Pitaya框架分析

简介

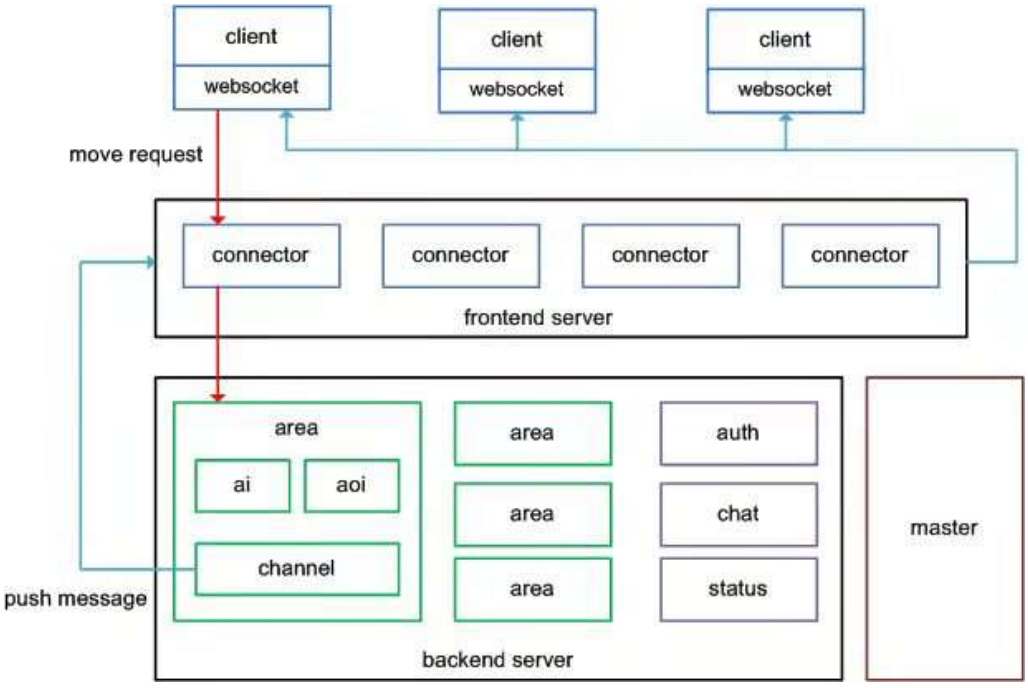
Pitaya 是一款由国外游戏公司 wildlifestudios 使用golang进行编写，易于使用，快速且轻量级的开源分布式游戏服务器框架。

Pitaya 是基于 nano 框架构建的，分布式架构和协议设计参考了 pomelo 框架。

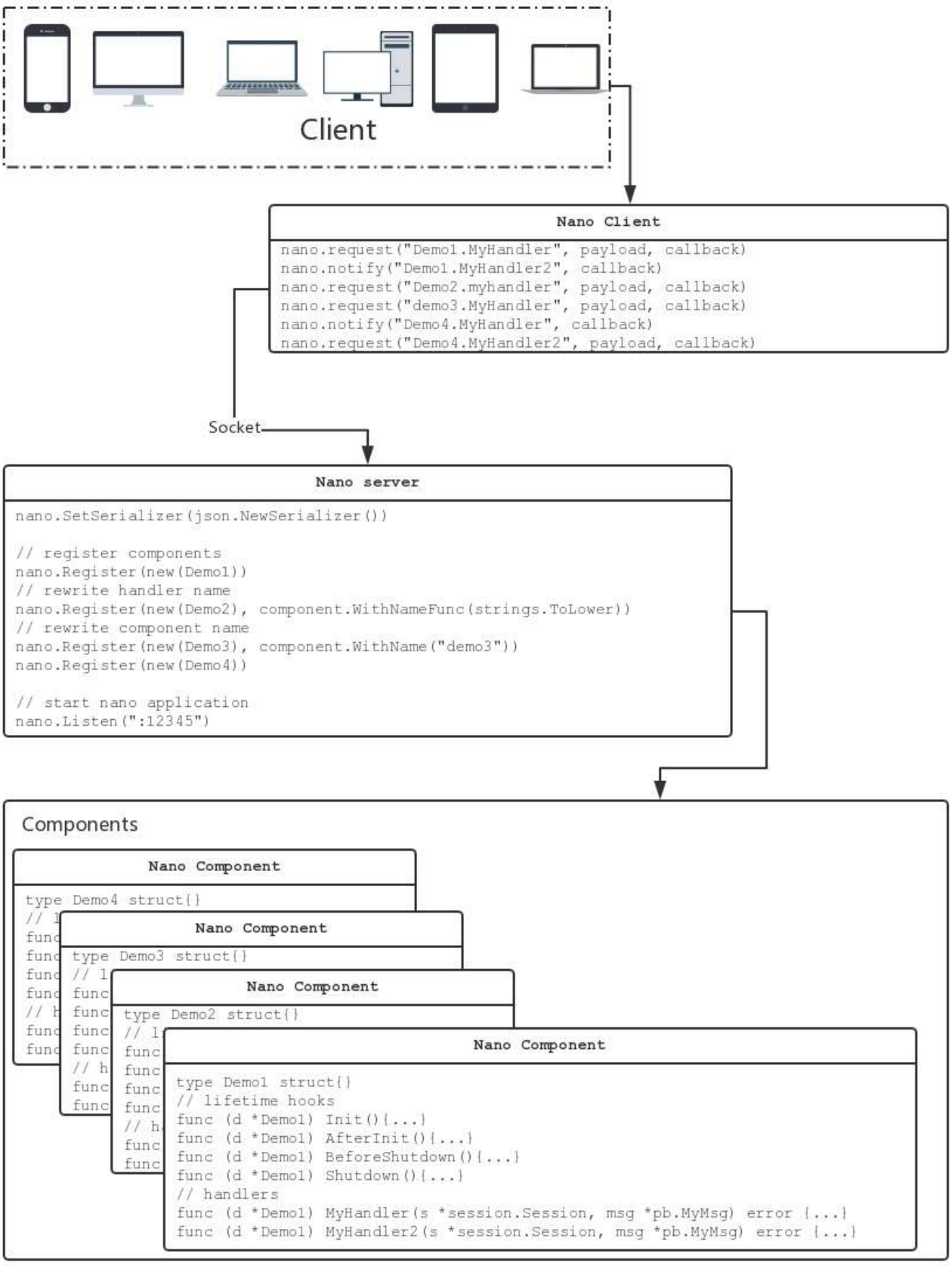
Pitaya 提供了 `Standalone` 和 `Cluster` 两种模式，在 `Cluster` 模式下使用 `etcd` 作为默认的服务发现组件，提供使用 `nats` 和 `grpc` 进行远程调用(server to server)的可选配置，并提供在docker中运行以上组件(etcd、nats)的docker-compose配置

