经过一个月， 我基本完成了 skynet 的 C 版本的编写。中间又反复重构了几个模块，精简下来的代码并不多：只有六千余行 C 代码，以及一千多 Lua 代码。虽然部分代码写的比较匆促，但我觉得还是基本符合我的质量要求的。Bug 虽不可避免，但这样小篇幅的项目，应该足够清晰方便修正了吧。

花在 Github 上的这个开源项目上的实际开发实现远小于一个月。我的大部分时间花了和过去大半年的 Erlang 框架的兼容，以及移植那些不兼容代码和重写曾经用 Erlang 写的服务模块上面了。这些和我们的实际游戏相关，所以就没有开源了。况且，把多出这个几倍的相关代码堆砌出来，未必能增加这个开源项目的正面意义。感兴趣的同学会迷失在那些并不重要，且有许多接口受限于历史的糟糕设计中。

在整合完我们自己项目的老代码后，确定移植无误，我又动手修改了 skynet 的部分底层设计。在保证安全迁移的基础上，做出了最大限度的改进，避免背上过多历史包袱。这些修改并不容易，但我觉得很有价值。是我最近一段时间仔细思考的结果。今天这一篇 blog ，我将最终定稿的版本设计思路记录下来，备日后查阅。

Skynet 核心解决什么问题

我希望我们的游戏服务器（但 skynet 不仅限于用于游戏服务器）能够充分利用多核优势，将不同的业务放在独立的执行环境中处理，协同工作。这个执行环境，最早的时候，我期望是利用 OS 的进程，后来发现，如果我们必定采用嵌入式语言，比如 Lua 的话，独立 OS 进程的意义不太大。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。这一点，Skynet 最终花了不到 3000 行 C 代码来实现核心层的代码，一个稍有经验的 C 程序员，都可以在短时间理解，做维护工作。

做为核心功能，Skynet 仅解决一个问题：

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。

Skynet 提供了名字服务，还可以给特定的服务起一个易读的名字，而不是用 id 来指代它。id 和运行时态相关，无法保证每次启动服务，都有一致的 id ，但名字可以。

Skynet 核心不解决什么问题

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 。

Skynet 原则上主张所有的服务都在同一个 OS 进程中协作完成。所以在核心层内，不考虑跨机通讯的机制，也不为单独一个服务的崩溃，重启等提供相应的支持。和普通的单线程程序一样，你要为你代码中的 bug 和意外负责。如果你的程序出了问题而崩溃，你不应该把错误藏起来，假装它们没有发生。至少，这些不是核心层做的事情。和操作系统不一样，操作系统会

认为用户进程都是不可靠的，不会让一个用户进程的错误影响到另一个进程。但在 Skynet 提供的框架内，所有的服务都有统一的目的，为游戏服务器的最终客户服务，如果有某个环节出了错都可能是致命的，没有必要被问题隔离开。

当然，这并不是说，最终用 Skynet 搭建的系统不具有健壮性，只是这些是在更高层去解决。比如，使用 Lua 的沙盒就可以隔绝大多数上层逻辑中的 bug 了。

简单说，Skynet 只负责把一个数据包从一个服务内发送出去，让同一进程内的另一个服务收到，调用对应的 callback 函数处理。它保证，模块的初始化过程，每个独立的 callback 调用，都是相互线程安全的。编写服务的人不需要特别的为多线程环境考虑任何问题。专心处理发送给它的一个个数据包。

熟悉 Erlang 的同学一眼就能明了，这就是 Erlang 的 Actor 模型。只不过，我嵌入了更为我熟悉的 Lua 语言。当然，如果查阅 Skynet 的代码就能发现，其实 Lua 并不是必须的，你完全可以用 C 编写服务模块。也可以方便的换成 Python 能其它可以嵌入 C 的动态语言。让 Lua 和 Python 共存也不难，这样就可以利用到我已经为 Skynet 编写的一些用 Lua 实现的基础服务了。

那么，为什么选择 Lua ?

最重要的原因是个人偏好。Lua 是我最熟悉和欣赏的语言之一。

它非常方便嵌入到 C 语言中协同工作。并且可以获得不错的执行性能。如果有必要，还可以引入 LuaJIT 来进一步提升运行速度。

它的运行时需要的必要库很少，让这种需要大量独立沙盒的系统更为轻量。启动和销毁一个服务都很快。

为了提供高效的服务间通讯，Skynet 采用了几点设计，获得了比多进程方案更高的性能。

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，以及你声称的数据包长度（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。

