ETCD 架构解析

王永刚

2019 年 12 月 31 日

目录

第一章　etcd 编译安装

1.1　golang 环境

当然在安装 etcd 前要先安装 go。并且设置好 GOPATH。可以用 apt,yum 或者源码安装，下载二进制安装等方式。从官网下载 golang 源码包。https://golang.org/dl/。

```
root@dockervm:~/go/src# go env
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/root/.cache/go-build"
GOENV="/root/.config/go/env"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GONOPROXY=""
GONOSUMDB=""
GOOS="linux"
GOPATH="/root/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/local/go"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
AR="ar"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0
ches"
```

图 1.1.1: golang 环境变量

```
wget https://golang.org/doc/install?download=go1.13.5.linux-amd64.tar.gz
tar xvf go1.13.5.linux-amd64.tar.gz -C /usr/local
并在 vim ~/.bashrc 加入
export PATH=$PATH:/user/local/go/bin
```

1.2 etcd 编译安装

```
#etcd 编译
mkdir -p  $GOPATH/src/go.etcd.io/
cd $GOPATH/src/go.etcd.io/
git clone https://github.com/etcd-io/etcd.git
./build
./bin/etcd
```

```
root@dockervm:~/go/src/go.etcd.io/etcd# ./bin/etcd
[WARNING] Deprecated '--logger=capnslog' flag is set; use '--logger=zap' flag instead
2019-12-12 12:56:57.536110 I | etcdmain: etcd Version: 3.5.0-pre
2019-12-12 12:56:57.536652 I | etcdmain: Git SHA: 378b05b8d
2019-12-12 12:56:57.536929 I | etcdmain: Go Version: go1.10.4
2019-12-12 12:56:57.537314 I | etcdmain: Go OS/Arch: linux/amd64
2019-12-12 12:56:57.537647 I | etcdmain: setting maximum number of CPUs to 1, total number of available CPUs is 1
2019-12-12 12:56:57.538025 W | etcdmain: no data-dir provided, using default data-dir ./default.etcd
[WARNING] Deprecated '--logger=capnslog' flag is set; use '--logger=zap' flag instead
2019-12-12 12:56:57.540593 I | embed: name = default
2019-12-12 12:56:57.541455 I | embed: data dir = default.etcd
2019-12-12 12:56:57.542031 I | embed: member dir = default.etcd/member
2019-12-12 12:56:57.542372 I | embed: heartbeat = 100ms
2019-12-12 12:56:57.542686 I | embed: election = 1000ms
2019-12-12 12:56:57.542978 I | embed: snapshot count = 100000
2019-12-12 12:56:57.543339 I | embed: advertise client URLs = http://localhost:2379
2019-12-12 12:56:57.861111 I | etcdserver: starting member 8e9e05c52164694d in cluster cdf818194e3a8c32
raft2019/12/12 12:56:57 INFO: 8e9e05c52164694d switched to configuration voters=()
raft2019/12/12 12:56:57 INFO: 8e9e05c52164694d became follower at term 0
raft2019/12/12 12:56:57 INFO: newRaft 8e9e05c52164694d [peers: [], term: 0, commit: 0, applied: 0, lastindex: 0, lastterm: 0]
raft2019/12/12 12:56:57 INFO: 8e9e05c52164694d became follower at term 1
raft2019/12/12 12:56:57 INFO: 8e9e05c52164694d switched to configuration voters=(10276657743932975437)
2019-12-12 12:56:58.012523 W | auth: simple token is not cryptographically signed
2019-12-12 12:56:58.128437 I | etcdserver: starting server... [version: 3.5.0-pre, cluster version: to_be_decided]
2019-12-12 12:56:58.134720 I | etcdserver: 8e9e05c52164694d as single-node; fast-forwarding 9 ticks (election ticks 10)
2019-12-12 12:56:58.135431 I | embed: listening for peers on 127.0.0.1:2380
raft2019/12/12 12:56:58 INFO: 8e9e05c52164694d switched to configuration voters=(10276657743932975437)
2019-12-12 12:56:58.136818 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhost:2380] to cluster cdf81819
4e3a8c32
raft2019/12/12 12:56:58 INFO: 8e9e05c52164694d is starting a new election at term 1
raft2019/12/12 12:56:58 INFO: 8e9e05c52164694d became candidate at term 2
raft2019/12/12 12:56:58 INFO: 8e9e05c52164694d received MsgVoteResp from 8e9e05c52164694d at term 2
raft2019/12/12 12:56:58 INFO: 8e9e05c52164694d became leader at term 2
raft2019/12/12 12:56:58 INFO: raft.node: 8e9e05c52164694d elected leader 8e9e05c52164694d at term 2
2019-12-12 12:56:58.263397 I | etcdserver: setting up the initial cluster version to 3.5
2019-12-12 12:56:58.263744 I | etcdserver: published {Name:default ClientURLs:[http://localhost:2379]} to cluster cdf818194e3a8
c32
2019-12-12 12:56:58.264386 I | embed: ready to serve client requests
2019-12-12 12:56:58.265136 N | embed: serving insecure client requests on 127.0.0.1:2379, this is strongly discouraged!
2019-12-12 12:56:58.295344 N | etcdserver/membership: set the initial cluster version to 3.5
2019-12-12 12:56:58.295855 I | etcdserver/api: enabled capabilities for version 3.5
```

图 1.2.2: etcd 运行

1.3 etcd 编译文件 build 分析

etcd 的 build 文件如图4.3.3所示。golang 的编译非常简洁快速。直接编译出了 etcd 和 etcdctl 两个可执行文件。

1.4 etcd 版本

本书开写时的 etcd 的最新版本。如果看代码，要用与本书一致比较好。

```
etcd_build() {                                  文件：go.etcd.io/etcd/build
    out="bin"
    if [[ -n "${BINDIR}" ]]; then out="${BINDIR}"; fi
    toggle_failpoints_default

    # Static compilation is useful when etcd is run in a container. $GO_BUILD_FLAGS is OK
    # shellcheck disable=SC2086
    CGO_ENABLED=0 go build $GO_BUILD_FLAGS \        build etcd: etcd/main.go -> etcd/etcdmain/etcdmain.go
        -installsuffix cgo \
        -ldflags "$GO_LDFLAGS" \
        -o "${out}/etcd" ${REPO_PATH} || return
    # shellcheck disable=SC2086
    CGO_ENABLED=0 go build $GO_BUILD_FLAGS \
        -installsuffix cgo \
        -ldflags "$GO_LDFLAGS" \
        -o "${out}/etcdctl" ${REPO_PATH}/etcdctl || return
}
```

图 1.3.3: etcd build

```
root@dockervm:~/go/src/go.etcd.io/etcd# ./bin/etcd --version
etcd Version: 3.5.0-pre
Git SHA: 378b05b8d
Go Version: go1.10.4
Go OS/Arch: linux/amd64
root@dockervm:~/go/src/go.etcd.io/etcd#
```

图 1.4.4: etcd 版本

1.5　学习目标与方法

学习 ETCD 是学的什么？怎么学？当然是 etcd 的知识。在学习过程中要多问自己问题，比如 ETCD 启动流程是什么？ETCD 是如何处理客户端请求的？ETCD 提供了什么服务？ 问题 –> 分析源码 –> 答案问题与答案便是监督学习的训练集。分析源码是有方法的，通过各种有效方法分析才能尽快的找到答案。那么，如何分析源码又转化问题，他的答案呢？

经过训练，大脑里会形成一个关于 ETCD 问题的模型，你以为你很了解这个模型，其实不然，至于大脑中形成了什么样的模型，目前无法知道。

如何知道自己学会了没。方法 1：讲出来，做笔记，或者讲给自己听。(解释法) 方法 2：网上查：ETCD 面试题，有很多面试题，看看自己能答出不？(测试集) 很不幸的是测试集一旦看过一遍，就会变成训练集。除非看了之后，不要记住，忘掉。方法 3：应用到实践当中去。

