# 实施微服务，我们需要哪些基础框架？

作者 **杨波** 发布于 2015年12月1日 | 3 **讨论**

微服务(MicroServices)架构是当前互联网业界的一个技术热点，圈里有不少同行朋友当前有计划在各自公司开展微服务化体系建设，他们都有相同的疑问：一个微服务架构有哪些技术关注点(technical concerns)？需要哪些基础框架或组件来支持微服务架构？这些框架或组件该如何选型？笔者之前在两家大型互联网公司参与和主导过大型服务化体系和框架建设，同时在这块也投入了很多时间去学习和研究，有一些经验和学习心得，可以和大家一起分享。

## 服务注册、发现、负载均衡和健康检查

和单块(Monolithic)架构不同，微服务架构是由一系列职责单一的细粒度服务构成的分布式网状结构，服务之间通过轻量机制进行通信，这时候必然引入一个服务注册发现问题，也就是说服务提供方要注册通告服务地址，服务的调用方要能发现目标服务，同时服务提供方一般以集群方式提供服务，也就引入了负载均衡和健康检查问题。根据负载均衡LB所在位置的不同，目前主要的服务注册、发现和负载均衡方案有三种：

第一种是集中式LB方案，如下图Fig 1，在服务消费者和服务提供者之间有一个独立的LB，LB通常是专门的硬件设备如F5，或者基于软件如LVS，HAproxy等实现。LB上有所有服务的地址映射表，通常由运维配置注册，当服务消费方调用某个目标服务时，它向LB发起请求，由LB以某种策略（比如Round-Robin）做负载均衡后将请求转发到目标服务。LB一般具备健康检查能力，能自动摘除不健康的服务实例。服务消费方如何发现LB呢？通常的做法是通过DNS，运维人员为服务配置一个DNS域名，这个域名指向LB。

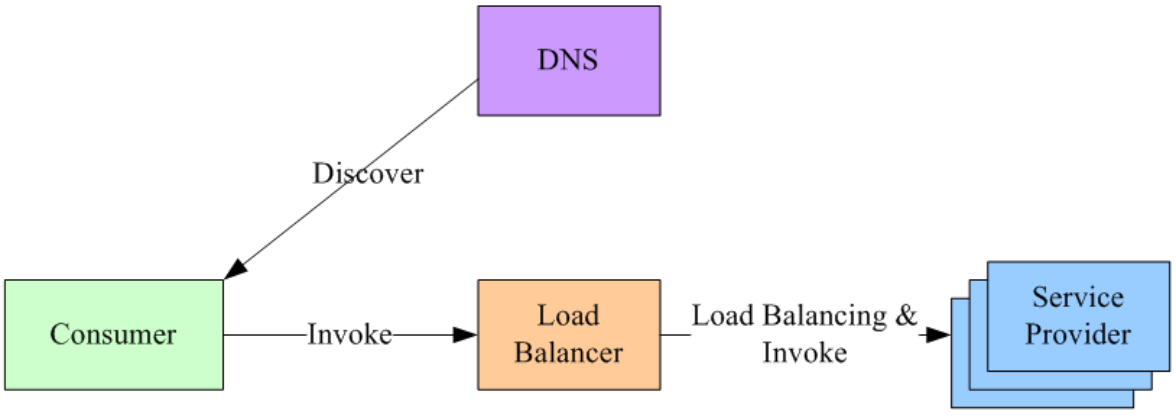


Fig 1, 集中式LB方案

集中式LB方案实现简单，在LB上也容易做集中式的访问控制，这一方案目前还是业界主流。集中式LB的主要问题是单点问题，所有服务调用流量都经过LB，当服务数量和调用量大的时候，LB容易成为瓶颈，且一旦LB发生故障对整个系统的影响是灾难性的。另外，LB在服务消费方和服务提供方之间增加了一跳(hop)，有一定性能开销。

第二种是进程内LB方案，针对集中式LB的不足，进程内LB方案将LB的功能以库的形式集成到服务消费方进程里头，该方案也被称为软负载(Soft Load Balancing)或者客户端负载方案，下图Fig 2展示了这种方案的工作原理。这一方案需要一个服务注册表(Service Registry)配合支持服务自注册和自发现，服务提供方启动时，首先将服务地址注册到服务注册表（同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查），服

务消费方要访问某个服务时，它通过内置的LB组件向服务注册表查询（同时缓存并定期刷新）目标服务地址列表，然后以某种负载均衡策略选择一个目标服务地址，最后向目标服务发起请求。这一方案对服务注册表的可用性(Availability)要求很高，一般采用能满足高可用分布式一致的组件（例如Zookeeper, Consul, Etcd等）来实现。

