年级　　　　　　专业　　　　　　　　　学号　　　　　　姓名

Spring Cloud Framework for Microsoft Service Architecture

With the continuous development of computer technology, Spring Cloud has gradually become the mainstream framework of micro-service architecture. Before understanding Spring Cloud, we first review the history of traditional architecture.

(1) Single Architecture: Single Architecture is common in small and micro enterprises. A typical representative is an application, a database, a Web container, which can run. It is the standard single architecture.

(2) Vertical Architecture: After the development of the single architecture for a period of time, the business model of the company has been recognized, and the volume of transactions has slowly increased. At this time, some enterprises will split the original business in order to cope with greater traffic, such as: background system, front-end system, trading system and so on. At this stage, the system is often divided into different levels, each level has corresponding responsibilities, UI layer is responsible for interaction with users, business logic layer is responsible for specific business functions, database layer is responsible for data exchange and storage with the upper level. This is the vertical architecture.

(3) Service Architecture: If the company further enlarges, the vertical subsystem will become more and more, and the invocation relationship between the system and the system shows an exponential upward trend. In this context, many companies will consider service SOA. SOA represents a service-oriented architecture, dividing applications into different modules according to different responsibilities, and different modules interact directly through specific protocols and interfaces. In this way, the whole system is divided into many individual component services to fulfill the request. When the traffic is too large, it is supported by expanding the

corresponding components horizontally. All components meet the overall business needs through interaction. The advantage of service-oriented SOA is that it can deploy, compose and use loosely coupled coarse-grained application components through the network according to requirements. Service layer is the basis of SOA, which can be directly invoked by applications, thus effectively controlling the human dependence of interaction with software agents in the control system. Service architecture is a loosely coupled architecture. The principle of service decoupling is high cohesion within services and low coupling between services. At this stage, the system is often divided into different levels, each level has corresponding responsibilities, UI layer is responsible for interaction with users, business logic layer is responsible for specific business functions, database layer is responsible for data exchange and storage with the upper level.

Spring provides a series of tools to implement SOA architecture, which can help developers quickly build common components in distributed systems (such as configuration management, service discovery, circuit breaker, intelligent routing, micro-agent, control bus, one-time token, global lock, primary node election, distributed session, cluster state). Coordinating various systems in the distributed environment to provide template configuration for various services. With Spring Cloud, developers can build applications that implement these templates and work very well in any distributed environment, ranging from laptops to data centers and cloud platforms.

1. Service Registration

The definition of Spring Cloud is abstract. We can start with something simple. Spring Cloud is based on Spring Boot and is best suited for managing the various micro-service applications created by Spring Boot. To manage all Spring Boot micro services in distributed environment, there must be a problem of service registration.

So let's start with the registration of services. Since it is registered, there must be a server that manages the registry. Each Spring Book application managed by Spring Cloud is the client that needs to be registered.

Spring Cloud uses erureka server, and then all applications that need to access configuration files are registered as an erureka client. Eureka is a highly available component. It has no backend cache. After each instance is registered, it needs to send a heartbeat to the registry. By default, erureka server is also an Eureka client, and a server must be specified.

2. Configuration Management

With Config Server, you can manage the external properties of your application in all environments. Conceptual mapping on client and server is the same abstraction as Spring Environment and Property Source, so they fit well with Spring applications, but can be used with any application running in any language. As applications go through the deployment process from developers to testing and production, you can manage the configuration between these environments and determine that the application has everything to run when it is migrated. The default implementation of the server storage backend uses git, so it easily supports the tag version configuration environment and accesses various tools for managing content. It's easy to add alternative implementations and insert them using Spring configuration. For example, to build a website, we need to configure database connection, specify the IP address of database server, database name, user name and password, etc. Normally, we can define this information in a configuration file, or develop a page to configure it specifically. It's convenient to have only one web server. But if you need to build the same servers, of course, each server can do the same configuration, but maintenance and synchronization will be very troublesome. Configuration services I understand have at least two different scenarios:

(1) Multiple clients use the same configuration: for example, in a cluster of servers, if the back end uses the same database, then each server uses the same configuration.

(2) Different customers use different configurations: for example, typical scenarios are development, testing, production using the same system, but using different databases.

If there is a unified basic configuration, is it convenient? A feasible way is to put these configuration files in a shared storage (such as network shared disk). In this way, only one or more configuration files need to be modified in shared storage. However, the way of sharing files is limited by the specific deployment environment, and it is very difficult for many Web servers to share the same storage hard disk. The disadvantage of shared disk is that it is difficult to locate resources. Spring Cloud's solution is to put these configuration files in the version management server. Spring Cloud default configuration uses GIT. All Web services retrieve these configuration files from GIT. Because there is no need to share storage between GIT server and specific Web server, as long as the network is accessible, the decoupling of storage location of Web service configuration information can be realized.