1.6　ETCD 应用

1.6.1　服务发现

1.6.2　配置管理

第二章　etcd server 分析

2.1　源码分析方法

2.1.1　配置文件法

一般在看一个服务器的代码，可以从其配置文件入手。比如 etcd, 我们知道 etcd 服务器启动后必然会监听一个端口来与客户端通信，同时也会监听一个端口与其他 etcd 服务器通信 (这是因为 RAFT 算法中需要多个结点)。etcd 的启动配置见附录.1

2.1.2　日志法

看 server 的日志是了解代码执行路径的比较好的办法。可以偿试打开 server 的 DEBUG 日志，得到更新详细的日志。还可以自己手动在代码加入日志，这就需要重新编译 server。但是像 golang 这种编译速度比较快的可以一试。如果是 C++ 等编译比较慢的语言，将严重影响分析源码的效率。

2.1.3　调试跟踪法

在进一步跟踪代码的过程，经常会遇到虚函数, 接口，框架。导致跟踪断开。这也表示，代码走到了更低一层。而一般开源代码也不会有文档明确指出这一层提供的所有接口，以及用法。这就需要用调试器进一步跟踪。可以利用调试器的 watch 指令来监控某个 socket 或者说变量的引用。同时在客户端发送请求。就可以进一步跟踪到框架底层。

2.1.4　框架预习法

如果代码中使用了框架，先花上几小时熟悉这个框架的用法。甚至花几天去熟习也很值。比如 etcd 中的用了 GRPC 框架，用了 http2, 用了 golang 自带的 http2 的库。如果对这些不熟悉，那分析源码将会举步维艰。

2.1.5　GRPC 框架

GRPC 框架支持多语言, 并且使用了 ProtoBuffer。ProtoBuffer 是一种高效的序列化反序列化通信协议。增加字段后能兼容增加字段前的协议，再也不用担心一个协议字段的添加而导致客户端服务器转文不兼容了。同时，ProtoBuffer 定义了一种新语法来定义协议字段。这些协议的定义会被放在 proto 文件里。再由

protoc 命令加一些插件来生成不同语言的代码。比如 golang, 就会生成 pb.go。原来的 proto 文件中定义的 message 会被转换成 golang 的 struct。当然如果生成 java 代码就会转成 java 的 class 文件。

GRPC 不但通过 proto 来定义 RPC 通信协议，同时也定义 RPC 服务器客户端的接口 (其实就是一个个函数)。然后用 GRPC 的的插件 +protoc 来生成 RPC 服务器和客户端的代码。这此代码会把需要我们实现的 RPC 接口暴露出来，我们只需要实现这些接口就能实现 RPC 调用。同时客户端接口也已经生成，我们自己只要调用生成的客户端接口，就能调用远程 RPC 服务器上的代码。Proto 定义如下图。

proto 定义：

```
// Copyright 2015 gRPC authors.

syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

服务器实现:

```
/*
 * Copyright 2015 gRPC authors.
 */

//go:generate protoc -I ../helloworld --go_out=plugins=grpc:../helloworld
↪  ../helloworld/helloworld.proto

// Package main implements a server for Greeter service.
package main

import (
        "context"
        "log"
        "net"

        "google.golang.org/grpc"
        pb "google.golang.org/grpc/examples/helloworld/helloworld"
)

const (
        port = ":50051"
)

// server is used to implement helloworld.GreeterServer.
type server struct {
        pb.UnimplementedGreeterServer
```

```
}

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest)
↪  (*pb.HelloReply, error) {
        log.Printf("Received: %v", in.GetName())
        return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}

func main() {
        lis, err := net.Listen("tcp", port)
        if err != nil {
                log.Fatalf("failed to listen: %v", err)
        }
        s := grpc.NewServer()
        pb.RegisterGreeterServer(s, &server{})
        if err := s.Serve(lis); err != nil {
                log.Fatalf("failed to serve: %v", err)
        }
}
```

客户端调用:

```
/*
 * Copyright 2015 gRPC authors.
 */

// Package main implements a client for Greeter service.
package main

import (
        "context"
        "log"
        "os"
        "time"

        "google.golang.org/grpc"
        pb "google.golang.org/grpc/examples/helloworld/helloworld"
)

const (
        address     = "localhost:50051"
        defaultName = "world"
)

func main() {
        // Set up a connection to the server.
        conn, err := grpc.Dial(address, grpc.WithInsecure(),
        ↪  grpc.WithBlock())
        if err != nil {
                log.Fatalf("did not connect: %v", err)
        }
        defer conn.Close()
        c := pb.NewGreeterClient(conn)

        // Contact the server and print out its response.
        name := defaultName
        if len(os.Args) > 1 {
                name = os.Args[1]
        }
        ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
        if err != nil {
                log.Fatalf("could not greet: %v", err)
        }
        log.Printf("Greeting: %s", r.GetMessage())
}
```

2.1.6 GRPC 安装

国内可能访问不了 google 的某些域名，可以从 github 来下载包。然后移动到对应的目录下。如果编译时发现缺少某些包，还是如这般操作，直到把依赖的包都下载下来.

```
#GRPC 安装

#protoc 的 golang 插件
go get github.com/golang/protobuf/protoc-gen-go
#golang 测试框架
go get github.com/golang/mock/gomock
#GRPC, 先从 github 上下载, 然后移动并重命名到 src/google.golang.org/grpc
go get github.com/grpc/grpc-go ; mkdir -p $(GOPATH)/src/google.golang.org; mv
↪  $(GOPATH)/src/github.com/grpc/grpc-go
↪  $(GOPATH)/src/google.golang.org/grpc

#GRPC 依赖库安装

#x 库
GOPATH=`go env GOPATH`
mkdir -p $(GOPATH)/src/golang.org/x
cd $(GOPATH)/src/golang.org/x
git clone https://github.com/golang/net.git net
git clone https://github.com/golang/text
git clone https://github.com/golang/sys

#GRPC 生成 GRPC 服务器用的库
cd $(GOPATH)/src/google.golang.org
git clone https://github.com/googleapis/go-genproto.git genproto

# 跑个例子看有没有安装好
启动 server: go run
↪  google.golang.org/grpc/examples/helloworld/greeter_server/main.go
启动 client: go run
↪  google.golang.org/grpc/examples/helloworld/greeter_client/main.go
启动 test:   go test  -test.v
↪  google.golang.org/grpc/examples/helloworld/mock_helloworld/hw_mock_test.go
```

2.2 etcd 启动调用关系

etcd 服务器启动，会启动很模块，对外主要是处理客户端请求，对内有 etcd 多个服务器进程之间通信，也有 etcd 服务器内部的功能，比如 KV 存储，WAL 日志等。源码分析第一步，搞清楚 etcd 的启动初始化步骤。见图2.2.1和图2.2.2。

2.2.1 从监听地址出发分析 etcd 如何处理客户端请求

etcd 的启动参数里有–listen-client-urls ’http://localhost:2379” 这么一项，见附录.1。这便是 etcd 为处理客户端请求而监听的地址。通过搜索代码，得到2.2.1。然后找 cfg.ec.LCUrls 的引用位置。然后发现在【embed/etcd.go:ConfigureClientListeners】中使用了 LCUrs, 并且监听了这个地址。使用的就是 golang 的 net 库。Configure-ClientListeners 是在【embed/etcd.go】中的 StartEtcd(inCfg) 调用, 调用图见2.2.2。StartEtcd 是 etcd 服务器的总控。这里启动了所有的服务。由于 Golang 的协程特

