Apache 内存池内幕(1)

对于 APR 中的所有的对象中，内存池对象应该是其余对象内存分配的基础，不仅是 APR 中的对象，而且对于整个 Apache 中的大部分对象的内存都是从内存池中进行分配的，因此我们将把内存池作为整个 APR 的基础。

2.1 内存池概述

在 C 语言中，内存管理的问题臭名昭著，一直是开发人员最头疼的问题。对于小型程序而言，少许的内存问题，比如内存泄露可能还能忍受，但是对于 Apache 这种大负载量的服务器而言，内存的问题变得尤其重要，因为丝毫的内存泄露以及频繁的内存分配都可能导致服务器的效率下降甚至崩溃。

通常情况下，内存的分配和释放通常都是 malloc 和 free 显式进行的。这样做显得单调无味，同时也可能充满各种令人厌恶的问题。对同一块内存的多次释放通常会导致页面错误，而一直不释放又导致内存泄露，并且使得服务器性能大大下降。

为了在大而且复杂的 Apache 中避免内在的内存管理问题，Apache 的开发者创建了一套基于池概念的内存管理方案，最后这套方法移到 APR 中成为通用的内存管理方案。

在这套方案中，核心概念是池的概念。Apache 中的内存分配的基本结构都是资源池，包括线程池，套接字池等等。内存池通常是一块很大的内存空 间，一次性被分配成功，然后需要的时候直接去池中取，而不需要重新分配，这样避免的频繁的 malloc 操作，而且另一方面，即时内存的使用者忘记释放内存 或者根本就不想分配，那么这些内存也不会丢失，它们仍然保存在内存池中，当内存池被销毁的时候这些内存将自动的被销毁。

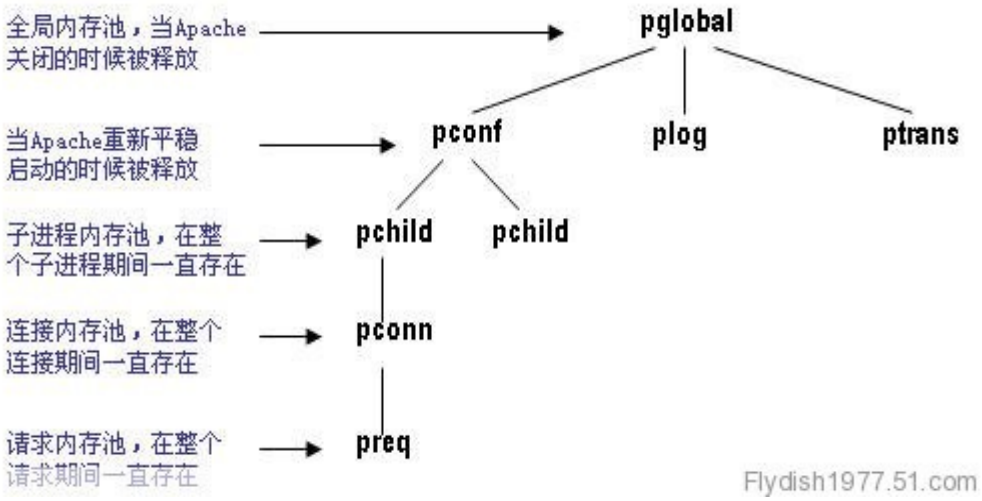
由于 Apache 中的大部分资源的分配都是从内存池中分配的，因此对于大部分的 Apache 函数，如果其内部需要进行资源分配，那么它的函数参数中总是会带有一个内存池参数，该内存池参数指明分配内存来自的内存池，比如下面的两个函数：

```
APR_DECLARE(apr_array_header_t *) apr_array_copy(apr_pool_t *p,const apr_array_header_t *arr);
APU_DECLARE_NONSTD(apr_status_t) apr_bucket_setaside_noop(apr_bucket *data,apr_pool_t *pool);
```

由于在函数的内部需要进行内存分配，因此这两个函数的参数中都指定了一个 apr_pool_t 的结构，用以指名函数内存分配来自的内存池。在后面的大部分过程中我们对于该参数将不再做多余的解释。

Apache 中的内存池并不是仅仅一个内存池，相反而是存在多个内存池，这些内存池之间形成层次结构。如果 Apache 中仅仅存在一个内存池的 话，潜在的问题是所有的内存分配都来自这个池，而且最要命的这些内存必须在整个 Apache 关闭时候才被释放，这一点显然不是那么合情合理，为此 Apache 中根据处理阶段的周期长短又引出了子内存池的概念，与之对应的是父内存池以及根内存池的概念，它们的唯一区别就是存在的周期的不同而已。比如 对于 HTTP 连接而言，包括两种内存池：连接内存池和请求内存池。由于一个连接可能包含多个请求，因此连接的生存周期总是比一个请求的周期长，为此连接处 理中所需要的内存

则从连接内存池中分配，而请求则从请求内存池中分配。而一个请求处理完毕后请求内存池被释放，一个连接处理后连接内存池被释放。根内存池 在整个 Apache 运行期间都存在。Apache 中一个内存池的层次结构图可以大致如下描述：



内存池的层次图

2.2 内存池分配结点

在了解内存池的概念之前，我们首先了解一些内存池分配结点的概念。为了能够方便的对分配的内存进行管理，Apache 中使用了内存结点的概念来 描述每次分配的内存块。其结构类型则描述为 apr_memnode_t，该结构定义在文件 Apr_allocator.h 中，其定义如下：

```
/** basic memory node structure */
struct apr_memnode_t {
    apr_memnode_t *next;              /**< next memnode */
    apr_memnode_t **ref;              /**< reference to self */
    apr_uint32_t   index;             /**< size */
    apr_uint32_t   free_index;        /**< how much free */
    char          *first_avail;       /**< pointer to first free
memory */
    char          *endp;              /**< pointer to end of free
memory */
};
```

该结点类型是整个 Apache 内存管理的基石，在后面的部分我们将其称之为“**内存结点类型**”或者简称为“**内存结点**”或者“**结点**”。在该结构中，不同的结点之间通过 next 指针形成结点链表；另外当在结点内部的时候为了方便引用结点本身，成员变量中还引入了 ref，该变量主要用来记录当前结点的首地址，即使身在结点内部，也可以通过 ref 指针得到该结点并对该结点进行操作。

从上面的结构中可以看出事实上在 apr_memnode_t 结构内部没有任何的“空闲空间”来容纳实际分配的内存，事实上，它从来不单独存在， 总是依附于具体

的分配的内存单元。通常情况下，一旦分配了实际的空间之后，Apache 总是将该结构置于整个单元的最顶部，如图 3.1 所示。

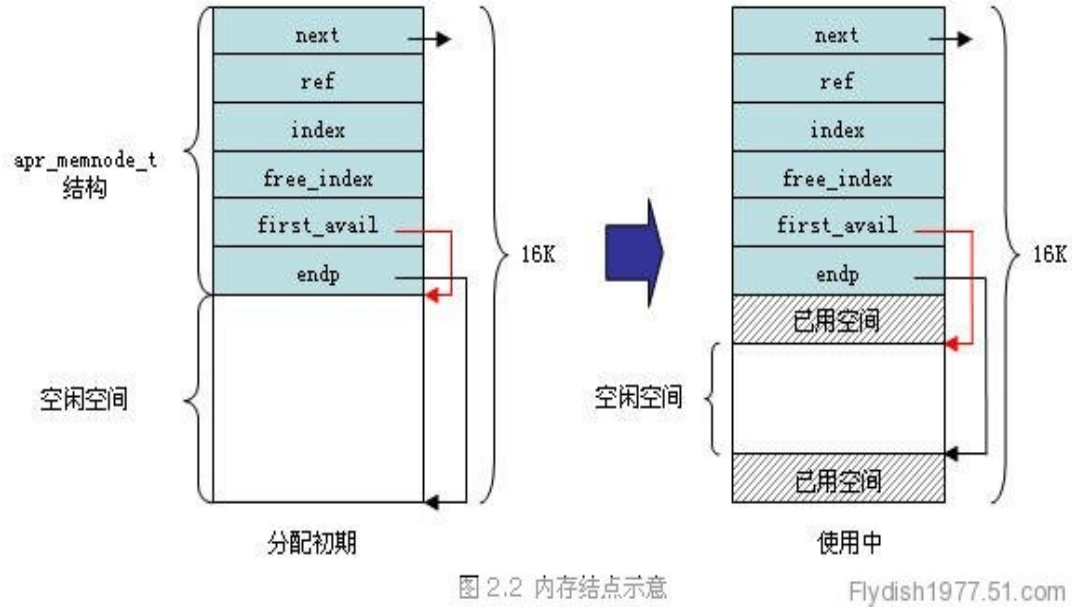


图 3.1 内存结点示意

在上图中，我们可能调用 malloc 函数分配了 16K 大小的空间，为了能够将该空间用 Apache 的结点进行记录，我们将 apr_memnode_t 置于整个空间的头部，此时剩下的可用空间大小应该为 16K-sizeof(apr_memnode_t)，同时结构中还提供了 first_avail 和 end_p 指针分别指向这块可用空间的首部和尾部。当这块可用空间被不断利用时，first_avail 和 end_p 指针也不断 随之移动，不过(end_p-first_avail)之间则永远是当前的空闲空间。上图的右边部分演示了这种布局。

通常情况下，其分配语句大致如下：

```
apr_memnode_t* node;
node=(apr_memnode_t*)malloc(size);
node->next = NULL;
node->index = index;
node->first_avail = (char *)node + APR_MEMNODE_T_SIZE;
node->endp = (char *)node + size;
```

Apache 中对内存的分配大小并不是随意的，随意的分配可能会造成更多的内存碎片。为此 Apache 采取的则是“**规则块**”分配原则。Apache 所支持的分配的最小空间是 8K，如果分配的空间达不到 8K 的大小，则按照 8K 去分配；如果需要的空间超过 8K，则将分配的空间往上调整为 4K 的倍数。为此我们在程序中很多地方会看到下面的宏 APR_ALIGN，其定义如下：

```
/* APR_ALIGN() is only to be used to align on a power of 2 boundary
*/
#define APR_ALIGN(size, boundary) \
    (((size) + ((boundary) - 1)) & ~((boundary) - 1))
```

该宏所做的无非就是计算出最接近 size 的 boundary 的整数倍的整数。通常情况下 size 大小为整数即可，而 boundary 则必须保证 为 2 的倍数。比如

APR_ALIGN(7,4)为 8；APR_ALIGN(21,8)为 24；APR_ALIGN(21,16)则为 32。不过 Apache 中用的最多的还是 APR_ALIGN_DEFAULT，其实际上是 APR_ALIGN(size,8)。在以后的地方，我们将这种处理方式称 之为“8 对齐”或者“4K 对齐”或者类似。
因此如果对于 APR_ALIGN_DEFAULT(sizeof(apr_memnode_t))，其等同于 APR_ALIGN(sizeof(apr_memnode_t),8)。与之对应，APR 中为了处理方便，同时也将 apr_memnode_t 结构的大 小从 sizeof(apr_memnode_t)调整为 APR_ALIGH_DEFAULT(sizeof(apr_memnode_t))。在前面的部 分我们曾经描述过，对于一块 16K 的内存区域，如果其用 apr_memnode_t 进行记录的话，实际的可用空间大小并不是 16K- sizeof(apr_memnode_t)，更精确地则应该是 16K-APR_ALIGN_DEFAULT(sizeof(apr_memnode_t))。
因此如果我们看到 Apache 中的下面的语句，我们就没有什么好惊讶的了。

```
size = APR_ALIGN(size + APR_MEMNODE_T_SIZE, 4096);
if (size <8192)
size = 8192;
```

在上面的代码中我们将实际的常量都替换成实际的整数。APR_MEMNODE_T 是对 sizeof(apr_memnode_t)进行调整后的 值。上面的语句所作的正是我们前面所说的分配策略：如果需要分配的空间累计结点头的空间总和小于 8K，则以 8K 进行分配，否则调整为 4K 的整数倍。按照这 种分配策略，如果我们要求分配的 size 大小为 4192，其按照最小单元分配，实际分配大小为 8192；如果我们要求分配的空间为 8192，由于其加上内 存结点头，大于 8192，此时将按照最小单元分配 4k，此时实际分配的空间大小为 8192+4996=12K。这样，每个结点的空间大小都不完全一样，为 此分配结点本身必须了解本结点的大小，这个可以使用 index 进行记录。
不过 Apache 记录内存的大小有自己的独特的方法。如果空间为 12K，那么 Apache 并不会直接将 12K 赋值给 index 变量。相反，index 只是记录当前结点大小相对于 4K 的倍数，计算方法如下：

```
index = (size >> BOUNDARY_INDEX) - 1;
```

这样如果 index =5，我们就可以知道该结点大小为 20K；反过来也是如此。通过这样方法，可以节省一定的存储空间，另一方面，也方便了程序处理。在后面的部分，我们将通过这种方法计算出来的值称之为“索引大小”，因此在后面的部分，我们如果需要描述内存结点大小的时候，我们直接称之为“索引大小为 n”或者“大小为 n”，后面不再赘述。与此相同，free_index 则是定义了当前结点中的可用的空间的大小。
Apache 内存池内幕(2)

2.3 内存池分配子 allocator

2.3.1 分配子概述

尽管我们可以通过 malloc 函数直接分配 apr_memnode_t 类型的结点，不过 Apache 中并不推

荐这种做法。事实上 Apache 中 的大部分的内存的分配都是由内存分配子 allocator 完成的。它隐藏了内部的实际的分配细节，对外提供了几个简单的接口供内存池函数调用。内存分配子 属于内部数据结构，外部程序不能直接调用。内存分配子(以后简称为分配子)在文件 apr_pools.c 中进行定义如下：

```
struct apr_allocator_t {
        apr_uint32_t       max_index;
        apr_uint32_t       max_free_index;
        apr_uint32_t       current_free_index;
#if APR_HAS_THREADS
        apr_thread_mutex_t  *mutex;
#endif /* APR_HAS_THREADS */
        apr_pool_t         *owner;
        apr_memnode_t     *free[MAX_INDEX];
};
```

该结构中最重要的无非就是 free 数组，数组的每个元素都是 apr_memnode_t 类型的地址，指向一个 apr_memnode_t 类型的结点链表。内存分配的时候则从实际的结点中进行分配，使用完毕后同时返回给分配子。

不过 free 中的链表中结点的大小并不完全相同，其取决于当前链表在 free 数组中的索引。此处 free 数组的索引 index 具有两层次的含 义：第一层，该结点链表在数组中的实际索引，这是最表层的含义；另外，它还标记了当前链表中结点的大小。索引越大，结点也就越大。同一个链表中的所有结点 大小都完全相等，结点的大小与结点所在链表的索引存在如下的关系：

结点大小 = 8K + 4K*(index-1)

因此如果链表索引为 2，则该链表中所有的结点大小都是 12K；如果索引为 MAX_INDEX，即 20，则结点大小应该为 8K+4K* (MAX_INDEX-1)=84K，这也是 Apache 中能够支持的“规则结点”的最大数目。不过这个公式仅仅适用于数组中 1 到 MAX_INDEX 的索 引，对于索引 0 则不适合。当且仅当用户申请的内存块太大以至于超过了规则结点所能承受的 84K 的时候，它才会到索引为 0 的链表中去查找。该链表中的结点通 常都大于 84K，而且每个结点的大小也不完全相同。

在后面的部分，我们将索引 1 到 MAX_INDEX 所对应的链表统称为“**规则链表**”，而每一个链表则分开称之为“**索引 n 链表**”，与之对应，规则链表中的结点则统称为“**规则结点**”，或者称则为“**索引 n 结点**”，这是因为它们的大小有一定的规律可遵循；而索引 0 对应的链表则称之为“**索引 0 链表**”，结点则称之为“**索引 0 结点**”。

根据上面的描述，我们可以给出分配子的内存结构如图 3.2 所示。

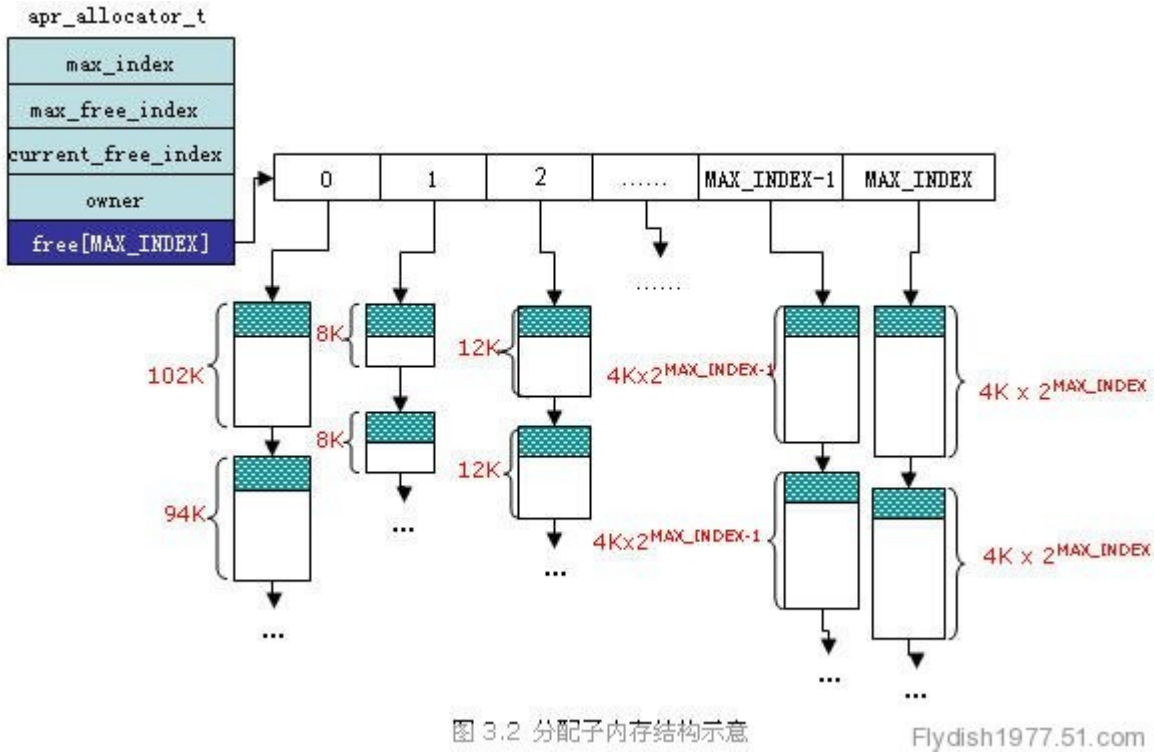


图 3.2 分配子内存结构示意

理论上，分配子中的最大的结点大小应该为 8K+4K*(MAX_INDEX-1)，但实际却未必如此，如果从来没有分配过 8K+4K* (MAX_INDEX-1)大小的内存，那么 MAX_INDEX 索引对应的链表很可能是空的。此时在分配子中我们用变量 max_index 表示实际的最大 结点。另外如果结点过大，则占用内存过多，此时有必要将该结点返回给操作系统，分配子将 max_free_index 作为内存回收的最低门槛。如果该结点 小于 max_free_index，则不做任何处理，否则使用后必须进行释放给操作系统。current_free_index 则是…。除此之 外，mutex 用户保证多线程访问时候的互斥，而 owner 则记录了当前分配子所属于的内存池。

针对分配子，Apache 中提供了几个相关的函数，函数名称和作用简要概述如表格 3.1。

表 3.1 Apache 中提供了分配子相关函数

分配子操作	函数名称	函数功能简单描述
创建	apr_allocator_create	创建一个新的分配子
销毁	apr_allocator_destroy	销毁一个已经存在的分配子
空间分配	apr_allocator_alloc	调用分配子分配一定的空间
空间释放	apr_allocator_free	释放分配子已经分配的空间，将它返回给分配子
其余设置	apr_allocator_owner_set apr_allocator_owner_get	设置和获取分配子所属的内存池
	apr_allocator_max_free_set apr_allocator_set_max_free	设置和获取分配子内部的互斥变量

2.3.2 分配子创建与销毁

分配子的创建是所有的分配子操作的前提，正所谓“毛之不存，皮将焉附”。分配子创建使用函数 apr_allocator_create 实现：

```
APR_DECLARE(apr_status_t) apr_allocator_create(apr_allocator_t **allocator)
{
    apr_allocator_t *new_allocator;
    *allocator = NULL;
    if ((new_allocator = malloc(SIZEOF_ALLOCATOR_T)) == NULL)
        return APR_ENOMEM;
    memset(new_allocator, 0, SIZEOF_ALLOCATOR_T);
    new_allocator->max_free_index = APR_ALLOCATOR_MAX_FREE_UNLIMITED;
    *allocator = new_allocator;
    return APR_SUCCESS;
}
```

分配子的创建非常的简单，它使用的函数则是最通常的 malloc，分配大小为 SIZEOF_ALLOCATOR_T 即 APR_ALIGN_DEFAULT(sizeof(apr_allocator_t))大小。当然这块分配的空间也包括了 MAX_INDEX 个指针变量 数组。一旦分配完毕，函数将 max_free_index 初始化为 APR_ALLOCATOR_MAX_FREE_UNLIMITED，该值实际为 0，表 明分配子对于回收空闲结点的大小并不设门槛，意味着即使结点再大，系统也不会回收。创建后，结构中的 max_inde，current_free_index 都被初始化为 0，这实际上是由 memset 函数隐式初始化的。一旦创建完毕，函数将返回创建的分配子。只不过此时返回的分配子中的 free 数组中不包含任何的实际的内存结点链表。

对分配子使用的正常的下一步就应该是对结构成员进行初始化。主要的初始化工作就是设置系统资源归还给操作系统的门槛 max_free_index。在后面我们会看到，对于使用 malloc 分配的内存，如果其大小小于该门槛值，那么这些资源并不释放，而是归还给内存池， 当内存池本身被释放的时候，这些内存才真正释放给操作系统；如果内存的大小大于这个门槛值，那么内存将直接释放给操作系统。这个门槛值的设置由函数 apr_allocator_max_free_set 完成：

```
APR_DECLARE(void) apr_allocator_max_free_set(apr_allocator_t *allocator,
                                             apr_size_t in_size)
{
    apr_uint32_t max_free_index;
    apr_uint32_t size = (APR_UINT32_TRUNC_CAST)in_size;
    max_free_index = APR_ALIGN(size, BOUNDARY_SIZE) >> BOUNDARY_INDEX;
    allocator->current_free_index += max_free_index;
    allocator->current_free_index -= allocator->max_free_index;
    allocator->max_free_index = max_free_index;
    if (allocator->current_free_index > max_free_index)
        allocator->current_free_index = max_free_index;
}
```

参数中的 size 经过适当的对齐调整赋值给分配子结构中的 max_free_index。除了

max_free_index 之外，另外一个重要 的成员就是 current_free_index，该成员记录当前内存池中实际的最大的内存块大小。当然，它的值不允许超出 max_free_index 的范围。
与分配子的创建对应的则是分配子的销毁，销毁使用的是函数 apr_allocator_destroy。当分配子被销毁的时候，我们需要确保下面两方面的内容都被正确的销毁：
(1)、分配子本身的内存被释放，这个可以直接调用 free 处理
(2)、由于分配子中内嵌的 free 数组都指向一个实际的结点链表，因此必须保证这些链表都被正确的释放。在释放链表的时候，通过一旦得到头结点，就可以沿着 next 遍历释放链表中的所有结点。
必须需要注意的是两种释放之前的释放顺序问题。正确的释放顺序应该是链表释放最早；其次才是分配子本身内存的释放。Apache 中对应该部分是释放代码如下：

```
APR_DECLARE(void) apr_allocator_destroy(apr_allocator_t *allocator)
{
    apr_uint32_t index;
    apr_memnode_t *node, **ref;
    for (index = 0; index < MAX_INDEX; index++) {
        ref = &allocator->free[index];
        while ((node = *ref) != NULL) {
            *ref = node->next;
            free(node);
        }
    }
    free(allocator);
}
```

Apache 内存池内幕(3)

2.3.3 分配子内存分配

使用分配子分配内存是最终的目的。Apache 对外提供的使用分配子分配内存的函数是 apr_allocator_alloc，然而实际在内部，该接口函数调用的则是 allocator_alloc。
allocator_alloc 函数原型声明如下：

```
apr_memnode_t *allocator_alloc(apr_allocator_t *allocator, apr_size_t size)
```

函数的参数非常简单，allocator 则是内存分配的时候调用的分配子，而 size 则是需要进行分配的大小。如果分配成功，则返回分配后的 apr_memnode_t 结构。

```
{
    apr_memnode_t *node, **ref;
    apr_uint32_t max_index;
    apr_size_t i, index;

    size = APR_ALIGN(size + APR_MEMNODE_T_SIZE, BOUNDARY_SIZE);
    if (size < MIN_ALLOC)
        size = MIN_ALLOC;
```

```
index = (size >> BOUNDARY_INDEX) - 1;
if (index > APR_UINT32_MAX) {
    return NULL;
}
```

函数所做的第一件事情就是按照我们前面所说的分配原则调整实际分配的空间大小：如果不满 8K，则以 8K 计算；否则调整为 4K 的整数倍。同时函数 还将计算该与该结点对应的索引大小。一旦得到索引大小，也就知道了结点链表。至此 Apache 可以去寻找合适的结点进行内存分配了。

从分配子中分配内存必须考虑下面三种情况：

(1)、如果需要分配的结点大小分配子中的“**规则结点**”能够满足，即 index<=allocator->max_index。此时，能够满足分配的最小结点就是 index 索引对应的链表结点，但此时该索引对 应的链表可能为空，因此函数将沿着数组往后查找直到找到第一个可用的不为空结点或者直到数组末尾。同时程序代码中还给出了另外一种策略以及不使用的原因：

NOTE: an optimization would be to check allocator->free[index] first and if no node is present, directly use allocator->free[max_index]. This seems like overkill though and could cause memory waste.

另外一种方案就是首先直接检查 allocator->free[index]，一旦发现不可用，直接使用最大的索引 allocator->free[max_index]，不过这种策略可能导致内存的浪费。Apache 采用的则是“最合适”原则，按照这种原则， 找到的第一个内存肯定是最合适的。下面的斜体代码所作的无非如此：

```
if (index <= allocator->max_index) {
    max_index = allocator->max_index;
    ref = &allocator->free[index];
    i = index;
    while (*ref == NULL && i < max_index) {
        ref++;
        i++;
    }
```

当循环退出的时候，意味着遍历结束，这时候可能产生两种结果：第一，找到一个非空的链表，这时候我们可以进入链表内部进行实际的内存分配；第二，从 index 开始往后的所有的链表都是空的，至此，循环退出的时候 i=max_index。这两种情况可以用下面的简图描述：

对于第一种情况，处理如下：

```
    if ((node = *ref) != NULL) {
        if ((*ref = node->next) == NULL && i >= max_index) {
            do {
                ref--;
                max_index--;
            }
            while (*ref == NULL && max_index > 0);
            allocator->max_index = max_index;
        }
        allocator->current_free_index += node->index;
        if (allocator->current_free_index > allocator->max_free_index)
```

```
            allocator->current_free_index = allocator->max_free_index;
        node->next = NULL;
        node->first_avail = (char *)node + APR_MEMNODE_T_SIZE;

        return node;
    }
```

(2)、如果分配的结点大小超过了 “规则结点”中的最大结点，函数将考虑索引 0 链表。索引 0 链表中的结点的实际大小通过成员变量 index 进行标记。
在通过 next 遍历索引 0 链表的时候，函数将需要的大小 index 和实际的结点的大小 node->index 进行比较。如果 index>node->index，则明显该结点无法满足分配要求，此时必须继续遍历。一旦找到合适的可供分配的结点大小，函数将调整 node->first_avail 指针指向实际可用的空闲空间。另外还需要调整分配子中的 current_free_index 为新的分配后的 值。
(3)、如果在 free[0]链表中都找不到合适的空间供分配，那么此时只能“另起炉灶”了。函数能做的事情无非就是调用 malloc 分配实际大小的空间，并初始化结点的各个变量，并返回，代码如下：

```
if ((node = malloc(size)) == NULL)
        return NULL;
node->next = NULL;
node->index = index;
node->first_avail = (char *)node + APR_MEMNODE_T_SIZE;
node->endp = (char *)node + size;
```

下面我们来看一个 Apache 中典型的调用分配子分配空间的情况，下面的代码你可以在 worker.c 中找到：

```
apr_allocator_t *allocator;
apr_allocator_create(&allocator);
apr_allocator_max_free_set(allocator, ap_max_mem_free);
apr_pool_create_ex(&ptrans, NULL, NULL, allocator);
apr_allocator_owner_set(allocator, ptrans);
```

当我顺着这段代码往下阅读的时候，我曾经感觉到很困惑。当一个分配子创建初始，内部的 free 数组中的索引链表都为空，因此当我们在 apr_pool_create_ex 中调用 node = allocator_alloc(allocator, MIN_ALLOC - APR_MEMNODE_T_SIZE)) == NULL 的时候，所需要的内存就不可能来自索引链表内的结点中，而只能就地分配，这些结点一旦分配后，它们就作为内存池的结点而被使用，但是分配后的结点 却并没有立即与 free 数组进行关联，即并没有对 free 数组中的元素进行赋值。这样，如果不将结点与 free 数组进行“挂接”，那么将永远都不可能形成 图一所示链表结构。
那么它们什么时候才挂接到 free 数组中的呢？原来所有的挂接过程都是在结点释放的时候才进行的。

2.3.4 分配子内存释放

正如前面所描述的，在分配内存的时候，Apache 首先尝试到现有的链表中去查找适合的空

间，如果没有适合的内存区域的话，Apache 必须按 照上述的分配原则进行实际的内存分配并使用。但是实际的内存块并不会立即挂接到链表中去，只有释放的时候，这些区域才挂接到内存中。所以从这个角度而言， 分配子内存的释放并不是真正的将内存调用 free 释放，而将其回收到分配链表池中。

Apache 中提供的内存释放函数是 apr_allocator_free。不过该函数仅仅是对外提供的接口而已，在函数内存调用的则实际上是 allocator_free。allocator_free 函数的原型如下：

static APR_INLINE void allocator_free(apr_allocator_t *allocator, apr_memnode_t *node)

函数中，node 是需要释放的内存结点，其最终归还给分配子 allocator。

```
{
    apr_memnode_t *next, *freelist = NULL;
    apr_uint32_t index, max_index;
    apr_uint32_t max_free_index, current_free_index;

    max_index = allocator->max_index;
    max_free_index = allocator->max_free_index;
    current_free_index = allocator->current_free_index;
```

由于 node 不仅仅可能是一个结点，而且可能是一个结点链表，因此如果需要完全释放该链表中的结点，则必须通过结点中的 next 进行依次遍历，因此下面的循环就是整个释放过程的框架结构：

```
    do {
        next = node->next;
        index = node->index;
        ……
    } while ((node = next) != NULL);
```

对于每个结点，我们将根据它的索引大小（即内存大小）采取不同的处理策略：

(1)、如果结点的大小超过了完全释放的阙值 max_free_index，那么我们就不能将其简单的归还到索引链表中，而必须将其完全归还给 操作系统。函数将所有的这样的需要完全释放的结点保存在链表 freelist 中，待所有的结点遍历完毕后，只需要释放 freelist 就可以释放所有的必 须释放的结点，如下所示：

```
        if (max_free_index != APR_ALLOCATOR_MAX_FREE_UNLIMITED
            && index > current_free_index) {
            node->next = freelist;
            freelist = node;
        }
```

如果 max_free_index 为 APR_ALLOCATOR_MAX_FREE_UNLIMITED 则意味着没有回收门槛。任何内存，不管它有多大，APR 都不会将其归还给操作系统。

(2)、如果 index<MAX_INDEX，则意味着该结点属于“规则结点”的范围。因此可以将该结点返回到对应的“规则链表”中。如 果需要释放的结点的索引大小为 index，则该结点将挂接于 free[index]链表中。如果当前的 free[index]为空，表明该大小的结点是第 一个结点，此时还必须比较 index 和 max_index。如果 index>max_index，则必须重新更新 max_index 的大小，同时将 该结点插入链表的首部，作为首结点，代码可以描述如下：

```
        else if (index < MAX_INDEX) {
            if ((node->next = allocator->free[index]) == NULL
                && index > max_index) {
```

```
                max_index = index;
            }
            allocator->free[index] = node;
            current_free_index -= index;
        }
```

(3)、如果结点超过了“规则结点”的范围，但是并没有超出回收结点的范围，此时我们则可以将其置于“索引 0”链表的首部中。代码如下：

```
        else {
            node->next = allocator->free[0];
            allocator->free[0] = node;
            current_free_index -= index;
        }
```

待所有的结点处理完毕后，我们还必须调整分配子中的各个成员变量，包括 max_index 和 current_free_index。同时不要忘记释放 freelist 链表。

```
    allocator->max_index = max_index;
    allocator->current_free_index = current_free_index;
    while (freelist != NULL) {
        node = freelist;
        freelist = node->next;
        free(node);
    }
```

当上面的工作都完成后，整个结点的释放也就完毕了。事实上整个内存池中的内存就是通过上面的不断地释放而构建起来的。一旦构建了内存池，下一次的时候则可以直接去内存池中获取了。

2.3.5 分配子内存管理流程

根据上面的描述，我们现在来串起来看一些整个分配子工作的流程。假如存在下面一段代码：

```
1. apr_allocator_t *allocator;
2. apr_allocator_create(&allocator);
3. apr_allocator_max_free_set(allocator, 0);//简单起见，不进行任何回收
4. apr_memnode_t *memnode1 = apr_allocator_alloc(allocator, 3000);
5. apr_allocator_free(memnode1);
6. apr_memnode_t *memnode2 = apr_allocator_alloc(allocator, 3000);
7. apr_allocator_free(memnode2);
8. apr_memnode_t *memnode3 = apr_allocator_alloc(allocator, 3000);
9. apr_allocator_free(memnode3);
```

当第一行执行完毕后，创建的分配子示意图如下图所以，该图中尚未有任何的内存块可供分配：

在第四行中，系统需要内存分配子分配 2000 字节的空间，但此时没有任何空间可供分配(index > allocator->max_index，同时 allocator->free[0]==NULL)，因此分配子将直接向操作系统索取 8K 的空间，剔除结构头的大小，实际可用的内存大小为 8k-

APR_MEMNODE_T_SIZE。
当执行完第五行的时候，该内存将被归还给分配子，同时保存在索引 1 链表中。下图中的虚线剔除后为释放前的状态，反之为释放后的状态。结果如下图：
现在我们来考虑第六行和第七行的执行结果。当再次向分配子申请 3000K 的内存的时候，经过计算发现，该内存必须到索引为 1 链表中去获取。如果索引 1 链表为 NULL，则重复前面的步骤；
Apache 内存池内幕(4)

2.4 内存池

2.4.1 内存池概述

在了解了内存分配子的概念之后，我们其实已经了解了 Apache 中内存分配的细节了。不过 Apache 中内存的层次结构关系则是由内存池负责组织，其数据结构 apr_pool_t 定义在 apr_pools.c 中，定义如下：

```
struct apr_pool_t {
    apr_pool_t          *parent;
    apr_pool_t          *child;
    apr_pool_t          *sibling;
    apr_pool_t          **ref;  //用于指向内存池本身
    cleanup_t           *cleanups;
    apr_allocator_t     *allocator;
    struct process_chain *subprocesses;
    apr_abortfunc_t     abort_fn;
    apr_hash_t          *user_data;
    const char          *tag;

#if !APR_POOL_DEBUG
    apr_memnode_t       *active;
    apr_memnode_t       *self; /* The node containing the pool itself */
    char                *self_first_avail;

#else /* APR_POOL_DEBUG */
    debug_node_t        *nodes;
    const char          *file_line;
    apr_uint32_t        creation_flags;
    unsigned int        stat_alloc;
    unsigned int        stat_total_alloc;
    unsigned int        stat_clear;
#if APR_HAS_THREADS
    apr_os_thread_t     owner;
```

```
    apr_thread_mutex_t   *mutex;
#endif /* APR_HAS_THREADS */
#endif /* APR_POOL_DEBUG */
#ifdef NETWARE
    apr_os_proc_t        owner_proc;
#endif /* defined(NETWARE) */
};
```

Apache 中存在的内存池个数通常多于一个，它们之间形成树型层次结构。每个内存池所存储的内容以及其存储周期都不一样，比如连接内存池在整 个 HTTP 连接期间存在，一旦连接结束，内存池也就被释放；请求内存池则周期要相对短，它仅仅在某个请求周期内存存在，一旦请求结束，请求内存池也就释 放。不过每个内存池都具有一个 apr_pool_t 结构。整个内存池层次树通过 parent、child 以及 sibling 三个变量构建起来。parent 指向当前内存池的父内存池；child 指向当前内存池的子内存池；而 sibing 则指向当前内存池的兄弟内存池。因此整个内存池树结构可以用图 3.3 描述：

图 3.3 内存池层次树结构图

在上面的图中，我们只是表示了层次结构，因此只是用了 child 和 sibling 两个成员，而忽略的 parent 的变量。从上面的图中我们可以 看出根结点具有 n 个孩子结点：child1，child2，child3…childn。而 child1，child2，child3 以及 childn 它 们属于同一个父亲，而且处于层次树的同一层次，因此它们通过链表连接，互为兄弟结点。同样 child10 和 child11 都是 child1 的子内存池结 点，互为兄弟结点。child21 是 child2 的唯一的子结点。其余结点类似。

除此之外 apr_pool_t 结构中最重要的成员变量无非就是 active 了。

图 3.4

Apache 中提供了大量的内存池管理函数，它们的功能和名称归纳在表格 3.2 中。

内存池操作	函数名称	函数功能简单描述
初始化	apr_pool_initialize	对内存池使用中需要的内部变量进行初始化
销毁	apr_pool_terminate	主要在终止内存池使用时销毁内部的结构
创建	apr_pool_create_ex apr_pool_create_ex_debug	创建一个新的内存池，另外还包括一个调试版本
清除	apr_pool_clear apr_pool_clear_debug	清除内存池中的所有的内存，另外包括一个调试版本
	apr_pool_destroy	

2.4.2 内存池的初始化

内存池的初始化是通过函数 apr_pool_initialize 实现的，在内部函数完成了下面几件事情：

```
APR_DECLARE(apr_status_t) apr_pool_initialize(void)
{
    apr_status_t rv;
    if (apr_pools_initialized++)
        return APR_SUCCESS;
```

(1)、确保 Apache 中只创建一个全局内存池，为此，Apache 中使用 apr_pools_initialized 进行记录。apr_pools_initialized 初始值为 0，初始化后该值更改为 1。每次初始化之前都检查该值，只有值为 0 的时候才允许继续执行初始化操作， 否则直接返回。通过这种手段可以确保只有一个全局内存池存在。

```
    if ((rv = apr_allocator_create(&global_allocator)) != APR_SUCCESS) {
        apr_pools_initialized = 0;
        return rv;
    }
    if ((rv = apr_pool_create_ex(&global_pool, NULL, NULL,
                                 global_allocator)) != APR_SUCCESS) {
        apr_allocator_destroy(global_allocator);
        global_allocator = NULL;
        apr_pools_initialized = 0;
        return rv;
    }
    apr_pool_tag(global_pool, "apr_global_pool");
```

(2)、创建了全局的分配子 global_allocator，并使用全局分配子 global_allocator 创建了全局内存池 global_pool，该内存池是所有的内存池的祖先。所有的内存池都从该内存池继承而来。它在整个 Apache 的生存周期都存在，即使重启机器，该内 存池也不会释放。除非你把 Apache 彻底关闭。该内存池在系统中命名为“apr_gloabl_pool”。

```
    if ((rv = apr_atomic_init(global_pool)) != APR_SUCCESS) {
        return rv;
    }
#if APR_HAS_THREADS
    {
        apr_thread_mutex_t *mutex;
        if ((rv = apr_thread_mutex_create(&mutex,APR_THREAD_MUTEX_DEFAULT,global_pool)) != APR_SUCCESS) {
            return rv;
        }
        apr_allocator_mutex_set(global_allocator, mutex);
    }
#endif /* APR_HAS_THREADS */
```

apr_allocator_owner_set(global_allocator, global_pool);

(3)、如果当前的操作系统允许多线程，为了确保内存池结构被多线程访问的时候的线程安全性，还必须设置 apr_pool_t 结构内的互斥锁变量 mutex。最后的任务就是将内存分配子和内存池进行关联。

Apache 内存池内幕(5)

2.4.3 内存池的创建

勿庸置疑，内存池的创建是内存池的核心操作之一。内存池创建函数的原型如下所示：

```
APR_DECLARE(apr_status_t) apr_pool_create_ex(apr_pool_t **newpool,
                                             apr_pool_t *parent,
                                             apr_abortfunc_t abort_fn,
                                             apr_allocator_t *allocator)
```

其中，newpool 是需要创建的新的内存池，并且创建后的内存池通过该参数返回。parent 则是当前创建的内存池的父亲；abort_fn 指明了当创建失败的时候所调用的处理函数；allocator 则是真正进行内存分配的分配子。

```
{
    apr_pool_t *pool;
    apr_memnode_t *node;

    *newpool = NULL;
    if (!parent)
        parent = global_pool;
    if (!abort_fn && parent)
        abort_fn = parent->abort_fn;
    if (allocator == NULL)
        allocator = parent->allocator;
```

在创建过程中，我们没有指定当前创建的内存池的父亲，则将其默认为父亲为根内存池 global_pool，同时如果内存池关联的 abort_fn 和分配子 allocator 没有定义，那么也直接继承父辈的相关信息。

```
    if ((node = allocator_alloc(allocator,MIN_ALLOC - APR_MEMNODE_T_SIZE)) == NULL)
{
        if (abort_fn)
            abort_fn(APR_ENOMEM);
        return APR_ENOMEM;
    }
    node->next = node;
    node->ref = &node->next;
    pool = (apr_pool_t *)node->first_avail;
    node->first_avail = pool->self_first_avail = (char *)pool + SIZEOF_POOL_T;

    pool->allocator = allocator;
    pool->active = pool->self = node;
    pool->abort_fn = abort_fn;
    pool->child = NULL;
```

```
    pool->cleanups = NULL;
    pool->free_cleanups = NULL;
    pool->subprocesses = NULL;
    pool->user_data = NULL;
    pool->tag = NULL;
```

在一切就绪之后，函数将必须首先创建 apr_pool_t 结构。但是前面我们曾经说过 Apache 中对所有内存 的分配都是以内存结点 apr_memnode_t 进行分配的，而且每次分配的最小单元为 8K，这对于创建 apr_pool_t 结构也不例外。因此函数将首 先调用分配子 allocator 分配 8K 的内存，然后将最顶端的内存分配给 apr_memnode_t 结构。此时接着 apr_memnode_t 结构下面 的内存才能继续分配给 apr_pool_t，用来表示内存池结构，apr_pool_t 结构之后才是真正可用的空间。在整个 8K 内存中，结点头和内存池头 部分别占用的空间大小为 APR_MEMNODE_T_SIZE 和 SIZEOF_POOL_T，因此用户真正可用的空间实际上只有(8k- APR_MEMNODE_T_SIZE-SIZEOF_POOL_T)大小了，至此我们还必须要调整 apr_memnode_t 中的 first_avail 指针和 apr_pool_t 结构中的 self_first_avail 指针指向真正可用空间。

经过两轮分配之后，8K 内存的布局如图 3.5 所示：

一旦完成了内存池结点的分配工作，我们必须将其挂结到内存池层次树上。挂结的过程无非就是设置 parent，child 以及 sibling 的过程。

```
    if ((pool->parent = parent) != NULL) {
        if ((pool->sibling = parent->child) != NULL)
            pool->sibling->ref = &pool->sibling;
        parent->child = pool;
        pool->ref = &parent->child;
    }
    else {
        pool->sibling = NULL;
        pool->ref = NULL;
    }
    *newpool = pool;
```

挂结的过程可以分为下面几个步骤：

(1)、将当前的结点的 parent 指针指向父结点，即 pool->parent = parent。

(2)、设定当前结点的 sibing。sibing 应该指向那些与当前结点处于同一层次，并且父结点也相同的结点，新的结点总是被插入到子结点链表的首部，插入通过下面的两句实现：

```
pool->sibling = parent->child;
parent->child = pool;
```

不过如果父结点为空，意味着该结点未有兄弟结点，故 pool->sibling = NULL。

(3)、设置 ref 成员。在 apr_pool_t 中，ref 用于指向

在内存池结点创建的过程中，我们可以看到，内存池创建后 active 仍然为空。因此当前内存池中能够被使用的内 存仅仅为 8k- APR_MEMNODE_T_SIZE-SIZEOF_POOL_T 大小。如果用户从内存池中申请更多的内存的时候，很明显，此时必须通过 active 去 扩展该内存池对应的内存结点。这一点我们可以在内存池的内存分配中看出来。

Apache 内存池内幕(6)

2.4.4 内存池的内存分配

从内存池中分配内存通过两个函数实现：apr_pcalloc 和 apr_palloc，这两个函数唯一的区别就是 apr_pcalloc 分配后的内存全部自动清零，而 apr_palloc 则省去了这一步的工作。apr_palloc 的函数原型如下所示：

APR_DECLARE (void *) apr_palloc (apr_pool_t *pool, apr_size_t size)

函数中 pool 是需要分配内存的内存池，size 则是需要分配的内存的大小。一旦分配成功则返回分配内存的地址。

在了解内存池的内存分配之前，我们应该对 active 链表有所了解。顾名思义，active 链表中保存的都是曾经被使用或者正在被使用的 apr_memnode_t 内存结点。这些结点都是由分配子进行分配，之所以被使用，一个重要的原因就是它们有足够空闲的空间。将这些结点保存在 active 上，这样下次需要内存的时候只要首先遍历 active 链表即可，只有在 active 链表中的结点不能够满足分配要求的时候才会重新跟分配子申 请新的内存。另一方面，一旦某个结点被选中进入 active 链表，那么它就不能在原先的分配子链表中存在。

对于每一个 apr_memnode_t 内存结点，它的实际可用空间为 endp-first_avail 的大小。但是正如前面所说，Apache 中衡量空间 通常使用索引的方法，对于所有的结点，它的空闲空间用 free_index 描述。为了加快查找速度，active 链表中的所有的结点按照其空间空间的大小进行反向排序，为此空闲空间大得总是排在前面，空闲空间最小的则肯定排在最末尾。对于指定的分配空间，只要将其与第一个结点的空闲空间进行比较，如果第一 个空闲都不满足，那么此时必须向分配子重新申请空间，否则直接从第一个结点中分配空间，同时调整分配后的结点次序。

```
apr_memnode_t *active, *node;
void *mem;
apr_size_t free_index;

size = APR_ALIGN_DEFAULT(size);
active = pool->active;

if (size < (apr_size_t)(active->endp - active->first_avail)) {
    mem = active->first_avail;
    active->first_avail += size;
    return mem;
}
```

分配首先计算需要分配的实际空间，这些空间都是使用对齐算法调整过的。Apache 首先尝试到 active 链表 的第一个结点中去分配空间，正如前面所言，这个是链表中空闲最多的结点，如果它能够满足需要，Apache 直接返回 size 大小的空间，同时调整新的 first_avail 指针。不过这里需要注意的是对于空链表的情况。当一个内存池使用 apr_create_pool_ex 新创建以后，它的 active 链表为空，不过此时 active 并不为 NULL，事实上 active=node，意味着 active 指向内存池所在的内存结点。因此这种情况 下，空间的分配并不会失败。

```
node = active->next;
if (size < (apr_size_t)(node->endp - node->first_avail)) {
    list_remove(node);
}
else {
```

```
        if ((node = allocator_alloc(pool->allocator, size)) == NULL) {
            if (pool->abort_fn)
                pool->abort_fn(APR_ENOMEM);

            return NULL;
        }
    }
```

如果active链表中的结点都不能满足分配需求，那么此时唯一能够做的就是直接向分配子申请更多的空间。至于分配子如何去分配，是从池中获取还是直接调用malloc分配，此处不再讨论。

```
    node->free_index = 0;
    mem = node->first_avail;
    node->first_avail += size;

    list_insert(node, active);
    pool->active = node;
    free_index = (APR_ALIGN(active->endp - active->first_avail + 1,
                            BOUNDARY_SIZE) - BOUNDARY_SIZE) >> BOUNDARY_INDEX;

    active->free_index = (APR_UINT32_TRUNC_CAST)free_index;
    node = active->next;
    if (free_index >= node->free_index)
        return mem;
```

一旦获取到分配的新的结点，那么下一步就是从该结点中分配需要的size大小的空间，由mem指针指向该空间首地址。同时将该结点立即插入到actie链表中作为首结点。插入通过宏list_insert实现：

```
#define list_insert(node, point) do {           \
    node->ref = point->ref;                     \
    *node->ref = node;                          \
    node->next = point;                         \
    point->ref = &node->next;                   \
} while (0)
```

前面我们说过，active链表的第一个结点肯定是空闲空间最大的结点。尽管从池中刚分配的时候，node结点的空闲空间确实是最大，但是一旦分配了size大小之后则情况未必，因此node作为第一个结点存在可能是不合适的，为此必须进行适当的调整。

1)、如果新插入结点node的空闲空间确实比后继结点node->next的空间大，那么此时，毫无疑问，node是所有结点中空闲空间最大的结点，物归其所，不需要再调整。

```
    do {
        node = node->next;
    }
    while (free_index < node->free_index);
    list_remove(active);
    list_insert(active, node);
```

2)、如果node的空间比node->next的空间小，那么意味着node放错了地方，为此必须从node->next开始往后遍历找到合适的位置，并从原位置移出，插入新位置。

Apache 内存池内幕(7)

2.4.5 内存池的销毁

由于 Apache 中所有的内存都来自内存池，因此当内存池被销毁的时候，所有从内存池中分配的空间都将受到直接的影响——被释放。但是不同的数据类型可能导致不同的释放结果，目前 Apache 中支持三种不同的数据类型的释放：

1)、普通的字符串数据类型

这类数据类型是最简单的数据类型，对其释放可以直接调用 free 而不需要进行任何的多余的操作

2)、带有析构功能的数据类型

这类数据类型类似于 C++中的对象。除了调用 free 释放之外还需要进行额外的工作，比如 apr_socket_t 结构，它是与套接字的描述结构，除了释放该结构之外，还必须 close 套接字。

3)、进程数据类型

APR 中的进程数据类型用结构 apr_proc_t 进行描述，当然它的分配内存也来自内存池。通常一个 apr_proc_t 对应一个正在运行的进程，因此从内存池中释放 apr_proc_t 结构的时候必然影响到正在运行的进程，如果处理释放和进程的关系是 内存释放的时候必须考虑的问题。

下面我们详细描述每一个中内存销毁策略

2.4.5.1 带有析构功能的数据类型的释放

Apache2.0 内存池中目前存放的数据种类非常繁多，既包括最普通的字符串，又包含各种复杂的数据类型，比 如套接字、进程和线程描述符、文件描述符、甚至还包括各种复杂的自定义数据类型。事实上这是必然的结果。Apache 中倡导一切数据都尽量从内存池中分 配，而实际需要的数据类型则千变万化，因此内存池中如果出现下面的内存布局则不应该有任何的惊讶：

在上面的图示中，从内存池中分配内存的类型包括 apr_bucket_brigade，apr_socket_t，apr_file_t 等等。一个很明显而且必须解决的问题就是如何释放这些内存。当内 存池被释放的时候，内存池中的各种数据结构自然也就被释放，这些都很容易就可以实现，比如 free(apr_socket_t)、 free(apr_dir_t)。不过有的时候情况并不是这么简单。比如对于 apr_socket_t，除了释放 apr_socket_t 结构之外，更重 要的是必须关闭该 socket。这种情况对于 apr_file_t 也类似，除了释放内存外，还必须关闭文件描述符。这项工作非常类似于对象的释放，除了释 放对象本身的空间，还需要调用对象的析构函数进行资源的释放。

因此正确的资源释放方式必须是能够识别内存池中的数据类型，在释放的时候完成与该类型相关的资源的释放工作。某一个数据结构除了调用 free 释放它的空间之外，其余的应该采取的释放措施用数据结构 cleanup_t 描述，其定义如下：

```
struct cleanup_t {
struct cleanup_t *next;
const void *data;
apr_status_t (*plain_cleanup_fn)(void *data);
apr_status_t (*child_cleanup_fn)(void *data);
};
```

该数据结构通常简称为清除策略数据结构。每一个结构对应一个处理策略，Apache 中允许一个数据类型对应多个 策略，各个处理策略之间通过 next 形成链表。data 则是清除操作需要的额

外的数据，由于数据类型的不确定性，因此只能定义为 void*，待真正需要的 时候在进行强制类型转换，通常情况下，该参数总是为当前操作的数据类型，因为清除动作总是与具体类型相关的。另外两个成员则是函数指针，指向真正的清除操 作函数。child_cleanup_fn 用于清除该内存池的子内存池，plain_cleanup_fn 则是用于清除当前的内存池。

为了能够在释放的时候调用对应的处理函数，首先必须在内存池中注册指定类型的处理函数。注册使用函数 apr_pool_cleanup_register，注册函数原型如下：

```
APR_DECLARE(void) apr_pool_cleanup_register(apr_pool_t *p, const void *data,
                    apr_status_t (*plain_cleanup_fn)(void *data),
                    apr_status_t (*child_cleanup_fn)(void *data))
```

p 是需要注册 cleanup 函数的内存池，当 p 被释放时，所有的 cleanup 函数将被调用。Data 是额外的 数据类型，通常情况下是注册的数据类型，plain_cleanup_fn 和 child_cleanup_fn 的含义与 cleanup_t 结构中对应成员 相同，因此假如需要在全局内存池 pglobal 中注册类型 apr_socket_t 类型变量 sock 的处理函数为 socket_cleanup，则注册过 程如下：

```
apr_pool_cleanup_register(pglobal,(void*)sock,socket_cleanup,NULL);
    cleanup_t *c;
    if (p != NULL) {
        if (p->free_cleanups) {
            c = p->free_cleanups;
            p->free_cleanups = c->next;
        } else {
            c = apr_palloc(p, sizeof(cleanup_t));
        }
        c->data = data;
        c->plain_cleanup_fn = plain_cleanup_fn;
        c->child_cleanup_fn = child_cleanup_fn;
        c->next = p->cleanups;
        p->cleanups = c;
    }
```

注册过程非常简单，无非就是将函数的参数赋值给 cleanup_t 结构中的成员，同时将该结点插入到 cleanup_t 链表的首部。

apr_pool_cleanup_kill 函数与 apr_pool_cleanup_register 相反，用于将指定的 cleanup_t 结构从链表中清除。

因此如果需要对一个内存池进行销毁清除操作，它所要做的事情就是遍历该内存池对应的 cleanup_t 结构，并调用 plain_cleanup_fn 函数，该功能有静态函数 run_cleanups 完成，其对应的代码如下：

```
static void run_cleanups(cleanup_t **cref){
cleanup_t *c = *cref;
while (c) {
*cref = c->next;
(*c->plain_cleanup_fn)((void *)c->data);
c = *cref;
}
}
```

2.4.5.2 进程描述结构的释放

尽管关于APR进程的描述我们要到后面的部分才能详细讨论，不过在这部分，我们还是首先触及到该内容。APR中 使用apr_proc_t数据结构来描述一个进程，同时使用apr_procattr_t结构来描述进程的属性。通常一个apr_proc_t对应系统中 一个正在运行的进程。

由于Apache的几乎所有的内存都来自内存池，apr_proc_t结构的分配也毫不例外，比如下面的代码将从内存池p中分配apr_proc_t和apr_procattr_t结构：

```
apr_proc_t    newproc;
apr_pool_t    *p;
apr_procattr_t *attr;
rv = apr_pool_initialize();
rv = apr_pool_create(&p, NULL);
rv = apr_procattr_create(&attr, p);
```

问题是当内存池p被销毁的时候，newproc和attr的内存也将被销毁。系统应该如何处理与newproc对应的运行进程。Apache中支持五种处理策略，这五种策略封装在枚举类型apr_kill_conditions_e中：

```
typedef enum {
    APR_KILL_NEVER,             /**< process is never sent any signals */
    APR_KILL_ALWAYS,            /**< process is sent SIGKILL on apr_pool_t cleanup
*/
    APR_KILL_AFTER_TIMEOUT,     /**< SIGTERM, wait 3 seconds, SIGKILL */
    APR_JUST_WAIT,              /**< wait forever for the process to complete */
    APR_KILL_ONLY_ONCE          /**< send SIGTERM and then wait */
} apr_kill_conditions_e;
```

APR_KILL_NEVER：该策略意味着即使进程的描述结构apr_proc_t被释放销毁，该进程也不会退出，进程将忽略任何发送的关闭信号。

APR_KILL_ALWAYS：该策略意味着当进程的描述结构被销毁的时候，对应的进程必须退出，通知进程退出使用信号SIGKILL实现。

APR_KILL_AFTER_TIMEOUT：该策略意味着当描述结构被销毁的时候，进程必须退出，不过不是立即退出，而是等待3秒超时候再退出。

APR_JUST_WAIT：该策略意味着描述结构被销毁的时候，进城必须退出，但不是立即退出，而是持续等待，直到该进程完成为止。

APR_KILL_ONLY_ONCE：该策略意味着当结构被销毁的时候，只发送一个SIGTERM信号给进程，然后等待，不再发送信号。

现在我们回过头来看一下内存池中的subprocesses成员。该成员定义为process_chain类型：

```
struct process_chain {
    /** The process ID */
    apr_proc_t *proc;
    apr_kill_conditions_e kill_how;
    /** The next process in the list */
    struct process_chain *next;
};
```

对于内存池p，任何一个进程如果需要从p中分配对应 的描述数据结构apr_proc_t，那么它首先必须维持一个process_chain结构，用于描述当p被销毁的时候，如何处理该进程。

Process_chain 的成员很简单，proc 是进程描述，kill_how 是销毁处理策略，如果存在多个进程都从 p 中分配内存，那么这些进程的 process_chain 通过 next 形成链表。反过来说，process_chain 链表中描述了所有的从当前内存池中分配 apr_proc_t 结构 的进程的销毁策略。正因为进程结构的特殊性，因此如果某个程序中需要使用进程结构的话，那么第一件必须考虑的事情就是进程的退出的时候处理策略，并将其保 存在 subprocess 链表中，该过程通过函数 apr_pool_note_subprocess 完成：

```
APR_DECLARE(void) apr_pool_note_subprocess(apr_pool_t *pool, apr_proc_t *proc,
                                           apr_kill_conditions_e how)
{
    struct process_chain *pc = apr_palloc(pool, sizeof(struct process_chain));
    pc->proc = proc;
    pc->kill_how = how;
    pc->next = pool->subprocesses;
    pool->subprocesses = pc;
}
```

apr_pool_note_subproces 的实现非常简单，无非就是对 process_chain 的成员进行赋值，并插入到 subprocess 链表的首部。

比如，在 URI 重写模块中，需要将客户端请求的 URI 更改为新的 URI，如果使用 map 文件进行映射的话，那么根据请求的 URI 到 map 文件中查找新的 URI 的过程并不是由主进程完成的，相反而是由主进程生成子进程，然后由子进程完成的，下面是精简过的代码：

```
static apr_status_t rewritemap_program_child(…)
{
    ……
    apr_proc_t *procnew;
    procnew = apr_pcalloc(p, sizeof(*procnew));
    rc = apr_proc_create(procnew, argv[0], (const char **)argv, NULL,procattr, p);
    if (rc == APR_SUCCESS) {
            apr_pool_note_subprocess(p, procnew, APR_KILL_AFTER_TIMEOUT);
            ……
    }
    ……
}
```

从上面的例子中可以看出，即使描述结构被删除，子进程也必须 3 秒后才被中止，不过 3 秒已经足够完成查询操作了。

同样的例子可以在下面的几个地方查找到：Ssl_engine_pphrase.c 文件的 577 行、Mod_mime_magic.c 文件的 2164 行、mod_ext_filter.c 文件的 478 行、Log.c 的 258 行、Mod_cgi.c 的 465 行等等。

现在我们来考虑内存池被销毁的时候处理进程的情况， 所有的处理由 free_proc_chain 完成，该函数通常仅仅由 apr_pool_clear 或者 apr_pool_destroy 调用，函数仅仅需 要一个参数，就是 process_chain 链表，整个处理框架就是遍历 process_chain 链表，并根据处理策略处理对应的进程，描述如下：

```
static void free_proc_chain(struct process_chain *procs)
{
    struct process_chain *pc;
```

```
    if (!procs)
        return; /* No work. Whew! */
    for (pc = procs; pc; pc = pc->next) {
        if(pc->kill_how == APR_KILL_AFTER_TIMEOUT)
               处理APR_KILL_AFTER_TIMEOUT策略;
        else if(pc->kill_how == APR_KILL_ALWAYS)
               处理APR_KILL_ALWAYS策略;
        else if(pc->kill_how == APR_KILL_NEVER)
               处理APR_KILL_NEVER策略;
        else if(pc->kill_how == APR_JUST_WAIT)
               处理APR_JUST_WAIT策略;
        else if(pc->kill_how == APR_KILL_ONLY_ONCE)
               处理APR_KILL_ONLY_ONCE策略;
    }
}
```

2.4.5.3 内存池释放

在了解了cleanup函数之后，我们现在来看内存池的销毁细节。Apache中对内存池的销毁是可以通过两个函数和apr_pool_clear和apr_pool_destroy进行的。我们首先来看apr_pool_clear的细节：

```
APR_DECLARE(void) apr_pool_clear(apr_pool_t *pool)
{
    apr_memnode_t *active;
```

内存池的销毁不仅包括当前内存池，而且包括当前内存池的所有的子内存池，对于兄弟内存池，apr_pool_destroy并不处理。销毁按照深度优先的原则，首先从最底层的销毁，依次往上进行。函数中通过递归调用实现深度优先的策略，代码如下：

```
while (pool->child)
apr_pool_destroy(pool->child);
```

apr_pool_destroy的细节在下面描述。对于每一个需要销毁的内存池，函数执行的操作都包括下面的几个部分：

```
    run_cleanups(&pool->cleanups);
    pool->cleanups = NULL;
    pool->free_cleanups = NULL;
```

(1)、执行run_cleanups函数遍历与内存池关联的所有的cleanup_t结构，并调用各自的cleanup函数执行清除操作。

```
    free_proc_chain(pool->subprocesses);
    pool->subprocesses = NULL;
    pool->user_data = NULL;
```

(2)、调用free_proc_chain处理使用当前内存池分配apr_proc_t结构的进程。

```
    active = pool->active = pool->self;
    active->first_avail = pool->self_first_avail;

    if (active->next == active)
        return;
```

```
    *active->ref = NULL;
    allocator_free(pool->allocator, active->next);
    active->next = active;
    active->ref = &active->next;
```

(3)、处理与当前内存池关联的 active 链表，主要的工作就是调用 allocator_free 将 active 链表中的所有的结点归还给该内存池的分配子。

apr_pool_destroy 函数与 apr_pool_clear 函数前部分工作非常的相似，对于给定内存 池，它的所有的子内存池将被完全释放，记住不是归还给分配子，而是彻底归还给操作系统。两者的区别是对给定当前内存池节点的处理。 apr_pool_clear 并不会释放内存中的任何内存，而 apr_pool_destroy 则正好相反：

```
    if (pool->parent) {
        if ((*pool->ref = pool->sibling) != NULL)
           pool->sibling->ref = pool->ref;
    }
    allocator = pool->allocator;
    active = pool->self;
    *active->ref = NULL;
    allocator_free(allocator, active);
    if (apr_allocator_owner_get(allocator) == pool) {
        apr_allocator_destroy(allocator);
    }
```

如果当前内存池存在父内存池，那么函数将自己从父内存池的孩子链表中脱离开来。然后调用 apr_allocator_free 将内存归还给关联分配子。如果被释放的内存池正好是分配子的属主，那么属于该内存池的所有的分配子也应该被完全的销 毁返回给操作系统。因此函数将调用 apr_allocator_owner_get(allocator)函数进行判断。

在销毁分配子的时候有一点需要注意的是，由于需要判断当前分配子的是否属于当前的内存池，而内存池结构在检测之前已经被释放，因此，在释放内存池之前必须将其记录下来以备使用。如果缺少了这一步，allocator 可能造成内存泄漏。

现在我们看一下 run_cleanups 函数，该函数很简单，无非就是遍历 cleanup_t 链表，并逐一调用它们的 plain_cleanup_fn 函数。

```
static void run_cleanups(cleanup_t **cref)
{
    cleanup_t *c = *cref;
    while (c) {
        *cref = c->next;
        (*c->plain_cleanup_fn)((void *)c->data);
        c = *cref;
    }
}
```

Apache 数组分析

3.1 数组

3.1.1 数组概述

数组是 Apache 中最经常也是最普通的数据结构，尽管 C 语言中已经提供了一定的数组功能，但是 C 语言数组运用到 Apache 中还存在下面的一些问题：

(1)、C 语言中的数组在定义的时候就必须确定维数目，一旦确定，其长度就不可更改。但是 Apache 中很多情况并不知道数组中能够保存的最大数目，如果 预定义的太大，可能会浪费过多的空间；定义的太小又可能不能满足系统要求；因此 Apache 中需要一种动态的具有弹性伸缩能力的数组，这样实际需要多少元 素就分配多少元素；

(2)、C 语言中提供了另外一种数组动态伸缩的方案，即就是使用 malloc 分配空间，将空间看作数组进行处理。不过对于 malloc 分配的空间只能保存一种类型的数据。如果需要保存其余类型，则需要重新分配。

Apache 中的数组实现方案类似于 STL 或者 Java 中的 Vector，其长度可以保持动态变化，而且其中可以存放不同类型的数据，包括基本数据类型到复合数据类型。

3.1.2 数组结构

Apache 中相关的数组结构都定义在文件 apr_table.h 和 apr_table.c 中了。apr_array_header_t 结构是 Apache 的核心结构，其定义如下：

```
struct apr_array_header_t {
apr_pool_t *pool;
int elt_size;
int nelts;
int nalloc;
char *elts;
};
```

数组的结构示意图片可以用下图表示：

图 2.1 Apache 数组结构图

apr_array_header_t 结构中的 elts 指针指向了实际的用 malloc 分配的空间，每个空间的大小为 elt_size。由于数组元素的个 数是可以动态变化的，因此结构中使用 nalloc 记录当前已经分配的数组中的元素个数。另外由于 Apache 中的数组元素并不是所有的都被用到，因此还需 要 nelts 记录当前已经使用的元素个数，自然剩余的可以利用的数组元素个数为 nalloc-nelts。分配数组所需要的所有空间都从内存池 pool 中 获取。

3.1.3 创建数组

数组的创建通过调用 apr_array_make 实现，不过其内部主要工作由 make_array_core 实现：

```
static void make_array_core(apr_array_header_t *res, apr_pool_t *p,int
nelts, int elt_size, int clear)
{
    if (nelts < 1) {
        nelts = 1;
    }
    if (clear) {
```

```
            res->elts = apr_pcalloc(p, nelts * elt_size);
        }
        else {
            res->elts = apr_palloc(p, nelts * elt_size);
        }
        res->pool = p;
        res->elt_size = elt_size;
        res->nelts = 0;     /* No active elements yet... */
        res->nalloc = nelts;     /* ...but this many allocated */
    }
    APR_DECLARE(apr_array_header_t *) apr_array_make(apr_pool_t *p,int nelts,
int elt_size)
    {
        apr_array_header_t *res;
        res = (apr_array_header_t *) apr_palloc(p, sizeof(apr_array_header_t));
        make_array_core(res, p, nelts, elt_size, 1);
        return res;
    }
```

apr_array_make 函数负责主要用于分配 apr_array_head_t 头结构，而 make_array_core 则主要负责创建和初始化 res 元素数组。Res 数组的大小中每一个元素的大小为 elt_size，总元素个数为 nelts，因此分配的总空间为 nelts*elt_size，空 间分配后继续初始化 apr_array_header_t 结构中的成员。如果 clear 为 1，则调用 apr_pcalloc 进行分配，否则调用 apr_palloc，两者唯一的区别就是 apr_pcalloc 会对分配的空间进行初始化清零；而 apr_palloc 则不会初始化。

除了从头开始创建一个新的数组之外，APR 中还允许使用 apr_array_copy 将一个数组拷贝到一个新的数组：

```
APR_DECLARE(apr_array_header_t *) apr_array_copy(apr_pool_t *p,const
apr_array_header_t *arr)
```

Arr 是需要拷贝的源数组，拷贝后的数组由函数返回。拷贝函数内部首先定义一个 arp_array_header_t 的变量 res，从内存池 p 中为其分配 足够的空间，然后调用 make_array_core 生成一个与 arr 数组大小相同的数组，再调用 memcpy 逐一进行内存拷贝。然后返回该空间。

3.1.4 元素压入和弹出

Apache 中对数组元素的操作类似于堆栈，只允许弹出(pop)和压入(push)两种，因此你无法象在 C 语言中随意的访问数组中的元素。

当数据需要保存到数组中的时候，通过函数 apr_array_push 函数进行，其原型声明如下：

```
APR_DECLARE(void *) apr_array_push(apr_array_header_t *arr)
```

arr 是需要保存到数组中的元素。元素的保存按照从左到右，即从低索引到高索引依次存储，不可能出现 C 语言中的随机保存的情况，因此从这个角度而言，数组称之为堆栈更合适。

在将元素压入数组之前函数首先检查数组中是否有可用的空闲空间；如果有剩余可用空间，其将直接将元素拷贝到第 nelts+1 数组的位置，调整 nelts 为 nelts++，同时返回该元素。如果没有，即意味着当前数据中 nalloc 个空间都已经被使用，即 nelts=nalloc，那么此时

必须扩大数组空间 用来保存压入的数据。不过数组空间的增加并不是需要几个就增加几个，而是遵循下面的批量增加原则：

■ 如果压入数据之前 nalloc 为 0，即尚未分配任何空间，那么此时只创建大小为 elt_size 的空间，即一个元素的空间。

■ 如果当前元素压入之前，系统分配的 nalloc 个单元都已经被使用完毕，那么此时将直接批量一次性将空间扩大到 nalloc*2 个元素空间。

一旦分配完这些空间，第一个可用的空闲空间则是第 nalloc+1 个元素。此时只需要将需要压入的元素拷贝到该地址空间即可，拷贝完之后，将 elt_size 加一，并返回新元素的地址。

与 apr_array_push 类似，Apache 提供了另外的一个版本的压入函数 apr_array_push_noclear，该函数与 apr_array_push 函数功能基本类似，其唯一区别正如函数名称，在于“noclear”。 apr_array_push 在产生新的空闲块，压入新元素之前调用 memset 对新内存块进行清零操作，而“noclear”函数则省去了这一步。

与元素压入匹配，从数组中取元素通过函数 apr_array_pop 进行，其函数声明如下：

APR_DECLARE(void *) apr_array_pop(apr_array_header_t *arr)

元素弹出遵循先进先出的弹出原则，因此被弹出的元素永远都是第 nelts 个，除非当前的数组为空，不在有任何的有效数据。返回语句可以简化为 return arr->elts + (arr->elt_size * (--arr->nelts));

3.1.5 数组合并

Apache 中使用 apr_array_cat 将两个元素进行合并，函数定义如下：

APR_DECLARE(void) apr_array_cat(apr_array_header_t *dst,const apr_array_header_t *src)

函数将数组 src 添加到数组 dst 的末尾。函数首先检查当前的空闲单元能不能容下 src 数组，这里容纳的概念并不是容纳 src 中的 nalloc 个元素，而 仅仅是容纳下 src 中的 nelts 个非空闲单元，并不包括 nalloc-nelts 空闲单元，因此函数需要比较的是(dst->nalloc) - (dst->nelts) >= (src->nelts)，而不是(dst->nalloc) - (dst->nelts) >= (src->nalloc)。如果能够容纳，则没有必要分配新的空间，直接内存拷贝就可以了。

如果 dst 中所有的空闲空单元都不够存放 src 中的空闲单元，那么此时毫无疑问 dst 需要分配新的空间，分配算法如下：

1)、如果 dst 中元素个数为零，此时，将产生一个新的空间。

2)、如果 dst 中元素个数 nalloc 不为零，则首先产生 nalloc*2 个空闲空间。

3)、尽管如此，如果 src 的非空闲元素实在太多，而 dst 本身空闲空间很小，那么即使一次产生 nalloc 个空闲块也不一定能够盛放 src 中的元素。唯一的办法就是不停的产生新的空闲块，每次产生后的总块数都是当前的一倍，直到空闲块总数能够容纳 src 中的非空闲元素为止。这真是下面的代码所做的事情：

```
if (dst->nelts + src->nelts > dst->nalloc) {
int new_size = (dst->nalloc <= 0) ? 1 : dst->nalloc * 2;
while (dst->nelts + src->nelts > new_size) {
new_size *= 2;
}
}
```

