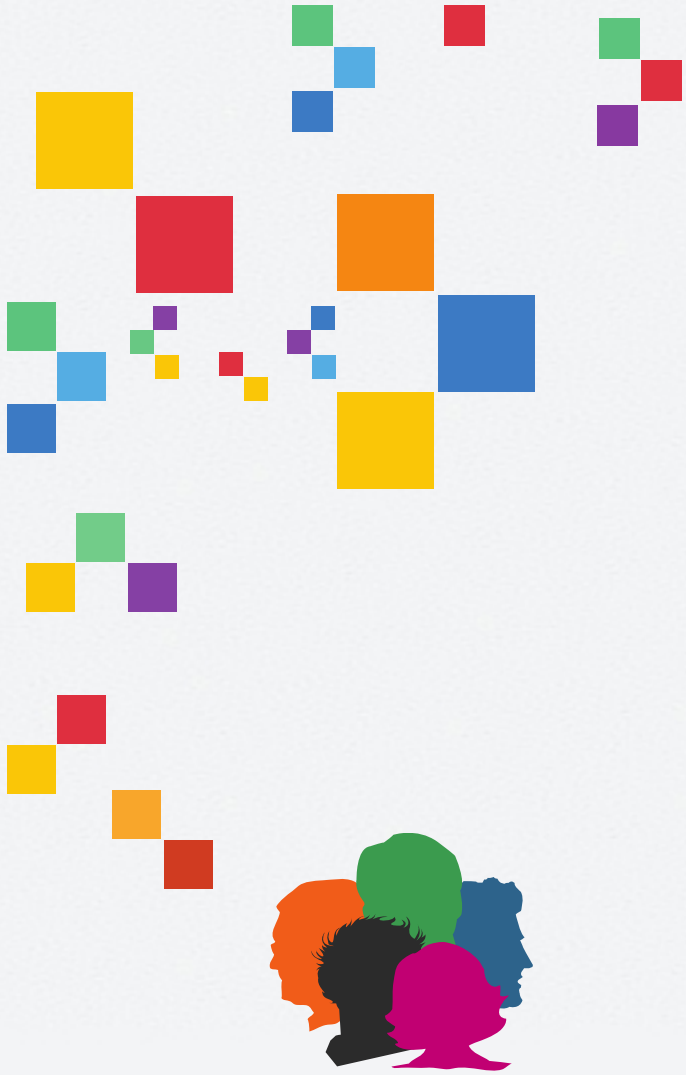
华为全云化战略的背后
—Cloud Native

王启军



讲师介绍

华为公司架构部架构师，负责华为公司的云化、微服务架构推进落地，前后参与了华为手机祥云4.0、物联网IoT2.0的架构设计。曾任当当网架构师，主导电商平台架构设计，包括订单、支付、价格、库存、物流等。曾就职于搜狐负责手机微博的研发。目前运营微信公众号“奔跑中的蜗牛”，热爱开源，热爱分享。

目录

- 背景
- 基本概念
- 原则
- 扩展性
- 服务划分
- 性能
- 一致性

背景

构建全云化网络

随着5G、视频、IoT等新技术的发展，未来网络和业务的复杂性呈现指数级增长。为了提升业务弹性、健壮性、敏捷性，创新商业模式，降低运维成本，网络全面云化成为必

推进“全面云化”战略，使能行业数字化转型

在2016年华为全球分析师大会上，华为轮值CEO徐直军向与会嘉宾解读了“全面云化”战略。

一是硬件资源池化，包括网络和IT设备，从而实现资源的最大共享，改变传统的一个应用一个硬件的“烟囱”架构。

二是软件架构的全分布化，这也是吸纳互联网公司的技术架构。全分布化是实现“大规模系统”的基本条件，分布式系统才能具备弹性能力，才能够实现故障的灵活处理和资源的调度，才能根据业务量大小，基于策略实现弹性伸缩和故障修复。

三是全自动化，无论是业务部署、资源调度，还是故障修复等，都是自动完成的，不需要人工干预。

Source：http://www.huawei.com/

Cloud native的起源

Source:https://wso2.com/about/leadership/paul_fremantle

2010年5月28日写的一篇blog中首次提出。他认为Cloud Native包含的属性包括：

- 分布式；
- 弹性；
- 多租户；
- 自服务；
- 按需计量和计费；
- 增量部署和测试。

Cloud native的起源

Adrian Cockcroft
VP Cloud Architecture Strategy at AWS

Source：http://searchcloudcomputing.techtarget.com

2013年，Netflix的云架构师（2016年10月成为AWS的VP）Adrian Cockcroft在Yow Conference上介绍了Netflix在AWS上基于Cloud Native的成功应用。Netflix在AWS上运行着上万个实例，每天都有数以千计的实例被创建或删除。

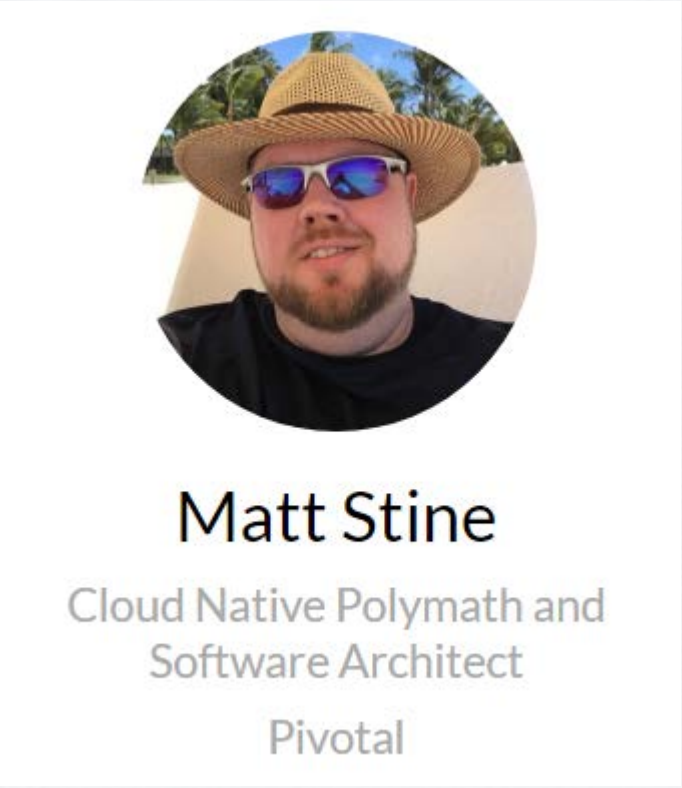
架构原则是：

- 不变性；
- 关注点分离；
- 反脆弱性；
- 高信任的组织；
- 分享。

采取的主要措施是：

- 利用AWS实现可扩展性、敏捷和共享；
- 利用微服务实现关注点分离；
- 利用非标准化数据实现关注点分离；
- 利用猴子工程师实现反脆弱性；
- 利用默认开源实现敏捷、共享；
- 利用持续部署实现敏捷、不变性；
- 利用DevOps实现高信任组织和共享；
- 利用运行自己写的代码实现反脆弱性开发演进。

Cloud native的起源

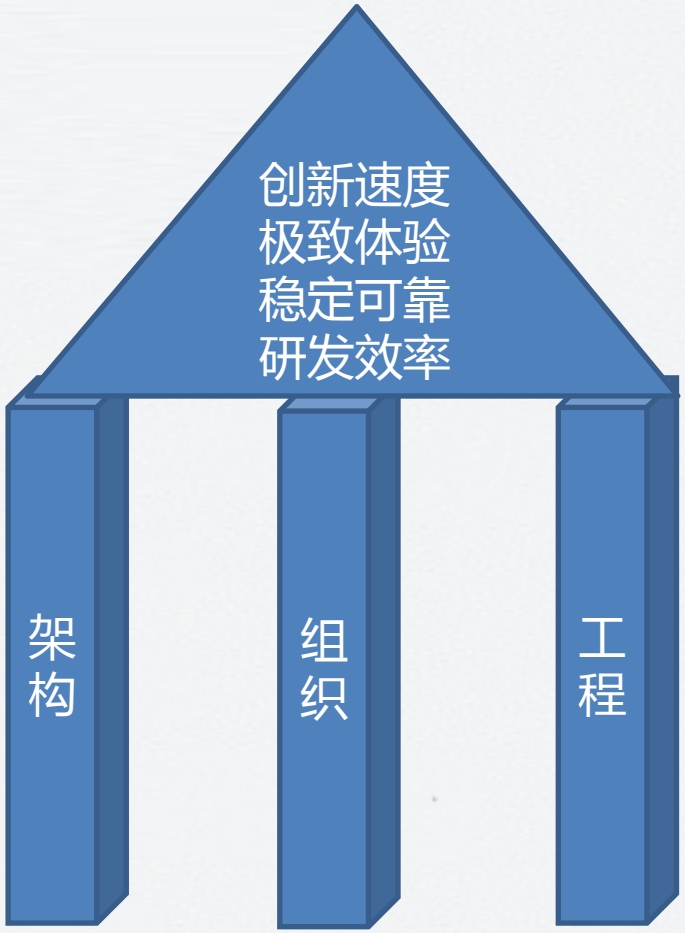


Source:http://www.mattstine.com/

《Migrating to Cloud-Native Application Architectures》一书认为单体服务向Cloud Native迁移的过程中，需要文化、组织、技术共同变革。该书把Cloud Native描述为一组最佳实践。包含如下几个重要内容：

