1. 演进中的架构
（架构并不是被发明出来的，而是持续演进的结果）

- 原始的分布式时代
  - DCE（Distributed Computing Environment）的研究是计算机科学中第一次对分布式有组织领导、有标准可循、有巨大投入的尝试
  - DCE 提出的分布式服务的设计主旨：“让开发人员不必关心服务是远程还是本地，都能够透明地调用服务或者访问资源”
  - DCE或者后来出现的CORBA都是没有取得成功
  - 将一个系统划分到不同的机器上运行，所带来的服务发现、路由、通信、容错、隔离、配置、传输、数据一致性和编码复杂度等方面复杂问题所付出的代价远大于分布式带来的收益
  - 某个功能能够进行分布式，并不意味着它就应该进行分布式，强行追求透明的分布式操作，只会自寻苦果——Kyle Brown
- 单体应用时代
  - 单体应用又被称为巨石系统（Monolithic Application）
  - 单体应用不应该被贴上“反派角色”的标签
  - 单体系统的缺陷，只有满足以下两个方面才有讨论的价值
    1. 基于软件的性能需求超过了单机的性能
    2. 软件的开发人员规模明显超过了“2 Pizza Team”范畴的前提下才有讨论的价值
  - 单体应用的缺陷：
    1. 隔离与自治能力上的欠缺
    2. 无法做到“停掉半个进程，重启 1/4 个程序”这样不合逻辑的操作
    3. 单体系统的技术栈异构能力较弱
  - 微服务，单体架构哪种更好用，更优秀？笔者认为“好用和优秀”不会是放之四海皆准的
  - 微服务超越单体应用成为主流的主要原因
    单体系统很难重新“Phoenix”的特性
    构筑可靠系统从“追求尽量不出错”，到正视“出错是必然”的观念转变，才是微服务架构得以跳战并逐步开始取代运作了数十年的单体架构的底气所在
- SOA时代
  - 面向服务的架构是一次具体地、系统性地成功解决分布式服务主要问题的架构模式。
  - 烟囱式架构
    微内核架构（插件形式存在）
    事件驱动架构
  - 企业服务总线（Enterprise Service Bus，ESB）的消息管道来实现各个子系统之间的通信交互，令各服务间在 ESB 调度下无须相互依赖即能相互通信，既带来了服务松耦合的好处，也为以后可以进一步实施业务流程编排（Business Process Management，BPM）提供了基础
  - SOAP 协议被通用记得化的本质原因：过于严格的规范定义带来过度的复杂性。而构建在 SOAP 基础之上的 ESB、BPM、SCA、SDO 等诸多上层框架，进一步加剧了这种复杂性
    SOA 最终没有获得成功的致命伤与当年的无限如此一辙，尽管有 Sun Microsystems 和 IBM 等一众巨头在背后力挺，ESB 仍然败于以 Spring、Hibernate 为代表的“草根框架”，可见一旦脱离人民群众，得究会淹没在群众的海洋之中，连信息技术也不例外的讨
- 微服务时代
  - 微服务是一种软件开发技术，是一种 SOA 的变体形式。
  - 微服务真正的崛起是在 2014 年， Martin Fowler 与 James Lewis 合写的文章《Microservices: A Definition of This New Architectural Term》中首次了解到微服务的
  - 微服务是一种通过多个小型服务组合来构建单个应用的架构风格
    这些服务围绕业务能力而非特定的技术标准构建。
    各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。
    服务采取轻量级的通信机制和自动化的部署机制实现通信与运维
  - Thoughtworks 首席咨询师 James Lewis 做了题为《Microservices - Java, the Unix Way》的主题演讲，其中提到了单一服务职责、廉威定律、自动扩展、伺城驱动设计等原则
  - 微服务的九个核心的业务与技术特征
    - 围绕业务能力构建
    - 分散治理
    - 通过服务来实现独立自治的组件
    - 产品化思维
    - 数据去中心化
      分布式中要处理好一致性的问题也相当困难，很多时候都没法使用传统的事务处理来保证，
      但是用害怕权宜其轻，有一些必要的代价仍是值得付出的
    - 强终端弱管道
      如果服务需要上面的额外通信能力，就应该在服务自己的 Endpoint 上解决，而不是在通信管道上一揽子处理。
      微服务提倡类似于经典 UNIX 过滤器那样简单直接的通信方式，RESTful 风格的通信在微服务中会是更加合适的选择
    - 容错性设计
    - 演进式设计
    - 基础设施自动化
  - 微服务时代充满着自由的气息，微服务时代充斥着迷茫的选择。软件架构不会止步于自由，微服务仍不是架构探索终点
    如果有下一个时代，笔者希望是信息系统能同时拥有微服务的自由权利，围绕业务能力构建自己的服务而不受技术规范塑束，但同时又不必以承担自行解决分布式的问题的责任为代价
- 后微服务时代
  - 从软件层面独力应对微服务架构问题，发展到此，硬一体，合力应对架构问题的时代，此即为“后微服务时代”
  - DCE 中未能实现的“透明的分布式应用”成为可能，Martin Fowler 设想的“凤凰服务器”成为可能，Chad Fowler 提出的“不可变基础设施”也成为可能，从软件层面独力应对分布式架构所带来的各种问题，发展到应用代码与基础设施软、硬一体，合力应对架构问题的时代，现在常被媒体称以“云原生”这个假为抽象的名字加以宣传
  - Kubernetes 成为容器战争胜利者标志着后微服务时代的开端
    哪吗的 Kubernetes 远历不如之前的 Spring Cloud 方案。
    1. 这是因为有一些问题处于应用系统与基础设施的边缘，使得完全在基础设施层面中确实很难精细化地处理（如针对某个应用服务的细断降级策略控制）
    2. 基础设施是针对整个容器来管理的，粒度相对粗矿，只能到容器层面，对单个应用服务就很难有效管控
  - 虚拟化的基础设施很快实现了第二次进化，引入了今天被称为“服务网格”（Service Mesh）的“边车代理模式”（Sidecar Proxy）
    服务网格将会成为微服务之间通信交互的方法模式，把“选择什么通信协议”、“怎样调度流量”、“如何认证授权”之类的技术问题隔离于程序代码之外
    这才是最理想的Smart Endpoints解决方案
  - 如果说微服务架构是分布式系统这条路的烟救，那无服务架构，也许就是“不分布式”的云端系统这条路的起点。
- 无服务时代
  - 涉及内容
    - 后端设施（BaaS）
    - 函数（FaaS）
  - 挑战
    - FaaS服务的冷启动问题
    - 应用场景无法全覆盖的缺陷（参管理系统、网络游戏等应用，又或者说所有具有业务逻辑复杂，依赖服务端状态，响应速度要求较高，需要长链接等这些特征的应用）
    - 软件开发的最大挑战就在于只能在不完整的信息下决定当前要处理的问题，时架今日，依然很难预想在架构演进之路的前方，微服务和无服务之后还会出现何种形式的架构风格
    - 尽管目光所及之处，只是不远的前方，即使如此，依然可以看到那里有许多值得去完成的工作在等待我们——图灵

