【EV】Golang最佳实践（一）

* 这个文档并不是编码规范，想看规范的同学可以移步：​[Golang编码规范](https://bytedance.feishu.cn/space/doc/doccnz0IicI5WDMNioHawDjlt2a" \t "/Users/yangwenshuo/Documents\\x/_blank)​ ​[Go项目编码规范 V2.0](https://bytedance.feishu.cn/space/doc/wgrVe5aKfIIsUpTIttZL0b" \t "/Users/yangwenshuo/Documents\\x/_blank)​ 等文档；也不是高级语法技巧，想看高级编码方法的请移步：​[Go并发编程研讨课.pdf](https://bytedance.feishu.cn/space/file/boxcniaWo4NwBKmrmjI6Lq" \t "/Users/yangwenshuo/Documents\\x/_blank)​ 《[Go语言圣经](https://books.studygolang.com/gopl-zh/" \t "/Users/yangwenshuo/Documents\\x/_blank)》《[Go语言高级编程](https://chai2010.cn/advanced-go-programming-book/" \t "/Users/yangwenshuo/Documents\\x/_blank)》

这是一份是基于目前EV的Go项目中一些代码，总结出一些共同存在的点，来探索一些更好的编码方法。一是面向刚学习Go没多久，缺乏实践经验的同学，希望能对大家有所启发；二是总结一些比较好的习惯，统一EV项目组的代码风格。

代码组织1 - RPC调用的组织

几个名词

* 调用：一次rpc调用
* 打包：将rpc的返回经过一定的逻辑，拼装成最终response的过程

在观察feed\_api和mine\_api的代码时，发现整体还是一个完全过程化的组织方式，没有分层，也没有编写原则，一个功能来了之后，发现添加在好多地方都可以，整体的组织很随意。并且有很多复杂的业务场景一个函数非常长，导致后来的同学很难维护

**现有代码**

示例1

[https://ee.byted.org/madeira/repo/edu/mine\_api/-/blob/method/m\_my\_banke\_detail.go](https://ee.byted.org/madeira/repo/edu/mine_api/-/blob/method/m_my_banke_detail.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

这个接口实现就是rpc调用、代码逻辑、打包等都混在一个地方的，没有清晰的组织。不了解的人一眼看过来就比较混乱。

示例2

[https://ee.byted.org/madeira/repo/edu/mine\_api/-/blob/method/m\_keshi\_detail.go](https://ee.byted.org/madeira/repo/edu/mine_api/-/blob/method/m_keshi_detail.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

函数很长，代码逻辑没有清晰的组织，没有并发，性能有优化空间。

示例3

[https://ee.byted.org/madeira/repo/edu/feed\_api/-/blob/method/m\_banke\_detail.go](https://ee.byted.org/madeira/repo/edu/feed_api/-/blob/method/m_banke_detail.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

调用、打包等组织的比较混乱。

这几段代码的共同点是，都是API层，调用了一批rpc接口，拼接成结果返回给客户端，但是业务逻辑复杂，都会导致整体调用、打包、过滤、错误处理的流程很混乱，最好是用一定的代码结构规划来组织。

代码组织

以mine\_api的my\_banke\_detail接口为例，这个接口调用了学生、班课、考试、标记等一些服务、对班课信息进行拼装然后返回。

方式1 业务分层

在原有基础上，按照过程组织的方式拆分函数

// 1. 检查参数和鉴权

if err = check(ctx, req); err != nil {

    return

}

// 2. 获取和检查学习购课情况

resp1, err = getStudentInfo(ctx, userId, bankeId)

if err != nil {

    return

}

// 3. 获取班课信息

resp2, err = getBankeInfo(ctx, bankeId, xiaobanId)

if err != nil {

    return

}

// 4. 打包结果1

// 打包的函数中适当的进行分层组织

// 4.1 pack1()

// 4.2 pack2()

resp.MybankeDetail = packer.PackBankeDetailV2()

// 5. 获取Homework信息，并打包

resp3, err = getHomework()