- 十二因子；
- 微服务；
- 自服务敏捷基础设施；
- 基于API的协作；
- 反脆弱性。

Cloud Native的组成



Cloud Native—工程

部署频率 – 部署到生产的频率
改变的交付时间 – 如何快速地将新的变化推向生产
平均恢复时间（MTTR） – 从故障中恢复的平均时间（中断）

Table 1: Changes in IT performance of high performers, 2016 to 2017

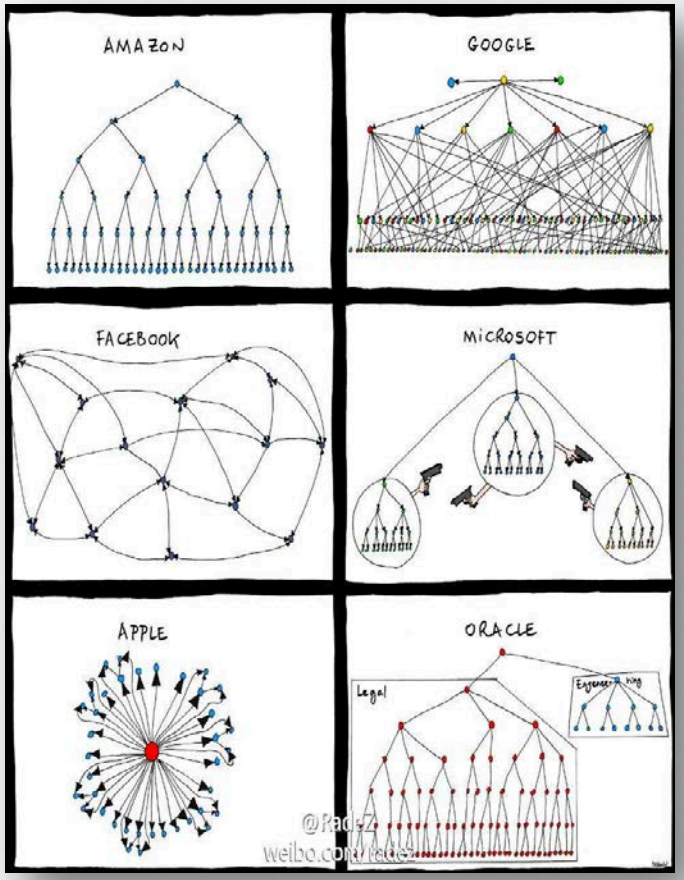
IT performance metrics	2016	2017
Deployment frequency	200x more frequent	46x more frequent
Lead time for changes	2,555x faster	440x faster
Mean time to recover (MTTR)	24x faster	96x faster
Change failure rate	3x lower (1/3 as likely)	5x lower (1/5 as likely)

Source： 2017 State of DevOps Report

Cloud Native—组织

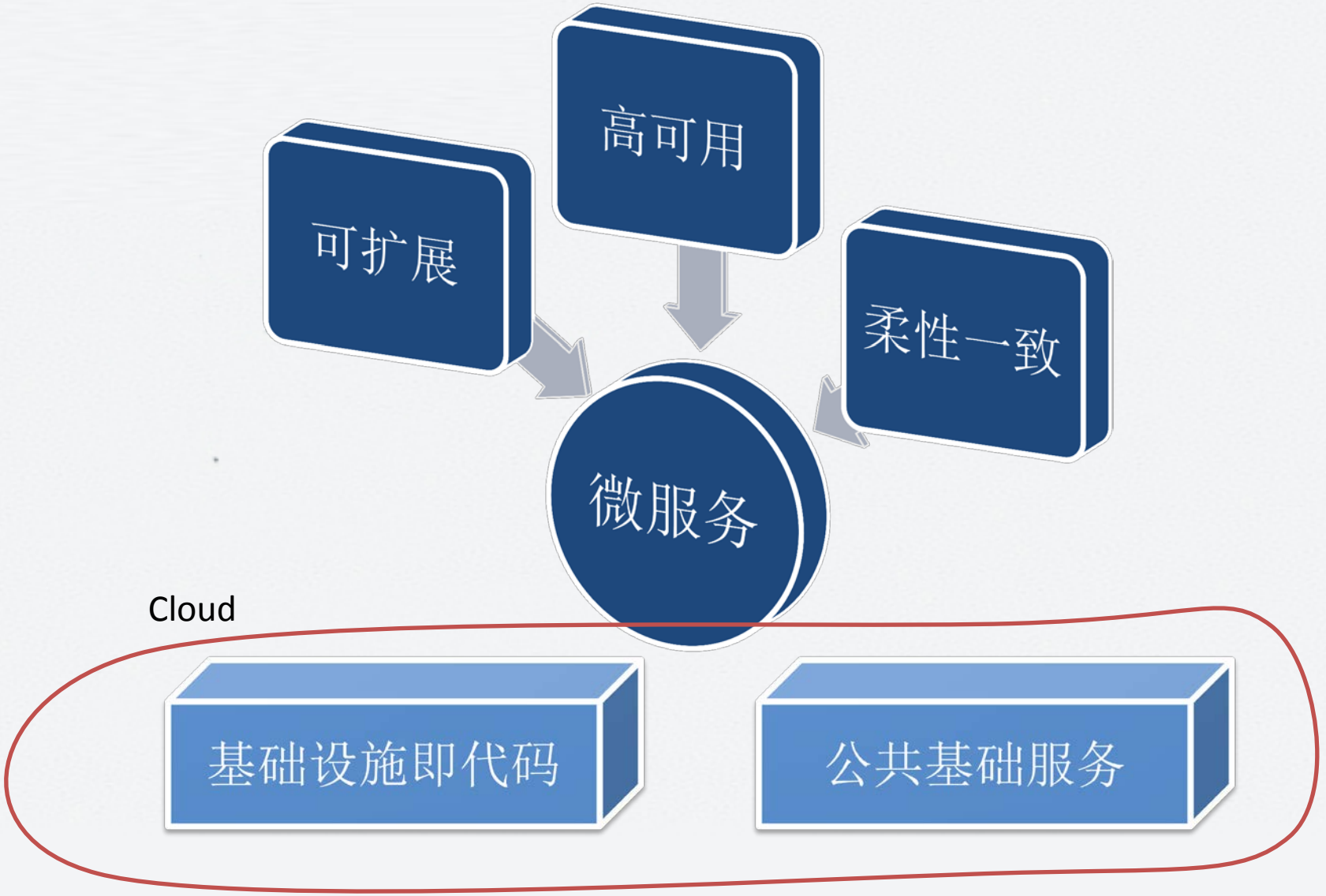
组织可以把Cloud Native推向新的高度

- ✓ 是否需要团队以外批准才能对系统架构大规模变更？
- ✓ 是否需要团队以外批准才能上线新版本？
- ✓ 是否需要在业务低峰才能完成部署？
- ✓ 团队成员是否端到端负责？



http://news.mydrivers.com/

Cloud Native—架构



原则

- Design for failure
- 不变性
- 独立自主
- 标准化
- 速度优先
- 简化设计
- 自动化驱动
- 持续演进

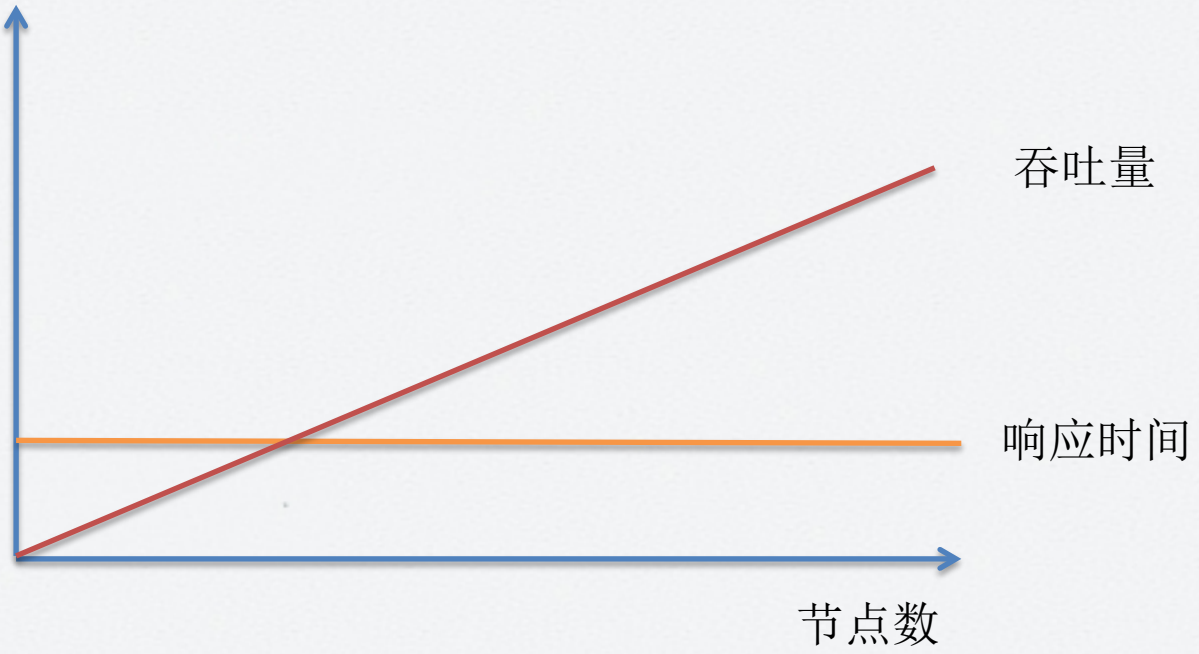
Make Professional Discovery
MPD

扩展性问题

Make Professional Discovery
MPD

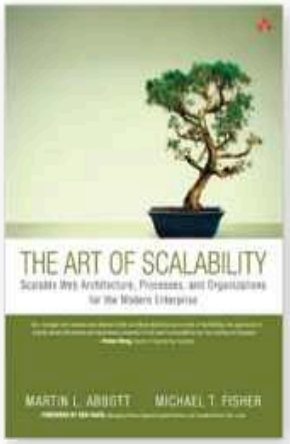
加机器能解决问题吗

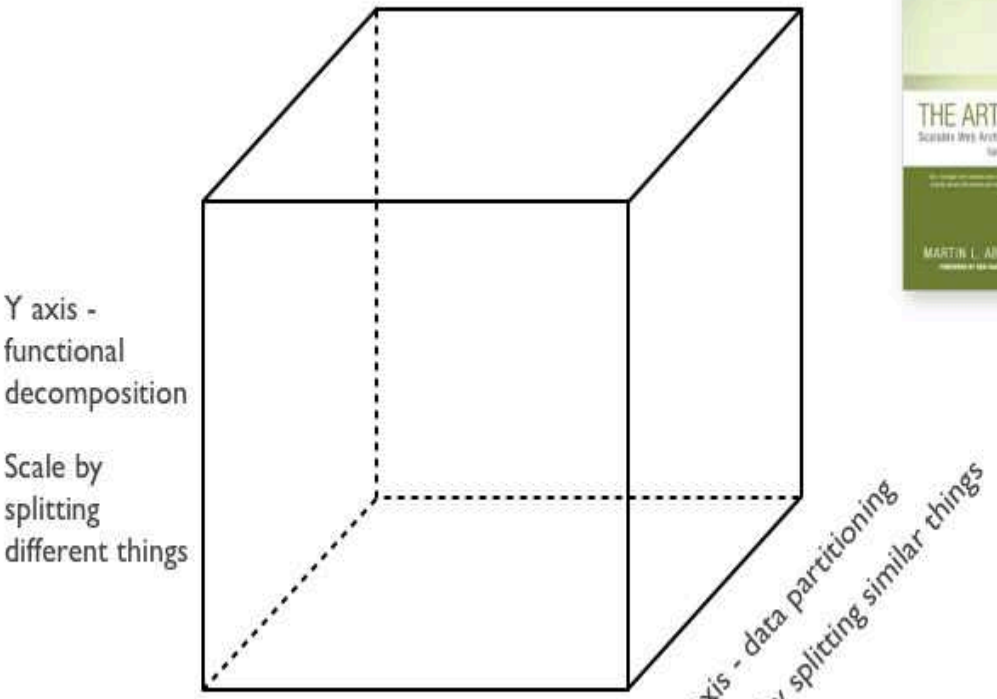
横向扩展扩展什么？



Make Professional Discovery
MPD

3 dimensions to scaling





Source：http://microservices.io/articles/scalecube.html

扩展顺序

- 第一阶段
  - 10人左右
  - 需求不确定
  - 单体
  - 负载均衡
  - 读写分离

扩展顺序

- 第二阶段
  - 用户快速增长
  - 需求快速增加
  - 团队近百人
  - 沟通效率低
  - 主从延迟
  - 微服务
  - 分库
  - 垂直分表

