Kevin 4.5k

golang etcd简明教程

golang etcd 更新于 2019-11-17 • 约 26 分钟

etcd 是一个高可用强一致性的键值仓库在很多分布式系统架构中得到了广泛的应用，本教程结合一些简单的例子介绍golang版本的`etcd/clientv3`中提供的主要功能及其使用方法。

如果还不熟悉etcd推荐先阅读：

看图轻松了解etcd

etcd常用操作介绍

Let's get started now!

安装package

我们使用v3版本的etcd client， 首先通过`go get`下载并编译安装`etcd clinet v3`。

```
go get github.com/coreos/etcd/clientv3
```

该命令会将包下载到`$GOPATH/src/github.com/coreos/etcd/clientv3`中，所有相关依赖包会自动下载编译，包括`protobuf`、`grpc`等。

官方文档地址：https://godoc.org/github.com/...

文档中列出了Go官方实现的etcd client中支持的所有方法，方法还是很多的，我们主要梳理一下使用etcd时经常用到的主要API并进行演示。

连接客户端

用程序访问etcd首先要创建client，它需要传入一个Config配置，这里传了2个选项：

- Endpoints：etcd的多个节点服务地址。
- DialTimeout：创建client的首次连接超时时间，这里传了5秒，如果5秒都没有连接成功就会返回err；一旦client创建成功，我们就不用再关心后续底层连接的状态了，client内部会重连。

```
cli, err := clientv3.New(clientv3.Config{
   Endpoints:   []string{"localhost:2379"},
   // Endpoints: []string{"localhost:2379", "localhost:22379", "localhost:32379"}
   DialTimeout: 5 * time.Second,
})
```

返回的`client`，它的类型具体如下：

```
type Client struct {
    Cluster
    KV
    Lease
    Watcher
    Auth
    Maintenance
    // Username is a user name for authentication.
    Username string
    // Password is a password for authentication.
    Password string
    // contains filtered or unexported fields
}
```

类型中的成员是etcd客户端几何核心功能模块的具体实现，它们分别用于：

- Cluster：向集群里增加etcd服务端节点之类，属于管理员操作。
- KV：我们主要使用的功能，即K-V键值库的操作。
- Lease：租约相关操作，比如申请一个TTL=10秒的租约（应用给key可以实现键值的自动过期）。
- Watcher：观察订阅，从而监听最新的数据变化。
- Auth：管理etcd的用户和权限，属于管理员操作。
- Maintenance：维护etcd，比如主动迁移etcd的leader节点，属于管理员操作。

我们需要使用什么功能，就去client里获取对应的成员即可。

Client.KV是一个interface`，提供了关于K-V操作的所有方法：

```
type KV interface {

    Put(ctx context.Context, key, val string, opts ...OpOption) (*PutResponse, error)

    Get(ctx context.Context, key string, opts ...OpOption) (*GetResponse, error)

    // Delete deletes a key, or optionally using WithRange(end), [key, end).
    Delete(ctx context.Context, key string, opts ...OpOption) (*DeleteResponse, error)

    // Compact compacts etcd KV history before the given rev.
    Compact(ctx context.Context, rev int64, opts ...CompactOption) (*CompactResponse, error)

    Do(ctx context.Context, op Op) (OpResponse, error)

    // Txn creates a transaction.
    Txn(ctx context.Context) Txn
}
```

我们通过方法clientv3.NewKV()来获得KV接口的实现（实现中内置了错误重试机制）：

```
kv := clientv3.NewKV(cli)
```

接下来，我们将通过kv操作etcd中的数据。

Put

```
putResp, err := kv.Put(context.TODO(),"/test/key1", "Hello etcd!")
```

第一个参数是goroutine的上下文Context。后面两个参数分别是key和value，对于etcd来说，key=/test/key1只是一个字符串而已，但是对我们而言却可以模拟出目录层级关系。

Put函数的声明如下：

```
// Put puts a key-value pair into etcd.
// Note that key,value can be plain bytes array and string is
// an immutable representation of that bytes array.
// To get a string of bytes, do string([]byte{0x10, 0x20}).
Put(ctx context.Context, key, val string, opts ...OpOption) (*PutResponse, error)
```

除了上面例子中的三个的参数，还支持一个变长参数，可以传递一些控制项来影响Put的行为，例如可以携带一个lease ID来支持key过期。

Put操作返回的是PutResponse，不同的KV操作对应不同的response结构，所有KV操作返回的response结构如下：

```
type (
   CompactResponse pb.CompactionResponse
   PutResponse     pb.PutResponse
   GetResponse     pb.RangeResponse
   DeleteResponse  pb.DeleteRangeResponse
   TxnResponse     pb.TxnResponse
)
```