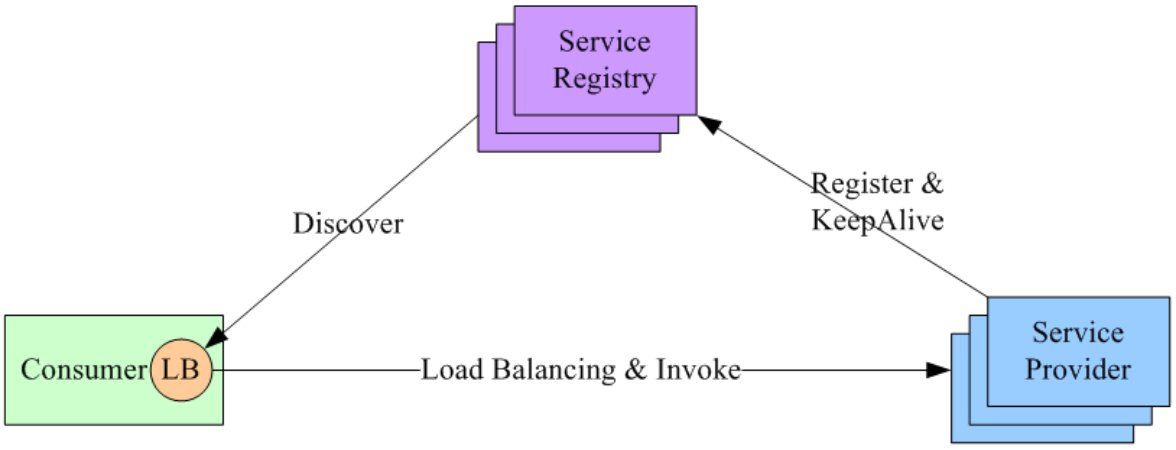


Fig 2, 进程内LB方案

进程内LB方案是一种分布式方案，LB和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。但是，该方案以客户库(Client Library)的方式集成到服务调用方进程里头，如果企业内有多种不同的语言栈，就要配合开发多种不同的客户端，有一定的研发和维护成本。另外，一旦客户端跟随服务调用方发布到生产环境中，后续如果要对客户库进行升级，势必要求服务调用方修改代码并重新发布，所以该方案的升级推广有不小的阻力。

进程内LB的案例是Netflix的开源服务框架，对应的组件分别是：Eureka服务注册表，Karyon服务端框架支持服务自注册和健康检查，Ribbon客户端框架支持服务自发现和软路由。另外，阿里开源的服务框架Dubbo也是采用类似机制。

第三种是主机独立LB进程方案，该方案是针对第二种方案的不足而提出的一种折中方案，原理和第二种方案基本类似，不同之处是，他将LB和服务发现功能从进程内移出来，变成主机上的一个独立进程，主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立LB进程做服务发现和负载均衡，见下图Fig 3。

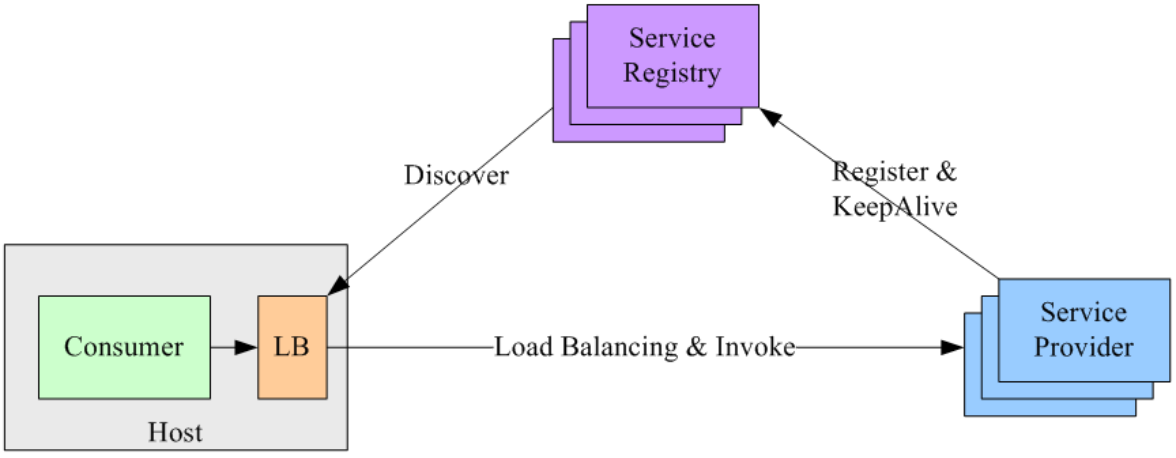

Fig 3 主机独立LB进程方案

该方案也是一种分布式方案，没有单点问题，一个LB进程挂了只影响该主机上的服务调用方，服务调用方和LB之间是进程内调用，性能好，同时，该方案还简化了服务调用方，不需要为不同语言开发客户库，LB的升级不需要服务调用方改代码。该方案的不足是部署较复杂，环节多，出错调试排查问题不方便。

该方案的典型案例是Airbnb的SmartStack服务发现框架，对应组件分别是：Zookeeper作为服务注册表，Nerve独立进程负责服务注册和健康检查，Synapse/HAproxy独立进程负责服务发现和负载均衡。Google最新推出的基于容器的PaaS平台Kubernetes，其内部服务发现采用类似的机制。

## 服务前端路由

微服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到，这就涉及服务的前端路由，对应的组件是服务网关(Service Gateway)，见图Fig 4，网关是连接企业内部和外部系统的一道门，有如下关键作用：

1. 服务反向路由，网关要负责将外部请求反向路由到内部具体的微服务，这样虽然企业内部是复杂的分布式微服务结构，但是外部系统从网关上看到的就像是一个统一的完整服务，网关屏蔽了后台服务的复杂性，同时也屏蔽了后台服务的升级和变化。
2. 安全认证和防爬虫，所有外部请求必须经过网关，网关可以集中对访问进行安全控制，比如用户认证和授权，同时还可以分析访问模式实现防爬虫功能，网关是连接企业内外系统的安全之门。
3. 限流和容错，在流量高峰期，网关可以限制流量，保护后台系统不被大流量冲垮，在内部系统出现故障时，网关可以集中做容错，保持外部良好的用户体验。
4. 监控，网关可以集中监控访问量，调用延迟，错误计数和访问模式，为后端的性能优化或者扩容提供数据支持。
5. 日志，网关可以收集所有的访问日志，进入后台系统做进一步分析。

