





基于actor模型的Linux百万并发后台框架

Auth : 黄文龙

Mail : 769651718@qq.com

O r g : 能众软件

版本信息:

[illegible]

## 1 skynet介绍

- 1.1 简介
- 1.2 特点
- 1.3 Actor模型
  - 1.3.1 Actor模型介绍
  - 1.3.2 Actor模型好处

## 2 在ubuntu上搭建skynet

- 2.1 获取skynet源代码
- 2.3 skynet代码目录结构
- 2.4 编译与运行skynet服务器
- 2.5 运行客户端

## 3 构建服务的基础API

- 3.1 编写一个test服务
- 3.2 另外一种启动服务的方式
- 3.3 环境变量
- 3.4 skynet.init的使用

## 4 服务类型

- 4.1 普通服务
- 4.2 全局唯一服务
  - 4.2.1 测试skynet.uniqueservice接口
  - 4.2.2 测试skynet.queryservice接口
- 4.3 多节点中的全局服务
  - 4.3.1 启动两个skynet节点
  - 4.3.2 测试全节点全局唯一服
  - 4.3.3 本地全局唯一服与全节点全局唯一服区别

## 5 服务别名

- 5.1 本地别名与全局别名
- 5.2 别名注册与查询接口
- 5.3 给服务注册别名
  - 5.3.1 给普通服取别名
  - 5.3.2 全局别名查询阻塞
  - 5.3.3 多节点中的全局别名
  - 5.3.4 杀死带别名的服务
- 5.4 全局别名与全局唯一服名区别

## 6 服务调度

- 6.1 使用sleep休眠
- 6.2 在服务中开启新的线程
- 6.3 长时间占用执行权限的任务
- 6.4 使用skynet.yield让出执行权
- 6.5 线程间的简单同步
- 6.6 定时器的使用
  - 6.6.1 启动一个定时器
  - 6.6.2 skynet.start源码分析
  - 6.6.3 循环启动定时器
- 6.7 获取时间
- 6.8 错误处理

## 7 服务间消息通信

- 7.1消息类型
- 7.2 注册消息处理函数
- 7.3 打包与解包消息
- 7.4 发送消息的方法
  - 7.4.1 发送无需响应的消息
  - 7.4.2 发送必须响应的消息
- 7.5 响应消息的方法
- 7.6 lua消息收发综合应用
- 7.7 session的意义
- 7.8 使用skynet.response响应消息
- 7.9 skynet.call失败的情况
- 7.10 服务重入问题
- 7.11 服务临界区
- 7.12 注册其他消息
- 7.13 代理服务
  - 7.13.1 简单服务代理
  - 7.13.2 转换协议实现代理
- 7.14 伪造消息
- 7.15 节点间消息通信

## 8 Multicast组播

- 8.1 Multicast介绍
- 8.2 应用组播技术
- 8.3 组播原理

## 9 socket网络服务

- 9.1 skynet.socket 常用api
- 9.2 写一个skynet TCP监听端
- 9.3 socket.readline使用
- 9.4 socket.readall的使用
- 9.5 低优先级的发送函数
- 9.6 skynet的socket代理服务
- 9.7 转交socket控制权
- 9.8 skynet的TCP主动连接端
- 9.9 同时发送多个请求

## 10 socketChannel

- 10.1 第一种模式的socketChannel
- 10.2 第二种模式的socketChannel

## 11 域名查询

- 11.1 DNS简单使用
- 11.2 封装一个DNS服务

## 12 snax框架

- 12.1 snax服务基础API
- 12.2 最简单snax服务
- 12.3 snax服务请求
  - 12.3.1 snax处理无响应请求
  - 12.3.2 处理有响应请求
- 12.4 snax全局唯一服
- 12.5 snax服务热更
  - 12.5.1 函数patch
  - 12.5.2 local变量patch
  - 12.5.3 修改snax服务线上状态

## 13 网关服务

- 13.1最简单网关服务

- 13.1.1 编写mygateserver.lua
- 13.1.2 启动mygateserver
- 13.2 gateserver应用协议
  - 13.2.1 两字数据长度协议
  - 13.2.2 打包与解包
  - 13.2.3 client使用长度协议发包
  - 13.2.4 gateserver解包
- 13.3 控制客户端连接数
- 13.4 gateserver其他回调函数
- 13.5 给gateserver发送lua消息
- 13.6 open与close两个lua消息
- 13.7 agent服务
- 13.8 skynet自带网关服务

## 14 登录服务

- 14.1 加密算法
  - 14.1.1 DHexchange密钥交换算法
  - 14.1.2 随机数
  - 14.1.3 hmac64哈希算法
  - 14.1.4 base64编解码
  - 14.1.5 DES加解密
  - 14.1.6 hashkey算法
- 14.2 loginserver原理
- 14.3 loginserver 模板
- 14.4 运用loginserver模板
- 14.5 账户核对失败的处理
- 14.6 login\_handler错误处理
- 14.7 登录重入报错
- 14.6 密钥交换失败

## 15 msgserver

- 15.1 msgserver
  - 15.1.1 msgserver api
  - 15.1.2 msgserver服务模板
  - 15.1.3 最简单msgserver
  - 15.1.4 发送lua消息启动msgserver
  - 15.1.5 发送lua消息给msgserver
- 15.2 loginserver与msgserver
  - 15.2.1 编写一个mymsgserver.lua
  - 15.2.2 编写一个mylogin.lua
  - 15.2.3 编写一个testmsgserver.lua来启动他们
  - 15.2.4 编写一个myclient.lua
  - 15.2.5 运行服务与客户端
- 15.3 服务握手应答包
  - 15.3.1 404用户未找到
  - 15.3.2 403 index已过期
  - 15.3.3 断线重连
- 15.4 请求与应答
  - 15.4.1 获取最后一次返回
  - 15.4.2 获取历史应答
  - 15.4.3 服务应答异常
- 15.5 agent服务

## 16 mysql

- 16.1 连接mysql
- 16.2 执行SQL语句

16.3 db:query的调度情况

## 17 protobuf

17.1 安装PBC

17.2 生成protobuf文件

17.3 使用protobuf文件

17.4 编写一个稍微复杂点pbc服务

17.5 protobuf数据类型

## 18 http协议的服务

18.1 http服务端

18.2 http客户端

## 19 ShareData

19.1 shareData API

# 1 skynet介绍

---

Skynet 是一个基于C跟lua的开源服务端并发框架，这个框架是单进程多线程Actor模型。是一个轻量级的为在线游戏服务器打造的框架。但从社区 Community 的反馈结果看，它也不仅仅使用在游戏服务器领域。也可以作为web服务器，负载能力与Nginx不分上下，skynet相对来说跟轻量一些。

使用skynet作为后端实现的游戏有：

- [天天来战](#)
- [陌陌弹珠](#)
- [心动庄园](#)
- [战神黎明](#)
- [麻将游戏服务器](#)

- [枪战英雄\(全民枪神,终极枪王\)](#)
- [逍遥西游](#)
- [仙语](#)
- [战斗方块](#)
- [战场双马尾](#)
- [六界](#)
- [永恒手游](#)
- [秘境物语](#)
- [美食战争](#)
- [钢炼之魂](#)
- [海龙王](#)
- [消消果缤纷](#)
- [狩猎场Online](#)
- [sailcraft 策略游戏, 全球同服](#)
- [武器之王](#)

## 1.1 简介

---

这个系统是单进程多线程模型。

每个服务都是严格的被动的消息驱动的，以一个统一的 callback 函数的形式交给框架。框架从消息队列里调度出接收的服务模块，找到 callback 函数入口，调用它。服务本身在没有被调度时，是不占用任何 CPU 的。

skynet虽然支持集群，但是作者云风主张能用一个节点完成尽量用一个节点，因为多节点通信方面的开销太大，如果一共有 100 个 skynet 节点，在它们启动完毕后，会建立起 9900条通讯通道。



## 1.2 特点

---

Skynet框架做两个必要的保证：

- 一、一个服务的 callback 函数永远不会被并发。
- 二、一个服务向另一个服务发送的消息的次序是严格保证的。

用多线程模型来实现它。底层有一个线程消息队列，消息由三部分构成：源地址、目的地址、以及数据块。框架启动固定的多条线程，每条工作线程不断从消息队列取到消息，调用服务的 callback 函数。

线程数应该略大于系统的 CPU 核数，以防止系统饥饿。（只要服务不直接给自己不断发新的消息，就不会有服务被饿死）

对于目前的点对点消息，要求发送者调用 malloc 分配出消息携带数据用到的内存；由接受方处理完后调用 free 清理（由框架来做）。这样数据传递就不需要有额外的拷贝了。

做为核心功能，Skynet 仅解决一个问题：

把一个符合规范的 C 模块，从动态库（so 文件）中启动起来，绑定一个永不重复（即使模块退出）的数字 id 做为其 handle。模块被称为服务（Service），服务间可以自由发送消息。每个模块可以向 Skynet 框架注册一个 callback 函数，用来接收发给它的消息。每个服务都是被一个个消息包驱动，当没有包到来的时候，它们就会处于挂起状态，对 CPU 资源零消耗。如果需要自主逻辑，则可以利用 Skynet 系统提供的 timeout 消息，定期触发。

## 1.3 Actor模型

---

### 1.3.1 Actor模型介绍

**Actor模型内部的状态由它自己维护即它内部数据只能由它自己修改(通过消息传递来进行状态修改)，所以使用 Actors模型进行并发编程可以很好地避免这些问题，Actor由状态(state)、行为(Behavior)和邮箱(mailBox)三部分组成**

1. 状态(state)：Actor中的状态指的是Actor对象的变量信息，状态由Actor自己管理，避免了并发环境下的锁和内存原子性等问题
2. 行为(Behavior)：行为指定的是Actor中计算逻辑，通过Actor接收到消息来改变Actor的状态

3. 邮箱(mailBox): 邮箱是Actor和Actor之间的通信桥梁, 邮箱内部通过FIFO消息队列来存储发送方Actor消息, 接受方Actor从邮箱队列中获取消息

Actor的基础就是消息传递, skynet中每个服务就是一个LUA虚拟机, 就是一个Actor。

### 1.3.2 Actor模型好处

1. 事件模型驱动: Actor之间的通信是异步的, 即使Actor在发送消息后也无需阻塞或者等待就能够处理其他事情。
2. 强隔离性: Actor中的方法不能由外部直接调用, 所有的一切都是通过消息传递进行的, 从而避免了Actor之间的数据共享, 想要观察到另一个Actor的状态变化只能通过消息传递进行询问。
3. 位置透明: 无论Actor地址是在本地还是在远程机上对于代码来说都是一样的。
4. 轻量性: Actor是非常轻量的计算单元, 只需少量内存就能达到高并发。

## 2 在ubuntu上搭建skynet

---

### 2.1 获取skynet源代码

---

## 1. 安装git代码管理工具

```
$ sudo apt-get update
$ sudo apt-get install git
```

注意：如果安装失败，请先安装一下只支持库

```
$ sudo apt-get install build-essential libssl-dev libcurl4-gnutls-dev libexpat1-dev
gettext unzip
```

2. 到github上面下载skynet的源代码 skynet的代码保存在[github](https://github.com/cloudwu/skynet)上面，大家可以去上面查看，现在我们用git把代码拷贝一份下来：

```
$ git clone https://github.com/cloudwu/skynet.git
```

## 2.3 skynet代码目录结构

skynet\_root\_dir

3rd	#第三方支持库，包括LUA虚拟机，jmalloc等
lua-lib	#lua语言封装的常用库，包括http、md5
lua-lib-src	#将c语言实现的插件捆绑成lua库，例如数据库驱动、bson、加密算法等
service	#使用lua写的Skynet的服务模块
service-src	#使用C写的Skynet的服务模块
skynet-src	#skynet核心代码目录
test	#使用lua写的一些测试代码
examples	#示例代码
Makefile	#编译规则文件，用于编译
platform.mk	#编译与平台相关的设置

## 2.4 编译与运行skynet服务器

### 1. 编译skynet

```
$ cd skynet #今后我们所有的工作都在这个目录中进行
$ make linux
```

```
#如果报错:
./autogen.sh: 5: ./autogen.sh: autoconf: not found
#安装autoconf
$ sudo apt-get install autoconf
```

```
#如果报错:
lua.c:83:31: fatal error: readline/readline.h: No such file or directory
#安装libreadline-dev
$ sudo apt-get install libreadline-dev
```

```
#编译成功出现以下提示
make[1]: Leaving directory '/home/ubuntu/workspace/skynet'
#并且在目录里出现一个可执行文件skynet
```

### 2. 运行第一个skynet节点

```
#启动一个skynet服务节点
$ ./skynet examples/config
```

## 2.5 运行客户端

我们要运行的客户端是example/client.lua 这个lua脚本文件，那么首先你要有一个lua虚拟机程序。

### 1. 编译lua虚拟机

```
#打开另一个终端，开始编译虚拟机
$ cd ./3rd/lua/
$ make linux
#编译成功则会在当前路径上面看到一个可执行文件lua
```

## 2. 运行客户端

```
#跑到skynet根目录
$ cd ../../
#运行client.lua这个脚本
$ ./3rd/lua/lua examples/client.lua
```

# 3 构建服务的基础API

```
local skynet = require "skynet"
--打印函数
skynet.error(...)
--用 func 函数初始化服务，并将消息处理函数注册到 C 层，让该服务可以工作。
skynet.start(func)
--若服务尚未初始化完成，则注册一个函数等服务初始化阶段再执行；若服务已经初始化完成，则立刻运行该函数。
skynet.init(func)
--结束当前服务
skynet.exit()
--获取当前服务的句柄handler
skynet.self()
--将handle转换成字符串
skynet.address(handler)
```

在配置文件中可以查看模块所在路径  
可以找到skynet并查看函数原型

把信息交给logger服务打印  
而不要用print函数打印，多线程模型，logger和print都用了标准输出的话可能产生冲突

每个服务都有唯一的16进制句柄

把整形转换成更好看的字符串形式

--退出skynet进程

`require "skynet.manager"` --除了需要引入skynet包以外还要再引入skynet.manager包。

`skynet.abort()` 安全地退出Skynet，而不是使用Ctrl+C  
停机使用代码更新的时候使用

--强制杀死其他服务

`skynet.kill(address)` --可以用来强制关闭别的服务。但强烈不推荐这样做。因为对象会在任意一条消息处理完毕后，毫无征兆的退出。所以推荐的做法是，发送一条消息，让对方自己善后以及调用 `skynet.exit` 。注：

`skynet.kill(skynet.self())` 不完全等价于 `skynet.exit()` ，后者更安全。

## 3.1 编写一个test服务

### 1. 编写一个最简单的服务test.lua

```
--引入或者说是创建一个skynet服务
local skynet = require "skynet"
--调用skynet.start接口，并定义传入回调函数
skynet.start(function()
    skynet.error("Server First Test")
end)
```

### 2. 修改exmaple/config文件中的start的值为test，表示启动test.lua，修改之前请备份

```
include "config.path"
-- preload = "./examples/preload.lua" -- run preload.lua before every lua service run
thread = 2
logger = nil
logpath = "."
harbor = 1
address = "127.0.0.1:2526"
master = "127.0.0.1:2013"
start = "test" -- main script --将start的值修改为test
bootstrap = "snlua bootstrap" -- The service for bootstrap
standalone = "0.0.0.0:2013"
-- snax_interface_g = "snax_g"
cpath = root.."cservice/?.so"
```

```
-- daemon = "./skynet.pid"
```

### 3. 通过skynet来运行test.lua

```
$ ./skynet examples/config
```

运行结果

```
$ ./skynet examples/config
[:01000001] LAUNCH logger
[:01000002] LAUNCH snlua bootstrap
[:01000003] LAUNCH snlua launcher
[:01000004] LAUNCH snlua cmaster
[:01000004] master listen socket 0.0.0.0:2013
[:01000005] LAUNCH snlua cslave
[:01000005] slave connect to master 127.0.0.1:2013
[:01000004] connect from 127.0.0.1:52132 4
[:01000006] LAUNCH harbor 1 16777221
[:01000004] Harbor 1 (fd=4) report 127.0.0.1:2526
[:01000005] Waiting for 0 harbors
[:01000005] Shakehand ready
[:01000007] LAUNCH snlua datacenterd
[:01000008] LAUNCH snlua service_mgr
[:01000009] LAUNCH snlua test
[:01000009] Server First Test
[:01000002] KILL self
```

注意：千万不要在skynet根目录以外的地方执行skynet，例如：

```
$ cd examples
$ ../skynet config
try open logger failed : ./cservice/logger.so: cannot open shared object file: No such file
or directory
Can't launch logger service
$
```

以上出现找不到logger.so的情况，其实不仅仅是这个模块找不到，所有的模块都找不到了，因为在config包含的路劲conf.path中，所有的模块路劲的引入全部依靠着相对路劲。一旦执行skynet程序的位置不一样了，相对路劲也会不一样。

```

root = "./"
luaservice =
root.."service/? .lua;"..root.."test/? .lua;"..root.."examples/? .lua;"..root.."test/?/init.lua"

lua_loader = root .. "lua/lib/loader.lua"
lua_path = root.."lua/lib/? .lua;"..root.."lua/lib/?/init.lua"
lua_cpath = root .. "lua/lib/? .so"
snax = root.."examples/? .lua;"..root.."test/? .lua"

```

← 当前路劲

#### 4. 添加自己的LUA脚本路劲

例如：添加my\_workspace目录，则只需在luaservice值的基础上再添加一个

root.."my\_workspace/? .lua;"，注意：各个路劲通过一个； 隔开。

```

my_workspace
luaservice =
root.."service/? .lua;"..root.."test/? .lua;"..root.."examples/? .lua;"..root.."test/?/init.lua"
;.."my_workspace/? .lua"
lua_loader = root .. "lua/lib/loader.lua"
lua_path = root.."lua/lib/? .lua;"..root.."lua/lib/?/init.lua"
lua_cpath = root .. "lua/lib/? .so"
snax = root.."examples/? .lua;"..root.."test/? .lua"

```

← 添加路劲

把我们的刚才写的test.lua丢到my\_workspace中

```
$ mv examples/test.lua my_workspace/
```

顺便将example下的config以及conf.path也拷贝一份到my\_workspace

```
$ cp examples/config my_workspace/
$ cp examples/config.path my_workspace/
```

这次运行的时候就可以这样了：

```
$ ./skynet my_workspace/config
```

## 3.2 另外一个种启动服务的方式



另一种方式启动想要的服务，可以在main.lua运行后，在console直接输入需要启动的服务名。

1. 先启动main.lua服务，注意还原examples/config默认配置，并且在example/config.path添加自己的服务目录。

```
$ ./skynet examples/config #config中start配置为启动main.lua，也可以直接配置console.lua
[:01000001] LAUNCH logger
[:01000002] LAUNCH snlua bootstrap
[:01000003] LAUNCH snlua launcher
[:01000004] LAUNCH snlua cmaster
[:01000004] master listen socket 0.0.0.0:2013
[:01000005] LAUNCH snlua cslave
[:01000005] slave connect to master 127.0.0.1:2013
[:01000004] connect from 127.0.0.1:52698 4
[:01000006] LAUNCH harbor 1 16777221
[:01000004] Harbor 1 (fd=4) report 127.0.0.1:2526
[:01000005] Waiting for 0 harbors
[:01000005] Shakehand ready
[:01000007] LAUNCH snlua datacenterd
[:01000008] LAUNCH snlua service_mgr
[:01000009] LAUNCH snlua main
[:01000009] Server start
[:0100000a] LAUNCH snlua protoloader
[:0100000b] LAUNCH snlua console
[:0100000c] LAUNCH snlua debug_console 8000
[:0100000c] Start debug console at 127.0.0.1:8000
[:0100000d] LAUNCH snlua simpledb
[:0100000e] LAUNCH snlua watchdog
[:0100000f] LAUNCH snlua gate
[:0100000f] Listen on 0.0.0.0:8888
[:01000009] Watchdog listen on 8888
[:01000009] KILL self
[:01000002] KILL self
```

2. 在启动的main服务中，直接输入test，回车

```
$ ./skynet examples/config
[:01000001] LAUNCH logger
[:01000002] LAUNCH snlua bootstrap
[:01000003] LAUNCH snlua launcher
[:01000004] LAUNCH snlua cmaster
[:01000004] master listen socket 0.0.0.0:2013
[:01000005] LAUNCH snlua cslave
[:01000005] slave connect to master 127.0.0.1:2013
[:01000004] connect from 127.0.0.1:52698 4
[:01000006] LAUNCH harbor 1 16777221
[:01000004] Harbor 1 (fd=4) report 127.0.0.1:2526
[:01000005] Waiting for 0 harbors
[:01000005] Shakehand ready
[:01000007] LAUNCH snlua datacenterd
[:01000008] LAUNCH snlua service_mgr
```

```

[:01000009] LAUNCH snlua main
[:01000009] Server start
[:0100000a] LAUNCH snlua protoloader
[:0100000b] LAUNCH snlua console
[:0100000c] LAUNCH snlua debug_console 8000
[:0100000c] Start debug console at 127.0.0.1:8000
[:0100000d] LAUNCH snlua simpladb
[:0100000e] LAUNCH snlua watchdog
[:0100000f] LAUNCH snlua gate
[:0100000f] Listen on 0.0.0.0:8888
[:01000009] Watchdog listen on 8888
[:01000009] KILL self
[:01000002] KILL self
test #终端输入
[:01000010] LAUNCH snlua test
[:01000010] Server First Test #服务已经启动

```

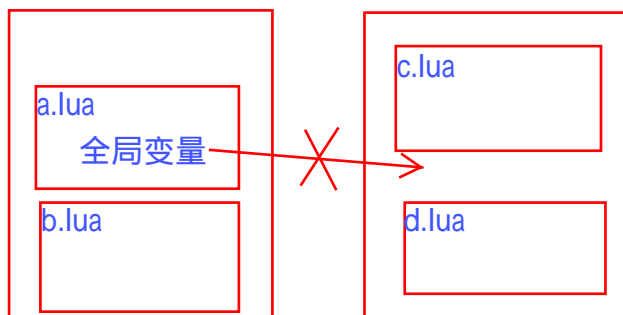
Skynet

这是一个专门保存环境变量的虚拟机

环境变量：Skynet中所有虚拟机都能访问

虚拟机1

虚拟机2



lua的全局变量，同一个虚拟机的问价可以访问  
不同虚拟机就不行  
这里的虚拟机相当于一个actor  
少量的环境变量可以实现服务之间的通信

### 3.3 环境变量

--获取环境变量

skynet.getenv(varName)

--设置环境变量，不能设置已经存的环境变量

skynet.setenv(varName, varValue)

不是linux的环境变量（echo env）

而是skynet中example/config配置文件  
中定义的环境变量

- 1、预先加载的环境变量是在config中配置的，加载完成后，所有的service都能去获取这些变量。
- 2、也可以去设置环境变量，但是不能修改已经存在的环境变量。
- 3、环境变量设置完成后，当前节点上的所有服务都能访问的到。
- 4、环境变量设置完成后，即使服务退出了，环境变量依然存在，所以不要滥用环境变量。

例如在config中添加：

设置环境变量的服务退出了

```
myname = "Dmaker"  
myage = 20
```

示例代码: testenv.lua

```
local skynet = require "skynet"  
  
skynet.start(function()  
    --获取环境变量myname和myage的值, 成功返回其值, 如果该环境变量不存在返回nil  
    local name = skynet.getenv("myname")  
    local age = skynet.getenv("myage")  
    skynet.error("My name is", name, ",", age, "years old.")  
  
    --skynet.setenv("myname", "coder")  --不要尝试设置已经存在的变量值, 会报错  
    --skynet.setenv("myage", 21)  
  
    skynet.setenv("mynewname", "coder") --设置一个新的变量  
    skynet.setenv("mynewage", 21)  
  
    name = skynet.getenv("mynewname")  
    age = skynet.getenv("mynewage")  
    skynet.error("My new name is", name, ",", age, "years old soon.")  
    skynet.exit()  
end)
```

其他服务也能获取环境变量, 例如再启动一个test2.lua:

```
local skynet = require "skynet"  
  
skynet.start(function()  
    local name = skynet.getenv("mynewname")  
    local age = skynet.getenv("mynewage")  
    skynet.error("My new name is", name, ",", age, "years old soon.")  
end)
```

先运行test再运行test2

```
$ ./skynet examples/config  
test #终端输入  
[:0100000a] LAUNCH snlua test  
[:0100000a] My name is Dmaker , 20 years old.  
[:0100000a] My new name is coder , 21 years old soon.  
[:0100000a] KILL self  
test2 #终端输入  
[:0100000b] LAUNCH snlua test2  
[:0100000b] My new name is coder , 21 years old soon.
```

总结：环境变量只能新增加无法修改。

## 3.4 skynet.init的使用

skynet.init用来注册服务初始化之前，需要执行的函数。也就是在skynet.start之前运行。

示例代码：testinit.lua

```
local skynet = require "skynet"
```

```
skynet.init(function()
    skynet.error("service init")
end)
```

服务开始之前做一些初始化工作

```
skynet.start(function()
    skynet.error("service start")
end)
```

服务开始

这两个注册函数start和init，只是负责把函数注册到C层，function的运行并不是按顺序

function的运行顺序是由框架决定的，把init代码写到start后面，运行skynet时仍然是init先于start运行

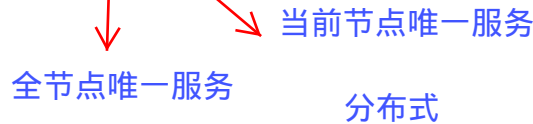
运行结果：

```
[ :01000009] service init    #先运行skynet.init
[ :01000009] service start    #运行skynet.start函数
```

## 4 服务类型

比如kira\_first可以在控制台启动多个

skynet中的服务分为普通服务与全局唯一服务。第3节启动方式就是一个普通服务，而全局唯一服务顾名思义就是在skynet中只能生成一个服务实例。



### 4.1 普通服务

命令行启动服务本质上就是调用“创建接口”

每调用一次创建接口就会创建出一个对应的服务实例，可以同时创建成千上万个，用唯一的id来区分每个服务实例。使用的创建接口是：

```
--[[
```

1. 用于启动一个新的 Lua 服务, luaServerName是脚本的名字 (不用写 .lua 后缀)。

2. 只有被启动的脚本的 start 函数返回后, 这个 API 才会返回启动的服务的地址, 这是一个阻塞 API。

3. 如果被启动的脚本在初始化环节抛出异常, skynet.newservice 也会执行失败。

4. 如果被启动的脚本的 start 函数是一个永不结束的循环, 那么 newservice 也会被永远阻塞住。

```
--]]
```

```
skynet.newservice(luaServerName, ...)
```

可变参数，是给脚本传递的参数（启动脚本后面追加参数）

start函数里面注册的  
function执行完才返回

#### 1. 启动一个test服务

testnewservice.lua

```
local skynet = require "skynet"
```

```
--调用skynet.start接口，并定义传入回调函数
```

```
skynet.start(function()
```

```
    skynet.error("My new service")
```

```
    skynet.newservice("test")
```

```
    skynet.error("new test service")
```

```
end)
```

启动服务函数的返回值是服务的句柄

启动main.lua，并且输入testnewservice

```
$ ./skynet examples/config #默认启动main.lua服务
testnewservice
[:01000010] LAUNCH snlua testnewservice #通过main服务, 启动testnewservice服务
[:01000010] My new service
[:01000012] LAUNCH snlua test #再启动test服务
[:01000012] Server First Test
[:01000010] new test service
```

## 2. 启动两个test服务

testnewservice2.lua

```
local skynet = require "skynet"

--调用skynet.start接口, 并定义传入回调函数
skynet.start(function()
    skynet.error("My new service")
    skynet.newservice("test") ←
    skynet.error("new test service 0")
    skynet.newservice("test") ←
    skynet.error("new test service 1")
end)
```

启动main.lua, 并且输入testnewservice

```
$ ./skynet examples/config #默认启动main.lua服务
testnewservice #终端输入
[:01000010] LAUNCH snlua testnewservice #通过main服务, 启动testnewservice服务
[:01000010] My new service
[:01000012] LAUNCH snlua test #启动一个test服务
[:01000012] Server First Test
[:01000010] new test service 0
[:01000019] LAUNCH snlua test #再启动一个test服务
[:01000019] Server First Test
[:01000010] new test service 1
```

传递参数：

命令行：

test arg1 arg2 arg3

标准输入把参数都识别为字符串

skynet.newservice的参数

skynet.newservice("test", "1", "false", "Tom") --参数类型必须是string

使用参数：

```
local skynet = require "skynet"
```

```
local args = {...}
```

```
skynet.start(function()
```

```
    for _, v in ipairs(args) do
```

```
        skynet.error(v, type(v))
```

```
    end
```

```
end)
```

## 4.2 全局唯一服务

### 返回唯一服务的句柄

全局唯一的服务等同于单例，即不管调用多少次创建接口，最后都只会创建一个此类型的服务实例，且全局唯一。

- 创建接口：

#### 返回服务句柄

```
skynet.uniqueservice(servicename, ...)    当前节点唯一
skynet.uniqueservice(true, servicename, ...) 全局节点唯一
```

当带参数 `true` 时，则表示此服务在所有节点之间是唯一的。第一次创建唯一服务，返回服务地址，第二次创建的时候不会正常创建服务，只是返回第一次创建的服务地址。

- 查询接口：假如不清楚当前创建了此全局服务没有，可以通过以下接口来查询：

```
skynet.queryservice(servicename, ...)
skynet.queryservice(true, servicename, ...)
```

如果还没有创建过目标服务则一直等下去，直到目标服务被(其他服务触发)创建。

当参数带 `true` 时，则表示查询在所有节点中唯一的的服务是否存在。

## 4.2.1 测试skynet.uniqueservice接口

示例：testunique.lua

```
local skynet = require "skynet"

local args = { ... }
if(#args == 0) then
    table.insert(args, "uniqueservice")
end

skynet.start(function()
    local us
    skynet.error("test unique service")
    if ( #args == 2 and args[1] == "true" ) then
        us = skynet.uniqueservice(true, args[2])
    else
        us =skynet.uniqueservice(args[1])
    end
    skynet.error("uniqueservice handler:", skynet.address(us))
end)
```

参数是字符串

通过参数决定，是全局唯一服务，还是当前节点唯一服务

示例：uniqueservice.lua

同一个服务  
可以创建一个当前节点唯一的  
可以再创建一个全局节点唯一的  
两个服务不冲突

```
local skynet = require "skynet"
```

```
skynet.start(function()
```

```
    skynet.error("Server First Test")
```

```
    --skynet.exit() 不要尝试服务初始化阶段退出服务，唯一服会创建失败
```

```
end)
```

如果作为全局唯一服务，不能再服务初始化阶段退出  
唯一的服务刚启动就退出，这种服务没有意义  
要保证全局唯一服务在全局一直存在能够被使用

运行结果：

```
$ ./skynet examples/config
```

```
testunique #终端输入
```

```
[ :01000010 ] LAUNCH snlua testunique
```

```
[ :01000010 ] test unique service
```

```
[ :01000012 ] LAUNCH snlua uniqueservice #第一次创建全局唯一服uniqueservice成功
```

```
[ :01000012 ] unique service start
```

```
[ :01000010 ] uniqueservice handler: :01000012
```

```
testunique #终端输入
```

```
[ :01000019 ] LAUNCH snlua testunique
```

```
[ :01000019 ] test unique service
```

```
[ :01000019 ] uniqueservice handler: :01000012 #第二次创建并没有创建全局唯一服
```

## 4.2.2 测试skynet.queryservice接口

只能查询节点唯一服务和全局唯一服务  
(不唯一的服务会有多个句柄)

示例: testqueryservice.lua

```
local skynet = require "skynet"
```

```
local args = { ... }
```

```
if(#args == 0) then
```

```
    table.insert(args, "uniqueservice")
```

```
end
```

```
skynet.start(function()
```

```
    local us
```

```
    skynet.error("start query service")
```

```
    --如果test服务未被创建，该接口将会阻塞，后面的代码将不会执行
```

```
    if ( #args == 2 and args[1] == "true" ) then
```

```
        us = skynet.queryservice(true, args[2])
```

```
    else
```

```
        us = skynet.queryservice(args[1])
```

```
    end
```

```
    skynet.error("end query service handler:", skynet.address(us))
```

```
end)
```



运行结果:

如果不启动test全局唯一服务, 直接执行查询函数

```
$ ./skynet examples/config
testqueryservice    #终端输入
[:01000010] LAUNCH snlua testqueryservice
[:01000010] start query service    #阻塞住, 不再执行后面的代码
```