程序代码里导入clientv3后在GoLand中可以很快定位到PutResponse的定义文件中，PutResponse只是pb.PutResponse的类型别名，通过Goland跳转过去后可以看到PutResponse的详细定义。

```
type PutResponse struct {
   Header *ResponseHeader `protobuf:"bytes,1,opt,name=header" json:"header,omitempty"`
   // if prev_kv is set in the request, the previous key-value pair will be returned.
   PrevKv *mvccpb.KeyValue `protobuf:"bytes,2,opt,name=prev_kv,json=prevKv" json:"prev_kv,omitempty"`
}
```

Header里保存的主要是本次更新的revision信息，而PrevKv可以返回Put覆盖之前的value是什么（目前是nil，后面会说原因），把返回的PutResponse打印出来看一下：

```
fmt.Printf("PutResponse: %v, err: %v", putResp, err)

// output
// PutResponse: &{cluster_id:14841639068965178418 member_id:10276657743932975437 revision:3 raft_term:7  <nil>},
err: <nil>%
```

我们需要判断err来确定操作是否成功。

我们再Put其他2个key，用于后续演示：

```
kv.Put(context.TODO(),"/test/key2", "Hello World!")
// 再写一个同前缀的干扰项
kv.Put(context.TODO(), "/testspam", "spam")
```

现在/test目录下有两个键: key1和key2， 而/testspam并不归属于/test目录

Get

使用KV的Get方法来读取给定键的值：

```
getResp, err := kv.Get(context.TODO(), "/test/key1")
```

其函数声明如下：

```
// Get retrieves keys.
// By default, Get will return the value for "key", if any.
// When passed WithRange(end), Get will return the keys in the range [key, end).
// When passed WithFromKey(), Get returns keys greater than or equal to key.
// When passed WithRev(rev) with rev > 0, Get retrieves keys at the given revision;
// if the required revision is compacted, the request will fail with ErrCompacted .
// When passed WithLimit(limit), the number of returned keys is bounded by limit.
// When passed WithSort(), the keys will be sorted.
Get(ctx context.Context, key string, opts ...OpOption) (*GetResponse, error)
```

和Put类似，函数注释里提示我们可以传递一些控制参数来影响Get的行为，比如：WithFromKey表示读取从参数key开始递增的所有key，而不是读取单个key。

在上面的例子中，我没有传递opOption，所以就是获取key=/test/key1的最新版本数据。

这里err并不能反馈出key是否存在（只能反馈出本次操作因为各种原因异常了），我们需要通过GetResponse（实际上是pb.RangeResponse）判断key是否存在：

```
type RangeResponse struct {
    Header *ResponseHeader `protobuf:"bytes,1,opt,name=header" json:"header,omitempty"`
    // kvs is the list of key-value pairs matched by the range request.
    // kvs is empty when count is requested.
    Kvs []*mvccpb.KeyValue `protobuf:"bytes,2,rep,name=kvs" json:"kvs,omitempty"`
    // more indicates if there are more keys to return in the requested range.
    More bool `protobuf:"varint,3,opt,name=more,proto3" json:"more,omitempty"`
    // count is set to the number of keys within the range when requested.
    Count int64 `protobuf:"varint,4,opt,name=count,proto3" json:"count,omitempty"`
}
```

Kvs字段，保存了本次Get查询到的所有k-v对，因为上述例子只Get了一个单key，所以只需要判断一下len(Kvs)是否等于1即可知道key是否存在。

`RangeResponse.More`和`Count`，当我们使用`withLimit()`等选项进行`Get`时会发挥作用，相当于翻页查询。

接下来，我们通过给Get查询增加WithPrefix选项，获取/test目录下的所有子元素：

```
rangeResp, err := kv.Get(context.TODO(), "/test/", clientv3.WithPrefix())
```

`WithPrefix()`是指查找以`/test/`为前缀的所有key，因此可以模拟出查找子目录的效果。

`etcd`是一个有序的k-v存储，因此/test/为前缀的key总是顺序排列在一起。

`withPrefix()`实际上会转化为范围查询，它根据前缀`/test/`生成了一个前闭后开的key range：`["/test/", "/test0")`，为什么呢？因为比`/`大的字符是`0`，所以以`/test0`作为范围的末尾，就可以扫描到所有以`/test/`为前缀的key了。

在之前，我们Put了一个`/testspam`键值，因为不符合`/test/`前缀（注意末尾的/），所以就不会被这次`Get`获取到。但是，如果查询的前缀是`/test`，那么`/testspam`就会被返回，使用时一定要特别注意。