架构分析

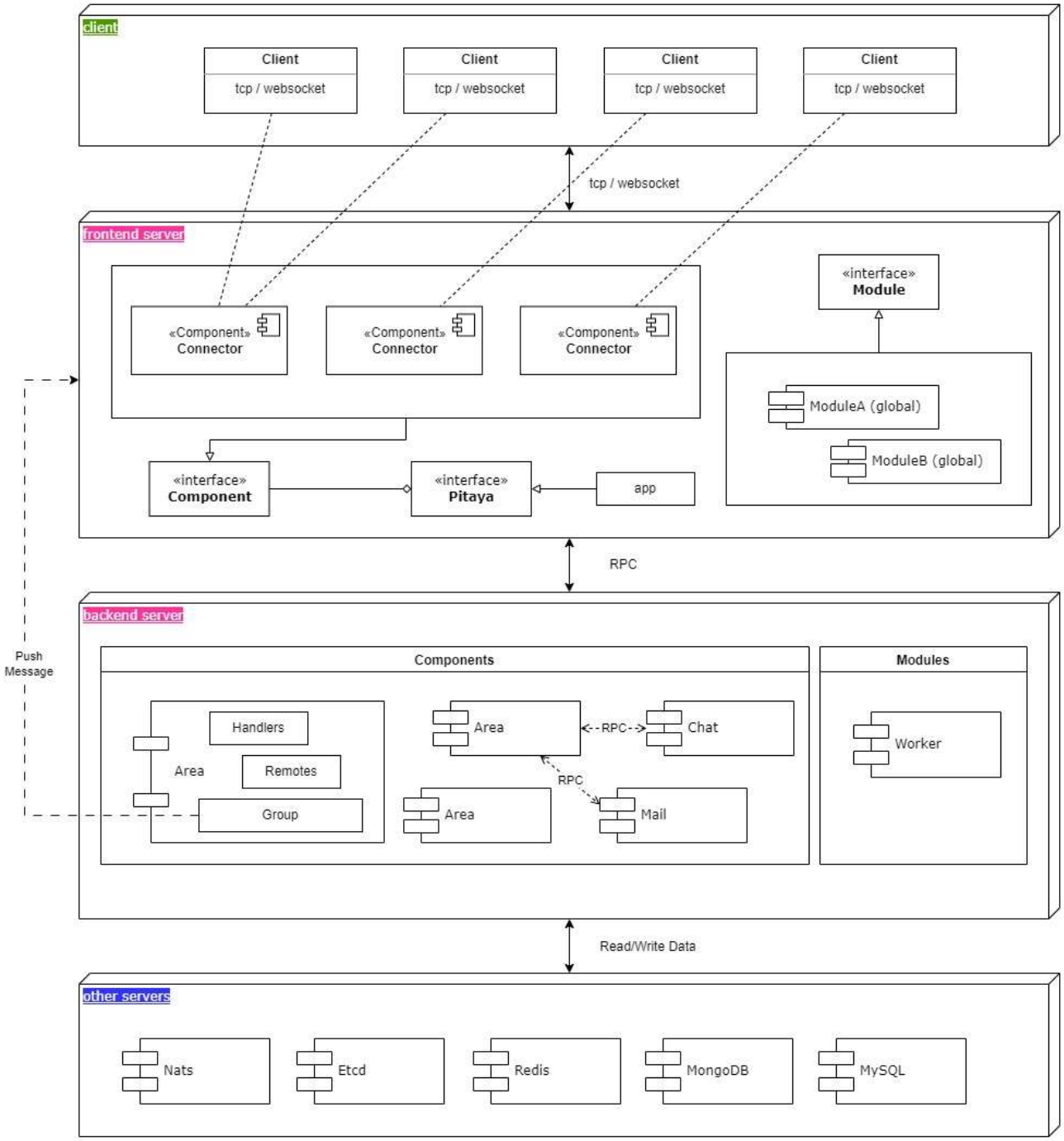
- Pomelo架构图（pitaya参考了pomelo的架构）



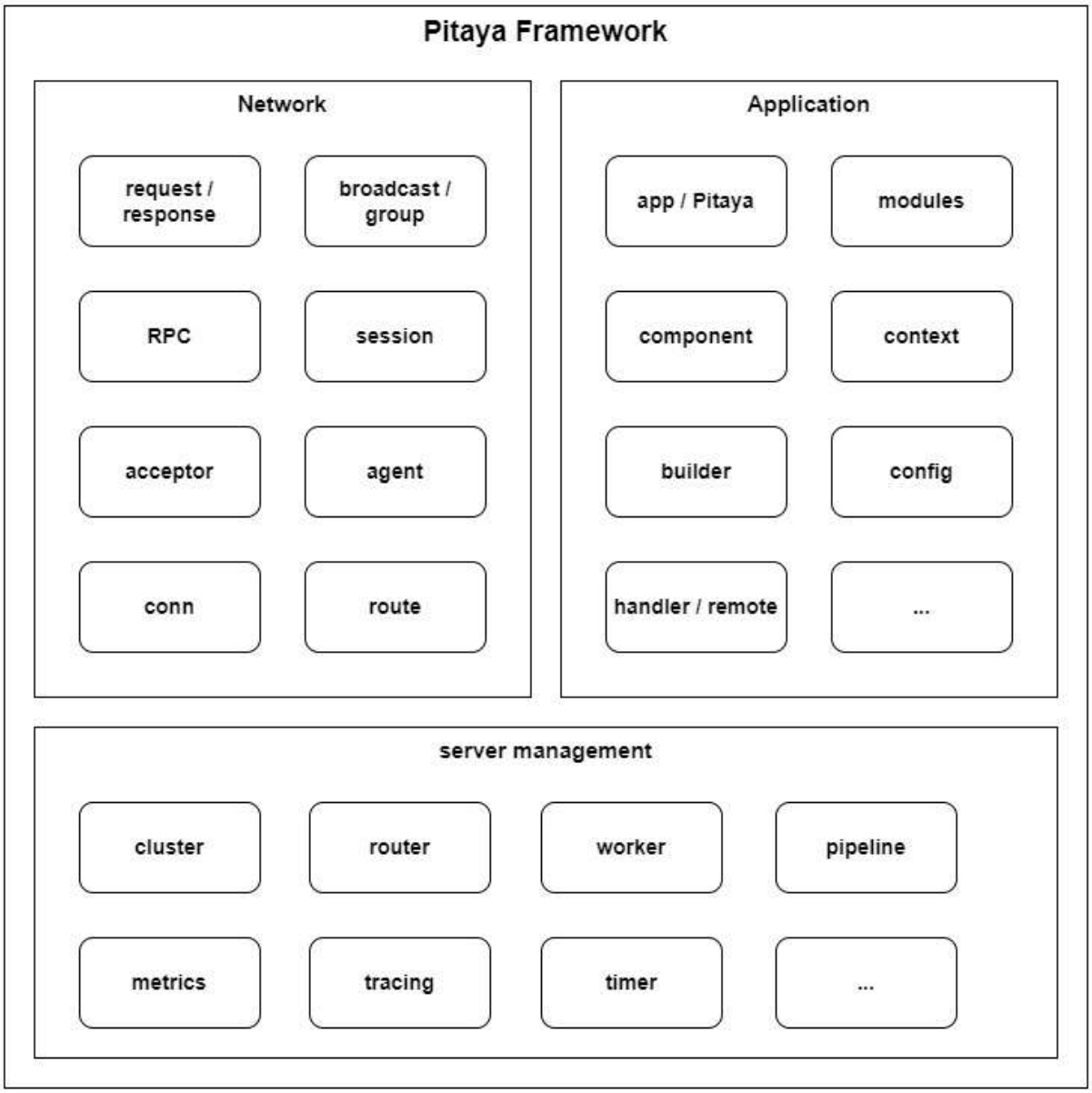
- Nano架构图（pitaya参考了nano的网络库）


Client
Nano Client
nano.request("Demo1.MyHandler", payload, callback)
nano.notify("Demo1.MyHandler2", callback)
nano.request("Demo2.myhandler", payload, callback)
nano.request("demo3.MyHandler", payload, callback)
nano.notify("Demo4.MyHandler", callback)
nano.request("Demo4.MyHandler2", payload, callback)
Socket
Nano server
nano.SetSerializer(json.NewSerializer())
// register components
nano.Register(new(Demo1))
// rewrite handler name
nano.Register(new(Demo2), component.WithNameFunc(strings.ToLower))
// rewrite component name
nano.Register(new(Demo3), component.WithName("demo3"))
nano.Register(new(Demo4))
// start nano application
nano.Listen(":12345")
Components
Nano Component
type Demo4 struct{}
Nano Component
type Demo3 struct{}
Nano Component
type Demo2 struct{}
Nano Component
type Demo1 struct{}
// lifetime hooks
func (d *Demo1) Init(){...}
func (d *Demo1) AfterInit(){...}
func (d *Demo1) BeforeShutdown(){...}
func (d *Demo1) Shutdown(){...}
// handlers
func (d *Demo1) MyHandler(s *session.Session, msg *pb.MyMsg) error {...}
func (d *Demo1) MyHandler2(s *session.Session, msg *pb.MyMsg) error {...}

