从架构演进的角度聊聊 Spring Cloud 都做了些什么？

Spring Cloud作为一套微服务治理的框架，几乎考虑到了微服务治理的方方面面，之前也写过一些关于Spring Cloud文章，主要偏重各组件的使用，本次分享主要解答这两个问题：Spring Cloud在微服务的架构中都做了哪些事情？Spring Cloud提供的这些功能对微服务的架构提供了怎样的便利？

我们先来简单回顾一下，我们以往互联网架构的发展情况：

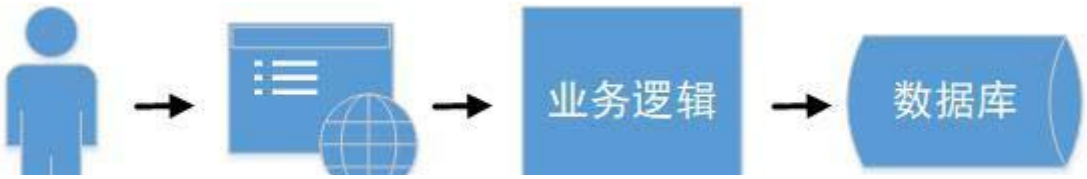
传统架构发展史

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，比如我们开发的开源软件云收藏，就是标准的单体架构。

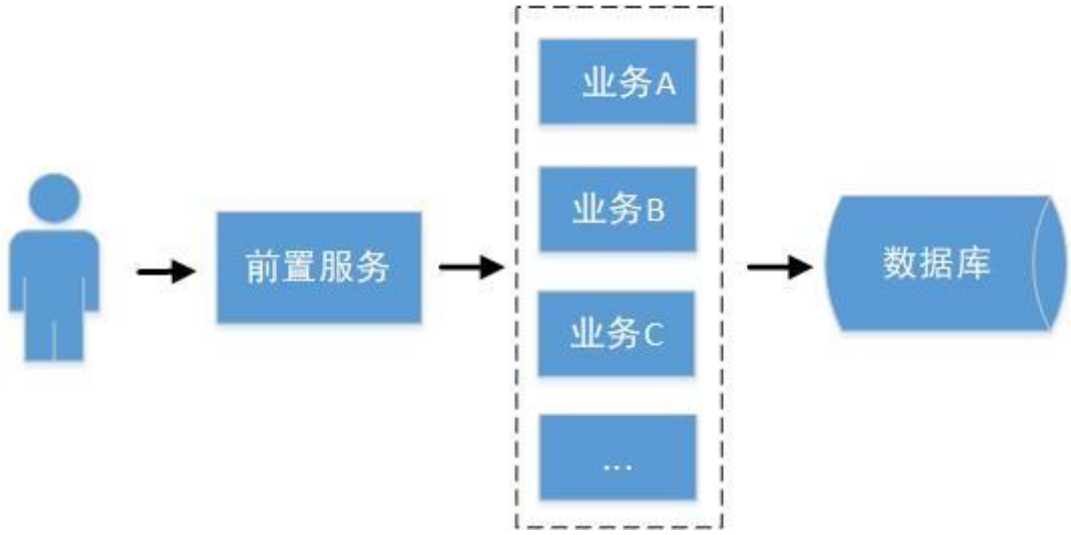
在两种情况下可能会选择单体架构：一是在企业发展的初期，为了保证快速上线，采用此种方案较为简单灵活；二是传统企业中垂直度较高，访问压力较小的业务。在这种模式下对技术要求较低，方便各层次开发人员接手，也能满足客户需求。