启动test全局唯一服务, 再执行查询函数

```
$ ./skynet examples/config
testunique    #终端输入
[:01000010] LAUNCH snlua testunique
[:01000010] test unique service
[:01000012] LAUNCH snlua uniqueservice #第一次创建全局唯一服务成功
[:01000012] Server First Test
[:01000010] uniqueservice handler: :01000012
testqueryservice
[:01000019] LAUNCH snlua testqueryservice    #再启动查询
[:01000019] start query service
[:01000019] end query service handler: :01000012 #skynet.queryservice将会返回
```

注意:

当调用uniqueservice只传一个服务名时, 表示创建当前skynet节点的全局唯一服务。当第一个参数传递true, 第二个参数传递服务名时, 则表示所有节点的全局唯一服务。

调用queryservice时, 也可以选择是否传递第一个参数true, 表示查询的是当前skynet节点的全局唯一服务, 还是所有节点的全局唯一服务。这两种全局唯一服务作用范围是不同的, 所以可以同时存在同名的但作用范围不同的全局唯一服务。

## 4.3 多节点中的全局服务

一台主机是可以创建多个Skynet节点的, 只有端口不冲突  
(只是考虑到配置时thread = 2就已经发挥出了一颗CPU地性能)

### 4.3.1 启动两个skynet节点

首先，我们先启动两个节点出来。copy两个份 `examp/config` 为 `config1` 与 `config2`，`config1`中修改如下：

`config1`：

```
include "config.path"

-- preload = "./examples/preload.lua" -- run preload.lua before every lua service run
thread = 2
logger = nil
logpath = "."
harbor = 1 --表示每个节点编号
address = "127.0.0.1:2526"
master = "127.0.0.1:2013"
start = "console" -- main script 只启动一个console.lua服务
bootstrap = "snlua bootstrap" -- The service for bootstrap
standalone = "0.0.0.0:2013" --主节点才会用到这个，绑定地址 告诉自己是主节点
-- snax_interface_g = "snax_g"
cpath = root.."cservice/?..so"
-- daemon = "./skynet.pid"
```

`config2`：

```
include "config.path"

-- preload = "./examples/preload.lua" -- run preload.lua before every lua service run
thread = 2
logger = nil
logpath = "."
harbor = 2 --编号需要改
address = "127.0.0.1:2527" --改一个跟config1不同的端口
master = "127.0.0.1:2013" --主节点地址不变
start = "console" -- main script
bootstrap = "snlua bootstrap" -- The service for bootstrap
--standalone = "0.0.0.0:2013" --作为从节点，就注释掉这里
-- snax_interface_g = "snax_g"
cpath = root.."cservice/?..so"
-- daemon = "./skynet.pid"
```

启动两个终端分别启动如下：

节点1启动：

```
./skynet examples/config1
[:01000001] LAUNCH logger
[:01000002] LAUNCH snlua bootstrap
[:01000003] LAUNCH snlua launcher
[:01000004] LAUNCH snlua cmaster #启动主节点cmaster服务
[:01000004] master listen socket 0.0.0.0:2013 #监听端口2013
[:01000005] LAUNCH snlua cslave #主节点也要启动一个cslave，去连接cmaster节点
```

```
[ :01000005] slave connect to master 127.0.0.1:2013 #cslave中一旦连接完cmaster就会启动一个harbor服务
[ :01000004] connect from 127.0.0.1:47660 4
[ :01000006] LAUNCH harbor 1 16777221 #cslave启动一个Harbor服务 用于节点间通信
[ :01000004] Harbor 1 (fd=4) report 127.0.0.1:2526 #报告cmaster cslave服务的地址
[ :01000005] Waiting for 0 harbors #cmaster告诉cslave还有多少个其他cslave需要连接
[ :01000005] Shakehand ready #cslave与cmaster握手成功
[ :01000007] LAUNCH snlua datacenterd
[ :01000008] LAUNCH snlua service_mgr
[ :01000009] LAUNCH snlua console
[ :01000002] KILL self
[ :01000004] connect from 127.0.0.1:47670 6 #cmaster收到其他cslave连接请求
[ :01000004] Harbor 2 (fd=6) report 127.0.0.1:2527 #其他cslave报告地址
[ :01000005] Connect to harbor 2 (fd=7), 127.0.0.1:2527 #让当前cslave去连接其他cslave
```

节点2启动:

```
./skynet examples/config2
[ :02000001] LAUNCH logger
[ :02000002] LAUNCH snlua bootstrap
[ :02000003] LAUNCH snlua launcher
[ :02000004] LAUNCH snlua cslave
[ :02000004] slave connect to master 127.0.0.1:2013 #cslave去连接主节点的cmaster服务
[ :02000005] LAUNCH harbor 2 33554436 #cslave也启动一个harbor服务
[ :02000004] Waiting for 1 harbors #等待主节点的cslave来连接
[ :02000004] New connection (fd = 3, 127.0.0.1:37470) #cslave与主节点cslave连接成功
[ :02000004] Harbor 1 connected (fd = 3)
[ :02000004] Shakehand ready #cslave与cmaster握手成功
[ :02000006] LAUNCH snlua service_mgr
[ :02000007] LAUNCH snlua console
[ :02000002] KILL self
```

## 4.3.2 测试全节点全局唯一服

在第一个节点中启动testunique.lua服务，然后第二个节点中启动testqueryservice.lua服务查询

节点1:

```
testunique true uniqueservice #所有节点全局唯一服方式启动
[ :0100000b] LAUNCH snlua testunique true uniqueservice
[ :0100000b] test unique service
[ :0100000c] LAUNCH snlua uniqueservice
[ :0100000c] Server First Test
[ :0100000b] uniqueservice handler: :0100000c
```

节点2:

```
testqueryservice true uniqueservice
[:02000012] LAUNCH snlua testqueryservice true uniqueservice
[:02000012] start query service
[:02000012] end query service handler: :0100000b#节点1已经启动了，所以节点2中能查询到
```

### 4.3.3 本地全局唯一服与全节点全局唯一服区别

节点2还可以创建一个同脚本的本地全局唯一服：

```
testunique uniqueservice
[:0100000c] LAUNCH snlua testunique
[:0100000c] test unique service
[:0100000d] LAUNCH snlua uniqueservice
[:0100000d] Server First Test
[:0100000c] uniqueservice handler: :0100000d #创建了一个本地全局唯一服
```

但是无法创建一个新的全节点全局唯一服：

```
testunique true uniqueservice
[:0100000e] LAUNCH snlua testunique true uniqueservice
[:0100000e] test unique service
[:0100000e] uniqueservice handler: :0100000b #还是节点1的全局唯一服句柄
```

## 5 服务别名

每个服务启动之后，都有一个整形数来表示id，也可以使用字符串id来表示，例如：`:01000010`，其实就是把id：`0x01000010`转换成字符串。

但是这个数字的表示方式会根据服务的启动先后顺序而变化，不是一个固定的值。如果想要方便的获取某个服务，那么可以通过给服务设置别名。

服务句柄 一个有意义的字符串

## 5.1 本地别名与全局别名

---

在skynet中，服务别名可以分为两种：

- 一种是本地别名，本地别名只能在当前skynet节点使用，本地别名必须使用`.` 开头，例如：`.testalias`
- 一种是全局别名，全局别名可以在所有skynet中使用，全局别名不能以`.` 开头，例如：`testalias`

## 5.2 别名注册与查询接口

---

```

-----[[取别名]]-----
local skynet = require "skynet"
require "skynet.manager"

--给当前服务定一个别名，可以是全局别名，也可以是本地别名
skynet.register(aliasname)

--给指定servicehandler的服务定一个别名，可以是全局别名，也可以是本地别名
skynet.name(aliasname, servicehandler)

```

给自己去别名

给别的服务取别名

```

-----[[查询别名]]-----
--查询本地别名为aliasname的服务，返回servicehandler，不存在就返回nil
skynet.localname(aliasname)

--[[
查询别名为aliasname的服务，可以是全局别名也可以是本地别名，
1、当查询本地别名时，返回servicehandler，不存在就返回nil
2、当查询全局别名时，返回servicehandler，不存在就阻塞等待到该服务初始化完成
]]--
local skynet = require "skynet.harbor"
harbor.queryname(aliasname)

```

不要使用localname查询全局别名

当前节点阻塞，其他节点可以创建

因为别名是全局的，多节点通信到需要用到harbor  
也可以查询本地别名

注意：本地别名与全局别名可以同时存在。

## 5.3 给服务注册别名

### 5.3.1 给普通服取别名

示例代码: testalias.lua

```
local skynet = require "skynet"
require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()

    local handle = skynet.newservice("test")

    skynet.name(".testalias", handle)    --给服务起一个本地别名
    skynet.name("testalias", handle)    --给服务起一个全局别名

    handle = skynet.localname(".testalias")
    skynet.error("localname .testalias handle", skynet.address(handle))

    handle = skynet.localname("testalias")    --只能查本地，不能查全局别名
    skynet.error("localname testalias handle", skynet.address(handle))

    handle = harbor.queryname(".testalias")
    skynet.error("queryname .testalias handle", skynet.address(handle))

    handle = harbor.queryname("testalias")
    skynet.error("queryname testalias handle", skynet.address(handle))

end)
```

- 上面服务通过skynet.newservice来启动一个test.lua服务，test.lua代码如下：

```
local skynet = require "skynet"
skynet.start(function()
    skynet.error("My new service")
end)
```

- 运行结果：

```

$ ./skynet examples/config
testalias #运行main.lua后在终端输入
[:0100000a] LAUNCH snlua testalias
[:0100000b] LAUNCH snlua test
[:0100000b] My new service
[:0100000a] localname .testalias handle :0100000b #skynet.localname查到本地名
[:0100000a] localname testalias handle nil #skynet.localname查不到全局名
[:0100000a] queryname .testalias handle :0100000b #harbor.queryname查到本地名
[:0100000a] queryname testalias handle :0100000b #harbor.queryname查到全局名

```

### 5.3.2 全局别名查询阻塞

如果全局别名不存在，那么这个时候调用函数 `harbor.queryname`，将会阻塞，直到全局别名的服务创建成功。

示例代码: testalias.lua

```

local skynet = require "skynet"
require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()

    handle = skynet.localname(".testalias")    --查询本地别名不阻塞
    skynet.error("localname .testalias handle", skynet.address(handle))

    handle = skynet.localname("testalias")     --无法查询全局别名
    skynet.error("localname testalias handle", skynet.address(handle))

    handle = harbor.queryname(".testalias")    --查询本地别名不阻塞
    skynet.error("queryname .testalias handle", skynet.address(handle))

    handle = harbor.queryname("testalias")     --查询全局别名阻塞 等待其他节点注册别名
    skynet.error("queryname testalias handle", skynet.address(handle))

end)

```

运行结果:



```
$ ./skynet examples/config
testalias
[:0100000a] LAUNCH snlua testalias
[:0100000a] localname .testalias handle nil #skynet.localname查到本地名
[:0100000a] localname testalias handle nil #skynet.localname查不到全局名
[:0100000a] queryname .testalias handle nil #harbor.queryname查到本地名
#harbor.queryname查不到全局名，函数阻塞
```

### 5.3.3 多节点中的全局别名

启动两个skynet节点，在节点1取别名，节点2查询别名：

- 节点1, testaliasname.lua

```
local skynet = require "skynet"
require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()
    local handle = skynet.newservice("test")
    skynet.name(".testalias", handle) --给服务起一个本地别名
    skynet.name("testalias", handle) --给服务起一个全局别名
end)
```

- 节点2, testaliasquery.lua

```
local skynet = require "skynet"
require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()

    handle = skynet.localname(".testalias")
    skynet.error("localname .testalias handle", skynet.address(handle))

    handle = skynet.localname("testalias")
    skynet.error("localname testalias handle", skynet.address(handle))

    handle = harbor.queryname(".testalias")
    skynet.error("queryname .testalias handle", skynet.address(handle))

    handle = harbor.queryname("testalias")
    skynet.error("queryname testalias handle", skynet.address(handle))

end)
```

- 先启动节点1运行testaliasname.lua，再启动节点2运行

```
testaliasquery
[:0200000a] LAUNCH snlua testaliasquery
[:0200000a] localname .testalias handle nil
[:0200000a] localname testalias handle nil
[:0200000a] queryname .testalias handle nil
[:0200000a] queryname testalias handle :0100000b --查询到节点1创建的服务
```

节点1的服务

节点2的服务

### 5.3.4 杀死带别名的服务

全局别名还能查到handle

给一个服务取了别名后，杀死它，本地别名将会注销掉，但是全局别名依然存在，通过全局别名查询到的handle已经没有任何意义。如果通过handle进行一些操作将得到不可预知的问题。

```
local skynet = require "skynet"
require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()

    local handle = skynet.newservice("test")

    skynet.name(".testalias", handle) --给服务起一个本地别名
    skynet.name("testalias", handle) --给服务起一个全局别名

    handle = skynet.localname(".testalias")
    skynet.error("localname .testalias handle", skynet.address(handle))

    handle = skynet.localname("testalias")
    skynet.error("localname testalias handle", skynet.address(handle))

    handle = harbor.queryname(".testalias")
    skynet.error("queryname .testalias handle", skynet.address(handle))

    handle = harbor.queryname("testalias")
    skynet.error("queryname testalias handle", skynet.address(handle))

    skynet.kill(handle) --杀死带别名服务

    handle = skynet.localname(".testalias")
    skynet.error("localname .testalias handle", skynet.address(handle))
```

查询出来并杀死他

```

handle = skynet.localname("testalias")
skynet.error("localname testalias handle", skynet.address(handle))

handle = harbor.queryname(".testalias")
skynet.error("queryname .testalias handle", skynet.address(handle))

handle = harbor.queryname("testalias")
skynet.error("queryname testalias handle", skynet.address(handle))

end)

```

运行结果:

```

testalias
[:0100000a] LAUNCH snlua testalias
[:0100000b] LAUNCH snlua test
[:0100000b] My new service
[:0100000a] localname .testalias handle :0100000b
[:0100000a] localname testalias handle nil
[:0100000a] queryname .testalias handle :0100000b
[:0100000a] queryname testalias handle :0100000b
[:0100000a] KILL :100000b
[:0100000a] localname .testalias handle nil
[:0100000a] localname testalias handle nil
[:0100000a] queryname .testalias handle nil
[:0100000a] queryname testalias handle :0100000b #全局别名还存在，但是已经杀死该服务了。

```

skynet的全局别名服务是在cslave里面实现的，现在不允许二次修改全局别名绑定关系，所以全局别名一般用来给一个永远不会退出的服务来启用。

但是有些情况下，我们确实需要二次修改全局别名绑定关系，那么这个时候，我们可以尝试去修改一下 `cslave.lua` 文件，修改内容如下：

```

function harbor.REGISTER(fd, name, handle)
    --assert(globalname[name] == nil) --将这一行注释掉 修改源码
    globalname[name] = handle
    response_name(name)
    socket.write(fd, pack_package("R", name, handle))
    skynet.redirect(harbor_service, handle, "harbor", 0, "N " .. name)
end

```

运行一个二次修改全局别名绑定关系的服务，例如：

```

local skynet = require "skynet"

```

```

require "skynet.manager"
local harbor = require "skynet.harbor"

skynet.start(function()

    local handle = skynet.newservice("test")

    skynet.name("testalias", handle)    --给服务起一个全局别名

    handle = harbor.queryname("testalias")
    skynet.error("queryname testalias handle", skynet.address(handle))

    skynet.kill(handle) --杀死带全局别名服务
    handle = skynet.newservice("test")
    skynet.name("testalias", handle)    --全局别名给其他服务使用

    handle = harbor.queryname("testalias")
    skynet.error("queryname testalias handle", skynet.address(handle))

end)

```

运行结果:

```

testalias
[:0100000a] LAUNCH snlua testalias
[:0100000b] LAUNCH snlua test
[:0100000b] My new service
[:0100000a] queryname testalias handle :0100000b
[:0100000a] KILL :100000b
[:0100000c] LAUNCH snlua test
[:0100000c] My new service
[:0100000a] queryname testalias handle :0100000c    #服务别名二次修改成功

```

## 5.4 全局别名与全局唯一服名区别

全局唯一服名与这里的全局别名是两个概念的名词。

全局唯一服名称: 是用来**标识服务是唯一的**，服务名称一般就是脚本名称，无法更改。

全局别名: 是用来给**服务起别名的**，既可以给普通服起别名，也可以给全局唯一服起别名。

他们两种名字是在不同的体系中的，有各种的起名字的方式，以及查询的方式。

所以不要尝试用skynet.queryservice查询一个全局别名，也不要尝试使用harbor.queryname去查询一个全局唯一服。

例如：

```
local handle = skynet.uniqueservice("test") --启动一个全局唯一服，名字为test

handle = harbor.queryname("test")           --查不到的，会一直阻塞
      查全局别名
```

或者：

```
local handle = skynet.uniqueservice("test") --启动一个全局唯一服，名字为test

skynet.name("testalias", handle)           --再起一个全局别名

handle = skynet.queryservice("testalias")   --查不到的，也会一直阻塞
      查全局唯一服务名（脚本名）
```

## 6 服务调度

```
local skynet = require "skynet"
--让当前的任务等待 time * 0.01s 。
skynet.sleep(time)

--启动一个新的任务去执行函数 func ，其实就是开了一个协程，函数调用完成将返回线程句柄
--虽然你也可以使用原生的coroutine.create来创建协程，但是会打乱skynet的工作流程
skynet.fork(func, ...)

--让出当前的任务执行流程，使本服务内其它任务有机会执行，随后会继续运行。
skynet.yield()

--让出当前的任务执行流程，直到用 wakeup 唤醒它。
skynet.wait()

--唤醒用 wait 或 sleep 处于等待状态的任务。
skynet.wakeup(co)

--设定一个定时触发函数 func ，在 time * 0.01s 后触发。
skynet.timeout(time, func)

--返回当前进程的启动 UTC 时间（秒）。
skynet.starttime()

--返回当前进程启动后经过的时间（0.01 秒）。
skynet.now()

--通过 starttime 和 now 计算出当前 UTC 时间（秒）。
skynet.time()
```

## 6.1 使用sleep休眠

示例代码: testsleep.lua

```

local skynet = require "skynet"

skynet.start(function ()
    skynet.error("begin sleep")
    skynet.sleep(500)
    skynet.error("begin end")
end)

```

运行结果(还是先运行main.lua):

```

$ ./skynet examples/config
testsleep                                #输入testsleep
[:01000010] LAUNCH snlua testsleep
[:01000010] begin sleep
#执行权限已经被第一个服务testsleep给使用了，这里输入个新的服务test，并不会马上就启动服务
test
[:01000010] begin end
[:01000012] LAUNCH snlua test  #等第一个服务完成任务了，才启动新的服务
[:01000012] My new service

```

在console服务中输入 `testsleep` 之后，马上再输入 `test`，会发现，`test` 服务不会马上启动，因为这个时候console正在忙于第一个服务 `testsleep` 初始化，需要等待5秒钟之后，输入的 `test` 才会被console处理。

**注意：**以上做法是不正确的，在 `skynet.start` 函数中的服务初始化代码不允许有阻塞函数的存在，服务的初始化要求尽量快的执行完成，所有的业务逻辑代码不会写在 `skynet.start` 里面。

## 6.2 在服务中开启新的线程

在skynet的服务中我们可以开一个新的线程来处理业务（注意这个的线程并不是传统意义上的线程，更像是一个虚拟线程，其实是通过协程来模拟的）。

示例代码: testfork.lua

```

local skynet = require "skynet"

function task(timeout)

```

```

skynet.error("fork co:", coroutine.runing())
skynet.error("begin sleep")
skynet.sleep(timeout)

skynet.error("begin end")
end

skynet.start(function ()
    skynet.error("start co:", coroutine.runing())
    skynet.fork(task, 500) --开启一个新的线程来执行task任务
    --skynet.fork(task, 500) --再开启一个新的线程来执行task任务
end)

```

运行结果:

```

$ ./skynet examples/config
testfork                                #输入testfork
[:0100000a] LAUNCH snlua testfork
[:0100000a] start thread: 0x7f684d967568 false #不同的两个协程
[:0100000a] fork thread: 0x7f684d969f68 false
[:0100000a] begin sleep

```

可以看到在 `testfork` 启动后, `console` 服务仍然可以接受终端输入的 `test`, 并且启动。

以后如果遇到需要长时间运行, 并且出现阻塞情况, 都要使用 `skynet.fork` 在创建一个新的线程(协程)。

- 查看源码skynet.lua了解底层实现, 其实就是使用coroutine.create实现  
每次使用skynet.fork其实都是从协程池中获取未被使用的协程, 并把该协程加入到fork队列中, 等待一个消息调度, 然后会依次把fork队列中协程拿出来执行一遍, 执行结束后, 会把协程重新丢入协程池中, 这样可以避免重复开启关闭协程的额外开销。

## 6.3 长时间占用执行权限的任务

示例代码: busytask.lua



```

local skynet = require "skynet"

function task(name)
    local i = 0
    skynet.error(name, "begin task")
    while ( i < 200000000)
    do
        i = i+1
    end
    skynet.error(name, "end task", i)
end

skynet.start(function ()
    skynet.fork(task, "task1")
    skynet.fork(task, "task2")
end)

```

运行结果：

```

$ ./skynet examples/config
busytask
[:01000010] LAUNCH snlua busytask
[:01000010] task1 begin task      --先运行task1
[:01000010] task1 end task 200000000
[:01000010] task2 begin task      --再运行task2
[:01000010] task2 end task 200000000

```

上面的运行结果充分说明了，`skynet.fork` 创建的线程其实通过lua协程来实现的，即一个协程占用执行权后，其他的协程需要等待。

## 6.4 使用skynet.yield让出执行权

示例代码：testyield.lua

```

local skynet = require "skynet"

```

```

function task(name)
    local i = 0
    skynet.error(name, "begin task")
    while ( i < 200000000)
    do
        i = i+1
        if i % 50000000 == 0 then
            skynet.yield()
            skynet.error(name, "task yield")
        end
    end
    skynet.error(name, "end task", i)
end

skynet.start(function ()
    skynet.fork(task, "task1")
    skynet.fork(task, "task2")
end)

```

运行结果:

```

$ ./skynet examples/config
testyield
[:01000010] LAUNCH snlua testyield
[:01000010] task1 begin task
[:01000010] task2 begin task
[:01000010] task1 task yield
[:01000010] task2 task yield
[:01000010] task1 task yield
[:01000010] task2 task yield
[:01000010] task1 task yield
[:01000010] task2 task yield
[:01000010] task1 task yield
[:01000010] task1 end task 200000000
[:01000010] task2 task yield
[:01000010] task2 end task 200000000

```

通过使用 `skynet.yield()` 然后同一个服务中的不同线程都可以得到执行权限。

## 6.5 线程间的简单同步

同一个服务之间的线程可以通过, `skynet.wait` 以及 `skynet.wakeup` 来同步线程

示例代码: testwakeup.lua

```
local skynet = require "skynet"
local cos = {}

function task1()
    skynet.error("task1 begin task")
    skynet.error("task1 wait")
    skynet.wait()          --task1去等待唤醒
    --或者skynet.wait(coroutine.running())
    skynet.error("task1 end task")
end

function task2()
    skynet.error("task2 begin task")
    skynet.error("task2 wakeup task1")
    skynet.wakeup(cos[1])  --task2去唤醒task1, task1并不是马上唤醒, 而是等task2运行完
    skynet.error("task2 end task")
end

skynet.start(function ()
    cos[1] = skynet.fork(task1)  --保存线程句柄
    cos[2] = skynet.fork(task2)
end)
```

运行结果:

```
$ ./skynet examples/config
testwakeup
[:01000010] LAUNCH snlua testwakeup
[:01000010] task1 begin task
[:01000010] task2 begin task
[:01000010] task1 wait
[:01000010] task2 wakeup task1
[:01000010] task2 end task
[:01000010] task1 end task
```

需要注意的是: `skynet.wakeup`除了能唤醒`wait`线程, 也可以唤醒`sleep`的线程

## 6.6 定时器的使用

skynet中的定时器，其实是通过给定时器线程注册了一个超时时间，并且占用了一个空闲协程，空闲协程也是从协程池中获取，超时后会使用空闲协程来处理超时回调函数。

### 6.6.1 启动一个定时器

示例代码：testtimeout.lua

```
local skynet = require "skynet"

function task()
    skynet.error("task", coroutine.running())
end

skynet.start(function ()
    skynet.error("start", coroutine.running())
    skynet.timeout(500, task) --5秒钟之后运行task函数,只是注册一下回调函数,并不会阻塞
end)
```

后面的内容可以继续运行

运行结果：

```
$ ./skynet examples/config
testtimeout
[:01000010] LAUNCH snlua testtimeout
[:01000010] task
```

### 6.6.2 skynet.start源码分析

其实skynet.start服务启动函数实现中，就已经启动了一个timeout为0s的定时器，来执行通过skynet.start函数传参得到的初始化函数。其目的是为了让skynet工作线程调度一次新服务。这一次服务调度最重要的意义在于把fork队列中的协程全部执行一遍。

### 6.6.3 循环启动定时器

```

local skynet = require "skynet"

function task()
    skynet.error("task", coroutine.running())
    skynet.timeout(500, task)
end

skynet.start(function () --skynet.start启动一个timeout来执行function, 创建了一个协程
    skynet.error("start", coroutine.running())
    skynet.timeout(500, task) --由于function函数还没用完协程, 所有这个timeout又创建了一个协程
end)

```

执行结果，交替使用协程池中的协程：

```

testtimeout
[:0100000a] LAUNCH snlua testtimeout
[:0100000a] start thread: 0x7f525b16a048 false #start函数也执行完, 这个协程就空闲下来了
[:0100000a] task thread: 0x7f525b16a128 false #当前服务的协程池中只有两个协程, 所以是交替使用
[:0100000a] task thread: 0x7f525b16a048 false
[:0100000a] task thread: 0x7f525b16a128 false
[:0100000a] task thread: 0x7f525b16a048 false

```

## 6.7 获取时间

示例代码：testtime.lua

```

local skynet = require "skynet"

function task()
    skynet.error("task")
    skynet.error("start time", skynet.starttime()) --获取skynet节点开始运行的时间
    skynet.sleep(200)
    skynet.error("time", skynet.time())           --获取当前时间
    skynet.error("now", skynet.now())             --获取skynet节点已经运行的时间
end                                               节点的运行时间，并不是服务运行的时间

skynet.start(function ()
    skynet.fork(task)
end)

```

运行结果：

```

$ ./skynet examples/config
testtime
[:01000010] LAUNCH snlua testtime
[:01000010] task
[:01000010] start time 1517161846
[:01000010] time 1517161850.73
[:01000010] now 473

```

## 6.8 错误处理

lua中的错误处理都是通过assert以及error来抛异常，并且中断当前流程，skynet也不例外，但是你真的懂assert以及error吗？

注意这里的error不是skynet.error，skynet.error单纯写日志的，并不会中断流程。

skynet中使用assert或error后，服务会不会中断，skynet节点会不会中断？

来看一个例子：

testassert.lua 使用虚拟机有隔离的效果，一个协程的致命错误不会影响到其他部分

```

local skynet = require "skynet"

function task1()
    skynet.error("task1", coroutine.running())
    skynet.sleep(100)
    assert(nil)
    --error("error occurred")
    skynet.error("task2", coroutine.running(), "end")
end

function task2()
    skynet.error("task2", coroutine.running())
    skynet.sleep(500)
    skynet.error("task2", coroutine.running(), "end")
end

skynet.start(function ()
    skynet.error("start", coroutine.running())
    skynet.fork(task1)
    skynet.fork(task2)
end)

```

运行结果:

```

testassert
[:0100000a] LAUNCH snlua testassert
[:0100000a] start thread: 0x7fd9b12938e8 false
[:0100000a] task1 thread: 0x7fd9b12939c8 false
[:0100000a] task2 thread: 0x7fd9b1293aa8 false
[:0100000a] lua call [0 to :100000a : 2 msgsz = 0] error : ./lualib/skynet.lua:534:
./lualib/skynet.lua:156: ./my_workspace/testassert.lua:6: assertion failed!
stack traceback:
  [C]: in function 'assert'
  ./my_workspace/testassert.lua:6: in function 'task1'
  ./lualib/skynet.lua:468: in upvalue 'f'
  ./lualib/skynet.lua:106: in function <./lualib/skynet.lua:105>
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet.lua:534: in function 'skynet.dispatch_message'
[:0100000a] task2 thread: 0x7fd9b1293aa8 end

```

上面的结果已经很明显了，开了两个协程分别执行task1、task2，task1断言后终止掉当前协程，不会再往下执行，但是task2还是能正常执行。skynet节点也没有挂掉，还是能正常运行。

那么我们在处理skynet的错误的时候可以大胆的使用assert与error，并不需要关注错误。当然，一个好的服务端，肯定不能一直出现中断掉的协程。

协程中断掉之后不会再加入协程池

- 如果不想把协程中断掉，可以使用pcall来捕捉异常，例如：

保护模式执行函数

```

local skynet = require "skynet"

```

```

function task1()
    skynet.error("task1", coroutine.running())
    skynet.sleep(100)
    assert(nil)
    skynet.error("task2", coroutine.running(), "end")
end

function task2()
    skynet.error("task2", coroutine.running())
    skynet.sleep(500)
    skynet.error("task2", coroutine.running(), "end")
end

skynet.start(function ()
    skynet.error("start", coroutine.running())
    skynet.fork(pcall, task1)
    skynet.fork(pcall, task2)
end)

```

pcall ( task1 )

fork的追加参数是pcall函数的参数

运行结果：

```

testassert
[:0100000a] LAUNCH snlua testassert
[:0100000a] start thread: 0x7f93d3967568 false
[:0100000a] task1 thread: 0x7f93d3967aa8 false
[:0100000a] task2 thread: 0x7f93d3967b88 false
[:0100000a] task2 thread: 0x7f93d3967b88 end

```

## 7 服务间消息通信

skynet中的每一个服务都有一个独立的lua虚拟机，逻辑上服务之间是相互隔离的，那么你就不能使用传统意义上的LUA全局变量来进行服务间通信了。



在skynet中服务之间可以通过skynet消息调度机制来完成通信。skynet中的服务是基于actor模型设计出来的，每个服务都可以接收消息，处理消息，发送应答消息。

每条 skynet 消息由 6 部分构成：消息类型、session、发起服务地址、接收服务地址、消息 C 指针、消息长度。

## 7.1消息类型

在 skynet 中消息分为多种类别，对应的也有不同的编码方式（即协议），消息类型的宏定义可以查看 skynet.h 中：

### 协议类型 protocol type

```
#define PTYPE_TEXT 0
#define PTYPE_RESPONSE 1 //表示一个回应包 是对请求的应答
#define PTYPE_MULTICAST 2 //广播消息
#define PTYPE_CLIENT 3 //用来处理网络客户端的请求消息
#define PTYPE_SYSTEM 4 //系统消息
#define PTYPE_HARBOR 5 //跨节点消息
#define PTYPE_SOCKET 6 //套接字消息
#define PTYPE_ERROR 7 //错误消息，一般服务退出的时候会发送error消息给关联的服务
#define PTYPE_QUEUE 8 //没有办法发送正常应答（PTYPE_RESPONSE）的时候
#define PTYPE_DEBUG 9
#define PTYPE_LUA 10 //lua类型的消息，最常用
#define PTYPE_SNAX 11 //snax服务消息

#define PTYPE_TAG_DONTCOPY 0x10000 不允许拷贝
#define PTYPE_TAG_ALLOCSESSION 0x20000 动态申请
```

上面的消息类型有多种，但是最常用的是PTYPE\_LUA，对应到lua层，叫做lua消息，大部分服务一般使用这种消息，默认情况下，PTYPE\_RESPONSE、PTYPE\_ERROR、PTYPE\_LUA三种消息类型已经注册（查看源码了解情况），如果想使用其他的消息类型，需要自己显示注册消息类型。

## 7.2 注册消息处理函数

当我们需要在一个服务中监听指定类型的消息，就需要在服务启动的时候先注册该类型的消息的监听，通常是在服务的入口函数 `skynet.start` 处通过调用 `skynet.dispatch` 来注册绑定：

```
--服务启动入口
skynet.start(function()
    --注册"lua"类型消息的回调函数
    skynet.dispatch("lua", function(session, address, ...)
        dosomething(...)
    end)
end)
```

一旦注册成功，那么只要是发送给这个服务的消息是lua类型消息，那么都会调用我们注册的function进行处理。

例如testluamsg.lua：

```
skynet = require "skynet"
require "skynet.manager"

local function dosomething(session, address, ...)
    skynet.error("session", session)
    skynet.error("address", skynet.address(address))
    local args = {...}
    for i,v in pairs(args) do
        skynet.error("arg"..i..":", v)
    end
end

skynet.start(function()
    --注册"lua"类型消息的回调函数
    skynet.dispatch("lua", function(session, address, ...)
        dosomething(session, address, ...)
    end)
    skynet.register(".testluamsg")
end)
```

## 7.3 打包与解包消息

skynet中的消息在发送之前必须要把参数进行打包，然后才发送，接受方收到消息后会自动根据指定的解包函数进行解包，最常用的打包解包函数为skynet.pack与skynet.unpack.

skynet.pack(...) 打包后，会返回两个参数，一个是C指针msg指向数据包的起始地址，sz一个是数据包的长度。msg指针的内存区域是动态申请的。在C语言层面会有动态内存申请

skynet.unpack(msg, sz) 解包后，会返回一个参数列表。需要注意这个时候C指针msg指向的内存不会释放掉。如果msg有实际的用途，skynet框架会帮你在合适的地方释放掉，如果没有实际的用途，自己想释放掉可以使用skynet.trash(msg, sz) 释放掉。

例如：

```
local msg, sz = skynet.pack("nengzhong", 8.8, false)
```

```
local arg1, arg2, arg3 = skynet.unpack(msg, sz)
```

```
skynet.error(arg1, arg2, arg3)
```

```
local arglist = {skynet.unpack(msg, sz)}
```

```
for i,v in pairs(arglist) do
```

```
    skynet.error("arg"..i..":", v)
```

```
end
```

```
skynet.trash(msg, sz) --没有用到skynet框架中，所以用完了需要自己释放一下
```

不知道返回参数个数时，可以使用一个表把参数装起来

注意：skynet.pack返回的msg与sz只用在skynet.unpack中使用才有意义。不要这么使用table.unpack(msg, sz).

