Welcome to the skynet wiki!

欢迎使用 skynet ，这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](https://github.com/cloudwu/skynet/wiki/Community) 的反馈结果看，它也不仅仅使用在游戏服务器领域。[Uses](https://github.com/cloudwu/skynet/wiki/Uses) 收集了很少一部分使用 skynet 的项目，但实际项目要多得多。欢迎你把你的项目也加到这个列表中。

如果你对 skynet 毫无了解，那么可以先阅读一下 [GettingStarted](https://github.com/cloudwu/skynet/wiki/GettingStarted)。由于 skynet 本身并不复杂，同时建议你阅读一下源代码。参考 [Blogs](https://github.com/cloudwu/skynet/wiki/Blogs) 会对理解设计思路和发展历史有所帮助。

[Build](https://github.com/cloudwu/skynet/wiki/Build) skynet 非常简单，动手编译一个试着玩一下是个很好的开始。examples 和 test 目录下有些例子。如果你想自己动手做二次开发，你可以从理解 [Bootstrap](https://github.com/cloudwu/skynet/wiki/Bootstrap) 开始，一开始不要尝试集群 [Cluster](https://github.com/cloudwu/skynet/wiki/Cluster) 。

虽然 skynet 的核心是由 C 语言编写，但如果只是简单使用 skynet ，并不要求 C 语言基础。但你需要理解 Actor 模式的工作方式，把你的业务拆分成多个服务来协同工作。Lua 是必要的开发语言，你只需要懂得 Lua 就可以使用 [LuaAPI](https://github.com/cloudwu/skynet/wiki/LuaAPI) 来完成服务间的通讯协作。另外，[Snax](https://github.com/cloudwu/skynet/wiki/Snax) 可能会是更简单的方式。关于服务间共享数据，除了用消息传递的方式外，还可以参考 [ShareData](https://github.com/cloudwu/skynet/wiki/ShareData) 。

当然只有这些仅仅可以让 skynet 内部的服务相互协作。要做到给客户端提供服务，还需要使用[Socket](https://github.com/cloudwu/skynet/wiki/Socket) API ，或者使用已经编写好的 [GateServer](https://github.com/cloudwu/skynet/wiki/GateServer) 模板解决大量客户端接入的问题。或许你还需要为 C/S 通讯制订一套通讯协议，skynet 并没有规定这个协议，可以自由选择。当然你也可以看看[Sproto](https://github.com/cloudwu/skynet/wiki/Sproto) 。

通过这套 [Socket](https://github.com/cloudwu/skynet/wiki/Socket) API以及更方便的 [SocketChannel](https://github.com/cloudwu/skynet/wiki/SocketChannel)（更容易实现 socket 池和断开重连） ，可以让 skynet 异步调度外部 socket 事件。对外部独立服务的访问，最好都通过这套 API 的封装。如果外部库直接调用系统的 socket ，很可能阻塞住 skynet 的工作线程，发挥不出性能。目前 redis 和 MongoDB 都有内置的封装好的 driver 可供使用。也有社区的同学封装了 [MySQL](https://github.com/cloudwu/skynet/wiki/MySQL) 库。

如果你找不到你需要的外部组件的 skynet driver ，可以自己来编写，社区欢迎你的贡献。当然，你也可以写一个独立程序和外部组件沟通，再和 skynet 通讯。通讯协议可以自行定义、也可以使用[Http](https://github.com/cloudwu/skynet/wiki/Http) 协议。另有第三方的库可以选择[1](https://github.com/xinmingyao/skynet_web) 支持 websocket。

skynet 由一个或多个进程构成，每个进程被称为一个 skynet 节点。本文描述了 skynet 节点的启动流程。

skynet 节点通过运行 skynet 主程序启动，必须在启动命令行传入一个 [Config](https://github.com/cloudwu/skynet/wiki/Config) 文件名作为启动参数。skynet 会读取这个 config 文件获得启动需要的参数。

第一个启动的服务是 logger ，它负责记录之后的服务中的 log 输出。logger 是一个简单的 C 服务，skynet\_error 这个 C API 会把字符串发送给它。在 config 文件中，logger 配置项可以配置 log 输出的文件名，默认是 nil ，表示输出到标准输出。

bootstrap 这个配置项关系着 skynet 运行的第二个服务。通常通过这个服务把整个系统启动起来。默认的 bootstrap 配置项为 "snlua bootstrap" ，这意味着，skynet 会启动 snlua 这个服务，并将 bootstrap 作为参数传给它。snlua 是 lua 沙盒服务，bootstrap 会根据配置的 luaservice 匹配到最终的 lua 脚本。如果按默认配置，这个脚本应该是 service/bootstrap.lua 。

如无必要，你不需要更改 bootstrap 配置项，让默认的 bootstrap 脚本工作。目前的 bootstrap 脚本如下：

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

skynet.start(function()

local standalone = skynet.getenv "standalone"

local launcher = assert(skynet.launch("snlua","launcher"))

skynet.name(".launcher", launcher)

local harbor\_id = tonumber(skynet.getenv "harbor")

if harbor\_id == 0 then

assert(standalone == nil)

standalone = true

skynet.setenv("standalone", "true")

local ok, slave = pcall(skynet.newservice, "cdummy")

if not ok then

skynet.abort()

end

skynet.name(".slave", slave)

else

if standalone then

if not pcall(skynet.newservice,"cmaster") then

skynet.abort()

end

end

local ok, slave = pcall(skynet.newservice, "cslave")

if not ok then

skynet.abort()

end

skynet.name(".slave", slave)

end

if standalone then

local datacenter = skynet.newservice "datacenterd"

skynet.name("DATACENTER", datacenter)

end

skynet.newservice "service\_mgr"

pcall(skynet.newservice,skynet.getenv "start" or "main")

skynet.exit()end)

这段脚本通常会根据 standalone 配置项判断你启动的是一个 master 节点还是 slave 节点。如果是 master 节点还会进一步的通过 harbor 是否配置为 0 来判断你是否启动的是一个单节点 skynet 网络。

单节点模式下，是不需要通过内置的 harbor 机制做节点间通讯的。但为了兼容（因为你还是有可能注册全局名字），需要启动一个叫做 cdummy 的服务，它负责拦截对外广播的全局名字变更。

如果是多节点模式，对于 master 节点，需要启动 cmaster 服务作节点调度用。此外，每个节点（包括 master 节点自己）都需要启动 cslave 服务，用于节点间的消息转发，以及同步全局名字。

接下来在 master 节点上，还需要启动 [DataCenter](https://github.com/cloudwu/skynet/wiki/DataCenter) 服务。

然后，启动用于 [UniqueService](https://github.com/cloudwu/skynet/wiki/UniqueService) 管理的 service\_mgr 。

最后，它从 config 中读取 start 这个配置项，作为用户定义的服务启动入口脚本运行。成功后，把自己退出。

这个 start 配置项，才是用户定义的启动脚本，默认值为 "main" 。如果你只是试玩一下 skynet ，可能有多份不同的启动脚本，那么建议你多写几份 config 文件，在里面配置不同的 start 项。examples 目录下有很多这样的例子。

skynet 支持两种集群模式。

如果你仅仅是单台物理机的计算能力不足，那么最优的策略是选用更多核心的机器，在同一进程内，skynet 可以保持最高的并行能力，充分利用物理机的多核心，远比增加物理机性价比高得多。

集群并不期望完全隐藏分布式的本质。任何企图抹平服务运行位置差异的设计都需要慎重考虑，很可能存在设计问题。切记，即使在 API 使用层面看起来在不同机器和进程上的服务可以像在同一进程内的服务一样的协作，但差别并不仅仅是消息传递速度不同。搭建集群不是一件简单的事情， skynet 本身要解决的核心问题是提供在同一机器上充分利用多核的处理能力，而并没有提供一套完善的集群方案。它只是提供了一些搭建集群所需要的必要基础设施。

# **master/slave 模式**

当单台机器的处理能力达到极限后，可以考虑通过内置的 master/slave 机制来扩展。具体的配置方法见 [Config](https://github.com/cloudwu/skynet/wiki/Config) 。

每个 skynet 进程都是一个 slave 节点。但其中一个 slave 节点可以通过配置 standalone 来多启动一个 cmaster 服务，用来协调 slave 组网。对于每个 slave 节点，都内置一个 harbor 服务用于和其它 slave 节点通讯。

每个 skynet 服务都有一个全网唯一的地址，这个地址是一个 32bit 数字，其高 8bit 标识着它所属 slave 的号码。即 harbor id 。在 master/slave 网络中，id 为 0 是保留的。所以最多可以有 255 个 slave 节点。

在 master/slave 模式中，节点内的消息通讯和节点间的通讯是透明的。skynet 核心会根据目的地址的 harbor id 来决定是直接投递消息，还是把消息转发给 harbor 服务。但是，两种方式的成本大为不同（可靠性也有所区别），在设计你的系统构架时，应充分考虑两者的性能差异，不应视为相同的行为。

这种模式的缺点也非常明显：它被设计为对单台物理机计算能力不足情况下的补充。所以忽略了系统一部分故障的处理机制，而把整个网络视为一体。即，整个网络中任意一个节点都必须正常工作，节点间的联系也不可断开。这就好比你一台物理机上如果插了多块 CPU ，任意一个损坏都会导致整台机器不能正常工作一样。

所以，不要把这个模式用于跨机房的组网。所有 slave 节点都应该在同一局域网内（最好在同一交换机下）。不应该把系统设计成可以任意上线或下线 slave 的模式。

slave 的组网机制也限制了这一点。如果一个 slave 意外退出网络，这个 harbor id 就被废弃，不可再使用。这样是为了防止网络中其它服务还持有这个断开的 slave 上的服务地址；而一个新的进程以相同的 harbor id 接入时，是无法保证旧地址和新地址不重复的。

如果你非要用 master/slave 模式来实现有一定弹性的集群。skynet 还是提供了非常有限的支持：

local harbor = require "skynet.harbor"

harbor.link(id) 用来监控一个 slave 是否断开。如果 harbor id 对应的 slave 正常，这个 api 将阻塞。当 slave 断开时，会立刻返回。

harbor.linkmaster() 用来在 slave 上监控和 master 的连接是否正常。这个 api 多用于异常时的安全退出（因为当 slave 和 master 断开后，没有手段可以恢复）。

harbor.connect(id) 和 harbor.link 相反。如果 harbor id 对应的 slave 没有连接，这个 api 将阻塞，一直到它连上来才返回。

harbor.queryname(name) 可以用来查询全局名字或本地名字对应的服务地址。它是一个阻塞调用。

harbor.globalname(name, handle) 注册一个全局名字。如果 handle 为空，则注册自己。skynet.name 和 skynet.register 是用其实现的。

你可以利用这组 api 来解决做一次跨节点远程调用，因为节点断开而无法收到回应的问题。注意：link 和 linkmaster 都有一定的开销，所以最好在一个节点中只用少量服务调用它来监控组网状态。由它再来分发到业务层。

对于 harbor id 不可复用的问题。你可以在 [Config](https://github.com/cloudwu/skynet/wiki/Config) 中将 harbor 配置为引用一个系统环境变量。然后给 skynet 编写一个启动脚本，利用一个额外的程序去某个管理器中获得尚未使用过的 harbor id ，设入环境变量，再启动 skynet 进程。这些 skynet 没有给出现成的解决方案，需要你自己来实现。

# **cluster 模式**

skynet 提供了更具弹性的集群方案。它可以和 master/slave 共存。也就是说，你可以部署多组 master/slave 网络，然后再用 cluster 将它们联系起来。当然，比较简单的结构是，每个集群中每个节点都配置为单节点模式（将 harbor id 设置为 0）。cluster 的具体设计可以参考 blog :<http://blog.codingnow.com/2017/03/skynet_cluster.html> 。

要使用它之前，你需要编写一个 cluster 配置文件，配置集群内所有节点的名字和对应的监听端口。并将这个文件事先部署到所有节点，并写在 [Config](https://github.com/cloudwu/skynet/wiki/Config) 中。这个配置文件的范例见 examples/clustername.lua ：

db = "127.0.0.1:2528"

这个配置文件也可以省略，直接通过 cluster.reload 传入一个 table ，参见后面 cluster.reload 的说明 。

这表示，集群中定义有一台叫做 db 的节点，通讯端口为 127.0.0.1:2528 。

接下来，你需要在 db 的启动脚本里写上 cluster.open "db" 。示例见 examples/cluster1.lua 。

local skynet = require "skynet"local cluster = require "cluster"

skynet.start(function()

local sdb = skynet.newservice("simpledb")

skynet.name(".simpledb", sdb)

print(skynet.call(".simpledb", "lua", "SET", "a", "foobar"))

print(skynet.call(".simpledb", "lua", "GET", "a"))

cluster.open "db"end)

它启动了 simpledb 这个服务，并起了一个本地名字 .simpledb ，然后打开了 db 节点的监听端口。

在 examples/cluster2.lua 中示范了如何调用 db 上的 .simpledb 服务。（ .simpledb 原本是一个本地服务，但通过 cluster 接口，其它节点也可以访问到它。）

local skynet = require "skynet"local cluster = require "cluster"

skynet.start(function()

local proxy = cluster.proxy("db", ".simpledb")

print(skynet.call(proxy, "lua", "GET", "a"))

print(cluster.call("db", ".simpledb", "GET", "a"))end)

有两种方式可以访问到 db.simpledb ：

可以通过 cluster.call(nodename, service, ...) 提起请求。这里 nodename 就是在配置表中给出的节点名。service 可以是一个字符串，或者直接是一个数字地址（如果你能从其它渠道获得地址的话）。当 service 是一个字符串时，只需要是那个节点可以见到的服务别名，可以是全局名或本地名。但更推荐是 . 开头的本地名，因为使用 cluster 模式时，似乎没有特别的理由还需要在那个节点上使用 master/slave 的架构（全局名也就没有特别的意义）。cluster.call 有可能因为 cluster 间连接不稳定而抛出 error 。但一旦因为 cluster 间连接断开而抛出 error 后，下一次调用前 cluster 间会尝试重新建立连接。

可以通过 cluster.proxy(nodename, service) 生成一个本地代理。之后，就可以像访问一个本地服务一样，和这个远程服务通讯。但向这个代理服务 send 消息，有可能因为 cluster 间的连接不稳定而丢失。详见 cluster.send 的说明。

如果想单项推送消息，可以调用 cluster.send(nodename, service, ...) 。但注意，跨越节点推送消息有丢失消息的风险。因为 cluster 基于 tcp 连接，当 cluster 间的连接断开，cluster.send 的消息就可能丢失。而这个函数会立刻返回，所以调用者没有机会知道发送出错。

注意：你可以为同一个 skynet 进程（集群中的节点）配置多个通道。这种策略有时会更有效。因为一个通道仅由一条 TCP 连接保持通讯。如果你有高优先级的集群间调用需要处理，那么单独开一个通道可能更好些。

# **Cluster 间的消息次序**

cluster 间的请求大部分会按调用次序排序，即先发出的请求或推送先到。但也有例外的情况。当发送包单个超过 32k 时，包会被切分成多块传输，大的包必须等到所有块传输完毕，在这种情况下，打包的请求逻辑上先发出，可能后收到。回应也有这种可能。

两个 cluster 间，如果有相互请求/推送的情况，会建立两个 tcp 连接。所以、 A 向 B 发起请求和 A 回应 B 的请求，这两类信息是不保证次序的。具体案例可见<https://github.com/cloudwu/skynet/issues/587> 。