```
(dlv) bt
0  0x0000000000b7db8b in go.etcd.io/etcd/etcdserver.(*EtcdServer).start
   at ./etcdserver/server.go:746
1  0x0000000000b7d8af in go.etcd.io/etcd/etcdserver.(*EtcdServer).Start
   at ./etcdserver/server.go:733
2  0x0000000000d307b9 in go.etcd.io/etcd/embed.StartEtcd
   at ./embed/etcd.go:228
3  0x0000000000d86ff0 in go.etcd.io/etcd/etcdmain.startEtcd
   at ./etcdmain/etcd.go:302
4  0x0000000000d85b6e in go.etcd.io/etcd/etcdmain.startEtcdOrProxyV2
   at ./etcdmain/etcd.go:144
5  0x0000000000d8fcff in go.etcd.io/etcd/etcdmain.Main
   at ./etcdmain/main.go:46
6  0x0000000000d94440 in main.main
   at ./main.go:28
7  0x000000000042c5e2 in runtime.main
   at /usr/lib/go-1.10/src/runtime/proc.go:198
8  0x0000000000459e51 in runtime.goexit
   at /usr/lib/go-1.10/src/runtime/asm_amd64.s:2361
```

图 2.2.1: etcd 的 dlv 调用栈

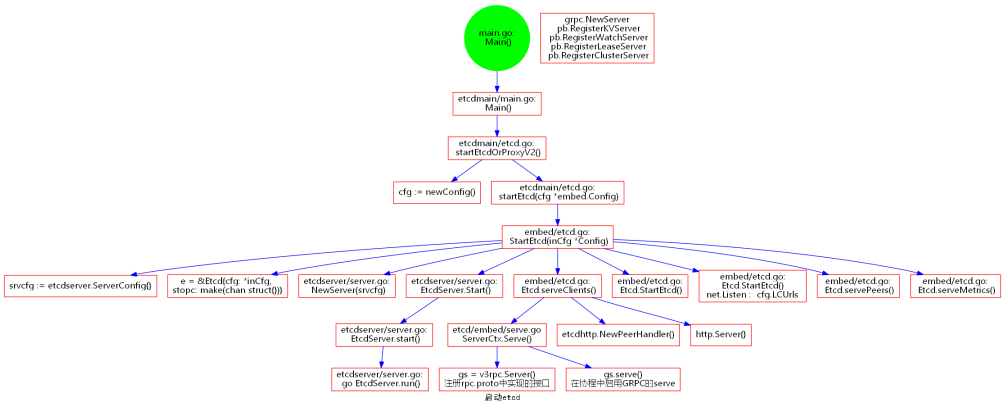

图 2.2.2: etcd 的启动初始化

性，可能在随时启动一个协程去服务。Start 可能只是在启动协议前做一些初始化，比如 HTTP 服务就需要提前注册各种处理 URL 的 handler。

从监听地址来分析，依然是细节太多。大概能搞清楚服务器的启动初始化流程。但是对于分析一次客户端的请求过程这样的事儿还是显示力不从心。

```
# 中使用 listen-client-urls
root@dockervm:~/go/src/go.etcd.io/etcd# find . -type f -name "*.go"|xargs
↪  grep -n 'listen-client-urls'
./etcdmain/config.go:139:
↪  flags.NewUniqueURLsWithExceptions(embed.DefaultListenClientURLs, ""),
↪  "listen-client-urls",
./etcdmain/config.go:330:       cfg.ec.LCUrls =
↪  flags.UniqueURLsFromFlag(cfg.cf.flagSet, "listen-client-urls")
# 使用 LCURLs 的代码
root@dockervm:~/go/src/go.etcd.io/etcd# find . -type f -name "*.go"|xargs
↪  grep -n 'LCUrls'
//下边两行是在 StartProxy 里的，如果不是 Proxy 就不会走到这里
./etcdmain/etcd.go:510: for _, u := range cfg.ec.LCUrls {
./etcdmain/etcd.go:531: for _, u := range cfg.ec.LCUrls {
./etcdmain/config.go:330:       cfg.ec.LCUrls =
↪  flags.UniqueURLsFromFlag(cfg.cf.flagSet, "listen-client-urls")
./tools/etcd-dump-metrics/etcd.go:55:   cfg.LCUrls, cfg.ACUrls = curls, curls
./embed/etcd.go:609:    for _, u := range cfg.LCUrls { # 这里是非 proxy 启动会
↪  走到的地方
./embed/config.go:181:  LPUrls, LCUrls []url.URL
```

```
//使用 listen-client-urls
```

```
2  func configureClientListeners(cfg *Config) (sctxs map[string]*serveCtx, err
   ↪  error) {
3      if err = updateCipherSuites(&cfg.ClientTLSInfo, cfg.CipherSuites); err !=
       ↪  nil {
4          return nil, err
5      }
6      if err = cfg.ClientSelfCert(); err != nil {
7          if cfg.logger != nil {
8              cfg.logger.Fatal("failed to get client self-signed certs",
               ↪  zap.Error(err))
9          } else {
10             plog.Fatalf("could not get certs (%v)", err)
11         }
12     }
13     if cfg.EnablePprof {
14         if cfg.logger != nil {
15             cfg.logger.Info("pprof is enabled", zap.String("path",
               ↪  debugutil.HTTPPrefixPProf))
16         } else {
17             plog.Infof("pprof is enabled under %s",
               ↪  debugutil.HTTPPrefixPProf)
18         }
19     }
20
21     sctxs = make(map[string]*serveCtx)
22     for _, u := range cfg.LCUrls {//可见可以监听多个地上，一般只有一个
23         sctx := newServeCtx(cfg.logger)
24         if u.Scheme == "http" || u.Scheme == "unix" {
25             // ......
26         }
27         if (u.Scheme == "https" || u.Scheme == "unixs") &&
           ↪  cfg.ClientTLSInfo.Empty() {
28             return nil, fmt.Errorf("TLS key/cert (--cert-file, --key-file)
               ↪  must be provided for client url %s with HTTPS scheme",
               ↪  u.String())
29         }
30
31         network := "tcp"
32         addr := u.Host
33         if u.Scheme == "unix" || u.Scheme == "unixs" {
34             network = "unix"
35             addr = u.Host + u.Path
36         }
37         sctx.network = network
38
39         sctx.secure = u.Scheme == "https" || u.Scheme == "unixs"
40         sctx.insecure = !sctx.secure
41         if oldctx := sctxs[addr]; oldctx != nil {
42             oldctx.secure = oldctx.secure || sctx.secure
43             oldctx.insecure = oldctx.insecure || sctx.insecure
44             continue
45         }
46         //这里监听了地址，sctx.l
47         if sctx.l, err = net.Listen(network, addr); err != nil {
48             return nil, err
49         }
50         // net.Listener will rewrite ipv4 0.0.0.0 to ipv6 [::], breaking
51         // hosts that disable ipv6. So, use the address given by the user.
52         sctx.addr = addr
53
54         if fdLimit, fderr := runtimeutil.FDLimit(); fderr == nil {
55             // ......
56             sctx.l = transport.LimitListener(sctx.l,
               ↪  int(fdLimit-reservedInternalFDNum))
57         }
58
59         if network == "tcp" {
60             if sctx.l, err = transport.NewKeepAliveListener(sctx.l, network,
               ↪  nil); err != nil {
61                 return nil, err
62             }
63         }
64
65         defer func() {
```

```
66              if err == nil {
67                  return
68              }
69              sctx.l.Close() //如果有什么错误就停止监听
70              // ......
71          }()
72          for k := range cfg.UserHandlers {
73              sctx.userHandlers[k] = cfg.UserHandlers[k]
74          }
75          sctx.serviceRegister = cfg.ServiceRegister
76          if cfg.EnablePprof || cfg.Debug {
77              sctx.registerPprof()
78          }
79          if cfg.Debug {
80              sctx.registerTrace()
81          }
82          sctxs[addr] = sctx //保存一个 sctx
83      }
84      return sctxs, nil
85  }

