Go语言游戏项目应用情况汇报

厦门真有趣信息科技有限公司

达达

项目介绍

- 2012年 - 2015年，仙侠道网页版，使用Go语言替代神仙道时期的Erlang，开发流程和主要架构不变

- 2014年 - 2015年，仙侠道手机版，在原有架构基础上拆分游戏逻辑服务器，分离互动功能

游戏服务端的挑战

- 请求频繁
- 实时性要求高（百毫秒的延迟便可被感知）
- 开发效率要求高（每周一更）
- 运维效率要求高（最好是别运维。。。）

我们做了哪些事情

- 通讯层：协议描述语言以及代码自动生成
- 业务层：顺序结构以及接口注册
- 数据层：映射MySQL的内存数据库以及代码自动生成

通讯层

- 通讯协议描述文档的格式选择
  - XML、JSON
  - Protobuf
  - 自定义语法
  - 可视化编辑

```
// 玩家模块
mod player = 0
{
    type login_status enum8 {
        FAILED     = 0 // 登录失败
        SUCCEED    = 1 // 登录成功
        FIRST_TIME = 2 // 首次登录
    }

    //
    // 玩家登录
    //
    api login = 0 {
        in {
            user : text // 平台帐号，最大长度100字符
        }
        out {
            status    : login_status // 登录返回结果
            player_id : int64        // 玩家在游戏中的ID
        }
    }
```

通讯协议描述文档片段

```go
type LoginStatus int8

const (
    LOGIN_STATUS_FAILED      LoginStatus = 0
    LOGIN_STATUS_SUCCEED     LoginStatus = 1
    LOGIN_STATUS_FIRST_TIME LoginStatus = 2
)

type Login_In struct {
    User []byte
}

type Login_Out struct {
    Status   LoginStatus
    PlayerId int64
}
```

```go
func (this *Login_In) ReadInBuffer(buffer *link.InBuffer) {
    this.User = buffer.ReadBytes(int(buffer.ReadUvarint()))
}

func (this *Login_In) WriteOutBuffer(buffer *link.OutBuffer) error {
    buffer.WriteUint8(0)
    buffer.WriteUint8(0)
    buffer.WriteUvarint(uint64(len(this.User)))
    buffer.WriteBytes(this.User)
    return nil
}

func (this *Login_In) OutBufferSize() int {
    size := 2
    size += link.UvarintSize(uint64(len(this.User)))
    size += len(this.User)
    return size
}
```

通讯协议解包封包代码片段

业务层

- 从结构上屏蔽所有可能发生的阻塞
  - 文件读写
  - RPC调用
  - chan阻塞
- 尽可能提高响应速度
  - 数据离CPU越近越好
  - 避免数据复制和大集合遍历
- 通过注册接口防止循环引用

```go
package module

// 玩家模块
type PlayerModule interface {
    // 扣除铜钱
    DecreaseCoins(num int)
}

// 物品模块
type ItemModule interface {
    // 购买物品
    BuyItem()
}

// 这些是接口的具体实现，等待外部主动注册进来，
// 这样module包永远是被引用的，不会出现递归引用问题。
var (
    Player PlayerModule
    Item   ItemModule
)
```

业务模块公共接口声明

```go
package player

import "log"
import "server1/module"

type PlayerModule struct {
}

// 在初始化的时候将模块注册到module包
func init() {
    module.Player = PlayerModule{}
}

// 扣除铜钱
func (player PlayerModule) DecreaseCoins(num int) {
    log.Printf("DecreaseCoins(%d)", num)
}
```

业务模块接口实现

```go
package item

import "server1/module"

type ItemModule struct {
}

// 在初始化的时候将模块注册到module包
func init() {
    module.Item = ItemModule{}
}

func (item ItemModule) BuyItem() {
    module.Player.DecreaseCoins(100)
}
```

业务模块接口调用

数据层

- 玩家数据库切片，减小查询时的集合
- 支持内存事务
- 以事务为单位同步到数据库
- 支持Redo、Undo以及数据挖掘的同步日志
- GC优化

```go
type PlayerRole struct {
    Id      int64 // 玩家角色ID
    Pid     int64 // 玩家ID
    RoleId int8  // 角色模板ID
    Level   int16 // 等级
    Exp     int64 // 经验
}

func (db *Database) SelectPlayerRole(callback func(*PlayerRoleRow)) {
    row := &PlayerRoleRow{}
    for crow := db.tables.PlayerRole; crow != nil; crow = crow.next {
        row.set(crow)
        callback(row)
        if row.isBreak {
            break
        }
    }
    row.c = nil
}
```