# **远端名字服务**

你可以如上面一节所述的方式，给 skynet 的服务命名，然后使用字符串来替代数字地址。同时，cluster 还提供另一套命名方案。

在本地进程内调用 cluster.register(name [,addr]) 可以把 addr 注册为 cluster 可见的一个字符串名字 name 。如果不传 addr 表示把自身注册为 name 。

远端可以通过调用 cluster.query(node, name) 查询到这个名字对应的数字地址。如果名字不存在，则抛出 error 。

由于历史原因，这套命名方案和上一节的方案并存。但这节描述的方案更为推荐。因为这套命名体系仅在 cluster 的模块中存在，并不影响 skynet 底层的命名系统，更容易为日后扩展。而 skynet 底层的命名系统已不再推荐使用。

# **Cluster 配置更新**

Cluster 是去中心化的，所以需要在每台机器上都放置一份配置文件（通常是相同的）。通过调用 cluster.reload 可以让本进程重新加载配置。如果你修改了每个节点名字对应的地址，那么 reload 之后的请求都会发到新的地址。而之前没有收到回应的请求还是会在老地址上等待。如果你老的地址已经无效（通常是主动关闭了进程）那么请求方会收到一个错误。

cluster.reload 也可以接收一个 table 来更新配置，如果你传入了 table，那么 table 内的数据优先级高于配置文件（配置文件被忽略）。如果一开始就没有配置文件，那么必须在使用 cluster 之前用 cluster.reload 传入最初的配置数据。

在线上产品中如何向集群中的每个节点分发新的配置文件，skynet 并没有提供方案。但这个方案一般比较容易实现。例如，你可以自己设计一个中心节点用来管理它。或者让系统管理员编写好同步脚本，并给程序加上控制指令来重载这些配置。或不依赖配置文件，而全部用 cluster.reload 来初始化。

# **Cluster 和 [Snax](https://github.com/cloudwu/skynet/wiki/Snax) 服务**

如果你使用 [Snax](https://github.com/cloudwu/skynet/wiki/Snax) 框架编写服务，且服务需要通过 Cluster 调用。那么需要做一些额外的工作。

首先，在 [Snax](https://github.com/cloudwu/skynet/wiki/Snax) 服务的 init 函数里，请调用一次 snax.enablecluster() ，否则它无法响应 Cluster 调用，而只能接收本地调用。

其次，你需要保证调用方和提供服务的机器上都能打开 snax 脚本。

如果全部条件满足，那么你可以用 cluster.snax(node, name [,address]) 来生成一个远程 snax 服务对象。

当你只给出 node 和 name 时，相当于去目标 node 上使用 snax.queryservice 获取一个服务名对应的地址；如果给出了第三个参数 address ，那么 address 就是 snax 服务的地址，而 name 则是它的服务类型（ 绑定 snax 服务需要这个类型，具体见 snax.bind ）。

kynet 修改了 Lua 的官方实现（可选），加入了一个新特性，可以让多个 Lua VM 共享相同的函数原型[1](http://blog.codingnow.com/2014/03/lua_shared_proto.html)。当在同一个 skynet 进程中开启了大量 lua VM 时，这个特性可以节省不少内存，且提高了 VM 启动速度。

这个特性的使用，对一般用户来说是透明的。它改写了 lua 的辅助 API luaL\_loadfilex ，所有直接或间接调用这个 api 都会受其影响。比如：loadfile 、require 等。它以文件名做 key ，一旦检索到之前有加载过相同文件名的 lua 文件，则从内存中找到之前的函数原型替代。注：Lua 函数是由函数原型以及 0 或多个 upvalue 绑定而成。

loadstring 不受其影响。所以，如果你需要多次加载一份 lua 文件，可以使用 io.open 打开文件，并使用 load 加载。

代码缓存采用只增加不删除的策略，也就是说，一旦你加载过一份脚本，那么到进程结束前，它占据的内存永远不会释放（也不会被加载多次）。在大多数情况下，这不会有问题。

skynet 留出了接口清理缓存，以做一些调试工作。接口模块叫做 skynet.codecache 。

local cache = require "skynet.codecache"

cache.clear()

这样就可以清理掉代码缓存。这个 api 是线程安全的，且老版本的数据依旧在内存中（可能被引用）。但需注意，单纯靠清理缓存的方式做热更新的方案是不完备的。这个完备性和是否引入这个特性无关。因为当你的系统在加载一批 lua 脚本时，单靠源文件的更新，无法保证这批脚本加载的原子性。（有部分是旧版本的，有部分是新版本的）

注意，codecache.clear() 仅仅只是创建一个新的 cache （ api 名字容易引起误会），而不释放内存。所以不要频繁调用。如果你需要你加载文件不受 cache 影响，正确的方式是自己读出代码文本，并用 loadstring 方式加载；而不是在加载前调用 codecache.clear 。

cache.mode(mode)

这个 API 可以修改 codecache 在当前服务中的工作模式。mode 可以是 "ON" "OFF" "EXIST" ，默认的 mode 为 "ON" 。

* 当 mode 为 "ON" 的时候，当前服务 cache 一切加载 lua 代码文件的行为。
* 当 mode 为 "OFF" 的时候，当前服务关闭任何重复利用 lua 代码文件的行为，即使在别的服务中曾经加载过同名文件。
* 当 mode 为 "EXIST" 的时候，当前服务在加载曾经在其它服务或自己的服务加载过同名文件时，复用之前的拷贝。但对新加载的文件则不进行 cache 。注：通常可以让 skynet 本身被 cache 。

当 api 参数为空时，返回当前的 mode 。

注意：由于默认模式是打开状态，所以你第一次调用 cache.mode 的所在文件一定是被 cache 的。

启动 skynet 服务器需要提供一个配置文件，配置文件的编写可以参考 examples/config ，下面是一个简单的配置文件范例：

root = "./"

thread = 8

logger = nil

harbor = 1

address = "127.0.0.1:2526"

master = "127.0.0.1:2013"

start = "main" -- main script

bootstrap = "snlua bootstrap" -- The service for bootstrap

standalone = "0.0.0.0:2013"

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

lualoader = "lualib/loader.lua"

snax = root.."examples/?.lua;"..root.."test/?.lua"

cpath = root.."cservice/?.so"

这个配置文件实际上就是一段 lua 代码，通常，我们以 key = value 的形式对配置项赋值。skynet 在启动时，会读取里面必要的配置项，并将暂时用不到的配置项以字符串形式保存在 skynet 内部的 env 表中。这些配置项可以通过 skynet.getenv 获取。

必要的配置项有：

* ****thread**** 启动多少个工作线程。通常不要将它配置超过你实际拥有的 CPU 核心数。
* ****bootstrap**** skynet 启动的第一个服务以及其启动参数。默认配置为 snlua bootstrap ，即启动一个名为 bootstrap 的 lua 服务。通常指的是 service/bootstrap.lua 这段代码。
* ****cpath**** 用 C 编写的服务模块的位置，通常指 cservice 下那些 .so 文件。如果你的系统的动态库不是以 .so 为后缀，需要做相应的修改。这个路径可以配置多项，以 ; 分割。

在默认的 bootstrap 代码中还会进一步用到一些配置项：

* ****logger**** 它决定了 skynet 内建的 skynet\_error 这个 C API 将信息输出到什么文件中。如果 logger 配置为 nil ，将输出到标准输出。你可以配置一个文件名来将信息记录在特定文件中。
* ****logservice**** 默认为 "logger" ，你可以配置为你定制的 log 服务（比如加上时间戳等更多信息）。可以参考 service\_logger.c 来实现它。注：如果你希望用 lua 来编写这个服务，可以在这里填写 snlua ，然后在 ****logger**** 配置具体的 lua 服务的名字。在 examples 目录下，有 config.userlog 这个范例可供参考。
* ****logpath**** 配置一个路径，当你运行时为一个服务打开 log 时，这个服务所有的输入消息都会被记录在这个目录下，文件名为服务地址。
* ****standalone**** 如果把这个 skynet 进程作为主进程启动（skynet 可以由分布在多台机器上的多个进程构成网络），那么需要配置standalone 这一项，表示这个进程是主节点，它需要开启一个控制中心，监听一个端口，让其它节点接入。
* ****master**** 指定 skynet 控制中心的地址和端口，如果你配置了 standalone 项，那么这一项通常和 standalone 相同。
* ****address**** 当前 skynet 节点的地址和端口，方便其它节点和它组网。注：即使你只使用一个节点，也需要开启控制中心，并额外配置这个节点的地址和端口。
* ****harbor**** 可以是 1-255 间的任意整数。一个 skynet 网络最多支持 255 个节点。每个节点有必须有一个唯一的编号。
* 如果 harbor 为 0 ，skynet 工作在单节点模式下。此时 ****master**** 和 ****address**** 以及 ****standalone****都不必设置。
* ****start**** 这是 bootstrap 最后一个环节将启动的 lua 服务，也就是你定制的 skynet 节点的主程序。默认为 main ，即启动 main.lua 这个脚本。这个 lua 服务的路径由下面的 ****luaservice**** 指定。

集群服务用到的配置项：

* ****cluster**** 它决定了集群配置文件的路径。

lua 服务由 snlua 提供，它会查找一些配置项以加载 lua 代码：

* ****lualoader**** 用哪一段 lua 代码加载 lua 服务。通常配置为 lualib/loader.lua ，再由这段代码解析服务名称，进一步加载 lua 代码。snlua 会将下面几个配置项取出，放在初始化好的 lua 虚拟机的全局变量中。具体可参考实现。
* ****SERVICE\_NAME**** 第一个参数，通常是服务名。
* ****LUA\_PATH**** config 文件中配置的 lua\_path 。
* ****LUA\_CPATH**** config 文件中配置的 lua\_cpath 。
* ****LUA\_PRELOAD**** config 文件中配置的 preload 。
* ****LUA\_SERVICE**** config 文件中配置的 luaservice 。
* ****luaservice**** lua 服务代码所在的位置。可以配置多项，以 ; 分割。 如果在创建 lua 服务时，以一个目录而不是单个文件提供，最终找到的路径还会被添加到 package.path 中。比如，在编写 lua 服务时，有时候会希望把该服务用到的库也放到同一个目录下。
* ****lua\_path**** 将添加到 package.path 中的路径，供 require 调用。
* ****lua\_cpath**** 将添加到 package.cpath 中的路径，供 require 调用。
* ****preload**** 在设置完 package 中的路径后，加载 lua 服务代码前，loader 会尝试先运行一个 preload 制定的脚本，默认为空。
* ****snax**** 用 snax 框架编写的服务的查找路径。
* ****profile**** 默认为 true, 可以用来统计每个服务使用了多少 cpu 时间。在 DebugConsole 中可以查看。会对性能造成微弱的影响，设置为 false 可以关闭这个统计。

另外，你也可以把一些配置选项配置在环境变量中。比如，你可以把 thread 配置在SKYNET\_THREAD 这个环境变量里。你可以在 config 文件中写：

thread=$SKYNET\_THREAD

这样，在 skynet 启动时，就会用 SKYNET\_THREAD 这个环境变量的值替换掉 config 中的$SKYNET\_THREAD 了。

# **后台模式**

****daemon**** 配置 daemon = "./skynet.pid" 可以以后台模式启动 skynet 。注意，同时请配置 logger 项输出 log 。

由于 skynet 框架的消息处理使用了 coroutine ，所以不可以将 lua 原本的 coroutine api 直接和 skynet 服务混用。否则，skynet 的阻塞 API （见 [LuaAPI](https://github.com/cloudwu/skynet/wiki/LuaAPI)）将调用 coroutine.yield 而使得用户写的 coroutine.resume 有不可预期的返回值，并打乱 skynet 框架本身的处理流程。

通常，你可以使用 skynet.fork ，skynet.wait，skynet.wakeup 在 skynet 服务中创建用户级线程。

如果你有其它原因想使用 coroutine ，那么可以使用 skynet.coroutine 模块。该模块的 API 含义和 Lua 原生的 coroutine 基本一致，所以一般可以这样使用：

local coroutine = require "skynet.coroutine"

该模块增加了一个 API ： skynet.coroutine.thread(co) ，它返回两个值，第一个是该 co 是由哪个 skynet thread 间接调用的。如果 co 就是一个 skynet thread ，那么这个值和coroutine.running() 一致，且第二个返回值为 true ，否则第二个返回值为 false 。这第二个返回值可以用于判断一个 co 是否是由 skynet.coroutine.create 或 skynet.coroutine.wrap 创建出来的 coroutine 。

这里的 co 的默认值为 coroutine.running()。

# **限制**

如果你没有调用 skynet.coroutine.resume 启动一个 skynet coroutine 而调用了skynet.coroutine.yield 的话，会返回错误。

你可以在不同的 skynet 线程（由 skynet.fork 创建，或由一条新的外部消息创建出的处理流程）中 resume 同一个 skynet coroutine 。但如果该 coroutine 是由 skynet 框架（通常是调用了 skynet 的阻塞 API）而不是 skynet.coroutine.yield 挂起的话，会被视为 normal 状态，resume 出错。

注：对于挂起在 skynet 框架下的 coroutine ，skynet.coroutine.status 会返回 "blocked" 。

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）。

skynet.queue 模块可以帮助你回避这些伪并发引起的复杂性。

local queue = require "skynet.queue"

这样获得的 queue 是一个函数，每次调用它都可以得到一个新的临界区。临界区可以保护一段代码不被同时运行。

local cs = queue() -- cs 是一个执行队列

local CMD = {}

function CMD.foobar()

cs(func1) -- push func1 into critical sectionend

function CMD.foo()

cs(func2) -- push func2 into critical sectionend

比如你实现了这样一个消息分发器，支持 foobar 和 foo 两类消息。如果你使用 cs 这个 skynet.queue 创建出来的队列。那么在上面的处理流程中， func1 和 func2 这两个函数，都不会在执行过程中相互被打断。

如果你的服务收到多条 foobar 或 foo 消息，一定是处理完一条后，才处理下一条，即使 func1 或 func2 中有 skynet.call 这类的阻塞调用。一旦它们被挂起，新的消息到来后，新的处理流程会被排到 cs 队列尾，等待前面的流程执行完毕才会开始。

注：在 func1 函数内部再调用 cs 是合法的。即：

local function func2()

-- step 3end

local function func1()

-- step 2

cs(func2)

-- step 4end

function CMD.foobar()

-- step 1

cs(func1) -- push func1 into critical section

-- step 5end

如果你这样写，每次收到 foobar 消息后，程序流程会按 step 1, step 2, step 3, step 4, step 5 这样执行，而不会死锁。

在这个过程中，如果 foobar 消息的处理流程被挂起，即使新的 foobar 消息到来，那么，新的消息会立刻执行 step 1 （因为没有被 cs 保护），然后等前一次的 step 4 结束后（step 5 不在 cs 保护中），开始新的 step 2 。

datacenter 可用来在整个 skynet 网络做跨节点的数据共享。

当你需要跨节点通讯时，虽然只要持有其它节点的地址，就可以发送消息。但地址如何获得，却是一个问题。

早期的 skynet 提供了具名服务的特性，可以给一个服务起一个唯一的名字，用名字即可发送消息。但目前更推荐的做法是通过 datacenter 或 [UniqueService](https://github.com/cloudwu/skynet/wiki/UniqueService)。

datacenter 类似一个全网络共享的注册表。它是一个树结构，任何人都可以向其中写入一些合法的 lua 数据，其它服务可以从中取出来。所以你可以把一些需要跨节点访问的服务，自己把其地址记在 datacenter 中，需要的人可以读出。

datacenter 是一个 lua 库，使用：

local datacenter = require "datacenter"