86  //在【embed/etcd.go】中的 StartEtcd(inCfg) 调用了 configureClientListeners
87  func StartEtcd(inCfg *Config) (e *Etcd, err error) {
88      //.......
89      serving := false
90      e = &Etcd{cfg: *inCfg, stopc: make(chan struct{})}
91      cfg := &e.cfg
92
93      //.......
94
95      //这里监听的地址是 server 之间通信的地址
96      if e.Peers, err = configurePeerListeners(cfg); err != nil {
97          return e, err
98      }
99
100     //.......
101
102     //这里监听的地址是 server 与 client 之间通信的地址
103     if e.sctxs, err = configureClientListeners(cfg); err != nil {
104         return e, err
105     }
106
107     //把 listen 状态的 conn 放到了 e.Clients 中，只用于服务器退出时关闭，可以检
        ↪ 查所有
108     //用到 e.Clients 的地方
109     for _, sctx := range e.sctxs {
110         e.Clients = append(e.Clients, sctx.l)
111     }
112
113     //.......
114
115     srvcfg := etcdserver.ServerConfig{
116         Name:                       cfg.Name,
117         ClientURLs:                 cfg.ACUrls,
118         PeerURLs:                   cfg.APUrls,
119         DataDir:                    cfg.Dir,
120         DedicatedWALDir:            cfg.WalDir,
121         //.......
122     }
123     print(e.cfg.logger, *cfg, srvcfg, memberInitialized)
124     if e.Server, err = etcdserver.NewServer(srvcfg); err != nil {
125         return e, err
126     }
127
128     //.......
129     //etcd 服务启动
130     e.Server.Start()
131     //server 之间服务启动
132     if err = e.servePeers(); err != nil {
133         return e, err
134     }
135     //处理客户端请求的服务启动
136     if err = e.serveClients(); err != nil {
137         return e, err
```

```
138         }
139
140         //
141         if err = e.serveMetrics(); err != nil {
142             return e, err
143         }
144
145         //......
146         serving = true
147         return e, nil
148     }
```

2.2.2 用调试器跟踪法继续分析客户端请求

此方法要谨慎使用，否则会陷入无限的细节当中不能自拔，进而会浪费大量时间。尤其是一开始分析源码的时候，会跟踪到底层，加上 golang 的协程交替执行，会让人一头雾水。

```
1  # 使用调试器 dlv 调试服务器监听端口
2  # 先安装好 dlv
3  # 同时修改 etcd/build 文件如下：和原来的相比加了-gcflags，这样 dlv 才能打印变量
   ↪  值
4  etcd_build() {
5  out="bin"
6  if [[ -n "${BINDIR}" ]]; then out="${BINDIR}"; fi
7  toggle_failpoints_default

8  # Static compilation is useful when etcd is run in a container.
   ↪  $GO_BUILD_FLAGS is OK
9  # shellcheck disable=SC2086
10 CGO_ENABLED=0 go build $GO_BUILD_FLAGS \
11 -gcflags=all="-N -l" \
12 -installsuffix cgo \
13 -ldflags "$GO_LDFLAGS" \
14 -o "${out}/etcd" ${REPO_PATH} || return
15 # shellcheck disable=SC2086
16 CGO_ENABLED=0 go build $GO_BUILD_FLAGS \
17 -gcflags=all="-N -l" \
18 -installsuffix cgo \
19 -ldflags "$GO_LDFLAGS" \
20 -o "${out}/etcdctl" ${REPO_PATH}/etcdctl || return
21 }

22 # 最后编译 ./build

23 # 开始调试
24 root@dockervm:~/go/src/go.etcd.io/etcd# dlv --check-go-version=false exec
   ↪  ./bin/etcd
25 Type 'help' for list of commands.
26 (dlv) b embed.serve
27 Breakpoint 1 set at 0xd3a9fb for go.etcd.io/etcd/embed.(*serveCtx).serve()
   ↪  ./embed/serve.go:85
28 (dlv) c
```

2.2.3 框架分析之 ETCD 的 GRPC 服务

这里才是分析源码最快的方法。没有一个服务器不使用框架，或使用自己写的框架。尤其是网络框架。比如 HTTP,GRPC。在分析源码前，一定要花时间先把框架学习了。至少把框架的使用方法学习了。手动写几个框架的小应用更好。当然，框架实现细节可以延后学习。就比如 ETCD 使用的 GRPC 框架。在学习 GRPC 框架前，无论是看代码，调试程序，都没法找到 ETCD 是如何处理客户端请求的。直到学了 GRPC 框架，才恍然大悟，原来 GRPC 框架要求所有 RPC 服

务接口都要定义在 proto 里。所以看了【etcdserver/etcdserverpb/rpc.proto】文件后对 ETCD 服务器提供的 RPC 服务一目了然。然后就是找到这些接口的实现和注册。看看生成的 rpc.pb.go 文件，了解了接口，和注册函数 RegisterKVServer。然后再代码里搜索一下 RegisterKVServer 得到【etcdserver/api/v3rpc/grpc.go:59】。同时我们知道 GRPC 的 server 的创建是调用 grpc.NewServer(), 同样搜索一下代码得到【./etcdserver/api/v3rpc/grpc.go:57: grpcServer := grpc.NewServer(append(opts, gopts...)...)】。最后在协程中调用 grpcServer.Serve() 开始服务。具体注册了哪些服务，见图2.2.3。

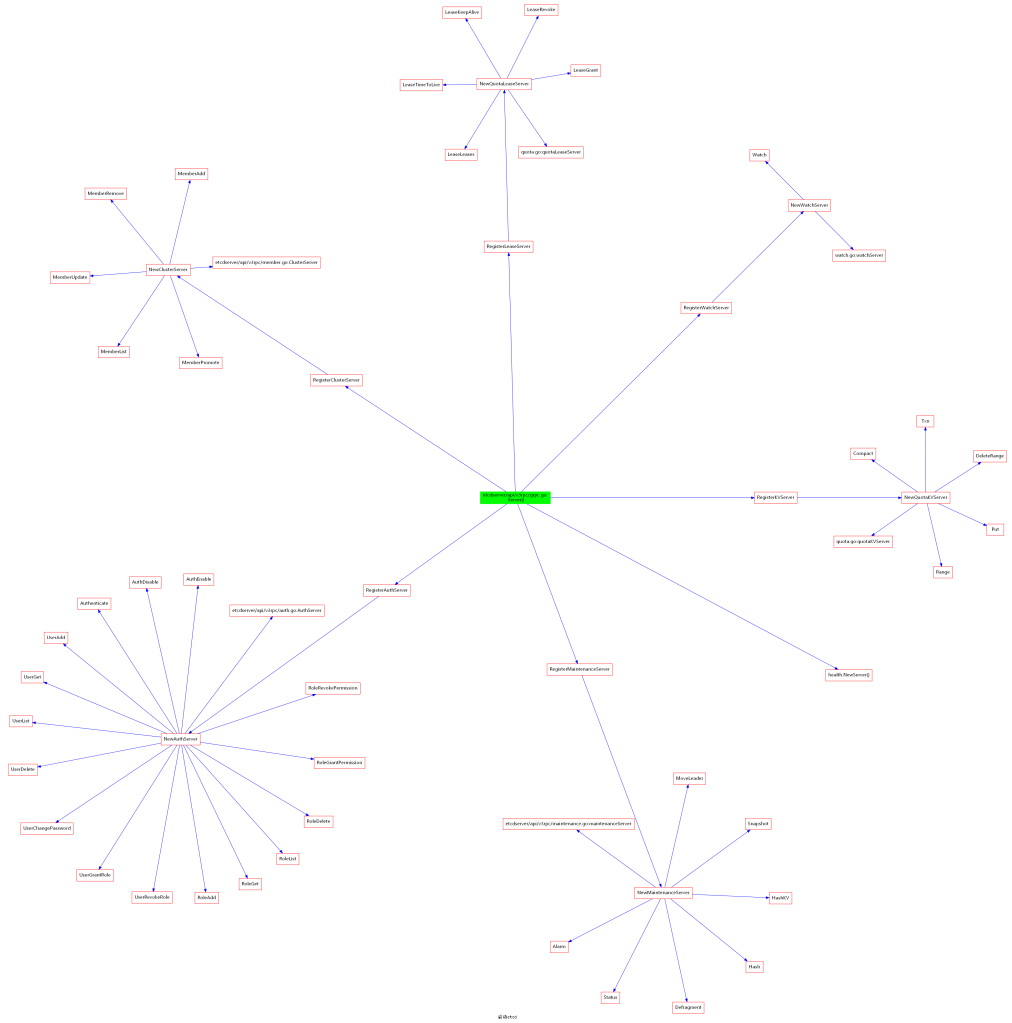

图 2.2.3: etcd 注册的 RPC 服务

