

化茧成蝶

GO

在FreeWheel 服务化中的实践



■ 卷首语

在 Go 开源之后的 7 年里，它已被全球采用。可以看到全球一批顶尖企业，如 Adobe、IBM、Intel 等都开始使用 Go 开发自己的服务，甚至完全转向 Go 开发。而令人惊讶的是，随着数据科学和机器学习技术的发展和成熟，Go 有了更大的展示舞台，它的易部署等特性也正推动其向现代架构走来。

2014 年，FreeWheel 开始使用 Go 语言。最初是用于重写某一视频信息上传程序以提升其性能，后来因其接近 C 语言的执行效率，具备较低的学习曲线，能帮助实现简单的部署以及拥有丰富的类库，Go 语言也在 FreeWheel 越来越多的项目中被使用，至今已累积超过 10 万行代码。近一年来，FreeWheel 正在将业务系统迁移到微服务架构，而 Go 语言因其原生对 HTTP 服务开发的良好支持及易于容器化部署的特点，成为我们微服务开发的首选语言。目前 Go 社区中已有不少针对微服务中特定问题的项目——从 HTTP-to-RPC Mapping 到 ORM。但纵观下来，仍缺少成熟的构建微服务系统的整体方案。

为了快速地将业务迁移至微服务系统并保持基础架构的一致性，

FreeWheel 利用业界优秀的工具和经过实践开发了一个微服务框架——Wheels，以解决业务服务中的共性问题，包括接口定义、访问控制、负载均衡、服务监控、数据库访问、事件消息、容器化部署和文档管理等。

在开始服务化改造之前，FreeWheel 的业务系统是一个基于 Ruby on Rails 的单体应用，大约有 60 万行代码，但其内部根据业务分为若干模块，且边界比较清晰，这也成为我们设计服务拆分时的基础。我们还有一个主要考量是尽量降低服务化带来的额外运维成本，这对于前期推动服务化非常必要。基于上述两点，我们设定了 FreeWheel 服务化的基本原则：

1. 服务分为业务服务和基础服务两种，业务服务拆分基于核心业务概念，从粗粒度开始，将相关功能都封装在同一服务内，尽量避免业务服务之间的直接调用；
2. 不提供跨业务服务的ACID性质的事务支持；
3. 用事件（Event）来传递服务状态的变化；
4. 服务通过容器来部署，并使用容器管理系统来运维；
5. 使用统一语言（Go）和统一框架（后面会提到的Wheels）。

Go 语言已经在我们向服务化迁移的进程中发挥了重大作用，无论是其简洁高效的语法，还是高性能的运行和并发支持，都为我们的开发和运维带来了极大的便利。我们将在公司继续推进 Go 语言的深入研究和挖掘，相信它也将在未来的舞台上扮演更重要的角色。

目录 | Contents

- 5 FreeWheel 服务化与 Wheels 框架从 0 到 1 实践**
- 18 深入理解 Go 语言对象模型 (Go Object Model Inside Out)**
- 34 说一说 Go 语言并发模型中的 Channel**
- 48 ETCD 背后分布式存储实现**
- 74 gRPC 实践中的两个问题与思考**

FreeWheel 服务化与 Wheels 框架 从 0 到 1 实践

序言

目前 Go 社区中已有不少针对微服务中特定问题的项目，从 HTTP-to-RPC Mapping 到 ORM，但还缺少成熟的构建微服务系统的整体方案，为了快速的将业务迁移至微服务系统，并保持基础架构的一致性，FreeWheel 利用业界优秀的工具和实践开发了一个微服务框架——Wheels，以解决业务服务中的共性问题，包括接口定义、访问控制、负载均衡、服务监控、数据库访问、事件消息、容器化部署、文档管理等。该框架还在不断完善中，但我们还是希望尽早与同行分享经验，互相借鉴。

FreeWheel 服务化的基本设计

在开始服务化改造之前，FreeWheel 的业务系统是一个基于 Ruby on Rails 的单体应用，大约有 60 万行代码，但其内部根据业务分为若干模块，且边界比较清晰，这也成为我们设计服务拆分时的基础。我们还有一个主

要考量是尽量降低服务化带来的额外运维成本，这对于前期推动服务化非常必要。基于上述两点， 我们设定了 FreeWheel 服务化的基本原则：

1. 服务分为业务服务和基础服务两种，业务服务拆分基于核心业务概念，从粗粒度开始，将相关功能都封装在同一服务内，尽量避免业务服务之间的直接调用；
2. 不提供跨业务服务的ACID性质的事务支持；
3. 用事件（Event）来传递服务状态的变化；
4. 服务通过容器来部署，并使用容器管理系统来运维；
5. 使用统一语言（Go）和统一框架（后面会提到的Wheels）。

这些原则基本与主流微服务的架构思想一致。基于本文开篇所提及的，目前 Go 社区在构建微服务系统的成熟整体方案上的缺乏，FreeWheel 才有了开发 Wheels 的想法，目标就是实现快速的将 FreeWheel 的业务迁移到微服务架构，并保持基础架构的一致性。

Wheels 简介

Wheels 是一个微服务框架，目标是简化微服务的开发。它试图解决微服务架构的通用问题，可以让开发人员集中精力在具体的业务模块开发上。

设计目标

我们希望这个框架能提供通用的约定、可重用的基础组件和一致性的抽象。它也应该能加速我们开发的生产效率，就像我们之前使用 Ruby on Rails 在 Web 开发中获得的体验一样。

在开发 Wheels 之前，我们也尝试了很多其他的 Go 语言的 Web 框架。但这些框架也都存在着各自的问题，或者复杂度过高，或者不适合我们的业务场景，所以我们最终决定开发自己的框架。

特性

集成最佳实践

Wheels 集成了一系列提炼自于 Go 社区的通用惯例、通用依赖包和最佳实践，希望提供一种易理解的、强壮可靠的方法去组织项目和构建微服务。

这些最佳实践包括了项目结构、包管理、配置管理、环境设置、日志记录等等。

设计优先 (Design First)

Wheels 鼓励你在实现具体的功能前仔细周全地进行设计。我们选用了 Google 的 Protobuf(以及部分 gRPC 扩展) 作为设计语言。

比如如果我们希望发布一个简单的 Hello 服务，我们首先使用 Protobuf 描述该服务：

```
service HelloService {
    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;
}
```

当设计完成后，我们可以使用 Wheels 来帮我们自动生成代码。自动生成的代码包括服务的脚手架以及相关的依赖代码，这样开发人员可以聚焦在业务逻辑的开发上。

代码自动生成

Go 是一种强类型的静态语言。Go 的反射和元编程的能力有限，因此 Go 社区在很多时候倾向于使用代码生成的方式，去完成一些其他语言在运行时通过反射完成的功能。

我们也拥抱了 Go 社区的这一惯例。此外，因为受到 Ruby on Rails 框架的启发，我们使用了一种更精巧的方式，除了生成代码以外，还生成了项目文件和其他的依赖文件。

所以开发人员可以使用框架内置的代码生成工具，自动生成项目结构、Protobuf 描述文件、服务实现文件、测试文件、Swagger 文档等等。

支持多种协议

Wheels 支持 gRPC 协议。你可以使用 Wheels 快速开发 gRPC 服务。

gRPC 确实有很多优势。高效、易于使用且节省带宽，而且它已经被 Google 在许多大型项目中使用和验证。此外，gRPC 可以天然地与 Protobuf 描述紧密集成。

但是我们的服务在很多时候仍然需要使用经典的 RESTful API。我们的许多面向客户的 API 都是通过 HTTP 协议暴露的，我们需要维持向后的兼容性。

而 Wheels 也能帮助我们解决这个问题。它可以自动地透明地在 HTTP 和 gRPC 协议间互相转换。

监控

Wheels 内置了统一的监控，日志和服务性能指标收集的接口和方法。

部署

Wheels 也希望能简化整个部署的流程。它也能够很好地与 Docker 和 Kubernetes 集成。

快速开始

Wheels 命令行简介

当编译安装好 Wheels 后，你就可以在命令行运行 wheels -h 命令获取基本使用帮助。

```
$ wheels -h
```

The output should look like:

Usage:

```
wheels [command]
```

Available Commands:

console	Runs your Wheels app in a REPL console
generate	A collection of generators
new	Creates a new Wheels application
server	Starts the Wheels application server

Use "wheels [command] --help" for more information about a command.

接下来可以使用子命令加上 -h 参数获取子命令的使用细节，如：

```
$ wheels generate -h
```

It will output something like:

A collection of generators

Usage:

```
wheels generate [command]
```

Aliases:

```
generate, g
```

Available Commands:

proto	Generates new proto
-------	---------------------

```
service      Generates new service

Use "wheels generate [command] --help" for more information about
a command.
```

让我们进一步挖掘：

```
$ wheels generate service -h
```

我们在这里省略了输出。通过内置的帮助系统，实际上使用者可以很容易地获取所有命令的使用方法。

创建新项目

现在，让我们来创建一个叫 mysrv 的新项目，我们将在这个项目中开发我们的服务。

```
$ wheels new mysrv
```

执行该命令后，一个叫做 mysrv 的新目录将自动被创建。

```
$ tree mysrv
mysrv
├── app
│   └── services
├── config
│   ├── config.go
│   ├── config.yml
│   └── dev
│       └── config.yml
└── main.go
└── proto
```

创建服务描述

接下来，我们将生成一个使用 Protobuf 语言描述的服务定义文件。假设我们定义的是一个叫 hello 的服务

```
$ wheels generate proto hello
```

执行完该命令后，一个叫 hello.proto 的文件被自动生成在了项目的 proto 文件夹下。

```

├── app
|   └── services
└── config
    ├── config.go
    ├── config.yml
    └── dev
        └── config.yml
├── main.go
└── proto
    └── hello.proto

```

我们依靠自动生成的 hello.proto 模板并稍加编辑得到以下内容：

```

syntax = "proto3";

package proto;

import "google/api/annotations.proto";

service HelloService {
    rpc SayHello (HelloRequest) returns (HelloReply) {
        option (google.api.http) = { get: "/v1/hello/{name}" };
    }
}

message HelloRequest {
    string name = 1;
}

message HelloReply {
    string msg = 1;
}

```

利用服务描述生成实现代码

现在我们有了服务的基本定义，可以利用它来生成实际的实现代码骨架。

```
$ wheels generate service hello --proto hello
Some new files will be generated:
```

```
├── app
│   └── services
│       └── hello.go
└── config
    ├── config.go
    ├── config.yml
    └── dev
        └── config.yml
├── main.go
└── proto
    ├── hello.pb.go
    ├── hello.pb.gw.go
    └── hello.proto
```

我们注意到自动生成的 app/services/hello.go 文件，可以在其中实现我们的具体业务逻辑。

```
package services

import (
    proto "mysrv/proto"
    "golang.org/x/net/context"
)

func (s *Server) SayHello(ctx context.Context, req *proto.HelloRequest) (*proto.HelloReply, error) {
    // TODO: implement your logic here
}

Let's add some code in this function

package services
```

```

import (
    proto "[your_directory]/proto"
    "golang.org/x/net/context"
)

func (s *Server) SayHello(ctx context.Context, req *proto.HelloRequest) (*proto.HelloReply, error) {
    return &proto.HelloReply{Msg: "Hello, " + req.Name}, nil
}

```

运行服务

现在服务已经准备好运行了，让我们来试试：

```
$ go run main.go
```

好的，服务已经启动，并且监听在默认的 8080 端口上，可以在这个端口上同时服务 HTTP 和 gRPC 接口。

接下来，我们可以利用 Wheels 框架来自动生成 gRPC 客户端，通过 gRPC 协议访问服务。

```
$ wheels generate client hello --proto hello
```

当然它也是支持 HTTP 协议的：

```
$ curl http://localhost:8080/v1/hello/wheels
{"msg": "Hello, wheels"}
```

好了，我们现在已经快速实现了第一个非常简单的服务。现在可以再来实现一个稍微复杂点的服务。

添加另一个服务

现在我们再来添加一个服务，这个服务可以创建和查询用户。

```

$ wheels generate proto user
proto/user.proto

syntax = "proto3";

package proto;

```

```

import "google/api/annotations.proto";

service UserService {
    rpc GetUser ( GetUserRequest ) returns ( User ) {
        option ( google.api.http ) = { get: "/v1/users/{id}" };
    }

    rpc CreateUser ( CreateUserRequest ) returns ( User ) {
        option ( google.api.http ) = {
            post: "/v1/users"
            body: "user"
        };
    }
}

message User {
    int64 id = 1;
    string name = 2;
}

message GetUserRequest {
    int64 id = 1;
}

message CreateUserRequest {
    User user = 1;
}

```

生成服务骨架

```
$ wheels generate service user --proto user
```

编辑自动生成的 app/services/user.go 来实现这个服务。

app/services/user.go

```
package services
```

```

import (
    "golang.org/x/net/context"
    "mysrv/proto"
)

func (s *Server) GetUser(ctx context.Context, req *proto.GetUserRequest) (*proto.User, error) {
    return &proto.User{Id: req.Id}, nil
}

func (s *Server) CreateUser(ctx context.Context, req *proto.CreateUserRequest) (*proto.User, error) {
    return &proto.User{Name: req.User.Name}, nil
}

```

运行服务

我们可以通过 gRPC 和 HTTP 访问这个服务，如使用 HTTP:

```

$ curl http://localhost:8080/v1/users/2
{"id":"2"}

$ curl -d "{\"name\": \"wheels\" }" http://localhost:8080/v1/users
{"name": "wheels"}

```

```

# Wrong URL
$ curl http://localhost:8080/v1/user/2
Not Found

```

生成 Swagger 文档

我们还可以利用 Wheels 的命令行工具，从 Protobuf 服务描述文件生成 Swagger 文档，用于服务的 API 参考文档或者发布给客户的文档。

```
$ wheels generate swagger --proto hello
proto/hello.swagger.json
```

```
{  
  "swagger": "2.0",  
  "info": {  
    "title": "proto/hello.proto",  
    "version": "version not set"  
  },  
  "schemes": [  
    "http",  
    "https"  
  ],  
  "consumes": [  
    "application/json"  
  ],  
  "produces": [  
    "application/json"  
  ],  
  "paths": {  
    "/v1/hello/{name)": {  
      "get": {  
        "operationId": "SayHello",  
        "responses": {  
          "200": {  
            "description": "",  
            "schema": {  
              "$ref": "#/definitions/protoHelloReply"  
            }  
          }  
        }  
      },  
      "parameters": [  
        {  
          "name": "name",  
          "in": "path",  
          "required": true,  
          "type": "string"  
        }  
      ]  
    }  
  }  
}
```

```
        }
    ],
    "tags": [
        "HelloService"
    ]
}
}

},
"definitions": {
    "protoHelloReply": {
        "type": "object",
        "properties": {
            "msg": {
                "type": "string"
            }
        }
    },
    "protoHelloRequest": {
        "type": "object",
        "properties": {
            "name": {
                "type": "string"
            }
        }
    }
}
}
```

深入理解 Go 语言对象模型 (Go Object Model Inside Out)

前言

无论您是程序语言的使用者、爱好者，或者是框架 (Framework) 作者，深入理解语言的对象模型 (Object Model) 都将极大地帮助您理解底层细节，写出更稳定更高效的代码。

不少编程语言都有此一领域的著作和文章。对于 C++，我们有 Lippman 的《Inside the C++ Object Model》；对于 Ruby，我们有《Metaprogramming Ruby》；对于 JavaScript，也有一系列的书籍和文章阐释 Object 和 Function 的内部机制。

本篇文章，我们也试图深入分析 Go 语言的对象模型，希望能借此讨论，帮助您更得心应手地驾驭 Go 语言的对象特性。这篇小文当然不能与我们之前提到的、其他语言领域的皇皇巨著相提并论，但如果能借此引起您的思考甚至反馈，那也算是抛砖引玉，足令我们感到欣慰了。

正文

既然我们提到了对象，那么首先问一个有趣的问题：Go 是一门面向对象的语言吗？[官网 FAQ](#) 的答案是：

Yes and no.

