Zookeeper

基础知识

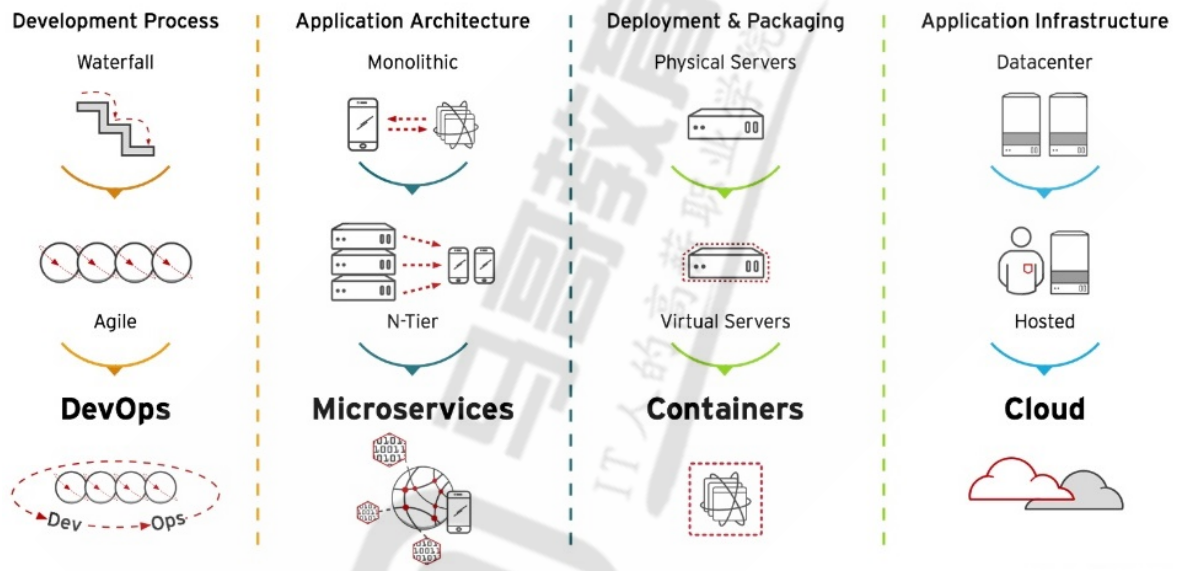
场景需求

学习目标

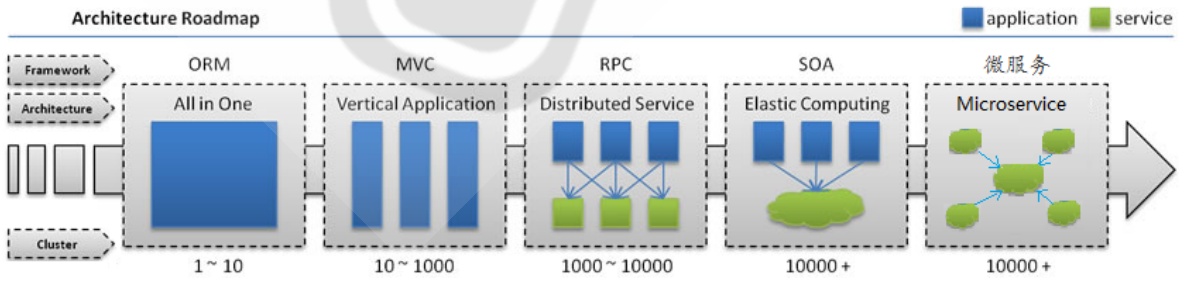
这一节，我们从 基础知识、分布式、小结 三个方面来学习

基础知识

技术变革

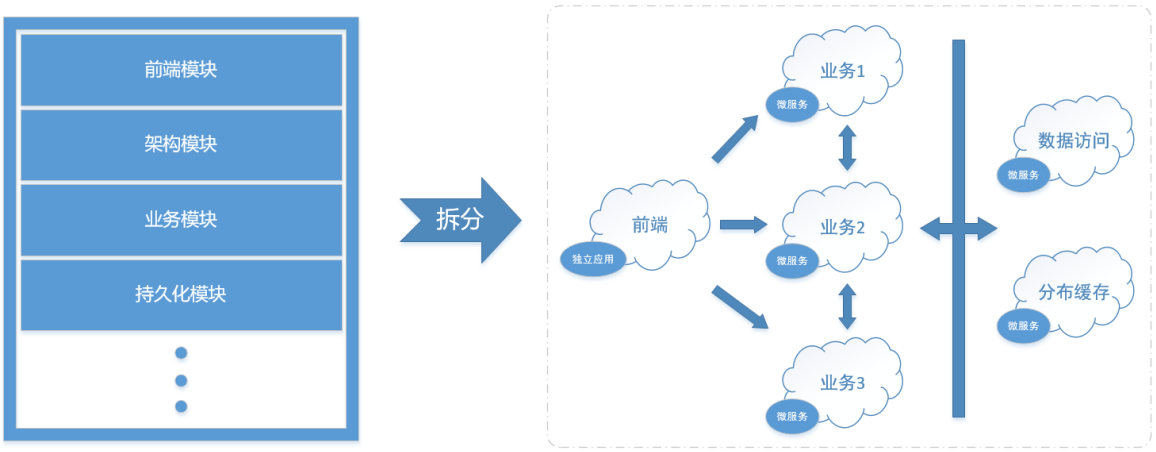


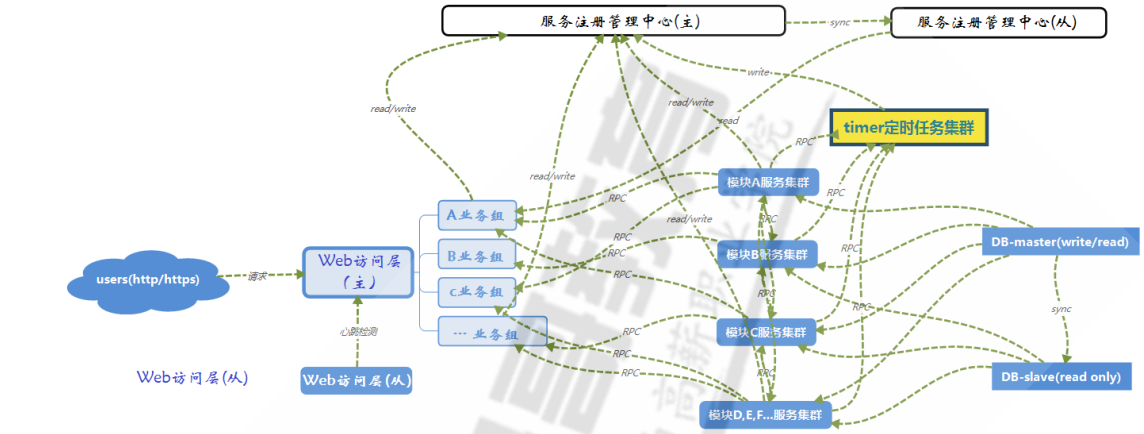
开发框架



ORM – 一台主机承载所有的业务应用
MVC – 多台主机分别承载业务应用的不同功能，通过简单的网络通信实现业务的正常访问
RPC – 应用业务拆分、多应用共用功能、核心业务功能 独立部署，基于远程过程调用技术(RPC)的分布式服务框架 提高业务功能复用及项目的整合
SOA – 粗放型的RPC分布式实现了大量的资源浪费，提高机器利用率的 资源调度和治理中心(SOA) ，基于现有资源的高效利用，进一步提高服务的能力
微服务 – 随着互联网的发展、各种技术的平台工具出现、编程语言的升级、开发规范的标准化等因素，中小型企业也有了相应的能力来发展更轻量级的SOA模式。

微服务





在微服务架构的场景中，有一个组件服务Service Registry,它是整个"微服务架构"中的核心，主要提供了四个功能：服务注册和服务发现、下线处理、健康检测等。

服务注册：

当服务启动后，将当前服务的相关配置信息都注册到一个公共的组件 -- Service Registry中。

服务发现：

当客户端调用操作某些已注册服务 或者 服务的新增或删除等，通过从Service Registry中读取这些服务配置的过程。

目前，Service Registry的最佳解决方案就是Zookeeper。这就是我们要学习Zookeeper的目的之一。

分布式

分布式系统

其实，除了ORM的部署场景之外，其他的几种模式，都是需要大量的主机服务组合在一起实现共同的业务功能，像这种多个服务彼此之间通过消息传递进行通信和协调的系统，我们都可以称之为 分布式系统 或者 分布式架构。

如果分布式系统内部多个功能，没有对数据通信、业务逻辑进行限制约束的情况下，可能会有如下常见问题：

问题	描述
随意分散	服务主机在空间上随意分布，主机角色随意变换
同等角色	服务主机之间没有主从角色之分，导致任意主机都处于同等地位，导致任意一台主机故障都影响全局。
并发请求	分布式系统的多个节点，可能会并发的操作一些共享的资源，例如数据库和分布式存储
资源抢占	服务主机的分散特性，缺乏统一管理，导致某一时刻或者某一时段导致资源冲突或抢占
故障发送	上面的各种因素导致各种故障层出不穷，而且无法快速定位

分布式特性

目前来说，随着互联网的发展，各种软件技术，尤其是设备计算能力的提升，所以很多企业在项目的开启就应用了 分布式架构。在分布式系统中各个节点之间的协作是通过高效网络进行消息数据传递，实现业务内部多个服务的通信和协调，基于服务本地设备的性能实现资源的高效利用。

分布式系统的设计目标通常包括几个方面：

可用性：

可用性是分布式系统的核心需求，衡量了一个分布式系统持续对外提供服务的能力。

可扩展性：

增加及其后不会改变或者极少改变系统行为，并且获得相似的线性的性能提升

容错性：

系统发生错误时，具有对错误进行规避以及从错误中恢复的能力

性能：

对外服务的响应延时和吞吐率要满足用户的需求

一致性协议

根据我们之前在 Redis部分的学习，我们为了满足分布式的各种场景需求，先后提出了 ACID、CAP、BASE等理论，其目的就是 在项目架构正常的运行过程中，即时出现各种问题，也能够保证业务保持基本可用的目标。

那么，我们在 项目架构在运行过程中 为了保证 业务保持基本可用 过程中定制的各种规约或者通信格式，都可以将其称为 一致性协议。

一般情况下，我们会基于 集群的方式实现分布式的 可用性、可扩展性、容错性等的目标，那么我们如何保证集群中的数据的一致性呢？

一致性模型

一般情况下，我们会基于 集群的方式实现分布式的 可用性、可扩展性、容错性等的目标，这个时候，集群中各个主机之间的通信信息是否一致的就非常重要了。

所谓的一致性是集群内部各个主机系统对外呈现的状态是否一致，即时业务出现问题的时候，这是所有的节点也要达成一个错误的共识。如果各个主机之间通信的数据不一致，就会导致各种分布式的场景问题。

在一个集群系统中，为了保证所有的主机系统能够处于一种相对的平衡状态，我们一般会基于传递数据本身和主机角色的方式来实现，所以我们可以从两个方面来进行分析：

数据本身：将所有的更新数据，同步到整个集群系统，保证数据的最终一致性。

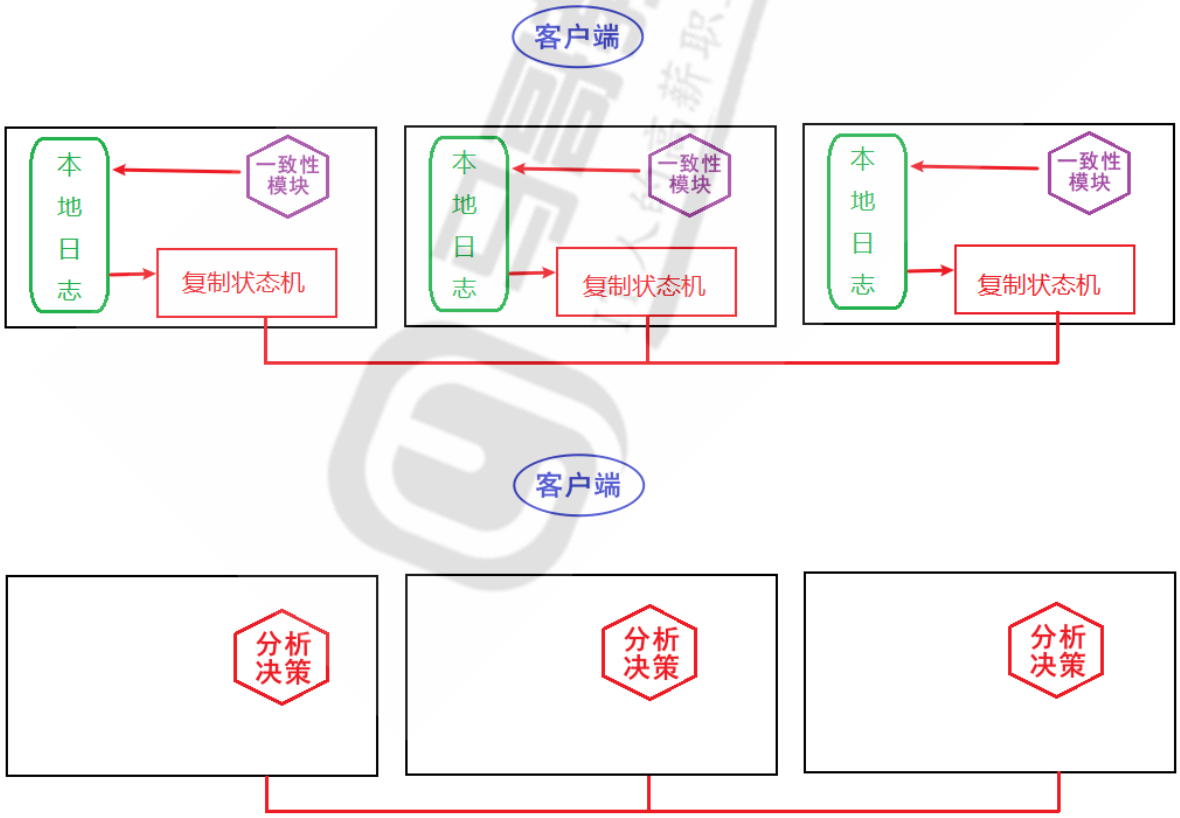
主机角色：client向多个server主机系统发起访问(包括并行访问)请求时，如何获取相同的更新后数据。

传递数据本身

分类	解析
强一致性(Strong Consistency)	通过对业务逻辑和数据顺序的控制，实现数据的读写完全是一致的，因为其在并行场景下的阻塞效果，所以在分布式场景中，实现的效果不是太好。
顺序一致性(Sequential Consistency)	服务端对于接收的客户端请求，将这些请求放到一个序列中，按照顺序来执行数据请求。 更新场景：5个客户端发起更新数据请求，服务端接到请求后，按照顺序将所有请求放到要给队列中，然后按照队列顺序，依次获取请求进行处理； 同步场景：服务端接收n个客户端发起的读取请求，因为受到本身的性能限制，所以划分一个队列，批量按照顺序，依次读取。
因果一致性(Causal Consistency)	一种特殊的顺序一致性，对于有事务性要求的多个请求，会自动通过其内部的机制，将这些请求进行顺序排列执行，因为这里涉及到事务性场景，所以对于非事务的操作请求，相对来说降低了一些标准。 比如：一个事务涉及到三个进程A-B-C，进程A在更新完数据，进程B基于进程A更新后的最新值进行操作，进程C基于进程A更新后的最新值进行操作，依次类推。与事务操作无关的进程X在操作的时候，无所谓。

从角色角度

分类	解析
状态复制机(State Machine Replication)	一个服务端集群，有多个server主机组成，每个server主机的更新都在本地实现。 每个服务端都有一个一致性模块来接收客户端请求，没接收一次用户请求，一致性模块的状态就发生改变，通过 状态机系统 对所有的一致性模块的状态进行管控，只要所有的模块状态是一样的，那么server主机本地执行后的最终数据值就是一样的，从而实现服务的容错效果。 GFS、HDFS、Chubby、ZooKeeper和etcd等分布式系统都是基于复制状态机模型实现的。
拜占庭将军问题(Byzantine Failures)	一个服务端集群，有多个server主机组成,每个server主机接收到client请求后，根据自己本身的特性进行分析并给出执行的策略，多个server主机通过专用的通讯方式来进行协商，并达成最终的共识结果(少数服从多数)，然后按照最终的结果进行操作执行，从而实现服务的容错效果。
FLP定理(Fischer, Lynch , Patterson)	三位科学家在1985年发表的分布式理论，最小化异步网络通信场景下，因为消息通信是延迟的，所以可能会出现 只有一个节点故障(没被其他节点发现)时，其他节点不能达成一致。这证明了在异步场景中永远无法避免的一种现象。 比如：三台主机ABC异步方式通信，在正常协商之间，因为C主机突然网络故障，导致无法实现剩余两台的少数服从多数，从而导致业务终止执行。



