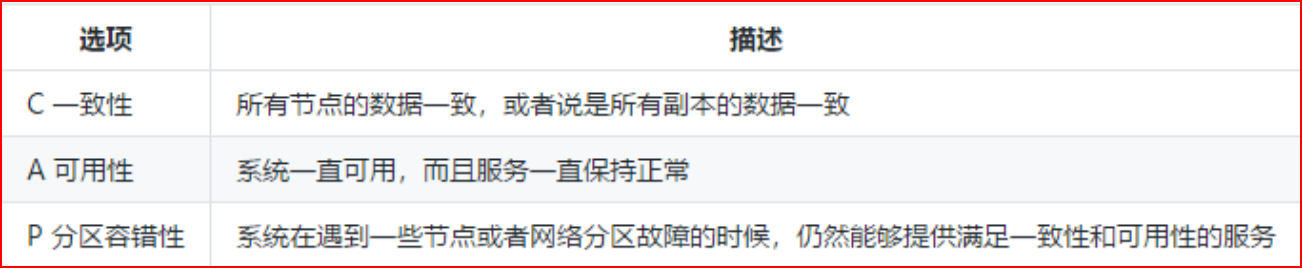
**分布式**

▶ **CAP**



◇**为什么CAP不能同时满足**

•舍弃可用性，保留一致性和分区容错性：也就是说允许在极端情况下系统无法访问情况出现， 牺牲了用户体验。-- 集群

•舍弃一致性，保留可用性和分区容错性：大部分的分布式系统的设计，保证高可用和分区容错， 但是牺牲一致性。-- 集群+主从

•舍弃分区容错，保留一致性和可用性：相当于舍弃分布式系统。--主从

◇**ACID模型**

•Atomicity 原子性

•Consistency 一致性

•Isolation隔离性

•Durability持久性

◇**BASE模型**

BASE模型反ACID模型，完全不同ACID模型，牺牲高一致性，获得可用性或可性：

•Basically Available： 基本可用。支持分区失败(e.g. sharding碎片划分数据库)

•Soft state：软状态 状态可以有一段时间不同步，异步。

•Eventually consistent：最终一致，最终数据是一致的就可以了，而不是时时高一致。

▶**分布式事务**

◇**二阶段提交**

•准备阶段（Prepare phase）

事务协调者给每个参与者发送Prepare消息，每个数据库参与者在本地执行事务，并写本 地的Undo/Redo日志，此时事务没有提交。（Undo日志是记录修改前的数据，用于数据 库回滚，Redo日志是记录修改后的数据，用于提交事务后写入数据文件）

•提交阶段（commit phase）

如果事务协调者收到了参与者的执行失败或者超时消息时，直接给每个参与者 发送回滚 (Rollback)消息；否则，发送提交(Commit)消息；参与者根据事务管理器的指令执行提交或 者回滚操作，并释放事务处理过程中使用的锁资源。注意:必须在最后阶段释放锁资源

◇**三阶段提交**

•CanCommit 阶段

协调者向参与者发送 canCommit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

•PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以继续进行，有以下两种可能。假如协调者从 所有的参与者获得Yes响应，执行事务的预执行。假如有其中一个参与者向协调者发送了 No响应，或者等待超时，协调者都没有接到参与者的响应，那么执行事务的中断。

•doCommit 阶段

该阶段进行真正的事务提交， 主要包含 1.协调者发送提交请求 2.参与者提交事务 3.参与 者响应反馈 4.协调者确定完成事务。

◇**消息中间件**

发送端

可靠性

发送端完成操作后一定能将消息成功发送到消息系统。

实现方法

在本地数据建一张消息表，将消息数据与业务数据保存在同一数据库实例里，这样就 可以利用本地数据库的事务机制。事务提交成功后，将消息表中的消息转移到消息中 间件，若转移消息成功则删除消息表中的数据，否则继续重传。