可以进入。

一共有三个方法：

datacenter.set(key1, key2, ... , value) 可以向 key1.key2 设置一个值 value 。这个 api 至少需要两个参数，没有特别限制树结构的层级数。

datacenter.get(key1, key2, ...) 从 key1.key2 读一个值。这个 api 至少需要一个参数，如果传入多个参数，则用来读出树的一个分支。

datacenter.wait(key1, key2, ...) 同 get 方法，但如果读取的分支为 nil 时，这个函数会阻塞，直到有人更新这个分支才返回。当读写次序不确定，但你需要读到其它地方写入的数据后再做后续事情时，用它比循环尝试读取要好的多。wait 必须作用于一个叶节点，不能等待一个分支。

注意：这三个 api 都会阻塞住当前 coroutine ，留心异步重入的问题。

和 [UniqueService](https://github.com/cloudwu/skynet/wiki/UniqueService) 相比较，datacenter 使用起来更加灵活。你还可以通过它来交换 [Multicast](https://github.com/cloudwu/skynet/wiki/Multicast) 的频道号等各种信息。但是，datacenter 其实是通过在 master 节点上部署了一个专门的数据中心服务来共享这些数据的。所有对 datacenter 的查询，都需要和中心节点通讯（如果你是多节点的架构的话），这有时会造成瓶颈。如果你只需要简单的服务地址管理，[UniqueService](https://github.com/cloudwu/skynet/wiki/UniqueService) 做的更好，它会在每个节点都缓存查询结果。

# **在cluster模式下使用**

该功能在cluster模式下不能直接使用，需要自己实现。

skynet 自带了一个调试控制台服务。你需要在你的启动脚本里启动它。

skynet.newservice("debug\_console",8000)

这里的示例是监听 8000 端口，你可以修改成别的端口。

出于安全考虑，调试控制台只能监听本地地址 127.0.0.1 ，所以如果需要远程使用，需要先登录到本机，然后再连接。

可以用 telnet 或 nc 登录调试控制台。启动后会显示

Welcome to skynet console

表示连接成功。

注：由于 skynet 使用自己的 IO 库，所以很难把 libreadline 接入（不能在 readline 的 hook 中 yield）。如果你希望在控制台中使用 readline 的 history 等特性，可以自己使用 rlwrap 。

这时，你可以输入调试指令，输入 help 可以列出目前支持的所有指令。这份文档可能落后于实际版本，所以应以 help 列出的指令为准。

命令的一般格式是 命令 地址 ，有些命令不带地址，会针对所有的服务。当输入地址时，可以使用 :01000001 这样的格式指代一个服务地址：由冒号开头的 8 位 16 进制数字，也可以省略前面两个数字的 harbor id 以及接下来的连续 0 ，比如 :01000001 可以简写为 1 。所有活动的服务可以输入 list 列出。

常用的针对所有 lua 服务的指令有：

* list 列出所有服务，以及启动服务的命令参数。
* gc 强制让所有 lua 服务都执行一次垃圾回收，并报告回收后的内存。
* mem 让所有 lua 服务汇报自己占用的内存。（注：它只能获取 lua 服务的 lua vm 内存占用情况，如果需要 C 模块中内存使用报告，请参考 [MemoryHook](https://github.com/cloudwu/skynet/wiki/MemoryHook) 。
* stat 列出所有 lua 服务的消息队列长度，以及被挂起的请求数量，处理的消息总数。如果在[Config](https://github.com/cloudwu/skynet/wiki/Config) 里设置 profile 为 true ，还会报告服务使用的 cpu 时间。
* service 列出所有的唯一 lua 服务。

注意，由于这些指令是挨个向每个服务发送消息并等待回应，所以当某个 lua 服务过载时，可能需要等待很长时间才有返回。

针对单个 lua 服务的指令有：

* exit address 让一个 lua 服务退出。
* kill address 强制中止一个 lua 服务。
* info address 让一个 lua 服务汇报自己的内部信息，参见 [Profile](https://github.com/cloudwu/skynet/wiki/Profile) 。
* signal address sig 向服务发送一个信号，sig 默认为 0 。当一个服务陷入死循环时，默认信号会打断正在执行的 lua 字节码，并抛出 error 显示调用栈。这是针对 endless loop 的 log 的有效调试方法。注：这里的信号并非系统信号。
* task address 显示一个服务中所有被挂起的请求的调用栈。
* debug address 针对一个 lua 服务启动内置的单步调试器。<http://blog.codingnow.com/2015/02/skynet_debugger.html>
* logon/logoff address 记录一个服务所有的输入消息到文件。需要在 [Config](https://github.com/cloudwu/skynet/wiki/Config) 里配置 logpath 。
* inject address script 将 script 名字对应的脚本插入到指定服务中运行（通常可用于热更新补丁）。

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。

## **用法**

local gateserver = require "snax.gateserver"

local handler = {}

-- register handlers here

gateserver.start(handler)

这样就可以启动一个网关服务。handler 是一组自定义的消息处理函数，分别有：

function handler.connect(fd, ipaddr)

当一个新连接建立后，connect 方法被调用。传入连接的 socket fd 和新连接的 ip 地址（通常用于 log 输出）。

function handler.disconnect(fd)

当一个连接断开，disconnect 被调用，fd 表示是哪个连接。

function handler.error(fd, msg)

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.command(cmd, source, ...)

如果你希望让服务处理一些 skynet 内部消息，可以注册 command 方法。收到 lua 协议的 skynet 消息，会调用这个方法。cmd 是消息的第一个值，通常约定为一个字符串，指明是什么指令。source 是消息的来源地址。这个方法的返回值，会通过 skynet.ret/skynet.pack 返回给来源服务。

open 和 close 这两个指令是保留的。它用于 gate 打开监听端口，和关闭监听端口。

function handler.open(source, conf)

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。

function handler.message(fd, msg, sz)

当一个完整的包被切分好后，message 方法被调用。这里 msg 是一个 C 指针、sz 是一个数字，表示包的长度（C 指针指向的内存块的长度）。注意：这个 C 指针需要在处理完毕后调用 C 方法skynet\_free 释放。（通常建议直接用封装好的库 netpack.tostring 来做这些底层的数据处理）；或是通过 skynet.redirect 转发给别的 skynet 服务处理。

function handler.warning(fd, size)

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。

在这些方法中，还可以调用 gateserver 模块的方法如下：

gateserver.openclient(fd) -- 允许 fd 接收消息

每次收到 handler.connect 后，你都需要调用 openclient 让 fd 上的消息进入。默认状态下， fd 仅仅是连接上你的服务器，但无法发送消息给你。这个步骤需要你显式的调用是因为，或许你需要在新连接建立后，把 fd 的控制权转交给别的服务。那么你可以在一切准备好以后，再放行消息。

gateserver.closeclient(fd) -- 关闭 fd

通常用于主动踢掉一个连接。

## **wire protocol**

这里尽可能的做最简单的约定：

每个包就是 2 个字节 + 数据内容。这两个字节是 Big-Endian 编码的一个数字。数据内容可以是任意字节。

所以，单个数据包最长不能超过 65535 字节。如果业务层需要传输更大的数据块，请在上层业务协议中解决。

### **netpack api**

lualib-src/lua-netpack.c 是处理这类数据包的库。

local netpack = require "netpack"

可以加载这个库。

* netpack.pack(msg, [sz]) 把一个字符串（或一个 C 指针加一个长度）打包成带 2 字节包头的数据块。这个 api 返回一个lightuserdata 和一个 number 。你可以直接送到 socket.write 发送（socket.write 负责最终释放内存）。
* 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。

netpack 还有一些内部 api 用于 gate server 的实现。

## **Gate 服务器**

service/gate.lua 是一个实现完整的网关服务器，同时也可以作为 snax.gateserver 的使用范例。examples/watchdog.lua 是一个可以参考的例子，它启动了一个 service/gate.lua 服务，并将处理外部连接的消息转发处理。

gate 服务启动后，并非立刻开始监听。要让 gate 服务器开启监听端口，可以通过 lua 协议向它发送一个 open 指令，附带一个启动参数表，下面是一个示范：

skynet.call(gate, "lua", "open", {

address = "127.0.0.1", -- 监听地址 127.0.0.1

port = 8888, -- 监听端口 8888

maxclient = 1024, -- 最多允许 1024 个外部连接同时建立

nodelay = true, -- 给外部连接设置 TCP\_NODELAY 属性

})

注: 这个模板不可以和 [Socket](https://github.com/cloudwu/skynet/wiki/Socket) 库一起使用。因为这个模板接管了 socket 类的消息。

## **其它方案**

skynet 并不限制你怎样编写网关，比如你还可以使用这个模块：<http://blog.codingnow.com/2016/03/skynet_tcp_package.html>

# **skynet 入门**

skynet 是一个为网络游戏服务器设计的轻量框架。但它本身并没有任何为网络游戏业务而特别设计的部分，所以尽可以把它用于其它领域。

skynet 并不是一个开箱即用的引擎，使用它需要先对框架本身的结构有所了解，理解框架到底帮助开发者解决怎样的问题。如果你希望使用这个框架来开发网络游戏服务器，你将发现，skynet 并不会引导你把服务器搭建起来。它更像是一套工具，只有你知道你想做什么，它才会帮助你更有效率的完成。

理解 skynet 并不复杂，希望通过读完本篇文章，你就能掌握它。这篇文章没有提及任何 api 的具体使用方法、如何搭建 skynet 开发环境、也没有手把手示范如何写出一个简单的游戏服务器，而仅仅介绍 skynet 的基础概念。所以在真正使用 skynet 做开发时还需要参考 wiki 中的其它文档。

## **框架**

作为服务器，通常需要同时处理多份类似的业务。例如在网络游戏中，你需要同时向数千个用户提供服务；同时运作上百个副本，计算副本中的战斗、让 NPC 通过 AI 工作起来，等等。在单核年代，我们通常在 CPU 上轮流处理这些业务，给用户造成并行的假象。而现代计算机，则可以配置多达数十个核心，如何充分利用它们并行运作数千个相互独立的业务，是设计 skynet 的初衷。

简单的 web 服务倾向于把和用户相关的状态信息（设计好数据结构）储存在数据库中，通过网络收到用户请求后，从数据库中读出关联该用户的状态信息，处理后再写回数据库。而网络游戏服务通常有更强的上下文状态，以及多个用户间更复杂的交互。如果采用相同的模式，数据库和业务处理模块间很容易出现瓶颈，这个瓶颈甚至不能通过增加一个内存 cache 层来完全解决。

在 skynet 中，用服务 (service) 这个概念来表达某项具体业务，它包括了处理业务的逻辑以及关联的数据状态。对，使用 skynet 实现游戏服务器时，不建议把业务状态同步到数据库中，而是存放在服务的内存数据结构里。服务、连同服务处理业务的逻辑代码和业务关联的状态数据，都是常驻内存的。如果数据库是你架构的一部分，那么大多数情况下，它扮演的是一个数据备份的角色。你可以在状态改变时，把数据推到数据库保存，也可以定期写到数据库备份。业务处理时直接使用服务内的内存数据结构。

由于 skynet 服务并非独立进程，所以服务间的通讯也可以被实现的高效的多。另一方面，由于这些服务同时存在于同一个 skynet 进程下，我们可以认为它们同生共死。在编写服务间协作的代码时，不用刻意考虑对方是否还活着、通讯是否可靠的问题。大多数 skynet 服务使用 lua 编写，lua 的虚拟机帮助我们隔离了服务。虽然 skynet 的基础框架设计时并没有限制服务的实现形式，理论上可以用其它语言实现 skynet 服务，但作为刚接触 skynet 的开发者，可以忽略这些细节，仅使用 Lua 做开发。

简单说，可以把 skynet 理解为一个简单的操作系统，它可以用来调度数千个 lua 虚拟机，让它们并行工作。每个 lua 虚拟机都可以接收处理其它虚拟机发送过来的消息，以及对其它虚拟机发送消息。每个 lua 虚拟机，可以看成 skynet 这个操作系统下的独立进程，你可以在 skynet 工作时启动新的进程、销毁不再使用的进程、还可以通过调试控制台监管它们。skynet 同时掌控了外部的网络数据输入，和定时器的管理；它会把这些转换为一致的（类似进程间的消息）消息输入给这些进程。

例如：

在网络游戏中，你可以为每个在线用户创建一个 lua 虚拟机（skynet 称之为 lua 服务），姑且把它称为 agent 。用户在不和其它用户交互而仅仅自娱自乐时，agent 完全可以满足要求。agent 在用户上线时，从数据库加载关联于它的所有数据到 lua vm 中，对用户的网络请求做出反应。当然你也可以让一个 lua 服务管理多个在线用户，每个用户是 lua 虚拟机内的一个对象。

你还可以用独立的服务处理网络游戏中的副本（或是战场），处理玩家和玩家间，玩家协同对战 AI 的战斗。agent 会和副本服务通过消息进行交互，而不必让用户客户端直接与副本通讯。

这些都是具体的游戏服务器架构设计，skynet 并没有要求你应该怎么做，甚至不会建议你该怎么做。一切等你设计时做出决断。

## **网络**

作为网络服务器框架，必然有封装好的网络层，对于 skynet 更是必不可少。由于 skynet 模拟了一个简单的操作系统，它最重要的工作就是调度数千个服务，如何让服务挂起时，尽量减少对系统的影响就是首要解决的问题。我们不建议你在 skynet 的服务中再使用任何直接和系统网络 api 打交道的模块，因为一旦这些模块被网络 IO 阻塞，影响的就不只是该服务本身，而是 skynet 里的工作线程了。skynet 会被配置成固定数量的工作线程，工作线程数通常和系统物理核心数量相关，而 skynet 所管理的服务数量则是动态的、远超过工作线程数量。skynet 内置的网络层可以和它的服务调度器协同工作，使用 skynet 提供的网络 API 就可以在网络 IO 阻塞时，完全释放出 CPU 处理能力。

skynet 有监听 TCP 连接，对外建立 TCP 连接，收发 UDP 数据包的能力。你只需要一行代码就可以监听一个端口，接收外部 TCP 连接。当有新的连接建立时，通过一个回调函数可以获得这个新连接的句柄。之后，和普通的网络应用程序一样，你可以读写这个句柄。与你写过的不同网络应用程序不太一样的是，你还可以把这个句柄转交给 skynet 中的其它服务去处理，以获得并行能力。这有点像传统 posix 系统中，接收一个新连接后，fork 一个子进程继承这个句柄来处理的模式。但不一样的是，skynet 的服务没有父子关系。

我们通常建议使用一个网关服务，专门监听端口，接受新连接。在用户身份确定后，再把真正的业务数据转交给特定的服务来处理。同时，网关还会负责按约定好的协议，把 TCP 连接上的数据流切分成一个个的包，而不需要业务处理服务来分割 TCP 数据流。业务处理的服务不必直接面对 socket 句柄，而由 skynet 正常的内部消息驱动即可。这样的网关服务，skynet 在发布版里就提供了一个，但它只是一个可选模块，你大可以不用它，或自己编写一个类似的服务以更符合你的项目需求。

## **客户端**

skynet 完全不在意如何实现客户端应用，基于 skynet 的服务器，可以用浏览器做客户端（基于 http 或 websocket 协议 通讯），也可以自己用 C / C++ / Flash / Unity3D 等等编写客户端。你可以选用 TCP socket 建立长连接通讯，也可以使用基于 http 协议的短连接，或者基于 UDP 来通讯。这都可以自由选择，skynet 没有提供直接相关的模块，都需要你自己实现。

