属性中的URLs（集群，多个）设置认证信息格式为：<http://user:password@localhost:8761/eureka>。若需要更加复杂的认证，你也可以创建一个带ClientFilter 注解的DiscoveryClientOptionalArgs 类实例。当Eureka客户端向服务端请求时，该实例将会起作用。

注意：由于Eureka的限制，不支持不同服务端不同认证方式，仅使用最先发现的认证方式

Status Page and Health Indicator

The status page and health indicators for a Eureka instance default to "/info" and "/health" respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (e.g. server.servletPath=/foo) or management endpoint path (e.g. management.contextPath=/admin). Example:

*application.yml*

eureka:

instance:

statusPageUrlPath: ${management.context-path}/info

healthCheckUrlPath: ${management.context-path}/health

These links show up in the metadata that is consumed by clients, and used in some scenarios to decide whether to send requests to your application, so it’s helpful if they are accurate.

状态页面和健康检查

浏览器中输入<http://服务客户端域名/info> 和[http:// 服务客户端域名/health](http://Eureka-server/health)可以查看当前服务状态和健康信息页面。这是spring boot默认提供的。如果你需要改变访问路径，则需要修改以下配置：

server.servletPath=/foo 或者management.contextPath=/admin

然后在eureka.instance配置如下：

*application.yml*

eureka:

instance:

statusPageUrlPath: ${management.context-path}/info healthCheckUrlPath: ${management.context-path}/health

Registering a Secure Application

If your app wants to be contacted over HTTPS you can set two flags in the EurekaInstanceConfig, *viz*eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true] respectively. This will make Eureka publish instance information showing an explicit preference for secure communication. The Spring CloudDiscoveryClient will always return an https://…​; URI for a service configured this way, and the Eureka (native) instance information will have a secure health check URL.

注册安全应用（HTTPS）

如果你的应用是基于https协议通信，你需要设置两个标志：

eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true]（我理解就是设置其中一个就可以了，比如设置eureka.instance. securePortEnabled=true）。Spring Cloud发现服务总是会返回https的地址。

Because of the way Eureka works internally, it will still publish a non-secure URL for status and home page unless you also override those explicitly. You can use placeholders to configure the eureka instance urls, e.g.

但是由于eureka的工作原理，状态和主页仍然是以非安全http形式发布，除非你显式的去指定为https，如下：

*application.yml*

eureka:

instance:

statusPageUrl: https://${eureka.hostname}/info

healthCheckUrl: https://${eureka.hostname}/health

homePageUrl: https://${eureka.hostname}/

(Note that ${eureka.hostname} is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well, e.g. using ${eureka.instance.hostName}.)

（注意，${eureka.hostname}是后面版本的eureka才有，也可以使用spring提供的${eureka.instance.hostName}（建议））

|  |  |
| --- | --- |
| **NOTE** | If your app is running behind a proxy, and the SSL termination is in the proxy (e.g. if you run in Cloud Foundry or other platforms as a service) then you will need to ensure that the proxy "forwarded" headers are intercepted and handled by the application. An embedded Tomcat container in a Spring Boot app does this automatically if it has explicit configuration for the 'X-Forwarded-\\*` headers. A sign that you got this wrong will be that the links rendered by your app to itself will be wrong (the wrong host, port or protocol). |

注意：如果你的应用是运行在SSL终端代理（例如作为服务运行在Cloud Foundry或其他平摊）后面，那你需要确认代理的"forwarded"头设置被应用的替换处理。如果有显式的'X-Forwarded-\\*`头设置，Spring Cloud的内嵌tomcat会自动完成这件事情。如果应用到自身的链接是错误的（错误的域名或者端口或者协议），则表明ssl设置有误。

Eureka’s Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise the Discovery Client will not propagate the current health check status of the application per the Spring Boot Actuator. Which means that after successful registration Eureka will always announce that the application is in 'UP' state. This behaviour can be altered by enabling Eureka health checks, which results in propagating application status to Eureka. As a consequence every other application won’t be sending traffic to application in state other then 'UP'.

Eureka健康检查

默认的情况下，eureka利用客户端心跳机制来确认客户端是否在线。除非特别指明，否则eureka客户端不会传递当前健康检查状态。也就是说，成功注册服务后，eureka总是认为客户端是在线的状态。可以通过启用eureka健康检查来改变默认行为，这样eureka客户端就会传递自身的健康信息到eureka服务端。结果其他的服务就不会请求到不是up状态的eureka客户端。（也就是流量控制，某些客户端流量过大处理不过来，将自身状态设置为繁忙之类的，后续请求将不会过来）

*application.yml*

eureka:

client:

healthcheck:

enabled: true

If you require more control over the health checks, you may consider implementing your owncom.netflix.appinfo.HealthCheckHandler.

如果需要更进一步对健康检查的控制，可以实现自己的com.netflix.appinfo.HealthCheckHandler类

Eureka Metadata for Instances and Clients

It’s worth spending a bit of time understanding how the Eureka metadata works, so you can use it in a way that makes sense in your platform. There is standard metadata for things like hostname, IP address, port numbers, status page and health check. These are published in the service registry and used by clients to contact the services in a straightforward way. Additional metadata can be added to the instance registration in the eureka.instance.metadataMap, and this will be accessible in the remote clients, but in general will not change the behaviour of the client, unless it is made aware of the meaning of the metadata. There are a couple of special cases described below where Spring Cloud already assigns meaning to the metadata map.

Eureka实例和客户端的元数据

花点时间去了解eureka元数据的工作原理是值得的，这样就可以在你的平台更好的使用eureka。Eureka为像主机名、ip、端口、状态页、健康检查等指定标准的元数据。这些都保存在服务注册表中，方便eureka客户端更直接获取服务。通过设置eureka.instance.metadataMap可以添加额外的元数据信息，这些信息也都可以在任意其他eureka客户端获取使用，但不会改变其他客户端的行为，除非其他客户端设置为可感知元数据。下面有一些例子来说明元数据映射（metadata map）在spring cloud中的意义。

Using Eureka on Cloudfoundry

在Cloudfoundry中使用eureka——不翻译了

Cloudfoundry has a global router so that all instances of the same app have the same hostname (it’s the same in other PaaS solutions with a similar architecture). This isn’t necessarily a barrier to using Eureka, but if you use the router (recommended, or even mandatory depending on the way your platform was set up), you need to explicitly set the hostname and port numbers (secure or non-secure) so that they use the router. You might also want to use instance metadata so you can distinguish between the instances on the client (e.g. in a custom load balancer). By default, theeureka.instance.instanceId is vcap.application.instance\_id. For example:

*application.yml*

eureka:

instance:

hostname: ${vcap.application.uris[0]}

nonSecurePort: 80

Depending on the way the security rules are set up in your Cloudfoundry instance, you might be able to register and use the IP address of the host VM for direct service-to-service calls. This feature is not (yet) available on Pivotal Web Services ([PWS](https://run.pivotal.io/)).

Using Eureka on AWS

在AWS中使用eureka——不翻译了

If the application is planned to be deployed to an AWS cloud, then the Eureka instance will have to be configured to be Amazon aware and this can be done by customizing the [EurekaInstanceConfigBean](http://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) the following way:

@Bean

@Profile("!default")

public EurekaInstanceConfigBean eurekaInstanceConfig() {

EurekaInstanceConfigBean b = new EurekaInstanceConfigBean();

AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");

b.setDataCenterInfo(info);

return b;

}

Changing the Eureka Instance ID

A vanilla Netflix Eureka instance is registered with an ID that is equal to its host name (i.e. only one service per host). Spring Cloud Eureka provides a sensible default that looks like this:${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}}. For example myhost:myappname:8080.

改变eureka实例id

通常情况下eureka实例id格式是：

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}.

比如myhost:myappname:8080.

注意，原文后面多了个} 。还有最后的${spring.application.instance\_id:${server.port}}表示若spring.application.instance\_id变量不存在，则取server.port，有点类似?:运算符。

Using Spring Cloud you can override this by providing a unique identifier in eureka.instance.instanceId. For example:

可以在配置文件中指定特定的instanceId，如下

*application.yml*

eureka:

instance:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

With this metadata, and multiple service instances deployed on localhost, the random value will kick in there to make the instance unique. In Cloudfoundry the vcap.application.instance\_id will be populated automatically in a Spring Boot application, so the random value will not be needed.

这样就可以在本地起多个相同的服务。

Using the EurekaClient

Once you have an app that is @EnableDiscoveryClient (or @EnableEurekaClient) you can use it to discover service instances from the [Eureka Server](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-eureka-server). One way to do that is to use the native com.netflix.discovery.EurekaClient (as opposed to the Spring Cloud DiscoveryClient), e.g.

使用eureka客户端

通过 @EnableDiscoveryClient或@EnableEurekaClient注解你就可以声明一个eureka客户端实例，去获取其他服务。一种获取服务的方式是使用 com.netflix.discovery.EurekaClient对象（注解后spring cloud会曝露该对象，直接注入即可使用）。

@Autowired

private EurekaClient discoveryClient;

public String serviceUrl() {

InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);

return instance.getHomePageUrl();

}

|  |  |
| --- | --- |
| **TIP** | Don’t use the EurekaClient in @PostConstruct method or in a @Scheduled method (or anywhere where the ApplicationContext might not be started yet). It is initialized in a SmartLifecycle (withphase=0) so the earliest you can rely on it being available is in another SmartLifecycle with higher phase. |

注意：EurekaClient 的注解不能在有@PostConstruct 或@Scheduled 的方法上（或者其他ApplicationContext 还没初始化的方法上）

Alternatives to the native Netflix EurekaClient

You don’t have to use the raw Netflix EurekaClient and usually it is more convenient to use it behind a wrapper of some sort. Spring Cloud has support for [Feign](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-feign) (a REST client builder) and also [Spring RestTemplate](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon) using the logical Eureka service identifiers (VIPs) instead of physical URLs. To configure Ribbon with a fixed list of physical servers you can simply set <client>.ribbon.listOfServers to a comma-separated list of physical addresses (or hostnames), where <client>is the ID of the client.

You can also use the org.springframework.cloud.client.discovery.DiscoveryClient which provides a simple API for discovery clients that is not specific to Netflix, e.g.

EurekaClient对象原生服务的替代方案

你也可以不使用netflix的EurekaClient 原生服务来获取服务等。实际上spring cloud提供了更方便的封装支持，比如[Feign](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-feign)（rest风格的服务请求），或者[Spring RestTemplate](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon) ，可以通过服务id（也可以是url）来调用服务。为ribbon（负载均衡）配置固定的服务器列表，只需要简单的在配置文件中设置 <client>.ribbon.listOfServers，指定服务器的服务地址（url或者域名）。其中<client>是当前eureka客户端的id（通常是 spring.application.name的值）

也可以注入DiscoveryClient对象，这个是spring cloud提供的不仅为netflix实现的通用服务发现客户端api，栗子如下：

@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;

}

Why is it so Slow to Register a Service?

