- ZK
  - 事务
    - 单机事务
      - ACID
        - 为了保证前后一致性、并发一致性，引入了原子性和隔离性
      - 实现
        - mysql的锁+MVCC
        - redis的单线程队列等
    - 分布式事务
      - 相关理论
        - **看起来redis的复制，mysql的主从是依据BASE理论**
        - CAP
          - 在分布式的情况下，很难做到ACID，所以引入CAP
          - C一致性（在发生脑裂的情况下，几个分区仍旧可用，那么孤岛分区之间无法保持一致）
          - A可用性（在发生脑裂的情况下，如果要保持几个孤岛分区一致，那么肯定要牺牲某些分区的可用性，将其shut down）
          - P分区容忍性（如果既要一直保持可用，也要一直保持一致，那么只能单机服务，不能做到分布式，这明显不可能）
        - BASE
          - 在分布式的情况下，可用性往往高于一致性，所以又引入BASE
          - Basically Available基本可用（允许部分不可用）
          - Soft State软状态（允许中间状态，不同副本之间有一定程度不一致）
          - Eventual Consistency最终一致性（一段时间后肯定一致）
      - 传统强一致性
        - XA规范-两阶段提交
          - 会一直持有锁（保证强一致性）
          - 强依赖于事务管理器和资源管理器（容易有单点故障）
          - 第一阶段让各RM执行事务命令，但先不要commit
          - 等到第二阶段时再统一commit
          - 任何一个节点失败都会导致事务失败（无失败容错）
        - 三阶段提交
          - 增加超时机制（默认commit，如abort超时也默认commit，那就完了）
          - 第一阶段RM没及时回复，则TM主动发起超时回滚（确保从RM到TM是通的）
          - 第二阶段RM没及时回复，则TM主动发起超时回滚
          - 第二阶段TM没及时发送commit，则RM主动发起超时回滚（确保TM到RM是通的）
        - paxos协议
          - 解决无限等待问题
          - 解决脑裂问题
          - 分布式锁实现：Chubby（ZK是它的开源版本，我说怎么那么像呢…）
        - java的JTA事务（要求RM都必须实现XA规范，比如mysql，某些mq）
      - 柔性事务
        - TCC两阶段补偿
          - 不是两阶段提交（是两阶段提交在业务层级的一种实现）
          - try-confirm-cancel（怎么那么像订单对接的几大步骤？）
          - 核心是try，confirm和cancel只不过是进一步执行或取消try而已
          - 业务检查和业务资源预留均在try中，confirm是使用try是预留的资源
          - 第二阶段confirm中使用单独的ACID，来补偿try中单独ACID造成的影响（相比之下，两阶段提交是一个超大超长的ACID事务）…
        - 最大努力通知型
          - 基本上是重复回调 + 推拉结合
          - 订单对接基本上也属于最大努力型
        - 本地消息表
          - 实现了最终一致性
          - 核心思想是要将分布式事务拆成本地事务
          - 和使用MQ来保证DB缓存最终一致性有点像
          - 消息可以从A发到B，也可以从B发到A（用于confirm或cancle）
        - 可靠消息最终一致性（少见）
          - 需要MQ本身支持预发送和确认发送
          - 本质上是要MQ也提供类似提交和回滚的能力
          - 阿里的RocketMQ支持，kafka和RabbitMQ不支持
  - ZK
    - 我自己理解的zk
      - 是主从绝对一致的分布式数据系统
      - 必不可少的步骤:选leader、分别同步数据、leader发布指令
      - 事务性能瓶颈在leader，且数据变更有延时（比如增加个node不会实时生效），大部分指令是增删改查
      - 拥有发布/订阅能力（RPC主要靠它）
    - 拥有的特性
      - 顺序一致性
        - 同一个客户端发送的请求，最终肯定顺序地实现在ZK上
        - 这是串行化的表现啊，基于它可以实现并发锁之类的东西吧
          - 使用队列和递增zxid实现
      - 原子性
        - 整个ZK集群要么都实现了客户端的请求，要么都没实现
          - leader统一管理
          - 所以有时并不能读取最新的状态吧，还没完成同步到整个集群（有延迟）
      - 单一视图
        - 不管**客户端连接哪一个zk节点，看到的内容都是相同的**…这点应该是靠原子性保障
    - 设计目标
      - 数据模型简单
        - 基于内存的目录树，有一系列znode
        - 和文件系统很类似
      - 集群化 && 高可用
        - 好像只有leader才能写，其他节点只能读?
          - **Follower只能将事务请求发送给leader**
        - 其中仅有一个master，客户端可以连接至任意的节点
      - 高性能
        - 数据均在内存中，据说3台机器有13万QPS？堪比redis（非事务请求才是这样，事务请求应该没这么多）
          - **内存读写本身就很快吧**
      - 顺序访问
        - 如果A修改在B之前，那所有服务器的A修改都在B之前（每个follower都有单独队列）
    - ZAB协议（zk原子广播协议）
      - 核心
        - 所有请求由全局唯一的leader处理，后者向follower发起事务提议，超过半数follower同意时再全部commit
          - 如果有节点不同意leader的发起的提议怎么办？似乎要该节点放弃leader？（对的，不同意提议，则抛弃leader，舍弃投票权，进入数据同步阶段）
          - 相比之下，两阶段会rollback
        - 怎么有点像两阶段提交?
