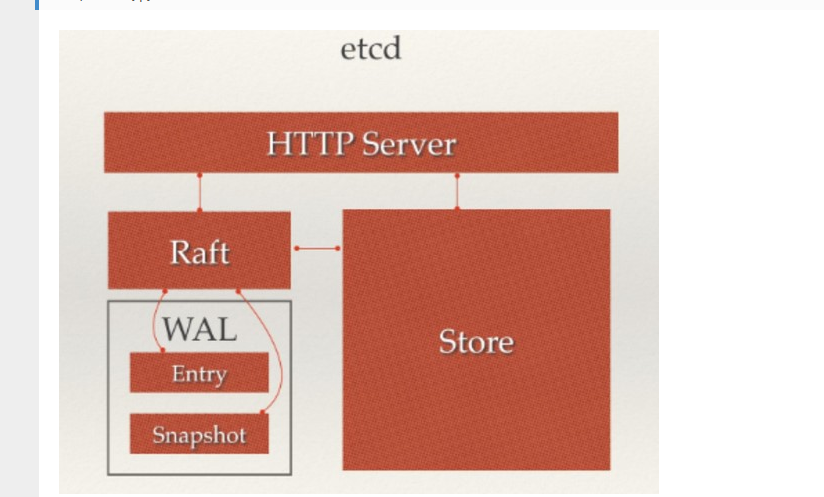
etcd是一个分布式的，一致的 key-value 存储，主要用途是共享配置和服务发现



Etcd 主要分为4部分

**HTTP Server**

**用于处理用户发送的API 请求 以及其他etcd节点的同步和心跳信息**

**Store**

#### 用于处理etcd支持的各类功能的事务，包括数据索引，节点状态变更，监控与反馈，事件处理与执行等等，是etcd对于用户提供的大多数API的实现

#### Raft

#### Raft强一致性算法的具体实现，是etcd的核心

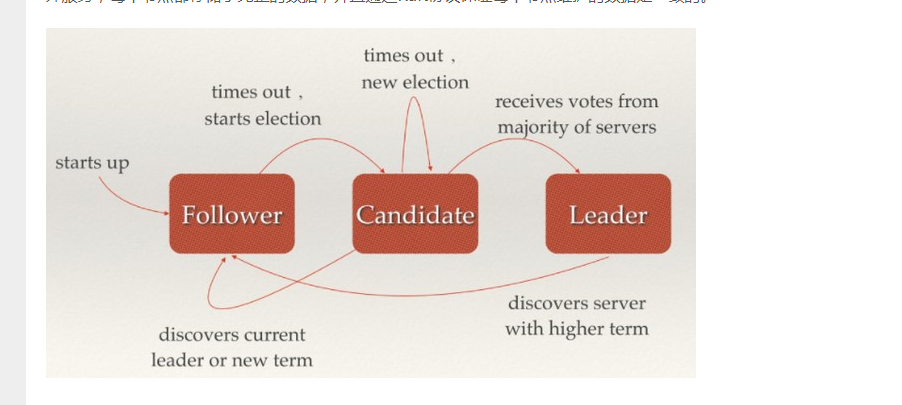
#### WAL

#### Write Ahead Log（预写式日志），是 etcd 的数据存储方式。除了在内存中存有所有数据的状态以及节点的索引以外，etcd 就通过 WAL 进行持久化存储。WAL 中， 所有的数据提交前都会事先记录日志。Snapshot 是为了防止数据过多而进行的状态快照；Entry 表示存储的具体日志内容。

#### 通常，一个用户的请求发送过来，会经由 HTTP Server 转发给 Store 进行具体的事务处理，如果涉及到节点的修改，则交给 Raft 模块进行状态的变更、日志的记录，然后再同步给别的 etcd节点以确认数据提交，最后进行数据的提交，再次同步

## etcd工作原理（Raft协议）

etcd使用Raft协议来维护集群内各个节点状态的一致性。简单说，etcd集群是一个分布式系统，由多个节点相互通信构成整体对外服务，每个节点都存储了完整的数据，并且通过Raft协议保证每个节点维护的数据是一致的。



如上图所示，每个etcd节点都维护了一个状态机，并且，任意时刻至多存在一个有效的主节点。主节点处理所有来自客户端写操作，通过Raft协议保证写操作对状态机的改动会可靠的同步到其他节点

etcd工作原理核心部分在于Raft协议。本节接下来将简要介绍Raft协议，   
Raft协议正如论文所述，确实方便理解。主要分为三个部分：选主，日志复制，安全性

当集群初始化时候，每个节点都是Follower角色；   
集群中存在至多1个有效的主节点，通过心跳与其他节点同步数据；   
当Follower在一定时间内没有收到来自主节点的心跳，会将自己角色改变为Candidate，并发起一次选主投票；当收到包括自己在内超过半数节点赞成后，选举成功；当收到票数不足半数选举失败，或者选举超时。若本轮未选出主节点，将进行下一轮选举（出现这种情况，是由于多个节点同时选举，所有节点均为获得过半选票）。   
Candidate节点收到来自主节点的信息后，会立即终止选举过程，进入Follower角色

为了避免陷入选主失败循环，每个节点未收到心跳发起选举的时间是一定范围内的随机值，这样能够避免2个节点同时发起选主

#### Raft协议-日志复制

所谓日志复制，是指主节点将每次操作形成日志条目，并持久化到本地磁盘，然后通过网络IO发送给其他节点。其他节点根据日志的逻辑时钟(TERM)和日志编号(INDEX)来判断是否将该日志记录持久化到本地。当主节点收到包括自己在内超过半数节点成功返回，那么认为该日志是可提交的(committed），并将日志输入到状态机，将结果返回给客户端

