# service设计综述

## 简介

在skynet中，一个service是基于一个独立lua沙盒的可运行单位，可以理解为一个进程，多个service之间相互独立，变量不通用，可以通过skynet.send 或者skynet.call实现service间的异步或者同步调用，skynet本身也说明了自己不是开箱即用的框架，所以我们对基础的service做了额外的包装，本文主要介绍这些包装的原理和使用方法

## 初始化

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。

以上是摘抄自skynet wiki的服务启动介绍

一个标准的service 最简单的service服务如下

主服务，一般是脚本配置的入口脚本，调用 skynet.newservice 或者skynet.uniqueservice来启动一个指定服务

服务本身的源码文件中一般如下

local skynet = require “skynet”

skynet.start(function()

print(“hello world”)

end),

一般我们会加入一个lua消息的dispatch方法来让服务有处理消息的能力

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

print("return", skynet.ret command)

end)

比如以上服务就会将发送给服务的内容打印出来

这是一个最简单的服务模型，启动服务，注册服务监听的消息即可

但是对于实际开发来说，这种还是显得有些简陋，所以我们对基础的skynet在不破坏原有功能的前提下进行了扩展，得到了smbb\_skynet

首先我们为skynet增加了一个退出回调的set和get方法，因为skynet目前没有标准的方法来退出一个服务，我们将所有可能会扩充skynet基础库的内容全部放到smbb\_skynet中集中管理，防止随意添加造成的后续无法维护

同时我们给skynet加入一个固定框架的handle，来实现类似erlang gen\_server的固定基础框架，handle内容如下

skynet.handle = {

info = nil,

on\_exit = nil,

init = nil,

handle\_lua\_msg = nil,

register = nil,

}

info 为debug时调用的info函数，一般是打印当前服务中的内部变量

on\_exit 为服务退出时的回调处理函数

init 函数指定的函数会作为skynet.init的参数被调用

handle\_lua\_msg 指定服务的lua处理逻辑

register 如果不为nil，则会为该服务调用 skynet. register注册这个别名

基于当前这个包装过的smbb\_skynet，我们上层再包装了一个base\_service用来实现handle的绑定，handle各个函数的绑定这里不再赘述，逻辑很简单，看一次源码就可以清楚，这里只解释 sys.REG(skynet)

之前我们在skynet的基础上扩展了消亡回调函数的set和get方法，最终就是sys模块实现的绑定，sys模块会为当前服务注册”sys”类型的消息，在收到stop命令后，就会调用之前set过的消亡回调函数。

同时我们扩展了这个逻辑，让他可以管理子服务，如果服务消亡被调用的时候有子服务在，则会调用子服务的消亡逻辑，成功后才会自我消亡，子服务在启动的时候需要调用sys.reg\_uniqueservice或sys.reg\_service来将将要启动的服务注册到调用者本身。如果在拉起子服务的时候做了注册，就不需要考虑消亡时的处理，调用顺序目前保持的是，先拉起的服务最后才会被消亡，最后拉起的服务最先调用消亡回调

之后基于base\_service，稍作包装就可以获得一个基础的项目通用的服务，以当前项目为例，我们require了基础的定义lua，同时改写了requre，实现了require配置文件的时候调用sharedata逻辑，最终我们就可以获得一个项目通用的smbb\_service

以下就是一个标准的smbb\_service的服务代码

*local* skynet = require "smbb\_skynet"  
*local* service = require "lualib.smbb\_sevice"  
*local* logger = require "lualib.logger"  
  
*local* server\_func = {}  
  
*local* skynet\_handle = skynet.handle  
  
-- 调用skynet.init  
skynet\_handle.init = *function*()  
*end*-- 服务关闭时的回调函数  
skynet\_handle.on\_exit = *function*()  
 ***logger***.*info*(SERVICE\_NAME .. "stop")  
*end*--handle\_lua\_msg和command参数互相冲突，如果有handle\_lua\_msg则command不会生效  
--skynet\_handle.handle\_lua\_msg = function(\_, source, command, ...)  
-- server\_func[command](...)  
--end  
  
skynet\_handle.command = server\_func  
  
skynet\_handle.register = SERVICE\_NAME

skynet\_handle.info = *function*()  
 *return  
end*service.*init*(skynet\_handle)

填充对应的函数，就可以获得一个具有基础功能的基本服务

## 注意点

skynet有个特性，支持一个服务“同时”执行多个任务，简单的来说就是如果有个消息引起的动作会造成阻塞，这个阻塞不会影响接下来的其他消息的处理，skynet也是通过这个特性完成的高并发

但是对于一个游戏逻辑来说这并不一定是一个好的特性。举个例子，A和B两人都要删除和C的好友关系，他们都会给friend\_server发送一个从C的好友信息里删除A和B的消息，假设A的删除消息先到，然后friend\_server发现C的信息内存中没有，然后需要从数据库中get，这时候就会阻塞，然后B的消息就会开始处理，这时候B对应的操作也会阻塞，然后两条操作返回的时候，他们都会认为从数据库里获得的就是正确的数据，然后基础这份C玩家的好友信息，分别删除自己，最后的结果就有可能是只删除了A或者只删除的了B，因为A和B是“并发”的。

所以绝大多数情况下我们推荐关闭掉这个特性，通过skynet.queue来维护一个任务队列，所有的任务都是基于任务队列运行，也就保持了和erlang相同的特性，单进程内永远是串行调度，这样可能会有性能的损失，但是避免了出现很多无法预估的问题

如果因为关闭这个特性引起了性能问题，需要解决的也应该是如何将阻塞点进行拆分而不是强行开启这套逻辑

我们并不拒绝开启这套逻辑，但是开启时请务必考虑清楚该服务当前的功能是否支持重入，他们是否会有阻塞操作而造成变量不一致，比如一个简单的datacenter这种没有io阻塞的数据中心，就完全可以启动这套逻辑，虽然启用不启用在这种情况下都一样。