iread总结整理

注释规范：

包名规范：

引入库规范：

常用基础库：

变量规范：

err处理规范：

参数规范：

方法规范：

正则规范：

返回值规范：

设计规范：

日志规范:

单元测试规范:

历届考题的设计问题及KCP

1、架构设计不合理（函数返回值使用方法，并且里面便令在此方法中获取）

2、如若考试代码为logic层，里面有对redis或mysql等其他存储层的操作，应该将pb进行转换成对应的实体...

3、sql语句拼接、redis相关的操作应该属于dao层或repo层而不应该放到logic层

4、time.Now()不是幂等，不方便写单测，可以考虑将now函数声明为成员变量，方便依赖注入

5、网络请求使用包方法，没办法写单元测试，应该将dao层声明为interface，方便依赖注入

6、循环中不要调用网络访问

7、redis与db的分布式事务不能保证一致性

8、存储方式考虑，是否可以抗住压力

9、redis形成大key问题和过期时间设置问题

10、代码中有对接口的定义，需要注意是否必要，绑定的结构体是否只有一个，并指出mock的问题（只是...

11、context不应该携带业务相关的数据

12、日志的打印跟错误返回要使用format能力，而不是字符串拼接

13、时间相关定义跟计算都要通过time包实现不要使用整数自己算

14、err返回后，如若没有退出要考虑后续逻辑会不会painc

15、URL的拼接后要做urlencode即使现在不需要

16、if else的逻辑要看跟if是否是一起，还是需要用新的if，提高可读性

17、代码大量重复应该是散弹式修改，需要提出来

18、report相关的逻辑属于次要逻辑，不要放到wg,add()中省得以为少了一个逻辑处理

19、函数过长要考虑是否可以拆分并给出合理建议（函数职责要单一）

20、不要使用sync.WaitGroup()方法，没有超时控制跟err处理

21、重试逻辑要跟业务逻辑进行解耦不要混在一起

22、rand函数的初始化要放到init中，不要每次用都调，并且要考虑概率均等问题

23、多个文件不同方法入参几乎一致，只是同一件事的不同实现，应该意识到提成接口

24、大量switch要考虑使用表驱动的方式（23、24联动）

25、协程并发问题（用errgroup）

26、协程返回值处理问题(用errgroup)

27、结构实体跟逻辑要分开不要写到一起

28、事务的能力不应该通过接口方式实现，而应该是实现层去关心的

29、如果switch里的判断条件跟返回值都是在一个包中，就应该定义在包中

30、接口的实现应该放到不同的包里的(包职责不单一)

31、使用模板简化复杂格式化

32、多次出现 switch-case，应根据类型抽象为接口

33、方法逻辑重复

34、反复创建client

35、client 操作细节与业务逻辑耦和，缺少分层

36、多次在函数内定义结构体

37、使用结构体成员变量作为逻辑中转

38、直接依赖全局配置 config 包

39、分页数据要不要先存储再取出，重复逻辑

注释规范：

1、包名要有注释

2、结构体要有注释

3、方法说明，要说清楚这个方法干了什么

4、注释要跟实际实现的一样，比如重启服务应该先杀死服务再启动不能直接启动

5、//跟注释之间要有空格，中文英文之间要有空格

包名规范：

1、包名不要过于通用 如：url，rpc等

2、coredump--> painc

引入库规范：

1、包别名不要用驼峰跟下划线，全小写

2、引入的包不要过于通用 如:comment，third这种

3、不要引入私有库如：git.code.oa.com/cheaterlin/process-watcher/server

4、尽量使用基础库，第三方库要甚用gjson.Get(string(rspBytes), "#.message").Array()直接用json库代替即可("encoding/json")

5、引入proto时别名要以pb结尾，如：onlinedocspb docx/backend/common/proto/3rd/oidb/docs

6、使用go.mod管理包，不要使用相对路径如：cgi_service_ilive_gift_solitaire_svr/rpcimpl

常用基础库：

1、限流器 "golang.org/x/time/rate"包

2、获取输入参数使用flag包

3、拼接目录使用path/filepath库的filepath.Join这样可以避免斜杠多或少的问题

4、读取文件目录不要使用(dir, err := os.Open(path);infos, err = dir.Readdir(0)) ---> dirs, err := os.ReadDir(procPath)

5、rpc服务的存储调用尽量使用trpc-database，自带的trpc client filter所有能力，不会丢失监控 日志 调用链

6、使用redis使用git.code.oa.com/trpc-go/trpc-database/redis

变量规范：

1、中间变量：current

2、结果变量：result

3、时间间隔：defaultInterval

4、如果变量是string用==""来判断，不要用len()==0

5、实例的名字叫instance即可不要dtserver这种

6、sync.once延时加载，是否需要直接定义 var dtService = &DependToolService{}

7、方法中的参数要跟原始定义相同 for _, msg := range commits {} --> for _, commit := range commits {}

8、map不要声明长度storyMap := make(map[string]int, 0) --> storyMap := make(map[string]int)

9、如果用map判重，后面用bool类型比较好storyMap := make(map[string]int) --> storyMap := make(map[string]bool)

10、变量不要跟包名有冲突如 url := ....

11、数组的声明用 var xxxx []string

12、不要出现xxxList这种变量用复数代替即可msgList := gjson.Get(string(rspBytes), "#.message").Array()---> msgs := gjson.Get(string(rspBytes), "#.message").Array()

13、变量意义要说清楚 story--> storyID, storyIDMap

```
for story := range storyMap {
        storyIds = append(storyIds, story)
}
---->
for storyID := range storyIDMap {
        storyIds = append(storyIds, storyID)
}
```

14、结构体要考虑是否需要导出，要是需要导出最好使用newXXXX

15、时间使用时间单位不要用数字const defaultInterval = 1000 * 60 * 30 ---> const defaultInterval = 30 * time.Minute

16、一个var或者const不需要 block

17、魔数要加入注释说明意义如：maxProcessNameLen = 15要说明为啥是15

18、时间类直接用time.Time没有必要使用引用

19、is开头的变量应该是bool类型

20、配置不要用全局配置 如：config.MyCustomConfig.MyConfig.DophinActID

err处理规范：

1、err的声明要看作用域，var err error 看是否可以不用声明

2、当只有err返回的时候用短命名： if err := checkPreviousModuleFilled(submodule);err != nil{}

3、返回的err如果有errors.New(fmt.Sprintf()) -- > fmt.Errorf(

4、返回错误时要用err不能放到别处

5、返回值的错误放到error中不要ret跟err都需要判断

ret.Code = retcode.ServerErrUnknown

ret.Msg = err.Error() --> fmt.Error()

6、err返回值要保持一致不要一会带code一会不带如：

Go 复制代码

```go
if err != nil {
    return nil, errutil.WrapInternalErr(ctx, err)
  }
  if int(total) != len(couponIDs) {
    return nil, trpcerr.New(errcode.ErrInvalidCoupon)
  }
```

参数规范：

1、专有名词：HTTP SSH都要用大写 mrCommitsUri --> mrCommitsURI，ids --> IDs，常用parse解析，conver转换

2、参数名已经在方法中申明了，就不要再出现了func ParseSubmodules(gitmodules []byte) ([]*Submodule, error) —> func ParseSubmodules(src []byte) ([]*Submodule, error)

3、同一个参数不要出现不同名称，如：previousModule，Submodule等

4、go的map就是指针，不用再输出map，直接内部修改map即可

5、参数要考虑是否可以是一个结构体代替，在内部分开解析而不是一个一个传增加复杂度如：

redirectLog(server.app.ConfigStore.LogSettings.FileLocation,

server.app.ConfigStore.LogSettings.MaxFileNum,

server.app.ConfigStore.LogSettings.MaxFileSize)

---->

redirectLog(server.app.ConfigStore.LogSettings)

6、单词不要过分缩写paraCheckLog---> paramCheckLog

7、常量要单独定义好，方法参数中最好不要有常量，放在方法中自己拼接 parseProcessName(procPath+info.Name()+"/comm"): /comm提出成常量，在方法中自己拼接如：parseProcessName(info.Name()):

8、参数数量不要超过5个

9、参数类型要有意义比如：func getUserWaitingObtainCouponListSql(tp int32, memberId string)

---->

type State int

func getUserWaitingObtainCouponListSql(tp State, memberId string)

方法规范：

1、包名跟方法名联动，不要出现gitmodules.ParseSubmodules

2、方法名不要包含参数 func parseModuleNameFromLine(sectionName string) (string, error) —> parseModuleName(line string)

3、当参数已经能提现输入参数的时候，就不要在方法中再提现了 func ConvertURLToSSH(url string) (string, error) -> func ConvertToSSH(url string) (string, error)

4、用is开头需要返回bool类型

5、当方法只是校验参数的时候，应该把方法绑定到结构体上 如：func checkPreviousModuleFilled(previousModule *Submodule) error ->

func (m *Submodule) validate() error

6、方法名要包含功能，不能起的太大func checkAndTrimLine(line string) (string, bool) -> func trimCommentAndSpace(line string) (string, bool)

7、同一个key的判断不要过多，用switch替换会更清楚

8、函数不应过度封装 func setValue(key, value string, module *Submodule) 直接放到parseValue中更合适，当方法中只用到了一个成员变量的就不要再封装了server.startServer() ---> server.app.Serve()即可

9、方法要考虑是有需要能导出

10、方法参数不要有换行

11、go支持多返回值，需要返回的参数不要当参数传入 func xxxx(context.Context,req,rsp) (err error) -> func XXX(context.Context, req) (rsp, error)

11、看两个方法是否可以放到一起

baseRsp, err := biz.GetMRInfoByID(ctx, req, rsp)

rsp.BaseRsp, err = handleBaseRsp(baseRsp, err) --> handleBaseRsp(biz.GetMRInfoByID(ctx, req, rsp))

