**CCS代码分析**

# 总体

MCP框架分为CCS、MCP、SCC三大模块，其间关系已经被描述很多遍了，这里不再赘述。从代码角度看，三者结构比较相似。上周五我阅读了MCP框架的CCS模块主要流程，这里将我的阅读体会总结一下。

CCS从结构上看是一个典型的多线程服务器，包含一个主线程和N个辅助线程。主线程负责监听接入的请求，accept后将FD传给辅助线程；辅助线程实际进行数据的收发工作。主线程和辅助线程都是通过epoll来监听socket的。辅助线程对每个客户端连接维护一个变长缓冲区，存储一次未读取完的数据。

总的来说，CCS的结构和之前看的Memcached结构比较像。Memcached用管道，CCS用CodeQueue；Memcached用Libevent，CCS用CEPollFlow。

# 关键类

关键类有如下几个：

* CMainCtrl
* CAuxHandle
* CCodeQueue、CCodeQueueMutil
* TIdxObjMng
* CBuffMng

下面对这些类依次作介绍。

* CMainCtrl
  1. 简介：整个程序的主要逻辑类，包含初始化和程序的主循环
  2. 主要接口：
     1. CMainCtrl::Initialize
        1. 作用：根据指定的配置文件（路径）初始化程序
        2. 主要流程：
           1. 读配置文件：调用CMainCtrl::ReadCfgFile。

在这里读取了要监听的所有端口，及其对应的包完整性检查SO，并挂此SO得到了完整性检查的函数，和包头长度，存入m\_stConfig.m\_stTcpIPPortStu[]中。

* + - * 1. 初始化日志：创建日志共享内存并调用CMainCtrl::LogInit。
        2. 如果配置了默认的完整性检查so，挂之。
        3. 创建从、到MCP的两个管道（CCodeQueueMutil）
        4. 创建辅助线程（CAuxHandle），数量与队列数量相等

这里的“队列数量”，指的是每个CCodeQueueMutil内包含的CCodeQueue的数量， 由MQ配置文件中的“Number”属性指定。

* + - * 1. 创建监听Socket

初始化主线程的epoll

对配置文件中指定的每个监听端口（由TcpIPAddrXX、UdpIPAddrXX）指定，创建一个监听Socket，并将其FD加入epoll，等待连接进入。

* + 1. CMainCtrl::Run
       1. 作用：主线程的主循环
       2. 主要流程：
          1. 处理退出和重读配置的信号
          2. 等待epoll。

如果主线程的epoll返回IN事件，那必是新连接进来了。因为主线程的epoll放的都是监听socket。Accept新的连接，检查源地址允许访问后，选择一个辅助线程，将新连接的fd传给辅助线程进行处理。

选择辅助线程的标准是：当前其负责的连接数最少。

* CAuxHandle
  1. 简介：“辅助”线程（其实主线程才应该叫“Aux”……）。负责实际与客户端通信。每个辅助线程都包含一个从主线程的CodeQueue m\_Main2MePipe，和与MCP通信的一对CodeQueue。另外，包含两对用于管理连接缓冲区的TIdxObjMng和CBuffMng。
  2. 主要接口：
     1. CAuxHandle::Initialize
        1. 作用：初始化辅助线程
        2. 主要流程：
           1. 从主线程获得与MCP通信的一对CodeQueue
           2. 创建与主线程通信的CodeQueue
           3. 创建消息缓冲区

消息缓冲区由TIdxObjMng和CBuffMng管理，存在两对，分别负责网络的收和发流量。

* + - * 1. 初始化本线程的epoll

此epoll负责监听主线程、MCP到自己的CodeQueue，和所有属于本线程负责的网络Socket。

* + - * 1. 绑定CPU
    1. CAuxHandle::Run
       1. 作用：线程的主循环
       2. 主要流程：
          1. 等待epoll。
          2. 处理与客户端的通信：CheckClientMessage
          3. 处理与主线程的通信：CheckMainMessage
          4. 处理与MCP的通信：CheckSvrMessage
          5. 处理通信超时：CheckTimeOut

