
目录

Introduction	1.1
微服务定义	1.2
Martin Fowler的《微服务》	1.2.1
微服务设计	1.3
微服务设计模式	1.3.1
微服务实现	1.4
核心技术	1.4.1
进程间通讯	1.4.1.1
网络类库	1.4.1.1.1
REST	1.4.1.1.2
RPC	1.4.1.1.3
消息队列	1.4.1.1.4
服务注册/服务发现	1.4.1.2
Airbnb SmartStack	1.4.1.2.1
Netflix Eureka	1.4.1.2.2
负载均衡	1.4.1.3
Netflix Ribbon	1.4.1.3.1
配置管理	1.4.1.4
Netflix Archaius	1.4.1.4.1
Disconf	1.4.1.4.2
熔断器	1.4.1.5
Netflix Hystrix	1.4.1.5.1
网关	1.4.1.6
Netflix Zuul	1.4.1.6.1
基础设施	1.4.2
Cisco Mantl	1.4.2.1
Vamp	1.4.2.2
微服务框架	1.4.3
阿里 Dubbo	1.4.3.1
微博 Motan	1.4.3.2

Netflix OSS	1.4.3.3
Spring Boot	1.4.3.4
Spring Cloud	1.4.3.5
Spring Cloud NetFlix	1.4.3.6
全文标签总览	1.5

微服务学习笔记

微服务架构风格是一种通过一套小型服务来开发单个应用的方法，每个服务运行在自己的进程中，并通过轻量级的机制进行通讯。

2015年之后微服务被开发社区接受并快速普及，尤其在 `docker` 等容器技术风行之后更是被认为是"天作之合"。

微服务定义

Martin Fowler在"microservices"一文中给出的定义:

the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms

微服务架构风格是一种通过一套小型服务来开发单个应用的方法，每个服务运行在自己的进程中，并通过轻量级的机制进行通讯

学习笔记: Martin Fowler的《微服务》

前言

Martin Fowler的《微服务》是第一篇详细介绍微服务的文章。对微服务进行了定义，并与传统架构进行了对比，阐述了微服务的优势。

- 原文: [microservices](#)
- 中文翻译: [微服务](#)
- 演说视频@GOTO Berlin 2014

注1: 上面的中文翻译是目前找到的最好的版本, 语句通顺而准确, 向作者致敬!

注2: 找到的第一个版本的翻译是[微服务中文翻译版本](#), 翻译质量很不理想, 非常拗口而且语义也和原文有差异. 看不下去, 我曾决定自己再润色一遍. 但翻译到中途时发现了上面的好版本就放弃了.

注3: 这个所谓好的翻译, 也只是前半段水准不错, 后半段就急剧下滑, 看不下去.....

术语表

术语(English)	翻译(中文)	备注
micro service	微服务	
monolithic	单块/单体	和微服务相对的传统架构模式
Conway's Law	康威定律	内容一句话: "组织决定架构"

读书笔记

注: 英文原文和翻译版本请见上面的链接. 以下部分为个人学习笔记.

微服务定义:

微服务架构风格是一种将一个单一应用程序开发为一组小型服务的方法, 每个服务运行在自己的进程中, 服务间通信采用轻量级通信机制(通常用HTTP资源API)。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式的管理, 服务可用不同的语言开发, 使用不同的数据存储技术。

和单体服务器的对比:

单体服务器是构建这样一个系统最自然的方式。处理请求的所有逻辑都运行在一个单一进程中，允许你使用编程语言的基本特性将应用程序划分类、函数和命名空间。你认真的在开发机上运行测试应用程序，并使用部署管道来保证变更已被正确地测试并部署到生产环境中。该单体的水平扩展可以通过在负载均衡器后面运行多个实例来实现。

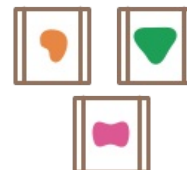
单体服务器的问题:

> 变更周期被捆绑在一起 —— 即使只变更应用程序的一部分，也需要重新构建并部署整个单体。长此以往，通常将很难保持一个良好的模块架构，这使得很难变更只发生在需要变更的模块内。程序扩展要求进行整个应用程序的扩展而不是需要更多资源的应用程序部分的扩展。

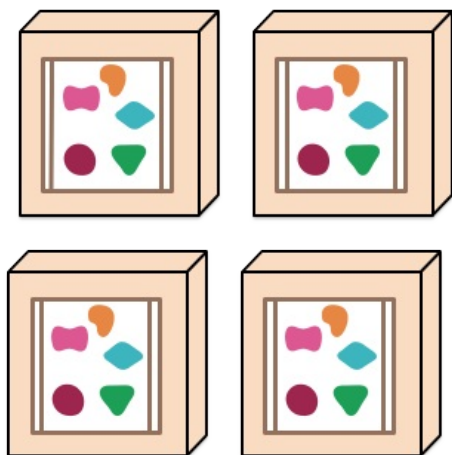
一个单体应用程序把它所有的功能放在一个单一进程中...



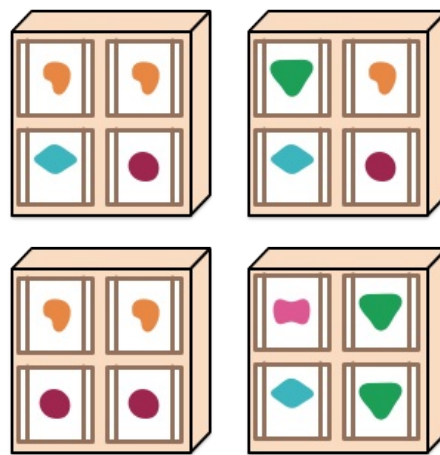
一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过在多个服务器上复制这个单体进行扩展



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制。



微服务架构风格：

构建应用程序为服务套件。除了服务是可独立部署、可独立扩展的之外，每个服务都提供一个固定的模块边界。甚至允许不同的服务用不同的语言开发，由不同的团队管理。

微服务架构的特征 / Characteristics of a Microservice Architecture

通过服务组件化 / Componentization via Services

组件的定义:

组件是一个可独立替换和独立升级的软件单元。

微服务架构组件化软件的主要方式:

分解成服务, 而服务是一种进程外的组件, 通过web服务请求或rpc机制通信

使用服务作为组件而不是使用库的主要原因:

1. 服务是可独立部署的
2. 更加明确的组件接口: 服务通过明确的远程调用机制可以避免组件间的紧耦合 **

缺点:

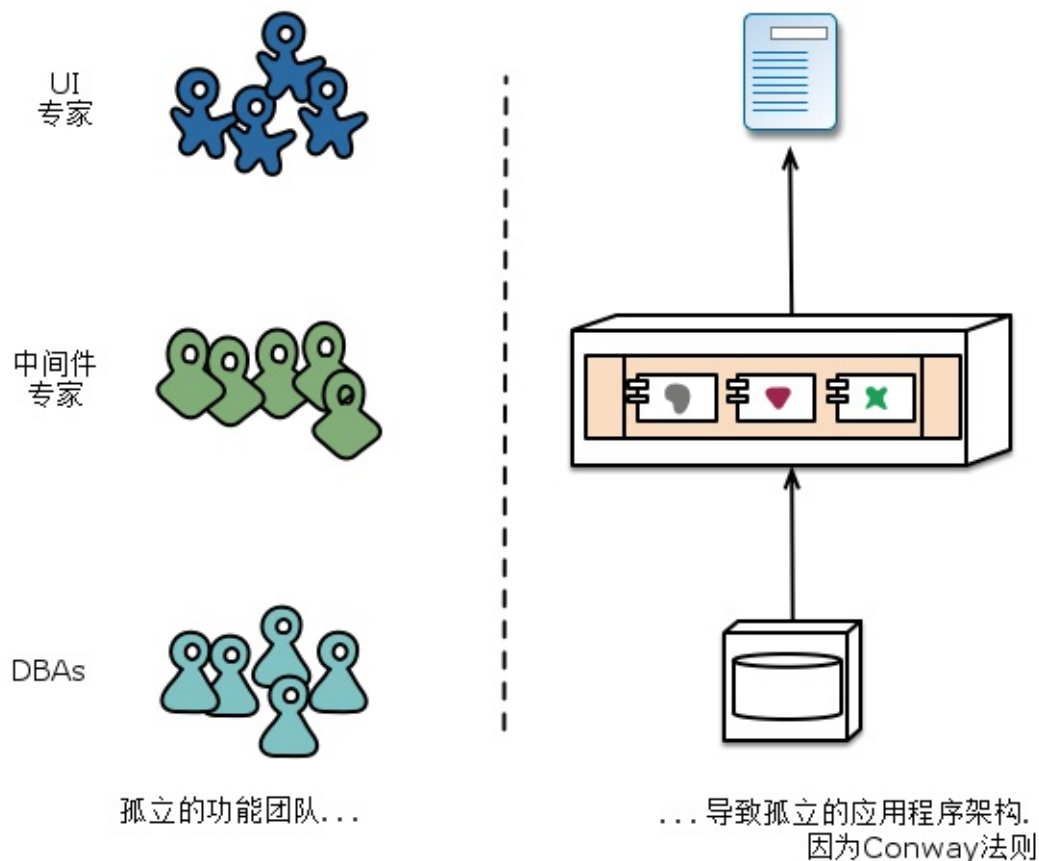
1. 远程调用比进程内调用更昂贵
2. 导致远程API被设计成粗粒度, 不便于使用

围绕业务能力组织 / Organized around Business Capabilities

康威定律(Conway's Law):

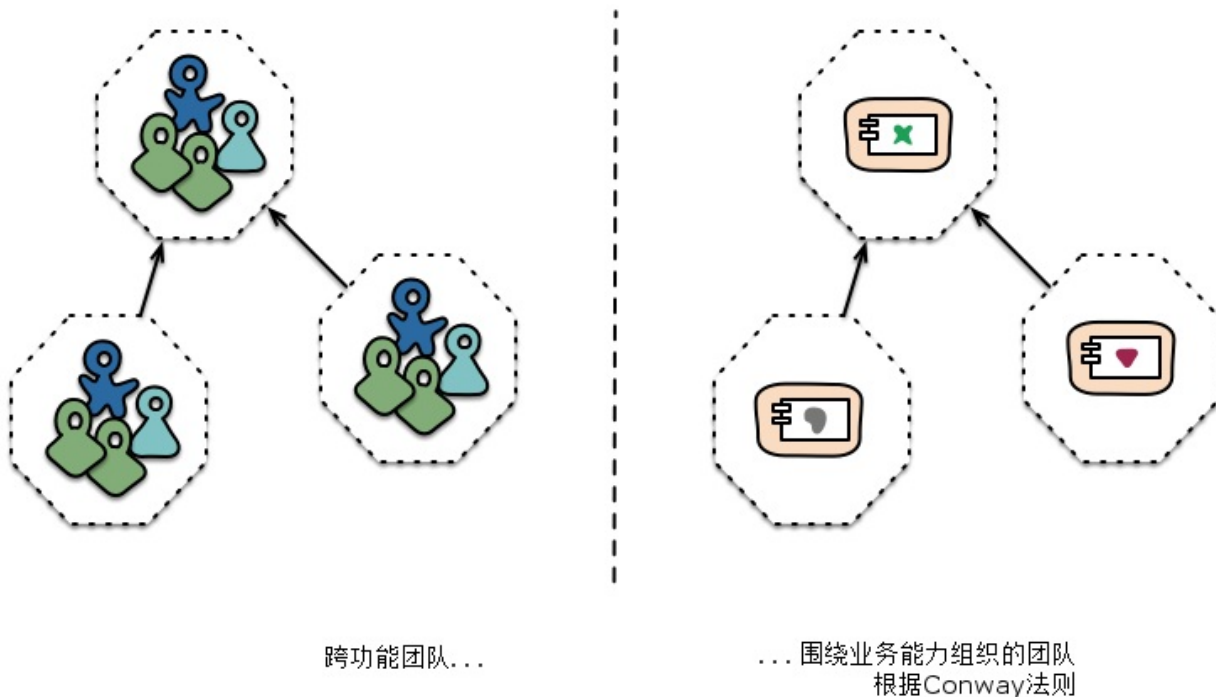
任何设计系统(广泛定义的)的组织将产生一种设计, 他的结构就是该住址的通信结构。 --
Melvyn Conway 1967

简单一句话就是: "组织决定架构"!