分布式共识算法

- 系统稳定标准
  - 可靠性
    - 可靠不一定是可用的
    - 可靠是一种最终状态，通过999的可用性，保证了系统的可靠性
    - 可靠性的方式有多重
  - 可用性
- 同步方式
  - 全同步复制
    - 状态转移
      - 牺牲可用性
  - 操作转移
    - 状态机复制
      - 令源状态转换为目标状态。能够使用确定的操作 在计算机科学中被称为状态机（ State Machine ）
    - 少数服从多数” 的原则 /Quorum 机制
    - 演变成一种协商共识算法
- Paxos
  - 开拓了分布式共识算法的发展思路，不会直接用于实践
  - 算法流程
    - 提案节点（Proposer）
      - 提出对某个值进行设置操作的节点
    - 决策节点（Acceptor）
      - 进行投票的节点
    - 记录节点（Learner）
      - 不参与提案和决策，如少数派的节点恢复时候的状态就是该状态
  - 算法机制
    - 加锁就不完全等同于开发控制中以互斥量来实现的加锁，还必须提供一个其他节点能抢占锁的机制，以避免因通信问题而出现死锁
    - 协必须是可抢占的
    - 两阶段
      - Prepare
      - Accept
    - 两个承诺
      - 不会接受提案id<=n的prepare请求
      - 不会接受无id<=n的Accept请求
    - 一个应答
      - 不违背以前作出的承诺的前提下，回复已经批准过的提案中 ID 最大的那个提案所设定的值和提案 ID，如果该值从来没有被任何提案设定过，则返回空值。如果违反此前做出的承诺，即收到的提案 ID 并不是决策节点收到过的最大的，那对并直接对此 Prepare 请求不予理会
    - 操作时序
  - 分布式系统有五个节点，分别命名为 S1、S2、S3、S4、S5
    - 背景
      - 两个开发的请求分别希望将同一个值分别设定为 X（由 S1作为提案节点提出）和 Y（由 S5作为提案节点提出）
      - 以 P 代表准备阶段，以 A 代表批准阶段

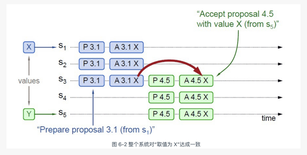


图 6-2 整个系统对“取值为 X”达成一致

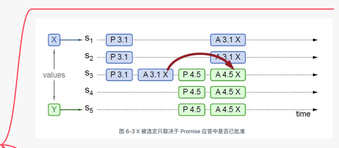


图 6-3 X 被选定只取决于 Promise 应答中是否已批准

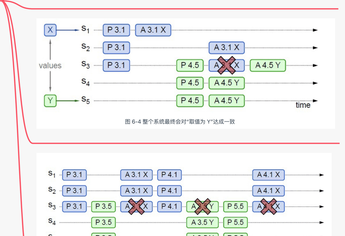


图 6-4 整个系统最终会对“取值为 Y”达成一致

图 6-5 批准阶段失败，形成活锁

  - 缺陷
    - 只能对单个值形成决议
    - 决议的形成至少需要两次网络请求和应答（准备和批准阶段各一次），高并发情况下两产生较大的网络开销，极端情况下甚至可能形成活锁
  - 解决的问题
    - 活锁问题与许多 Basic Paxos 异常场景中所遭遇的麻烦
  - 改进点
    - 增加选主过程
      - 提案节点会通过定时轮询（心跳）寻找有一个主提案节点
      - 没有的话就会在心跳超时后使用 Basic Paxos 中定义的准备、批准的两轮网络交互过程，向所有其他节点广播自己希望竞选主节点的请求
    - 角色转换 ，主、从代替原来的提案，决策、记录节点
  - 时序

