skynet启动流程

- 1：main — 检查参数数量，取得配置文件的路径
- 2：skynet_globalinit — 初始化节点的全局参数，设置线程的私有数据表示
- 3：skynet_env_init — 初始化进程的环境参数，创建一个lua虚拟机将来存放配置文件解析出来的参数
- 4：sigign — 获得SIGPIPE信号量的控制权，防止因为往一个已关闭socket写数据而造成进程crash
- 5：luaL_newstate — 生成一个lua虚拟机，解析配置文件
- 6：_init_env — 利用刚才解析出来的配置，初始化第三步生成的虚拟机的全局参数表
- **启动框架这个是核心函数**
- **7：skynet_start**
  - 1：接管，SIGHUP，信号量，防治终端关闭造成进程退出
  - 2：根据配置fork出守护进程，彻底把自己和终端断开联系
  - 3：skynet_harbor_init — 利用配置初始化节点的全局ID
  - 4：skynet_handle_init
    - 这个很重要之后生成的所有actor对象指针都存放在这，并返回一个句柄
    - //全局管理actor对象的对象指针
    - static struct handle_storage *H = NULL;
  - 5：skynet_mq_init
    - 初始化全局有消息的actor的列表容器，这个非常重要，这个是框架的核心
    - //全局有消息需要处理的actor列表
    - static struct global_queue *Q = NULL;
  - 6：skynet_module_init — 动态库管理容器
  - 7：skynet_timer_init — 初始化定时器容器
  - 8：skynet_socket_init
    - 初始化全局的socket管理对象，包括创建epoll句柄，管道句柄，初始化每个socket管理结构
    - 1：sp_create — 创建epoll句柄
    - 2：pipe — 创建管道用于工作线程发控制指令到网络线程
    - 3：生成全局socket管理对象
      - struct socket_server
      - struct socket
    - 子主题
  - 9：skynet_context_new(config->logservice, config->logger) — 创建第一个actor对象，日志actor
    - 1：skynet_module_query — 从模块管理容器查找对应的so
      - _query — 查找是否已经加载了
      - _try_open — 尝试根据名字和路径加载这个so
      - open_sym — 获得so的四个标准函数并保存函数指针
        - _create — typedef void * (*skynet_dl_create(void);
        - _init — typedef int (*skynet_dl_init)(void * inst, struct skynet_context *, const char * parm);
        - _release — typedef void (*skynet_dl_release)(void * inst);
        - _signal — typedef void (*skynet_dl_signal)(void * inst, int signal);
    - 2：skynet_module_instance_create — 调用so的创建接口创建一个实例对象
    - 3：创建一个actor对象，skynet_context
      - 1：ctx->instance = inst,把刚才的实习对象指针保存在actor对象中
      - 2：ctx->handle = skynet_handle_register(ctx);把actor放入actor管理容器中
      - 3：skynet_mq_create(ctx->handle);创建一个和actor关联的message_queue对象，这个对象将来会放到全局消息列表中
      - 4：skynet_module_instance_init，初始化第一步创建的对象
        - [redacted]
      - 5：skynet_globalmq_push，把第三部创建的消息对象放入全局消息列表
  - 10：skynet_handle_namehandle(skynet_context_handle(ctx), "logger") — 把actor对象和对应的名字放入一个容器之后可以用名字找到这个actor
  - 11：bootstrap — 根据配置中的actor名字再创建一个对应的actor
    - c服务启动流程
      - 如果是c服务
    - lua服务启动流程
      - 如果是lua服务 — 通过skynet_context_new加载snlua.so模块
        - snlua_create — 创建一个snlua
          - 分配一个snlua的数据结构
          - 设置内存警告阈值
          - 设置内存上限
          - 分配一个lua虚拟机
        - snlua_init — 初始化刚才生成的snlua对象
          - 解析配置参数，bootstrap = "snlua bootstrap"
          - 设置actor回调函数为，launch_cb
          - skynet_send，给自己发一个消息，消息内容就是刚才解析出来的参数bootstrap — 发送的消息经过消息总线分发进入launch_cb回调函数
            - skynet_callback，设置回调函数为null，组织其他回调处理，这个在lua代码中会调用c的接口重新设置回调，把消息的分发处理权限转移到lua代码中
            - init_cb，这个函数做了很多的初始化工作
              - lua_gcl, LUA_GCSTOP, 0);，设置lua不要gc
              - lua_requiefl1, "skynet.profile", init_profile, 在注册性能检测函数到lua
              - 替换lua携带的两个重要函数resume，wrap进入我们的函数，加入性能检测代码，有点类似于c的函数hook
              - 设置ctx指针进lua
              - 设置几个全局变量LUA_PATH，LUA_CPATH，LUA_SERVICE，LUA_PRELOAD
              - 把参数压入lua并执行loader.lua
                - 解析传入的参数（bootstrap）
                - 根据参数初始LUA_PATCH，找到文件./service/bootstrap.lua
                - 加载./service/bootstrap.lua
                - 替换_G.require为skynet.require中的require函数，相当于hook了系统的require
                - M.require
                  - 查看模块是否已经加载过了，加载过就直接返回
                  - 在配置的路径中寻找模块，如果找不到调用系统的require函数
                  - 加载文件，如果加载失败调用系统的require
                  - 维护一个loading_queue检查是否其他携程也在运行require函数，如果有的话，使用skynet.wakeup函数把自己挂起
                    - skynet的携程和调度机制，单独写一个文档
                - M.init_all
                - M.init
                - main(select(2, table.unpack(args)))，运行bootstrap.lua — skynet.start 这个函数的作用是设置了一个定时器还是触发要启动函数，并且设置这个lua虫回调函数为lua函数，把消息的分发处理权已从c转移到了lua
                  - c.callback(skynet.dispatch_message)，消息分发转到lua函数
                  - skynet.timeout，延迟触发bootstrap的匿名函数
                    - skynet.launch("snlua","launcher") — 启动launcher服务，这个服务负责之后所有其他的服务启动，skynet.launch流程c中的cmd_launch，马上启动一个新的actor，是个很重要的
