零声教育 Mark 老师 QQ: 2548898954

多核并发编程

多线程

> 在一个进程中开启多线程，为了充分利用多核，一般设置工作线程的个数为 cpu 的核心数；
>
> memcached 就是采用这种方式；
>
> 多线程在一个进程当中，所以数据共享来自进程当中的内存；这里会涉及到很多临界资源的访问，所以需要考虑加锁；

多进程

> 在一台机器当中，开启多个进程充分利用多核，一般设置工作进程的个数为 cpu 的核心数；
>
> nginx 就是采用这种方式；`ngx_shmtx_t` 自旋锁 信号量 文件锁
>
> nginx 当中的worker进程，通过共享内存来进行共享数据；也需要考虑使用锁；

CSP

> 以 go 语言为代表，并发实体是协程（用户态线程、轻量级线程）；内部也是采用多少个核心开启多少个内核线程来充分利用多核；

Actor

> erlang 从语言层面支持 actor 并发模型，并发实体是 actor（在 skynet 中称之为**服务**）；skynet 采用 c + lua来实现 actor 并发模型；底层也是通过采用多少个核心开启多少个内核线程来充分利用多核；

总结

> 不要通过共享内存来通信，而应该通过通信来共享内存；
>
> CSP 和 Actor 都符合这一哲学；
>
> 通过通信来共享数据，其实是一种解耦合的过程；并发实体之间可以分别开发并进行单独优化，而它们唯一的耦合在于消息；这能让我们快速地进行开发；同时也符合我们开发的思路，将一个大的问题拆分成若干个小问题；

skynet

> 它是一个轻量级游戏服务器框架，而不仅仅用于游戏；
>
> 轻量级体现在：
>
> 1. 实现了 actor 模型，以及相关的脚手架（工具集）；
>   - actor 间数据共享机制；
>   - c 服务扩展机制；
> 2. 实现了服务器框架的基础组件；
>   - 实现了 reactor 并发网络库；并提供了大量连接的接入方案；

> - 基于自身网络库，实现了常用的数据库驱动（异步连接方案），并融合了 lua 数据结构；
> - 实现了网关服务；
> - 时间轮用于处理定时消息；

环境准备

centos

```
yum install -y git gcc readline-devel autoconf
```

ubuntu

```
apt-get install git build-essential readline-dev autoconf
# 或者
apt-get install git build-essential libreadline-dev autoconf
```

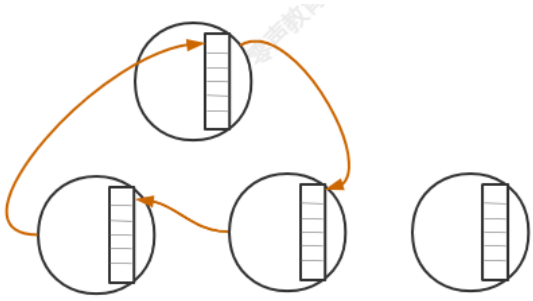
mac

```
brew install git gcc readline autoconf
```

编译安装

```
git clone https://github.com/cloudwu/skynet.git
cd skynet
# centos or ubuntu
make linux
# mac
make macosx
```

Actor 模型



> 有消息的 actor 为活跃的 actor，没有消息为非活跃的 actor；

定义

- 用于并行计算；
- Actor 是最基本的计算单元；
- 基于消息计算；
- Actor 通过消息进行沟通；

组成

- 隔离的环境

  > 主要通过 lua 虚拟机来实现；

- 消息队列

  > 用来存放有序（先后到达）的消息；

- 回调函数

  > 用来运行 Actor；从 Actor 的消息队列中取出消息，并作为该回调函数的参数来运行 Actor；

Actor 创建

> `skynet_context_new` 中会创建一个隔离的环境（lua 虚拟机），一个消息队列，并且需要设置回调函数；

Actor 底层关键接口

```c
// 用于创建隔离的环境
void * skynet_module_instance_create(struct skynet_module *m);
// 用于设置回调函数
int skynet_module_instance_init(struct skynet_module *m, void * inst, struct skynet_context *ctx, const char * parm);
// 用于释放 actor 对象
void skynet_module_instance_release(struct skynet_module *m, void *inst);
// 用于处理 信号 消息
void skynet_module_instance_signal(struct skynet_module *m, void *inst, int signal);
```

