

硕 士 研 究 生 读 书 报 告



题目Skynet:基于Actor模式的服务端框架

作者姓名 陈军

作者学号 2165113

指导教师 李启雷

学科专业 移动互联网与游戏开发

所在学院 软件学院

提交日期 二○ 17 年 1月

Skynet：A Lightweight Concurrent Server Framework Based On The Actor Model

Major Subject: Software Engineering

Advisor: QiLei Li

By

Chen jun

Zhejiang University, P.R. China

2017

摘要

Skynet是一个专为在线游戏服务器打造的轻量级框架，但应用场景并不局限于游戏服务器，框架基于Actor模型，把业务拆分成一个个服务，每个服务可以看成一个Actor ，它们之间只能通过收发消息进行通信。底层核心用C实现，上层业务常用Lua实现。本文主要是深入学习和研究开源服务端框架Skynet，首先介绍框架的发展历史、框架的设计理念，然后对部分核心代码进行简单的剖析，再分析框架的优缺点和适用范围，最后对Skynet未来的进行展望。

**关键词**：Actor，并发，消息队列，集群，lua VM

Abstract

Skynet is a specially designed for online game server lightweight frameworks, but application scenario is not confined to the game server, framework based on the Actor model, the business split into service one by one, each service can be seen as an Actor, can only communicate by sending and receiving messages between them.The underlying core C implementation, the upper business commonly used Lua implementation.This paper is the in-depth study and research of open source server-side framework Skynet, first introduces the development history of framework, the framework design idea, and then to part of the core code carries on the simple analysis, the analysis framework of advantages and disadvantages and applicable scope.

**Keywords：**Actor，concurrent，message queue，cluster，lua VM

一、引言

随着计算机硬件高速发展，我们拥有更多的核心、更多的线程和带宽，但是我们编写的软件如何充分利用硬件带来的优势、提高服务端性能，特别是在不改变现有高层业务的代码基础上，那么选用一个性能高效、支持高并发的服务端框架成为关键。考虑到游戏领域的特点，服务器除了高性能的需求，还需要能够进行快速的业务逻辑的开发。

国内目前的开源游戏服务端框架不多，主要有NetEase的pomelo和云风的Skynet。pomelo是基于Node.js的高性能、分布式游戏服务框架。Skynet是基于Actor模型的轻量级并发框架，核心代码5k左右，充分结合C和Lua的优势，提供了服务端底层服务的通信机制，并在上层封装了网关、数据库等库并以服务的形式提供。

**二、**发展历史

Skynet开发的想法最早源于2010年，作者当时离开网易，自己创业，决定重新开发一个服务端函数库，但后来发现开发成一个框架更方便，于是在2011年12月基于Erlang的第一版Skynet完成，后来应用到正在开发的项目中发现

很多性能上的问题，作者便决定用自己最熟悉的语言重新实现，在2012年7月基于C的实现完成，2012年8月1号开源发布，2014年4月22号v0.1.0发布，

2014年11月28日v0.9.1发布，2015年12月v1.0.0 RC版发布。Skynet从一开始便是开源的，截止到目前Skynet在Github上已有Star数4615，Fork数2102。

**三、设计理念**

Skynet发展到现在，一直强调和追求的是高效和并发的性能，充分发挥计算机硬件资源的优势。Skynet采用Actor模型的思想，把业务拆分成一个个具体的服务，包括处理逻辑和数据结构，每个服务可以看成一个Actor，它们之间只能通过消息进行通讯。Skynet底层核心做的工作主要就是处理消息的接受与发送，可以说Skynet的核心就是围绕着消息来工作的，但没有规定和限制具体的通通信协议，可以使用Google 的Protobuf，也可以使用框架提供的Sproto，当然也可以自定义协议格式。用作者的话来说，可以把Skynet看成一个操作系统，可以调度数千个Lua VM，每个Lua VM可以看出这个Skynet操作系统的一个独立的进程。

1. Actor模型

计算机科学中的Actor模型是并行计算中的一个数学模型，它把Actor视为并行计算的基元。Actor可以响应接收到的消息，并作出响应的动作，发送消息，创建其他Actor，可以修改自身的状态，但是Actor与Actor之间只能通过消息进行通信和相互影响，不关心其他Actor是否存在、是否可靠，这种机制可以避免对锁的更多需求。

在并发编程中，Actor模型是一种常见且优势明显的模型，很多开 发语言都直接提供了原生的Actor模型，例如Erlang，Scala等。一个Actor可以看作是一个独立的实体，它们之间是毫无关联的，但是它们之间可以通过且只能通过消息进行通信，消息的类型可以是任意的，消息的内容也可以是任意的。