#### 这里需要注意的是，每次选主都会形成一个唯一的TERM编号，相当于逻辑时钟。每一条日志都有全局唯一的编号

主节点通过网络IO向其他节点追加日志。若某节点收到日志追加的消息，首先判断该日志的TERM是否过期，以及该日志条目的INDEX是否比当前以及提交的日志的INDEX跟早。若已过期，或者比提交的日志更早，那么就拒绝追加，并返回该节点当前的已提交的日志的编号。否则，将日志追加，并返回成功。

当主节点收到其他节点关于日志追加的回复后，若发现有拒绝，则根据该节点返回的已提交日志编号，发生其编号下一条日志

主节点像其他节点同步日志，还作了拥塞控制。具体地说，主节点发现日志复制的目标节点拒绝了某次日志追加消息，将进入日志探测阶段，一条一条发送日志，直到目标节点接受日志，然后进入快速复制阶段，可进行批量日志追加。

按照日志复制的逻辑，我们可以看到，集群中慢节点不影响整个集群的性能。另外一个特点是，数据只从主节点复制到Follower节点，这样大大简化了逻辑流程

#### Raft协议-安全性

截止此刻，选主以及日志复制并不能保证节点间数据一致。试想，当一个某个节点挂掉了，一段时间后再次重启，并当选为主节点。而在其挂掉这段时间内，集群若有超过半数节点存活，集群会正常工作，那么会有日志提交。这些提交的日志无法传递给挂掉的节点。当挂掉的节点再次当选主节点，它将缺失部分已提交的日志。在这样场景下，按Raft协议，它将自己日志复制给其他节点，会将集群已经提交的日志给覆盖掉。

这显然是不可接受的。

其他协议解决这个问题的办法是，新当选的主节点会询问其他节点，和自己数据对比，确定出集群已提交数据，然后将缺失的数据同步过来。这个方案有明显缺陷，增加了集群恢复服务的时间（集群在选举阶段不可服务），并且增加了协议的复杂度。

Raft解决的办法是，在选主逻辑中，对能够成为主的节点加以限制，确保选出的节点已定包含了集群已经提交的所有日志。如果新选出的主节点已经包含了集群所有提交的日志，那就不需要从和其他节点比对数据了。简化了流程，缩短了集群恢复服务的时间。

这里存在一个问题，加以这样限制之后，还能否选出主呢？答案是：只要仍然有超过半数节点存活，这样的主一定能够选出。因为已经提交的日志必然被集群中超过半数节点持久化，显然前一个主节点提交的最后一条日志也被集群中大部分节点持久化。当主节点挂掉后，集群中仍有大部分节点存活，那这存活的节点中一定存在一个节点包含了已经提交的日志了。

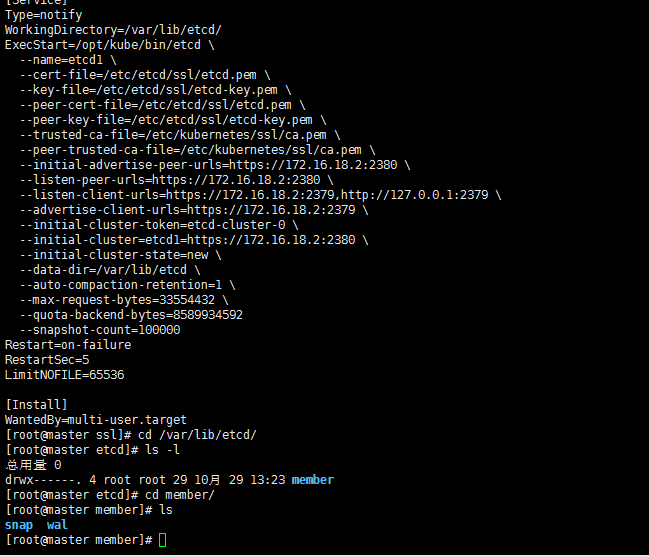
至此，关于Raft协议的简介就全部结束了。

## etcd 数据存储

etcd 的存储分为 内存存储 和 持久化（硬盘）存储 两部分。

内存中的存储除了顺序化的记录下所有用户对节点数据变更的记录外，还会对用户数据进行索引、建堆等方便查询的操作。

而持久化则使用预写式日志（WAL：Write Ahead Log）进行记录存储。



在 WAL 的体系中，所有的数据在提交之前都会进行日志记录。在 etcd 的持久化存储目录中，有两个子目录。一个是 WAL，存储着所有事务的变化记 录；另一个则是 snapshot，用于存储某一个时刻 etcd 所有目录的数据。通过 WAL 和 snapshot 相结合的方式，etcd 可以有效的进行数据存储和节点故障恢复等操作。

既然有了 WAL 实时存储了所有的变更，为什么还需要 snapshot 呢？随着使用量的增加，WAL 存储的数据会暴增，为了防止磁盘很快就爆满，etcd 默认每 10000 条记录做一次 snapshot，经过 snapshot 以后的 WAL 文件就可以删除。而通过 API 可以查询的历史 etcd 操作默认为 1000 条。

首次启动时，etcd 会把启动的配置信息存储到data-dir参数指定的数据目录中。配置信息包括本地节点的 ID、集群 ID 和初始时集群信息。用户需要避免 etcd 从一个过期的数据目录中重新启动，因为使用过期的数据目录启动的节点会与集群中的其他节点产生不一致（如： 之前已经记录并同意 Leader 节点存储某个信息，重启后又向 Leader 节点申请这个信息）。所以，为了最大化集群的安全性，一旦有任何数据损坏或丢失 的可能性，你就应该把这个节点从集群中移除，然后加入一个不带数据目录的新节点。