Go 有类型 (Type) 和方法 (Method)，包括接口 (Interface)，可以让你使用面向对象的风格进行编程；但 Go 又缺少很多常见 OO 语言的特性，如继承、多态等等。

为什么 Go 会选择这种方式呢？以我的理解，无论是从语言的特性还是编译器实现的角度，简单性都是 Go 语言的核心设计原则。也许对 Go 语言的作者来讲，这种设计风格既能保留普通 OO 语言的核心优点，又能摈弃继承及复杂类型系统带来的额外负担。

好的，接下来我们将从结构体 (Struct)，函数 (Function)，方法 (Method)，接口 (Interface)，反射 (Reflect) 等几个方面深入分析 Go 语言的对象模型。

首先是结构体 (Struct)。

结构体 (Struct)

结构体与数值、字符串、数组等等类型一样，属于 Go 语言的基本类型。

结构体在 Go 语言中扮演着非常重要的角色。由于 Go 语言中没有常见 OO 语言中的类 (Class)，结构体实际承担了类的一部分职责。与常见 OO 语言类似，结构体的实例可以作为数据容器 (Data Container，想象下 OO 语言中常见的通过 Getter/Setter 对数据的访问)，或是方法接收者 (Method Receiver，后面 Method 一节中会详细说明)。结构体声明方式如下：

```
type Person struct {
    Name string
    Age  int
}
```

我们前面提到过，Go 没有类，也没有继承，那么 Go 是如何如常见 OO 语言般通过继承来实现代码复用的呢？答案是通过组合（Composition）。

```
type Employee struct {
    Person
    Job string
}
```

结构体 Employee 内嵌（Embedded）了一个结构体 Person。这样，Employee 的实例能访问到 Person 上的所有字段（Field）和方法（Method，我们会在后面的章节中详述），一定程度上实现了类似继承的效果。

```
e := &Employee{}
fmt.Println(e.Person.Name, e.Person.Age, e.Job)
```

注意到我们可以通过 e.Person.Name 访问到内嵌结构体的字段。如果仅仅这样，那仍然不是特别方便；而实际上 Go 编译器提供了隐式查找的功能，甚至使我们能通过 e.Name 访问到内嵌结构体的字段。

```
e := &Employee{}
fmt.Println(e.Name, e.Age, e.Job)
```

这样是不是有点 OO 语言继承的感觉了？

隐式查找也支持多级嵌套：

```
type Manager struct {
    Employee
    Title string
}
m := &Manager{}
fmt.Println(m.Name, m.Age, m.Job, m.Title)
```

通过以上几例，我们可以看到，Go 语言可以通过结构体的组合来实现类似 OO 继承的概念。但这与真正的继承还是有一定区别，比如 Employee 的实例并不是（is-a）Person 的实例，因此更无法实现 OO 中的多态（Polymorphism）。

我们定义一个接收 Person 作为参数的函数来进行说明：

```
func playWith(p *Person) {}
```

然后初始化一个 Employee 的实例传入：

```
e := &Employee{}
```

```
playWith(e)
```

报错， cannot use type *Employee as type *Person in argument

所以我们无法通过结构体组合来实现 OO 继承和多态的全部功能。但在后面接口 (Interface) 的章节中会看到，可以通过接口来实现对象间 is-a 的关系。

虽然结构体组合相比继承在某些方面有一定的局限性，但由于我们可以内嵌多个结构体，因此得以方便地实现类似多重继承般的代码复用，规避了如多重继承在对象层级上的复杂性。

```
type Address struct {
    Number string
    City   string
}
```

```
type Contact struct {
    Person
    Address
}
c := &Contact{}
fmt.Println(c.Name, c.City)
```

很多设计模式的书籍都鼓吹组合优于继承 (Composition Over Inheritance)，我想 Go 的作者也是赞同这一看法的，最后以一则小趣闻作为本章的结束：

I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: “If you could do Java

over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.

函数 (Function)

接下来我们讨论函数。

函数同样是 Go 语言的基本类型。编译器视相同签名（参数和返回值列表）的函数为同一类型。另外值得注意的是，函数是 Go 语言的第一类公民 ([First-class citizen](#))。

```
// 声明了一个函数命名类型
type operator func(int, int) int

// 返回值为函数，可以确认函数确实是第一类公民
func getOperator(op string) operator {
    switch op {
    case "+":
        // 1. 返回一个匿名函数
        // 2. 函数签名一致视为同类型
        return func(x, y int) int {
            return x + y
        }
    // 略 ...
    }
}
```

调用方式如下：

```
// 函数是第一类公民，所以可以通过变量来引用，然后调用
op := getOperator("+")
op(1, 2)
```

```
// 当然也可以直接调用
```

```
getOperator("*")(1, 2)
```

Go 语言的函数有一些不太方便的限制，如：

- 不支持同名函数重载 (Overload)
- 不支持默认参数
- 不支持模式匹配 (Pattern Match, 函数式编程语言常见概念)

```
// 如果我们希望扩展上面的函数，定义一个带初始值的版本
```

```
// 因为Go不支持同名函数重载，也不支持默认参数
```

```
// 所以一种可能的解决方案是再重新定义一个函数
```

```
func getOperatorWithValue(op string, v int) operator {
    switch op {
    case "+":
        return func(x, y int) int {
            return v + x + y
        }
    // 略 ...
    }
}
```

调用方式如下：

```
getOperatorWithValue("+", 1)(2, 3)
```

但这种解决方案很不优雅。尤其是当参数变化较多时，如果需要重新定义大量的函数尤其显得繁杂。我们接下来会看到更优雅的解决方案。

前面我们说了 Go 语言函数的一些限制，但它也有自己的特点和优势，如：

- 支持可变长参数
- 支持多返回值，支持命名返回值
- 支持匿名函数和闭包 (Closure)

```

// 可变长参数函数
type operator func(...int) int
// 多返回值，命名返回值。Go的惯例是多返回值中返回error
func getOperatorWithValue(op string, v int) (fn operator, err error) {
    switch op {
    case "+":
        // 匿名函数赋值给变量，然后作为返回值
        fn := func(args ...int) int {
            sum := v
            for _, a := range args {
                sum += a
            }
            return sum
        }
        return fn, nil
    // 略 ...
    default:
        return nil, errors.New("不支持的操作符")
    }
}

```

调用方式如下：

```

if op, err := getOperatorWithValue("+", 1); err == nil {
    op(2, 3, 4)
}

```

再来说明下闭包 ([Closure](#)) 的使用：

```

// 稍微修改了之前函数的逻辑，用来说明闭包的概念
func getOperatorWithValue(op string, v int) (fn operator, err error) {
    switch op {
    case "+":
        // 返回的函数修改了上层的变量v
        // 这相当于延长了v的生命周期

```

```

// 因此fn是一个闭包
fn := func(args ...int) int {
    for _, a := range args {
        v += a
    }
    return v
}
return fn, nil
// 略 ...
default:
    return nil, errors.New("不支持的操作符")
}
}

```

调用如下：

```

if op, err := getOperatorWithValue("+", 1); err == nil {
    op(2, 3) // 返回6
    op(2, 3) // 返回11
}

```

最后，让我们来看一下如何通过 Go 特有的设计模式 (Design Pattern) 来解决语法本身的限制。还是以重载为例。

一种方式是通过函数匹配参数列表，效果如下：

```

func Brew(shots int, variety string, cups int) []*Coffee {
    // Brew my coffee
}

func ALargeCoffee() (int, string, int) {
    return 3, "Robusta", 1
}

func ForTheOffice(cups int) (int, string, int) {
    return 1, "Arabica", cups
}

```

```

func AnEspresso(shots int) (int, string, int) {
    return shots, "Arabica", 1
}

func main() {
    myCoffee := Brew(ALargeCoffee())
    coffeesForEverybody := Brew(ForTheOffice(6))
    wakeUpJuice := Brew(AnEspresso(3))
}

```

或者

```

func main() {
    t := FillTemplate(FromReader(myReader, tokens))
    t = FillTemplate(FromFile(filename, tokens))
    t = FillTemplate(FromURLWithJSON(templateURL, restServiceURL))
}

```

// FillTemplate will accept a template and a slice of name-values
and

// replace the named tokens with the given values and return the
result.

```

func FillTemplate(template string, tokens map[string]interface{})  
string {
}

```

// 略 ...

另一种方式是通过 [Functional Options](#) 模式，效果如下：

```

emptyFile, err := file.New("/tmp/empty.txt")
if err != nil {
    panic(err)
}

fillerFile, err := file.New("/tmp/file.txt", file.UID(1000), file.
Contents("Lorem Ipsum Dolor Amet"))

```

```

if err != nil {
    panic(err)
}

```

重载 (Overload) 等特性的缺失，究其原因，还是如我们在前面提到的设计原则，官网 [FAQ](#) 中也对这个问题进行了回答。通过上面的例子我们可以看到，这些问题一定程度上通过一些模式来进行规避。类似的例子还有 [Decorator](#) 等等。

方法 (Method)

方法可以看作特殊的函数。与普通函数不同，方法需要与对象实例绑定，在定义语法上方法有前置的接收者 (Receiver)。

可以为除接口和指针外的任何类型定义方法。如：

```

type N int

func (n N) double() int {
    return n*2
}

func main() {
    var a N = 5
    println(a.double())
}

```

方法同样不支持重载。方法 Receiver 的类型可以是基础类型或指针类型。这会决定方法调用时对象实例是否被复制。

Go 语言中虽然没有类 (Class)，但它的方法巧妙地达到了与 OO 语言中方法类似的使用体验和效果。类似于我们在第一部分 Struct 中介绍过的，甚至可以通过在 Struct 上定义方法来达到面向对象语言中继承的效果。

我们扩展第一部分的例子：

```
type Person struct {
```

```

Name string
Age int
}

func (p *Person) Talk() {
    ...
}

type Employee struct {
    Person
    Job string
}

func (e *Employee) Work() {
    ...
}

```

接下来，我们可以通过：

```
e := &Employee{}
e.Talk()
```

来调用被嵌入的 Struct 上的方法，这也达到了类似于 OO 中方法继承的效果。但需要注意的是，如同我们在 Struct 部分提到的，这样的方法“继承”是无法实现 OO 中的多态的。

最后再来介绍一个比较有意思特性。如同我们在前面提到过的，方法可以定义在几乎任何的类型上。利用这个特性，我们把方法定义在函数上：

```

type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}

```

这样，任一满足了 `func(ResponseWriter, *Request)` 签名的函

数，我都可以把它转化为 HandlerFunc 类型的对象，然后再调用它的 ServeHTTP 方法：

```
func doSomethingWithHTTP(w ResponseWriter, r *Request) {
    ...
}
```

这样有什么好处呢？这样可以使传入的函数经过类型转化后满足方法参数中的接口，正是我们要在下一部分中介绍的接口：

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

接口 (Interface)

Go 语言中接口的概念与常见 OO 语言中接口类似。但在实现角度存在着比较显著的区别。

Go 语言中类型实现接口无须显式声明，即只要目标类型方法集中包含接口声明的全部方法，就会被编译器和运行时认为实现了该接口。这种实现机制是比较简洁的，非侵入式的设计也带来了很多便利性。

```
type Talker interface {
    Talk()
}
```

因为我们之前定义的 Person 实现了 Talk 方法，所以它也就实现了 Talker 接口。

```
var t Talker = &Person{}
t.Talk()
```

接下来就更有意思了。因为 Employee “继承” 了 Person，所以 Employee 也实现了 Talker 接口。

```
var t Talker = &Employee{}
t.Talk()
```

虽然我们在前面章节中提到过 Go 语言的“继承”是无法实现多态的，但聪明的读者马上能联想到，我们完全可以利用接口来巧妙地实现

Dynamic Dispatch! 这确实是非常有意思特性！

```
func justTalk(t Talker) {
    t.Talk()
}

func (e *Employee) Talk() {
    ...
}

p := &Person{}
e := &Employee{}

justTalk(p)
justTalk(e)
```

最后再说明一下，如果接口没有任何方法声明，那么它就是一个空接口 interface{}, 类似于 OO 中的根类型 Object，可被赋值为任何类型的对象。如果使用空接口作为方法参数，该参数可以接受任何值。使用空接口作为参数一定要谨慎——这样固然灵活，但失去了类型安全和类型检查等一系列的好处。

反射 (Reflect)

反射可以让我们在运行时获取对象的类型信息，这从一定程度上弥补了静态语言相对动态语言在灵活性上的缺失。此外，反射也是实现元编程 (Metaprogramming) 的重要手段。

和 C 数据结构一样，Go 对象头部没有类型指针，所以无法通过自身获取任何类型相关信息。Go 也不像 Java、C# 那样有元数据，更不像动态语言一样可以在运行时获取类型、方法等各种信息。

Go 的反射操作所需的全部信息都来自于接口变量。接口变量除存储自身类型外，还会保存实际对象的类型数据。总体而言，Go 的反射具有的功能是较为有限的。

```
func TypeOf(i interface{}) Type
func Valueof (i interface{}) Value
```

在使用反射时，尤其需要注意 Type 和 Kind 的区别。Type 表示对象的类型，Kind 表示对象类型所对应的底层类型。如：

```
type X int

func main() {
    var a X = 1
    t := reflect.TypeOf(a)

    fmt.Println(t.Name(), t.Kind())
}
```

则对应的输出应为 X int

接下来看一个反射实际使用的范例：

```
package main

import (
    "fmt"
    "reflect"
)

type Foo struct {
    FirstName string `tag_name:"tag 1"`
    LastName  string `tag_name:"tag 2"`
    Age        int    `tag_name:"tag 3"`
}

func (f *Foo) reflect() {
    val := reflect.ValueOf(f).Elem()

    for i := 0; i < val.NumField(); i++ {
        valueField := val.Field(i)
        typeField := val.Type().Field(i)
```

```

tag := typeField.Tag

    fmt.Printf("Field Name: %s,\t Field Value: %v,\t Tag Value:
%s\n", typeField.Name, valueField.Interface(), tag.Get("tag_name"))
}

}

func main() {
    f := &Foo{
        FirstName: "Drew",
        LastName:  "Olson",
        Age:       30,
    }

    f.reflect()
}

```

关于反射使用的原则，我们还可以参考 [The Laws of Reflection](#):

- Reflection goes from interface value to reflection object.
- Reflection goes from reflection object to interface value.
- To modify a reflection object, the value must be settable.