12、方法名跟绑定的结构体名字不要重复如：

func (s *Server) startServer() error { ----> func (s *Server) start() error {

13、使用exec.Command不要出现 exec.Command("sh", "-c", fmt.Sprintf("cd %s;%s", path, cmd))而是拆成两个先设置 Command 的 Dir，然后再跑命令防止在参数path中做一些有恶意的操作比如删除等

14、方法里的功能要单一，如有别的处理可以单独抽象，如判断目录是否正确等

15、看方法是否可以合并,如后面没有再使用projectPath参数

projectPath = url.PathEscape(projectPath)

urlPath := fmt.Sprintf(tGitHost+mrCommitsUri, projectPath, mrID)

--->

urlPath := fmt.Sprintf(tGitHost+mrCommitsUri, url.PathEscape(projectPath), mrID)

16、时间间隔使用time.Since不用使用timeNow.Sub(*c.lastTime) / time.Millisecond < time.Duration(c.diff)这种

17、时间间隔用time.Tick不要使用sleep避免时间偏移

18、不要使用httpPattern.MatchString(url)直接使用 m := httpPattern.FindStringSubmatch(url); m != nil {}

19、方法实现要符合方法名，比如叫checkProcess()但是里面还有其他功能，可以叫handle里面再细分方法如：check, gather(收集), notify, start等

20、判断逻辑不要过长如：!solitaireSucc || (afterTimeRate > 0 && afterTimeRate != timeRateScore) || isInvisible

正则规范：

1、正则起名用RE结尾

2、http的验证使用net/url包

3、开始就写好，不要每次都调用如:nameRE = `^\[\s*gitmodule\s+"([^"]+)"\]\s*$`——>nameRE = regexp.MustCompile(`^\[\s*gitmodule\s+"([^"]+)"\]\s*$`)

4、正则匹配开始就要匹配结果

5、尽量避免使用贪婪匹配regexp.MustCompile(`(.*)=(.*)`) -> regexp.MustCompile(`^\s*([^=\s]+)\s*=\s*(.+)\s*$`)

6、如果使用正则要匹配完成 strings.TrimSpace() 这种直接在正则中处理 (.*)=(.*) --> ^([^=]+)\s*=\s*(.+)$

返回值规范：

1、返回值要跟方法实现保持一致 cfg := bufio.NewScanner(bytes.NewReader(gitmodules)) cfg-->san

2、返回值有判断是否合法的使用ok line, isValid := checkAndTrimLine(cfg.Text()) --> line, ok := checkAndTrimLine(cfg.Text())

3、要看下是否会返回多个还是一个就可result := httpPattern.FindAllStringSubmatch(url, -1)...result[0][2] -> httpPattern.FindStringSubmatch(url);...result[2]

4、返回值的赋值不要离着太远

5、返回值要有一些意义不要ret.Code = retcode.ServerErrUnknown这种的

6、设置结果放到一起，中间不要有空行

```
resp.StoryIds = getStoryListByMRInfo(commits)


ret.Code = retcode.SUCCESS
```

--->

```
resp.StoryIds = getStoryListByMRInfo(commits)
ret.Code = retcode.SUCCESS
```

7、go返回值于逻辑之间不需要空格

```
for story := range storyMap {
        storyIds = append(storyIds, story)
}


return storyIds
```

---->

```
for story := range storyMap {
```

storyIds = append(storyIds, story)

}

return storyIds

8、返回值尽可能不要return nil，将上面的方法直接返回即可

if err = server.startServer(); err != nil {

return err

}

return nil

---->

return server.startServer()

9、返回参数不要过多如：err, _, score, _ = rpcimpl.GetMemberRankInfo(anchorUid, rankID, objID)

10、err要放到返回值以后一个参数

设计规范：

1、参数校验放到协议层，不要放到业务层

2、req或者rsp这种属于协议层，不要传入业务成，变成业务需要的结构体或者其他的再传入业务层

3、token验证应该放在协议层，不应该放到第三方库里比如third库

4、减少引入不必要的引包

command.RunE = server.Run ---> command.RunE = func(cmd *cobra.Command, args []string) error {

// get config from command

server.Run(config)

}

查看server.Run方法，发现就是通过获取cmd参数如：configDSN, err := cmd.PersistentFlags().GetString("config")

后续没有再依赖参数，就可以拆出来，放到command里避免再server再引入cmd包

5、主函数异常退出后，要添加错误处理

if err := cmd.Run(os.Args[1:]); err != nil {

```
        os.Exit(1)
} ---->
if err := cmd.Run(os.Args[1:]); err != nil {
        log.fatal(err)
        os.Exit(1)
}
```

6、相同的设置放到一起不要分开

7、fatal错误一般都是程序直接退出，而不是打印完日志就完了 fatalLog ---> errorLog

8、初始化结构体，直接提供一个构造函数，不要用SetDefault这种形式

```
app.ConfigStore = &Config{}
app.ConfigStore.SetDefault()
----->
app.ConfigStore = NewStore()
```

9、参数检测应该放到config文件初始化中

10、启动进程的时候最好不要用&在后台启动，这样会导致监控挂了后，进程可能仍然存在，要不就是在启动的时候先检查进程是否存活

11、参数声明放到使用前

12、方法要根据调用顺序放置

13、只取一个最大值没有必要取出再排序，直接用在循环里判断即可

14、读取日志使用bufio.NewScanner(file)不要使用bufReader := bufio.NewReader(file);bufReader.ReadBytes('\n')这种魔数方法

例子如：

```
    var (
        inPanicLines bool
        nPanicLines int
        result bytes.Buffer
        scn = bufio.NewScanner(file)
    )
    for scn.Scan() && nPanicLines < maxPanicLines {
```

```
        line := scn.Text()
        if !inPanicLines {
            if !strings.HasPrefix(line, panicPrefix) {
                continue
            }
            inPanicLines = true
        }
        nPanicLines++
        result.WriteString(line)
        result.WriteRune('\n')
    }
    return result.String(), scn.Err()
```

15、http请求要添加上下文，做好超时处理不要出现http.Get(requestURL)

16、rpc服务的存储调用尽量使用trpc-database，自带的trpc client filter所有能力，不会丢失监控 日志 调用链

17、传入方法不要嵌套太多层

18、分层要清晰，业务层不要暴露内部实现

19、网路请求要带上context做超时控制

20、要看清是否使用的是rpc请求，要是有dao层的封装就是网络请求，不能在for循环中调用网络请求，要做并发控制【语言组织：后台服务尽量不要for循环发起网络请求，容易导致耗时过长，甚至请求卡死】

21、dao层不要出现业务逻辑如：dao.IsNewUser(memberId)

22、相同代码逻辑要考虑是否能提出如： （homeCouponList跟advCouponList结构一样）

Go | 复制代码

```go
if len(homeCouponList) > 0 {
    homeIssueEndTime = homeCouponList[0].IssueEndTime
  }
  if len(advCouponList) > 0 {
    advIssueEndTime = advCouponList[0].IssueEndTime
  }
```

23、不要出现魔数判断如:

```go
 switch tp {
  case 0:
    sql = fmt.Sprintf(waitingObtainCouponSqlTmp, sqlHomePageAdv+` union `+sqlHomePagePopup, inParam)
  case 1:
    sql = fmt.Sprintf(waitingObtainCouponSqlTmp, sqlHomePageAdv, inParam)
  case 2:
    sql = fmt.Sprintf(waitingObtainCouponSqlTmp, sqlHomePagePopup, inParam)
  }
  return sql
}
```

添加判断类型如：(default默认要有)

```
type State int

const (
   Running State = iota
)
switch tp {
  case Running:
  default:
  return sql
}
```

28、设计成接口要考虑是否必要，未来是否有可拓展的可能语言组织：

【1、在这里 interface 只有一个实现，没有抽象的必要，也就不需要定义 interface

2、把 interface 定义在实现侧，会强迫用户侧在测试的时候 mock 自己不需要的方法以满足定义】

29、context要应该携带跟业务无关的值，比如：取消信号、超时时间、截止时间、k-v 等而不应该携带业务相关的值语言组织：【alues of the context.Context type carry security credentials, tracing information, deadlines, and cancellation signals across API and process boundaries.