## 部署etcd

etcd 一般部署集群推荐奇数个节点，推荐的数量为 3、5 或者 7 个节点构成一个集群。

etcd安装很简单，直接从github下载，无需编译，或者yum安装都可以

#@yum install from epel

# yum install etcd -y

# etcd --version

etcd Version: 3.2.9

Git SHA: f1d7dd8

Go Version: go1.8.3

Go OS/Arch: linux/amd64

Etcd 3.X的版本没有配置文件，也不能指定配置文件，所有选项都可以使用命令行参数或者环境变量来传递。

所以在CentOS7中直接编辑启动脚本就可以了。

# cat /usr/lib/systemd/system/etcd.service

[Unit]

Description=Etcd Server

After=network.target

After=network-online.target

Wants=network-online.target

[Service]

Type=notify

WorkingDirectory=/var/lib/etcd/

User=etcd

# set GOMAXPROCS to number of processors

ExecStart=/bin/bash -c "GOMAXPROCS=$(nproc) /usr/bin/etcd \

--name infra1 \

--listen-peer-urls http://10.140.100.12:2380 \

--listen-client-urls http://10.140.100.12:2379,http://127.0.0.1:2379 \

--initial-advertise-peer-urls http://10.140.100.12:2380 \

--advertise-client-urls http://10.140.100.12:2379 \

--initial-cluster-token etcd-cluster \

--initial-cluster-state new \

--initial-cluster infra1=http://10.140.100.12:2380,infra2=http://10.140.100.13:2380,infra3=http://10.140.100.15:2380 \

--data-dir=/var/lib/etcd/ "

Restart=on-failure

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

[root@idc-sm-rd-k8s-1 ~]# etcdctl cluster-health

member 27683ef1237cf56 is healthy: got healthy result from http://10.140.100.15:2379

member 2ce5820bdb490d73 is healthy: got healthy result from http://10.140.100.13:2379

member 534827736701c1b8 is healthy: got healthy result from http://10.140.100.12:2379

cluster is healthy

[root@idc-sm-rd-k8s-1 ~]#

[root@idc-sm-rd-k8s-1 ~]# etcdctl member list

27683ef1237cf56: name=infra3 peerURLs=http://10.140.100.15:2380 clientURLs=http://10.140.100.15:2379 isLeader=false

2ce5820bdb490d73: name=infra2 peerURLs=http://10.140.100.13:2380 clientURLs=http://10.140.100.13:2379 isLeader=false

534827736701c1b8: name=infra1 peerURLs=http://10.140.100.12:2380 clientURLs=http://10.140.100.12:2379 isLeader=true

[root@idc-sm-rd-k8s-1 ~]#

[root@idc-sm-rd-k8s-1 ~]#etcdctl set /foo/bar "hello world"