打印rangeResp.Kvs可以看到获得了两个键值：

```
[key:"/test/key1" create_revision:2 mod_revision:13 version:6 value:"Hello etcd!" key:"/test/key2" create_revision:5
mod_revision:14 version:4 value:"Hello World!" ]
```

Lease

etcd客户端的Lease对象可以通过以下的代码获取到

```
lease := clientv3.NewLease(cli)
```

lease对象是Lease接口的实现，Lease接口的声明如下：

```
type Lease interface {
    // Grant 创建一个新租约
    Grant(ctx context.Context, ttl int64) (*LeaseGrantResponse, error)

    // Revoke 销毁给定租约ID的租约
    Revoke(ctx context.Context, id LeaseID) (*LeaseRevokeResponse, error)

    // TimeToLive retrieves the lease information of the given lease ID.
    TimeToLive(ctx context.Context, id LeaseID, opts ...LeaseOption) (*LeaseTimeToLiveResponse, error)

    // Leases retrieves all leases.
    Leases(ctx context.Context) (*LeaseLeasesResponse, error)

    // KeepAlive keeps the given lease alive forever.
    KeepAlive(ctx context.Context, id LeaseID) (<-chan *LeaseKeepAliveResponse, error)

    // KeepAliveOnce renews the lease once. In most of the cases, KeepAlive
    // should be used instead of KeepAliveOnce.
    KeepAliveOnce(ctx context.Context, id LeaseID) (*LeaseKeepAliveResponse, error)

    // Close releases all resources Lease keeps for efficient communication
    // with the etcd server.
    Close() error
}
```

Lease提供了以下功能：

- Grant：分配一个租约。
- Revoke：释放一个租约。
- TimeToLive：获取剩余TTL时间。
- Leases：列举所有etcd中的租约。
- KeepAlive：自动定时的续约某个租约。
- KeepAliveOnce：为某个租约续约一次。
- Close：释放当前客户端建立的所有租约。

要想实现key自动过期，首先得创建一个租约，下面的代码创建一个TTL为10秒的租约：

```
grantResp, err := lease.Grant(context.TODO(), 10)
```

返回的grantResponse的结构体声明如下：

```
// LeaseGrantResponse wraps the protobuf message LeaseGrantResponse.
type LeaseGrantResponse struct {
    *pb.ResponseHeader
    ID    LeaseID
    TTL   int64
    Error string
}
```

在应用程序代码中主要使用到的是租约ID。

接下来我们用这个Lease往etcd中存储一个10秒过期的key：

```
kv.Put(context.TODO(), "/test/vanish", "vanish in 10s", clientv3.WithLease(grantResp.ID))
```

这里特别需要注意，有一种情况是在Put之前Lease已经过期了，那么这个Put操作会返回error，此时你需要重新分配Lease。

当我们实现服务注册时，需要主动给Lease进行续约，通常是以小于TTL的间隔循环调用Lease的KeepAliveOnce()方法对租约进行续期，一旦某个服务节点出错无法完成租约的续期，等key过期后客户端即无法在查询服务时获得对应节点的服务，这样就通过租约到期实现了服务的错误隔离。

```
keepResp, err := lease.KeepAliveOnce(context.TODO(), grantResp.ID)
```

或者使用`KeepAlive()`方法，其会返回`<-chan *LeaseKeepAliveResponse`只读通道，每次自动续租成功后会向通道中发送信号。一般都用`KeepAlive()`方法

KeepAlive和Put一样，如果在执行之前Lease就已经过期了，那么需要重新分配Lease。etcd并没有提供API来实现原子的Put with Lease，需要我们自己判断err重新分配Lease。

Op

Op字面意思就是"操作"，Get和Put都属于Op，只是为了简化用户开发而开放的特殊API。

KV对象有一个Do方法接受一个Op：

```
// Do applies a single Op on KV without a transaction.
// Do is useful when creating arbitrary operations to be issued at a
// later time; the user can range over the operations, calling Do to
// execute them. Get/Put/Delete, on the other hand, are best suited
// for when the operation should be issued at the time of declaration.
Do(ctx context.Context, op Op) (OpResponse, error)
```

其参数Op是一个抽象的操作，可以是Put/Get/Delete...；而OpResponse是一个抽象的结果，可以是PutResponse/GetResponse...