- Pitaya架构图

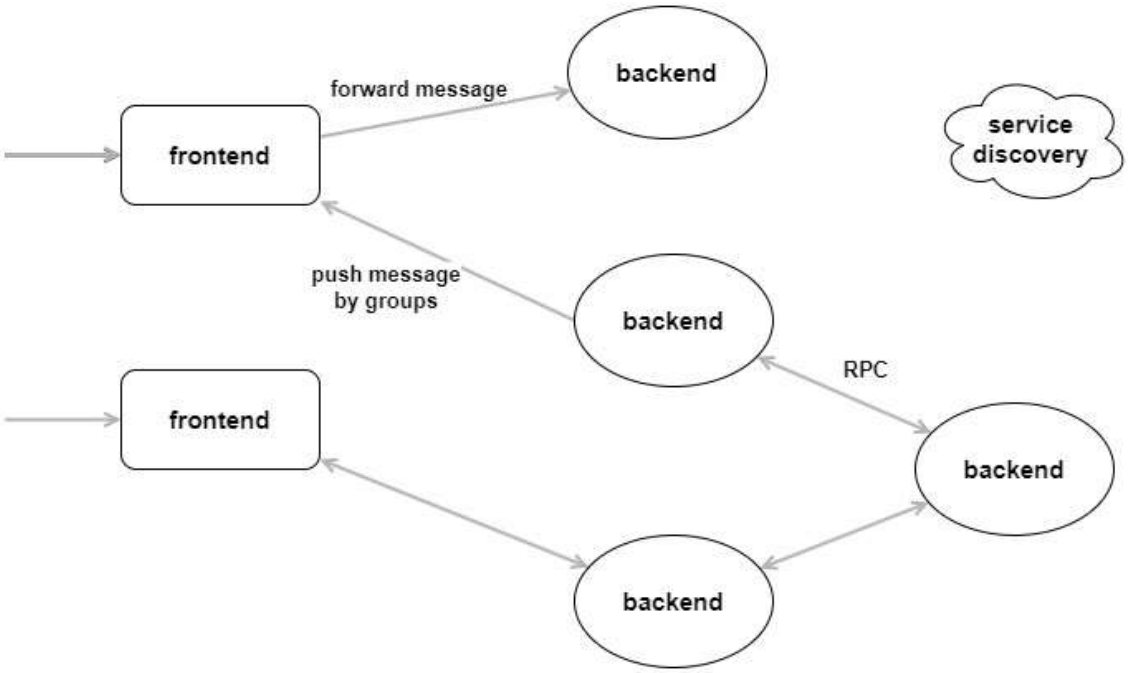


- Pitaya Framework



服务器（进程）的抽象与扩展介绍

> 该架构把游戏服务器做了抽象， 抽象成为两类： 前端服务器 和 后端服务器 ， 如图：



服务器职责介绍

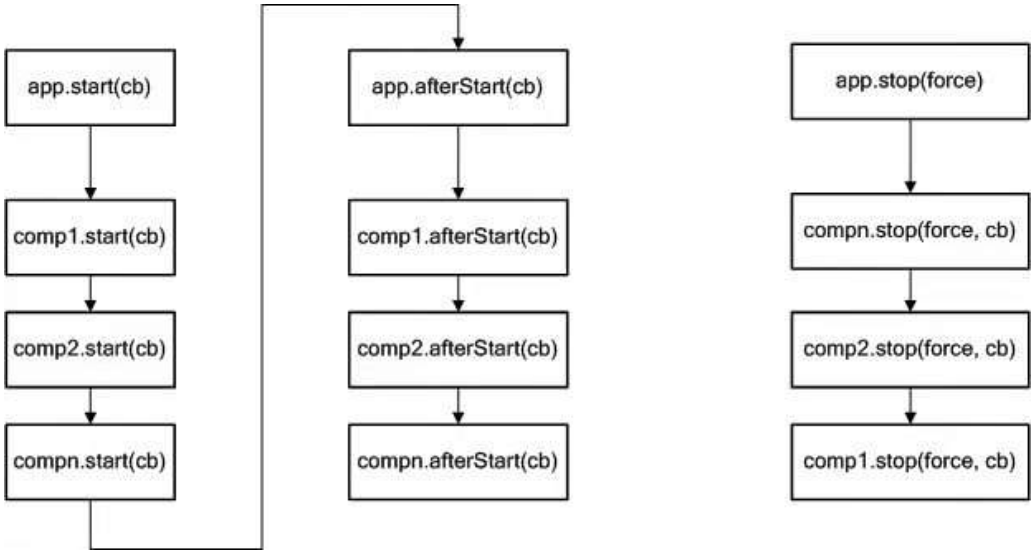
- 前端服务器(frontend)的职责：
  - 负责承载客户端请求的连接
  - 维护session信息
  - 把请求转发到后端
  - 把后端需要广播的消息发到前端
- 后端服务器(backend)的职责：
  - 处理业务逻辑， 包括RPC和前端请求的逻辑
  - 把消息推送回前端

服务器的鸭子类型

> Go语言的面向对象有个基本概念叫鸭子类型。 服务器的抽象也同样可以比喻为鸭子， 服务器的对外接口只有两类:
>
> - 一类是接收客户端的请求， 叫做 `handler`
> - 一类是接收RPC请求， 叫做 `remote`
>
> `handler` 和 `remote` 的行为决定了服务器长什么样子。 因此我们只要定义好 `handler` 和 `remote` 两类的行为， 就可以确定这个服务器的类型。

Component架构

> `component` 是pitaya自定义组件，开发者可自加载自定义的component。 以下是component的生命周期图：



> 注册组件

```
var app pitaya.Pitaya
...

room := services.NewRoom(app)

// 注册普通组件
// 可响应客户端请求: starx.request("room.join", {}, function (data){ });
app.Register(room,
    component.WithName("room"),
    component.WithNameFunc(strings.ToLower),
)

// 注册RPC组件
// 可响应服务端请求: m.app.RPC(ctx, "store.store.save", &reply, &args)
app.RegisterRemote(room,
    component.WithName("room"),
    component.WithNameFunc(strings.ToLower),
)
```

接口分析

- PlayerConn
  > `PlayerConn` 是一个封装的连接对象，继承net.Conn，并提供一个获取下一个数据包的方法

```
type PlayerConn interface {
    GetNextMessage() (b []byte, err error)
    net.Conn
}
```

- Acceptor
  > `Acceptor` 代表一个服务端端口进程，接收客户端连接，并用一个内部Chan来维护这些连接对象

```
type Acceptor interface {
    ListenAndServe()
    Stop()
    GetAddr() string
    GetConnChan() chan PlayerConn
}
```

- Acceptorwrapper

  > `Acceptorwrapper` 义如其名就是Acceptor的包装器，因为Acceptor的通过Chan来保存连接，所以wrapper可以通过遍历这个Chan来实时包装这些连接

```
type Wrapper interface {
    Wrap(acceptor.Acceptor) acceptor.Acceptor
}
```

- Agent

  > `Agent` 是一个服务端的应用层连接对象，包含了：
  >
  > - Session信息
  > - 服务器预发送消息队列
  > - 拆解包对象
  > - 最后心跳时间
  > - 停止发送心跳的chan
  > - 关闭发送数据的chan
  > - 全局的关闭信号
  > - 连接对象
  > - Agent当前状态
  > - ...

```
type (
    // Agent corresponds to a user and is used for storing raw Conn information
    Agent struct {
        Session            *session.Session  // session
        appDieChan         chan bool         // app die channel
        chDie              chan struct{}     // wait for close
        chSend             chan pendingWrite // push message queue
        chStopHeartbeat    chan struct{}     // stop heartbeats
        chStopWrite        chan struct{}     // stop writing messages
        closeMutex         sync.Mutex
        conn               net.Conn            // low-level conn fd
        decoder            codec.PacketDecoder // binary decoder
        encoder            codec.PacketEncoder // binary encoder
        heartbeatTimeout   time.Duration
        lastAt             int64 // last heartbeat unix time stamp
        messageEncoder     message.Encoder
        ... ...
        state              int32               // current agent state
    }

    pendingWrite struct {
        ctx  context.Context
        data []byte
        err  error
    }
)
```