hello world

[root@idc-sm-rd-k8s-1 ~]# etcdctl get /foo/bar

hello world

[root@idc-sm-rd-k8s-1 ~]# etcdctl ls /foo

/foo/bar

[root@idc-sm-rd-k8s-1 ~]#

<https://www.ipcpu.com/2017/09/etcd-start/>

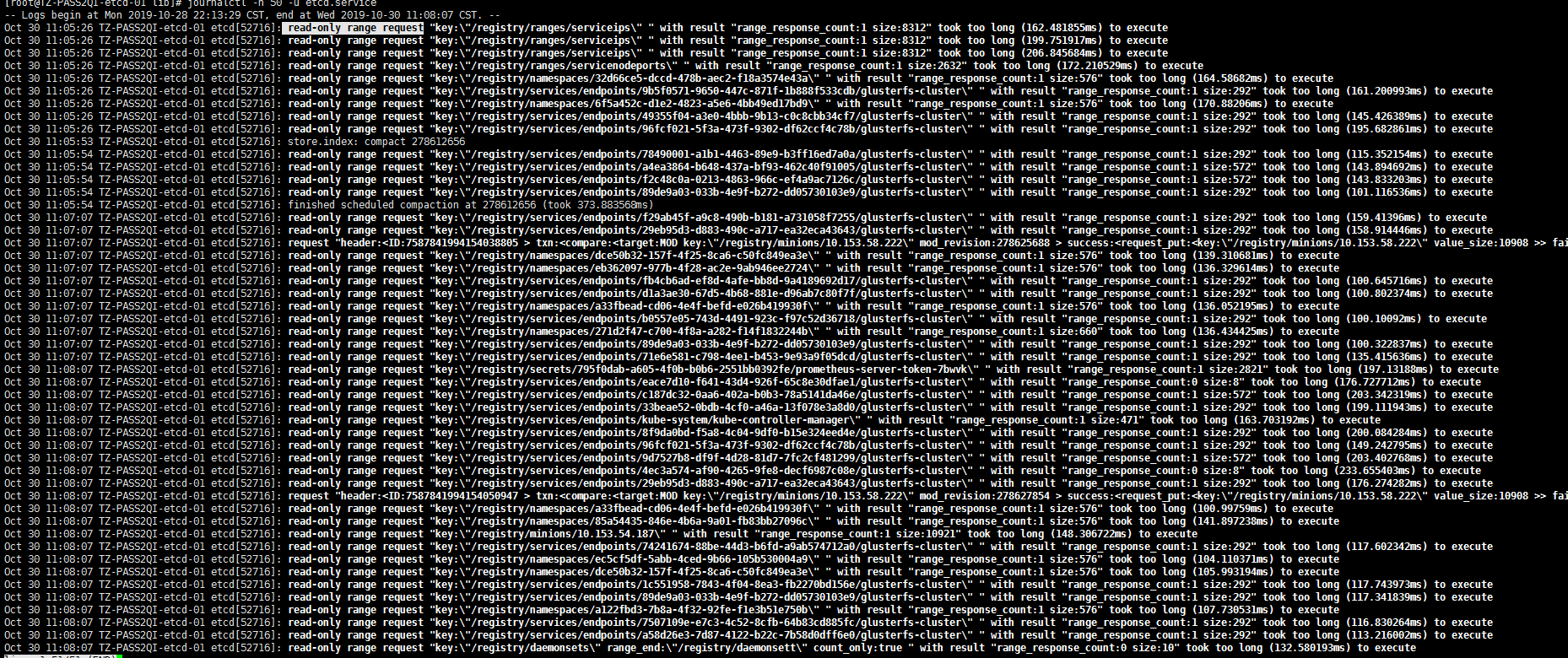
### etcd 概念词汇表

* Raft：etcd 所采用的保证分布式系统强一致性的算法。
* Node：一个 Raft 状态机实例。
* Member： 一个 etcd 实例。它管理着一个 Node，并且可以为客户端请求提供服务。
* Cluster：由多个 Member 构成可以协同工作的 etcd 集群。
* Peer：对同一个 etcd 集群中另外一个 Member 的称呼。
* Client： 向 etcd 集群发送 HTTP 请求的客户端。
* WAL：预写式日志，etcd 用于持久化存储的日志格式。
* snapshot：etcd 防止 WAL 文件过多而设置的快照，存储 etcd 数据状态。
* Proxy：etcd 的一种模式，为 etcd 集群提供反向代理服务。
* Leader：Raft 算法中通过竞选而产生的处理所有数据提交的节点。
* Follower：竞选失败的节点作为 Raft 中的从属节点，为算法提供强一致性保证。
* Candidate：当 Follower 超过一定时间接收不到 Leader 的心跳时转变为 Candidate 开始竞选。
* Term：某个节点成为 Leader 到下一次竞选时间，称为一个 Term。
* Index：数据项编号。Raft 中通过 Term 和 Index 来定位数据。

**journalctl \_SYSTEMD\_UNIT=etcd.service --since "2019-10-29 10:00" --until "2019-10-29 11:00" 查看指定事件段的日志**

**journalctl -n 200 -u etcd.service 查看最后200条**

**现象**



### Etcd端口问题

Etcd默认占用端口 2379

2380 http端口

4001 或者 7001 peer之间的通信

### What does the etcd warning "apply entries took too long" mean?

[**https://github.com/etcd-io/etcd/blob/master/Documentation/faq.md#what-does-the-etcd-warning-apply-entries-took-too-long-mean**](https://github.com/etcd-io/etcd/blob/master/Documentation/faq.md#what-does-the-etcd-warning-apply-entries-took-too-long-mean) **官网解释**

此问题是由磁盘速度慢引起的 磁盘可能正在etcd和其他应用程序之间争用，或者磁盘太慢

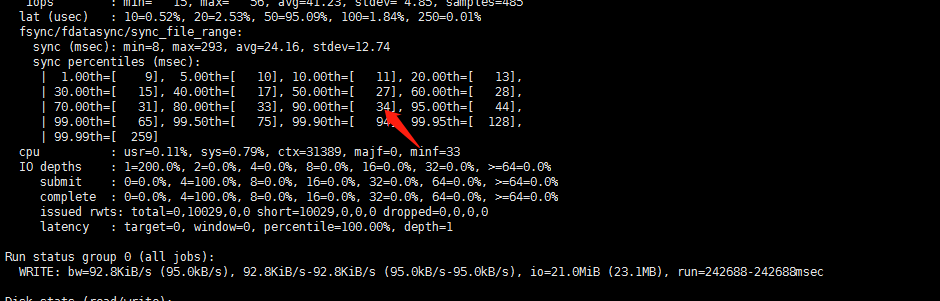
backend\_commit\_duration\_seconds 通过graph监控

后端调用的提交的延迟分布

Etcd 磁盘要求

<https://github.com/etcd-io/etcd/blob/master/Documentation/faq.md#system-requirements>

磁盘问题 通过fio



[**https://www.ibm.com/cloud/blog/using-fio-to-tell-whether-your-storage-is-fast-enough-for-etcd**](https://www.ibm.com/cloud/blog/using-fio-to-tell-whether-your-storage-is-fast-enough-for-etcd)

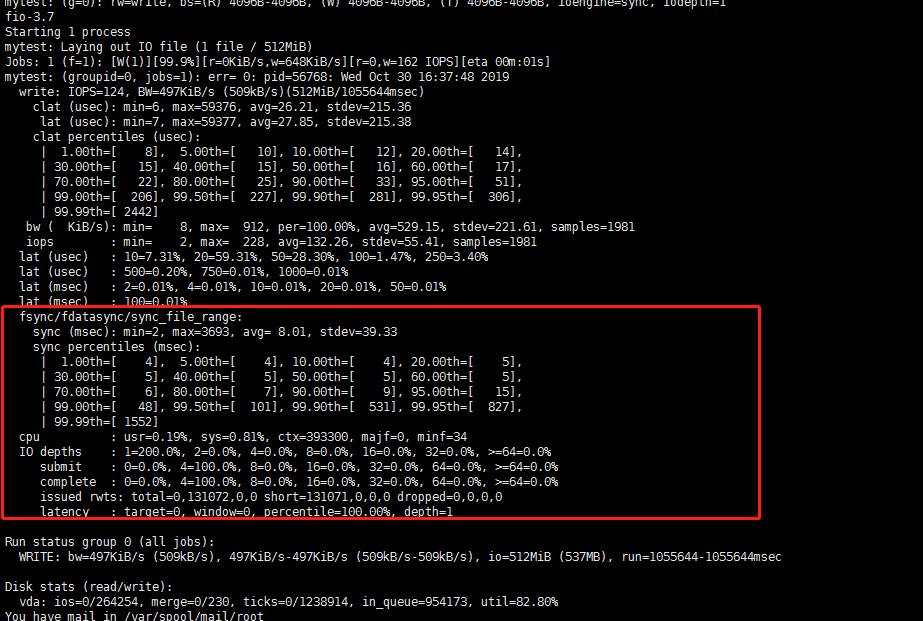
etcd强制将可配置的存储大小配额默认设置为2GB。为避免交换或内存不足，计算机应至少具有足够的RAM来覆盖配额。建议在正常环境下使用8GB的最大大小，如果配置的值超过该值，etcd将在启动时发出警告。在CoreOS上，通常在具有双核处理器，至少2GB RAM和80GB SSD的专用CoreOS Container Linux机器上部署etcd群

