Go语言在游戏项目中的工程实践

三大哲学问题

我是谁？

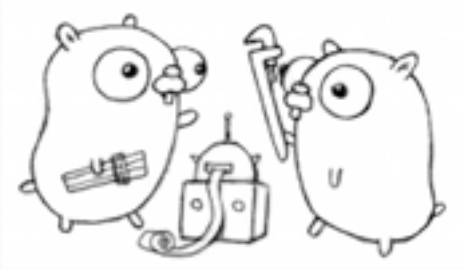
- 网名叫达达
- 是一名程序员
- 是一名创业者
- 从事游戏行业

我从哪来？

- 我来自厦门
- 我们公司叫真有趣
- 是一家游戏研发公司

我要到哪里去？

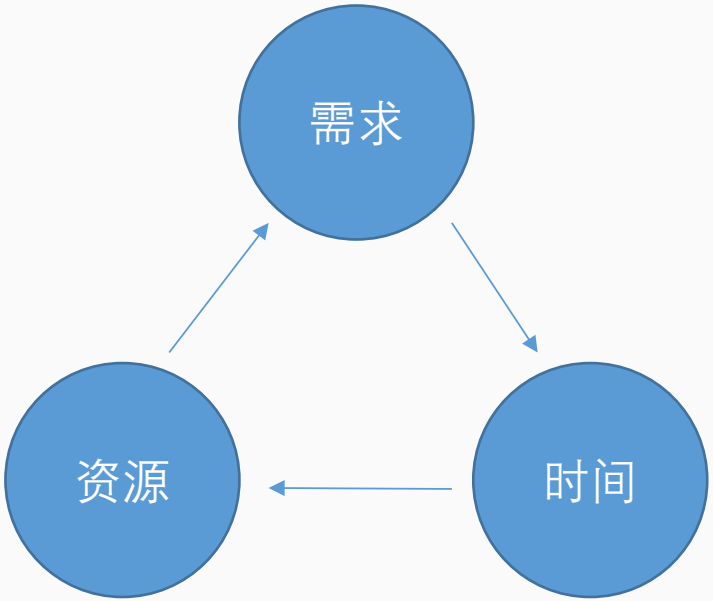
- 我要去ECUG
- 分享Go的工程经验
- 讲错误和异常处理
- 讲interface的应用
- 讲去DSL的尝试



先从工程说起

软件工程三要素

- 需求
  - 提供哪些功能
  - 数据量是怎样的规模
  - 可靠性要求是什么样的
- 时间
  - 什么时候发布
  - 持续多长更新周期
- 资源
  - 多少人
  - 什么样能力的人
  - 多少资金



游戏行业的软件工程

- 需求不确定性很大
- 通常会有时间结点的要求
  - 长假前必须上一批拉活跃的运营活动
  - 广告资源联系好了，务必XX时候做好开服准备
  - 某某游戏很火，大家都在山寨，先下手为强
- 稳定性和实时性要求双高
  - 正在导量，千万不要卡，不要挂
  - 花几十万买的一身极品装备，可不能随便就没了
- 人员普遍缺乏工程经验
  - 原来做XXX，很兴趣游戏所以转行
  - 之前做的项目基本上都跪了

保持简单

- 门槛低
- 实现快
- 验证快
- 迭代快
- 容易说清楚
- 不容易出错
- 出错了也比较容易定位

我采取的策略

- 自动映射数据库的缓存系统
  - 降低开发难度，运行效率让框架负责
  - 让程序员把主要精力放在业务逻辑开发
- 贯彻速错理念
  - 让程序员以简洁的方式开发业务逻辑
  - 尽早的暴露问题
- 简单的业务模块管理
  - 无心智负担的项目规范
  - 代码写得不好只会恶心自己

错误和异常处理

错误

- Go提供了error类型用来表达错误
- Go语法上允许函数有多个返回值
- 返回error的设计在runtime中随处可见

```
import “errors”

func EvenNumAdd(a, b int) (int, error) {
      if a %2 != 0 || b % 2 != 0 {
            return 0, errors.New(
                  “parameter not even number”,
            )
      }
      return a + b, nil
}
```

```
func main() {
	a, b := 1, 2
	c, err := EvenNumAdd(a, b)
	if err != nil {
		fmt.Println(err.Error())
	} else {
		fmt.Printf(“%d + %d = %d”, a, b, c)
	}
}
```

错误处理

- 有时候只是判断nil是不够的
- 有时候只给一个字符串信息是不够

```
// 全局变量
var NotEvenNumErr = errors.New(
    “not even number”,
)
```

```
// 判断错误类型
if err == NotEvenNumErr {
    // . . .
}
```

```
type NotEvenNumErr struct {
        Name      string
        Value     int
}
// 实现error接口
func (e *NotEvenNumErr) Error() string {
        return fmt.Sprintf(
                “%s = %d not even number”,
                e.Name,
                e.Value,
        )
}
```

```
return nil, &NotEvenNumErr{“a”, a}
```

异常

- 运行时抛出
  - 数组越界
  - 空对象
  - 类型断言失败
- panic() 手动抛出
- defer() + recover() 捕获
- 不捕获进程就会退出

```
func EvenNumAdd(a, b int) int {
    if a % 2 != 0 || b % 2 != 0 {
        // 参数可以是任何类型
        panic(NotEvenNumErr)
    }
    return a + b
}
```

```
func main() {
	defer func() {
		if err := recover(); err != nil {
			fmt.Printf(“%v”, err)
		}
	}()

	a, b := 1, 2
	c := EvenNumAdd(a, b)
	fmt.Printf(“%d + %d = %d”, a, b, c)
}
```