## 7.4 发送消息的方法

### 7.4.1 发送无需响应的消息

框架只是注册了三种消息：response、lua、error

其中只有error有指定消息处理函数dispatch

--用 type 类型向 addr 发送未打包的消息。该函数会自动把...参数列表进行打包，默认情况下lua消息使用 skynet.pack打包。addr可以是服务句柄也可以是别名。

```
skynet.send(addr, type, ...)
```

--用 type 类型向 addr 发送一个打包好的消息。addr可以是服务句柄也可以是别名。

```
skynet.rawsend(addr, type, msg, sz)
```

需要自己打包，得到指针和长度

例如testsendmsg.lua:

```
skynet = require "skynet"
```

```
skynet.start(function()
```

```
    skynet.register(".testsendmsg")
```

```
    local testluamsg = skynet.localname(".testluamsg")
```

```
    --发送lua类型的消息给testluamsg，发送成功后立即返回，r的值为0
```

```
    local r = skynet.send(testluamsg, "lua", 1, "nengzhong", true) --申请了C内存 (msg, sz) 已经用与发送，所以不用自己再释放内存了。
```

```
    skynet.error("skynet.send return value:", r)
```

```
    --通过skynet.pack来打包发送
```

```
    r = skynet.rawsend(testluamsg, "lua", skynet.pack(2, "nengzhong", false)) --申请了C内存 (msg, sz) 已经用于发送，所以不用自己再释放内存了。 返回msg和sz
```

```
    skynet.error("skynet.rawsend return value:", r)
```

```
end)
```

先运行testluamsg.lua再运行testsendmsg.lua, 结果如下:

```
$ ./skynet examples/config
```

```
testluamsg
```

```
[ :0100000a] LAUNCH snlua testluamsg
```

```
testsendmsg
```

```
[ :0100000b] LAUNCH snlua testsendmsg
```

```
[ :0100000b] skynet.send return value: 0 #发送完消息马上返回
```

```
[ :0100000b] skynet.rawsend return value: 0 #发送完消息马上返回
```

```
[ :0100000a] session 0 #接收端接到消息
```

```
[ :0100000a] address :0100000b
```

```
[ :0100000a] arg1: 1
```

```
[ :0100000a] arg2: nengzhong
```

```
[ :0100000a] arg3: true
```

```
[ :0100000a] session 0
```

```
[ :0100000a] address :0100000b
```

```
[ :0100000a] arg1: 2
```

```
[ :0100000a] arg2: nengzhong
```

```
[ :0100000a] arg3: false
```

上面的代码我们隐式或显示调用了skynet.pack，一共申请了两段C内存，但是并不需要我们释放C内存。因为已经把这段内存用于发送了，skynet会等到该消息处理完后，自动释放掉它的内存。

## 7.4.2 发送必须响应的消息

两个都是阻塞函数

```
--用默认函数打包消息, 向addr发送type类型的消息并等待返回响应, 并对回应信息进行解包。(自动打包与解包。)  
skynet.call(addr, type, ...)  
--直接向addr发送type类型的msg,sz并等待返回响应, 不对回应信息解包。(需要自己打包与解包)  
skynet.rawcall(addr, type, msg, sz)
```

例如testcallmsg.lua:

```
skynet = require "skynet"  
skynet.start(function()  
    skynet.register(".testcallmsg")  
    --发送lua类型的消息给service, 发送成功, 该函数将阻塞等待响应返回, r的值为响应的返回值  
    local r = skynet.call(".testluamsg", "lua", 1, "nengzhong", true)  
    skynet.error("skynet.call return value:", r)  
  
    --通过skynet.pack来打包发送, 返回的值也需要自己解包  
    r = skynet.unpack(skynet.rawcall(".testluamsg", "lua", skynet.pack(2, "nengzhong", false)))  
    skynet.error("skynet.rawcall return value:", r)  
end)
```

先运行testluamsg.lua再运行testcallmsg.lua, 结果如下:

```
$ ./skynet examples/config  
testluamsg  
[:0100000a] LAUNCH snlua testluamsg  
testcallmsg  
[:0100000b] LAUNCH snlua testcallmsg  
[:0100000a] session 2 #只发送出第一个消息, 现在已经阻塞住, 由于testluamsg并没有返回应答。  
[:0100000a] address :0100000b  
[:0100000a] arg1: 1  
[:0100000a] arg2: nengzhong  
[:0100000a] arg3: true
```

## 7.5 响应消息的方法

对lua消息响应的时候，使用的是PTYPE\_RESPONSE这种消息，也是需要打包，**打包的时候必须与接收到的消息打包方法一致。**

```
skynet.ret -- 目标服务消息处理后需要通过该函数将结果返回
skynet.retpack(...) -- 将消息用skynet.pack 打包，并调用 ret 回应。
```

修改testluamsg.lua

```
skynet = require "skynet"
require "skynet.manager"

local function dosomething(session, address, ...)
    skynet.error("session", session)
    skynet.error("address", skynet.address(address))
    local args = {...}
    for i,v in pairs(args) do
        skynet.error("arg"..i..":", v)
    end
    return 100, false
end

skynet.start(function()
    --注册"lua"类型消息的回调函数
    skynet.dispatch("lua", function(session, address, ...)
        --skynet.sleep(500)
        skynet.retpack(dosomething(session, address, ...)) --申请响应消息C内存
        --或者skynet.ret(skynet.pack(dosomething(session, address, ...)))
    end) --skynet.dispatch完成后,释放调用接收消息C内存
    skynet.register(".testluamsg")
end)
```

先运行testluamsg.lua再运行testcallmsg.lua, 结果如下:

```
$ ./skynet examples/config
testluamsg
[:0100000a] LAUNCH snlua testluamsg
testcallmsg
[:0100000b] LAUNCH snlua testcallmsg
[:0100000a] session 2
[:0100000a] address :0100000b
[:0100000a] arg1: 1
[:0100000a] arg2: nengzhong
[:0100000a] arg3: true
[:0100000b] skynet.call return value: 100 #第一个call返回, 返回值为100
[:0100000a] session 3 #通过skynet.call方式发送消息session才起作用了
[:0100000a] address :0100000b
[:0100000a] arg1: 2
[:0100000a] arg2: nengzhong
[:0100000a] arg3: false
```

```
[ :0100000b] skynet.rawcall return value: 100 false #第二个call也返回
```

注意：

- 1、应答消息打包的时候，打包方法必须与接收消息的打包方式一致。
- 2、skynet.ret不需要指定应答消息是给哪个服务的。

因为每次接收到消息时，服务都会启动一个协程来处理，并且把这个协程与源服务地址绑定在一起了（其实就是把协程句柄作为key，源服务地址为value，记录在一张表中）。需要响应的时候可以根据协程句柄找到源服务地址。

## 7.6 lua消息收发综合应用

mydb.lua

```
local skynet = require "skynet"
require "skynet.manager" -- import skynet.register
local db = {}

local command = {}

function command.GET(key)
    return db[key]
end

function command.SET(key, value)
    db[key] = value
end

skynet.start(function()
    --注册该服务的lua消息回调函数
    skynet.dispatch("lua", function(session, address, cmd, ...)
        --接受到的第一个参数作为命令使用

        cmd = cmd:upper()
    end)
end)
```

```

local f = command[cmd] --查询cmd命令的具体处理方法
if f then
    --执行查询到的方法，并且通过skynet.ret将执行结果返回
    skynet.retpack(f(...))
else
    skynet.error(string.format("Unknown command %s", tostring(cmd)))
end
end)
skynet.register ".mydb" --给当前服务注册一个名字，便于其他服务给当前服务发送消息
end)

```

testmydb.lua

```

local skynet = require "skynet"
local key,value = ...
function task()
    --给.mydb服务发送一个lua消息，命令为set
    --发送成功后直接返回，不管接收消息端的是否调用skynet.ret，skynet.send的返回值都为0
    local r = skynet.send(".mydb", "lua", "set", key, value)
    skynet.error("mydb set Test", r)

    --给.mydb服务发送一个lua消息，命令为get
    --如果接收端没有调用skynet.ret，则skynet.call将一直阻塞
    r = skynet.call(".mydb", "lua", "get", key)
    skynet.error("mydb get Test", r)
    skynet.exit()
end
skynet.start(function()
    skynet.fork(task)
end)

```

## 7.7 session的意义

session只有在使用skynet.call或者skynet.rawcall发送消息的时候才有意义。



因为有可能一个服务开了多个协程去call消息，然后多个协程都在等待应答消息，回来了一个应答，那么到底是唤醒哪个协程，就可以通过session来判断了，skynet中的session能保证本服务中发出的消息是唯一的。消息与响应一一对应起来。

例如

echoluamsg.lua

```
skynet = require "skynet"

skynet.start(function()
    --注册"lua"类型消息的回调函数
    skynet.dispatch("lua", function(session, address, msg)
        skynet.sleep(math.random(100, 500))
        skynet.retpack(msg:upper())
    end)
    skynet.register(".echoluamsg")
end)
```

testforkcall.lua:

```
skynet = require "skynet"

local function task(id)
    for i = 1,5 do
        skynet.error("task"..id .." call")
        skynet.error("task"..id .." return:", skynet.call(".echoluamsg", "lua", "task"..id))
    end
end

skynet.start(function()
    skynet.fork(task, 1)    --开两个线程去执行
    skynet.fork(task, 2)
end)
```

先运行echoluamsg，再运行testforkcall，结果如下：

```
testforkcall
[:0100000b] LAUNCH snlua testforkcall
[:0100000b] task1 call
[:0100000b] task2 call
[:0100000b] task2 return: TASK2          #消息能够一一对应在一起
[:0100000b] task2 call
[:0100000b] task1 return: TASK1
```

```
[ :0100000b] task1 call
[ :0100000b] task2 return: TASK2
[ :0100000b] task2 call
[ :0100000b] task1 return: TASK1
[ :0100000b] task1 call
[ :0100000b] task1 return: TASK1
[ :0100000b] task1 call
[ :0100000b] task2 return: TASK2
[ :0100000b] task2 call
[ :0100000b] task1 return: TASK1
[ :0100000b] task1 call
[ :0100000b] task1 return: TASK1
[ :0100000b] task2 return: TASK2
[ :0100000b] task2 call
[ :0100000b] task2 return: TASK2
```

## 7.8 使用skynet.response响应消息

在使用skynet.ret或者skynet.retpack进行应答时，必须要保证与接受请求时在同一个协程中（源服务地址与协程句柄已经一一对应），也就是说在哪个协程接受的请求也必须在这个协程去做响应。看下面的一个例子：

errormydb.lua

```
local skynet = require "skynet"
require "skynet.manager" -- import skynet.register
local db = {}

local command = {}

function command.GET(key)
    return db[key]
end

function command.SET(key, value)
    db[key] = value
end
```

```

skynet.start(function()
    skynet.dispatch("lua", function(session, address, cmd, ...)
        skynet.error("lua dispatch ", coroutine.running()) --这个协程接收消息的
        skynet.fork(function(cmd, ...) --开启一个新的协程来处理响应
            skynet.error("fork ", coroutine.running())
            cmd = cmd:upper()
            local f = command[cmd]
            if f then
                skynet.retpack(f(...))
            else
                skynet.error(string.format("Unknown command %s", tostring(cmd)))
            end
        end, cmd, ...)
    end)
    skynet.register ".mydb"
end)

```

运行效果:

```

$ ./skynet examples/config
errormydb #先运行errormydb
[:0100000a] LAUNCH snlua errormydb
testmydb name nengzhong #再运行testmydb
[:0100000b] LAUNCH snlua testmydb name nengzhong
[:0100000b] mydb set Test 0
[:0100000a] lua dispatch thread: 0x7ff359169088 false
[:0100000a] fork thread: 0x7ff35928e0c8 false
[:0100000a] lua call [100000b to :100000a : 0 msgsz = 19] error : ./lualib/skynet.lua:534:
./lualib/skynet.lua:178: dest address type (nil) must be a string or number. #报错误
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet.lua:534: in function 'skynet.manager.dispatch_message'
[:0100000a] lua dispatch thread: 0x7ff359169088 false
[:0100000a] fork thread: 0x7ff35928e1a8 false
[:0100000a] lua call [100000b to :100000a : 2 msgsz = 9] error : ./lualib/skynet.lua:534:
./lualib/skynet.lua:178: dest address type (nil) must be a string or number.
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet.lua:534: in function 'skynet.manager.dispatch_message'

```

在skynet中, 当一个服务收到一个消息的时候, 会启动一个协程来处理, 并且把协程句柄与发送消息的服务地址进行一一对应记录在table中, 当需要响应时, 就使用当前调用 `skynet.ret` 的协程句柄去table中查询对应的服务地址, 然后把消息发给这个服务地址。

如果开了一个新的协程去调用skynet.ret, 那么这个时候使用新启动的协程句柄去查询服务地址, 肯定是查不到的。

当不想接受请求与响应请求在同一个协程中完成时, 我可以使用response替代ret。

```

local skynet = require "skynet"
--参数pack指定应答打包函数，不填默认使用skynet.pack，必须根据接收到消息的打包函数一致
--返回值是一个闭包函数
local response = skynet.response(pack)

--闭包函数使用方式
--参数ok的值可以是 "test"、true、false，为"test"时表示检查接收响应的服务是否存在，为true时表示发送应答
PTYPE_RESPONSE，为false时表示发送PTYPE_ERROR错误消息。
response(ok, ...)

```

代码如下：

```

local skynet = require "skynet"
require "skynet.manager" -- import skynet.register
local db = {}

local command = {}

function command.GET(key)
    return db[key]
end

function command.SET(key, value)
    db[key] = value
end

skynet.start(function()
    skynet.dispatch("lua", function(session, address, cmd, ...)
        --先把发送服务地址以及session打包到闭包中，可以在任意地方调用
        local response = skynet.response(skynet.pack) --指定打包函数，必须根据接收到的消息打包函数一
致
        skynet.fork(function(cmd, ...) --开启一个新的协程来处理响应
            skynet.sleep(500)
            cmd = cmd:upper()
            local f = command[cmd]
            if f then
                response(true, f(...)) --第一个参数true表示应答成功，false则应答个错误消息
            else
                skynet.error(string.format("Unknown command %s", tostring(cmd)))
            end
        end, cmd, ...)
    end)
    skynet.register ".mydb"
end)

```

把发送服务地址以及session打包到闭包中，就可以在任意地方调用了。尽量多用skynet.response进行应答。

## 7.9 skynet.call失败的情况

当一个服务发起请求 `skynet.call` 后等待应答，但是响应服务却退出了（调用 `skynet.exit`），响应服务退出的时候，会自动给未答复的请求发送一个 `error` 消息，告诉它可以从 `skynet.call` 阻塞返回了，请求的服务会直接报一个错误。

`noresponse.lua`如下：

```
local skynet = require "skynet"
require "skynet.manager" -- import skynet.register

skynet.start(function()
    skynet.dispatch("lua", function(session, address, cmd, ...)
        skynet.exit() --在这里退出了服务
    end)
    skynet.register ".noresponse"
end)
```

`testnoresponse.lua`如下：

```
local skynet = require "skynet"

function task()
    r = skynet.call(".noresponse", "lua", "get")
    skynet.error("get Test", r)

    skynet.exit()
end

skynet.start(function()
    skynet.fork(task)
end)
```

运行结果：

```

$ ./skynet examples/config
noresponse
[:01000010] LAUNCH snlua noresponse
testnoresponse
[:01000012] LAUNCH snlua testnoresponse
[:01000010] KILL self
[:01000012] lua call [1000010 to :1000012 : 2 msgsz = 0] error : ./lualib/skynet.lua:534:
./lualib/skynet.lua:156: ./lualib/skynet.lua:391: call failed #call报错
stack traceback:
  [C]: in function 'error'
  ./lualib/skynet.lua:391: in upvalue 'yield_call'
  ./lualib/skynet.lua:402: in function 'skynet.call'
  ./my_workspace/testnoresponse.lua:4: in function 'task'
  ./lualib/skynet.lua:468: in upvalue 'f'
  ./lualib/skynet.lua:106: in function <./lualib/skynet.lua:105>
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet.lua:534: in function 'skynet.dispatch_message'

```

skynet.sleep  
skynet.yield

## 7.10 服务重入问题

如skynet.sleep，会让出CUP使用权，  
这样可以相应其他服务  
如果是运行很久的操作如自加到一亿  
是不会让出CPU使用权的，但是这种  
情况为避免超时太久，会使用yield挂起  
服务，这样就让出了CUP，然后就会有  
重入的问题

同一个 skynet 服务中的一条消息处理中，如果调用了一个阻塞 API，那么它会被挂起。挂起过程中，这个服务可以响应其它消息。这很可能造成时序问题，要非常小心处理。

换句话说，一旦你的消息处理过程有外部请求，那么先到的消息未必比后到的消息先处理完。且每个阻塞调用之后，服务的内部状态都未必和调用前的一致（因为别的消息处理过程可能改变状态）。

来看一个服务重入的情况：

改造一下mydb.lua

```

local skynet = require "skynet"
require "skynet.manager" -- import skynet.register
local db = {}

local command = {}

function command.GET(key)
    skynet.sleep(1000) --这里加了一个阻塞函数

```

```

        return db[key]
    end

    function command.SET(key, value)
        db[key] = value
    end

    skynet.start(function()
        skynet.dispatch("lua", function(session, address, cmd, ...)
            cmd = cmd:upper()
            local f = command[cmd]
            if f then
                skynet.retpack(f(...))
            else
                skynet.error(string.format("Unknown command %s", tostring(cmd)))
            end
        end)
        skynet.register ".mydb"
    end)

```

testmydb.lua还是一样

```

local skynet = require "skynet"
local key,value = ...
function task()
    local r = skynet.send(".mydb", "lua", "set", key, value)
    skynet.error("mydb set Test", r)

    r = skynet.call(".mydb", "lua", "get", key)
    skynet.error("mydb get Test", r)

    skynet.exit()
end
skynet.start(function()
    skynet.fork(task)
end)

```

运行情况:

```

$ ./skynet examples/config
mydb    #先启动mydb
[:01000010] LAUNCH snlua mydb
testmydb name xm    #再启动一个testmydb设置键值对name=xm，并获取name的值
[:01000012] LAUNCH snlua testmydb name xm
[:01000012] mydb set Test 0
testmydb name xh    #再启动一个testmydb设置键值对name=xh，并获取name值
[:01000019] LAUNCH snlua testmydb name xh
[:01000019] mydb set Test 0
[:01000012] mydb get Test xh    #获取到的值都是xh，我们希望获取的是xm
[:01000012] KILL self
[:01000019] mydb get Test xh    #获取到的值都是xh
[:01000019] KILL self

```

上面出现的情况，就是因为，mydb服务处理上一个请求还没结束时，又来了一个新的请求，并且新的请求改变的mydb中name的值，所以等到第一个请求从阻塞状态恢复时，获取到的值也变了。

上面的例子即使把skynet.sleep去掉，换成其他造成协程挂起的函数，依然存在重入的问题。

[skynet.yield](#)

## 7.11 服务临界区

skynet.queue 模块可以帮助你回避这些服务重入或者伪并发引起的复杂性。

```

local queue = require "skynet.queue"
local cs = queue() --获取一个执行队列
cs(f, ...) --将f丢到队列中执行

```

改进mydb.lua服务如下：

```

local skynet = require "skynet"
require "skynet.manager" -- import skynet.register
local queue = require "skynet.queue"
local cs = queue() --获取一个执行队列

```



```

local db = {}
local command = {}

function command.GET(key)
    skynet.sleep(1000)    --这里加了一个阻塞函数
    return db[key]
end

function command.SET(key, value)
    db[key] = value
end

skynet.start(function()
    skynet.dispatch("lua", function(session, address, cmd, ...)
        cmd = cmd:upper()
        local f = command[cmd]
        if f then
            --将f丢到队列中去执行，队列中的函数严格按照先后顺序进行执行
            skynet.retpack(cs(f, ...))
        else
            skynet.error(string.format("Unknown command %s", tostring(cmd)))
        end
    end)
    skynet.register ".mydb"
end)

```

同样的执行步骤：

```

$ ./skynet examples/config
mydb          #先启动mydb
[:01000012] LAUNCH snlua mydb
testmydb name xm      #再启动一个testmydb设置设置键值对name=xm，并获取name的值
[:01000019] LAUNCH snlua testmydb name xm
[:01000019] mydb set Test 0
testmydb name xh      #再启动一个testmydb设置设置键值对name=xh，并获取name值
[:01000020] LAUNCH snlua testmydb name xh
[:01000020] mydb set Test 0
[:01000019] mydb get Test xm    #获取到的值都是xm
[:01000019] KILL self
[:01000020] mydb get Test xh    #获取到的值都是xh
[:01000020] KILL self

```

上面的输出结果就是我们想要的了，把所有不希望重入的函数丢到cs队列中去执行，队列将依次执行每一个函数，前一个函数还没执行完的时候，后面的函数永远不会被执行。

执行队列虽然解决了重入的问题，但是明显降低了服务的并发处理能力，所以使用执行队列的时候尽量缩小临界区的颗粒度大小。

## 7.12 注册其他消息

如果想要使用其他的消息，那么需要显示注册一下：

othermsg.lua代码

```
local skynet = require "skynet"
require "skynet.manager" -- inject skynet.forward_type

skynet.register_protocol {          --注册system消息
    name = "system",
    id = skynet.PTYPE_SYSTEM,
    --pack = skynet.pack,
    unpack = skynet.unpack, --unpack必须指定一下，接收到消息后会自动使用unpack解析
}

skynet.start(function()
    skynet.dispatch("system", function(session, address, ...) --使用unpack解包
        skynet.ret(skynet.pack("nengzhong"))
        --使用skynet.retpack的时候，必须要在skynet.register_protocol指定pack
        --skynet.retpack("nengzhong")
    end)
    skynet.register ".othermsg"
end)
```

testothermsg.lua代码

```
skynet = require "skynet"

skynet.register_protocol {          --注册system消息
    name = "system",
    id = skynet.PTYPE_SYSTEM,
    --pack = skynet.pack,
    --unpack = skynet.unpack
}

skynet.start(function()
    local othermsg = skynet.localname(".othermsg")
    local r = skynet.unpack(skynet.rawcall(othermsg, "system", skynet.pack(1, "nengzhong", true)))
    --使用skynet.call的时候必须要在skynet.register_protocol指定pack与unpack
    --local r = skynet.call(othermsg, "system", 1, "nengzhong", true)
    skynet.error("skynet.call return value:", r)

end)
```

需要注意，error、lua、response都是已经默认注册的消息类型，不要尝试修改他们的协议定义。

## 7.13 代理服务

在 skynet 中，有时候为一个服务实现一个前置的代理服务是很有必要的。所谓代理服务，就是向真正的功能服务发起请求时，请求消息发到另一个代理服务中，由这个代理服务转发这个请求给真正的功能服务。同样，回应消息也会被代理服务转发回去。

### 7.13.1 简单服务代理

示例代码：proxy.lua

```
local skynet = require "skynet"
require "skynet.manager"

local realsvr = ...

skynet.start( function() --注册消息处理函数
    skynet.dispatch("lua", function (session, source, ...) --接收到消息msg, sz
        skynet.ret(skynet.rawcall(realsvr, "lua", skynet.pack(...))) --根据参数列表重新打包消息转发
    end) --释放消息msg, sz
    skynet.register(".proxy")
end)
```

示例代码：testmydb.lua

```
local skynet = require "skynet"
local key,value = ...
function task()
    local r = skynet.send(".proxy", "lua", "set", key, value) --修改为.proxy, 发给代理
    skynet.error("mydb set Test", r)

    r = skynet.call(".proxy", "lua", "get", key) --修改为.proxy, 发给代理
    skynet.error("mydb get Test", r)
    skynet.exit()
end
skynet.start(function()
    skynet.fork(task)
end)
```

运行结果：

```

$ ./skynet examples/config
mydb
[:0100000a] LAUNCH snlua mydb
proxy .mydb
[:0100000b] LAUNCH snlua proxy .mydb
testmydb name nengzhong
[:0100000c] LAUNCH snlua testmydb name nengzhong
[:0100000c] mydb set Test 0
[:0100000c] mydb get Test nengzhong
[:0100000c] KILL self

```

这份代理重复的打包解包效率上有很大问题。

## 7.13.2 转换协议实现代理

上一节效率的问题是因为服务默认接收到lua消息后，会解包消息，但是lua消息已经注册了无法更改，那么我么可以使用skynet.forward\_type进行协议转换。

使用skynet.forward\_type也是启动服务的一种方法，跟skynet.start类似，只不过skynet.forward\_type还需要提供一张消息转换映射表forward\_map, 其他的方法与skynet.start一样。

```

local skynet = require "skynet"
require "skynet.manager" -- inject skynet.forward_type

skynet.register_protocol { --注册system消息
    name = "system",
    id = skynet.PTYPE_SYSTEM,
    unpack = function (...) return ... end, --unpack直接返回不解包了
}

local forward_map = {
    --发送到代理服务的lua消息全部转成system消息,不改变原先LUA的消息协议处理方式
    [skynet.PTYPE_LUA] = skynet.PTYPE_SYSTEM, 只影响当前虚拟机
    --如果接收到应答消息，默认情况下会释放掉消息msg,sz, forward的方式处理消息不会释放掉消息msg,sz
    [skynet.PTYPE_RESPONSE] = skynet.PTYPE_RESPONSE,
}

local realsvr = ...

skynet.forward_type( forward_map ,function() --注册消息处理函数
    skynet.dispatch("system", function (session, source, msg, sz)
        --直接转发给realsvr, 接收到realsvr响应后也不释放内存, 直接转发
        skynet.ret(skynet.rawcall(realsvr, "lua", msg, sz))
    end)--处理完不释放内存msg
    skynet.register(".proxy")
end)

```

mydb.lua代码不变，运行结果：

```
$ ./skynet examples/config
mydb          #先运行mydb
[:01000010] LAUNCH snlua mydb
proxy .mydb   #启动代理服务，并且告诉它真正的处理服务是.mydb
[:01000012] LAUNCH snlua proxy .mydb
testmydb name xx  #启动testmydb服务
[:01000019] LAUNCH snlua testmydb name xx
[:01000019] mydb set Test 0  #send方法成功
[:01000019] mydb get Test xx #get方法成功
[:01000019] KILL self
```

## 7.14 伪造消息

伪造其他服务地址来发送一个消息，可以使用到skynet.redirect

```
local skynet = require "skynet"
--使用source服务地址，发送typename类型的消息给dest服务，不需要接收响应，（source，dest只能是服务ID）
--msg sz一般使用skynet.pack打包生成
skynet.redirect(dest,source,typename, session, msg, sz)
```

例如：testredirect.lua

```
local skynet = require "skynet"

local source, dest = ...
skynet.start(function()
    source = skynet.localname(source)
    dest = skynet.localname(dest)
    skynet.redirect(source, dest, "lua", 0, skynet.pack("nengzhong", 8.8, false))
end)
```

运行结果：

```
testluamsg      #先启动testluamsg.lua
[:0100000a] LAUNCH snlua testluamsg
testsendmsg     #再启动testsendmsg.lua
[:0100000b] LAUNCH snlua testsendmsg
[:0100000b] skynet.send return value: 0
[:0100000b] skynet.rawsend return value: 0
[:0100000a] session 0
[:0100000a] address :0100000b
[:0100000a] arg1: 1
[:0100000a] arg2: nengzhong
[:0100000a] arg3: true
[:0100000a] session 0
[:0100000a] address :0100000b
[:0100000a] arg1: 2
[:0100000a] arg2: nengzhong
[:0100000a] arg3: false
testredirect .testluamsg .testsendmsg #伪造testsendmsg给testluamsg发送一个消息
[:0100000c] LAUNCH snlua testredirect .testluamsg .testsendmsg
[:0100000a] session 0
[:0100000a] address :0100000b
[:0100000a] arg1: nengzhong
[:0100000a] arg2: 8.8
[:0100000a] arg3: false
```

## 7.15 节点间消息通信

globalluamsg.lua

```

skynet = require "skynet"
require "skynet.manager"
skynet.start(function()
    --注册"lua"类型消息的回调函数
    skynet.dispatch("lua", function(session, address, msg)
        skynet.retpack(msg:upper())
    end)
    skynet.register("globalluamsg")
end)

```

globalcall.lua

```

skynet = require "skynet"
harbor = require "skynet.harbor"
skynet.start(function()
    local globalluamsg = harbor.queryname("globalluamsg")
    local r = skynet.call(globalluamsg, "lua", "nengzhong")
    skynet.error("skynet.call return value:", r)

end)

```

先在节点1运行globalluamsg，然后在节点2运行globalcall。

节点1运行情况：

```

globalluamsg
[:0100000a] LAUNCH snlua globalluamsg

```

节点2运行情况：

```

globalcall
[:02000008] LAUNCH snlua globalcall
[:02000008] skynet.call return value: NENGZHONG

```

# 8 Multicast组播

## 8.1 Multicast介绍

```
local mc = require "skynet.multicast"
```

引入 multicast 模块后，你可以使用 skynet 的组播方案。你可以自由创建一个频道，并向其中投递任意消息。频道的订阅者可以收到投递的消息。

你可以通过 new 函数来创建一个频道对象。你可以创建一个新频道，也可以利用已知的频道 id 绑定一个已有频道。

```
local channel = mc.new() -- 创建一个频道，成功创建后，channel.channel 是这个频道的 id 。
local channel2 = mc.new {
    channel = channel.channel, -- 绑定上一个频道
    dispatch = function (channel, source, ...) end, -- 设置这个频道的消息处理函数
}
```

如上面的例子，new 函数可以接收一个参数表。channel 是频道 id，dispatch 是订阅消息的回调函数。如果你不给出 channel id，则新创建一个频道来。

通常，由一个服务创建出频道，再将 channel.channel 这个 id 通知别的地方。获得这个 id 的位置，都可以绑定这个频道。

`channel:publish(...)` 可以向一个频道发布消息。消息可以是任意数量合法的 lua 值。

绑定到一个频道后，默认并不接收这个频道上的消息（也许你只想向这个频道发布消息）。你需要先调用 `channel:subscribe()` 订阅它。

如果不再想收到该频道的消息，调用 `channel:unsubscribe`。

当一个频道不再使用，你可以调用 `channel:delete()` 让系统回收它。注：多次调用 `channel:delete` 是无害的，因为 channel id 不会重复使用。在频道被销毁后再调用 subscribe 或 publish 等也不会引起异常，但订阅是不起作用的，消息也不再广播。



## 8.2 应用组播技术

示例代码推送端: multicastpub.lua

```
local skynet = require "skynet"
local mc = require "skynet.multicast"
local channel

function task()
    local i = 0
    while(i < 100) do
        skynet.sleep(100)
        channel:publish("data"..i) --推送数据
        i = i + 1
    end
    channel:delete()
    skynet.exit()
end
skynet.start(function()
    channel = mc.new() -- 创建一个频道, 成功创建后, channel.channel 是这个频道的 id 。
    skynet.error("new channel ID", channel.channel)
    skynet.fork(task)
end)
```