```
# 搜索 RegisterKVServer
root@dockervm:~/go/src/go.etcd.io/etcd# find . -type f -name "*.go" |xargs
↪  grep -n -i 'RegisterKVServer'
./proxy/grpcproxy/kv_test.go:88:        pb.RegisterKVServer(kvts.server,
↪  kvts.kp)
./etcdmain/grpc_proxy.go:367:   pb.RegisterKVServer(server, kvp)
./pkg/mock/mockserver/mockserver.go:135:        pb.RegisterKVServer(svr,
↪  &mockKVServer{})
./etcdserver/api/v3rpc/grpc.go:59:      pb.RegisterKVServer(grpcServer,
↪  NewQuotaKVServer(s))
```

```
./etcdserver/etcdserverpb/rpc.pb.go:3535:func RegisterKVServer(s *grpc.Server, srv KVServer) {

# 搜索 grpc.NewServer
root@dockervm:~/go/src/go.etcd.io/etcd# find . -type f -name "*.go" |xargs grep -n -i 'grpc\.NewServer'
./proxy/grpcproxy/kv_test.go:87:        kvts.server = grpc.NewServer(opts...)
./proxy/grpcproxy/cluster_test.go:109:  cts.server = grpc.NewServer(opts...)
./etcdmain/grpc_proxy.go:361:   server := grpc.NewServer(
./pkg/mock/mockserver/mockserver.go:134:        svr := grpc.NewServer()
./etcdserver/api/v3rpc/grpc.go:57:      grpcServer := grpc.NewServer(append(opts, gopts...)...)
./vendor/github.com/grpc-ecosystem/go-grpc-middleware/doc.go:24:        myServer := grpc.NewServer(
./functional/agent/server.go:95:        srv.grpcServer = grpc.NewServer(opts...)

etcd/etcdserver/api/v3rpc/grpc.go 中 Server 的定义如下 。可以返回了 grpc 的 Server。
func Server(s *etcdserver.EtcdServer, tls *tls.Config, gopts ...grpc.ServerOption) *grpc.Server
```

```
//ETCD 提供了很多服务注册服务的代码。
func Server(s *etcdserver.EtcdServer, tls *tls.Config, gopts ...grpc.ServerOption) *grpc.Server {
        var opts []grpc.ServerOption
        opts = append(opts, grpc.CustomCodec(&codec{}))
        if tls != nil {
                bundle := credentials.NewBundle(credentials.Config{TLSConfig: tls})
                opts = append(opts, grpc.Creds(bundle.TransportCredentials()))
        }
        opts = append(opts, grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
        newLogUnaryInterceptor(s),
        newUnaryInterceptor(s),
        grpc_prometheus.UnaryServerInterceptor,
        )))
        opts = append(opts, grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
        newStreamInterceptor(s),
        grpc_prometheus.StreamServerInterceptor,
        )))
        opts = append(opts, grpc.MaxRecvMsgSize(int(s.Cfg.MaxRequestBytes+grpcOverheadBytes)))
        opts = append(opts, grpc.MaxSendMsgSize(maxSendBytes))
        opts = append(opts, grpc.MaxConcurrentStreams(maxStreams))
        //这里 new 了 GRPC 的服务
        grpcServer := grpc.NewServer(append(opts, gopts...)...)

        //这里注册了一系统功能
        pb.RegisterKVServer(grpcServer, NewQuotaKVServer(s))
        pb.RegisterWatchServer(grpcServer, NewWatchServer(s))
        pb.RegisterLeaseServer(grpcServer, NewQuotaLeaseServer(s))
        pb.RegisterClusterServer(grpcServer, NewClusterServer(s))
        pb.RegisterAuthServer(grpcServer, NewAuthServer(s))
        pb.RegisterMaintenanceServer(grpcServer, NewMaintenanceServer(s))

        // server should register all the services manually
        // use empty service name for all etcd services' health status,
        // see https://github.com/grpc/grpc/blob/master/doc/health-checking.md for more
        hsrv := health.NewServer()
        hsrv.SetServingStatus("", healthpb.HealthCheckResponse_SERVING)
        healthpb.RegisterHealthServer(grpcServer, hsrv)

        // set zero values for metrics registered for this grpc server
        grpc_prometheus.Register(grpcServer)

        return grpcServer
}
```

第三章　ETCD 应用场景

3.1　ETCD 集群

ETCD 从集群必须是奇数个结点，每个结点要监听两个端口，一个是用于接收客户端请求 (client-urls)，另一个用于接收集群中其他 ETCD 服务器的请求 (peer-urls)。可以用 docker 或者多个机器来部署。每个机器一个 ETCD 服务器。这里用于测试，就在一个机器的不同端口上启动了三个 ETCD 服务器。

```
# 启动三个 etcd 服务器组成一个集群
# 可以把下面的脚本放在
#root/go/src/go.etcd.io/etcd/etcd_cluster 目录下。然后执行，就能构建出一个集群了。其中/root/go 是 GOPATH 目录。
TOKEN=token-01
CLUSTER_STATE=new
NAME_1=m1
NAME_2=m2
NAME_3=m3
# 在同一个机器可以用不同的端口
HOST_1=127.0.0.1
HOST_2=127.0.0.1
HOST_3=127.0.0.1

CLIENT_PORT_1=2379
PEER_PORT_1=2380
CLIENT_PORT_2=2381
PEER_PORT_2=2382
CLIENT_PORT_3=2383
PEER_PORT_3=2384

#data 目录用来存放结点数据
DATA1=data1
DATA2=data2
DATA3=data3
CLUSTER=${NAME_1}="http://${HOST_1}:$PEER_PORT_1,\
${NAME_2}=http://${HOST_2}:$PEER_PORT_2,\
${NAME_3}=http://${HOST_3}:$PEER_PORT_3"

# 分别启动三个结点
$etcd --data-dir=$DATA1 --name ${NAME_1} \
--initial-advertise-peer-urls http://${HOST_1}:$PEER_PORT_1 \
--listen-peer-urls http://$HOST_1:$PEER_PORT_1 \
--advertise-client-urls http://$HOST_1:$CLIENT_PORT_1 \
--listen-client-urls http://${HOST_1}:$CLIENT_PORT_1 \
```

```
--initial-cluster ${CLUSTER} \
--initial-cluster-state ${CLUSTER_STATE} \
--initial-cluster-token ${TOKEN} &

$etcd --data-dir=$DATA2 --name ${NAME_2} \
--initial-advertise-peer-urls http://${HOST_2}:$PEER_PORT_2 \
--listen-peer-urls http://$HOST_2:$PEER_PORT_2 \
--advertise-client-urls http://$HOST_2:$CLIENT_PORT_2 \
--listen-client-urls http://${HOST_2}:$CLIENT_PORT_2 \
--initial-cluster ${CLUSTER} \
--initial-cluster-state ${CLUSTER_STATE} \
--initial-cluster-token ${TOKEN} &

$etcd --data-dir=$DATA3 --name ${NAME_3} \
--initial-advertise-peer-urls http://${HOST_3}:$PEER_PORT_3 \
--listen-peer-urls http://$HOST_3:$PEER_PORT_3 \
--advertise-client-urls http://$HOST_3:$CLIENT_PORT_3 \
--listen-client-urls http://${HOST_3}:$CLIENT_PORT_3 \
--initial-cluster ${CLUSTER} \
--initial-cluster-state ${CLUSTER_STATE} \
--initial-cluster-token ${TOKEN} &