总结

本文从结构体、函数、方法、接口、反射等几个方面深入讨论了 Go 语言的对象模型。这篇文章的讨论主要是从语法 (Syntax) 的层面上展开的。我们期待着下次有机会从编译器和运行时的角度来更深入的探讨这个问题。

参考资料：

- <https://golang.org/doc/faq#types>
- https://golang.org/doc/effective_go.html
- <http://spf13.com/post/is-go-object-oriented>

- <https://www.goinggo.net/2014/05/methods-interfaces-and-embedded-types.html>
- <https://github.com/tmrts/go-patterns>
- <http://blog.ralch.com/tutorial/golang-reflection/>
- <https://blog.golang.org/laws-of-reflection>

说一说 Go 语言并发模型中的 Channel

Go 语言的并发模型以其简练而独特的设计、对开发者的简洁易用以及高性能而区别于其他的传统开发语言，如 C++、Java 等。Go 语言提供了围绕着 Goroutine、Channel 为核心概念的并发特性，正确地理解和使用它们，不仅可以避免程序出现一些并发或同步方面难以预知的运行问题，也有助于开发者更优雅地设计出利用 Go 并发模式的模块及系统。

本文主要针对 Go 并发模型下的 Channel 的使用，结合实际开发中的体会做一下总结，希望借此帮助接触 Go 语言不久的开发者们，正确掌握 Channel 这一重要并发特性的打开方式，并在实际项目中自如地使用 Go 语言的并发特性，开发出正确而高效的程序。

Channel 概述

开始学习 Go 的同学们一定都知道 Channel 了，这可以说是 Go 语言提供的最亮眼的一个语言特性了。顾名思义，Channel 即通道，是 Go 语言

中用来在不同 Goroutine 之间进行通讯的管道。Go 语言中的各种类型，从原生类型 Int、Int64、String、Map 等，到 Struct、Interface、Pointer，都可以用 Channel 来在各个 Gouroutine 之间传递。

Channel 的声明和初始化非常简单，同时需要指定 Channel 中所传递的数据的类型：

```
var ch1 chan interface{} //声明一个传递类型为interface{}的channel
ch2 := make(chan int)    //创建一个传递的数据类型为int的unbuffered
channel
ch3 := make(chan string, 5) //创建一个传递string类型的buffered
channel,buffer的容量为5
```

这里要注意的是，已声明但是没有初始化的 Channel，它的值是 nil，nil 的 Channel 无论发送和读取都会阻塞。后面的章节会列出几种使用 Channel 时的坑，并一一说明。

往 Channel 中写入及读取：

```
ch1 <- x //往Channel中写入值
y = <- ch2 //从channel中取出值
-< ch3 //取值并丢弃
close(ch3) //关闭channel
```

以上是 Channel 的定义和基本使用方式，Golang 官方以及中文社区都有大量的资料说明。在这里要展开深入讨论的是 Channel 的正确打开方式。因为仅仅了解了 Channel 怎样定义，怎样写入和读取，仍然不足以让你写出正确使用 Channel 的 Go 语言程序。下面一部分，我们会从 Channel 的使用细节和实例，来看看它的正确打开方式是怎样的，在实际使用中有哪些坑需要注意。

深入理解 Channel 中的 Buffer 与阻塞

首先举个例子来描述 Go 语言中不同 Goroutine 的工作方式。比如一个进程中的两个 Goroutine，简单来说就像一个汽车工厂里的两个生产车间，同时在进行汽车部件的生产。每个车间在分到 CPU 时间片的时候，

就可以将自己的生产过程继续下去。这两个车间之间是有墙分隔开的，而 Channel 就是两个车间之间传递工具或者零件的一个小窗口。两个车间互相不可见，生产过程也互相不干扰，只通过小窗口来互相传递东西。在 Goroutine 中可以用 Channel 来传递消息、同步状态。

本章概述中提到过，Channel 可以定义为有 Buffer 的 Channel，或者无 Buffer 的 Channel。这其中可是大有玄机，因为 Channel 会阻塞，而有 Buffer 和无 Buffer 的 Channel 阻塞的条件和行为也大不一样。

无 Buffer 的 Channel

无 Buffer 的 Channel 意味着没有缓冲，只有读写双方都准备好的时候，写操作和读操作才能完成，否则就一定会阻塞。实际上是什么意思呢？还是以上面的例子来说明，车间 A 和车间 B 之间有一个无 Buffer 的 Channel 叫 ab，车间 A 生产过程中需要通过 Channel ab 往车间 B 传递一个零件。这时候车间 A 的工人把一个零件放到小窗 ab 这里，但是他不能放手，因为 ab 没有 Buffer，只有等对面 B 车间有一个人来接他手里的零件时，他才能松开手，回去继续车间 A 的生产过程。这就是无 Buffer 的 Channel 阻塞时的状态，写操作时如果该 Channel 没有被读取，那么写操作将一直被阻塞，直到读一方来读取它。

而对于读操作，同样，如果车间 B 的工人需要从 Channel ab 这个小窗取一个零件，取到了才能继续他的生产过程，那么他只能在窗口一直伸手等待，直到另一方来到小窗 ab 前，把他要取的零件放到他手上，这个取东西的操作才能结束。在此之前，B 车间中的生产过程是必须等待的。

有 Buffer 的 Channel

有 Buffer 的 Channel 就有了缓冲区，不像无 Buffer 的 Channel，需要读写双方都准备好才能完成读写操作，在有 Buffer 的 Channel 上的读写，只要 Buffer 没满就不会阻塞。写的 Goroutine 可以放下就走，读的

Goroutine 可以取了就走。再以工厂车间为例，有 Buffer 的 Channel 相当于在两个车间通讯的小窗窗台上放了个盒子，Buffer 的大小就是盒子的容量。例如 C 车间和 D 车间，相互之间通过 Channel cd 通讯，假如定义 cd 时我们使用的 Buffer 大小是 10，那么意味着 Channel cd 这个小窗里有个大小为 10 的盒子，C 车间的工人到小窗前给 D 车间传递零件时，无须等 D 车间的人来取，只要往盒子里放就好了，当然前提是盒子里有空格。如果盒子满了，那么 C 也必须等到 D 至少取走一个零件，盒子里空出一个格的时候才能往里放。

简言之，对于带 Buffer 的 Channel 的写入方，只有当 Buffer 满了的时候才会阻塞写入行为。对于读取方，在 Buffer 为空时会阻塞（和无 Buffer 时行为相同），直到它能读到值才会往下走。

Channel 的数据读取

在前面讲到 Channel 定义时我们介绍过从 Channel 中读取数据的最简单方式。其实在实际程序中，读取 Channel 往往是用于监听一些状态的变化和感知从其他 Goroutine 来的数据传递，既然使用了 Goroutine 并发地去运行程序，我们自然是希望各个 Goroutine 不受阻塞地做各自的处理，只有在监听的 Channel 里有当前 Goroutine 关心的数据到达时，再做相应的处理。所以 Channel 的读取往往就不是简单地使用了，需要做一些包装和改进来更好地利用其在 Goroutine 之间同步的能力。

从 Channel 中读取数据的方式

直接读取

前面总结过，直接读取即用 Channel 的读取操作符 `<-` 直接从 Channel 中读值，例如：

```
<- inbox
```

当 Channel 无 Buffer，或 Buffer 中没有数据时，程序会在此处阻

塞，直到有数据可读。在实际的 Go 程序中，还是有几种常用的模式来从 Channel 中读取数据的。对于一个同步执行的 Goroutine 来说，从 Channel 读取的场景包括程序等待同步状态、监听通知、获取数据等等。

用 select 读取

最常见的监听某个 Channel 的方式莫过于另起一个 Goroutine，在当中用 for 循环做 select 操作，循环地从 Channel 中读取数据，例如：

```
func main() {
    ch1 := make(chan int)

    go func() {
        for {
            select {
            case val := <-ch1:
                fmt.Println("Received value: ", val)
            }
        }
    }()

    for i := 0; i <= 10; i++ {
        ch1 <- i * 2
    }
    fmt.Println("Process done.")
}
```

用 for 循环读取

用 for 循环读取，无 Buffer 的 Channel 行为与 select 方式一致，如下所示。读取无 Buffer 的 Channel：

```
func main() {
    ch1 := make(chan int)

    go func() {
        fmt.Println("-- start read")
        for r := range ch1 {
```

```

        fmt.Println("Received value: ", r)
    }
}

for i := 0; i <= 10; i++ {
    ch1 <- i * 2
}

fmt.Println("Process done.")

}

```

读取有 Buffer 的 Channel

```

func main() {
    ch1 := make(chan int, 20)
    for i := 0; i <= 10; i++ {
        ch1 <- i * 2
    }
    go func() {
        l := len(ch1)
        fmt.Println("-- start read, the length of channel is: ",
l)
        for r := range ch1 {
            fmt.Printf("Received value: %d, the length is:
%d\n", r, len(ch1))
        }
        fmt.Println("Drain channel out done.")
    }()
    time.Sleep(5*100*time.Millisecond)
    fmt.Println("Process done.")

}

```

区别在于，有 Buffer 的 Channel 在 Buffer 未满之前写入是不会阻塞的，所以这段示例代码可以先往 Channel 中写入 10 个值，写入操作的完成无需等待读取端准备好。

特殊状态下的 Channel 数据读写

以上关于 Channel 的数据写入和读取的说明及示例都是基于正常状态（已初始化，未被关闭）下的 Channel。如果一个 Channel 被关闭，或者 Channel 为零值 nil（未被初始化，被赋值为 nil），其读写的行为都有所不同。说到这里，我们总结一下 Channel 在几种特殊状态下读写数据的情况。这部分内容对于用好 Channel 这一武器，避免程序出现非预期的状态非常重要，也是 Dave Cheney 在他的博客里总结过的。

关闭 Channel 即：

`close(ch)`

但 Channel 的关闭是不应当随意进行的，因为关闭 Channel 后的读和写的行为都需要我们深刻理解，才能选择在程序适当的地方去做 Channel 的关闭操作。

首先，已关闭的 Channel 永远不会阻塞。很好，不阻塞。至少在错误状态出现的时候程序不会 Block，错误比较容易被发现。上面刚刚说过 Channel 的关闭，那么对于已关闭的 Channel 的读写操作会是什么样的状况，又如何避免错误使用带来的非预期结果呢？

写入已关闭的 Channel

注意，向已关闭的 Channel 写入会 panic。例如这段代码，在关闭 Channel 后再往里写入值

```
ch1 := make(chan string, 1)
go func() {
    v := <- ch1
    fmt.Println("Received from ch1: ", v)
}()
close(ch1)
ch1 <- "Hi, nice to meet you"
fmt.Println("Done")
```

会输出

```
panic: send on closed channel
try
```

从已关闭的 Channel 读取

Channel 关闭后从其中读取会是什么样的行为？一个好消息是，Channel 关闭后，读取操作就再也不会 Block 了。是不是感觉松一口气？对于无 Buffer 的 Channel，关闭后再读取，不会 Block，会读取到该 Channel 类型的零值。具体来讲，就是如果这个 Channel 是 String 类型的，就会读到 "", 如果 Channel 是 interface{} 类型，将会读到空指针 nil，以此类推。

对于 Buffered Channel，关闭后读取操作依然可以把其中的数据读取出来，直到这个 Channel 被取空。当所有值都读完后，继续读该 Channel 会得到该 Channel 的类型的零值数据。

所以在这里我们可以很清楚地推断出，如果用 select 无限循环地读取，那么 Channel 被关闭后就会死循环了，永远能读到值，但每次取到的都是该 Channel 类型的零值。而如果用 for... range 方式来读取的话，循环就会在 Channel 中的数据被全部取出后结束。还有一种方式可以用来判断已关闭的 Channel 是否已经被取空，就是读取 Channel 时使用读取操作的第二个返回值，如下所示

```
x, ok := <-ch
```

当 Channel 被关闭并且为空的时候，ok 为 false。下面看看两种情况的例子。

用 select 读取

用常用的 select 方式读取已关闭的 Channel 会是怎样的行为？如上所述，Channel 被关闭后不会 Block，会永远返回零值，所以如果在代码里不做判断的话将进入无限循环状态。还是用上面那段代码做例子

```

func main() {
    ch1 := make(chan int)
    go func() {
        for {
            select {
            case val := <-ch1:
                fmt.Println("Received value: ", val)
            }
        }
    }()
    for i := 0; i <= 10; i++ {
        ch1 <- i * 2
    }
    fmt.Println("Process done.")
}

```

没问题，正常输出了 0-10 的 2 倍

```

Received value:  0
Received value:  2
Received value:  4
Received value:  6
Received value:  8
Received value:  10
Received value:  12
Received value:  14
Received value:  16
Received value:  18
Received value:  20
Process done.

```

如果在发送完毕后关闭 Channel，循环读取将进入无限循环状态

```

close(ch1)
fmt.Println("Process done.")

```

输出将变为：

```
.... .
Received value: 0
.... .
```

死循环，无论有 Buffer 还是无 Buffer 的 Channel 都一样。所以如果用 select 无限循环方式作为 Channel 接收的方式，可以利用读 Channel 的第二个返回值加一个判断来处理 Channel 关闭并被读取完的行为

```
func main() {
    ch1 := make(chan int)
    go func() {
        var closed bool
        for {
            if closed {
                break
            }
            select {
            case val, ok := <-ch1:
                fmt.Println("Received value: ", val)
                if !ok {
                    closed = true
                    break
                }
            }
        }
        fmt.Println("Channel is closed and drain out.")
    }()
    for i := 0; i <= 10; i++ {
```

```

    ch1 <- i * 2
}
close(ch1)
fmt.Println("Process done.")
}

```

这时候的输出是

```

Received value:  0
Received value:  2
Received value:  4
Received value:  6
Received value:  8
Received value:  10
Received value:  12
Received value:  14
Received value:  16
Received value:  18
Received value:  20
Process done.
Received value:  0
Channel is closed and drain out.

```

使用有 Buffer 的 Channel 时，行为也是类似的。

用 for 循环读取

在 Channel 被关闭后，for...range 方式就比较适合用来 Drain out Channel 中的所有数据了，下面这段代码在发送完毕数据后关闭 Channel，而读取方的 for...range 循环会在接收完所有数据之后退出循环。

```

func main() {
    ch1 := make(chan int, 20)
    for i := 0; i <= 10; i++ {
        ch1 <- i * 2
    }
    close(ch1)
}

```

```

go func() {
    l := len(ch1)
    fmt.Println("-- start read, the length of channel is: ",
l)
    for r := range ch1 {
        fmt.Printf("Received value: %d, the length is:
%d\n", r, len(ch1))
    }
    fmt.Println("Drain channel out done.")
}()

fmt.Println("Process done.")
}

```

输出为：

```

Process done.
-- start read, the length of channel is: 11
Received value: 0, the length is: 10
Received value: 2, the length is: 9
Received value: 4, the length is: 8
Received value: 6, the length is: 7
Received value: 8, the length is: 6
Received value: 10, the length is: 5
Received value: 12, the length is: 4
Received value: 14, the length is: 3
Received value: 16, the length is: 2
Received value: 18, the length is: 1
Received value: 20, the length is: 0
Drain channel out done.

