go-zero解读与最佳实践

Kevin@开源说

About me

万俊峰 / Kevin

- github @kevwan
- https://github.com/tal-tech/go-zero
- https://zero.gocn.vip

微服务实践

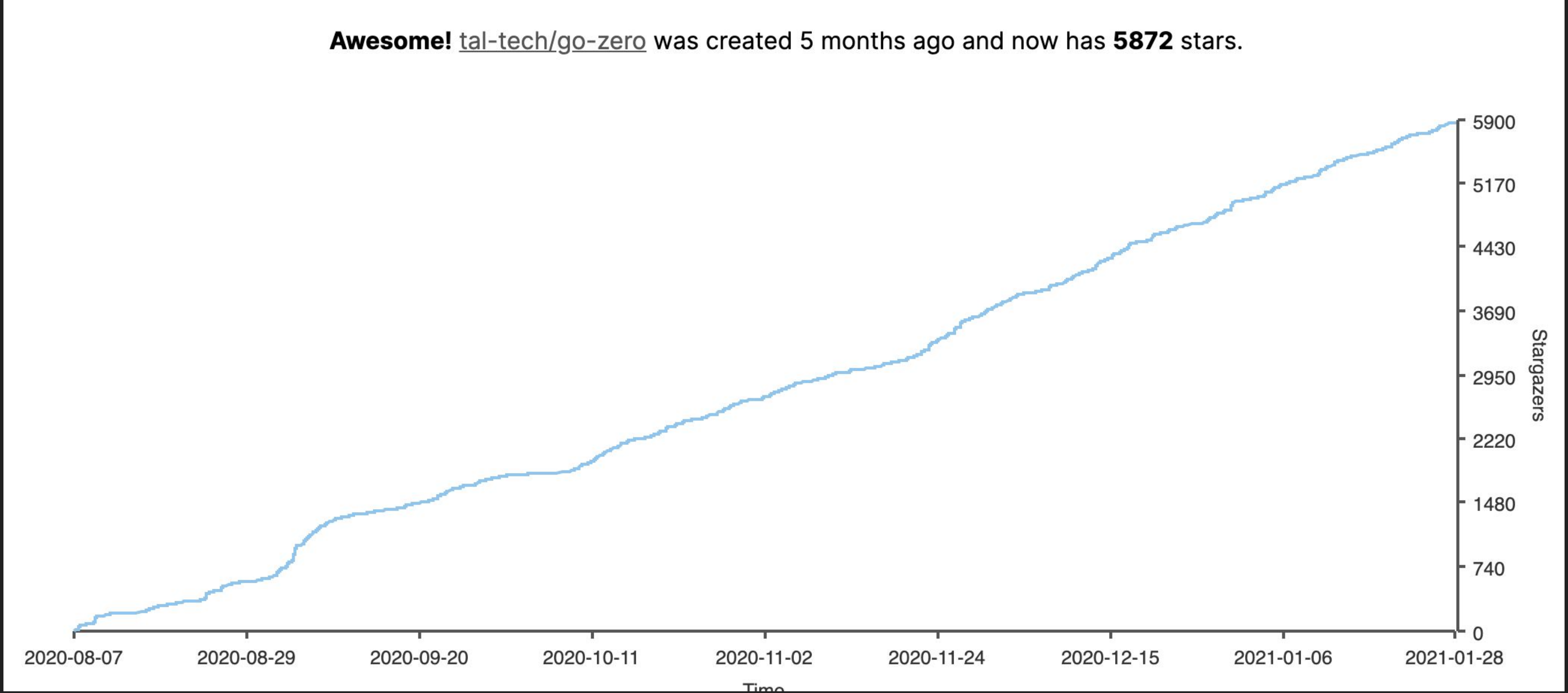
微信扫描二维码，关注我的公众号

go-zero社区⑦

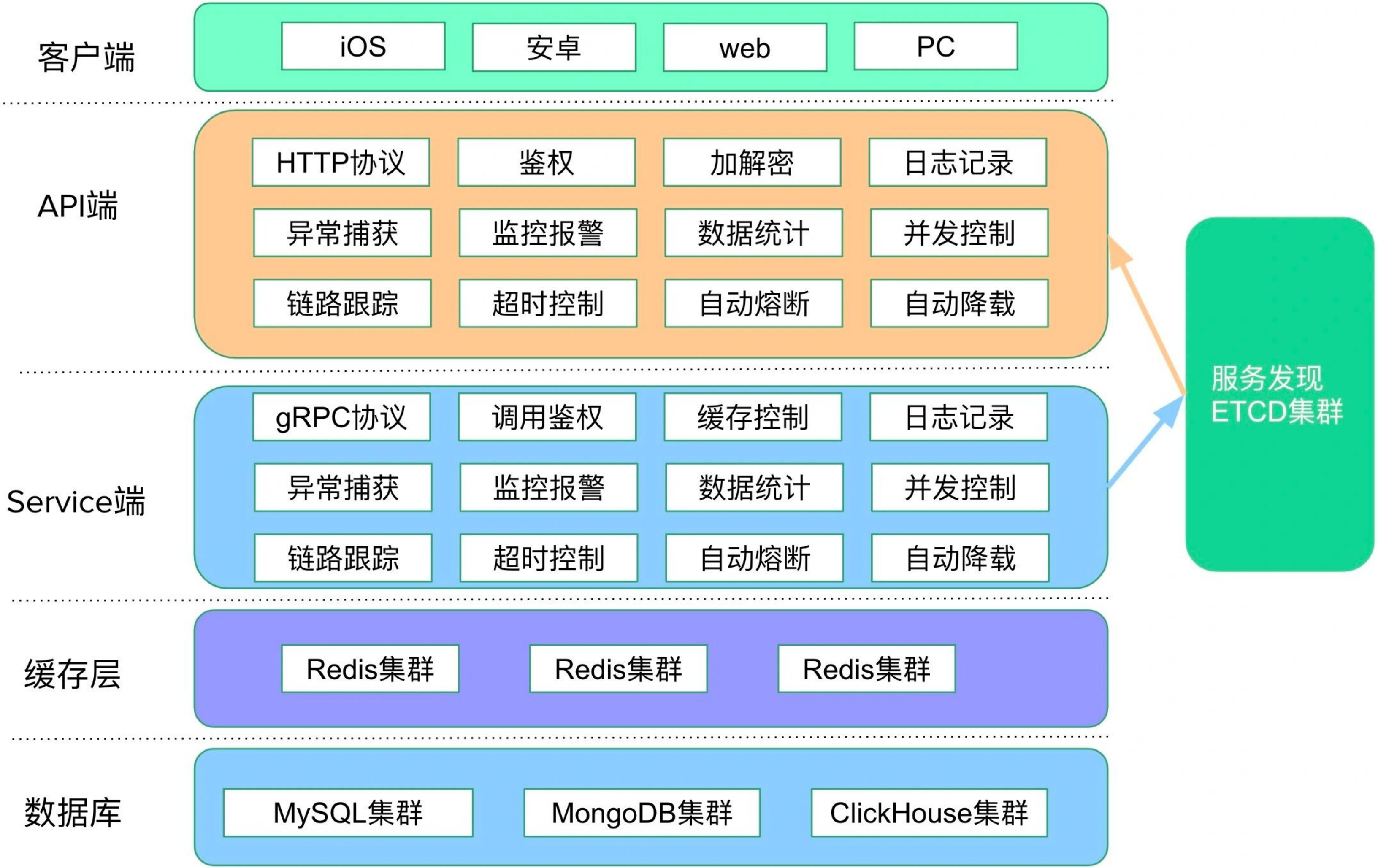
该二维码7天内(2月4日前)有效，重新进入将更新

go-zero概览

go-zero增长趋势



Awesome! tal-tech/go-zero was created 5 months ago and now has **5872** stars.



客户端：iOS　安卓　web　PC

API端：HTTP协议　鉴权　加解密　日志记录　异常捕获　监控报警　数据统计　并发控制　链路跟踪　超时控制　自动熔断　自动降载

Service端：gRPC协议　调用鉴权　缓存控制　日志记录　异常捕获　监控报警　数据统计　并发控制　链路跟踪　超时控制　自动熔断　自动降载

缓存层：Redis集群　Redis集群　Redis集群

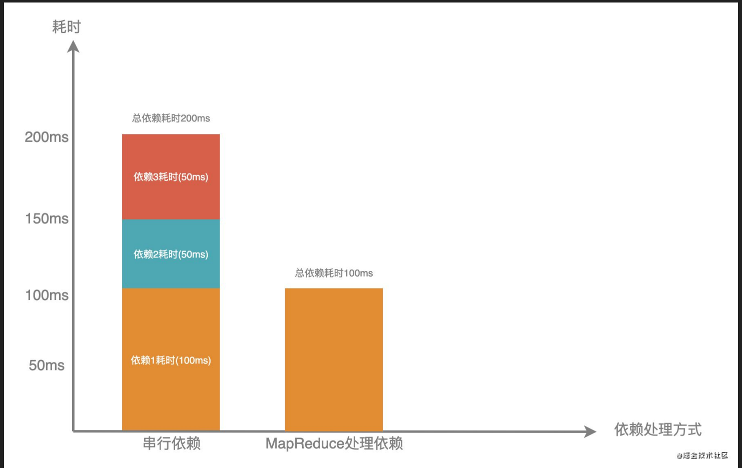
数据库：MySQL集群　MongoDB集群　ClickHouse集群

服务发现
ETCD集群

微服务系统设计中的痛点

微服务系统如何拆分？

- 先粗后细，切忌过细，切忌一个请求一个服务
- 横向拆分，而非纵向，我们一般不会超过三层
- 单向调用，严禁循环调用
- 禁止接口类型透传
- 没有依赖关系的串行调用改为并行调用 - core/mr

