```
// main->ngx_init_cycle
```

该段程序使用 for 循环结构对 Nginx 服务器中的模块进行遍历，如果模块的类型为 NGX_CORE_MODULE，即模块属于 core 模块，则调用 create_conf（cycle）建立 core 模块上下文，最后给该模块赋值对应的 cycle->conf_ctx 成员。create_conf 是函数指针，定义在 ngx_core_module_t 结构体中：

```
typedef struct {
   ngx_str_t           name;
   void                *(*create_conf)(ngx_cycle_t *cycle);
   char                *(*init_conf)(ngx_cycle_t *cycle, void *conf);
} ngx_core_module_t;
```

该指针实际指向函数 ngx_core_module_create_conf()，用于对 core 模块的 ngx_core_conf_t 结构的各个成员进行空间分配和置零（置空）：

```
// ngx_core_module_create_conf
static void *
ngx_core_module_create_conf(ngx_cycle_t *cycle)
{
   ngx_core_conf_t  *ccf;
   ccf = ngx_pcalloc(cycle->pool, sizeof(ngx_core_conf_t));  //从内存池给 core 模块配置结构体申请空间
   if (ccf == NULL) {
      return NULL;
   }
                                            //以下初始化 core 模块配置结构体的成员变量
   ccf->daemon = NGX_CONF_UNSET;
   ccf->master = NGX_CONF_UNSET;
   ccf->timer_resolution = NGX_CONF_UNSET_MSEC;
   ccf->worker_processes = NGX_CONF_UNSET;
   ccf->debug_points = NGX_CONF_UNSET;
   ccf->rlimit_nofile = NGX_CONF_UNSET;
   ccf->rlimit_core = NGX_CONF_UNSET;
   ccf->rlimit_sigpending = NGX_CONF_UNSET;
   ccf->user = (ngx_uid_t) NGX_CONF_UNSET_UINT;
   ccf->group = (ngx_gid_t) NGX_CONF_UNSET_UINT;
#if (NGX_THREADS)                              //多线程运行 Nginx(实际没有使用多线程)
   ccf->worker_threads = NGX_CONF_UNSET;
   ccf->thread_stack_size = NGX_CONF_UNSET_SIZE;
#endif

   if (ngx_array_init(&ccf->env, cycle->pool, 1, sizeof(ngx_str_t))
      != NGX_OK)
   {
      return NULL;
   }
   return ccf;
}
// ngx_core_module_create_conf
```

这里需要指出的是，ngx_core_conf_t 结构的其他成员，如 pid、oldpid、priority、cpu_affinity_n、cpu_affinity 等，这里没有对它们进行显式的初始化，而是在使用 ngx_pcalloc()函数进行内存分配时设置它们为 0 或者 NULL。解析配置文件时，这些成员将被一一赋值。

到此为止，ngx_init_cycle()函数对 cycle 结构的初始化工作就暂时告一段落了。接下来，该函数从内存池中为 ngx_conf_t 结构申请内存，并用 conf 指针指向该内存区，为解析和处理 Nginx 配置文件做准备工作：

```
// main->ngx_init_cycle
   ngx_memzero(&conf, sizeof(ngx_conf_t));
                                              //以下继续初始化 core 模块上下文结构
   conf.args = ngx_array_create(pool, 10, sizeof(ngx_str_t));
   if (conf.args == NULL) {
      ngx_destroy_pool(pool);
      return NULL;
   }
   conf.temp_pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);   //建立临时内存池
   if (conf.temp_pool == NULL) {
      ngx_destroy_pool(pool);
      return NULL;
   }
   conf.ctx = cycle->conf_ctx;
   conf.cycle = cycle;
   conf.pool = pool;
   conf.log = log;
   conf.module_type = NGX_CORE_MODULE;
   conf.cmd_type = NGX_MAIN_CONF;
// main->ngx_init_cycle
```

该段代码对 conf 结构主要成员进行空间分配和初始化。其中需要注意的是，函数使用 ngx_create_pool()函数建立了另一个大小为 16384 字节的临时内存池，该内存池专门用于配置文件的解析，解析结束时，该内存池即被释放。

12.2.5　解析配置文件

解析配置文件的主要过程放在一个 for 循环中，基本的工作原理是提取配置文件中的每条指令，检查其语法是否正确，解析具体配置内容。涉及的主要数据结构是 ngx_conf_t 结构体。下面我们通过源码分析进一步理解。

解析配置文件的入口源码为：

```
// main->ngx_init_cycle
   if (ngx_conf_param(&conf) != NGX_CONF_OK) {        //内部解析 Nginx 配置文件
      environ = senv;
      ngx_destroy_cycle_pools(&conf);
      return NULL;
   }
// main->ngx_init_cycle
```

代码中调用了ngx_conf_param()函数，我们进一步查看该函数的源码片段：

```
// main->ngx_init_cycle->ngx_conf_param
char *
ngx_conf_param(ngx_conf_t *cf)
{
    …
    param = &cf->cycle->conf_param;
    …
                                          //以下为初始化解析配置文件时要用到的保存配置
                                          //信息的结构体和指示解析位置的“游标”
    ngx_memzero(&conf_file, sizeof(ngx_conf_file_t));
    ngx_memzero(&b, sizeof(ngx_buf_t));
                                          //以下为初始化解析配置文件时用到的位置标识
    b.start = param->data;
    b.pos = param->data;
    b.last = param->data + param->len;
    b.end = b.last;
    b.temporary = 1;
    …
    conf_file.file.fd = NGX_INVALID_FILE;
    …
    cf->conf_file = &conf_file;
    cf->conf_file->buffer = &b;
    rv = ngx_conf_parse(cf, NULL);        //执行具体的配置文件解析工作
    cf->conf_file = NULL;
    return rv;
}
// main->ngx_init_cycle->ngx_conf_param
```

函数主要调用ngx_conf_parse()函数执行具体的配置文件解析工作。在整个配置解析过程中，每执行一次for循环，ngx_conf_parse()函数实际上会被调用两次，这里提到的是第一次。我们注意到，此处调用ngx_conf_parse()函数传入的第一个参数是cf结构，其主要成员cf->conf_file中存放的数据来源于cf->cycle->conf_param，继续往前追溯，实际上是来自启动Nginx服务器时指定的-g参数。第二个参数为NULL，也就是说，此次解析的内容为补充配置，实际上ngx_conf_parse()函数执行的主要工作有：

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse
char *
ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename)
{
    …
    if (filename) {
        fd = ngx_open_file(filename->data, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);
        …
    } else if (cf->conf_file->file.fd != NGX_INVALID_FILE) {
        type = parse_block;
```

```
    } else {
        type = parse_param;
    }
    for ( ;; ) {
        rc = ngx_conf_read_token(cf);                    //依次读出指令
        …
        if (cf->handler) {
            rv = (*cf->handler)(cf, NULL, cf->handler_conf);
            …
        }
        rc = ngx_conf_handler(cf, rc);                   //调用指令的处理函数
        …
    }
    …
}
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse
```

该段代码的主体是 for 循环，采用“读取一条、检查一条、解析一条”的方式解析配置文件。这里要注意，我们说的“一条指令”是以分号或花括号“{”、“}”为单位的，如果遇到嵌套的花括号，则先处理嵌套的指令。比如，以第 2 章中的配置实例为例，解析该配置文件时，首先以分号为标识符解析全局块中的每条指令，遇到 events 指令时，将调用 events 块的解析函数解析该配置块中的各条指令；同样在遇到 http 指令时，将调用 http 块的解析函数解析该配置块中的各条指令。当在 http 块中遇到 server 块时将首先调用 server 块的解析函数解析 server 块中的指令，之后再回到 http 块中继续解析剩余的指令。其他的配置块解析原理相同。

ngx_conf_read_token()函数将配置文件中当前的一条配置指令读取到内存中，判断配置语法是否使用正确，并将该条指令保存到 cycle->args 数组中。该函数通常可以检查以下几类情况语法是否正确：

- 配置指令末尾的分号“;”是否遗漏；
- 用于形成配置块作用域的花括号“{}”是否匹配；
- 注释符号、转移符号以及其他符号使用是否符合规范。

该函数的返回值我们需要关注一下：

- NGX_ERROR 出错
- NGX_OK 遇到分号“;”
- NGX_CONF_BLOCK_START 遇到花括号“{”
- NGX_CONF_BLOCK_DONE 遇到花括号“}”
- NGX_CONF_FILE_DONE 到达配置文件末尾

如果指令通过语法检查，程序首先检测该条指令是否是用户自定义指令，并调用相应的自定义处理函数进行解析。如果不是用户自定义指令，就调用 ngx_conf_handler()函数进行配置分析。

ngx_conf_handler()函数主要完成配置文件的解析工作，在传入的参数中，cf->args 成员中保存当前读取的一条指令，last 变量是 ngx_conf_read_token()函数的返回值。函数的主体是两个嵌套的 for 循环结构，具体的解析工作包含以下几点：

- 根据 cf->module_type 成员查找特定类型的模块；
- 遍历模块的指令数组 ngx_modules[i]->commands，匹配指令名称和类型；
- 检查指令的参数个数；
- 执行该条指令，即使配置生效。

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
static ngx_int_t
ngx_conf_handler(ngx_conf_t *cf, ngx_int_t last)
{
    char           *rv;
    void           *conf, **confp;
    ngx_uint_t      i, found;
    ngx_str_t      *name;
    ngx_command_t  *cmd;
    ......
    name = cf->args->elts;
    found = 0;
    for (i = 0; ngx_modules[i]; i++) {
        cmd = ngx_modules[i]->commands;
        ......
        for ( /* void */ ; cmd->name.len; cmd++) {
            ......
            if (ngx_strcmp(name->data, cmd->name.data) != 0) { //查找当前解析到的指令
                continue;
            }
            found = 1;                                         //查找成功
            if (ngx_modules[i]->type != NGX_CONF_MODULE
                && ngx_modules[i]->type != cf->module_type) //避免解析不同模块中的相同指令
            {
                continue;
            }
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
```

该段代码首先对当前模块 ngx_modules[i]的指令集进行遍历，如果找到与 cf->args 匹配的指令，则对该模块本身的类型做判断，检查该模块是否为指定的模块类型，直到匹配成功。如果当前模块的指令没有通过检查，遍历下一模块。

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
            if (!(cmd->type & cf->cmd_type)) {
                continue;
            }
            if (!(cmd->type & NGX_CONF_BLOCK) && last != NGX_OK) {
                ......
            }
            if ((cmd->type & NGX_CONF_BLOCK) && last != NGX_CONF_BLOCK_START) {
                ......
            }
```

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
```

在上段代码检查通过的基础上，该段代码通过指令的类型和 last 值判定当前指令是否配置在正确的地方。我们在前面相关章节中介绍每条指令时均强调该指令可以出现在配置文件中的位置。这里的代码就是用来检查当前指令是否处于配置文件的正确位置。

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
        if (!(cmd->type & NGX_CONF_ANY)) {
            if (cmd->type & NGX_CONF_FLAG) {                         //布尔参数
                if (cf->args->nelts != 2) {
                    goto invalid;
                }
            } else if (cmd->type & NGX_CONF_1MORE) {                 //1 个以上参数
                if (cf->args->nelts < 2) {
                    goto invalid;
                }
            } else if (cmd->type & NGX_CONF_2MORE) {                 //2 个以上参数
                if (cf->args->nelts < 3) {
                    goto invalid;
                }
            } else if (cf->args->nelts > NGX_CONF_MAX_ARGS) { //少于允许的最多参数
                goto invalid;
            } else if (!(cmd->type & argument_number[cf->args->nelts - 1]))
            {
                goto invalid;
            }
        }
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
```

该段代码比较容易理解。我们通过上一章的学习知道，Nginx 服务器程序对每条指令可以包含的参数个数使用宏定义规定。因此这段代码用来判断当前指令是否包含个数合法的参数，如果个数不合法，程序执行将转到 invalid 标号指向的错误处理程序。

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
        conf = NULL;
        if (cmd->type & NGX_DIRECT_CONF) {
            conf = ((void **) cf->ctx)[ngx_modules[i]->index];
        } else if (cmd->type & NGX_MAIN_CONF) {
            conf = &(((void **) cf->ctx)[ngx_modules[i]->index]);
        } else if (cf->ctx) {
            confp = *(void **) ((char *) cf->ctx + cmd->conf);
            if (confp) {
                conf = confp[ngx_modules[i]->ctx_index];
            }
        }
        rv = cmd->set(cf, cmd, conf);
        ……
    }
}
```

```
// main->ngx_init_cycle->ngx_conf_param->ngx_conf_parse->ngx_conf_handler
```

该段代码完成的主要工作是调用 cmd->set 指针指向的处理函数执行当前配置指令，使配置生效。在执行指令之前，是对 conf 变量的赋值。conf 是指向存放配置上下文结构的首地址，配置上下文具体是什么结构不用关心，反正是 void *，指向什么都可以。

从源码可以看到，指令被分为三个主要类型：

- NGX_DIRECT_CONF
- NGX_MAIN_CONF
- 其他

在这里再次强调的是，NGX_DIRECT_CONF 类指令拥有自己的初始化函数，其在解析之前就已经被初始化过，可以直接存储到对应的 ngx_conf_t 结构中。NGX_MAIN_CONF 类型指令没有自己的初始化函数，所以需要在 cmd->set 指针指向的处理函数中先进行初始化。

"其他"类型的指令，主要包括 server 块、location 块中的配置指令。我们在第 11 章学习配置上下文*ctx 时提到，这类指令通常通过指针以数组的形式保存在**main_conf 和**serv_conf 中，这样我们就可以理解这句源码：

```
confp = *(void **) ((char *) cf->ctx + cmd->conf);
```

其中的 cf->ctx 变量指向存放配置上下文数组的首地址，cmd->conf 为对应配置指令在数组中的偏移量。这样，confp 变量中存放的就是"其他"类指令配置上下文所在地址。

我们看到，配置上下文实际上并不是同一类数据结构，并且在层次关系上也不完全相同，因此也就可以理解为什么指向配置上下文的指针被声明为 void，并且包含多级指针。

对于 cmd->set 指针，我们在上一章学习 ngx_command_s 结构体时已经提到，并展示了 Nginx 服务器中基本类型模块各自定义的指令集数组。在处理 http 指令时，set 指针指向的函数为 ngx_http_block()，在处理 event 指令时，set 指针指向的函数为 ngx_events_block()等。对于这些函数的具体实现，鉴于篇幅所限，笔者不打算在这里进一步讲解，感兴趣的朋友可以进一步阅读相关的源码。

到这里，我们基本上将解析配置文件的关键函数 ngx_conf_parse ()和它调用的 ngx_conf_handler()函数学习完了。接下来我们再回到 ngx_init_cycle()函数。

```
// main->ngx_init_cycle
    if (ngx_conf_parse(&conf, &cycle->conf_file) != NGX_CONF_OK) {
        environ = senv;
        ngx_destroy_cycle_pools(&conf);
        return NULL;
    }
// main->ngx_init_cycle
```

前面我们提到，ngx_conf_parse()函数在解析配置文件的整个过程中一共调用了两次，这里是第二次。我们注意到，此次传入的参数与上次传入的不同，第二个参数为&cycle->conf_file，其中存放了 Nginx 服务器的配置文件路径。因此，此次调用该函数是对 Nginx 标准配置文件的解析。

到此，Nginx 配置的解析工作就全部完成了。我们接着来看 ngx_init_cycle()函数中进行的下一步主要工作。在解析配置文件之前，程序建立了 core 模块的上下文。现在可以根据 Nginx 的配置对 core

模块的上下文进行初始化了。

12.2.6 初始化 core 模块上下文

初始化 core 模块上下文结构，主要调用 ngx_core_module_t 结构体中定义的*(*init_conf)指向的初始化函数。*(*init_conf)实际指向的初始化函数是 ngx_core_module_init_conf()。我们大致浏览一下该函数执行的主要工作：

```
// ngx_core_module_init_conf
static char *
ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf)
{
    ngx_core_conf_t  *ccf = conf;
    ngx_conf_init_value(ccf->daemon, 1);
    ngx_conf_init_value(ccf->master, 1);
    ngx_conf_init_msec_value(ccf->timer_resolution, 0);
    ngx_conf_init_value(ccf->worker_processes, 1);
    ngx_conf_init_value(ccf->debug_points, 0);
#if (NGX_HAVE_CPU_AFFINITY)                          //定义了 Nginx 在多核 CPU 上的调度规则
    if (ccf->cpu_affinity_n
        && ccf->cpu_affinity_n != 1
        && ccf->cpu_affinity_n != (ngx_uint_t) ccf->worker_processes)
    {
        ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
                      "the number of \"worker_processes\" is not equal to "
                      "the number of \"worker_cpu_affinity\" masks, "
                      "using last mask for remaining worker processes");
    }
#endif
#if (NGX_THREADS)                                    //多线程下的初始化（实际没有使用）
    ngx_conf_init_value(ccf->worker_threads, 0);
    ngx_threads_n = ccf->worker_threads;
    ngx_conf_init_size_value(ccf->thread_stack_size, 2 * 1024 * 1024);
#endif
    if (ccf->pid.len == 0) {                         //设置 Nginx 进程的 PID
        ngx_str_set(&ccf->pid, NGX_PID_PATH);
    }
    if (ngx_conf_full_name(cycle, &ccf->pid, 0) != NGX_OK) {  //判断 Nginx 进程 ID 文件
的路径是否完整
        return NGX_CONF_ERROR;
    }
    ccf->oldpid.len = ccf->pid.len + sizeof(NGX_OLDPID_EXT);
    ccf->oldpid.data = ngx_pnalloc(cycle->pool, ccf->oldpid.len);
    if (ccf->oldpid.data == NULL) {
        return NGX_CONF_ERROR;
    }
     //更新旧进程 ID 文件中的数据
```

```
    ngx_memcpy(ngx_cpymem(ccf->oldpid.data, ccf->pid.data, ccf->pid.len),
               NGX_OLDPID_EXT, sizeof(NGX_OLDPID_EXT));
// ngx_core_module_init_conf
```

在上面的代码中，程序对 ngx_core_module_t 结构体中基本成员进行了初始化，并且创建了 Nginx 服务器程序的 PID 文件。ngx_conf_init_value 以宏的形式定义为：

```
#define ngx_conf_init_value(conf, default)        \
    if (conf == NGX_CONF_UNSET) {             \
        conf = default;                       \
    }
```

在该段代码中我们还应该注意的是 ccf->oldpid 成员和 ccf->pid 成员的赋值。从源码中可以看到，ccf->oldpid 成员实际是由 ccf->pid 成员的值尾部添加了 NGX_OLDPID_EXT 宏定义的字符串组成的。NGX_OLDPID_EXT 宏定义我们在/nginx/src/core/nginx.h 中可以找到：

```
#define NGX_OLDPID_EXT     ".oldbin"
```

为什么要存储这样一个 oldpid 呢？我们在第 2 章中提到，Nginx 服务器在平滑升级过程中，会对旧的 nginx.pid 文件添加后缀.oldbin，变为 nginx.pid.oldbin 文件。所以，这里的 ccf->oldpid 成员存放了平滑升级过程中 Nginx 服务器原 PID 文件的新文件名。

```
// ngx_core_module_init_conf
#if !(NGX_WIN32)
    if (ccf->user == (uid_t) NGX_CONF_UNSET_UINT && geteuid() == 0) {
        struct group   *grp;
        struct passwd  *pwd;
        ngx_set_errno(0);
        pwd = getpwnam(NGX_USER);                    //获取运行 Nginx 的用户的相关信息
        if (pwd == NULL) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "getpwnam(\"" NGX_USER "\") failed");
            return NGX_CONF_ERROR;
        }
                                                     //保存 Nginx 运行用户的相关信息
        ccf->username = NGX_USER;
        ccf->user = pwd->pw_uid;
        ngx_set_errno(0);
        grp = getgrnam(NGX_GROUP);                   //获取 Nginx 运行用户组的相关信息
        if (grp == NULL) {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "getgrnam(\"" NGX_GROUP "\") failed");
            return NGX_CONF_ERROR;
        }
        ccf->group = grp->gr_gid;                    //保存 Nginx 运行用户组的 ID
    }
// ngx_core_module_init_conf
```

该部分代码用于获取和设置运行 Nginx 服务器程序的用户名和用户组相关信息，并将这些信息保存到 ccf->username 成员、ccf->user 成员以及 ccf->group 成员中。

```
// ngx_core_module_init_conf
    if (ccf->lock_file.len == 0) {
        ngx_str_set(&ccf->lock_file, NGX_LOCK_PATH);  //保存新的nginx.lock文件的路径
    }
    if (ngx_conf_full_name(cycle, &ccf->lock_file, 0) != NGX_OK) {
        return NGX_CONF_ERROR;
    }
    ngx_str_t  lock_file;
    lock_file = cycle->old_cycle->lock_file;
    if (lock_file.len) {
        lock_file.len--;
        if (ccf->lock_file.len != lock_file.len
            || ngx_strncmp(ccf->lock_file.data, lock_file.data, lock_file.len) //更
新旧的nginx.lock文件
               != 0)
        {
            ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
                          "\"lock_file\" could not be changed, ignored");
        }
        cycle->lock_file.len = lock_file.len + 1;
        lock_file.len += sizeof(".accept");
        cycle->lock_file.data = ngx_pstrdup(cycle->pool, &lock_file);
        if (cycle->lock_file.data == NULL) {
            return NGX_CONF_ERROR;
        }
    } else {                                         //没有创建好的nginx.lock文件，重新创建
        cycle->lock_file.len = ccf->lock_file.len + 1;
        cycle->lock_file.data = ngx_pnalloc(cycle->pool, ccf->lock_file.len +
sizeof(".accept"));
        if (cycle->lock_file.data == NULL) {
            return NGX_CONF_ERROR;
        }
                                                     //使用原nginx.lock文件信息
        ngx_memcpy(ngx_cpymem(cycle->lock_file.data, ccf->lock_file.data,
                              ccf->lock_file.len), ".accept", sizeof(".accept"));
    }
#endif
    return NGX_CONF_OK;
}
// ngx_core_module_init_conf
```