Actor在很多语言实现中，都以轻量级进程的形式存在，占用资源少，创建速度快。一个Actor就是一个执行上下文，绑定了Actor的相关数据，实际运行时被调度到真正的VM线程中运行。基于Actor模型的万物皆Actor的思想，我们可以抽象分布式的计算，把任务切分到细粒度，并发完成任务。

1. 服务

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。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。每个服务可以看成一个Actor，服务之间只能通过收发消息进行通信。每个服务对应一个handle，一个handle标识一个服务，同时每个handle可以对应多个名字，可以提供更加方便的使用与交流。每个服务首先都要注册到框架中，从Skynet底层框架来看，每个服务可以看成一个消息处理器，当有需要处理的消息时，才会通过回调函数执行自己的处理逻辑，否则处于挂起状态。服务的生命周期分为三个阶段：创建阶段、初始化阶段和销毁阶段，分别对应服务的create、init、release函数。

1. 消息

在Skynet中，服务之间通信的消息称为Skynet消息，Skynet预定义了一组消息类型，主要有响应消息、网络消息、文本消息、错误消息等。响应消息不需要特别处理，由框架的基础库管理，用来调度服务内的coroutine。网络消息也不必直接处理它，Skynet对开源项目socket-server进行了一层封装，提供了一套简单的socket api。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,可以驱动业务逻辑。

1. 网络

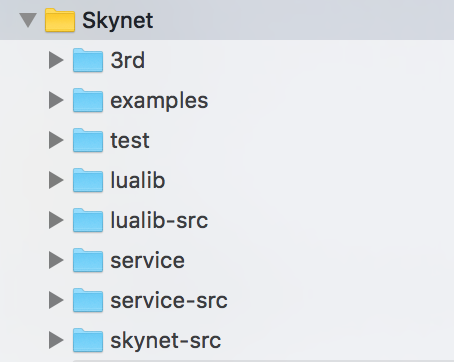
作为网络服务器框架，必然有封装好的网络层，对于 skynet 更是必不可少。由于 skynet 模拟了一个简单的操作系统，它最重要的工作就是调度数千个服务，如何让服务挂起时，尽量减少对系统的影响就是首要解决的问题。我们不建议你在 skynet 的服务中再使用任何直接和系统网络 api 打交道的模块，因为一旦这些模块被网络 IO 阻塞，影响的就不只是该服务本身，而是 skynet 里的工作线程了。skynet 会被配置成固定数量的工作线程，工作线程数通常和系统物理核心数量相关，而 skynet 所管理的服务数量则是动态的、远超过工作线程数量。skynet 内置的网络层可以和它的服务调度器协同工作，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

1. 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。

**四、部分核心源码的剖析**

Skynet源码可在Github上下载，主要的目录结构如下如所示，



其中3rd主要是一些第三方库，包括作者修改过的Lua版本、jemalloc库、加密库；lualib主要是用lua封装好的一些lua库，包括http库、数据库库以及skynet库等；lualib-src主要是用c封装的以.so形式供lua调用的库；service主要是框架提供的一些基础服务，包括启动服务bootstrap.lua，网关服务gate.lua等；service-src主要是c实现的一些基础服务；skynet-src则是整个框架的最核心、最底层的实现。本文重点研究和剖析这部分代码。

前文提到，Skynet主要是围绕着消息来工作的，那么先看消息是如何定义以及是如何存储和转发的。

Skynet中的消息定义如下：

struct skynet\_message {

uint32\_t source;

int session;

void \* data;

size\_t sz;

};

其中source表示消息的发送地址，用32位正整数表示，其实就是接受消息的服务对应的handle，session用来区分不同的响应消息，data表示具体的消息内容，sz表示数据大小，其中高8位表示type，用于标识消息的类型，也就是说消息内容大小最多只有16M。

整个Skynet框架维护的消息队列是一个二级结构，一个全局消息队列global message queue和若干个局部消息队列loacl message queue,全局队列由Skynet节点维护，局部消息队列由每个服务维护，也就是说每个服务拥有自己的私有消息队列。具体的定义如下：

struct message\_queue {

struct spinlock lock;

uint32\_t handle;

int cap;

int head;

int tail;

int release;

int in\_global;

int overload;

int overload\_threshold;

struct skynet\_message \*queue;

struct message\_queue \*next;

};

struct global\_queue {

struct message\_queue \*head;

struct message\_queue \*tail;

struct spinlock lock;