3. Load Balancing

Load balancing can be divided into server-side load balancing and client-side load balancing. Server-side load balancing is completely handled by the server, and the client does not need to do anything. In client load balancing technology, the client needs to maintain a set of server references. Every time the client requests to the server, in order to achieve load balancing, Ribbon components will start to work, and it will actively select a service node according to the algorithm. The commonly used load balancing algorithms are Round Robbin, Random, Hash, Static Weighted and so on. Spring provides two types of service scheduling: Ribbon + restful and Feign.

Ribbon is a client-based load balancer. Ribbon provides a lot of control over HTTP and TCP clients.

4. Feign Components

Let's start with a brief explanation of declarative implementation: to do one thing, you need to know three elements: where, what, how. That is, where, how and what. When do we fall into the category of how?

(1) Programming Implementation: Every element (where, what, how) needs to be represented by specific code implementation. Traditionally, programmatic implementations are common, and business developers need to care about every logic.

(2) Declarative Implementation: You only need to declare where and what, not how. Spring's AOP is a declarative implementation, such as checking whether the website is logged in or not. When developing the page logic, you only need to check whether the user is logged in or not through the AOP configuration declaration loading page (where), and you don't need to care how to check whether the user is logged in or not. How to check this logic is implemented by AOP mechanism, which has nothing to do with the logic of the page being developed.

In Spring Cloud Netflix stack, each micro-service exposes its own service in the form of HTTP interface, so HTTP client must be used when calling remote service. Feign is a declarative REST client provided by Spring Cloud. The REST interface provided by remote microservices can be invoked through Feign access. Now we use Feign to call SERVICE-HELLOWORLD exposed REST interface to obtain "Hello World" information. When using Feign, Spring Cloud integrates Ribbon and Eureka to provide load balancing for HTTP clients.

5. Circuit Breakers

In the distributed environment, especially in the distributed system with micro-service structure, it is very common for one software system to call another remote system. The callee of this remote call may be another process or another host across the network. The biggest difference between this remote call and the internal call of the process is that the remote call may fail or hang up without any response until the time-out. Worse still, if multiple callers invoke the same pending service, it is likely that a service's timeout waiting spreads rapidly to the entire distributed system, causing a chain reaction, thus consuming a large amount of resources of the entire distributed system. It may eventually lead to system paralysis.

Circuit Breaker (CB) mode is designed to prevent the catastrophe caused by this cascading chain reaction in distributed systems. If something goes wrong with an electrical appliance, the fuse of the circuit will break in order to prevent disaster. The circuit breaker is similar to the fuse of the circuit. The idea of implementation is very simple. It can encapsulate the remote service that needs to be protected. It can monitor the number of failures internally. Once the number of failures reaches a threshold, all subsequent calls to the service will return to the caller directly after interception, instead of continuing to invoke the service that has been problematic, so as to achieve protection. The purpose of the guardian caller is that the entire system will not have a waterfall chain reaction due to timeouts.

6. Zuul Components

In micro-service architecture, several key components are needed, such as service registration and discovery, service consumption, load balancing, circuit breaker, intelligent routing, configuration management, etc. These components can form a simple micro-service architecture. Client requests first go through load balancing (zuul, Ngnix), then to the service gateway (zuul cluster), and then to specific services. Services are registered uniformly to the highly available service registry cluster. All configuration files of services are managed by configuration services (described in the next article). The configuration files of configuration services are placed in Git warehouse, which is convenient for developers to change at any time. Place. Zuul's main functions are routing and filtering. Routing functions are part of micro services, such as / API / user mapping to user services and / API / shop mapping to shop services. Zuul achieves load balancing.

Imagine that in a distributed system, if there are many clients that need to be refreshed and reconfigured, refreshing in this way is also a very painful thing. Is there any way to automate the system? Previously, we mentioned using external tools such as githook or Jenkins to trigger. Now another idea is that if the refresh command can be sent to config server and config server automatically notifies all config clients, then configuration refresh can be greatly simplified. In this way, although the refresh command still needs to be triggered, through hooks such as webhook, we only need to hang the associated commands on the configuration center, instead of having to associate each configuration client. Now, we do this by integrating the message queue Rabbitmq. Our goal is to send the refresh command to the configuration center once the configuration in the GIT repository changes, and then the configuration center automatically notifies all the refresh configurations that use the configuration through the message queue.