一致性协议

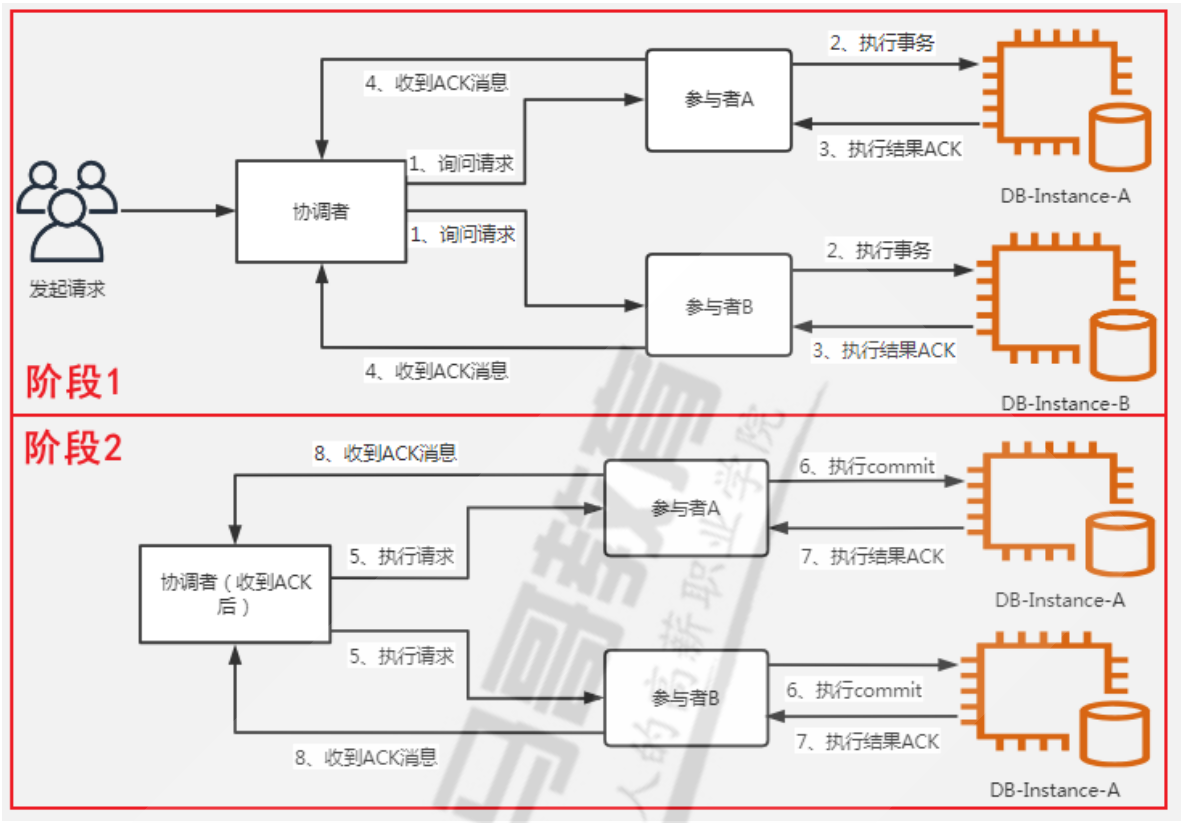
为了保证集群内部的多个节点间状态是一致的，无论是 状态复制机、专用通信方式、亦或是其他模型，必不可少的一个内容就是 如何实现多主机间的 状态机信息同步、专用通信等 -- 一致性协议。
常见的一致性协议主要有两种：

2PC：二阶段提交(Two-Phase-Commit)

事务的提交过程分为两个阶段来处理，提交事务请求和执行事务提交。

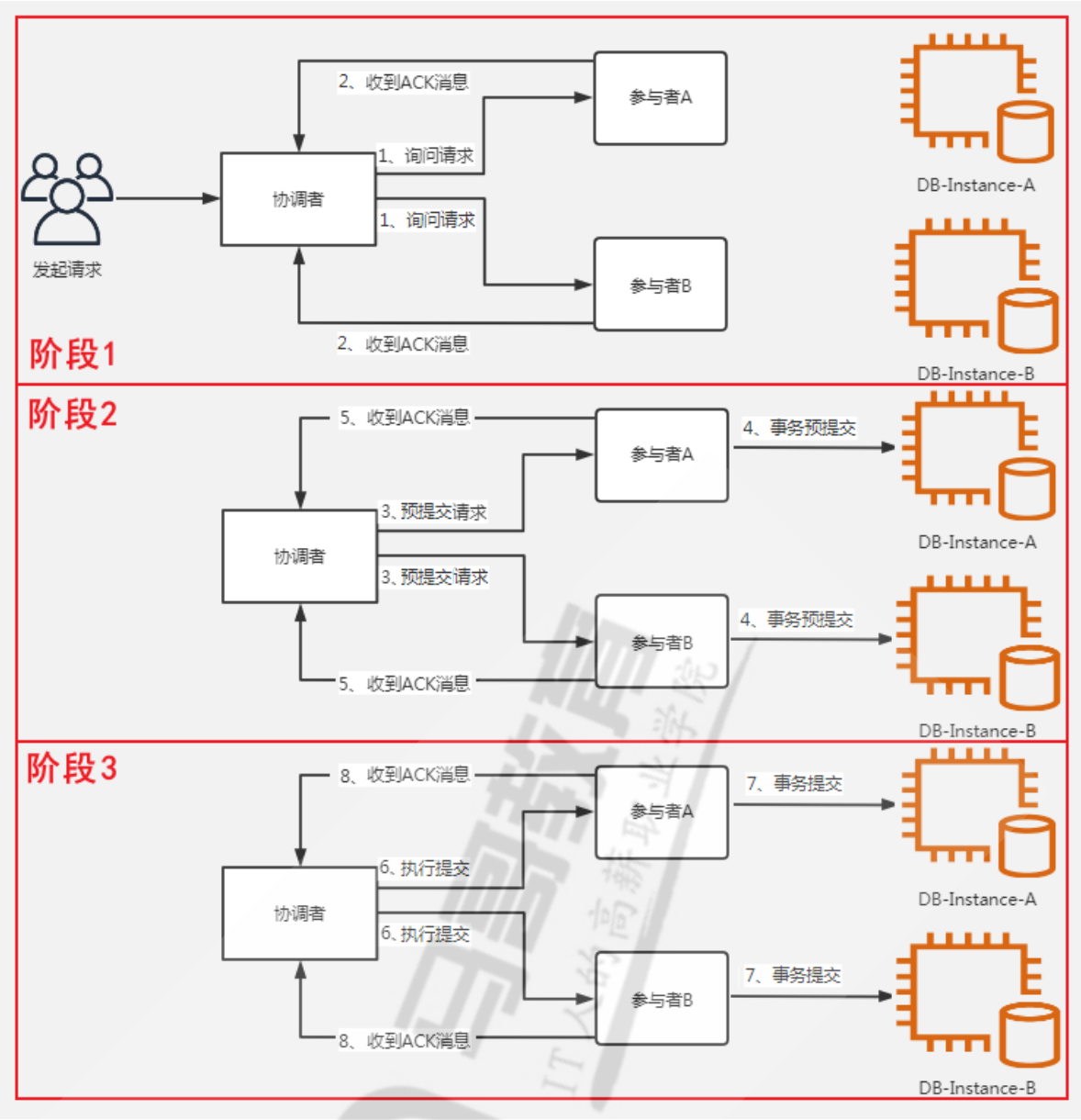
3PC：三阶段提交(Three-Phase-Commit)

事务的提交过程分为三个阶段来处理，提交事务申请、提交事务预操作、执行事务提交。



第二步的时候在本地记录相关日志，便于异常情况下的数据恢复

优点：简单方便
缺点：同步阻塞、单点问题、及协调者异常导致的其他数据不一致问题



第四步的时候在本地记录相关日志，便于异常情况下的数据恢复

优点：降低了参与者的阻塞范围，并且能够在出现单点故障后继续达成一致。
缺点：参与者在接收到预提交请求后，如果网络无法正常通信，可能导致异常情况下，依然执行事务

小结

Zookeeper简介

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

Zookeeper，英文字面意思就是"动物管理员"，因为动物园里面的所有动物的特殊性，需要管理员必须具备观察动物状态和管理动物行为等方面的协调的能力，为动物们建立友好生存的生活环境。Zookeeper就是纷乱的软件服务世界中的一名管理者，为繁杂的软件服务环境提供统一的协调管理服务。

Zookeeper是Yahoo基于 Google的 Chubby 论文实现的一款解决分布式数据一致性问题的开源实现，它是使用Java语言开发的，目前是Hadoop项目中的一个子项目。它在Hadoop、HBase、Kafka、Dubbo等技术中充当了非常重要的核心组件角色。

官方网站：https://zookeeper.apache.org/
最新版本：3.7.0

What is ZooKeeper?

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

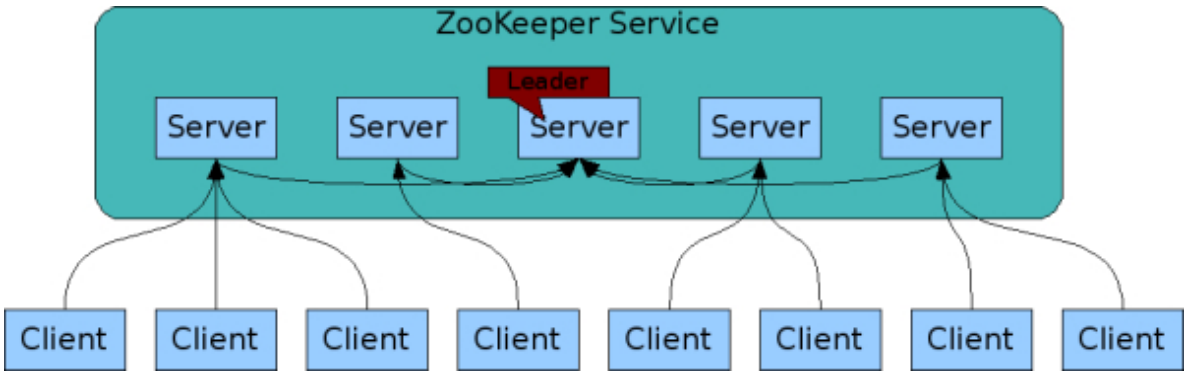
ZooKeeper is a centralized[集中] service for maintaining[维护] configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form[形式] or another by distributed applications. Each time they are implemented[部署、实施] there is a lot of work that goes into fixing the bugs and race[竞争] conditions that are inevitable[必然]. Because of the difficulty of implementing these kinds of services, applications initially[最初] usually skimp[吝啬] on them ,which make them brittle[脆弱] in the presence[场景] of change and difficult to manage. Even when done correctly[正确的], different implementations of these services lead to management complexity when the applications are deployed.

一句话：ZooKeeper是复杂分布式场景下的应用程序协调服务，监控并协助管理所有资源，提高程序系统稳定性。

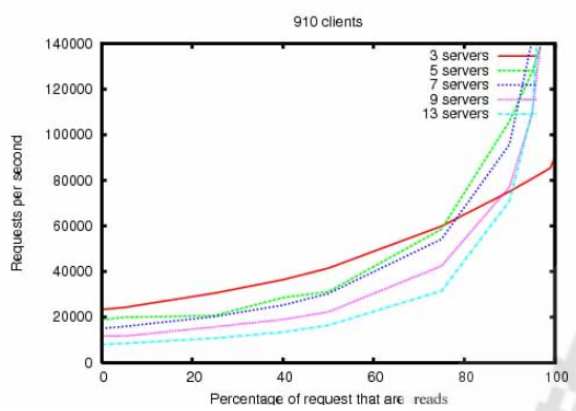
设计目标

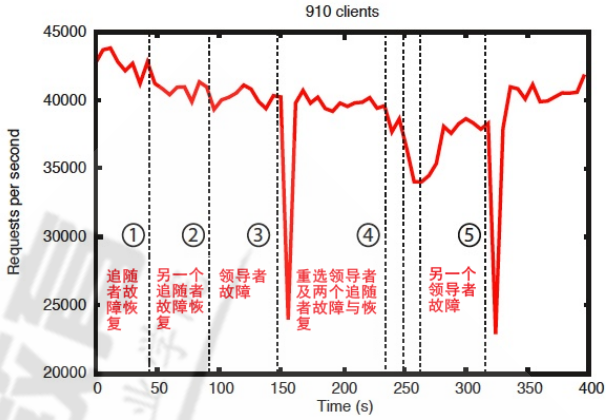
Zookeeper的设计目标就是在复杂且容易出错的分布式服务进行封装，以统一的简单易用接口形式，给用户提供一个稳定且高效可靠的服务。

Zookeeper作为一个典型的分布式数据一致性解决方案，依赖Zookeeper的分布式应用程序，可以基于Zookeeper实现数据发布/订阅、负载均衡、命名服务、服务注册与发现、分布式协调/事件通知、集群管理、Leader 选举、 分布式锁和队列 等功能



性能效果





特性解析

简介

在生产中，Zookeeper一般以集群的方式对外提供服务，一个Zookeeper集群包含多个Zookeeper服务主机节点，所有的节点环境一致，共同对外提供服务。
整个集群对分布式数据一致性提供了全面的支持，具体包括以下两类共计五个特性：

服务相关

顺序性：
　　为了保证同一个客户端发出的多个请求按顺序进行处理，那么就必须有这个一个功能，让所有请求按顺序的方式进入到Zookeeper中，并且以一个队列的形式存在，结合队列的"先进先出"的原则，实现请求处理的顺序性。

原子性：
　　Zookeeper集群中所有的主机，对所有客户端请求的处理结果应该只有两种状态：成功/失败，不能出现一部分处理成功，另一部分处理失败。

可靠性：
　　Zookeeper集群任意一台服务器都可接收信息请求，而且一旦服务端数据接收数据修改请求，其数据状态一定发生变化。一旦数据状态发生变化，就会被存储起来，从而变保证集群中的所有数据都是可靠的

数据相关

一致性：

对于客户端来说，无论请求Zookeeper集群中的哪一个节点，看到效果和处理的结构都是完全一样的。那么这点就要求我们的Zookeeper集群节点间必须做好高效的数据同步功能。

实效性：

当请求被成功的处理后，客户端能立即获取服务端的最新数据变化。

由于网络延时等原因，Zookeeper保证客户端在一个时间范围内获得集群主机节点的更新/失效信息，如果需要最新数据，读数据前同步一下即可。

顺序性实现

顺序性主要有两种：全局有序和偏序。

全局有序：如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；

偏序：如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

基本概念

角色

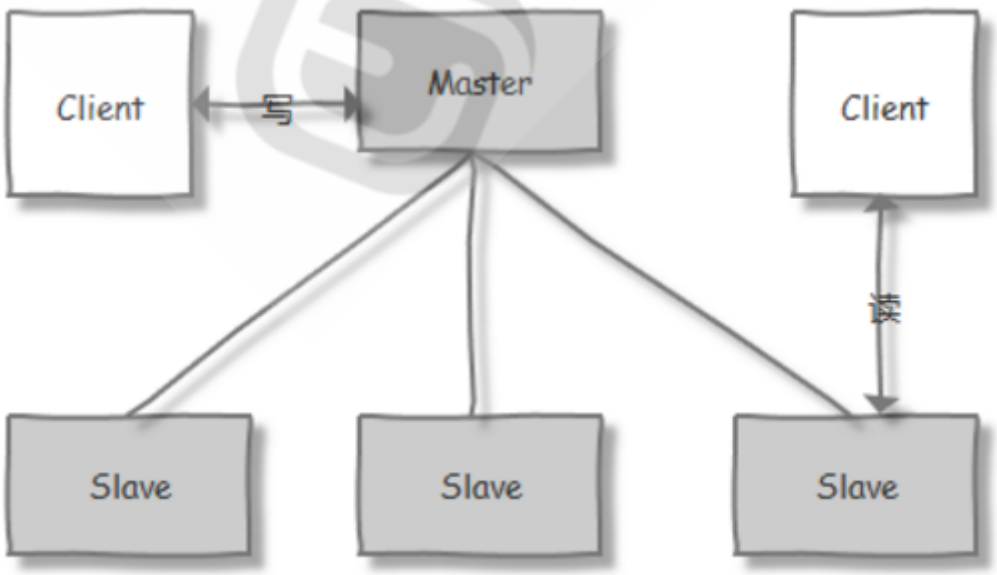
学习目标：

这一节，我们从集群模式、角色解析、小结 三个方面来学习。

集群模式

简介

基本上所有的集群模式中的主机都有自己的角色，最为典型的集群模式就是 M/S 主备模式。在这种模式下，我们把处于主要地位(处理写操作)的主机称为 Master 节点，处于次要地位(处理读操作)的主机称为 Slave 节点，生产中读取的方式一般是以异步复制方式来实现的。



Zookeeper集群就是这种M/S的模型，集群通常由2n+1台Server节点组成，每个Server都知道彼此的存在。对于2n+1台server，只要有>=(n+1)台server节点可用，整个Zookeeper系统保持可用。

角色划分

只不过在它内部进行了业务场景下的更细的业务场景角色划分：

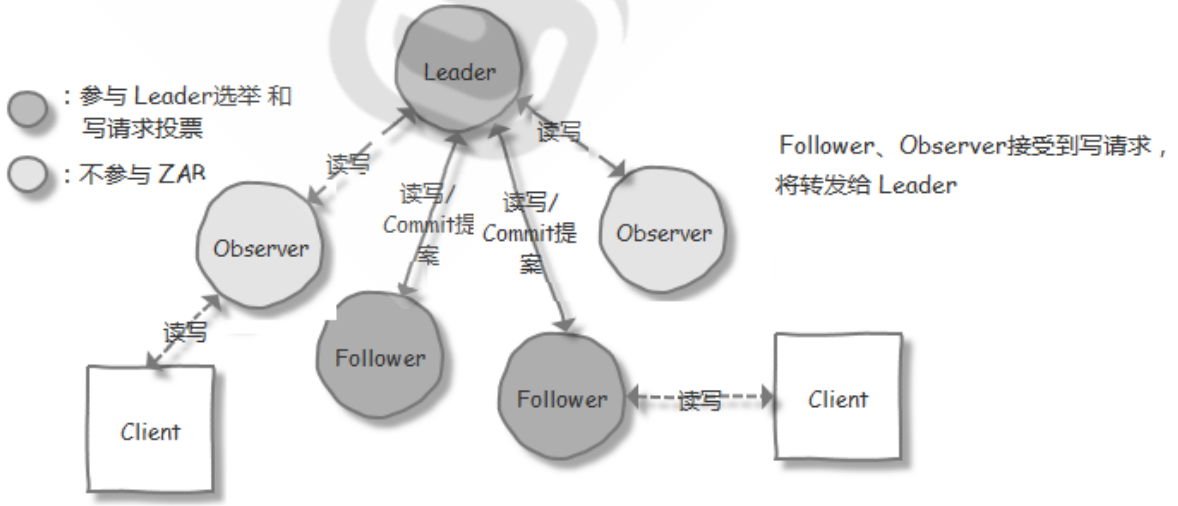
序号	角色	责任描述
1	领导者(Leader)	领导者不接受client读请求，负责进行投票发起和决议，更新系统状态
2	跟随者(Follower)	接收客户请求并向客户端返回结果，在选Leader过程中参与投票
3	观察者(Observer)	转交客户端写请求给leader节点，和同步leader状态，不参与选主投票
4	学习者(Learner)	和leader进行状态同步的节点统称Learner，Follower和Observer都是
5	客户端(client)	请求发起方