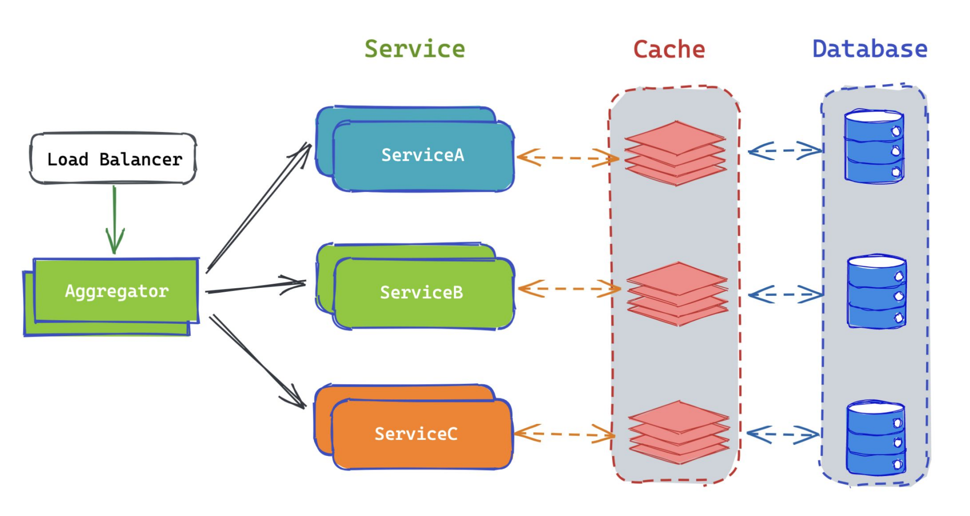


如何保障高并发高可用？

- 良好的数据边界
- 高效的缓存管理
- 优雅的熔断降载保护
- 弹性伸缩能力
- 清晰的资源使用定义
- 高效的监控报警

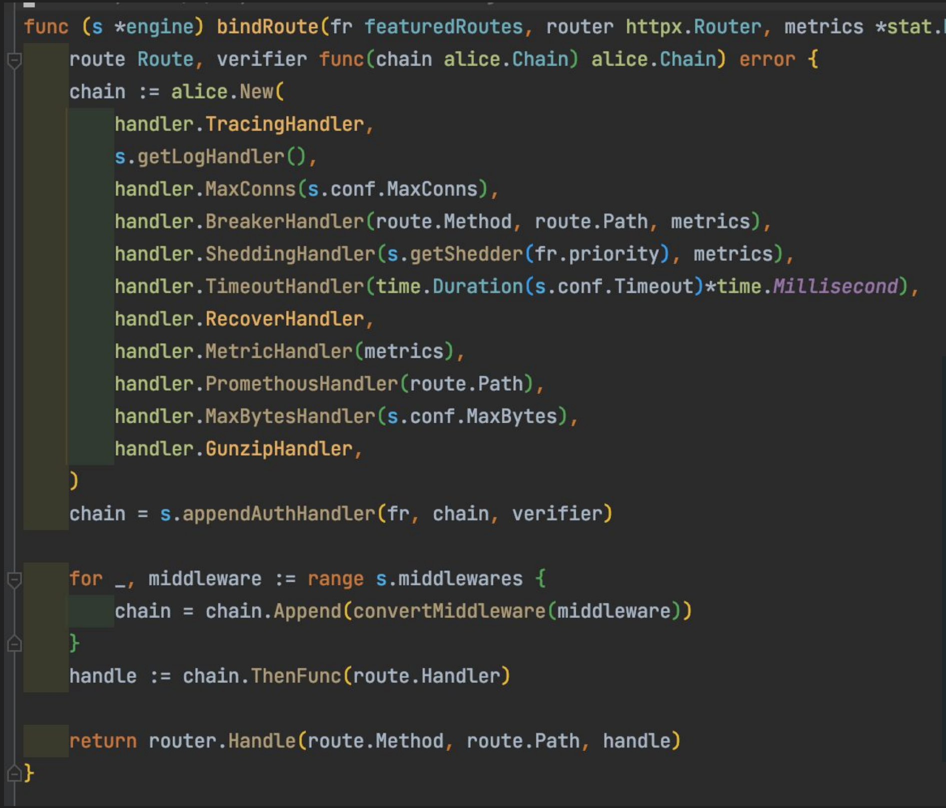
大型微服务项目从何下手？

- 从单体服务开始
- 业务优先，技术支撑
- 服务指标监控
- 数据拆分+缓存管理
- 服务拆分
- 支撑系统建设
- 自动化+工程建设

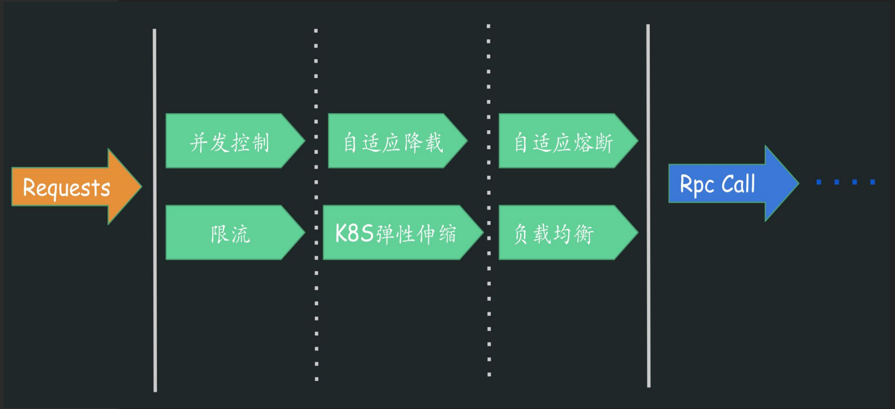


go-zero部分组件剖析

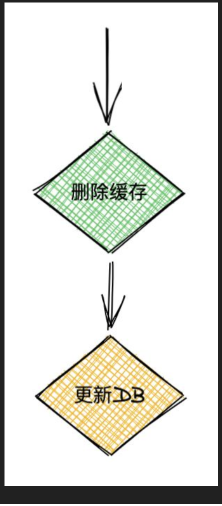
go-zero如何保障高并发、高可用

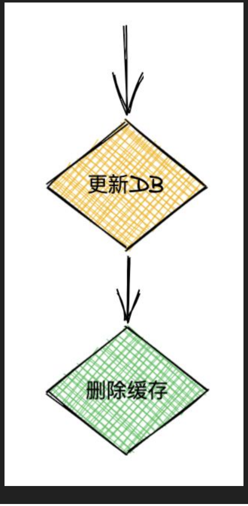


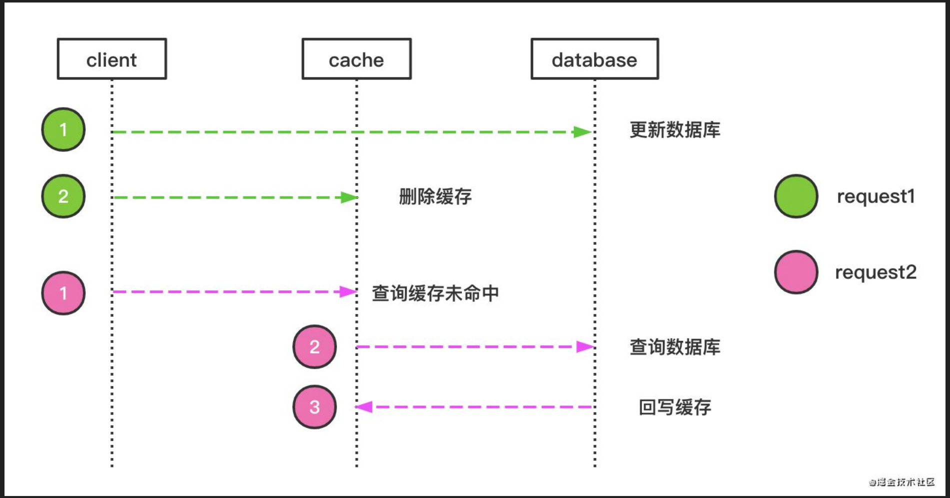
```go
func (s *engine) bindRoute(fr featuredRoutes, router httpx.Router, metrics *stat.Metrics,
    route Route, verifier func(chain alice.Chain) alice.Chain) error {
    chain := alice.New(
        handler.TracingHandler,
        s.getLogHandler(),
        handler.MaxConns(s.conf.MaxConns),
        handler.BreakerHandler(route.Method, route.Path, metrics),
        handler.SheddingHandler(s.getShedder(fr.priority), metrics),
        handler.TimeoutHandler(time.Duration(s.conf.Timeout)*time.Millisecond),
        handler.RecoverHandler,
        handler.MetricHandler(metrics),
        handler.PromethousHandler(route.Path),
        handler.MaxBytesHandler(s.conf.MaxBytes),
        handler.GunzipHandler,
    )
    chain = s.appendAuthHandler(fr, chain, verifier)

    for _, middleware := range s.middlewares {
        chain = chain.Append(convertMiddleware(middleware))
    }
    handle := chain.ThenFunc(route.Handler)

    return router.Handle(route.Method, route.Path, handle)
}
```