he second most common cause is CPU starvation. If monitoring of the machine’s CPU usage shows heavy utilization, there may not be enough compute capacity for etcd. Moving etcd to dedicated machine, increasing process resource isolation cgroups, or renicing the etcd server process into a higher priority can usually solve the problem.

Expensive user requests which access too many keys (e.g., fetching the entire keyspace) can also cause long apply latencies. Accessing fewer than a several hundred keys per request, however, should always be performant.

If none of the above suggestions clear the warnings, please [open an issue](https://github.com/etcd-io/etcd/issues/new) with detailed logging, monitoring, metrics and optionally workload information.

**fio --rw=write --ioengine=sync --fdatasync=1 --directory=test-data --size=512M --bs=4k --name=mytest**



<https://bingohuang.com/etcd-monitor-guides/>

<https://github.com/etcd-io/etcd/blob/master/Documentation/faq.md#system-requirements> etcd常见问题集锦

2380 和集群中其他节点通信

2379 提供 HTTP API 服务，供客户端交互

--name

方便理解的节点名称，默认为 default

在集群中应该保持唯一，可以使用 hostname

--data-dir

服务运行数据保存的路径，默认为 ${name}.etcd

--snapshot-count

指定有多少事务（transaction）被提交时，触发截取快照保存到磁盘

* --heartbeat-interval  
  ：leader 多久发送一次心跳到 followers。默认值是 100ms
* --eletion-timeout  
  ：重新投票的超时[时间](http://www.liuhaihua.cn/archives/tag/%e6%97%b6%e9%97%b4" \t "_blank" \o "View all posts in 时间)，如果 follow 在该时间间隔没有收到心跳包，会触发重新投票，默认为 1000 ms

--listen-peer-urls

和同伴通信的地址，比如 <http://ip:2380>

--listen-peer-urls=https://10.153.58.209:2380

--listen-client-urls

--listen-client-urls=https://10.153.58.209:2379,http://127.0.0.1:2379

对外提供服务的地址：比如 <http://ip:2379,http://127.0.0.1:2379>

客户端会连接到这里和 etcd 交互

--advertise-client-urls

--advertise-client-urls=https://10.153.58.209:2379

对外公告的该节点客户端监听地址，这个值会告诉集群中其他节点

--initial-advertise-peer-urls

--initial-advertise-peer-urls=https://10.153.58.209:2380

对外公告的该节点客户端监听地址，这个值会告诉集群中其他节点

--initial-cluster

集群中所有节点的信息，格式为 node1=http://ip1:2380,node2=http://ip2:2380,…

注意：这里的 node1  
是节点的 --name  
指定的名字；后面的 ip1:2380  
是 --initial-advertise-peer-urls  
指定的值

--initial-cluster-state

：新建集群的时候，这个值为 new

假如已经存在的集群，这个值为 existing

--initial-cluster-token

创建集群的 token，这个值每个集群保持唯一。这样的话，如果你要重新创建集群，即使配置和之前一样，也会再次生成新的集群和节点 uuid；否则会导致多个集群之间的冲突，造成未知的错误

所有以 --init  
开头的配置都是在 bootstrap 集群的时候才会用到，后续节点的重启会被忽略

NOTE：所有的参数也可以通过环境变量进行设置， --my-flag  
对应环境变量的 ETCD\_MY\_FLAG  
；但是命令行指定的参数会覆盖环境变量对应的值。

在后面的文章中，为了简单起见，我们采用了单点的 etcd server， **请在生产环境中配置 etcd 集群，并使用 SSL 安全机制**

Etcd基础知识

每个 etcd cluster 都是有若干个 member 组成的，每个 member 是一个独立运行的 etcd 实例，单台机器上可以运行多个 member。

在正常运行的状态下，集群中会有一个 leader，其余的 member 都是 followers。leader 向 followers [同步](http://www.liuhaihua.cn/archives/tag/%e5%90%8c%e6%ad%a5)日志，保证数据在各个 member 都有副本。leader 还会定时向所有的 member 发送心跳报文，如果在规定的时间里 follower 没有收到心跳，就会重新进行选举。

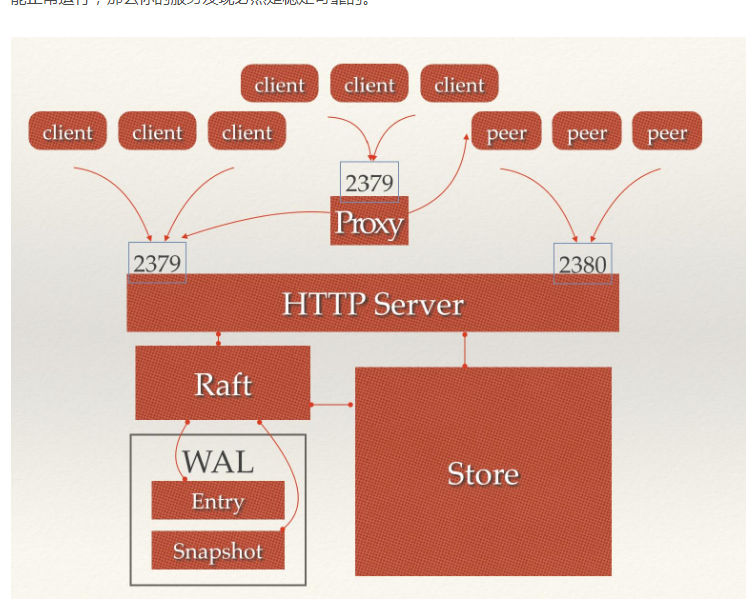
客户端所有的请求都会先发送给 leader，leader 向所有的 followers 同步日志，等收到超过半数的确认后就把该日志存储到磁盘，并返回响应客户端。