微服务的组织方式:

微服务采用不同的分割方法，划分成围绕业务能力组织的**服务**。这些服务采取该业务领域软件的宽栈实现，包括用户接口、持久化存储和任何外部协作。因此，团队都是跨职能的，包括开发需要的全方位技能：用户体验、数据库、项目管理。



建议:

敦促创建单体应用程序的大型团队将团队本身按业务线拆分

是产品不是项目 / Products not Projects

项目模式:

目标是交付将要完成的一些软件。完成后的软件被交接给维护组织，然后它的构建团队就解散了。

微服务支持者倾向于避免这种模式，而是认为一个团队应该负责产品的整个生命周期。

产品模式:

产品思想与业务能力紧紧联系在一起。要持续关注软件如何帮助用户提升业务能力，而不是把软件看成是将要完成的一组功能。

微服务相对单体应用的优势:

同样的方法也可以用在单体应用程序上，但服务的粒度更小，使得它更容易在服务开发者和用户之间建立个人关系。

智能端点和哑管道 / Smart endpoints and dumb pipes

在不同进程间创建通信结构时, 智能放在哪里, 有两种不同思路:

1. 把显著的智慧强压进通信机制本身

一个很好的例子就是企业服务总线(ESB), 在ESB产品中通常为消息路由、编排(choreography)、转化和应用业务规则引入先进的设施。

2. 智能端点和哑管道

微服务社区主张另一种方法, 基于微服务构建的应用程序的目标是尽可能的解耦和尽可能的内聚 - 他们拥有自己的领域逻辑, 他们的行为更像经典UNIX理念中的过滤器 - 接收请求, 应用适当的逻辑并产生响应。

使用简单的REST风格的协议来编排, 而不是使用像WS-Choreography或者BPEL或者通过中心工具编制(orchestration)等复杂的协议。

最常用的两种协议:

1. 使用rest API的HTTP请求-响应和轻量级消息传送
2. 在轻量级消息总线上传递消息

把单体变成微服务:

最大的问题在于通信模式的改变。一种幼稚的转换是从内存方法调用转变成RPC, 这导致频繁通信且性能不好。相反, 你需要用粗粒度通信代替细粒度通信。

去中心化治理 / Decentralized Governance

集中治理的一个后果是技术平台的单一标准化发展趋势。不是每个问题都是钉子, 不是每个问题都是锤子。我们更喜欢使用正确的工具来完成工作。

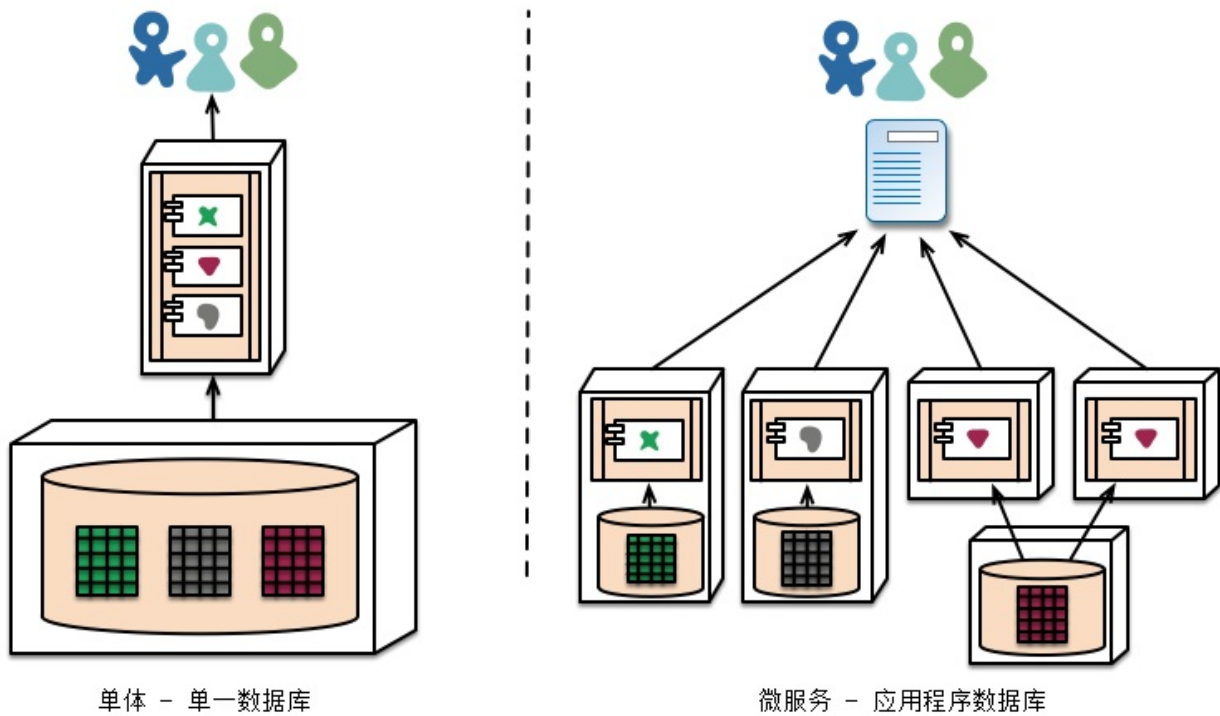
把单体的组件分裂成服务, 在构建这些服务时可以有自己的选择。

去中心化治理的最高境界就是亚马逊广为流传的build it/run it理念: 团队要对他们构建的软件的所有方面负责, 包括7*24小时的运营。

去中心化数据管理 / Decentralized Data Management

DDD把一个复杂域划分成多个有界的上下文, 并且映射出它们之间的关系。这个过程对单体架构和微服务架构都是有用的, 但在服务和上下文边界间有天然的相关性, 边界有助于澄清和加强分离, 就像业务能力部分描述的那样。

微服务更倾向于让每个服务管理自己的数据库, 或者同一数据库技术的不同实例, 或完全不同的数据库系统。



分布式事务:

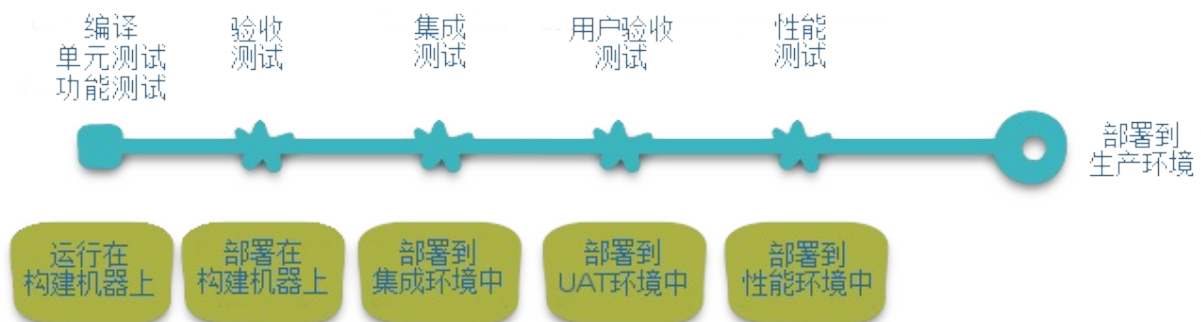
微服务架构强调服务间的无事务协作，关注最终一致性和事后补偿。

权衡

修复错误的代价是否小于一致性下业务损失的代价？

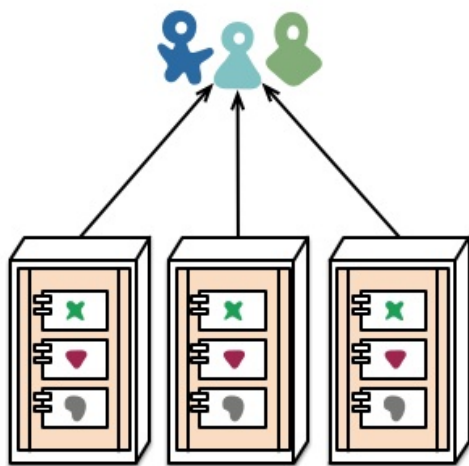
基础设施自动化 / Infrastructure Automation

基于持续部署(和它的前身持续集成), 构建管线图:

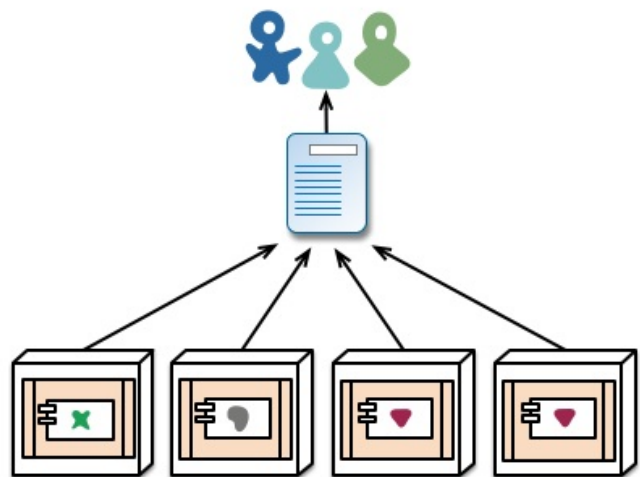


关键特性:

1. 自动化测试
2. 自动化部署
3. 在生产环境中管理微服务



单体 - 多个模块在同一个进程中



微服务 - 每个模块运行在不同的进程中

为失效设计 / Design for Failure

使用服务作为组件的一个后果:

应用程序需要被设计成能够容忍服务失效。任何服务调用都可能因为服务提供者不可用而失败，客户端必须尽可能优雅的应对这种失败。

与单体应用设计相比这是一个劣势，因为它引入额外的复杂度。

为因对随时可能失败的服务, 微服务:

1. 快速检测故障
2. 自动恢复服务
3. 应用程序的实时监测: 包括性能指标(如TPS)和业务指标(如订单数)
4. 告警系统: 通知开发团队跟进和调查

微服务希望看到为每个单独的服务设置的完善的监控和日志记录:

1. 控制面板上显示启动/关闭状态
2. 各种各样的运营和业务相关指标
3. 断路器状态
4. 当前吞吐量和时延

进化式设计 / Evolutionary Design

分割应用的原则: 组件可独立的更换和升级

微服务强调可替代性, 通过变更模式来驱动模块化: 同时变化的东西保持在同一模块中。系统中很少变更的部分应该和正在经历大量扰动的部分放在不同的服务里。如果你发现你自己不断地一起改变两个服务, 这是它们应该被合并的一个标志。

微服务是未来吗？ / Are Microservices the Future?

Martin-Fowler 在2014年出写下这个文章时, 还是"怀着谨慎乐观的态度". 而这之后的一年半的时间, 业界对微服务的接受程度远远超过当时的预期, 几乎可是说是推崇甚至有成为业界标杆的趋势.

个人感言

去年这篇文章出来不久, 就认真拜读过一遍(当时还只有英文版), 收益颇多.

一年半之后, 细细重温这篇老文, 还是收益颇多, 有些当时还不是太懂的地方, 现在理解的深刻多了, 比如说"康威定律".

计划一两年之后再回来重读一遍, 很期待届时会有什么想法和感触.

微服务学习资料

待学习的资料

- [微服务中的耦合与自治](#)
- [微服务最佳实践](#)
- [Martin Fowler谈微服务的优缺点](#)

学习笔记：微服务设计模式

前言

发表在Java Code Geeks的一篇文章，介绍了六种微服务架构的设计模式。

- 原文: [Microservice Design Patterns](#)
- 中文翻译: [微服务设计模式@infoQ](#)

注: infoq的这个翻译做的非常的离谱, 对原文内容做了大量的改写和删除, 简直乱来. 很多重要信息都莫名其妙的在翻译的时候砍掉了, 本来这个文档篇幅也不多, 不知所谓. 建议阅读时多看看英文原文.

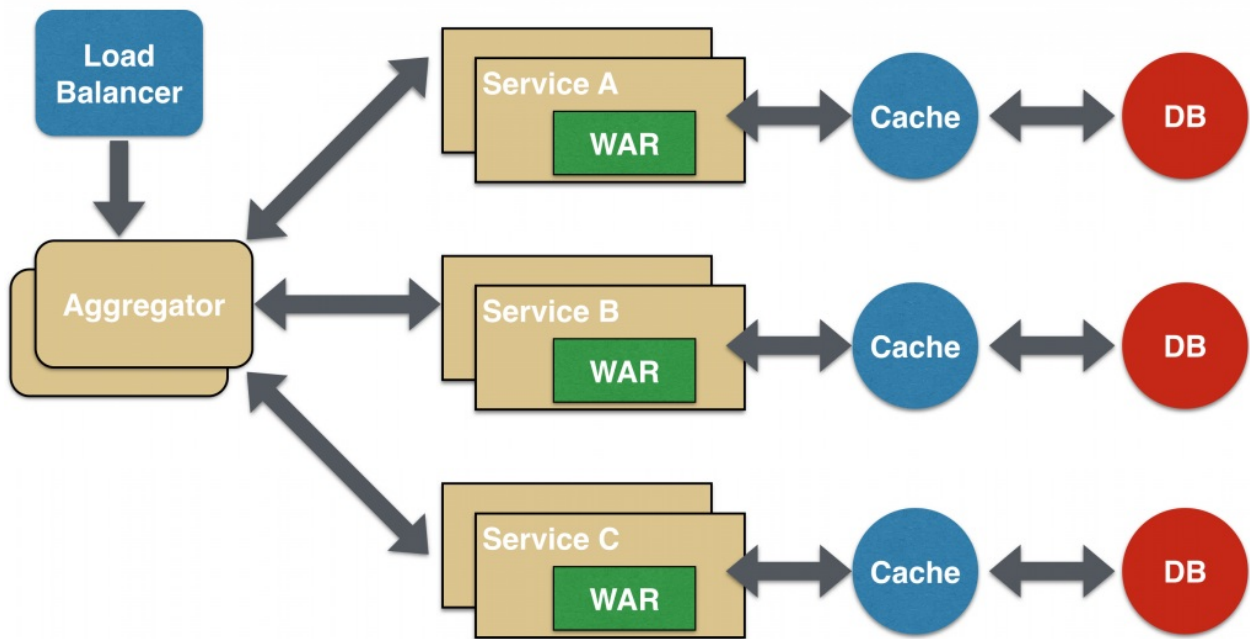
术语表

术语(English)	翻译(中文)
monolithic	单块/单体
Aggregator	聚合器
dumb proxy	哑代理

学习笔记

聚合器微服务设计模式 / Aggregator Microservice Design Pattern

最常用也最简单的设计模式:



聚合器调用多个服务实现应用程序所需的功能, 有多种形式

1. 简单的显式逻辑和业务逻辑

例如一个简单的Web页面. 这种情况下Aggregator扮演的是一个简单的微服务消费者, 通过简单的调用一个或者多个微服务来实现所需的功能. 也可以有一些业务逻辑, 完成某个业务请求.