  - 主从机制下的协商共识时序

- Multi Paxos
  - 网关路由
    - 网络I/O模型
      - 同步I/O
        - 异步 I/O
          - 你在美团外卖订了个盒饭，付款之后你自己选干嘛还干嘛去，饭做好了骑手自然会到门口打电话通知你
          - 异步 I/O 中数据到达缓冲区后，不需要由调用进程主动进行从缓冲区复制数据的操作，而是复制完成后由操作系统向线程发送信号
        - 阻塞 I/O
          - 你去到饭堂，发现饭还没做好，你也干不了别的，只能打个瞌睡（线程休眠），直到饭做好，这就是被阻塞了
          - 阻塞 I/O 是最直观的 I/O 模型，逻辑清晰，也比较节省 CPU 资源，但缺点就是线程休眠所带来的上下文切换
        - 非阻塞 I/O
          - 你去到饭堂，发现饭还没做好，你就回去了，然后每隔 3 分钟来一次饭堂看饭做好了没，直到饭做好
          - 非阻塞 I/O 能够避免线程休眠，对于一些很快就能返回结果的请求，非阻塞 I/O 可以节省切换上下文切换的消耗，但是对于较长时间才能返回的请求，非阻塞 I/O 反而白白浪费了 CPU 资源
        - 多路复用 I/O
          - 代表整个宿舍去饭堂打饭，去到饭堂，发现饭还没做好，还是继续打瞌睡，但哪个舍友的饭好了，你就马上把那份饭送回去
          - 多路复用 I/O 是目前的高并发网络应用的主流
          - 实现
            - select
            - epoll
            - kqueue
      - 信号驱动 I/O
        - 信号驱动 I/O 与异步 I/O 的区别是 “从缓冲区获取数据” 这个步骤的处理
        - 前者收到的通知是可以开始进行复制操作了，即要你自己从饭堂拿回宿舍，在复制完成之前线程处于阻塞状态，所以它仍属于同步 I/O 操作，而后者收到的通知是复制操作已经完成
      - Zuul
        - 1.0
          - 阻塞I/O
          - 后端I/O密集型比较消耗网关的cpu。阻塞的睡眠协调需要做线程切换
          - 后端是CPU密集型，比较介绍网关的cpu消耗
        - 2.0
          - netty多模型
      - Nginx Ingress Controller
      - KONG
    - 性能对比
      - 做定性分析，不好做定量分析
    - BFF 网关

图 7-3 BFF 网关

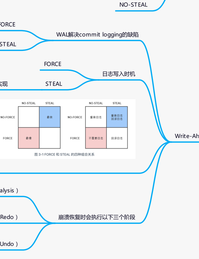
  - 客户端负载均衡器
    - 代理负载均衡器
      - Netflix Ribbon
      - Spring Cloud Load Balancer
      - Service Mesh
      - 代理均衡器对此前的客户端负载均衡器的改进是将原本嵌入在服务进程中的均衡器提取出来，作为一个进程之外，同一 Pod 之内的特殊服务，放到边车代理中去实现
      - 增加的消耗
        - 从进程内通信到通过网络协议栈进行数据交换,要进行打包拆包、计算校验和、维护序列号等网络数据的收发步骤，流量比起之前的客户端均衡器确实多增加了一系列处理步骤
      - 优化处理
        - Kubernetes 严格保证了同一个 Pod 中的容器不会跨越不同的节点，这些容器共享着同一个网络名称空间，因此代理均衡器与服务实例的交互，实质上是对本机回环设备的访问，仍然要比真正的网络交互高效且稳定得多
        - 代理均衡器不再受编程语言的限制

- 从类库到服务
  - 应用
    - 比特币
  - Xerox公司（施乐公司）
    - RPC的提出者
    - MVC的提出者
  - 执行过程
    - 固定周期（1s）随机选择它相连接的k个节点
    - 每一个节点收到消息后，如果这个消息是它之前没有收到过的，将在下一个周期内，选择除了发送消息给它的那个节点外的其他相邻 k 个节点发送相同的消息，直到最终网络中所有节点都收到了消息，尽管这个过程需要一定时间，但是理论上最终网络的所有节点都会拥有相同的消息
  - 缺陷
    - 无法评估完成时间
    - 消息的冗余，增加网络传输的压力
  - 解决
    - 反熵（Anti-Entropy）
      - 去除老异化，同步节点的全部数据
    - 传谣（Rumor-Mongering）
      - 以传播消息为目标，只对外发送变更信息，这样消息数据量将显著缩减
  - 解决的问题
    - 从进程内演变成进程外调用
  - 需要做的事情
    - 服务发现
    - 服务的网关路由
    - 服务的负载均衡
  - 服务发现
    - 如何在基础设施和网络协议层面，对应用尽可能无感知。方便地实现服务发现是目前服务发现的一个主要发展方向
  - 可用与可靠
    - 服务的注册
    - 服务的维护
    - 服务的发现
    - 注册中心的高可用
      - Eureka 的选择是优先保证高可用性 AP
      - Consul 的选择是优先保证高可靠性 CP
        - Raft算法实现数据的同步+Gossip支持多数据中心之间更大规模的服务同步
      - ZK CP
  - 注册中心实现
    - 在分布式 K/V 存储框架上自己开发的服务发现
      - ZooKeeper
        - ZAB
      - Doozerd
      - Etcd
        - Raft
    - 以基础设施（主要是指 DNS 服务器）来实现服务发现（以基础设施来做服务发现，好处是对应用透明，任何语言、框架、工具都肯定是支持 HTTP、DNS 的，所以完全不受程序技术选型的约束，但环处是透明的并不一定是简单的）
      - SkyDNS
      - CoreDNS
    - 专门用于服务发现的框架和工具
      - Eureka
      - Consul
      - Nacos

