一个基于 skynet 框架开发的服务器，是由若干服务构成。你可以将 skynet 看成一个类似操作系统的东西，而服务则可以视为操作系统下的进程。但实际上，单个 skynet 节点仅使用一个操作系统进程，服务间的通讯是在进程内完成的，所以性能比普通的操作系统进程间通讯要高效的多。

skynet 框架是用 C 语言编写，所以它的服务也是用 C 语言开发。但框架已经提供了一个叫做 snlua 的用 C 开发的服务模块，它可以用来解析一段 Lua 脚本来实现业务逻辑。也就是说，你可以在 skynet 启动任意份 snlua 服务，只是它们承载的 Lua 脚本不同。这样，我们只使用 Lua 来进行开发就足够了。

skynet 提供了一个叫做 skynet 的 lua 模块提供给 snlua 服务承载的 Lua 脚本使用。你只需要编写一个后缀为 .lua 的脚本文件，把文件名作为启动参数，启动 snlua 即可。（关于脚本路径的配置，见 Config）

通常，你需要在脚本的第一行写上：

```
local skynet = require "skynet"
```

注：skynet 这个模块不能在 skynet 框架之外使用，所以你用标准的 lua 解析器运行包含了 skynet 模块的代码会立即出错。这是因为，每个 skynet 服务都依赖一个 `skynet_context` 的 C 对象，它是由 snlua 导入到 lua 虚拟机中的。

每个 skynet 服务，最重要的职责就是处理别的服务发送过来的消息，以及向别的服务发送消息。每条 skynet 消息由五个元素构成。

1.

session：大部分消息工作在请求回应模式下。即，一个服务向另一个服务发起一个请求，而后收到请求的服务在处理完请求消息后，回复一条消息。session 是由发起请求的服务生成的，对它自己唯一的消息标识。回

应方在回应时，将 session 带回。这样发送方才能识别出哪条消息是针对哪条的回应。session 是一个非负整数，当一条消息不需要回应时，按惯例，使用 0 这个特殊的 session 号。session 由 skynet 框架生成管理，通常不需要使用者关心。

2.
3.

source：消息源。每个服务都由一个 32bit 整数标识。这个整数可以看成是服务在 skynet 系统中的地址。即使在服务退出后，新启动的服务通常也不会使用已用过的地址（除非发生回绕，但一般间隔时间非常长）。每条收到的消息都携带有 source ，方便在回应的时候可以指定地址。但地址的管理通常由框架完成，用户不用关心。

4.
5.

type：消息类别。每个服务可以接收 256 种不同类别的消息。每种类别可以有不同的消息编码格式。有十几种类别是框架保留的，通常也不建议用户定义新的消息类别。因为用户完全可以利用已有的类别，而用具体的消息内容来区分每条具体的含义。框架把这些 type 映射为字符串便于记忆。最常用的消息类别名为 "lua" 广泛用于用 lua 编写的 skynet 服务间的通讯。

6.
7.

message：消息的 C 指针，在 Lua 层看来是一个 lightuserdata 。框架会隐藏这个细节，最终用户处理的是经过解码过的 lua 对象。只有极少情况，你才需要在 lua 层直接操作这个指针。

8.
9.

size ： 消息的长度。通常和 **message** 一起结合起来使用。

10.

另外，有部分 API 只在搭建框架时用到，普通服务并不会使用。所以这些 API 被抽离出来放在 skynet.manager 模块中。需要使用时，需要先调用：

```
require "skynet.manager"
```

为了兼容老代码，skynet.manager 共享 skynet 名字空间。require 这个模块后，这些额外的 API 依旧放在 skynet 下。而 `require "skynet.manager"` 的返回值同样是 skynet 名字空间。

这些额外的 API 列在下方，具体的解释放在后续的文字中：

- `skynet.launch` 启动一个 C 服务。
- `skynet.kill` 强行杀掉一个服务。
- `skynet.abort` 退出 skynet 进程。
- `skynet.register` 给自身注册一个名字。
- `skynet.name` 为一个服务命名。
- `skynet.forward_type` 将本服务实现为消息转发器，对一类消息进行转发。
- `skynet.filter` 过滤消息再处理。（注：filter 可以将 type, msg, sz, session, source 五个参数先处理过再返回新的 5 个参数。）
- `skynet.monitor` 给当前 skynet 进程设置一个全局的服务监控。