// 6. packHomework()

packHomework(resp3, )

改进点：代码拆分成小函数，更容易看懂

问题：在逻辑组织上还是比较混乱，很多时候没办法很好的按业务拆分

方式2 逻辑分层

// 班课详情的逻辑处理

type bankeDetailLogic struct {

    kittyContext \*kitty\_context.KittyContext

    ctx          context.Context

    resp         \*pb\_gen.MineV1MyBankeDetailResponse

    userId  int64

    bankeId int64

    xiaobanId        int64

    studyKeciInfoMap map[int64][]\*student.KeshiInfo

    bankeStatus      student.BankeStatus

    bankeXiaoban     \*class\_business.BankeWithXiaoban

    banxing          \*class\_common.Banxing

    keciRoomList     []\*class\_common.KeciRoom

    markCountMap     map[int64]int32

    // 完成的课次和未完成课次

    unfinishedKeciList []\*class\_common.KeciRoom

    finishedKeciList   []\*class\_common.KeciRoom

}

func (l \*bankeDetailLogic) Run() error {

    // 1. 远程加载数据

    if err := l.load(); err != nil {

        return err

    }

    // 2. 打包数据，构造结果

    if err := l.pack(); err != nil {

        return err

    }

    return nil

}

func (l \*bankeDetailLogic) load() error {

    var err error

    // 1. 根据用户ID和班课ID查询xiaobanId，学过的课次信息，班课状态

    if err = l.loadStudentKeciInfo(); err != nil {

      return err

    }

    // 2. 加载班课信息

    if err = l.loadBankeInfo(); err != nil {

      return err

    }

    // 3. 加载mark信息

    if err = l.loadMarkCount(); err != nil {

      return err

    }

    // 4. 加载homework信息

    if err = l.loadHomeWork(); err != nil {

      return err

    }

}

func (l \*bankeDetailLogic) pack() error {

    // 1. 打包班课信息

    l.packBankeInfo()

    // 2. 打包老师信息

    l.packTeacher()

    //

    l.packMark()

    // ...其它pack

}

整体按照流程逻辑来分层，目前我这边的经验是按照load -> filter -> pack的阶段来组织代码具体普遍性，这么组织代码时，可以用一个公用的结构体来保存中间结果，将流程串起来。在每个阶段，再按业务功能拆分成不同的函数

优势：

* 整体流程清晰，有一定的代码设计，对于每个新功能写在什么位置，相对比较明确。
* 方便做并发处理，提升响应速度

对于不复杂的接口而言，做到这一步已经基本OK了

方式3 阶段抽象

针对类似mine\_api/feed\_api的接口逻辑，需要在此基础上做更进一步的抽象，例如feed\_api中的精选和学科频道，目前的接口中的逻辑较混乱和不通用，而且看目前的接口都没有设计分页逻辑，数据全都是一次性返回的。

// 伪代码: 用interface将流程串起来

// 执行Load操作

type LoaderInterface interface {

    Execute() error

}

// 执行过滤操作

type FilterInterface interface {

    Filter()

}

// 执行打包操作

type PackerInterface interface {

    PackData(Item, Resp)

}

type LoadMgr struct {

    loaders []LoaderInterface

}

type FilterMgr struct {

    filters []FilterInterface

}

type PackerMgr struct {

    packers []PackerInterface

}

type Handler struct {

    loaderMgr \*LoadMgr

    filterMgr \*FilterMgr

    packerMgr \*PackMgr

}

func (h \*Handler) Run() {

    loaderMgr.Run()

    filterMgr.Run()

    packerMgr.Run()

}

这种模式的好处是：

1. 代码逻辑组织进一步解耦，方便复用而不会互相影响。
2. 整体逻辑组织有层次感，流程化比较清晰，能够方便的看出哪里可以并发处理，进行提速。
3. 新加功能时位置明确，而不会到处引用和互相调用。