#ENDPONTS 是给 etcdctl 用的
export ETCDCTL_API=3
export ENDPOINTS="$HOST_1:$CLIENT_PORT_1,\
$HOST_2:$CLIENT_PORT_2,\
$HOST_3:$CLIENT_PORT_3"

echo waiting for etcd server cluster init ...
sleep 5
echo
echo List all members in cluster:
$etcdctl --endpoints=$ENDPOINTS member list
echo
echo client used endpoints:
echo "$ENDPOINTS"
```

3.2 ETCD 客户端命令

ETCD 自还了一个 etcdctl 客户端工具。用它来熟悉基本功能十分好。

3.2.1 设置一些常用变量

```
EP=127.0.0.1:2379,127.0.0.1:2381,127.0.0.1:2383
alias ec="../bin/etcdctl --endpoints=$EP"
cd ~/go/src/go.etcd.io/etcd/etcd_cluster
#~/go 为 GOPATH 指定目录
```

3.2.2　基本的增删改查 (get/put/del)

```
../bin/etcdctl --endpoints=$EP put foo "Hello, World"
../bin/etcdctl --endpoints=$EP get foo
../bin/etcdctl --endpoints=$EP --write-out="json" get foo
```

3.2.3　带前缀的查删 (–prefix)

```
# 带前缀的查找:
../bin/etcdctl --endpoints=$EP put web1 value1
../bin/etcdctl --endpoints=$EP put web2 value2
../bin/etcdctl --endpoints=$EP put web3 value3
../bin/etcdctl --endpoints=$EP get web --prefix
# 输出:
web1
value1
web2
value2
web3
value3
# 带前缀的删除:
../bin/etcdctl --endpoints=$EP put key myvalue
../bin/etcdctl --endpoints=$EP put k1 value1
../bin/etcdctl --endpoints=$EP put k2 value2
# 删除所有以 k 开头的键
../bin/etcdctl --endpoints=$EP del k --prefix
```

3.2.4　事务 (txn)

```
# 实现类型下面代码的功能
#if value("user1") = "bad" :
#    del user1
#else:
#    put user1 good
ec put user1 bad
ec txn --interactive
compares:
value("user1") = "bad"

success requests (get, put, del):
del user1

failure requests (get, put, del):
put user1 good

SUCCESS

1
# 上边最终会执行 del user1
```

3.2.5 监听 (watch)

```
# 需要开两个终端
# 终端 1 执行下面命令会阻塞
ec watch stock --prefix
# 终端 2 执行
EP=127.0.0.1:2379,127.0.0.1:2381,127.0.0.1:2383
alias ec="../bin/etcdctl --endpoints=$EP"
ec put stock1 10
ec put stock2 20
# 终端 1 上会显示
PUT
stock1
10
PUT
stock2
20
```

3.2.6 租约 (lease)

租约可以认为是一个会定时删除的对象，这个对象超时删除时，会把与它绑定的所有 key 都连带删除。租约可以不断续租，也可以立即回收。

```
# 分配 lease
lease=`ec lease grant 5|awk '{print $2}'`
echo "Lease:" $lease

# 设置 k v 时关联一个 lease
ec put sample value --lease=$lease
echo "Get1:"
ec get sample

# 保持 lease 存在
ec lease keep-alive $lease &
# 或者可以立即收回 lease
#ec lease lease revoke $lease
sleep 10

# 依然可以获得
echo "Get2:"
ec get sample

# 停止保持
pkill etcdctl

#5 秒之后就自动删除了
echo "Sleep 6 seconds"
sleep 6
echo "Get3:"
ec get sample
```

3.2.7　分布式锁 (lock)

```
#term1 执行:
ec lock mutex1
mutex1/93e6f5a8fafa72c

# 打印出这个表示获得了锁，并阻塞在这里了
#term2 执行:
ec lock mutex1
# 直接阻塞了，因为锁被 term1 占用了

#term1 CTRL+C 停止，释放了锁
^C

#term2 此时输出:
mutex1/93e6f5a8fafa72f # 表示获得了锁，并阻塞在这里，一直战胜
```

3.2.8　选主 (elect)

```
#ETCD 多个服务器组成集群，其中要用一个来当 Leader。也就是用 RAFT
↪  算法选出 Leader。
ec elect one p1 &
ec elect one p2 &
```

3.2.9　查看集群状态 (status/health)

```
# 查看集群状态
ec --write-out=table endpoint status

将会输出:
 ENDPOINT|ID|VERSION| DB SIZE | IS LEADER | IS LEARNER |
 RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS

ec --write-out=table endpoint healt
# 将会输出:
ENDPOINT    | HEALTH |    TOOK    | ERROR
```

3.2.10　生成快照 (snapshot)

```
# 生成快照
../bin/etcdctl --endpoints=$ENDPOINT_ONE snapshot save my.db
# 查看快照
../bin/etcdctl --write-out=table --endpoints=$ENDPOINT_ONE
↪  snapshot status my.db
```

3.2.11　集群添加删除成员 (member remove/add)

删除老的成员，加一个新的成员，而不是把老成员加回来。

```
ec --write-out=table member list
```

```
# 得到 m2 的 member id
m2_id=`ec member list|grep m2|awk -F ',' '{print $1}'`

# 删除 member
ec member remove $m2_id
echo Sleep to wait removing $m2_id finish...
sleep 5

# 添加新结点
ec --write-out=table member list
../bin/etcdctl --endpoints=127.0.0.1:2379,127.0.0.1:2383 member
↪  add m4 \
--peer-urls=http://127.0.0.1:2386

echo "Sleep to wait add new member finish..."
sleep 5

# 启动新结点，注册 cluster 和 initial-cluster-state 变了
../bin/etcd --data-dir=data4 --name m4 \
--initial-advertise-peer-urls http://127.0.0.1:2386 \
--listen-peer-urls http://127.0.0.1:2386 \
--advertise-client-urls http://127.0.0.1:2385 \
--listen-client-urls http://127.0.0.1:2385 \
--initial-cluster "m1=http://127.0.0.1:2380,\
m4=http://127.0.0.1:2386,m3=http://127.0.0.1:2384" \
--initial-cluster-state existing \
--initial-cluster-token token-01 &

echo "Sleep to wait new node starting..."
sleep 5
ec --write-out=table member list
```

3.2.12 认证 (auth)

删除老的成员，加一个新的成员，而不是把老成员加回来。

```
# 添加角色
ec role add root
# 给角色授权
ec role grant-permission root readwrite foo
# 查看角色
ec role get root
# 必须有一个叫 root 的 user
ec user add root
# 让用户担任某角色
ec user grant-role root root
# 启用认证
ec auth enable
# 读写
ec --user=root:root put foo bar
```

```
15 ec --user=root:root get foo
16 # 停止认证
17 ec auth disable --user=root:root
```

第四章　etcd 编译安装

4.1　golang 环境

当然在安装 etcd 前要先安装 go。并且设置好 GOPATH。可以用 apt,yum 或者源码安装，下载二进制安装等方式。

```
root@dockervm:~/go/src# go env
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/root/.cache/go-build"
GOENV="/root/.config/go/env"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GONOPROXY=""
GONOSUMDB=""
GOOS="linux"
GOPATH="/root/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/local/go"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
AR="ar"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0
ches"
```

图 4.1.1: golang 环境变量

4.2 etcd 编译安装