一旦确定确定需要产生的空闲块的总数，函数将一次性从内存池 dst->pool 中申请。然后将 src 中的数据拷贝空间空间即可。

4.1 文件概述

文件 I/O 在 Unix 下占据着非常重要的地位，曾有一句经典语句绝对可以说明 file 在 Unix 下的重要性， "In UNIX, everything is a file"，APR 就是本着这个思想对 Unix 文件 I/O 进行了再一次的抽象封装，以提供更为强大和友善的文件 I/O 接口。

APR File I/O 源代码的位置在$(APR_HOME)/file_io 目录下针对不同类型的操作系统，file_io 下又进一步细分为四个目录：netware，os2，unix 和 win32。每个目录中又包含多个.c 文件，每个.c 文件对应一种文件操作，比如

open.c -- 封装了文件的打开、关闭、改名和删除等操作

readwrite.c -- 顾名思义，它里面包含了文件的读写操作；

pipe.c -- 包含了 pipe 相关操作。

还有许多这里不多说，由于文件 I/O 操作复杂，我们下面将仅挑出最常用的文件 I/O 操作进行分析。

4.2 APR 文件定义

正如第一章所言，APR 中为了移植方便，通常会使用自定义的常量来代替原有系统中存在的常量，并提供它们之间的相互转换，这一点对于文件也不例外。APR 几乎对文件中所用到的所有的常量都进行了重新定义，包括文件权限、文件打开方式、文件类型定义等等。这部分我们集中了解 APR 中的这些定义。在后面的部分将很快就会使用到。

4.2.1 文件类型

尽管在 Unix 中，一切皆文件，但是文件又分为不同类型：

1)、普通文件(regular file)。这是最常见的文件类型。UNIX 中使用 S_IFREG 描述该类文件，而 APR 中使用 APR_REG 描述。

2)、目录文件(directory file)。这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。UNIX 中使用 S_IFDIR 描述，在 APR 中使用 APR_DIR 描述。

3)、字符特殊文件(character special file)。这类文件通常对应系统中某些类型的设备。UNIX 中使用 S_IFCHR 描述，而 APR 中使用 APR_CHR 描述。

4)、块特殊文件(block special file)。这类文件典型的用于磁盘设备。系统中的所有设备或者是字符特殊文件，或者是块特殊文件。UNIX 中使用 S_IFBLK 描述，而 APR 中使用 APR_BLK 描述该类文件。

5)、FIFO 文件。这类文件通常用于进程间通信，有时也称之为管道。APR 中使用 APR_PIPE 描述该类文件。

6)、套接字描述文件。这类文件用于进程间的网络通信。UNIX 中使用 S_IFSOCK 描述，而 APR 中使用 APR_SOCK 描述该类文件。

7)、链接文件。最后一种文件类型就是链接文件。这种文件通常仅仅指向另外一个文件。UNIX 中使用 S_IFLNK 描述，而 APR 中使用 APR_LNK 描述该类文件。

综合上面的内容，APR 定义了枚举类型 apr_filetype_e 来描述所有的 Unix 文件类型。

```
typedef enum {
    APR_NOFILE = 0,     /**< no file type determined */
    APR_REG,            /**< a regular file */
    APR_DIR,            /**< a directory */
    APR_CHR,            /**< a character device */
    APR_BLK,            /**< a block device */
```

```
    APR_PIPE,          /**< a FIFO / pipe */
    APR_LNK,           /**< a symbolic link */
    APR_SOCK,          /**< a [unix domain] socket */
    APR_UNKFILE = 127  /**< a file of some other unknown type */
} apr_filetype_e;
```

为了实现 APR 定义和 UNIX 定义的转换，APR 中通过函数 filetype_from_mode 实现从系统定义到 APR 定义的转换。该函数定义在 filestat.c 中：

```
static apr_filetype_e filetype_from_mode(mode_t mode)
{
    apr_filetype_e type;
    switch (mode & S_IFMT) {
    case S_IFREG:
        type = APR_REG; break;
    case S_IFDIR:
        type = APR_DIR; break;
    ……
    }
}
```

4.2.2 文件访问权限

在 UNIX 系统中，每一个文件都对应三组不同的访问权限，即用户存取权限、组用户存取权限和其余用户存取权限。在 UNIX 中，通常用 S_IXXXX 常量来描述这些访问权限。APR 中则使用 APR_FPROT_XXXX 来进行替代，对应的关系如下表所示：

权限标志	含义	值	标准库值
APR_FPROT_USETID	允许设置用户号	0x8000	S_ISUID
APR_FPROT_UREAD	允许用户读操作	0x0400	S_IRUSR
APR_FPROT_UWRITE	允许用户写操作	0x0200	S_IWUSR
APR_FPROT_UEXECUTE	允许用户执行	0x0100	S_IXUSR
APR_FPROT_GSETID	用于设置组号	0x4000	S_ISGID
APR_FPROT_GREAD	允许组成员读取	0x0040	S_IRGRP
APR_FPROT_GWRITE	允许组成员写入	0x0020	S_IWGRP
APR_FPROT_GEXECUTE	允许组成员执行	0x0010	S_IXGRP
APR_FPROT_WSTICKY	粘贴位	0x2000	S_ISVTX
APR_FPROT_WREAD	允许其余成员读取	0x0004	S_IROTH
APR_FPROT_WWRITE	允许其余成员写入	0x0002	S_IWOTH
APR_FPROT_WEXECUTE	允许其余成员执行.	0x0001	S_IXOTH
APR_FPROT_OS_DEFAULT	操作系统的默认的属性值	0x0FFF	0666

在早期版本中，访问权限使用的是 APR_XXXX 形式，比如 APR_UREAD、APR_UWRITE 等等不过目前已经作废。为了保持与低版本的兼容性，你在源文件中还能看到它们。

在 APR 中，文件的访问权限被定义为 apr_fileperms_t 类型，该类型本质上是一个 32 位的整数而已：

```
typedef apr_int32_t             apr_fileperms_t;
```

APR 中提供了两个函数用于实现从 APR 权限标志到 UNIX 系统标志位的相互转换。apr_unix_perms2mode 函数用于将 APR 定义转换为 Unix 定义，apr_unix_mode2perms 用于将 Unix 定义转换为 APR 定义。这两个函数都定义在 fileacc.c 中。从 Unix 转换至 APR 的过程无非如下：

```
if (mode & S_IXXXX)
    perms |= APR_XXXX;
```

而从 APR 转换为 Unix 过程无非如下：

```
if (perms & APR_XXXX)
    mode |= S_IXXXX;
```

4.2.3 文件打开方式

4.2.4 其余类型重定义

APR 中除了对上面的常量进行了重定义之外，它还对一些类型进行了重定义，不过这些类型都仅仅是使用 typedef 而已，非常简单，总结归纳如下：

1)、文件属性类型 apr_fileattrs_t

typedef apr_uint32_t apr_fileattrs_t;

2)、文件定位基准 apr_seek_where_t

typedef int apr_seek_where_t;

3)、文件访问权限 apr_fileperms_t

typedef apr_int32_t apr_fileperms_t;

4)、文件 i-node 结点编号 apr_ino_t

typedef ino_t apr_ino_t;

5)、文件所在设备号 apr_dev_t

typedef dev_t apr_dev_t;

4.3 文件描述

在 Unix 系统中，与文件关联的两个数据结构通常是两个：FILE 和 stat。前者通常称之为文件句柄，而后者则通常称之为文件的状态信息，用于描述文件的内部信息。APR 中，与之对应提供了两个封装数据结构 apr_file_t 和 apr_finfo_t，前者描述文件句柄信息，后者描述文件内部信息。

根据操作系统支持的不同，apr_file_t 了可以分为四个版本，不过我们仅仅介绍 Unix 版本，至于 Window 版本，我们会提及，而其余的 netware 和 OS/2 版本我们不打算做任何分析。在 Unix 系统中，apr_file_t 定义如下：

```
struct apr_file_t {
    apr_pool_t *pool;
    int filedes;
    char *fname;
    apr_int32_t flags;
    int eof_hit;
    int is_pipe;
    apr_interval_time_t timeout;
    int buffered;
    enum {BLK_UNKNOWN, BLK_OFF, BLK_ON } blocking;
```

```
    int ungetchar;    /* Last char provided by an unget op. (-1 = no char)*/
#ifndef WAITIO_USES_POLL
    /* if there is a timeout set, then this pollset is used */
    apr_pollset_t *pollset;
#endif
    /* Stuff for buffered mode */
    char *buffer;
    int bufpos;               /* Read/Write position in buffer */
    unsigned long dataRead;   /* amount of valid data read into buffer */
    int direction;            /* buffer being used for 0 = read, 1 = write */
    unsigned long filePtr;    /* position in file of handle */
#if APR_HAS_THREADS
    struct apr_thread_mutex_t *thlock;
#endif
};
```

该结构描述了一个文件的大部分的属性，也是整个文件 I/O 系统的核心数据结构之一。filedes 是文件的描述符；fname 则是打开的文件的名称；is_pipe 用以标记当前文件是否是管道文件；Unix 中在创建匿名管道的时候会生成一个管道文件，在管道中传输的数据实际上最终都保存在匿名文件中。对于这种临时的管道文件，它的 is_pipe 为 1；filePtr 是读写文件的时候文件内部的文件指针，通常情况下这个成员只有 seek 函数的时候才需要使用，由于 seek 需要给出当前文件的内部指针位置，因此在任何的文件读和写之后，我们都必须立即调整 filePtr 的值，以使它指向正确的位置。direction 则是记录了当前的操作类型，0 是读操作，1 是写操作；buffer 缓冲区用以保存从文件中读取的数据，dataRead 则用以记录从文件中读取到缓冲区中的有效的字节数；blocking 则是记录的读取的方式，一般允许两种，即阻塞和非阻塞。对于阻塞，那么读取将等待，直到文件中有新的数据或者读取超时，超时的时间由 timeout 决定。另外，如果支持多线程，那么为了保证线程访问安全性，在可能出现互斥的数据结构中都要额外的增加内部互斥锁。文件结构的内部互斥锁由变量 thlock 决定。任何人访问该数据结构之前都必须先获取该互斥锁，同时访问结束后该互斥锁将被释放。

apache 中的文件与目录(2)

4.4 打开文件

文件打开应该是使用的最多的文件操作了，任何文件在使用之前都必须首先打开。ANSI C 标准库和 Unix 系统库函数都提供对“打开文件”这个操作语义的支持。他们提供的接口很相似，参数一般都为“文件名+打开标志位+权限标志位 ”，Apache 中提供了 apr_file_open 函数来支持文件打开操作，该函数只是在原有的标准库的基础上进行了少许的封装。 apr_file_open 无法忽略习惯的巨大力量，它提供了与 ANSI C 以及 Unix 系统库函数类似的接口如下：

```
APR_DECLARE(apr_status_t) apr_file_open(apr_file_t **new,
                                        const char *fname,
                                        apr_int32_t flag,
                                        apr_fileperms_t perm,
                                        apr_pool_t *pool);
```

其中 fname、flag 和 perm 三个参数与普通的 open 函数相同，fname 分别表示打开的文件的路径名 称，可以是相对路径，也可以是绝对路径。每个封装都有自定义的一些标志宏，这里也不例外，flag 和 perm 参数都需要用户传入 APR 自定义的一些宏组 合，不过由于这些宏的可读性都

很好，不会成为你使用过程的绊脚石。flag 是打开文件的标志，包括可读可写，Apache 中打开标志可以概括为下面的几 种：

打开标志	含义	值
APR_READ	打开文件为只读	0x00001
APR_WRITE	打开文件为只写	0x00002
APR_CREATE	如果文件不存在，创建一个新的文件	0x00004
APR_APPEND	允许将内容追加到文件的末尾，而不是重新覆盖	0x00008
APR_TRUNCATE	如果文件存在，将其长度设置为 0	0x00010
APR_BINARY	打开的不是文本文件，而是二进制文件，在 UNIX 上，该标志将被忽略	0x00020
APR_BUFFERED	缓存数据，默认情况下不进行数据缓存	0x00040
APR_EXCL	return error if APR_CREATE and file exists	0x00080
APR_DELONCLOSE	关闭后删除该文件	0x00100
APR_XTHREAD	该标志依赖于具体的平台，用以确保文件在跨线程访问中的安全	0x00200
APR_SHARELOCK	平台依赖标记，用以支持上层的读写所访问从而支持跨进程或者跨机器访问	0x00400
APR_FILE_NOCLEANUP	通知系统不要在内池中注册文件清除函数，因此当内存池被销毁的时候，apr_file_t 中的 apr_os_file_t 句柄将不会被销毁	0x00800
APR_SENDFILE_ENABLED	Open with appropriate platform semantics for sendfile operations. Advisory only, apr_socket_sendfile does not check this flag.	0x01000

perm 则是用以记录文件的存取权限，通常情况下是一个整数，比如 0x777，0x666 等等。Apache 中支持的权限如表 4-3 所示。

文件的读取结果由指针 apr_file_t 返回。整个 open 过程可以分为四部分：

1)、“打开标志位”转换；

如前所述，APR 定义了自己的“文件打开标志位”，所以在 apr_file_open 的开始需要将这些专有的“文件打开标志位”转换为 Unix 平台通用的“文件打开标志位”，对应的转换表如下：

Apache 打开标志	UNIX 库函数内部标志
APR_READ&&APR_WRITE	O_RDWR
APR_READ	O_RDONLY
APR_WRITE	O_WRONLY
APR_CREATE	O_CREAT
O_CREAT&&APR_EXCL	O_CREAT&O_EXCL
APR_EXCL&&!APR_CREATE	允许组成员读取
APR_APPEND	O_APPEND
APR_TRUNCATE	O_TRUNC
O_BINARY	O_BINARY
APR_BUFFERED&&APR_XTHREAD	允许其余成员读取

2)、权限标志位”转换；

同1)理，专有的APR“权限标志位”需要转换为Unix平台通用的“权限标志位”；转换使用函数apr_unix_perms2mode实现，转换根据表4-2的对应关系实现。函数返回的权限最终传递给open函数的标志位。
(3)、调用Unix的本地API打开文件；
(4)、设置apr_file_t变量相关属性值；
APR 文件I/O封装支持非阻塞I/O带超时等待以及缓冲I/O，默认情况下为阻塞的，即BLK_ON。APR文件的另外一个特殊之处就是支持缓冲特性。由于磁盘读取的速度瓶颈，使得频繁的从磁盘读取文件在一定程度 上会影响执行效率，因此为了提高读取效率，APR支持文件的缓存读写，即开辟一块大的缓冲内存区，用以保存实际从磁盘中读取得数据，这样用户每次就不需要 读写磁盘，而只要读写内存，通过这种缓冲策略，可以改善一定的性能。是否缓冲可通过“文件打开标志位APR_BUFFERED”设置。一旦设置为缓冲读 写，则apr_file_open会在pool中开辟大小为APR_FILE_BUFSIZE(4096)的缓冲区供使用。
创建函数中一个比较重要的就是内存池中apr_file_t类型的清除函数注册。当内存池被销毁的时候，对于所有的apr_file_t类型将调用apr_unix_file_cleanup函数进行清除。与创建类似，清除也包括四方面的工作：
(1)、
(2)、关闭文件描述符filedes，如果文件的打开标志是APR_DELONCLOSE，那么在关闭之后还得将该文件删除；如果文件可能跨进程使用，那么还得销毁互斥锁。
(3)、

4.5 文件读取

4.5.1 普通文件读取

文件的读写操作定义在readwrite.c中。函数的原型与标准的接口非常类似：
APR_DECLARE(apr_status_t) apr_file_read(apr_file_t *thefile, void *buf, apr_size_t *nbytes)
thefile是需要读取的文件的描述结构，而buf是文件读取保存的缓冲区。nbytes则是从文件中需要读 取的字节数。Apache中的文件读取在内部分为两种机制：支持读写缓存的和不支持读写缓存的，读写是否需要支持缓存，有apr_file_t内部的 buffered成员决定，buffered=1表示内部必须支持缓存，否则不需要使用缓存。
我们首先分析最普通的内部不使用缓存的读写情况，下面是读取的核心代码：

```
{
    apr_ssize_t rv;
    apr_size_t bytes_read;

    if (*nbytes <= 0) {
        *nbytes = 0;
        return APR_SUCCESS;
    }

    {
        bytes_read = 0;
        if (thefile->ungetchar != -1) {
```

```
            bytes_read = 1;
            *(char *)buf = (char)thefile->ungetchar;
            buf = (char *)buf + 1;
            (*nbytes)--;
            thefile->ungetchar = -1;
            if (*nbytes == 0) {
                *nbytes = bytes_read;
                return APR_SUCCESS;
            }
        }

        do {
            rv = read(thefile->filedes, buf, *nbytes);      u
        } while (rv == -1 && errno == EINTR);
#ifdef USE_WAIT_FOR_IO
        if (rv == -1 &&
            (errno == EAGAIN || errno == EWOULDBLOCK) &&
            thefile->timeout != 0) {
            apr_status_t arv = apr_wait_for_io_or_timeout(thefile, NULL, 1);
            if (arv != APR_SUCCESS) {                v
                *nbytes = bytes_read;
                return arv;
            }
            else {
                do {
                    rv = read(thefile->filedes, buf, *nbytes);
                } while (rv == -1 && errno == EINTR);
            }
        }
#endif
        *nbytes = bytes_read;
        if (rv == 0) {
            thefile->eof_hit = TRUE;                    w
            return APR_EOF;
        }

        if (rv > 0) {
            *nbytes += rv;                              x
            return APR_SUCCESS;
        }
        return errno;
    }
}
```

读取的主要操作就是调用标准的文件读操作 read，文件中读取的数据直接保存到输出缓冲区 buf 中，确实没有进行任何的内部缓存。

第一次读取结束后，函数将根据读取的返回值做进一步的处理：

■ **返回 EINTR，见代码 u 处**

返回 EINTR 意味着读取操作被中断信号无意打断，而不是读取操作本身出现任何问题，因此此时必须无条件重新启动读操作，这就是将读操作放在循环中的原因之一，代码见。

■ **返回 EAGAIN 或者 EWOULDBLOCK，见代码 v 处**

读操作发生失败的另外一个可能性就是文件暂时不可用，因此此时应该稍等片刻再尝试，这种情况通常会返回 EAGAIN 错误码，对于 GNU C 库而言，EWOULDBLOCK 与 EAGAIN 的含义完全相同，只是换了一个名字而已，不过对于早期的版本可能存在差异。因此稳妥的做法就是同时检测 EAGAIN 和 EWOULDBLOCK 错误码。

一般下面的两种情况可能返回这两个错误码：

(1)、对非阻塞模式的对象进行某个会阻塞的操作可能会返回该错误码，再次做同样的操作就会阻塞直到某种条件使它可以读写。

(2)、某个资源上的故障使操作不能进行。例如 fork 有可能返回着个错误，这也表示这个故障可以被克服，所以 你的程序可以以后尝试这个操作。停几秒让其他进程释放资源然后再试也许是个好主意。这种故障可能很严重并会影响整个系统，所以通常交互式的程序会报告用户 这个错误并返回命令循环。

如果 apr_file_t 结构中设置了超时时间 timeout，同时发现必须重新读取(返回码为 EAGAIN 或者 EWOULDBLOCK)，那么 Apache 将调用 apr_wait_for_io_or_timeout 函数等待重新读取文件。apr_wait_for_io_or_timeout 函数内部使用了 I/O 的多路复用技术 poll，具体的细节我们在网络 I/O 章节描述。如果在给定的 超时时间内，文件还是不允许进行读取操作，那么此时函数将直接返回，否则函数将重新调用下面的语句：

```
do {
    rv = read(thefile->filedes, buf, *nbytes);
} while (rv == -1 && errno == EINTR);
```

由于此时文件的状态处于肯定允许进行操作状态，因此不再需要进行额外的异常处理，唯一的就是防止被信号无意打断。

■ **到达文件末尾(rv==0)，见代码 w**

如果 read 函数返回 0，此时意味着已经读取到整个文件，此时设置文件的 eof_hit=1，至此整个文件读取结束。

■ **读取非空字节(rv>0)，见代码 x**

此时累计读取的总字节数目。

上面的几个文件读取的处理步骤是大多数文件读取的标准的读取操作，因此非常的好理解。

事实上，对于文件读取分析的重点并不是上面的普通的读取，而是使用缓存的读取。一般情况下 I/O 操作是相当耗费 时间的，因此仅仅一次从文件中直接读取数据保存到缓冲区中所花的时间可能允许忽略不计，但是如果文件读取操作非常的频繁的话，那么这将无疑是一个不小的时 间耗费，甚至可能是性能瓶颈。因此为了有效地提高读写性能，Apache 提供了缓存读取的策略。所谓缓存读取，就是先把文件中的数据读取到一个缓存中，然 后以后再次需要读取数据的时候，不再从文件本身去读取，而是从缓存中去读取。通过这种策略使文件 I/O 读取变成内存读取，从而提高了读取速度。而写入时候 也是类似，先写入到缓存中，然后在一次写入磁盘中，从而将多次磁盘写入变为多次内存写入。apr_file_t 结构中的 buffer 结构的作用正是这个目 的。缓存读取的策略可以用下面的示意图描述：

使用了缓冲区后，文件操作者将通过缓冲区与磁盘文件打交道。整个缓冲区的实现机制，我们给出下面的一个更详细的 图片，通过图示，我们可以理解 apr_file_t 中一些很晦涩的成员变量，事实上这些成员变量仅仅是配合缓冲区机制而使用的，而且仅仅使用缓冲区的时候 才起作用。

上面的图示被我们分为了三个层次：最底层的是进行文件读写的用户，它拥有自定义的缓冲区，我们称之为用户缓冲 区；中间的是 apr_file_t 结构内的缓冲区，它用以保存读写缓存数据。事实上用户总是跟这个层次的缓冲区打交道；最上层的则是磁盘文件。在各个层次 中我们用”/////”表示模拟当前的缓冲区大小。

从图示中我们可以看到至少存在四种缓冲区长度：

1)、磁盘文件的实际长度。当使用 read 在文件中读写的时候，文件内部会维护一个内部指针指示当前的读取位置，apr_file_t 结构中使用 filePtr 模拟该内部指针，因此 filePtr 总是指向实际文件内部的当前读取位置。

2)、文件缓冲区中现有数据的长度，它的大小由 apr_file_t 结构内的 dataRead 指示。通常情况下，dataRead 的大小与 filePtr 指向的位置偏移相等。

3)、最终用户在文件缓冲区中读取的数据的当前指针，由 apr_file_t 结构内的 bufpos 指针指示

4)、用户缓冲区的长度，其最新读写位置由 pos 指示。

基于缓冲区，整个读取操作的流程发生了根本的变化。任何读取首先尝试从文件缓冲区中读取，如果请求读取的长度在 文件缓冲区的长度范围之内，那么直接返回数据。如果需要读取的内容超出了文件缓冲区的范围，那么我们还必须再去实际的磁盘文件中去读取，并返回，同时更新缓存区中的数据。

```
    if (thefile->buffered) {
        char *pos = (char *)buf;
        apr_uint64_t blocksize;
        apr_uint64_t size = *nbytes;

#if APR_HAS_THREADS
        if (thefile->thlock) {
            apr_thread_mutex_lock(thefile->thlock);
        }
#endif
```

如果支持多线程的操作，那么在对文件进行操作之前必须互斥量锁定，确保操作的安全性。同样在读取结束后还必须是 unlock 该互斥变量。

```
        if (thefile->direction == 1) {
            apr_file_flush(thefile);
            thefile->bufpos = 0;
            thefile->direction = 0;
            thefile->dataRead = 0;
        }
```

```
        rv = 0;
        if (thefile->ungetchar != -1) {
            *pos = (char)thefile->ungetchar;
            ++pos;
            --size;
            thefile->ungetchar = -1;
        }
        while (rv == 0 && size > 0) {
            if (thefile->bufpos >= thefile->dataRead) {
                int bytesread = read(thefile->filedes, thefile->buffer,
APR_FILE_BUFSIZE);
                if (bytesread == 0) {
                    thefile->eof_hit = TRUE;
                    rv = APR_EOF;
                    break;
                }
                else if (bytesread == -1) {
                    rv = errno;
                    break;
                }
                thefile->dataRead = bytesread;
                thefile->filePtr += thefile->dataRead;
                thefile->bufpos = 0;
            }

            blocksize = size > thefile->dataRead - thefile->bufpos ? thefile-
>dataRead - thefile->bufpos : size;
            memcpy(pos, thefile->buffer + thefile->bufpos, blocksize);
            thefile->bufpos += blocksize;
            pos += blocksize;
            size -= blocksize;
        }
```

while 循环读取的核心代码就位于 while 循环中。dataRead 是实际读取到缓冲区中的数据的长度当 然它肯定小于或者等于实际的文件的总长度。bufpos 则是用户读取的缓冲区中的实际字节数目，当然 bufpos 肯定也是小于或者等于 dataRead 的。它们的关系如上图所示。用户如果从缓冲区中读取数据的话它最多只能读取到 dataRead 字节的长度。当用户从缓冲区中读取数据的时候，缓冲区的内部 的指针变量 bufpos 不断往后移动。

函数将根据 bufpos 的位置做不同的策略：

(1)、起始的时候，bufpos=0，缓冲区中的数据为 dataRead，用户请求的数据为 size 大小。此时将开始尝试读取缓冲区中的数据。如果请求的字节小于缓冲区的 dataRead 长度，那么直接将缓冲区中 size 大小的数据拷贝到输出缓冲区 pos 中。这 种情况是最简单的一种。

由于 filepos 为 0，这时候执行的代码将演变为如下：

```
            blocksize = size;//size > thefile->dataRead -0 ? thefile->dataRead:
size;
            memcpy(pos, thefile->buffer + 0, blocksize);
            thefile->bufpos += blocksize;
            pos += size;
            size -= size;
```

(2)、当 bufpos=0，即开始读取的时候，如果用户请求的数据长度超出了 dataRead 的长度，那么此时用户第一次读取只能读取到 dataRead 的数据，还剩余 size-dataRead 需要从文件中读取。因此，第一次的读取演变为如下的代码：

```
            blocksize = thefile->dataRead;//size > thefile->dataRead - 0 ? thefile-
>dataRead - 0 : size;
            memcpy(pos, thefile->buffer + 0, blocksize);
            thefile->bufpos += blocksize;
            pos += thefile->dataRead;
            size -= thefile->dataRead;
```

此时各个指针的位置如下图所示意

从上图可以看到，第一次读取之后，各个指针都指向相应的缓冲区的末尾，pos 指向用户缓冲区，随后的数据直接从 pos 位置往后拷贝；bufpos 指向文件缓冲区的末尾。由于此时尚有 size-dataRead 大小的数据尚未读取，因此，函数将从 thefile->filedes 中去读取：

```
            if (thefile->bufpos >= thefile->dataRead) {
                int bytesread = read(thefile->filedes, thefile->buffer,
APR_FILE_BUFSIZE);
                if (bytesread == 0) {
                    thefile->eof_hit = TRUE;
                    rv = APR_EOF;
                    break;
                }
                else if (bytesread == -1) {
                    rv = errno;
                    break;
                }
                thefile->dataRead = bytesread;
                thefile->filePtr += thefile->dataRead;
                thefile->bufpos = 0;
            }
```

但实际上，当 Apache 再次到磁盘文件中去读取的时候，它并不会吝啬的仅读取 size-dataRead 大小，相反它会调用 read 函数一次性的从文件中读取 **APR_FILE_BUFSIZE(4K)** 大小的数据到文件缓冲区中，然后设置 dataRead 为新的实际的读取字节长度。

```
        *nbytes = pos - (char *)buf;
        if (*nbytes) {
```

```
            rv = 0;
        }
#if APR_HAS_THREADS
        if (thefile->thlock) {
            apr_thread_mutex_unlock(thefile->thlock);
        }
#endif
        return rv;
    }
```

在apr_file_read函数的基础之上，APR提供了额外的几个辅助函数：

■ APR_DECLARE(apr_status_t) apr_file_getc(char *ch, apr_file_t *thefile);
用以从文件中获取一个字符

■ APR_DECLARE(apr_status_t) apr_file_read_full(apr_file_t *thefile, void *buf,
apr_size_t nbytes,
apr_size_t *bytes_read)
apr_file_read函数通常用于读取指定字节的数据，并且永远不会超过这个指定值，但是实际的读取字节数则可以少于这个值。

而apr_file_read_full与apr_file_read非常类似，主要的区别就是，对于指定的字 节数nbytes，函数必须全部读取完毕才肯罢休，任何时候如果发现读取得字节数小于nbytes，函数都将继续等待。这种读取方式我们称之为“全字节读 取方式”。

4.5.2 字符串读取

4.5.3 文件写入

文件写入通过函数apr_file_write实现，与标准的写入函数相同，它也具有三个参数：
APR_DECLARE(apr_status_t) apr_file_write(apr_file_t *thefile, const void *buf,
apr_size_t *nbytes)
thefile是写入的文件描述符，buf是写入的源目标字符串，nbytes是需要写入的字节总数目。

与读取类似，写入也分为缓冲写入和非缓冲写入两种。对于非缓冲写入，所有的数据是直接写入到文件中，而对于缓冲写入，所有的数据先写入到文件缓冲区中，然后再从缓冲区中写入到文件中。

对于非缓冲写入的核心代码如下所示：

```
        apr_size_t rv;

        do {
            rv = write(thefile->filedes, buf, *nbytes);
        } while (rv == (apr_size_t)-1 && errno == EINTR);
#ifdef USE_WAIT_FOR_IO
        if (rv == (apr_size_t)-1 &&
            (errno == EAGAIN || errno == EWOULDBLOCK) &&
            thefile->timeout != 0) {
            apr_status_t arv = apr_wait_for_io_or_timeout(thefile, NULL, 0);
```

```
            if (arv != APR_SUCCESS) {
                *nbytes = 0;
                return arv;
            }
            else {
                do {
                    do {
                        rv = write(thefile->filedes, buf, *nbytes);
                    } while (rv == (apr_size_t)-1 && errno == EINTR);
                    if (rv == (apr_size_t)-1 &&
                        (errno == EAGAIN || errno == EWOULDBLOCK)) {
                        *nbytes /= 2;
                    }
                    else {
                        break;
                    }
                } while (1);
            }
        }
#endif
        if (rv == (apr_size_t)-1) {
            (*nbytes) = 0;
           return errno;
        }
        *nbytes = rv;
        return APR_SUCCESS;
```

对于缓冲写入，其实现代码如下所示：

```
    apr_size_t rv;

    if (thefile->buffered) {
        char *pos = (char *)buf;
        int blocksize;
        int size = *nbytes;

        if ( thefile->direction == 0 ) {
            apr_int64_t offset = thefile->filePtr - thefile->dataRead + thefile-
>bufpos;
            if (offset != thefile->filePtr)
                lseek(thefile->filedes, offset, SEEK_SET);                  u
            thefile->bufpos = thefile->dataRead = 0;
            thefile->direction = 1;
        }
```

```
        rv = 0;
        while (rv == 0 && size > 0) {
            if (thefile->bufpos == thefile->bufsize)   /* write buffer is full*/
                rv = apr_file_flush(thefile);                v

            blocksize = size > thefile->bufsize - thefile->bufpos ?
                        thefile->bufsize - thefile->bufpos : size;
            memcpy(thefile->buffer + thefile->bufpos, pos,
blocksize);
            thefile->bufpos += blocksize;                   w
            pos += blocksize;
            size -= blocksize;
        }

        return rv;
    }
```

在分析文件缓冲写之前，我们先看一下实际的文件系统的读写情况。在文件系统内部始终维持一个内部位置指针，该指 针随着当前读取和写入的位置的改变而不停的改变，同时任何一次读取和写入都是在内部指针指向的位置基础上进行的，因此如果在写之前，文件内部的指针偏移为offset，则写入的数据将插入到offset之后。

这种情况对于APR文件缓冲而言也必须实现同样的效果。不过，如果文件thefile在写之前刚被读过，则存在两个offset偏移位置：

1)、filePtr。任何时候，实际文件内部的指针用filePtr进行模拟指示，它始终与内部指针保持同步。

2)、bufpos。该位置指示用户实际已经读取的缓冲区位置，它在实际文件内部的偏移量为filePtr-(dataRead-bufpos)。

那么在写入时候，新的数据是插入到filePtr之后还是filePtr-(dataRead-bufpos) 之后呢？答案只有一个：一切从用户的角度出发。对于最终用户而言，它能够观察到的现象是：数据刚被读取到bufpos位置，至于文件缓冲区的文件长度多少 用户根本无法了解到。因此对于用户而言文件的偏移量应该是filePtr-(dataRead-bufpos)，而不是filePtr，即使实际的内部 文件指针偏移量确实是filePtr。

因此如果filePtr和filePtr-(dataRead-bufpos)不一致的时候，必须调整文件内部的实际偏移指针为filePtr-(dataRead-bufpos)，以保证正确的写入位置。另外必要的工作就是初始化写入缓冲区，这正是代码u所实现的任务。

一旦定位完毕，那么就可以进行数据写入了。用户缓冲区中所有的数据首先都立即写入到文件缓冲区中，写入的时候可能发生的情况包括下面几种：

1)、缓冲区已满。apr_file_t中缓冲区的大小由bufsize确定，默认大小为4K。一旦缓冲区已 满，那么缓冲区中的数据将使用apr_file_flush一次性写入到实际的文件中，同时再次初始化缓冲区。Apr_file_flush函数的实现非 常简单，顺便一看：

```
APR_DECLARE(apr_status_t) apr_file_flush(apr_file_t *thefile)
{
    if (thefile->buffered) {
        if (thefile->direction == 1 && thefile->bufpos) {
            apr_ssize_t written;
```

```
            do {
                written = write(thefile->filedes, thefile->buffer, thefile-
>bufpos);
            } while (written == -1 && errno == EINTR);
            if (written == -1) {
                return errno;
            }
            thefile->filePtr += written;
            thefile->bufpos = 0;
        }
    }
    return APR_SUCCESS;
}
```

函数中所作的无非就是不断调用 write 写入。写入成功后 filePtr 和缓冲区的开始位置，这两个步骤都被隐藏在该函数中，因此你在写入函数中看不到也就不奇怪了。

2)、缓冲区的空闲空间足够写入，即 size <= thefile->bufsize - thefile->bufpos。此时，直接调用 memcpy 将用户缓冲区中的数据拷贝到文件缓冲区中，并调整用户缓冲区 pos 和文件缓冲区 bufpos 的位置。

3)、用户要求写入的数据空闲空间不够一次写入，此时将分为多次写入。如果写入过程中空间已满，使用 1)的方法，否则使用 2)的方法。

在该函数的基础之上，APR 还提供了一些辅助扩充函数：

■ APR_DECLARE(apr_status_t) apr_file_putc(char ch, apr_file_t *thefile);

用以向文件中写入一个字符。

■ APR_DECLARE(apr_status_t) apr_file_puts(const char *str, apr_file_t *thefile)

用以向文件中写入一个字符串。

■ APR_DECLARE(apr_status_t) apr_file_writev(apr_file_t *thefile, const struct iovec *vec,

apr_size_t nvec, apr_size_t *nbytes)

批量文件写入。需要批量写入的数据保存在 vec 中。如果系统中定义了 writev 函数，则调用 writev 函数批量写入。如果系统不支持 writev 函数，那么如果要写入整个 iovec 数组，可以使用两种变通策略：

第一种就是逐一遍历 iovec 数组中的每一个元素并将其写入到文件中。这种做法存在一个问题，就是原子性写入。Writev 函数写入所有数据的时候是保持原子性的。而迭代则明显无法保持这种特性。

另一种可选策略就是首先将 iovec 数组中的数据集中写入到一个缓冲区中，然后再将该缓冲区写入到文件中。这种策略也是不合理的，因此你根本就不知道一个 iovec 数组中会包含多少数据你也就无法确定缓冲区的大小。

为了保持 writev 的真正语义，最合理的策略就是仅写入 iovec 数组中的第一个数据，即 vec[0]。Callers of file_writev() must deal with partial writes as they normally would. If you want to ensure an entire iovec is written, use apr_file_writev_full()。

■APR_DECLARE(apr_status_t) apr_file_write_full(apr_file_t *thefile,

const void *buf,

apr_size_t nbytes,
apr_size_t *bytes_written)

与 apr_file_read_full 类似，该函数使得写入的字符串必须达到指定的 nbytes，任何时候如果写入的字符数小于 nbytes，函数都将等待。这种方式我们称之为“全字节写入方式”。

¢ APR_DECLARE(apr_status_t) apr_file_writev_full(apr_file_t *thefile,
const struct iovec *vec,
apr_size_t nvec,
apr_size_t *bytes_written)

apr_file_writev_full 是 apr_file_writev 函数的全字节写入方式的版本。该函数定义在 fullrw.c 中。

文件打开、文件读取以及文件写入是最普通三种文件操作，也是 APR 中使用的最多的操作，APR 中相当一部分的文件操作函数都是基于这三个函数构建起来的，最典型的就是文件内容拷贝和追加。在接下来的部分我们在分析文件内容拷贝和追加的时候更应该关注的是这三个函数的使用

Apache 中的挂钩剖析(1)

5.5 挂钩(HOOK)

5.5.1 为什么引入挂钩

在 Apache1.3 版本中，对 HTTP 请求的处理包括若干个固定阶段，比如地址转换阶段、身份确认阶段、身份认证阶段、权限确认阶段、MIME 类型识别阶段等等，这也意味着 Apache1.3 中的挂钩数目是有限的，固定的。这个反映在模块结构中就是针对每个 HOOK 都对应一个函数指针。比如如果需要检查用户的身份是否合法则只需要调用 ap_check_user_i；如果需要核查用户的权限是否满足则只需要调用 auth_checker；而如果需要记录日志，则只需要调用 logger 函数即可。这种对请求的处理策略非常清晰明了。不过这种方法也有明显的局限性，首先就是所有的模块都维护所有的挂钩，但对于某个模块而言只有很少的几个挂 钩会被使用，比如 ap_check_user_id 挂钩可能只有安全模块才用到，模块中不使用的挂钩都被置为 NULL；最重要的问题是很难增加新的挂钩。 如果需要增加一个新的挂钩，就必须修改 module 结构，这涉及到所有的模块的修改，工作量很大，而且需要重新进行编译，因此这种策略使得模块的扩展性能极差。

为了使得 Apache 2.0 版本为了能够更具模块化，更具扩展性，apache 采取了更加灵活的处理策略，即“模块自行管理”策略，即挂钩(Hooks)的概念，Hooks 的 使用使得函数从静态的变为动态的，模块编写者可以通过 Hooks 自行增加处理句柄，而不需要所有的模块都千篇一律。因此每次增加新函数，唯一必须修改的就 是增加函数的模块而已。

为了对挂钩有大体的了解，我们首先来看一下 Apache2.0 的 HTTP 请求处理流程。

从大的方面来看，Apache 对 HTTP 的请求可以分为连接、处理和断开连接三个阶段。从小的方面而言，每个阶段又可以分为更多的子阶段。比如对 HTTP 的请求，我们可以进一步划分为客户身份验证、客户权限认证、请求校验、URL 重定向等阶段，每一个 阶段调用相应的函数进行处理。在 Apache 中，这些子阶段可以用术语“挂钩(HOOK)”来描述。Apache 中对请求的处理过程实质上就是依次调用一 系列挂钩的过程，不过由于 HTTP 请求类型的不同，它们所对应的挂钩数目和类型也不尽相同。对于典型的 HTTP 请求，有一个默认的挂钩调用顺序，你可以按 照这个默认的顺序进行调用，也可以不遵守这个顺序，你可以根据自己的情况调整调用顺序。整个 HTTP 的请求可以用下图来描述：

在上面的图示中，存在六种不同的挂钩，对于请求 a，其指需要调用 Hook2.、1；而对于请求 b，则需要调用 1、6、5；请求 c 则需要调用 1、4、3、6 四个挂钩。

在 Apache 中，挂钩总是和挂钩函数联系在一起的。挂钩是用来表示在处理 HTTP 请求中一组类似的操作，与之 对应，挂钩函数就是操作函数。不过即使挂钩相同，对应的挂钩函数也未必相同。举个简单的例子，配置文件模块和虚拟主机模块都需要身份验证挂钩来确认访问者 能否访问相应的资源，但两个模块的验证方法则差别很大。因此一个挂钩同时是和一组挂钩函数联系在一起的。因此请求处理过程中，调用挂钩的时候实际上就是调 用挂钩函数组。调用过程可以用下图示意。Apache 对指定挂钩的挂钩函数调用有两种，一种就是将挂钩函数组中的每个函数都调用一次，比如图中的挂钩 4； 另一种就是从前往后调用，一旦找到合适的就停止继续调用，图中 1、2、3 就是这种情况。

通过挂钩机制，你可以自行修改服务器的行为，比如修改挂钩函数或者增加挂钩函数，甚至增加挂钩。不过增加挂钩只是 2.0 才提供的功能。

5.5.2 声明挂钩

Apache 中关于挂钩的实现大部分是通过宏来实现的，而且这些宏大多数非常复杂。挂钩的实现主要定义在文件 apr_hook.h 和 apr_hook.c 中，另外在 config.c 中也有部分定义。

Apache 中对挂钩的使用总是从定义一个挂钩开始的，在 Apache 中声明一个挂钩，总是通过宏

```
#define AP_DECLARE_HOOK(ret,name,args) \
    APR_DECLARE_EXTERNAL_HOOK(ap,AP,ret,name,args)
```

来实现的。在该宏中，ret 是定义的挂钩 的返回类型；而 name 则是定义的挂钩的名称；args 则是挂钩函数需要的额外参数，通常是以 bracket 形式出现的。例如下面的语句就声明了一个返回 类型为整数，函数名称为 do_something，函数参数为(request_rec *r，int n)的挂钩：

```
AP_DECLARE_HOOK(int , do_something , (request_rec *r , int n))
```

不过 AP_DECLARE_HOOK 内部则是调用的 APR_DECLARE_EXTERNAL_HOOK 宏。该宏定义如下：

```
#define APR_DECLARE_EXTERNAL_HOOK(ns,link,ret,name,args) \
typedef ret ns##_HOOK_##name##_t args; \
link##_DECLARE(void) ns##_hook_##name(ns##_HOOK_##name##_t *pf, \
                                      const char * const *aszPre, \
                                      const char * const *aszSucc, int
nOrder); \
link##_DECLARE(ret) ns##_run_##name args; \
APR_IMPLEMENT_HOOK_GET_PROTO(ns,link,name); \
typedef struct ns##_LINK_##name##_t \
{ \
```

```
        ns##_HOOK_##name##_t *pFunc; \
        const char *szName; \
        const char * const *aszPredecessors; \
        const char * const *aszSuccessors; \
        int nOrder; \
    } ns##_LINK_##name##_t;
```

从上面的定义可以看出，该宏定义冗长，而且事实上，Apache 中的关于挂钩定义的每个宏都是很长的。分析颇为费力。在该宏中出现最多的符号当之不让的就是“##”。##宏主要用来实现连接前后的字符串。

APR_DECLARE_EXTERNAL_HOOK 宏展来后实现了五个子功能的定义，这几个功能我们分别介绍，其所用的示例则是配置模块的 post_config 挂钩，在 Http_config.中 Apache 定义了 post_config 挂钩如下：

AP_DECLARE_HOOK(int,post_config,(apr_pool_t *pconf,apr_pool_t *plog, apr_pool_t *ptemp,server_rec *s))，该宏进一步替换展开后为

APR_DECLARE_EXTERNAL_HOOK(ap,AP,int,post_config, (apr_pool_t *pconf,apr_pool_t *plog, apr_pool_t *ptemp,server_rec *s))，在该宏中：

(1)、typedef ret ns##_HOOK_##name##_t args;

该宏主要用来定义对应的 post_config 的挂钩的执行函数原型，比如 post_config 挂钩执行函数原型定义则是：

```
typedef int ap_HOOK_post_config_t (apr_pool_t *pconf,apr_pool_t *plog,apr_pool_t *ptemp,server_rec *s);
```

(2)、link##_DECLARE(void) ns##_hook_##name(ns##_HOOK_##name##_t *pf, \
const char * const *aszPre, \
const char * const *aszSucc, int nOrder); \

该宏则用来定义指定挂钩的函数原型，其中 ns##_HOOK_##name##_t *pf 则是前面定义了对应于该挂钩的执行函数指针，这个函数最终会加入到在请求处理期间的指定时刻进行调用的列表中，而 aszPre 和 aszSucc 在形 式和功能上都很相近，都是以 NULL 为结束符的字符串数组，比如{“mod_mime.c”,NULL}。aszPre 数组会为这个挂钩规定必须在这个函 数之前调用的函数模块，而 aszSucc 则是规定必须在这个函数之后进行调用的函数模块。上面的 post_config 挂钩其定义展开后如下：

```
AP_DECLARE(void) ap_hook_post_config(ap_HOOK_post_config* pf;
    const char * const *aszPre,
        const char * const *aszSucc,
        int nOrder);
```

函数的第四个参数 nOrder 则是该挂钩的综合排序参数，整数类型。该整数代表了这个函数与所有的 其余同一阶段的其他函数相比较时候的综合位置。如果这个数值越低，则这个挂钩函数将在列表中排列越靠前，因此也越早被调用。如果两个模块为它们的模块设定 相同的 nOrder，而且模块之间没有依赖关系，则它们的调用顺序则不确定。针对这个参数，Apache 提供了 5 个通用的 宏：APR_HOOK_REALLY_FIRST、APR_HOOK_FIRST、APR_HOOK_MIDDLE、APR_HOOK_LAST、APR_HOOK_REALLY_LAST，其定义如下：

```
#define APR_HOOK_REALLY_FIRST     (-10)
#define APR_HOOK_FIRST             0
#define APR_HOOK_MIDDLE            10
```

```
#define APR_HOOK_LAST             20
#define APR_HOOK_REALLY_LAST      30
```

从上面可以看出，nOrder 的值得范围 介于-10 到 30 之间。如果函数是在列表中是必须第一个得到的，则必须将其设置为 APR_HOOK_FIRST 或者 APR_HOOK_REALLY_FIRST；如果是必须最后一个调用的，则将其设置为 APR_HOOK_LAST 或者 APR_HOOK_REALLY_LAST。如果其调用次序无关紧要，则可以设置为 APR_HOOK_MIDDLE。

所有挂钩最终通过排序函数进行，这个我们在后面的部分介绍。

(3)、link##_DECLARE(ret) ns##_run_##name args; \

挂钩的定义最终是为了被调用，因此 Apache 中定义了对挂钩的调用函数。通常挂钩调用函数形式如下：ap_run_*hookname()*；上面的宏真是用来定义挂钩调用函数。

比如对于 post_config 挂钩而言，其对外的调用函数则是 ap_run_post_config()。

(4)、APR_IMPLEMENT_HOOK_GET_PROTO(ns,link,name); \

该宏展开后如下所示

```
#define APR_IMPLEMENT_HOOK_GET_PROTO(ns,link,name) \
link##_DECLARE(apr_array_header_t *) ns##_hook_get_##name(void)
```

该宏了用于返回挂钩访问函数原型，在模块外部可以调用改函数获得注册为该挂钩的所有函数。参数 ns 和 link 分别为挂钩函数名字空间前缀和联接申明前缀，一般情况为 ap 和 AP，name 为挂钩名。访问函数的原型一般为 AP_DECLARE(apr_array_header_t *) ap_hook_get_name(void)。如果对于 post_config 挂钩而言，该宏对应的则是 AP_DECLARE(apr_array_header_t *) ap_hook_get_post_config(void)。通过该函数，Apache 可以获取挂钩定义的原型。

```
(5)、typedef struct ns##_LINK_##name##_t \
{ \
    ns##_HOOK_##name##_t *pFunc; \
    const char *szName; \
    const char * const *aszPredecessors; \
    const char * const *aszSuccessors; \
    int nOrder; \
} ns##_LINK_##name##_t;
```

该宏定义了一个结构类型，用来保存挂钩的相关定义信息，比如挂钩的调用函数指针、挂钩名称等等，这些信息与在挂钩定义宏中的信息完全相同。比如对于 post_config 来说，其展开后的结果如下所示，结构中每个字段的含义前面已经说明，此处不再赘述。

```
typedef struct ap_LINK_post_config_t
{
    ap_HOOK_post_config_t *pFunc;
    const char *szName;
    const char * const *aszPredecessors;
    const char * const *aszSuccessors;
    int nOrder;
} ap_LINK_post_config_t;
```

为此，我们可以看到，尽管对外我们声明的仅仅是 AP_DECLARE_HOOK 宏，而内部却一口气实现了五个功能，每个功能彼此相关。

5.5.3 挂钩链声明(APR_HOOK_LINK)

在Apache中，系统定义了一定数量的 挂钩，这些挂钩总的来说可以分为两大类：启动挂钩和请求挂钩。启动挂钩是随着服务器启动进行调用的挂钩；而请求挂钩则是服务器处理请求时候进行调用的挂 钩。这两种挂钩的实质唯一区别就是它们在服务器中的平均调用频率以及它们在被调用时候服务器充当的角色。所有启动挂钩作为启动服务器的用户进行调用，通常 是指UNIX超级用户，而所有请求挂钩都是作为配置文件中指定的用户进行调用。

Apache中预定义了大量的挂钩，启动挂钩有pre_config、 post_config、open_logs以及child_init；请求挂钩包括pre_connection、 process_connection、create_request等等。Apache中声明的挂钩通常都是全局的，存在且仅存在一个。不过对应于挂钩 的调用函数则通常不止一个。比如对于挂钩post_config，在核心模块core.c中调用的挂钩函数是core_post_config，在 status模块mod_status.c中调用的挂钩函数是status_init，而在include模块mod_include.c中对应的挂钩函 数则是include_post_config。对于同一个挂钩，不同模块对应于其的处理函数各不相同，为了能够保存各个模块中对同一挂钩的使用信 息，Apache使用apr_array_header_t数组进行保存。

为此该宏的定义很简单，实际上无非就是声明了一个apr_array_header_t类型的数组，用来保存对应于某个挂钩的所有挂钩函数。比如APR_HOOK_LINK(post_config)展开后定义如下：

apr_array_header_t *link_post_config;

各个模块中对挂钩post_config 进行处理的信息都保存在数组link_post_config中，其中link_post_config数组中每个元素的类型都是 ap_LINK_post_config_t结构；当然不同的数组，其类型也不相同，不过形式上都是ap_LINK_XXXX格式的。关于挂钩的大部分信 息都由这个结构提供。

5.5.4 挂钩结构(APR_HOOK_STRUCT)

正如前面所说，对于每一个挂钩，Apache 都会定义一个apr_array_header_t数组来保存它的相关信息。通常，该数组定义在实现挂钩的文件中，比如在config.c文件 Apache实现了header_parser、pre_config、post_config、open_logs、child_init以及 handler、quick_handler、optional_fn_retrieve八个挂钩，那么该文件中就应该定义八个数组来保存它们的信息。一 旦定义挂钩数组，那么该数组将在整个Apache中保持唯一。当某个模块想要使用该挂钩的时候，其就向模块内对于该挂钩的处理结构压入数组。为了便于各模 块对数组的访问，原则上必须将数组声明为全局变量，这是最简单的实现方式。

不过Apache2.0中并不支持直接访问挂钩数组，因此你想直接将数据压入处理结构那是“妄想”。为此Apache2.0中引入了APR_HOOK_STRUCT宏。该宏定义如下：

```
#define APR_HOOK_STRUCT(members) \
static struct { members } _hooks;
```

该宏展开后实际上定义了一个限于模块内使用的结构_hook，该模块内实现的所有挂钩的对应数组都保存为_hook的成员。比如config.c中的APR_HOOK_STRUCT定义如下：

```
APR_HOOK_STRUCT(
        APR_HOOK_LINK(header_parser)
        APR_HOOK_LINK(pre_config)
        APR_HOOK_LINK(post_config)
        APR_HOOK_LINK(open_logs)
        APR_HOOK_LINK(child_init)
        APR_HOOK_LINK(handler)
        APR_HOOK_LINK(quick_handler)
        APR_HOOK_LINK(optional_fn_retrieve)
```

)

其展开后变为如下：

```
static struct{
            apr_array_header_t *link_header_parser;
            apr_array_header_t *link_pre_config;
            apr_array_header_t *link_post_config;
            apr_array_header_t *link_open_logs;
            apr_array_header_t *link_child_init;
            apr_array_header_t *link_handler;
            apr_array_header_t *link_quick_handler;
            apr_array_header_t *link_optional_fn_retrieve;
}_hook;
```

因此任何对挂钩数组的访问都必须通过_hook 来实现。比如我们在某个模块中使用挂钩 post_config，那么在数组中增加数据可以用下面的代码：

```
    ap_LINK_post_config_t *pHook;
    if (!_hooks.link_post_config) {
        _hooks.link_post_config = apr_array_make(apr_hook_global_pool, 1,
                                                 sizeof(ap_LINK_post_config_t));
        apr_hook_sort_register("post_config", &_hooks.link_post_config);
    }
    pHook = apr_array_push(_hooks.link_post_config);
    pHook->pFunc = …;
    pHook->aszPredecessors = …;
    pHook->aszSuccessors = …;
    pHook->nOrder = …;
    pHook->szName = …;
```

不过有几点必须注意的是：

(1)、_hook 在某个文件中如果定义，则只能定义一次。该文件中使用的定义的挂钩数组统统添加到该结构中，正所谓“大肚能容，容天下难容之事”。

(2)、_hook 的定义为 static，这意味这该结构实际上是模块内部的私有结构，外部模块无法直接访问_hook 结构，而且_hook 结构不仅仅在一个文件中出现，只要实现了 挂钩，按道理就应该有一个_hook 结构。尽管如此，由于 static 属性，它们相互之间“绝缘”，不会相互干扰。

(3)、当某个模块想使用某个挂钩，比如 post_config 的时候，其一不能直接访问 post_config 挂钩数组，二不能访问被屏蔽的模块内_hook，那么它应该怎么办呢？它只能使用挂钩注册函数。正如前面所述，挂钩注册函数通常形如 ap_hook_*name*。*比如模块需要使用 post_config 挂钩，其必须调用 ap_hook_post_config 进行注册。现在如果我们将在编写模块的时候遇到类似下面的代码，你就不会纳闷了：*

```
    static void register_hooks(apr_pool_t *p)
    {
        ap_hook_handler(status_handler, NULL, NULL, APR_HOOK_MIDDLE);
        ap_hook_post_config(status_init, NULL, NULL, APR_HOOK_MIDDLE);
    }
```

register_hooks 是模块 mod_status.c 中的挂钩注册函数，在该模块中，Apache 注册了两个挂钩：handler 和 status_init，分别对应的挂钩函数为 status_handler 和 status_init。

不过即使你查遍 Apache 的是所有的文件，你也不可能找到 ap_hook_handler 和 ap_hook_post_config 的函数声明和实现。不仅如此，所有的 ap_hook_*HOOKNAME* *形式的函数你都不可能找到你平常希望的函数声明和实现。那么 Apache 到底是在哪儿实现了这些挂钩函数呢？？一切答案都在宏 APR_IMPLEMENT_EXTERNAL_HOOK_BASE 里面。*

Apache 中的挂钩剖析(2)

5.5.5 挂钩函数 (APR_IMPLEMENT_EXTERNAL_HOOK_BASE)

从宏的名字我们就可以大体看出该宏实际上是实现了具体的挂钩注册函数，如果将其展开后我们会更加一目了然。该宏的定义也是冗长的很，如下所示：

```
    #define APR_IMPLEMENT_EXTERNAL_HOOK_BASE(ns,link,name) \
    link##_DECLARE(void) ns##_hook_##name(ns##_HOOK_##name##_t *pf, \
const char * const *aszPre, const char * const *aszSucc,int nOrder) \
    { \
        ns##_LINK_##name##_t *pHook; \
        if(!_hooks.link_##name) \
         { \

_hooks.link_##name=apr_array_make(apr_hook_global_pool,1,sizeof(ns##_LINK_##name##_
t)); \
          apr_hook_sort_register(#name,&_hooks.link_##name); \
          } \
        pHook=apr_array_push(_hooks.link_##name); \
        pHook->pFunc=pf; \
        pHook->aszPredecessors=aszPre; \
        pHook->aszSuccessors=aszSucc; \
        pHook->nOrder=nOrder; \
        pHook->szName=apr_hook_debug_current; \
        if(apr_hook_debug_enabled) \
         apr_hook_debug_show(#name,aszPre,aszSucc); \
    } \
    APR_IMPLEMENT_HOOK_GET_PROTO(ns,link,name) \
    { \
        return _hooks.link_##name; \
    }
```

对于 post_config 挂钩，我们该宏展开的结果如下：

```
AP_DECLARE(int) ap_hook_post_config(ap_HOOK_post_config_t *pf,
                                    const char * const *aszPre,
                                    const char * const *aszSucc,
                                    int nOrder)
{
```

```
    ap_LINK_post_config_t *pHook;
    if (!_hooks.link_post_config) {
        _hooks.link_post_config = apr_array_make(apr_hook_global_pool, 1,
                                                 sizeof(ap_LINK_post_config_t));
        apr_hook_sort_register("post_config", &_hooks.link_post_config);
    }
    pHook = apr_array_push(_hooks.link_post_config);
    pHook->pFunc = pf;
    pHook->aszPredecessors = aszPre;
    pHook->aszSuccessors = aszSucc;
    pHook->nOrder = nOrder;
    pHook->szName = apr_hook_debug_current;
    if (apr_hook_debug_enabled)
        apr_hook_debug_show("post_config", aszPre, aszSucc);
}

AP_DECLARE(apr_array_header_t *) ap_hook_get_post_config(void) {
    return _hooks.link_post_config;
}
```

从上面的展开结果中我们可以很清楚的看出 APR_IMPLEMENT_EXTERNAL_HOOK_BASE 宏实现了我们所需要的挂钩注册函数以及挂钩信息获取函数。

挂钩注册函数中的很多代码非常熟悉，一看原来就是前面我们 APR_HOOK_STRUCT 中用过的示例代码。挂钩注册函数首先检查挂钩数组是否为空，如果为空则说明是第一次注册该挂钩，所以创建新数组并注册该挂钩类型以供以后排序用；否则，直接加入一条记录。

5.5.6 使用挂钩

一旦各个模块在挂钩数组中注册了自己感兴趣的挂钩，那么剩下的事情无非就是调用这些挂钩，实际上也就是最终调用挂钩对应的函数。Apache 中的挂钩调用函数形式通常如 ap_run_*HOOKNAME 所 示，比如 ap_run_post_config 就是调用 post_config 挂钩。尽管所有的挂钩对外提供的调用形式都是一样的，但是内部实现却不尽相 同，分别体现于三个宏：AP_IMPLEMENT_HOOK_VOID、AP_IMPLEMENT_HOOK_RUN_FIRST 以及 AP_IMPLEMENT_HOOK_RUN_ALL。*

(1)、对于 AP_IMPLEMENT_HOOK_VOID，调用函数将逐个的调用挂钩数组中的所有的挂钩函数直到遍历调用结束或者发生错误为止。这种类型通常称之为 VOID，是由于其没有任何返回值，其声明如下：

```
    #define APR_IMPLEMENT_EXTERNAL_HOOK_VOID(ns,link,name,args_decl,args_use) \
    APR_IMPLEMENT_EXTERNAL_HOOK_BASE(ns,link,name) \
    link##_DECLARE(void) ns##_run_##name args_decl \
        { \
        ns##_LINK_##name##_t *pHook; \
        int n; \
    \
        if(!_hooks.link_##name) \
         return; \
    \
```

```
    pHook=(ns##_LINK_##name##_t *)_hooks.link_##name->elts; \
    for(n=0 ; n < _hooks.link_##name->nelts ; ++n) \
     pHook[n].pFunc args_use; \
    }
```

比如对于config.c中的child_init挂钩，其就是VOID类型，声明语句如下：

```
AP_IMPLEMENT_HOOK_VOID(child_init,
                       (apr_pool_t *pchild, server_rec *s),
                       (pchild, s))
```

撇去APR_IMPLEMENT_EXTERNAL_HOOK_BASE(ns,link,name)不管，这里我们仅仅关心剩下的的展开结果，如下：

```
AP_DECLARE(void) ap_run_child_init(apr_pool_t *pchild,server_rec* s)
{
    ap_LINK_child_init_t pHook;
    int n;
    if(!_hooks.link_child_init)
        return;
    pHook=(ap_LINK_child_init_t)_hooks.link_child_init->elts;
    for(n=0;n<_hooks.link_child_init->nelts;++n)
        pHook[n].pFunc(pchild, s);
}
```

从展开结果可以看出，即使在逐次调用过程中发生了错误，调用也不会停止，它就是“一头拉不回头的牛”。

(2)、AP_IMPLEMENT_HOOK_ALL简称ALL类型，其与AP_IMPLEMENT_HOOK_VOID几乎相同，唯一不同的就是ALL类型具有返回值。宏声明如下：

```
#define
APR_IMPLEMENT_EXTERNAL_HOOK_RUN_ALL(ns,link,ret,name,args_decl,args_use,ok,decline)
\
    APR_IMPLEMENT_EXTERNAL_HOOK_BASE(ns,link,name) \
    link##_DECLARE(ret) ns##_run_##name args_decl \
        { \
        ns##_LINK_##name##_t *pHook; \
        int n; \
        ret rv; \
    \
        if(!_hooks.link_##name) \
         return ok; \
    \
        pHook=(ns##_LINK_##name##_t *)_hooks.link_##name->elts; \
        for(n=0 ; n < _hooks.link_##name->nelts ; ++n) \
         { \
         rv=pHook[n].pFunc args_use; \
    \
         if(rv != ok && rv != decline) \
            return rv; \
```

```
        } \
        return ok; \
        }
```

open_logs 挂钩就是属于这种类型，其在 config.c 中的定义如下：

```
AP_IMPLEMENT_HOOK_RUN_ALL(int, open_logs,
                          (apr_pool_t *pconf, apr_pool_t *plog,
                           apr_pool_t *ptemp, server_rec *s),
                          (pconf, plog, ptemp, s), OK, DECLINED)
```

因此将它展开后的结果如下：

```
AP_DECLARE(int) ap_run_open_logs(apr_pool_t *pconf, apr_pool_t *plog,
                           apr_pool_t *ptemp, server_rec *s)
{
    ap_LINK_open_logs_t *pHook;
    int n;
    ret rv;
    if(!_hooks.link_open_logs)
          return ok;
    pHook=(ap_LINK_open_logs_t *)_hooks.link_open_logs->elts; \
    for(n=0 ; n < _hooks.link_open_logs->nelts ; ++n) \
     {
          rv=pHook[n].pFunc(pconf, plog, ptemp, s);
          if(rv != ok && rv != decline) \
              return rv;
     }
    return ok;
}
```

从展开的结果来看，ALL 类型在对挂钩数组进行遍历调用的时候，即使调用的请求被“DECLINE”，调用也将继续；只有调用请求发生错误才返回该错误值，同时退出遍历。

(3)、AP_IMPLEMENT_HOOK_FIRST 简称为 FIRST 类型，对于该类型 Apache 核心从头逐一遍历挂钩数组中所注册的挂钩函数，直到遇到一个能够完成所提交任务的函数或者发生错误为止。该宏的定义如下：

```
#define
APR_IMPLEMENT_EXTERNAL_HOOK_RUN_FIRST(ns,link,ret,name,args_decl,args_use,decline)
\
    APR_IMPLEMENT_EXTERNAL_HOOK_BASE(ns,link,name) \
    link##_DECLARE(ret) ns##_run_##name args_decl \
    { \
        ns##_LINK_##name##_t *pHook; \
        int n; \
        ret rv; \
    \
        if(!_hooks.link_##name) \
         return decline; \
    \
```

```
    pHook=(ns##_LINK_##name##_t *)_hooks.link_##name->elts; \
    for(n=0 ; n < _hooks.link_##name->nelts ; ++n) \
     { \
     rv=pHook[n].pFunc args_use; \
\
     if(rv != decline) \
         return rv; \
     } \
    return decline; \
}
```

quick_handler 就是属于该类型的挂钩，其在 config.c 中定义如下：

```
AP_IMPLEMENT_HOOK_RUN_FIRST(int, quick_handler, (request_rec *r, int lookup),
                            (r, lookup), DECLINED)
```

该宏展开后的结果如下所示：

```
AP_DECLARE(ret) ap_run_quick_handler(request_rec *r, int lookup)
{
    ap_LINK_quick_handler_t  *pHook;
    int n;
    ret rv;
    if(!_hooks.link_quick_handler)
          return decline;
    pHook=(ap_LINK_quick_handler_t *)_hooks.link_quick_handler->elts;
    for(n=0 ; n < _hooks.link_quick_handler->nelts ; ++n) \
     {
     rv=pHook[n].pFunc(r,look_up);
     if(rv != decline)
         return rv;
     }
    return decline;
}
```

从展开结果中可以看出，在遍历调用过程中一旦找到合适的函数，即 rv!=decline 的时候，函数即返回，不再继续遍历调用，即使后面仍然有合法的可调用函数。

任何挂钩都必须而且只能是三种类型中的一种。任何挂钩在被真正执行之前都必须调用这三个宏中的一个进行声明。

5.5.6 编写自己挂钩

使用挂钩所带来的最大的一个好处就是可以自行增加挂钩，而不需要全局统一。下面的部分，我们通过编写一个简单的挂钩同时在模块中使用该挂钩从而来加深理解上面所分析的内容。比如现在我们定义在了一个能够在 Apache 处理请求时调用的函数 some_hook，其函数原型为：

```
int some_hook(request_rec* r,int n);
```

那么我们分为四个步骤来声明我们的挂钩。

(1)、挂钩声明

我们使用 AP_DECLARE_HOOK 宏声明一个名称为 some_hook 的挂钩，声明如下：

```
AP_DECLARE_HOOK(int,some_hook,(request_rec* r,int n))
```

(2)、挂钩数组声明

由于 some_hook 是新声明的挂钩，为此，我们必须同时声明新的挂钩数组来保存各个模块对挂钩的注册使用。声明的语句如下：

```
APR_HOOK_STRUCT(
APR_HOOK_LINK(some_hook)
……
)
```

(3)、声明函数调用类型

挂钩声明完毕之后，真正被 ap_run_name 调用之前，我们还必须声明挂钩的调用类型，或者是 VOID 类型，或者是 FIRST 类型，或者是 ALL 类型。此处我们声明 some_hook 挂钩为 VOID 类型，声明语句如下：

```
AP_IMPLEMENT_HOOK_RUN_VOID(some_hook,(request_rec* r,int n),(r,n))
```

定义完该宏，实际上也对外声明了该挂钩的执行函数 ap_run_some_hook。

(4)、注册挂钩函数

至第三步为止，对挂钩的声明已经基本结束，也意味着下一步的工作实际上应该落实到挂钩使用者身上了。比如如果模块 som_module 中想使用挂钩 some_hook，则其必须在挂钩注册函数中注册该挂钩，挂钩注册函数为 ap_hook_some_hook，代码示例如下：

```
static void register_hooks()
{
    ……
    ap_hook_some_hook(some_hook_function,NULL,NULL,HOOK_MIDDLE);
}
AP_DECLARE_DATA module core_module = {
    ……
   register_hooks                /* register hooks */
};
```

(5)、编写挂钩函数

最后剩下的内容就是编写具体的挂钩调用函数。比如在 some_module 模块中，我们希望挂钩函数只是打印出“Hello World”语句，而且从(4)中看出挂钩函数名称为 some_hook_function，因此挂钩函数声明为如下：

```
static void some_hook_function(request_rec* r,int n)
{
    ap_rputs(“Hello World\n”);
    return;
}
```

需要注意的是，这边的挂钩函数必须符合 AP_IMPLEMENT_HOOK_RUN_XXX 中声明的格式。

Apache 中的挂钩剖析(3)

5.5.7 可选挂钩

与标准挂钩相比，可选挂钩基本上没有太大的差异，唯一的区别就在于可选挂钩不一定需要被实现——这看起来令人迷惑的。不过你很快就会明白了。考虑一下，如果某个挂钩 Hook_A 是声明在一个可选模块中，那么正常情况下该模块没有被加载。如此此时某个模块想使用挂钩 Hook_A，那么会发生什么情况呢。对于标准模块，Apache 可能根本就无法进行编译。而可选挂钩则可以解决这种问题。对于可选挂钩，即使它没有被导入并运行，其余的模块也可以使用它。

可选挂钩的声明方法与标准挂钩声明没有任何区别，都是通过 AP_DECLARE_HOOK 进行的，比如下面的语句声明一个可选挂钩：

```
AP_DECLARE_HOOK(int , optional_hook , (request_rec *r , int n))
```

与标准挂钩相比，可选挂钩没有内部私有的数据结构。在标准挂钩中，为了保存各个模块对声明的挂钩的使用情况，通过声明 AP_HOOK_STRUCT 结构来实现。这种做法实际上是由挂钩实现者自行进行维护；而对于可选挂钩，模块编写者可以不需要维护该 AP_HOOK_STRUCT 结构了，该结构则转交内核维护。

在实现上，可选挂钩的声明从标准挂钩的 AP_IMPLEMENT_HOOK_RUN_ALL 形式转变为 AP_IMPLEMENT_OPTIONAL_HOOK_RUN_ALL

5.5.7.1 可选挂钩数组

在 Apache2.0 中，任何模块对可选挂钩的调用信 息都由 Apache 核心进行维护。在 Apache 内部，核心定义了两个全局哈希表 s_phOptionalHooks 和 s_phOptionalFunctions 分别保存所有的可选挂钩以及对应的挂钩处理句柄。S_phOptionalHooks 哈希表中，可选挂钩名称 用来作为哈希表 Key 键，而挂钩对应的挂钩数组则作为哈希表的 Value 值。其结构如上图所示。在 Apr_hooks.c 中，Apache 提供了两个支持 函数：apr_optional_hook_get 和 apr_optional_hook_add。

apr_optional_hook_get 函数用来在哈希表 s_phOptionalHooks 中查找指定挂钩的挂钩数组，如果找到了则返回数组；否则返回 NULL。

apr_optional_hook_add 函数的原型声明如下：

```
APU_DECLARE(void) apr_optional_hook_add(const char *szName,void (*pfn)(void),
                    const char * const *aszPre,
                    const char * const *aszSucc,int nOrder)
```

该函数主要在可选挂钩 szName 数组中，增加一个登记项，登记的挂钩函数为 pfn。同时 aszPre、aszSucc 以及 nOrder 与标准挂钩的含义相同。

在登记之前，函数必须能够在哈希表中找到挂钩 szName 对应的挂钩数组，这个可以通过 apr_optional_hook_get 来完成。由于可选挂钩可以没有任何挂钩函数，因此上图中挂钩数组为 NULL 也是可能的，此时必须为该挂钩首先生成挂钩数组，将该挂钩数组在哈希表中与键 szName 关联起来，同时进行排序。

如果 szName 挂钩数组已经存在，则直接调用 apr_array_push 相关信息压入数组并赋值。

可选挂钩数组中每个元素的结构都是 apr_LINK__optional_t 类型，其是宏 APR_DECLARE_EXTERNAL_HOOK 展开的结果，apr_LINK__optional_t 结构实际如下所示：

```
typedef struct ap_LINK_optional_t
    {
        ap_HOOK_optional_t *pFunc;
        const char *szName;
        const char * const *aszPredecessors;
        const char * const *aszSuccessors;
        int nOrder;
    } ap_LINK_optional_t;
```

5.5.7.2 声明可选挂钩(APR_OPTIONAL_HOOK)

对于标准挂钩，注册使用挂钩通常使用 ap_hook_*name* 之类的函数，这些函数最终将使用信息登记到挂钩数组中去。而对于可选挂钩，由于不存在 AP_HOOK_STRUCT 宏，因此也就不存在挂钩数组了。在前面我们提到过，可选挂钩的保存是由 Apache 内核维护的，我们展开宏 APR_OPTIONAL_HOOK 就知道了。