          - **两阶段也保证强一致性，这里可以关注zk怎么解决两阶段的缺陷（用抛弃代替rollback，用事务日志和队列解决commit失败）**
      - 崩溃恢复模式
        - 用于重新选leader、初始化集群、新增follower（复制leader的数据）
        - Leader**提交**某提议后崩溃，恢复后其它节点必须提交该提议
          - 拥有最新提议的服务器当选leader，它拥有原leader已提交的提议（原leader已经让超半数follower记下事务日志，并让新leader commit，新leader才能拥有最新的提议）
        - Leader**提出**某提议但未提交，恢复后其它节点必须废弃该提议
          - 旧leader未提交提议，若它恢复，则新leader会让它丢弃新leader
        - 上述2点有点像mysql的redo和undo?
          - **放到zk来看，只需要选出持有最大zxid提议的节点就好**
          - 怎么样才知道leader挂了?
          - 我猜测是有首位发现者，将纪元+1，询问其他人的意见
          - 选出新leader之后，新leader要让所有follower都跟它同步后，才能对外提供服务
      - 消息广播模式
        - Leader用类似两阶段的协议向follower发出提议（用全局单调递增的zxid表示），不需要像两阶段协议那样等所有人回复，只**需超过半数就开始提交**
          - zxid为64位，前32位用来记录纪元（新选leader则纪元+1），后32位给该leader生成自增zxid
          - 全局单调是由每一届的leader来保证的（注定事务性能高不了）
          - 怎么保证广播收到yes的回复后，在commit阶段也一定成功呢？莫非是串行化?
          - 已经记录到事务日志中，串行化修改，后续修改内存肯定成功
        - Leader为每一位follower都创建一个队列，用于存放新提议
    - 典型应用
      - 数据发布订阅（美团Lion原理）
        - 借助zk的watcher机制
        - 推拉结合：zk仅告知事件发生，最新数据需客户端主动获取
          - 不过，怎么保证客户端都在同一时间拿到最新数据呢？各个客户端的网络延迟都不一样啊
          - 这个保证不了，除非zk给变更加上一个”最终生效时间”
      - 负载均衡
        - 其实只是利用zk来存储一个域名对应的多个ip而已
        - 具体的分发算法要么由后台决定，要么由客户端自己决定
      - 命名服务（全局唯一ID生成）
        - 本质上是利用了顺序znode的特性
        - Zk怎么实现顺序znode？我猜测是leader自身在内存中维护了自增ID
      - Master选举（美团redis多机房）
        - 本质上是争抢某个临时节点
      - 分布式锁
        - 排他锁（本质上是争抢某个临时节点，抢不到则排队）
        - 共享锁（本质上是读写所有请求都借助临时顺序节点排队）
          - 读请求必须确保自己之前没有写请求
          - 写请求必须确保前面没有任何请求
          - 羊群效应：每次有人释放锁，所有人都要出来判断一遍（可以只监听之前一个请求）
      - 分布式队列
        - 本质上是分布式写锁，所有节点都得一一排队
        - 如果要实现栅栏式队列，等待人齐再action，则可以将数量放在父节点
      - Hadoop的HA
        - 主备切换，克服单点故障，类似master选举
        - 脑裂处理：假死RM恢复后，不再有权限修改zk节点（不是它争抢的节点）
      - Kafka的应用
        - topic分区
          - 每个topic都分成N个分区，topic涉及的消息尽可能均匀地存放在各个分区
          - **每个分区只能被同个消费组中的一个消费者消费（避免重复消费）**
          - 如果分区数量少于消费者，会出现“争抢不到分区”
          - 分区的存在，主要是使得集群分流吧，提供并发能力
          - 美团默认会分成几个区？18个，4个broker
        - broker
          - kafka负责存储消息的机器
          - 所有broker都会主动申请并存储topic的部分分区（同时有可能会**保存该topic其他分区的副本**）
        - 消费者负载均衡
          - 若有4个分区和2个消费者，则前2个分区分给消费者A，后俩给B
        - 生产者负载均衡
          - 生产者可以向任意的分区发消息咯？
          - 似乎是轮询机制
        - 推拉结合，producer推，consumer拉
      - 大部分应用都基于几个特性：争抢临时节点、顺序节点、监视器
    - 重点技术内幕
      - 内部数据模型
        - 一个znode树，每个节点上记录自己的创建zxid等信息。
        - 临时节点只能作为叶子节点，顺序节点的编号由父节点维护。
        - 每个节点还会记录自己的版本号（用于实现原子性：版本+1）
      - 通信协议
        - 在tcp的基础上开发自己的协议，比较简单
      - Leader选举
        - 启动时选举概述
          - 每台机器都向外发送（myId，zxid），其中myId是指服务器id，zxid是每台服务器的起始事务Id，一般都从0开始
          - 先挑选zxid最新的服务器，如果zxid都一样，则选myId最大的服务器，再进行下一轮投票
          - 只要得到超过半数(n/2 + 1)服务器的投票，即可当选leader（如果超过一半意见一致，那么这个事实就不会改变）
        - 运行时选举概述
          - 类似启动时的选举
          - 问题是：怎么知道leader挂了？靠投票得出结论么（若启动新一轮投票，纪元会+1）
      - 事务日志
        - 事务落地之前必须写入日志文件（默认预分配64M，满则开辟新文件），将事务请求逐行记录在日志中
          - **zookeeper瓶颈也在磁盘速度**
        - 若follower的zxid大于leader，则follower必须自行截断大的zxid日志，和leader保持一致

邱永臣
（www.iloveqyc.com）