下面是单体架构的架构图：



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。

下面是垂直架构的架构图：



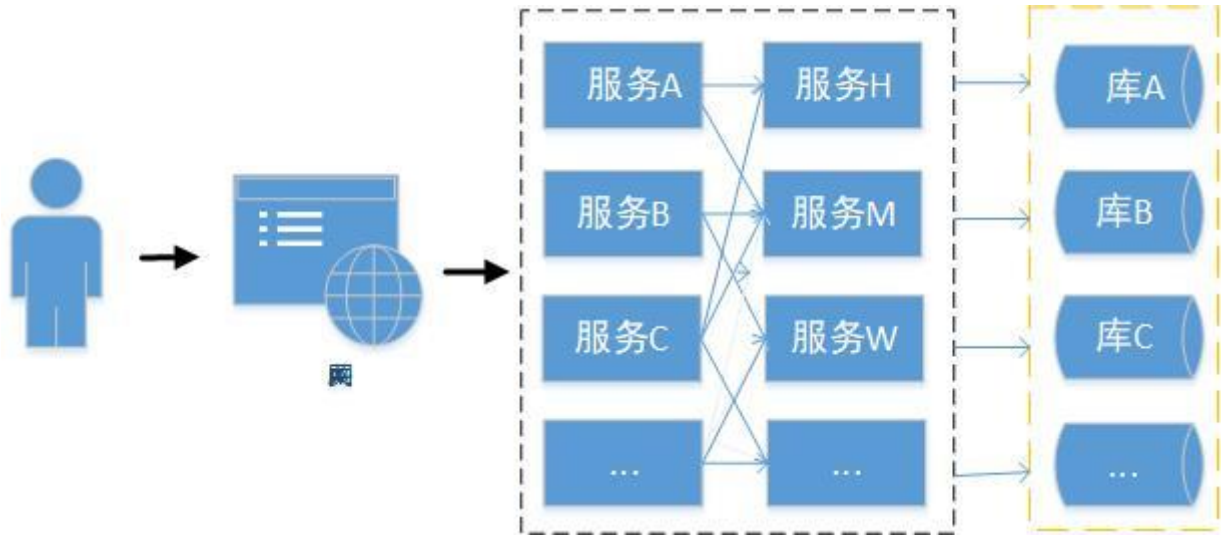
在这个阶段SSH（struts+spring+hibernate）是项目的关键技术，Struts负责web层逻辑控制、Spring负责业务层管理Bean、Hibernate负责数据库操作进行封装，持久化数据。

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，将应用程序根据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。这样使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



在这个阶段可以使用WebService或者dubbo来服务治理。

SOA和微服务架构

SOA和微服务的区别

其实服务化架构已经可以解决大部分企业的需求了，那么我们为什么要研究微服务呢？先说说它们的区别；

- 微服务架构强调业务系统需要彻底的组件化和服务化，一个组件就是一个产品，可以独立对外提供服务
- 微服务不再强调传统SOA架构里面比较重的ESB企业服务总线
- 微服务强调每个微服务都有自己独立的运行空间，包括数据库资源。
- 微服务架构本身来源于互联网的思路，因此组件对外发布的服务强调了采用HTTP Rest API的方式来进行
- 微服务的切分粒度会更小