示例代码接受端: multicastsb.lua

```
local skynet = require "skynet"
local mc = require "skynet.multicast"
local channel
local channelId = ... --从启动参数获取channelID
channelID = tonumber(channelID)

local function recvChannel(channel, source, msg, ...)
    skynet.error("channel ID:", channel, "source:", skynet.address(source), "msg:", msg)
end
skynet.start(function()
    channel = mc.new {
        channel = channelID, -- 绑定上一个频道
        dispatch = recvChannel, -- 设置这个频道的消息处理函数
    }
    channel:subscribe()
    skynet.timeout(500, function() channel:unsubscribe() end) --5秒钟后取消订阅
end)
```

运行结果:

```
$ ./skynet examples/config
multicastpub #终端输入
[:01000010] LAUNCH snlua multicastpub
[:01000012] LAUNCH snlua multicastd
[:01000010] new channel ID 1 #channel ID为1
multicastsub 1 #终端输入
[:01000019] LAUNCH snlua multicastsub 1
[:01000019] channel ID: [Multicast:1] source: :01000010 msg: data5
[:01000019] channel ID: [Multicast:1] source: :01000010 msg: data6
[:01000019] channel ID: [Multicast:1] source: :01000010 msg: data7
[:01000019] channel ID: [Multicast:1] source: :01000010 msg: data8
[:01000019] channel ID: [Multicast:1] source: :01000010 msg: data9 #5秒钟后不再订阅
```

## 8.3 组播原理

只考虑一个进程（一个skynet节点）时，由于同一进程共享地址空间。当发布消息时，由同一节点内的一个 multicastd 服务接收这条消息，并打包成一个 C 数据结构（包括消息指针、长度、引用计数），并把这个结构指针分别发送给同一节点的接收者。

虽然这并没有减少消息的数量、但每个接受者只需要接收一个数据指针。当组播的消息较大时，可以节省内部的带宽。引用计数会在每个接收者收到消息后减一、最终由最后一个接收者销毁。注：如果一个订阅者在收到消息、但没有机会处理它时就退出了。则有可能引起内存泄露。少量的内存泄露影响不大，所以 skynet 没有特别处理这种情况。

当有多个节点时，每个节点内都会启动一个 multicastd 服务。它们通过 DataCenter 相互了解。multicastd 管理了本地节点创建的所有频道。在订阅阶段，如果订阅了一个远程的频道，当前节点的 multicastd 会通知远程频道的管理方，在该频道有发布消息时，把消息内容转发过来。涉及远程组播，不能直接共享指针。这时，multicastd 会将消息完整的发送到接收方所属节点上的同僚，由它来做节点内的组播。

# 9 socket网络服务

## 9.1 skynet.socket 常用api

只需要填写点分十进制的IP地址和端口，简化了一系列地址结构  
体、accept、connect等操作

```
--这样就可以在你的服务中引入这组 api 。
local socket = require "skynet.socket"

--建立一个 TCP 连接。返回一个数字 id，底层使用linux socket api 的connect。
socket.open(address, port)

--关闭一个连接，这个 API 有可能阻塞住执行流。因为如果有其它 coroutine
--正在阻塞读这个 id 对应的连接，会先驱使读操作结束，close 操作才返回。
socket.close(id)

--在极其罕见的情况下，需要粗暴的直接关闭某个连接，而避免 socket.close 的阻塞等待流程，可以使用它。
socket.close_fd(id)

--强行关闭一个连接。和 close 不同的是，它不会等待可能存在的其它 coroutine 的读操作。
--一般不建议使用这个 API，但如果你需要在 __gc 元方法中关闭连接的话，__gc是垃圾回收
--shutdown 是一个比 close 更好的选择（因为在 gc 过程中无法切换 coroutine）。与close_fd类似
socket.shutdown(id)

--[[
    从一个 socket 上读 sz 指定的字节数。
    如果读到了指定长度的字符串，它把这个字符串返回。
    如果连接断开导致字节数不够，将返回一个 false 加上读到的字符串。
    如果 sz 为 nil，则返回尽可能多的字节数，但至少读一个字节（若无新数据，会阻塞）。
--]]
socket.read(id, sz)

--从一个 socket 上读所有的数据，直到 socket 主动断开，或在其它 coroutine 用 socket.close 关闭它。
socket.readall(id)

--从一个 socket 上读一行数据。sep 指行分割符。默认的 sep 为 "\n"。读到的字符串是不包含这个分割符的。
--如果另外一端就关闭了，那么这个时候会返回一个nil，如果buffer中有未读数据则作为第二个返回值返回。
socket.readline(id, sep)

--等待一个 socket 可读。
```

封装了底层的api，这个id可以像文件描述符那样使用，但并不是真正的文件描述符

阻塞

相当于read参数sz是nil

```
socket.block(id)
```

--把一个字符串置入正常的写队列，skynet 框架会在 socket 可写时发送它。

```
socket.write(id, str)
```

--把字符串写入低优先级队列。如果正常的写队列还有写操作未完成时，低优先级队列上的数据永远不会被发出。

--只有在正常写队列为空时，才会处理低优先级队列。但是，每次写的字符串都可以看成原子操作。

--不会只发送一半，然后转去发送正常写队列的数据。

```
socket.lwrite(id, str)
```

--监听一个端口，返回一个 id，供 start 使用。

```
socket.listen(address, port, backlog)
```

监听套接字用于产生新的套接字

一旦有连接过来，没救创建新的套接字进行通信

```
--[[
```

accept 是一个函数。每当一个监听的 id 对应的 socket 上有连接接入的时候，都会调用 accept 函数。这个函数会得到接入连接的 id 以及 ip 地址。你可以做后续操作。

每当 accept 函数获得一个新的 socket id 后，并不会立即收到这个 socket 上的数据。这是因为，我们有时希望把这个 socket 的操作权转让给别的服务去处理。accept(id, addr)

```
]]--
```

socket.start(id, accept)      socket.listen只是创建了套接字，要使用的时候，还要调用socket.start  
注意socket.start和skynet.start的区别

```
--[[
```

任何一个服务只有在调用 socket.start(id) 之后，才可以读到这个 socket 上的数据。向一个 socket id 写数据也需要先调用 start。

socket 的 id 对于整个 skynet 节点都是公开的。也就是说，你可以把 id 这个数字通过消息发送给其它服务，其他服务也可以去操作它。skynet 框架是根据调用 start 这个 api 的位置来决定把对应 socket 上的数据转发到哪里去的。

```
--]]
```

```
socket.start(id)
```

--清除 socket id 在本服务内的数据结构，但并不关闭这个 socket。

--这可以用于你把 id 发送给其它服务，以转交 socket 的控制权。

```
socket.abandon(id)
```

不等于关闭套接字，只是让出控制权，可以给其他服务使用  
其他服务可以通过socket.start，使用这个套接字

```
--[[
```

当 id 对应的 socket 上待发的数据超过 1M 字节后，系统将回调 callback 以示警告。

function callback(id, size) 回调函数接收两个参数 id 和 size，size 的单位是 K。

如果你不设回调，那么将每增加 64K 利用 skynet.error 写一行错误信息。

```
--]]
```

```
socket.warning(id, callback)
```

## 9.2 写一个skynet TCP监听端

示例代码: socketserver.lua

```
local skynet    = require "skynet"
local socket    = require "skynet.socket"

--简单echo服务
function echo(cID, addr)
    socket.start(cID)
    while true do
        local str, endstr = socket.read(cID)
        if str then
            skynet.error("recv " .. str)
            socket.write(cID, string.upper(str))
        else
            socket.close(cID)
            skynet.error(addr .. " disconnect, endstr", endstr)
            return
        end
    end
end

function accept(cID, addr)
    skynet.error(addr .. " accepted")
    skynet.fork(echo, cID, addr) --来一个连接, 就开一个新的协程来处理客户端数据
end

--服务入口
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    --或者 local lID = socket.listen("0.0.0.0", 8001, 128)
    assert(lID)
    socket.start(lID, accept) --把套接字与当前服务绑定
end)
```

运行:

```
$ ./skynet examples/config
socketserver      #终端中手动输入该服务, 例如, 输入socketserver回车
[:01000010] LAUNCH snlua socketserver
[:01000010] listen 0.0.0.0:8001
```

来一个c写的客户端:

示例代码: socketclient.c

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

#define MAXLINE 128
#define SERV_PORT 8001

void* readthread(void* arg)
{
    pthread_detach(pthread_self());
    int sockfd = (int)arg;
    int n = 0;
    char buf[MAXLINE];
    while (1)
    {
        n = read(sockfd, buf, MAXLINE);
        if (n == 0)
        {
            printf("the other side has been closed.\n");
            close(sockfd);
            exit(0);
        }
        else
            write(STDOUT_FILENO, buf, n);
    }
    return (void*)0;
}

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    int sockfd;
    char buf[MAXLINE];
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);
    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    pthread_t thid;
    pthread_create(&thid, NULL, readthread, (void*)sockfd);

    while (fgets(buf, MAXLINE, stdin) != NULL)
        write(sockfd, buf, strlen(buf));
    close(sockfd);
    return 0;
}

```

编译与运行: `gcc client.c -lpthread -o client`

```
$ gcc socketclient.c -lpthread -o socketclient
$ ./socketclient
helloworld      #发送请求
HELLOWORLD      #收到响应
^C              # ctrl+c终止掉客户端
$
```

skynet服务器端显示:

```
socketserver    #终端输入
[:01000010] LAUNCH snlua socketserver
[:01000010] listen 0.0.0.0:8001
[:01000010] 127.0.0.1:41644 accepted #产生请求
[:01000010] recv helloworld          #接受请求, 并响应

[:01000010] 127.0.0.1:41644 disconnect #另一端已经断开连接
```

## 9.3 socket.readline使用

```
local skynet    = require "skynet"
local socket    = require "skynet.socket"

--简单echo服务
function echo(cID, addr)
    socket.start(cID)
    while true do
        local str, endstr = socket.readline(cID)
        --local str, endstr = socket.readline(cID, "\n")
    end
end
```

```

        if str then
            skynet.error("recv " ..str)
            socket.write(cID, string.upper(str))
        else
            socket.close(cID)
            if endstr then
                skynet.error("last recv " ..endstr)
            end
            skynet.error(addr .. " disconnect")
            return
        end
    end
end

function accept(cID, addr)
    skynet.error(addr .. " accepted")
    skynet.fork(echo, cID, addr)
end

--服务入口
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID, accept)
end)

```

## 9.4 socket.readall的使用

```

local skynet    = require "skynet"
local socket    = require "skynet.socket"

--简单echo服务
function echo(cID, addr)
    socket.start(cID)
    local str = socket.readall(cID)

    if str then

```



```

        skynet.error("recv " .. str)
    end
    skynet.error(addr .. " close")
    socket.close(cID)
    return
end

function accept(cID, addr)
    skynet.error(addr .. " accepted")
    skynet.fork(echo, cID, addr)
end

--服务入口
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID, accept)
end)

```

## 9.5 低优先级的发送函数

```

local skynet    = require "skynet"
local socket    = require "skynet.socket"

--简单echo服务
function echo(cID, addr)
    socket.start(cID)
    while true do
        local str = socket.read(cID)
        if str then
            skynet.error("recv " .. str)
            --由于数据量少处理非常快，无法看到效果，只有发送大量数据的时候，才会出现低优先级后发送的现象
            socket.lwrite(cID, "l:" .. string.upper(str))
            socket.write(cID, "h:" .. string.upper(str))
        else
            socket.close(cID)
        end
    end
end

```

```

        skynet.error(addr .. " disconnect")
        return
    end
end
end

function accept(cID, addr)
    skynet.error(addr .. " accepted")
    --如果不开协程，那么同一时刻肯定只能处理一个客户端的连接请求
    skynet.fork(echo, cID, addr)
end

--服务入口
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID, accept)
end)

```

## 9.6 skynet的socket代理服务

我们可以把监听的服务拆分为两个，一个是专门负责监听的服务，一旦有新的连接产生，那么监听的服务会启动一个agent服务，专门用来处理数据请求与应答。这样可以让每个服务分工明确，各司其职，服务拆分成更小的服务也便于书写业务逻辑。

示例代码：socketlisten.lua

```

local skynet    = require "skynet"
local socket    = require "skynet.socket"
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID , function(cID, addr)
        skynet.error(addr .. " accepted")
        skynet.newservice("socketagent", cID, addr)
    end)
end)

```

示例代码: socketagent.lua

```

local skynet    = require "skynet"
local socket    = require "skynet.socket"
function echo(cID, addr)
    socket.start(cID)
    while true do
        local str = socket.read(cID)
        if str then
            skynet.error("recv " ..str)
            socket.write(cID, string.upper(str))
        else
            socket.close(cID)
            skynet.error(addr .. " disconnect")
            return
        end
    end
end
end

local cID, addr = ...
cID = tonumber(cID)

skynet.start(function()
    skynet.fork(function()
        echo(cID, addr)
        skynet.exit()
    end)
end)

```

运行如下:

- 启动socketlisten

```

$ ./skynet examples/config
socketlisten #终端输入
[:01000010] LAUNCH snlua socketlisten
[:01000010] listen 0.0.0.0:8001
[:01000010] 127.0.0.1:41936 accepted #产生新连接
[:01000012] LAUNCH snlua socketagent 9 127.0.0.1:41936 #启动代理服务
[:01000012] recv aaaaaaaaaa #接受请求，并响应

```

- 启动c语言写的socketclient

```

$ ./socketclient
aaaaaaaaaa #stdin输入aaaaaaaaaa
AAAAAAAAAA #收到服务器的应答

```

## 9.7 转交socket控制权

示例代码：socketabandon.lua

```

local skynet = require "skynet"
local socket = require "skynet.socket"
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID, function(cID, addr)
        skynet.error(addr .. " accepted")
        --当前服务开始使用套接字
        socket.start(cID)
        local str = socket.read(cID)

```

```

    if(str) then
        socket.write(cID, string.upper(str))
    end
    --不想使用了, 这个时候遗弃控制权
    socket.abandon(cID)

    skynet.newservice("socketagent", cID, addr) --代理服务不变
end)
end)

```

- 运行：先运行socketabandon，再运行c写的socketclient

skynet服务端输出：

```

$ ./skynet examples/config
socketabandon          #终端输入
[:01000010] LAUNCH snlua socketabandon  #启动socketabandon服务
[:01000010] listen 0.0.0.0:8001
#另外一个终端启动socketclient, 这边接受请求, 并且使用新的socket, read write, 然后遗弃控制权
[:01000010] 127.0.0.1:41950 accepted
[:01000012] LAUNCH snlua socketagent 9 127.0.0.1:41950 #启动代理服务, 把socket控制权交给它
[:01000012] recv bbbbbbbbbbbbbbb

[:01000012] 127.0.0.1:41950 disconnect
[:01000012] KILL self

```

socketclient客户端输出：

```

$ ./socketclient
aaaaaaaaaaaaaaaa
AAAAAAAAAAAAAAAA
bbbbbbbbbbbbbbbb
BBBBBBBBBBBBBBBB
^C

```

## 9.8 skynet的TCP主动连接端

1、有些时候服务要跟其他的外部服务进行交互，那么这个时候skynet的服务会是主动去连接的一端。

2、不仅如此，其实skynet中的两个服务也可以通过socket进行通信。

示例代码：socketclient.lua

```
local skynet    = require "skynet"
local socket    = require "skynet.socket"

function asclient(id)
    local i = 0
    while(i < 3) do
        skynet.error("send data"..i)
        socket.write(id, "data"..i.."\\n")
        local str = socket.readline(id)
        if str then
            skynet.error("recv " .. str)
        else
            skynet.error("disconnect")
        end
        i = i + 1
    end
    socket.close(id) --不主动关闭也行，服务退出的时候，会自动将套接字关闭
    skynet.exit()
end

skynet.start(function()
    local addr = "127.0.0.1:8001"
    skynet.error("connect ".. addr)
    local id = socket.open(addr)
    assert(id)
    --启动读协程
    skynet.fork(asclient, id)
end)
```

监听端代码如下：serverreadline.lua

```
local skynet    = require "skynet"
local socket    = require "skynet.socket"

--简单echo服务
function echo(cID, addr)
    socket.start(cID)
    while true do
        local str = socket.readline(cID)
        if str then
            skynet.fork(function()
                skynet.error("recv " ..str)
                skynet.sleep(math.random(1, 5) * 100)
                socket.write(cID, string.upper(str) .. "\\n")
            end)
        end
    end
end
```

```

        end)
    else
        socket.close(cID)
        skynet.error(addr .. " disconnect")
        return
    end
end
end

function accept(cID, addr)
    skynet.error(addr .. " accepted")
    skynet.fork(echo, cID, addr)
end

--服务入口
skynet.start(function()
    local addr = "0.0.0.0:8001"
    skynet.error("listen " .. addr)
    local lID = socket.listen(addr)
    assert(lID)
    socket.start(lID, accept)
end)

```

运行结果:

```

$ ./skynet examples/config
serverreadline #终端输入
[:01000010] LAUNCH snlua serverreadline #启动监听服务
[:01000010] listen 0.0.0.0:8001
socketclient #终端输入
[:01000012] LAUNCH snlua socketclient #启动另一个充当客户端的服务
[:01000012] connect 127.0.0.1:8001 #客户端服务连接监听服务
[:01000010] 127.0.0.1:44034 accepted #监听服务接受连接
[:01000012] send data0 #依次处理
[:01000010] recv data0
[:01000012] recv DATA0
[:01000012] send data1
[:01000010] recv data1
[:01000012] recv DATA1
[:01000012] send data2
[:01000010] recv data2
[:01000012] recv DATA2
[:01000012] KILL self #客户端服务退出，主动断开连接
[:01000010] 127.0.0.1:44034 disconnect #监听服务也关闭连接

```

- 需要注意的是:

虽然这样也可以让两个服务之间进行通信，但是如果在同一个节点的服务，通信一般就用lua消息来通信就好，毕竟维护套接字的成本远比本地消息调度要高的多。

## 9.9 同时发送多个请求

9.8中的例子我们只能严格按照发送请求，然后等待响应的时序来完成与其他服务的主动交互。现在我们启动多个协程来发请求。

代码如下：socketforkclient.lua

```
local skynet    = require "skynet"
local socket    = require "skynet.socket"

local function recv(id)
    local i = 0
    while(i < 3) do
        local str = socket.readline(id)
        if str then
            skynet.error("recv " .. str)
        else
            skynet.error("disconnect")
        end
        i = i + 1
    end
    socket.close(id)      --未接收完不要关闭
    skynet.exit()
end

local function send(id)   --不用管有没有接受到数据直接发送三次
    local i = 0
    while(i < 3) do
        skynet.error("send data"..i)
        socket.write(id, "data"..i.."\\n")
        i = i + 1
    end
end

skynet.start(function()
    local addr = "127.0.0.1:8001"
```



```

skynet.error("connect ".. addr)
local id = socket.open(addr)
assert(id)
--启动读协程
skynet.fork(recv, id)
--启动写协程
skynet.fork(send, id)
end)

```

运行结果（serverreadline.lua还是使用9.8中的例子）：

```

$ ./skynet examples/config
serverreadline
[:01000010] LAUNCH snlua serverreadline
[:01000010] listen 0.0.0.0:8001
socketforkclient
[:01000012] LAUNCH snlua socketforkclient
[:01000012] connect 127.0.0.1:8001
[:01000010] 127.0.0.1:44072 accepted
[:01000012] send data0          #一次性发完三次命令不用等待
[:01000012] send data1
[:01000012] send data2
[:01000010] recv data0
[:01000010] recv data1
[:01000010] recv data2
[:01000019] recv DATA2        #但是接收回来的数据并不是有序的了
[:01000019] recv DATA0
[:01000019] recv DATA1
[:01000012] KILL self
[:01000010] 127.0.0.1:44072 disconnect

```

可以看到发送请求的一方可以不受响应速度的影响直接发送，但是由于每个请求的处理时间是不一样的，所以接受到的响应信息并不是有序的了。这也是这种模型带来的问题，解决办法就是每个请求发送都携带一个session，接受到的响应信息也携带一个session，那么这样我们就能把请求与响应一一对应，在这一节我们不做扩展了。

与组播的频道不同  
组播的频道是服务间一对多通信，  
传播的是消息  
socketchannel还是一对一，是网络间  
通信

# 10 socketChannel

在与外部服务交互式时，请求回应模式是最常用模式之一。通常的协议设计方式有两种。

1. 每个请求包对应一个回应包，由 TCP 协议保证时序。
2. 发起每个请求时带一个唯一 session 标识，在发送回应时，带上这个标识。这样设计可以不要求每个请求都一定要有回应，且不必遵循先提出的请求先回应的时序。

对于第一种模式，用 skynet 的 Socket API 很容易实现，但如果在一个 coroutine 中读写一个 socket 的话，由于读的过程是阻塞的，这会导致吞吐量下降（前一个回应没有收到时，无法发送下一个请求，9.8我们就是这么设计的）。

对于第二种模式，需要用 skynet.fork 开启一个新线程来收取回应包，并自行和请求对应起来，实现比较繁琐，比如9.9中我们遇到的困惑。

所以skynet 提供了一个更高层的封装： `socket channel`。

## 10.1 第一种模式的socketChannel

示例代码如下：

```
local skynet = require "skynet"
require "skynet.manager"
local sc = require "skynet.socketchannel"

local channel = sc.channel { --创建一个 channel 对象出来，其中 host 可以是 ip 地址或者域名，port 是
    host = "127.0.0.1",
    port = 8001,
}

--接收响应的数据必须这么定义，sock就是与远端的TCP服务相连的套接字,通过这个套接字可以把数据读出来
function response(sock)
    --返回值必须要有两个，第一个如果是true表示响应数据是有效的，
    return true, sock:read() --读到什么就返回什么
end

local function task()
    local resp
```

response函数验证响应是否合法  
(是否是请求向要的响应)  
如果返回false会报错

这是channel:request返回值  
而前面的true是给框架用的

```

local i = 0
while(i < 3) do
    --第一参数是需要发送的请求，第二个参数是一个函数，用来接收响应的数据。
    --调用channel:request会自动连接指定的TCP服务，并且发送请求消息。
    --该函数阻塞，返回读到的内容。
    resp = channel:request("data"..i.."\\n", response)
    skynet.error("recv", resp)
    i = i + 1
end
--channel:close() --channel可以不用关闭，当前服务退出的时候会自动关闭掉
skynet.exit()
end

skynet.start(function()
    skynet.fork(task)
end)

```

看下运行结果（serverreadline.lua是9.8中编写的）：

```

$ ./skynet examples/config
serverreadline #先运行serverreadline.lua
[:01000010] LAUNCH snlua serverreadline
[:01000010] listen 0.0.0.0:8001
channelclient
[:01000012] LAUNCH snlua channelclient
[:01000010] 127.0.0.1:44098 accepted
[:01000010] recv data0
[:01000012] recv DATA0
[:01000010] recv data1
[:01000012] recv DATA1
[:01000010] recv data2
[:01000012] recv DATA2
[:01000012] KILL self
[:01000010] 127.0.0.1:44098 disconnect

```

sock 是由 request 方法传入的一个对象，sock 有两个方法：`read(self, sz)` 和 `readline(self, sep)`。

response 函数的第一个返回值需要是一个 boolean，如果为 true 表示协议解析正常；如果为 false 表示协议出错，这会导致连接断开且让 request 的调用者也获得一个 error。

在 response 函数内的任何异常以及 sock:read 或 sock:readline 读取出错，都会以 error 的形式抛给 request 的调用者。

比如将上面的response函数第一个返回值改为false，运行结果如下：

```

$ ./skynet examples/config
serverreadline
[:01000010] LAUNCH snlua serverreadline
[:01000010] listen 0.0.0.0:8001
channelclient
[:01000012] LAUNCH snlua channelclient
[:01000010] 127.0.0.1:44120 accepted

```

```

[:01000010] recv data0
[:01000012] lua call [0 to :1000012 : 0 msgsz = 24] error : ./lualib/skynet.lua:534:
./lualib/skynet.lua:156: ./lualib/skynet/socketchannel.lua:377: DATA0
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet/socketchannel.lua:377: in function <./lualib/skynet/socketchannel.lua:360>
  (...tail calls...)
  ./my_workspace/channelclient.lua:19: in upvalue 'func'
  ./lualib/skynet.lua:468: in upvalue 'f'
  ./lualib/skynet.lua:106: in function <./lualib/skynet.lua:105>
stack traceback:
  [C]: in function 'assert'
  ./lualib/skynet.lua:534: in function 'skynet.manager.dispatch_message'

```

### 模式一

```
local channel = sc.channel { host , port }
```

```
resp01 = channel:request("data"..i.."\\n", response01)
```

```
resp02 = channel:request("data"..i.."\\n", response02)
```

可以为发送的每一条消息指定验证函数response

### 模式二

```
local channel = sc.channel { host , port , dispatch }
```

```
resp = channel01:request("data"..i.."\\n", session)
```

指定通用的验证函数dispatch

## 10.2 第二种模式的socketChannel

第二种模式需要在 channel 创建时给出一个通用的 response 解析函数。

```

local channel = sc.channel {
  host = "127.0.0.1",
  port = 8002,
  response = dispatch,
}

```

--这里 dispatch 是一个解析回应包的函数，和上面提到的模式 1 中的解析函数类似。但其返回值需要有三个。第一个是这个回应包的 session，第二个是包是否解析正确（同模式 1），第三个是回应内容。

**socket channel 就是依靠创建时是否提供 response 函数来决定工作在模式 1 还是模式 2 下的。**

在模式 2 下，channel.request 的参数有所变化。第 2 个参数不再是 response 函数（它已经在创建时给出），而是一个 session。这个 session 可以是任意类型，但需要和 dispatch 函数返回的类型一致。socket channel 会帮你匹配 session 而让 channel.request 返回正确的值。

示例代码：channelclient2.lua

```

local skynet = require "skynet"
require "skynet.manager"
local sc = require "skynet.socketchannel"

```

返回数据是DATA0

这里从返回的消息中解析出session = 0

函数要在使用之前定义

```

function dispatch(sock)
    local r = sock:readline()
    local session = tonumber(string.sub(r,5))
    return session, true, r
end
--创建一个 channel 对象出来, 其中 host 可以是 ip 地址或者域名, port 是端口号。
local channel = sc.channel {
    host = "127.0.0.1",
    port = 8001,
    response = dispatch
}

```

程序员要从应答中解析出session告诉框架

--返回值必须要有三个, 第一个session

通过有无第三个参数判断是模式一还是模式二

```

local function task()
    local resp
    local i = 0
    while(i < 3) do
        skynet.fork(function(session)
            resp = channel:request("data"..session.."\\n", session)
            skynet.error("recv", resp, session)
        end, i)
        i = i + 1
    end
end

skynet.start(function()
    skynet.fork(task)
end)

```

程序员设定了这个请求的session

运行结果:

channel:request函数是阻塞函数, 完成了对应session的应答的投递  
但是还是没有实现异步请求应答, 因为函数是阻塞的, 函数后面增加请求也还是按顺序处理  
实现异步要靠开协程  
由协程发送请求——一个模式二的channel:request  
同时服务端也要开协程处理, 返回应答  
不然的话服务端顺序应答, 也没有实现异步

```
$ ./skynet examples/config
serverreadline
[:01000010] LAUNCH snlua serverreadline
[:01000010] listen 0.0.0.0:8001
channelclient2
[:01000012] LAUNCH snlua channelclient2
[:01000010] 127.0.0.1:44172 accepted
[:01000010] recv data0
[:01000010] recv data1
[:01000010] recv data2
[:01000012] recv DATA1 1 #能够知道DATA1就是对应session 1的应答
[:01000012] recv DATA2 2
[:01000012] recv DATA0 0
```

## 11 域名查询

在 skynet 的底层，当使用域名而不是 ip 时，由于调用了系统 api `getaddrinfo`，有可能阻塞住整个 socket 线程（不仅仅是阻塞当前服务，而是阻塞整个 skynet 节点的网络消息处理）。虽然大多数情况下，我们并不需要向外主动建立连接。但如果你使用了类似 httpc 这样的模块以域名形式向外请求时，一定要关注这个问题。

skynet 暂时不打算在底层实现非阻塞的域名查询。但提供了一个上层模块来辅助你解决 dns 查询时造成的线程阻塞问题。

```
local dns = require "skynet.dns"
```

在使用前，必须设置 dns 服务器。

- `dns.server(ip, port)`：port 的默认值为 53。如果不填写 ip 的话，将从 `/etc/resolv.conf` 中找到合适的 ip。

指定所使用的DNS服务器

也可以自己在文件中添加一个  
114.114.114.114

## 查询

- `dns.resolve(name, ipv6)`: 查询 `name` 对应的 `ip`，如果 `ipv6` 为 `true` 则查询 `ipv6` 地址，默认为 `false`。如果查询失败将抛出异常，成功则返回 `ip`，以及一张包含所有 `ip` 的 `table`。一个域名可以对应多个IP
- `dns.flush()`: 默认情况下，模块会根据 `TTL` 值 `cache` 查询结果。在查询超时的情况下，也可能返回之前的结果。`dns.flush()` 可以用来清空 `cache`。注意：cache 保存在调用者的服务中，并非针对整个 skynet 进程。所以，推荐写一个独立的 `dns` 查询服务统一处理 `dns` 查询。

节省内存

time to live

网络包中也有一个TTL

TTL=64

64跳，表示一个数据包允许在跳转64个路由

## 11.1 DNS简单使用

示例代码: `test_dns.lua`

```
local skynet = require "skynet"
local dns = require "skynet.dns"

skynet.start(function()
    skynet.error("nameserver:", dns.server()) --设置DNS服务器地址
    -- you can specify the server like dns.server("8.8.4.4", 53)

    local ip, ips = dns.resolve "github.com" --调用成功，则把结果缓存到这个服务的内存中，便于下次使用
    skynet.error("dns.resolve return:", ip)

    for k,v in ipairs(ips) do
        skynet.error("github.com",v)
    end
    dns.flush()
end)
```

运行结果:

```
$ ./skynet examples/config
test_dns
[:01000010] LAUNCH snlua test_dns
[:01000010] nameserver: 127.0.1.1
[:01000010] dns.resolve return: 192.30.255.112 #返回查询到的ip地址
[:01000010] github.com 192.30.255.112
[:01000010] github.com 192.30.255.113
```

## 11.2 封装一个DNS服务

由于每个服务去调用dns接口查询IP时都会在这个服务上缓存一份，下次查询的时候速度就会快很多，但是如果每个服务都保存一份，显示是浪费了资源空间，下面我们来封装用"lua"消息进行查询的DNS服务。

示例代码：dnsservice.lua

```
local skynet = require "skynet"
require "skynet.manager"
local dns = require "skynet.dns"

local command = {}

function command.FLUSH()
    return dns.flush()
end

function command.GETIP(domain)
    return dns.resolve(domain)
end

skynet.start(function()
    dns.server()
    skynet.dispatch("lua", function(session, address, cmd, ...)
        cmd = cmd:upper()
        local f = command[cmd]
        if f then
            skynet.retpack(f(...))
        else
            skynet.error(string.format("Unknown command %s", tostring(cmd)))
        end
    end)
    skynet.register ".dnsservice"
end)
```



测试代码: testdnsservice.lua

```
local skynet = require "skynet"
local cmd, domain = ...
function task()
    local r, ips = skynet.call(".dnsservice", "lua", cmd, domain)
    skynet.error("dnsservice Test:", domain, r)
    skynet.exit()
end
skynet.start(function()
    skynet.fork(task)
end)
```

运行结果:

```
$ ./skynet examples/config
dnsservice
[:01000010] LAUNCH snlua dnsservice
testdnsservice getip www.baidu.com
[:01000012] LAUNCH snlua testdnsservice getip www.baidu.com
[:01000012] dnsservice Test 14.215.177.39
[:01000012] KILL self
```

## 12 snax框架

snax 是一个方便 skynet 服务实现的简单框架。（简单是相对于 skynet 的 api 而言）

使用 snax 服务先要在 Config 中配置 snax 用于路径查找。每个 snax 服务都有一个用于启动服务的名字，推荐按 lua 的模块命名规则，但目前不推荐在服务名中包含"点"（在路径搜索上尚未支持 . 与 / 的替换）。在启动服务时会按查找路径搜索对应的文件。

## 12.1 snax服务基础API

- 启动snax服务的API **snax服务不能再控制台启动，要通过服务去启动**