这里建议feed\_api重构的核心逻辑：

1. 在执行上面的操作前，最好将数据的排序做好，**可以采用刷索引，且单独抽象排序服务的方式来做**
2. loader过程最大可能做成并发处理，关于如何并发可以在后面介绍
3. 有翻页功能，用cursor和has\_more控制会比较方便，也更容易扩展双向翻页

性能优化1 - 合并请求

现有代码

示例

**func** setHomework(ctx context.Context, resp \*pb\_gen.MineV1MyBankeDetailResponse, userId **int64**) {

    keciInfoList := resp.MybankeDetail.GetFinishedKecis()

    keciInfoList = append(keciInfoList, resp.MybankeDetail.GetUnfinishedKecis()...)

**for** \_, keci := **range** keciInfoList {

**for** \_, keshi := **range** keci.GetKeshiList() {

**if** keshi.GetHomework() != **nil** {

                // 查看

                examId, err := strconv.Atoi(keshi.GetHomework().GetExamId())

**if** err != **nil** {

                    logs.CtxError(ctx, "setHomework failed, user\_id:%v keshi:%+v, err:%v", userId, keshi, err)

**continue**

                }

                rpcResp, err := rpc.GetUserExaminationReport(ctx, **int64**(examId), userId)

**if** err != **nil** {

                    logs.CtxError(ctx, "setHomework rpc examination err:%s,userId:%s,examId:%+v", err, userId, examId)

                    // rpc 查询课时作业出错，直接return

**continue**

                }

                keshi.Homework.UserExamStatus = pb\_gen.UserExamStatus(rpcResp.GetUserExamStatus())

                keshi.Homework.Score = rpcResp.GetUserScore()

            }

        }

    }

}

存在问题：在循环中调了RPC，一个请求会产生大量的RPC调用

**解决方式**：类似的需求，需要改造Server端的代码，支持批量获取

性能优化2 - 并发调用

[https://ee.byted.org/madeira/repo/edu/mine\_api/-/blob/method/m\_keshi\_detail.go](https://ee.byted.org/madeira/repo/edu/mine_api/-/blob/method/m_keshi_detail.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

    // 获取课时评价入口

    setKeshiEvaluation(ctx, resp, userId, keshiId)

    // 设置课时课后作业信息

    setKeshiHomework(ctx, resp, userId, homeworkId, keciId)

    // 设置测验信息

    setQuizInfo(ctx, resp, userId, keshiRoom)

    // 设置标记信息

    setMarkInfo(ctx, resp, userId, keshiId)

存在问题：这几个操作是串行的，都的rpc调用，其实互相之间没有依赖，是可以改成并行操作的，提高响应速度

解决方式：可以简单改成并发调用。**但对于这个接口而言，这不是最好的调用方式，可以参考「代码组织1 - 批量RPC」中所描述的进行抽象**

// 简单的并发实现

var wg sync.WaitGroup

wg.Add(1)

go func(resp, userId, keshiId) {

    setKeshiEvaluation(ctx, resp, userId, keshiId)

    wg.Done()

}(resp, userId, keshiId)

wg.Add(1)

go func(resp, userId, homeworkId, keciId) {

    setKeshiHomework(ctx, resp, userId, homeworkId, keciId)

    wg.Done()

}(resp, userId, homeworkId, keshiId)

wg.Wait()

支持依赖关系的并发调用框架

代码地址：[https://ee.byted.org/madeira/repo/gopkg/smartloader/](https://ee.byted.org/madeira/repo/gopkg/smartloader/" \t "/Users/yangwenshuo/Documents\\x/_blank)

简单的使用示例：

ctx := context.Background()

pLoader := NewSmartLoader(ctx)

lData1 := loadData{l1: 2}

loader1 := NewLoaderWrapByFunc("loader1", genTestRunner("loader1", &lData1))

loader2 := NewLoaderWrapByFunc("loader2", genTestRunner("loader2", &lData1))

loader3 := NewLoaderWrapByFunc("loader3", genTestRunner("loader3", &lData1))

loader4 := NewLoaderWrapByFunc("loader4", genTestRunner("loader4", &lData1))

loader5 := NewLoaderWrap(&testLoader5{lData: &lData1})