> 总结:微服务架构是 SOA 架构思想的一种扩展，更加强调服务个体的独立性、拆分

它的特性

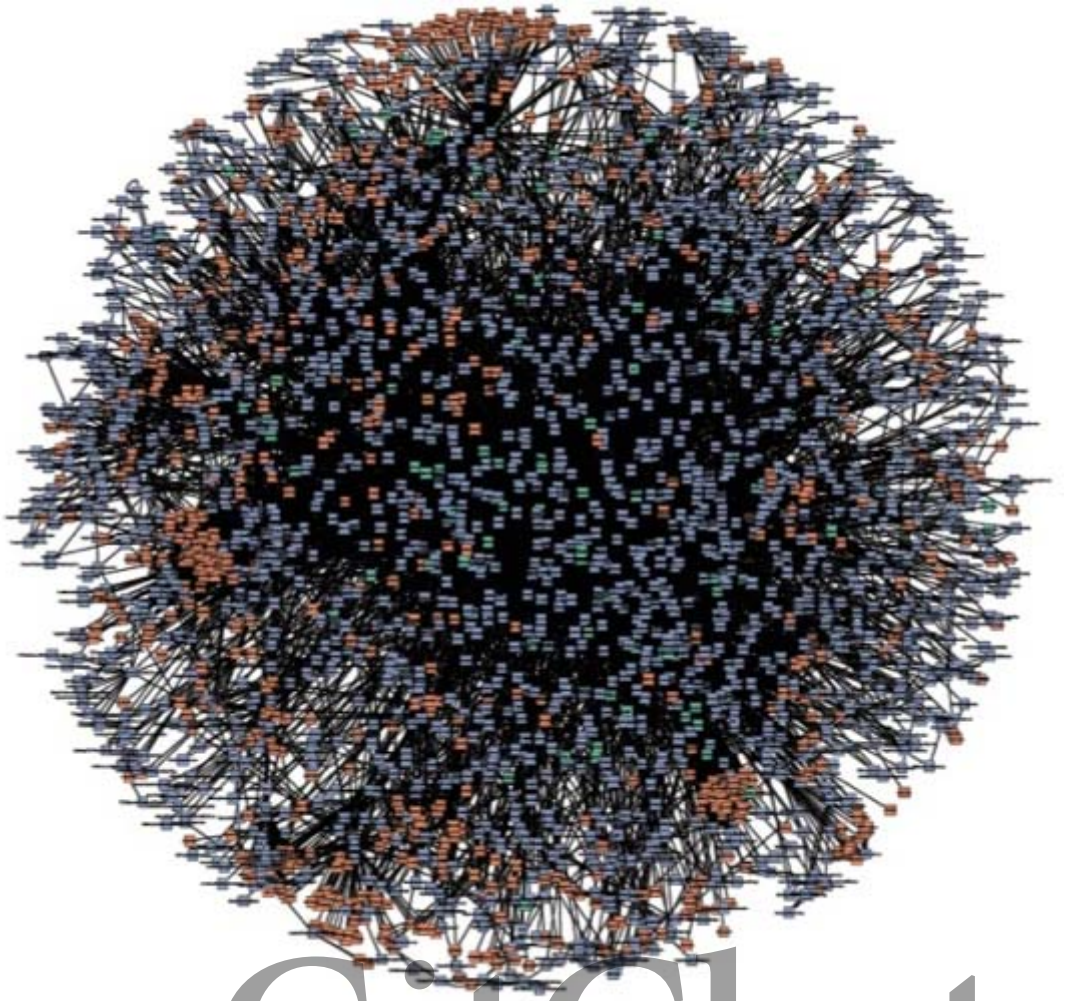
以下为Spring Cloud的核心特性：

- 分布式/版本化配置
- 服务注册和发现
- 路由
- 服务和服务之间的调用
- 负载均衡
- 断路器
- 分布式消息传递

这些特性都是由不同的组件来完成，在架构的演进过程中扮演着重要的角色，接下来我们一起看看。

微服务架构

Spring Cloud解决的第一个问题就是：服务与服务之间的解耦。很多公司在业务高速发展的时候，服务组件也会相应的不断增加。服务和服务之间有着复杂的相互调用关系，经常有服务A调用服务B，服务B调用服务C和服务D …，随着服务化组件的不断增多，服务之间的调用关系成指数级别的增长，极端情况下就如下图所示：



这样最容易导致的情况就是牵一发而动全身。经常出现由于某个服务更新而没有通知到其它服务，导致上线后惨案频发。这时候就应该进行服务治理，将服务之间的直接依赖转化为服务对服务中心的依赖。Spring Cloud 核心组件Eureka就是解决这类问题。

