零声教育 Mark 老师 QQ: 2548898954

etcd 简介

> *etcd* 诞生于 *CoreOS* 公司，最初用于解决集群管理系统中 *os* 升级时的分布式并发控制、配置文件的存储与分发等问题。基于此，*etcd* 设计为提供高可用、强一致性的小型 *kv* 数据存储服务。项目当前隶属于 *CNCF* 基金会，被包括 *AWS*、*Google*、*Microsoft*、*Alibaba* 等大型互联网公司广泛使用；
>
> *etcd* 基于 *Go* 语言实现，主要用于共享配置和服务发现；
>
> *etc* 在 *Linux* 系统中是配置文件目录名；*etcd* 就是配置服务；

etcd 安装

> 1. 在 gitHub releases page 下载相应平台的软件压缩包；
> 2. 解压缩
> 3. 如果是 *linux* 系统，进入解压目录，将 *etcd* 和 *etcdctl* 可执行文件移动到 `/usr/local/bin` 目录下
> 4. 执行 `etcd --version` 如果能看到版本信息，说明安装成功；

etcd v2 和 v3 比较

- 使用 *gRPC* + *protobuf* 取代 *http* + *json* 通信，提高通信效率；*gRPC* 只需要一条连接；*http* 是每个请求建立一条连接；*protobuf* 加解密比 *json* 加解密速度得到数量级的提升；包体也更小；

  > v3 也可使用 *http* + *json* 访问，etcd 使用 *gRPC-Gateway* 对外提供 *HTTP API* 接口（有的语言客户端不支持 *gRPC* 通信协议）；
  >
  > http://www.etcd.cn/docs/current/learning/api/

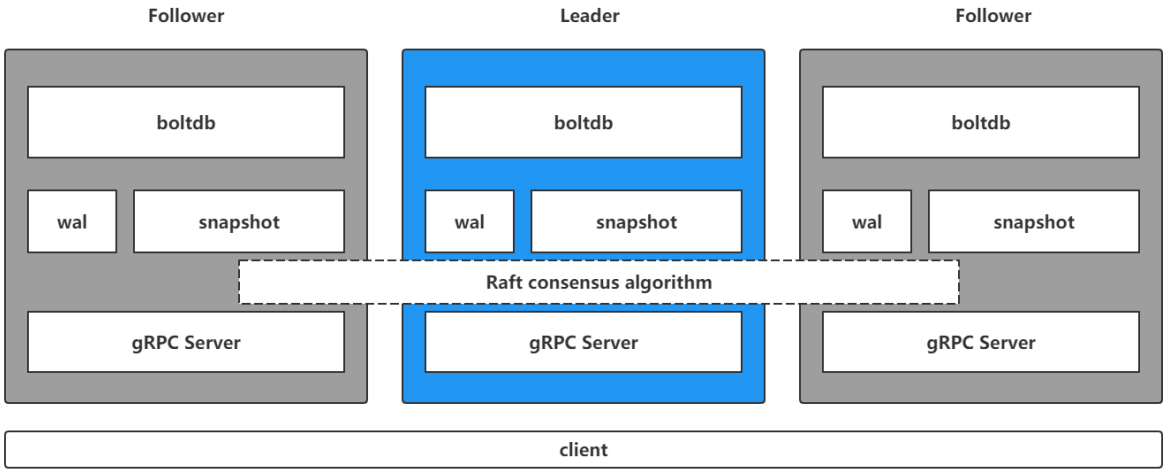
```
url -L http://localhost:2379/v3/kv/put \
 -X POST -d '{"key": "Zm9v", "value": "YmFy"}'
# {"header":
{"cluster_id":"12585971608760269493","member_id":"13847567121247652
255","revision":"2","raft_term":"3"}}

curl -L http://localhost:2379/v3/kv/range \
 -X POST -d '{"key": "Zm9v"}'
# {"header":
{"cluster_id":"12585971608760269493","member_id":"13847567121247652
255","revision":"2","raft_term":"3"},"kvs":
[{"key":"Zm9v","create_revision":"2","mod_revision":"2","version":"
1","value":"YmFy"}],"count":"1"}

# get all keys prefixed with "foo"
curl -L http://localhost:2379/v3/kv/range \
 -X POST -d '{"key": "Zm9v", "range_end": "Zm9w"}'
# {"header":
{"cluster_id":"12585971608760269493","member_id":"13847567121247652
255","revision":"2","raft_term":"3"},"kvs":
[{"key":"Zm9v","create_revision":"2","mod_revision":"2","version":"
1","value":"YmFy"}],"count":"1"}
```