注意：角色是Zookeeper内部的业务场景角色，不是集群节点职责角色。

角色解析

简介

Zookeeper集群系统启动时，集群中的主机会选举出一台主机为Leader，其它的就作为Learner(包括Follower和Observer)。接着由follower来服务client的请求，对于不改变系统一致性状态的读操作，由follower的本地内存数据库直接给client返回结果；对于会改变Zookeeper系统状态的更新操作，则交由Leader进行提议投票，超过半数通过后返回将结果给client。



Zookeeper是采用ZAB协议（Zookeeper Atomic Broadcast，Zookeeper原子广播协议）来保证主从节点数据一致性的，ZAB协议支持「崩溃恢复和消息广播」两种模式，很好解决了这两个问题：

崩溃恢复：

Leader挂了，进入该模式，选一个新的leader出来，

接着，新的Leader服务器与集群中Follower服务进行数据同步，

当集群中超过半数机器与该 Leader服务器完成数据同步之后，退出恢复模式进入消息广播模式。

消息广播：

把更新的数据，从Leader同步到所有Follower

Leader 服务器开始接收客户端的事务请求生成事务Proposal进行事务请求处理。

主机状态

Zookeeper集群中每个Server主机在工作过程中有四种状态：

Looking(迷茫者)：我老大是谁？

当前Server主机不知道集群中的Leader是谁，陷入深深的不安，正在搜寻主心骨。

Leading(领导者)：我是总统。

当前Server主机被集群选举策略确定的Leader，我的地盘我做主。

Following(执行者)：我是员工。

集群的Leader主机已经选举出来，当前Server主机与之同步信息，严格执行规章制度，保质保量完成任务。

Observing(观望者)：我是看客

observer会观察leader是否有改变，然后同步leader的状态，是系统扩展的一种方法

事务id

所谓的事务id -- zxid。ZooKeeper的在选举时通过比较各结点的zxid和机器ID选出新的主结点的。zxid由Leader节点生成，有新写入事件时，Leader生成新zxid并随提案一起广播，每个结点本地都保存了当前最近一次事务的zxid，zxid是递增的，所以谁的zxid越大，就表示谁的数据是最新的。

ZXID有两部分组成：

任期：完成本次选举后，直到下次选举前，由同一Leader负责协调写入；

事务计数器：单调递增，每生效一次写入，计数器加一。

--同一任期内，ZXID是连续的，每个结点又都保存着自身最新生效的ZXID，通过对比新提案的ZXID与自身最新ZXID是否相差“1”，来保证事务严格按照顺序生效的。

小结

会话

学习目标：

这一节，我们从 基础知识、流程解析、小结这三个方面来学习。

基础知识

会话

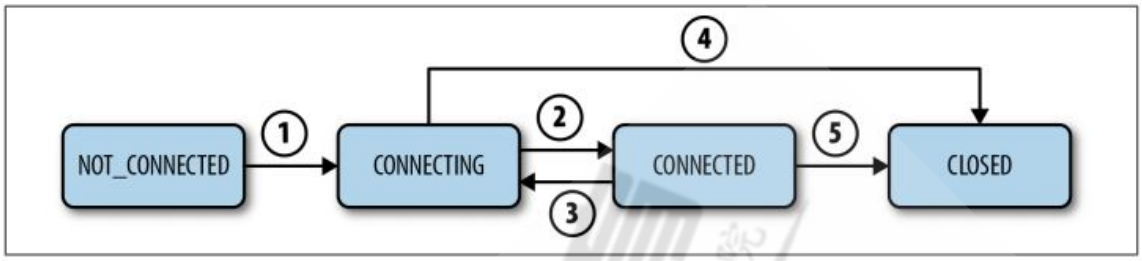
会话(Session)指客户端启动时候和Zookeeper集群主机节点之间创建的 TCP 长连接。

既然是会话，那么就必须说该会话的生命周期：

会话是从客户端和服务端第一次连接建立开始算起。通过这个连接，客户端能够通过心跳检测与服务节点保持有效的会话，并向 Zookeeper 服务器发送请求并接收响应，以及接收来自服务端的监听事件通知

会话状态

客户端和Zookeeper集群主机建立连接，整个session状态变化如图所示



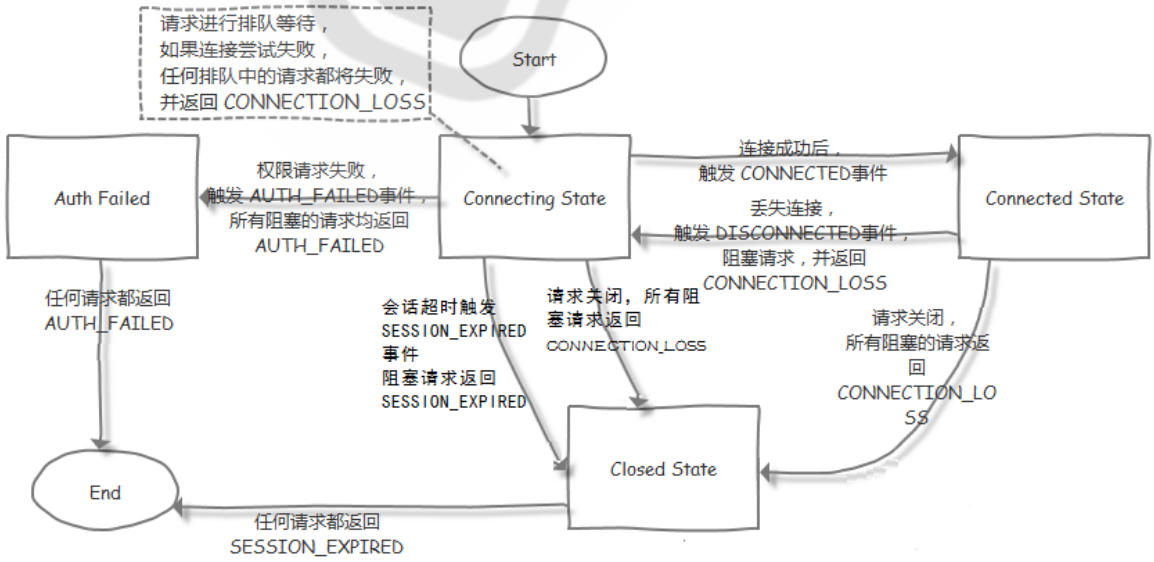
如果Client因为Timeout和Zookeeper Server失去连接，client处在CONNECTING状态，会自动尝试再去连接Server，如果在session有效期内再次成功连接到某个Server，则回到CONNECTED状态，否则的话，原来的会话中断，进入CLOSED状态。

注意：

如果因为网络状态不好，client和Server失去联系，client会停留在当前状态，会尝试主动再次连接Zookeeper Server。client不能宣称自己的session expired，session expired是由Zookeeper Server来决定的，client可以选择自己主动关闭session。

流程解析

流程示意图



小结

数据模型

学习目标：

这一节，我们从 基础知识、类型解析、小结 三个方面来学习

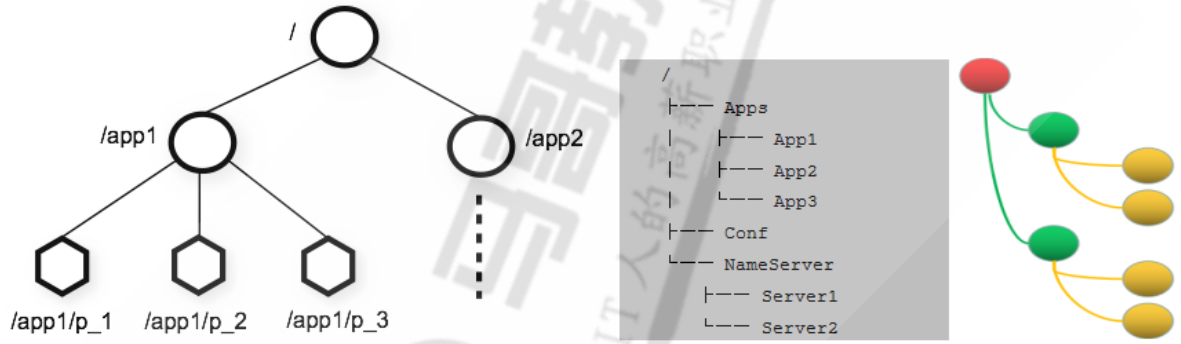
基础知识

简介

在 Zookeeper 中，节点分为两类，第一类是指 构成Zookeeper集群的主机，称之为主机节点；第二类则是指内存中 Zookeeper数据模型中的数据单元，用来存储各种数据内容，称之为数据节点 ZNode。

Zookeeper内部维护了一个层次关系(树状结构)的数据模型，它的表现形式类似于linux的文件系统，甚至操作的种类都一致。
Zookeeper数据模型中有自己的根目录(/)，根目录下有多个子目录，每个子目录后面有若干个文件,由斜杠(/)进行分割的路径，就是一个 ZNode,每个 ZNode 上都会保存自己的数据内容 和 一系列属性信息。

样式



注意：
ZNode是一个类似树状结构，有自己的根节点和子节点。
每个子目录项都是ZNode，这个ZNode是被它所在的路径唯一标识，如App1的znode的标识为/Apps/App1

每个ZNode都有自己对应的目录路径和内部的文件数据，当Zookeeper的客户端在和服务端创建连接后，服务端会给客户端创建一个会话(Session),Zookeeper客户端就可以在这个会话中，对自己的ZNode进行增删改查等操作了。
也就是说：ZNode的所有操作是随着Zookeeper的连接/断开实现的。

类型解析

简介

虽然ZNode的样式跟Linux文件系统类似，根据节点的生命周期，在Zookeeper中的ZNode有四种独有的特性,有时候页称为四种类型：

基本节点：
Persistent(持久节点)：会话断开后，除非主动进行移除操作，否则该节点一直存在
Ephemeral(临时节点)：会话断开后，该节点被删除

序列节点：
    Persistent Sequential:按顺序编号的持久节点
        该节点被创建的时候，Zookeeper 会自动在其子节点名上，加一个由父节点维护的、自增整数的后缀
    Ephemeral Sequential：按顺序编号的临时节点
        该节点被创建的时候，Zookeeper 会自动在其子节点名上，加一个由父节点维护的、自增整数的后缀

    注意：
        只有持久性节点(持久节点和顺序持久节点)才有资格创建子节点

自增后缀格式：
    10位10进制数的序号

有序和无序区别：
    多个客户端同时创建同一无序ZNode节点时，只有一个可创建成功，其它匀失败。并且创建出的节点名称与创建时指定的节点名完全一样
    多个客户端同时创建同一有序ZNode节点时，都能创建成功，只是序号不同。

功能简介

    Zookeeper使用这个基于内存的树状模型来存储分布式数据，正因为所有数据都存放在内存中，所以Zookeeper才能实现高性能的目的，提高数据的吞吐率。特别是在集群主机节点间的数据同步。

    Znode包含了 存储数据(data)、访问权限(acl)、子节点引用(child)、节点状态(stat)信息等信息

注意：
    为了保证高吞吐和低延迟，以及数据的一致性，znode只适合存储非常小的数据，不能超过1M，最好都小于1K

小结

版本

学习目标：

这一节，我们从 基础知识、小结 两个方面来学习。

基础知识

业务需求

工作中，Zookeeper在服务架构中，主要是做服务注册和服务发现的角色的，这两个功能主要体现在服务的配置文件上，而由于服务的版本更新，所以配置文件也需要有相应的版本控制功能，版本控制的本质就是历史记录的追踪，即"溯源性"。
那么如何将这些不同版本的配置信息都存储下来呢？Zookeeper的ZNode提供了一个"版本"的功能来解决这个问题。

版本样式

ZNode中可以基于"版本"的功能来存储多版本的数据，其实就是在一个访问路径中可以访问不同版本的数据，存储的样式就是，在同一个ZNode目录中使用多个子目录来存储不同版本的数据，只不过每份数据的version号是自动增加的。

数据结构

```
Zookeeper 的 ZNode 上都会存储数据，对应于每个 ZNode，Zookeeper 都会为其维护一个叫做 Stat 的数据结构，Stat 中记录了这个 ZNode 的三个数据版本：
    dataversion 当前 ZNode 数据内容的版本
    cversion    当前 ZNode 子节点的版本
    aversion    当前 ZNode 的 ACL 变更版本。

这里的版本起到了控制 Zookeeper 操作原子性的作用，基于这些功能，才能更好实现了分布式锁的功能。
```

小结

监听

学习目标：

这一节，我们从 基础知识，原理解析、小结 三个方面来学习。

基础知识

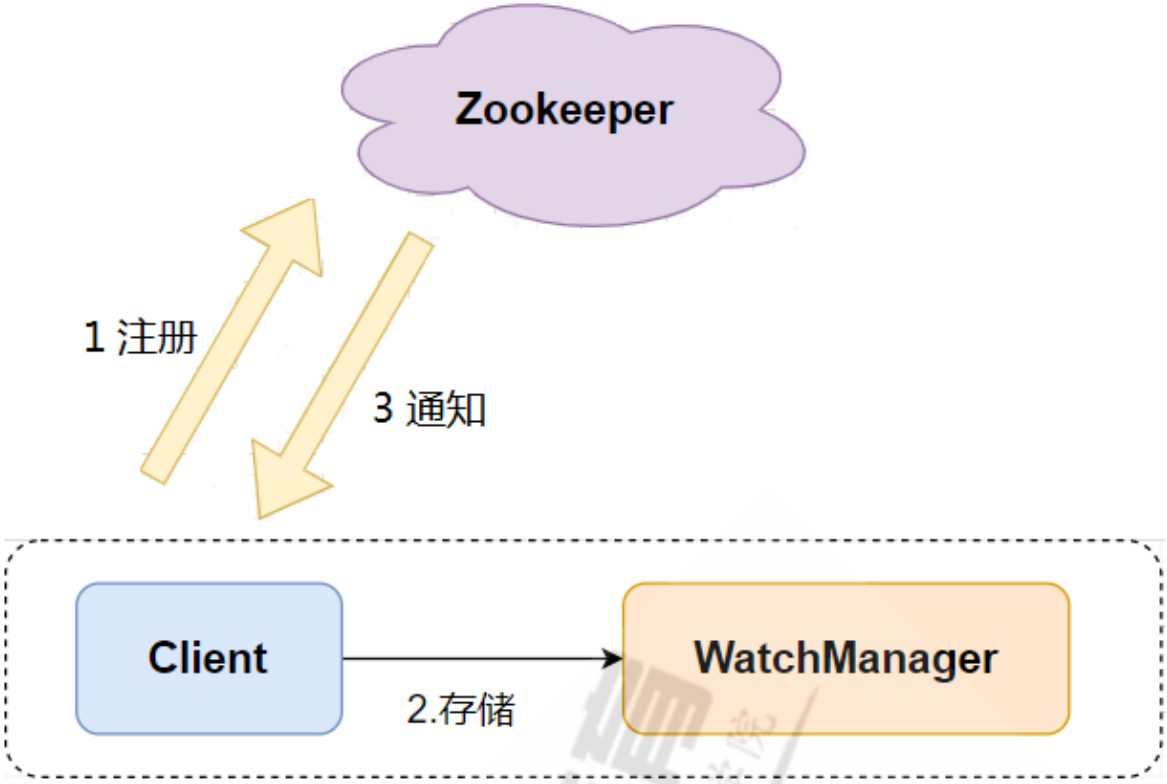
监听简介

```
    Watcher(事件监听器)是 Zookeeper 提供的一种 发布/订阅的机制。

    Zookeeper 允许客户端在指定的集群数据节点上注册一些 Watcher，当发生ZNode存储数据的修改，子节点目录的变化等情况的时候，Zookeeper 服务端会将事件通知给监听的客户端，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

    该机制是 Zookeeper 实现分布式协调的重要特性，也是Zookeeper的核心特性,Zookeeper的很多功能都是基于这个特性实现的。
    -- 这个过程是异步的。
```

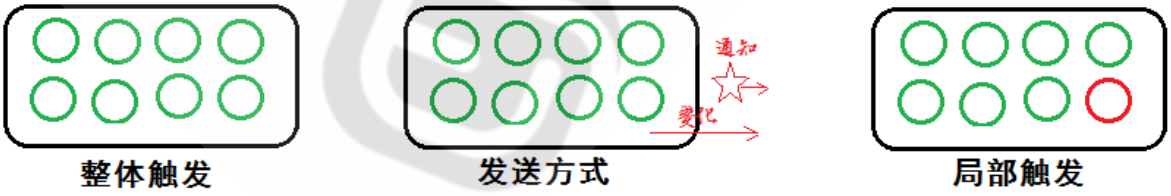
工作原理



ZooKeeper的Watcher机制主要包括客户端线程、客户端 WatcherManager、Zookeeper服务器三部分。客户端向ZooKeeper服务器注册Watcher的同时，会将Watcher对象存储在客户端的WatchManager中。当zookeeper服务器触发watcher事件后，会向客户端发送通知， 客户端线程从 WatcherManager 中取出对应的 Watcher 对象来执行回调逻辑。

关键点

Zookeeper的监听机制主要有三个关键点：整体触发、发送方式、局部触发。



整体触发：

当设置监视的数据发生改变时，该监视事件会被发送到客户端，常见的就是监控ZNode中的数据或子目录发生变化。

发送方式：

Zookeeper服务端被触发的时候，会基于会话给客户端发送信息，但是由于网络的原因，经常会出现网络延迟的因素，造成客户端接收的结构不一致，而Zookeeper有一个很重要的特点就是：一致性，为了达到这个目标，Zookeeper的监听机制在信息发送的方式上，就有了一个发送特点：