在 skynet 发布版的示例中，实现了一个用 C + Lua 编写的最简单的客户端 demo ，仅供参考。它基于 TCP 长连接，基础协议是用 2 字节大端字来表示每个数据包的长度，skynet 的网关服务根据这个包长度切割成业务逻辑数据包，分发给对应的内部服务处理。如果你想使用 skynet 内置的网关模块，只需要遵循这个基础的分包约定即可。

对于每个业务包的编码协议约定，在这个 demo 中，使用了一种叫 sproto 自定义协议，它包含在 skynet 的发布版中。demo 演示了 sproto 如何打包数据，解包数据。但是否使用 sproto 协议，skynet 没有任何约束。你也可以使用 json 或是 google protocol buffers 等，只要你知道怎样将对应的协议解析模块自己集成进 Lua 即可。建议在网关，或是使用一个独立服务，将网络消息解包翻译成 skynet 内部消息再转发给对应服务，内部服务不必关心网络层如何传输这些消息的。

## **服务**

skynet 的服务使用 lua 5.3 编写。只需要把符合规范的 .lua 文件放在 skynet 可以找到的路径下就可以由其它服务启动。在 skynet 的配置文件里配置了服务查询路径，以及需要启动的第一个服务，而其它服务都是由该服务直接或间接启动的。每个服务拥有一个唯一的 32bit id ，skynet 把这个 id 称为服务地址，由 skynet 框架分配。即使服务退出，该地址也会尽可能长时间的保留，以避免当消息发向一个正准备退出的服务后，新启动的服务顶替该地址，而导致消息发向了错误的实体。

每个服务分三个运行阶段：

首先是服务加载阶段，当服务的源文件被加载时，就会按 lua 的运行规则被执行到。这个阶段不可以调用任何有可能阻塞住该服务的 skynet api 。因为，在这个阶段中，和服务配套的 skynet 设置并没有初始化完毕。

然后是服务初始化阶段，由 skynet.start 这个 api 注册的初始化函数执行。这个初始化函数理论上可以调用任何 skynet api 了，但启动该服务的 skynet.newservice 这个 api 会一直等待到初始化函数结束才会返回。

最后是服务工作阶段，当你在初始化阶段注册了消息处理函数的话，只要有消息输入，就会触发注册的消息处理函数。这些消息都是 skynet 内部消息，外部的网络数据，定时器也会通过内部消息的形式表达出来。

从 skynet 底层框架来看，每个服务就是一个消息处理器。但在应用层看来并非如此。它是利用 lua 的 coroutine 工作的。当你的服务向另一个服务发送一个请求（即一个带 session 的消息）后，可以认为当前的消息已经处理完毕，服务会被 skynet 挂起。待对应服务收到请求并做出回应（发送一个回应类型的消息）后，服务会找到挂起的 coroutine ，把回应信息传入，延续之前未完的业务流程。从使用者角度看，更像是一个独立线程在处理这个业务流程，每个业务流程有自己独立的上下文，而不像 nodejs 等其它框架中使用的 callback 模式。

和 erlang 不同，一个 skynet 服务在某个业务流程被挂起后，即使回应消息尚未收到，它还是可以处理其他的消息的。所以同一个 skynet 服务可以同时拥有多条业务执行线。所以，你尽可以让同一个 skynet 服务处理很多消息，它们会看起来并行，和真正分拆到不同的服务中处理的区别是，这些处理流程永远不会真正的并行，它们只是在轮流工作。一段业务会一直运行到下一个 IO 阻塞点，然后切换到下一段逻辑。你可以利用这一点，让多条业务线在处理时共享同一组数据，这些数据在同一个 lua 虚拟机下时，读写起来都比通过消息交换要廉价的多。

互有利弊的是，一旦你当前业务处理线挂起，等回应到来继续运行时，内部状态很可能被同期其它业务处理逻辑所改变，请务必小心。在 skynet api 文档中，已经注明了哪些 API 可能导致阻塞。两次阻塞 API 调用之间，运行过程是原子的，利用这个特性，会比传统多线程程序更容易编写。

在同一服务内还可以有多个用户线程，这些线程可以用 skynet.fork 传入一个函数启动，也可以利用 skynet 的定时器的回调函数启动。上面提到的消息处理函数其实也是一条独立的用户线程（可以理解为：响应任何一个请求，都启动了一条新的独立用户线程）。这些并不像真正操作系统的线程那样，可以利用多个核心并行运行。同一服务内的不同用户线程永远是轮流获得执行权的，每个线程都会需要一个阻塞操作而挂起让出控制权，也会在其它线程让出控制权后再延续运行。

如果一条用户线程永远不调用阻塞 API 让出控制权，那么它将永远占据系统工作线程。skynet 并不是一个抢占式调度器，没有时间片的设计，不会因为一个工作线工作时间过长而强制挂起它。所以需要开发者自己小心，不要陷入死循环。不过，skynet 框架也做了一些监控工作，会在某个服务内的某个工作线程占据了太长时间后，以 log 的形式报告。提醒开发者修正导致死循环的 bug 。对于 lua 代码中的死循环 bug （而不是由 lua 调用的 C 模块导致的死循环）还可以由框架强制中断。具体知识可以在开发中遇到后逐步了解。

## **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。

每个 skynet 服务都可以处理多类消息。在 skynet 中，是用 type 这个词来区分消息的。但与其说消息类型不同，不如说更接近网络端口 (port) 这个概念。每个 skynet 服务都支持 255 个不同的 port 。消息分发函数可以根据不同的 port 来为不同的消息定制不同的消息处理流程。

skynet 预定义了一组消息类型，需要开发者关心的有：回应消息、网络消息、调试消息、文本消息、Lua 消息、错误。

回应消息通常不需要特别处理，它由 skynet 基础库管理，用来调度服务内的 coroutine 。当你对外发起一个请求后，对方会回应一个消息，这个消息的类型就是回应消息。发起请求方收到回应消息，会根据消息的 session 来找到之前对应的请求所在的 coroutine ，并延续它。

网络消息也不必直接处理它，skynet 提供了 socket 封装库，封装管理这类消息，改由一组更友好的 socket api 方便使用。

调试消息已经被默认的 skynet 基础库处理了。它使得所有 skynet 服务都提供有一些共同的能力。比如反馈自身虚拟机所占用的内存情况、当前被挂起的任务数量、动态注入一段 lua 代码帮助调试、等等。是的、skynet 并不是通过外框架直接控制每个 lua 虚拟机，调试控制台只是通过向对应的服务发送调试消息，服务自身配合运行才得以反馈自身的状态。

真正的业务逻辑是由文本类消息和 Lua 类消息驱动的。它们的区别仅在于消息的编码形式不同，文本类消息主要方便一些底层的，直接使用 C 编写的服务处理，它就是简单字节串；而 Lua 类消息则可以序列化 Lua 的复杂数据类型。大多数情况下，我们都只使用 lua 类消息。

接管某类消息需要在服务的初始化过程中注册该消息的序列化及反序列化函数，以及消息回调函数。lua 类的序列化函数已经由 skynet 基础库默认注册，它们会把框架传入的消息 C 指针及长度信息转换为一组 Lua 数据。编写业务的开发者只需要注册消息回调函数即可。这个回调函数会接收到别的服务发过来的一系列 Lua 值，以及发送服务的地址和该请求的 session 号（一个 31bit 正整数）。一般我们不必关心地址和 session ，因为 skynet.ret 和 skynet.response 这两个 api 可以帮助你正确的将回应消息发还给请求者。另外，skynet 还约定，如果一个请求不需要回应（单向推送），就置 session 为 0 。

skynet 在应用层还约定了错误类消息，不需要开发者主动处理。这类消息一般没有实际内容，只有发送源地址和 session 号。它专门用来表示某个请求发生了异常，或是某个服务即将或已经退出，无法完成请求。这类错误消息或由 skynet 基础库转换为 lua 层的 error ，抛给调用者。你可以将其理解为 RPC 调用的异常。

## **外部服务**

我们应尽量可能的在同一个 skynet 进程内完成所有的业务逻辑，这样可以最大化利用系统的硬件能力，但有时又必不可少的使用一些外部进程。例如，你可以将 SQLite 封装为一个服务供其它内部服务使用；但你也可能希望使用独立的 MySQL 或是 Redis MongoDB 等独立的外部数据库服务。

skynet 发布版中提供了 mysql redis mongo 的驱动模块，省去了开发者自行封装的烦恼。这些驱动模块都是基于 skynet 的 socket API 实现的，可以很好的协同工作。如果你希望使用别的外部数据库，则需要自行封装。需要注意的是，大多数外部数据库的默认驱动模块都内含了网络部分，它们直接使用了系统的 socket api ，和 skynet 的网络层有一定的性能冲突。一个比较简单的兼容方案是额外再自定义一个中间进程，一边使用外部数据库的默认驱动模块，另一边用 skynet 提供的 socket channel 和 skynet 交互。

你还可能需要让 skynet 提供 http 协议的 web 服务，或是使用 http 协议和外部 web 服务对接。skynet 自带了 http 模块，实现一个简单的 http 服务器不会比用其它框架开发更复杂。即使用 skynet 做一个 web 服务器也可以轻松获得高性能。但是，出于尽量简化框架的理念，skynet 并不提供 https 连接的能力。所以，如果你希望 skynet 对外提供 https 服务时，可以采用 nginx 制作一个 https 反向代理服务器；如果需要对外提起 https 请求，则可以选择任意一款 https 正向代理服务。

不建议把连接管理的网关实现成一个外部服务，因为 skynet 在管理大量 TCP 连接这方面已经做的很好了。放在 skynet 内部做可以减少大量不必要的进程间数据传输。

## **集群**

skynet 在最开始设计的时候，是希望把集群管理做成底层特性的。所以，每个服务的地址预留了 8bit 作为集群节点编号。最多 255 台机器可以组成一个集群，不同节点下的服务可以像同一节点进程内部那样自由的传递消息。

随着 skynet 的演进和实际项目的实践，发现其实把节点间的消息传播透明化，抹平节点间和节点进程内的消息传播的区别并不是一个好主意。在同一进程内，我们可以认为服务以及服务间的通讯都是可靠的，如果自身工作所处的硬件环境正常，那么对方也一定是正常的。而当服务部署在不同进程（不同机器）上时，不可能保证完全可靠。另外一些在同一进程内可以共享访问的内存（skynet 提供的共享数据模块就基于此）也变得不可共享，这些差异无法完全被开发者忽视。

所以，虽然 skynet 可以被配置为多节点模式，但不推荐使用。

目前推荐把不同的 skynet 服务当作外部服务来对待，skynet 发布版中提供了 cluster 模块来简化开发。

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](https://github.com/cloudwu/skynet/wiki/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：

-- examples/simpleweb.lua

local skynet = require "skynet"local socket = require "socket"local httpd = require "http.httpd"local sockethelper = require "http.sockethelper"local urllib = require "http.url"local table = tablelocal string = string

local mode = ...

if mode == "agent" then

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))

endend

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。

local tmp = {}

if header.host then

table.insert(tmp, string.format("host: %s", header.host))

end

local path, query = urllib.parse(url)

table.insert(tmp, string.format("path: %s", path))

if query then

local q = urllib.parse\_query(query)

for k, v in pairs(q) do

table.insert(tmp, string.format("query: %s= %s", k,v))

end

end

response(id, code, table.concat(tmp,"\n"))

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)

else

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(SERVICE\_NAME, "agent")

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)

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。

# **httpc**

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）。

另外，httpc 还提供了简单的 httpc.get 以及 httpc.post 的封装，具体可以参考源代码。

如果有https的需求，可使用 [lua-webclient](http://github.com/dpull/lua-webclient) 它是libcurl multi interface的简单封装，支持单线程，非阻塞的大量http、https请求。

httpc 可以通过设置 httpc.timeout 的值来控制超时时间。时间单位为 1/100 秒。

skynet 提供了一个通用的登陆服务器模版 snax.loginserver 。

## **架构**

先做如下定义：

* 登陆服务器 L 。****这即是本篇介绍的 LoginServer****
* 登陆点若干 G1, G2, G3 ...
* 认证平台 A
* 用户 C

当 C 试图登陆 G1 时，它进行下列流程：

1. C 向 A 发起一次认证请求 (A 通常是第三方认证平台)，获得一个 token 。这个 token 里通常包含有用户名称以及用于校验用户合法性的其它信息。
2. C 将他希望登陆的登陆点 G1 (或其它登陆点，可由系统设计的负载均衡器来选择）以及 step 1 获得的 token 一起发送给 L 。
3. C 和 L 交换后续通讯用的密钥 secret ，并立刻验证。
4. L 校验登陆点是否存在，以及 token 的合法性（此处有可能需要 L 和 A 做一次确认）。
5. （可选步骤）L 检查 C 是否已经登陆，如果已经登陆，向它所在的登陆点（可以是一个，也可以是多个）发送信号，等待登陆点确认。通常这个步骤可以将已登陆的用户登出。
6. L 向 G1 发送用户 C 将登陆的请求，并同时发送 secret 。
7. G1 收到 step 6 的请求后，进行 C 登陆的准备工作（通常是加载数据等），记录 secret ，并由 G1 分配一个 subid 返回给 L。通常 subid 对于一个 userid 是唯一不重复的。
8. L 将子 id 发送给 C 。子 id 多用于多重登陆（允许同一个账号同时登陆多次），一个 userid 和一个 subid 一起才是一次登陆的 username 。而每个 username 都对应有唯一的 secret 。
9. C 得到 L 的确认后，断开和 L 的连接。然后连接 G1 ，并利用 username 和 secret 进行握手。

以上流程，任何一个步骤失败，都会中断登陆流程，用户 C 会收到错误码。

登陆点会按照业务的需要，在确认用户登出后，通知 L 。登出可能发生在连接断开（基于长连接的应用）、用户主动退出、一定时间内没有收到用户消息等。

对于同一个用户登陆时，该用户已经在系统中时，通常有三种应对策略：

1. 允许同时登陆。由于每次登陆的 subid/登陆点 不同，所以可以区分同一个账号下的不同实体。
2. 不允许同时登陆，当新的登陆请求到达并验证后，命令上一次登陆的实体登出。登出完成后，接受新的登陆。
3. 如果一个用户在系统中，禁止该用户再次进入。

LoginServer 并不干涉你用哪种策略，可以自由定制。但对于后两种策略，给于一定的支持，简化业务逻辑实现的复杂性。

## **使用**

lualib/snax/loginserver.lua 是一个辅助库，帮助你实现登陆模块。

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)

取到 snax.loginserver 模块后，构造配置表，然后调用它就可以启动一个登陆服务器。

* 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。
* 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的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方法。

function server.auth\_handler(token)

你需要实现这个方法，对一个客户端发送过来的 token （step 2）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（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）；以及用户名。

在这个方法内做远程调用（skynet.call）是安全的。

function server.login\_handler(server, uid, secret)

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。

如果关闭了 multilogin ，那么对于同一个 uid ，框架不会同时调用多次 login\_handler 。在执行这个函数的过程中，如果用户发起了新的请求，他将直接收到拒绝的返回码。

如果打开 multilogin ，那么 login\_handler 有可能并行执行。由于这个函数在实现时，通常需要调用 skynet.call 让出控制权。所以请小心维护状态。例如，你希望在这个函数中将上一个实例踢下线。那么你需要在踢人操作后再次确认用户是否真的不在线（很有可能另一个登陆的竞争者恰好在此时又登陆成功了）。

