WOTA

51CTO

World Of Tech 2017

全球架构与运维技术峰会

2017年4月14日-15日　北京富力万丽酒店

ARCHITECTURE

WOTA

51CTO

出品人及主持人：

沈　剑

58到家 高级技术总监

微服务架构实践

WOTA

51CTO

微服务在大型互联网公司的应用及其挑战

WOTA

51CTO

罗轶民

LinkedIn

资深软件工程技术经理

分享主题：

微服务在大型互联网公司的应用及其挑战

罗轶民

- 互联网构架以及技术团队管理专家
- 现LinkedIn（领英）软件工程经理
- 前乐视美国视频平台工程技术总监
- 前Netflix(奈飞）视频内容平台工程技术负责人
- 前PayPal资深软件工程构架师
- 主要负责的领域是高并发后端服务构架，微服务构架，大数据平台构架等，以及端对端的整个产品开发

Agenda

- 微服务产生的历史
- 微服务和单片服务(Monolithic service)之间的区别
- 软件构架从单片服务向微服务转型过程中带来的技术挑战
- 软件构架从单片服务向微服务转型过程中带来的企业组织构架的变化和挑战
- 如何合理地选择单片服务构架和微服务构架

微服务的起源和历史

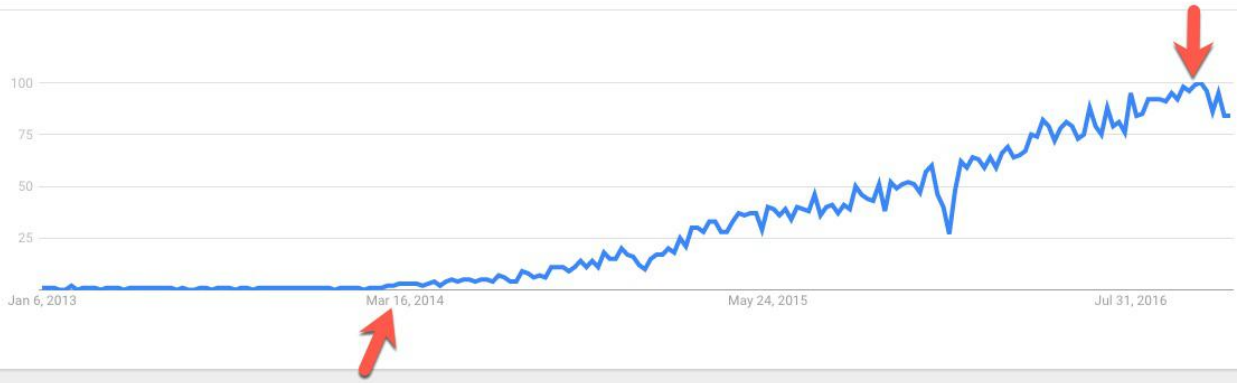
Google Trends | Explore

Microservices
Search term

+ Compare

Worldwide | 1/1/13 - 11/30/16 | All categories | Web Search

Interest over time



Interest by region

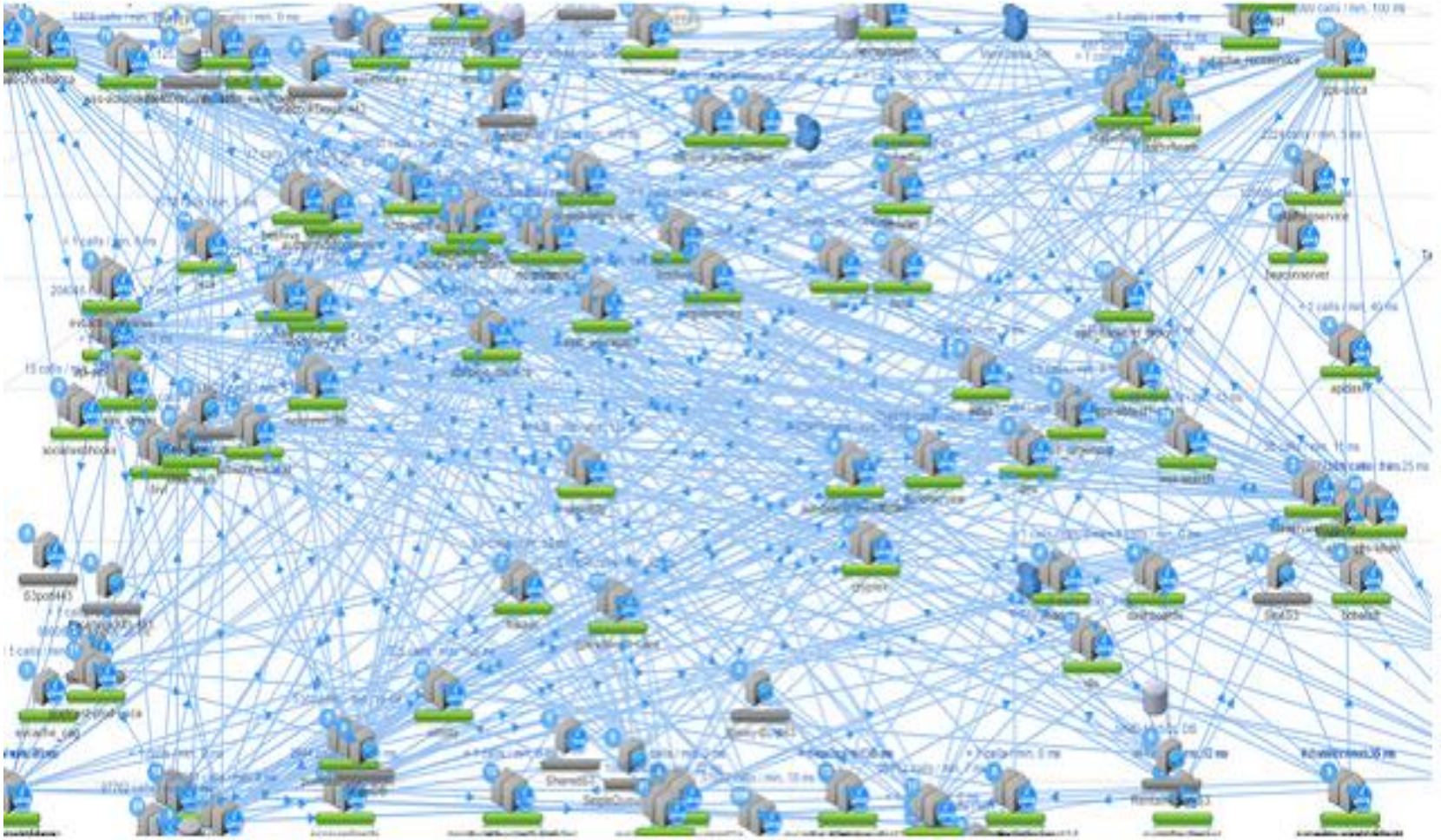
Region

1	India	100
2	Australia	87
3	Germany	71
4	United Kingdom	68
5	United States	65

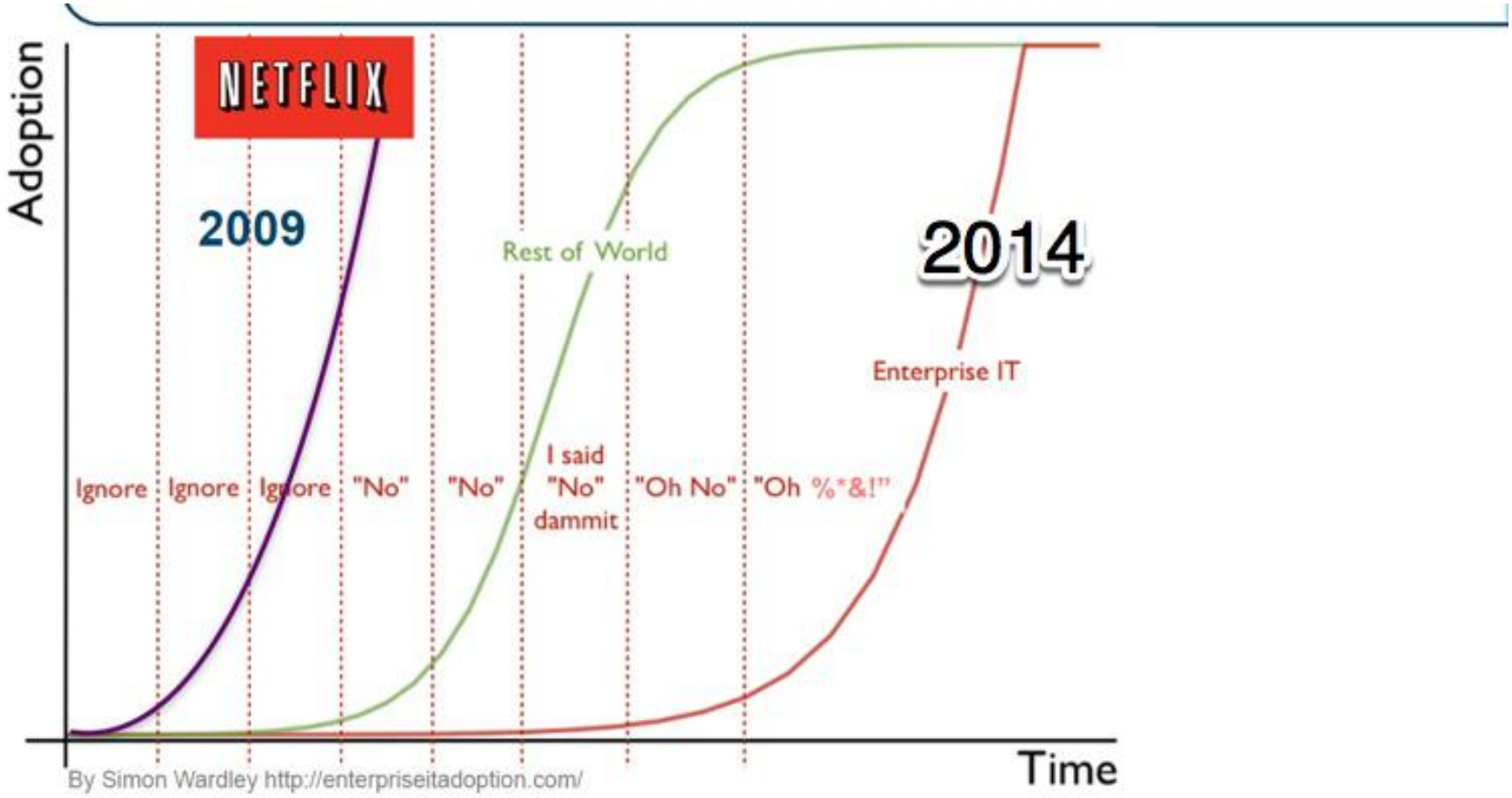
- 2012年硅谷一些科技团队开始用微服务这个词
- 2014年开始得到更广泛的关注度
- Netflix 在2008 – 2009 年开始就探索微服务构架，后来也发布了很多OPEN SOURCE的微服务工具，以下会以Netflix 为例子来讲讲微服务的演变过程以及带来的挑战

Netflix 微服务演化

2009： Monolithic app in its own datacenter
2010： Movie Encoding -> AWS with Microservices architect (non customer facing app)
2011 - 2013: Other customer facing apps: account service, movie selection service, metadata service etc.
2013 – 2014: Open source Netflix Microservices tools
2014 - 2015: all the service in cloud, 600+ services