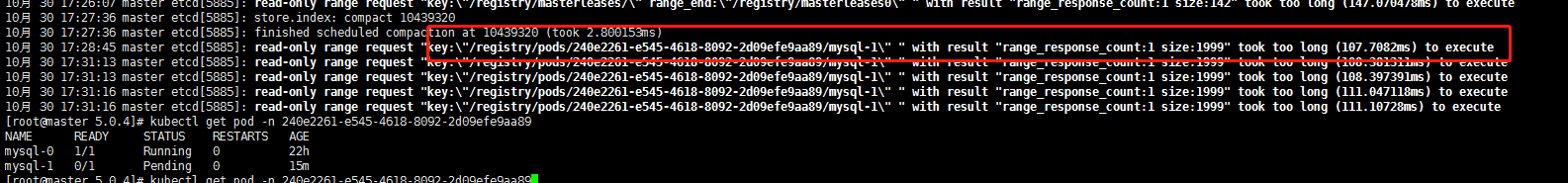
每个 etcd 服务有三大主要部分组成：raft 实现、WAL 日志存储、数据的存储和索引。WAL 会在本地磁盘（就是之前提到的--data-dir  
）上存储日志内容（wal file）和快照（snapshot）

安装和启动 etcd 服务的时候，各个节点需要知道集群中其他节点的信息（一般是 ip 和 port 信息）。根据你是否可以提前知道每个节点的 ip，有几种不同的启动方案

静态配置：在启动 etcd server 的时候，通过 --initial-cluster

Etcd 架构图





**解释**既然有了 WAL 实时存储了所有的变更，为什么还需要 snapshot 呢？随着使用量的增加，WAL 存储的数据会暴增，为了防止磁盘很快就爆满，etcd 默认每 10000 条记录做一次 snapshot，经过 snapshot 以后的 WAL 文件就可以删除。而通过 API 可以查询的历史 etcd 操作默认为 1000 条

#### 预写式日志（WAL）

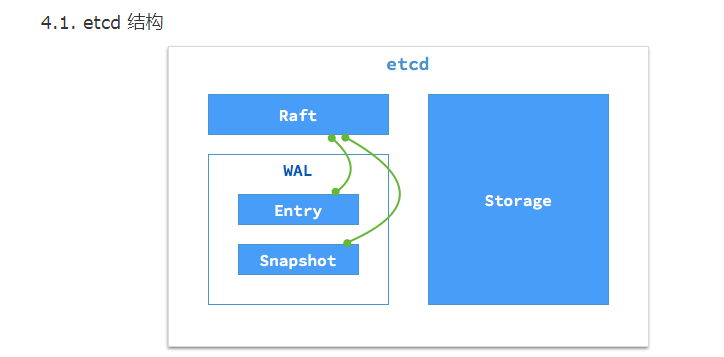
WAL（Write Ahead Log）最大的作用是记录了整个数据变化的全部历程。在 etcd 中，所有数据的修改在提交前，都要先写入到 WAL 中。使用 WAL 进行数据的存储使得 etcd 拥有两个重要功能。

* **故障快速恢复**： 当你的数据遭到破坏时，就可以通过执行所有 WAL 中记录的修改操作，快速从最原始的数据恢复到数据损坏前的状态。
* **数据回滚（undo）/ 重做（redo）**：因为所有的修改操作都被记录在 WAL 中，需要回滚或重做，只需要方向或正向执行日志中的操作即可。



<https://www.infoq.cn/article/coreos-analyse-etcd/>

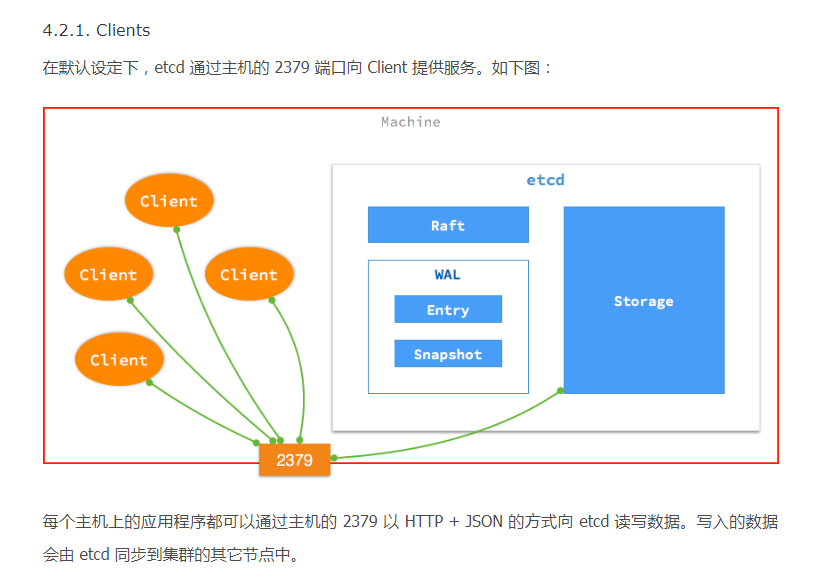
<https://www.infoq.cn/article/etcd-interpretation-application-scenario-implement-principle/>

