**Etcd**

**一、基础知识**

**1 etcd作为一个受到ZooKeeper与doozer启发而催生的项目**，除了拥有与之类似的功能外，更专注于以下四点。

简单：基于HTTP+JSON的API让你用curl就可以轻松使用。

安全：可选SSL客户认证机制。

快速：每个实例每秒支持一千次写操作。

可信：使用Raft算法充分实现了分布式。

**2 分布式系统中的数据分为控制数据和应用数据**。etcd的使用场景默认处理的数据都是控制数据，对于应用数据，只推荐数据量很小，但是更新访问频繁的情况。

**3 应用场景有如下几类：**

场景一：服务发现（Service Discovery）

场景二：消息发布与订阅

场景三：负载均衡

场景四：分布式通知与协调

场景五：分布式锁、分布式队列

场景六：集群监控与Leader竞选

举个最简单的例子，如果你需要一个分布式存储仓库来存储配置信息，并且希望这个仓库读写速度快、支持高可用、部署简单、支持http接口，那么就可以使用etcd。目前，cloudfoundry使用etcd作为hm9000的应用状态信息存储，kubernetes用etcd来存储docker集群的配置信息等。

**4 ETCD是什么**

ETCD是用于共享配置和服务发现的分布式，一致性的KV存储系统。该项目目前最新稳定版本为2.3.0. 具体信息请参考[项目首页]和[Github]。ETCD是CoreOS公司发起的一个开源项目，授权协议为Apache。提供配置共享和服务发现的系统比较多，其中最为大家熟知的是[Zookeeper]（后文简称ZK），而ETCD可以算得上是后起之秀了。在项目实现，一致性协议易理解性，运维，安全等多个维度上，ETCD相比Zookeeper都占据优势。

**5 ETCD vs ZK**

本文选取ZK作为典型代表与ETCD进行比较，而不考虑[Consul]项目作为比较对象，原因为Consul的可靠性和稳定性还需要时间来验证（项目发起方自身服务并未使用Consul, 自己都不用)。

