

# Table of Contents

Zinx--Golang轻量级并发服务器框架	1.1
一、引言	1.2
1、写在前面	1.2.1
2、初探Zinx架构	1.2.2
二、初识Zinx框架	1.3
1. Zinx-V0.1-基础Server	1.3.1
2.Zinx-V0.2-简单的连接封装与业务绑定	1.3.2
三、Zinx框架基础路由模块	1.4
3.1 IRequest 消息请求抽象类	1.4.1
3.2 IRouter 路由配置抽象类	1.4.2
3.3 Zinx-V0.3-集成简单路由功能	1.4.3
3.4 Zinx-V0.3代码实现	1.4.4
3.5 使用Zinx-V0.3完成应用程序	1.4.5
四、Zinx的全局配置	1.5
4.1 Zinx-V0.4增添全局配置代码实现	1.5.1
4.2 使用Zinx-V0.4完成应用程序	1.5.2
五、Zinx的消息封装	1.6
5.1 创建消息封装类型	1.6.1
5.2 消息的封包与拆包	1.6.2
5.3 Zinx-V0.5代码实现	1.6.3
5.4 使用Zinx-V0.5完成应用程序	1.6.4
六、Zinx的多路由模式	1.7
6.1 创建消息管理模块	1.7.1
6.2 Zinx-V0.6代码实现	1.7.2
6.3 使用Zinx-V0.6完成应用程序	1.7.3
七、Zinx的读写分离模型	1.8
7.1 Zinx-V0.7代码实现	1.8.1

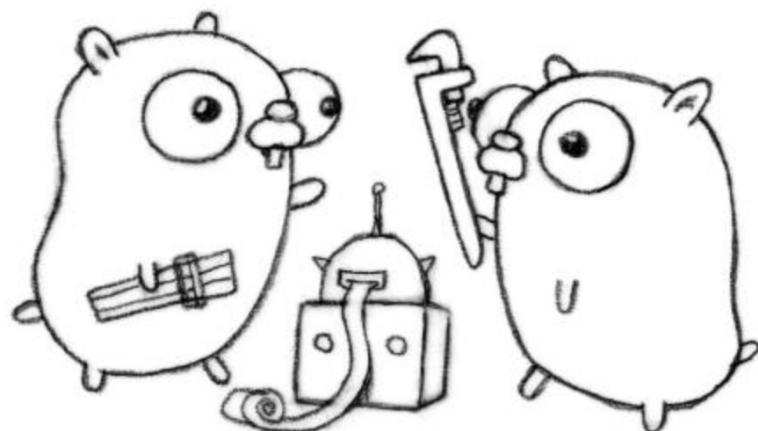
7.2 使用Zinx-V0.7完成应用程序	1.8.2
八、Zinx的消息队列及多任务机制	1.9
8.1 创建消息队列	1.9.1
8.2 创建及启动Worker工作池	1.9.2
8.3 发送消息给消息队列	1.9.3
8.4 Zinx-V0.8代码实现	1.9.4
8.5 使用Zinx-V0.8完成应用程序	1.9.5
九、Zinx的链接管理	1.10
9.1 创建链接管理模块	1.10.1
9.2 链接管理模块集成到Zinx中	1.10.2
9.3 链接的带缓冲的发包方法	1.10.3
9.4 注册链接启动/停止自定义Hook方法功能	1.10.4
9.5 使用Zinx-V0.9完成应用程序	1.10.5
十、Zinx的连接属性设置	1.11
10.1 给链接添加链接配置接口	1.11.1
10.2 链接属性方法实现	1.11.2
10.3 链接属性Zinx-V0.10单元测试	1.11.3
基于Zinx的应用案例	1.12
一、应用案例介绍	1.12.1
二、服务器应用基础协议	1.12.2
三、MMO多人在线游戏AOI算法	1.12.3
3.1 网络法实现AOI算法	1.12.3.1
3.2 实现AOI格子结构	1.12.3.2
3.3 实现AOI管理模块	1.12.3.3
3.4 求出九宫格	1.12.3.4
3.5 AOI格子添加删除操作	1.12.3.5
3.6 AOI模块单元测试	1.12.3.6
四、数据传输协议protocol buffer	1.12.4
4.1 简介	1.12.4.1
4.2 数据交换格式	1.12.4.2

4.3 protobuf环境安装	1.12.4.3
4.4 protobuf语法	1.12.4.4
4.5 编译protobuf	1.12.4.5
4.6 利用protobuf生成的类来编码	1.12.4.6
五、MMO游戏的Proto3协议	1.12.5
六、构建项目与用户上线	1.12.6
6.1 构建项目	1.12.6.1
6.2 用户上线流程	1.12.6.2
七、世界聊天系统实现	1.12.7
7.1 世界管理模块	1.12.7.1
7.2 世界聊天系统实现	1.12.7.2
八、上线位置信息同步	1.12.8
九、移动位置与AOI广播(未跨越格子)	1.12.9
十、玩家下线	1.12.10
十一、移动与AOI广播(跨越格子)	1.12.11

## Zinx--Golang轻量级并发服务器框架

zinX

Golang轻量级并发服务器框架



作者:刘丹冰(Aceld)

**zinX源代码:**

<https://github.com/aceld/zinx>

作者: Aceld(刘丹冰)

简书号: IT无崖子

mail:[danbing.at@gmail.com](mailto:danbing.at@gmail.com)

github:<https://github.com/aceld>

原创书籍gitbook:<http://legacy.gitbook.com/@aceld>

## 一、引言

**zinx**源代码：

<https://github.com/aceld/zinx>

作者： Aceld(刘丹冰)

简书号： IT无崖子

mail:[danbing.at@gmail.com](mailto:danbing.at@gmail.com)

github:<https://github.com/aceld>

原创书籍gitbook:<http://legacy.gitbook.com/@aceld>

## 1、写在前面

我们为什么要做Zinx，Golang目前在服务器的应用框架很多，但是应用在游戏领域或者其他长链接的领域的轻量级企业框架甚少。

设计Zinx的目的是我们可以通过Zinx框架来了解基于Golang编写一个TCP服务器的整体轮廓，让更多的Golang爱好者能深入浅出的去学习和认识这个领域。

Zinx框架的项目制作采用编码和学习教程同步进行，将开发的全部递进和迭代思维带入教程中，而不是一下子给大家一个非常完整的框架去学习，让很多人一头雾水，不知道该如何学起。

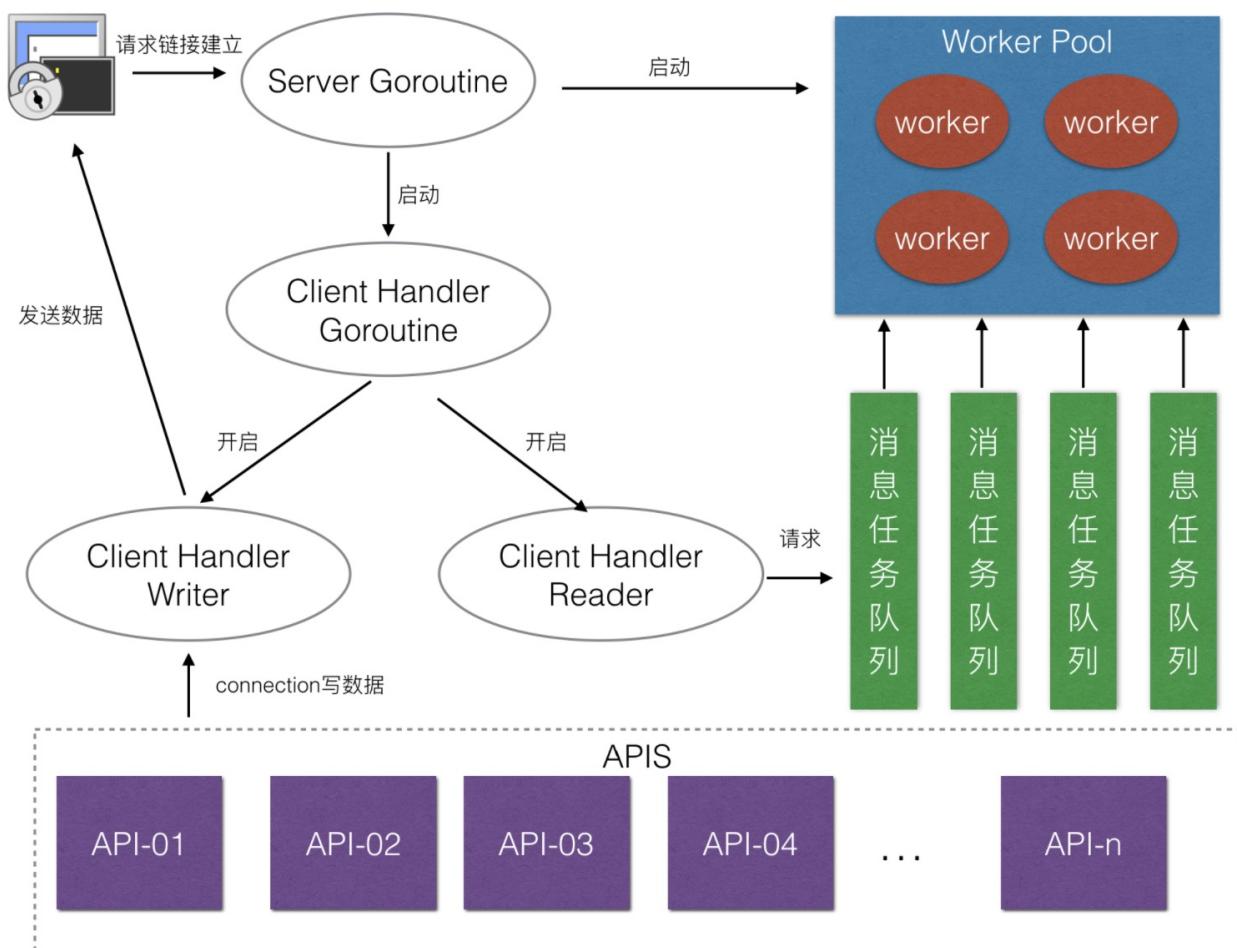
教程会一个版本一个版本迭代，每个版本的添加功能都是微小的，让一个服务框架小白，循序渐进的曲线方式了解服务器框架的领域。

当然，最后希望Zinx会有更多的人加入，给我们提出宝贵的意见，让Zinx成为真正的解决企业的服务器框架！在此感谢您的关注！

zinx源代码：<https://github.com/aceld/zinx>

## 2、初探Zinx架构

以下架构图是初期的设计思路，暂定我们ZinxV1.0版本，该教程也是从Zinx一步一步实现这个V1.0发布版本。



## 二、初识**Zinx**框架

这里先看一下Zinx最简单的Server雏形。

## 1. Zinx-V0.1-基础Server

为了更好的看到Zinx框架，首先Zinx构建Zinx的最基本的两个模块 `ziface` 和 `znet`。

`ziface` 主要是存放一些Zinx框架的全部模块的抽象层接口类，Zinx框架的最基本的是服务类接口 `iserver`，定义在`ziface`模块中。

`znet` 模块是Zinx框架中网络相关功能的实现，所有网络相关模块都会定义在 `znet` 模块中。

### 1.1 Zinx-V0.1 代码实现

#### A) 创建zinx框架

在`$GOPATH/src`下创建 `zinx` 文件夹

#### B) 创建ziface、znet模块

在`zinx/`下 创建`ziface`、`znet`文件夹，使当前的文件路径如下：

```
└── zinx
    ├── ziface
    │   └──
    └── znet
        └──
```

#### C) 在`ziface`下创建服务模块抽象层 `iserver.go`

`zinx/ziface/iserver.go`

## 1. Zinx-V0.1-基础Server

```
package ziface

//定义服务器接口
type IServer interface{
    //启动服务器方法
    Start()
    //停止服务器方法
    Stop()
    //开启业务服务方法
    Serve()
}
```

### D) 在znet下实现服务模块server.go

```
package znet

import (
    "fmt"
    "net"
    "time"
    "zinx/ziface"
)

//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
}

//===== 实现 ziface.IServer 里的全部接口方法 =====

//开启网络服务
```

## 1. Zinx-V0.1-基础Server

```
func (s *Server) Start() {
    fmt.Printf("[START] Server listenner at IP: %s, Port %d, is
starting\n", s.IP, s.Port)

    //开启一个go去做服务端Linster业务
    go func() {
        //1 获取一个TCP的Addr
        addr, err := net.ResolveTCPAddr(s.IPVVersion, fmt.Sprintf(
"%s:%d", s.IP, s.Port))
        if err != nil {
            fmt.Println("resolve tcp addr err: ", err)
            return
        }

        //2 监听服务器地址
        listenner, err:= net.ListenTCP(s.IPVVersion, addr)
        if err != nil {
            fmt.Println("listen", s.IPVVersion, "err", err)
            return
        }

        //已经监听成功
        fmt.Println("start Zinx server ", s.Name, " succ, now l
istenning...")
    }

    //3 启动server网络连接业务
    for {
        //3.1 阻塞等待客户端建立连接请求
        conn, err := listenner.AcceptTCP()
        if err != nil {
            fmt.Println("Accept err ", err)
            continue
        }

        //3.2 TODO Server.Start() 设置服务器最大连接控制,如果超过
        最大连接,那么则关闭此新的连接

        //3.3 TODO Server.Start() 处理该新连接请求的 业务 方法,
        此时应该有 handler 和 conn是绑定的
    }
}
```

## 1. Zinx-V0.1-基础Server

```
//我们这里暂时做一个最大512字节的回显服务
go func () {
    //不断的循环从客户端获取数据
    for {
        buf := make([]byte, 512)
        cnt, err := conn.Read(buf)
        if err != nil {
            fmt.Println("recv buf err ", err)
            continue
        }
        //回显
        if _, err := conn.Write(buf[:cnt]); err !=nil
    {
        fmt.Println("write back buf err ", err)
        continue
    }
    }
}

func (s *Server) Stop() {
    fmt.Println("[STOP] Zinx server , name " , s.Name)

    //TODO Server.Stop() 将其他需要清理的连接信息或者其他信息 也要一并
停止或者清理
}

func (s *Server) Serve() {
    s.Start()

    //TODO Server.Serve() 是否在启动服务的时候 还要处理其他的事情呢 可
以在这里添加

    //阻塞,否则主Go退出, listenner的go将会退出
    for {
        time.Sleep(10*time.Second)
    }
}
```

```
}

/*
 创建一个服务器句柄
*/
func NewServer (name string) ziface.IServer {
    s:= &Server {
        Name :name,
        IPVersion:"tcp4",
        IP:"0.0.0.0",
        Port:7777,
    }

    return s
}
```

好了，以上我们已经完成了Zinx-V0.1的基本雏形了，虽然只是一个基本的回写客户端数据(我们之后会自定义处理客户端业务方法)，那么接下来我们就应该测试我们当前的zinx-V0.1是否可以使用了。

### 1.2 Zinx框架单元测试样例

理论上我们应该可以在导入zinx框架，然后写一个服务端程序，再写一个客户端程序进行测试，但是我们可以通过Go的单元Test功能，进行单元测试

创建zinx/znet/server\_test.go

```
package znet

import (
    "fmt"
    "net"
    "testing"
    "time"
)

/*
 模拟客户端

```

## 1. Zinx-V0.1-基础Server

```
/*
func ClientTest() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn,err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client start err, exit!")
        return
    }

    for {
        _, err := conn.Write([]byte("hello ZINX"))
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }

        buf :=make([]byte, 512)
        cnt, err := conn.Read(buf)
        if err != nil {
            fmt.Println("read buf error ")
            return
        }

        fmt.Printf(" server call back : %s, cnt = %d\n", buf,
        cnt)

        time.Sleep(1*time.Second)
    }

    //Server 模块的测试函数
    func TestServer(t *testing.T) {

        /*
         * 服务端测试
        */
    }
}
```

## 1. Zinx-V0.1-基础Server

```
//1 创建一个server 句柄 s
s := NewServer("[zinx V0.1]")

/*
    客户端测试
*/
go ClientTest()

//2 开启服务
s.Serve()
}
```

在zinx/znet下执行

```
$ go test
```

执行结果，如下：

```
[START] Server listenner at IP: 0.0.0.0, Port 7777, is starting
Client Test ... start
listen tcp4 err listen tcp4 0.0.0.0:7777: bind: address already
in use
server call back : hello ZINX, cnt = 6
server call back : hello ZINX, cnt = 6
server call back : hello ZINX, cnt = 6
server call back : hello ZINX, cnt = 6
```

说明我们的zinx框架已经可以使用了。

### 1.3 使用Zinx-V0.1完成应用程序

当然，如果感觉go test 好麻烦，那么我们可以完全基于zinx写两个应用程序，  
Server.go , Client.go

Server.go

## 1. Zinx-V0.1-基础Server

```
package main

import (
    "zinx/znet"
)

//Server 模块的测试函数
func main() {

    //1 创建一个server 句柄 s
    s := znet.NewServer("[zinx V0.1]")

    //2 开启服务
    s.Serve()
}
```

启动Server.go

```
go run Server.go
```

Client.go

```
package main

import (
    "fmt"
    "net"
    "time"
)

func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
```

## 1. Zinx-V0.1-基础Server

```
        fmt.Println("client start err, exit!")
        return
    }

    for {
        _, err := conn.Write([]byte("hahaha"))
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }

        buf :=make([]byte, 512)
        cnt, err := conn.Read(buf)
        if err != nil {
            fmt.Println("read buf error ")
            return
        }

        fmt.Printf(" server call back : %s, cnt = %d\n", buf, cnt)

        time.Sleep(1*time.Second)
    }
}
```

启动Client.go进行测试

```
go run Client.go
```

## 2.Zinx-V0.2-简单的连接封装与业务绑定

V0.1版本我们已经实现了一个基础的Server框架，现在我们需要对客户端链接和不同的客户端链接所处理的不同业务再做一层接口封装，当然我们先是把架构搭建起来。

现在在 `ziface` 下创建一个属于链接的接口文件 `iconnection.go`，当然他的实现文件我们放在 `znet` 下的 `connection.go` 中。

### 2.1 Zinx-V0.2代码实现

#### A) ziface创建iconnection.go

`zinx/ziface/iconnection.go`

```
package ziface

import "net"

//定义连接接口
type IConnection interface {
    //启动连接，让当前连接开始工作
    Start()
    //停止连接，结束当前连接状态
    Stop()
    //从当前连接获取原始的socket TCPConn
    GetTCPConnection() *net.TCPConn
    //获取当前连接ID
    GetConnID() uint32
    //获取远程客户端地址信息
    RemoteAddr() net.Addr
}

//定义一个统一处理链接业务的接口
type HandFunc func(*net.TCPConn, []byte, int) error
```

该接口的一些基础方法，代码注释已经介绍的很清楚，这里先简单说明一个 HandFunc这个函数类型，这个是所有conn链接在处理业务的函数接口，第一参数是socket原生链接，第二个参数是客户端请求的数据，第三个参数是客户端请求的数据长度。这样，如果我们想要指定一个conn的处理业务，只要定义一个 HandFunc类型的函数，然后和该链接绑定就可以了。

### B) znet 创建iconnection.go

zinx/znet/connection.go

```
package znet

import (
    "fmt"
    "net"
    "zinx/ziface"
)

type Connection struct {
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool

    //该连接的处理方法api
    handleAPI ziface.HandFunc

    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
}

//创建连接的方法
func NewConntion(conn *net.TCPConn, connID uint32, callback_api
ziface.HandFunc) *Connection{
    c := &Connection{
        Conn:      conn,
        ConnID:    connID,
```

```

        isClosed: false,
        handleAPI: callback_api,
        ExitBuffChan: make(chan bool, 1),
    }

    return c
}

/* 处理conn读数据的Goroutine */
func (c *Connection) StartReader() {
    fmt.Println("Reader Goroutine is running")
    defer fmt.Println(c.RemoteAddr().String(), " conn reader exit!")
    defer c.Stop()

    for {
        //读取我们最大的数据到buf中
        buf := make([]byte, 512)
        cnt, err := c.Conn.Read(buf)
        if err != nil {
            fmt.Println("recv buf err ", err)
            c.ExitBuffChan <- true
            continue
        }
        //调用当前链接业务(这里执行的是当前conn的绑定的handle方法)
        if err := c.handleAPI(c.Conn, buf, cnt); err !=nil {
            fmt.Println("connID ", c.ConnID, " handle is error")
            c.ExitBuffChan <- true
            return
        }
    }
}

//启动连接，让当前连接开始工作
func (c *Connection) Start() {

    //开启处理该链接读取到客户端数据之后的请求业务
    go c.StartReader()

    for {

```

```

        select {
        case <- c.ExitBuffChan:
            //得到退出消息，不再阻塞
            return
        }
    }

//停止连接，结束当前连接状态M
func (c *Connection) Stop() {
    //1. 如果当前链接已经关闭
    if c.isClosed == true {
        return
    }
    c.isClosed = true

    //TODO Connection Stop() 如果用户注册了该链接的关闭回调业务，那么在
    //此刻应该显示调用

    // 关闭socket链接
    c.Conn.Close()

    //通知从缓冲队列读数据的业务，该链接已经关闭
    c.ExitBuffChan <- true

    //关闭该链接全部管道
    close(c.ExitBuffChan)
}

//从当前连接获取原始的socket TCPConn
func (c *Connection) GetTCPConnection() *net.TCPConn {
    return c.Conn
}

//获取当前连接ID
func (c *Connection) GetConnID() uint32{
    return c.ConnID
}

//获取远程客户端地址信息

```

```
func (c *Connection) RemoteAddr() net.Addr {
    return c.Conn.RemoteAddr()
}
```

**C) 重新更正一下Server.go中 处理conn的连接业务**

zinx/znet/server.go

```
package znet

import (
    "errors"
    "fmt"
    "net"
    "time"
    "zinx/ziface"
)

//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
}

//===== 定义当前客户端链接的handle api ======
func CallBackToClient(conn *net.TCPConn, data []byte, cnt int) error {
    //回显业务
    fmt.Println("[Conn Handle] CallBackToClient ... ")
    if _, err := conn.Write(data[:cnt]); err !=nil {
        fmt.Println("write back buf err ", err)
        return errors.New("CallBackToClient error")
    }
    return nil
}
```

```

}

//===== 实现 ziface.IServer 里的全部接口方法 =====

//开启网络服务
func (s *Server) Start() {
    fmt.Printf("[START] Server listenner at IP: %s, Port %d, is
starting\n", s.IP, s.Port)

    //开启一个go去做服务端Listen业务
    go func() {
        //1 获取一个TCP的Addr
        addr, err := net.ResolveTCPAddr(s.IPVersion, fmt.Sprintf(
"%s:%d", s.IP, s.Port))
        if err != nil {
            fmt.Println("resolve tcp addr err: ", err)
            return
        }

        //2 监听服务器地址
        listenner, err:= net.ListenTCP(s.IPVersion, addr)
        if err != nil {
            fmt.Println("listen", s.IPVersion, "err", err)
            return
        }

        //已经监听成功
        fmt.Println("start Zinx server ", s.Name, " succ, now 1
istenning...")
    }
}

//TODO server.go 应该有一个自动生成ID的方法
var cid uint32
cid = 0

//3 启动server网络连接业务
for {
    //3.1 阻塞等待客户端建立连接请求
    conn, err := listenner.AcceptTCP()
    if err != nil {
        fmt.Println("Accept err ", err)
}

```

```

        continue
    }

    //3.2 TODO Server.Start() 设置服务器最大连接控制,如果超过
    最大连接,那么则关闭此新的连接

    //3.3 处理该新连接请求的 业务 方法, 此时应该有 handler 和
    conn是绑定的
    dealConn := NewConntion(conn, cid, CallBackToClient)
    cid++

    //3.4 启动当前链接的处理业务
    go dealConn.Start()
}

}()

}

func (s *Server) Stop() {
    fmt.Println("[STOP] Zinx server , name " , s.Name)

    //TODO Server.Stop() 将其他需要清理的连接信息或者其他信息 也要一并
    停止或者清理
}

func (s *Server) Serve() {
    s.Start()

    //TODO Server.Serve() 是否在启动服务的时候 还要处理其他的事情呢 可
    以在这里添加

    //阻塞,否则主Go退出, listenner的go将会退出
    for {
        time.Sleep(10*time.Second)
    }
}

/*
    创建一个服务器句柄
*/
func NewServer (name string) ziface.IServer {

```

```

s := &Server {
    Name :name,
    IPVersion:"tcp4",
    IP:"0.0.0.0",
    Port:7777,
}

return s
}

```

`CallBackToClient` 是我们给当前客户端`conn`对象绑定的`handle`方法，当然目前是`server`端强制绑定的回显业务，我们之后会丰富框架，让这个用户可以让用户自定义指定`handle`。

在 `start()` 方法中，我们主要做了如下的修改：

```

//3.3 处理该新连接请求的 业务 方法， 此时应该有 handler 和
conn是绑定的
dealConn := NewConntion(conn, cid, CallBackToClient)
cid ++

//3.4 启动当前链接的处理业务
go dealConn.Start()

```

好了，现在我们已经将`connection`的连接和`handle`绑定了，下面我们在测试一下 Zinx-V0.2 的框架是否可以使用吧。

## 2.2 使用Zinx-V0.2完成应用程序

实际上，目前Zinx框架的对外接口并未改变，所以V0.1的测试依然有效。

`Server.go`

```

package main

import (
    "zinx/znet"
)

//Server 模块的测试函数
func main() {

    //1 创建一个server 句柄 s
    s := znet.NewServer("[zinx V0.1]")

    //2 开启服务
    s.Serve()
}

```

启动Server.go

```
go run Server.go
```

Client.go

```

package main

import (
    "fmt"
    "net"
    "time"
)

func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {

```

```
        fmt.Println("client start err, exit!")
        return
    }

    for {
        _, err := conn.Write([]byte("hahaha"))
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }

        buf :=make([]byte, 512)
        cnt, err := conn.Read(buf)
        if err != nil {
            fmt.Println("read buf error ")
            return
        }

        fmt.Printf(" server call back : %s, cnt = %d\n", buf, cnt)

        time.Sleep(1*time.Second)
    }
}
```

启动Client.go进行测试

```
go run Client.go
```

现在我们已经简单初始化了Zinx的雏形，但是目前离我们真正的框架还很远，接下来我们来改进zinx框架。

### 三、Zinx框架基础路由模块

现在我们就给用户提供一个自定义的conn处理业务的接口吧，很显然，我们不能把业务处理业务的方法绑死在 `type HandFunc func(*net.TCPConn, []byte, int) error` 这种格式中，我们需要定一些 `interface{}` 来让用户填写任意格式的连接处理业务方法。

那么，很显然func是满足不了我们需求的，我们需要再做几个抽象的接口类。

## 3.1 IRequest 消息请求抽象类

我们现在需要把客户端请求的连接信息和请求的数据，放在一个叫Request的请求类里，这样的好处是我们可以从Request里得到全部客户端的请求信息，也为我们之后拓展框架有一定的作用，一旦客户端有额外的含义的数据信息，都可以放在这这个Request里。可以理解为每次客户端的全部请求数据，Zinx都会把它们一起放到一个Request结构体里。

### A) 创建抽象 IRequest 层

在 ziface 下创建新文件 `irequest.go`。

`zinx/ziface/irequest.go`

```
package ziface

/*
 IRequest 接口：
 实际上是把客户端请求的链接信息 和 请求的数据 包装到了 Request里
*/
type IRequest interface{
    GetConnection() IConnection // 获取请求连接信息
    GetData() []byte           // 获取请求消息的数据
}
```

不难看出，当前的抽象层只提供了两个Getter方法，所以有个成员应该是必须的，一个是客户端连接，一个是客户端传递进来的数据，当然随着Zinx框架的功能丰富，这里面还应该继续添加新的成员。

### B) 实现 Request 类

在znet下创建IRequest抽象接口的一个实例类文件 `request.go`

`zinx/znet/request.go`

### 3.1 IRequest 消息请求抽象类

```
package znet

import "zinx/ziface"

type Request struct {
    conn ziface.IConnection //已经和客户端建立好的 链接
    data []byte //客户端请求的数据
}

//获取请求连接信息
func(r *Request) GetConnection() ziface.IConnection {
    return r.conn
}

//获取请求消息的数据
func(r *Request) GetData() []byte {
    return r.data
}
```

好了现在我们Request类创建好了，稍后我们会用到它。

## 3.2 IRouter 路由配置抽象类

现在我们来给Zinx实现一个非常简单基础的路由功能，目的当然就是为了快速的让Zinx步入到路由的阶段。后续我们会不断的完善路由功能。

### A) 创建抽象的IRouter层

在 ziface 下创建 irouter.go 文件

zinx/ziface/irouter.go

```
package ziface

/*
    路由接口， 这里面路由是 使用框架者给该链接自定的 处理业务方法
    路由里的 IRequest 则包含用该链接的链接信息和该链接的请求数据信息
*/
type IRouter interface{
    PreHandle(request IRequest) //在处理conn业务之前的钩子方法
    Handle(request IRequest)    //处理conn业务的方法
    PostHandle(request IRequest) //处理conn业务之后的钩子方法
}
```

我们知道router实际上的作用就是，服务端应用可以给Zinx框架配置当前链接的处理业务方法，之前的Zinx-V0.2我们的Zinx框架处理链接请求的方法是固定的，现在是可以自定义，并且有3种接口可以重写。

**Handle** : 是处理当前链接的主营业务函数

**PreHandle** : 如果需要在主营业务函数之前有前置业务，可以重写这个方法

**PostHandle** : 如果需要在主营业务函数之后又后置业务，可以重写这个方法

当然每个方法都有一个唯一的形参 **IRequest** 对象，也就是客户端请求过来的连接和请求数据，作为我们业务方法的输入数据。

### B) 实现Router类

在 znet 下创建 router.go 文件

```
package znet

import "zinx/ziface"

//实现router时，先嵌入这个基类，然后根据需要对这个基类的方法进行重写
type BaseRouter struct {}

//这里之所以BaseRouter的方法都为空，  

// 是因为有的Router不希望有PreHandle或PostHandle  

// 所以Router全部继承BaseRouter的好处是，不需要实现PreHandle和PostHandle  

// 也可以实例化
func (br *BaseRouter)PreHandle(req ziface.IRequest){}
func (br *BaseRouter)Handle(req ziface.IRequest){}
func (br *BaseRouter)PostHandle(req ziface.IRequest){}
```

我们当前的Zinx目录结构应该如下：

```
.
├── README.md
├── ziface
│   ├── iconnection.go
│   ├── irequest.go
│   ├── irouter.go
│   └── iserver.go
└── znet
    ├── connection.go
    ├── request.go
    ├── router.go
    ├── server.go
    └── server_test.go
```

### 3.3 Zinx-V0.3-集成简单路由功能

#### A) IServer增添路由添加功能

我们需要给IServer类，增加一个抽象方法 AddRouter，目的也是让Zinx框架使用者，可以自定一个Router处理业务方法。

zinx/ziface/irouter.go

```
package ziface

//定义服务器接口
type IServer interface{
    //启动服务器方法
    Start()
    //停止服务器方法
    Stop()
    //开启业务服务方法
    Serve()
    //路由功能：给当前服务注册一个路由业务方法，供客户端链接处理使用
    AddRouter(router IRouter)
}
```