我们来看看 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
);
```

暂且不去关注 type 和 session 两个参数。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。我们可以在 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。

接下来，我想谈谈 session 和 type 两个参数。

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，我们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，我们正是这样做的，这个一致的通讯协议编码方式就是 google protocol buffer 。后来，我们发现，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。

session 是什么？

虽然 skynet 核心只解决单向的消息包发送问题，正如 ip 协议只解决 ip 包在互连网中从一个 ip 地址传输到另一个 ip 地址的问题。但是，我们的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，我们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。

type 的作用

正如上一节所写，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。所以我们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区分。

一开始的时候，我利用 session 的高位来区分这两种不同的消息。正的 session 号表示对以前发出的请求消息的回应，而负的 session 号表示对其的请求。后来，我发现只区分两种消息类型是不够的。

诚然，我们可以约定，整个系统统一使用一种消息编码协议。这样，所有服务间都不会有沟通障碍。这样，甚至 session 参数也可以编码在数据包中。那仅仅是一丁点效率问题。

但是，在真正的项目开发中，这其实做起来很难。尤其在使用第三方库的时候，你需要做很多低效的封装工作，才可以把交互协议都统一起来。而且，这个一致的编码协议也就成了系统底层约定的一部分，成为所有开发人员都需要了解的知识。像 google protocol buffer 这种协议，固然可以满足要求，但对于 C 语言层次上的开发还不够方便。对 lua 的开发也不够高效和方便。当然，为此，我也专门开发过对应的库，来解决性能和便捷性问题 。

如果我们退一步，看起来会好一点。那就是一个服务只使用一种消息编码协议，但不强求每个服务使用相同的编码协议。那么你去调用一个服务时，去适应对方的通讯协议即可。有了独立的 session 在消息包之外，你在处理不同协议的对外请求的回应包时，就可以用不同的编码方式来解码回应包了。这就是我在最初实现 C 版本的 skynet 时采用的方式。

在做老代码移植的工作中，我发现这样还是颇为掣肘。原来的基于 protobuffer 的协议设计的比较难受，如果我新写的服务需要提供相同的功能，就需要基于老协议来实现。受限于同一服

务只能使用一种消息编码方式，无法用更高效的方式实现新功能。有时为了一些兼容问题，我需要在 callback 函数中加了许多 filter ，把不同的协议转换为相同的形式，为此多写了许多无谓的代码，只为了解决那些历史遗留问题。虽然为此，我们已经内部达成一致意见，会慢慢淘汰大多数不合理的代码，但过渡时期却一定要维持一个可用的版本。

最终，我决定引入一个 type 参数。

其实，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，我限制了 type 的范围是 0 到 255 ，由一个字节标识。在实现时，我把 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 我们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，我们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。

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。

我们最终允许 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，我们就需要一个机构能够同步这些全局名字。

为此，我实现了一个叫做 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。

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，我们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，我们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，我们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。我希望的效果是 ，如果你不用这个机制（比如组播）就不必为之付出成本。所以，我在 skynet 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。

我用 lua 编写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。原本我并不打算把名字服务放在底层，但由于历史原因，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。所以我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）

---

skynet 的消息调度

Skynet 维护了两级消息队列。

每个服务实体有一个私有的消息队列，队列中是一个个发送给它的消息。消息由四部分构成：

