skynet 入门 Quickstart

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 Framework

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。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。

另外， skynet 的 websocket 支持目前处于实验阶段，需要切换到 websocket 分支。

客户端 Client

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 Service

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 Message

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。

Each Skynet message contains 6 parts: message type, session, address of request Service, address of respond Service, the message C pointer, length of a message.

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 External Service

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，出于简化编译依赖的想法，skynet 的默认编译脚本并没有将 openssl 链接进去，而 https 支持需要它。如果需要支持 https ，需要额外

设置 `TLS_MODULE=ltls`。另外，你也可以用 nginx 制作一个 https 反向代理服务器，而不必直接使用 skynet 的 https 模块。

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 Cluster

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。