一般你还希望这个登陆服务器可以接受一些 skynet 内部控制指令，比如让登陆点可以通知玩家下线了，动态注册新的登陆点等等操作。所以你可以定义这个函数来接收 skynet 内部传递过来的 lua 协议的消息：

function server.command\_handler(command, ...)

command 是第一个参数，通常约定为指令类型。这个函数的返回值会作为回应返回给请求方。

你可以把登陆服务器做为一个单独的 skynet 进程使用，并用 cluster 模块和其它 skynet 进程做集群间通讯；也可以启动在一个 skynet 节点中。在附带的例子 examples/login/logind.lua 中，使用的后一种形式。

你可以参考 examples/login/client.lua 来实现配套的客户端。

## **wire protocol**

登陆服务器和客户端的交互协议基于文本。每个请求和回应包，都以换行符 \n 分割。用户名、服务器名、token 等，为了保证可以正确在文本协议中传输，全部经过了 base64 编码。所以这些业务相关的串可以包含任何字符。

下列通讯流程的协议描述中，S2C 表示这是一个服务器向客户端发送的包；C2S 表示是一个客户端向服务器发送的包。

1. S2C : base64(8bytes random challenge) 这是一个 8 字节长的随机串，用于后序的握手验证。
2. C2S : base64(8bytes handshake client key) 这是一个 8 字节的由客户端发送过来，用于交换 secret 的 key 。
3. Server: Gen a 8bytes handshake server key 生成一个用户交换 secret 的 key 。
4. S2C : base64(DH-Exchange(server key)) 利用 DH 密钥交换算法，发送交换过的 server key 。
5. Server/Client secret := DH-Secret(client key/server key) 服务器和客户端都可以计算出同一个 8 字节的 secret 。
6. C2S : base64(HMAC(challenge, secret)) 回应服务器第一步握手的挑战码，确认握手正常。
7. C2S : DES(secret, base64(token)) 使用 DES 算法，以 secret 做 key 加密传输 token 串。
8. Server : call auth\_handler(token) -> server, uid (A user defined method)
9. Server : call login\_handler(server, uid, secret) -> subid (A user defined method)
10. S2C : 200 base64(subid) 发送确认信息 200 subid ，或发送错误码。

### **错误码**

* 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

一个基于 skynet 框架开发的服务器，是由若干服务构成。你可以将 skynet 看成一个类似操作系统的东西，而服务则可以视为操作系统下的进程。但实际上，单个 skynet 节点仅使用一个操作系统进程，服务间的通讯是在进程内完成的，所以性能比普通的操作系统进程间通讯要高效的多。

skynet 框架是用 C 语言编写，所以它的服务也是用 C 语言开发。但框架已经提供了一个叫做 snlua 的用 C 开发的服务模块，它可以用来解析一段 Lua 脚本来实现业务逻辑。也就是说，你可以在 skynet 启动任意份 snlua 服务，只是它们承载的 Lua 脚本不同。这样，我们只使用 Lua 来进行开发就足够了。

skynet 提供了一个叫做 skynet 的 lua 模块提供给 snlua 服务承载的 Lua 脚本使用。你只需要编写一个后缀为 .lua 的脚本文件，把文件名作为启动参数，启动 snlua 即可。（关于脚本路径的配置，见[Config](https://github.com/cloudwu/skynet/wiki/Config)）

通常，你需要在脚本的第一行写上：

local skynet = require "skynet"

注：skynet 这个模块不能在 skynet 框架之外使用，所以你用标准的 lua 解析器运行包含了 skynet 模块的代码会立即出错。这是因为，每个 skynet 服务都依赖一个 skynet\_context 的 C 对象，它是由 snlua 导入到 lua 虚拟机中的。

每个 skynet 服务，最重要的职责就是处理别的服务发送过来的消息，以及向别的服务发送消息。每条 skynet 消息由五个元素构成。

****session**** ：大部分消息工作在请求回应模式下。即，一个服务向另一个服务发起一个请求，而后收到请求的服务在处理完请求消息后，回复一条消息。session 是由发起请求的服务生成的，对它自己唯一的消息标识。回应方在回应时，将 session 带回。这样发送方才能识别出哪条消息是针对哪条的回应。session 是一个非负整数，当一条消息不需要回应时，按惯例，使用 0 这个特殊的 session 号。session 由 skynet 框架生成管理，通常不需要使用者关心。

****source**** ：消息源。每个服务都由一个 32bit 整数标识。这个整数可以看成是服务在 skynet 系统中的地址。即使在服务退出后，新启动的服务通常也不会使用已用过的地址（除非发生回绕，但一般间隔时间非常长）。每条收到的消息都携带有 source ，方便在回应的时候可以指定地址。但地址的管理通常由框架完成，用户不用关心。

****type**** ：消息类别。每个服务可以接收 256 种不同类别的消息。每种类别可以有不同的消息编码格式。有十几种类别是框架保留的，通常也不建议用户定义新的消息类别。因为用户完全可以利用已有的类别，而用具体的消息内容来区分每条具体的含义。框架把这些 type 映射为字符串便于记忆。最常用的消息类别名为 "lua" 广泛用于用 lua 编写的 skynet 服务间的通讯。

****messsage**** ：消息的 C 指针，在 Lua 层看来是一个 lightuserdata 。框架会隐藏这个细节，最终用户处理的是经过解码过的 lua 对象。只有极少情况，你才需要在 lua 层直接操作这个指针。

****size**** ：消息的长度。通常和 ****message**** 一起结合起来使用。

另外，有部分 API 只在搭建框架时用到，普通服务并不会使用。所以这些 API 被抽离出来放在 skynet.manager 模块中。需要使用时，需要先调用：

require "skynet.manager"

为了兼容老代码，skynet.manager 共享 skynet 名字空间。require 这个模块后，这些额外的 API 依旧放在 skynet 下。而 require "skynet.manager" 的返回值同样是 skynet 名字空间。

这些额外的 API 列在下方，具体的解释放在后续的文字中：

* skynet.launch 启动一个 C 服务。
* skynet.kill 强行杀掉一个服务。
* skynet.abort 退出 skynet 进程。
* skynet.register 给自身注册一个名字。
* skynet.name 为一个服务命名。
* skynet.forward\_type 将本服务实现为消息转发器，对一类消息进行转发。
* skynet.filter 过滤消息再处理。（注：filter 可以将 type, msg, sz, session, source 五个参数先处理过再返回新的 5 个参数。）
* skynet.monitor 给当前 skynet 进程设置一个全局的服务监控。

# **服务地址**

每个服务都有一个 32bit 的数字地址，这个地址的高 8bit 表明了它所属的节点。

skynet.self() 用于获得服务自己的地址。

skynet.harbor() 用于获得服务所属的节点。

skynet.address(address) 用于把一个地址数字转换为一个可用于阅读的字符串。

同时，我们还可以给地址起一个名字方便使用。

skynet.register(name) 可以为自己注册一个别名。（别名必须在 32 个字符以内）

skynet.name(name, address) 为一个地址命名。skynet.name(name, skynet.self()) 和skynet.register(name) 功能等价。

这个名字一旦注册，是在 skynet 系统中通用的，你需要自己约定名字的管理的方法。

以 . 开头的名字是在同一 skynet 节点下有效的，跨节点的 skynet 服务对别的节点下的 . 开头的名字不可见。不同的 skynet 节点可以定义相同的 . 开头的名字。

以字母开头的名字在整个 skynet 网络中都有效，你可以通过这种全局名字把消息发到其它节点的。原则上，不鼓励滥用全局名字，它有一定的管理成本。管用的方法是在业务层交换服务的数字地址，让服务自行记住其它服务的地址来传播消息。

skynet.localname(name) 用来查询一个 . 开头的名字对应的地址。它是一个非阻塞 API ，不可以查询跨节点的全局名字。

下面的 API 说明中，如非特别提及，所有接受服务地址的参数，都可以传入这个地址的字符串别名。

# **消息分发和回应**

skynet.dispatch(type, function(session, source, ...) ... end) 注册特定类消息的处理函数。大多数程序会注册 lua 类消息的处理函数，惯例的写法是：

local CMD = {}

skynet.dispatch("lua", function(session, source, cmd, ...)

local f = assert(CMD[cmd])

f(...)end)

这段代码注册了 lua 类消息的分发函数。通常约定 lua 类消息的第一个元素是一个字符串，表示具体消息对应的操作。我们会在脚本中创建一个 CMD 表，把对应的操作函数定义在表中。每条 lua 消息抵达后，从 CMD 表中查到处理函数，并把余下的参数传入。这个消息的 session 和　source 可以不必传递给处理函数，因为除了主动向 source 发送类别为 "response" 的消息来回应它以外，还有更简单的方法。框架记忆了这两个值。

这仅仅是一个惯用法，你也可以用其它方法来处理消息。skynet 并未规定你必须怎样做。

虽然并不推荐，但你还可以注册新的消息类别，方法是使用 skynet.register\_protocol。例如你可以注册一个以文本方式编码消息的消息类别。通常用 C 编写的服务更容易解析文本消息。skynet 已经定义了这种消息类别为 skynet.PTYPE\_TEXT，但默认并没有注册到 lua 中使用。

skynet.register\_protocol {

name = "text",

id = skynet.PTYPE\_TEXT,

pack = function(m) return tostring(m) end,

unpack = skynet.tostring,

}

新的类别必须提供 pack 和 unpack 函数，用于消息的编码和解码。

pack 函数必须返回一个 string 或是一个 userdata 和 size 。在 Lua 脚本中，推荐你返回 string 类型，而用后一种形式需要对 skynet 底层有足够的了解（采用它多半是因为性能考虑，可以减少一些数据拷贝）。

unpack 函数接收一个 lightuserdata 和一个整数 。即上面提到的 message 和 size 。lua 无法直接处理 C 指针，所以必须使用额外的 C 库导入函数来解码。skynet.tostring 就是这样的一个函数，它将这个 C 指针和长度翻译成 lua 的 string 。

接下来你可以使用 skynet.dispatch 注册 text 类别的处理方法了。当然，直接在skynet.register\_protocol 时传入 dispatch 函数也可以。

dispatch 函数会在收到每条类别对应的消息时被回调。消息先经过 unpack 函数，返回值被传入 dispatch 。每条消息的处理都工作在一个独立的 coroutine 中，看起来以多线程方式工作。但记住，在同一个 lua 虚拟机（同一个 lua 服务）中，永远不可能出现多线程并发的情况。\*\*你的 lua 脚本不需要考虑线程安全的问题，但每次有阻塞 api 调用时，脚本都可能发生重入，这点务必小心。\*\*[CriticalSection](https://github.com/cloudwu/skynet/wiki/CriticalSection) 模块可以帮助你减少并发带来的复杂性。

回应一个消息可以使用 skynet.ret(message, size) 。它会将 message size 对应的消息附上当前消息的 session ，以及 skynet.PTYPE\_RESPONSE 这个类别，发送给当前消息的来源 source 。由于某些历史原因（早期的 skynet 默认消息类别是文本，而没有经过特殊编码），这个 API 被设计成传递一个 C 指针和长度，而不是经过当前消息的 pack 函数打包。或者你也可以省略 size 而传入一个字符串。

由于 skynet 中最常用的消息类别是 lua ，这种消息是经过 skynet.pack 打包的，所以惯用法是skynet.ret(skynet.pack(...)) 。btw，skynet.pack(...) 返回一个 lightuserdata 和一个长度，符合 skynet.ret 的参数需求；与之对应的是 skynet.unpack(message, size) 它可以把一个 C 指针加长度的消息解码成一组 Lua 对象。

skynet.ret 在同一个消息处理的 coroutine 中只可以被调用一次，多次调用会触发异常。有时候，你需要挂起一个请求，等将来时机满足，再回应它。而回应的时候已经在别的 coroutine 中了。针对这种情况，你可以调用 skynet.response(skynet.pack) 获得一个闭包，以后调用这个闭包即可把回应消息发回。这里的参数 skynet.pack 是可选的，你可以传入其它打包函数，默认即是 skynet.pack 。

skynet.response 返回的闭包可用于延迟回应。调用它时，第一个参数通常是 true 表示是一个正常的回应，之后的参数是需要回应的数据。如果是 false ，则给请求者抛出一个异常。它的返回值表示回应的地址是否还有效。如果你仅仅想知道回应地址的有效性，那么可以在第一个参数传入 "TEST" 用于检测。

注：skynet.ret 和 skynet.response 都是非阻塞 API 。

****关于消息数据指针****

skynet 服务间传递的消息在底层是用 C 指针/lightuserdata 加一个数字长度来表示的。当一条消息进入 skynet 服务时，该消息会根据消息类别分发到对应的类别处理流程，（由skynet.register\_protocol ) 。这个消息数据指针是由发送消息方生成的，通常是由skynet\_malloc 分配的内存块。默认情况下，框架会在之后调用 skynet\_free 释放这个指针。

如果你想阻止框架调用 skynet\_free 可以使用 skynet.forward\_type 取代 skynet.start 调用。和 skynet.start 不同，skynet\_forwardtype 需要多传递一张表，表示哪些类的消息不需要框架调用 skynet\_free 。例如：

skynet.forward\_type( { [skynet.PTYPE\_LUA] = skynet.PTYPE\_USER }, start\_func )

表示 PTYPE\_LUA 类的消息处理完毕后，不要调用 skynet\_free 释放消息数据指针。这通常用于做消息转发。

这里由于框架默认定义了 PTYPE\_LUA 的处理流程，而 skynet.register\_protocol 不准重定义这个流程，所以我们可以重定向消息类型为 PTYPE\_USER 。

还有另一种情况也需要用 skynet.forward\_type 阻止释放消息数据指针：如果针对某种特别的消息，传了一个复杂对象（而不是由 skynet\_malloc 分配出来的整块内存；那么就可以让框架忽略数据指针，而自己调用对象的释放函数去释放这个指针。

# **消息的序列化**

在上一节里我们提到，每类消息都应该定义该类型的打包和解包函数。

当我们能确保消息仅在同一进程间流通的时候，便可以直接把 C 对象编码成一个指针。因为进程相同，所以 C 指针可以有效传递。但是，skynet 默认支持有多节点模式，消息有可能被传到另一台机器的另一个进程中。这种情况下，每条消息都必须是一块连续内存，我们就必须对消息进行序列化操作。

skynet 默认提供了一套对 lua 数据结构的序列化方案。即上一节提到的 skynet.pack 以及skynet.unpack 函数。skynet.pack 可以将一组 lua 对象序列化为一个由 malloc 分配出来的 C 指针加一个数字长度。你需要考虑 C 指针引用的数据块何时释放的问题。当然，如果你只是将skynet.pack 填在消息处理框架里时，框架解决了这个管理问题。skynet 将 C 指针发送到其他服务，而接收方会在使用完后释放这个指针。

如果你想把这个序列化模块做它用，建议使用另一个 api skynet.packstring 。和 skynet.pack不同，它返回一个 lua string 。而 skynet.unpack 即可以处理 C 指针，也可以处理 lua string 。

这个序列化库支持 string, boolean, number, lightuserdata, table 这些类型，但对 lua table 的 metatable 支持非常有限，所以尽量不要用其打包带有元方法的 lua 对象。

