golang开发规范

一 . 项目目录结构规范

/cmd　　cmd 目录中存储的都是当前项目中的可执行文件，如果我们的项目是一个 grpc 服务的话，可能在 /cmd/server/main.go 中就包含了启动服务进程的代码，编译后生成的可执行文件就是 server

/internal　　internal　私有代码存放位置，internal 中被go编译支持，当我们在其他项目引入包含 internal 的依赖时会发生错误。

/pkg　　这个目录中存放的就是项目中可以被外部应用使用的代码库，其他的项目可以直接通过 import 引入这里的代码，所以当我们将代码放入 pkg 时一定要慎重，不过如果我们开发的是 HTTP 或者 RPC 的接口服务或者公司的内部服务，将私有和公有的代码都放到 /pkg

　　中也没有太多的不妥

/api　　目录中存放的就是当前项目对外提供的各种不同类型的 API 接口定义文件了

/scripts　　存放辅助相关的脚本文件，如sh

/vendor　　Go1.1.3 可以使用模块代理服务器，不建议存在该目录，其他历史版本可以使用，用于存放第三方依赖。

/web　　用于存放静态web资产

/configs　　配置文件模板或默认配置。

/init　　系统init (systemd, upstart, sysv)和进程管理器/监控器(runit, supervise ord)配置。

/build　　打包和持续集成相关。

/test　　单元测试相关

/docs　　存放文档相关

私有代码详解

/internal

　　/app　# 存放项目相关代码

　　/pkg　# 应用程序共享的代码可以放在这里

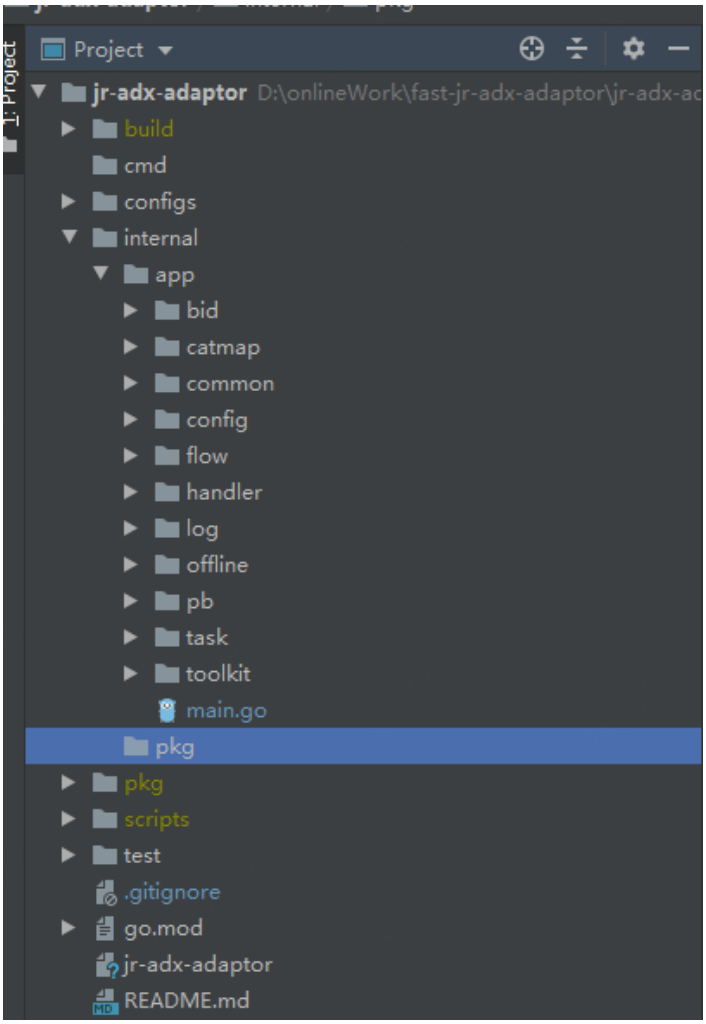
注意点

　go项目中原则上不允许使用src目录，原因其实是 Go 语言的项目在默认情况下都会被放置到 **$GOPATH/src** 目录下，这个目录中存储着我们开发和依赖的全部项目代码，如果我们在自己的项目中使用 **/src** 目录，该项目的 **PATH** 中就会出现两个 **src**：

　例如: $GOPATH/src/github.com/draveness/project/src/code.go

社区规范参考: https://github.com/golang-standards/project-layout

目录结构示例(参考adaptor)