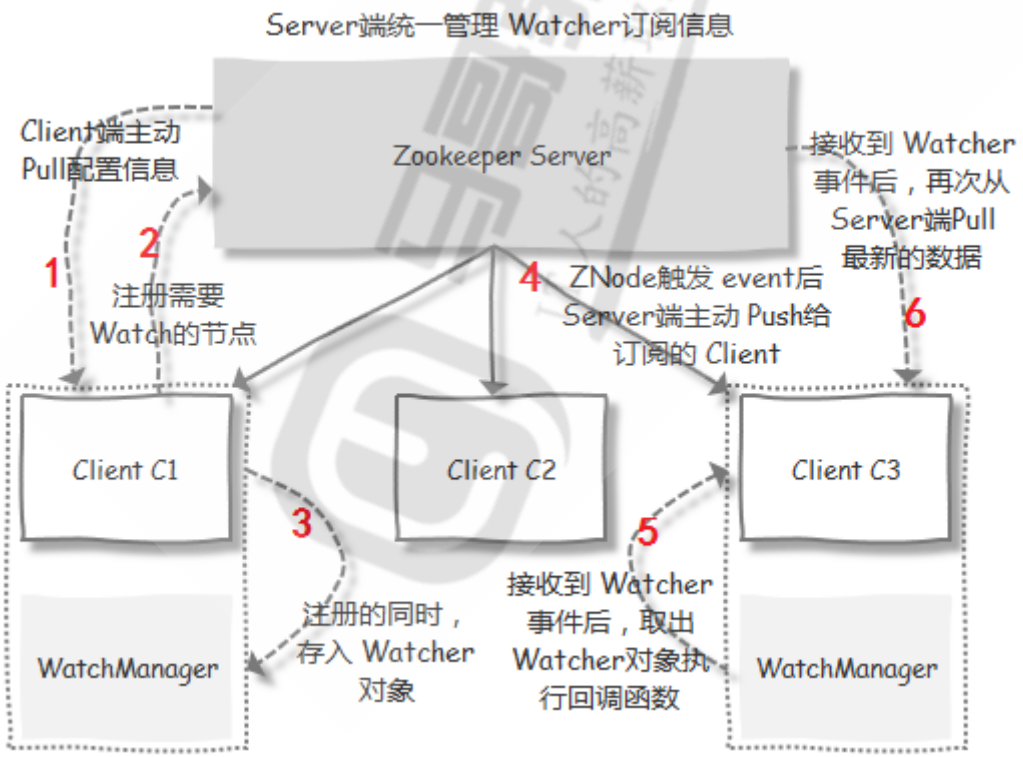
所有的监视事件被触发后，不会立即发送至客户端，而是以异步的方式发送至监视者的，而且 Zookeeper 本身提供了顺序保证。效果就是：客户端首先看到监视事件，然后才会感知到它所设置监视的znode发生了变化。这样就达到了，虽然不同的客户端在不同的时刻感知到了监视事件，但是客户端所看到的效果都是真实一致的。

局部触发

当客户端监视的Zookeeper节点ZNode内部有比较多的子目录数据的时候，这种场景下，我们只需要监视其中的一个小部分重要的数据，其他的数据是一些无关紧要的，所以就没有必要监视全部的ZNode数据变化，这意味着znode节点本身就应该具有不同的触发事件方式：即支持对ZNode数据事件的局部触发。

原理解析

实现流程



异常处理

Zookeeper 中的监视功能是轻量级的，易设置、维护和分发。

当客户端与 Zookeeper 服务器失去联系时，客户端并不会收到监视事件的通知，只有当客户端重新连接后，若在必要的情况下，以前注册的监视会重新被注册并触发，对于开发人员来说这通常是透明的。

只有一种情况会导致监视事件的丢失。即通过设置了某个znode节点的监视，但是如果某个客户端在此znode节点被创建和删除的时间间隔内与zookeeper服务器失去了联系，该客户端即使稍后重新连接zookeeper服务器后也得不到事件通知。

小结

快速入门

环境安装

学习目标

这一节，我们从 基础环境、简单实践、进阶实践、小结 四个方面来学习。

基础环境

简介

```
zookeeper 是依赖于java环境的，所以我们需要提前定制java环境
```

定制java环境

```
创建目录
mkdir /data/{softs,server} -p
cd /data/softs

下载java或者上传java
ls /data/softs
安装java
tar xf jdk-8u121-linux-x64.tar.gz -C /data/server
cd /data/server/
ln -s jdk1.8.0_121 java

配置java环境变量
echo 'export JAVA_HOME=/data/server/java' >> /etc/profile
echo 'export JRE_HOME=$JAVA_HOME/jre' >> /etc/profile
echo 'export CLASSPATH=$JAVA_HOME/lib/tools.jar:$JAVA_HOME/lib/dt.jar' >>
/etc/profile
echo 'export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH' >> /etc/profile
source /etc/profile

检查效果
java -version

检查java目录效果
tree -L 1 /data/server/java/
```

简单实践

安装软件

```
软件准备
cd /data/softs
wget http://archive.apache.org/dist/zookeeper/zookeeper-3.7.0/apache-zookeeper-
3.7.0-bin.tar.gz
```

```
wget http://archive.apache.org/dist/zookeeper/zookeeper-3.7.0/apache-zookeeper-3.7.0-bin.tar.gz.asc

校验软件安全
gpg --verify apache-zookeeper-3.7.0-bin.tar.gz.asc

对比 MD5 码一致后进行解压安装
tar zxvf apache-zookeeper-3.7.0-bin.tar.gz -C /data/server
cd /data/server
ln -s apache-zookeeper-3.7.0-bin zookeeper

echo 'export PATH=/data/server/zookeeper/bin:$PATH' > /etc/profile.d/zk.sh
source /etc/profile.d/zk.sh
```

查看配置

```
查看配置模板文件
cat zookeeper/conf/zoo_sample.cfg
grep -ni '^[a-z]' zookeeper/conf/zoo_sample.cfg
```

```
root@python-auto:/data/server/zookeeper# grep -ni '^[a-z]' conf/zoo_sample.cfg
2:tickTime=2000
5:initLimit=10
8:syncLimit=5
12:dataDir=/tmp/zookeeper
14:clientPort=2181
```

序号	配置项	作用
1	tickTime	"滴答时间"，用于配置Zookeeper中最小的时间单元长度，单位毫秒，是其他时间配置的基础
2	initLimit	初始化时间，包含启动和数据同步，其值是tickTime的倍数
3	syncLimit	正常工作，心跳监测的时间间隔，其值是tickTime的倍数
4	dataDir	配置Zookeeper服务存储数据的目录
5	clientPort	配置当前Zookeeper服务对外暴露的端口，用户客户端和服务端建立连接会话

```
设置配置文件
cp conf/zoo_sample.cfg conf/zoo.cfg
```

启动服务

```
在Zookeeper的bin目录下有很多执行文件，其中zkServer.sh是启动服务的脚本文件
ls bin/

查看帮助信息
bin/zkServer.sh

命令参数功能详解
    start：用于后台启动Zookeeper服务器
    start-foreground：用于前台启动Zookeeper服务器，常用来排查失败原因
    stop：用于关闭Zookeeper服务器
```

```
    restart：用于重启Zookeeper服务器
    status：用于查看Zookeeper服务器状态
    upgrade：用于升级Zookeeper服务器
    print-cmd：用于打印Zookeeper程序命令行及其相关启动参数

启动服务
bin/zkServer.sh start
```

```
root@python-auto:/data/server/zookeeper# bin/zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /data/server/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

注意：

如果我们下载的软件是 apache-zookeeper-3.7.0.tar.gz 的话，在启动的时候，会发生报错

错误：找不到或无法加载主类 org.apache.zookeeper.server.quorum.QuorumPeerMain

检查效果

```
Zookeeper的检查有很多种方式，主要有以下四种：端口、服务、进程、连接
端口检查
    netstat -tnulp | grep 2181
服务检查
    bin/zkServer.sh status
进程检查
    ps aux | grep zoo
连接检查
    bin/zkCli.sh
```

进阶实践

需求

在生产中，我们一般会讲Zookeeper的数据目录和日志目录都放在一个专用的路径下，而我们刚才实践的效果是数据目录在临时文件夹/tmp下，而且没有设置日志文件配置信息，那么接下来我们就按照生产环境的部署方法先来做一个单机版的Zookeeper环境。

简单实践

```
关闭刚才的服务
bin/zkServer.sh stop

创建专用的数据和日志目录
cd /data/server/zookeeper
mkdir {data,logs}

在默认的配置文件中，没有日志的配置项，日志的配置项是dataLogDir
# vim conf/zoo.cfg
# grep -ni '^[a-z]' conf/zoo.cfg
2:tickTime=2000
5:initLimit=10
8:syncLimit=5
```

```
12:dataDir=/data/server/zookeeper/data
13:dataLogDir=/data/server/zookeeper/logs
15:clientPort=2181

启动之前注意权限
ll
chown 1000.1000 -R /data/server/zookeeper*

启动当前Zookeeper的服务
bin/zkServer.sh start

三种方式查看不同的关注点
bin/zkServer.sh status
ps aux | grep zoo
bin/zkCli.sh

查看产生的数据
ls /data/server/zookeeper/data/
ls /data/server/zookeeper/logs/
```

小结

基本信息

学习目标

这一节，我们从 基本操作、信息查看、小结 三个方面来学习。

基本操作

本地连接服务

```
    当Zookeeper服务器正常启动后，我们就可以使用Zookeeper自带的zkCli.sh脚本，以命令行的方式连接到Zookeeper。使用方法非常简单：
    bin/zkCli.sh

如果出现下面信息，就表示命令行客户端已经成功连入到Zookeeper
    WATCHER::

    WatchedEvent state:SyncConnected type:None path:null
    [zk: localhost:2181(CONNECTED) 0]
```

远程连接服务

```
zkCli.sh 脚本还提供了远程连接非本地的Zookeeper服务器的参数 -server，使用这个参数就可以连接到远程的Zookeeper服务主机

命令格式：
    bin/zkCli.sh -server <zk_ip>:<zk_port>

远程连接
    bin/zkCli.sh -server 192.168.8.14:2181
```

命令帮助

当客户端成功的连接到Zookeeper服务后，我们可以输入任意非法的命令都可以获取Zookeeper客户端相关的命令使用方法

```
dsafsa
```

连接到Zookeeper服务后，输入help查看相关命令

```
ZooKeeper -server host:port cmd args                宿主机命令行执行Zookeeper客户端命令
    stat path [watch]                               查看节点状态或者判断结点是否存在
    set path data [version]                         设置节点数据
    ls path [watch]                                 列出节点信息
    delquota [-n|-b] path                           删除节点个数(-n)或数据长度(-b)配额
    ls2 path [watch]                                ls命令的加强版，列出更多信息
    setAcl path acl                                 设置节点的权限信息
    setquota -n|-b val path                         设置节点个数(-n)或数据长度(-b)的配额
    history                                         列出最近的命令历史，可以和redo配合使用
    redo cmdid                                      再次执行某个命令，结合history使用
    printwatches on|off                             设置和显示监视状态
    delete path [version]                           删除节点，不可删除有子节点的节点
    sync path                                       强制数据同步
    listquota path                                  显示节点资源配额信息
    rmr path                                        强制删除节点
    get path [watch]                                获取节点数据
    create [-s] [-e] path data acl                  创建顺序(-s)或临时(-e)结点
    addauth scheme auth                             配置节点认证信息
    quit                                            退出连接
    getAcl path                                     获取节点的权限信息
    close                                           断开当前Zookeeper连接
    connect host:port                               连接Zookeeper服务端
```

这些命令主要分为五大类：基本信息查看、节点基本操作、资源配额、权限设置、其他操作等，接下来我们分别对这些命令进行学习。

关闭连接

使用close命令可以关闭当前的连接

关闭连接后，查看效果

```
    [zk: localhost:2181(CONNECTED) 2] close
    2021-07-01 20:24:02,434 [myid:] - INFO  [main:ZooKeeper@684] - Session:
0x16455baaf1f0002 closed
    [zk: localhost:2181(CLOSED) 3] 2021-07-01 20:24:02,438 [myid:] - INFO
[main-EventThread:ClientCnxn$EventThread@519] - EventThread shut down for
session: 0x16455baaf1f0002

    [zk: localhost:2181(CLOSED) 3] ls /
    Not connected
```

重新连接

使用connect host:port命令可以重新连接Zookeeper服务

重新连接

```
[zk: localhost:2181(CLOSED) 4] connect 192.168.8.14:2181
...
WatchedEvent state:SyncConnected type:None path:null

[zk: 192.168.8.14:2181(CONNECTED) 5] ls /
[sswang_quota, zookeeper, sswang1]
```

退出服务

使用quit命令可以退出Zookeeper服务

退出连接

```
[zk: 192.168.8.14:2181(CONNECTED) 6] quit
Quitting...
2021-07-01 20:26:41,582 [myid:] - INFO  [main:ZooKeeper@684] - Session: 0x16455baaf1f0003 closed
2021-07-01 20:26:41,582 [myid:] - INFO  [main-EventThread:ClientCnxn$EventThread@519] - EventThread shut down for session: 0x16455baaf1f0003
[root@controller zookeeper]#
```

信息查看

节点列表

关于节点列表主要有两种命令ls和ls2，他们的意思都是"获取路径下的节点信息"

命令格式：

```
ls path      显示当前的节点列表
```

注意：

此path路径为绝对路径

显示普通效果

```
[zk: 127.0.0.1:2181(CONNECTED) 4] ls /
[zookeeper]
```

注意：

该结果表示Zookeeper服务端的根目录下有一个Zookeeper的子节点，

它是Zookeeper的保留节点，一般不用。

节点状态

stat命令作用：判断节点是否存在，节点不存在则报错，否则显示节点的状态信息，

命令格式：

```
stat [ -w ] path
```

注意：

注意事项同ls

查看未知节点

```
[zk: 127.0.0.1:2181(CONNECTED) 7] stat /nihao
```

```
    Node does not exist: /nihao

查看已知节点
    [zk: 127.0.0.1:2181(CONNECTED) 8] stat /zookeeper
    cZxid = 0x0                                     节点创建时的zxid
    ctime = Thu Jan 01 08:00:00 CST 1970            节点创建时间
    mZxid = 0x0                                     节点最近一次更新时的zxid
    mtime = Thu Jan 01 08:00:00 CST 1970            节点最近一次更新的时间
    pZxid = 0x0                                     父节点创建时的zxid
    cversion = -1                                   子节点数据更新次数
    dataVersion = 0                                 本节点数据更新次数
    aclVersion = 0                                  节点ACL(授权信息)的更新次数
    ephemeralOwner = 0x0                            持久节点值为0，临时节点值为sessionid
    dataLength = 0                                  节点数据长度
    numChildren = 1                                 子节点个数
```

小结

节点操作

学习目标:

这一节，我们从 节点创建、节点删除、小结 三个方面来学习。

节点创建

简介

```
使用create命令可以来创建一个节点，命令格式如下：
create [-s] [-e] [-c] [-t ttl] path [data] [acl]
注意：
    -s 表示创建的节点是顺序节点。
    -e 表示创建的节点是临时节点，这个是create的默认参数。
    acl 用于权限控制，Zookeeper的权限控制很强大，默认不使用。
```

创建持久节点

```
[zk: 127.0.0.1:2181(CONNECTED) 11] create /sswang sswang
Created /sswang

[zk: 127.0.0.1:2181(CONNECTED) 12] ls /
[zookeeper, sswang]

[zk: 127.0.0.1:2181(CONNECTED) 13] create /sswang sswang
Node already exists: /sswang

[zk: 127.0.0.1:2181(CONNECTED) 14] create -s /sswang1 sswang1
Created /sswang10000000001
```

```
[zk: 127.0.0.1:2181(CONNECTED) 15] ls /
[sswang10000000001, zookeeper, sswang]
注意：
    顺序节点的命名格式：名称+序列号(10位数字)

[zk: 127.0.0.1:2181(CONNECTED) 16] create -s /sswang1 sswang1
Created /sswang10000000002

[zk: 127.0.0.1:2181(CONNECTED) 17] create -s /sswang1 sswang1
Created /sswang10000000003

[zk: 127.0.0.1:2181(CONNECTED) 18] create -s /sswang1 sswang1
Created /sswang10000000004

[zk: 127.0.0.1:2181(CONNECTED) 19] ls /
[sswang, sswang1, sswang10000000001, sswang10000000002, sswang10000000003,
sswang10000000004, zookeeper]
```

创建临时节点

```
[zk: 127.0.0.1:2181(CONNECTED) 16] create -e /sswang2 sswang2
Created /sswang2

[zk: 127.0.0.1:2181(CONNECTED) 17] ls /
[sswang2, sswang10000000001, zookeeper, sswang]

[zk: 127.0.0.1:2181(CONNECTED) 18] create -s -e /nihao nihao
Created /nihao0000000003

[zk: 127.0.0.1:2181(CONNECTED) 19] ls /
[nihao0000000003, sswang2, sswang10000000001, zookeeper, sswang]
注意：
    临时节点，其实就是会话节点，当我们的会话结束时候，该节点就会自动删除
```

查看状态

```
查看三者的状态信息
[zk: 127.0.0.1:2181(CONNECTED) 19] stat /sswang
...
ephemeralOwner = 0x0
...
[zk: 127.0.0.1:2181(CONNECTED) 20] stat /sswang2
...
ephemeralOwner = 0x16454e2c6580007          # 这是临时节点的特点
...
[zk: 127.0.0.1:2181(CONNECTED) 21] stat /sswang10000000001
...
ephemeralOwner = 0x0
...
```

创建子节点

```
[zk: 127.0.0.1:2181(CONNECTED) 22] create -e /sswang2/nihao nihao
Ephemerals cannot have children: /sswang2/nihao

[zk: 127.0.0.1:2181(CONNECTED) 23] create -s /sswang10000000001/nihao nihao
Created /sswang10000000001/nihao0000000000

[zk: 127.0.0.1:2181(CONNECTED) 24] ls /sswang10000000001
[nihao0000000000]
[zk: 127.0.0.1:2181(CONNECTED) 25] ls /sswang10000000001/nihao0000000000
[]
```

可以看到：

只有持久节点才可以创建子节点，这也应照了四种节点的状态

节点删除

简介

删除节点有两种方式：正规删除和强制删除。

命令格式：

正规删除：delete path [version]

强制删除：deleteall path

注意：

注意事项同ls

delete只能删除不包含子节点的节点

正规删除

```
delete删除非空节点
[zk: localhost:2181(CONNECTED) 2] ls /sswang
[nihao]
[zk: localhost:2181(CONNECTED) 3] delete /sswang
Node not empty: /sswang

先删子节点
[zk: localhost:2181(CONNECTED) 4] delete /sswang/nihao
[zk: localhost:2181(CONNECTED) 5] ls /sswang
[]

再删节点
[zk: localhost:2181(CONNECTED) 6] delete /sswang
[zk: localhost:2181(CONNECTED) 7] ls /
[sswang2, sswang10000000001, zookeeper]
```

可以看到：

使用delete删除节点的时候，只能删除不包含子节点的节点

强制删除