2. 更高层次的组合微服务

Aggregator可以在组合多个微服务的基础上, 实现一定的业务逻辑和功能, 然后对外暴露为一个微服务供使用者调用.

在某些地方将这种方式称之为"API Gateway".

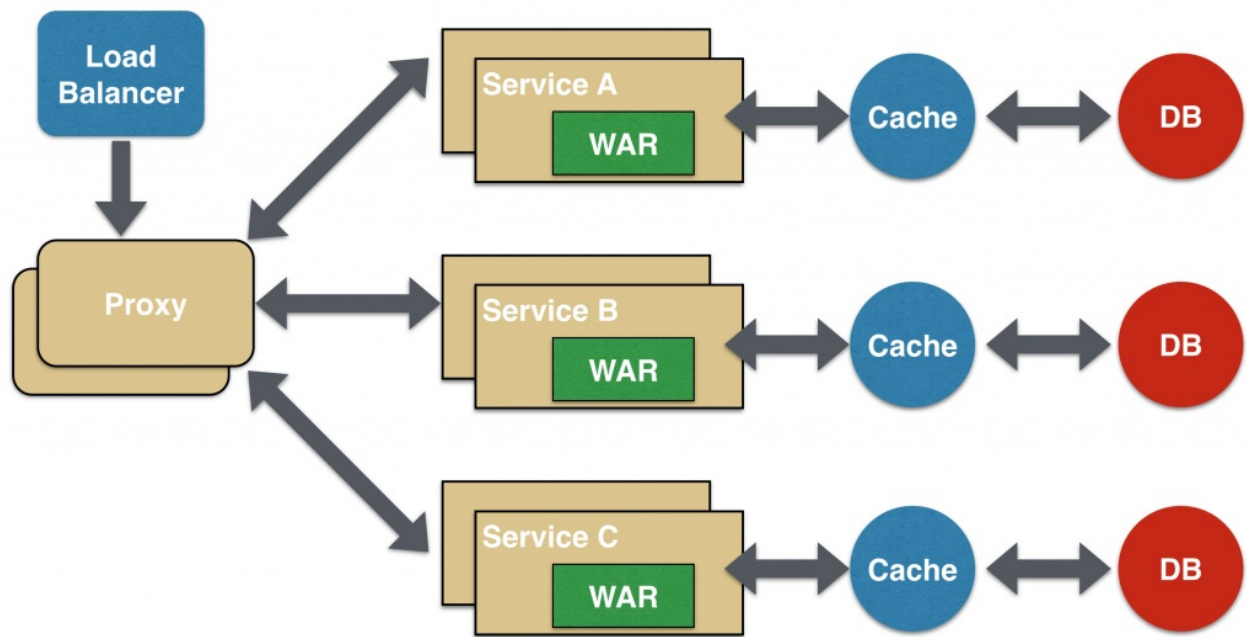
代理微服务设计模式 / Proxy Microservice Design Pattern

这是聚合器模式的一个变种:

- 客户端并不聚合数据, 但会根据业务需求的差别调用不同的微服务

即前面的聚合模式是先后调用a/b/c三个服务, 而代理模式下只会根据某个条件if一下然后选择a/b/c中的某一个做调用.

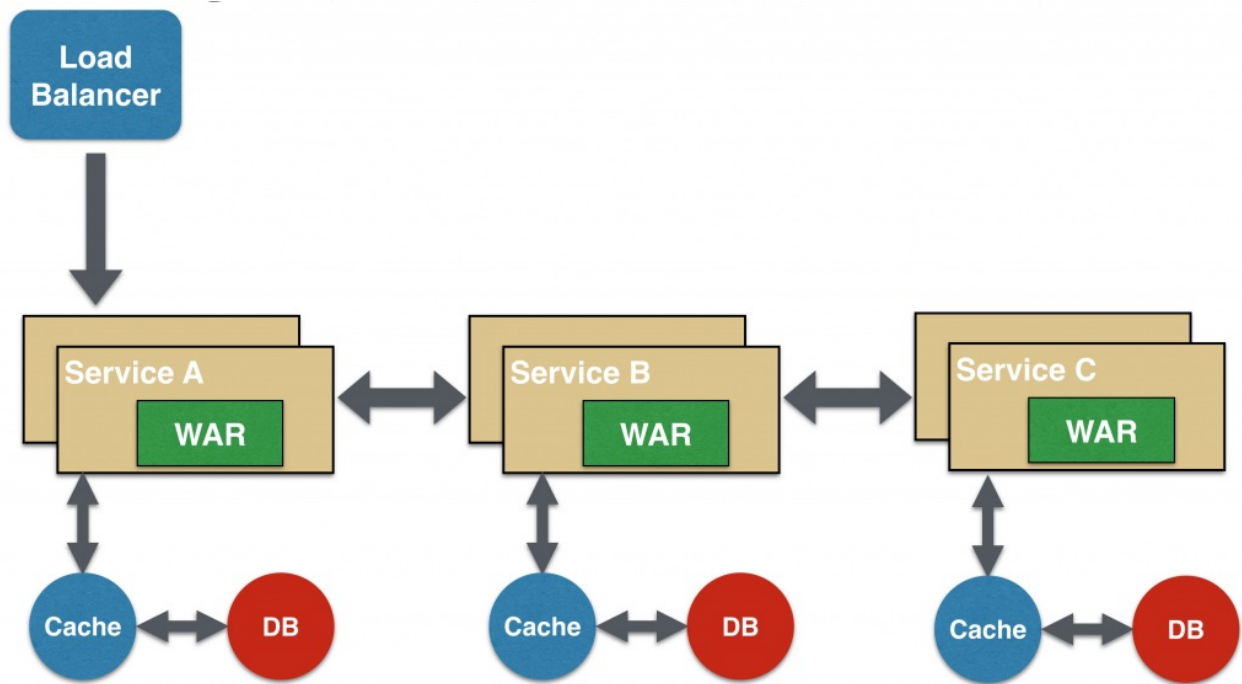
- 代理可以仅仅委派请求, 也可以进行数据转换工作



代码模式还有一个变种,即proxy将请求代理到固定的某一个服务,称为"dumb proxy".它很可能是一个"smart proxy",将某个服务的返回转换为另外一种格式再返回给客户.

链式微服务设计模式 / Chained Microservice Design Pattern

链式模式在接收到请求后会先后调用多个服务,产生一个经过合并的响应给客户.



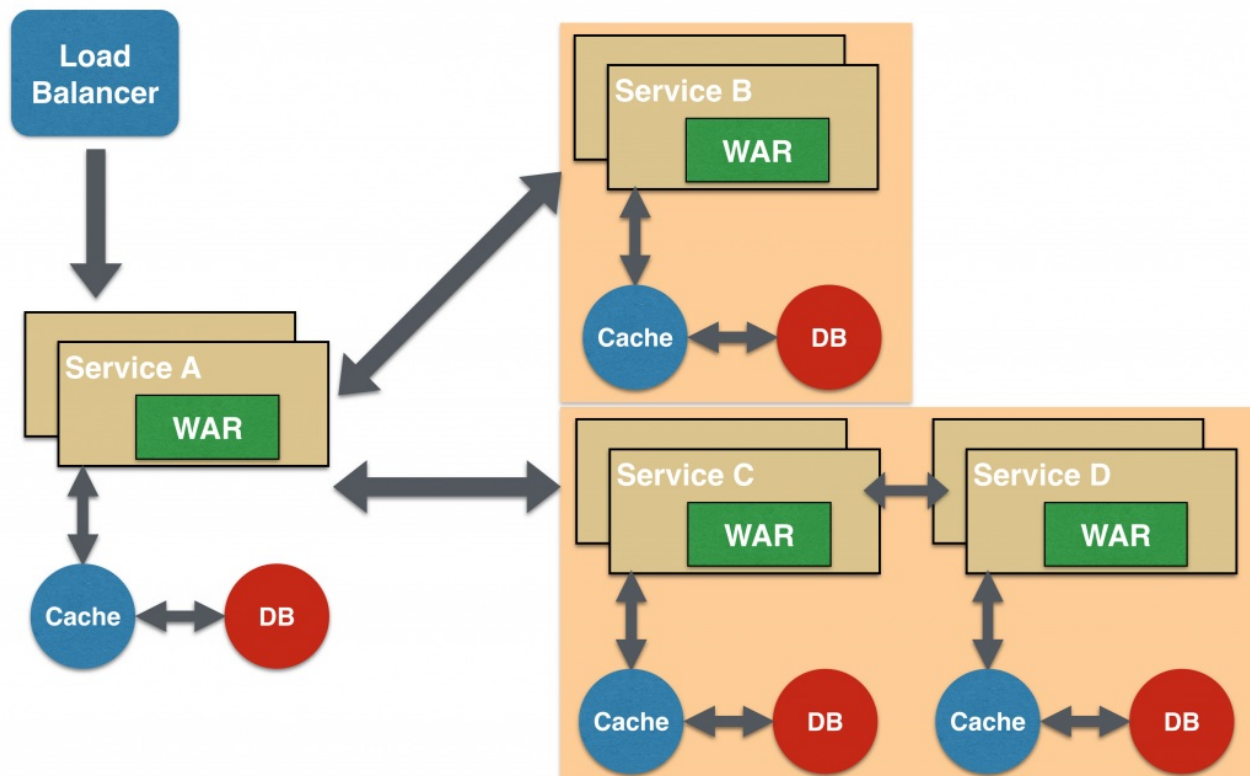
所有服务都使用同步消息传递:

- 在整个链式调用完成之前,客户端会一直阻塞

- 服务调用链不宜过长，以免客户端长时间等待

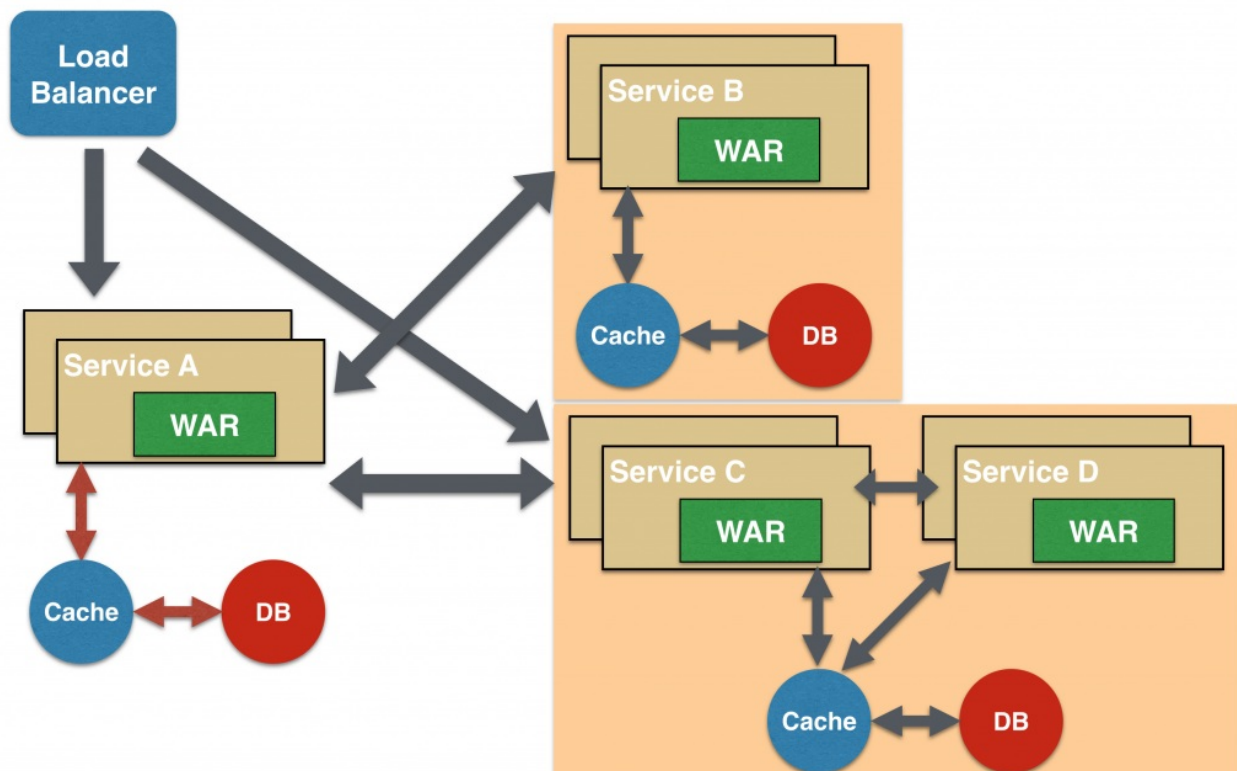
分支微服务设计模式 / Branch Microservice Design Pattern

分支模式是聚合器模式的扩展, 结合了链式模式.