#### B) Server类增添Router成员

有了抽象的方法，自然Server就要实现，并且还要添加一个Router成员。

zinx/znet/server.go

```
//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
    //当前Server由用户绑定的回调router,也就是Server注册的链接对应的处理
    //业务
    Router ziface.IRouter
}
```

然后 `NewServer()` 方法，初始化`Server`对象的方法也要加一个初始化成员

```
/*
 * 创建一个服务器句柄
 */
func NewServer (name string) ziface.IServer {
    s:= &Server {
        Name :name,
        IPVersion:"tcp4",
        IP:"0.0.0.0",
        Port:7777,
        Router: nil,
    }

    return s
}
```

#### C) Connection类绑定一个Router成员

`zinx/znet/connection.go`

```
type Connection struct {
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool

    //该连接的处理方法router
    Router ziface.IRouter

    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
}
```

D) 在**Connection**调用注册的**Router**处理业务

zinx/znet/connection.go

```

func (c *Connection) StartReader() {
    fmt.Println("Reader Goroutine is running")
    defer fmt.Println(c.RemoteAddr().String(), " conn reader exit!")
    defer c.Stop()

    for {
        //读取我们最大的数据到buf中
        buf := make([]byte, 512)
        _, err := c.Conn.Read(buf)
        if err != nil {
            fmt.Println("recv buf err ", err)
            c.ExitBuffChan <- true
            continue
        }
        //得到当前客户端请求的Request数据
        req := Request{
            conn:c,
            data:buf,
        }
        //从路由Routers 中找到注册绑定Conn的对应Handle
        go func (request ziface IRequest) {
            //执行注册的路由方法
            c.Router.PreHandle(request)
            c.Router.Handle(request)
            c.Router.PostHandle(request)
        }(&req)
    }
}

```

这里我们在conn读取完客户端数据之后，将数据和conn封装到一个Request中，作为Router的输入数据。

然后我们开启一个goroutine去调用给Zinx框架注册好的路由业务。

## 3.4 Zinx-V0.3代码实现

zinx/znet/server.go

```

package znet

import (
    "fmt"
    "net"
    "time"
    "zinx/ziface"
)

//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
    //当前Server由用户绑定的回调router，也就是Server注册的链接对应的处理
    //业务
    Router ziface.IRouter
}

/*
 * 创建一个服务器句柄
 */
func NewServer (name string) ziface.IServer {
    s:= &Server {
        Name :name,
        IPVersion:"tcp4",
        IP:"0.0.0.0",
        Port:7777,
        Router: nil,
    }
}

```

```

    return s
}

//===== 实现 ziface.IServer 里的全部接口方法 =====

//开启网络服务
func (s *Server) Start() {
    fmt.Printf("[START] Server listenner at IP: %s, Port %d, is
starting\n", s.IP, s.Port)

    //开启一个go去做服务端Linster业务
    go func() {
        //1 获取一个TCP的Addr
        addr, err := net.ResolveTCPAddr(s.IPVersion, fmt.Sprintf(
"%s:%d", s.IP, s.Port))
        if err != nil {
            fmt.Println("resolve tcp addr err: ", err)
            return
        }

        //2 监听服务器地址
        listenner, err:= net.ListenTCP(s.IPVersion, addr)
        if err != nil {
            fmt.Println("listen", s.IPVersion, "err", err)
            return
        }

        //已经监听成功
        fmt.Println("start Zinx server ", s.Name, " succ, now 1
istenning...")
    }
}

//TODO server.go 应该有一个自动生成ID的方法
var cid uint32
cid = 0

//3 启动server网络连接业务
for {
    //3.1 阻塞等待客户端建立连接请求
    conn, err := listenner.AcceptTCP()
    if err != nil {

```

```

        fmt.Println("Accept err ", err)
        continue
    }

    //3.2 TODO Server.Start() 设置服务器最大连接控制,如果超过
    最大连接,那么则关闭此新的连接

    //3.3 处理该新连接请求的 业务 方法, 此时应该有 handler 和
    conn是绑定的
    dealConn := NewConntion(conn, cid, s.Router)
    cid ++
}

//3.4 启动当前链接的处理业务
go dealConn.Start()
}

}()

}

func (s *Server) Stop() {
    fmt.Println("[STOP] Zinx server , name " , s.Name)

    //TODO Server.Stop() 将其他需要清理的连接信息或者其他信息 也要一并
    停止或者清理
}

func (s *Server) Serve() {
    s.Start()

    //TODO Server.Serve() 是否在启动服务的时候 还要处理其他的事情呢 可
    以在这里添加

    //阻塞,否则主Go退出, listenner的go将会退出
    for {
        time.Sleep(10*time.Second)
    }
}

//路由功能:给当前服务注册一个路由业务方法,供客户端链接处理使用
func (s *Server)AddRouter(router ziface.IRouter) {
    s.Router = router
}

```

```
    fmt.Println("Add Router succ! ")
}
```

zinx/znet/conneciont.go

```
package znet

import (
    "fmt"
    "net"
    "zinx/ziface"
)

type Connection struct {
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool

    //该连接的处理方法router
    Router ziface.IRouter

    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
}

//创建连接的方法
func NewConntion(conn *net.TCPConn, connID uint32, router ziface.IRouter) *Connection{
    c := &Connection{
        Conn:      conn,
        ConnID:    connID,
        isClosed:  false,
        Router:   router,
        ExitBuffChan: make(chan bool, 1),
    }
}
```

```

    }

    return c
}

func (c *Connection) StartReader() {
    fmt.Println("Reader Goroutine is running")
    defer fmt.Println(c.RemoteAddr().String(), " conn reader exit!")
    defer c.Stop()

    for {
        //读取我们最大的数据到buf中
        buf := make([]byte, 512)
        _, err := c.Conn.Read(buf)
        if err != nil {
            fmt.Println("recv buf err ", err)
            c.ExitBuffChan <- true
            continue
        }
        //得到当前客户端请求的Request数据
        req := Request{
            conn:c,
            data:buf,
        }
        //从路由Routers 中找到注册绑定Conn的对应Handle
        go func (request ziface IRequest) {
            //执行注册的路由方法
            c.Router.PreHandle(request)
            c.Router.Handle(request)
            c.Router.PostHandle(request)
        }(&req)
    }
}

//启动连接，让当前连接开始工作
func (c *Connection) Start() {

    //开启处理该链接读取到客户端数据之后的请求业务
    go c.StartReader()
}

```

```

for {
    select {
        case <- c.ExitBuffChan:
            //得到退出消息，不再阻塞
            return
    }
}

//停止连接，结束当前连接状态M
func (c *Connection) Stop() {
    //1. 如果当前链接已经关闭
    if c.isClosed == true {
        return
    }
    c.isClosed = true

    //TODO Connection Stop() 如果用户注册了该链接的关闭回调业务，那么在此刻应该显示调用

    // 关闭socket链接
    c.Conn.Close()

    //通知从缓冲队列读数据的业务，该链接已经关闭
    c.ExitBuffChan <- true

    //关闭该链接全部管道
    close(c.ExitBuffChan)
}

//从当前连接获取原始的socket TCPConn
func (c *Connection) GetTCPConnection() *net.TCPConn {
    return c.Conn
}

//获取当前连接ID
func (c *Connection) GetConnID() uint32{
    return c.ConnID
}

```

```
//获取远程客户端地址信息
func (c *Connection) RemoteAddr() net.Addr {
    return c.Conn.RemoteAddr()
}
```

## 3.5 使用Zinx-V0.3完成应用程序

接下来我们在基于Zinx写服务器，就可以配置一个简单的路由功能了。

### A) 测试基于Zinx完成的服务端应用

Server.go

```
package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter //一定要先基础BaseRouter
}

//Test PreHandle
func (this *PingRouter) PreHandle(request ziface IRequest) {
    fmt.Println("Call Router PreHandle")
    _, err := request.GetConnection().GetTCPConnection().Write([]byte("before ping ....\n"))
    if err !=nil {
        fmt.Println("call back ping ping error")
    }
}
//Test Handle
func (this *PingRouter) Handle(request ziface IRequest) {
    fmt.Println("Call PingRouter Handle")
    _, err := request.GetConnection().GetTCPConnection().Write([]byte("ping...ping...ping\n"))
    if err !=nil {
        fmt.Println("call back ping ping error")
    }
}
```

```
//Test PostHandle
func (this *PingRouter) PostHandle(request ziface IRequest) {
    fmt.Println("Call Router PostHandle")
    _, err := request.GetConnection().GetTCPConnection().Write([]byte("After ping .....\\n"))
    if err !=nil {
        fmt.Println("call back ping ping ping error")
    }
}

func main(){
    //创建一个server句柄
    s := znet.NewServer("[zinx V0.3]")

    s.AddRouter(&PingRouter{})

    //2 开启服务
    s.Serve()
}
```

我们这里自定义了一个类似Ping操作的路由，就是当客户端发送数据，我们的处理业务就是返回给客户端"ping...ping..ping.."，为了测试，当前路由也同时实现了PreHandle和PostHandle两个方法。实际上Zinx会利用模板的设计模式，依次在框架中调用 PreHandle 、 Handle 、 PostHandle 三个方法。

### B) 启动Server.go

```
go run Server.go
```

### C) 客户端应用测试程序

和之前的Client.go一样 没有改变

```
package main

import (
    "fmt"
```

```

    "net"
    "time"
)

/*
    模拟客户端
*/
func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client start err, exit!")
        return
    }

    for {
        _, err := conn.Write([]byte("Zinx V0.3"))
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }

        buf :=make([]byte, 512)
        cnt, err := conn.Read(buf)
        if err != nil {
            fmt.Println("read buf error ")
            return
        }

        fmt.Printf(" server call back : %s, cnt = %d\n", buf, cnt)

        time.Sleep(1*time.Second)
    }
}

```

#### D) 启动**Client.go**

```
go run Client.go
```

运行结果如下：

服务端：

```
$ go run Server.go
Add Router succ!
[START] Server listenner at IP: 0.0.0.0, Port 7777, is starting
start Zinx server [zinx V0.3] succ, now listenning...
Reader Goroutine is running
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
Call Router PreHandle
Call PingRouter Handle
Call Router PostHandle
...
...
```

客户端：

```
$ go run Client.go
Client Test ... start
    server call back : before ping ....
, cnt = 17
    server call back : ping...ping...ping
After ping ....
, cnt = 36
    server call back : before ping ....
ping...ping...ping
After ping ....
, cnt = 53
    server call back : before ping ....
ping...ping...ping
After ping ....
, cnt = 53
    server call back : before ping ....
ping...ping...ping
After ping ....
, cnt = 53
...
...
```

现在Zinx框架已经有路由功能了，虽然说目前只能配置一个，不过不要着急，很快我们会增加配置多路由的能力。

## 四、Zinx的全局配置

随着架构逐步的变大，参数就会越来越多，为了省去我们后续大频率修改参数的麻烦，接下来Zinx需要做一个加载配置的模块，和一个全局获取Zinx参数的对象。

## 4.1 Zinx-V0.4增添全局配置代码实现

我们先做一个简单的加载配置模块，要加载的配置文件的文本格式，就选择比较通用的 `json` 格式，配置信息暂时如下：

zinx.json

```
{  
    "Name": "demo server",  
    "Host": "127.0.0.1",  
    "TcpPort": 7777,  
    "MaxConn": 3  
}
```

现在我们需要建立一个全局配置信息的对象

### A) 创建全局参数文件

创建 `zinx/utils` 文件夹，在下面创建 `globalobj.go` 文件，暂时编写如下。

zinx/utils/globalobj.go

```

package utils

import (
    "encoding/json"
    "io/ioutil"
    "zinx/ziface"
)

/*
存储一切有关Zinx框架的全局参数，供其他模块使用
一些参数也可以通过 用户根据 zinx.json来配置
*/
type GlobalObj struct {
    TcpServer ziface.IServer //当前Zinx的全局Server对象
    Host      string         //当前服务器主机IP
    TcpPort   int            //当前服务器主机监听端口号
    Name      string         //当前服务器名称
    Version   string         //当前Zinx版本号

    MaxPacketSize uint32      //都需数据包的最大值
    MaxConn      int           //当前服务器主机允许的最大链接个数
}

/*
定义一个全局的对象
*/
var GlobalObject *GlobalObj

```

我们在全局定义了一个 `GlobalObject` 对象，目的就是让其他模块都能访问到里面的参数。

### B) 提供init初始化方法

然后我们提供一个 `init()` 方法，目的是初始化 `GlobalObject` 对象，和加载服务端应用配置文件 `conf/zinx.json`

`zinx/utils/globalobj.go`

```

//读取用户的配置文件
func (g *GlobalObj) Reload() {
    data, err := ioutil.ReadFile("conf/zinx.json")
    if err != nil {
        panic(err)
    }
    //将json数据解析到struct中
    //fmt.Printf("json :%s\n", data)
    err = json.Unmarshal(data, &GlobalObject)
    if err != nil {
        panic(err)
    }
}

/*
提供init方法，默认加载
*/
func init() {
    //初始化GlobalObject变量，设置一些默认值
    GlobalObject = &GlobalObj{
        Name:      "ZinxServerApp",
        Version:   "V0.4",
        TcpPort:   7777,
        Host:      "0.0.0.0",
        MaxConn:   12000,
        MaxPacketSize: 4096,
    }

    //从配置文件中加载一些用户配置的参数
    GlobalObject.Reload()
}

```

### C) 硬参数替换与**Server**初始化参数配置

zinx/znet/server.go

```

/*
    创建一个服务器句柄
*/
func NewServer () ziface.IServer {
    //先初始化全局配置文件
    utils.GlobalObject.Reload()

    s:= &Server {
        Name :utils.GlobalObject.Name,//从全局参数获取
        IPVersion:"tcp4",
        IP:utils.GlobalObject.Host,//从全局参数获取
        Port:utils.GlobalObject.TcpPort,//从全局参数获取
        Router: nil,
    }
    return s
}

```

我们未来方便验证我们的参数已经成功被值，在 `Server.Start()` 方法中加入几行调试信息

`zinx/znet/server.go`

```

//开启网络服务
func (s *Server) Start() {
    fmt.Printf("[START] Server name: %s, listenner at IP: %s, Port %d is starting\n", s.Name, s.IP, s.Port)
    fmt.Printf("[Zinx] Version: %s, MaxConn: %d, MaxPacketSize: %d\n",
        utils.GlobalObject.Version,
        utils.GlobalObject.MaxConn,
        utils.GlobalObject.MaxPacketSize)

    //...
    //...
}

```

当然还有一些其他的之前写死的数值，均可以在配置文件配置，用全局参数替换，这里不一一列举。

### 当前zinx框架目录结构

```
├── README.md
├── utils
│   └── globalobj.go
└── ziface
    ├── iconnection.go
    ├── irequest.go
    ├── irouter.go
    └── iserver.go
└── znet
    ├── connection.go
    ├── request.go
    ├── router.go
    ├── server.go
    └── server_test.go
```

## 4.2 使用Zinx-V0.4完成应用程序

我们这回再基于Zinx完成服务器就必须要提前先写好一个 `conf/zinx.json` 配置文件了。

```
|── Client.go
├── conf
│   └── zinx.json
└── Server.go
```

Server.go

```

package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Test Handle
func (this *PingRouter) Handle(request ziface IRequest) {
    fmt.Println("Call PingRouter Handle")
    _, err := request.GetConnection().GetTCPConnection().Write([]byte("ping...ping...ping\n"))
    if err != nil {
        fmt.Println("call back ping ping error")
    }
}

func main() {
    //创建一个server句柄
    s := znet.NewServer()

    //配置路由
    s.AddRouter(&PingRouter{})

    //开启服务
    s.Serve()
}

```

```
$go run Server.go
```

结果:

```
$ go run Server.go
Add Router succ!
[START] Server name: demo server, listenner at IP: 127.0.0.1, Port 7777 is starting
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
start Zinx server    demo server  succ, now listenning...
```

现在配置已经加载成功了。

## 五、Zinx的消息封装

接下来我们再对Zinx做一个简单的升级，现在我们把服务器的全部数据都放在一个Request里，当前的Request结构如下：

```
type Request struct {
    conn ziface.IConnection //已经和客户端建立好的链接
    data []byte              //客户端请求的数据
}
```

很明显，现在是用一个 `[]byte` 来接受全部数据，又没有长度，又没有消息类型，这不科学。怎么办呢？我们现在就要自定义一种消息类型，把全部的消息都放在这种消息类型里。

## 5.1 创建消息封装类型

在 `zinx/ziface/` 下创建 `imessage.go` 文件

`zinx/ziface/imessage.go`

```
package ziface

/*
将请求的一个消息封装到message中，定义抽象层接口
*/
type IMessage interface {
    GetDataLen() uint32      //获取消息数据段长度
    GetMsgId() uint32        //获取消息ID
    GetData() []byte         //获取消息内容

    SetMsgId(uint32)         //设计消息ID
    SetData([]byte)          //设计消息内容
    SetDataLen(uint32)        //设置消息数据段长度
}
```

同时创建实例message类，在 `zinx/znet/` 下，创建 `message.go` 文件

`zinx/znet/message.go`

```
package znet

type Message struct {
    Id      uint32 //消息的ID
    DataLen uint32 //消息的长度
    Data    []byte  //消息的内容
}

//创建一个Message消息包
func NewMsgPackage(id uint32, data []byte) *Message {
    return &Message{
        Id:      id,
        DataLen: uint32(len(data)),
        Data:    data,
```

## 5.1 创建消息封装类型

```
    }

}

//获取消息数据段长度
func (msg *Message) GetDataLen() uint32 {
    return msg.DataLen
}

//获取消息ID
func (msg *Message) GetMsgId() uint32 {
    return msg.Id
}

//获取消息内容
func (msg *Message) GetData() []byte {
    return msg.Data
}

//设置消息数据段长度
func (msg *Message) SetDataLen(len uint32) {
    msg.DataLen = len
}

//设计消息ID
func (msg *Message) SetMsgId(msgId uint32) {
    msg.Id = msgId
}

//设计消息内容
func (msg *Message) SetData(data []byte) {
    msg.Data = data
}
```

整理一个基本的**message**包，会包含**消息ID**，**数据**，**数据长度**三个成员，提供基本的**setter**和**getter**方法，目的是为了以后做封装优化的作用。同时也提供了一个创建一个**message**包的初始化方法 `NewMegPackage`。



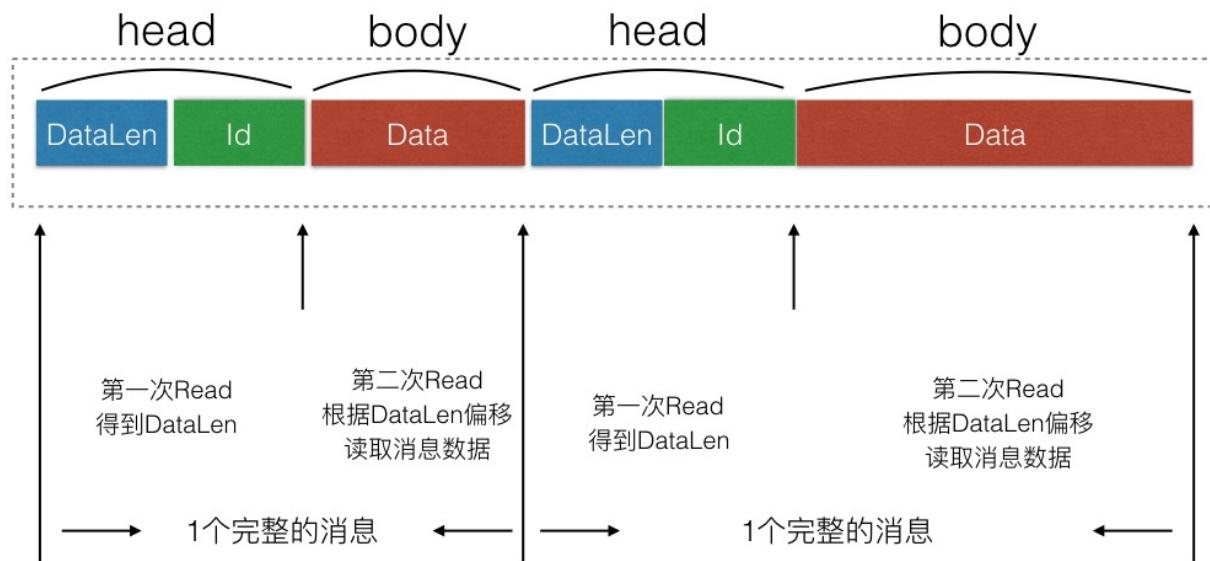
## 5.2 消息的封包与拆包

我们这里就是采用经典的TLV(Type-Len-Value)封包格式来解决TCP粘包问题吧。



解决TCP粘包问题

IT无崖子



由于Zinx也是TCP流的形式传播数据，难免会出现消息1和消息2一同发送，那么zinx就需要有能力区分两个消息的边界，所以Zinx此时应该提供一个统一的拆包和封包的方法。在发包之前打包成如上图这种格式的有head和body的两部分的包，在收到数据的时候分两次进行读取，先读取固定长度的head部分，得到后续Data的长度，再根据DataLen读取之后的body。这样就能够解决粘包的问题了。

### A) 创建拆包封包抽象类

在 `zinx/ziface` 下，创建 `idatapack.go` 文件

```
| zinx/ziface/idatapack.go
```

```
package ziface

/*
 封包数据和拆包数据
 直接面向TCP连接中的数据流，为传输数据添加头部信息，用于处理TCP粘包问题。
 */

type IDataPack interface{
    GetHeadLen() uint32           //获取包头长度方法
    Pack(msg IMessage)([]byte, error) //封包方法
    Unpack([]byte)(IMessage, error) //拆包方法
}
```

### B) 实现拆包封包类

在 `zinx/znet/` 下，创建 `datapack.go` 文件。

`zinx/znet/datapack.go`

```
package znet

import (
    "bytes"
    "encoding/binary"
    "errors"
    "zinx/utils"
    "zinx/ziface"
)

//封包拆包类实例，暂时不需要成员
type DataPack struct {}

//封包拆包实例初始化方法
func NewDataPack() *DataPack {
    return &DataPack{}
}

//获取包头长度方法
func(dp *DataPack) GetHeadLen() uint32 {
    //Id uint32(4字节) +  DataLen uint32(4字节)
```

```

    return 8
}

//封包方法(压缩数据)
func(dp *DataPack) Pack(msg ziface.IMessage)([]byte, error) {
    //创建一个存放bytes字节的缓冲
    dataBuff := bytes.NewBuffer([]byte{})

    //写dataLen
    if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetdataLen()); err != nil {
        return nil, err
    }

    //写msgID
    if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetMsgId()); err != nil {
        return nil, err
    }

    //写data数据
    if err := binary.Write(dataBuff, binary.LittleEndian, msg.GetData()); err != nil {
        return nil, err
    }

    return dataBuff.Bytes(), nil
}

//拆包方法(解压数据)
func(dp *DataPack) Unpack(binaryData []byte)(ziface.IMessage, error) {
    //创建一个从输入二进制数据的ioReader
    dataBuff := bytes.NewReader(binaryData)

    //只解压head的信息，得到dataLen和msgID
    msg := &Message{}

    //读dataLen
    if err := binary.Read(dataBuff, binary.LittleEndian, &msg.DataLen); err != nil {

```

## 5.2 消息的封包与拆包

```
        return nil, err
    }

    //读msgID
    if err := binary.Read(dataBuff, binary.LittleEndian, &msg.Id);
    err != nil {
        return nil, err
    }

    //判断dataLen的长度是否超出我们允许的最大包长度
    if (utils.GlobalObject.MaxPacketSize > 0 && msg.DataLen > utils.GlobalObject.MaxPacketSize) {
        return nil, errors.New("Too large msg data received")
    }

    //这里只需要把head的数据拆包出来就可以了，然后再通过head的长度，再从conn读取一次数据
    return msg, nil
}
```

需要注意的是整理的 `Unpack` 方法，因为我们从上图可以知道，我们进行拆包的时候是分两次过程的，第二次是依赖第一次的 `dataLen` 结果，所以 `Unpack` 只能解压出包头 `head` 的内容，得到 `msgId` 和 `dataLen`。之后调用者再根据 `dataLen` 继续从 `io` 流中读取 `body` 中的数据。

### C) 测试拆包封包功能

为了容易理解，我们先不用集成 `zinx` 框架来测试，而是单独写一个 `Server` 和 `Client` 来测试一下封包拆包的功能

Server.go

```
package main

import (
    "fmt"
    "io"
    "net"
    "zinx/znet"
```

```

)

//只是负责测试datapack拆包，封包功能
func main() {
    //创建socket TCP Server
    listener, err := net.Listen("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("server listen err:", err)
        return
    }

    //创建服务器gotoutine，负责从客户端goroutine读取粘包的数据，然后进行
    //解析

    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("server accept err:", err)
        }

        //处理客户端请求
        go func(conn net.Conn) {
            //创建封包拆包对象dp
            dp := znet.NewDataPack()
            for {
                //1 先读出流中的head部分
                headData := make([]byte, dp.GetHeadLen())
                _, err := io.ReadFull(conn, headData) //ReadFull
                //会把msg填充满为止
                if err != nil {
                    fmt.Println("read head error")
                    break
                }
                //将headData字节流 拆包到msg中
                msgHead, err := dp.Unpack(headData)
                if err != nil {
                    fmt.Println("server unpack err:", err)
                    return
                }
            }
        }
    }
}

```

## 5.2 消息的封包与拆包

```
        if msgHead.GetDataLen() > 0 {
            //msg 是有data数据的，需要再次读取data数据
            msg := msgHead.(*znet.Message)
            msg.Data = make([]byte, msg.GetDataLen())

            //根据dataLen从io中读取字节流
            _, err := io.ReadFull(conn, msg.Data)
            if err != nil {
                fmt.Println("server unpack data err:", e
rr)
                return
            }

            fmt.Println("==> Recv Msg: ID=", msg.Id, ",",
len=", msg.DataLen, ", data=", string(msg.Data))
        }
    }
}(conn)
}

}
```

### Client.go

```
package main

import (
    "fmt"
    "net"
    "zinx/znet"
)

func main() {
    //客户端goroutine，负责模拟粘包的数据，然后进行发送
    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client dial err:", err)
        return
    }
```

```

//创建一个封包对象 dp
dp := znet.NewDataPack()

//封装一个msg1包
msg1 := &znet.Message{
    Id:      0,
    DataLen: 5,
    Data:    []byte{'h', 'e', 'l', 'l', 'o'},
}

sendData1, err := dp.Pack(msg1)
if err != nil {
    fmt.Println("client pack msg1 err:", err)
    return
}

msg2 := &znet.Message{
    Id:      1,
    DataLen: 7,
    Data:    []byte{'w', 'o', 'r', 'l', 'd', '!', '!'},
}
sendData2, err := dp.Pack(msg2)
if err != nil {
    fmt.Println("client temp msg2 err:", err)
    return
}

//将sendData1，和 sendData2 拼接一起，组成粘包
sendData1 = append(sendData1, sendData2...)

//向服务器端写数据
conn.Write(sendData1)

//客户端阻塞
select {}

}

```

运行Server.go

## 5.2 消息的封包与拆包

```
go run Server.go
```

运行Client.go

```
go run Client.go
```

我们从服务端看到运行结果

```
$go run Server.go
==> Recv Msg: ID= 0 , len= 5 , data= hello
==> Recv Msg: ID= 1 , len= 7 , data= world!!
```

我们成功的得到了客户端发送的两个包，并且成功的解析出来。

## 5.3 Zinx-V0.5代码实现

现在我们需要把封包和拆包的功能集成到Zinx中，并且测试Zinx该功能是否生效。

### A) Request字段修改

首先我们要将我们之前的Request中的`[]byte`类型的`data`字段改成`Message`类型。

`zinx/znet/request.go`

```
package znet

import "zinx/ziface"

type Request struct {
    conn ziface.IConnection //已经和客户端建立好的 链接
    msg ziface.IMessage     //客户端请求的数据
}

//获取请求连接信息
func(r *Request) GetConnection() ziface.IConnection {
    return r.conn
}

//获取请求消息的数据
func(r *Request) GetData() []byte {
    return r.msg.GetData()
}

//获取请求的消息的ID
func (r *Request) GetMsgID() uint32 {
    return r.msg.GetMsgId()
}
```

### B) 集成拆包过程

接下来我们需要在`Connection`的`StartReader()`方法中，修改之前的读取客户端的这段代码：

```

func (c *Connection) StartReader() {
    // ...

    for {
        //读取我们最大的数据到buf中
        buf := make([]byte, utils.GlobalObject.MaxPacketSize)
        _, err := c.Conn.Read(buf)
        if err != nil {
            fmt.Println("recv buf err ", err)
            c.ExitBuffChan <- true
            continue
        }

        // ...
    }
}

```

改成如下：

| zinx/znet/connection.go

StartReader()方法

```

func (c *Connection) StartReader() {
    fmt.Println("Reader Goroutine is running")
    defer fmt.Println(c.RemoteAddr().String(), " conn reader exit!")
    defer c.Stop()

    for {
        // 创建拆包解包的对象
        dp := NewDataPack()

        //读取客户端的Msg head
        headData := make([]byte, dp.GetHeadLen())
        if _, err := io.ReadFull(c.GetTCPConnection(), headData)
; err != nil {

```

```

        fmt.Println("read msg head error ", err)
        c.ExitBuffChan <- true
        continue
    }

    //拆包，得到msgid 和 datalen 放在msg中
    msg , err := dp.Unpack(headData)
    if err != nil {
        fmt.Println("unpack error ", err)
        c.ExitBuffChan <- true
        continue
    }

    //根据 dataLen 读取 data，放在msg.Data中
    var data []byte
    if msg.GetDataLen() > 0 {
        data = make([]byte, msg.GetDataLen())
        if _, err := io.ReadFull(c.GetTCPConnection(), data)
; err != nil {
            fmt.Println("read msg data error ", err)
            c.ExitBuffChan <- true
            continue
        }
    }
    msg.SetData(data)

    //得到当前客户端请求的Request数据
    req := Request{
        conn:c,
        msg:msg, //将之前的buf 改成 msg
    }
    //从路由Routers 中找到注册绑定Conn的对应Handle
    go func (request ziface IRequest) {
        //执行注册的路由方法
        c.Router.PreHandle(request)
        c.Router.Handle(request)
        c.Router.PostHandle(request)
    }(&req)
}

}

