**服务器底层平台移植工作分析与设计**

本文是对服务器在linux平台下底层技术设计的分析，主要用来分析底层平台（主要是网络模型、线程架构、定时器）移值需要做哪些工作以及这些工作使用的技术。

由于windows平台与linux平台一些关键技术上的差异（最主要是网络模型），服务器底层需要重新设计，工作主要包括两大块——I/O模型及线程架构。在linux平台，公认的高性能的I/O模型是epoll，我们根据epoll的特性以及当前游戏服务器的架构来选择线程架构（也就是并发模型）。

**一、确定几种关键技术：**

***（一）I/O模型。***epoll不仅可用用在套接字的监听，还可以使用在定时器、信号的监听，这些技术都可以有epoll进行无缝连接，用来提高性能。

套接字不用多说，使用epoll解决C10K问题已经是公认的事实。

定时器是游戏服务器不可或缺的技术，由于游戏逻辑的复杂性，可能同一时刻，系统内有上万个定时器在同时运行，使用epoll这种高效的I/O模型可以降低对定时器的超时的检测，提高服务器性能。

信号。信号在游戏服务器可以用来做运维控制，常见的比如说刷脚本。使用epoll来做非阻塞处理，简单且高效。

***（二）网络层的设计模式。***Reactor模式是当前比较流行的网络层设计模式，它的好处可以把不同类型的I/O事件抽象成统一的接口，简化并发的设计，而且具有良好的扩展性。

***（三）并发模型。***并发模式主要是指线程的架构，最常见得两种设计是领导者-跟随者模型（leader-followers）和半同步半异步模型（half-sync/half-async）。

leader-followers模型性能高效，但设计上相对复杂，比较适合简单、可并发的大规模逻辑应用（如果不能并发就无法体现这种模型的高性能）。

half-sync/half-async模型其实就是生产者-消费者模型，这种模型设计简单，性能好，但实现上可能出现较多的内存拷贝以及动态内存的申请，比较适合做游戏服务器，最主要的原因是因为游戏的逻辑比较复杂，很难去把逻辑做成多线程。对于动态内存申请的问题是可以通过数据结构的设计来避免的，而对于内存拷贝的问题理论上是无法完全避免的，但这相对于逻辑的复杂性而言，少量的内存拷贝并不是问题。这种模型最重要的是如何平衡好生产者的生产速度与消费者的消费速度，这是长期调试才能得出一个比较好的平衡点。

***（四）线程处理模型。***我们这里使用I/O事件非阻塞加同一个I/O事件只固定在一个线程中处理的模式（也就是一个线程对应多个I/O事件），这种设计简洁、安且性能高效。

如果使用I/O事件阻塞模式，线程在某一个I/O事件没有触发时就会被挂起，这样线程就不能去处理其它I/O事件，这样对CPU的利用率会降低，性能远远达不到要求。

如果I/O事件与线程的关系是多对多的关系，那么模型的复杂度就大大增加了，其主要目的多是为了平衡每个线程的负载，而多对一的关系下线程的负载问题一般是按I/O事件的个数来，以游戏服务器数据量来说，按事件个数来做线程负载平衡已经相当有效了，没有必要做到绝对意义上的平衡。

**二、具体分析：**

***（一）服务器线程架构分析。***我们需要把不同的功能划分到不同的线程去处理，让线程各司其职，相互合作。

***1. 主线程。***由于主线程的默认优先级是比较高的，所以让主线程只占有少量的CPU时间是正确的选择，利用这个性质我们可以让主线程等待客户端的连接，这里有两种处理方法：

*(1) 阻塞方式。*主线程阻塞等待连接的到达，这种方法的好外是主线程几乎不占CPU时间，也可以及时相应连接的到达。

*(2) 非阻塞方式。*主线程使用非阻塞方式来探测连接是否到达，这种方式下主线程还要以去做一些其它事情，比如回收连接等。

***2. 日志线程。***服务器在windows平台下使用一个单独的日志线程，通过共享队列（相当于一个生产者-消费者模型）来读取逻辑线程SetEvent触发的日志信息。日志线程可以保留这种设计方式，日志线程是一个消费者，逻辑线程是一个生产者，由于消费者日志线程只是用来打印日志的，所以它可以是阻塞的（当没有日志时，一直处于阻塞状态，一旦队列里出现日志时，线程就被激活），而生产者逻辑线程除了日志还需要处理其它逻辑，所以它应该是非阻塞的，那么这就要求我们的通信的队列是无边界的，我们可以使用无边界的环形缓冲区来做，同时考虑是否要避免动态内存分配的开销。