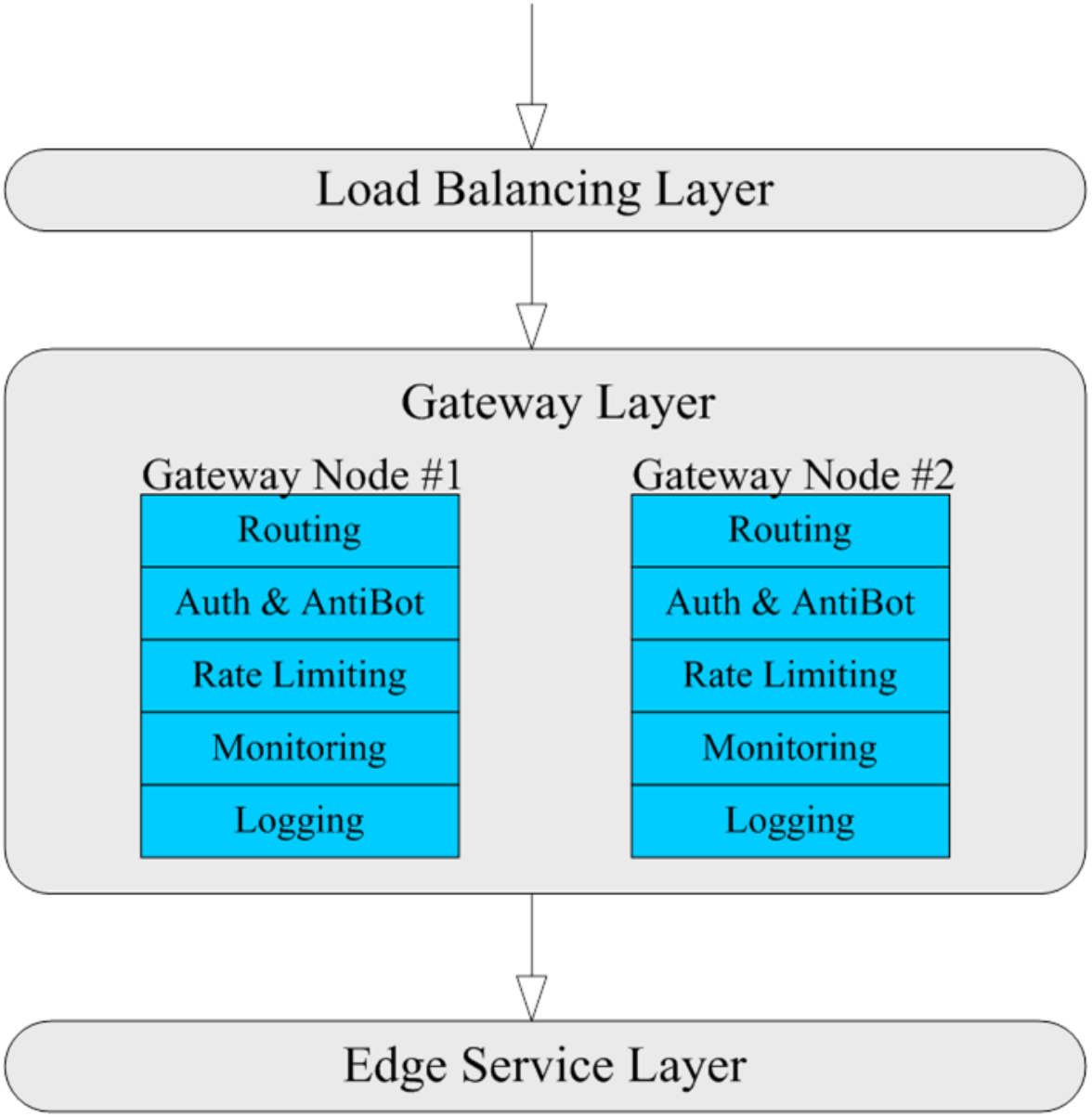


Fig 4, 服务网关

除以上基本能力外，网关还可以实现线上引流，线上压测，线上调试(Surgical debugging)，金丝雀测试(Canary Testing)，数据中心双活(Active-Active HA)等高级功能。

网关通常工作在7层，有一定的计算逻辑，一般以集群方式部署，前置LB进行负载均衡。

开源的网关组件有Netflix的Zuul，特点是动态可热部署的过滤器(filter)机制，其它如HAproxy，Nginx等都可以扩展作为网关使用。

在介绍过服务注册表和网关等组件之后，我们可以通过一个简化的微服务架构图(Fig 5)来更加直观地展示整个微服务体系内的服务注册发现和路由机制，该图假定采用进程内LB服务发现和负载均衡机制。在下图Fig 5的微服务架构中，服务简化为两层，后端通用服务（也称中间层服务Middle Tier Service）和前端服务（也称边缘服务Edge Service，前端服务的作用是对后端服务做必要的聚合和裁剪后暴露给外部不同的设备，如PC，Pad或者Phone）。后端服务启动时会将地址信息注册到服务注册表，前端服务通过查询服务注册表就可以发现然后调用后端服务；前端服务启动时也会将地址信息注册到服务注册表，这样网关通过查询服务注册表就可以将请求路由到目标前端服务，这样整个微服务体系的服务自注册自发现和软路由就通过服务注册表和网关串