```

基于 Channel 关闭后的行为，如何关闭它才是安全的呢？有一条比较通用的适用原则，即不要从接收端关闭 Channel。如果发送端有多个并发发送者，也不要从其中一个发送端去关闭 Channel。换句话说，如果发送者是唯一的 Sender，或者发送者是 Channel 最后一个活跃的 Sender，那

么应该在发送者的 Goroutine 关闭 Channel，通过这个关闭行为来通知所有的接收者们，已经没有值可以读了。只要在使用 Channel 时遵循这条原则，就可以保证永远不会发生向一个已经关闭的 Channel 写入值，或者关闭一个已经关闭的 Channel 这样的情况。

写入值为 nil 的 Channel

当一个 Channel 的值为 nil 时，写入这个 Channel 的操作会永远阻塞。例如下面这段代码：

```
func main() {
    var c chan string
    c <- "hello, David" // block forever
}
```

从值为 nil 的 Channel 读取

同样的，从一个 nil 的 Channel 读取，也会永远阻塞：

```
func main() {
    var c chan string
    str := <- c // block forever
}
```

一个声明后但尚未通过 make 语句初始化的 Channel 的默认值就是其零值，也就是 nil。所以在使用 Channel 时一定要注意这一点，在对其写入或者读取时一定要保证这个 Channel 已经被初始化了。阻塞是一种可怕的安静，没有任何错误，没有 panic，进程还活着，但是当前 Goroutine 陷入了无尽的等待中，无法继续往下走，无法感知到。

从 Channel 的原理上来看，如何解释这种阻塞行为？Dave Cheney 很好地给出了他的解释，我来简单描述一下：

首先，Channel 的类型定义并不包括 Buffer，所以 Buffer 是 Channel 本身的值的一部分，Channel 的声明只会声明其类型。Channel 只是被声明而没有被初始化的时候，它的 Buffer size 是 0。这种情况下相当于

一个无 Buffer 的 Channel 了，只有在发送和接收双方都准备好的情况下，无 Buffer 的 Channel 才不会阻塞。而对于未初始化的 Channel，其值是 nil，发送和接收双方都无法获取到对方的引用，无法和对方通讯，双方都被阻塞在这个未初始化的 Channel 里。

小结

本文结合日常工作中的开发实践，将 Channel 的正确使用方式和行为做了一个简单归纳，希望能有助于各位进入 Go 语言世界不久的开发者们少踩一些坑，更加高效而合理地使用 Channel 这一特性。

参考资料

- <https://dave.cheney.net/>
- <http://studygolang.com/articles/9518>
- <https://blog.golang.org/>

ETCD 背后分布式存储实现

ETCD 是 Go 生态圈中的分布式 K/V 实现，它背后是第一个生产级别的 Raft 协议实现，一些分布式系统中，如 TiKV(Rust 实现) 等，也借鉴了它的 Raft 协议实现。

最初 ETCD 的定位仅仅是一个高可用的配置管理服务，在 *NIX 系统中 “/etc” 目录用以存储单机系统配置，而一个分布式 (Distributed) 的配置管理服务 (/etc) 就是 ETCD。随着越来越广泛的应用，ETCD V3 已经演变为一个通用 K/V 系统。

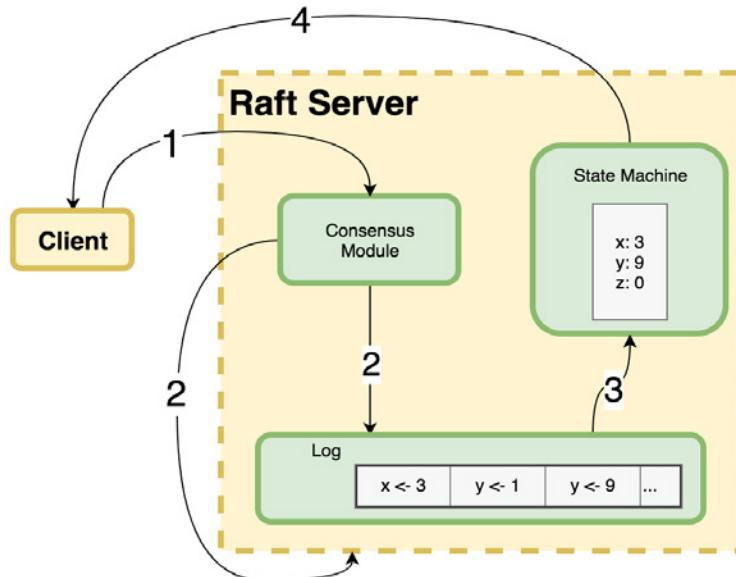
结合 ETCD 历史版本中存储系统的优化，本文系统介绍了 ETCD 中的存储系统实现。弄清楚 ETCD 背后分布式存储的实现不仅对故障排除、日常运维助益颇多，也可以为分布式存储的实现提供思路。

Raft 模块原理简介

Raft 模块承担了集群管理和指令决策、执行，以及相应的存储管理层，

即 WAL(Write-Ahead-Log) 和快照 (Snapshot)。Fig 1.1 中从单个节点的角度看 Raft 模块的原理。

Fig 1.1 简化的 Raft 协议模型



Raft 模块可以从功能上分成三部分：

- 共识模块 (Consensus Module): 接收和处理指令
- Log: 存储历史指令，和当前状态机的状态 (Commit到哪一条等)
- State Machine: 集群最新状态，ETCD里面主要就是数据库

客户端向集群发起一次写操作的数据流大致涉及 4 个步骤：

1. 客户端发起一个需要和集群一起决策的指令，如发起一个写请求
2. 将指令持久化到WAL(Write-Ahead-Log)，共识模块会更新Log状态 (如Committed等信息)
3. 当指令在集群中达成了一致 (Committed)，就需要在状态机中执行 (Apply) 这条指令，状态机主要就是一个内存中的数据库
4. 节点向客户端返回更新操作的结果

Raft 协议层采用消息模型实现，无论是对配置（集群的节点）还是

数据进行的修改，最终都会传入 Raft 模块与整个集群一起决策，决策过程就是一些 gRPC 封装的请求、响应的过程，每个请求或响应都可能改变 Raft 状态机的状态，后文中会详细介绍。

Raft 协议实现

ETCD 中每个节点都是一个有限状态机，本节从节点启动时的入口函数开始，逐步梳理了整个 Raft 协议层的框架和数据流（基于 V3.1.8 源码）。

首先，在没有任何初始化数据文件时，节点启动时需要创建数据文件目录并与集群中其他的节点建立连接。此处不考虑通过服务发现来注册和配置集群节点，只考察用“`--initial-cluster`”静态配置节点列表的方式，本质上没有区别。

Raft 协议的基础是消息和日志的管理，每条日志都对应至少四个状态：

- `Unstable`: 消息已经成功收到，但还没有持久化到 WAL
- `Stable`: 这是一个隐含的状态，即日志已经持久化到当前节点的 WAL 中，但还没有和集群达成一致
- `Committed`: 消息已经跟整个集群协商完毕，达成一致
- `Applied`: 整个集群已经协商并达成一致的指令在当前节点执行完毕

简而言之，每个进入 Raft 状态机的消息首先会被缓存在 `Unstable Storage`，这是一个内存中的数组，然后异步收集并持久化到 WAL 日志文件，日志提交后更新 WAL 文件的 Metadata，最后提交完毕的消息会被批量在当前节点执行。Fig 2.1 为 ETCD 节点初始化的总入口。

Fig 2.1 节点初始化

```

243 // NewServer creates a new EtcdServer from the supplied configuration. The
244 // configuration is considered static for the lifetime of the EtcdServer.
245 func NewServer(cfg *ServerConfig) (srv *EtcdServer, err error) {
246     st := store.New(StoreClusterPrefix, StoreKeysPrefix)
247
248     --- 7 lines: var (
249         --- 尝试初始化数据文件（没有就新建）
250         --- 256 --- 43 lines: if err := fileutil.TouchDirAll(cfg.DataDir); err != nil { ---
251             --- 299
252             --- 300         switch {
253                 case !haveWAL && !cfg.NewCluster: 没有WAL且需要加入一个已有的集群
254                 --- 302 --- 24 lines: if err = cfg.VerifyJoinExisting(); err != nil { ---
255                     --- 326         case !haveWAL && cfg.NewCluster: 没有WAL且参与组建一个集群
256                         --- 327 --- 33 lines: if err = cfg.VerifyBootstrap(); err != nil { ---
257                             --- 360         case haveWAL: 已经有WAL, 使用WAL来恢复节点和集群配置
258                             --- 361 --- 34 lines: if err = fileutil.IsDirWritable(cfg.MemberDir()); err != nil { ---
259                                 --- 395         default:
260                                     --- 396             return nil, fmt.Errorf("unsupported bootstrap config")
261                                 --- 397             }
262
263         --- 398         --- 399 --- 12 lines: if err := fileutil.TouchDirAll(cfg.MemberDir()); err != nil { ---
264             --- 411 --- 29 lines: srv = &EtcdServer{
265                 --- 440
266                 --- 441 --- 64 lines: srv.applyV2 = &applierV2{store: srv.store, cluster: srv.cluster}
267
268             --- 505
269             --- 506             return srv, nil
270         }
271
272     }
273
274     --- 507 }

NORMAL +0 ~16 -0 ⌂ d267ca9 etcdserver/server.go

```

函数一开始会初始化数据目录和文件，Fig 2.2 中重点关注最简单的逻辑，即节点参与组建一个新集群，即命令行初始化时指定参数：

`--initial-cluster-state new`.

Fig 2.2 --initial-cluster-state new 初始化 ETCD 节点

```

326     case !haveWAL && cfg.NewCluster:
327         if err = cfg.VerifyBootstrap(); err != nil { //{{{
328             --- 先初始化并校验输入配置的合法性
329             --- 329         return nil, err
330         }
331         cl, err = membership.NewClusterFromURLsMap(cfg.InitialClusterToken, cfg.InitialPeerURLsMap)
332         if err != nil {
333             --- 332             return nil, err
334         }
335         m := cl.MemberByName(cfg.Name)
336         if !isMemberBootstrapped(cl, cfg.Name, m.ID, cfg.BootstrapTimeout()) {
337             --- 336             return nil, fmt.Errorf("member %s has already been bootstrapped", m.ID)
338         }
339         if cfg.ShouldDiscover() {
340             --- 339             var str string
341             str, err = discovery.JoinCluster(cfg.DiscoveryURL, cfg.DiscoveryProxy, m.ID, cfg.InitialPeerURLsMap.String())
342             if err != nil {
343                 --- 342                 return nil, &DiscoveryError{Op: "join", Err: err}
344             }
345             var urlsmap types.URLsMap
346             urlsmap, err = types.NewURLsMap(str)
347             if err != nil {
348                 --- 347                 return nil, err
349             }
350             if checkDuplicateURL(urlsmap) {
351                 --- 350                 return nil, fmt.Errorf("discovery cluster %s has duplicate url", urlsmap)
352             }
353             if cl, err = membership.NewClusterFromURLsMap(cfg.InitialClusterToken, urlsmap); err != nil {
354                 --- 353                 return nil, err
355             }
356             cl.SetStore(st)
357             cl.SetBackend(be)
358             cfg.PrintlnInitial()
359             id, n, s, w = startNode(cfg, cl, cl.MemberIDs() // ))
360             case haveWAL:
361             --- 361 --- 34 lines: if err = fileutil.IsDirWritable(cfg.MemberDir()); err != nil { ---
362                 --- 395             default:
363
NORMAL +0 ~16 -0 ⌂ d267ca9 etcdserver/server.go

```

节点启动时会校验参数的合法性，比如 initial-cluster 的格式以及是否有重复记录等等，当参数校验成功后，就开始正式启动一个 Raft 模块。Fig 2.3 为初始化 Raft 节点。

Fig 2.3 初始化 Raft 节点

```

334 func startNode(cfg *ServerConfig, cl *membership.RaftCluster, ids []types.ID) (id types.ID, n raft.Node, s *raft.MemoryStorage, w *wal.WAL) {
335     var err error
336     member := cl.MemberByName(cfg.Name)
337     metadata := pbutil.MustMarshal(
338         &pb.Metadata{
339             NodeID: uint64(member.ID),
340             ClusterID: uint64(cl.IDC),
341         },
342     )  
          WAL开头写入一个metadata记录： NodeID, ClusterID
343     if w, err = wal.Create(cfg.WALDir); err != nil {
344         plog.Fatalf("create wal error: %v", err)
345     }
346     peers := make([]raft.Peer, len(ids))
347     for i, id := range ids {
348         ctx, err := json.Marshal(&cl.Member(id))    初始化peer，即节点
349         if err != nil {
350             plog.Panicf("marshal member should never fail: %v", err)
351         }
352         peers[i] = raft.Peer{ID: uint64(id), Context: ctx}
353     }
354     id = member.ID
355     plog.Infof("starting member %s in cluster %s", id, cl.IDC)
356     s = raft.NewMemoryStorage()
357     c := raft.Config{  
          初始化RAFT状态机配置
358         ID: id,
359         ElectionTick: cfg.ElectionTicks,
360         HeartbeatTick: 1,
361         Storage: s,
362         MaxSizePerMsg: maxSizePerMsg,
363         MaxInflightMsgs: maxInflightMsgs,
364         CheckQuorum: true,
365     }
366
367     n = raft.StartNode(c, peers) 启动RAFT状态机
368     raftStatusMu.Lock()
369     raftStatus = n.Status
370     raftStatusMu.Unlock()
371     advanceTicksForElection(n, c.ElectionTick)
372     return
373 }
NORMAL +0 ~0 ~0 ! d267ca9 etcdserver/raft.go

```

启动节点时，伴随着启动 Raft 状态机（如 Fig 2.4）。

Fig 2.4 初始化 Raft 状态机

```

172 // StartNode returns a new Node given configuration and a list of raft peers.
173 // It appends a ConfChangeAddNode entry for each given peer to the initial log.
174 func StartNode(c *Config, peers []Peer) Node {
175     r := newRaft(c) 用配置初始化raft状态机
176     // become the follower at term 1 and apply initial configuration
177     // entries of term 1
178     r.becomeFollower(1, None)   初始化集群节点，写入WAL文件
179     for _, peer := range peers {
180         cc := pb.ConfChange{Type: pb.ConfChangeAddNode, NodeID: peer.ID, Context: peer.Context}
181         d, err := cc.Marshal()
182         if err != nil {
183             panic("unexpected marshal error")
184         }
185         e := pb.Entry{Type: pb.EntryConfChange, Term: 1, Index: r.raftLog.lastIndex() + 1, Data: d}
186         r.raftLog.append(e)
187     }
188     // Mark these initial entries as committed.
189     // TODO(barnelli): These entries are still unstable; do we need to move them?
190     // the invariant that committed < uncommitted.
191     r.raftLog.committed = r.raftLog.lastIndex()  
强行将前面这几个WAL日志设置已经提交状态
192     // Now apply them, mainly so that the application can call Campaign
193     // immediately after StartNode in tests. Note that these nodes will
194     // be added to raft twice: here and when the application's Ready
195     // loop calls ApplyConfChange. The calls to addNode must come after
196     // all calls to raftLog.append so progress.next is set after these
197     // bootstrapping entries (it is an error if we try to append these
198     // entries since they have already been committed).
199     // We do not set raftLog.applied so the application will be able
200     // to observe all conf changes via Ready.CommittedEntries.
201     for _, peer := range peers {
202         r.addNode(peer.ID) 向raft状态机注册节点信息
203     }
204     n := newNode()
205     n.logger = c.Logger
206     go n.run(r) 启动状态机
207     return &n
208 }
NORMAL +0 ~0 ~0 ! d267ca9 raft/node.go

