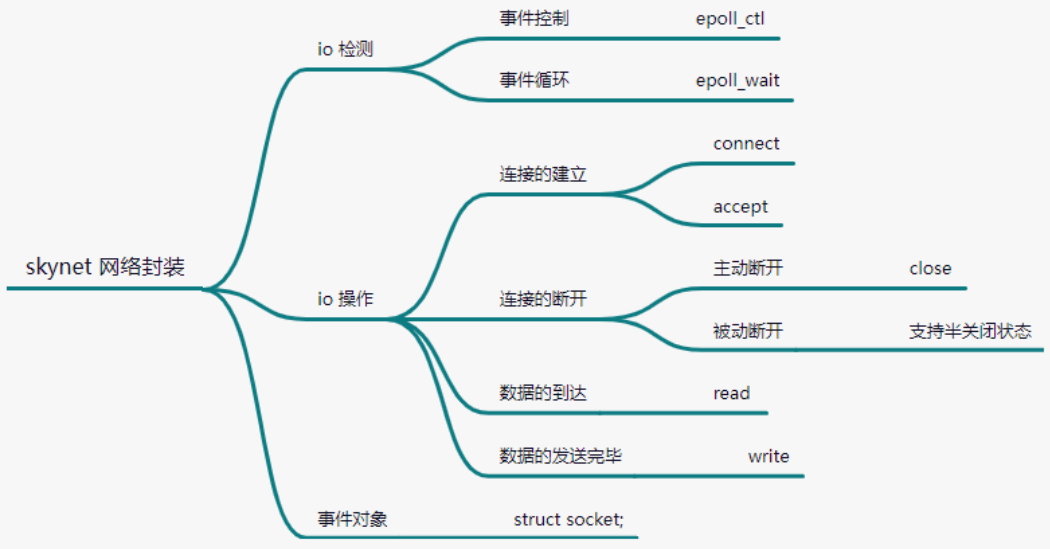
零声教育 Mark 老师 QQ: 2548898954

skynet 设计原理总结

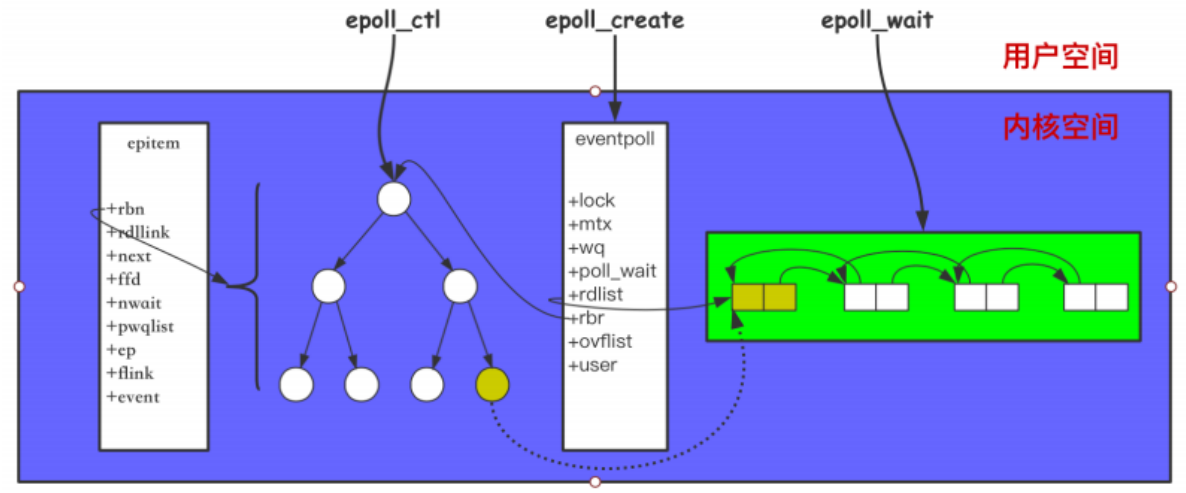
> 1. actor 内部若涉及多线程应考虑加自旋锁或原子操作；避免在工作线程执行过程中被切换；
> 2. actor 内部若涉及多线程应考虑临界区域操作不能过于耗时；避免长期占用工作线程让同消息队列中其他消息得不到及时执行；
> 3. actor 单个消息业务应避免阻塞线程（注意不是协程）的操作；如果这个操作是必不可少，另起一个外部进程，skynet 进程用 socket 与之通信；这种阻塞或者耗时操作的任务交由外部服务来处理；

skynet 网络封装层次



> 思考几个问题?
>
> 1. 事件与消息的关系?
> 2. 消息与actor如何绑定? 具体接口绑定流程?
> 3. 消息与协程的关系?

```
// 在 linux 系统中，采用 epoll 来检测管理网络事件；
int epoll_create(int size);
// 对红黑树进行增删改操作
int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int
timeout);
```