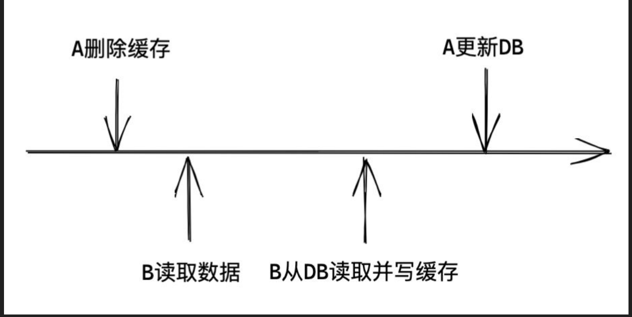


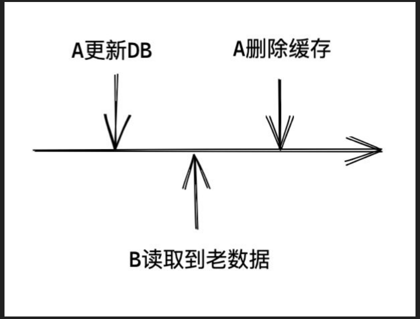
go-zero如何自动管理缓存



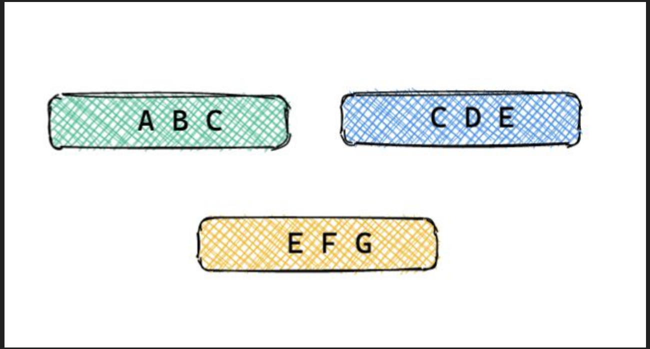


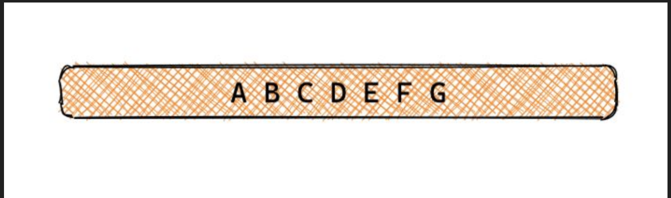


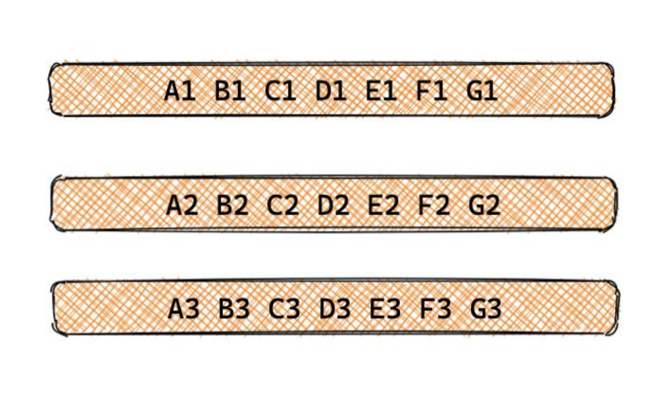




go-zero如何自动管理缓存 - 查询种类

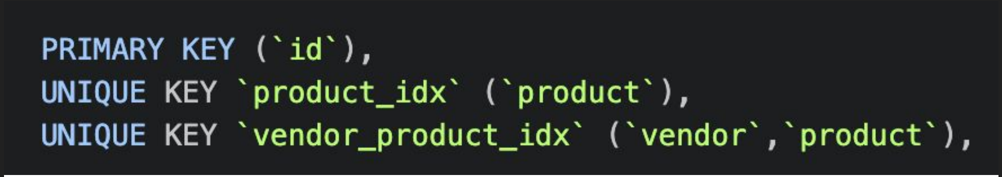




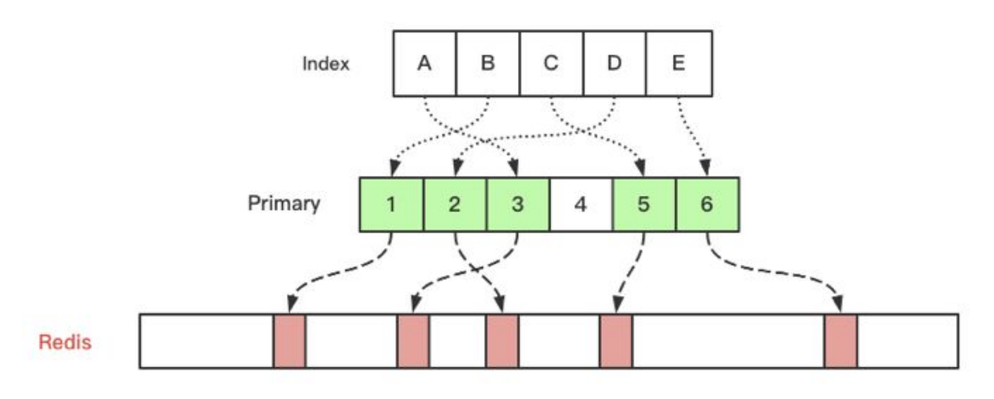


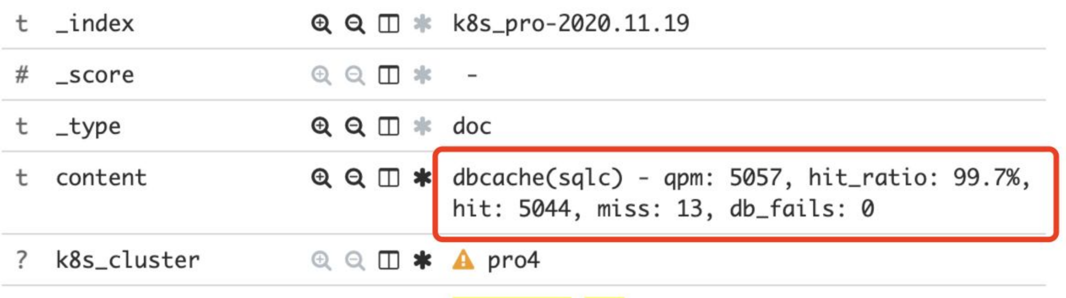
go-zero如何自动管理缓存 - 续

- 基于主键的缓存
- 基于唯一索引的缓存
- 基于组合唯一索引的缓存
- 缓存部分还是完整行记录
- 数据一致性
- 击穿、穿透、雪崩
- 缓存访问量、命中率统计



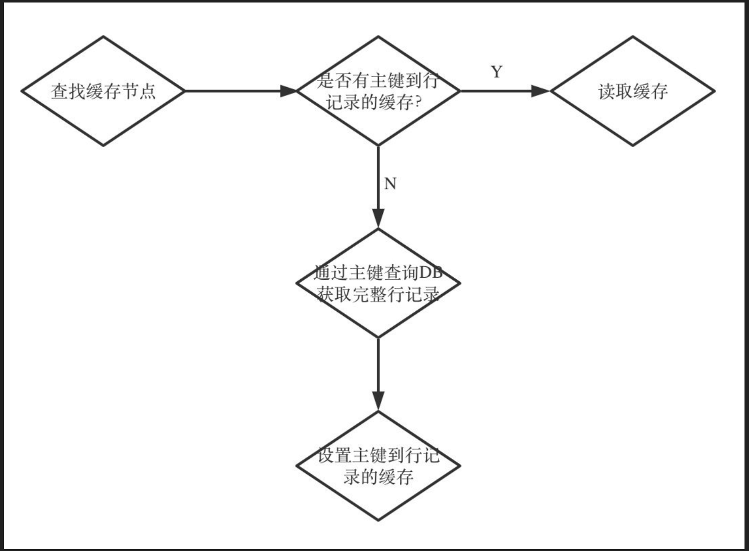
```
PRIMARY KEY (`id`),
UNIQUE KEY `product_idx` (`product`),
UNIQUE KEY `vendor_product_idx` (`vendor`,`product`),
```





t	_index	k8s_pro-2020.11.19
#	_score	-
t	_type	doc
t	content	dbcache(sqlc) - qpm: 5057, hit_ratio: 99.7%, hit: 5044, miss: 13, db_fails: 0
?	k8s_cluster	⚠ pro4

基于主键的缓存自动管理



查找缓存节点

是否有主键到行记录的缓存?