```
[zk: localhost:2181(CONNECTED) 9] deleteall /sswang10000000001
[zk: localhost:2181(CONNECTED) 10] ls /
[sswang2, zookeeper]
[zk: localhost:2181(CONNECTED) 11] deleteall /sswang2
```

小结

```
```

资源操作

学习目标

这一节，我们从 数据操作、资源配额、小结 三个方面来学习。

数据操作

准备工作

```
创建一个持久目录/sswang
[zk: localhost:2181(CONNECTED) 0] create /sswang sswang
Created /sswang
[zk: localhost:2181(CONNECTED) 1] ls /
[zookeeper, sswang]
```

获取节点数据

```
使用get命令可以获取节点数据，命令格式如下：
get [-s] [-w] path
注意：
    注意事项同ls

获取/sswang节点数据
[zk: localhost:2181(CONNECTED) 2] get -s /sswang
sswang
cZxid = 0x22
ctime = Sun Jul 01 18:45:01 CST 2021
mZxid = 0x22
mtime = Sun Jul 01 18:45:01 CST 2021
pZxid = 0x22
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
```

设置节点数据

```
使用set命令可以获取节点数据，命令格式如下：
set [-s] [-v version] path data
注意：
    注意事项同ls
    如果要指定 version，一定要是当前的version值，默认情况就是当前的version值，可以不写

更新/sswang节点数据
[zk: localhost:2181(CONNECTED) 3] set /sswang sswang1
```

```
cZxid = 0x22
ctime = Sun Jul 01 18:45:01 CST 2021
mZxid = 0x23
mtime = Sun Jul 01 18:48:29 CST 2021
pZxid = 0x22
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 7
numChildren = 0
```

注意：

dataVersion的内容是每次更新，该值会递增下去

当前的version值是1，如果要使用指定版本设置数据，version一定是当前的值，否则报错

```
[zk: localhost:2181(CONNECTED) 6] set /sswang sswang3 -v 8
version No is not valid : /sswang
[zk: localhost:2181(CONNECTED) 10] set /sswang sswang -v 1
cZxid = 0x22
ctime = Sun Jul 01 18:45:01 CST 2021
mZxid = 0x28
mtime = Sun Jul 01 18:51:37 CST 2021
pZxid = 0x22
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 7
numChildren = 0
```

资源配额

查看资源配额

使用listquotat命令可以获取节点数据，命令格式如下：

```
listquota path
```

注意：

注意事项同ls

查看指定结点的资源

```
[zk: localhost:2181(CONNECTED) 28] create /sswang sswang
Created /sswang
[zk: localhost:2181(CONNECTED) 29] listquota /sswang
absolute path is /zookeeper/quota/sswang/zookeeper_limits
quota for /sswang does not exist.
```

可以看到：

默认新创建的结点资源，是没有资源配额的。

设置资源配额

使用listquotat命令可以获取节点资源配额数据，命令格式如下：

```
setquota -n|-b|-N|-B val path
```

注意：

-n 设置节点的节点个数

-b 设置节点的数据长度

如果超出了配置限制，不会停止行为操作，只是ZooKeeper将会在log日志中打印WARN日志。
其他注意事项同ls

设置节点的数量资源

```
[zk: localhost:2181(CONNECTED) 34] setquota -n 2 /sswang
Comment: the parts are option -n val 2 path /sswang
[zk: localhost:2181(CONNECTED) 35] listquota /sswang
absolute path is /zookeeper/quota/sswang/zookeeper_limits
Output quota for /sswang count=2,bytes=-1
Output stat for /sswang count=1,bytes=6
```

可以看到：

Output stat 后面的 count 表示的是总数量，bytes指定的是数据总长度，包括的子节点的数据长度

指定数量的话，数据长度默认没有限制

测试效果：

在/sswang节点下创建多个子节点

```
create /sswang/child1 1
create /sswang/child2 1
create /sswang/child3 1
create /sswang/child4 1
```

查看zookeeper-root-server-python-auto.out文件信息，会有WARN提示信息

```
2021-07-01 20:07:30,649 [myid:] - WARN  [SyncThread:0:DataTree@301] - Quota exceeded: /sswang count=3 limit=2
2021-07-01 20:08:06,715 [myid:] - WARN  [SyncThread:0:DataTree@301] - Quota exceeded: /sswang count=4 limit=2
```

设置节点的数据长度资源

```
[zk: localhost:2181(CONNECTED) 54] create /sswang1 sswang
Created /sswang1
[zk: localhost:2181(CONNECTED) 57] setquota -b 8 /sswang1
Comment: the parts are option -b val 8 path /sswang1
[zk: localhost:2181(CONNECTED) 58] listquota /sswang1
absolute path is /zookeeper/quota/sswang1/zookeeper_limits
Output quota for /sswang1 count=-1,bytes=8
Output stat for /sswang1 count=1,bytes=6
```

可以看到：

指定数据长度的话，节点数量默认没有限制

测试效果

修改/sswang1的数据长度

```
set /sswang1 777777779
set /sswang1 7777777790
```

查看zookeeper-root-server-python-auto.out文件信息，会有WARN提示信息

```
2021-07-01 20:15:00,763 [myid:] - WARN  [SyncThread:0:DataTree@347] - Quota exceeded: /sswang1 bytes=9 limit=8
2021-07-01 20:15:17,330 [myid:] - WARN  [SyncThread:0:DataTree@347] - Quota exceeded: /sswang1 bytes=10 limit=8
```

删除资源配额

使用delquota命令可以删除节点资源配额数据，命令格式如下：

```
delquota [-n|-b] path
```

注意：

-n 删除节点的节点个数

-b 删除节点的数据长度

```
    其他注意事项同ls

删除资源配额
    [zk: localhost:2181(CONNECTED) 11] delquota -n /sswang
    [zk: localhost:2181(CONNECTED) 12] delquota -b /sswang1

检查效果
    [zk: localhost:2181(CONNECTED) 14] listquota /sswang
    [zk: localhost:2181(CONNECTED) 15] listquota /sswang1

补充：
    资源限额可以给Znode节点设置，也可以局部给子节点设置。
```

小结

日志实践

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

```
    zookeeper服务器会产生三类日志：事务日志、快照日志和系统日志。

    我们可以在zookeeper的配置文件zoo.cfg中 通过
        dataDir 设定数据快照日志的存储位置
        dataLogDir 设定事务日志的存储位置，如果不设置该项，这默认保存在 dataDir目录下
        注意：
            事务日志和快照日志都会保存在 指定目录的 version-2 子目录下。
            我们倾向于 dataLogDir 和 dataLog 单独配置，因为zookeeper集群频繁读写操作，
            可能会产生大龄日志，有可能影响系统性能，可以根据日志的特性，使用不同的存储介质
    zookeeper的系统运行日志是可以通过三个位置来进行设置
        1 在log4j.properties文件中 通过
            zookeeper.log.dir=.  来设置，这里的'.'指的是zkServer.sh坐在的目录
        2 在 zkEnv.sh 文件中通过
            ZOO_LOG_DIR="$ZOOKEEPER_PREFIX/logs" 来设置
        3 在 zkServer.sh 文件中 通过
            ZOO_LOG_FILE=zookeeper-$USER-server-$HOSTNAME.log
            _ZOO_DAEMON_OUT="$ZOO_LOG_DIR/zookeeper-$USER-server-$HOSTNAME.out"
            来指定
```

事务日志

日志简介
    事务日志是指 Zookeeper 系统在正常运行过程中，针对所有的更新操作，在返回客户端 更新成功 的响应前，Zookeeper 会保证已经将本次更新操作的事务日志写到磁盘上，只有这样，整个更新操作才会生效。

日志查看
    zookeeper的事务日志为二进制文件，不能通过vim等工具直接访问,可以通过zookeeper自带的功能文件读取事务日志文件。
    对于3.5.5 之前的zookeeper需要借助于大量的jar包来实现日志的查看，比如
    java -cp .:zookeeper-3.7.0.jar:slf4j-api-1.7.30.jar org.apache.zookeeper.server.LogFormatter log.1

    zookeeper 从 3.5.5 版本之后，就取消的LogFormatter ，使用了一个更好的 TxnLogToolkit 工具,这个工具放置在了 bin/ 目录下，文件名是 zkTxnLogToolkit.sh
    使用方式： zkTxnLogToolkit.sh log_file

快照日志

日志简介
    zookeeper的数据在内存中是以树形结构进行存储的，而快照就是每隔一段时间就会把整个DataTree的数据序列化后存储在磁盘中。

日志查看
    同样在 3.5.5 版本之后，我们可以基于 zkSnapShotToolkit.sh 命令来进行快照日志的查看

简单实践

事务日志查看

```
日志文件查看
# ls logs/version-2/ -lh
总用量 24K
-rw-r--r-- 1 root root 65M 8月  10 16:38 log.100000005
-rw-r--r-- 1 root root 65M 8月  10 16:06 log.800000001
-rw-r--r-- 1 root root 65M 8月  10 16:23 log.b00000001
结果显示：
    每个日志文件大小是 65M，文件名规则 'log.9个字符',这9个字符指的是事务id

日志内容查看
# ./bin/zkTxnLogToolkit.sh logs/version-2/log.b00000001
ZooKeeper Transactional Log File with dbid 0 txnlog format version 2
    # 这是每个事务日志文件都有的日志头，输出了 dbid 还有 version等信息
2021-08-10 17:07:23,272 [myid:] - INFO  [main:ZookeeperBanner@42] -
...
2021-08-10 17:07:23,289 [myid:] - INFO  [main:ZooKeeperServer@260] -
zookeeper.intBufferStartingSizeBytes = 1024
21-8-10 下午04时23分21秒 session 0x100009bc0f50000 cxid 0x0 zxid 0xb00000001
closeSession v{}
    # 这是xx时候，sessionid 请求类型为 closeSession，表示关闭了会话
EOF reached after 1 txns.
```

快照日志查看

```
日志文件查看
```

```
# ls data/version-2/ -lh
总用量 20K
-rw-r--r-- 1 root root    2 8月  10 16:45 acceptedEpoch
-rw-r--r-- 1 root root    2 8月  10 16:45 currentEpoch
-rw-r--r-- 1 root root 2.1K 8月  10 16:05 snapshot.600000000
-rw-r--r-- 1 root root 2.1K 8月  10 16:06 snapshot.700000000
-rw-r--r-- 1 root root 2.0K 8月  10 16:15 snapshot.800000001
结果显示:
    快照日志的命名规则为'snapshot.9个字符',
    这9个字符表示zookeeper触发快照的那个瞬间，提交的最后一个事务的ID。

日志内容查看
# ./bin/zkSnapShotToolkit.sh data/version-2/snapshot.800000001
2021-08-10 17:08:25,876 [myid:] - INFO  [main:SnapStream@61] -
zookeeper.snapshot.compression.method = CHECKED
...
----
/zookeeper/quota/sswang2/zookeeper_stats        路径
  cZxid = 0x00000000000051                      创建节点时的 Zxid
  ctime = Tue Aug 10 14:44:11 CST 2021          创建节点的时间
  mZxid = 0x00000000000051                      节点最近一次更新对应的 Zxid
  mtime = Tue Aug 10 14:44:11 CST 2021          节点最近一次更新的时间
  pZxid = 0x00000000000051                      父节点的 Zxid
  cversion = 0                                  子节点更新次数
  dataVersion = 0                               数据更新次数
  aclVersion = 0                                节点 acl 更新次数
  ephemeralOwner = 0x00000000000000             节点的 sessionid值
  dataLength = 16                               存储的数据长度
----
这里表达的是当前抓取快照日志文件的时间记录
Session Details (sid, timeout, ephemeralCount):
0x1000048d7820002, 30000, 0
0x100009bc0f50000, 30000, 0
----
Last zxid: 0x800000001
```

系统日志查看

```
日志查看
# ls bin/zook*
bin/zookeeper_audit.log

# ls logs/zook*
logs/zookeeper_audit.log  logs/zookeeper-root-server-python-auto.out

查看集群运行日志
# cat logs/zookeeper-root-server-python-auto.out
...
2021-08-10 16:45:54,068 [myid:1] - INFO  [NIOWorkerThread-2:Learner@158] -
Revalidating client: 0x1000048d7820002
```

小结

配置进阶

学习目标

这一节，我们从 基础配置、集群配置、小结 三个方面来学习。

基础配置

在Zookeeper的默认配置文件中，主要有以下7个配置项用的还算是比较多的：

配置	解析
tickTime（SS / CS）	用来指示 服务器之间或客户端与服务器之间维护心跳机制的 最小时间单元，Session 最小过期时间默认为两倍的 tickTime（default：2000ms）
initLimit（LF）	集群中的 Leader 节点和 Follower 节点之间初始连接时能容忍的最多心跳数（default：10 tickTime）
syncLimit（LF）	集群中的 Leader 节点和 Follower 节点之间请求和应答时能容忍的最多心跳数（default：5 tickTime）
dataDir	Zookeeper 保存服务器存储快照文件的目录，默认情况，Zookeeper 将 写数据的日志文件也保存在这个目录里（default：/tmp/zookeeper）
clientPort	客户端连接 Zookeeper 服务器的端口，Zookeeper 会监听这个端口，接受客户端的访问请求（default：2181）
dataLogDir	用来存储服务器事务日志
minSessionTimeout & maxSessionTimeout	默认分别是 2 * tickTime ~ 20 * tickTime，来用控制 客户端设置的 Session 超时时间。如果超出或者小于，将自动被服务端强制设置为最大或者最小

集群配置

简介

```
为了配置 Zookeeper 集群，会在配置文件末尾增加如下格式的服务器节点配置
    格式：server.<myid>=<server_ip>:<LF_Port>:<L_Port>
```

格式解析

```
<myid>
    表示节点编号，是该节点在集群中唯一的编号，取值范围是1~255之间的整数，而且我们必须在dataDir目录下创建一个myid的文件，将节点对应的<myid>值输入到该节点的myid文件。

<server_ip>
    表示集群中的节点ip地址，可以使用主机名或ip来表示，生产中如果配置好内部dns的话，推荐使用主机名，本机地址的表示方法是：127.0.0.1或者localhost

<LF_Port>
    表示Leader节点和Follower节点进行心跳检测与数据同步所使用的端口。
```

<L_Port>
表示进行领导选举过程中，用于投票通信的端口。

注意：
这些端口可以随机自己定义。
真正的生产环境中，不同主机上的clientPort、LF_Port、L_Port三个端口一般可以配置成一样，因为生产集群中每个server主机都分布在不同的主机上，都有独立的ip地址，不会造成端口冲突

小结

集群操作

集群基础

学习目标

这一节，我们从 基础知识、流程解析、小结 三个方面来学习。

基础知识

zookeeper集群

Zookeeper为了更好的实现生产的业务场景，一般都会采用分布式的集群架构。
Zookeeper集群通常由2n+1台Server节点组成，每个Server都知道彼此的存在。每个server都维护的内存状态镜像以及持久化存储的事务日志和快照。
对于2n+1台server，只要有>=(n+1)台server节点可用，整个Zookeeper系统保持可用。

为了维护集群内部所有主机信息的一致性，他们自己参考Paxos协议自己设计了一个更加轻量级的协议:Zab(Zookeeper Atomic Broadcast)来解决集群数据一致性的问题。
这个协议主要包括两部分：领导选举、日常通信，这两步就组成了Zookeeper的集群流程。
为了更好的理解集群流程，我们首先来学习一下Zookeeper集群的角色信息

集群角色

序号	角色	责任描述
1	领导者(Leader)	领导者不直接接受client请求，负责进行投票发起和决议，更新系统状态
2	跟随者(Follower)	接收客户请求并向客户端返回结果，在选Leader过程中参与投票
3	观察者(Observer)	转交客户端写请求给leader节点，和同步leader状态，不参与选主投票
4	学习者(Learner)	和leader进行状态同步的节点统称Learner，Follower和Observer都是
5	客户端(client)	请求发起方

流程解析

集群流程

集群流程主要包括两阶段组成：Leader选举和日常操作。
Leader选举阶段：
    Leader选举主要有两种场景：集群启动、Leader恢复。
    集群启动：当Zookeeper集群启动时候，将会选择一台server节点为为Leader，其它的server节点就作为follower(暂不考虑observer)，接着就进入日常操作阶段。
    Leader恢复：当集群Leader主机服务重启或者崩溃后，当Zookeeper集群中所有Server主机基于Zab协议选举新的Leader者，接着就进入日常操作阶段。然后其他Server主机和新的Leader主机进行数据信息同步，当状态同步完成以后，

日常操作阶段：
    日常操作阶段主要有两种场景：主机间心跳监测和数据操作。
    主机间心跳监测：
        当Leader选举完毕后，就进入日常操作阶段，第一步就是所有集群节点都互相保持通信，
        然后Leader和Follower节点间进行数据同步，确保所有主机节点都是相同的状态，
        当所有Follower节点和新的Leader主机完成数据信息同步以后，就开始进行日常的数据操作。
    数据操作：
        follower来接收client的请求，对于不改变系统一致性状态的读操作，
        由follower的本地内存数据库直接给client返回结果；对于会改变Zookeeper系统状态的更新操作，
        则转交由Leader进行提议处理，处理完毕后，将成功的结果给client。

主机状态

Zookeeper集群中每个Server主机在工作过程中有以下种状态：

Looking(迷茫者)：我老大是谁？
    当前Server主机不知道集群中的Leader是谁，陷入深深的不安，正在搜寻主心骨。
Leading(领导者)：我是总统。
    当前Server主机被集群选举策略确定的Leader，我的地盘我做主。
Following(执行者)：我是员工。
    集群的Leader主机已经选举出来，当前Server主机与之同步信息，严格执行规章制度，保质保量完成任务。
Observing(观望者)：我是看客
    observer会观察leader是否有改变，然后同步leader的状态，是系统扩展的一种方法

通信机制

对于Zookeeper集群来说，我们要考虑的内容主要有三大块：客户端连接、主机通信、选举Leader。
    客户端连接：   客户端连接服务端功能，进行相关请求操作
    主机通信：   集群各服务节点进行信息交流的功能
    选举Leader：   集群中各服务节点共同选举主节点的功能
格式：
    server.<myid>=<server_ip>:<LF_Port>:<L_Port>

为了同时满足这三者的要求，Zookeeper分别将这三种功能以三个常见端口对外提供服务：
    客户端操作： 2181
    主机通信：   2182
    选举Leader： 2183
注意：
    这三端口都是自定义的，在生产中，因为每台主机都有独立的ip，所以三个端口一般都设置一样。

小结

集群部署

学习目标

这一节，我们从 部署方案、环境部署、小结 三个方面来学习。

部署方案

集群分类

对于Zookeeper的集群来说，有两种情况：生产集群和伪集群。
生产集群：
　　使用多个独立的主机，每个主机上都部署同样环境的Zookeeper环境，基于内部的Zab协议达到数据的一致性，然后统一对外提供服务。客户端连接任意一节点，效果都一样。

伪集群：
　　对于Zookeeper集群来说，主机节点可以分布在不同的主机上，也可以部署在同一台主机上，只需要各个Zookeeper服务之间使用不同的通信接口即可，因为不是真正意义上的"集群"，所以称为"伪集群"。
　　伪集群本质上还是集群，只不过是部署在了一台主机，其他功能完全一样。

我们接下来的操作就是在一台主机上部署三个Zookeeper服务，形成一个伪集群。

集群规划

　　对于伪集群来说，就不能像生产部署方式一致了，因为使用同一个ip，同一个端口会造成冲突，所以我们要对整个单一节点的Zookeeper伪集群进行相应的规划

主机节点	主机ip	通信端口	心跳端口	选举端口	服务目录	myid
zk1	192.168.8.14	2181	2182	2183	/data/server/zk1/{data,logs}	1
zk2	192.168.8.14	2281	2282	2283	/data/server/zk2/{data,logs}	2
zk3	192.168.8.14	2381	2382	2383	/data/server/zk3/{data,logs}	3

环境部署

准备工作

　　对于Zookeeper集群来说，本质上是部署了一个个单独的Zookeeper服务，所以准备环境跟单机部署Zookeeper完全一致。
　　操作内容参考 快速入门 部分的环境配置

软件安装

我们是三节点的Zookeeper集群，所以我们在安装软件的时候，分别在三个不同的Zookeeper目录中执行软件安装
安装三个节点

