snax 框架的简介

snax 仅仅解决一个简单问题：一个 skynet 内部服务，处理发送给它的消息。

使用 snax 服务先要在 Config 中配置 snax 用于路径查找。每个 snax 服务都有一个用于启动服务的名字，推荐按 lua 的模块命名规则，但目前不推荐在服务名中包含"点"（在路径搜索上尚未支持 . 与 / 的替换）。在启动服务时会按查找路径搜索对应的文件。

snax 服务用 lua 编写，但并不是一个独立的 lua 程序。它里面包含了一组 lua 函数，会被 snax 框架分析加载。

test/pingserver.lua 就是一个简单的 snax 服务范例：

```
local skynet = require "skynet"
local i = 0 local hello = "hello"
function response.ping(hello)
        skynet.sleep(100)
        return hello end
function accept.hello()
        i = i + 1
        print (i, hello) end
function response.error()
        error   "throw an error" end
function init( ... )
        print ("ping server start:", ...) end
```

```
function exit(...)
        print ("ping server exit:", ...) end
```

snax 服务的启动和退出

每个 snax 服务中都需要定义一个 init 函数，启动这个服务会调用这个函数，并把启动参数传给它。

snax 服务还可以定义一个 **exit** 函数用于响应服务退出的事件，同样能接收一些参数。

和标准的 skynet 服务不同，这些参数的类型不受限制，可以是 lua 的复杂数据类型。（而 skynet 服务受底层限制，只可以接受字符串参数）

启动一个 snax 服务有三种方式：

```
local snax = require "snax"
```

1、**snax.newservice(name, ...)**：可以把一个服务启动多份。传入服务名和参数，它会返回一个对象，用于和这个启动的服务交互。如果多次调用 newservice ，即使名字相同，也会生成多份服务的实例，它们各自独立，由不同的对象区分。

2、**snax.uniqueservice(name, ...)**：和上面 api 类似，但在一个节点上只会启动一份同名服务。如果你多次调用它，会返回相同的对象。

3、**snax.globalservice(name, ...)**：和上面的 api 类似，但在整个 skynet 网络中（如果你启动了多个节点），只会有一个同名服务。

1.

前一种方式可以看成是启动了一个匿名服务，启动后只能用地址（以及对服务地址的对象封装）与之通讯；后两种方式都可以看成是具名服务，之后可以用名字找到它。

后两种方式是对具名服务惰性初始化。如果你在代码中写了多处服务启动，第一次会生效，后续只是对前面启动的服务的查询。往往我们希望明确服务的启动流程（在启动脚本里就把它们启动好）；尤其是全局（整个 skynet 网络可见）的服务，我们还希望明确它启动在哪个结点上（如果是惰性启动，你可能无法预知哪个节点先把这个服务启动起来的）。这时，可以使用下面两个 api ：

1、snax.queryservice(name)：查询当前节点的具名服务，返回一个服务对象。如果服务尚未启动，那么一直阻塞等待它启动完毕。

2、snax.queryglobal(name)：查询一个全局名字的服务，返回一个服务对象。如果服务尚未启动，那么一直阻塞等待它启动完毕。

对于匿名服务，你无法在别处通过名字得到和它交互的对象。如果你有这个需求，可以把对象的 .handle 域通过消息发送给别人。 handle 是一个数字，即 snax 服务的 skynet 服务地址。

这个数字的**接收方可以通过 snax.bind(handle, typename) 把它转换成服务对象。这里第二个参数需要传入服务的启动名，以用来了解这个服务有哪些远程方法可以供调用。当然，你也可以直接把 .type 域和 .handle 一起发送过去，而不必在源代码上约定。**

如果你想让一个 snax 服务退出，调用 snax.kill(obj, ...) 即可。

snax.self() 用来获取自己这个服务对象，它等价于 snax.bind(skynet.self(), SERVER_NAME)

snax.exit(...) 退出当前服务，它等价于 snax.kill(snax.self(), ...) 。

注：**snax 的接口约定是通过分析对应的 lua 源文件完成的**。所以无论是消息发送方还是消息接收方都必须可以读到其消息处理的源文件。不仅 snax.newservice 会加载对应的源文件生成新的服务；调用者本身也会在自身的 lua 虚拟机内加载对应的 lua 文件做一次分析；同样， snax.bind 也会按 typename 分析对应的源文件。

RPC 调用

snax 服务中可以定义一组函数用于响应其它服务提出的请求，并给出（或不给出）回应。一个非 snax 服务也可以调用 snax 服务的远程方法。

要定义这类远程方法，可以通过定义 function response.foobar(...) 来声明一个远程方法。**foobar 是方法名，response 前缀表示这个方法一定有一个回应**。你可以通过函数返回值来回应远程调用。

调用这个远程方法，可**以通过 obj.req.foobar(...) 来调用它。obj 是服务对象，req 表示这个调用需要接收返回值。**foobar 是方法的名字。其实，在创建出 obj 对象时，框架已经了解了这个 foobar 方法有返回值，这里多此一举让使用者明确 req 类型是为了让调用者（以及潜在的代码维护者）清楚这次调用是阻塞的，会导致服务挂起，等待对方的回应。

如果你设计的协议不需要返回值，那么可以通过定义 function accept.foobar(...) 来声明。这里可以有和 response 组相同的方法名。通过 **accept** 前缀来区分 **response** 组下同名的方法。这类方法的实现函数不可以有返回值。

调用这类不需要返回值的远程方法，应该使用 obj.post.foobar(...) 。这个调用不会阻塞。所以你也不用担心这里服务会挂起，因为别的消息进入改变了服务的内部状态。

注: post 不适用于 Cluster 模式，只能用于本地消息或 master/slave 结构下的消息。

热更新

snax 是支持热更新的（只能热更新 snax 框架编写的 lua 服务）。但**热更新更多的用途是做不停机的 bug 修复，不应用于常规的版本更新。**所以，热更新的 api 被设计成下面这个样子。更适合打补丁。

你可以通过 **snax.hotfix(obj, patchcode) 来向 obj 提交一个 patch 。**

举个例子，你可以向上面列出的 pingserver 提交一个 patch ：

```
snax.hotfix(obj, [[
local i
local hello
function accept.hello()
        i = i + 1
        print ("fix", i, hello)
```

```
end

function hotfix(...)

        local temp = i

        i = 100

        return temp

end]])
```

这样便**改写了 accept.hello 方法。在 patch 中声明的 local i 和 local hello 在之前的版本中也有同名的 local 变量。 snax 的热更新机制会重新映射这些 local 变量。让 patch 中的新函数对应已有的 local 变量，所以你可以安全的继承服务的内部状态。**

patch 中可以包含一个 function hotfix(...) 函数，在 patch 提交后立刻执行。这个函数可以用来查看或修改 snax 服务的线上状态（因为 local 变量会被关联）。hotfix 的函数返回值会传递给 snax.hotfix 的调用者。

所以，你也可以提交一个仅包含 hotfix 函数的 patch ，而不修改任何代码。这样的 patch 通常用于查看 snax 服务的内部状态（内部 local 变量的值），或用于修改它们。

注：不可以在 patch 中增加新的远程方法。

---

test/testping.lua 是对 pingserver 的调用范例，你可以查看它加深理解。

snax 的实现分两部分：让服务工作起来的框架，实现在 service/snaxd.lua ；snax 的 api ，在 lualib/snax.lua 以及 lualib/snax/* 中。有兴趣的同学可以阅读源码。