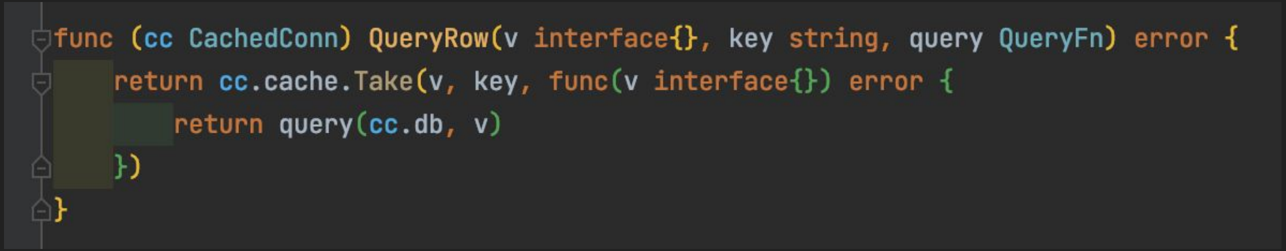
Y

读取缓存

N

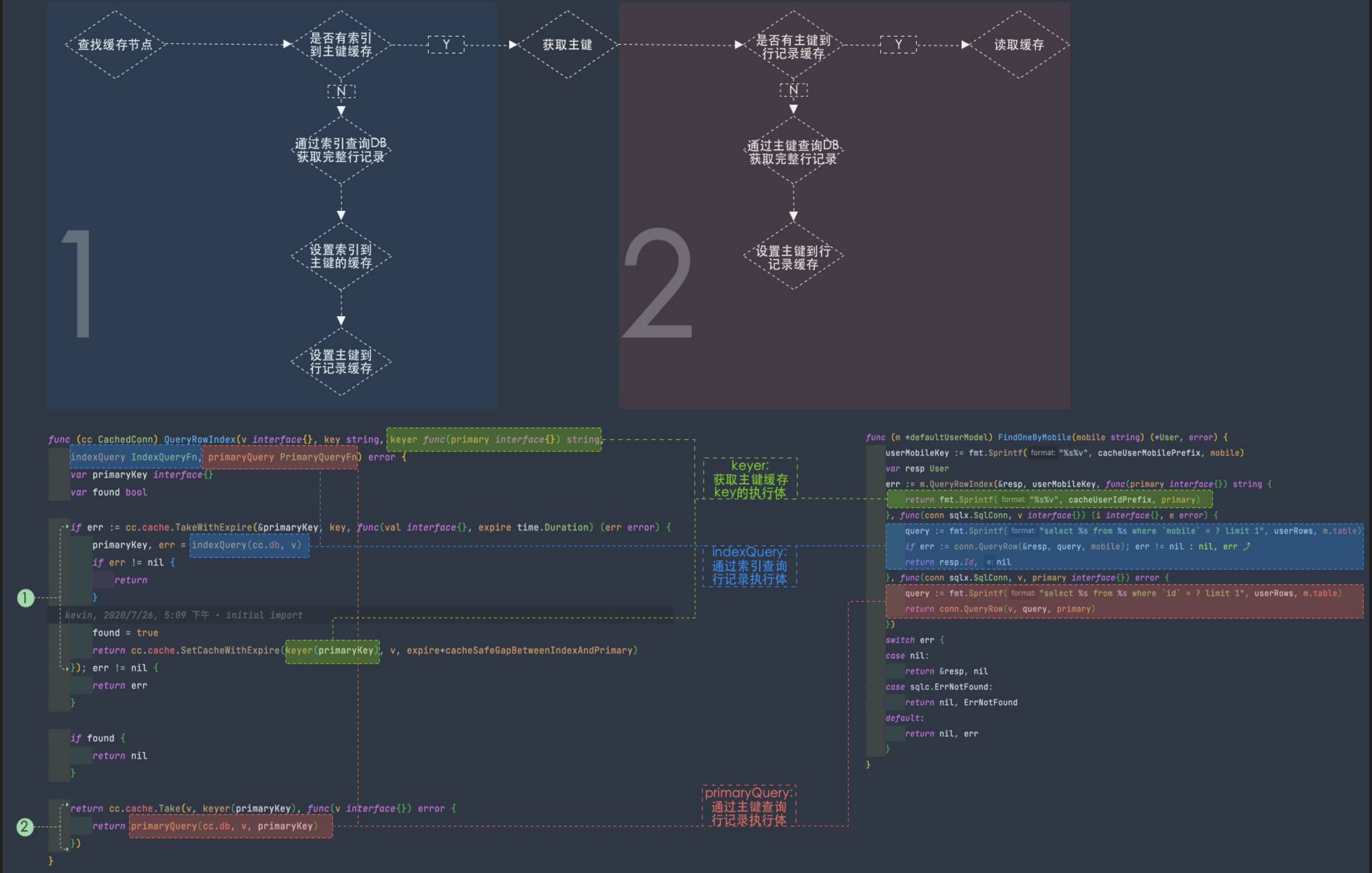
通过主键查询DB获取完整行记录

设置主键到行记录的缓存



```
func (cc CachedConn) QueryRow(v interface{}, key string, query QueryFn) error {
	return cc.cache.Take(v, key, func(v interface{}) error {
		return query(cc.db, v)
	})
}
```

基于索引的缓存自动管理



```go
func (cc CachedConn) QueryRowIndex(v interface{}, key string, keyer func(primary interface{}) string,
	indexQuery IndexQueryFn, primaryQuery PrimaryQueryFn) error {
	var primaryKey interface{}
	var found bool

	if err := cc.cache.TakeWithExpire(&primaryKey, key, func(val interface{}, expire time.Duration) (err error) {
		primaryKey, err = indexQuery(cc.db, v)
		if err != nil {
			return
		}
kevin, 2020/7/26, 5:09 下午 • initial import
		found = true
		return cc.cache.SetCacheWithExpire(keyer(primaryKey), v, expire+cacheSafeGapBetweenIndexAndPrimary)
	}); err != nil {
		return err
	}

	if found {
		return nil
	}

	return cc.cache.Take(v, keyer(primaryKey), func(v interface{}) error {
		return primaryQuery(cc.db, v, primaryKey)
	})
}
```

keyer:
获取主键缓存key的执行体

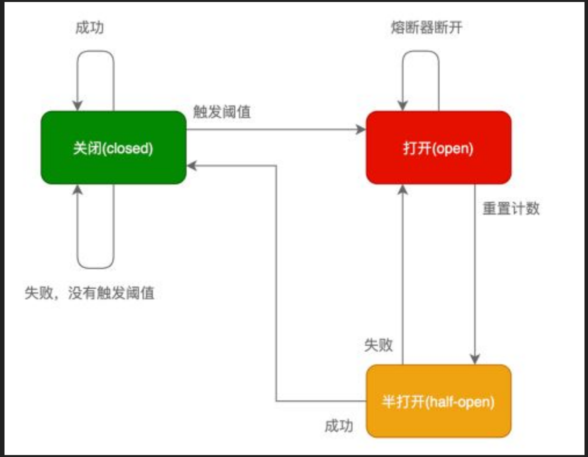
indexQuery:
通过索引查询行记录执行体

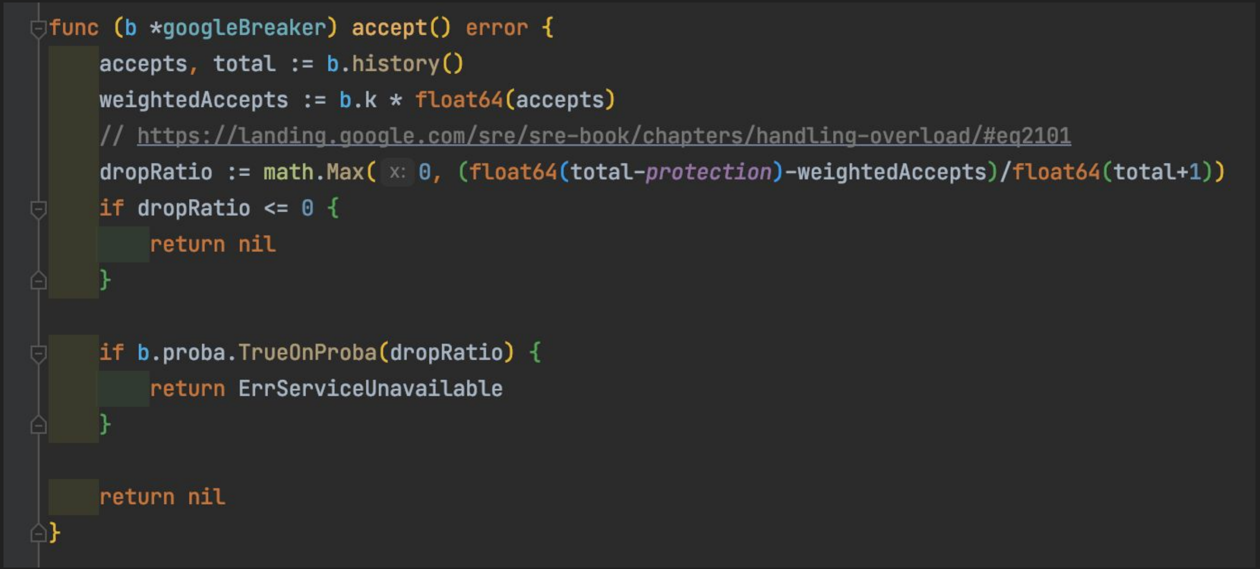
primaryQuery:
通过主键查询行记录执行体

```go
func (m *defaultUserModel) FindOneByMobile(mobile string) (*User, error) {
	userMobileKey := fmt.Sprintf( format: "%s%v", cacheUserMobilePrefix, mobile)
	var resp User
	err := m.QueryRowIndex(&resp, userMobileKey, func(primary interface{}) string {
		return fmt.Sprintf( format: "%s%v", cacheUserIdPrefix, primary)
	}, func(conn sqlx.SqlConn, v interface{}) (i interface{}, e error) {
		query := fmt.Sprintf( format: "select %s from %s where `mobile` = ? limit 1", userRows, m.table)
		if err := conn.QueryRow(&resp, query, mobile); err != nil : nil, err ⤴
		return resp.Id,  e: nil
	}, func(conn sqlx.SqlConn, v, primary interface{}) error {
		query := fmt.Sprintf( format: "select %s from %s where `id` = ? limit 1", userRows, m.table)
		return conn.QueryRow(v, query, primary)
	})
	switch err {
	case nil:
		return &resp, nil
	case sqlc.ErrNotFound:
		return nil, ErrNotFound
	default:
		return nil, err
	}
}
```