扩展顺序

- 第三阶段
  - 数据库性能瓶颈
  - MQ
  - Cache
  - 水平分表
  - 数据库中间件
  - 异地多活

如何拆分服务？

代码行数？包大小？
团队规模？
越小越好？
为什么是10个，不是15个？

MonolithFirst

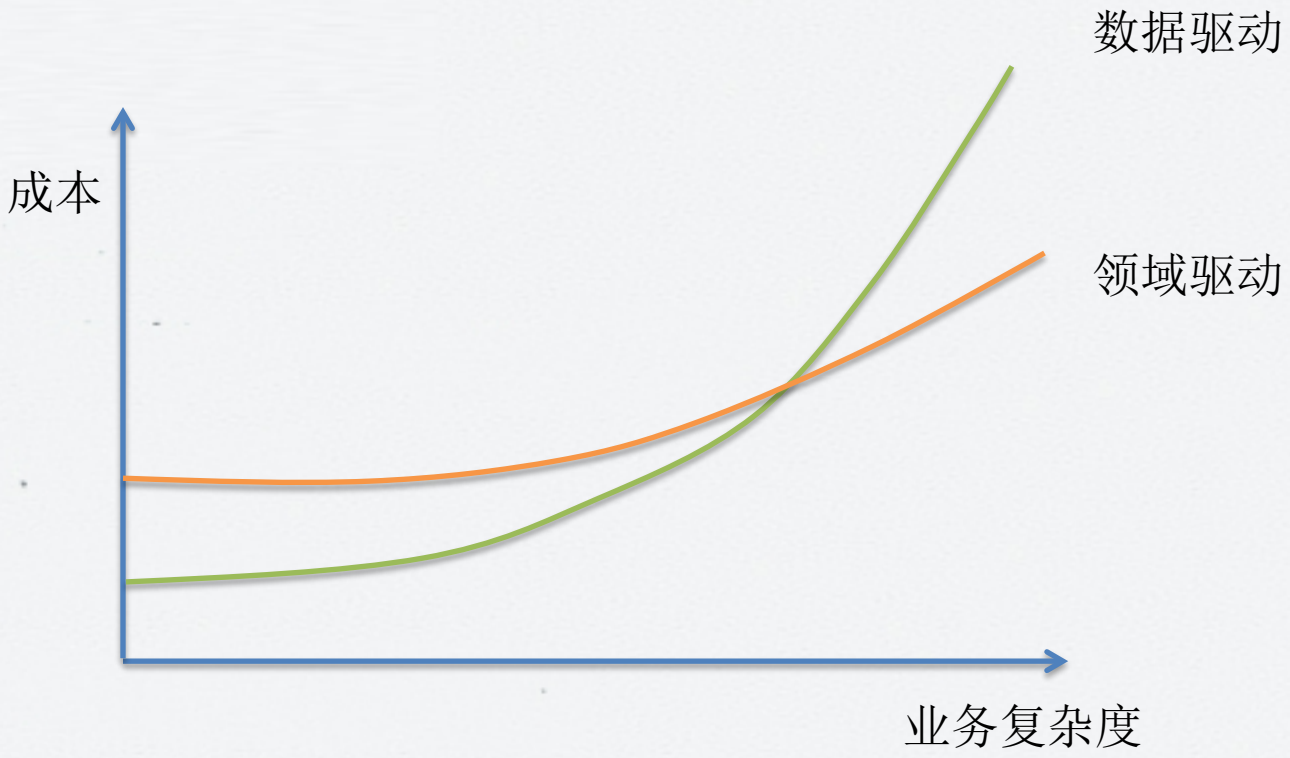
Martin Fowler

3 June 2015

As I hear stories about teams using a microservices architecture, I've noticed a common pattern.

1. Almost all the successful microservice stories have started with a monolith that got too big and was broken up
2. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