服务地址

每个服务都有一个 32bit 的数字地址，这个地址的高 8bit 表明了它所属的节点。

`skynet.self()` 用于获得服务自己的地址。

`skynet.harbor()` 用于获得服务所属的节点。

`skynet.address(address)` 用于把一个地址数字转换为一个可用于阅读的字符串。

同时，我们还可以给地址起一个名字方便使用。

`skynet.register(name)` 可以为自己注册一个别名。（别名必须在 16 个字符以内）

`skynet.name(name, address)` 为一个地址命名。`skynet.name(name, skynet.self())` 和 `skynet.register(name)` 功能等价。

这个名字一旦注册，是在 skynet 系统中通用的，你需要自己约定名字的管理的方法。

以 . 开头的名字是在同一 skynet 节点下有效的，跨节点的 skynet 服务对别的节点下的 . 开头的名字不可见。不同的 skynet 节点可以定义相同的 . 开头的名字。

以字母开头的名字在整个 skynet 网络中都有效，你可以通过这种全局名字把消息发到其它节点的。原则上，不鼓励滥用全局名字，它有一定的管理成本。管用的方法是在业务层交换服务的数字地址，让服务自行记住其它服务的地址来传播消息。

`skynet.localname(name)` 用来查询一个 . 开头的名字对应的地址。它是一个非阻塞 API ，不可以查询跨节点的全局名字。

下面的 API 说明中，如非特别提及，所有接受服务地址的参数，都可以传入这个地址的字符串别名。

消息分发和回应

`skynet.dispatch(type, function(session, source, ...) ... end)` 注册特定类消息的处理函数。大多数程序会注册 lua 类消息的处理函数，惯例的写法是：

```
local CMD = {}



skynet.dispatch("lua", function(session, source, cmd, ...)
  local f = assert(CMD[cmd])
  f(...)end)
```

这段代码注册了 lua 类消息的分发函数。通常约定 lua 类消息的第一个元素是一个字符串，表示具体消息对应的操作。我们会在脚本中创建一个 CMD 表，把对应的操作函数定义在表中。每条 lua 消息抵达后，从 CMD 表中查到处理函数，并把余下的参数传入。这个消息的 session 和 source 可以不必传递给处理函数，因为除了主动向 source 发送类别为 "response" 的消息来回应它以外，还有更简单的方法。框架记忆了这两个值。

这仅仅是一个惯用法，你也可以用其它方法来处理消息。skynet 并未规定你必须怎样做。

虽然并不推荐，但你还可以注册新的消息类别，方法是使用 `skynet.register_protocol`。例如你可以注册一个以文本方式编码消息的消息类别。通常用 C 编写的服务更容易解析文本消息。skynet 已经定义了这种消息类别为 `skynet.PTYPE_TEXT`，但默认并没有注册到 lua 中使用。

```
skynet.register_protocol {
  name = "text",
  id = skynet.PTYPE_TEXT,
```

```
  pack = function(m) return tostring(m) end,

  unpack = skynet.tostring,

}
```

新的类别必须提供 pack 和 unpack 函数，用于消息的编码和解码。

pack 函数必须返回一个 string 或是一个 userdata 和 size 。在 Lua 脚本中，推荐你返回 string 类型，而用后一种形式需要对 skynet 底层有足够的了解（采用它多半是因为性能考虑，可以减少一些数据拷贝）。

unpack 函数接收一个 lightuserdata 和一个整数 。即上面提到的 message 和 size 。lua 无法直接处理 C 指针，所以必须使用额外的 C 库导入函数来解码。skynet.tostring 就是这样的一个函数，它将这个 C 指针和长度翻译成 lua 的 string 。

接下来你可以使用 skynet.dispatch 注册 text 类别的处理方法了。当然，直接在 `skynet.register_protocol` 时传入 dispatch 函数也可以。

dispatch 函数会在收到每条类别对应的消息时被回调。消息先经过 unpack 函数，返回值被传入 dispatch 。每条消息的处理都工作在一个独立的 coroutine 中，看起来以多线程方式工作。但记住，在同一个 lua 虚拟机（同一个 lua 服务）中，永远不可能出现多线程并发的情况。**你的 lua 脚本不需要考虑线程安全的问题，但每次有阻塞 api 调用时，脚本都可能发生重入，这点务必小心。** CriticalSection 模块可以帮助你减少并发带来的复杂性。

回应一个消息可以使用 `skynet.ret(message, size)` 。它会将 message size 对应的消息附上当前消息的 session ， 以及 `skynet.PTYPE_RESPONSE` 这个类别，发送