***3. 逻辑线程。***逻辑线程主要用于处理应用层的逻辑，与网络层进行通信，与日志线程通信，这些线程间的通信都使用生产者-消费者模型。逻辑线程还有一个重要的功能，就是逻辑定时器，由于应用逻辑需要大量的定时器，所以必须放在同一个线程里处理，这个见下文所述。

***4. 网络线程。***网络线程由主线程的线程池进行管理，线程池管理多组网络线程以及所有的TCP连接，网络线程主用是连接建立后与客户端通讯的代理，网络线程和逻辑线程是一个全双工的生产者-消费者模型，而网络线程也有两种划分方式：

*(1) 平等式划分。*平等式划分是指网络线程只有一种类型的线程，从连接建立成功后，到连接结束都在同一个线程中处理，内部需要有一个类似状态机，在不同状态处理不同操作。这种设计的优点是线程的个数比较少，缺点是线程内部处理相对复杂，无谓的状态判断会有些费时。

*(2) 按功能划分。*按功能划分是指网络线程按职责划分成几种不同功能的线程，按职责划分为验证线程（连接建立后，需要一次登录，防止非法连接进入）、通信线程（接收消息，发往逻辑线程，从逻辑线程接收消息，发送给客户端）、回收线程（回收死连接或需退出去连接），使连接在不同的状态让不同的线程去处理，每种类型的线程只提供一种功能。这种设计的优点是结构清晰，避免了无谓的状态判断，缺点是增加了线程个数（一般可能会多两到三个线程），有可能会增加线程之间的通讯量。对于增加线程之间的通讯量主要是指连接对象在状态改变时需要切换线程，但这个是可以解决的，在架构设计中，TCP连接不应该属于任何一个网络线程（也就是连接对象是不放入线程对象中的），只放入线程池统一进行管理（线程池可以由主线程来管理），需要哪个线程进行处理时，只需让线程的epoll关注这个连接的socket描述符即可（原有线程取消关注），这样就没有线程之间的通讯了。

不管在设计上如何选择，在线程的实现上，不同功能的线程应该在形态上是一样的，也就是说，专门为验证线程做一个类不是一个好的设计，验证线程和回收线程应该是相同的类，而划分是在线程池中进行的，线程本身并不知道自己是哪种类型的线程。

***5. 客户端网络线程。***客户网络线程是指负责服务器主动发起的TCP连接的线程，这里是把它单独列出了，而在实现上，即可以把这种线程并入到处理服务器的网络线程中，也可以单独设置一个或多个线程，这是根据实现而定的。在我们游戏服务器的实现中，主动发起的TCP连接是不多的，以功能划分的方式，设置一个线程完全就可以满足需求，为了更好的利用线程的CPU时间片，我们可以把负载少的线程进行合并，提高线程循环利用率。

***（二）网络线程池设计。***网络线程池主要是管理调度网络线程，负责启动线程、关闭线程、分配TCP连接以及网络线程的均衡负载。

***（三）定时器设计。***这里的定时器是指逻辑上所使用的定时器，所以它必须是跑在逻辑线程上的，而这里所讲的是定时器的触发方式，并非定时器的内部设计（这个还是使用之前服务器的设计方法，使用观察者模式实现），这里定时器的触发设计有两种实现方式：

*1. 帧循环触发。*这种方式是指在逻辑线程的帧循环中，得到一次系统时间，传给定时器内部，进行一些比较，对于超时观察者进行到时回调。

*2. 使用timerfd\_xxx调用。*这类API都有关联一个fd描述符，可以无缝的和epoll进行关联，这样每一次逻辑线程的帧循环，只需要从epoll得到发生的事件（就是超时的定时器fd），然后做观察者回调即可。但这里有个问题，如果每做一次定时，都关联一个描述符，这个并发量可能会达到几万或几十万，这样的数据量对于系统可能有较大的负担，解决这个问题，我们可以把定时器分为几类，如10毫秒、100毫秒、1秒、5秒、10秒，30秒、1分钟、5分钟、10分钟、30分钟、1小时、1天，这样数量级在100个以下的，对系统就没有任何负担了，思想是这样的，具体算法还需要更具体的设计。