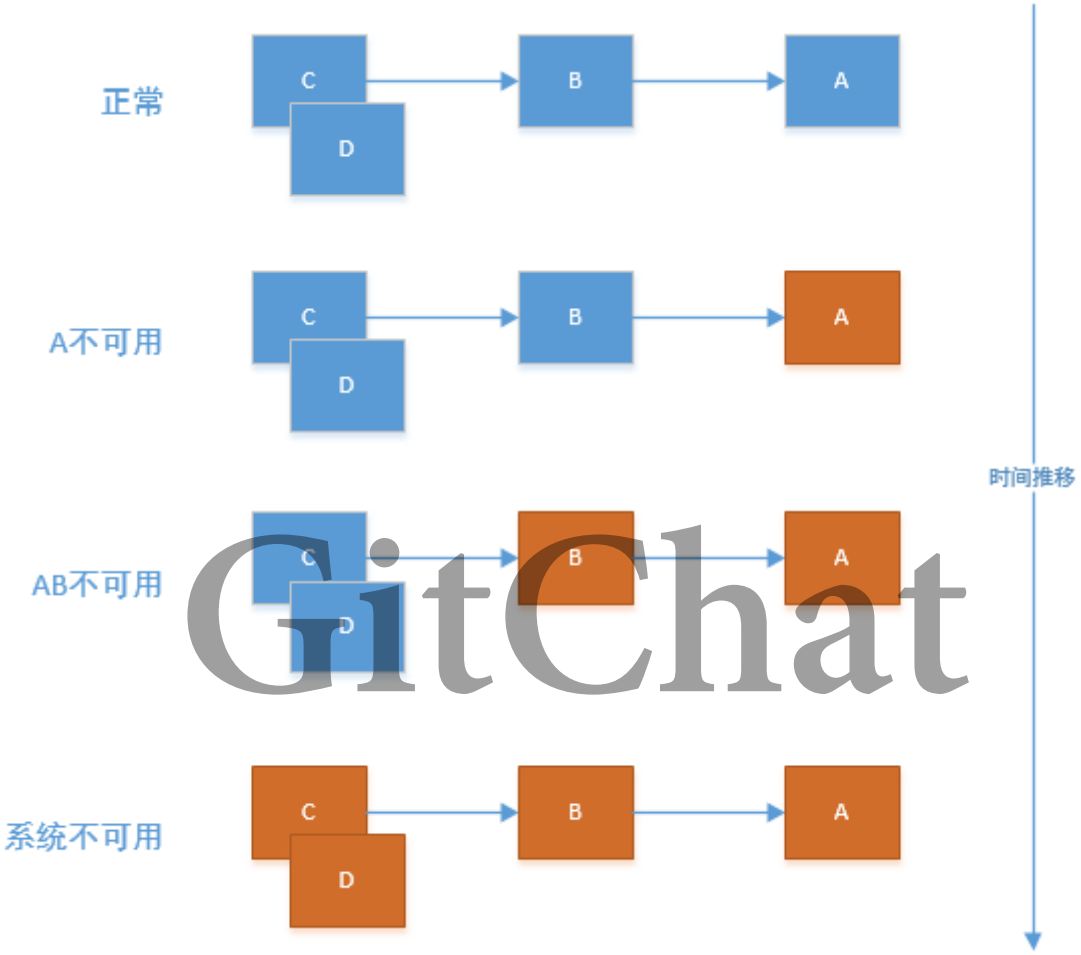
Eureka

Eureka是Netflix开源的一款提供服务注册和发现的产品，它提供了完整的Service Registry和Service Discovery实现。也是Spring Cloud体系中最重要最核心的组件之一。