```

启动 Raft 状态机时，将集群节点信息写入 WAL 并强行提交，这个逻辑有可能是为了让集群中节点快速找到彼此，快速收敛。Raft 状态机初始化完毕后，Fig 2.5 启动这个状态机，接收和响应 Raft 消息。

Fig 2.5 启动 Raft 状态机

```

269 func (n *node) run(r *raft) {
270     var propc chan pb.Message
271     lead := None
272     prevSoftSt := r.softState()
273     prevHardSt := emptyState
274
275     for {
276         if advancec != nil { -->
277             if lead != r.lead {
278                 if r.hasLeader() { -->
279                     select {
280                         // TODO: maybe buffer the config propose if there exists one (the way
281                         // described in raft dissertation)
282                         // Currently it is dropped in Step silently.
283                         case m := <-propc:
284                             m.From = r.id
285                             r.Step(m)
286                         case m := <-n.recvC:
287                             // filter out response message from unknown From.
288                             if _, ok := r.prs[m.From]; ok || !IsResponseMsg(m.Type) {
289                                 r.Step(m) // raft never returns an error
290                             }
291                         case cc := <-n.confC: -->
292                             case <-n.tickC:
293                                 r.tick()
294                         case readyC <- rd: -->
295                             case <-advancec: -->
296                                 case c := <-n.status:
297                                     c <- getStatus(r)
298                                 case <-n.stop:
299                                     close(n.done)
300                                     return
301                                 }
302                         }
303                     }
304                 }
305             }
306         }
307     }
308
309     NORMAL +0 ~12 -0 ↵ d267ca9 raft/node.go

```

这个函数在这个 Raft 模块的逻辑中，算是相当晦涩难懂的了，其中使用了很多私有的 Channel，没有文档介绍，也没有注释。Table 1 大致列出关键的私有 Channel 变量的主要用途。

Table 1 私有 Channel 的主要用途

Channel\变量	主要用途
advancec	通知 Raft 节点上一次 Ready 已经执行 (Apply) 完了，可以开始处理下一个 Ready
propc	传入 Raft 状态机的指令的入口
recvC	从其他节点接收到的消息
tickC	定时器

readyc	传递当前Raft节点在指定时间的状态，如待写入WAL的消息和已经Commit的日志等等
applyc	传递已经Commit，等待在Raft状态机中执行的消息
confc	传递已经Commit，等待在Raft状态机中执行的动态修改消息

整个状态机的数据流非常复杂，这里主要关注非配置数据的写操作与 Raft 状态机的数据流。

Fig 2.6 中消息进入 Unstable Storage 之后，需要等待上一次 Apply 执行完毕，然后状态机会发起一个 Ready 消息，封装了还未持久化的消息和已经 Commit 的消息。

Fig 2.6 readyc 传递 Ready 消息

在 Fig 2.7 中，readyc 的接收者收到一个 Ready 消息之后，对其中

```

269 func (n *node) run(r *raft) {
270     --- 7 lines: var propc chan pb.Message -----
271
272     lead := None
273     prevSoftSt := r.softState()
274     prevHardSt := emptyState
275
276     for {
277         if advancec != nil { // {{{
278             readyc = nil      存在未提交的数据时
279         } else {
280             rd = newReady(r, prevSoftSt, prevHardSt)
281             if rd.containsUpdates() {
282                 readyc = n.readyc
283             } else {
284                 readyc = nil
285             }
286         } //}}}
287
288         if lead != r.lead {
289             --- 13 lines: if r.hasLeader() { -----
290
291             --- 12 lines: select {
292                 case cc := <-n.confc:
293                     case <-n.tick():
294                         r.tick()
295                     case readyc <- rd: // {{{
296                         if rd.SoftState != nil {
297                             prevSoftSt = rd.SoftState
298                         }
299                         if len(rd.Entries) > 0 {
300                             prevLastUnstableIndex = rd.Entries[Len(rd.Entries)-1].Index
301                             prevLastUnstableTerm = rd.Entries[Len(rd.Entries)-1].Term
302                             havePrevLastUnstableIndex = true
303                         }
304                         if !IsEmptyHardState(rd.HardState) {
305                             prevHardSt = rd.HardState
306                         }
307                         if !IsEmptySnapshot(rd.Snapshot) {
308                             prevSnapshot = rd.Snapshot.Metadata.Index
309                         }
310
311                         r.msgs = nil
312                         r.readStates = nil
313                         advancec = n.advancec //}}}
314             --- 10 lines: case <-advancec: -----
315                 case c := <-n.status:
316                     c <- getStatus(r)
317
318     NORMAL +0 ~14 -0 ↵ d267ca9 raft/node.go

```

的两种数据分别做了处理，没有持久化 WAL 的消息做持久化，而已经提交的消息就在状态机中执行。

Fig 2.7 处理 readyc 接收到的数据

Fig 2.6 与 Fig 2.7 分别显示了消息或日志的两种状态，即

```

132 // start prepares and starts raftNode in a new goroutine. It is no longer safe
133 // to modify the fields after it has been started.
134 func (r *raftNode) start(rd *raftReadyHandler) {
135     r.applyc = make(chan apply)
136     r.stopped = make(chan struct{})
137     r.done = make(chan struct{})
138     internalTimeout := time.Second
139
140     go func() {
141         defer r.onStop()
142         islead := false
143
144         for {
145             select {
146                 case <-r.ticker:
147                     r.Tick()
148                 case rd := <-r.ReadyC:    接收readyc的输入
149             +- 18 lines: if rd.SoftState != nil { -----
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168 +- 9 lines: if len(rd.ReadStates) != 0 { -----
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234 }O
NORMAL +0 ~8 -0 ↵ d267c09 | etcdserver/raft.go
start() go utf-8[unix] 42% ⏹

```

Unstable 和 Committed，在 Fig 2.8 中，ETCD 节点将消息传入 Raft 状态机，和整个集群一起来协商、决策，其实就是让 Stable 状态的 WAL 日志转变为已经提交 (Committed) 的日志。

Fig 2.8 消息传入 Raft 状态机的入口

```

269 func (n *node) run(r *raft) {
270     7 lines: var propc chan pb.Message -----
277
278     lead := None
279     prevSoftSt := r.softState()
280     prevHardSt := emptyState
281
282     for {
283         10 lines: if advanced != nil { -----
293
294         if lead != r.lead {
295             13 lines: if r.hasLeader() { -----
308
309             select { //{{{
310                 // TODO: maybe buffer the config propose if there exists one (the way
311                 // described in raft dissertation)
312                 // Currently it is dropped in Step silently.
313                 case m := <-propc: 本地产生的消息要经过raft状态机决策
314                     m.From = r.id
315                     r.Step(m)
316                 case m := <-n.recv: 从外部接收的消息经过raft状态机决策
317                     // filter out response message from unknown From.
318                     if _, ok := r.prs[m.From]; ok || !IsResponseMsg(m.Type) {
319                         r.Step(m) // raft never returns an error
320                     } //}}}
321         28 lines: case cc := <-n.confc: -----
NORMAL +0 ~14 -0 ⌂ d267ca9 raft/node.go

```

消息进入状态机之后，过程就变得简单多了。先来看一下整个状态机的入口，即 Fig 2.9 中的 Raft.Step 函数。

Fig 2.9 Raft.Step 函数

```

679 func (r *raft) Step(m pb.Message) error {
680     // Handle the message term, which may result in our stepping down to a follower.
681     switch {
682         case m.Term == 0:
683             // local message
684         29 lines: case m.Term > r.Term: -----
713         24 lines: case m.Term < r.Term: -----
737         switch m.Type {
738             21 lines: case pb.MsgHup: -----
759             18 lines: case pb.MsgVote, pb.MsgPreVote: -----
777             default:
778                 r.step(r, m) 正式开始状态机的迭代
779             }
780             return nil
781     }
NORMAL +0 ~8 -0 ⌂ d267ca9 raft/raft.go

```

值得一提的是，分布式系统中，通常都需要对收到的消息做一些基本的校验，数据修改相关的消息必须保持和节点在同一个 Lead Election 的上下文之内（Term 相同）。不符合要求的消息会被过滤，如 Fig 2.9 中提到的逻辑中，检查并对消息的 Term 和当前节点状态机的 Term 不一致的情况做了预处理：

- $m.Term > r.Term$: 说明当前节点已经相对落后集群状态了，由于集群状态比当前节点新，以集群最新状态为准，当前节点会变为 Follower。
- $m.Term < r.Term$: 说明收到的消息落后于当前节点，除了 Heartbeat 消息以外的消息都会被直接丢弃。

只有消息的 Term 与当前节点一致时，消息才会正式进入状态机进行决策。Fig 2.10 中的 STEP 实际上是一个函数变量，在状态机变化时可以修改这个变量的赋值，初始状态始终是 Follower。从这个角度看 Raft 的实现，其实就是看 STEP 函数对不同消息的处理逻辑。

Fig 2.10 STEP 函数

```
785 +-200 lines: func stepLeader(r *raft, m pb.Message) { -----
985
986 // stepCandidate is shared by StateCandidate and StatePreCandidate; the differ
987 // whether they respond to MsgVoteResp or MsgPreVoteResp.
988 +- 41 lines: func stepCandidate(r *raft, m pb.Message) { -----
1029
1030 +- 53 lines: func stepFollower(r *raft, m pb.Message) { -----
1083
NORMAL +0 ~14 -0 ↵ d267ca9 raft/raft.go
```

一共有三个 STEP 函数的实现，分别对应了 Raft 状态机中节点的三种状态：Follower、Candidate 和 Leader。

在正式介绍逻辑之前，先看一下常见消息类型：

- **MsgHup**: 本地节点中触发一次 Lead Election
- **MsgBeat**: 触发 Leader 向 Follower 发送一个 **MsgHeartbeat** 消息
- **MsgProp**: 尝试向本地日志中 Append 一条记录，在非 Leader 节点收到写请求的时候，会通过 **MsgProp** 类型将该请求转发给当前 Leader 节点
- **MsgApp/ MsgAppResp**: Leader 通知其他节点来同步日志的消息和响应

- **MsgVote/MsgVoteResp**: 要求其他节点给自己投票的消息和结果
- **MsgPreVote/MsgPreVote**: 模拟一次Lead Election, 但不修改集群的状态, 用来恢复脑裂状态下的集群
- **MsgSnap/MsgSnapStatus**: Leader发起让其他节点同步一个Snapshot的请求和响应
- **MsgHeartbeat/MsgHeartbeatResp**: 集群中的心跳和响应
- **MsgUnreachable**: 这是节点内部用以暂时停止向节点发消息的优化机制, 比如Follower节点正在生成快照, Leader节点就会暂时关闭向Follower发送MsgApp消息。

状态机逻辑也很复杂, 本文只梳理 Leader/Follower 状态的主要处理逻辑, 即 stepLeader 和 stepFollower 两个函数, 分别处理了对应状态。

Fig 2.11 stepLeader

```

785 func stepLeader(r *raft, m pb.Message) {
786     // These message types do not require any progress for m.From.
787     switch m.Type {           处理消息
788     --- 3 lines: case pb.MsgBeat: -----
789     --- 6 lines: case pb.MsgCheckQuorum: -----
790     --- 27 lines: case pb.MsgProp: -----
791     --- 31 lines: case pb.MsgReadIndex: -----
792
793     // All other message types require a progress for m.From (pr).
794     pr, prOk := r.pr[m.From]
795     if !prOk {
796         r.logger.Debugf("%x no progress available for %x", r.id, m.From)
797         return
798     }
799     switch m.Type {           处理响应
800     --- 40 lines: case pb.MsgAppResp: -----
801     --- 30 lines: case pb.MsgHeartbeatResp: -----
802     --- 16 lines: case pb.MsgSnapStatus: -----
803     --- 7 lines: case pb.MsgUnreachable: -----
804     --- 27 lines: case pb.MsgTransferLeader: -----
805     }
806 }
807
808 }
```

先看 Leader 节点对 MsgProp (如写数据的请求) 的处理逻辑。

Fig 2.12 MsgProp 在 Leader 节点中的处理逻辑

```

785 func stepLeader(r *raft, m pb.Message) {
786     // These message types do not require any progress for m.From.
787     switch m.Type {
788     case pb.MsgBeat:
789     case pb.MsgCheckQuorum:
790     case pb.MsgProp: //{{{
791         if len(m.Entries) == 0 {
792             r.logger.Panicf("%x stepped empty MsgProp", r.id) 空的消息不处理打个日志就算了，为什么panic?
793         }
794         if _, ok := r.prs[r.id]; !ok {
795             // If we are not currently a member of the range (i.e. this node
796             // was removed from the configuration while serving as leader),
797             // drop any new proposals. 检查当前节点是否还是集群的一员
798             return
799         }
800         if r.leadTransferee != None { 检查集群是否正处在手动切换Leader的过程中
801             r.logger.Debugf("%x [term %d] transfer leadership to %x is in progress; dropping proposal", r.id, r.Term, r.leadTransferee)
802             return
803         }
804         for i, e := range m.Entries {
805             if e.Type == pb.EntryConfChange {
806                 if r.pendingConf {
807                     r.logger.Infof("propose conf %s ignored since pending unapplied configuration", e.String())
808                     m.Entries[i] = pb.EntryNormal{Type: pb.EntryNormal}
809                 }
810                 r.pendingConf = true
811             }
812             r.appendEntry(m.Entries...) 消息正确接收，先放到内存中的unstable队列里面
813             r.broadcast() 通知其他节点
814         }
815     return //}}}
816     // All other message types require a progress for m.From (pr).
817     +0 -28 -0 ↵ d267ca9 raft/raft.go

```

再来看看 Follower 节点中的消息处理，Fig 2.13 主要关注 MsgProp、MsgApp 和 MsgHeartbeat。

Fig 2.13 MsgProp 在 Follower 节点中的处理

```

1030 func stepFollower(r *raft, m pb.Message) {
1031     switch m.Type {
1032     case pb.MsgProp: //{{{
1033         if r.lead == None {
1034             r.logger.Infof("%x no leader at term %d; dropping proposal", r.id, r.Term)
1035             return
1036         }
1037         m.To = r.lead 直接把请求转发给当前的Leader了
1038         r.send(m) //}}}

```

Follower 会把 MsgProp 转发给 Leader 节点，这就是为什么在一个集群中可以在任意节点发起读写请求，这样外部模块做负载均衡的逻辑也可以大大简化。