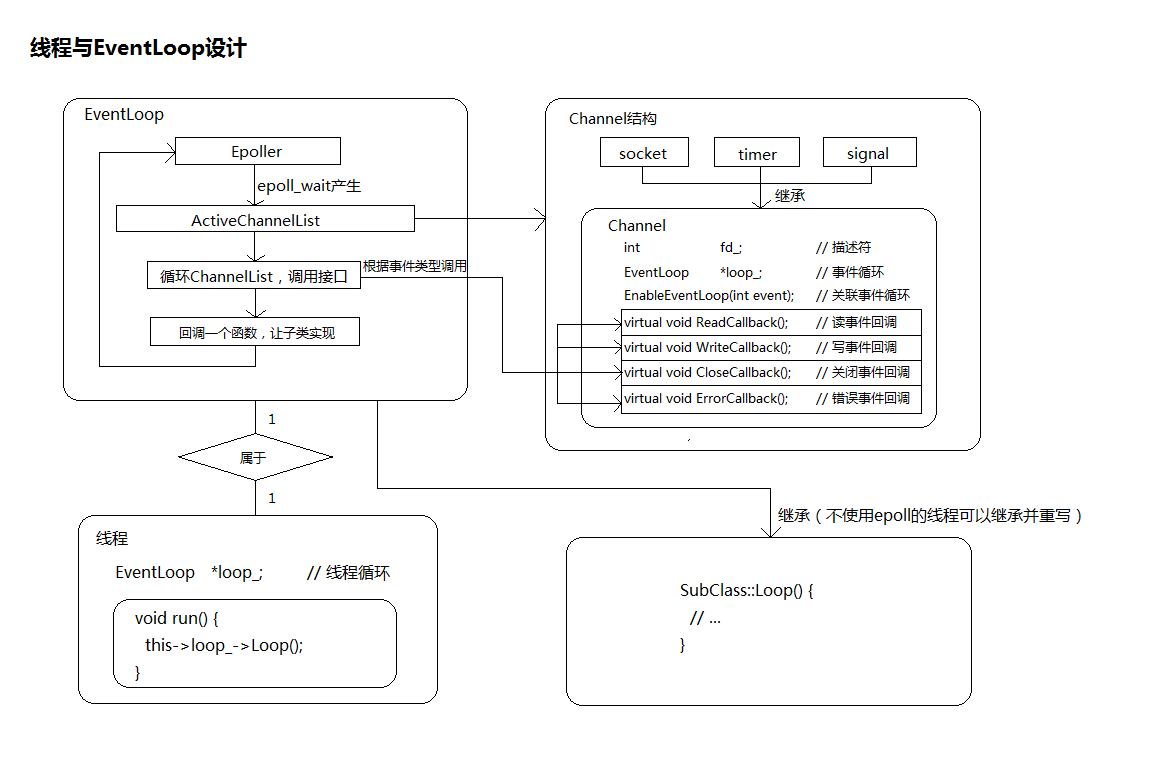
***（四）连接关闭流程。***关闭流程一般分为客户端主动关闭、和服务器主动关闭。客户端主动关闭从服务器角度来看，是由逻辑线程触发的关闭，网络线程与调度线程通过状态来检测，从而达到连接的关闭；而服务器主动关闭是由调度线程触发的关闭，也通过状态来通知逻辑与网络线程，从而达到连接的关闭。

**三、具体设计：**

整个设计使用Reactor模式，核心使用一个Loop做为线程循环体，线程与Loop的对应关系肯定是一对一的。一个Loop是一个简单的线程循环，它需要一个while循环体，使用一个状态字段控制循环结束，循环体需要支持非阻塞模式的epoll\_wait所主导的事件触发后的处理，也需要支持阻塞模式的处理（没有epoll\_wait的最好都不能使用非阻塞模式，因为完全非阻塞模式的生产者-消费者可能会浪费CPU时间，例如主线程、日志线程等）。所以就需要有两种处理方式：