> 通过 `epoll_ctl` 设置 `struct epoll_event` 中 `data.ptr = (struct socket *)ud;` 来完成 fd 与 actor绑定;
>
> skynet 通过 `socket.start(fd, func)` 来完成 actor 与 fd 的绑定;

skynet 对连接半关闭状态的支持

读写端都关闭

```
// 事件：1. EPOLLHUP 读写段都关闭
e[i].eof = (flag & EPOLLHUP) != 0;
// 处理：直接关闭并向 actor 返回事件 SOCKET_CLOSE
int halfclose = halfclose_read(s);
force_close(ss, s, &l, result);
if (!halfclose) { // 如果前面因为关闭读端已经发送 SOCKET_CLOSE，在这里避免重复 SOCKET_CLOSE
    return SOCKET_CLOSE;
}
```

读端关闭

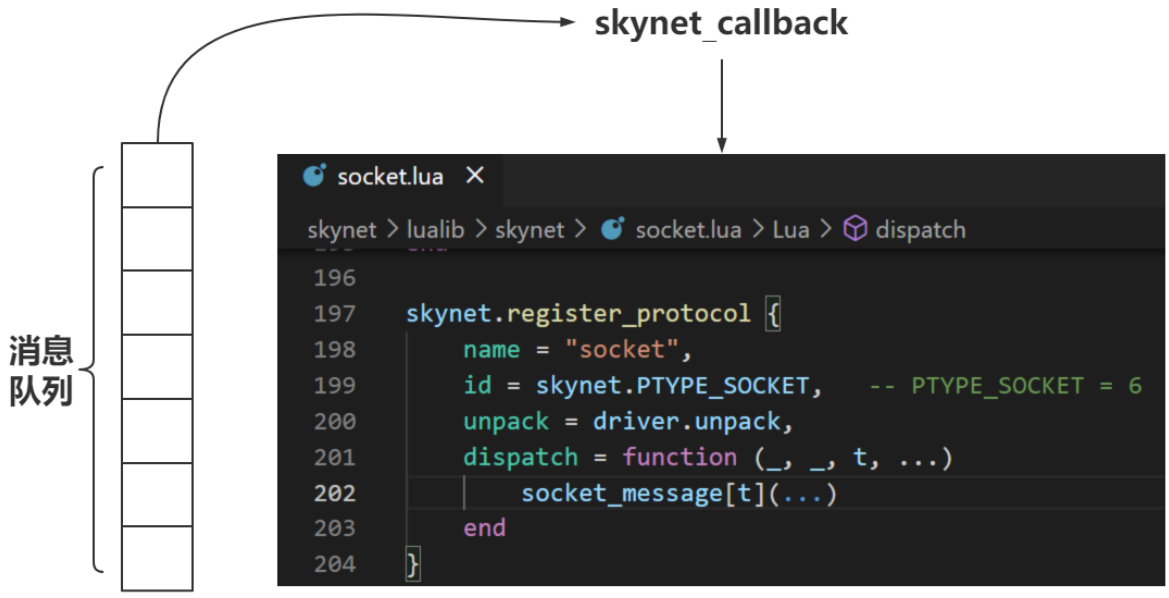
```
int n = (int)read(s->fd, buffer, sz);
// 事件：2. 读端关闭 注意：EPOLLRDHUP 也可以检测，但是这个 read = 0 更为及时；因为事件处理先处理读事件，再处理异常事件
if (n == 0) {
    if (s->closing) { // 如果该连接的 socket 已经关闭
        // Rare case : if s->closing is true, reading event is disable, and SOCKET_CLOSE is raised.
        if (nomore_sending_data(s)) {
            force_close(ss,s,l,result);
        }
        return -1;
    }
    int t = ATOM_LOAD(&s->type);
    if (t == SOCKET_TYPE_HALFCLOSE_READ) { // 如果已经处理过读端关闭
        // Rare case : Already shutdown read.
        return -1;
    }
    if (t == SOCKET_TYPE_HALFCLOSE_WRITE) { // 如果之前已经处理过写端关闭，则直接 close
        // Remote shutdown read (write error) before.
```

```
        force_close(ss,s,l,result);
    } else { // 如果之前没有处理过，则只处理读端关闭
        close_read(ss, s, result);
    }
    return SOCKET_CLOSE;
}
```