}

从定义中我们可以知道，local message queue采用动态数组存储消息skynet\_message,同时记录队列容量cap、队首head、队尾tail、负载阈值、负载值、是否在全局队列中的标记in\_global、所属服务对应的handle；global message queue采用单链表的方式存储local message queue，维护一个队首和队尾指针。作为队列这种数据结构，自然支持基本的pop、push操作。此外，还包括必要的资源释放工作，支持基本的数据的修改和读取操作。这里具体分析几个典型的操作。

void

skynet\_mq\_push(struct message\_queue \*q, struct skynet\_message \*message) {

assert(message);

SPIN\_LOCK(q)

q->queue[q->tail] = \*message;

if (++ q->tail >= q->cap) {

q->tail = 0;

}

//如果队列满，则扩大队列大小为当前的2倍

if (q->head == q->tail) {

expand\_queue(q);

}

if (q->in\_global == 0) {

q->in\_global = MQ\_IN\_GLOBAL;

skynet\_globalmq\_push(q);

}

这段代码是服务消息队列的push操作，首先把消息存到队列的尾部，如果这时队尾超过队列的容量，那么将队尾指向0；如果这时队列满载，那么扩大队列容量为当前2倍，如果服务消息队列不在全局队列中，那么将它push到全局队列中。从这里我们可以发现，服务消息队列其实是一个循环队列，支持动态扩容。下面是服务消息队列的扩容操作代码。

//扩大服务消息队列

static void

expand\_queue(struct message\_queue \*q) {

struct skynet\_message \*new\_queue = skynet\_malloc(sizeof(struct skynet\_message) \* q->cap \* 2);

int i;

for (i=0;i<q->cap;i++) {

new\_queue[i] = q->queue[(q->head + i) % q->cap];

}

q->head = 0;

q->tail = q->cap;

q->cap \*= 2;

skynet\_free(q->queue);

q->queue = new\_queue;

}

扩容代码很简单，重新申请2倍于原来大小的内存空间，把原来的数据拷贝到新分配的空间，修改相关字段，释放原来的内存空间。

//服务消息队列的pop

int

skynet\_mq\_pop(struct message\_queue \*q, struct skynet\_message \*message) {

int ret = 1;

SPIN\_LOCK(q)

if (q->head != q->tail) {

\*message = q->queue[q->head++];

ret = 0;

int head = q->head;

int tail = q->tail;

int cap = q->cap;

if (head >= cap) {

q->head = head = 0;

}

int length = tail - head;

if (length < 0) {

length += cap;

}

while (length > q->overload\_threshold) {

q->overload = length;

q->overload\_threshold \*= 2;

}

} else {

// reset overload\_threshold when queue is empty

q->overload\_threshold = MQ\_OVERLOAD;

}

if (ret) {

q->in\_global = 0;

}

SPIN\_UNLOCK(q)

return ret;

}

这段代码是服务消息队列的pop操作，我们可以发现，作者在这个地方检查队列负载的情况，同时在队列过载的情况下，成倍扩大负载阈值。

知道消息的定义和消息队列的定义后，我们再来看看handle的定义和基本操作。

// reserve high 8 bits for remote id

#define HANDLE\_MASK 0xffffff

#define HANDLE\_REMOTE\_SHIFT 24

结合两个宏定义和注释，我们可以知道handle是用32位正整数表示，其中高8位用来表示远程节点的id，也就是harbor，所以一个Skynet节点最多只能有16M个服务。

struct handle\_name {

char \* name;

uint32\_t handle;

}

struct handle\_storage {

struct rwlock lock;

uint32\_t harbor;

uint32\_t handle\_index;

int slot\_size;

struct skynet\_context \*\* slot;

int name\_cap;

int name\_count;

struct handle\_name \*name;//动态分配的数组，按name字典序排序

};

所有handle保存在handle\_storage中，以handle\_name的形式，即handle-name对的形式。handle\_storage同时还记录了handle\_name的容量、个数，对应的harbor，所有的skynet\_context（简单理解就是服务对应的上下文，后面详细介绍）。

//插入一个name-handle映射，如果存在则不插入

const char \*

skynet\_handle\_namehandle(uint32\_t handle, const char \*name) {

rwlock\_wlock(&H->lock);

const char \* ret = \_insert\_name(H, name, handle);

rwlock\_wunlock(&H->lock);

return ret;