该部分代码用于获取 nginx.lock 文件，并将其信息保存到 cycle->lock_file 成员中。lock_file 与 Nginx 服务器执行 accept()系统调用有关，在常用系统平台下不会使用到 lock_file 文件。

到此，core 模块上下文结构的初始化就全部完成了，我们回到 ngx_init_cycle()函数继续分析。

12.2.7 创建 PID 文件

ngx_init_cycle()函数的下一个主要工作是调用 ngx_create_pidfile()函数创建新的 PID 文件。该过程比较简单，值得注意的是在 Nginx 服务器平滑升级过程中对旧 PID 文件的处理。我们来看一下涉及的相关源码实现：

```
// main->ngx_init_cycle
old_ccf = (ngx_core_conf_t *) ngx_get_conf(old_cycle->conf_ctx, ngx_core_module);
if (ccf->pid.len != old_ccf->pid.len || ngx_strcmp(ccf->pid.data,
old_ccf->pid.data) != 0)
{
    if (ngx_create_pidfile(&ccf->pid, log) != NGX_OK) {    //创建新的 PID 文件
        goto failed;
    }
    ngx_delete_pidfile(old_cycle);                         //删除旧的 PID 文件
}
// main->ngx_init_cycle
```

从源码中可以看到，程序首先判断备份的 Nginx 程序 PID 与当前的程序 PID 是否相同，如果不同，会创建新的 PID 文件，并且将旧的删除。

ngx_init_cycle()函数接下来的工作是填充 cycle->pathes 数组，遍历 cycle->open_files 链表并打开文件，初始化 shared_memory 链表，遍历 cycle->listening 数组并打开所有监听 socket. cycle->pathes 数组。cycle->open_files 链表的遍历相对简单，笔者不在这里过多介绍。shared_memory 链表的填充涉及 Nginx 服务器程序共享内存的分配和管理，我们在专门的章节中介绍。这里重点介绍 cycle->listening 数组的遍历和监听 socket 的打开。

12.2.8 处理监听 socket

我们先来看处理监听 socket 的相关源码：

```
// main->ngx_init_cycle
  if (old_cycle->listening.nelts) {                      //旧的监听 socket 正在使用
    ls = old_cycle->listening.elts;
    for (i = 0; i < old_cycle->listening.nelts; i++) { //设置原有监听 socket 的标识
      ls[i].remain = 0;
    }
// main->ngx_init_cycle
```

如果从原来 init_cycle->listening 中继承过来的 socket 不为空，则这部分代码遍历原来 init_cycle->listening 中的所有 socket。因为要使用新的 cycle 结构，所以先将原来所有 socket 的 remain 标志位初始化为 0，再根据具体情况进行设置。

```
// main->ngx_init_cycle
    nls = cycle->listening.elts;
    for (n = 0; n < cycle->listening.nelts; n++) {
      for (i = 0; i < old_cycle->listening.nelts; i++) {
        if (ls[i].ignore) {
          continue;
```

```
            }
            if (ngx_cmp_sockaddr(nls[n].sockaddr, ls[i].sockaddr) == NGX_OK)
            {                                         //复制原来监听 socket 的标志信息
                nls[n].fd = ls[i].fd;
                nls[n].previous = &ls[i];
                ls[i].remain = 1;
                if (ls[n].backlog != nls[i].backlog) {
                    nls[n].listen = 1;
                }
                …
                break;
            }
        }
        if (nls[n].fd == -1) {                        //socket 未打开时，需要打开
            nls[n].open = 1;
        }
    }
// main->ngx_init_cycle
```

该段代码使用嵌套的两个 for 循环结构对 cycle 结构中的 socket 进行了筛选。如果 cycle->listening 中的 socket 同时存在于 init_cycle->listening 中，则直接继承原来的打开描述符，并将 init_cycle->listening 中该 socket 的 remain 标志置为 1。

```
// main->ngx_init_cycle
  } else {                                            //旧的监听 socket 没有使用
      ls = cycle->listening.elts;
      for (i = 0; i < cycle->listening.nelts; i++) {
          ls[i].open = 1;
      }
  }
// main->ngx_init_cycle
```

如果原来 init_cycle->listening 中继承过来的 socket 为空，就直接将 cycle->listening 中所有 socket 的 open 标志置为 1。

```
// main->ngx_init_cycle
  if (ngx_open_listening_sockets(cycle) != NGX_OK) { //打开 open 标志为 1 的监听 socket
      goto failed;
  }
  …
// main->ngx_init_cycle
```

ngx_open_listening_sockets()函数的主体是两个 for 循环结构的嵌套：

```
// main->ngx_init_cycle->ngx_open_listening_sockets
ngx_int_t
ngx_open_listening_sockets(ngx_cycle_t *cycle)
{
  …
  for (tries = 5; tries; tries--) {                   //每个 socket 最多尝试打开 5 次
```

```
    ...
    for (i = 0; i < cycle->listening.nelts; i++) {
        ...
        s = ngx_socket(ls[i].sockaddr->sa_family, ls[i].type, 0);
        ...
        if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR,      //配置 socket
                    (const void *) &reuseaddr, sizeof(int)) == -1) {
            ...
        }
        ......
        if (bind(s, ls[i].sockaddr, ls[i].socklen) == -1) {     //绑定监听地址
            ...
        }
        if (listen(s, ls[i].backlog) == -1) {            //开始监听
            ...
        }
        ...
    }
    ...
    ngx_msleep(500);                                    //每次尝试间隔为 500 ms
  }
  ...
  return NGX_OK;
}
// main->ngx_init_cycle->ngx_open_listening_sockets
```

第一层 for 循环是用来控制多次尝试 socket 连接的，默认的尝试次数是 5 次，每次间隔 500 ms。第二层 for 循环遍历 cycle->listening 中的 socket，调用 bind()和 listen()打开并绑定监听地址。有关套接字 socket 的编程我们不在这里详述。

到这里，我们完成了对 ngx_init_cycle()函数执行的主要工作的梳理。回到 main()函数，我们继续进行分析。ngx_init_cycle()函数执行完以后，主要的工作就是调用 ngx_init_signals()函数对信号进行设置：

```
// main
if (ngx_init_signals(cycle->log) != NGX_OK) {                    //初始化信号
        return 1;
  }
// main
```

由于 Nginx 服务器的启停服务等都是通过信号来控制的，所以我们将 Nginx 服务器程序信号设置部分的源码详细分析一下。

12.2.9　信号设置

Nginx 服务器程序的信号设置主要由 ngx_init_signals()函数完成。该函数的源码很简单：

```
// main->ngx_init_signals
ngx_int_t
```

```
ngx_init_signals(ngx_log_t *log)
{
    ngx_signal_t      *sig;
    struct sigaction   sa;
    for (sig = signals; sig->signo != 0; sig++) {
        ngx_memzero(&sa, sizeof(struct sigaction));
        sa.sa_handler = sig->handler;                          //信号的处理函数
        sigemptyset(&sa.sa_mask);
        if (sigaction(sig->signo, &sa, NULL) == -1) {          //设置信号处理方式
            ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                          "sigaction(%s) failed", sig->signame);
            return NGX_ERROR;
        }
    }
    return NGX_OK;
}
// main->ngx_init_signals
```

信号设置工作主要围绕存放信号信息的结构数组 signals 进行。数组中的每个元素是一个 ngx_signal_t 结构体，我们在上一章学习 ngx_signal_t 结构体时对源码定义的细节做了详细的分析，这里主要分析一下信号处理函数的源码。signals->handler 成员是指向信号处理函数的指针，指向 ngx_signal_handler()函数。

```
// ngx_signal_handler
void
ngx_signal_handler(int signo)
{
    …
    switch (ngx_process) {
    case NGX_PROCESS_MASTER:
    case NGX_PROCESS_SINGLE:                  //Master 模式和 Single 模式下对信号的处理是一样的
        switch (signo) {
        case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
            ngx_quit = 1;
            action = ", shutting down";
            break;
        case ngx_signal_value(NGX_TERMINATE_SIGNAL):
        case SIGINT:
            ngx_terminate = 1;
            action = ", exiting";
            break;
        …
        break;
    case NGX_PROCESS_WORKER:
    case NGX_PROCESS_HELPER:                            //工作进程和帮助对不同的信号进行处理
        switch (signo) {
        case ngx_signal_value(NGX_NOACCEPT_SIGNAL):
```

```
        if (!ngx_daemonized) {
            break;
        }
        ngx_debug_quit = 1;
    case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
        ngx_quit = 1;
        action = ", shutting down";
        break;
    ...
    break;
  }
......
  if (signo == SIGCHLD) {
      ngx_process_get_status();
  }
  ...
}
// ngx_signal_handler
```

函数的结构非常简单，它将 Nginx 服务器支持的信号分为四个大类。这些信号的处理结果是为一些全局变量赋值。通过这些全局变量，可以对 Nginx 服务器的状态进行调整。

有关 Nginx 服务器的信号设置，我们就简单介绍到这里。接下来 main()函数要进行的一个重要工作是调用 ngx_daemon 函数创建守护进程，该守护进程就是 Nginx 服务器的 master process。

12.2.10　启动 Master Process

```
// main->ngx_daemon
ngx_int_t
ngx_daemon(ngx_log_t *log)
{
    ...
    switch (fork()) {                              //生成一个子进程，准备启动守护进程
    case -1:
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "fork() failed");
        return NGX_ERROR;
    case 0:                                        //子进程要作为守护进程
        break;
    default:
        exit(0);
    }
    ngx_pid = ngx_getpid();
    if (setsid() == -1) {
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "setsid() failed");
        return NGX_ERROR;
    }
    umask(0);
    fd = open("/dev/null", O_RDWR);
```

```
    ...
    if (dup2(fd, STDIN_FILENO) == -1) {                  //设置守护进程的输入终端（为空）
        ...
    }
    if (dup2(fd, STDOUT_FILENO) == -1) {                 //设置守护进程的输出终端（为空）
        ...
    }
    if (fd > STDERR_FILENO) {                            //设置守护进程的错误终端
        if (close(fd) == -1) {
            ...
        }
    }
    return NGX_OK;
}
// main->ngx_daemon
```

Nginx 服务器创建守护进程和一般的 Linux 程序没什么太大的区别。我们这里要注意的是，全局变量 ngx_pid 被赋值为 Nginx 守护进程的 PID。后面的工作将全部在守护进程中完成。

到这里，Nginx 服务器的初始化过程我们就全部梳理完了。在这个过程中 ngx_init_cycle()函数是最主要的初始化函数，其对类型为 ngx_cycle_s 结构体的 cycle 结构进行了大量的初始化工作，这是我们学习的重点。下一节，我们将详细介绍 Nginx 服务器的启动过程。

12.2.11　Nginx 初始化过程总结

在本节的最后，我们通过 main()函数的函数调用树形表，对 Nginx 服务器的初始化过程进行总结，帮助大家进一步加深对这一过程的宏观把握。

表 12.1　main()函数的函数调用树形表

函数调用				功能
main()				入口函数
	ngx_strerror_init()			初始化错误输出列表
	ngx_get_options()			读取启动参数
	ngx_time_init()			初始化时间
		ngx_time_update()		更新缓存时间
	ngx_log_init()			初始化日志功能
	ngx_create_pool()			创建内存池
	ngx_save_argv()			保存启动参数到全局变量
	ngx_process_options()			保存启动参数到 init_cycle
	ngx_os_init()			获取或设置环境参数
	ngx_crc32_table_init()			初始化循环冗余校验表
	ngx_add_inherited_sockets()			继承之前程序的 socket
	ngx_init_cycle()			初始化 cycle 结构成员
		ngx_timezone_update()		更新时区信息

续表

函数调用				功能
		ngx_time_update()		更新时间缓存
		ngx_create_pool()		建立新的内存池
		module->create_conf()		建立 core 模块上下文结构
		ngx_conf_param()		准备解析配置文件
			ngx_conf_parse()	解析附加配置文件并使附加配置生效
		ngx_conf_parse()		解析主配置文件
			ngx_conf_handler()	使配置生效
		module -> *(*init_conf)		初始化 core 模块上下文结构
		ngx_create_pidfile()		创建 PID 文件
		init_cycle->listening 相关工作		处理监听 socket
			ngx_open_listening_sockets()	连接并打开 socket
	ngx_init_signals()			信号设置
	ngx_daemon()			启动 Master Process
	ngx_master_process_cycle()			启动 Work Processes

12.3　Nginx 的启动

在 main()函数的最后，调用 ngx_single_process_cycle()函数或者 ngx_master_process_cycle()函数启动 Nginx 服务器的工作进程：

```
// main
if (ngx_process == NGX_PROCESS_SINGLE) {
   ngx_single_process_cycle(cycle);
} else {
   ngx_master_process_cycle(cycle);
}
// main
```

使用 Nginx 服务器 Single 模式的情况比较少，我们重点学习 ngx_master_process_cycle()函数的源码。ngx_master_process_cycle()函数本身结构并不复杂，主要工作可以分为以下几步：

- 主进程设置信号阻塞。
- 设置进程标题。
- 启动 worker 进程。
- 启动缓存索引重建（Cache Loader）进程及管理（Cache Manager）进程。

- 主进程循环处理信号。

12.3.1　主进程设置信号阻塞

Nginx 服务器程序完成这一步工作的源码实现与其他 Linux 程序没有区别，我们简单浏览一下源码即可：

```
// main->ngx_master_process_cycle
sigemptyset(&set);
sigaddset(&set, SIGCHLD);
sigaddset(&set, SIGALRM);
sigaddset(&set, SIGIO);
sigaddset(&set, SIGINT);
sigaddset(&set, ngx_signal_value(NGX_RECONFIGURE_SIGNAL));
sigaddset(&set, ngx_signal_value(NGX_REOPEN_SIGNAL));
sigaddset(&set, ngx_signal_value(NGX_NOACCEPT_SIGNAL));
sigaddset(&set, ngx_signal_value(NGX_TERMINATE_SIGNAL));
sigaddset(&set, ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
sigaddset(&set, ngx_signal_value(NGX_CHANGEBIN_SIGNAL));
if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {            //改变目前的信号屏蔽字为阻塞状态
   ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno, "sigprocmask() failed");
}

sigemptyset(&set);
// main->ngx_master_process_cycle
```

源码很简单，首先调用 sigaddset()函数将关心的信号添加到信号集中，然后调用 sigprocmask()函数将信号集中信号的处理方式设置为 block 状态。涉及的信号有：SIGCHLD、SIGALRM、SIGIO、SIGINT、NGX_RECONFIGURE_SIGNAL、NGX_REOPEN_SIGNAL、NGX_NOACCEPT_SIGNAL、NGX_TERMINATE_SIGNAL、NGX_SHUTDOWN_SIGNAL 及 NGX_CHANGEBIN_SIGNAL 等。

注意

我们在上一节提到，在 Nginx 服务器初始化过程中进行信号设置时，一共涉及 12 个信号，其中有两个信号 SIGSYS 和 SIGPIPE，不包含在此列。

12.3.2　设置进程标题

Nginx 主进程的标题是什么呢？设置进程标题代码如下：

```
// main->ngx_master_process_cycle
size = sizeof(master_process);
for (i = 0; i < ngx_argc; i++) {
   size += ngx_strlen(ngx_argv[i]) + 1;
}
title = ngx_pnalloc(cycle->pool, size);
p = ngx_cpymem(title, master_process, sizeof(master_process) - 1);
for (i = 0; i < ngx_argc; i++) {
```

```
    *p++ = ' ';
    p = ngx_cpystrn(p, (u_char *) ngx_argv[i], size);
}
ngx_setproctitle(title);                                    //设置进程标题
// main->ngx_master_process_cycle
```

我们在/nginx/src/os/unix/ngx_process_cycle.c 中可以找到 master_process 变量的具体定义：

```
static u_char  master_process[] = "master process";
```

为什么要设置主进程的标题呢？这是为了将主进程和工作进程进行区分，这样我们在 Linux 命令行终端使用 ps x|grep nginx 查看 Nginx 服务器的相关进程时，就可以根据进程的标题区分哪个进程是 Nginx 服务器的主进程了。主要调用 ngx_setproctitle()函数对主进程的标题进行设置，我们来看一下该函数的源码片段：

```
// main->ngx_master_process_cycle->ngx_setproctitle
void
ngx_setproctitle(char *title)
{
    u_char     *p;
    ngx_os_argv[1] = NULL;
    p = ngx_cpystrn((u_char *) ngx_os_argv[0], (u_char *) "nginx: ",
                    ngx_os_argv_last - ngx_os_argv[0]);
    p = ngx_cpystrn(p, (u_char *) title, ngx_os_argv_last - (char *) p);
    if (ngx_os_argv_last - (char *) p) {
        ngx_memset(p, NGX_SETPROCTITLE_PAD, ngx_os_argv_last - (char *) p);
    }
    ngx_log_debug1(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
                   "setproctitle: \"%s\"", ngx_os_argv[0]);
}
// main->ngx_master_process_cycle->ngx_setproctitle
```

从这段源码可以分析出设置进程标题的原理。我们知道，通过 Linux 命令行终端启动 Nginx 程序时，所有的参数都将通过 argv 数组传给程序，其中 argv[0]中保存的是程序的名称，argv[1]中保存的是启动参数。通过修改 argv[0]的内容就可以设置进程的标题，同时还需要将 argv[1]的内容设置为 NULL。不过这里还有个问题，argv[0]的实际空间可能小于设置的标题大小。

Linux 平台运行程序时，一方面会使用 argv 数组存放参数，另一方面会使用 environ 数组存放系统的环境变量设置的标题。这两个数组在内存中是连续存放的。为了能够解决 argv[0]空间太小的问题，唯一的办法就是为 environ 数组分配一块新的内存区并拷贝出来，这样 argv[0]的空间就可以向后拓展了。Nginx 服务器程序借助 ngx_init_setproctitle()函数完成了 environ 数组拷贝的工作。

```
// main->ngx_os_init->ngx_init_setproctitle
ngx_int_t
ngx_init_setproctitle(ngx_log_t *log)
{
    u_char      *p;
    size_t       size;
    ngx_uint_t   i;
    size = 0;
```

```
    for (i = 0; environ[i]; i++) {              //前面都是执行准备工作，获取environ数组
        size += ngx_strlen(environ[i]) + 1;
    }
    p = ngx_alloc(size, log);
    if (p == NULL) {
        return NGX_ERROR;
    }
    ngx_os_argv_last = ngx_os_argv[0];
    for (i = 0; ngx_os_argv[i]; i++) {
        if (ngx_os_argv_last == ngx_os_argv[i]) {
            ngx_os_argv_last = ngx_os_argv[i] + ngx_strlen(ngx_os_argv[i]) + 1;
        }
    }
    for (i = 0; environ[i]; i++) {
        if (ngx_os_argv_last == environ[i]) {
            size = ngx_strlen(environ[i]) + 1;
            ngx_os_argv_last = environ[i] + size;
            ngx_cpystrn(p, (u_char *) environ[i], size);    //这里执行真正的拷贝工作
            environ[i] = (char *) p;
            p += size;
        }
    }
    ngx_os_argv_last--;
    return NGX_OK;
}
// main->ngx_os_init->ngx_init_setproctitle
```

该函数在 Nginx 服务器程序初始化时的 ngx_os_init()函数中被调用。

12.3.3 启动工作进程

接下来 Nginx 服务器程序的主要工作就是启动工作进程了。

```
// main->ngx_master_process_cycle
ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
ngx_start_worker_processes(cycle, ccf->worker_processes, NGX_PROCESS_RESPAWN);
// main->ngx_master_process_cycle
```

从源码来看，我们需要进一步看一下 ngx_start_worker_processes()函数的实现。

```
// main->ngx_master_process_cycle->ngx_start_worker_processes
static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
{
    ngx_int_t      i;
    ngx_channel_t  ch;
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start worker processes");
    ch.command = NGX_CMD_OPEN_CHANNEL;
    for (i = 0; i < n; i++) {
        ngx_spawn_process(cycle, ngx_worker_process_cycle,
```

```
                        (void *) (intptr_t) i, "worker process", type);//创建工作进程
        ch.pid = ngx_processes[ngx_process_slot].pid;
        ch.slot = ngx_process_slot;
        ch.fd = ngx_processes[ngx_process_slot].channel[0];
        ngx_pass_open_channel(cycle, &ch);                           //开启进程通信通道
    }
}
// main->ngx_master_process_cycle->ngx_start_worker_processes
```

在该段代码中，Nginx 服务器主要完成了两个工作。一是设置父子进程通信通道，二是根据配置文件的配置，使用 for 循环结构调用 ngx_spawn_process()函数完成所有工作进程的创建。我们会在相关章节详细介绍 Nginx 服务器的进程通信，这里主要分析 ngx_spawn_process()函数。

```
// main->ngx_master_process_cycle->ngx_start_worker_processes-> ngx_spawn_process
ngx_pid_t
ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
    char *name, ngx_int_t respawn)
{
    …
    if (respawn != NGX_PROCESS_DETACHED) {
        if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)  {
            …
        }
        …
        if (ioctl(ngx_processes[s].channel[0], FIOASYNC, &on) == -1) {
                                                          //设置信号驱动异步 I/O 标志
            …
        }
        if (fcntl(ngx_processes[s].channel[0], F_SETOWN, ngx_pid) == -1) {
                                                          //设置文件的进程 ID
            …
        }
        if (fcntl(ngx_processes[s].channel[0], F_SETFD, FD_CLOEXEC) == -1) {
                                                          //设置文件描述符标记
            …       }
        if (fcntl(ngx_processes[s].channel[1], F_SETFD, FD_CLOEXEC) == -1) {
                                                          //同上
            …
        }
        ngx_channel = ngx_processes[s].channel[1];
    } else {
        …
    }
// main->ngx_master_process_cycle->ngx_start_worker_processes-> ngx_spawn_process
```