用大白话讲，Eureka就是一个服务中心，将所有的可以提供的服务都注册到它这里来管

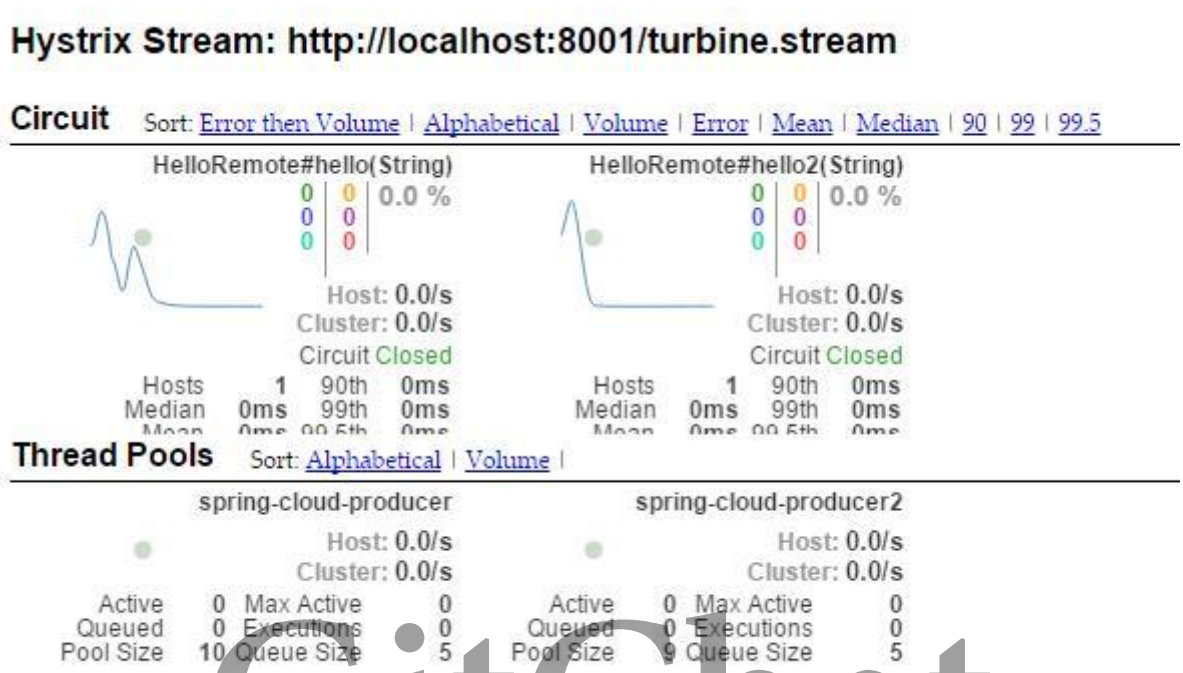
在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



在这种情况下就需要整个服务机构具有故障隔离的功能，避免某一个服务挂掉影响全

Hystrix-dashboard是一款针对Hystrix进行实时监控的工具，通过Hystrix Dashboard我们可以直观地看到各Hystrix Command的请求响应时间, 请求成功率等数据。但是只使用Hystrix Dashboard的话, 你只能看到单个应用内的服务信息, 这明显不够. 我们需要一个工具能让我们汇总系统内多个服务的数据并显示到Hystrix Dashboard上, 这个工具就是Turbine.
监控的效果图如下：



配置中心

随着微服务不断的增多，每个微服务都有自己对应的配置文件。在研发过程中有测试环境、UAT环境、生产环境，因此每个微服务又对应至少三个不同环境的配置文件。这么多的配置文件，如果需要修改某个公共服务的配置信息，如：缓存、数据库等，难免会产生混乱，这个时候就需要引入Spring Cloud另外一个组件：Spring Cloud Config。