- 事务处理
  - 全局的事务管理器
  - 局部的资源管理器
  - 本地事务
    - 隔离
      - 隔离级别
        - 串行化
        - 可重复度
          - 幻读
        - 读已提交
          - 不可重复读
        - 读未提交
          - 脏读
        - 不同隔离级别以及幻读、不可重复读、脏读等问题都只是表面现象，是各种锁在不同加锁时间上组合应用所产生的结果，以锁为手段来实现隔离性才是数据库表现出不同隔离级别的根本原因
      - 现代数据库提供的锁机制
        - 写锁
          - 也叫做排它锁
          - 如果数据有加写锁，就只有持有写锁的事务才能对数据进行写入操作，数据加持着写锁时，其他事务不能写入数据，也不能施加读锁
        - 读锁
          - 也叫做共享锁,多个事务可以对同一个数据添加多个读锁，数据被加上读锁后就不能再被加上写锁
          - 对于持有读锁的事务，如果该数据只有它自己一个事务加了读锁，允许直接将其升级为写锁，然后写入数据
        - 范围锁
          - 对于某个范围直接加排他锁，在这个范围内的数据不能被写入
      - MVCC
        - 控制原理
          - 解决“一个事务读+另一个事务写”的隔离问题
          - MVCC 是一种读取优化策略，它的“无锁”是特指读取时不需要加锁
          - ：CREATE_VERSION 和 DELETE_VERSION，这两个字段记录的值都是事务 ID，事务 ID 是一个全局严格递增的数值
          - 插入数据时：CREATE_VERSION 记录插入数据的事务 ID，DELETE_VERSION 为空
          - 删除数据时：DELETE_VERSION 记录删除数据的事务 ID，CREATE_VERSION 为空
          - 修改数据时：将修改数据视为“删除旧数据，插入新数据”的组合，即先将原有数据复制一份，原有数据的 DELETE_VERSION 记录修改数据的事务 ID，CREATE_VERSION 为空。复制出来的新数据的 CREATE_VERSION 记录修改数据的事务 ID，DELETE_VERSION 为空
        - 读取机制
          - 隔离级别是可重复读：总是读取 CREATE_VERSION 小于或等于当前事务 ID 的记录，在这个前提下，如果数据仍有多个版本，则取最新（事务 ID 最大）的
          - 隔离级别是读已提交：总是取最新的版本即可，即最近被 Commit 的那个版本的数据记录
          - 读未提交不涉及
          - 串行化不涉及
        - MVCC只是针对读+写的场景做的性能优化，无法解决写+写的场景，写+写的场景还是需要通过锁进行解决
          - 悲观加锁
          - 乐观加锁
    - 隔离
      - 解决
        - 一个事务读+另一个事务写”的隔离问题
      - MVCC 是一种读取优化策略，它的“无锁”是特指读取时不需要加锁
      - ：CREATE_VERSION 和 DELETE_VERSION，这两个字段记录的值都是事务 ID，事务 ID 是一个全局严格递增的数值
      - 插入数据时：CREATE_VERSION 记录插入数据的事务 ID，DELETE_VERSION 为空
      - 删除数据时：DELETE_VERSION 记录删除数据的事务 ID，CREATE_VERSION 为空
      - 修改数据时：将修改数据视为“删除旧数据，插入新数据”的组合，即先将原有数据复制一份，原有数据的 DELETE_VERSION 记录修改数据的事务 ID，CREATE_VERSION 为空。复制出来的新数据的 CREATE_VERSION 记录修改数据的事务 ID，DELETE_VERSION 为空
      - 隔离级别是可重复读：总是读取 CREATE_VERSION 小于或等于当前事务 ID 的记录，在这个前提下，如果数据仍有多个版本，则取最新（事务 ID 最大）的
      - 隔离级别是读已提交：总是取最新的版本即可，即最近被 Commit 的那个版本的数据记录
      - 读未提交不涉及
      - 串行化不涉及
      - 悲观加锁
      - 乐观加锁
    - 隔离级别
      - 串行化
      - 可重复度
        - 幻读
      - 读已提交
        - 不可重复读
      - 读未提交
        - 脏读
      - 不可重复读的过程是因为两个事务都读取了同一个数据，然后其中一个事务先修改了这个数据并且提交了，导致另一个事务后续读取的结果与第一次读取的结果不一致
      - 幻读则是因为一个事务先后读取了一个范围的记录，然后另一个事务在这个范围插入了新的记录，导致后续读取时两次读到的记录数量不一致
  - 全局事务
    - XA
      - 三阶段提交
        - 过程
          - 准备
            - CanCommit
            - PreCommit
          - 提交
        - 单点问题和准备阶段的性能问题
      - 局部的提交
        - 时序图