# **消息推送和远程调用**

有了处理别的服务发送过来的请求的能力，势必就该有向其他服务发送消息或请求的能力。

skynet.send(address, typename, ...) 这条 API 可以把一条类别为 typename 的消息发送给 address 。它会先经过事先注册的 pack 函数打包 ... 的内容。

skynet.send 是一条非阻塞 API ，发送完消息后，coroutine 会继续向下运行，这期间服务不会重入。

skynet.call(address, typename, ...) 这条 API 则不同，它会在内部生成一个唯一 session ，并向 address 提起请求，并阻塞等待对 session 的回应（可以不由 address 回应）。当消息回应后，还会通过之前注册的 unpack 函数解包。表面上看起来，就是发起了一次 RPC ，并阻塞等待回应。call 不支持超时，如果有超时的需求，可以参考[这篇 blog](http://blog.codingnow.com/2015/10/timeout_skynetcall.html) 。

尤其需要留意的是，skynet.call 仅仅阻塞住当前的 coroutine ，而没有阻塞整个服务。在等待回应期间，服务照样可以响应其他请求。所以，尤其要注意，****在** skynet.call **之前获得的服务内的状态，到返回后，很有可能改变****。

还有三个 API 与之相关，但并非常规开发所需要：

skynet.redirect(address, source, typename, session, ...) 它和 skynet.send 功能类似，但更细节一些。它可以指定发送地址（把消息源伪装成另一个服务），指定发送的消息的 session 。注：address 和 source 都必须是数字地址，不可以是别名。skynet.redirect 不会调用 pack ，所以这里的 ... 必须是一个编码过的字符串，或是 userdata 加一个长度。

skynet.genid() 生成一个唯一 session 号。

skynet.rawcall(address, typename, message, size) 它和 skynet.call 功能类似（也是阻塞 API）。但发送时不经过 pack 打包流程，收到回应后，也不走 unpack 流程。

# **服务的启动和退出**

每个 skynet 服务都必须有一个启动函数。这一点和普通 Lua 脚本不同，传统的 Lua 脚本是没有专门的主函数，脚本本身即是主函数。而 skynet 服务，你必须主动调用 skynet.start(function() ... end) 。

skynet.start 注册一个函数为这个服务的启动函数。当然你还是可以在脚本中随意写一段 Lua 代码，它们会先于 start 函数执行。但是，不要在外面调用 skynet 的阻塞 API ，因为框架将无法唤醒它们。

如果你想在 skynet.start 注册的函数之前做点什么，可以调用 skynet.init(function() ... end) 。这通常用于 lua 库的编写。你需要编写的服务引用你的库的时候，事先调用一些 skynet 阻塞 API ，就可以用 skynet.init 把这些工作注册在 start 之前。

skynet.exit() 用于退出当前的服务。skynet.exit 之后的代码都不会被运行。而且，当前服务被阻塞住的 coroutine 也会立刻中断退出。这些通常是一些 RPC 尚未收到回应。所以调用skynet.exit() 请务必小心。

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。

skynet.newservice(name, ...) 用于启动一个新的 Lua 服务。name 是脚本的名字（不用写 .lua 后缀）。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 。如果被启动的脚本在初始化环节抛出异常，或在初始化完成前就调用 skynet.exit 退出，｀skynet.newservice` 都会抛出异常。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。

注意：启动参数其实是以字符串拼接的方式传递过去的。所以不要在参数中传递复杂的 Lua 对象。接收到的参数都是字符串，且字符串中不可以有空格（否则会被分割成多个参数）。这种参数传递方式是历史遗留下来的，有很多潜在的问题。目前推荐的惯例是，让你的服务响应一个启动消息。在 newservice 之后，立刻调用 skynet.call 发送启动请求。

skynet.launch(servicename, ...) 用于启动一个 C 模块的服务。由于 skynet 主要用 lua 编写服务，所以它用的并不多。

注意：同一段 lua 脚本可以作为一个 lua 服务启动多次，同一个 C 模块也可以作为 C 服务启动多次。服务的地址是区分运行时不同服务的唯一标识。如果你想编写一个服务，在系统中只存在一份，可以参考 [UniqueService](https://github.com/cloudwu/skynet/wiki/UniqueService) 。

# **时钟和线程**

skynet 的内部时钟精度为 1/100 秒。

skynet.now() 将返回 skynet 节点进程启动的时间。这个返回值的数值本身意义不大，不同节点在同一时刻取到的值也不相同。只有两次调用的差值才有意义。用来测量经过的时间。每 100 表示真实时间 1 秒。这个函数的开销小于查询系统时钟。在同一个时间片内这个值是不变的。(****注意:这里的时间片表示小于skynet内部时钟周期的时间片,假如执行了比较费时的操作如超长时间的循环,或者调用了外部的阻塞调用,如**os.execute('sleep 1')**, 即使中间没有skynet的阻塞api调用,两次调用的返回值还是会不同的.****)

skynet.starttime() 返回 skynet 节点进程启动的 UTC 时间，以秒为单位。

skynet.time() 返回以秒为单位（精度为小数点后两位）的 UTC 时间。它时间上等价于：skynet.now()/100 + skynet.starttime()

skynet.sleep(ti) 将当前 coroutine 挂起 ti 个单位时间。一个单位是 1/100 秒。它是向框架注册一个定时器实现的。框架会在 ti 时间后，发送一个定时器消息来唤醒这个 coroutine 。这是一个阻塞 API 。它的返回值会告诉你是时间到了，还是被 skynet.wakeup 唤醒 （返回 "BREAK"）。

skynet.yield() 相当于 skynet.sleep(0) 。交出当前服务对 CPU 的控制权。通常在你想做大量的操作，又没有机会调用阻塞 API 时，可以选择调用 yield 让系统跑的更平滑。

skynet.timeout(ti, func) 让框架在 ti 个单位时间后，调用 func 这个函数。这不是一个阻塞 API ，当前 coroutine 会继续向下运行，而 func 将来会在新的 coroutine 中执行。

skynet 的定时器实现的非常高效，所以一般不用太担心性能问题。不过，如果你的服务想大量使用定时器的话，可以考虑一个更好的方法：即在一个service里，尽量只使用一个 skynet.timeout ，用它来触发自己的定时事件模块。这样可以减少大量从框架发送到服务的消息数量。毕竟一个服务在同一个单位时间能处理的外部消息数量是有限的。

timeout 没有取消接口，这是因为你可以简单的封装它获得取消的能力：

function cancelable\_timeout(ti, func)

local function cb()

if func then

func()

end

end

local function cancel()

func = nil

end

skynet.timeout(ti, cb)

return cancelend

local cancel = cancelable\_timeout(ti, dosomething)

cancel() -- canel dosomething

skynet.fork(func, ...) 从功能上，它等价于 skynet.timeout(0, function() func(...) end) 但是比 timeout 高效一点。因为它并不需要向框架注册一个定时器。

skynet.wait() 把当前 coroutine 挂起。通常这个函数需要结合 skynet.wakeup 使用。

skynet.wakeup(co) 唤醒一个被 skynet.sleep 或 skynet.wait 挂起的 coroutine 。在 1.0 版中 wakeup 不保证次序，目前的版本则可以保证。

skynet 默认使用了 jemalloc 作为内存管理模块，但这并非必须。jemalloc 能带来多大好处也和实际应用有关。是否连接 jemalloc 参 [Build](https://github.com/cloudwu/skynet/wiki/Build) 。

skynet 实现了一个和 jemalloc 无关的 memory hook ，用来做服务占用内存的统计，可以用来在 C 层面分析是否有内存泄露。

它的原理是为每个工作线程分配一个 TLS 区，在 worker 处理服务的消息前，先设置当前服务的地址。这样在内存分配发生时就可以知道是由哪个服务分配的内存。这有一点点开销（每块内存多了几个字节，且有一些额外的运行开销），如果你计较它，可以关闭。

通常我们不太需要这个统计，因为大部分服务是用 lua 编写的，可以通过 [DebugConsole](https://github.com/cloudwu/skynet/wiki/DebugConsole) 向 lua 服务索要内存使用情况。它主要用于自己编写的 Lua C 扩展库的统计。统计接口被封装为一个叫作 memory 的库供 lua 调用。或者你可以直接启动 cmemory 这个 lua 服务来输出统计信息。

注: 在 1.0 beta 之后，对于 lua 服务使用的默认分配器不再 hook 统计内存开销。它只统计 C 模块中的内存使用。如果你想修改这个行为，可以阅读<https://github.com/cloudwu/skynet/blob/master/skynet-src/malloc_hook.c>

参考： <https://github.com/cloudwu/skynet/blob/master/service/cmemory.lua>

单个 VM 可以限制内存上限，见：<https://github.com/cloudwu/skynet/blob/master/test/testmemlimit.lua>

skynet集成了mongo的driver，mongo\_collection:findOne、mongo\_collection:findAndModify、mongo\_cursor:next、mongo\_client:runCommand、mongo\_db:runCommand等接口会返回bson库的解析结果，对于返回需要获得某字段的类型时，使用bson.type会更严谨。 bson.type会返回两个值，第一个是数据的类型，第二个是数据本身，特别的，为了区分bson的null与lua层的nil，bson.type在检查到数据类型为TNIL时，数据类型返回为"nil"，数据本身返回nil，当检查到数据类型为bson的null时，数据类型返回为"nil"，数据本身返回一个特殊的字符串。若只是想比较bson的某个字段是否为bson的null，则可以直接与bson.null进行比较(如在检查findAndModify的结果时，检查是否找到并修改了数据，可以检查返回值的value字段与bson.null是否相等)。

## **设计**

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。

和传统的 HTTP 服务不同。在同一个连接上，用户可以发起任意多个请求，并接收多次回应。请求和回应次序不必保证一致，所以内部用 session 来关联请求和回应。回应不保证和请求次序相同，也方便在同一个 TCP 连接上发起一些服务器需要很长时间的请求，而不会阻塞后续的快速响应。

在请求回应模式下，服务器无法单向向客户端推送消息。但可以很容易模拟这种业务。你只需要在登入系统后，立刻发起一个应用层协议要求服务器返回给你最近需要推送给你的消息；如果服务器暂时没有消息推送给客户端，那么只需要简单的挂起这个 session ，待有消息推送时，服务器通过这个 session 返回即可。

和一般的长连接服务也不同。当客户端和服务器失去连接后，只需要简单的重新接入，就可以继续工作。在 snax.msgserver 的实现中，我们要求客户端和服务器同时只保证一条有效的通讯连接，但你可以轻易扩展成多条（但意义不大，因为 session 可以保证慢请求不会阻塞快请求的回应）并发。

## **架构**

消息服务器 snax.msgserver (M) 必须和 [LoginServer](https://github.com/cloudwu/skynet/wiki/LoginServer) (L) 一起使用。用户 C 的业务流程通常是这样的：

1. C 向 L 表明想登陆 M。
2. L 验证 C 的身份，交换 secret ，并转告 M 说 C 想登陆。
3. M 确认 C 的 secret ，做好登陆准备，生成一个 subid (subid 用于多重登陆）。
4. L 构造当次登陆用的用户名（用 uid 和 subid 联合生成）返回给 C 向 C 确认可以登陆。
5. C 以这个用户名加上一个自增序列号（用于断线重连），利用 secret 签名和 M 握手接入。
6. M 检查签名是否正确，以及序列号是否使用过（一次有效）。然后等待 M 的请求，并在当前连接上回应。

如果 C 和 M 断开连接，并不意味着用户从系统登出，这时 C 可以自增序列号，重新和 M 握手。这称为修复连接，C 不需要重复进行和 L 的登陆认证流程。

在前次连接上，C 向 M 发送的请求，M 会缓存一部分。如果连接修复后，C 向 M 发起过去用过的 session 号的请求，M 将之前缓存的回应包直接发回，而不去重新做一次业务处理。但是，在同一连接上发起的相同的 session ，后一次会覆盖前一次，请求都会交给业务层处理。业务层收到相同的 session ，可以认为是前一次的 session 的回应已经被客户端收到，所以复用了这个 session ；因为在同一条 TCP 连接上，不会发生数据丢失的情况；一旦有回应包没有发到，只可能是连接断开，新的连接可以重新发起相同的请求。

注：这里 cache 最近的一部分请求的回应可以满足大部分需求。同时也可以约定 session 的自增性，发现请求过去的 session 而没有缓存时，强制将用户登出。

C 也可以重新去 L 处交换新的用户名和 serect 。这种情况下，L 会通知 M 将登陆状态中的 C 登出，更新新版的 serect ，重置所有之前的请求 session 和回应缓存。最后让 C 重新登陆。

## **使用**

和 [GateServer](https://github.com/cloudwu/skynet/wiki/GateServer) 和 [LoginServer](https://github.com/cloudwu/skynet/wiki/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。

local msgserver = require "snax.msgserver"

你需要注册这样一些 handler 来处理一些事件：

function server.login\_handler(uid, secret)

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 serect ，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。（一般是可以的，如果想阻止用户多重登陆，在登陆服务器里就完成了）

然后为用户生成一个 subid ，如果你的用户同时只允许一个实例存在于系统中，你可以生成固定的 subid （但建议还是生成不同的）。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。

接着你应该做好用户进入的准备。常规做法是启动一个 agent 服务，然后命令它从数据库加载这个用户的数据。如果启动 agent 需要消耗大量的 CPU 时间，你也可以预先启动好多份 agent 放在一个池中，这里只需要简单的取出一个可用的空 agent 即可。

当一切准备好后，把 subid 返回。

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.logout\_handler(uid, subid)

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。这个事件通常是由 agent 的消息触发。

function server.kick\_handler(uid, subid)

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。通常你需要在里面通知 agent 将用户数据写数据库，并且让它在善后工作完成后，发起一个 logout 消息（最终会触发logout\_handler）

function server.disconnect\_handler(username)

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.request\_handler(username, msg, sz)

如果用户提起了一个请求，就会被这个 request\_handler 捕获到。这里隐藏了 session 信息，因为你可以在这个函数中调用 skynet.call 等 RPC 调用，但一般的做法是简单的把 msg,sz 转发给 agent 即可。这里使用的是 client 协议通道，agent 只需要处理这个协议通道，具体的业务层通讯协议并无限制。

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.register\_handler(name)

因为 snax.msgserver 实际上是 snax.gateserver 的一个特殊实现。同样有打开监听端口的指令。在打开端口时，会触发这个 register\_handler name 是在配置信息中配置的当前登陆点的名字，你需要把这个名字注册到登陆服务器。登陆服务器就可以按约定，在用户登陆你的时候把消息转发给你。

### **api**

以下 api 可以在实现上述的 handler 时，在恰当的时机使用：

* msgserver.userid(username) : uid, subid, server 把一个登陆名转换为 uid, subid, servername 三元组
* msgserver.username(uid, subid, server) : username 把 uid, subid, servername 三元组构造成一个登陆名
* msgserver.login(username, secret) 你需要在 login\_handler 中调用它，注册一个登陆名对应的 serect
* msgserver.logout(username) 让一个登陆名失效（登出），通常在 logout\_handler 里调用。
* msgserver.ip(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.start(conf) 启动一个 msgserver 。conf 是配置表。配置表和 [GateServer](https://github.com/cloudwu/skynet/wiki/GateServer) 相同，但增加一项 servername ，你需要配置这个登陆点的名字。

## **范例**

你可以在 examples/login/gated.lua 看到一个实现的范例，它可以和 examples/login/logind.lua 协同工作。

客户端的示范在 examples/login/client.lua 。

你可以先启动

./skynet examples/config.login

然后在同一台机器上启动

lua examples/login/client.lua

## **wire protocol**

基础封包协议遵循的 [GateServer](https://github.com/cloudwu/skynet/wiki/GateServer) ，即一个 2 字节的包头加内容，以下说明的是内容的编码。

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：

base64(uid)@base64(server)#base64(subid):index:base64(hmac)