Actor 运行

> skynet.start 会设置回调函数，一个消息执行的时候，会获取一个协程执行它；
>
> lua虚拟机有一个限制，同时只有一个协程在运行；
>
> 内核线程 取出消息队列 找到lua虚拟机 从协程池中取出一个协程来执行消息运行

Actor 消息

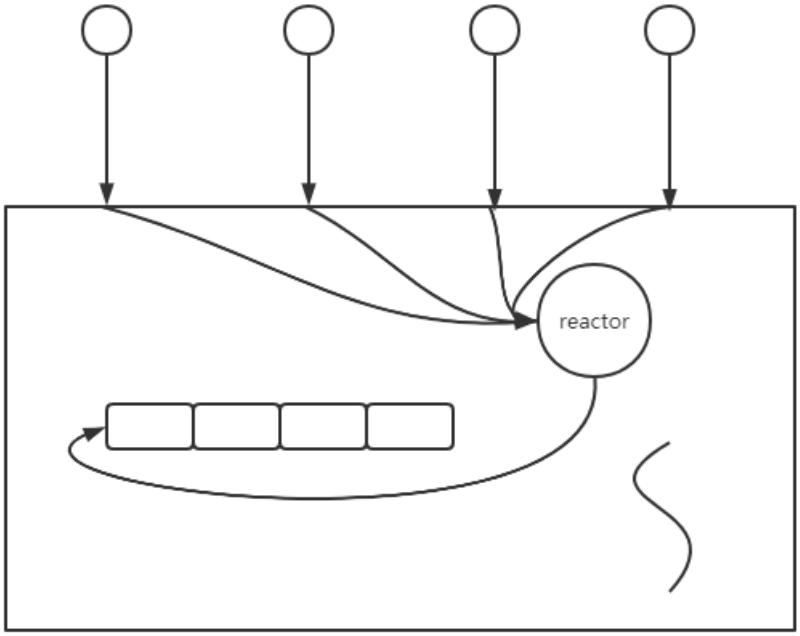
> Actor 模型基于消息计算，在 skynet 框架中，消息包含 Actor （之间）消息、网络消息以及定时消息；

Actor 之间消息

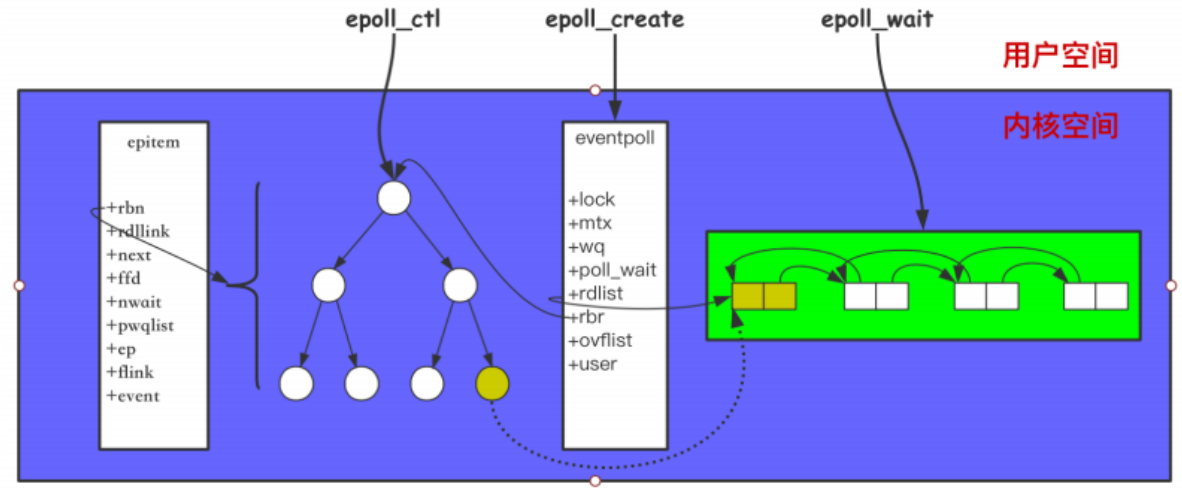
```lua
-- addr 对端服务的地址
-- typename 消息类型 actor内部间通常为 lua 类型消息
-- ... 为可变参
-- skynet.send  socket
skynet.send(addr, typename, ...)

-- addr 对端服务的地址
-- typename 消息类型 actor内部间通常为 lua 类型消息
-- ... 为可变参
-- 注意：
-- 对端需要显示调用 skynet.ret(...) 回应 skynet.call 的请求
-- 或者通过调用 skynet.response() 延迟回应 skynet.call 的请求

-- 在一个协程当中
local ret = skynet.call(addr, typename, ...)