Fig 2.14 MsgApp 在 Follower 节点中的处理

```
1030 func stepFollower(r *raft, m pb.Message) {
1031     switch m.Type {
1032     case pb.MsgProp: //{{{
1033         if r.lead == None {
1034             r.logger.Infof("%s no leader at term %d; dropping proposal", r.id, r.Term)
1035             return
1036         }
1037         m.To = r.lead
1038         r.send(m) //}}}
1039     case pb.MsgApp: //{{{
1040         r.electionElapsed = 0
1041         r.lead = m.From
1042         r.handleAppendEntries(m) //}}}
1043     case pb.MsgHeartbeat: //{{{
1044         r.electionElapsed = 0
1045         r.lead = m.From
1046         r.handleHeartbeat(m) //}}}}
1047     --- 4 lines: case pb.MsgSnap: ---
1048     --- 7 lines: case pb.MsgTransferLeader: ---
1049     --- 10 lines: case pb.MsgTimeoutNow: ---
1050     --- 7 lines: case pb.MsgReadIndex: ---
1051     --- 7 lines: case pb.MsgReadIndexResp: ---
1052 }
1053
1054 func (r *raft) handleAppendEntries(m pb.Message) {
1055     if m.Index < r.raftLog.committed { 消息状态早于节点状态，通过响应告诉Leader当前节点的状态
1056         r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: r.raftLog.committed})
1057         return
1058     }
1059     if m.lastIndex, ok := r.raftLog.maybeAppend(m.Index, m.logTerm, m.Commit, m.Entries...); ok {
1060         r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: m.lastIndex})
1061     } else {
1062         r.logger.Debugf("%s [logterm: %d, index: %d] rejected msgApp [logterm: %d, index: %d] from %s",
1063                         r.id, r.raftLog.zeroTermOnErrCompacted(r.raftLog.term(m.Index)), m.Index, m.logTerm, m.Index, m.From)
1064         r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: m.Index, Reject: true, RejectHint: r.raftLog.lastIndex()})
1065     }
1066 }
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097 }
```

收到 App 消息时，Follower 首先会“毫不犹豫地”将 Lead Election 计时器清零；紧接着当前节点的 Leader 会直接切换成消息的来源节点；最后进入消息处理逻辑，需要关注两个小细节：

- 如果消息晚于当前节点状态，会给对应的Leader返回响应来告知自己当前的状态
 - 如果消息出现冲突（Term/Index和当前节点不一致），就会直接拒绝MsgApp消息

ETCD 对写请求的处理

前文简要的梳理了 Raft 协议在 ETCD 节点中的实现和数据流。本节以 ETCD3 中的一次普通的数据写 (V2/V3 API) 为例介绍 ETCD 中存储系统的设计和实现。注意本节之所以只关注“写”操作，是因为读操作直接访问本地存储，不涉及复杂的分布式决策流程。V3 API 中引入了 MVCC 存储，本质上与 V2 中的处理流程是一样的，区别在于最终执行时针对的本地存储不同（基于 V3.1.8 源码）。

先看 V3 API 中的 Put 操作的实现，Fig 3.1 为 V3 中 Put 操作的

gRPC Handler 定义。

Fig 3.1 V3 Put 的 gRPC Handler

```
func _KV_Put_Handler(serv Interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (Interface{}, error) {
    2293     l := log.New("kv", "Put", 0)
    2294     l.Infoln("11 lines: in := new(OutRequest)")
    2295     in := new(OutRequest)
    2296     l.Infoln("12 lines: handler := funcCtx context.Context, req Interface{} {Interface{}, error} {")
    2297     handler := funcCtx context.Context, req Interface{} {Interface{}, error} {
    2298         l.Infoln("13 lines:     return srv.(KVServer).Put(ctx, req.(*PutRequest))")
    2299         return srv.(KVServer).Put(ctx, req.(*PutRequest))
    2300     }
    2301     l.Infoln("14 lines: }")
    2302     l.Infoln("15 lines: return interceptor(ctx, in, info, handler)")
    2303     l.Infoln("16 lines: }")
    2304 }
```

这里的 KVServer 是一个 Interface，具体的 KVServer.Put 实现在 ETCDServer 中。

Fig 3.2 V3 API Put 操作的实现

这里有两个比较有意思的点：

- V3 Put消息（指令）传入Raft状态机后是怎么被执行的？
 - Raft状态机是一个分布式的一步过程，客户端访问V3 API时如何实现等待分布式计算的结果？

先看 V3 Put 指令在 Raft 状态机的执行过程。在 ETCD 中 Raft 状态机是对每个消息进行“协商”的过程，这个过程其实是不关注其具体内容的，因此这里提到的“执行”的范畴，实际上是消息已经通过分布式节点协商（Committed），要在本地存储中执行（Apply）的过程。

Fig 3.3 applyEntryNormal 中执行 V3 指令

```

1277 // applyEntryNormal applies an EntryNormal type raftpb request to the EtcdServer
1278 func (s *EtcdServer) applyEntryNormal(e *raftpb.Entry) {
1279     7 lines: shouldApplyV3 := false
1280
1281     // raft state machine may generate noop entry when leader confirmation.
1282     // skip it in advance to avoid some potential bug in the future
1283     12 lines: if len(e.Data) == 0 {
1284
1285         7 lines: var raftReq_pb.InternalRaftRequest
1286         10 lines: if raftReq.V2 != nil {
1287
1288             id := raftReq.ID //{{{
1289             if id == 0 {
1290                 id = raftReq.Header.ID
1291             }
1292
1293             var ar *applyResult
1294             needResult := s.w.IsRegistered(id) 通知waiter
1295             if needResult || !noSideEffect(&raftReq) {
1296                 if !needResult && raftReq.Txn != nil {
1297                     removeNeedlessRangeReqs(raftReq.Txn)
1298                 }
1299                 ar = s.applyV3.Apply(&raftReq) 写入本地MVCC存储
1300             } //}}}
1301
1302         8 lines: if ar == nil {
1303
1304             11 lines: plog.Errorf("applying raft message exceeded backend quota")
1305         }
1306
1307 NORMAL +0 ~30 -0 ↵ d267ca9 etcdserver/server.go

```

值得一提的是，ETCD 中采用一种有趣的方式来实现客户端等待异步任务的策略，客户端请求注册一个 Waiter 并监听这个 Waiter 对应的一个 Channel，客户端会一直阻塞，直到这个 Waiter 被触发或超时。

V2 指令的执行过程基本类似，Fig 3.4 为 V2 API 一次 Set 操作的 gRPC 接口定义。

Fig 3.4 V2 Set 操作 gRPC Handler

```

51 func (a *v2apiStore) processRaftRequest(ctx context.Context, r *pb.Request) (Response, error) {
52     data, err := r.Marshal() //{{{
53     if err != nil {
54         return Response{}, err
55     }
56     ch := a.s.w.Register(r.ID) 注册一个客户端的waiter
57
58     start := time.Now() //}}}
59     a.s.r.Propose(ctx, data) 请求传入raft状态机
60     14 lines: proposalsPending.Inc()
61
62 }
NORMAL +0 ~4 -0 ↵ d267ca9 etcdserver/v2_server.go

```

决策并提交指令之后，就需要在状态机中执行并修改内存数据库的状态。

Fig 3.5 applyEntryNormal 中执行 V2 指令

```

1277 // applyEntryNormal applies an EntryNormal type raftpb request to the EtcdServer
1278 func (s *EtcdServer) applyEntryNormal(e *raftpb.Entry) {
1279     --- 7 lines: shouldApplyV3 := false -----
1280
1281     // raft state machine may generate noop entry when leader confirmation.
1282     // skip it in advance to avoid some potential bug in the future
1283     --- 12 lines: if len(e.Data) == 0 { -----
1284
1285         var raftReq pb.InternalRaftRequest           //{{{
1286         if !poutil.MaybeUnmarshal(&raftReq, e.Data) { // backward compatible
1287             var r pb.Request
1288             poutil.MustUnmarshal(&r, e.Data)   写入v2的内存数据库
1289             s.w.Trigger(r.ID, s.applyV2Request(&r))
1290             return
1291         } //}}}
1292         if raftReq.V2 != nil { //{{{
1293             req := raftReq.V2
1294             s.w.Trigger(req.ID, s.applyV2Request(req))
1295             return
1296         } }
1297
1298         // do not re-apply applied entries.
1299         if !shouldApplyV3 {
1300             return
1301         } //}}}
1302     } --- 13 lines: id := raftReq.ID -----
1303
1304     --- 8 lines: if ar == nil { -----
1305
1306     --- 11 lines: plog.Errorf("applying raft message exceeded backend quota") -----
1307
1308 }

```

NORMAL +0 ~30 -0 ↵ d267ca9 etcdserver/server.go

本地存储的演变

随着存储系统的改良，ETCD3 可以提供生产级别（可靠性）的分布式存储。ETCD 本地存储系统的演变分为三个阶段：V1(0 ~ V0.4.9)，V2(V2.x) 和 V3(>= V3.x)，本节结合不同版本中与存储有关的核心逻辑的实现，分析这些改进为系统带来的好处，也希望为高可用 Go 模块设计提供一些思路。

WAL 日志持久化时机优化

首先考察 WAL (Write-Ahead-Log) 日志处理机制。在 V1 中 WAL 日志是随着每个消息的到来同步写磁盘的，使得集群稳定性对磁盘稳定性很敏感。

Fig 4.1.1 V1 中 WAL 日志直接写磁盘

```

466 // Appends a series of entries to the log.
467 func (l *Log) appendEntries(entries []*protobuf.LogEntry) error {
468     l.mutex.Lock()
469     defer l.mutex.Unlock()
470
471 +- 6 lines: startPosition, _ := l.file.Seek(0, os.SEEK_CUR) -----
472     // Append each entry but exit if we hit an error.
473 +- 6 lines: for i := range entries { -----
474
475     3 lines: if size, err = l.writeEntry(logEntry, w); err != nil {
476
477     10 lines: startPosition += size -----
478 }
479
480
481 // Writes a single log entry to the end of the log.
482 func (l *Log) appendEntry(entry *LogEntry) error {
483     l.mutex.Lock()
484     defer l.mutex.Unlock()
485
486 +- 17 lines: if l.file == nil { -----
487
488     3 lines: if _, err := entry.Encode(l.file); err != nil { -----
489
490     3 lines: // Append to entries list if stored on disk.
491     3 lines: l.entries = append(l.entries, entry) -----
492
493 }
494
495 NORMAL +0 ~27 -0 ¶ 9fa3bea third_party/github.com/gorraft/raft/log.go

```

Fig 4.1.1 中是 V0.4.9（此前的版本这部分实现没有变化）中的代码片段。在 appendEntries 里让我有些疑惑的地方是前后加了两次锁，从实现上来看，就是希望写入 WAL 的操作绝对互斥，非常容易成为整个系统的瓶颈。

同步写磁盘的弊端一方面会导致后续的命令处理都被阻塞，节点可能会丢失心跳，最后整个集群会不稳定；另一方面整个集群的可靠性非常依赖磁盘的稳定性，任何一次写入失败都可能损坏数据。道理很简单，多次写磁盘就会导致本来小概率的事件被放大成大概率事件了。

从 V2 开始，WAL 的文件改为异步写，不是每个请求都写文件，转而用一个时钟来控制，事件到来时先写入 Unstable 队列，这是内存中的一个数组，然后批量写入 WAL 文件。

WAL 文件管理优化

先看 V1 中的 WAL 文件管理，这个阶段的实现整体还比较粗糙，没有完善的 Write-Ahead-Log 的概念，目录也叫做 Log。

Fig 4.2.1 V1 中 Log 存储编码和格式

```

70 // Encodes the log entry to a buffer. Returns the number of bytes
71 // written and any error that may have occurred.
72 func (e *LogEntry) Encode(w io.Writer) (int, error) { %8x
73 --- 4 lines: b, err := proto.Marshal(e.pb) ----- protobuf编码的record
77
78 --- 3 lines: if _, err = fmt.Fprintf(w, "%8x\n", len(b)); err != nil { -----
81
82     return w.Write(b)
83 }

```

NORMAL +0 ~4 -0 P 9fa3bea third_party/github.com/goraft/raft/log_entry.go Encode() go utf-8[1]

V1 中 Log 每条记录格式都是两行，前一行为数据长度，后一行为 Protobuf 编码的日志记录。V2 开始 WAL 日志格式更紧凑，同时每条记录都有对应的 crc 字段，数据更可靠。

工程上不可能让这样的 Log 无休止地增长，V1 中的解决方案是随着 Snapshot 的时机回收一部分日志，如 Fig 4.2.2 所示。

Fig 4.2.2 V1 中截断历史日志实现

```

569 --- 17 lines: var entries []logEntry
570
571 // create a new log file and add all the entries
572 new_filepath := l.path + ".new"
573 file, err := os.OpenFile(new_filepath, os.O_APPEND|os.O_WRONLY, 0600)
574 if err != nil {
575     return err
576 }
577
578 --- 10 lines: for _, entry := range entries {
579     file.WriteString(entry.String())
580 }
581
582 file.Sync()
583
584 old_file := l.file
585
586 // rename the new log file 然后用.new文件覆盖当前的log文件
587 err = os.Rename(new_filepath, l.path)
588 if err != nil {
589     file.Close()
590     os.Remove(new_filepath)
591     return err
592 }
593 l.file = file
594
595 // close the old log file
596 old_file.Close()
597
598 --- 2 lines: old_file.Close()
599
600

```

s.DispatchEvent(milieuEvent{leasePerfEventType, name, n1})

// Write the configuration to File.

s.writeConf()

return nil

// log compaction

1386 func (s *server) TakeSnapshot() error {
1387 --- 42 lines: if s.stateMachine == nil {
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451 }

每次回收200条

// We keep some log entries after the snapshot. 每次回收200条

// We do not want to end the whole snapshot to the slightly slow machines

(l.lastTermIndex, l.log, l.logOffsetOfLastEntry) = l.takeSnapshot()

compactIndex := l.logIndex - NumberOfEntriesInSnapshot

compactTerm := s.log.getEntry(compactIndex).Term

s.log.compact(compactIndex, compactTerm)

return nil

NORMAL +0 ~2 -0 P 9fa3bea ./server.go TakeSnapshot() go utf-8(unix) 83k 1234/1484 in : 25

V1 中的 Log 只有一个文件，显然是一种“把所有鸡蛋放在一个篮子里”的策略，风险很大。一旦这个 Log 文件因为各种原因损坏，那么这个节点就无法恢复了。

V2 开始 Log 开始有规范的管理，进入了多个 WAL 文件分片存储阶段，目前正在读写的 WAL 文件达到 64MB 后，文件写入流会被切换到一个新的

文件，可靠性和性能都有显著提升：

- 数据分片保存，就算损坏也仅仅损失了64MB日志
 - 每次不需要对老文件作覆盖操作，新的日志也不会因为覆盖失败（权限不够）一直不成功
 - 不需要回收历史日志，一旦日志分片保存，只需删除过期的日志文件即可