联起来了。如果以面向对象设计模式的视角来看，网关类似Proxy代理或者Façade门面模式，而服务注册表和服务自注册自发现类似IoC依赖注入模式，微服务可以理解为基于网关代理和注册表IoC构建的分布式系统。

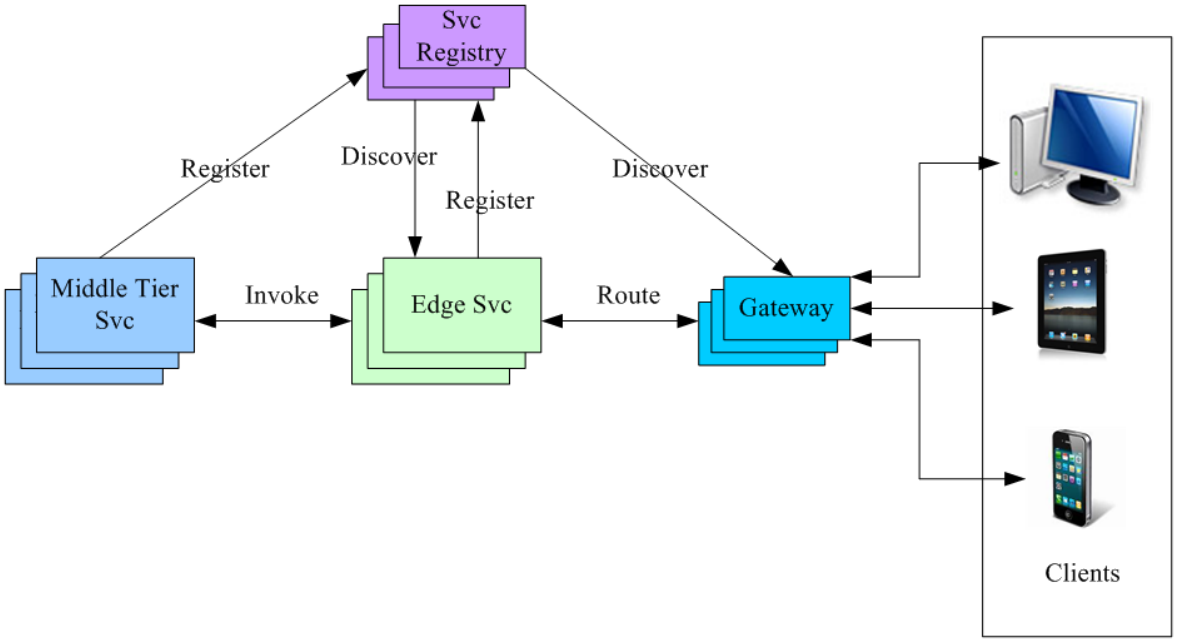


Fig 5, 简化的微服务架构图

## 服务容错

当企业微服务化以后，服务之间会有错综复杂的依赖关系，例如，一个前端请求一般会依赖于多个后端服务，技术上称为1 -> N扇出(见图Fig 6)。在实际生产环境中，服务往往不是百分百可靠，服务可能会出错或者产生延迟，如果一个应用不能对其依赖的故障进行容错和隔离，那么该应用本身就处在被拖垮的风险中。在一个高流量的网站中，某个单一后端一旦发生延迟，可能在数秒内导致所有应用资源(线程，队列等)被耗尽，造成所谓的雪崩效应(Cascading Failure，见图Fig 7)，严重时可致整个网站瘫痪。