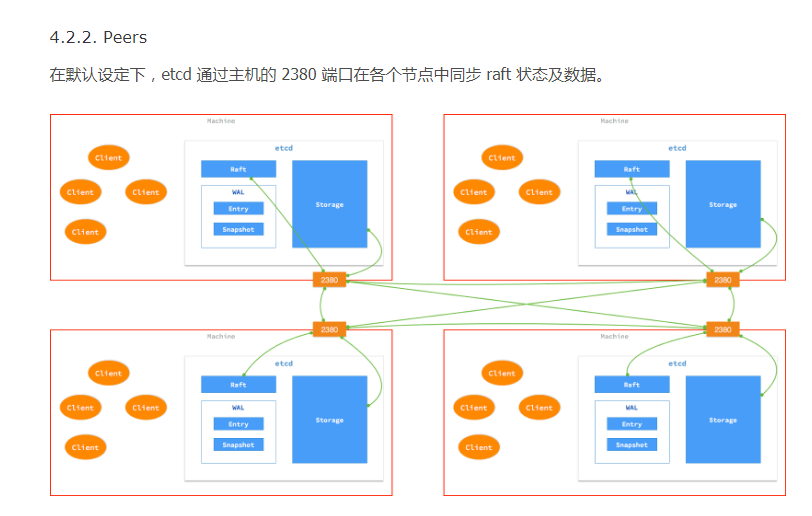


一个 etcd 节点的核心由三部分组成

* Raft：raft 状态机是对 raft 共识算法的实现
* WAL：raft 日志存储
* Storage：数据的存储与索引

WAL (Write-ahead logging)，是用于向系统提供原子性和持久性的一系列技术。在使用 WAL 的系提供中，所有的修改在提交之前都要先写入 log 文件中。etcd 的 WAL 由日志存储与快照存储两部分组成，其中 Entry 负责存储具体日志的内容，而 Snapshot 负责在日志内容发生变化的时候保存 raft 的状态。WAL 会在本地磁盘的一个指定目录下分别日志条目与快照内容





## 5. 创建

从方法上来划分，创建 etcd 集群的方式分为两种：Static （通过制定 peers 的 IP 和端口创建）与 Discovery （通过一个发现服务创建）。

Static 方式需要预先知道集群所有节点的 IP，所以适合小规模的集群或者搭建一个临时的开发与测试环境。

Discovery 方式不需要预先了解其他节点的 IP。启动时 etcd 通过访问一个 Discovery URL 来注册自己并获取其他节点的信息。这种方式通常适合将 etcd 部署在某个云服务平台或是一个 DHCP 环境中。其中 Discovery 服务可以使用 CoreOS 提供的一个公共地址 https://discovery.etcd.io/new 来申请一个 token，或者自己搭建一个这样的服务并设定一个 token。出于安全的考虑，这个 token 应该只在集群初始引导时短暂存在，因为集群建立之后将不再需要这一地址，而集群中节点的变更可以通过 etcd 运行时重构的能力来进行配置。

**Etcd 搭建集群**<https://www.cnblogs.com/yuhuLin/p/11046822.html>

<https://ywnz.com/linuxysjk/4269.html> 安装 **Prometheus**

**优化背景:**

由于阿里巴巴内部集群规模大，这种方案虽然解决了数据存储容量的问题，但是弊端也是比较明显的，由于 proxy 需要将数据进行搬移，因此操作的延时比原生存储大了很多。除此之外，由于多了 tair 这个组件，运维和管理成本较高。因此我们就想到底是什么原因限制了 etcd 的存储容量，我们是否可以通过技术手段优化解决呢？

提出了如上问题后我们首先进行了压力测试不停地像 etcd 中注入数据，当 etcd 存储数据量超过 40GB 后，经过一次 compact(compact 是 etcd 将不需要的历史版本数据删除的操作) 后发现 put 操作的延时激增，很多操作还出现了超时。监控发现 boltdb 内部 spill 操作 (具体定义见下文) 耗时显著增加 (从一般的 1ms 左右激增到了 8s)。之后经过反复多次压测都是如此，每次发生 compact 后，就像世界发生了停止，所有 etcd 读写操作延时比正常值高了几百倍，根本无法使用。

经过一次 compact(compact 是 etcd 将不需要的历史版本数据删除的操作) 后发现 put 操作的延时激增，很多操作还出现了超时

做etcd数据镜像的时候出现如下错误  Error: etcdserver: mvcc: database space exceeded

**[root@master sh]# ETCDCTL\_API=3 etcdctl --endpoints=https://172.16.18.2:2379 --cacert=/etc/kubernetes/ssl/ca.pem --cert=/etc/etcd/ssl/etcd.pem --key=/etc/etcd/ssl/etcd-key.pem alarm list**

**2019-11-01 09:38:24.134009 W | pkg/flags: unrecognized environment variable ETCDCTL\_CA\_FILE=/etc/kubernetes/ssl/ca.pem**

**2019-11-01 09:38:24.389896 W | pkg/flags: unrecognized environment variable ETCDCTL\_KEY\_FILE=/etc/etcd/ssl/etcd-key.pem**

**2019-11-01 09:38:24.406891 W | pkg/flags: unrecognized environment variable ETCDCTL\_CERT\_FILE=/etc/etcd/ssl/etcd.pem**