第一种是EventLoop，每一个EventLoop需要对应一个epoll描述符，每个循环中，使用epoll\_wait()来得到触发的事件（Channel，是一个抽象类，由具体实现来继承），根据事件类型调用事件所指向的函数回调处理，处理完成后需要执行一个函数，来做一些类似于sleep的调用（由具体实现进行处理），需要注意的是，每一个Channel同一时间只与一个EventLoop关联，也就是说一个Channel只由一个线程处理。

第二种是非Event的Loop，由具体需要继承Loop类（比如主线程、日志线程、调度线程），自主实现线程循环里的处理。



***（一）服务器线程架构设计。***

***1. 主线程。***主线程启动后，读取配置文件，创建并启动日志线程、逻辑线程，创建网络线程池，网络线程池根据配置文件创建并启动各种网络线程，建立各种数据结构，然后主线程进入循环中，线程循环中，需要阻塞模式地等待tcp连接的到来。每当tcp连接到来时，把连接写入调度线程（也是阻塞的写入）。

主线程的Loop是实现一个具体的类，继承Loop，在线程循环中，阻塞地调用accept函数，等待连接的到来，如果有连接到达，线程被激活，创建一个TCP连接对象，放入与调度线程通信的有边界的环形队列（阻塞的放入）。

***2. 日志线程。***使用生产者-消费者模型与逻辑线程做日志通信，日志线程做为消费者，使用无边界环形队列，以阻塞模式（用条件变量实现，没有日志时，阻塞等待，不占CPU）读取日志信息，把日志打印到文件中。

日志线程的Loop是实现一个具体的类，继承Loop，在线程循环中，阻塞的读取与逻辑线程的无边界环形队列，读到日志信息时，把日志打印到文件中。

***3. 逻辑线程。***逻辑线程与网络线程的通信使用无边界的环形队列进行非阻塞模式的通信，逻辑线程与日志线程的通信使用无边界的环形队列进行单阻塞模式的通信（日志线程阻塞，逻辑线程非阻塞）。逻辑线程需要实现epoll处理信号、定时器两种channel的EventLoop处理，在每个线程循环中，先使用epoll\_wait处理信息与定时器逻辑，完成后从环形队列中读取数据包，做循环解析处理。

逻辑线程也同样的实现一个具体的Loop类，继承Loop，在线程循环中做以上处理。

***4. 网络线程。***网络线程按功能划分，不同的线程处理不同的工作，具体如下：

*(1) 调度线程。*调度线程只需要一个，调度线程是负责接收主线程发来的TCP连接，负载均衡分配到验证线程组，检测超时的TCP连接，回收需要终止的TCP连接，发起客户端TCP连接（使用非阻塞模式连接，然后关联到一个客户端线程组的一个EvenLoop中）并设置连接成功后的回调。需要实现一个具体的Loop类，继承Loop，在线程循环中使用生产者-消费者模型读取主线程发来的TCP连接，读取TCP回收列表，回收列表中的TCP连接，循环正在连接的TCP连接，找到超时连接，放入回收列表，循环结束。

*(2) 验证线程组。*其中的每个验证线程处于平等地位，使用EventLoop做为线程循环，epoll\_wait得到触发的事件（应该只关注读事件）后，这里的事件Channel都是TCP连接对象指针，TCP连接对象实现一个登录验证函数，验证通过后，通过网络线程池得到通讯线程组（网络线程池实现成一个singleton），再通过线程组调用相关函数得到一个EventLoop，把Channel关联到这个EventLoop，就完成了TCP连接的验证。

*(3) 通讯线程组。*每个通讯线程处于平等地位，使用EventLoop做为线程循环，epoll\_wait得到触发事件后，每个事件Channel（TCP连接对象）实现通讯函数（接收与发送网络消息），与逻辑线程通过生产者-消费者模型进行通信。