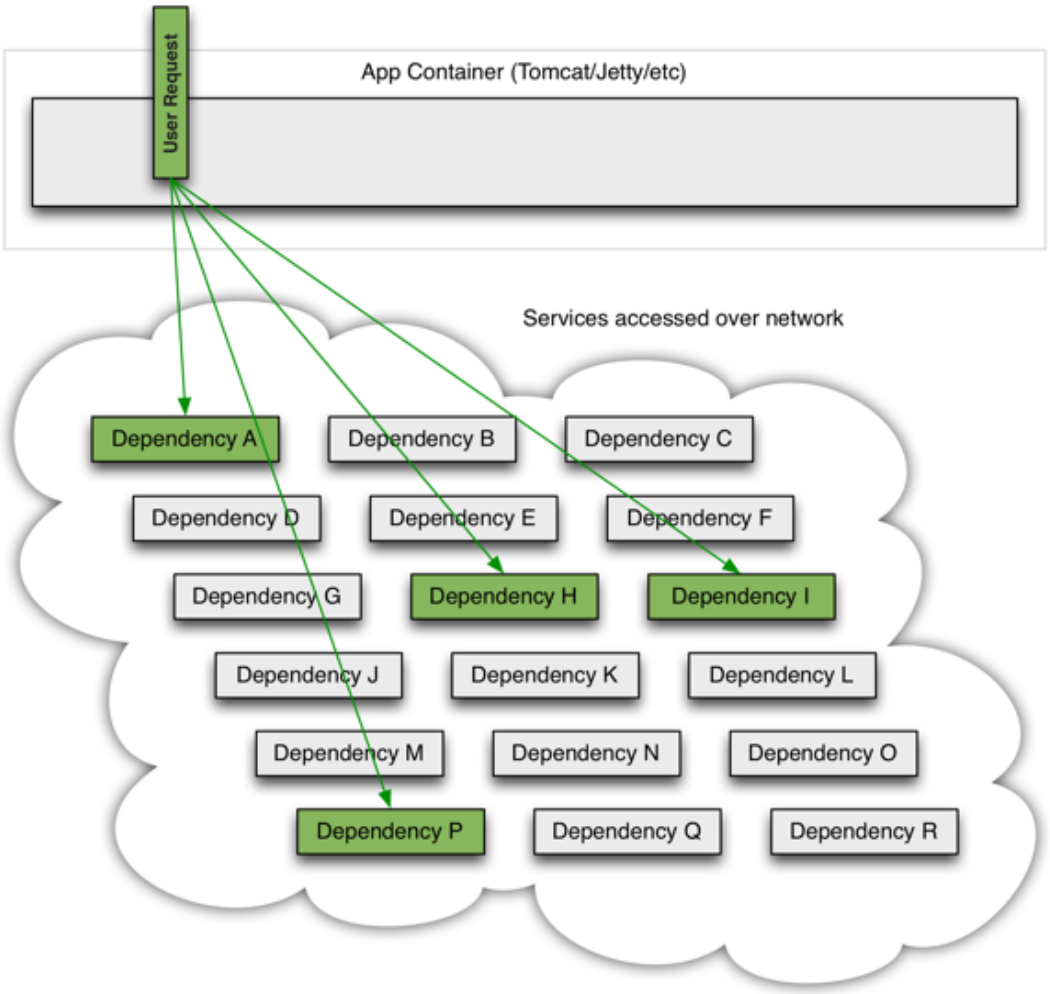
User Request
App Container (Tomcat/Jetty/etc)
Services accessed over network
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