数据共享微服务设计模式 / Shared Data Microservice Design Pattern

重构现有的"单体应用"时，SQL数据库反规范化可能会导致数据重复和不一致。因此，在单体应用到微服务架构的过渡阶段，可以使用这种设计模式, 让两个微服务共享数据和缓存:

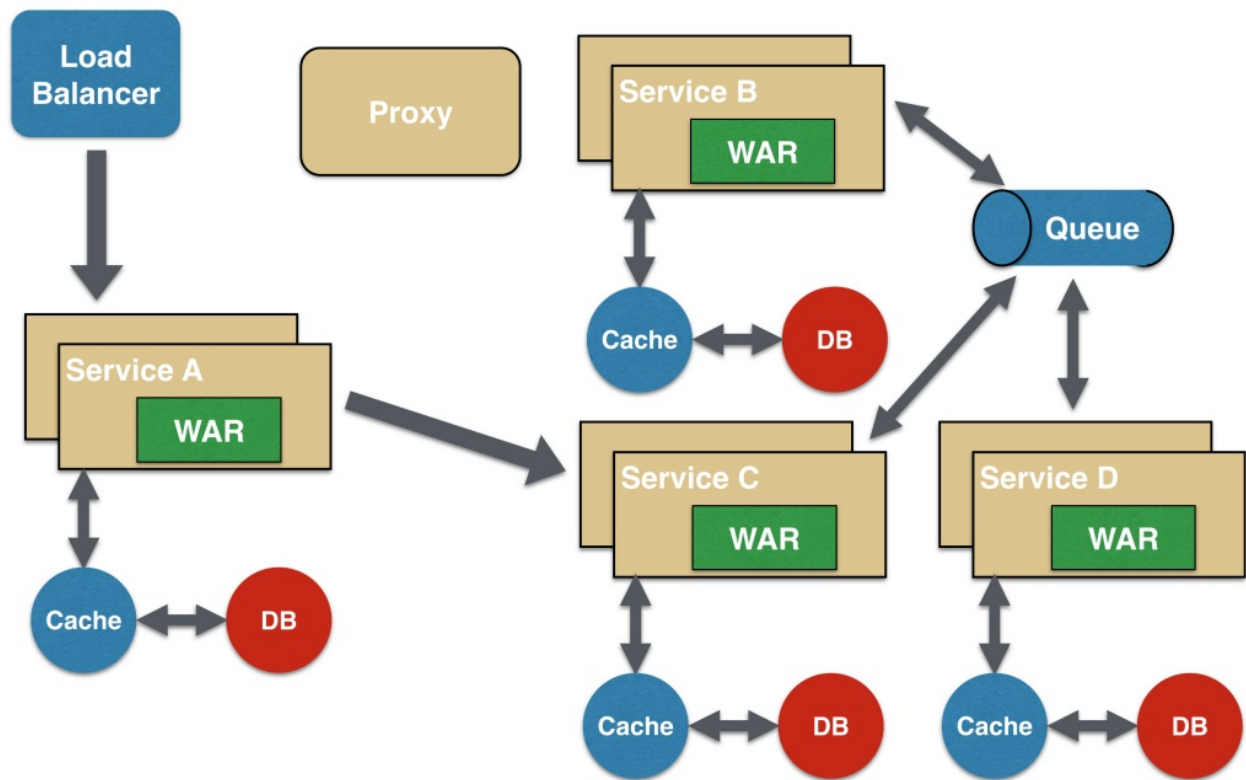


不到万不得已不要这样使用!

- 只有在两个服务之间存在强耦合关系时才可以
- 对于基于微服务的新建应用程序而言，这是一种反模式
- 这也可以是暂时的过度阶段，最终还是要消除共享实现完全自治

异步消息传递微服务设计模式 / Asynchronous Messaging Microservice Design Pattern

同步请求会造成阻塞, 可以选择使用消息队列代替同步请求/响应:



总结

常见的微服务设计模式并不复杂, 按照实际情况权衡选择.

技术选型

这里收集和整理在实现微服务框架和相关的配套基础设施时，可以选择的技术方案和产品。

主要是以开源产品和免费产品为主，商业方案只会简单一笔带过。

参考资料

- [微服务技术体系](#)

核心技术

通讯机制

前言

在微服务架构中，为了彻底隔绝不同服务，采用了最坚决的方案：强制要求不同服务之间通过远程访问方式进行通讯。

在这点上，微服务和以OSGi为代表的Java模块化方案形成鲜明对比。

访问方式

在微服务间做远程访问，目前主要的方式是有以下三种类型：

1. REST
2. RPC
3. 定制

后面我们详细介绍前两种方式的做法和可选的常见实现方案。

第三中定制方式，简单说就是自己按照需要选择不同的类库：

1. 网络通讯，可以选择 netty/mina等
2. 协议可以选择 HTTP/HTTP2/TCP/UDP 等
3. 编解码方案 可以选择 Rest/protocol buffer/thrift 等

然后以上三种情况可以有多种排列组合，最后的结果可以是五花八门，就不一一列举了。

网络通讯类库

在涉及网络通讯时，必然会有选择网络通讯类库的问题，目前主要是各种 NIO 类库。

在介绍具体的访问方式之前，先过一下 NIO 类库，后面的各种访问方式都有可能使用到这些 NIO 类库。

交互方式

进程间通讯的交互方式，可以分为两个维度：

1. 第一个维度是交互对象的数量

- 一对一：每个客户端请求确切地被一个服务实例处理
- 一对多：每个客户端请求被多个服务实例处理

2. 第二个维度是交互时应答的返回方式

- 同步：客户端期待服务在一定时间内应答，当等待时客户端甚至会阻塞
- 异步：客户端在等待应答时不阻塞，而应答不需要立即发送

两个维度组合之后得到多种可能的方式：

	One-to-One	One-to-Many
Synchronous	Request/response	
Asynchronous	Notification	Publish/subscribe
Asynchronous	Request/async response	Publish/async responses

一对一的交互方式：

- **Request/response**：标准的请求/应答方式。客户端发送请求到服务并等待应答。客户端期待应答在适当的时间内到达。在基于线程的程序中，发起请求的线程在等待时可能阻塞。
- **Notification(或者单路请求)**：客户端发送请求到服务但是不期待有答复。
- **Request/async response**：客户端发送请求到服务，服务器端异步给回复。客户端在等待时不阻塞，而在设计上通常是假定可能不会很快就有应答。

一对多的交互方式：

- **Publish/subscribe**：客户端发布通知消息，然后被0个或者多个关注的服务消费
- **Publish/async responses**：客户端发布请求消息，然后等待一定数量的从关注的服务返回的应答

参考文档

- [Building Microservices: Inter-Process Communication in a Microservices Architecture](#)

网络类库

NIO 框架

目前Java的 NIO 类库的选择很多，但是社区使用比较多也比较推荐的是 Netty 。

如日中天的 Netty



- <http://netty.io/>

Netty是由JBOSS提供的一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

是目前最流行的 NIO 方案，最新版本 4.1, 主要提供了对 HTTP/2 的支持。

特别提醒：**Netty 5.*** 已经被废弃！

- 具体的说明请见：<https://github.com/netty/netty/issues/4466>
- 主要理由是: 使用 ForkJoinPool 的主要改动增加了复杂度但是并没有显示出明确的性能收益。(原文：The major change of using a ForkJoinPool increases complexity and has not demonstrated a clear performance benefit.)
- 因此目前 Netty 官方已经放弃了 netty 5.x 版本的开发和更新，并且推荐已经使用 netty 5.x alpha版本的同学退回到 4.0或者4.1版本

英雄迟暮的 Mina



- <http://mina.apache.org/>

Apache Mina是一个能够帮助用户开发高性能和高伸缩性网络应用程序的框架。它通过Java nio技术基于TCP/IP和UDP/IP协议提供了抽象的、事件驱动的、异步的API。

传闻说 mina 已经很少更新，但是看最近(2016年9月)又连续发布了 2.0.14/2.0.15 两个版本。

个人建议：如果没有特殊原因，在 netty 和 mina 之间推荐选择 netty。

注: Netty和Mina是Java社区知名的通讯框架，出自同一个作者: 韩国人 Trustin Lee。

Mina略早，属于Apache基金会，而Netty开始在Jboss名下，后来出来自立门户 netty.io。

鲜为人知的 XNIO

- <https://github.com/xnio/xnio>

XNIO 是一个鲜为人知的 Java NIO 类库，由 Jboss/Redhat 提供，主要是用在 Jboss/Redhat 自家的Web 服务器 Undertow 上。

XNIO 最新版本 3.4.0.Final，发布于2016年8月。



Undertow 是一个采用 Java 开发的灵活的高性能 Web 服务器，提供包括阻塞和基于 NIO 的非堵塞机制。Undertow 是 Jboss/Redhat 公司的开源产品，是 Wildfly 默认的 Web 服务器。

Undertow 比较有意思的特性是它"特别轻量"：Undertow core jar 不到1M，运行时简单的内嵌服务器使用不到4M的堆空间。

另外，Undertow 居然也提供了对HTTP2的支持。

HTTP 类库

支持Android的 Okhttp

- <http://square.github.io/okhttp/>

An HTTP & HTTP/2 client for Android and Java applications

亮点：

1. 支持 HTTP/2
2. 支持 Android

REST

介绍

Rest方式是业界最流行的方案.

注：我因为个人原因偏好 RPC 方案，因此对 REST 方式研究不多，这里只做简单介绍。

框架

- [Dropwizard](#)
- [Jersey](#)
- [Play Framework](#)
- [Spring MVC](#)
- [Spark Framework](#): 轻量级的 Java web 框架,受 Ruby 框架 Sinatra 启发,适合快速开发

如果开发的 REST 服务是使用 REST 成熟度 最高的 HATEOAS (Hypermedia as the engine of application state)，可以尝试 spring 为此推出的项目：

- [Spring HATEOAS](#)

个人推荐，比较大众化的选择是：

1. 服务器端用 Spring mvc
2. 客户端用 Jersey

参考资料

- 使用 [Spring HATEOAS](#) 开发 REST 服务

RPC

Google gRPC



- <https://github.com/grpc/grpc>

gRPC 是 google 最新发布的开源 RPC 框架, 声称是"一个高性能, 开源, 将移动和HTTP/2放在首位的通用的RPC框架.", 支持 android 和 iOS. 技术栈非常的新, 基于HTTP/2, netty4.1, proto3, 拥有非常丰富而实用的特性, 堪称新一代 RPC 框架的典范.