```
struct skynet_message {
    uint32_t source;
    int session;
    void * data;
    size_t sz;
};
```

向一个服务发送一个消息，就是把这样一个消息体压入这个服务的私有消息队列中。这个结构的值复制进消息队列的，但消息内容本身不做复制。

Skynet 维护了一个全局消息队列，里面放的是诸个不为空的次级消息队列。

在 Skynet 启动时，建立了若干工作线程（数量可配置），它们不断的从主消息列队中取出一个次级消息队列来，再从次级队列中取去一条消息，调用对应的服务的 callback 函数进行出来。为了调用公平，一次仅处理一条消息，而不是耗净所有消息（虽然那样的局部效率更高，因为减少了查询服务实体的次数，以及主消息队列进出的次数），这样可以保证没有服务会被饿死。

用户定义的 callback 函数不必保证线程安全，因为在 callback 函数被调用的过程中，其它工作线程没有可能获得这个 callback 函数所熟服务的次级消息队列，也就不可能被并发了。一旦一个服务的消息队列暂时为空，它的消息队列就不再被放回全局消息队列了。这样使大部分不工作的服务不会空转 CPU 。

btw，在做这部分代码实现时，我曾经遇到过一些并发引起的 bug ，好在最终都解决了。这或许是整个系统中，并发问题最复杂的部分，但也仅仅是这一小部分了。不会让并发的复杂性蔓延出去。

Gate 和 Connection

以上提到的都是 skynet 内部的协作机制。但一个完整的游戏服务器避免不必和外界通讯。

外界通讯有两种，一是游戏客互端使用 TCP 连接接入 skynet 节点。如果你对游戏不关心，那换个角度看，如果你用 skynet 实现一个 web 服务器的话，游戏客户端就可以等价于一个浏览器请求。

另一个是第三方的服务，比如数据库服务，它接受一个或多个 TCP 连接。你需要从 skynet 内部建立一个 TCP 连接出去使用。虽然，完全可以编写一个以 skynet 接口规范实现的数据库，那会更高效，但现实中恐怕很难做到。能做的仅仅是实现一个内存 cache 而已。（比如，我用了不到 10 行 lua 代码，实现了一个简单的 key-value 的建议内存数据库的范例）

前者，我称为 gate 服务。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。我实现了一个 gate 服务，它按两字节的大头字节序来表示一个分包长度。这个模块基于我前段时间的一个子项目 。理论上我可以实现的更为通用，可以支持可配置的分包方案（Erlang 在这方面做的很全面）。但我更想保持代码的精简。固然，如果用 skynet 去实现一个通用的 web server ，这个 gate 就不太合适了。但重写一个定制的 Gate 服务并不困难。为 web server 定制一个 gate 甚至更简单，因为不再需要分包了。

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我们需要保持 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，我们可以用 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，我们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，我分别称为 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，你也可以自己定制消息类型让 gate 通知你。

Skynet 的基础服务中，关于集群间通讯的那部分，已经采用了 gate 模块作为实现的一部分。但是 gate 模块是一个纯粹的 skynet 服务组件，仅使用了 skynet 对外的 api ，而没有涉及

skynet 内部的任何细节。在 Harbor 模块使用 gate 时，启用的 broker 模块，且定制了消息包类型为 PTYPE_HARBOR 。

在开源项目的示范代码中，我们还启动了一个简单的 gate 服务，以及对应的 watchdog 和 agent 。可以用附带的 client 程序连接上去，通过文本协议和 skynet 进行交流。agent 会转发所有的 client 输入给 skynet 内部的 simpledb 服务，simpledb 是一个简易的 key-value 内存数据库。这样，从 client 就可以做基本的数据库查询和更新操作了。

注意，Gate 只负责读取外部数据，但不负责回写。也就是说，向这些连接发送数据不是它的职责范畴。作为示范，skynet 开源项目实现了一个简单的回写代理服务，叫做 service_client 。启动这个服务，启动时绑定一个 fd ，发送给这个服务的消息包，都会被加上两字节的长度包头，写给对应的 fd 。根据不同的分包协议，可以自己定制不同的 client 服务来解决向外部连接发送数据的模块。

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 的可读状态，这是用 epoll 实现的。如果在没有 epoll 支持的环境（比如 freebsd 下），可以很轻松的实现一个替代品。它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。这和 gate 的行为不太一致，这是因为 connection 多用于使用外部第三方数据库，我们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。

我试着编写了 Redis 的支持模块。Redis 采用的文本协议，比较好解析，也就容易说明这个东西的用法。我们可以从 redis 支持模块中利用 connection 服务监视一个 tcp socket ，一旦收到任何数据就发送回来。这就不必在服务中阻塞读取外部 TCP 连接。任何 skynet 内部服务都不建议阻塞使用外部 IO 的，那会造成 CPU 的浪费。

通过消息包的 type ，我们可以轻易识别出那些包是外部 tcp 连接上的数据块。使用提供好的 lua 模块，可以轻松的对这些数据分包（读一个指定字节数的数据块，或是读一个以回车结束的文本行）。Lua 的 coroutine 支持，可以轻松的在数据包并不完整时挂起，却不打断执行流程。

另一个使用 Connection 模块的例子是 console 服务。skynet 的开源部分实现了一个简单的 Console 模块，可以读取进程的标准输入，按回车分割，并以用户输入的文本行去启动一个 lua 编写的服务。代码很短，很容易理解其工作方式。

---

lua 层的设计

Lua 是 skynet 的标准配置，它不是必须的，但实际上被用在很多部分了。虽然完全可以用另一种语言比如 python 来替代掉 lua ，但我没有这个开发计划。

在 Lua 的底层，skynet 封装了 skynet 最基本的 C 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 可以帮助我们把一个个在 C 层面分离的 callback 调用串成逻辑上连续的线索。当 Lua 编写的服务接收到一个外部请求时，对应的底层 callback 函数被调用，既而转发到 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 中发生远程调用，即发出一个消息包，coroutine 会挂起。在 C 层面，这次 callback 函数正常返回了。但在 Lua 中，则是记录下这个发出的消息包的 session ，记录 session 和挂起的 corutine 在一张对应表中。之后，一但收到回应包里有相同的 session ，对应的 coroutine 则被唤醒 resume 。

每个服务可以使用不同的协议组，则是在底层由 type 参数区分的。在 lua 层，可以为每个不同的 type 编写不同的 dispatch 函数。默认仅提供了 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。

比如，我已经提供了一种远程对象的支持方法 ，你可以选择使用它，也可以选择不用。如果你想用它，只需要 require 对应的 lua 模块，其处理函数就被自动注入了消息分发器中。

大多数 lua 服务也可以使用一种简易的消息编码协议，我称为 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。