在此场景下，reminder 不符合上述的任何一种使用场景，因此应显式传递，而非使用context.Context。】

30、时间相关不要自定义，使用time库提供的功能time.Weekday()获取星期几（周日为0）

31、空结构的绑定没有意义 语言组织：【对于该包而言，实现只包含在单个文件中，而 reminder 包内又不包含其他业务逻辑。使用空的 reminderImpl 结构除了引入更多复杂度并没有意义】

32、redis跟mysql进行封装，最好绑到结构体上语言组织：对 redis 的相关操作可以做一些封装，让相关的可逆操作更加集中，也避免重复的调用 config.GetConfig(ctx).RemindRedisKey 拼接相关 redis 请求：

```
var DefualtImpl = &reminderImpl {
store: newStore(newRedisClient(...)),
}

impl.store.Add(...)
impl.store.Remove(..)
impl.store.List(...)
```

33、避免出现metrics.IncrCounter("remind_failure", 1)，容易写错内容语言组织：metrics 应使用更为结构化的调用方式，以保存相关元数据（如类型、metric 功能描述等），避免随意使用字符串寻找 metric 造成的误用。

```
var remindFailureCounter = metrics.NewCounter("remind_failure")

remindFailureCounter.Incr()
```

34、考虑重启情况任务是否会有丢失情况

35、select要看是否启了goroutine，语言组织：没有意义的 chan。由于该函数会阻塞调用方，所以不存在异步结束的情况，自然就不需要使用 chan 来进行通知。

36、定义了接口就不要定义fake，如：

Go 复制代码

```go
func RemoveRemindTask(ctx context.Context, globalPadID string) error {
  return DefaultReminder.RemoveRemindTask(ctx, globalPadID)
}
```

语言组织：【如果定义接口是为了让调用方可以进行依赖注入，那么这三个方法不会被用到。反之，如果使用了这三个方法，用户就不会用到 Reminder 接口。】

37、使用 g, ctx := errgroup.WithContext(ctx)来控制goroutine

38、重试机制使用https://github.com/avast/retry-go语言组织：【重试控制逻辑和业务逻辑耦合，重试逻辑实现不当会带来难以预留的后果。另外控制逻辑参杂业务逻辑，实现业务的过程中，还得审慎的

考量何时发起重试，何时退出控制流程等，开发和维护都有不少心智负担，同时可读性和业务理解上都带来困难，比如需要去猜测3，或者5这些数字的背景到底是什么。建议重试控制逻辑尽量和业务逻辑分离，聚焦业务流程即可。比如使用已有库：https://github.com/avast/retry-go】

39、rand.Seed()要初始化一次，里面带锁会影响性能，用到随机数用：rand.Intn()

40、面临代码霰弹式修改的问题

41、authenticator验证器

日志规范:

1、日志输出要有意义，并且中间用空格log.InfoContextf(ctx, "url:%s", url) -->log.InfoContextf(ctx, "url :%s", url)

2、日志封装要到位不要再单独加日志设置逻辑如：

```
func needInfoLog(b bool) {
        if !b {
                rlog.SetLevel(zapcore.ErrorLevel)
        }
}
```

----> 直接封装到rlog里

3、日志中不要出现中文

单元测试规范:

1、不要出现time.Now()这种不方便做单元测试【语言组织：time.Now() 不是幂等的，不方便写单测，可以考虑将now函数声明为类成员变量，方便依赖注入 如

```
type xx struct {
 now func() time.Time
}】
```

历届考题的设计问题及KCP

1、架构设计不合理（函数返回值使用方法，并且里面便令在此方法中获取）

例子：

```go
func CreateUserCoupons(ctx context.Context, req *pb.CreateUserCouponsRequest,
  userID string, wrapUserCoupons ...func(tx *gorm.DB, userCoupon *model.UserCoupon) error) (err error) {
  coupons, err := getCoupons(ctx, req.CouponIds)
......
......
return dao.TxProcess(
        func(tx *gorm.DB) error {
            for _, coupon := range coupons {
                if err := checkCouponTimesForUser(ctx, coupon, userID); err != nil {
                    return err
                }
......
```

语言组织：【全文全部使用了函数，不方便依赖注入，可测试性不好，没法写单测，有其他语言转化过来的痕迹】

2、如若考试代码为logic层，里面有对redis或mysql等其他存储层的操作，应该将pb进行转换成对应的实体结构

例子：

```go
func CreateUserCoupons(ctx context.Context, req *pb.CreateUserCouponsRequest,
  userID string, wrapUserCoupons ...func(tx *gorm.DB, userCoupon *model.UserCoupon) error) (err error) {
    ......
    ......
    return dao.TxProcess(
    ......
    ......
        if req.Channel == model.CouponChannelAdv {
                key := redismodel.GetUsersJoinedAdvKey()
                return redis.GetClient().SAdd(ctx, key, userID).Err()
        }
    ......
```

语言组织：【协议层的结构直接穿透到了存储层。如果协议产生变化，相应的变化也会带到存储层，如从TAF的jce到trpc的pb。分层不合理】

3、sql语句拼接、redis相关的操作应该属于dao层或repo层而不应该放到logic层

4、time.Now()不是幂等，不方便写单测，可以考虑将now函数声明为成员变量，方便依赖注入

语言组织：【time.Now() 不是幂等的，不方便写单测，可以考虑将now函数声明为类成员变量，方便依赖注入
如
type xx struct {
 now func() time.Time
}】

5、网络请求使用包方法，没办法写单元测试，应该将dao层声明为interface，方便依赖注入

例子：

Go | 复制代码

```go
return dao.TxProcess(
    func(tx *gorm.DB) error {
      for _, coupon := range coupons {
        if err := checkCouponTimesForUser(ctx, coupon, userID); err != nil {
          return err
        }
......
```

6、循环中不要调用网络访问

例子：

```go
return dao.TxProcess(
    func(tx *gorm.DB) error {
      for _, coupon := range coupons {
        if err := checkCouponTimesForUser(ctx, coupon, userID); err != nil {
          return err
        }
......
......
func checkCouponTimesForUser(ctx context.Context, coupon *model.Coupon, userID string) error {
    if coupon.ObtainTimes == 0 {
        return nil
    }

    total, err := dao.Count(model.UserCoupon{}, func(db *gorm.DB) *gorm.DB {
        return db.Where("member_id = ?", userID).
            Where("coupon_id = ?", coupon.ID)
    })
```

语言组织：【这个方法在外层大循环内，会带来多次网络请求，应考虑通过更好的实现避免这个循环。

如此处可通过定义“用户优惠券总计”类型可以一次性获取用户持有的优惠券，可以避免在事务中使用循环反复查询。也可以帮助解耦领域层和存储层】

7、redis与db的分布式事务不能保证一致性

8、存储方式考虑，是否可以抗住压力

例子：如优惠券活动用mysql存储，请求量大的情况下mysql会扛不住

语言组织：【这里的存储方案选择mysql有可能会有性能问题，弹窗活动这类会有突发请求，请求量大的情况下mysql会扛不住】

9、redis形成大key问题和过期时间设置问题

例子：（key为公共的，并且没有设置过期时间）

```go
if req.Channel == model.CouponChannelAdv {
        key := redismodel.GetUsersJoinedAdvKey()
        return redis.GetClient().SAdd(ctx, key, userID).Err()
```

语言组织：【如果用户很多，那么redis中可能会形成大key，并且没有过期时间】

10、代码中有对接口的定义，需要注意是否必要，绑定的结构体是否只有一个，并指出mock的问题（只是想测试一个方法需要将其他的都mock出来）

例子：

```go
type Reminder interface {
  // AddRemindTask 计算下次提醒时间，如果需要提醒，则添加提醒任务。
  AddRemindTask(ctx context.Context, globalPadID string, formData *text.FormData) error
......
......
type reminderImpl struct{}
......
......
// DefaultReminder 默认 reminder 对象
var DefaultReminder Reminder = &reminderImpl{}
......
......
// AddRemindTask 计算下次提醒时间，如果需要提醒，则添加提醒任务。
func (r *reminderImpl) AddRemindTask(ctx context.Context, globalPadID string, formData *text.FormData) error {
......
......
```

语言组织：【

1. 在这里 interface 只有一个实现，没有抽象的必要，也就不需要定义 interface
2. 把 interface 定义在实现侧，会强迫用户侧在测试的时候 mock 自己不需要的方法以满足定义

】

11、context不应该携带业务相关的数据

例子：

```go
func GetReminder(ctx context.Context) Reminder {
  v := ctx.Value(reminderContextKey{})
  if v == nil {
    return DefaultReminder
  }
  return v.(Reminder)
}
```

语言组织：【context应该携带跟业务无关的值，比如：取消信号、超时时间、截止时间、k-v 等而不应该携带业务相关的值（Values of the context.Context type carry security credentials, tracing information, deadlines, and cancellation signals across API and process boundaries. ）

在此场景下，reminder 不符合上述的任何一种使用场景，因此应显式传递，而非使用context.Context。】

12、日志的打印跟错误返回要使用format能力，而不是字符串拼接

例子：

```go
if err != nil {
      logger.Errorf("FetchAndParse failed: " + err.Error())
      return errors.New("FetchAndParse failed: " + err.Error())
```

语言组织：【1、既然 logger 提供了 Errorf，为什么不用 format 能力，而是拼接字符串?
一般来说，一个合格的 log 库会在持久化时独立存储 format 和 args，以便提供结构化的检索能力。

2、errors.New---> fmt.Errorf】

13、时间相关定义跟计算都要通过time包实现不要使用整数自己算

例子：

```go
const (
  secondsPerHour    = 3600
  secondsPerMiniute = 60
  secondsPerDay     = 24 * 3600
  hoursPerDay       = 24
  daysPerWeek       = 7
)
records, err := store.GetStore(ctx).GetRangeRecords(ctx, globalPadID, &store.GetRangeRecordsParams{
        StartTime: dtime.GetAPI(ctx).Now().Unix() - 7*secondsPerDay,
```

语言组织：【1、不要定义 time 包未提供的时间“常量”，因为他们不是常量。

2、dtime.GetAPI(ctx).Now().Unix() – 7*secondsPerDay ---> time.Now().AddDate(0,0,–7).Unix()

】

14、err返回后，如若没有退出要考虑后续逻辑会不会painc

例子：

```go
docUserData, err := store.GetDocUserData(ctx).GetFormCollectDocUserData(ctx, uid, globalPadID)
    if err != nil {
      log.ErrorContextf(ctx, "GetFormCollectDocUserData failed: %s", err.Error())
    }
    if docUserData.DisableRemindNotify {
      continue
    }
```

语言组织：【出错后是否应该 continue？docUserData 可能是 nil，5 行直接 panic。】

15、URL的拼接后要做urlencode即使现在不需要

例子：

```go
        "uid":     uid,
        "pushMsg": "请完成今日打卡任务",
        "url": fmt.Sprintf("https://%s/form/page/clock/D%s", domain.Host(),
          padid.GlobalPadIDToURLID(globalPadID)),
        "page": fmt.Sprintf("pages/detail/detail?url=https://%s/form/page/clo
    ck/D%s",
          domain.Host(), padid.GlobalPadIDToURLID(globalPadID)),
      })
```