年级　　　　　　专业　　　　　　　　　学号　　　　　　姓名

微服务架构之Spring Cloud框架

随着计算机技术的不断发展，Spring Cloud逐渐成为主流的微服务架构框架，在理解Spring Cloud之前，先回顾一下传统的架构史：

（1）单体架构：单体架构在小微企业比较常见，典型代表就是一个应用、一个数据库、一个Web容器就可以跑起来，就是标准的单体架构。

（2）垂直架构：在单体架构发展一段时间后，公司的业务模式得到了认可，交易量也慢慢的大起来，这时候有些企业为了应对更大的流量，就会对原有的业务进行拆分，比如说：后台系统、前端系统、交易系统等。在这一阶段往往会将系统分为不同的层级，每个层级有对应的职责，UI 层负责和用户进行交互、业务逻辑层负责具体的业务功能、数据库层负责和上层进行数据交换和存储。这就是垂直架构。

（3）服务化架构：如果公司进一步的做大，垂直子系统会变的越来越多，系统和系统之间的调用关系呈指数上升的趋势。在这样的背景下，很多公司都会考虑服务的SOA化。SOA代表面向服务的架构，将应用程序按照不同的职责划分为不同的模块，不同的模块直接通过特定的协议和接口进行交互。这样将整个系统切分成很多单个组件服务来完成请求，当流量过大时通过水平扩展相应的组件来支撑，所有的组件通过交互来满足整体的业务需求。SOA服务化的优点是，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是SOA的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。服务化架构是一套松耦合的架构，服务的拆分原则是服务内部高内聚，服务之间低耦合。在这一阶段往往会将系统分为不同的层级，每个层级有对应的职责，UI 层负责和用户进行交互、业务逻辑层负责具体的业务功能、数据库层负责和上层进行数据交换和存储。

Spring提供了一系列工具用来实现SOA架构，可以帮助开发人员迅速搭建

分布式系统中的公共组件（比如：配置管理，服务发现，断路器，智能路由，微代理，控制总线，一次性令牌，全局锁，主节点选举，分布式 session，集群状态）。协调分布式环境中各个系统，为各类服务提供模板性配置。使用 Spring Cloud，开发人员可以搭建实现了这些样板的应用，并且在任何分布式环境下都能工作得非常好，小到笔记本电脑，大到数据中心和云平台。

（一）服务注册

Spring Cloud 官网的定义比较抽象，我们可以从简单的东西开始。Spring Cloud 是基于 Spring Boot 的，最适合用于管理 Spring Boot 创建的各个微服务应用。要管理分布式环境下的各个 Spring Boot 微服务，必然存在服务的注册问题。所以我们先从服务的注册谈起。既然是注册，必然有个管理注册中心的服务器，各个在 Spring Cloud 管理下的 Spring Boot 应用就是需要注册的 client Spring Cloud 使用 erureka server，然后所有需要访问配置文件的应用都作为一个 erureka client 注册上去。eureka 是一个高可用的组件，它没有后端缓存，每一个实例注册之后需要向注册中心发送心跳，在默认情况下 erureka server 也是一个 eureka client ,必须要指定一个 server。

（二）配置管理

使用 Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念映射与 Spring Environment 和 PropertySource 抽象相同，因此它们与 Spring 应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用 git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。很容易添加替代实现，并使用 Spring 配置将其插入，比如我们要搭建一个网站，需要配置数据库连接，指定数据库服务器的 IP 地址，数据库名称，用户名和口令等信息。通常的方法，我们可以在一个配置文件中定义这些信息，或者开发一个页面专门配置这些东西。只有一个 web 服务器

的时候，很方便。但假如需要搭建同多台服务器时，当然可以每台服务器做同样配置，但维护和同步会很麻烦。我理解的配置服务至少有两种不同场景：

（1）多个客户使用同一配置：比如，多台服务器组成的集群，假如后端使用同一数据库，那么每台服务器都是用相同的配置。

（2）不同客户使用不同的配置：比如典型的场景是，开发，测试，生产使用相同的系统，但使用不同的数据库。

如果有个统一的根本配置，是不是就很方便，一个可行的办法是，把这些配置文件放到一个共享存储（比如网络共享盘）中。这样只需要在共享存储修改一个或多个配置文件就可以了。但共享文件的方式受到具体布署环境的限制，很多时候很难达到多台 Web 服务器共享同一个存储硬盘。共享盘的缺点是资源定位比较困难，Spring Cloud 的解决方案是：将这些配置文件放到版本管理服务器里面。Spring Cloud 缺省配置使用 GIT 中。所有 Web 服务均从 GIT 中获取这些配置文件。由于 GIT 服务器与具体 Web 服务器之间不需要共享存储，只要网络可达就行，从而可以实现 Web 服务于配置信息的存放位置的解耦。