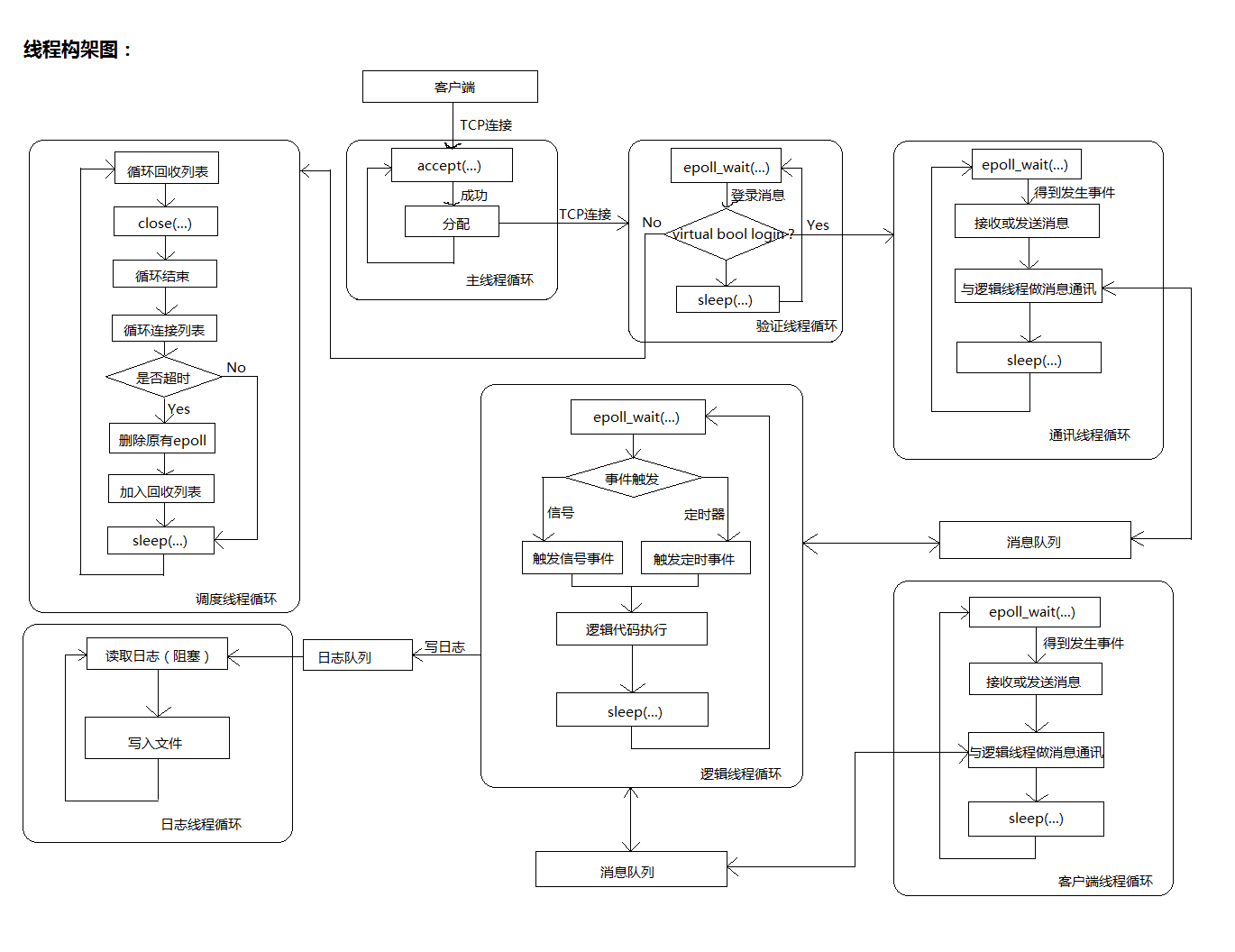
*(4) 客户端线组。*每一个客户端线程是一个EventLoop循环，使用epoll\_wait得到触发的事件后，如果是按之前注册的回调去调用Channel的函数（如果响应的是连接成功，则通知逻辑线程新建客户端连接对象，这里的通信应该在线程建立时，在逻辑线程与客户端线程之间建立一个共享队列），每个事件Channel（是一个主动发起的TCP连接对象）与逻辑线程通过生产者-消费者模型进行通信。