写端关闭

```
for (;;) {
    ssize_t sz = write(s->fd, tmp->ptr, tmp->sz);
    if (sz < 0) {
        switch(errno) {
            case EINTR:
                continue;
            case AGAIN_WOULDBLOCK:
                return -1;
        }
        // sz < 0 && errno = EPIPE,  fd is connected to a pipe or socket whose reading end is closed.
        // 在这里的处理是只要sz < 0，且不是被中断打断以及写缓冲满的情况下，直接关闭本地写端
        return close_write(ss, s, l, result);
    }
    ...
}
```

`socket.lua` 封装



> 加载 socket.lua 的服务，就具备了处理网络消息的能力，`skynet_callback` 设置服务的回调函数，在回调函数中可以设置不同类型的子回调；如图：在 socket.lua 设置了 `"socket"` 类型的处理方法；`pack` 指定发送数据的压缩方法，`unpack` 指定接收数据的解压缩的方法（`unpack` 的参数为 `msg, sz`），`dispatch` 指定处理消息的方法，它的参数来自于 `unpack` 的返回值；`socket_message` 处理不同类型的网络消息；此时需要重点关注，如何处理拆包和粘包问题？
>
> 首先需要有一个读缓冲区缓存网络发送过来的数据；在 lua-socket.c 中封装了 buffer 对象操作接口，用来缓存网络数据包，并且封装了 socket 的 io 操作接口；

```lua
-- SKYNET_SOCKET_TYPE_DATA = 1
socket_message[1] = function(id, size, data) ... end
-- SKYNET_SOCKET_TYPE_CONNECT = 2
socket_message[2] = function(id, _ , addr) ... end
-- SKYNET_SOCKET_TYPE_CLOSE = 3
socket_message[3] = function(id) ... end
-- SKYNET_SOCKET_TYPE_ACCEPT = 4
socket_message[4] = function(id, newid, addr) ... end
-- SKYNET_SOCKET_TYPE_ERROR = 5
socket_message[5] = function(id, _, err) ... end
-- SKYNET_SOCKET_TYPE_UDP = 6
socket_message[6] = function(id, size, data, address) ... end
-- SKYNET_SOCKET_TYPE_WARNING = 7
socket_message[7] = function(id, size) ... end
```

> 在 skynet 中，协程与网络消息的关系是有 slot 所存储的 socket 池的索引 id 来联系的；协程与 actor 消息以及定时消息的关系是由 session（全局唯一 id ）来联系的；

```lua
-- 连接第三方服务
local fd, err = socket.open(ip, port)
```

> 在 work 线程通过 pipe 发送消息到 socket 线程去调用 connect，并让出当前协程；socket 线程检测连接建立成功，唤醒被阻塞的协程；

```lua
local listenfd = socket.listen(host, port, backlog)
local accept_callback(id, addr)
    -- 这里可以绑定 actor 与网络消息的关系，同时会开启读事件；
    socket.start(id)
    ...
end
-- 注意：此时接收到的客户端连接，socket已经分配，但是没有开启读，需要在 accept_callback 回调当中进行绑定 actor 并开启读操作；
-- report_accept 中调用 newfd 的最后一个参数为 false，把读事件关闭了；
socket.start(listenfd, accept_callback)
```

> 在 work 线程通过 pipe 发送消息到 socket 线程去调用 bind、listen，并让出当前协程；

```lua
-- 假设现在解析网络数据包 header | body header存储的内容是body的长度 header 占用字节数为4
local sz = socket.header(socket.read(id, 4))
local bin = socket.read(id, sz)
-- 向id发送数据
socket.write(id, str)
```

应用

服务设置

> 服务的确定需要考虑以下几点：
>
> 1. 功能独立性，可独立测试；
> 2. 需要大致估计它的运算程度；如果密集计算，需要考虑拆分成多个服务；
> 3. 需要考虑lua gc 压力；通常服务中存放的对象数据越多，gc 压力越大；

服务拆分

> 如果一个服务涉及的功能太多，不能用简单案例来测试的时候，那么服务设置有问题，此时要按**功能拆分**；
>
> 如果某个服务功能检测，也可以用简单案例来测试，但是计算比较密集，此时要将该服务拆分成**核心数*N**个服务；
>
> - 设置核心数，是希望它们调度的时候能得到公平处理；
> - 为什么是N，根据 lua gc 压力，拆分成核心数的倍数；