```
#etcd 编译
mkdir -p  $GOPATH/src/go.etcd.io/
cd $GOPATH/src/go.etcd.io/
git clone https://github.com/etcd-io/etcd.git
./build
./bin/etcd
```

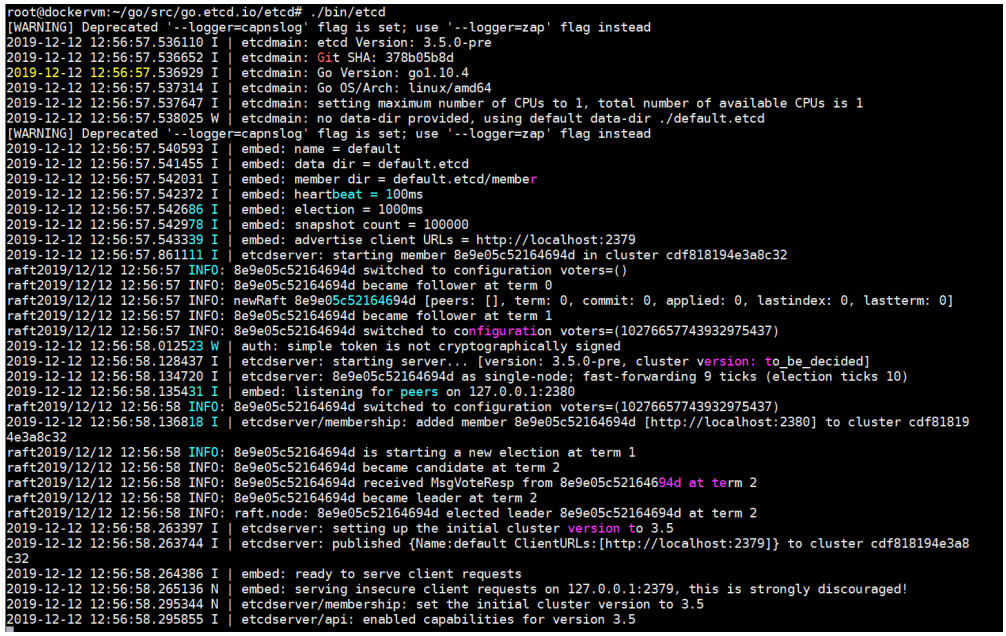

图 4.2.2: etcd 运行

4.3 etcd 编译文件 build 分析

etcd 的 build 文件如图4.3.3所示。golang 的编译非常简洁快速。直接编译出了 etcd 和 etcdctl 两个可执行文件。

4.4 etcd 版本

本书开写时的 etcd 的最新版本。如果看代码，要用与本书一致比较好。

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12

```
etcd_build() {                                  文件：go.etcd.io/etcd/build
    out="bin"
    if [[ -n "${BINDIR}" ]]; then out="${BINDIR}"; fi
    toggle_failpoints_default

    # Static compilation is useful when etcd is run in a container. $GO_BUILD_FLAGS is OK
    # shellcheck disable=SC2086
    CGO_ENABLED=0 go build $GO_BUILD_FLAGS \        build etcd: etcd/main.go -> etcd/etcdmain/etcdmain.go
        -installsuffix cgo \
        -ldflags "$GO_LDFLAGS" \
        -o "${out}/etcd" ${REPO_PATH} || return
    # shellcheck disable=SC2086
    CGO_ENABLED=0 go build $GO_BUILD_FLAGS \
        -installsuffix cgo \
        -ldflags "$GO_LDFLAGS" \
        -o "${out}/etcdctl" ${REPO_PATH}/etcdctl || return
}
```

图 4.3.3: etcd build

```
root@dockervm:~/go/src/go.etcd.io/etcd# ./bin/etcd --version
etcd Version: 3.5.0-pre
Git SHA: 378b05b8d
Go Version: go1.10.4
Go OS/Arch: linux/amd64
root@dockervm:~/go/src/go.etcd.io/etcd#
```

图 4.4.4: etcd 版本

月 Adobe PhotoshopCC2017 所有数据类型见表4.1。至于详情可以参考 [1] 和 [2]。Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软

表 4.1: OpenFLow 包格式

序号	类型	描述
1	Packet-In	发送到控制器
2	Packet-Out	发到交换机
3	Flow-Mod	修改流表项，增删改查, 控制器发送到交换机的

件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。跳转到第四章。这里是百度后面还有字这是楷体吗？

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop

CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。这是楷体吗？

$$f(x) = 3x^2 + 6(x-2) - 1 \tag{4-1}$$

$$g(x) = 4x^2 + 6(x-8) + 1 \tag{4-2}$$

$$E = mc^2 \tag{4-3}$$

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。质能公式如公式(4-3)。这是楷体吗？

4.4.1 画布操作

1. 新建画布　Ctrl + N
2. 画布切换　F
3. 复位工作区　Alt -> W -> K -> R
4. 放大缩小　Alt + 鼠标滚轮 (Ctrl + +, Ctrl + -)
5. 缩放工具　Z(放大：鼠标点击画布，或按下 Alt 点击画布，可以放大缩小)
6. 移动画布　Space + 鼠标左键按下拖动
7. 切换画布　Ctrl + Tab
8. 显示网格　Ctrl + ’
9. 显示参考线　Ctrl + ;
10. 显示标尺　Ctrl + R

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。这是楷体吗？

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工

具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。 Adobe Photoshop，简称“PS”，是

图 4.4.5: 风景 1

图 4.4.6: 风景 2

由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。如图4.4.5所示，又如图4.4.6所示。这是楷体吗？

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。这是楷体吗？

Adobe Photoshop，简称“PS”，是由 Adobe Systems 开发和发行的图像处理软件。Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以有效地进行图片编辑工作。PS 有很多功能，在图像、图形、文字、视频、出版等各方面都有涉及。2003 年，Adobe Photoshop 8 被更名为 Adobe Photoshop CS。2013 年 7 月，Adobe 公司推出了新版本的 Photoshop CC，自此，Photoshop CS6 作为 Adobe CS 系列的最后一个版本被新的 CC 系列取代。截止 2016 年 12 月 Adobe PhotoshopCC2017 为市场最新版本。这是楷体吗？

4.4.2 phtotshop 简介

参考文献

[1] 广西壮族自治区林业厅. 广西自然保护区 [M]. 北京: 中国林业出版社, 1993.

[2] International Federation of Library Association and Institutions. Names of persons: national usages for entry in catalogues[M]. 3rd ed. London: IFLA International Office for UBC, 1977.

[3] GANZHA V G, MAYR E W, VOROZHTSOV E V. Computer algebra in scientific computing: CASC 2000: proceedings of the Third Workshop on Computer Algebra in Scientific Computing, Samarkand, October 5-9, 2000[C]. Berlin: Springer, c2000.

附录

.1 ETCD 启动参数

etcd 配置参数

```
root@dockervm:~/go/src/go.etcd.io/etcd# pwd
/root/go/src/go.etcd.io/etcd
root@dockervm:~/go/src/go.etcd.io/etcd# ./bin/etcd --help
Usage:

etcd [flags]
Start an etcd server.

etcd --version
Show the version of etcd.

etcd -h | --help
Show the help information about etcd.

etcd --config-file
Path to the server configuration file. Note that if a
    ↪ configuration file is provided, other command line flags
    ↪ and environment variables will be ignored.

etcd gateway
Run the stateless pass-through etcd TCP connection forwarding
    ↪ proxy.

etcd grpc-proxy
Run the stateless etcd v3 gRPC L7 reverse proxy.


Member:
--name 'default'
Human-readable name for this member.
--data-dir '${name}.etcd'
Path to the data directory.
--wal-dir ''
Path to the dedicated wal directory.
--snapshot-count '100000'
Number of committed transactions to trigger a snapshot to disk.
--heartbeat-interval '100'
Time (in milliseconds) of a heartbeat interval.
--election-timeout '1000'
```

```
29 Time (in milliseconds) for an election to timeout. See tuning
   ↪ documentation for details.
30 --initial-election-tick-advance 'true'
31 Whether to fast-forward initial election ticks on boot for
   ↪ faster election.
32 --listen-peer-urls 'http://localhost:2380'
33 List of URLs to listen on for peer traffic.
34 --listen-client-urls 'http://localhost:2379'
35 List of URLs to listen on for client traffic.
36 --max-snapshots '5'
37 Maximum number of snapshot files to retain (0 is unlimited).
38 --max-wals '5'
39 Maximum number of wal files to retain (0 is unlimited).
40 --quota-backend-bytes '0'
41 Raise alarms when backend size exceeds the given quota (0
   ↪ defaults to low space quota).
42 --backend-bbolt-freelist-type 'map'
43 BackendFreelistType specifies the type of freelist that boltdb
   ↪ backend uses(array and map are supported types).
44 --backend-batch-interval ''
45 BackendBatchInterval is the maximum time before commit the
   ↪ backend transaction.
46 --backend-batch-limit '0'
47 BackendBatchLimit is the maximum operations before commit the
   ↪ backend transaction.
48 --max-txn-ops '128'
49 Maximum number of operations permitted in a transaction.
50 --max-request-bytes '1572864'
51 Maximum client request size in bytes the server will accept.
52 --grpc-keepalive-min-time '5s'
53 Minimum duration interval that a client should wait before
   ↪ pinging server.
54 --grpc-keepalive-interval '2h'
55 Frequency duration of server-to-client ping to check if a
   ↪ connection is alive (0 to disable).
56 --grpc-keepalive-timeout '20s'
57 Additional duration of wait before closing a non-responsive
   ↪ connection (0 to disable).