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。 hmac 是在登陆环节获得的共识 serect ，对前面一串数据做的挑战确认，具体算法可以见范例代码。

服务器会针对这次握手做一个回应，回应信息是一行文本：

* 404 User Not Found
* 403 Index Expired
* 401 Unauthorized
* 400 Bad Request
* 200 OK

如果握手成功，后续的包就是按照请求回应模式进行：

客户端发起请求：

bytes 任意内容 dword session

session 原则上以 Big-Endian 方式编码一个数字，但实际上 session 也可以看成是一个四字节的 token ，并无强制要求客户端如何生成和编码。

服务器回应：

bytes 内容(或异常信息） byte 标记。1 表示正常返回，0 表示异常返回。 dword session

session 用于匹配客户端提起的请求

local mc = require "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，并可以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 是这个频道的 id 。local c2 = 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 ，则新创建出一个频道来。

通常，由一个服务创建出频道，再将 .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。

光绑定到一个频道后，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。

# **组播的原理**

当只考虑一个进程时，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](https://github.com/cloudwu/skynet/wiki/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。

# **在cluster模式下使用**

该功能在cluster模式下不能直接使用，需要自己实现。

虽然 skynet 可以尽可能的利用多核计算，但要特别小心在不能并发的流程上某个环节的处理能力过低。如果在一条处理流水线上，某个服务的处理能力明显低于前一个环节，消息就很有可能堆积在这个服务的消息队列里。

所以应该尽量避免设计出单点服务。不要把太多不相关的处理放在同一个服务内（因为单一服务的消息处理是不能并行的）。对于复杂的系统，靠猜测找到这些瓶颈非常困难，需要利用一些分析工具，然后再从设计上拆分优化。

skynet.stat(what) 可以返回当前服务的性能统计信息，what 可以是以下字符串。

* "mqlen" 消息队列中堆积的消息数量。如果消息是均匀输入的，那么 mqlen 不断增长就说明已经过载。你可以在消息的 dispatch 函数中首先判断 mqlen ，在过载发生时做一些处理（至少 log 记录下来，方便定位问题）。
* "cpu" 占用的 cpu 总时间。需要在 [Config](https://github.com/cloudwu/skynet/wiki/Config) 配置 profile 为 true 。
* "message" 处理的消息条数。

profile 模块可以帮助统计一个消息处理使用的系统时间。

使用 skynet 内置的 profile 记时而不用系统带的 os.time 是因为 profile 可以剔除阻塞调用的时间，准确统计出当前 coroutine 真正的开销。

下面是一个简单的实例：

-- 一个典型的消息分发函数可以是以消息的第一个字符串参数来标识消息类型。local command = {}

function command.foobar(...)end

local function message\_dispatch(cmd, ...)

local f = command[cmd]

f(...)end

加上 profile 就变成了这样：

local profile = require "profile"

local ti = {}

local function message\_dispatch(cmd, ...)

profile.start()

local f = command[cmd]

f(...)

local time = profile.stop()

local p = ti[cmd]

if p == nil then

p = { n = 0, ti = 0 }

ti[cmd] = p

end

p.n = p.n + 1

p.ti = p.ti + timeend

-- 注册 info 函数，便于 debug 指令 INFO 查询。

skynet.info\_func(function()

return tiend)

这段代码中，使用 profile.start() 和 profile.stop() 统计出其间的时间开销（返回单位是秒）。然后按消息类型分别记录在一张表 ti 中。

注：profile.start() 和 profile.stop() 必须在 skynet 线程中调用（记录当前线程），如果在 skynet[Coroutine](https://github.com/cloudwu/skynet/wiki/Coroutine) 中调用的话，请传入指定的 skynet 线程对象，通常可通过skynet.coroutine.thread() 获得。

使用 skynet.info\_func() 可以注册一个函数给 debug 消息处理。向这个服务发送 debug 消息 INFO 就会调用这个函数取得返回值。ps. 使用 debug console 可以主动向服务发送 debug 消息。

当你把业务拆分到多个服务中去后，数据如何共享，可能是最易面临的问题。

最简单粗暴的方法是通过消息传递数据。如果 A 服务需要 B 服务中的数据，可以由 B 服务发送一个消息，将数据打包携带过去。如果是一份数据，很多地方都需要获得它，那么用一个服务装下这组数据，提供一组查询接口即可。[DataCenter](https://github.com/cloudwu/skynet/wiki/DataCenter) 模块对此做了简单的封装。

如果你仅仅需要一组只读的结构信息分享给很多服务（比如一些配置数据），你可以把数据写到一个 lua 文件中，让不同的服务加载它。[Cluster](https://github.com/cloudwu/skynet/wiki/Cluster) 的配置文件就是这样做的。注意：默认 skynet 使用自带的修改版 lua ，会缓存 lua 源文件。当一个 lua 文件通过 loadfile 加载后，磁盘上的修改不会影响下一次加载。所以你需要直接用 io.open 打开文件，再用 load 加载内存中的 string 。

另一个更好的方法是使用 sharedata 模块。

# **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。

如果使用 [DataCenter](https://github.com/cloudwu/skynet/wiki/DataCenter) 这种中心式管理方案，却无法避免每次在检索数据时都要进行一次 RPC 调用，性能或许无法承受。

sharedata 模块正是为了解决这种需求而设计出来的。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。

local sharedata = require "sharedata"

可以引入这个模块。

* ****sharedata.new(name, value)**** 在当前节点内创建一个共享数据对象。
* value 可以是一张 lua table ，但不可以有环。且 key 必须是字符串和正整数。
* 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。
* 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.update(name, value)**** 更新当前节点的共享数据对象。
* ****sharedata.delete(name)**** 删除当前节点的共享数据对象。
* ****sharedata.query(name)**** 获取当前节点的共享数据对象。

一旦 query 到一个共享数据对象，你可以像普通 lua 表那样读取其中的数据，但禁止改写。把其中的分支赋值给 local 变量是安全的，但如果你把最终的叶节点的值取出来后，就不可能被数据源的 update 操作更新了。所以，一般你需要持有至少一级表，每次用它来索引其下的数据。

注意: query 到一个对象后，除非该对象在系统中被 delete ，暂没有任何手段可以清除本地服务中的代理对象。

一旦有人调用 sharedata.update ，所有持有这个对象的服务都会自动去数据源头更新数据。但由于这是一个并行的过程，更新并不保证立刻生效。但使用共享数据的读取方一定能在同一个时间片（单次 skynet 消息处理过程）访问到同一版本的共享数据。

更新过程是惰性的，如果你持有一个代理对象，但在更新数据后没有访问里面的数据，那么该代理对象会一直持有老版本的数据直到第一次访问。这个行为的副作用是：老版本的 C 对象会一直占用内存。如果你需要频繁更新数据，那么，为了加快内存回收，可以通知持有代理对象的服务在更新后，主动调用 sharedata.flush() 。

sharedata 是基于共享内存工作的，且访问共享对象内的数据并不会阻塞当前的服务。所以可以保证不错的性能，并节省大量的内存。

sharedata 的缺点是更新一次的成本非常大，所以不适合做服务间的数据交换。你可以考虑它的替代品：stm 模块。

sharedata 的实现是将原来的数据存储在自定义的table中，并在访问上保持原来的层级结构。自定义的table是一个userdata，其它lua VM通过自定义table的指针访问自定义table中的内容。corelib.lua中定义了userdata的元表。元表中包含\_\_index、\_\_len、\_\_pairs等方法，使得访问userdata像访问原有数据一样。

如果你仅仅想利用 sharedata 模块分发配置表数据，而不关心自动更新数据、共享一部分内存。那么可以使用更高效的接口：

****sharedata.deepcopy(name, keys, ...)**** 这个 api 会获得 name 对应的数据表，并根据你需要的 key 将数据做一次深拷贝，生成一个 lua table 。这样访问的效率更高。例如：sharedata.deepcopy("foobar", "x", "y") 会返回 foobar.x.y 的内容。

# **STM**

STM (Software transactional memory) 模块同样基于共享内存，所以也只能用于同一个 skynet 节点内。它是一个试验性模块，不一定比消息传递的方式更好。只是提供一个新思路来进行同一节点内的服务间数据交换。

因为它不经过 skynet 的消息投递，信息传递的时效性比消息投递要好一些。但由于依旧需要在不同的 lua vm 间交换数据，序列化（使用 skynet.pack 和 skynet.unpack）必不可少。因为绕过了消息投递，它还可以用于广播。多个读取者可以同时去读一个写入者更新的数据。

stm 是以一个 C 编写的 lua 模块形式提供的。

local stm = require "stm"

可以引入这个模块。由于 api 多是操作 C 指针，所以调用其中的 api 上需要小心（否则会有内存泄露）。

* ****stm.new(pointer, size)**** 可以生成一个共享对象，生成者可以改写这个对象。pointer/size 是一个 C 指针以及长度。skynet.pack 可以正确生成它们。它返回一个 stmobj ，是一个 userdata ，lua 的 gc 会正确的回收它引用的内存。
* ****stm.copy(stmobj)**** 可以从一个共享对象中生成一份读拷贝。它返回一个 stmcopy ，是一个 lightuserdata 。通常一定要把这个 lightuserdata 传到需要读取这份数据的服务。随意丢弃这个指针会导致内存泄露。注：一个拷贝只能供一个读取者使用，你不可以把这个指针传递给多个服务。如果你有多个读取者，为每个人调用 copy 生成一份读拷贝。
* ****stm.newcopy(stmcopy)**** 把一个 C 指针转换为一份读拷贝。只有经过转换，stm 才能正确的管理它的生命期。

持有 stmobj ，则是这个共享对象的写入者。你可以用 stmobj(pointer, size) 的方式更新其中携带的信息。（这个 userdata 重载了 call 方法）。

持有 stmcopy ，则是这个共享对象的读取者。stmcpy 是用 stm.copy 生成的那个指针，传递给 stm.newcopy 构造出来的。你可以用 stmcopy(function (pointer, size [,ud]) ... end [,ud]) 的方式读出其中的数据。如果数据没有更新，将返回 false ；否则，将更新过的数据指针 ponter 以及长度，传递给传入的反序列化函数，并返回 true 以及反序列化函数的结果。

test/teststm.lua 是一个简单的使用范例。

# **ShareMap**

sharemap 是对 stm 的简单应用。你可以用 sharemap 创建一个对象负责读写一张预定义的数据结构（使用 sproto 描述结果）。然后构造出读对象传递给其它服务。

当读写方修改了数据内容后，可以通过调用 commit 将修改后的副本同步给所有读取方。而读取方则需要主动调用 update 获得最新副本。

test/testsm.lua 是一个简单的使用范例。

注意：如果需要同步的数据结构比较大，这种方式的成本也会增加。因为每次 commit 都会全量序列化整个结构。

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](https://github.com/cloudwu/skynet/wiki/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。

snax 服务用 lua 编写，但并不是一个独立的 lua 程序。它里面包含了一组 lua 函数，会被 snax 框架分析加载。

test/pingserver.lua 就是一个简单的 snax 服务范例：

local skynet = require "skynet"

local i = 0local hello = "hello"

function response.ping(hello)

skynet.sleep(100)

return helloend

function accept.hello()

i = i + 1

print (i, hello)end

function response.error()

error "throw an error"end

function init( ... )

print ("ping server start:", ...)end

function exit(...)

print ("ping server exit:", ...)end

## **snax 服务的启动和退出**

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）

启动一个 snax 服务有三种方式：

local snax = require "snax"

****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。

****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。

前一种方式可以看成是启动了一个匿名服务，启动后只能用地址（以及对服务地址的对象封装）与之通讯；后两种方式都可以看成是具名服务，之后可以用名字找到它。

后两种方式是对具名服务惰性初始化。如果你在代码中写了多处服务启动，第一次会生效，后续只是对前面启动的服务的查询。往往我们希望明确服务的启动流程（在启动脚本里就把它们启动好）；尤其是全局（整个 skynet 网络可见）的服务，我们还希望明确它启动在哪个结点上（如果是惰性启动，你可能无法预知哪个节点先把这个服务启动起来的）。这时，可以使用下面两个 api ：

****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。

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**** 域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。

这个数字的接收方可以通过 ****snax.bind(handle, typename)**** 把它转换成服务对象。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。

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

****snax.self()**** 用来获取自己这个服务对象，它等价于 snax.bind(skynet.self(), SERVER\_NAME)

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

注：snax 的接口约定是通过分析对应的 lua 源文件完成的。所以无论是消息发送方还是消息接收方都必须可以读到其消息处理的源文件。不仅 snax.newservice 会加载对应的源文件生成新的服务；调用者本身也会在自身的 lua 虚拟机内加载对应的 lua 文件做一次分析；同样， snax.bind 也会按 typename 分析对应的源文件。

## **RPC 调用**

snax 服务中可以定义一组函数用于响应其它服务提出的请求，并给出（或不给出）回应。一个非 snax 服务也可以调用 snax 服务的远程方法。

要定义这类远程方法，可以通过定义 ****function response.foobar(...)**** 来声明一个远程方法。foobar 是方法名，****response**** 前缀表示这个方法一定有一个回应。你可以通过函数返回值来回应远程调用。

调用这个远程方法，可以通过 ****obj.req.foobar(...)**** 来调用它。obj 是服务对象，req 表示这个调用需要接收返回值。foobar 是方法的名字。其实，在创建出 obj 对象时，框架已经了解了这个 foobar 方法有返回值，这里多此一举让使用者明确 req 类型是为了让调用者（以及潜在的代码维护者）清楚这次调用是阻塞的，会导致服务挂起，等待对方的回应。

如果你设计的协议不需要返回值，那么可以通过定义 ****function accept.foobar(...)**** 来声明。这里可以有和 response 组相同的方法名。通过 ****accept**** 前缀来区分 ****response**** 组下同名的方法。这类方法的实现函数不可以有返回值。

调用这类不需要返回值的远程方法，应该使用 ****obj.post.foobar(...)**** 。这个调用不会阻塞。所以你也不用担心这里服务会挂起，因为别的消息进入改变了服务的内部状态。

注: post 不适用于 [Cluster](https://github.com/cloudwu/skynet/wiki/Cluster) 模式, 只能用于本地消息或 master/slave 结构下的消息。

## **热更新**

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 。

举个例子，你可以向上面列出的 pingserver 提交一个 patch ：

snax.hotfix(obj, [[local ilocal hellofunction accept.hello() i = i + 1 print ("fix", i, hello)endfunction hotfix(...) local temp = i i = 100 return tempend]]))

这样便改写了 accept.hello 方法。在 patch 中声明的 local i 和 local hello 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。

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。

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（内部 local 变量的值），或用于修改它们。

注：不可以在 patch 中增加新的远程方法。

test/testping.lua 是对 pingserver 的调用范例，你可以查看它加深理解。

snax 的实现分两部分：让服务工作起来的框架，实现在 service/snaxd.lua ；snax 的 api ，在 lualib/snax.lua 以及 lualib/snax/\* 中。有兴趣的同学可以阅读源码。

skynet 的 C API 采用异步读写，你可以使用 C 调用，监听一个端口，或发起一个 TCP 连接。但具体的操作结果要等待 skynet 的事件回调。skynet 会把结果以 PTYPE\_SOCKET 类型的消息发送给发起请求的服务。（参考skynet\_socket.h）