函数的前一部分主要调用 socketpair()函数创建用于进程间通信的一对 socket 句柄，并且调用 ioctl()和 fcntl()对通信管道进行相应的设置。这些过程中涉及进程通信的基本知识，这不是本书讨论的重点，在这里就不过多介绍了。我们继续看该函数的后一部分。

```
// main->ngx_master_process_cycle->ngx_start_worker_processes-> ngx_spawn_process
    pid = fork();                                           //创建子进程（工作进程）
    switch (pid) {
    case -1:
        …
        return NGX_INVALID_PID;
    case 0:                                                 //执行工作进程
        ngx_pid = ngx_getpid();
        proc(cycle, data);                                  //工作进程的具体工作
        break;
    default:
        break;
    }
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start %s %P", name, pid);
    ngx_processes[s].pid = pid;
    ngx_processes[s].exited = 0;
    if (respawn >= 0) {
        return pid;
    }
// main->ngx_master_process_cycle->ngx_start_worker_processes->ngx_spawn_process
```

这段代码调用 fork()函数创建了一个 Nginx 服务器的工作进程，并且调用 proc 指向的函数完成工作进程的具体工作。

```
// main->ngx_master_process_cycle->ngx_start_worker_processes->ngx_spawn_process
    ngx_processes[s].proc = proc;
    ngx_processes[s].data = data;
    ngx_processes[s].name = name;
    ngx_processes[s].exiting = 0;
    switch (respawn) {
    case NGX_PROCESS_NORESPAWN:
        ngx_processes[s].respawn = 0;
        ngx_processes[s].just_spawn = 0;
        ngx_processes[s].detached = 0;
        break;
    case NGX_PROCESS_JUST_SPAWN:
        …
        break;
   case NGX_PROCESS_RESPAWN:
        …
        break;
    case NGX_PROCESS_JUST_RESPAWN:
        …
        break;
    case NGX_PROCESS_DETACHED:
        …
        break;
    }
```

```
    if (s == ngx_last_process) {
        ngx_last_process++;
    }
    return pid;
}
// main->ngx_master_process_cycle->ngx_start_worker_processes->ngx_spawn_process
```

ngx_spawn_process()函数的最后一部分主要根据传入的respawn标志对Nginx服务器进程表中当前的进程进行相关设置，包括程序的执行函数、运行状态等。当然在这里 respawn 的值为 NGX_PROCESS_RESPAWN。

12.3.4　启动缓存索引重建及管理进程

我们在前面的相关章节介绍过 Nginx 服务器的缓存索引重建及管理进程。Nginx 服务器程序启动工作进程后紧接着就调用下面的函数启动缓存索引重建进程。

```
// main->ngx_master_process_cycle
ngx_start_cache_manager_processes(cycle, 0);
// main->ngx_master_process_cycle
```

该进程在整个 Nginx 服务器运行过程中只存在很短的时间，主要用来遍历磁盘上的缓存数据，在内存中建立数据索引，提高 Nginx 服务器检索缓存的效率。

```
// main->ngx_master_process_cycle->ngx_start_cache_manager_processes
static void
ngx_start_cache_manager_processes(ngx_cycle_t *cycle, ngx_uint_t respawn)
{
    ...
    ngx_spawn_process(cycle, ngx_cache_manager_process_cycle,  //创建缓存索引管理进程
                      &ngx_cache_manager_ctx, "cache manager process",
                      respawn ? NGX_PROCESS_JUST_RESPAWN : NGX_PROCESS_RESPAWN);
    ch.command = NGX_CMD_OPEN_CHANNEL;
    ch.pid = ngx_processes[ngx_process_slot].pid;
    ch.slot = ngx_process_slot;
    ch.fd = ngx_processes[ngx_process_slot].channel[0];
    ngx_pass_open_channel(cycle, &ch);                        //创建进程间通信管道
// main->ngx_master_process_cycle->ngx_start_cache_manager_processes
```

ngx_start_cache_manager_processes()函数的前半部分调用 ngx_spawn_process()函数创建了缓存索引管理进程 Cache Manager Process，传入的 respawn 值为 0，因此仍然是 NGX_PROCESS_RESPAWN。ngx_pass_open_channel()函数用于打开通道向工作进程发送该进程创建的消息。

```
// main->ngx_master_process_cycle->ngx_start_cache_manager_processes
    if (loader == 0) {
        return;
    }
    ngx_spawn_process(cycle, ngx_cache_manager_process_cycle, //创建缓存索引管理进程
                      &ngx_cache_loader_ctx, "cache loader process",
                      respawn ? NGX_PROCESS_JUST_SPAWN : NGX_PROCESS_NORESPAWN);
    ch.command = NGX_CMD_OPEN_CHANNEL;
```

```
    ch.pid = ngx_processes[ngx_process_slot].pid;
    ch.slot = ngx_process_slot;
    ch.fd = ngx_processes[ngx_process_slot].channel[0];
    ngx_pass_open_channel(cycle, &ch);                          //创建进程间通信管道
}
// main->ngx_master_process_cycle-> ngx_start_cache_manager_processes
```

ngx_start_cache_manager_processes()函数的后半部分和前半部分相同，调用 ngx_spawn_process()函数创建了缓存索引管理进程 Cache Loader Process，传入的 respawn 值为 0，因此为 NGX_PROCESS_NORESPAWN。该进程在完成预定工作后自行退出。

12.3.5 循环处理信号

信号处理是 Nginx 服务器主进程在整个服务器运行期间的主要工作，程序通过一个 for 循环结构实现对信号的循环处理。

```
// main->ngx_master_process_cycle
delay = 0;
sigio = 0;
live = 1;
for ( ;; ) {
    if (delay) {                                            //等待工作进程退出的时间
        if (ngx_sigalrm) {
            sigio = 0;
            delay *= 2;
            ngx_sigalrm = 0;
        }
       ...
        itv.it_interval.tv_sec = 0;                         //初始化一个定时器
        itv.it_interval.tv_usec = 0;
        itv.it_value.tv_sec = delay / 1000;
        itv.it_value.tv_usec = (delay % 1000 ) * 1000;
        if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {     //设置定时器
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "setitimer() failed");
        }
    }
    ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "sigsuspend");
// main->ngx_master_process_cycle
```

这里的 delay 变量中保存的是等待工作进程退出的时间。主进程接收到结束程序的信号后，首先需要将退出信号发送给各个工作进程，等待它们退出后再退出主进程。为此该段代码中还设置了定时器。

```
// main->ngx_master_process_cycle
    sigsuspend(&set);                                       //挂起进程，等待接收到信号
// main->ngx_master_process_cycle
```

这里调用 sigsuspend()函数等待信号的到来。在不向主进程发送信号的情况下，该进程将一直挂起

在这里等待。

```
// main->ngx_master_process_cycle
   ngx_time_update();                                          //更新缓冲时间
   …
   if (ngx_reap) {                                             //是否需要重启工作进程
      ngx_reap = 0;
      ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "reap children");
      live = ngx_reap_children(cycle);                         //重启工作进程
   }
// main->ngx_master_process_cycle
```

这段代码处理工作进程退出的情况，如果有工作进程异常退出，则调用 ngx_reap_children()重启该工作进程。

```
// main->ngx_master_process_cycle
   if (!live && (ngx_terminate || ngx_quit)) {
      ngx_master_process_exit(cycle);                          //主进程退出
   }
// main->ngx_master_process_cycle
```

如果主进程接收到的是 NGX_CMD_TERMINATE 信号、SIGTERM 信号、SIGINT 信号（ngx_terminate= 1）或者收到的是 NGX_CMD_QUIT 信号、SIGQUIT 信号（ngx_quit = 1），并且工作进程退出，则主进程调用 ngx_master_process_exit()函数退出。

```
// main->ngx_master_process_cycle
   if (ngx_terminate) {                                        //处理 SIGINT 信号
      if (delay == 0) {
         delay = 50;
      }
      if (sigio) {
         sigio--;
         continue;
      }
      sigio = ccf->worker_processes + 2                        //包括两个管理内存索引的进程
      if (delay > 1000) {
         ngx_signal_worker_processes(cycle, SIGKILL);          //超时 1s，直接终止工作进程
      } else {
         ngx_signal_worker_processes(cycle,                    //工作进程正常退出
                           ngx_signal_value(NGX_TERMINATE_SIGNAL));
      }
      continue;
   }
// main->ngx_master_process_cycle
```

这一段代码用来完成主进程接收到 NGX_CMD_TERMINATE 信号、SIGTERM 信号或者 SIGINT 信号时的工作。首先设置等待各个工作进程的退出时间，然后调用 ngx_signal_worker_processes()函数向各个工作进程发送 NGX_TERMINATE_SIGNAL 信号。这里还处理了延时时间超过 1000 ms 的情况，就是给所有没有结束的工作进程发送 SIGKILL 信号，强制杀死这些进程。

```
// main->ngx_master_process_cycle
   if (ngx_quit) {                                            //处理 SIGQUIT 信号
       ngx_signal_worker_processes(cycle,
                              ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
       ls = cycle->listening.elts;
       for (n = 0; n < cycle->listening.nelts; n++) {
           if (ngx_close_socket(ls[n].fd) == -1) {            //关闭所有的 socket
               ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
                             ngx_close_socket_n " %V failed",
                             &ls[n].addr_text);
           }
       }
       cycle->listening.nelts = 0;
       continue;
   }
// main->ngx_master_process_cycle
```

这一段代码用来完成主进程接收到 NGX_CMD_QUIT 信号或 SIGQUIT 信号时的工作。程序将调用 ngx_signal_worker_processes()函数向各个工作进程发送 SIGKILL 信号，并遍历 cycle->listening 中的所有监听 socket 关闭它们。

```
// main->ngx_master_process_cycle
   if (ngx_reconfigure) {                                     //处理 SIGHUP 信号
       ngx_reconfigure = 0;
       if (ngx_new_binary) {
           ngx_start_worker_processes(cycle, ccf->worker_processes,
                                      NGX_PROCESS_RESPAWN);   //平滑升级
           ngx_start_cache_manager_processes(cycle, 0);
           ngx_noaccepting = 0;
           continue;
       }
       ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");
       cycle = ngx_init_cycle(cycle);                         //建立新的 cycle 结构
       if (cycle == NULL) {
           cycle = (ngx_cycle_t *) ngx_cycle;
           continue;
       }
       ngx_cycle = cycle;
       ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,    //读取 Nginx 配置
                                              ngx_core_module);
       ngx_start_worker_processes(cycle, ccf->worker_processes,   //创建工作进程
                                  NGX_PROCESS_JUST_RESPAWN);
       ngx_start_cache_manager_processes(cycle, 1);               //创建缓存管理进程
       // allow new processes to start
       ngx_msleep(100);
       live = 1;
       ngx_signal_worker_processes(cycle,
ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
```

```
                                                        //关闭旧的 Nginx 进程
    }
// main->ngx_master_process_cycle
```

这一段代码用来完成主进程接收到 SIGHUP 信号时的工作。当 ngx_new_binary 变量等于 1 时，说明主进程本身需要升级，但是不需要重新初始化 Nginx 配置，因此可以直接调用 ngx_start_worker_processes()函数重启工作进程和缓存索引管理进程。但如果 ngx_new_binary 不为 1，则说明是 Nginx 服务器配置改变，需要调用 ngx_init_cycle()函数初始化 Nginx 配置，并按照新的配置启动工作进程和缓存索引管理进程，向之前的所有进程发送 NGX_SHUTDOWN_SIGNAL 信号。这样就实现了 Nginx 服务器的平滑升级。

```
// main->ngx_master_process_cycle
    if (ngx_restart) {                                              //重启工作进程
        ngx_restart = 0;
        ngx_start_worker_processes(cycle, ccf->worker_processes,    //启动工作进程
                                   NGX_PROCESS_RESPAWN);
        ngx_start_cache_manager_processes(cycle, 0);          //启动缓存管理程序刷新缓存
        live = 1;
    }
// main->ngx_master_process_cycle
```

这一段源码是处理 ngx_restart 变量为 1 时的情况。只有一种情况可以将 ngx_restart 变量赋值为 1，就是在调用 ngx_reap_children()函数重启工作进程的时候。当主进程接收到 NGX_NOACCEPT_SIGNAL 信号（不再接收请求，退出工作进程）时，会设置全局变量 ngx_noaccept 为 1，然后在 Nginx 初始化信号设置时会将全局变量 ngx_noaccepting 设置为 1，于是在 ngx_reap_children()函数中就会将 ngx_restart 变量设置为 1，进而执行该段源码，重启工作进程和缓存索引管理进程。

```
// main->ngx_master_process_cycle
    if (ngx_reopen) {                                     //处理 SIGUSR1 信号
        ngx_reopen = 0;
        ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
        ngx_reopen_files(cycle, ccf->user);
        ngx_signal_worker_processes(cycle,
                                    ngx_signal_value(NGX_REOPEN_SIGNAL));
    }
// main->ngx_master_process_cycle
```

这一段源码是处理 Nginx 主进程收到 SIGUSR1 信号时的情况，重新打开日志文件。

```
// main->ngx_master_process_cycle
    if (ngx_change_binary) {                          //处理 SIGUSR2 信号，热代码切换
        ngx_change_binary = 0;
        ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "changing binary");
        ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
    }
// main->ngx_master_process_cycle
```

这一段源码是处理 Nginx 主进程收到 SIGUSR2 信号时的情况，进行热代码切换。

```
// main->ngx_master_process_cycle
    if (ngx_noaccept) {                               //处理NGX_NOACCEPT_SIGNAL 信号
```

```
        ngx_noaccept = 0;
        ngx_noaccepting = 1;
        ngx_signal_worker_processes(cycle,          //退出工作进程，不接收网络请求
                    ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
    }
}
// main->ngx_master_process_cycle
```

在前面讨论处理 ngx_restart 变量为 1 时的情况时我们介绍过 ngx_noaccept 变量为 1 的情况。当主进程接收到 NGX_NOACCEPT_SIGNAL 信号时，ngx_noaccept 变量赋值为 1，Nginx 服务器不再接收请求，并将工作进程全部退出，该段代码就实现了该功能。

到这里，我们就将 Nginx 服务器的启动过程全部分析完了，其中还包括了对主进程工作循环处理信号源码的分析。

12.3.6 Nginx 启动过程总结

最后我们使用一张流程示意图对 Nginx 服务器启动过程进行总结，帮助大家进一步加深对这一过程的宏观把握。流程示意图如图 12.3 所示。

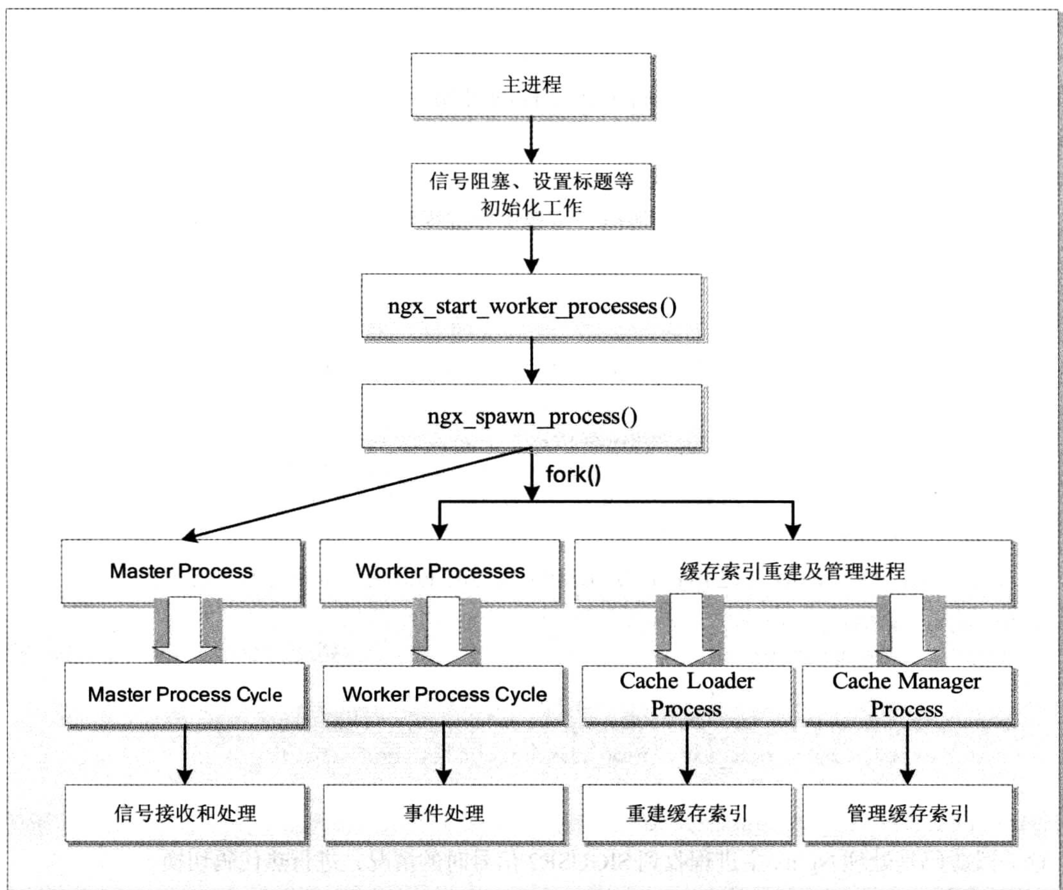

图 12.3　Master 模型下多进程启动过程

该图实际上是基于图 12.2，结合我们前面学习过的 Nginx 启动知识，将其进一步细化，这个过程在以上基础上非常容易理解，唯一需要注意的是 Cache Loader Process 子进程在完成重建缓存索引的任务后直接退出，而其他各个进程则在整个 Nginx 服务器的运行过程中一直存在。

12.4　本章小结

在本章中，我们以 Nginx 服务器程序的 main()函数作为切入点，通过对源码的分析，将 Nginx 服务器的初始化过程和启动过程进行了详细的分析。在 Nginx 服务器的初始化过程中，init_cycle 结构与 cycle 结构的关系和区别、cycle 结构的初始化是学习的重点；在 Nginx 服务器的启动过程中，对主进程的主要工作的理解是学习的重点。相信通过本章的学习，大家对 Nginx 服务器的这两个过程中进行了哪些工作有了比较深入的了解。

在学习过程中，我们频繁地和本书之前的相关章节进行关联，在学习源码的基础上强化了对前面相关知识的理解，尤其是对 Nginx 配置文件解析、主进程和工作进程的关联、网络套接字管理、Nginx 信号处理、缓存索引重建和管理进程等内容有了清晰的认识。笔者建议大家在学习本章的源码时，注意对前面相关章节内容的复习，加深对 Nginx 服务器工作原理的认识。

在本章学习 main()函数的源码中，我们涉及到 Nginx 服务器的时间管理、内存管理和进程通信方面的知识。我们将在后面的相关章节中对其详细介绍，对其中用到的内存、进程、通信方面的基础知识笔者将尽量详细描述，但鉴于篇幅所限，没有涉及到的地方也建议大家自行学习。

第 13 章

Nginx 的时间管理

从 Nginx 服务器的时间管理实现机制，我们能够体会到 Nginx 程序在设计过程中为了进一步降低系统开销，在处理系统调用时是相当谨慎的。这一章我们结合相关的程序源码对 Nginx 服务器的时间管理机制进行详细说明。

在本章中，我们将要学习到以下内容：

- Linux 平台获取系统时间的方法
- Nginx 时间缓存机制的基本工作原理
- 缓存时间的精度设置

13.1 获取系统时间的一般方法

在 Linux 平台上获取系统时间的方法有很多，比如使用 time()函数、gettimeofday()函数和 localtime()函数等，大部分函数本质上都是通过系统调用来获取时间的。但是使用系统调用时，有一个问题需要特别关注，就是系统调用开销的问题。

13.1.1 系统调用的开销

Nginx 服务器程序在获取系统时间时，本质上使用的是 gettimeofday()函数，该函数是 C 语言库提供的函数，从严格意义上讲，该函数不是系统调用，但它是对系统调用 sys_gettimeofday()的封装，因此一般也就认为它是一个系统调用了。

大家都知道，Linux 平台上程序函数调用可以分为库调用和系统调用两大类。我们在这里不过多地解释这两个机制的不同，主要是明确系统调用与库调用相比，时间成本是相当巨大的。程序执行一

次系统调用将至少经过以下步骤：

- 应用程序调用库函数 API。
- API 将系统调用号存入寄存器 EAX，然后通过中断调用使系统进入内核态（陷入内核空间）。
- 内核中的中断处理函数根据寄存器 EAX 中的系统调用号，调用对应的内核函数（系统调用）。
- 系统调用完成相应功能，将返回值存入寄存器 EAX，返回到中断处理函数。
- 中断处理函数返回到 API 中。
- API 将寄存器 EAX 返回给应用程序。

我们可以看到，在执行系统调用的过程中要涉及内核空间和用户空间转换、系统中断处理等过程，这些都增加了大量的系统开销。在实际进行程序设计的过程中，如果不注意减少系统调用的使用，应用程序将无法得到很好的执行效率。

Nginx 服务器在设计之初，主要关注的一方面就是效率问题，并且从实际的使用情况来看，Nginx 服务器的系统开销非常低，而运行效率也是十分令人满意的，由此可以推断出，它在对系统调用的使用问题上，处理方案是谨慎和有效的。为了获取精确的系统时间，Nginx 服务器使用了 gettimeofday() 系统调用。那么，Nginx 服务器是怎么做到既使用系统调用获取精确时间，又保证程序的运行效率和系统开销的呢？我们将在接下来的内容中为大家详细分析。Nginx 服务器获取系统时间的机制是值得我们在实际程序设计中模仿和借鉴的。

13.1.2 gettimeofday()

在 Linux 平台上使用 C 语言进行程序设计时，为了精确获取系统时间，我们经常会使用 gettimeofday()这个函数。该函数的原型为：

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz )
```

调用该函数后，当前的时间将用 timeval 结构体返回，时间可以精确到微妙；当地时区的信息则放到 timezone 结构体中。timeval 结构体的定义为：

