

硕 士 研 究 生 读 书 报 告



题目 skynet框架研究

作者姓名 杨修一

作者学号 21651064

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 二○一七年一月

The Research Of Skynet Framework

A Dissertation Submitted to

Zhejiang University

in partial fulfillment of the requirements for

the degree of

Master of Engineering

Major Subject: Software Engineering

Advisor: Qilei Li

By

Xiuyi Yang

Zhejiang University, P.R. China

2017

摘要

本文重点探讨了skynet游戏服务器框架的设计及其组成，如何利用如今CPU的多核能力将不同的业务放在独立的执行环境中处理，协同工作，实现多线程高并发的服务器开发。使用嵌入式语言Lua，Lua State 已经提供了良好的沙盒，隔离不同执行环境。而多线程模式，可以使得状态共享、数据交换更加高效。而多线程模型的诸多弊端，比如复杂的线程锁、线程调度问题等，都可以通过减小底层的规模，精简设计，最终把危害限制在很小的范围内。

**关键词**：服务器， 多线程，Lua

Abstract

This paper focuses on the skynet game server framework design and its composition, how to use the multi-core capability of CPU will be different business in a separate execution environment to deal with and work together to achieve multi-threaded high concurrent server development. Using the embedded language Lua, Lua State has provided a good sandbox to isolate different execution environments. The multi-threaded mode, you can make the state of sharing, data exchange more efficient. The multi-threaded model of the many drawbacks, such as complex thread locks, thread scheduling problems, can reduce the size of the underlying, streamlining the design, and ultimately limit the harm to a very small range.

**Key words**: server, multithreading, Lua

**1. 简介**

skynet是一个为网络游戏服务器设计的轻量框架。基于Actor模式，但它的设计有很强的通用性，所以可以把它开发用于其它领域。

skynet并不是一个开箱即用的引擎，是服务端的最底层框架,和游戏有关的各种服务都是基于架构之上开发的。skynet的功能只有管理好服务(加载与调度)和服务之间的调用(请求与响应)。skynet大部分都要在Lua脚本下开发,只有需要考虑性能的模块才用C语言开发成Lua库,提供给Lua调用。

**2. 基本原理**

2.1启动流程

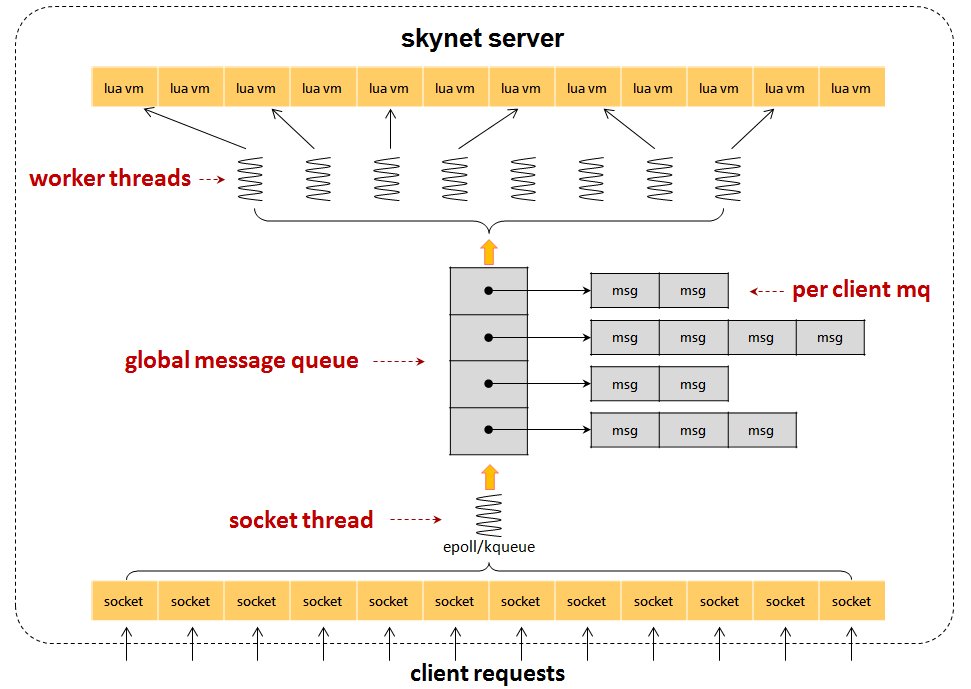
首先设置一下Lua的环境、默认的配置、打开config配置文件,并修改默认配置。最后调用位于skynet\_start.c文件中skynet\_start()函数。