遍历本线程负责的socket，检查其创建时间。若超过TimeOutSecs则清理此Socket。

* + 1. CAuxHandle::CheckClientMessage
       1. 作用：处理epoll返回的events，对各种event做出不同的处理
       2. 主要流程（按不同的event分）：
          1. TCP客户端传来数据：

从缓冲区管理器CBuffMng中获取当前socket已用的缓冲长度。若不为0，则将当前已用的缓冲区部分拷贝出来，放在一个数组内，并使用包完整性检查函数计算还需多少字节可构建完整包。

从socket接收数据，拼接在上述数组的末尾。

如果本次接收数据量达到预期，则使用包完整性检查函数检查是否获得了完整包。

如果完整了，则放入到MCP的CodeQueue，并将剩余的数据存入本socket对应的缓冲区。如果仍不完整，则将本次接收的数据存入socket对应的缓冲区，返回。

如果上一步中检查出错，则丢弃数据并断开此socket。

* + - * 1. UDP客户端传来数据：

检查客户端IP是否允许

接收数据并放入到MCP的CodeQueue（不对UDP维护缓冲区）。

* + - * 1. Socket可以发送数据（EPOLLOUT）：

将此socket对应的发送缓冲内容发送出去。

若此次将发送缓冲发完，则去掉此socket的EPOLLOUT。

* + - * 1. 主线程或MCP的CodeQueue传来数据（Fifo）：

调用CheckMainMessage或CheckSvrMessage

* + 1. CheckMainMessage和CheckSvrMessage
       1. 作用：检查从主线程和MCP有没有传来的信息。二者只是操作的CodeQueue不一样，为什么不合并成一个函数……？
       2. 主要流程：
          1. 循环从CodeQueue中取出一条消息
          2. 调用DoSvrMessage
    2. DoSvrMessage
       1. 作用：实际处理从CodeQueue中取出的消息
       2. 主要流程（按消息命令类型分）：
          1. 新TCP和新UDP Socket（主线程通知辅助线程新的连接即走这一分支）：

调用CreateSocketNode保存Socket信息并加入本线程epoll。UDP不保存客户端IP和端口（无连接）。

* + - * 1. 被通知断开指定Socket（CMD\_CCS\_NOTIFY\_DISCONN）：

取出指定Socket

断开并清理

* + - * 1. 被通知断开所有客户端连接（CMD\_CCS\_MCP\_NOTIFY\_DISCONN）：

向本进程所有辅助线程广播CMD\_CCS\_THREAD\_NOTIFY\_DISCONN

* + - * 1. 被通知断开所有客户端连接（CMD\_CCS\_THREAD\_NOTIFY\_DISCONN）

断开并清理本线程所有的socket

* + - * 1. MCP发来的数据（CMD\_DATA\_TRANS）

根据发回的suffix找到对应的客户端socket，若找不到说明已经超时，原socket已经被清理了。

如果原Socket是TCP协议，查看其发送缓冲是否有内容。如果发送缓冲没有内容，直接尝试发送，若一次没发完，则剩余内容加入发送缓冲区；若发送缓冲本来就有内容，说明上次没发完，将新数据加入缓冲区末尾。如果本步结束后发送缓冲区处于非空状态，则在epoll中添加此socket的发送事件监控（当socket可写时提醒线程）。

如果原Socket是UDP协议，则使用sendto直接发送。

但凡发送发生错误，都关闭并清理此socket。

* CCodeQueue、CCodeQueueMutil
  1. 参见MCP的doc文件夹下《CCS学习笔记》
* TidxObjMng
  1. 简介：TidxObjMng索引对象管理器。TidxObjMng可将受管理的块组织成“多维”链表，即为每个块附加“DSNum”（Dimensions Number，维度数）个index域。同时对上提供接口。CBuffMng等一系列封装类将此类管理的空间通过这DSNum个封装成缓冲区、树、哈希等数据。此类的DSNum、块大小都可通过参数设置，极端灵活！
  2. 内存组织：