内存数据库代码片段

```go
// 在一个内存数据库事务中执行一段代码，执行过程出错，内存数据将被回滚
// 所有事务被顺序执行，等同于MySql串行事务隔离界别
func (db *Database) Transaction(work func()) {
    db.lock.Lock()
    defer db.lock.Unlock()

    // 事务控制
    defer func() {
        if err := recover(); err == nil {
            db.commit(info)
        } else {
            db.rollback()
            panic(TransError{err})
        }
    }()

    work()
}
```

内存数据库事务（示意）

```go
func (this *PlayerInsertLog) Rollback() {
    if this.db.tables.Player != this.CNew {
        panic("Bad PlayerInsertLog")
    }
    this.db.tables.Player = nil
    C.FreePlayer(this.CNew)
}

func (this *PlayerDeleteLog) Rollback() {
    if this.db.tables.Player != nil {
        panic("Bad PlayerDeleteLog")
    }
    this.db.tables.Player = this.COld
}

func (this *PlayerUpdateLog) Rollback() {
    if this.db.tables.Player != this.CNew {
        panic("Bad PlayerUpdateLog")
    }
    this.db.tables.Player = this.COld
    C.FreePlayer(this.CNew)
}
```

内存数据库事务回滚

```
{"Time":1429494711, "API":"make_item", "Pid":123, "Actions": [
    {"Type":"Update", "Table":"player_info",
        "OldData":{ "pid":123, "coins":2000 },
        "NewData":{ "pid":123, "coins":1000 }
    },
    {"Type":"Delete", "Table":"player_item",
        "OldData":{ "id":1, "pid":123, "item_id":100 }
    },
    {"Type":"Delete", "Table":"player_item",
        "OldData":{ "id":2, "pid":123, "item_id":200 }
    },
    {"Type":"Insert", "Table":"player_item",
        "NewData":{ "id":3, "item_id":789, "item_level":0 }
    }
]}
```

事务日志（示意）

```lua
-- 查询玩家体力变化
-- 用法: -table=player_physical -s="2013-10-10 00:00:01" -pid=12884902017

function count(t, n, o)
    if t.table == 'player_physical' then
        print(
            os.date("%x %X", t.op_time),
            n.value - o.value,
            os.date("%x %X", o.update_time),
            os.date("%x %X", n.update_time)
        )
    end
end

function last()
    io.flush()
end
```

使用*lua*脚本对同步日志进行数据挖掘

```c
typedef struct PlayerRole {
    int64_t Id;
    int64_t Pid;
    int8_t RoleId;
    int16_t Level;
    int64_t Exp;
    struct PlayerRole* next;
} PlayerRole;

extern PlayerRole* NewPlayerRole();
extern void FreePlayerRole(PlayerRole*);
```

内存数据库的*GC*优化 - *CGO*

```go
type PlayerRoleRow struct {
    c       *C.PlayerRole
    isBreak bool
}

func (row *PlayerRoleRow) Get() *PlayerRole {
    return &PlayerRole{
        Id:     int64(row.c.Id),
        Pid:    int64(row.c.Pid),
        RoleId: int8(row.c.RoleId),
        Level:  int16(row.c.Level),
        Exp:    int64(row.c.Exp),
    }
}
```

内存数据库的GC优化 - 数据读取

```go
type PlayerRole struct {
    Id      int64 // 玩家角色ID
    Pid     int64 // 玩家ID
    RoleId int8  // 角色模板ID
    Level   int16 // 等级
    Exp     int64 // 经验
}

func (this *PlayerRole) c() *C.PlayerRole {
    c := C.NewPlayerRole()
    c.Id = C.int64_t(this.Id)
    c.Pid = C.int64_t(this.Pid)
    c.RoleId = C.int8_t(this.RoleId)
    c.Level = C.int16_t(this.Level)
    c.Exp = C.int64_t(this.Exp)
    return c
}
```

内存数据库的*GC*优化 - 数据存入

谢谢大家