```
local snax = require "snax"
```

**并不是简单的句柄，而是对象**

**snax.newservice(name, ...)** --可以把一个服务启动多份。传入服务名和参数，它会返回一个对象，用于和这个启动的服务交互。如果多次调用 newservice，即使名字相同，也会生成多份服务的实例，它们各自独立，由不同的对象区分。注意返回的不是服务地址，是一个对象。

**snax.uniqueservice(name, ...)** --和上面 api 类似，但在一个节点上只会启动一份同名服务。如果你多次调用它，会返回相同的对象。

**snax.globalservice(name, ...)** --和上面的 api 类似，但在整个 skynet 网络中（如果你启动了多个节点），只会有一个同名服务。

- 查询snax服务

**snax.queryservice(name)** --查询当前节点的具名服务，返回一个服务对象。如果服务尚未启动，那么一直阻塞等待它启动完毕。

**snax.queryglobal(name)** --查询一个全局名字的服务，返回一个服务对象。如果服务尚未启动，那么一直阻塞等待它启动完毕。

**snax.self()** --用来获取自己这个服务对象，与skynet.self不同，它不是地址。

- snax服务退出

**snax.kill(obj, ...)** --如果你想让一个 snax 服务退出，调用

**snax.exit(...)** --退出当前服务，它等价于 snax.kill(snax.self(), ...)。

- 通过snax服务地址获取snax服务对象

对于匿名服务，你无法在别处通过名字得到和它交互的对象。如果你有这个需求，可以把对象的handle通过消息发送给别人。handle 是一个数字，即 snax 服务的 skynet 服务地址。

```
--把handle转换成服务对象。这里第二个参数需要传入服务的启动名，以用来了解这个服务有哪些远程方法可以供调用。当然，你也可以直接把 .type 域和 .handle 一起发送过去，而不必在源代码上约定。
```

```
snax.bind(handle, typename)
```

- snax启动查找服务路径是config.path的snax变量来指定

```
snax = root.."examples/?lua;"..root.."test/?lua;".."my_workspace/?lua" --添加my_workspace路径
```

## 12.2 最简单snax服务

每个 snax 服务中都需要定义一个 **init** 函数，启动这个服务会调用这个函数，并把启动参数传给它。

snax 服务还可以定义一个 **exit** 函数用于响应服务退出的事件，同样能接收一些参数。

和标准的 skynet 服务不同，这些参数的类型不受限制，可以是 lua 的复杂数据类型。（而 skynet 服务受底层限制，只可以接受字符串参数）

写一个最简单的snax服务simplesnax.lua如下：

```
local skynet = require "skynet"
local snax = require "skynet.snax"

function init( ... )      --snax服务初始化时会调用该回调函数，可以获取到启动参数
    skynet.error ("snax server start:", ...)
end

function exit(...) --snax服务初始化时会调用该回调函数，可以获取到退出参数
    skynet.error ("snax server exit:", ...)
end
```

snax不是普通服务，需要与snax框架配合使用，必须使用snax.newservice、snax.uniqueservice、snax.globalservice这三个函数来启动。

在console服务中启动snax服务时，需要指定为snax：

```
snax simplesnax nengzhong    #输入这一行
[:0100000a] LAUNCH snlua snaxd simplesnax
[:0100000a] snax server start: nengzhong
```

## 12.3 snax服务请求

snax请求分为无响应请求与有响应请求。

对snax服务发请求的方法

对应accept

```
--无响应请求，obj是snax对象，post表示无响应请求，CMD具体的请求命令，...为请求参数列表，发送完马上返回
obj.post.CMD(...)
--有响应请求，obj是snax对象，req表示有响应请求，CMD具体的请求命令，...为请求参数列表，发送完等待响应
obj.req.CMD(...)
```

对应response

### 12.3.1 snax处理无响应请求

修改simplesnax.lua:

```
local skynet = require "skynet"
local snax = require "skynet.snax"

function accept.hello(...) --通过obj.post.hello
    skynet.error("hello", ...)
end

function accept.quit(...) --obj.post.quit来触发回调函数
```

```

snax.exit(...)
--等同snax.kill(snax.self(), ...)
end

function init( ... )
    skynet.error("snax server start:", ...)
end

function exit(...)
    skynet.error("snax server exit:", ...)
end

```

编写testsimplesnax.lua:

```

local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.newservice("simplesnax", 123, "abc", false) --启动simplessnax服务,并传递参数
    skynet.error("snax service", obj, "startup")

    local r = obj.post.hello(123, "abc", false) --调用simplesnax中的accept.hello方法
    skynet.error("hello return:", r)

    obj.post.quit("exit now") --退出服务
end)

```

运行结果:

```

testsimplesnax
[:0100000a] LAUNCH snlua testsimplesnax
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start: 123 abc false
[:0100000a] snax service [simplesnax:100000b] startup
[:0100000a] hello return: nil #post方法没有返回值
[:0100000b] hello 123 abc false
[:0100000b] snax server exit: exit now #服务退出的时候exit函数调用
[:0100000b] KILL self

```

## 12.3.2 处理有响应请求

修改simplesnax.lua

```

local skynet = require "skynet"
local snax = require "skynet.snax"

```

```
function response.echo(str) --当其他服通过obj.req.echo调用的时候,触发该回调函数,并返回应答
    skynet.error("echo", str)
    return str:upper()
end

function init( ... )
    skynet.error("snax server start:", ...)
end

function exit(...)
    skynet.error("snax server exit:", ...)
end
```

修改testsimplesnax.lua

```
local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.newservice("simplesnax", 123, "abc", false)
    skynet.error("snax service", obj, "startup")

    local r = obj.req.echo("nengzhong") --调用simplesnax中的response.echo方法
    skynet.error("echo return:", r)

end)
```

运行结果:

```
testsimplesnax
[:0100000a] LAUNCH snlua testsimplesnax
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start: 123 abc false
[:0100000a] snax service [simplesnax:100000b] startup
[:0100000b] echo nengzhong
[:0100000a] echo return: NENGZHONG #得到返回值
```

## 12.4 snax全局唯一服

修改testsimplesnax.lua:

```
local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.uniqueservice("simplesnax", 123, "abc") --启动simplessnax服务
    obj = snax.queryservice("simplesnax") --查询全局唯一服
    snax.kill(obj, 123, "abc")

    local gobj = snax.globalservice("simplesnax", 123, "abc") --启动simplessnax服务
    gobj = snax.queryglobal("simplesnax") --查询全节点全局唯一服
    snax.kill(gobj, 123, "abc")

    skynet.exit()
end)
```

运行结果:

```
testsimplesnax
[:0100000a] LAUNCH snlua testsimplesnax
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start: 123 abc
[:0100000b] snax server exit: 123 abc
[:0100000b] KILL self
[:0100000c] LAUNCH snlua snaxd simplesnax
[:0100000c] snax server start: 123 abc
[:0100000c] snax server exit: 123 abc
[:0100000c] KILL self
[:0100000a] KILL self
```

## 12.5 snax服务热更

snax 是支持热更新的（只能热更新 snax 框架编写的 lua 服务）。但热更新更多的用途是做不停机的 bug 修复，不应用于常规的版本更新。所以，热更新的 api 被设计成下面这个样子。更适合打补丁。

你可以通过 `snax.hotfix(obj, patchcode)` 来向 obj 提交一个 patch。

### 函数补丁

#### 12.5.1 函数patch

simplesnax.lua

```
local skynet = require "skynet"
local snax = require "skynet.snax"
local i = 10
gname = "nengzhong"
function accept.hello(...) --通过obj.post.hello
    skynet.error("hello", i, gname, ...)
end

function accept.quit(...) --obj.post.quit来触发回调函数
    snax.exit(...)
    --等同snax.kill(snax.self(), ...)
end

function init( ... )
    skynet.error("snax server start:", ...)
end

function exit(...)
    skynet.error("snax server exit:", ...)
end
```

testhotfix.lua

```
local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.newservice("simplesnax") --启动simplessnax服务

    obj.post.hello() --未更新之前调用一次

    local r = snax.hotfix(obj, [[

        function accept.hello(...)

            print("fix hello", i, gname, ...) --skynet.error不能用了
        end
    ]])
end)
```



```

end

]])
skynet.error("hotfix return:", r)

obj.post.hello() --更新之后再调用一次

obj.post.quit() --没更新quit函数，还是能调用

skynet.exit()
end)

```

运行结果：

```

testhotfix
[:0100000a] LAUNCH snlua testhotfix
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start:
[:0100000b] hello 10 nengzhong
fix hello nil nengzhong #热更成功，但是局部变量成了nil，全局变量gname还存在
[:0100000a] hotfix return: nil
[:0100000a] KILL self
[:0100000b] snax server exit: #热更完quit函数还存在
[:0100000b] KILL self

```

## 局部变量补丁

### 12.5.2 local变量patch

上面的结果，我们发现local i不能用了，skynet也不能用了，这是因为local变量没有映射，需要我们自己来映射一下。

修改testhotfix.lua

```

local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.newservice("simplesnax") --启动simplessnax服务

    obj.post.hello() --未更新之前调用一次

    local r = snax.hotfix(obj, [[
        local skynet
        local i
        function accept.hello(...)

```

补丁patch

```

        skynet.error("fix hello", i, gname, ...)
    end
end
]])
skynet.error("hotfix return:", r)

obj.post.hello() --更新之后再调用一次
obj.post.quit() --没更新quit函数，还是能调用

skynet.exit()
end)

```

运行结果：

```

testhotfix
[:0100000a] LAUNCH snlua testhotfix
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start:
[:0100000b] hello 10 nengzhong
[:0100000a] hotfix return: nil
[:0100000a] KILL self
[:0100000b] fix hello 10 nengzhong #变量skynet,i映射成功
[:0100000b] snax server exit:
[:0100000b] KILL self

```

在 patch 中声明的 local skynet 和 local i 在之前的版本中也有同名的 local 变量。snax 的热更新机制会重新映射这些 local 变量。让 patch 中的新函数对应已有的 local 变量，所以你可以安全的继承服务的内部状态，local 变量也可以在申明映射时改变其值。

需要注意的是：全局变量本身就已经映射成功，不要重新申明映射。如果对全局变量重新申明赋值无法改变其值。

## 12.5.3 修改snax服务线上状态

patch 中可以包含一个 **function hotfix(...)** 函数，在 patch 提交后立刻执行。这个函数可以用来查看或修改 snax 服务的线上状态（因为 local 变量会被关联）。hotfix 的函数返回值会传递给 snax.hotfix 的调用者。

修改testhotfix.lua

```

local skynet = require "skynet"
local snax = require "skynet.snax"

skynet.start(function ()
    local obj = snax.newservice("simplesnax") --启动simplessnax服务

    obj.post.hello() --未更新之前调用一次

    local r = snax.hotfix(obj, [[
        local skynet

        local i
    ]])
end)

```

```

function accept.hello(...)
    skynet.error("fix hello", i, gname, ...)
end

function hotfix(...)
    local temp = i
    i = 100
    gname = "nzhsoft"
    return temp
end

]])
skynet.error("hotfix return:", r)

obj.post.hello() --更新之后再调用一次
obj.post.quit() --没更新quit函数，还是能调用
skynet.exit()
end)

```

运行结果：

```

testhotfix
[:0100000a] LAUNCH snlua testhotfix
[:0100000b] LAUNCH snlua snaxd simplesnax
[:0100000b] snax server start:
[:0100000b] hello 10 nengzhong
[:0100000a] hotfix return: 10 #hotfix有返回值
[:0100000a] KILL self
[:0100000b] fix hello 100 nzhsoft #修改了local i的值，也能修改全局变量gname值
[:0100000b] snax server exit:
[:0100000b] KILL self

```

所以，你也可以提交一个仅包含 hotfix 函数的 patch，而不修改任何代码。这样的 patch 通常用于查看 snax 服务的内部状态，或用于修改它们（包括全局变量的值）。

## 13 网关服务

skynet 提供了一个通用模板 `lualib/snax/gateserver.lua` 来启动一个网关服务器，通过 TCP 连接和客户端交换数据。

TCP 基于数据流，但一般我们需要以带长度信息的数据包的结构来做数据交换。gateserver 做的就是这个工作，把数据流切割成包的形式转发到可以处理它的地址。

```
local gateserver = require "snax.gateserver"

local handler = {}          --必须提供一张表，表里面定义connect、message等相关回调函数

-- register handlers here

gateserver.start(handler)  --网关服务的入口函数
```

### 13.1 最简单网关服务

#### 13.1.1 编写mygateserver.lua

示例代码：mygateserver.lua

```
local skynet = require "skynet"
local gateserver = require "snax.gateserver"

local handler = {}

-- 当一个客户端链接进来，gateserver自动处理链接，并且调用该函数，必须要有
function handler.connect(fd, ipaddr)  fd是套接字号码（与文件描述符不一样）
    skynet.error("ipaddr:", ipaddr, "fd:", fd, "connect")
    gateserver.openclient(fd) --链接成功不代表马上可以读到数据，需要打开这个套接字，允许fd接收数据
end
```

```

--当一个客户端断开链接后调用该函数，必须要有
function handler.disconnect(fd)
    skynet.error("fd:", fd, "disconnect")
end

--当fd有数据到达了，会调用这个函数，前提是fd需要调用gateserver.openclient打开
function handler.message(fd, msg, sz)
    skynet.error("recv message from fd:", fd)

end

gateserver.start(handler)

```

还要另一个服务发消息去启动网关服务

### 13.1.2 启动mygateserver

可以使用普通服务创建方式来创建一个mygateserver服务，但是这个服务启动后，并不能马上开始工作，

需要你给mygateserver发送一个lua消息open并且告诉gateserver监听的端口、最大连接数、延时等信息来开启mygateserver服务。

代码如下openmygateserver.lua

```

local skynet = require "skynet"

skynet.start(function()
    skynet.error("Server start")
    local gateserver = skynet.newservice("mygateserver") --启动刚才写的网关服务
    skynet.call(gateserver, "lua", "open", {
        port = 8002, --监听的端口
        maxclient = 64, --客户端最大连接数
        nodelay = true, --是否延迟TCP
    })

    skynet.error("gate server setup on", 8002)
    skynet.exit()
end)

```

体现网关功能的一个重要参数  
延迟：发送少量数据时会缓存等待数据多了再发出去

运行结果：

```
$ ./skynet examples/config
openmygateserver #启动openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver #启动mygateserver
[:01000012] Listen on 0.0.0.0:8002 #开始监听端口8002
[:01000010] gate server setup on 8002
[:01000010] KILL self #openmygateserver退出
```

重开一个终端，启动一个C语言的socketclient客户端（在代码在9.2中）去连接8002端口，观察skynet服务情况：

```
$ ./skynet examples/config
openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:48008 fd: 9 connect #连接进入
[:01000012] fd: 9 disconnect #断开连接
```

上面的结果可以看连接与断开连接都能执行，但是handler.message并没有执行。这是由于snax.gateserver

基于TCP协议包装了一个两字节数据长度的协议。而现在版本的socketclient并没有按照这种协议发送。

## 13.2 gateserver应用协议

### 13.2.1 两字数据长度协议

gateserver应用协议是基于TCP协议做了一层简单的封装，前两个字节表示数据包的长度len（不计算这两个表示长度的字节），高字节在前低字节在后（大端序），后面紧跟len字节数的数据。例如：

```
\x00\x05    \x31\x32\x33\x34\x35
  |          |
  len        data
```

由于只用两字节表示数据长度，那么这个包的data最大只能是65535字节。这种协议包方式可以解决TCP粘包的问题，也是TCP通信当中最常用的一种应用层协议包定义方式。

所以如果想通过TCP与gateserver通信必须要按照这种协议进行组包解包。否则gateserver肯定是不识别的。

## 13.2.2 打包与解包

打包与解包TCP网路数据可以使用skynet.netpack库

```
local netpack = require "skynet.netpack" --使用netpack
--打包数据str，返回一个C指针msg,sz，申请内存
netpack.pack(str)

--解包数据，返回一个lua的字符串，会释放内存
netpack.tostring(msg, sz)
```

## 13.2.3 client使用长度协议发包

下面我们通过改写socketclient.c文件来组包发送数据：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

#define MAXLINE 128

void* readthread(void* arg)
{
    pthread_detach(pthread_self());
    int sockfd = (int)arg;

    int n = 0;
```

```

char buf[MAXLINE];
while (1)
{
    n = read(sockfd, buf, MAXLINE);
    if (n == 0)
    {
        printf("the other side has been closed.\n");
        close(sockfd);
        exit(0);
    }
    else
        write(STDOUT_FILENO, buf, n);
}
return (void*)0;
}

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("usage:%s port", argv[0]);
        return -1;
    }
    int port = atoi(argv[1]);
    struct sockaddr_in servaddr;
    int sockfd;
    short size, nsize;
    char buf[MAXLINE];
    unsigned char sendbuf[MAXLINE];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(port);
    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    pthread_t thid;
    pthread_create(&thid, NULL, readthread, (void*)sockfd);

    while (fgets(buf, MAXLINE, stdin) != NULL)
    {
        size = (short)strlen(buf); //计算需要发送的数据包长度
        nsize = htons(size); //转换成大端序
        memcpy(sendbuf, &nsize, sizeof(nsize)); //nsize先填入sendbuf
        memcpy(sendbuf+sizeof(nsize), buf, size); //再填入buf内容
        write(sockfd, sendbuf, size + sizeof(nsize));
    }
    close(sockfd);
    return 0;
}

```

先运行skynet服务，然后客户端运行：



```
$ gcc socketclient.c -o socketclient -lpthread
./socketclient 8002
```

切回skynet运行的终端:

```
$ ./skynet examples/config
testmygateserver
[:01000010] LAUNCH snlua testmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:48012 fd: 9 connect
[:01000012] recv message from fd: 9 #已经调用handler.message
```

### 13.2.4 gateserver解包

上面的实验中mygateserver.lua中的handler.message已经被调用, 并且告诉fd为9的套接字发来了数据, msg表示C数据指针, sz表示数据长度, 在lua中无法直接使用, 需要转换成lua可识别的数据, 需要调用netpack.tostring函数来解包。下面来改写一下mygateserver.lua:

```
local skynet = require "skynet"
local gateserver = require "snax.gateserver"
local netpack = require "skynet.netpack" --使用netpack

local handler = {}

--当一个客户端链接进来, gateserver自动处理链接, 并且调用该函数
function handler.connect(fd, ipaddr)
    skynet.error("ipaddr:", ipaddr, "fd:", fd, "connect")
    gateserver.openclient(fd)
end

--当一个客户端断开链接后调用该函数
function handler.disconnect(fd)
    skynet.error("fd:", fd, "disconnect")
end

--接收消息
function handler.message(fd, msg, sz)
    skynet.error("recv message from fd:", fd)
    skynet.error(netpack.tostring(msg, sz)) --把 handler.message 方法收到的 msg,sz 转换成一个
lua string, 并释放 msg 占用的 C 内存。
end
```

```
gateserver.start(handler)
```

运行结果:

```
$ ./skynet examples/config
openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:48018 fd: 9 connect
[:01000012] recv message from fd: 9
[:01000012] aaaaaaaaaa #正常接收到数据。

[:01000012] fd: 9 disconnect
```

需要注意的是: `msg` 是一个C指针指向了一块堆空间, 如果你不进行任何处理, 那么也要调用 `skynet.trash` 来释放底层的内存。

如果进行处理, `netpack.tostring` 会释放内存, 返回lua的string

## 13.3 控制客户端连接数

网关服务最重要的任务就是控制客户端连接数, 避免大量客户登录到这个服务上。

修改openmygateserver.lua

```
local skynet = require "skynet"

skynet.start(function()
    skynet.error("Server start")
    local gateserver = skynet.newservice("mygateserver")
    skynet.call(gateserver, "lua", "open", {
```

```

    port = 8002,          --监听的端口
    maxclient = 2,        --客户端最大连接数改为2个
    nodelay = true,       --是否延迟TCP
})

skynet.error("gate server setup on", 8002)
skynet.exit()
end)

```

运行openmygateserver.lua, 再运行三个socketclient, 结果如下:

```

openmygateserver
[:0100000a] LAUNCH snlua openmygateserver
[:0100000a] Server start
[:0100000b] LAUNCH snlua mygateserver
[:0100000b] Listen on 0.0.0.0:8002
[:0100000b] open by :0100000a
[:0100000b] listen on 8002
[:0100000b] client max 2
[:0100000b] nodelay true
[:0100000a] gate server setup on 8002
[:0100000a] KILL self
[:0100000b] ipaddr: 127.0.0.1:46650 fd: 7 connect #第一个客户端连接成功
[:0100000b] ipaddr: 127.0.0.1:46652 fd: 8 connect #第二个客户端连接成功
[:0100000b] fd: 9 disconnect #第三个客户端连接成功后, 马上关闭

```

## 13.4 gateserver其他回调函数

```

--如果你希望在监听端口打开的时候,做一些初始化操作,可以提供 open 这个方法。
--source 是请求来源地址, conf 是开启 gate 服务的参数表(端口,连接数,是否延迟)。
function handler.open(source, conf)
end

--当一个连接异常(通常意味着断开), error 被调用,除了 fd,还会拿到错误信息 msg(通常用于 log 输出)。
function handler.error(fd, msg)
end

--当 fd 上待发送的数据累积超过 1M 字节后,将回调这个方法。你也可以忽略这个消息。
function handler.warning(fd, size)
end

```

代码如下:

mygateserver.lua

```

local skynet = require "skynet"
local gateserver = require "snax.gateserver"
local netpack = require "skynet.netpack"

local handler = {}

--当一个客户端链接进来, gateserver自动处理链接,并且调用该函数
function handler.connect(fd, ipaddr)
    skynet.error("ipaddr:", ipaddr, "fd:", fd, "connect")
    gateserver.openclient(fd)
end

--当一个客户端断开链接后调用该函数
function handler.disconnect(fd)
    skynet.error("fd:", fd, "disconnect")
end

--接收数据
function handler.message(fd, msg, sz)
    skynet.error("recv message from fd:", fd)
    --把 handler.message 方法收到的 msg,sz 转换成一个 lua string,并释放 msg 占用的 C 内存。
    skynet.error(netpack.tostring(msg, sz))
end

--如果报错就关闭该套接字
function handler.error(fd, msg)
    gateserver.closeclient(fd)
end

--fd中待发送数据超过1M时调用该函数,可以不处理
function handler.warning(fd, size)
    skynet.skynet("warning fd=", fd, "unsent data over 1M")
end

```

```
--一旦gateserver打开监听成功后就会调用该接口
--testmygateserver.lua通过给mygateserver.lua发送lua消息open触发该函数调用
function handler.open(source, conf)
    skynet.error("open by ", skynet.address(source))
    skynet.error("listen on", conf.port)
    skynet.error("client max", conf.maxclient)
    skynet.error("nodelay", conf.nodelay)
end

gateserver.start(handler)
```

运行结果:

```
$ ./skynet examples/config
openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000012] open by :01000010
[:01000012] listen on 8002
[:01000012] client max 64
[:01000012] nodelay true
[:01000010] gate server setup on 8002
[:01000010] KILL self
```

## 13.5 给gateserver发送lua消息

gateserver除了能接收socket消息以为，当然也是可以接受skynet的lua消息，并且gateserver还对lua消息注册函数进行了封装，只需提供handler.command回调函数就能处理lua消息，不需要我们自己调用skynet.dispatch来注册。

继续改写mygateserver.lua

```
local skynet = require "skynet"
local gateserver = require "snax.gateserver"
local netpack = require "skynet.netpack"

local handler = {}
local CMD = {}

-- 当一个客户端链接进来, gateserver自动处理链接, 并且调用该函数
function handler.connect(fd, ipaddr)
    skynet.error("ipaddr:", ipaddr, "fd:", fd, "connect")
    gateserver.openclient(fd)
end

-- 当一个客户端断开链接后调用该函数
function handler.disconnect(fd)
    skynet.error("fd:", fd, "disconnect")
end

--
function handler.message(fd, msg, sz)
    skynet.error("recv message from fd:", fd)
    -- 把 handler.message 方法收到的 msg,sz 转换成一个 lua string, 并释放 msg 占用的 C 内存。
    skynet.error(netpack.tostring(msg, sz))
end

function handler.error(fd, msg)
    skynet.closeclient(fd)
end

function handler.warning(fd, size)
    skynet.skynet("warning fd=", fd, "unsend data over 1M")
end

function handler.open(source, conf) --testmygateserver
    skynet.error("open by ", skynet.address(source))
    skynet.error("listen on", conf.port)
    skynet.error("client max", conf.maxclient)
    skynet.error("nodelay", conf.nodelay)
end

function CMD.kick(source, fd)
    skynet.error("source:", skynet.address(source), "kick fd:", fd)
    gateserver.closeclient(fd)
end
```

```
function handler.command(cmd, source, ...)
    local f = assert(CMD[cmd])
    return f(source, ...)
end
-- 调用命令cmd对应的函数

gateserver.start(handler)
```

再编写一个给mygateserver发送命令的服务：kickmygateserver.lua

```
local skynet = require "skynet"

local gateserver, fd= ...
fd = tonumber(fd) -- 必须要转换成整形数, skynet命令行传入的参数都是字符串
skynet.start(function()
    skynet.call(gateserver, "lua", "kick", fd)
    skynet.exit()
end)
```

- 运行

- 1、先运行openmygateserver。
- 2、再在另一个终端运行socketclient。
- 3、然后回到skynet这边启动kickmygateserver关闭连接。

```
$ ./skynet examples/config
openmygateserver    #终端输入
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver    #网关启动
[:01000012] Listen on 0.0.0.0:8002
[:01000012] open by :01000010
[:01000012] listen on 8002
[:01000012] client max 64
[:01000012] nodelay true
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:49038 fd: 9 connect #一个先的客户端连接进来, fd为9
kickmygateserver :01000012 9    #终端输入给网关服务也就是:01000012发送lua消息关闭9号fd
[:01000020] LAUNCH snlua kickmygateserver :01000012 9
[:01000012] source: :01000020 kick fd: 9
[:01000020] KILL self
[:01000012] fd: 9 disconnect    #fd 9关闭收到反馈。
```

## 13.6 open与close两个lua消息

gateserver保留了open与close两个lua消息用来打开关闭监听的端口，所以大家在定义命令的时候不要再使用open与close了。通过这两个消息，我们可以轻松的管理gateserver的开关。

其实在openmygateserver.lua中我们已经发送过一个lua消息open给mygateserver。下面我们来试一试发送lua消息close给mygateserver看看。

代码示例: closemygateserver.lua

```
local skynet = require "skynet"

local gateserver = ...
skynet.start(function()
    skynet.call(gateserver, "lua", "close")
    skynet.exit()
end)
```

先运行openmygateserver再运行closemygateserver:

```
$ ./skynet examples/config
openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000012] open by :01000010
[:01000012] listen on 8002
[:01000012] client max 64
[:01000012] nodelay true
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:49048 fd: 9 connect #可以正常连接
closemygateserver :01000012
[:01000019] LAUNCH snlua closemygateserver :01000012 #发送完CLOSE命令后无法连接新的客户端
[:01000019] KILL self
```



## 13.7 agent服务

前面几节我们讲到gateserver的网络数据的读取，那么是不是也可以通过gateserver来发送网络数据呢？很遗憾gateserver并没有提供相关的写函数，因为gateserver本身就只负责管理网络连接（即TCP连接的处理），涉及到请求处理与答复一般是交给一个叫agent的服务，agent可以由一个普通服务来充当。

来写一个简单的agent.lua服务

```
local skynet = require "skynet"
local netpack = require "skynet.netpack"
local socket = require "skynet.socket"

local client_fd = ...
client_fd = tonumber(client_fd)

skynet.register_protocol {
    name = "client",
    id = skynet.PTYPE_CLIENT,
    --需要将网路数据转换成lua字符串，不需要打包，所以不用注册pack函数
    unpack = netpack.tostring,
}

local function task(msg)
    print("recv from fd", client_fd, msg)
    --响应消息的时候直接通过fd发送出去
    socket.write(client_fd, netpack.pack(string.upper(msg)))
end

skynet.start(function()
    --注册client消息专门用来接收网络数据
    skynet.dispatch("client", function(_,_, msg)
        task(msg)
    end)

    skynet.dispatch("lua", function(_,_, cmd) --注册lua消息，来退出服务
        if cmd == "quit" then
            skynet.error(fd, "agent quit")
            skynet.exit()
        end
    end)
end)
```

改写一下mygateserver.lua，让它处理好一个链接后就创建一个agent服务，并且把fd传给agent，一旦收到数据就转发给agent服务。其他与客户端的交流工作都通过agent来解决。

```
local skynet = require "skynet"
local gateserver = require "snax.gateserver"
local netpack = require "skynet.netpack"
```

```

local handler = {}
local CMD = {}
local agents = {}
--注册client消息专门用来将接收到的网络数据转发给agent,不需要解包,也不需要打包
skynet.register_protocol {
    name = "client",
    id = skynet.PTYPE_CLIENT,
}

function handler.connect(fd, ipaddr)
    skynet.error("ipaddr:", ipaddr, "fd:", fd, "connect")
    gateserver.openclient(fd)
    local agent = skynet.newservice("myagent", fd) --连接成功就启动一个agent来代理
    agents[fd] = agent
end

function handler.disconnect(fd) --断开连接后, agent服务退出
    skynet.error("fd:", fd, "disconnect")
    local agent = agents[fd]
    if(agent) then
        --通过发送消息的方式来退出不要使用skynet.kill(agent)
        skynet.send(agent, "lua", "quit")
        agents[fd] = nil
    end
end

function handler.message(fd, msg, sz) fd中有数据来了会调用这个函数
    local agent = agents[fd]
    skynet.redirect(agent, 0, "client", 0, msg, sz) --收到消息就转发给agent
end

function handler.error(fd, msg)
    skynet.closeclient(fd)
end

function handler.warning(fd, size)
    skynet.skynet("warning fd=", fd, "unsend data over 1M")
end

function handler.open(source, conf)
    skynet.error("open by ", skynet.address(source))
    skynet.error("listen on", conf.port)
    skynet.error("client max", conf.maxclient)
    skynet.error("nodelay", conf.nodelay)
end

function CMD.kick(source, fd)
    skynet.error("source:", skynet.address(source), "kick fd:", fd)
    gateserver.closeclient(fd)
end

function handler.command(cmd, source, ...)

    local f = assert(CMD[cmd])

```

```

    return f(source, ...)
end

gateserver.start(handler)

```

直接在skynet上运行openmygateserver,然后再另一个终端启动socketclient:

```

$ ./skynet examples/config
openmygateserver
[:01000010] LAUNCH snlua openmygateserver
[:01000010] Server start
[:01000012] LAUNCH snlua mygateserver
[:01000012] Listen on 0.0.0.0:8002
[:01000012] open by :01000010
[:01000012] listen on 8002
[:01000012] client max 64
[:01000012] nodelay true
[:01000010] gate server setup on 8002
[:01000010] KILL self
[:01000012] ipaddr: 127.0.0.1:49076 fd: 9 connect #socketclient链接进来
[:01000019] LAUNCH snlua myagent 9 #启动一个新的myagent来处理
recv from fd 9 aaaaaaaaaaaaaa

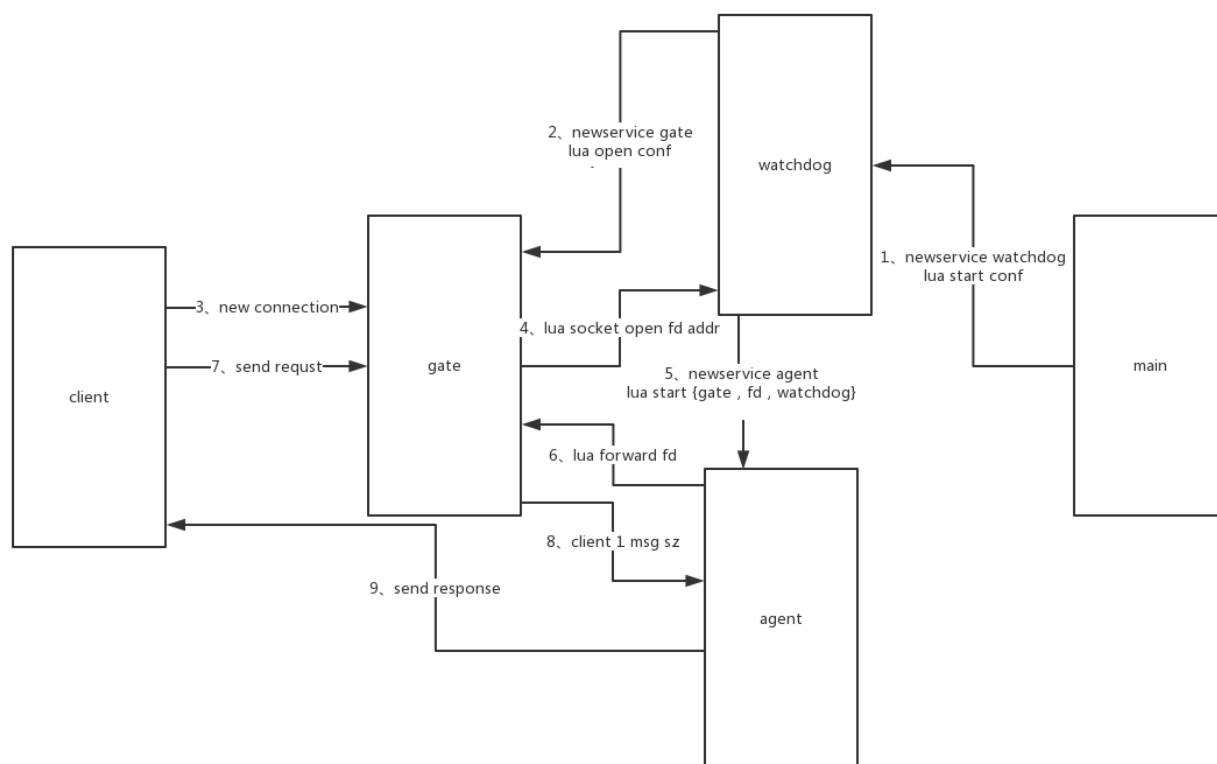
[:01000012] fd: 9 disconnect #socketclient退出后
[:01000019] 9 agent quit #对应的agent服务也退出了
[:01000019] KILL self