                    - skynet.name("launcher", launcher) — 注册名字和对应的actor句柄
                    - harbor_id == 0如果是非集群模式
                      - pcall(skynet.newservice, "cdummy")
                        - 调用skynet.newservice创建新的cdummy服务，这个服务是发消息给luncher服务，让luncher服务创建的
                      - skynet.name(".cslave", slave) — 把cdummy服务，注册成.cslave的名字，代替真正的.cslave
                    - harbor_id == 1如果是集群模式
                      - pcall(skynet.newservice,"cmaster") — 如果是单例集群模式，自己启动一个master
                      - pcall(skynet.newservice, "cslave") — 启动cslave服务
                      - skynet.name(".cslave", slave) — 把".cslave"名字和slave句柄，注册到管理容器
                    - 非集群和集群单例模式
                      - skynet.newservice "datacenterd" — 启动daatacenterd服务
                      - skynet.name("DATACENTER", datacenter) — 注册进容器
                    - skynet.newservice "service_mgr" — 启动service_mgr服务，这个服务和 luncher服务有点类似，调用接口是skynet.uniqueservice和skynet.queryservice，启动的服务会记录下来，luncher服务不会提供注册发现的功能
                    - 如果设置了enablessl — 这个是websocket的问题，这个也单独写文档
                    - 查找配置中start的配置，如果没有就运行main.lua — pcall(skynet.newservice,skynet.getenv "start" or "main") — 生成一个新的服务
                      - skynet.newservice("logind") — 启动logind服务 — 开新的文档这个文档内容太多了
                      - skynet.newservice("gated", loginserver) — 启动gated服务 — 开新的文档这个文档内容太多了
                      - skynet.call(gate, "lua", "open" , {
                        port = 8888,
                        maxclient = 64,
                        servername = "sample",
                        }) — 告诉gate服务按参数开启socket监听
                      - 这个actor也不干啥，为啥不像examples/main.lua一样退出啦？
                    - bootstrap的使命完成了，退出这个服务 — skynet.exit()
  - 12：start
    - 非常重要这个地方开启了核心的工作线程
    - 1：struct monitor *m = skynet_malloc(sizeof(*m));　分配一个监控结构
    - 2：struct skynet_monitor　每个线程分配一个监控子对象
    - 3：create_thread(&pid[0], thread_monitor, m);//创建监控线程　skynet_monitor_check　根据监控结构的数据循环判断是都有线程死循环如果有就输出日志
    - 4：create_thread(&pid[1], thread_timer, m); //创建定时器线程　定时器线程就是轮询个每个定时器到时间了就发送一个消息给定时器师定的actor
    - 5：create_thread(&pid[2], thread_socket, m);//创建网络线程　thread_socket　skynet_socket_poll
      - 线程中循环调用这个函数处理管道和socket数据，管道用的select,socket用的epoll水平触发模式
      - socket_server_poll
        - has_cmd — 判断管道时候有数据
          - //管道操作码
          - /*
            The first byte is TYPE

            S Start socket
            B Bind socket
            L Listen socket
            K Close socket
            O Connect to (Open)
            X Exit
            D Send package (high)
            P Send package (low)
            A Send UDP package
            T Set opt
            U Create UDP socket
            C set udp address
            Q query info
            */
        - ctrl_cmd — 读出指令处理
          - 真正的socket的对外连接，监听端口等都是通过actor发送指令到网络层，网络线程来实际执行的，创建到socket就会和actor句柄进行绑定，之后socket的所有事件都是传给actor去处理
        - sp_wait — epoll_wait,等待socket数据
        - [redacted] — [redacted]
          - 真正的从socket读取数据的接口，这个框架使用的epoll水平触发模式这个有改进的空间
        - [redacted] — [redacted]
        - send_buffer — send_buffer_ — send_list
          - 并没有实际的发送数据只是把数据复制到发送队列
        - [redacted] — [redacted]
        - //socket事件类型
        - #define SOCKET_DATA 0
        - #define SOCKET_CLOSE 1
        - #define SOCKET_OPEN 2
        - #define SOCKET_ACCEPT 3
        - #define SOCKET_ERR 4
        - #define SOCKET_EXIT 5
        - #define SOCKET_UDP 6
        - #define SOCKET_WARNING 7
    - 6：create_thread(&pid[i+3], thread_worker, &wp[i];创建工作线程　skynet_context_message_dispatch
      - skynet_globalmq_pop — 从全局消息容器弹出一个消息对象
      - skynet_mq_handle — 从消息对象得到处理消息的actor句柄
      - skynet_handle_grab — 从actor句柄得到actor对象
      - skynet_mq_pop — 从全局消息对象得到实际的消息对象
      - 核心函数分发开处理消息
        - [redacted] — reserve_msg = ctx->cb(ctx, ctx->cb_ud, type, msg->session, msg->source, msg->data, sz);//调用actor回调指针处理消息
      - 把当前消息对象压入容器，并弹出下一个
    - 7：pthread_join(pid[i], NULL);　阻塞主线程并等待所有线程退出，框架到现在真的运行起来了
  - 12：退出主线程
- 8：退出主线程，走到这一步就退出进程了