```

### C) 提供封包方法

现在我们已经将拆包的功能集成到Zinx中了，但是使用Zinx的时候，如果我们希望给用户返回一个TLV格式的数据，总不能每次都经过这么繁琐的过程，所以我们应该给Zinx提供一个封包的接口，供Zinx发包使用。

zinx/ziface/iconnection.go

新增 SendMsg() 方法

```
package ziface

import "net"

//定义连接接口
type IConnection interface {
    //启动连接，让当前连接开始工作
    Start()
    //停止连接，结束当前连接状态
    Stop()
    //从当前连接获取原始的socket TCPConn
    GetTCPConnection() *net.TCPConn
    //获取当前连接ID
    GetConnID() uint32
    //获取远程客户端地址信息
    RemoteAddr() net.Addr
    //直接将Message数据发送数据给远程的TCP客户端
    SendMsg(msgId uint32, data []byte) error
}
```

zinx/znet/connection.go

SendMsg() 方法实现:

```
//直接将Message数据发送数据给远程的TCP客户端
func (c *Connection) SendMsg(msgId uint32, data []byte) error {
    if c.isClosed == true {
        return errors.New("Connection closed when send msg")
    }
    //将data封包，并且发送
    dp := NewDataPack()
    msg, err := dp.Pack(NewMsgPackage(msgId, data))
    if err != nil {
        fmt.Println("Pack error msg id = ", msgId)
        return errors.New("Pack error msg ")
    }

    //写回客户端
    if _, err := c.Conn.Write(msg); err != nil {
        fmt.Println("Write msg id ", msgId, " error ")
        c.ExitBuffChan <- true
        return errors.New("conn Write error")
    }

    return nil
}
```

## 5.4 使用Zinx-V0.5完成应用程序

现在我们可以基于Zinx框架完成发送msg功能的测试用例了。

| Server.go

```

package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Test Handle
func (this *PingRouter) Handle(request ziface.IRequest) {
    fmt.Println("Call PingRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
    ", data=", string(request.GetData()))

    //回写数据
    err := request.GetConnection().SendMsg(1, []byte("ping...pin
g...ping"))
    if err != nil {
        fmt.Println(err)
    }
}

func main() {
    //创建一个server句柄
    s := znet.NewServer()

    //配置路由
    s.AddRouter(&PingRouter{})

    //开启服务
    s.Serve()
}

```

当前Server端是先把客户端发送来Msg解析，然后返回一个MsgId为1的消息，消息内容是"ping...ping...ping"

### Client.go

```
package main

import (
    "fmt"
    "io"
    "net"
    "time"
    "zinx/znet"
)

/*
    模拟客户端
*/
func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client start err, exit!")
        return
    }

    for {
        //发封包message消息
        dp := znet.NewDataPack()
        msg, _ := dp.Pack(znet.NewMsgPackage(0, []byte("Zinx V0.5
Client Test Message")))
        _, err := conn.Write(msg)
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }
    }
}
```

```

//先读出流中的head部分
headData := make([]byte, dp.GetHeadLen())
_, err = io.ReadFull(conn, headData) //ReadFull 会把msg填
充满为止
if err != nil {
    fmt.Println("read head error")
    break
}
//将headData字节流 拆包到msg中
msgHead, err := dp.Unpack(headData)
if err != nil {
    fmt.Println("server unpack err:", err)
    return
}

if msgHead.GetDataLen() > 0 {
    //msg 是有data数据的，需要再次读取data数据
    msg := msgHead.(*znet.Message)
    msg.Data = make([]byte, msg.GetDataLen())

    //根据dataLen从io中读取字节流
    _, err := io.ReadFull(conn, msg.Data)
    if err != nil {
        fmt.Println("server unpack data err:", err)
        return
    }

    fmt.Println("==> Recv Msg: ID=", msg.Id, ", len=", msg.DataLen, ", data=", string(msg.Data))
}
time.Sleep(1*time.Second)
}
}

```

这里Client客户端，模拟了一个MsgId为0的"Zinx V0.5 Client Test Message"消息，然后把服务端返回的数据打印出来。

我们分别在两个终端运行

```
$go run Server.go
```

```
$go run Client.go
```

服务端结果：

```
$ go run Server.go
Add Router succ!
[START] Server name: zinx v-0.5 demoApp,listener at IP: 127.0.0
.1, Port 7777 is starting
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
start Zinx server zinx v-0.5 demoApp succ, now listenning...
Reader Goroutine is running
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.5 Client Test Message
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.5 Client Test Message
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.5 Client Test Message
...
...
```

客户端结果：

```
$ go run Client.go
Client Test ... start
==> Recv Msg: ID= 1 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 1 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 1 , len= 18 , data= ping...ping...ping
...
...
```

好了，我们的Zinx已经成功的集成消息的封装功能了，这样我们就有Zinx的通信的基本协议标准了。



## 六、Zinx的多路由模式

我们之前在已经给Zinx配置了路由模式，但是很惨，之前的Zinx好像只能绑定一个路由的处理业务方法。显然这是无法满足基本的服务器需求的，那么现在我们要在之前的基础上，给Zinx添加多路由的方式。

既然是多路由的模式，我们这里就需要给MsgId和对应的处理逻辑进行捆绑。所以我们需要一个Map。

```
Apis map[uint32] ziface.IRouter
```

这里起名字是 `Apis`，其中key就是msgId，value就是对应的Router，里面应是使用者重写的Handle等方法。

那么这个`Apis`应该放在哪呢。

我们再定义一个消息管理模块来进行维护这个 `Apis`。

## 6.1 创建消息管理模块

### A) 创建消息管理模块抽象类

在 `zinx/ziface` 下创建 `imsghandler.go` 文件。

```
package ziface
/*
消息管理抽象层
*/
type IMsgHandle interface{
    DoMsgHandler(request IRequest)           //马上以非阻塞方式处
理消息
    AddRouter(msgId uint32, router IRouter)   //为消息添加具体的处
理逻辑
}
```

这里面有两个方法，`AddRouter()` 就是添加一个`msgId`和一个路由关系到`Apis`中，那么 `DoMsgHandler()` 则是调用`Router`中具体 `Handle()` 等方法的接口。

### B) 实现消息管理模块

在 `zinx/znet` 下创建 `msghandler.go` 文件。

```
package znet

import (
    "fmt"
    "strconv"
    "zinx/ziface"
)

type MsgHandle struct{
    Apis map[uint32] ziface.IRouter //存放每个MsgId 所对应的处理方法
的map属性
}

func NewMsgHandle() *MsgHandle {
```

## 6.1 创建消息管理模块

```
return &MsgHandle {
    Apis:make(map[uint32]ziface.IRouter),
}

//马上以非阻塞方式处理消息
func (mh *MsgHandle) DoMsgHandler(request ziface IRequest) {
    handler, ok := mh.Apis[request.GetMsgID()]
    if !ok {
        fmt.Println("api msgId = ", request.GetMsgID(), " is not
FOUND!")
        return
    }

    //执行对应处理方法
    handler.PreHandle(request)
    handler.Handle(request)
    handler.PostHandle(request)
}

//为消息添加具体的处理逻辑
func (mh *MsgHandle) AddRouter(msgId uint32, router ziface.IRout
er) {
    //1 判断当前msg绑定的API处理方法是否已经存在
    if _, ok := mh.Apis[msgId]; ok {
        panic("repeated api , msgId = " + strconv.Itoa(int(msgId
)))
    }
    //2 添加msg与api的绑定关系
    mh.Apis[msgId] = router
    fmt.Println("Add api msgId = ", msgId)
}
```

## 6.2 Zinx-V0.6代码实现

首先 `iserver` 的 `AddRouter()` 的接口要稍微改一下，增添`MsgId`参数

`zinx/ziface/iserver.go`

```
package ziface

//定义服务器接口
type IServer interface{
    //启动服务器方法
    Start()
    //停止服务器方法
    Stop()
    //开启业务服务方法
    Serve()
    //路由功能：给当前服务注册一个路由业务方法，供客户端链接处理使用
    AddRouter(msgId uint32, router IRouter)
}
```

其次，`Server` 类中之前有一个 `Router` 成员，代表唯一的处理方法，现在应该替换成 `MsgHandler` 成员

`zinx/znet/server.go`

```
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
    //当前Server的消息管理模块，用来绑定MsgId和对应的处理方法
    msgHandler ziface.IMsgHandle
}
```

初始化Server自然也要更正，增加msgHandler初始化

```
/*
 创建一个服务器句柄
 */
func NewServer () ziface.IServer {
    utils.GlobalObject.Reload()

    s:= &Server {
        Name :utils.GlobalObject.Name,
        IPVersion:"tcp4",
        IP:utils.GlobalObject.Host,
        Port:utils.GlobalObject.TcpPort,
        msgHandler: NewMsgHandle(), //msgHandler 初始化
    }
    return s
}
```

然后当Server在处理conn请求业务的时候，创建conn的时候也需要把msgHandler作为参数传递给Connection对象

```
//...
dealConn := NewConntion(conn, cid, s.msgHandler)
//...
```

那么接下来就是Connection对象了。固然在Connection对象中应该有MsgHandler的成员，来查找消息对应的回调路由方法

zinx/znet/connection.go

```

type Connection struct {
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool
    //消息管理MsgId和对应处理方法的消息管理模块
    MsgHandler ziface.IMsgHandle
    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
}

//创建连接的方法
func NewConntion(conn *net.TCPConn, connID uint32, msgHandler zi
face.IMsgHandle) *Connection{
    c := &Connection{
        Conn:      conn,
        ConnID:    connID,
        isClosed:  false,
        MsgHandler: msgHandler,
        ExitBuffChan: make(chan bool, 1),
    }

    return c
}

```

最后，在conn已经拆包之后，需要调用路由业务的时候，我们只需要让conn调用MsgHandler中的 DoMsgHander() 方法就好了

zinx/znet/connection.go

```

func (c *Connection) StartReader() {
    fmt.Println("[Reader Goroutine is running]")
    defer fmt.Println(c.RemoteAddr().String(), "[conn Reader exi
t!]")
    defer c.Stop()

    for {

```

```

    // 创建拆包解包的对象
    dp := NewDataPack()

    //读取客户端的Msg head
    headData := make([]byte, dp.GetHeadLen())
    if _, err := io.ReadFull(c.GetTCPConnection(), headData)
; err != nil {
        fmt.Println("read msg head error ", err)
        break
    }

    //拆包，得到msgid 和 datalen 放在msg中
    msg, err := dp.Unpack(headData)
    if err != nil {
        fmt.Println("unpack error ", err)
        break
    }

    //根据 dataLen 读取 data，放在msg.Data中
    var data []byte
    if msg.GetDataLen() > 0 {
        data = make([]byte, msg.GetDataLen())
        if _, err := io.ReadFull(c.GetTCPConnection(), data)
; err != nil {
            fmt.Println("read msg data error ", err)
            continue
        }
    }
    msg.SetData(data)

    //得到当前客户端请求的Request数据
    req := Request{
        conn:c,
        msg:msg,
    }
    //从绑定好的消息和对应的处理方法中执行对应的Handle方法
    go c.MsgHandler.DoMsgHandler(&req)
}

}

```

好了，大功告成，我们来测试一下Zinx的多路由设置功能吧。

## 6.3 使用Zinx-V0.6完成应用程序

Server.go

```
package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Ping Handle
func (this *PingRouter) Handle(request ziface IRequest) {
    fmt.Println("Call PingRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
    ", data=", string(request.GetData()))

    err := request.GetConnection().SendMsg(0, []byte("ping...pin
g...ping"))
    if err != nil {
        fmt.Println(err)
    }
}

//HelloZinxRouter Handle
type HelloZinxRouter struct {
    znet.BaseRouter
}

func (this *HelloZinxRouter) Handle(request ziface IRequest) {
    fmt.Println("Call HelloZinxRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
```

## 6.3 使用Zinx-V0.6完成应用程序

```
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
", data=", string(request.GetData()))

    err := request.GetConnection().SendMsg(1, []byte("Hello Zinx
Router V0.6"))
    if err != nil {
        fmt.Println(err)
    }
}

func main() {
    //创建一个server句柄
    s := znet.NewServer()

    //配置路由
    s.AddRouter(0, &PingRouter{})
    s.AddRouter(1, &HelloZinxRouter{})

    //开启服务
    s.Serve()
}
```

Server端设置了2个路由，一个是MsgId为0的消息会执行PingRouter{}重写的 Handle() 方法，一个是MsgId为1的消息会执行HelloZinxRouter{}重写的 Handle() 方法。

我们现在写两个客户端，分别发送0消息和1消息来进行测试Zinx是否能够处理2个不同的消息业务。

### Client0.go

```
package main

import (
    "fmt"
    "io"
    "net"
    "time"
    "zinx/znet"
```

```

)
/*
    模拟客户端
*/
func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client start err, exit!")
        return
    }

    for {
        //发封包message消息
        dp := znet.NewDataPack()
        msg, _ := dp.Pack(znet.NewMsgPackage(0, []byte("Zinx V0.6
Client0 Test Message")))
        _, err := conn.Write(msg)
        if err !=nil {
            fmt.Println("write error err ", err)
            return
        }

        //先读出流中的head部分
        headData := make([]byte, dp.GetHeadLen())
        _, err = io.ReadFull(conn, headData) //ReadFull 会把msg填
充满为止
        if err != nil {
            fmt.Println("read head error")
            break
        }
        //将headData字节流 拆包到msg中
        msgHead, err := dp.Unpack(headData)
        if err != nil {
            fmt.Println("server unpack err:", err)
        }
    }
}

```

```

        return
    }

    if msgHead.GetDataLen() > 0 {
        //msg 是有data数据的，需要再次读取data数据
        msg := msgHead.(*znet.Message)
        msg.Data = make([]byte, msg.GetDataLen())

        //根据dataLen从io中读取字节流
        _, err := io.ReadFull(conn, msg.Data)
        if err != nil {
            fmt.Println("server unpack data err:", err)
            return
        }

        fmt.Println("==> Recv Msg: ID=", msg.Id, ", len=", msg.DataLen, ", data=", string(msg.Data))
    }

    time.Sleep(1*time.Second)
}
}

```

## Client1.go

```

package main

import (
    "fmt"
    "io"
    "net"
    "time"
    "zinx/znet"
)

/*
    模拟客户端
*/
func main() {

```

```

fmt.Println("Client Test ... start")
//3秒之后发起测试请求，给服务端开启服务的机会
time.Sleep(3 * time.Second)

conn, err := net.Dial("tcp", "127.0.0.1:7777")
if err != nil {
    fmt.Println("client start err, exit!")
    return
}

for {
    //发封包message消息
    dp := znet.NewDataPack()
    msg, _ := dp.Pack(znet.NewMsgPackage(1, []byte("Zinx V0.6
Client1 Test Message")))
    _, err := conn.Write(msg)
    if err !=nil {
        fmt.Println("write error err ", err)
        return
    }

    //先读出流中的head部分
    headData := make([]byte, dp.GetHeadLen())
    _, err = io.ReadFull(conn, headData) //ReadFull 会把msg填
充满为止
    if err != nil {
        fmt.Println("read head error")
        break
    }
    //将headData字节流 拆包到msg中
    msgHead, err := dp.Unpack(headData)
    if err != nil {
        fmt.Println("server unpack err:", err)
        return
    }

    if msgHead.GetDataLen() > 0 {
        //msg 是有data数据的，需要再次读取data数据
        msg := msgHead.(*znet.Message)
    }
}

```

## 6.3 使用Zinx-V0.6完成应用程序

```
msg.Data = make([]byte, msg.GetDataLen())

//根据dataLen从io中读取字节流
_, err := io.ReadFull(conn, msg.Data)
if err != nil {
    fmt.Println("server unpack data err:", err)
    return
}

fmt.Println("==> Recv Msg: ID=", msg.Id, ", len=", msg.DataLen, ", data=", string(msg.Data))
}

time.Sleep(1*time.Second)
}
}
```

分别执行服务端和两个客户端

```
$go run Server.go
```

```
$go run Client0.go
```

```
$go run Client1.go
```

服务端显示结果

```
$ go run Server.go
Add api msgId = 0
Add api msgId = 1
[START] Server name: zinx v-0.6 demoApp, listenner at IP: 127.0.0
.1, Port 7777 is starting
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
start Zinx server zinx v-0.6 demoApp succ, now listenning...
Reader Goroutine is running
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.6 Client0 Test Message
Reader Goroutine is running
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.6 Client1 Test Message
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.6 Client0 Test Message
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.6 Client1 Test Message
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.6 Client0 Test Message
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.6 Client1 Test Message
```

客户端0显示结果

```
$ go run Client0.go
Client Test ... start
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
```

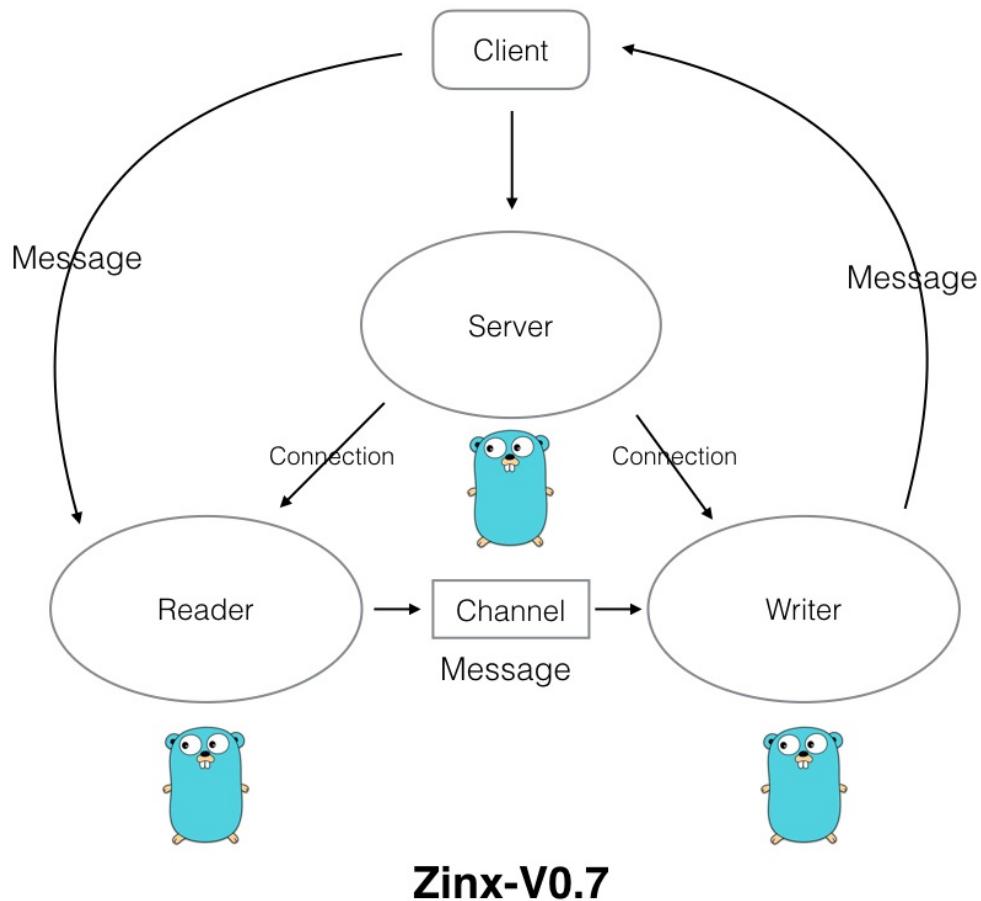
客户端1显示结果

```
$ go run Client1.go
Client Test ... start
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.6
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.6
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.6
```

## 七、Zinx的读写分离模型

好了，接下来我们就要对Zinx做一个小小的改变，就是与客户端进修数据交互的Gouroutine由一个变成两个，一个专门负责从客户端读取数据，一个专门负责向客户端写数据。这么设计有什么好处，当然是目的就是高内聚，模块的功能单一，对于我们今后扩展功能更加方便。

我们希望Zinx在升级到V0.7版本的时候，架构是下面这样的：



Server依然是处理客户端的响应，主要关键的几个方法是Listen、Accept等。当建立与客户端的套接字后，那么就会开启两个Goroutine分别处理读数据业务和写数据业务，读写数据之间的消息通过一个Channel传递。

## 7.1 Zinx-V0.7代码实现

我们的代码改动并不是很大。

### A) 添加读写模块交互数据的管道

zinx/znet/connection.go

```

type Connection struct {
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool
    //消息管理MsgId和对应处理方法的消息管理模块
    MsgHandler ziface.IMsgHandle
    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
    //无缓冲管道，用于读、写两个goroutine之间的消息通信
    msgChan      chan []byte
}

//创建连接的方法
func NewConntion(conn *net.TCPConn, connID uint32, msgHandler zi
face.IMsgHandle) *Connection{
    c := &Connection{
        Conn:      conn,
        ConnID:    connID,
        isClosed:  false,
        MsgHandler: msgHandler,
        ExitBuffChan: make(chan bool, 1),
        msgChan:make(chan []byte), //msgChan初始化
    }
    return c
}

```

我们给 `Connection` 新增一个管道成员 `msgChan` ,作用是用于读写两个go的通信。

## B) 创建Writer Goroutine

`zinx/znet/connection.go`

```
/*
    写消息Goroutine， 用户将数据发送给客户端
*/
func (c *Connection) StartWriter() {

    fmt.Println("[Writer Goroutine is running]")
    defer fmt.Println(c.RemoteAddr().String(), "[conn writer exit!"])

    for {
        select {
            case data := <-c.msgChan:
                //有数据要写给客户端
                if _, err := c.Conn.Write(data); err != nil {
                    fmt.Println("Send Data error:, ", err, " Conn Writer exit")
                    return
                }
            case <- c.ExitBuffChan:
                //conn已经关闭
                return
        }
    }
}
```

## C) Reader讲发送客户端的数据改为发送至Channel

修改Reader调用的 `SendMsg()` 方法

`zinx/znet/connection.go`

```
//直接将Message数据发送数据给远程的TCP客户端
func (c *Connection) SendMsg(msgId uint32, data []byte) error {
    if c.isClosed == true {
        return errors.New("Connection closed when send msg")
    }
    //将data封包，并且发送
    dp := NewDataPack()
    msg, err := dp.Pack(NewMsgPackage(msgId, data))
    if err != nil {
        fmt.Println("Pack error msg id = ", msgId)
        return errors.New("Pack error msg ")
    }

    //写回客户端
    c.msgChan <- msg    //将之前直接回写给conn.Write的方法 改为 发送给
    Channel 供Writer读取

    return nil
}
```

## D) 启动Reader和Writer

zinx/znet/connection.go

```
//启动连接，让当前连接开始工作
func (c *Connection) Start() {
    //1 开启用户从客户端读取数据流程的Goroutine
    go c.StartReader()
    //2 开启用于写回客户端数据流程的Goroutine
    go c.StartWriter()

    for {
        select {
        case <- c.ExitBuffChan:
            //得到退出消息，不再阻塞
            return
        }
    }
}
```

## 7.2 使用Zinx-V0.7完成应用程序

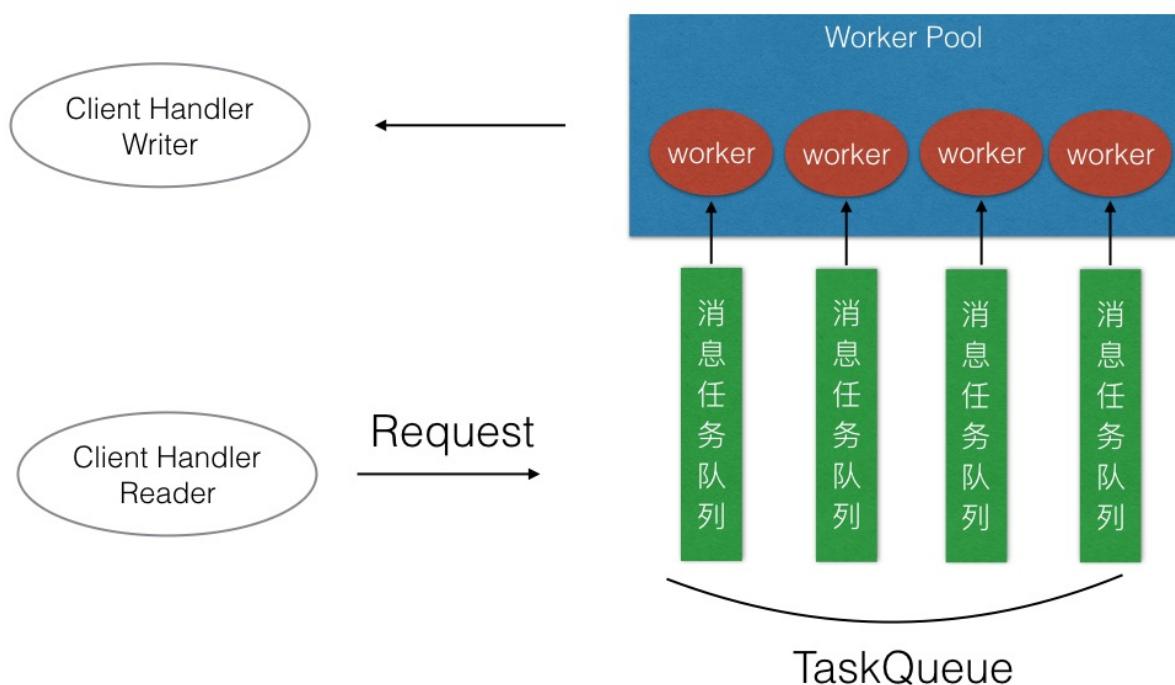
测试代码和V0.6的代码一样。

现在我们已经将读写模块分离了，那么接下来我们就可以再升级添加任务队列机制了。

## 八、Zinx的消息队列及多任务机制

接下来我们就需要给Zinx添加消息队列和多任务Worker机制了。我们可以通过worker的数量来限定处理业务的固定goroutine数量，而不是无限制的开辟Goroutine，虽然我们知道go的调度算法已经做的很极致了，但是大数量的Goroutine依然会带来一些不必要的环境切换成本，这些本应该是服务器应该节省掉的成本。我们可以用消息队列来缓冲worker工作的数据。

初步我们的设计结构如下图：



## 8.1 创建消息队列

首先，处理消息队列的部分，我们应该集成到 `MsgHandler` 模块下，因为属于我们消息模块范畴内的

`zinx/znet/msghandler.go`

```
type MsgHandle struct {
    Apis          map[uint32]ziface.IRouter //存放每个MsgId 所对应的处理方法的map属性
    WorkerPoolSize uint32                  //业务工作worker池的数量
    TaskQueue     []chan ziface IRequest   //Worker负责取任务的消息队列
}

func NewMsgHandle() *MsgHandle {
    return &MsgHandle{
        Apis: make(map[uint32]ziface.IRouter),
        WorkerPoolSize:utils.GlobalObject.WorkerPoolSize,
        //一个worker对应一个queue
        TaskQueue:make([]chan ziface IRequest, utils.GlobalObject.WorkerPoolSize),
    }
}
```

这里添加两个成员

`WorkerPoolSize` :作为工作池的数量，因为 `TaskQueue` 中的每个队列应该是和一个 `Worker` 对应的，所以我们在创建 `TaskQueue` 中队列数量要和 `Worker` 的数量一致。

`TaskQueue` 真是一个 `Request` 请求信息的 `channel` 集合。用来缓冲提供 `worker` 调用的 `Request` 请求信息，`worker` 会从对应的队列中获取客户端的请求数据并且处理掉。

当然 `WorkerPoolSize` 最好也可以从 `GlobalObject` 获取，并且 `zinx.json` 配置文件可以手动配置。

`zinx/utils/globalobj.go`

## 8.1 创建消息队列

```
/*
    存储一切有关Zinx框架的全局参数，供其他模块使用
    一些参数也可以通过 用户根据 zinx.json来配置
*/
type GlobalObj struct {
    /*
        Server
    */
    TcpServer ziface.IServer //当前Zinx的全局Server对象
    Host      string         //当前服务器主机IP
    TcpPort   int            //当前服务器主机监听端口号
    Name      string         //当前服务器名称

    /*
        Zinx
    */
    Version      string //当前Zinx版本号
    MaxPacketSize uint32 //都需数据包的最大值
    MaxConn      int     //当前服务器主机允许的最大链接个数
    WorkerPoolSize uint32 //业务工作Worker池的数量
    MaxWorkerTaskLen uint32 //业务工作Worker对应负责的任务队列最大任务
    //存储数量
}

/*
    config file path
*/
ConfFilePath string
}

//...
//...

/*
    提供init方法，默认加载
*/
func init() {
    //初始化GlobalObject变量，设置一些默认值
    GlobalObject = &GlobalObj{
        Name:      "ZinxServerApp",
        Version:   "V0.4",
    }
}
```

## 8.1 创建消息队列

```
TcpPort:      7777,  
Host:         "0.0.0.0",  
MaxConn:      12000,  
MaxPacketSize: 4096,  
ConfFilePath: "conf/zinx.json",  
WorkerPoolSize: 10,  
MaxWorkerTaskLen: 1024,  
}  
  
//从配置文件中加载一些用户配置的参数  
GlobalObject.Reload()  
}
```

## 8.2 创建及启动**Worker**工作池

现在添加**Worker**工作池，先定义一些启动工作池的接口

| zinx/ziface/msghandler.go

```
/*
消息管理抽象层
*/
type IMsgHandle interface{
    DoMsgHandler(request IRequest)           //马上以非阻塞方式处
    理消息
    AddRouter(msgId uint32, router IRouter)   //为消息添加具体的处
    理逻辑
    StartWorkerPool()                      //启动worker工作池
    SendMsgToTaskQueue(request IRequest)     //将消息交给TaskQueue,
    由worker进行处理
}
```

| zinx/znet/msghandler.go

```

//启动一个Worker工作流程
func (mh *MsgHandle) StartOneWorker(workerID int, taskQueue chan
ziface.IRequest) {
    fmt.Println("Worker ID = ", workerID, " is started.")
    //不断的等待队列中的消息
    for {
        select {
            //有消息则取出队列的Request，并执行绑定的业务方法
            case request := <-taskQueue:
                mh.DoMsgHandler(request)
        }
    }
}

//启动worker工作池
func (mh *MsgHandle) StartWorkerPool() {
    //遍历需要启动worker的数量，依此启动
    for i:= 0; i < int(mh.WorkerPoolSize); i++ {
        //一个worker被启动
        //给当前worker对应的任务队列开辟空间
        mh.TaskQueue[i] = make(chan ziface.IRequest, utils.Globa
lObject.MaxWorkerTaskLen)
        //启动当前Worker，阻塞的等待对应的任务队列是否有消息传递进来
        go mh.StartOneWorker(i, mh.TaskQueue[i])
    }
}

```

`StartWorkerPool()` 方法是启动Worker工作池，这里根据用户配置好的 `WorkerPoolSize` 的数量来启动，然后分别给每个Worker分配一个 `TaskQueue`，然后用一个goroutine来承载一个Worker的工作业务。

`StartOneWorker()` 方法就是一个Worker的工作业务，每个worker是不会退出的（目前没有设定worker的停止工作机制），会永久的从对应的`TaskQueue`中等待消息，并处理。