Fig 6, 服务依赖

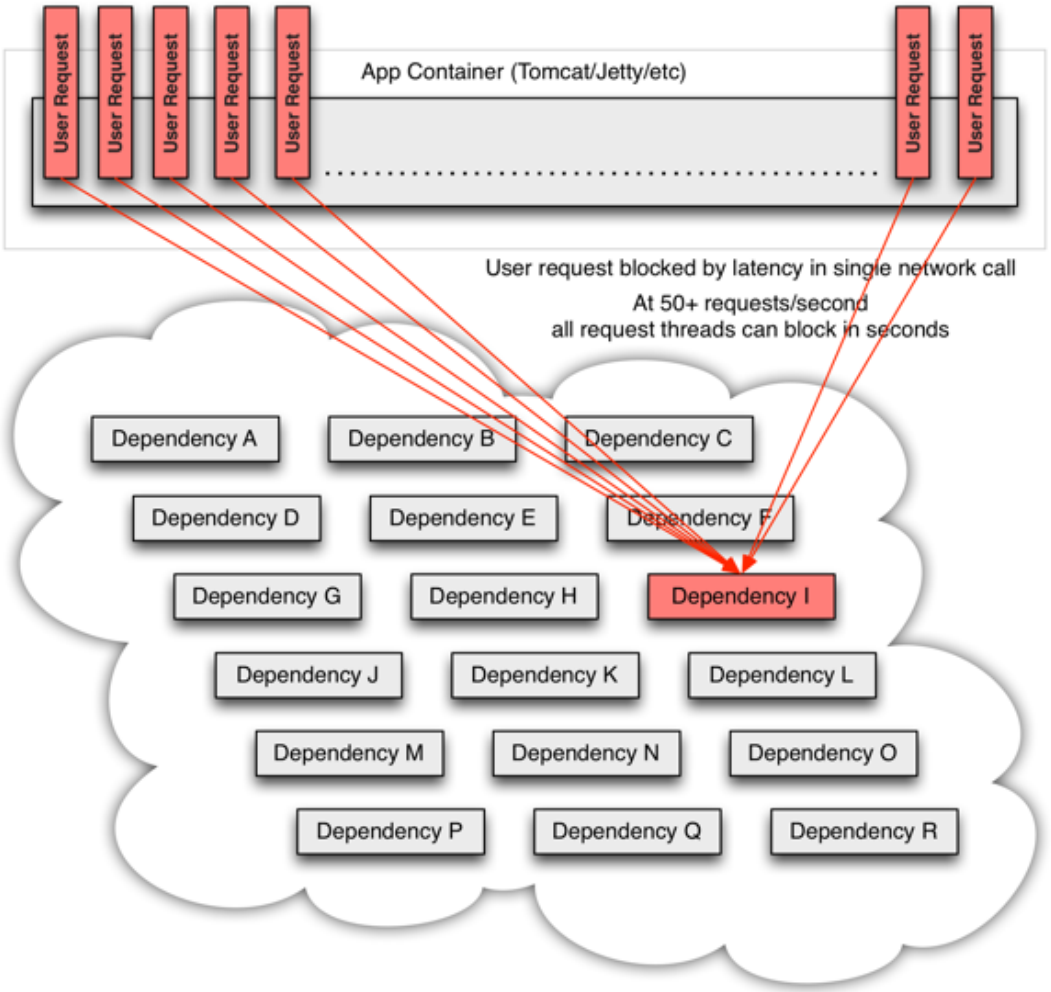


Fig 7, 高峰期单个服务延迟致雪崩效应

经过多年的探索和实践，业界在分布式服务容错一块探索出了一套有效的容错模式和最佳实践，主要包括：

1. 电路熔断器模式(Circuit Breaker Patten), 该模式的原理类似于家里的电路熔断器，如果家里的电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失。在分布式系统中应用电路熔断器模式后，当目标服务慢或者大量超时，调用方能够主动熔断，以防止服务被进一步拖垮；如果情况又好转了，电路又能自动恢复，这就是所谓的弹性容错，系统有自恢复能力。下图Fig 8是一个典型的具备弹性恢复能力的电路保护器状态图，正常状态下，电路处于关闭状态(Closed)，如果调用持续出错或者超时，电路被打开进入熔断状态(Open)，后续一段时间内的所有调用都会被拒绝(Fail Fast)，一段时间以后，保护器会尝试进入半熔断状态(Half-Open)，允许少量请求进来尝试，如果调用仍然失败，则回到熔断状态，如果调用成功，则回到电路闭合状态。

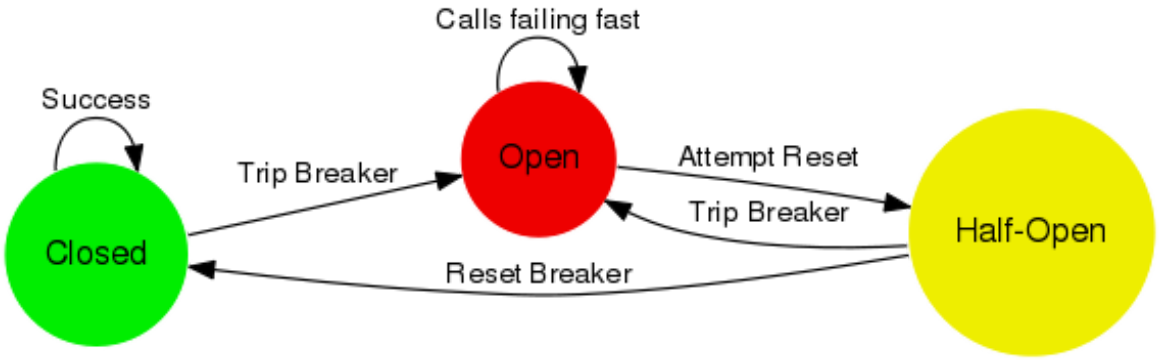

Fig 8, 弹性电路保护状态图

2. 舱壁隔离模式(Bulkhead Isolation Pattern)，顾名思义，该模式像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。线程隔离(Thread Isolation)就是舱壁隔离模式的一个例子，假定一个应用程序A调用了Svc1/Svc2/Svc3三个服务，且部署A的容器一共有120个工作线程，采用线程隔离机制，可以给对Svc1/Svc2/Svc3的调用各分配40个线程，当Svc2慢了，给Svc2分配的40个线程因慢而阻塞并最终耗尽，线程隔离可以保证给Svc1/Svc3分配的80个线程可以不受影响，如果没有这种隔离机制，当Svc2慢的时候，120个工作线程会很快全部被对Svc2的调用吃光，整个应用程序会全部慢下来。
3. 限流(Rate Limiting/Load Shedder)，服务总有容量限制，没有限流机制的服务很容易在突发流量(秒杀，双十一)时被冲垮。限流通常指对服务限定并发访问量，比如单位时间只允许100个并发调用，对超过这个限制的请求要拒绝并回退。
4. 回退(fallback)，在熔断或者限流发生的时候，应用程序的后续处理逻辑是什么？回退是系统的弹性恢复能力，常见的处理策略有，直接抛出异常，也称快速失败(Fail Fast)，也可以返回空值或缺省值，还可以返回备份数据，如果主服务熔断了，可以从备份服务获取数据。

Netflix将上述容错模式和最佳实践集成到一个称为Hystrix的开源组件中，凡是需要容错的依赖点(服务，缓存，数据库访问等)，开发人员只需要将调用封装在Hystrix Command里头，则相关调用就自动置于Hystrix的弹性容错保护之下。Hystrix组件已经在Netflix经过多年运维验证，是Netflix微服务平台稳定性和弹性的基石，正逐渐被社区接受为标准容错组件。

## 服务框架

微服务化以后，为了让业务开发人员专注于业务逻辑实现，避免冗余和重复劳动，规范研发提升效率，必然要将一些公共关注点推到框架层面。服务框架(Fig 9)主要封装公共关注点逻辑，包括：