Being an instance also involves a periodic heartbeat to the registry (via the client’s serviceUrl) with default duration 30 seconds. A service is not available for discovery by clients until the instance, the server and the client all have the same metadata in their local cache (so it could take 3 heartbeats). You can change the period usingeureka.instance.leaseRenewalIntervalInSeconds and this will speed up the process of getting clients connected to other services. In production it’s probably better to stick with the default because there are some computations internally in the server that make assumptions about the lease renewal period.

为什么注册服务如此慢

注册为服务实例，同时也涉及到周期性的心跳机制（通过serviceUrl请求），默认持续30秒。在实例、服务端、客户端的本地缓存元数据信息一致之前，服务发现是不可用的（所以需要3次心跳确认，可能类似3次握手，保证客户端和服务端的元数据一致）。可以通过eureka.instance.leaseRenewalIntervalInSeconds改变心跳间隔，这将加快客户端连接到其他服务的过程。在生产环境中，可能默认的设置更好，因为服务端内部需要假定的一定的处理时间

Service Discovery: Eureka Server

Example eureka server (e.g. using spring-cloud-starter-eureka-server to set up the classpath):

服务发现：eureka服务端

eureka服务端栗子，设置spring-cloud-starter-eureka-server依赖：

@SpringBootApplication

@EnableEurekaServer

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

The server has a home page with a UI, and HTTP API endpoints per the normal Eureka functionality under /eureka/\*.

Eureka background reading: see [flux capacitor](https://github.com/cfregly/fluxcapacitor/wiki/NetflixOSS-FAQ#eureka-service-discovery-load-balancer) and [google group discussion](https://groups.google.com/forum/?fromgroups#!topic/eureka_netflix/g3p2r7gHnN0).

访问eureka服务的/eureka/\* 可以访问主页和api页面。

|  |  |
| --- | --- |
| **TIP** | Due to Gradle’s dependency resolution rules and the lack of a parent bom feature, simply depending on spring-cloud-starter-eureka-server can cause failures on application startup. To remedy this the Spring Boot Gradle plugin must be added and the Spring cloud starter parent bom must be imported like so:  由于gradle的依赖规则，缺少父pom依赖特性，仅依赖spring-cloud-starter-eureka-server应用启动时可能会引起一些问题。要解决这个问题需要添加Spring Boot Gradle plugin插件，并需要引入Spring cloud starter parent pom：  *build.gradle*  buildscript {  dependencies {  classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.5.RELEASE")  }  }  apply plugin: "spring-boot"  dependencyManagement {  imports {  mavenBom "org.springframework.cloud:spring-cloud-dependencies:Brixton.RELEASE"  }  } |

High Availability, Zones and Regions

The Eureka server does not have a backend store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of eureka registrations (so they don’t have to go to the registry for every single request to a service).

By default every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you don’t provide it the service will run and work, but it will shower your logs with a lot of noise about not being able to register with the peer.

See also [below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon) on the client side for Zones and Regions.

高可用特性，区块和区域

Eureka服务端没有后端存储，但是注册表中的服务实例都必须发送心跳消息来保证注册信息更新（在内存中进行）。客户端同时也保留有本地的注册表缓存（这就不需要每次请求都到服务端获取服务信息）。

默认每个eureka服务端也是eureka客户端，需要（至少）服务端url来定位其他等价服务端（也就是eureka服务端集群中的其他服务端——peer）。如果没有提供其他服务端的url，也能正常使用，只是不断的日志输出无法注册到peer的信息（若只有一个服务端，则设置serviceUrl.defaultZone为自身，如[http://discovery:${server.port}/eureka/](http://discovery:$%7bserver.port%7d/eureka/)）。

请阅 [below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon)客户端中ribbon对区块和区域的支持。

Standalone Mode

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime keeping it alive (e.g. Cloud Foundry). In standalone mode, you might prefer to switch off the client side behaviour, so it doesn’t keep trying and failing to reach its peers. Example:

单例模式

客户端和服务端注册表缓存的结合，以及心跳机制使得单例服务端能够相当弹性应对错误。同时还有其他的监视器和弹性机制保证单例服务端可用（比如cloud foundry）。在单例模式中，更好的做法是关闭客户端行为（也就是单例模式中服务端仅作为服务端，不要同时也声明为客户端），这样就不会尝试去连接其他（不存在的）服务端。例如：

*application.yml (Standalone Eureka Server)*

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

Notice that the serviceUrl is pointing to the same host as the local instance.

注意：serviceUrl 指向自身

Peer Awareness

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behaviour, so all you need to do to make it work is add a valid serviceUrl to a peer, e.g.

同位端感知

Eureka在多实例下更有弹性和可靠，多个服务端实例相互注册。这也是默认的行为，因此需要配置serviceUrl 来达到集群效果，如：

*application.yml (Two Peer Aware Eureka Servers)*

---

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/

In this example we have a YAML file that can be used to run the same server on 2 hosts (peer1 and peer2), by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there’s not much value in doing that in production) by manipulating /etc/hosts to resolve the host names. In fact, theeureka.instance.hostname is not needed if you are running on a machine that knows its own hostname (it is looked up using java.net.InetAddress by default).

在这个栗子中，YAML文件配置了两个服务端（peer1和peer2），运行时通过指定profile来启动不同的peer。通过该配置然后修改/etc/hosts设置两个host可在同一个主机中测试同位感知（生产环境通常不会这么做）。实际若主机可以获知自身的主机名，eureka.instance.hostname不需要特别指定（默认spring cloud会自动从java.net.InetAddress获取）

You can add multiple peers to a system, and as long as they are all connected to each other by at least one edge, they will synchronize the registrations amongst themselves. If the peers are physically separated (inside a data centre or between multiple data centres) then the system can in principle survive split-brain type failures.

可以添加多个同位端，不同端之间相互感知连接，这样同位端之间将同步服务注册表。如果同位端被物理隔离（一个或多个数据中心之间），则系统也可以使用。

Prefer IP Address

In some cases, it is preferable for Eureka to advertise the IP Adresses of services rather than the hostname. Seteureka.instance.preferIpAddress to true and when the application registers with eureka, it will use its IP Address rather than its hostname.

Ip地址更好

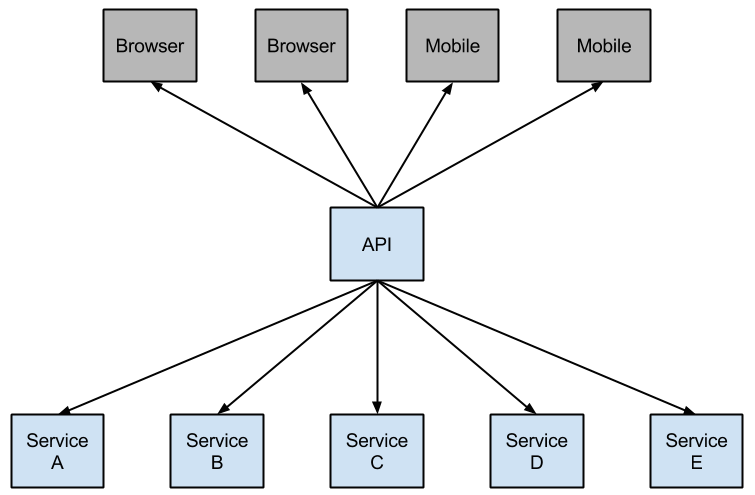
在一些情况下，更建议使用ip地址而不是域名。设置teureka.instance.preferIpAddress为true，则在应用注册到eureka后，将使用ip代替域名进行服务之间的连接。

Circuit Breaker: Hystrix Clients

Netflix has created a library called [Hystrix](https://github.com/Netflix/Hystrix) that implements the [circuit breaker pattern](http://martinfowler.com/bliki/CircuitBreaker.html). In a microservice architecture it is common to have multiple layers of service calls.

断路器：Hystrix客户端

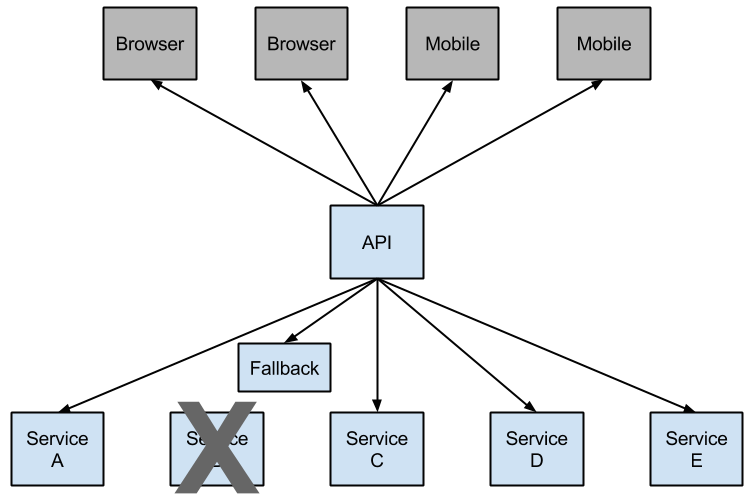
Netflix创建了一个Hystrix的库来实现断路器模式。在微服务架构中，多层服务调用是非常常见的。



*Figure 1. Microservice Graph*

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service reach a certain threshold (20 failures in 5 seconds is the default in Hystrix), the circuit opens and the call is not made. In cases of error and an open circuit a fallback can be provided by the developer.

低层次的服务失败总是会传导到用户。当调用某个特定的服务失败此时达到了一个阈值（Hystrix是5秒20次），断路将开启，调用失败。为了处理错误，开发者可提供断路回调处理。



*Figure 2. Hystrix fallback prevents cascading failures*

Having an open circuit stops cascading failures and allows overwhelmed or failing services time to heal. The fallback can be another Hystrix protected call, static data or a sane empty value. Fallbacks may be chained so the first fallback makes some other business call which in turn falls back to static data.

开启断路器防止失败传导，并给充分的时间处理失败的服务。失败回调可以是方法、静态数据或者空值。回调可能是串联的，第一个回调可能进行一些是业务逻辑处理，该处理也可能错误而导致回调，并存储到静态数据中。

Example boot app:

@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 \*/;

}

}

The @HystrixCommand is provided by a Netflix contrib library called ["javanica"](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica). Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit, and what to do in case of a failure.