Spring Cloud Config

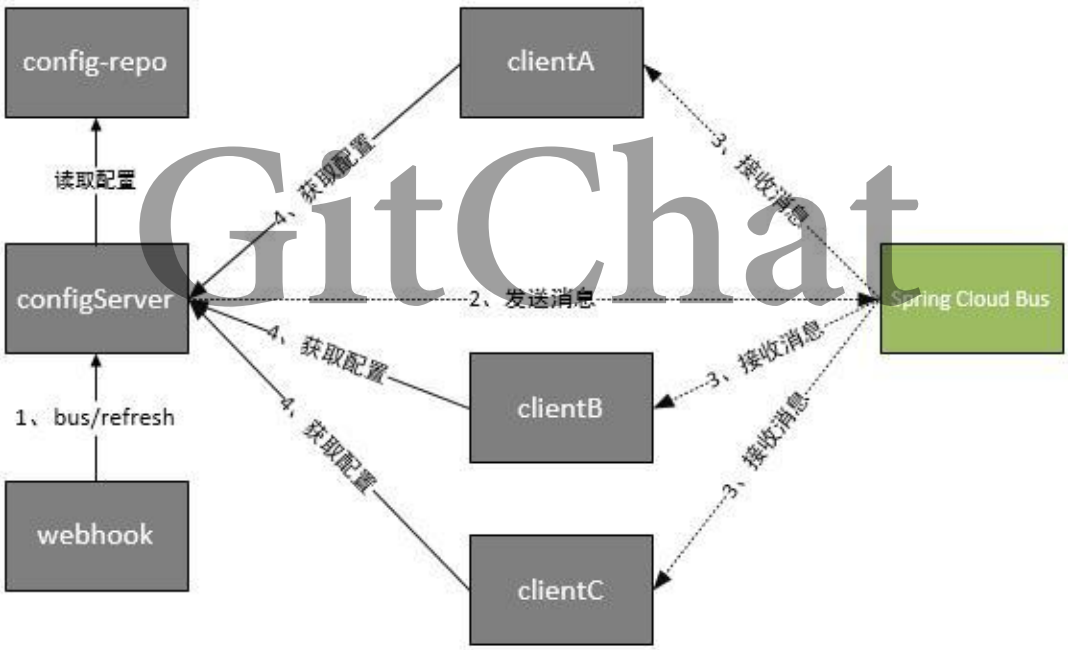
来支持配置中心高可用性。

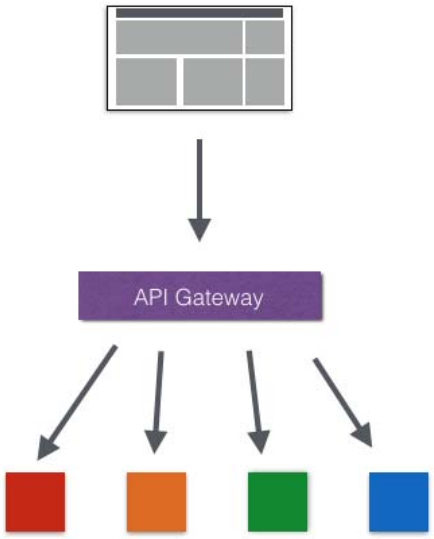
Spring Cloud Bus

上面的Refresh方案虽然可以解决单个微服务运行期间重载配置信息的问题，但是在真正的实践生产中，可能会有N多的服务需要更新配置，如果每次依靠手动Refresh将是一个巨大的工作量，这时候Spring Cloud提出了另外一个解决方案：Spring Cloud Bus

Spring Cloud Bus通过轻量消息代理连接各个分布的节点。这会用在广播状态的变化（例如配置变化）或者其它的消息指令中。Spring Cloud Bus的一个核心思想是通过分布式的启动器对Spring Boot应用进行扩展，也可以用来建立一个或多个应用之间的通信频道。目前唯一实现的方式是用AMQP消息代理作为通道。

Spring Cloud Bus是轻量级的通讯组件，也可以用在其它类似的场景中。有了Spring Cloud Bus之后，当我们改变配置文件提交到版本库中时，会自动的触发对应实例的Refresh，具体的工作流程如下：