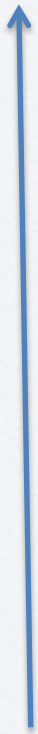
如何拆分服务？



附加条件：团队对领域驱动的熟悉程度

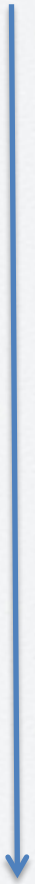
数据驱动

- 需求分析
- 抽象数据结构
- 划分服务
- 确定调用关系
- 业务流程验证
- 持续优化



领域驱动

- 统一语言
- 业务分析
- 寻找聚合
  - 事件风暴
  - 四色模型
- 确定调用关系
- 业务流程验证
- 持续优化



拆分策略

- 新业务优先
- 通用服务优先
- 核心服务优先
- 边界明显的服务优先
- 独立属性的服务优先

来，给你两天时间，帮我把服务拆分一下。

Source： https://baike.so.com

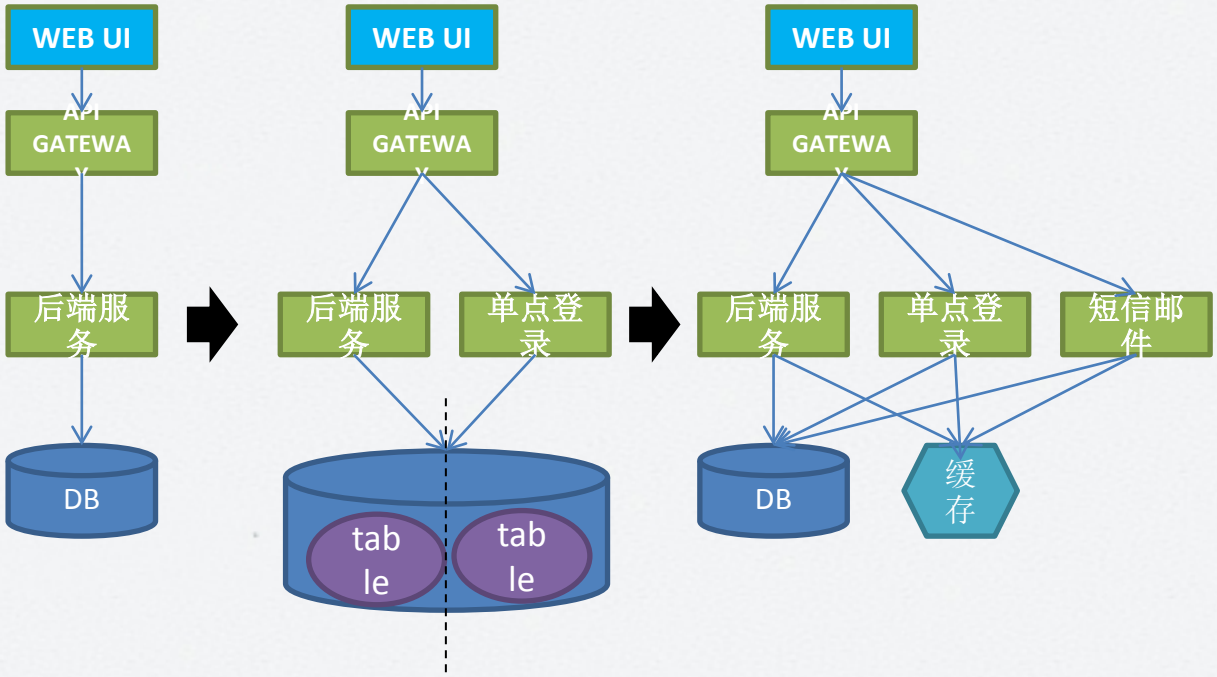
是不是关联性不强的数据就不应该放在一个服务里？

两个服务的关系数据要不要独立出一个服务？

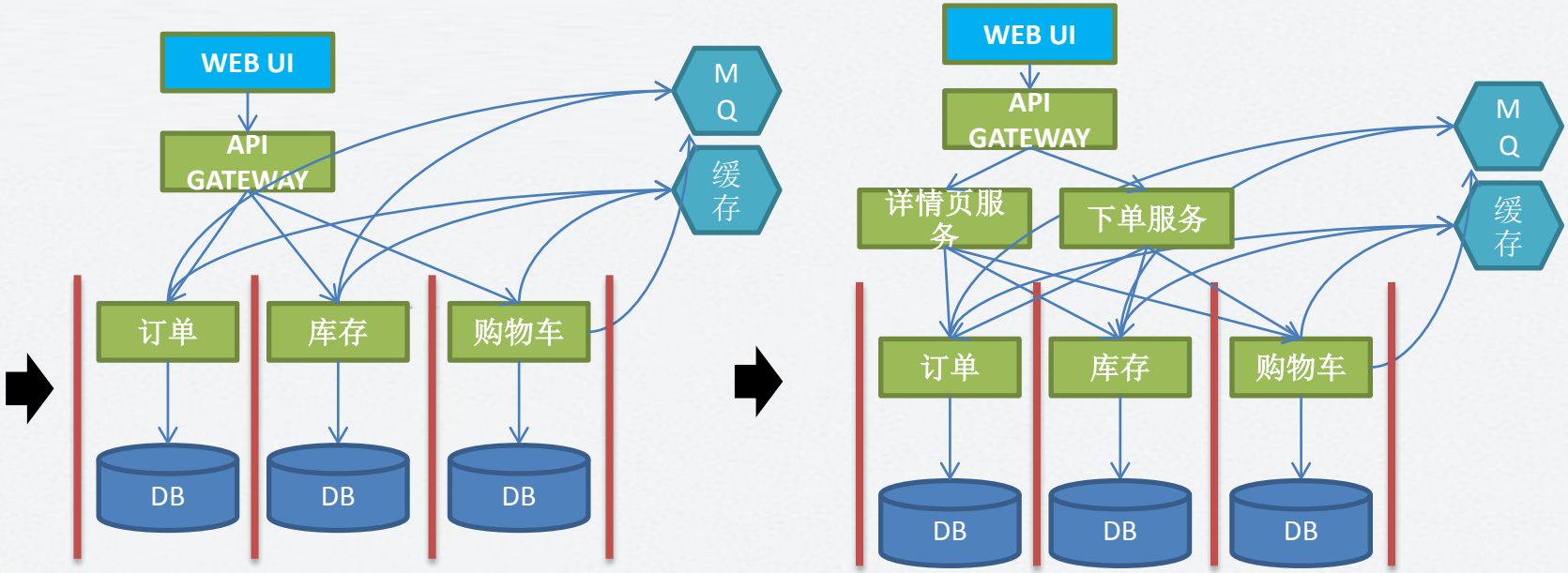
水平划分 VS 垂直划分
优雅 VS 实用

演变过程（一）

- 垂直划分优先；
- 应该尽量保证单向调用，避免两个服务互相调用。水平划分能有效的解决这个问题；
- 随着业务逐步变得复杂，团队、代码规模不断增长，水平划分不可避免；
- 水平层级应该尽量减少，因为单次请求跨越层级过多会导致整体响应时间和可用性的降低；



演变过程（二）

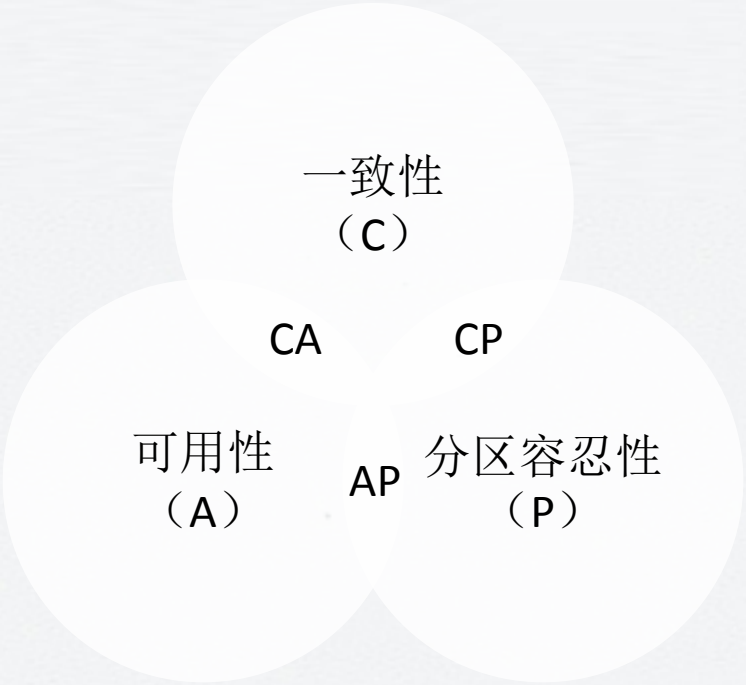

WEB UI
API GATEWAY
MQ
缓存
订单
库存
购物车
DB
DB
DB
WEB UI
API GATEWAY
详情页服务
下单服务
MQ
缓存
订单
库存
购物车
DB
DB
DB

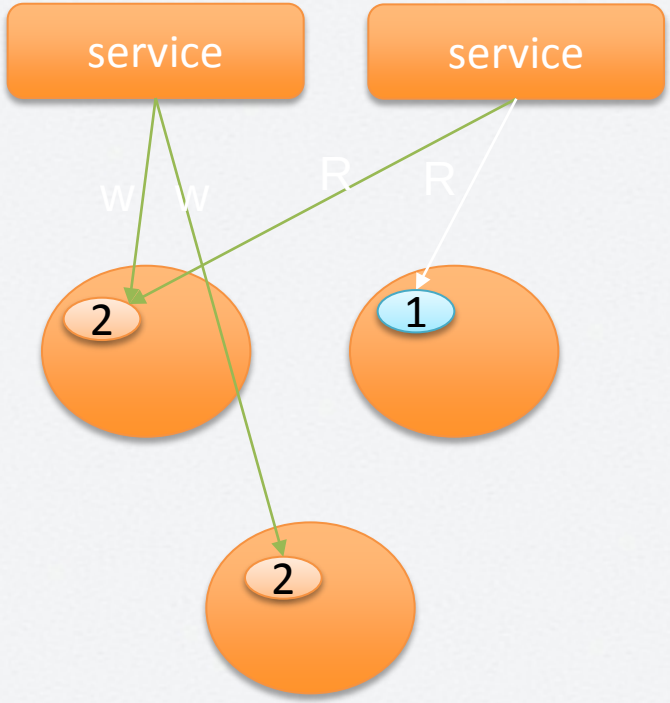
Make Professional Discovery
MPD

分布式之后.....

写多份数据，
怎么保证一致性？

理论

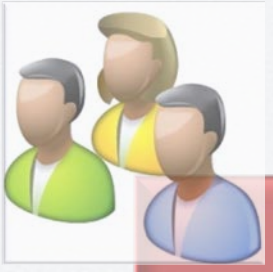




一致性分类

以数据为中心的一致性模型

- 严格一致性（Strict Consistency）
- 顺序一致性（Sequential Consistency）
- 因果一致性（Causal Consistency）
- FIFO一致性（FIFO Consistency）
- 弱一致性（Weak Consistency）
- 释放一致性（Release Consistency）
- 入口一致性（Entry Consistency）



以用户为中心的一致性模型

- 单调读一致性（Monotonic-read Consistency）
- 单调写一致性（Monotonic-write Consistency）
- 写后读一致性（Read-your-writes Consistency）
- 读后写一致性（Writes-follow-reads Consistency）

业界常用一致性分类

弱一致性 Weak
最终一致性 Eventually
强一致性 Strong

一致性是个全局问题

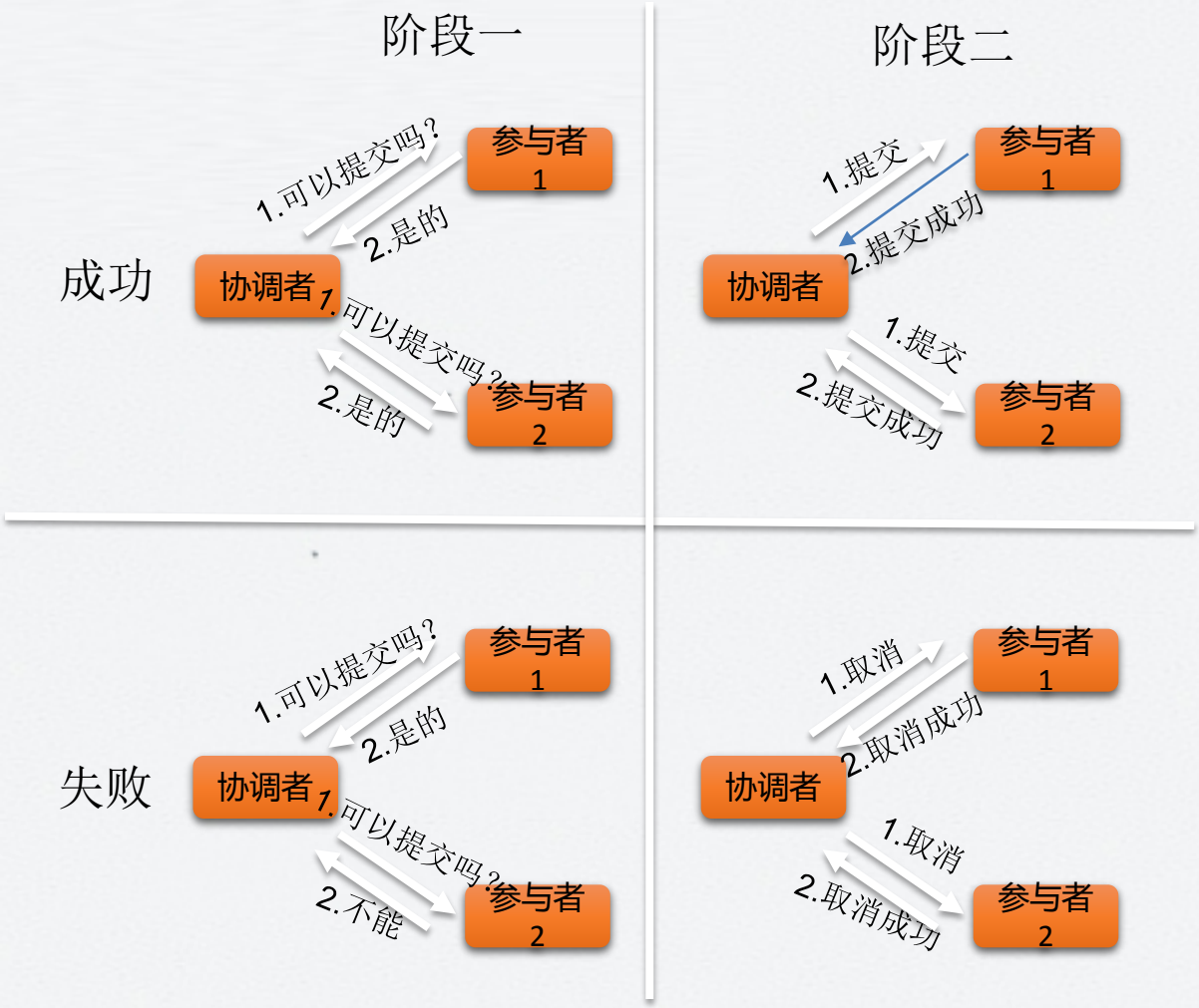
- Backups，通常不会用在生产环境；
- M/S = master/slave，可以异步，也可以同步，写是瓶颈点；
- MM - multi-master，可以解决写的问题，复杂度在于如何解决冲突；
- 2PC - 2 Phase Commit，强一致，性能低，容易死锁；
- Paxos，它是完全分布的。没有单一的主协调员。

