Skynet 阅读笔记

- 彩笔 . 2014-06-03

Skynet 初探

Skynet 描绘

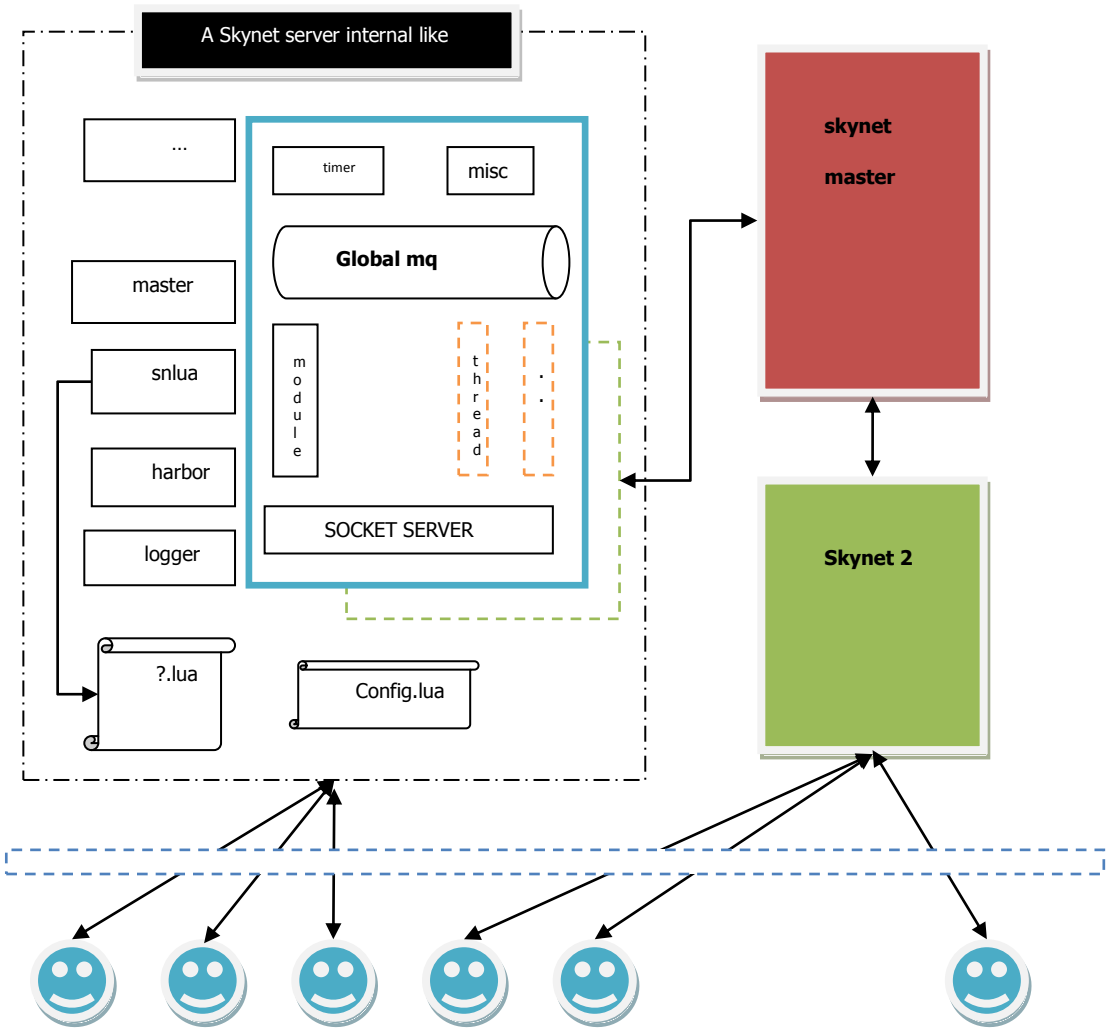
早期的 skynet (v0) 使用 erlang + C driver 开发 ， erlang 多进程 ，似乎是每个 player 有一个 agent 服务，在 40v40 时，在台式机上测试时存在困难 。于是 cloud 滋生了重新考虑之前的设计的念头，当时，同图超过 50 人区域 player 的状态同步时，由于设计方案是每个 agent 的状态变化都需要通知到其他 agent ，造成大量内部消息复制 ，这个是一个明显可优化点，惦念了下，设计了一个共享无锁队列，避免了无谓复制 ，内部消息包降低 90%，整体性能提升 15%左右 。后来随着项目细节逐渐展开，skynet 似乎开始膨胀，性能测试/代码复杂度似乎不甚满意，cloud 反复思虑 skynet 需求，认为目前 skyent (v0) 过于复杂，尤其底层 erlang 可能存在一些不透明的地方，项目中使用各种语言 （ erlang ， C ， lua ），为了完成不同语言 binding ，做了很多额外的（似乎是无价值）的代码 。为了验证 skynet (v0) 的性能，cloud 编写了一个简单的 skynet (v1) ， v1 完全使用 C 做底层通信框架（摒弃了之前 ZeroMQ 提供通信的部分），提供 lua 接口编写应用服务。整体代码在 4000 C+1000 Lua 完成了和之前 v0 一样的功能 ，后来逐步演进到现在的版本 … (见云风博客博文描绘)