To configure the @HystrixCommand you can use the commandProperties attribute with a list of @HystrixPropertyannotations. See [here](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration) for more details. See the [Hystrix wiki](https://github.com/Netflix/Hystrix/wiki/Configuration) for details on the properties available.

 Netflix的@HystrixCommand注解贡献了["javanica"](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica)库。Spring cloud自动封装加了该注解的bean成代理，该代理会连接到Hystrix断路器。断路器根据情况确定什么时候打开回路什么时候关闭回路，以及出现失败后如何处理。

@HystrixCommand注解中你可以使用commandProperties 属性去设置一些特性，例如

@HystrixCommand(commandProperties = {

@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "500")

})

Propagating the Security Context or using Spring Scopes

If you want some thread local context to propagate into a @HystrixCommand the default declaration will not work because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller using some configuration, or directly in the annotation, by asking it to use a different "Isolation Strategy". For example:

传导安全上下文或者使用spring范围

如果你希望将本地线程的上下文传导到@HystrixCommand中，默认的设置是不行的。因为HystrixCommand方法会在线程池中执行（以防超时）。可以设置Hystrix和调用者使用同一个线程，或者直接将在注解中声明不同的隔离策略，例如：

@HystrixCommand(fallbackMethod = "stubMyService",

commandProperties = {

@HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")

}

)

...

The same thing applies if you are using @SessionScope or @RequestScope. You will know when you need to do this because of a runtime exception that says it can’t find the scoped context.

使用@SessionScope or @RequestScope也可以完成同样的事情。如果抛出运行时异常提示无法找到上下文，你就知道需要使用这些注解了。

You also have the option to set the hystrix.shareSecurityContext property to true. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the SecurityContext from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple hystrix concurrency strategy to be registered so an extension mechanism is available by declaring your own HystrixConcurrencyStrategy as a Spring bean. Spring Cloud will lookup for your implementation within the Spring context and wrap it inside its own plugin.

你也可以选择设置hystrix.shareSecurityContext为true，这样会自动设置Hystrix并发策略插件钩子（什么玩意），将SecurityContext 从主线程传递到Hystrix command线程。Hystrix不允许注册多个并发策略插件钩子，所以一种扩展是机制是创建你自己的HystrixConcurrencyStrategy 对象作为spring bean。Spring cloud会在上下文中搜索该对象并装载。

Health Indicator

The state of the connected circuit breakers are also exposed in the /health endpoint of the calling application.

健康指示器

断路器状态同样在调用应用的/health中

{

"hystrix": {

"openCircuitBreakers": [

"StoreIntegration::getStoresByLocationLink"

],

"status": "CIRCUIT\_OPEN"

},

"status": "UP"

}

Hystrix Metrics Stream

To enable the Hystrix metrics stream include a dependency on spring-boot-starter-actuator. This will expose the/hystrix.stream as a management endpoint.

Hystrix度量流

引入依赖spring-boot-starter-actuator就可以启用Hystrix度量流，通过/hystrix.stream可以看到。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

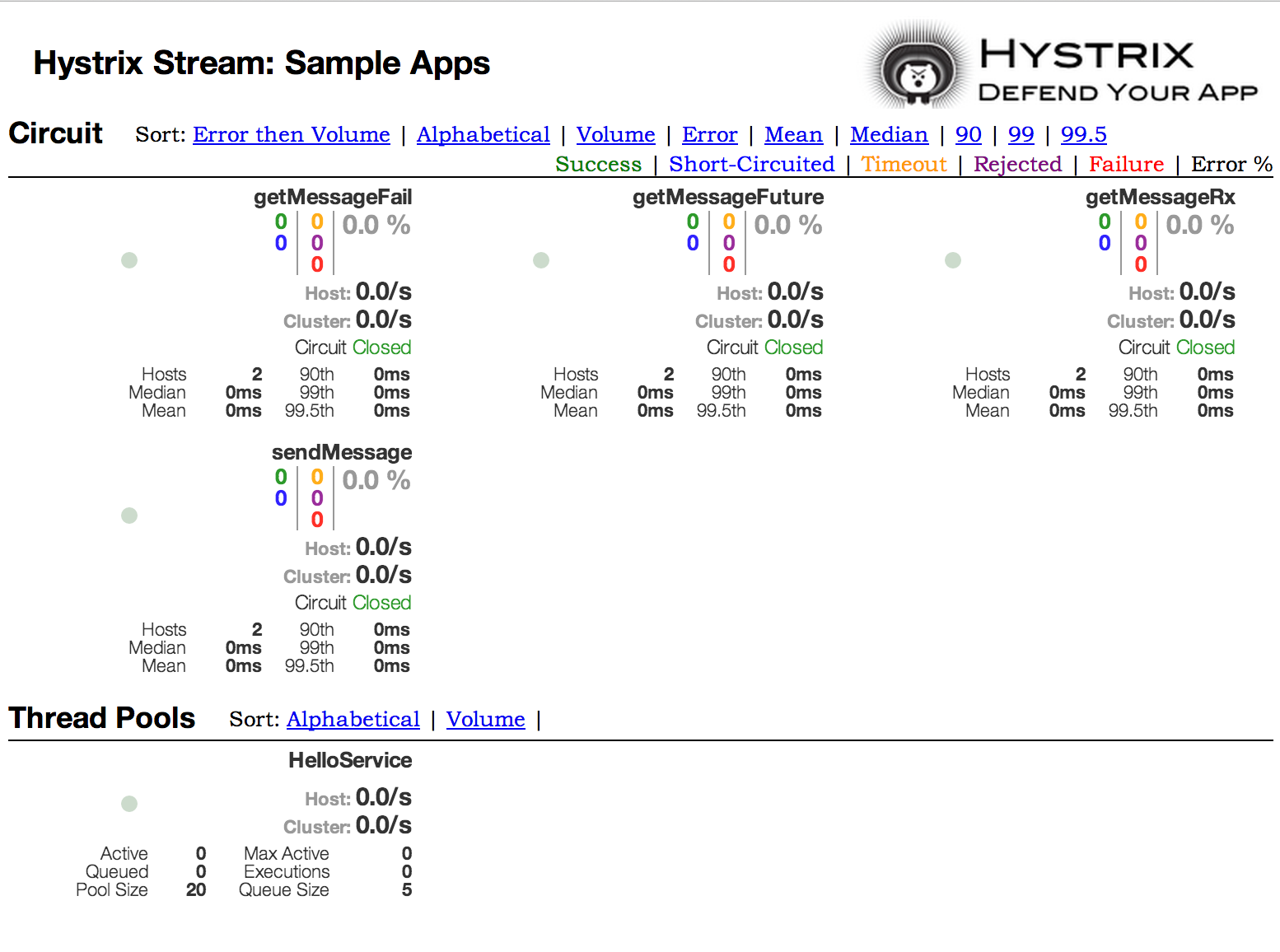
</dependency>

Circuit Breaker: Hystrix Dashboard

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

断路器：hystrix dashboard

Hystrix的一个最大的好处就是聚合了每一个HystrixCommand的度量指标。Hystrix 报表展示了每一个断路器的健康状态。



*Figure 3. Hystrix Dashboard*

To run the Hystrix Dashboard annotate your Spring Boot main class with @EnableHystrixDashboard. You then visit/hystrix and point the dashboard to an individual instances /hystrix.stream endpoint in a Hystrix client application.

启用hystrix报表你需要在应用启动类中加入注解 @EnableHystrixDashboard。通过/hystrix.stream就可以查看相关信息了。

Turbine

Looking at an individual instances Hystrix data is not very useful in terms of the overall health of the system. [Turbine](https://github.com/Netflix/Turbine) is an application that aggregates all of the relevant /hystrix.stream endpoints into a combined /turbine.stream for use in the Hystrix Dashboard. Individual instances are located via Eureka. Running Turbine is as simple as annotating your main class with the @EnableTurbine annotation (e.g. using spring-cloud-starter-turbine to set up the classpath). All of the documented configuration properties from [the Turbine 1 wiki](https://github.com/Netflix/Turbine/wiki/Configuration-(1.x)) apply. The only difference is that theturbine.instanceUrlSuffix does not need the port prepended as this is handled automatically unlessturbine.instanceInsertPort=false.

Turbine

单看一个实例的Hystrix信息对整个系统的健康来说并不是很有意义。Turbine是一个将系统中所有 /hystrix.stream聚合在一起的应用，使用/turbine.stream来展示，实例通过eureka来进行定位。启用turbine也很简单，在应用启动类加入注解 @EnableTurbine （需要加入依赖spring-cloud-starter-turbine）。所有在 [the Turbine 1 wiki](https://github.com/Netflix/Turbine/wiki/Configuration-(1.x))文档中的配置都被应用。不同的是turbine.instanceUrlSuffix属性不需要预先设定端口（会自动处理），除非turbine.instanceInsertPort=false.

The configuration key turbine.appConfig is a list of eureka serviceIds that turbine will use to lookup instances. The turbine stream is then used in the Hystrix dashboard using a url that looks like:[http://my.turbine.sever:8080/turbine.stream?cluster=<CLUSTERNAME>](http://my.turbine.sever:8080/turbine.stream?cluster=%3CCLUSTERNAME%3E); (the cluster parameter can be omitted if the name is "default"). The cluster parameter must match an entry in turbine.aggregator.clusterConfig. Values returned from eureka are uppercase, thus we expect this example to work if there is an app registered with Eureka called "customers":

turbine.appConfig配置项是服务id的集合，turbine会根据这个集合去获取服务实例。Turbine stream的使用格式是：

like:[http://my.turbine.sever:8080/turbine.stream?cluster=<CLUSTERNAME>](http://my.turbine.sever:8080/turbine.stream?cluster=%3CCLUSTERNAME%3E); （若cluster参数值为default，则可忽略）

cluster 参数匹配turbine.aggregator.clusterConfig中设置的值（需要设置为大写）。下面的栗子就是去获取服务id为customers的Hystrix到turbine中。

turbine:

aggregator:

clusterConfig: CUSTOMERS

appConfig: customers

The clusterName can be customized by a SPEL expression in turbine.clusterNameExpression with root an instance of InstanceInfo. The default value is appName, which means that the Eureka serviceId ends up as the cluster key (i.e. the InstanceInfo for customers has an appName of "CUSTOMERS"). A different example would beturbine.clusterNameExpression=aSGName, which would get the cluster name from the AWS ASG name. Another example:

clusterName （也就是url中的cluster=<CLUSTERNAME>）可以通过SPEL表达式进行设置，对应的属性是turbine.clusterNameExpression 。默认值是应用名（spring.application.name），也就是应用注册到eureka的服务id也是turbine cluster的key值。另一个栗子turbine.clusterNameExpression=aSGName,表示clustername将来自AWS ASG name。下面还有一个栗子：

turbine:

aggregator:

clusterConfig: SYSTEM,USER

appConfig: customers,stores,ui,admin

clusterNameExpression: metadata['cluster']

In this case, the cluster name from 4 services is pulled from their metadata map, and is expected to have values that include "SYSTEM" and "USER".