// 指定loader依赖关系

loader2.AddPreLoader(loader1)

loader3.AddPreLoader(loader2)

loader4.AddPreLoader(loader3)

loader5.AddPreLoader(loader4)

// 添加loader

pLoader.AddLoader(loader1)

pLoader.AddLoader(loader2)

pLoader.AddLoader(loader4)

pLoader.AddLoader(loader3)

pLoader.AddLoader(loader5)

// 执行

pLoader.Execute()

// 文本形式打印loader依赖关系，debug用

pLoader.GenLoaderRelation()

关键代码

// LoaderWrap function execute the real loader.

type LoaderWrap struct {

    // Loader 耗时

    usedTime int64

    loader LoaderInterface

    // 是否执行成功

    success bool

    // 等待当前Loader执行完的WaitGroup列表

    waitList []\*sync.WaitGroup

    // 当前LoaderWrap等待的WaitGroup

    wg \*sync.WaitGroup

    // 记录preLoaders

    preLoaders []\*LoaderWrap

    // 服务属性选项

    opts \*options

}

        loaderNum := sl.LoaderNum()

        indexChan := make(chan int, loaderNum)

        logs.CtxDebug(sl.ctx, "[SmartLoader.Execute] num: %d", loaderNum)

        for i, loader := range sl.loaders {

                go func(loader \*LoaderWrap, index int) {

                        defer func() {

                                if e := recover(); e != nil {

                                        const size = 64 << 10

                                        buf := make([]byte, size)

                                        buf = buf[:runtime.Stack(buf, false)]

                                        logs.CtxError(sl.ctx, "[Loader.Execute] LoaderName %s %s %s", loader.Name(), e, buf)

                                        metricsClient.EmitCounter(fmt.Sprintf("%s.loader.panic", curPsm), 1, "", getMetricsKv(loader.Name()))

                                        loader.SetSuccess(false)

                                }

                        }()

                        logs.CtxDebug(sl.ctx, "[SmartLoader.Execute] index: %d", i)

                        if err := loader.Execute(); err != nil {

                                if loader.opts.level == LEVEL\_NORMAL {

                                        logs.CtxWarn(sl.ctx, "Loader execute error, name: %s, error: %s", loader.Name(), err.Error())

                                } else if loader.opts.level == LEVEL\_HIGH {

                                        logs.CtxError(sl.ctx, "Loader execute error, name: %s, error: %s", loader.Name(), err.Error())

                                }

                                metricsClient.EmitCounter(fmt.Sprintf("%s.loader.error", curPsm), 1, "", getMetricsKv(loader.Name()))

                                loader.SetSuccess(false)

                        } else {

                                loader.SetSuccess(true)

                        }

                        indexChan <- index

                        metricsClient.EmitTimer(fmt.Sprintf("%s.loader.latency", curPsm), loader.UsedTime(), "", getMetricsKv(loader.Name()))

                }(loader, i)

        }

        finishNum := 0

        var logInfo string

        timeoutChan := time.After(sl.timeout)

        for finishNum < loaderNum {

                select {

                case index := <-indexChan:

                        finishNum += 1

                        loader := sl.loaders[index]

                        logInfo = fmt.Sprintf("%s loader\_%s:%dus", logInfo, loader.Name(), loader.UsedTime())

                case <-timeoutChan:

                        logs.CtxWarn(sl.ctx, "[SmartLoader.Execute] %s, %d loaders run finish, timeout after %v", logInfo, finishNum,

                                sl.timeout.String())