然后调用skynet\_start()函数初始化skynet的各个模块,包括harbor节点、handle服务ID、mq消息队列、module加载动态链接库、timer时钟、socket套接字以及加载一些服务，logger日志服务、master主服务、harbor节点服务、snLua加载Lua模块的服务;以及最后启动几种线程包括\_moitor、\_timer、\_socket和根据线程数启动n个工作线程。skynet将通过snlua服务加载第一个用户的服务:

即:ctx=skynet\_context\_new("snlua",config->start);这行,意思就是使用snlua加载config->start这个服务。而config->start指向配置文件config的start="main"这行。

2.2 总体架构

skynet服务处理的简化框架如下图所示：



 每个在线客户的客户端，在skynet server上都对应有一个socket与其连接。一个socket在skynet内部对应一个Lua虚拟机和一个”客户特定消息队列“(per client mq)。当客户特定消息队列中有消息时，该队列就会挂载到全局队列（global message queue）上，供工作线程（worker threads）进行调度处理。

   skynet的服务处理主流程如下：一个socket线程轮询所有的socket，收到客户端请求后将请求打包成一个消息，发送到该socket对应的客户特定消息队列中，然后将该消息队列挂到全局队列队尾；N个工作线程从全局队列头部获取client特定的消息队列，从客户特定消息队列中取出一个消息进行处理，处理完后将该消息队列重新挂到全局队列队尾。

  实际代码要更复杂一些：定时器线程会周期性检查一下设置的定时器，将到期的定时器消息发送到client消息队列中；每个Lua vm在运行过程中也会向其他Lua vm（或自己）的客户特定消息队列发送消息；monitor线程监控各个客户端状态，看是否有死循环的消息等等。

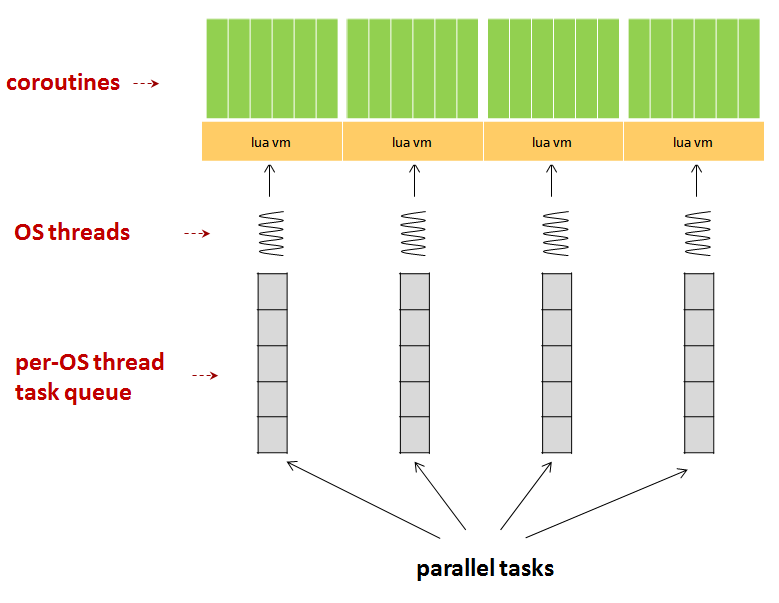
  每个客户处理消息时，都是按照消息到达的顺序进行处理。同一时刻，一个客户的消息只会被一个工作线程调度，因此客户处理逻辑无需考虑多线程并发，基本不需要加锁。

**3. 运行机理**

3.1并发任务调度方式

  Lua支持non-preemptive的coroutine，一个Lua虚拟机中可以支持海量并发的协作任务，coroutine主要的问题是不支持多核，无法充分利用当前服务器普遍提供的多核能力。所以目前有很多项目为Lua添加OS thread支持，比如[LuaLanes](https://github.com/LuaLanes/lanes" \t "/home/aleafall/文档\\x/_blank)，[LuaProc](https://github.com/askyrme/luaproc" \t "/home/aleafall/文档\\x/_blank)等，这些项目都要解决的一个问题就是并发任务的组成以及调度问题。