二. 注释

- 应用下的README.md文件,说明主要功能和迭代;
- 每个包下应有对应的README.md文件，说明该包下文件主要功能，重要的迭代维护到其中。

三. 命名

包命名与引用

包名全部小写，不用复数， 避免和官方包或者三方包名的冲突

包引用的顺序,

包的循环调用（在编译包的时候可校验出，但为提高效率，在开发阶段就应该避免

函数与变量名

- 函数名采用 MixedCaps ， func GetTagFromDmp( )
- 本地变量声采用 短变量 := 的方式

- 需要注意的，在对接流量做协议适配（json格式的协议），结构体的命名采用如下方式：

IFlytecRequest

IFlytecRequestBid

IFlytecRequestBidSeats

IFlytecRequestBidSeatsAds

IFlytecRequestBidSeatsAdsImg

IFlytecResponse

四. error的处理

对于调用外部包函数的，对返回值中的error一定要处理。避免panic。调用recover函数也应在出现异常后及时处理。严禁在调用外部函数时，不对返回的error做处理。

```
v, err := GetSomeThing()
if err != nil {
}
```

五. 数组

数组取值必须判断数组长度，严禁角标越界。

六. string 字符串

字符串引号尽量用 ``， 避免因为转义出现异常

字符串拼接

关于拼接字符串的3种方式

- 利用 运算符 + 进行拼接，不推荐；
- fmt.Sprintf()进行拼接，在不同类型数据并存的情况下常用；
- strings.Join() ，拼接的数据都为string类型的时候是效率比较高的方式；
- bytes.Buffer效率和内存上都比较有保障的方式。

对于流量高且并发高的应用，字符串拼接尽量避免使用如下两种方式：

1. 1. a+b；
2. 2. fmt.Sprintf("a%s",b)。

主要问题是频繁开辟内存空间。

推荐如下方式：

```
func ContactString(item ...string) string {
        var item_buffer bytes.Buffer
        for _, v := range item {
                item_buffer.WriteString(v)
        }
        return item_buffer.String()
}
//
ContactString("s","_","123456")
```

string 和[]byte

string.go文件中的下面两个函数内存开销很大：

rawstringtmp() —> slicebytetostring() 。

string底层存储依然为[]byte，如果只是需要转换类型而不是修改数据，不建议采用string() 的方式进行转化 。

参考转换函数如下：

```
//
func str2bytes(s string) []byte {
        x := (*[2]uintptr)(unsafe.Pointer(&s))
        h := [3]uintptr{x[0], x[1], x[1]}
        return *(*[]byte)(unsafe.Pointer(&h))
}

func bytes2str(b []byte) string {
        return *(*string)(unsafe.Pointer(&b))
}
```

七. map

- 对于已知map长度的，在创建map时指定map的len；
- 对于未知map长度的，上线后续应预估好map的len；

对于hashmap中不用的数据，进行delete调用并不会释放内存，而是保留内存在需要的时候进行复用。

hashmap内存分配以倍数扩增。扩增后即使map内元素减少，也不会释放掉内存。比如当前已有内存为128M，由于要新增1M的数据，那么分配给它的内存会是128M，这样它的内存大小就是256M了，并且不会释放。

八. 关于调用redis

1. 对于频繁的incrby或者hincrby操作，可定时或者批量操作，避免r2m应用的tps过高。这里需要注意的是，即使是使用pipeline，到了操作redis节点这一层，并不能减少tps。
2. 尽量降低热key tps的原因是， 目前golang，lua等应用采用的是代理接入r2m，目前线上只有部分应用切换了来客的独立代理，对于接入公共代理的情况，tps过高可能会影响到应用。
3. 避免集合过大设置过期时间 （热key，大key），尽量减少无意义的重复设置过期时间。

```
redis_cli.hincrby(key,field,1)
redos_cli.expire(key86400)
```

可只在第一次自增设置过期时间，例如：

```
result := redis_cli.hincrby(key,field,1)
if result==1 {
        redos_cli.expire(key86400)
}
```

需求在指定的时间（unix时间戳）过期时，可利用 **EXPIREAT** key 1355292000 。

格式化key ，字段为空可不再从redis获取值（热key）。

```
key := fmt.Sprintf("key_%s",value)
redis_cli.get(key)
```

修改为：

```
if value != `` {
        key := fmt.Sprintf("key_%s",value)
        redis_cli.get(key)
}
```

已弃的业务逻辑有涉及 热key 大key 的，及时下线。

九. 测试