这个栗子中，cluster名是从它的四个服务的元数据中获取的，并匹配含有"SYSTEM" and "USER"的值。

To use the "default" cluster for all apps you need a string literal expression (with single quotes, and escaped with double quotes if it is in YAML as well):

使用字符串表达式，设置为default，对所有应用使用turbine（也就是说，可通过clusterNameExpression和clusterConfig配合达到只聚合部分的应用服务）

turbine:

appConfig: customers,stores

clusterNameExpression: "'default'"

Spring Cloud provides a spring-cloud-starter-turbine that has all the dependencies you need to get a Turbine server running. Just create a Spring Boot application and annotate it with @EnableTurbine.

|  |  |
| --- | --- |
| **NOTE** | by default Spring Cloud allows Turbine to use the host and port to allow multiple processes per host, per cluster. If you want the native Netflix behaviour built into Turbine that does *not* allow multiple processes per host, per cluster (the key to the instance id is the hostname), then set the propertyturbine.combineHostPort=false. |

Spring cloud提供了spring-cloud-starter-turbine依赖，包括了所有的turbine需要的依赖。要启用turbine只需要创建并在启动类中添加@EnableTurbine注解即可。

注意：默认spring cloud允许使用host和端口来在一个主机上起多个turbine实例。但是原生的Netflix不允许这样做，这个时候需要设置turbine.combineHostPort=false

Turbine Stream

In some environments (e.g. in a PaaS setting), the classic Turbine model of pulling metrics from all the distributed Hystrix commands doesn’t work. In that case you might want to have your Hystrix commands push metrics to Turbine, and Spring Cloud enables that with messaging. All you need to do on the client is add a dependency to spring-cloud-netflix-hystrix-stream and the spring-cloud-starter-stream-\* of your choice (see Spring Cloud Stream documentation for details on the brokers, and how to configure the client credentials, but it should work out of the box for a local broker).

Turbine流

在一些环境中（比如paas设置），传统的从分布式hustrix command 中获取urbine度量的模式行不通。在这种情况下，你也许同样希望hustrix command能够将他们的度量指标推送到turbine。Spring cloud使用消息实现了这个功能。你需要做的仅是在客户端中添加 spring-cloud-netflix-hystrix-stream 和spring-cloud-starter-stream-\* 依赖（更多细节可以查阅Spring Cloud Stream的文档）

On the server side Just create a Spring Boot application and annotate it with @EnableTurbineStream and by default it will come up on port 8989 (point your Hystrix dashboard to that port, any path). You can customize the port using eitherserver.port or turbine.stream.port. If you have spring-boot-starter-web and spring-boot-starter-actuator on the classpath as well, then you can open up the Actuator endpoints on a separate port (with Tomcat by default) by providing a management.port which is different.

另一方面，仅是创建spring boot应用并添加 @EnableTurbineStream注解，在默认的情况下将会启动到8989端口（将Hystrix dashboard指定到该端口，任意路径）。可以通过server.port or turbine.stream.port改变该端口。如果同时引入了spring-boot-starter-web and spring-boot-starter-actuator依赖，还能在management.port 指定的不同端口打开指示器。

You can then point the Hystrix Dashboard to the Turbine Stream Server instead of individual Hystrix streams. If Turbine Stream is running on port 8989 on myhost, then put [http://myhost:8989](http://myhost:8989/) in the stream input field in the Hystrix Dashboard. Circuits will be prefixed by their respective serviceId, followed by a dot, then the circuit name.

这样你就可以将Hystrix Dashboard指向到Turbine Stream服务端，而不只是Hystrix streams。假设Turbine Stream运行在myhost的8989端口上，则设置Hystrix Dashboard中的输入流属性为[http://myhost:8989](http://myhost:8989/)回路以他们的服务id作为前缀，后面跟点号，然后是回路名。

Spring Cloud provides a spring-cloud-starter-turbine-stream that has all the dependencies you need to get a Turbine Stream server running - just add the Stream binder of your choice, e.g. spring-cloud-starter-stream-rabbit. You need Java 8 to run the app because it is Netty-based.

Spring cloud提供spring-cloud-starter-turbine-stream给 Turbine Stream服务运行所需要的所有依赖。还需要添加你选择的Stream binder（应该是消息队列），比如 spring-cloud-starter-stream-rabbit。另外需要java 8来运行该应用。

Client Side Load Balancer: Ribbon

Ribbon is a client side load balancer which gives you a lot of control over the behaviour of HTTP and TCP clients. Feign already uses Ribbon, so if you are using @FeignClient then this section also applies.

客户端负载均衡：ribbon

Ribbon是一个客户端侧的负载均衡，对http和tcp客户端进行负载控制。Feign是基于ribbon之上，所以使用@FeignClient也就使用了ribbon。（所谓客户端侧负载均衡也就是在请求的发起端就进行负载均衡选择，而不是一般上的通过代理（nginx等）进行）

A central concept in Ribbon is that of the named client. Each load balancer is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer (e.g. using the @FeignClient annotation). Spring Cloud creates a new ensemble as an ApplicationContexton demand for each named client using RibbonClientConfiguration. This contains (amongst other things) anILoadBalancer, a RestClient, and a ServerListFilter.

Ribbon的一个核心概念是命名客户端。Ribbon均衡器包含一系列的组件集合。开发者通过某种方式给该集合赋予名字（例如使用@FeignClient注解）。使用RibbonClientConfiguration可以替换默认的组件实现。这个组件集合包括ILoadBalancer,  RestClient, 和 ServerListFilter。

Customizing the Ribbon Client

You can configure some bits of a Ribbon client using external properties in <client>.ribbon.\*, which is no different than using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in CommonClientConfigKey (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of theRibbonClientConfiguration) using @RibbonClient. Example:

自定义ribbon客户端

配置文件的<client>.ribbon.\*对ribbon客户端进行一些配置，这和使用原生的netflix api没什么不同。原生api对象通过 CommonClientConfigKey作为静态对象提供。

Spring cloud也能够让你完全控制客户端，只需要添加 @RibbonClient注解，例如：

@Configuration

@RibbonClient(name = "foo", configuration = FooConfiguration.class)

public class TestConfiguration {

}

In this case the client is composed from the components already in RibbonClientConfiguration together with any inFooConfiguration (where the latter generally will override the former).

|  |  |
| --- | --- |
| **WARNING** | The FooConfiguration has to be @Configuration but take care that it is not in a @ComponentScanfor the main application context, otherwise it will be shared by all the @RibbonClients. If you use@ComponentScan (or @SpringBootApplication) you need to take steps to avoid it being included (for instance put it in a separate, non-overlapping package, or specify the packages to scan explicitly in the@ComponentScan). |

在这个栗子中，客户端通过RibbonClientConfiguration 的组件进行曝露，FooConfiguration是自定义的配置类，可以进行自定义的负载均衡控制。

警告：FooConfiguration类必须添加@Configuration注解，但需要注意的是它不能在@ComponentScan自动注入扫描的范围内。否则会被其他@RibbonClient共享（变成全局配置）。如果你使用了@ComponentScan或@SpringBootApplication，则需要将FooConfiguration类放到不被扫到的包中。

Spring Cloud Netflix provides the following beans by default for ribbon (BeanType beanName: ClassName):

* IClientConfig ribbonClientConfig: DefaultClientConfigImpl
* IRule ribbonRule: ZoneAvoidanceRule
* IPing ribbonPing: NoOpPing
* ServerList<Server> ribbonServerList: ConfigurationBasedServerList
* ServerListFilter<Server> ribbonServerListFilter: ZonePreferenceServerListFilter
* ILoadBalancer ribbonLoadBalancer: ZoneAwareLoadBalancer

Spring Cloud Netflix默认会为ribbon提供如下bean：

IClientConfig配置实现类：DefaultClientConfigImpl

IRule ribbon选择服务规则实现类：ZoneAvoidanceRule

IPing服务健康检查确认实现类：NoOpPing

ServerList<Server>服务列表：ConfigurationBasedServerList

ServerListFilter<Server>过滤列表：ZonePreferenceServerListFilter

ILoadBalancer  均衡器ZoneAwareLoadBalancer

（也就是上面这些bean在是默认的实现，可以通过自定义替换这些bean来实现自定义逻辑，这也是ribbon灵活的地方）

Creating a bean of one of those type and placing it in a @RibbonClient configuration (such as FooConfigurationabove) allows you to override each one of the beans described. Example:

FooConfiguration的一个栗子如下：

@Configuration

public class FooConfiguration {

@Bean

public IPing ribbonPing(IClientConfig config) {

return new PingUrl();

}

}

This replaces the NoOpPing with PingUrl.

这里使用PingUrl替换了默认的NoOpPing

Using Ribbon with Eureka

When Eureka is used in conjunction with Ribbon the ribbonServerList is overridden with an extension ofDiscoveryEnabledNIWSServerList which populates the list of servers from Eureka. It also replaces the IPing interface with NIWSDiscoveryPing which delegates to Eureka to determine if a server is up. The ServerList that is installed by default is a DomainExtractingServerList and the purpose of this is to make physical metadata available to the load balancer without using AWS AMI metadata (which is what Netflix relies on). By default the server list will be constructed with "zone" information as provided in the instance metadata (so on the remote clients seteureka.instance.metadataMap.zone), and if that is missing it can use the domain name from the server hostname as a proxy for zone (if the flag approximateZoneFromHostname is set). Once the zone information is available it can be used in a ServerListFilter. By default it will be used to locate a server in the same zone as the client because the default is aZonePreferenceServerListFilter. The zone of the client is determined the same way as the remote instances by default, i.e. via eureka.instance.metadataMap.zone.