这个项目开始于2015年2月, 1.0正式版本正式发布于2016年8月, 已经可以用于生产环境了。

注: 这是我个人最喜欢的一个 RPC 框架, 推崇备至。这里有一份我的 [gRPC的学习笔记](#), 有兴趣的同学可以过去看看

Apache Thrift



- <http://thrift.apache.org/>

Thrift 源于 facebook, 在2007年捐献给 Apache, 目前最新版本 0.9.3 (发布于 2015-10-06)

Thrift 可以算是业界最经典的 RPC 框架之一, 效率极高, 使用广泛, 成熟稳定。

缺点:

1. 项目似乎不再继续演进, 看不到未来的路线图
2. 和 gRPC 相比, 缺乏对HTTP/2的支持, 缺乏对移动设备的支持
3. 底层通讯机制不够理想: 唯品会的OSP框架干脆就直接放弃 thrift 底层通讯直接用 netty4 重写

Hessian/Hessian2



- <http://hessian.caucho.com/>

基于HTTP协议传输的二进制 web service 方案，由 caucho 公司提供（他们家还出产 Resin，一个曾经很不错的 web container）。

缺点：

1. 效率和 thrift / gRPC 等相比较差，甚至不如REST风格的Jackson
2. 基本停止发展，最后一个大版本 4.0.* 发布于2009年，之后只有bugfix版本
3. 以今天的眼光看 hession 的特性，只能说陈旧

不过由于历史原因，还是有不少框架在使用 hessian，比如 dubbo 默认采用 hessian2 序列化，微博新开源的 Motan 框架也是用的 hession 4.0.38。

Dubbo 协议



阿里在 Dubbo 中实现了基于 TCP 的 RPC 协议，名字也就简单的叫做 Dubbo 协议(当然 Dubbo 框架也支持其他协议如hession2/thrift/RMI/WebService)。当时的目标是希望开发一个高效的java序列化实现，但是为开发成熟，一度不建议在生产环境使用它。后来 dubbo 自生停止发展了，这个 Dubbo 协议也自然废弃。

建议：没有特殊情况，不要使用。

总结

个人意见，选择 RPC 框架时：

1. 追新的同学请选择 gRPC
2. 求稳的同学请选择 thrift
3. 怀旧的同学请选择 hessian2

强调：如果想要高效率，支持 HTTP/2,支持 android 和 iOS，实现高效率的服务器端主动推送，gRPC 是目前唯一的选择。

消息队列

主流实现

RabbitMQ



- <https://www.rabbitmq.com>

Apache Kafka



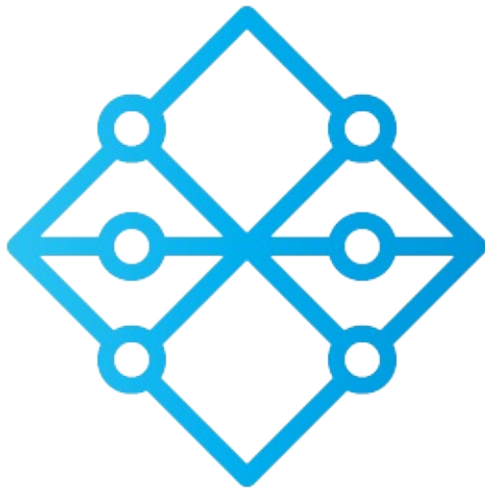
- <http://kafka.apache.org>

Apache ActiveMQ



- <http://activemq.apache.org>

NSQ



- <http://nsq.io/>

NSQ最初为提供短链接服务的应用 Bitly 而开发。

建议

对于消息系统，个人的建议：

- 轻量级 选择 RabbitMQ
- 重量级 选择 Kafka
- 如果没有历史原因，不要再选择 ActiveMQ
- 要求非常高，考虑一下 NSQ

参考资料

- RabbitMQ
 - [Event-driven Microservices Using RabbitMQ](#)
 - [使用RabbitMQ的事件驱动微服务: 上文的中文翻译](#)
- NSQ
 - [NSQ：分布式的实时消息平台](#)
 - [Scaling NSQ to 750 Billion Messages](#)
 - [\[译\]我们是如何使用NSQ处理7500亿消息的: 上文的中文翻译](#)

服务注册/发现

强一致性存储

- [Zookeeper](#)

来自 Apache，以Java语言编写，强一致性(CP), 使用 Zab 协议 (基于PAXOS)。

郑重提醒：使用zookeeper时，不要直接使用zk的Java api，请尽量使用 **Curator** ！

- [Doozer](#)

doozerd 的 Go 语言客户端，强一致性，使用 PAXOS 协议。

这是一个很多年前就存在的项目，已经停滞很久，有大量的fork。

个人建议：没有特殊理由，不要选择。

- [Etcd](#)

据说是受 zookeeper 和 Doozer 启发，用 Go 语言编写，使用 Raft 协议。

etcd 2.* 版本提供 HTTP+JSON 的 API，而最新的 etcd 3.0 版本修改为 gRPC，效率大为提升。

- [Consul](#)

来自 hashicorp 公司，和 etcd 一样也是基于 Raft 协议。

Consul 最大的优势，是提供可以直接使用的成品，如服务注册，健康检查，配置等，相比之下 zk，etcd 等更像是提供原材料。

- [SmartStack](#)

来自 Airbnb，由 [Nerve](#) 和 [Synapse](#) 两个部分组成，依赖zookeeper和haproxy。用 Ruby 语言编写

- [Eureka](#)

来自 Netflix，服务器端和客户端都是用 Java 语言编写，因此只能用于Java和基于JVM的语言。

- [NSQ](#)

来自 Bitly，Go 语言编写。

- [Serf](#)

Go 语言编写,采用的是基于 gossip 的 SWIM 协议。

主要的一致性协议有：

- PAXOS：Zookeeper
- Raft：Consul / Etcd

弱一致性存储

也有不是很介意分布式一致性，而采用自行实现的简单方案：

- 新浪微博 Vintage

基于redis。

用于新浪的 Motan 框架："Motan 可以支持不同的注册中心，如 ZK、Consul，目前使用的注册中心是平台开发的 Vintage，Vintage 是一个基于 Redis 的轻量级 KV 存储系统，能够提供命名空间服务、服务注册、服务订阅等功能。"。

注：已经和 Tim Yang 老师确认，目前的确还是如此。

DNS相关

- Spotify 和 DNS
- [SkyDNS](#)
- consul提供的DNS

Name	Type	AP or CP	Language	Dependencies	Integration
Zookeeper	General	CP	Java	JVM	Client Binding
Doozer	General	CP	Go		Client Binding
Etcd	General	Mixed (1)	Go		Client Binding/HTTP
SmartStack	Dedicated	AP	Ruby	haproxy/Zookeeper	Sidekick (nerve/synapse)
Eureka	Dedicated	AP	Java	JVM	Java Client
NSQ (lookupd)	Dedicated	AP	Go		Client Binding
Serf	Dedicated	AP	Go		Local CLI
Spotify (DNS)	Dedicated	AP	N/A	Bind	DNS Library
SkyDNS	Dedicated	Mixed (2)	Go		HTTP/DNS Library

内建服务注册的框架

服务注册是任何一个SOA/服务化/微服务框架的必不可少的一部分，因此很多框架都内建了对服务注册的支持：

- dubbo：支持注册中心扩展，支持多个实现
- motan：提供对 Zookeeper 和 Consul 的支持
- Spring Cloud：提供子项目如Spring Cloud Consul/Spring Cloud Zookeeper

参考资料

- [Open-Source Service Discovery](#)
- [Service Discovery in a Microservices Architecture](#)
- [服务发现方案梳理及NetflixEureka简介](#)

SmartStack

TBD

参考资料

- [SmartStack: Service Discovery in the Cloud](#)
- [SmartStack 介绍 —— 云端的服务发现: 上文的中文翻译版本](#)
- [SmartStack: Airbnb的自动服务发现和注册框架](#)

Eureka

介绍

- [eureka源码 @ github](#)
- [eureka wiki](#)

按照官方介绍：

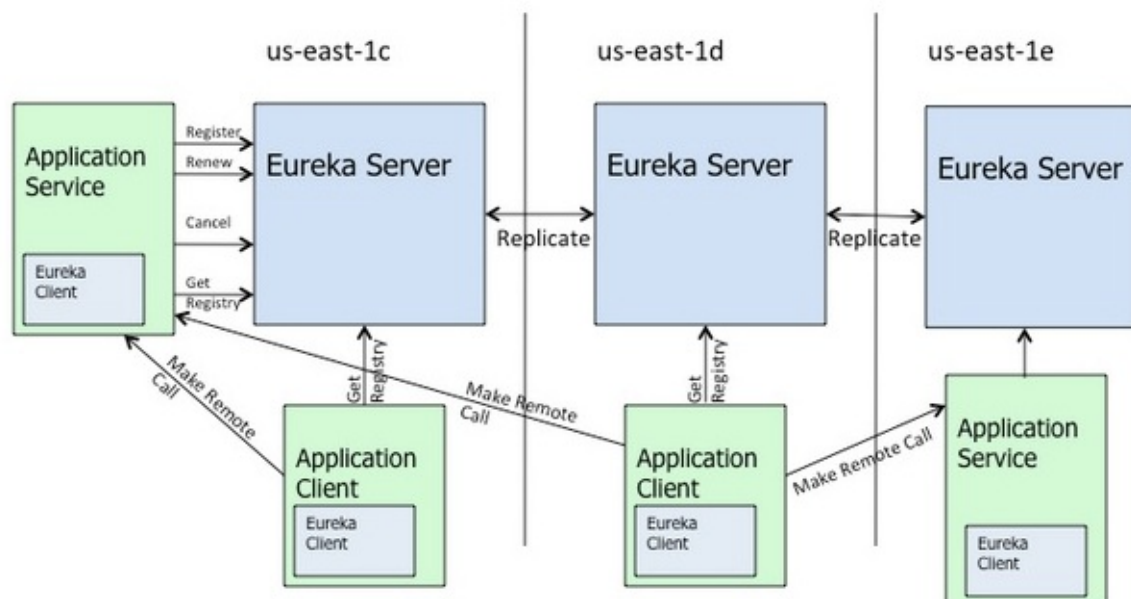
Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

Eureka 是一个基于 REST 的服务，主要在 AWS 云中使用，定位服务来进行中间层服务器的负载均衡和故障转移。

分析

工作方式

Eureka 提供服务注册与服务发现功能：



Eureka 是一个 RESTful 服务，使用 REST 进行通讯。

Eureka 用来定位运行在AWS地区中的中间层服务。由两个组件组成：

1. Eureka服务器：Eureka服务器用作服务注册服务器。
2. Eureka客户端：

Eureka客户端是一个java客户端，用来简化与服务器的交互、作为轮询负载均衡器，并提供服务的故障切换支持。

注：Netflix 在其生产环境中使用的是另外的客户端，它提供基于流量、资源利用率以及出错状态的加权负载均衡。

负载均衡

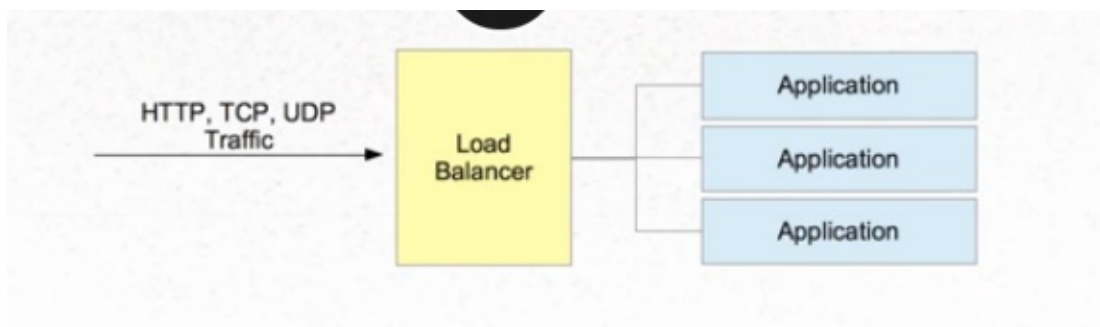
定义和做法

负载均衡就是分发请求流量到不同的服务器,

实现方式

服务器端负载均衡

服务器端负载均衡是指在服务器端架设负载均衡服务器, 用户请求到中间层的负载均衡服务器, 由负载均衡服务器分发控制到真实提供服务的应用服务器.



重点：

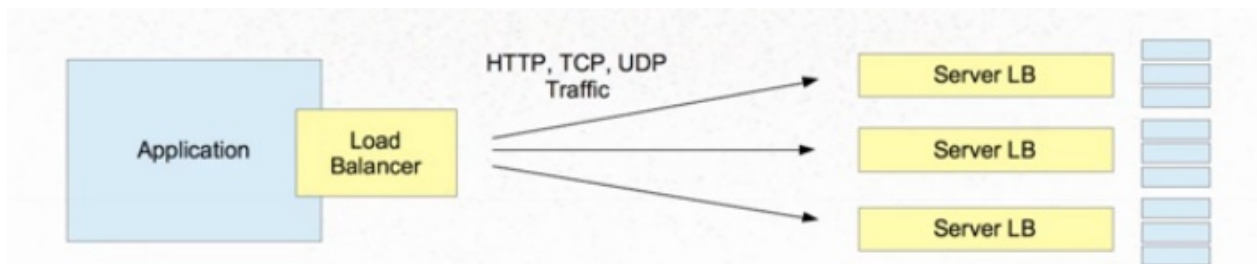
1. 对客户端透明的：客户端不知道服务器端的服务列表，甚至不知道自己发送请求的目标地址存在负载均衡器
2. 服务器端维护负载均衡服务器，控制负载均衡策略和算法

目前常见的服务器端实现有:

- 软件
 - Ngnix
 - HA Proxy
 - Apache
 - LVS
- 硬件
 - F5
 - NSX
 - BigIP

客户端负载均衡

客户端负载均衡是指负载均衡器作为客户端软件的一部分,客户端得到可用的服务器列表然后按照特定的负载均衡策略,分发请求到不同的服务器端.