  并发任务可以使用coroutine表示：每个OS线程上创建一个Lua虚拟机(Lua\_State)，虚拟机上可以创建海量的coroutine，这种调度如下图所示：



这种OS线程与Luavm1:1的调度方式有很多优点：

每个OS线程都有私有的消息队列，该队列有多个写入者，但只有一个读取者，可以实现读端免锁设计。

OS线程可以与Lua vm绑定，也可以不绑定。由于现代OS都会尽量将CPU core和OS线程绑定，所以如果OS线程与Lua vm绑定的话，可以大大减少cpu cache的刷新，提高cache命中率

Lua vm与OS线程个数相当，与任务数无关。大量任务可以共用同一个Lua vm，共享其Lua byte code，字符串常量等信息，极大减少每个任务的内存占用。

但该方式也有比较严重的缺陷：

不支持任务跨Lua vm迁移。每个任务是一个coroutine，而coroutine是Lua vm内部的数据结构，执行中其stack引用了Lua vm内部的大量共享数据，无法迁移到另一个Lua vm上执行。当一个Lua vm上的多个任务都比较繁忙时，只能由一个OS线程串行执行，无法通过work stealing等方式交给其他OS线程并行处理。

在同一个Lua vm内的多个任务，共享Lua vm的内存空间。一个任务出现问题时，很容易影响到其他任务。简单说就是任务间的隔离性不好。

  另一种处理方式是每个Lua vm表示一个任务，系统中的海量并发任务由海量Lua vm处理。skynet就是采用的这种方式。这种方式有效解决了上一种处理方式的两个缺陷，每个任务完全独立，可以交给任意一个OS线程处理；同时任务不会共享Lua vm内存空间，隔离性非常好，一个任务的问题不会影响其他任务的执行。这种方式的主要问题就是存在大量内存浪费。每个Lua vm都要加载大量相同的Lua字节码和常量，对内存需求量非常高。这也造成每个任务执行时无法重用cpu cache，导致cache命中率降低很多。云风对这一问题的解决方案是修改Lua vm的代码加载机制，同一进程内部的多个Lua vm共享字节码。具体实现上，有一个独立的Lua vm专门负责加载字节码，并负责字节码的垃圾回收。同一进程中的其他Lua vm共享该独立Lua vm加载的字节码。这种方式无法解决字符串常量共享的问题，仅仅解决了字节码共享的问题。不过即使这样，每个在线用户也节省了[1M](http://blog.codingnow.com/2014/03/lua_shared_proto.html" \t "/home/aleafall/文档\\x/_blank)的内存。

3.2 全局消息队列

  skynet中有一个”全局消息队列“和在线用户特定的”任务特定消息队列“。与go语言1.1版本之前runtime对goroutine的调度类似，所有工作线程使用了一个公共的全局队列。skynet全局队列是一个循环队列，使用数组实现。使用全局队列不是一种很高效的消息调度方式，每个消息的出队入队一般都需要加全局锁。go1.1中，优化了调度器算法，[每个线程使用局部队列](http://morsmachine.dk/go-scheduler" \t "/home/aleafall/文档\\x/_blank)，性能提高很多（据称有些应用性能提升近40%）。

3.3服务间通讯设计

skynet的服务间通讯性能强于多进程的方案。

数据包通常是在一个服务内打包生成的，skynet 并不关心数据包是怎样被打包的，它甚至不要求这个数据包内的数据是连续的（虽然这样很危险，在后面会谈及的跨机通讯中会出错，除非你保证你的数据包绝对不被传递出当前所在的进程）。它仅仅是把数据包的指针，以及声称的数据包长度（并不一定是真实长度）传递出去。由于服务都是在同一个进程内，接收方取得这个指针后，就可以直接处理其引用的数据了。