V3 这部分也没有明显变化，仅仅对 WAL 文件分片部分的逻辑进行了局部的重构。

Fig 4.2.3 V2 开始的 WAL 分片

每次写入时检查一下当前的文件大小，超过 64MB 就截断并切换到新的文件，切分文件的过程也比较容易理解：

- 创建 tmp 文件，写入crc校验结果、Metadata和集群的状态信息
 - 将tmp文件重命名为正式的WAL文件
 - 将节点的WAL日志写入流切换到新的WAL文件

Snapshot(快照)触发时机的优化

值得一提的是快照 (Snapshotting) 触发时机的优化，频繁的快照一方面会带来频繁的磁盘 I/O，并且快照时实际上需要对存储加互斥锁，不仅影响可靠性，也影响性能。本节分析了 V1-V3 中快照触发机制和优化。

Fig 4.3.1 V0.4.9 中持久化快照时机

```

427 func (s *PeerServer) monitorSnapshot() {
428     for {
429         time.Sleep(s.snapConf.checkingInterval)
430         currentIndex := s.RaftServer().CommitIndex()
431         count := currentIndex - s.snapConf.lastIndex
432         if uint64(count) > s.snapConf.snapshotThr { 触发时机是根据commit日志来计算的
433             err := s.raftServer.TakeSnapshot()
434             s.logSnapshot(err, currentIndex, count)
435             s.snapConf.lastIndex = currentIndex
436         }
437     }
438 }
NORMAL +0 ~0 -0 ↵ f9d27c3 server/peer_server.go

```

raftEventLogger

V1 中快照的触发是根据 Commit Index 来计算绝对偏移量的，这就导致会出现大量不必要的 Snapshot，一方面系统性能、稳定性会受到影响，另一方面频繁的 I/O 也对文件系统的可靠性形成很强的依赖，最终影响整个集群的稳定性。

Fig 4.3.2 V2/V3 中触发快照时机

```

574 func (s *EtcdServer) applyAll(ep *etcdProgress, apply *apply) {
575     -- 6 lines: s.applySnapshot(ep, apply)
581     s.triggerSnapshot(ep)
582     -- 7 lines: select {
583     }
589 }
590
591     -- 84 lines: func (s *EtcdServer) applySnapshot(ep *etcdProgress, apply *apply) {
675
676 func (s *EtcdServer) triggerSnapshot(ep *etcdProgress) {
677     if ep.appliedi->ep.snapi <= s.snapCount {
678         return
679     }
680
681     // When sending a snapshot, etcd will pause compaction.
682     // After receives a snapshot, the slow follower needs to get all the entries right after
683     // the snapshot sent to catch up. If we do not pause compaction, the log entries right after
684     // the snapshot sent might already be compacted. It happens when the snapshot takes long time
685     // to send and save. Pausing compaction avoids triggering a snapshot sending cycle.
686     if atomic.LoadInt64(&s.inflightSnapshots) != 0 {
687         return
688     }
689
690     plog.Infof("start to snapshot (applied: %d, lastsnap: %d)", ep.appliedi, ep.snapi)
691     s.snapshot(ep.appliedi, ep.confState)
692     ep.snapi = ep.appliedi
693 }
NORMAL +0 ~6 -0 ↵ 7e4fc7e etcdserver/server.go

```

V3 中触发快照的时机和 V2 相同，都是在 Apply 阶段（执行已经 Commit 的 Log 的时候）。看似 V2/V3 的实现是一样的，没有变化？

其实在设置相同的一snapshot-count 时，V3 做了一个优化，可以很大程度上减少不必要的 Snapshot 开销。尽管 V2/V3 中触发 Snapshot 的时机都在执行 (Apply) 一条指令的时间，SYNC 指令会触发频繁的 Snapshot。

Fig 4.3.3 V2 中 SYNC 指令的触发时机

500ms 一次的 SYNC 操作是频繁 Snapshot 的罪魁祸首之一，这里的 SYNC 指令是用来干什么的呢？这个 SYNC 就是用来清除过期的 TTL key 的。精度写死了 500ms，这就不难理解为什么 ETCD 里面 TTL 精度是 1s 了。V3 里对这个地方做了一个判断，只在 V2 存储中有 TTL key 的时候才发起 SYNC 指令。

Fig 4.3.4 V3 中的 SYNC 指令的触发时机

```
740     case <-getSyncC():
741         if s.store.HasTTLKeys() {
742             s.sync(s.Cfg.RqTimeout())
743         }
744     case <-s.stop:
```

此处的优化可以减少没有 TTL key 时的 Snapshot 操作，也可以降低频繁 SYNC 指令造成的 V2 存储的全局互斥锁，性能更稳定；当然，如果 V3 中始终存在 TTL key，这个优化就不起作用了。

需要强调的是，以上提到的 TTL key 是 V1 中引入的特性。在 V3 的存储中，TTL key 从属于 Lease，由 Leader 来统一管理，就不需要每个节点频繁、定时（500ms）清理过期的 Key 了。

Snapshot 文件管理优化

ETCD 中快照的主要内容是内存数据库的 JSON 编码版本，可以用此数据直接将内存数据库快速恢复至某一个稳定状态。从 ETCD 的数据模型上看，快照可以加速节点启动，当日志比较多的时候，快照可以让节点从某一个状态直接恢复。所以就算快照损坏了，也不应该影响节点启动。

Fig 4.4.1 V1 Snapshot 实现

```
    // logSnapshot logs about the snapshot that was taken.
    func (s *peerServer) logSnapshot(err error, currentIndex, count int) {
        info := fmt.Sprintf("%s: snapshot of %d events at index %d", s.Config.Name, count, currentIndex)
        log.Infof("%s", info)
    }

    if err != nil {
        log.Infof("%s attempted and failed: %v", info, err)
    } else {
        log.Infof("%s completed", info)
    }
}

func (s *peerServer) startRoutine(f func()) {
    s.routineGroup.Add(f)
    go f()
    defer s.routineGroup.Done()
    f()
}
}

定时生成快照

func (s *peerServer) monitorSnapshot() {
    for {
        // 7 lines: timer := time.NewTimer(s.snapConf.checkInInterval)
        currentIndex := s.RaftServer.CommittedIndex()
        count := currentIndex + s.snapConf.LastIndex
        if uint64(count) > s.snapConf.snapshotThreshold {
            err := s.raftServer.TakeSnapshot()
            if err != nil {
                log.Errorf("err: %v", err)
                s.logSnapshot(err, currentIndex, count)
                s.snapConf.lastIndex = currentIndex
            }
        }
    }
}

只保留最新的快照文件
```

V1 中的快照还比较简单，始终只保留 1 个最新的快照。V1 中快照如果损坏，节点就会直接挂掉无法重启，需要手动清除损坏的快照。

Fig 4.4.2 V2 Snapshot 实现

```
574 func (s *EtcdServer) applyAll(ep *etcdProgress, apply *apply) {
575     // 6 lines: s.applySnapshot(ep, apply)
576     s.triggerSnapshot(ep)
577     // 起发快照的时机就是apply
578     // 7 lines: select {
579     // ...
580 }
581
591     // 63 lines: func (s *EtcdServer) applySnapshot(ep *etcdProgress, apply *apply) {
592     // ...
593 }
594
595     // 20 lines: func (s *EtcdServer) applyEntries(ep *etcdProgress, apply *apply) {
596     // ...
597 }
598
599 func (s *EtcdServer) triggerSnapshot(ep *etcdProgress) {
600     // 3 lines: if ep.appliedi->ep.snapi <= s.snapCount {
601
602         // When sending a snapshot, etcd will pause compaction.
603         // After receives a snapshot, the slow follower needs to get all the entries right after
604         // the snapshot sent to catch up. If we do not pause compaction, the log entries right after
605         // the snapshot sent might already be compacted. It happens when the snapshot takes long time
606         // to send and save. Pausing compaction avoids triggering a snapshot sending cycle.
607     // 3 lines: if atomic.LoadInt64(&s.inflightSnapshots) != 0 {
608
609         log.Infof("start to snapshot (applied: %d, lastsnap: %d)", ep.appliedi, ep.snapi)
610     // 2 lines: s.snapshot(ep.appliedi, ep.confState)
611
612 }
613
614
615 // Stop stops the server gracefully, and shuts down the running goroutine.
616 // Stop should be called after a Start(s), otherwise it will block forever.
617 // 473 lines: func (s *EtcdServer) Stop() {
618
619     // TODO: non-blocking snapshot
620
621 func (s *EtcdServer) snapshot(snapshot uint64, confState raftpb.ConfState) {
622     // 3 lines: clone := s.store.Clone()
623
624     go func() {
625         defer s.wg.Done()
626
627         // 6 lines: d, err := clone.SaveNoCopy()
628
629     // 14 lines: snap, err := s.r.raftStorage.CreateSnapshot(snapshot, &confState, d)
630         // SaveSnap saves the snapshot and releases the locked wal files
631         // to the snapshot index.
632
633     // 4 lines: if err = s.r.storage.SaveSnap(snap); err != nil {
634
635         // keep some in memory log entries for slow followers.
636
637         // 4 lines: compacti := uint64(1)
638
639         // 10 lines: err = s.r.raftStorage.Compact(compacti)
640
641     }()
642 }
```

```

61 func (s *Snapshotter) SaveSnap(snapshot raftpb.Snapshot) error {
62     3 lines: if raft.IsEmptySnap(snapshot) { -----
63         return s.save(&snapshot)
64     }
65
66     func (s *Snapshotter) save(snapshot *raftpb.Snapshot) error {
67         start := time.Now()
68
69         fname := fmt.Sprintf("%016x-%016x%s", snapshot.Metadata.Term, snapshot.Metadata.Index, snapSuffix)
70
71         b := pbutil.MustMarshal(snapshot)
72         d, err := snap.Marshal()
73
74         6 lines: err = pioutil.WriteAndSyncFile(path.Join(s.dir, fname), d, 0666)
75
76         10 lines: if err != nil {
77             log.Panicf("Failed to write snapshot %s: %v", fname, err)
78         }
79     }
80
81     10 lines: err = pioutil.WriteAndSyncFile(path.Join(s.dir, fname), d, 0666)
82
83 }
NORMAL +0 ~8 -0 ↵ v2.3.8-local snap/snapshotter.go

```

触发快照的时间是 Apply 阶段，即执行已经 Commit 的 Log 阶段。保存快照操作改为异步执行，不再阻塞节点。存储格式相比 V1 发生了改变，数据结构更清晰。这里需要特别提一下在 V2 中的一个优化：当 Snapshot 损坏时，不再需要手动清理损坏的文件了。

Fig 4.4.3 节点启动读取 Snapshot 实现

```

94 func (s *Snapshotter) Load() (*raftpb.Snapshot, error) {
95     4 lines: names, err := s.snapNames()
96
97     var snap *raftpb.Snapshot
98     for _, name := range names {
99         if snap, err = loadSnap(s.dir, name); err == nil {
100             break
101         }
102     }
103
104     if err != nil {
105         return nil, ErrNoSnapshot
106     }
107     return snap, nil
108 }
109
110 func loadSnap(dir, name string) (*raftpb.Snapshot, error) {
111     fpath := path.Join(dir, name)
112     snap, err := Read(fpath)
113     if err != nil {
114         renameBroken(fpath)
115     }
116     return snap, err
117 }
118
119
NORMAL +0 ~14 -0 ↵ v2.3.8-local snap/snapshotter.go

```

Snapshot 文件损坏考虑了两种情况：

- 文件损坏，内容已经无法解析
- 内容可以读取，但crc校验失败，数据可能被篡改

如果文件损坏，读取 Snapshot 时会自动将该文件重命名为 .broken 文件，不过节点会启动失败，然后重启即可自动恢复。

总结

ETCD 到目前为止经历了三个阶段：

- V1：概念验证阶段，Raft 协议层和存储实现都比较粗糙
- V2：形成了稳定、可扩展的 Raft 协议层和存储层
- V3：增加了 V3 存储实现，这一阶段关注性能、安全性和运维成本

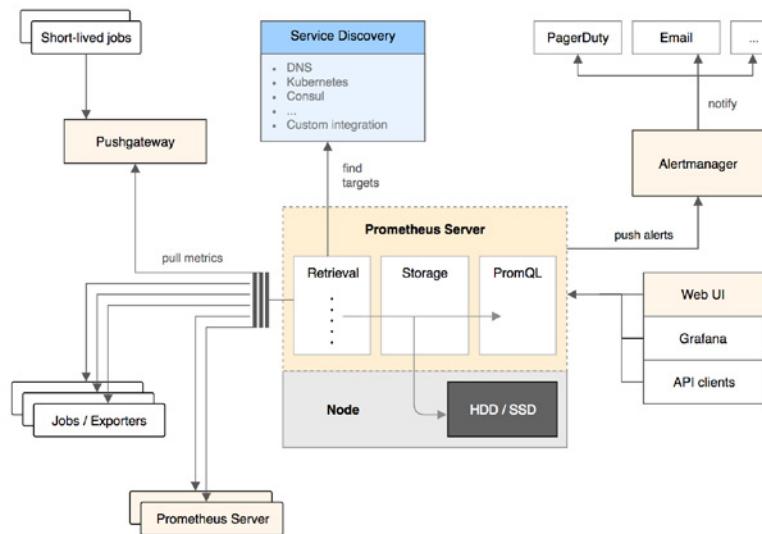
ETCD 的存储在 V2 之后形成了稳定可扩展的 Raft 协议层，在 V3 中又加入了在本地持久化的 BoltDB 实现，支持了更大的数据集，并对客户端的访问做了一些优化，系统也更可靠了，但整体数据模型并没有变化，V3 存储扩展了消息类型，同时增加了对这些消息的处理逻辑（对 V3 存储的读 / 写操作），整个 Raft 协议层改动不大。

另外不得不提的是它的监控方案。ETCD 从 V2 开始引入原生的 Prometheus 支持，V3 中又增加了很多 Metrics，基于 Metrics 可以从宏观的角度来刻画整个集群的状态和变化，相比基于日志的监控手段，这样的方案减少了大量的 I/O 操作。

为什么减少 I/O 操作对 ETCD 这么重要？这是因为 ETCD 本身是一个对磁盘高度敏感的系统，如果存在大量的日志读写，可能导致节点在 I/O 上出现阻塞，进而丢失“心跳”，集群就会出现频繁的主从切换，影响稳定性。

而基于 Prometheus 的监控，完全在内存中维护一个最新的状态，由 Prometheus 定时主动 Pull 某个时间点的数据。Fig 5.1 是基于 Prometheus 监控的架构。

Fig 5.1 Prometheus 监控系统架构



当然它还支持由被监控对象自己 Push Metrics，但通常来讲，都是让 Prometheus Server 主动 Pull Metrics。这样的好处是显然的，被监控对象不需要在内存中维护历史数据，只维护当前最新的数据即可。

另外，让人爱不释手的是与 Prometheus 完美集成的 Grafana，尽管以前接触得不多，但在 Prometheus 的应用中，它发挥了非常关键的作用，提供了非常丰富的可视化功能，非常容易上手。