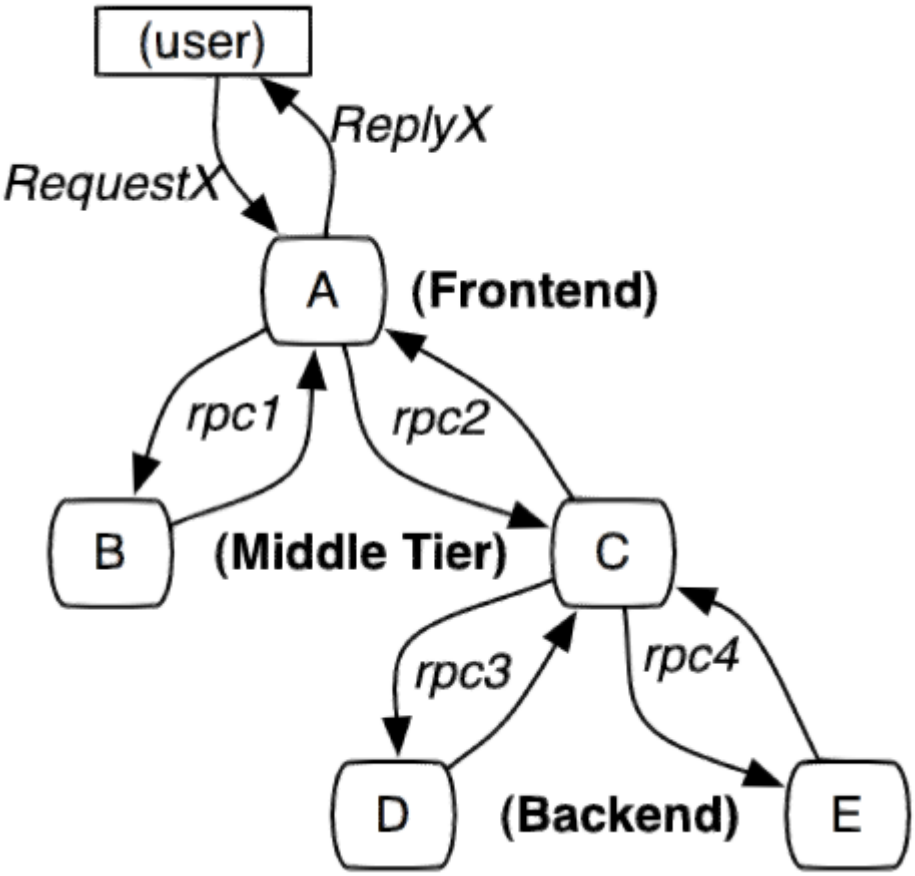


Spring Cloud体系中支持API Gateway落地的技术就是Zuul。Spring Cloud Zuul路由是微服务架构中不可或缺的一部分，提供动态路由，监控，弹性，安全等的边缘服务。Zuul是Netflix出品的一个基于JVM路由和服务端的负载均衡器。

它的具体作用就是服务转发，接收并转发所有内外部的客户端调用。使用Zuul可以作为资源的统一访问入口，同时也可以在网关做一些权限校验等类似的功能。

链路跟踪

随着服务的越来越多，对调用链的分析会越来越复杂，如服务之间的调用关系、某个请求对应的调用链、调用之间消费的时间等，对这些信息进行监控就成为一个问题。在实际的使用中我们需要监控服务和服务之间通讯的各项指标，这些数据将是我们改进系统架构的主要依据。因此分布式的链路跟踪就变的非常重要，Spring Cloud也给出了具体的解决方案：Spring Cloud Sleuth和Zipkin