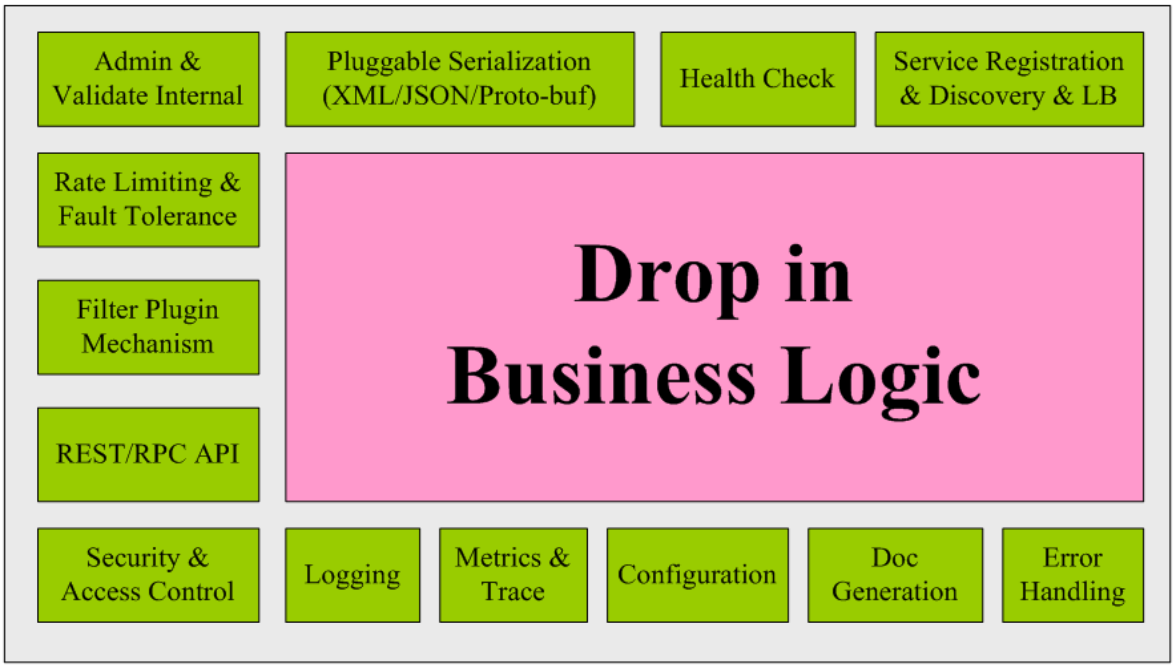


Fig 9, 服务框架

1. 服务注册、发现、负载均衡和健康检查，假定采用进程内LB方案，那么服务自注册一般统一做在服务器端框架中，健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制，服

务发现和负载均衡则集成在服务客户端框架中。

2. 监控日志，框架一方面要记录重要的框架层日志、metrics和调用链数据，还要将日志、metrics等接口暴露出来，让业务层能根据需要记录业务日志数据。在运行环境中，所有日志数据一般集中落地到企业后台日志系统，做进一步分析和处理。
3. REST/RPC和序列化，框架层要支持将业务逻辑以HTTP/REST或者RPC方式暴露出来，HTTP/REST是当前主流API暴露方式，在性能要求高的场合则可采用Binary/RPC方式。针对当前多样化的设备类型(浏览器、普通PC、无线设备等)，框架层要支持可定制的序列化机制，例如，对浏览器，框架支持输出Ajax友好的JSON消息格式，而对无线设备上的Native App，框架支持输出性能高的Binary消息格式。
4. 配置，除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。
5. 限流和容错，框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。
6. 管理接口，框架集成管理接口，一方面可以在线查看框架和服务内部状态，同时还可以动态调整内部状态，对调试、监控和管理能提供快速反馈。Spring Boot微框架的Actuator模块就是一个强大的管理接口。
7. 统一错误处理，对于框架层和服务的内部异常，如果框架层能够统一处理并记录日志，对服务监控和快速问题定位有很大帮助。
8. 安全，安全和访问控制逻辑可以在框架层统一进行封装，可做成插件形式，具体业务服务根据需要加载相关安全插件。
9. 文档自动生成，文档的书写和同步一直是一个痛点，框架层如果能支持文档的自动生成和同步，会给使用API的开发和测试人员带来极大便利。Swagger是一种流行Restful API的文档方案。

当前业界比较成熟的微服务框架有Netflix的Karyon/Ribbon，Spring的Spring Boot/Cloud，阿里的Dubbo等。

## 运行期配置管理

服务一般有很多依赖配置，例如访问数据库有连接字符串配置，连接池大小和连接超时配置，这些配置在不同环境(开发/测试/生产)一般不同，比如生产环境需要配连接池，而开发测试环境可能不配，另外有些参数配置在运行期可能还要动态调整，例如，运行时根据流量状况动态调整限流和熔断阈值。目前比较常见的做法是搭建一个运行时配置中心支持微服务的动态配置，简化架构如下图(Fig 10):