```
struct timeval {
      long  tv_sec;
      long  tv_usec;
};
```

- tv_sec，从当天零时开始经过的秒数。
- tv_usec，从当天零时开始经过的毫秒数。

timezone 结构体的定义为：

```
struct  timezone{
     int  tz_minuteswest;
     int  tz_dsttime;
}
```

- tv_minuteswest，格林威治时间往西方的时差。
- tv_dsttime，时间的修正方式，也就是日光节约时间（夏时制）的使用情况。

目前，在 Linux 平台下，timezone 结构体已经废除不再使用。因为有许多地区和国家的日光节约时间使用与否，与相关的政策相关，没有简单的方法来实现这项设计。因此在调用 gettimeofday()函数时可以将第二个参数设置为 NULL。

gettimeofday()函数在执行过程中，一般情况下实际上是调用了另一个函数 sys_gettimeofday()，该函数才是名副其实的系统调用，通过该调用就可以获取保存在系统内核中的时间信息。在实际程序设计时是不提倡频繁使用 gettimeofday()函数的。

这里有一点需要说明的是，Linux 内核在不同的体系结构下，执行系统调用的方式是不尽相同的。在一般的 i386 体系结构中，执行系统调用的过程就是我们上面描述的步骤。但对于 x86_64 体系结构来说，除了普通的系统调用外，还提供了 sysenter 和 vsyscall 方式来获取内核态的数据，而目前我们使用的操作系统大都是 x86_64 体系的。vsyscall 方式在内存中创建了一个内核态的共享页面，它的数据由内核来维护，但这块区域用户态也有权限访问，通过这样的机制，不经过系统中断和陷入内核也能获取一些内核信息。幸运的是，x86_64 体系上使用 vsyscall 方式实现了 gettimeofday()的功能，这样系统开销比普通的系统调用要小得多。

13.2 Nginx 时间管理的工作原理

Nginx 服务器重视程序的高效运行，因此对于耗费时间的系统调用当然是敬而远之，但是在必要的情况下也不能完全抛弃不用。于是，如何降低在获取时间时对系统调用的使用频率，就成为了解决问题的突破口。最终，Nginx 程序采取缓存时间的方法来减少对 gettimeofday()的调用，并且每个工作进程会自行维护时间缓存。Nginx 的时间缓存一般会赋值给以下四种变量，更新缓存时是同步更新的。

- ngx_cached_http_time
- ngx_cached_err_log_time
- ngx_cached_http_log_time
- ngx_cached_http_log_iso8601

既然是对时间做了缓存，就必须有相关的程序来完成更新缓存的工作。Nginx 程序中有两个相关函数在不同的情况下来完成这个工作。

13.2.1 时间缓存的更新

Nginx 服务器更新时间缓存的两个函数是 ngx_time_update()和 ngx_time_sigsafe_update()，具体的实现源码都在/nginx/src/core/ngx_time.c 中可以找到。我们分别来分析一下它们的源码。

1.ngx_time_update()

该函数是 Nginx 服务器时间管理的核心函数，我们通过分析该函数的源码来深入理解 Nginx 时间管理的工作原理。

```
// ngx_time_update
void
ngx_time_update(void)
{
```

```
    u_char          *p0, *p1, *p2, *p3;
    ngx_tm_t         tm, gmt;
    time_t           sec;
    ngx_uint_t       msec;
    ngx_time_t      *tp;
    struct timeval   tv;
    if (!ngx_trylock(&ngx_time_lock)) {              //使用 ngx_time_lock 进行写加锁
        return;
    }
// ngx_time_update
```

更新时间缓存的过程实际上是一个写缓存的过程，Nginx 服务器为了解决信号处理过程中更新时间缓存产生的数据一致性问题，需要使用原子变量 ngx_time_lock 进行写加锁。

```
// ngx_time_update
    ngx_gettimeofday(&tv);
// ngx_time_update
```

ngx_gettimeofday()为宏定义：

```
#define ngx_gettimeofday(tp)  (void) gettimeofday(tp, NULL);
```

实际上就是使用系统调用 gettimeofday()获取的时间。

```
// ngx_time_update
    sec = tv.tv_sec;
    msec = tv.tv_usec / 1000;
    ngx_current_msec = (ngx_msec_t) sec * 1000 + msec;         //统一时间单位为毫秒
// ngx_time_update
```

这里是将 gettimeofday()获取的时间统一转换成毫秒。

```
// ngx_time_update
    tp = &cached_time[slot];
    if (tp->sec == sec) {
        tp->msec = msec;
        ngx_unlock(&ngx_time_lock);
        return;
    }
// ngx_time_update
```

这里有一个数组 cached_time[slot]需要重点说明一下。先看它的定义：

```
static ngx_time_t   cached_time[NGX_TIME_SLOTS];
#define NGX_TIME_SLOTS   64
```

Nginx 服务器程序在设计时，考虑到读时间缓存的操作要比写时间缓存的操作频繁得多，同样出于性能的考虑就没有对读操作加锁，这样一来，如何才能避免读操作产生的冲突呢？Nginx 程序通过维护多个时间 slot 的方法来减少读操作冲突。我们通过一个例子来说明其中的原理。

假设现在有个工作进程正在执行读缓存操作，当它读取了一半的时间数据时，接收到一个信号，从而中断当前的读缓存操作去执行信号处理函数，而信号处理函数中更新了时间缓存。当信号处理函数执行完毕后，进程回到之前中断的读缓存操作继续执行，这样读取出来的时间势必是无效的。现在，Nginx 程序维护了一个时间 slot 数组，一共有 64 个元素，每次更新时间缓存时都更新数组中的一个新

的元素，而中断之前操作的那个元素并不改变，这样，进程执行完信号处理函数后回到之前中断的读缓存操作时，读取的仍然是中断之前的那个时间 slot，读取出来的时间无效的概率就很小了。

通过这样的方式，Nginx 服务器程序不但能够解决读缓存的冲突问题，同时也避免了给读操作频繁加锁带来的性能损耗问题。

```
// ngx_time_update
    if (slot == NGX_TIME_SLOTS - 1) {
        slot = 0;
    } else {
        slot++;
    }
// ngx_time_update
```

通过这段代码可以看到，Nginx 服务器程序维护的时间 slot 数组是循环使用的。这里确定了当前要更新的时间 slot。

```
// ngx_time_update
    tp = &cached_time[slot];
    tp->sec = sec;
    tp->msec = msec;
    ngx_gmtime(sec, &gmt);                                //转换成可读的时间格式
// ngx_time_update
```

ngx_gmtime()函数的作用是将刚才获取到的时间（毫秒）转换成可读的时间格式，包括年、月、日、星期、时、分、秒等，存放在 gmt 结构中，其类型为 ngx_tm_t 结构体，实际上就是 tm 结构体：

```
typedef struct tm    ngx_tm_t;
```

有关 ngx_gmtime()函数的具体实现比较简单，在这里就不展示了。

```
// ngx_time_update
    p0 = &cached_http_time[slot][0];                  //用于记录 http 请求的时间
    (void) ngx_sprintf(p0, "%s, %02d %s %4d %02d:%02d:%02d GMT",
                   week[gmt.ngx_tm_wday], gmt.ngx_tm_mday,
                   months[gmt.ngx_tm_mon - 1], gmt.ngx_tm_year,
                   gmt.ngx_tm_hour, gmt.ngx_tm_min, gmt.ngx_tm_sec);
    p1 = &cached_err_log_time[slot][0];               //用于记录错误日志的时间
    (void) ngx_sprintf(p1, "%4d/%02d/%02d %02d:%02d:%02d",
                   tm.ngx_tm_year, tm.ngx_tm_mon,
                   tm.ngx_tm_mday, tm.ngx_tm_hour,
                   tm.ngx_tm_min, tm.ngx_tm_sec);
    p2 = &cached_http_log_time[slot][0];              //用于记录 http 请求日志的时间
    (void) ngx_sprintf(p2, "%02d/%s/%d:%02d:%02d:%02d %c%02d%02d",
                   tm.ngx_tm_mday, months[tm.ngx_tm_mon - 1],
                   tm.ngx_tm_year, tm.ngx_tm_hour,
                   tm.ngx_tm_min, tm.ngx_tm_sec,
                   tp->gmtoff < 0 ? '-' : '+',
                   ngx_abs(tp->gmtoff / 60), ngx_abs(tp->gmtoff % 60));
    p3 = &cached_http_log_iso8601[slot][0];           //符合 iso8601 标准格式的时间
    (void) ngx_sprintf(p3, "%4d-%02d-%02dT%02d:%02d:%02d%c%02d:%02d",
```

```
                tm.ngx_tm_year, tm.ngx_tm_mon,
                tm.ngx_tm_mday, tm.ngx_tm_hour,
                tm.ngx_tm_min, tm.ngx_tm_sec,
                tp->gmtoff < 0 ? '-' : '+',
                ngx_abs(tp->gmtoff / 60), ngx_abs(tp->gmtoff % 60));
    ngx_memory_barrier();
    ngx_cached_time = tp;
    ngx_cached_http_time.data = p0;
    ngx_cached_err_log_time.data = p1;
    ngx_cached_http_log_time.data = p2;
    ngx_cached_http_log_iso8601.data = p3;
    ngx_unlock(&ngx_time_lock);
}
// ngx_time_update
```

最后这段代码结构比较清晰，但比较难理解。我们先来看一下其中的 ngx_memory_barrier()函数。该函数实际上又是一个宏，具体的定义和 Nginx 服务器当前的运行环境以及编译器有关。比如，在 x86 下使用 gcc 编译器的环境中，它的定义是这样的：

```
#define ngx_memory_barrier()    __asm__ volatile ("" ::: "memory")
```

虽然我们可能对这个宏定义不理解，但是当看到 volatile 关键字时，大多数人可以猜到这条宏定义的作用是什么。它是用来告诉编译器，不需要对其后面的语句进行优化。但这和这一段代码有什么关系呢？将这条语句注释掉后重新浏览这部分源码，就会发现一些问题了。

这条语句之前是对 tp、p0、p1、p2、p3 的赋值，而在该语句之后，是将 tp、p0、p1、p2、p3 的值更新到不同的时间缓存。如果没有中间这条语句，编译器很有可能将这两部分语句进行合并优化。但事实上，只有先执行完 p3 被赋值的语句后，5 个时间缓存才能保证数据的一致性。如果进行合并优化，在 p3 赋值之前可能 5 个时间缓存中有的时间缓存已经被更新，如果期间正好有进程读取了该时间缓存中的时间，就会导致读取时间无效的不一致的情况发生。

以上就是对 ngx_time_update()函数源码的分析，其中涉及了 Ngixn 时间管理的主要工作原理，重点是理解 Nginx 服务器避免读写时间缓存区产生冲突的问题。

2. ngx_time_sigsafe_update()

该函数实际上是 Nginx 服务器程序更新缓存机制中的一个特例，主要用来更新 ngx_cached_err_log_time 中的时间。

```
// ngx_time_sigsafe_update
void
ngx_time_sigsafe_update(void)
{
    u_char          *p;
    ngx_tm_t         tm;
    time_t           sec;
    ngx_time_t      *tp;
    struct timeval   tv;
    if (!ngx_trylock(&ngx_time_lock)) {                    //加写锁
        return;
```

```
    }
    ngx_gettimeofday(&tv);                              //获取时间
    sec = tv.tv_sec;
    tp = &cached_time[slot];
    if (tp->sec == sec) {                               //如果和缓存的时候一样，不需要更新
        ngx_unlock(&ngx_time_lock);
        return;
    }
    if (slot == NGX_TIME_SLOTS - 1) {
        slot = 0;
    } else {
        slot++;
    }
    tp = &cached_time[slot];
    tp->sec = 0;
    ngx_gmtime(sec + cached_gmtoff * 60, &tm);     //转换成可读时间格式
    p = &cached_err_log_time[slot][0];
    (void) ngx_sprintf(p, "%4d/%02d/%02d %02d:%02d:%02d",
                    tm.ngx_tm_year, tm.ngx_tm_mon,
                    tm.ngx_tm_mday, tm.ngx_tm_hour,
                    tm.ngx_tm_min, tm.ngx_tm_sec);
    ngx_memory_barrier();
    ngx_cached_err_log_time.data = p;          //更新 ngx_cached_err_log_time 中的时间
    ngx_unlock(&ngx_time_lock);
}
// ngx_time_sigsafe_update
```

以上是该函数的实现源码。在我们分析了 ngx_time_update()函数的源码之后，相信大家可以轻松理解该函数的执行过程，笔者不再赘述。

13.2.2 更新时间缓存的时机

我们在前面已经提到，Nginx 服务器程序采取了时间缓存的办法减少对系统调用的使用频率，这体现在源码上就是对时间缓存更新函数的调用频率很低。选择恰到好处的时机更新时间缓存才能达到这样的目标。

1. ngx_time_sigsafe_update()的调用

我们先来分析 ngx_time_sigsafe_update()函数的调用时机。在整个 Nginx 服务器的程序中，只有一处调用了该函数，就是在 ngx_signal_handler()函数中。

```
// ngx_signal_handler
void
ngx_signal_handler(int signo)
{
    char            *action;
    ngx_int_t        ignore;
    ngx_err_t        err;
    ngx_signal_t    *sig;
```

```
    ignore = 0;
    err = ngx_errno;
    for (sig = signals; sig->signo != 0; sig++) {
        if (sig->signo == signo) {
            break;
        }
    }
    ngx_time_sigsafe_update();                          //更新时间
    ……                                                  //信号处理
// ngx_signal_handler
```

我们在前面的相关章节中介绍过该函数，它是信号处理函数。当 Nginx 服务器的进程收到某信号执行信号处理函数时，会调用 ngx_time_sigsafe_update()函数更新 ngx_cached_err_log_time 时间。可以看到，调用该函数更新缓存的频率是很小的。

2. ngx_time_update()的调用

该函数的调用时机主要有三个：一是在 Nginx 服务器主进程捕捉、处理完一个信号返回的时候；二是在缓存索引管理进程中调用该函数，用于标记缓存数据的时间属性；三是在 Nginx 服务器工作进程中在进行事件处理时调用了该函数。前两种情况下对该函数的调用频率不高，我们主要来看第三种情况。

事件处理函数我们在后面的相关章节还会详细介绍，这里先向大家说明一些必要的细节。ngx_process_events()实际上被定义为宏：

```
#define ngx_process_events   ngx_event_actions.process_events
```

其中的 ngx_event_actions 为 Nginx 程序的 I/O 模型接口函数结构体，其封装了 epoll、kqueue、select、poll 等事件驱动模型接口。我们以 epoll 为例，ngx_event_actions.process_events 指向的接口为/nginx/src/event/modules/ngx_epoll_module.c 文件中的 ngx_epoll_process_events()函数。该文件完成了 epoll 事件驱动模型的具体实现。在第 3 章中详细介绍了 epoll 事件驱动模型的工作原理，在此就不详细分析相关源码了，这里我们只关心与 Nginx 时间管理相关的部分源码片段：

```
// ngx_epoll_process_events
static ngx_int_t
ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
{
    // 变量声明
    …
    events = epoll_wait(ep, event_list, (int) nevents, timer);     //等待事件的产生
    err = (events == -1) ? ngx_errno : 0;
    if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
        ngx_time_update();                                          //更新时间
    }
    …
// ngx_epoll_process_events
```

在该函数片段中，epoll_wait()函数用于等待事件的产生，执行 epoll_wait()函数返回后会调用 ngx_time_update()函数更新时间缓存。从两个函数的执行顺序可以看出，当 epoll 机制通知有事件到达或者

epoll 机制超时退出时，Nginx 服务器程序就会更新一次缓存时间，然后调用各个事件对应的处理函数处理事件。

也许有细心的读者会发现，按照这样的方法获取时间虽然减少了系统开销，但是由于该时间是缓存好的时间，事件处理函数执行完成后的时间已经不是刚才更新过的时间了。确实是这样的，这里存在一个时间精度的问题，程序获取到的时间比当前时间要早，但这样的误差并不会给 Nginx 服务器程序的运行带来错误。事件处理函数自身是无阻塞的，执行时间控制在毫秒级范围内。这样，在任一处理函数中，取到的时间虽然都是刚才缓存的时间，但误差很小，不会对 Nginx 服务器的整体运行造成影响。

13.3 缓存时间的精度

上面提到了在更新缓存时间工作中涉及的时间精度问题。实际上，Nginx 服务器对此做了比较细致的考虑，在配置中支持设置用户自行控制缓存时间精度，使用的指令是 timer_resolution。

13.3.1 设置缓存时间的精度

指令 timer_resolution 用于设置执行两次缓存时间更新工作之间的间隔时间。通过上面学习 Nginx 时间管理的工作原理我们知道，该指令的参数设置越大，则对系统调用 gettimeofday()的使用频率就越低，但缓存时间的精度也就越低。该指令的语法结构为：

```
timer_resolution interval;
```

其中的 interval 参数就是更新时间间隔，默认值为 100 ms。该指令只能在 Nginx 配置文件的全局块中进行设置。

13.3.2 缓存时间精度的控制原理

在 Nginx 服务器程序中，指令 timer_resolution 的参数被保存在全局变量 ngx_timer_resolution 中。解析 Nginx 配置时在 event 模块初始化过程中对其赋值：

```
// ngx_event_module_init
static ngx_int_t
ngx_event_module_init(ngx_cycle_t *cycle)
{
    ngx_core_conf_t    *ccf;
    …
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module); //获
取配置上下文
    ngx_timer_resolution = ccf->timer_resolution;     //提取 timer_resolution 的参数
    …
}
// ngx_event_module_init
```

在初始化事件处理机制的过程中，该全局变量将被用来设置缓存时间更新的时间间隔：

```
// ngx_event_process_init
static ngx_int_t
```

```
ngx_event_process_init(ngx_cycle_t *cycle)
{
struct sigaction  sa;
   ...
ngx_memzero(&sa, sizeof(struct sigaction));
   sa.sa_handler = ngx_timer_signal_handler;
   sigemptyset(&sa.sa_mask);
   if (ngx_timer_resolution && !(ngx_event_flags & NGX_USE_TIMER_EVENT)) {
       struct itimerval  itv;
       ...
       if (sigaction(SIGALRM, &sa, NULL) == -1) {          //初始化一个 SIGALRM 信号
           ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                         "sigaction(SIGALRM) failed");
           return NGX_ERROR;
       }
       itv.it_interval.tv_sec = ngx_timer_resolution / 1000;
       itv.it_interval.tv_usec = (ngx_timer_resolution % 1000) * 1000;
       itv.it_value.tv_sec = ngx_timer_resolution / 1000;
       itv.it_value.tv_usec = (ngx_timer_resolution % 1000 ) * 1000;
       if (setitimer(ITIMER_REAL, &itv, NULL) == -1) { //每隔一段时间产生一个SIGALRM 信号
           ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                         "setitimer() failed");
       }
   }
...
}
// ngx_event_process_init
```

从上面的源码片段可以看到，ngx_event_process_init()函数首先初始化一个信号 SIGALRM，然后调用setitimer()函数实现了每隔ngx_timer_resolution设置的时间产生一个SIGALRM信号。而SIGALRM信号在这里的信号处理函数为 ngx_timer_signal_handler()（由 sa.sa_handler 指针指向）:

```
// ngx_timer_signal_handler
static void
ngx_timer_signal_handler(int signo)
{
   ngx_event_timer_alarm = 1;
#if 1
   ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ngx_cycle->log, 0, "timer signal");
#endif
}
// ngx_timer_signal_handler
```

该函数非常简单，当信号发生时，就设置一个全局变量 ngx_event_timer_alarm。查看该全局变量的所有调用情况，发现全部出现在事件处理模型的相关实现源码中，我们以 ngx_epoll_process_events()为例，涉及的相关源码片段为：