```
    tar xf /data/softs/apache-zookeeper-3.7.0.tar.gz -C /data/server/
    mv /data/server/apache-zookeeper-3.7.0 /data/server/zk1
    mkdir /data/softs/zk1/{data,log} -p
```

注意：
    三个节点执行同样的操作，唯一的区别是数字不一致，分别是1-2-3

配置管理

准备配置文件

```
cd /data/server/
mv zk1/conf/zoo_sample.cfg zk1/conf/zoo.cfg

修改配置文件
# grep -ni '^[a-z]' zk1/conf/zoo.cfg
2:tickTime=2000
5:initLimit=10
8:syncLimit=5
12:dataDir=/data/server/zk1/data
13:dataLogDir=/data/server/zk1/log
15:clientPort=2181
30:server.1=192.168.8.14:2182:2183
31:server.2=192.168.8.14:2282:2283
32:server.3=192.168.8.14:2382:2383

设置myid文件
echo 1 > zk1/data/myid
```

注意：
    三个节点执行同样的操作，唯一的区别是绿色背景的字体不一致，分别是1-2-3

以上三个节点内容都配置完毕后，我们就可以启动集群了。

    与单机模式的启动方法一致，只需一次启动所有Zookeeper节点即可启动整个集群。我们还可以一个一个的手工启动，当然了我们还可以使用脚本方式一次启动所有Zookeeper主机服务。

```
启动服务
/data/server/zk1/bin/zkServer.sh start
/data/server/zk2/bin/zkServer.sh start
/data/server/zk3/bin/zkServer.sh start
```

服务检查

```
检查集群服务一般有两类方法：检查端口和检查集群服务状态
检查端口
netstat -tnulp | grep 218

查看集群服务状态
[root@controller ~]# /data/server/zk1/bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /data/server/zk1/bin/../conf/zoo.cfg
Mode: follower
[root@controller ~]# /data/server/zk2/bin/zkServer.sh status
```

```
ZooKeeper JMX enabled by default
Using config: /data/server/zk2/bin/../conf/zoo.cfg
Mode: leader
[root@controller ~]# /data/server/zk3/bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /data/server/zk3/bin/../conf/zoo.cfg
Mode: follower
```

结果显示：

查看集群状态，关键就是看Mode:的值，我们可以看到，目前Zookeeper三节点集群中，处于leader的是zk2节点，其他两个节点是follower角色。

同时连接多个server的方法

```
bin/zkCli -server <zk1_ip>:<zk1_port>,<zk2_ip>:<zk2_port>,<zk3_ip>:<zk3_port>
```

注意：

同时连接多个server节点的时候，彼此间使用逗号隔开

使用任意一个zkCli.sh连接三个Zookeeper节点

```
cd /data/server/zk2/bin/
./zkCli.sh -server 192.168.8.14:2181,192.168.8.14:2281,192.168.8.14:2381
```

专用检测

简介

因为使用telnet方法来检查集群的节点状态信息比较繁琐，而且经常中断，所以生产中我们一般使用nc软件来检查Zookeeper集群状态

nc全称NetCat，在网络工具中有“瑞士军刀”美誉，支持Windows和Linux。因为它短小精悍(不过25k)、功能实用，被设计为一个简单、可靠的网络工具，可通过TCP或UDP协议传输读写数据。同时，它还是一个网络应用Debug分析器，因为它可以根据需要创建各种不同类型的网络连接。

简单实践

安装软件

```
apt-get -y install netcat-traditional
```

使用方式

```
    echo "命令" | nc <server_ip> <server_port>
```

检查集群状态

```
echo stat | nc 127.0.0.1 2181
echo stat | nc 127.0.0.1 2281
echo stat | nc 127.0.0.1 2381
```

小结

集群操作

这一节，我们从 命令简介、命令实践、角色实践、小结 四个方面来学习。

命令简介

常见命令

命令	内容
conf	输出相关服务配置的详细信息
cons	列出所有连接到服务器的客户端的完全的连接/会话的详细信息
envi	输出关于服务环境的详细信息
dump	列出未经处理的会话和临时节点
stat	查看哪个节点被选择作为 Follower 或者 Leader
ruok	测试是否启动了该 Server，若回复 imok 表示已经启动
mntr	输出一些运行时信息
reqs	列出未经处理的请求
wchs	列出服务器 watch 的简要信息
wchc	通过 session 列出服务器 watch 的详细信息
wchp	通过路径列出服务器 watch 的详细信息
srvr	输出服务的所有信息
srst	重置服务器统计信息
kill	关掉 Server
isro	查看该服务的节点权限信息

ZooKeeper 支持某些特定的四字命令字母与其的交互。它们大多是查询命令，用来获取 ZooKeeper 服务的当前状态及相关信息。用户在客户端可以通过 telnet 或 nc 向 ZooKeeper 提交相应的命令。

默认情况下，这些4字命令有可能会被拒绝，发送如下报错

```
xxx is not executed because it is not in the whitelist.
```

解决办法：向 zoo.cfg 文件中添加如下配置

```
4lw.commands.whitelist=*
```

命令实践

基础命令

```
查看节点服务状态
echo stat | nc 127.0.0.1 2281

查看节点服务配置
echo conf | nc 127.0.0.1 2281

查看节点服务环境
echo envi | nc 127.0.0.1 2281

查看节点服务会话
echo cons | nc 127.0.0.1 2281
echo dump | nc 127.0.0.1 2281
```

基本安全

在这么多的服务状态查看命令中有很多存在隐患的命令，所以为了避免生产中因为这些命令的安全隐患，所以我们要对这些命令进行一些安全限制，只需要编辑服务的zoo.cfg文件即可

```
# vim /data/server/zk1/conf/zoo.cfg
4lw.commands.whitelist=stat, ruok, conf, isro

重启服务后
/data/server/zk1/bin/zkServer.sh restart

查看允许通过的命令效果
echo isro | nc 127.0.0.1 2181
echo conf | nc 127.0.0.1 2181
echo stat | nc 127.0.0.1 2181

检查不允许通过的命令
[root@controller bin]# echo dump | nc 127.0.0.1 2181
dump is not executed because it is not in the whitelist.

测试没有设置命令过滤的节点
[root@controller bin]# echo dump | nc 127.0.0.1 2281
SessionTracker dump:
Session Sets (0):
ephemeral nodes dump:
Sessions with Ephemerals (0):

所以生产中，我们一定要把不知道或者不想用的命令全部过滤掉，这样才能保证基本的安全。
```

角色实践

当前状态

```
当前角色效果
/data/server/zk1/bin/zkServer.sh status
/data/server/zk2/bin/zkServer.sh status
/data/server/zk3/bin/zkServer.sh status
可以看到：
    zk2的角色是Leader
```

关闭服务

```
关闭zk2服务
/data/server/zk2/bin/zkServer.sh stop

检查节点角色
/data/server/zk1/bin/zkServer.sh status
/data/server/zk3/bin/zkServer.sh status
可以看到：
    zk3已经成为了新的主
```

恢复服务

启动zk2服务
/data/server/zk2/bin/zkServer.sh start

检查节点角色
/data/server/zk1/bin/zkServer.sh status
/data/server/zk2/bin/zkServer.sh status
/data/server/zk3/bin/zkServer.sh status

注意

当集群中的服务主机数量少于一半的时候，整个集群服务就崩溃了。

小结

管理技巧

学习目标

这一节，我们从 状态监控、常见优化、小结 三个方面来学习。

状态监控

监控指标

在Zookeeper服务端的操作中，有一个命令非常有用就是mntr，可以查看节点服务的所有运行时信息，这些信息就是我们平常要监控到的内容。

命令示例

命令效果
```
# echo mntr | nc 127.0.0.1 2281
zk_version  3.7.0-39d3a4f269333c922ed3db283be479f9deacaa0f, built on 03/23/2021 10:13 GMT
zk_avg_latency  0
zk_max_latency  0
zk_min_latency  0
zk_packets_received 8
zk_packets_sent 7
zk_num_alive_connections    1
zk_outstanding_requests 0
zk_server_state leader
zk_znode_count  4
zk_watch_count  0
zk_ephemerals_count 0
zk_approximate_data_size    27
zk_open_file_descriptor_count   36
zk_max_file_descriptor_count    4096
zk_followers    2
zk_synced_followers 2
zk_pending_syncs    0
```

指标分类

网络响应延迟信息

```
    zk_avg_latency、zk_max_latency、zk_min_latency

网络请求(数据包和连接状态数量)
    数据包相关：zk_packets_received、zk_packets_sent
    连接状态相关：zk_num_alive_connections(活跃连接)、zk_outstanding_requests

节点数量信息：
    zk_znode_count、zk_watch_count、zk_ephemerals_count(临时节点数)

服务状态
    zk_server_state、zk_open_file_descriptor_count、zk_max_file_descriptor_count

Leader特有：
    zk_followers、zk_synced_followers(同步数量)、zk_pending_syncs(阻塞数量)
```

常见优化

文件隔离

生产中Zookeeper的dataDir 和 dataLogDir 应该分开部署，因为事务日志非常重要而且内容比较多，所以在配置的时候，dataLogDir所在的目录，要保证目录空间足够大，并挂载到单独的磁盘上，如果可以的话，磁盘应该开启实时刷新功能。

日志滚动

默认情况下，一般日志是放在一个文件中，为了更好的查看日志效果，我们一般会将日志进行切割，接下来我们配置一下日志的切割功能。

Zookeeper的默认日志切割配置文件是 项目目录的conf/log4j.properties,和切割配置主要相关的是：

```
log4j.appender.ROLLINGFILE=org.apache.log4j.RollingFileAppender
```

如果想按天进行日志切割的话，可以修改为 DaliyRollingFileAppender

Zookeeper使用日志切割功能

```
# vim /data/server/zk1/bin/zkServer.sh
...
 30 # 增加 ZOO_LOG_DIR 配置
 31 ZOO_LOG_DIR="$ZOOBINDIR/../log4j"
...

# vim /data/server/zk1/bin/zkEnv.sh
 59 if [ "x${ZOO_LOG4J_PROP}" = "x" ]
 60 then
 61     ZOO_LOG4J_PROP="INFO,ROLLINGFILE"       # 注意：原CONSOLE 修改为
ROLLINGFILE
 62 fi
```

日志清理

自动清理：自从Zookeeper 3.4.0版本之后，配置文件中多了两个和日志自动清理相关的配置

autopurge.purgeInterval：指定清理频率，单位为小时(默认是0，表示不开启自动清理)

autopurge.snapRetainCount：和purgeInterval配合使用，指定需要保留的文件数目

注意：

Zookeeper 重启会自动清除 zookeeper-root-server-python-auto.out 日志，如果有排错需要，则应先备份好日志文件

配置效果：

```
# vim /data/server/zk1/conf/zoo.cfg
...
autopurge.purgeInterval=1
autopurge.snapRetainCount=3
```

手工清理：

如果发现单事务日志量过大，导致定时清理无法及时处理，我们可以基于自定义脚本或者 zookeeper提供的 zkCleanup.sh 进行 结合 定时任务来实现自动清理的任务

```
#!/bin/bash
# 定制日志目录
zookeeperDir='/data/server/zookeeper'
dataDir="$zookeeperDir/data/version-2"
dataLogDir=$zookeeperDir/logs/version-2
# 保留文件60
count=60
count=$[$count+1]  # 从61行开始删除
ls -t $dataLogDir/log.* | tail -n +$count | xargs rm -f
ls -t $dataDir/snapshot.* | tail -n +$count | xargs rm -f
```

注意：

```
ls -t 是顺序排列，
tail -n +5  是从第 5 个至最新文件
```

节点扩展

在Zookeeper集群中有一个角色是observer，它主要的作用仅仅是增加额外的接收客户端请求的扩展节点，将接收到的请求，转交给Leader处理，不会影响集群的其他任何操作。

我们只需要在Observe节点的zoo.cfg配置文件中添加如下配置即可

```
peerType=observer
server.n:localhost:2181:3181:observer
```

修改配置文件

zk1节点：

```
vim /data/server/zk1/conf/zoo.cfg
# 修改如下配置
server.3=192.168.8.14:2382:2383:observer
```

zk2节点：

```
vim /data/server/zk2/conf/zoo.cfg
# 修改如下配置
server.3=192.168.8.14:2382:2383:observer
```

zk3节点：

```
vim /data/server/zk3/conf/zoo.cfg
# 增加如下配置
peerType=observer
# 修改如下配置
server.3=192.168.8.14:2382:2383:observer
```

重启相关服务