图 3-1 两段式提交的交互时序示意图

        - 问题
          - 单点问题
          - 性能问题
            - 两次远程调用
            - 三次数据持久化
          - 一致性风险
    - 全局事务
      - 准备阶段
        - 准备操作是尽量做日志中记录全部事务提交操作所要做的内容，它与本地事务中真正提交的区别只是暂不写入最后一条 Commit Record 而已，这意味着在做完数据持久化后并不立即就拉隔离性，即仍继续持有锁，维持数据对其他非事务问题要求的隔离状态
      - 提交阶段
        - 协调者如果在上一阶段收到所有事务参与者回复的 Prepared 消息，则先自己在本地持久化事务状态为 Commit
        - 在此操作完成后向所有参与者发送 Commit 指令，所有参与者立即执行提交操作；否则，任意一个参与者回复了 Non-Prepared 消息，或任意一个参与者超时未回复，协调者将自己的事务状态转为火化为 Abort 之后，向所有参与者发送 Abort 指令，参与者立即执行回滚操作



    - 全局的事务管理器
    - 局部的资源管理器
    - 准备阶段
    - 提交阶段
    - 时序图
    - 局部的提交
    - 问题
      - 单点问题
      - 性能问题
        - 两次远程调用
        - 三次数据持久化
      - 一致性风险
  - 本地事务
    - 原子性和持久性
      - 事务描述
        - 原子性
        - 隔离性
        - 持久性
      - Commit Logging
        - oceanbase
        - 缺陷
          - 所有对数据的真实修改都必须发生在事务提交以后，即日志写入了 Commit Record 之后
        - NO-FORCE
        - NO-STEAL
      - Write-Ahead Logging
        - WAL解决commit logging的缺陷
          - NO-FORCE
          - NO-STEAL
        - 日志写入时机
          - FORCE
          - STEAL
        - 通过undolog实现

图 3-5 FORCE 和 STEAL 的四种组合关系

      - 崩溃恢复时会执行以下三个阶段
        - 分析阶段（Analysis）
          - 找出所有没有 End Record 的事务，组成待恢复的事务集合，
          - 这个集合至少会包括 Transaction Table 和 Dirty Page Table 两个组成部分
        - 重做阶段（Redo）
          - 找出所有包含 Commit Record 的日志，将这些日志修改的数据写入磁盘，
          - 写入完成后在日志中增加一条 End Record，然后移除出待恢复事务集合
        - 回滚阶段（Undo）
          - 它对被称为 Loser，根据 Undo Log 中的信息，
          - 将已经提前写入磁盘的信息重新改写回去，以达到回滚这些 Loser 事务的目的
    - 隔离
      - 隔离级别
        - 串行化
        - 可重复度
          - 幻读
        - 读已提交
          - 不可重复读
        - 读未提交
          - 脏读
      - 现代数据库提供的锁机制
        - 写锁
          - 也叫做排它锁
          - 如果数据有加写锁，就只有持有写锁的事务才能对数据进行写入操作，数据加持着写锁时，其他事务不能写入数据，也不能施加读锁
        - 读锁
          - 也叫做共享锁,多个事务可以对同一个数据添加多个读锁，数据被加上读锁后就不能再被加上写锁
          - 对于持有读锁的事务，如果该数据只有它自己一个事务加了读锁，允许直接将其升级为写锁，然后写入数据
        - 范围锁
          - 对于某个范围直接加排他锁，在这个范围内的数据不能被写入
      - MVCC
        - 控制原理
          - 解决“一个事务读+另一个事务写”的隔离问题
          - MVCC 是一种读取优化策略，它的“无锁”是特指读取时不需要加锁
          - ：CREATE_VERSION 和 DELETE_VERSION，这两个字段记录的值都是事务 ID，事务 ID 是一个全局严格递增的数值
          - 插入数据时：CREATE_VERSION 记录插入数据的事务 ID，DELETE_VERSION 为空
          - 删除数据时：DELETE_VERSION 记录删除数据的事务 ID，CREATE_VERSION 为空
          - 修改数据时：将修改数据视为“删除旧数据，插入新数据”的组合，即先将原有数据复制一份，原有数据的 DELETE_VERSION 记录修改数据的事务 ID，CREATE_VERSION 为空。复制出来的新数据的 CREATE_VERSION 记录修改数据的事务 ID，DELETE_VERSION 为空
        - 读取机制
          - 隔离级别是可重复读：总是读取 CREATE_VERSION 小于或等于当前事务 ID 的记录，在这个前提下，如果数据仍有多个版本，则取最新（事务 ID 最大）的
          - 隔离级别是读已提交：总是取最新的版本即可，即最近被 Commit 的那个版本的数据记录
          - 读未提交不涉及
          - 串行化不涉及
        - MVCC只是针对读+写的场景做的性能优化，无法解决写+写的场景，写+写的场景还是需要通过锁进行解决
          - 悲观加锁
          - 乐观加锁
    - 崩溃恢复时会执行以下三个阶段
      - 分析阶段（Analysis）
      - 重做阶段（Redo）
      - 回滚阶段（Undo）
    - 原子性和持久性
      - Write-Ahead Logging
        - 日志写入时机
          - FORCE
          - STEAL
        - 通过undolog实现
        - WAL解决commit logging的缺陷
          - NO-FORCE
          - NO-STEAL
      - Commit Logging
        - oceanbase
        - 缺陷
          - 所有对数据的真实修改都必须发生在事务提交以后，即日志写入了 Commit Record 之后
        - NO-FORCE
        - NO-STEAL
    - 事务描述
      - 原子性
      - 隔离性
      - 持久性
  - AREIS理论
    - （Algorithms for Recovery and Isolation Exploiting Semantics，ARIES），直接翻译过来是“基于语义的恢复与隔离算法

