NGINX 数据结构设计

内存池

为什么需要内存池

- 提升内存分配效率。虽然系统自带的ptmalloc内存分配管理器，也有自己的内存优化管理方案（申请内存块以及将内存交还给系统都有自己的优化方案，具体可以研究一下ptmalloc的源码），但是直接使用malloc/alloc/free，仍然会导致内存分配的性能比较低。
- 降低内存碎片。频繁使用这些函数分配和释放内存，会导致内存碎片，不容易让系统直接回收内存。典型的例子就是大并发频繁分配和回收内存，会导致进程的内存产生碎片，并且不会立马被系统回收。
- 防止内存泄漏。让内存的管理变得更加简单。内存的分配都会在一块大的内存上，回收的时候只需要回收大块内存就能将所有的内存回收，防止了内存管理混乱和内存泄露问题。

需要解决的问题点

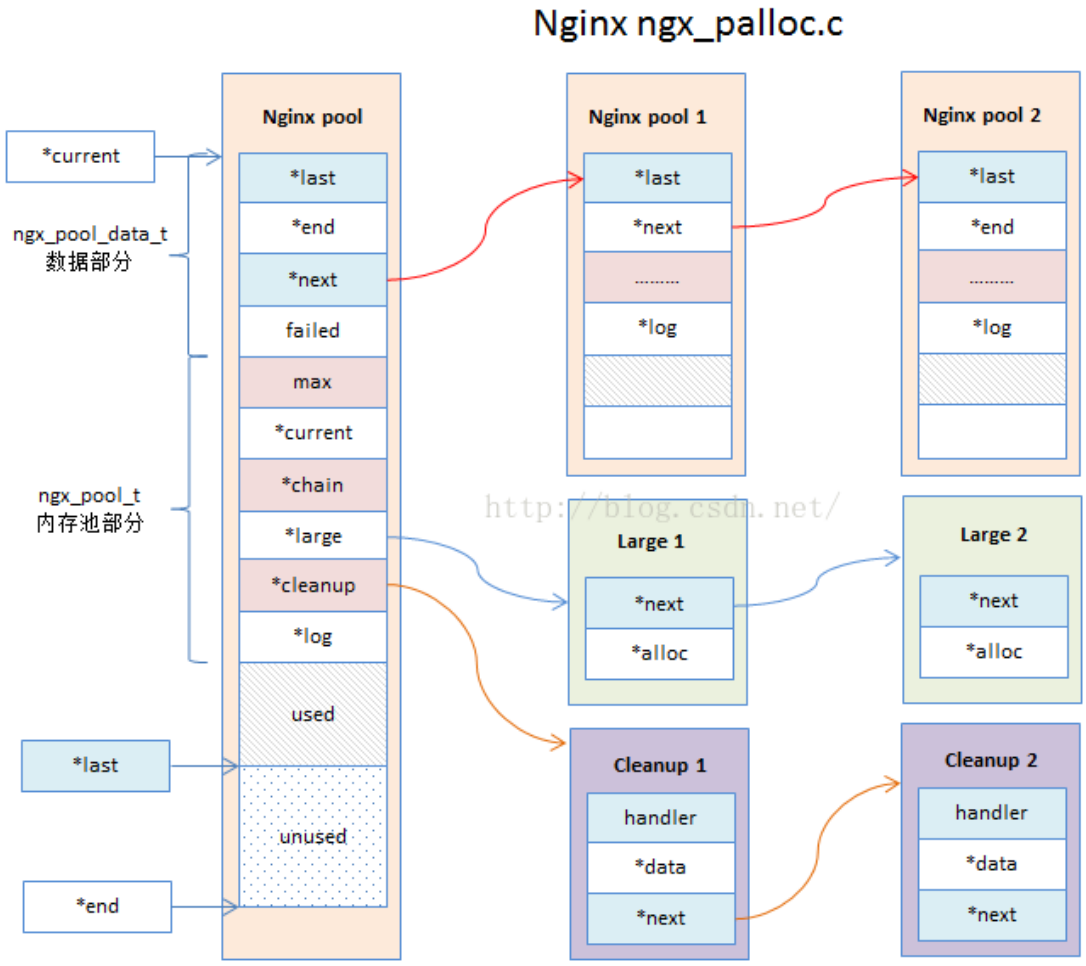
base

- 申请任意大小的内存
- 释放任意大小的内存

improve

- 提升内存分配效率
- 支持文件句柄fd的资源的存储
- 解决内存泄漏问题

内存池数据结构图



内存池实现

分配内存

内存分配根据max(小块内存最大值)来决定走大内存分配还是小内存分配。

- 大块内存分配

```
/**
* 当分配的内存块大小超出pool->max限制的时候,需要分配在pool->large上
*/
static void *
ngx_palloc_large(ngx_pool_t *pool, size_t size) {
  void *p;
  ngx_uint_t n;
  ngx_pool_large_t *large;

  /* 分配一块新的大内存块 */
  p = ngx_alloc(size, pool->log);
  if (p == NULL) {
      return NULL;
  }
```

```
    n = 0;

    /* 去pool->large链表上查询是否有NULL的，只在链表上往下查询3次，主要判断大数
据块是否有被释放的，如果没有则只能跳出*/
    for (large = pool->large; large; large = large->next) {
        if (large->alloc == NULL) {
            large->alloc = p;
            return p;
        }

        if (n++ > 3) {
            break;
        }
    }

    /* 分配一个ngx_pool_large_t 数据结构 */
    large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
    if (large == NULL) {
        ngx_free(p); //如果分配失败，删除内存块
        return NULL;
    }

    large->alloc = p;
    large->next = pool->large;
    pool->large = large;

    return p;
  }
```

- 小块内存分配

```
  void *
  ngx_pnalloc(ngx_pool_t *pool, size_t size) v
    u_char *m;
    ngx_pool_t *p;

    /* 判断每次分配的内存大小，如果超出pool->max的限制，则需要走大数据内存分配策
略 */
    if (size <= pool->max) {
        //小内存块分配的起点是current 内存池，这个是根据failed参数决定的。
        p = pool->current;

        /* 循环读取数据区域的各个ngx_pool_t缓存池链，如果剩余的空间可以容纳
size，则返回指针地址*/
        do {
            m = p->d.last; //分配的内存块的地址

            if ((size_t)(p->d.end - m) >= size) {
                p->d.last = m + size;
```

```
                return m;
            }

            p = p->d.next;

        } while (p);

        /* 如果没有缓存池空间没有可以容纳大小为size的内存块，则需要重新申请一个缓存池*/
        return ngx_palloc_block(pool, size);
    }

    /* 走大数据分配策略 */
    return ngx_palloc_large(pool, size);
  }
```