}

这段代码是插入一个name-handle的映射，\_insert\_name的具体操作是以name为关键字二分查找，如果已存在名字那么不作任何修改，如果不存在那么就生成一个handle\_name并插入到正确的位置上。从这里我们可以知道，一个name只对应一个handle，但是一个handle可以对应多个名字。

uint32\_t

skynet\_handle\_register(struct skynet\_context \*ctx) {

struct handle\_storage \*s = H;

rwlock\_wlock(&s->lock);

for (;;) {

int i;

for (i=0;i<s->slot\_size;i++) {

uint32\_t handle = (i+s->handle\_index) & HANDLE\_MASK;

int hash = handle & (s->slot\_size-1);

if (s->slot[hash] == NULL) {

s->slot[hash] = ctx;

s->handle\_index = handle + 1；

rwlock\_wunlock(&s->lock)；

handle |= s->harbor;

return handle;

}

}

assert((s->slot\_size\*2 - 1) <= HANDLE\_MASK);//一个harbor下的handle最多有16M个

struct skynet\_context \*\* new\_slot = skynet\_malloc(s->slot\_size \* 2 \* sizeof(struct skynet\_context \*));

memset(new\_slot, 0, s->slot\_size \* 2 \* sizeof(struct skynet\_context \*));

for (i=0;i<s->slot\_size;i++) {

int hash = skynet\_context\_handle(s->slot[i]) & (s->slot\_size \* 2 - 1);

assert(new\_slot[hash] == NULL);

new\_slot[hash] = s->slot[i];

}

skynet\_free(s->slot);

s->slot = new\_slot;

s->slot\_size \*= 2;

}

}

这段代码是给skynet\_context注册一个唯一的handle，透过代码我可以发现作者不是通过简单的数组映射，而是用散列映射，目的是为了更加有效率的不复用用曾经用过的handle。具体是每次分配新的 handle 时，首先递增上次分配出去的 handle（允许回绕，但会跳过 0 。0 被当成无效 handle ）。然后检查 散列表中对应的 slot 是否有冲突，若有冲突就继续递增。如果散列表的池不够用了，则成倍扩大池，并将内部的数据重新散列一次。这样虽然会浪费一些 handle，但是可以保证 hash 表类的 key 永远不发生碰撞，这样可以让查询速度最快。hash 表的实现也相对简单一些。

除此之外，还提供了根据名字查找对应的handle、根据handle查找对应的context、注销指定的handle、注销所有的hanlde等操作。

struct skynet\_context {

void \* instance;

struct skynet\_module \* mod;

void \* cb\_ud;

skynet\_cb cb; //消息回调处理函数

struct message\_queue \*queue; //消息队列

FILE \* logfile; //日志文件

uint64\_t cpu\_cost; // in microsec

uint64\_t cpu\_start; // in microsec

char result[32];

uint32\_t handle; //对应的handle

int session\_id;

int ref; //引用计数

int message\_count; //消息个数

bool init;

bool endless;

bool profile; //标识是否可配置

CHECKCALLING\_DECL

};

skynet\_context这个数据结构是服务对应的上下文，它包括服务全部的信息和相关辅助数据，主要有skynet\_module、回调处理函数、私有的消息队列、日志文件句柄、对应的handle，引用计数ref等。

struct skynet\_node {

int total; //总的模块数量

int init;

uint32\_t monitor\_exit;

pthread\_key\_t handle\_key;

bool profile; // default is off

};

static struct skynet\_node G\_NODE;

skynet\_node这个数据结构定义了skynet节点，一个skynet节点对应一个进程，可以包括多个服务，每个服务可以看成是一个线程，这个结构包括了总的服务数量total、是否初始化标识等。

//分发消息，由worker线程调用，weight表示每个服务的消息处理权重

struct message\_queue \*

skynet\_context\_message\_dispatch(struct skynet\_monitor \*sm, struct message\_queue \*q, int weight) {

if (q == NULL) {

q = skynet\_globalmq\_pop();

if (q==NULL)

return NULL;

}

uint32\_t handle = skynet\_mq\_handle(q);

struct skynet\_context \* ctx = skynet\_handle\_grab(handle);

if (ctx == NULL) {

struct drop\_t d = { handle };

skynet\_mq\_release(q, drop\_message, &d);

return skynet\_globalmq\_pop();

}

int i,n=1;

struct skynet\_message msg;

for (i=0;i<n;i++) {

if (skynet\_mq\_pop(q,&msg)) { //消息队列为空

skynet\_context\_release(ctx);

return skynet\_globalmq\_pop();

} else if (i==0 && weight >= 0) {

n = skynet\_mq\_length(q);

n >>= weight;

}

int overload = skynet\_mq\_overload(q);

if (overload) {

skynet\_error(ctx, "May overload, message queue length = %d", overload);