语言组织：【应进行 url encode（哪怕你认为现在的数据是合规的）
假设 padid.GlobalPadIDToURLID 进行了修改，返回包含 `+` 和 `/` 的 base64 字符串，那么这里必将出现 bug。】

16、if else的逻辑要看跟if是否是一起，还是需要用新的if，提高可读性

例子：

```go
if isStarted(ctx, meta.StartTime) {
          logger.Infof("form is out of date, stop remind")
          return nil
        } else if isEnded(ctx, meta.EndTime) {
          logger.Infof("form is not start, stop remind")
          return nil
        }
```

语言组织：【对于另一个 guard statement，使用新的 if。】

17、对metrics的定义可以提成全局或者封装起来避免到处复制的问题

例子：

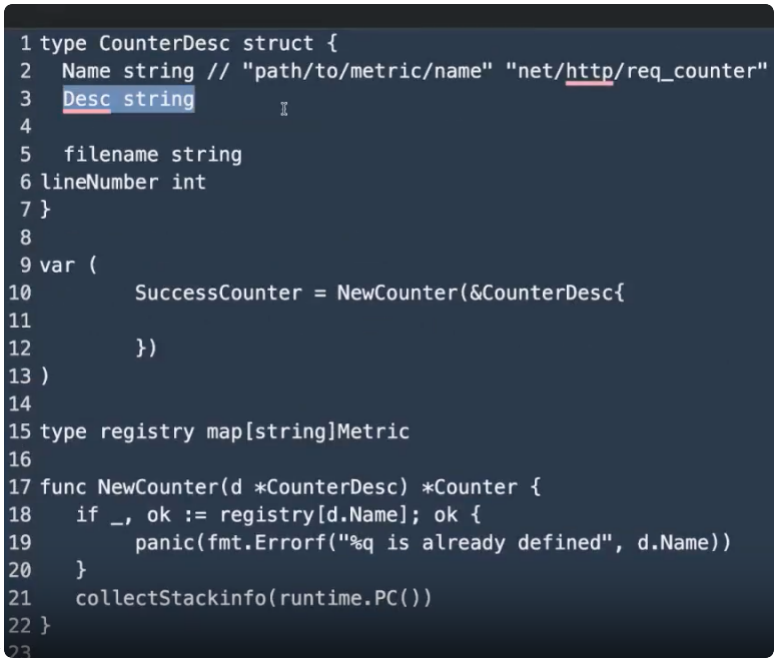
```go
if err != nil {
        metrics.IncrCounter("remind_failure", 1)
        log.ErrorContextf(ctx, "SendNotify RPC failed: %s", err.Error())
        return
```

语言组织：【metrics 应使用更为结构化的调用方式，以保存相关元数据（如类型、metric 功能描述等），避免随意使用字符串寻找 metric 造成的误用。

var remindFailureCounter = metrics.NewCounter("remind_failure")

remindFailureCounter.Incr()】

更出彩的答案



```
type CounterDesc struct {
  Name string // "path/to/metric/name" "net/http/req_counter"
  Desc string

  filename string
lineNumber int
}

var (
        SuccessCounter = NewCounter(&CounterDesc{

        })
)

type registry map[string]Metric

func NewCounter(d *CounterDesc) *Counter {
   if _, ok := registry[d.Name]; ok {
        panic(fmt.Errorf("%q is already defined", d.Name))
   }
   collectStackinfo(runtime.PC())
}
```

17、定义接口，就不要定义Default实现

如：

Go | 复制代码

```go
type Reminder interface {
  // AddRemindTask 计算下次提醒时间，如果需要提醒，则添加提醒任务。
  AddRemindTask(ctx context.Context, globalPadID string, formData *text.FormData) error
.......
var DefaultReminder Reminder = &reminderImpl{}
......
// AddRemindTask 同 Reminder.AddRemindTask
func AddRemindTask(ctx context.Context, globalPadID string, formData *text.FormData) error {
    return DefaultReminder.AddRemindTask(ctx, globalPadID, formData)
}
......
```

语言组织：【与 Reminder 接口定义存在矛盾。

如果定义接口是为了让调用方可以进行依赖注入，那么这三个方法不会被用到。反之，如果使用了这三个方法，用户就不会用到 Reminder 接口。】

17、代码大量重复应该是散弹式修改，需要提出来

例子：

Go 复制代码

```go
go func() {
    defer wg.Done()
    rankID := config.MyCustomConfig.MyConfig.DophinAnchorUserRank
    objID := constant.GetAnchorFansTotalSolitaireRankKey(anchorUid)
    err, _ := rpcimpl.UpdateRankScore(rankID, objID, sendUid, score, giftMsg.GetBillno())
    if err != nil {
      _ = attr.AttrAPI(34639112, 1) //[榜单]更新主播维度的用户贡献榜失败
    } else {
      _ = attr.AttrAPI(34639113, 1) //[榜单]更新主播维度的用户贡献榜成功
    }


  }()
  ......
    //5. 主播日榜
    go func() {
      defer wg.Done()
      rankID := config.MyCustomConfig.MyConfig.DophinAnchorDayRank
      objID := ""
      err, _ := rpcimpl.UpdateRankScore(rankID, objID, anchorUid, score, giftMsg.GetBillno())
      if err != nil {
        _ = attr.AttrAPI(34639108, 1) //[榜单]更新主播排行榜失败
      } else {
        _ = attr.AttrAPI(34639109, 1) //[榜单]更新主播排行榜成功
      }
    }()
  ......
```

语言组织：【1、代码高度相似重复，line 142 ～ line 168 等部分也存在类似的代码高度相似重复的问题：仅参数和上报类型不同。如果后续需要新增榜单或者涉及榜单业务逻辑调整，面临代码霰弹式修改的问题。

修改建议：函数参数封装为结构体，attr 上报封装到底层 rpcimpl 实现中，并使用 errgroup 加 for 循环简化代码

伪代码如下：

```
type RankScore struct {
    RankID uint64
```

```
        ObjID uint64
        UID uint64
        Score uint64
        BillNo string
        // 如果继续使用 attr 做打点上报，还可以在这里加上 attr 信息
        // 但更建议使用 007/伽利略 等多维上报替换 attr
        AttrSuccess uint64
        AttrFailed uint64
}

rankScores := []*RankScore{
        {
                RankID: config.MyCustomConfig.MyConfig.DophinRankId,
                ObjID: constant.GetAnchorTotalSolitaireRankKey(),
                UID: anchorUid,
                Score: score,
                BillNo: giftMsg.GetBillno(),
        },
        {
                RankID: config.MyCustomConfig.MyConfig.DophinUserRank,
                ObjID: constant.GetFansTotalSolitaireRankKey()
                UID: sendUid,
                Score: score,
                BillNo: giftMsg.GetBillno(),
        },
        ...
        ...
}
g, ctx := errgroup.WithContext(ctx)
for _, rankScore := range rankScores {
        rankScore := rankScore // https://golang.org/doc/faq#closures_and_goroutines
        g.Go(func() error {
                return rpcimpl.UpdateRankScore(rankScore)
        })
}
g.Go(func() error {
```

```
        return WriteSolitaireKing(ctx, score, giftMsg)
    })
    if err := g.Wait(); err != nil {
        return err
    }】
```

18、report相关的逻辑属于次要逻辑，不要放到wg,add()中省得以为少了一个逻辑处理

例子：

Go 复制代码

```go
wg := sync.WaitGroup{}
wg.Add(5)
go func{ defer wg.Done().....
go func{ defer wg.Done().....
go func{ defer wg.Done().....
go func{ defer wg.Done().....
go func{ defer wg.Done().....
go func() {
        report.TdbankReport(ctx, sendUid, anchorUid, giftMsg.GetGiftId(), giftMsg.GetGiftNum(), score, actId, giftMsg.GetBillno())
```

语言组织：【这里的本意是把tdbank上报逻辑作为次要逻辑并采用goroutine异步处理的方式避免阻塞主流程。但不应该随意 leave goroutine in-fight。 另外既然当成了次要逻辑，就不要把这部分代码嵌套在 wg.Add(5) ~ wg.Wait() 之间，否则非常容易带来困扰。

建议使用带有超时和取消的Context来进行协程生命周期管理，另外为了应对流量暴涨可以采用worker pool模式。】

19、函数过长要考虑是否可以拆分并给出合理建议（函数职责要单一）

例子：

Go 复制代码

```go
func WriteSolitaireKing(ctx *ilives.Context, score uint64, giftMsg *UnifiedMsg.NowGiftData) {
  anchorUid := giftMsg.GetAnchorUid()
  sendUid := giftMsg.GetSendUid()
  giftTs := giftMsg.GetTimeMs() / 1000
  //1.获取本轮该主播最近一次送礼时间，判断是否接龙成功;
  var awardCas int
  var awardStatus *ilive_gift_solitaire_svr.AnchorAwardStatus
  var solitaireSucc bool
  var solitaireErr, awardErr error
  {…}
  //无法判断接龙是否成功，直接停止更新
  if solitaireErr != nil {…}
  //无法判断抽奖是否成功，直接停止更新
  if awardErr != nil {…}


  //2.更新【主播+粉丝】临时榜，获取加成后的结果----接龙成功则incr，失败则强制set
  var anchorAfterScore uint64 = 0
  var fansAfterScore uint64 = 0
  var anchorErr error
  var fansErr error
  {…}
  /*
    异步更新内容:
    1.本轮赠送的金币价值;
    2.本轮用户的贡献值;
    3.本轮最后一名赠送指定礼物的用户;
  */
  ctx.DoAsync(func(c *cat.Context) {…}, time.Second*10)
  //3.往最大值榜单里面更新加成后的结果
  {
    wg := sync.WaitGroup{}
    //主播接力榜
    if anchorErr == nil {…}
    //粉丝助攻榜
    if fansErr == nil {…}


    wg.Wait()
  }
```

语言组织：【1、函数名 WriteSolitaireKing 和代码实现不一致，除了写，还有很多类似最近送礼时间、抽奖状态检查等的（Read）读逻辑，函数名无法体现出代码逻辑，更多的还是 solitaireKingProcess。

2、职责上也不单一，函数太大了，揉杂了非常多的逻辑。用了很多 {} 作用域的方式进行了代码分块，把 检查逻辑、更新【主播+粉丝】临时榜、主播接力榜、粉丝助攻榜、接力逻辑等等都揉杂在一起。既然都认识到需要 {} 分块了，为何不封装相关函数?