回收内存

- 大块内存回收

```
  /**
  * 大内存块释放  pool->large
  */
  ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p) {
    ngx_pool_large_t *l;

    /* 在pool->large链上循环搜索，并且只释放内容区域，不释放ngx_pool_large_t数据结构*/
    for (l = pool->large; l; l = l->next) {
        if (p == l->alloc) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                    "free: %p", l->alloc);
            ngx_free(l->alloc);
            l->alloc = NULL;

            return NGX_OK;
        }
    }

    return NGX_DECLINED;
  }
```

- 小块内存回收
    - 小内存块并不会回收，所以会导致小块内存随着分配越来越大，系统的内存肯定是分配失败。这里通过销毁内存池和充值内存池进行回收。 /** * 销毁内存池。 */ void

ngx_destroy_pool(ngx_pool_t *pool) { ngx_pool_t *p, *n; ngx_pool_large_t *l; ngx_pool_cleanup_t *c;

```
    /* 首先清理pool->cleanup链表 */
    for (c = pool->cleanup; c; c = c->next) {
        /* handler 为一个清理的回调函数 */
        if (c->handler) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                    "run cleanup: %p", c);
            c->handler(c->data);
        }
    }

    /* 清理pool->large链表（pool->large为单独的大数据内存块）  */
    for (l = pool->large; l; l = l->next) {

        ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free:
%p", l->alloc);

        if (l->alloc) {
            ngx_free(l->alloc);
        }
    }

  #if (NGX_DEBUG)

    /*
    * we could allocate the pool->log from this pool
    * so we cannot use this log while free()ing the pool
    */

    for (p = pool, n = pool->d.next; /* void */; p = n, n = n-
>d.next) {
        ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                "free: %p, unused: %uz", p, p->d.end - p-
>d.last);

        if (n == NULL) {
            break;
        }
    }

  #endif

    /* 对内存池的data数据区域进行释放 */
    for (p = pool, n = pool->d.next; /* void */; p = n, n = n-
>d.next) {
        ngx_free(p);

        if (n == NULL) {
```

```
            break;
        }
    }
}

/**
* 重设内存池
*/
void ngx_reset_pool(ngx_pool_t *pool) {
    ngx_pool_t *p;
    ngx_pool_large_t *l;

    /* 清理pool->large链表（pool->large为单独的大数据内存块）  */
    for (l = pool->large; l; l = l->next) {
        if (l->alloc) {
            ngx_free(l->alloc);
        }
    }

    pool->large = NULL;

    /* 循环重新设置内存池data区域的 p->d.last；data区域数据并不擦除*/
    for (p = pool; p; p = p->d.next) {
        p->d.last = (u_char *) p + sizeof(ngx_pool_t);
    }
}
```

文件句柄fd等资源管理

文件句柄fd资源管理是使用cleanup机制解决的。pool->cleanup本身是一个链表，每个ngx_pool_cleanup_t的数据结构上，保存着内存数据的本身cleanup->data和回调清理函数cleanup->handler。

- 分配一个cleanup结构

```
/**
* 分配一个可以用于回调函数清理内存块的内存
* 内存块仍旧在p->d或p->large上
*
* ngx_pool_t中的cleanup字段管理着一个特殊的链表，该链表的每一项都记录着一个特殊的需要释放的资源。
* 对于这个链表中每个节点所包含的资源如何去释放，是自说明的。这也就提供了非常大的灵活性。
* 意味着，ngx_pool_t不仅仅可以管理内存，通过这个机制，也可以管理任何需要释放的资源，
* 例如，关闭文件，或者删除文件等等的。下面我们看一下这个链表每个节点的类型
*
* 一般分两种情况：
* 1. 文件描述符
* 2. 外部自定义回调函数可以来清理内存
```

```
  */
  ngx_pool_cleanup_t *
  ngx_pool_cleanup_add(ngx_pool_t *p, size_t size) {
    ngx_pool_cleanup_t *c;
    /* 分配一个ngx_pool_cleanup_t */
    c = ngx_palloc(p, sizeof(ngx_pool_cleanup_t));
    if (c == NULL) {
        return NULL;
    }

    /* 如果size !=0 从pool->d或pool->large分配一个内存块 */
    if (size) {
        /*  */
        c->data = ngx_palloc(p, size);
        if (c->data == NULL) {
            return NULL;
        }

    } else {
        c->data = NULL;
    }

    /* handler为回调函数 */
    c->handler = NULL;
    c->next = p->cleanup;

    p->cleanup = c;

    ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, p->log, 0, "add cleanup: %p",
c);

    return c;
  }
```

- 回收一个cleanup内存

```
  /**
  * 清除 p->cleanup链表上的内存块（主要是文件描述符）
  * 回调函数：ngx_pool_cleanup_file
  */
  void ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd) {
    ngx_pool_cleanup_t *c;
    ngx_pool_cleanup_file_t *cf;

    for (c = p->cleanup; c; c = c->next) {
        if (c->handler == ngx_pool_cleanup_file) {

            cf = c->data;
```

```
            if (cf->fd == fd) {
                c->handler(cf); /* 调用回调函数 */
                c->handler = NULL;
                return;
            }
        }
    }
}
```

缓冲区

Nginx的buf缓冲区数据结构，主要用来存储非常大块的内存。ngx_buf_t数据结构也贯穿了整个Nginx。Nginx的缓冲区设计是比较灵活的。

- 可以自定义管理业务层面的缓冲区链表；
- 可以将空闲的缓冲区链表交还给内存池pool->chain结构。缓冲区ngx_buf_t是nginx处理大数据的关键数据结构，它既应用于内存数据也应用于磁盘数据。如果将缓冲区放在pool->chain结构中，后续缓冲区的回收可让内存池代理。

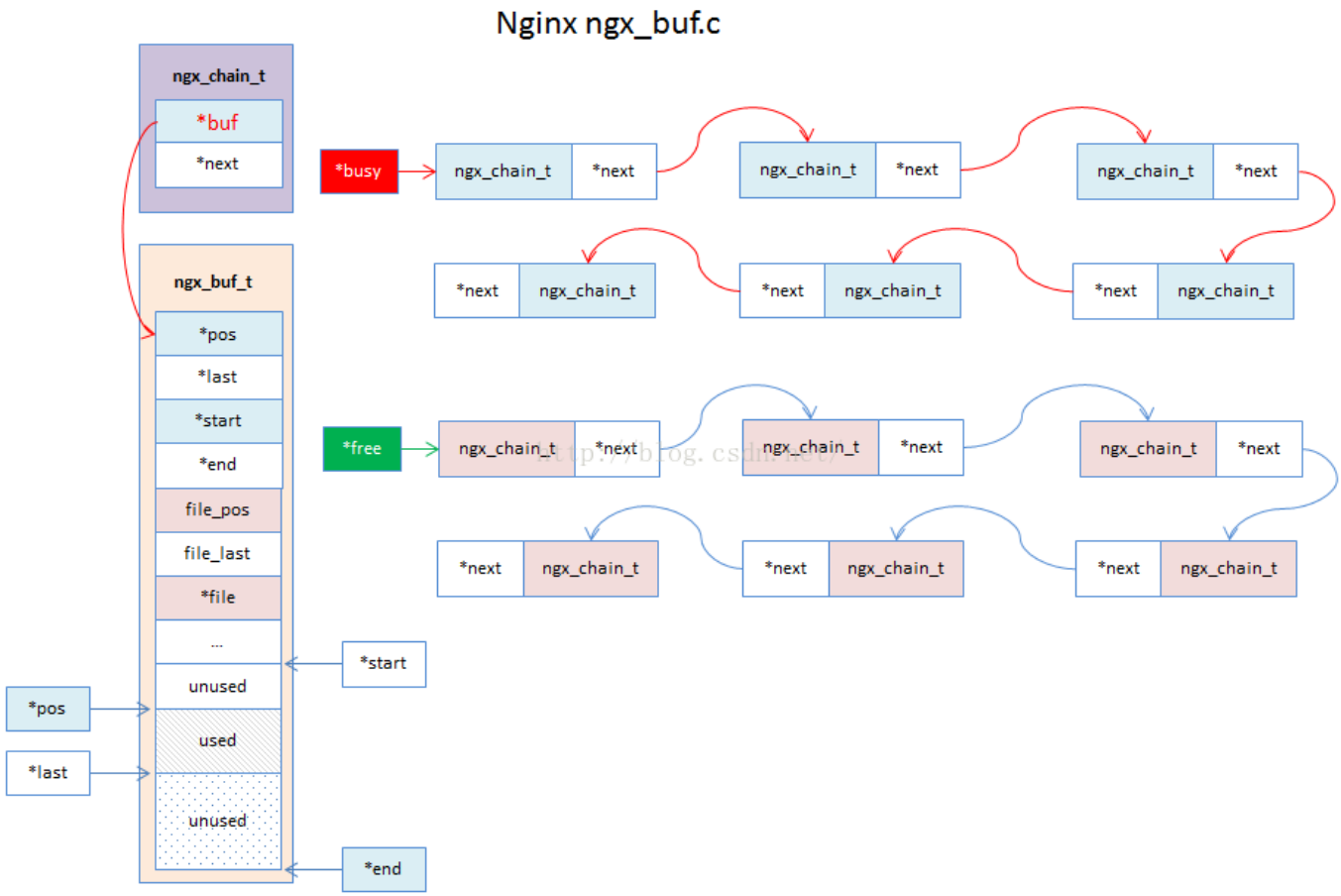
为什么需要缓冲区

- 可以解除两者的制约关系，数据可以直接送往缓冲区，高速设备不用再等待低速设备，提高了计算机的效率。例如：我们使用打印机打印文档，由于打印机的打印速度相对较慢，我们先把文档输出到打印机相应的缓冲区，打印机再自行逐步打印，这时我们的CPU可以处理别的事情。

- 可以减少数据的读写次数，如果每次数据只传输一点数据，就需要传送很多次，这样会浪费很多时间，因为开始读写与终止读写所需要的时间很长，如果将数据送往缓冲区，待缓冲区满后再进行传送会大大减少读写次数，这样就可以节省很多时间。例如：我们想将数据写入到磁盘中，不是立马将数据写到磁盘中，而是先输入缓冲区中，当缓冲区满了以后，再将数据写入到磁盘中，这样就可以减少磁盘的读写次数，不然磁盘很容易坏掉。

- 简单来说，缓冲区就是一块内存区，它用在输入输出设备和CPU之间，用来存储数据。它使得低速的输入输出设备和高速的CPU能够协调工作，避免低速的输入输出设备占用CPU，解放出CPU，使其能够高效率工作。

缓冲区数据结构



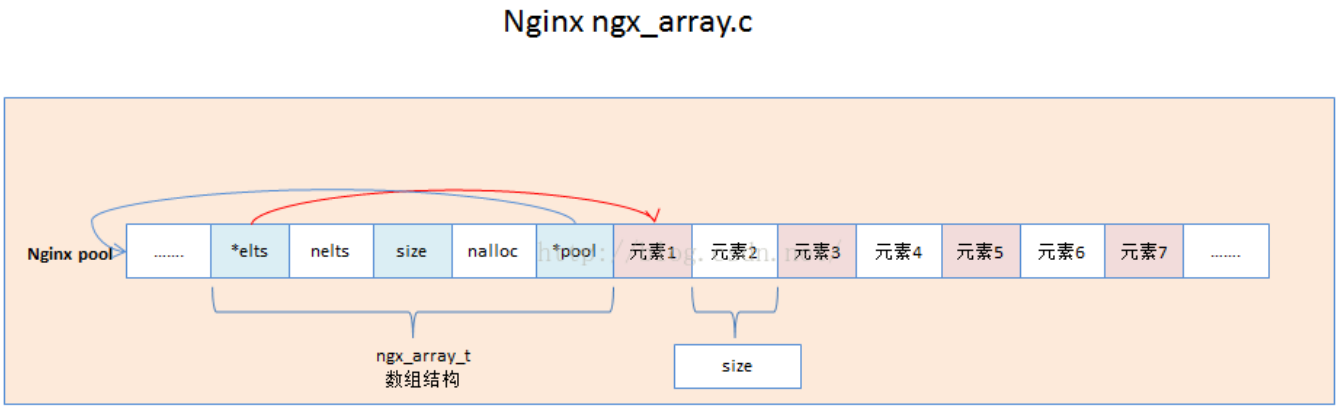
数组

Nginx的Array结构设计得非常小巧，主要用于存储小块内存。Nginx的数组每个元素的大小是固定的。

基础结构

```
/* 数组Array数据结构 */
typedef struct {
    void        *elts;      /* 指向数组第一个元素指针*/
    ngx_uint_t   nelts;     /* 未使用元素的索引*/
    size_t       size;      /* 每个元素的大小，元素大小固定*/
    ngx_uint_t   nalloc;    /* 分配多少个元素 */
    ngx_pool_t  *pool;      /* 内存池*/
} ngx_array_t;
```

数组数据结构



扩容机制

扩容有两种方式

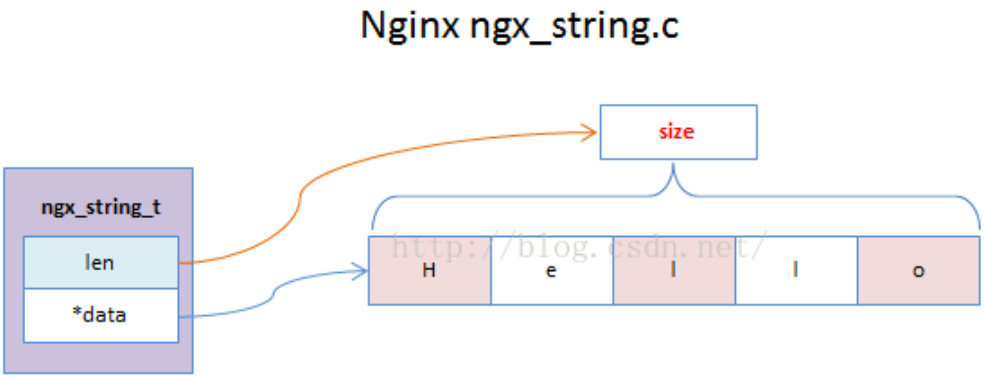
- 如果数组元素的末尾和内存池pool的可用开始的地址相同，并且内存池剩余的空间支持数组扩容，则在当前内存池上扩容
- 如果扩容的大小超出了当前内存池剩余的容量或者数组元素的末尾和内存池pool的可用开始的地址不相同，则需要重新分配一个新的内存块存储数组，并且将原数组拷贝到新的地址上。

字符串结构

字符串基本定义

```
/**
* 字符串结构
*/
typedef struct {
    size_t      len; //字符串长度
    u_char     *data; //具体的指针地址
} ngx_str_t;
```

字符串数据结构



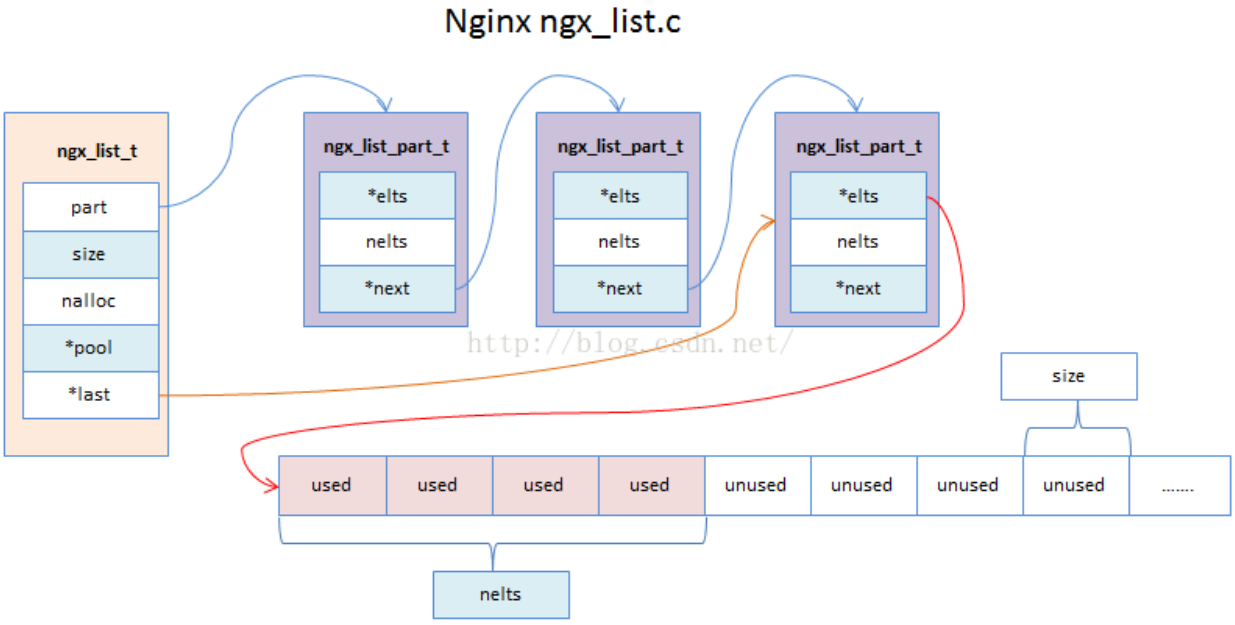
基本操作

```
//初始化一个字符串
#define ngx_string(str)     { sizeof(str) - 1, (u_char *) str }

//将一个字符串设置为NULL
#define ngx_null_string     { 0, NULL }

//设置一个字符串
#define ngx_str_set(str, text)
(str)->len = sizeof(text) - 1; (str)->data = (u_char *) text

#define ngx_str_null(str)
(str)->len = 0; (str)->data = NULL
```

单向链表

Nginx的list单向链表的结构和Nginx的数组结构Array有点类似，总体来说，数据结构也是非常简单清晰的。Nginx的单向链表也是固定了每个元素的大小，并且用单向链表的方式连接。

单向链表数据结构



双向链表

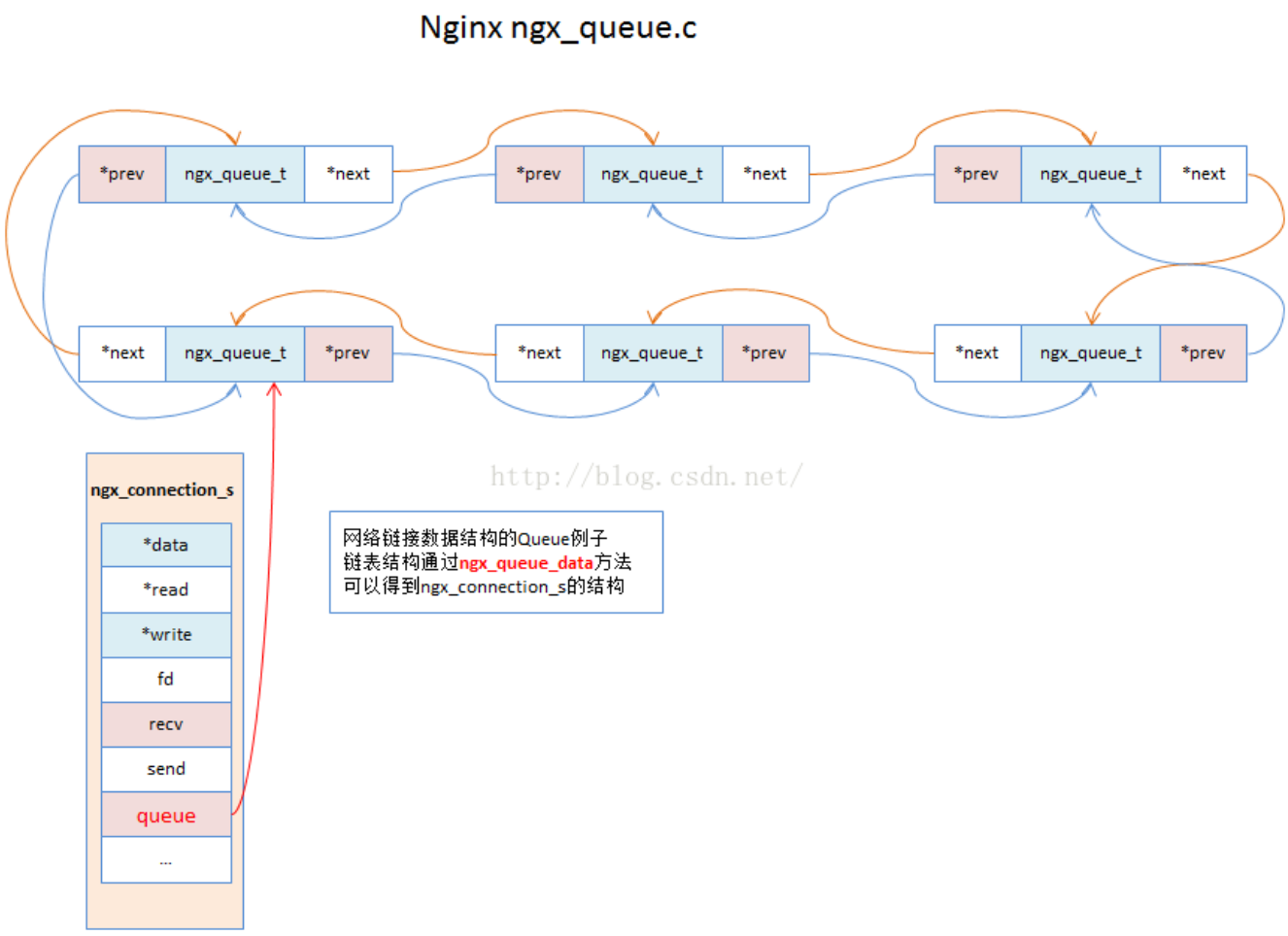
Nginx的链表结构非常小巧和简单。设计的非常精巧。通过链表的简单和精巧的设计，让Nginx的链表的数据结构和具体业务依赖进行了解耦。一般我们在设计c语言程序的时候，完全可以学习Nginx的这种数据结构的设计方式。

双向链表的基础结构

```
typedef struct ngx_queue_s ngx_queue_t;

/**
* 链表的数据结构非常简单，ngx_queue_s会挂载到实体
* 结构上。然后通过ngx_queue_s来做成链表
*/
struct ngx_queue_s {
    ngx_queue_t *prev;
    ngx_queue_t *next;
};

/**
* 该结构体用于描述一个网络连接
*/
struct ngx_connection_s {
    void *data; //连接未使用时，data用于充当连接池中空闲链表中的next指针。连接使用时由模块而定，HTTP中，data指向ngx_http_request_t
    ngx_event_t *read; //连接对应的读事件
    ngx_event_t *write; //连接对应的写事件

    ngx_socket_t fd; //套接字句柄

    ngx_recv_pt recv; //直接接受网络字节流
```

```
    ngx_send_pt send; //直接发送网络字节流
    ngx_recv_chain_pt recv_chain; //网络字节流接收链表
    ngx_send_chain_pt send_chain; //网络字节流发送链表

    /*用来将当前连接以双向链表元素的形式添加到ngx_cycle_t核心结构体
    * 的reuseable_connection_queue双向链表中，表示可以重用的连接*/
    ngx_queue_t queue;

    /* 省去部分 */
};
```

双向链表数据结构



核心操作

```
小技巧:
&((type)0)->xxx 结构体地址为0时，成员的绝对地址等于相对地址。
顺便提一下，空指针 只要你不解引用，是可以用的，这里取的是成员的地址，而不是成员的值，如果
是取空指针指向的成员值，程序肯定挂了
/**
* 通过链表可以找到结构体所在的指针
* typedef struct {
*       ngx_queue_s queue;
```

```
*       char * x;
*       ....
* } TYPE
* 例如：ngx_queue_data(&type->queue, TYPE, queue)
*/
#define ngx_queue_data(q, type, link)   (type *) ((u_char *) q -
offsetof(type, link))
```

基本操作

```
/**
* 初始化一个Q
*/
#define ngx_queue_init(q)
    (q)->prev = q;
    (q)->next = q
/**
* 判断是否是空Q
*/
#define ngx_queue_empty(h)
    (h == (h)->prev)

/**
* 向链表H后面插入一个x的Q，支持中间插入
*/
#define ngx_queue_insert_head(h, x)
    (x)->next = (h)->next;
    (x)->next->prev = x;
    (x)->prev = h;
    (h)->next = x

#define ngx_queue_insert_after   ngx_queue_insert_head

/**
* 向链表H前面插入一个x的Q，支持中间插入
*/
#define ngx_queue_insert_tail(h, x)
    (x)->prev = (h)->prev;
    (x)->prev->next = x;
    (x)->next = h;
    (h)->prev = x

/**
* h是尾部，链表的第一个元素
*/
#define ngx_queue_head(h)
    (h)->next

// h 是头，h 的上一个就是尾
```

```
#define ngx_queue_last(h)
    (h)->prev

#define ngx_queue_sentinel(h)
    (h)

/**
* 返回节点Q的下一个元素
*/
#define ngx_queue_next(q)
    (q)->next

/**
* 返回节点Q的上一个元素
*/
#define ngx_queue_prev(q)
    (q)->prev

#if (NGX_DEBUG)

/**
* 移除某一个节点
*/
#define ngx_queue_remove(x)
    (x)->next->prev = (x)->prev;
    (x)->prev->next = (x)->next;
    (x)->prev = NULL;
    (x)->next = NULL

#else

#define ngx_queue_remove(x)
    (x)->next->prev = (x)->prev;
    (x)->prev->next = (x)->next

#endif

//分割一个链表
#define ngx_queue_split(h, q, n)
    (n)->prev = (h)->prev;
    (n)->prev->next = n;
    (n)->next = q;
    (h)->prev = (q)->prev;
    (h)->prev->next = h;
    (q)->prev = n;


#define ngx_queue_add(h, n)
    (h)->prev->next = (n)->next;
    (n)->next->prev = (h)->prev;
```

```
    (h)->prev = (n)->prev;
    (h)->prev->next = h;
```

hash表

Nginx的hash表结构主要几个特点：

- 静态只读。当初始化生成hash表结构后，是不能动态修改这个hash表结构的内容。
- 将内存利用最大化。Nginx的hash表，将内存利用率发挥到了极致，并且很多设计上面都是可以供我们学习和参考的。
- 查询速度快。Nginx的hash表做了内存对齐等优化。
- 主要解析配置数据。

抽象成一个需要解决的问题

由于nginx的hash表结构是静态只读的，所以可以将nginx的hash表变成一个问题。已知有一个key数组和一个value数组，其中数据根据下标一一对应，让你设计一个hash结构具备以下能力：

- 查询速度快
- 内存利用最大化

hash表基本定义

- ngx_hash_elt_t hash表的元素结构

```
  /**
  * 存储hash的元素
  */
  typedef struct {
    void             *value;    /* 指向value的指针 */
    u_short           len;      /* key的长度 */
    u_char            name[1];  /* 指向key的第一个地址，key长度为变长(设计上
的亮点)*/
  } ngx_hash_elt_t;
```

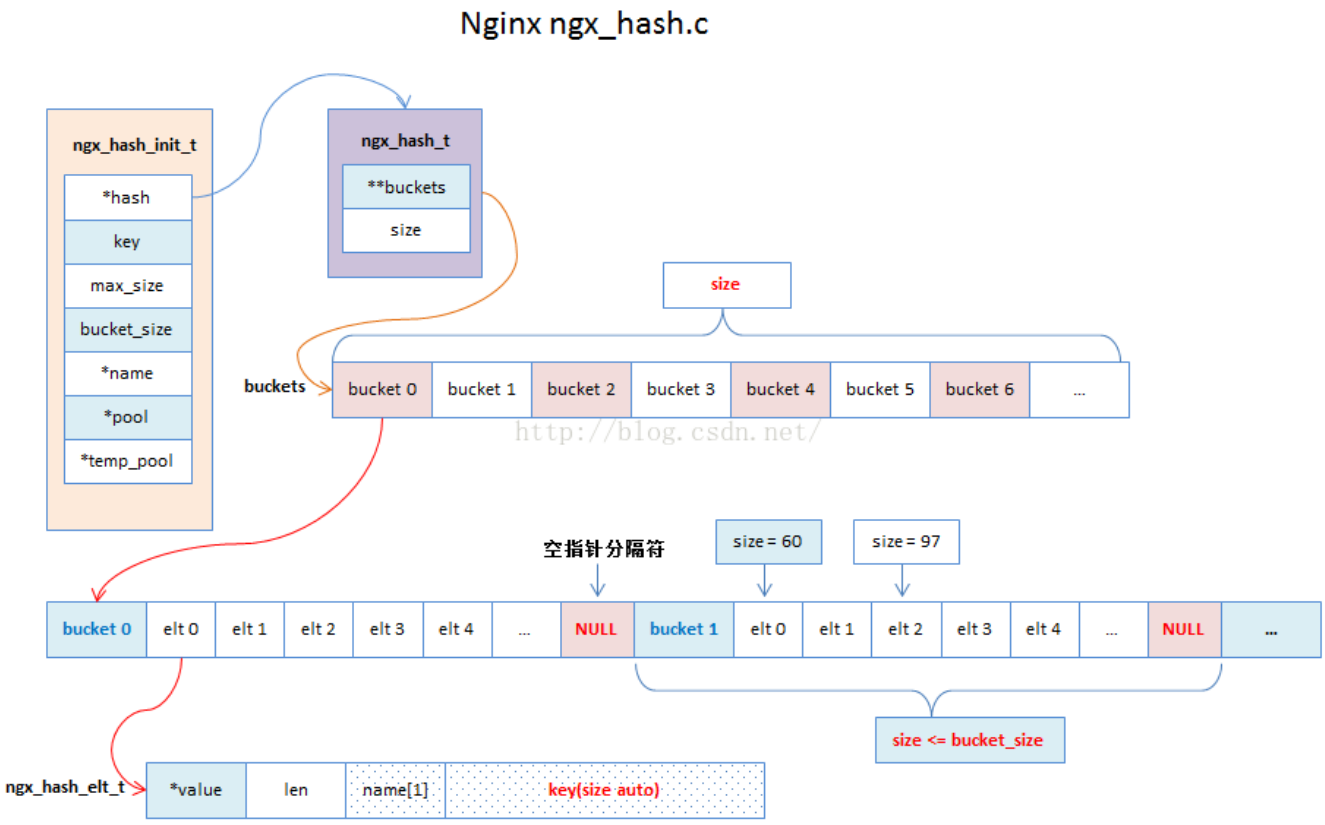
- ngx_hash_t hash表结构

```
  /**
  * Hash的桶
  */
  typedef struct {
    ngx_hash_elt_t  **buckets;  /* hash表的桶指针地址值 */
    ngx_uint_t        size;     /* hash表的桶的个数*/
  } ngx_hash_t;
```

- ngx_hash_init_t hash表初始化结构

```
/**    hash表主体结构       */
typedef struct {
  ngx_hash_t       *hash; /* 指向hash数组结构 */
  ngx_hash_key_pt   key;  /* 计算key散列的方法 */
  ngx_uint_t        max_size;     /* 最大多少个 */
  ngx_uint_t        bucket_size;  /* 桶的存储空间大小 */
  char             *name; /* hash表名称 */
  ngx_pool_t       *pool; /* 内存池 */
  ngx_pool_t       *temp_pool; /* 临时内存池*/
} ngx_hash_init_t;
```

hash 表数据结构



hash 表的实现跟普通的hash差不多,主要做了一些优化

不定长key

- 目的: 节省内存

内存对齐

- 目的: 查询快速

内存对齐方案：ngx_align_ptr 方法, 其中p是开始地址，a是页面大小。

```
#define ngx_align_ptr(p, a) \
    (u_char*)(((uintptr_t)(p) + ((uintptr_t)a - 1)) & ~((uintptr_t)a - 1))
```

这里有两个知识点(高效算法的奥秘.pdf)：

向下调整： $x\ \&((-1)\ll k)$

$(x\overset{n}{\gg}k)\ll k$

向上调整： $t\leftarrow(1\ll k)-1;(x+t)\&\neg t$

$t\leftarrow(-1)\ll k;(x-t-1)\&t$

1. 找到一个数，大于等于p，而且是a的最小倍数（其中a是2的幂次）。

```
 (p+(a-1))&~(a-1) 2
```

2. 找到一个数，小于等于p，而且是a的最大倍数（其中a是2的幂次）。

```
  p&(-a) 或者 (x>>log2a)<<log2a
```

常用操作

1. 构造hash

- 计算bucket的个数
- 计算内存的总大小(需要计算变长元素的大小)
- 分配一整块内存
- 将数据分配上去

```
   /**
   * 获取元素的大小
   * 元素大小主要是ngx_hash_elt_t结构，包括:
   * 1. name的长度    (name)->key.len
   * 2. len的长度 其中的"+2"是要加上该结构中len字段(u_short类型)的大小
   * 3. value指针的长度    "sizeof(void *)"相当于 value的长度
   */
   #define NGX_HASH_ELT_SIZE(name)
\
     (sizeof(void *) + ngx_align((name)->key.len + 2, sizeof(void *)))


   /**
   * 初始化一个hash表
```

```
  */
  ngx_int_t ngx_hash_init(ngx_hash_init_t *hinit, ngx_hash_key_t
*names,
        ngx_uint_t nelts)
  {
    u_char *elts;
    size_t len;
    u_short *test;
    ngx_uint_t i, n, key, size, start, bucket_size;
    ngx_hash_elt_t *elt, **buckets;

    /**
    * 先检查每个元素是否会超过bucket_size的限制
    * 如果超过限制，则说明需要重新处理
    * hash表的每一个bucket桶中的元素elt都是被分配到一块完整的内存块上的，
    * 每个bucket的内存块结尾会有一个void *的空指针作为表示符号用于分隔bucket
    */
    for (n = 0; n < nelts; n++) {
        if (hinit->bucket_size
                < NGX_HASH_ELT_SIZE(&names[n]) + sizeof(void *)) {
            ngx_log_error(NGX_LOG_EMERG, hinit->pool->log, 0,
                    "could not build the %s, you should "
                            "increase %s_bucket_size: %i", hinit-
>name,
                    hinit->name, hinit->bucket_size);
            return NGX_ERROR;
        }
    }

    /*
    * test是用来做探测用的，探测的目标是在当前bucket的数量下，冲突发生的是否频
繁。
    * 过于频繁则需要调整桶的个数。
    * 检查是否频繁的标准是：判断元素总长度和bucket桶的容量bucket_size做比较
    */
    test = ngx_alloc(hinit->max_size * sizeof(u_short), hinit->pool-
>log);
    if (test == NULL) {
        return NGX_ERROR;
    }

    /**
    * 每个桶的元素实际所能容纳的空间大小
    * 需要减去尾部的NULL指针结尾符号
    */
    bucket_size = hinit->bucket_size - sizeof(void *);

    /**
    * 通过一定的小算法，计算得到从哪个桶开始test（探测）
    */
    start = nelts / (bucket_size / (2 * sizeof(void *)));
```

```
    start = start ? start : 1;

    if (hinit->max_size > 10000 && nelts && hinit->max_size / nelts <
100) {
        start = hinit->max_size - 1000;
    }

    /**
    * 这边就是真正的探测逻辑
    * 探测会遍历所有的元素，并且计算落到同一个bucket上元素长度的总和和
bucket_size比较
    * 如果超过了bucket_size，则说明需要调整
    * 最终会探测出比较合适的桶的个数 ： size
    */
    for (size = start; size < hinit->max_size; size++) {

        ngx_memzero(test, size * sizeof(u_short));

        for (n = 0; n < nelts; n++) {
            if (names[n].key.data == NULL) {
                continue;
            }

            key = names[n].key_hash % size;
            test[key] = (u_short)(test[key] +
NGX_HASH_ELT_SIZE(&names[n]));

  #if 0
            ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
                    "%ui: %ui %ui \"%V\"",
                    size, key, test[key], &names[n].key);
  #endif

            /* 比较bucket_size和落到该bucket上的元素长度总和*/
            if (test[key] > (u_short) bucket_size) {
                goto next;
            }
        }

        goto found;

        next:

        continue;
    }

    ngx_log_error(NGX_LOG_EMERG, hinit->pool->log, 0,
            "could not build the %s, you should increase "
                    "either %s_max_size: %i or %s_bucket_size: %i",
hinit->name,
            hinit->name, hinit->max_size, hinit->name, hinit-
```

```
>bucket_size);

    ngx_free(test);

    return NGX_ERROR;

    /**
    * 探测成功，则size为bucket桶的个数
    */
    found:

    /**
    * 为了确定bucket的实际长度，初始化每个桶的长度计数器，初始值为一个NULL空指针
长度