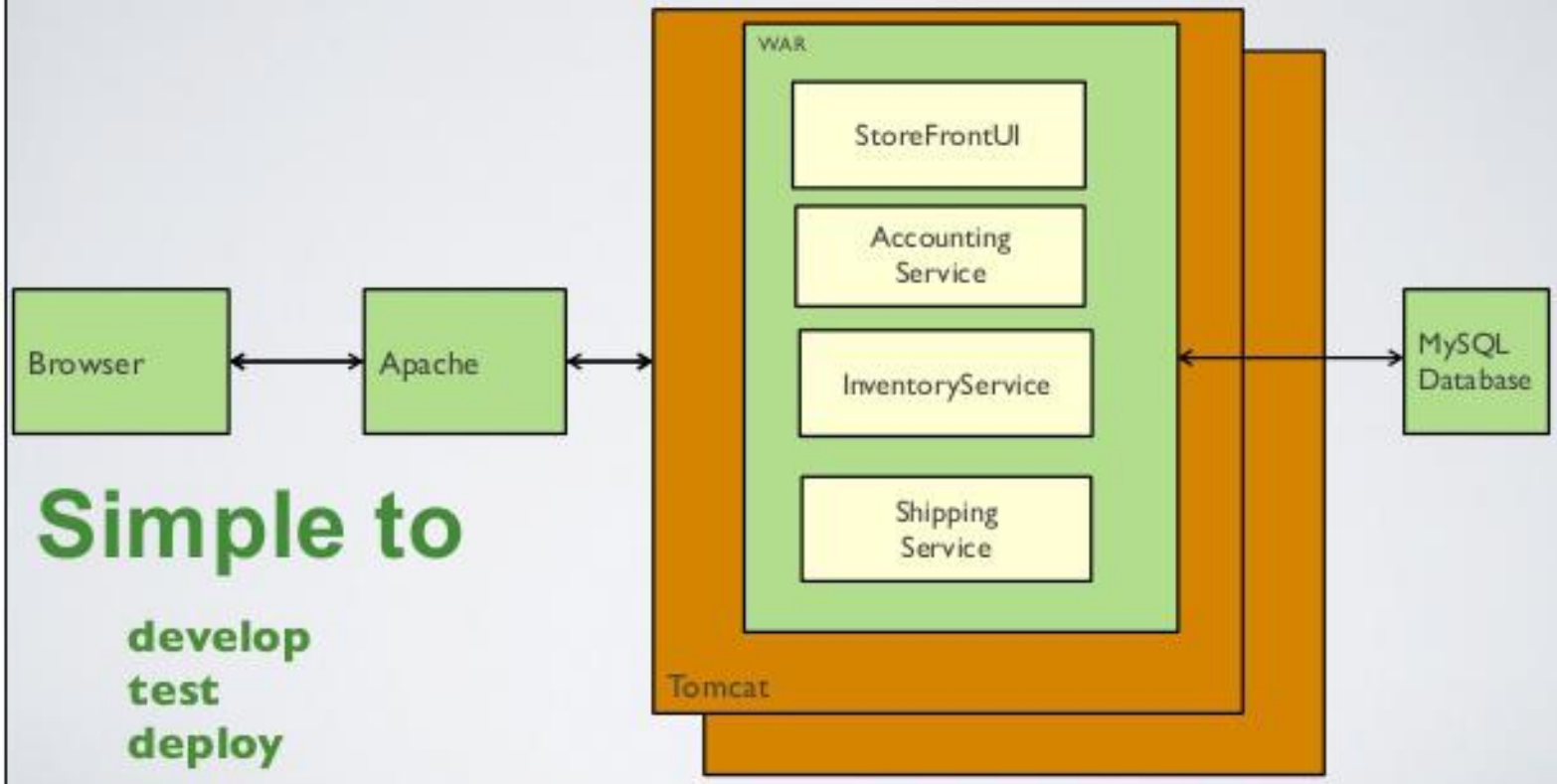


Microservices on cloud adoption



Monolithic app

Traditional web application architecture



Simple to

develop
test
deploy
scale

Monolithic APP 优点

- 易于开发，很多IDE和框架都支持，比如SPRINT MVC，RUBY RAILS，PYTHON DJANGOO等。
- 易于测试。 可以通过简单启动应用程序并使用Selenium测试UI来实现端到端测试。
- 易于部署。 只需将打包的应用程序复制到服务器。
- 通过在负载平衡器后运行多个副本，可以轻松地水平扩展。
- DEV Op比较简单，一支专门DEVOP 团队负责即可

Monolithic APP 缺点

- 应用程序太大且复杂，很难完全理解并快速正确地进行更改。
- 应用程序的会越变越大，可能会减慢启动时间。
- 必须在每次更新时重新部署整个应用程序。
- 如果代码库有新的变化，变化的影响通常不是很清楚，这导致广泛的手动测试。

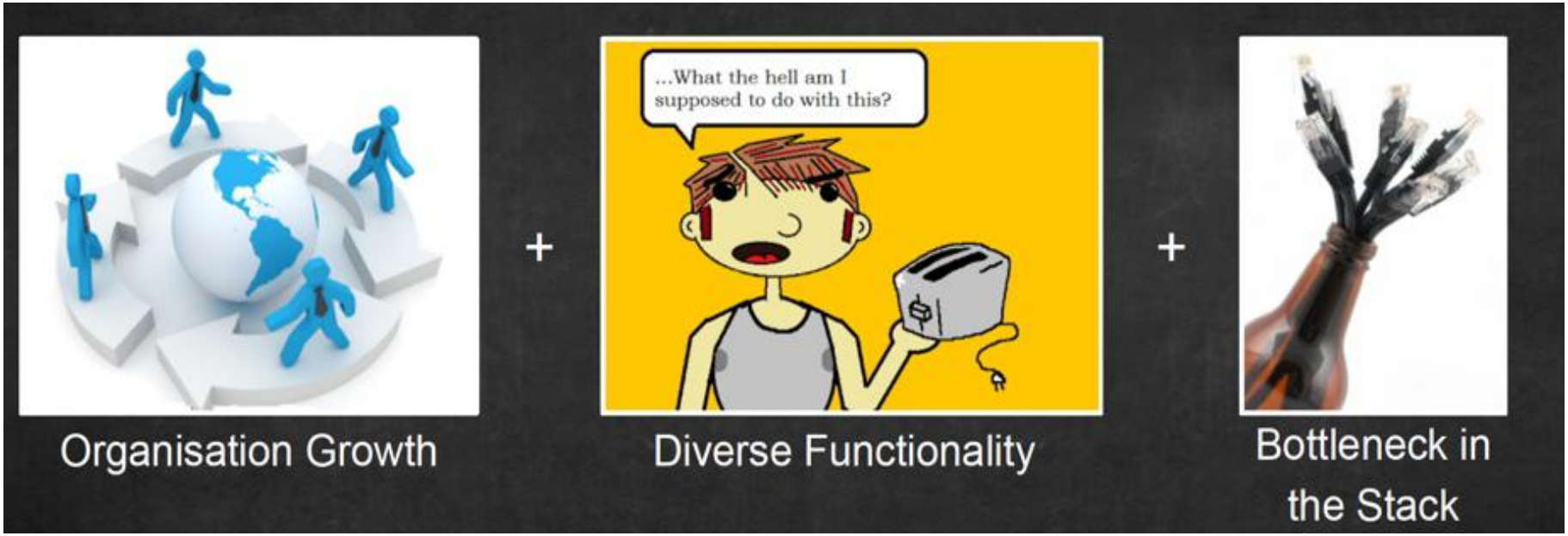
Monolithic APP 缺点

- 连续部署很困难
- 当不同模块具有冲突的资源需求时，单片应用也可能难以扩展
- 可靠性差。 任何模块中的错误（例如内存泄漏）都可能会导致整个过程。 此外，由于应用程序的所有实例是相同的，该错误将影响整个应用程序的可用性
- 采用新技术或框架很困难。 由于框架或语言的变化将影响整个应用程序，因此在时间和成本上都是非常昂贵的

随着代码库，组件和团队规模增长，各种问题会相继出现：

- 原代码太大，IDE打不开了
- 单机的内存不够，没法编译和跑代码
- 部署一次要花很长时间
- 开发速度跟不上产品的需求，一个小小的变化需要整个源代码重新编译
- 某一个模块里的一个小错误，可能导致整个网站宕机

随着组织的成长，功能的增多以及技术栈的瓶颈出现，需要有新的变革



什么是微服务？

团队大小？ No

代码行数的大小？ No

API 端口的数目大小？ No

Loosely coupled service oriented architecture with **bounded contexts**

- 独立部署
- 独立技术栈
- 有界上下文
- 明确的所有权


TEAM
WORK

Unix philosophy

Write programs that do one thing and do well.

Write programs that work together.

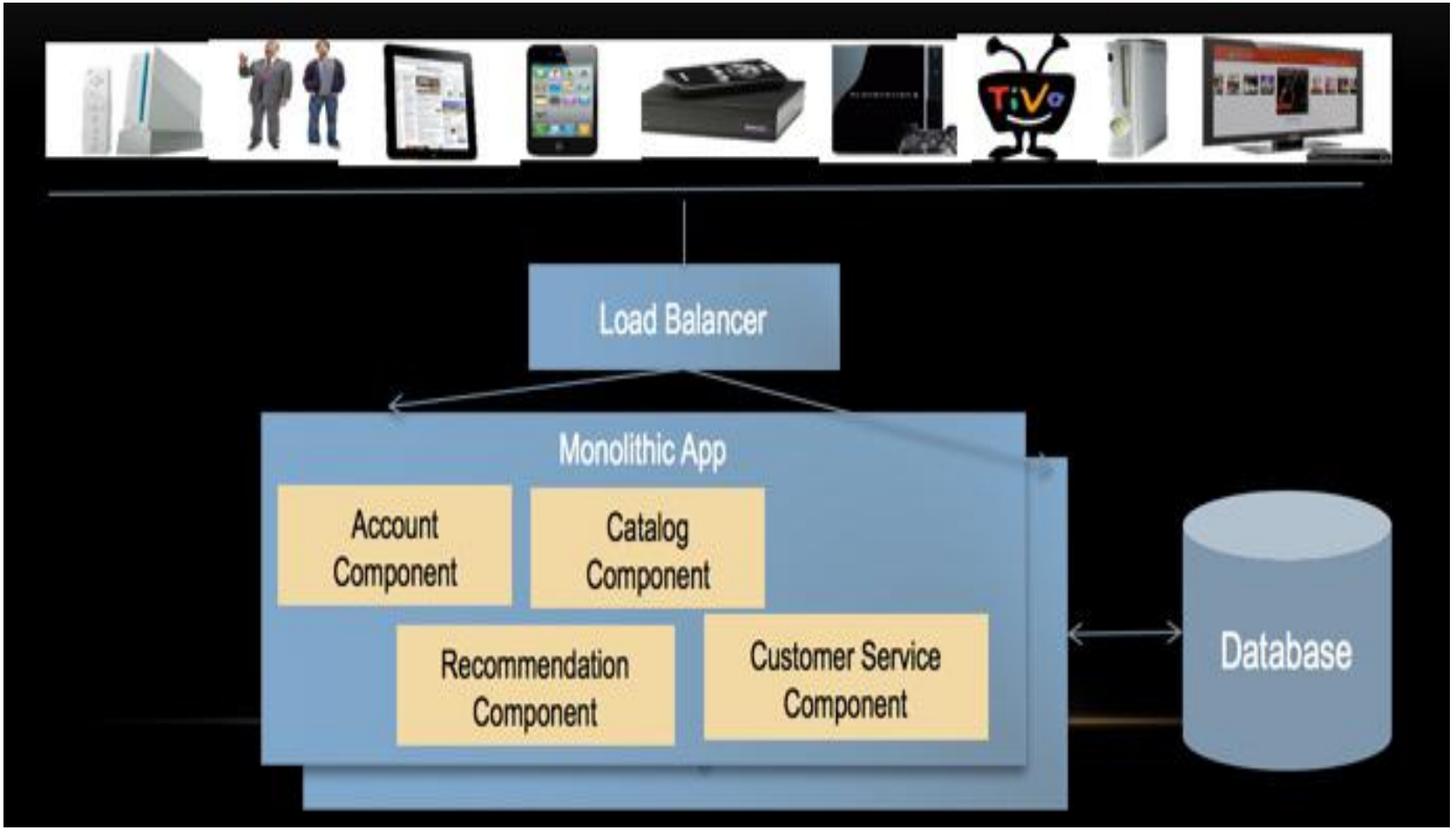
```
tr 'A-Z' 'a-z' < doc.txt | tr -cs 'a-z' "\n" | sort | uniq
| comm -23 ~/dict.txt
```

