**Skynet注释**

1. **Skynet中lua服务的消息处理**
2. 服务初始化时,skynet.start函数会调用skynet.core.callback函数，重置消息处理函数指向lua-skynet的\_cb函数，该函数又会调用lua函数skynet.dispath\_message函数，并传入参数 @1消息type @2消息data @3数据length @4消息session @消息source
3. skynet.dispatch\_message函数会调用raw\_dispatch\_message函数，参数依次传入
4. **协程间的交互**
5. 服务初始化时，Skynet.start函数会调用skynet.timeout函数，push服务处理的第一条消息，消息类型为PTYPE\_RESPONSE，消息号为session。然后调用co\_create（func）创建一个协程，func指向服务实际的消息处理函数。消息号session和消息处理协程co保存为kv对session\_id\_coroutine[session] = co，当服务处理到消息号为session的消息时，调用suspend(co,coroutine\_resume(co,session,source,msg,sz))的方式唤醒协程，并传入参数。
6. 服务处理非PTYPE\_RESPONSE类型的消息时，会根据类型选取对应的proto，该对象包含服务对于该种消息的pack、unpack、消息处理函数。然后把消息处理函数传递给一个空闲协程（如无空闲则新建），使用两个table session\_coroutine\_id、session\_coroutine\_address分别保存协程对应处理消息的session及消息source。调用suspend(co, coroutine\_resume(co, session,source, p.unpack(msg,sz)))函数唤醒协程并传递参数。
7. 消息处理结束后，协程保存到一个局部table coroutine\_pool中，以备后用。并向主协程返回”EXIT”命令，用于清理。
8. **Skynet的类库**
9. **skynet类**
10. skynet.call(addr, typename, ...)

消息发送，需要目标服务响应。向目标服务@addr发送类型为@typename的消息，变参为消息数据，调用yiled\_call执行等待。

1. yiled\_call(service,session)

挂起协程。返回“CALL”命令，挂起并保存协程，等待消息响应。响应的消息类型为PTYPE\_RESPONSE。

1. skynet.send(addr, typename, ...)

消息发送，无需响应。

1. skyent.wait(co)

挂起协程并等待。逻辑为分配一个session，并调用“SLEEP”命令，保存session-co对，sleep\_session[co]=session，等待调用skynet.wakeup(co)。

1. skynet.wakeup(co)

唤醒协程。在wakeup\_session表中标记co状态为可唤醒。每次消息处理结束，会调用一次dispatch\_wakeup函数。

1. dispatch\_wakeup()

从wakeup\_session表中依次取出挂出的co，通过suspend(co, coroutine\_resume(co, false, "BREAK"))唤醒。

1. skynet.newservice(name, ...)

通过skynet.call启动服务。具体过程如下：实例，A服务调用该函数启动服务B

* A调用skynet.call(".launcher", "lua" , "LAUNCH", "snlua", name, ...)，向launcher服务发送一条"LAUNCH"命令，等待响应。
* Launcher服务通过command.launch函数处理该消息。
* command.launch函数调用launch\_service函数。
* launch\_service函数调用skynet.launch函数启动服务B，调用skynet.response函数取得响应函数并保存。
* 服务B启动后，会想launcher服务发送一条”LAUNCHOK”命令
* Laucher服务执行command.LAUNCHOK命令，通过保存的响应函数向服务A返回响应信息，即服务B的handle值

1. skynet.ret(msg,sz)

调用“RETURN”命令，返回msg，sz。

1. skynet.response(pack)

调用“RESPONSE”命令，命令执行时，会取出消息源服务，返回一个响应函数，该函数执行时，会把数据发送到数据源服务中。

1. skynet.register\_protocol(class)

class为table类型，定义了不同消息类型的unpack函数、消息处理函数、类型名、类型标记。Lua服务里的所有class会保存在一个局部变量table proto中。

1. **socket类**
2. socket.listen(host,port,backlog)

调用socketdriver.listen函数完成监听操作，返回skynet socket id。

1. socket.start(id,func)

第一步调用socketdriver.start函数，把skynet socket添加到epoll中，监听socket fd的读事件；第二步调用connect(id,func)函数，对于server sock，func实际为accpet到客户端连接后的处理函数，对于client sock，func为nil。Connect函数会调用suspend函数挂起协程。

1. socket\_message[SKYNET\_SOCKET\_TYPE\_ACCEPT ]（id,newid,addr）

处理SKYNET\_SOCKET\_TYPE\_ACCEPT 类socket消息，使用socket.start传递的func函数执行func(newid,addr)

1. socket\_message[SKYNET\_SOCKET\_TYPE\_CONNECT]（id,newid,addr）

处理SKYNET\_SOCKET\_TYPE\_CONNECT 类socket消息，当skynet socket添加到epoll后发送到相关服务。

消息处理函数会修改table socket的connected状态，并唤醒调用connect函数挂起的协程。

1. socket\_message[SKYNET\_SOCKET\_TYPE\_DATA]（id,newid,addr）

处理SKYNET\_SOCKET\_TYPE\_DATA 类socket消息，当skynet socket接收到数据时发送该类型消息。在socket.lua文件中，有一个table保存了所有的s（socket）描述符，每个s有个数据缓冲池，接收到数据后，把数据push到缓冲池中。

1. socket.read(id,sz)

从skynet socket读取大小sz的数据。如果sz为0，则读取缓冲池里的所有数据。如果调用时，缓存池为空，则挂起协程，等到

1. socket.write(fd,content)

指向socket.driver.send函数

1. socket.close(id)
2. **命令**
3. CALL命令:

当服务A向服务B发送了一条消息，并且等待服务B响应时使用，通过在服务A中保存了消息号session-协程co实现。响应的消息类型为PTYPE\_RESPONSE。

1. SLEEP命令：
2. RETURN命令
3. REPONSE命令
4. EXIT命令
5. QUIT命令
6. USER命令
7. **管道请求和epoll事件**
8. Listen请求

表示server sock已经执行完listen操作，但还未添加到epoll中。Socket\_server.c文件中的Listen\_socket函数处理L请求，执行skynet socket初始化操作

1. Start请求

开始监听sock event，对于server sock开始监听客户端连接，对于client sock，开始监听数据传输，设置skynet socket相应的状态，并向所属的服务发送SKYNET\_SOCKET\_TYPE\_CONNECT类型的socket消息

1. D（SEND）请求

socket\_server.c文件的send\_socket函数用以处理D请求。

1. K（CLOSE）请求

socket\_server.c文件的close\_socket函数用以处理D请求。

1. Accept事件

Server sock发出Start请求并得到处理后，会开始监听客户端连接，Socket\_server.c文件的report\_accept函数处理客户端连接。会分配一个skynet socket，并初始化。

1. **Socket消息类型**
   * + 1. SKYNET\_SOCKET\_TYPE\_ACCEPT

Server sock接收到客户端连接，分配skynet socket并完成初始化后，Server sock的所属服务发送一条SKYNET\_SOCKET\_TYPE\_ACCEPT的socket消息。

2.

1. **Sproto**
2. **HTTP**
3. **Test**