这个机制可以在必要时，保证绝对的零拷贝，几乎等价于在同一线程内做一次函数调用的开销。

但，这只是 skynet 提供的性能上的可能性。它推荐的是一种更可靠，性能略低的方案：它约定，每个服务发送出去的包都是复制到用 malloc 分配出来的连续内存。接收方在处理完这个数据块（在处理的 callback 函数调用完毕）后，会默认调用 free 函数释放掉所占的内存。即，发送方申请内存，接收方释放。

skynet\_send 和 callback 函数的定义：

int skynet\_send(

struct skynet\_context \* context,

uint32\_t source,

uint32\_t destination,

int type,

int session,

void \* msg,

size\_t sz

);

typedef int (\*skynet\_cb)(

struct skynet\_context \* context,

void \*ud,

int type,

int session,

uint32\_t source ,

const void \* msg,

size\_t sz

);

这里，source 和 destination 都是 32 位整数，表示地址。原则上不需要填写 source 地址，因为默认就是它自己。0 是系统保留的 handle ，可以指代自己。这里允许填写 source 值，是因为在某些特殊场合，需要伪造一个由别人发出的包。姑且可以理解 source 为 reply address 。

发送一个数据包，就是发送 msg/sz 对。可以在 type 里打上 dontcopy 的 tag (PTYPE\_TAG\_DONTCOPY) ，让框架不要复制 msg/sz 指代的数据包。否则 skynet 会用 malloc 分配一块内存，把数据复制进去。callback 函数在处理完这块数据后，会调用 free 释放内存。你可以通过让 callback 返回 1 ，阻止框架释放内存。这通常和在 send 时标记 dontcopy 标记配对使用。

在早期的设计中，是没有 session 和 type 的。初看它们也的确很多余。因为，完全可以把所有信息都编码进数据包内，这些都并非底层通讯框架所关心的事情。只需要上层设施约定一致的通讯协议就够了。的确，在过去的版本中，正是这样做的，这个一致的通讯协议编码方式就是 google protocol buffer 。后来发现这个方式增加了无谓的性能开销。而无论你怎样定义服务间的通讯方式，session 都是必不可少的，很大程度上，type 也是。

把这两个量提取出来，可以方便不同的协议间协作，而不必强行统一一致的编码方式，那才是对开发人员的约束。

另外，需要明确的一点是，skynet 核心并不解决进程间通讯的问题。数据交换方式并非类似 TCP 的数据流，所以，没有必要把服务间的通讯形式强行统一为单个数据块。最合适做进程内通讯的方式就是 C 结构。消息发送方和接收方都处于同一个进程内时，它们一定可以识别同一个 C 结构映射的内存块，不必考虑内存布局，字节序等问题。在这个层面上使用这种更高效的数据交换方式，可以极大的提升性能。

虽然 skynet 核心只解决单向的消息包发送问题，但是，应用很大部分都需要使用请求回应的模式。即，一个服务向另一个服务提出一个请求包，对方处理完这个请求后，把结果返回。由于每个服务仅有一个 callback 函数，好比在 ip 协议中去掉了端口的设定，所有发送到一个 ip 地址上的 ip 包就无法被分发到不同的进程了。这时，就需要有另一个东西来区分这个包。这就是 session 的作用。

使用 skynet\_send 发送一个包的时候，你可以在 type 里设上 alloc session 的 tag (PTYPE\_TAG\_ALLOCSESSION)。send api 就会忽略掉传入的 session 参数，而会分配出一个当前服务从来没有使用过的 session 号，发送出去。同时约定，接收方在处理完这个消息后，把这个 session 原样发送回来。这样，编写服务的人只需要在 callback 函数里记录下所有待返回的 session 表，就可以在收到每个消息后，正确的调用对应的处理函数。

服务间的交互，只有很少的服务只需要处理别人发送过来的请求，而不需要向外提出请求。所以至少需要区分请求包和回应包。这两种包显然是有不同的处理方式，但它们需要从同一个 callback 函数入口进入。这就需要用一个额外的参数type区分。

type 表示的是当前消息包的协议组别，而不是传统意义上的消息类别编号。协议组别类型并不会很多，所以，skynet限制了 type 的范围是 0 到 255 ，由一个字节标识。在实现时，skynet把 type 编码到了 size 参数的高 8 位。因为单个消息包限制长度在 16 M （24 bit)内，是个合理的限制。这样，为每个消息增加了 type 字段，并没有额外增加内存上的开销。