开发笔记：http://blog.codingnow.com/cloud/DevNotes

提供一个简洁，稳定，高效，高可用的分布式服务开发框架 .

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

不解决：*Skynet 的消息传递都是单向的，以数据包为单位传递的。并没有定义出类似 TCP 连接的概念。也没有约定 RPC 调用的协议。不规定数据包的编码方式，没有提供一致的复杂数据结构的列集 API 。*



Lua

Lua 是一个娇小玲珑的解释性语言，使用标准 C 实现，可以完美的嵌入在有 C binds 的宿主语言中 。

沙盒保护：宿主语言在执行 lua 代码时，或者 lua 代码执行 lua 代码时，可以完全定制执行代码的运行环境，并捕获语言层面的异常 。

Coroutine：lua 支持的非常轻量的非抢占式协程(线程)，可以讲流散各处的异步处理逻辑代码通过 coroutine 串行组织，有助于编写更稳定正确安全的逻辑 。

Luastate：Lua 和宿主语言数据交互的虚拟栈 。

Skynet 中的惯用法：（待总结?）

无锁队列

指令并行访问共享数据时，当不能确保数据访问时序，会出现数据竞争，可能出现逻辑错误，需要同步手段（协议）明确一段数据的并行访问状态 。

原子操作：一般高级语言的语句执行可能不是原子操作，并行执行时，无法确定并行执行语句的时序 ，但是平台提供了一些某些“高级语言语句”可以保证原子执行，比如 CAS ， FAA 指令 。