- Component
  > `Component` 代表业务组件，提供若干个接口，通过Component生成处理请求的Service

```
type Component interface {
    Init()
    AfterInit()
    BeforeShutdown()
    Shutdown()
}
```

- Handler、Remote、Service
  - `Handler` 和 `Remote` 分别代表本地逻辑执行器和远程逻辑执行器
  - `Service` 是一组服务对象，包含若干Handler和Remote

  这里有个细节：`Receiver` `reflect.Value`

  设计者为了降低引用，采取在逻辑执行器中保留方法的Receiver以达到在Handler和Remote对象中，只需要保存类型的Method，而无需保存带对象引用的Value.Method

```
type (
    //Handler represents a message.Message's handler's meta information.
    Handler struct {
        Receiver    reflect.Value  // receiver of method
        Method      reflect.Method // method stub
        Type        reflect.Type   // low-level type of method
        IsRawArg    bool           // whether the data need to serialize
        MessageType message.Type   // handler allowed message type (either request or
notify)
    }

    //Remote represents remote's meta information.
    Remote struct {
        Receiver reflect.Value  // receiver of method
        Method   reflect.Method // method stub
        HasArgs  bool           // if remote has no args we won't try to serialize
received data into arguments
        Type     reflect.Type   // low-level type of method
    }

    // Service implements a specific service, some of it's methods will be
    // called when the correspond events is occurred.
    Service struct {
        Name     string              // name of service
        Type     reflect.Type        // type of the receiver
        Receiver reflect.Value       // receiver of methods for the service
        Handlers map[string]*Handler // registered methods
        Remotes  map[string]*Remote  // registered remote methods
        Options  options             // options
    }
)
```

- Modules

> Modules 模块和 Component 结构一致，唯一的区别在于使用上
>
> Modules 主要是面向系统的一些**全局存活的对象**，方便在统一的时机，集中进行启动和关闭

```
type Base struct{}

func (c *Base) Init() error {
    return nil
}

func (c *Base) AfterInit() {}

func (c *Base) BeforeShutdown() {}

func (c *Base) Shutdown() error {
    return nil
}
```

> 集中管理的对象容器在外部module.go中定义

```
var (
    modulesMap = make(map[string]interfaces.Module)
    modulesArr = []moduleWrapper{}
)

type moduleWrapper struct {
    module interfaces.Module
    name   string
}
```

- HandleService
  > `HandleService` 就是服务端的主逻辑对象，负责处理一切数据包
  >
  > - `chLocalProcess` 用于保存待处理的客户端数据包
  > - `chRemoteProcess` 用于保存待处理的来自其他服务器的数据包
  > - `services` 注册了处理客户端的服务
  > - 内部聚合一个 `RemoteService` 对象，专门负责处理服务器间的数据包

```
type (
    HandlerService struct {
        appDieChan         chan bool             // die channel app
        chLocalProcess     chan unhandledMessage // channel of messages that will be
processed locally
        chRemoteProcess    chan unhandledMessage // channel of messages that will be
processed remotely
        decoder            codec.PacketDecoder   // binary decoder
        encoder            codec.PacketEncoder   // binary encoder
        heartbeatTimeout   time.Duration
        messagesBufferSize int
        remoteService      *RemoteService
        serializer         serialize.Serializer          // message serializer
        server             *cluster.Server               // server obj
        services           map[string]*component.Service // all registered service
        messageEncoder     message.Encoder
        metricsReporters   []metrics.Reporter
    }

    unhandledMessage struct {
        ctx   context.Context
        agent *agent.Agent
        route *route.Route
        msg   *message.Message
    }
)
```

- RemoteService
  > `RemoteService` 中维护服务发现和注册提供的远程服务

```
type RemoteService struct {
    rpcServer              cluster.RPCServer
    serviceDiscovery       cluster.ServiceDiscovery
    serializer             serialize.Serializer
    encoder                codec.PacketEncoder
    rpcClient              cluster.RPCClient
    services               map[string]*component.Service // all registered service
    router                 *router.Router
    messageEncoder         message.Encoder
    server                 *cluster.Server // server obj
    remoteBindingListeners []cluster.RemoteBindingListener
}
```

- Timer

  > `Timer` 模块中维护一个全局定时任务管理者，使用线程安全的map来保存定时任务，通过time.Ticker的chan信号来定期触发

```
var (
    Manager = &struct {
        incrementID    int64
        timers         sync.Map
        ChClosingTimer chan int64
        ChCreatedTimer chan *Timer
    }{}

    Precision = time.Second

    GlobalTicker *time.Ticker
)
```

- pipeline

  > `pipeline` 模块提供全局钩子函数的配置
  >
  > - `BeforeHandler` 在业务逻辑之前执行
  > - `AfterHandler` 在业务逻辑之后执行

```
var (
    BeforeHandler = &pipelineChannel{}
    AfterHandler = &pipelineAfterChannel{}
)

type (
    HandlerTempl func(ctx context.Context, in interface{}) (out interface{}, err error)

    AfterHandlerTempl func(ctx context.Context, out interface{}, err error) (interface{},
error)

    pipelineChannel struct {
        Handlers []HandlerTempl
    }

    pipelineAfterChannel struct {
        Handlers []AfterHandlerTempl
    }
)
```