## 8.3 发送消息给消息队列

现在，worker工作池已经准备就绪了，那么就需要有一个给到worker工作池消息的入口，我们再定义一个方法

zinx/ziface/imsghandler.go

```
//将消息交给TaskQueue,由worker进行处理
func (mh *MsgHandle)SendMsgToTaskQueue(request ziface.IRequest)
{
    //根据ConnID来分配当前的连接应该由哪个worker负责处理
    //轮询的平均分配法则

    //得到需要处理此条连接的workerID
    workerID := request.GetConnection().GetConnID() % mh.WorkerPoolSize
    fmt.Println("Add ConnID=", request.GetConnection().GetConnID(),
               " request msgID=", request.GetMsgID(), "to workerID=", workerID)
    //将请求消息发送给任务队列
    mh.TaskQueue[workerID] <- request
}
```

SendMsgToTaskQueue() 作为工作池的数据入口，这里面采用的是轮询的分配机制，因为不同链接信息都会调用这个入口，那么到底应该由哪个worker处理该链接的请求处理，整理用的是一个简单的求模运算。用余数和workerID的匹配来进行分配。

最终将request请求数据发送给对应worker的TaskQueue，那么对应的worker的Goroutine就会处理该链接请求了。

## 8.4 Zinx-V0.8代码实现

好了，现在需要将消息队列和多任务worker机制集成到我们Zinx的中了。我们在Server的 `Start()` 方法中，在服务端Accept之前，启动Worker工作池。

zinx/znet/server.go

```
//开启网络服务
func (s *Server) Start() {
    //...
    //开启一个go去做服务端Linster业务
    go func() {
        //0 启动worker工作池机制
        s.msgHandler.StartWorkerPool()

        //1 获取一个TCP的Addr
        addr, err := net.ResolveTCPAddr(s.IPVersion, fmt.Sprintf(
            "%s:%d", s.IP, s.Port))
        if err != nil {
            fmt.Println("resolve tcp addr err: ", err)
            return
        }

        //...
        //...

    }
}()
```

其次，当我们已经得到客户端的连接请求过来数据的时候，我们应该将数据发送给Worker工作池进行处理。

所以应该在Connection的 `StartReader()` 方法中修改：

zinx/znet/connection.go

```

/*
    读消息Goroutine，用于从客户端中读取数据
*/
func (c *Connection) StartReader() {
    fmt.Println("Reader Goroutine is running")
    defer fmt.Println(c.RemoteAddr().String(), " conn reader exit!")
    defer c.Stop()

    for {
        // 创建拆包解包的对象...

        //读取客户端的Msg head...

        //拆包，得到msgid 和 datalen 放在msg中...

        //根据 dataLen 读取 data，放在msg.Data中...

        //得到当前客户端请求的Request数据
        req := Request{
            conn:c,
            msg:msg,
        }

        if utils.GlobalObject.WorkerPoolSize > 0 {
            //已经启动工作池机制，将消息交给Worker处理
            c.MsgHandler.SendMsgToTaskQueue(&req)
        } else {
            //从绑定好的消息和对应的处理方法中执行对应的Handle方法
            go c.MsgHandler.DoMsgHandler(&req)
        }
    }
}

```

这里并没有强制使用多任务Worker机制，而是判断用户配置 WorkerPoolSize 的个数，如果大于0，那么我就启动多任务机制处理链接请求消息，如果=0或者<0那么，我们依然只是之前的开启一个临时的Goroutine处理客户端请求消息。



## 8.5 使用Zinx-V0.8完成应用程序

测试代码和V0.6、V0.7的代码一样。因为Zinx框架对外接口没有发生改变。

我们分别启动Server、Client

```
$go run Server.go
```

```
$go run Client0.go
```

```
$go run Client1.go
```

```
$go run Client0.go
```

结果：

服务端：

```
$ go run Server.go
Add api msgId = 0
Add api msgId = 1
[START] Server name: zinx v-0.8 demoApp, listener at IP: 127.0.0
.1, Port 7777 is starting
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
Worker ID = 4 is started.
start Zinx server zinx v-0.8 demoApp succ, now listenning...
Worker ID = 9 is started.
Worker ID = 0 is started.
Worker ID = 5 is started.
Worker ID = 6 is started.
Worker ID = 1 is started.
Worker ID = 2 is started.
Worker ID = 7 is started.
Worker ID = 8 is started.
Worker ID = 3 is started.
```

```
Reader Goroutine is running
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Reader Goroutine is running
Add ConnID= 1 request msgID= 1 to workerID= 1
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.8 Client1 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Reader Goroutine is running
Add ConnID= 2 request msgID= 0 to workerID= 2
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 1 request msgID= 1 to workerID= 1
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.8 Client1 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 2 request msgID= 0 to workerID= 2
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 1 request msgID= 1 to workerID= 1
Call HelloZinxRouter Handle
recv from client : msgId= 1 , data= Zinx V0.8 Client1 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
```

### 客户端0

```
$ go run Client0.go
Client Test ... start
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
```

### 客户端1

```
$ go run Client1.go
Client Test ... start
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.8
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.8
==> Recv Msg: ID= 1 , len= 22 , data= Hello Zinx Router V0.8
```

### 客户端2

```
$ go run Client0.go
Client Test ... start
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
```

## 九、**Zinx**的链接管理

现在我们要为Zinx框架增加链接个数的限定，如果超过一定量的客户端个数，Zinx为了保证后端的及时响应，而拒绝链接请求。

## 9.1 创建链接管理模块

这里面我们就需要对链接有一个管理的模块。

我们在 `ziface` 和 `znet` 分别建立 `iconnmanager.go` 和 `connmanager.go` 文件

`zinx/ziface/iconmanager.go`

```
package ziface

/*
    连接管理抽象层
*/
type IConnManager interface {
    Add(conn IConnection)           //添加链接
    Remove(conn IConnection)        //删除连接
    Get(connID uint32) (IConnection, error) //利用ConnID获取链接
    Len() int                       //获取当前连接
    ClearConn()                    //删除并停止所有链接
}
```

这里定义了一些接口方法，添加链接、删除链接、根据ID获取链接、链接数量、和清除链接等。

`zinx/znet/connmanager.go`

```
package znet

import (
    "errors"
    "fmt"
    "sync"
    "zinx/ziface"
)

/*
    连接管理模块
*/
type ConnManager struct {
```

## 9.1 创建链接管理模块

```
connections map[uint32]ziface.IConnection //管理的连接信息
connLock sync.RWMutex //读写连接的读写锁
}

/*
    创建一个链接管理
*/
func NewConnManager() *ConnManager {
    return &ConnManager{
        connections:make(map[uint32] ziface.IConnection),
    }
}

//添加链接
func (connMgr *ConnManager) Add(conn ziface.IConnection) {
    //保护共享资源Map 加写锁
    connMgr.connLock.Lock()
    defer connMgr.connLock.Unlock()

    //将conn连接添加到ConnMananger中
    connMgr.connections[conn.GetConnID()] = conn

    fmt.Println("connection add to ConnManager successfully: conn num = ", connMgr.Len())
}

//删除连接
func (connMgr *ConnManager) Remove(conn ziface.IConnection) {
    //保护共享资源Map 加写锁
    connMgr.connLock.Lock()
    defer connMgr.connLock.Unlock()

    //删除连接信息
    delete(connMgr.connections, conn.GetConnID())

    fmt.Println("connection Remove ConnID=", conn.GetConnID(), " successfully: conn num = ", connMgr.Len())
}

//利用ConnID获取链接
```

## 9.1 创建链接管理模块

```
func (connMgr *ConnManager) Get(connID uint32) (ziface.IConnection, error) {
    //保护共享资源Map 加读锁
    connMgr.connLock.RLock()
    defer connMgr.connLock.RUnlock()

    if conn, ok := connMgr.connections[connID]; ok {
        return conn, nil
    } else {
        return nil, errors.New("connection not found")
    }
}

//获取当前连接
func (connMgr *ConnManager) Len() int {
    return len(connMgr.connections)
}

//清除并停止所有连接
func (connMgr *ConnManager) ClearConn() {
    //保护共享资源Map 加写锁
    connMgr.connLock.Lock()
    defer connMgr.connLock.Unlock()

    //停止并删除全部的连接信息
    for connID, conn := range connMgr.connections {
        //停止
        conn.Stop()
        //删除
        delete(connMgr.connections, connID)
    }

    fmt.Println("Clear All Connections successfully: conn num = "
    , connMgr.Len())
}
```

## 9.1 创建链接管理模块

---

这里面 `ConnManager` 中，其中用一个`map`来承载全部的连接信息，`key`是连接 ID，`value`则是连接本身。其中有一个读写锁 `connLock` 主要是针对`map`做多任务修改时的保护作用。

`Remove()` 方法只是单纯的将`conn`从`map`中摘掉，而 `ClearConn()` 方法则会先停止链接业务，`c.Stop()`，然后再从`map`中摘除。

## 9.2 链接管理模块集成到**Zinx**中

### A) ConnManager 集成到 Server 中

现在需要将 ConnManager 添加到 Server 中

| zinx/znet/server.go

```
//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
    //当前Server的消息管理模块，用来绑定MsgId和对应的处理方法
    msgHandler ziface.IMsgHandle
    //当前Server的链接管理器
    ConnMgr ziface.IConnManager
}

/*
    创建一个服务器句柄
*/
func NewServer () ziface.IServer {
    utils.GlobalObject.Reload()

    s:= &Server {
        Name :utils.GlobalObject.Name,
        IPVersion:"tcp4",
        IP:utils.GlobalObject.Host,
        Port:utils.GlobalObject.TcpPort,
        msgHandler: NewMsgHandle(),
        ConnMgr:NewConnManager(), //创建ConnManager
    }
    return s
}
```

那么，既然server具备了ConnManager成员，在获取的时候需要给抽象层提供一个获取ConnManager方法

zinx/ziface/iserver.go

```
type IServer interface{
    //启动服务器方法
    Start()
    //停止服务器方法
    Stop()
    //开启业务服务方法
    Serve()
    //路由功能：给当前服务注册一个路由业务方法，供客户端链接处理使用
    AddRouter(msgId uint32, router IRouter)
    //得到链接管理
    GetConnMgr() IConnManager
}
```

zinx/znet/server.go

```
//得到链接管理
func (s *Server) GetConnMgr() ziface.IConnManager {
    return s.ConnMgr
}
```

因为我们现在在server中有链接的管理，有的时候conn也需要得到这个ConnMgr的使用权，那么我们需要将 Server 和 Connection 建立能够互相索引的关系，我们在 Connection 中，添加Server当前conn隶属的server句柄。

zinx/znet/connection.go

```

type Connection struct {
    //当前Conn属于哪个Server
    TcpServer      ziface.IServer           //当前conn属于哪个server，在conn初始化的时候添加即可
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool
    //消息管理MsgId和对应处理方法的消息管理模块
    MsgHandler ziface.IMsgHandle
    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
    //无缓冲管道，用于读、写两个goroutine之间的消息通信
    msgChan       chan []byte
    //有关冲管道，用于读、写两个goroutine之间的消息通信
    msgBuffChan  chan []byte
}

```

## B) 链接的添加

那么我们什么选择将创建好的连接添加到 ConnManager 中呢，这里我们选择在初始化一个新链接的时候，加进来就好了

zinx/znet/connection.go

```

//创建连接的方法
func NewConnnection(server ziface.IServer, conn *net.TCPConn, connID uint32, msgHandler ziface.IMsgHandle) *Connection{
    //初始化Conn属性
    c := &Connection{
        TcpServer:server,      //将隶属的server传递进来
        Conn:      conn,
        ConnID:    connID,
        isClosed:  false,
        MsgHandler: msgHandler,
        ExitBuffChan: make(chan bool, 1),
        msgChan:make(chan []byte),
        msgBuffChan:make(chan []byte, utils.GlobalObject.MaxMsgChanLen),
    }

    //将新创建的Conn添加到链接管理中
    c.TcpServer.GetConnMgr().Add(c) //将当前新创建的连接添加到ConnManager中
    return c
}

```

### C) Server中添加链接数量的判断

在server的 Start() 方法中，在Accept与客户端链接建立成功后，可以直接对链接的个数做一个判断

zinx/znet/server.go

```

//开启网络服务
func (s *Server) Start() {
    fmt.Printf("[START] Server name: %s, listener at IP: %s, Port %d is starting\n", s.Name, s.IP, s.Port)
    fmt.Printf("[Zinx] Version: %s, MaxConn: %d, MaxPacketSize: %d\n",
        utils.GlobalObject.Version,
        utils.GlobalObject.MaxConn,
        utils.GlobalObject.MaxPacketSize)
}

```

```

//开启一个go去做服务端Linster业务
go func() {
    // ...

    //3 启动server网络连接业务
    for {
        //3.1 阻塞等待客户端建立连接请求
        conn, err := listenner.AcceptTCP()
        if err != nil {
            fmt.Println("Accept err ", err)
            continue
        }

        //=====
        //3.2 设置服务器最大连接控制，如果超过最大连接，那么则关闭此新的连接
        if s.ConnMgr.Len() >= utils.GlobalObject.MaxConn {
            conn.Close()
            continue
        }

        //=====

        //3.3 处理该新连接请求的 业务 方法， 此时应该有 handler 和 conn是绑定的
        dealConn := NewConntion(s, conn, cid, s.msgHandler)
        cid++

        //3.4 启动当前链接的处理业务
        go dealConn.Start()
    }
}()
}

```

当然，我们应该在配置文件 `zinx.json` 或者在 `GlobalObject` 全局配置中，定义好我们期望的连接的最大数目限制 `MaxConn`。

#### D) 连接的删除

我们应该在连接停止的时候，将该连接从ConnManager中删除，所以在 `connection` 的 `Stop()` 方法中添加。

## 9.2 链接管理模块集成到Zinx中

zinx/znet/connecion.go

```
func (c *Connection) Stop() {
    fmt.Println("Conn Stop()...ConnID = ", c.ConnID)
    //如果当前链接已经关闭
    if c.isClosed == true {
        return
    }
    c.isClosed = true

    // 关闭socket链接
    c.Conn.Close()
    //关闭Writer Goroutine
    c.ExitBuffChan <- true

    //将链接从连接管理器中删除
    c.TcpServer.GetConnMgr().Remove(c) //删除conn从ConnManager中

    //关闭该链接全部管道
    close(c.ExitBuffChan)
    close(c.msgBuffChan)
}
```

当然，我们也应该在 `server` 停止的时候，将全部的连接清空

zinx/znet/server.go

```
func (s *Server) Stop() {
    fmt.Println("[STOP] Zinx server , name " , s.Name)

    //将其他需要清理的连接信息或者其他信息 也要一并停止或者清理
    s.ConnMgr.ClearConn()
}
```

现在我们已经将连接管理成功的集成到了Zinx之中了。



## 9.3 链接的带缓冲的发包方法

我们之前给 `Connection` 提供了一个发消息的方法 `SendMsg()`，这个是将数据发送到一个无缓冲的channel中 `msgChan`。但是如果客户端链接比较多的话，如果对方处理不及时，可能会出现短暂的阻塞现象，我们可以做一个提供一定缓冲的发消息方法，做一些非阻塞的发送体验。

zinx/ziface/iconnection.go

```
// 定义连接接口
type IConnection interface {
    // 启动连接，让当前连接开始工作
    Start()
    // 停止连接，结束当前连接状态
    Stop()
    // 从当前连接获取原始的socket TCPConn
    GetTCPConnection() *net.TCPConn
    // 获取当前连接ID
    GetConnID() uint32
    // 获取远程客户端地址信息
    RemoteAddr() net.Addr
    // 直接将Message数据发送数据给远程的TCP客户端(无缓冲)
    SendMsg(msgId uint32, data []byte) error
    // 直接将Message数据发送给远程的TCP客户端(有缓冲)
    SendBuffMsg(msgId uint32, data []byte) error // 添加带缓冲发送消息接口
}
```

zinx/znet/connection.go

```
type Connection struct {
    // 当前Conn属于哪个Server
    TcpServer    ziface.IServer
    // 当前连接的socket TCP套接字
    Conn *net.TCPConn
    // 当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    // 当前连接的关闭状态
}
```

```

isClosed bool
//消息管理MsgId和对应处理方法的消息管理模块
MsgHandler ziface.IMsgHandle
//告知该链接已经退出/停止的channel
ExitBuffChan chan bool
//无缓冲管道，用于读、写两个goroutine之间的消息通信
msgChan chan []byte
//有关冲管道，用于读、写两个goroutine之间的消息通信
msgBuffChan chan []byte //定义channe
1成员
}

//创建连接的方法
func NewConntion(server ziface.IServer, conn *net.TCPConn, connID uint32, msgHandler ziface.IMsgHandle) *Connection{
    //初始化Conn属性
    c := &Connection{
        TcpServer:server,
        Conn:      conn,
        ConnID:    connID,
        isClosed:  false,
        MsgHandler: msgHandler,
        ExitBuffChan: make(chan bool, 1),
        msgChan:make(chan []byte),
        msgBuffChan:make(chan []byte, utils.GlobalObject.MaxMsgC
hanLen), //不要忘记初始化
    }

    //将新创建的Conn添加到链接管理中
    c.TcpServer.GetConnMgr().Add(c)
    return c
}

```

然后将 `SendBuffMsg()` 方法实现一下：

```
func (c *Connection) SendBuffMsg(msgId uint32, data []byte) error {
    if c.isClosed == true {
        return errors.New("Connection closed when send buff msg")
    }
    //将data封包，并且发送
    dp := NewDataPack()
    msg, err := dp.Pack(NewMsgPackage(msgId, data))
    if err != nil {
        fmt.Println("Pack error msg id = ", msgId)
        return errors.New("Pack error msg ")
    }

    //写回客户端
    c.msgBuffChan <- msg

    return nil
}
```

我们在Writer中也要有对 msgBuffChan 的数据监控：

```

/*
    写消息Goroutine， 用户将数据发送给客户端
*/
func (c *Connection) StartWriter() {
    fmt.Println("[Writer Goroutine is running]")
    defer fmt.Println(c.RemoteAddr().String(), "[conn Writer exit!"])

    for {
        select {
            case data := <-c.msgChan:
                //有数据要写给客户端
                if _, err := c.Conn.Write(data); err != nil {
                    fmt.Println("Send Data error:, ", err, " Conn Writer exit")
                    return
                }
                //针对有缓冲channel需要些的数据处理
            case data, ok:= <-c.msgBuffChan:
                if ok {
                    //有数据要写给客户端
                    if _, err := c.Conn.Write(data); err != nil {
                        fmt.Println("Send Buff Data error:, ", err, " Conn Writer exit")
                        return
                    }
                } else {
                    break
                    fmt.Println("msgBuffChan is Closed")
                }
            case <-c.ExitBuffChan:
                return
        }
    }
}

```



## 9.4 注册链接启动/停止自定义Hook方法功能

有的时候，在创建链接的时候，希望在创建链接之后、和断开链接之前，执行一些用户自定义的业务。那么我们就需要给Zinx增添两个链接创建后和断开前时机的回调函数，一般也称作Hook(钩子)函数。

我们可以通过Server来注册conn的hook方法

zinx/ziface/iserver.go

```
type IServer interface{
    //启动服务器方法
    Start()
    //停止服务器方法
    Stop()
    //开启业务服务方法
    Serve()
    //路由功能：给当前服务注册一个路由业务方法，供客户端链接处理使用
    AddRouter(msgId uint32, router IRouter)
    //得到链接管理
    GetConnMgr() IConnManager
    //设置该Server的连接创建时Hook函数
    SetOnConnStart(func (IConnection))
    //设置该Server的连接断开时的Hook函数
    SetOnConnStop(func (IConnection))
    //调用连接OnConnStart Hook函数
    CallOnConnStart(conn IConnection)
    //调用连接OnConnStop Hook函数
    CallOnConnStop(conn IConnection)
}
```

zinx/znet/server.go

```
//iServer 接口实现，定义一个Server服务类
type Server struct {
    //服务器的名称
    Name string
    //tcp4 or other
    IPVersion string
    //服务绑定的IP地址
    IP string
    //服务绑定的端口
    Port int
    //当前Server的消息管理模块，用来绑定MsgId和对应的处理方法
    msgHandler ziface.IMsgHandle
    //当前Server的链接管理器
    ConnMgr ziface.IConnManager

    // =====
    //新增两个hook函数原型

    //该Server的连接创建时Hook函数
    OnConnStart func(conn ziface.IConnection)
    //该Server的连接断开时的Hook函数
    OnConnStop func(conn ziface.IConnection)

    // =====
}
```

实现添加hook函数的接口和调用hook函数的接口

```

//设置该Server的连接创建时Hook函数
func (s *Server) SetOnConnStart(hookFunc func (ziface.IConnection)) {
    s.OnConnStart = hookFunc
}

//设置该Server的连接断开时的Hook函数
func (s *Server) SetOnConnStop(hookFunc func (ziface.IConnection)) {
    s.OnConnStop = hookFunc
}

//调用连接OnConnStart Hook函数
func (s *Server) CallOnConnStart(conn ziface.IConnection) {
    if s.OnConnStart != nil {
        fmt.Println("---> CallOnConnStart....")
        s.OnConnStart(conn)
    }
}

//调用连接OnConnStop Hook函数
func (s *Server) CallOnConnStop(conn ziface.IConnection) {
    if s.OnConnStop != nil {
        fmt.Println("---> CallOnConnStop....")
        s.OnConnStop(conn)
    }
}

```

那么接下来，需要选定两个Hook方法的调用位置。

一个是创建链接之后：

zinx/znet/connection.go

```
//启动连接，让当前连接开始工作
func (c *Connection) Start() {
    //1 开启用户从客户端读取数据流程的Goroutine
    go c.StartReader()
    //2 开启用于写回客户端数据流程的Goroutine
    go c.StartWriter()

    //=====
    //按照用户传递进来的创建连接时需要处理的业务，执行钩子方法
    c.TcpServer.CallOnConnStart(c)
    //=====
}
```

一个是停止链接之前：

| zinx/znet/connection.go

```
//停止连接，结束当前连接状态M
func (c *Connection) Stop() {
    fmt.Println("Conn Stop()...ConnID = ", c.ConnID)
    //如果当前链接已经关闭
    if c.isClosed == true {
        return
    }
    c.isClosed = true

    //=====
    //如果用户注册了该链接的关闭回调业务，那么在此刻应该显示调用
    c.TcpServer.CallOnConnStop(c)
    //=====

    // 关闭socket链接
    c.Conn.Close()
    //关闭writer
    c.ExitBuffChan <- true

    //将链接从连接管理器中删除
    c.TcpServer.GetConnMgr().Remove(c)

    //关闭该链接全部管道
    close(c.ExitBuffChan)
    close(c.msgBuffChan)
}
```

## 9.5 使用Zinx-V0.9完成应用程序

好了，现在我们基本上已经将全部的连接管理的功能集成到Zinx中了，接下来就需要测试一下链接管理模块是否可以使用了。

写一个服务端：

Server.go

```
package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Ping Handle
func (this *PingRouter) Handle(request ziface IRequest) {
    fmt.Println("Call PingRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
    ", data=", string(request.GetData()))

    err := request.GetConnection().SendBuffMsg(0, []byte("ping..
.ping...ping"))
    if err != nil {
        fmt.Println(err)
    }
}

type HelloZinxRouter struct {
    znet.BaseRouter
}
```

```

//HelloZinxRouter Handle
func (this *HelloZinxRouter) Handle(request ziface.IRequest) {
    fmt.Println("Call HelloZinxRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
", data=", string(request.GetData())))

    err := request.GetConnection().SendBuffMsg(1, []byte("Hello
Zinx Router V0.8"))
    if err != nil {
        fmt.Println(err)
    }
}

//创建连接的时候执行
func DoConnectionBegin(conn ziface.IConnection) {
    fmt.Println("DoConnecionBegin is Called ... ")
    err := conn.SendMsg(2, []byte("DoConnection BEGIN..."))
    if err != nil {
        fmt.Println(err)
    }
}

//连接断开的时候执行
func DoConnectionLost(conn ziface.IConnection) {
    fmt.Println("DoConneciotnLost is Called ... ")
}

func main() {
    //创建一个server句柄
    s := znet.NewServer()

    //注册链接hook回调函数
    s.SetOnConnStart(DoConnectionBegin)
    s.SetOnConnStop(DoConnectionLost)

    //配置路由
    s.AddRouter(0, &PingRouter{})
    s.AddRouter(1, &HelloZinxRouter{})
}

```

```
//开启服务
s.Serve()
}
```

我们这里注册了两个Hook函数一个是链接初始化之后 `DoConnectionBegin()` 和链接停止之前 `DoConnectionLost()`。

`DoConnectionBegin()` 会发给客户端一个消息2的文本，并且在服务端打印一个调试信息"DoConnecionBegin is Called ..."

`DoConnectionLost()` 在服务端打印一个调试信息"DoConneciotnLost is Called ..."

客户端：

Client.go

```
package main

import (
    "fmt"
    "io"
    "net"
    "time"
    "zinx/znet"
)

/*
    模拟客户端
*/
func main() {

    fmt.Println("Client Test ... start")
    //3秒之后发起测试请求，给服务端开启服务的机会
    time.Sleep(3 * time.Second)

    conn, err := net.Dial("tcp", "127.0.0.1:7777")
    if err != nil {
        fmt.Println("client start err, exit!")
    }
}
```

```

        return
    }

    for {
        //发封包message消息
        dp := znet.NewDataPack()
        msg, _ := dp.Pack(znet.NewMsgPackage(0, []byte("Zinx V0.8
Client0 Test Message")))
        _, err := conn.Write(msg)
        if err != nil {
            fmt.Println("write error err ", err)
            return
        }

        //先读出流中的head部分
        headData := make([]byte, dp.GetHeadLen())
        _, err = io.ReadFull(conn, headData) //ReadFull 会把msg填
充满为止
        if err != nil {
            fmt.Println("read head error")
            break
        }
        //将headData字节流 拆包到msg中
        msgHead, err := dp.Unpack(headData)
        if err != nil {
            fmt.Println("server unpack err:", err)
            return
        }

        if msgHead.GetDataLen() > 0 {
            //msg 是有data数据的，需要再次读取data数据
            msg := msgHead.(*znet.Message)
            msg.Data = make([]byte, msg.GetDataLen())

            //根据dataLen从io中读取字节流
            _, err := io.ReadFull(conn, msg.Data)
            if err != nil {
                fmt.Println("server unpack data err:", err)
                return
            }
        }
    }
}

```

```
        fmt.Println("==> Recv Msg: ID=", msg.Id, ", len=", msg.DataLen, ", data=", string(msg.Data))
    }

    time.Sleep(1*time.Second)
}
}
```

代码不变。

启动服务端

```
$go run Server.go
```

启动客户端

```
$go run Client.go
```

服务端结果：

```
$ go run Server.go
Add api msgId = 0
Add api msgId = 1
[START] Server name: zinx v-0.8 demoApp, listenner at IP: 127.0.0
.1, Port 7777 is starting
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
start Zinx server  zinx v-0.8 demoApp  succ, now listenning...
Worker ID = 9  is started.
Worker ID = 5  is started.
Worker ID = 6  is started.
Worker ID = 7  is started.
Worker ID = 8  is started.
Worker ID = 1  is started.
Worker ID = 0  is started.
Worker ID = 2  is started.
Worker ID = 3  is started.
Worker ID = 4  is started.
```

```
connection add to ConnManager successfully: conn num = 1
---> CallOnConnStart....
DoConnecionBegin is Called ...
[Writer Goroutine is running]
[Reader Goroutine is running]
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0 request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
read msg head error  read tcp4 127.0.0.1:7777->127.0.0.1:49510:
read: connection reset by peer
Conn Stop()...ConnID = 0
---> CallOnConnStop....
DoConneciotnLost is Called ...
connection Remove ConnID= 0 successfully: conn num = 0
127.0.0.1:49510 [conn Reader exit!]
127.0.0.1:49510 [conn Writer exit!]
```

客户端结果：

```
$ go run Client0.go
Client Test ... start
==> Recv Msg: ID= 2 , len= 21 , data= DoConnection BEGIN...
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
^Csignal: interrupt
```

客户端创建成功，回调Hook已经执行，并且Conn被添加到ConnManager中，  
conn num = 1，

当我们手动CTRL+C 关闭客户端的时候，服务器ConnManager已经成功将Conn摘掉，conn num = 0.