Eureka中使用ribbon

Eureka和ribbon结合使用后，ribbonServerList 被重写为DiscoveryEnabledNIWSServerList，从eureka中获取服务列表。同样会使用NIWSDiscoveryPing 替换IPing默认实现，NIWSDiscoveryPing 会向eureka确认服务是否在线。而 ServerList 被DomainExtractingServerList 替换，这主要是为了能够在不使用AWS AMI元数据（Netflix原生依赖）情况下，对负载均衡器来说机器元数据也是可用的。服务端列表默认会在实例信息中添加zone相关的信息（所以在远程客户端设置eureka.instance.metadataMap.zone）。如果没有设置zone，可以用服务端hostname域名作为zone的代理（需要设置approximateZoneFromHostname）。一旦zone信息可用，ServerListFilter中就可以获取到。默认使用ZonePreferenceServerListFilter，被用来在同一个zone中定位服务。客户端的zone配置也是一样的通过eureka.instance.metadataMap.zone进行

|  |  |
| --- | --- |
| **NOTE** | The orthodox "archaius" way to set the client zone is via a configuration property called "@zone", and Spring Cloud will use that in preference to all other settings if it is available (note that the key will have to be quoted in YAML configuration). |
| **NOTE** | If there is no other source of zone data then a guess is made based on the client configuration (as opposed to the instance configuration). We take eureka.client.availabilityZones, which is a map from region name to a list of zones, and pull out the first zone for the instance’s own region (i.e. theeureka.client.region, which defaults to "us-east-1" for comatibility with native Netflix). |

注意：客户端zone的一般设置方式是通过@zone配置项进行，spring cloud优先使用该注解的配置（需要再yaml文件引用该键值）

啊啊啊啊啊，这节太难翻译了，算了

Example: How to Use Ribbon Without Eureka

Eureka is a convenient way to abstract the discovery of remote servers so you don’t have to hard code their URLs in clients, but if you prefer not to use it, Ribbon and Feign are still quite amenable. Suppose you have declared a@RibbonClient for "stores", and Eureka is not in use (and not even on the classpath). The Ribbon client defaults to a configured server list, and you can supply the configuration like this

栗子：eureka之外使用ribbon

Eureka是一种便捷的抽象远端服务的方式，这样你就不用在客户端中硬编码服务url。但是如果你不打算使用eureka，ribbon和feign还是可以正常使用的。假设stores服务声明了@RibbonClient，但是没有使用eureka，ribbon客户端默认可以配置服务端列表，如下：

*application.yml*

stores:

ribbon:

listOfServers: example.com,google.com

Example: Disable Eureka use in Ribbon

Setting the property ribbon.eureka.enabled = false will explicitly disable the use of Eureka in Ribbon.

栗子：ribbon中禁用eureka

*application.yml*

ribbon:

eureka:

enabled: false

Using the Ribbon API Directly

You can also use the LoadBalancerClient directly. Example:

直接使用ribbon api

可以通过注入LoadBalancerClient bean直接使用ribbon api

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

}

}

Declarative REST Client: Feign

[Feign](https://github.com/Netflix/feign) is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the sameHttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

声明式rest客户端：feign

[Feign](https://github.com/Netflix/feign) 是声明式的web服务客户端，让写web服务客户端变得更加的容易。使用feign创建接口并加上相关注解即可。支持插件式的注解支持，包括feign注解和JAX-RS注解。Feign也支持可插入的编解码。Spring cloud为feign提供同spring MVC注解和HttpMessageConverters一样的注解支持。通过feign spring cloud为ribbon和eureka结合的负载均衡应用提供更便捷的使用方式。

Example spring boot app

@Configuration

@ComponentScan

@EnableAutoConfiguration

@EnableEurekaClient

@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);

}

In the @FeignClient annotation the String value ("stores" above) is an arbitrary client name, which is used to create a Ribbon load balancer (see [below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon)). You can also specify a URL using the url attribute (absolute value or just a hostname). The name of the bean in the application context is the fully qualified name of the interface. An alias is also created which is the 'name' attribute plus 'FeignClient'. For the example above,@Qualifier("storesFeignClient") could be used to reference the bean. If you want to change the default@Qualifier value, this can be done with the qualifier value in @FeignClient.

The Ribbon client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don’t want to use Eureka, you can simply configure a list of servers in your external configuration (see [above for example](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon-without-eureka)).

上面例子的@FeignClient注解中，stores字符串可以是任务的客户端名，用来创建ribbon负载均衡器（参考[below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon)），也可以通过url来指定服务（服务的域名）。@FeignClient注解的接口类生成的bean名和接口名一直。当然也可以通过@FeignClient注解的name属性来指定不同的名字，例如：上面的栗子中默认为@Qualifier("storesFeignClient")，用于引用bean，若需要改变@Qualifier的默认值，可以在@FeignClient去改变qualifier 的值（通过name属性）

上面栗子的ribbon客户端将获取stores服务的物理地址。如果你的应用是eureka客户端，也可用使用服务id来指定服务。如果不想用eureka，为ribbon指定服务端列表即可（see [above for example](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon-without-eureka)）

Overriding Feign Defaults

A central concept in Spring Cloud’s Feign support is that of the named client. Each feign client is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer using the @FeignClient annotation. Spring Cloud creates a new ensemble as anApplicationContext on demand for each named client using FeignClientsConfiguration. This contains (amongst other things) an feign.Decoder, a feign.Encoder, and a feign.Contract.

重写feigh默认实现

Spring cloud feign的一个核心概念是命名客户端。每一个feign客户端都是由一些特定组件集合在一起，完成连接到远程服务端的工作。这个组件集合由开发者通过@FeignClient注解赋予名字。spring cloud为每一个设置了FeignClientsConfiguration的@FeignClient创建新的组件集合（代替默认实现）。这些组件包括： feign.Decoder, a feign.Encoder, and a feign.Contract.

Spring Cloud lets you take full control of the feign client by declaring additional configuration (on top of theFeignClientsConfiguration) using @FeignClient. Example:

Spring cloud 让你能够完全控制feign客户端，只要设置自己的FeignClientsConfiguration类，例如：

@FeignClient(name = "stores", configuration = FooConfiguration.class)

public interface StoreClient {

//..

}

In this case the client is composed from the components already in FeignClientsConfiguration together with any inFooConfiguration (where the latter will override the former).

这里通过inFooConfiguration 重写FeignClientsConfiguration 曝露的组件来替换默认的实现行为。

|  |  |  |
| --- | --- | --- |
| **WARNING** | | The FooConfiguration has to be @Configuration but take care that it is not in a @ComponentScanfor the main application context, otherwise it will be used for every @FeignClient. If you use@ComponentScan (or @SpringBootApplication) you need to take steps to avoid it being included (for instance put it in a separate, non-overlapping package, or specify the packages to scan explicitly in the@ComponentScan). |
| **NOTE** | The serviceId attribute is now deprecated in favor of the name attribute. | |

|  |  |
| --- | --- |
| **WARNING** | Previously, using the url attribute, did not require the name attribute. Using name is now required. |

警告：FooConfiguration 类需要加上@Configuration注解，但是不能被@ComponentScan扫描注入。否则将被用于该应用的所有@FeignClient注解的接口（成为全局配置）。如果应用是通过@ComponentScan (or @SpringBootApplication)进行注入，需要将FooConfiguration 类放到扫描不到的包路径中。

注意：serviceId 属性已经过期，建议使用name属性

警告：以前用url属性是不需要指定name属性的，现在都需要制定name属性。

Placeholders are supported in the name and url attributes.

Name和url属性支持占位符

@FeignClient(name = "${feign.name}", url = "${feign.url}")

public interface StoreClient {

//..

}

Spring Cloud Netflix provides the following beans by default for feign (BeanType beanName: ClassName):

* Decoder feignDecoder: ResponseEntityDecoder (which wraps a SpringDecoder)
* Encoder feignEncoder: SpringEncoder
* Logger feignLogger: Slf4jLogger
* Contract feignContract: SpringMvcContract
* Feign.Builder feignBuilder: HystrixFeign.Builder
* Client feignClient: if Ribbon is enabled it is a LoadBalancerFeignClient, otherwise the default feign client is used.

Spring Cloud Netflix为feign提供下列默认实现的bean：

Decoder feign解码：默认实现类ResponseEntityDecoder （封装了SpringDecoder）

Encoder feign编码：默认SpringEncoder

Logger ：Slf4jLogger

Contract  通信方式：SpringMvcContract风格实现

Feign.Builder： HystrixFeign.Builder

Client ：如果ribbon开启，则是LoadBalancerFeignClient，否则是默认的（我也不知道是什么）

The OkHttpClient and ApacheHttpClient feign clients can be used by setting feign.okhttp.enabled orfeign.httpclient.enabled to true, respectively, and having them on the classpath.

设置feign.okhttp.enabled orfeign.httpclient.enabled=true可以启用OkHttpClient and ApacheHttpClient，但是记得引入相关依赖。

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

* Logger.Level
* Retryer
* ErrorDecoder
* Request.Options
* Collection<RequestInterceptor>

默认Spring Cloud Netflix不提供下列的bean实现，但是仍然会搜索相关类型的bean，若存在则添加到上下文中。

Creating a bean of one of those type and placing it in a @FeignClient configuration (such as FooConfiguration above) allows you to override each one of the beans described. Example:

创建类似FooConfiguration 的bean，使用如下方式可以替换默认的组件实现：

@Configuration

public class FooConfiguration {

@Bean

public Contract feignContract() {

return new feign.Contract.Default();

}

@Bean

public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

return new BasicAuthRequestInterceptor("user", "password");

}

}

This replaces the SpringMvcContract with feign.Contract.Default and adds a RequestInterceptor to the collection of RequestInterceptor.

Default configurations can be specified in the @EnableFeignClients attribute defaultConfiguration in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

上面栗子用feign.Contract.Default替换了默认的SpringMvcContract ，并提供了一个RequestInterceptor。

@EnableFeignClients中的defaultConfiguration 属性定了所有默认组件实现，使用方式和上面栗子同样的友好，只是defaultConfiguration 会作用到应用的所有feign接口。

Feign Hystrix Support

If Hystrix is on the classpath, by default Feign will wrap all methods with a circuit breaker. Returning acom.netflix.hystrix.HystrixCommand is also available. This lets you use reactive patterns (with a call to.toObservable() or .observe() or asynchronous use (with a call to .queue()).

To disable Hystrix support for Feign, set feign.hystrix.enabled=false.

To disable Hystrix support on a per-client basis create a vanilla Feign.Builder with the "prototype" scope, e.g.:

Feign hystrix 支持

若hystrix在类路径中，默认feign会封装断路器的所有操作。也可以使用com.netflix.hystrix.HystrixCommand。通过.toObservable() or .observe()或者异步调用（ .queue()）实现反馈机制。

 feign.hystrix.enabled=false.关闭hystrix对feign支持（全局）。

可以对每个客户端设置是否支持hystrix，[在如FooConfiguration类中的Feign.Builder加入@Scope("prototype")](mailto:在如FooConfiguration类中的Feign.Builder加入@Scope(%22prototype%22))注解即可关闭hystrix对氮气feign接口的支持。

@Configuration

public class FooConfiguration {

@Bean

@Scope("prototype")

public Feign.Builder feignBuilder() {

return Feign.builder();

}

}

Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given @FeignClient set the fallback attribute to the class name that implements the fallback.