微服务和单片服务的对比

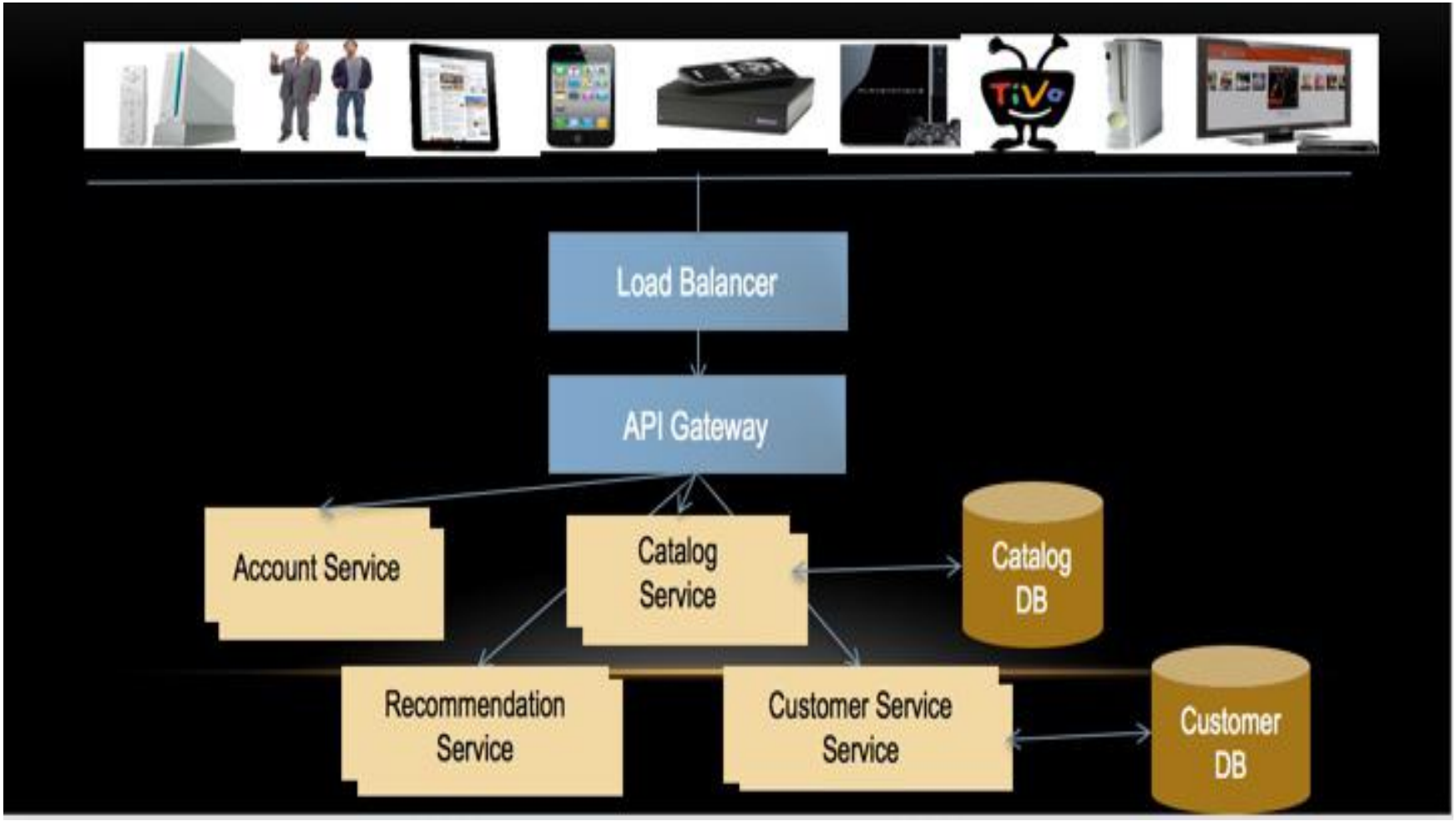
微服务和单片服务的对比

- 在MONOLITHIC APP中，所有的部件都在一个巨大的软件包中
- 在微服务的构建下，有很多独立存在的小服务，通过API接口连接成大的系统

Monolithic APP



微服务构架


Load Balancer
API Gateway
Account Service
Catalog Service
Catalog DB
Recommendation Service
Customer Service Service
Customer DB

为什么NETFLIX要用微服务？

- **可用性(Availability)**
  24 x 7
  防止单点失败（single point of failure）

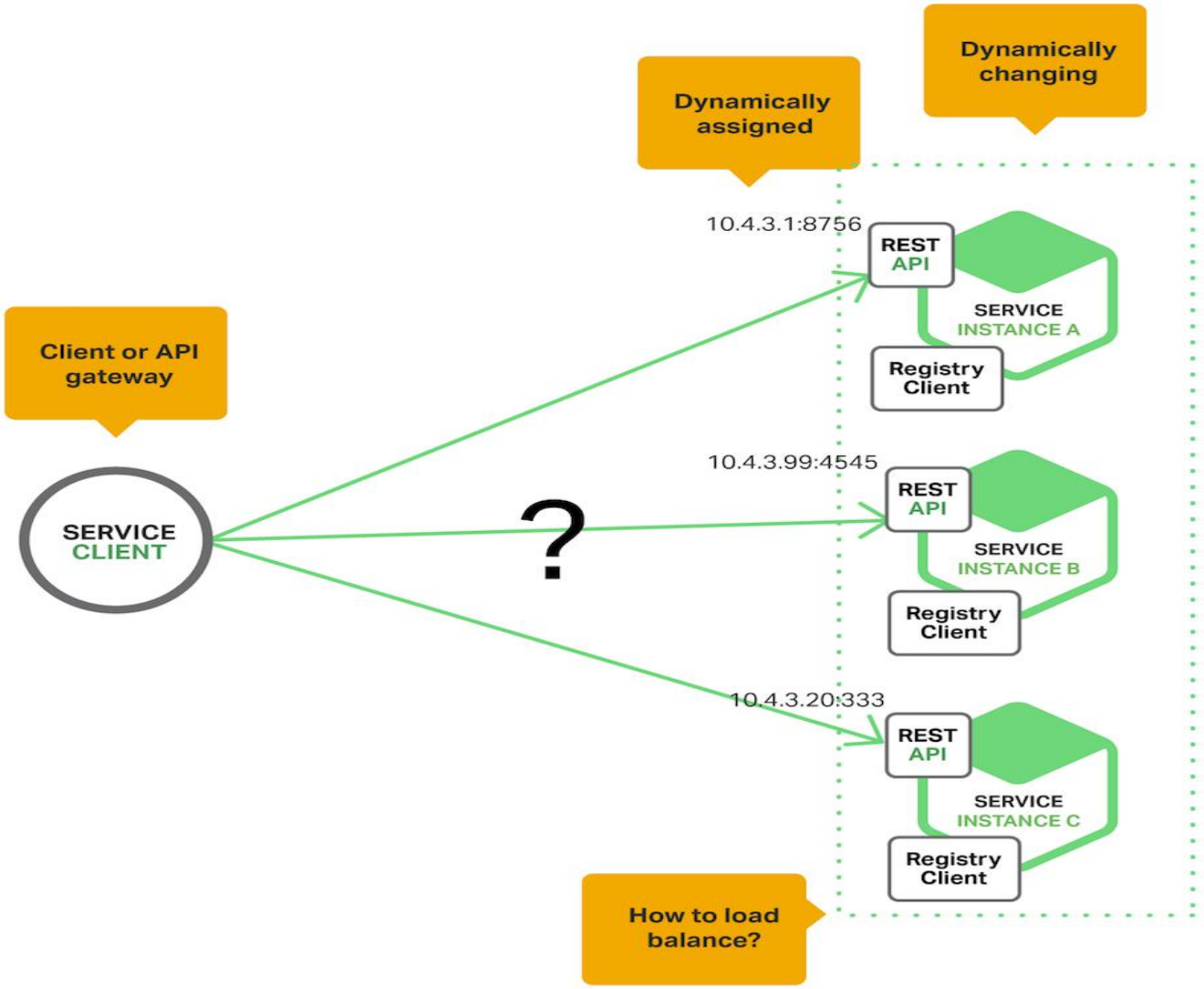
- **可拓展性**
  1/3 的互联网流量
  超过9000W的付费用户

- **速度**
  快速推出新的功能
  速度是在互联网时代致胜的关键

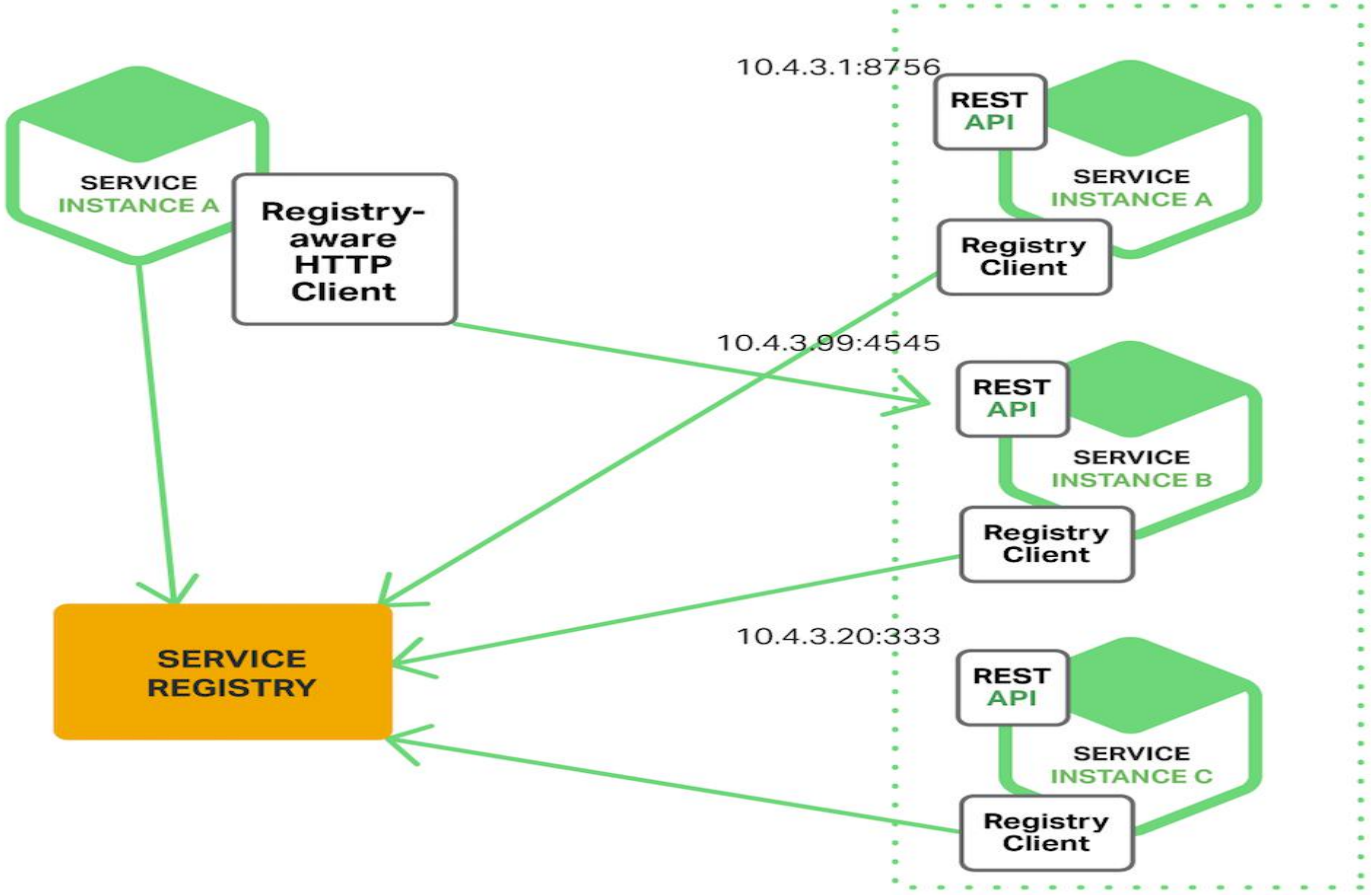
微服务带来的技术挑战

- 服务发现(Service discovery)
- 运维复杂度增加 – DevOps
- 分布式系统本质上带来的复杂度的
- 网络延迟，容错
- 服务接口版本控制，不匹配？
- 测试（需要整个生态系统来测试）
- FAN OUT - >增加网络流量