服务数据共享

> 当服务拆分成多个，数据需要共享应该如何实现呢？
>
> 第一种，通过消息来交换数据，这也复合“通过消息来共享数据”这一哲学；
>
> 第二种，如果这些数据大部分场景下只是只读的，为了数据一致性，通常将数据放在一个服务中，让不同的服务从这个服务中获取数据；
>
> 第三种，就是使用 sharetable 模块；原理是 skynet 修改了 lua，为了让服务不重复加载数据，会共享一些函数原型和常量表，其次也默认会缓存 lua 文件，这样其他服务能快速加载这些 lua 文件（一个 lua 文件通过 `loadfile` 加载后，磁盘对该文件修改不会影响下一次加载）；
>
> skynet 增加了设置文件缓存方式的模块 `skynet.codecache`；

```
-- 从一个源文件读取一个共享表，这个文件需要返回一个 table ，这个 table 可以被多个不同的服务读取。... 是传给这个文件的参数。
sharetable.loadfile(filename, ...)
-- 更新一个或多个 key
sharetable.update(filenames)
-- 以 filename 为 key 查找一个被共享的 table
sharetable.query(filename)

local cache = require "skynet.codecache"
--[[
当 mode 为 "ON" 的时候，当前服务 cache 一切加载 lua 代码文件的行为。
当 mode 为 "OFF" 的时候，当前服务关闭任何重复利用 lua 代码文件的行为，即使在别的服务中曾经加载过同名文件。
当 mode 为 "EXIST" 的时候，当前服务在加载曾经在其它服务或自己的服务加载过同名文件时，复用之前的拷贝。但对新加载的文件则不进行 cache 。注：通常可以让 skynet 本身被 cache。
]]
cache.mode(mode)
```

网关设计

> 网关需要解决的问题：负责处理接收客户端的连接，并且验证后将连接分配到不同的服务中进行处理；
>
> 需要考虑的点：
>
> - 限制最大连接数；
> - 连接的验证；
> - 连接的处理；将连接移交到不同服务中去处理；

`gate-watchdog-agent`

> 官方提供的网关接入服务；gate 负责接收网络数据，watchdog 处理连接数据验证（注册和登录），验证成功后分配一个agent，接着 agent 通知 gate 接管数据处理；之后该连接的数据直接由 gate 流向 agent；

简单游戏实现

游戏介绍

> 目的：掌握 actor 模型开发思路；
>
> 游戏：猜数字的游戏；
>
> 条件：满3人开始游戏，游戏开始后不能退出，直到这个游戏结束；
>
> 规则：系统当中会随机 1-100 之间的数字，参与游戏的玩家依次猜测规定范围内的数字；如果猜测正确那么该玩家就输了，如果猜测错误，游戏继续；直到有玩家猜测成功，游戏结束，该玩家失败；

设计原则

> 简单可用，持续优化，而不是一开始就过度优化；

接口设计

> skynet中，从 actor 底层看是通过消息进行通信；从 actor 应用层看是通过 api 来进行通信；
>
> 接口隔离原则：不应该强迫客户依赖于他们不用的方法；从安全封装的角度出发，只暴露客户需要的接口；服务间不依赖彼此的实现；

- agent

  > `login`：实现登录功能；
  >
  > `ready`：准备，转发到大厅，加入匹配队列；
  >
  > `guess`：猜测数字，转发到房间；
  >
  > `help`：列出所有操作说明；
  >
  > `quit`：退出；

- hall

  > `ready`：加入匹配队列；
  >
  > `offline`：用户掉线，需要从匹配队列移除用户；

- room

  > `start`：初始化房间；
  >
  > `online`：用户上线，如果用户在游戏中，告知游戏进度；
  >
  > `offline`：用户下线，通知房间内其他用户；
  >
  > `guess`：猜测数字，推动游戏进程；

游戏演示

- 客户端

  > telnet 127.0.0.1 8888

- 服务端

  > 先启动 redis，然后启动 skynet
  >
  > ```
  > redis-server redis.conf
  > ./skynet/skynet config.game
  > ```

- 如何优化

  > 1. agent 服务不要实时创建，可以采用预先创建；用户验证通过后再分配 agent 地址，避免无效分配；
  > 2. 创建 gate 服务：登陆验证、流程验证、心跳检测、验证成功之后再分配一个 agent；
  > 3. 如果 agent（lua虚拟机）功能比较简单，那么可以创建固定数量的 agent；
  > 4. 如果 room（lua虚拟机）功能比较简单，那么可以创建固定数量的 room；
  > 5. 如果是万人同时在线游戏，agent room 需要预先分配，长时间运行会让服务内存膨胀，同时也会造成 lua gc 负担会加重；
  > 6. 重启服务策略，创建同样数量的 agent 服务组，新进来的玩家，分配到新的服务组；而旧的玩家在旧的服务组操作结束后，就淘汰该玩家，直到旧的服务组没有玩家，这时旧服务组退出；保证旧的服务组只处理旧的任务，新连接进来的用户在新的服务组进行工作；