Feign的hystrix回调

Hystrix支持回调：当回路开启或者发生错误后被执行的代码。@FeignClient中启用回调需要设置fallback 属性，指定类名，并实现回调代码。

@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");

}

}

|  |  |
| --- | --- |
| **WARNING** | There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return com.netflix.hystrix.HystrixCommandand rx.Observable. |

警告：feign中实现回调有一定的限制。不支持返回com.netflix.hystrix.HystrixCommand和rx.Observable对象的方法

Feign Inheritance Support

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

Feign继承支持

Feign支持继承。允许将通用的操作抽象到一个基础接口中

*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 {

}

|  |  |
| --- | --- |
| **NOTE** | It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn’t work with Spring MVC in its current form (method parameter mapping is not inherited). |

注意：服务端和客户端共享同一接口是不合适的。这导致强关联，并且在spring MVC下也并不起作用（参数映射注解不会被继承）

Feign request/response compression

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

Feign请求/响应压缩

可以对feign的请求和响应进行GZIP压缩，设置如下：

feign.compression.request.enabled=true

feign.compression.response.enabled=true

Feign request compression gives you settings similar to what you may set for your web server:

Feign请求压缩设置类似web服务器设置

feign.compression.request.enabled=true

feign.compression.request.mime-types=text/xml,application/xml,application/json

feign.compression.request.min-request-size=2048

These properties allow you to be selective about the compressed media types and minimum request threshold length.

这些配置项允许你有选择的对媒体消息和请求最小压缩长度进行压缩。

Feign logging

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the DEBUG level.

Feign日志

每个feign客户端都会创建日志。默认日志名是feign接口类的全称。Debug级别。

可以对每个feign接口类定义日志级别 Logger.Level

*application.yml*

logging.level.project.user.UserClient: DEBUG

The Logger.Level object that you may configure per client, tells Feign how much to log. Choices are:

* NONE, No logging (**DEFAULT**).
* BASIC, Log only the request method and URL and the response status code and execution time.
* HEADERS, Log the basic information along with request and response headers.
* FULL, Log the headers, body, and metadata for both requests and responses.

For example, the following would set the Logger.Level to FULL:

日志级别选择有：

NONE，不进行日志（默认）

BASIC，仅日志请求方法名和url和响应状态码和执行时长

HEADERS，日志请求和响应头的基本信息

FULL，日志请求和响应头、body以及请求和响应的元数据信息

例如，下面将日志基本设置为FULL

@Configuration

public class FooConfiguration {

@Bean

Logger.Level feignLoggerLevel() {

return Logger.Level.FULL;

}

}

External Configuration: Archaius

[Archaius](https://github.com/Netflix/archaius) is the Netflix client side configuration library. It is the library used by all of the Netflix OSS components for configuration. Archaius is an extension of the [Apache Commons Configuration](http://commons.apache.org/proper/commons-configuration) project. It allows updates to configuration by either polling a source for changes or for a source to push changes to the client. Archaius uses Dynamic<Type>Property classes as handles to properties.

*Archaius Example*

class ArchaiusTest {

DynamicStringProperty myprop = DynamicPropertyFactory

.getInstance()

.getStringProperty("my.prop");

void doSomething() {

OtherClass.someMethod(myprop.get());

}

}

Archaius has its own set of configuration files and loading priorities. Spring applications should generally not use Archaius directly, but the need to configure the Netflix tools natively remains. Spring Cloud has a Spring Environment Bridge so Archaius can read properties from the Spring Environment. This allows Spring Boot projects to use the normal configuration toolchain, while allowing them to configure the Netflix tools, for the most part, as documented.

Router and Filter: Zuul

Routing in an integral part of a microservice architecture. For example, / may be mapped to your web application,/api/users is mapped to the user service and /api/shop is mapped to the shop service. [Zuul](https://github.com/Netflix/zuul) is a JVM based router and server side load balancer by Netflix.

[Netflix uses Zuul](http://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27) for the following:

* Authentication
* Insights
* Stress Testing
* Canary Testing
* Dynamic Routing
* Service Migration
* Load Shedding
* Security
* Static Response handling
* Active/Active traffic management

Zuul’s rule engine allows rules and filters to be written in essentially any JVM language, with built in support for Java and Groovy.

|  |  |
| --- | --- |
| **NOTE** | The configuration property zuul.max.host.connections has been replaced by two new properties,zuul.host.maxTotalConnections and zuul.host.maxPerRouteConnections which default to 200 and 20 respectively. |
| **NOTE** | Default Hystrix isolation pattern (ExecutionIsolationStrategy) for all routes is SEMAPHORE.zuul.ribbonIsolationStrategy can be changed to THREAD if this isolation pattern is preferred. |

Embedded Zuul Reverse Proxy

Spring Cloud has created an embedded Zuul proxy to ease the development of a very common use case where a UI application wants to proxy calls to one or more back end services. This feature is useful for a user interface to proxy to the backend services it requires, avoiding the need to manage CORS and authentication concerns independently for all the backends.

To enable it, annotate a Spring Boot main class with @EnableZuulProxy, and this forwards local calls to the appropriate service. By convention, a service with the ID "users", will receive requests from the proxy located at /users (with the prefix stripped). The proxy uses Ribbon to locate an instance to forward to via discovery, and all requests are executed in a hystrix command, so failures will show up in Hystrix metrics, and once the circuit is open the proxy will not try to contact the service.

|  |  |
| --- | --- |
| **NOTE** | the Zuul starter does not include a discovery client, so for routes based on service IDs you need to provide one of those on the classpath as well (e.g. Eureka is one choice). |

To skip having a service automatically added, set zuul.ignored-services to a list of service id patterns. If a service matches a pattern that is ignored, but also included in the explicitly configured routes map, then it will be unignored. Example:

*application.yml*

zuul:

ignoredServices: '\*'

routes:

users: /myusers/\*\*

In this example, all services are ignored **except** "users".

To augment or change the proxy routes, you can add external configuration like the following:

*application.yml*

zuul:

routes:

users: /myusers/\*\*

This means that http calls to "/myusers" get forwarded to the "users" service (for example "/myusers/101" is forwarded to "/101").

To get more fine-grained control over a route you can specify the path and the serviceId independently:

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users\_service

This means that http calls to "/myusers" get forwarded to the "users\_service" service. The route has to have a "path" which can be specified as an ant-style pattern, so "/myusers/\*" only matches one level, but "/myusers/\*\*" matches hierarchically.

The location of the backend can be specified as either a "serviceId" (for a service from discovery) or a "url" (for a physical location), e.g.

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

url: http://example.com/users\_service

These simple url-routes don’t get executed as a HystrixCommand nor can you loadbalance multiple URLs with Ribbon. To achieve this, specify a service-route and configure a Ribbon client for the serviceId (this currently requires disabling Eureka support in Ribbon: see [above for more information](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-ribbon-without-eureka)), e.g.

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users

ribbon:

eureka:

enabled: false

users:

ribbon:

listOfServers: example.com,google.com

You can provide convention between serviceId and routes using regexmapper. It uses regular expression named groups to extract variables from serviceId and inject them into a route pattern.

*ApplicationConfiguration.java*

@Bean

public PatternServiceRouteMapper serviceRouteMapper() {

return new PatternServiceRouteMapper(

"(?<name>^.+)-(?<version>v.+$)",

"${version}/${name}");

}

This means that a serviceId "myusers-v1" will be mapped to route "/v1/myusers/\*\*". Any regular expression is accepted but all named groups must be present in both servicePattern and routePattern. If servicePattern does not match a serviceId, the default behavior is used. In the example above, a serviceId "myusers" will be mapped to route "/myusers/\*\*" (no version detected) This feature is disable by default and only applies to discovered services.

To add a prefix to all mappings, set zuul.prefix to a value, such as /api. The proxy prefix is stripped from the request before the request is forwarded by default (switch this behaviour off with zuul.stripPrefix=false). You can also switch off the stripping of the service-specific prefix from individual routes, e.g.

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

stripPrefix: false

In this example, requests to "/myusers/101" will be forwarded to "/myusers/101" on the "users" service.

The zuul.routes entries actually bind to an object of type ZuulProperties. If you look at the properties of that object you will see that it also has a "retryable" flag. Set that flag to "true" to have the Ribbon client automatically retry failed requests (and if you need to you can modify the parameters of the retry operations using the Ribbon client configuration).

The X-Forwarded-Host header is added to the forwarded requests by default. To turn it off set zuul.addProxyHeaders = false. The prefix path is stripped by default, and the request to the backend picks up a header "X-Forwarded-Prefix" ("/myusers" in the examples above).

An application with @EnableZuulProxy could act as a standalone server if you set a default route ("/"), for examplezuul.route.home: / would route all traffic (i.e. "/\*\*") to the "home" service.

If more fine-grained ignoring is needed, you can specify specific patterns to ignore. These patterns are evaluated at the start of the route location process, which means prefixes should be included in the pattern to warrant a match. Ignored patterns span all services and supersede any other route specification.

*application.yml*

zuul:

ignoredPatterns: /\*\*/admin/\*\*

routes:

users: /myusers/\*\*

This means that all calls such as "/myusers/101" will be forwarded to "/101" on the "users" service. But calls including "/admin/" will not resolve.

|  |  |
| --- | --- |
| **WARNING** | If you need your routes to have their order preserved you need to use a YAML file as the ordering will be lost using a properties file. For example: |

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

legacy:

path: /\*\*

If you were to use a properties file, the legacy path may end up in front of the users path rendering the users path unreachable.

Cookies and Sensitive Headers

It’s OK to share headers between services in the same system, but you probably don’t want sensitive headers leaking downstream into external servers. You can specify a list of ignored headers as part of the route configuration. Cookies play a special role because they have well-defined semantics in browsers, and they are always to be treated as sensitive. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user because they all get jumbled up (all downstream services look like they come from the same place).

If you are careful with the design of your services, for example if only one of the downstream services sets cookies, then you might be able to let them flow from the backend all the way up to the caller. Also, if your proxy sets cookies and all your back end services are part of the same system, it can be natural to simply share them (and for instance use Spring Session to link them up to some shared state). Other than that, any cookies that get set by downstream services are likely to be not very useful to the caller, so it is recommended that you make (at least) "Set-Cookie" and "Cookie" into sensitive headers for routes that are not part of your domain. Even for routes that **are** part of your domain, try to think carefully about what it means before allowing cookies to flow between them and the proxy.

The sensitive headers can be configured as a comma-separated list per route, e.g.