框架流程

- app.go 是系统启动的入口
  - 创建HandlerService
  - 并根据启动模式如果是集群模式创建RemoteService
  - 开启服务端事件监听
  - 开启监听服务器关闭信号的Chan

```
var (
    app = &App{
        ... ..
    }

    remoteService  *service.RemoteService
    handlerService *service.HandlerService
)
func Start() {
    ... ..
    if app.serverMode == Cluster {
        ... ..
        app.router.SetServiceDiscovery(app.serviceDiscovery)

        remoteService = service.NewRemoteService(
            app.rpcClient,
            app.rpcServer,
            app.serviceDiscovery,
            app.router,
            ... ..
        )

        app.rpcServer.SetPitayaServer(remoteService)

        initSysRemotes()
    }

    handlerService = service.NewHandlerService(
        app.dieChan,
        app.heartbeat,
        app.server,
        remoteService,
        ... ..
    )

    ... ..

    listen()
    ... ..
    // stop server
    select {
    case <-app.dieChan:
        logger.Log.Warn("the app will shutdown in a few seconds")
    case s := <-sg:
        logger.Log.Warn("got signal: ", s, ", shutting down...")
        close(app.dieChan)
    }
    ... ..
}
```

- `listen` 方法也就是开启服务，具体包括以下步骤：

> i. 注册Component
> ii. 注册定时任务的GlobalTicker
> iii. 开启Dispatch处理业务和定时任务（ticket）的goroutine
> iv. 开启acceptor处理连接的goroutine
> v. 开启主逻辑的goroutine
> vi. 注册Modules

```
func listen() {
    startupComponents()

    timer.GlobalTicker = time.NewTicker(timer.Precision)

    logger.Log.Infof("starting server %s:%s", app.server.Type, app.server.ID)
    for i := 0; i < app.config.GetInt("pitaya.concurrency.handler.dispatch"); i++ {
        go handlerService.Dispatch(i)
    }
    for _, acc := range app.acceptors {
        a := acc
        go func() {
            for conn := range a.GetConnChan() {
                go handlerService.Handle(conn)
            }
        }()

        go func() {
            a.ListenAndServe()
        }()

        logger.Log.Infof("listening with acceptor %s on addr %s", reflect.TypeOf(a),
a.GetAddr())
    }
    ... ..
    startModules()

    logger.Log.Info("all modules started!")

    app.running = true
}
```

> startupComponents对Component进行初始化
>
> 然后把Component注册到handlerService和remoteService上

```
func startupComponents() {
    // component initialize hooks
    for _, c := range handlerComp {
        c.comp.Init()
    }

    // component after initialize hooks
    for _, c := range handlerComp {
        c.comp.AfterInit()
    }

    // register all components
    for _, c := range handlerComp {
        if err := handlerService.Register(c.comp, c.opts); err != nil {
            logger.Log.Errorf("Failed to register handler: %s", err.Error())
        }
    }

    // register all remote components
    for _, c := range remoteComp {
        if remoteService == nil {
            logger.Log.Warn("registered a remote component but remoteService is not
running! skipping...")
        } else {
            if err := remoteService.Register(c.comp, c.opts); err != nil {
                logger.Log.Errorf("Failed to register remote: %s", err.Error())
            }
        }
    }
    ... ..
}
```

> 比如HandlerService的注册，反射得到component类型的全部方法，判断isHandlerMethod就加入services里面
>
> 并聚合Component对象的反射Value对象为全部Handler的Method Receiver，减少了对象引用

```go
func NewService(comp Component, opts []Option) *Service {
    s := &Service{
        Type:     reflect.TypeOf(comp),
        Receiver: reflect.ValueOf(comp),
    }
    ... ..
    return s
}

func (h *HandlerService) Register(comp component.Component, opts []component.Option) error
{
    s := component.NewService(comp, opts)
    ... ..
    if err := s.ExtractHandler(); err != nil {
        return err
    }

    h.services[s.Name] = s
    for name, handler := range s.Handlers {
        handlers[fmt.Sprintf("%s.%s", s.Name, name)] = handler
    }
    return nil
}
func (s *Service) ExtractHandler() error {
    typeName := reflect.Indirect(s.Receiver).Type().Name()
    ... ..
    s.Handlers = suitableHandlerMethods(s.Type, s.Options.nameFunc)
    ... ..
    for i := range s.Handlers {
        s.Handlers[i].Receiver = s.Receiver
    }
    return nil
}
func suitableHandlerMethods(typ reflect.Type, nameFunc func(string) string)
map[string]*Handler {
    methods := make(map[string]*Handler)
    for m := 0; m < typ.NumMethod(); m++ {
        method := typ.Method(m)
        mt := method.Type
        mn := method.Name
        if isHandlerMethod(method) {
            ... ..
            handler := &Handler{
                Method:      method,
                IsRawArg:    raw,
                MessageType: msgType,
            }
            ... ..
            methods[mn] = handler
        }
    }
```

```
    return methods
}
```

- `handlerService.Dispatch` 方法负责各种业务的处理，包括：
    i. 处理chLocalProcess中的本地Message
    ii. 使用remoteService处理chRemoteProcess中的远程Message
    iii. 在定时ticket到达时调用timer.Cron执行定时任务
    iv. 管理定时任务的创建
    v. 管理定时任务的删除

```
func (h *HandlerService) Dispatch(thread int) {
    defer timer.GlobalTicker.Stop()

    for {
        select {
        case lm := <-h.chLocalProcess:
            metrics.ReportMessageProcessDelayFromCtx(lm.ctx, h.metricsReporters, "local")
            h.localProcess(lm.ctx, lm.agent, lm.route, lm.msg)

        case rm := <-h.chRemoteProcess:
            metrics.ReportMessageProcessDelayFromCtx(rm.ctx, h.metricsReporters, "remote")
            h.remoteService.remoteProcess(rm.ctx, nil, rm.agent, rm.route, rm.msg)

        case <-timer.GlobalTicker.C: // execute cron task
            timer.Cron()

        case t := <-timer.Manager.ChCreatedTimer: // new Timers
            timer.AddTimer(t)

        case id := <-timer.Manager.ChClosingTimer: // closing Timers
            timer.RemoveTimer(id)
        }
    }
}
```