*这里有几个问题：*

第一，由于验证线程是处理连接登录验证的，那么当有一段时间没有连接登录时，验证线程会在循环中空转，从而浪费CPU时间，所以这个验证线程需要避免这种浪费，这需要验证线程做一个关联计数（使用条件变量来做线程同步，互斥锁保证线程安全），调度线程每次向验证线程的EventLoop关联Channel（TCP连接）时，并把关联计数加一，调用一次signal。对于验证线程，每个EventLoop循环中，检查关联计数，如果关联计数是小于等于0，则进行条件变量的wait，等待有TCP连接进入，每当完成一个TCP连接验证，把关联计数减一。这种方式加入了互斥锁，但互斥锁带来的效率影响应该是远远比验证线程空转CPU小。

第二，如果调度线程检测到TCP连接（Channel）超时，会把这个Channel在EventLoop（位于通讯线程中）关联进行删除，但由于是多线程程序，如果EventLoop那个线程执行到了触发事件，从epoll\_wait得到了Channel的指针，还有没来的急处理这个Channel，这时线程失去了CPU，CPU给了调度线程，调度线程又检测到这个Channel超时，把这个Channel在EventLoop的关联进行了删除，并释放了Channel对象，那么在EventLoop下一次得到CPU时，这个Channel对象指针被从栈中弹出，变成了一个野指针（同理逻辑线程也有一个这样的对象）。解决方案：在Channel中设计一个状态，表示这个Channel是否已超时，由调度线程去除与EventLoop的关联后，关闭socket，并设置这个状态并移入等待释放队列，通讯线程与逻辑线程读取并判断。通讯线程需要在每次epoll\_wait得到触发事件时，判断是否已超时（不能修改EventLoop设计，应该在回调函数中实现），如果超时设计另一个状态（如通讯关闭状态）；对于逻辑线程每次从Channel列表（就是逻辑的TCP连接任务列表）中读取消息时，先判断是否已超时，如果超时则设置第三个状态（如逻辑关闭状态）并从队列中删除；在调度线程每次检测等待释放队列中的Channel时，检测通讯关闭状态与逻辑关闭状态是否都设置，如果都设置了，就释放这个对象。

第三，什么时候把连接成功的Channel放入逻辑线程，按流程说，应该在验证线程验证完成后，放入逻辑线程的Channel列表中，但由于系统中有多个验证线程，这样多对一的线程通信方式是必须加互斥锁的，影响性能，所以我们可以在进入调度线程后，在线程分配验证线程时，同时写入逻辑线程的Channel列表，这样的通信是一对一的，可以做成使用无锁环形队列做生产者-消费者通信，只需要设置一个状态，就是是否TCP连接是否验证通过，在逻辑线程中设置两个队列，一个队列是未验证完成的TCP连接，另一个队列是工作队列，调度线程发来的TCP连接对象放入未验证完成的TCP连接队列，每个逻辑线程循环开始时，遍历这个队列，把验证完成的TCP连接放入工作队列。