APR_OPTIONAL_HOOK 宏定义在 ap_optional_hooks.h 中：

```
#define APR_OPTIONAL_HOOK(ns,name,pfn,aszPre,aszSucc,nOrder) do { \
 ns##_HOOK_##name##_t *apu__hook = pfn; \
 apr_optional_hook_add(#name,(void (*)(void))apu__hook,aszPre, aszSucc,
nOrder); \
} while (0)
```

5.5.7.3 APR_IMPLEMENT_OPTIONAL_HOOK_RUN_ALL

对于标准挂钩，其实现分为 VOID、FIRST 和 ALL 三种，而对于可选挂钩，实现则归结只有一种 ALL 类型，即 APR_IMPLEMENT_OPTIONAL_HOOK_RUN_ALL，该宏定义如下：

```
#define
APR_IMPLEMENT_OPTIONAL_HOOK_RUN_ALL(ns,link,ret,name,args_decl,args_use,ok,decline)
\
link##_DECLARE(ret) ns##_run_##name args_decl \
    { \
    ns##_LINK_##name##_t *pHook; \
    int n; \
    ret rv; \
    apr_array_header_t *pHookArray=apr_optional_hook_get(#name); \
    if(!pHookArray) \
     return ok; \
    pHook=(ns##_LINK_##name##_t *)pHookArray->elts; \
    for(n=0 ; n < pHookArray->nelts ; ++n) \
     { \
     rv=(pHook[n].pFunc)args_use; \
\
     if(rv != ok && rv != decline) \
         return rv; \
     } \
    return ok; \
    }
```

在 status 模块 mod_status.c 中我们声明的挂钩 status_hook 就是一个可选挂钩，该挂钩实现如下：

```
APR_IMPLEMENT_OPTIONAL_HOOK_RUN_ALL(ap, STATUS, int, status_hook,
                                    (request_rec *r, int flags),
                                    (r, flags),
                                    OK, DECLINED)
```

该宏展开后即实现了 ap_run_status_hook 函数，其实现过程，即展开结果如下所示：

```
AP_DECLARE(int) ap_run_status_hook(request_rec* r,int flags)
```

```
    {
        ap_LINK_status_hook_t *pHook;
        int n;
        int rv;
    apr_array_header_t *pHookArray=apr_optional_hook_get(status_name);
    if(!pHookArray)
return ok;
    pHook=(ap_LINK_status_hook_t *)pHookArray->elts;
    for(n=0 ; n < pHookArray->nelts ; ++n)
{
rv=(pHook[n].pFunc)(r,flags);
if(rv != ok && rv != decline)
    return rv;
}
    return ok;
     }
```

5.5.8 可选函数

与挂钩存在类似的问题，函数也可能存在可选挂钩的问 题。如果 Apache 在调用某一个函数的时候，该函数尚未被加载，会发生什么呢。你可能觉得 DSO 是个好的解决方法。如果指定的函数没有，则动态加载 DSO 模块进行查找。不过这种策略并不但是最完美的方案。首先，不是所有的平台都支持 DSO；另一方面，

可选函数的特点正如其名，这就决定了它相对于 Apache 的动态性。正常的函数都是编写之后才会加入 Apache 中进行编译，而且一旦编译就无法更改。而可选函数则是在需要的时候动态产生的。在产生 之前，没有人为之进行过专门的编写，因此链接代码中自然也就找不到该函数的实现。

可选函数的使用可以分为五大步骤：声明、实现、注册、获取以及函数调用。

5.5.8.1.可选函数的声明

声明一个可选函数通过宏 APR_DECLARE_OPTIONAL_FN 来实现，比如我们如果想声明一个 optional_fun 可选函数，其返回 int 类型，参数需要字符串类型，那么声明可以如下：

```
APR_DECLARE_OPTIONAL_FN(int,optional_fun,(const char* params))
```

APR_DECLARE_OPTIONAL_FN 宏定义如下：

```
 #define APR_DECLARE_OPTIONAL_FN(ret,name,args) \
typedef ret (APR_OPTIONAL_FN_TYPE(name)) args
```

如果将上面的宏展开，则可以看出，该宏只是声明了一个 apr_OFN_optional_fun_t 类型的函数指针：

```
typedef int (apr_OFN_optional_fun_t)(const char* params)
```

一旦声明完毕，我们则将其进行实现如下，在实现中必须注意名称以及函数参数类型的匹配：

```
int optional_fun(const char* params)
{
       ……
       return 0;
}
```

5.5.8.2. 可选函数的注册

可选函数由 Apache 核心统一维护。与可选挂钩类似，Apache 核心维护了一个 全局的哈希表 s_phOptionalFunctions，该哈希表的键为可选函数的名称，而值则是对应的函数指针。为了便于 Apache 使用，我们必须 在 s_phOptionalFunctions 中登记可选函数。函数的注册通过 APR_REGISTER_OPTIONAL_FN 进行，APR_REGISTER_OPTIONAL_FN 定义如下：

```
#define APR_REGISTER_OPTIONAL_FN(name) do { \
 APR_OPTIONAL_FN_TYPE(name) *apu__opt = name; \
 apr_dynamic_fn_register(#name,(apr_opt_fn_t *)apu__opt); \
} while(0)
```

该宏内部实际调用了函数 apr_dynamic_fn_register 进行实际的注册。事实上，Apache 中提供了相关函数来支持对 s_phOptionalFunctions 的操作，除了 apr_dynamic_fn_register 之外，还包括 apr_register_optional_fn、apr_dynamic_fn_retrieve、apr_retrieve_optional_fn。不过其中 apr_retrieve_optional_fn 和 apr_register_optional_fn 在 Apache2.0 中已经被废弃，因此不再多说。

apr_dynamic_fn_register 的原型如下：

```
APU_DECLARE_NONSTD(void) apr_dynamic_fn_register(const char *szName,
                                                 apr_opt_fn_t *pfn)
```

参数 szName 是可选函数的名称，pfn 则是实际可选函数的指针。如果 s_phOptionalFunctions 哈希表存在，函数只是简单的调用 apr_hash_set 将记录插入表中。

apr_dynamic_fn_retrieve 函数原型为

```
APU_DECLARE(apr_opt_fn_t *) apr_dynamic_fn_retrieve(const char *szName)
```

该函数根据给定的函数名称获取实际的函数指针。

事实上，这两个函数都不允许直接调用，它们作为 Apache 的内部函数而存在，对外提供的则是宏 APR_REGISTER_OPTIONAL_FN 和 APR_RETRIEVE_OPTIONAL_FN。

对于 optional_fun 可选函数，注册语句如下：

```
APR_REGISTER_OPTIONAL_FN(optional_fun);
```

当用户想使用可选函数的时候，首先必须获得其函数指针，用法如下：

```
APR_OPTIONAL_FN_TYPE(some_fn) *pfn;
pfn=APR_RETRIEVE_OPTIONAL_FN(some_fn);
```

5.5.8.3. 可选函数的使用

5.5.9 智能挂钩

5.5.10 挂钩工作机制

在前面的部分，我们对挂钩进行了详细的分析，但是还缺乏一个整体上的概念。从整体上来看挂钩的工作机制可以用下图来描述：

一个模块从挂钩的角度来看，其最重要的无非是两个方面：挂钩注册函数和挂钩处理函数。挂钩注册函数通常是模块结构中的 register_hooks 函数指针，该函数指针将调用实际的挂钩注册函数进行挂钩注册。挂钩注册的过程很简单，通过宏 ap_hook_*xxx* 实现。比如上图中声明了两个挂钩 abc 和 xyz。

与此同时，模块中也将声明与挂钩对应的挂钩函数，该挂钩被触发的时候，该函数将被调用。正如前面描述，可以使用宏 ap_run_*xxx* 触发指定的挂钩。不过挂钩的触发通常是由核心模块在对客户端请求进行处理的过程中进行。

我们来看一个具体的例子，这是核心模块中关于挂钩的部分。

从模块的结构中可以看出，模块结构中包含一个指针 register_hook，该指针通常指向模块中的实际的挂钩注册函数，比如，对于核心模块而言，其挂钩注册函数 register_hooks，那么结构中的该指针也为 register_hooks：

```
AP_DECLARE_DATA module core_module = {
    STANDARD20_MODULE_STUFF,
    create_core_dir_config,        /* create per-directory config structure */
    merge_core_dir_configs,        /* merge per-directory config structures */
    create_core_server_config,     /* create per-server config structure */
    merge_core_server_configs,     /* merge per-server config structures */
    core_cmds,                     /* command apr_table_t */
    register_hooks                 /* register hooks */
};
```

而具体的 register_hooks 函数则如下：

```
static void register_hooks(apr_pool_t *p)
{
    ap_hook_create_connection(core_create_conn, NULL, NULL,
                              APR_HOOK_REALLY_LAST);
    ap_hook_pre_connection(core_pre_connection, NULL, NULL,
                           APR_HOOK_REALLY_LAST);

    ap_hook_post_config(core_post_config,NULL,NULL,APR_HOOK_REALLY_FIRST);
    ap_hook_translate_name(ap_core_translate,NULL,NULL,APR_HOOK_REALLY_LAST);
    ap_hook_map_to_storage(core_map_to_storage,NULL,NULL,APR_HOOK_REALLY_LAST);
    ap_hook_open_logs(ap_open_logs,NULL,NULL,APR_HOOK_REALLY_FIRST);
    ap_hook_handler(default_handler,NULL,NULL,APR_HOOK_REALLY_LAST);
    ap_hook_type_checker(do_nothing,NULL,NULL,APR_HOOK_REALLY_LAST);
    ap_hook_fixups(core_override_type,NULL,NULL,APR_HOOK_REALLY_FIRST);
    ap_hook_access_checker(do_nothing,NULL,NULL,APR_HOOK_REALLY_LAST);
    ap_hook_create_request(core_create_req, NULL, NULL, APR_HOOK_MIDDLE);
    APR_OPTIONAL_HOOK(proxy, create_req, core_create_proxy_req, NULL, NULL,
                      APR_HOOK_MIDDLE);
    ap_hook_pre_mpm(ap_create_scoreboard, NULL, NULL, APR_HOOK_MIDDLE);
        ……
}
```

该函数的任务非常的简单，即声明该模块需要实现的挂钩，以及对应的处理函数，供主程序调用。

Apache 中的表格实现剖析(1)

3.2 表格(TABLE)

3.2.1 表格概述

尽管 apr_array_header_t 数组已经可以完成大部分的任务，但是对于 Apache 而言，apr_array_header_t 更倾向于内部 数据结构，它通常作为其余的线性数据结构的实现基础，比如表格、队列以及哈希表。表格是 Apache 中用的最频繁的数据结构，比如 HTTP 请求中的域以及 配置文件中的命令都是通过表格进行保存的。不过与通常的表格的含义不太相似，Apache 表格更加与 perl 中的哈希表相似，唯一的区别就是在 Apache 表格中同样的键值你可以存在两次，而且表格是大小写字母敏感的。

Apache 中表格的数据结构是 apr_table_t 结构，该结构在 apache 的 apr_table.h 中定义如下：

```
struct apr_table_t {
apr_array_header_t a;
#ifdef MAKE_TABLE_PROFILE
void *creator;
#endif
apr_uint32_t index_initialized;
int index_first[TABLE_HASH_SIZE];
int index_last[TABLE_HASH_SIZE];
};
```

从表格的结构可以看出，表格的内部还是数组结构，表格的各种操作实质上无非就是对数组元素操作的一种封装而已。**为了保持向下兼容，apr_array_header_t 必须位于表格结构的首部**。由于表格结构是 Apache 中新的数据结构，因此如果需要某些旧的只支持数组的版本如果需要使用表格，只要使用将 apr_table_t 强制转换为 apr_array_header_t 结构就可以了。

creator 用来跟踪表格的创建者。另外为了能够加快对表格的访问，结构中增加了三个辅助的成员变量，即 index_initialized，index_first，index_last。index_intialized 是一个 32 位的无符号整数，共对 应了 32 Í 8 个 bit，系统中用 256 位中的第 n 位来记录当前分配空间的第 n 个元素是否已经被初始化，如果初始化，该位为 1，否则 0。因此我们可以看到表格中的元素最多也只能为 256 个。

表格中的一个元素要被使用之前必须对其所在的区域进行初始化，初始化非常的简单，只是将 index_initialized 中对应的 bit 位置为 1 即可。可以通过宏 TABLE_SET_INDEX_INITIALIZED 实现初始化。

```
#define TABLE_SET_INDEX_INITIALIZED(t, i) ((t)->index_initialized |= (1 << (i)))
```

而判断一个对应的元素是否被初始化，则用宏 TABLE_INDEX_IS_INITIALIZED 实现：

```
#define TABLE_INDEX_IS_INITIALIZED(t, i) ((t)->index_initialized & (1 << (i)))
```

index_last 和 index_first 数组的用法我们在后面的部分会详细描述。

表格中存放的每一个元素用结构 apr_table_entry_t 描述：

```
struct apr_table_entry_t {
char *key;
char *val;
apr_uint32_t key_checksum;
```

```
};
```

结构中，key 是键值，目前用来标记表格中的每个元素，通常只有在对表格中的元素进行迭代的时候才能对该值进行检查。在以后的版本中，该值可能被设置为 NULL。val 则是当前元素的值。Key_checksum 则是对键值 key 的校验值，一般在表格内部使用。

由于表格的核心数据结构还是 apr_array_header_t 结构，因此对表格的大部分操作实际上还是对数组类型的操作。只不过此时数组的每个元素结构变成了 apr_table_entry_t 而已。下面我们来看看表格是如何进行操作的。

3.2.2 创建表格

为了创建一个表格，我们可以使用函数 ap_make_table 和 apr_make_table，前者适用于 Apache1.3，后者适用于 2.0 版本。函数定义生命如下：

```
APR_DECLARE(apr_table_t *) apr_table_make(apr_pool_t *p, int nelts)
{
    apr_table_t *t = apr_palloc(p, sizeof(apr_table_t));
    make_array_core(&t->a, p, nelts, sizeof(apr_table_entry_t), 0);
#ifdef MAKE_TABLE_PROFILE
    t->creator = __builtin_return_address(0);
#endif
    t->index_initialized = 0;
    return t;
}
```

函数首先从内存池 p 中分配处 apr_table_t 结构大小的内存块，然后调用 make_array_core(&t->a, p, nelts, sizeof(apr_table_entry_t), 0)为创建 apr_table_t 的内部数组 a，数组个数为 nelts 个，每个元素的大小为 sizeof(apr_table_entry_t)。如果 nelts 为零，函数将推迟内存分配直到表格第一次使用为止。正如在数组部分看到的，表格会自动的分配其需要的内存空间，而不需要手工干涉。另外数组创建 之后 index_initialized 被初始化为 0，此时没有任何数据被使用。

下面的代码演示了表格的创建操作：

```
apr_table_t *my_table;
my_table = apr_table_make(r->pool,10);
```

至此函数将创建了一个空空如也的表格，下面要做的就是往里面不断的放入 apr_table_entry_t 结构的数据了。

除了可以从头开始创建一个新的表格，Apache 中还允许从一个原有的表格创建一个相同的表格，我们称之为表格复制。表格复制实现如下：

```
APR_DECLARE(apr_table_t *) apr_table_copy(apr_pool_t *p, const apr_table_t *t)
{
    apr_table_t *new = apr_palloc(p, sizeof(apr_table_t));

    make_array_core(&new->a, p, t->a.nalloc, sizeof(apr_table_entry_t), 0);
    memcpy(new->a.elts, t->a.elts, t->a.nelts * sizeof(apr_table_entry_t));
    new->a.nelts = t->a.nelts;
    memcpy(new->index_first, t->index_first, sizeof(int) * TABLE_HASH_SIZE);
    memcpy(new->index_last, t->index_last, sizeof(int) * TABLE_HASH_SIZE);
    new->index_initialized = t->index_initialized;
```

```
    return new;
}
```

Apache 中的表格实现剖析(2)

3.2.3 表格元素查找

表格元素查找是表格的一个非常重要的功能。目前存在很多的线性表查找算法，最基本的无非三种：顺序查找，二分查找以及哈希查找。相对而言，顺序查找是最简 单也是最容易实现，但是其通常在插入数据时候较为快捷，而查找的时候则就相对较慢。二分查找是一个较快的查找方法，但是其前提必须数据进行排序，因此排序 使得插入较慢，因此也不是所有场合均适合。而哈希查找则既实现简单，又插入和查找速度较快。因此在 Apache 中队表格的插入和查找则采取了哈希算法。

Apache 中在表格中查找一个键值为 key 的元素的函数为 apr_table_get，其定义如下：

```
APR_DECLARE(const char *) apr_table_get(const apr_table_t *t, const char *key)
{
    apr_table_entry_t *next_elt;
    apr_table_entry_t *end_elt;
    apr_uint32_t checksum;
    int hash;

    if (key == NULL) {
     return NULL;
    }
    hash = TABLE_HASH(key);
    if (!TABLE_INDEX_IS_INITIALIZED(t, hash)) {
        return NULL;
    }
    COMPUTE_KEY_CHECKSUM(key, checksum);
    next_elt = ((apr_table_entry_t *) t->a.elts) + t->index_first[hash];;
    end_elt = ((apr_table_entry_t *) t->a.elts) + t->index_last[hash];

    for (; next_elt <= end_elt; next_elt++) {
     if ((checksum == next_elt->key_checksum) &&
            !strcasecmp(next_elt->key, key)) {
         return next_elt->val;
     }
    }
    return NULL;
}
```

正如前面的 apr_table_entry_t 中看到的，对于表格中的每一个元素我们都维护一个 key 成员，为字符指针变量，该 key 用来标识该元素，在 Apache 中由于允许重复 key 的存在，因此在表格中可能存在多个 key 相同的元素，不过 apr_table_get 函数只返回查找到的第一个符合要求 的元素。

对于任何一个键值，我们可以通过 TABLE_HASH(key)找到对应元素在表格中存放的索引。TABLE_HASH 为一个宏，定义为(TABLE_INDEX_MASK & *(unsigned char *)(key))。一旦得到哈

希索引 hash，函数将判断该索引所定应的内存块是否已经被初始化。如果该索引块还没有初始化，则理所当然，里面什么都没 有，也就没法取了，此时直接返回 NULL。

当然如果内存中确实有“货”可用，则函数将进一步调用宏 COMPUTE_KEY_CHECKSUM(key, checksum)来计算键值 key 所对应的校验值，校验结果即为 checksum，该 checksum 主要作为宏 TABLE_HASH 的参数。

现在我们再来回顾前面提到的 apr_table_t 中的剩余的两个辅助成员变量：

```
int index_first[TABLE_HASH_SIZE];
int index_last[TABLE_HASH_SIZE];
```

前面我们提到这二个辅助成员主要用在加速对表格中成员的查找，那么我们现在看看这两个成员变量是如何实现查找加速的。

我们首先来看宏 TABLE_HASH(key)的实现：

```
#define TABLE_HASH(key) (TABLE_INDEX_MASK & *(unsigned char *)(key))
```

从上面的哈希值得计算中可以看出，不管对于任何一个键值 key，只有该键值的第一个字母参与了运算，因此比如对于“hello”和“house”而言，其 计算出来的结果都是 8。我们将整个表格分为多个“类别”，这些类别通过 TABLE_HASH(key)进行区分，实际上也就是按照 key 的第一个字母进行 分类。因此“hello”和“house”属于同一个类，而和“cat”则属于不同的类别。Apache 中对键值的查找是基于类别进行的。如果一个类别在 表格中具有多个元素，那么毫无疑问，肯定就发生了哈希冲突。

为了解决索引冲突的问题，apr_table_t 中引入 index_first 和 index_last 数组，分别记录给定的类别在表格中的起始索引和终止 索引。对于某个类别，比如“h”类别(所有的以 h 开始的键值都属于这个类)，它在 index_first 和 index_last 中的索引可以通过 TABLE_HASH(key)计算得到，结果为 8。因此整个表格中第一个 h 类元素在表格中的索引保存在 index_first[8](即 index_first[TABLE_HASH(key)]中，而最后一个 h 类的成员的索引则保存在 index_last[8]中。同理如果对于“c”类 别，它的 TABLE_HASH(key)为 3，因此整个表格的第一个 c 类元素的索引则保存在 index_first[3]中，最后一个在保存在 index_last[3]中。

比如，目前在表格中仅存在三个键值以字母 h 开头，分别为 “house”、“hello”、“horse”，它们在表格中的索引分别为 7、9、12，同时还存在四个以 c 开始的键值，分别是“cat”、“copy” 、“catch”、“cite”，它们在表格中的索引分别为 5、6、8、10。对于 h 类而言，它们对应的 index_first 中的索引为 8，值为 7，意味着所有的以 h 开始的键值第一次出现是在表格的索引为 7 的位置，同样从上表可以看出，所有以 c 开始的键值第一次出现是在表格的索引为 5 的位 置。同样对于 h 类而言，它们对应的 index_last 中的索引也是 8，值为 12，意味着表格中最后一个以 h 开始的键值在表格中的索引为 12，同时，表格 中的最后一个以 c 起始的键值的索引是 10。

如果现在需要查找”hello”，经过 COMPUTE_KEY_CHECKSUM(key, checksum)计算后，它的 checksum 为 8。因此函数此时将检查 index_first[8]和 index_last[8]中的数据，如图，分 别为 7 和 11。这意味着所有的以’h’开始的键值都在表格的第 7 和第 11 之间，因此此时只需要搜索索引从 7 到 11 的元素就可以了。

一旦明确 index_first 和 index_last 数组的用途，我们就可以使用这两个 数组加快搜索速度。比如如果需要搜索“hello”，我们不再需要从第一个元素查找到最后一个元素，相反，我们仅仅需要搜索表格中索引位于 index_first[TABLE_HASH(“hello”)]和 index_last[TABLE_HASH(“hello”)]之间的元素，即表 格中第 7 个和第 12 个之间的元素。通

过这种策略，可以大大缩小查询的范围。对于缩小范围之内的查找，我们则使用普通的查找方法逐一比较。

上述算法的实现代码如下：

```
next_elt = ((apr_table_entry_t *) t->a.elts) + t->index_first[hash];;
end_elt = ((apr_table_entry_t *) t->a.elts) + t->index_last[hash];
for (; next_elt <= end_elt; next_elt++)
{
      if ((checksum == next_elt->key_checksum) &&!strcasecmp(next_elt->key, key))
      {
          return next_elt->val;
      }
}
```

在进行比较的时候，上面的代码采取了一定的变通。正常情况下，将需要查询的键值与查询范围内的每一个字符串键值进行比较的时候可以调用 strcmp 实现， 对于 Apache 而言则就是 strcasecmp。为了能够加快查找的速度，Apache 中对于每一个字符串，取其前面的 4 个字符计算该字符串的校验码：

```
#define COMPUTE_KEY_CHECKSUM(key, checksum)    \
{                                              \
    const char *k = (key);                     \
    apr_uint32_t c = (apr_uint32_t)*k;         \
    (checksum) = c;                            \
    (checksum) <<= 8;                          \
    if (c) {                                   \
        c = (apr_uint32_t)*++k;                \
        checksum |= c;                         \
    }                                          \
    (checksum) <<= 8;                          \
    if (c) {                                   \
        c = (apr_uint32_t)*++k;                \
        checksum |= c;                         \
    }                                          \
    (checksum) <<= 8;                          \
    if (c) {                                   \
        c = (apr_uint32_t)*++k;                \
        checksum |= c;                         \
    }                                          \
    checksum &= CASE_MASK;                     \
}
```

宏 COMPUTE_KEY_CHECKSUM 计算给定 key 的校验码 checksum，因此对于任何的字符串比较都会分为两个步骤：

```
checksum == next_elt->key_checksum) &&!strcasecmp(next_elt->key, key
```

首先比较它们的校验码，即前四个字符，如果相等，才会进一步调用 strcasecmp 进行比较；如果校验整数值不相等，那么就没有必要调用 strcasecmp 进一步比较。由于整数之间的比较速度比较快，因此通过这种两阶段比较能够有效的提高比较速度。

Apache 中的哈希表剖析(1)

3.4 哈希表

3.4.1 哈希表概述

作为线性数据结构，与前面所说的表格和队列等相比，哈希表无疑是查找速度比较快的一种。APR 中较好的支持哈希表。APR 中哈希表在文件 apr_hash.h 和 apr_hash.c 中实现。其数据结构定义如下：

```
struct apr_hash_t {
    apr_pool_t          *pool;
    apr_hash_entry_t   **array;
    apr_hash_index_t     iterator; /* For apr_hash_first(NULL, ...) */
    unsigned int         count, max;
    apr_hashfunc_t       hash_func;
    apr_hash_entry_t    *free; /* List of recycled entries */
};
```

与其余的数据结构类似，第一个成员通常是分配该结构的内存池。哈希表中的每一个元素都用 apr_hash_entry_t 结构描述，array 指向一个数 组，该数组中每一个元素都是一个指向 apr_hash_entry_t 类型的链表。为了方便对哈希表的迭代循环遍历，每个结构中都维持一个 apr_hash_index_t 用以辅助迭代，该成员的详细细节我们在后面的部分会深入讨论。

count 用以记录当前整个哈希表中结点的总数目。max 则是当前哈希表中允许的哈希值的最大值，反映到结构中就是 array 数组中的元素的个数。

hash_func 则是哈希函数，通过该函数可以确定给定元素在哈希表中的索引，可以使用默认的哈希函数，也可以使用自定义的哈希函数。

现在我们来看一下哈希表元素数据结构 apr_hash_entry_t，该结构定义如下：

```
struct apr_hash_entry_t {
apr_hash_entry_t     *next;
unsigned int          hash;
const void           *key;
apr_ssize_t          klen;
const void           *val;
};
```

hash 是当前元素在哈希表中所对应的哈希索引值，key 则是哈希项的键，value 则是哈希项的值。哈希表中以键值 key 作为唯一的标识索引。如果键值存在冲突，那么这些冲突键将用链表关联起来。整个哈希表的结构可以用下面的图描述：

3.4.2 哈希表创建

3.4.2.1 零创建

APR 中创建一个哈希表可以通过两种途径：apr_hash_make 和 apr_hash_make_custom 实现：

```
APR_DECLARE(apr_hash_t *) apr_hash_make(apr_pool_t *pool);
```

APR_DECLARE(apr_hash_t *) apr_hash_make_custom(apr_pool_t *pool, apr_hashfunc_t hash_func);

两者的区别就是哈希算法的不同。对于apr_hash_make而言，它的主要的工作就是创建apr_hash_t结构，并对其中的成员进行初始化，其中 哈希元素的个数被初始化为16个，同时使用默认的哈希算法apr_hashfunc_default，而apr_hash_make_custom则使用 自定义的哈希函数hash_func。

```
unsigned int apr_hashfunc_default(const char *char_key, apr_ssize_t *klen)
{
    unsigned int hash = 0;
    const unsigned char *key = (const unsigned char *)char_key;
    const unsigned char *p;
    apr_ssize_t i;
    if (*klen == APR_HASH_KEY_STRING) {
        for (p = key; *p; p++) {
            hash = hash * 33 + *p;
        }
        *klen = p - key;
    }
    else {
        for (p = key, i = *klen; i; i--, p++) {
            hash = hash * 33 + *p;
        }
    }
    return hash;
}
```

对于给定的键值key，apr_hashfunc_default返回它在哈希表中的索引。默认哈希算法采用了目前最为流行的times 33哈希算法，目前该算法被广泛使用在多个软件项目包括perl和巴克利(Berkeley DB)数据库中。对于字符串而言这是目前所知道的最好的哈希算法，原因在于该算法的速度非常快，而且分类非常好。

不过你不愿意使用该索引算法而希望使用自己的，那么你可以使用apr_hash_make_custom函数，它接受一个自定义的哈希算法函数，并将其赋值给apr_hash_t结构内的func成员，从而取代默认算法函数。

apr_hashfunc_t函数指针定义如下：

typedef unsigned int (*apr_hashfunc_t)(const char *key, apr_ssize_t *klen);

它需要两个参数，一个是需要进行哈希计算的键，另一个则是该键的长度。函数返回计算后的索引。

3.4.2.2 拷贝创建

与大部分数据结构一样，对于哈希表，APR也提供了拷贝创建方法，允许从一个已有的哈希表创建一个新的哈希表，拷贝函数声明如下：

APR_DECLARE(apr_hash_t *) apr_hash_copy(apr_pool_t *pool,const apr_hash_t *orig)

orig 是源哈希表，在拷贝中所用所有的内存都来自内存池 pool，拷贝后的哈希表由函数返回。

```
APR_DECLARE(apr_hash_t *) apr_hash_copy(apr_pool_t *pool,const apr_hash_t
*orig)
{
    apr_hash_t *ht;
    apr_hash_entry_t *new_vals;
    unsigned int i, j;
    ht = apr_palloc(pool, sizeof(apr_hash_t) +
                sizeof(*ht->array) * (orig->max + 1) +
                sizeof(apr_hash_entry_t) * orig->count);
    ht->pool = pool;
    ht->free = NULL;
    ht->count = orig->count;
    ht->max = orig->max;
    ht->hash_func = orig->hash_func;
    ht->array = (apr_hash_entry_t **)((char *)ht + sizeof(apr_hash_t));
    new_vals = (apr_hash_entry_t *)((char *)(ht) + sizeof(apr_hash_t) +
                                    sizeof(*ht->array) * (orig->max + 1));
```

尽管称之为哈希表拷贝，但是 apr_hash_copy 实现的仅仅是一种**影像拷贝**。之所以称之为影像拷贝，是因为尽管拷贝后的哈希表能够实现与源哈希表相同的功能，但是内部数据组织已经发生了变化，**最大的变化就是从源哈希表的不连续的链表结构转换为连续的块状结构。**

既然是块状数据结构，我们首先就必须考虑块状结构的大小，然后才能分配。新的块状结构的大小应该与原有的链表结构大小相等。总的大小包括三方面：

1)、apr_hash_t 结构的大小 sizeof(apr_hash_t)

2)、apr_hash_t 结构内 array 数组的大小，数组的总元素个数为 max+1，每一个元素都是一个 apr_hash_entry_t 类型的指针，因此整个数组的大小为(orig->max+1)*sizeof(*ht->array)，或 者也可以写成(orig->max+1)*sizeof(apr_hash_entry_t*)。

3)、整个哈希表中 apr_hash_entry_t 类型结点的总数，其值由 orig->count 决定，每个结点的大小为 sizeof(apr_hash_entry_t)，故总大小为 sizeof(apr_hash_entry_t) * orig->count。

一旦确定了总的需要分配的内存大小，APR 将从内存池中一次性分配足够的连续内存。这些内存将被分为三部分：apr_hash_t 部分、max+1 个 apr_hash_entry_t 指针部分以及 count 个哈希元素的大小。因此内存一旦分配完毕，除了使用原有哈希表结构成员初始化新的哈希表成员包括 pool、count、max 以及 hash_func 等之外，最重要的就是设置指针指向后面的两个内存部分初始化后的布局如下所示：

```
1.     j =0;
2.     for (i = 0; i <= ht->max; i++) {
3.         apr_hash_entry_t **new_entry = &(ht->array[i]);     u
4.         apr_hash_entry_t *orig_entry = orig->array[i];
5.         while (orig_entry) {
6.             *new_entry = &new_vals[j++];
```

```
7.              (*new_entry)->hash = orig_entry->hash;
8.              (*new_entry)->key = orig_entry->key;
9.              (*new_entry)->klen = orig_entry->klen;
10.            (*new_entry)->val = orig_entry->val;
11.             new_entry = &((*new_entry)->next); v
12.             orig_entry = orig_entry->next;w
13.         }
14.         *new_entry = NULL;
15.     }
16.     return ht;
```

在源哈希表中，ht->array 数组中的每一个元素都是一个 apr_hash_entry_t 类型的指针，指向的是一个链表，而到了目标哈希表 中，该指针指向的则是一个数组。因此，拷贝的一个重要任务就是调整 array 数组中的各个指针将其指向 new_vals 内存的适当的部分。调整后的布局如 下图所示。

调整过程包括三个大步骤，图示中用 jkl 进行标识：

j array 数组中的每一个元素都是一个指针，必须调整数组中的每一个指针指向 new_vals 部分的合适的位置。原则是：如果源哈希表中对应元素为 NULL，则新指针也为 NULL；如果对应的结点链表结点数目为 n，则下一个索引指针与前一个偏移 n*sizeof(apr_hash_entry_t) 个。具体如灰色代码部分所示。

k 对每一个结点进行内容拷贝，如 7.8.9.10 行。

l 调整各个结点内部的 next 指针，指向它的直接后继结点，或者是紧靠它的下一个 apr_hash_entry_t 结点，或者是 NULL。

```
new_entry = &((*new_entry)->next);
```

在上图中我们用虚线将链表结构和块状结构中对应得结点连接起来，以方便对比。

Apache 中的哈希表剖析(2)

3.4.3 数据插入和获取

对于哈希表而言，一个重要的任务就是插入 key/value 数据以及根据键值获取相应的值。APR 中定义了函数 apr_hash_set 和 apr_hash_get 分别实现上面的功能。

首先我们来看 apr_hash_get 函数，该函数需要三个参数，分别用以描述操作的哈希表，键值以及键的长度。

```
APR_DECLARE(void *) apr_hash_get(apr_hash_t *ht,const void *key,apr_ssize_t klen)
{
    apr_hash_entry_t *he;
    he = *find_entry(ht, key, klen, NULL);
    if (he)
        return (void *)he->val;
    else
        return NULL;
}
```

从上面的函数可以看到，通过键值查找对应的哈希元素的过程真正的工作是由find_entry完成的，尽管find_entry是一个内部函数，不过它是很多函数的基础，其定义如下：

```
static apr_hash_entry_t **find_entry(apr_hash_t *ht,const void *key,apr_ssize_t klen,
                                     const void *val)
{
    apr_hash_entry_t **hep, *he;
    unsigned int hash;

    hash = ht->hash_func(key, &klen);
   for (hep = &ht->array[hash & ht->max], he = *hep;
         he; hep = &he->next, he = *hep) {
        if (he->hash == hash
            && he->klen == klen
            && memcmp(he->key, key, klen) == 0)
            break;
    }
    if (he || !val)
        return hep;

    /* add a new entry for non-NULL values */
    if ((he = ht->free) != NULL)
        ht->free = he->next;
    else
        he = apr_palloc(ht->pool, sizeof(*he));
    he->next = NULL;
    he->hash = hash;
    he->key = key;
    he->klen = klen;
    he->val = val;
    *hep = he;
    ht->count++;
    return hep;
}
```

整个函数的处理过程可以分为三大部分：

1)、根据键值key计算出它在整个表array中的索引值hash，这个过程非常的简单，通常可以直接调用apr_hash_t内部的hash_func计算获取

2)、正如前面所说，由于哈希值冲突的存在，因此存在多个键值对应同一个索引的情况，所有索引计算结果相同的键都保存在以array[hash]为首地址 的链表中。因此对于在确定hash索引之后，下一步必须处理的就是遍历该索引对应的链表，在其中查找是否存在目标结点，当一个结点必须满足下面的条件的时 候才能称之为完全匹配：

■ 当前结点的hash值与计算出的哈希值相等。一般情况下，这个条件都是满足的，因为不相等的话不可能挂接到hash索引对应的链表。

■ 键值长度完全相同

■ 键的值完全满足条件

3)、尽管函数的名称为 find_entry，这个名字容易让人误会为函数仅仅完成查找功能，事实上并不如此。函数 apr_hash_get 和 apr_hash_set 内部都调用了该函数。因此，find_entry 并不仅仅是查找，同时还具备增加结点的功能。至于到底是查找还是增加由最后一个 参数 val 决定，如果 val 为 NULL，则函数理所当然为查找，否则为添加。

对于查找功能，函数直接返回找到的元素结点。如果是增加，则将当前结点增加到 hash 索引链表的最末尾。

3.4.4 哈希表迭代遍历

APR 中哈希表的遍历遵循的一个原则就是深度遍历原则，按照这种思路，遍历过程首先将 hash=0 的链表中的所有结点遍历完毕，而后继续遍历 hash=1 的链表，直至 hash=max 的链表，这种思路可以用下面的图示描述，红虚线就是哈希表元素的遍历顺序：

在讲解 APR 中遍历算法之前，我们首先用最普通的方法来实现对哈希表的遍历，下面是遍历的框架代码：

```
int i=0;
for(;i<max;i++)
{
    apr_hash_entry_t *ht;
    if(array[i]==NULL)
         continue;
    ht=array[i];
    while(ht!=NULL)
   {
        //处理该结点，或者打印或者其余操作
        DoSomethingAboutTheNode();
        ht=ht->next;
   }
}
```

上面的程序基本能够实现对整个哈希表的遍历，而且也很容易理解。但是如果从使用者的角度去考虑一下就很容易发现尽管上面的遍历没有问题，但是对用户而言则 并不是很现实。使用上面的遍历算法的一个前提就是用户必须能够知道哈希表的内部实现结构，事实上这个对大部分的用户并不现实，而且哈希表的内部结构通常总 是对用户屏蔽的。如果这样，问题又出来了，既然用户不需要了解哈希表的内部实现细节，那么他又从何知道使用上面的遍历代码呢？？

在 C++ STL 中大部分的容器，比如 vector，stack，list 以及 queue 等等，它们内部都支持一种称之为迭代子的类型，该类型允许将容器类型和算法 分开来，彼此独立设计，最后再通过胶合剂将它们关联起来，不过这种关联是一种松耦合关系，下面是使用向量(vector)迭代子对向量进行迭代的过程：

```
vector<int> coll;
……
vector<int>::iterator pos;
```

```
for (pos=coll.begin(); pos<coll.end(); ++pos) {
printf(“%d ”,*pos);// cout << *pos << ’ ’;
}
```

从上面的代码中我们可以看到，对于用户而言根本不需要知道 vector 的内部细节，它需要知道的仅仅是从 begin()访问到 end()，至于 begin()和 end()返回的是不是向量中真正的第一个和最后一个，那不是用户需要考虑的事情。

不过迭代子是范型技术的产物，是和 C++紧密联系的，而 Apache 则是纯 C 编写的，因此当然不可能直接使用 STL，不过它显然是受了 STL 的影响——它 的目的就是使用 C 语言仿造一个哈希表上的迭代子，使得算法与细节分开。下面是 APR 中使用仿迭代子对哈希表 ht 进行遍历的代码：

```
apr_hash_index_t *hi;
for (hi = apr_hash_first(NULL, ht); hi; hi = apr_hash_next(hi)) {
    const char *k;
    const char *v;
    apr_hash_this(hi, (const void**)&k, NULL, (void**)&v);
    printf("ht iteration: key=%s, val=%s\n", k, v);
}
```

在上面的代码中你根本就看不出哈希表的内部细节。这实际上也达到了我们前面提出的要求为了实现仿造的效果，哈希表中引入了一个辅助数据结构 apr_hash_index_t，用于记录当前访问的结点的相关索引信息：

```
struct apr_hash_index_t {
    apr_hash_t          *ht;
    apr_hash_entry_t    *this, *next;
    unsigned int         index;
};
```

ht 是当前访问的哈希表，index 是当前访问的结点所在的哈希索引，值从 0 到 max 之 间。this 和 next 是这个结构的最重要的成员。this 用以指向当前正在访问的结点，而 next 则指向按照深度遍历时 this 之后下一个将要被访问的 结点。在整个遍历过程中，最重要的就是不断地调整 this 和 next 的指针，下面是可能出现的情况：

在分析上面的图示的时候，我们给出两个定义：**直接后继结点**和**间接后继结点**。

对于某个结点，它的直接后继结点就是可以通过 next 指针直接获取的结点。比如上图的(1)-(4)，this 结点的直接后继结点都是 NULL，而(5)中，this 的直接后继存在，且为 next。

对于某个结点，它的间接后继结点就是按照前述的哈希表深度遍历访问顺序中的下一个结点。对于图(5)，this 的间接后继结点就是它的直接后继结点。下表给出了上图中的各个直接后继和间接后继结点的情况：

图	直接后继结点	间接后继结点

(1)	NULL	NULL
(2)	NULL	Not NULL
(3)	NULL	NULL
(4)	NULL	Not NULL
(5)	Not NULL	Not NULL

从上表可以看出(1)-(4)图可以归结为一个大类，那就是this的直接后继都是为 NULL，这导致this结点和next结点位于不同的索引链表中，而(5)则是另外一个大类，它的直接后继不为NULL，这导致this和next位于 同一链表中。对于给定的this结点，如何获取它的下一个结点？？Apache中使用apr_hash_next函数来获取this结点的下一个结点：

```
APR_DECLARE(apr_hash_index_t *) apr_hash_next(apr_hash_index_t *hi)
{
    hi->this = hi->next;
    while (!hi->this) {
        if (hi->index > hi->ht->max)
            return NULL;
        hi->this = hi->ht->array[hi->index++];
    }
    hi->next = hi->this->next;
    return hi;
}
```

this结点由传入参数hi指定，函数内部计算this的next结点，并继续通过hi返回出去。函数中正是利用了上面所说的分类规律：

■ while循环用于处理上图的(1)(2)(3)(4)四种情况，如果直接后继为NULL，则直接跳至下一个索引链表工作，直到索引为max为止。

■ hi->next=hi->this->next用以处理直接后继不为NULL的情况，那么此时只需要调整next就可以了。

除了apr_hash_next之外，Apache中还提供了另外两个辅助函数 apr_hash_first和ap_hash_this。apr_hash_first主要为遍历整个哈希表作必要的准备，并返回整个哈希表的第一个元 素的索引结构apr_hash_index_t。该函数需要两个参数：内部索引分配所需要的内存池以及需要遍历的哈希表：

```
APR_DECLARE(apr_hash_index_t *) apr_hash_first(apr_pool_t *p, apr_hash_t *ht)
{
    apr_hash_index_t *hi;
    if (p)
        hi = apr_palloc(p, sizeof(*hi));
    else
        hi = &ht->iterator;

    hi->ht = ht;
    hi->index = 0;
    hi->this = NULL;
    hi->next = NULL;
    return apr_hash_next(hi);
```

```
}
```

函数的前半部分主要进行 apr_hash_index_t 结构的初始化工作，如果内部具有迭代子，那么直接使用该迭代子，否则创建新的迭代子，并使用该迭代子进行第一次迭代，apr_hash_next 将返回哈希表中的第一个元素，同时下一个间接后继结点同时也返回。

apr_hash_this 函数主要返回给定结点的各种信息，通常情况下，总是将 this 指针传递个该函数：

```
APR_DECLARE(void) apr_hash_this(apr_hash_index_t *hi,const void **key,
                                apr_ssize_t *klen,void **val)
{
    if (key) *key = hi->this->key;
    if (klen) *klen = hi->this->klen;
    if (val) *val = (void *)hi->this->val;
}
```

Apache APR 可移植运行库简介(1)

APR(Apache portable Run-time libraries，Apache 可移植运行库)的目的如其名称一样，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。在早期 的 Apache 版本中，应用程序本身必须能够处理各种具体操作系统平台的细节，并针对不同的平台调用不同的处理函数。随着 Apache 的进一步开 发，Apache 组织决定将这些通用的函数独立出来并发展成为一个新的项目。这样，APR 的开发就从 Apache 中独立出来，Apache 仅仅是使用 APR 而已。目前 APR 主要还是由 Apache 使用，不过由于 APR 的较好的移植性，因此一些需要进行移植的 C 程序也开始使用 APR，开源项目比如 Flood loader tester(http://httpd.apache.org/test/flood/，该项目用于服务器压力测试，不仅仅适用于 Apache)、FreeSwitch(www.freeswitch.org)，JXTA-C(http://jxta-c.jxta.org,C 版本的 JXTA 点对点 平台实现)；商业的项目则包括 Blogline(http://www.bloglines.com/，covalent(http: //www.covalent.net)等等。

APR 使得平台细节的处理进行下 移。对于应用程序而言，它们根本就不需要考虑具体的平台，不管是 Unix、Linux 还是 Window，应用程序执行的接口基本都是统一一致的。因此对于 APR 而言可移植性和统一的上层接口是其考虑的一个重点。而 APR 最早的目的并不是如此，它最早只是希望将 Apache 中用到的所有代码合并为一个通用 的代码库，然而这不是一个正确的策略，因此后来 APR 改变了其目标。有的时候使用公共代码并不是一件好事，比如如何将一个请求映射到线程或者进程是平台相 关的，因此仅仅一个公共的代码库并不能完成这种区分。APR 的目标则是希望安全合并所有的能够合并的代码而不需要牺牲性能。

APR 的最早的一个目标就是为***所有的平台***(不 是部分)提供一个公共的统一操作函数接口，这是一个非常了不起的目的，当然也是不现实的一个目标。我们不可能支持所有平台的所有特征，因此 APR 目前只能 为大多数平台提供所有的 APR 特性支持，包括 Win32、OS/2、BeOS、Darwin、Linux 等等。为了能够实现这个目标，APR 开发者必须为 那些不能运行于所有平台的特性创建了一系列的特征宏(FEATURE MACROS)以在各个平台之间区分这些特征。这些特征宏定义非常简单，通常如下：

APR_HAS_FEATURE

如果某个平台具有这个特性，则该宏 必须设置为 true，比如 Linux 和 window 都具有内存映射文件，同时 APR 提供了内存映射文件的操作接口，因此在这两个平台 上，APR_HAS_MMAP 宏必须设置，同时 ap_mmap_*函数应该将磁盘文件映射为内存并返回适当的状态码。如果你的操作系统并不支持内存映射， 那么 APR_HAS_MMAP 必须设置为 0，而且所有的 ap_mmap_*函数也可以不需要定义。第二步就是对于那些在程序中使用了不支持的函数必须提出 警告。

目前APR中支持的基本类型包括下面几种：

表3-1 APR中支持的基本类型

类型名称	文件夹名称	描述
atomic	/srclib/apr/atomic	原子操作
dso	/srclib/apr/dso	动态加载共享库
file io	/srclib/apr/file_io	文件IO处理
mmap	/srclib/apr/mmap	内存映射文件
locks	/srclib/apr/locks	进程和线程互斥锁
memory	/srclib/apr/memory	内存池操作
network_io	/srclib/apr/network_io	网络IO处理
poll	/srclib/apr/poll	轮询IO
table	/srclib/apr/tables	Apache数组(堆栈)和表格以及哈希表
process	/srclib/apr/threadproc	进程和线程操作
user	/srclib/apr/user	用户和用户组操作
time	/srclib/apr/time	时间操作
string	/srclib/apr/strings	字符串操作
password	/srclib/apr/passwd	终端密码处理
misc	/srclib/apr/misc	大杂烩，不属于其余类的任何apr类型都可以放在里面
shmem	/srclib/apr/shmem	共享内存
random	/srclib/apr/random	随机数生成库

每一个APR的实现我们都在后面会详细描述。

1.2 APR版本规则

由于Apache组织的目标是将APR独立出来形成单独的第三方库，因此对其而言稳定的API接口就成为一个非常重要的必须考虑的方面。不过由于APR需要不断的往前方展，因此API接口的变化又是必然的趋势，因此如何平衡稳定性和变化性是APR开发 者面临的一个极需解决的问题。为此APR采用了严格的版本规则来实现这一点。用户只需要简单的判断APR版本号，就可以很容易确定当前版本的兼容性：向前 兼容、向后兼容还是前后同时兼容。

1.2.1 版本概述

APR中使用三个整数来记录APR版本号：**MAJOR.MINOR.PATCH**。MAJOR表示当前APR的主版本号，它的变化通常意味着APR的巨大的变化，比如体系结构的重新设计，API的重新设计等等，而且这种变化通常会导致APR版本的向前不兼容。MINOR称之为APR的次版本号，它通常只反映了一些较大的更改，比如APR的API的增加等等，但是这些更改并不影响与旧版本源代码和二进制代码之间的兼容性。PATCH通常称之为补丁版本，通常情况下如果只是对APR函数的修改而不影响API接口的话都会导致PATCH的变化。

目前为止APR的最高版本是1.2.2，最早遵循这种规则的版本号是0.9.0，不过在0.9.0之前，APR还推出了两个版本a8和a9。不过有一点需要注意的是，我们后面描述的版本规则并不适合1.0.0以前的版本。对于1.0.0以前的版本(0.x.y)，APR提供的API是可以任意的改变而没有任何的限制，因此这些版本的变化不遵循后面描述的版本规则。从1.0.0以后的所有版本都遵循。切记。

除非主版本号发生变化，否则如果某个应用程序使用了低版本的APR，那么如果将该版本用高版本的APR替代，应用程序必须能够无错误的编译通过，通常我们称之为前向兼容行；反之很明显，如果应用程序中使用了高版本的APR，那么如果将该版本用低版本的APR替代，则未必能够编译通过，通常我们称之为后向不兼容。

APR的发展中力图总是保持与旧版 本的源代码和二进制版本之间的兼容性。通过源代码兼容，应用程序就可以在使用新版本的APR进行编译的时候不会报错，这样应用程序就不需要为了适应新的 APR而做出调整，从而保持应用开发的一致性和持续性。除非APR的主版本号发生变更。这种兼容性反之则不成立。如果一个应用程序使用较高的MINOR版 本开发，那么很明显，如果将该版本替换为MINOR相对较低的版本进行编译，则成功的可能性应该不是很大。

除了源代码方面的兼容性，APR还 希望能够保持二进制之间的兼容性。通过保持二进制兼容，应用程序可以直接使用高版本的APR库(或者是DLL，或者使so文件)替换低版本的库文件，而不 需要做任何修改，就可以链接成功。与源代码兼容一样，二进制的兼容也是向前兼容，而不保证向后兼容。

下面的表格演示了APR的版本规则策略：

原始版本	新版本	兼容性	原因
2.2.3	2.2.4	前后兼容	PATCH版本号的变化保持前向和后向兼容
2.2.3	2.2.1	前后兼容	PATCH版本号的变化保持前向和后向兼容
2.2.3	2.3.1	向前兼容	次版本号的变化保持向前兼容，但并不保持向后兼容
2.2.3	2.1.7	不兼容	次版本号的降低不能保证向后兼容
2.2.3	3.0.0	不兼容	主版本号的变化不保证兼容性
2.2.3	1.4.7	不兼容	主版本号的变化不保证兼容性

1.2.2版本策略

为了控制APR接口的稳定，APR制定了严格的版本变化策略。

1.2.2.1PATCH版本策略

在前表中我们看到，PATCH的变化并不影响版本的源代码和二进制级别的兼容性，包括向前和向后兼 容，因此，我们很容易看出，PATCH版本变化通常意味着对版本的修修补补，即BUG的修复。这些工作通常被局限于函数内部的修改，或者是API函数内 部，或者是APR内部static函数的变化。任何对API的增加、修改、删除都是不允许的。

1.2.2.1次版本号策略

任何新函数，新变量以及新常量的引入以及任何现有函数的废除都将可能导致次版本号的变化：

1)、新函数的引入

An application coded against an older minor release will still have all of its functions available with their original signatures. Once an application begins to use a new function, however, they will be unable to work against older minor versions.

It is tempting to say that introducing new functions might create incompatibility across minor releases. If an application takes advantage of an API that was introduced in version 2.3 of a library, then it is not going to work against version 2.2. However, we have stated that an any application built against version

2.2 will continue to work for all 2.x releases. Thus, an application that states "requires 2.3 or later" is perfectly acceptable -- the user or administrator simply upgrades the installed library to 2.3. This is a safe operation and will not break any other application that was using the 2.2 library.
In other words, yes an incompatibility arises by mandating that a specific version needs to be installed. But in practice, this will not be a problem since upgrading to newer versions is always safe.
2)、新常量的引入
Similar to functions, all of the original (old) constants will be available to an application. An application can then choose to use new constants to pick up new semantics and features.
3)、函数替换
This gets a bit trickier. The original function **must** remain available at the link-level so that an application compiled against a minor version will continue to work with later minor versions. Further, if an application is *designed* to work with an earlier minor version, then we don't want to suddenly change the requirements for that application. This means that the headers cannot silently map an old function into a newer function, as that would turn an application, say, based on 1.2 into an application requiring the 1.4 or later release.
This means that functions cannot truly be replaced. The new, alternate function can be made available in the header and applications can choose to use it (and become dependent upon the minor release where the function appears).
It is possible to design a set of headers where a macro will always refer to the "latest" function available. Of course, if an application chooses to use this macro, then the resulting compiled-binary will be dependent upon whatever version it was compiled against. This strategy adds the new functionality for applications, yet retains the necessary source and binary compatibility for applications designed or built against previous minor releases.
Constants (enumerated values and preprocessor macros) are **not** allowed to change since an older application will still be using them. Similarly, function signatures at the link-level may not change, so that support for older, compiled applications is maintained.
4)、函数作废
随着APR的升级，APR中的一些API可能将作废，不再使用，但是这些API并不能从APR库中移除。因为一旦API被移除，向后兼容性将被破坏。因此我们能够做的仅仅是宣布其作废。
If you deprecate a function in APR, please mark it as such in the function documentation, using the doxygen "\deprecated" tag. Deprecated functions can only be removed in major releases.
A deprecated function should remain available *through* the original header. The function prototype should remain in the same header, or if moved to a "deprecated functions" header, then the alternate header should be included by the original header. This requirement is to ensure that source compatibility is retained.

Finally, if you are deprecating a function so that you can change the name of the function, please use the method described above under "Replacing functions", so that projects which use APR can retain binary compatibility.
Note that all deprecated functions will be removed at the next major version bump.

1.2.2.3 主版本号策略

下面的任何一种变化都将可能导致主版本号的变化：
1)、常量的移除或者更改
2)、函数移除或者作为
3)、fold together macro-ized function replacements

1.2.3 版本检查

由于APR严格的版本控制策略，使得应用程序在使用APR库之前必须能够检测使用的APR库的版本号。APR允许在编译以及使用APR的时候检测它的版本号。

1.2.3.1 编译时版本检查

Libraries should make their version number available as compile-time constants. For example:
#define FOO_MAJOR_VERSION 1
#define FOO_MINOR_VERSION 4
#define FOO_PATCH_VERSION 0
The above symbols are the minimum required for this specification.
An application that desires, at compile-time, to decide on whether and how to use a particular library feature needs to only check two values: the major and the minor version. Since, by definition, there are no API changes across patch versions, that symbol can be safely ignored. Note that any kind of a check for a minimum version will then pin that application to at least that version. The application's installation mechanism should then ensure that that minimal version has been installed (for example, using RPM dependency checks).
If the feature changes across minor versions are source compatible, but are (say) simply different choices of values to pass into the library, then an application can support a wider variety of installed libraries if it avoids compile-time checks.

1.2.3.2 执行时版本检查

A library meeting this specification should support a way for an application to determine the library's version at *run-time*. This will usually be emboded as a simple function which returns the MAJOR, MINOR, and PATCH triplet in some form. Run-time checks are preferable in all cases. This type of check enables an application to run against a wider variety of minor releases of a library (the application is *"less coupled"* to a particular library release). Of course, if an application requires a function that was introduced in a later, minor release, then the application will require that, at least, that release is installed on the target system.

Run-time checks are particurly important if the application is trying to determine if the library has a particular bug that may need to be worked around, but has been fixed in a later release. If the bug is fixed in a patch release, then the only avenue for an application is to perform a runtime check. This is because an application cannot require a specific patch level of the library to be installed -- those libraries are perfectly forward and backwards compatible, and the administrator is free to choose any patch release, knowing that all applications will continue to function properly. If the bug was fixed in a minor release, then it is possible to use a compile-time check, but that would create a tighter coupling to the library.

1.2.3.3 版本API

与前面的版本规则定义一致，APR中定义了数据结构apr_version_t来描述版本规则：

```
typedef struct {
    int major;      /**< major number */
    int minor;      /**< minor number */
    int patch;      /**< patch number */
    int is_dev;     /**< is development (1 or 0) */
} apr_version_t;
```

major是当前APR版本的主版本号，minor则是次版本号，patch对应的则是APR的补丁号。es_dev则描述了当前APR库的状态：开发版还是发行版，分别对应1和0。

一旦定义了apr_version_t结构，APR就将使用它作为基本的版本控制单位。APR中提供了函数apr_version和apr_version_string分别设置和返回APR的版本。

```
APR_DECLARE(void) apr_version(apr_version_t *pvsn)
{
    pvsn->major = APR_MAJOR_VERSION;
    pvsn->minor = APR_MINOR_VERSION;
    pvsn->patch = APR_PATCH_VERSION;
#ifdef APR_IS_DEV_VERSION
    pvsn->is_dev = 1;
#else
    pvsn->is_dev = 0;
#endif
}
```

apr_version函数仅仅就是设置apr_version_t结构中的各个成员。对于每一个APR版本，APR都会将当前的版本号分别用三个常量定义在version.h中，比如，如果版本号是2.2.0，则常量定义应该如下：

```
#define APR_MAJOR_VERSION       2
#define APR_MINOR_VERSION       2
#define APR_PATCH_VERSION       0
```

apr_version_string函数仅仅是返回APR_VERSION_STRING宏，该宏定义为：

```
#define APR_VERSION_STRING \
     APR_STRINGIFY(APR_MAJOR_VERSION) "." \
     APR_STRINGIFY(APR_MINOR_VERSION) "." \
```

```
    APR_STRINGIFY(APR_PATCH_VERSION) \
    APR_IS_DEV_STRING
```

APR_STRINGIFY 用于将给定的数值转换为字符串，因此 APR_VERSION_STRING 宏就是无非将 APR_MAJOR_VERSION，APR_MINOR_VERSION，APR_PATCH_VERSION 分别转换为字符串，再用"."连接起来，最后的形式应该为“2.2.0”。

尽管一般情况下，APR 的版本号是“x.x.x”的格式，不过在 Window 的资源文件.rc 中通常是“x,x,x”格式，因此 APR 中也提供了 APR_VERSION_STRING_CSV 来提供这种格式的版本号：

```
#define APR_VERSION_STRING_CSV APR_MAJOR_VERSION ##, \
                               ##APR_MINOR_VERSION ##, \
                               ##APR_PATCH_VERSION
```

##宏用于将变量与特定字符进行连接形成新的字符串。在后面的部分，我们会不断看到它的用法

在一些情况下，应用程序需要使用的 APR 版本达到一定的版本号，为此，APR 中提供了 APR_VERSION_AT_LEAST 宏用以检测给定的 APR 库是否达到给定的版本要求：

```
#define APR_VERSION_AT_LEAST(major,minor,patch)                    \
(((major) < APR_MAJOR_VERSION)                                     \
 || ((major) == APR_MAJOR_VERSION && (minor) < APR_MINOR_VERSION) \
 || ((major) == APR_MAJOR_VERSION && (minor) == APR_MINOR_VERSION && (patch) <=
APR_PATCH_VERSION))
```

如果我们希望当前使用的 APR 库的版本不的低于”1.2.0”，那么我们就使用使用 APR_VERSION_AT_LEAST(1,2,0)对当前的 APR 库版本进行检查。

Apache APR 可移植运行库简介(2)

1.3.1 获取 APR

编译 APR 的第一个步骤就是获取 APR 开发包。通常情况下，你可以到 APR 的官方网站 http://apr.apache.org/download.cgi 去下载。

一般情况下，APR 开发包很容易理解为仅仅是一个开发包，不过事实上并不是。目前，完整的 APR 实际上包含了三个开发包：apr、apr-util 以及 apr-iconv，每一个开发包分别独立开发，并拥有自己的版本。

apr 开发包位于目录${APR}\apr 下，其中包含了一些通用的开发组件，包括 mmap，文件等等，前面已经描述过。

apr-util 开发包位于目录${APR}\apr-util 下，该目录中也是包含了一些常用的开发组件。这些组件与 apr 目录下的相比，它们与 apache 的关系更加密切一些。比如存储段和存储段组，加密等等，具体的各个组件的含义如下表所示：

组件名称	文件夹名称	描述
buckets	/srclib/apr-util/buckets	存储段和存储段组
crypto	/srclib/apr-util/crypto	加密和解密
hooks	/srclib/apr-util/hooks	apache 挂钩
dbd	/srclib/apr-util/dbd	数据库连接管理
dbm	/srclib/apr-util/dbm	
ldap	/srclib/apr-util/ldap	轻量级目录访问协议
strmatch	/srclib/apr-util/strmatch	字符串匹配，包括普通字符串匹配以及正则表达式匹配，正则表达式匹配中使用 prec 库
uri	/srclib/apr-util/uri	uri 操作例程

xml	/srclib/apr-util/xml	xml 支持例程，其中使用 expat 作为 xml 解析器
xlate	/srclib/apr-util/xlate	i18n 转换库
encoding	/srclib/apr-util/encoding	编码转换库，其中实现了各种编码之间的转换
misc	/srclib/apr-util/misc	大杂烩

apr-util 的当前版本为 1.2.2，最早的版本为 0.9.1。

apr-iconv 包中的文件主要用于实现 iconv 编码。目前的大部分编码转换过程都是与本地编 码相关的。在进行转换之前必须能够正确地设置本地编码。因此如果两个非本地编码 A 和 B 需要转换，则转换过程大致为 A->Local 以及 Local->B 或者 B->Local 以及 Local->A。

XPG2 标准中另外定义了一组全新的函式接口 ICONV，该接口是一种更广义的字集转换系统。也是一个与本地编码无关的字集转换系统。APR 中也支持这种字集转换系统，为此它提供了完整的实现。Apr-iconv 正是为此目的而产生。

apr-iconv 的当前版本为 1.1.1，最早的版本从 0.9.2 开始。

1.3.2APR 的目录组织

从 http://www.apache.org/上下载 apr-1.1.1.tar.gz 到本地解压后，可以发现 APR 的目录结构很清晰。

1) 所有的头文件都放在$(APR)/include 目录中;

2) 所有功能接口的实现都放在各自的独立目录下，如 threadproc、mmap 等目录中，而 这些目录的子目录中包含实际的实现代码。Apache 中划分子目录的根据就是编译代码所在的平台，目前 APR 能够支持 Unix、BeOS、Windows 以及 OS2 四种平台，因此正常情况下，每一个目录中会包含这四个子目录。比如文件 I/O 的目录就如下所示：

```
apr
 |
  -> file_io
         |
          -> unix            The Unix and common base code
         |
          -> win32           The Windows code
         |
          -> os2             The OS/2 code
```

在上面的四种子目录中，unix 是一个比较特殊的目录。由于 Unix 的种类很多，比如 FreeBSD，Linux 等等，原则上应该都为这些平台分别建立各自的子目录，然后分别实现代码。不过所有的 Unix 都大同小异，如果为了稍许的差异就 “大动干戈”，不太划算，因此 APR 中将所有的 Unix 的平台的操作合并到一起，而这些平台这些的差异就用前面的预定义宏来区别。目前的 unix 目录中以 POSIX 为主，同时兼顾 System V。上面的文件 I/O 操作中明显的缺少了 BeOS 平台的实现，那是因为 BeOS 的实现合并到 Unix 目录中去了。

另外两个特殊的目录就是 include 和 test 目录了。include 目录中包含了所有的外部使 用所需要的头文件。其中 APR.h 和 apr_private.h 是两个特殊的文件，如果要使用 APR，必须包含 apr.h 头文件。但在原文件中并看不到这 两个文件。事实上，这两个文件都是自动生成的。由于 windows 和 netware 平台下并不使用 autoconf，因此 APR 在 windows 和 netware 平台下的行为与其余所有平台都不相同。在 UNIX 上，apr_private.h(APR 私有文件，仅仅 APR 内部使用)和 apr.h(APR 公共文件，可以其余的文件使用)实际上在代码中并不存在，它们都是由 autoconf

从 acconfig.h 和 apr.h.in 中自动 生成的。而在 Window 中，这两个文件都由 apr_private.hw(apr_private.hwn)和 apr.hw(apr_private.hwn)中自动生成。
test 则是测试程序的目录。每一个 APR 类型在使用之前都必须经过测试。事实上对于 APR 的使用者而言，这个目录还有另外的一个好处，就是快速掌握 APR 类型的使用。每一个测试例子都给出了具体类型的使用方法。
3) 此外就是相关平台构建工具文件如 Makefile.in，configure.in 等等。

1.3.3APR 构建

在 Window 和 Unix 上编译 APR 的方法不太相同，我们分开来描述。

1.3.3.1 Unix 上编译

We've attempted to ensure that compiling apr, apr-iconv and apr-util distribution tarballs requires nothing more than what comes installed by default on various UNIX platforms.
All you should have to do is this:

```
./configure
make
make install
```

As of this writing, APR is not quite ready to be installed as a system-wide shared library; it currently works best when tied directly to the application using it.
Note that if you are compiling directly from the SVN repository, you'll need GNU autoconf and GNU libtool installed, and you'll need to run ./buildconf before running the commands listed above.

1.3.3.2 Window 平台上编译

The apr-util/aprutil.dsw workspace builds the .dsp projects of the Apache server listed with dependent projects preceeding their dependencies:
apr-util\aprutil.dsp
apr-util\libaprutil.dsp
apr-util\uri\gen_uri_delims.dsp
apr-util\xml\expat\lib\xml.dsp
apr-iconv\apriconv.dsp
apr-iconv\libapriconv.dsp
apr\apr.dsp
apr\libapr.dsp
The libXXX projects create .dll targets, dynamic shared libraries. Their non-libXXX counterparts create static .lib targets.
To compile code for the libraries, the consuming compiliation must include the apr/include and apr-util/include directories in their include file search paths. To target the static .lib versions of the library, the consuming compiliation must define the macros APR_DECLARE_STATIC and APU_DECLARE_STATIC. This prevents the apr and apr-util symbols from being tagged as __declspec(dllimport), eliminating compiliation warnings and speeding up execution.

在Window平台上要成功编译apr、apr-iconv以及apr-util，必须具备一下的几个条件：
1)、可用的微软编译器：比如微软的Visual C++ 5.0或者更高的版本，比如Visual C++6.0，Microsoft Visual Studio.NET 2002，Microsoft Visual Studio.NET 2003(必须具备Visual C++ .NET编译器)。
对于Visual C++ 5.0的用户，为了能够使用一些APR中的新特性，你必须更新Windows平台开发包(Windows Platform SDK)。对于Visual C++ 6.0则没有这些多余的事情，因为这些SDK随Visual C++6.0一起发布了。如果没有这些新的SDK，使用MSVC++5.0编译的时候，编译中会出现大量新特性不支持的警告，甚至完全编译失败。至于具体 的SDK，你可以到Window的网站上去下载。
目前最近的APR版本是2.2.0，不过你如果使用老一些的APR，比如1.1.1以前，那么你还 需要awk。你可以到http://cm.bell-labs.com/cm/cs/who/bwk/awk95.exe去下载二进制可执行文件。不过从 1.1.1以后的版本就可以省去这个麻烦了。
2)、正确的目录布局
除了必要的编译工具之外，APR开发包还必须具备正确地目录布局。apr，apr-util以及apr-iconv必须同时具备，并且它们必须位于同一目录之下，比如：

C:\work\apr\
C:\work\apr-iconv\
C:\work\apr-util\

对于发行版本，直接将发行的文件包解压到指定目录下即可；而对于开发版本，你必须能够从subversion中自行检出，Window平台下，APR推荐的SVN是TortoiseSVN。
万事具备，只欠东风。现在你可以编译APR了。你可以选择两种方式，或者是命令行编译，或者是使用IDE编译。

■ 命令行方式编译

使用命令行进行编译的第一步就是修改 vcvars32.bat，通常情况下该文件位于C:\Program Files\Microsoft Visual Studio\VC98\Bin目录下，其中C:\Program Files\Microsoft Visual Studio\是VC的安装目录，它根据安装目录的不同会不同。

```
"C:\Program Files\DevStudio\VC\Bin\vcvars32.bat"
```

If necessary, you will also need to prepare the Platform SDK environment:
如果有必要，你还必须准备Platform SDK相关的环境变量，这个通常修改setenv.bat文件就可以实现：

```
"C:\Program Files\Platform SDK\setenv.bat"
```

一旦设置完毕，你首先必须切换到apr-util目录下，然后简单的执行下面的指令就可以编译APR了。

```
msdev aprutil.dsw /MAKE \
    apriconv - Win32 Release" \
    apr - Win32 Release" \
    libapr - Win32 Release" \
    gen_uri_delims - Win32 Release" \
    xml - Win32 Release" \
    "aprutil - Win32 Release" \

msdev aprutil.dsw /MAKE \
    libapr - Win32 Release" \
    libapriconv - Win32 Release" \
```

gen_uri_delims - Win32 Release" \
xml - Win32 Release" \
libaprutil - Win32 Release" \

这两个命令都可以编译APR，不过它们的区别就是后 一个编译结果是动态链接库.dll，而前者则是静态连接库.lib。不过它们编译的都是发行版，如果你需要编译调试版本，只需要简单的将命令中的 ”Release”替换为”Debug”即可，这样，你就可以方便的进行调试了。

For Visual Studio C++ 5.0 command line users: Only the .dsp files are maintained within SVN. Win32 .mak files are NOT maintained in SVN, due to the tremendous waste of reviewer's time. Therefore, you cannot rely on the NMAKE commands above to build revised .dsp project files unless you then export all .mak files yourself from the project.