*application.yml*

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders: Cookie,Set-Cookie,Authorization

url: https://downstream

Sensitive headers can also be set globally by setting zuul.sensitiveHeaders. If sensitiveHeaders is set on a route, this will override the global sensitiveHeaders setting.

|  |  |
| --- | --- |
| **NOTE** | this is the default value for sensitiveHeaders, so you don’t need to set it unless you want it to be different. N.B. this is new in Spring Cloud Netflix 1.1 (in 1.0 the user had no control over headers and all cookies flow in both directions). |

In addition to the per-route sensitive headers, you can set a global value for zuul.ignoredHeaders for values that should be discarded (both request and response) during interactions with downstream services. By default these are empty, if Spring Security is not on the classpath, and otherwise they are initialized to a set of well-known "security" headers (e.g. involving caching) as specified by Spring Security. The assumption in this case is that the downstream services might add these headers too, and we want the values from the proxy.

The Routes Endpoint

If you are using @EnableZuulProxy with tha Spring Boot Actuator you will enable (by default) an additional endpoint, available via HTTP as /routes. A GET to this endpoint will return a list of the mapped routes. A POST will force a refresh of the existing routes (e.g. in case there have been changes in the service catalog).

|  |  |
| --- | --- |
| **NOTE** | the routes should respond automatically to changes in the service catalog, but the POST to /routes is a way to force the change to happen immediately. |

Strangulation Patterns and Local Forwards

A common pattern when migrating an existing application or API is to "strangle" old endpoints, slowly replacing them with different implementations. The Zuul proxy is a useful tool for this because you can use it to handle all traffic from clients of the old endpoints, but redirect some of the requests to new ones.

Example configuration:

*application.yml*

zuul:

routes:

first:

path: /first/\*\*

url: http://first.example.com

second:

path: /second/\*\*

url: forward:/second

third:

path: /third/\*\*

url: forward:/3rd

legacy:

path: /\*\*

url: http://legacy.example.com

In this example we are strangling the "legacy" app which is mapped to all requests that do not match one of the other patterns. Paths in /first/\*\* have been extracted into a new service with an external URL. And paths in /second/\*\*are forwared so they can be handled locally, e.g. with a normal Spring @RequestMapping. Paths in /third/\*\* are also forwarded, but with a different prefix (i.e. /third/foo is forwarded to /3rd/foo).

|  |  |
| --- | --- |
| **NOTE** | The ignored patterns aren’t completely ignored, they just aren’t handled by the proxy (so they are also effectively forwarded locally). |

Uploading Files through Zuul

If you @EnableZuulProxy you can use the proxy paths to upload files and it should just work as long as the files are small. For large files there is an alternative path which bypasses the Spring DispatcherServlet (to avoid multipart processing) in "/zuul/\*". I.e. if zuul.routes.customers=/customers/\*\* then you can POST large files to "/zuul/customers/\*". The servlet path is externalized via zuul.servletPath. Extremely large files will also require elevated timeout settings if the proxy route takes you through a Ribbon load balancer, e.g.

*application.yml*

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000

ribbon:

ConnectTimeout: 3000

ReadTimeout: 60000

Note that for streaming to work with large files, you need to use chunked encoding in the request (which some browsers do not do by default). E.g. on the command line:

$ curl -v -H "Transfer-Encoding: chunked" \

-F "file=@mylarge.iso" localhost:9999/zuul/simple/file

Plain Embedded Zuul

You can also run a Zuul server without the proxying, or switch on parts of the proxying platform selectively, if you use@EnableZuulServer (instead of @EnableZuulProxy). Any beans that you add to the application of type ZuulFilter will be installed automatically, as they are with @EnableZuulProxy, but without any of the proxy filters being added automatically.

In this case the routes into the Zuul server are still specified by configuring "zuul.routes.\*", but there is no service discovery and no proxying, so the "serviceId" and "url" settings are ignored. For example:

*application.yml*

zuul:

routes:

api: /api/\*\*

maps all paths in "/api/\*\*" to the Zuul filter chain.

Disable Zuul Filters

Zuul for Spring Cloud comes with a number of ZuulFilter beans enabled by default in both proxy and server mode. See[the zuul filters package](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters) for the possible filters that are enabled. If you want to disable one, simply set zuul.<SimpleClassName>.<filterType>.disable=true. By convention, the package after filters is the Zuul filter type. For example to disable org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter setzuul.SendResponseFilter.post.disable=true.

Polyglot support with Sidecar

Do you have non-jvm languages you want to take advantage of Eureka, Ribbon and Config Server? The Spring Cloud Netflix Sidecar was inspired by [Netflix Prana](https://github.com/Netflix/Prana). It includes a simple http api to get all of the instances (ie host and port) for a given service. You can also proxy service calls through an embedded Zuul proxy which gets its route entries from Eureka. The Spring Cloud Config Server can be accessed directly via host lookup or through the Zuul Proxy. The non-jvm app should implement a health check so the Sidecar can report to eureka if the app is up or down.

To enable the Sidecar, create a Spring Boot application with @EnableSidecar. This annotation includes@EnableCircuitBreaker, @EnableDiscoveryClient, and @EnableZuulProxy. Run the resulting application on the same host as the non-jvm application.

To configure the side car add sidecar.port and sidecar.health-uri to application.yml. The sidecar.portproperty is the port the non-jvm app is listening on. This is so the Sidecar can properly register the app with Eureka. Thesidecar.health-uri is a uri accessible on the non-jvm app that mimicks a Spring Boot health indicator. It should return a json document like the following:

*health-uri-document*

{

"status":"UP"

}

Here is an example application.yml for a Sidecar application:

*application.yml*

server:

port: 5678

spring:

application:

name: sidecar

sidecar:

port: 8000

health-uri: http://localhost:8000/health.json

The api for the DiscoveryClient.getInstances() method is /hosts/{serviceId}. Here is an example response for/hosts/customers that returns two instances on different hosts. This api is accessible to the non-jvm app (if the sidecar is on port 5678) at [http://localhost:5678/hosts/{serviceId}](http://localhost:5678/hosts/%7BserviceId%7D).

*/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

}

]

The Zuul proxy automatically adds routes for each service known in eureka to /<serviceId>, so the customers service is available at /customers. The Non-jvm app can access the customer service via <http://localhost:5678/customers>(assuming the sidecar is listening on port 5678).

If the Config Server is registered with Eureka, non-jvm application can access it via the Zuul proxy. If the serviceId of the ConfigServer is configserver and the Sidecar is on port 5678, then it can be accessed at<http://localhost:5678/configserver>

Non-jvm app can take advantage of the Config Server’s ability to return YAML documents. For example, a call to<http://sidecar.local.spring.io:5678/configserver/default-master.yml> might result in a YAML document like the following

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

password: password

info:

description: Spring Cloud Samples

url: https://github.com/spring-cloud-samples

RxJava with Spring MVC

Spring Cloud Netflix includes the [RxJava](https://github.com/ReactiveX/RxJava).

*RxJava is a Java VM implementation of*[*Reactive Extensions*](http://reactivex.io/)*: a library for composing asynchronous and event-based programs by using observable sequences.*

Spring Cloud Netflix provides support for returning rx.Single objects from Spring MVC Controllers. It also supports using rx.Observable objects for [Server-sent events (SSE)](https://en.wikipedia.org/wiki/Server-sent_events). This can be very convenient if your internal APIs are already built using RxJava (see [Feign Hystrix Support](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/1.2.0.M1/#spring-cloud-feign-hystrix) for examples).

Here are some examples of using rx.Single:

@RequestMapping(method = RequestMethod.GET, value = "/single")

public Single<String> single() {

return Single.just("single value");

}

@RequestMapping(method = RequestMethod.GET, value = "/singleWithResponse")

public ResponseEntity<Single<String>> singleWithResponse() {

return new ResponseEntity<>(Single.just("single value"),

HttpStatus.NOT\_FOUND);

}

@RequestMapping(method = RequestMethod.GET, value = "/throw")

public Single<Object> error() {

return Single.error(new RuntimeException("Unexpected"));

}

If you have an Observable, rather than a single, you can use .toSingle() or .toList().toSingle(). Here are some examples:

@RequestMapping(method = RequestMethod.GET, value = "/single")

public Single<String> single() {

return Observable.just("single value").toSingle();

}

@RequestMapping(method = RequestMethod.GET, value = "/multiple")

public Single<List<String>> multiple() {

return Observable.just("multiple", "values").toList().toSingle();

}

@RequestMapping(method = RequestMethod.GET, value = "/responseWithObservable")

public ResponseEntity<Single<String>> responseWithObservable() {

Observable<String> observable = Observable.just("single value");

HttpHeaders headers = new HttpHeaders();

headers.setContentType(APPLICATION\_JSON\_UTF8);

return new ResponseEntity<>(observable.toSingle(), headers, HttpStatus.CREATED);

}

@RequestMapping(method = RequestMethod.GET, value = "/timeout")

public Observable<String> timeout() {

return Observable.timer(1, TimeUnit.MINUTES).map(new Func1<Long, String>() {

@Override

public String call(Long aLong) {

return "single value";

}

});

}

If you have a streaming endpoint and client, SSE could be an option. To convert rx.Observable to a Spring SseEmitteruse RxResponse.sse(). Here are some examples:

@RequestMapping(method = RequestMethod.GET, value = "/sse")

public SseEmitter single() {

return RxResponse.sse(Observable.just("single value"));

}

@RequestMapping(method = RequestMethod.GET, value = "/messages")

public SseEmitter messages() {

return RxResponse.sse(Observable.just("message 1", "message 2", "message 3"));

}

@RequestMapping(method = RequestMethod.GET, value = "/events")

public SseEmitter event() {

return RxResponse.sse(APPLICATION\_JSON\_UTF8,

Observable.just(new EventDto("Spring io", getDate(2016, 5, 19)),

new EventDto("SpringOnePlatform", getDate(2016, 8, 1))));

}

Metrics: Spectator, Servo, and Atlas

When used together, Spectator/Servo and Atlas provide a near real-time operational insight platform.

Spectator and Servo are Netflix’s metrics collection libraries. Atlas is a Netflix metrics backend to manage dimensional time series data.

Servo served Netflix for several years and is still usable, but is gradually being phased out in favor of Spectator, which is only designed to work with Java 8. Spring Cloud Netflix provides support for both, but Java 8 based applications are encouraged to use Spectator.