                        return nil

                }

        }

复杂参数的函数设计

**如果需要设计一个公共函数，但是参数较复杂，应该如何去做？**

这里要解决的问题通常要满足以下特征

* 会在较多的业务、服务中进行使用，面向不同的场景
* 不同场景的使用方式有较大差别，传的参数也不太一样，有可能大多数情况下不是每个参数都需要

如图片URL的处理、各类client的构造等，都符合这类特征

方法1 用Config结构体传参

设计一个Config类，来传递参数，如redis、mysql客户端的参数传递

// Mysql的Client构造

func NewDBHandlerWithOptional(optional \*conf.DBOptional) \*DBHandler {}

// Redis的Client构造

func NewClientWithOption(cluster string, opt \*Option) (\*Client, error) {}

实际上这种方式也可以应用到我们的业务代码中，尤其是在API层的业务中对于不同的机型、版本、平台、用户等，需要控制不同的返回结果时，可以将各类参数封装到结构体中，在具体的业务代码中去判断。

比如feed\_api中的这个函数

func PackBankeDetail(bankeDetailThrift \*show\_hub.BankeDetail, devicePlatform, resolution string) \*pb\_gen.BankeDetail {}

就可以换一种写法

func PackBankeDetail(params \*Params，data \*Data, resp \*Resp) {}

更好一点，可以用结构体来封装

type BankePacker struct {

    params \*Params

    data \*Data

}

func(p \*BankePacker) PackBankeDetail(resp \*Resp) {

    p.PackBanke(resp)

    p.PackKeci(resp)

    // ....

}

优势：在扩展功能、临时根据某一端、某个版本的bug时，不用再添加函数参数、不用对代码进行过大的改动。

方法2 用可变参数维护选项

如这个工具库：[https://ee.byted.org/madeira/repo/edu/common/-/blob/tosimage/image.go](https://ee.byted.org/madeira/repo/edu/common/-/blob/tosimage/image.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

kitc的Client构造方式：[https://ee.byted.org/madeira/repo/kite/kitc/-/blob/options.go](https://ee.byted.org/madeira/repo/kite/kitc/-/blob/options.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

这两个库都是这种写法

优势：还是普通的函数调用，不需要的参数可以不传，调用函数很简便。

方法3 链式操作

gorm的Sql操作就是一个很典型的例子

conn.Table("module").Where("grade\_id in (?)", gradeIdList).Order("create\_time desc").Find(&moduleModelList)

这个做法在于结构体成员函数的返回值必须是结构体的指针，好处是在函数调用端写出来的代码很易于阅读。

总结

这一节汇总了几种需要有多种复杂参数时的函数设计思路，实际编码过程中可以灵活使用。

方式1：比较直观，在业务代码中可以经常使用

方式2/方式3：用于一些控制项多，但整体输入输出比较确定的场景中，在业务层的代码中场景不太多，但如果有合适的场景还是建议使用。

编程思想

**「没有银弹」（**出自《人月神话》）：没有任何技术或管理上的进展， 能够独立地许诺十年内使软件系统项目生产率、 可靠性或简洁性获得数量级上的进步。

将这句话用在编程思想上，没有一种编程思想或编程范式可以很好的解决所有问题，关键在于我们要丰富自己的武器库，熟悉每种武器最适合的场景，来写出最适合的代码。

函数式编程(FP) & 面向对象(OOP)

结论：FP和OOP是两种不同的看待事物的方式，**到底用哪种编程模式，要看问题本身适合哪个。**哪个用起来自然，和问题本身特质搭配，那就用哪个。用对了，事半功倍；用错了，就各种不舒服。你希望你一个东西模拟为Object，前提是这个东西本身容易抽象成一个Object；你希望你一个数据可以抽象为一组函数组合，前提是这样理解更自然。

对于Golang的设计来说，既有FP的部分特性、也有OOP的部分特性，也形成了Golang本身独特的编程范式。建议Golang的编写者一定要看把「Effective Go」熟读一遍，[https://golang.org/doc/effective\_go.html](https://golang.org/doc/effective_go.html" \t "/Users/yangwenshuo/Documents\\x/_blank)，理解Golang独特的编程思维