***（二）网络线程池设计。***

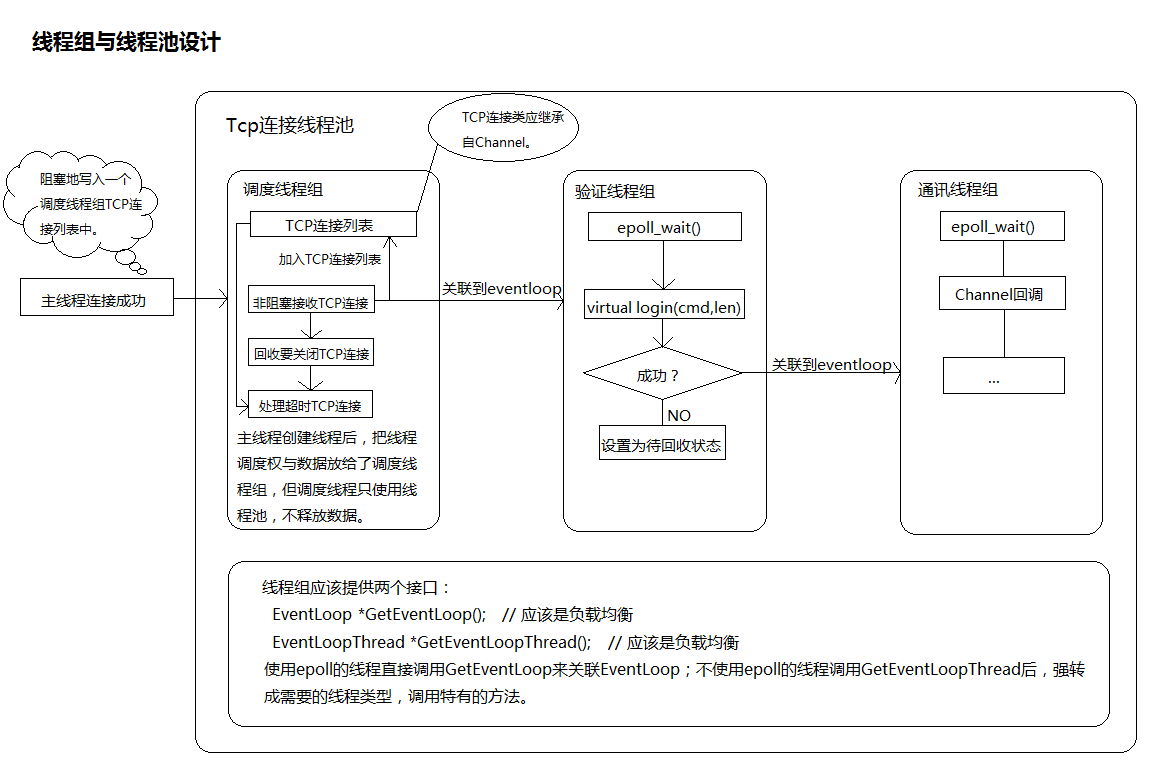
在网络线程中，理论上所有种类的线程都是支持多个的，但因为调度线程的任务量较小，而且需要与逻辑线程进行通信，所以设计上没有让调度线程支持多开。这样网络线程池中就有一个调度线程与三个功能线程组，分别是验证线程组，通讯线程组，客户端线程组。除此之外，线程池中应该管理所有TCP连接，复杂TCP连接的负载平衡到线程组的每个线程。

***1. 线程组。***线程组。线程组是一组线程的集合，也相当于一个小的线程池，这个线程组需要提供两个基本接口：

*(1) 获取一个EventLoop指针。*要求负载平衡，这个主要用在不同种类的线程之间的切换，为了达到线程之间的Channel无缝的切换，切换的目标应该是EventLoop方式的线程，使用关联EventLoop（实际上是关联Epoll）来提高性能。

*(2) 获取一个EventLoopThread指针。*要求负载平衡，这个主要用在切换到不使用EventLoop方式的线程情况，这种情况需要得到一个负载最小的线程，从而实现Channel切换。

***2. TCP连接池。***TCP连接是网络线程池的任务，这些连接对象需要在不同的线程之间切换，但需要注意的是，一个对象在不同的线程上修改时，如果没有进行互斥，这个对象就是不安全的，所以在管理上TCP对象只存放在网络线程池中，不属于某一个线程，某个线程只负责调用当前回调函数，而这些TCP连接的管理是放在调度线程中管理，调度线程从主线程中接管网络线程池，拥有最高权限。



***（三）定时器设计。***

定时器的设计使用linux提供的timerfd\_xxx系统调用，因为定时器是提供给逻辑上应用的，所以把定时器放在逻辑线程中，分成多个时间精度的定时器，逻辑线程的每个EventLoop（当然是继承EventLoop后重新实现）中需要使用epoll\_wait得到超时的定时器，并回调注册的回调函数。

***（四）连接关闭流程设计。***

连接中需要设置三个状态，分别是逻辑关闭标志、通讯关闭标志、调度（回收）关闭标志，再具体地看两种情况：

***1. 客户端主动关闭。***客户端主动关闭连接是逻辑线程收到关闭消息后：

*（1）逻辑线程：*设置与通讯线程的RingQueue设置为完成状态，再设置logic的关闭标志，做上层析构动作，再从逻辑连接队列中删除连接对象。

*（2）通讯线程：*读取与逻辑线程的RingQueue时，读取结果为完成状态时，把缓冲区中的数据发送到客户端，完成后关闭连接（解除epoll的关联并丢弃掉接收缓冲区中的所有数据），设置network的关闭标志。