Source：http://snarfed.org/transactions_across_datacenters_io.html

Make Professional Discovery
MPD

如何实现强一致

两阶段提交（2PC）

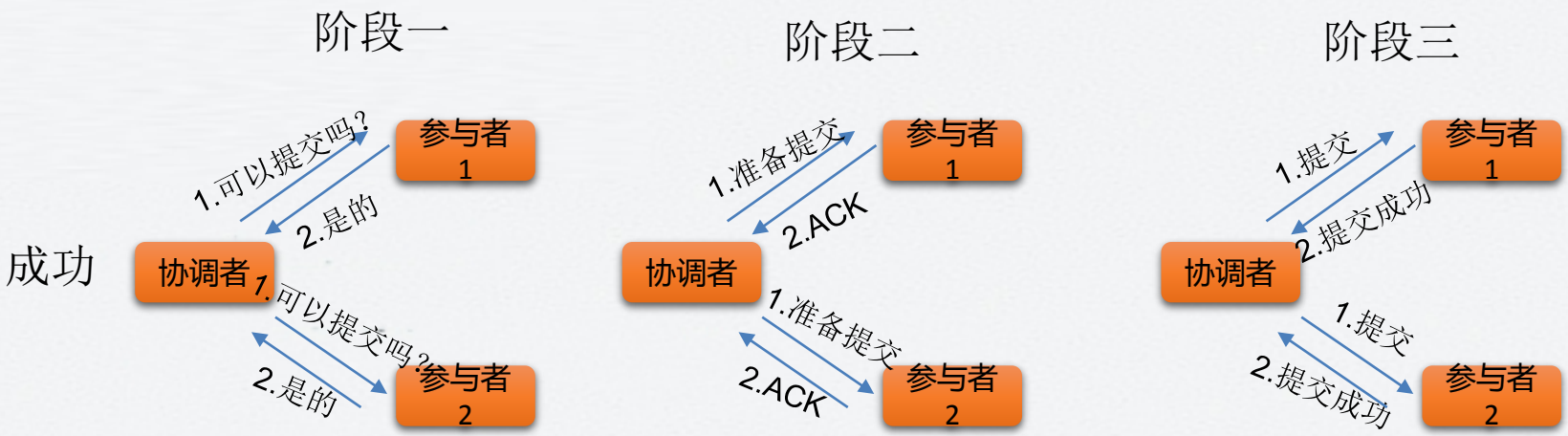


① 第一阶段锁定数据之后，如果协调者挂掉了，将没有角色去指挥参与者是否应该提交，这个数据会一直被锁定。也就是说如何保证协调者的可用性？

② 如果第二阶段参与者1提交成功，参与者2提交失败了（挂掉了，没返回ACK），此时协调者不知道该如何处理，因为参与者1已经提交成功，外部可以访问了。

③ 可扩展性极差。

三阶段提交（3PC）



实现的难度更大。

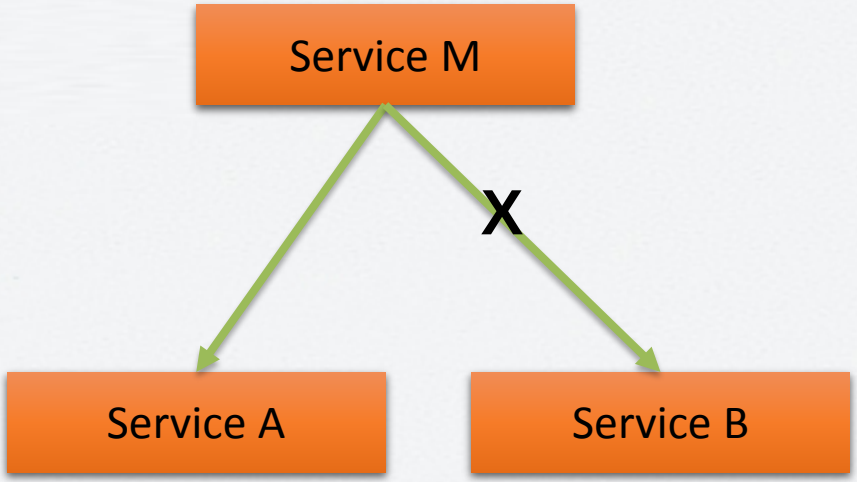
性能也更低。

可扩展性极差。

Make Professional Discovery
MPD

如何实现最终一致

调用失败怎么办？



重试！！！

超时时间。

重试的次数。

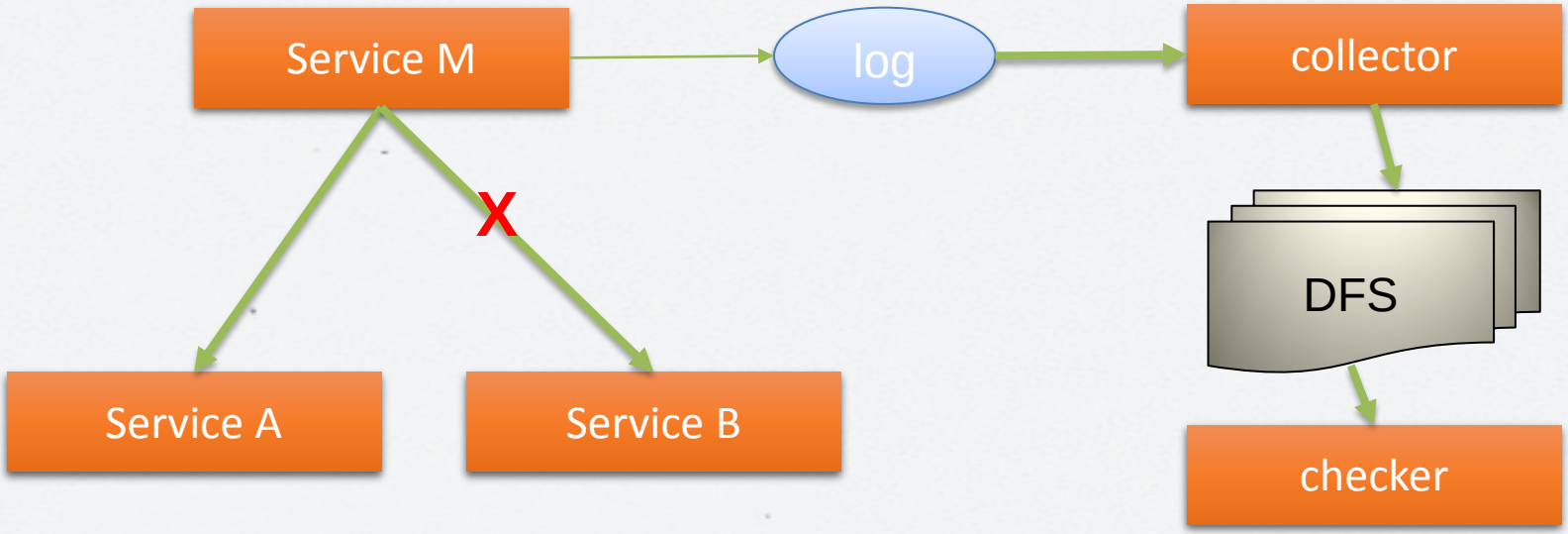
重试的间隔时间。

重试间隔时间的衰减度。

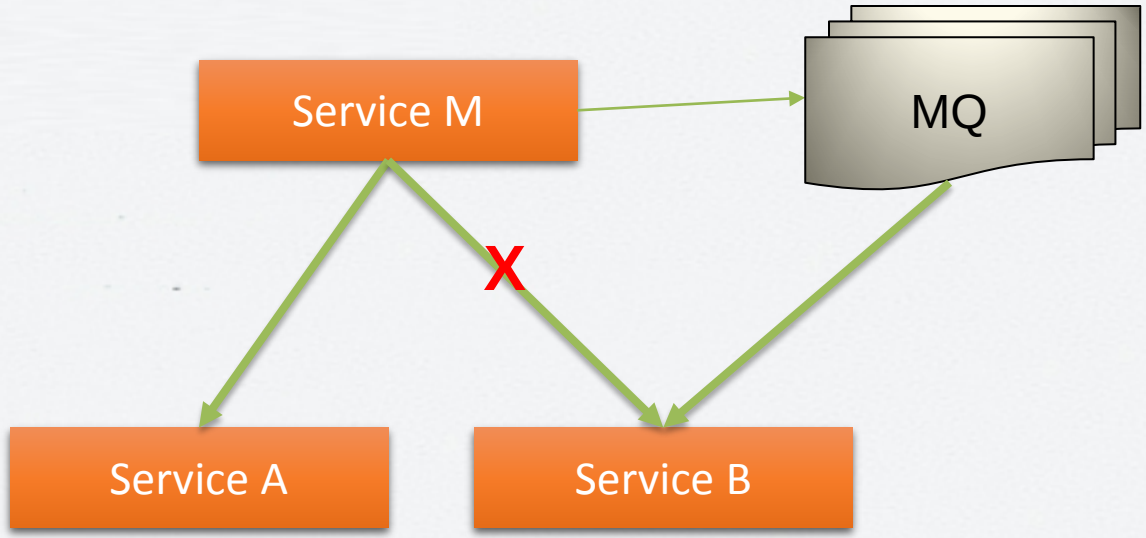
Make Professional Discovery
MPD

状态？？？状态！！！

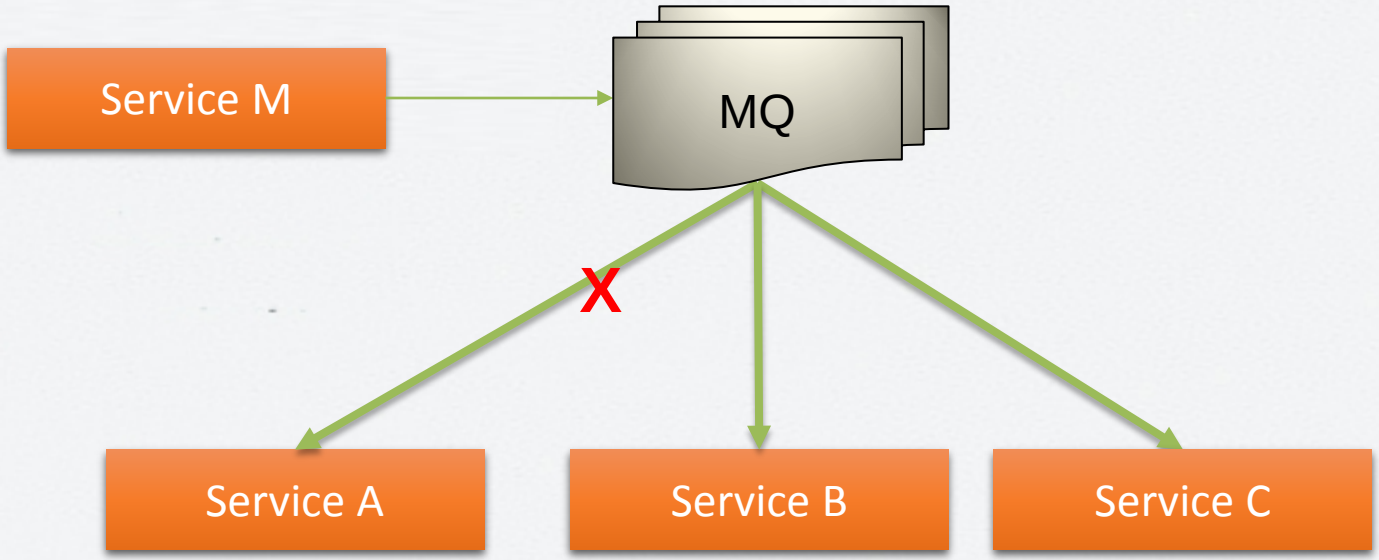
改进方案



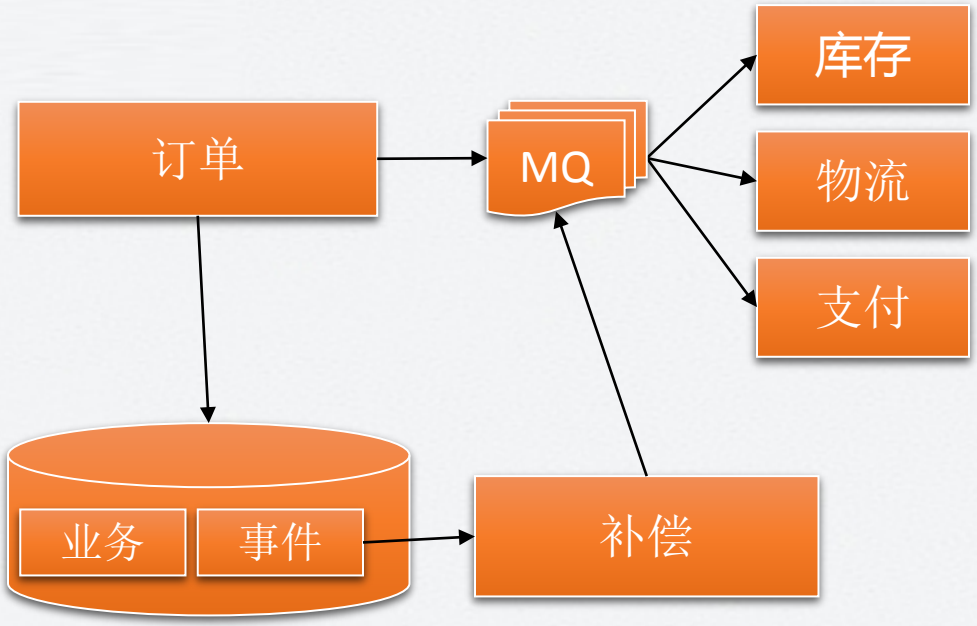
可靠事件