查看 skynet.h 可以看到已经定义出来的消息类型：

#define PTYPE\_TEXT 0

#define PTYPE\_RESPONSE 1

#define PTYPE\_MULTICAST 2

#define PTYPE\_CLIENT 3

#define PTYPE\_SYSTEM 4

#define PTYPE\_HARBOR 5

#define PTYPE\_TAG\_DONTCOPY 0x10000

#define PTYPE\_TAG\_ALLOCSESSION 0x20000

0 是内部服务最为常用的文本消息类型。 1 表示这是一个回应包，应该依据对方的规范来解码。后面定义出来的几种类型暂时不解释，也可以自定义更多的类型，比如在目前的版本中，lua 层面还定义出了一些专用于 lua state 间通讯的消息编码类型。

3.4 集群间通讯

虽然设计上围绕单进程多线程模块进行的，但 skynet 其实并不仅限于单进程。它实际是可以部署到不同机器上联合工作的。这虽然不是核心特性，但核心层为此做了许多配合。

单个 skynet 进程内的服务数量被 handle 数量限制。handle 也就是每个服务的地址，在接口上看用的是一个 32 位整数。但实际上单个服务中 handle 的最终限制在 24bit 内，也就是 16M 个。高 8 位是保留给集群间通讯用的。

最终允许 255 个 skynet 节点部署在不同的机器上协作。每个 skynet 节点有不同的 id 。这里被称为 harbor id 。这个是独立指定，人为管理分配的（也可以写一个中央服务协调分配）。每个消息包产生的时候，skynet 框架会把自己的 harbor id 编码到源地址的高 8 位。这样，系统内所有的服务模块，都有不同的地址了。从数字地址，可以轻易识别出，这个消息是远程消息，还是本地消息。

这也是 skynet 核心层做的事情，核心并不解决远程数据交互的工作。

集群间的通讯，是由一个独立的 harbor 服务来完成的。所有的消息包在发送时，skynet 识别出这是一个远程消息包时，都会把它转发到 harbor 服务内。harbor 服务会建立 tcp 连接到所有它认识的其它 skynet 节点内的 harbor 服务上。

Harbor 间通过单向的 tcp 连接管道传输数据，完成不同的 skynet 节点间的数据交换。

skynet 目前支持一个全局名字服务，可以把一个消息包发送到特定名字的服务上。这个服务不必存在于当前 skynet 节点中。这样，就需要一个机构能够同步这些全局名字。

为此，skynet实现了一个叫做 master 的服务。它的作用就是广播同步所有的全局名字，以及加入进来的 skynet 节点的地址。本质上，这些地址也是一种名字。同样可以用 key-value 的形式储存。即，每个 skynet 节点号对应一个字符串的地址。

3.5 组播

对于游戏服务，组播服务是一个重要的优化。如果没有组播服务，可以自己用一个循环，把一个包发送给不同的地址。但游戏中大量需要把一条消息发送给不同的实体。对于传统的做法，需要组播的场合，把所有的实体实现在同一个线程内。但是，在使用 skynet 时，往往一个服务只实现少部分功能，大量依赖服务间的通讯。这样，优化这些组播包，就显得有必要了。

组播必须让 skynet 在底层做一些支持，而很全部放在核心层之外的实现。

这是因为，组播包的分配和释放策略和其它包不同。它需要有引用计数。和别的消息包，发送方分配，接收方释放是不同的。固然，可以把消息包统一成全部带有引用计数，只是单播包记数为 1 。但这样，就牺牲了单播包的性能。skynet希望的效果是 ，如果你不用这个机制（比如组播）就不必为之付出成本。所以， skynet在底层做了有限的支持。

skynet 会识别消息的 type 是否为 PTYPE\_MULTICAST ，然后有不同的生命期管理策略，并把组播包交给组播服务处理。这一点，和集群间通讯的做法非常类似。

组播服务并不解决分熟在不同集群节点上的服务归组的问题。即，每个分组内的成员都必须在同一系统进程内。这可以极大的简化设计。用户可以让不同的服务 handle 归属一个组号。向 skynet 索取这个组号对应的 handle 。向这个组的 handle 发送消息，就等同于向组内所有 handle 发送消息。