给当前消息的来源 source 。由于某些历史原因（早期的 skynet 默认消息类别是文本，而没有经过特殊编码），这个 API 被设计成传递一个 C 指针和长度，而不是经过当前消息的 pack 函数打包。或者你也可以省略 size 而传入一个字符串。

由于 skynet 中最常用的消息类别是 lua ，这种消息是经过 `skynet.pack` 打包的，所以惯用法是 `skynet.ret(skynet.pack(...))` 。btw，`skynet.pack(...)` 返回一个 lightuserdata 和一个长度，符合 skynet.ret 的参数需求；与之对应的是 `skynet.unpack(message, size)` 它可以把一个 C 指针加长度的消息解码成一组 Lua 对象。

skynet.ret 在同一个消息处理的 coroutine 中只可以被调用一次，多次调用会触发异常。有时候，你需要挂起一个请求，等将来时机满足，再回应它。而回应的时候已经在别的 coroutine 中了。针对这种情况，你可以调用 `skynet.response(skynet.pack)` 获得一个闭包，以后调用这个闭包即可把回应消息发回。这里的参数 skynet.pack 是可选的，你可以传入其它打包函数，默认即是 skynet.pack 。

skynet.response 返回的闭包可用于延迟回应。调用它时，第一个参数通常是 true 表示是一个正常的回应，之后的参数是需要回应的数据。如果是 false ，则给请求者抛出一个异常。它的返回值表示回应的地址是否还有效。如果你仅仅想知道回应地址的有效性，那么可以在第一个参数传入 "TEST" 用于检测。

注：skynet.ret 和 skynet.response 都是非阻塞 API 。

如果你没有回应（ret 或 response）一个外部请求，session 不为 0 时，skynet 会写一条 log 提醒你这里可能有问题。你可以调用 skynet.ignoreret() 告诉框

架你打算忽略这个 session 。这通常在你想利用 session 传递其它数据时（即，不用 skynet.call 调用）使用。比如，你可以将客户端的 socket fd 当成 session ，把外部消息直接转发给内部服务处理。

关于消息数据指针

skynet 服务间传递的消息在底层是用 C 指针/lightuserdata 加一个数字长度来表示的。当一条消息进入 skynet 服务时，该消息会根据消息类别分发到对应的类别处理流程，（由 skynet.register_protocol ）。这个消息数据指针是由发送消息方生成的，通常是由 skynet_malloc 分配的内存块。默认情况下，框架会在之后调用 skynet_free 释放这个指针。

如果你想阻止框架调用 skynet_free 可以使用 skynet.forward_type 取代 skynet.start 调用。和 skynet.start 不同，skynet_forwardtype 需要多传递一张表，表示哪些类的消息不需要框架调用 skynet_free 。例如：

```
skynet.forward_type( { [skynet.PTYPE_LUA] = skynet.PTYPE_USER }, start_func )
```

表示 PTYPE_LUA 类的消息处理完毕后，不要调用 skynet_free 释放消息数据指针。这通常用于做消息转发。

这里由于框架默认定义了 PTYPE_LUA 的处理流程，

而 skynet.register_protocol 不准重定义这个流程，所以我们可以重定向消息类型为 PTYPE_USER 。

还有另一种情况也需要用 skynet.forward_type 阻止释放消息数据指针：如果针对某种特别的消息，传了一个复杂对象（而不是由 skynet_malloc 分配出来的整块内存；那么就可以让框架忽略数据指针，而自己调用对象的释放函数去释放这个指针。

消息的序列化

在上一节里我们提到，每类消息都应该定义该类型的打包和解包函数。

当我们能确保消息仅在同一进程间流通的时候，便可以直接把 C 对象编码成一个指针。因为进程相同，所以 C 指针可以有效传递。但是，skynet 默认支持有多节点模式，消息有可能被传到另一台机器的另一个进程中。这种情况下，每条消息都必须是一块连续内存，我们就必须对消息进行序列化操作。

skynet 默认提供了一套对 lua 数据结构的序列化方案。即上一节提到的 `skynet.pack` 以及 `skynet.unpack` 函数。`skynet.pack` 可以将一组 lua 对象序列化为一个由 malloc 分配出来的 C 指针加一个数字长度。你需要考虑 C 指针引用的数据块何时释放的问题。当然，如果你只是将 `skynet.pack` 填在消息处理框架里时，框架解决了这个管理问题。skynet 将 C 指针发送到其他服务，而接收方会在使用完后释放这个指针。