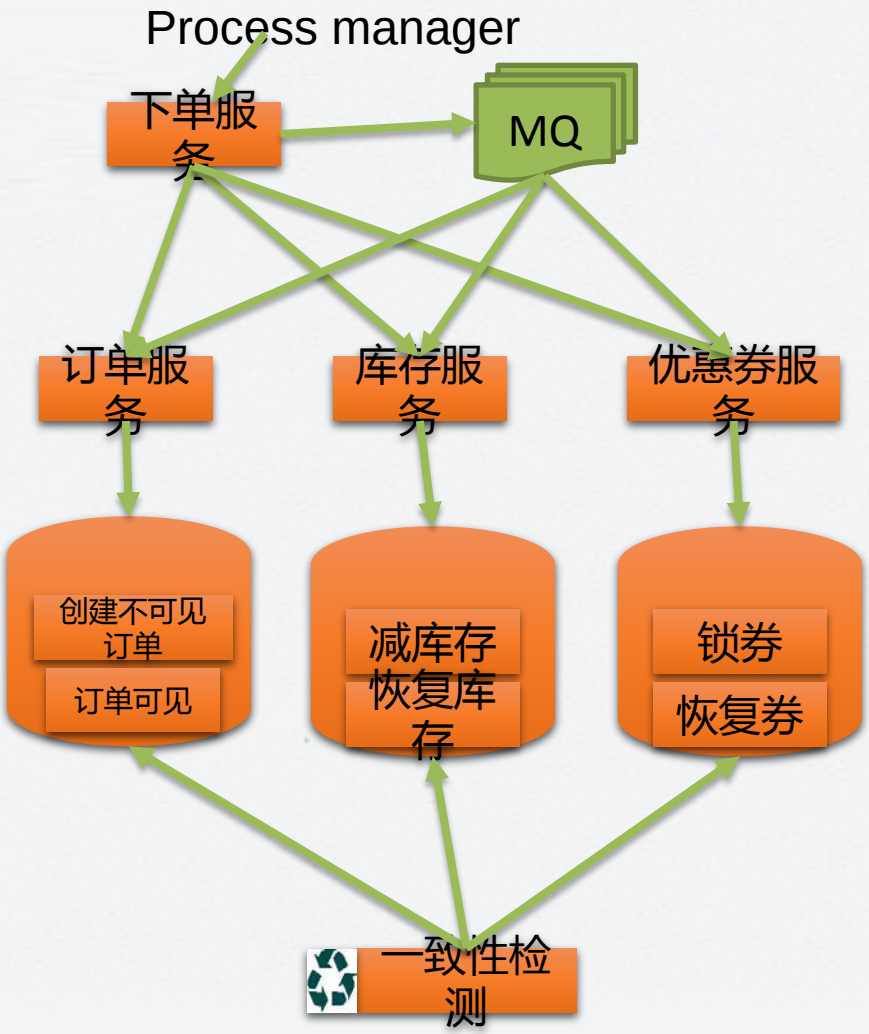
可靠事件



可靠事件

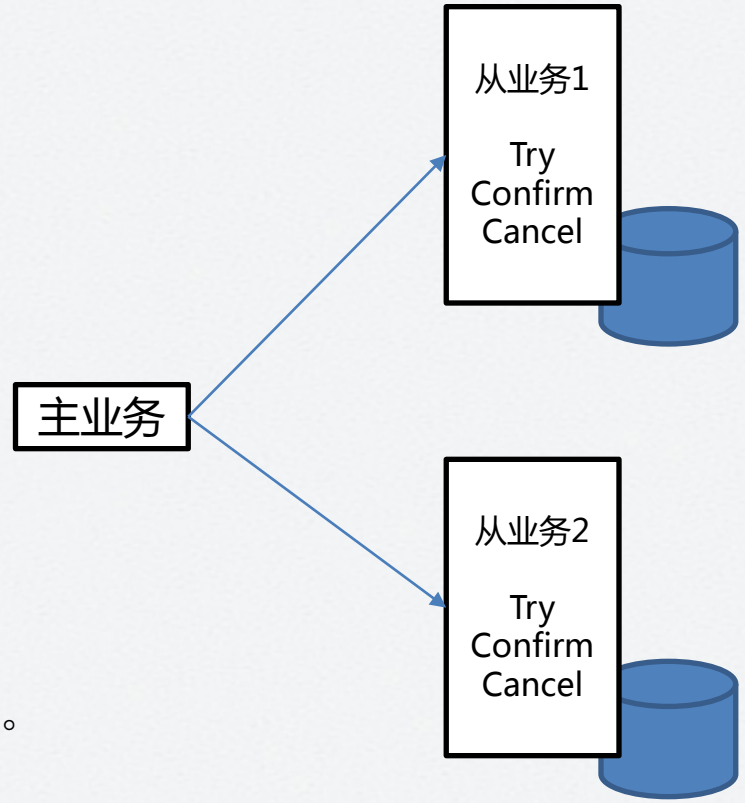


Saga事务模型


Process manager
下单服务
MQ
订单服务
库存服务
优惠券服务
创建不可见订单
订单可见
减库存
恢复库存
锁券
恢复券
一致性检测


开始
1
创建订单（不可见）
2.成功
锁券
3.失败
废单
3.成功
减库存
4.失败
恢复券
4.成功
订单可见
5.失败
恢复库存
5.成功
结束

TCC（Try-Confirm-Cancel）



TCC的优势：

- 在业务层处理，平衡数据库的压力。

TCC的代价：

- 增加业务复杂度，需要提供相应的Try、Confirm、Cancel接口。
- 需要提供幂等性接口。

如何保障幂等？

把编号为5的记录的A字段设置为0，这种操作不管执行多少次都是幂等的。

把编号为5的记录的A字段增加1。这种操作显然就不是幂等的。

方案一、数据库加锁

方案二、分布式锁

方案三、唯一约束

方案四、增加流水表

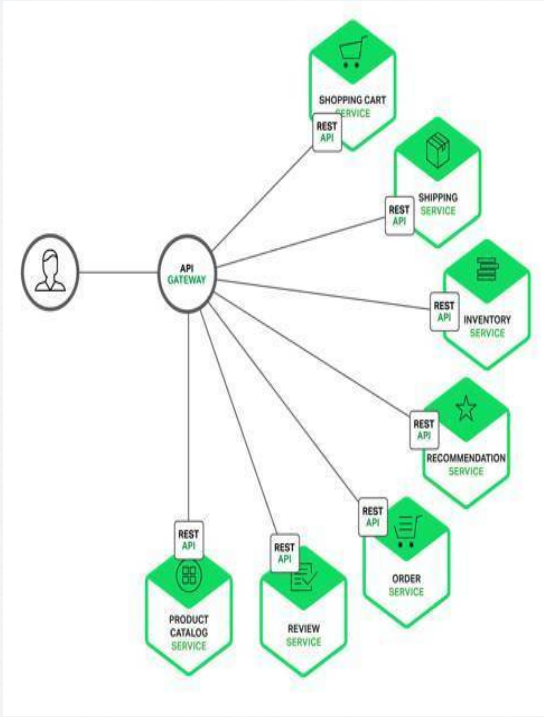
商品ID	库存	状态	……
1101	99	1	……

商品ID	库存	状态	……
1101	99	1	……

拆分后的性能问题

为了提升性能要分布式。
分布式之后……

调用次数暴增，怎么提升性能？



Source：
http://dockone.io/article/482?utm_medium=hao.caibaojian.com&utm_source=hao.caibaojian.com