go-zero的自适应熔断算法

- 基于滑动窗口（10秒/40窗口）
- 支持自定义触发条件
- 支持自定义fallback
- http/rpc框架内建
- 自动触发，自动恢复



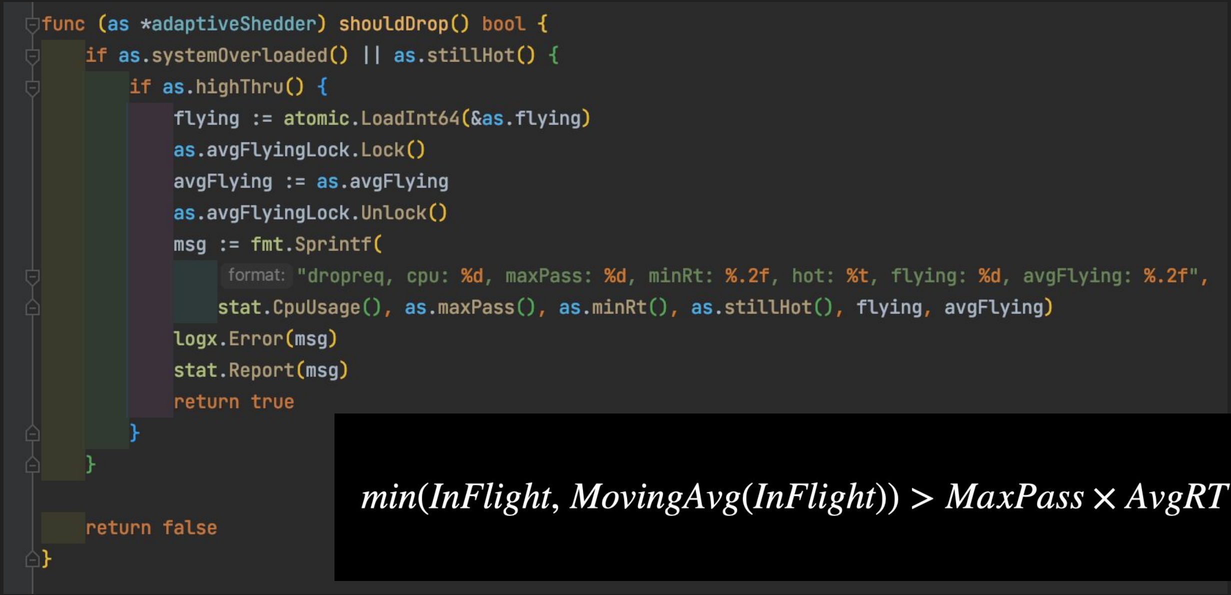


```go
func (b *googleBreaker) accept() error {
	accepts, total := b.history()
	weightedAccepts := b.k * float64(accepts)
	// https://landing.google.com/sre/sre-book/chapters/handling-overload/#eq2101
	dropRatio := math.Max( x: 0, (float64(total-protection)-weightedAccepts)/float64(total+1))
	if dropRatio <= 0 {
		return nil
	}

	if b.proba.TrueOnProba(dropRatio) {
		return ErrServiceUnavailable
	}

	return nil
}
```

$$dropRatio = max(0, \frac{(requests - protection) - K \times accepts}{requests + 1})$$

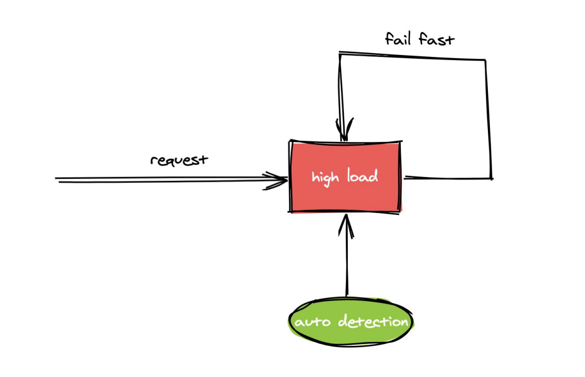
go-zero的服务过载保护

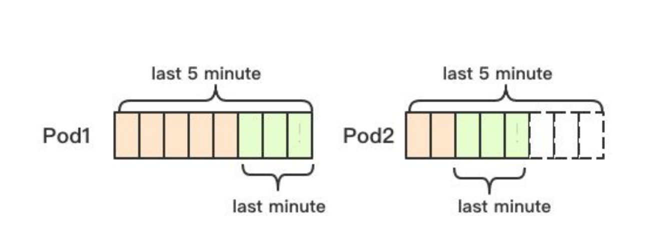
- CPU负载采样
- 冷却期
- 当前是否请求过量
- 分级降载



```go
func (as *adaptiveShedder) shouldDrop() bool {
	if as.systemOverloaded() || as.stillHot() {
		if as.highThru() {
			flying := atomic.LoadInt64(&as.flying)
			as.avgFlyingLock.Lock()
			avgFlying := as.avgFlying
			as.avgFlyingLock.Unlock()
			msg := fmt.Sprintf(
				format: "dropreq, cpu: %d, maxPass: %d, minRt: %.2f, hot: %t, flying: %d, avgFlying: %.2f",
				stat.CpuUsage(), as.maxPass(), as.minRt(), as.stillHot(), flying, avgFlying)
			logx.Error(msg)
			stat.Report(msg)
			return true
		}
	}

	return false
}
```

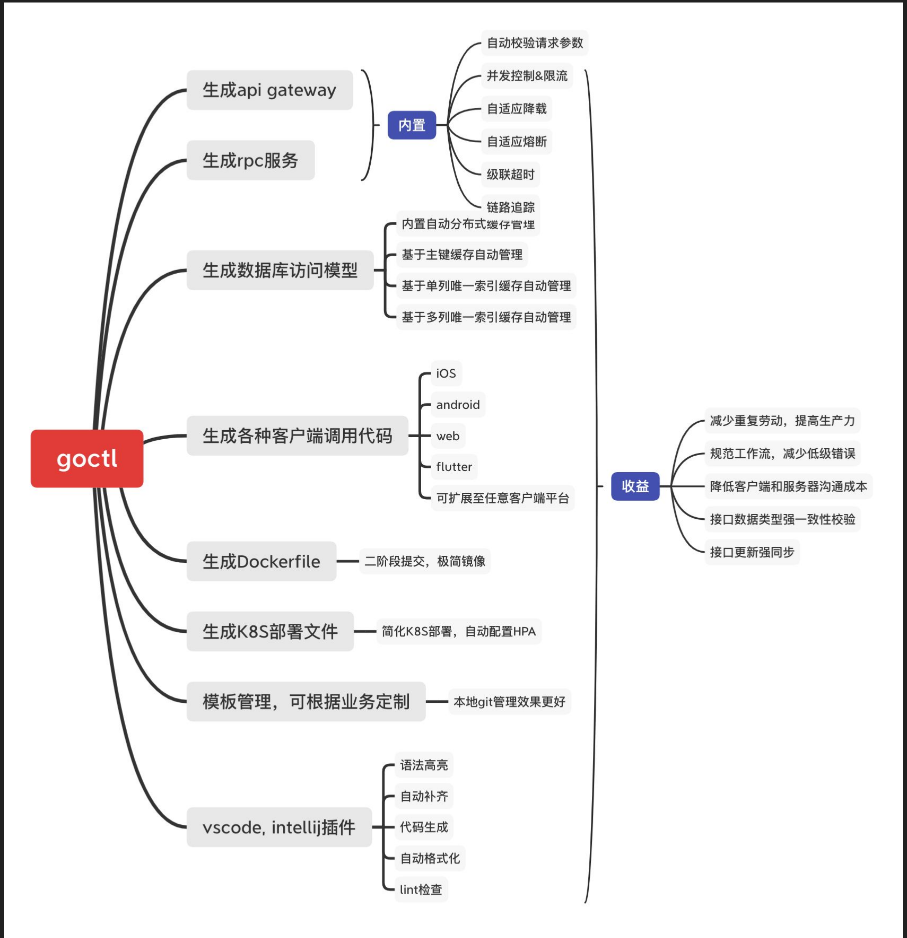
$$min(InFlight, MovingAvg(InFlight)) > MaxPass \times AvgRT$$