```
// ngx_epoll_process_events
   ...
```

```
events = epoll_wait(ep, event_list, (int) nevents, timer);
    err = (events == -1) ? ngx_errno : 0;
    if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
        ngx_time_update();
    }
    …
// ngx_epoll_process_events
```

可以看到，epoll_wait()函数在处理事件的过程中，如果全局变量 ngx_event_timer_alarm 被赋值为 1，也即产生了 SIGALRM 信号，就能够产生中断进而调用 ngx_time_update()函数完成缓存时间的更新。

由于 SIGALRM 信号是按照 ngx_timer_resolution 变量保存的时间为间隔以一定的频率产生 SIGALRM 信号，因此也就可以保证程序按照指定的时间间隔更新缓存时间。

13.4 本章小结

在这一章中，我们主要学习了 Nginx 服务器程序的时间管理机制，其中涉及 Linux 系统调用的相关概念和频繁使用系统调用对应用程序的影响，这也是 Nginx 服务器程序为什么要谨慎处理时间管理问题的主要原因。本章学习的重点是对 Nginx 时间管理工作原理的理解，主要包括两点：一是如何避免对系统调用的频繁使用，二是对时间缓存读写操作的数据冲突问题的解决。Nginx 服务器程序在时间管理方面的设计思路和解决问题的办法值得我们深入探讨和借鉴。最后，我们还对缓存时间精度问题进行了讨论，这一部分内容体现出 Nginx 服务器程序在设计思想上的完整性和对用户的开放性，这也是值得我们称道的。

第 14 章

Nginx 的内存管理

在 C 语言程序设计中，内存的分配和管理是完全交由程序员来控制的，因此内存管理是每个程序员必须熟练掌握的技能之一。使用内存池的方式对程序使用的内存进行统一分配和回收，是当下比较时尚的做法，也是行之有效的管理手段，这能够在很大程度上降低内存管理的难度，减少程序的潜在缺陷，提高整个程序的稳定性。Nginx 服务器程序实现了这样的内存管理手段，本章我们将结合 Nginx 程序源码详细讨论相关的技术细节。

在本章中，我们将学习到以下知识点：

- 内存池的逻辑结构
- 内存池的创建、销毁以及重置操作
- 内存的申请、释放和回收操作

14.1 内存池的逻辑结构

我们在第 11 章学习 Nginx 服务器的重要数据结构时已经学习了用于描述 Nginx 内存池结构的 ngx_pool_t 结构体以及相关的 ngx_pool_data_t 结构体、ngx_pool_large_t 结构体和 ngx_pool_cleanup_t 结构体，对它们的定义细节有了完整的理解。在这里我们主要从逻辑结构对这些结构体的关系进行介绍和分析。

Nginx 服务器在实际运行过程中占用系统内存很少，这说明程序在对内存管理方面采取的措施对降低系统内存开销是非常有效果的。这样的优点既是 Nginx 服务器博得广大用户青睐的主要原因之一，也是我们在进行程序设计过程中应该借鉴和学习的。

Nginx 内存池本质上是一个链表结构，链表的每一个节点称为一个数据块，由 ngx_pool_data_t 结

构体描述。我们需要对内存池进行管理和分配，这依赖于 ngx_pool_t 结构体，可以认为该结构体描述了 Nginx 内存池的分配管理模块。遇到分配大数据内存的情况时，还要使用 ngx_pool_large_t 结构体，也形成一个链表，挂接在 ngx_pool_t 结构体上便于 Nginx 程序的管理。分配的内存使用完成后，需要对内存进行释放和回收，这需要借助 ngx_pool_cleanup_t 结构体，同样形成一个链表，挂接在 ngx_pool_t 结构体上。

我们通过示意图 14.1 来形象地展示各个结构体之间的关联关系。

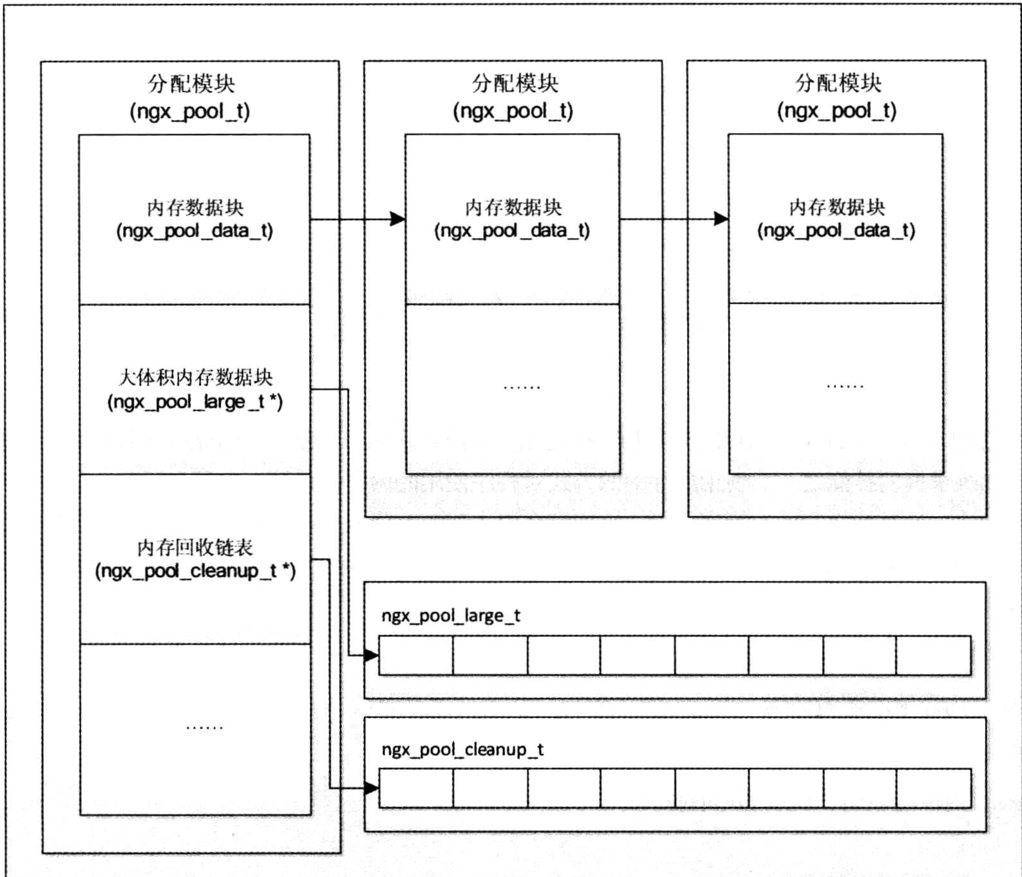

图 14.1　Nginx 的内存池结构示意图

示意图中的多个内存数据块形成一个链表，组成了 Nginx 服务器的内存池。Nginx 内存池的管理基于每个内存分配模块。用于大数据内存块分配的链表由 ngx_pool_large_t 结构体形成，挂接在每个分配模块上。用于内存回收的链表 ngx_pool_cleanup_t 结构体同样挂接在每个分配模块上。

14.2　内存池的管理

Nginx 内存池的管理涉及内存池的创建、销毁以及数据重置等操作，相关的源码可以在 /ngxin/src/core/ngx_palloc.c 文件中找到。我们通过对相关源码的分析来梳理相关的操作。

14.2.1　创建内存池

创建内存池的操作主要由 ngx_create_pool()函数完成，代码如下：

```
// ngx_create_pool
ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ngx_pool_t  *p;
    p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);  //分配进行了内存对齐操作的内存
    if (p == NULL) {
        return NULL;
    }
// ngx_create_pool
```

在该段源码中，调用 ngx_memalign()函数进行内存对齐操作。根据实际情况的不同，ngx_memalign()函数的具体执行情况也不相同。程序中对该函数的声明和定义是这样的：

```
#if (NGX_HAVE_POSIX_MEMALIGN || NGX_HAVE_MEMALIGN)
    void *ngx_memalign(size_t alignment, size_t size, ngx_log_t *log);//对齐内存并
分配内存
#else
    #define  ngx_memalign(alignment, size, log)  ngx_alloc(size, log) //直接分配内存
#endif
```

Linux 平台分配内存时有多个系统调用可以选择，如果不考虑数据对齐的问题，则有 malloc()；如果考虑到数据对齐，有两个系统调用可供选择：分别是 memalign()和 posix_memalign()。Nginx 程序在使用系统调用分配内存时，可以根据情况的不同选择这三种的一种。如果定义了 NGX_HAVE_POSIX_MEMALIGN，将调用 posix_memalign()：

```
#if (NGX_HAVE_POSIX_MEMALIGN)
void *
ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
{
    void  *p;
    int   err;
    err = posix_memalign(&p, alignment, size);
    if (err) {
        ...
    }
    ...
    return p;
}
```

如果定义了 NGX_HAVE_MEMALIGN，将调用 memalign()：

```
#elif (NGX_HAVE_MEMALIGN)
void *
ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
{
    void  *p;
```

```
    p = memalign(alignment, size);
    if (p == NULL) {
        ...
    }
    ...
    return p;
}
#endif
```

在以上两种情况下，alignment 变量的值来源于宏 NGX_POOL_ALIGNMENT，由它的定义：

```
#define NGX_POOL_ALIGNMENT       16
```

可以知道，这两种情况下分配的内存是以 16 字节为边界对齐的。如果 NGX_HAVE_POSIX_MEMALIGN 和 NGX_HAVE_MEMALIGN 均未定义，则调用 malloc()：

```
void *
ngx_alloc(size_t size, ngx_log_t *log)
{
    void  *p;
    p = malloc(size);
    if (p == NULL) {
        ...
    }
    ...
    return p;
}
```

通过调用 ngx_memalign()函数，Nginx 程序就向内存申请了一块内存区域，类型为 ngx_pool_t，这样的一块内存区域就可以用作内存池的一个内存分配模块，其中包含了类型为 ngx_pool_data_t 的数据存储空间（p->d）和用于挂载其他链表的指针空间等。

```
// ngx_create_pool
    p->d.last = (u_char *) p + sizeof(ngx_pool_t);
    p->d.end = (u_char *) p + size;
    p->d.next = NULL;
    p->d.failed = 0;
// ngx_create_pool
```

这里对内存分配模块中的内存池链表节点进行初始化赋值。

```
// ngx_create_pool
    size = size - sizeof(ngx_pool_t);
    p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_POOL;
// ngx_create_pool
```

这段代码中有一个重要的宏：

```
#define NGX_MAX_ALLOC_FROM_POOL  (ngx_pagesize - 1)
```

ngx_pagesize 中存放的是当前 Nginx 服务器运行的系统平台的一页内存页的大小，一般是 4KB 或者 8KB，我们这里假设为 4KB（x86 架构的机器上都是 4KB）。因此，当申请的内存空间大于 4KB 时，该内存空间可用于分配的最大内存数据块不能超过 4KB。

```
    p->current = p;
```

```
    p->chain = NULL;
    p->large = NULL;
    p->cleanup = NULL;
    p->log = log;
    return p;
}
```

这里对内存分配模块的其他成员初始化赋值。最后返回指向分配好的内存空间的指针。

为了更加清晰地展现内存池一个分配模块的物理结构，我们举一个例子来说明。比如，我们执行main()函数中的下列源码，向内存申请一块内存作为内存池的一个分配模块：

```
init_cycle.pool = ngx_create_pool(1024, log);
```

我们不妨假设申请的这块内存地址从 10 开始。执行完以上调用后，内存片段如图 14.2 所示。

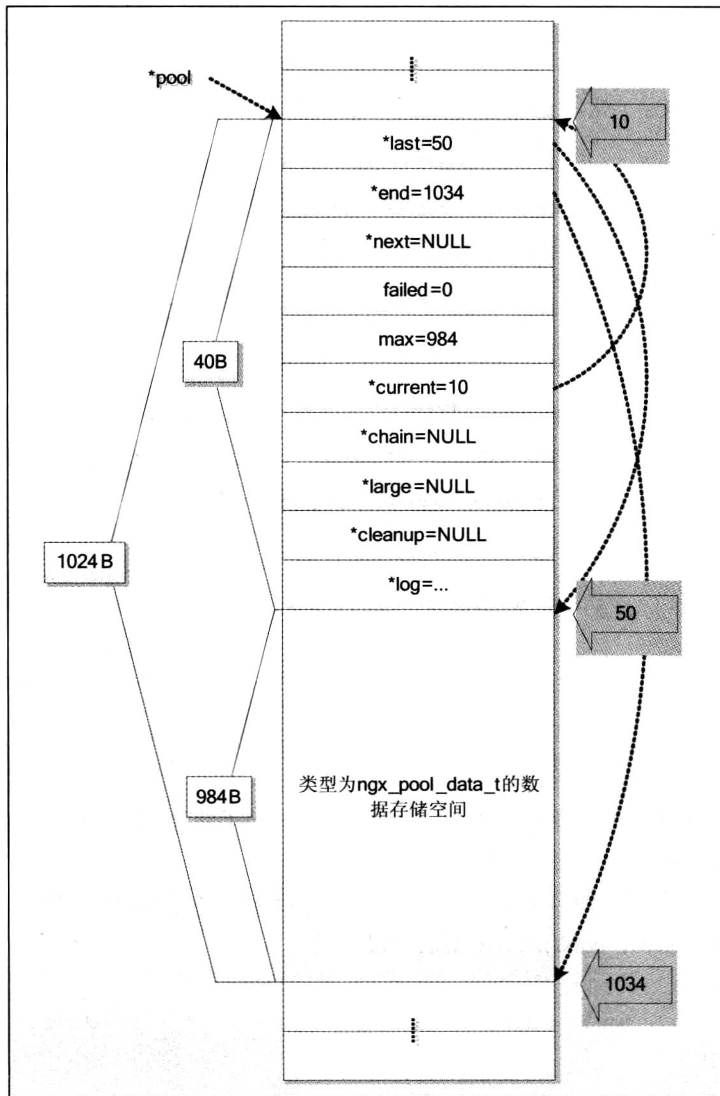

图 14.2　调用结果内存分配示意图

这段源码执行的结果是向内存分配了 1024 字节的空间供 init_cycle 结构使用。在物理内存中，申请到的整个空间被分为了两大部分，前面一部分是 ngx_pool_t 结构各个成员变量占用的空间，固定大小为 40 字节，剩余的 984 字节才是真正供 init_cycle 结构存放数据的。

以上就是 Nginx 内存池每个分配节点创建的主要原理。

14.2.2 销毁内存池

销毁内存池的工作主要由 ngx_destroy_pool()函数完成。代码如下：

```
// ngx_destroy_pool
void
ngx_destroy_pool(ngx_pool_t *pool)
{
    ngx_pool_t          *p, *n;
    ngx_pool_large_t    *l;
    ngx_pool_cleanup_t  *c;
    for (c = pool->cleanup; c; c = c->next) {            //遍历链表上的各个节点
        if (c->handler) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                           "run cleanup: %p", c);
            c->handler(c->data);                         //对该块内存中数据进行清除操作
        }
    }
// ngx_destroy_pool
```

在第 11 章中学习 ngx_pool_cleanup_s 结构体时我们知道，该结构体中的 handler 成员是一个函数指针，指向对该块内存中数据进行清除操作的处理函数。这一段代码完成的主要工作就是遍历该内存池链表中的各个节点，调用各节点的数据处理函数完成数据的清理工作。

```
// ngx_destroy_pool
    for (l = pool->large; l; l = l->next) {             //遍历链表上的各个节点
        ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free: %p", l->alloc);
        if (l->alloc) {
            ngx_free(l->alloc);                         //释放节点占用的内存空间
        }
    }
// ngx_destroy_pool
```

这一段源码是对大块数据内存的清理。其中调用 ngx_free()函数释放挂载在该内存池链表节点上的大块内存数据。ngx_free 是宏定义，就是 free：

```
#define  ngx_free          free
```

我们继续看 ngx_destroy_pool()函数的最后一段源码：

```
// ngx_destroy_pool
    for (p = pool, n = pool->d.next; // void ; p = n, n = n->d.next) {
        ngx_free(p);                                   //释放整个内存池占用的内存空间
        if (n == NULL) {
            break;
        }
```

```
    }
}
// ngx_destroy_pool
```

这一段源码是将指定的内存池本身占用的内存空间释放掉。

从上面源码可以看到，在整个内存池销毁的过程中，主要的工作有调用数据清理函数清理内存池中每个节点的数据，清理并释放挂接在内存池分配模块上的大数据块内存，释放内存池本身占用的内存空间。

14.2.3　重置内存池

重置内存池，顾名思义，就是将内存池恢复到初始分配的状态。这项工作是由 ngx_reset_pool()函数完成的。

```
void
ngx_reset_pool(ngx_pool_t *pool)
{
    ngx_pool_t       *p;
    ngx_pool_large_t  *l;
    for (l = pool->large; l; l = l->next) {
        if (l->alloc) {
            ngx_free(l->alloc);                              //释放大数据块内存
        }
    }
    pool->large = NULL;
    for (p = pool; p; p = p->d.next) {
        p->d.last = (u_char *) p + sizeof(ngx_pool_t);      //小数据块内存结尾指针指向刚
分配时的位置
    }
}
```

从源码可以看到，程序重置内存池的工作很简单，首先将挂接在分配模块上的大数据块内存释放掉，然后将指向小数据块内存结尾的 last 指针重置到刚分配时的位置。小数据块内存中存储的数据并没有被释放，其在以后内存池使用的过程中将被覆盖更新。

14.3　内存的使用

内存池创建好以后，如何向内存池申请内存空间呢？这些申请的内存在使用完后是如何重新回放到内存池中循环利用的呢？这是下面内容要解决的问题。

14.3.1　申请内存

Nginx 服务器程序向内存池申请内存的方法一共有三个，分别是调用 ngx_palloc()、ngx_pnalloc()和 ngx_calloc()函数。我们在第 12 章学习 Nginx 服务器初始化过程的相关内容时，多次遇到过申请内存的情况。比如在创建 cycle 结构的时候，向内存池申请 size 为一个 ngx_cycle_t 结构体大小的内存。

```
// main->ngx_init_cycle
```

```
    cycle = ngx_pcalloc(pool, sizeof(ngx_cycle_t));   //使用ngx_palloc()函数申请内存
    if (cycle == NULL) {
        ngx_destroy_pool(pool);
        return NULL;
    }
// main->ngx_init_cycle
```

这里程序调用的是 ngx_pcalloc()函数。再比如，在初始化 core 模块上下文时，向内存池申请 size 为一个 pid 字符串长度大小的内存：

```
// ngx_core_module_init_conf
    ccf->oldpid.data = ngx_pnalloc(cycle->pool, ccf->oldpid.len);     //同样使用
ngx_palloc()函数申请内存
    if (ccf->oldpid.data == NULL) {
        return NGX_CONF_ERROR;
    }
// ngx_core_module_init_conf
```

这里程序调用的是 ngx_pnalloc()函数。有关 ngx_calloc()函数的使用，我们暂时还没有碰到，在后面相关章节学习工作进程的工作原理时就会看到该函数的使用，比如，在工作进程进入事件处理循环之前会向内存池申请一些内存作为解析正则表达式的空间：

```
// ngx_worker_thread_cycle
    tls = ngx_calloc(sizeof(ngx_core_tls_t), cycle->log);              //使用
ngx_calloc()函数申请内存
    if (tls == NULL) {
        return (ngx_thread_value_t) 1;
    }
// ngx_worker_thread_cycle
```

那么这三个函数之间有什么不同呢？我们通过它们的源码来分析一下。

1. ngx_pcalloc()

该函数的实现源码在/nginx/src/core/ngx_pallpc.c 文件中可以找到：

```
// ngx_palloc
void *
ngx_palloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t  *p;
    if (size <= pool->max) {
        p = pool->current;
        do {
            m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);
            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size;                    //从Nginx的内存池中划出内存空间
                return m;
            }
            p = p->d.next;
        } while (p);
```

```
        return ngx_palloc_block(pool, size);
    }
// ngx_palloc
```

这是 ngx_pcalloc()函数的前一部分，传入的参数有两个：第一个是在哪个内存池上申请内存，第二个是申请内存的大小。进入函数后，首先是判断申请的内存大小是否超过该内存池允许分配的最大内存，如果没有，就从 pool->current 指针指向的内存池（链表）节点开始使用循环遍历以后的各个节点，找到满足申请大小的内存空间，设置 p->d.last 指针，并返回指向该空间的起始地址。在循环结构中一开始，程序调用了下面这个宏：

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

该操作用于计算以 NGX_ALIGNMENT 对齐数据后的偏移指针。实际上，我们最后分配到的内存空间是从该指针指向的地址开始的，而不是从*p 开始的，也就是说申请的内存大小 size=p->d.end-m，而不是 size= p->d.last - p->d.end。

如果遍历完整个内存池都没有找到满足申请大小的内存，则程序调用 ngx_palloc_block()函数。该函数实现了对内存池中内存空间的扩展，也就是申请一个新的内存池（链表）节点（程序中称为一个 block），然后挂接在内存池的最后面。

```
// ngx_palloc->ngx_palloc_block
static void *
ngx_palloc_block(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    size_t       psize;
    ngx_pool_t  *p, *new, *current;
    psize = (size_t) (pool->d.end - (u_char *) pool);
// ngx_palloc->ngx_palloc_block
```

这里计算当前内存池最后一个节点的大小 psize。该大小为要扩展的内存空间的大小。

```
// ngx_palloc->ngx_palloc_block
    m = ngx_memalign(NGX_POOL_ALIGNMENT, psize, pool->log);    //创建新的内存池节点
    if (m == NULL) {
        return NULL;
    }
// ngx_palloc->ngx_palloc_block
```

调用 ngx_memalign()创建新的内存池节点，大小为 psize。

```
// ngx_palloc->ngx_palloc_block
    new = (ngx_pool_t *) m;
    new->d.end = m + psize;
    new->d.next = NULL;
    new->d.failed = 0;
    m += sizeof(ngx_pool_data_t);
    m = ngx_align_ptr(m, NGX_ALIGNMENT);
    new->d.last = m + size;                         //分配要申请的大小为 size 的内存空间
// ngx_palloc->ngx_palloc_block
```

这一段程序对新的内存池节点中存储数据的空间进行初始化，并在该空间上分配要申请的大小为 size 的内存空间。

```
// ngx_palloc->ngx_palloc_block
    current = pool->current;
    for (p = current; p->d.next; p = p->d.next) {
        if (p->d.failed++ > 4) {
            current = p->d.next;                          //将申请好的节点加入到内存池
        }
    }
    p->d.next = new;
    pool->current = current ? current : new;
    return m;
}
// ngx_palloc->ngx_palloc_block
```

这一段程序将刚申请的节点挂接在内存池链表的末端，最后返回申请内存的起始地址。

以上申请的内存大小没有超过该内存池允许分配的最大内存的情况，如果超过怎么办呢？我们从前面的学习中已经知道，这种情况下，申请的内存将被看作是大数据块，将从大数据块内存链表上分配。我们来看下面的代码：

```
// ngx_palloc
    return ngx_palloc_large(pool, size);
}
// ngx_palloc
```

其中调用了 ngx_palloc_large()函数。该函数中实现了在大数据块内存链表上申请内存空间的功能。

```
// ngx_palloc->ngx_palloc_large
static void *
ngx_palloc_large(ngx_pool_t *pool, size_t size)
{
    void              *p;
    ngx_uint_t         n;
    ngx_pool_large_t  *large;
    p = ngx_alloc(size, pool->log);      //在大数据块内存链表中申请调用了 ngx_alloc()函数
    if (p == NULL) {
        return NULL;
    }
// ngx_palloc->ngx_palloc_large
```

程序调用了 ngx_alloc()函数申请内存空间。该函数是 Nginx 服务器程序底层的一个函数，实际上是被 ngx_calloc()函数调用分配内存的，我们在下面会介绍到。

```
// ngx_palloc->ngx_palloc_large
    n = 0;
    for (large = pool->large; large; large = large->next) {
        if (large->alloc == NULL) {
            large->alloc = p;
            return p;
```