Dimensional vs. Hierarchical Metrics

Spring Boot Actuator metrics are hierarchical and metrics are separated only by name. These names often follow a naming convention that embeds key/value attribute pairs (dimensions) into the name separated by periods. Consider the following metrics for two endpoints, root and star-star:

{

"counter.status.200.root": 20,

"counter.status.400.root": 3,

"counter.status.200.star-star": 5,

}

The first metric gives us a normalized count of successful requests against the root endpoint per unit of time. But what if the system had 20 endpoints and you want to get a count of successful requests against all the endpoints? Some hierarchical metrics backends would allow you to specify a wild card such as counter.status.200.**that would read all 20 metrics and aggregate the results. Alternatively, you could provide a**HandlerInterceptorAdapter**that intercepts and records a metric like**counter.status.200.all**for all successful requests irrespective of the endpoint, but now you must write 20+1 different metrics. Similarly if you want to know the total number of successful requests for all endpoints in the service, you could specify a wild card such as**counter.status.2.\*.

Even in the presence of wildcarding support on a hierarchical metrics backend, naming consistency can be difficult. Specifically the position of these tags in the name string can slip with time, breaking queries. For example, suppose we add an additional dimension to the hierarchical metrics above for HTTP method. Then counter.status.200.rootbecomes counter.status.200.method.get.root, etc. Our counter.status.200.\* suddenly no longer has the same semantic meaning. Furthermore, if the new dimension is not applied uniformly across the codebase, certain queries may become impossible. This can quickly get out of hand.

Netflix metrics are tagged (a.k.a. dimensional). Each metric has a name, but this single named metric can contain multiple statistics and 'tag' key/value pairs that allows more querying flexibility. In fact, the statistics themselves are recorded in a special tag.

Recorded with Netflix Servo or Spectator, a timer for the root endpoint described above contains 4 statistics per status code, where the count statistic is identical to Spring Boot Actuator’s counter. In the event that we have encountered an HTTP 200 and 400 thus far, there will be 8 available data points:

{

"root(status=200,stastic=count)": 20,

"root(status=200,stastic=max)": 0.7265630630000001,

"root(status=200,stastic=totalOfSquares)": 0.04759702862580789,

"root(status=200,stastic=totalTime)": 0.2093076914666667,

"root(status=400,stastic=count)": 1,

"root(status=400,stastic=max)": 0,

"root(status=400,stastic=totalOfSquares)": 0,

"root(status=400,stastic=totalTime)": 0,

}

Default Metrics Collection

Without any additional dependencies or configuration, a Spring Cloud based service will autoconfigure a ServoMonitorRegistry and begin collecting metrics on every Spring MVC request. By default, a Servo timer with the namerest will be recorded for each MVC request which is tagged with:

1. HTTP method
2. HTTP status (e.g. 200, 400, 500)
3. URI (or "root" if the URI is empty), sanitized for Atlas
4. The exception class name, if the request handler threw an exception
5. The caller, if a request header with a key matching netflix.metrics.rest.callerHeader is set on the request. There is no default key for netflix.metrics.rest.callerHeader. You must add it to your application properties if you wish to collect caller information.

Set the netflix.metrics.rest.metricName property to change the name of the metric from rest to a name you provide.

If Spring AOP is enabled and org.aspectj:aspectjweaver is present on your runtime classpath, Spring Cloud will also collect metrics on every client call made with RestTemplate. A Servo timer with the name of restclient will be recorded for each MVC request which is tagged with:

1. HTTP method
2. HTTP status (e.g. 200, 400, 500), "CLIENT\_ERROR" if the response returned null, or "IO\_ERROR" if an IOExceptionoccurred during the execution of the RestTemplate method
3. URI, sanitized for Atlas
4. Client name

Metrics Collection: Spectator

To enable Spectator metrics, include a dependency on spring-boot-starter-spectator:

<dependency>

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

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

</dependency>

In Spectator parlance, a meter is a named, typed, and tagged configuration and a metric represents the value of a given meter at a point in time. Spectator meters are created and controlled by a registry, which currently has several different implementations. Spectator provides 4 meter types: counter, timer, gauge, and distribution summary.

Spring Cloud Spectator integration configures an injectable com.netflix.spectator.api.Registry instance for you. Specifically, it configures a ServoRegistry instance in order to unify the collection of REST metrics and the exporting of metrics to the Atlas backend under a single Servo API. Practically, this means that your code may use a mixture of Servo monitors and Spectator meters and both will be scooped up by Spring Boot Actuator MetricReader instances and both will be shipped to the Atlas backend.

Spectator Counter

A counter is used to measure the rate at which some event is occurring.

// create a counter with a name and a set of tags

Counter counter = registry.counter("counterName", "tagKey1", "tagValue1", ...);

counter.increment(); // increment when an event occurs

counter.increment(10); // increment by a discrete amount

The counter records a single time-normalized statistic.

Spectator Timer

A timer is used to measure how long some event is taking. Spring Cloud automatically records timers for Spring MVC requests and conditionally RestTemplate requests, which can later be used to create dashboards for request related metrics like latency:

*Request Latency*

image::RequestLatency.png []

// create a timer with a name and a set of tags

Timer timer = registry.timer("timerName", "tagKey1", "tagValue1", ...);

// execute an operation and time it at the same time

T result = timer.record(() -> fooReturnsT());

// alternatively, if you must manually record the time

Long start = System.nanoTime();

T result = fooReturnsT();

timer.record(System.nanoTime() - start, TimeUnit.NANOSECONDS);

The timer simultaneously records 4 statistics: count, max, totalOfSquares, and totalTime. The count statistic will always match the single normalized value provided by a counter if you had called increment() once on the counter for each time you recorded a timing, so it is rarely necessary to count and time separately for a single operation.

For [long running operations](https://github.com/Netflix/spectator/wiki/Timer-Usage#longtasktimer), Spectator provides a special LongTaskTimer.

Spectator Gauge

Gauges are used to determine some current value like the size of a queue or number of threads in a running state. Since gauges are sampled, they provide no information about how these values fluctuate between samples.

The normal use of a gauge involves registering the gauge once in initialization with an id, a reference to the object to be sampled, and a function to get or compute a numeric value based on the object. The reference to the object is passed in separately and the Spectator registry will keep a weak reference to the object. If the object is garbage collected, then Spectator will automatically drop the registration. See [the note](https://github.com/Netflix/spectator/wiki/Gauge-Usage#using-lambda) in Spectator’s documentation about potential memory leaks if this API is misused.

// the registry will automatically sample this gauge periodically

registry.gauge("gaugeName", pool, Pool::numberOfRunningThreads);

// manually sample a value in code at periodic intervals -- last resort!

registry.gauge("gaugeName", Arrays.asList("tagKey1", "tagValue1", ...), 1000);

Spectator Distribution Summaries

A distribution summary is used to track the distribution of events. It is similar to a timer, but more general in that the size does not have to be a period of time. For example, a distribution summary could be used to measure the payload sizes of requests hitting a server.

// the registry will automatically sample this gauge periodically

DistributionSummary ds = registry.distributionSummary("dsName", "tagKey1", "tagValue1", ...);

ds.record(request.sizeInBytes());

Metrics Collection: Servo

|  |  |
| --- | --- |
| **WARNING** | If your code is compiled on Java 8, please use Spectator instead of Servo as Spectator is destined to replace Servo entirely in the long term. |

In Servo parlance, a monitor is a named, typed, and tagged configuration and a metric represents the value of a given monitor at a point in time. Servo monitors are logically equivalent to Spectator meters. Servo monitors are created and controlled by a MonitorRegistry. In spite of the above warning, Servo does have a [wider array](https://github.com/Netflix/servo/wiki/Getting-Started) of monitor options than Spectator has meters.

Spring Cloud integration configures an injectable com.netflix.servo.MonitorRegistry instance for you. Once you have created the appropriate Monitor type in Servo, the process of recording data is wholly similar to Spectator.

Creating Servo Monitors

If you are using the Servo MonitorRegistry instance provided by Spring Cloud (specifically, an instance ofDefaultMonitorRegistry), Servo provides convenience classes for retrieving [counters](https://github.com/Netflix/spectator/wiki/Servo-Comparison#dynamiccounter) and [timers](https://github.com/Netflix/spectator/wiki/Servo-Comparison#dynamictimer). These convenience classes ensure that only one Monitor is registered for each unique combination of name and tags.

To manually create a Monitor type in Servo, especially for the more exotic monitor types for which convenience methods are not provided, instantiate the appropriate type by providing a MonitorConfig instance:

MonitorConfig config = MonitorConfig.builder("timerName").withTag("tagKey1", "tagValue1").build();

// somewhere we should cache this Monitor by MonitorConfig

Timer timer = new BasicTimer(config);

monitorRegistry.register(timer);

Metrics Backend: Atlas

Atlas was developed by Netflix to manage dimensional time series data for near real-time operational insight. Atlas features in-memory data storage, allowing it to gather and report very large numbers of metrics, very quickly.

Atlas captures operational intelligence. Whereas business intelligence is data gathered for analyzing trends over time, operational intelligence provides a picture of what is currently happening within a system.

Spring Cloud provides a spring-cloud-starter-atlas that has all the dependencies you need. Then just annotate your Spring Boot application with @EnableAtlas and provide a location for your running Atlas server with thenetflix.atlas.uri property.

Global tags

Spring Cloud enables you to add tags to every metric sent to the Atlas backend. Global tags can be used to separate metrics by application name, environment, region, etc.

Each bean implementing AtlasTagProvider will contribute to the global tag list:

@Bean

AtlasTagProvider atlasCommonTags(

@Value("${spring.application.name}") String appName) {

return () -> Collections.singletonMap("app", appName);

}

Using Atlas

To bootstrap a in-memory standalone Atlas instance:

$ curl -LO https://github.com/Netflix/atlas/releases/download/v1.4.2/atlas-1.4.2-standalone.jar

$ java -jar atlas-1.4.2-standalone.jar

|  |  |
| --- | --- |
| **TIP** | An Atlas standalone node running on an r3.2xlarge (61GB RAM) can handle roughly 2 million metrics per minute for a given 6 hour window. |

Once running and you have collected a handful of metrics, verify that your setup is correct by listing tags on the Atlas server:

$ curl http://ATLAS/api/v1/tags

|  |  |
| --- | --- |
| **TIP** | After executing several requests against your service, you can gather some very basic information on the request latency of every request by pasting the following url in your browser:[http://ATLAS/api/v1/graph?q=name,rest,:eq,:avg](http://atlas/api/v1/graph?q=name,rest,:eq,:avg) |

The Atlas wiki contains a [compilation of sample queries](https://github.com/Netflix/atlas/wiki/Single-Line) for various scenarios.

Make sure to check out the [alerting philosophy](https://github.com/Netflix/atlas/wiki/Alerting-Philosophy) and docs on using [double exponential smoothing](https://github.com/Netflix/atlas/wiki/DES) to generate dynamic alert thresholds.

Last updated 2016-08-22 14:34:48 EDT