队列： FIFO 结构将数据串行化，可以表达生产者消费者模型，多线程间共享数据（包）的一种方式 ，匹配不同处理速度。

多线程陷阱

线程开销：

Pthread 库，创建线程空间开销，时间开销，最终系统负载开销（待总结）。

同步代价：

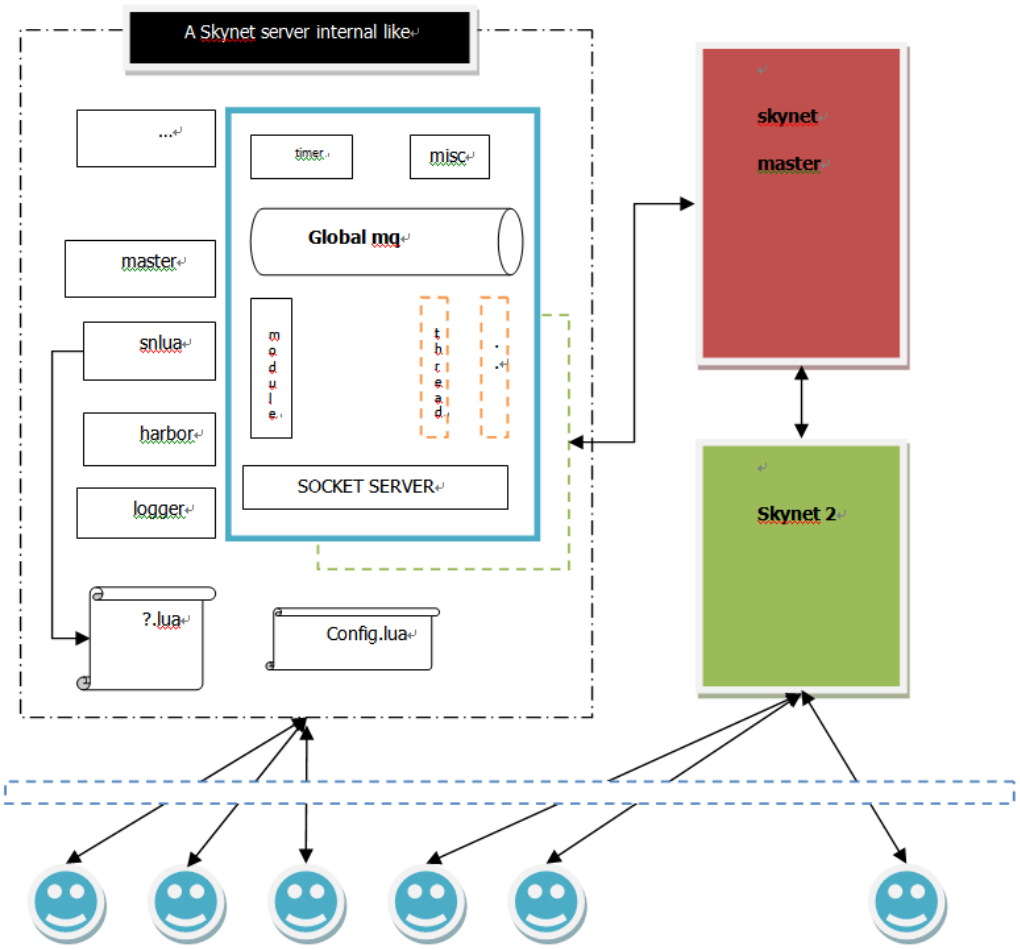
Mutex ， pthread_cond，semaphore 作用以及代价，应用场景（待总结）。

安全(稳定)性：

数据共享造成的境界条件，并行执行时序可能性，数据溢出，核心数据保护，单模块 core （待总结）。

Skynet 进程框架

结构回顾



分布式消息存储转发

Lua 逻辑处理器

关键结构

Skynet_context ：Skynet 中的通信实体环境，每个 context 都有一个唯一 handle 。

Skynet_module：C 服务的模块描述，skynet 中基础服务有 C 编写，可以通过 so 动态加载 。

Timer:定时器服务，到期后会产生定时器消息，near 和 level t 记录分级到期时间 。

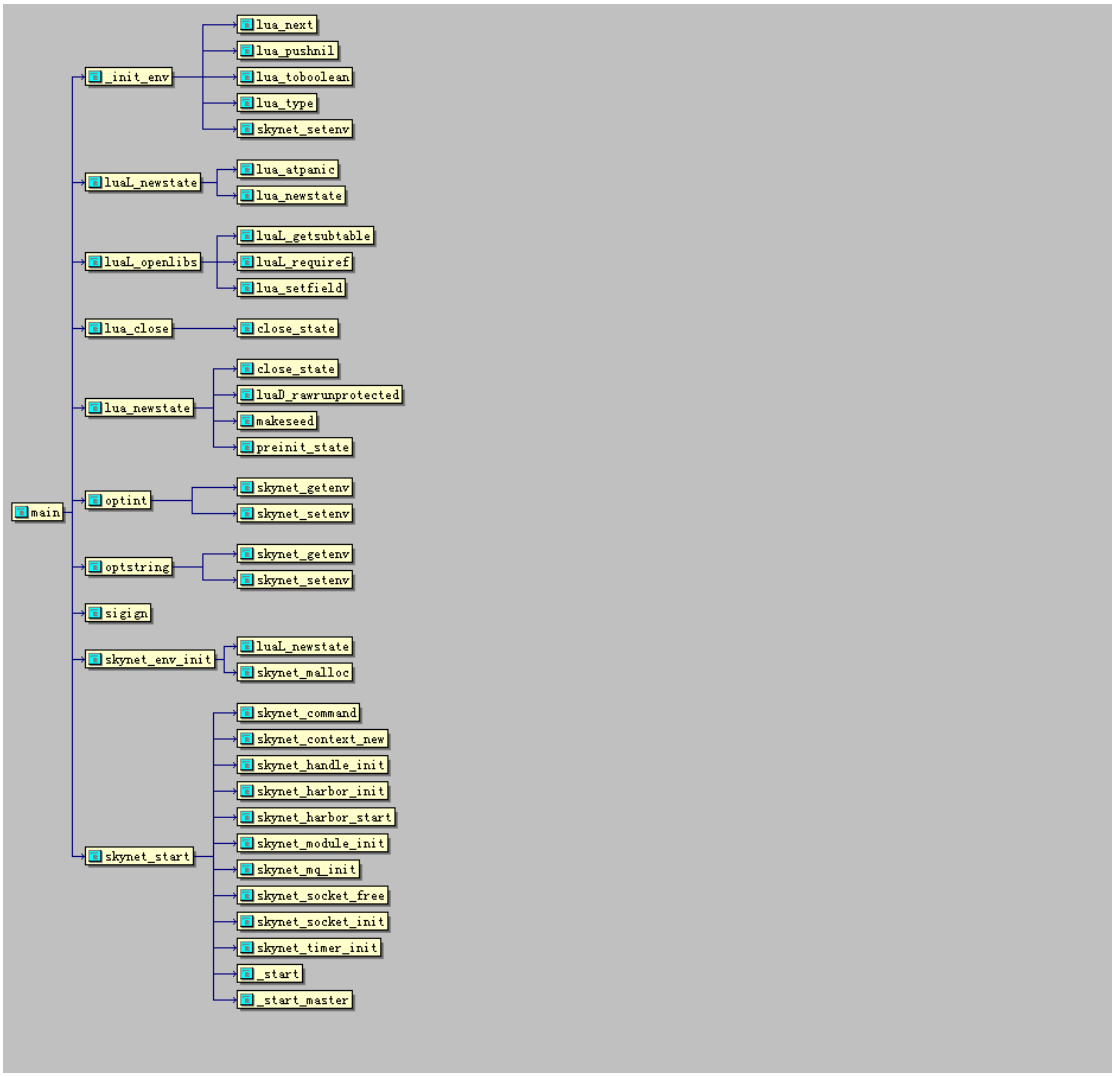
message_queue：消息队列 ， 每个 handle-context 都会有一个 mq .