■ IDE方式编译

与命令行编译相比，使用IDE编译更简单。事实上，在使用的时候我更倾向于使用IDE进行编译。：）。不过如果你是那种什么都得挖到底的人，前面的命令行编译你也可以试试。

IDE编译，你需要的仅仅是一个dsw工作区aprutil.dsw，它位于apr-util目录下，该工作区中包含了完整编译整个APR所需要的所有的.dsp项目文件，以及各个dsp文件之间的依赖关系，以确保它们之间的正确的编译顺序。

打开aprutil.dsw，整个工作区如下图所示：

从上图中可以看出，apr-util.dsw工作区中包含了十个dsp工作项目。Apr、 apriconv以及aprutil分别对应静态编译库，而libXXX则对应的是动态编译库。默认情况下，编译的是apr项目，不过你可以通过 project->Set Active Project选择你需要编译的实际项目：

编译后aprXXX项目生成的静态库通常位于对应目录下的LibD目录中，而libXXX项目生成的动态库则位于Debug或者Release目录下，具体取决于当前是发行版本还是调试版本。

Apache APR可移植运行库简介(3)

我们首先make install一下，比如我们在Makefile中指定prefix=$(APR)/dist，则make install后，在$(APR)/dist下会发现4个子目录，分别为bin、lib、include和build，其中我们感兴趣的只有 include和lib。下面是一个APR app的例子project。

该工程的目录组织如下：

```
$(apr_path)
dist
    - lib
    - include
 - examples
    - apr_app
      - Make.properties
      - Makefile
      - apr_app.c
```

我们的Make.properties文件内容如下：

```
#
# The APR app demo
#
CC              = gcc -Wall
BASEDIR         = $(HOME)/apr-1.1.1/examples/apr_app
APRDIR          = $(HOME)/apr-1.1.1
APRVER          = 1
APRINCL         = $(APRDIR)/dist/include/apr-$(APRVER)
APRLIB          = $(APRDIR)/dist/lib
DEFS            = -D_REENTRANT -D_POSIX_PTHREAD_SEMANTICS -D_DEBUG_
LIBS            = -L$(APRLIB) -lapr-$(APRVER) \
                  -lpthread -lxnet -lposix4 -ldl -lkstat -lnsl -lkvm -lz -lelf -lm
-lsocket  - ladm
INCL            = -I$(APRINCL)
CFLAGS          = $(DEFS) $(INCL)
```

Makefile文件内容如下：

```
include Make.properties
TARGET  = apr_app
OBJS    = apr_app.o
all: $(TARGET)
$(TARGET): $(OBJS)
        $(CC) ${CFLAGS} -o $@ $(OBJS) ${LIBS}
clean:
        rm -f core $(TARGET) $(OBJS)
```

而apr_app.c文件采用的是$(apr_path)/test目录下的proc_child.c文件。编译运行一切OK。

1.5 APR的可移植性

正如前面所描述，APR的目前的首要目标就是设计为一个跨平台的通用库，因此在APR的整个设计过程中无不体现了可移植的思想，APR附带一个简短的设计文档，文字言简意赅，其中很多的移植设计思想都值得我们所借鉴，主要从四个方面谈。

1.5.1APR类型

为了支持可移植性，APR 中的一个策略就是尽量使用APR自定义的类型来代替平台相关类型。这样的好处很多，比如便于代码移植，避免数据间进行不必要的类型转换(如果你不使用 APR自定义的数据类型，你在使用某些APR提供的接口时，就需要进行一些参数的类型转换)；自定义数据类型的名字更加具有自描述性，提高代码可读性。 APR提供的基本自定义数据类型包括apr_byte_t，apr_int16_t，apr_uint16_t，apr_size_t等 等。通常情况下这些类型都定义在apr.h中，不过你找遍整个APR包也不会找到apr.h这个文件，不过include目录下倒是存在类似于apr.h 的apr.h.in和apr.hw，这两个文件是生成apr.h的模版，在apr.h.in中通用APR类型定义如下：

```
typedef unsigned char              apr_byte_t;
typedef @short_value@           apr_int16_t;
typedef unsigned @short_value@       apr_uint16_t;
```

```
typedef @int_value@              apr_int32_t;
typedef unsigned @int_value@           apr_uint32_t;
typedef @long_value@            apr_int64_t;
typedef unsigned @long_value@         apr_uint64_t;
typedef @size_t_value@            apr_size_t;
typedef @ssize_t_value@         apr_ssize_t;
typedef @off_t_value@            apr_off_t;
typedef @socklen_t_value@              apr_socklen_t;
```

@xxx@变量的值是可变的，不同的平台其值可能不一样。其值由configure配置过程自动生成，configue脚本中设置@xxx@变量的部分大致如下：

```
AC_CHECK_SIZEOF(char, 1)
AC_CHECK_SIZEOF(short, 2)
AC_CHECK_SIZEOF(int, 4)
AC_CHECK_SIZEOF(long, 4)
AC_CHECK_SIZEOF(long long, 8)

if test "$ac_cv_sizeof_short" = "2"; then
    short_value=short
fi
if test "$ac_cv_sizeof_int" = "4"; then
    int_value=int
fi
if test "$ac_cv_sizeof_int" = "8"; then
    int64_value="int"
    long_value=int
elif test "$ac_cv_sizeof_long" = "8"; then
    int64_value="long"
    long_value=long
elif test "$ac_cv_sizeof_long_long" = "8"; then
    int64_value="long long"
    long_value="long long"
elif test "$ac_cv_sizeof_longlong" = "8"; then
    int64_value="__int64"
    long_value="__int64"
else
    AC_ERROR([could not detect a 64-bit integer type])
fi
if test "$ac_cv_type_size_t" = "yes"; then
    size_t_value="size_t"
else
    size_t_value="apr_int32_t"
fi
if test "$ac_cv_type_ssize_t" = "yes"; then
    ssize_t_value="ssize_t"
```

```
else
    ssize_t_value="apr_int32_t"
fi
   .. ..
```

Configure 的具体的细节并不是本书描述的细节，如果你想了解更多细节，你可以去阅读 GNU 的 AutoConf、AutoMake 等使用手册。

不同的操作系统中各个变量的值如下表所示：

变量类型	硬件平台							
	I686	I386	alpha	IA64	M68k	MIPS	Sparc	Spar64
apr_byte_t	1	1	1	1	1	1	1	1
apr_int16_t	2	2	2	2	2	2	2	2
apr_uint16_t	2	2	2	2	2	2	2	2
apr_int32_t	4	4	4	4	4	4	4	4
apr_uint32_t	4	4	4	4	4	4	4	4
apr_int64_t	4	4	8	8	4	4	4	4
apr_uint64_t	4	4	8	8	4	4	4	4

不过不同的操作系统中，定义各不相同，在 Red Hat 9.0 Linux 中，生成的定义如下：

```
typedef short                   apr_int16_t;        //16 位整数
typedef unsigned short          apr_uint16_t;       //16 位无符号整数
typedef int                     apr_int32_t;        //32 位整数
typedef unsigned int            apr_uint32_t;       //32 位无符号整数
typedef long long               apr_int64_t;        //64 位整数
typedef unsigned long long      apr_uint64_t;       //64 位无符号整数

typedef size_t                  apr_size_t;         //
typedef ssize_t                 apr_ssize_t;
typedef off64_t                 apr_off_t;
typedef socklen_t               apr_socklen_t;      //套接字长度
```

通用数据类型的另外一个定义之处就是文件 apr_portable.h 中，APR 中提供了通用的数据类型以及对应的操作系统依赖类型如下表：

通用类型	含义	Win32 类型	BEOS 类型	UNIX
apr_os_file_t	文件类型	HANDLE	int	Int
apr_os_dir_t	目录类型	HANDLE	dir	DIR
apr_os_sock_t	套接字类型	SOCKET	int	int
apr_os_proc_mutex_t	进程互斥锁	HANDLE	apr_os_proc_mutex_t	pthread_mutex_t
apr_os_thread_t	线程类型	HANDLE	thread_id	pthread_t
apr_os_proc_t	进程类型	HANDLE	thread_id	pid_t
apr_os_threadkey_t	线程 key 类型	DWORD	int	pthread_key_t
apr_os_imp_time_t		FILETIME	struct timeval	struct timeval
apr_os_exp_time_t		SYSTEMTIME	struct tm	tm
apr_os_dso_handle_t	DSO 加载	HANDLE	image_id	void*

apr_os_shm_t	共享内存	HANDLE	void*	void*

一旦定义了这些通用的数据类型，APR 不再使用系统类型，而是上述的 APR 类型。不过由于系统底层仍然使用系统类型，因此在使用通用类型的时候一项必须的工作就是用实际的类型来真正替代通用类型，比如 apr_os_file_t，如果是 Win32 平台，则必须转换为 HANDLE。对于上面表格的每一个通用数据类型，Apache 都提供两个函数支持这种转换：

```
APR_DECLARE(apr_status_t) apr_os_XXX_get(…);
APR_DECLARE(apr_status_t) apr_os_XXX_put(…);
```

get 函数用于将通用的数据类型转换为特定操作系统类型；而 put 函数则是将特定操作系统类型转换为通用数据类型。比如对于 file 类型，则对应的函数为：

```
APR_DECLARE(apr_status_t) apr_os_file_get(apr_os_file_t *thefile,
                                          apr_file_t *file);
APR_DECLARE(apr_status_t) apr_os_file_put(apr_file_t **file,
                                          apr_os_file_t *thefile,
                                          apr_int32_t flags, apr_pool_t *cont);
```

前者将通用的文件类型 apr_os_file_t 转换为特定操作系统类型 apr_file_t，后者则是将 apr_file_t 转换为 apr_os_file_t。

在后面的分析中我们可以看到，对于每一个组件类型，比如 apr_file_t 中都会包含系统定义类型，APR 类型都是围绕系统类型扩充起来的，比如 apr_file_t，在 Unix 中为：

```
struct apr_file_t
{
    int filedes;                //UNIX 下的实际的文件系统类型
    … …
}
```

而在 Window 中则是：

```
struct apr_file_t
{
    HANDLE filedes;     //Window 下的实际的文件系统类型
    ……
}
```

因此 apr_os_file_get 函数无非就是返回结构中的文件系统类型，而 apr_os_file_put 函数则无非就是根据系统文件类型创建 apr_file_t 类型。

类似的例子还有 apr_os_thread_get 和 apr_os_thread_put 等等。

1.5.2 函数

APR 中函数可以分为两大类：内部函数和外部接口函数。顾名思义，内部函数仅限于 APR 内部使用，外部无法调用；而外部结构函数则作为 API 结构，由外部程序调用。

1.5.2.1 内部函数

APR 中所有的内部函数都以 static 进行修饰。通常理解 static 只是指静态存储的概念，事实上在里面 static 包含了两方面的含义。

1)、在固定地址上的分配，这意味着变量是在一个特殊的静态区域上创建的，而不是每次函数调用的时候在堆栈上动态创建的，这是 static 的静态存储的概念。

2)、另一方面，static 能够控制变量和函数对于连接器的可见性。一个 static 变量或者函 数，对于特定的编译单元来说总是本地范围的，这个范围在 C 语言中通常是指当前文件，超过这个

范围的文件或者函数是不可以看到 static 变量和函数的，因 此编译器也无法访问到这些变量和函数，它们对编译器是不可见的。因此内部函数是不允许被直接调用的，任何直接调用都导致“尚未定义”的错误。不过潜在的好 处就是，内部函数的修改不影响 API 接口。
static 的这两种用法 APR 中都存在，但是第二种用法较多。

1.5.2.2 外部 API 函数

对于 APR 用户而言，它们能够调用的只能是 APR 提供的 API。要识别 APR 中提供的 API 非常的简单，如果函数是外部 API，那么它的返回值总是用 APR_DECLARE 或者 APR_DECLARE_NONSTD 进行包装，比如：

```
APR_DECLARE(apr_hash_t *) apr_hash_make(apr_pool_t *pool);
APR_DECLARE(int) apr_fnmatch_test(const char *pattern);
```

APR_DECLARE 和 APR_DECLARE_NONSTD 是两个宏定义，它们在 apr.h 中定义如下：

```
#define APR_DECLARE(type)            type
#define APR_DECLARE_NONSTD(type)     type
```

APR_DECLARE 和 APR_DECLARE_NONSTD 到底是什么意思呢？为什么要将返回类型封装为宏呢？在 apr.h 中有这样的解释:

```
/**
 * The public APR functions are declared with APR_DECLARE(), so they may
 * use the most appropriate calling convention. Public APR functions with
 * variable arguments must use APR_DECLARE_NONSTD().
 *
 * @remark Both the declaration and implementations must use the same macro.
 * @example
 */
/** APR_DECLARE(rettype) apr_func(args)
 * @see APR_DECLARE_NONSTD @see APR_DECLARE_DATA
 * @remark Note that when APR compiles the library itself, it passes the
 * symbol -DAPR_DECLARE_EXPORT to the compiler on some platforms (e.g. Win32)
 * to export public symbols from the dynamic library build.\n
 * The user must define the APR_DECLARE_STATIC when compiling to target
 * the static APR library on some platforms (e.g. Win32.) The public symbols
 * are neither exported nor imported when APR_DECLARE_STATIC is defined.\n
 * By default, compiling an application and including the APR public
 * headers, without defining APR_DECLARE_STATIC, will prepare the code to be
 * linked to the dynamic library.
 */
   #define APR_DECLARE(type)            type
/**
 * The public APR functions using variable arguments are declared with
 * APR_DECLARE_NONSTD(), as they must follow the C language calling convention.
 * @see APR_DECLARE @see APR_DECLARE_DATA
 * @remark Both the declaration and implementations must use the same macro.
 * @example
 */
```

```
/** APR_DECLARE_NONSTD(rettype) apr_func(args, ...);
 */
    #define APR_DECLARE_NONSTD(type)      type
```

从上面的解释中我们可以看出“APR 的固定个数参数公共函数的声明形式 APR_DECLARE(rettype) apr_func(args)；而非固定个数参数的公共函数的声明形式为 APR_DECLARE_NONSTD(rettype) apr_func(args, ...);”。

在 apr.h 文件中解释了这么做就是为了在不同平台上编译时使用“the most appropriate calling convention”，这里的“calling convention”是一术语，翻译过来叫“调用约定”。

我们知道函数调用是通过栈操作来完成的，在栈操作过程中需要函数的调用者和被调用者在下面的两个问题上做出协调，达成协议：

a) 当参数个数多于一个时，按照什么顺序把参数压入堆栈

b) 函数调用后，由谁来把堆栈恢复原来状态

c) 产生函数修饰名的方法

在像 C/C++这样的中、高级语言中，使用“调用约定”来说明这两个问题。常见的调用约定有：__stdcall、__cdecl、__fastcall、thiscall 和 naked call。

__stdcall 调用约定：函数的参数自右向左通过栈传递，被调用的函数在返回前清理传送参数的内存栈。

__cdecl 是 C 和 C++程序的缺省调用方式。每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用_stdcall 函数的大。函数采用从右到左的压栈方式。***注意：对于可变参数的成员函数，始终使用__cdecl 的转换方式***。

__fastcall 调用约定规定通过寄存器来传送参数的(实际上，它用 ECX 和 EDX 传送前两个双字(DWORD)或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈)。

thiscall 仅仅应用于"C++"成员函数。this 指针存放于 CX 寄存器，参数从右到左压。thiscall 不是关键词，因此不能被程序员指定。

naked call 采用上述调用约定时，如果必要的话，进入函数时编译器会产生代码来保存 ESI，EDI，EBX，EBP 寄存器，退出函数时则产生代码恢复这些寄存器的内容。naked call 不产生这样的代码。naked call 不是类型修饰符，故必须和_declspec 共同使用。

另外不同的调用约定对于函数内部修饰名处理的方法也不一样。所谓修饰名是 C 或者 C++函数在内部编译和链接的时候产生的唯一的标识名称。

对于 C 语言而言，__stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个"@"符号和其参数的字节数，格式为_functionname@number,例如 ：function(int a, int b)，其修饰名为：_function@8

__cdecl 调用约定仅在输出函数名前加上一个下划线前缀，格式为_functionname。

__fastcall 调用约定在输出函数名前加上一个"@"符号，后面也是一个"@"符号和其参数的字节数，格式为@functionname@number。

如果是 C++，不同调用约定处理要稍微复杂一点。由于 Apache 是基于 C 语言开发，因此本处不再描述。

1.5.2.3 内存池参数

关于函数的最后一个问题就是它的参数，如果函数内部需要分配空间，那么你就可以看到参数的参数中肯定包含一个 apr_pool_t 参数，比如：

```
APR_DECLARE(apr_status_t) apr_shm_attach(apr_shm_t **m,
                                         const char *filename,
```

```
                                        apr_pool_t *pool);
```

由于 Apache 服务器所具有的一 些特性，APR 中并没有使用普通的 malloc/free 内存管理策略，而是使用了自行设计的内存池管理策略。APR 中所有的需要的内存都不再直接使用 malloc 分配，然后首先分配一块足够大的内存块，然后每次需要的时候再从中获取；当内存不再使用的时候也不是直接调用 free，而是直接归还给内存 池。只有当内存池本身被释放的时候，这些内存才真正的被 free 给操作系统。Apache 中使用 apr_pool_t 描述一个内存池，因此毫无疑问，由于这种特殊的内存分配策略，对于任何一个函数，如果你需要使用内存，那么你就应该指定内存所源自的内存池。这就是为什么大部分函数参数中都具有 apr_pool_t 的原因。关于内存池的详细细节，我们在第二章详细讨论。

Apache APR 可移植运行库简介(4)

1.5.3 错误处理

大型的系统程序的错误处理是十分重要的，APR 作为一个通用库接口集合详细的说明了使用 APR 时如何进行错误处理。

1.5.3.1 错误码定义

错误处理的第一步就是定义返回码，包括“错误码和状态码分类”。APR 的函数大部分都返回 int 类型作为返回码的，不过为了更明确易懂，APR 在 apr_errno.h 中使用 **typedef int apr_status_t** 将其进行了重新定义。它在一起定义的还有 apr 所用的所有错误码和状态码。如果一个 APR 函数绝对的不可能出错，那么此时就允许不返回 ap_status_t 错误码，只需要返回 void 类型即可。不过 APR 中大部分的函数都是返回 apr_status_t 值。

APR 中的返回码的定义并不是随意的，没有规则的。相反，APR 给出了定义返回码的严格的规定。APR 中根据返回信息的相似性将它们分为七大类，分别定义如下所示：

```
#define APR_OS_ERRSPACE_SIZE     50000
#define APR_SUCCESS                                  0
#define APR_OS_START_ERROR                 20000
#define APR_OS_START_STATUS                 (APR_OS_START_ERROR + APR_OS_ERRSPACE_SIZE)
#define APR_OS_START_USERERR     (APR_OS_START_STATUS + APR_OS_ERRSPACE_SIZE)
#define APR_OS_START_CANONERR (APR_OS_START_USERERR \
                                          + (APR_OS_ERRSPACE_SIZE * 10))
#define APR_OS_START_EAIERR                 (APR_OS_START_CANONERR +
APR_OS_ERRSPACE_SIZE)
#define APR_OS_START_SYSERR                 (APR_OS_START_EAIERR +
APR_OS_ERRSPACE_SIZE)
```

正常情况下，函数返回 APR_SUCCESS 作为成功标志。否则根据情况返回上述类别中的任一种。每一大类的返回信息又可以具体细分为实际的值。在 Apache 中，每一类返回信息所允许的数目由 APR_OS_ERRSPACE_SIZE 决定，目前为 50000，APR_OS_CANONERR 为 500000。另一方面，APR_OS_START_ERROR、 APR_OS_START_STATUS、APR_OS_START_USEERR、APR_OS_START_CANONERR、APR_OS_START_EAIERR 和 APR_OS_START_SYSERR，它们每个都拥有自己独自的偏移量，具体偏移量的值以及含义如下表描 述：

错误名称	含义
0	每个平台都有 0，但是都没有实际的定义，0 又的确是一个 errno value 的 offset，但是它是“匿名的”，它不像 EEXIST 那样有着可以“自描述”的名字。

APR_OS_START_ERROR	该定义是平台相关的，不同平台的值可能不同，它定义了 APR 中所允许的**错误码**的起始偏移量，即 20000，这意味着所有的错误码值不能低于 20000。至于错误码，它可以是导致 APR 函数失败的任何原因。在这个范围内定义的所有错误码形式都必须是 APR_E*格式，比如 APR_ENOSTAT、APR_ENOSOCKET、APR_ENOPOOL。该类别中允许定义最多 50000 种错误码。
APR_OS_START_STATUS	该定义也是平台相关的，它定义了 APR 中所允许的返回**状态值**的 起始偏移量，即 APR_OS_START_ERROR + APR_OS_ERRSPACE_SIZE，即 70000 开始。不过需要注意的是，状态值并不能表示失败还是成功。如果要表示返回成功，必须使用 APR_SUCCESS 返回。在这个范围的状态码形式都必须是 APR_*格式，比如 APR_DETACH、APR_INCHILD。
APR_OS_START_USERERR APR_OS_START_USEERR	该定义也是平台相关的。当用户使用 APR 库的时候，如果它希望定义 APR 返回码之外的其余的自定义返回码，那么这些返回码必须从 APR_OS_START_USEERR 开始，即 APR_OS_START_STATUS + APR_OS_ERRSPACE_SIZE，即从 120000 开始。
APR_OS_START_CANONERR	APR_OS_START_CANONERR is where APR versions of errno values are defined on systems which don't have the corresponding errno.对于这类返回码的定义通常如下： #ifdef EEXIST #define APR_EEXIST EEXIST #else #define APR_EEXIST (APR_OS_START_CANONERR + 2) #endif
APR_OS_START_EAIERR	在 使用 socket 编程的时候如果调用 getaddrinfo()函数，该函数会返回一系列的以 EAI_*开始的错误码，比如 EAI_AGAIN，EAI_FAIL 等等。这些不规则的错误码最终都要转换为 APR 中对应的返回码。这些转换码从 APR_OS_START_EAIERR 开始，最多允许 50000 个(当然事实上肯定没有这么多)。
APR_OS_START_SYSERR	由 于 APR 必须保持跨平台的特性，因此不同的操作系统平台肯定有自己所独有的一些返回码，这些返回码不具有移植性，是与平台相关的。尽管如此，他们也必须转 换为 APR 内部返回码。APR_OS_START_SYSERR 指定了定义这些不具有移植性返回码的起始偏移，为 720000，容量为 50000。

各返回信息起始便移量及其区别可以用下图描述：

从上表可以看出，所有的 APR 定义 的返回码都是从 APR_OS_START_ERROR 开始，那么 0 到 APR_OS_START_ERROR 之间将近 20000 个空位岂不是浪费了？事实上 这部分空间并没有浪费。我们后面所描述的”native error”会占用这部分的空隙。

Apache 中的返回码如下表所示：

错误名称	含义	值
APR_ENOSTAT	APR 无法对一个文件执行 stat 操作	20001
APR_ENOPOOL	APR 没有提供内存池来执行内存分配操作	20002
APR_EBADDATE	APR 给出了一个无效的日期	20003

APR_EINVALSOCK	APR 给出了一个无效的 socket	20004
APR_ENOPROC	APR 没有给定一个进程的结构	20005
APR_ENOTIME	APR 没有给定一个时间结构	20006
APR_ENODIR	APR 没有给定一个目录结构	20007
APR_ENOLOCK	APR 没有给定一个互斥锁结构	20008
APR_ENOPOLL	APR 没有给定一个 Poll 结构	20009
APR_ENOSOCKET	APR 没有给定一个 socket	20010
APR_ENOTHREAD	APR 没有给定一个线程结构	20011
APR_ENOTHDKEY	APR 没有给定一个线程 Key 结构	20012
APR_ENOSHMAVAIL	APR 中没有更多的可用共享内存	20013
APR_EDSOOPEN	APR 中无法打开一个 DSO 对象	20014
APR_EGENERAL	APR 中的通常的错误	20015
APR_EBADIP	描述的 IP 地址错误	20016
APR_EBADMASK	描述的 IP 地址掩码错误	20017
APR_ESYMNOTFOUND	无法查找到请求 symbo	20018
APR_INCHILD	程序正在执行子进程	70001
APR_INPARENT	程序正在执行父进程	70002
APR_DETACH	线程从主线程中被分离出来	70003
APR_NOTDETACH	线程尚未从主线程中分离出来	70004
APR_CHILD_DONE	子进程已经执行完毕	70005
APR_CHILD_NOTDONE	子进程尚未执行完毕	70006
APR_TIMEUP	执行操作超时	70007
APR_INCOMPLETE	The operation was incomplete although some processing was performed and the results are partially valid	70008
APR_BADCH	Getopt 函数查找到一个不在选项字符串中的选项	70012
APR_BADARG	Getopt 发现一个选项缺少参数，而在选项字符串中该选项必须指定	70013
APR_EOF	APR 已经到达文件的末尾	70014
APR_NOTFOUND	APR 在 poll 结构中无法发现 socket	70015
APR_ANONYMOUS	APR 正在使用匿名的共享内存	70019
APR_FILEBASED	APR 正在使用文件名作为共享内存的 key	70020
APR_KEYBASED	APR 正在使用共享 key 作为共享内存的 key	70021
APR_EINIT		70022
APR_ENOTIMPL	在该平台上，该 APR 函数尚未实现	70023
APR_EMISMATCH	输入的两个密码不匹配	70024

APR_EABSOLUTE	给定的路径值是绝对路径	70019
APR_ERELATIVE	给定的路径是相对路径	70020
APR_EINCOMPLETE	给定的路径既不是相对路径也不是绝对路径	70021
APR_EABOVEROOT	给定的路径在跟路径上	70022
APR_EBUSY	给定的互斥锁正忙，已经被锁定	70025
APR_EPROC_UNKNOWN	该进程无法被 APR 所识别	70024

有一点必须明确的是返回码并不一定总是错误码。如果你的函数返回多个值，它们中的每一个都意味着执行成功，但是它们的值却不一样，或者你的函数仅仅返回成功或者失败两种情况，那么 APR 中通常仍然会返回一个 apr_status_t 值。

在第一种情况下，即如果执行成功具有多种状态，那么可以为每一个状态定义一个 APR 返回状态码，典型的例子就是 apr_proc_wait 函数，它用于等待子进程结束，它会返回三种情况：APR_CHILDDONE 表示子进程已经执行完 毕；APR_CHILDNOTDONE 表示子进程尚未执行完毕；错误码则意味着等待子进程失败。对于前两种返回不能称之为失败，它们都是成功返回，只是返 回状态不一样而已，为此 APR 中定义两个状态码表示返回状态，**记住不是错误码**。

对于第二种情况，即执行成功后仅有一种状态，那么如果执行成功，APR 中通常返回 APR_SUCCESS，而不是什么都不返回；如果执行失败，则定义新的 APR 状态码来描述这种失败。比如 apr_compare_users 函数，它返回 APR_SUCCESS 表示失败，同时定义 APR_EMISMATCH 和其余错误码表示失败。

根据上面的原则，你就会发现 APR 中的函数很少有返回类型为 void 或者 void*的。更多的都是返回 apr_status_t。

APR 中所有的错误码的定义在 apr_errno.h 中都可以找到。当 APR 函数中发生错误的时候，这些函数必须返回一个错误码。如果这些错误是在系统调用错误，那么 APR 将使用系统调用返回的错误码 errno 作为返回码原样返回，比如：

```
if(open(fname,oflags,0777)<0)
    return errno;
```

对于系统调用，除了直接返回系统错误码之外，另外一种策略就是使用新的 APR 错误码替代原始的系统错误码，比如：

```
if (CreateFile(fname, oflags, sharemod, NULL,
      createflags, attributes, 0) == INVALID_HANDLE_VALUE
return (GetLAstError() + APR_OS_START_SYSERR);
```

上面的两个例子在不同的平台上实现了相同的功能。显而易见，即使在两个平台上存在的潜在问题都是一样的，那么也会导致返回截然不同的错误码。不过对于这两种情况 APR 都是支持的。事实上 APR 的观点是，当一个错误发生的时候，程序通常是记录该错误并且继续往下执行，默认情况下它并不会去尝试解决发生的错误。不过这并不意味着 APR 根本不捕获错误并且提供解决方案。事实上，在后面的部分我们会看到 APR 中如何处理错误。

1.5.3.2 返回码信息映射

大部分情况下，状态码主要用于系统内部使用，因此它的含义隐晦，对于用户影响不是特别的大，但是错误码则不一 样。用户更多的是希望系统返回足够多的信息以便直到发生错误的原因，从而进行跟踪和调试。因此这种情况下，如果如果仅仅返回一个整数给用户，用户可能会莫 名其妙，一头雾水。最好的方法就是能够将该返回码所代表的实际的含义以字符串的形式返回出去

事实上大部分操作系统平台都提供了这种对应函数，比如 stderror。APR 中使用 apr_strerror 函数将上述的返回码转换为实际的字符串信息：

```
APR_DECLARE(char *) apr_strerror(apr_status_t statcode, char *buf,
                                 apr_size_t bufsize)
```

statcode 是需要转换的返回码，转换后的字符串保存在缓冲区 buf 中，函数所允许的字符串的长度由 bufsize 控制。

对于不同的返回码，APR 采取不同的转换策略：

■ 系统错误码(statcode< APR_OS_START_ERROR)

这类错误码 APR 中称之为”native error”，即这些错误码并不是 APR 定义的，而是系统返回的。比如 open 函数的返回码就应该算”native error”。对于这类错误码，APR 的处理就是直接使用 stderror_r 函数返回系统提示的消息，如果找不到该系统错误码，返回”APR does not understand this error code”信息。之所以不使用 stderror 是因为 stderror 不是线程安全的如果平台能够实现线程安全的 stderror 函数，比如 Solaris, OS/390，那么你也可以使用 stderror，否则不要这么做。

这类”native error”通常小于 20000，因此它们会使用 APR_OS_START_ERROR 之前的空隙。

■ APR 定义返回码(APR_OS_START_ERROR <= statcode <= APR_OS_START_USERERR)

这类返回码通常是 APR 定义并使用的，因此 APR 本身必须能够维持这些返回码和返回信息之间的对应关系。APR 中采取的是最原始的”switch/case”方法来对传入的返回码逐一进行判断：

```
static char *apr_error_string(apr_status_t statcode)
{
    switch (statcode) {
    case APR_ENOPOOL:
        return "A new pool could not be created.";
    case APR_EBADDATE:
        return "An invalid date has been provided";
    case APR_EINVALSOCK:
        return "An invalid socket was returned";
    case APR_ENOPROC:
        return "No process was provided and one was required.";
    ......
}
```

因此如果需要了解 APR_ENOPOOL 返回码的确切含义，只需要使用语句

printf(“%s”,apr_error_string(APR_ENOPOOL))即可。

■ APR 自定义返回码(APR_OS_START_USERERR <= statcode <= APR_OS_START_EAIERR)

■ EAI 错误码(APR_OS_START_EAIERR <= statcode <= APR_OS_START_SYSERR)

该范围内的错误码主要对应 getaddrinfo 函数返回的 EAI_*系列错误码。因此在对 statcode 进行调整后直接调用 gai_strerror()返回对应的信息即。如果操作系统不支持 gai_strerror 函数那么直接返回"APR does not understand this error code"。

■ 平台相关错误码(APR_OS_START_SYSERR <= statcode)

这类错误码总是与平台相关的，APR 返回码与实际的错误码之间保持下面的关系：

APR_Error = APR_OS_START_SYSERR + 系统错误码

由于这类错误码与平台相关，因此处理也是与平台相关的。

1)、对于 WINDOW 平台而言，所有的 Window 平台所特有的错误码及实际含义都保存在数组 gaErrorList 中：

```
static const struct {
    apr_status_t code;
    const char *msg;
} gaErrorList[] = {
    WSAEINTR,           "Interrupted system call",
    WSAEBADF,           "Bad file number",
    WSAEACCES,          "Permission denied",
    ……
}
```

因此处理过程就很简单，一旦获取了错误码比如 WSAEACCES 后通过查找 gaErrorList 数组就可以获取其实际含义。返回后的字符串使用 FormatMessage 格式化输出。

2)、对于 Unix 平台而言主要是由于 Linux 在调用诸如 gethostbyname 或者 gethostbyaddr 等函数时候会返回一些特有的错误码，比如 HOST_NOT_FOUND、NO_ADDRESS、NO_DATA、 NO_RECOVERY、TRY_AGAIN 等。Linux 专门提供了 hstrerror 函数获取这些错误码的实际含义。因此对这类错误码的处理如果系统 实现了 hstrerror，则调用 hstrerror 处理，否则使用最原始的 switch/case 进行处理：

```
switch(err) {
    case HOST_NOT_FOUND:
        msg = "Unknown host";
        break;
    case NO_ADDRESS:
        msg = "No address for host";
        break;
    case NO_DATA:
        msg = "No address for host";
        break;
    default:
        msg = "Unrecognized resolver error";
    }
```

3)、另一种需要考虑的就是 OS/2。由于 OS/2 不是我们的重点，此处不再描述。需要了解的可以参考源代码。

1.5.3.3 错误捕捉策略

对于 APR 自定义错误码，APR 可以直接通过返回值获取，而对于系统调用错误码，Apache 则必须使用系统函数来获取。对于所有的操作系统平台，Apache 提供了四个函数捕获系统相关错误码：

```
apr_get_os_error()
apr_set_os_error(e)
apr_get_netos_error()
apr_set_netos_error(e)
```

前两个分别用于或者和设置系统错误码，而后者则主要用于获取和设置网络错误码，之所以进行这种区分，是因为大部分操作系统平台上获取系统错误码和获取网络错误码的方法不同。在 Window 平台上获取错误码主要是调用函数 GetLastError 和 SetLastError，对于网络错误的获

取则是 WSAGetLastError 和 WSASetLastError。Unix 下则与之不同。下表列出了各个平台对应的处理：

	apr_get_os_error	apr_set_os_error(e)	apr_get_netos_error	apr_set_netos_error(e)
Window	GetLastError	SetLastError(e)	WSAGetLastError	WSASetLastError
Unix	(errno)	(errno = (e))	(errno)	(errno = (e))
Netware	(errno)	(errno = (e))	WSAGetLastError	WSASetLastError

Apache 中大部分的操作都是将返回码与指定的返回码进行比较，然后根据比较结果做出进一步的操作。不过 Apache 中对于错误码的比较并不是使用最简单的格式，比如“if(s==APR_EBUSY)”，取而代之的是，Apache 中使用一些简单的宏来执行比较任务，通常格式为：

APR_STATUS_IS_*XXX*

XXX 通常是错误码 APR_*XXX* 的 XXX 部分，比如 APR_STATUS_IS_EOF、APR_STATUS_IS_BADCH 等等。而这些宏的实现非常简单，一看就明白：

```
#define APR_STATUS_IS_BADCH(s)          ((s) == APR_BADCH)
```

APR_errno.h 中的大部分内容就是这种宏定义，每一个错误码都会对应这样一个宏定义。

另外系统中还定义了两个宏分别用于实现 APR 返回码与操作系统本地码的转换：

```
#define APR_FROM_OS_ERROR(e) (e == 0 ? APR_SUCCESS : e + APR_OS_START_SYSERR)
#define APR_TO_OS_ERROR(e) (e == 0 ? APR_SUCCESS : e - APR_OS_START_SYSERR)
```

前者用于将操作系统平台相关的返回码转换为 APR 定义的返回码，而后者则相反，用于将 APR 定义的返回码转换为操作系统平台相关码。

由于 APR 是可移植的，这样就可能遇到这样一个问题：不同平台错误码的不一致。如何处理呢？APR 给我们提供了 2 种策略：

a)对于所有的操作系统平台都返回相同的错误码

这种策略的缺点是从平台相关错误码转换为通用错误码比较耗费时间，而且大多数情况下，开发人员需要的仅仅是输出一个错误字符串而已。如果我们将所有的错误码转换为一个通用的公共的错误码子集，那么为了输出一个错误字符串信息，我们必须完成四个步骤：

```
make syscall that fails
    convert to common error code                step 1
    return common error code
        check for success
        call error output function              step 2
            convert back to system error        step 3
            output error string                 step 4
```

相比而言，这是一个比较耗时的步骤。通过使用使用平台相关错误码的话，那么整个步骤可以压缩为只有两步：

```
make syscall that fails
    return error code
        check for success
        call error output function              step 1
            output error string                 step 2
```

这种策略的第二个可能造成的问题就是错误码的损耗，这种问题源自各个平台错误码数目的不均衡性。比如 Windows 和 OS/2 操作系统定义了成百上千错误码，而 POSIX 才定义了 50 错误码，

如果都转换为规范统一的错误码，势必会有Window平台错误码含义的丢失（错误码多的比如Window平台），或者可能得不到拥有真正含义的错误码(错误码少的，比如POSIX)。

b）返回平台相关错误码，如果需要将它转换为APR通用错误码

第二种策略中，程序的执行路线往往要根据函数返回错误码来定，这么做的缺点就是把这些工作推给了程序员。执行流程如：

```
make syscall that fails
        convert to common error code
        return common error code
            decide execution based on common error code
```

如果考虑到将平台相关的错误码转换为通用的错误码，那么上面的代码段可以修改为如下：

```
make syscall that fails
        return error code
            convert to common error code (using ap_canonical_error)
            decide execution based on common error code
```

1.5.4 宏处理

Apache目前能够支持五个大种类的运行平台，包括Window、OS/2、BeOS、 Unix、NetWare，而Window又可以细分为Window98、Window2000等等。Unix则又可以进一步细分，包括Linux，ScoUNIX，DARWIN等等。为了能够让Apache运行在如此之多的操作系统平台上，Apache在源代码中增加了许多的编译开关。

举个例子，比如utime.h头文件的包含问题。因为文件在Linux(gcc)下面和Windows(cl)下所处的C Library目录不同。包含的处理办法就不一样。可能需要这样写才能完全正确的包含。

```
#if HAVE_UTIME_H                    //---- 如果有utime.h 文件
#    ifdef WIN32                    //-----如果是win32环境
#        include <sys/utime.h>     //-----包含sys/utime.h
#    endif
#    ifdef LINUX                    //---- 如果是Linux环境
#        include <utime.h>         //---- 包含utime.h
#    endif
#else                               //--- 如果没有utime.h定义出需要的结构
struct utimbuf
{
long actime;
long modtime;
};
#endif
```

Apache处理与之类似。根据编译环境的不同来编译不同的代码。这样的#define的区隔，主要就是为了区隔不同平台的不同细微区别。有的区别也许是某些常量没有定义，有些区别是某些函数不存在。

Apache中使用的很多的编译开关是各个操作系统或者各个编译器已经确定的，通过这些预定义就可以很容易的区分使用的操作系统平台，比如__osf__和__alpha是DEC的OSF/1 1.3操作系统中的定义，因此如果某个函数只能运行于OSF/1 1.3中，则可以使用下面的编译处理代码：

```
#ifdefine __osf__||__aplpa
        //调用函数
```

#endif
下面的表格中给出了目前大部分的操作系统以及编译器的编译开关：

机器硬件	生产商	操作系统	编译器	能够识别的编译其开关变量
AMIGA	Commodore	AMIGA-OS (AMIGADOS)	GNU	amiga or AMIGA, __GNUC__, maybe MC68000 or AMIGA3000
any	any	UNIX	GNU	unix, __GNUC__, ...
any	any	UNIX	CC	unix, ...
Amiga 3000	Commodore	Amiga UNIX 2.1 SVR4.0	GNU	unix,__unix__,AMIX,__AMIX__,__svr4__,m68k, __m68k__, __motorola__, __GNUC__
SUN-3	Sun	SUN-OS3 (UNIX BSD 4.2)	GNU	sun, unix, mc68020, __GNUC__
SUN-3	Sun	SUN-OS4 (UNIX SUNOS 4.1)	GNU	sun, unix, mc68020, __GNUC__
SUN-386	Sun	SUN-OS4 (UNIX SUNOS 4.0)	GNU	sun, unix, sun386, i386, __GNUC__
SUN-386	Sun	SUN-OS4 (UNIX SUNOS 4.0)	CC	sun, unix, sun386, i386
SUN-4	Sun	SUN-OS4 (UNIX SUNOS 4.1)	GNU	sun, unix, sparc, __GNUC__
SUN-4	Sun	SUN-OS4 (UNIX SUNOS 4.1)	CC	sun, unix, sparc
SUN-4	Sun	SUN-OS5 (UNIX Solaris)	GCC	sun, unix, sparc, __GNUC__
UltraSparc	Sun	Solaris 7 (UNIX SUNOS 5.7)	CC	sun, unix, __sparc, __sparcv9
UltraSparc	Sun	Solaris 7 (UNIX SUNOS 5.7)	GCC	sun, unix, __sparc, __arch64__, __GNUC__
IBM-PC/386	any	SUN-OS5 (UNIX Solaris)	GCC	sun, unix, __svr4__, i386, __GNUC__
HP9000-300	Hewlett-Packard	NetBSD 0.9 (UNIX BSD 4.3)	GNU	unix, __NetBSD__, mc68000, __GNUC__
HP9000-300	Hewlett-Packard	HP-UX 8.0 (UNIX SYS V)	GNU	[__]hpux, [__]unix, [__]hp9000s300, mc68000, __GNUC__
HP9000-800	Hewlett-Packard	HP-UX 8.0 (UNIX SYS V)	GNU	[__]hpux, [__]unix, [__]hp9000s800
IRIS	Silicon Graphics	IRIX (UNIX SYS V 3.2)	GNU	unix, SVR3, mips, sgi, __GNUC__
IRIS	Silicon Graphics	IRIX (UNIX SYS V)	cc -ansi	[__]unix, [__]SVR3, [__]mips, [__]sgi

IRIS Silicon Graphics IRIX 5 (UNIX SYS V 4) GNU [__]unix, [__]SYSTYPE_SVR4, [__]mips, [__]host_mips, [__]MIPSEB, [__]sgi, __DSO__, [__]_MODERN_C, __GNUC__

DECstation 5000 RISC/OS (Ultrix V4.2A) GNU unix, [__]mips, [__]ultrix

DG-UX 88k Data General DG/UX GNU unix, m88000, DGUX

DEC Alpha DEC OSF/1 1.3 cc [unix,] __unix__, __osf__, __alpha

DEC Alpha DEC OSF/1 1.3 GNU unix, __unix__, __osf__, __alpha, __alpha__, _LONGLONG

Apple MacII Apple A/UX (UNIX SYS V 2) GNU [__]unix, [__]AUX, [__]macII, [__]m68k, mc68020, mc68881, __GNUC__

NeXT NeXT NeXTstep 3.1 (UNIX) cc NeXT, m68k; NEXTAPP for NeXTstep Application

PowerPC Apple Mach 3.0 + MkLinux GNU unix, __powerpc__, __PPC__, _ARCH_PPC, _CALL_SYSV, __ELF__, __linux__

PowerPC Apple Mach + Rhapsody cc __MACH__, __APPLE__, __ppc[__], __GNUC__, __APPLE_CC__

PowerPC Apple Mach + MacOS X cc __MACH__, __APPLE__, __ppc__, __GNUC__, __APPLE_CC__

Sequent Sequent PTX 3.2.0 V2.1.0 i386 (SYS V) GNU unix, i386, _SEQUENT_, __GNUC__

Sequent Sequent PTX V4.1.3 GNU unix, i386, _SEQUENT_, __svr4__, __GNUC__

Convex C2 Convex ConvexOS 10.1 GNU __convex__, __GNUC__

IBM RS/6000 IBM AIX 3.2 GNU _AIX, _AIX32, _IBMR2, __CHAR_UNSIGNED__, __GNUC__

IBM-PC/386 any LINUX (free UNIX) GNU unix, linux, i386, __GNUC__

IBM-PC/386 any LINUX (free UNIX) Intel 5.0 __unix__, __linux__, __INTEL_COMPILER, __ICC, __USLC__

IBM-PC/386 any 386BSD 0.1 (UNIX BSD 4.2) GNU unix, __386BSD__, i386, __GNUC__

IBM-PC/386 any NetBSD 0.9 (UNIX BSD 4.3) GNU unix, __NetBSD__, i386, __GNUC__

IBM-PC/386 any FreeBSD 4.0 (UNIX BSD 4.4) GNU unix, __FreeBSD__, i386, __GNUC__

IBM-PC/386 any EMX 0.9c (UNIXlike on OS/2) GNU [unix,] i386, __GNUC__, __EMX__

IBM-PC/386 any Cygwin32 on WinNT/Win95 GNU _WIN32, __WINNT__, __CYGWIN32__, __POSIX__, _X86_, i386, __GNUC__

IBM-PC/386	any	Mingw32 on WinNT/Win95	GNU	_WIN32, __WINNT__, __MINGW32__, _X86_, i386, __GNUC__
IBM-PC/386	any	WinNT/Win95	MSVC4.0,5.0	_WIN32, _M_IX86, _MSC_VER
IBM-PC/386	any	WinNT/Win95	Borland 5.0	__WIN32__, _M_IX86, __TURBOC__, __BORLANDC__
IBM-PC/386	any	WinNT/Win95 and Cygwin32	GNU	_WIN32, __WINNT__, __CYGWIN32__, __POSIX__, __i386__, _X86_, __GNUC__
IBM-PC/586	any	BeOS 5	GNU	__BEOS_, __INTEL__, __i386__, _X86_, __GNUC__
IBM-PC/586	any	HP NUE/ski, Linux	GNU	unix, linux, __ia64[__], __GNUC__, __LP64__
RM400	Siemens-Nixdorf	SINIX-N 5.42	c89	unix, mips, MIPSEB, host_mips, sinix, SNI, _XPG_IV
Acorn	Risc PC	RISC OS 3.x	GNU	[__]arm, [__]riscos, __GNUC__
Acorn	Risc PC	RISC OS 3.x	Norcroft	[__]arm, [__]riscos
APPLE IIGS	Apple	??	??	

当然，列出上面的大部分的编译开关，并不是说Apache都支持它们，事实上Apache仅仅支持一部分，如果仅仅对某个操作系统或者某个机器有兴趣，则可以挑选对应的宏定义中的代码。上述的编译器开发都是某个平台相关的，事实上只要运行于该平台，该开关自然就成立，不需要APR本 身重新定义。另外有一些特性开关则是必须由APR自行定义。这些特性通常是操作系统之间很小的区别，比如同样是Unix系统，可能有的支持共享内存，有的 不支持，为此在使用共享内存之前必须能够判断当前的平台是否支持。这些特性宏的定义可以从apr.h.in模板中定义：

#define APR_HAVE_ARPA_INET_H	@arpa_ineth@	#define APR_USE_FLOCK_SERIALIZE
#define APR_HAVE_CONIO_H	@conioh@	#define APR_USE_SYSVSEM_SERIALI
#define APR_HAVE_CRYPT_H	@crypth@	#define APR_USE_POSIXSEM_SERIAL
#define APR_HAVE_CTYPE_H	@ctypeh@	#define APR_USE_FCNTL_SERIALIZE
#define APR_HAVE_DIRENT_H	@direnth@	#defineAPR_USE_PROC_PTHREAD_SER

```
#define APR_HAVE_ERRNO_H         @errnoh@
#define APR_HAVE_FCNTL_H         @fcntlh@
#define APR_HAVE_IO_H            @ioh@
#define APR_HAVE_LIMITS_H        @limitsh@
#define APR_HAVE_NETDB_H         @netdbh@
#define APR_HAVE_NETINET_IN_H    @netinet_inh@
#define APR_HAVE_NETINET_SCTP_H @netinet_sctph@
#define APR_HAVE_NETINET_SCTP_UIO_H @netinet_sctp_uioh@
#define APR_HAVE_NETINET_TCP_H   @netinet_tcph@
#define APR_HAVE_PTHREAD_H       @pthreadh@
#define APR_HAVE_SEMAPHORE_H     @semaphoreh@
#define APR_HAVE_SIGNAL_H        @signalh@
#define APR_HAVE_STDARG_H        @stdargh@
#define APR_HAVE_STDINT_H        @stdint@
#define APR_HAVE_STDIO_H         @stdioh@
#define APR_HAVE_STDLIB_H        @stdlibh@
#define APR_HAVE_STRING_H        @stringh@
#define APR_HAVE_STRINGS_H       @stringsh@
#define APR_HAVE_SYS_IOCTL_H     @sys_ioctlh@
#define APR_HAVE_SYS_SENDFILE_H @sys_sendfileh@
#define APR_HAVE_SYS_SIGNAL_H    @sys_signalh@
#define APR_HAVE_SYS_SOCKET_H    @sys_socketh@
```

```
#define APR_USE_PTHREAD_SERIALI
#define APR_HAS_FLOCK_SERIALIZE
#define APR_HAS_SYSVSEM_SERIALI
#define APR_HAS_POSIXSEM_SERIAL
#define APR_HAS_FCNTL_SERIALIZE
#define APR_HAS_PROC_PTHREAD_SE
#define APR_HAS_RWLOCK_SERIALIZ
#define APR_PROCESS_LOCK_IS_GLO
#define APR_HAVE_CORKABLE_TCP
#define APR_HAVE_GETRLIMIT
#define APR_HAVE_IN_ADDR
#define APR_HAVE_INET_ADDR
#define APR_HAVE_INET_NETWORK
#define APR_HAVE_IPV6
#define APR_HAVE_MEMMOVE
#define APR_HAVE_SETRLIMIT
#define APR_HAVE_SIGACTION
#define APR_HAVE_SIGSUSPEND
#define APR_HAVE_SIGWAIT
#define APR_HAVE_STRCASECMP
#define APR_HAVE_STRDUP
```

```
#define APR_HAVE_SYS_SOCKIO_H    @sys_sockioh@
#define APR_HAVE_SYS_SYSLIMITS_H @sys_syslimitsh@
#define APR_HAVE_SYS_TIME_H      @sys_timeh@
#define APR_HAVE_SYS_TYPES_H     @sys_typesh@
#define APR_HAVE_SYS_UIO_H       @sys_uioh@
#define APR_HAVE_SYS_UN_H        @sys_unh@
#define APR_HAVE_SYS_WAIT_H      @sys_waith@
#define APR_HAVE_TIME_H          @timeh@
#define APR_HAVE_UNISTD_H        @unistdh@
#define APR_HAVE_SHMEM_MMAP_TMP     @havemmaptmp@
#define APR_HAVE_SHMEM_MMAP_SHM     @havemmapshm@
#define APR_HAVE_SHMEM_MMAP_ZERO    @havemmapzero@
#define APR_HAVE_SHMEM_SHMGET_ANON @haveshmgetanon@
#define APR_HAVE_SHMEM_SHMGET       @haveshmget@
#define APR_HAVE_SHMEM_MMAP_ANON    @havemmapanon@
#define APR_HAVE_SHMEM_BEOS         @havebeosarea@

#define APR_USE_SHMEM_MMAP_TMP     @usemmaptmp@
#define APR_USE_SHMEM_MMAP_SHM     @usemmapshm@
#define APR_USE_SHMEM_MMAP_ZERO    @usemmapzero@
#define APR_USE_SHMEM_SHMGET_ANON @useshmgetanon@
#define APR_USE_SHMEM_SHMGET       @useshmget@
```

```
#define APR_HAVE_STRICMP
#define APR_HAVE_STRNCASECMP
#define APR_HAVE_STRNICMP
#define APR_HAVE_STRSTR
#define APR_HAVE_MEMCHR
#define APR_HAVE_STRUCT_RLIMIT
#define APR_HAVE_UNION_SEMUN
#define APR_HAVE_SCTP

/* APR Feature Macros */
#define APR_HAS_SHARED_MEMORY
#define APR_HAS_THREADS
#define APR_HAS_SENDFILE
#define APR_HAS_MMAP
#define APR_HAS_FORK
#define APR_HAS_RANDOM
#define APR_HAS_OTHER_CHILD
#define APR_HAS_DSO
#define APR_HAS_SO_ACCEPTFILTER
#define APR_HAS_UNICODE_FS
#define APR_HAS_PROC_INVOKED
#define APR_HAS_USER
```

#define APR_USE_SHMEM_MMAP_ANON @usemmapanon@ #define APR_USE_SHMEM_BEOS @usebeosarea@	#define APR_HAS_LARGE_FILES #define APR_HAS_XTHREAD_FILES #define APR_HAS_OS_UUID

在使用 configure 进行配置的时候，apr.h.in 模板作为输入最终生成 apr.h 文件。不过 apr.h.in 中的@xx@将被 0 或者 1 所取代：如果该平台支持某个特定，相应的宏将定义为 1，否则定义为 0。

Apache 源代码分析——关于模块结构的几个重要概念

本文分析了 Apache 中关于模块的几个重要的概念

//

关于模块的几个重要的全局变量

理解 Apache 模块的概念之前我们首先必须弄清楚 apache 中关于模块的几个重要概念和数据结构。

1. DSO(Dynamic Shared Object，动态共享对象)

Apache 服务器的体系结构的最大的特点就是高度模块化，这一点到 2.0 版本的时候几乎发挥到了极致。除了少数的几个核心文件外，Apache 中大部分功能都被模块 化。模块化的最大的优势就是用户可以根据自己的实际需要进行裁减和增加。模块的存在有两种方式，一种就是在编译 Apache 的时候跟核心文件一起编译，这 时候的模块我们称之为静态链接编译模块；另一种存在方式就是独立于 Apache 的核心文件，这种文件何时编译与 Apache 的核心文件无关，核心文件也不 关心其存在。核心文件只有在需要的时候才去找它并将它装入自己的执行空间。这种方式称之为动态状态模块。

DSO 的产生当然离不开操作系统的支 持。目前的不管 UNIX 还是 Linux 大多提供了对动态共享对象或者动态链接库进行加载何卸载的机制。加载的方法通常有两种：其一是在可执行文件启动时候 由系统程序 ld.so 自动加载；其二是在执行程序中手工地通过 Unix 提供的动态链接库加载接口进行加载。

按照第一种方法，动态链接库比如 libfoo.so 或者 libfoo.so.1.2 通常被存储在目录/usr/lib 中。使用 libfoo.so 库的程序只需要在编译程序的时候加上编译选项-lfoo 就可以建立到动态链接库的链接。通常

而 在第二种方法种，动态链接库可以使用任何的文件名(规范的文件名称为 foo.so)，而且也不一定要存储在/usr/lib 目录中，其可以存储于任何目录 中，也不会自动建立指向其所用的可执行程序的链接，而是由可执行文件在运行的时候自己调用类似 dlopen 之类的接口将动态链接库加载到自己的地址空间， 同时也不会为可执行程序解析动态链接库中的符号，包括函数名称、变量名称等等，这些工作必须由可执行程序自行调用 dlsym 之类的函数进行解析。

。 。 。 。 。

Apache 2.0 提供了下面的几个命令用于生成 DSO 共享链接库，简要说明如下：

1. 编译并安装已发布的 Apache 模块，比如编译 mod_foo.c 为 mod_foo.so 的 DSO 模块：

```
$ ./configure --prefix=/path/to/install --enable-foo=shared
$ make install
```

2. 编译并安装第三方 Apache 模块，比如编译 mod_foo.c 为 mod_foo.so 的 DSO 模块：

```
$ ./configure --add-module=module_type:/path/to/3rdparty/mod_foo.c --enable-foo=shared
$ make install
```

3. 配置Apache以共享后安装的模块：

```
$ ./configure --enable-so
$ make install
```

4. 用apxs在Apache源代码树以外编译并安装第三方Apache模块，比如编译mod_foo.c为mod_foo.so的DSO模块：

```
$ cd /path/to/3rdparty
$ apxs -c mod_foo.c
$ apxs -i -a -n foo mod_foo.la
```

共享模块编译完毕以后，都必须在httpd.conf中用LoadModule指令使Apache激活该模块。

2. Linux中DSO函数操作

```
void* dlopen(const char *pathname, int mode);
```

该函数函数来加载动态库，其中pathname是需要加载的库的路径名称，mode则是加载的方式，可以是三个值：RTLD_LAZY用来表示认为未定义的符号是来自动态链接库的代码；RTLD_NOW则表示要在dlopen返回前确定所有未定义的符号，如果不能完成则失败；而RTLD_GLOBAL则意味着动态链接库中定义的外部符号将可以由随后加载的库所使用。如果函数执行成功，其将返回动态链接库的一个句柄。

一旦对动态库进行了加载，我们则可以通过dlsym函数获取库中的函数调用以及各种定义的符号等等。其函数原型如下：

```
void* dlsym(void* handle, char* symbol);
```

其中，handle是加载的动态链接库的句柄，其通常是dlopen函数的操作结果；symbol则是需要得到的动态链接库中的符号名称。如果找不到symbol，函数将返回NULL。

在所有的操作结束后，Linux可以通过dlclose将dlopen先前打开的共享对象从当前进程断开，不过只有动态链接库的使用记数为0的时候该共享对象才会真真被卸载。dlclose的函数原型如下：

```
int dlclose(void* handle);
```

一旦使用dlclose关闭了对象，dlsym就再也不能使用它的符号了。

下面的代码简单的演示了dlopen、dlsym、dlclose的三步曲的使用。

```
void* handle, *handle2;
handle = dlopen("libdisplay.so", RTLD_LAZY);
if (handle != NULL)
{
handle2 = dlsym(handle, "draw");
if (handle2 != NULL){
…… /* use the function */
}
/* When finished, unload the shared library */
dlclose(handle);
}
```

3. Apache中DSO处理模块

Apache中对动态链接库的处理是通过模块mod_so来完成的，该模块与其余的所有模块相比，没有什么特殊之处，因此从某种角度而言，其就是一个普通的模块。不过由于其作用是用来装载其余的模块，因此该模块不能被动态加载，它只能被静态编译进Apache的核心。这意味着Apache一启动，该模块就必须立即起作用。mod_so模块是Apache中除了core模块之外唯一不能作为DSO模块而存在的模块。想想，如果mod_so模块也要被动态加载，那

么 谁 来 加 载 它 呢 ？

4 . Apache 中 关 于 模 块 的 几 个 全 局 变 量

Apache 中定义了一些与模块相关的全局变量，它们的作用相似，但存在差异。

■ap_preloaded_modules[]

该 数组称之为预装载模块数组，其中定义了所有的 Apache 中默认的静态编译的模块。尽管这些模块已经被编译进 Apache 中，但是其不一定能够发挥作用， 模块能否发挥作用，取决于该模块是否被“激活”。只有激活的模块才能起作用。在 Apache1.3 中，可以通过指令 AddModule 来 激 活 某 个 模 块 。

```
module *ap_preloaded_modules[] = {
        &core_module,
    &mpm_netware_module,
        &http_module,
        &so_module,
    &mime_module,
        ……
        NULL
};
```

■ap_prelinked_modules[]

该数组称之为预链接模块数组，该数组定义了所有的 Apache 中默认的在 Apache 启动后就处于“激活”状态的模块，其定义简化如下：

```
module *ap_prelinked_modules[] = {
        &core_module,
    &mpm_netware_module,
        &http_module,
        &so_module,
    &mime_module,
        ……
        NULL
};
```

该数组中定义的所有的模块在 Apache 启动后由于处于“激活”状态，因此可以立即相应核心 的 请 求

这 两个全局数组是从以前的 Apache 版本延续而来的。正如前面所说，在 Apache1.3 版本中，这两个数组未必完全相同。不过在 Apache2.0 中， 两个数组已经没有差异了，这意味着一个模块一旦被静态编译，其就立即处于激活状态，这也是 Apache2.0 中删除了 AddModule 指 令 的 原 因 。

■ap_loaded_modules 数 组

尽 管 Apache 对于所有的静态编译的模块都是立即激活它，这并意味着 Apache 对所有的模块都是这样。如果你某个模块不是默认的，而是第三方模块的话， 那么 Apache 即使将其装入也不一定就立即激活它。ap_loaded_modules 数组用来保存所有的已经被装入的模块，包括默认的和第三方的；包 括 激 活 的 和 非 激 活 的 。

■ap_top_modules 链 表

与 ap_loaded_modules 数组对应，ap_top_modules 链表用来保存 Apache 中所有的被激活的模块，包括默认的激活模块和激活的第三方模块。

Apache 源代码分析——模块的加载

本文分析了 Apache 中关于模块的加载过程

阅读本文之前，请先阅读 Apache 源代码分析——关于模块结构的几个重要概念一文

//
在了解了上面的四个重要的概念后，我们现在来看看 Apache 中是如何进行模块加载的。
Apache 中对模块的装载处理的相关函数主要集中于 mod_so.c 中，其本身也是一个子模块，该子模块用来动态状态其余的模块。在 mod_so.c 文件的开始位置有很长的一段文件描述，将该描述翻译成中文如下：
该 模块(指 Mod_so)主要用来在运行的时候动态装入 Apache 模块，这意味着对服务器可以进行功能扩展而不需要重新对源代码进行编译，甚至根本不需要 停止服务器。我们所需要做的仅仅是给服务器发送信号 HUP 或者 AP_SIG_GRACEFUL 通知服务器重新载入模块。
当然，在动态加载之前，你首先必须将你的模块编译成为动态链接库，然后更新你所指定的配置文件，通常情况下是 httpd.conf。这样 Apache 核心就可以在启动的时候调用你的模块了。
将模块编译成为共享受库的最简单的方法就是在配置中使用 ShareModule 命令，而不是使用 AddModule 命令，而且你必须将文件的扩展名称从‘.o’改变为‘.so’。比如如果我们想将 status 模块变为共享库，在配置文件中我们只需要将
AddModule modules/standard/mod_status.o
更改为
SharedModule modules/standard/mod_status.so
一旦更改完毕，运行配置文件同时进行编译。现在 Apache 的 httpd 的二进制文件中并没有包含 mod_status 模块，。。。。。
为了使用共享模块，将.so 文件拷贝到适当的目录中。你可能需要在服务器根目录下创建一个名称为“modules”的目录，比如
“/usr/local/httpd/modules”。
下面的事情就是编辑你的 conf/httpd.conf 文件，同时增加一行
“LoadModule”命令。比如：
LoadModule status_module modules/mod_status.so
该命令的第一个参数是模块的名称，名称可以在 module_source 的最后找到。第二个选项是模块所处的路径，这个路径是相对于服务器路径而言。
如果服务器还在运行的时候，你就编辑 LoadModule 命令，那么你可以通过发送信号 HUP 或者 AP_SIG_GRACEFUL 给服务器，一旦接受到该信号，Apache 将重新装载模块，而不需要重新启动服务器。

下面我们将来看看 Apache 是如何动态装载模块的。在分析每个模块之前，我们首先需要分析的就是模块的数据结构以及该模块能够处理的命令表。对于

mod_so 模块也不例外。mod_so 模块的结构和其中的命令表格如下所示：

```
static const command_rec so_cmds[] = {
    AP_INIT_TAKE2("LoadModule", load_module, NULL, RSRC_CONF |
EXEC_ON_READ,
      "a module name and the name of a shared object file to load it
from"),
    AP_INIT_ITERATE("LoadFile", load_file, NULL, RSRC_CONF  |
EXEC_ON_READ,
      "shared object file or library to load into the server at
runtime"),
    { NULL }
};
module AP_MODULE_DECLARE_DATA so_module = {
   STANDARD20_MODULE_STUFF,
   NULL,      /* create per-dir config */
   NULL,      /* merge per-dir config */
   so_sconf_create, /* server config */
   NULL,      /* merge server config */
   so_cmds,     /* command apr_table_t */
   NULL    /* register hooks */
};
```

模 块中能够处理的所有命令都保存在 so_cmds 中，从命令表中可以看出，mod_so 模块可以处理的命令只有“LoadModule”和 “LoadFile”，相应的处理函数分别为 load_module 和 loadfile。下面我们首先来看 load_module 函数的实现，该函数也是 模块装载处理的入口。

```
static const char *load_module(cmd_parms *cmd, void *dummy,
                               const char *modname, const char
*filename)
```

该函数用来将共享对象载入到服务器的地址空间中。其中，cmd 和 dummy 是所有的命令处理程序都必须具有的，其用于 Apache 核心给模块传递相应的信息。modname 是需要状态的模块的路径名称。

函数首先要做的事情就是在现有的所有的模块中查找是否已经存在需要载入的模块，如果该模块已经载入，则什么都不处理，否则则对其进行装载。遍历的模块包括动态装载的和静态编译两种。

对于动态载入模块，函数首先得到模块 so_module 中的模块配置信息，这个通过宏 ap_get_module_config 来实现。

在 server_rec 结构中，成员 module_config 是一个一维向量结构，专门用来存储各个模块针对本服务器的配置信息，向量结构中的每一个元素对应存储一个模块针对本服务器的所有配置信息，结构如下所示：

从 上面的结构图中可以看出，如果要获取第 i 个即索引为 i 的模块针对本服务器的配置信息的话，可以通过下面的表示式来获 取：module_config[i]，事实上模块的索引由模块的 module_index 决定，因此式子展开 为：module_config[(m)->module_index]。这实际上真是宏 ap_get_module_config 展开后的结果， 在 Apache 中，该宏定义为

```
#define ap_get_module_config(v,m)
    (((void **)(v))[(m)->module_index])
```
根 据上面的描述，我们不难理解该宏的含义。不过通过该宏获取的信息各个模块是千差万别，因此返回的类型只能是void**，因此如果需要使用最好进行转换。 就so_module模块而言，其每个元素都是so_server_conf类型的。该类型非常简单，其内部就是一个简单的数组：
```
typedef struct so_server_conf {
    apr_array_header_t *loaded_modules;
} so_server_conf;
```
模 块对应的配置信息实际上都保存在数组loaded_modules中。数组中的每个元素是moduleinfo结构，因此函数需要做的实质上就是遍历查找loaded_modules数组，判断其中是否存在给定的模块，比较只需要通过模块名称进行。下面的代码完成的就是查找动态加载模块：
```
sconf = (so_server_conf *)ap_get_module_config(cmd->server->module_config,
                                               &so_module);
modie = (moduleinfo *)sconf->loaded_modules->elts;
for (i = 0; i < sconf->loaded_modules->nelts; i++) {
modi = &modie[i];
if (modi->name != NULL && strcmp(modi->name, modname) == 0) {
ap_log_perror(APLOG_MARK, APLOG_WARNING, 0,
                            cmd->pool, "module %s is already loaded, skipping",
                            modname);
return NULL;
}
    }
```
对 动态加载模块检查完毕后，Apache将检查静态链接模块数组ap_preloaded_modules。在ap_preloaded_modules数 组中查找指定模块相对简单。对于每一个模块，Apache必须保证其文件名是以“mod_”开始的，比如mod_so.c、mod_alias.c等等， 如果命名格式不是这样，函数将认为模块不是合法的模块。函数的名称最终通过宏STANDARD20_MODULE_STUFF以__FILE__反映到 module结构的name属性中，因此函数只需要判断name是否合法就可以了。如果需要载入的模块没有被载入，则函数首先在动态载入模块 数组sconf->loaded_modules中压入一个新的module元素，同时将该结构的名称置为modname，既而函数调用 apr_dso_load将文件载入到Apache的地址空间中，同时调用apr_dso_sym获取动态链接库中的module结构，返回的结构保存在modsym中。一旦得到modsym，我们就相当于得到了该模块的module结构了，用modp表示，同时该模块内的动态加载句柄 dynamic_load_handle设置为dlopen取得的句柄。

在真正的使用即激活加载模块之前，Apache必须确保加载的模块确 实是Apache模块，为此Apache在模块结构中设定了magic字段，通过检查magic字段，apache确定加载的模块是否是apache模 块。对于Apache2.0而言，该值为“AP2.0”。另外apache还可以通过结构中的version来判断模块的兼容性。如

果加载的是合法的 2.0 模块，函数将立即调用 ap_add_loaded_module 将模块激活，所谓的激活无非就是将模块放入 ap_top_modules 链表中。
此外 apache 含需要在配置内存池 pconf 中注册 cleanup 处理函数。这样当我们重新启动或者关闭服务器的时候，cleanup 函数将自动调用将共享模块卸载；
最后我们需要做的就是为模块运行配置过程。

//
APR_DECLARE(apr_status_t) apr_dso_load(apr_dso_handle_t **res_handle,

const char *path, apr_pool_t *ctx);
该函数用来载入 DSO 动态共享库。res_handle_Location 用来存储新的 DSO 的处理句柄；path 则是 DSO 库的路径位置；
apr_dso_load 的实现分为七种平台：AIX、beos、netware、OS2、OS390、Unix 以及 Win32。我们首先来看看 Linux 平台下的实现。为此我们首先了解一下 Linux 下对动态链接库是如何的操作。
dso.c 中的大部分函数都是对 dl_xx 实现了简单的封装而已。
现 在我们再来看 apr_dso_load 的实现。函数首先调用 dlopen 对模块进行加载，返回加载后的句柄 os_handle，同时将句柄保存在 apr_dso_handle_t 类型变量 res_handle 的 handle 中。res_handle 所需要的所有资源来自内存池 ctx。最后调用 apr_pool_cleanup_register 函数注册内存池 ctx 的清除函数 dso_cleanup。

apr_dso_unload 函数则更简单，其只是调用了 apr_pool_cleanup_run 函数清除分配的内存池，同时调用 dso_cleanup 函数进行模块卸载。
//
AP_DECLARE(void) ap_add_loaded_module(module *mod, apr_pool_t *p)
该函数功能非常的简单，其实现了模块的动态加载和激活。所谓加载就是将模块假如 ap_loaded_modules 数组中；所谓的激活就是调用 ap_add_module 将模块放入 ap_top_modules 链表中。
//
AP_DECLARE(void) ap_add_module(module *m, apr_pool_t *p)
该函数用于在服务器中激活指定的模块 m。通常其对应于 httpd.conf 文件中的 AddModule 命令。
在 将模块加入服务器之前，Apache 必须判断模块的版本是否与当前 Apache 服务器的版本相同。Apache2.0 的所有模块的版本号统一为 MODULE_MAGIC_NUMBER_MAJOR；min_version 则为 MODULE_MAGIC_NUMBER_MINOR。不过 apache 仅仅核对主版本号码，次版本号不同无所谓。
在 Apache 中，主版本号的变更意味着模块的兼容性出现了问题，包括模块数据结构的变化等等，次版本号则意味着模块的小范围的调整，兼容性没有出现问题。
所 有的模块最终都被加入到全局模块列表 ap_top_module 中，所有新增加的模

块都插入 ap_top_module 链表的起始位置。一旦插入完 成，apache 将对插入的模块进行索引标记。标记的方法很简单，无非就是将当前的总模块数作为标记索引；同时累加总模块数和动态载入的模块数，有一点需 要注意的事，累加后的总模块数不能超过动态载入模块的最大数 DYNAMIC_MODULE_LIMIT。
另外还有一个小小的实现细节必须考虑 到。正如我们前面曾经说过，module 结构中的 name 值最终实际上填充的是__FILE__，不过不同的 C 编译器对__FILE__支持不一样。有的 C 编译器其值仅仅是文件名称，而有的 C 编译器其值则是完整的文件路径名称。而我们需要的仅仅是个文件名而已，因此对于第二种情况，我们需要将名称提取出 来，在 DOS 和 Unix 下实现代码如下：

```
if (ap_strrchr_c(m->name, '/'))
m->name = 1 + ap_strrchr_c(m->name, '/');
if (ap_strrchr_c(m->name, ''))
m->name = 1 + ap_strrchr_c(m->name, '');
```

至此函数需要做的最后一件事情就是调用 ap_register_hook 注册该模块的所有钩子。ap_register_hook 的内部无非还是调用的模块本身的钩子注册函数句柄 register_hooks。

```
////////////////////////////////////////////////////////////////////////////////
```

AP_DECLARE(void) ap_remove_module(module *m)
该函数的功能与 ap_add_module 对应，其用于将指定的模块修改为非激活状态。在函数内部，apache 遍历整个 ap_top_modules 链表查找模块 m。查找分为三种情况：
(1)要查找的模块位于 ap_top_modules 的头部，此时仅仅需要修改 ap_top_modules 指向下一个结点。
(2)要查找的模块不在链表头部，函数将顺着 next 往后遍历，一旦找到模块 m，则将指针定位于其前面的一个结点，修改其 next 指向模块 m 后面的结点，即 modp->next = modp->next->next。
(3)如果查找到链表的末尾都没有找到，则意味着该模块不存在。
一旦将 module 被移除，就修改 total_modules 和 dynamic_modules 的值。

```
////////////////////////////////////////////////////////////////////////////////
```

AP_DECLARE(void) ap_setup_prelinked_modules(process_rec *process)
该函数将 Apache 中的缺省模块加载并激活。该函数也是 Apache 中所有的模块处理的入口函数，最早出现在 main.c 中。
函 数首先对 ap_preloaded_modules 进行索引标记工作，然后将其中的元素逐一拷贝到 ap_loaded_modules 数组中。加载后需要 激活的模块都保存在 ap_prelinked_modules 中，为此函数将对该数组中的每一个元素调用 ap_add_modules 进行激活。
Apache2.0 中 ap_preloaded_modules 和 ap_prelinked_modules 是完全相同的，该函数刻意分为两个数组进行处理实际上延续的还是老版本的。
一旦激活所有的模块，函数调用 apr_hook_sort_all 对所有模块内的钩子进行排序，以便于处理。

```
////////////////////////////////////////////////////////////////////////////////
```

AP_DECLARE(const char *) ap_find_module_name(module *m)
函数获取模块 m 的名称
//
/////////////////
AP_DECLARE(module *) ap_find_linked_module(const char *name)
函数在所有的激活模块中查找名称为 name 的模块，如果查到，返回该模块；否则返回 NULL。
//
/////////////////
AP_DECLARE(int) ap_add_named_module(const char *name, apr_pool_t *p)
该函数将已经装入即存在于 ap_loaded_modules 数组中的具有指定名称 name 的模块激活。成功时返回 1；否则返回 0。

Apache 源代码分析——配置命令执行过程
该文章分析了 Apache 中的配置命令执行过程
该部分是与前面的命令表的部分紧密关联的，因此阅读这部分请先阅读前面的命令表分析。

//
static const char *invoke_cmd(const command_rec *cmd, cmd_parms *parms,
void *mconfig, const char *args)
apache 内部对命令的实现最终是通过 invoke_cmd 实现的，在 invoke_cmd 的基础之上进行了简单的包装，构成了 execute_now 函数。
函数首先检测该指令是否出现在它应该出现的地方。这个可以通过将通过 AllowOverride 设置的 parms->override 和 cmd->req_override 进行比较，如果相等，则表明合法，否则表明指令不应该出现在这个位置。
函数最终将根据 cmd->args_how 来进行实际的函数调用，args_how 指明了调用函数的实际的原型。首先我们看一下最简单的 args_how 为 RAW_ARGS 的情况，简化后函数的相关执行语句如下：

```
switch (cmd->args_how) {
    case RAW_ARGS:
        return cmd->AP_RAW_ARGS(parms, mconfig, args);
……
}
```

在 cmd 中我们声明了执行函数为 func，其类型为 cmd_fun。在前面我们说过不同指令最终实际的函数原型是不一样，其函数的参数由 args_how 决定，那么 Apache 是如何作到仅用一个函数名就达到调用各种不同函数的目的的呢。在 C++下这个倒是可以通过重载实现，而 C 中则没有直接的方法。为此 Apache 采取了一些变通的方法。我们看一下下面的定义就会明白实现策略了。

```
typedef union {
    const char *(*no_args) (cmd_parms *parms, void *mconfig);
    const char *(*raw_args) (cmd_parms *parms, void *mconfig,
        const char *args);
    const char *(*take1) (cmd_parms *parms, void *mconfig, const char *w);
```

const char *(*take2) (cmd_parms *parms, void *mconfig, const char *w, const char *w2);
const char *(*take3) (cmd_parms *parms, void *mconfig, const char *w, const char *w2, const char *w3);
const char *(*flag) (cmd_parms *parms, void *mconfig, int on);
} cmd_func