风险架构

1. 风险架构概述

1.1 风险架构的定义与目标

1.2 风险架构的核心原则

1.3 风险架构的范围与边界

2. 风险架构视角

2.1 业务风险架构

2.1.1 业务流程风险

2.1.2 业务规则风险

2.1.3 业务决策风险

2.2 应用风险架构

2.2.1 应用组件风险

2.2.2 应用交互风险

2.2.3 应用数据风险

2.3 数据风险架构

2.3.1 数据存储风险

2.3.2 数据流转风险

2.3.3 数据质量风险

2.4 技术风险架构

2.4.1 基础设施风险

2.4.2 平台风险

2.4.3 网络风险

2.5 安全风险架构

2.5.1 身份与访问风险

2.5.2 数据安全风险

2.5.3 应用安全风险

2.6 运维风险架构

2.6.1 监控风险

2.6.2 变更风险

2.6.3 应急风险

2.7 合规风险架构

2.7.1 法规遵从风险

2.7.2 审计风险

2.7.3 隐私风险

3. 风险识别方法

3.1 风险识别概述

3.2 自上而下的风险识别

3.2.1 战略目标分析

3.2.2 业务流程分析

3.2.3 专家访谈

3.3 自下而上的风险识别

3.3.1 资产清单梳理

3.3.2 威胁场景分析

3.3.3 漏洞扫描与评估

3.4 风险识别工具与技术

3.4.1 风险登记册

3.4.2 风险矩阵

3.4.3 风险地图

4. 风险评估

4.1 风险评估概述

4.2 风险评估方法

4.2.1 定性评估

4.2.2 定量评估

4.2.3 半定量评估

4.3 风险评估流程

4.3.1 风险分析

4.3.2 风险评价

4.4 风险评估工具

5. 风险应对策略

5.1 风险规避

5.2 风险降低

5.2.1 预防性控制

5.2.2 检测性控制

5.2.3 纠正性控制

5.3 风险转移

5.3.1 保险

5.3.2 外包

5.3.3 合同约束

5.4 风险接受

5.4.1 风险容忍

5.4.2 风险偏好

5.4.3 应急计划

6. 风险监控与报告

6.1 风险监控概述

6.2 关键风险指标（KRI）

6.2.1 KRI 的定义与选择

6.2.2 KRI 的阈值设定

6.2.3 KRI 的监控频率

6.3 风险报告

6.3.1 风险报告的内容

6.3.2 风险报告的受众

6.3.3 风险报告的频率

6.4 风险沟通

6.4.1 内部沟通

6.4.2 外部沟通

7. 风险架构的治理与组织

7.1 风险治理结构

7.1.1 董事会与高管层

7.1.2 风险管理委员会