```

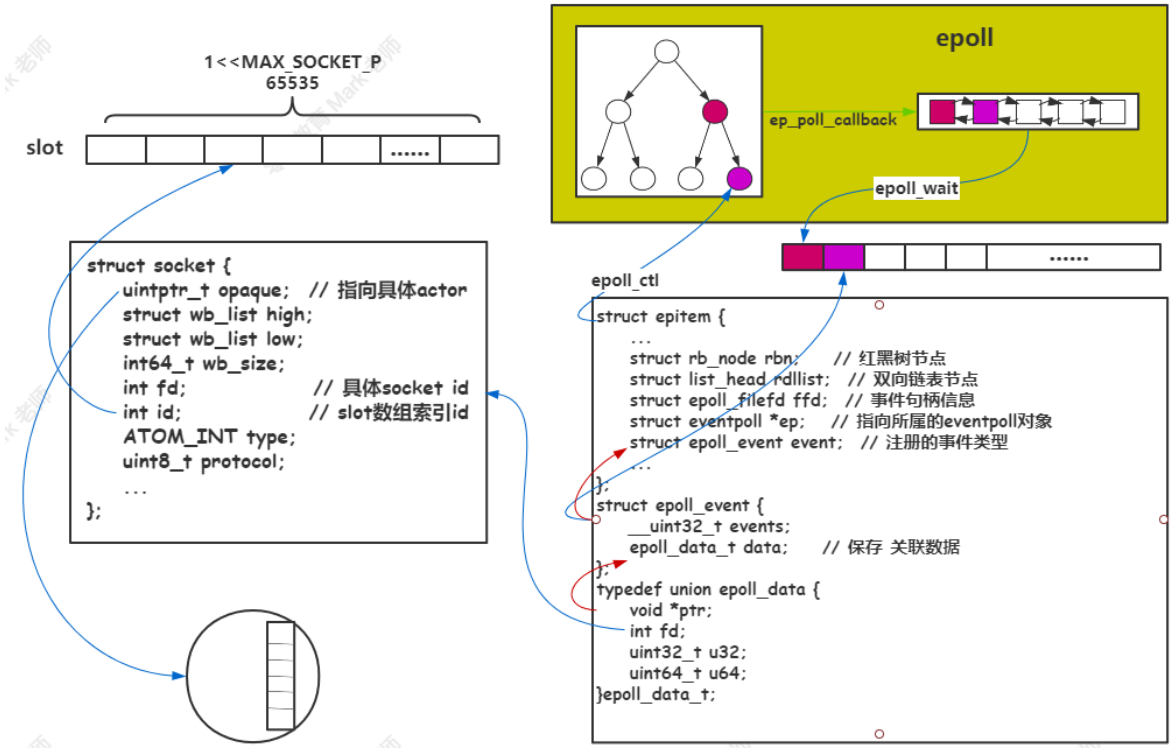
网络消息

> skynet 当中采用一个 socket 线程来处理网络信息；skynet 基于 reactor 网络模型；
>
> 问题：网络当中获取数据，怎么知道传递到哪个服务的消息队列当中去？




```c
// 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；

定时消息

> skynet 采用多层级时间轮来解决多线程环境下定时任务的管理；时间复杂度为 $O(1)$；
>
> 当定时任务被触发，将会向目标 Actor 发送定时消息，从而驱动 Actor 的运行；

消息推送到 Actor

skynet_socket_poll

socket_server_poll

forward_message 将消息推送到所属 actor 的消息队列

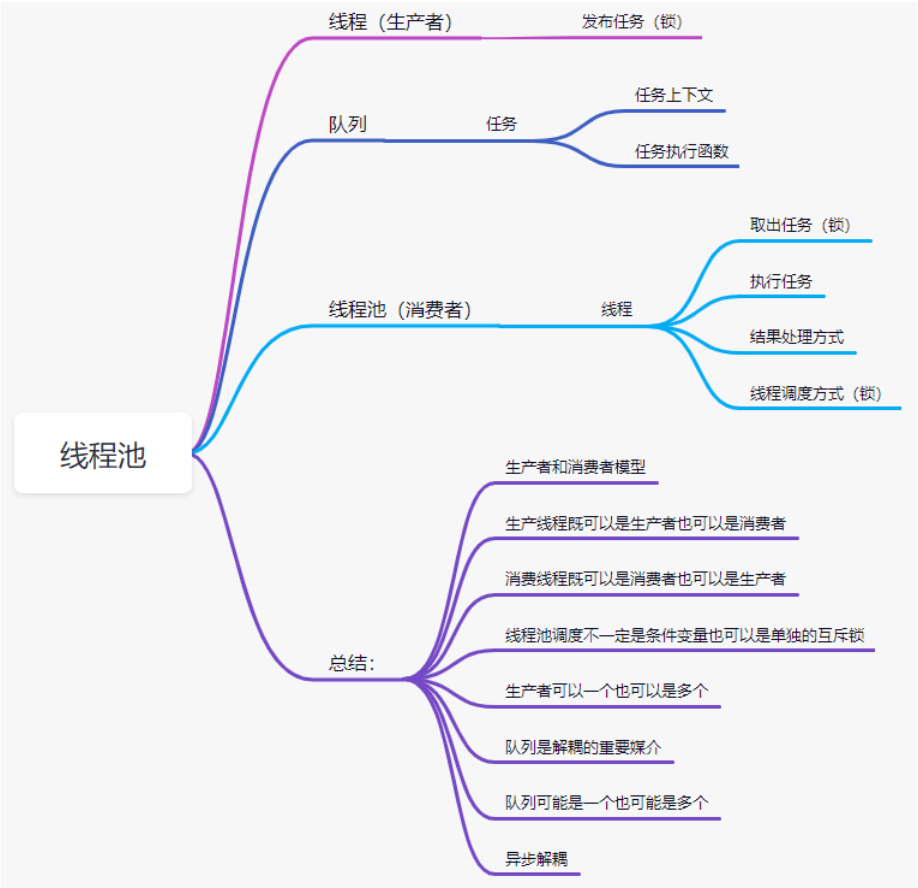
ctrl_cmd	**处理pipe事件（work线程发送过来的事件）**
sp_wait	**阻塞处理io事件**