同时服务端也打印出 conn停止之后的回调信息。

## 十、Zinx的连接属性设置

当我们在使用链接处理的时候，希望和链接绑定一些用户的数据，或者参数。那么我们现在可以把当前链接设定一些传递参数的接口或者方法。

## 10.1 给链接添加链接配置接口

zinx/ziface/iconnection.go

```
//定义连接接口
type IConnection interface {
    //启动连接，让当前连接开始工作
    Start()
    //停止连接，结束当前连接状态
    Stop()

    //从当前连接获取原始的socket TCPConn
    GetTCPConnection() *net.TCPConn
    //获取当前连接ID
    GetConnID() uint32
    //获取远程客户端地址信息
    RemoteAddr() net.Addr

    //直接将Message数据发送数据给远程的TCP客户端(无缓冲)
    SendMsg(msgId uint32, data []byte) error
    //直接将Message数据发送给远程的TCP客户端(有缓冲)
    SendBuffMsg(msgId uint32, data []byte) error

    //设置链接属性
    SetProperty(key string, value interface{})
    //获取链接属性
    GetProperty(key string)(interface{}, error)
    //移除链接属性
    RemoveProperty(key string)
}
```

这里增添了3个方法 `SetProperty()` , `GetProperty()` , `RemoveProperty()` .那么`property`是什么类型的呢，我么接下来看看`Connection`的定义。

## 10.2 链接属性方法实现

zinx/znet/connection.go

```

type Connection struct {
    //当前Conn属于哪个Server
    TcpServer ziface.IServer
    //当前连接的socket TCP套接字
    Conn *net.TCPConn
    //当前连接的ID 也可以称作为SessionID，ID全局唯一
    ConnID uint32
    //当前连接的关闭状态
    isClosed bool
    //消息管理MsgId和对应处理方法的消息管理模块
    MsgHandler ziface.IMsgHandle
    //告知该链接已经退出/停止的channel
    ExitBuffChan chan bool
    //无缓冲管道，用于读、写两个goroutine之间的消息通信
    msgChan chan []byte
    //有关冲管道，用于读、写两个goroutine之间的消息通信
    msgBuffChan chan []byte

    // =====
    //链接属性
    property map[string]interface{}
    //保护链接属性修改的锁
    propertyLock sync.RWMutex
    // =====
}

//创建连接的方法
func NewConnection(server ziface.IServer, conn *net.TCPConn, connID uint32, msgHandler ziface.IMsgHandle) *Connection {
    //初始化Conn属性
    c := &Connection{
        TcpServer:    server,
        Conn:         conn,
        ConnID:       connID,
        isClosed:     false,
}

```

```

        MsgHandler:    msgHandler,
        ExitBuffChan: make(chan bool, 1),
        msgChan:       make(chan []byte),
        msgBuffChan:   make(chan []byte, utils.GlobalObject.MaxMs
gChanLen),
        property:      make(map[string]interface{}), //对链接属性ma
p初始化
    }

    //将新创建的Conn添加到链接管理中
    c.TcpServer.GetConnMgr().Add(c)
    return c
}

// ...

//设置链接属性
func (c *Connection) SetProperty(key string, value interface{}) {
    c.propertyLock.Lock()
    defer c.propertyLock.Unlock()

    c.property[key] = value
}

//获取链接属性
func (c *Connection) GetProperty(key string) (interface{}, error) {
    c.propertyLock.RLock()
    defer c.propertyLock.RUnlock()

    if value, ok := c.property[key]; ok {
        return value, nil
    } else {
        return nil, errors.New("no property found")
    }
}

//移除链接属性
func (c *Connection) RemoveProperty(key string) {
}

```

```
c.propertyLock.Lock()  
defer c.propertyLock.Unlock()  
  
delete(c.property, key)  
}
```

## 10.3 链接属性Zinx-V0.10单元测试

那么，接下来，我们简单测试一下链接属性的设置与提取是否可用。

Server.go

```
package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/znet"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Ping Handle
func (this *PingRouter) Handle(request ziface IRequest) {
    fmt.Println("Call PingRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
    ", data=", string(request.GetData()))

    err := request.GetConnection().SendBuffMsg(0, []byte("ping..
.ping...ping"))
    if err != nil {
        fmt.Println(err)
    }
}

type HelloZinxRouter struct {
    znet.BaseRouter
}

//HelloZinxRouter Handle
func (this *HelloZinxRouter) Handle(request ziface IRequest) {
```

```

    fmt.Println("Call HelloZinxRouter Handle")
    //先读取客户端的数据，再回写ping...ping...ping
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
    ", data=", string(request.GetData()))

    err := request.GetConnection().SendBuffMsg(1, []byte("Hello
Zinx Router V0.10"))
    if err != nil {
        fmt.Println(err)
    }
}

//创建连接的时候执行
func DoConnectionBegin(conn ziface.IConnection) {
    fmt.Println("DoConnectionBegin is Called ... ")

    //=====设置两个链接属性，在连接创建之后=====
    fmt.Println("Set conn Name, Home done!")
    conn SetProperty("Name", "Aceld")
    conn SetProperty("Home", "https://www.jianshu.com/u/35261429
b7f1")
    //=====

    err := conn.SendMsg(2, []byte("DoConnection BEGIN..."))
    if err != nil {
        fmt.Println(err)
    }
}

//连接断开的时候执行
func DoConnectionLost(conn ziface.IConnection) {
    //=====在连接销毁之前，查询conn的Name，Home属性=====
    if name, err := conn.GetProperty("Name"); err == nil {
        fmt.Println("Conn Property Name = ", name)
    }

    if home, err := conn.GetProperty("Home"); err == nil {
        fmt.Println("Conn Property Home = ", home)
    }
    //=====
}

```

```

    fmt.Println("DoConneciotnLost is Called ... ")
}

func main() {
    //创建一个server句柄
    s := znet.NewServer()

    //注册链接hook回调函数
    s.SetOnConnStart(DoConnectionBegin)
    s.SetOnConnStop(DoConnectionLost)

    //配置路由
    s.AddRouter(0, &PingRouter{})
    s.AddRouter(1, &HelloZinxRouter{})

    //开启服务
    s.Serve()
}

```

这里主要看 `DoConnectionBegin()` 和 `DoConnectionLost()` 两个函数的实现，利用在两个Hook函数中，设置链接属性和提取链接属性。链接创建之后给当前链接绑定两个属性"Name","Home"，那么我们在随时可以通过 `conn.GetProperty()` 方法得到链接已经设置的属性。

```
$go run Server.go
```

```
$go run Client0.go
```

服务端：

```

$ go run Server.go
Add api msgId = 0
Add api msgId = 1
[START] Server name: zinx v-0.10 demoApp, listenner at IP: 127.0.
0.1, Port 7777 is starting

```

```
[Zinx] Version: V0.4, MaxConn: 3, MaxPacketSize: 4096
start Zinx server    zinx v-0.10 demoApp  succ, now listenning...
Worker ID = 9  is started.
Worker ID = 5  is started.
Worker ID = 6  is started.
Worker ID = 7  is started.
Worker ID = 8  is started.
Worker ID = 1  is started.
Worker ID = 0  is started.
Worker ID = 2  is started.
Worker ID = 3  is started.
Worker ID = 4  is started.
connection add to ConnManager successfully: conn num = 1
---> CallOnConnStart....
DoConnecionBegin is Called ...
Set conn Name, Home done!
[Writer Goroutine is running]
[Reader Goroutine is running]
Add ConnID= 0  request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0  request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
Add ConnID= 0  request msgID= 0 to workerID= 0
Call PingRouter Handle
recv from client : msgId= 0 , data= Zinx V0.8 Client0 Test Message
read msg head error  read tcp4 127.0.0.1:7777->127.0.0.1:55208:
read: connection reset by peer
Conn Stop()...ConnID = 0
---> CallOnConnStop....
Conn Property Name = Aceld
Conn Property Home = https://www.jianshu.com/u/35261429b7f1
DoConneciotnLost is Called ...
connection Remove ConnID= 0  successfully: conn num = 0
127.0.0.1:55208 [conn Reader exit!]
127.0.0.1:55208 [conn Writer exit!]
```

客户端：

```
$ go run Client0.go
Client Test ... start
==> Recv Msg: ID= 2 , len= 21 , data= DoConnection BEGIN...
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
==> Recv Msg: ID= 0 , len= 18 , data= ping...ping...ping
^Csignal: interrupt
```

当我们终止客户端链接，那么服务端在断开链接之前，已经读取到了conn的两个属性Name和Home



## 一、应用案例介绍

好了，以上Zinx的框架的一些核心功能我们已经完成了，那么接下来我们就要基于Zinx完成一个服务端的应用程序了，整理用一个游戏应用服务器作为Zinx的一个应用案例。

游戏场景是一款MMO大型多人在线游戏，带unity3d客户端的服务器端demo，该demo实现了mmo游戏的基础模块aoi(基于兴趣范围的广播)，世界聊天等。



## 二、服务器应用基础协议

<b>MsgID</b>	<b>Client</b>	<b>Server</b>	描述
1	-	SyncPid	同步玩家本次登录的ID(用来标识玩家)
2	Talk	-	世界聊天
3	MovePackege	-	移动
200	-	BroadCast	广播消息(Tp 1 世界聊天 2 坐标(出生点同步) 3 动作 4 移动之后坐标信息更新)
201	-	SyncPid	广播消息 掉线/aoi消失在视野
202	-	SyncPlayers	同步周围的人位置信息(包括自己)

## 三、MMO多人在线游戏AOI算法

游戏的AOI(Area Of Interest)算法应该算作游戏的基础核心了，许多逻辑都是因为AOI进出事件驱动的，许多网络同步数据也是因为AOI进出事件产生的。因此，良好的AOI算法和基于AOI算法的优化，是提高游戏性能的关键。

为此，需要为每个玩家设定一个AOI，当一个对象状态发生改变时，需要将信息广播给全部玩家，那些AOI覆盖到的玩家都会收到这条广播消息，从而做出对应的响应状态。

功能：

1. 服务器上的玩家或 NPC 状态发生改变时，将消息广播到附近的玩家。
2. 玩家进入NPC警戒区域时，AOI 模块将消息发送给NPC，NPC再做出相应的AI反应。

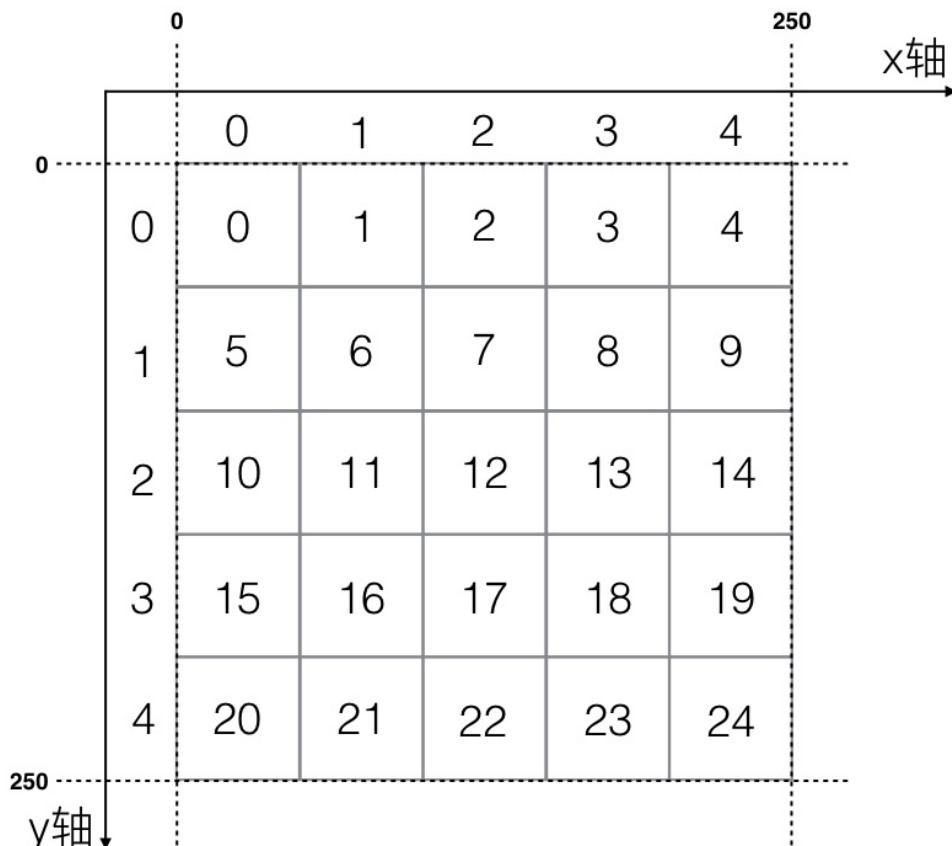
下面我们来创建一个mmo游戏，首先创建一个文件夹

mmo\_game/

mmo\_game 作为我们服务端游戏应用的主项目目录

## 3.1 网络法实现AOI算法

让我们首先绘制一个2D的地图



我们给这个地图定义一些数值：

场景相关数值计算

- 场景大小： $250 \times 250$ ，  $w(x\text{轴宽度}) = 250$ ，  $l(y\text{轴长度}) = 250$
- $x$ 轴格子数量： $nx = 5$
- $y$ 轴格子数量： $ny = 5$
- 格子宽度:  $dx = w / nx = 250 / 5 = 50$
- 格子长度:  $dy = l / ny = 250 / 5 = 50$
- 格子的 $x$ 轴坐标： $idx$
- 格子的 $y$ 轴坐标： $idy$

- 格子编号： $\text{id} = \text{idy} * \text{nx} + \text{idx}$  (利用格子坐标得到格子编号)
- 格子坐标： $\text{idx} = \text{id} \% \text{nx}$ ,  $\text{idy} = \text{id} / \text{nx}$  (利用格子id得到格子坐标)
- 格子的x轴坐标： $\text{idx} = \text{id} \% \text{nx}$  (利用格子id得到x轴坐标编号)
- 格子的y轴坐标： $\text{idy} = \text{id} / \text{nx}$  (利用格子id得到y轴坐标编号)

以上几个数值，请参考图，简单过一下，就可以理解的，初中的几何计算而已。

## 3.2 实现AOI格子结构

将aoi模块放在一个 core 模块中

mmo\_game/core/grid.go

```
package core

import "sync"

/*
    一个地图中的格子类
*/
type Grid struct {
    GID        int          //格子ID
    MinX      int          //格子左边界坐标
    MaxX      int          //格子右边界坐标
    MinY      int          //格子上边界坐标
    MaxY      int          //格子下边界坐标
    playerIDs map[int]bool //当前格子内的玩家或者物体成员ID
    pIDLock   sync.RWMutex //playerIDs的保护map的锁
}

//初始化一个格子
func NewGrid(gID, minX, maxX, minY, maxY int) *Grid {
    return &Grid{
        GID:gID,
        MinX:minX,
        MaxX:maxX,
        MinY:minY,
        MaxY:maxY,
        playerIDs:make(map[int] bool),
    }
}

//向当前格子中添加一个玩家
func (g *Grid) Add(playerID int) {
    g.pIDLock.Lock()
    defer g.pIDLock.Unlock()
```

```

    g.playerIDs[playerID] = true
}

//从格子中删除一个玩家
func (g *Grid) Remove(playerID int) {
    g.pIDLock.Lock()
    defer g.pIDLock.Unlock()

    delete(g.playerIDs, playerID)
}

//得到当前格子中所有的玩家
func (g *Grid) GetPlayerIDs() (playerIDs []int) {
    g.pIDLock.RLock()
    defer g.pIDLock.RUnlock()

    for k, _ := range g.playerIDs {
        playerIDs = append(playerIDs, k)
    }

    return
}

//打印信息方法
func (g *Grid) String() string {
    return fmt.Sprintf("Grid id: %d, minX:%d, maxX:%d, minY:%d,
maxY:%d, playerIDs:%v",
        g.GID, g.MinX, g.MaxX, g.MinY, g.MaxY, g.playerIDs)
}

```

`Grid` 这个格子类型，很好理解，分别有上下左右四个坐标，确定格子的领域范围，还是有格子ID，其中 `playerIDs` 是一个map，表示当前格子中存在的玩家有哪些。这里提供了一个方法 `GetPlayerIDs()` 可以返回当前格子中所有玩家的ID切片。

### 3.3 实现AOI管理模块

那么接下来我们就要对格子添加到一个AOI模块中进行管理。我们创建一个aoi模块文件。

mmo\_game/core/aoi.go

```
package core

/*
AOI管理模块
*/
type AOIManager struct {
    MinX  int          //区域左边界坐标
    MaxX  int          //区域右边界坐标
    CntsX int          //x方向格子的数量
    MinY  int          //区域上边界坐标
    MaxY  int          //区域下边界坐标
    CntsY int          //y方向的格子数量
    grids map[int]*Grid //当前区域中都有哪些格子，key=格子ID， value=
格子对象
}

/*
初始化一个AOI区域
*/
func NewAOIManager(minX, maxX, cntsX, minY, maxY, cntsY int) *AO
IManager {
    aoiMgr := &AOIManager{
        MinX:  minX,
        MaxX:  maxX,
        CntsX: cntsX,
        MinY:  minY,
        MaxY:  maxY,
        CntsY: cntsY,
        grids: make(map[int]*Grid),
    }
    //给AOI初始化区域中所有的格子
}
```

### 3.3 实现AOI管理模块

```
for y := 0; y < cntsY; y++ {
    for x := 0; x < cntsX; x++ {
        //计算格子ID
        //格子编号：id = idy *nx + idx (利用格子坐标得到格子编号)
        gid := y*cntsX + x

        //初始化一个格子放在AOI中的map里，key是当前格子的ID
        aoiMgr.grids[gid] = NewGrid(gid,
            aoiMgr.MinX+ x*aoiMgr.gridWidth(),
            aoiMgr.MinX+(x+1)*aoiMgr.gridWidth(),
            aoiMgr.MinY+ y*aoiMgr.gridLength(),
            aoiMgr.MinY+(y+1)*aoiMgr.gridLength())
    }
}

return aoiMgr
}

//得到每个格子在x轴方向的宽度
func (m *AOIManager) gridWidth() int {
    return (m.MaxX - m.MinX) / m.CntsX
}

//得到每个格子在x轴方向的长度
func (m *AOIManager) gridLength() int {
    return (m.MaxY - m.MinY) / m.CntsY
}

//打印信息方法
func (m *AOIManager) String() string {
    s := fmt.Sprintf("AOIManagr:\nminX:%d, maxX:%d, cntsX:%d, mi
nY:%d, maxY:%d, cntsY:%d\n Grids in AOI Manager:\n",
        m.MinX, m.MaxX, m.CntsX, m.MinY, m.MaxY, m.CntsY)
    for _,grid := range m.grids {
        s += fmt.Sprintln(grid)
    }

    return s
}
```

### 3.3 实现AOI管理模块

---

以上是创建一个AOI模块(可以理解为一个2D的矩形地图)，里面有若干份 grids 。

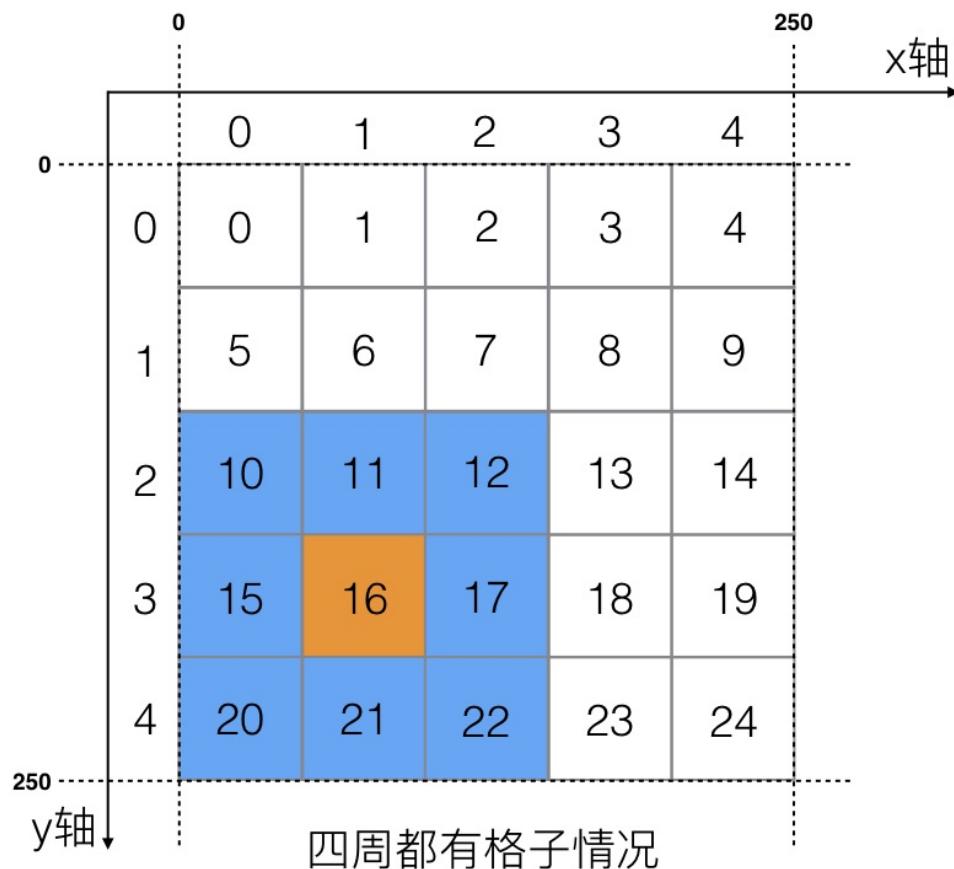
NewAOIManager() 会平均划分多分小格子，并初始化格子的坐标，计算方式很简单，初步的几何计算。

## 3.4 求出九宫格

### A) 根据格子ID求出九宫格

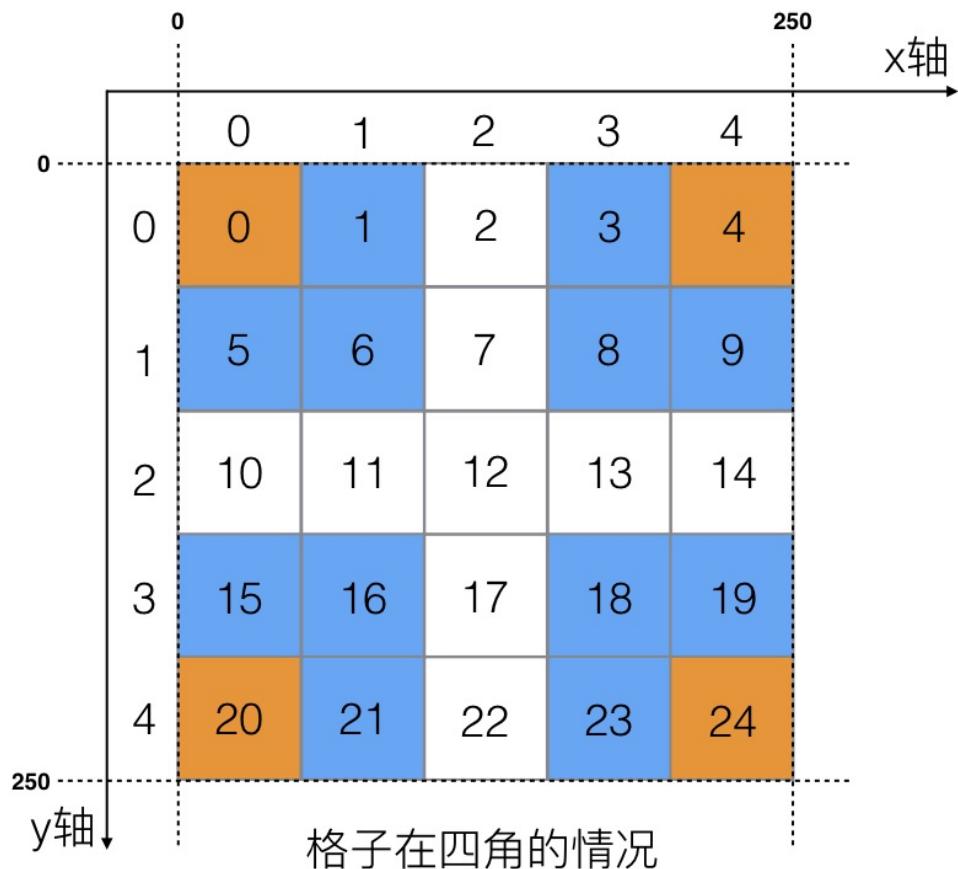
现在我们根据格子ID `gid` 求出周边的格子有哪些？

我们可能要考虑一些情况，比如格子的四周都有，如下图：



或者，格子所在AOI区域的四个顶角，如下图：

### 3.4 求出九宫格



或者给所在AOI边界，周边的格子缺少一列，或者缺少一行。

思考一下，我们是否可以想一个统一的方法，将所有的条件都满足。

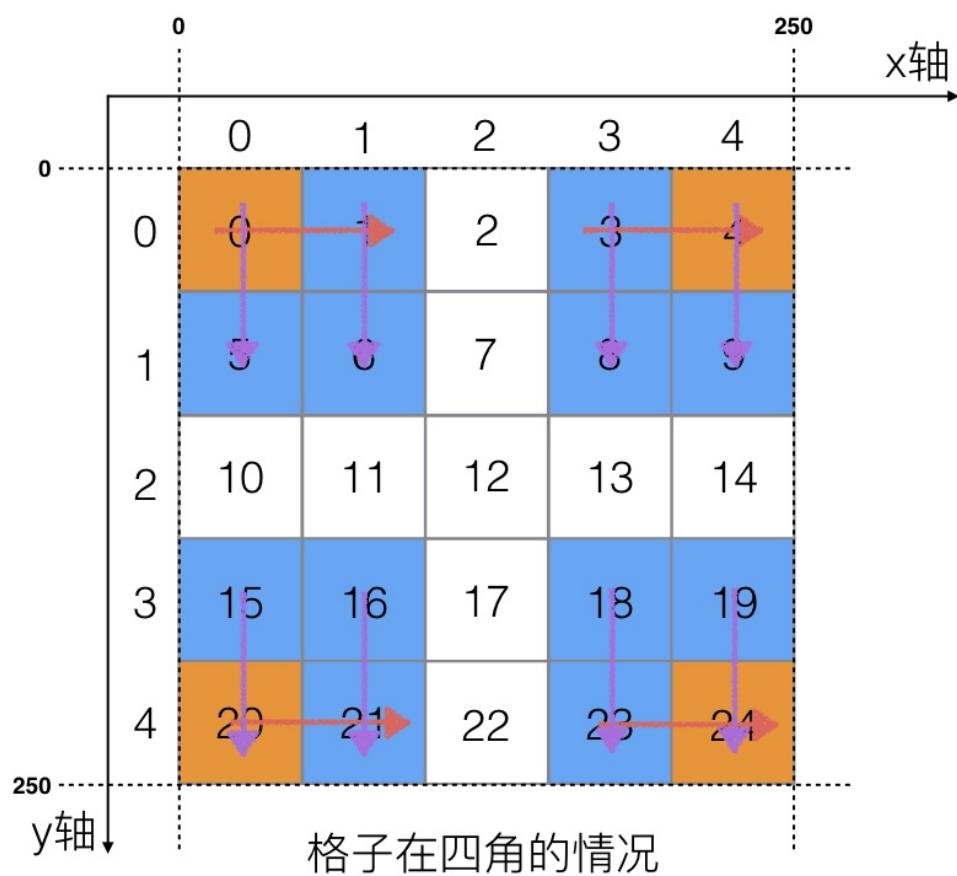
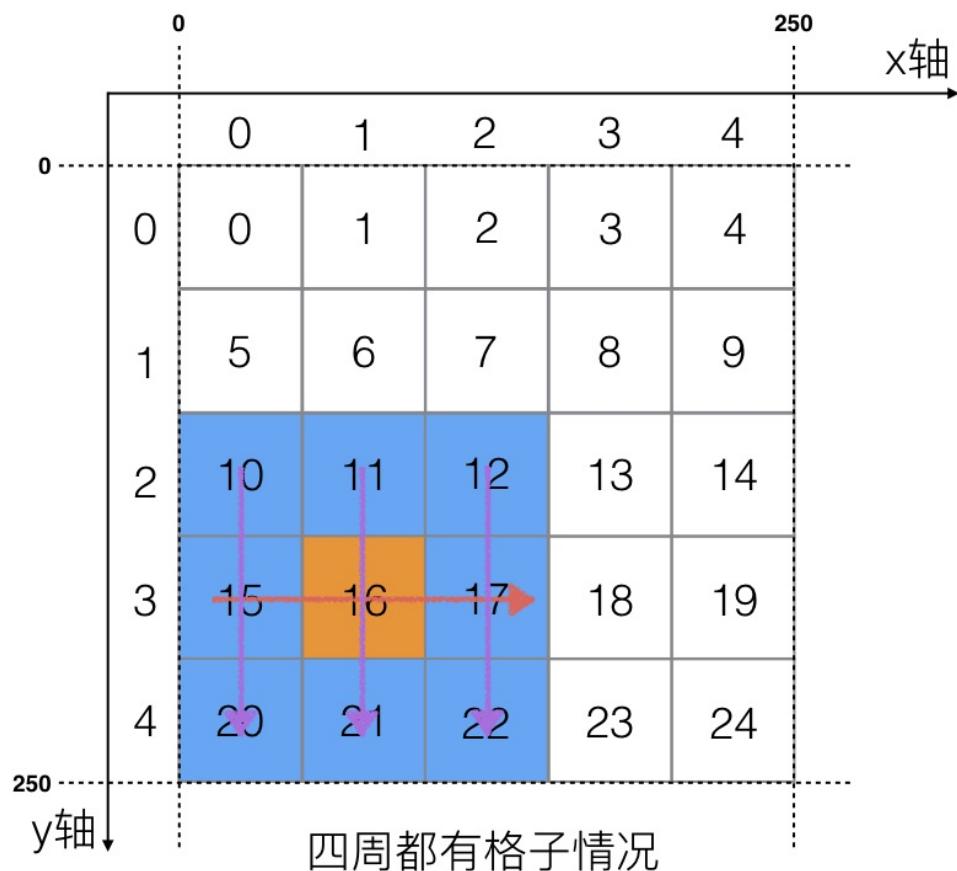
如果求出一个gid的周边九宫格，那么可以先算出该gid所处一行左边和右边是否有，然后在分别计算这一行的上边和下边的格子是否有，就可以了。

思路如下：

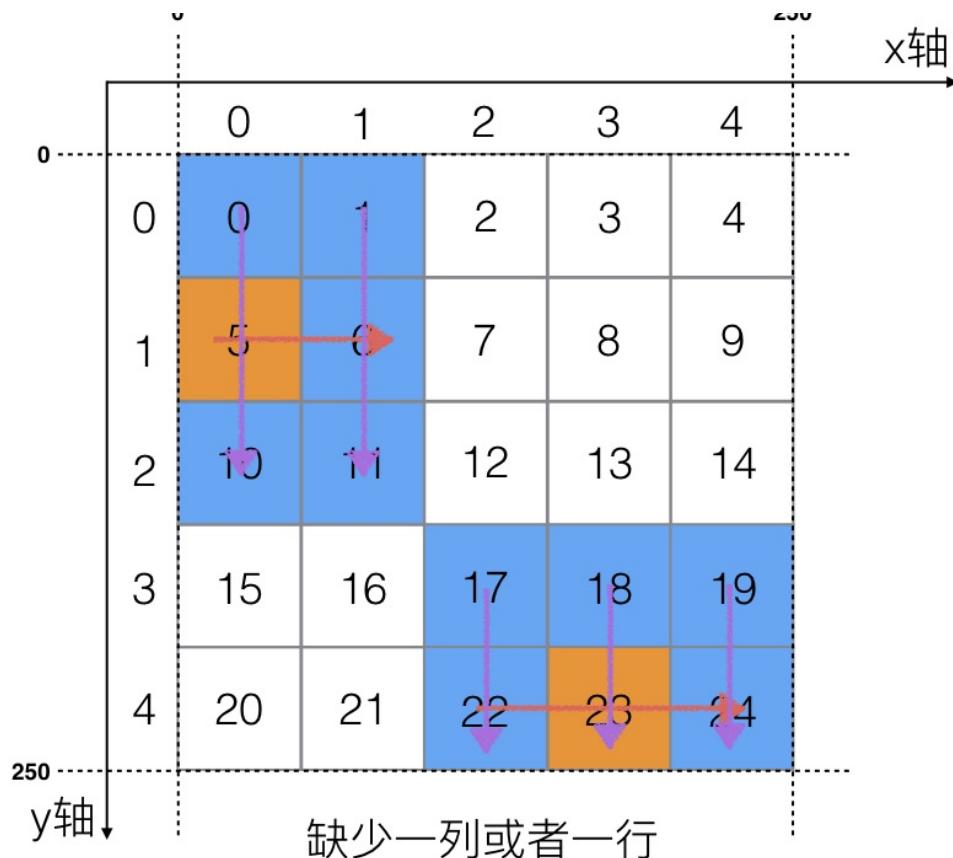
### 3.4 求出九宫格

---

### 3.4 求出九宫格



### 3.4 求出九宫格



参考代码如下：

mmo\_game/core/aoi.go

```
//根据格子的gID得到当前周边的九宫格信息
func (m *AOIManager) GetSurroundGridsByGid(gID int) (grids []*Grid)
{
    //判断gID是否存在
    if _, ok := m.grids[gID]; !ok {
        return
    }

    //将当前gid添加到九宫格中
    grids = append(grids, m.grids[gID])

    //根据gid得到当前格子所在的X轴编号
    idx := gID % m.CntsX

    //判断当前idx左边是否还有格子
    if idx > 0 {
        grids = append(grids, m.grids[gID-1])
    }
}
```