我们现在就会发现 cmd_func 原来是一个联合类型，apache 将所有可能用到的函数原型都放在里面的。因此需要的时候只需要从中去取就可以了。那么现在我们在看看宏 AP_RAW_ARGS 的实现：

define AP_RAW_ARGS func.raw_args

原来 AP_RAW_ARGS 宏无非就是用来获取 RAW_ARGS 类型指令处理函数的名字，一旦得到名称就可以将参数传递进去，进行类似下面的函数调用：

cmd->AP_RAW_ARGS(parms, mconfig, args);

至此，函数开始执行指令对应的函数。所有的指令的调用都大同小异，因此我们不再对调用方式做过多的描述，我们下面来看看不同类型的指令 apache 是如何处理的。

(1)，对于 RAW_ARGS 类型的指令，服务器不对其做任何修改，因此，传进来什么参数，服务器原封不动的再传给指令处理函数。

(2)，NO_ARGS 类型指令也很简单，服务器什么都不需要传入给执行函数。

(3)，如果是 TAKE1 类型的指令，由于指令后面带一个参数，为此必须得到该参数，函数将调用 ap_getword_conf 得到该参数，然后将其传入处理函数。对于 TAKE1 指令，即使后面存在多个指令参数，处理器也只取第一个。

(4)，如果是 TAKE2 或者 TAKE3 指令，处理方法与 TAKE1 指令类似，只是要调用两次或三次 ap_getword_conf 得到个个参数。如果指令后面所带的参数不能满足要求，函数将报错，返回错误信息。

(5)，如果指令是 TAKE12，TAKE13，TAKE23 或者 TAKE123，处理函数处理之前必须保证指令参数在规定范围之内。如果不足，相应参数设置为 NULL。

(6)，如果指令是 ITERATE，那么指令的参数个数将不确定，为此处理器将循环迭代调用 ap_getword_conf，每次取一个参数，然后调用 TAKE1 指令处理函数；处理完继续取下一个，处理代码如下：

```
while (*(w = ap_getword_conf(parms->pool, &args)) != '') {
errmsg = cmd->AP_TAKE1(parms, mconfig, w);
if (errmsg && strcmp(errmsg, DECLINE_CMD) != 0)
return errmsg;
}
```

(7)，ITERATE2 指令与 ITERAT1 不同，其对指令的迭代处理是从第二个参数开始，第一个指令不参与迭代。处理器取出第一个参数后，然后将它和第二个参数开始的每一个参数联合作为指令函数的两个处理参数，处理代码如下：

```
w = ap_getword_conf(parms->pool, &args);
……
while (*(w2 = ap_getword_conf(parms->pool, &args)) != '') {
errmsg = cmd->AP_TAKE2(parms, mconfig, w, w2);
……
}
```

(8)，FLAG 指令与前面的所有指令处理方法都不一样，该指令后面的参数只有两种：On 和

Off，其余的都是非法。如果为“On”，传递给处理函数的相应参数为 1，否则为 0，具体代码如下：

```
w = ap_getword_conf(parms->pool, &args);
if (*w == '' || (strcasecmp(w, "on") && strcasecmp(w, "off")))
return apr_pstrcat(parms->pool, cmd->name, " must be On or Off",NULL);
return cmd->AP_FLAG(parms, mconfig, strcasecmp(w, "off") != 0);
```

Apache 源代码分析——命令表解析

该文章主要对 Apache 中的命令表进行了介绍和分析

命令行参数处理

如果用户是通过命令行进行 Apache 启动，那么启动语法如下：

Httpd [-d directory][-D parameter] [-f file] [-C directive] [-c directive] [-L] [-l] [-S] [-V] [-X]

其中，-d 命令用来设置 ServerRoot，即服务器的根目录。-D 用来定义<IiDefine>的参数值，即预先定义一些变量。-f 用来设置配置文件的路径，正常情况下，我们使用 httpd.conf，不过通过-f 选项，我们可以进行修改。-L 用来列出当前可用的所有的命令，并退出。-l 选项用来列出 Apache 中编译的模块并退出。-S 用来显示虚拟主机的相关信息。-v 用来显示 Apache 的版本号以及 Apache 编译的时间，将其打印出来，同时退出。-V 显示编译设置，并退出。-X 用来将 Apache 设置成为单进程模式。这种模式通常用来进行调试使用。

在这些命令中与配置命令相关的选项有三个：-f，-C 和-c。由于命令行的命令和配置文件中命令的设置可能会相互覆盖，因此我们有必要考虑到这种覆盖可能。-C 和-c 就是处理这种情况。-C 列出了各种指令，这些指令必须在读入配置文件之前进行处理，而-c 列出的命令则必须在读入配置文件之后进行处理。Apache 中给出了两个 ap_array_header_t 类型的数组 ap_server_pre_read_config 和 ap_server_post_read_config 分别记录-C 和-c 之后所需要处理的命令。为了处理命令行参数，Apache 中定义了 apr_getopt_t 结构来保存，apr_getopt_t 结构如下：

```
struct apr_getopt_t {
    apr_pool_t *cont;
    apr_getopt_err_fn_t *errfn;
    void *errarg;
    int ind;
    int opt;
    int reset;
    int argc;
    const char **argv;
    char const* place;
    int interleave;
    int skip_start;
    int skip_end;
};
```

errfn 则是函数在发生错误的时候调用来打印错误消息。如果 errfn 为 NULL，则表明不是输出错误信息。errarg 则是用户定义的传给错误消息的第一个参数。

Ind 标记该选项在父选项中的索引。如果 lnd 与 argc 即参数的总个数相同，则表明已经到了处理的末尾。该选项应该是最后一个选项。

argc 和 argv 与通常的含义相同，分别表示参数的个数和参数字符串。

Place 通常用来记录与选项关联的一些参数。

Apache 对 ap_getopt_t 结构的初始化是从 apr_getopt_init 开始的。一旦初始化完毕，Apache 将不断调用 apr_getopt 对命令行进行解析，同时根据命令进行相应的设置。对命令行参数进行处理的函数大多数都集中在 getopt.c 文件中。

//

APR_DECLARE(apr_status_t) apr_getopt_init(apr_getopt_t **os, apr_pool_t *cont, int argc, const char *const *argv)

该函数主要对 apr_getopt()中需要使用的 ap_getopt_t 结构进行初始化。os 是 apr_getopt 函数中需要使用的 ap_getopt_t 结构。cont 则是初始化过程中需要使用的内存池。argc 和 argv 则是参数个数和参数字符串，其通常来源于 main(argc,argv) 中的相应变量。

//

APR_DECLARE(apr_status_t) apr_getopt(apr_getopt_t *os, const char *opts, char *optch, const char **optarg)

apr_getopt 是命令行参数解析的核心函数。os 则是前面使用 apr_getopt_init 初始化的结构，一旦进行初始化，main 函数中的 argc 和 argv 都保存 到了 os 中的 argc 和 argv 成员中。opts 是应用程序能够接受的可选项的字符串。Optch 则是下一个需要解析的字符串，函数会返回四个值，同时退出。返回值及其含义如下所示：

APR_EOF -- 没有更多的选项进行解析

APR_BADCH -- 发现一个错误的选项字符

APR_BADARG -- 该选项参数后面没有参数

APR_SUCCESS -- 下一个选项被发现

函数流程：

//

Apache 配置指令

//

Apache 中关于配置指令最重要的数据结构就是指令表了，指令表的结构为 command_rec，其定义如下：

```
typedef struct command_struct command_rec;
struct command_struct {
    const char *name;
    cmd_func func;
    void *cmd_data;
    int req_override;
    enum cmd_how args_how;
    const char *errmsg;
};
```

command_rec 结构描述了与处理命令相关的各个信息，与 ap_directive_t 不同，其通常位于模块内部，由模块使用。name 给出了命令的名称，其值与 ap_directive_t 中的 directive 相同；func 则是每个模块中用来处理该命令的方法，其是 cmd_func 类型，Apache 中定义 为 typedef const char *(*cmd_func) ()；由于 cmd_func 不带有任何参数，因此如果需要传入适当的参数的话只能通过 cmd_data 进行。

为了能够阐述 req_override 和 args_how 的具体含义，我们还需要阐述一些相关的概念。

1. 指令类型

Apache 中提供了 12 种类型的指令，这些类型是与实际的配置文件中指令处理相一致的。每

种指令都大同小异，唯一的区别就在于其处理的参数的数目以及在将指令传递给 指令实现函数之前，服务器如何解释这些参数的方式。由于各个指令的参数不相同，为此也导致了指令的处理函数的格式不相同。

apache 中对于指令类型的定义是通过枚举类型 cmd_how 来实现的，cmd_how 定义如下：

```
enum cmd_how {
    RAW_ARGS,
    TAKE1,
    TAKE2,
    ITERATE,
    ITERATE2,
    FLAG,
    NO_ARGS,
    TAKE12,
    TAKE3,
    TAKE23,
    TAKE123,
    TAKE13
};
```

对于所有的指令处理函数，其都将返回字符串。如果指令处理函数正确的处理了指令，那么函数返回 NULL，否则应该返回错误提示信息。对于各种指令，服务器的处理方法如下：

RAW_ARGS

该指令会通知 apache 不要对传入的参数做任何的处理，只需要原封不动的传递个指令处理函数即可。使用这种指令会存在一定的风险，因为服务器不做任何的语法检查，因此难免会有错误成为“漏网之鱼”。

这种指令的处理函数通常如下所示：

```
const char * func(cmd_parms* parms , void* mconfig,char* args);
```

args 只是简单的命令行内容，当然也包括指令在内。

TAKE1

顾名思义，这种类型的指令“Take 1 argument”，其只允许传入一个参数。 这种指令的处理函数通常如下所示：

```
const char * func(cmd_parms* parms , void* mconfig,const char* first);
```

ITERATE

该类型指令属于迭代类型。这种指令允许传入多个参数，不过一次只能处理一个，服务器必须遍历处理它们。每次遍历处理的过程又与 TAKE1 类型指令相同。因此这种指令的处理函数与 TAKE1 指令相同。

TAKE2 , TAKE12 , ITERATE2

TAKE2 类型必须向指令处理函数传入两个参数；而 TAKE12 可以接受一个或者两个参数。ITERATE2 与 ITERATE1 类似，都属于参数迭代处理类型，不过 ITERATE2 要求至少传递两个参数给函数。不过，第二个参数能够使用多次，服务器会遍历它们，直到所有的参数都传递给处理函数。如果只向 TAKE12 传递一个参数，那么服务器将把第二个参数设置为 NULL。这三种类型的指令处理函数原型如下：

```
const char *two_args_func(cmd_parms* parms , void* mconfig,
const char* first,const char* second);
```

TAKE3 , TAKE23 , TAKE13 , TAKE123

这 组指令最多都可以接受 3 个参数，如果参数超过三个，则处理函数将会报错。TAKE3 意味着参数必须是三个；TAKE23 则意味着至少两个参数，也可以为三 个，不能少于两个或者多于三个。TAKE13 则意味这可以接受一个参数或者三个参数，除此之外都是非法。TAKE123 意味着可以接受一个，两个或者三个 参数。这四种指令的处理函数原型如下：

```
const char *three_args_func(cmd_parms* parms , void* mconfig,
const char* first,const char* second,const char* third);
```

NO_ARGS

该 类型的指令不接受任何的参数，其最常用的就是作为复杂指令的闭标签存在。比如<Directory>总是必须 有</Directory>与之对应。尽管<Directory>通常需要一个参数来指定其所设置的目录，不过对于< /Directory>则没有这个参数。因此其就是 NO_ARGS 指令类型。这种指令的处理函数通常如下所示：

```
const char *no_args_func(cmd_parms* parms , void* mconfig);
```

FLAG

这 种类型最简单，其只允许用来启动或者关闭的指令。与前面的几种类型中，服务器直接将配置命令后的参数传递给函数不同，这种指令不会直接传递参数，而是首先 对参数进行处理，并将处理的结果 true 或者 false 作为进一步的参数传递给函数。这种指令的处理函数通常如下所示：

```
const char *flag_args_func(cmd_parms* parms , void* mconfig,int flag);
```

不 管是什么指令，其对应的处理函数都是以两个参数开始：cmd_parms* parms 和 void* mconfig。cmd_parms 结构用来存储处理配置命令时候所需要的辅助内容。在处理任何配置信息文件的时候，该结构都将被创建。其用于 apache 核心传递各种参数给指令处理方法。关于 cmd_parms 的具体解释，我们在后面将给出。

另一个参数 void* mconfig 表示针对指令位置的配置记录，基于所遇到的指令位置的不同，该配置记录可以是服务器配置记录，也可以是目录配置记录。

2. 指令位置

关 于指令的另外一个重要的概念就是指令位置字段的概念(location field)。位置字段主要用于控制各个指令在配置文件中允许出现的位置，包括三种：顶层位置，目录区和虚拟主机区。如果服务器发现一个指令不允许出现在 出现的地方，比如 LoadModule 只允许出现在顶层位置，如果发现其在<Directory>中出现，服务器将报错，同时打印错误信息退 出。另外位置字段还将控制指令是否允许在文件 .htaccess 中使用。

对于指令位置字段， Apache 中提供了下面几个控制选项：

```
#define OR_NONE 0
#define OR_LIMIT 1
#define OR_OPTIONS 2
#define OR_FILEINFO 4
#define OR_AUTHCFG 8
#define OR_INDEXES 16
#define OR_UNSET 32
#define ACCESS_CONF 64
#define RSRC_CONF 128
#define EXEC_ON_READ 256
#define OR_ALL (OR_LIMIT|OR_OPTIONS|OR_FILEINFO|OR_AUTHCFG|OR_INDEXES)
```

对于上面的选项，Apache 又分成两类：配置文件说明选项和.htaccess 文件说明选项。

ACCESS_CONF

该选项允许指令出现在 Directory 或者 Location 区间以内的顶级配置文件中，因此该选项通常用来对 Apache 中目录或者文件进行某些控制。

RSRC_CONF

该选项允许指令出现在 Directory 或者 Location 区间以外的顶级配置文件中，当然也可以出现在 VirtualHost 区间中，因此该选项通用用来对 Apache 服务器或者虚拟主机的进行某些控制。

EXEC_ON_READ

该选项是 Apache2.0 中新增加的。在 Apache1.3 中，对指令的处理是边读边执行的，而 Apache2.0 中并不是这样。Apache2.0 首先预处理配置文件，将所有的配置命令读取到一个树型结构中，树的每个结点为 ap_directive_t 类型。我们称之为配置树。一旦建立配置树，Apache 然后才会遍历并处理所有指令。通常情况下，这种处理方式会很好，因为延后处理使的可以控制模块之间的依赖性。比如，线程化的 MPM 需要在定义 MaxClients 指令之前就必须定义 ThreadsPerChild 指令。MPM 不会强求管理员处理这种情况，即使在配置文件中 ThreadsPerChild 定义在 MaxClients 之后 Apache 也会在分析之前在配置树中进行次序调整。不过延后处理也不是完美无缺，有的时候可能导致问题。比如，Include 通常用于在配置文件中读取另外一个配置文件。如果不能立即读取到配置文件，那么第二个配置文件中的配置命令将不可能生成到配置树中。解决的方法就是在读取到 Include 命令的时候取消延后策略，而是立即执行该命令。EXEC_ON_READ 选项可以强制服务器在将指令从配置文件中读取之后立即执行。

通过前面的分析，我们可以看出，如果某个指令可能会改变配置树，那么该指令就应该为 EXEC_ON_READ，不过所谓的改变配置树不是指改变配置树的顺序，而是改变配置树的信息。如果想要改变配置树的次序，那么应该在预先配置阶段进行或者在处理配置指令时候完成这样的工作。

除了上面的三个用于对配置文件进行控制，Apache 中还提供了八个选项用于控制.htaccess 文件中指令。

OR_NONE

该选项则不允许在.htaccess 文件中使用任何指令，这是默认值。不过大多数指令都会对其进行修改。只有那些仅仅在顶级配置文件中才有效的指令以及那些仅仅有服务器管理员才可以使用的指令才会设置该选项。

OR_LIMIT

只有那些可能涉及虚拟主机访问的指令才会设置该选项。在核心服务器上，Allow，Deny 以及 Order 指令都会设置该选项。使用这个选项的指令允许放置在 Directory 和 Location 标签中，以及 AllowOverride 设置为 Limit 的.htaccess 文件中。

OR_OPTIONS

使用该选项的指令通常用来控制特定的目录设置。在标准的 Apache 发行版本中，Options 指令和 XbitHack 指令都会使用该选项。使用该选项的指令必须置于 Directory 和 Location 以及 AllowOverride 设置为 Options 的.htaccess 文件中。

OR_FILEINFO

使用该选项的指令通常用于控制文档类型或者文档信息。设置该选项的指令包括 SetHandler，ErrorDocument，DefaultType 等等。这中类型的指令可以存在于 Directory 和 Location 标签以及 AllowOverride 设置为 FileInfo 的.htaccess 文件中。

OR_AUTHCFG

使用该选项的指令通常用语控制授权或者认证等信息。设置该选项的命令包括 AuthUserFile，AuthName 以及 Require。这种命令可以存在于 Directory 和 Location 以及

AllowOverride 设置为 AuthConfig 的 .htaccess 文件中。

OR_INDEXS

使用该选项的命令通常用于 控制目录索引的输出。示例指令包括 AddDescription，AddIcon，AddIconByEncoding。这种命令可以存在于 Directory 和 Location 以及 AllowOverride 设置为 Indexs 的 .htaccess 文件。

OR_ALL

这个选项是前面所有选项的组合。使用该选项的命令可以存在于 Directory 和 Location 标签中，以及只要 AllowOverride 不为 None 的 .htaccess 文件中。

OR_UNSET

这个特殊的值指出，这个目录没有设置重写。模块不应该使用值。核心会使用这个值来正确的控制继承。

目前在配置文件中，对于<Directory>标签外部的部分，req_override 状态为：

RSRC_CONF|OR_OPTIONS|OR_FILEINFO|OR_INDEXS

而在 <Directory> 标签内部的部分，状态为：

ACCESS_CONF|OR_LIMIT|OR_OPTIONS|OR_FILEINFO|OR_AUTHCFG|OR_INDEXS

而在 .htaccess 文件中，状态则由 AllowOverride 命令决定。

3. 指令处理宏

apache 中定义了十二种指令，为此 apache 中也相应的定义的配置处理命令。Apache 中定义的十二个处理宏定义如下：

```
# define AP_INIT_NO_ARGS(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, RAW_ARGS, help }
# define AP_INIT_RAW_ARGS(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, RAW_ARGS, help }
# define AP_INIT_TAKE1(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE1, help }
# define AP_INIT_ITERATE(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, ITERATE, help }
# define AP_INIT_TAKE2(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE2, help }
# define AP_INIT_TAKE12(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE12, help }
# define AP_INIT_ITERATE2(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, ITERATE2, help }
# define AP_INIT_TAKE13(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE13, help }
# define AP_INIT_TAKE23(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE23, help }
# define AP_INIT_TAKE123(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE123, help }
# define AP_INIT_TAKE3(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, TAKE3, help }
# define AP_INIT_FLAG(directive, func, mconfig, where, help)
        { directive, func, mconfig, where, FLAG, help }
```

从上面的定义可以看出，十二个宏，每个宏都具有相同的格式，唯一不同的就是其传入的

指令类型。正常的情况下在模块的 command_rec 表格中增加一个新 的指令有两种方法：一种是直接填充 command_rec 结构中的各个成员变量，另一种就是使用宏填充。如果使用宏，那么代码编译时就不会产生警告消息。 但是如果直接填充结构，那么在采用维护模式进行编译时，代码就会产生警告。下面我们来看一下 AP_INIT_TAKE1 的一个具体实现。

```
AP_INIT_TAKE1(“AuthType”,
ap_set_string_slot,
(void*)APR_XtOffsetOf(core_dir_config,ap_auth_type),
OR_AUTHCFG,
“An Http authorization(e.g,”Basic”)”
);
```

该 宏的第一个字段 AuthType 是指令的名称。该名称可以是任何合法的内容，但是不应该包含空格字符，Apache 在处理命令时候以空格作为结束符，一旦 遇到空格即停止。当服务器读取配置文件的时候，它就会读入各行代码，并且搜索所有已经激活模块中的指令。如果找不到指令，服务器就终止。

该宏 的第二个字段是该模块中对该命令的处理函数。在上面的示例中，模块中处理 AuthType 命令的处理函数为 ap_set_string_slog。尽管所 有的声明的函数的原型都相同，但是这个函数的原型会根据所定义的指令类型而改变。不同的类型的指令的原型在前面我们已经介绍过。

第三个字段是 应该向函数传递的附加数据。不同的函数附加数据不同。如果没有附加数据，则此处为 NULL 通过该字段的设置，Apache 可以用一个函数来处理多个指令。比如在 core 模块中 dirsection 函数方法同时可以处理 Directory 和 DirectoryMatch 两个命 令，那么该字段设置为 0(NULL) 和 1 就可以分开。在该示例中，附加数据为 (void*)APR_XtOffsetOf(core_dir_config,ap_auth_type)。核心服务器会使用 ap_set_string_slot 函数来设置结构中的字符串。因为这个函数可以用于设置任何结构中的任何字符串，所以服务器必须要传递结构中正确字段 的偏移量，以便 芃恢 浪 谋涞哪谌荨

◆PR_XtOffsetOf 宏能够计算这个偏移量，通过将这个宏放入到指令定义的第三个字段中，就可以根据第二 个字段中规定的函数得到这个偏移量。

第四个字段将会描述指令可以放置到配置文件的哪些位置。配置文件解析器能够确定指令是在目录中找到还是在虚拟主机区间找到。通过在这里规定这个值，核心服务器就能够进行基本的错误检查以确保没有在不合理的位置中规定指令。

最后一个字段就是在管理员不正确配置服务器时输出的错误字符串。这个错误的消息不应该解释指令所进行的工作。相反，它应该解释怎样使用指令。这样就可以帮助管理员改正可能存在的配置错误。

该宏最终的产生效果类似于下面的语句：

```
command_rec.name = “AuthType”;
command_rec.func = ap_set_string_slot;
command_rec.cmd_data = (void*)APR_XtOffsetOf(core_dir_config,ap_auth_type);
command_rec.req_override = OR_AUTHCFG;
command_rec.args_how = TAKE1;
command_rec.errmsg = “An Http authorization(e.g,”Basic”);
```

我们来看实际的例子，模块 mod_alias 中的命令表格如下：

```
static const command_rec alias_cmds[] =
{
    AP_INIT_TAKE2("Alias", add_alias, NULL, RSRC_CONF,
```

```
                                                    "a fakename and a realname"),
        AP_INIT_TAKE2("ScriptAlias", add_alias, "cgi-script", RSRC_CONF,
                                                    "a fakename and a realname"),
    AP_INIT_TAKE23("Redirect", add_redirect, (void *) HTTP_MOVED_TEMPORARILY,
                                                                OR_FILEINFO,
                        "an optional status, then document to be redirected and "
                                                                "destination URL"),
        AP_INIT_TAKE2("AliasMatch", add_alias_regex, NULL, RSRC_CONF,
                                                    "a regular expression and a filename"),
    AP_INIT_TAKE2("ScriptAliasMatch", add_alias_regex, "cgi-script", RSRC_CONF,
                                                    "a regular expression and a filename"),
                AP_INIT_TAKE23("RedirectMatch", add_redirect_regex,
                    (void *) HTTP_MOVED_TEMPORARILY, OR_FILEINFO,
                    "an optional status, then a regular expression and "
                                                                "destination URL"),
                AP_INIT_TAKE2("RedirectTemp", add_redirect2,
                    (void *) HTTP_MOVED_TEMPORARILY, OR_FILEINFO,
                    "a document to be redirected, then the destination URL"),
                AP_INIT_TAKE2("RedirectPermanent", add_redirect2,
                    (void *) HTTP_MOVED_PERMANENTLY, OR_FILEINFO,
                    "a document to be redirected, then the destination URL"),
                                                                {NULL}
};
```

Apache 中的进程剖析(1)

6.1 Apache 进程概述

对于大负载的服务器而言，系统内部的并发处理以及进程之间的通信处理非常的重要。良好的设计可以使服务器的效率和稳定性加倍提升。Apache 是服务器设计的一个典范，尤其是它的进程和线程处理，一直为人称道。本章我们详细分析 APR 中对进程和线程的封装。在下一章，我们将分析进程和线程间的通信。这两张综合起来构成了进程和线程的大部分内容。APR 中所有进程封装相关的源代码都保存在$(APR_HOME)/threadproc 目录下，其相应头文件则为$(APR_HOME)/include/apr_thread_proc.h。

尽管都称之为进程，但是不同的操作系统对进程的实现却是不同的。由于 APR 支持五种大的操作系统平台，OS/2，BeOS，Unix，Window32 以及 NetWare，因此进程的封装也细分为五种。我们仅仅讨论 Window 和 Unix 平台上的进程和线程的封装细节。本章我们首先分别讨论 Unix 和 Window 上的进程，继而讨论 Unix 和 Window 上的线程。

6.1.1 进程数据结构描述

尽管 APR 提供了五种进程的封装实现，但是 Apache 中统一使用数据结构 apr_proc_t 来描述某个进程：

```
typedef struct apr_proc_t {
    pid_t pid;
    apr_file_t *in;
    apr_file_t *out;
    apr_file_t *err;
#if defined(WIN32)
    HANDLE hproc;
#endif
} apr_proc_t;
```

上述的进程数据结构非常容易理解，pid 描述的是进程 id 号，对于 Unix 平台，该变量可以用 getpid() 函数的值进行填充，而对于 Window 平台，则用 PROCESS_INFORMATION 结构中的 dwProcessId 进行填充。in、out 和 err 则是与该进程关联的输入，输出以及错误描述符，正常情况下这三个对应标准输入输出 stdin、stdout 和 stderr。不过如果你愿意，你可以进行重定向。使用的最多的就是使用管道进行重定向。具体的细节部分，我们在后面的部分描述。

对于 Window 平台而言，描述一个进程，除了需要一个 DWORD 类型的全局进程 ID 号之外，还需要一个更重要的进程句柄。两者结合在一起才能完整地描述一个 Window 进程，因此对于 Window 平台而言,apr_proc_t 中还需要额外的 hproc 成员描述进程句柄。

6.1.2 进程属性描述

与进程相关联的另外一个最重要的数据结构应该就是进程属性描述数据结构 apr_procattr_t，每一个进程都会关联一个 apr_procattr_t 结构，该结构是个大杂烩，凡是跟进程相关的属性都可以塞到该结构中。不过由于 Unix 和 Window 中的进程属性描述存在一定的差异，因此我们分开描述

6.1.2.1Unix 平台

Unix 下的进程属性结构定义在 srclib\apr\include\arch\unix\apr_arch_threadproc.h 中，如下：

```
struct apr_procattr_t {
    /*Part 1*/
    apr_pool_t *pool;

    /*Part 2*/
```

```
    apr_file_t *parent_in;
    apr_file_t *child_in;
    apr_file_t *parent_out;
    apr_file_t *child_out;
    apr_file_t *parent_err;
    apr_file_t *child_err;

    /*Part 3*/
    char *currdir;
    apr_int32_t cmdtype;
    apr_int32_t detached;

    /*Part 4*/
    struct rlimit *limit_cpu;
    struct rlimit *limit_mem;
    struct rlimit *limit_nproc;
    struct rlimit *limit_nofile;

    /*Part 5*/
    apr_child_errfn_t *errfn;
    apr_int32_t errchk;

    /*Part 6*/
    apr_uid_t   uid;
    apr_gid_t   gid;
};
```

apr_procattr_t 结构中描述的进程信息包括六个部分：

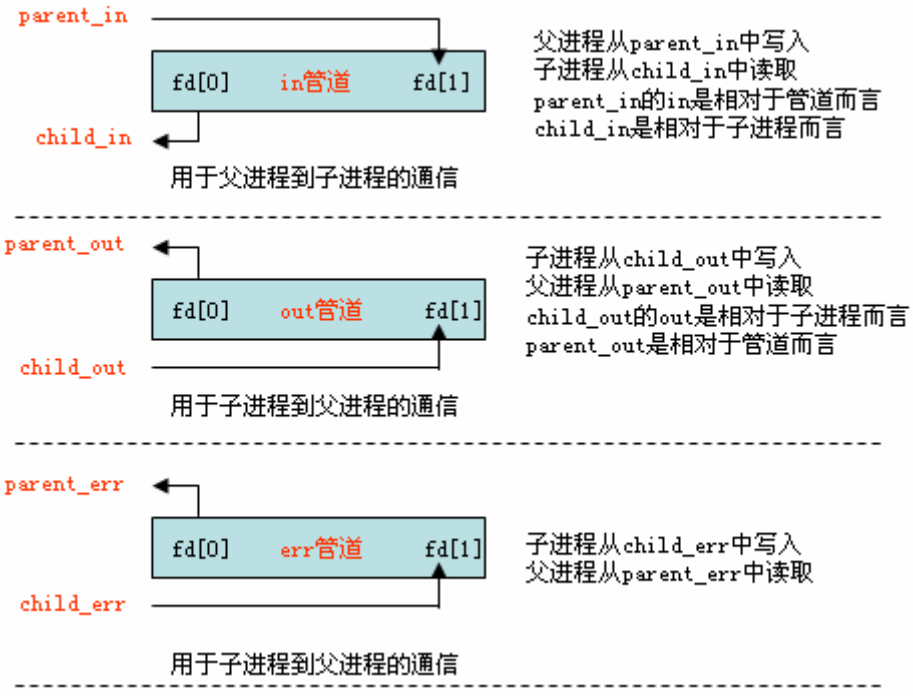
- **第一部分**

  pool 描述了 apr_procattr_t 结构的关联内存池，apr_procattr_t 结构分配所需要的内存都来自 pool。

- **第二部分**

该部分描述了父进程和子进程之间通信的管道文件。六 个描述符对应三个管道，分别用于父进程和子进程之间的通信：一个管道用于从父进程写入数据，从子进程中读出；一个管道用于从子进程写入数据，从父进程中读 出；一个管道负责子进程写入错误信息通知父进程。父进程或者子进程通常只使用六个描述符中的三个，而其余三个则关闭；至于关闭哪三个则由进程的角色决定： 对于父进程，通常关闭 child_XXX，保留 parent_XXX，而对于子进程则通常关闭 parent_XXX，保留 child_XXX。

它们之间的管道通信示意图如下描述：



为什么需要三对管道而不是一对，两对或者四对？这是跟前面的 apr_proc_t 结构中的描述符对应的，正如前面所说，每一个进程都维持三个描述符，正常情况下为 stdin、stdout 和 stderr，如果需要进行父子进程通信，那么我们就可以很快地将上面的管道描述符重定向到对应进程的描述符中。这就是为什么要建立三对的原因。

尽管如此，上面三对管道与进程并没有必然的联系。进程是否创建并不影响上面三对管道的创建，不过通常情况下总是先创建上面的管道，然后创建进程，然后再进程重定向。

apr_procattr_t 结构中的六个描述符一旦确定，那么 APR 就可以使用 apr_procattr_io_set 函数创建三对管道：

```
APR_DECLARE(apr_status_t) apr_procattr_io_set(apr_procattr_t *attr,
                                              apr_int32_t in,
                                              apr_int32_t out,
                                              apr_int32_t err)
```

attr 是管道的进程的描述结构，创建后的三对管道通过其返回出去。in、out 和 err 都是整数类型。in 用以标示是否需要创建上图中的“in 管道”，即从父进程到子进程的通信管道；out 则标记是否需要创建“out 管道”，即使从子进程到父进程的管道；err 则标记是否需要创建“err 管道”。这三个标记可能的取值包括下面五种：

1)、APR_NO_PIPE：该标记意味着不需要创建对应的管道，因此如果仅仅需要实现父进程到子进程的通信，那么 out 和 err 都可以设置为 APR_NO_PIPE。如果标记为不是该值，则必须创建管道，至于管道的属性则由具体的标记决定。

2)、APR_FULL_BLOCK：如果是该标记，则意味着必须创建管道，并且管道的两端都是阻塞的。默认情况下，管道创建后就是阻塞的，因此这个标记处理最简单，不需要做额外的工作。

3)、APR_FULL_NOBLOCK：创建管道并同时将管道对应的两个描述符都设置为非阻塞。将管道设置为非阻塞我们在管道部分描述过，请参考 apr_file_pipe_timeout_set 函数。

4)、APR_PARENT_BLOCK：仅仅父进程端的管道描述符设置为阻塞，这意味着子进程的管道描述符必须设置为非阻塞。

5)、APR_CHILD_BLOCK：仅仅子进程的管道描述符设置为阻塞，而父进程则设置为非阻塞。

下面是对 in 管道部分的处理，out 和 err 部分类似：

```
    apr_status_t status;
    if (in != 0) {
        if ((status = apr_file_pipe_create(&attr->child_in, &attr->parent_in,
                                           attr->pool)) != APR_SUCCESS) {
            return status;
        }
        switch (in) {
        case APR_FULL_BLOCK:
            break;
        case APR_PARENT_BLOCK:
            apr_file_pipe_timeout_set(attr->child_in, 0);
            break;
        case APR_CHILD_BLOCK:
            apr_file_pipe_timeout_set(attr->parent_in, 0);
            break;
        default:
            apr_file_pipe_timeout_set(attr->child_in, 0);
            apr_file_pipe_timeout_set(attr->parent_in, 0);
        }
    }
```

下面是一个典型的创建双向通信管道的示例代码：

```
apr_status_t rv;
apr_procattr_t *attr;
...
rv = apr_pool_create(&p, NULL);
rv = apr_procattr_create(&attr, p);
rv = apr_procattr_io_set(attr, APR_FULL_BLOCK,
APR_FULL_BLOCK,APR_NO_PIPE);
```

除了可以使用 apr_procattr_io_set 函数进行批量的一次性创建管道之外，APR 中还可以使用 apr_procattr_child_***XXX***_set 函数单独创建三个管道中的某一个，比如如果创建 in 管道，则函数名称为 apr_procattr_child_in_set，其定义如下：

```
APR_DECLARE(apr_status_t) apr_procattr_child_in_set(apr_procattr_t *attr,
                                                    apr_file_t *child_in,
                                                    apr_file_t *parent_in)
{
    apr_status_t rv = APR_SUCCESS;
    if (attr->child_in == NULL && attr->parent_in == NULL)
        rv = apr_file_pipe_create(&attr->child_in, &attr->parent_in, attr-
>pool);
    if (child_in != NULL && rv == APR_SUCCESS)
        rv = apr_file_dup2(attr->child_in, child_in, attr->pool);
```

```
    if (parent_in != NULL && rv == APR_SUCCESS)
        rv = apr_file_dup2(attr->parent_in, parent_in, attr->pool);
    return rv;
}
```

该过程可以创建管道，但是它的更重要的责任并不仅仅限于此。它隐含的另外一个重要的功能就是实现管道的重定向。

- **第三部分**

这部分描述了进程的一些常规属性。

currdir 标识新进程启动时的工作路径(执行路径)，默认时为和父进程相同；

cmdtype 标识新的子进程将执行什么类型的任务，APR 中使用枚举类型 apr_cmd_type_e 总共定义了五种任务类型，默认为 APR_PROGRAM：

```
typedef enum {
    APR_SHELLCMD,                /**< use the shell to invoke the program */
        APR_SHELLCMD_ENV         /**< use the shell to invoke the program
                                  replicating our environment*/
    APR_PROGRAM,                 /**< invoke the program directly, no copied
env */
     APR_PROGRAM_ENV,            /**< invoke the program, replicating our
environment */
      APR_PROGRAM_PATH,          /**< find program on PATH, use our
environment */
} apr_cmdtype_e;
```

为了能够深入理解 APR 中为什么将程序划分为这几种类型，我们有必要详细了解 Unix 中是如何执行一个新的应用程序的。Unix 中执行一个新的程序通常使用 exec 函数实现。当进程调用 exec 的时候，调用进程将被新的执行程序完全替代。新程序从 main 开始执行。由于 exec 函数的执行并不创建新的进程，因此新程序的进程号仍然是原有的进程号。exec 只是用另外一个程序替换了当前进程的正文、数据、堆栈。

Unix 中提供六个 exec 函数的变种供使用，APR 中应用程序的类型正是根据这六种类型划分的。

```
#include <unistd.h>
int execl(const char * pathname, const char * arg 0, ... /* (char *) 0 */);
int execv(const char * pathname, char *const argv [] );
int execle(const char * pathname, const char * arg 0, .../* (char *)0, char
*const envp [] */);
int execve(const char * pathname, char *const argv [], char *const envp [] );
int execlp(const char * filename, const char * arg 0, ... /* (char *) 0 */);
int execvp(const char * filename, char *const argv [] );
```

上面的六个函数都能执行一个新的应用程序，不过它们之间的差异主要在三方面：

1)、应用程序参数的传递方法的差别

exec 允许使用两种途径进行应用程序的参数传递。或者每一个命令行参数都单独作为函数的参数，或者先构造一个参数数组，将所有的命令行参数保存在该数组中，然后用数组的形式传递给函数。前一种的参数格式通常如下：

```
char *arg0, char *arg1, char *arg2,…,char *argn, (char*)0
```

最后一个实际的命令行参数后面必须跟一个(char*)0 空指针，这个不能省略，而且也不

能写为整数 0，必须将其转换为字符指针。ASCII 为 0 的字符正是空字符\0。
后一种用法的参数通常就是直接一个 char* argv[]数组而已，因此如果用这种策略传递的话，通常先将所有的参数生成字符串数组，然后再将该数组传入，在后面的使用示例中可以看到。
函数名称中通过 l 和 v 来进行这两个格式的区分，l 意味着参数为列表，使用第一个方法传递；v 意味着参数为向量，使用第二种途径传递。因此 execl，execle，execlp 属于前一种传递策略，execv，execve，execvp 属于后一种传递策略。不过 APR 中仅仅使用向量传递的方法，因此 APR 中只是用到了 execv，execve 和 execvp。
2)、环境变量表传递的差别
Unix 中每个进程都将收到两份表，一份就是参数表，另一份就是环境变量表。在最原先的 main 函数中参数是三个而不是我们现在所看到的两个：
int main(int argc, char *argv[], char *envp[])　　/*原始的 main 函数参数 */
第三个参数就是必须传递的环境变量表，通常情况下它是一个字符指针数组，其中每一个指针包含一个以 NULL 结尾的字符串的地址。不过 ANSI C 规定了 main 只能有两个参数，因此现在的做法通常是使用全局环境变量 environ 进行传递，比如：
上面函数的第二个差异就是对环境参数表的传递的差异。一种策略就是传递默认的系统全局环境参数表 environ，这种情况下该参数列表将是隐含的，因此不需要在参数列表中明确列出。另一种策略就是传递自定义的环境参数表，此时通过 environ 行不通，因此必须通过函数参数明确传递。上面的六个函数中，名称最后为 e 的则表明可以传递自定义的环境变量数组，比如 execle 和 execve。
应用程序类型 APR_SHELLCMD_ENV 和 APR_PROGRAM_ENV 都是需要传递自定义的环境变量参数，因此毫无疑问，对于这两种程序类型肯定是调用 execve(execle 是传递参数列表，APR 中不使用这种策略)。
3)、启动应用程序路径的差异
前面的四个程序都必须使用路径名称明确指定需要执行的程序的路径和名称，而且路径必须是绝对路径。而后两者则不一定，它们仅仅使用文件名称指名启动的程序。如果该名称是绝对路径名称(以/开始)，那么该名称被视为路径。如果给出的仅仅是文件名或者一个相对的文件名称，那么程序将在 PATH 和当前目录下查找对应的文件。
APR 中不支持使用程序名称启动程序的策略，毕竟在 PATH 指定的目录下进行搜索与直接给定程序路径相比显然要耗费时间的多。
下面是几个函数的使用示例：

```
1. execl("/bin/ls","ls","-al","/etc/passwd",(char * )0);
2. execlp("ls","ls","-al","/etc/passwd",(char *)0);
3. char * argv[ ]={"ls","-al","/etc/passwd",(char*) }};
       execv("/bin/ls",argv);
   4. char * argv[ ]={"ls","-al","/etc/passwd",(char *)0};
      char * envp[ ]={"PATH=/bin",0}
      execve("/bin/ls",argv,envp);
   5. char * argv[ ] ={ "ls","-al","/etc/passwd",0};
      execvp("ls",argv);
   6. char * envp[ ]={"PATH=/bin",0}
      execve("/bin/ls","ls","-al","/etc/passwd",(char * )0,envp);
```

因此尽管 UNIX 提供了六种调用接口，不过 APR 中仅仅使用了三种而已：

execv，execve 和 execvp。根据前面的分析，我们可以看到上面五种任务的区别，如下表所示：

任务类型	含义	环境变量传递方式	执行路径方式
APR_SHELLCMD	执行普通 SHELL 命令	自定义环境变量传递方式	
APR_SHELLCMD_ENV	执行普通 SHELL 命令	默认隐含传递方式	
APR_PROGRAM	普通应用程序	自定义环境变量传递方式	
APR_PROGRAM_ENV	普通应用程序	默认隐含传递方式	
APR_PROGRAM_PATH	普通应用程序		

detached 标识新进程是否为分离后台进程，默认为前台进程。

- **第四部分**

这 4 个字段标识平台对进程资源的限制，一般我们接触不到。struct rlimit 的定义在/usr/include/sys/resource.h 中。这四个字段的值是与 Apache 配置文件中的指令对应的。

为了防止某个子进程占用过多的 CPU，RLimitCPU 指令限制了 Apache 载入的子进程的 CPU 占用率。如果没有定义的话，则使用操作系统的默认值，该配置值最终保存在进程结构的 limit_cpu 中。

与此类似，为了防止子进程过度占用内存，Apache 中提供了 RLimitMEM 指令，该指令参数值最终保存在 limit_mem 中。为了限制由 Apache 载入的子进程的数目，配置中提供了 RLimitNPROC 指令，它的参数值由 limit_nproc 保存。

Apache 中可能会遇到的另外一个问题就是文件描述符耗尽。Unix 操作系统限制了每个进程允许使用的文件描述符的数目，典型的默认上限是 64。不过 64 个有时候不够用，为此需要进行扩充，直到到达一个很大的硬限制位置。调整后的文件描述符上限保存在 limit_nofile 中。

- **第五部分**

errfn 为一函数指针，原型为 typedef void (apr_child_errfn_t)(apr_pool_t *proc, apr_status_t err, const char *description); 这个函数指针如果被赋值，那么当子进程遇到错误退出前将调用该函数，从而可以完成一些清理工作。

errchk 一个标志值，用于告知 apr_proc_create 是否对子进程属性进行检查，如检查 curdir 的 access 属性等。

- **第六部分**

最后两个成员描述了创建进程的用户 ID 和组 ID，用于检索允许该用户所使用的权限。

Apache 中的进程剖析(2)

6.1.2.2Window 平台

相比于 Unix 下的 apr_procattr_t 结构，Window 下的该结构定义要简单一些，它定义在 Unix 下的进程属性结构定义在 srclib\apr\include\arch\win32\apr_arch_threadproc.h 中，如下：

```
struct apr_procattr_t {
    apr_pool_t *pool;
```

```
    /*Part 1*/
    apr_file_t *parent_in;
    apr_file_t *child_in;
    apr_file_t *parent_out;
    apr_file_t *child_out;
    apr_file_t *parent_err;
    apr_file_t *child_err;


    /*Part 2*/
    char *currdir;
    apr_int32_t cmdtype;
    apr_int32_t detached;
    apr_child_errfn_t *errfn;
    apr_int32_t errchk;


    /*Part 3*/
    HANDLE     user_token;
    LPSECURITY_ATTRIBUTES   sa;
    LPVOID                  sd;
};
```

Window 中该结构包含三方面的内容：

- **第一部分**

该部分与 Unix 中的功能相同，描述了父进程和子进程之间通信的三个管道文件：一个管道用于从父进程写入数据， 从子进程中读出；一个管道用于从子进程写入数据，从父进程中读出；一个管道负责子进程写入错误信息通知父进程。与 Unix 类似，Window 中的管道创建 可以使用 apr_procattr_io_set 统一创建，也可以使用 apr_procattr_child_in_set 单独创建。

apr_procattr_io_set 函数实现如下：

```
APR_DECLARE(apr_status_t) apr_procattr_io_set(apr_procattr_t *attr,
                                              apr_int32_t in,
                                              apr_int32_t out,
                                              apr_int32_t err)
{
    apr_status_t stat = APR_SUCCESS;


    if (in) {
        if (in == APR_CHILD_BLOCK)
            in = APR_READ_BLOCK;
        else if (in == APR_PARENT_BLOCK)     u
            in = APR_WRITE_BLOCK;
```

```
        stat = apr_create_nt_pipe(&attr->child_in, &attr->parent_in, in,
                                  attr->pool);
        if (stat == APR_SUCCESS)
            stat = apr_file_inherit_unset(attr->parent_in);
    }
    if (out && stat == APR_SUCCESS) {
        stat = apr_create_nt_pipe(&attr->parent_out, &attr->child_out, out,
                                  attr->pool); v
        if (stat == APR_SUCCESS)
            stat = apr_file_inherit_unset(attr->parent_out);
    }
    if (err && stat == APR_SUCCESS) {
        stat = apr_create_nt_pipe(&attr->parent_err, &attr->child_err, err,
                                  attr->pool); w
        if (stat == APR_SUCCESS)
            stat = apr_file_inherit_unset(attr->parent_err);
    }
    return stat;
}
```

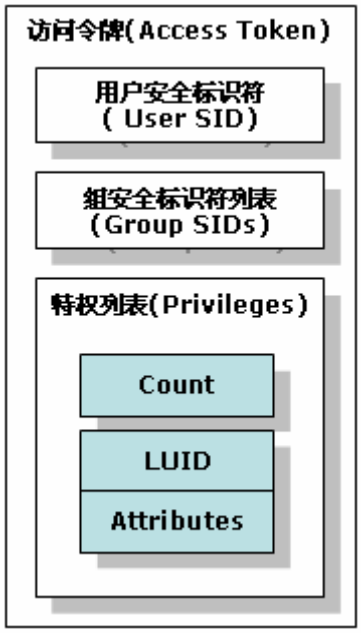
在 Window 下，管道的创建通过内部函数 apr_create_nt_pipe 实现，具体的创建细节我们在第七章的”管道章节”会详细描述。不过在 Window 管道中，

- **第二部分**
- **第三部分**

第三部分的内容是 Window 所独有的，主要用于 Window 的安全性。为了了解这几个成员的真实含义，我们必须对 Window 的安全性有一定的了解。

首先我们来看 use_token，它是一个 HANDLE 类型，表示用户的访问令牌。与 Window9X 系列操作系统相比，Window NT 以上的版本的安全性都得到了极大的提高。在 Windows 中，安全性是指将对象所受的保护与用户的访问权进行比较。如果用户拥有足够的访问权（访问权 等于或超过对象所受的保护），则用户能够使用这个对象。在 Windows 的文档中将对象所受保护的级别称作安全描述符（securitydescriptor），这是一种结构，它能告诉安全系统：用户需要什么样的权力才能访问 这个对象；而用户则有一个访问令牌（Access），它是另一种结构，它能告诉安全系统：在一个给定的位置，用户有什么样的访问权。令牌包括多方面的内容：用户安全标示符(SID)、组 SID 列表、特权列表和模拟(Impersonating)信息，可以用下图描述：

Token



用户将访问令牌交给 Windows NT 系统，并以此获得对对象的访问。当用户登录系统时，Windows NT/2000 将验证他的密码，如果用户验证成功，系统产生一个访问令牌。该用户启动的任何进程都将附加该令牌，访问令牌代表进程的安全环境，它控制了进程与可保护对象 (securable object)的交互。当进程访问一个可保护对象时，系统将该对象的访问控制列表(ACL)中的每个访问控制项(ACE)和访问令牌中的 SID 进行比较以确定进程是否可以访问该对象。由于用户启动的任何进程都将附加该用户的访问令牌，因此任何进程都知道用户的 SID 并且可以访问它。
Window 中不同的用户登录所产生的访问令牌是不一样的，因此它们的访问权限也不一样。在 Window 中大部分服务进程都是以本地系统帐户(local system)运行的，这是一个特殊的帐户，所以以本地系统帐户运行的进程和普通进程不同之处在于：
1) 注册表的 HKEY_CURRENT_USER 键是和缺省用户而不是当前用户相联系的，要访问其他用户的配置文件，需要先模拟该用户，然后再访问 HKEY_CURRENT_USER 。
2) 可以打开 HKEY_LOCAL_MACHINE\SECURITY 注册表键
3) 该进程不能访问网络资源，如共享、管道，因为它不能提供信任凭证，而只能使用空连接。在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters 处的 NullSessionPipes 和 NullSessionShares 的值指明了可以被空连接访问的管道和共享。或者也可以设置 RestrictNullSessAccess = 0，以允许空连接的用户访问该计算机上的所有管道和共享。(呵呵，这个是个安全隐患哦，可不要干呀。上边两个值最好也全都设成空)
4) 不能和其他程序共享对象，除非在创建对象时设置 DACL 允许用户访问。
5) 如果启动命令行提示符来运行批处理文件，用户可以按 Ctrl+C 来终止批处理的运行，并且用户就获得了一个 Local System 权限的 shell。
正是由于服务程序运行在权限极大的 Local System 账号下，如果网络客户也使用该账号来访问系统将给系统带来安全隐患，因此 NT/2000 提供了模拟功能 --- 服务程序在处理客户请求时使用一个权限较低的客户身份运行，处理完客户请求再恢复。user_token 则用于指定模拟的用户。不过 Window 中并不是 直接使用字符串类型来描述账户。由于每一个账户都对应一个访问令牌，因此，window 总是使用访问令牌标记模拟的用户。不过 APR 中提供了从用户名到访 问令牌的转换函数 apr_procattr_user_set。apr_proc_attr_user_set 的实现细节我们在下一章描述

如果 user_token 指定，那么在创建进程的时候，APR 将使用该模拟用户创建进程。创建的过程我们在进程创建章节描述。

Apache 中的进程剖析(3)

6.2 进程创建

6.2.1 Unix 系统中进程创建

APR 中通过 apr_proc_create 函数实现进程的创建，不过对于 APR 而言，创建进程并不仅仅是调用 fork 生成子进程就完毕了。整个创建可以用下面的伪码描述：

```
apr_proc_create
{
if (attr->errchk)
对 attr 做有效性检查，让错误尽量发生在 parent process 中，而不是留给 child process;
----(1)
fork 子进程;
{ /* 在子进程中 */
清理一些不必要的从父进程继承下来的描述符等，为 exec 提供一个“干净的”环境；------
(2)
关闭 attr->parent_in、parent_out 和 parent_err，
并分别重定向 attr->child_in、child_out 和 child_err 为 STDIN_FILENO、
STDOUT_FILENO 和 STDERR_FILENO; -----(3)
判断 attr->cmdtype，选择执行 exec 函数; ------(4)
}
/* 在父进程中 */
关闭 attr->child_in、child_out 和 child_err;
}
```

下面我们将上面的部分展开详细描述。

```
{
    int i;
    new->in = attr->parent_in;
    new->err = attr->parent_err;
    new->out = attr->parent_out;
```

除了调用 fork 简单的生成子进程之外，创建进程的一个重要的任务就是创建父进程和子进程之间的管道、重定向并确保父进程和子进程之间能够通过管道通信。

```
    if (attr->errchk) {
        if (attr->currdir) {
            if (access(attr->currdir, X_OK) == -1) {
```

```
                return errno;
            }
        }

        if (attr->cmdtype == APR_PROGRAM ||
            attr->cmdtype == APR_PROGRAM_ENV ||
            *progname == '/') {
            if (access(progname, R_OK|X_OK) == -1) {
                return errno;
            }
        }
        else {
            /* todo: search PATH for progname then try to access it */
        }
    }
```

是否需要对子进程进行安全性检查由父进程的 errchk 成员决定。通常情况下推荐进行检查，这样一旦子进程有问题的话，该错误将被扼杀在“襁褓”之中，而不错遗留到子进程中。可以通过函数 apr_procattr_error_check_set 设置该成员。检查包括：

1)、检查子进程是否具有对当前父进程路径的更改权限。因为在子进程中需要调用 chdir 函数，如果没有权限，自然不成功。

2)、如果子进程任务是普通的应用程序，并且使用的路径名称是绝对路径，那么必须确保它具有读取和修改权限，因此子进程中需要调用 exec()函数，如果权限不具备，该调用将不成功。

错误预处理的目的只有一个，就是让错误发生在 fork 前，不要等到在子进程中出错。

```
    if ((new->pid = fork()) < 0) {
        return errno;
    }
```

函数真正的调用 fork 产生子进程，此时程序将兵分两组执行。我们首先来看父进程中的工作：

6.2.1.1 父进程中的处理

```
    /* Parent process */
    if (attr->child_in) {
        apr_file_close(attr->child_in);
    }
    if (attr->child_out) {
        apr_file_close(attr->child_out);
    }
    if (attr->child_err) {
        apr_file_close(attr->child_err);
    }
```

父进程在创建 apr_procattr_t 结构的时候创建了 in、out 和 err 三个管道共计六个描述 符：parent_in、parent_out、parent_err、child_in、child_out 和 child_err，但是正如我们前面所言，对于父进程而言，与子进程通信仅仅需要 parent_in、parent_out、parent_err 三个，另外三个 child_XXX 则可以关 闭。

6.2.1.2 子进程中的处理

子进程中所作的工作与父进程类似：

```
        /* Part 1 */
        if (attr->child_in) {
            apr_pool_cleanup_kill(apr_file_pool_get(attr->child_in),
                                  attr->child_in, apr_unix_file_cleanup);
        }
        if (attr->child_out) {
            apr_pool_cleanup_kill(apr_file_pool_get(attr->child_out),
                                  attr->child_out, apr_unix_file_cleanup);
        }
        if (attr->child_err) {
            apr_pool_cleanup_kill(apr_file_pool_get(attr->child_err),
                                  attr->child_err, apr_unix_file_cleanup);
        }
       /*Part 2 */
        apr_pool_cleanup_for_exec();
       /*Part 3 */
        if (attr->child_in) {
            apr_file_close(attr->parent_in);
            dup2(attr->child_in->filedes, STDIN_FILENO);
            apr_file_close(attr->child_in);
        }
        if (attr->child_out) {
            apr_file_close(attr->parent_out);
            dup2(attr->child_out->filedes, STDOUT_FILENO);
            apr_file_close(attr->child_out);
        }
        if (attr->child_err) {
            apr_file_close(attr->parent_err);
            dup2(attr->child_err->filedes, STDERR_FILENO);
            apr_file_close(attr->child_err);
        }
```

上面的代码可以分为三部分：

① 子进程清理

由于子进程中的大部分属性都是从父进程进程而来，这些属性中并不是全部有用，为此子进程首先清除从父进程中进程 的与自己无关的垃圾信息，从而为 exec 提供一个干净的环境。清理工作由函数 apr_pool_cleanup_for_exec 实现。我们来看一下函数 内到底对子进程进行了哪些清理：

```
APR_DECLARE(void) apr_pool_cleanup_for_exec(void)
{
    cleanup_pool_for_exec(global_pool);
}
```

```
static void cleanup_pool_for_exec(apr_pool_t *p)
{
    run_child_cleanups(&p->cleanups);
    for (p = p->child; p; p = p->sibling)
        cleanup_pool_for_exec(p);
}
```

从上面的代码中可以看出，清理的过程实际上是一个递归的过程。它从内存池根结点开始，逐一遍历内存池中的每一个 结点，并调用结点内部对应的 cleanup_t 链表中的各个 cleaup_t 函数，对于管道而言，cleanup_t 函数的注册是在使用 apr_file_pipe_create 函数的时候注册的：

```
apr_pool_cleanup_register((*in)->pool, (void *)(*in),
                          apr_unix_file_cleanup,apr_pool_cleanup_null);
apr_pool_cleanup_register((*out)->pool, (void *)(*out),
                          apr_unix_file_cleanup,apr_pool_cleanup_null);
```

因此，cleanup_pool_for_exec 函数对于每一个内存池结点调用的实际上就是 apr_unix_file_cleanup 和 apr_pool_cleanup_null 函数。在 apr_unix_file_cleanup 中，对于普通文件描述符，如果该文件描述符进行了缓冲，那么首先要调用 apr_file_flush 进行缓冲刷新。由于管道是不使用缓冲的，因此缓冲的处理对管道 不进行任何处理。事实上，对于管道描述符，清理操作所作的事情主要就是调用 close 关闭，如果文件的标志为 APR_DELONCLOSE，意味着该文件 在关闭后必须删除，那么同时调用 unlink 删除该文件。

```
        /*
         * If we do exec cleanup before the dup2() calls to set up pipes
         * on 0-2, we accidentally close the pipes used by programs like
         * mod_cgid.
         *
         * If we do exec cleanup after the dup2() calls, cleanup can accidentally
         * close our pipes which replaced any files which previously had
         * descriptors 0-2.
         *
         * The solution is to kill the cleanup for the pipes, then do
         * exec cleanup, then do the dup2() calls.
         */
```

② 建立子进程与父进程的通信管道

父进程在创建 apr_procattr_t 时就建立了若干个管道，fork 后子进程继承了这些管道，因此子进程 内部同时也具备了 in、out 和 err 三个管道共计六个描述符：parent_in、parent_out、parent_err、child_in、 child_out 和 child_err，但是子进程仅仅需要 child_in、child_out、child_err 三个，另外三个 parent_XXX 则可以关闭，如下图的(1)，(2)所示。整个子进程中的描述符变化如下图所示：

子进程中的管道描述符变化

关于子进程，另外的问题就是子进程所拥有的描述符。通常的进程都拥有三个最基本的描述符：标准输入描述符，标准 输出描述符以及标准错误描述符，分别对应 stdin，stdout 和 stderr 三个标准设备。除此之外，APR 中创建的进程还拥有 child_XXX 和 parent_XXX 六个描述符，共计九个描述符。当所有的 parent_XXX 描述符关闭后，子进程中还拥有六个描述符。

子进程中标准输入，标准输出以及标准错误三个描述符的存在，意味着子进程能够从标准输入接受数据，并向标准输出 设备和标准错误设备输出数据。APR 中并不希望子进程具有这种能力，

它希望子进程所有的交互都来自父进程。如果需要从输入设备接受数据，也是父进程进程接 受，然后通过管道传递给子进程；同样，如果子进程需要输出数据到屏幕，也必须首先将数据通过管道传递给父进程，然后由父进程输出。这样带来的好处就是避免 了子进程的中可能遇到的错误，而由父进程统一管理。比如最简单的，如果子进程需要接受命令行，那么每个子进程必须对命令行进行预处理，这样无疑使得子进程 变得臃肿和复杂。为此 APR 中对子进程中的 STDIN_FILENO，STDOUT_FILENO 和 STDERR_FILENO 使用管道描述符进行了重定 向：

```
dup2(attr->child_in->filedes, STDIN_FILENO);
apr_file_close(attr->child_in);
```

上面的代码将 child_in 重定向到 STDIN_FILENO，这样，由 于 STDIN_FILENO 被覆盖，子进程所有的数据只能来自父进程；与此类似，子进程所有的数据只能输出到父进程，而不能输出到其余的输出设备。这样， 即使在子进程中调用 scanf 或者 printf，实际上也并不来自 stdin 和 stdout，而是来自父进程。重定向后的父子进程的描述符和管道的关系如 下：

③ 启动程序前的准备工作

在执行应用程序之前，子进程进行一些启动相关的准备工作，包括：

1)、包括切换执行目录。子进程的工作目录必须与父进程相同。

2)、切换用户组 Id 和用户 Id。Apache 中子进程通常是实际的与客户进行通信的实体，为了防止可能潜在的 黑客攻击，APR 中希望子进程在正常运行的情况下，执行权限保持尽可能的低，这样即使黑客控制了子进程也对系统不会产生太大的影响。这种设置通常只有父进 程使用 root 权限创建子进程的时候才需要设置。如果父进程本身的权限比较低，那么子进程继承的权限自然也很低，此时就不需要调整。

3)、设置进程极限值，包括 CPU 的极限，子进程使用内存的极限，启动的子进程的数目以及打开的文件描述符的数目。设置通过专门的 limit_proc 过程实现。函数内部无非调用的是 setrlimit 函数，比如：

```
setrlimit(RLIMIT_CPU, attr->limit_cpu);
setrlimit(RLIMIT_NPROC, attr->limit_nproc);
```

④ 启动应用程序

尽管子进程被 fork 后它就被处于活动状态，但是它到目前为止还没有获得实际的执行任务。Unix 中通常通过 exec 系列函数来启动一个新的应用程序。

对于子进程最后的任务就是执行实际的任务。如果启动应用程序，由需要启动的程序类型即 cmd_type 决定。cmd_type 的值以及含义如下表所示：

cmd_type 类型	含义	是否需要指定程序路径	是否使用自定义环境变量
APR_PROGRAM	启动的是普通的应用程序	是	是
APR_PROGRAM_ENV	启动的是普通的应用程序	是	否
APR_PROGRAM_PATH	启动的是普通的应用程序	否	是
APR_SHELLCMD	启动的是 Shell 应用程序	否	是
APR_SHELLCMD_ENV	启动的是 Shell 应用程序	否	否

对于每一种类型，函数处理如下：

1)、普通的应用程序(cmdtype=APR_PROGRAM)

```
if (attr->cmdtype == APR_PROGRAM) {
```

```
     if (attr->detached) {
          apr_proc_detach(APR_PROC_DETACH_DAEMONIZE);
    }
         execve(progname, (char * const *)args, (char * const *)env);
}
```

APR_PROGRAM 类型对应的是普通的应用程序，但是它并不使用全局的环境变量 environ，而是使用自 定义的环境变量数组。因此函数必须调用 execve。由于 APR 中不支持参数列表，为此 execle 不再考虑。另外如果子进程需要与父进程脱离开成后后台 进程，那么还需调用 apr_proc_detach 进行脱离操作。

2)、普通应用程序(cmdtype=APR_PROGRAM_ENV)

```
if (attr->cmdtype == APR_PROGRAM_ENV) {
     if (attr->detached) {
          apr_proc_detach(APR_PROC_DETACH_DAEMONIZE);
     }
     execv(progname, (char * const *)args);
}
```

对于 APR_PROGRAM_ENV，它与 APR_PROGRAM 的唯一的区别就是它使用默认的全局环境变量，因此不需要在函数参数中明确传递环境变量数组。这可以由 execv 函数实现。

3)、普通应用程序(cmdtype=APR_PROGRAM_PATH)

```
if(attr->cmdtype == APR_PROGRAM_PATH)
{
     if (attr->detached) {
          apr_proc_detach(APR_PROC_DETACH_DAEMONIZE);
     }
     execvp(progname, (char * const *)args);
}
```

如果程序的类型为 APR_PROGRAM_PATH，那么意味着这是一个普通的应用程序，但是与 APR_PROGRAM 和 APR_PROGRAM_ENV 不同，它允许仅仅指定应用程序的名称，而不需要指明完整的绝对路径，具体的路径则由操作系统在 PATH 目录下查找。这种情况使用 execvp 正好合适。

4)、普通 Shell 程序

```
    if (attr->cmdtype == APR_SHELLCMD ||
       attr->cmdtype == APR_SHELLCMD_ENV) {
            int onearg_len = 0;
            const char *newargs[4];

            newargs[0] = SHELL_PATH;
            newargs[1] = "-c";
            i = 0;
            while (args[i]) {
                onearg_len += strlen(args[i]);
                onearg_len++; /* for space delimiter */
                i++;
            }
            switch(i) {
```

```
            case 0:
                break;
            case 1:
                newargs[2] = args[0];
                break;
            default:
            {
                char *ch, *onearg;
                ch = onearg = apr_palloc(pool, onearg_len);
                i = 0;
                while (args[i]) {
                    size_t len = strlen(args[i]);

                    memcpy(ch, args[i], len);
                    ch += len;
                    *ch = ' ';
                    ++ch;
                    ++i;
                }
                --ch; /* back up to trailing blank */
                *ch = '\0';
                newargs[2] = onearg;
            }
        }
        newargs[3] = NULL;
        if (attr->detached) {
            apr_proc_detach(APR_PROC_DETACH_DAEMONIZE);
        }
        if (attr->cmdtype == APR_SHELLCMD) {
            execve(SHELL_PATH, (char * const *) newargs, (char * const *)env);
        }
        else {
            execv(SHELL_PATH, (char * const *)newargs);
        }
    }
```

对于Shell程序而言，它也是一个普通的应用程序，无非需要程序的名称，程序提供 的参数等等，因此与前面的类似，可以通过execve和execv执行，具体调用哪一个，则取决于程序的类型。APR_SHELLCMD类型意味着应用程 序是Shell，但是使用自定义的环境变量数组；而APR_SHELLCMD_ENV则意味着使用默认的environ数组。

不过与普通的应用程序不同的是，对于shell程序仅仅给定程序的名称和路径还不 够，还必须给出shell执行程序的路径名称，对于Unix系统，通常执行shell通常是位于“/bin/sh”，而Window下则通常是 “cmd.exe”。因此如果shell应用的用法是”run －n tingya”，则真正执行的时候应该变换为”/bin/sh run －n tingya”，所以对于shell应用而言不能像

APR_PROGRAM 直接将传入的 args 参数传递给 exec 函数，而需要进行额外的处理。这就是 为什么要把 SHELL 单独辟出来成为两个类型的应用。
参数数组的转换非常的简单，只是在原有的数组的前面插入两个新的字符串“/bin/sh -c”，这样传入的所有的命令 xxx 都变为“/bin/sh -c xxx”，-c 选项的用途是通知 shell 处理程序把-c 后面的字符串作为一个参数处理
Apache 中的进程剖析(4)

6.2.2 Window 系统中进程创建

6.2.2.1 进程创建概述

Window 系统中创建进程毫无疑问，肯定是使用 CreateProcess 函数， 或者是 Unicode 版本的 CreateProcessW，或者是 ASCII 版本的 CreateProcessA。不过与 Unix 中创建进程不单是调用 fork 一样，Window 中创建进程也不仅是调用 CreateProcess 这么简单而已。事实上 Window 中对进程的创建要比 Unix 中还要复杂的 多，一方面是 Window 操作系统的版本比较多，为了考虑移植性，必须考虑到多个操作系统；另一方面，Window 中对。。。。
在大部分的 Window 相关的程序中我们都会看到下面几个预处理宏，这里有必要解释一下。
_WIN32_WCE
该宏意味着当前的应用程序运行在 Window CE 平台上；因此宏内部的代码仅仅适用于 Window CE 平台。
APR_HAS_UNICODE_FS
该宏意味着当前的文件系统支持 Unicode。对于 Window 系统而言，则主要只 Window NT 以上的版本；
APR_HAS_ANSI_FS
该宏意味着当前的文件系统是 ASCII 编码，对于 Window 系统而言，则主要指 Window 9X 系列，包括 Window 95，Window 98 以及 Window ME。
Window 中进程的创建过程可以用下面的伪码描述整体概况：
apr_proc_create
{
if (attr->errchk)
对 attr 做有效性检查，让错误尽量发生在 parent process 中，而不是留给 child process; ----(1)
fork 子进程;
{ /* 在子进程中 */
清理一些不必要的从父进程继承下来的描述符等，为 exec 提供一个“干净的”环境；------(2)
关闭 attr->parent_in、parent_out 和 parent_err，
并分别重定向 attr->child_in、child_out 和 child_err 为 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO; -----(3)
判断 attr->cmdtype，选择执行 exec 函数; ------(4)
}
/* 在父进程中 */
关闭 attr->child_in、child_out 和 child_err;
}
下面的部分我们将针对每一部分详细展开描述。

6.2.2.2 创建过程

```
        new->in = attr->parent_in;
    new->out = attr->parent_out;
    new->err = attr->parent_err;

    if (attr->detached) {
        if (apr_os_level >= APR_WIN_NT) {
            dwCreationFlags |= DETACHED_PROCESS;
        }
    }
```

DETACHED_PROCESS 标志是一个与控制台相关的创建标志。默认情况下， 如果应用程序创建一个控制台应用程序，那么该进程将继承共享父进程的控制台，并且该子进程的所有的输出信息都将在父进程的控制台中显示，而且交互也只能通 过父进程的控制台显示。不过这并不能保证一定会成功。

有的时候并不希望子进程继承父进程的控制台，而是拥有自己的控制台。此时有几种途径可以实现这种效果：

1)、A GUI or console process can use the CreateProcess function with CREATE_NEW_CONSOLE to create a console process with a new console. (By default, a console process inherits its parent's console, and there is no guarantee that input is received by the process for which it was intended.)

2)、A graphical user interface (GUI) or console process that is not currently attached to a console can use the AllocConsole function to create a new console. (GUI processes are not attached to a console when they are created. Console processes are not attached to a console if they are created using CreateProcess with DETACHED_PROCESS.)