如果你想把这个序列化模块做它用，建议使用另一个 api `skynet.packstring` 。和 `skynet.pack` 不同，它返回一个 lua string 。而 `skynet.unpack` 即可以处理 C 指针，也可以处理 lua string 。

这个序列化库支持 string, boolean, number, lightuserdata, table 这些类型，但对 lua table 的 metatable 支持非常有限，所以尽量不要用其打包带有元方法的 lua 对象。

消息推送和远程调用

有了处理别的服务发送过来的请求的能力，势必就该有向其他服务发送消息或请求的能力。

`skynet.send(address, typename, ...)` 这条 API 可以把一条类别为 typename 的消息发送给 address 。它会先经过事先注册的 pack 函数打包 ... 的内容。

`skynet.send` 是一条非阻塞 API ，发送完消息后，coroutine 会继续向下运行，这期间服务不会重入。

`skynet.call(address, typename, ...)` 这条 API 则不同，它会在内部生成一个唯一 session ，并向 address 提起请求，并阻塞等待对 session 的回应（可以不由 address 回应）。当消息回应后，还会通过之前注册的 unpack 函数解包。表面上看起来，就是发起了一次 RPC ，并阻塞等待回应。call 不支持超时，如果有超时的需求，可以参考 TimeOutCall 或 这篇 blog 。

尤其需要留意的是，`skynet.call` 仅仅阻塞住当前的 coroutine ，而没有阻塞整个服务。在等待回应期间，服务照样可以响应其他请求。所以，尤其要注意，**在 `skynet.call` 之前获得的服务内的状态，到返回后，很有可能改变**。

还有三个 API 与之相关，但并非常规开发所需要：

`skynet.redirect(address, source, typename, session, ...)` 它和 `skynet.send` 功能类似，但更细节一些。它可以指定发送地址（把消息源伪装成另一个服务），指定发送的消息的 session 。注：address 和 source 都必须是数字地址，不可以是别名。`skynet.redirect` 不会调用 pack ，所以这里的 ... 必须是一个编码过的字符串，或是 userdata 加一个长度。

`skynet.genid()` 生成一个唯一 session 号。

`skynet.rawcall(address, typename, message, size)` 它和 `skynet.call` 功能类似（也是阻塞 API）。但发送时不经过 pack 打包流程，收到回应后，也不走 unpack 流程。

服务的启动和退出

每个 skynet 服务都必须有一个启动函数。这一点和普通 Lua 脚本不同，传统的 Lua 脚本是没有专门的主函数，脚本本身即是主函数。而 skynet 服务，你必须主动调用 `skynet.start(function() ... end)` 。

`skynet.start` 注册一个函数为这个服务的启动函数。当然你还是可以在脚本中随意写一段 Lua 代码，它们会先于 start 函数执行。但是，不要在外面调用 skynet 的阻塞 API ，因为框架将无法唤醒它们。

如果你想在 `skynet.start` 注册的函数之前做点什么，可以调用 `skynet.init(function() ... end)` 。这通常用于 lua 库的编写。你需要编写的服务引用你的库的时候，事先调用一些 skynet 阻塞 API ，就可以用 `skynet.init` 把这些工作注册在 start 之前。

`skynet.exit()` 用于退出当前的服务。skynet.exit 之后的代码都不会被运行。而且，当前服务被阻塞住的 coroutine 也会立刻中断退出。这些通常是一些 RPC 尚未收到回应。所以调用 `skynet.exit()` 请务必小心。

`skynet.kill(address)` 可以用来强制关闭别的服务。但强烈不推荐这样做。因为对象会在任意一条消息处理完毕后，毫无征兆的退出。所以推荐的做法是，发送

一条消息，让对方自己善后以及调用 `skynet.exit` 。注：`skynet.kill(skynet.self())` 不完全等价于 `skynet.exit()` ，后者更安全。

`skynet.newservice(name, ...)` 用于启动一个新的 Lua 服务。name 是脚本的名字（不用写 .lua 后缀）。只有被启动的脚本的 start 函数返回后，这个 API 才会返回启动的服务的地址，这是一个阻塞 API 。如果被启动的脚本在初始化环节抛出异常，或在初始化完成前就调用 `skynet.exit` 退出，`skynet.newservice` 都会抛出异常。如果被启动的脚本的 start 函数是一个永不结束的循环，那么 newservice 也会被永远阻塞住。

注意：启动参数其实是以字符串拼接的方式传递过去的。所以不要在参数中传递复杂的 Lua 对象。接收到的参数都是字符串，且字符串中不可以有空格（否则会被分割成多个参数）。这种参数传递方式是历史遗留下来的，有很多潜在的问题。目前推荐的惯例是，让你的服务响应一个启动消息。在 newservice 之后，立刻调用 `skynet.call` 发送启动请求。