```

## 13.8 skynet自带网关服务

skynet有自带的网关服务代码,在service/gate.lua,这个网关代码写的足够好了,完全可以直接使用,使用这个网关服务的例子代码在examples/watchdog.lua,watchdog.lua中使用到examples/agent.lua文件。三个代码需要结合则去看。启动watchdog服务的代码在main.lua。

它们的大致关系：



- 1、newservice watchdog启动看门狗服务，lua start conf给看门狗发送lua消息start参数为conf。
- 2、newservice gate启动网关服务，lua open conf给网关发送lua消息open参数为 conf。
- 3、new connection客户端连接进来。
- 4、lua socket open fd addr给看门狗发送lua消息socket参数为open fd addr。
- 5、newservice agent启动一个客户端代理服务，lua start {gate, fd, watchdog}给代理发送lua消息start参数为 {gate, fd, watchdog}。
- 6、lua forward fd给网关发送lua消息forward参数为fd。（到此，连接建立成功，可以进行网络通信了）
- 7、send request客户端发送请求给看门狗。
- 8、client 1 msg sz 看门狗把请求转发成client消息，session为1。
- 9、send response 代理直接给客户端发送响应。

按照从1到9的依次去查看代码，需要强调的是：

- 1、gate主要负责的是client连接创建与断开以及接受到消息转发给agent。
- 2、watchdog主要负责gate的创建，agent的创建与退出。
- 3、agent主要负责接受gate转发的请求，处理业务，然后直接把应答发送给client。

## 14 登录服务

---

现在的网络游戏大部分是需要登录的，一般会有一个专门的登录服务来处理，登录服务要解决的问题：1、用户登录信息保密工作。2、实际登录点分配工作。

### 14.1 加密算法

---

#### 14.1.1 DHexchange密钥交换算法

DHexchange密钥交换算法主要用来协商一个服务器与客户端的密钥。云风已经帮我们封装好了这个加密方法，可以直接这么使用：

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
```

```
--如果在skynet中使用直接 local crypt = require "skynet.crypt"
```

服务器

```
--dhexchange转换8字节的key
```

```
crypt.dhexchange(key)
```

```
--通过key1与key2得到密钥
```

```
crypt.dhsecret(key1, key2)
```

示例代码testdhexchange.lua:

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"

local clientkey = "11111111" --8byte random
print("clientkey:" , clientkey)
local ckey = crypt.dhexchange(clientkey)
print("ckey:\t" , crypt.hexencode(ckey))

local serverkey = "22222222"
print("serverkey:" , serverkey)

local skey = crypt.dhexchange(serverkey)
print("skey:\t" , crypt.hexencode(skey))

local csecret = crypt.dhsecret(skey, clientkey)
print("use skey clientkey dhsecret:", crypt.hexencode(csecret)) --交换成功

local ssecret = crypt.dhsecret(ckey, serverkey)
print("use ckey serverkey dhsecret:", crypt.hexencode(ssecret)) --交换成功

local ssecret = crypt.dhsecret(ckey, skey) --交换失败
print("use ckey skey dhsecret:\t", crypt.hexencode(ssecret))
```

直接在终端运行结果:

```
$ ./3rd/lua/lua my_workspace/testdhexchange.lua
clientkey: 11111111
ckey:      D5 8A 46 9C FD ED 70 5E
serverkey: 22222222
skey:      B3 60 21 D9 C4 C5 1B 0C
use skey clientkey dhsecret:  95 69 12 B6 88 B3 3B 42 #交换成功
use ckey serverkey dhsecret:  95 69 12 B6 88 B3 3B 42 #交换成功
use ckey skey dhsecret:      C7 19 E5 5F 0A 34 DC E8  #交换不成功
```

需要注意的是, 这个库是独立的库, 不需要在skynet的lua虚拟机里运行, 普通虚拟机也运行使用。

## 14.1.2 随机数

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
--如果在skynet中使用直接 local crypt = require "skynet.crypt"

--产生一个8字节的随机数, 一般作为对称加密算法的随机密钥
crypt.randomkey()
```

## 14.1.3 hmac64哈希算法

hmac64算法主要用于密钥验证。

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
--如果在skynet中使用直接 local crypt = require "skynet.crypt"

--HMAC64运算利用哈希算法, 以一个密钥secret和一个消息challenge为输入, 生成一个消息摘要hmac作为输出。
local hmac = crypt.hmac64(challenge, secret)
```

## 14.1.4 base64编解码

Base64就是一种基于64个可打印字符来表示二进制数据的方法。

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
--如果在skynet中使用直接 local crypt = require "skynet.crypt"

--编码
crypt.base64encode(str)

--解码
crypt.base64decode(str)
```

## 14.1.5 DES加解密

```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
--如果在skynet中使用直接 local crypt = require "skynet.crypt"

--用key加密plaintext得到密文, key必须是8字节
crypt.desencode(key, plaintext)
--用key解密ciphertext得到明文, key必须是8字节
crypt.desdecode(key, ciphertext) 密文会自动补齐到8字节的整数倍
```

### 14.1.6 hashkey算法

云风自实现的hash算法, 只能哈希小于8字节的数据, 返回8字节数据的hash

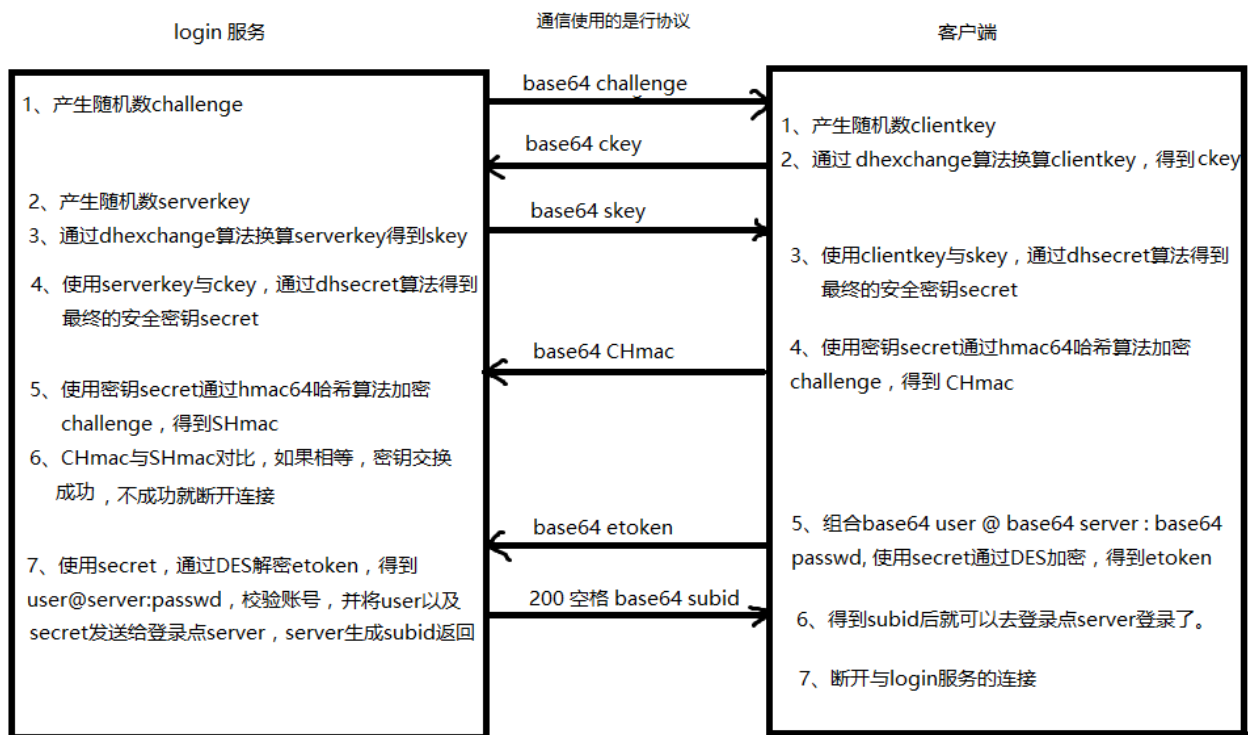
```
package.cpath = "luaclib/?.so"
local crypt = require "client.crypt"
--如果在skynet中使用直接 local crypt = require "skynet.crypt"

--云风自实现的hash算法, 只能哈希小于8字节的数据, 返回8字节数据的hash
crypt.hashkey(str)
```

## 14.2 loginserver原理

skynet 提供了一个通用的登陆服务器模版 `snax.loginserver`。框架原理如下图:





login服务开启监听，客户端主动去连接login服务，他们之间的通信协议是行结尾协议（即：每个数据包都是一行ascii字符，如果要发送byte字节流，则通过base64编码）。这假如称login服务为L，客户端为C。

- (1) L产生随机数challenge，并发送给C，主要用于最后验证密钥secret是否交换成功。
- (2) C产生随机数clientkey，clientkey是保密的，只有C知道，并通过dhexchange算法换算clientkey，得到ckey。把base64编码的ckey发送给L。
- (3) L也产生随机数serverkey，serverkey是保密的，只有L知道，并通过dhexchange算法换算serverkey，得到skey。把base64编码的skey发送给C。
- (4) C使用clientkey与skey，通过dhsecret算法得到最终安全密钥secret。
- (5) L使用serverKey与ckey，通过dhsecret算法得到最终安全密钥secret。C和L最终得到的secret是一样的，而传输过程只有ckey skey是通过网络公开的，即使ckey skey泄露了，也无法推算出secret。
- (6) 密钥交换完成后，需要验证一下双方的密钥是否是一致的。C使用密钥secret通过hmac64哈希算法加密第1步中接收到的challenge，得到CHmac，然后转码成base64 CHmac发送给L。
- (7) L收到CHmac后，自己也使用密钥secret通过hmac64哈希算法加密第1步中发送出去的challenge，得到SHmac，对比SHmac与CHmac是否一致，如果一致，则密钥交换成功。不成功就断开连接。
- (8) C组合base64 user@base64 server:base64 passwd字符串（server为客户端具体想要登录的登录点，远端服务器可能有多个实际登录点），使用secret通过DES加密，得到etoken，发送base64 etoken。
- (9) 使用secret通过DES解密etoken，得到user@server:passwd，校验user与passwd是否正确，通知实际登录点server，传递user与secret给server，server生成subid返回。发送状态码 200 base64 subid给C。
- (10) C得到subid后就可以断开login服务的连接，然后去连接实际登录点server了。（实际登录点server，可以由L通知C，也可以C指定想要登录哪个点，将在下一个章提到）


## 14.3 loginserver 模板

```
local login = require "snax.loginserver"
local server = {
    host = "127.0.0.1",
    port = 8001,
    multilogin = false, -- disallow multilogin
    name = "login_master",
    -- config, etc
}
```

login(server) --服务启动点

- host 是监听地址，通常是 "0.0.0.0"。
- port 是监听端口。
- name 是一个内部使用的名字，不要和 skynet 其它服务重名。在上面的例子，登陆服务器会注册为 `.login_master` 这个名字，相当于 `skynet.register(".login_master")`
- multilogin 是一个 boolean，默认是 false。关闭后，当一个用户正在走登陆流程时，禁止同一用户名进行登陆。如果你希望用户可以同时登陆，可以打开这个开关，但需要自己处理好潜在并行的状态管理问题。

同时，你还需要注册一系列业务相关的必要方法。

--你需要实现这个方法  一个客户端发送过来的 token 做验证。如果验证不能通过，可以通过 error 抛出异常。如果验证通过，需要返回用户希望进入的登陆点以及用户名。（登陆点可以是包含在 token 内由用户自行决定，也可以在这里实现一个负载均衡器来选择）

```
function server.auth_handler(token) end
```

--你需要实现这个方法，处理当用户已经验证通过后，该如何通知具体的登陆点（server）。框架会交给你用户名（uid）和已经安全交换到的通讯密钥。你需要把它们交给登陆点，并得到确认（等待登陆点准备好后）才可以返回。

```
function server.login_handler(server, uid, secret) end
```

--实现command\_handler，用来处理lua消息，必须注册

```
function server.command_handler(command, ...) end
```

登录服务返回给客户端的状态码：

```
200 [base64(subid)] --登录成功会返回一个subid，这个subid是这次登录的唯一标识
400 Bad Request --握手失败
401 Unauthorized --自定义的 auth_handler 不认可 token
403 Forbidden --自定义的 login_handler 执行失败
406 Not Acceptable --该用户已经在登陆中。（只发生在 multilogin 关闭时）
```

## 14.4 运用loginserver模板

示例代码: myloginserver.lua

```
local login = require "snax.loginserver"
local crypt = require "skynet.crypt"
local skynet = require "skynet"

local server = {
    host = "127.0.0.1",
    port = 8001,
    multilogin = false, -- disallow multilogin
    name = "login_master",
}

function server.auth_handler(token)
    -- the token is base64(user)@base64(server):base64(password)
    --通过正则表达式, 解析出各个参数
    local user, server, password = token:match("([^\@]+)@([^\:]+):(.+)")
    user = crypt.base64decode(user)
    server = crypt.base64decode(server)
    password = crypt.base64decode(password)
    skynet.error(string.format("%s@%s:%s", uid, server, password))
    --密码不对直接报错中断当前协程, 千万不要返回nil值, 一定要用assert中断或者error报错终止掉当前协程
    assert(password == "password", "Invalid password")
    return server, user
end

local subid = 0
function server.login_handler(server, uid, secret)
    skynet.error(string.format("%s@%s is login, secret is %s", uid, server,
    crypt.hexencode(secret)))
    subid = subid + 1 --分配一个唯一的subid
    return subid
end

local CMD = {}

function CMD.register_gate(server, address)
    skynet.error("cmd register_gate")
end

--实现command_handler, 必须要实现, 用来处理lua消息
function server.command_handler(command, ...)
    local f = assert(CMD[command])
    return f(...)
end

login(server) --服务启动需要参数
```

示例代码: myclient.lua

```
package.cpath = "luaclib/?.so"

local socket = require "client.socket"
local crypt = require "client.crypt"

if _VERSION ~= "Lua 5.3" then
    error "Use lua 5.3"
end

local fd = assert(socket.connect("127.0.0.1", 8001))

local function writeline(fd, text)
    socket.send(fd, text .. "\n")
end

local function unpack_line(text)
    local from = text:find("\n", 1, true)
    if from then
        return text:sub(1, from-1), text:sub(from+1)
    end
    return nil, text
end

local last = ""

local function unpack_f(f)
    local function try_recv(fd, last)
        local result
        result, last = f(last)
        if result then
            return result, last
        end
        local r = socket.recv(fd)
        if not r then
            return nil, last
        end
        if r == "" then
            error "Server closed"
        end
        return f(last .. r)
    end

    return function()
        while true do
            local result
            result, last = try_recv(fd, last)
            if result then
                return result
            end
            socket.usleep(100)
        end
    end
end
```

```

end
end

local readline = unpack_f(unpack_line)

local challenge = crypt.base64decode(readline()) --接收challenge

local clientkey = crypt.randomkey()
--把clientkey换算后比如称它为ckey，发给服务器
writeline(fd, crypt.base64encode(crypt.dhexchange(clientkey)))
--服务器也把serverkey换算后比如称它为skey，发给客户端，客户端用clientkey与skey所出secret
local secret = crypt.dhsecret(crypt.base64decode(readline()), clientkey)
--secret一般是8字节数据流，需要转换成16字节的hex字符串来显示。
print("secret is ", crypt.hexencode(secret))
--加密的时候还是需要直接传递secret字节流
local hmac = crypt.hmac64(challenge, secret)
writeline(fd, crypt.base64encode(hmac))

local token = {
    server = "sample",
    user = "hello",
    pass = "password",
}

local function encode_token(token)
    return string.format("%s@%s:%s",
        crypt.base64encode(token.user),
        crypt.base64encode(token.server),
        crypt.base64encode(token.pass))
end

--使用DES加密token得到etoken，etoken是字节流
local etoken = crypt.desencode(secret, encode_token(token))
etoken = crypt.base64encode(etoken)
--发送etoken，mylogin.lua将会调用auth_handler回调函数，以及login_handler回调函数。
writeline(fd, etoken)

local result = readline() --读取最终的返回结果。
print(result)
local code = tonumber(string.sub(result, 1, 3))
assert(code == 200)
socket.close(fd) --可以关闭链接了

local subid = crypt.base64decode(string.sub(result, 5)) --解析出subid

print("login ok, subid=", subid)

```

先运行服务器端：

```
$ ./skynet examples/config
mylogin #终端输入
[:01000010] LAUNCH snlua mylogin #会发现默认会启动多个服务
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
```

再运行客户端：

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 57943de9381fce1e
200 MQ==
login ok, subid= 1 #登录成功了
```

回头再来看看服务器输出：

```
$ ./skynet examples/config
mylogin #终端输入
[:01000010] LAUNCH snlua mylogin #会发现默认会启动多个服务
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
[:01000010] login server listen at : 127.0.0.1 8001 #第一个启动的服务去监听
[:01000012] connect from 127.0.0.1:47964 (fd = 9) #有链接来就去处理
[:01000012] hello@sample:password
[:01000010] hello@sample is login, secret is 57943de9381fce1e
```

## 14.5 账户核对失败的处理

账户核对失败，无非是账户密码不匹配时，那么这个时候，我们来观察一下返回客户端的状态码。

在14.4的myclient.lua的基础上修改passwd，再次登录，例如：

```
local token = {  
    server = "sample",  
    user = "hello",  
    pass = "wrongpasswd",  
}
```

客户端运行：

```
$ 3rd/lua/lua my_workspace/myclient.lua  
sceret is 66ec31781d628739  
401 Unauthorized #密码错误，认证不成功返回错误401  
3rd/lua/lua: my_workspace/myclient.lua:88: assertion failed!  
stack traceback:  
  [C]: in function 'assert'  
  my_workspace/myclient.lua:88: in main chunk  
  [C]: in ?
```

服务器端状况：

```
$ ./skynet examples/config  
mylogin  
[:01000010] LAUNCH snlua mylogin  
[:01000012] LAUNCH snlua mylogin  
[:01000019] LAUNCH snlua mylogin  
[:0100001a] LAUNCH snlua mylogin  
[:0100001b] LAUNCH snlua mylogin  
[:0100001c] LAUNCH snlua mylogin  
[:0100001d] LAUNCH snlua mylogin  
[:0100001e] LAUNCH snlua mylogin  
[:0100001f] LAUNCH snlua mylogin  
[:01000010] login server listen at : 127.0.0.1 8001  
[:01000012] connect from 127.0.0.1:48260 (fd = 9)  
[:01000012] hello@sample:passwords  
[:01000010] invalid client (fd = 9) error = ./lualib/snax/loginserver.lua:127:  
./my_workspace/mylogin.lua:20: Invalid password #报错终止掉当前协程。
```

需要注意，一旦有登录请求进来，在调用回调函数 `server.auth_handler` 以及 `server.login_handler` 都是开启了一个协程来处理，`assert`与`error`都能终止掉当前协程，并不是终止掉整个服务。

虽然我们在启动mylogin服务的时候一下启动的了9个服务，但这9个服务中一个是监听使用，其他服务负责与客户端交换密钥以及处理账号验证，八个服务共同分担处理任务，可以通过不断启动客户端来观察。八个服务轮流处理一次登录请求。

例如：运行9次客户端：

```
$ ./skynet examples/config
mylogin
[:01000010] LAUNCH snlua mylogin
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
[:01000010] login server listen at : 127.0.0.1 8001
[:01000012] connect from 127.0.0.1:48268 (fd = 9)
[:01000012] hello@sample:password
[:01000010] hello@sample is login, secret is b7d37b00ed49bf50
[:01000019] connect from 127.0.0.1:48270 (fd = 10)
[:01000019] hello@sample:password
[:01000010] hello@sample is login, secret is 373dc4f4876d7636
[:0100001a] connect from 127.0.0.1:48272 (fd = 11)
[:0100001a] hello@sample:password
[:01000010] hello@sample is login, secret is 9a667284488692b8
[:0100001b] connect from 127.0.0.1:48274 (fd = 12)
[:0100001b] hello@sample:password
[:01000010] hello@sample is login, secret is 19178e716fb886ff
[:0100001c] connect from 127.0.0.1:48276 (fd = 13)
[:0100001c] hello@sample:password
[:01000010] hello@sample is login, secret is 039badb8016f59e6
[:0100001d] connect from 127.0.0.1:48278 (fd = 14)
[:0100001d] hello@sample:password
[:01000010] hello@sample is login, secret is 636e4ed64a797d36
[:0100001e] connect from 127.0.0.1:48280 (fd = 15)
[:0100001e] hello@sample:password
[:01000010] hello@sample is login, secret is e3ad3e8f070fe6c6
[:0100001f] connect from 127.0.0.1:48282 (fd = 16)
[:0100001f] hello@sample:password
[:01000010] hello@sample is login, secret is ee5d95258b470809
[:01000012] connect from 127.0.0.1:48284 (fd = 17)
[:01000012] hello@sample:password
[:01000010] hello@sample is login, secret is bf88c2f81ab14031
```

上面可以看到服务轮流着去处理请求。



## 14.6 login\_handler错误处理

修改14.4中的mylogin.lua的 `skynet.login_handler` 函数：

```
function server.login_handler(server, uid, secret)
    skynet.error(string.format("%s@%s is login, secret is %s", uid, server,
    crypt.hexencode(secret)))
    error("login_handler") --加入这一行,终止掉当前协程
    subid = subid + 1 --分配一个唯一的subid
    return subid
end
```

然后重启mylogin.lua,在启动myclient.lua:

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 312fab4e6cd9908d
403 Forbidden #自定义的 login_handler 执行失败
3rd/lua/lua: my_workspace/myclient.lua:88: assertion failed!
stack traceback:
  [C]: in function 'assert'
  my_workspace/myclient.lua:88: in main chunk
  [C]: in ?
```

服务器端显示:

```
$ ./skynet examples/config
mylogin
[:01000010] LAUNCH snlua mylogin
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
[:01000010] login server listen at : 127.0.0.1 8001
[:01000012] connect from 127.0.0.1:48288 (fd = 9)
[:01000012] hello@sample:password
[:01000010] hello@sample is login, secret is 312fab4e6cd9908d
[:01000010] invalid client (fd = 9) error = ./lualib/snax/loginserver.lua:148:
./my_workspace/mylogin.lua:26: login_handler
```

与skynet.auth\_handler的处理一样，一旦错误，也不需要我们返回任何值，只要终止掉当前协程，skynet.loginserver框架就会自动发送 `406 Not Acceptable` 给客户端。

## 14.7 登录重入报错

在mylogin.lua中，`multilogin` 设置为false表示不允许同时重复登录，这里的同时重复登录不是说一个client登录完成之后，另一个client端使用相同的账号密码就不能登录。而是说在登录过程当中，还没完成登录，这个时候突然又有一个client尝试登录。那么会报给这个客户端 `406 Not Acceptable` 。

由于正常情况下，同时登录比较难模拟，所以我们在login\_handler（不能在auth\_handler）里面添加一个sleep延时5秒钟。例如在14.4的基础上：

```
function server.login_handler(server, uid, secret)
    skynet.error(string.format("%s@%s is login, secret is %s", uid, server,
    crypt.hexencode(secret)))
    subid = subid + 1
    skynet.sleep(500) --添加延时
    return subid
end
```

先运行服务，再运行两个客户端，查看第二客户端的运行结果：

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 1b739592afbcca437
406 Not Acceptable #该用户已经在登陆中
3rd/lua/lua: my_workspace/myclient.lua:88: assertion failed!
stack traceback:
  [C]: in function 'assert'
  my_workspace/myclient.lua:88: in main chunk
  [C]: in ?
```

其实不允许重复登录主要是login\_handler不允许重入，因为如果重入了login\_handler会造成subid分配出现并入。

下面就来模拟一下，允许重入的情况，我们把 `multilogin` 改为true：

先运行服务，再运行两个客户端，查看两个客户端的运行结果：

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 33769a939dc21114
200 Mg==
login ok, subid= 2 #分配到的subid为2
$

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 01588be8f9aeee99
200 Mg==
login ok, subid= 2 #分配到的subid也为2
$
```

所以为了减少这种麻烦事的出现，大家尽量让multilogin为false。

## 14.6 密钥交换失败

密钥交换失败一般不会发生在前几个步骤，因为前几个步骤不会去验证双方的数据是否正确，只要在交换完密钥使用密钥加密challenge的时候才会验证一下，如果这个时候验证不成功将会返回给客户端一个：400 Bad Request。

下面我们就来模拟一下，修改14.4中的myclient.lua

```
local hmac = crypt.hmac64(challenge, secret) --加密的时候还是需要直接传递secret字节流
--改为
local hmac = crypt.hmac64("11111111", secret) --加密的时候还是需要直接传递secret字节流
```

运行服务，再运行myclient：

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is da1f3e758d04fc56
400 Bad Request #握手失败
3rd/lua/lua: my_workspace/myclient.lua:88: assertion failed!
stack traceback:
  [C]: in function 'assert'
  my_workspace/myclient.lua:88: in main chunk
  [C]: in ?
$
```

# 15 msgserver

snax.msgserver 是一个基于消息请求和回应模式的网关服务器模板。它基于 snax.gateserver 定制，可以接收客户端发起的请求数据包，并给出对应的回应。

和 service/gate.lua 不同，用户在使用它的时候，一个用户的业务处理不基于连接。即，它不把连接建立作为用户登陆、不在连接断开时让用户登出。用户必须显式的登出系统，或是业务逻辑设计的超时机制导致登出。

## 15.1 msgserver

和 GateServer 和 LoginServer 一样，snax.msgserver 只是一个模板，你还需要自定义一些业务相关的代码，才是一个完整的服务。与客户端的通信协议使用的是两字节数据长度协议。

### 15.1.1 msgserver api

userid

~~uid, subid, server~~ 把一个登陆名转换为 uid, subid, servername 三元组

```
msgserver.userid(username)
```

---username 把 uid, subid, servername 三元组构造成一个登陆名

```
msgserver.username(uid, subid, server)
```

--你需要在 login\_handler 中调用它，注册一个登陆名username对应的 serect

```
msgserver.login(username, secret)
```

--让一个登陆名失效（登出），通常在 logout\_handler 里调用。

```
msgserver.logout(username)
```

--查询一个登陆名对应的连接的 ip 地址，如果没有关联的连接，会返回 nil 。

```
msgserver.ip(username)
```

## 15.1.2 msgserver服务模板

```
local msgserver = require "snax.msgserver"
local server = {}
```

```
msgserver.start(server) --服务初始化函数，要把server表传递进去。
```

--在打开端口时，会触发这个 register\_handler函数参数name是在配置信息中配置的当前登陆点的名字

--你在这个回调要做的事件是通知登录服务器，我这个登录点准备好了

```
function server.register_handler(name)
end
```

--当一个用户登陆后，登陆服务器会转交给你这个用户的 uid 和 secret，最终会触发 login\_handler 方法。

--在这个函数里，你需要做的是判定这个用户是否真的可以登陆。然后为用户生成一个 subid，使用 msgserver.username(uid, subid, servername) 可以得到这个用户这次的登陆名。这里 servername 是当前登陆点的名字。

--在这个过程中，如果你发现一些意外情况，不希望用户进入，只需要用 error 抛出异常。

```
function server.login_handler(uid, secret)
end
```

--当一个用户想登出时，这个函数会被调用，你可以在里面做一些状态清除的工作。

```
function server.logout_handler(uid, subid)
end
```

--当外界（通常是登陆服务器）希望让一个用户登出时，会触发这个事件。

--发起一个 logout 消息（最终会触发 logout\_handler）

```
function server.kick_handler(uid, subid)
end
```

--当用户的通讯连接断开后，会触发这个事件。你可以不关心这个事件，也可以利用这个事件做超时管理。

--（比如断开连接后一定时间不重新连回来就主动登出。）

```
function server.disconnect_handler(username)
end
```

--如果用户发起了一个请求，就会被这个 request\_handler会被调用。这里隐藏了 session 信息，

--等请求处理完后，只需要返回一个字符串，这个字符串会回到框架，加上 session 回应客户端。

--这个函数中允许抛出异常，框架会正确的捕获这个异常，并通过协议通知客户端。

```
function server.request_handler(username, msg, sz)
end
```

要把启动函数写在这里

## 15.1.3 最简单msgserver

编写一个最简单的simplemsgserver.lua:

```

local msgserver = require "snax.msgserver"
local crypt = require "skynet.crypt"
local skynet = require "skynet"

local subid = 0
local server = {}  --一张表，里面需要实现前面提到的所有回调接口
local servername
--外部发消息来调用，一般用来注册可以登陆的登录名
function server.login_handler(uid, secret)
    skynet.error("login_handler invoke", uid, secret)
    subid = subid + 1
    --通过uid以及subid获得username
    local username = msgserver.username(uid, subid, servername)
    skynet.error("uid",uid, "login, username", username)
    msgserver.login(username, secret)--正在登录，给登录名注册一个secret
    return subid
end

--外部发消息来调用，注销掉登陆名
function server.logout_handler(uid, subid)
    skynet.error("logout_handler invoke", uid, subid)
    local username = msgserver.username(uid, subid, servername)
    msgserver.logout(username)
end

--外部发消息来调用，用来关闭连接
function server.kick_handler(uid, subid)
    skynet.error("kick_handler invoke", uid, subid)
end

--当客户端断开了连接，这个回调函数会被调用
function server.disconnect_handler(username)
    skynet.error(username, "disconnect")
end

--当接收到客户端的请求，这个回调函数会被调用，你需要提供应答。
function server.request_handler(username, msg)
    skynet.error("recv", msg, "from", username)
    return string.upper(msg)
end

--监听成功会调用该函数，name为当前服务别名
function server.register_handler(name)
    skynet.error("register_handler invoked name", name)
    servername = name
end

msgserver.start(server) --需要配置信息

```

## 15.1.4 发送lua消息启动msgserver

要启动msgserver，需要给msgserver发一个lua消息open（msgserver框架已经能处理open消息）例如我们编写一个msgserver的启动msgserver服务，代码startmsgserver.lua：

```
local skynet = require "skynet"

skynet.start(function()
    local gate = skynet.newservice("simplemsgserver")
    --网关服务需要发送lua open来打开，open也是保留的命令
    skynet.call(gate, "lua", "open", {
        port = 8002,
        maxclient = 64,
        servername = "sample", --取名叫sample，跟使用skynet.name(".sample")一样
    })
end)
```

运行结果：

```
startmsgserver
[:0100000a] LAUNCH snlua startmsgserver
[:0100000b] LAUNCH snlua simplemsgserver
[:0100000b] Listen on 0.0.0.0:8002 #开启监听端口
[:0100000b] register_handler invoked name sample #register_handler触发
```

## 15.1.5 发送lua消息给msgserver

sendtomsgserver.lua

```
local skynet = require "skynet"

skynet.start(function()
    local gate = skynet.newservice("simplemsgserver")
    --网关服务需要发送lua open来打开，open也是保留的命令
    skynet.call(gate, "lua", "open", {
        port = 8002,
        maxclient = 64,
        servername = "sample", --取名叫sample，跟使用skynet.name(".sample")一样
    })

    local uid = "nzhsoft"
    local secret = "11111111"
    local subid = skynet.call(gate, "lua", "login", uid, secret) --告诉msgserver，nzhsoft这个用户可以登陆
    skynet.error("lua login subid", subid)
```

```

skynet.call(gate, "lua", "logout", uid, subid) --告诉msgserver, nzhsoft登出

skynet.call(gate, "lua", "kick", uid, subid) --告诉msgserver, 剔除nzhsoft连接

skynet.call(gate, "lua", "close") --关闭gate, 也就是关掉监听套接字

end)