消息系统（CP）

高可用

集群

主从

可靠性

持久化

接收端

可靠性

接收端仅且能够从消息中间件成功消费一次消息。

实现方法

保证接收端处理消息的业务逻辑具有幂等性； 保证消息具有唯一编号，并使用一张 日志表来记录已经消费的消息编号。

◇TCC 方案

•Try 阶段

完成所有业务检查（一致性） 预留必须业务资源（准隔离性）中间状态或者执行部分内容

•Confirm 阶段

真正执行业务 只使用Try阶段预留的业务资源 Confirm操作满足幂等性(失败重试)

•Cancel阶段

释放Try阶段预留的业务资源 Cancel操作满足幂等性(失败重试)

•订单服务和库存服务示例

○Try阶段

订单服务:修改订单的状态为支付中

账户服务:账户余额不变，可用余额减1，然后将1这个数字冻结在一个单独的字段里

库存服务:库存数量不变，可销售库存减1，然后将1这个数字冻结在一个单独的字里

○confirm阶段

订单服务:修改订单的状态为支付完成

账户服务:账户余额变为(当前值减冻结字段的值)，可用余额不变(Try阶段减过了),冻结 字段清0。

库存服务:库存变为(当前值减冻结字段的值)，可销售库存不变(Try阶段减过了)，冻结 字段清0。

○cancel阶段

订单服务:修改订单的状态为未支付

账户服务:账户余额不变，可用余额变为(当前值加冻结字段的值)，冻结字段清0。

库存服务:库存不变，可销售库存变为(当前值加冻结字段的值)，冻结字段清0。

◇定时器

将需要分布式处理的任务通过消息或者日志的方式来异步执行，失败重试，要求各服务的接口 是幂等的，全局表描述整个事务

◇Seata框架 -- AT

•@GlobalTransactional注解

•被这个注解包围的方法，执行流程

第一步：初始化一个全局事务的实例，分布式事务的唯一标识，称之为XID

第二步：开启全局事务，设置超时时间等属性，以避免拿不到锁时，无限的等待。

第三步：执行逻辑，各个服务执行各自的本地事务。如果任何一个服务的本地事务出 现异常，则进入回滚全局事务。如果回滚成功，就抛出那个本地事务失败的 异常。如果回滚全局事务失败，则抛出异常原因。将本地事务提交的结果上 报给事务协调者

第四步：标记该次全局事务状态为完成，通知各单体服务该次事务已经完成。各单体 服务收到该请求后，会立刻返回成功，然后异步删除之前本地事务前保存的 回滚信息。

•和XA区别

减少了XA两阶段提交的锁的时间，不依赖于数据库本身的回滚机制

•回滚时，发现老数据已经被其他的事务修改

如果 tx1 的二阶段全局回滚，则 tx1 需要重新获取该数据的本地锁，进行反向补偿 的更新操作，实现分支的回滚。

此时，如果 tx2 仍在等待该数据的 全局锁，同时持有本地锁，则 tx1 的分支回滚会 失败。分支的回滚会一直重试，直到 tx2 的 全局锁 等锁超时，放弃 全局锁 并回滚 本地事务释放本地锁，tx1 的分支回滚最终成功。

▶**一致性算法**

◇**Paxos**

•问题解决

像 2PC，3PC 一样引入一个协调者，谁先到，听谁的协调者宕机了呢？

所以需要对协调者也做备份，也要做集群。这么多协调者，听谁的呢？

•相关概念

Client：客户端 客户端向分布式系统发出请求，并等待响应。例如，对分布式文件服务器 中文件的写请求。

Proposer：提案发起者 提案者提倡客户请求，试图说服Acceptor对此达成一致，并在发生 冲突时充当协调者以推动协议向前发展

Acceptor：决策者，可以批准提案 Acceptor可以接受（accept）提案；如果某个提案被选 定（chosen），那么该提案里的value就被选定

Learners：最终决策的学习者 学习者充当该协议的复制因素

•算法描述

○**三个阶段**

✱**第一阶段**：Prepare阶段。Proposer向Acceptors发出Prepare请求，Acceptors针对 收到的Prepare请求进行Promise承诺。

◾Prepare: Proposer生成全局唯一且递增的Proposal ID (可使用时间戳加 Server ID)，向所有Acceptors发送Prepare请求，这里无需携带 提案内容，只携带Proposal ID即可。