go-zero最佳实践

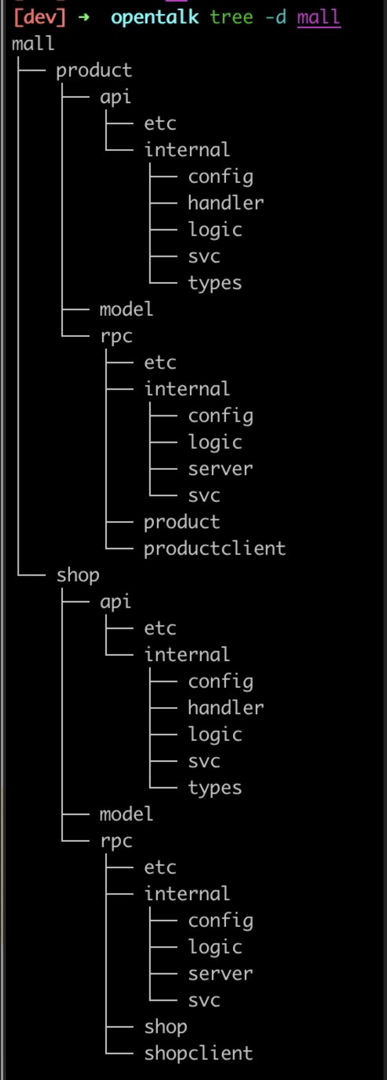
goctl解决了哪些问题？



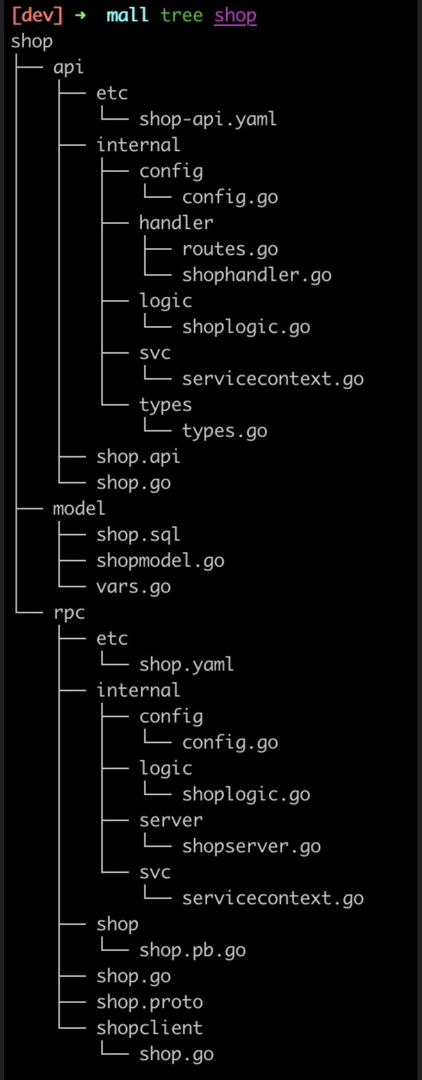
- goctl
  - 生成api gateway
  - 生成rpc服务
    - 内置
      - 自动校验请求参数
      - 并发控制&限流
      - 自适应降载
      - 自适应熔断
      - 级联超时
      - 链路追踪
  - 生成数据库访问模型
    - 内置自动分布式缓存管理
    - 基于主键缓存自动管理
    - 基于单列唯一索引缓存自动管理
    - 基于多列唯一索引缓存自动管理
  - 生成各种客户端调用代码
    - iOS
    - android
    - web
    - flutter
    - 可扩展至任意客户端平台
  - 生成Dockerfile
    - 二阶段提交，极简镜像
  - 生成K8S部署文件
    - 简化K8S部署，自动配置HPA
  - 模板管理，可根据业务定制
    - 本地git管理效果更好
  - vscode, intellij插件
    - 语法高亮
    - 自动补齐
    - 代码生成
    - 自动格式化
    - lint检查
- 收益
  - 减少重复劳动，提高生产力
  - 规范工作流，减少低级错误
  - 降低客户端和服务器沟通成本
  - 接口数据类型强一致性校验
  - 接口更新强同步

go-zero最佳实践

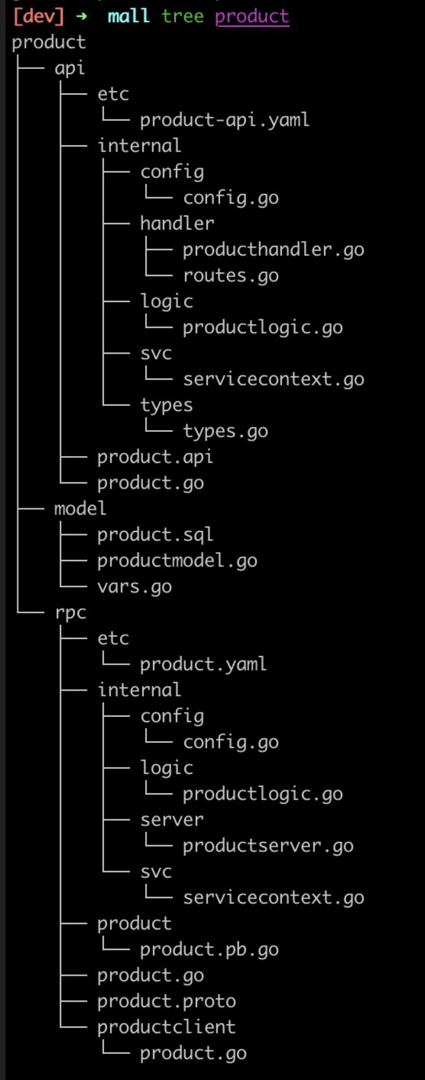
- mono repo
- 项目结构
- api单独repo
- goctl一键生成



```
[dev] ➜  opentalk tree -d mall
mall
├── product
│   ├── api
│   │   ├── etc
│   │   └── internal
│   │       ├── config
│   │       ├── handler
│   │       ├── logic
│   │       ├── svc
│   │       └── types
│   ├── model
│   └── rpc
│       ├── etc
│       ├── internal
│       │   ├── config
│       │   ├── logic
│       │   ├── server
│       │   └── svc
│       ├── product
│       └── productclient
└── shop
    ├── api
    │   ├── etc
    │   └── internal
    │       ├── config
    │       ├── handler
    │       ├── logic
    │       ├── svc
    │       └── types
    ├── model
    └── rpc
        ├── etc
        ├── internal
        │   ├── config
        │   ├── logic
        │   ├── server
        │   └── svc
        ├── shop
        └── shopclient
```



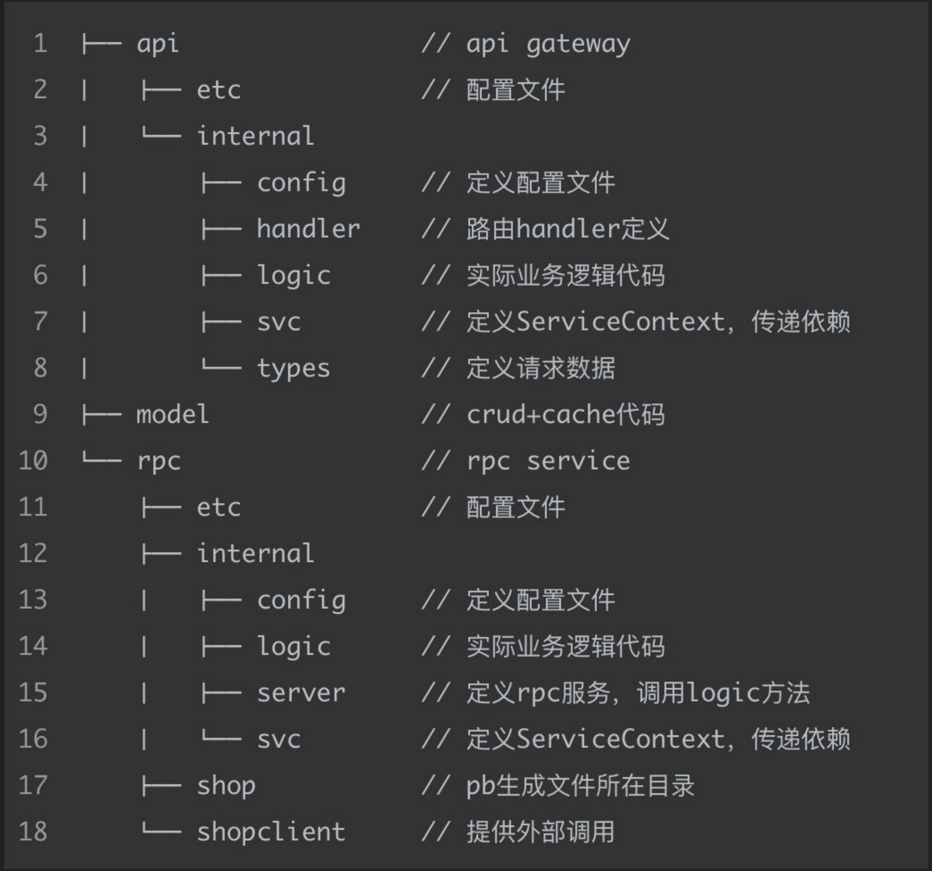
```
[dev] ➜  mall tree shop
shop
├── api
│   ├── etc
│   │   └── shop-api.yaml
│   ├── internal
│   │   ├── config
│   │   │   └── config.go
│   │   ├── handler
│   │   │   ├── routes.go
│   │   │   └── shophandler.go
│   │   ├── logic
│   │   │   └── shoplogic.go
│   │   ├── svc
│   │   │   └── servicecontext.go
│   │   └── types
│   │       └── types.go
│   ├── shop.api
│   └── shop.go
├── model
│   ├── shop.sql
│   ├── shopmodel.go
│   └── vars.go
└── rpc
    ├── etc
    │   └── shop.yaml
    ├── internal
    │   ├── config
    │   │   └── config.go
    │   ├── logic
    │   │   └── shoplogic.go
    │   ├── server
    │   │   └── shopserver.go
    │   └── svc
    │       └── servicecontext.go
    ├── shop
    │   └── shop.pb.go
    ├── shop.go
    ├── shop.proto
    └── shopclient
        └── shop.go
```



```
[dev] ➜  mall tree product
product
├── api
│   ├── etc
│   │   └── product-api.yaml
│   ├── internal
│   │   ├── config
│   │   │   └── config.go
│   │   ├── handler
│   │   │   ├── producthandler.go
│   │   │   └── routes.go
│   │   ├── logic
│   │   │   └── productlogic.go
│   │   ├── svc
│   │   │   └── servicecontext.go
│   │   └── types
│   │       └── types.go
│   ├── product.api
│   └── product.go
├── model
│   ├── product.sql
│   ├── productmodel.go
│   └── vars.go
└── rpc
    ├── etc
    │   └── product.yaml
    ├── internal
    │   ├── config
    │   │   └── config.go
    │   ├── logic
    │   │   └── productlogic.go
    │   ├── server
    │   │   └── productserver.go
    │   └── svc
    │       └── servicecontext.go
    ├── product
    │   └── product.pb.go
    ├── product.go
    ├── product.proto
    └── productclient
        └── product.go
```

go-zero最佳实践

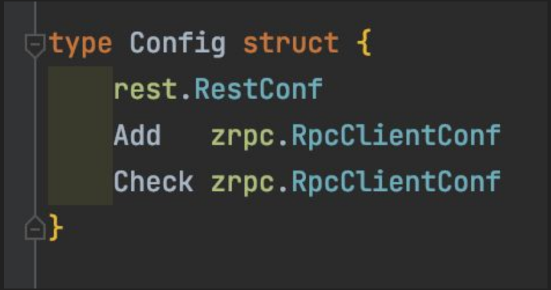
- 不需要修改的顶部标注为DO NOT EDIT
- api单独repo
- goctl一键生成