```
        }
        if (n++ > 3) {                                  //循环 3 次退出
            break;
        }
    }
// ngx_palloc->ngx_palloc_large
```

该段代码循环三次，如果在三次内碰到大数据块内存链表上某个节点的数据为 NULL，则直接将各节点的数据指针指向上面申请好的空间并返回。

```
// ngx_palloc->ngx_palloc_large
    large = ngx_palloc(pool, sizeof(ngx_pool_large_t));//为大数据块链表申请新的节点空间
    if (large == NULL) {
        ngx_free(p);
        return NULL;
    }
// ngx_palloc->ngx_palloc_large
```

如果大数据块内存链表上的节点超过三个，则不再向后遍历，而是重新申请一块大小为 ngx_pool_large_t 结构体大小的内存，建立一个新的节点。

```
// ngx_palloc->ngx_palloc_large
    large->alloc = p;
    large->next = pool->large;
    pool->large = large;
    return p;
}
// ngx_palloc->ngx_palloc_large
```

最后，将新申请好的节点插入到大数据块内存链表的开头，返回申请的内存空间的起始地址。

2. ngx_pnalloc()

该函数的实现源码同样在/nginx/src/core/ngx_pallpc.c 文件中可以找到：

```
void *
ngx_pnalloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t  *p;
    if (size <= pool->max) {
        p = pool->current;
        do {
            m = p->d.last;
            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size;
                return m;
            }
            p = p->d.next;
        } while (p);
        return ngx_palloc_block(pool, size);
    }
```

```
    return ngx_palloc_large(pool, size);
}
```

对比该函数与 ngx_pcalloc()函数的实现，我们可以发现，两个函数在实现原理上是基本相同的，只是该函数在实现内存分配时不考虑内存数据对齐的问题。

3. ngx_calloc()

该函数的实现在/nginx/src/os/unix/ngx_alloc.c 文件中可以找到：

```
// ngx_ calloc
void *
ngx_calloc(size_t size, ngx_log_t *log)
{
    void  *p;
    p = ngx_alloc(size, log);                    //对malloc()调用的封装，马上就会看到
// ngx_ calloc
```

这是函数的前一段源码。函数接受的两个参数分别是要申请的内存大小和日志结构体。在实现中调用了 ngx_alloc()函数，该函数的实现也在相同的文件中：

```
// ngx_ calloc ->ngx_alloc
void *
ngx_alloc(size_t size, ngx_log_t *log)
{
    void  *p;
    p = malloc(size);
    if (p == NULL) {
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                      "malloc(%uz) failed", size);
    }
    ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, log, 0, "malloc: %p:%uz", p, size);
    return p;
}
// ngx_ calloc ->ngx_alloc
```

看到 ngx_alloc()函数的源码，我们就明白了，ngx_calloc()函数实际上使用 malloc()来直接从系统内存中申请内存，而不是从内存池中申请。这与前面两个函数有本质上的不同。

回到 ngx_ calloc()函数的源码继续，我们看到下面的操作：

```
// ngx_ calloc
    if (p) {
        ngx_memzero(p, size);                    //初始化内存空间
    }
    return p;
}
// ngx_ calloc
```

在这里程序调用 ngx_memzero()函数将刚才申请到的内存空间全部初始化为 0：

```
#define ngx_memzero(buf, n)       (void) memset(buf, 0, n)
```

这是与前面两个函数的另一个不同之处。

14.3.2 释放内存

Nginx 服务器程序释放内存的函数非常简单，和销毁内存池过程中使用的是同一个：

```
#define  ngx_free          free
```

不管是通过哪种方式申请的内存，都是使用这个函数进行释放的。这里需要说明的是，对于在各种不同场合下从内存池中申请的内存空间释放的时机是不一样的。一般只有大数据内存才直接调用 ngx_free()进行释放，其他数据空间的释放统统交给内存池销毁的过程了。

14.3.3 回收内存

在 Nginx 服务器程序中，有些数据类型在回收其所占的内存资源时不能直接通过释放内存空间的方式进行，而需要在释放之前对数据进行指定的处理操作。ngx_pool_cleanup_s 结构体的 handle 成员提供了这样的一个指针，可以指向释放内存的处理函数。将这些类型的数据结构存放到 ngx_pool_cleanup_s 结构体组成的内存回收链表中,就可以实现在释放内存前对数据进行指定处理的目的。

ngx_pool_cleanup_add()函数用于向内存回收链表中添加节点数据，在/nginx/src/core/ngx_palloc.c 文件中可以找到它的实现源码：

```
ngx_pool_cleanup_t *
ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)
{
    ngx_pool_cleanup_t  *c;
    c = ngx_palloc(p, sizeof(ngx_pool_cleanup_t));          //申请存放
ngx_pool_cleanup_t 结构体数据的空间
    if (c == NULL) {
        return NULL;
    }
    if (size) {
        c->data = ngx_palloc(p, size);              //申请存放目标数据的空间
        if (c->data == NULL) {
            return NULL;
        }
    } else {
        c->data = NULL;
    }
    c->handler = NULL;
    c->next = p->cleanup;
    p->cleanup = c;
    ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, p->log, 0, "add cleanup: %p", c)
    return c;
}
```

源码很简单，首先调用 ngx_palloc()函数申请一块内存作为存放 ngx_pool_cleanup_t 结构体数据的空间，然后再分配一块指定大小的内存作为程序存放目标数据的空间，该内存空间被挂接在 ngx_pool_cleanup_t 结构体的*data 成员上。最后，将新生成的 ngx_pool_cleanup_t 结构体挂接在内存回

收链表中。ngx_pool_cleanup_t 结构体的 handler 成员将在向*data 成员指向的空间填充数据时指定。

为了更好地理解回收内存机制，我们举一个 Nginx 服务器程序中的例子。该例子的主要功能是创建临时文件用于存放从后端服务器接收到的响应数据。这里涉及 ngx_pool_cleanup_t 结构体的*data 成员经常会指向的 ngx_pool_cleanup_file_t 结构体，它的定义如下：

```
typedef struct {
    ngx_fd_t              fd;
    u_char               *name;
    ngx_log_t            *log;
} ngx_pool_cleanup_file_t;
```

每个临时文件的文件描述符、路径、名称等信息都存放在它对应的 ngx_pool_cleanup_t 结构体中。Nginx 服务器程序调用 ngx_create_temp_file()函数创建临时文件。我们来看一下相关的代码片段。

```
// ngx_create_temp_file
ngx_int_t
ngx_create_temp_file(ngx_file_t *file, ngx_path_t *path, ngx_pool_t *pool,
    ngx_uint_t persistent, ngx_uint_t clean, ngx_uint_t access)
{
    …
    ngx_pool_cleanup_t      *cln;
    ngx_pool_cleanup_file_t  *clnf;
    file->name.len = path->name.len + 1 + path->len + 10;
    file->name.data = ngx_pnalloc(pool, file->name.len + 1);
    if (file->name.data == NULL) {
        return NGX_ERROR;
    }
    ngx_memcpy(file->name.data, path->name.data, path->name.len);
    n = (uint32_t) ngx_next_temp_number(0);
    cln = ngx_pool_cleanup_add(pool, sizeof(ngx_pool_cleanup_file_t));
    if (cln == NULL) {
        return NGX_ERROR;
    }
// ngx_create_temp_file
```

该段代码向内存回收链表申请了一块大小为 sizeof(ngx_pool_cleanup_file_t)的内存空间，临时文件的信息将存放在该块内存中。

```
// ngx_create_temp_file
    for ( ;; ) {
        …
        file->fd = ngx_open_tempfile(file->name.data, persistent, access);
        …
        if (file->fd != NGX_INVALID_FILE) {
            cln->handler = clean ? ngx_pool_delete_file : ngx_pool_cleanup_file;
                              //如果处理函数可用，则先调用函数对数据进行适当的处理，然后释放空间
            clnf = cln->data;
            clnf->fd = file->fd;
            clnf->name = file->name.data;
```

```
            clnf->log = pool->log;
            return NGX_OK;
        }
    }
    ...
}
// ngx_create_temp_file
```

这一段代码是将实际的临时文件信息与内存回收链表中刚才申请的内存空间关联到一起。可以看到 cln->handler 成员指向了临时文件的处理程序 ngx_pool_delete_file()或者 ngx_pool_cleanup_file()函数，而 cln->data 成员所指的申请好的内存空间赋值给 clnf 结构，用于存放信息。这样，当程序需要回收临时文件所占用的内存时，可以直接调用临时文件的处理程序先对数据进行适当的处理，然后再释放它所占用的内存空间。

14.4　本章小结

在前几章的基础上，这一章我们学习了 Nginx 服务器程序对内存和内存池的管理，其中详细学习了内存池的创建、销毁和重置机制，内存的申请、释放和回收机制。在学习过程中，我们一定要借助内存池的逻辑结构，深入理解相关的各项操作。

使用内存池对程序内存进行统一管理，是降低程序对系统资源的压力、提高程序稳定性的重要手段。通过本章的学习，相信大家对 Nginx 内存管理机制有了更深入的理解，这有助于我们进一步深入研究 Nginx 服务器程序的实现机制。而另一方面，Nginx 服务器对内存的高效管理方案对我们自己的程序设计具有很好的指导作用，给我们提供了不错的参考模型。

第 15 章

Nginx 工作进程

在本章中，我们详细介绍一下与 Nginx 服务器工作进程相关的内容，同时也对 Nginx 事件模型、Nginx 服务器主进程与工作进程之间的交互进行深入的分析。相信大家在学习完本章之后，能够对 Nginx 服务器的工作进程有一个更加深入的了解，对它采用的进程模型有一个更加清晰的认识。

在本章中，我们学习的主要内容有：

- Nginx 服务器工作进程主体工作的源码实现
- Nginx 服务器进程间通信机制

15.1 工作进程概览

在第 12 章中我们学习了工作进程的启动过程。Nginx 服务器主进程调用 fork()函数循环创建了 Nginx 服务器的所有工作进程，并且调用相关的函数来完成工作进程的具体工作。这些工作都是通过 Nginx 服务器程序调用 ngx_spawn_process()实现的，相关的源码为：

```
// main->ngx_master_process_cycle->ngx_start_worker_processes-> ngx_spawn_process
ngx_pid_t
ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
    char *name, ngx_int_t respawn)
{
    …
    pid = fork();                                              //创建子进程
    switch (pid) {
    case -1:                                                   //出错处理
        …
```

```
        return NGX_INVALID_PID;
    case 0:                                            //子进程中运行
        ngx_pid = ngx_getpid();
        proc(cycle, data);                             //指向工作进程要具体执行的函数
        break;
    default:                                           //主进程中返回子进程 PID
        break;
    }
    ...
}
// main->ngx_master_process_cycle->ngx_start_worker_processes-> ngx_spawn_process
```

传入的 proc 指针指向工作进程要具体执行的函数。ngx_spawn_process()函数的调用是这样的：

```
ngx_spawn_process(cycle, ngx_worker_process_cycle, (void *) (intptr_t) i, "worker
process", type);
```

那么，ngx_worker_process_cycle()就是工作进程执行工作的具体实现了。

根据图 13.2，Nginx 服务器的工作进程完成的主要工作是创建一个 Worker Process Cycle 处理事件。这里的事件就是客户端向 Nginx 服务器发起的网络请求。当然，工作进程在进行主要工作之前，还进行了初始化工作。笔者对这些工作进行了总结，主要包括以下几项：

- 设置工作进程运行环境
- 设置进程通信通道，监听和处理进程控制事件
- 设置工作进程标题
- 创建 Worker Process Cycle 循环机制，启动事件驱动机制
- 执行进程控制

接下来，我们从 ngx_worker_process_cycle()函数开始，通过源码分析来探讨上面各个工作具体做了哪些事情。

15.2 相关源码分析

ngx_worker_process_cycle()函数的实现过程我们在/nginx/src/os/unix/ngx_process_cycle.c 文件中可以找到，我们来详细分析一下。

```
// ngx_worker_process_cycle
static void
ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data)
{
    ngx_int_t worker = (intptr_t) data;                //获取进程上下文
    ngx_uint_t        i;
    ngx_connection_t  *c;
    ngx_process = NGX_PROCESS_WORKER;                  //标识为工作进程
    ngx_worker_process_init(cycle, worker);            //工作进程的初始化
// ngx_worker_process_cycle
```

这一段函数中需要关注的重点是调用 ngx_worker_process_init()函数完成工作进程的初始化工作。

15.2.1 设置工作进程运行环境

我们来详细看一下 ngx_worker_process_init()函数的实现源码：

```
// ngx_worker_process_cycle ->ngx_worker_process_init
static void
ngx_worker_process_init(ngx_cycle_t *cycle, ngx_int_t worker)
{
    sigset_t          set;
    uint64_t          cpu_affinity;
    ngx_int_t         n;
    ngx_uint_t        i;
    struct rlimit     rlmt;
    ngx_core_conf_t   *ccf;
    ngx_listening_t   *ls;
    if (ngx_set_environment(cycle, NULL) == NULL) {  //设置了 Nginx 服务器运行的环境变量
        // fatal
        exit(2);
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码设置了 Nginx 服务器运行的环境变量。在 Nginx 配置文件中，支持使用 evn 指令更改或者添加新的环境变量，其语法结构为：

```
env  variable  [= value]
```

- variable，变量名；
- value，变量值，如果不设置将继承运行启动 Nginx 服务器命令的 shell 环境的变量值。

该指令默认的 variable 值为“TZ”。ngx_set_environment()函数就是用来读取配置文件中该指令的 TZ 值的，并且将其应用于工作进程的运行环境中。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
     //获取配置上下文
    if (worker >= 0 && ccf->priority != 0) {
        if (setpriority(PRIO_PROCESS, 0, ccf->priority) == -1) {   //设置工作进程优先级
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "setpriority(%d) failed", ccf->priority);   //出错处理
        }
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码设置了 Nginx 服务器工作进程的优先级。同样，在 Nginx 配置文件中，支持使用 worker_priority 指令来设置工作进程的优先级，其语法结构为：

```
worker_priority  number
```

其中的 number 为优先级，允许范围为–20 ~ 20，负值优先级高于正值的优先级，默认为 0。程序调用 setpriority()函数设置工作进程的优先级。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    if (ccf->rlimit_nofile != NGX_CONF_UNSET) {
        rlmt.rlim_cur = (rlim_t) ccf->rlimit_nofile;        //获取配置信息
        rlmt.rlim_max = (rlim_t) ccf->rlimit_nofile;
        if (setrlimit(RLIMIT_NOFILE, &rlmt) == -1) {        //设置打开文件描述符数的上限
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "setrlimit(RLIMIT_NOFILE, %i) failed",
                          ccf->rlimit_nofile);              //出错处理
        }
    }
    if (ccf->rlimit_core != NGX_CONF_UNSET) {
        rlmt.rlim_cur = (rlim_t) ccf->rlimit_core;          //获取配置信息
        rlmt.rlim_max = (rlim_t) ccf->rlimit_core;
        if (setrlimit(RLIMIT_CORE, &rlmt) == -1) {          //设置内核转存文件的最大长度
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "setrlimit(RLIMIT_CORE, %O) failed",
                          ccf->rlimit_core);                //出错处理
        }
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段程序用来设置 Nginx 服务器工作进程使用打开文件描述符和一个 core 文件大小的资源上限。同样，在 Nginx 配置指令中使用 worker_rlimit_nofile 和 worker_rlimit_core 指令可以设置该值，其语法结构分别是：

```
worker_rlimit_nofile  size
worker_rlimit_core  number
```

程序调用 setrlimit()函数设置工作进程的这两个资源使用上限。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    if (geteuid() == 0) {
        if (setgid(ccf->group) == -1) {                             //设置组 ID
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "setgid(%d) failed", ccf->group);         //出错处理
            // fatal
            exit(2);
        }
        if (initgroups(ccf->username, ccf->group) == -1) {          //初始化组清单
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "initgroups(%s, %d) failed",
                          ccf->username, ccf->group);               //出错处理
        }
        if (setuid(ccf->user) == -1) {                              //设置用户 ID
            ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
                          "setuid(%d) failed", ccf->user);          //出错处理
            // fatal
            exit(2);
        }
```

```
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码的主要工作是设置工作进程运行环境的用户 ID，读取当前系统的组文件并初始化组，设置组 ID。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    if (worker >= 0) {
        cpu_affinity = ngx_get_cpu_affinity(worker);   //获取配置文件中的worker_cpu_
affinit 指令的配置
        if (cpu_affinity) {
            ngx_setaffinity(cpu_affinity, cycle->log); //解析并使配置生效
        }
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码是根据配置文件中 worker_cpu_affinity 指令的配置为工作进程分配 CPU 的工作内核。程序主要调用 ngx_setaffinity()函数来完成这一配置的应用。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    if (ccf->working_directory.len) {
        if (chdir((char *) ccf->working_directory.data) == -1) {  //设置进程的工作目录
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "chdir(\"%s\") failed", ccf->working_directory.data); //出错处理
            // fatal
            exit(2);
        }
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码是根据配置文件中 working_directory 指令的配置为工作进程设置当前工作目录。工作进程在执行过程中会在当前目录下写入运行数据，因此需要对工作目录有相应的权限。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    sigemptyset(&set);
    if (sigprocmask(SIG_SETMASK, &set, NULL) == -1) {          //改变目前的信号屏蔽字
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "sigprocmask() failed");                  //出错处理
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

该段源码的主要工作是取消对所有信号的屏蔽。sigprocmask()函数使用 SIG_SETMASK 参数本来是对 set 集合中的信号进行屏蔽，但是这里的 set 集合已经被清空。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    ls = cycle->listening.elts;
    for (i = 0; i < cycle->listening.nelts; i++) {
        ls[i].previous = NULL;                          //遍历并设置所有监听套接字的状态
    }
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->init_process) {
            if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) { //初始化Nginx 各个模块
```

```
            // fatal
            exit(2);
        }
    }
}
// ngx_worker_process_cycle ->ngx_worker_process_init
```

上面这段源码分别对 cycle->listening 中保存的所有监听套接字的 previous 状态设置为 NULL，并且初始化了所有的模块。这样，每个工作进程就把 Nginx 服务器程序中维护的主要资源都获取到了。

15.2.2　监听和处理进程控制事件

这一段源码对各个工作进程的通信管道进行了设置。首先，遍历所有其他的工作进程，调用 close()函数将它们用于监听的 channel[1]关闭；然后，遍历结束后再调用 close()函数将自己用于发送消息的 channel[0]关闭，只留下 channel[1]监听事件的到来。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    for (n = 0; n < ngx_last_process; n++) {
        if (ngx_processes[n].pid == -1) {
            continue;
        }
        if (n == ngx_process_slot) {                        //判别是否是当前的工作进程
            continue;
        }
        if (ngx_processes[n].channel[1] == -1) {            //判别进程间通信管道是否正常
            continue;
        }
        if (close(ngx_processes[n].channel[1]) == -1) {     //关闭其他进程的 channel[1]
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "close() channel failed");
        }
    }
    if (close(ngx_processes[ngx_process_slot].channel[0]) == -1) {   //关闭当前进程的
channel[0]
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "close() channel failed");
    }
// ngx_worker_process_cycle ->ngx_worker_process_init
```

这些事件一般是用来控制工作进程调整启停状态的。

```
// ngx_worker_process_cycle ->ngx_worker_process_init
    if (ngx_add_channel_event(cycle, ngx_channel, NGX_READ_EVENT,
ngx_channel_handler)
        == NGX_ERROR)                           //设置当前工作进程从 channel[1]中监听事件
    {
        // fatal
        exit(2);
    }
```

```
}
// ngx_worker_process_cycle ->ngx_worker_process_init
```

ngx_worker_process_init()函数的最后一段源码调用 ngx_add_channel_event()函数设置当前工作进程从 channel[1]中监听事件，我们看一下该函数的源码片段：

```
// ngx_worker_process_cycle ->ngx_worker_process_init-> ngx_add_channel_event
ngx_int_t
ngx_add_channel_event(ngx_cycle_t *cycle, ngx_fd_t fd, ngx_int_t event,
    ngx_event_handler_pt handler)
{
    ngx_event_t       *ev, *rev, *wev;
    ngx_connection_t  *c;
    c = ngx_get_connection(fd, cycle->log);        //为 ngx_connection_t 的成员赋值
    ……
    ev = (event == NGX_READ_EVENT) ? rev : wev;   //初始化监听任务
    ev->handler = handler;                         //设置事件处理函数
    if (ngx_add_conn && (ngx_event_flags & NGX_USE_EPOLL_EVENT) == 0) {
        ……
    } else {
        if (ngx_add_event(ev, event, 0) == NGX_ERROR) {    //将监听任务添加到任务队列
            ngx_free_connection(c);
            return NGX_ERROR;
        }
    }
    return NGX_OK;
}
// ngx_worker_process_cycle ->ngx_worker_process_init-> ngx_add_channel_event
```

函数先调用 ngx_get_connection()在指定的连接描述符（channel[1]的）上获取一个连接，然后再根据传入的 event 参数初始化一个监听任务 ev，最后将这个监听任务通过 ngx_add_event()添加到连接的事件监听任务队列。handle 指针指向的是事件处理函数，也就是 ngx_channel_handler()函数。我们在本章的后面会系统地介绍 Nginx 进程间通信和通信管道的相关内容。

接下来我们回到 ngx_worker_process_cycle()函数继续分析它的源码实现。函数的下一个主要工作是将工作进程的标题设置为“worker process”，调用的函数是 ngx_setproctitle()。

再接下来，ngx_worker_process_cycle()函数创建一个 for 循环结构，用来执行事件驱动机制，对客户端的网络请求事件进行监控。这是工作进程的主要工作。

15.2.3　接收网络请求事件

从这段源码片段可以看到，事件接收工作的所有实现是在 ngx_process_events_and_timers()函数中完成的：

```
// ngx_worker_process_cycle
for ( ;; ) {
    …
    ngx_process_events_and_timers(cycle);                       //处理网络请求事件
```

```
    …
}
// ngx_worker_process_cycle
```