- v3 使用 *lease* （租约）替换 *key ttl* 自动过期机制；
- v3 支持事务和多版本并发控制（一致性非锁定读）的磁盘数据库；而 v2 是简单的 *kv* 内存数据库；
- v3 是扁平的kv结构；v2 是类型文件系统的存储结构；

etcd 架构（体系结构）



> *boltdb* 是一个单机的支持事务的 *kv* 存储，*etcd* 的事务是基于 *boltdb* 的事务实现的；*boltdb* 为每一个 *key* 都创建一个索引（*B+*树）；该 *B+* 树存储了 *key* 所对应的版本数据；
>
> *wal*（write ahead log）预写式日志实现事务日志的标准方法；执行写操作前先写日志，跟 *mysql* 中 *redo* 类似，*wal* 实现的是顺序写，而若按照 *B+* 树写，则涉及到多次 *io* 以及 随机写；
>
> *snapshot* 快照数据，用于其他节点同步主节点数据从而达到一致性地状态；类似 *redis* 中主从复制中 *rdb* 数据恢复；流程：1. *leader* 生成 *snapshot*；2. *leader* 向 *follower* 发送 *snapshot*；3. *follower* 接收并应用*snapshot*；
>
> *gRPC server*: *ectd* 集群间以及 *client* 与 *etcd* 节点间都是通过 *gRPC* 进行通讯；

etcd APIs

设置

```
# Puts the given key into the store
PUT key val
--ignore-lease[=false]    updates the key using its current lease
--ignore-value[=false]    updates the key using its current value
--lease="0"               lease ID (in hexadecimal) to attach to the key
--prev-kv[=false]         return the previous key-value pair before
modification
```

删除

```
# Removes the specified key or range of keys [key, range_end)
DEL key

DEL keyfrom keyend

--from-key[=false]        #delete keys that are greater than or equal to the given key using byte compare
--prefix[=false]          #delete keys with matching prefix
--prev-kv[=false]         #return deleted key-value pairs
```

获取

> 问题：从前缀相同的一系列 *key* 中，获取第一个创建的 *kv*?
>
> `.\etcdctl.exe get key --prefix --sort-by="create" --limit=1`
>
> 问题：从前缀相同的一系列 *key* 中，获取恰好小于某个 *kv* 的版本且存在的某个 *kv*（注意前一个可能已经删除了）；

```
.\etcdctl.exe get key:3 -w json
{"header":
{"cluster_id":14841639068965178418,"member_id":10276657743932975437,"revision":152,"raft_term":18},"kvs":
[{"key":"a2V5OjM=","create_revision":152,"mod_revision":152,"version":1,"value":"dmFsOjM="}],"count":1}
.\etcdctl.exe get key --prefix --sort-by="create" --order="descend" --limit=1 --rev=151
```

```
# Gets the key or a range of keys
GET key

GET keyfrom keyend

--consistency="l"                 #Linearizable(l) or Serializable(s)
--count-only[=false]              #Get only the count
--from-key[=false]                #Get keys that are greater than or equal to the given key using byte compare
--keys-only[=false]               #Get only the keys
--limit=0                         #Maximum number of results
--order=""                        #Order of results; ASCEND or DESCEND (ASCEND by default)
--prefix[=false]                  #Get keys with matching prefix
--print-value-only[=false]        #Only write values when using the "simple" output format
--rev=0                           #Specify the kv revision
--sort-by=""                      #Sort target; CREATE, KEY, MODIFY, VALUE, or VERSION
```