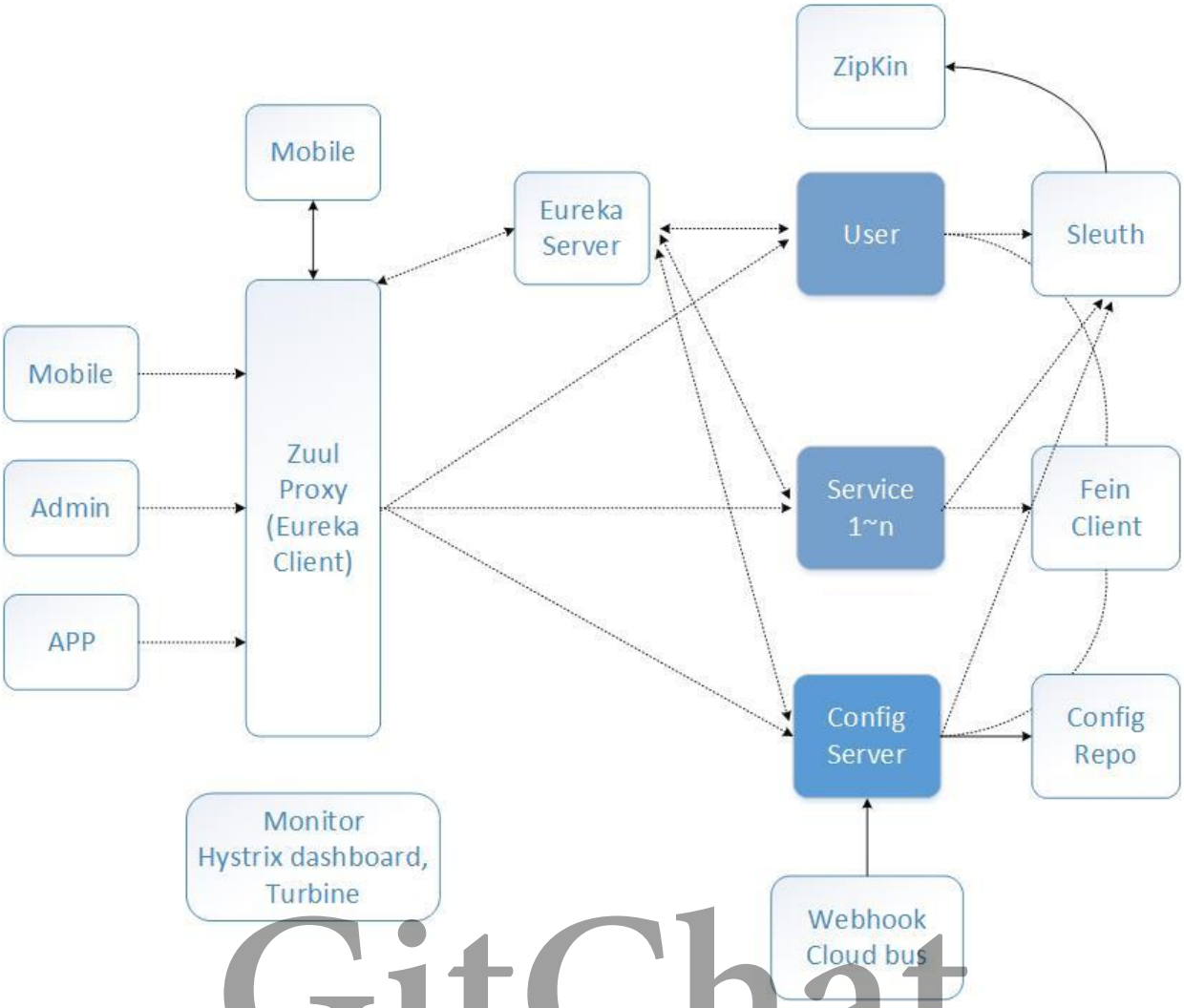


Spring Cloud Sleuth为服务之间调用提供链路追踪。通过Sleuth可以很清楚的了解到一个服务请求经过了哪些服务，每个服务处理花费了多长时间。从而让我们可以很方便的理清各微服务间的调用关系。

Zipkin是Twitter的一个开源项目，允许开发者收集 Twitter 各个服务上的监控数据，并提供查询接口

总结

我们从整体上来看一下Spring Cloud各个组件如何来配套使用：



从上图可以看出Spring Cloud各个组件相互配合，合作支持了一套完整的微服务架构。

- 其中Eureka负责服务的注册与发现，很好将各服务连接起来
- Hystrix 负责监控服务之间的调用情况，连续多次失败进行熔断保护。
- Hystrix dashboard,Turbine 负责监控 Hystrix的熔断情况，并给予图形化的展示
- Spring Cloud Config 提供了统一的配置中心服务
- 当配置文件发生变化的时候， Spring Cloud Bus 负责通知各服务去获取最新的配置信息
- 所有对外的请求和服务，我们都通过Zuul来进行转发，起到API网关的作用