```
/data/server/zk1/bin/zkServer.sh restart
/data/server/zk2/bin/zkServer.sh restart
/data/server/zk3/bin/zkServer.sh restart
```

再次查看集群状态
    /data/server/zk3/bin/zkServer.sh status
    /data/server/zk2/bin/zkServer.sh status
    /data/server/zk1/bin/zkServer.sh status
    可以看到：
        zk3的集群角色就变成了观察者

验证observer是否参与选举
/data/server/zk2/bin/zkServer.sh stop

查看集群状态
/data/server/zk1/bin/zkServer.sh status
可以看到：
    集群节点有三个，zk3是观察者，真正提供服务的是两个，我们关闭了一个，集群服务就崩溃了，所以observer没 有参与集群的选举工作。

小结

Kafka

基础知识

场景需求

学习目标

这一节，我们从 基础知识、消息模式、小结 三个方面来学习。

基础知识

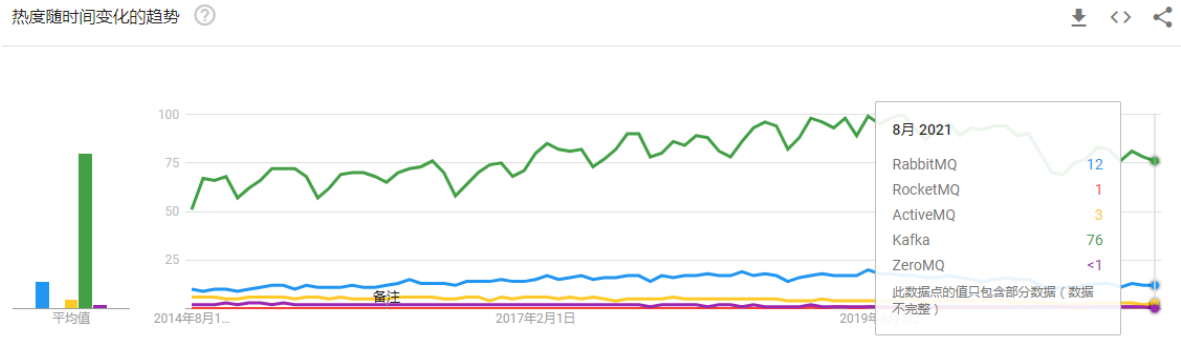
需求

在分布式场景中，相对于大量的用户请求来说，内部的功能主机之间、功能模块之间等，数据传递的数据量是无法想象的，因为一个用户请求，会涉及到各种内部的业务逻辑跳转等操作。
那么，在打用户量的业务场景中，如何保证所有的内部业务逻辑请求都处于稳定而且快捷的数据传递呢？
    -- 消息队列(Message Queue)

消息队列

    所谓的消息队列，我们可以通过名称来感受的出来，系统平台接收到的消息，临时存储到一个队列中，然后根据内部的排序机制，保证所有的消息依次到达对应的目标。消息队列不仅仅可以提高用户请求接收速度，还可以降低后端应用程序的压力。

    常见的消息队列实现软件有：
        专用软件：RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、MetaMq等
        数据库功能软件：Redis、Mysql、phxsql等
        其他具备消息队列功能的软件



功能特点

有了消息队列，原来拥挤的事务，我们可以非常轻松的进行处理了，这个特点主要体现在四个方面：

1 应用耦合：

多应用模块间通信，通过消息队列进行处理，避免调用接口异常导致整个过程失败

2 异步处理：

多应用模块间不用阻塞方式进行信息的处理，提高了消息并发处理的能力

3 限流削峰：

避免用户流量过大导致应用系统崩溃的情况，尤其是各种突发的场景，比如秒杀、双xx活动等

4 消息驱动的系统：

消息功能拆分，不同的角色进行不同功能的处理，

消息发送方

客户端，主动发出请求给服务端，服务端接收请求后，临时存储到消息队列

消息承载队列

接收客户端发送过来的请求，临时存储

消息接收方

获取消息队列中的请求后，然后本地执行处理，当然可以存在多个消息接收方

消息模式

基本队列

在生产方和消息队列中，他们是有方向关系的，而对于后两者来说，没有所谓的方向关系，也就是说，谁都可以处于主角色的定位，所以关于系统内部的消息传递会出现两种情况：

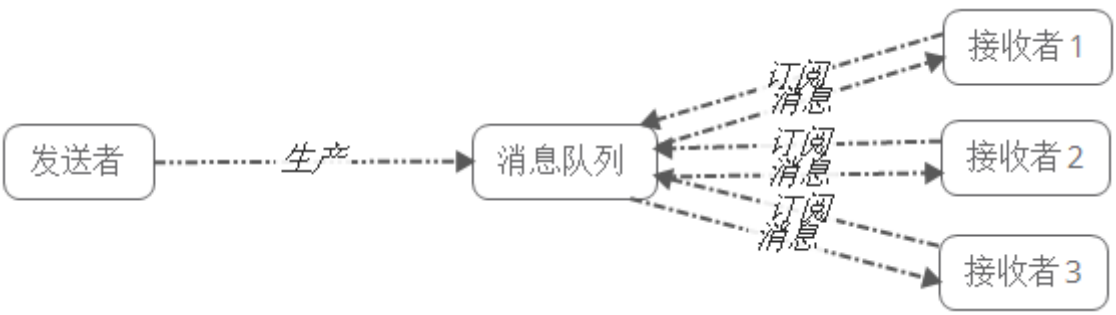
推模型

- 消息队列将消息推送给消息接收方
- 消息接收方如果消费能力不足的话，导致消息丢失。

拉模型

- 消息接收方主动从消息队列中获取数据
- 消费接收方可以根据自己的实际情况来调整消息的获取速度，但是有可能导致消息的积压问题。

发布订阅



在实际的业务架构中，尤其是项目内部的功能，生产方有很多，但是消息接收方会更多，往往一个生产方会有很多消费者来进行处理，为了互相不影响彼此的功能正常进行，就出现了 发布订阅模式。

为了

防止 推模型 中的单一消息接收者的压力导致消息丢失

防止 拉模型 中的消息队列过满导致生产方无法正常推送数据

将消息队列和消息接收者之间的处理机制进行改造：

- 每个消息可以有多个订阅者；
- 针对某个消息，它必须创建一个订阅者之后，才能消费发布者的消息。
- 为了消费消息，订阅者需要提前订阅该角色主题，并保持在线运行；

消息路由

无论是 基本队列模式，还是发布订阅模式，队列在其中都扮演了举足轻重的角色。然而，在企业应用系统中，当系统变得越来越复杂时，对性能的要求也会越来越高，此时对于系统而言，可能就需要支持同时部署多个队列，并可能要求分布式部署不同的队列。

这些队列可以根据定义接收不同的消息，例如订单处理的消息，日志信息，查询任务消息等。这时，对于消息的生产者和消费者而言，并不适宜承担决定消息传递路径的职责。事实上，根据S单一职责原则，这种职责分配也是不合理的，它既不利于业务逻辑的重用，也会造成生产者、消费者与消息队列之间的耦合，从而影响系统的扩展。

所以这时引入一个新的对象专门负责传递路径选择的功能，这就是所谓的消息路由模式

通过消息路由，我们可以配置路由规则指定消息传递的路径，以及指定具体的消费者消费对应的生产者。例如指定路由的关键字，并由它来绑定具体的队列与指定的生产者（或消费者）。路由的支持提供了消息传递与处理的灵活性，也有利于提高整个系统的消息处理能力。同时，路由对象有效地封装了寻找与匹配消息路径的逻辑，负责协调消息、队列与路径寻址之间关系。

小结

Kafka基础

学习目标

这一节，我们从 基础知识、原理解析、小结 三个方面来学习。

基础知识

简介

APACHE KAFKA

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

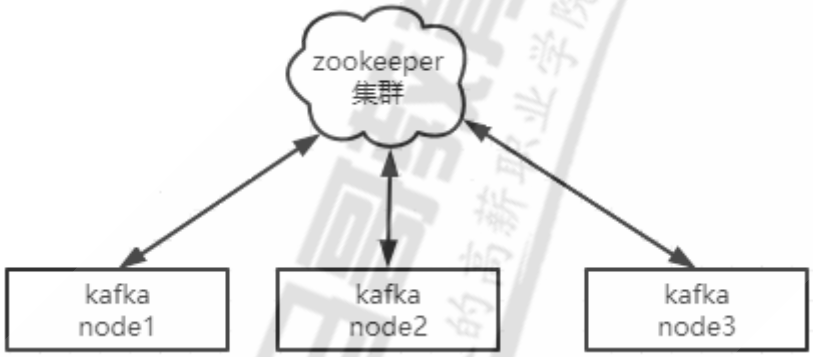
Kafka是最初由Linkedin公司开发，是一个分布式、分区的、多副本的、多订阅者，基于zookeeper协调的分布式日志系统，常见可以用于web/nginx日志、访问日志，消息服务等等。

Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

应用场景是：日志收集系统和消息系统。

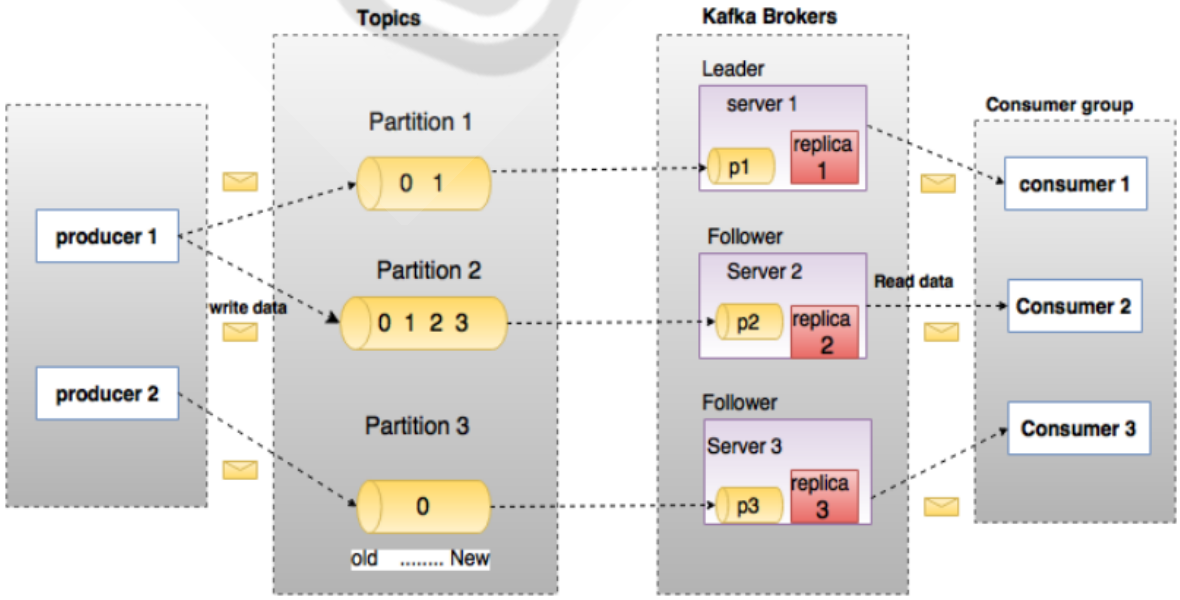
官网地址：https://kafka.apache.org/

vs zookeeper



ZooKeeper 是安装 Kafka 集群的必要组件，Kafka 通过 ZooKeeper 来实施对元数据信息的管理，包括集群、broker、主题、分区等内容。

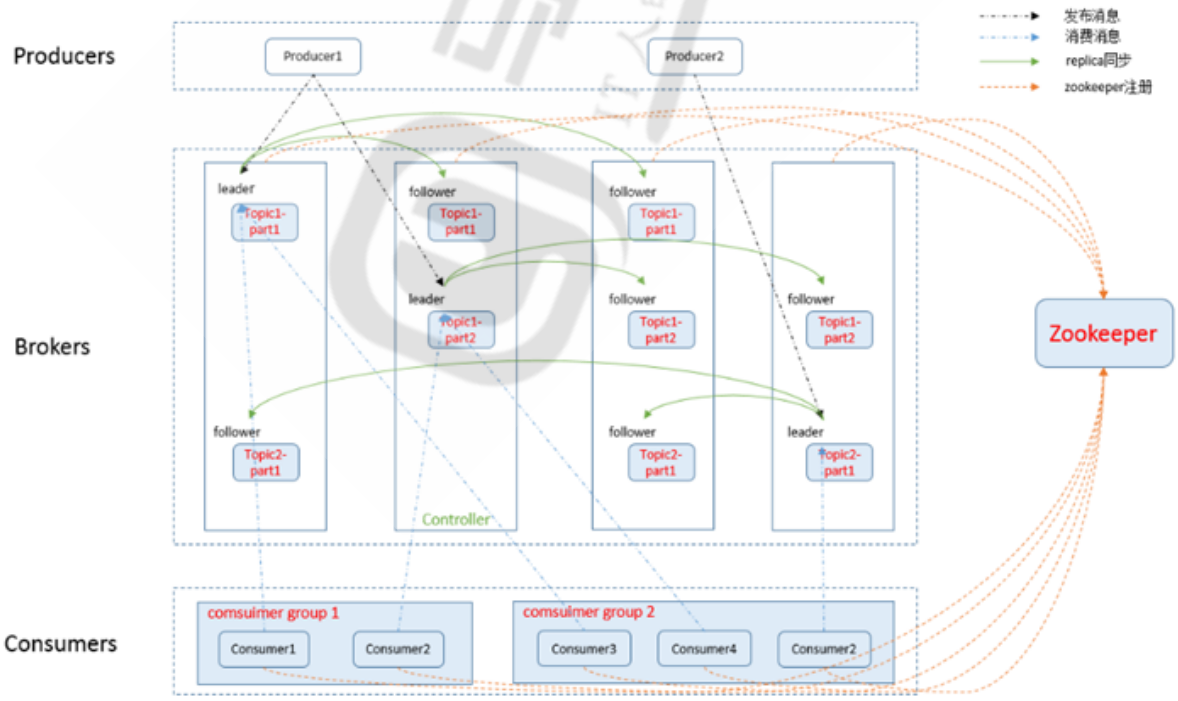
原理解析



基本概念

术语	解析
broker	kafka每个节点都称为broker
leader	所有broker中的主节点
follower	其他的非主broker节点
topic	存储关键字，或者理解为消息类别，可以理解为一个队列
partition	消息最终存放的地方，归属于topic，一个topic有多个存储地方.在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件。
replication	同样数据的副本，防止整体损坏，是Kafka的高可靠性的保障来源
group	接收消息的群组
offset	偏移量，指向开始获取消息的位置
producer	生产消息的地方
consumer	获取并指向消息的地方
AR	Assigned Replicas，分区中的所有副本的统称，AR = ISR + OSR
ISR	In Sync Replicas，所有与leader副本保持一定程度同步的副本集合，是AR的子集
OSR	Out-of-Sync Replied，所有与leader副本同步滞后过多的副本集合，是AR的子集

整体效果图



Kafka集群中包含如下几个部分：

若干Producer(任何可以产生流数据的位置)、大量broker节点

大量Consumer Group 及Zookeeper集群

作为一个消息系统，Kafka遵循了传统的消息处理流程：

Producer向broker push消息

Consumer从broker pull消息

Kafka通过Zookeeper管理集群
- 在所有broker节点中选举leader角色节点
- Consumer Group发生变化时进行rebalance

Producer使用push模式将消息发布到broker
- 每条消费都必须指定它的Topic，为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件。
- Producer发送消息到broker时，会根据Paritition机制选择将其存储到哪一个Partition
- 所有消息可以均匀分布到不同的Partition里，最终实现负载均衡。

Consumer使用pull模式从broker订阅并消费消息。
- Kafka可以基于广播或单播的方式，实现Topic消息通知给消费者
- Kafka还可以同时提供离线处理和实时处理。
  1 使用Storm这种实时流处理系统对消息进行实时在线处理，
  2 用Hadoop这种批处理系统进行离线处理
  3 将数据实时备份到另一个数据中心
  只需要保证这三个操作所使用的Consumer属于不同的Consumer Group即可。

通信质量

Kafka为了保证整个过程中，通信的有效性，采取了多种策略方式，主要有以下几种：

At most once	消息可能会丢，但绝不会重复传输
At least one	消息绝不会丢，但可能会重复传输
Exactly once	每条消息肯定会被传输一次且仅传输一次。

当Producer向broker发送消息时，一旦这条消息被commit，因数replication的存在，它就不会丢。

如果Producer发送数据给broker后，遇到网络问题导致通信中断，Producer无法判断消息是否commit。

为了保证Kafka确定故障，Producer生成一种类主键的标识
一旦识别出来这可以实现故障时幂等性的重试多次，最终实现Exactly once。

Consumer在从broker读取消息后，有多种处理机制：
- commit操作后在Zookeeper中保存该Consumer在Partition中读取的消息的offset。
  Consumer下一次再读该Partition时会从下一条开始读取。
- 不commit，下一次读取的开始位置会跟上一次commit之后的开始位置相同。
- autocommit，Consumer一旦读到数据立即自动commit。

Kafka默认保证At least once，并且允许通过设置Producer异步提交来实现At most once。而Exactly once要求与外部存储系统协作，Kafka提供的offset机制，可以非常容易得使用这种方式。

小结

简单实践

环境安装

学习目标

这一节，我们从 环境部署、简单实践、小结 三个方面来学习。

环境部署

获取软件

```
cd /data/softs
wget https://mirrors.tuna.tsinghua.edu.cn/apache/kafka/2.8.0/kafka_2.12-2.8.0.tgz
```

安装软件

```
解压软件
tar xf kafka_2.12-2.8.0.tgz -C /data/server/
cd /data/server
ln -s kafka_2.12-2.8.0 kafka

定制环境变量
echo 'export PATH=/data/server/kafka/bin:$PATH' > /etc/profile.d/kafka.sh
source /etc/profile.d/kafka.sh
```

基本配置

```
目录结构
# tree kafka
kafka
├── bin          kafka的脚本文件
|   ├── ...
├── config       配置相关的文件
|   ├── connect-console-sink.properties
|   ├── connect-console-source.properties
|   ├── connect-distributed.properties
|   ├── connect-file-sink.properties
|   ├── connect-file-source.properties
|   ├── connect-log4j.properties
|   ├── connect-mirror-maker.properties
|   ├── connect-standalone.properties
|   ├── consumer.properties
|   ├── kraft
|   |   ├── broker.properties
|   |   ├── controller.properties
|   |   ├── README.md
|   |   └── server.properties
|   ├── log4j.properties
|   ├── producer.properties
|   ├── server.properties                核心的配置文件
|   ├── tools-log4j.properties
|   ├── trogdor.conf
|   └── zookeeper.properties             Kafka内带的zk集群配置，不推荐
├── libs         运行库文件
|   ├── ...
├── LICENSE
├── licenses
|   ├── ...
├── NOTICE
└── site-docs
```

```
    └── kafka_2.12-2.8.0-site-docs.tgz

7 directories, 198 files

查看配置信息
# grep -Env '#|^$' server.properties
21:broker.id=0                                  主机唯一标识，如果有配套zk集群的话，推荐与myid
一致
31:#listeners=PLAINTEXT://:9092                 kafka服务器监听的端口
42:num.network.threads=3                        borker进行网络处理的线程数
45:num.io.threads=8                             borker进行I/O处理的线程数
48:socket.send.buffer.bytes=102400              发送缓冲区buffer大小
51:socket.receive.buffer.bytes=102400           接收缓冲区buffer大小
54:socket.request.max.bytes=104857600           kafka消息请求处理的最大数，不要超过java的堆栈
大小
60:log.dirs=/tmp/kafka-logs                     消息存放的目录，多目录间用逗号隔开，小于
num.io.threads
65:num.partitions=1                             默认的分区数，一个topic默认1个分区数
69:num.recovery.threads.per.data.dir=1          数据目录用来日志恢复的线程数目
74:offsets.topic.replication.factor=1           配置offset记录的topic的partition的副本个数
75:transaction.state.log.replication.factor=1   事务topic的复制个数
76:transaction.state.log.min.isr=1
103:log.retention.hours=168                     默认消息的最大持久化时间，168小时，7天
110:log.segment.bytes=1073741824                日志文件的最大容量，1G
# default.replication.factor=2                  默认的备份的复制自动创建topics的个数
114:log.retention.check.interval.ms=300000      日志间隔检查的时间300000毫秒
123:zookeeper.connect=localhost:2181            设置zookeeper的连接端口，多个地址间使用逗
号隔开
126:zookeeper.connection.timeout.ms=18000       设置zookeeper的连接超时时间
136:group.initial.rebalance.delay.ms=0
```

配置修改

```
创建日志目录
mkdir /data/server/kafka/logs

修改配置
# grep -Env '#|^$' config/server.properties
21:broker.id=1
32:listeners=PLAINTEXT://192.168.8.12:9092
42:num.network.threads=8
45:num.io.threads=8
48:socket.send.buffer.bytes=102400
51:socket.receive.buffer.bytes=102400
54:socket.request.max.bytes=104857600
60:log.dirs=/data/server/kafka/logs
65:num.partitions=3
69:num.recovery.threads.per.data.dir=1
74:offsets.topic.replication.factor=3
75:transaction.state.log.replication.factor=3
76:transaction.state.log.min.isr=2
103:log.retention.hours=3
110:log.segment.bytes=1073741824
114:log.retention.check.interval.ms=300000
115:default.replication.factor=2
123:zookeeper.connect=192.168.8.12:2181,192.168.8.13:2181,192.168.8.14:2181
126:zookeeper.connection.timeout.ms=18000
```

```
136:group.initial.rebalance.delay.ms=0
```

启动程序

```
前台启动
bin/kafka-server-start.sh config/server.properties

    注意：如果出现如下报错
        ERROR Fatal error during KafkaServer startup. Prepare to shutdown
        将 logs目录下的所有日志都清空掉，然后重新执行服务启动命令

后台启动
grep '-daemon' bin/kafka-server-start.sh
bin/kafka-server-start.sh -daemon config/server.properties
netstat -tnulp | grep 9092
jps