### 3.4 求出九宫格

```
//判断当前的idx右边是否还有格子
if idx < m.CntsX - 1 {
    grids = append(grids, m.grids[gID+1])
}

//将x轴当前的格子都取出，进行遍历，再分别得到每个格子的上下是否有格子

//得到当前x轴的格子id集合
gidsX := make([]int, 0, len(grids))
for _, v := range grids {
    gidsX = append(gidsX, v.GID)
}

//遍历x轴格子
for _, v := range gidsX {
    //计算该格子处于第几列
    idy := v / m.CntsX

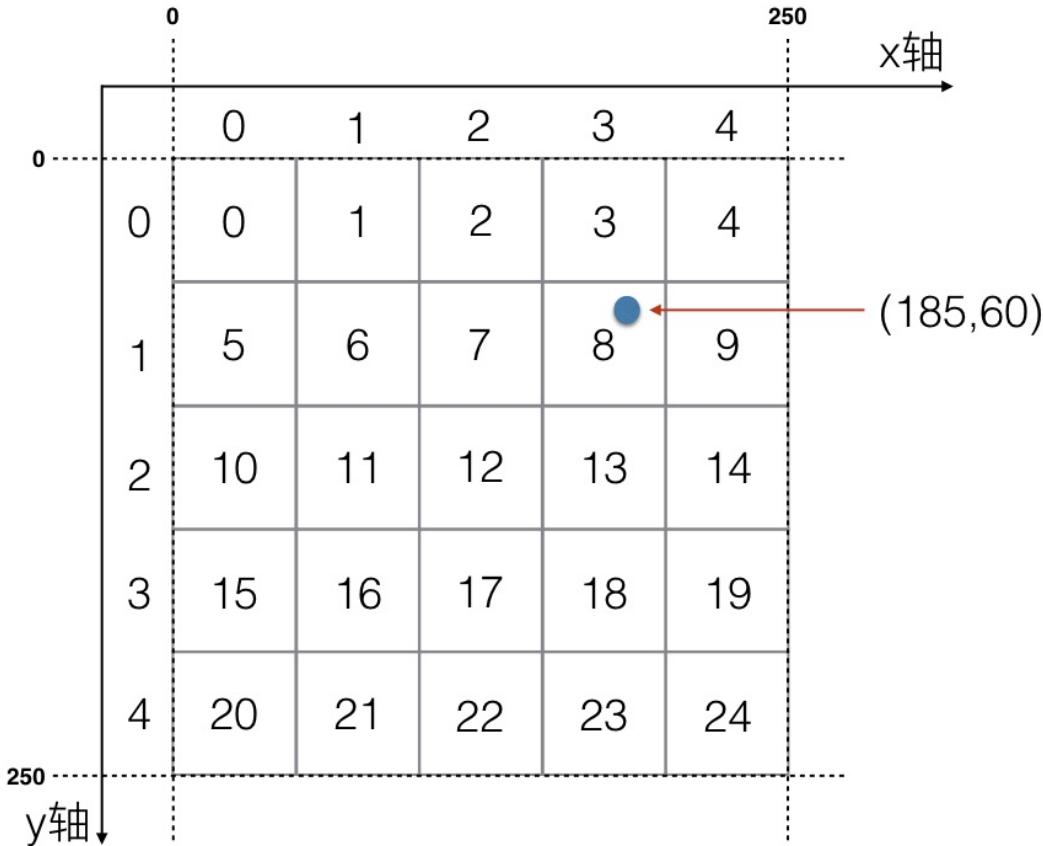
    //判断当前的idy上边是否还有格子
    if idy > 0 {
        grids = append(grids, m.grids[v-m.CntsX])
    }
    //判断当前的idy下边是否还有格子
    if idy < m.CntsY - 1 {
        grids = append(grids, m.grids[v+m.CntsX])
    }
}

return
}
```

#### B)根据坐标求出九宫格

还有一种情况是玩家只知道自己的坐标，那么如何确定玩家AOI九宫格的区域都有哪些玩家呢，那就需要设计一个根据坐标求出周边九宫格中玩家的接口。

### 3.4 求出九宫格



我们首先应该根据坐标得到所属的格子ID，然后再根据格子ID获取九宫格信息就可以了。

mmo\_game/core/aoi.go

### 3.4 求出九宫格

```
//通过横纵坐标获取对应的格子ID
func (m *AOIManager) GetGIDByPos(x, y float32) int {
    gx := (int(x) - m.MinX) / m.gridWidth()
    gy := (int(x) - m.MinY) / m.gridLength()

    return gy * m.CntsX + gx
}

//通过横纵坐标得到周边九宫格内的全部PlayerIDs
func (m *AOIManager) GetPIDsByPos(x, y float32) (playerIDs []int) {
    //根据横纵坐标得到当前坐标属于哪个格子ID
    gID := m.GetGIDByPos(x, y)

    //根据格子ID得到周边九宫格的信息
    grids := m.GetSurroundGridsByGid(gID)
    for _, v := range grids {
        playerIDs = append(playerIDs, v.GetPlayerIDs()...)
        fmt.Printf("====> grid ID : %d, pids : %v ====%", v.GID,
v.GetPlayerIDs())
    }

    return
}
```

## 3.5 AOI格子添加删除操作

mmo\_game/core/aoi.go

```
//通过GID获取当前格子的全部playerID
func (m *AOIManager) GetPidsByGid(gID int) (playerIDs []int) {
    playerIDs = m.grids[gID].GetPlayerIDs()
    return
}

//移除一个格子中的PlayerID
func (m *AOIManager) RemovePidFromGrid(pID, gID int) {
    m.grids[gID].Remove(pID)
}

//添加一个PlayerID到一个格子中
func (m *AOIManager) AddPidToGrid(pID, gID int) {
    m.grids[gID].Add(pID)
}

//通过横纵坐标添加一个Player到一个格子中
func (m *AOIManager) AddToGridByPos(pID int, x, y float32) {
    gID := m.GetGidByPos(x, y)
    grid := m.grids[gID]
    grid.Add(pID)
}

//通过横纵坐标把一个Player从对应的格子中删除
func (m *AOIManager) RemoveFromGridByPos(pID int, x, y float32) {
    gID := m.GetGidByPos(x, y)
    grid := m.grids[gID]
    grid.Remove(pID)
}
```

## 3.6 AOI模块单元测试

```

package core

import (
    "fmt"
    "testing"
)

func TestNewAOIManager(t *testing.T) {
    aoiMgr := NewAOIManager(100, 300, 4, 200, 450, 5)
    fmt.Println(aoiMgr)
}

func TestAOIManagerSuroundGridsByGid(t *testing.T) {
    aoiMgr := NewAOIManager(0, 250, 5, 0, 250, 5)

    for k, _ := range aoiMgr.grids {
        //得到当前格子周边的九宫格
        grids := aoiMgr.GetSurroundGridsByGid(k)
        //得到九宫格所有的IDs
        fmt.Println("gid : ", k, " grids len = ", len(grids))
        gIDs := make([]int, 0, len(grids))
        for _, grid := range grids {
            gIDs = append(gIDs, grid.GID)
        }
        fmt.Printf("grid ID: %d, surrounding grid IDs are %v\n",
            k, gIDs)
    }
}

```

结果

```

AOIManagr:
minX:100, maxX:300, cntsX:4, minY:200, maxY:450, cntsY:5
Grids in AOI Manager:
Grid id: 1, minX:150, maxX:200, minY:200, maxY:250, playerIDs:ma

```

```
p[]  
Grid id: 5, minX:150, maxX:200, minY:250, maxY:300, playerIDs:ma  
p[]  
Grid id: 6, minX:200, maxX:250, minY:250, maxY:300, playerIDs:ma  
p[]  
Grid id: 12, minX:100, maxX:150, minY:350, maxY:400, playerIDs:m  
ap[]  
Grid id: 19, minX:250, maxX:300, minY:400, maxY:450, playerIDs:m  
ap[]  
Grid id: 7, minX:250, maxX:300, minY:250, maxY:300, playerIDs:ma  
p[]  
Grid id: 8, minX:100, maxX:150, minY:300, maxY:350, playerIDs:ma  
p[]  
Grid id: 10, minX:200, maxX:250, minY:300, maxY:350, playerIDs:m  
ap[]  
Grid id: 11, minX:250, maxX:300, minY:300, maxY:350, playerIDs:m  
ap[]  
Grid id: 15, minX:250, maxX:300, minY:350, maxY:400, playerIDs:m  
ap[]  
Grid id: 18, minX:200, maxX:250, minY:400, maxY:450, playerIDs:m  
ap[]  
Grid id: 0, minX:100, maxX:150, minY:200, maxY:250, playerIDs:ma  
p[]  
Grid id: 3, minX:250, maxX:300, minY:200, maxY:250, playerIDs:ma  
p[]  
Grid id: 4, minX:100, maxX:150, minY:250, maxY:300, playerIDs:ma  
p[]  
Grid id: 14, minX:200, maxX:250, minY:350, maxY:400, playerIDs:m  
ap[]  
Grid id: 16, minX:100, maxX:150, minY:400, maxY:450, playerIDs:m  
ap[]  
Grid id: 2, minX:200, maxX:250, minY:200, maxY:250, playerIDs:ma  
p[]  
Grid id: 9, minX:150, maxX:200, minY:300, maxY:350, playerIDs:ma  
p[]  
Grid id: 13, minX:150, maxX:200, minY:350, maxY:400, playerIDs:m  
ap[]  
Grid id: 17, minX:150, maxX:200, minY:400, maxY:450, playerIDs:m  
ap[]
```

```
gid : 3 grids len = 6
grid ID: 3, surrounding grid IDs are [3 2 4 8 7 9]
gid : 5 grids len = 6
grid ID: 5, surrounding grid IDs are [5 6 0 10 1 11]
gid : 6 grids len = 9
grid ID: 6, surrounding grid IDs are [6 5 7 1 11 0 10 2 12]
gid : 11 grids len = 9
grid ID: 11, surrounding grid IDs are [11 10 12 6 16 5 15 7 17]
gid : 18 grids len = 9
grid ID: 18, surrounding grid IDs are [18 17 19 13 23 12 22 14 2
4]
gid : 2 grids len = 6
grid ID: 2, surrounding grid IDs are [2 1 3 7 6 8]
gid : 4 grids len = 4
grid ID: 4, surrounding grid IDs are [4 3 9 8]
gid : 7 grids len = 9
grid ID: 7, surrounding grid IDs are [7 6 8 2 12 1 11 3 13]
gid : 8 grids len = 9
grid ID: 8, surrounding grid IDs are [8 7 9 3 13 2 12 4 14]
gid : 19 grids len = 6
grid ID: 19, surrounding grid IDs are [19 18 14 24 13 23]
gid : 22 grids len = 6
grid ID: 22, surrounding grid IDs are [22 21 23 17 16 18]
gid : 0 grids len = 4
grid ID: 0, surrounding grid IDs are [0 1 5 6]
gid : 1 grids len = 6
grid ID: 1, surrounding grid IDs are [1 0 2 6 5 7]
gid : 13 grids len = 9
grid ID: 13, surrounding grid IDs are [13 12 14 8 18 7 17 9 19]
gid : 14 grids len = 6
grid ID: 14, surrounding grid IDs are [14 13 9 19 8 18]
gid : 16 grids len = 9
grid ID: 16, surrounding grid IDs are [16 15 17 11 21 10 20 12 2
2]
gid : 17 grids len = 9
grid ID: 17, surrounding grid IDs are [17 16 18 12 22 11 21 13 2
3]
gid : 23 grids len = 6
grid ID: 23, surrounding grid IDs are [23 22 24 18 17 19]
gid : 24 grids len = 4
```

```
grid ID: 24, surrounding grid IDs are [24 23 19 18]
gid : 9 grids len = 6
grid ID: 9, surrounding grid IDs are [9 8 4 14 3 13]
gid : 10 grids len = 6
grid ID: 10, surrounding grid IDs are [10 11 5 15 6 16]
gid : 12 grids len = 9
grid ID: 12, surrounding grid IDs are [12 11 13 7 17 6 16 8 18]
gid : 15 grids len = 6
grid ID: 15, surrounding grid IDs are [15 16 10 20 11 21]
gid : 20 grids len = 4
grid ID: 20, surrounding grid IDs are [20 21 15 16]
gid : 21 grids len = 6
grid ID: 21, surrounding grid IDs are [21 20 22 16 15 17]
PASS
ok      zinx/zinx_app_demo/mmo_game/core      0.002s
```

我们可以用我们的AOI地图验证一下，是一致的。



## 4.1 简介

**Google Protocol Buffer**(简称 Protobuf)是google旗下的一款轻便高效的结构化数据存储格式，平台无关、语言无关、可扩展，可用于通讯协议和数据存储等领域。所以很适合用做数据存储和作为不同应用，不同语言之间相互通信的数据交换格式，只要实现相同的协议格式即同一 proto文件被编译成不同的语言版本，加入到各自的工程中去。这样不同语言就可以解析其他语言通过 protobuf序列化的数据。目前官网提供了 C++,Python,JAVA,GO等语言的支持。google在2008年7月7号将其作为开源项目对外公布。

**tips :**

1. 啥叫平台无关？Linux、mac和Windows都可以用，32位系统，64位系统通吃
2. 啥叫语言无关？C++、Java、Python、Golang语言编写的程序都可以用，而且可以相互通信
3. 那啥叫可扩展呢？就是这个数据格式可以方便的增删一部分字段啦~
4. 最后，啥叫序列化啊？解释得通俗点儿就是把复杂的结构体数据按照一定的规则编码成一个字节切片

## 4.2 数据交换格式

### A) 常用的数据交换格式有三种：

1. `json`：一般的web项目中，最流行的主要还是 `json`。因为浏览器对于 `json` 数据支持非常好，有很多内建的函数支持。
2. `xml`：在 `webservice` 中应用最为广泛，但是相比于 `json`，它的数据更加冗余，因为需要成对的闭合标签。`json` 使用了键值对的方式，不仅压缩了一定的数据空间，同时也具有可读性。
3. `protobuf`：是后起之秀，是谷歌开源的一种数据格式，适合高性能，对响应速度有要求的数据传输场景。因为 `profobuf` 是二进制数据格式，需要编码和解码。数据本身不具有可读性。因此只能反序列化之后得到真正可读的数据。

### B) `protobuf` 的优势与劣势

优势：

- 1：序列化后体积相比Json和XML很小，适合网络传输
- 2：支持跨平台多语言
- 3：消息格式升级和兼容性还不错
- 4：序列化反序列化速度很快，快于Json的处理速度

劣势：

- 1：应用不够广(相比xml和json)
- 2：二进制格式导致可读性差
- 3：缺乏自描述

## 4.3 protobuf环境安装

### A) protobuf 编译工具安装

1、下载 protoBuf：

```
cd $GOPATH/src/  
git clone https://github.com/protocolbuffers/protobuf.git
```

2、或者直接将压缩包拖入后解压

```
unzip protobuf.zip
```

3、安装依赖库

```
sudo apt-get install autoconf automake libtool curl make g++  
unzip libffi-dev -y
```

4、进入目录

```
cd protobuf/
```

5、自动生成configure配置文件：

```
./autogen.sh
```

6、配置环境：

```
./configure
```

7、编译源代码(时间比较长)：

```
make
```

### 8、安装

```
sudo make install
```

### 9、刷新共享库（很重要的一步啊）

```
sudo ldconfig
```

### 10、成功后需要使用命令测试

```
protoc -h
```

## B) protobuf 的 go 语言插件安装

由于protobuf并没直接支持go语言需要我们手动安装相关插件

### 1. 获取 proto包(Go语言的proto API接口)

```
go get -v -u github.com/golang/protobuf/proto  
go get -v -u github.com/golang/protobuf/protoc-gen-go
```

### 1. 编译

```
cd $GOPATH/src/github.com/golang/protobuf/protoc-gen-go/  
go build
```

### 1. 将生成的 protoc-gen-go可执行文件，放在/bin目录下

```
sudo cp protoc-gen-go /bin/
```

## 4.4 protobuf语法

protobuf通常会把用户定义的结构体类型叫做一个消息，这里我们遵循惯例，统一称为消息。protobuf消息的定义（或者称为描述）通常都写在一个以.proto结尾的文件中。

### A) 一个简单的例子

```

syntax = "proto3";                                //指定版本信息，不指定会
报错

package pb;                                       //后期生成go文件的包名

//message为关键字，作用为定义一种消息类型
message Person {
    string     name = 1;                           //姓名
    int32      age = 2;                            //年龄
    repeated string emails = 3;                   //电子邮件（repeated表示字
段允许重复）
    repeated PhoneNumber phones = 4;             //手机号
}

//enum为关键字，作用为定义一种枚举类型
enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
}

//message为关键字，作用为定义一种消息类型可以被另外的消息类型嵌套使用
message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
}

```

### B) 消息格式说明

消息由字段组成，每个消息的字段格式为：

(字段修饰符 +) 数据类型 + 字段名称 + 唯一的编号标签值；

- 字段名称：蛇形或者驼峰
- 唯一的编号标签：代表每个字段的一个唯一的编号标签，在同一个消息里不可以重复。这些编号标签用与在消息二进制格式中标识你的字段，并且消息一旦定义就不能更改。需要说明的是标签在1到15范围的采用一个字节进行编码，所以通常将标签1到15用于频繁发生的消息字段。编号标签大小的范围是1到229
- 注释格式：向.proto文件添加注释，可以使用C/C++/java/Go风格的双斜杠  
(//) 语法格式

### C) 数据类型

.proto Type	Go Type	Notes
double	float64	64位浮点数
float	float32	32位浮点数
int32	int32	使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用sint64替代
uint32	uint32	使用变长编码
uint64	uint64	使用变长编码
sint32	int32	使用变长编码，这些编码在负值时比int32高效的多
sint64	int64	使用变长编码，有符号的整型值。编码时比通常的int64高效。
fixed32	uint32	总是4个字节，如果数值总是比228大的话，这个类型会比uint32高效。
fixed64	uint64	总是8个字节，如果数值总是比256大的话，这个类型会比uint64高效。
sfixed32	int32	总是4个字节
sfixed32	int32	总是4个字节
sfixed64	int64	总是8个字节
bool	bool	
string	string	一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。
bytes	[]byte	可能包含任意顺序的字节数据。

更多详情请看：<https://developers.google.com/protocol-buffers/docs/encoding>

#### D) 默认缺省值

当一个消息被解析的时候，如果被编码的信息不包含一个特定的元素，被解析的对象锁对应的域被设置位一个默认值，对于不同类型指定如下：

- 对于strings，默认是一个空string
- 对于bytes，默认是一个空的bytes
- 对于bools，默认是false
- 对于数值类型，默认是0



## 4.5 编译protobuf

通过如下方式调用protocol编译器，把 .proto 文件编译成代码：

```
protoc --proto_path=IMPORT_PATH --go_out=DST_DIR path/to/file.proto
```

其中：

1. `--proto_path`，指定了 .proto 文件导包时的路径，可以有多个，如果忽略则默认当前目录。
2. `--go_out`，指定了生成的go语言代码文件放入的文件夹
3. 允许使用 `protoc --go_out=./ *.proto` 的方式一次性编译多个 .proto 文件
4. 编译时，protobuf 编译器会把 .proto 文件编译成 .pd.go 文件

## 4.6 利用protobuf生成的类来编码

```
package main

import (
    "fmt"
    "github.com/golang/protobuf/proto"
    "protocolbuffer_excise/pb"
)

func main() {
    person := &pb.Person{
        Name:      "Aceld",
        Age:       16,
        Emails:   []string{"https://legacy.gitbook.com/@aceld", "h
https://github.com/aceld"},

        Phones: []*pb.PhoneNumber{
            &pb.PhoneNumber{
                Number: "13113111311",
                Type:   pb.PhoneType_MOBILE,
            },
            &pb.PhoneNumber{
                Number: "14141444144",
                Type:   pb.PhoneType_HOME,
            },
            &pb.PhoneNumber{
                Number: "19191919191",
                Type:   pb.PhoneType_WORK,
            },
        },
    }

    data, err := proto.Marshal(person)
    if err != nil {
        fmt.Println("marshal err:", err)
    }

    newdata := &pb.Person{}
```

```
err = proto.Unmarshal(data, newdata)
if err != nil {
    fmt.Println("unmarshal err:", err)
}

fmt.Println(newdata)

}
```

## 五、MMO游戏的Proto3协议

MsgID	Client	Server	描述
1	-	SyncPid	同步玩家本次登录的ID(用来标识玩家)
2	Talk	-	世界聊天
3	Position	-	移动
200	-	BroadCast	广播消息(Tp 1 世界聊天 2 坐标(出生点同步) 3 动作 4 移动之后坐标信息更新)
201	-	SyncPid	广播消息 掉线/aoi消失在视野
202	-	SyncPlayers	同步周围的人位置信息(包括自己)

### MsgID :1

SyncPid :

- 同步玩家本次登录的ID(用来标识玩家), 玩家登陆之后, 由Server端主动生成玩家ID发送给客户端
- 发起者: Server
- Pid: 玩家ID

```
message SyncPid{
    int32 Pid=1;
}
```

### MsgID :2

Talk :

- 同步玩家本次登录的ID(用来标识玩家), 玩家登陆之后, 由Server端主动生成玩家ID发送给客户端

- 发起者： Client
- Content: 聊天信息

```
message Talk{  
    string Content=1;  
}
```

### MsgID :3

MovePackege :

- 移动的坐标数据
- 发起者： Client
- P: Position类型，地图的左边点

```
message Position{  
    float X=1;  
    float Y=2;  
    float Z=3;  
    float V=4;  
}
```

### MsgID :200

BroadCast :

- 广播消息
- 发起者： Server
- Tp: 1 世界聊天, 2 坐标, 3 动作, 4 移动之后坐标信息更新
- Pid: 玩家ID

```
message BroadCast{
    int32 Pid=1;
    int32 Tp=2;
    oneof Data {
        string Content=3;
        Position P=4;
        int32 ActionData=5;
    }
}
```

## MsgID :201

SyncPid :

- 广播消息 掉线/aoi消失在视野
- 发起者： Server
- Pid: 玩家ID

```
message SyncPid{
    int32 Pid=1;
}
```

## MsgID :202

- 同步周围的人位置信息(包括自己)
- 发起者： Server
- ps: Player 集合,需要同步的玩家

```
message SyncPlayers{
    repeated Player ps=1;
}

message Player{
    int32 Pid=1;
    Position P=2;
}
```

## 六、构建项目与用户上线

现在，我们应该基于Zinx框架来构建一个MMO的游戏服务器应用程序的项目了。

我们这里创建一个项目 `mmo_game` ,在项目内分别创建几个文件夹 `api` , `conf` , `core` , `game_client` , `pb` 等

## 6.1 构建项目

`api` :主要是注册一些mmo业务的一些Router处理业务。

`conf` :存放mmo\_game的一些配置文件,比如"zinx.json"。

`core` :存放一些核心算法,或者游戏控制等模块。

`game_client` :存放游戏客户端。

`pb` :存放一些protobuf的协议文件和go文件。

1、我们在 `mmo_game` 下,创建一个 `server.go` 作为我们main包,主要作为服务器程序的主入口。

mmo\_game/server.go

```
package main

import (
    "zinx/znet"
)

func main() {
    //创建服务器句柄
    s := znet.NewServer()

    //启动服务
    s.Serve()
}
```

2、在 `conf` 文件添加 `zinx.conf`

mmo\_game/conf/zinx.conf

## 6.1 构建项目

```
{  
    "Name": "Zinx Game",  
    "Host": "0.0.0.0",  
    "TcpPort": 8999,  
    "MaxConn": 3000,  
    "WorkerPoolSize": 10  
}
```

3、在 pb 下创建msg.proto文件和build.sh编译指令脚本

mmo\_game/pb/msg.proto

```
syntax="proto3";          //Proto协议  
package pb;                //当前包名  
option csharp_namespace="Pb"; //给C#提供的选项
```

mmo\_game/pb/build.sh

```
#!/bin/bash  
protoc --go_out=. *.proto
```

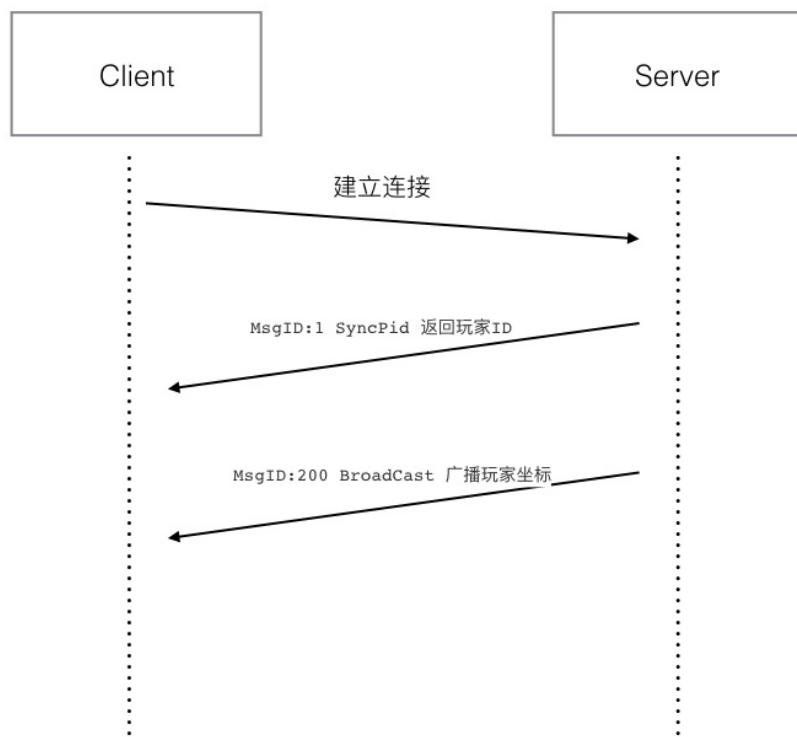
当前我们的项目路径应该结构如下：

```
.  
└── mmo_game  
    ├── api  
    ├── conf  
    │   └── zinx.json  
    ├── core  
    │   ├── aoi.go  
    │   ├── aoi_test.go  
    │   ├── grid.go  
    ├── game_client  
    │   └── client.exe  
    ├── pb  
    │   ├── build.sh  
    │   └── msg.proto  
    ├── README.md  
    └── server.go
```

## 6.2 用户上线流程

好了，那么我们第一次就要尝试将客户端的MMO游戏和移动端做一次上线测试了。

我们第一个测试用户上线的流程比较简单：



### A) 定义proto协议

我们从图中可以看到，上线的业务会涉及到MsgID:1 和 MsgID:200 两个消息，根据我们上一个章节的介绍，我们需要在msg.proto中定义出两个proto类型，并且声称对应的go代码.

mmo\_game/pb/msg.proto

```

syntax="proto3"; //Proto协议
package pb; //当前包名
option csharp_namespace="Pb"; //给C#提供的选项

//同步客户端玩家ID
message SyncPid{
    int32 Pid=1;
}

//玩家位置
message Position{
    float X=1;
    float Y=2;
    float Z=3;
    float V=4;
}

//玩家广播数据
message BroadCast{
    int32 Pid=1;
    int32 Tp=2;
    oneof Data {
        string Content=3;
        Position P=4;
        int32 ActionData=5;
    }
}

```

执行build.sh生成对应的 msg.pb.go 代码.

## B) 创建Player模块

- 首先我们先创建一个Player玩家模块

mmo\_game/core/player.go

```

//玩家对象
type Player struct {
    Pid int32          //玩家ID
    Conn ziface.IConnection //当前玩家的连接
    X    float32        //平面X坐标
    Y    float32        //高度
    Z    float32        //平面y坐标 (注意不是Y)
    V    float32        //旋转0-360度
}

/*
    Player ID 生成器
*/
var PidGen int32 = 1      //用来生成玩家ID的计数器
var IdLock sync.Mutex     //保护PidGen的互斥机制

//创建一个玩家对象
func NewPlayer(conn ziface.IConnection) *Player {
    //生成一个PID
    IdLock.Lock()
    id := PidGen
    PidGen ++
    IdLock.Unlock()

    p := &Player{
        Pid : id,
        Conn:conn,
        X:float32(160 + rand.Intn(10)), //随机在160坐标点 基于X轴偏移
        若干坐标
        Y:0, //高度为0
        Z:float32(134 + rand.Intn(17)), //随机在134坐标点 基于Y轴偏
        移若干坐标
        V:0, //角度为0，尚未实现
    }

    return p
}

```

Plyaer类中有当前玩家的ID，和当前玩家与客户端绑定的conn，还有就是地图的坐标信，`NewPlayer()` 提供初始化玩家方法。

- 由于 `Player` 经常需要和客户端发送消息，那么我们可以给 `Player` 提供一个 `SendMsg()` 方法，供客户端发送消息

mmo\_game/core/player.go

```
/*
发送消息给客户端,
主要是将pb的protobuf数据序列化之后发送
*/
func (p *Player) SendMsg(msgId uint32, data proto.Message) {
    fmt.Printf("before Marshal data = %v\n", data)
    //将proto Message结构体序列化
    msg, err := proto.Marshal(data)
    if err != nil {
        fmt.Println("marshal msg err: ", err)
        return
    }
    fmt.Printf("after Marshal data = %v\n", msg)

    if p.Conn == nil {
        fmt.Println("connection in player is nil")
        return
    }

    //调用Zinx框架的SendMsg发包
    if err := p.Conn.SendMsg(msgId, msg); err != nil {
        fmt.Println("Player SendMsg error !")
        return
    }

    return
}
```

这里要注意的是，`SendMsg()` 是将发送的数据，通过proto序列化，然后再调用 `Zinx` 框架的`SendMsg`方法发送给对方客户端.