（三）负载均衡

负载均衡可分为服务端负载均衡和客户端负载均衡，服务端负载均衡完全由服务器处理，客户端不需要做任何事情。而客户端负载均衡技术，客户端需要维护一组服务器引用，每次客户端向服务端发请求的时候，为了实现负载均衡，Ribbon 组件会开始工作，它会根据算法主动选中一个服务节点。常用的负载均衡算法有：Round Robbin, Random，Hash，StaticWeighted 等。Spring 提供两辆种服务调度方式：Ribbon+restful 和 Feign。Ribbon 就是一个基于客户端的负载均衡器，Ribbon 提供了很多在 HTTP 和 TCP 客户端之上的控制。

（四）Feign 组件

首先简单解释一下声明式实现：要做一件事，需要知道三个要素，where, what, how。即在哪里（ where）用什么办法（how）做什么（what）。什么时候做（when）我们纳入 how 的范畴。

（1）编程式实现：每一个要素（where，what，how）都需要用具体代码实现来表示。传统的方式一般都是编程式实现，业务开发者需要关心每一处逻辑

（2）声明式实现：只需要声明在哪里（where）做什么（what），而无需关心如何实现（how）。Spring 的 AOP 就是一种声明式实现，比如网站检查是否登录，开发页面逻辑的时候，只需要通过 AOP 配置声明加载页面（where）需要做检查用户是否登录（what），而无需关心如何检查用户是否登录（how)。如何检查这个逻辑由 AOP 机制去实现，而 AOP 的登录检查实现机制与正在开发页面的逻辑本身是无关的。

在 Spring Cloud Netflix 栈中，各个微服务都是以 HTTP 接口的形式暴露自身服务的，因此在调用远程服务时就必须使用 HTTP 客户端。Feign 就是 Spring Cloud 提供的一种声明式 REST 客户端。可以通过 Feign 访问调用远端微服务提供的 REST 接口。现在我们就用 Feign 来调用 SERVICE-HELLOWORLD 暴露的 REST 接口，以获取到“Hello World”信息。在使用 Feign 时，Spring Cloud 集成了 Ribbon 和 Eureka 来提供 HTTP 客户端的负载均衡。

（五）断路器

在分布式环境下，特别是微服务结构的分布式系统中，一个软件系统调用另外一个远程系统是非常普遍的。这种远程调用的被调用方可能是另外一个进程，或者是跨网路的另外一台主机, 这种远程的调用和进程的内部调用最大的区别是，远程调用可能会失败，或者挂起而没有任何回应，直到超时。更坏的情况是，如果有多个调用者对同一个挂起的服务进行调用，那么就很有可能的是一个服务的超时等待迅速蔓延到整个分布式系统，引起连锁反应，从而消耗掉整个分布式系统大量资源。最终可能导致系统瘫痪。

断路器（Circuit Breaker）模式就是为了防止在分布式系统中出现这种瀑布似的连锁反应导致的灾难。一旦某个电器出问题，为了防止灾难，电路的保险丝就会熔断。断路器类似于电路的保险丝，实现思路非常简单，可以将需要保护的远程服务嗲用封装起来，在内部监听失败次数， 一旦失败次数达到某阀值后，

所有后续对该服务的调用，断路器截获后都直接返回错误到调用方，而不会继续调用已经出问题的服务， 从而达到保护调用方的目的，整个系统也就不会出现因为超时而产生的瀑布式连锁反应。

（六）Zuul 组件

微服务架构中，需要几个关键的组件，服务注册与发现、服务消费、负载均衡、断路器、智能路由、配置管理等，由这几个组件可以组建一个简单的微服务架构。客户端的请求首先经过负载均衡（zuul、Ngnix），再到达服务网关（zuul 集群），然后再到具体的服务，服务统一注册到高可用的服务注册中心集群，服务的所有的配置文件由配置服务管理（下一篇文章讲述），配置服务的配置文件放在 Git 仓库，方便开发人员随时改配置。Zuul 的主要功能是路由和过滤器。路由功能是微服务的一部分，比如 / api/user 映射到 user 服务，/api/shop 映射到 shop 服务。zuul 实现了负载均衡。

试想一下，在分布式系统中，如果存在很多个客户端都需要刷新改配置，通过这种方式去刷新也是一种非常痛苦的事情。那有没有什么办法让系统自动完成呢？之前我们提到用 githook 或者 jenkins 等外部工具来触发。现在说另外一种思路， 如果 refresh 命令可以发送给 config server，然后 config server 自动通知所有 config client，那么就可以大大简化配置刷新工作。这样，虽然仍然需要通过 refresh 命令触发，但通过 webhook 等钩子方式，我们只需要将关联的命令挂到配置中心上，而不需要每个配置客户端都去关联。现在，我们通过整合消息队列 Rabbitmq 来完成这件事。我们的目标是，当 git 仓库中的配置一旦更改，将 refresh 命令发送给配置中心，然后配置中心通过消息队列，自动通知所有使用了该配置的刷新各自配置。