}

skynet\_monitor\_trigger(sm, msg.source , handle);

if (ctx->cb == NULL) {

skynet\_free(msg.data);

} else {

dispatch\_message(ctx, &msg);

}

skynet\_monitor\_trigger(sm, 0,0);

}

assert(q == ctx->queue);

struct message\_queue \*nq = skynet\_globalmq\_pop();

if (nq) {

// If global mq is not empty , push q back, and return next queue (nq)

// Else (global mq is empty or block, don't push q back, and return q again (for next dispatch)

skynet\_globalmq\_push(q);

q = nq;

}

skynet\_context\_release(ctx);

return q;

}

这段代码用来分发消息，由worker线程不断的循环调用，其中的形参weight表示每个服消息队列处理的权重，这样做可以自定义消息处理的优先级。分发步骤主要如下： 首先通过消息队列获取对应的Handle，再通handle获取对应服务的context，如果对应的context为空，则使用自定义的方法drop\_message释放该消息队列，从全局队列中pop出下一个服务消息队列并返回；如果对应的context不为空，则从服务消息队列中pop出下一个消息，如果消息队列为空，则释放该context，从全局队列中pop出下一个服务消息队列并返回;如果消息队列不为空，则通过weight修改循环控制变量的值，然后检查队列是否过载，过载的话进行错误处理。然后分发消息，由服务的回调函数来处理消息。

//更加名字查找handle

uint32\_t

skynet\_queryname(struct skynet\_context \* context, const char \* name) {

switch(name[0]) {

case ':':

return strtoul(name+1,NULL,16);

case '.':

return skynet\_handle\_findname(name + 1);

}

skynet\_error(context, "Don't support query global name %s",name);

return 0;

}

这段代码是根据名字查找对应的Handle，如果名字是以：开头则名字就包含了handle，如果是以.开头，则在handle\_storage中查找。

//删除context

static void

delete\_context(struct skynet\_context \*ctx) {

if (ctx->logfile) {

fclose(ctx->logfile); //关闭文件

}

skynet\_module\_instance\_release(ctx->mod, ctx->instance);

skynet\_mq\_mark\_release(ctx->queue);//标记消息队列释放

CHECKCALLING\_DESTROY(ctx)

skynet\_free(ctx);

context\_dec();

}

这段代码用来删除context，如果context中的日志文件不为空，则需要关闭文件，然后调用服务的release函数进行释放，然后将标记消息队列释放。

struct skynet\_context \*

skynet\_context\_release(struct skynet\_context \*ctx) {

if (ATOM\_DEC(&ctx->ref) == 0) {

delete\_context(ctx);

return NULL;

}

return ctx;

这段代码用来释放context， 减小context对应的引用计数，只有当引用计数为0时才真正去进行删除操作delete\_context。

**五、应用现状及优缺点**

Skynet是一个轻量级的专为游戏服务器打造的框架，但它不局限于游戏领域，目前已知的应用场景除了游戏之外，还包括金融、交换机、web应用等。其中游戏使用Skynet框架主要包括天天来战、陌陌争霸、枪战英雄等。

优点主要有：非常轻量级，核心代码只有5000行左右，可控性好；高效的性能对并发的支持；基于Actor模型，避免了对更多锁的需求，使得使用者不用过多去考虑传统多线程中的可能会面临的死锁等问题；运行效率高，在需要效率优化的地方可以用C实现；上层业务逻辑开发简单快速；

缺点主要有：提供使用的库太少，更像是框架的框架；作为一个服务器框架，没有完备的服务器开发相关的支持，很多工作都需要自己来做；对具有大量数据传输需求的应用的支持不够好；缺乏完善的说明文档。

**六、未来发展的展望**

Skynet在Github上的表现还是很活跃的，有很多人参与到这个开源项目。希望Skynet在未来能够进一步打磨好它的底层核心，使其更加健壮、完善。另外作为服务器框架，希望它能提供一套完整的服务器开发方案，提供更多的工具、库支持；提供完整的API文档和使用说明。当然我也会去尝试贡献自己的一份力量，无论是提bug还是添加新功能，希望能够为Skynet的强大做出自己的努力。

参考文献

[1] Agha G A. Actors: A model of concurrent computation in distributed systems[R]. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[2]《Unix networking programming volume 1 》section3.2 p68-74

[3] Hansen P B. The architecture of concurrent programs[M]. Prentice-Hall, Inc., 1977.

[4] <https://github.com/cloudwu/skynet/wiki>

[5] https://en.wikipedia.org/wiki/Actor\_model