    * 前面说过，每个bucket的内存块之间，使用一个NULL空指针进行分割，所以长度需要
加上去
    */
    for (i = 0; i < size; i++) {
        test[i] = sizeof(void *);
    }

    /**
    * 通过遍历，计算每个桶的大小。并且将每个桶的大小存储在test[n]数组上
    */
    for (n = 0; n < nelts; n++) {
        if (names[n].key.data == NULL) {
            continue;
        }

        key = names[n].key_hash % size;
        test[key] = (u_short)(test[key] +
NGX_HASH_ELT_SIZE(&names[n]));
    }

    len = 0;

    /**
    * 获取所有元素需要分配的内存的总大小
    * len = 总的内存大小，所有的桶都会放在一块内存上，并且做了手工内存对齐
    */
    for (i = 0; i < size; i++) {
        if (test[i] == sizeof(void *)) {
            continue;
        }

        /* 总内存大小，需要通过内存对齐函数 */
        test[i] = (u_short)(ngx_align(test[i], ngx_cacheline_size));

        len += test[i];
    }
```

```
    /**
    * 分配一块内存空间，存储：ngx_hash_t *hash和ngx_hash_elt_t *
    * ngx_hash_elt_t用于存储桶。指针指向元素地址
    */
    if (hinit->hash == NULL) {
        hinit->hash = ngx_pcalloc(hinit->pool,
                sizeof(ngx_hash_wildcard_t) + size *
sizeof(ngx_hash_elt_t *));
        if (hinit->hash == NULL) {
            ngx_free(test);
            return NGX_ERROR;
        }

        buckets = (ngx_hash_elt_t **) ((u_char *) hinit->hash
                + sizeof(ngx_hash_wildcard_t));

    } else {
        buckets = ngx_pcalloc(hinit->pool, size *
sizeof(ngx_hash_elt_t *));
        if (buckets == NULL) {
            ngx_free(test);
            return NGX_ERROR;
        }
    }

    /**
    * 分配一个桶，用于存储所有元素数据
    */
    elts = ngx_palloc(hinit->pool, len + ngx_cacheline_size);
    if (elts == NULL) {
        ngx_free(test);
        return NGX_ERROR;
    }

    elts = ngx_align_ptr(elts, ngx_cacheline_size); //内存对齐

    /**
    * 将elts元素的内存，分割到buckets桶上
    */
    for (i = 0; i < size; i++) {
        if (test[i] == sizeof(void *)) {
            continue;
        }

        buckets[i] = (ngx_hash_elt_t *) elts;
        elts += test[i];

    }

    /**
    * 将test清空，利用test于元素填充计数器
```

```
    */
    for (i = 0; i < size; i++) {
        test[i] = 0;
    }

    /**
    * 往bucket的元素位上填充数据
    */
    for (n = 0; n < nelts; n++) {
        if (names[n].key.data == NULL) {
            continue;
        }

        /* 计算在哪个桶上 */
        key = names[n].key_hash % size;
        elt = (ngx_hash_elt_t *) ((u_char *) buckets[key] +
test[key]);

        elt->value = names[n].value;
        elt->len = (u_short) names[n].key.len;

        /* 拷贝key数据,并且小写 */
        ngx_strlow(elt->name, names[n].key.data, names[n].key.len);

        /* test计数器计算新元素需要存放的位置 */
        test[key] = (u_short)(test[key] +
NGX_HASH_ELT_SIZE(&names[n]));
    }

    /**
    *  设置bucket桶上最后一个元素设置为value为NULL
    */
    for (i = 0; i < size; i++) {
        if (buckets[i] == NULL) {
            continue;
        }
        /**
        * 这边的设计 Nice！！！
        * test[i] 其实是bucket的元素块的结束位置
        * 由于前面bucket的处理中多留出了一个指针的空间，而此时的test[i]是
bucket中实际数据的共长度，
        * 所以bucket[i] + test[i]正好指向了末尾null指针所在的位置。处理的时
候，把它当成一个ngx_hash_elt_t结构看，
        * 在该结构中的第一个元素，正好是一个void指针，我们只处理它，别的都不去
碰，所以没有越界的问题。
        */
        elt = (ngx_hash_elt_t *) ((u_char *) buckets[i] + test[i]);

        elt->value = NULL;
    }
```

```
    ngx_free(test);

    hinit->hash->buckets = buckets;
    hinit->hash->size = size;

    return NGX_OK;
  }
```

1. 查找一个元素 ngx_hash_find

```
 /**
 * 从hash表中读取一个元素
 */
 void *
 ngx_hash_find(ngx_hash_t *hash, ngx_uint_t key, u_char *name, size_t
len)
 {
    ngx_uint_t i;
    ngx_hash_elt_t *elt;

 #if 0
    ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "hf:\"%*s\"", len,
name);
 #endif

    /* 获取对应的桶 */
    elt = hash->buckets[key % hash->size];

    if (elt == NULL) {
        return NULL;
    }

    /* 在桶的链表上，查找具体的值;elt元素最后一个elt->value==NULL */
    while (elt->value) {
        if (len != (size_t) elt->len) {
            goto next;
        }

        for (i = 0; i < len; i++) {
            if (name[i] != elt->name[i]) {
                goto next;
            }
        }

        return elt->value;

        next:

        /* 因为在内存池上申请内存，并且是自己处理整块内存，为了CPU读取速度更快，
```

```
进行了内存对齐 */
        elt = (ngx_hash_elt_t *) ngx_align_ptr(&elt->name[0] + elt-
>len,
                sizeof(void *));
        continue;
    }

    return NULL;
 }
```