重点：

1. 对客户端不透明的：客户端需要知道服务器端的服务列表，需要自行决定请求要发送的目标地址
2. 客户端维护负载均衡服务器，控制负载均衡策略和算法

目前单独提供的客户端实现比较少，只有：

- Netflix Ribbon

大部分都是在框架内部自行实现。

参考资料

- [负载均衡层设计方案（1）——负载场景和解决方式](#)
- [基于队列的负载均衡模式](#)

Netflix Ribbon

介绍

- <https://github.com/Netflix/ribbon>

Ribbon is a Inter Process Communication (remote procedure calls) library with built in software load balancers. The primary usage model involves REST calls with various serialization scheme support

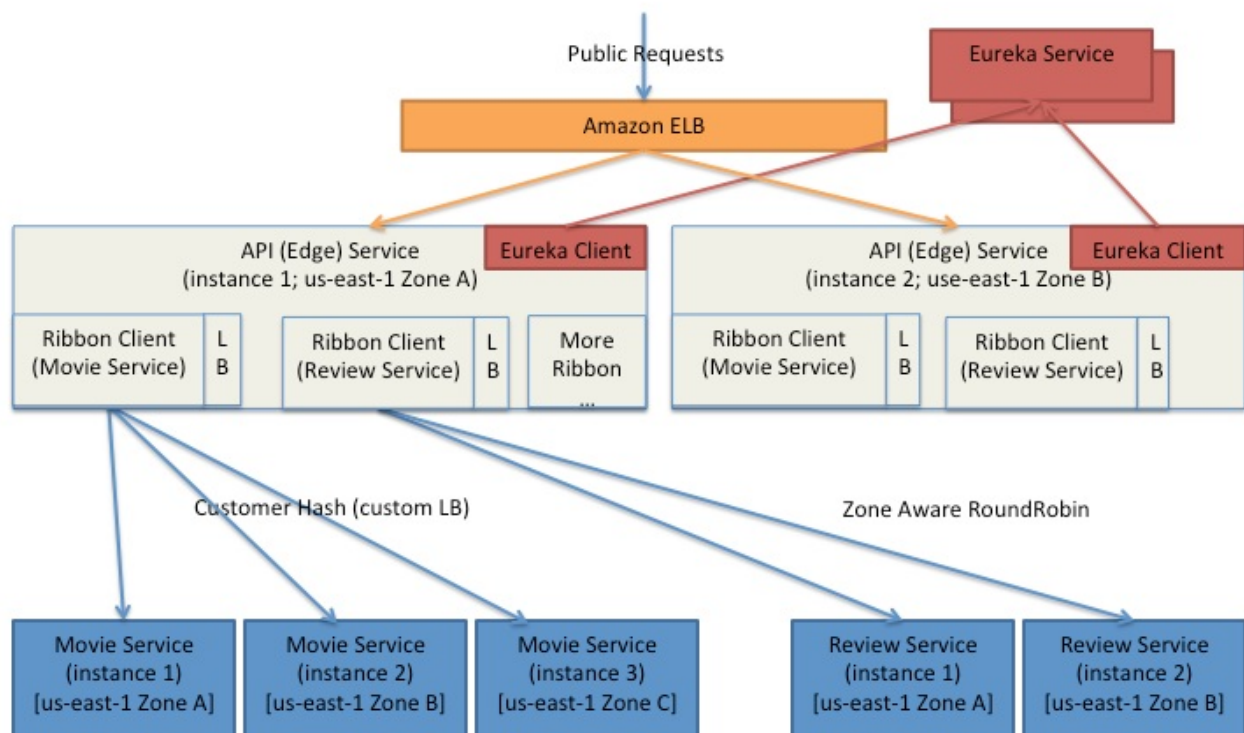
Ribbon 是一个进程间通讯 (RPC/remote procedure calls) 类库，内建软件负载均衡器。主要使用模式是围绕 Rest 调用，用各种序列化模式。

Ribbon 是客户端进程间通讯类库，在云中经受过考验。它提供下列特性：

- 负载均衡
- 容错
- 以异步和反应式模型执行多协议 (HTTP, TCP, UDP)
- 缓存和批量

分析

Ribbon 提供客户端负载均衡的支持，通常配合 Eureka 一起使用。



Ribbon客户端组件提供一系列完善的配置选项，比如连接超时、重试、重试算法等。Ribbon内置可插拔、可定制的负载均衡组件。下面是用到的一些负载均衡策略：

- 简单轮询负载均衡
- 加权响应时间负载均衡
- 区域感知轮询负载均衡
- 随机负载均衡

Ribbon中还包括以下功能：

- 易于与服务发现组件（比如Netflix的Eureka）集成
- 使用Archaius完成运行时配置
- 使用JMX暴露运维指标，使用Servo发布
- 多种可插拔的序列化选择
- 异步和批处理操作（即将推出）
- 自动SLA框架（即将推出）
- 系统管理/指标控制台（即将推出）

分布式配置管理

对于传统的单体应用，配置文件可以解决配置问题，但是当多机部署时，修改配置依然是个繁琐的问题。

在微服务中，由于系统拆分的粒度更小，微服务的数量比单体应用要多的多（基本上是多一个数量级），以配置文件来管理配置变得更多不可行。

所以，对于微服务架构而言，一个通用的分布式配置管理是必不可少的。在大多数微服务系统中，都会有一个名为 配置文件 的功能模块来提供统一的分布式配置管理。

Netflix Archaius

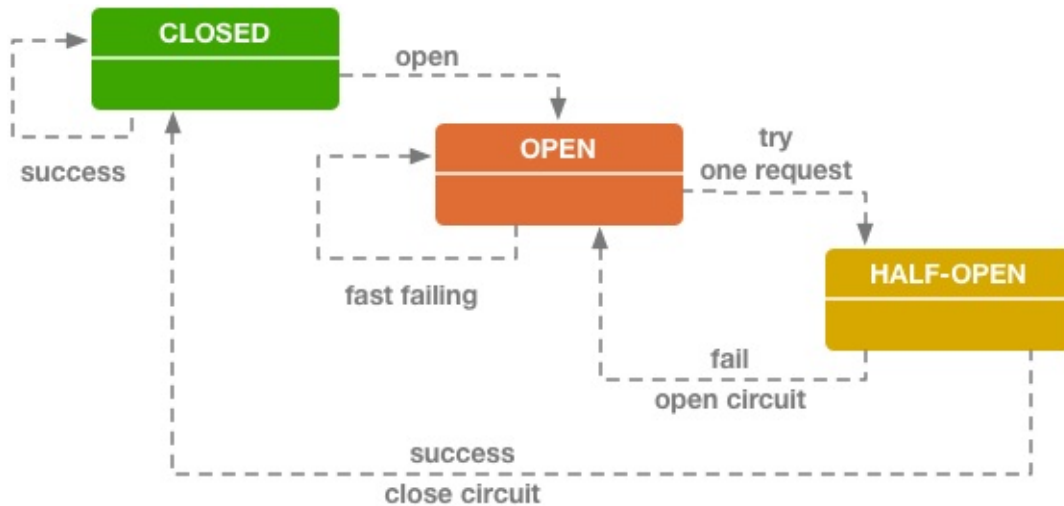
介绍

- <https://github.com/Netflix/archaius>
- <https://github.com/Netflix/archaius/wiki>