全局性问题，木桶效应。
最大弥补，以不影响体验为前提。

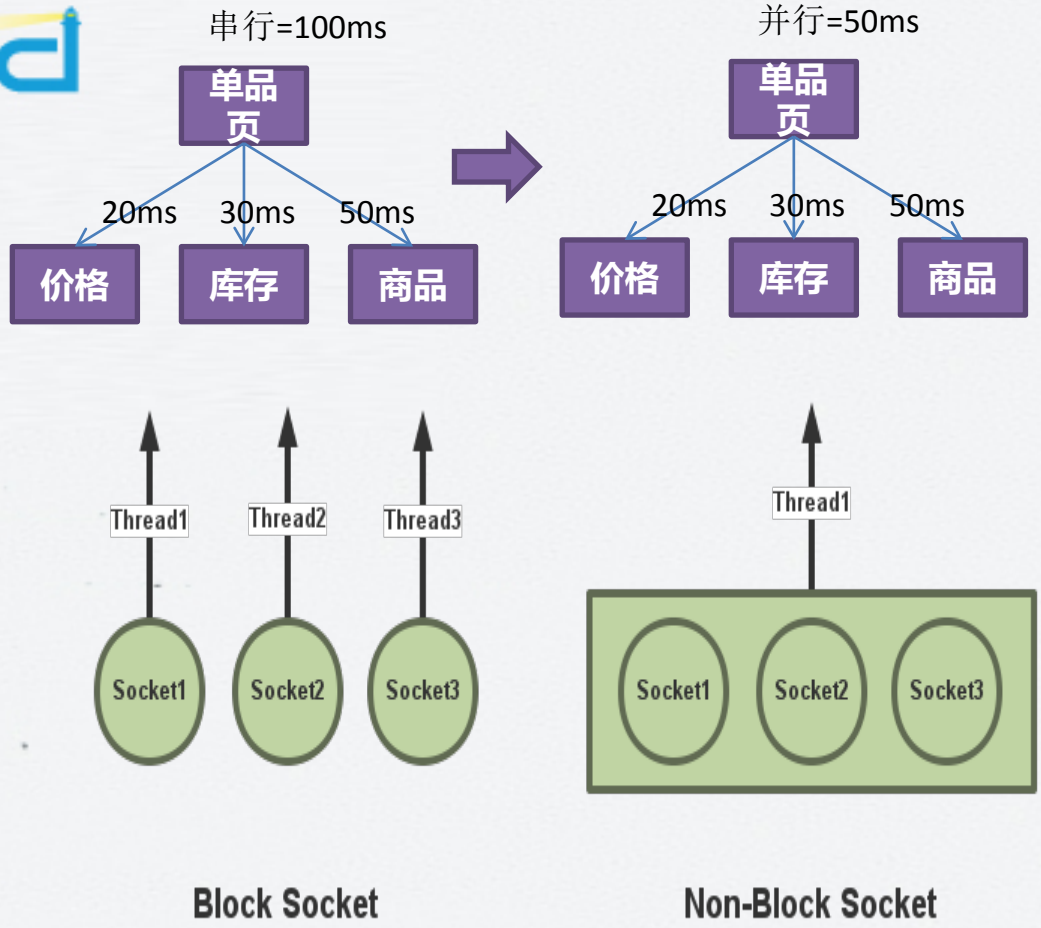
性能是什么？

一定资源占用情况下

- Throughput 吞吐量
- Latency 平均时延

常见的性能问题

1. 内存泄露——导致内存耗尽
2. 过载——突发流量，大量超时重试
3. 网络瓶颈——需要加载的内容太多
4. 阻塞——无尽的等待
5. 锁——并行变串行
6. IO繁忙——大量的读写，分布式
7. CPU繁忙——计算型常见问题
8. 长连接拥塞——连接耗尽

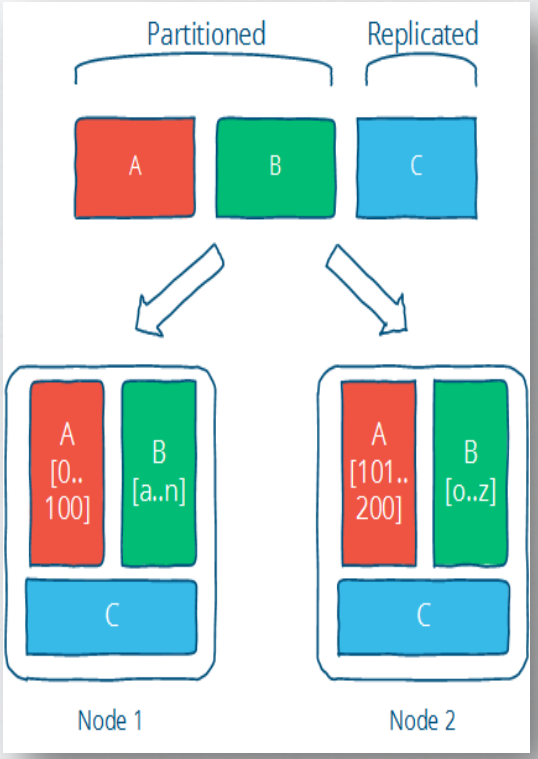


Source：https://segmentfault.com/a/1190000004537204

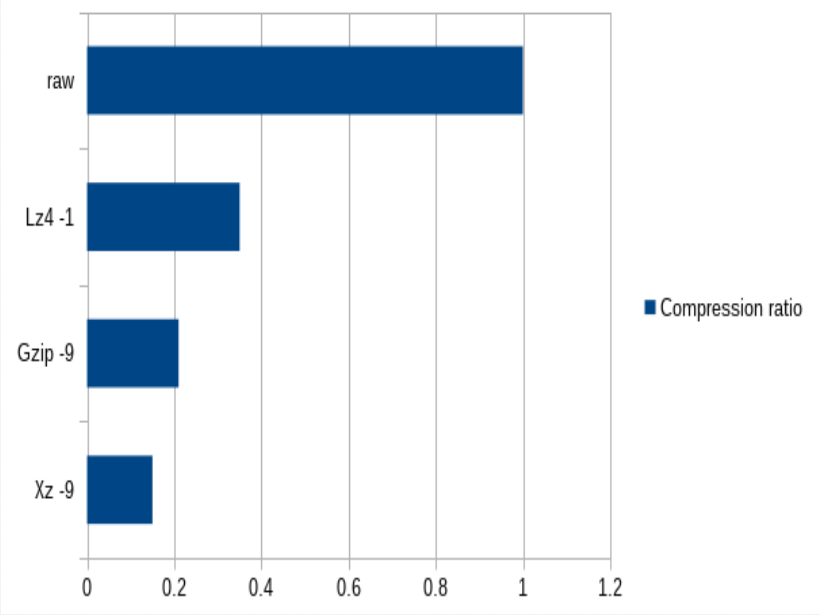
串行转并行，阻塞转非阻塞。

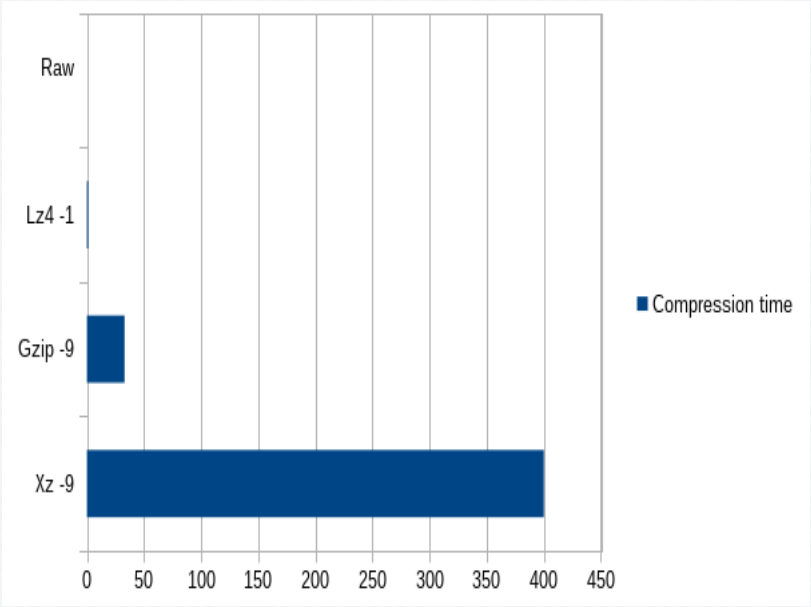
id	title	content	coment_count	status
1	……	……	5	1
2	……	……	1233	2
3	……	……	78	1

冗余：提升读性能



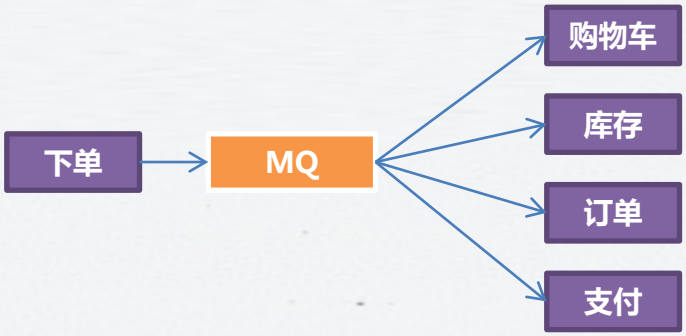
分片、副本突破单节点能力，平衡压力。



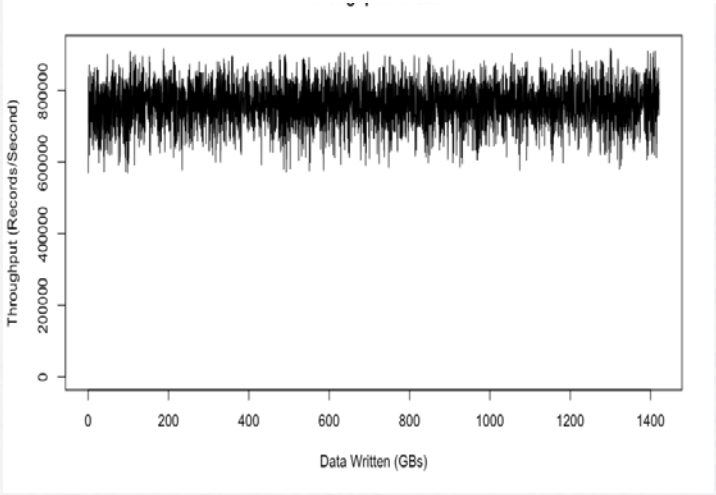


Source：https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO

压缩数据，合并请求，减少传输。



Single producer thread, 3x synchronous replication
421,823 records/sec(40.2 MB/sec)