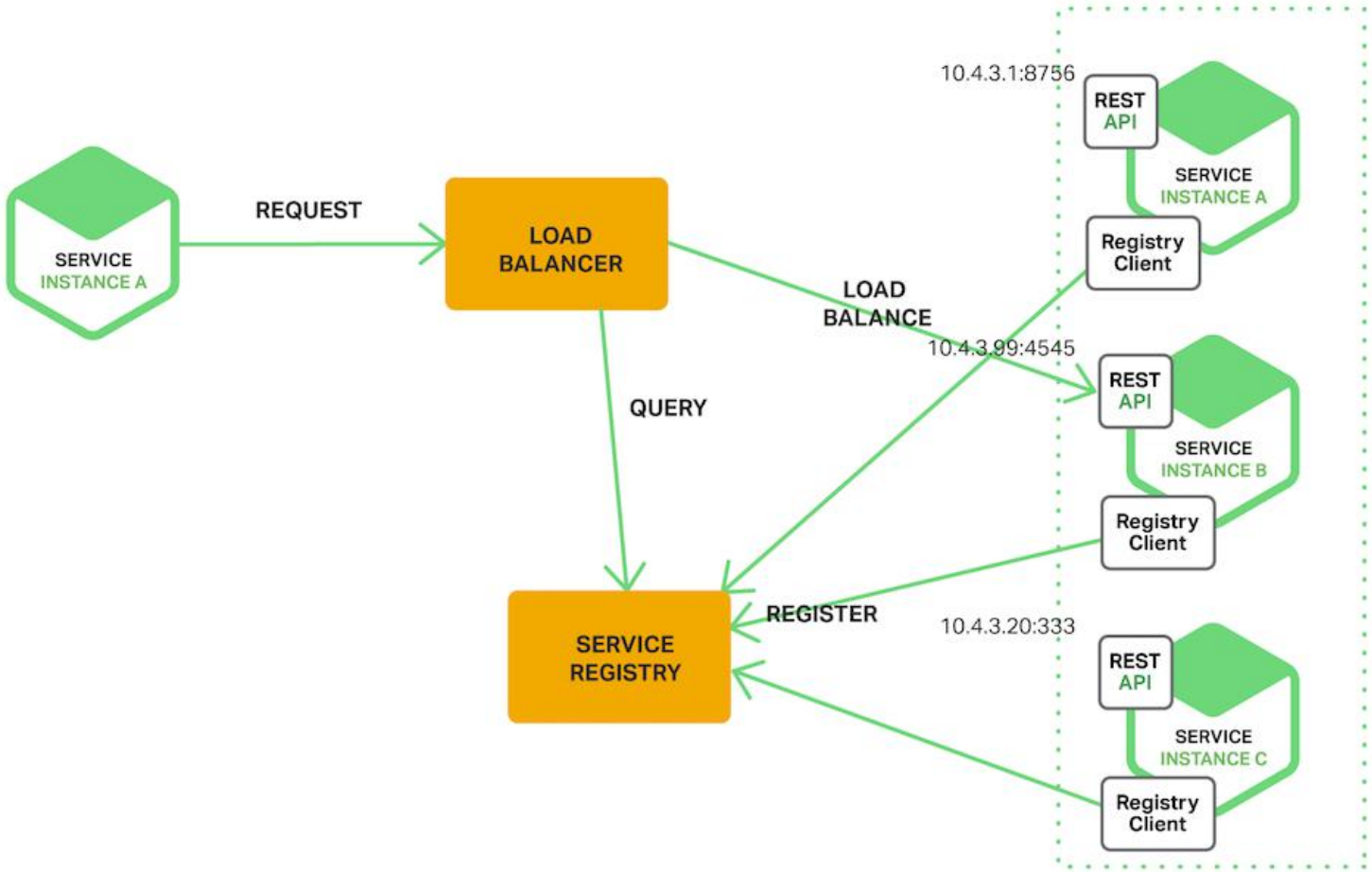
Service discovery



服务发现 – 客户端


SERVICE
INSTANCE A
Registry-
aware
HTTP
Client
SERVICE
REGISTRY
10.4.3.1:8756
REST
API
SERVICE
INSTANCE A
Registry
Client
10.4.3.99:4545
REST
API
SERVICE
INSTANCE B
Registry
Client
10.4.3.20:333
REST
API
SERVICE
INSTANCE C
Registry
Client

服务发现 – 服务器端


SERVICE
INSTANCE A
REQUEST
LOAD
BALANCER
LOAD
BALANCE
QUERY
REGISTER
SERVICE
REGISTRY
10.4.3.1:8756
REST
API
SERVICE
INSTANCE A
Registry
Client
10.4.3.99:4545
REST
API
SERVICE
INSTANCE B
Registry
Client
10.4.3.20:333
REST
API
SERVICE
INSTANCE C
Registry
Client

服务注册

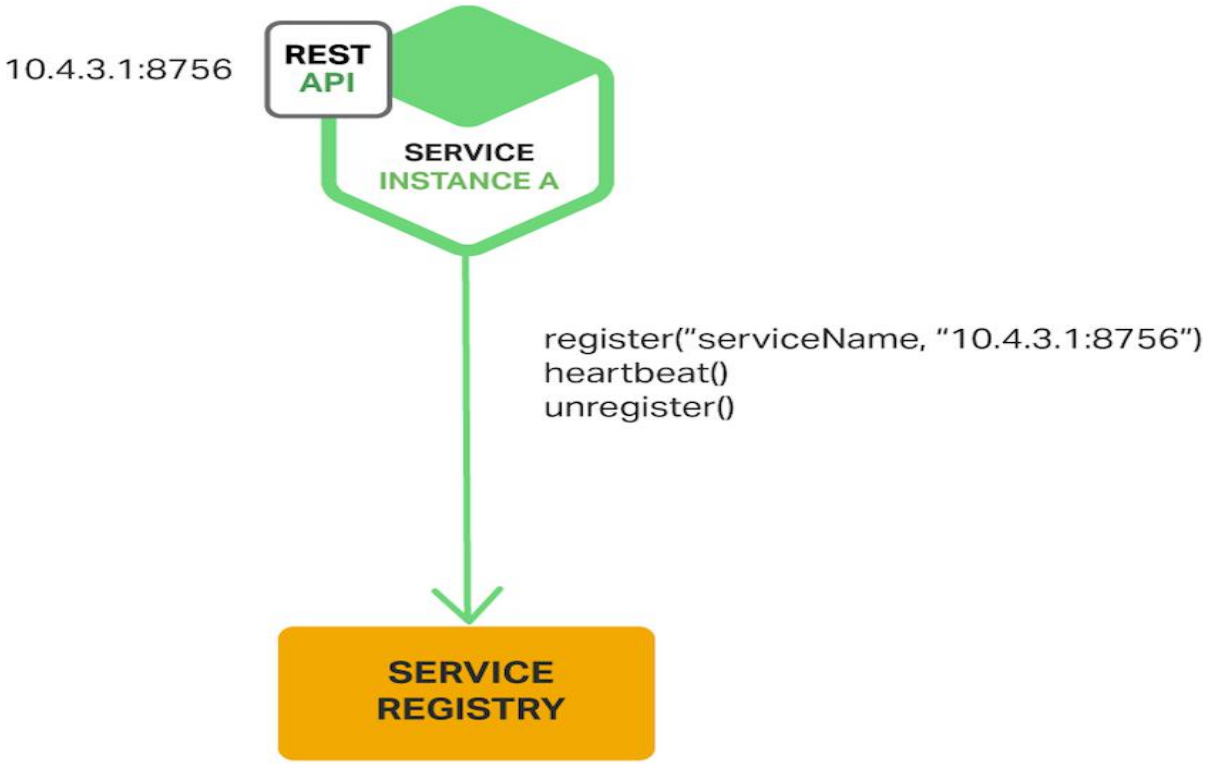
- 服务注册表是服务发现的关键部分
- 它是一个包含服务实例的网络位置的数据库
- 服务注册表需要高度可用并且是最新的

服务注册

- Eureka -> 推荐
- Zookeeper
- Etcd
- Consul

服务注册方式

自注册模式


10.4.3.1:8756
REST
API
SERVICE
INSTANCE A
register("serviceName, "10.4.3.1:8756")
heartbeat()
unregister()
SERVICE
REGISTRY

自注册模式

- 这种方法的一个很好的例子是Netflix OSS Eureka客户端。
- Eureka客户端处理服务实例注册和注销的所有方面。
- Spring Cloud项目实现了各种模式，包括服务发现，使得可以轻松地使用Eureka自动注册服务实例。 您只需使用@EnableEurekaClient注释注释您的Java配置类。