```

运行结果:

```

sendtomsgserver
[:0100000a] LAUNCH snlua sendtomsgserver
[:0100000b] LAUNCH snlua simplemsgserver
[:0100000b] Listen on 0.0.0.0:8002
[:0100000b] register_handler invoked name sample
[:0100000b] login_handler invoke nzhsoft 11111111 #login_handler调用
[:0100000b] uid nzhsoft login, username bnpoc29mdA==@c2FtcGx1#MQ== #login_handler调用成功, 并生成一个登陆名
[:0100000a] lua login subid 1 #返回一个唯一的subid
[:0100000b] logout_handler invoke nzhsoft 1 #登出调用
[:0100000b] kick_handler invoke nzhsoft 1 #kick_handler调用

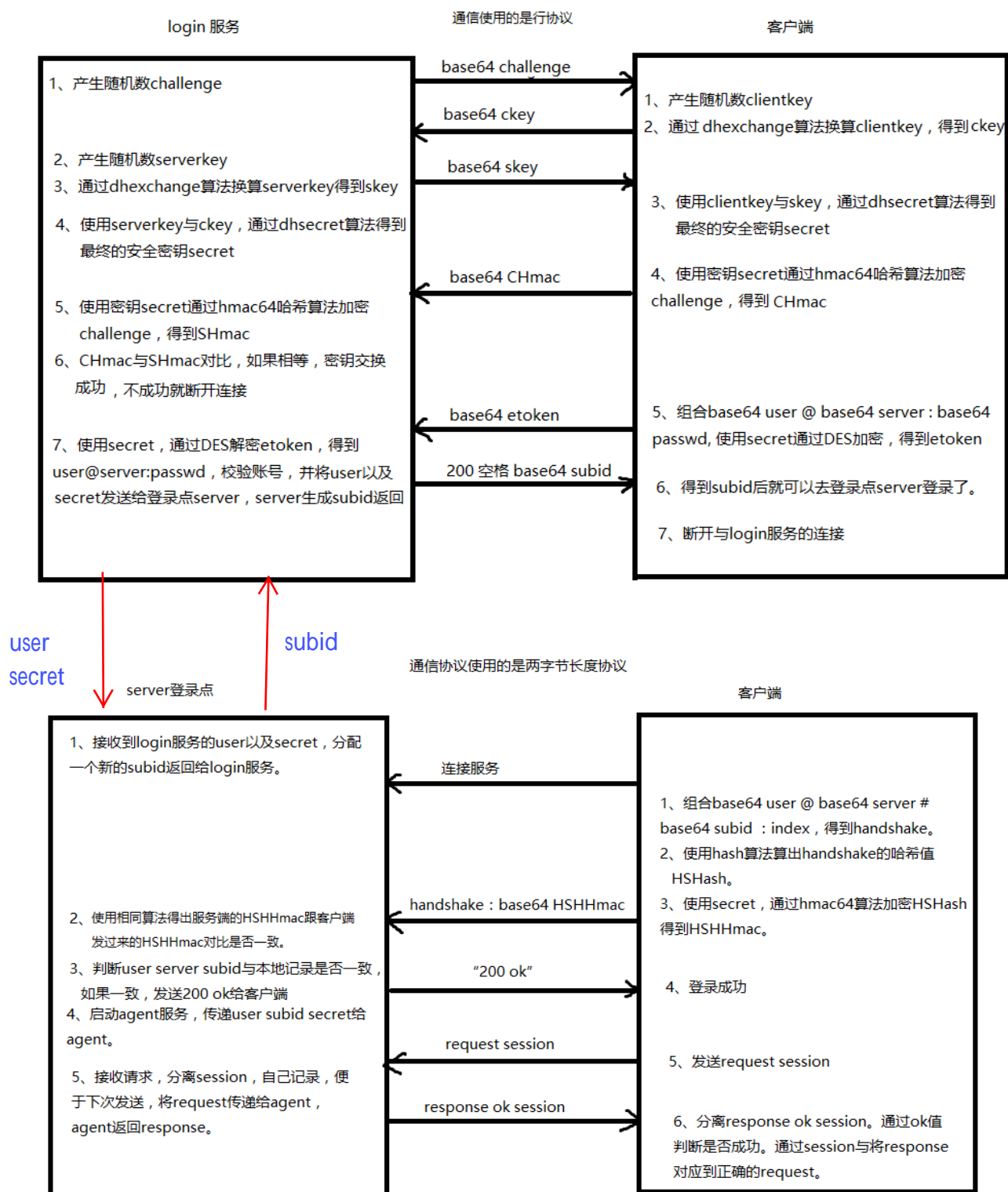
```

## 15.2 loginserver与msgserver

要使用msgserver一般都要跟loginserver一起使用, 下面我们让他们一起工作。

客户端登录的时候, 一般先登录loginserver, 然后再去连接实际登录点, msgserver一般充当真实登录点的角色, 原理图如下:





### 15.2.1 编写一个mymsgserver.lua

```

local msgserver = require "snax.msgserver"
local crypt = require "skynet.crypt"
local skynet = require "skynet"

local loginserver = tonumber(...) --从启动参数获取登录服务的地址
local server = {} --一张表，里面需要实现前面提到的所有回调接口
local servername
local subid = 0

--外部发消息来调用，一般是loginserver发消息来，你需要产生唯一的subid，如果loginserver不允许
multilogin，那么这个函数也不会重入。
function server.login_handler(uid, secret)
    subid = subid + 1
    --通过uid以及subid获得username
    local username = msgserver.username(uid, subid, servername)
    skynet.error("uid",uid, "login, username", username)
    msgserver.login(username, secret)--正在登录，给登录名注册一个secret
    return subid
end

--外部发消息来调用，登出uid对应的登录名
function server.logout_handler(uid, subid)
    local username = msgserver.username(uid, subid, servername)
    msgserver.logout(username) --登出
end

--一般给loginserver发消息来调用，可以作为登出操作
function server.kick_handler(uid, subid)
    server.logout_handler(uid, subid)
end

--当客户端断开了连接，这个回调函数会被调用
function server.disconnect_handler(username)
    skynet.error(username, "disconnect")
end

--当接收到客户端的网络请求，这个回调函数会被调用，需要给与应答
function server.request_handler(username, msg)
    skynet.error("recv", msg, "from", username)
    return string.upper(msg)
end

--注册一下登录点服务，主要是告诉loginserver这个有这个登录点的存在
function server.register_handler(name)
    servername = name
    skynet.call(loginserver, "lua", "register_gate", servername, skynet.self())
end

msgserver.start(server) --需要配置信息表server

```

## 15.2.2 编写一个mylogin.lua

修改14.2中mylogin.lua

```
local login = require "snax.loginserver"
local crypt = require "skynet.crypt"
local skynet = require "skynet"
local server_list = {}

local server = {
    host = "127.0.0.1",
    port = 8001,
    multilogin = false, -- disallow multilogin
    name = "login_master",
}

function server.auth_handler(token)
    -- the token is base64(user)@base64(server):base64(password)
    local user, server, password = token:match("([^\@]+)@([^\:]+):(.+)")
    user = crypt.base64decode(user)
    server = crypt.base64decode(server)
    password = crypt.base64decode(password)
    skynet.error(string.format("%s@%s:%s", user, server, password))
    assert(password == "password", "Invalid password")
    return server, user
end

function server.login_handler(server, uid, secret)
    local msgserver = assert(server_list[server], "unknow server")
    skynet.error(string.format("%s@%s is login, secret is %s", uid, server,
    crypt.hexencode(secret)))
    --将uid以及secret发送给登陆点，告诉登陆点，这个uid可以登陆，并且让登陆点返回一个subid
    local subid = skynet.call(msgserver, "lua", "login", uid, secret)
    return subid --返回给客户端subid，用跟登录点握手使用
end

local CMD = {}

function CMD.register_gate(server, address)
    skynet.error("cmd register_gate")
    server_list[server] = address --记录已经启动的登录点
end

function server.command_handler(command, ...)
    local f = assert(CMD[command])
    return f(...)
end

login(server) --服务启动需要参数
```

### 15.2.3 编写一个testmsgserver.lua来启动他们

```
local skynet = require "skynet"

skynet.start(function()
    --启动mylogin监听8001
    local loginserver = skynet.newservice("mylogin")
    --启动mymsgserver传递loginserver地址
    local gate = skynet.newservice("mymsgserver", loginserver)
    --网关服务需要发送lua open来打开, open也是保留的命令
    skynet.call(gate, "lua", "open", {
        port = 8002,
        maxclient = 64,
        servername = "sample", --取名叫sample, 跟使用skynet.name(".sample")一样
    })
end)
```

运行testmsgserver:

```
$ ./skynet examples/conf
testmsgserver
[:01000010] LAUNCH snlua testmsgserver
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
[:01000020] LAUNCH snlua mylogin
[:01000012] login server listen at : 127.0.0.1 8001
[:01000022] LAUNCH snlua mymsgserver 16777234 #启动msgserver
[:01000022] Listen on 0.0.0.0:8002 #监听8002端口
register_handler      #向loginserver发送登录点注册信息
[:01000012] cmd register_gate #loginserver记录下来登录点的信息
```

### 15.2.4 编写一个myclient.lua

在14.4中的myclient.lua, 只连接了loginserver, 并没有连接具体的登录点, 现在来改写一下myclient.lua

示例代码: myclient.lua

```
package.cpath = "luaclib/?.so"

local socket = require "client.socket"
```

```

local crypt = require "client.crypt"

if _VERSION ~= "Lua 5.3" then
    error "Use lua 5.3"
end

local fd = assert(socket.connect("127.0.0.1", 8001))

local function writeline(fd, text)
    socket.send(fd, text .. "\n")
end

local function unpack_line(text)
    local from = text:find("\n", 1, true)
    if from then
        return text:sub(1, from-1), text:sub(from+1)
    end
    return nil, text
end

local last = ""

local function unpack_f(f)
    local function try_recv(fd, last)
        local result
        result, last = f(last)
        if result then
            return result, last
        end
        local r = socket.recv(fd)
        if not r then
            return nil, last
        end
        if r == "" then
            error "Server closed"
        end
        return f(last .. r)
    end

    return function()
        while true do
            local result
            result, last = try_recv(fd, last)
            if result then
                return result
            end
            socket.usleep(100)
        end
    end
end

local readline = unpack_f(unpack_line)

```

```

local challenge = crypt.base64decode(readline()) --接收challenge

local clientkey = crypt.randomkey()
--把clientkey换算后比如称它为ckey，发给服务器
writeline(fd, crypt.base64encode(crypt.dhexchange(clientkey)))
local secret = crypt.dhsecret(crypt.base64decode(readline()), clientkey)

print("secret is ", crypt.hexencode(secret)) --secret一般是8字节数据流，需要转换成16字节的hex字符串来显示。

local hmac = crypt.hmac64(challenge, secret) --加密的时候需要直接传递secret字节流
writeline(fd, crypt.base64encode(hmac))

local token = {
    server = "sample",
    user = "nzhsoft",
    pass = "password",
}

local function encode_token(token)
    return string.format("%s@%s:%s",
        crypt.base64encode(token.user),
        crypt.base64encode(token.server),
        crypt.base64encode(token.pass))
end

local etoken = crypt.desencode(secret, encode_token(token)) --使用DES加密token得到etoken，etoken是字节流
writeline(fd, crypt.base64encode(etoken)) --发送etoken，mylogin.lua将会调用auth_handler回调函数，以及login_handler回调函数。

local result = readline() --读取最终的返回结果。
print(result)
local code = tonumber(string.sub(result, 1, 3))
assert(code == 200)
socket.close(fd) --可以关闭链接了

local subid = crypt.base64decode(string.sub(result, 5)) --解析出subid

print("login ok, subid=", subid)

----- connect to gate server 新增内容，以下通信协议全是两字节数据长度协议。

local function send_request(v, session) --打包数据v以及session
    local size = #v + 4
    -->I2大端序2字节unsigned int, >I4大端序4字节unsigned int
    local package = string.pack(">I2", size)..v..string.pack(">I4", session)
    socket.send(fd, package)
    return v, session
end

local function recv_response(v)--解包数据v得到content (内容)、ok (是否成功)、session (会话序号)

```

```

    local size = #v - 5
    --cn: n字节字符串 ; B>I4: B unsigned char, >I4, 大端序4字节unsigned int
    local content, ok, session = string.unpack("c"..tostring(size).."B>I4", v)
    return ok ~=0 , content, session
end

local function unpack_package(text)--读取两字节数据长度的包
    local size = #text
    if size < 2 then
        return nil, text
    end
    local s = text:byte(1) * 256 + text:byte(2)
    if size < s+2 then
        return nil, text
    end

    return text:sub(3,2+s), text:sub(3+s)
end

local readpackage = unpack_f(unpack_package)

local function send_package(fd, pack)
    local package = string.pack(">s2", pack) -->大端序, s计算字符串长度, 2字节整形表示
    socket.send(fd, package)
end

local text = "echo"
local index = 1

print("connect")
fd = assert(socket.connect("127.0.0.1", 8002 )) --连接登录点对应的ip端口
last = ""

local handshake = string.format("%s@%s#s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server),crypt.base64encode(subid) , index) --index用于断链恢复
local hmac = crypt.hmac64(crypt.hashkey(handshake), secret) --加密握手hash值得到hmac, 保证
handshake数据接收无误, 没被篡改。
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake

print(readpackage()) --接收应答
print("==>",send_request(text,0)) --发送请求, 同时将当前的session 0组合发送, session用于匹配应答
print("<==",recv_response(readpackage()))
print("disconnect")
socket.close(fd)

```

- 握手包

当一个连接接入后, 第一个包是握手包。握手首先由客户端发起:

--固定握手信息组合

base64(uid)@base64(server)#base64(subid):index:base64(hmac)

--用户名

登录点

断线重登次数

handshake的杂凑值

index 至少是 1，每次连接都需要比之前的大。这样可以保证握手包不会被人恶意截获复用。

## 15.2.5 运行服务与客户端

在服务端运行testmsgserver，再起一个终端运行myclient：

```
$ ./skynet examples/conf
testmsgserver
[:01000010] LAUNCH snlua testmsgserver
[:01000012] LAUNCH snlua mylogin
[:01000019] LAUNCH snlua mylogin
[:0100001a] LAUNCH snlua mylogin
[:0100001b] LAUNCH snlua mylogin
[:0100001c] LAUNCH snlua mylogin
[:0100001d] LAUNCH snlua mylogin
[:0100001e] LAUNCH snlua mylogin
[:0100001f] LAUNCH snlua mylogin
[:01000020] LAUNCH snlua mylogin
[:01000012] login server listen at : 127.0.0.1 8001
[:01000022] LAUNCH snlua mymsgserver 16777234
[:01000022] Listen on 0.0.0.0:8002
register_handler
[:01000012] cmd register_gate sample #loginserver register_gate调用
[:01000019] connect from 127.0.0.1:48822 (fd = 10) #loginserver有新连接产生
[:01000019] nzhsft@sample:password #loginserver author_handler调用
[:01000012] nzhsft@sample is login, secret is 2828d352698fff21 # loginserver login_handler调用
[:01000022] uid nzhsft login, username bnpoc29mdA==@c2FtcGx1#MQ== #msgserver login_handler调用
[:01000022] recv echo from bnpoc29mdA==@c2FtcGx1#MQ== #msgserver接收到消息并且应答
[:01000022] bnpoc29mdA==@c2FtcGx1#MQ== disconnect #断开连接，调用msgserver的logout_handler函数。
```

客户端运行结果：



```

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 2828d352698fff21
200 MQ==
login ok, subid= 1
connect #连接登录点
200 OK #登录成功
==> echo 0 #发送请求
<== true ECHO 0 #收到应答
disconnect #断开连接
$

```

## 15.3 服务握手应答包

msgserver服务给与客户端应答如下：

```

200 OK --成功
404 User Not Found --用户未找到
403 Index Expired --index已经过期了
401 Unauthorized --账号密码校验失败
400 Bad Request --密钥交换失败

```

404与403是这登录msgserver这个登录点的时候可能出现的状态码，剩下的状态码在之前loginserver的时候都已经讲过，这边不再复述。

以上这些包全部是两字节数据长度协议包。例如：

```

\x00\x06 \x32\x30\x30\x20\x4f\x4b
|         |
len      200 ok

```

### 15.3.1 404用户未找到

修改myclient.lua中的handshake数据如下：

```
local handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),  
crypt.base64encode(token.server),crypt.base64encode(subid) , index)  
--改为  
local handshake = string.format("%s@%s#%s:%d", crypt.base64encode("username"),  
crypt.base64encode(token.server),crypt.base64encode(subid) , index) --token.use改成了username
```

这样，登录loginserver时给出的账号是 `nzhsoft`，msgserver中也只在login\_handler中记录的 `nzhsoft` 这个用户，如果客户端使用非 `nzhsoft` 的握手信息登录，就会报一个404

运行myclient.lua

```
$ 3rd/lua/lua my_workspace/myclient.lua  
sceret is 4f9b6da27dc2d414  
200 MQ== #登录loginserver成功  
login ok, subid= 1  
connect  
404 User Not Found #登录点登录失败，状态码为404  
==> echo 0  
3rd/lua/lua: my_workspace/myclient.lua:38: Server closed  
stack traceback:  
[C]: in function 'error'  
my_workspace/myclient.lua:38: in upvalue 'try_recv'  
my_workspace/myclient.lua:46: in local 'readpackage'  
my_workspace/myclient.lua:144: in main chunk  
[C]: in ?  
$
```

### 15.3.2 403 index已过期

403状态码表示index已经过期了，index主要用于防止他们恶意使用handshake来登录，handshake使用后一次必须累加index，例如登录完成后index为1，断线重连，这个时候index=2。如果还是使用之前的index=1

那么就会直接返回403.状态码。

下面我们来再次改写myclient.lua，修改如下：

```

--在末尾添加这几行代码，即使用相同的handshake (index) 不变的情况下，再次尝试登录连接。
print("connect")
fd = assert(socket.connect("127.0.0.1", 8002 )) --连接登录点对应的ip端口
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake

print(readpackage()) --接收应答
print("==>", send_request(text, 0))
print("<===", recv_response(readpackage()))
print("disconnect")
socket.close(fd)

```

运行客户端：

```

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is ecc19383c970bdb9
200 NQ==
login ok, subid= 5
connect
200 OK
==> echo 0
<=== true ECHO 0
disconnect #断开连接
connect #使用相同的handshake重新登录
403 Index Expired #状态码index已经过期
==> echo 0
3rd/lua/lua: my_workspace/myclient.lua:38: Server closed
stack traceback:
 [C]: in function 'error'
 my_workspace/myclient.lua:38: in upvalue 'try_recv'
 my_workspace/myclient.lua:46: in local 'readpackage'
 my_workspace/myclient.lua:154: in main chunk
 [C]: in ?
$

```

### 15.3.3 断线重连

- 如果想要断线重连使用当前subid恢复连接，需要给index+1，重新计算handshake，需要这么改写myclient.lua

```

--第二次连接
print("connect")
fd = assert(socket.connect("127.0.0.1", 8002 )) --连接登录点对应的ip端口
index = index + 1      --index加一
handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server),crypt.base64encode(subid) , index) --重新计算handshake
hmac = crypt.hmac64(crypt.hashkey(handshake), secret) --
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake

print(readpackage()) --接收应答
print("==>",send_request(text,0))
print("<==",recv_response(readpackage()))
print("disconnect")
socket.close(fd)

```

运行结果:

```

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 8878e738e831e1f0
200 NQ==
login ok, subid= 5
connect
200 OK
==> echo 0
<== true ECHO 0
disconnect
connect #重新连接
200 OK #连接成功
==> echo 0
<== true ECHO 0
disconnect
$

```

## 15.4 请求与应答

- 请求包发送给msgserver的，但是除了遵循两字节数据长度协议外，数据内容还需要遵循以下规则：

len	request	session
两字节长度	请求内容	四字节sessionID

由于需要msgserver的请求应答并不需要同步，可以是多个请求一起发送，不用等上一个应答到了，才请求下一个，为了把请求与应答对应起来，就需要添加一个sessionID。整个数据包如下：

--发送"12345" sessionID为1、组合好的数据包如下：

```
\x00\x09 \x31\x32\x33\x34\x35 \x00\x00\x00\x01
```

- 应答包收到后需要解析，需要遵循以下规则来解析：

len	response	ok	session
两字节长度	响应内容	一字节状态值	四字节sessionID

例如：

--应答返回"12345"

```
\x00\x0a \x31\x32\x33\x34\x35 \x01 \x00\x00\x00\x01
```

### 15.4.1 获取最后一次返回

- 如果发送完请求后，还未等到响应就断开连接了，断线重连后，想获取最后一次的返回，这可以这么写客户端代码：

```
local text = "echo"
local index = 1

print("connect")
fd = assert(socket.connect("127.0.0.1", 8002)) --连接登录点对应的ip端口
last = ""

local handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server), crypt.base64encode(subid) , index) --index用于断链恢复
local hmac = crypt.hmac64(crypt.hashkey(handshake), secret) --加密握手hash值得到hmac，最要是保证
handshake数据接收无误
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake
```

```

print(readpackage()) --接收应答
print("==>", send_request(text, 0))      --session 0 的请求发送出去
--print("<==", recv_response(readpackage())) --不接收应答就断开连接
print("disconnect")
socket.close(fd)

--断线重连
print("connect")
fd = assert(socket.connect("127.0.0.1", 8002))
last = ""
index = index + 1
local handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server), crypt.base64encode(subid) , index)
local hmac = crypt.hmac64(crypt.hashkey(handshake), secret)

send_package(fd, handshake .. ":" .. crypt.base64encode(hmac))
print(readpackage())

print("==>", send_request("fake", 0))      --伪装session0请求，再发送出去一次，发送内容可以随便填
print("==>", send_request("again", 1))     --发送请求again, session+1.
print("<==", recv_response(readpackage()))
print("<==", recv_response(readpackage()))

print("disconnect")
socket.close(fd)

```

运行结果：

```

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 2986b5b04a669ea7
200 Ng==
login ok, subid= 6
connect
200 OK
==> echo 0 #发送完请求，不接收
disconnect
connect
200 OK
==> fake 0 #假装session 0的发送，
==> again 1
<== true ECHO 0 #应答返回并没有返回fake，而是之前的echo
<== true AGAIN 1 #使用session1的发送请求就能得到正常的应答
disconnect
$

```

上面可以看到，想得到以后一次的响应，就把任意的请求内容和最后一次的对应的session组合再发送一次。

## 15.4.2 获取历史应答

只要是对应的session已经发送过了，就能获取到响应。

代码如下：

```
print("connect")
fd = assert(socket.connect("127.0.0.1", 8002)) --连接登录点对应的ip端口
last = ""

local handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server),crypt.base64encode(subid) , index) --index用于断链恢复
local hmac = crypt.hmac64(crypt.hashkey(handshake), secret) --加密握手hash值得到hmac，最要是保证
handshake数据接收无误
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake

print(readpackage()) --接收应答
print("==>", send_request(text,0)) --发送两次
print("==>", send_request(text,1))
--print("<==", recv_response(readpackage())) --不管是否已经接受了
--print("<==", recv_response(readpackage()))
print("disconnect")
socket.close(fd)

print("connect")
fd = assert(socket.connect("127.0.0.1", 8002))
last = ""
index = index + 1
handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server),crypt.base64encode(subid) , index)
hmac = crypt.hmac64(crypt.hashkey(handshake), secret)

send_package(fd, handshake .. ":" .. crypt.base64encode(hmac))

print(readpackage())
print("==>", send_request("fake",0)) -- request again (use last session 0, so the request
message is fake)
print("==>", send_request("again",1)) -- request again (use new session)
print("<==", recv_response(readpackage()))
print("<==", recv_response(readpackage()))

print("disconnect")
socket.close(fd)
```

运行结果：

```
$ 3rd/lua/lua my_workspace/myclient.lua
sceret is 6b679b5ac461ce45
200 MjU=
```

```

login ok, subid=    25
connect
200 OK
===>    echo      0
===>    echo      1
disconnect
connect
200 OK
===>    fake      0
===>    again     1
<===    true      ECHO    0    #获取到是上面的echo 0 的返回
<===    true      ECHO    1    #获取到的是上面echo 1 的返回
disconnect

```

### 15.4.3 服务应答异常

在服务mymsgserver中的接收到请求后，会自动剥离协议中的len以及session，得到请求内容，如果在处理请求内容的时候，应答出现异常，那么会返回ok的值为0.

- 例如，在mymsgserver.lua中的request\_handler添加一行 `error("request_handler")` 运行结果：

```

$ 3rd/lua/lua my_workspace/myclient.lua
sceret is  a572bd350328d80d
200 MQ==
login ok, subid=    1
connect
200 OK
===>    echo      0
disconnect
connect
200 OK
===>    fake      0
===>    again     1
<===    false     0    #返回false 并且没有响应内容
<===    false     1    #返回false 并且没有响应内容
disconnect

```



## 15.5 agent服务

一般网关服务登录完毕后，会启动一个agent服务来专门处理客户端的请求。下面我们来写一个mymsgagent.lua

```
local skynet = require "skynet"

skynet.register_protocol {
    name = "client",
    id = skynet.PTYPE_CLIENT,
    unpack = skynet.tostring,
}

local gate
local userid, subid

local CMD = {}

function CMD.login(source, uid, sid, secret) --登录成功, secret可以用来加解密数据
    -- you may use secret to make a encrypted data stream
    skynet.error(string.format("%s is login", uid))
    gate = source
    userid = uid
    subid = sid
    -- you may load user data from database
end

local function logout() --退出登录, 需要通知gate来关闭连接
    if gate then
        skynet.call(gate, "lua", "logout", userid, subid)
    end
    skynet.exit()
end

function CMD.logout(source)
    -- NOTICE: The logout MAY be reentry
    skynet.error(string.format("%s is logout", userid))
    logout()
end

function CMD.disconnect(source) --gate发现client的连接断开了, 会发disconnect消息过来这里不要登出
    -- the connection is broken, but the user may back
    skynet.error(string.format("disconnect"))
end

skynet.start(function()
    -- If you want to fork a work thread , you MUST do it in CMD.login
    skynet.dispatch("lua", function(session, source, command, ...)
        local f = assert(CMD[command])
        skynet.ret(skynet.pack(f(source, ...)))
    end)

    skynet.dispatch("client", function(_,_, msg)
```

```

        skynet.error("recv:", msg)
        skynet.ret(string.upper(msg))
        if(msg == "quit")then --一旦收到的消息是quit就退出当前服务，并且关闭连接
            logout()
        end
    end)
end)
end)

```

修改mymsgserver.lua

```

local msgserver = require "snax.msgserver"
local crypt = require "skynet.crypt"
local skynet = require "skynet"

local loginserver = tonumber(...) --从启动参数获取登录服务的地址
local server = {} --一张表，里面需要实现前面提到的所有回调接口
local servername
local subid = 0
local agents = {}

function server.login_handler(uid, secret)
    subid = subid + 1
    local username = msgserver.username(uid, subid, servername)--通过uid以及subid获得username
    skynet.error("uid",uid, "login, newusername", username)
    msgserver.login(username, secret)--正在的登录
    agent = skynet.newservice("mymsgagent")
    skynet.call(agent, "lua", "login", uid, subid, secret)
    agents[username] = agent
    return subid
end

--一般给agent调用
function server.logout_handler(uid, subid)
    local username = msgserver.username(uid, subid, servername)
    msgserver.logout(username) --登出
    skynet.call(loginserver, "lua", "logout",uid, subid) --通知一下loginserver已经退出
    agents[username] = nil
end

--一般给loginserver调用
function server.kick_handler(uid, subid)
    local username = msgserver.username(uid, subid, servername)
    local agent = agents[username]
    if agent then
        --这里使用pcall来调用skynet.call避免由于agent退出造成异常发生
        pcall(skynet.call, agent, "lua", "logout") --通知一下agent，让它退出服务。
    end
end
end

```

```

--当客户端断开了连接, 这个回调函数会被调用
function server.disconnect_handler(username)
    skynet.error(username, "disconnect")
end

--当接收到客户端的请求, 跟gateserver一样需要转发这个消息给agent, 不同的是msgserver还需要response返回值
--, 而gateserver并不负责这些事
function server.request_handler(username, msg)
    skynet.error("recv", msg, "from", username)
    --返回值必须是字符串, 所以不管之前的数据是否是字符串, 都转换一遍
    return skynet.toString(skynet.rawcall(agents[username], "client", msg))
end

--注册一下登录点服务, 主要是告诉loginservice这个登录点
function server.register_handler(name)
    servername = name
    skynet.call(loginservice, "lua", "register_gate", servername, skynet.self())
end

msgserver.start(server) --需要配置信息, 跟gateserver一样, 端口、ip, 外加一个登录点名称

```

## 修改mylogin.lua

```

local login = require "snax.loginservice"
local crypt = require "skynet.crypt"
local skynet = require "skynet"
local server_list = {}
local login_users = {}

local server = {
    host = "127.0.0.1",
    port = 8001,
    multilogin = false, -- disallow multilogin
    name = "login_master",
}

function server.auth_handler(token)
    -- the token is base64(user)@base64(server):base64(password)
    local user, server, password = token:match("(^[^@]+)@([^\:]+):(.+)") --通过正则表达式, 解析出各个参数
    user = crypt.base64decode(user)
    server = crypt.base64decode(server)
    password = crypt.base64decode(password)
    skynet.error(string.format("%s@%s:%s", user, server, password))
    assert(password == "password", "Invalid password")
    return server, user
end

function server.login_handler(server, uid, secret)
    local msgserver = assert(server_list[server], "unknow server")

    skynet.error(string.format("%s@%s is login, secret is %s", uid, server,

```

```

crypt.hexencode(secret)))
    local last = login_users[uid]
    if last then --判断是否登录, 如果已经登录了, 那就退出之前的登录
        skynet.call(last.address, "lua", "kick", uid, last.subid)
    end

    local id = skynet.call(msgserver, "lua", "login", uid, secret) --将uid以及secret发送给登陆点,
    让它做好准备, 并且返回一个subid
    login_users[uid] = { address=msgserver, subid=id}
    return id
end

local CMD = {}

function CMD.register_gate(server, address)
    skynet.error("cmd register_gate")
    server_list[server] = address
end

function CMD.logout(uid, subid) --专门用来处理登出的数据清除, 用户信息保存等
    local u = login_users[uid]
    if u then
        print(string.format("%s@%s is logout", uid, u.server))
        login_users[uid] = nil
    end
end

function server.command_handler(command, ...)
    local f = assert(CMD[command])
    return f(...)
end

login(server) --服务启动需要参数

```

修改myclient.lua

```

package.cpath = "luaclib/?.so"

local socket = require "client.socket"
local crypt = require "client.crypt"

if _VERSION ~= "Lua 5.3" then
    error "Use lua 5.3"
end

local fd = assert(socket.connect("127.0.0.1", 8001))

local function writeline(fd, text)
    socket.send(fd, text .. "\n")
end

local function unpack_line(text)

```

```

    local from = text:find("\n", 1, true)
    if from then
        return text:sub(1, from-1), text:sub(from+1)
    end
    return nil, text
end

local last = ""

local function unpack_f(f)
    local function try_recv(fd, last)
        local result
        result, last = f(last)
        if result then
            return result, last
        end
        local r = socket.recv(fd)
        if not r then
            return nil, last
        end
        if r == "" then
            error "Server closed"
        end
        return f(last .. r)
    end

    return function()
        while true do
            local result
            result, last = try_recv(fd, last)
            if result then
                return result
            end
            socket.usleep(100)
        end
    end
end

local readline = unpack_f(unpack_line)

local challenge = crypt.base64decode(readline()) --接收challenge

local clientkey = crypt.randomkey()
writeline(fd, crypt.base64encode(crypt.dhexchange(clientkey))) --把clientkey换算后比如称它为
ckey, 发给服务器
local secret = crypt.dhsecret(crypt.base64decode(readline()), clientkey) --服务器也把serverkey换算
后比如称它为skey, 发给客户端, 客户端用clientkey与skey所出secret

print("secret is ", crypt.hexencode(secret)) --secret一般是8字节数据流, 需要转换成16字节的hex字符串
来显示。

local hmac = crypt.hmac64(challenge, secret) --加密的时候还是需要直接传递secret字节流

writeline(fd, crypt.base64encode(hmac))