> 接下来看看 `Acceptor` 的工作，以下为Tcp实现，就是负责接收连接，流入acceptor的Chan

```
func (a *TCPAcceptor) ListenAndServe() {
    if a.hasTLSCertificates() {
        a.ListenAndServeTLS(a.certFile, a.keyFile)
        return
    }

    listener, err := net.Listen("tcp", a.addr)
    if err != nil {
        logger.Log.Fatalf("Failed to listen: %s", err.Error())
    }
    a.listener = listener
    a.running = true
    a.serve()
}
func (a *TCPAcceptor) serve() {
    defer a.Stop()
    for a.running {
        conn, err := a.listener.Accept()
        if err != nil {
            logger.Log.Errorf("Failed to accept TCP connection: %s", err.Error())
            continue
        }

        a.connChan <- &tcpPlayerConn{
            Conn: conn,
        }
    }
}
```

> 前面讲过对于每个Acceptor开启了一个goroutine去处理连接，也就是下面代码

```
for conn := range a.GetConnChan() {
    go handlerService.Handle(conn)
}
```

> 所以流入Chan的连接就会被实时的开启一个goroutine去处理，处理过程就是先创建一个Agent对象
>
> 并开启一个goroutine给Agent负责维护连接的心跳
>
> 然后开启死循环，读取连接的数据processPacket

```
func (h *HandlerService) Handle(conn acceptor.PlayerConn) {
    // create a client agent and startup write goroutine
    a := agent.NewAgent(conn, h.decoder, h.encoder, h.serializer, h.heartbeatTimeout,
h.messagesBufferSize, h.appDieChan, h.messageEncoder, h.metricsReporters)

    // startup agent goroutine
    go a.Handle()
    ... ..
    for {
        msg, err := conn.GetNextMessage()

        if err != nil {
            logger.Log.Errorf("Error reading next available message: %s", err.Error())
            return
        }

        packets, err := h.decoder.Decode(msg)
        if err != nil {
            logger.Log.Errorf("Failed to decode message: %s", err.Error())
            return
        }

        if len(packets) < 1 {
            logger.Log.Warnf("Read no packets, data: %v", msg)
            continue
        }

        // process all packet
        for i := range packets {
            if err := h.processPacket(a, packets[i]); err != nil {
                logger.Log.Errorf("Failed to process packet: %s", err.Error())
                return
            }
        }
    }
}
```

> 这时如果使用了pitaya提供的漏桶算法实现的限流wrap来包装acceptor，则会对客户端发送的消息进行限流限速
>
> 这里也是灵活利用for循环遍历chan的特性，所以也是实时地对连接进行包装

```
func (b *BaseWrapper) ListenAndServe() {
    go b.pipe()
    b.Acceptor.ListenAndServe()
}

// GetConnChan returns the wrapper conn chan
func (b *BaseWrapper) GetConnChan() chan acceptor.PlayerConn {
    return b.connChan
}

func (b *BaseWrapper) pipe() {
    for conn := range b.Acceptor.GetConnChan() {
        b.connChan <- b.wrapConn(conn)
    }
}
type RateLimitingWrapper struct {
    BaseWrapper
}

func NewRateLimitingWrapper(c *config.Config) *RateLimitingWrapper {
    r := &RateLimitingWrapper{}
    r.BaseWrapper = NewBaseWrapper(func(conn acceptor.PlayerConn) acceptor.PlayerConn {
        ... ..
        return NewRateLimiter(conn, limit, interval, forceDisable)
    })
    return r
}

func (r *RateLimitingWrapper) Wrap(a acceptor.Acceptor) acceptor.Acceptor {
    r.Acceptor = a
    return r
}

func (r *RateLimiter) GetNextMessage() (msg []byte, err error) {
    if r.forceDisable {
        return r.PlayerConn.GetNextMessage()
    }

    for {
        msg, err := r.PlayerConn.GetNextMessage()
        if err != nil {
            return nil, err
        }

        now := time.Now()
        if r.shouldRateLimit(now) {
            logger.Log.Errorf("Data=%s, Error=%s", msg, constants.ErrRateLimitExceeded)
            metrics.ReportExceededRateLimiting(pitaya.GetMetricsReporters())
            continue
        }

        return msg, err
```

```
    }
}
```

> processPacket对数据包解包后，执行processMessage

```
func (h *HandlerService) processPacket(a *agent.Agent, p *packet.Packet) error {
    switch p.Type {
    case packet.Handshake:
        ... ..
    case packet.HandshakeAck:
        ... ..
    case packet.Data:
        if a.GetStatus() < constants.StatusWorking {
            return fmt.Errorf("receive data on socket which is not yet ACK, session will
be closed immediately, remote=%s",
                a.RemoteAddr().String())
        }
        msg, err := message.Decode(p.Data)
        if err != nil {
            return err
        }
        h.processMessage(a, msg)
    case packet.Heartbeat:
        // expected
    }
    a.SetLastAt()
    return nil
}
```

> processMessage中包装数据包为unHandledMessage
>
> 根据消息类型，流入chLocalProcess 或者chRemoteProcess 也就转交给上面提到的负责Dispatch的goroutine去处理了

```
func (h *HandlerService) processMessage(a *agent.Agent, msg *message.Message) {
    requestID := uuid.New()
    ctx := pcontext.AddToPropagateCtx(context.Background(), constants.StartTimeKey,
time.Now().UnixNano())
    ctx = pcontext.AddToPropagateCtx(ctx, constants.RouteKey, msg.Route)
    ctx = pcontext.AddToPropagateCtx(ctx, constants.RequestIDKey, requestID.String())
    tags := opentracing.Tags{
        "local.id":   h.server.ID,
        "span.kind":  "server",
        "msg.type":   strings.ToLower(msg.Type.String()),
        "user.id":    a.Session.UID(),
        "request.id": requestID.String(),
    }
    ctx = tracing.StartSpan(ctx, msg.Route, tags)
    ctx = context.WithValue(ctx, constants.SessionCtxKey, a.Session)

    r, err := route.Decode(msg.Route)
    ... ..
    message := unhandledMessage{
        ctx:   ctx,
        agent: a,
        route: r,
        msg:   msg,
    }
    if r.SvType == h.server.Type {
        h.chLocalProcess <- message
    } else {
        if h.remoteService != nil {
            h.chRemoteProcess <- message
        } else {
            logger.Log.Warnf("request made to another server type but no remoteService
running")
        }
    }
}
```

