etcd/client

etcd/client is the Go client library for etcd.

For full compatibility, it is recommended to install released versions of clients using go modules.

Install

```
go get go.etcd.io/etcd/v3/client
```

Usage

```
package main

import (
    "log"
    "time"
    "context"

    "go.etcd.io/etcd/v3/client"
)

func main() {
    cfg := client.Config{
        Endpoints:               []string{"http://127.0.0.1:2379"},
        Transport:               client.DefaultTransport,
        // set timeout per request to fail fast when the target endpoint is
unavailable
        HeaderTimeoutPerRequest: time.Second,
    }
    c, err := client.New(cfg)
    if err != nil {
        log.Fatal(err)
    }
    kapi := client.NewKeysAPI(c)
    // set "/foo" key with "bar" value
    log.Print("Setting '/foo' key with 'bar' value")
    resp, err := kapi.Set(context.Background(), "/foo", "bar", nil)
    if err != nil {
        log.Fatal(err)
    } else {
        // print common key info
        log.Printf("Set is done. Metadata is %q\n", resp)
    }
    // get "/foo" key's value
```

```
    log.Print("Getting '/foo' key value")
    resp, err = kapi.Get(context.Background(), "/foo", nil)
    if err != nil {
        log.Fatal(err)
    } else {
        // print common key info
        log.Printf("Get is done. Metadata is %q\n", resp)
        // print value
        log.Printf("%q key has %q value\n", resp.Node.Key, resp.Node.Value)
    }
}
```

Error Handling

etcd client might return three types of errors.

- context error

Each API call has its first parameter as `context`. A context can be canceled or have an attached deadline. If the context is canceled or reaches its deadline, the responding context error will be returned no matter what internal errors the API call has already encountered.

- cluster error

Each API call tries to send request to the cluster endpoints one by one until it successfully gets a response. If a requests to an endpoint fails, due to exceeding per request timeout or connection issues, the error will be added into a list of errors. If all possible endpoints fail, a cluster error that includes all encountered errors will be returned.

- response error

If the response gets from the cluster is invalid, a plain string error will be returned. For example, it might be a invalid JSON error.

Here is the example code to handle client errors:

```
cfg := client.Config{Endpoints:
[]string{"http://etcd1:2379","http://etcd2:2379","http://etcd3:2379"}}
c, err := client.New(cfg)
if err != nil {
    log.Fatal(err)
}

kapi := client.NewKeysAPI(c)
resp, err := kapi.Set(ctx, "test", "bar", nil)
if err != nil {
    if err == context.Canceled {
        // ctx is canceled by another routine
    } else if err == context.DeadlineExceeded {
        // ctx is attached with a deadline and it exceeded
    } else if cerr, ok := err.(*client.ClusterError); ok {
        // process (cerr.Errors)
    } else {
        // bad cluster endpoints, which are not etcd servers
```

```
        }
}
```

Caveat

1. etcd/client prefers to use the same endpoint as long as the endpoint continues to work well. This saves socket resources, and improves efficiency for both client and server side. This preference doesn't remove consistency from the data consumed by the client because data replicated to each etcd member has already passed through the consensus process.

2. etcd/client does round-robin rotation on other available endpoints if the preferred endpoint isn't functioning properly. For example, if the member that etcd/client connects to is hard killed, etcd/client will fail on the first attempt with the killed member, and succeed on the second attempt with another member. If it fails to talk to all available endpoints, it will return all errors happened.

3. Default etcd/client cannot handle the case that the remote server is SIGSTOPed now. TCP keepalive mechanism doesn't help in this scenario because operating system may still send TCP keep-alive packets. Over time we'd like to improve this functionality, but solving this issue isn't high priority because a real-life case in which a server is stopped, but the connection is kept alive, hasn't been brought to our attention.

4. etcd/client cannot detect whether a member is healthy with watches and non-quorum read requests. If the member is isolated from the cluster, etcd/client may retrieve outdated data. Instead, users can either issue quorum read requests or monitor the /health endpoint for member health information.