report_connect	**连接第三方服务 建立成功的标识**
.report_accept	**接收客户端的连接 在这里可以绑定不同的client对应不同的服务**
forward_message_tcp	**读事件**
send_buffer	**写事件**

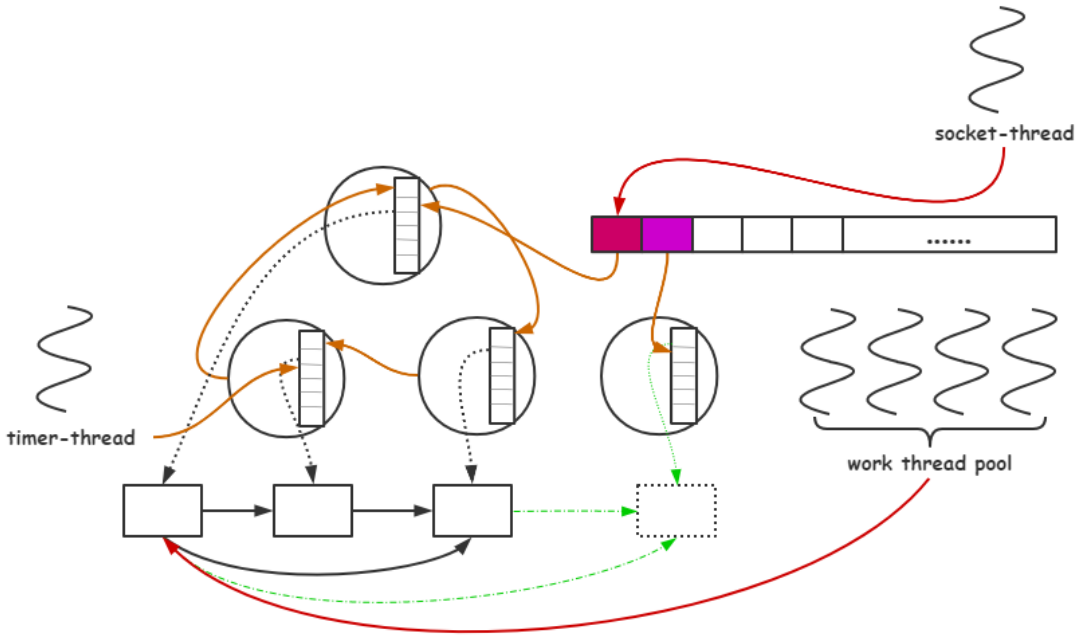
消息处理

> 在 skynet 中，主要通过设置回调函数，为每一条消息选择一个协程去处理；

Actor 调度

> actor 的调度是由线程池的调度来驱动的；





工作线程流程

> 工作线程从全局队列中 `pop` 出单个 Actor 消息队列；从 Actor 消息队列中按照**规则** `pop` 出一定数量的消息进行执行；若 Actor 消息队列中仍有消息继续放入全局队列队尾；若 Actor 消息队列中没有消息则不放入全局队列中；全局队列只存活跃的 Actor 消息队列；

工作线程权重

> 工作线程数量是按照 cpu 核心数来设置的；工作线程按照下面工作线程权重图来设置每个工作线程的权重；