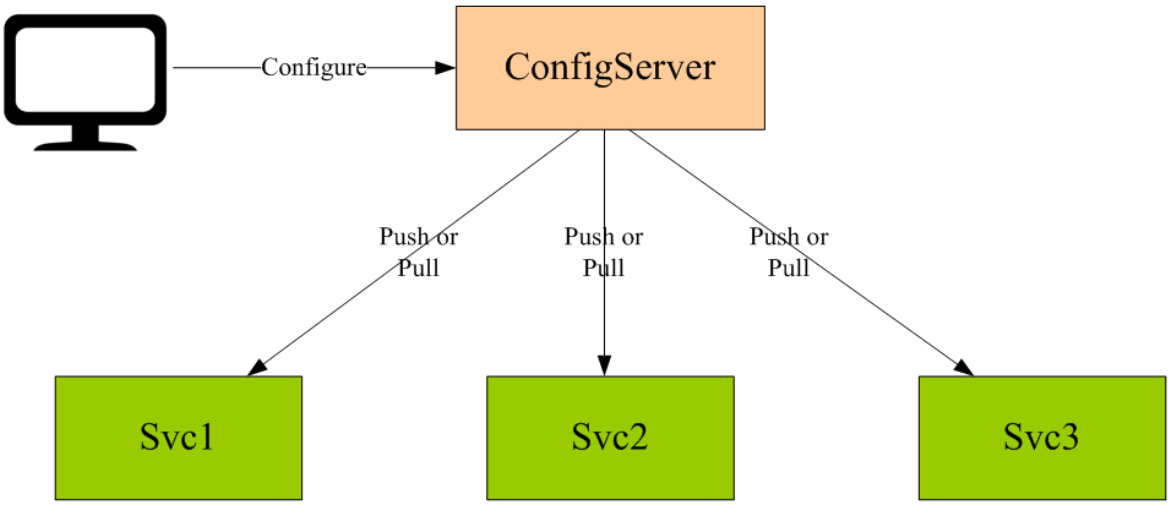


Fig 10, 服务配置中心

动态配置存放在集中的配置服务器上，用户通过管理界面配置和调整服务配置，具体服务通过定期拉(Scheduled Pull)的方式或者服务器推(Server-side Push)的方式更新动态配置，拉方式比较可靠，但会有延迟同时有无效网络开销(假设配置不常更新)，服务器推方式能及时更新配置，但是实现较复杂，一般在服务和配置服务器之间要建立长连接。配置中心还要解决配置的版本控制和审计问题，对于大规模服务化环境，配置中心还要考虑分布式和高可用问题。

配置中心比较成熟的开源方案有百度的Disconf，360的QConf，Spring的Cloud Config和阿里的Diamond等。

## Netflix的微服务框架

Netflix是一家成功实践微服务架构的互联网公司，几年前，Netflix就把它的几乎整个微服务框架栈开源贡献给了社区，这些框架和组件包括：

1. Eureka:　服务注册发现框架
2. Zuul:　服务网关
3. Karyon:　服务端框架
4. Ribbon:　客户端框架
5. Hystrix: 服务容错组件
6. Archaius: 服务配置组件
7. Servo: Metrics组件
8. Blitz4j: 日志组件

下图Fig 11展示了基于这些组件构建的一个微服务框架体系，来自recipes-rss。

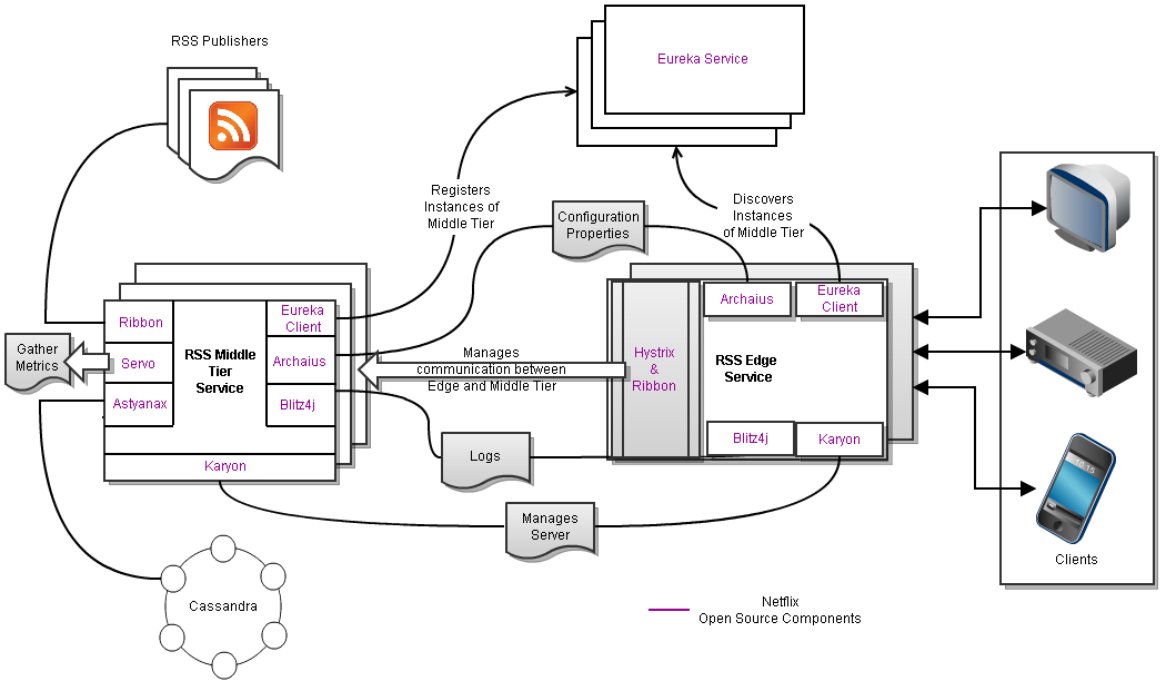


Fig 11, 基于Netflix开源组件的微服务框架

Netflix的开源框架组件已经在Netflix的大规模分布式微服务环境中经过多年的生产实战验证，正逐步被社区接受为构造微服务框架的标准组件。Pivotal去年推出的Spring Cloud开源产品，主要是基于对Netflix开源组件的

进一步封装，方便Spring开发人员构建微服务基础框架。对于一些打算构建微服务框架体系的公司来说，充分利用或参考借鉴Netflix的开源微服务组件(或Spring Cloud)，在此基础上进行必要的企业定制，无疑是通向微服务架构的捷径。

感谢郭蕾对本文的策划和审校。