自注册模式

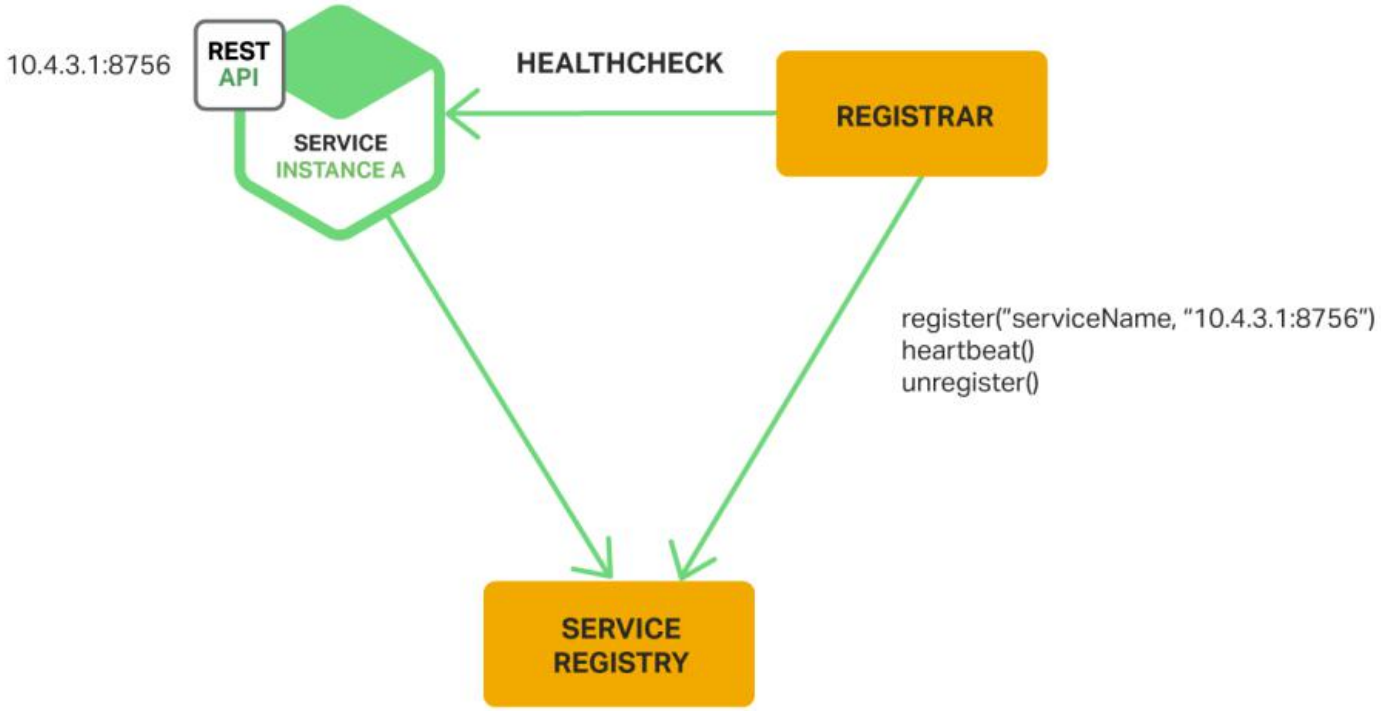
- 优点

  相对简单，并且不需要任何其他系统组件。

- 缺点

  它将服务实例耦合到服务注册表。 您必须在您的服务使用的每种编程语言和框架中实现注册码。

第三方注册模式


10.4.3.1:8756
REST
API
SERVICE
INSTANCE A
HEALTHCHECK
REGISTRAR
register("serviceName, "10.4.3.1:8756")
heartbeat()
unregister()
SERVICE
REGISTRY

第三方注册模式

- 开源注册器项目—Registrator 它会自动注册和注销部署为Docker容器的服务实例。 注册器支持多个服务注册表，包括etcd和Consul。

- NetflixOSS Prana 主要用于以非JVM语言编写的服务，它是与服务实例并行运行的边路应用程序。Prana 使用Netflix Eureka注册和注销服务实例。

第三方注册模式

优点:

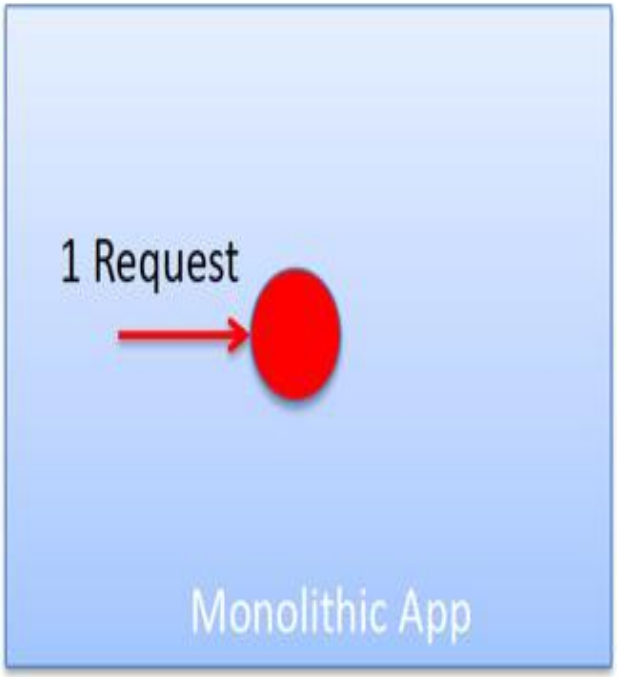
是服务与服务注册表断开连接。 不需要为开发人员使用的每种编程语言和框架实现服务注册逻辑。 相反，在专用服务内以集中方式处理服务实例注册。