◾Promise: Acceptors收到Prepare请求后，做出“两个承诺，一个应答”。

两个承诺

1. 不再接受Proposal ID小于等于（注意：这里是<= ）当前请求的 Prepare请求。

2. 不再接受Proposal ID小于（注意：这里是< ）当前请求的Propose 请求。

一个应答

不违背以前作出的承诺下，回复已经Accept过的提案中Proposal ID 最大的那个提案的Value和Proposal ID，没有则返回空值。

✱**第二阶段**：Accept阶段。Proposer收到多数Acceptors承诺的Promise后，向Acceptors 发出Propose请求，Acceptors针对收到的Propose请求进行Accept处理。

✱**第三阶段**：Learn阶段。Proposer在收到多数Acceptors的Accept之后，标志着本次 Accept成功，决议形成，将形成的决议发送给所有Learners。

○**Paxos算法流程**

◾获取一个Proposal ID n，为了保证Proposal ID唯一，可采用时间戳+Server ID生成；

◾Proposer向所有Acceptors广播Prepare(n)请求；

◾Acceptor比较n和minProposal，如果n>minProposal，minProposal=n，并且将 acceptedProposal 和 acceptedValue 返回；

◾Proposer接收到过半数回复后，如果发现有acceptedValue返回，将所有回复中 acceptedProposal最大的acceptedValue作为本次提案的value，否则可以任意决定 本次提案的value；

◾到这里可以进入第二阶段，广播Accept (n,value) 到所有节点；

◾Acceptor比较n和minProposal，如果n>=minProposal，则 acceptedProposal=minProposal=n，acceptedValue=value，本地持久化后，返回；否 则，返回minProposal。

◾提议者接收到过半数请求后，如果发现有返回值result >n，表示有更新的提议，跳 转到1；否则value达成一致。

○**如何保证Paxos算法的活性**

◾活性：最终一定会发生的事情：最终一定要选定value。

◾假设存在这样一种极端情况，有两个Proposer依次提出了一系列编号递增的提案， 导致最终陷入死循环，没有value被选定。

◾通过选取主Proposer，并规定只有主Proposer才能提出议案。这样一来只要主 Proposer和过半的Acceptor能够正常进行网络通信，那么但凡主Proposer提出一个 编号更高的提案，该提案终将会被批准，这样通过选择一个 主Proposer，整套Paxos 算法就能够保持活性

◇**Raft**

•领导人选举

○服务器都可以扮演角色

◾领导者(leader)：处理客户端交互，日志复制等动作，一般一次只有一个领导者

◾候选者(candidate)：候选者就是在选举过程中提名自己的实体，一旦选举成功，则 成为领导者

◾跟随者(follower)：类似选民，完全被动的角色，这样的服务器等待被通知投票

○Raft使用心跳机制来触发选举。当server启动时，初始状态都是follower。每一个server 都有一个定时器，超时时间为（一般为150-300ms），如果某server没有超时的情况下 收到来自领导者或者候选者的任何消息，定时器重启，如果超时，它就开始一次选举。

○初始状态下集群中的所有节点都处于follower状态。 某一时刻，其中的一个follower 由于没有收到leader的heartbeat率先election timeout进而发起选举. 只要集群中超过 半数的节点接受投票，candidate节点将成为即切换leader状态。 成为leader节点之后， leader将定时向follower节点同步日志并发送heartbeat.

○Leader节点不可用

◾一般情况，leader节点定时发送heartbeat到follower节点。

◾由于某些异常导致leader不再发送heartbeat，或follower无法收到heartbeat， 当 某一follower发生election timeout时，其状态变更为candidate，并向其他follower 发起投票. 当超过半数的follower接受投票后，这一节点将成为新的leader，leader 的步讲数加1并开始向follower同 步日志。

◾当一段时间之后，如果之前的leader再次加入集群，则两个leader比较彼此的步进 数，步进数较低的leader将 切换自己的状态为follower。较早前leader中不一致的 日志将被清除，并与现有leader中的日志伯封寺一致。