可以通过Client中定义的一些方法来创建Op：

- func OpDelete(key string, opts ...OpOption) Op
- func OpGet(key string, opts ...OpOption) Op
- func OpPut(key, val string, opts ...OpOption) Op
- func OpTxn(cmps []Cmp, thenOps []Op, elseOps []Op) Op

其实和直接调用KV.Put，KV.GET没什么区别。

下面是一个例子：

```
cli, err := clientv3.New(clientv3.Config{
    Endpoints:   endpoints,
    DialTimeout: dialTimeout,
})
if err != nil {
    log.Fatal(err)
}
defer cli.Close()

ops := []clientv3.Op{
    clientv3.OpPut("put-key", "123"),
    clientv3.OpGet("put-key"),
    clientv3.OpPut("put-key", "456")}

for _, op := range ops {
    if _, err := cli.Do(context.TODO(), op); err != nil {
        log.Fatal(err)
    }
}
```

把Op交给Do方法执行，返回的opResp结构如下：

```
type OpResponse struct {
    put *PutResponse
    get *GetResponse
    del *DeleteResponse
    txn *TxnResponse
}
```

你的操作是什么类型，你就用哪个指针来访问对应的结果。

Txn事务

etcd中事务是原子执行的，只支持if ... then ... else ...这种表达。首先来看一下Txn中定义的方法：

```
type Txn interface {
    // If takes a list of comparison. If all comparisons passed in succeed,
    // the operations passed into Then() will be executed. Or the operations
    // passed into Else() will be executed.
    If(cs ...Cmp) Txn

    // Then takes a list of operations. The Ops list will be executed, if the
    // comparisons passed in If() succeed.
    Then(ops ...Op) Txn

    // Else takes a list of operations. The Ops list will be executed, if the
    // comparisons passed in If() fail.
    Else(ops ...Op) Txn

    // Commit tries to commit the transaction.
    Commit() (*TxnResponse, error)
}
```

Txn必须是这样使用的：If(满足条件) Then(执行若干Op) Else(执行若干Op)。

If中支持传入多个Cmp比较条件，如果所有条件满足，则执行Then中的Op（上一节介绍过Op），否则执行Else中的Op。

首先，我们需要开启一个事务，这是通过KV对象的方法实现的：

```
txn := kv.Txn(context.TODO())
```

下面的测试程序，判断如果k1的值大于v1并且k1的版本号是2，则Put 键值k2和k3，否则Put键值k4和k5。

```
kv.Txn(context.TODO()).If(
 clientv3.Compare(clientv3.Value(k1), ">", v1),
 clientv3.Compare(clientv3.Version(k1), "=", 2)
).Then(
 clientv3.OpPut(k2,v2), clentv3.OpPut(k3,v3)
).Else(
 clientv3.OpPut(k4,v4), clientv3.OpPut(k5,v5)
).Commit()
```

类似于clientv3.Value()用于指定key属性的，有这么几个方法：

- func CreateRevision(key string) Cmp：key=xxx的创建版本必须满足...
- func LeaseValue(key string) Cmp：key=xxx的Lease ID必须满足...
- func ModRevision(key string) Cmp：key=xxx的最后修改版本必须满足...
- func Value(key string) Cmp：key=xxx的创建值必须满足...
- func Version(key string) Cmp：key=xxx的累计更新次数必须满足...

Watch

Watch用于监听某个键的变化, `Watch`调用后返回一个`WatchChan`，它的类型声明如下:

```
type WatchChan <-chan WatchResponse

type WatchResponse struct {
    Header pb.ResponseHeader
    Events []*Event

    CompactRevision int64

    Canceled bool

    Created bool
}
```

当监听的key有变化后会向`WatchChan`发送`WatchResponse`。Watch的典型应用场景是应用于系统配置的热加载，我们可以在系统读取到存储在etcd key中的配置后，用Watch监听key的变化。在单独的goroutine中接收WatchChan发送过来的数据，并将更新应用到系统设置的配置变量中，比如像下面这样在goroutine中更新变量appConfig，这样系统就实现了配置变量的热加载。

```
type AppConfig struct {
  config1 string
  config2 string
}

var appConfig Appconfig

func watchConfig(clt *clientv3.Client, key string, ss interface{}) {
    watchCh := clt.Watch(context.TODO(), key)
    go func() {
        for res := range watchCh {
            value := res.Events[0].Kv.Value
            if err := json.Unmarshal(value, ss); err != nil {
                fmt.Println("now", time.Now(), "watchConfig err", err)
                continue
            }
            fmt.Println("now", time.Now(), "watchConfig", ss)
        }
    }()
}

watchConfig(client, "config_key", &appConfig)
```

golang etcd clientv3的主要功能就是这些，希望能帮大家梳理出学习脉络，这样工作中应用到etcd时再看官方文档就会容易很多。