建议按照职责进行拆解，明确好边界，拆解成更内聚的业务逻辑模块：判断主播是否接龙成功/更新主播+粉丝临时榜单，接龙时间/更新主播接力榜/粉丝助攻榜/更新隐形接龙信息/更新最近一次送。也可以使用面向对象的思路优化这里。 抽像出 Solitaire、Rank、Award 等几个结构体，并将相关函数绑定到对应结构体上，使代码层次、函数职责更清晰。】

20、不要使用sync.WaitGroup()方法，没有超时控制跟err处理

例子：

Go 复制代码

```
1   wg := sync.WaitGroup{}
2       wg.Add(3)
3       //主播临时榜
4       go func() {…}()
17      //粉丝临时榜
18      go func() {…}()
31      //更新接龙开始时间
32      go func() {…}()
44      wg.Wait()
45    }
```

语言组织：【通篇大量使用 WaitGroup 做并发控制，但错误处理不合理：
要么大部分错误没有向上抛出，直接让可能的错误带来的风险裸奔；
要么每个子协程给一个xxxErr变量，然后分别串行判断每个xxxErr的错误信息，冗余了大量的不必要的变量，也比较低效。

此外，通篇较多error和bool并用的情况，例如这里的solitaireErr, solitaireSucc，也存在 error 并不是惯用法作为最后一个参数返回的情况，有其他语言翻译转化到golang的痕迹。

建议直接使用 errorgroup 等现成能力，简化代码的同时，也可以更好的进行错误管理，同时也避免添加新的协程需要不断手动add带来的维护负担。

```
g, ctx := errgroup.WithContext(ctx)
.....
```

```
if err := g.Wait(); err != nil {
        return err
    }】
```

21、重试逻辑要跟业务逻辑进行解耦不要混在一起

例子：

Go 复制代码

```go
for i := 0; i < 5; i++ {
      e, cas, invisibleStatus := GetRecentInvisibleStatus(anchorUid)
      if e != nil {…}
      ctx.Debug("Invisible info:%v; info:%v", anchorUid, invisibleStatus.String())
      isInvisible, _ = IsInvisible(giftTs, invisibleStatus)
      if solitaireSucc && anchorErr == nil && isInvisible {…}
      if !solitaireSucc {…}
      if !solitaireSucc && afterTimeRate == timeRateScore || solitaireSucc && afterTimeRate > 0 && afterTimeRate != timeRateScore {…}


      e = CasSetRecentInvisibleStatus(anchorUid, cas, invisibleStatus)
      if e != nil {
        continue
      }


      break
    }
```

语言组织：【重试控制逻辑和业务逻辑耦合，重试逻辑实现不当会带来难以预留的后果。另外控制逻辑参杂业务逻辑，实现业务的过程中，还得审慎的考量何时发起重试，何时退出控制流程等，开发和维护都有不少心智负担，同时可读性和业务理解上都带来困难，比如需要去猜测3，或者5这些数字的背景到底是什么。建议重试控制逻辑尽量和业务逻辑分离，聚焦业务流程即可。比如使用已有库：https://github.com/avast/retry-go

```
err := retry.Do(

    func() error {
        // 要重试的业务逻辑。
    },
)】
```

22、要有面相对象的思想，不要面相过程

例子：

Go 复制代码

```go
func WriteSolitaireKing(ctx *ilives.Context, score uint64, giftMsg *UnifiedMsg.NowGiftData) {
  anchorUid := giftMsg.GetAnchorUid()
  sendUid := giftMsg.GetSendUid()
  giftTs := giftMsg.GetTimeMs() / 1000
  //1.获取本轮该主播最近一次送礼时间，判断是否接龙成功;
  var awardCas int
  var awardStatus *ilive_gift_solitaire_svr.AnchorAwardStatus
  var solitaireSucc bool
  var solitaireErr, awardErr error
  {…}
  //无法判断接龙是否成功，直接停止更新
  if solitaireErr != nil {…}
  //无法判断抽奖是否成功，直接停止更新
  if awardErr != nil {…}


  //2.更新【主播+粉丝】临时榜，获取加成后的结果----接龙成功则incr，失败则强制set
  var anchorAfterScore uint64 = 0
  var fansAfterScore uint64 = 0
  var anchorErr error
  var fansErr error
  {…}
  /*
    异步更新内容:
    1.本轮赠送的金币价值;
    2.本轮用户的贡献值;
    3.本轮最后一名赠送指定礼物的用户;
  */
  ctx.DoAsync(func(c *cat.Context) {…}, time.Second*10)
  //3.往最大值榜单里面更新加成后的结果
  {
    wg := sync.WaitGroup{}
    //主播接力榜
    if anchorErr == nil {…}
    //粉丝助攻榜
    if fansErr == nil {…}


    wg.Wait()
  }
```

语言组织：【WriteSolitaireKing 看上去只是 WriteRankProcess 在同一层次的拆分，但拆分后 WriteSolitaireKing 依然很复杂。究其原因，是因为整个逻辑是面向过程的，而不是面向对象的，导致过多不同层次的逻辑被杂糅到了一个函数里，最后不可控。

建议使用面向对象的思路优化这里。 抽像出 Solitaire、Rank、Award 等几个结构体，并将相关函数绑定到对应结构体上，使代码层次、函数职责更清晰。

例如：接龙成功与否对流程影响较大，很多流程依赖接龙状态，业务上正常接龙和隐形接龙逻辑又紧密相关。可以抽象接龙逻辑和状态驱动机制，通过接龙状态在正常接龙和隐形接龙等不同接龙形式中转换，明确好边界。】

22、rand函数的初始化要放到init中，不要每次用都调，并且要考虑概率均等问题

例子：

Go | 复制代码

```go
func IsInInvisibleProbability() bool {
  rand.Seed(time.Now().UnixNano())
  randNum := rand.Uint64() % 100
  return randNum <= config.MyCustomConfig.MyConfig.InvisibleProbability
}
```

语言组织：【

1. 频繁seed，每次seed 出来都是独立的随机序列，同一个纳秒访问的出来的值也是同一个，另外会带来 lock contention 的问题。 建议通过init 函数来 seed。
2. rand.Uint64() % 100 出来的概率不均等。必须采用 rand.Intn(100) 的方式。
3. < 写成了 <=，导致有 1% 的偏移。 例如，即使配置为 0 的时候，其实还是有 1% 的概率。

】

23、多个文件不同方法入参几乎一致，只是同一件事的不同实现，应该意识到提成接口

1、例子

```
auth.go文件
......
func (s *AuthService) AuthEntry(ctx context.Context, in *pb.AuthEntryReq)
(*pb.AuthEntryReply, error) {
  start := time.Now()
  logger.Info(ctx, "auth begin",
    zap.String("nick", in.User),
  )


  queryEntryNum := len(in.Contents)
  if queryEntryNum == 0 {
    return nil, ErrQueryEntryEmpty
  }
......
auth_answer.go文件
......
// AuthQuestion 验证乐问查看权限
func (s *AuthService) AuthAnswer(ctx context.Context, in *pb.AuthEntryReq
, reply *EntryReply, done chan struct{}) {
  defer logger.Recover(ctx)
  defer func() {
    done <- struct{}{}
  }()
......
auth_doc.go文件
......
// AuthDoc 验证文档查看权限
func (s *AuthService) AuthDoc(ctx context.Context, in *pb.AuthEntryReq, re
ply *EntryReply, done chan struct{}) {
    defer logger.Recover(ctx)
    defer func() {
        done <- struct{}{}
    }()

    pLen := len(in.Contents)
    if pLen == 0 {
        return
    }
......
```

24、大量switch要考虑使用表驱动的方式（23、24联动）

例子：

Go | 复制代码

```go
func (s *AuthService) getAllReqEntryFromContents(contents []*pb.EntryInfo
, req *AuthReqEntryInfo) {
  for _, entry := range contents {
    switch entry.Type {
    case PostType:
      req.post.Contents = append(req.post.Contents, &pb.KbaEntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case QuestionType:
      req.question.Contents = append(req.question.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case AnswerType:
      req.answer.Contents = append(req.answer.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case GroupType:
      req.group.Contents = append(req.group.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case DocType:
      req.doc.Contents = append(req.doc.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case TopicType:
      req.topic.Contents = append(req.topic.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case EventType:
      req.event.Contents = append(req.event.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case SurveyType:
      req.survey.Contents = append(req.survey.Contents, &pb.EntryInfo{
        Id:   entry.Id,
        Type: entry.Type,
      })
    case KnowledgeType:
```

```
        req.knowledge.Contents = append(req.knowledge.Contents, &pb.EntryInfo{
          Id:   entry.Id,
          Type: entry.Type,
        })
      case SuperPageType:
        req.superPage.Contents = append(req.superPage.Contents, &pb.EntryInfo{
          Id:   entry.Id,
          Type: entry.Type,
        })
      default:
        req.invalid.Contents = append(req.invalid.Contents, &pb.EntryInfo{
          Id:   entry.Id,
          Type: entry.Type,
        })
      }
    }
  }
```

