MsgServer

设计

snax.msgserver 是一个基于**消息请求和回应模式**的网关服务器模板。它基于 snax.gateserver 定制，可以接收客户端发起的请求数据包，并给出对应的回应。

和 service/gate.lua 不同，用户在使用它的时候，一个**用户的业务处理不基于连接**。即，它不把连接建立作为用户登陆、不在连接断开时让用户登出。用户必须显式的登出系统，或是业务逻辑设计的超时机制导致登出。

和传统的 HTTP 服务不同。在同一个连接上，用户可以发起任意多个请求，并接收多次回应。请求和回应次序不必保证一致，所以内部用 session 来关联请求和回应。回应不保证和请求次序相同，也方便在同一个 TCP 连接上发起一些服务器需要很长时间的请求，而不会阻塞后续的快速响应。**即请求和回应基于 session 来区别消息类别。**

在请求回应模式下，服务器无法单向向客户端推送消息。但可以很容易模拟这种业务。你只需要在登入系统后，立刻发起一个应用层协议要求服务器返回给你最近需要推送给你的消息；如果服务器暂时没有消息推送给客户端，那么只需要简单的挂起这个 session ，待有消息推送时，服务器通过这个 session 返回即可。

和一般的长连接服务也不同。**当客户端和服务器失去连接后，只需要简单的重新接入，就可以继续工作**。在 snax.msgserver 的实现中，我们要求客户端和服务器同时只保证一条有效的通讯连接，但你可以轻易扩展成多条（但意义不大，因为 session 可以保证慢请求不会阻塞快请求的回应）并发。

架构

消息服务器 snax.msgserver (M) 必须和 LoginServer (L) 一起使用。用户 C 的业务流程通常是这样的：

1. C 向 L 表明想登陆 M。
2. L 验证 C 的身份，交换 secret ，并转告 M 说 C 想登陆。
3. M 确认 C 的 secret ，做好登陆准备，生成一个 subid (subid 用于多重登陆）。
4. L 构造当次登陆用的用户名（用 uid 和 subid 联合生成）返回给 C 向 C 确认可以登陆。
5. C 以这个用户名加上一个自增序列号（用于断线重连），利用 secret 签名和 M 握手接入。
6. M 检查签名是否正确，以及序列号是否使用过（一次有效）。然后等待 M 的请求，并在当前连接上回应。

如果 C 和 M 断开连接，并不意味着用户从系统登出，这时 C 可以自增序列号，重新和 M 握手。这称为修复连接，C 不需要重复进行和 L 的登陆认证流程。

在前次连接上，C 向 M 发送的请求，M 会缓存一部分。如果连接修复后，C 向 M 发起过去用过的 session 号的请求，M 将之前缓存的回应包直接发回，而不去重新做一次业务处理。但是，在同一连接上发起的相同的 session ，后一次会覆盖前一次，请求都会交给业务层处理。业务层收到相同的 session ，可以认为是前一次的 session 的回应已经被客户端收到，所以复用了这个 session ；因为在同一条 TCP 连接上，不会发生数据丢失的情况；一旦有回应包没有发到，只可能是连接断开，新的连接可以重新发起相同的请求。

注：这里 cache 最近的一部分请求的回应可以满足大部分需求。同时也可以约定 session 的自增性，发现请求过去的 session 而没有缓存时，强制将用户登出。

C 也可以重新去 L 处交换新的用户名和 serect 。这种情况下，L 会通知 M 将登陆状态中的 C 登出，更新新版的 serect ，重置所有之前的请求 session 和回应缓存。最后让 C 重新登陆。**即 C 可以重新去 L 出交换新的用户名和 secret，重置所有之前的请求 session 和回应缓存，C 重新登录。**

使用

和 GateServer 和 LoginServer 一样，**snax.msgserver 只是一个模板，你还需要自定义一些业务相关的代码，才是一个完整的服务。**

```
local msgserver = require "snax.msgserver"
```

你需要注册这样一些 handler 来处理一些事件：

```
function server.login_handler(uid, secret)
```

当一个用户登陆后，登陆服务器会转交给你这个用户的 uid 和 serect ，最终会触发 login_handler 方法。在这个函数里，你需要做的是判定这个用户是否真的可以登陆。（一般是可以的，如果想阻止用户多重登陆，在登陆服务器里就完成了）

然后为用户生成一个 subid ，如果你的用户同时只允许一个实例存在于系统中，你可以生成固定的 subid （但建议还是生成不同的）。

msgserver.username(uid, subid, servername) 可以得到这个用户这次的登陆名。这里 servername 是当前登陆点的名字。