速错实践

- 只处理业务上定义的错误，不做多余的事
- 分清会话级别的异常和业务级别的异常
- 所有goroutine都应有明确的入口
- 所有入口都应有recover()和日志记录
- 会话入口只记录日志，不抛出异常
- 业务入口记录日志后，抛出异常让进程崩溃

以装备强化为例

```
if player.Coins < cost {
        return NotEnoughtMoneyError
}

if len(strengthenQueue) <= queue {
        return StrengthQueueOutOfIndexError
}

q := strengthenQueue[queue]
if q.EndTime > time.Now().Unix() {
        return InCoolDownTimeError
}

playerEquip = db.LookupPlayerEquip(playerEquipID)
if playerEquip == nil {
        return EquipNotExistsError
}
```

速错的写法

- 错误的客户端逻辑才会触发fatal和panic
- 不必为外挂开发者提供友好的错误提示

```
fatal.When(player.Coins < cost)

q := strengthenQueue[queue]

fatal.When(q.EndTime > time.Now().Unix())

playerItem := db.LookupPlayerItem(playerItemID)

// 如果playerItem是nil，接下去的业务逻辑访问它，将会panic
```

实践总结

- 需要让用户做出响应的才需要管
- 模块调用者是模块开发者的用户
- 客户端开发者是服务端开发者的用户
- 服务端开发者是自己的用户

interface的应用

interface是Go的核心

- interface之于Go就像指针之于C
- interface的实现是隐式的
- interface是可组合的

```
type MyInterface interface {
     DoSomthing()
}

type MyImplement struct {
}

// 偷偷的就实现了MyInterface
func (m *MyImplement) DoSomthing() {
}
```

```
// 显式的声明
var _ MyInterface = (*MyImplement)(nil)
```

```
// “结构体”和“结构体指针”是不一样的
// 这句话将导致编译失败
var _ MyInterface = MyImplement{}
```

io包是最好的示例

- io包定义了实现io所需的基本接口
- 通过基本接口组出不同的复合io接口
- os包实现了文件io
- net包实现了网络io
- bytes包实现了内存io
- bufio包实现了带缓冲的io
- bufio包利用接口解耦了底层io实现

```
type Reader interface {
     Read([]byte) (int, error)
}

type Writer interface {
     Writer([]byte) (int, error)
}

type Closer interface {
     Close() error
}
```

```
type ReadWriter interface {
      Reader
      Writer
}

type ReadCloser interface {
      Reader
      Closer
}

type ReadWriteCloser interface {
      Reader
      Writer
      Closer
}
```

依赖注入

- 业务模块不像runtime，交叉引用很常见
- 但是Go不允许package之间交叉引用
- 所以我们需要换一种思路
- 业务模块只要声明它想要什么
- 要的东西从哪里来不用业务模块管

```
package module

var Player IPlayer

type IPlayer interface {
     DecreaseCoins()
}

var Equip Iequip

type Equip interface {
     GiveEquip()
}
```

```
package player

import “module”

func init() {
	module.Player = &Player{}
}

type Player struct {}

// 玩家注册时送装备
func (p *Player) PlayerRegister() {
	module.Equip.GiveEquip()
}
```

```
package equip

import “module”

func init() {
	module.Equip = &Equip{}
}

type Equip struct {}

// 升级装备时扣钱
func (p *Equip) UpgradeEquip() {
	module.Player.DecreaseCoins()
}
```

实践总结

- 接口在Go语言中是非常重要的，要重点掌握
- 接口的运用是很灵活的，脑筋要会急转弯
- 解耦并不需要很厚重的封装
- 可以恶心自己，但不要恶心别人

去DSL的尝试

DSL的问题

- 功能是隐藏在语法和工具参数背后的
- 开发者要在DSL和原生语言间人肉映射
- 开发过程中需要反复在不同工具间切换

代码本身就是文档

```
type Status int8

const (
        STATUS_ONLINE  Status = 0 // 在线
        STATUS_OFFLINE Status = 1 // 离线
)

// 获取玩家简要信息参数
type GetPlayerShortInfoArgs struct {
        PlayerIds []int64 // 玩家列表
}

// 获取玩家简要信息返回
type GetPlayerShortInfoReply struct {
        PlayerShortInfos []PlayerShortInfo // 玩家简要信息列表
}

// 玩家简要信息
type PlayerShortInfo struct {
        RoleId      int32  // 主角模版ID
        Level       int16  // 主角等级
        OfflineTime int64  // 离线时间
        Status      Status // 在线状态
}
```

定义格式

- 简单类型
  - 整形
  - 浮点型
- 复合类型
  - 字符串
  - 数组
  - slice
  - map
  - 指针
  - 结构体

基于AST的实现

- go/parser分析类型
- go/doc分析注释
- text/template做代码生成
- 配合go generate使用
- 包引用关系不好分析
- 类型转义不好分析
- 持续增加功能的时候代码变得异常复杂
- go/types包也许可以帮忙（有待进一步实践）

基于反射的实现

- 实现起来简单多了
- 代价是要注册类型
- 并且要编译两次代码
- 关闭编译优化可以缓解一下

顺便优化了RPC

- Gob支持BinaryMarshaler和BinaryUnmarshaler接口
- 生成对应方法，类型的序列化和反序列化就被接管了
- 基于原生rpc包，稍微封装一下就得到了符合项目需要的RPC

```
//
// 获取玩家简要信息
//
func (this *PlayerRPC) GetPlayerShortInfo(
    args  *GetPlayerShortInfoArgs,
    reply *GetPlayerShortInfoReply,
) error {
    // ...
}
```

实践总结

- 适合服务端只用Go以及客户端语言单一的项目
- 多语言协作的项目还是推荐Protobuf、JSON
- 可视化也是一种选择
  - 想想边预览边编辑markdown的体验