一致性协议： ETCD使用[Raft]协议， ZK使用ZAB（类PAXOS协议），前者容易理解，方便工程实现；

运维方面：ETCD方便运维，ZK难以运维；

项目活跃度：ETCD社区与开发活跃，ZK已经快死了；

API：ETCD提供HTTP+JSON, gRPC接口，跨平台跨语言，ZK需要使用其客户端；

访问安全方面：ETCD支持HTTPS访问，ZK在这方面缺失；

**6 ETCD的使用场景**

和ZK类似，ETCD有很多使用场景，包括：

配置管理

服务注册于发现

选主

应用调度

分布式队列

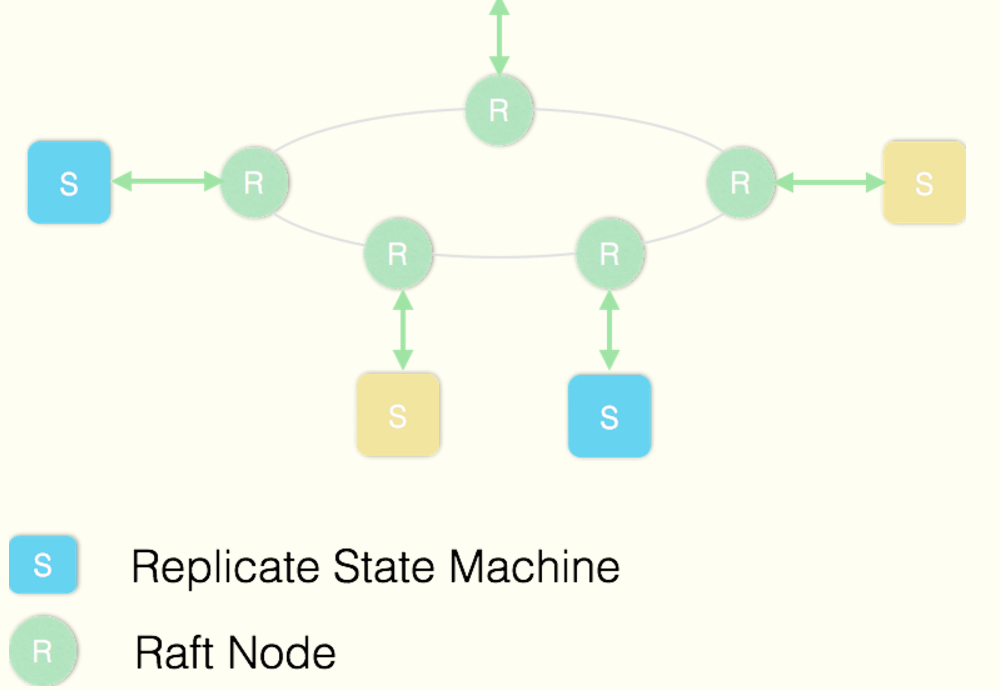
分布式锁

**7 ETCD读写性能**

按照官网给出的[Benchmark], 在2CPU，1.8G内存，SSD磁盘这样的配置下，单节点的写性能可以达到16K QPS, 而先写后读也能达到12K QPS。这个性能还是相当可观的。

**8 ETCD工作原理**

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，具体细节请参考其[论文]。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。

**9 选主**

Raft协议是用于维护一组服务节点数据一致性的协议。这一组服务节点构成一个集群，并且有一个主节点来对外提供服务。当集群初始化，或者主节点挂掉后，面临一个选主问题。集群中每个节点，任意时刻处于Leader, Follower, Candidate这三个角色之一。选举特点如下：

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；

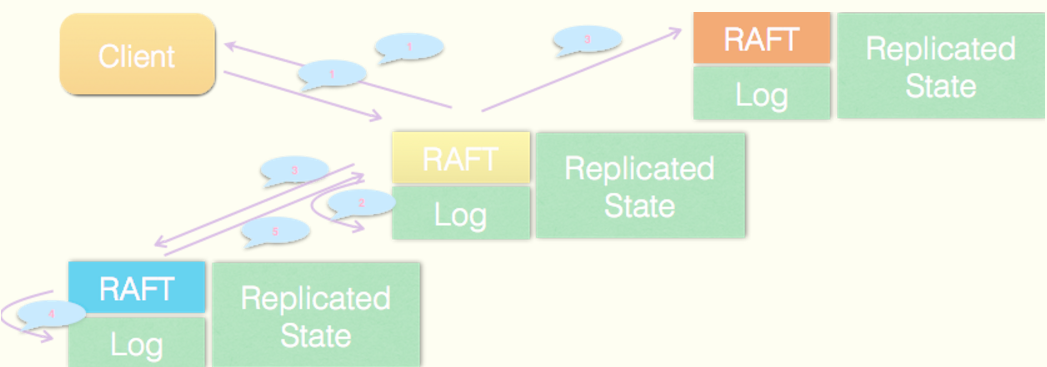
当Follower在一定时间内没有收到来自主节点的心跳，会将自己角色改变为Candidate，并发起一次选主投票；当收到包括自己在内超过半数节点赞成后，选举成功；当收到票数不足半数选举失败，或者选举超时。若本轮未选出主节点，将进行下一轮选举（出现这种情况，是由于多个节点同时选举，所有节点均为获得过半选票）。

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。

**10 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。



**11 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。

**12 ETCD使用案例**

据公开资料显示，至少有CoreOS, Google Kubernetes, Cloud Foundry, 以及在Github上超过500个项目在使用ETCD。

**13 ETCD接口**

ETCD提供HTTP协议，在最新版本中支持Google gRPC方式访问。具体支持接口情况如下：

ETCD是一个高可靠的KV存储系统，支持PUT/GET/DELETE接口；

为了支持服务注册与发现，支持WATCH接口（通过http long poll实现）；

支持KEY持有TTL属性；

CAS（compare and swap)操作;

支持多key的事务操作；

支持目录操作

## 二、部署集群

**1. 概述**

想必很多人都知道ZooKeeper，通常用作配置共享和服务发现。和它类似，ETCD算是一个非常优秀的后起之秀了。本文重点不在描述他们之间的不同点。首先，看看其官网关于ETCD的描述1:

A distributed, reliable key-value store for the most critical data of a distributed system.

在云计算大行其道的今天，ETCD有很多典型的使用场景。常言道，熟悉一个系统先从部署开始。本文接下来将描述，如何部署ETCD集群。

安装官网说明文档，提供了3种集群启动方式，实际上按照其实现原理分为2类：

通过静态配置方式启动

通过服务发现方式启动

在部署集群之前，我们需要考虑集群需要配置多少个节点。这是一个重要的考量，不得忽略。

**2. 集群节点数量与网络分割**

ETCD使用RAFT协议保证各个节点之间的状态一致。根据RAFT算法原理，节点数目越多，会降低集群的写性能。这是因为每一次写操作，需要集群中大多数节点将日志落盘成功后，Leader节点才能将修改内部状态机，并返回将结果返回给客户端。

也就是说在等同配置下，节点数越少，集群性能越好。显然，只部署1个节点是没什么意义的。通常，按照需求将集群节点部署为3，5，7，9个节点。

这里能选择偶数个节点吗？ 最好不要这样。原因有二：

偶数个节点集群不可用风险更高，表现在选主过程中，有较大概率或等额选票，从而触发下一轮选举。

偶数个节点集群在某些网络分割的场景下无法正常工作。试想，当网络分割发生后，将集群节点对半分割开。此时集群将无法工作。按照RAFT协议，此时集群写操作无法使得大多数节点同意，从而导致写失败，集群无法正常工作。

当网络分割后，ETCD集群如何处理的呢?

当集群的Leader在多数节点这一侧时，集群仍可以正常工作。少数节点那一侧无法收到Leader心跳，也无法完成选举。

当集群的Leader在少数节点这一侧时，集群仍可以正常工作，多数派的节点能够选出新的Leader, 集群服务正常进行。

当网络分割恢复后，少数派的节点会接受集群Leader的日志，直到和其他节点状态一致。

**3. ETCD参数说明**

这里只列举一些重要的参数，以及其用途。

—data-dir 指定节点的数据存储目录，这些数据包括节点ID，集群ID，集群初始化配置，Snapshot文件，若未指定—wal-dir，还会存储WAL文件；

—wal-dir 指定节点的was文件的存储目录，若指定了该参数，wal文件会和其他数据文件分开存储。

—name 节点名称

—initial-advertise-peer-urls 告知集群其他节点url.

— listen-peer-urls 监听URL，用于与其他节点通讯

— advertise-client-urls 告知客户端url, 也就是服务的url

— initial-cluster-token 集群的ID

— initial-cluster 集群中所有节点

**4. 通过静态配置方式启动ETCD集群**

按照官网中的文档，即可完成集群启动。这里略。

**5. 通过服务发现方式启动ETCD集群**

ETCD还提供了另外一种启动方式，即通过服务发现的方式启动。这种启动方式，依赖另外一个ETCD集群，在该集群中创建一个目录，并在该目录中创建一个\_config的子目录，并且在该子目录中增加一个size节点，指定集群的节点数目。

在这种情况下，将该目录在ETCD中的URL作为节点的启动参数，即可完成集群启动。使用

--discovery https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83 配置项取代静态配置方式中的--initial-cluster 和inital-cluster-state参数。其中https://myetcd.local/v2/keys/discovery/6c007a14875d53d9bf0ef5a6fc0257c817f0fb83是在依赖etcd中创建好的目录url。

**6. 节点迁移**

在生产环境中，不可避免遇到机器硬件故障。当遇到硬件故障发生的时候，我们需要快速恢复节点。ETCD集群可以做到在不丢失数据的，并且不改变节点ID的情况下，迁移节点。

具体办法是：

1）停止待迁移节点上的etc进程；

2）将数据目录打包复制到新的节点；

3）更新该节点对应集群中peer url，让其指向新的节点；

4）使用相同的配置，在新的节点上启动etcd进程；