我们详细分析一下该函数的主要源码。

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
void
ngx_process_events_and_timers(ngx_cycle_t *cycle)
{
    …
    if (ngx_use_accept_mutex) {                          //如果启用了accept_mutex 互斥量
        if (ngx_accept_disabled > 0) {
            ngx_accept_disabled--;                       //设置 accept_mutex 互斥量
        } else {
            …
        }
    }
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

该段源码是用来设置 accept_mutex 互斥量的。我们在第 2 章介绍 Nginx 服务器基础配置时，提到过对网络连接序列化的配置，其中谈到了“惊群现象”，并介绍了 accept_mutex 指令。该互斥量的主要作用是避免“惊群现象”的。如果 accept_mutex 指令设置为 on，就启用 accept_mutex 互斥量。

该互斥量还有一个重要作用，就是保证各个工作进程平衡接收连接请求事件。其在这一段源码中也有体现。为了说明该互斥量的这个作用，我们首先要了解全局变量 ngx_accept_disabled。该变量用来标记当前工作进程接收连接请求事件的能力，在函数 ngx_event_accept()中被设置：

```
// ngx_event_accept
void
ngx_event_accept(ngx_event_t *ev)
{
    …
    if (err == NGX_EMFILE || err == NGX_ENFILE) {
        if (ngx_disable_accept_events((ngx_cycle_t *) ngx_cycle) != NGX_OK) {  //是否接收了太多的事件
            return;
        }
        if (ngx_use_accept_mutex) {
            …
            ngx_accept_disabled = 1;                 //接收了太多的事件，则拒绝更多事件
        } else {
            …
        }
    }
    …
}
// ngx_event_accept
```

ngx_event_accept()函数用于实现工作进程接收连接请求事件的功能。在源码中我们看到，该函数

会调用 ngx_disable_accept_events()函数检测当前工作进程是否已经接收了太多连接请求事件，如果检测结构为真，则要将变量 ngx_accept_disabled 置为 1。这样，当各个工作进程在争抢 accept_mutex 互斥量的使用机会时就会因为 ngx_accept_disabled 大于 0 而放弃此次争抢，同时，ngx_accept_disabled 减 1。

我们回到 ngx_process_events_and_timers()函数继续分析刚才的 if 判断语句：

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
if (ngx_use_accept_mutex) {
    if (ngx_accept_disabled > 0) {
        …
    } else {
        if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {     //尝试获取
accept_mutex 互斥量
            return;
        }
        if (ngx_accept_mutex_held) {                          //获取互斥量成功
            flags |= NGX_POST_EVENTS;
        } else {                                              //获取互斥量失败
            if (timer == NGX_TIMER_INFINITE
                || timer > ngx_accept_mutex_delay)
            {
                timer = ngx_accept_mutex_delay;               //推迟一段时间重新尝试获取互斥量
            }
        }
    }
}
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

如果 ngx_accept_disabled 为 0，首先调用 ngx_trylock_accept_mutex()函数尝试获取 accept_mutex 互斥量，只有获取成功才继续向下执行。我们来简单看一下该函数的源码片段：

```
// ngx_worker_process_cycle
->ngx_process_events_and_timers->ngx_trylock_accept_mutex
ngx_int_t
ngx_trylock_accept_mutex(ngx_cycle_t *cycle)
{
    if (ngx_shmtx_trylock(&ngx_accept_mutex)) {
        ……
// ngx_worker_process_cycle
->ngx_process_events_and_timers->ngx_trylock_accept_mutex
```

程序首先获取 accept_mutex 互斥量，如果成功，执行下面的语句：

```
// ngx_worker_process_cycle
->ngx_process_events_and_timers->ngx_trylock_accept_mutex
        if (ngx_accept_mutex_held && ngx_accept_events == 0
            && !(ngx_event_flags & NGX_USE_RTSIG_EVENT)) {
            return NGX_OK;
        }
```

```
// ngx_worker_process_cycle
->ngx_process_events_and_timers->ngx_trylock_accept_mutex
```

如果网络连接上没有返回任何事件并且不是使用 rtsig 驱动机制，则直接返回获取 accept_mutex 互斥量成功的信息，但此时没有事件可以接收，ngx_accept_mutex_held 变量的值不设置为 1。

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers->ngx_trylock_accept_mutex
        if (ngx_enable_accept_events(cycle) == NGX_ERROR) {
            ngx_shmtx_unlock(&ngx_accept_mutex);
            return NGX_ERROR;
        }
        ngx_accept_events = 0;
        ngx_accept_mutex_held = 1;
        return NGX_OK;
    }
    …
    return NGX_OK;
}
// ngx_worker_process_cycle
->ngx_process_events_and_timers->ngx_trylock_accept_mutex
```

该段源码用来将事件接收到事件队列，并将 ngx_accept_mutex_held 变量的值设置为 1。最后返回获取 accept_mutex 互斥量成功的信息。

回到 ngx_process_events_and_timers()函数，开始执行之前源码中的最后一个 if 条件判断：

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
if (ngx_accept_mutex_held) {
    flags |= NGX_POST_EVENTS;                    //标记当前到来的事件已经被该工作进程接收
} else {
    if (timer == NGX_TIMER_INFINITE
        || timer > ngx_accept_mutex_delay)
    {
        timer = ngx_accept_mutex_delay;          //推迟一段时间
    }
}
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

函数获取互斥量以后，如果 ngx_accept_mutex_held 变量的值为 1，则首先在 flags 中添加 NGX_POST_EVENTS 标记，表明当前到来的事件已经被该工作进程接收（虽然事实上还没有处理），将其放入该工作进程的事件处理队列。如果 ngx_accept_mutex_held 变量的值不为 1，则设置一个延迟时间，延时一段时间后再重新尝试获取 accept_mutex 互斥量。

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
    delta = ngx_current_msec;
    (void) ngx_process_events(cycle, timer, flags);//调用各种事件驱动机制下的时间处理函数
    delta = ngx_current_msec - delta;
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

这里调用的 ngx_process_events()实际上调用的是各种事件驱动机制下的事件处理函数，我们以 epoll 机制为例，ngx_process_events()的实现实际上是 ngx_epoll_process_events()函数。该函数就调用

epoll_wait()函数处理接收到的事件。delta 变量用来在毫秒级别上记录此次处理事件耗费的时间。

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
   …
   if (ngx_posted_accept_events) {
      ngx_event_process_posted(cycle, &ngx_posted_accept_events);
   }
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

ngx_posted_accept_events 就是我们前面提到的存放事件的队列。这里开始处理这些事件。

```
   if (ngx_accept_mutex_held) {
      ngx_shmtx_unlock(&ngx_accept_mutex);
   }
```

释放 accept_mutex 互斥量。

```
// ngx_worker_process_cycle ->ngx_process_events_and_timers
   …
   if (ngx_posted_events) {                          //非网络请求事件
      if (ngx_threaded) {
         ngx_wakeup_worker_thread(cycle);
      } else {
         ngx_event_process_posted(cycle, &ngx_posted_events);
      }
   }
}
// ngx_worker_process_cycle ->ngx_process_events_and_timers
```

ngx_posted_events 变量中保存了连接上获取到的非网络连接请求事件，这些事件来源于事件驱动模块本身，都调用每个事件自己的处理方法进行处理。

到这里，我们就将 Nginx 服务器工作进程的主要工作——监听和处理网络请求事件分析完了。回到 ngx_worker_process_cycle()函数，我们继续解析剩余的源码。

15.2.4 执行进程控制

我们在第 15 章介绍了工作进程监听和处理进程控制事件的机制,其中提到的 ngx_channel_handler()函数就是进程控制事件处理函数。我们现在来看一下该函数的源码片段：

```
// ngx_channel_handler
static void
ngx_channel_handler(ngx_event_t *ev)
{
   …
   ngx_channel_t    ch;
   ngx_connection_t  *c;
   ……
   c = ev->data;
   …
   for ( ;; ) {
```

```
        n = ngx_read_channel(c->fd, &ch, sizeof(ngx_channel_t), ev->log);  //读取进
程通信信息
        …
        switch (ch.command) {                                        //解析通信信息
        case NGX_CMD_QUIT:
            ngx_quit = 1;
            break;
        case NGX_CMD_TERMINATE:
            ngx_terminate = 1;
            break;
        case NGX_CMD_REOPEN:
            ngx_reopen = 1;
            break;
        case NGX_CMD_OPEN_CHANNEL:
            …
            ngx_processes[ch.slot].pid = ch.pid;
            ngx_processes[ch.slot].channel[0] = ch.fd;
            break;
        case NGX_CMD_CLOSE_CHANNEL:
            …
            if (close(ngx_processes[ch.slot].channel[0]) == -1) {
                ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                              "close() channel failed");
            }
            ngx_processes[ch.slot].channel[0] = -1;
            break;
        }
    }
}
// ngx_channel_handler
```

代码的主体结构是一个 for 循环，在 for 循环中读取来自 channel[1]中的信息，ch 结构的 command 成员中存放了每条信息的指令。程序使用一个 switch 结构针对不同的指令进行相应的操作。其中，如果命令为 NGX_CMD_QUIT，将全局变量 ngx_quit 置为 1；命令为 NGX_CMD_TERMINATE，将全局变量 ngx_terminate 置为 1；命令为 NGX_CMD_REOPEN，将全局变量 ngx_reopen 置为 1。

再来看 ngx_worker_process_cycle()函数的最后一部分源码：

```
// ngx_worker_process_cycle
if (ngx_terminate) {                                   //根据指令执行终止命令
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exiting");
    ngx_worker_process_exit(cycle);
}
if (ngx_quit) {                                        //根据指令关闭监听套接字，退出程序
    ngx_quit = 0;
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
                  "gracefully shutting down");
    ngx_setproctitle("worker process is shutting down");
```

```
    if (!ngx_exiting) {
        ngx_close_listening_sockets(cycle);
        ngx_exiting = 1;
    }
}
if (ngx_reopen) {                                   //根据指令重新开启工作进程
    ngx_reopen = 0;
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
    ngx_reopen_files(cycle, -1);
}
// ngx_worker_process_cycle
```

了解了 ngx_channel_handler()函数的具体实现，这一段源码就容易理解了。程序使用了三个条件判断分别对全局变量 ngx_terminate、ngx_quit 和 ngx_reopen 进行判断，进而实施对工作进程的控制。而这三个变量是与 NGX_CMD_TERMINATE、NGX_CMD_QUIT 和 NGX_CMD_REOPEN 三个指令相关的。这样我们也就明白了控制工作进程的原理和具体实现。

15.3 Nginx 服务器进程间通信

当 Nginx 服务器使用多进程模式时，进程间需要进行频繁的通信，尤其是主进程和工作进程之间，主进程需要向子进程发送控制命令，还要能够方便地获取子进程的运行状态，同时，各个工作进程之间也是有交互的。进程间必须要实现一个高效稳定的双向通信渠道。

我们在前面分析 Nginx 主进程以及子进程的实现时，已经接触到了 Nginx 进程间通信的相关内容，但是没有深入分析，下面我们将专门就 Nginx 服务器进程间通信的相关主题做一个详细的说明。

15.3.1 Linux 进程间通信方式

Linux 平台上进程间通信的方式有很多，经典的解决方案包括以下几种：

- 使用 IPC（包括消息队列、信号量、共享存储）
- 管道
- 套接字 socket

这三种方式中，IPC 的三种方式功能很强大，但是不能支持 Nginx 服务器使用的事件驱动模型。

管道简单易用，但是限制也比较多，一般只是单向通道，只能在父子进程间使用，即使是流管道（可以进行双向通信）、命名管道（可以在不相关进程间使用）等也不能同时支持双向传输和不相关进程之间的通信。因此这两种进程间的通信方式在 Nginx 服务器实现时是不考虑的。

套接字 socket 用于网络通信，同时也能用于系统内进程间通信，并且这样的通信是双向通信，对进程的类型也没有特殊的要求。由于系统内进程的 IP 地址都是相同的，因此只需使用进程号就可以确定通信的双方。这正好满足了 Nginx 服务器进程间通信的要求。而且在 Linux 平台上有封装好的函数用来实现这一机制，使用起来非常方便。

15.3.2 Linux 进程间双工通信的实现

Linux 平台上使用 socketpair()函数创建用于进程间双向通信的 socket，该函数的原型为：

```
int socketpair(int d, int type, int protocol, int sv[2]);
```

使用该函数时，需要包含头文件 sys/types.h，这里定义了一些用到的宏常量。该函数的 4 个参数的含义分别是：

- d，socket 的域，一般设置为 AF_UNIX。
- type，socket 的类型，可以选择的有 SOCK_STREAM 和 SOCK_DGRAM，前者提供面向连接的可靠传输，后者提供面向无连接的传输，用于进程间通信选择前者。
- protocol，使用的协议，用于进程间通信时，赋值为 0。
- sv[2]，指向存储文件描述符的指针，也就是创建好的两个 socket 的文件描述符指针。

使用该函数创建好的一对套接字是一对未命名的相互连接的 UNIX 族套接字。在实际的使用中，完全可以把这一对 socket 当作普通的文件描述符进行读写操作。该函数在 Nginx 服务器初始化过程中的 ngx_spawn_process()函数中被调用：

```
if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
{
    ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                  "socketpair() failed while spawning \"%s\"", name);
    return NGX_INVALID_PID;
}
```

这一对套接字被称为 channel，分别是 channel[0]和 channel[1]，一般 channel[0]用于向其他进程发送消息，channel[1]用于监听其他进程发送来的消息事件。

在上边的源码中，socketpair()函数的最后一个参数为 ngx_processes[s].channel，这里的 ngx_processes 是一个 ngx_process_t 结构体。

通过这对套接字发送消息和接收消息的方法和网络 socket 的方式是相同的，都使用 read()和 write()函数，这里就不再详细介绍了。

15.3.3 通信通道的建立和设置

上面已经提到在 Nginx 服务器初始化过程中的 ngx_spawn_process()函数中完成了通信通道的建立。同时，这里也对通信通道进行了设置。我们来看完整的源码：

```
// ngx_spawn_process
if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
{                                                          //创建进程通信 socket
   ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
              "socketpair() failed while spawning \"%s\"", name);
   return NGX_INVALID_PID;
}
// ngx_spawn_process
```

该段源码调用 socketpair()函数创建进程通信 socket，前面已经介绍过了。

```
// ngx_spawn_process
```

```
...
if (ngx_nonblocking(ngx_processes[s].channel[0]) == -1) {    //设置 channel[0]非阻塞
    ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                  ngx_nonblocking_n " failed while spawning \"%s\"",
                  name);
    ngx_close_channel(ngx_processes[s].channel, cycle->log);
    return NGX_INVALID_PID;
}
if (ngx_nonblocking(ngx_processes[s].channel[1]) == -1) {   //设置 channel[1]非阻塞
    ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                  ngx_nonblocking_n " failed while spawning \"%s\"",
                  name);
    ngx_close_channel(ngx_processes[s].channel, cycle->log);
    return NGX_INVALID_PID;
}
// ngx_spawn_process
```

该段代码将创建好的 channel[0]和 channel[1]设置为非阻塞方式。ngx_nonblocking()函数是对 ioctl()系统调用的封装。

```
// ngx_spawn_process
on = 1;
if (ioctl(ngx_processes[s].channel[0], FIOASYNC, &on) == -1) {    //设置 channel[0]
接收异步 I/O 信号
    ...
}
// ngx_spawn_process
```

设置 channel[0]可以接收异步 I/O 信号。

```
// ngx_spawn_process
if (fcntl(ngx_processes[s].channel[0], F_SETOWN, ngx_pid) == -1) {   //设置接收信号
    ...
}
// ngx_spawn_process
```

设置 channel[0]可以接收 SIGIO 信号和 SIGURG 信号。

```
// ngx_spawn_process
if (fcntl(ngx_processes[s].channel[0], F_SETFD, FD_CLOEXEC) == -1) { //文件描述符
不传递到其他子进程
    ...
}
if (fcntl(ngx_processes[s].channel[1], F_SETFD, FD_CLOEXEC) == -1) { //同上
    ...
}

ngx_channel = ngx_processes[s].channel[1];
// ngx_spawn_process
```

设置 channel[0]和 channel[1]的文件描述符不传递到该进程的子进程。

以上的过程就是 Nginx 进程间通信管道的建立和设置。该段源码处于创建工作进程的循环结构中，会对每一个创建的子进程都生效。

15.3.4　通信通道的使用

Nginx 服务器程序创建工作进程时，主进程的资源会被子进程继承，因此子进程能够获取到之前创建的工作进程的通信通道，但由于进程之间的通信是双向的，以前创建的工作进程怎么能知道后面新创建的工作进程的通信通道呢？新的工作进程在创建完毕后会通过之前工作进程的通信通道通知它们自己的通道信息。这个工作是由 ngx_pass_open_channel()函数完成的。我们现在来详细看一下该函数的实现过程。

```
// ngx_pass_open_channel
static void
ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch)
{
    ngx_int_t  i;
    for (i = 0; i < ngx_last_process; i++) {                //遍历所有工作进程
        if (i == ngx_process_slot
            || ngx_processes[i].pid == -1
            || ngx_processes[i].channel[0] == -1)
        {
            continue;                                        //当前工作进程没有工作
        }
        ……
        ngx_write_channel(ngx_processes[i].channel[0],
                          ch, sizeof(ngx_channel_t), cycle->log);
    }
}
// ngx_pass_open_channel
```

源码很容易理解，在 for 循环结构中遍历之前创建好的所有工作进程，调用 ngx_write_channel()函数将自己的信息通知给所有这些工作进程。有关消息的读写我们马上就会介绍。

Nginx 服务器程序在创建好每个工作进程后，会调用 ngx_worker_process_init()函数进行相关的初始化工作，其中有一项工作是监听和处理进程控制事件，这就涉及到通信通道的使用。

15.3.5　消息的读写

最后，我们来介绍一下 Nginx 进程间通信的消息读写。首先我们先要明确 Nginx 服务器对进程间传递的消息是如何封装的。Nginx 服务器程序使用 ngx_channel_t 结构体来封装进程间的消息。该结构体我们在第 11 章中没有涉及，在这里补充一下。该结构体被定义在/nginx/src/os/unix/ngx_channel.h 文件中：

```
typedef struct {
    ngx_uint_t  command;
    ngx_pid_t   pid;
    ngx_int_t   slot;
    ngx_fd_t    fd;
```

```
} ngx_channel_t;
```

- command，发送给目标进程的指令，可选的指令有 NGX_CMD_OPEN_CHANNEL、NGX_CMD_CLOSE_CHANNEL、NGX_CMD_QUIT、NGX_CMD_TERMINATE、NGX_CMD_REOPEN 等 5 种。
- pid，发送消息进程的进程 ID。
- slot，发送消息进程在 Nginx 进程表中的索引。
- fd，发送消息进程接收消息的 socket 描述符，也就是发送消息进程的 channel[1]。

现在，我们来看一下 Nginx 服务器程序对消息写操作的实现，由 ngx_write_channel()函数完成：

```
// ngx_write_channel
ngx_int_t
ngx_write_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size,
   ngx_log_t *log)
{
   ssize_t            n;
   ngx_err_t          err;
   struct iovec       iov[1];
   struct msghdr      msg;
#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
   union {
      struct cmsghdr  cm;
      char           space[CMSG_SPACE(sizeof(int))];
   } cmsg;
   if (ch->fd == -1) {                                    //打开的管道出错
      msg.msg_control = NULL;
      msg.msg_controllen = 0;
   } else {
        //以下设置发送消息时的相关数据
      msg.msg_control = (caddr_t) &cmsg;
      msg.msg_controllen = sizeof(cmsg);
                                                          //设置消息发送缓冲区
      cmsg.cm.cmsg_len = CMSG_LEN(sizeof(int));
      cmsg.cm.cmsg_level = SOL_SOCKET;
      cmsg.cm.cmsg_type = SCM_RIGHTS;
                                                          //准备发送的信息复制到发送缓冲区
      ngx_memcpy(CMSG_DATA(&cmsg.cm), &ch->fd, sizeof(int));
   }
   msg.msg_flags = 0;
#else                                                     //不使用管道进行进程间通信
   if (ch->fd == -1) {
      msg.msg_accrights = NULL;
      msg.msg_accrightslen = 0;
   } else {
      msg.msg_accrights = (caddr_t) &ch->fd;
      msg.msg_accrightslen = sizeof(int);
```

```
    }
#endif
    iov[0].iov_base = (char *) ch;
    iov[0].iov_len = size;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    n = sendmsg(s, &msg, 0);                          //通过通信通道发送消息
    if (n == -1) {
        err = ngx_errno;
        if (err == NGX_EAGAIN) {
            return NGX_AGAIN;
        }
        ngx_log_error(NGX_LOG_ALERT, log, err, "sendmsg() failed");
        return NGX_ERROR;
    }
    return NGX_OK;
}
// ngx_write_channel
```

在该函数的实现中，有两条语句是最关键的。一条是调用 ngx_memcpy()函数将准备发送的信息复制到发送缓冲区 CMSG_DATA(cmsg)，这是一个宏定义，并考虑了内存对齐的因素：

```
#define CMSG_DATA(cmsg)     ((u_char *)(cmsg) + ALIGN(sizeof(struct cmsghdr)))
```

调用 ngx_memcpy()函数之前的工作是对发送缓冲区 cmsg 结构的初始化以及对 msg 结构成员的设置，都比较简单。第二条关键的语句是调用 sendmsg()函数通过通信通道发送消息。sendmsg()函数是 Linux 平台上的系统调用，用来通过 socket 发送消息。

消息的写操作也不复杂，主要是通过 ngx_read_channel()函数实现：

```
// ngx_ read _channel
ngx_int_t
ngx_read_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size, ngx_log_t *log)
{
    ssize_t           n;
    ngx_err_t         err;
    struct iovec      iov[1];
    struct msghdr     msg;
#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
    union {
        struct cmsghdr  cm;
        char            space[CMSG_SPACE(sizeof(int))];
    } cmsg;
#else
    int      fd;
#endif
    iov[0].iov_base = (char *) ch;
```