## C) 实现上线业务

我们先在 Server 的 main 入口，给链接绑定一个创建之后的 hook 方法，因为上线的时候是服务器自动回复客户端玩家 ID 和坐标，那么需要我们在连接创建完毕之后，自动触发，正好我们可以利用 Zinx 框架的 SetOnConnStart 方法。

| mmo\_game/server.go

```

package main

import (
    "fmt"
    "zinx/ziface"
    "zinx/zinx_app_demo/mmo_game/core"
    "zinx/znet"
)

//当客户端建立连接的时候的hook函数
func OnConnecionAdd(conn ziface.IConnection) {
    //创建一个玩家
    player := core.NewPlayer(conn)
    //同步当前的PlayerID给客户端，走MsgID:1 消息
    player.SyncPid()
    //同步当前玩家的初始化坐标信息给客户端，走MsgID:200消息
    player.BroadCastStartPosition()

    fmt.Println("===== Player pidId = ", player.Pid, " arrived =====")
}

func main() {
    //创建服务器句柄
    s := znet.NewServer()

    //注册客户端连接建立和丢失函数
    s.SetOnConnStart(OnConnecionAdd)

    //启动服务
    s.Serve()
}

```

根据我们之前的流程分析，那么在客户端建立连接过来之后，Server要自动的回复给客户端一个玩家ID，同时也要讲当前玩家的坐标发送给客户端。所以我们这里面给Player定制了两个方

法 `Player.SyncPid()` 和 `Player.BroadCastStartPosition()`

## 6.2 用户上线流程

SyncPid() 则为发送 MsgID:1 的消息，将当前上线的用户 ID 发送给客户端

mmo\_game/core/player.go

```
// 告知客户端 pid，同步已经生成的玩家 ID 给客户端
func (p *Player) SyncPid() {
    // 组建 MsgID:0 proto 数据
    data := &pb.SyncPid{
        Pid: p.Pid,
    }

    // 发送数据给客户端
    p.SendMsg(1, data)
}
```

BroadCastStartPosition() 则为发送 MsgID:200 的广播位置消息，虽然现在没有其他用户，不是广播，但是当前玩家自己的坐标也是要告知玩家的。

mmo\_game/core/player.go

```
// 广播玩家自己的出生地点
func (p *Player) BroadCastStartPosition() {

    msg := &pb.BroadCast{
        Pid: p.Pid,
        Tp: 2, // TP2 代表广播坐标
        Data: &pb.BroadCast_P{
            &pb.Position{
                X: p.X,
                Y: p.Y,
                Z: p.Z,
                V: p.V,
            },
        },
    }

    p.SendMsg(200, msg)
}
```

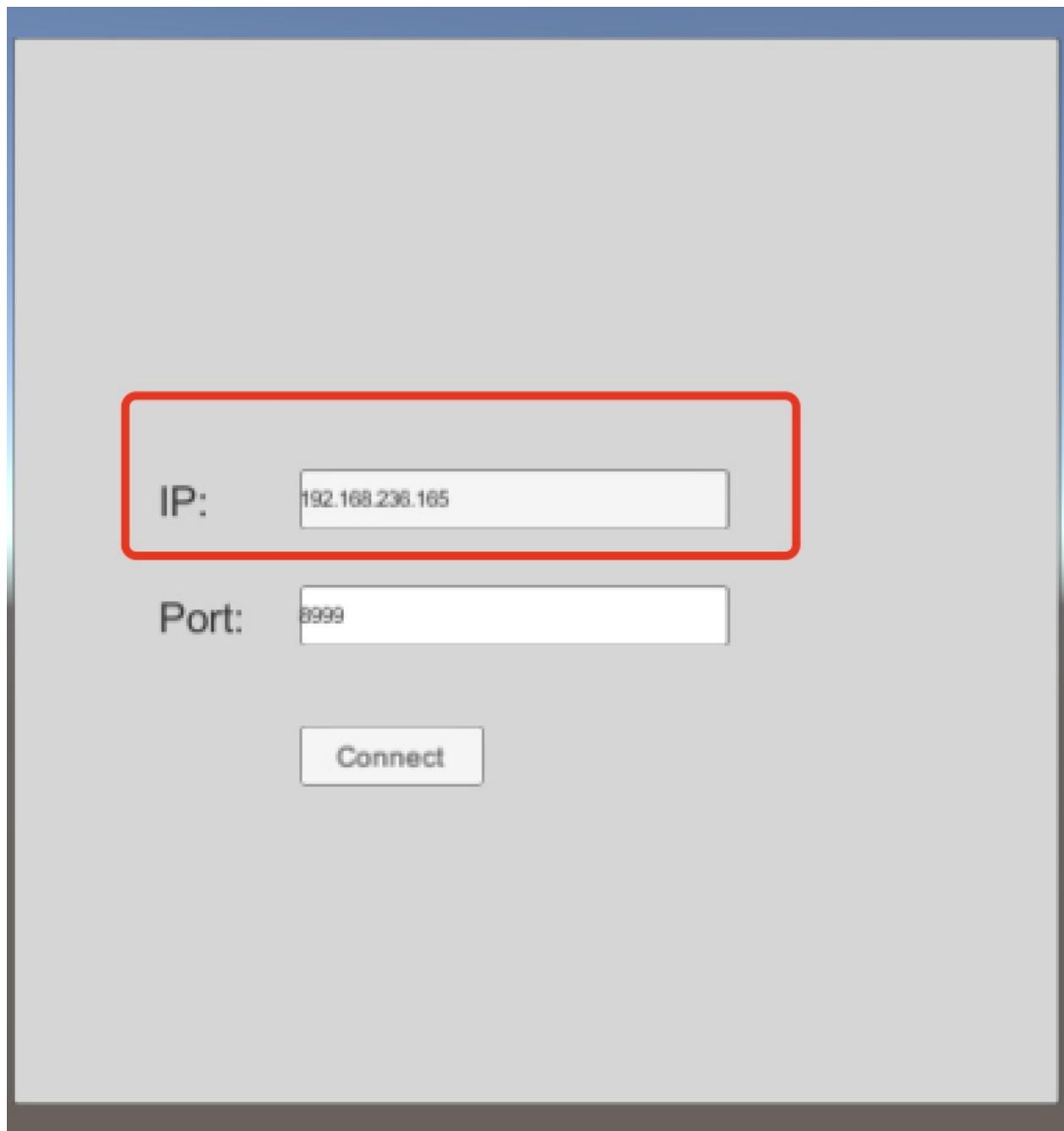
## D) 测试用户上线业务

```
$cd mmo_game/  
$go run server.go
```

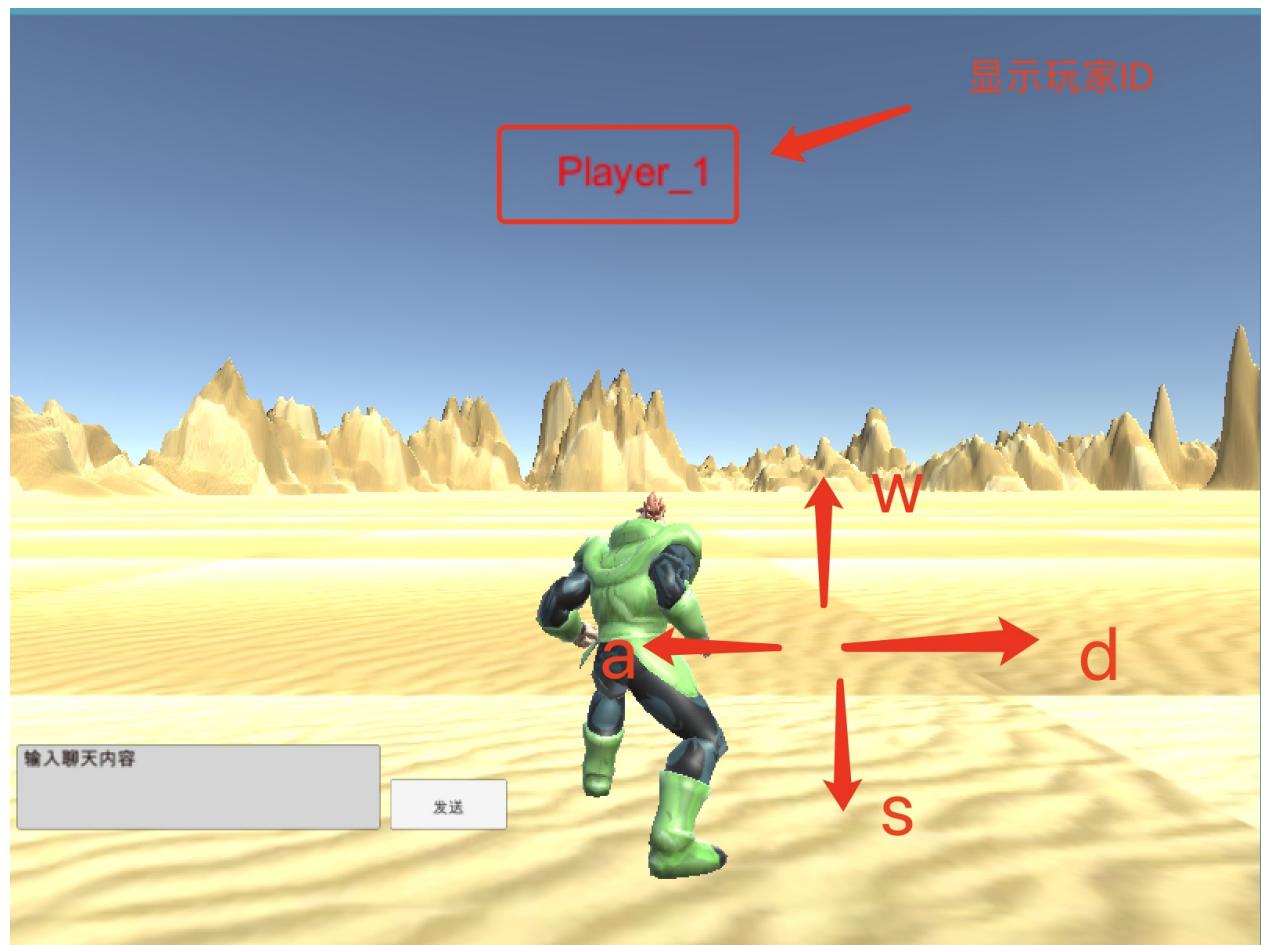
启动服务端程序。

然后再windows终端打开 `client.exe`

注意，要确保windows和启动服务器的Linux端要能够ping通，为了方便测试，建议将Linux的防火墙设置为关闭状态，或者要确保服务器的端口是开放的，以免耽误调试



在此处输入服务器的IP地址，和服务器 `zinx.json` 配置的端口号。然后点击 Connect。



如果游戏界面顺利进入，并且已经显示为 Player\_1 玩家ID，表示登录成功，并且我们在服务端也可以看到一些调试信息。操作WASD也可以进行玩家移动。如果没有显示玩家ID或者为 TextView 则为登录失败，我们需要再针对协议的匹配进行调试。

## 七、世界聊天系统实现

## 7.1 世界管理模块

现在需要一个管理当前世界所有玩家的一个管理器，管理器应该拥有全部的当前在线玩家信息和当前世界的AOI划分规则。方便玩家与玩家之间进行聊天，同步位置等功能。

首先，在创建一个 `world_manager.go` 文件作为世界管理器模块。

`mmo_game/core/world_manager.go`

```
package core

import (
    "sync"
)

/*
    当前游戏世界的总管理模块
*/
type WorldManager struct {
    AoiMgr *AOIManager //当前世界地图的AOI规划管理器
    Players map[int32]*Player //当前在线的玩家集合
    pLock sync.RWMutex //保护Players的互斥读写机制
}

//提供一个对外的世界管理模块句柄
var WorldMgrObj *WorldManager

//提供WorldManager 初始化方法
func init() {
    WorldMgrObj = &WorldManager{
        Players: make(map[int32]*Player),
        AoiMgr: NewAOIManager(AOI_MIN_X, AOI_MAX_X, AOI_CNTS_X,
            AOI_MIN_Y, AOI_MAX_Y, AOI_CNTS_Y),
    }
}

//提供添加一个玩家的功能，将玩家添加进玩家信息表Players
func (wm *WorldManager) AddPlayer(player *Player) {
```

```

//将player添加到 世界管理器中
wm.pLock.Lock()
wm.Players[player.Pid] = player
wm.pLock.Unlock()

//将player 添加到AOI网络规划中
wm.AoiMgr.AddToGridByPos(int(player.Pid), player.X, player.Z
)
}

//从玩家信息表中移除一个玩家
func (wm *WorldManager) RemovePlayerByPid(pid int32) {
    wm.pLock.Lock()
    delete(wm.Players, pid)
    wm.pLock.Unlock()
}

//通过玩家ID 获取对应玩家信息
func (wm *WorldManager) GetPlayerByPid(pid int32) *Player {
    wm.pLock.RLock()
    defer wm.pLock.RUnlock()

    return wm.Players[pid]
}

//获取所有玩家的信息
func (wm *WorldManager) GetAllPlayers() []*Player {
    wm.pLock.RLock()
    defer wm.pLock.RUnlock()

    //创建返回的player集合切片
    players := make([]*Player, 0)

    //添加切片
    for _, v := range wm.Players {
        players = append(players, v)
    }

    //返回
    return players
}

```

```
}
```

该模块主要是将AOI和玩家做了一层统一管理，起到协调其他模块的中间功能。其中有一个全局变量 `WorldMgrObj` 是对外开放的管理模块句柄。供其他模块使用。

现在我们应该在玩家上线的时候，也将玩家添加到 `WorldMgrObj` 中。

mmo\_game/server.go

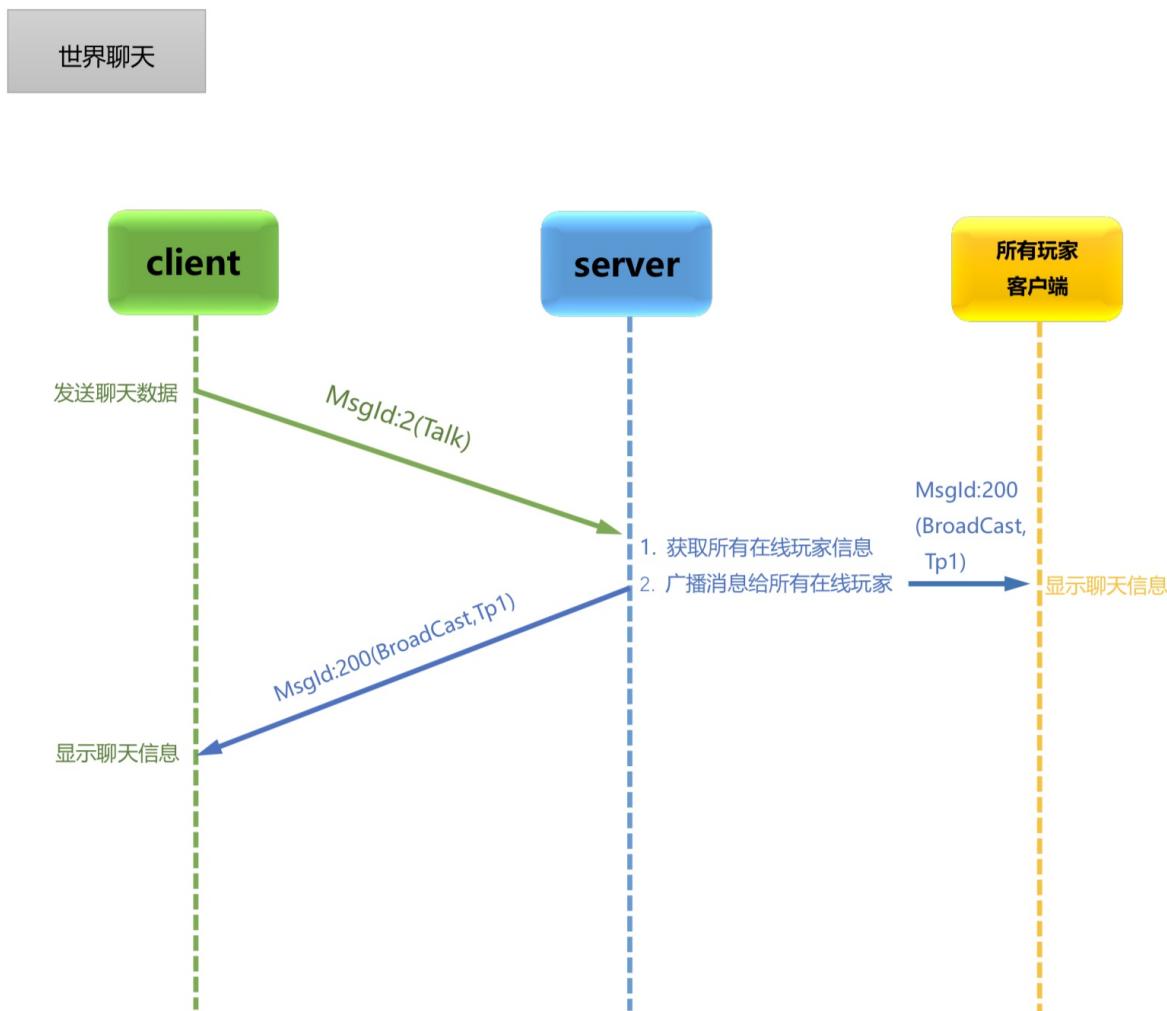
```
//当客户端建立连接的时候的hook函数
func OnConnnectionAdd(conn ziface.IConnection) {
    //创建一个玩家
    player := core.NewPlayer(conn)
    //同步当前的PlayerID给客户端，走MsgID:1 消息
    player.SyncPid()
    //同步当前玩家的初始化坐标信息给客户端，走MsgID:200消息
    player.BroadCastStartPosition()

    //=====将当前新上线玩家添加到worldManager中
    core.WorldMgrObj.AddPlayer(player)
    //=====

    fmt.Println("=====> Player pidId = ", player.Pid, " arrived
=====")
}
```

## 7.2 世界聊天系统实现

接下来，我们来做一个玩家和玩家之间的世界聊天广播功能。



### A) proto3协议定义

这里涉及到了*MsgId:2*的指令，还有对应的*Talk*的proto协议。

**MsgID :2**

**Talk :**

- 同步玩家本次登录的ID(用来标识玩家), 玩家登陆之后, 由Server端主动生成玩家ID发送给客户端
- 发起者： Client

- Content: 聊天信息

```
message Talk{  
    string Content=1;  
}
```

所以我们应该先修改proto文件

mmo\_game/pb/msg.proto

```

syntax="proto3";                                //Proto协议
package pb;                                     //当前包名
option csharp_namespace="Pb";                   //给C#提供的选项

//同步客户端玩家ID
message SyncPid{
    int32 Pid=1;
}

//玩家位置
message Position{
    float X=1;
    float Y=2;
    float Z=3;
    float V=4;
}

//玩家广播数据
message BroadCast{
    int32 Pid=1;
    int32 Tp=2;                               //1-世界聊天 2-玩家位置
    oneof Data {
        string Content=3;                  //聊天的信息
        Position P=4;                     //广播用户的位置
        int32 ActionData=5;
    }
}

//=====
//玩家聊天数据
message Talk{
    string Content=1;                         //聊天内容
}
//=====
```

执行build.sh 生成新的 msg.proto.go 文件。

## B) 聊天业务API建立

接下来，我们创建一个api文件

```
mmo_game/api/world_chat.go
```

```

package api

import (
    "fmt"
    "github.com/golang/protobuf/proto"
    "zinx/ziface"
    "zinx/zinx_app_demo/mmo_game/core"
    "zinx/zinx_app_demo/mmo_game/pb"
    "zinx/znet"
)

//世界聊天 路由业务
type WorldChatApi struct {
    znet.BaseRouter
}

func (*WorldChatApi) Handle(request ziface IRequest) {
    //1. 将客户端传来的proto协议解码
    msg := &pb.Talk{}
    err := proto.Unmarshal(request.GetData(), msg)
    if err != nil {
        fmt.Println("Talk Unmarshal error ", err)
        return
    }

    //2. 得知当前的消息是从哪个玩家传递来的,从连接属性pid中获取
    pid, err := request.GetConnection().GetProperty("pid")
    if err != nil {
        fmt.Println("GetProperty pid error", err)
        request.GetConnection().Stop()
        return
    }

    //3. 根据pid得到player对象
    player := core.WorldMgrObj.GetPlayerByPid(pid.(int32))

    //4. 让player对象发起聊天广播请求
    player.Talk(msg.Content)
}

```

这里实际上对于msgID : 2的路由业务函数的实现。其中有个小细节需要注意一下。第2步，根据链接conn得到当前玩家的pid，应该是我们之前在玩家上线的时候，将pid和conn做一个属性绑定，如下：

mmo\_game/server.go

```
//当客户端建立连接的时候的hook函数
func OnConnecionAdd(conn ziface.IConnection) {
    //创建一个玩家
    player := core.NewPlayer(conn)
    //同步当前的PlayerID给客户端，走MsgID:1 消息
    player.SyncPid()
    //同步当前玩家的初始化坐标信息给客户端，走MsgID:200消息
    player.BroadCastStartPosition()
    //将当前新上线玩家添加到worldManager中
    core.WorldMgrObj.AddPlayer(player)
    //=====将该连接绑定属性Pid=====
    conn SetProperty("pid", player.Pid)
    //=====
    fmt.Println("====> Player pidId = ", player.Pid, " arrived
    ===")
}
```

接下来，我们来看一下Player里的Talk实现方法：

mmo\_game/core/player.go

```
//广播玩家聊天
func (p *Player) Talk(content string) {
    //1. 组建MsgId200 proto数据
    msg := &pb.BroadCast{
        Pid:p.Pid,
        Tp:1, //TP 1 代表聊天广播
        Data: &pb.BroadCast_Content{
            Content: content,
        },
    }

    //2. 得到当前世界所有的在线玩家
    players := WorldMgrObj.GetAllPlayers()

    //3. 向所有的玩家发送MsgId:200消息
    for _, player := range players {
        player.SendMsg(200, msg)
    }
}
```

## C) 测试世界聊天功能

我们在服务端运行server

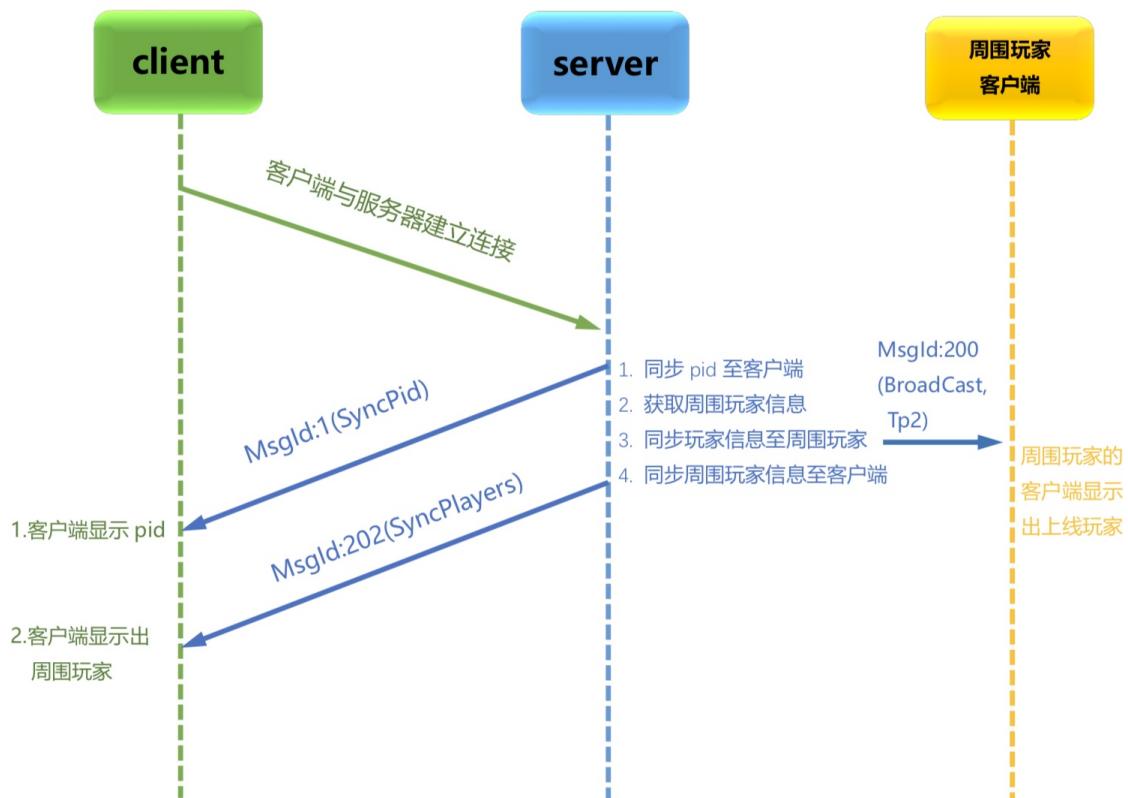
```
$go run server.go
$ go run server.go
Add api msgId = 2
[START] Server name: Zinx Game, listenner at IP: 0.0.0.0, Port 89
99 is starting
[Zinx] Version: V0.11, MaxConn: 3000, MaxPacketSize: 4096
start Zinx server    Zinx Game  succ, now listenning...
Worker ID = 9 is started.
Worker ID = 4 is started.
Worker ID = 5 is started.
Worker ID = 6 is started.
Worker ID = 7 is started.
Worker ID = 8 is started.
Worker ID = 0 is started.
Worker ID = 1 is started.
Worker ID = 2 is started.
Worker ID = 3 is started.
```

打开两个客户端，分别互相聊天，效果如下，我们的聊天功能已经实现了。





## 八、上线位置信息同步



这里涉及到了MsgID:202消息，我们应该在proto文件中，再添加两个消息

mmo\_game/pb/msg.proto

```
//玩家信息
message Player{
    int32 Pid=1;
    Position P=2;
}

//同步玩家显示数据
message SyncPlayers{
    repeated Player ps=1;
}
```

执行build.sh 生成对应的 msg.proto.go 文件。

## 八、上线位置信息同步

接下来，我们就要给player提供一个同步位置的方法了。

mmo\_game/core/player.go

```
//给当前玩家周边的(九宫格内)玩家广播自己的位置，让他们显示自己
func (p *Player) SyncSurrounding() {
    //1 根据自己的位置，获取周围九宫格内的玩家pid
    pids := WorldMgrObj.AoiMgr.GetPidsByPos(p.X, p.Z)
    //2 根据pid得到所有玩家对象
    players := make([]*Player, 0, len(pids))
    //3 给这些玩家发送MsgID:200消息，让自己出现在对方视野中
    for _, pid := range pids {
        players = append(players, WorldMgrObj.GetPlayerById(int
32(pid)))
    }
    //3.1 组建MsgId200 proto数据
    msg := &pb.BroadCast{
        Pid:p.Pid,
        Tp:2,//TP2 代表广播坐标
        Data: &pb.BroadCast_P{
            P:&pb.Position{
                X:p.X,
                Y:p.Y,
                Z:p.Z,
                V:p.V,
            },
        },
    }
    //3.2 每个玩家分别给对应的客户端发送200消息，显示人物
    for _, player := range players {
        player.SendMsg(200, msg)
    }
    //4 让周围九宫格内的玩家出现在自己的视野中
    //4.1 制作Message SyncPlayers 数据
    playersData := make([]*pb.Player, 0, len(players))
    for _, player := range players {
        p := &pb.Player{
            Pid:player.Pid,
            P:&pb.Position{
                X:player.X,
```

## 八、上线位置信息同步

```
        Y:player.Y,
        Z:player.Z,
        V:player.V,
    },
}
playersData = append(playersData, p)
}

//4.2 封装SyncPlayer protobuf数据
SyncPlayersMsg := &pb.SyncPlayers{
    Ps:playersData[:],
}

//4.3 给当前玩家发送需要显示周围的全部玩家数据
p.SendMsg(202, SyncPlayersMsg)
}
```

这里的过程只有两个重要过程，一个是将自己的坐标信息发送给AOI范围周边的玩家，一个是将周边玩家的坐标信息发送给自己的客户端。

最后我们在用户上线的时候，调用同步坐标信息的方法。

mmo\_game/server.go

## 八、上线位置信息同步

```
//当客户端建立连接的时候的hook函数
func OnConnecionAdd(conn ziface.IConnection) {
    //创建一个玩家
    player := core.NewPlayer(conn)
    //同步当前的PlayerID给客户端，走MsgID:1 消息
    player.SyncPid()
    //同步当前玩家的初始化坐标信息给客户端，走MsgID:200消息
    player.BroadCastStartPosition()
    //将当前新上线玩家添加到worldManager中
    core.WorldMgrObj.AddPlayer(player)
    //将该连接绑定属性Pid
    conn SetProperty("pid", player.Pid)

    //=====同步周边玩家上线信息，与现实周边玩家信息=====
    player.SyncSurrounding()
    //=====

    fmt.Println("====> Player pidId = ", player.Pid, " arrived
    ===")
}
```

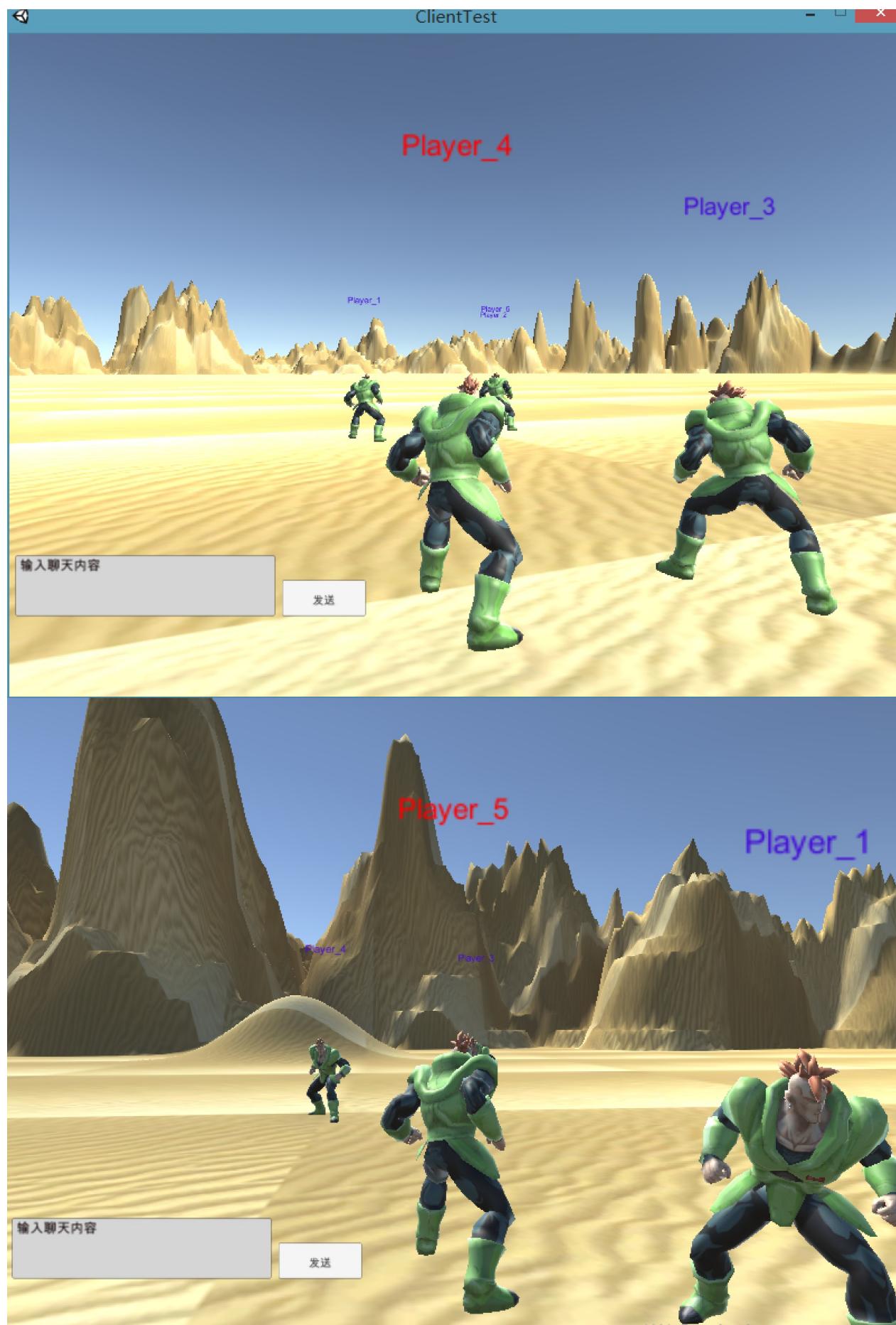
最后我们运行程序进行一下简单的测试.