语言组织：【代码霰弹式修改：这里的代码设计可扩展性非常差，每新增一个鉴权类型，就需要在结构体中新增一个pb.AuthEntryReq类型，同时在AuthService.AuthEntry,AuthService.getAllReqEntryFromContents等方法中对新增的鉴权类型做响应的修改；

建议：将鉴权功能接口化，每一种鉴权类型对应一个鉴权接口的实现；鉴权实例的构造通过表驱动的方式生成不同的鉴权实例；同时authPost (domain 层)包含 post 的 id 信息，controller 负责从 pb.AuthEntryReq 中提取 post id 信息，实例化成一个 authPost，伪代码如下：

```
// package post

type AuthReq struct {

 ID int64 // id of the requested post
}
// 实现 Authenticator
func (r *AuthReq) Auth(ctx context.Context)(AuthResult, error){
 // r.ID...
}
....
package controller
type Authenticator interface {
```

```
 Auth(ctx context.Context) (AuthResult, error)
}
var authReqConverters map[string]func() Authenticator
type AuthService struct {}
// rpc 入口
func (s *AuthService) AuthEntry(ctx context.Context, in *pb.AuthEntryReq) (*pb.AuthEntryReply,
error) {

 var entries []Authenticatior
 for _, item := range in.Contents {
 convert, ok := authReqConverters[item.Type]
 if !ok {
 continue
 }
 entries = append(entries, convert(in))
 }

 result, err := doTask(ctx, entries)
 return convertToRsp(result), err
}
func doTask(ctx context.Context, entries []Authenticator) (AuthResult, error) {
 var (
 g errgroup.Group
 r AuthResult
 )
 for _, entry := range entries {
 g.Go(func () error {
 result, err := entry.Auth(ctx)
 if err != nil {
 return err
 }
 // Add result to AuthResult
 })
 }
 if err := g.Wait(); err != nil {
 // ...
```

```
 }
 return r, nil
}】
```

25、协程并发问题（用errgroup）

语言组织：【协程并行处理控制的问题：
1、通过 done chan + QueryEntryMax 来控制整个并行流程的执行，比较脆弱，倘若在每个 goroutine 中没有严格执行只写一次 chan 的规则，则会造成死锁问题；
2、固定协程的执行数量（bufNum=10）,代码扩展性差；
3、该处迫使 AuthAnswer，AuthDoc 等函数依赖 reply 这样的包含多个 chan 的入参来返回结果，而 reply 结构只是为了 rpc 返回层面处理结果而使用的一种结构。AuthAnswer 等函数其实只需要返回每个 entry 的鉴权结果(allowed/limited/invalid)，而不应该关心这些结果应该以什么样的结构放到最后的 rpc 返回中。
建议：
使用errgroup包或者trpc.GoAndWait来处理并行逻辑

```
errGrp := errgroup.Group{}
for _, item := range auths {

 errGrp.Go(func() error {
 rsp, err := item(ctx)
 // 处理 rsp
 return err
 })
if err := errGrp.Wait(); err != nil {
 return err
 }
 return finalRsp, nil
}
```

使用trpc.GoAndWait:

```
var funcs []func()error
for _, item := range auths {
 funcs = append(funcs, func()error {
 rsp, err := item(ctx)
 // 处理 rsp
 return err
 })
```

}
if err := trpc.GoAndWait(); err != nil {
 return err
}
return finalRsp, nil】

26、协程返回值处理问题(用errgroup)

语言组织：【方法逻辑处理结果与错误的返回：每一种具体的鉴权逻辑都没有对逻辑错误进行处理（鉴权的错误信息没有返回给调用方，也没有打点监控/日志，也没有注释说明为什么不处理错误）；且方法的最终处理结果通过入参返回给调用方；在通过chan返回处理结果时，在reply中使用了多个chan来返回处理结果；
建议：
在设计函数/方法是考虑对错误处理，尽量避免通过入参来返回处理结果；在使用协程并行处理任务时可以通过errgroup控制并行处理和错误结果；即使在使用chan来处理返回结果时也应该每个goroutine将具体结果写入到一个 chan 中，统一在一个chan中处理返回的结果。
Auth(ctx context.Context)(rsp, error)】

27、结构实体跟逻辑要分开不要写到一起

例子：

Go 复制代码

```go
// Task 测试任务
type Task struct {
  Repo      string `gorm:"default:(-); <-:create"` // 仓库地址
  CommitID  string `gorm:"default:(-); <-:create"` // 提交 ID
  Tool      string `gorm:"default:(-); <-:create"` // 测试工具
  Target    string `gorm:"default:(-); <-:create"` // 测试目标
  Workspace string `gorm:"default:(-); <-:create"` // 工作目录
}


// getInfo 获取 Task 的 Info
func (t *Task) getInfo(ctx context.Context) (*Info, error) {
  op, err := GetOp(ctx)
  if err != nil {
    return nil, err
  }
  return op.InfoGet(t)
}


// save 保存到数据库
func (t *Task) save(ctx context.Context) error {
  op, err := GetOp(ctx)
  if err != nil {
    return err
  }
  return op.TaskSave(t)
}
......
// Submit 提交一个任务
func Submit(ctx context.Context, task *Task) (*Info, error) {
    // 检查 Task 是否已经存在
    info, err := task.getInfo(ctx)
    if err != nil {
        return nil, fmt.Errorf("查询 Task 信息 %v 失败，%w", task, err)
    }
    // 根据 info 的状态分别处理
    switch info.TaskStatus {

    case int32(object.TaskStatus_Unknown):
        // Task 不存在，添加 Task
        if err := task.save(ctx); err != nil {
            return nil, fmt.Errorf("保存 Task %v 失败，%w", task, err)
        }
    case int32(object.TaskStatus_Error):
```

```
            // Task 执行失败，重置状态
            if err := info.ResetStatus(ctx); err != nil {
                return nil, fmt.Errorf("更新 Task 状态失败 %v, %w", info, err)
            }
        default:
            // 直接返回 Info 信息
            return info, nil
        }
        // 再次查询修改后的 Info 信息
        return task.getInfo(ctx)
    }
```

语言组织：【

1. 既然做了model抽象，就不要在model抽象里看出具体实现。指 `gorm:"default:(-); <-:create"` 这一段。
2. model的业务逻辑，可以成为"member func"的。但是，持久化相关的逻辑，不应该成为model下具体struct的"member func"。
   当然，另一个kcp提到，这个pkg的定位就有问题。不好的设计常常会引出更多不好的设计。
3. object.TaskStatus本来是枚举，本来是有类型的。不应该去给它脱去类型变成int32放到info.TaskStatus。数据持久化层应该是直接把model落地，而不是脱去业务信息后落地，多做一个中间层。除非有什么必须要这么做的理由。实际上，及时使用gorm，也是不需要做着一层的。如果确实需要这一层，请在info.TaskStatus处做出合理的comment，甚至留下TODO，以待未来优化。

】

28、事务的能力不应该通过接口方式实现，而应该是实现层去关心的

例子:

```go
// Op Task 对象数据库操作接口
// 该接口对 Task 对象进行各类数据操作
type Op interface {
  // Task 数据库操作
  TaskSave(task *Task) error


  // Info 数据库操作
  InfoGet(task *Task) (*Info, error)
  InfoUpdate(info *Info) error


  // Batch 数据库操作
  // Save 数据库新增一个 Batch 对象
  BatchSave(batch *Batch) error
  // AddTasks 向 Batch 添加 Task
  BatchAddTasks(b *Batch, infos []*Info) error
  // Update 更新
  BatchUpdate(batch *Batch) error
  // GetTasks 获取 batch 关联的任务
  BatchGetTasks(batch *Batch) ([]*Info, error)
}


// Tx Op 事务版
type Tx interface {
  database.Transaction
  Op
}
```

语言组织：【这里Op的抽象和下面Tx的抽象设计有问题。一般来说，我们是针对每个model，给一套注入实现的针对model持久化的持久化实现，比如：

type XXXModel interface {

GetXXX() (data, error) // get函数

SetModelSubItem(data) error // 对模型局部某个数据的持久化落地

}

上面这个设计，和这里有相似之处。但是，要让XXXModel具备事务的能力，应该在实现层面去关心。而不是通过给model添加database.Transaction的几个函数来实现。这里主要在Submit里使用了Transaction的函数，那么Submit也应该抽象进model。所以，domain/task pkg到底是数据持久化（model）层，还是逻辑层？层次也不清晰。把这个pkg做成持久化层，内含事务能力，是一种合适的做

法。这里的interface也不应该叫Op，应该就是task.Store interface{}。
再进一步描述，
1）这里并没有因为做了interface设计而得到任何好处。不会抽象，就不要抽象model。老老实实做个client。或者传统的DAO也是不太好，但是能够被接受的做法。
2）会做抽象，就应该把上面提到的做到位。错误的抽象比不抽象还要害人。】

29、如果switch里的判断条件跟返回值都是在一个包中，就应该定义在包中

例子：

Go 复制代码

```go
func taskStatusConvert(currTaskStatus int) string {
    switch currTaskStatus {
    case admindb.WAIT_APPROVAL:
        return admindb.WAIT_APPROVAL_TEXT
    case admindb.APPROVAL_REJECT:
        return admindb.APPROVAL_REJECT_TEXT
    case admindb.WAIT_RELEASE:
        return admindb.WAIT_RELEASE_TEXT
    case admindb.MAIN_RELEASE:
        return admindb.MAIN_RELEASE_TEXT
    case admindb.GRAY_RELEASE:
        return admindb.GRAY_RELEASE_TEXT
    case admindb.ROLLBACK:
        return admindb.ROLLBACK_TEXT
    default:
        return ""
    }
}
```

