**Go游戏服务器框架Leaf学习分享**

**Leaf 游戏服务器框架简介：**

Leaf 是一个由 Go 语言（golang）编写的开发效率和执行效率并重的开源游戏服务器框架。Leaf 适用于各类游戏服务器的开发，包括 H5（HTML5）游戏服务器。

Wiki地址[https://github.com/name5566/leaf/wiki]

**Leaf关注点：**

良好的使用体验。Leaf 总是尽可能的提供简洁和易用的接口，尽可能的提升开发的效率。

稳定性。Leaf 总是尽可能的恢复运行过程中的错误，避免崩溃。

多核支持。Leaf 通过模块机制和 [leaf/go](https://github.com/name5566/leaf/tree/master/go) 尽可能的利用多核资源，同时又尽量避免各种副作用。

模块机制。

**Leaf 服务器的主要模块：**

**ChanRpc**

Rpc是远程过程调用的简称，原来是通过Tpc手段使得一个本地的函数调用，将调用信息传递给其他服务器执行，并通过Tcp返回结果的一种技术手段，在分布式中，这种调用十分常见。但是这里的ChanRpc是在不同协程中进行函数调用，其实现的手段是Chan，所以成为ChanRpc。你可以将不同的功能模块实现为ChanRpc并提供给其他模块调用。

**Cluster**

Cluster 主要是管理集群，但是Leaf本身专注的还是单机服务，所以这个模块的功能现在还没有实现。

**Conf**

Conf 是Leaf的配置管理模块。里面主要是Leaf启动的一些必要信息。

**Console**

Console 模块为Leaf管理提供了一个终端接口，你可以使用Telnet连接上去动态的修改参数，或者指向命令。其内部实现了Help, CpuProf, Prof命令，并提供扩展，可以方便的添加其他命令。另外，扩展命令是通过ChanRpc实现的。

**DB**

DB模块提供里Mongo支持，也可以在这里聚合其他DB模块。

**Gate**

Gate 模块为Leaf提供接入功能。这个模块的功能很重要，是服务器的入口。它能同时监听TcpSocket和WebSocket。主要流程是在接入连接的时候创建一个Agent，并将这个Agent通知给AgentRpc。其核心其实是一个TcpServer和WebScoketServer，他的协议函数能够将socket字节流分包，封装为Msg传递给Agent。其工作流可以查看Server模块。

**Go**

Go模块是对golang中go提供一些额外功能。Go提供回调功能，LinearContext提供顺序调用功能。

**Log**

Log主要提供日志分级功能。

**Module**

Module 为Leaf提供模块化支持。Skeleton是Leaf的整体骨架，它聚合了Leaf中其他一些异步调用模块的功能，使得各模块之间能够协同工作。

**Network**

Network是Leaf的网络部分，这部分比较大，包含一个json和protobuf解包模块。

**Recordfile**

Recordfile 提供序列化和反序列化为文本的功能。

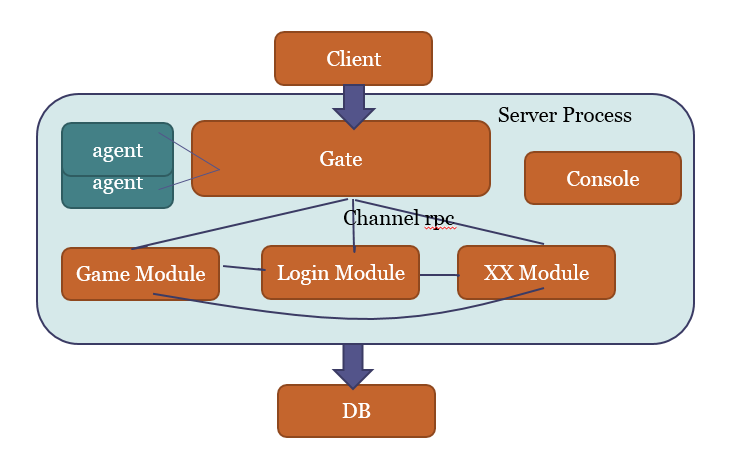
**Timer**

Timer主要是提供一个Cron功能的定时器服务，其中Timer是time.AfterFunc的封装，是为了方便聚合到Skeleton中。

**Util**

Util是工具类的集合

**Leaf服务器结构图：**



服务器通过Gate接受来自Client的消息，Gate上会有对应的Agent负责收发消息。Agent上有对应的user data。

针对不同服务，服务器创建不同的Module。每个Module跑在不同的goroutine上，Module之间通过Channel通信。具体方式为，各个模块对其他模块公开自己的Channel，其他模块将请求传入对应模块的Channel。在处理请求时，agent也被传递到对应Module，进行各种操作。