TidxObjMng将被管理的内存空间按照如下格式进行划分：

| TIdxObjHead | TIdx \* n | Obj \* n |

TidxObjHead记录了：

* + - * ssize\_t m\_iFreeHead; //第一个空闲的块下标
      * ssize\_t m\_iUsedCount; //当前使用了多少块
      * ssize\_t m\_iObjSize; //每个Obj的大小
      * ssize\_t m\_iIdxObjNum; //本实例的容量（可容纳多少个Obj）
      * ssize\_t m\_iIdxSize; //每个Obj附加的DSNum个Index所占的大小

TIdx的定义：

* idx\_t m\_piDsInfo[0]; //不定长数组，通过下标访问每个维度的index

实际格式化此块内存时Obj数量，动态将m\_piDsInfo的长度设为DSNum，即每个Obj对应DSNum个index域。

* Obj：

Obj是被管理的对象。一块TidxObjMng中实际可容纳的Obj数量计算方法：

N = （TotalSize – sizeof（TidxObjHead））/（sizeof（Obj）+ sizeof（TIdx））

TidxObjMng将未使用的Obj通过对应的TIdx:: m\_piDsInfo[0]（第0维度的下标）串成一个FreeList，用以快速分配。因此，DSNum要求最小为1（体现在代码中）。

* 1. 主要方法：
     1. TIdxObjMng**::**AttachMem
        1. 作用：使用TIdxObjMng实例管理某块内存，并按照上述格式格式化。
        2. 主要流程：
           1. 根据所需的DSNum计算每个对象的Index域的大小（TIdx的实际大小）、本对象实际可管理的Obj的数量
           2. 格式化Index域：将所有Obj通过第0维下标串成Freelist，其他维度下标初始化为-1。
     2. TIdxObjMng::GetOneFreeDS
        1. 作用：获取本实例尚未使用的第一个DS，并将其标记为使用（分配DS）。
        2. 主要流程：
           1. TIdxObjMng包含成员ssize\_t m\_piDsUseFlag[MAX\_DSINFO\_NUM]，是保存哪个DS被使用的Bitmap。遍历此成员，找到第一个未使用的DS，标记为使用，并返回DS号。
     3. TIdxObjMng::CreateObject
        1. 作用：从Obj域中分配一个未使用的
        2. 主要流程：
           1. 从m\_iFreeHead处取出一个Obj
           2. 将m\_iFreeHead修改为指向此Obj的下一个Obj
           3. 将此Obj的Index域清空（TIdxObjMng不再负责管理这个Obj）
           4. 返回此Obj的Index
     4. TIdxObjMng::DestroyObject
        1. 作用：释放一个Obj块，还原到Freelist中
        2. 主要流程：将Obj插入Freelist的头部
* CBuffMng
  1. 简介：

本类是在TidxObjMng定长块管理器上层封装出的不定长缓冲区管理器。它负责为各个TCP连接维护各自的缓冲区。此缓冲区可由TidxObjMng中的多个块链式连接组成。

本类使用了一个DS。

* 1. 内存组织：

本类同样需要一块预先分配的内存，并将其组织成如下格式：

| TBuffMngHead | TBuffItem \* n |

TBuffMngHead的定义：

* ssize\_t m\_iBuffNum; //缓冲块个数（对多少个连接维护缓冲区）
* ssize\_t m\_iFreeBuffNum; //空闲缓冲块数（还能服务多少个连接）
* ssize\_t m\_iDSSuffix; //在对象管理器中使用的链（使用了第几个DS）

TBuffItem的定义：

* idx\_t m\_iBuffIdx; //第一块位置（对应于TidxObjMng中Obj的下标）
* idx\_t m\_iBuffSize; //总数据长度
* idx\_t m\_iLastBuffIdx; //最后一块位置
* idx\_t m\_iBuffOffset; //起点偏移