**memberID:8594911445404878656 alarm:NOSPACE**

<https://github.com/etcd-io/etcd/tree/master/Documentation/op-guide> etcd官方文档

cpu要求

**很少有etcd部署需要大量的CPU能力。 典型的集群需要2到4个核心才能平稳运行。 每秒处理数千个客户端或成千上万个请求的重载etcd部署往往受CPU限制，因为etcd可以处理来自内存的请求。 如此繁重的部署通常需要八到十六个专用内核。**

**内存**

**etcd的内存占用相对较小，但其性能仍然取决于是否有足够的内存。 etcd服务器将积极缓存键值数据，并花费其其余大部分内存跟踪监视程序。 通常8GB就足够了。 对于具有数千名观察者和数百万个密钥的繁重部署，请相应地将16GB分配给64GB内存。**

**磁盘:**

**快速磁盘是etcd部署性能和稳定性的最关键因素。**

**缓慢的磁盘会增加etcd请求的延迟，并有可能损害群集的稳定性。由于etcd的共识协议依赖于将元数据持久存储到日志中，因此大多数etcd集群成员必须将每个请求写入磁盘。此外，ETCD也将逐步检查点的状态到磁盘上，因此可以截断该日志。如果这些写入花费的时间太长，则心跳可能会超时并触发选举，从而破坏了群集的稳定性。通常，要判断磁盘是否足够快用于etcd，可以使用诸如fio之类的基准测试工具。阅读此处的示例。**

**etcd对磁盘写入延迟非常敏感。通常需要50个连续的IOPS（例如7200 RPM磁盘）。对于负载较重的群集，建议使用500个顺序IOPS（例如，典型的本地SSD或高性能虚拟化块设备）。请注意，大多数云提供商都发布并发IOPS，而不是顺序IOPS。已发布的并发IOPS可以比顺序IOPS大10倍。为了测量实际的顺序IOPS，我们建议使用磁盘基准测试工具，例如diskbench或fio。**

**etcd仅需要适度的磁盘带宽，但是当故障成员必须赶上集群时，更多的磁盘带宽可以缩短恢复时间。通常，每秒10MB /秒将在15秒内恢复100MB数据。对于大的集群，100MB / s或更高，建议为15秒内恢复1GB的数据。**

**如果可能的话，请使用SSD恢复etcd的存储。与旋转磁盘相比，SSD通常提供较低的写入延迟和较小的方差，从而提高了etcd的稳定性和可靠性。如果使用旋转磁盘，请获得最快的磁盘（15,000 RPM）。对于旋转磁盘和SSD，使用RAID 0也是提高磁盘速度的有效方法。对于至少三个集群成员，不需要RAID的镜像和/或奇偶校验变体； etcd的一致复制已经获得了高可用性。**

**压测磁盘etcd** <https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/performance.md>

由于etcd会保留其键空间的确切历史记录，因此应定期压缩此历史记录，以避免性能下降和最终的存储空间耗尽。压缩键空间历史记录会删除有关在给定键空间修订之前被取代的键的所有信息。然后，这些键使用的空间可用于对键空间的其他写入。

可以使用etcd的时间窗口历史记录保留策略自动压缩键空间，或者使用etcdctl手动压缩键空间。 etcdctl方法提供了对压缩过程的细粒度控制，而自动压缩适合仅需要关键历史记录一段时间的应用程序

可以将etcd设置为使用--auto-compaction- \*选项在几个小时内自动压缩键空间

可以使用etcd的时间窗口历史记录保留策略自动压缩键空间，或者使用etcdctl手动压缩键空间。 etcdctl方法提供了对压缩过程的细粒度控制，而自动压缩适合仅需要关键历史记录一段时间的应用程序

带有--auto-compaction-retention = 10的v3.0.0和v3.1.0在v3键值存储上每10小时运行一次定期压缩。压缩器仅支持定期压缩。 Compactor每5分钟记录一次最新修订，直到达到第一个压缩期（例如10小时）。为了保留上次压缩期的键值历史记录，它使用了每5分钟收集一次的修订记录中压缩期之前获取的最新修订。当--auto-compaction-retention = 10时，compactor使用修订版100进行压缩修订，其中修订版100是从10小时前获取的最新修订版。如果压缩成功或请求的修订已被压缩，它将重置周期计时器并以新的历史修订记录重新开始（例如，在下一个10个小时内重新启动修订收集并压缩）。如果压缩失败，它将在5分钟后重试。

<https://github.com/etcd-io/etcd/blob/master/Documentation/tuning.md>

[当 K8s 集群达到万级规模，阿里巴巴如何解决系统各组件性能问题？](https://blog.csdn.net/alex_yangchuansheng/article/details/102493816)

<https://cloud.tencent.com/developer/article/1520057>

将

**配置推荐**

<https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/configuration.md#configuration-flags>

如果 Etcd 的内存使用和磁盘使用过高，那么应该尝试调低快照触发的阈值，具体请参考如下命令。

如果将etcd用作服务发现，每次服务注册和更新都可以看做一条新数据，日积月累，这些数据的量会导致etcd占用内存越来越大，直到etcd到达空间配额限制的时候，etcd的写入将会被静止，影响线上服务，定期删除历史记录就是避免这种情况。

|  |  |
| --- | --- |
| 1 2 | *# 只保留一个小时的历史数据* $ etcd --auto-compaction-retention=1 |

### 使用磁盘defragmentation

<https://www.ibm.com/support/knowledgecenter/SSBS6K_3.2.0/manage_cluster/manage_etcd_clusters.html>

**Github 调优**<https://songrgg.github.io/etcd/etcd-for-production/>

etcdctl member add myetcd4 http:*//192.168.9.101:2383*

添加新节点

<https://blog.csdn.net/kunyus/article/details/88549755>

<https://blog.csdn.net/yunlilang/article/details/79726424>

<https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/runtime-configuration.md#add-a-new-member>