Server Process中的数据在特定时候持久化到DB上（目前支持Mongo DB）。

Console用于监控服务器状态，接收管理指令。

**Leaf游戏服务器示例：**

[LeafServer](https://github.com/name5566/leafserver) 是一个基于 Leaf 开发的游戏服务器，我们以 LeafServer 作为起点。

获取 LeafServer：

```

git clone https://github.com/name5566/leafserver

```

设置 leafserver 目录到 GOPATH 环境变量后获取 Leaf：

```

go get github.com/name5566/leaf

```

编译 LeafServer：

```

go install server

```

如果一切顺利，运行 server 你可以获得以下输出：

```

2015/08/26 22:11:27 [release] Leaf 1.1.2 starting up

```

敲击 Ctrl + C 关闭游戏服务器，服务器正常关闭输出：

```

2015/08/26 22:12:30 [release] Leaf closing down (signal: interrupt)

```

### Hello Leaf

现在，在 LeafServer 的基础上，我们来看看游戏服务器如何接收和处理网络消息。

首先定义一个 JSON 格式的消息（protobuf 类似）。打开 LeafServer msg/msg.go 文件可以看到如下代码：

```go

package msg

import (

"github.com/name5566/leaf/network"

)

var Processor network.Processor

func init() {

}

```

Processor 为消息的处理器（可由用户自定义），这里我们使用 Leaf 默认提供的 JSON 消息处理器并尝试添加一个名字为 Hello 的消息：

```go

package msg

import (

"github.com/name5566/leaf/network/json"

)

// 使用默认的 JSON 消息处理器（默认还提供了 protobuf 消息处理器）

var Processor = json.NewProcessor()

func init() {

// 这里我们注册了一个 JSON 消息 Hello

Processor.Register(&Hello{})

}

// 一个结构体定义了一个 JSON 消息的格式

// 消息名为 Hello

type Hello struct {

Name string

}

```

客户端发送到游戏服务器的消息需要通过 gate 模块路由，简而言之，gate 模块决定了某个消息具体交给内部的哪个模块来处理。这里，我们将 Hello 消息路由到 game 模块中。打开 LeafServer gate/router.go，敲入如下代码：

```go

package gate

import (

"server/game"

"server/msg"

)

func init() {

// 这里指定消息 Hello 路由到 game 模块

// 模块间使用 ChanRPC 通讯，消息路由也不例外

msg.Processor.SetRouter(&msg.Hello{}, game.ChanRPC)

}

```

一切就绪，我们现在可以在 game 模块中处理 Hello 消息了。打开 LeafServer game/internal/handler.go，敲入如下代码：

```go

package internal

import (

"github.com/name5566/leaf/log"

"github.com/name5566/leaf/gate"

"reflect"

"server/msg"

)

func init() {

// 向当前模块（game 模块）注册 Hello 消息的消息处理函数 handleHello

handler(&msg.Hello{}, handleHello)

}

func handler(m interface{}, h interface{}) {

skeleton.RegisterChanRPC(reflect.TypeOf(m), h)

}

func handleHello(args []interface{}) {

// 收到的 Hello 消息

m := args[0].(\*msg.Hello)

// 消息的发送者

a := args[1].(gate.Agent)

// 输出收到的消息的内容

log.Debug("hello %v", m.Name)

// 给发送者回应一个 Hello 消息

a.WriteMsg(&msg.Hello{

Name: "client",

})

}

```

到这里，一个简单的范例就完成了。为了更加清楚的了解消息的格式，我们从 0 编写一个最简单的测试客户端。

Leaf 中，当选择使用 TCP 协议时，在网络中传输的消息都会使用以下格式：

```

--------------

| len | data |

--------------

```

其中：

1. len 表示了 data 部分的长度（字节数）。len 本身也有长度，默认为 2 字节（可配置），len 本身的长度决定了单个消息的最大大小

2. data 部分使用 JSON 或者 protobuf 编码（也可自定义其他编码方式）

测试客户端同样使用 Go 语言编写：

```go

package main

import (

"encoding/binary"

"net"

)

func main() {

conn, err := net.Dial("tcp", "127.0.0.1:3563")

if err != nil {

panic(err)

}

// Hello 消息（JSON 格式）

// 对应游戏服务器 Hello 消息结构体

data := []byte(`{

"Hello": {

"Name": "leaf"

}

}`)

// len + data

m := make([]byte, 2+len(data))

// 默认使用大端序

binary.BigEndian.PutUint16(m, uint16(len(data)))

copy(m[2:], data)

// 发送消息

conn.Write(m)

}

```

执行此测试客户端，游戏服务器输出：

```

2015/09/25 07:41:03 [debug ] hello leaf

2015/09/25 07:41:03 [debug ] read message: read tcp 127.0.0.1:3563->127.0.0.1:54599: wsarecv: An existing connection was forcibly closed by the remote host.