skynet_message：在 skynet 的通信实体间传递的消息

socket_server：一个 TCP SOCKET SERVER 包含一个高优先级 pipe 控制通道（其他服务可以通过控制命令协议执行一些常规功能，具体可以见 ctr_cmd ），和一堆由 skynet 通信需要或者命令建立起来的 tcp server 。

Socket： 每个 TCP 连接维护，包括 opaque (通常是转发实体的 handle) ，发送数据分为高和低 2 个优先级队列 。

处理流程



Skynet 启动时主要读取解析设置配置，准备环境．

启动初期，会初始化一系列全局结构包括：E(全局 luastate)， HARBOR (当前 skynet 的 harbor 号)，H (全局 handle storage 结构，管理所有 context),Q (全局(两级)消息队列)，M (全局 modules，存储所有 module 信息)，TI(全局定时器)，SOCKET_SERVER (TCP SOCKET Server)，REMOTE (harbor 服务)．

Skynet_context_new 会创建一个 skynet_context，在 skynet，每个注册在 skynet 的服务都会分配一个 context 用于记录当前服务的状态 。

创建 context 时，会根据传入的 service name 和 param 在配置的 so 目录中查找到对应 name 的 so ， 加载并在合适的时机执行其中的 <name>_create, <name>_init 和<name>_release 函数 。

Skynet 会启动几个基础服务，依次是 logger，master （根据配置），harbor，snlua．这是 skynet 的基础服务，提供了核心功能分别描述如下。

Logger 是个提供日志输出的服务（异步日志）。

如果配置了 config->standalone，会启动 master 服务，参数为 config->standalone．此服务容后再表．（master 服务）

接着 启动配置所配置的 config->master．（master client）

最后启动 snlua (launcher), 这个是 skynet 的 lua 服务启动器，缩这个吧 skynet lua， launcher 启动后 执行控制命令 REG .launcher 注册名字 launcher 为刚才 通过 snlua 启动的 launcher．

最最后启动的是配置文件的 start lua 文件．即 snlua config->start

之后开始创建各个基础线程 ， 如下．

_start 会启动 3+thread 个线程，3 个基础线程分别是 monitor 线程，timer 线程和 socket 线程．接着启动配置所配的 thread 个 worker 线程．

每个线程职责描述如下 。

Monitor：

函数：_monitor

职责：监测每个 worker 线程消息是否出现无限循环

Timer：

函数：_timer

职责：更新定时器结构，给超时事件订阅者发送超时事件，唤醒所有 worker 线程

Socket

函数：_socket

职责：通过 SOCKET_SERVER 监听（收集）发给当前 skynet（harbor）上所有网络事件(也包括控制命令)，做初步转换解析(skynet_message 格式)，放入全局消息队列中.