实践中，我们也发现 Prometheus 默认的启动配置存在优化空间，如果没有独立的服务器部署，一定要注意控制它的资源占用，比较可控的部署方式是通过容器或 Systemd，在 Systemd 中可以通过加 Fig 5.2 中的配置来控制它的 RAM/CPU 占用率。这里限制 CPU 占用最多 20%，内存最多 8G。

Fig 5.2 Systemd 限制 Prometheus 的 RAM/CPU 占用

```
[Unit]
Description=Prometheus
Documentation=https://prometheus.io/docs/introduction/overview/

[Service]
User=application
EnvironmentFile=/etc/systemd/system/multi-user.target.wants/prometheus.env
WorkingDirectory=/opt/freewheel/7.5.0.4.0/
# ExecStartPre=/bin/bash -c "rm -rf /tmp/prometheus.* & rm -rf /tmp/prometheus.log"
ExecStart=/bin/bash -c "${BIN_PATH} ${CONFIG_FILE} ${WEB_LISTEN_ADDRESS} ${STORAGE_LOCAL_PATH} ${STORAGE_LOCAL_RETENTION} ${STORAGE_LOCAL_TARGETHEAPSIZE} >> ${LOG}"
TimeoutStopSec=20
CPUQuota=20%
MemoryLimit=8192M
LimitNOFILE=1024:4096
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

内存还有一处优化，就是“`-storage.local.target-heap-size 8589934592`”启动参数，这个非常重要。官方文档中建议将这个参数设置为可用内存的 2/3，我限制了 8G 的内存，其实这里最好设置为 5G 左右。这个参数可以将数据直接加载到内存中，让分析计算变得更加高效。如果查询的时间跨度比较大，建议给大一点的值。

最后就是磁盘的优化了。Prometheus 在启动时，可以指定仅保留过去多长时间内的数据：“`-storage.local.retention 120h0m0s`”，这是非常美妙的，结合这个参数，可以让磁盘占用总是处在可控的范围内。

gRPC 实践中的两个问题与思考

gRPC 与 Protocol Buffer

如果你熟悉 Thrift，那这部分内容你可以直接跳过，以后面的内容来说，你可以认为 Thrift 与 gRPC 做的是相同的事情。

要介绍 gRPC，首先要知道 Protocol Buffer（以下简称 Protobuf）是什么。

Protobuf 是谷歌公布的一种与平台、编程语言无关的数据序列化和反序列化机制，通过官方对多种语言提供的库和编译器，可以实现跨语言的数据传递，目前支持的编程语言有 Java、C++、Python、Ruby、PHP、Go 等。目前的主要版本有 Protobuf2 和 Protobuf3。下文讨论中涉及的是 Protobuf3。

gRPC 也是谷歌公司公布的技术，是一个高性能的、基于 HTTP/2 协议的 RPC 框架，默认以 Protobuf 作为数据传输方案。gRPC 也提供了对多种

编程语言的支持，比如 Java、C++、Python、Ruby、Go 等。

换句话说，Protobuf 首先解决“如何在不同语言之间传递数据”的问题，而 gRPC 在这个基础之上，解决了“如何跨语言进行 RPC 调用”的问题。

通过 gRPC，你可以很方便地实现一个 Client 端和 Server 端，而它们可能是通过两种不同的语言实现的。

下文中，我将介绍两个我们在实际使用 gRPC 过程中遇到的问题，以及我们是如何考虑的。

三状态问题

我们提供了一些 API 给客户使用，并要求将数据以 JSON 格式发给我们。由于业务上的需求，我们需要允许客户在 JSON 中表达这样三种信息：“有 key 且有值”，“有 key 但值为 null”和“key 不存在”。

例如，某 API 中需要客户提供“a”，“b”，“c”三个属性的值，而客户发出请求的 JSON 数据可能是： { "b": "xxx", "c": null } 这里表达的含义是“a”不存在，“b”的值是“xxx”，“c”的值是 null，在业务逻辑中，它们代表着不同的处理方式。

如何表达两种状态：“有 key 且有值”与“有 key 但值为 null”

首先来看看如何表达两种状态，这可能是多数人使用 gRPC 或者 Protobuf3 时会遇到的。

常见的有下面四种办法。

方案一：使用 oneof

oneof 是 Protobuf 的一个关键字，在官方介绍中，oneof 的用途是：“如果你的 Message 中有很多可选的（Optional）属性，并且这些属性在同一时刻最多只有一个有值，那么你可以使用 oneof 功能做到，同时还能节省存储空间。”

所以，如果被 oneof 限制的属性只有一个，表达的含义就等于“这个属性可能有值，也可能没值（相当于 null）”。

比如有 Message 定义如下：

```
message Request {
    oneof body_oneof {
        string body = 1;
    }
}
```

生成的代码中会有方法“GetBodyOneof()”，返回类型是接口，可以通过判断该接口的实际类型是否是“Request_Body”，来判断值是否是 null，比如：

```
if x, ok := GetBodyOneof().(*Request_Body); ok {
    // body不为null
}
```

可见，虽然功能上能够做到，但 oneof 从设计本意上来说并不是为了“值可以为 null”这种需求，而且这些代码也显得比较啰嗦，所以总的来说不是一个很好的办法。

方案二：使用标记 map

在 GitHub 的一个有很长评论列表的 [Issue](#) 中，有人提出了这个方案。

简单来说，就是利用一个 map 来记录 Message 中哪些属性被赋了值，map 的 key 为属性自己的编号，如果被赋值，则在 map 中将对应编号 key 的值为 true。因此，想知道某个属性的值是否是 null，只需要检查这个 map 中对应的 key 的值是否为 true。

具体实现来说，可以自己写一个 Protobuf 的插件，该插件可以对每个 Message 生成一个 map，还可以顺便生成一个 HasXXX() 方法来方便编码做判断。

似乎很美，是么？

注意，Protobuf 生成的 Struct 的属性都是“导出”的（也就是大写

字母开头），相当于 Java 中的“public”成员变量，而 SetXXX() 方法更是压根儿没有。这可能是 Go 语言的风格，好坏在此暂且不论。至少对这个方案来说，运行程序时，需要在每次给属性赋值后手动去修改这个 map，取值前也要专门做判断，这是很容易出错的，所以也不是一个很好的方案。

当然，你也可以进一步修改代码生成的结果，干脆将属性的名字改成“未导出”的，生成对应的 Setter 方法，并修改 Getter 方法的逻辑，然而这样做的代价太大了，也不符合 Go 的风格，而不符合大家一致遵守的风格的后果，可能就是你的代码与一些第三方库不兼容，这会是一个更让人头疼的问题。

方案三：使用 Wrapper

可能是因为在 Protobuf3 中这个问题太常见了，为了平息“众怒”，谷歌官方“丢”了“wrappers.proto”这么个东西出来，里面定义了各种基本类型的包装器，有点儿像 Java 中的 int 与“Integer”的关系，比如：

```
message Int64Value {
    int64 value = 1;
}
```

你可以用“Int64Value”这个“message”，来替换掉“Int64”这个“基本类型”，而 Message 在 Go 中的零值（Zero Value）是 nil，问题也就解决了。不管怎么说，与前面两种方案比起来，这个方案至少看起来更合理一些，而且毕竟是谷歌自己提出的方案，官方“钦定”的感觉总是能让人不好拒绝。

前面的三个方案中，多数人最终选择的是第三种，Wrapper 方案。

如何表达第三种状态：“key 不存在”

好了，再往前进一步，看看再增加一种状态该怎么办。

方案一：在 Wrapper 方案的基础上做扩展

自然而然的，在Wrapper方案的基础上，如果要做扩展，就会是这样的：

```
message Int64Value {
    int64 value = 1;
    bool exist = 2;
}
```

添加一个布尔类型的属性，为 true 时表示 key 存在，否则表示 key 不存在。可以看到有一些第三方库也是这么做的。

这么做的优点很明显，容易实现并且容易理解。而缺点也很明显——这一次，没有一个像谷歌这样的组织来带头提出一份定义文件。所以，你需要自己定义这个“包装器”。而你自己定义的“包装器”是绝对不会第三方项目支持的，这会在未来给你造成一些兼容性方面的不便利，这一点需要你自己去衡量。

结合我们自己的情况考虑，最终选择的是这个方案。

方案二：通过额外的数据传递第三种状态

这种思路的目的是，在只使用基本类型的基础上解决“三状态”问题，如此一来也就不会有前一种方案中的兼容性问题了。

该方案最终没有被采用，原因是在权衡了复杂度和风险，与比较紧迫的进度安排之后，决定放弃，但我认为仍然值得一提。

gRPC 支持通过 Header 和 Trailer 在 Client 和 Server 端传递 Metadata，我们可以利用这个功能。

我们知道，在将用户输入的 JSON 转成 Protobuf 对象的时候，会丢失一些信息，使得无法表示“三种状态”。“丢失的信息”包括两部分，分别是“哪些 key 的值是 null”和“哪些 key 在 JSON 中不存在”。通过对输入的 JSON 和 Protobuf Message 定义，可以拿到这些丢失的信息，之后就可以利用 Metadata 将 Protobuf 对象与这些信息一起发给后端，后端得到的实际就是完整的数据了。这时只要再提供一个封装好的方法，将 Protobuf 对象与 Metadata 中的数据组合到一起，就能够方便地拿到与

Client 端输入的相同的数据。

从 Server 端到 Client 端，反之亦然。

实现该方案时，相对复杂一点的地方是“对比用户输入 JSON 与 Message 定义，拿到丢失的信息”，需要递归得利用反射来得到，但只要多花些时间，相信并不难。

与前一种方案相比，这个方案的优点是不定义新的基本类型。至于缺点，就是给不同服务间通过 gRPC 调用增加了额外的复杂度，每一个服务都需要使用一个专门的方法来调用其他服务的接口，其中封装了前面提到的对比和组合过程。

如何扩展 Server 端返回的 Error

在 gRPC 的 Server 端返回 Error 时，虽然接口的返回值中声明的是标准库的 Error，但 gRPC 内部会判断 Error 是否为特定的类型，如果不是，则会统一返回 Code 为“Unknown”的 Error。因此，多数情况下我们需要用专门的方法构造一个 Error，比如：

```
import "google.golang.org/grpc/status"
...
return status.Error(codes.Aborted, "aborted")
```

实际上，这个 Error 的类型是“[google.golang.org/genproto/googleapis/rpc/status](#)”包里面的“Status”，定义如下：

```
type Status struct {
    Code int32
    Message string
    Details []*google_protobuf.Any
}
```

除了 Details 字段外，只有 Code 和 Message，十分简洁，但如果我们要有更复杂的业务需求，这些字段是不能满足的，因此我们需要扩展 Error。

方案零：不要扩展 gRPC 预定义的 Status Code

前面的例子中，“codes.Aborted”是 gRPC 预定义的一些 Code 之一，自然而然，我们会想到是否可以扩展这个 Code 列表。

我曾经尝试过扩展这个列表，在 Go 语言的 gRPC 客户端中是可以拿到自定义的 Code 的，但官方的开发人员在一些问答中表示不推荐这么做，而且一些其他语言的 gRPC 客户端中，一旦发现 Code 不在预定义的列表中，有可能直接替换成预定义的“Unknown”错误，甚至直接抛出异常，因此不要这么做。

方案一：在 Response 中添加 err 属性

这也是一种显而易见的方案，既然已有的 Error 不能满足需求，那就在 Response 对象中加入一个“err”属性，而它的类型是自己定义的，大概是这个样子：

```
type MyResponse struct {
    err MyErr
    ...
}
```

可以满足需求，但很不优雅，同意么？这么做直接违反了 gRPC 的错误处理机制，甚至不符合 Go 语言的规范，所以也不推荐这么做。

方案二：通过 Metadata 传递

首先要说，这才是一个靠谱的方案，也是官方曾经推荐的方案。

先不说“曾经”是什么意思，我们来看看这个方案是怎么玩的。

gRPC 的 Client 端和 Server 端之间，可以借助名为“Metadata”的数据结构来传递额外的信息，而我们自己扩展的 Error 信息就属于这个“额外信息”。

详细的用法可以参考在 GitHub 上关于“gRPC-metadata”的文档：

<https://github.com/gRPC/gRPC-go/blob/master/Documentation/>

gRPC-metadata.md

此处附上其中的一些代码示例，以便让大家快速地获得一个直观的印象。

以从 Server 端向 Client 端传递 Metadata 为例，首先在 Server 端将数据准备好：

```
func (s *server) SomeRPC(ctx context.Context, in *pb.someRequest)
(*pb.someResponse, error) {
    // 创建并发送Header
    header := metadata.Pairs("header-key", "val")
    gRPC.SendHeader(ctx, header)
    // 创建并发送Trailer
    trailer := metadata.Pairs("trailer-key", "val")
    gRPC.SetTrailer(ctx, trailer)
}
```

接着在 Client 端接收数据：

```
var header, trailer metadata.MD // 用来保存header和trailer的变量
r, err := client.SomeRPC(
    ctx,
    someRequest,
    gRPC.Header(&header), // 将会接收header
    gRPC.Trailer(&trailer), // 将会接收trailer
)
```

// 按需求对header和trailer做处理

方案三：通过 Status 中的 Details 属性传递

最后出场的是我们实际采用的方案。

我在前面将扩展的 Error 信息称作“额外信息”，其实准确的说应该是“额外的 Error 信息”，因此，直觉上最自然的方式还是在 Error 对象内部携带这个信息。

你一定注意到了“Status”对象里的那个“Details”属性，它是一

个“Any”对象的数组，字面上看似乎是“可以保存任何类型对象的数组”的意思，确实是这样。

gRPC 最近刚刚添加了两个工具方法，使得“Details”属性变得非常易用：

```
// WithDetails返回一个新创建的Status对象，其中附加了参数details传入的
Message列表,
// 如果有error发生，将返回nil和第一个遇到的error。
func (s *Status) WithDetails(details ...proto.Message) (*Status,
error)

// Details返回Status中Details携带的Message列表,
// 如果decode某个Message时发生错误，那这个错误会被添加到结果列表中返回。
func (s *Status) Details() []interface{}
```

实际使用时很方便。

附加 Details:

```
s, _ := status.New(codes.Aborted, "message").WithDetails(d1, d2)
return nil, s.Err()
```

获取 Details:

```
details, _ := s.Details()
for _, d := range details {
    m := d.(YourType)
    // ...
}
```

好了，以上就是我们在使用 gRPC 过程中遇到的两个问题和相应的思考，也许并不是最优的方案，但希望能给你带来一些提示。

Go 语言作为一种快速发展变化中的语言，相应的技术生态还不是十分健全，包括 gRPC 和由此衍生的 gRPC-Gateway 等项目，仍然有不少提升空间。中国是目前 Go 语言应用人数最多、气氛最火热的国家，我们希望能看到越来越多的国内开发者参与到开源项目的发展中，也希望有越来越多的优秀项目出现。

版权声明

InfoQ 中文站出品

化茧成蝶：Go 在 FreeWheel 服务化中的实践

©2018 飞维美地信息技术（北京）有限公司

本书版权为飞维美地信息技术（北京）有限公司和北京极客邦科技有限公司共同所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营容和路叶青大厦（北园）五层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn