ngx_stream_limit_conn_module 模块解析

ngx_stream_limit_conn_module 模块在 stream 子系统中是用来限制某个 ip 的并发连接数的。在 stream 子系统中，虽然没有像 http 子系统那样在代码中明确地定义处理阶段，但是其处理流程也是按照一定的阶段来划分的，stream 处理阶段包括：Post-accept、Pre-access、Access、SSL、Preread、Content、Log。

按照上面的阶段划分，ngx_stream_limit_conn_module 模块就是处在 Pre-access 阶段的，下面便结合代码来分析下该模块的实现。

一、配置命令解析

ngx_stream_limit_conn_module 模块包括三条配置指令，分别是：limit_conn_zone、limit_conn 和 limit_conn_log_level。其中 limit_conn_log_level 主要是用来设置日志级别的，这里暂时不讨论。下面的配置文件就是使用 ngx_stream_limit_conn_module 模块实现限制客户端并发连接数的例子，配置文件如下：

```
stream {
    limit_conn_zone $binary_remote_addr zone=addr:10m;
    ...
    server {
        ...
        limit_conn       addr     5;
    }
}
```

Nginx 在代码中对 limit_conn_zone 和 limit_conn 指令的定义如下：

```
static ngx_command_t    ngx_stream_limit_conn_commands[] = {

    { ngx_string("limit_conn_zone"),
      NGX_STREAM_MAIN_CONF|NGX_CONF_TAKE2,
      ngx_stream_limit_conn_zone,
      0,
      0,
      NULL },

    { ngx_string("limit_conn"),
      NGX_STREAM_MAIN_CONF|NGX_STREAM_SRV_CONF|NGX_CONF_TAKE2,
      ngx_stream_limit_conn,
      NGX_STREAM_SRV_CONF_OFFSET,
      0,
      NULL },
     ……
   ngx_null_command
};
```

先来看下 limit_conn_zone 命令，该命令主要是用来申请一块用于存储客户端 ip 并发连接数的共享内存。Nginx 是如何解析和组织 limit_conn_zone 指令参数的？其解析函数 ngx_stream_limit_conn_zone()的执行流程如下：

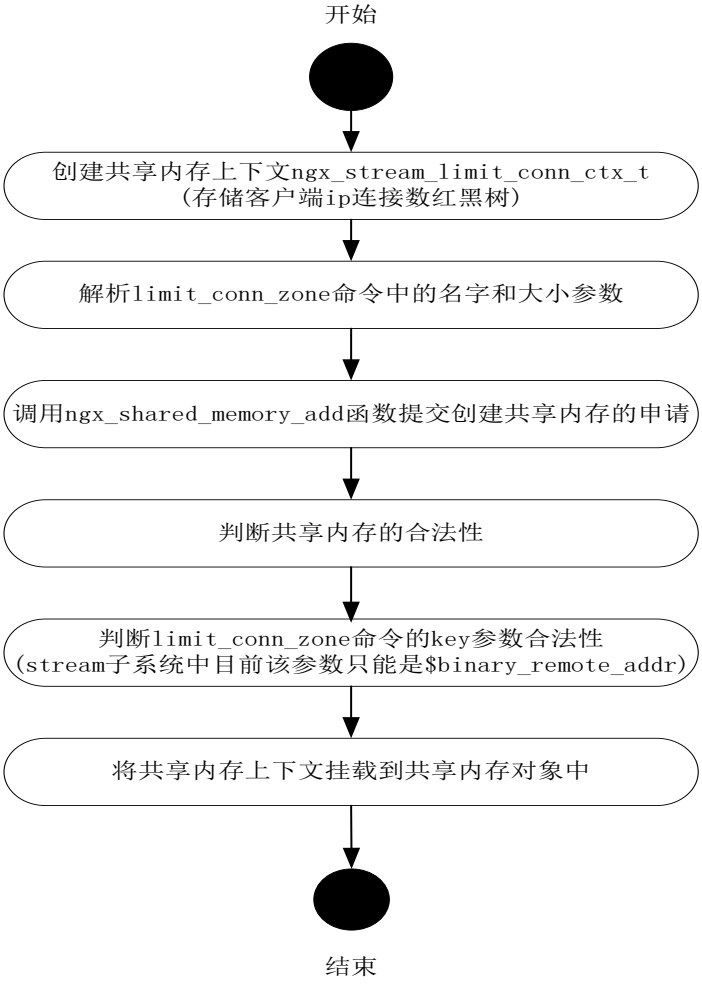

图 1 limit_conn_zone 指令解析函数

从上面的流程中可以看出，一条 limit_conn_zone 命令在 Nginx 内部对应一块共享内存，而一块共享内存就会对应一个模块上下文，那这个模块上下文是用来做什么的呢？先来看下它的定义：

```
typedef struct {
    ngx_rbtree_t                  *rbtree;
} ngx_stream_limit_conn_ctx_t;
```

从上面的定义中可以看到，这个模块上下文其实就是定义了一颗红黑树，从后面的功能实现我们会发现 Nginx 就是利用这棵红黑树来组织不同客户端 ip 的并发连接信息的，一个客户端 ip 对应一个红黑树节点，红黑树的节点的内存都是从 limit_conn_zone 命令定义的共享内存中获取的。为什么要使用共享内存呢？因为一个客户端的多个并发连接请求不一定会被同一个 worker 子进程处理，所以需要用共享内存来存储同一个 ip 的并发连接信息已使所有的子进程都可见。

再来看下 limit_conn 命令，该命令主要是用来设置限制客户端 ip 并发连接数所使用的共享内存和并发连接数大小。那么 Nginx 又是如何解析和组织 limit_conn_zone 指令参数的，其解析函数 ngx_stream_limit_conn()执行流程如下：

```
static char *
ngx_stream_limit_conn(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
```

```
    ngx_shm_zone_t                          *shm_zone;
    ngx_stream_limit_conn_conf_t      *lccf = conf;
    ngx_stream_limit_conn_limit_t    *limit, *limits;

    ngx_str_t      *value;
    ngx_int_t       n;
    ngx_uint_t      i;

    value = cf->args->elts;

    /*  申请一块 shm 共享内存  */
    shm_zone = ngx_shared_memory_add(cf, &value[1], 0,
                          &ngx_stream_limit_conn_module);
    ……

    limits = lccf->limits.elts;
     ……
    /*
     *判断是否已经有重复的共享内存，也就是说不同的 limit_conn 命令
     *不能指定同一个共享内存
     */
    for (i = 0; i < lccf->limits.nelts; i++) {
          if (shm_zone == limits[i].shm_zone) {
                return "is duplicate";
          }
    }

    /*
      *  获取 limit_conn 第二个参数，因为是限制的 ip 的连接数，
      *  所以将其转换为数字
      */
    n = ngx_atoi(value[2].data, value[2].len);
    ……
    limit = ngx_array_push(&lccf->limits);
    if (limit == NULL) {
          return NGX_CONF_ERROR;
    }

    /*  保存解析结果  */
    limit->conn = n;
    limit->shm_zone = shm_zone;

    return NGX_CONF_OK;
}
```

从上面的执行流程中可以看到，在解析 limit_conn 命令的时候也会调用 ngx_shared_memory_add 函数告诉 Nginx 内核需要使用一块共享内存，而在解析 limit_conn_zone 命令的时候也会调用 ngx_shared_memory_add 函数告诉 Nginx 需要使用一块共享内存。可以注意到，这两个地方调用这个函数的时候所使用的共享内存名字和参数 tag 都是一样的，换句话说他们都是指向同一块共享内存，只是在解析 limit_conn 命令调用 ngx_shared_memory_add 函数时指定的共享内存大小是 0，而解析 limit_conn_zone 命令调用 ngx_shared_memory_add 函数时指定的共享内存大小是一个确切的值（配置文件中配置），这么处理也是符合常理的，因为 limit_conn 也确实不知道共享内存大小，所以便指定共享内存大小为 0。另外，Nginx 并没有规定这两条命令出现的先后顺序，所以此时解析配置文件的时候会有如下两种情况：

1、 配置文件中先配置了 limit_conn_zone 命令。这种情况下，当解析到 limit_conn_zone 指令时，因为配置文件中之前没有配置过同名字和 tag 的共享内存，所以此时会新增一个共享内存节点，挂载到全局唯一的 ngx_cycle_t 对象的 shared_memory 成员中。然后解析到 limit_conn 指令的时候则会通过匹配名字和 tag 直接返回刚刚新增的那块共享内存。

2、 配置文件中先配置了 limit_conn 命令。这种情况下，当解析到 limit_conn 指令时，也会因为配置文件中没有配置过同名字和 tag 的共享内存，所以也会新增一个共享内存节点到 ngx_cycle_t 对象的 shared_memory 中，但是这个时候该节点指定的共享内存大小是 0。然后解析到 limit_conn_zone 指令时，由于名字和 tag 一样会索引到刚刚创建的那块共享内存，并且会发现共享内存大小为 0，此时则会把 limit_conn_zone 指令指定的大小替换 0，并返回这块同乡内存。

这样就可以不用规定这两条命令出现的先后顺序，增加了配置文件的自由性。从上面的流程图可以看出，对于每一条 limit_conn 指令，Nginx 都会用如下的结构体进行抽象：

```
typedef struct {
    ngx_shm_zone_t   *shm_zone; // limit_conn 第一个参数指定的共享内存
    ngx_uint_t    conn;   // 一个 ip 同时可以发起的最大连接数
} ngx_stream_limit_conn_limit_t;
```

limit_conn 指令的解析结果会存储在这个对象中，并挂载到 ngx_stream_limit_conn_module 模块的 server 级别配置项中。另外，还有一点需要注意的就是，Nginx 允许一个 server 块内出现多条 limit_conn 指令，当然前提是这些指令使用不同的共享内存，这个时候就会以动态数组的方式来组织多条 limit_conn 指令的配置信息，这一点可以参见 ngx_stream_limit_conn_conf_t 的定义。

如果出现了多条 limit_conn 指令，则所有的 limit_conn 指令都会生效，并且会以并发连接数最小的那个配置来决定客户端的连接是否会被断开。举个例子，如果有如下配置:server { limit_conn zone0 2; limit_conn zone1 3; }。对于一个客户端来说，如果并发连接数大于 2 的时候，那么会由于 zone0 配置的限制而导致第三个开始的后续连接都会被断开。这点从后面的功能实现部分可以看出。

二、阶段介入

如果配置文件的 stream 块中配置了 limit_conn 和 limit_conn_zone 指令，在 Nginx 完成配置文件解析并提供 stream 四层反向代理服务后，客户端向 stream

块中的 server 发送请求时 Nginx 就会限制客户端的并发连接数了，那么 Nginx 是通过什么方式来介入到四层方向代理流程中的呢？在 stream 框架重要组成部分--ngx_stream_core_module 模块的 stream main 级别配置项结构体中可以发现 Nginx 的处理方式，该配置项结构体如下：

```
typedef struct {
    /*
     *servers 动态数组存放的是代表出现在 stream 块内的 server 块的
     *配置项结构体，在 ngx_stream_core_server 函数中会将生成的
     *ngx_stream_core_module 模块的 srv 级别配置项结构体添加到
     *这个动态数组中。
     */
    ngx_array_t                servers;         /* ngx_stream_core_srv_conf_t */

   /*
    * 存放的是 stream 块内所有 server 块内出现的 listen 指令的参数，
    * 一个 listen 对应其中的一个元素
    */
    ngx_array_t                listen;          /* ngx_stream_listen_t */

    /* stream limit conn 模块注册的处理函数 */
    ngx_stream_access_pt       limit_conn_handler;

    /* stream access 模块注册的处理函数 */
    ngx_stream_access_pt       access_handler;
} ngx_stream_core_main_conf_t;
```

从这个配置项结构体中可以发现 Nginx 会将 ngx_stream_limit_conn_module 模块处理函数挂载到 limit_conn_handler 字段中。那 Nginx 是什么时候把 ngx_stream_limit_conn_module 模块的处理函数注册到这里的呢，又是什么时候会调用这个函数呢？

先来看下 Nginx 什么时候会把 ngx_stream_limit_conn_module 模块的处理函数注册到 ngx_stream_core_main_conf_t 对象的 limit_conn_handler 中。来看下 ngx_stream_limit_conn_module 模块实现的 NGX_STREAM_MODULE 模块类型的接口 ngx_stream_module_t：

```
static ngx_stream_module_t  ngx_stream_limit_conn_module_ctx = {
    ngx_stream_limit_conn_init,          /* postconfiguration */

    NULL,                                /* create main configuration */
    NULL,                                /* init main configuration */

    ngx_stream_limit_conn_create_conf,   /* create server configuration */
    ngx_stream_limit_conn_merge_conf,    /* merge server configuration */
};
```

从上面这个结构体中我们可以看到，ngx_stream_limit_conn_module 模块注册了 postconfiguration 字段的回调函数，而这个回调函数正是在解析完配置文件

之后被调用的，该模块正是在这个回调函数中将其对应的处理函数挂载到了 ngx_stream_core_main_conf_t 对象的 limit_con_handler 中，这从函数 ngx_stream_limit_conn_init 就可以看到，函数实现如下：

```
static ngx_int_t
ngx_stream_limit_conn_init(ngx_conf_t *cf)
{
    ngx_stream_core_main_conf_t    *cmcf;

    cmcf = ngx_stream_conf_get_module_main_conf(cf,
                ngx_stream_core_module);

    cmcf->limit_conn_handler = ngx_stream_limit_conn_handler;

    return NGX_OK;
}
```

再来看下 Nginx 又是什么时候会调用 ngx_stream_limit_conn_module 模块的处理函数来实现限制客户端的并发连接数的。从一开始的时候有说该模块是介于 Pre-access 阶段的，根据这个阶段的定位，这个时候 Nginx 已经和客户端建立了连接，但是还没有提供服务，所以是在 ngx_stream_init_connection()函数中（该函数会注册给 ngx_listening_t 监听对象的 handler）调用的。limit_conn_handler 调用如下：

```
void
ngx_stream_init_connection(ngx_connection_t *c)
{
    ……
    /*
     * 如果注册了 ngx_stream_limit_conn_module 模块的处理方法，
     * 则会对来自同一个 ip 的并发连接数进行限制
     */
    if (cmcf->limit_conn_handler) {
        rc = cmcf->limit_conn_handler(s);

        if (rc != NGX_DECLINED) {
            ngx_stream_close_connection(c);
            return;
        }
    }
    ……
}
```

从这里可以看出如果 limit_conn_handler 返回的不是 NGX_DECLINED，则会结束与客户端之间的连接。

三、功能实现

通过阶段介入小节的介绍，我们已经知道 ngx_stream_limit_conn_module 模块是如何介入到 stream 子系统的处理流程中的。那么该模块又是如何根据解析

到的配置信息来实现限制客户端的并发连接数的呢？这个就是ngx_stream_limit_conn_handler 函数实现的了，其执行流程图如下：

```
static ngx_int_t
ngx_stream_limit_conn_handler(ngx_stream_session_t *s)
{
    size_t                        n;
    uint32_t                      hash;
    ngx_str_t                     key;
    ngx_uint_t                    i;
    ngx_slab_pool_t              *shpool;
    ngx_rbtree_node_t            *node;
    ngx_pool_cleanup_t           *cln;
    struct sockaddr_in           *sin;
    ngx_stream_limit_conn_ctx_t  *ctx;
    ngx_stream_limit_conn_node_t *lc;
    ngx_stream_limit_conn_conf_t *lccf;
    ngx_stream_limit_conn_limit_t *limits;

    /* 判断客户端和 nginx 之间的连接协议族 */
    switch (s->connection->sockaddr->sa_family) {

    case AF_INET:
        sin = (struct sockaddr_in *) s->connection->sockaddr;

        /* 获取客户端 ip 地址 */
        key.len = sizeof(in_addr_t);
        key.data = (u_char *) &sin->sin_addr;

        break;
     ……
    }

    /* 用 crc32 计算 ip 地址对应的 hash 值 */
    hash = ngx_crc32_short(key.data, key.len);

    lccf = ngx_stream_get_module_srv_conf(s, ngx_stream_limit_conn_module);
    limits = lccf->limits.elts;
    for (i = 0; i < lccf->limits.nelts; i++) {
        ctx = limits[i].shm_zone->data;
        ……
        /* 用 ip 地址 hash 值到红黑树中查找对应的节点 */
        node = ngx_stream_limit_conn_lookup(ctx->rbtree, &key, hash);

        /* 如果 node == NULL，说明该 ip 地址还没有发起过连接 */
```

```
if (node == NULL) {

    n = offsetof(ngx_rbtree_node_t, color)
        + offsetof(ngx_stream_limit_conn_node_t, data)
        + key.len;

    /*
     * 从共享内存中分配用于存储红黑树节点的内存。一个红黑树节
     * 点用来存储同一个 ip 对应的多个连接之间共有的状态信息。
     */
    node = ngx_slab_alloc_locked(shpool, n);
     /*
     * 申请内存失败，说明达到了共享内存支持的最大 ip 个数，
     * 返回 NGX_ABORT，主流程会结束 Nginx 与这个客户端连接
     */
    if (node == NULL) {
        ……
        return NGX_ABORT;
    }
    lc = (ngx_stream_limit_conn_node_t *) &node->color;

    node->key = hash;  // 红黑树节点的 key 值为客户端 ip 的 hash 值
    lc->len = (u_char) key.len;  // 记录客户端 ip 地址长度
    lc->conn = 1;  // 该 ip 地址首次连接，conn 置为 1
    ngx_memcpy(lc->data, key.data, key.len);  // 记录客户端 ip 地址

    ngx_rbtree_insert(ctx->rbtree, node); //将当前节点加入到红黑树中

} else {
    lc = (ngx_stream_limit_conn_node_t *) &node->color;

    /*
     * 判断 ip 地址已经发起的连接总数是否达到了限制的阈值，
     * 如果达到了，返回 NGX_ABORT，断开此次连接
     */
    if ((ngx_uint_t) lc->conn >= limits[i].conn) {
        ……
        return NGX_ABORT;
    }

    /*
     * 程序执行到这里表明当前 ip 已经发起的连接数还没有
     * 达到限制的阈值，递增已连接数，然后做后续处理
     */
```

```
            lc->conn++;
        }
        ……
    }
    /* 返回 NGX_DECLINED，则主流程接着往下处理 */
    return NGX_DECLINED;
}
```

从上面的流程中，可以清楚地看到其功能逻辑：

1、 如果配置文件中没有配置 limit_conn_zone 和 limit_conn 命令，则不会对客户端 ip 进行并发连接数的限制。这个从 ngx_stream_init_connection()中可以看到。

2、 遍历配置文件中配置的所有 limit_conn 指令的规则，对于每一条规则，如果某个客户端 ip 第一次和 Nginx 建立连接，则会从共享内存中申请内存，存放已经建立的连接数（此时为 1）以及客户端地址信息；如果某个客户端 ip 之前已经和 Nginx 建立过连接，则会判断已经建立的连接数是否达到了配置的阈值，如果达到了，则返回 NGX_ABORT，在主流程 ngx_stream_init_connection()函数中就会断开与客户端的连接。如果没有达到配置阈值，则会递增已建立连接数，并返回 NGX_DECLINED，主流程则会继续往后续阶段执行。只要有一条规则不符合条件，则会导致 Nginx 断开与客户端的连接。