在CCS代码中，n的值为MAX\_SOCKET\_NUM。故保证可为每个连接都维护一个缓冲区。同时，SocketSuffix可直接用于BuffSuffix。SocketSuffix是Socket的FD模MAX\_SOCKET\_NUM的值，类似一个哈希值。根据SocketSuffix可快速找到Socket实例和BufItem。

刚刚初始化并附加到内存块时，内存被初始化为：

|…………….. TBuffMngHead……………………………..| TBuffItem…………..|

|MAX\_SOCKET\_NUM|MAX\_SOCKET\_NUM|-1|[-1|-1|-1|-1]|......|

即：可为MAX\_SOCKET\_NUM个连接提供服务，空闲MAX\_SOCKET\_NUM个，尚未附加到TidxObjMng。

* 1. 主要方法
     1. CBuffMng::AttachIdxObjMng
        1. 作用：绑定到对象管理器实例
        2. 主要流程：
           1. 从TidxObjMng分配一个DS（调用）GetOneFreeDS
           2. 将DS号记录在head中。
        3. 此函数完成后的内存：

|MAX\_SOCKET\_NUM|MAX\_SOCKET\_NUM|DS\_ALLOCATED|[-1|-1|-1|-1]|......|

表明CBuffMng使用了TidxObjMng的第DS\_ALLOCATED维下标。（实际上是0）。

* + 1. CBuffMng::AppendBuffer
       1. 作用：将指定数组内的内容添加到某个连接的缓冲区末尾。如果当前此连接未使用缓冲区，还要负责分配。
       2. 主要流程：
          1. 如果当前Socket尚未使用缓冲区，为socket分配BuffItem：

直接以传入的SocketSuffix当下标查BuffItem数组，若BuffItem的m\_iBuffIdx（TidxObjMng中的index）为-1，则代表未使用。

调用TidxObjMng的CreateObject，分配一个初始块。

将BuffItem的BuffIdx和LastBuffIdx都指向刚分配的Obj（此块内存从此归CBuffMng）管理。

将传入数组的头sizeof（Obj）个字节拷贝到分配的Obj中。

调用TIdxObjMng::SetAttachObj

如果传入数组没拷贝完，则重复从ii开始的步骤，并把新的Obj串到之前的Obj之后，组成一个链表。最终效果为BuffItem的BuffIdx指向此链表的头，LastBuffIdx指向此链表的尾。

* + - * 1. 如果当前Socket已经使用了若干缓冲块

计算当前已使用最后一块还能装多少数据

填满最后一块

若还需要新的空间，则按a.v处理。

* + 1. CBuffMng**::**GetBuffer
       1. 作用：根据传入的SocketSuffix，将其缓冲区内容拷贝到一个数组中
       2. 主要流程：
          1. 根据BuffItem找到第一个缓冲区Obj块（位于TIdxObjMng的内存中）
          2. 从第一块拷贝数据
          3. 若未拷贝完，则遍历Obj组成的链表，一块一块拷贝出来。
    2. CBuffMng::SendBufferToSocket
       1. 作用：将指定SocketSuffix的缓冲区发送到指定SocketFD
       2. 主要流程：
          1. 基本过程同CBuffMng::GetBuffer
          2. 不拷贝，而是直接调用send发送内容。
    3. CBuffMng::FreeBuffer
       1. 作用：释放指定SocketSuffix对应的缓冲区
       2. 主要流程：
          1. 基本过程为一个单链表的释放
          2. 对每一个块调用TIdxObjMng::DestroyObject

# 总结

以上分析了CCS除了CLoadGrid、Log等辅助功能以外的代码，基本覆盖了所有主要执行流。看了这个代码让我感触良多，尤其是TidxObjMng和CBuffMng部分，实在是牛！这实际上是自己实现了一套内存分配管理机制，保留了灵活性的同时避免了大量的系统调用。尤其是DS机制，为上层提供了灵活的接口，使得TidxObjMng可被封装为表、数、哈希等各种结构，自己扩展起来也很方便！这是我闻所未闻，见所未见的。

写的比较乱，也没有配图，让大家见笑了！