MQ 削峰填谷，极大提升吞吐量。

缓存被称为性能提升的瑞士军刀。分布式系统的各个角落都充斥着缓存。

本地缓存　　磁盘缓存

CDN

浏览器缓存

客户端缓存

数据库缓存

代理缓存

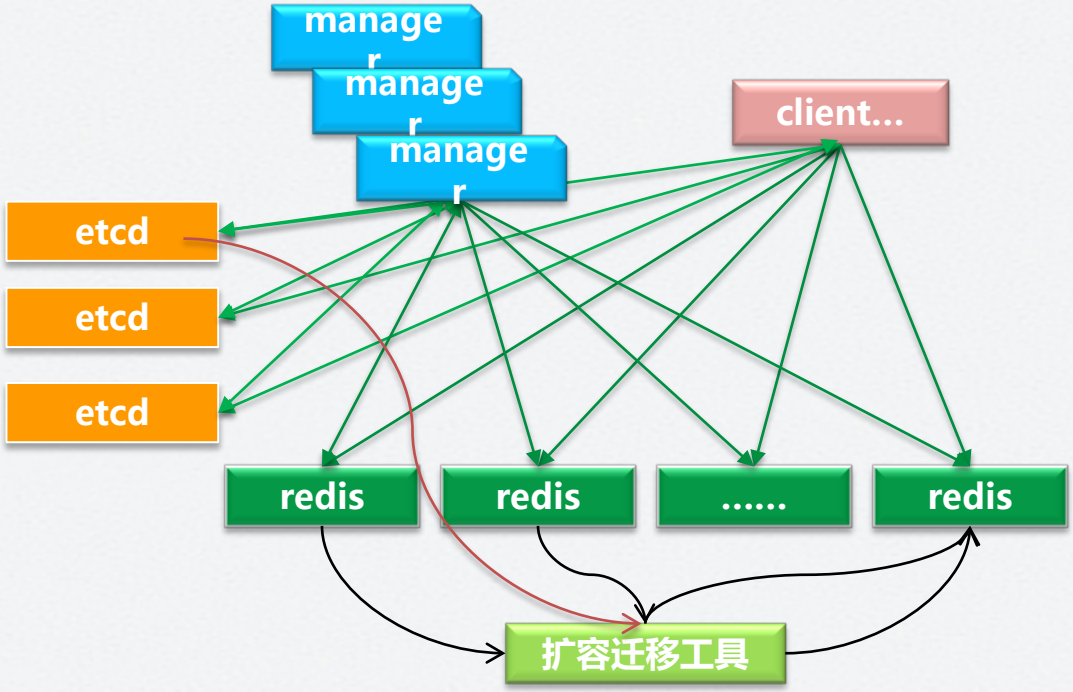
本地缓存？
有状态；
冗余浪费；
FullGC。

分布式缓存

Redis2.X方案：客户端负载均衡方式

- 自动分片，完全分布式；
- 不容易导致雪崩；
- 通过etcd实现动态配置变更；
- 扩容可以采用presharding、数据迁移两种方式；
- 可以根据场景选择是否做可靠的缓存，是否持久化；

- 优势：直连redis，性能较高（有的人可能不在乎这一点，实际上redis cluster做负载均衡的方式也在客户端，作者也是为了追求性能）；
- 问题：SDK升级比较复杂，不过SDK本身升级频率也不高；



分布式缓存

Redis2.X方案：代理负载均衡方式

□开源的例如Twemproxy，codis等，和客户端类似，不同点在于负载均衡放在代理上来做。Proxy作为一个有效的控制点是有很多好处的。

□优势：

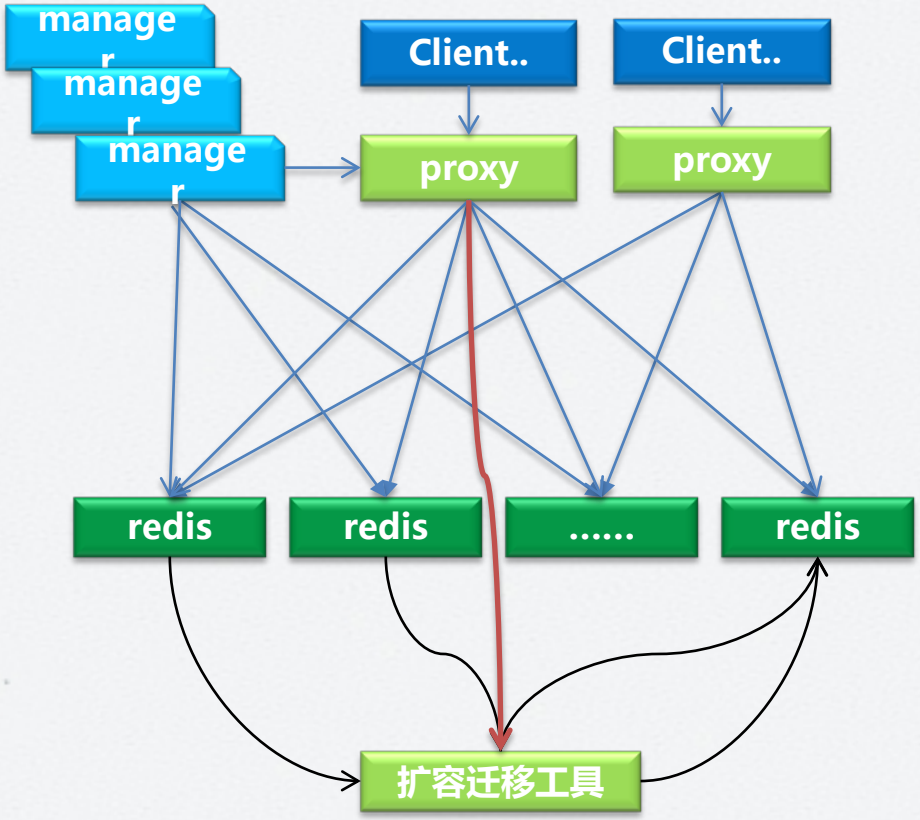
✓控制力强；

✓简单，客户端只需要连接代理服务，以一种服务的方式对外提供，升级方便；

✓收敛连接数；

□问题：

✓性能有损耗，虽然损耗不大；

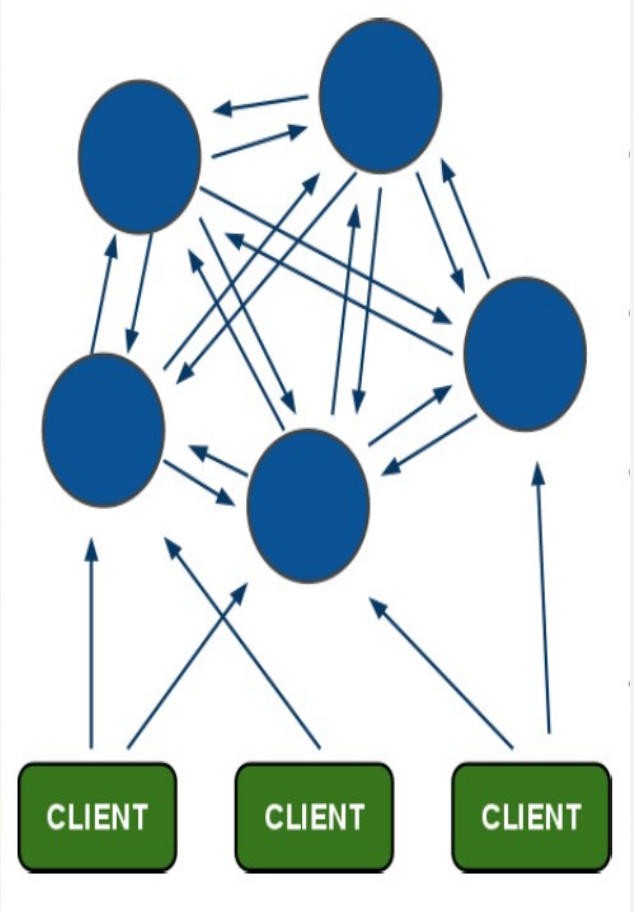


分布式缓存

Redis3.X以上方案

- 所有的redis节点彼此互联(PING-PONG机制)，节点之间使用 Gossip 协议；
- 节点的fail是通过集群中超过半数的master节点检测失效时才生效.
- 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

- 优势：
  - ✓无中心架构（也是很多问题的源泉）；
  - ✓性能高；
- 劣势：
  - ✓SDK过重，升级麻烦；
  - ✓以节点为单位，分区不够小；
  - ✓不一致问题，主从异步、重新选主过程中，Redis 集群不保证强一致性；
  - ✓不够自动化，包括节点变动，数据迁移等操作，都需要执行命令完成；
  - ✓Gossip协议导致的性能瓶颈；



交付型项目复杂的业务场景

- 很难持续交付
- 实现异构I层
- 既要满足大规模部署，又要满足小规模部署
- 项目 VS 产品
- 如何衡量？

无论你选择做什么，总有人说你是错的，又总有这样或那样的困难诱使你相信批评你的人是对的。要找到一条正确的路坚持到最后，你需要的是勇气。

——Ralph Waldo Emerson

工作坊回顾

- 背景
- 基本概念
- 原则
- 扩展性
- 服务划分
- 性能
- 一致性

Make Professional Discovery

MPD

感谢您参加本届MPD！

www.mpd.org.cn

400-812-8020