不过只有 Window NT 以上的版本才能支持新的 DETACHED_PROCESS 标志，而 Win9X 系列的操作系统则无能为力。

```
    if (progname[0] == '\"') {
        progname = apr_pstrndup(pool, progname + 1, strlen(progname) - 2);
    }
```

Window 中不允许传入的运行程序名称中包含双引号”“，因此如果发现程序名称被”“包含，则首先必须将”“剔除，才能继续往下操作。

```
    if (attr->cmdtype == APR_PROGRAM || attr->cmdtype == APR_PROGRAM_ENV) {
        char *fullpath = NULL;
       if ((rv = apr_filepath_merge(&fullpath, attr->currdir, progname,
                                    APR_FILEPATH_NATIVE, pool)) != APR_SUCCESS) {
            if (attr->errfn) {
                attr->errfn(pool, rv,
                            apr_pstrcat(pool, "filepath_merge failed.",
                                        " currdir: ", attr->currdir,
                                        " progname: ", progname, NULL));
            }
            return rv;
```

```
        }
        progname = fullpath;
    }
    else {
        char *fullpath = NULL;
        if ((rv = apr_filepath_merge(&fullpath, "", progname,
                                     APR_FILEPATH_NATIVE, pool)) == APR_SUCCESS) {
            progname = fullpath;
        }
    }
```

在前面的部分我们曾经描述过五种应用程序类型的实际含义。对于一些应用类型，用户只 需要指定应用程序的名称，而不需要指定完整的路径名称就可以执行，比如 APR_SHELLCMD、APR_SHELLCMD_ENV 和 APR_PROGRAM_PATH。但是不管哪一种应用程序类型，最终它们的执行都是 CreateProcess 函数，而该函数需要完整的程序路径作为参 数，因此函数内部必须能够根据传入的程序名称和程序类型确定出完整的。不同的程序类型，绝对路径确定的方法可以用下表描述

程序类型 cmd_type	确定绝对路径的方法
APR_PROGRAM	使用启动进程的当前路径作为路径
APR_PROGRAM_ENV	使用启动进程的当前路径作为路径
APR_PROGRAM_PATH	查找环境变量”PATH”指定的路径下是否存在该程序，如果存在，使用该路径作为绝对路径
APR_SHELLCMD	使用”COMSPEC”指定的路径作为绝对路径
APR_SHELLCMD_ENV	使用”COMSPEC”指定的路径作为绝对路径

从上表中可以看出，对于 APR_PROGRAM 和 APR_PROGRAM_ENV 类 型的程序，它们的绝对路径实际上是执行进程的当前路径 currdir 和程序名称的组合，即 currdir+progname。而对于其余三种类型，暂时只 是简单的处理，在后面的部分它们将被继续处理。

```
    if (has_space(progname)) {
        argv0 = apr_pstrcat(pool, "\"", progname, "\"", NULL); u
    }
    else {
        argv0 = progname;
    }
/* Handle the args, seperate from argv0 */
    cmdline = "";
    for (i = 1; args && args[i]; ++i) {
        if (has_space(args[i])) {
            cmdline = apr_pstrcat(pool, cmdline, " \"", args[i], "\"", NULL);
        }                          v
        else {
            cmdline = apr_pstrcat(pool, cmdline, " ", args[i], NULL);
        }
    }
```

对于 CreateProcess 函数，Window 规定如果传入的启动程序名称和参 数中包含空格，那么这些名称和参数在传入给 CreateProcess 函数之前必须用双引号”“进行包含，比如 c:\program

files\sub dir\program name，如果不用”“包含，则Window可能会产生歧异，因为解释有多种(黑体部分为可执行程序名称，而细体部分为参数)：

c:\program.exe files\sub dir\program name

c:\program files\sub.exe dir\program name

c:\program files\sub dir\program.exe name

c:\program files\sub dir\program name.exe

在u中，函数首先判断程序名称中是否包含空格，如果是，则将各部分用”“包含起来。同样在v中， 对于传入的执行程序需要的参数列表args，应用程序也必须检查各个参数中是否包含空格，比如如果某个参数为”hello world”，那么直接传入，可能会被程序误解为两个不同的参数”hello”和”world”，因此，对于这些包含空格的参数也必须使用”“包含起来。 这些处理后的参数最终保存在cmdline中。

```
    if (attr->cmdtype == APR_SHELLCMD || attr->cmdtype == APR_SHELLCMD_ENV) {
        char *shellcmd = getenv("COMSPEC");
        if (!shellcmd) {
            if (attr->errfn) {
                attr->errfn(pool, APR_EINVAL, "COMSPEC envar is not set");
            }
            return APR_EINVAL;
        }
        if (shellcmd[0] == '"') {
            progname = apr_pstrndup(pool, shellcmd + 1, strlen(shellcmd) - 2);
        }
        else {
            progname = shellcmd;
            if (has_space(shellcmd)) {
                shellcmd = apr_pstrcat(pool, "\"", shellcmd, "\"", NULL);
            }
        }
        /* Command.com does not support a quoted command, while cmd.exe demands
one.
         */
        i = strlen(progname);
        if (i >= 11 && strcasecmp(progname + i - 11, "command.com") == 0) {
            cmdline = apr_pstrcat(pool, shellcmd, " /C ", argv0, cmdline, NULL);
        }
        else {
            cmdline = apr_pstrcat(pool, shellcmd, " /C \"", argv0, cmdline, "\"",
NULL);
        }
    }
```

在前面我们描述过，对于APR_SHELLCMD和APR_SHELLCMD_ENV类型的应用程序，它的绝对路径由环境变量COMSPEC指定。如果COMSPEC环境变量不存在，则执行将失败。如果存在，则同样将其中的空格字符串用”“包围起来。

shell 应用程序的程序名称或者是 cmd.exe(Window NT 以上版本)或者是 command.com(Window 9X 系列)。command.com 程序不支持命令中出现双引号，而 cmd.exe 则必须用双引号将命令包括起来。

```
i = strlen(progname);
if (i >= 4 && (strcasecmp(progname + i - 4, ".bat") == 0
            || strcasecmp(progname + i - 4, ".cmd") == 0))
{
      char *shellcmd = getenv("COMSPEC");
      if (!shellcmd) {
           if (attr->errfn) {
                attr->errfn(pool, APR_EINVAL, "COMSPEC envar is not set");
           }
           return APR_EINVAL;
      }
      if (shellcmd[0] == '"') {
          progname = apr_pstrndup(pool, shellcmd + 1, strlen(shellcmd) - 2);
      }
      else {
          progname = shellcmd;
          if (has_space(shellcmd)) {
              shellcmd = apr_pstrcat(pool, "\"", shellcmd, "\"", NULL);
         }
      }
      i = strlen(progname);
      if (i >= 11 && strcasecmp(progname + i - 11, "command.com") == 0) {
          cmdline = apr_pstrcat(pool, shellcmd, " /C ", argv0, cmdline, NULL);
      }
      else {
          cmdline = apr_caret_escape_args(pool, cmdline);
          if (*argv0 != '"') {
              cmdline = apr_pstrcat(pool, shellcmd, " /C \"\"", argv0, "\"",
cmdline, "\"", NULL);
          }
          else {
              cmdline = apr_pstrcat(pool, shellcmd, " /C \"", argv0, cmdline, "\"",
NULL);
          }
    }
}
```

如果应用程序是批处理程序(.bat)或者命令程序(.com)，则处理过程是一样的。

为了创建一个新的进程，至少必须具备下面的几个要素：

1、程序所在的所在的绝对路径

2、进程所需要的环境变量

在下面的函数中代码将会因为操作系统编码的不同而导致差异。Window 9X 系列的操作系统是基于 ASCII 编码，而 Window NT 以上版本则是基于 Unicode 编码，这由宏 APR_HAS_UNICODE_FS 和 APR_HAS_ANSI_FS 区分：

```
#if APR_HAS_UNICODE_FS
……
#endif
#if APR_HAS_ANSI_FS
……
#endif
```

我们首先讨论简单的 Window 9X 操作系统中的细节。

```
    if (!env || attr->cmdtype == APR_PROGRAM_ENV ||
        attr->cmdtype == APR_SHELLCMD_ENV) {
        pEnvBlock = NULL;
    }
```

Window 在调用 CreateProcess 的时候需要传递一个环境变量块，如果为 NULL，则新进程将使用调用进程的环境变量。

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

name=value

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by lpEnvironment contains Unicode characters, be sure that dwCreationFlags includes CREATE_UNICODE_ENVIRONMENT.

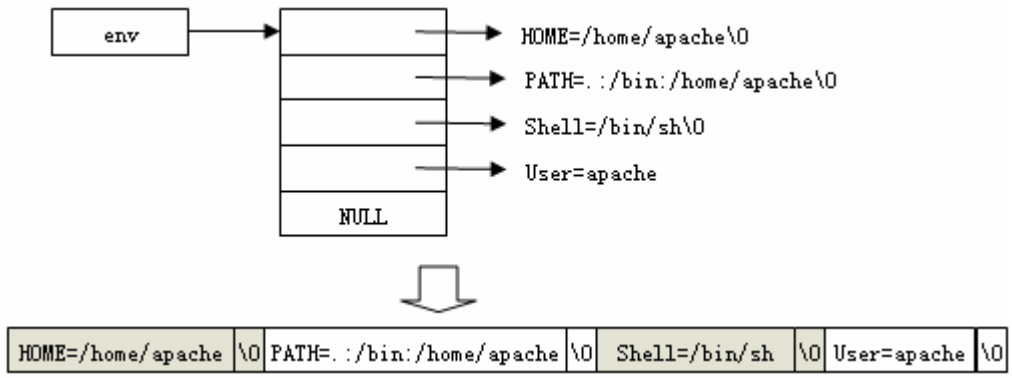
Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

对于 APR_PROGRAM_ENV 和 APR_SHELLCMD_ENV 程序类型，它们使用父进程的环境变量。因此传递给 CreateProcess 的环境变量块 pEnvBlock 为 NULL。

```
    else {
        apr_size_t iEnvBlockLen;
        i = 0;
        iEnvBlockLen = 1;
        while (env[i]) {
            iEnvBlockLen += strlen(env[i]) + 1;
            i++;
        }
        if (!i)
            ++iEnvBlockLen;
        {
            char *pNext;
            pEnvBlock = (char *)apr_palloc(pool, iEnvBlockLen);
```

```
            i = 0;
            pNext = pEnvBlock;
            while (env[i]) {
                strcpy(pNext, env[i]);
                pNext = strchr(pNext, '\0') + 1;
                i++;
            }
             if (!i)
                *(pNext++) = '\0';
             *pNext = '\0';
        }
    }
```

由于CreateProcess所需要的环境变量块实际上是一个字符串，而传入的参数env则是字符串数组，因此必须完成转换。转换的过程对于ASCII操作系统而言，无非是执行strcpy进行拷贝而已。转换前后示意如下所示：

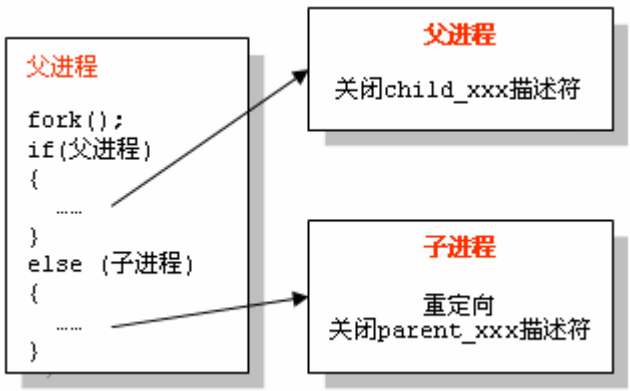



```
    new->invoked = cmdline;
    {
        STARTUPINFOA si;
        memset(&si, 0, sizeof(si));
        si.cb = sizeof(si);

        if (attr->detached) {
            si.dwFlags |= STARTF_USESHOWWINDOW; u
            si.wShowWindow = SW_HIDE;
        }

        if ((attr->child_in && attr->child_in->filehand)
            || (attr->child_out && attr->child_out->filehand)
            || (attr->child_err && attr->child_err->filehand))
        {
            si.dwFlags |= STARTF_USESTDHANDLES;
            si.hStdInput = (attr->child_in)
                                ? attr->child_in->filehand   v
                                : INVALID_HANDLE_VALUE;
```

```
            si.hStdOutput = (attr->child_out)
                                ? attr->child_out->filehand
                                : INVALID_HANDLE_VALUE;
            si.hStdError = (attr->child_err)
                                ? attr->child_err->filehand
                                : INVALID_HANDLE_VALUE;
        }
        rv = CreateProcessA(progname, cmdline, /* Command line */
                            NULL, NULL,        /* Proc & thread security attributes
*/
                            TRUE, w              /* Inherit handles */
                            dwCreationFlags,   /* Creation flags */
                            pEnvBlock,         /* Environment block */
                            attr->currdir,     /* Current directory name */
                            &si, &pi);
    }
    if (!rv)
        return apr_get_os_error();

    new->hproc = pi.hProcess;
    new->pid = pi.dwProcessId;

    if (attr->child_in) {
        apr_file_close(attr->child_in); x
    }
    if (attr->child_out) {
        apr_file_close(attr->child_out);
    }
    if (attr->child_err) {
        apr_file_close(attr->child_err);
    }

    CloseHandle(pi.hThread);
    return APR_SUCCESS;
```

与Unix下的操作相似，创建子进程的最后一个任务就是创建父进程和子进程之间的通信管道。不过由于Unix和Window中产生进程以及进程执行的机制不同，而导致通信管道的建立也存在差异。

我们回忆一下，在Unix 中，起始的时候主进程中连同管道描述符，一共拥有九个描述符，而子进程从父进程fork之后将继承所有九个描述符。由于fork之后，子进程和父进程同时 可以执行，因此对于它们来说，可以在各自的代码中进行重定向以及关闭多余的描述符，流程可以用下图描述：



但是对于 Window 而言，则并没有 fork 这样的机制。Window 中调用 CreateProcess 创建进 程之后，尽管进程也可以运行，但是与 Unix 相比，父进程并无法在自己的代码中过多的进行控制，如果想控制，只能由子进程本身去完成。在 Unix 下，管道 的建立由父进程和子进程协作创建，而在 Window 中，更多的则必须由父进程完成。对于 Window 下的父进程，创建管道包括下面的三个步骤：

1)、默认情况下，子进程将继承父进程中所有的句柄。不过有些句柄对于子进程并不需要，比如 parent_XXX，只用于父进程，因此它就没有必要被子进程继承。因此父进程在创建管道的时候就必须指定所有 parent_xxx 不被子进程继承。不 需要继承的描述符在函数 apr_create_nt_pipe 中由函数 apr_file_inherit_unset 指定，比如父进程在创建管道 child_in-parent_in 的时候同时指定 parent_in 不被子进程进程，代码如下：

```
if (in) {
    stat = apr_create_nt_pipe(&attr->child_in, &attr->parent_in, in,
                              attr->pool);
    if (stat == APR_SUCCESS)
        stat = apr_file_inherit_unset(attr->parent_in);
}
```

因此当子进程创建后，实际的描述符如下图的(b)所示，其内部只有六个描述符。

2)、设置父进程的继承标志。

子进程默认情况下并不会继承父进程中的句柄。为了允许继承，父进程必须设置进程标志。通过两种方法可以设置继承 标志，或者在创建句柄的时候设置 SECURITY_ATTRIBUTES 参数中的 bInheritHandle 成员为 TRUE，或者在使用 CreateProcess 创建子进程的时候设置函数的 bInheritHandles 参数为 TRUE，如 w 所示。

3)、子进程中描述符重定向

正如前面所分析的，子进程中不允许直接从控制台接受输入或者进行输出，而必须通过父进程来完成这些。因此子进程 的标准输入，标准输出，以及标准错误都必须重定向到 child_xxx 中。在 CreateProcess 创建过程中，通过设置 STARUUPINFO 参数 可以实现子进程的重定向。不过子进程重定向必须具备下面的几个条件：

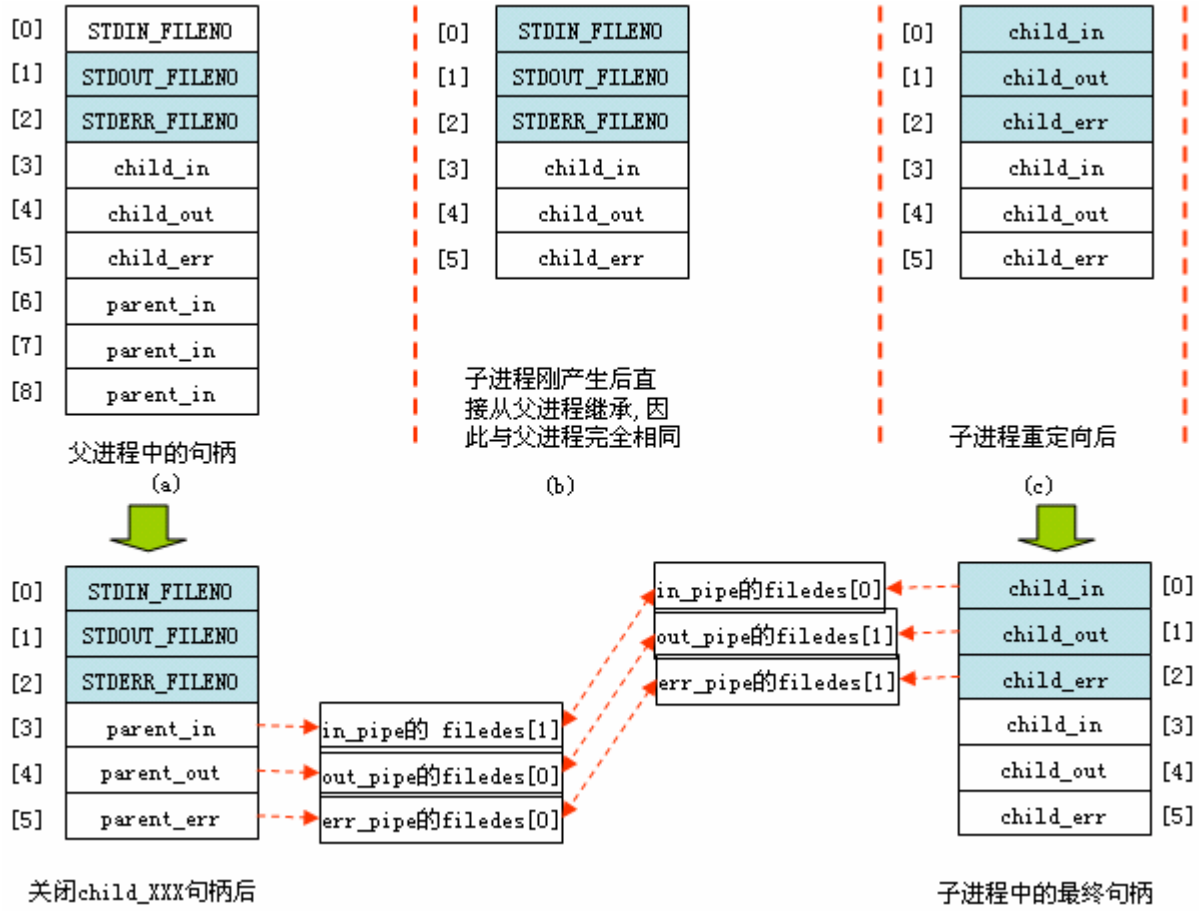
■ STARTUPINFOA 结构中的 dwFlags 必须设置 STARTF_USESTDHANDLES 标志位。该标志位的设立允许进程将它的标准输入、标准 输出以及标准错误设备用该结构中的指定的输入 hStdInput、输出 hStdOutput 以及错误设备 hStdError 替换，从而实现重定向。如果该 标志位不设定，hStdInput，hStdOutput 和 hStdError 将被忽略。

■ CreateProcess 函数中的 fInheritHandles 参数必须设置为 TRUE。

重定向代码如 v 所示。

4)、父进程关闭多余的 child_xxx 描述符。

最终在创建管道的过程中，父进程和子进程中的描述符的变化如下图所示。



上面的描述只是针对 Window 9X 的 ASCII 版本的操作系统，对于 Unicode 版本的操作系统，比如 window NT，Window 2003 等处理则存在一些差异。ASCII 和 Unicode 版本对进程创建的差异包括在下面两个方面：

1)、环境变量块的差异

2)、进程路径的差异

3)、进程在安全性方面的考虑。

首先我们看第一个差异：环境变量块的差异。

如前所述，使用 CreateProcess 的过程中可能需要传递环境变量。环境变量块中既允许包含 ANSI 字符，又允许包含 Unicode 字符。对于 Unicode 的操 作系统，环境变量块总是 Unicode 编码的，但是如果要 CreateProcess 将传递的环境变量当 Unicode 编码处理，则 CreateProcess 的 dwCreationFlags 标志必须包含 CREATE_UNICODE_ENVIRONMENT，否则即使环境变量块是 Unicode 编码，也会被视为 ANSI 处理。

对于 ANSI 编码环境变量 块，它总是以两个’\0’作为结束符：一个作为最后一个字符串的结束符，另一个作为整个环境变量块的结束符。而由于 Unicode 是双字节编码，因此 Unicode 编码的环境变量块则以四个’\0’做为结束标志：两个’\0’作为最后一个字符的结束符，另外两个则作为整个环境变量块的结束符。

与 ANSI 中使用 char 定义一个字符类似，Unicode 版本则用 apr_wchar_t 定义一个双字节字符，它的原始定义为 apr_uint16_t。Unicode 存在两种编码方式：UTF 或者 UCS。UTF-8 通常使用进行网络传输，比如网页的传输，URL 的传输，路径的传输。主要的原因就是对于本地编码为 Unicode 的系统，由于网络传输以字节作为单位，因此如果传输中某个字节为 0 的话，将会干

扰正常传输。这在Window中页不例外，Window中的环境变量块和路径名称都是基于UTF-8编码，因此必须将它们转换为本地的Unicode编码。转换由apr_conv_utf8_to_ucs2函数完成。反之，当使用路径的时候，则必须使用apr_conv_ucs2_to_utf8将本地Unicode编码转换为UTF-8。
环境变量块的处理如下所示：

```
#if APR_HAS_UNICODE_FS
        IF_WIN_OS_IS_UNICODE
        {
            apr_wchar_t *pNext;
            pEnvBlock = (char *)apr_palloc(pool, iEnvBlockLen * 2);
            dwCreationFlags |= CREATE_UNICODE_ENVIRONMENT;

            i = 0;
            pNext = (apr_wchar_t*)pEnvBlock;
            while (env[i]) {
                apr_size_t in = strlen(env[i]) + 1;
                if ((rv = apr_conv_utf8_to_ucs2(env[i], &in,
                                                pNext, &iEnvBlockLen))
                        != APR_SUCCESS) {
                    if (attr->errfn) {
                        ……
                    }
                    return rv;
                }
                pNext = wcschr(pNext, L'\0') + 1;
                i++;
            }
             if (!i)
                *(pNext++) = L'\0';
             *pNext = L'\0';
        }
#endif /* APR_HAS_UNICODE_FS */
```

路径的处理过程与之类似，此处不在赘述。
我们现在来看ANSI和Unicode版本在进程安全性方面的差异。Unicode版本的进程创建代码如下所示：

```
        if (attr->user_token) {
            si.lpDesktop = L"Winsta0\\Default";                u

            if (!ImpersonateLoggedOnUser(attr->user_token)) {
                rv = apr_get_os_error();
                CloseHandle(attr->user_token);           v
                attr->user_token = NULL;
                return rv;
            }
```

```
            rv = CreateProcessAsUserW(attr->user_token,
                                      wprg, wcmd,
                                      attr->sa,            W
                                      NULL,
                                      TRUE,
                                      dwCreationFlags,
                                      pEnvBlock,
                                      wcwd,
                                      &si, &pi);

            RevertToSelf();
        }
        else {
            rv = CreateProcessW(wprg, wcmd,          /* Executable & Command line */
                                NULL, NULL,          /* Proc & thread security
attributes */
                                TRUE,                /* Inherit handles */     x
                                dwCreationFlags,     /* Creation flags */
                                pEnvBlock,           /* Environment block */
                                wcwd,                /* Current directory name */
                                &si, &pi);
        }
```

如果指定了模拟用户，即 user_token 不为 NULL，那么进程将进行用户模拟。模拟又分两种，进程内使用 线程模拟和创建新进程模拟。这两种模拟方法在 Apache 中都有使用。只不过在该函数中是创建新进程，因此函数使用后一种方法。在进程内的通常使用 ImpersonateLoggedOnUser 函数模拟用户时，这个线程就是该模拟令牌代表的用户的身份，处理完成后使用 RevertToSelf 函数恢复自己的身份。如果该用户能够模拟成功，则立即使用该用户身份创建一个新的进程。这由函数 CreateProcessAsUser 完成，与 CreateProcess 相比， CreateProcessAsUser 函数多一个主要令牌的参数，这样启动的新进程就不是父进程的身份，而是 user_token 令牌代表的登录用户。

当然，如果不需要进行用户模拟，则只要在调用 CreateProcess 的时候将进程的安全属性设置为 NULL 就可以了。

Apache 中的网络地址处理

9.1 套接字地址

9.1.1 套接字地址

在了解 APR 中对 IP 地址的封装之前，我们首先看一下通常情况下对 IP 地址的使用情况。下面的代码掩饰了简单的服务器端套接字的地址初始化过程：

```
struct sockaddr_in server_addr; /* 本机地址信息 */
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(SERVPORT);
server_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(server_addr.sin_zero),8);
```

……

```
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size);
```

Socket API 中提供了三种类型的地址：sockaddr，sockaddr_in 和 sockaddr_un。sockaddr 是通用的套接字结 构，sockaddr_in 为 Internet 协议族的地址描述结构，sockaddr_un 则是 Unix 协议组的地址描述结构。sockaddr_in 结构中的 sa_family 决定是 sockaddr_in 还是 sockaddr_un。

如果直接使用 Socket API 提供的地址结构，则至少存在下面的几个问题：

1、在网络应用程序中，对于 internet 地址，如上面的程序代码所示，通常总是使用 sockaddr_in 描述，而在一些 Socket API 函数中则使用 sockaddr 作为套接字地址，因此在使用的时候必须将 sockaddr_in 强制转换为 sockaddr 类型，这是一个麻烦而且容 易出错的地方。

2、sockaddr_in 也不是一个特别容易理解的数据结构。通常情况下，sin_family 和 sin_port 相对容易记忆，而套接字地址 sin_addr.s_addr 则未必。套接字的这种结构对一般人而言无疑是一种噩梦。

3、另外一个问题则是 Ipv6 的地址问题。目前，Apache 已经开始同时支持 Ipv4 和 Ipv6 两种类型的地址，如果用户需要支持 Ipv6，则还必须使用 Ipv6 对应的地址数据结构。

对于一个良好的类库，不管是 Ipv4 还是 Ipv6 协议，都必须提供同样的接口，这种接口必须简单易懂，同时必须尽可能的隐藏内部的细节，比如对于 sin_addr.s_addr 无非暴露给用户。

基于上面的分析，APR 中只使用一种数据结构 apr_sockaddr_t 来描述 IP 地址，该结构定义在文件 apr_network_io.h 中：

```
struct apr_sockaddr_t {
    apr_pool_t *pool;
/*第一部分*/
    char *hostname;
    char *servname;
/*第二部分*/
    apr_port_t port;
    apr_int32_t family;
    union {
        struct sockaddr_in sin;
#if APR_HAVE_IPV6
        struct sockaddr_in6 sin6;
#endif
#if APR_HAVE_SA_STORAGE
        struct sockaddr_storage sas;
#endif
    } sa;
/*第三部分*/
    apr_socklen_t salen;
    int ipaddr_len;
    int addr_str_len;
    void *ipaddr_ptr;
    apr_sockaddr_t *next;
};
```

该结构描述了 socket 地址的三部分的信息内容：
第一部分：
Hostname 是该地址对应的主机名称，而 servname 则是对应端口的服务名称，比如 80 对应的名称为 ”www”，21 端口对应的 servname 则是”FTP”。如果某个端口比如 9889 并没有对应某个众所皆知的服务，那么 servname 则直接是端口 的字符串描述。
第二部分：
该部分则对应的是 sockaddr 结构中的内容，port 是端口，family 则是地址协议族类型，包括 AF_INET，AF_UNIX 等。sa 则为联合类型，用以描述对应的套接字地址，或者是 Ipv4 类型，或者是 Ipv6 类型，两者只能居其一。
第三部分：
这部分主要是一些与套接字地址相关的附加信息。Salen 是当前套接字地址的长度，通常情况下它的值为 sizeof(struct sockaddr_in)，对于 IPV6，则是 sizeof(struct sockaddr_in6)；ipiaddr_len 则是对应得 IP 地址结构的长度，对于 Ipv4 总是 sizeof(struct in_addr)，而对于 Ipv6，则是 sizeof(struct in6_addr)；addr_str_len 则是 IP 地址缓冲的长度，对于 Ipv4，该值为 14，而对于 IPV6，则是 46。这三个地址的含义完全不 同。
Ipaddr_ptr 指针指向 sockaddr 结构中的 IP 地址结构，通常情况下，它的初始化使用下面的代码：

```
apr_socketaddr_t addr;
addr->ipaddr_ptr = &(addr->sa.sin.sin_addr);
```

对于一些服务器而言，可能会使用多个 IP 地址。这些 IP 地址之间通过 next 指针形成单链表结构。
从 next 可以看出各个 socket 地址之间可以形成链表。

9.1.2 子网掩码结构

与此同时，APR 中也定义了数据结构 apr_ipsubnet_t 来描述 IP 地址掩码，当然由于 IP 地址分为 Ipv4 和 Ipv6，因此掩码描述也可以分为两种，apr_ipsubnet_t 结构定义在文件 apr_sockaddr.c 中，属于内部数据结构，具体如下：

```
struct apr_ipsubnet_t {
    int family;
#if APR_HAVE_IPV6
    apr_uint32_t sub[4]; /* big enough for IPv4 and IPv6 addresses */
    apr_uint32_t mask[4];
#else
    apr_uint32_t sub[1];
    apr_uint32_t mask[1];
#endif
};
```

family 是当前掩码所属于的地址族，APR_INET 表示 Ipv4，而 APR_INET6 则表示 Ipv6。
对于 Ipv4 而言，该结构演变为如下：

```
struct apr_ipsubnet_t {
    int family;
    apr_uint32_t sub[1];
    apr_uint32_t mask[1];
};
```

而对于 Ipv6，则该结构可以演变为如下：

```
struct apr_ipsubnet_t {
    int family;
    apr_uint32_t sub[4]; /* big enough for IPv4 and IPv6 addresses */
    apr_uint32_t mask[4];
};
```

9.1.3 Socket 地址处理接口

为了处理 Socket 地址，APR 中提供了四个操作接口，这些接口定义在 apr_network_io.h 中，而实现则 sockaddr.c 中。这四个接口分别是：

9.1.3.1 地址获取

由于 APR 中仅仅使用 apr_sockaddr_t 结构描述套接字地址，因此其余的各类描述信息最终都要转换为该结构，APR 中提供 apr_sockaddr_info_get 函数实现该功能：

```
APR_DECLARE(apr_status_t) apr_sockaddr_info_get(apr_sockaddr_t **sa,
                                                const char *hostname,
                                                apr_int32_t family,
                                                apr_port_t port,
                                                apr_int32_t flags,
                                                apr_pool_t *p);
```

该函数允许从主机名 hostname，地址协议族 family 和端口 port 创建新的 apr_sockaddr_t 地址，并由 sa 返回。

hostname 参数允许是实际的主机名称，或者也可以是字符串类型的 IP 地址，比如”127.0.0.1”，甚至可以是 NULL，此时默认的地址是”0.0.0.0”。

family 的值可以是 AF_INET，AF_UNIX 等系统定义类型，也可以是 APR_UNSPEC 类型，此时，地址协议族由系统决定。

flags 参数用以指定 Ipv4 和 Ipv6 处理的 优先级，它的取值包括两种：APR_IPV4_ADDR_OK 和 APR_IPV6_ADDR_OK。这两个标志并不是在所有的情况下都有效，这可以从函数的实现中看出它的用法：

```
{
    apr_int32_t masked;
    *sa = NULL;

    if ((masked = flags & (APR_IPV4_ADDR_OK | APR_IPV6_ADDR_OK))) {
        if (!hostname ||
            family != APR_UNSPEC ||
            masked == (APR_IPV4_ADDR_OK | APR_IPV6_ADDR_OK)) {
            return APR_EINVAL;u
        }
#if !APR_HAVE_IPV6
        if (flags & APR_IPV6_ADDR_OK) {
            return APR_ENOTIMPL;
        }
#endif
```

```
    }

#if !APR_HAVE_IPV6
    if (family == APR_UNSPEC) {
        family = APR_INET; v
    }
#endif

    return find_addresses(sa, hostname, family, port, flags, p); w
}
```

从实现代码可以看出，函数的内部实际的地址转换过程是由函数 find_address 完成的。不过在调用 find_address 之前，函数进行了相关检查和预处理，这些检查和预处理包括：

1、APR_IPV4_ADDR_OK 标记只有在 hostname 为 NULL，同时 family 为 APR_UNSPEC 的时候才会有效，而 APR_IPV6_ADDR_OK 和 APR_IPV4_ADDR_OK 是相互排斥的，一旦定义了 APR_IPV4_ADDR_OK，就不能使用 APR_IPV6_ADDR_OK，反之亦然。只有在 hostname 为 NULL，同时 family 为 APR_UNSPEC 并且没有定义 APR_IPV4_ADDR_OK 的时候 APR_IPV6_ADDR_OK 才会有效。

2、如果操作系统平台并不支持 IPV6，同时并没有限定获取的地址族，那么此时将默认为 IPV6。如果指定必须获取 IPV6 的地址信息，但系统并不提供支持，此时返回 APR_EINVAL。

一般情况下，在 IPV4 中从主机名到网络地址的解析可以通过 gethostbyname()函数完成，不过该 API 不允许调用者指定所需地址类型的任何信息，这意味着它仅返回包含 IPV4 地址的信息，对于目前新的 IPV6 则无能为力。一些平台中为了支持 IPV6 地址的解析，提供了新的地址解析函数 getaddrinfo()以及新的地址描述结构 struct addrinfo。APR 中通过宏 HAVE_GETADDRINFO 判断是否支持 IPV6 地址的解析。目前 Window 2000/XP 以上的操作系统都能支持新特性。为此 APR 中根据系统平台的特性采取不同的方法完成地址解析。

首先我们来看支持 IPV6 地址解析平台下的实现代码，find_address 函数的实现如下：

```
static apr_status_t find_addresses(apr_sockaddr_t **sa,
                                   const char *hostname, apr_int32_t family,
                                   apr_port_t port, apr_int32_t flags,
                                   apr_pool_t *p)
{
    if (flags & APR_IPV4_ADDR_OK) {
        apr_status_t error = call_resolver(sa, hostname, AF_INET, port, flags, p);
#if APR_HAVE_IPV6
        if (error) {
            family = AF_INET6; /* try again */ u
        }
        else
#endif
        return error;
    }

#if APR_HAVE_IPV6
    else if (flags & APR_IPV6_ADDR_OK) {
        apr_status_t error = call_resolver(sa, hostname, AF_INET6, port, flags, p);
```

```
        if (error) {                                v
            family = AF_INET; /* try again */
        }
        else {
            return APR_SUCCESS;
        }
    }
#endif

    return call_resolver(sa, hostname, family, port, flags, p);    w
}
```

从上面的代码可以清晰的看到APR_IPV4_ADDR_OK和APR_IPV6_ADDR_OK的含义：对于 前者，函数内部首先查询对应主机的IPV4地址，只有在IPV4查询失败的时候才会继续查询IPV6地址；而后者则与之相反，对于给定的主机名称，首先查 询IPV6地址，只有在查询失败的时候才会查询IPV4。因此APR_IPV4_ADDR_OK和APR_IPV6_ADDR_OK决定了查询的优先性， 任何时候一旦查询成功都不会继续查询另外协议地址，即使被查询主机具有该协议地址。

查询的核心代码封装在内部函数call_resolve中，该函数的参数和apr_sockaddr_info_get函数的参数完全相同且对应，call_resolve中的宏处理比较的多，因此我们将分开描述：

```
static apr_status_t call_resolver(apr_sockaddr_t **sa,
                                  const char *hostname, apr_int32_t family,
                                  apr_port_t port, apr_int32_t flags,
                                  apr_pool_t *p)
{
    struct addrinfo hints, *ai, *ai_list;
    apr_sockaddr_t *prev_sa;
    int error;
    char *servname = NULL;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = family;
    hints.ai_socktype = SOCK_STREAM;
#ifdef HAVE_GAI_ADDRCONFIG
    if (family == APR_UNSPEC) {
        hints.ai_flags = AI_ADDRCONFIG;
    }
#endif
```

在了解上面的代码之前我们首先简要的了解一些getaddrinfo函数的用法，该函数定义如下：

```
int getaddrinfo(const char *hostname, const char *service, const struct addinfo
*hints,struct addrinfo **result);
```

hostname是需要进行地址解析的主机名称或者是二进制的地址串(IPV4的点分十进制或者Ipv6的十六进制数串)，service则是一个服务名或者是一个十进制的端口号数串。其中hints是addfinfo结构，该结构定义如下：

```
struct addrinfo {
```

```
int ai_flags;                                   /* AI_PASSIVE, AI_CANONNAME,
AI_NUMERICHOST */
int ai_family;                                  /* PF_xxx */
int ai_socktype;                               /* SOCK_xxx */
int ai_protocol;                                 /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
size_t ai_addrlen;                            /* length of ai_addr */
char *ai_canonname;                        /* canonical name for nodename */
struct sockaddr *ai_addr;                  /* binary address */
struct addrinfo *ai_next;                     /* next structure in linked list */
};
```

hints 参数可以是一个空置针，也可以是一个指向某个 addrinfo 结构的指针，调用者在该结构中填入关于 期望返回的信息类型的暗示，这些暗示将控制内部的转换细节。比如，如果指定的服务器既支持 TCP，也支持 UDP，那么调用者可以把 hints 结构中的 ai_socktype 成员设置为 SOCK_DGRAM，使得返回的仅仅是适用于数据报套接口的信息。

hints 结构中调用者可以设置的成员包括 ai_flags，ai_family，ai_socktype 和 ai_protocol。其中，ai_flags 成员可用的标志值及含义如下：

标志名称	标志含义
AI_PASSIVE	套接口将用于被动打开
AI_CANONNAME	告知 getaddrinfo 函数返回主机的规范名称
AI_NUMERICHOST	防止任何类型的名字到地址的映射；hostname 必须是一个地址串
AI_NUMERICSERV	防止任何类型的名字到服务的映射，service 参数必须是一个十进制端口号数串
AI_V4MAPPED	如果同时指定 ai_family 成员的值为 AF_INET6 和 AF_INET，那么如果没有可用的 AAAA 记录就返回与 A 记录对应得 Ipv4 映射的 IPV6 地址
AI_ALL	如果同时指定 AI_V4MAPPED 标志，那么除了返回与 AAAA 对应得 IPV6 地址之外，还会返回与 A 记录对应的 IPV4 映射的 Ipv6 地址。
AI_ADDRCONFIG	按照所在主机的配置选择返回的地址类型，也就是只查找与所在主机回馈接口以外的网络接口配置的 IP 地址版本一直的地址。只有当本地系统中配置仅仅配置了 IPV4 地址才会将主机名称转换位 IPV4 地址；同样只有当本地系统中仅配置了 IPV6 地址的时候才会返回 IPV6 地址。Loopback 地址并不在这种限制之中。

ai_family 参数指定调用者期待返回的套接口地址结构的类型。它的值包括三 种：AF_INET，AF_INET6 和 AF_UNSPEC。如果指定 AF_INET，那么函数九不能返回任何 IPV6 相关的地址信息；如果仅指定了 AF_INET6，则就不能返回任何 IPV4 地址信息。AF_UNSPEC 则意味着函数返回的是适用于指定主机名和服务名且适合任何协议族的地址。如果某 个主机既有 AAAA 记录(IPV6)地址，同时又有 A 记录(IPV4)地址，那么 AAAA 记录将作为 sockaddr_in6 结构返回，而 A 记录则作为 sockaddr_in 结构返回。

```
    if(hostname == NULL) {
#ifdef AI_PASSIVE
        hints.ai_flags |= AI_PASSIVE;
#endif
#ifdef OSF1
        hostname = family == AF_INET6 ? "::" : "0.0.0.0";
        servname = NULL;
#ifdef AI_NUMERICHOST
```

```
        hints.ai_flags |= AI_NUMERICHOST;
#endif
#else
#ifdef _AIX
        if (!port) {
            servname = "1";
        }
        else
#endif /* _AIX */
        servname = apr_itoa(p, port);
#endif /* OSF1 */
    }

#ifdef HAVE_GAI_ADDRCONFIG
    if (error == EAI_BADFLAGS && family == APR_UNSPEC) {
        hints.ai_flags = 0;
        error = getaddrinfo(hostname, servname, &hints, &ai_list);
    }
#endif
    if (error) {
#ifndef WIN32
        if (error == EAI_SYSTEM) {
            return errno;
        }
        else
#endif
        {
#if defined(NEGATIVE_EAI)
            error = -error;
#endif
            return error + APR_OS_START_EAIERR;
        }
    }
```

9.1.3.2 主机名获取

apr_sockaddr_info_get 函数用以完成从主机名到网络地址的转换，而 APR 中提供的 apr_getnameinfo 则可以实现从网络地址到主机名的转换，该函数定义如下：

```
APR_DECLARE(apr_status_t) apr_getnameinfo(char **hostname, apr_sockaddr_t *sa,
apr_int32_t flags);
```

参数 sa 指定需要转换的网络地址，转换后的主机名由 hostname 返回。fags 是标志位，用以控制内部的转换过程。

```
{
#if defined(HAVE_GETNAMEINFO)
    int rc;
```

```
#if defined(NI_MAXHOST)
    char tmphostname[NI_MAXHOST];
#else
    char tmphostname[256];
#endif

    SET_H_ERRNO(0);

#if APR_HAVE_IPV6
    if (sockaddr->family == AF_INET6 &&
        IN6_IS_ADDR_V4MAPPED(&sockaddr->sa.sin6.sin6_addr)) {
        struct sockaddr_in tmpsa;
        tmpsa.sin_family = AF_INET;
        tmpsa.sin_addr.s_addr = ((apr_uint32_t *)sockaddr->ipaddr_ptr)[3];
#ifdef SIN6_LEN
        tmpsa.sin_len = sizeof(tmpsa);
#endif
        rc = getnameinfo((const struct sockaddr *)&tmpsa, sizeof(tmpsa),
                         tmphostname, sizeof(tmphostname), NULL, 0,
                         flags != 0 ? flags : NI_NAMEREQD);
    }
    else
#endif
#endif
    rc = getnameinfo((const struct sockaddr *)&sockaddr->sa, sockaddr->salen,
                     tmphostname, sizeof(tmphostname), NULL, 0,
                     flags != 0 ? flags : NI_NAMEREQD);
```

在函数的内部实现从地址到主机名称的解析是由函数 getnameinfo 完成的，该函数是 getaddrinfo 的互补函数，它以一个套接口地址为参数，返回描述其中的主机的一个字符串和描述其中的服务的另一个字符串。另外该函数以协议无关的 方式提供这些信息，调用者必须关心存放在套接口地址结构中的协议地址的类型，这些由函数自行处理。

需要转换的地址到底是 IPv4 还是 IPv6，这由地址结构中的 family 参数决定。尽管理想中的做法是将 apr_getnameinfo()中的参数直接传递给 getnameinfo()函数，但是在一些平台上还是会出现一些问题。

MacOS X Panther has a lousy getnameinfo() implementation that doesn't fill the buffer when no DNS entry is found for a host and a numerical result wasn't explicitely asked. As a result, Pure-FTPd didn't even start on Panther (saying "bad IP address") . We now check for EAI_NONAME if available and we retry with NI_NUMERICHOST if this is what getnameinfo() returns. Thanks to Yann Bizeul for his valuable help on this issue. Will research it more and see if I can come up with a patch (I am NOT good at C!)

在一些操作系统中，比如老版本的 Mac OS X，如果 Ipv6 地址是由 Ipv4 地址映射的结果，那么该地址在传递给 getnameinfo 函数的时候将会产生错误，这是系统本身实现的 BUG。因此对 于这种情况，解决的方法就是将这种 Ipv6 地址重新转换为 Ipv4 地址。Ipv6 地址是否是由 Ipv4 地

址进行映射而成，通过宏 IN6_IS_ADDR_V4MAPPED 可以实现检测。IPV4 到 IPV6 地址的映射可以用下图描述：

Ipv4 地址通过在其十六进制前面添加前导零的方式映射为 IPV6 地址。反之如果一个 IPV6 地址是由 IPV4 地址映射而成，则只要剔除前面的前导零即可，剔除后的地址通常为((apr_uint32_t *)sockaddr->ipaddr_ptr)[3];一旦获取了实际的 IPV4 地址，则可以将其传递给 getnameinfo 函数。

对于其余的 IP 地址，包括普通的 Ipv4 地址，非 Ipv4 映射的 Ipv6 地址，由于不存在 BUG，因此可以直接调用 getnameinfo。

getnameinfo 函数原型如下：

```
Int getnameinfo(const struct sockaddr* sockaddr, socklen_t addrlen,
                    char *host, socklen_t hostlen,
                    char *serv, socklen_t servlen, int flag);
```

函数的前面几个参数都非常容易理解，只有最后一个参数 flag，它用于控制 getnameinfo 的操作，它允许的值如下面所列：

NI_DGRAM

当知道处理的是数据报套接口的时候，调用者应该设置 NI_DGRAM 标志，因为在套接口地址结构中给出的仅仅是 IP 地址和端口号，getnameinfo 无法就此确定所用协议是 TCP 还是 UDP。比如端口 514，在 TCP 端口上提供 rsh 服务，而在 UDP 端口上则 提供 syslog 服务。

NI_NOFQDN

该标志导致返回的主机名称被截去第一个点号之后的内容。比如假设套接口结构中的 IP 地址为 912.168.42.2，那么不设置该标志返回的主机名为 aix.unpbook.com，那么如果设置了该标志后返回的主机名则为 aix。

NI_NUMERICHOST，NI_NUMERICSERV，NI_NUMERICSCOPE

该标志通知 getnameinfo 不要调用 DNS，而是以数值表达格式作为字符串返回 IP 地址；类似 的，NI_NUMERICSERV 标志指定以十进制数格式作为字符串返回端口号，以代替查找服务名；NI_NUMERICSCOPE 则指定以数值格式作为 字符串返回范围标识，以代替其名字

NI_NAMEREQD

该标志通知 getnameinfo 函数如果无法适用 DNS 反向解析出主机名，则直接返回一个错误。需要把客户的 IP 地址映射成主机名的那些服务器可以使用该特性。

如果 flag 没有指定，即为零，那么 NI_NAMEREQD 将是 Apache 中默认的标志项，如果不设置该标志，那么在反向解析失败的时候 getnameinfo 将返回一个数值地址字符串，显然这并不是 Apache 所需要的结果。

```
    if (rc != 0) {
        *hostname = NULL;
#ifndef WIN32
        if (rc == EAI_SYSTEM) {
            if (h_errno) { /* for broken implementations which set h_errno */
                return h_errno + APR_OS_START_SYSERR;
            }
            else { /* "normal" case */
                return errno + APR_OS_START_SYSERR;
            }
        }
        else
```

```
#endif
        {
#if defined(NEGATIVE_EAI)
            if (rc < 0) rc = -rc;w
#endif
            return rc + APR_OS_START_EAIERR; /* return the EAI_ error */
        }
    }
    *hostname = sockaddr->hostname = apr_pstrdup(sockaddr->pool, tmphostname);
    return APR_SUCCESS;
```

上面的代码是对 getnameinfo 发生错误时候的处理(rc==0 意味着成功，否则意味着转换失败)。此时 将需要返回的主机名称设置为 NULL。当 getnameinfo 发生错误的时候通常会返回 EAI_XXXX 的错误码，在所有这些错误码中比较特殊的就是 EAI_SYSTEM，它意味着同时在 errno 变量中有系统错误返回，而其余的 EAI_XXXX 错误并不会设置 errno 变量。

对于非 EAI_SYSTEM 错误码，APR 并不能直接返回。正如第一章所说，APR 中对于 apr_status_t 返回码有自己的布局和规则，因此这些错误码必须转换至 APR 返回码。EAI_XXXX 错误码的起始偏移是 APR_OS_START_EAIERR，因此返回值实际上是 rc+APR_OS_START_EAIERR。不过在一些平台上比如 glibc，为了防止 和 h_errno 的值冲突，系统将使用 EAI_XXXX 的负值， 这正是上面的代码 w 的原因。

上面的代码有一个假设前提，就是系统中必须提供 getnameinfo()函数。但是由于 getnameinfo()是比较新的一个函数，并不是每个操作系统平台都支持该函数。目前大部分 Ipv4 平台上不过都提供了 gethostbyaddr()函数，通过该函数也能完成从主机地址到主机名称的转换，不过该函数仅仅支持 Ipv4 协议，不支持 Ipv6 协议。具体的代码 如下所示：

```
#else
#if APR_HAS_THREADS && !defined(GETHOSTBYADDR_IS_THREAD_SAFE) && \
    defined(HAVE_GETHOSTBYADDR_R) && !defined(BEOS)
#ifdef GETHOSTBYNAME_R_HOSTENT_DATA
    struct hostent_data hd;
#else
    char tmp[GETHOSTBYNAME_BUFLEN];
#endif

    int hosterror;
    struct hostent hs, *hptr;

#if defined(GETHOSTBYNAME_R_HOSTENT_DATA)
    /* AIX, HP/UX, D/UX et alia */
    gethostbyaddr_r((char *)&sockaddr->sa.sin.sin_addr, u
                    sizeof(struct in_addr), AF_INET, &hs, &hd);
    hptr = &hs;
#else
#if defined(GETHOSTBYNAME_R_GLIBC2)
    /* Linux glibc2+ */
    gethostbyaddr_r((char *)&sockaddr->sa.sin.sin_addr, v
```

```
                    sizeof(struct in_addr), AF_INET,
                    &hs, tmp, GETHOSTBYNAME_BUFLEN - 1, &hptr, &hosterror);
#else
    /* Solaris, Irix et alia */
    hptr = gethostbyaddr_r((char *)&sockaddr->sa.sin.sin_addr, w
                           sizeof(struct in_addr), AF_INET,
                           &hs, tmp, GETHOSTBYNAME_BUFLEN, &hosterror);
#endif /* !defined(GETHOSTBYNAME_R_GLIBC2) */
    if (!hptr) {
        *hostname = NULL;
        return hosterror + APR_OS_START_SYSERR;
    }
#endif /* !defined(GETHOSTBYNAME_R_HOSTENT_DATA) */
#else
    struct hostent *hptr;
    hptr = gethostbyaddr((char *)&sockaddr->sa.sin.sin_addr, x
                         sizeof(struct in_addr), AF_INET);
#endif
    if (hptr) {
        *hostname = sockaddr->hostname = apr_pstrdup(sockaddr->pool, hptr->h_name);
        return APR_SUCCESS;
    }
    *hostname = NULL;
#if defined(WIN32)
    return apr_get_netos_error();
#elif defined(OS2)
    return h_errno;
#else
    return h_errno + APR_OS_START_SYSERR;
#endif
#endif
```

函数中众多的预定义让人眼花缭乱。不过最主要的预定义处理还在于对 gethostbyaddr()函数的调用。 从上面的代码中可以看出，gethostbyaddr 有一个函数变形 gethostbyaddr_r，而且不同平台下的 gethostbyaddr_t 函 数的参数也不相同，要了解详细的原因，必须了解一些函数可重入的概念。

所谓可重入函数是指一个可以被多个任务调用的函数，任务在调用时候不必担心数据会出错；

通常情况下下面的函数是不可重入的：

(1)、函数体内使用了静态的数据结构；

(2)、函数体内调用了 malloc()或者 free()函数；

(3)、函数体内调用了标准 I/O 函数。

通常情况下，在一个 UNIX 进程中发生重入问题的条件是：从主程序中和某个信号处理函数中同时调用某个不可重入 函数.。另外在多线程应用中也会出现函数重入的问题。不幸的是由于历史的原因，我们经常使用的 gethostbyaddr 也是一个不可重入的函数，因为它 们都返回指向同一

个静态结构的指针。关于gethostbyaddr的重入问题，《Unix网络编程 第一卷：套接口API》中文版第二版的第207页中有一段描述，摘抄如下：
不幸的是，重入问题比他表面看起来更要严重。首先，关于gethostbyname和gethostbyaddr的重入问题无标准可循。POSIX规范声明这两个函数不必是可重入的。Unix98只说这两个函数必须是线程安全的。
其次，关于_r函数也没有标准可循。Solaris 2.X，Digital Unix 4.0和HP-UX 10.30都提供了可重入版本的gethostbyaddr_r函数，不过它们的参数并不相同，不同版本的gethostbyaddr_r函数原型如下表所示：

操作系统平台	函数原型
solaris	struct hostent* gethostbyaddr_r(const char *addr, int len, int type, struct hostent *result, **char *buf, int buflen, int * h_errnop**);
AIX,HP-UX,Digital Unix	int gethostbyaddr_r(const char *addr, int len, int type, struct hostent *result, **struct hostent_data *buffer**);
Linux glibc2+	int gethostbyaddr_r(const char *addr,int len, int type,struct hostent *result, **char* buf, int buflen, struct hostent *hs, int* h_errnop**);

大部分gethostbyaddr_r函数的前四个参数都相同，第一个是需要转换的地址；第二个地址的字节大 小，用sizeof(struct in_addr)表示；第三个是需要转换地址的协议族，或者是AF_INET，或者是AF_INET6；第四个则是描述主机的hostent结构。区别通 常在后几个参数：
对于Solaris，Irix等操作系统而言，后面还需要三个额外的参数，buf是由调用者分配的并且大小为 buflen的缓冲区，该缓冲区用于存放规范主机名称，别名指针数组，各个别名字符串，地址指针数组以及各个实际地址。如果初出错，错误码通过 h_errnop指针返回，注意不是我们通常所说的h_errno返回。
对于AIX，HP-UX，Digital Unix等平台而言，后面的三个参数则被组合为一个新的数据结构hostent_data，指向该结构的指针构成本函数的第三个和最后一个参数。 Apache中默认的缓冲区大小为GETHOSTBYNAME_BUFLEN，即512字节。
对于Linux glibc2+而言，gethostbyaddr_r的参数与前两者又存在一定的差异，它共计有八个参数，与Solaris平台相比多了struct hostent* hs参数。
如果操作系统平台不支持可重入的gethostaddr_r函数，那么只能使用不可重入的gethostbyaddr函数，如x所示。
返回的主机名称保存在hostent结构中，如果查询成功，从hostname参数中返回即可。

9.1.3.3 IP地址解析

```
APR_DECLARE(apr_status_t) apr_parse_addr_port(char **addr,
                                              char **scope_id,
                                              apr_port_t *port,
                                              const char *str,
```

apr_pool_t *p);

9.1.3.4 子网掩码

Apache 中多任务并发处理机制研究(1)

6.1 多进程并发处理概述

6.1.1 概述

第五章中我们讨论 Apache 主程序的时候，当主程序调用了函数 ap_mpm_run 之后，整个主程序就算结束了。那么函数进入到 ap_mpm_run 之后它干什么去了呢？
如果让我们来写服务器程序的话，按照正常的思路，通常主程序在进行了必要的准备工作后会调用诸如 fork 之类的函数产生一个新的进程或者线程，然后由子进程进行并发处理。事实上，Apache 尽管是一个先进的服务器，但是它也不能脱离窠臼。
主进程一旦调用 ap_mpm_run 之后，它就进入多进程并发处理状态。为了并发处理客户端请求，Apache 或者产生多个进程，或者产生多个线程，或者产生多个进程，每个进程又产生一定数目的线程等等。
Apache HTTP 服务器从一开始就被设计为一个强大、灵活的能够在多种平台上及不同的环境下工作的服务器。 不同的平台和不同的环境经常产生不同的需求，或是会为了达到同样的最佳效果而采用不同的方法。当然，Apache 中提供了多种多进程并发模型，比如 Prefork，Window NT，Event，perchild 等等。并且为了方便移植和替换，Apache 将这些多进程并发处理模型设计成模块。Apache 凭借它的模块设计很好的适应了大量不同的环境。这一设计使得网站管理员能够在编译时和运行时凭借载入不同的模块来决定服务器的不同附加功能。
Apache 2.0 将这种模块式设计延伸到 web 服务器的基础功能上。这个发布版本带有多道处理模块的选择以处理网络端口绑定、接受请求并指派子进程来处理这些请求。
将模块设计延伸到这一层面主要有以下两大好处：
Apache 可以更简洁、更有效地支持各种操作系统。 尤其是在 mpm_winnt 使用本地网络特性以代替 Apache 1.3 中使用的 POSIX 层后， Windows 版本的 Apache 现在有了更好的性能。 这个优势借助特定的 MPM 同样延伸到其他各种操作系统。
服务器可以为某些特定的站点进行自定义。比如， 需要更好缩扩性的站点可以选择象 worker 这样线程化的 MPM， 而需要更好的稳定性和兼容性以适应一些旧的软件的站点可以用 prefork。 此外，象用不同的用户号(perchild)伺服不同的站点这样的特性也能提供了。
从用户层面来讲，MPMs 更像其他 Apache 模块。而主要的不同在于：不论何时，有且仅有一个 MPM 必须被载入到服务器中。现有的 MPM 列表可以在这里找到模块索引。
下表列出了不同操作系统下默认的 MPMs。如果你在编译时没有进行选择，这将是默认选择的 MPM。

BeOS	beos
Netware	mpm_netware
OS/2	mpmt_os2
Unix	prefork

Windows	mpm_winnt

在详细深入的描述各个 MPM 之前，我们有必要了解一下 MPM 中所使用到的公共数据结构主要包括两种：记分板和父子进程的通信管道。记分板类似于共享内存，主要用于父子进程之间进行数据交换，类似于白板。任何一方都可以将对方需要的信息写入到记分板上，同时任何一方也可以到记分板上获取需要的数据。

6.2 MPM 公共数据结构

6.2.1 记分板

6.2.1.1 记分板概述

Apache 的 MPM 中通常总是包含一个主服务进程以及若干个子进程，因此不可避免的存在主进程和子进程通信的问题。Apache 中采用了两种主要的通信方法：记分板和管道。
记分板就是一块共享内存块，同时可以被父进程和子进程访问，通过共享实现了父子之间的通信。尽管如此，但是记分板则更主要用于父进程对子进程进行控制。在 Apache 中 主进程的一个重要的职责就是控制空闲子进程的数目：如果空闲子进程过多，则父进程将终止一些子进程；如果空闲子进程太少，则父进程将创建一些新的空闲子进 程以备使用。因此，父进程必须随时能够知道子进程的数目以便进行调整。子进程把自己的状态信息忙碌或者空闲写入到记分板中，这样通过读取记分板，父进程就 可以知道子进程的数目了。
记分板的数据结构可以描述如下：

```
typedef struct {
    global_score *global;
    process_score *parent;
    worker_score **servers;
} scoreboard;
```

该结构定义在 scoreboard.h 中，由该数据结构可见，Apache 中的记分板可以记录三种类型的信息：全局信息、进程间共享信息以及线程间共享信息。
global_score 是记分板中描述全局信息的结构，通常这些信息是针对整个 Apache 服务器的，而不是针对某个进程或者某个线程的，该结构定义如下：

```
typedef struct {
    int  server_limit;
    int  thread_limit;
    ap_scoreboard_e sb_type;
    ap_generation_t running_generation;
    apr_time_t restart_time;
} global_score;
```

从该结构中我们可以看出，全局的共享信息包括下面的几个内容：server_limit 描述系统中所存在的服务进程的极限值，thread_limit 则是描述的线程的极限值。ap_scoreboard 是枚举类型，只有两个值 SB_NOT_SHARED 和 SB_SHARED，分别表示该记分板是否

进程间共享还是不共享。
ap_generation_t 的定义实际上是整数值：typedef int ap_generation_t。该值主要用于“平稳启动(graceful restart)”。 Apache 中允许在不终止 Apache 的情况下对 Apache 进行重新启动，这种启动称之为“平稳启动”。平稳启动的时候，主服务进程将退出，同时创建新的子进程。此时这些子进程由父进程创建，它们形成一个继承称此上的家族概念，只要是主进程产生的所有子进程都属于这个主进程家族，因此我们称它们称之为为新的“代(generation)”，在本书中我们统一用“家族”这个术语进行描述。只有子进程与父进程具有亲缘关系，它们才是一个家族。每一个进程在执行完任务之后都会检查它与当前的主进程是否属于同一个家族。如果属于，则继续等待处理下一个任务；否则其将退出。由于 Apache 在进行平稳启动的时候对于那些尚未结束的进程并不强行将其终止，而是让其继续执行，但是主进程必须退出重新启动。因此当新的主进程启动之后，这些残余的子进程显然已经跟它不是同一个家族，它们属于上一辈的。因此他们在执行完任务之后立即退出。某个主进程产生后它就产生一个唯一的家族号，用 running_generation 进行记录。该值永远不会重复。主进程的所有子进程将继承该家族号。running_generation 是识别其家族的唯一标记。如果子进程的 running_generation 与父进程相同，则说明本家族的进程尚存在；反之，如果不相同，则它们执行完后必须结束。这正应了一句古语：”覆巢之下无完卵”或者为”树倒猢狲散”阿。
restart_time 则记录了主服务器重新启动的时间。
进程间通信则可以使用 process_score 进行，其定义如下：

```
typedef struct process_score process_score;
struct process_score{
    pid_t pid;
    ap_generation_t generation;    /* generation of this child */
    ap_scoreboard_e sb_type;
    int quiescing;
};
```

通常情况下，父进程往该数据结构中写入数据，而子进程则从其中读取数据。其中 pid 是主进程的进程号；generation 则是当前主进程以及其产生的所有子进程的家族号。sb_type 的含义与 global_score 中的 sb_type 含义相同。
Quiescing 则
与 process_score 用于主进程和子进程通信不同，worker_score 则用于记录线程的运行信息，其定义如下：

```
typedef struct worker_score worker_score;
struct worker_score {
/*第一部分*/
    int thread_num;
#if APR_HAS_THREADS
    apr_os_thread_t tid;
#endif
    unsigned char status;
/*第二部分*/
    unsigned long access_count;
    apr_off_t     bytes_served;
    unsigned long my_access_count;
```

```
    apr_off_t     my_bytes_served;
    apr_off_t     conn_bytes;
    unsigned short conn_count;
/*第三部分*/
    apr_time_t start_time;
    apr_time_t stop_time;
#ifdef HAVE_TIMES
    struct tms times;
#endif
    apr_time_t last_used;
/*第四部分*/
    char client[32];              /* Keep 'em small... */
    char request[64];       /* We just want an idea... */
    char vhost[32];             /* What virtual host is being accessed? */
};
```

整个 worker_score 结构可以被分成四部分理解：

第一部分，主要描述线程的状态和识别信息

thread_num 是 Apache 识别该线程的唯一识别号，tid 则是该线程的线程号。两者是不同的概念：后者是由操作系统或者线程库分配，应用程序无法参与，而前者是 Apache 设定，跟操作系统无关，具体的含义也只有 Apache 本身理解。不过 thread_num 通常遵循下面的设定原则：

thread_num = 线程所在的进程的索引 * 每个进程允许产生的线程极限 + 线程在进程内的索引

status 则是当前线程的状态，它的状态种类与进程的状态种类相同，用 SERVER_XXX 常量进行识别。

第二部分，主要描述线程的状态和识别信息

第三部分，主要描述线程相关的时间信息

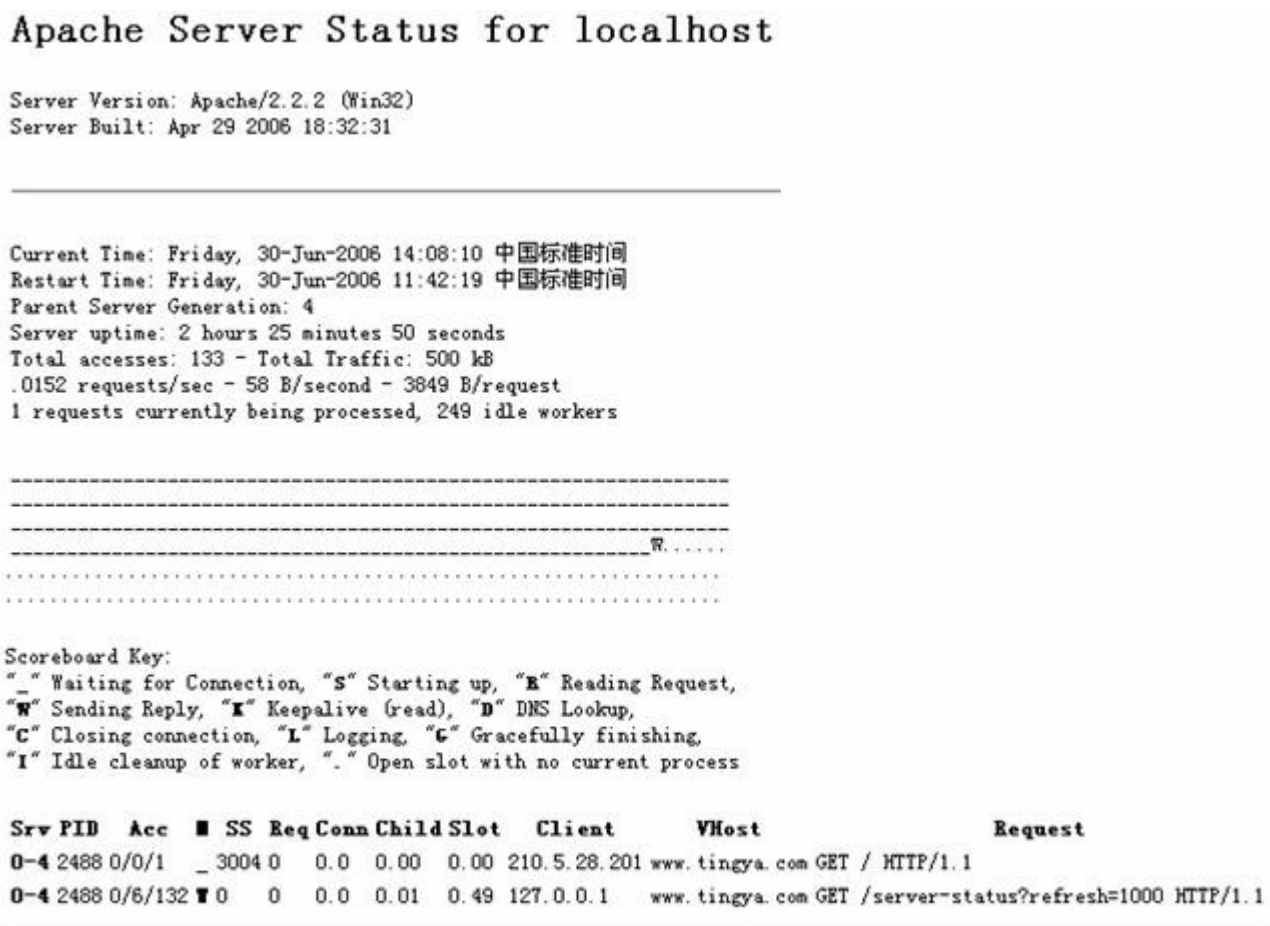
start_time 和 stop_time 分别是记录线程的启动和停止时间。last_used 则用于记录线程最后一次使用的时间。

第四部分，主要描述线程的状态和识别信息

该部分主要描述当前线程处理的请求连接上的相关信息。client 是请求客户端的主机名称或者是 IP 地址。request 则是客户端发送的请求行信息，比如”GET /server-status?refresh=100 HTTP/1.1”，而 vhost 则是当前请求所请求的虚拟主机名称，比如”www.myserver.com”。

从 worker_score 结构中可以看出，该结构中的记录的大部分信息并不是线程间通信而需要的。那么这些信息到底做什么用的呢？为什么 worker_score 结构中需要记录这些信息呢？为此我们必须了解 Apache 中的一个特殊的功能模块 mod_status。尽管这个模块要到第三卷才能详细介绍，但是我们还是提前描述。

为了时刻了解 Apache 的运行状态，一种方法就是直接在服务器上检测，另一种方法就是远程监控，通过 http://www.xxxx.com/server-status URI 在浏览器中显示服务器的信息：



每一个进程或者线程都将自身信息写入到记分板上，这样，mod_status 通过读取记分板，就可以知道各个线程的运行状态信息。当然，如果需要显示的信息越多，记分板上需要保存的信息也就越多，worker_score 结构也就需要扩展。

除此之外，Apache 中还定义了几个与记分板相关的全局变量，它们是记分板的核心变量：

(1)、AP_DECLARE_DATA extern scoreboard *ap_scoreboard_image;

Apache 使用使用该变量记录全局记分板，任何进程或者线程都可以通过 ap_scoreboard_image 直接访问记分板。

(2)、AP_DECLARE_DATA extern const char *ap_scoreboard_fname;

该全局变量描述了记分板的名称。

(3)、AP_DECLARE_DATA extern int ap_extended_status;

该全局变量描述了当前记分板的状态，

(4)、AP_DECLARE_DATA extern ap_generation_t volatile ap_my_generation;

该全局变量描述了当前 Apache 中的主进程的家族号，任何时候，主进程只要退出进行重新启动，ap_my_generation 都会跟着发生变化。该变量与其余的变量相比特殊的地方在于它被声明为 volatile 类型。

(5)、static apr_size_t scoreboard_size;

该变量记录整个记分板所占用的内存的大小。

在了解了记分板的数据结构之后，我们有必要了解一下记分板的内存组织结构，它的内存布局可以用下图进行描述：

6.1.1.2 记分板处理函数

从前一节的图片中我们看一看出，每个记分板都包括多个插槽，每一个插槽分别用于记录一个进程的相关信息，不过这种记录的进程信息相对非常的简单，仅仅包括进程的当前状态以及进程号。从记分板的角度而言，每一个进程可以处于 12 中不同的状态：

```
#define SERVER_DEAD 0                    /* 当前的进程执行完毕*/
#define SERVER_STARTING 1                /* 进程刚开始执行 */
#define SERVER_READY 2                   /* 进程已经准备就绪，正在等待客户端连接 */
#define SERVER_BUSY_READ 3               /* 进程正在读取客户端的请求*/
#define SERVER_BUSY_WRITE 4              /* 进程正在处理客户端的请求*/
#define SERVER_BUSY_KEEPALIVE 5          /* 进程在同一个活动连接上正在等待更多的请求*/
#define SERVER_BUSY_LOG 6                /* 进程正在进行日志操作*/
#define SERVER_BUSY_DNS 7                /* 进程正在查找主机名称 */
#define SERVER_CLOSING 8                 /* 进程正在关闭连接 */
#define SERVER_GRACEFUL 9                /* 进程正在平稳的完成请求 */
#define SERVER_IDLE_KILL 10              /* 进程正在清除空闲子进程. */
#define SERVER_NUM_STATUS 11             /* 进程的多个状态都被设置 */
```

进程总是从状态 SERVER_STARTING 开始，最后在 SERVER_DEAD 状态结束。当一个进程的状态处于 SERVER_DEAD 的时候，意味着记分板中的该插槽可以被重新利用。

6.1.1.2.1 创建记分板

记分板的所有的操作都是从创建开始的，通常只有在刚启动 Apache 或者平稳启动之后才需要创建。Apache 中通过 ap_create_scoreboard 函数实现记分板的创建，该函数在 scoreboard.c 中实现，函数原型如下：

```
int ap_create_scoreboard(apr_pool_t *p, ap_scoreboard_e sb_type);
```

参数 p 指定创建记分板中所需要的内存来自的内存池，而 sb_type 则是创建的记分板的类型，或者为 SB_SHARED，或者为 SB_NOT_SHARED。前者允许记分板在不同的进程之间共享，而后者则不允许。

```
    int running_gen = 0;
    int i;
    if (ap_scoreboard_image) {
        running_gen = ap_scoreboard_image->global->running_generation;
        ap_scoreboard_image->global->restart_time = apr_time_now();
         memset(ap_scoreboard_image->parent, 0, sizeof(process_score) *
server_limit);
        for (i = 0; i < server_limit; i++) {
                    memset(ap_scoreboard_image->servers[i],          0,
```

```
sizeof(worker_score) * thread_limit);
        }
        if (lb_limit) {
            memset(ap_scoreboard_image->balancers, 0, sizeof(lb_score) *
lb_limit);
        }
        return OK;
    }
```

公告板的创建与几个系统值密切相关的，比如 server_limit 和 thread_limit。server_limit 描述了允许同时存在的进程的最大极限，包括父进程和子进程，每一个进程通常都是用上面的 process_score 数据结构进行描述；而 thread_limit 则是每一个子进程又允许生成的子线程的数目，这些子线程用 worker_score 进行描述。因此创建记分板的一个重要的步骤就是分配足够的插槽。由于 Apache 需要记录每一个进程以及进程中的每一个线程的运行信息，因此，创建记分板之前必须能够分配足够多的空间以容纳 process_score 和 worker_score 结构。

正如前面描述，系统中允许存在 server_limit 个进程，它们中的每一个都必须在记分板中拥有一个插槽，因此我们至少必须分配 sizeof(process_score)*server_limit 大小的内存空间，同时使用 ap_scoreboard_image->parent 指向该空间。

同时对于 server_limit 个进程中的每一个进程，他们可能产生的线程数为 thread_limit，这些线程也必须在记分板中拥有相应的插槽，为此共分配 server_limit*sizeof(worker_score)*thread_limit 的内存大小。

Apache 按照最大化的原则进行分配，一旦分配完毕肯定能够保证需要。不过这样的话可能存在很多的空闲插槽。因为即使只有一个进程和一个线程存在，Apache 也是会分配所有的内存的。

现在回到上面的代码中。如果是平稳启动，那么在创建新的记分板之前系统中应该已经存在一个旧的记分板(ap_scoreboard_image 不为 NULL)。在这种情况下，Apache 首先得到当前记分板的家族号，同时重新设置启动时间。另外一个重要的任务就是清理初始化记分板上的数据，将其全部清零，彻底扫荡前一个家族的所有信息，并将其返回出去供使用。

```
    ap_calc_scoreboard_size();
```

如果创建的时候发现公告板不存在，那么这意味着这是 Apache 启动以来的第一次记分板创建。因此创建之前必须计算记分板分配的空间大小。创建记分板的内存大小由函数 ap_calc_scoreboard_size()函数完成：

```
AP_DECLARE(int) ap_calc_scoreboard_size(void)
{
    ap_mpm_query(AP_MPMQ_HARD_LIMIT_THREADS, &thread_limit);
    ap_mpm_query(AP_MPMQ_HARD_LIMIT_DAEMONS, &server_limit);

    if (!proxy_lb_workers)
                                    proxy_lb_workers                    =
APR_RETRIEVE_OPTIONAL_FN(ap_proxy_lb_workers);
    if (proxy_lb_workers)
        lb_limit = proxy_lb_workers();
    else
        lb_limit = 0;
```

```
    scoreboard_size = sizeof(global_score); ❶
    scoreboard_size += sizeof(process_score) * server_limit; ❷
    scoreboard_size += sizeof(worker_score) * server_limit * thread_limit; ❸

    if (lb_limit)
        scoreboard_size += sizeof(lb_score) * lb_limit;❹

    return scoreboard_size;
}
```

如前所述，记分板内存大小由 thread_limit 和 server_limit 决定，这两个值可以通过 ap_mpm_query 进行查询键值 AP_MPMQ_HARD_LIMIT_DAEMONS 和 AP_MPMQ_HARD_LIMIT_THREADS 获取。一旦确定这两个核心值，那么我们就可以计算记分板的分配空间，它包括四个部分：

(1)、global_score 的大小，因此在整个系统中 global_score 只有一个，因此它所占的大小为 sizeof(global_score)，如 说示。

(2)、由于每一个进程都必须在记分板中拥有一个插槽来记录其相关信息，因此 server_limit 个进程所占的插槽的大小为 server_limit*sizeof(process_score)，如 所示。

(3)、对于每一个进程而言，其允许产生的线程的数目为 thread_limit 个，因此 server_limit 个进程允许产生的总线程数目为 thread_limit 个，这些线程也必须在记分板中拥有各自的信息插槽它们所占的内存为 server_limit*thread_limit*sizeof(worker_score)，如 所示。

(4)、，如 所示。

很容易看出，Apache 必须为记分板分配的空间大小为 sizeof(global_score) + server_limit*sizeof(process_score) + server_limit*thread_limit*sizeof(worker_thread)+ sizeof(lb_score) * lb_limit。计算后的内存大小保存在 scoreboard_size 全局变量中。

```
#if APR_HAS_SHARED_MEMORY
    if (sb_type == SB_SHARED) {
        void *sb_shared;
        rv = open_scoreboard(p);
        if (rv || !(sb_shared = apr_shm_baseaddr_get(ap_scoreboard_shm)))
{
            return HTTP_INTERNAL_SERVER_ERROR;
        }
        memset(sb_shared, 0, scoreboard_size);
        ap_init_scoreboard(sb_shared);
    }
    else
#endif
    {
        /* A simple malloc will suffice */
        void *sb_mem = calloc(1, scoreboard_size);
```

```
        if (sb_mem == NULL) {
            ap_log_error(APLOG_MARK, APLOG_CRIT, 0, NULL,
                         "(%d)%s: cannot allocate scoreboard",
                         errno, strerror(errno));
            return HTTP_INTERNAL_SERVER_ERROR;
        }
        ap_init_scoreboard(sb_mem);
    }
```

正常情况下，记分板应该作为共享内存存在从而被访问，但是并不是所有的操作系统都支持共享内存的操作。因此不同的操作系统，可能会采取不同的措施。

对于那些不支持共享内存的操作系统，Apache 只是简单的调用 calloc 函数分配 scoreboard_size 大小的内存块，同时调用 ap_init_scoreboard 对其进行初始化而已；如果操作系统支持共享内存，那么 Apache 将采用 IPC 技术创建一块共享内存，同时使用 apr_shm_baseaddr_get 得到该共享内存的首地址，并对其进行初始化。

```
    ap_scoreboard_image->global->sb_type = sb_type;
    ap_scoreboard_image->global->running_generation =
running_gen;
    ap_scoreboard_image->global->restart_time =
apr_time_now();

        apr_pool_cleanup_register(p, NULL, ap_cleanup_scoreboard,
apr_pool_cleanup_null);
```

在整个记分板创建完毕之后，对 global 中的全局属性进行设定，同时在内存池销毁链表中注册清除函数。当内存池被销毁的时候，其将调用 ap_cleanup_scoreboard 对记分板进行清除。

现在我们再回头看看共享内存的初始化函数 ap_init_scoreboard()，该函数主要用于对给定的共享内存块进行初始化，只有了解 ap_init_scoreboard 函数的初始化细节我们才能够明白记分板的内存布局状况。

在初始化的过程中一直存在两个内存块：一个是记分板本身的的内存块，即 scoreboard 数据结构内存块；一个是分配的共享内存块，其中保存实际的进程以及线程信息。初始化的任务实际上就是将记分板结构中的各个指针指向共享内存块中的相应的位置。

```
void ap_init_scoreboard(void *shared_score)
{
    char *more_storage;
    int i;

    ap_calc_scoreboard_size();
     ap_scoreboard_image = calloc(1, sizeof(scoreboard) + server_limit *
sizeof(worker_score *) +
                server_limit * lb_limit * sizeof(lb_score *));
```

在初始化之前，分配记分板所用内容。从上面的代码中，我们看到，分配的内存除了 sizeof(scoreboard) 大小是我们意料之内的，剩余的两部分 server_limit*sizeof(worker_score) 和 server_limit*lb_limit*sizeof(lb_score*) 则有点出乎意料之外。这两部分内存的用处我们稍后介绍。

```
        more_storage = shared_score;
        ap_scoreboard_image->global = (global_score
*)more_storage;
        more_storage += sizeof(global_score);
    ap_scoreboard_image->parent = (process_score *)more_storage;
    more_storage += sizeof(process_score) * server_limit;

    ap_scoreboard_image->servers =
            (worker_score        **)((char*)ap_scoreboard_image        +
sizeof(scoreboard));
    for (i = 0; i < server_limit; i++) {
        ap_scoreboard_image->servers[i] = (worker_score *)more_storage;
        more_storage += thread_limit * sizeof(worker_score);
    }

    if (lb_limit) {
        ap_scoreboard_image->balancers = (lb_score *)more_storage;
        more_storage += lb_limit * sizeof(lb_score);
    }
```

对于传入的共享内存按照下面的顺序进行布局初始化：首先保存全局共享信息 global_score，占用内存 sizeof(global_score)；然后保存 server_limit 个进程的信息，占用内存为 server_limit *sizeof(process_score)；第三部分的内存保存所有进程中产生的线程的信息，占用内存为 server_limit*thread_limit*sizeof(worker_score)；最后一部分的内存用于保存 lb_limit 个 lb_score 结构，四部分合计正好是分配的内存大小。上述的四部分分别通过 ap_scoreboard_image 中相关的成员指针指向：global 指针指向第一部分，parent 指向第二部分，balancers 指向第四部分。稍微复杂的这是的三部分线程信息的指定。共享内存中从 more_storage 指针往后 sizeof(process_score)*server_limit 的内存区域用于保存进程信息，因此将这块区域赋值给 parent 指针。

global_score 和 process_score 它们在本质上都可以用一维线性结构进行保存，而 worker_score 则更类似于二维结构，一方便它要记录本身的信息，另一方面还必须知道它是哪一个进程产生的线程，因此它们的保存方法不太一样。在 ap_scoreboard_image 分配的内存中我们可以看到除了正常的 sizeof(scoreboard) 大小之外，还包括了 server_limit * sizeof(worker_score *)大小的内存区域。该数组中的每一个元素都是一个指向 worker_score 结构的指针，指向 thread_limit 个元素的 worker_score 类型的数组。因此 servers[i]对应的数组记录的则是进程 i 创建的所有 thread_limit 个线程的信息。按照这种规律，第 i 个进程内的第 j 个线程可以用 server[i][j]进行描述。这个表达式在后面我们会多次使用到。

经过分配，整个记分板的内存布局可以用下图描述。

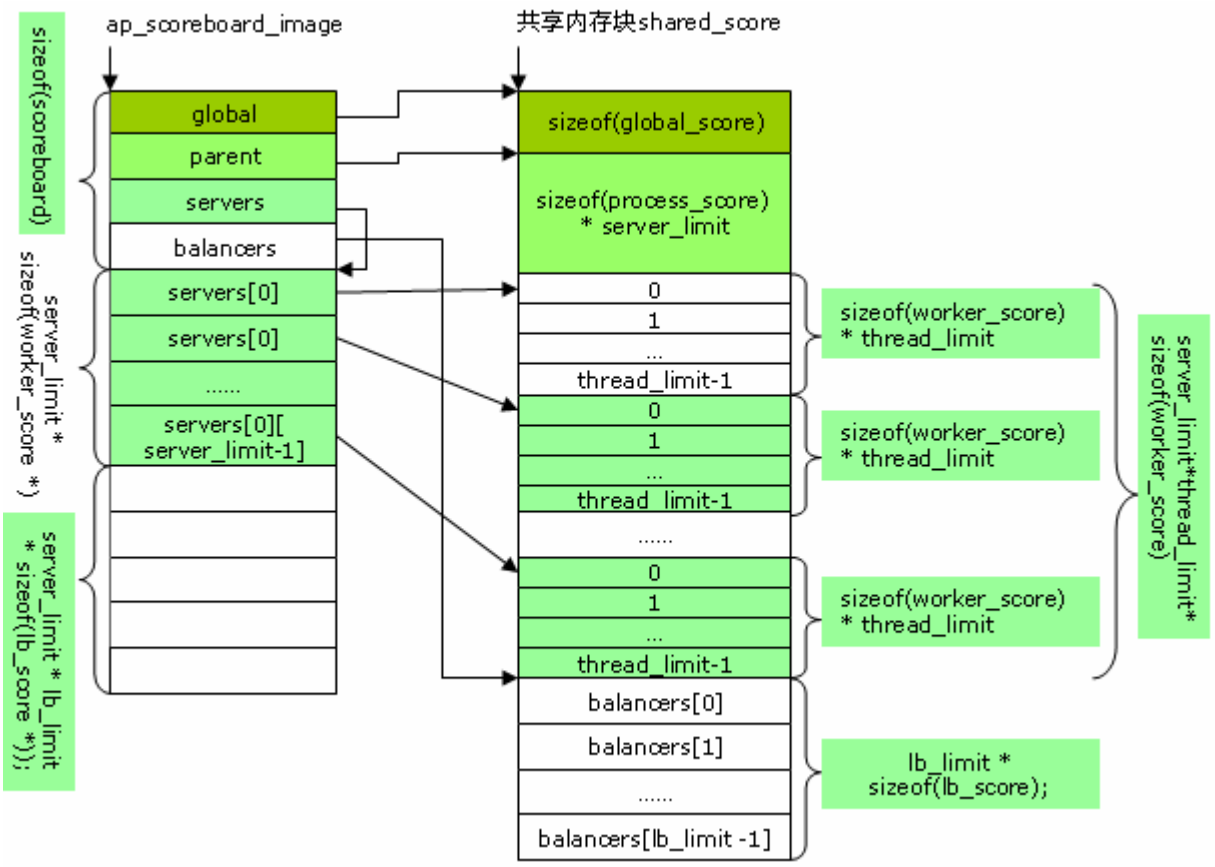


图 4.1 记分板内存分配图

```
        ap_assert(more_storage == (char*)shared_score +
scoreboard_size);
        ap_scoreboard_image->global->server_limit = server_limit;
        ap_scoreboard_image->global->thread_limit = thread_limit;
        ap_scoreboard_image->global->lb_limit     = lb_limit;
```

在所有的处理结束后判断 more_storage 指针时候指向了需要初始化空间的末尾，即判断 more_storage 是否与(char*)shared_score + scoreboard_size 相等，如果不相等，表明可能出错；

至此，一个完整的记分板已经创建完毕。

6.1.1.2.2 记分板插槽管理

对于记分板而言，其最频繁使用的一个功能就是在记分板中查找指定进程信息，函数 find_child_by_pid(apr_proc_t *pid)用以完成该功能。

```
AP_DECLARE(int) find_child_by_pid(apr_proc_t *pid)
{
    int i;
    int max_daemons_limit;
    ap_mpm_query(AP_MPMQ_MAX_DAEMONS,
&max_daemons_limit);
    for (i = 0; i < max_daemons_limit; ++i) {
```

```
            if (ap_scoreboard_image->parent[i].pid == pid->pid) {
                return i;
            }
        }
        return -1;
}
```

函数只需要一个参数，就是进程的描述数据结构。函数所做的事情无非就是对记分板中的 parent 数组逐一比较，判断其进程 id 是否是指定的 id，如果是则表明找到，并返回其索引值；否则意味着没有找到该进程信息。

记分板中的每一个插槽都记录了进程的相关信息，其中一个重要的属性就是当前进程的状态。当进程的状态发生变化的时候，记分板中的状态字段也应该随之变化。ap_update_child_status_from_indexes 函数用以更新记分板中指定进程的状态。

```
AP_DECLARE(int) ap_update_child_status_from_indexes(int
child_num,
                                                    int thread_num,
                                                    int status,
                                                    request_rec *r)
```

函数需要四个参数，child_num 是需要更新状态的进程的索引；thread_num 则是线程在该进程的线程组中的索引；status 是设置的新的状态值；r 则是与当前工作线程关联的请求结构。

```
        ws = &ap_scoreboard_image->servers[child_num]
[thread_num];
        old_status = ws->status;
        ws->status = status;

        ps = &ap_scoreboard_image->parent[child_num];
```

更新之前首先的任务就是获取索引为 child_num 的进程以及该进程内索引为 thread_num 的线程的描述数据结构。根据记分板的内存布局很容易理解上面的语句。不过对于线程，在对其进行更改之前必须保存其以前的状态。

```
    if (status == SERVER_READY
        && old_status == SERVER_STARTING) {
        ws->thread_num = child_num * thread_limit +
thread_num;
        ps->generation = ap_my_generation;
    }
```

Apache 中每个进程都会用一个唯一的整数进行标识。与此类似，每一个线程也会用一个唯一的整数进行标识。线程的识别号取决于它所在的进程索引以及它在进程内的索引：

线程号 = 线程所在的进程的索引 * 每个进程允许产生的线程极限 + 线程在进程内的索引

线 程号实际上就是该线程描述结构在整个线性描述数组中的索引，同时线程的家族号就是父进程的家族号。不过并不是每一个线程都会有一个线程号。只

有处于就绪状 态或者正在工作的线程才会安排到对应的线程号。如果一个进程刚刚创建尚未准备好处理客户端请求，那么它暂时还不会分配线程号。

```
        if (ap_extended_status) {
            ws->last_used = apr_time_now();
            if (status == SERVER_READY || status ==
SERVER_DEAD) {
                if (status == SERVER_DEAD) {
                    ws->my_access_count = 0L;
                    ws->my_bytes_served = 0L;
                }
                ws->conn_count = 0;
                ws->conn_bytes = 0;
            }
            if (r) {
                conn_rec *c = r->connection;
                apr_cpystrn(ws->client, ap_get_remote_host(c, r-
>per_dir_config,
                                    REMOTE_NOLOOKUP, NULL), sizeof(ws-
>client));
                if (r->the_request == NULL) {
                    apr_cpystrn(ws->request, "NULL", sizeof(ws-
>request));
                } else if (r->parsed_uri.password == NULL) {
                    apr_cpystrn(ws->request, r->the_request,
sizeof(ws->request));
                } else {
                    apr_cpystrn(ws->request, apr_pstrcat(r->pool,
r->method, " ",
                                        apr_uri_unparse(r->pool, &r-
>parsed_uri,
                                        APR_URI_UNP_OMITPASSWORD),
                                        r->assbackwards ? NULL : " ", r-
>protocol, NULL),
                                        sizeof(ws->request));
                }
                apr_cpystrn(ws->vhost, r->server-
>server_hostname,
                                    sizeof(ws->vhost));
            }
        }
```

在前面部分，我们讨论 worker_score 结构的时候讨论了 mod_status 模块，它允许详细的显示进程和线程的状态信息。不过这也是有条件的。通常情况下很容易理解线程信息显示的越详细肯定会影响服务器的效率。因此通常情况下，监控信息显示的都不是很详细。除非你手工设置。Apache 中提供了一个指令

ExtendedStatus 来控制是否需要在记分板中记录每个线程的详细信息。该指定反映到程序中则是通过全局变量 ap_extended_status 来控制。ap_extended_stauts 为 1 的话则意味着必须将线程的详细信息写入到记分板中。

上面的代码正是在记分板中记录线程的详细信息，包括请求客户端的 IP 地址，请求行以及请求虚拟主机名称。

6.1.1.2.3 记分板内存释放

当记分板不再使用的时候，记分板占用的内存必须被使用。记分板的释放通常只在 Apache 完全重新启动的时候才会进行。对于平稳启动，记分板不会被释放，只是完成重新初始化。

记分板通过函数 ap_cleanup_scoreboard()完成内存释放。

```
apr_status_t ap_cleanup_scoreboard(void *d)
{
    if (ap_scoreboard_image == NULL) {
        return APR_SUCCESS;
    }
    if (ap_scoreboard_image->global->sb_type == SB_SHARED)
{
        ap_cleanup_shared_mem(NULL);
    }
    else {
        free(ap_scoreboard_image->global);
        free(ap_scoreboard_image);
        ap_scoreboard_image = NULL;
    }
    return APR_SUCCESS;
}
```

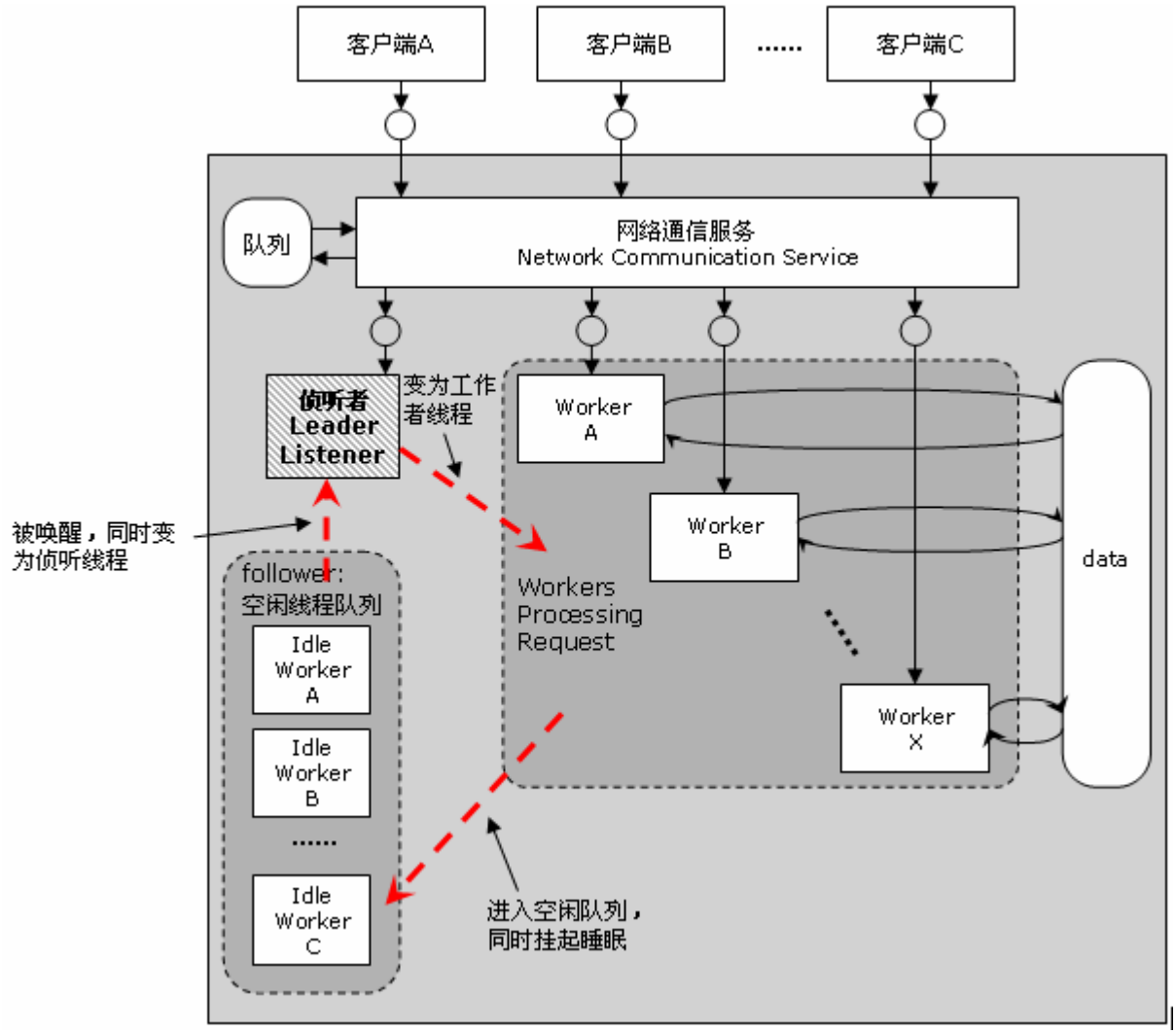
如果记分板没有被共享，那么对它的释放就非常简单，直接调用 free 即可，如前所述，记分板中的两个内存块分别用 ap_scoreboard_image 和 ap_scoreboard_image->global 分别标识。

如果记分板被用于进程间共享，则还必须调用共享内存的相关删除函数。

Apache 中预创建 Preforking MPM 机制剖析(1)

6.3.1 Leader/Follow 模式

在了解 Preforking MPM 之前有必要首先了解 Leader/Follow 模型。Preforking 模型本质上也属于 Leader/Follow 模型。通常情况下，L/F 可以用下图进行描述：



通常情况下，对于服务器中的进程采用的都是即时创建的策略，即一旦有一个新的客户端请求立即创建一个新的进程或 者线程，而当进程或者线程执行完毕后，进程和线程也随之退出。显然这种策略对于小规模的服务器还能接受，但是如果对于大规模的服务器而言，创建进程或者线 程的时间将增加，最终会导致响应时间变长，单位时间内请求处理效率降低。L/F 模式则不同，它首先一次性创建多个进程或者线程，包括到系统中，这些进程或 者线程担任三种不同的角色：侦听者、工作者以及空闲者，其含义分别如下：

1）、侦听者的角色。该线程负责侦听客户端的请求。在 L/F 模式中它属于 Leader 的角色。通常情况下只允许一个进程或者线程担当侦听者的角色。

2）、工作者的角色。当侦听者侦听到客户端的请求之后，它将立即转换为工作者角色并开始处理客户端的请求。工作者角色的线程可以有多个。

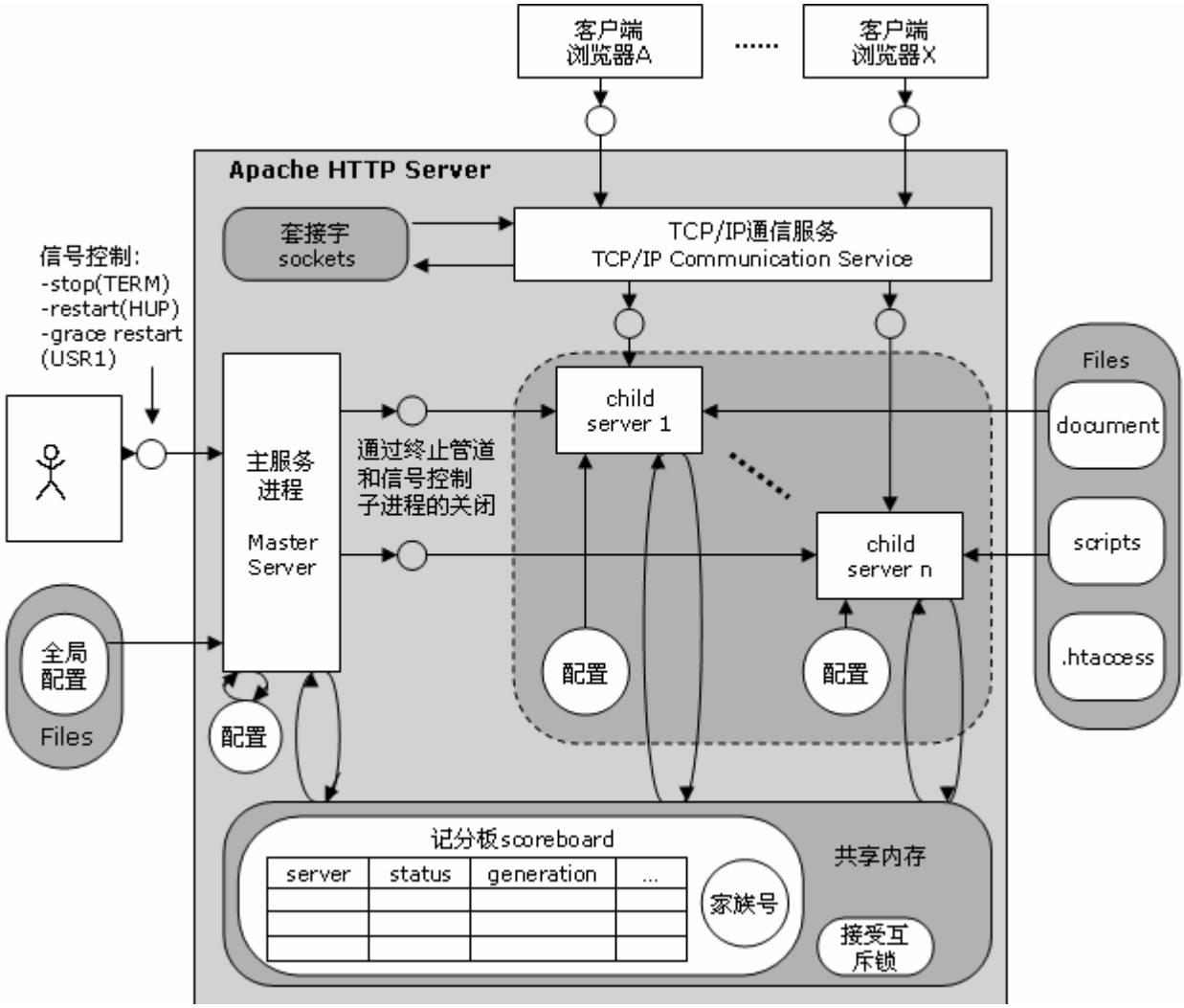
3）、空闲者的角色。当工作者执行任务完毕后它并不会立即退出，而是转变它的角色为空闲者，并呆在空闲队列中。 空闲者出现的原因是因为客户端请求不够多。空闲者们等待变为侦听者。而当侦听者变为工作者之后，空闲者中的每一个都相互竞争，最终将会有一个线程变为侦听 者，其余的继续保持空闲者的状态。

4）、几个极端的情况也是可能出现的：所有线程都变为工作者，忙于处理客户端的请求，没有线程担任侦听者的角色，因此此时客户端的请求都被拒绝；没有工作者，如果没有任何请求到达，那么所有的线程都处于空闲状态。

线程的三个角色的相互转换关系可以用上图的红线进行描述。

6.3.2 Preforking MPM概述

UNIX平台上可以使用的第一个MPM就是预先派生(Preforking)MPM，也是默认的MPM。该模型在功能上等同于Apache1.3上的模型，该MPM的示意图如下所示：



该MPM中，存在一个主进程和多个子进程。每个子进程 都会为所进行的请求侦听一个套接字。当接受到请求之后，子进程就会接受它并且提供响应。父进程会监控所有的子进程以确保总是可以使用最少数量的进程来处理 请求，并且确保等候请求到达的闲置进程不能过少。如果没有足够的空闲进程来处理潜在的请求高峰，那么父进程就会启动新的子进程。如果存在过多的进程，那么 父进程会每次终止一个空闲进程，直到服务器回到最大空闲子进程数量之下。通过保持一定数量的空闲子进程来接受所引入的请求，服务器就可以避免在接受到请求 时再去启动新进程的开销。

父进程和子进程之间通过记分板进行通信。对于每一个产生的子进程，它的状态信息都写入到记分板中，父进程通过读取记分板可以了解子进程的状态。当需要关闭子进程的时候它将通过终止管道发送终止信息给子进程，另外的一种通知方法就是通过信号。

预先派生模型有一些优点，例如健壮性以及可靠性。 Apache允许使用动态模块将第三方的代码加入到服务器。这意味着如果管理员在服务器中增加了第三方软件，而且模块导致了子进程出现段故障，那么服务器 就会丢失一个连接，而且仅仅丢失一个连接。服务器的其余部分还将继续运行，并且可以为请求提供服务。唯一可以注意到问题出现的用户就是不幸地进行了导致问

题地请求的用户。另一方面必须要意识到，可能并不是遭遇了故障的请求而导致了问题，可能会是因为有一系列请求而导致了问题的出现。

这个模型的另外一个优点就是可以很容易地编写采用这种 方式运行的MPM。如果每个进程每次只需要处理一次请求，那么就没有过多的边界条件需要考虑。例如，如果管理员想平稳重新启动服务器，那么她就要等待所有 的子进程完成当前请求，并且适时强制其终止来完成任务，然后父进程就可以启动新的进程来代替旧的进程了。

不过预派生模型也有自己的缺点，比如扩充性。因为预先 派生模型是依赖于进程，所以在某些平台上并不能很好的执行，比如在Window平台上则由于进程的代价太高，耗时太长的原因则弃用该方案。当然 Window并不是唯一的遭遇该问题的OS。比如在高负载的情况下AIX也会遇到这个问题。

预先派生模型的另外一个问题就是安全性。许多ISP都 会在相同的计算机上使用Apache来为多个公司的Web站点提供需要的Web服务。为了完成这个任务，每个公司都要被赋予一个虚拟主机，但是因为所有的 站点都需要访问Web服务器，所以必须通过运行子进程的用户ID来读取所有的页面。大多数Apache用户运行Apache的方式是作为超级用户运行父进 程，然后作为专用于Web服务器的用户运行子进程。这样就可以让服务器打开特权端口80，而且还可以确保侦听网络的进程不作为根用户运行，因此就减少了发 现安全漏洞所带来的风险。因为所有的虚拟主机都会使用相同的用户ID运行，所以它们的CGI教本也会使用这个ID运行。这意味着任何为这个站点存储信息的 数据库都必须要通过这个用户可读。这样就会让任何站点访问其他站点的私有信息。当然，Apache已经针对这个问题提供了解决方案，它可以让站点规定它们 的CGI脚本作为哪个用户运行但是这只是解决了实际的CGI教本的问题，并没有解决PHP，Apache模块或者通过mod_perl运行的Perl脚本 的问题。

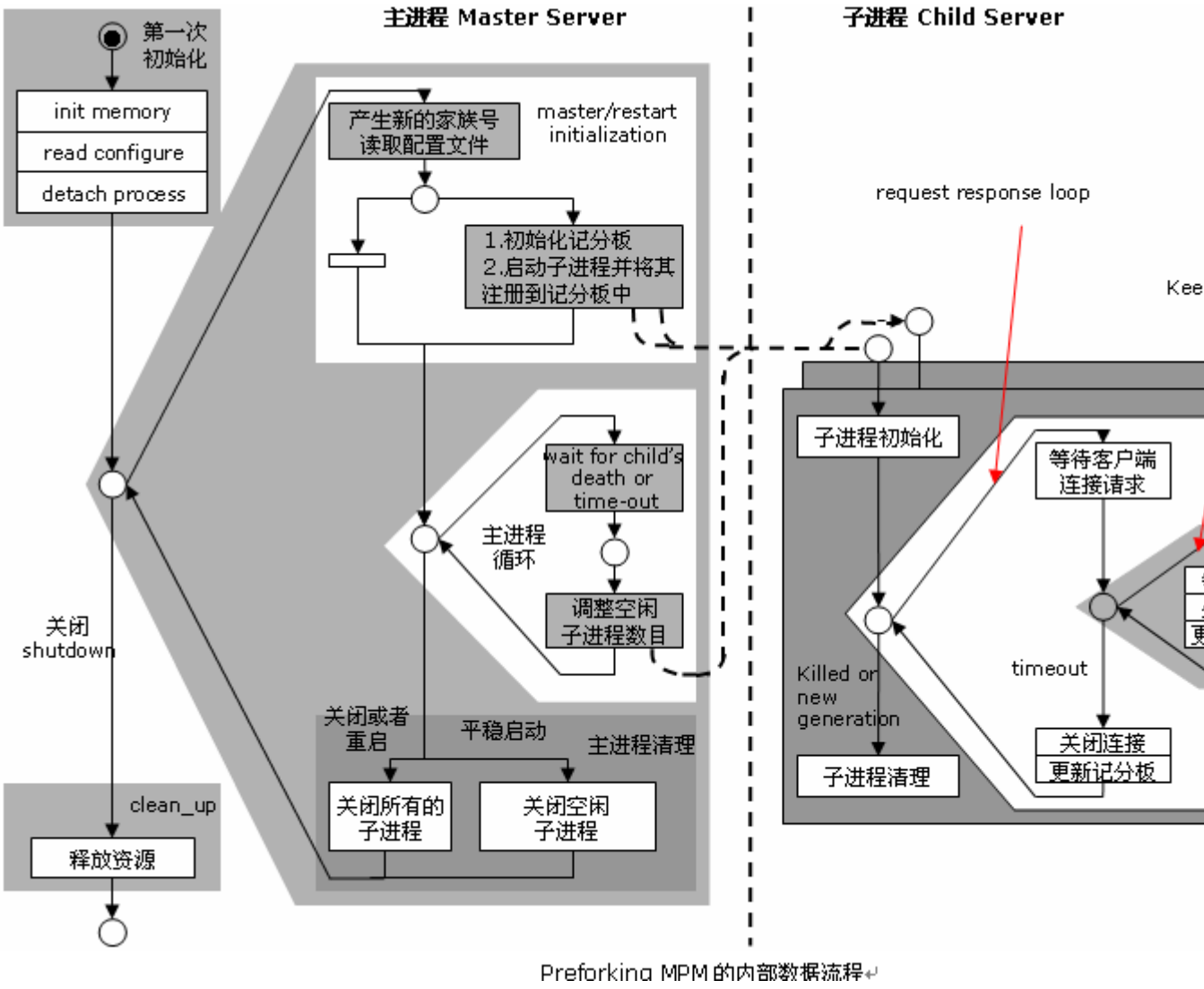
这种设计的最后问题就是它会消弱某些优化。因为每个请 求都会在它自己的进程中运行，所以进程之间很难共享任何信息。增加Web服务器性能的常见方式就是缓存最近所发送的所有的页面。但是，如果每个进程都要保 存它自己的缓存，那么缓存的作用就会降低。只有多次获取缓存中所缓存的页面缓存才会有作用。大多数情况下，这种方式会随着时间的推移逐渐显示其作用缓存 总是会缓存最常受到请求的页面。这意味着，到缓存起作用的时候，进程就要受到没有缓存的新进程的替换。Apache中的子进程会在由配置文件中的 MaxRequestsPerChild指令所控制的指定时间间隔终止。Apache在指定的请求数量之后强制终止子进程的原因是为了防止内存泄漏。因为 Apache要在相当长的时间内运行，所以很小的内存泄漏也会导致服务器出现问题。而且因为子进程是唯一分配内存的进程——所以强制其退出，并且偶尔对其重新启动——就有可能避免由于内存泄漏而导致的问题。

在下面的部分，我们将描述MPM的实现细节。首先我们分析主进程管理细节，然后分析子进程的工作细节，最后我们分析主进程是如何与子进程进行通信并对其进行管理的。

6.3.3 Preforking MPM实现

6.3.3.1 内部结构

在对Preforking MPM进行深入的分析之前，我们先从整体上了解Preforking MPM的内部结构，从而能够从整体上有一个认识，这样在后面的具体分析中不至于迷失方向。下图给出的则是Preforking MPM的中的内部大概的实现机制：



Preforking MPM 的内部数据流程

一个完整的 MPM 包括下面几个数据流程：

1)、第一次初始化

第一次初始化的时候是分配资源(主要是内存池)，读取以及检查配置文件，然后服务器进程将其变为一个后台进程。

2)、重启循环

重启循环是指不关闭 Apache 而进行的启动。重启循环中主要重新读取配置文件防止配置文件发生变化，创建子进程池并进入服务器主循环。

3)、服务器主进程循环

服务器主循环主要是控制进程池中空闲子进程的数目，具体到细节中则是循环监控记分板，并根据记分板中子进程的状态作出反应。

4)、客户端请求/响应循环

这个循环只适合于子进程。在该循环中，子进程等待自己变为侦听者，然后等待客户端的连接，一旦获取到连接则变为工作者开始处理请求，同时进入 Keep-alive 循环中。

5)、Keep-alive 循环

Keep-alive 循环主要是处理客户端的请求，该循环仅仅适合子进程。

6)、在退出之前进行清理工作

6.3.3.2 MPM 中的定义

预创建 MPM 所对应的文件为 prefork.c。通常情况下，MPM 的名称总是和 MPM 的文件具有相同的名称，这样做可以让配置更合理一些。对于每一个 MPM，其遇到的第一件事情就是定义一些全局变量，它们各自的含义描述在右边的注释中：

```
int ap_threads_per_child=0;                                    /* 每个进程对应的
线程数目 */
static apr_proc_mutex_t *accept_mutex;                         /*连接接受互斥锁，
用以确保在任何时候只有一个连
                                                               接被接受*/
static int ap_daemons_to_start=0;                              /*初始启动的进程数目*
/
static int ap_daemons_min_free=0;                              /*可以接受的空
闲进程的最小数目*/
static int ap_daemons_max_free=0;                              /*允许空闲的进程
的最大数目*/
static int ap_daemons_limit=0;                                 /*允许同时
运行的进程的最大值*/
static int server_limit = DEFAULT_SERVER_LIMIT;        /**/
static int first_server_limit;
static int changed_limit_at_restart;
static int mpm_state = AP_MPMQ_STARTING;                       /*描述当前*/
static ap_pod_t *pod;

int ap_max_daemons_limit = -1;

server_rec *ap_server_conf;
static int one_process = 0;

static apr_pool_t *pconf;                                      /* Pool for config stuff */
static apr_pool_t *pchild;                                     /* Pool for httpd child
stuff */

static pid_t ap_my_pid;                                /* it seems silly to call getpid
all the time */
static pid_t parent_pid;
#ifndef MULTITHREAD
static int my_child_num;
#endif
ap_generation_t volatile ap_my_generation=0;
static int die_now = 0;
```

另外一个所有的 MPM 都用到的函数就是 ap_mpm_query()，外界调用该函数通常是想了解当前 MPM 的 一些私有属性。比如在 mod_snake(在 Apache 进程中嵌入 python 解释器的模块)中就会使

用这个函数来查询给定的MPM是否进行了线程化。 如果进行了线程化，那么该模块就必须同步某些python函数，否则就不需要进行同步了。

Apache中关于MPM的状态分类可以归结为两大类：运行状态和内部状态。

```
#define AP_MPMQ_STARTING              0
#define AP_MPMQ_RUNNING               1
#define AP_MPMQ_STOPPING              2
```

上面三种属于运行状态，分别表示Apache处理启动、运行和停止状态。

```
#define AP_MPMQ_MAX_DAEMON_USED       1 /* 所有的进程都已经满了          */
#define AP_MPMQ_IS_THREADED           2 /* MPM 能够支持线程化            */
#define AP_MPMQ_IS_FORKED             3 /* MPM 能够调用fork产生子进程 */
#define AP_MPMQ_HARD_LIMIT_DAEMONS    4 /* The compiled max # daemons   */
#define AP_MPMQ_HARD_LIMIT_THREADS    5 /* The compiled max # threads   */
#define AP_MPMQ_MAX_THREADS           6 /* # of threads/child by config */
#define AP_MPMQ_MIN_SPARE_DAEMONS     7 /* Min # of spare daemons       */
#define AP_MPMQ_MIN_SPARE_THREADS     8 /* Min # of spare threads       */
#define AP_MPMQ_MAX_SPARE_DAEMONS     9 /* Max # of spare daemons       */
#define AP_MPMQ_MAX_SPARE_THREADS    10 /* Max # of spare threads       */
#define AP_MPMQ_MAX_REQUESTS_DAEMON 11 /* Max # of requests per daemon */
#define AP_MPMQ_MAX_DAEMONS          12 /* Max # of daemons by config   */
#define AP_MPMQ_MPM_STATE            13 /* starting, running, stopping */
```

上面的都属于Apache的内部状态。预创建MPM中的函数定义如下：

```
AP_DECLARE(apr_status_t) ap_mpm_query(int query_code, int *result)
{
    switch(query_code){
        case AP_MPMQ_MAX_DAEMON_USED:
            *result = ap_daemons_limit;
            return APR_SUCCESS;
        case AP_MPMQ_IS_THREADED:
            *result = AP_MPMQ_NOT_SUPPORTED;
            return APR_SUCCESS;
        case AP_MPMQ_IS_FORKED:
            *result = AP_MPMQ_DYNAMIC;
            return APR_SUCCESS;
        case AP_MPMQ_HARD_LIMIT_DAEMONS:
            *result = server_limit;
            return APR_SUCCESS;
        case AP_MPMQ_HARD_LIMIT_THREADS:
            *result = HARD_THREAD_LIMIT;
            return APR_SUCCESS;
        case AP_MPMQ_MAX_THREADS:
            *result = 0;
            return APR_SUCCESS;
        case AP_MPMQ_MIN_SPARE_DAEMONS:
            *result = ap_daemons_min_free;
```

```
            return APR_SUCCESS;
        case AP_MPMQ_MIN_SPARE_THREADS:
            *result = 0;
            return APR_SUCCESS;
        case AP_MPMQ_MAX_SPARE_DAEMONS:
            *result = ap_daemons_max_free;
            return APR_SUCCESS;
        case AP_MPMQ_MAX_SPARE_THREADS:
            *result = 0;
            return APR_SUCCESS;
        case AP_MPMQ_MAX_REQUESTS_DAEMON:
            *result = ap_max_requests_per_child;
            return APR_SUCCESS;
        case AP_MPMQ_MAX_DAEMONS:
            *result = server_limit;
            return APR_SUCCESS;
        case AP_MPMQ_MPM_STATE:
            *result = mpm_state;
            return APR_SUCCESS;
    }
    return APR_ENOTIMPL;
}
```

这个是ap_mpm_query函数是所有的MPM都要求的函数，它可以让模块发现MPM的运行特性。尽管这个 函数在所有的MPM中看起来很相似，但是细节还是十分重要，因为每个MPM都必须实现自己的这个函数。使用这个函数最常见的原因就是要通过web或者管理 应用程序报告信息。可以采用很多方式来使用这个函数中的信息，所以应该保证它正确无误。比如mod_snake(在Apache中嵌入python解释器 的模块)就会使用这个查询模块来确定MPM是否进行了线程化，如果进行了线程化，那么该模块可能就必须同步某些python函数；否则就不需要进行同步。

6.3.3.3 主服务进程管理

6.3.3.3.1 主服务进程概述

所有的MPM都是从ap_mpm_run()函数开始执行，对此预创建MPM也不例外。 ap_mpm_run()函数通常由Apache核心在main()中进行调用，一旦调用，运行服务器的职责就从Apache核心移交给了MPM。这个函 数是所有的MPM都必须实现的。通常情况下，ap_mpm_run的实现会比较复杂。对于Preforking MPM，它的执行流程可以用下图描述：

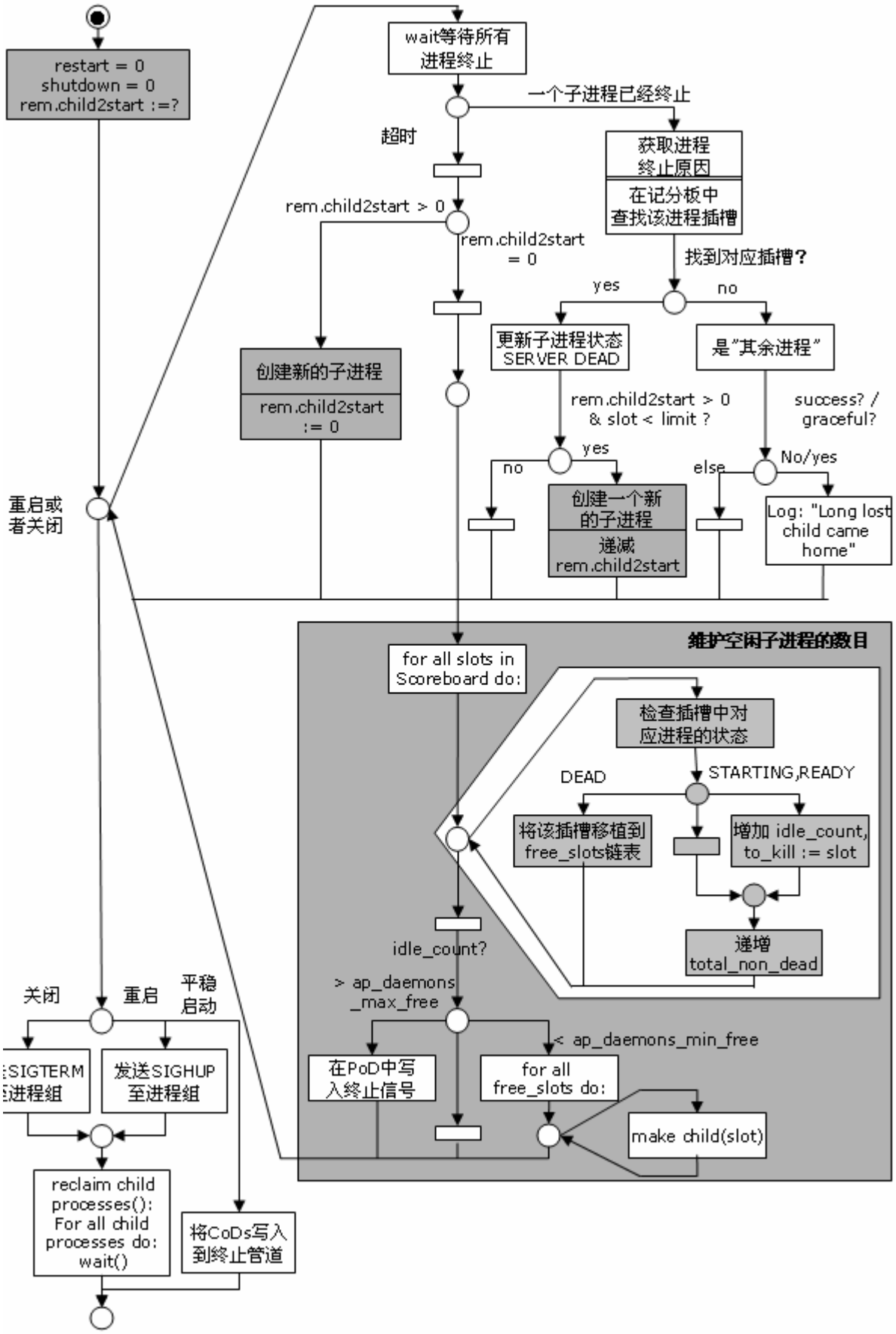

restart = 0
shutdown = 0
rem.child2start :=?
wait等待所有进程终止
一个子进程已经终止
超时
获取进程终止原因
在记分板中查找该进程插槽
rem.child2start > 0
rem.child2start = 0
找到对应插槽?
yes
no
更新子进程状态 SERVER DEAD
是"其余进程"
创建新的子进程
rem.child2start := 0
rem.child2start > 0 & slot < limit ?
success? / graceful?
yes
no
else
No/yes
创建一个新的子进程
递减 rem.child2start
Log: "Long lost child came home"
重启或者关闭
维护空闲子进程的数目
for all slots in Scoreboard do:
检查插槽中对应进程的状态
DEAD
STARTING,READY
将该插槽移植到free_slots链表
增加 idle_count, to_kill := slot
递增 total_non_dead
idle_count?
> ap_daemons _max_free
< ap_daemons_min_free
关闭
重启
平稳启动
SIGTERM 进程组
发送SIGHUP 至进程组
在PoD中写入终止信号
for all free_slots do:
make child(slot)
reclaim child processes(): For all child processes do: wait()
将CoDs写入到终止管道

图 主进程工作流程

图 主进程工作流程

从上图中可以看出，主服务进程的功能主要包括下面几部分：

1)、接受进程外部信号进行重启，关闭以及平稳启动等等操作。外部进程通过发送信号给主服务进程以实现控制主服务进程的目的。

2)、在启动的时候创建子进程或者在平稳启动的时候用新进程替代原有进程。

3)、监控子进程的运行状态，并根据运行负载自动调节空闲子进程的数目：在存在过多空闲子进程的时候终止部分空闲进程；在空闲子进程较少的时候创建更多的空闲进程以便将空闲进程的数目维持在一定的数目之内。

上图的上半部分主要处理子进程的终止以及平稳启动的一些内容，而标有”维护空闲子进程数目”的下半部分则主要处理平稳启动的一些内容。在该循环中，主进程统计空闲子进程的数目并从记分板中获得详细的空闲进程列表，然后将统计得到的空闲进程数目 idle_count 与系统设置的极限 ap_daemons_max_free 和 ap_daemons_min_free)进行对比。如果有过多的空闲子进程，那么它将每次循环中终止一个子进程；反之它将一次性创建足够的空闲进程。

下面的部分我们概要的描述一下主服务进程是如何处理平稳启动，子进程终止以及如何维持空闲子进程数目的。

平稳启动以及处理子进程终止

u 设置 remaining_children_to_start(以后简称 rem.child2start)。变量 rem.child2start 只有在平稳启动的时候才会用到。它记录的是服务器启动后需要启动的子进程的数目。需要注意的是主服务进程并没有使用 startup_children 过程创建子进程。对于每一个被终止的子进程，主服务进程通过调用 wait()可以得到 终止通知。如果初始的子进程的数目在配置文件中被更改了那么仅仅用新进程替代终止的子进程肯定会出错，因此主服务进程用 rem.child2start 控制这个数目。

vpid = wait 或者超时。对于使用 fork 创建的子进程，主服务进程调用 wait 等待它们的终止，这种做法确保没有僵尸进程的产生。如果在指定的时间内进程仍然没有终止，那么主服务进程算超时处理，即使没有等到终止通知，主服务进程将继续执行它的循环。

w 等待成功的情况。如果在指定的时间内子进程终止，此时主服务器进程将完成下面的各项工作

- process_child_status：获取子进程终止的原因。子进程的终止可能有很多情况，比如正常终止，异常终止等等。正常终止对于主服务进程到无所谓，异常终止主进程则必须知道具体原因。
- find_child_by_pid： 当进程终止后，必须在记分板中更新它的状态信息，因此首先必须在记分板中查找该进程对应的插槽信息。如果查找，则将进程的状态信息设置为 SERVER_DEAD。如果 remaining_children_to_start 不为零的话，创建一个新的子进程来代替终止的子进程。
- 如 果没有在记分板中没有找到终止进程的对应插槽，那么检查该进程是否是“其余子进程”。一些情况下，主服务进程会创建一些非子服务进程的进程，它们称之为“ 其余进程”，并用一个单独的列表进行登记。比如，一般情况下，Apache 会将日志写入到文件中，但是有的时候 Apache 则希望将数据写入到一个给定的 应用程序中。因此主服务器进程必须为该应用程序创建该进程，并且将该进程的标准输入 STDIN 关联道主服务进程的日志流中。这种进程并不是用来执行处理 HTTP 请求的子服务进程，因此称之为“其余进程”。任何时候只要服务器重启，日志应用进程都会接受到 SIGHU 和 SIGUSR1 信号，然后终止退出，对 应的模块必须重新创建这种进程。如果进程既不是“其余进程”，在记分板中也找不到对应的插槽，并且设置了平稳启动模式，那么肯定发生了下面的情况：管理员 减少了允许的子进程数目同时进行了平稳启动或者。。。

x 等待超时：如果所有的终止了的进程被新创建的新进程代替之后 rem.child2start 变量的值仍然不为零，那么这意味着必须创建更多的子进程。创建由 startup_children()函数完成。

yz

空闲子进程维护

下面的一节我们将详细的分析主进程以及子进程相关的源代码。

6.3.3.3.2 主服务进程概述

```
int ap_mpm_run(apr_pool_t *_pconf, apr_pool_t *plog, server_rec *s)
{
    int index;
    int remaining_children_to_start;
    apr_status_t rv;

    ap_log_pid(pconf, ap_pid_fname);
```

对于所有的 Apache MPM 而言，其应该首先完成的工作就是在文件 pidfile 中记录进程的 ID。因为启动和终止 Apache 的默认脚本通常会读取 pidfile 文件，从中 查找所有记录的进程然后逐个终止它。因此如果不进行记录的话，启动的这些进程可能无法通过脚本进行终止，这项操作进行的越快越好。

```
    first_server_limit = server_limit;
    if (changed_limit_at_restart) {
        ap_log_error(APLOG_MARK, APLOG_WARNING, 0, s,
                     "WARNING: Attempt to change ServerLimit "
                     "ignored during restart");
        changed_limit_at_restart = 0;
    }
```

理解上面这段代码的关键在于理解两个变量 first_server_limit 和 changed_limit_at_restart 的作用。

server_limit 变量用以记录服务器内允许同时存在的子服务进程的数目，用过通过配置文件中的 ServerLimit 指令可以修改这个值。当每次 Apache 启动的时候，通过读取配置文件这个指令的参数值最终保存到了 server_limit 变量 中并影响服务器的进程产生。由于当 Apache 重新启动(restart)的时候也会读取配置文件，因此如果服务器重新启动之前修改了配置文件中的 ServerLimit 指令参数，那么毫无疑问，这种变化 Apache 重启的时候肯定会看到的。那么 Apache 该如何处理这种变化呢？是使用新的 server_limit 还是使用原有的 server_limit? Apache 的做法是不允许在重启的时候修改 server_limit 值，即使你修改 了 Apache 也会忽略。为了能够检查出这种修改，在 Apache 第一次启动的时候，ServerLimit 的值就被记录在 first_server_limit 变量中，在整个 Apache 运行期间即使重新启动，这个值也不会变化。first_server_limit=server_limit 就是保存初始的值。

按正常的处理策略，对于每次重启后都应该把 server_limit 与 first_server_limit 的 值进行比较判断是否发生变化，如果发生变化就给出警告，但是上面的代码中并没有这种比较。那么比较在哪儿发生的呢？钥匙在 change_limit_at_restart 变量上。当重新启动后读取配置文件的时候，遇到 ServerLimit 指令会调用函数 set_server_limit 处理该指令，该函数中会将指令参数后面的值与 first_server_limit 进行比较：

```
    int tmp_server_limit;
    tmp_server_limit = atoi(arg);//ServerLimit 指令后的参数值
    if (first_server_limit &&tmp_server_limit != server_limit) {
        changed_limit_at_restart = 1;
        return NULL;
    }
    server_limit = tmp_server_limit;
```

从上面的代码中可以看出，changed_limit_at_restart 反映了 ServerLimit 在重 启期间是否发生了更改。对于这种更改，Apache 并不理会，只是简单的警告，并将 changed_limit_at_restart 设置为零，这样下次 重启就不要进行判断了。

```
    /* Initialize cross-process accept lock */
    ap_lock_fname = apr_psprintf(_pconf, "%s.%" APR_PID_T_FMT,
                                 ap_server_root_relative(_pconf, ap_lock_fname),
                                 ap_my_pid);

    rv = apr_proc_mutex_create(&accept_mutex, ap_lock_fname,
                               ap_accept_lock_mech, _pconf);
    if (rv != APR_SUCCESS) {
        ap_log_error(APLOG_MARK, APLOG_EMERG, rv, s,
                     "Couldn't create accept lock (%s) (%d)",
                     ap_lock_fname, ap_accept_lock_mech);
        mpm_state = AP_MPMQ_STOPPING;
        return 1;
    }

#if APR_USE_SYSVSEM_SERIALIZE
    if (ap_accept_lock_mech == APR_LOCK_DEFAULT ||
        ap_accept_lock_mech == APR_LOCK_SYSVSEM) {
#else
    if (ap_accept_lock_mech == APR_LOCK_SYSVSEM) {
#endif
        rv = unixd_set_proc_mutex_perms(accept_mutex);
        if (rv != APR_SUCCESS) {
            ap_log_error(APLOG_MARK, APLOG_EMERG, rv, s,
                         "Couldn't set permissions on cross-process lock; "
                         "check User and Group directives");
            mpm_state = AP_MPMQ_STOPPING;
            return 1;
        }
    }
```

在预创建 MPM 中由于存在多个子进程侦听指定的套接字，因此如果不加以控制可能会出现几个子进程同时对一个连接进行处理的情况，这是不允许的。因此我们必须采取一定的措施确保在任何时候一个客户端连接请求只能由一个子进程进程处理。为此 Apache 中引入了接受互斥锁(Accept Mutex)的概念。接受互斥锁是控制访问 TCP/IP 服务的一种手段，它能够确保在任何时候只有一个进程在等待 TCP/IP 的连接请求，从而对于指定的连接请求也只会有一个进程进行处理。

为此 MPM 紧接着必须创建接受互斥锁。

```
        if (!is_graceful) {
            if (ap_run_pre_mpm(s->process->pool, SB_SHARED) != OK) {
                mpm_state = AP_MPMQ_STOPPING;
                return 1;
```

```
            }
            ap_scoreboard_image->global->running_generation = ap_my_generation;
        }
```

多数的 MPM 紧接着会立即创建记分板，并将它设置为共 享，以便所有的子进程都可以使用它。记分板在启动的时候被创建一次，直到服务器终止时才会被释放。上面的代码就是用于创建记分板，但是你可能很奇怪，因为 你看不到我们描述的记分板创建函数 ap_create_scoreboard()。事实上，创建过程由挂钩 pre_mpm 完成，通过使用 pre_mpm， 服务器就可以让其他的模块在分配记分板之前访问服务器或者在建立子进程之前访问记分板。

ap_run_pre_mpm()运行挂钩 pre_mpm，该挂钩通常对应类似 ap_hook_name 之类的函数，对于 pre_mpm 挂钩，对应的函数则是 ap_hook_pre_mpm。在 core.c 中的 ap_hook_pre_mpm(ap_create_scoreboard, NULL, NULL, APR_HOOK_MIDDLE)设定挂钩 pre_mpm 的对应处理函数则正是记分板创建函数 ap_create_scoreboard。

ap_run_pre_mpm 挂钩也只有在进行重新启 动的时候才会调用，而在进行平稳启动的时候，并不调用这个挂钩，这样做会丢失所有的仍然正在为长期请求提供服务的子进程的信息。挂钩的引入是 Apache2.0 版本的一个新的实现机制，也是理解 Apache 核心的一个重要机制之一，关于挂钩的具体的实现细节我们在后面的部分会详细分析。

对于每次冷启动，Apache 启动之后，内部的记分板的家族号都是从 0 开始，而每一次平稳启动后家族号则是在原先的家族号加一。

```
    set_signals();
```

当分配了记分板之后，MPM 就应该设置信号处理器，一方面允许外部进程通过信号通知其停止或者重新启动，另一方面服务器应该忽略尽可能多的信号，以确保它不会被偶然的信号所中断。正常情况下，父进程需要处理三种信号：正常的重新启动、非正常的重新启动以及关闭的信号。

SIGTERM：该信号用于关闭主服务进程。信号处理函数 sig_term 中设置 shutdown_pending=true;

SIGHUP：该信号用于重启服务器，信号处理函数中设置 restart_pending=true 和 graceful_mode=false

SIGUSR1：该信号用于平稳启动服务器，信号处理函 restart 数中设置 restart_pending=true 和 graceful_mode=true

至于信号 SIGXCPU、SIGXFSZ 则由默认的信号处理函数 SIG_DFL 处理，SIGPIPE 则会被忽略。

在 Apache 主程序的循环中，程序不断的检测 shutdown_pending，restart_pending 和 graceful_mode 三个变量的值。通常并不是外部程序一发送信 号，Apache 就立即退出。最差的情况就是信号是在刚检测完就发送了，这样，主程序需要将该次循环执行完毕后才能发现发送的信号。

```
    if (one_process) {
        AP_MONCONTROL(1);
        make_child(ap_server_conf, 0);
    }
```

至此大部分准备工作已经完成，剩下的任务就是创建进程。进程的创建包括两种模式：单进程模式和多进程模式。

单进程模式通常用于 Apache 调试，由于不管多进程还是单进程，对 HTTP 请求处理以及模块等的使用都是完全相同的，区别仅仅在于效率。而多线程的调试要比单进程复杂的多。

如果是单进程调试模式，那么上面的两句程序将被程序。我们首先解释一下 AP_MONCONTROL 宏的含义。对 于调试，一方面可能比较关心执行的正确与否，内存是否溢出等等，另外一方面就是能够找出整个服务器的运行瓶颈，只有找到了运行的瓶颈才能进行改善，从而提 高效率。例如，

假设应用程序花了 50% 的时间在字符串处理函数上，如果可以对这些函数进行优化，提高 10% 的效率，那么应用程序的总体执行时间就会改进 5%。因此，如果希望能够有效地对程序进行优化，那么精确地了解时间在应用程序中是如何花费的，以及真实的输入数据，这一点非常重要。这种行为就称为代码剖析（code profiling）。

An executable program compiled using the -pg option to cc(1) automatically cally includes calls to collect statistics for the gprof(1) call-graph execution profiler. In typical operation, profiling begins at program startup and ends when the program calls exit. When the program exits, the profiling data are written to the file gmon.out, then gprof(1) can be used to examine the results.

一个可执行的应用程序可以在使用 gcc 编译的时候利用-pg 选项自动的调用相关函数收集一些执行统计信息以便 gprof execution profiler 使用。

moncontrol() selectively controls profiling within a program. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use moncontrol(0); to resume the collection of his-histogram togram ticks and call counts use moncontrol(1). This feature allows the cost of particular operations to be measured. Note that an output file will be produced on program exit regardless of the state of moncontrol().

Programs that are not loaded with -pg may selectively collect profiling statistics by calling monstartup() with the range of addresses to be pro-profiled.

filed. lowpc and highpc specify the address range that is to be sampled; the lowest address sampled is that of lowpc and the highest is just below highpc. Only functions in that range that have been compiled with the -pg option to cc(1) will appear in the call graph part of the output;

however, all functions in that address range will have their execution time measured. Profiling begins on return from monstartup().

单进程的另外一个任务就是调用 make_child。对于单进程，make_child 非常的简单：

```
static int make_child(server_rec *s, int slot)
{
    int pid;
    ……
    if (one_process) {
        apr_signal(SIGHUP, sig_term);
        apr_signal(SIGINT, sig_term);
        apr_signal(SIGQUIT, SIG_DFL);
        apr_signal(SIGTERM, sig_term);
        child_main(slot);
        return 0;
    }
    /*多进程处理代码*/
}
```

从代码中可以看出，单进程直接调用了 child_main，该函数用于直接处理与客户端的请求。在整个系统中只有一个主服务进程存在。

```
    else {
    if (ap_daemons_max_free < ap_daemons_min_free + 1) /* Don't thrash... */
```

```
        ap_daemons_max_free = ap_daemons_min_free + 1;

    remaining_children_to_start = ap_daemons_to_start;
    if (remaining_children_to_start > ap_daemons_limit) {
        remaining_children_to_start = ap_daemons_limit;
    }
    if (!is_graceful) {
        startup_children(remaining_children_to_start);
        remaining_children_to_start = 0;
    }
    else {
        hold_off_on_exponential_spawning = 10;
    }
```

对于多进程模式而言，处理要复杂的多。与多进程类似，上面的代码负责创建子进程。在创建之前对其中使用到的几个核心变量进行必要的调整，这几个变量的含义分别如下：

ap_daemons_max_free：服务器中允许存在的空闲进程的最大数目，一旦超过这个数目，一部分空闲进程就会被迫终止，直到最后的空闲进程数目降低到该值。

ap_daemons_min_free：服务器中允许存在的空闲进程的最小数目，一旦低于这个数目，服务器将创建新的进程直到最后空闲进程数目抵达这个数目。任何时候空闲进程的数目都维持在 ap_daemons_max_free 和 ap_daemons_min_free 之间。

ap_daemons_limit：服务器中允许存在的进程的最大数目。包括空闲进程、忙碌进程以及当前的记分板中的空余插槽。

ap_daemons_to_start：服务器起始创建的进程数目。这个值不能超出 ap_daemons_limit。

remaining_child_to_start：需要启动的子进程的数 目。对于初始启动，remaining_child_to_start 的值就是 ap_daemons_to_start 的值。因此服务器是刚启动，那么函 数直接调用 start_children 创建 remaining_child_to_start 个子进程，同时将 remaing_child_to_start 设置为零。

对于平稳启动，remaining_child_to_start 的含义则要发生一些变化。

如果我们所进行的是平稳启动，那么在我们进入下面的主循环之前应该可以观察到相当多的子进程立即陆续退出，其中 的原因则是因为我们向它们发出了 AP_SIG_GRACEFUL 信号。这一切发生的非常的快。对于每一个退出的子进程，我们都将启动一个新的进程来替换它 直到进程数目达到 ap_daemons_min_free。因此

```
    restart_pending = shutdown_pending = 0;
    mpm_state = AP_MPMQ_RUNNING;

    while (!restart_pending && !shutdown_pending) {
        int child_slot;
        apr_exit_why_e exitwhy;
        int status, processed_status;
        apr_proc_t pid;
```

至此，主服务进程则可以进入循环，它所作的事情只有两件事情，一个是负责对服务器重新启动或者关闭，另一个就是 负责监控子进程的数目，或者关闭多余的空闲子进程或者在空闲子进程不够的时候启动新的子进程。restart_pending 用于指示服务器是否需要进行 重新启动，为 1

的话表明需要重启；shutdown_pending 则指示是否需要关闭服务器，为 1 则表明需要关闭。除此之外，graceful 用于指示 是否进行平稳启动。当外界需要对主进程进行控制的时候只需要设置相应的变量的值即可，而主进程中则根据这些变量进行相应的处理。

```
ap_wait_or_timeout(&exitwhy, &status, &pid, pconf);
```

如果 restart_pending=0 并且 shutdown_pending=0 的话意味着外部进程不需要服 务器终止或者重新启动，此时主服务进程将进入无限循环，监视子进程。对于平稳启动而言，正常情况下，在每一轮循环中，主服务进程都会调用 ap_wait_or_timeout()等待子进程终止。通常情况下，子进程的退出有三种可能，分别枚举类型 apr_exit_why_e 进行描述

1．正常退出，此时 APR_PROC_EXIT＝1，这种情况通常是进程所有任务完成后退出

2．信号退出，此时 APR_PROC_SIGNAL=2，这种情况通常是进程在执行过程中接受到信号半途退出

3．非正常退出，此时 APR_PROC_SIGNAL_CORE＝4，通常是进程意外中断，同时生成 core dump 文件。

```
if (pid.pid != -1) {
    processed_status = ap_process_child_status(&pid, exitwhy, status);
    if (processed_status == APEXIT_CHILDFATAL) {
        mpm_state = AP_MPMQ_STOPPING; uvwxy
        return 1;
    }
```

主进程通过 ap_wait_or_timeout 监视等待每一个子进程退出，同时在 exitwhy 中保存它们退出的原因。尽管如此，主进程并不会无限制的等待下去。主进程会给出一个等待的超时时间，一旦超时，主进程将会不再理会那些尚未结束的进程，继续执行主循环 的剩余部分。如果子进程在规定的时间内完成，那么即等待成功，此时该被终止的子进程的 pid.pid 将不为-1，否则 pid.pid 将为-1。

一旦等待到子进程退出，那么进程退出的原因保存在 processed_status 中。如果 processed_satus 为 APEXIT_CHILDFATAL，则表明发生了致命性的错误，这时候将导致整个服务器的崩溃，此时主进程直接退出，不对记分板做任何的处理，如 u 所示；

```
    child_slot = find_child_by_pid(&pid);
    if (child_slot >= 0) {
        (void) ap_update_child_status_from_indexes(child_slot, 0,
SERVER_DEAD,
                                                   (request_rec *) NULL);u

        if (processed_status == APEXIT_CHILDSICK) {
            idle_spawn_rate = 1; v
        }
        else if (remaining_children_to_start
            && child_slot < ap_daemons_limit) {
            make_child(ap_server_conf, child_slot);
            --remaining_children_to_start;
        }
#if APR_HAS_OTHER_CHILD
    }
```

如果子进程发生的错误并不是致命性的，那么一切都得按部就班的处理——更新记分板中对应的插槽中的信息。首要的前提就是在记分板中找到该进程对应的插槽，这由函数find_child_by_pid()完成。
如果能够成功找到终止进程对应的插槽，那么直接在记分板中将该终止进程的状态更新为SERVER_DEAD，这样，该插槽将会再次可用，如u所示。
如果进程退出是因为资源受限，比如磁盘空间不够，内存空间不够等等，此时Apache必须降低生成子进程的速度至最低。如v所示。
如果Apache进行的是平稳重启，那么在进入主循环之前，通过发送终止信号，很多的子进程都将被终止，这些被 终止的进程在系统重启后必须被新的进程替换，直到总的进程数目达到daemons_min_free。 remaining_children_to_start记录了当前需要重启的子进程的数目。

```
            else if (apr_proc_other_child_alert(&pid, APR_OC_REASON_DEATH, status)
                        == APR_SUCCESS) {
#endif
            }
            else if (is_graceful) {
                ap_log_error(APLOG_MARK, APLOG_WARNING,
                             0, ap_server_conf,
                             "long lost child came home! (pid %ld)", (long)pid.pid);
            }
            continue;
        }
```

如果进程在公告板中没有找到相应记录，此时检查子进程是否是“其余子进程”(reap_other_child)。 一些情况下，主服务进程会创建一些进程，这些进程并不是用来接受并处理客户端连接的，而是用作其余用途，通常称之为“其余进程”，并用一个单独的列表进行 登记。比如，一般情况下，Apache会将日志写入到文件中，但是有的时候Apache则希望将数据写入到一个给定的应用程序中。因此主服务器进程必须为 该应用程序创建该进程，并且将该进程的标准输入STDIN关联道主服务进程的日志流中。这种进程并不是用来执行处理HTTP请求的子服务进程，因此称之为 “其余进程”。任何时候只要服务器重启，日志应用进程都会接受到SIGHU和SIGUSR1信号，然后终止退出，对应的模块必须重新创建这种进程。对于其 余进程，主服务进程不做任何事情，因为这不是主进程所管辖的范围。
如果既不是“其余子进程”，又没有在公告板中找到相应记录，同时管理员还设置了热启动选项，那么发 生这种情况只有一个可能性：管理员减少了允许的子进程的数目同时强制执行了热启动。而一个正在忙碌的子进程拥有的入口记录号比允许的值大。此时它终止的时 候自然就不可能在公告板中找到相应的记录入口。

```
            else if (remaining_children_to_start) {
                startup_children(remaining_children_to_start);
                remaining_children_to_start = 0;
                continue;
            }
```

如果当所有的终止的子进程都被替换之后，remaining_children_to_start还不为零，此时意味着主服务进程必须创建更多的子进程，这个可以通过函数startup_children()实现。

```
            perform_idle_server_maintenance(pconf);
    #ifdef TPF
            shutdown_pending = os_check_server(tpf_server_name);
```

```
            ap_check_signals();
            sleep(1);
        #endif /*TPF */
        }
        } /* one_process */
```

一旦启动完毕，那么主进程将使用 perform_idle_server_maintenance 进入空闲子进程维护阶段，同时主进程还得监视相关的信号，比如关闭信号，重启信号等等。空闲进程的维护在 4.2.1.2 中详细描述。

当主进程退出循环 while (!restart_pending && !shutdown_pending) 的时候只有两种情况发生，或者被通知关闭，或者被通知重启。一旦如此，Apache 将着手进行相关的清除工作。

```
        mpm_state = AP_MPMQ_STOPPING;

        if (shutdown_pending) {
            if (unixd_killpg(getpgrp(), SIGTERM) < 0) {
                ap_log_error(APLOG_MARK, APLOG_WARNING, errno, ap_server_conf,
                                        "killpg SIGTERM");
            }
           ap_reclaim_child_processes(1);          /* Start with SIGTERM */

            /* cleanup pid file on normal shutdown */
            {
                const char *pidfile = NULL;
                pidfile = ap_server_root_relative (pconf, ap_pid_fname);
                if ( pidfile != NULL && unlink(pidfile) == 0)
                    ap_log_error(APLOG_MARK, APLOG_INFO,
                                 0, ap_server_conf,
                                 "removed PID file %s (pid=%ld)",
                                 pidfile, (long)getpid());
            }

            ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
                        "caught SIGTERM, shutting down");
            return 1;
        }
```

如果 Apache 需要进行关闭，那么 Apache 的清除工作工作包括下面的几个方面：

■ 如果服务器被要求关闭，那么主服务进程将向整个进程组中的所有子进程发送终止信号，通知其调用 child_exit 正常退出。

■ 调用 ap_reclain_child_process 回收相关的子进程。

■ 清除父子进程之间通信的“终止管道”。

```
        apr_signal(SIGHUP, SIG_IGN);
        if (one_process) {
            /* not worth thinking about */
            return 1;
        }
```

```
        ++ap_my_generation;
        ap_scoreboard_image->global->running_generation = ap_my_generation;

        if (is_graceful) {
            ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
                         "Graceful restart requested, doing restart");

            /* kill off the idle ones */
            ap_mpm_pod_killpg(pod, ap_max_daemons_limit);

            /* This is mostly for debugging... so that we know what is still
             * gracefully dealing with existing request. This will break
             * in a very nasty way if we ever have the scoreboard totally
             * file-based (no shared memory)
             */
            for (index = 0; index < ap_daemons_limit; ++index) {
                if (ap_scoreboard_image->servers[index][0].status != SERVER_DEAD) {
                    ap_scoreboard_image->servers[index][0].status =
SERVER_GRACEFUL;
                }
            }
        }
```