*（3）调度线程：*检测是否可回收（逻辑与通讯关闭标志是否都已设置），直接回收连接、析构，并从连接队列中删除连接。

***2. 服务器主动关闭。***服务器主动关闭连接是由调度线程收到信号时，开始进行的关闭流程：

*（1）调度线程：*收到信号时，设置连接的调度关闭标志，循环中检测是否可以回收（逻辑与通讯关闭标志是否都已设置），如果一旦可回收时，进行回收连接、析构，并从连接队列中删除连接。

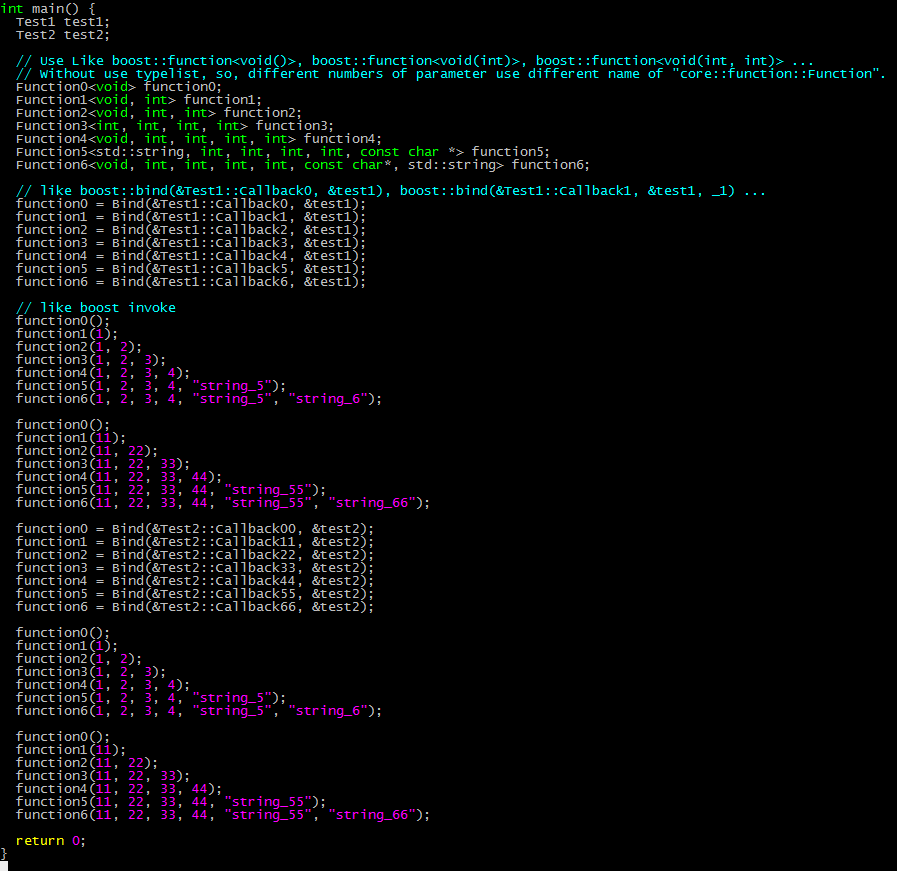
*（2）逻辑线程：*线程循环中检测调度线程关闭标志，如果关闭标志被设置，则发送踢连接消息给客户端，并设置与通讯线程的RingQueue为完成状态，再设置logic的关闭标志，做上层析构动作，从逻辑连接队列中删除连接对象。

*（3）通讯线程：*读取到与逻辑线程的RingQueue为完成状态时，把缓冲区中的数据发送到客户端，完成后关闭连接（解除epoll的关联并丢掉接收缓冲区中的所有数据），设置network的关闭标志。

当然，以后实现了一个服务器监控系统时，做类似客户端主动关闭的方式，实现服务器主动关闭的另一种方式。

**四、实现：**

因为线程在使用epoll\_wait方式的EventLoop，EventLoop循环是不需要重新实现的，所以只使用继承+virtual的方式是没有办法实现的（因为C++中使用virtual的多态只能实现动态绑定一个虚函数），由于我们不使用Boost库，所以我们要实现一个类似于Boost::bind+Boost::function的机制，但实现上只需要满足EventLoop这种一个回调能多次进行动态绑定即可。在这里我们使用C++的成员函数指针，具体设计和实例如下：



基本类结构图：