```

**实现重点分析：**

一个 Leaf 开发的游戏服务器由多个模块组成，模块有以下特点：

每个模块运行在一个单独的 goroutine 中

模块间通过一套轻量的 RPC 机制通讯(chanrpc)

游戏服务器在启动时进行模块的注册，例如：

```go

leaf.Run(

game.Module,

gate.Module,

login.Module,

)

```

这里按顺序注册了 game、gate、login 三个模块。每个模块都需要实现接口：

```go

type Module interface {

OnInit()

OnDestroy()

Run(closeSig chan bool)

}

```

Leaf 首先会在同一个 goroutine 中按模块注册顺序执行模块的 OnInit 方法，等到所有模块 OnInit 方法执行完成后则为每一个模块启动一个 goroutine 并执行模块的 Run 方法。最后，游戏服务器关闭时（Ctrl + C 关闭游戏服务器）将按模块注册相反顺序在同一个 goroutine 中执行模块的 OnDestroy 方法。

模块中最关键的就是 skeleton（骨架），skeleton 实现了 Module 接口的 Run 方法并提供了ChanRPC，Goroutine，Timer

**### Leaf ChanRPC**

由于 Leaf 中，每个模块跑在独立的 goroutine 上，为了模块间方便的相互调用就有了基于 channel 的 RPC 机制。一个 ChanRPC 需要在游戏服务器初始化的时候进行注册（注册过程不是 goroutine 安全的），例如 LeafServer 中 game 模块注册了 NewAgent 和 CloseAgent 两个 ChanRPC：

```go

package internal

import (

"github.com/name5566/leaf/gate"

)

func init() {

skeleton.RegisterChanRPC("NewAgent", rpcNewAgent)

skeleton.RegisterChanRPC("CloseAgent", rpcCloseAgent)

}

func rpcNewAgent(args []interface{}) {

}

func rpcCloseAgent(args []interface{}) {

}

**```**

使用 skeleton 来注册 ChanRPC。RegisterChanRPC 的第一个参数是 ChanRPC 的名字，第二个参数是 ChanRPC 的实现。这里的 NewAgent 和 CloseAgent 会被 LeafServer 的 gate 模块在连接建立和连接中断时调用。ChanRPC 的调用方有 3 种调用模式：

1. 同步模式，调用并等待 ChanRPC 返回

2. 异步模式，调用并提供回调函数，回调函数会在 ChanRPC 返回后被调用

3. Go 模式，调用并立即返回，忽略任何返回值和错误

gate 模块这样调用 game 模块的 NewAgent ChanRPC（这仅仅是一个示例，实际的代码细节复杂的多）：

```go

game.ChanRPC.Go("NewAgent", a)

```

这里调用 NewAgent 并传递参数 a，我们在 rpcNewAgent 的参数 args[0] 中可以取到 a（args[1] 表示第二个参数，以此类推）。

**### Leaf Go**

善用 goroutine 能够充分利用多核资源，Leaf 提供的 Go 机制解决了原生 goroutine 存在的一些问题：

\* 能够恢复 goroutine 运行过程中的错误

\* 游戏服务器会等待所有 goroutine 执行结束后才关闭

\* 非常方便的获取 goroutine 执行的结果数据

\* 在一些特殊场合保证 goroutine 按创建顺序执行

我们来看一个例子（可以在 LeafServer 的模块的 OnInit 方法中测试）：

```go

log.Debug("1")

// 定义变量 res 接收结果

var res string

skeleton.Go(func() {

// 这里使用 Sleep 来模拟一个很慢的操作

time.Sleep(1 \* time.Second)

// 假定得到结果

res = "3"

}, func() {

log.Debug(res)

})

log.Debug("2")

```

**### Leaf timer**

Go 语言标准库提供了定时器的支持：

```go

func AfterFunc(d Duration, f func()) \*Timer

```

AfterFunc 会等待 d 时长后调用 f 函数，这里的 f 函数将在另外一个 goroutine 中执行。Leaf 提供了一个相同的 AfterFunc 函数，相比之下，f 函数在 AfterFunc 的调用 goroutine 中执行，这样就避免了同步机制的使用：

```go

skeleton.AfterFunc(5 \* time.Second, func() {

// ...

})

``` ```