在处理实际业务中，这样的 API 很难使用，所以又提供了一组阻塞模式的 lua API 用于 TCP socket 的读写。它是对 C API 的封装。

所谓阻塞模式，实际上是利用了 lua 的 coroutine 机制。当你调用 socket api 时，服务有可能被挂起（时间片被让给其他业务处理)，待结果通过 socket 消息返回，coroutine 将延续执行。

local socket = require "socket"

这样就可以在你的服务中引入这组 api 。

* ****socket.open(address, port)**** 建立一个 TCP 连接。返回一个数字 id 。
* ****socket.close(id)**** 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\_fd(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.shutdown(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，shutdown 是一个比 close 更好的选择（因为在 gc 过程中无法切换 coroutine）。
* ****socket.read(id, sz)**** 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.readall(id)**** 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.readline(id, sep)**** 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。
* ****socket.block(id)**** 等待一个 socket 可读。

socket api 中有两个不同的写操作。对应 skynet 为每个 socket 设定的两个写队列。通常我们只需要用：

* ****socket.write(id, str)**** 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。

但同时 skynet 还提供一个低优先级的写操作（如果你不需要这个设计，可以不使用它）：

* ****socket.lwrite(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。

对于服务器，通常我们需要监听一个端口，并转发某个接入连接的处理权。那么可以用如下 API ：

* ****socket.listen(address, port)**** 监听一个端口，返回一个 id ，供 start 使用。
* ****socket.start(id , accept)**** 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，我们有时会希望把这个 socket 的操作权转让给别的服务去处理。

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。任何一个服务只有在调用 ****socket.start(id)**** 之后，才可以收到这个 socket 上的数据。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。

向一个 socket id 写数据也需要先调用 start ，但写数据不限制在调用 start 的同一个服务中。也就是说，你可以在一个服务中调用 start ，然后在另一个服务中向其写入数据。skynet 可以保证一次 write 调用的原子性。即，如果你有多个服务同时向一个 socket id 写数据，每个写操作的串不会被分割开。

****socket.abandon(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.warning(id, callback)**** 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。

### **UDP**

skynet 为 udp 协议做了有限的支持。和 tcp 协议不同，udp 协议不需要阻塞读取。这是因为 udp 是不可靠协议，无法预期下一个读到的数据包是什么（协议允许乱序和丢包）。所以 skynet 的 udp 协议封装采用的是 callback 的方式。

你可以用 socket.udp 这个 API 创建一个 udp handle ，并给它绑定一个 callback 函数。当这个 handle 收到 udp 消息时，callback 函数将被触发。

****socket.udp(function(str, from), address, port) : id****

* 第一个参数是一个 callback 函数，它会收到两个参数。str 是一个字符串即收到的包内容，from 是一个表示消息来源的字符串用于返回这条消息（见 socket.sendto）。
* 第二个参数是一个字符串表示绑定的 ip 地址。如果你不写，默认为 ipv4 的 0.0.0.0 。
* 第三个参数是一个数字， 表示绑定的端口。如果不写或传 0 ，这表示仅创建一个 udp handle （用于发送），但不绑定固定端口。

这个函数会返回一个 handle id 。

****socket.udp\_connect(id, address, port, callback)****

你可以给一个 udp handle 设置一个默认的发送目的地址。当你用 socket.udp 创建出一个非监听状态的 handle 时，设置目的地址非常有用。因为你很难有别的方法获得一个有效的供 socket.sendto 使用的地址串。这里 callback 是可选项，通常你应该在 socket.udp 创建出 handle 时就设置好 callback 函数。但有时，handle 并不是当前 service 创建而是由别处创建出来的。这种情况，你可以用 socket.start 重设 handle 的所有权，并用这个函数设置 callback 函数。

设置完默认的目的地址后，之后你就可以用 socket.write 来发送数据包。

注：handle 只能属于一个 service ，当一个 handle 归属一个 service 时，skynet 框架将对应的网络消息转发给它。向一个 handle 发送网络数据包则不需要当前 service 拥有这个 handle 。

****socket.sendto(id, from, data)****

向一个网络地址发送一个数据包。第二个参数 from 即是一个网络地址，这是一个 string ，通常由 callback 函数生成，你无法自己构建一个地址串，但你可以把 callback 函数中得到的地址串保存起来以后使用。发送的内容是一个字符串 data 。

这个字符串可以用 ****socket.udp\_address(from) : address port**** 转换为可读的 ip 地址和端口，用于记录。

通过阅读实现代码 lualib/socket.lua ，你可以更清楚的了解 socket api 是如何工作的。你也可能会发现一些并未在上面列出的 api 。它们可能在未来的版本中废弃。

如果你需要一个网关帮你接入大量连接并转发它们到不同的地方处理。service/gate.lua 可以直接使用，同时也是用于了解 skynet 的 socket 模块如何工作的不错的参考。它还有一个功能近似的，但是全部用 C 编写的版本 service-src/service\_gate.c 。

如果仅仅是了解 socket api 的基本用法，以及搭建一个简单的服务器。请参考 test/testsocket.lua ；udp 部分见 test/testudp.lua 。

### **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 "dns"

这样可以加载这个模块，它有两个方法。

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 。
* 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 ，以及一张包含有所有 ip 的 table 。
* 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。

你可以阅读 test/testdns.lua 看看范例。同时，这个模块也可以作为在 skynet 中使用 udp 的范例。

请求回应模式是和外部服务交互时所用到的最常用模式之一。通常的协议设计方式有两种。

每个请求包对应一个回应包，由 TCP 协议保证时序。redis 的协议就是一个典型。每个 redis 请求都必须有一个回应，但不必收到回应才可以发送下一个请求。

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。MongoDB 的通讯协议就是这样设计的。

对于第一种模式，用 skynet 的 [Socket](https://github.com/cloudwu/skynet/wiki/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）。

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。

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

local sc = require "socketchannel"

local channel = sc.channel {

host = "127.0.0.1",

port = 3271,

}

这样就可以创建一个 channel 对象出来，其中 host 可以是 ip 地址或者域名，port 是端口号。

接下来，我们就可以工作在模式 1 下了:

local resp = channel:request(req [, response[, padding]])

这里，req 是一个字符串，即请求包。response 是一个 function ，用来收取回应包。返回值是一个字符串，是由 response 函数返回的回应包的内容（可以是任意类型）。response 函数需要定义成这个样子：

function response(sock)

return true, sock:readline()end

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) 。read 可以读指定字节数；readline 可以读以 sep 分割（默认为 \n）的一个字符串（不包含分割符）。

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。

如果协议模式是第 2 种情况，那么你需要在 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 = 3271,

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，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 response 函数返回的类型一致。socket channel 会帮你匹配 session 而让 request 返回正确的值。

****channel:close()**** 可以关闭一个 channel ，通常你可以不必主动关闭它，gc 会回收 channel 占用的资源。

socket channel 在创建时，并不会立即建立连接。如果你什么都不做，那么连接建立会推迟到第一次 request 请求时。这种被动建立连接的过程会不断的尝试，即使第一次没有连接上，也会重试。

你也可以主动调用 channel:connect(true) 尝试连接一次。如果失败，抛出 error 。这里参数 true 表示只尝试一次，如果不填这个参数，则一直重试下去。

由于连接可能发生在任何 request 之前（只要前一次操作检测到连接是断开状态就会重新发起连接），所以 socket channel 支持认证流程，允许在建立连接后，立刻做一些交互。如果开启这个功能，需要在创建 channel 时，填写一个 auth 函数。和 response 函数一样，会给它传入一个 sock 对象。auth 函数不需要返回值，如果认证失败，在 auth 函数中抛出 error 即可。

由于对端有可能在任何时候断开连接, 所以任何一次 request 都有可能抛出 error 而失败，socket channel 将在下一次 request 时重新建立连接。注意：重连并不会重发过去发生 error 的请求，连接断开并不是隐藏在内部的。所以，如果有必要，你应该在请求失败时，业务层重新提出请求。

socket channel 也可用于仅发包而不接收回应。只需要在 request 调用时不填写 response 即可。

channel:response 则可以用来单向接收一个包。

channel:request(req)local resp = channel:response(dispatch)

-- 等价于

local resp = channel:request(req, dispatch)

request 还有第三个参数 padding ，这是用来将体积巨大的消息拆分成多个包发出用的（如果你的协议支持）。padding 需求是一个 table ，里面有若干字符串。如果提供了 padding 参数，socket channel 将连同 req 以及 padding 数组里的字符串，利用 [Socket](https://github.com/cloudwu/skynet/wiki/Socket) 的低优先级通道发出（使用 socket.lwrite）。

这种用法下的 response 函数，应该多返回一个 padding 值。即，对于模式一返回 succ:boolean data:string padding:boolean 三个值；对于模式二，返回 session:number succ:boolean data:string padding:boolean 四个值。

padding 表明了后续是否还有该长消息的后续部分。

如果回应消息是由多个短小的消息合成。channl:request 将返回一个 table ，里面有所有短消息的内容，由调用者来连接这些短消息。

关于 socket channel 的具体用法除了阅读 lualib/socketchannel.lua （同时这也是理解 socket 模块的好材料）的实现外，也可以阅读 lualib/redis.lua 和 lualib/mongo.lua 这两个为 skynet 编写的数据库 driver 。

在和客户端通讯时，需要制订一套通讯协议。 skynet 并没有规定任何通讯协议，所以你可以自由选择。

sproto 是一套由 skynet 自身提供的协议，并没有特别推荐使用，只是一个选项。[sproto 有一个独立项目存在](https://github.com/cloudwu/sproto) 。同时也复制了一份在 skynet 的源码库中。

它类似 google protobuffers ，但设计的更简单，也更利于 lua 使用。同时还提供了一套简单的 rpc 方案。

关于 sproto 的编码协议，在 sproto 的 readme 中已有详述，下面介绍其 RPC 部分。

# **RPC**

首先我们需要定义一个消息包的主体格式。它必须有一个叫 type 的字段，描述 RPC 到底是哪一条消息。还需要有一个 session 字段来表示回应消息的对应关系。通常这两个字段都被定义成 integer 。

.package {

type 0 : integer

session 1 : integer

}

使用 sproto 的 rpc 框架，每条消息都会以这条消息开头，接上真正的消息内容；连接在一起后用 sproto 的 0-pack 方式打包。注意，这个数据包并不包含长度信息，所以真正在网络上传输，还需要添加长度信息，方便分包。当然，如果你使用 skynet 的 gate 模块的话，约定了以两字节大端表示的长度加内容的方式分包。

构造一个 sproto rpc 的消息处理器，应使用：

local host = sproto:host(packagename) -- packagename 默认值为 "package" 即对应前面的 .package 类型。你也可以起别的名字。

这条调用会返回一个 host 对象，用于处理接收的消息。

host:dispatch(msgcontent)

用于处理一条消息。这里的 msgcontent 也是一个字符串，或是一个 userdata（指针）加一个长度。它应符合上述的以 sproto 的 0-pack 方式打包的包格式。

dispatch 调用有两种可能的返回类别，由第一个返回值决定：

REQUEST : 第一个返回值为 "REQUEST" 时，表示这是一个远程请求。如果请求包中没有 session 字段，表示该请求不需要回应。这时，第 2 和第 3 个返回值分别为消息类型名（即在 sproto 定义中提到的某个以 . 开头的类型名），以及消息内容（通常是一个 table ）；如果请求包中有 session 字段，那么还会有第 4 个返回值：一个用于生成回应包的函数。

RESPONSE ：第一个返回值为 "RESPONSE" 时，第 2 和 第 3 个返回值分别为 session 和消息内容。消息内容通常是一个 table ，但也可能不存在内容（仅仅是一个回应确认）。

local sender = host:attach(sp) -- 这里的 sp 指向外发出的消息协议定义。

attach 可以构造一个发送函数，用来将对外请求打包编码成可以被 dispatch 正确解码的数据包。

这个 sender 函数接受三个参数（name, args, session）。name 是消息的字符串名、args 是一张保存用消息内容的 table ，而 session 是你提供的唯一识别号，用于让对方正确的回应。 当你的协议不规定需要回应时，session 可以不给出。同样，args 也可以为空。

# **Sproto Loader**

由于 skynet 采用的是多 lua 虚拟机。如果在每个 VM 里都加载相同的 sproto 协议定义就略显浪费。所以 skynet 还提供了一个叫 sprotoloader 的模块来共享它们。

其实现原理是在 C 模块中提供了 16 个全局的 slot ，可以通过 sprotoloader.register 或 sprotoloader.save 在初始化时，加载需要的协议，并保存在这些 slot 里。通常我们只需要两个 slot ，一个用于保存客户端到服务器的协议组，另一个用于保存服务器到客户端的协议组。分别位于 slot 1 和 2 。

这样，在每个 vm 内，都可以通过 sprotoloader.load 把协议加载到 vm 中。

注意：这套 api 并非线程安全。所以必须自行保证在初始化完毕后再做 load 操作。（load 本身是线程安全的）。

具体的使用范例，可以参考 examples 下的 agent.lua 以及 examples 下的 client.lua 。理解 RPC 是如何工作的。

我们可以通过 skynet.newservice 启动一个 lua 编写的服务。同一段脚本可以启动多份，每个有不同的地址。地址是区分不同服务的唯一标识。

但有时，整个系统中解决一类事务只需要一个服务，在系统启动时，它便启动好，而其它服务需要知道它的地址以便于使用它。这个时候，使用 skynet.uniqueservice 是更好的选择。

skynet.uniqueservice 和 skynet.newservice 的输入参数相同，都可以以一个脚本名称找到一段 lua 脚本并启动它，返回这个服务的地址。但和 newservice 不同，每个名字的脚本在同一个 skynet 节点只会启动一次。如果已有同名服务启动或启动中，后调用的人获得的是前一次启动的服务的地址。

它很大程度上取代了具名服务（不再推荐使用的早期特性）的功能。很多 skynet 库都附带有一个独立服务，你可以在库的初始化时，写上类似的语句：

local SERVICE

skynet.init(function()

SERVICE = skynet.uniqueservice "foobar"end)

这个范例会注册一个初始化函数去初始化 SERVICE 变量。而你的库函数就可以使用 SERVICE 这个地址来访问对应的唯一的 foobar 服务了。

uniqueservice 采用的是惰性初始化的策略。整个系统中第一次调用时，服务才会被启动起来。有时，你并不希望做惰性初始化，而在 skynet 启动脚本里明确把必须的服务初始化好（这个初始化过程可能比较漫长，惰性初始化会导致不必要的运行时延迟）。那么，如果你明确知道服务已经启动好，可以使用 skynet.queryservice 来查询已有服务。如果这个服务不存在，这个 api 会一直阻塞到它启动好为止。

默认情况下，uniqueservice 是不跨节点的。也就是说，不同节点上调用 uniqueservice 即使服务脚本名相同，服务也会独立启动起来。如果你需要整个网络有唯一的服务，那么可以在调用 uniqueservice 的参数前加一个 true ，表示这是一个全局服务。

对应的，查询服务 queryservice 也支持第一个参数为 true 的情况。这种全局服务，queryservice 更加有用。往往你需要明确知道一个全局服务部署在哪个节点上，以便于合理的架构。你可以在你设计的节点上的启动脚本中调用 skynet.uniqueservice(true, "foobar") 将服务启动后，然后再在其它使用它的地方调用 skynet.queryservice(true, "foobar") 。

和 [DataCenter](https://github.com/cloudwu/skynet/wiki/DataCenter) 不同，uniqueservice 是一个专用于服务管理的模块。它在服务地址管理上做了特别的优化。因为对于同一个名字，只允许启动一次，且不准更换。所以，在实现上，我们可以在每个节点缓存查询过的结果，而不必每次都去中心节点查询。