而跨集群分组又如何做到呢？这里是在上层用 Lua 来做了进一步的封装。

首先，提供了一个简单的，用 C 编写的服务，叫做 tunnel 。它可以把发送给它的消息，无条件的转发到另一个 handle 上。这个转发 handle 可以是在不同 skynet 节点上的。

skynet用 Lua 实现了一个全局的分组管理器，协调在不同节点上，创建出相同组名的分组来。然后用 tunnel 服务连接不同节点上的同一分组就够了。具体细节就不在此赘述。

skynet 的核心功能就是发送消息和处理消息。它体现在 skynet\_send 和 skynet\_callback 两个 API 上。历史原因，增加了 skynet\_sendname 这个 API 。数字地址也有一个字符串名字，以 : 开头，跟上 8 字节的 16 进制的数字串标识。同一节点内的服务地址用 . 开头，而全局名字则是其它的字符串。为了代码和协议简洁，做了一些小限制：全局名字不可以超过 16 个字符。

但是，skynet 本身还需要另一些 API 才能工作。比如启动一个新服务、退出一个服务、组管理、timer 管理、等等。

这些都可以用某种服务的形式提供。但这样，就需要约定服务的通讯协议。而且，跨服务的调用也有其缺点，那就是一旦你做了一次跨服务的远程调用，即使没有性能上太多的损失，也会遇到对服务内状态改变不可预知的烦恼。就是说，你发起一个远程请求后，下一个收到的消息包，不一定是对这次请求的回应。它可能是其它服务对你的请求，这个请求很可能改变你自己的内存状态。

但是，这些基础服务（上面列举的服务启动器，组管理器，timer 模块等），变化性又可能很大，如果每个都设计成 C API 又不太合适。所以skynet选择了一个折中方案：

skynet 提供了一个叫做 skynet\_command 的 C API ，作为基础服务的统一入口。它接收一个字符串参数，返回一个字符串结果。你可以看成是一种文本协议。但 skynet\_command 保证在调用过程中，不会切出当前的服务线程，导致状态改变的不可预知性。其每个功能的实现，其实也是内嵌在 skynet 的源代码中，相同上层服务，还是比较高效的。（因为可以访问许多内存 api ，而不必用消息通讯的方式实现）

**4. 主要模块**

4.1 Master.so

主服务,就做两件事,回应名字的查询和在更新名字后,同步到其他节点。其他节点在启动时会把自己的harbor注册到master服务中。这样透过master服务就能把各个节点都连在一起,相互沟通了。每个节点之间都相互建立一条双向的通讯通道

4.2 harbor.so

节点服务,每个Skynet运行都是一个节点,在examples/main.Lua中使用harbor来配置节点的编号,skynet限制只有1到255个节点(保留0给系统内部使用),并把harbor的8bit值放在服务Id的最高8bit,因为服务Id是32bit的,所以skynet的本地服务只能用低24位来表示,一个skynet最多可以启动16M个服务模块。

4.3 logger.so

日志服务,一个简单的日志系统,可以用来记录服务的相关信息。

4.4 Gate.so

它的特征是监听一个TCP端口，接受连入的TCP连接，并把连接上获得的数据转发到skynet内部。Gate可以用来消除外部数据包和skynet内部消息包的不一致性。外部TCP流的分包问题，是Gate实现上的约定。

Gate会接受外部连接，并把连接相关信息转发给另一个服务去处理。它自己不做数据处理是因为需要保持gate实现的简洁高效。C语言足以胜任这项工作。而包处理工作则和业务逻辑精密相关，可以用Lua完成。

外部信息分两类，一类是连接本身的接入和断开消息，另一类是连接上的数据包。一开始，Gate无条件转发这两类消息到同一个处理服务。但对于连接数据包，添加一个包头无疑有性能上的开销。所以Gate还接收另一种工作模式：把每个不同连接上的数据包转发给不同的独立服务上。每个独立服务处理单一连接上的数据包。

或者，也可以选择把不同连接上的数据包从控制信息包（建立/断开连接）中分离开，但不区分不同连接而转发给同一数据处理服务（对数据来源不敏感，只对数据内容敏感的场合）。