关闭服务
./bin/kafka-server-stop.sh config/server.properties
netstat -tnulp | grep 9092
jps

服务文件
# cat /lib/systemd/system/kafka.service
[Unit]
Description=Apache kafka
After=network.target

[Service]
Type=simple
Environment=JAVA_HOME=/data/server/java
PIDFile=/data/server/kafka/kafka.pid
ExecStart=/data/server/kafka/bin/kafka-server-start.sh
/data/server/kafka/config/server.properties
ExecStop=/bin/kill -TERM ${MAINPID}
Restart=always
RestartSec=20

[Install]
WantedBy=multi-user.target

重载服务
systemctl daemon-reload
systemctl start kafka.service
systemctl status kafka.service
systemctl stop kafka.service
```

简单实践

topic基本操作

```
文件 kafka-topics.sh
常见选项如下：
```

选项	描述
--alter	更改分区数，副本分配，和/或主题的配置。
--bootstrap-server <String: server to connect to>	必需：要连接的 Kafka 服务器。 如果提供此项，则不需要直接的 Zookeeper 连接。
--config <String: name=value>	设定配置属性
--create	创建一个新的topic
--delete	删除一个topic
--describe	列出给定主题的详细信息。
--help	打印帮助信息。
--list	列出所有可用的topic。
--partitions <Integer: # of partitions>	设置topic 分区数
--replication-factor <Integer:replication factor>	指定topic的副本数
--topic <String: topic>	指定topic 名称
--topics-with-overrides	如果在描述主题时设置，则仅显示已覆盖配置的主题
--version	展示Kafka版本
--zookeeper <String: hosts>	已弃用，zookeeper 连接的连接字符串，格式为 host:port。可以提供多个主机以允许故障转移。

创建topic

```
命令格式
    kafka-topics.sh --create --zookeeper <host>:<port> --if-not-exists --
replication-factor <副本数> --partitions <分区数> --topic <副本名称>


命令演示
    kafka-topics.sh --create --zookeeper 192.168.8.12:2181 --replication-factor
2 --partitions 1 --topic kafkatest
```

查看状态

```
命令格式
    kafka-topics.sh --zookeeper <host>:<port> --list

查看所有的topic
    kafka-topics.sh --list --zookeeper 192.168.8.12:2181
```

查看详情

```
命令格式
    kafka-topics.sh --zookeeper <host>:<port> --topic <副本名称> --describe

查看信息
    kafka-topics.sh --describe --zookeeper 192.168.8.12:2181 --topic kafkatest
```

```
root@python-auto:~# kafka-topics.sh --describe --zookeeper 192.168.8.12:2181 --topic kafkatest
Topic: kafkatest        TopicId: Gpj4bHJ_T0ORSMFOS6YPVg PartitionCount: 1       ReplicationFactor: 2    Configs:
        Topic: kafkatest        Partition: 0    Leader: 3       Replicas: 3,1   Isr: 3,1
```

参数	描述	参数	描述
Topic	topic名称	PartitionCount	分区数
ReplicationFactor	定义的分区数	Configs	配置
Partition	当前分区位置	Leader	当前那个broker为Leader
Replicas	副本位置	Isr	Isr同步队列

消息测试

```
终端1 - 创建一个broker，发布者发布消息
命令格式：
    kafka-console-producer.sh --broker-list <kafkaIP1>:<端口> <kafkaIP2>:<端口> --topic <topic名称>

命令示例：
    kafka-console-producer.sh --broker-list 192.168.8.12:9092 --topic kafkatest
    >first message      输入消息后，Enter发布
    >


终端2 - 创建一个consumer，接收消息：
命令格式：
    kafka-console-consumer.sh --bootstrap-server <host>:<post> --topic <topic名称> --from-beginning

命令示例：
    kafka-console-consumer.sh --bootstrap-server 192.168.8.12:9092 --topic kafkatest --from-beginning
--- 命令执行后，界面处于阻塞状态，当发布端发布消息的时候，这里会自动输出消息
first message
```

信息查看

```
连接zookeeper
zkCli.sh -server 192.168.8.12:2181
    注意：最好使用指定主机方式

查看效果
ls /
ls /brokers/ids
get /brokers/ids/3
ls /brokers/topics
get /brokers/topics/kafkatest
```

```
[zk: 192.168.8.12:2181(CONNECTED) 0] ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, feature, isr_change_notifi
cation, latest_producer_id_block, log_dir_event_notification, zookeeper]
[zk: 192.168.8.12:2181(CONNECTED) 1] ls /brokers
[ids, seqid, topics]
[zk: 192.168.8.12:2181(CONNECTED) 2] ls /brokers/ids
[1, 2, 3]
[zk: 192.168.8.12:2181(CONNECTED) 3] get /brokers/ids/3
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://192.168.8.14:90
92"],"jmx_port":-1,"features":{},"host":"192.168.8.14","timestamp":"1628738497907","port":9092,"versi
on":5}
[zk: 192.168.8.12:2181(CONNECTED) 4] ls /brokers/topics
[__consumer_offsets, kafkatest]
[zk: 192.168.8.12:2181(CONNECTED) 5] get /brokers/topics/kafkatest
{"removing_replicas":{},"partitions":{"0":[3,1]},"topic_id":"Gpj4bHJ_T0ORSMFOS6YPVg","adding_replicas
":{},"version":3}
[zk: 192.168.8.12:2181(CONNECTED) 6]
```

topic删除

```
命令格式
    kafka-topics.sh --zookeeper <host>:<port> --topic <副本名称> --delete

删除topic
    kafka-topics.sh --delete --zookeeper 192.168.8.12:2181 --topic kafkatest

检查效果
    [zk: 192.168.8.12:2181(CONNECTED) 6] ls /brokers/topics
    [__consumer_offsets]
```

小结

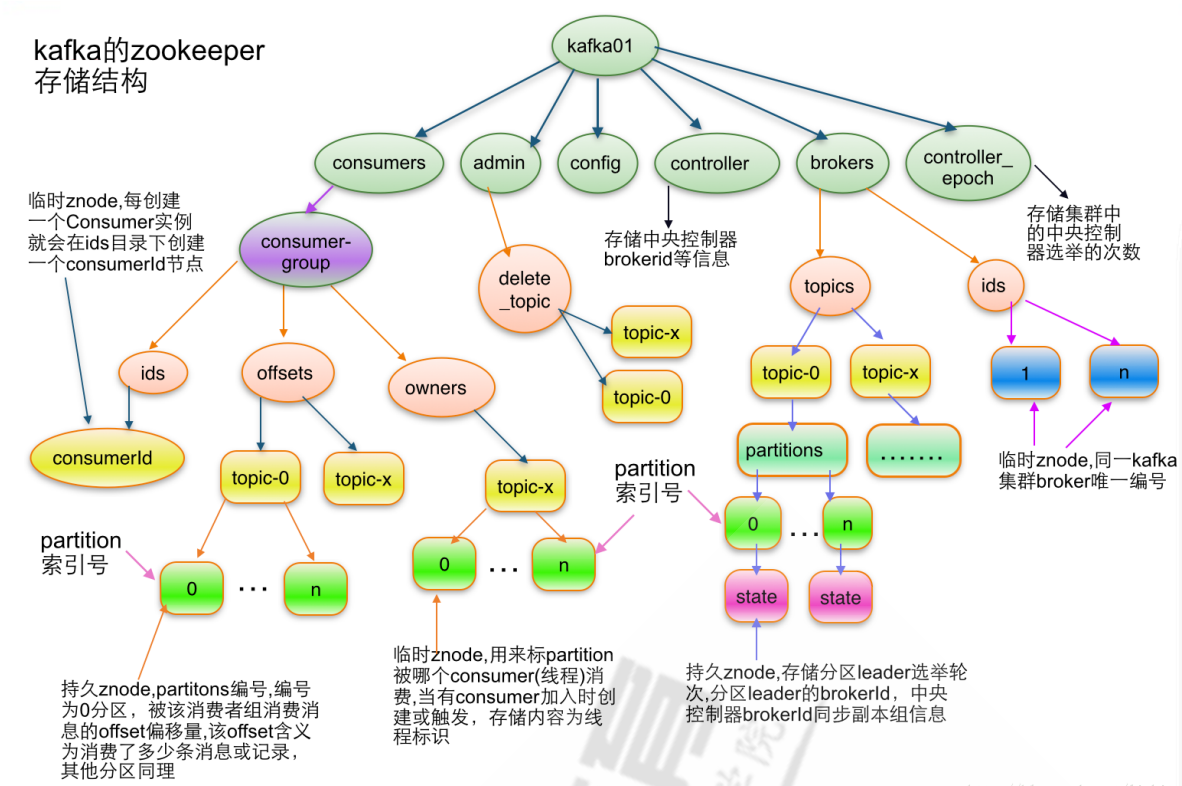
原理解析

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

存储结构



topic结构

```
帮助信息
    /brokers/topics/[topic]

简单实践
    ] ls /brokers
    [ids, seqid, topics]

    ] ls /brokers/topics
    [__consumer_offsets, kafkatest]

    ] get /brokers/topics/kafkatest
    {"removing_replicas":{},"partitions":{"0":
[3,1]},"topic_id":"x8qpqpH_QYycilmNhWOMNw","adding_replicas":{},"version":3}

属性解析
    version：  当前的版本号
    partitions：  分区的信息
```

partition结构

```
帮助信息
    /brokers/topics/[topic]/partitions/[partitionId]/state

命令演示
    ] ls /brokers/topics/kafkatest
    [partitions]

    ] ls /brokers/topics/kafkatest/partitions
    [0]

    ] get /brokers/topics/kafkatest/partitions/0
    null
```

```
    ] get /brokers/topics/kafkatest/partitions/0/state
    {"controller_epoch":7,"leader":3,"version":1,"leader_epoch":4,"isr":[3,1]}

属性解析
    controller_epoch: 表示kafka集群中的中央控制器选举次数,
    leader: 表示该partition选举leader的brokerId,
    version: 版本编号默认为1,
    leader_epoch: 该partition leader选举次数,
    isr: [同步副本组brokerId列表]
```

broker信息

```
帮助信息
    /brokers/ids/[0...N]

简单实践
    ] ls /brokers
    [ids, seqid, topics]

    ] ls /brokers/ids
    [1, 2, 3]

    ] get /brokers/ids/1
    {"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":
["PLAINTEXT://192.168.8.12:9092"],"jmx_port":-1,"features":
{},"host":"192.168.8.12","timestamp":"1628742173381","port":9092,"version":5}

属性解析:
    jmx_port: jmx端口号,
    timestamp: kafka broker初始启动时的时间戳,
    host: 主机名或ip地址,
    version: 版本编号默认为1,
    port: kafka broker的服务端端口号,由server.properties中参数port确定
```

控制器

```
帮助信息
    /controller
    存储center controller中央控制器所在kafka broker的信息

简单实践
    ] ls /controller
    []

    ] get /controller
    {"version":1,"brokerid":2,"timestamp":"1628741540481"}

属性解析
    version: 版本编号默认为1,
    brokerid: kafka集群中broker唯一编号,
    timestamp: kafka broker中央控制器变更时的时间戳
```

控制策略

```
帮助信息
    /controller_epoch
    此值为一个数字,每次center controller变更controller_epoch值就会 + 1

简单实践
] ls /controller_epoch
[]

] get /controller_epoch
7
```

消费者

```
帮助信息
    消费者信息:
        /consumers/[groupId]/ids/[consumerIdString]
        每个consumer都有一个唯一的ID,此id用来标记消费者信息
    消费者管理者:
        /consumers/[groupId]/owners/[topic]/[partitionId]


简单实践
```

流程梳理

发布消息

```
基本逻辑
    producer 采用 push 模式将消息发布到 broker，broker接收到消息，会根据分区算法选择将其存储到哪一个 partition，然后消息都被 append 到指定的 patition 中。

细节逻辑
    producer 先从 zookeeper 的 "/brokers/.../state" 节点找到该 partition 的 leader
    producer 将消息发送给该 leader
    leader 将消息写入本地 log
    followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK
    leader 收到所有 ISR 中的 replica 的 ACK 后，更新 commit 并向 producer 发送 ACK
```

broker保存消息

```
基本逻辑
    默认情况下，一个 topic 分成一个或多个 patition(num.partitions)，每个 patition 就是一个文件夹，保存了所有消息和索引文件。

    这些数据不会无限制的存储下去，会在某个时候自动清空，这是由以下两种配置决定的。
        基于时间：log.retention.hours=168
        基于大小：log.retention.bytes=1073741824

细节展示
    # ls kafka/logs | grep foo
    foobar-0
    foobar-2
```

```
    # ls kafka/logs/foobar-0
    00000000000000000000.index  00000000000000000000.timeindex
partition.metadata
    00000000000000000000.log    leader-epoch-checkpoint
```

topic逻辑

```
新增逻辑
    controller 在 ZK 的 /brokers/topics 节点上注册 watcher
        - 当 topic 被创建，则 controller 会通过 watch 得到该 topic 的
partition/replica 分配。
    controller从 /brokers/ids 读取当前所有可用的 broker 列表，对于 set_p 中的每一个
partition：
        - 从分配给该 partition 的所有 replica中任选一个可用的 broker 作为新的 leader，
并将AR设置为新的 ISR
        - 将新的 leader 和 ISR 写入
/brokers/topics/[topic]/partitions/[partition]/state
    controller 通过 RPC 向相关的 broker 发送 LeaderAndISRRequest。

删除逻辑
    controller 在 ZK 的 /brokers/topics 节点上注册 watcher，
        - 当 topic 被删除，则 controller 会通过 watch 得到该 topic 的
partition/replica 分配。
    若 delete.topic.enable=false，结束
        - 否则 controller 注册在 /admin/delete_topics 上的 watch 被 fire，
controller 通过回调向对应的 broker 发送 StopReplicaRequest。
```

broker故障

```
    controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker
宕机时 zookeeper 会 fire watch
    controller 从 /brokers/ids 节点读取可用broker
    controller决定set_p，该集合包含宕机 broker 上的所有 partition
    对 set_p 中的每一个 partition
        - 从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR
        - 决定新 leader
        - 将新 leader、ISR、controller_epoch 和 leader_epoch 等信息写入 state 节点
    通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令
```

将一个kafka节点故障，查看效果

```
tail -f kafka/logs/controller.log
```

```
root@python-auto:/data/server# kafka-topics.sh --describe --zookeeper 192.168.8.12:2181 --topic foobar
Topic: foobar   TopicId: WN5UJ2_TSJ-ZcYcpedbtLg PartitionCount: 3       ReplicationFactor: 2    Configs:
        Topic: foobar   Partition: 0    Leader: 3       Replicas: 3,1   Isr: 3,1
        Topic: foobar   Partition: 1    Leader: 1       Replicas: 1,2   Isr: 1,2
        Topic: foobar   Partition: 2    Leader: 2       Replicas: 2,3   Isr: 2,3
root@python-auto:/data/server# kafka-topics.sh --describe --zookeeper 192.168.8.12:2181 --topic foobar
Topic: foobar   TopicId: WN5UJ2_TSJ-ZcYcpedbtLg PartitionCount: 3       ReplicationFactor: 2    Configs:
        Topic: foobar   Partition: 0    Leader: 3       Replicas: 3,1   Isr: 3
        Topic: foobar   Partition: 1    Leader: 2       Replicas: 1,2   Isr: 2
        Topic: foobar   Partition: 2    Leader: 2       Replicas: 2,3   Isr: 2,3
```

python实践

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

模块简介

```
    在python的web项目中，我们需要基于 kafka 来实现数据的存储和获取。在python库中有一个功能模块 kafka-python，它可以作为 Apache Kafka 分布式流处理系统的 Python 客户端。

    kafka-python 最好与较新的代理（0.9+）一起使用，但向后兼容旧版本（到 0.8.0）。
```

模块简介

```
    kafka-python 提供了两个类来分别实现 数据推送 和 数据获取 两种场景。
    KafkaConsumer
        一个高级消息消费者，消费者迭代器返回 ConsumerRecords，它们是简单的命名元组
        公开基本消息属性：topic, partition, offset, key, value
    KafkaProducer
        一个高级的异步消息生产者，KafkaProducer 可以跨线程使用而不会出现问题。
```

环境准备

```
mkvirtualenv python3
pip install kafka-python
```

简单实践

简单实践1

```
终端1 - 基于ipython3的终端来发出请求
from kafka import KafkaProducer
producer = KafkaProducer(
bootstrap_servers=["192.168.8.12:9092", "192.168.8.13:9092",
"192.168.8.14:9092"]
)
producer.send("test",b"Hello Kafka")
    注意：
        不要用中文，会导致不识别，因为代码默认识别如下类型
        bytes, bytearray, memoryview, type(None)

终端2 - 基于ipython3的终端获取消息效果
from kafka import KafkaConsumer
consumer = KafkaConsumer(
bootstrap_servers = "192.168.8.12:9092,192.168.8.13:9092,192.168.8.14:9092",
group_id = "my.group",
enable_auto_commit = True,
auto_commit_interval_ms = 5000,
)
consumer.subscribe(["my.topic"])
for msg in consumer:
    print(msg)
```

场景实践2

```
--- 定制读取文件内容并且推送到kafka中
import json, time
from kafka import KafkaProducer
#生产的机器
producer =
KafkaProducer(bootstrap_servers='192.168.8.12:9092,192.168.8.13:9092,192.168.8.1
4:9092')

# 消息体
data = {"message":"test kafka data"}
#获取文件数据
with open("passwd", "r") as f:
    count = 0
    for line in f.readlines():
        #更新json数据
        count += 1
        data["Key"] = "ubuntu-passwd" + str(count)
        #传入文件变量
        data["Content"] = str(line)
        #格式为字符类型
        msg = json.dumps(data).encode()
        #写入消息
        producer.send('my.topic', msg)
        time.sleep(1)
print("文件读取完毕")

--- 定制从kafka中读取数据
from kafka import KafkaConsumer
consumer = KafkaConsumer(
    bootstrap_servers = "192.168.8.12:9092,192.168.8.13:9092,192.168.8.14:9092",
    group_id = "my.group",
    enable_auto_commit = True,
    auto_commit_interval_ms = 5000,
)
consumer.subscribe(["my.topic"])
for msg in consumer:
    print(msg)
```

小结