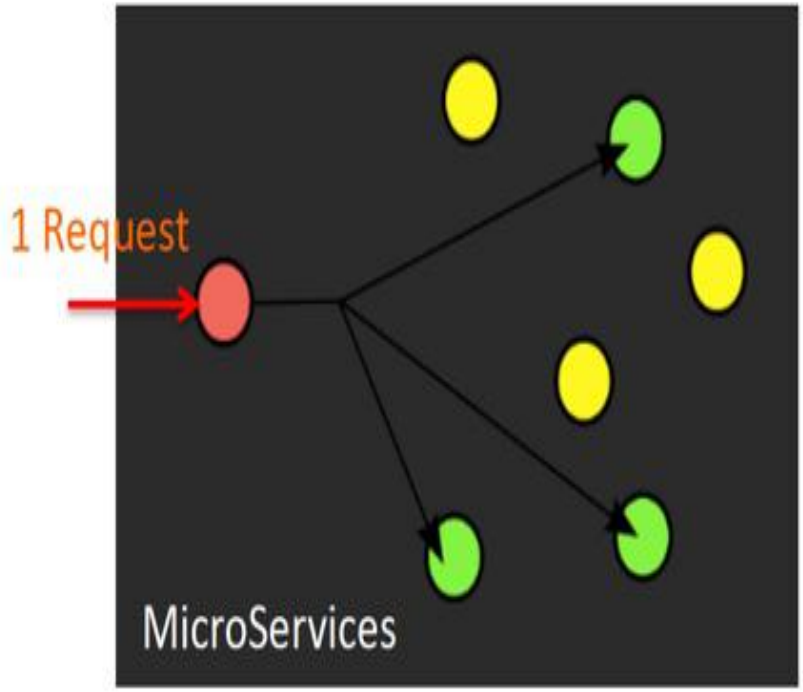
缺点

除非它内置在部署环境中，它是另一个高可用性的系统组件，需要设置和管理。

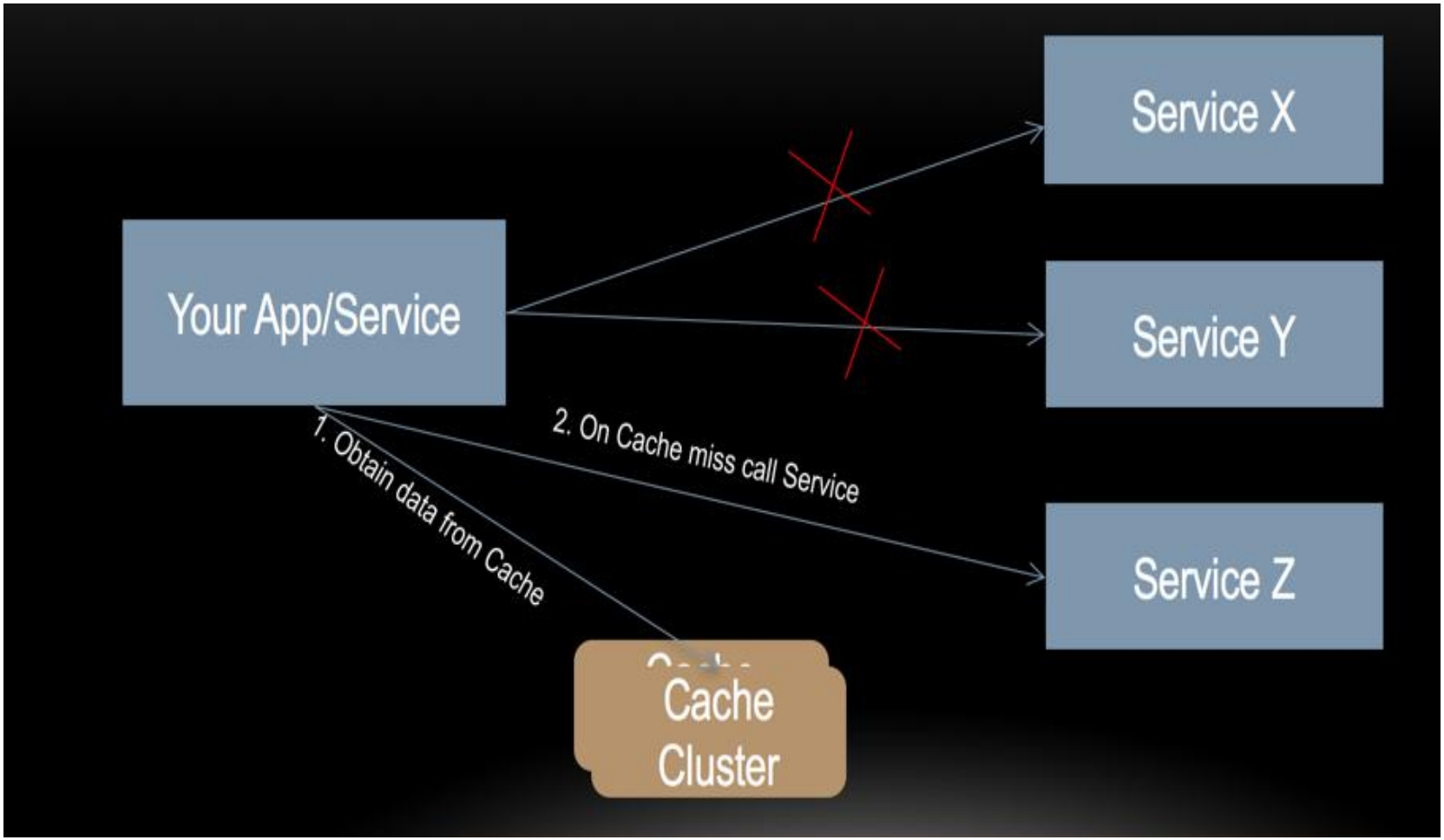
如何处理 FAN OUT

Fan out

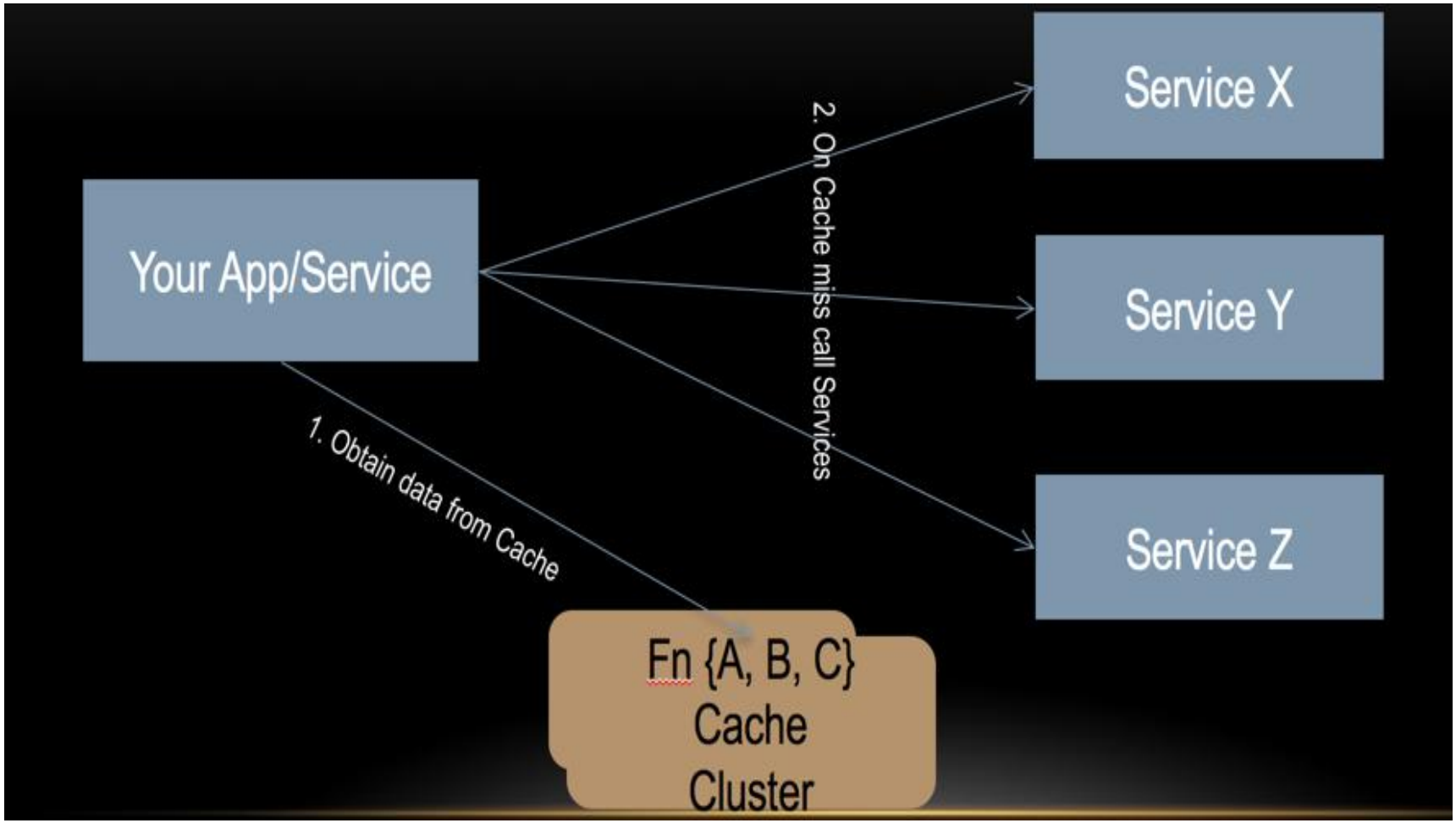




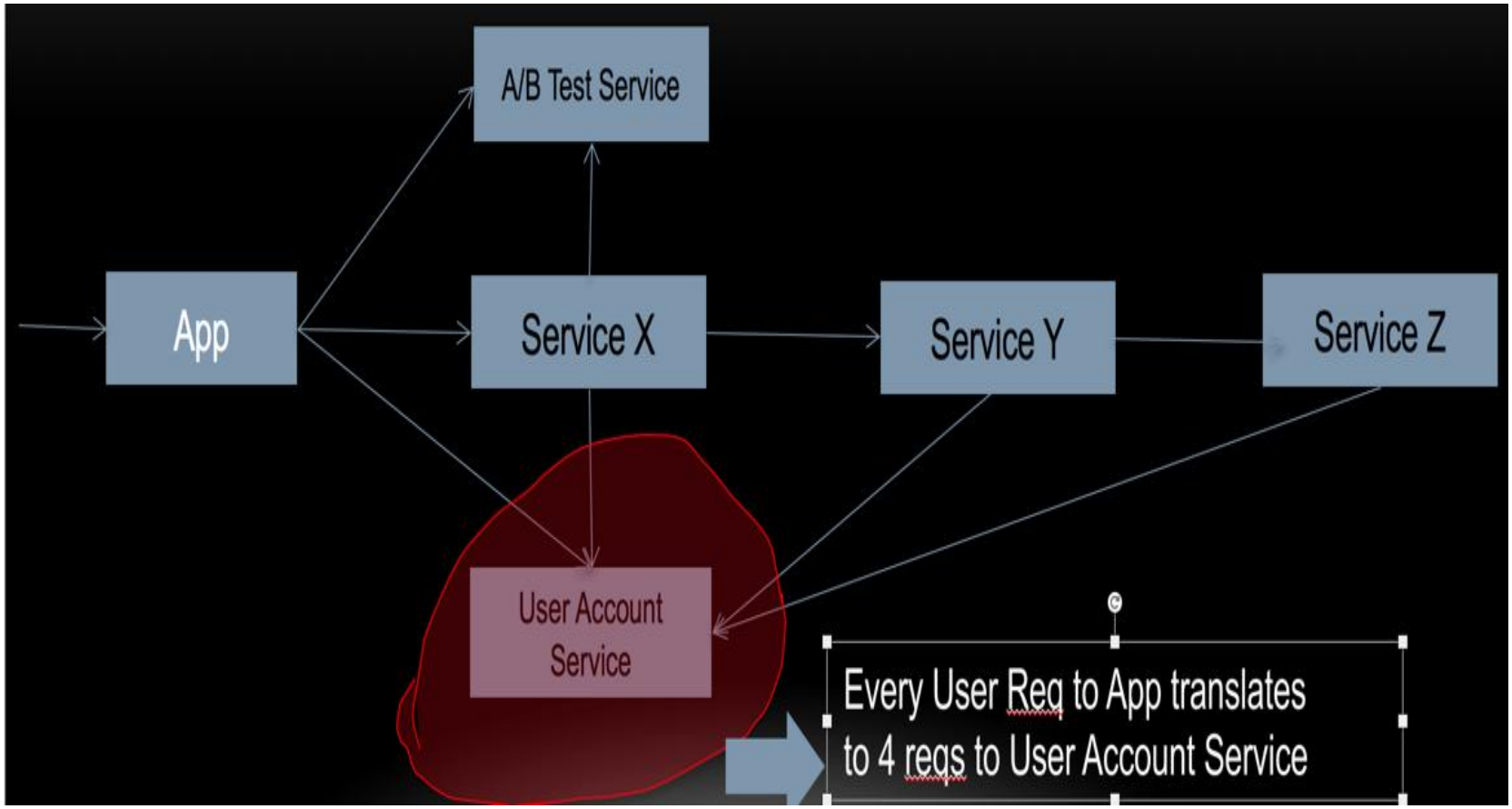
服务器缓存


Service X
Your App/Service
Service Y
2. On Cache miss call Service
1. Obtain data from Cache
Service Z
Cache
Cluster

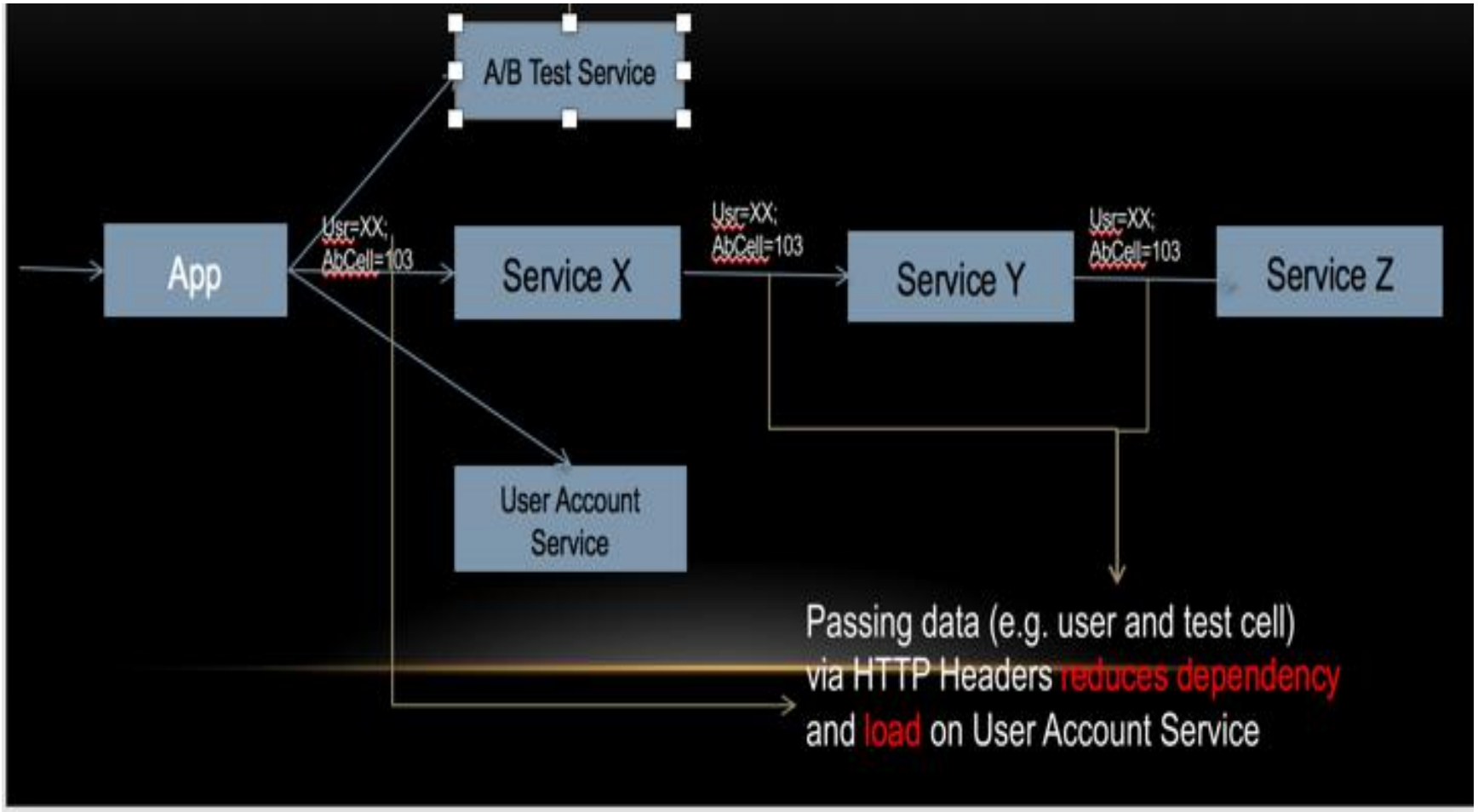
复合缓存


Service X
Service Y
Service Z
Your App/Service
2. On Cache miss call Services
1. Obtain data from Cache
Fn {A, B, C}
Cache
Cluster

瓶颈/热点


A/B Test Service
App
Service X
Service Y
Service Z
User Account Service
Every User Req to App translates to 4 reqs to User Account Service