```
// 工作线程权重图    32个核心
static int weight[] = {
    -1, -1, -1, -1, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1,  // 1/2
    2, 2, 2, 2, 2, 2, 2, 2,  // 1/4
    3, 3, 3, 3, 3, 3, 3, 3, }; // 1/8
```

工作线程执行规则

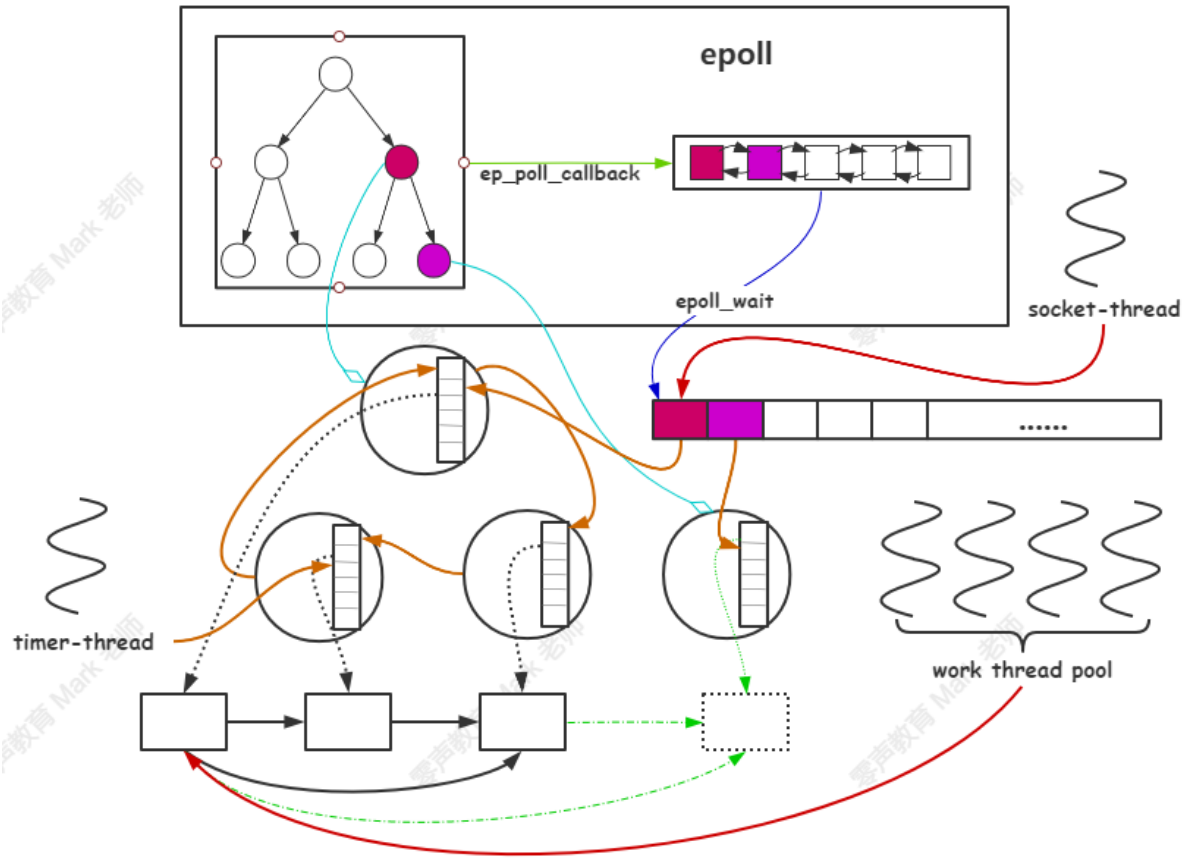
```
int i,n=1;
for (i=0; i<n; i++) {
    // 注意: skynet_mq_pop pop出消息则返回0，没有pop消息返回1
    if (skynet_mq_pop(q, &msg)) {
        skynet_context_release(ctx);
        return skynet_globalmq_pop();
    } else if (i==0 && weight >= 0) {
        n = skynet_mq_length(q);
        n >>= weight;   // n >> 1  = n / 8
    }
    ...
    // 调用 actor 回调函数消费消息
    dispatch_message(ctx, &msg);