启动server

```
$go run server.go
```

分别启动3个客户端，看是否能够互相看到对方。

## 八、上线位置信息同步

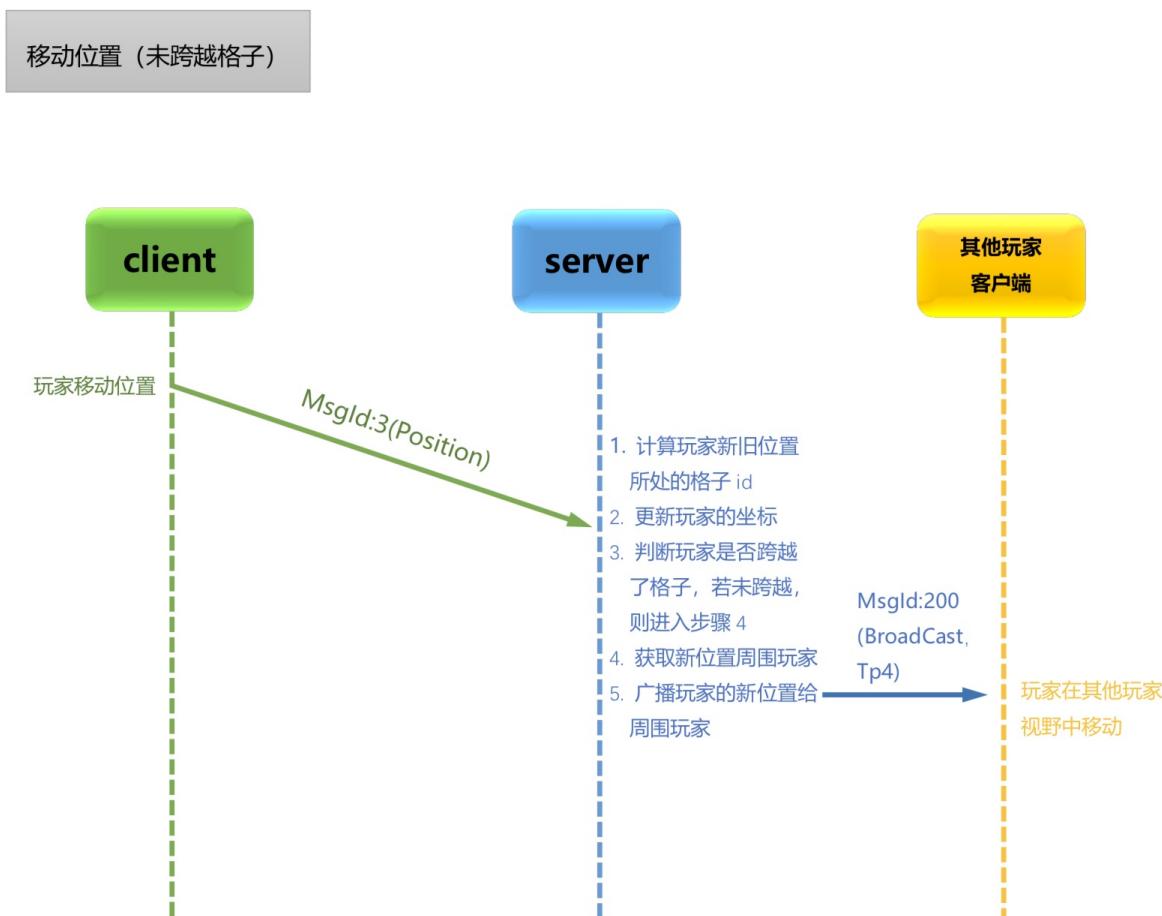


## 八、上线位置信息同步

---

## 九、移动位置与AOI广播(未跨越格子)

现在我们来添加玩家移动的时候，周边玩家显示同步位置，具体流程图，如下：



这里面涉及到两个消息 `MsgID:3` 和 `MsgID:200`，`Tp=4`。当玩家移动的时候，客户端会主动给服务端发送 `MsgID:3` 的消息。所以首先，我们应该给服务端注册 `MsgID:3` 的路由处理业务。

mmo\_game/server.go

```
func main() {
    //创建服务器句柄
    s := znet.NewServer()

    //注册客户端连接建立和丢失函数
    s.SetOnConnStart(OnConnecionAdd)

    //注册路由
    s.AddRouter(2, &api.WorldChatApi{})      //聊天
    s.AddRouter(3, &api.MoveApi{})           //移动

    //启动服务
    s.Serve()
}
```

接下来，我们需要创建一个api接口，实现 `MoveApi{}` 模块.

mmo\_game/api/move.go

```
package api

import (
    "fmt"
    "github.com/golang/protobuf/proto"
    "zinx/ziface"
    "zinx/zinx_app_demo/mmo_game/core"
    "zinx/zinx_app_demo/mmo_game/pb"
    "zinx/znet"
)

//玩家移动
type MoveApi struct {
    znet.BaseRouter
}

func (*MoveApi) Handle(request ziface IRequest) {
    //1. 将客户端传来的proto协议解码
```

```
msg := &pb.Position{}
err := proto.Unmarshal(request.GetData(), msg)
if err != nil {
    fmt.Println("Move: Position Unmarshal error ", err)
    return
}

//2. 得知当前的消息是从哪个玩家传递来的,从连接属性pid中获取
pid, err := request.GetConnection().GetProperty("pid")
if err != nil {
    fmt.Println("GetProperty pid error", err)
    request.GetConnection().Stop()
    return
}

fmt.Printf("user pid = %d , move(%f,%f,%f,%f)", pid, msg.X,
msg.Y, msg.Z, msg.V)

//3. 根据pid得到player对象
player := core.WorldMgrObj.GetPlayerByPid(pid.(int32))

//4. 让player对象发起移动位置信息广播
player.UpdatePos(msg.X, msg.Y, msg.Z, msg.V)
}
```

`move.go` 的业务和我们之前的 `world_chat.go` 的业务很像。最后调用了 `Player.UpdatePos()` 方法，该方法是主要处理及发送同步消息的方法。我们接下来一起实现这个方法。

mmo\_game/core/player.go

```
//广播玩家位置移动
func (p *Player) UpdatePos(x float32, y float32, z float32, v float32) {
    //更新玩家的位置信息
    p.X = x
    p.Y = y
    p.Z = z
    p.V = v
```

```
//组装protobuf协议，发送位置给周围玩家
msg := &pb.BroadCast{
    Pid:p.Pid,
    Tp:4,           //4 - 移动之后的坐标信息
    Data: &pb.BroadCast_P{
        P:&pb.Position{
            X:p.X,
            Y:p.Y,
            Z:p.Z,
            V:p.V,
        },
    },
}

//获取当前玩家周边全部玩家
players := p.GetSurroundingPlayers()
//向周边的每个玩家发送MsgID:200消息，移动位置更新消息
for _, player := range players {
    player.SendMsg(200, msg)
}
}

//获得当前玩家的AOI周边玩家信息
func (p *Player) GetSurroundingPlayers() []*Player {
    //得到当前AOI区域的所有pid
    pids := WorldMgrObj.AoiMgr.GetPidsByPos(p.X, p.Z)

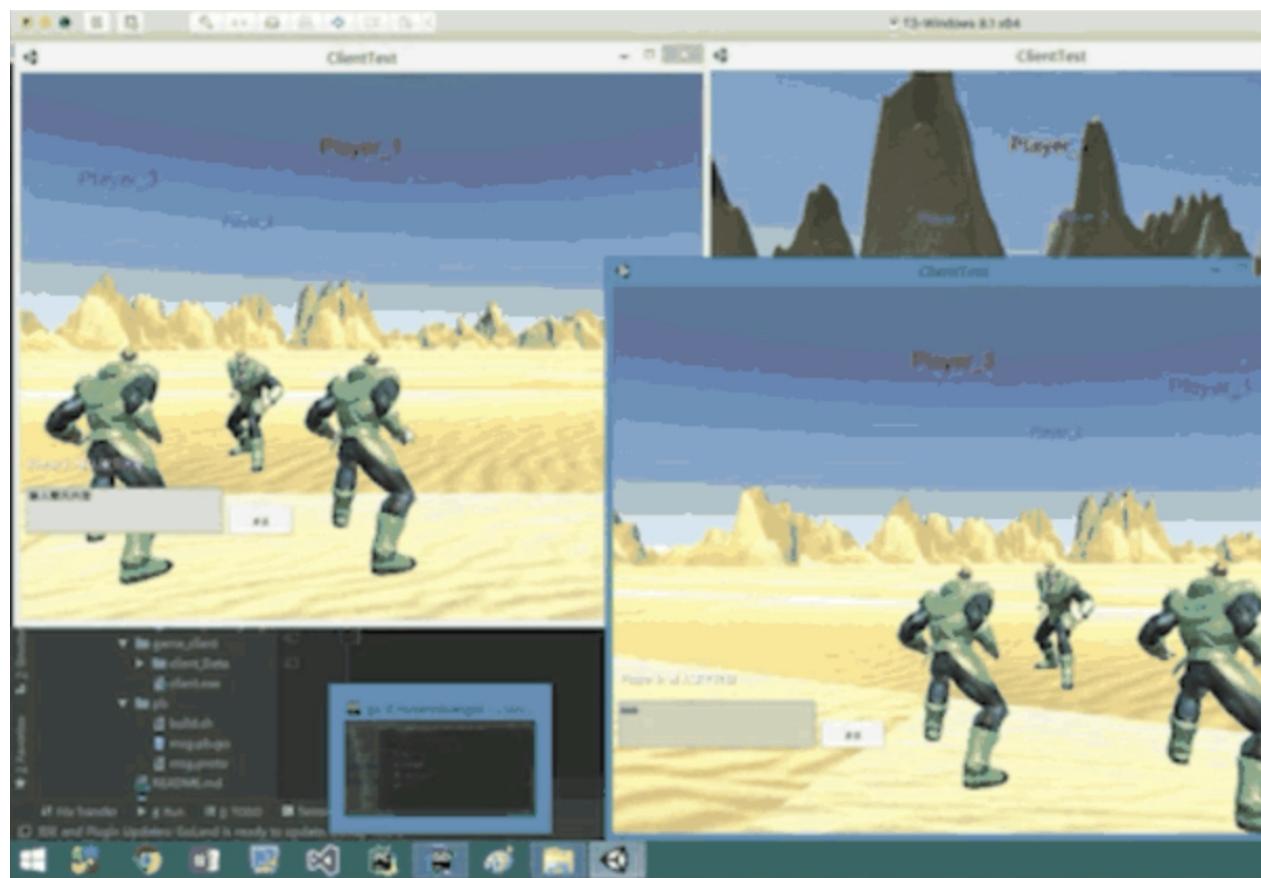
    //将所有pid对应的Player放到Player切片中
    players := make([]*Player, 0, len(pids))
    for _, pid := range pids {
        players = append(players, WorldMgrObj.GetPlayerByPid(int32(pid)))
    }

    return players
}
```

其中 `GetSurroundingPlayers()` 是获取当前玩家AOI周边的玩家Player对象有哪些。

该方法的整体思路是获取周边的所有玩家，发送位置更新信息。

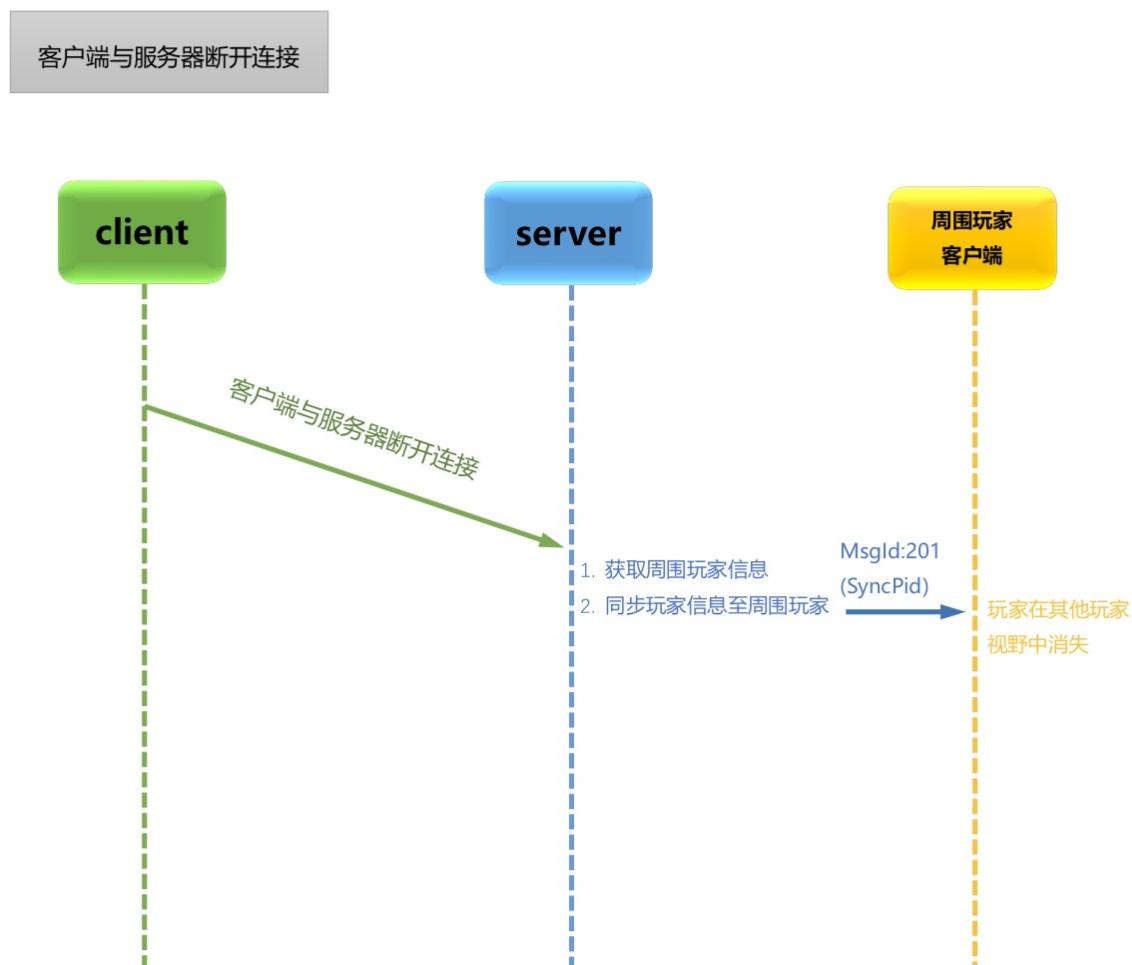
下面我们再次启动服务器，同时开3个客户端，看看最后的效果。



显示证明，3个客户端已经可以实现移动同步的过程，那么实际上，我们基本的MMO大型网游在线游戏的基础模型已经搭建完成了，接下来至于添加一些其他的游戏机制，比如对战，积分等。实际上可以基于这个开发架构和流程继续迭代开发了。

# 十、玩家下线

那么玩家客户端点击关闭应该触发玩家下线的功能。玩家应该在其他的客户端消失。  
具体流程如下：



玩家下线，这里采用了 `MsgID:201` 消息。触发该流程的时机是客户端与服务端断开链接，那么我们就可以在连接断开前的Hook方法中，实现此业务。

mmo\_game/server.go

```
//当客户端断开连接的时候的hook函数
func OnConnectionLost(conn ziface.IConnection) {
    //获取当前连接的Pid属性
    pid, _ := conn.GetProperty("pid")

    //根据pid获取对应的玩家对象
    player := core.WorldMgrObj.GetPlayerByPid(pid.(int32))

    //触发玩家下线业务
    if pid != nil {
        player.LostConnection()
    }

    fmt.Println("====> Player ", pid, " left ====")

}

func main() {
    //创建服务器句柄
    s := znet.NewServer()

    //注册客户端连接建立和丢失函数
    s.SetOnConnStart(OnConnecionAdd)
    // ===== 注册 hook 函数 =====
    s.SetOnConnStop(OnConnectionLost)
    // =====

    //注册路由
    s.AddRouter(2, &api.WorldChatApi{})
    s.AddRouter(3, &api.MoveApi{})

    //启动服务
    s.Serve()
}
```

然后我们就要给player模块提供一个 `LostConnection()` 方法。

mmo\_game/core/player.go

```
//玩家下线
func (p *Player) LostConnection() {
    //1 获取周围AOI九宫格内的玩家
    players := p.GetSurroundingPlayers()

    //2 封装MsgID:201消息
    msg := &pb.SyncPid{
        Pid:p.Pid,
    }

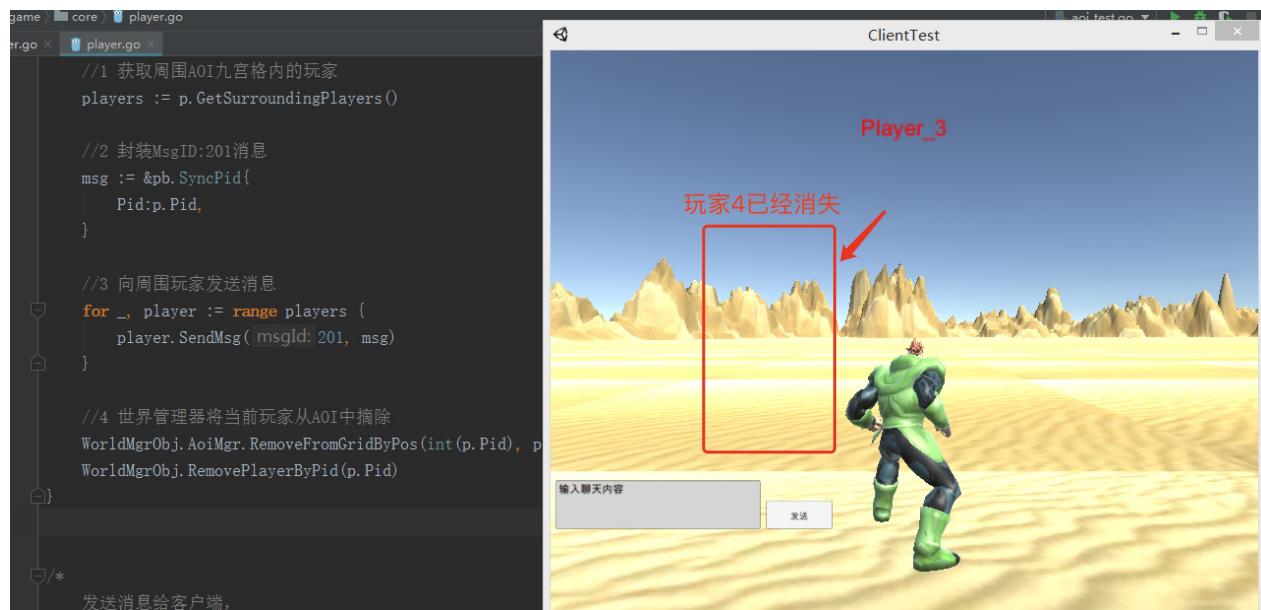
    //3 向周围玩家发送消息
    for _, player := range players {
        player.SendMsg(201, msg)
    }

    //4 世界管理器将当前玩家从AOI中摘除
    WorldMgrObj.AoiMgr.RemoveFromGridByPos(int(p.Pid), p.X, p.Z)
    WorldMgrObj.RemovePlayerByPid(p.Pid)
}
```

接下来我们启动服务器，再启动两个客户端分别测试一下结果。

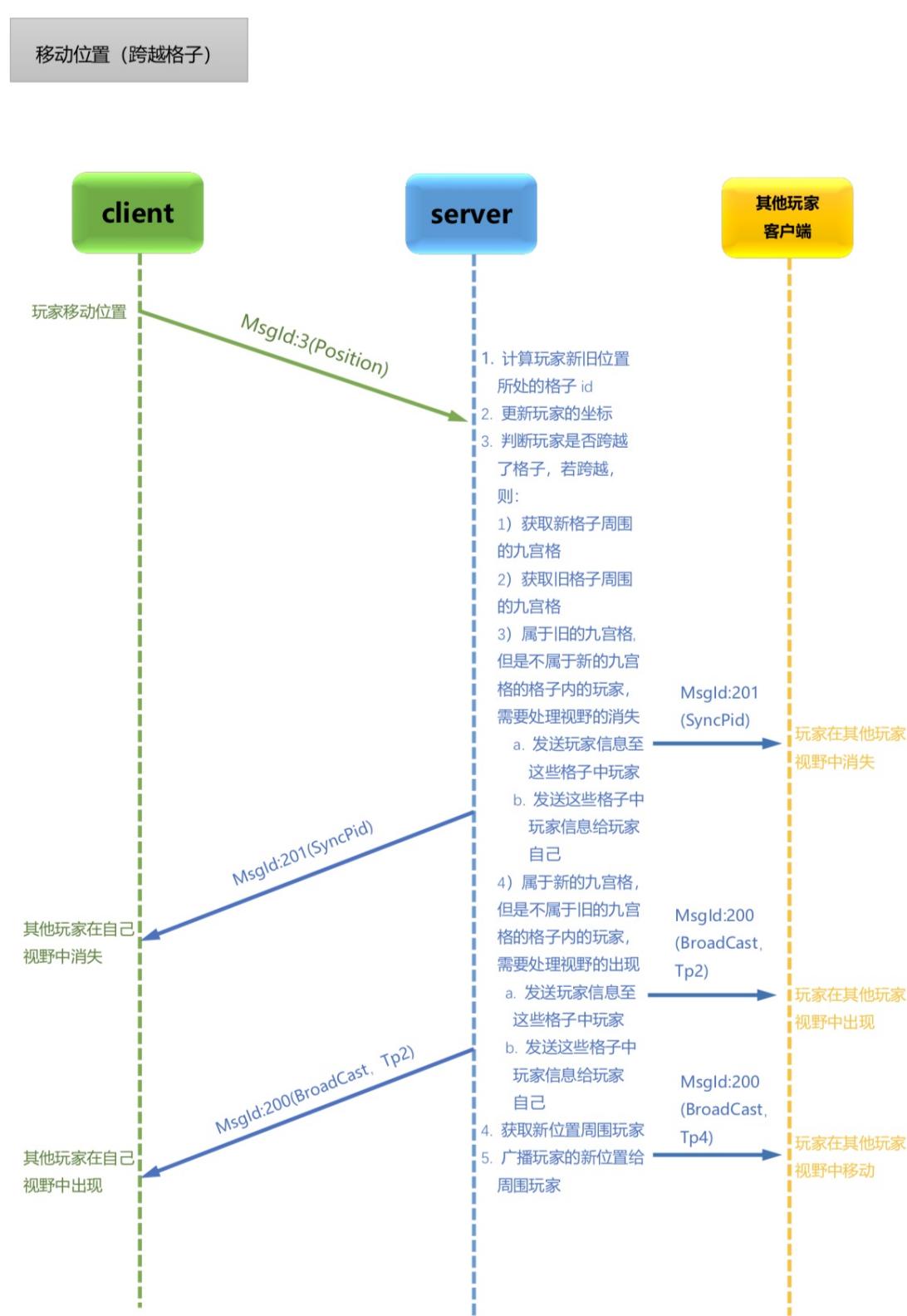


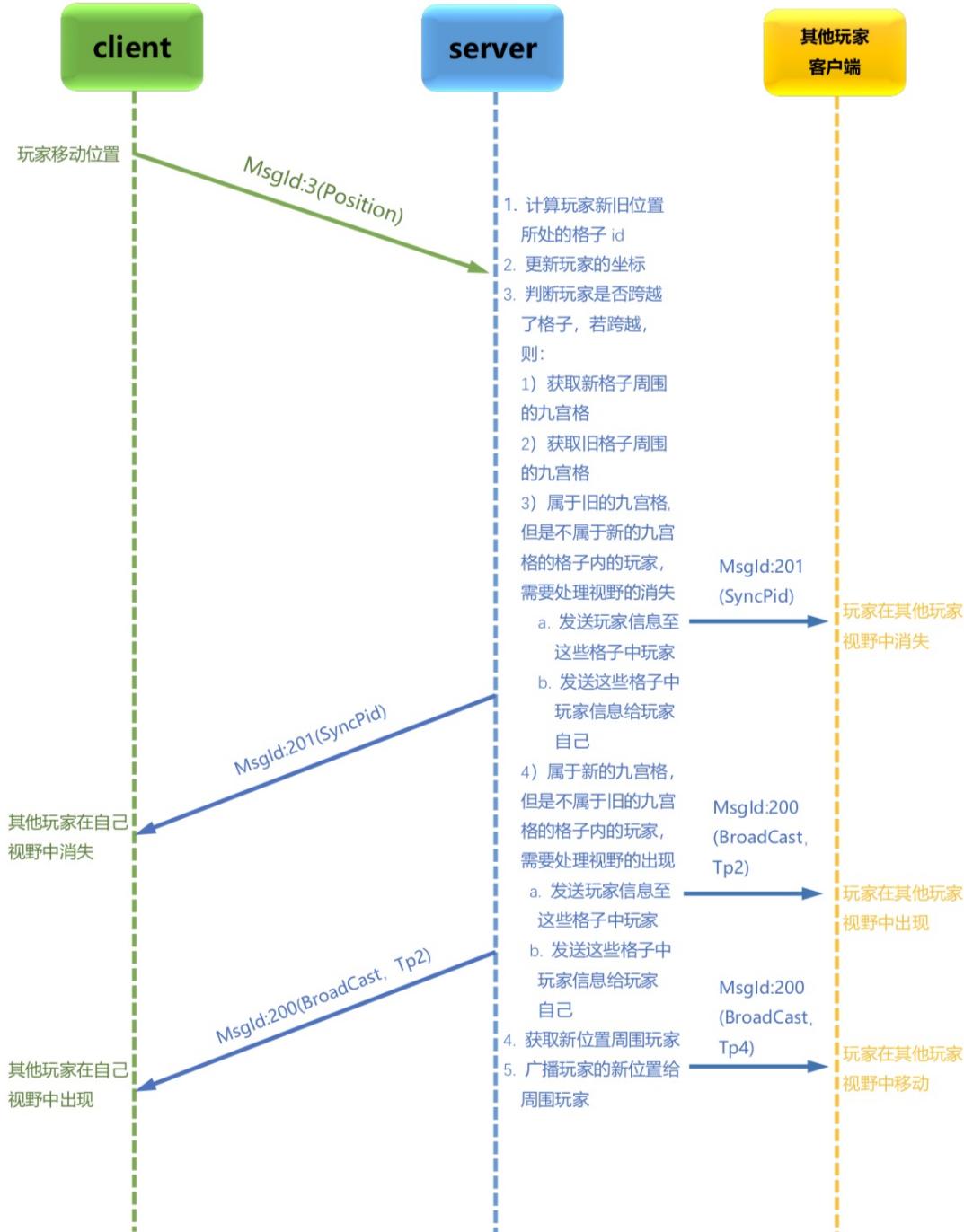
## 十、玩家下线

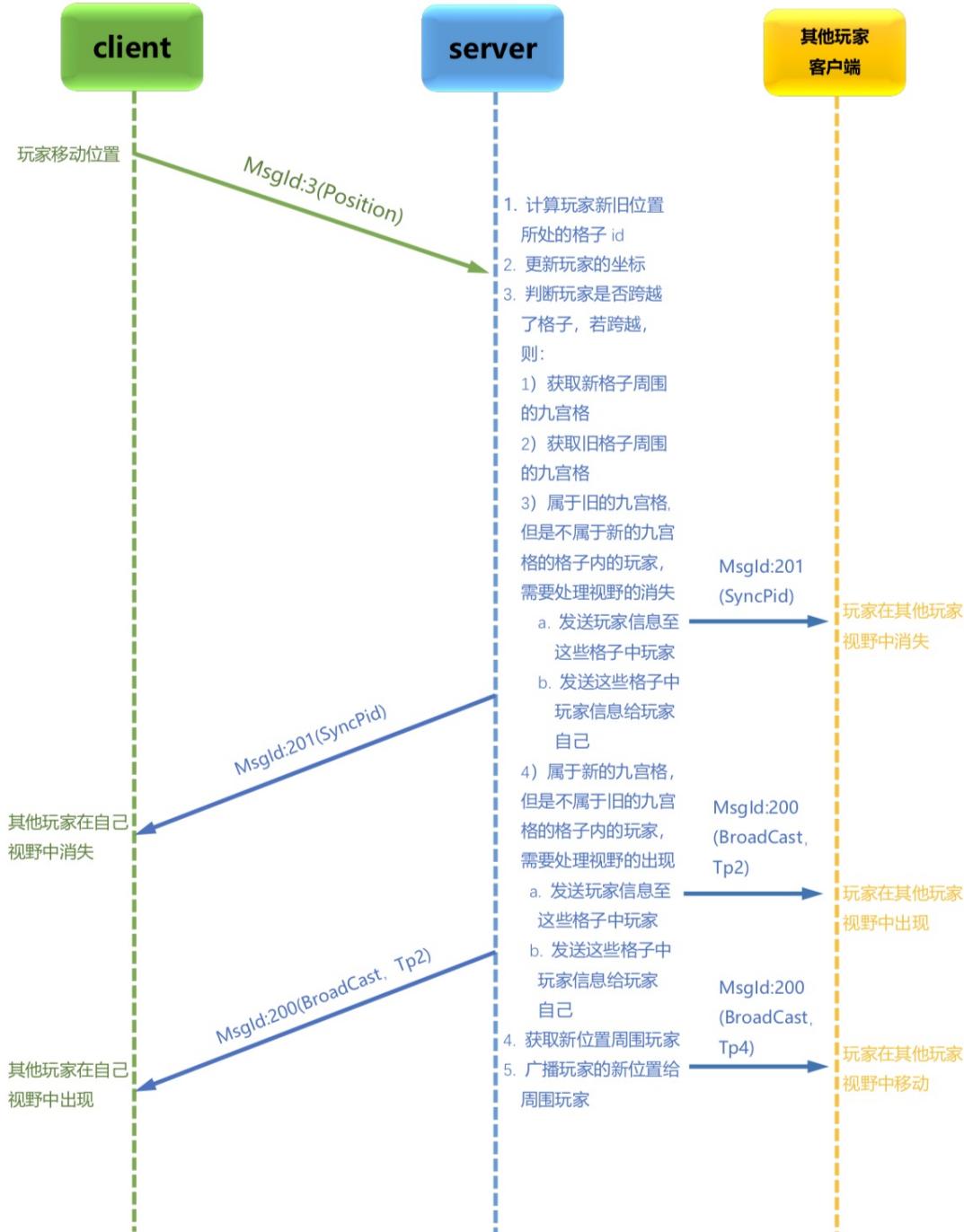


当我们退出一个客户端的时候，另外一个客户端的玩家的地图已经会摘除当前玩家了。

# 十一、移动与AOI广播(跨越格子)







移动位置 (跨越格子)