这三种模式，分别称为watchdog模式，由gate加上包头，同时处理控制信息和数据信息的所有数据；agent模式，让每个agent处理独立连接；以及broker模式，由一个broker服务处理不同连接上的所有数据包。无论是哪种模式，控制信息都是交给watchdog去处理的，而数据包如果不发给watchdog而是发送给agent或broker的话，则不会有额外的数据头（也减少了数据拷贝）。识别这些包是从外部发送进来的方法是检查消息包的类型是否为PTYPE\_CLIENT。当然，也可以自己定制消息类型让gate通知你。

另一个重要组件叫Connection。它和Gate不同，它负责从skynet内部建立socket到外部服务。

Connection分两个部分，一部分用于监听不同的系统fd（Filedescriptor

）的可读状态，是用epoll实现的。如果在没有epoll支持的环境（比如freebsd下），可以实现一个替代品。收到这个连接上的数据后，会把所有数据不做任何分包，转发到另一个服务里去处理。和gate的行为不太一致，这是因为onnection多用于使用外部第三方数据库，很难统一其分包的格式。

另一部分是Lua相关的底层支持库，可以用于建立连接，以及对连接上数据常用的分包规则。

对于Lua层的设计，Lua是skynet的标准配置，不是必须的，但实际上被用在很多部分。当然完全可以用另一种语言比如python来替代掉Lua，但是Lua的性能相对更高。

在Lua的底层，skynet封装了skynet最基本的C语言API。但是开发人员不必工作在这些底层API上，以C语言的思维来编写服务。

Lua的coroutine可以把一个个在C层面分离的callback调用串成逻辑上连续的线索。当一个由Lua编写的服务接收到一个外部请求时，对应的底层callback函数就会被调用，既而转发到相应的Lua虚拟机中。skynet的Lua层会为每个请求创建一个独立的coroutine。

一旦在处理这个请求的coroutine中产生远程调用，即发出一个消息包，coroutine就会挂起。在C层面上，这次callback函数正常返回了。但在Lua中，则会记录下这个发出的消息包的session，将session和挂起的corutine记录在一张对应表中。随后，一旦收到回应包里有相同的session，对应的coroutine则被唤醒然后恢复。

每个服务使用不同的协议组，则是在底层由type参数进行区分。在Lua层，可以为每个不同的type编写不同的dispatch函数。默认提供RESPONSE消息的处理方法，每个独立的Lua服务，都需要去实现自己可以支持的协议类型的处理函数。

例如，源码已经提供了[一种远程对象的支持方法](http://blog.codingnow.com/2012/08/dev_note_25.html) ，如果要使用它，只需要require对应的Lua模块，其处理函数就被自动注入消息分发器中。

大多数Lua服务也可以使用一种简易的消息编码协议，称为Lua协议。因为它仅仅是简单的把Lua支持的类型系列化起来，另一个Lua服务可以顺利的解开它们。这样，看起来请求（用Lua编写的）远程服务和调用本地函数一样简单。

如果考虑到跨语言兼容性的话，也可以使用文本协议。从C服务去请求一个Lua服务处理起来可能更简单。当然用的更多的是反过来的场合，用Lua去调用C编写的一些基础服务。若是使用方便Lua规范的编码方式，C服务的编写就会相对复杂。而空格分割的字符串参数，C语言则可以用sscanf轻松分割开。

4.5 SnLua.so

Lua服务,用于加载Lua写的skynet服务模块,而不仅仅限制于C语言才能写服务模块。十分核心和重要的模块。也可以换成python等其他脚本引擎。

参考文献

[1] 杨玲.一种高性能网络游戏服务器架构设计[j].网络安全,2010,4.

[2]张渊,崔滨,等.线程池技术在网络游戏服务器中的应用[j].微 计 算 机 信 息,2006,22.

[3]王华峰, 张新家,等.三层结构的网络游戏服务器设计及其性能分析[j].计算机工程与应用,2007,43.

[4]王瑞彪,李凤岐,等.基于IOCP机制的网络游戏服务器通信层的实现[j].计算机工程与应用,2009,45.

[5]崔滨,万旺根,等.基于EPOLL机制的LINUX网络游戏服务器实现方法[j].微计算机信息,2006,22.