```

```

local token = {
    server = "sample",
    user = "nzhsoft",
    pass = "password",
}

local function encode_token(token)
    return string.format("%s@%s:%s",
        crypt.base64encode(token.user),
        crypt.base64encode(token.server),
        crypt.base64encode(token.pass))
end

local etoken = crypt.desencode(secret, encode_token(token)) --使用DES加密token得到etoken, etoken
是字节流
writeline(fd, crypt.base64encode(etoken)) --发送etoken, mylogin.lua将会调用auth_handler回调函数,
以及login_handler回调函数。

local result = readline() --读取最终的返回结果。
print(result)
local code = tonumber(string.sub(result, 1, 3))
assert(code == 200)
socket.close(fd) --可以关闭链接了

local subid = crypt.base64decode(string.sub(result, 5)) --解析出subid

print("login ok, subid=", subid)

----- connect to game server 新增内容, 以下通信协议全是两字节数据长度协议。

local function send_request(v, session) --打包数据v以及session
    local size = #v + 4
    local package = string.pack(">I2", size)..v..string.pack(">I4", session)
    socket.send(fd, package)
    return v, session
end

local function recv_response(v)--解包数据v得到content (内容)、ok (是否成功)、session (会话序号)
    local size = #v - 5
    local content, ok, session = string.unpack("c"..tostring(size)..">I4", v)
    return ok ~= 0, content, session
end

local function unpack_package(text)--解析两字节数据长度协议包
    local size = #text
    if size < 2 then
        return nil, text
    end
    local s = text:byte(1) * 256 + text:byte(2)
    if size < s+2 then
        return nil, text
    end

```

```

end

return text:sub(3,2+s), text:sub(3+s)
end

local readpackage = unpack_f(unpack_package)

local function send_package(fd, pack)
    local package = string.pack(">s2", pack)
    socket.send(fd, package)
end

local text = "echo"
local index = 1
local session = 0

print("connect")
fd = assert(socket.connect("127.0.0.1", 8002 )) --连接登录点对应的ip端口
last = ""

local handshake = string.format("%s@%s#%s:%d", crypt.base64encode(token.user),
crypt.base64encode(token.server),crypt.base64encode(subid) , index) --index用于断链恢复
local hmac = crypt.hmac64(crypt.hashkey(handshake), secret) --加密握手hash值得到hmac, 最要是保证
handshake数据接收无误
send_package(fd, handshake .. ":" .. crypt.base64encode(hmac)) --发送handshake

print(readpackage()) --接收应答

while(true) do
    text = socket.readstdin() --循环读取标准数据发送给服务器
    if text then
        print("==>",send_request(text,session))
        print("<==",recv_response(readpackage()))
        session = session + 1 --会话ID自动递增
    end
    socket.usleep(100)
end

print("disconnect")
socket.close(fd)

```

先运行服务testmsgserver, 再运行两个myclient, 运行结果:

```

$ ./skynet examples/conf
testmsgserver
[:0100000a] LAUNCH snlua testmsgserver
[:0100000b] LAUNCH snlua mylogin
[:0100000c] LAUNCH snlua mylogin
[:0100000d] LAUNCH snlua mylogin
[:0100000e] LAUNCH snlua mylogin
[:0100000f] LAUNCH snlua mylogin

[:01000010] LAUNCH snlua mylogin

```

```
[ :01000012] LAUNCH snlua mylogin
[ :01000013] LAUNCH snlua mylogin
[ :01000014] LAUNCH snlua mylogin
[ :0100000b] login server listen at : 127.0.0.1 8001
[ :01000015] LAUNCH snlua mymsgserver 16777227
[ :01000015] Listen on 0.0.0.0:8002
[ :0100000b] cmd register_gate
[ :0100000c] connect from 127.0.0.1:51882 (fd = 8)
[ :0100000c] nzhssoft@sample:password
[ :0100000b] nzhssoft@sample is login, secret is c4e4d6986346fca2
[ :01000015] uid nzhssoft login, newusername bnpoc29mdA==@c2FtcGx1#MQ==
[ :01000016] LAUNCH snlua mymsgagent #启动了一个新的agent来处理nzhssoft用户的请求
[ :01000016] nzhssoft is login
[ :01000015] recv aaaaaaaaaa from bnpoc29mdA==@c2FtcGx1#MQ==
[ :01000016] recv: aaaaaaaaaa
[ :01000015] recv quit from bnpoc29mdA==@c2FtcGx1#MQ==
[ :01000016] recv: quit #收到quit消息就退出服务
[ :0100000b] nzhssoft@nil is logout
[ :01000016] KILL self
```

## 16 mysql

skynet封装了mysql的驱动，主要文件为 lualib/skynet/db/mysql.lua。

先在ubuntu下安装mysql

```
sudo apt-get install mysql-server
```

设置mysql的用户密码为root 123456，并且创建一个skynet数据库。



## 16.1 连接mysql

连接mysql的API

```
local skynet = require "skynet"
local mysql = require "skynet.db.mysql" --引入模块
--连接成功db返回非nil
local db=mysql.connect({
    host="127.0.0.1",
    port=3306,
    database="skynet",
    user="root",
    password="123456",
    max_packet_size = 1024 * 1024, --数据包最大字节数
    on_connect = on_connect      --连接成功的回调函数
})

--关闭连接
db:disconnect()
```

示例代码connectmysql.lua:

```
local skynet = require "skynet"
local mysql = require "skynet.db.mysql"

skynet.start(function()
    local function on_connect(db)
        skynet.error("on_connect")
    end
    local db=mysql.connect({
        host="127.0.0.1",
        port=3306,
        database="skynet",

        user="root",
```

```

        password="123456",
        max_packet_size = 1024 * 1024,
        on_connect = on_connect
    })
    if not db then
        skynet.error("failed to connect")
    else
        skynet.error("success to connect to mysql server")
    end

    db:disconnect() --关闭连接
end)

```

在mysql中先创建一个skynet数据库然后再运行：

```

connectmysql
[:0100000a] LAUNCH snlua connectmysql
[:0100000a] on_connect
[:0100000a] success to connect to mysql server

```

## 16.2 执行SQL语句

执行SQL语句可以使用db:query(sql)，参数sql可以填任何你想要执行的SQL语句。

示例代码querymysql.lua

```

local skynet = require "skynet"
local mysql = require "skynet.db.mysql"

local function dump(res, tab)
    tab = tab or {}
    if(tab == {}) then
        skynet.error(".....dump.....")
    end
end

```

```

if type(res) == "table" then
    skynet.error(string.rep("\t", tab).."{")
    for k,v in pairs(res) do
        if type(v) == "table" then
            dump(v, tab + 1)
        else
            skynet.error(string.rep("\t", tab), k, "=", v, ",")
        end
    end
    skynet.error(string.rep("\t", tab).."}")
else
    skynet.error(string.rep("\t", tab) , res)
end
end

skynet.start(function()
    local function on_connect(db)
        skynet.error("on_connect")
    end
    local db=mysql.connect({
        host="127.0.0.1",
        port=3306,
        database="skynet",
        user="root",
        password="123456",
        max_packet_size = 1024 * 1024,
        on_connect = on_connect
    })
    if not db then
        skynet.error("failed to connect")
        skynet.exit()
    else
        skynet.error("success to connect to mysql server")
    end

    --设置utf8字符集
    local res = db:query("set charset utf8");
    dump(res)

    --删除数据表
    res = db:query("drop table if exists dogs")
    dump(res)

    --创建数据表
    res = db:query("create table dogs (id int primary key,name varchar(10))")
    dump(res)

    --插入数据
    res = db:query("insert into dogs values (1, \'black\'), (2, \'red\)")

    dump(res)

```

```

--查询数据
res = db:query("select * from dogs")
dump(res)

--多条语句查询
res = db:query("select * from dogs; select * from dogs")
dump(res)

--查询错误
res = db:query("select * from noexist;")
dump(res)

db:disconnect() --关闭连接
skynet.exit()
end)

```

运行结果:

```

querymysql
[:0100000a] LAUNCH snlua querymysql
[:0100000a] on_connect
[:0100000a] success to connect to mysql server
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]   server_status = 2 , #状态2表示成功
[:0100000a]   warning_count = 0 ,
[:0100000a]   affected_rows = 0 ,
[:0100000a]   insert_id = 0 ,
[:0100000a] }
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]   server_status = 2 ,
[:0100000a]   warning_count = 0 ,
[:0100000a]   affected_rows = 0 ,
[:0100000a]   insert_id = 0 ,
[:0100000a] }
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]   server_status = 2 ,
[:0100000a]   warning_count = 0 ,
[:0100000a]   affected_rows = 0 ,
[:0100000a]   insert_id = 0 ,
[:0100000a] }
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]   message = &Records: 2 Duplicates: 0 Warnings: 0 ,
[:0100000a]   warning_count = 0 ,
[:0100000a]   server_status = 2 ,
[:0100000a]   affected_rows = 2 ,
[:0100000a]   insert_id = 0 ,
[:0100000a] }

```

```

[:0100000a] .....dump.....
[:0100000a] {                                     #查询数据, 返回一张表
[:0100000a]     {
[:0100000a]         id = 1 ,
[:0100000a]         name = black ,
[:0100000a]     }
[:0100000a]     {
[:0100000a]         id = 2 ,
[:0100000a]         name = red ,
[:0100000a]     }
[:0100000a] }
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]     {
[:0100000a]         {
[:0100000a]             id = 1 ,
[:0100000a]             name = black ,
[:0100000a]         }
[:0100000a]         {
[:0100000a]             id = 2 ,
[:0100000a]             name = red ,
[:0100000a]         }
[:0100000a]     }
[:0100000a]     {
[:0100000a]         {
[:0100000a]             id = 1 ,
[:0100000a]             name = black ,
[:0100000a]         }
[:0100000a]         {
[:0100000a]             id = 2 ,
[:0100000a]             name = red ,
[:0100000a]         }
[:0100000a]     }
[:0100000a] }
[:0100000a] multiresultset = true , #多条sql语句执行结果中multiresultset为true
[:0100000a] }
[:0100000a] .....dump.....
[:0100000a] {
[:0100000a]     errno = 1146 ,
[:0100000a]     badresult = true ,
[:0100000a]     sqlstate = 42S02 ,
[:0100000a]     err = Table 'skynet.noexist' doesn't exist ,
[:0100000a] }
[:0100000a] KILL self

```

## 16.3 db:query的调度情况

db:query通过给mysql服务端发送sql语句，并且阻塞等待mysql服务端返回结果，这个过程当中，db.query会自动让出当前协程的执行权，等待skynet的下次调度。

来看一个例子：

testquerymysql.lua

```
local skynet = require "skynet"
local mysql = require "skynet.db.mysql"

local function test( db , name)
    local i=1
    while true do
        local res = db:query("select * from dogs") --每个协程十秒查询两遍
        skynet.error(name, "loop times=", i, res)

        res = db:query("select * from dogs")
        skynet.error(name, "loop times=", i, res)

        skynet.sleep(1000)
        i=i+1
    end
end

skynet.start(function()
    local function on_connect(db)
        skynet.error("on_connect")
    end
    local db=mysql.connect({
        host="127.0.0.1",
        port=3306,
        database="skynet",
        user="root",
        password="123456",

        max_packet_size = 1024 * 1024,
```

```

        on_connect = on_connect
    })
    if not db then
        skynet.error("failed to connect")
        skynet.exit()
    else
        skynet.error("success to connect to mysql server")
    end
    skynet.fork(test, db, "test0")
    skynet.fork(test, db, "test1")
    skynet.fork(test, db, "test2")
end)

```

运行结果:

```

$ ./skynet examples/config
testquiremysql
[:0100000a] LAUNCH snlua testquiremysql
[:0100000a] on_connect
[:0100000a] success to connect to mysql server
[:0100000a] test0 loop times= 1 table: 0x7fc0e4a937c0 #协程test0的两次查询并没有都执行完, 因为
db.query会让出协程执行权限
[:0100000a] test1 loop times= 1 table: 0x7fc0e4a93a80
[:0100000a] test2 loop times= 1 table: 0x7fc0e4a93d40
[:0100000a] test0 loop times= 1 table: 0x7fc0e4b07040
[:0100000a] test1 loop times= 1 table: 0x7fc0e4b07300
[:0100000a] test2 loop times= 1 table: 0x7fc0e4b075c0
[:0100000a] test0 loop times= 2 table: 0x7fc0e4afd680
[:0100000a] test1 loop times= 2 table: 0x7fc0e4afd940
[:0100000a] test2 loop times= 2 table: 0x7fc0e4afdc00
[:0100000a] test0 loop times= 2 table: 0x7fc0e4afdec0
[:0100000a] test1 loop times= 2 table: 0x7fc0e4b081c0
[:0100000a] test2 loop times= 2 table: 0x7fc0e4b08480

```

上面开了三个协程每十秒分别调用两次db.query, 但是test0协程并没有一次性把两次调用给执行完, 而是调用完一次db.query阻塞让出执行权, 然后test1调用阻塞让出执行权, test2调用也阻塞让出执行权, 三个协程都在等待资源, 这个时候资源到达是通过skynet框架来通知, 再分别上test0、test1、test2从db.query返回。

# 17 protobuf

假如我们要建立的skynet服务器与客户端的连接方式为长连接，且选择了Google的Protobuf来定制我们的网络协议，那么，接下来我们要解决的问题就是：**如何在skynet框架中使用socket+protobuf。**

由于protobuf的lua版本的支持存在着部分缺陷，为了避免踩坑，这里我们直接使用云风博客中推荐的 pbc 动态proto解析库。

## 17.1 安装PBC

1、下载 pbc：跟下载skynet源码一样，通过 git 将 pbc 的源码克隆到本地：

```
$ cd skynet/3rd/  
$ git clone https://github.com/cloudwu/pbc.git
```

2、编译安装：

```
$ cd pbc  
$ make
```

3、注意如果报如下错，表示protobuf未安装，没有报错就跳到第6步

```
make: protoc: 命令未找到  
Makefile:79: recipe for target 'build/addressbook.pb' failed  
make: *** [build/addressbook.pb] Error 127
```

4、安装protobuf

```
$ sudo apt-get install protobuf-c-compiler protobuf-compiler  
$ protoc --version
```

5、再次编译



```
$ make
```

## 6、工具编译

```
$ cd ./binding/lua53  
$ sudo make
```

## 7、如果报错如下，没有就跳到第9步

```
$ make  
gcc -O2 -fPIC -Wall -shared -o protobuf.so -I../.. -I/usr/local/include -L../..../build pbc-lua.c  
-lpbc  
pbc-lua.c:4:17: fatal error: lua.h: 没有那个文件或目录  
compilation terminated.  
Makefile:11: recipe for target 'protobuf.so' failed  
make: *** [protobuf.so] Error 1  
$
```

## 8、上面的错误是因为没有安装 lua5.3 skynet本身已经有lua5.3，只不过路径没指定，修改makefile如下

```
CC = gcc  
CFLAGS = -O2 -fPIC -Wall  
LUADIR = ../../../../lua #这个路径就是skynet/3rd/lua  
TARGET = protobuf.so  
  
.PHONY : all clean  
  
all : $(TARGET)  
  
$(TARGET) : pbc-lua53.c  
    $(CC) $(CFLAGS) -shared -o $@ -I../.. -I$(LUADIR) -L../..../build $^ -lpbc  
  
clean :  
    rm -f $(TARGET)
```

## 9、编译成功的话，将 protobuf.so 放在config文件中 lua\_cpath 项配置的目录下面，同时将 protobuf.lua 放在config文件 lua\_path 配置的目录下，就可以调用protobuf中的库方法

```
$ cp protobuf.so ../../../../luaclib/  
$ cp protobuf.lua ../../../../luaclib/
```

## 17.2 生成protobuffer文件

1、先在项目根目录下创建一个 `protos` 文件夹，用来存放协议文件，比如创建一个 `Person.proto` 协议文件，内容如下：

```
$ cd skynet
$ mkdir protos
$ cd protos
$ vi test.proto #你也可以取他名字myname.proto
```

test.proto

教程中使用proto2，建议使用proto3

```
package cs; //定义包名
message test { //定义消息结构
    required string name = 1; //name为string类型，并且是必须的，1表示第一个字段
    required int32 age = 2; //age为int32类型，并且是必须的
    optional string email = 3; //email为string类型，并且是可选的
    required bool online = 4; //online为bool类型，并且是必须的
    required double account = 5; //account为double类型，并且是必须的
}
```

syntax = "proto3"

```
package cs;
message test {
    string name = 1;
    int32 age = 2;
    string email = 3;
    bool online = 4;
    double account = 5;
}
```

required 修饰的字段如果没有指定值，将采用默认值填充；

optional修饰的字段如果没有指定值，直接为空；

2、将协议文件解析成 `.pb` 格式：

```
$ protoc --descriptor_set_out=test.pb test.proto
```

lua-protobuf的命令

```
protoc -o addressbook.pb addressbook.proto
```

## 17.3 使用protobuffer文件

使用.pb文件

```

local protobuf = require "protobuf" --引入文件protobuf.lua
--注册protobuf文件
protobuf.register_file (protofile) .pb文件

--根据注册的protofile中的类定义进行序列化, 返回得到一个stringbuffer
protobuf.encode("package.message", { ... }) 使用哪一个包中的哪一个消息类型对消息进行序列化

--根据注册的protofile中的类定义进行反序列化
protobuf.decode("package.message", stringbuffer)

```

示例代码: testpb.lua

```

local skynet = require "skynet"
local protobuf = require "protobuf" --引入文件protobuf.lua

skynet.start(function()
    protobuf.register_file "./protos/test.pb"
    skynet.error("protobuf register: test.pb")

    stringbuffer = protobuf.encode("cs.test", --对应person.proto协议的包名与类名
    {
        name = "xiaoming",
        age = 1,
        --email = "xiaoming@163.com",
        online = true,
        account = 888.88,
    })

    local data = protobuf.decode("cs.test",stringbuffer)
    skynet.error("-----decode----- \nname=",data.name
        , "\nage=", data.age
        , "\nemail=", data.email)
        , "\nonline=", data.online
        , "\naccount=", data.account)

end)

```

运行结果:

```

$ ./skynet examples/config
testpb
[:01000010] LAUNCH snlua testpb
[:01000010] protobuf register: test.pb
[:01000010] -----decode-----
name= xiaoming ,
age= 1 ,
email= ,
online= true ,
account= 888.88

```

## 17.4 编写一个稍微复杂点pb服务

person.proto

```
package cs;
message Person {
    required string name = 1;    //Person第一个字段name为string类型，并且是必须的
    required int32 id = 2;
    optional string email = 3;    //Person第三个字段email为string类型，并且是可选的

    enum PhoneType {            //定义一个枚举类型
        MOBILE = 0;              不是字段
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {        //再定义一个消息类型PhoneNumber不是字段
        required string number = 1;    //PhoneNumber第一个字段number为String类型，并且是必须的
        optional PhoneType type = 2 [default = HOME];    //第二个字段type为PhoneType类型，可选的
    }

    repeated PhoneNumber phone = 4;    //Person第四个字段phone为PhoneNumber类型，是可重复的，相当于是数组
}
```

将协议文件解析成 .pb 格式：

```
$ protoc --descriptor_set_out=person.pb person.proto
```

示例代码 testpb.lua

```
local skynet = require "skynet"
local protobuf = require "protobuf" --引入文件protobuf.lua
```

```

skynet.start(function()
    protobuf.register_file "./protos/person.pb"
    skynet.error("protobuf register: person.pb")

    stringbuffer = protobuf.encode("cs.Person", --对应person.proto协议的包名与类名
    {
        name = "xiaoming",
        id = 1,
        email = "xiaoming@163.com",
        phone = {
            {
                number = "13888888888",
                type = "MOBILE",
            },
            {
                number = "88888888",
            },
            {
                number = "87878787",
                type = "WORK",
            },
        }
    })

    local data = protobuf.decode("cs.Person", stringbuffer)
    skynet.error("decode name="..data.name..",id="..data.id..",email="..data.email)
    skynet.error("decode
phone.type="..data.phone[1].type..",phone.number="..data.phone[1].number)
    skynet.error("decode
phone.type="..data.phone[2].type..",phone.number="..data.phone[2].number)
    skynet.error("decode
phone.type="..data.phone[3].type..",phone.number="..data.phone[3].number)
end)

```

运行结果:

```

$ ./skynet examples/conf
testpbcc
[:01000010] LAUNCH snlua testpbcc
[:01000010] protobuf register: person.pb
[:01000010] decode name=xiaoming,id=1,email=xiaoming@163.com
[:01000010] decode phone.type=MOBILE,phone.number=13888888888
[:01000010] decode phone.type=HOME,phone.number=88888888
[:01000010] decode phone.type=WORK,phone.number=87878787

```

## 17.5 protobuf数据类型

标量类型列表

proto类型	C++类型	备注
double	double	
float	float	
int32	int32	使用可变长编码，编码负数时不够高效——如果字段可能含有负数，请使用 sint32
int64	int64	使用可变长编码，编码负数时不够高效——如果字段可能含有负数，请使用 sint64
uint32	uint32	使用可变长编码
uint64	uint64	使用可变长编码
sint32	int32	使用可变长编码，有符号的整型值，编码时比通常的int32高效
sint64	int64	使用可变长编码，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
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	string	可能包含任意顺序的字节数据

# 18 http协议的服务

http协议的服务分为http服务端与http客户端，skynet服务作为http服务端的时候，可以像其他的web服务器一样，接收http请求并给与应答。skynet的服务作为http客户端的时候，可以通过http协议像远端发送请求并等待得到应答。

## 18.1http服务端

skynet 从 v0.5.0 开始提供了简单的 http 服务器的支持。skynet.httpd 是一个独立于 skynet 的，用于 http 协议解析的库，它本身依赖 socket api 的注入。使用它，你需要把读写 socket 的 API 封装好，注入到里面就可以工作。

skynet.sockethelper 模块将 skynet 的 Socket API 封装成 skynet.httpd 可以接受的形式：阻塞读写指定的字节数、网络错误以异常形式抛出。

下面是一个简单的范例：testhttpd.lua

```
local skynet = require "skynet"
local socket = require "skynet.socket"

skynet.start(function()
    local agent = {}
    for i= 1, 20 do
        -- 启动 20 个代理服务用于处理 http 请求
        agent[i] = skynet.newservice("testhttpagent")
    end
    local balance = 1
    -- 监听一个 web 端口
    local id = socket.listen("0.0.0.0", 8001)
    socket.start(id , function(id, addr)
        -- 当一个 http 请求到达的时候，把 socket id 分发到事先准备好的代理中去处理。
        skynet.error(string.format("%s connected, pass it to agent :%08x", addr,
agent[balance]))
        skynet.send(agent[balance], "lua", id)
```

```

        balance = balance + 1
        if balance > #agent then
            balance = 1
        end
    end)
end)

```

http代理服务代码: testagenthttp.lua

```

-- examples/simpleweb.lua

local skynet = require "skynet"
local socket = require "skynet.socket"
local httpd = require "http.httpd"
local sockethelper = require "http.sockethelper"
local urllib = require "http.url"
local string = string

local function response(id, ...)
    local ok, err = httpd.write_response(sockethelper.writefunc(id), ...)
    if not ok then
        -- if err == sockethelper.socket_error , that means socket closed.
        skynet.error(string.format("fd = %d, %s", id, err))
    end
end

skynet.start(function()
    skynet.dispatch("lua", function (_,_,id)
        socket.start(id) -- 开始接收一个 socket
        -- limit request body size to 8192 (you can pass nil to unlimit)
        -- 一般的业务不需要处理大量上行数据, 为了防止攻击, 做了一个 8K 限制。这个限制可以去掉。
        local code, url, method, header, body = httpd.read_request(sockethelper.readfunc(id),
8192)
        if code then
            if code ~= 200 then -- 如果协议解析有问题, 就回应一个错误码 code 。
                response(id, code)
            else
                -- 这是一个示范的回应过程, 你可以根据你的实际需要, 解析 url, method 和 header 做出回
应。

                if header.host then
                    skynet.error("header host", header.host)
                end

                local path, query = urllib.parse(url)
                skynet.error(string.format("path: %s", path))
                local color,text = "red", "hello"
                if query then
                    local q = urllib.parse_query(query) --获取请求的参数

                    for k, v in pairs(q) do

```



页

```
        skynet.error(string.format("query: %s= %s", k,v))
        if(k == "color") then
            color = v
        elseif (k == "text") then
            text = v
        end
    end
end
local reshtml = "<body bgcolor=\"\"..color..\">\"..text..\"</body>\n" --返回一张网

response(id, code, reshtml) --返回状态码200, 并且跟上内容
end
else
    -- 如果抛出的异常是 sockethelper.socket_error 表示是客户端网络断开了。
    if url == sockethelper.socket_error then
        skynet.error("socket closed")
    else
        skynet.error(url)
    end
end
socket.close(id)
end)
end)
```

这个 httpd 模块最初是用于服务器内部管理，以及和其它平台对接。所以只提供了最简单的功能。如果是重度的业务要使用，可以考虑再其上做进一步的开发。

运行httpd服务，然后在浏览器地址栏上输入 `http://127.0.0.1:8001/?color=blue&text=abc`，httpd服务输出如下：

```
$ ./skynet examples/conf
testhttpd
[:01000010] LAUNCH snlua testhttpd
[:01000012] LAUNCH snlua testhttpagent
[:01000019] LAUNCH snlua testhttpagent
[:0100001a] LAUNCH snlua testhttpagent
[:0100001b] LAUNCH snlua testhttpagent
[:0100001c] LAUNCH snlua testhttpagent
[:0100001d] LAUNCH snlua testhttpagent
[:0100001e] LAUNCH snlua testhttpagent
[:0100001f] LAUNCH snlua testhttpagent
[:01000020] LAUNCH snlua testhttpagent
[:01000022] LAUNCH snlua testhttpagent
[:01000029] LAUNCH snlua testhttpagent
[:01000031] LAUNCH snlua testhttpagent
[:01000033] LAUNCH snlua testhttpagent
[:01000034] LAUNCH snlua testhttpagent
```

```
[ :01000035] LAUNCH snlua testhttpagent
[ :01000036] LAUNCH snlua testhttpagent
[ :01000037] LAUNCH snlua testhttpagent
[ :01000038] LAUNCH snlua testhttpagent
[ :01000039] LAUNCH snlua testhttpagent
[ :0100003a] LAUNCH snlua testhttpagent
[ :01000010] 127.0.0.1:50614 connected, pass it to agent :01000012
[ :01000012] header host 127.0.0.1:8001
[ :01000012] path: /
[ :01000012] query: color= blue
[ :01000012] query: text= abc
```

## 18.2 http客户端

skynet 提供了一个非常简单的 http 客户端模块。你可以用:

```
httpc.request(method, host, uri, recvheader, header, content)
```

来提交一个 http 请求, 其中

- method 是 "GET" "POST" 等。
- host 为目标机的地址
- uri 为请求的 URI
- recvheader 可以是 nil 或一张空表, 用于接收回应的 http 协议头。
- header 是自定义的 http 请求头。注: 如果 header 中没有给出 host , 那么将用前面的 host 参数自动补上。
- content 为请求的内容。

它返回状态码和内容。如果网络出错, 则抛出 error 。

```
httpc.dns(server, port)
```

可以用来设置一个异步查询 dns 的服务器地址。如果你不给出地址，那么将从 `/etc/resolv.conf` 查找地址。如果你没有调用它设置异步 dns 查询，那么 skynet 将在网络底层做同步查询。这很有可能阻塞住整个 skynet 的网络消息处理（不仅仅阻塞单个 skynet 服务）。

示例代码: httpclient.lua

```
local skynet = require "skynet"
local httpc = require "http.httpc"
local dns = require "skynet.dns"

local function main()
    httpc.dns() -- set dns server
    httpc.timeout = 100 -- set timeout 1 second
    print("GET baidu.com")
    local respheader = {}
    local status, body = httpc.request("GET", "baidu.com", "/", respheader, { host = "baidu.com" })
    --local status, body = httpc.get("baidu.com", "/", respheader, { host = "baidu.com" })
    print("[header] =====>")
    for k,v in pairs(respheader) do
        print(k,v)
    end
    print("[body] =====>", status)
    print(body)
end

skynet.start(function()
    print(pcall(main))
    skynet.exit()
end)
```

运行结果:

```
$ ./skynet examples/conf
testhttpclient
[:0100010] LAUNCH snlua testhttpclient
GET baidu.com
[header] =====> #百度返回的响应头部分
connection Keep-Alive
date Fri, 09 Feb 2018 16:49:34 GMT
last-modified Tue, 12 Jan 2010 13:48:00 GMT
expires Sat, 10 Feb 2018 16:49:34 GMT
cache-control max-age=86400
accept-ranges bytes
content-type text/html
content-length 81
etag "51-47cf7e6ee8400"
```

```

server Apache
[body] =====> 200 #状态码, 以及响应体
<html>
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
</html>

true #执行成功
[:01000010] KILL self

```

尝试连接一下上一节的http服务, 修改testhttpclient.lua:

```

local skynet = require "skynet"
local httpc = require "http.httpc"
local dns = require "skynet.dns"

local function main()
    httpc.dns() -- set dns server
    httpc.timeout = 100 -- set timeout 1 second
    print("GET 127.0.0.1:8001")
    local respheader = {}
    local status, body = httpc.request("GET", "127.0.0.1:8001", "/?color=blue&text=abc",
respheader)
    print("[header] =====>")
    for k,v in pairs(respheader) do
        print(k,v)
    end
    print("[body] =====>", status)
    print(body)
end

skynet.start(function()
    print(pcall(main))
    skynet.exit()
end)

```

先运行testhttpd再运行testhttpclient:

```

$ ./skynet examples/conf
testhttpd
[:01000010] LAUNCH snlua testhttpd
[:01000012] LAUNCH snlua testhttpagent
[:01000019] LAUNCH snlua testhttpagent
[:0100001a] LAUNCH snlua testhttpagent
[:0100001b] LAUNCH snlua testhttpagent
[:0100001c] LAUNCH snlua testhttpagent
[:0100001d] LAUNCH snlua testhttpagent
[:0100001e] LAUNCH snlua testhttpagent
[:0100001f] LAUNCH snlua testhttpagent
[:01000020] LAUNCH snlua testhttpagent

```

```
[ :01000022] LAUNCH snlua testhttpagent
[ :01000029] LAUNCH snlua testhttpagent
[ :01000031] LAUNCH snlua testhttpagent
[ :01000033] LAUNCH snlua testhttpagent
[ :01000034] LAUNCH snlua testhttpagent
[ :01000035] LAUNCH snlua testhttpagent
[ :01000036] LAUNCH snlua testhttpagent
[ :01000037] LAUNCH snlua testhttpagent
[ :01000038] LAUNCH snlua testhttpagent
[ :01000039] LAUNCH snlua testhttpagent
[ :0100003a] LAUNCH snlua testhttpagent
testhttpclient
[ :0100003b] LAUNCH snlua testhttpclient
GET 127.0.0.1:8001
[ :01000010] 127.0.0.1:50646 connected, pass it to agent :01000012
[ :01000012] header host 127.0.0.1:8001 #testhttpd服务获取GET请求
[ :01000012] path: /
[ :01000012] query: color= blue
[ :01000012] query: text= abc
[header] =====> #testhttpclient得到响应
content-length 32
[body] =====> 200
<body bgcolor="blue">abc</body>

true
[ :0100003b] KILL self
```

## 19 ShareData

当大量的服务可能需要共享一大块并不太需要更新的结构化数据，每个服务却只使用其中一小部分。你可以设想成，这些数据在开发时就放在一个数据仓库中，各个服务按需要检索出需要的部分。

整个工程需要的数据仓库可能规模庞大，每个服务却只需要使用其中一小部分数据，如果每个服务都把所有数据加载进内存，服务数量很多时，就因为重复加载了大量不会触碰的数据而浪费了大量内存。在开发期，却很难把数据切分成更小的粒度，因为很难时刻根据需求的变化重新切分。

sharedata 只支持在同一节点内（同一进程下）共享数据，如果需要跨节点，需要自行同步处理。

## 19.1 shareData API

```
local sharedata = require "skynet.sharedata"
```

```
--[[在当前节点内创建一个共享数据对象,name是字符串
```

```
1、 value 可以是一张 lua table , 但不可以有环,key 必须是字符串和正整数
```

```
2、 value 还可以是一段 lua 文本代码, 而 sharedata 模块将解析这段代码, 把它封装到一个沙盒中运行, 最终取得它返回的 table。如果它不返回 table , 则采用它的沙盒全局环境。
```

```
3、 如果 value 是一个以 @ 开头的字符串, 这个字符串会被解释为一个文件名。sharedata 模块将加载该文件名指定的文件。
```

```
]]--
```

```
sharedata.new(name, value)
```

```
--更新当前节点的共享数据对象。
```

```
sharedata.update(name, value)
```

```
--删除当前节点的共享数据对象。
```

```
sharedata.delete(name)
```

```
--获取当前节点的共享数据对象。
```

```
sharedata.query(name)
```