disconf

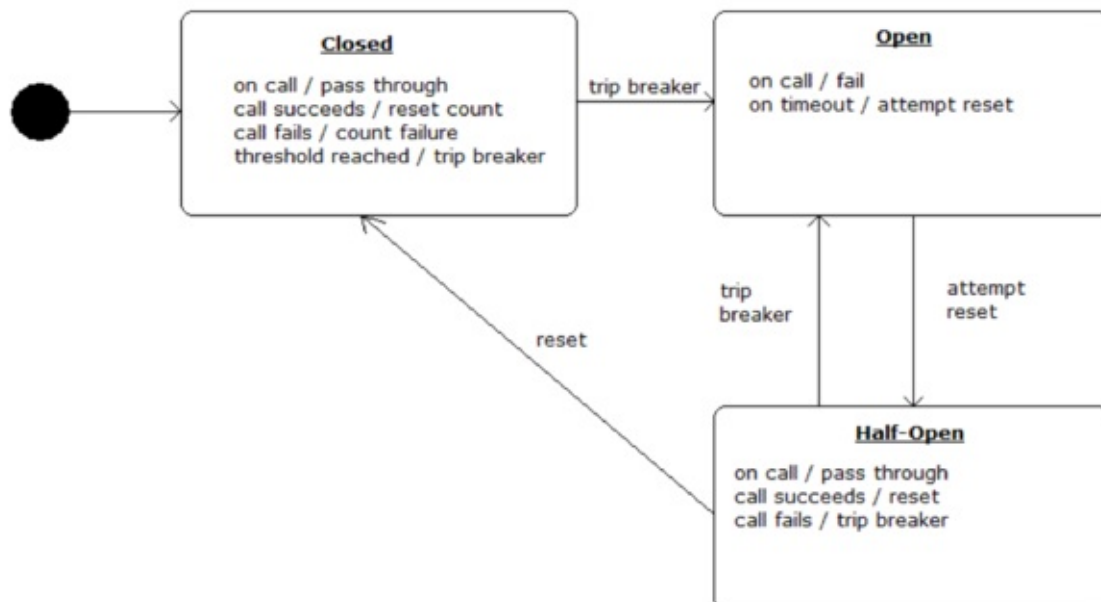
熔断器

断路器可以防止一个应用程序多次试图执行一个操作，即很可能失败，允许它继续而不等待故障恢复或者浪费 CPU 周期，而它确定该故障是持久的。



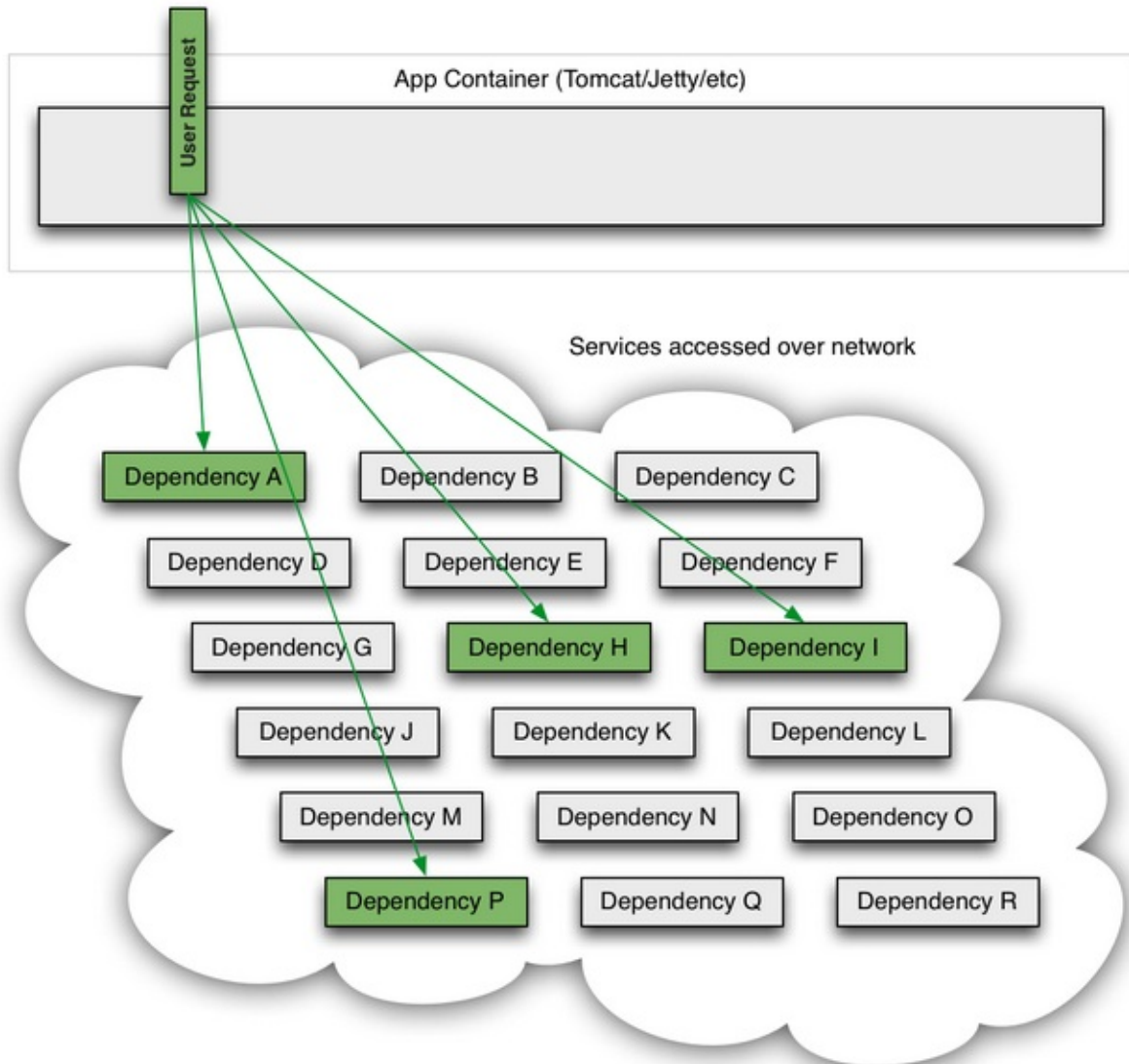
Circuit Breaker State Diagram

断路器模式也使应用程序能够检测故障是否已经解决。如果问题似乎已经得到纠正，应用程序可以尝试调用操作。

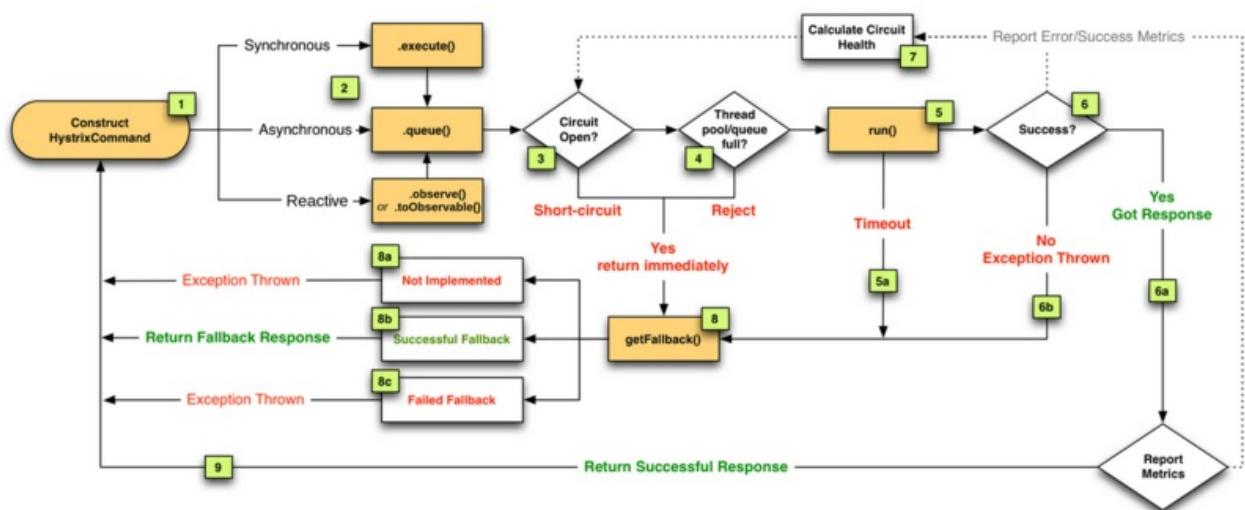


断路器增加了稳定性和灵活性，以一个系统，提供稳定性，而系统从故障中恢复，并尽量减少此故障的对性能的影响。它可以帮助快速地拒绝对一个操作，即很可能失败，而不是等待操作超时（或者不返回）的请求，以保持系统的响应时间。如果断路器提高每次改变状态的

时间的事件，该信息可以被用来监测由断路器保护系统的部件的健康状况，或以提醒管理员当断路器跳闸，以在打开状态。



流程图：



参考资料

- [CircuitBreaker \(by Martin Fowler\)](#)
- [Circuit Breaker Pattern](#)
- [断路器模式: ircuit Breaker Pattern 一文的中文翻译版本](#)

Netflix Hystrix

介绍



Hystrix: Latency and Fault Tolerance for Distributed Systems

Hystrix：用于分布式系统的延迟和错误容错。

- <https://github.com/Netflix/hystrix>

分析

Netflix Zuul

介绍

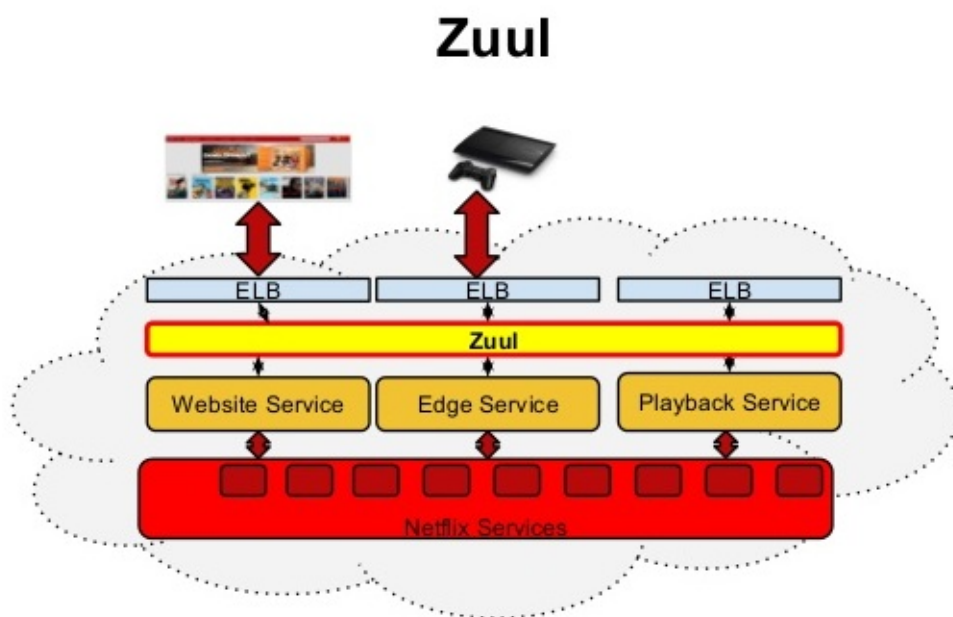
Zuul 是在云平台上提供动态路由，监控，弹性，安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。Zuul 可以适当的对多个 Amazon Auto Scaling Groups 进行路由请求。

分析

定位

Zuul 在 Netflix 的微服务体系中的定位：

1. Zuul只做跨横切面的功能: 如路由，安全认证，容错，限流，日志等
2. Zuul只是一个HTTP网关: 聚合或者协议转换(如rpc转换成http)，在Edge Service层做



参考资料

- [Netflix学习笔记：Zuul](#)

基础设施

Mantl

介绍



- <https://mantl.io/>
- <https://github.com/CiscoCloud/mantl>

Mantl 是 Cisco 开源的微服务平台，为用户提供了部署微服务平台所需的所有基础设施组件。

注：Mantl之前的名字是 "microservices-infrastructure" / "微服务-基础设置", 名字很直白但是稍微长了一点, 可能因为这个原因(我瞎猜的)所以后来更名为 mantl.

分析

定位

Mantl 对自己定义的描述：

The bedrock of microservices infrastructure

微服务基础设施的基岩

Mantl 为微服务基础设施提供端到端的解决方案。目标是让用户专注于应用程序的代码与业务的敏捷性，而不是如何设计基础设施API。

功能

以下功能介绍来自 Mantl 的高层 (Cisco Intercloud Services 的 CTO Ken Owens)访谈：

- Mantl为用户提供了部署微服务平台所需的所有基础设施组件。所选择的组件都是符合业界标准的，例如Docker、Mesos 和 Terraform，并且让他们能够良好地配合运行，使用户免于编写用于整合这些组件的代码。
- Mantl能够自动完成各种任务。这些任务在一般情况下往往需要投入几个月的时间进行DevOps。通过这种自动化能力，用户就能够专注于应用的开发，而无需在基础设施上投入过多精力。Mantl的目标是让用户专注于应用程序的代码与业务的敏捷性，而不是如何设计基础设施API。

- Mantl框架能够搭建出基础设施即服务以及软件即服务平台，使用户能够通过一种可编程的、可重复的方式部署他们的服务与应用。Mantl不限于所对应的云环境，它支持在多个云提供商的平台上进行部署，包括Rackspace、AWS、DigitalOcean和Google Cloud Platform等等。另外，用户也可以选择部署至私有云平台，包括OpenStack、VMWare、裸机，或者运行CentOS的任何系统。

评价

以下只是一家之言，仅供参考：

曾经推荐过 mantl，不过被诸多朋友诟病，大家都认为Cisco/思科这种"大公司病"比较严重的企业，推出的东东一定会很"重"，不适合。去 mantl 网站逛了一下，的确是有一种"厚重"的感觉.....

参考资料

- [通过Cisco发布的Mantl 1.0创建微服务基础设施](#)
- [变革中的Cisco：Matntl、Contiv、Shipped和Cisco Cloud](#)

Vamp

介绍



- <http://vamp.io>
- <https://github.com/magneticio/vamp/>

Vamp 是 "Very Awesome Microservices Platform" 的缩写, 翻译成中文, 就是大家耳熟能详的"一个神奇的微服务平台".

VAMP 由 Magnetic.io 打造, 是一个开源微服务部署平台, 该平台为开发、A/B测试、金丝雀发布 (Canary releasing)、自动缩放, 以及集成式度量指标和事件引擎提供了一种“平台中立”的微服务DSL”。

分析

定位

VAMP本身并不是一个完整的微服务/容器类PaaS/堆栈, 而只是专注于专门为通用微服务/容器堆栈提供高层次的金丝雀测试/发布和自动缩放功能。

VAMP能与很多容器和微服务平台集成, 例如Mesosphere的Open DC/OS、Apache Mesos/Mesosphere Marathon、Docker Swarm以及(很快即将支持的) Kubernetes, 当然还有诸如Cisco的Mantl、CapGemini的Apollo, 或Rancher Labs的Rancher等堆栈, 以及诸如MS Azure Container Service等容器云, 这些容器的编排程序都能与VAMP进行集成。

参考资料

- 专访VAMP创作者Olaf Molenveld: 为微服务平台探寻适合的抽象

微服务框架

其他

Restful 风格

Microserver

Microserver is a Java 8 native, zero configuration, standards based, battle hardened library to run Java Rest Microservices via a standard Java main class. Supporting pure Microservice or Micro-monolith styles.

- <http://micro-server.io/>
- <https://github.com/aol/micro-server>

dubbo

介绍

"DUBBO是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，是阿里巴巴SOA服务化治理方案的核心框架，每天为2,000+个服务提供3,000,000,000+次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点。"

注：这是来自 dubbo 官方的介绍，一个令人叹息的历史。



dubbo的一些资料：

- 官网 dubbo.io/
- [dubbo源码 @ github](#)
- [dubbo的微博](#)

分析

发展现状

dubbo 可以说是国内 SOA 框架（也可以拿来做微服务框架）的集大成之作，使用者众，影响力大。

但是，目前处境非常不好，在被阿里放弃之后 dubbo 项目接近废弃。

1. 阿里放弃了 dubbo

原阿里的 dubbo 开发团队已经解散，合并到 HSF，阿里也停止对 dubbo 的支持和后续更新。

阿里内部改为使用 HSF 框架，彻底断了 dubbo 在阿里和淘宝的路。

看到有评价，有些为 dubbo 可惜：

- "我觉得不是hsf更好，hsf在设计上和淘宝的遗留的项目接合的比较紧密。没有必要在迁移到dubbo上对原有的系统进行整改，并且加大风险。"
- HSF的缺点是其要使用指定的JBoss等容器，还需要在JBoss等容器中加入sar包扩展，对用户运行环境的侵入性大，如果你要运行在Weblogic或Websphere等其它容器上，需要自行扩展容器以兼容HSF的ClassLoader加载。"

2. dubbo 项目在2012年底之后基本就不再继续开发，后续也只有极少量的更新。

2012年是 dubbo 频繁发布的一年，如火如荼，然后2013年就嘎然而止，就此沉沦：



2014年最后一次发布，只有少量bugfix：

Latest release

 dubbo-2.4.11
 0e1752e

dubbo-2.4.11

 oldratlee released this on 30 Oct 2014 · **164 commits** to master since this release

- see [Issues](#)
- check [available dubbo versions](#) in maven center repository.