语言组织：【WAIT_APPROVAL_TEXT 和 WAIT_APPROVAL 的映射关系应该在 admindb 中定义，而非在外部。此外，应直接使用 adminpb 中定义的对应类型，然后直接使用 .String() 方法获得文字版本。

另外，上下文中实际需要的是对应状态的中文表述，所以这个实现并没有实际意义。】

30、接口的实现应该放到不同的包里的(包职责不单一)

例子：

```go
type Msger interface {
    SendMsg(string) error
}
......
func NewCorpSender(users string) Msger {
    return &corpWx{
        toUsers: users,
    }
}
......
func NewRobotSender(chatid string) Msger {
    return &robotWx{
        chatId: chatid,
    }
}
......
```

语言组织：【git.code.oa.com/rainbow/rainbow/pkg/wechat 属于通用能力，对外直接提供调用企微 API 的接口。
当前包是拼接具体消息并负责调用 wechat api 的业务层。
那么 pkg/wechat 封装得不够彻底，下面的 corpWx 和 robotWx 与业务无关，都应该并入 pkg/wechat.】

31、使用模板简化复杂格式化

例子：

```go
const taskDetail = "http://%s/console/%s/%s/task/detail/%d"


const (
    approvalText = `%s 发起了七彩石审批申请，详情请点击 <a href="%s">这里</a> 进行处理`
    passText     = `%s 审批了你申请的七彩石权限，并已通过`
    refuseText   = `%s 审批了您申请的七彩石权限，但已被拒绝，您可以联系他(她)了解情况`
)


const (
    waitApprovalRobotText  = "您好，%s 提交了发布任务\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】%s\n【描述】%s\n【可审批人】%s\n[查看详情](%s)  [立即审批](%s)"
    approvalPassRobotText  = "您好，%s %s了发布任务\n【原因】%s\n\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】%s\n【描述】%s\n[查看详情](%s)"
    releaseStatusRobotText = "配置发布提醒，当前状态：%s\n\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】 %s\n【描述】%s\n[查看详情](%s)"
)


const (
    waitApprovalCorpText  = "您好，%s 提交了发布任务\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】%s\n【描述】%s\n【可审批人】%s\n<a href='%s'>查看详情</a>  <a href='%s'>立即审批</a>  "
    approvalPassCorpText  = "您好，%s %s了发布任务\n【原因】%s\n\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】%s\n【描述】%s\n<a href='%s'>查看详情</a>"
    releaseStatusCorpText = "配置发布提醒，当前状态：%s\n\n【项目】%s\n【环境】%s\n【分组】%s\n【版本】%s\n【描述】%s\n<a href='%s'>查看详情</a>"
)
......
```

语言组织：【建议使用 text/template 或 http/template 配合上下文结构进行内容生成，避免无意义的 placeholder 和超长的 Sprintf 参数列表。同时避免 `<a href="%s">` 的显式 escape sequence 处理（当前没有，可能存在安全隐患）。】

32、多次出现 switch-case，应根据类型抽象为接口

例子：

Go | 复制代码

```go
func (c ScanResultsCache) fillInstancesPolicies() error {
  for _, instance := range c.instanceDetail {
    if instance.InstanceID == nil {
      continue
    }
    if instance.InstanceRegion == nil {
      continue
    }


    switch *instance.InstanceServiceType {
    case config.TagToServiceType[…]:
      //log.Debugf("get policy for redis instance: %v\n", *instance.InstanceID)
      rsp, err := c.getRedisPolicies(*instance.InstanceRegion,
        "1252068000", *instance.InstanceID)
      if err != nil {…}
      if rsp != nil && rsp.Response != nil {…} else {…}
      //log.Debugf("get policy for redis cluster: %v\n", *instance.InstanceID)
      response, err := c.getRedisClusterPolicies(*instance.InstanceRegion,
        "1252068000", *instance.InstanceID)
      if err != nil {…}
      if response != nil && response.Response != nil {…} else {…}
    case config.TagToServiceType[…]:
      //log.Debugf("get policy for mysql instance: %v\n", *instance.InstanceID)
      rsp, err := c.getMySQLPolicies(*instance.InstanceRegion, *instance.InstanceID)
      if err != nil {
        log.Debugf("getMySQLPolicies  instanceID: %v, err: %v\n", *instance.InstanceID, err)
      }
      if rsp != nil && rsp.Response != nil {
        instance.Policies = append(instance.Policies, rsp.Response.Policies...)
      } else {
        log.Debugf("getMySQLPolicies response.Response is empty  instanceID: %v\n",
          *instance.InstanceID)
      }
    //  【认证信息】case kafka 相关代码已被删除
    default:
      //log.Debugf("wrong service type: %v", *instance.InstanceServiceType)
```

```
52          }
53          time.Sleep(50 * time.Millisecond)
54      }
55
56
57      return nil
58  }
```

语言组织：【根据资源类型，整个流程可以进行一定的抽象
所有switch *instance.InstanceServiceType 的地方都可以做
抽象一个Policy的interface，实现几个实例（redis, mysql等），将实例放在instanceDetail上，这里可以直接调用】

33、方法逻辑重复

例子：

```go
// getRedisClusterPolicies 查询redis集群关联的告警策略
func (c ScanResultsCache) getRedisClusterPolicies(region, appID,
  instanceID string) (*monitor.DescribeAlarmPoliciesResponse, error) {
  dimensionsStr, err := getRedisClusterDimensionStr(appID, instanceID)
  if err != nil {
    return nil, fmt.Errorf("getRedisClusterDimensionStr err: %v", err)
  }
  // "REDIS-CLUSTER"  "redis_mem_edition",  "redisUuid"
  return c.getPolicies(region, "REDIS-CLUSTER", dimensionsStr)
}


// getRedisClusterPolicies 查询redis实例关联的告警策略
func (c ScanResultsCache) getRedisPolicies(region, appID,
  instanceID string) (*monitor.DescribeAlarmPoliciesResponse, error) {
  dimensionsStr, err := getRedisDimensionStr(appID, instanceID)
  if err != nil {
    return nil, fmt.Errorf("getRedisDimensionStr err: %v", err)
  }
  // "REDIS-CLUSTER"  "redis_mem_edition",  "redisUuid"
  return c.getPolicies(region, "redis_mem_edition", dimensionsStr)
}


// getMySQLPolicies 查询MySQL关联的告警策略
func (c ScanResultsCache) getMySQLPolicies(region,
  instanceID string) (*monitor.DescribeAlarmPoliciesResponse, error) {
  dimensionsStr, err := getMySQLDimensionStr(instanceID)
  if err != nil {
    return nil, fmt.Errorf("getMySQLDimensionStr err: %v", err)
  }
  // "REDIS-CLUSTER"  "redis_mem_edition",  "redisUuid"
  return c.getPolicies(region, "cdb_detail", dimensionsStr)
}
```

语言组织：【这几个函数是复制粘贴的典型（注释都没改）
即使不做上面的更好的抽象，这三个函数也可以简单合并成一个函数，输入demensionstr，之后都是调用getpolicies】

34、反复创建client

例子：

```go
// getPolicies 查询实例关联的告警策略
func (c ScanResultsCache) getPolicies(region, namespace,
  dimensionsStr string) (*monitor.DescribeAlarmPoliciesResponse, error) {
  cpf := profile.NewClientProfile()
  cpf.HttpProfile.Endpoint = monitorEndpoint
  client, _ := monitor.NewClient(c.credential, region, cpf)
  request := monitor.NewDescribeAlarmPoliciesRequest()
  request.Module = common.StringPtr("monitor")
  request.Namespaces = append(request.Namespaces, common.StringPtr(namespace))
  request.Dimensions = common.StringPtr(dimensionsStr)
  request.MonitorTypes = common.StringPtrs([]string{"MT_QCE"})
  response, err := client.DescribeAlarmPolicies(request)
  if err != nil {
    return nil, err
  }
  return response, nil
}
```

语言组织：【这个client不需要重复建立（跟上次trpc new client一样），并且不好测试，依赖注入实现】

35、client 操作细节与业务逻辑耦和，缺少分层

例子：

Go | 复制代码

```go
func (c ScanResultsCache) deletePolicy(region, policyID, uniqueID string) error {
  cpf := profile.NewClientProfile()
  cpf.HttpProfile.Endpoint = monitorEndpoint
  client, _ := monitor.NewClient(c.credential, region, cpf)
  request := monitor.NewUnBindingPolicyObjectRequest()
  request.Module = common.StringPtr("monitor") // 固定值
  request.GroupId = common.Int64Ptr(0)         // 0是填充，无意义
  request.PolicyId = common.StringPtr(policyID)
  request.UniqueId = append(request.UniqueId, common.StringPtr(uniqueID))
  response, err := client.UnBindingPolicyObject(request)
  if err != nil {
    return fmt.Errorf("an api error has returned: %v, rsp: %v", err, response)
  }
  return nil
}
......
// addPolicy 给单个实例添加告警策略
func (c ScanResultsCache) addPolicy(region, dimensionsStr, policyID string) error {
    cpf := profile.NewClientProfile()
    cpf.HttpProfile.Endpoint = monitorEndpoint
    client, _ := monitor.NewClient(c.credential, region, cpf)

    request := monitor.NewBindingPolicyObjectRequest()
    request.Module = common.StringPtr("monitor") // 固定值
    request.GroupId = common.Int64Ptr(0)         // 0是填充，无意义
    request.PolicyId = common.StringPtr(policyID)
    request.Dimensions = []*monitor.BindingPolicyObjectDimension{
        {
            Region:     common.StringPtr(region),
            Dimensions: common.StringPtr(dimensionsStr),
        },
    }
    response, err := client.BindingPolicyObject(request)
    if err != nil {
        return fmt.Errorf("an api error has returned: %v, rsp: %v", err, response)
    }
    return nil
}
......
```