7.1.3 风险管理部门

7.2 三道防线模型

7.2.1 第一道防线：业务部门

7.2.2 第二道防线：风险管理与合规

7.2.3 第三道防线：内部审计

7.3 风险文化建设

7.3.1 风险意识培训

7.3.2 风险文化评估

8. 风险架构的持续改进

8.1 风险架构的评审机制

8.1.1 定期评审

8.1.2 触发式评审

8.2 风险架构的优化

8.2.1 优化方法

8.2.2 优化实施

8.3 风险架构的成熟度评估

8.3.1 成熟度模型

8.3.2 评估结果应用

9. 风险架构的实施路线图

9.1 实施阶段划分

9.1.1 第一阶段：基础建设

9.1.2 第二阶段：深化应用

9.1.3 第三阶段：持续优化

9.2 关键里程碑

9.3 资源与预算

9.3.1 人力资源

9.3.2 技术投入

9.3.3 外部支持

10. 风险架构的挑战与应对

10.1 常见挑战

10.1.1 组织层面挑战

10.1.2 技术层面挑战

10.1.3 业务层面挑战

10.2 应对策略

10.2.1 组织层面应对

10.2.2 技术层面应对

10.2.3 业务层面应对

11. 风险架构的未来趋势

11.1 技术趋势

11.1.1 人工智能与机器学习

11.1.2 大数据与实时分析

11.1.3 云原生与微服务

11.2 业务趋势

11.2.1 数字化转型

11.2.2 生态化协同

11.3 监管趋势

11.3.1 监管科技（RegTech）

11.3.2 监管沙盒

12. 风险架构的评估与认证

12.1 评估方法

12.1.1 自评估

12.1.2 第三方评估

12.1.3 监管检查

12.2 认证与标准

12.2.1 ISO 31000

12.2.2 COSO ERM

12.2.3 巴塞尔协议

13. 风险架构的案例分析

13.1 成功案例

13.1.1 案例一：某金融机构的风险架构建设

13.1.2 案例二：某互联网企业的风险架构建设

13.2 失败案例

13.2.1 案例一：某企业的风险架构失败

13.2.2 案例二：某企业的风险架构失败

13.3 经验教训

13.3.1 成功因素

13.3.2 失败原因

14. 风险架构的实施建议

14.1 实施原则

14.1.1 循序渐进

14.1.2 业务驱动

14.1.3 持续改进

14.2 实施步骤

14.2.1 准备阶段

14.2.2 试点阶段

14.2.3 推广阶段

14.2.4 优化阶段

14.3 实施建议

14.3.1 对企业的建议