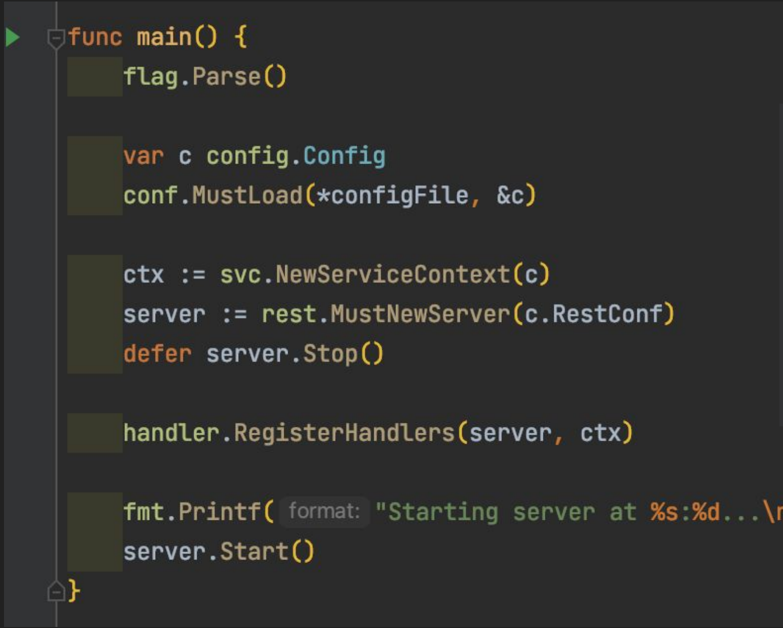
```
├── api                     // api gateway
|   ├── etc                 // 配置文件
|   └── internal
|       ├── config          // 定义配置文件
|       ├── handler         // 路由handler定义
|       ├── logic           // 实际业务逻辑代码
|       ├── svc             // 定义ServiceContext，传递依赖
|       └── types           // 定义请求数据
├── model                   // crud+cache代码
└── rpc                     // rpc service
    ├── etc                 // 配置文件
    ├── internal
    |   ├── config          // 定义配置文件
    |   ├── logic           // 实际业务逻辑代码
    |   ├── server          // 定义rpc服务，调用logic方法
    |   └── svc             // 定义ServiceContext，传递依赖
    ├── shop                // pb生成文件所在目录
    └── shopclient          // 提供外部调用
```

go-zero最佳实践 - 初始化

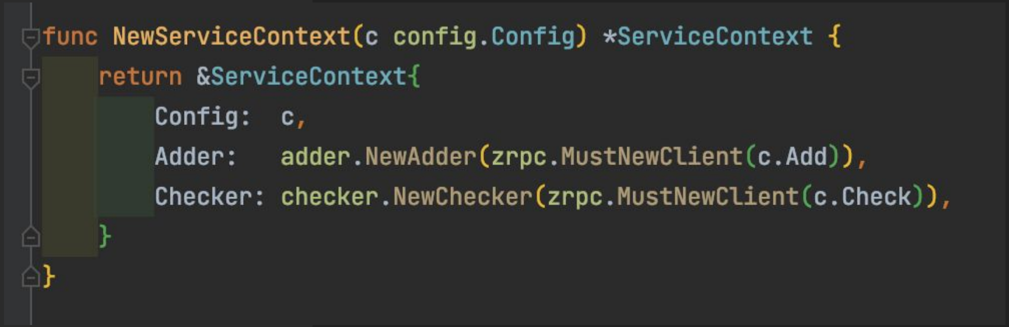
- 依赖初始化
- 同步初始化vs惰性初始化



```go
type Config struct {
	rest.RestConf
	Add   zrpc.RpcClientConf
	Check zrpc.RpcClientConf
}
```

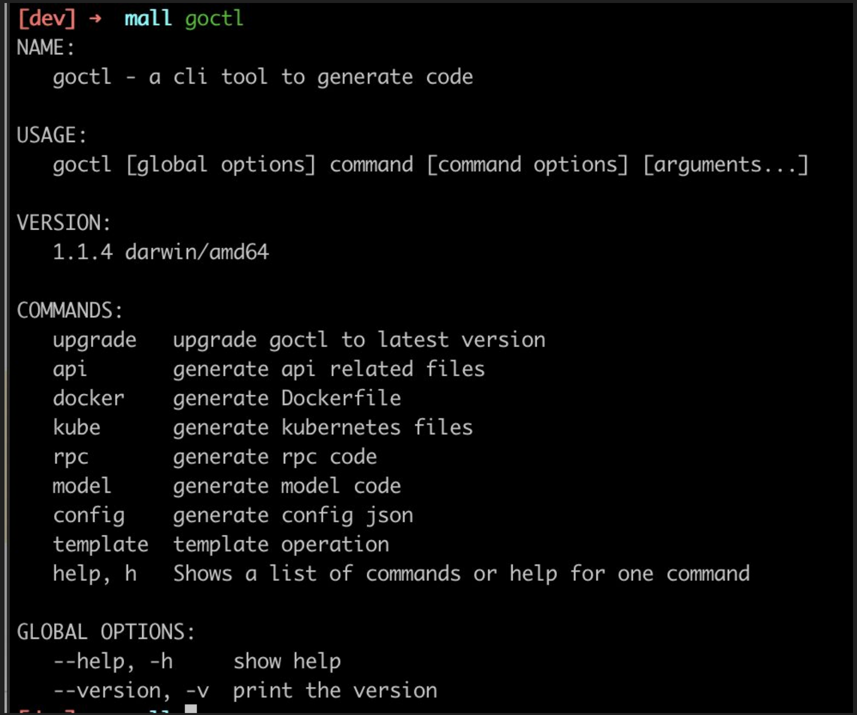


```go
func main() {
	flag.Parse()

	var c config.Config
	conf.MustLoad(*configFile, &c)

	ctx := svc.NewServiceContext(c)
	server := rest.MustNewServer(c.RestConf)
	defer server.Stop()

	handler.RegisterHandlers(server, ctx)

	fmt.Printf( format: "Starting server at %s:%d...\n", c.Host, c.Port)
	server.Start()
}
```

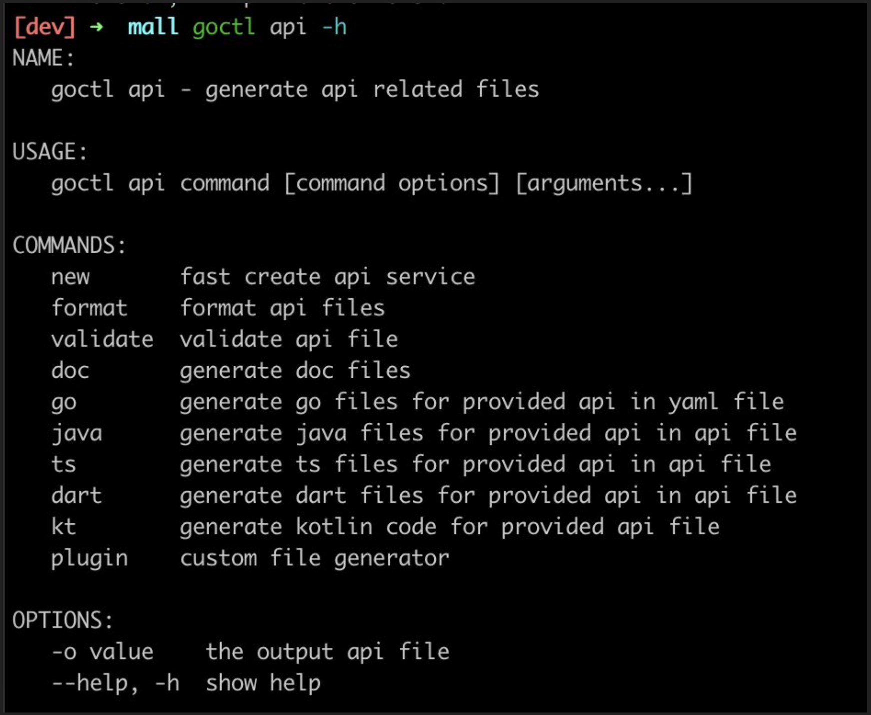


```go
func NewServiceContext(c config.Config) *ServiceContext {
	return &ServiceContext{
		Config:  c,
		Adder:   adder.NewAdder(zrpc.MustNewClient(c.Add)),
		Checker: checker.NewChecker(zrpc.MustNewClient(c.Check)),
	}
}
```