监听

> 用来实现监听和推送服务；

```
# Watches events stream on keys or prefixes
WATCH key

-i, --interactive[=false]       #Interactive mode
--prefix[=false]                #Watch on a prefix if prefix is set
--prev-kv[=false]               #get the previous key-value pair before the event happens
--progress-notify[=false]       #get periodic watch progress notification from server
--rev=0                         #Revision to start watching
```

事务

> 用于分布式锁以及 *leader* 选举；保证多个操作的原子性；确保多个节点数据读写的一致性；
>
> 问题：通过事务实现 *redis* 中的 `setnx`；
>
> 注意：
>
> 1. 比较
>   1. 比较运算符 `> = < !=` ；比较运算符两侧需要**空格**；
>   2. `create` 获取 *key* 的 *create_revision*
>   3. `mod` 获取 *key* 的 *mod_revision*
>   4. `value` 获取 *key* 的 *value*
>   5. `version` 获取 *key* 的修改次数
> 2. 比较成功
>   1. 成功后可以操作多个 `del put get`
>   2. 这些操作保证原子性
> 3. 比较失败
>   1. 失败后可以操作多个 `del put get`
>   2. 这些操作保证原子性

```
# Txn processes all the requests in one transaction
TXN if/ then/ else ops
-i, --interactive[=false]     #Input transaction in interactive mode
```

租约

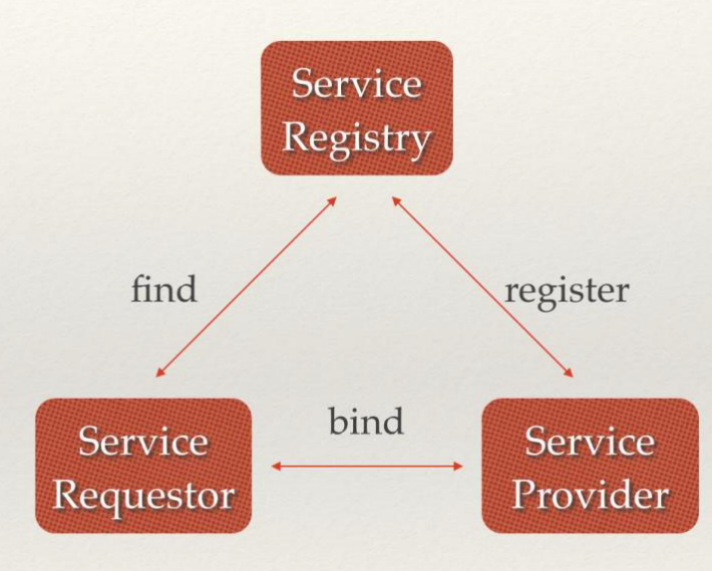
> 用于集群监控以及服务注册发现

```
lease grant             # 创建一个租约
lease keep-alive        # 续约
lease list              # 枚举所有的租约
lease revoke            # 销毁租约
lease timetolive        # 获取租约信息
```

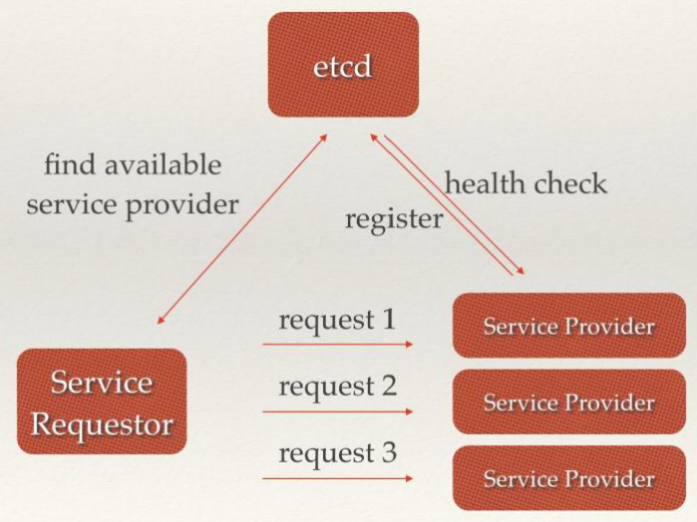
应用场景

> **图片来源于网上**

服务发现



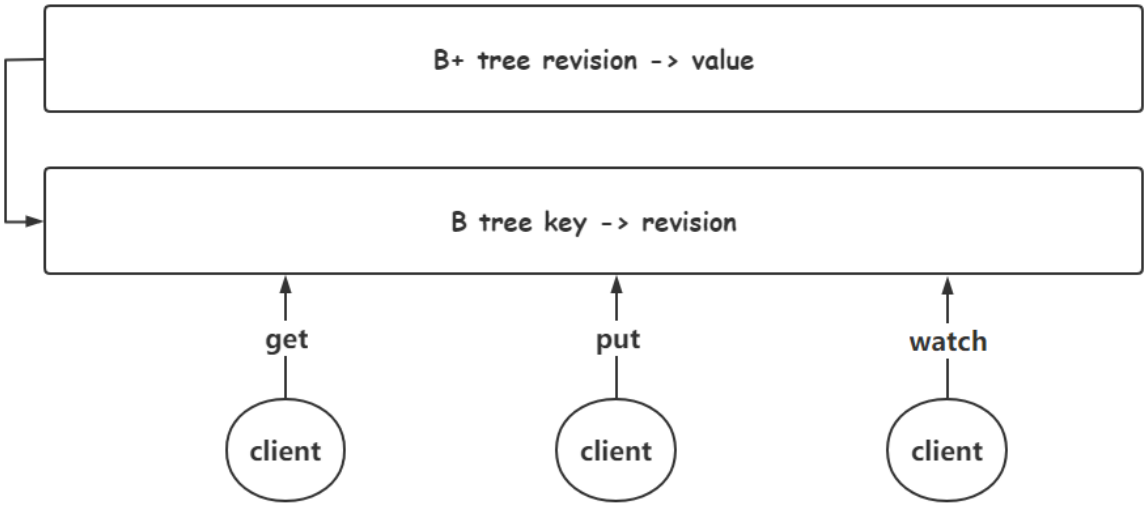
负载均衡



数据版本号机制

> - *term*：
>
>   *leader* 任期，*leader* 切换时 *term* 加一；全局单调递增，64 bits；
> - *revision*：
>
>   *etcd* 键空间版本号，*key* 发生变更，则 *revision* 加一；全局单调递增，64 bits；用来支持 *MVCC*；
> - *kv*：
>   - *create_revision*
>
>   创建数据时，对应的版本号；
>   - *mod_revision*
>
>   数据修改时，对应的版本号；
>   - *version*
>
>   当前的版本号；标识该 *val* 被修改了多少次；

etcd 存储原理

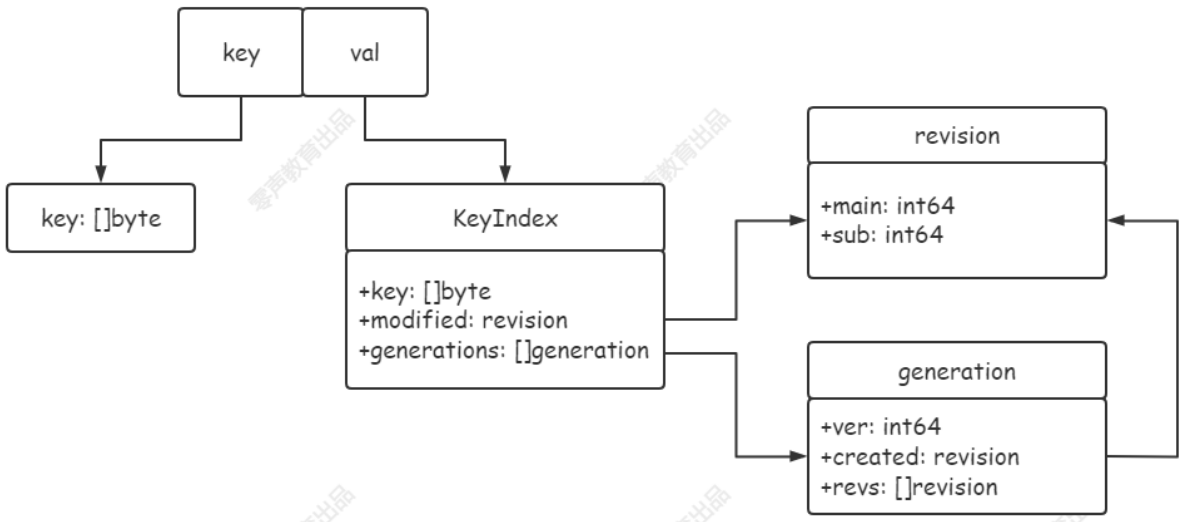


> *etcd* 为每个 *key* 创建一个索引；一个索引对应着一个 *B+* 树；*B+* 树 *key* 为 *revision*，*B+* 节点存储的值为*value*；*B+* 树存储着 *key* 的版本信息从而实现了 *etcd* 的 *mvcc*；*etcd* 不会任由版本信息膨胀，通过定期的 *compaction* 来清理历史数据；
>
> 思考：*mysql* 的 *mvcc* 是通过什么实现的(*undolog*)？ *mysql* B+ 树存储什么内容?
>
> 为了加速索引数据，在内存中维持着一个 B 树；B 树 *key* 为 *key*，*value* 为该 *key* 的 *revision*；
>
> 思考：*mysql* 为了加快索引数据，采用什么数据结构?
>
> *MySQL* 采用自适应 *hash* 来加速索引；

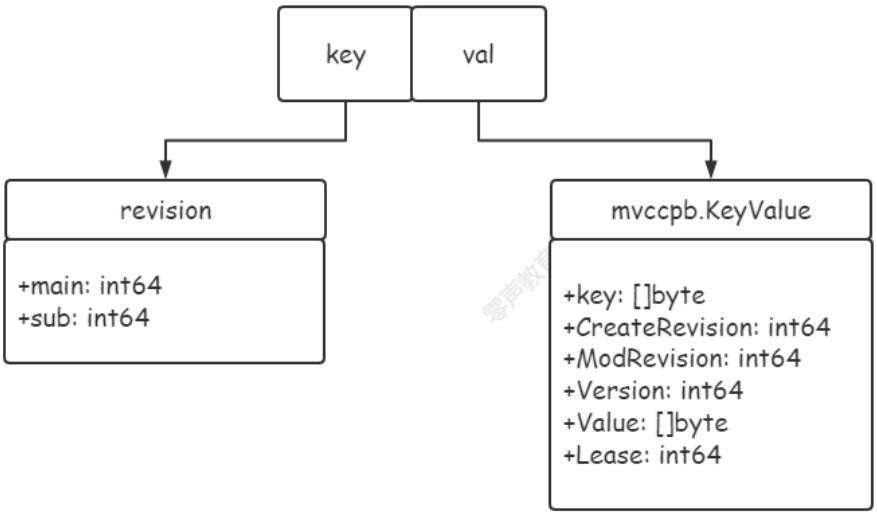
读写机制

> *etcd* 是串行写，并发读；
>
> 并发读写时（读写同时进行），读操作是通过 *B+* 树 *mmap* 访问磁盘数据；写操作走日志复制流程；可以得知如果此时读操作走 *B* 树出现脏读幻读问题；通过 *B+* 树访问磁盘数据其实访问的事务开始前的数据，由 *mysql* 可重复读隔离级别下 *MVCC* 读取规则可知能避免脏读和幻读问题；
>
> 并发读时，可走内存 B 树；

节点在内存



节点在磁盘



分布式锁

技术重点

- 锁是一种资源
- 互斥语义
- 获取锁和释放锁必须为同一对象
- 锁超时（获取锁和释放锁是通过网路通讯实现的）
- 锁释放通知问题
- 是否允许同一对象多次获取锁（可选）

数据库实现

> 互斥语义由数据库唯一约束来实现；

表结构

```
DROP TABLE IF EXISTS `dislock`;
CREATE TABLE `dislock` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `lock_type` varchar(64) NOT NULL COMMENT '锁类型',
  `owner_id` varchar(255) NOT NULL COMMENT '持锁对象',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `idx_lock_type` (`lock_type`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='分布式锁表';
```

获取锁

```
INSERT INTO dislock (`lock_type`, `owner_id`) VALUES ('mark_lock',
'ad2daf3');
```

释放锁

```
DELETE FROM dislock WHERE `lock_type` = 'mark_lock' AND `owner_id` =
'ad2daf3';
```

Redis 实现

获取锁

```
set lock_type uuid ex 30 nx
```

释放锁

```
local lock_type = KEYS[1]
local uuid = KEYS[2]
local real_uuid = redis.call("get", lock_type)
if uuid == real_uuid then
    redis.call("del", lock_type)
end
```

Etcd 实现

获取锁

> 获取锁时，需要在 *etcd* 做一个标记；根据这些标记进行排序（根据标记的创建版本号排序）；
>
> 做标记的同时，需要获取当前持有锁的对象；
>
> 如果持有锁的对象不是自己，监听版本号刚好小于自己的对象的删除信息；
>
> 问题：如果当前 *watch* 的对象已经退出但自己仍没有获取锁的权限，怎么修改监听对象？
>
> 继续监听版本号刚好小于自己的对象的删除信息；
>
> 问题：监听动作什么情况下结束？
>
> 没有比自己更小的版本号对象，那么自己就获取了持锁权限；

```
func (m *Mutex) Lock(ctx context.Context) error {
    s := m.s
    client := m.s.Client()

    m.myKey = fmt.Sprintf("%s%x", m.pfx, s.Lease())
    cmp := v3.Compare(v3.CreateRevision(m.myKey), "=", 0)
    // put self in lock waiters via myKey; oldest waiter holds lock
    put := v3.OpPut(m.myKey, "", v3.WithLease(s.Lease()))
    // reuse key in case this session already holds the lock
    get := v3.OpGet(m.myKey)
    // fetch current holder to complete uncontended path with only one RPC
    getOwner := v3.OpGet(m.pfx, v3.WithFirstCreate()...)
    resp, err := client.Txn(ctx).If(cmp).Then(put, getOwner).Else(get,
getOwner).Commit()
    if err != nil {
        return err
    }
    m.myRev = resp.Header.Revision
```

```
    if !resp.Succeeded {
        m.myRev = resp.Responses[0].GetResponseRange().Kvs[0].CreateRevision
    }
    // if no key on prefix / the minimum rev is key, already hold the lock
    ownerKey := resp.Responses[1].GetResponseRange().Kvs
    if len(ownerKey) == 0 || ownerKey[0].CreateRevision == m.myRev {
        m.hdr = resp.Header
        return nil
    }

    // wait for deletion revisions prior to myKey
    hdr, werr := waitDeletes(ctx, client, m.pfx, m.myRev-1)
    // release lock key if wait failed
    if werr != nil {
        m.Unlock(client.Ctx())
    } else {
        m.hdr = hdr
    }
    return werr
}

func waitDeletes(ctx context.Context, client *v3.Client, pfx string,
maxCreateRev int64) (*pb.ResponseHeader, error) {
    getOpts := append(v3.WithLastCreate(),
v3.WithMaxCreateRev(maxCreateRev))
    for {
        resp, err := client.Get(ctx, pfx, getOpts...)
        if err != nil {
            return nil, err
        }
        if len(resp.Kvs) == 0 {
            return resp.Header, nil
        }
        lastKey := string(resp.Kvs[0].Key)
        if err = waitDelete(ctx, client, lastKey, resp.Header.Revision); err
!= nil {
            return nil, err
        }
    }
}
```

释放锁

```
func (m *Mutex) Unlock(ctx context.Context) error {
    client := m.s.Client()
    if _, err := client.Delete(ctx, m.myKey); err != nil {
        return err
    }
    m.myKey = "\x00"
    m.myRev = -1
    return nil
}
```