14.3.2 对风险管理者的建议

15. 风险架构的附录

15.1 术语表

15.2 参考资料

15.3 模板与工具

15.4 常见问题解答（FAQ）

16. 风险架构的总结与展望

16.1 总结

16.2 展望

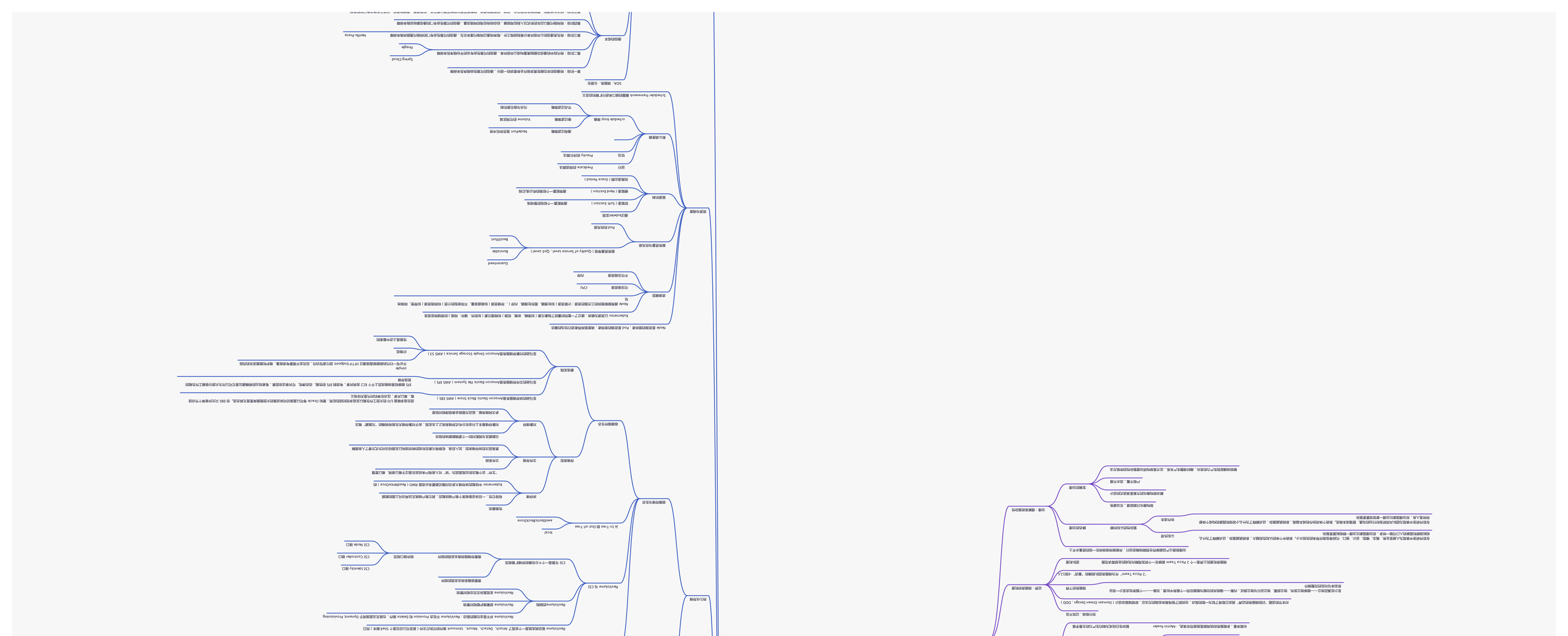
16.2.1 短期展望

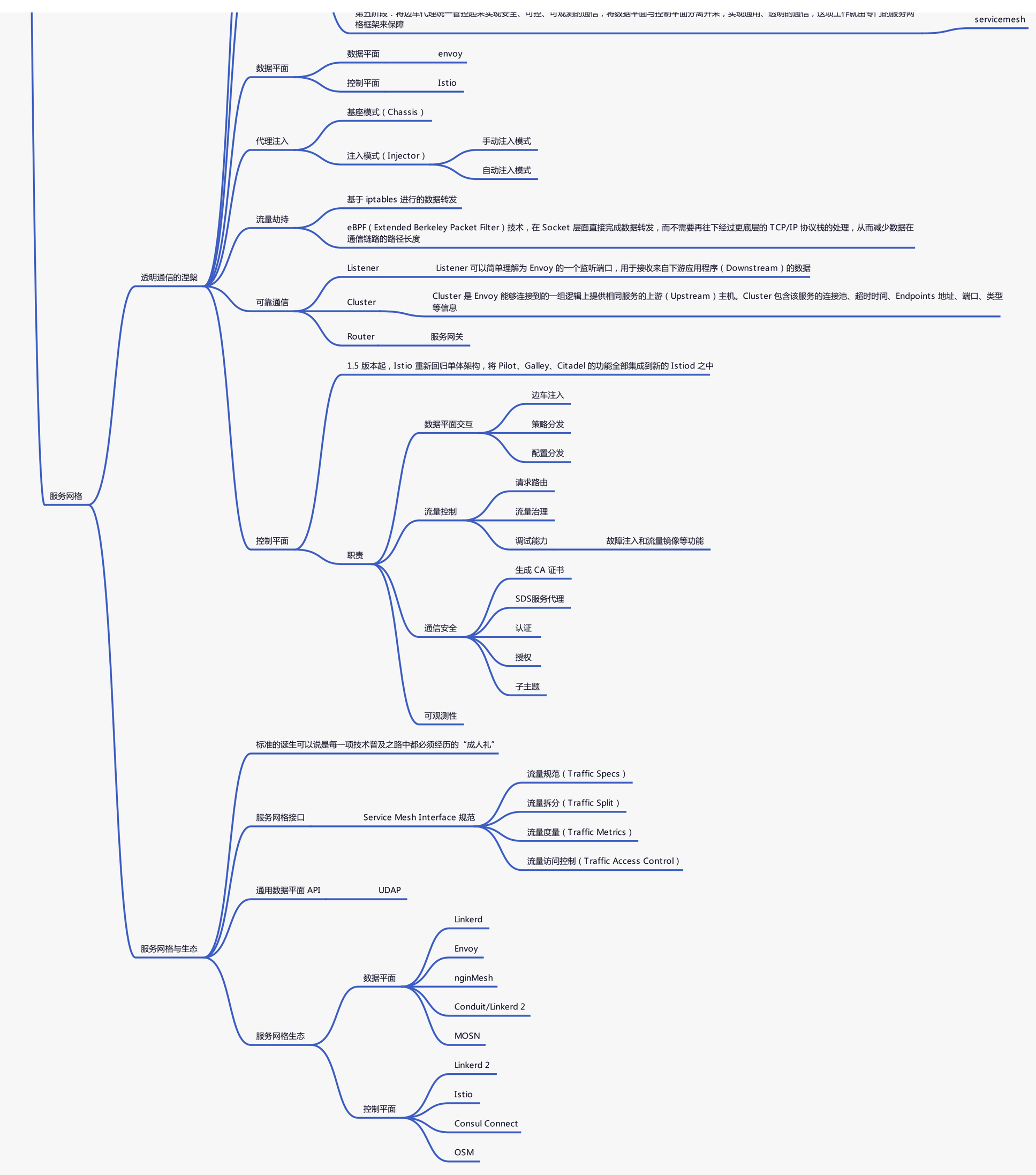
16.2.2 长期展望

16.3 行动号召

16.3.1 对读者的建议

16.3.2 对组织的建议




格框架来保障
servicemesh
服务网格
透明通信的涅槃
数据平面
数据平面
envoy
控制平面
Istio
代理注入
基座模式（Chassis）
注入模式（Injector）
手动注入模式
自动注入模式
流量劫持
基于 iptables 进行的数据转发
eBPF（Extended Berkeley Packet Filter）技术，在 Socket 层面直接完成数据转发，而不需要再往下经过更底层的 TCP/IP 协议栈的处理，从而减少数据在通信链路的路径长度
可靠通信
Listener
Listener 可以简单理解为 Envoy 的一个监听端口，用于接收来自下游应用程序（Downstream）的数据
Cluster
Cluster 是 Envoy 能够连接到的一组逻辑上提供相同服务的上游（Upstream）主机。Cluster 包含该服务的连接池、超时时间、Endpoints 地址、端口、类型等信息
Router
服务网关
控制平面
1.5 版本起，Istio 重新回归单体架构，将 Pilot、Galley、Citadel 的功能全部集成到新的 Istiod 之中
职责
数据平面交互
边车注入
策略分发
配置分发
流量控制
请求路由
流量治理
调试能力
故障注入和流量镜像等功能
通信安全
生成 CA 证书
SDS服务代理
认证
授权
子主题
可观测性
服务网格与生态
标准的诞生可以说是每一项技术普及之路中都必须经历的“成人礼”
服务网格接口
Service Mesh Interface 规范
流量规范（Traffic Specs）
流量拆分（Traffic Split）
流量度量（Traffic Metrics）
流量访问控制（Traffic Access Control）
通用数据平面 API
UDAP
服务网格生态
数据平面
Linkerd
Envoy
nginMesh
Conduit/Linkerd 2
MOSN
控制平面
Linkerd 2
Istio
Consul Connect
OSM