接着你应该做好用户进入的准备。常规做法是启动一个 agent 服务，然后命令它从数据库加载这个用户的数据。如果启动 agent 需要消耗大量的 CPU 时间，你也可以预先启动好多份 agent 放在一个池中，这里只需要简单的取出一个可用的空 agent 即可。

当一切准备好后，把 subid 返回。

在这个过程中，如果你发现一些意外情况，不希望用户进入，只需要用 error 抛出异常。

```
function server.logout_handler(uid, subid)
```

当一个用户想登出时，这个函数会被调用，你可以在里面做一些状态清除的工作。这个事件通常是由 agent 的消息触发。

```
function server.kick_handler(uid, subid)
```

当外界（通常是登陆服务器）希望让一个用户登出时，会触发这个事件。通常你需要在里面通知 agent 将用户数据写数据库，并且让它在善后工作完成后，发起一个 logout 消息（最终会触发 logout_handler）

```
function server.disconnect_handler(username)
```

当用户的通讯连接断开后，会触发这个事件。你可以不关心这个事件，也可以利用这个事件做超时管理。（比如断开连接后一定时间不重新连回来就主动登出。

```
function server.request_handler(username, msg, sz)
```

如果用户提起了一个请求，就会被这个 request_handler 捕获到。这里隐藏了 session 信息，因为你可以在这个函数中调用 skynet.call 等 RPC 调用，但一般的做法是简单的把 msg,sz 转发给 agent 即可。这里使用的是 **client 协议通道**，agent 只需要处理这个协议通道，具体的业务层通讯协议并无限制。

等请求处理完后，只需要返回一个字符串，这个字符串会回到框架，加上 session 回应客户端。这个函数中允许抛出异常，框架会正确的捕获这个异常，并通过协议通知客户端。

```
function server.register_handler(name)
```

因为 snax.msgserver 实际上是 snax.gateserver 的一个特殊实现。同样有**打开监听端口的指令，触发 register_handler name** 。在打开端口时，会触发这个 register_handler name 是在配置信息中配置的当前登陆点的名字，你需要把这个名字**注册到登陆服务器。登陆服务器转发消息**就可以按约定，在用户登陆你的时候把消息转发给你。

api

以下 api 可以在实现上述的 handler 时，在恰当的时机使用：

msgserver.userid(username)：uid, subid, server 把一个登陆名转换为 uid, subid, servername 三元组

msgserver.username(uid, subid, server)：username 把 uid, subid, servername 三元组构造成一个登陆名

msgserver.login(username, secret) 你需要在 login_handler 中调用它，注册一个登陆名对应的 serect

msgserver.logout(username) 让一个登陆名失效（登出），通常在 logout_handler 里调用。

msgserver.ip(username) 查询一个登陆名对应的连接的 ip 地址，如果没有关联的连接，会返回 nil 。

msgserver.start(conf) 启动一个 msgserver 。conf 是配置表。配置表和 GateServer 相同，但增加一项 servername ，你需要配置这个登陆点的名字。

范例

你可以在 examples/login/gated.lua 看到一个实现的范例，它可以和 examples/login/logind.lua 协同工作。

客户端的示范在 examples/login/client.lua 。

你可以先启动

```
./skynet examples/config.login
```

然后在同一台机器上启动

```
lua examples/login/client.lua
```

wire protocol

基础封包协议遵循的 GateServer，即**一个 2 字节的包头加内容**，以下说明的是内容的编码。

当一个连接接入后，第一个包是握手包。握手首先由客户端发起：

```
base64(uid)@base64(server)#base64(subid):index:base64(hmac)
```

index 至少是 1，每次连接都需要比之前的大。这样可以保证握手包不会被人恶意截获复用。 hmac 是在登陆环节获得的共识 serect，对前面一串数据做的挑战确认，具体算法可以见范例代码。

服务器会针对这次握手做一个回应，回应信息是一行文本：

- 404 User Not Found
- 403 Index Expired
- 401 Unauthorized
- 400 Bad Request
- 200 OK

如果握手成功，后续的包就是按照请求回应模式进行：

客户端发起请求：

bytes 任意内容 dword session

session 原则上以 Big-Endian 方式编码一个数字，但实际上 session 也可以看成是一个四字节的 token ，并无强制要求客户端如何生成和编码。

服务器回应：

bytes 内容(或异常信息） byte 标记。1 表示正常返回，0 表示异常返回。 dword session

session 用于匹配客户端提起的请求。