`skynet.launch(servicename, ...)` 用于启动一个 C 模块的服务。由于 skynet 主要用 lua 编写服务，所以它用的并不多。

注意：同一段 lua 脚本可以作为一个 lua 服务启动多次，同一个 C 模块也可以作为 C 服务启动多次。服务的地址是区分运行时不同服务的唯一标识。如果你想编写一个服务，在系统中只存在一份，可以参考 UniqueService 。

如果你对 skynet 的原理非常熟悉，当你对单个简单服务的性能有极端要求的时候，可以参考 TinyService 。

时钟和线程

skynet 的内部时钟精度为 1/100 秒。。

`skynet.now()` 将返回 skynet 节点进程内部时间。这个返回值的数值不是真实时间, 本身意义不大。 不同节点在同一时刻取到的值也不相同。只有两次调用的差值才有意义。用来测量经过的时间, 单位是 1/100 秒。 **(注意:这里的时间片表示小于 skynet 内部时钟周期的时间片,假如执行了比较费时的操作如超长时间的循环,或者调用了外部的阻塞调用,如 `os.execute('sleep 1')`, 即使中间没有 skynet 的阻塞 api 调用,两次调用的返回值还是会不同的.)**

`skynet.hpc()` 如果你需要做性能分析，可以使用 skynet.hpc ，它会返回精度为 ns （1000000000 分之一 秒）的 64 位整数。

`skynet.starttime()` 返回 skynet 节点进程启动的 UTC 时间，以秒为单位。

`skynet.time()` 返回以秒为单位（精度为小数点后两位）的 UTC 时间。它时间上等价于: `skynet.now()/100 + skynet.starttime()`

`skynet.sleep(ti)` 将当前 coroutine 挂起 ti 个单位时间。一个单位是 1/100 秒。它是向框架注册一个定时器实现的。框架会在 ti 时间后，发送一个定时器消息来唤醒这个 coroutine 。这是一个阻塞 API 。它的返回值会告诉你是时间到了，还是被 `skynet.wakeup` 唤醒 （返回 "BREAK"）。

`skynet.yield()` 相当于 `skynet.sleep(0)` 。交出当前服务对 CPU 的控制权。通常在你想做大量的操作，又没有机会调用阻塞 API 时，可以选择调用 yield 让系统跑的更平滑。

`skynet.timeout(ti, func)` 让框架在 ti 个单位时间后，调用 func 这个函数。这不是一个阻塞 API ，当前 coroutine 会继续向下运行，而 func 将来会在新的 coroutine 中执行。

skynet 的定时器实现的非常高效，所以一般不用太担心性能问题。不过，如果你的服务想大量使用定时器的话，可以考虑一个更好的方法：即在一个 service 里，尽量只使用一个 `skynet.timeout` ，用它来触发自己的定时事件模块。这样可以减少大量从框架发送到服务的消息数量。毕竟一个服务在同一个单位时间能处理的外部消息数量是有限的。

timeout 没有取消接口，这是因为你可以简单的封装它获得取消的能力：

```
function cancelable_timeout(ti, func)
  local function cb()
    if func then
      func()
    end
  end
  local function cancel()
    func = nil
  end
  skynet.timeout(ti, cb)
  return cancelend
local cancel = cancelable_timeout(ti, dosomething)
cancel()  -- canel dosomething
```

`skynet.fork(func, ...)` 从功能上，它等价于 `skynet.timeout(0, function() func(...) end)` 但是比 timeout 高效一点。因为它并不需要向框架注册一个定时器。

`skynet.wait(token)` 把当前 coroutine 挂起，之后由 `skynet.wakeup` 唤醒。token 必须是唯一的，默认为 `coroutine.running()` 。

`skynet.wakeup(token)` 唤醒一个被 `skynet.sleep` 或 `skynet.wait` 挂起的 coroutine 。在 1.0 版中 wakeup 不保证次序，目前的版本则可以保证。

日志跟踪

`skynet.error(log)` 可以用来写文本日志。日志输出的目的地请参考 Config 。

`skynet.trace()` 在一个消息处理流程中，如果调用了这个 api ，将开启消息跟踪日志。每次调用都会生成一个唯一 tag ，所有当前执行流，和调用到的其它服务，都会计入日志中。具体解释，可以参考 https://blog.codingnow.com/2018/05/skynet_trace.html