58 Clustering:
59 --initial-advertise-peer-urls 'http://localhost:2380'
60 List of this member's peer URLs to advertise to the rest of the
   ↪ cluster.
61 --initial-cluster 'default=http://localhost:2380'
62 Initial cluster configuration for bootstrapping.
63 --initial-cluster-state 'new'
64 Initial cluster state ('new' or 'existing').
65 --initial-cluster-token 'etcd-cluster'
66 Initial cluster token for the etcd cluster during bootstrap.
```

```
67  Specifying this can protect you from unintended cross-cluster
    ↪  interaction when running multiple clusters.
68  --advertise-client-urls 'http://localhost:2379'
69  List of this member's client URLs to advertise to the public.
70  The client URLs advertised should be accessible to machines
    ↪  that talk to etcd cluster. etcd client libraries parse
    ↪  these URLs to connect to the cluster.
71  --discovery ''
72  Discovery URL used to bootstrap the cluster.
73  --discovery-fallback 'proxy'
74  Expected behavior ('exit' or 'proxy') when discovery services
    ↪  fails.
75  "proxy" supports v2 API only.
76  --discovery-proxy ''
77  HTTP proxy to use for traffic to discovery service.
78  --discovery-srv ''
79  DNS srv domain used to bootstrap the cluster.
80  --discovery-srv-name ''
81  Suffix to the dns srv name queried when bootstrapping.
82  --strict-reconfig-check 'true'
83  Reject reconfiguration requests that would cause quorum loss.
84  --pre-vote 'false'
85  Enable to run an additional Raft election phase.
86  --auto-compaction-retention '0'
87  Auto compaction retention length. 0 means disable auto
    ↪  compaction.
88  --auto-compaction-mode 'periodic'
89  Interpret 'auto-compaction-retention' one of:
    ↪  periodic|revision. 'periodic' for duration based retention,
    ↪  defaulting to hours if no time unit is provided (e.g.
    ↪  '5m'). 'revision' for revision number based retention.
90  --enable-v2 'false'
91  Accept etcd V2 client requests.

92  Security:
93  --cert-file ''
94  Path to the client server TLS cert file.
95  --key-file ''
96  Path to the client server TLS key file.
97  --client-cert-auth 'false'
98  Enable client cert authentication.
99  --client-crl-file ''
100 Path to the client certificate revocation list file.
101 --client-cert-allowed-hostname ''
102 Allowed TLS hostname for client cert authentication.
103 --trusted-ca-file ''
104 Path to the client server TLS trusted CA cert file.
105 --auto-tls 'false'
106 Client TLS using generated certificates.
107 --peer-cert-file ''
```

```
108 Path to the peer server TLS cert file.
109 --peer-key-file ''
110 Path to the peer server TLS key file.
111 --peer-client-cert-auth 'false'
112 Enable peer client cert authentication.
113 --peer-trusted-ca-file ''
114 Path to the peer server TLS trusted CA file.
115 --peer-cert-allowed-cn ''
116 Required CN for client certs connecting to the peer endpoint.
117 --peer-cert-allowed-hostname ''
118 Allowed TLS hostname for inter peer authentication.
119 --peer-auto-tls 'false'
120 Peer TLS using self-generated certificates if --peer-key-file
    ↪ and --peer-cert-file are not provided.
121 --peer-crl-file ''
122 Path to the peer certificate revocation list file.
123 --cipher-suites ''
124 Comma-separated list of supported TLS cipher suites between
    ↪ client/server and peers (empty will be auto-populated by
    ↪ Go).
125 --cors '*'
126 Comma-separated whitelist of origins for CORS, or cross-origin
    ↪ resource sharing, (empty or * means allow all).
127 --host-whitelist '*'
128 Acceptable hostnames from HTTP client requests, if server is
    ↪ not secure (empty or * means allow all).

129 Auth:
130 --auth-token 'simple'
131 Specify a v3 authentication token type and its options
    ↪ ('simple' or 'jwt').
132 --bcrypt-cost 10
133 Specify the cost / strength of the bcrypt algorithm for hashing
    ↪ auth passwords. Valid values are between 4 and 31.

134 Profiling and Monitoring:
135 --enable-pprof 'false'
136 Enable runtime profiling data via HTTP server. Address is at
    ↪ client URL + "/debug/pprof/"
137 --metrics 'basic'
138 Set level of detail for exported metrics, specify 'extensive'
    ↪ to include server side grpc histogram metrics.
139 --listen-metrics-urls ''
140 List of URLs to listen on for the metrics and health endpoints.

141 Logging:
142 --logger 'capnslog'
143 Specify 'zap' for structured logging or 'capnslog'. [WARN]
    ↪ 'capnslog' will be deprecated in v3.5.
144 --log-outputs 'default'
```

```
145 Specify 'stdout' or 'stderr' to skip journald logging even when
    ↪ running under systemd, or list of comma separated output
    ↪ targets.
146 --log-level 'info'
147 Configures log level. Only supports debug, info, warn, error,
    ↪ panic, or fatal.

148 v2 Proxy (to be deprecated in v4):
149 --proxy 'off'
150 Proxy mode setting ('off', 'readonly' or 'on').
151 --proxy-failure-wait 5000
152 Time (in milliseconds) an endpoint will be held in a failed
    ↪ state.
153 --proxy-refresh-interval 30000
154 Time (in milliseconds) of the endpoints refresh interval.
155 --proxy-dial-timeout 1000
156 Time (in milliseconds) for a dial to timeout.
157 --proxy-write-timeout 5000
158 Time (in milliseconds) for a write to timeout.
159 --proxy-read-timeout 0
160 Time (in milliseconds) for a read to timeout.

161 Experimental feature:
162 --experimental-initial-corrupt-check 'false'
163 Enable to check data corruption before serving any client/peer
    ↪ traffic.
164 --experimental-corrupt-check-time '0s'
165 Duration of time between cluster corruption check passes.
166 --experimental-enable-v2v3 ''
167 Serve v2 requests through the v3 backend under a given prefix.
168 --experimental-enable-lease-checkpoint 'false'
169 ExperimentalEnableLeaseCheckpoint enables primary lessor to
    ↪ persist lease remainingTTL to prevent indefinite
    ↪ auto-renewal of long lived leases.
170 --experimental-compaction-batch-limit 1000
171 ExperimentalCompactionBatchLimit sets the maximum revisions
    ↪ deleted in each compaction batch.
172 --experimental-peer-skip-client-san-verification 'false'
173 Skip verification of SAN field in client certificate for peer
    ↪ connections.

174 Unsafe feature:
175 --force-new-cluster 'false'
176 Force to create a new one-member cluster.

177 CAUTIOUS with unsafe flag! It may break the guarantees given by
    ↪ the consensus protocol!

178 TO BE DEPRECATED:
```

```
179 --debug 'false'
180 Enable debug-level logging for etcd. [WARN] Will be deprecated
    ↪ in v3.5. Use '--log-level=debug' instead.
181 --log-package-levels ''
182 Specify a particular log level for each etcd package (eg:
    ↪ 'etcdmain=CRITICAL,etcdserver=DEBUG').
```

致谢

谢谢！

作者简介