•日志复制

○客户端的每一个请求都包含被复制状态机执行的指令。

○leader把这个指令作为一条新的日志条目添加到日志中，然后同步follower。

○follower响应ACK,如果follower宕机或者运行缓慢或者丢包，leader会不断的重试，直到 所有的 follower最终都复制了所有的日志条目。

○通知所有的Follower提交日志，同时领导人提交这条日志到自己的状态机中，并返回给 客户端

•安全性

▶**分布式锁**

◇基于数据库

采用乐观锁增加版本号

1. 悲观锁

利用select … where … for update 排他锁

注意: 其他附加功能与实现一基本一致，这里需要注意的是“where name=lock ”，name字段 必须 要走 索引，否则会锁表。有些情况下，比如表不大，mysql优化器会不走这个索引， 导致锁表题。

2. 乐观锁

所谓乐观锁与前边最大区别在于基于CAS思想

◇基于Redis的分布式锁

1. 使用命令介绍：

（1）SETNX

SETNX key val：当且仅当key不存在时，set一个key为val的字符串，返回1；若key 存在，则什么都不做，返回0。

（2）expire

expire key timeout：为key设置一个超时时间，单位为second，超过这个时间锁会自 动释放，避免死锁。

（3）delete

delete key：删除key

在使用Redis实现分布式锁的时候，主要就会使用到这三个命令。

2. 实现思想：

（1）获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该 时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进 行判断。

（2）获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

（3）释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放

redis官方推荐redisson

◇基于zk分布式锁

利用有序临时节点特性。程序和zk中断，节点删除；释放锁，watch监听机制

实现细节

ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系统 目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的 步骤如下：

（1）创建一个目录mylock；

（2）线程A想获取锁就在mylock目录下创建临时顺序节点；

（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则 说明当前线程顺序号最小，获得锁；

（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；

（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节 点，如果是则获得锁。

这里推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的 InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。 ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系 统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁 的步骤如下：

（1）创建一个目录mylock；

（2）线程A想获取锁就在mylock目录下创建临时顺序节点；

（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则 说明当前线程顺序号最小，获得锁；

（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；

（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节 点，如果是则获得锁。

这里推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的 InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。

▶**分布式ID**

Snowflake 算法

如 Twitter 的 Snowflake 算法 1 bit位 是0 二进制代表正数 41 bit 是当前毫秒单位的一个时间 戳；

5 bit 是机房 id（但是最大只能是 32 以内），

5 bit 是机器 id（但是最大只能是 32 以内），

12 bit序列号，就是如果跟你上次生成 id 的时间还在一个毫秒内，那么会把顺序给你累 加，最多在 4096 个序号以内。

▶**分布式开发优缺点**

◇优点

模块解耦：把模块拆分,使⽤用接⼝口通信,降低模块之间的耦合度

项⽬目拆分，不不同团队负责不不同的⼦子项⽬目：把项⽬目拆分成若⼲干个⼦子项⽬目 ,不不 同的团队负责不不同的⼦子项⽬目

提⾼高项⽬扩展性：增加功能时只需要再增加⼀一个⼦子项目 ,调⽤用其他系统的接⼝口就以。

分布式部署：可以灵活的进⾏行行分布式部署

提⾼高代码的复⽤用性：比如service层,

◇缺点

系统之间的交互要使⽤用远程通信,接⼝口开发增⼤大⼯工作量量；

网络请求有延时；

事务处理理比较麻烦，需要使⽤用分布式事务。

▶**分布式一致性问题**

◇在大多数分布式环境中，尤其是涉及到数据存储的场景，数据一致性应该是首先被保证的，这也是 zookeeper设计成CP的原因。

◇同样也可以理解，像redis、rocketmq、kafka集群等等严格意义都是说与CP设计，涉及数据一致性 问题

◇euraka没有数据一致性的烦恼

▶**红黑树**

规则

每个结点或红或黑

根结点是黑色

空叶子结点是黑色

如果一个结点是红色，那么它的子节点是黑色

从根节点到叶节点或者到空子节点的每条路径，必须包含相同数目的黑色节点