3. 当当 fork 并推出了 dubbbox


<https://github.com/dangdangdotcom/dubbbox>


DubboX 是基于 Dubbo 框架开发的 RPC 框架，支持 REST 风格远程调用，并增加了一些新的 feature


"dubbbox和dubbo 2.x是兼容的，没有改变dubbo的任何已有的功能和配置方式（除了升级了Spring之类的版本）"


DubboX 陆续有一些版本发布，但是基本都只是简单修复，后续没有任何功能性的发展计划：

 **dangdangdotcom / dubbbox**
forked from [alibaba/dubbo](#)

 Code


 Issues **84**


 Pull requests **32**




 Wi


Releases


Tags




on 31 Mar 2015 


dubbbox-2.8.4 


 3be9af7  zip  tar.gz




on 26 Nov 2014 

dubbbox-2.8.3 

 7635e79  zip  tar.gz

on 12 Nov 2014 

dubbbox-2.8.2 

 0c30bb2  zip  tar.gz

从目前看，基本看不到 dubbo 有任何死灰复燃的可能性。

总结

1. 非常好的一个 SOA 框架，极富研究价值，即便是在废弃多年后的今天
2. 如果当年能一直发展下来，前途不可限量.....
3. 可惜，没有如果，dubbo 已死，在生产上使用时请谨慎

注：个人对 dubbo 停止发展深表遗憾，这个项目本来可以成为一个伟大的项目，尤其在 2015 年微服务和 docker 盛行之后，本可以有更大的发挥空间。

Motan

介绍

Motan 是微博在2016年8月开源的用于高性能分布式服务快速开发的 RPC 框架,但是也兼具基本的服务化框架的功能。

Motan偏重于简洁实用的服务治理功能和优秀的RPC协议扩展能力,既可以提供高效的RPC远程调用,又能提供服务发现、服务高可用 (High Available)、负载均衡、服务监控、管理等服务治理功能。通过SPI机制提供强大的扩展能力,可以支持不同的RPC协议、传输协议。Motan能够无缝支持Spring配置方式使用RPC服务,通过简单、灵活的配置就可以提供或使用RPC服务。通过使用Motan框架,可以十分方便的进行服务拆分、分布式服务部署。

- [Motan源码 @ github](#)

微博解释他们开发 Motan 而不是使用 dubbo 的原因：

Dubbo 功能上比较丰富,但当时我们想要一个比较轻量的 RPC 框架,方便我们做一些适合自己业务场景的改造和功能 feature,以达到内部业务平滑改造和迁移的目的。这种情况下,在 dubbo 上改的成本可能比重新写一套更高。最终我们决定开发 motan RPC。

分析

更新情况

Motan 相比 dubbo,就如同活力四射的少年一般,充满生机和活力,这是 github 上的更新情况：

 rayzhang0603	fix roundrobin lb index overflow	Latest commit 5271da9 4 days ago
 docs/wiki	update wiki	13 days ago
 motan-benchmark	update changelog && new version 0.2.2-SNAPSHOT	26 days ago
 motan-core	fix roundrobin lb index overflow	4 days ago
 motan-demo	add yar protocol demo	20 days ago
 motan-extension	Merge pull request #183 from andot/master	13 days ago
 motan-manager	update changelog && new version 0.2.2-SNAPSHOT	26 days ago
 motan-registry-consul	update changelog && new version 0.2.2-SNAPSHOT	26 days ago

当然,年轻也意味着可能不够成熟,在稳定性和可能出现的问题尚待检验。

注：我们曾经在Motan开源的第一天就下载下来研究,结果 helloworold 都没有跑起来,直接 NullPointerException,里面代码让人哭笑不得,大意是: `if (a == null) {a.dosomething()}`

功能

功能上，对比成熟而完善的 dubbo，motan 的功能还嫌单薄，不过 motan 本来也声称是轻量级 RPC 框架，定位不同，情有可原。Motan 的解释：

与同类型的 Dubbo 相比，Motan 在功能方面并没有那么全面，也没有实现特别多的扩展，但 Motan 是一个小而精的 RPC 框架，它的特点是简单、易用，是一个不断向着实用、易用方向发展的 RPC 服务框架。

Motan 提供的主要功能包括：

- 服务发现：服务发布、订阅、通知
- 高可用策略：失败重试（Failover）、快速失败（Failfast）、异常隔离（Server 连续失败超过指定次数置为不可用，然后定期进行心跳探测）
- 负载均衡：支持低并发优先、一致性 Hash、随机请求、轮询等
- 扩展性：支持 SPI 扩展（service provider interface）
- 其他：调用统计、访问日志等

Motan 功能特点：简单、易用、高可用

- 无侵入集成、简单易用，通过 Spring 配置方式，无需额外代码即可集成分布式调用能力。
- 集成服务发现和服务治理能力，灵活支持多种配置管理组件，如 Consul、ZooKeeper 等。
- 支持自定义动态负载均衡、跨机房流量调整等高级服务调度能力。
- 基于高并发、高负载场景优化，具备 Failover、Failfast 能力，保障 RPC 服务高可用。

技术栈

微博的 Motan RPC 服务，底层通讯引擎采用了 Netty 网络框架，序列化协议支持 Hessian2 和 Java 序列化，通讯协议支持 Motan、http、tcp、mc 等。

Motan 除了支持常见的 zookeeper 之外，还支持使用 consul 作为注册中心。

应用场景

在应用方面，目前主要是微博自己大量使用，考虑到微博平台的超大规模，应该说是经历了实战考验。

2013 年微博 RPC 框架 Motan 在前辈大师们（福林、fishermen、小麦、王喆等）的精心设计和辛勤工作中诞生，向各位大师们致敬，也得到了微博各个技术团队的鼎力支持及不断完善，如今 Motan 在微博平台中已经广泛应用，每天为数百个服务完成近千亿次的调用。

Motan 是微博技术团队研发的基于 Java 的轻量级 RPC 框架，已在微博内部大规模应用多年，每天稳定支撑微博上亿次的内部调用。

总结

1. 年轻，更新积极，有极大的发展潜力
2. 刚出来，功能和稳定性有待观望
3. 轻量级，入门门槛比 dubbo 低
4. 对跨语言调用支持较差，主要支持java

Netflix OSS

介绍

- <https://netflix.github.io/>

Netflix OSS 指的是 "Netflix Open Source Software"

这里我们关注的是 OSS 中的 `cloud platform`，也就是 "Common Runtime Services & Libraries"，包括为微服务提供支持的运行时容器，类库和服务。

Netflix OSS 是一组开源的框架和组件库，是Netflix公司开发出来解决分布式系统的一些有趣的可扩展类库。对于Java开发者来说，它们是在云端环境中开发微服务的非常棒的工具代名词。在服务发现，负载均衡，容错等模式方面，都给出了非常重要的概念，并带来了漂亮的解决方案。

分析

组件和功能

主要组件包括：

- [Eureka](#)：服务注册和服务发现
- [Archaius](#)：分布式配置管理
- [Ribbon](#)：弹性而智能的进程间和服务通讯机制，客户端负载均衡
- [Hystrix](#)：熔断器，在运行时提供延迟和容错的隔离
- [Karyon](#)：
- [Governator](#)
- [Prana](#)
- [Zuul](#): 服务网关
- [Fenzo](#)

总结

Netflix OSS 是一套非常好的组件，很多情况下我们可以考虑先看看 Netflix OSS 是否有提供现成的方案，再看是否有必要自己动手造轮子。

但是，Netflix OSS 也有自生的一些不足，典型原因就是 Netflix OSS 问世比较早，在多数 Netflix 的开源项目开发的时期，只有 AWS 公有云可以选择而没有其他的替代。这个因素导致这些库并不是直接为今天采用的运行环境（如 Linux 上的容器）开发的。

如果开始构建微服务的方法，肯定容易被 Netflix OSS/Java/Spring/SpringCloud 所吸引。但是要你知道你不是 Netflix，也不需要直接使用 AWS EC2，使得应用程序变得很复杂。如今使用 docker 和采用 memos/kubernetes 是明智之举，它们已经具备大量的分布式系统特性。在应用层进行分层，是因为 netflix 5 年前面临的问题，而不得不这样做（可以说如果那时有了 kubernetes，netflix OSS 栈会大不相同）。

因此，建议谨慎选择，按需选择，避免给应用程序带来不必要的复杂度。

参考资料

- [Netflix OSS, Spring Cloud 以及 Kubernetes？关于它们的种种！](#)

SpringBoot

介绍

Spring Boot是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Boot致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

Spring Boot的目标不在于为已解决的问题域提供新的解决方案，而是为平台带来另一种开发体验，从而简化对这些已有技术的使用。对于已经熟悉Spring生态系统的开发人员来说，Boot是一个很理想的选择，不过对于采用Spring技术的新人来说，Boot提供一种更简洁的方式来使用这些技术。

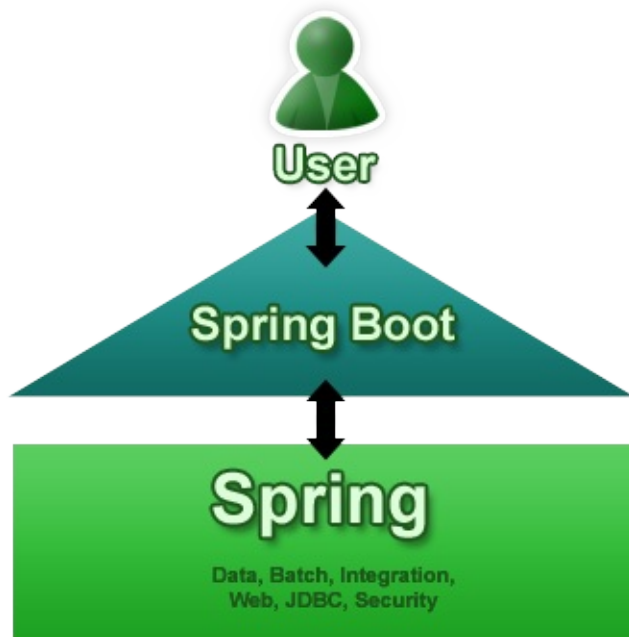
- <http://projects.spring.io/spring-boot/>

分析

目标定位

Spring 框架是非常著名的 Java 开源框架，历经十多年的发展，整个生态系统已经非常完善甚至是繁杂，Spring Boot 正是为了解决这个问题而开发的，为 Spring 平台和第三方库提供了开箱即用的设置，只需要很少的配置就可以开始一个 Spring 项目。

Spring Boot 的定位：



功能

Spring Boot 的主要功能：

- 创建独立 Spring 应用
- 直接内嵌 Tomcat，Jetty 或者 Undertow (不需要部署 WAR 文件)
- 提供 'starter' POM 文件来简化 maven 配置
- 尽可能的自动配置 Spring
- 提供产品级别的功能如 metrics，健康检查和外部化配置
- 完全没有代码生成并不需要 XML 配置

总结

1. Spring Boot 是一个"微框架"

离微服务框架还是有很大距离,比如没有提供最基本的服务注册和服务发现。

当然 Pivotal 团队为此推出了解决方案 Spring Cloud。

2. Spring Boot 更适合作为一个微服务框架的基石

推荐在 Spring Boot 的基础上，而不是 Spring 的基础上搭建自己的微服务框架。

3. Spring Boot 特别适合已经熟悉 Spring 体系的开发团队

Spring Cloud

介绍

- <http://projects.spring.io/spring-cloud/>

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

分析

目标定位

前面谈到 Spring Boot的特点比较适合用来做微服务的基础框架,但是要开发一个完整的微服务系统远没有这么简单. 因此 Pivotal 为此推出了 Spring Cloud.

Spring Cloud 完全基于 Spring Boot，是一个非常新的项目，2016年才 1.0 release。版本提升非常迅速，发展势头良好，看 Pivotal 的意义应该是冲着占领企业微服务架构这个领域去的。

子项目

Spring Cloud包含了多个子项目（针对分布式系统中涉及的多个不同开源产品，有近20个之多），比如：

- Spring Cloud Config

中心化的外部配置管理，由 git 仓库支持。配置资源直接映射到 Spring Environment 但是如果有需要也可以被非Spring应用使用。

注：这个是用来解决微服务环境下配置文件分散管理的难题，但是奇怪的是为什么要基于 git 仓库？

- Spring Cloud Netflix

和多个 Netflix OSS 组件集成(Eureka, Hystrix, Zuul, Archaius, etc.).

这个算是最有价值的部分，下一节单独详细介绍。

- Spring Cloud Bus

用于将服务和实例连接到分布式消息的 **event bus**。用于在集群内传播状态变更。
(如配置变更事件)

- **Spring Cloud CloudFoundry**

集成应用到 **Pivotal Cloudfoundry**。提供服务发现实现，并可以简化实现 **SSO** 和 **OAuth2** 保护的资源，也可以创建 **Cloudfoundry** 服务中介。

- **Spring Cloud Cloud Foundry Service Broker**

提供构建服务中介的起点，管理 **Cloud Foundry** 管理的服务。

- **Spring Cloud Cluster**

leader 选举和通用有状态模式，有抽象和用于 **Zookeeper**, **Redis**, **Hazelcast**, **Consul** 的实现。

- **Spring Cloud Consul**

使用 **Hashicorp** 公司的 **Consul** 来进行服务发现和配置管理。

- **Spring Cloud Security**

为负载均衡的 **OAuth2 rest** 客户端和在 **Zuul** 代理中认证 **header** 转发提供支持。

- **Spring Cloud Sleuth**

用于 **Spring Cloud** 应用的分布式追踪，兼容 **Zipkin**，**HTrace** 和基于 **log**(如 **ELK**) 的追踪。

- **Spring Cloud Stream**

消息中间件抽象层，目前支持 **Redis**, **Rabbit MQ** 和 **Kafka**

- **Spring Cloud Zookeeper**

使用 **Apache Zookeeper** 做服务发现和配置管理。

总结

1. 很年轻的项目，可以关注，前景看好
2. 但是很少见到国内业界有人在生产上成套使用，一般都是只有其中一两个组件

参考资料

- [SpringCloud分布式开发五大神兽](#)

Spring Cloud Netflix

介绍

- <http://cloud.spring.io/spring-cloud-netflix/>
- <http://github.com/spring-cloud/spring-cloud-netflix>

分析

总结

Tags