函数式编程

* 函数式编程的几个明显特点
  + 函数是first class：即函数和其它数据类型一样，可以赋值，也可以传参和作为返回值
  + 没有「副作用」：函数不可以修改外部变量的状态，除了产生函数运算的结果之外，不能有其它的影响。
  + 「引用透明」：函数的运行不依赖于外部变量或"状态"，只依赖于输入的参数，任何时候只要参数相同，所得到的返回值总是相同的
  + 支持闭包
* 优势
  + 代码简洁，易于理解阅读
  + 没有副作用，更方便的进行单元测试
  + 不修改变量，不存在并发死锁问题，更容易进行「并发编程」

函数式编程思想，不在这里展开，大家可以自行学习，有精力的同学建议学习一门函数式编程语言，会对FP编程思维有深入的理解 。具体到实际业务场景以及Go语言，不必要用纯粹的函数式，也无法用纯粹的函数式来编写。不过掌握一些编程思想，在适当的时候使用，能起到很好的效果。

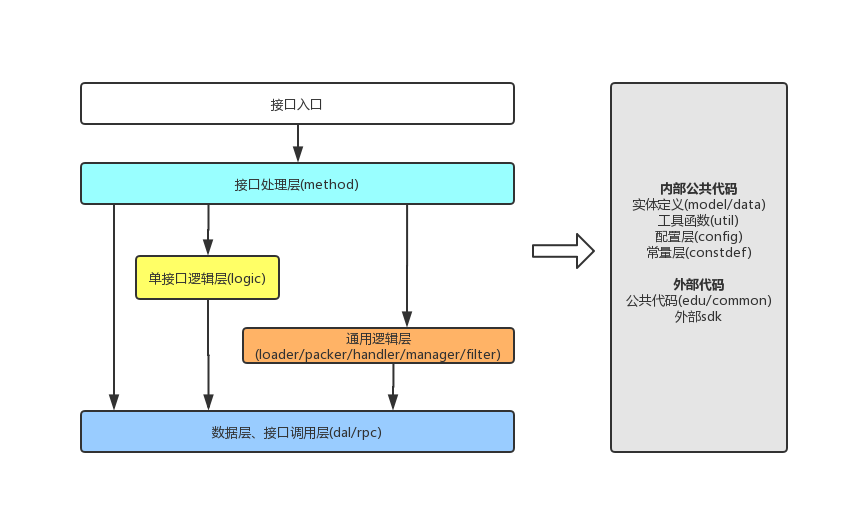
Middleware

[https://ee.byted.org/madeira/repo/kite/endpoint/-/blob/endpoint.go](https://ee.byted.org/madeira/repo/kite/endpoint/-/blob/endpoint.go" \t "/Users/yangwenshuo/Documents\\x/_blank)

代码组织2 - 工程组织结构

现在EV的Go项目大的结构都是依赖kitty和ev\_code\_gen生成，风格较统一，但是到具体的项目上，部分go的包的引用还是不太统一。

先看一个参考图，箭头表示包的依赖关系，图中的结构是从上到下依赖，从左到右依赖



* **接口处理层(method)**：代码生成工具为每个接口自动生成一个文件，把整个接口的逻辑串起来，常见流程1. 参数检查，2. 调用xxx，3. 打包结果 或者1. 参数检查，2. 组装data， 3. 写入DB， 4. 返回
  + 禁止两个不同的method文件中的函数互相调用，如果有需要，最好还是抽出到另一个package中
* **单接口逻辑层(logic)**：如果接口的逻辑太复杂，建议单独将核心逻辑抽出来做，这样核心逻辑能够理的比较清楚一些。
* **通用逻辑层**：如果有公共的逻辑需要统一，最好是抽象到公共的package中，利用封装、接口等形式，最大限度的进行抽象。
* **数据层、接口调用层**：DB、Redis的读写，远程接口调用，不依赖上层逻辑
* **内部公共代码**：工程内部的数据定义、工具函数、配置读取、常量定义等，原则上这些package禁止依赖逻辑代码。
* **外部代码**：对外部代码产生依赖