如果 Apache 被要求的是重新启动，那么对于平稳启 动和非平稳启动处理则不太相同。对于平稳启动而言，主进程需要终止的仅仅是那些目前空闲的子进程，而忙碌的进程则不进行任何处理。空闲进程终止通过 ap_mpm_pod_killpg 实现；同时由于记分板并不销毁，因此对于那些终止的进程，必须更新其在记分板中的状态信息为 SERVER_DEAD； 而对于那些仍然活动的子进程，则将其状态更新为 SERVER_GRACEFUL。

```
        else {
           /* Kill 'em off */
            if (unixd_killpg(getpgrp(), SIGHUP) < 0) {
                ap_log_error(APLOG_MARK, APLOG_WARNING, errno, ap_server_conf,
"killpg SIGHUP");
            }
            ap_reclaim_child_processes(0);          /* Not when just starting up */
            ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
                         "SIGHUP received. Attempting to restart");
        }
```

如果服务器被要求强制重启，那么所有的子进程包括那些仍然在处理请求的都将被统统终止。

Apache 中预创建 Preforking MPM 机制剖析(2)

6.3.3.4 空闲子进程维护

6.3.3.4.1 概述

主服务进程一方面除了必须维护平稳启动之外，另外一个最重要的职责就是对空闲子进程的数目进行管理，整个空闲管理功能在 perform_idle_server_maintenance()中描述。

空闲进程的整个内部是示意图可以用下面的图进行描述。

6.3.3.4.2 代码分析

```
static void perform_idle_server_maintenance(apr_pool_t *p)
{
    int i;
    int to_kill;
    int idle_count;
    worker_score *ws;
    int free_length;
    int free_slots[MAX_SPAWN_RATE];
    int last_non_dead;
    int total_non_dead;

    /* initialize the free_list */
    free_length = 0;
    to_kill = -1;
    idle_count = 0;
    last_non_dead = -1;
    total_non_dead = 0;
```

在分析具体的空闲子进程维护过程之前，函数中的几个重要的变量需要解释一下：

ap_daemons_limit，该值描述了当前 Apache 中允许存在的子进程的最多数目，或者是空闲进程、非空闲进程数目以及公告板中空闲的插槽数目的总和、或者是公告板中插槽的总和。

ap_daemons_max_free，该值描述了当前系统中允许存在的空闲子进程的最大数目。如果空闲子进程数目过多，那么在每一次循环中都会有一个空闲进程被杀死。

ap_daemons_min_free，该值描述了系统中至少必须存在的空闲子进程的数目。如果当前的空闲子进程数目过低，那么主服务进程将创建新的子进程。如果当前公告板中没有可用插槽(因为 ap_daemons_limit 已经达到)，此时将产生一个警告。

如果在短时间内系统中创建了太多的子进程，一些操作系 统可能会性能降低。因此主服务进程并不是立即调用 make_child()创建所需的所有子进程。相反，它采用递增创建的策略：在第一次循环中创建一个子 进程；在第二次循环中创建二个子进程，第三次创建四个，第 N 次创建 2^N 个。下一个循环中需要创建的子进程数目用变量 idle_spawn_rate 进行记录，每次循环中都会对该变量进行递增直到其达到极限值。

举例说明：如果给定的 ap_daemons_min_free 的值为 5，但是系统中的空闲进程数目仅为 1。因此主服务进程此时创建一个新的进程，同时进行等待。当然两个进程是 不够的，因此主进程在第二次循环中创建 2 个进程同时继续等待。如果此时一个新的客户端连接发生，那么一个空闲进程将转换忙碌进程。因此主服务进程统计的空 闲进程数目为 3，同时创建四个空闲进程。当超时的时候，主服务进程统计出 7 个空闲子进程同时重新设置 idle_spawn_rate 为 1。

ap_max_daemons_limit， Necessary to deal with MaxClients changes across AP_SIG_GRACEFUL restarts. 尽管Apache内部允许生成的最大进程数为ap_daemons_limit，但是实际上每次产生的进程数目不一定会有这么多。每 一个进程都对应记分板中的一个插槽。为了了解各个进程的状态，MPM必须逐一循环遍历记分板中的每一个插槽，这样共计ap_daemons_limit 次。显然一些无效的插槽也进行了遍历，这部分本来可以避免的。为此，MPM中使用ap_max_daemons_limit记录记分板中曾经使用的最大的 插槽号，一旦记录下来，遍历不再是从0到ap_daemons_limit，而是从0到ap_max_daemons_limit，可以省去ap_daemons_limit - ap_max_daemons_limit-1次的循环。这是一种优化策略。

last_non_dead与ap_max_daemons_limit的含义非常的相近。

idle_count：当前服务器中的空闲进程的数目。

totaol_non_dead：当前服务器中的活动进程的数目，包括空闲进程和非空闲进程。

last_non_dead：

idle_spawn_rate： 这是另外一个重要的变量。当MPM模块发现空闲子进程数目current_num少于ap_min_daemons_limit的时候，它将会产生足够的 进程ap_min_daemons_limit-current_num个。一般的操作系统允许一次性的产生所需要的进程。不过一些操作系统则不然，如果 一次性在很短的时间内产生大量的系统进程，操作系统性能会明显的降低，从而导致服务器响应变慢。这种情况要尽量避免。因此Apache并没有采用这种“激 进”的创建措施，而是采取了折衷的温和的逐步递增创建策略：在第一次循环中创建一个，第二次循环中创建二个，第三次循环中创建四个，第N次循环中创建2^N个，直到1+2+4+…+2^N值达到所需要的进程数。

idle_spawn_rate指示下一个循环中必须创建的进程数目，记住不是本次循环中。

两个变量就是free_length和free_slots，这两个变量我们在稍后描述。

只有对这些变量的含义有了清晰的认识之后我们才能进行分析。

```
        for (i = 0; i < ap_daemons_limit; ++i) {
            int status;

            if (i >= ap_max_daemons_limit && free_length == idle_spawn_rate)
                break;u
            ws = &ap_scoreboard_image->servers[i][0];
            status = ws->status;
            if (status == SERVER_DEAD) {
                if (free_length < idle_spawn_rate) {
                    free_slots[free_length] = i;
                    ++free_length;
                }
            }
            else {
                if (status <= SERVER_READY) {
                    ++ idle_count;
                    to_kill = i;
                }

                ++total_non_dead;
                last_non_dead = i;
            }
```

```
}
```

Apache 所作的第一件事情就是要统计当前运行的各类进程的信息，包括空闲进程，终止进程以及忙碌进程的数目。为此它必须能够逐一访问记分板中的每一个插槽并读取相应的进程信息其中我们最关心的就是进程的状态信息：

如果插槽的状态 SERVER_DEAD，则意味着对应 的进程已经终止，该插槽可以被再次利用；所有可以被再次利用的插槽统一的保存在 free_slot 数组中。数组中仅仅保存插槽的索引号。其能保存的最多数 目为 32 个，但是通常情况下不是所有的元素都会被使用，因此配合 free_slots 数组，free_length 用于记录当前的最高可用的元素的索引。 因此 free_slots 的操作更像是一个堆栈，每次的元素总是压入最顶部。

比如上图的 free_slots 就反映了当前记分板中插槽索引为 2，3，7，13，14 的进程已经终止。

如果插槽的状态为 SERVER_STARTING 或者是 SERVER_READY，则意味着当前的进程处于空闲状态；在遍历过程中，一旦发现空闲进程，idle_count 的值将递增 1，因此当上面的整个循环结束的时候，idle_count 则就是实际的空闲进程数。

另外在 Linux 中，进程编号如果较低的话，在调度的时候其被命中的概率也就越高，因此如果需要终止某些进程的话，Apache 会倾向于终止那些编号较高的进程，为此模块中使用 **to_kill** 变量跟踪当前的最高进程编号，这样下次终止进程直接终止 to_kill 指定的即可。

如果进程状态既不是 SERVER_DEAD，也不是 SERVER_READY，则意味着该线程正在处理客户端的请求。对于这些进程直接累计 total_non_dead 变量并设置 last_non_dead。

不过如前所叙，经过优化后，MPM 不需要遍历所有的记分板插槽了，只需要遍历 ap_max_daemons_limit 即可。

free_length == idle_spawn_rate 意味着

```
    ap_max_daemons_limit = last_non_dead + 1;
    if (idle_count > ap_daemons_max_free) {
        ap_mpm_pod_signal(pod);
        idle_spawn_rate = 1;
    }
```

一旦获取了系统中空闲进程的数目，模块则开始对进程进行调整：

ap_daemons_max_free 是 Apache 中允许的空闲进程的最大值，如果当前的空闲进程数目 idle_count 超过该值，那么多余的空闲进程必须退出。一般情况下，父进程可以强制 子进程立即退出，但是如果某些进程正在处理客户端的请求，那么该连接将会被粗鲁的终止，造成数据丢失，为此 Apache 使用“终止管道(Pipe of Death)”通知空闲子进程退出，这样需要退出的子进程可以执行平稳的退出，以防止连接数据丢失。ap_mpm_pod_signal 函数用于在终止管 道中写入终止数据。不过需要注意的是，每次循环只能退出一个子进程，因此如果需要终止的线程有 N 个，那么至少需要循环 N 次才能全部退出。**这种策略称之为“缓慢退出”，有利于防止进程“创建/终止”摆动。**

```
    else if (idle_count < ap_daemons_min_free) {
        if (free_length == 0) {
            static int reported = 0;
            if (!reported) {
                ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,
```

```
                             "server reached MaxClients setting, consider"
                             " raising the MaxClients setting"); u
                reported = 1;
            }
            idle_spawn_rate = 1;
        }
        else {
            if (idle_spawn_rate >= 8) {
                ap_log_error(APLOG_MARK, APLOG_INFO, 0, ap_server_conf,
                    "server seems busy, (you may need "
                    "to increase StartServers, or Min/MaxSpareServers), "
                    "spawning %d children, there are %d idle, and "
                    "%d total children", idle_spawn_rate,
                    idle_count, total_non_dead);   v
            }
```

如果当前空闲进程的数目低于允许的最少进程数目，那么 此时MPM必须增加空闲进程数目。在前面的部分，我们曾经说过，idle_spawn_rate表示本次循环中需要产生的子进程的数目。如果 idle_spawn_rate为8，则意味着本次循环需要产生8个子进程。至此，系统必须连续产生1+2+4+8=15个进程，显然如果出现这种情况则 意味着系统实在太忙了。这可能是因为初始启动的服务太低或者允许的空闲进程指标Min/MaxSpareServers太小，因此此时必须增加这些参数的 值，并在日志中写入警告。如v所示。

如果free_length=0，意味着当前free_slots中没有可用的插槽，这表明当前系统中的所有的进程都处于活动状态：或者忙碌或者空闲等待请求。这种情况的出现可能是因为MaxClient参数设置过低，因此有必要增加MaxClient的值。

```
            for (i = 0; i < free_length; ++i) {
#ifdef TPF
                if (make_child(ap_server_conf, free_slots[i]) == -1) {
                    if(free_length == 1) {
                        shutdown_pending = 1;
                        ap_log_error(APLOG_MARK, APLOG_EMERG, 0,
ap_server_conf,
                                     "No active child processes: shutting
down");
                    }
                }
#else
                make_child(ap_server_conf, free_slots[i]);
#endif /* TPF */
            }
```

尽管原则上第N次可以产生2^N的进程，但是实际上真正能够产生的进程数目还得由记分板中的空闲插槽数目free_length决定。生成具体的子进程使用make_child例程，在后面的部分会详细的对其进行分析。

```
            if (hold_off_on_exponential_spawning) {
                --hold_off_on_exponential_spawning;
```

```
            }
            else if (idle_spawn_rate < MAX_SPAWN_RATE) {
                idle_spawn_rate *= 2;
            }
```

在进程产生后紧接着的任务就是设置下一循环中需要产生的进程数目：idle_spawn_rate*2；当然这个值不能超出允许产生的最大值 MAX_SPAWN_RATE。

```
        }
    }
    else {
        idle_spawn_rate = 1;
    }
```

6.3.3.4.3 子进程创建

主进程的一个很重要的任务就是在空闲进程不够的情况下创建足够的子进程。子进程的创建在函数 make_child 中进行。

```
static int make_child(server_rec *s, int slot)
```

对于任何一个创建的子进程，其都必须在计分板中占据一 个插槽来保存自己的信息，slot 就是创建的进程在计分板中的插槽索引，不过有一点需要确保的 slot 的值不能超过系统中允许的进程极限数，即 slot 必 须满足 slot <= ap_max_daemons_limit -1

整个子进程的创建过程可以分割为下面二个步骤：

(1)、更新计分板。一旦进程创建，它的状态将被设置为 SERVER_STARTING，表明该进程开始运行。

(2)、调用 fork()真正生成子进程。如果生成子 进程失败，还得将原先的计分板 SERVER_STARTING 状态更新为 SERVER_DEAD。一旦更新完毕，该插槽将再次变得可用。当 fork 函数调 用失败的时候，为了防止系统不停的尝试去重新 fork 从而将 CPU 资源耗尽，因此一旦如果 fork 失败，那么 Apache 将等待 10 毫秒后再去尝试新的 fork。

对于子进程而言，一旦其创建完毕，除了正常了父子进程 之间的通信，子进程不应该再受到父进程的其余的无端打断，因此子进程必须重新 SIGHUP 和 SIGTERM 信号。任何时候子进程接受到 SIGHUP 和 SIGTERM 信号之后，除了退出之外，再退出之前还需要完成几个清除工作，包括：

- 关闭父子进程之间通信的“终止管道”
-

尽管父进程会发送 AP_SIG_GRACEFUL 给子进程，但子进程并不对其进行处理：apr_signal(AP_SIG_GRACEFUL, SIG_IGN);一旦子进程处理完所有的准备工作，其将开始进入子进程的内部处理过程 child_main()。

另一方面，在生成子进程之后，主进程则必须将子进程号填入记分板的相关插槽中：

```
ap_scoreboard_image->parent[slot].pid = pid;
```

如果服务器是单进程运行模式，那么处理的工程要简单的多。由于不涉及到创建子进程，因此实际上主进程本身直接进入内部循环操作。另外与多进程相比，它仅仅处理三种信号：SIGINT、SIGTERM、SIGQUIT。

另外一种子进程生成方式就是批量子进程生成，即函数 static void startup_children(int number_to_start)。number_to_start 是批量产生的进程的数目。

```
static void startup_children(int number_to_start)
```

```
{
    int i;
    for (i = 0; number_to_start && i < ap_daemons_limit; ++i) {
            if (ap_scoreboard_image->servers[i][0].status != SERVER_DEAD) {
                continue;
            }
            if (make_child(ap_server_conf, i) < 0) {
                break;
            }
            --number_to_start;
    }
}
```

在计分板中，如果某个进程的状态为 SRVER_DEAD，则意味着当前的记分板插槽可用。因此对于需要创建的 number_to_start 个进程，需要通过逐一遍历计分板从而才可以给创建的进程分配插槽。一旦分配成功，那么函数将调用 make_child 创建进程。

Apache 中预创建 Preforking MPM 机制剖析(3)

6.3.3.5 工作子进程管理

子进程通常被视为工作者，其组成了 HTTP 服务器的核心。它们负责处理对客户端的请求的处理。尽管多任务体系结构并不负责对请求的处理，不过他仍然负责创建子进程、对其进行初始化并且将客户端请求转交给它们进行处理。子进程的所有的行为都被封状在函数 child_main() 中。

6.3.3.5.1 子进程的创建

在深入到子进程工作的内部细节之前，我们有必要了解一下主服务进程是如何创建子进程的。事实上，从主服务进程的最后的代码中也可以看出，主服务进程是通过调用 make_child 函数来创建一个子进程的，该函数定义如下：

```
static int make_child(server_rec *s, int slot)
```

该函数具有两个参数，slot 是当前进程在记分板中的索引。

的代码主要用于处理单进程。如前所述，单进程主要用于调试。对于单进程服务器而言，唯一的进程就是主服务进程，因此不需要创建任何额外的子进程，需要处理的就是转换主服务进程的角色为子服务进程，这种转变包括两部分：

```
{
    int pid;


    if (slot + 1 > ap_max_daemons_limit) {
        ap_max_daemons_limit = slot + 1;
```

```
    }


    if (one_process) {
        apr_signal(SIGHUP, sig_term);
        apr_signal(SIGINT, sig_term);
        apr_signal(SIGQUIT, SIG_DFL);u
        apr_signal(SIGTERM, sig_term);
        child_main(slot);
        return 0;
    }
上面
```

1)、处理信号。

6.3.3.5.2 初始化、配置以及服务重启

```
 (void) ap_update_child_status_from_indexes(slot, 0, SERVER_STARTING
,
                                            (request_rec *) NULL
);



    if ((pid = fork()) == -1) {
        ap_log_error(APLOG_MARK, APLOG_ERR, errno, s, "fork: Unable
to fork new process");
        (void) ap_update_child_status_from_indexes(slot, 0, SERVER_
DEAD,
                                                   (request_rec *)
NULL);
```

```
            sleep(10);
            return -1;
        }
        if (!pid) {
            RAISE_SIGSTOP(MAKE_CHILD);
            AP_MONCONTROL(1);
            apr_signal(SIGHUP, just_die);
            apr_signal(SIGTERM, just_die);
            apr_signal(AP_SIG_GRACEFUL, stop_listening);
            child_main(slot);
        }

        ap_scoreboard_image->parent[slot].pid = pid;

        return 0;
    }
```

主服务进程使用 fork()创 建子进程。每个子进程都具有独立的内存区域并且不允许读取其余子进程的内存。因此由主服务器一次处理配置文件要比由各个子进程各自处理明智的多。配置文件 的相关信息可以保存在共享内存区域中，该内存区域可以被每一个子进程读取。由于不是每一个操作系统平台都支持共享内存的概念，因此主服务进程在创建子进程 之前处理配置文件。子服务进程通常是父进程的克隆，因此它们与主服务进程具有相同的配置信息而且从来不会改变。

任何时候，如果管理员想更改服务器配置。他都必须让主服务进程重新读取配置信息。当前存在的子进程所拥有的则是旧的配置信息，因此它们都必须被替换为新产生的子进程。为了避免打断正在处理的 HTTP 请求，Apache 提供了一种平稳启动的方法，该模式下允许子进程使用旧的配置信息进行处理直到其退出。

子进程的初始化可以在相应的 MPM 中找到，对于预创建 Preforking MPM 而言就是 child_main()。该函数包括下面的几个步骤：

调用 ap_init_child_modules()重新初始化模块：每一个模块都由主进程预先进行初始化。如果模块中分配系统资源或者决定于进程号，那么模块重新进行初始化就是必须的。

建立超时处理句柄：为了避免子进程的无线阻塞，Apache 对客户端请求使用超时处理。其中使用警告，警告的概念与信号的概念非常类似，通常将报警时钟设置为给定的时间，而当报警响起的时候系统将离开请求的处理。

循环内还有两种初始化：

清除超时设置：重置报警定时器

清除透明内存池：在请求响应循环中每个内存的分配都涉及到透明内存池。在循环的开始，内存池必须进行清理。

将公告板中的 status 设置为 ready。

ptrans 的创建，以及访问公告板，同时由于多个进程之间可能存在竞争，因此另外一个准备工作就是创建进程间的接受互斥锁。由于通常情况下，父进程都是使用 fork 生成子进程，此时子进程基本是父进程的克隆。一般情况下，Apache 的启动都是使用超级用户进行的，因此子进程实际上也就具有与父进程等同的操作权限，父进程能够访问的资源子进程都能够访问。

```
static void child_main(int child_num_arg)

{

    apr_pool_t *ptrans;

    apr_allocator_t *allocator;

    conn_rec *current_conn;

    apr_status_t status = APR_EINIT;

    int i;

    ap_listen_rec *lr;

    int curr_pollfd, last_pollfd = 0;

    apr_pollfd_t *pollset;

    int offset;

    void *csd;

    ap_sb_handle_t *sbh;

    apr_status_t rv;
```

```
    apr_bucket_alloc_t *bucket_alloc;


    mpm_state = AP_MPMQ_STARTING;


    my_child_num = child_num_arg;
    ap_my_pid = getpid();
    csd = NULL;
    requests_this_child = 0;


    ap_fatal_signal_child_setup(ap_server_conf);


    apr_allocator_create(&allocator);
    apr_allocator_max_free_set(allocator, ap_max_mem_free);
    apr_pool_create_ex(&pchild, pconf, NULL, allocator);
    apr_allocator_owner_set(allocator, pchild);
    apr_pool_create(&ptrans, pchild);
    apr_pool_tag(ptrans, "transaction");


    ap_reopen_scoreboard(pchild, NULL, 0);
    rv = apr_proc_mutex_child_init(&accept_mutex, ap_lock_fname, pc
hild);
    if (rv != APR_SUCCESS) {
        ap_log_error(APLOG_MARK, APLOG_EMERG, rv, ap_server_conf,
```

```
                    "Couldn't initialize cross-process lock in chi
ld");
        clean_child_exit(APEXIT_CHILDFATAL);
    }
```

每个子进程在真正处理请求之前，都必须进行相关的资源准备工作，包括信号设置，私有内存池

Apache 通常会将子进程的用户设置为一个普通的用户，比如 nobody 或者 WWWRun 之类从而来降低子进程的执行权限，原则上，子进程用户的权力应该尽可能的小。Unixd_setup_child 将用户的 ID 从正在运行父进程的用户改变为在配置文件中规定的用户，如果不能改变用户的 ID，子进程就立即退出。另外相关的初始化工作必须在 unixd_setup_child()调用之前进行，因为一旦子进程权限降低，一些只能超级用户进行的初始化可能无法正常进行。

```
if (unixd_setup_child()) {
        clean_child_exit(APEXIT_CHILDFATAL);
    }
```

但是子进程具有与父进程相同的权限具有一定的潜在的危险。由于网络连接通常由子进程直接处理，因此如果黑客通过某种权限控制了子进程，那么他就能够任意的控制系统的。因此通常在进行了资源准备工作之后，Ap_run_child_init 调用 child_init 挂钩进行子进程本身的初始化。

```
 ap_run_child_init(pchild, ap_server_conf);

 ap_create_sb_handle(&sbh, pchild, my_child_num, 0);

 (void) ap_update_child_status(sbh, SERVER_READY, (request_rec *) NU
LL);
```

当所有的准备工作结束以后，子进程可以与客户进行会话。

80，但是在 Apache 中，则允许服务器在多个端口上同时进行侦听，这些侦听端口用结构 ap_listen_rec 进行描述：

```
 listensocks = apr_pcalloc(pchild,
                           sizeof(*listensocks) * (num_listensocks));

for (lr = ap_listeners, i = 0; i < num_listensocks; lr = lr-
>next, i++) {
```

```
    listensocks[i].accept_func = lr->accept_func;
    listensocks[i].sd = lr->sd;
}


 pollset = apr_palloc(pchild, sizeof(*pollset) * num_listensocks);
 pollset[0].p = pchild;
for (i = 0; i < num_listensocks; i++) {
     pollset[i].desc.s = listensocks[i].sd;
    pollset[i].desc_type = APR_POLL_SOCKET;
     pollset[i].reqevents = APR_POLLIN;
 }
 mpm_state = AP_MPMQ_RUNNING;
  bucket_alloc = apr_bucket_alloc_create(pchild);
```

一般的情况下，服务器只会侦听固定的端口，比如

描述了绑定到该端口的套接字，而 bind_addr 则描述了套接字必需关联的地址。Accept_func 是一个回调函数，当从该侦听端口上接受到客户端连接的时候，该函数将被执行从而来处理连接。Active 用以描述当前端口是否处于活动状态。

```
struct ap_listen_rec {
    ap_listen_rec *next;
    apr_socket_t *sd;
    apr_sockaddr_t *bind_addr;
    accept_function accept_func;
    int active;
```

```
};
```

sd

对于服务器端的多个侦听端口，Apache 使用链表进行保存，因此 next 用以指向下一个侦听套接字结构。整个链表的用 ap_listen_rec 全局变量记录，因此沿着 ap_listen_rec 可以遍历所有的侦听套接字。与此同时，侦听端口的数目也保存在全局变量 num_listensocks 中。

上面的代码所作的事情无非就是生成指定的需要逐一遍历的文件结果集合。

任何时候，子进程如果要正常退出，其都必须由主进程通过“终止管道”通知，另一方面，子进程也将不停的检查终止管道。一旦发现需要退出，子进程将 die_now 设置为 1，这时候实际上就自动退出循环。相反，如果子进程不需要退出，那么它所作的事情只有一个，就是使用 poll 对所有的端口进行轮询，直到某个端口准备完毕，则调用相关的连结处理函数进行处理。

```
while (!die_now) {

        current_conn = NULL;

        apr_pool_clear(ptrans);

        if ((ap_max_requests_per_child > 0

            && requests_this_child+
+ >= ap_max_requests_per_child)) {

            clean_child_exit(0);

        }

        (void) ap_update_child_status(sbh, SERVER_READY, (request_re
c *) NULL);
```

SAFE_ACCEPT(accept_mutex_on());

虽然多个子进程同属于一个父进程，但是多个子进程之间则是相互并行的，当多个子进程同时扫描侦听端口的时候，很可能发生多个子进程同时竞争一个侦听端口的情况。因此所有的子进程有必要互斥的等待 TCP 请求。

接受互斥锁能够确保只有一个子进程独占的等待 TCP 请求(使用系统调用 accept())——这些都是侦听者所做的事情。接受互斥锁是控制访问 TCP/IP 服务的一种手段。它的使用能够确保在任何时候只有一个进程在等待 TCP/IP 的连接请求。

不同的操作系统有不同的接受互斥锁(Accept Mutex)的实现。有一些操作系统对于每一个子进程需要一个特殊的初始化阶段。它的工作方式如下：

调用过程 accept_mutex_on()：申请互斥锁或者等待直到该互斥锁可用

调用过程 accept_mutex_off()：释放互斥锁

prefork MPM 中通过 SAFE_ACCEPT(accept_mutex_on())实现子进程对互斥锁的锁定；而 SAFE_ACCEPT(accept_mutex_off())则是完成互斥锁的释放。

```
if (num_listensocks == 1) {
        offset = 0;
}
```

对于整个侦听套接字数组而言，任何时候只有一个侦听端口能被处理。Offset 实际上描述了当前正在被处理的侦听端口在数组中的索引。如果当前服务器的侦听端口只有一个，那么几乎没有任何事情要做，也就没有所谓的轮询。

如果服务器配置使用多个侦听端口，那么 Apache 就必须使用 poll()来确定客户正在连接哪个端口，然后我们就可以知道哪个端口正在受到访问，这样才能在这个端口上调用接受函数。如果轮询返回的值是 EBADF，EINTR 或者 EINVAL 之类的错误，那么轮询并不应该被终止，但是如果返回的不是这些错误，那么子进程应该调用 clean_child_exit 退出。

```
else {
            for (;;) {
                apr_status_t ret;
                apr_int32_t n;
                ret = apr_poll(pollset, num_listensocks, &n, -1);
                 if (ret != APR_SUCCESS) {
                     if (APR_STATUS_IS_EINTR(ret)) {
                        continue;
                    }
                ap_log_error(APLOG_MARK, APLOG_ERR, ret, ap_server_
conf,
                             "apr_poll: (listen)");
                clean_child_exit(1);
                }
```

尽管服务器可能会同时侦听多个端口，但有的时候各个端口的重要性并不是一样的。比如某个服务器开通了 80 和 8080 两个端口，但是可能频繁使用的是 80，而 8080 则只是偶尔使用，如果不加任何控制的话，8080 端口可能会被忽略。为此，大多数 MPM 都会记

住最后提供服务的那个端口，并且从这个端口开始搜索新的连接，通过这种方法，就可以确保服务器不会忽略任何端口。Last_pollfd 用于标记最后提供服务的端口。Curr_pollfd 是当前需要处理的端口的索引。

```
  curr_pollfd = last_pollfd;

                do {

                    curr_pollfd++;

                    if (curr_pollfd >= num_listensocks) {

                        curr_pollfd = 0;

                    }

                    if (pollset[curr_pollfd].rtnevents & APR_POLLI
N) {

                        last_pollfd = curr_pollfd;

                        offset = curr_pollfd;

                        goto got_fd;

                    }

                } while (curr_pollfd != last_pollfd);

                continue;

            }

        }
```

```
got_fd:
        status = listensocks[offset].accept_func(&csd,
                                                 &listensocks[offset], ptrans);
        SAFE_ACCEPT(accept_mutex_off());  /* unlock after "accept" */
        if (status == APR_EGENERAL) {
            clean_child_exit(1);
        }
        else if (status != APR_SUCCESS) {
```

```
            continue;
        }
```

上面的代码用于接受客户端的连接。事实上，从上面的代码可以看到，接受客户端连接并没有直接使用 apr_accept 函数，而是将其作为上面描述的 ap_listen_rec 结构组成部分的函数指针。接受函数会进行有效的错误检查，并返回有效的套接字。使用这种函数指针策略的还会得到额外的好处。模块可以在侦听套接字列表中增加他们自己的通信原语。如果这些套接字不是正常的套接字，比如是 UNIX IPC 套接字，那么他们可能就需要不同的函数进行处理，此时，accept_func 指着就可以指向这些函数。增加这么代码，不仅可以允许在内核缓存和 Web 服务器之间进行通信，而且可以用于其余的方面，例如，许多 UNIX MPM 都同通过 POD 向子进程发出关闭信号，如前所述，如果将 POD 强行编入代码中，它就不容易维护，而使用 accept_func 函数，POD 就可以实现特殊的处理函数，实现无缝处理。

一旦接受了连接请求，一个子进程将释放互斥锁同时处理请求——此时它变成一个工作者，而下一个进程将继续申请互斥锁从而进行等待。这通常称之为 Leader-follower 模式：侦听者是 leader，而空闲的工作者则是 follower。由于 Apache 实现互斥锁使用的是操作系统相关技术，因此有些操作系统上，当一个子进程接受到连接释放互斥锁的时候，当前的所有的阻塞的子进程都将被唤醒。如果是这样，那么一些过分的调度将是不必要的，因此只有一个子进程会得到互斥锁，而其余的都将继续被阻塞睡眠。这个问题是 Leader MPM 需要解决的，在该 MPM 中，所有的 follower 进行一定的组织，从而当互斥锁释放的时候，只有它们中间的一个会被唤醒。

一旦子进程接受到一个客户端连接，那么多任务模块的职责也就结束了。子进程将继续调用请求处理例程进行处理。不管对于任何 MPM，它们都是一样的。

```
current_conn = ap_run_create_connection(ptrans, ap_server_conf, csd,
my_child_num, sbh, bucket_alloc);
if (current_conn) {
        ap_process_connection(current_conn, csd);
        ap_lingering_close(current_conn);
}
```

客户端连接一旦接受成功，此时就在客户端和服务器之间存在一条 TCP 连接。ap_process_connection 会处理这个连接上的所有的请求，然后退出。完成这些工作的第一步就是建立连接结构，在该结构中会存储客户端的套接字，以及所有的连接相关信息，比如，服务器的 IP 地址和唯一标示符 ID。一旦连接确定好，它就可以通过 ap_process_connection 接受服务了。ap_process_connection 的内部隐藏了相当多的细节，这些细节，我们在后面的部分会详细介绍。

在每个请求的最后阶段，我们都要调用 ap_lingering_close。这个函数据说是 Apache 中最糟糕的函数之一，这不仅是因为它很难理解，而且也涉及到了许多的 OS 的问题。必须要解决的问题是，在客户承认已经接受到所有的响应数据之前，都不可以关闭连接的服务器端。如果这样做了，那么客户就会丢失你最后发送的数据包。为了防止这种情况的发生，就必须要保持连接处于打开状态，直到出现超时或者客户端关闭连接。大多数 OS 都可以设置套接字选项来实现延迟关闭。遗憾的是，对于 Web 服务器而言，套接字选先不会总是进行了设置，与此相反，服务器需要实现延迟关闭，并且确保在每个连接结束时被调用。为了完成这项工作，核心服务器需要为请求的终止注册清除程序，以确保可以调用 lingering_close。然而，类似 Window 的一些 OS 可以让你在某些条件下重用套接字。如果你打算重用套接字，那么就不需要将其关闭，因而也就不需要调用 lingering_close 函数。如果你正在编写用鱼支持重用套接字的 MPM，ap_process_connection 之后的程序就可以删除。

```
if (ap_mpm_pod_check(pod) == APR_SUCCESS) { /* selected as idle? */
            die_now = 1;
        }
        else if (ap_my_generation !=
                 ap_scoreboard_image->global->running_generation) { /*
restart? */
            die_now = 1;
        }
    }
    clean_child_exit(0);
```

每次连接处理完毕，子进程都必须检查终止管道。如果父进程通知它退出，那么此时 die_now=1，下次子进程直接跳出循环执行 clean_child_exit 退出。

另一个可能导致子进程立即退出的原因就是该子进程属于上一家族的残留子进程。这通常是因为非强制启动引起的。因此每个子进程在退出之前要将自己的家族号 ap_my_generation 与父进程的家族号即记分板中的 running_generation 进行对比。如果不符合，表示子进程不属于本家族，将立即退出。除此两种情况之外，子进程将继续循环。

Apache 中的配置指令概述

//本文是《Apache 源代码全景分析》第二卷《体系结构和核心模块》中的第八章《配置文件管理》的草稿部分中，主要描述 Apache 中的指令概 念，在后续的章节中我们将继续深入 Apache 中的配置文件的处理细节，包括 Apache 如何读取命令行参数，如何读取配置文件，如何执行配置文件中的指 令以及如何存储指令等等。

第七章 配置文件管理

Apache 作为一个强大的非常灵活的 Web 服务器，配置文件功不可没，通过修改和调整配置文件，用户可以将 Apache 的功能发挥到极限。事实上，大部分的 Apache 管理员的工作就是调整 Apache 的配置文件，调整指令的参数。但是 Apache 的配置文件也是庞大的，指令的数目就够令人望而生畏，而且还在不断的扩充之中。尽管目前有很多的 Apache 的管理宝典之类的书指导用户如何使用这些指令，但是大部分的系统管理员对配置文件的机制已经指令的内部运行并不能很好的理解。它们并不了解指令是如何对 Apache 产生效果的，因此指令的使用也仅仅是人云亦云，简单模范而已。

古人云：知其然而知其所以然。或者说“授之以鱼，不如授之以渔”。本章我们将详细的对 Apache 的配置文件进行了深入的剖析，同时我们将追踪配置指令的作用流程，从而明白配置文件是如何产生效果的，部分的内容我们可能需要放到下一章结合模块部分描述。

4.1 Apache 配置系统

4.1.1 配置系统概述

在第七章我们描述 worker MPM 的时候曾经提到过系统中每个进程所能产生的线程数目并不是任意的，而是通过指令 ThreadsPerChild 指定的，同时系统中所能产生的进程数目也不是无限的，这由指令 ServerLimit 指令指定，比如

ThreadsPerChild 25
ServerLimit 16

上面的指令指定每个进程所能产生的线程数目为 25 个，而进程的最大产生数目为 16。

那么这些指令应该保存在哪儿？Apache 是什么时候读取这些指定的？它是怎么读取的？读取之后这些指令保存在哪儿，怎么保存的？这些指令最终是如何影响 Apache 的行为的呢？这些都是 Apache 配置系统需要解决的问题.

从整体上描述 Apache 配置系统，它应该包含三个主要部分：

1)、配置文件。通常情况下，配置系统会指定一些固定的文件作为配置文件，比如目前最主要的配置文件就是 httpd.conf。

2)、配置指令。配置系统必须能够决定各个指令的含义，这样配置系统才能够正确的对其进行解释和处理。配置正确的指令或者是默认的值，或者由管理员进行修改；而解释配置指令则由 Apache 的核心以及各个模块来处理。

3)、

本章我们先重点描述前两个部分，在模块章节中我们描述第三部分。

4.2 配置文件

Apache 服务器的配置是通过文本格式的配置文件来实现的，在文本文件中包含逐条的配置指令，正是通过这些逐条的指令从而实现对 Apache 运行的方方面面进行控制。在 Apache2.0 中涉及到的配置文件包括下面的两种：

■ httpd.conf

httpd.conf 是 Apache 中最重要的配置文件，通常位于$ServerRoot 下的 conf 目录中。不过在一些特殊的发行版本中，可能并不是这个名字，比如在许多支持 SSL 的 Apache 二进制发行版本中都会将二进制文件命名为 httdsd，与之对应，配置文件也相应的改名为 httdsd.conf。不管名称如何，文件内部的指令都是一样的。Httpd.conf 是默认的配置文件，一般情况下不建议对其进行修改，因此通常的建议是你重新拷贝一份，对该拷贝进行修改，因此这种情况下，你可以在指令行中使用-f 参数来指定新的非默认的配置文件。

从 Apache 1.3.13 起通过-f 指令不仅可以指定配置文件，还可以指定配置目录，即，如果配置文件是一个目录，Apache 会解析该目录及其子目录中的所有文件作为配置文件。一种可能的用途是，可以通过在这个目录中建立小的配置

文件来设置虚拟主机，这样就可以简单地增加和删除虚拟主机，而不用修改其他任何文件，使类似操作的自动化容易了许多。
通常，在服务器启动的时候，该文件被读取处理一次，同时在每次重新启动的时候又会被处理一次，因此对配置文件的任何修改都要等待到服务器重启后才能生效。
■ .htaccess
httpd.conf 文件通常用于控制全局的配置信息，但是有的时候 Apache 需要提供目录级别的控制，比如定制特定目录被访问或者被列表显示等等。尽管 httpd.conf 内部提供了相关的目录配置指令，但是如果需要控制的目录数目较大的话，httpd.conf 无疑会急剧膨胀。因此 Apache 中提供了另外的一种目录级别的配置，就是.htaccess。通常情况下，.htaccess 文件位于需要进行控制的目录之内，因此系统中可能存在多个.htaccess。每个.htaccesss 文件都有能力为它所处的目录以及所有的子目录设置授权、目录索引、过滤器以及其余的各种相关指令。因为.htaccess 文件总是包含在用户自己的共享目录文档中，因此用户完全可以建立、更新和修改自己的.htaccess 文件，而不需要直接去修改 httpd.conf 文件，从而可以保证 httpd.conf 的安全性，你要知道，每个人都去修改 httpd.conf 的话，造成的问题，你可能甚至都无法预料。
■ access.conf 和 srm.conf[1]
在 Apache1.3 以前的版本中，除了 httpd.conf 和.htaccess 之外，还有两个相关的配置文件，就是 access.conf 和 srm.conf。access.conf 用于配置服务器的访问权限，控制不同用户和计算机的访问限制，srm.conf 是服务器的资源映射文件，告诉服务器各种文件的 MIME 类型，以及如何支持这些文件。这两个文件都是从 NCSA 服务器继承而来的，具体的文件可以通过 httpd.conf 中的 AccessConfig 和 ResourceConfig 指令进行指定。不过从 Apache1.3 开始，这两个文件就已经废弃不用了，因此在 Apache2.0 中你看不到这两个文件，不过如果你非要设置 AccessConfig 和 ResourceConfig 指令，那么你可以将它们设置为“/dev/null”。
除了这三个设置文件之外，Apache 还使用 mime.types 文件用于标识不同文件 。其文件名由 TypesConfig 指定，缺省时是 mime.types。对应的 MIME 类型， magic 文件设置不同 MIME 类型文件的一些特殊标识，使得 Apache 服务器从文档后缀不能判断出文件的 MIME 类型时，能通过文件内容中的这些特殊标记来判断文档的 MIME 类型。
图 4.1 描述了各个配置文件在整个请求中的的位置。
图 4.1 配置文件处理
从上图中我们可以看出四个配置文件的处理时机是不一样的：在 Apache 启动或者重新启动的时候三个文件 httpd.conf，access.conf 以及 srm.conf 都会被处理，而.htaccess 只有在特定的 HTTP 请求到来的时候才有可能被处理。
图 4.2 配置文件在整个 Apache 中的位置

4.3 指令相关概念

4.3.1 指令概述

由于 Apache 只定义了一些配置的框架和配置段规则，因此 Apache 配置文件的结构通常很容易理解。每个可用的指令以及指令的参数并不是由 Apache 核心决定的，而是由模块完全决定并实现和控制。因此，一般情况下，Apache 配置文件的结构可以使用如下的语法片段进行描述：

configuration ::= directive*

directive ::= section-directive | simple-directive

section-directive ::= section-open configuration section-close

section-close ::= "<"directive-name directive-argument">"*

*simle-directive ::= directive-name directive-argument**

directive-name ::= "directory" | "documentroot"|…

directive-argument ::= …

换句话说，一个 Apache 配置文件可能是一个空文件，或者是包含了一个或者多个配置的指令，每个指令包含指令名称以及指令所需要的参数。指令的名称唯一的标识该指令本身，指令参数差异性则很大，参数类型，参数的个数都不尽相同，下面的是一个指令片断：

```
……
ServerRoot "C:/Program Files/Apache Group/Apache2"
TimeOut 30
<Directory "C:/Program Files/Apache Group/Apache2/manual">
    Options Indexs
    <Files *.html>
        SetHandler type-map
    </Files>
</Directory>
……
```

从上面的片断可以看出，指令只是用于控制 Apache 的简单的命令而已，Apache 从配置文件中读取这些指令，然后执行相应的操作从而实现执行这些指令。通过使用指令，Apache 管理员可以控制整个 Web 服务器的行为。由于 Apache 中提供了种类繁多的指令，这些指令使得的 Apache 是一个高度可配置的 Web 服务器。

Apache 的指令可以分为两种：简单指令以及配置段指令。配置段指令都是被包含在"<…>"中的指令，比如<Directory>…</Directory>。配置段指令总是会包含其余的指令。

尽管从上面的语法我们可以看到 Apache 的指令的语法非常的复杂，但实际上却非常的简单。在 Apache 进行指令处理的时候，Apache 将逐行的读取这些配置指令，如果某行不是空行(即不匹配正则表达式"^[\t]*$")，同时也不是一个注释行(不匹配正则表达式"^[\t]*#.*$")，那么 Apache 将该行的第一个单词视为指令字，后面的其余的单词全部算作参数。如果某个行以"\"结尾，则下一行视上一行的继续。

因此，Apache 配置指令的规则可以概括如下：
对于配置文件中的指令，其规则如下：
■ 使用 UNIX 路径法则：在所有的路径中使用“/”代替 DOS 下的“\”作为路径的分隔符。
■ 所有的注释行以“#”开始，同时注释行必须在一行结束，如果一行注释容不下，下一行必须继续以“#”开始。
■ 配置文件对大小写不敏感。但建议对非关键字均小写，而关键字则使用匈牙利方法，比如 ServerRoot、TypesConfig 等等。
■ 每行只能配置一个参数，配置的基本格式为：
参数　　参数值
■ 如果指令过长，不能够在一行中完整地放置，此时需要分割成为多行，各个行之间用 \ 进行组合。如果使用 \ 字符，那么在反斜杠和行的末尾不能存在任何内容和字符，包括空格或者水平制表符。
■ 系统将忽略配置文件中多余的空白字符。

4.3.2 指令参数

4. 3. 2. 1 参数类型

从原则上讲指令的参数可以是任何的字符串，只要指令处理函数能够理解即可。不过对于一些常用的指令参数，Apache 中有一些默认的规定。一般，指令名称后面可以跟一个或多个用空格分开的参数。如果参数中有空格，则必须用双引号括起来， 用方括号括起来的是可选的参数。如果一个参数可以取多个值，则各个可能的值用"|"分开。 应该原样输入的文字使用缺省的字体，而可变的必须按实际情况加以替换的会加强显示。 使用可变参数个数的指令以"..."结尾，以示最后一个参数可以重复。
指令的参数类型非常多，以下列出很常用的部分：

(1) URL

一个完整的包括类型、主机名和可选的路径名的统一资源引用名，如 http://www.example.com/path/to/file.html

(2) URL-path

即 url 中类型和主机名之后的部分，如/path/to/file.html. url-path 是表示资源在网络空间而不是在文件系统中的位置。

(3) file-path

即文件在本地文件系统中相对于根目录的路径，如 /usr/local/apache/htdocs/path/to/file.html. 除非指定了其他的值，不以斜杠开头的 file-path 将被视为对 ServerRoot 的相对路径。

(4) directory-path

即目录在本地文件系统中相对于根目录的路径，如 /usr/local/apache/htdocs/path/to/.

(5) filename

即不带路径信息的文件名，如 file.html.

(6) regex

正则表达式，是对文本匹配模式的描述。指令的定义中会说明应该使用什么 regex.

(7) extension

一般是指 filename 中最后一个"."号后面的部分。 但是，Apache 可以辨认多个文件后缀，如果 filename 含有多个"."， 则第一个"."后面由每个"."分隔开的部分都是此文件的后缀。 比如 filename，file.html.en 有两个后缀：.html 和.en. 在 Apache 指令中指定 extension 时，可以有也可以没有前导的"."，而且不区分大小写。

(8) MIME-type

一种用一个主格式类型和一个副格式类型并用斜杠分隔的描述文件格式的方法，如 text/html，img/jpeg 等等。

(9) env-variable

这是 Apache 配置进程中定义的环境变量的名称。 注意，它不一定与操作系统中的环境变量相同。比如：

SetEnvIf Referer "^http://www.example.com/" local_referal

SetEnvIf Referer "^$" local_referal

<Directory /web/images>
Order Deny,Allow
Deny from all
Allow from ***env=local_referal***
</Directory>

上面的代码就使用了环境变量 local_referal，使用的时候必须通过 env=xxxx 进行指定。

4.3.2.2 参数默认值

如果该指令有默认值(即，如果你没有在配置中明确指定， 那么 Apache 网站服务器会设置一个特定的值，并认为它是你设置的)，会在此处说明。 如果没有，则会指明是"None"。注意，此处的默认值并不一定与服务器发行版中默认的 httpd.conf 中该指令的取值相同。

4.3.2.3 配置段指令

4.3.3 指令上下文

4.3.3.1 上下文介绍

配置文件中的各个指令的作用范围是不一样的，可以分为**全局指令**，**局部指令**以及**条件指令**。默认情况下，配置文件中的指令是作用于整个服务器的，比如上面的示例中的 ServerRoot 和 TimeOut 指令，它们的作用范围则是针对整个服务器而言，但并不是所有的指令都这样，有些指令只是针对某个特定的目录，文件或者 URL 地址，通常情况下，那么我们将这类指令称之为局部指令，这类指令总是嵌在相关的配置指令段中，比如<Directory>, <DirectoryMatch>, <Files>, <FilesMatch>, <Location>, 以及 <LocationMatch>，比如上面的示例片断中<Directory "C:/Program Files/Apache Group/Apache2/manual">…</Directory>中的指令仅仅对目录 C:/Program Files/Apache Group/Apache2/manual 中的文件产生作用。
另外还有一些指令并不是针对某个目录的，而是在特定的条件下才会产生效果的，我们将它们称之为条件指令。
类似于<Director>的这类指令我们称之为容器指令或者称之为配置段指令。比如<IfDefine>，<IfModule>以及<IfVersion>等等。
局部指令和条件指令都是以<…>…</…>之间，我们将这两种指令称之为配置段指令。
一个指令它所能影响的范围以及它产生效果的条件，我们称之为指令的上下文，在用户使用任何一个核心指令之前，了解指令能够使用的上下文环境是一件非常重要的工作，换句话说，你必须能够知道指令的作用上下文或者指令的范围。

4.3.3.2 主服务器上下文(Server Config)

如果指令的上下文是主服务器，那么它能够作用的范围将是配置文件中容器配置段之外的所有的范围，即可以出现在 httpd.conf，srm.conf 以及 access.conf，但却不能出现在<VirtualHost>或者<Directory>配置指令片断中。该指令也不能出现在.htaccess 文件中。

4.3.3.3 局部上下文(Local Config)

局部上下文通常是指某个虚拟主机，某个目录，某个 URI 以及某个 Location，这种上下文之间的关系可以用下图进行描述。
图 4.3 配置指令上下文
从上图可以看出，局部上下文可以分为两大类，一种是直接通过局部配置段指定，另外一种则是通过.htaccess 文件进行指定。
最常用的配置段是针对文件系统和网络空间特定位置的配置段。首先必须理解文件系统和网络空间这两个概念的区别，文件系统是指操作系统所看见的磁盘视图，比如，在 Unix 文件系统中，Apache 会被默认安装到/usr/local/apache2 ，在 Windows 文件系统中，Apache 会被默认安装到"C:/Program Files/Apache Group/Apache2"(注意：Apache 始终用正斜杠而不是反斜杠作为路径的分隔符，即使是在 Windows 中)。相反，网络空间是网站被 web 服务器发送以及被客户在浏览器中所看到的视图。所以网络空间中的路径/dir/ 在 Apache 采用默认安装路径的情况下对应于 Unix 文件系统

中的路径/usr/local/apache2/htdocs/dir/ 。由于网页可以从数据库或其他地方动态生成，因此，网络空间无须直接映射到文件系统。

文件系统容器

<Directory>和<Files>指令与其相应的正则表达式版本(<DirectoryMatch>和<FilesMatch>)一起作用于文件系统的特定部分，<Directory>配置段中的指令作用于指定的文件系统目录及其所有子目录，.htaccess 文件可以达到同样的效果。下例中，/var/web/dir1 及其所有子目录被允许进行目录索引。

```
<Directory /var/web/dir1>
        Options +Indexes
</Directory>
```

<Files>配置段中包含的指令总是应用于特定的文件而无论这个文件实际存在于哪个目录，指定的文件可以是普通的文件名称，另外可以使用<FilesMatch>指定正则表达式，是正则表达式的文件名称。下例中的配置指令如果出现在配置文件的主服务器段，则会拒绝对位于任何目录下的 private.html 的访问。

```
<Files private.html>
        Order allow,deny
        Deny from all
</Files>
```

<Files>和<Directory>段的组合可以作用于文件系统中的特定文件。下例中的配置会拒绝对 /var/web/dir1/private.html ，/var/web/dir1/subdir2/private.html，/var/web/dir1/subdir3/private.html 等任何 /var/web/dir1/ 目录下 private.html 的访问。

```
<Directory /var/web/dir1>
        <Files private.html>
            Order allow,deny
            Deny from all
        </Files>
</Directory>
```

网络空间容器

<Location>指令与其相应的正则表达式版本(<LocationMatch>)一起作用于网络空间的特定部分。<Location>或者<LocationMatch>中包含的指令通常应用于特定的 URL 或者它的一部分，URL 可以是普通的 URL 地址，也可以是正则表达式格式的 URI。

如果指令的作用范围是仅仅限于.htaccess 文件中，那么该指令的上下文应该是属于针对目录级别的。当 Apache 处理 HTTP 请求从而遍历文件系统的时候进行读取并将指令作用于对应的目录。该上下文也被细分为五种子上下文，这些上下文在 httpd.conf 中的 AllowOverride 指令允许的时候发生作用。

下例中的配置会拒绝对任何以"/private"开头的 URL 路径的访问，比如：http://yoursite.example.com/private、http://yoursite.example.com/private123、

http://yoursite.example.com/private/dir/file.html 等所有以"/private"开头的 URL 路径。

```
<Location /private>
        Order Allow,Deny
        Deny from all
</Location>
```

<Location>指令与文件系统无关，下例演示了如何将特定的 URL 映射到 Apache 内部的处理器 mod_status ，而并不要求文件系统中确实存在 server-status 文件。

```
<Location /server-status>
        SetHandler server-status
</Location>
```

<Directory>和<Files>都提供了正则表达式版本的<Directory>,<Files>, 和<Location>指令可以使用类似 C 标准库中的 fnmatch 的外壳通配符。符号"*"匹配任何字符串，"?"匹配任何单个的字符，"[*seq*]" 匹配 *seq* 序列中的任何字符，符号"/"不匹配为任何通配符所匹配，所以不能显式使用。

这些指令都有一个正则的配对指令，<DirectoryMatch>, <FilesMatch>和<LocationMatch>，可以使用与 perl 一致的正则表达式，以提供更复杂的匹配。但是还须注意下文配置的合并中有关使用正则表达式会如何作用于配置指令的内容。

下例使用非正则表达式的通配符来改变所有用户目录的配置：

```
<Directory /home/*/public_html>
        Options Indexes
</Directory>
```

下例使用正则表达式一次性拒绝对多种图形文件的访问：

```
<FilesMatch \.(?i:gif|jpe?g|png)$>
        Order allow,deny
        Deny from all
</FilesMatch>
```

虚拟主机空间容器<VirtualHost>

包含的指令应用于特定的虚拟主机，这些虚拟主机之间通过唯一的 IP 地址和端口对进行区分。

.htaccess 中的隐含上下文

除了上面的显式的指令上下文之外，上下文之间还包含了隐含的上下文关系，包括：

■ .htaccess 文件上下文 AuthConfig 和 Limit 通常总是包含服务器配置<Directory>，<Files>和<Location>配置段。

■ .htaccess 文件上下文 Options，FileInfo 以及 Indexs 总是包含所有的针对服务器的配置，即整个 httpd.conf。

除此之外，<Location>和<Files>上下文中允许的指令与<Directory>中的指令一样对待。

为了明确的表示指令的作用上下文，Apache 在 http_config.h 中定义了相关的常量来进行描述：

```
#define  NOT_IN_VIRTUALHOST     0x01 /**< Forbidden in <Virtualhost> */
#define  NOT_IN_LIMIT           0x02 /**< Forbidden in <Limit> */
#define  NOT_IN_DIRECTORY       0x04 /**< Forbidden in <Directory> */
#define  NOT_IN_LOCATION        0x08 /**< Forbidden in <Location> */
#define  NOT_IN_FILES           0x10 /**< Forbidden in <Files> */
/** Forbidden in <Directory>/<Location>/<Files>*/
#define  NOT_IN_DIR_LOC_FILE    (NOT_IN_DIRECTORY|NOT_IN_LOCATION|NOT_IN_FILES)
/** Forbidden in <VirtualHost>/<Limit>/<Directory>/<Location>/<Files> */
#define  GLOBAL_ONLY    (NOT_IN_VIRTUALHOST|NOT_IN_LIMIT | NOT_IN_DIR_LOC_FILE)
```

同时还定义了函数 ap_check_cmd_context 用于检查指令是否出现在其应该出现的上下文。关于该该函数，我们在后面第五章详细描述。

4.3.3.4 条件上下文

Apache 中允许设定某些指令在特定的条件下才产生效果。这三种上下文主要是指**<IfDefine>**，**<IfModule>**以及**<IfVersion>**。

<IfDefine>容器中的指令只有在 httpd 命令行中设定了特定的参数后才有效。下例中，只有在服务器用 httpd -DClosedForNow 方式启动时，所有的请求才会被重定向到另一个站点：

```
<IfDefine ClosedForNow>
        Redirect / http://otherserver.example.com/
</IfDefine>
```

<IfModule>容器很相似，但是其中的指令只有当服务器启用特定的模块时才有效(或是被静态地编译进了服务器，或是被动态装载进了服务器)，注意，配置文件中该模块的装载指令 LoadModule 行必须在出现在此容器之前。这个容器应该仅用于你希望无论特定模块是否安装，配置文件都能正常运转的场合；而不应该用于容器中的指令在任何情况下都必须生效的场合，因为它会抑制类似模块没找到之类的有用出错信息。

下例中，MimeMagicFiles 指令仅当 mod_mime_magic 模块启用时才有效。

```
<IfModule mod_mime_magic.c>
        MimeMagicFile conf/magic
</IfModule>
```

<IfVersion>指令与<IfDefine>和<IfModule>很相似，但是其中的指令只有当正在执行的服务器版本与指定的版本要求相符时才有效。这个模块被设计用于测试套件、以及在一个存在多个不同 httpd 版本的大型网络中需要分别针对不同版本使用不同配置的情况。

```
<IfVersion >= 2.1>
        # 仅在版本高于 2.1.0 的时候才生效
</IfVersion>
```

<IfDefine>、<IfModule>、<IfVersion>都可以在条件前加一个"!"以实现条件的否定，而且都可以嵌套以实现更复杂的配置。

4.3.3.5 上下文嵌套关系

Apache 中各个上下文之间所能影响的范围并不是相同的，相反，它们之间有着严格的包含差异关系，通常各个上下文之间的包含关系从大到小可以用下图描述：

从上图中可以看出，包含范围大小依次为<VirtualHost>、<Directory>或者<Location>、<Files>以及<Limit>。因此各种上下文在嵌套中必须遵循下面的嵌套规则：

■ <Directory>配置段不允许出现在<Limit>，<Location>，<Files>或者其余的<Directory>配置段之间

■ <Location>配置段不允许出现在<Limit>，<Directory>，<Files>以及其余的<Location>配置段之间

■ <Files>配置段不允许出现在<Limit>，<Directory>，<Location>以及其余的<Files>配置段之间

■ <Directory>和<Location>配置段不允许出现在.htaccess 文件中，但是<Files>则允许出现。

更加详细的嵌套关系可以用下表进行描述：

	<VirtualHost>	<Directory>	<Location>	<Files>	<Limit>	.htaccess
<VirtualHost>	×	×	×	×	×	×
<Directory>	√	×	×	×	×	×
<Location>	√	×	×	×	×	×
<Files>	√	√	×	×	×	√
<Limit>	√	√	×	√	×	√

从语义上看，允许在<Directory>段中使用的指令当然也可以在<DirectoryMatch>、<Files>, <FilesMatch>、<Location>、<LocationMatch>、<Proxy>和<ProxyMatch>段中使用，但是有几个例外：

■ AllowOverride 指令只能出现在<Directory>段中；

■ Options 中的 FollowSymLinks 和 SymLinksIfOwnerMatch 只能出现在<Directory>段或者.htaccess 文件中；
■ Options 指令不能用于<Files>和<FilesMatch>段。

4.3.3.6 上下文合并和继承

配 置段会按非常特别的顺序依次生效，由于这会对配置指令的处理结果产生重大影响，理解它的流程尤为重要。另外，如果不同的配置段中同时出现相同的指令，那么 这些指令之间也存在不同的产生作用的方式，包括完全覆盖，继承合并。覆盖的概念很好理解，继承合并遵循一定的规则，合并的顺序是：
<Directory> (除了正则表达式)和.htaccess 同时处理；(如果允许的话，.htaccess 的设置会覆盖<Directory>的设置)
<DirectoryMatch> (和<Directory ~>)
<Files>和<FilesMatch>同时处理；
<Location>和<LocationMatch>同时处理；
除了<Directory>，每个组都按它们在配置文件中出现的顺序被依次处理，而<Directory>组，会按字典顺序由短到长被依次处理。例如，<Directory /var/web/dir>会先于<Directory /var/web/dir/subdir>被处理。如果有多个指向同一个目录的<Directory>段，则按它们在配置文件中的顺序被依次处理。用 Include 指令包含进来的设置被视为按原样插入到 Include 指令的位置。位于<VirtualHost>段中的配置段在外部相对应的段处理完毕*以后*再处理，这样就允许虚拟主机覆盖主服务器的设置。
后面的段覆盖前面的相应的段。
这是一个假设的演示合并顺序的例子。如果这些指令都起作用，则会按 A > B > C > D > E 的顺序依次生效。

```
<Location />
        E
</Location>

<Files f.html>
        D
</Files>

<VirtualHost *>
        <Directory /a/b>
            B
        </Directory>
</VirtualHost>

<DirectoryMatch "^.*b$">
        C
</DirectoryMatch>
```

```
<Directory /a/b>
        A
</Directory>
```

在这个更具体的例子中，无论在<Directory>段中加了多少访问限制，由于<Location>段将会被最后处理，从而会允许不加限制的对服务器的访问，可见合并的顺序是很重要的，千万小心！

```
<Location />
        Order deny,allow
        Allow from all
    </Location>
    # Woops! This <Directory> section will have no effect
    <Directory />
        Order allow,deny
        Allow from all
        Deny from badguy.example.com
    </Directory>
```

通常情况下各个指令完全独立，这意味着指令的次序并不重要，大多数指令都不会影响服务器怎样解释其余的指令，不过也有例外，有些指令相互之间还是具有依赖性的，比如 LoadModule 指令，当配置文件读取每一行指令的时候，都会尝试在当前已经加载到服务器的模块中找到该指令。如果你试图在加载实现指令的模块之前就使用指令，那么服务器将会输出错误信息，并且退出。

4.3.3.7 指令位置

从前面的讨论中我们已经知道了指令上下文的概念。一个指令能够出现的位置即它能出现在哪些上下文环境中我们称之为指令位置控制。Apache 中提供了指令位置字段的概念来控制一个指令所允许出现的上下文位置。位置字段主要用于控制各个指令在配置文件中允许出现的位置，包括三种：顶层位置，目录区和虚拟主机区。如果服务器发现一个指令不允许出现在出现的地方，比如 LoadModule 只允许出现在顶层位置，如果发现其在<Directory>中出现，服务器将报错，同时打印错误信息退出。另外位置字段还将控制指令是否允许在文件.htaccess 中使用。

对于指令位置字段，Apache 中提供了下面几个控制选项：

```
#define OR_NONE                 0
#define OR_LIMIT                 1
#define OR_OPTIONS        2
#define OR_FILEINFO       4
#define OR_AUTHCFG        8
#define OR_INDEXES        16
#define OR_UNSET          32
#define ACCESS_CONF       64
#define RSRC_CONF         128
#define EXEC_ON_READ      256
```

#define OR_ALL (OR_LIMIT|OR_OPTIONS|OR_FILEINFO|
OR_AUTHCFG|OR_INDEXES)
对于上面的选项，Apache 又分成两类：普通配置文件说明选项和.htaccess 文件说明选项。

ACCESS_CONF

该位置字段允许指令出现在 Directory 或者 Location 区间以内的顶级配置文件中，因此该选项通常用来对 Apache 中目录或者文件进行某些控制。

RSRC_CONF

该选项允许指令出现在 Directory 或者 Location 区间以外的顶级配置文件中，当然也可以出现在 VirtualHost 区间中，因此该选项通用用来对 Apache 服务器或者虚拟主机的进行某些控制。

EXEC_ON_READ

该选项是 Apache2.0 中新增加的。在 Apache1.3 中，对指令的处理是边读边执行的，而 Apache2.0 中并不是这样。Apache2.0 首先预处理配置文件，将所有的配置指令读取到一个树型结构中，树的每个结点为 ap_directive_t 类型。我们称之为配置树。一旦建立配置树，Apache 然后才会遍历并处理所有指令。通常情况下，这种处理方式会很好，因为延后处理使的可以控制模块之间的依赖性。比如，线程化的 MPM 需要在定义 MaxClients 指令之前就必须定义 ThreadsPerChild 指令。MPM 不会强求管理员处理这种情况，即使在配置文件中 ThreadsPerChild 定义在 MaxClients 之后，Apache 也会在分析之前在配置树中进行次序调整。

不过延后处理也不是完美无缺，有的时候可能导致问题。比如，Include 通常用于在配置文件中读取另外一个配置文件。如果不能立即读取到配置文件，那么第二个配置文件中的配置指令将不可能生成到配置树中。解决的方法就是在读取到 Include 指令的时候取消滞后策略，而是立即执行该指令。EXEC_ON_READ 选项可以强制服务器在将指令从配置文件中读取之后立即执行。

通过前面的分析，我们可以看出，如果某个指令可能会改变配置树，那么该指令就应该为 EXEC_ON_READ，不过所谓的改变配置树不是指改变配置树的顺序，而是改变配置树的信息。如果想要改变配置树的次序，那么应该在预先配置阶段进行或者在处理配置指令时候完成这样的工作。

除了上面的三个用于对配置文件进行控制，Apache 中还提供了八个选项用于控制.htaccess 文件中指令。

OR_NONE

该选项则不允许在.htaccess 文件中使用任何指令，这是默认值。不过大多数指令都会对其进行修改。只有那些仅仅在顶级配置文件中才有效的指令以及那些仅仅有服务器管理员才可以使用的指令才会设置该选项。

OR_LIMIT

只有那些可能涉及虚拟主机访问的指令才会设置该选项。在核心服务器上，Allow，Deny 以及 Order 指令都会设置该选项。使用这个选项的指令允许放置在 Directory 和 Location 标签中，以及 AllowOverride 设置为 Limit 的.htaccess 文件中。

OR_OPTIONS

使用该选项的指令通常用来控制特定的目录设置。在标准的 Apache 发行版本中 Options 指令和 XbitHack 指令都会使用该选项。使用该选项的指令必须置于

Directory 和 Location 以及 AllowOverride 设置为 Options 的.htaccess 文件中。

OR_FILEINFO

使用该选项的指令通常用于控制文档类型或者文档信息。设置该选项的指令包括 SetHandler，ErrorDocument，DefaultType 等等。这中类型的指令可以存在于 Directory 和 Location 标签以及 AllowOverride 设置为 FileInfo 的.htaccess 文件中。

OR_AUTHCFG

使用该选项的指令通常用语控制授权或者认证等信息。设置该选项的指令包括 AuthUserFile，AuthName 以及 Require。这种指令可以存在于 Directory 和 Location 以及 AllowOverride 设置为 AuthConfig 的.htaccess 文件中。

OR_INDEXS

使用该选项的指令通常用于控制目录索引的输出。示例指令包括 AddDescription，AddIcon，AddIconByEncoding。这种指令可以存在于 Directory 和 Location 以及 AllowOverride 设置为 Indexs 的.htaccess 文件。

OR_ALL

这个选项是前面所有选项的组合。使用该选项的指令可以存在于 Directory 和 Location 标签中，以及只要 AllowOverride 不为 None 的.htaccess 文件中。

OR_UNSET

这个特殊的值指出，这个目录没有设置重写。模块不应该使用值。核心会使用这个值来正确的控制继承。

Apache 中使用 req_override 记录指令的位置字段，目前在配置文件中，对于<Directory>标签外部的部分，req_override 状态为：
RSRC_CONF|OR_OPTIONS|OR_FILEINFO|OR_INDEXS
而在<Directory>标签内部的部分，状态为：
ACCESS_CONF|OR_LIMIT|OR_OPTIONS|OR_FILEINFO|
OR_AUTHCFG|OR_INDEXS
而在.htaccess 文件中，状态则由 AllowOverride 指令决定。

4.3.4 指令参数类型

Apache 中不同指令的参数差异很大，从没有参数到多个参数不等。为了能够准确的处理各种指令以及它的参数，Apache 中使用指令参数类型来标识一个指令的参数。不同的指令参数类型指导指令处理程序如何处理指令的参数。

Apache 中提供了 12 种类型的指令，这些类型是与实际的配置文件中指令处理相一致的。每种指令都大同小异，唯一的区别就在于其**处理的参数的数目以及在将指令传递给指令实现函数之前，服务器如何解释这些参数的方式**。由于各个指令的参数不相同，为此也导致了指令的处理函数的格式不相同。

Apache 中对于指令类型的定义是通过枚举类型 cmd_how 来实现的，
cmd_how 定义如下：

```
enum cmd_how {
    RAW_ARGS,
    TAKE1,
    TAKE2,
```

```
    ITERATE,
    ITERATE2,
    FLAG,
    NO_ARGS,
    TAKE12,
    TAKE3,
    TAKE23,
    TAKE123,
    TAKE13
};
```

对于所有的指令处理函数，其都将返回字符串。如果指令处理函数正确的处理了指令，那么函数返回 NULL，否则应该返回错误提示信息。对于各种指令，服务器的处理方法如下：

RAW_ARGS

该指令会通知 Apache 不要对传入的参数做任何的处理，只需要原封不动的传递给指令处理函数即可。使用这种指令会存在一定的风险，因为服务器不做任何的语法检查，因此难免会有错误成为“漏网之鱼”。

这种指令的处理函数通常如下所示：

```
const char * func(cmd_parms* parms , void* mconfig , char* args);
```

args 是传入的参数，其只是简单的指令行内容，当然也包括指令在内，对于该参数 func 函数不做任何检查，直接使用。

TAKE1

顾名思义，这种类型的指令“Take 1 argument”，其只允许传入一个参数。这种指令的处理函数通常如下所示：

```
const char * func(cmd_parms* parms , void* mconfig , const char*
first);
```

first 则是需要传入的第一个指令参数。

ITERATE

该类型指令属于迭代类型。这种指令允许传入多个参数，不过一次只能处理一个，服务器必须遍历处理它们。每次遍历处理的过程又与 TAKE1 类型指令相同。因此这种指令的处理函数与 TAKE1 指令相同：

```
const char * func(cmd_parms* parms , void* mconfig , const char*
first);
```

由于一次只能处理一个参数，因此如果指令具有 N 个参数的化，则 func 函数必须调用 N 次，每次将第 N 个参数传入给函数。

TAKE2，TAKE12，ITERATE2

TAKE2 类型必须向指令处理函数传入两个参数；而 TAKE12 可以接受一个或者两个参数。ITERATE2 与 ITERATE1 类似，都属于参数迭代处理类型，不过 ITERATE2 要求至少传递两个参数给函数。不过，第二个参数能够使用多次，服务器会遍历它们，直到所有的参数都传递给处理函数。如果只向 TAKE12 传递一个参数，那么服务器将把第二个参数设置为 NULL。这三种类型的指令处理函数原型如下：

```
const char *two_args_func(cmd_parms* parms , void*
mconfig,const char* first,const char* second);
```

对于 TAKE2 类型而言，函数 two_args_func 只被调用一次，两个参数一次性传递给函数；对于 TAKE12 而言，如果只传递一个参数的话，第二个参数将被设置为 NULL，如果传递两个参数的话，与 TAKE2 相同；对于 ITERATE2 而言，如果有 N 个参数的话，则处理函数至少将调用 N-1 次。

TAKE3，TAKE23，TAKE13，TAKE123

这组指令最多都可以接受 3 个参数，如果参数超过三个，则处理函数将会报错。TAKE3 意味着参数必须是三个；TAKE23 则意味着至少两个参数，也可以为三个，不能少于两个或者多于三个。TAKE13 则意味这可以接受一个参数或者三个参数，除此之外都是非法。TAKE123 意味着可以接受一个，两个或者三个参数。这四种指令的处理函数原型如下：

```
const char *three_args_func(cmd_parms* parms , void*
mconfig,const char* first,const char* second,const char* third);
```

NO_ARGS

该类型的指令不接受任何的参数，其最常用的就是作为复杂指令的闭标签存在。比如<Directory>总是必须有</Directory>与之对应。尽管<Directory>通常需要一个参数来指定其所设置的目录，不过对于</Directory>则没有这个参数。因此其就是 NO_ARGS 指令类型。这种指令的处理函数通常如下所示：

```
const char *no_args_func(cmd_parms* parms , void* mconfig);
```

FLAG

这种类型最简单，其只允许用来启动或者关闭的指令。与前面的几种类型中，服务器直接将配置指令后的参数传递给函数不同，这种指令不会直接传递参数，而是首先对参数进行处理，并将处理的结果 true 或者 false 作为进一步的参数传递给函数。这种指令的处理函数通常如下所示：

```
const char *flag_args_func(cmd_parms* parms , void* mconfig,int
flag);
```

不管是什么指令，其对应的处理函数都是以两个参数开始：cmd_parms* parms 和 void* mconfig。cmd_parms 结构用来存储处理配置指令时候所需要的辅助内容。在处理任何配置信息文件的时候，该结构都将被创建。其用于 Apache 核心传递各种参数给指令处理方法。关于 cmd_parms 的具体解释，我们在后面将给出。

另一个参数 void* mconfig 表示针对指令位置的配置记录，基于所遇到的指令位置的不同，该配置记录可以是服务器配置记录，也可以是目录配置记录。

---

[1] Apache 最早是从 NCSA HTTP 服务器继承而来，在 NCSA 服务器中，httpd.conf、access.conf 和 srm.conf 都被使用而且分工非常明确。Httpd.conf 用以保存通用的服务器配置指令；access.conf 则用于包含访问控制指令；srm.conf 则包含资源配置指令。

但是 Apache 从来都没有严格的区分这三个文件的使用，即使从最早的 Apache 版本算起，Apache 对待这三个配置文件也是一视同仁的，任何一个指令都可以包含在任何一个文件中。Apache 中之所以继续保留种两个文件，完全是为了保证向后兼容性。不过这种兼容性经常因此误会，不知道的用户通常会很谨慎，防

止指令放错在不该出现的文件中。因此Apache最终只能官方声明这两个文件作废。这就是为什么在Apache 2.0中你见不到它们的原因。
当然将httpd.conf的功能分割到多个文件中的想法却并不过时，而且有的时候甚至是必须的。为此Apache中提供了更通用的Include指令。具体的使用我们在后面的部分将详细描述。