goctl

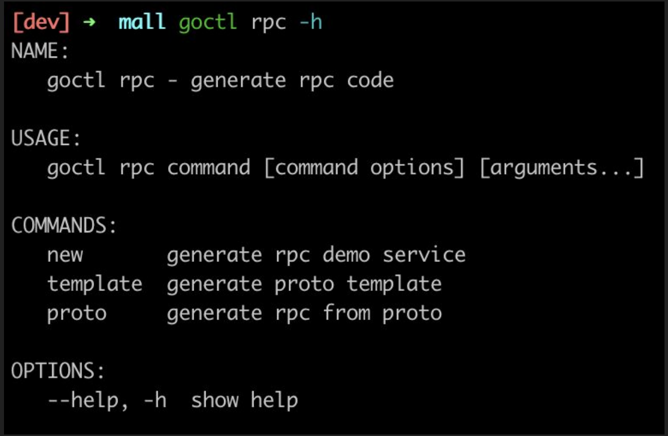


```
[dev] ➜  mall goctl
NAME:
   goctl - a cli tool to generate code

USAGE:
   goctl [global options] command [command options] [arguments...]

VERSION:
   1.1.4 darwin/amd64

COMMANDS:
   upgrade   upgrade goctl to latest version
   api       generate api related files
   docker    generate Dockerfile
   kube      generate kubernetes files
   rpc       generate rpc code
   model     generate model code
   config    generate config json
   template  template operation
   help, h   Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

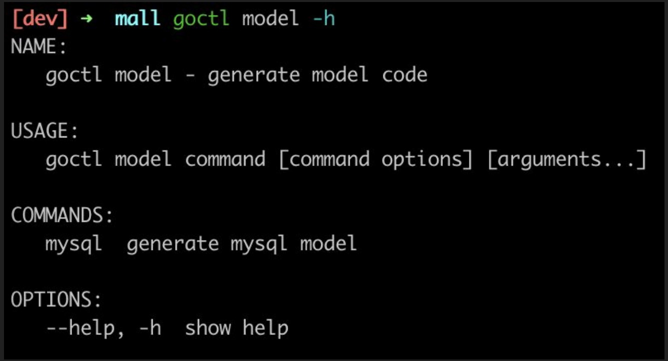


```
[dev] ➜  mall goctl api -h
NAME:
   goctl api - generate api related files

USAGE:
   goctl api command [command options] [arguments...]

COMMANDS:
   new       fast create api service
   format    format api files
   validate  validate api file
   doc       generate doc files
   go        generate go files for provided api in yaml file
   java      generate java files for provided api in api file
   ts        generate ts files for provided api in api file
   dart      generate dart files for provided api in api file
   kt        generate kotlin code for provided api file
   plugin    custom file generator

OPTIONS:
   -o value    the output api file
   --help, -h  show help
```

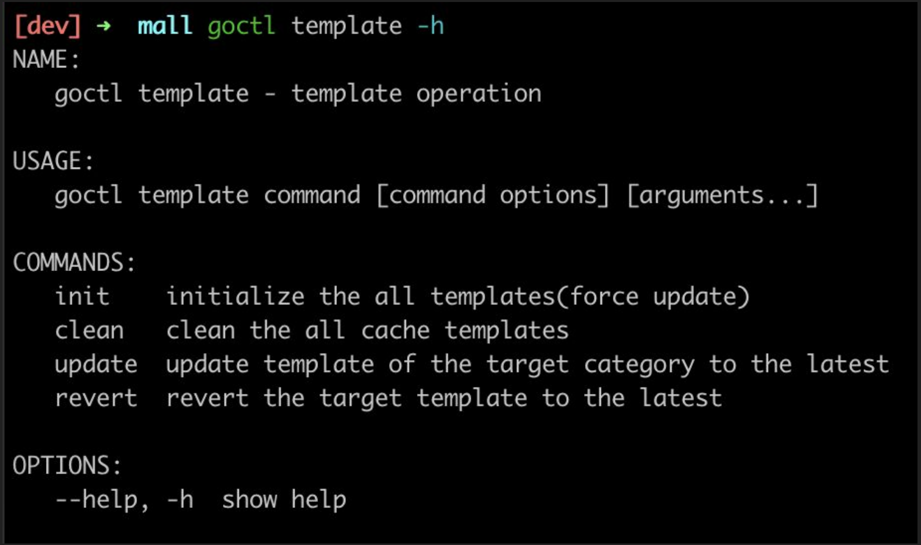
goctl



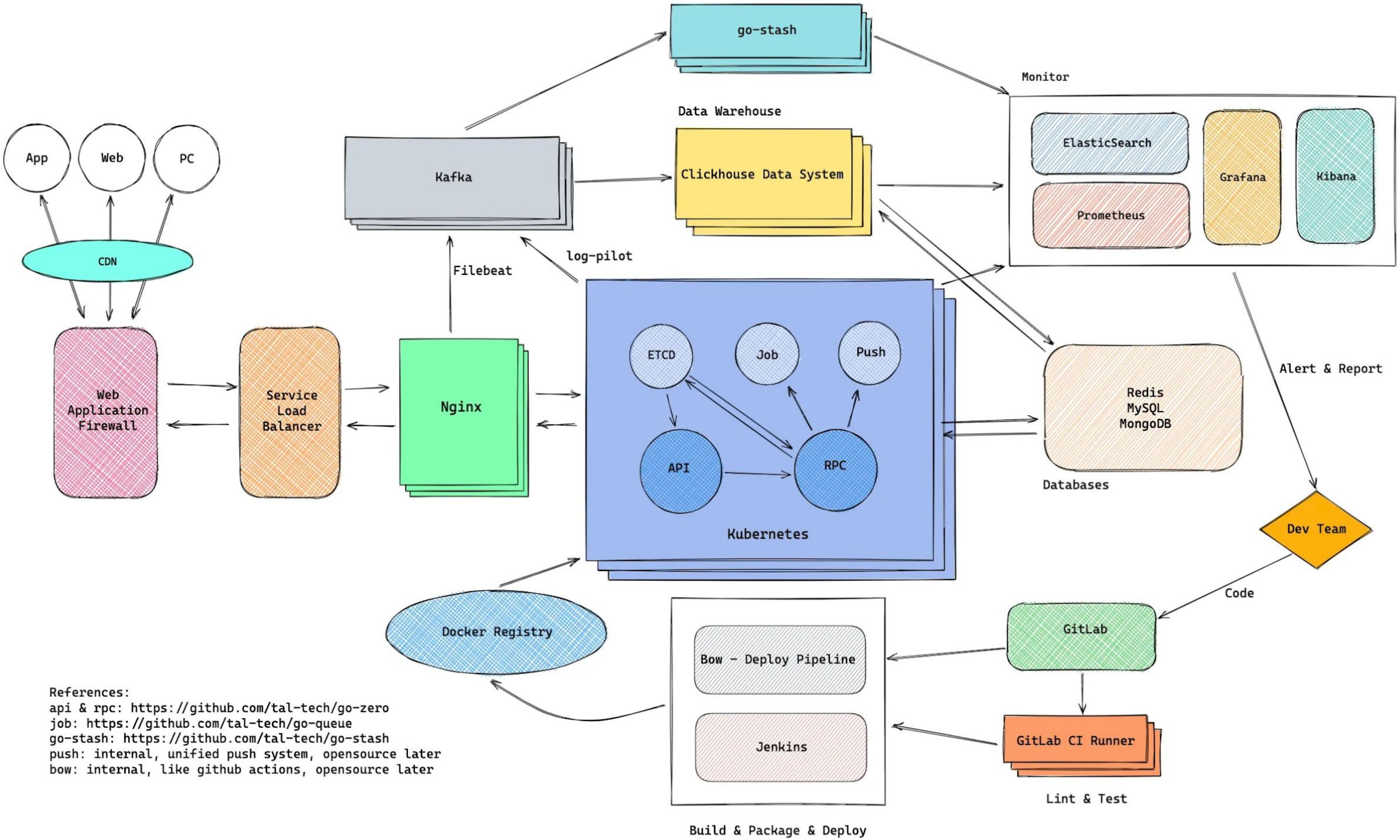
```
[dev] ➜  mall goctl rpc -h
NAME:
   goctl rpc - generate rpc code

USAGE:
   goctl rpc command [command options] [arguments...]

COMMANDS:
   new       generate rpc demo service
   template  generate proto template
   proto     generate rpc from proto

OPTIONS:
   --help, -h  show help
```



```
[dev] ➜  mall goctl model -h
NAME:
   goctl model - generate model code

USAGE:
   goctl model command [command options] [arguments...]

COMMANDS:
   mysql  generate mysql model

OPTIONS:
   --help, -h  show help
```



```
[dev] ➜  mall goctl template -h
NAME:
   goctl template - template operation

USAGE:
   goctl template command [command options] [arguments...]

COMMANDS:
   init    initialize the all templates(force update)
   clean   clean the all cache templates
   update  update template of the target category to the latest
   revert  revert the target template to the latest

OPTIONS:
   --help, -h  show help
```



References:
api & rpc: https://github.com/tal-tech/go-zero
job: https://github.com/tal-tech/go-queue
go-stash: https://github.com/tal-tech/go-stash
push: internal, unified push system, opensource later
bow: internal, like github actions, opensource later

谢谢！
star支持一下？

Kevin Wan

中国大陆

扫一扫上面的二维码图案，加我微信

微服务实践

微信扫描二维码，关注我的公众号

go-zero社区⑦

该二维码7天内(2月4日前)有效，重新进入将更新