语言组织：【这个client上有各种操作，可以独立抽取一个类放各种操作的实现，现在在这个client上不同类型的req resp与主逻辑混】

36、多次在函数内定义结构体

例子：

Plain Text | 复制代码

```
case config.TagToServiceType["REDIS"]:
      type DimensionsValue struct {
        AppID      string `json:"appid"`
        InstanceID string `json:"instanceid"`
      }
      dimensions := DimensionsValue{
        AppID:      "1252068000",
        InstanceID: *instance.InstanceID,
      }
      dimensionsStr, err := json.Marshal(dimensions)
      if err != nil {
        return fmt.Errorf("json marshal err: %v", err)
      }
......
    case config.TagToServiceType["MYSQL"]:
      type DimensionsValue struct {
        InstanceID string `json:"uInstanceId"`
      }
      dimensions := DimensionsValue{
        InstanceID: *instance.InstanceID,
      }
      dimensionsStr, err := json.Marshal(dimensions)
      if err != nil {
        return fmt.Errorf("json marshal err: %v", err)
      }
 ......
```

语言组织：【结构体的定义只是为了 marshal，且不仅使用一次，因此应该定义在函数的外部。】

37、使用结构体成员变量作为逻辑中转

例子：

```
oldInstances, newInstances, err := c.updateInstances(config.GlobalConfig.CaredTags)
  if err != nil {
    log.Debugf("getAllInstanceList err: %v\n", err)
    return nil, err
  }
  log.Debugf("getAllInstanceList done\n")


  if err := c.getInstancesTags(); err != nil {
    log.Debugf("getInstancesTags err:%v\n", err)
  }
  log.Debugf("getInstancesTags done\n")


  if err := c.fillInstancesPolicies(); err != nil {
    log.Debugf("fillInstancesPolicies err:%v\n", err)
  }
  log.Debugf("fillInstancesPolicies done\n")


  if err := c.addPolicyForInstances(newInstances); err != nil {
    log.Debugf("addPolicyForInstances err: %v\n", err)
  }
  log.Debugf("addPolicyForInstances done\n")


  if err := c.deleteInstancePolicies(oldInstances); err != nil {
    log.Debugf("deleteInstancePolicies err: %v\n", err)
  }
  log.Debugf("deleteInstancePolicies done\n")
......
```

语言组织：【这段逻辑使用 struct 进行中转，建议使用返回的值前后传递。

insts := c.listInstances()

insts = c.filterInstances(insts)

insts = c.filterUnmonitoredInstances(insts)

c.addMonitorForInstances(insts)】

38、直接依赖全局配置 config 包

例子：

Plain Text 复制代码

```
switch serviceType {
  case config.TagToServiceType["REDIS"]:
    policyID = config.TagToDataSource["REDIS"].PolicyID
    dimensions := RedisDimensionsValue{
      AppID:      "1251316161",
      InstanceID: instanceID,
    }
    dimensionsStr, err := json.Marshal(dimensions)
    request.Dimensions = []*monitor.DescribeBindingPolicyObjectListDimension{
      {
        Dimensions: common.StringPtr(string(dimensionsStr)),
      },
    }
    if err != nil {
      return "", fmt.Errorf("json marshal err: %v", err)
    }
  case config.TagToServiceType["MYSQL"]:
    policyID = config.TagToDataSource["MYSQL"].PolicyID
    dimensions := MySQLDimensionsValue{
      InstanceID: instanceID,
    }
    dimensionsStr, err := json.Marshal(dimensions)
    request.Dimensions = []*monitor.DescribeBindingPolicyObjectListDimension{
      {
        Dimensions: common.StringPtr(string(dimensionsStr)),
      },
    }
    if err != nil {
      return "", fmt.Errorf("json marshal err: %v", err)
    }
  default:
    log.Debugf("wrong service type: %v", serviceType)
  }
```

语言组织：【代码中多处使用全局配置读取，配置读取不应该在函数内部直接使用外部包的全局配置，建议使用依赖注入的方式读配置，减小配置作用域。即使是使用外部包内的全局配置，也应该提供包级别的Get() 、Load()方法，而不是直接调用外部包内的全局配置，并且可以看到直接读取的有这种全局配置的map，若配置有对应的热更新逻辑，可能引发并发读写问题，对应的依赖配置应该提供相应的Get()接口，内部合理进行加锁等原子实现，而不是直接读写。

并且这里对于配置中map的读取没有判断key是否存在，直接读取对应结构的子成员，如果对象不存在，

可能会有空指针问题，建议读取外部依赖的map，应先判断值是否存在，再使用，或者相关逻辑封装成对应的get函数。】

39、分页数据要不要先存储再取出，重复逻辑

例子：

Plain Text | 复制代码

```
func getRowkeys(ctx context.Context, uid string, puinSlice []string, timestamp int64, pageNo uint32, pageSize uint32, specialRowkey string) (rowkeyInfoSlice []*list_pb.RowKeyInfo, newNumber int64, IsEnd uint32, err error) {
  rowkeyInfoSlice = make([]*list_pb.RowKeyInfo, 0, pageSize)
  key := uid + "_rowkeysinfo"
  log.Info("GetKbDynamicInfo getRowkeys user key: ", key)
  redisRowkeyStringSlice := make([]string, 0, 0)
  if pageNo == 1 {
    var newRowkeyInfoSlice []*list_pb.RowKeyInfo
    newRowkeyInfoSlice, err = getEsRowkeys(ctx, puinSlice, 1, 2000)
    if err != nil {
      log.Error("GetKbDynamicInfo getRowkeys err: ", err)
      return
    }


    if len(newRowkeyInfoSlice) == 0 {
      IsEnd = 1
      return
    }


    _, err = getRedisClient().ZRemRangeByRank(key, 0, 3000).Result()
    if err != nil {
      log.Error("GetKbDynamicInfo getRowkeys err: ", err)
      return
    }
    redisRowkeyInfoSlice := make([]*redis.Z, 0, len(newRowkeyInfoSlice))
    for _, rowkeyInfo := range newRowkeyInfoSlice {
      if rowkeyInfo.Rowkey == specialRowkey {
        continue
      }
      rowkeyAndTime := &RowkeyAndTime{
        Rowkey: rowkeyInfo.Rowkey,
        Time:   rowkeyInfo.PubTime,
      }
      var value []byte
      value, err = json.Marshal(rowkeyAndTime)
      if err != nil {
        log.Error("GetKbDynamicInfo getRowkeys err: ", err)
        return
      }
      z := &redis.Z{
        Score:  float64(rowkeyInfo.PubTime),
```

```
        Member: string(value),
      }
      redisRowkeyInfoSlice = append(redisRowkeyInfoSlice, z)
    }
    _, err = getRedisClient().ZAdd(key, redisRowkeyInfoSlice...).Result()
    if err != nil {
      log.Error("GetKbDynamicInfo getRowkeys err: ", err)
      return
    }
    _, err = getRedisClient().Expire(key, 48*time.Hour).Result()
    if err != nil {
      log.Error("GetKbDynamicInfo getRowkeys err: ", err)
      return
    }


    yesterday := time.Now().Add(-time.Hour * 24).Unix()
    log.Info("yesterday: ", yesterday)
    log.Info("timestamp: ", timestamp)
    if timestamp < yesterday {
      timestamp = yesterday
    }
    newNumber, err = getRedisClient().ZCount(key, strconv.Itoa(int(timest
amp+1)), strconv.Itoa(int(time.Now().Unix()))).Result()
    log.Info("newNumber: ", newNumber, " timestamp: ", timestamp)
    if err != nil {
      log.Error("GetKbDynamicInfo getRowkeys err: ", err)
      return
    }
  }
  redisRowkeyStringSlice, err = getRedisClient().ZRevRange(key, int64((pa
geNo-1)*pageSize), int64(pageNo*pageSize-1)).Result()
  if err != nil {
    log.Error("GetKbDynamicInfo getRowkeys err: ", err)
    return
  }
  if len(redisRowkeyStringSlice) < int(pageSize) {
    IsEnd = 1
  }
  log.Info("size: ", len(redisRowkeyStringSlice))
  if specialRowkey != "" {
    rowkeyInfo := &list_pb.RowKeyInfo{Rowkey: specialRowkey}
    rowkeyInfoSlice = append(rowkeyInfoSlice, rowkeyInfo)
    log.Info("red point: ", specialRowkey)
  }
  for _, rowkeyString := range redisRowkeyStringSlice {
    rowkeyAndTime := &RowkeyAndTime{}
    err = json.Unmarshal([]byte(rowkeyString), rowkeyAndTime)
```

```
      if err != nil {
        log.Info("contentt: ", rowkeyString)
        log.Error("GetKbDynamicInfo getRowkeys err: ", err)
        return
      } else {
        rowkeyInfo := &list_pb.RowKeyInfo{}
        rowkeyInfo.Rowkey = rowkeyAndTime.Rowkey
        rowkeyInfo.PubTime = rowkeyAndTime.Time
        rowkeyInfoSlice = append(rowkeyInfoSlice, rowkeyInfo)
      }
    }
    if len(rowkeyInfoSlice) > 1 && specialRowkey != "" {
      rowkeyInfoSlice[0].PubTime = rowkeyInfoSlice[1].PubTime
    }
    log.Info("GetKbDynamicInfo getRowkeys rowkeyInfoSlice: ", rowkeyInfoSli
ce)
    return
}
```

语言组织：【在第一页数据取回填入redis后，其实已经可以返回正确数据，这个if后的逻辑又重新从redis中读了一次，这个设计有问题。】