Worker

函数：_worker

职责：不停从全局消息队列取出 1 条消息分发给注册在当前 skynet 的服务．

关键流程

初始化环境

Lua_path

Lua_cpath

Luaservice for skynet lua

Harbor,handle,mq,module,timer,socket

启动（bootstrap）

Logger.so init <config.logger>

Master.so init <config.master>

Harbor.so init master local

Launcher <= Snlua.so init launcher

REG .launcher launcher

Snlua.so init <config.start>

Thread：monitor，timer，socket

Config.thread (def:8)：woker

运行

异常状态监测 (monitor)：对每个 worker 线程的消息处理检测，是否出现了无限循环，即一次消息消息分发处理中，又一直出现了分发，且一段时间 5s 还未结束 。那么可以断定这个消息处理存在了无限循环 。

(网络)消息驱动 (socket) ：从全局 SOCKET_SERVER 中取(SOCKET)消息，分发 。SOCKET 消息是对 SS 收到数据一层封装，具体来说，更底层优先从命理通道取命理（select）并处理，然后通过 poller 取到所有 SOCKET 事件结合 SOCKET 现有状态解析为一个 SOCKET_TYPE_***，并生成一个 socket_message 作为处理结果返回给 skynet_socket_poll ， skynet 将这些 SOCKET 事件打包成类型为 PTYPE_SOCKET 的 skynet_message 压入到对应 context 的消息队列中 。

定时器 (timer)：每 2.5ms 唤醒有睡眠的线程(tick) ， 以 10ms 的精度更新 timer 表，更新算法是：

1. 将时间分为不同粒度 tick，每次可以根据当前时间直接定位到最近最小粒度 (near)，利用 hash 即可得到于当前时间到期的所有定时器，检测定时器到期事件，产生 PTYPE_RESPONSE 以及定时器回调 SESSION 的消息压入对应定时器 handle 的消息队列。
2. 将当前时间增加（tick），当前时间到了跨粒度单位时，需要将下个高粒度单位的定时器事件更新到最近定时器缓存中（将下个粒度定时器删除后全部重新插入） 。

   Skynet 中的定时器将时间分级，将最近 256 个 tick 放在 near 中，可以通过 hash 直接提取，将其他时间分为 4 个级别，最近{2^6，2^12,2^18,2^24}*256 个 tick ， 即最大可表达 2^32 tick = 13y .

   插入时间节点时：依次找寻最近是否是最近 256tick 区间，最近不同级的 2^6*256tick ,... 插入节点 .
3. 再次检测定时器是否到期时间 。

消息分发 (worker)：从全局消息队列中取出一个队列，取出一条消息分发，将队列再放回去 . 有消息继续，没消息休眠线程 ， 有 timer 会唤醒 。分发消息是交给

skynet_context 的 callback 处理，提供了 ctx，cb_ud，type，session （通过标签可以在 send 时自动分配），source，data．

流程总结

Skynet 框架底层完全用 C 实现，定义了通信实体(skynet_context)，和通信手段(消息转发)， 提供了基础服务样例 。代码风格精简至极 ，，， 通过 snlua 和 skynet 接口导出可以完全将 skynet 的功能扩展交给 lua 实现 ，哎，咋觉得这个代码风格看起来有点跳呢 。 。

Skynet 服务(模块)

服务描述

描述下 skynet 服务作用，原理，有什么优缺点，提供了哪些基础服务 。

服务接口

编写 C 服务时可用的接口集合，编写 lua 服务时可用的接口集合 ，注意点。

Lua 服务

梳理下 lua 服务执行过程，哪些注意点，分析下提供的 lua 例子服务，还有哪些 lualib

Skynet example

Watchdog

看下 ex 代码，读一读，。。。

Simple db

看下 ex 代码，读一读，。。。

Agent

看下 ex 代码，读一读，。。。

Skynet etc

代码风格

。。。

性能测试

todo

应用场景

需要 YY

扩展

需要 YY