}
```

> 从上面逻辑可以看出，当工作线程的权重为 `-1` 时，该工作线程每次只 pop 一条消息；当工作线程的权重为 0 时，该工作线程每次消费完所有的消息；当工作线程的权重为 1 时，每次消费消息队列中 $\frac{1}{2}$ 的消息；当工作线程的权重为 2 时，每次消费消息队列中 $\frac{1}{4}$ 的消息；以此类推；通过这种方式，完成消息队列梯度消费，从而不至于让某些队列过长；这种消息调度的方式不是最优的调度方式（相较于 go 语言），云风也在尝试修改更优的方式来调度；但是目前从多年线上实践情况来看，skynet 运行良好；

调度问题

> 1. 多个工作线程从全局消息队列中取次级消息队列，应该采用什么锁?
> 2. 当 skynet 全局消息队列节点很少的时候，怎么让多余的工作线程得到休眠?
> 3. 在问题 2 的基础上，如果此时全局消息队列节点很多后，怎么让休眠的工作线程得到唤醒?

总体原理图



> 操作粒度时间 小于 cpu 切换时间
>
> actor 是抽象的用户态进程，相对于linux内核 ，有进程调度，那么skynet也要实现actor调度
>
> 1. 将活跃的actor 通过**全局队列**组织起来； actor 当中的**消息队列**有消息就是活跃的actor；
> 2. 线程池去 全局队列中取出 actor 的消息队列，接着运行actor；