> 服务器进程启动的最后一步是对全局模块启动
>
> 在外部的module.go文件中，提供了对module的全局注册方法、全部顺序启动方法、全部顺序关闭方法

```go
func RegisterModule(module interfaces.Module, name string) error {
    ... ..
}

func startModules() {
    for _, modWrapper := range modulesArr {
        modWrapper.module.Init()
    }
    for _, modWrapper := range modulesArr {
        modWrapper.module.AfterInit()
    }
}

func shutdownModules() {
    for i := len(modulesArr) - 1; i >= 0; i-- {
        modulesArr[i].module.BeforeShutdown()
    }

    for i := len(modulesArr) - 1; i >= 0; i-- {
        mod := modulesArr[i].module
        mod.Shutdown()
    }
}
```

处理细节

- localProcess

  > 接下来看看 `localprocess` 对于消息的处理细节(为了直观省略部分异常处理代码)
  >
  > 使用 `processHandlerMessagef` 方法对包装出来的 `ctx` 对象进行业务操作
  >
  > 最终根据消息的类型 `notify / Request` 区分是否需要响应，执行不同处理

```
func (h *HandlerService) localProcess(ctx context.Context, a *agent.Agent, route
*route.Route, msg *message.Message) {
    var mid uint
    switch msg.Type {
    case message.Request:
        mid = msg.ID
    case message.Notify:
        mid = 0
    }

    ret, err := processHandlerMessage(ctx, route, h.serializer, a.Session, msg.Data,
msg.Type, false)
    if msg.Type != message.Notify {
        ... ..
        err := a.Session.ResponseMID(ctx, mid, ret)
        ... ..
    } else {
        metrics.ReportTimingFromCtx(ctx, h.metricsReporters, handlerType, nil)
        tracing.FinishSpan(ctx, err)
    }
}
```

- processHandlerMessage

  > 这里面负进行业务逻辑
  >
  > 会先调用executeBeforePipeline(ctx, arg)，执行前置的钩子函数
  >
  > 再通过util.Pcall(h.Method, args)反射调用handler方法
  >
  > 再调用executeAfterPipeline(ctx, resp, err)，执行后置的钩子函数
  >
  > 最后调用serializeReturn(serializer, resp)，对请求结果进行序列化

```go
func processHandlerMessage(
    ctx context.Context,
    rt *route.Route,
    serializer serialize.Serializer,
    session *session.Session,
    data []byte,
    msgTypeIface interface{},
    remote bool,
) ([]byte, error) {
    if ctx == nil {
        ctx = context.Background()
    }
    ctx = context.WithValue(ctx, constants.SessionCtxKey, session)
    ctx = util.CtxWithDefaultLogger(ctx, rt.String(), session.UID())

    h, err := getHandler(rt)
    ... ..

    msgType, err := getMsgType(msgTypeIface)
    ... ..

    logger := ctx.Value(constants.LoggerCtxKey).(logger.Logger)
    exit, err := h.ValidateMessageType(msgType)
    ... ..

    arg, err := unmarshalHandlerArg(h, serializer, data)
    ... ..

    if arg, err = executeBeforePipeline(ctx, arg); err != nil {
        return nil, err
    }
    ... ..

    args := []reflect.Value{h.Receiver, reflect.ValueOf(ctx)}
    if arg != nil {
        args = append(args, reflect.ValueOf(arg))
    }

    resp, err := util.Pcall(h.Method, args)
    if remote && msgType == message.Notify {
        resp = []byte("ack")
    }

    resp, err = executeAfterPipeline(ctx, resp, err)
    ... ..

    ret, err := serializeReturn(serializer, resp)
    ... ..

    return ret, nil
}
```

- executeBeforePipeline
  > 实际就是执行pipeline的BeforeHandler

```
func executeBeforePipeline(ctx context.Context, data interface{}) (interface{}, error) {
    var err error
    res := data
    if len(pipeline.BeforeHandler.Handlers) > 0 {
        for _, h := range pipeline.BeforeHandler.Handlers {
            res, err = h(ctx, res)
            if err != nil {
                logger.Log.Debugf("pitaya/handler: broken pipeline: %s", err.Error())
                return res, err
            }
        }
    }
    return res, nil
}
```

- executeAfterPipeline
  > 实际就是执行pipeline的AfterHandler

```
func executeAfterPipeline(ctx context.Context, res interface{}, err error) (interface{},
error) {
    ret := res
    if len(pipeline.AfterHandler.Handlers) > 0 {
        for _, h := range pipeline.AfterHandler.Handlers {
            ret, err = h(ctx, ret, err)
        }
    }
    return ret, err
}
```

- util.pcall
  > `util.pcall` 里展示了golang反射的一种高级用法
  >
  > `method.Func.Call` ，第一个参数是 `Receiver` ，也就是调用对象方法的实例
  >
  > 这种设计对比直接保存Value对象的method，反射时直接call，拥有的额外好处就是降低了对象引用，方法不和实例绑定

```go
func Pcall(method reflect.Method, args []reflect.Value) (rets interface{}, err error) {
    ... ..
    r := method.Func.Call(args)
    if len(r) == 2 {
        if v := r[1].Interface(); v != nil {
            err = v.(error)
        } else if !r[0].IsNil() {
            rets = r[0].Interface()
        } else {
            err = constants.ErrReplyShouldBeNotNull
        }
    }
    return
}
```