通过HTTP HEADER传递数据



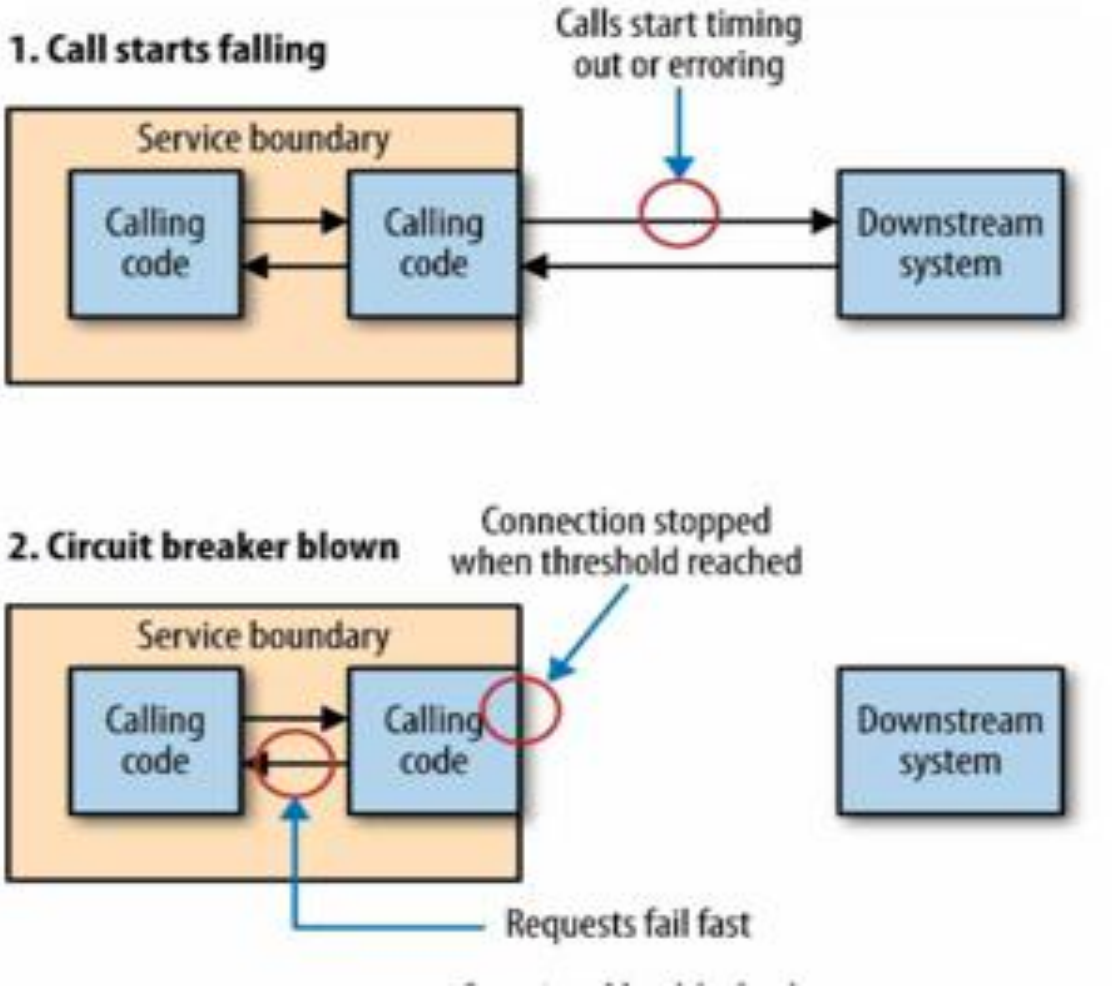
如何提高可用性（Availability）？

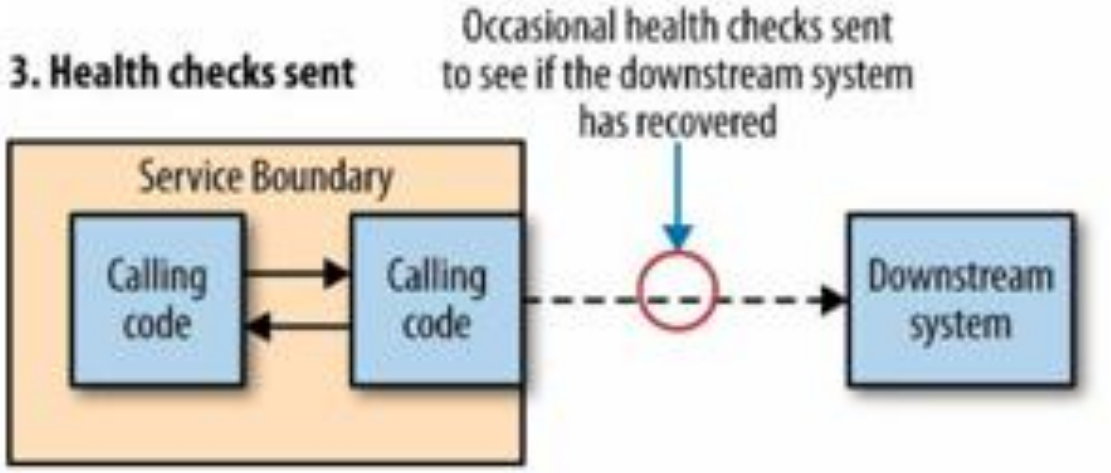
微服务不会自动意味着更好的可用性

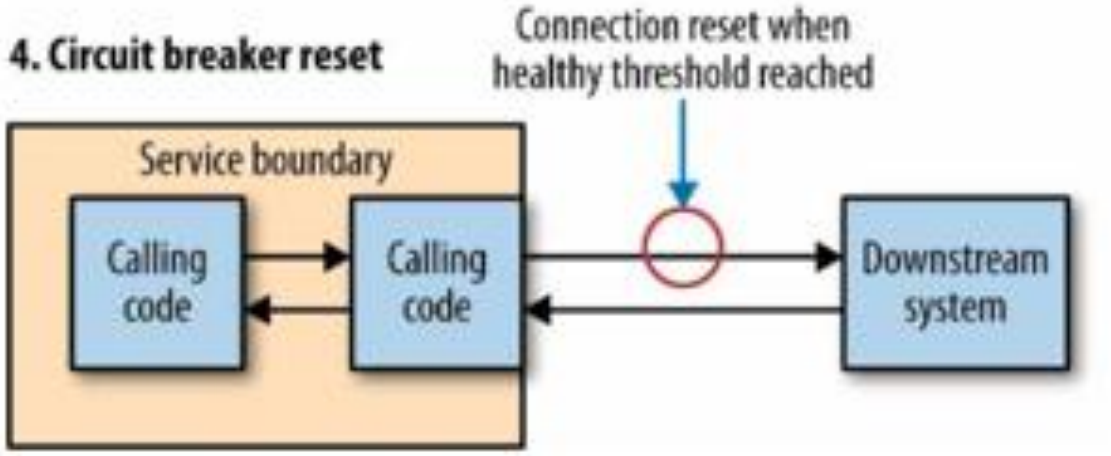
- 除非采用合理的容错架构

- Time out
- Circuit breaker(熔断器）
- Bulkheads (舱壁）- Reject new request

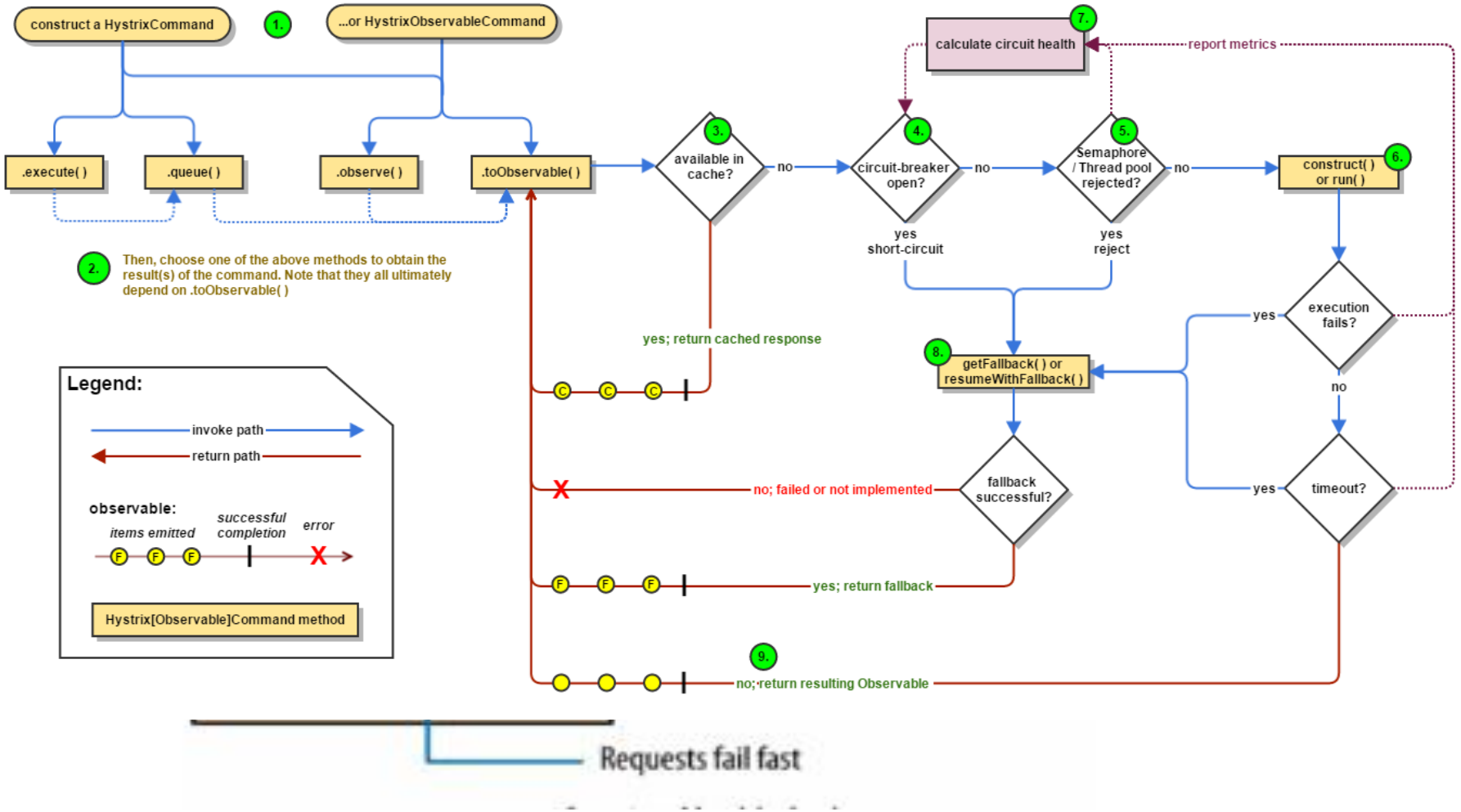
熔断器


1. Call starts falling
Calls start timing out or erroring
Service boundary
Calling code
Calling code
Downstream system
2. Circuit breaker blown
Connection stopped when threshold reached
Service boundary
Calling code
Calling code
Downstream system
Requests fail fast


3. Health checks sent
Occasional health checks sent to see if the downstream system has recovered
Service Boundary
Calling code
Calling code
Downstream system


4. Circuit breaker reset
Connection reset when healthy threshold reached
Service boundary
Calling code
Calling code
Downstream system

Hystrix


construct a HystrixCommand
1.
...or HystrixObservableCommand
.execute( )
.queue( )
.observe( )
.toObservable( )
2.
Then, choose one of the above methods to obtain the result(s) of the command. Note that they all ultimately depend on .toObservable( )
3.
available in cache?
no
4.
circuit-breaker open?
no
5.
Semaphore / Thread pool rejected?
no
6.
construct( ) or run( )
7.
calculate circuit health
report metrics
yes
short-circuit
yes
reject
yes; return cached response
8.
getFallback( ) or resumeWithFallback( )
execution fails?
yes
no
timeout?
yes
fallback successful?
no; failed or not implemented
yes; return fallback
9.
no; return resulting Observable
Legend:
invoke path
return path
observable:
items emitted
successful completion
error
Hystrix[Observable]Command method
Requests fail fast

微服务测试

- 单元测试（Unit Test）
  Yes
- 服务测试（Service Test）
  Yes
- 端对端测试（End to End Test)
  尽量避免
- 延迟和服务可靠性测试
  Chaos Monkey

微服务带来的企业组织结构挑战

Conway's law

- Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication.

- 企业的组织构架往往会反映在技术构架中，微服务在企业内部是否能够成功很大程度上取决于企业的组织构架和技术构架是否能够匹配。

- 以Netflix 为例子来讲讲在向微服务构架转变的过程中对团队和企业构架带来的挑战

优化速度，而不是效率

- 速度是赢得市场最重要的因素
- 速度意味着了解客户需求，并以比竞争对手更快的速度给予他们想要的东西。 在竞争对手准备跟进的时候，您已经转到下一组改进。
- 如果您问任何开发人员是否较慢的开发过程更好，没有人会说是的。 管理层或客户也不会抱怨说您的开发周期对他们来说太快了

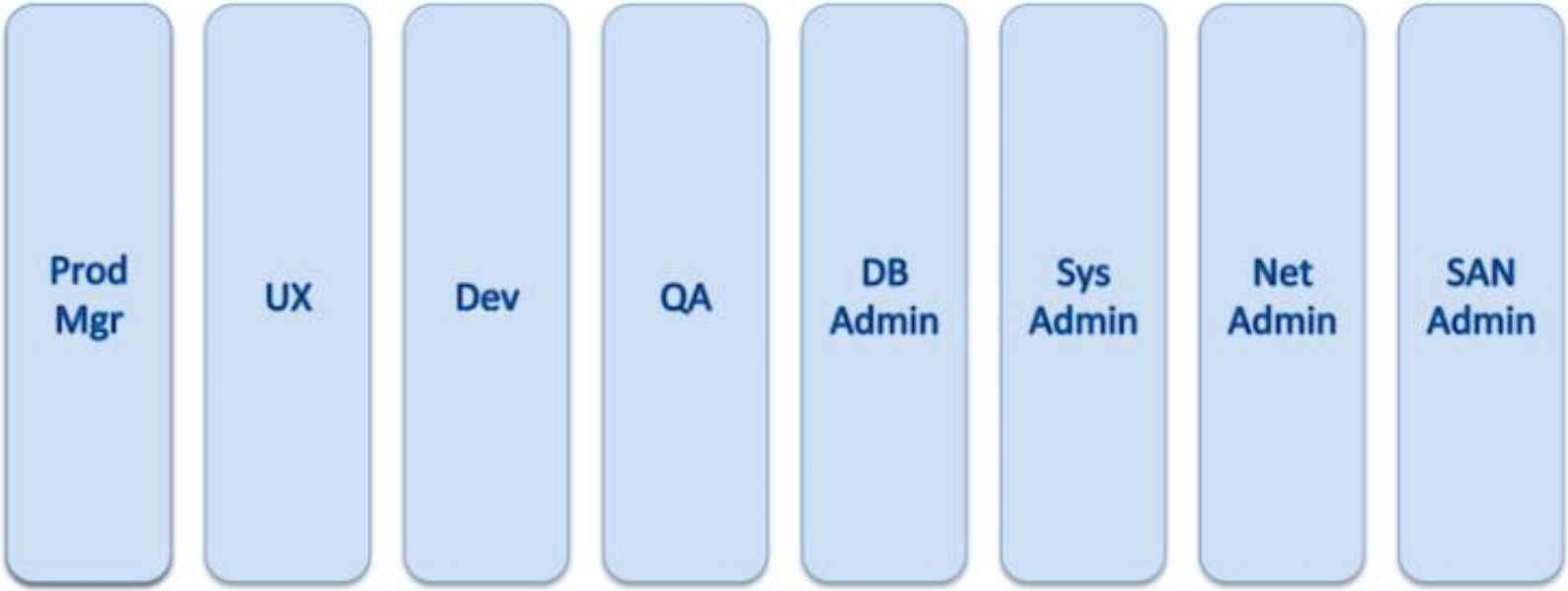
优化速度，而不是效率

- 很多公司强调效率
- 强调效率通常意味着试图控制开发过程的整体流程，以消除重复的工作并避免错误，同时注意降低成本。 常见的结果是，注重节流，而不是开源
- 如果你说“我这样做是因为它更有效率”，那么这个意想不到的结果就是让你慢下来。 这不是鼓励浪费和重复开发，但是应该先优化速度。 效率成为次要的。
- 提高效率不是一个企业的终极目标，提高效率要以业务增长更快为结果

以结果为导向，减少不必要流程

- 尽量增加每个微服务团队的自由度：开发工具，开发流程等。
- 尽量减少流程，过多的流程会减慢对新生事物和突发情况的反映速度 =》 流程都是对过去的总结
- 明确每个团队的目标，减少互相依赖关系

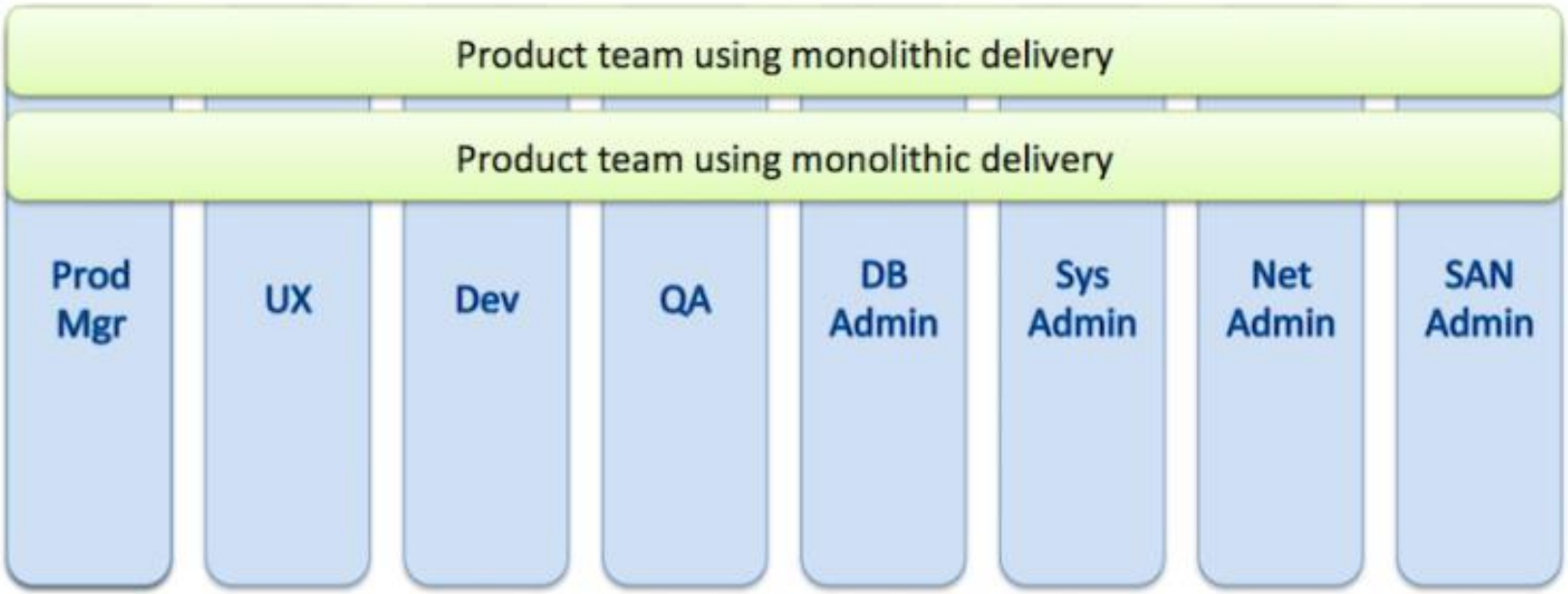
传统公司产品开发流程



传统公司产品研发团队

- 大多数软件开发团队呈孤岛状，他们之间没有人员重叠
- 软件开发项目的标准过程从与用户体验和开发组的产品经理会议开始，讨论新功能的想法。 在代码中实现该思想之后，代码被传递给质量保证（QA）和数据库管理团队，经常需要开很多会议
- 与系统，网络和SAN管理员的沟通往往是通过内部的TICKET 系统。 整个过程往往是缓慢的。

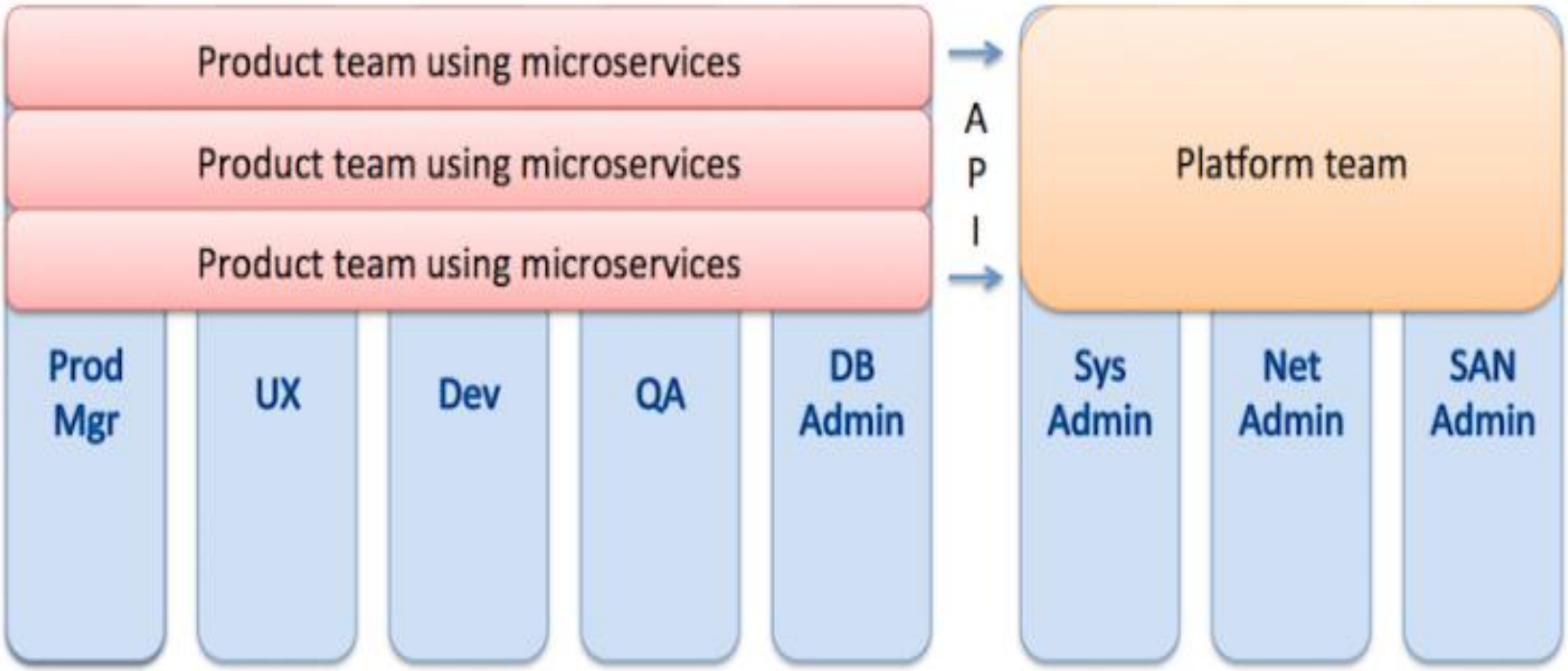
创业公司形式的产品研发团队



初创公司形态的产品研发团队

- 有些公司试图用初创公司的形式开发产品。
- 初创公司开发团队 == 微服务开发团队？ NO
- 公司会有很多个小的初创公司形式的小团队，但是每个团队内部结构和传统公司的团队结构一样

微服务产品研发研发团队



微服务产品研发研发团队

- 不再有不同的产品经理，UX经理，开发经理等，在其孤岛中向下管理
- 每个产品功能（实现为微服务）都有一名经理，他负责监督一个团队，从构思到部署来处理微服务的软件开发各个方面
- 平台团队提供产品团队通过API访问的基础架构支持
- 采用DEVOP形式发布和维护产品

微服务应用场景

- 流量巨大
- 系统复杂
- 需要快速开发出新的产品
- 用户数量增长迅速
- 绝大多数TO C的互联网公司

单片服务应用场景

- 流量较小
- 单一或非常简单的功能
- 没有快速迭代的需求
- 企业内部工具，POC工具或网站


Q & A


WOTA
51CTO
Thank you !