```
    iov[0].iov_len = size;
    //设置消息接收缓冲区
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
    msg.msg_control = (caddr_t) &cmsg;
    msg.msg_controllen = sizeof(cmsg);
#else
    msg.msg_accrights = (caddr_t) &fd;
    msg.msg_accrightslen = sizeof(int);
#endif
    n = recvmsg(s, &msg, 0); //接收消息
// ngx_ read _channel
```

函数的前一段定义了接收缓冲区 cmsg，并且初始化了 msg 结构，接收到的消息最后要存放在该结构中。重要的语句是调用 recvmsg()函数完成对消息的接收工作。该函数是 Linux 平台上专门用于接收来自 socket 的消息的系统调用。

```
// ngx_ read _channel
    if (n == -1) {                                          //接收消息出错
        …
        return NGX_ERROR;
    }
    if (n == 0) {                                           //接收到的消息长度为 0
        ngx_log_debug0(NGX_LOG_DEBUG_CORE, log, 0, "recvmsg() returned zero");
        return NGX_ERROR;
    }
    if ((size_t) n < sizeof(ngx_channel_t)) {               //接收到的消息有丢失
        …
        return NGX_ERROR;
    }
// ngx_ read _channel
```

该段源码对接收到的消息进行判断，处理异常情况。

```
// ngx_ read _channel
#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
    if (ch->command == NGX_CMD_OPEN_CHANNEL) {              //按照指令处理管道里的消息
        //以下判断消息的正确性
        if (cmsg.cm.cmsg_len < (socklen_t) CMSG_LEN(sizeof(int))) {
            ……
            return NGX_ERROR;
        }
        if (cmsg.cm.cmsg_level != SOL_SOCKET || cmsg.cm.cmsg_type != SCM_RIGHTS)
        {
            ……
            return NGX_ERROR;
```

```
        }
        ngx_memcpy(&ch->fd, CMSG_DATA(&cmsg.cm), sizeof(int));  //提取缓冲区中的消息
    }
    if (msg.msg_flags & (MSG_TRUNC|MSG_CTRUNC)) {
        ngx_log_error(NGX_LOG_ALERT, log, 0, "recvmsg() truncated data");
    }
#else
    if (ch->command == NGX_CMD_OPEN_CHANNEL) {
        if (msg.msg_accrightslen != sizeof(int)) {
            ……
            return NGX_ERROR;
        }
        ch->fd = fd;
    }
#endif
    return n;
}
// ngx_ read _channel
```

函数的最后一部分源码将接收到的消息从接收缓冲区中提取出来并保存到 msg 结构中。

15.4 本章小结

本章的主要内容实际上是对第 12 章内容的延续和补充,通过对源码的分析梳理了 Nginx 服务器工作进程的工作流程，重点介绍了运行环境的设置、进程控制事件的接收和处理、网络请求事件的接收和处理。还介绍了 Nginx 主进程和工作进程之间的进程通信机制，其采取的实现方案和源码设计是 Nginx 服务器可以高效运行的一个重要原因，具有很有益的参考价值。

第 16 章

Nginx 的模块编程

Nginx 服务器作为优秀高效的 Web 服务器，从基本核心功能到其他附加功能，都是通过模块来实现的，并且也支持第三方功能模块的增加。本章与前面几章不同，本章将从“模块”这样一个新的角度切入，简单介绍 Nginx 服务器模块编程的基本方法和应该遵循的基本原则，讨论 Nginx 大量模块之间如何协同工作等问题，让大家从一个新的角度加深对 Nginx 服务器模块化架构的理解，掌握编写 Nginx 模块的基本方法和规则。

在本章中，我们学习的主要内容有：

- Nginx 模块的分类。
- “Hello_Nginx”模块开发示范。
- 模块结构分析。
- 模块的编译和安装。

16.1 模块的种类

Nginx 服务器程序中的模块根据不同划分方法有不同的划分结果。我们在第 3 章中，根据模块在 Nginx 服务器中承担作用的重要程度，将模块划分为核心模块、标准模板、可选 HTTP 模块和第三方模块。

- 核心模块：Nginx 服务器中的重要模块，其提供了 Nginx 服务器最基本的初始化启动功能，包括网络管理、文件管理、内存管理、配置解析、模块加载等基本功能的实现。
- 标准模块：Nginx 服务器编译配置时无需指明编译的重要模块，其提供了用于实现 HTTP Web 服务的基础功能，包括代理、反向代理、URL 重写、GZIP 等功能的实现。邮件功能虽然在

实际应用中使用的频率不是很高，但从历史角度来看，也应该归为该类。

- 可选 HTTP 模块：Nginx 服务器发布时自带的，但编译时需要指明编译 HTTP 模块，其提供了 HTTP Web 服务提供过程中经常使用的功能。这类模块大多数是为了提高 Web 服务器的性能或者扩展额外功能而提供的，比如 SSL 功能的支持、select 和 poll 事件驱动的支持等。
- 第三方模块：不属于 Nginx 服务器发布时自带的模块，是由其他开发人员根据实际应用情况对 Nginx 服务器的功能、性能等进行扩展、改进等而自行编写供大家使用的模块。部分第三方模块虽然不是由官方提供，但也是十分优秀的，在实际的应用过程中是对 Nginx 服务器的极大补充。

在第 12 章中，根据 Nginx 服务器程序源码的实现，又将模块划分为 core 模块、http 模块、event 模块和 mail 模块 4 种，这实际上是根据模块的不同功能进行划分的。

- core 模块：主要负责 Nginx 服务器的核心功能。
- http 模块：主要负责 Nginx 服务器的 HTTP Web 服务。
- event 模块：主要负责 Nginx 服务器的事件处理服务。
- mail 模块：主要负责 Nginx 服务器的邮件服务。

Nginx 服务器的模块还可以按照角色的不同划分为请求处理模块、资源过滤模块和代理转发模块。这三种角色在 Nginx 服务器处理一次完整网络请求的过程中各自负责自己的事务。

- 请求处理模块：主要负责与客户端网络连接、请求收发的处理。
- 资源过滤模块：主要负责对 Nginx 服务器接收到的各类网络资源进行管理和筛选。资源来源于客户端或者后端服务器，比如客户端的 URL、后端服务器返回的响应数据等。
- 代理转发模块：主要负责 Nginx 服务器和后端服务器的交互工作，比如实现后端服务器的选择、资源递送等服务。

进行 Nginx 服务器模块编程的主要目的就是为了让 Nginx 服务器在实际的应用过程中能够在功能上按照我们自己的要求有所扩展，或者在行为上能够按照我们的需求有所改变，也可能是在性能上针对我们的实际环境进行优化。而 Nginx 服务器在实际应用中很大程度上是被用于 Web 服务器，因此，在编写第三方模块时，更多的是将这些模块按照角色划分，从而在思路上对它们承担什么样的角色、完成什么样的功能有清晰的认识。

16.2　模块开发实践

在本节中，我们从“Hello_Nginx”这样一个第三方模块的源码实例开始，逐步分析一个 Nginx 服务器模块应该包含的基本结构和编写过程中应该遵循的步骤和基本原则。该模块是一个 http 类模块，但实际上 Nginx 服务器中的所有模块都是按照同一个基本的结构和原则编写完成的。

16.2.1　“Hello_Nginx”模块编程实例

编写一个 Nginx 模块实际上和编写一个普通 C 语言程序没有太大的区别，主要是在源码的具体实现上，为了让 Nginx 服务器能够识别我们编写的模块，需要按照一定的要求定义数据结构，并且要遵

循一定的命名规范。在这里我们先不多讲，直接向大家展示一个“Hello_Nginx”模块的具体源码。该源码实现了这样的简单功能：当客户端访问 Nginx 服务器时，可以在浏览器中打印出“Hello, my new Nginx!”和配置文件中的附加信息。

1. 创建目录结构

该模块源码较简单，我们在某个目录中建立一个目录名为 ngx_http_hello_nginx_module 的目录。笔者将其建在了 Linux 平台的/home/中：

```
mkdir /home/ngx_http_hello_nginx_module
```

目录中新建两个文件：

```
touch /home/ngx_http_hello_nginx_module/{ngx_http_hello_nginx_module.c,config}
```

其中，ngx_http_hello_nginx_module.c 文件中存放“Hello_Nginx”模块的源码实现，config 文件中存放该模块的配置信息，我们后文会讲到。

2. 编写源码

打开 ngx_http_hello_nginx_module.c 文件，在其中输入以下内容。

```
   #include <ngx_config.h>
   #include <ngx_core.h>
   #include <ngx_http.h>
  typedef struct {
      ngx_str_t output_words;
   } ngx_http_hello_nginx_loc_conf_t;                          //定义配置上下文
 static char* ngx_http_hello_nginx(ngx_conf_t* cf, ngx_command_t* cmd, void* conf);
  static void* ngx_http_hello_nginx_create_loc_conf(ngx_conf_t* cf);
   static char* ngx_http_hello_nginx_merge_loc_conf(ngx_conf_t* cf, void* parent,
void* child);
   static ngx_command_t ngx_http_hello_nginx_commands[] = {  //定义模块支持的指令数组
     {
        ngx_string("hello_nginx"),                              //模块名称
        NGX_HTTP_LOC_CONF | NGX_CONF_TAKE1,                      //合法位置和参数个数
       ngx_http_hello_nginx,
        NGX_HTTP_LOC_CONF_OFFSET,
       offsetof(ngx_http_hello_nginx_loc_conf_t, output_words), //在 location 块中
的偏移
        NULL
     },
     ngx_null_command
  };
 u_char ngx_print[] = "Hello, my new Nginx!";                   //要输出的内容
static ngx_http_module_t ngx_http_hello_nginx_module_ctx = { //定义模块上下文结构体
     NULL,
     NULL,
     NULL,
     NULL,
     NULL,
     NULL,
```

```
    ngx_http_hello_nginx_create_loc_conf,
    ngx_http_hello_nginx_merge_loc_conf
};
ngx_module_t ngx_http_hello_nginx_module = {            //定义模块的主结构体
    NGX_MODULE_V1,
    &ngx_http_hello_nginx_module_ctx,
    ngx_http_hello_nginx_commands,
    NGX_HTTP_MODULE,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NGX_MODULE_V1_PADDING
};
static ngx_int_t ngx_http_hello_nginx_handler(ngx_http_request_t* r) {
    ngx_int_t rc;
    ngx_buf_t* b;
    ngx_chain_t out[2];
    ngx_http_hello_nginx_loc_conf_t* hlcf;
    hlcf = ngx_http_get_module_loc_conf(r, ngx_http_hello_nginx_module);
    r->headers_out.content_type.len = sizeof("text/plain") - 1;
    r->headers_out.content_type.data = (u_char*)"text/plain";
    b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    out[0].buf = b;
    out[0].next = &out[1];
    b->pos = (u_char*)ngx_print;
    b->last = b->pos + sizeof(ngx_print) - 1;
    b->memory = 1;
    b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    out[1].buf = b;
    out[1].next = NULL;
    b->pos = hlcf->output_words.data;
    b->last = hlcf->output_words.data + (hlcf->output_words.len);
    b->memory = 1;
    b->last_buf = 1;
    r->headers_out.status = NGX_HTTP_OK;
    r->headers_out.content_length_n = hlcf->output_words.len + sizeof(ngx_print) - 1;
    rc = ngx_http_send_header(r);
    if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
        return rc;
    }
    return ngx_http_output_filter(r, &out[0]);
}
```

```
  static void* ngx_http_hello_nginx_create_loc_conf(ngx_conf_t* cf) {
     ngx_http_hello_nginx_loc_conf_t* conf;
    conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_hello_nginx_loc_conf_t));
     if (conf == NULL) {
        return NGX_CONF_ERROR;
    }
    conf->output_words.len = 0;
    conf->output_words.data = NULL;
    return conf;
}
 static char* ngx_http_hello_nginx_merge_loc_conf(ngx_conf_t* cf, void* parent,
void* child) {
    ngx_http_hello_nginx_loc_conf_t* prev = parent;
    ngx_http_hello_nginx_loc_conf_t* conf = child;
    ngx_conf_merge_str_value(conf->output_words, prev->output_words, "Nginx");
   return NGX_CONF_OK;
}
 static char* ngx_http_hello_nginx(ngx_conf_t* cf, ngx_command_t* cmd, void* conf)
{    //模块入口函数
   ngx_http_core_loc_conf_t* clcf;
    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);      //传
入模块主结构体
    clcf->handler = ngx_http_hello_nginx_handler;              //见 55 行的处理函数
    ngx_conf_set_str_slot(cf, cmd, conf);
    return NGX_CONF_OK;
}
```

以上就是“Hello_Nginx”模块的源码实现，虽然比较简单，但是其中包含了编写一个 Nginx 第三方模块用到的基本数据结构和源码组织结构，我们来分别介绍一下。

16.2.2　模块的结构

“Hello_Nginx”模块中用到了许多数据结构，这些数据结构在 Nginx 模块开发中是必不可少的。Nginx 服务器程序就是通过这些数据结构来识别我们编写的模块、解析配置文件中使用的指令、查找和调用模块的执行函数的。

1. ngx_module_t 结构体

我们在第 11 章学习 Nginx 服务器程序基本结构体时，对 ngx_module_s 结构体进行过详细的讨论，该结构体是所有 Nginx 模块的基础数据结构。在“Hello_Nginx”模块中，该结构体以下行所示被使用：

```
ngx_module_t ngx_http_hello_nginx_module = {
    NGX_MODULE_V1,
   &ngx_http_hello_nginx_module_ctx,
    ngx_http_hello_nginx_commands,
    NGX_HTTP_MODULE,
    NULL,
    NULL,
    NULL,
```

```
    NULL,
    NULL,
    NULL,
    NULL,
    NGX_MODULE_V1_PADDING
};
```

在 ngx_http_hello_nginx_module 结构中，加粗代码 NGX_MODULE_V1 和加粗代码 NGX_MODULE_V1_PADDING 分别用 Nginx 程序中的两个宏将结构体的前 7 个成员和后 8 个成员初始化为 0（版本标识初始化为 1）：

```
#define NGX_MODULE_V1    0, 0, 0, 0, 0, 0, 1
#define NGX_MODULE_V1_PADDING  0, 0, 0, 0, 0, 0, 0, 0
```

在 ngx_http_hello_nginx_module_ctx 结构中存放该模块的上下文信息，在 ngx_http_hello_nginx_commands 结构中存放该模块在配置文件中使用的指令及其相关信息，这两个结构我们马上就要分析到。NGX_HTTP_MODULE 宏指明我们编写的模块是 http 模块。斜体加粗部分这 7 行是模块在主进程初始化、模块初始化、工作进程初始化、线程初始化、线程退出、工作进程退出和主进程退出的时候被调用的回调函数，我们在这里没有使用，因此置为 NULL。

2. 模块上下文结构

第 11 章讲过，HTTP 类模块的模块上下文结构体是 ngx_http_module_t。在 Hello_Nginx 模块中，该结构体在以下行被使用：

```
static ngx_http_module_t ngx_http_hello_nginx_module_ctx = {
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    ngx_http_hello_nginx_create_loc_conf,
    ngx_http_hello_nginx_merge_loc_conf
};
```

可以看到，在所有的回调函数指针中，我们使用了*create_loc_conf 指针和*merge_loc_conf 指针。*create_loc_conf 指针指向的函数 ngx_http_hello_nginx_create_loc_conf()在初始化 location 块之前调用，*merge_loc_conf 指针指向的函数 ngx_http_hello_nginx_merge_loc_conf()实现合并 location 块和 server 块中 hello_nginx 指令的配置。这两个函数的实现为：

```
static void* ngx_http_hello_nginx_create_loc_conf(ngx_conf_t* cf);
static char* ngx_http_hello_nginx_merge_loc_conf(ngx_conf_t* cf, void* parent, void*
child);
...
static void* ngx_http_hello_nginx_create_loc_conf(ngx_conf_t* cf) {
    ngx_http_hello_nginx_loc_conf_t* conf;
  conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_hello_nginx_loc_conf_t));
    if (conf == NULL) {
      return NGX_CONF_ERROR;
```

```
    }
    conf->output_words.len = 0;
    conf->output_words.data = NULL;
    return conf;
}
static char* ngx_http_hello_nginx_merge_loc_conf(ngx_conf_t* cf, void* parent,
void* child) {
    ngx_http_hello_nginx_loc_conf_t* prev = parent;
    ngx_http_hello_nginx_loc_conf_t* conf = child;
    ngx_conf_merge_str_value(conf->output_words, prev->output_words, "Nginx");
    return NGX_CONF_OK;
}
```

第一个函数是在初始化 location 块之前为 ngx_http_hello_nginx_loc_conf_t 结构分配空间并初始化。第二个函数的实现相对比较固定，就是调用 ngx_conf_merge_str_value()函数来合并相同指令的配置。

3. ngx _command_t 结构体

接下来使用到的重要结构体是 ngx_command_t，我们在第 11 章学习过该结构体的定义细节。“Hello_Nginx”模块源码中以下代码使用该结构体保存模块在配置文件中使用的指令及其相关信息。

```
static ngx_command_t ngx_http_hello_nginx_commands[] = {
    {
        ngx_string("hello_nginx"),
        NGX_HTTP_LOC_CONF | NGX_CONF_TAKE1,
        ngx_http_hello_nginx,
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(ngx_http_hello_nginx_loc_conf_t, output_words),
        NULL
    },
    ngx_null_command
};
```

我们注意到，ngx_http_hello_nginx_commands 实际上是一个结构体数组，每一个元素的类型为 ngx_command_t 结构体，因此我们可以在其中定义多个指令。在这里我们只定义了一条指令 hello_nginx。根据结构体各个成员的含义，该指令可以在 location 块中进行配置（NGX_HTTP_LOC_CONF），且只支持一个参数（NGX_CONF_TAKE1），指令参数的转换函数为 ngx_http_hello_nginx():

```
static char* ngx_http_hello_nginx(ngx_conf_t* cf, ngx_command_t* cmd, void* conf);
...
static char* ngx_http_hello_nginx(ngx_conf_t* cf, ngx_command_t* cmd, void* conf) {
    ngx_http_core_loc_conf_t* clcf;
    clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
    clcf->handler = ngx_http_hello_nginx_handler;
    ngx_conf_set_str_slot(cf, cmd, conf);
    return NGX_CONF_OK;
}
```

在该函数中，首先调用 ngx_http_conf_get_module_loc_conf()函数获取 location 块的配置结构，然后为该结构设置一个回调函数，该函数就是 hello_nginx 指令的处理函数 ngx_http_hello_nginx_handler()。由

于该指令支持参数，因此还要调用 ngx_conf_set_str_slot()函数获取指令的参数并把参数存储在配置文件结构体中。

接下来我们简单看一下 hello_nginx 指令的处理函数 ngx_http_hello_nginx_handler()，其实现很简单：

```
static ngx_int_t ngx_http_hello_nginx_handler(ngx_http_request_t* r) {
    ngx_int_t rc;
    ngx_buf_t* b;
    ngx_chain_t out[2];
    ngx_http_hello_nginx_loc_conf_t* hlcf;
    hlcf = ngx_http_get_module_loc_conf(r, ngx_http_hello_nginx_module);
    r->headers_out.content_type.len = sizeof("text/plain") - 1;
    r->headers_out.content_type.data = (u_char*)"text/plain";
    b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    out[0].buf = b;
    out[0].next = &out[1];
     b->pos = (u_char*)ngx_print;
     b->last = b->pos + sizeof(ngx_print) - 1;
     b->memory = 1;
     b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
     out[1].buf = b;
     out[1].next = NULL;
     b->pos = hlcf->output_words.data;
     b->last = hlcf->output_words.data + (hlcf->output_words.len);
     b->memory = 1;
     b->last_buf = 1;
     r->headers_out.status = NGX_HTTP_OK;
     r->headers_out.content_length_n = hlcf->output_words.len + sizeof(ngx_print) - 1;
     rc = ngx_http_send_header(r);
     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
         return rc;
     }
   return ngx_http_output_filter(r, &out[0]);
 }
```

该函数内部主要完成应答客户端请求的工作。ngx_http_request_t 结构的成员 header_in 中包含有客户端请求头部的内容，而我们要发送给客户端的应答存放在该结构体的成员 header_out 中。

ngx_http_hello_nginx_loc_conf_t 结构是自定义的结构体，代码如下：

```
typedef struct {
    ngx_str_t output_words;
 } ngx_http_hello_nginx_loc_conf_t;
```

该结构体中只有一个成员 output_words，它的作用是什么呢？在配置文件中启用“Hello_Nginx”模块时使用 hello_nginx 指令，该指令是我们自己定义的，它可以拥有一个参数。该结构体中的 output_words 成员中存放的就是这个参数。如果不需要给 hello_nginx 指令提供参数支持，就可以省略该结构体了。

另外，ngx_print 字符串是要在客户端浏览器上打印出来的一部分字符串：

```
u_char ngx_print[] = "Hello, my new Nginx!";
```

比如我们在配置文件中使用 hello_nginx 指令的配置是：

```
hello_nginx " This is my world!"
```

那么，最后在客户端浏览器上打印出来的字符串应该是：Hello, my new Nginx! This is my world!

16.2.3 模块命名规则

从上面“Hello_Nginx”模块的开发过程中，我们看到模块的命名和源码中各种变量的命名是有一定的规则的，这是在模块开发过程中最值得注意的一个原则，它包括两方面的含义。一方面，第三方模块本身的命名要规范；另一方面，在实现模块功能的源码中，各种变量的命名要规范。我们还是以“Hello_Nginx”模块为例来说明。

模块本身的名字我们命名为 ngx_http_hello_nginx_module，规则是：

ngx_模块的类型_模块的功能描述_module

在创建模块目录结构时，为了便于模块源码的管理，建议大家对模块源码顶层目录的目录名也使用模块的名称，在给该目录中的文件命名时，模块的主源码文件命名为：

模块名.c

比如，ngx_http_hello_nginx_module.c，配置文件的命名 config 是固定的。

在模块的实现过程中，主要的数据结构在命名时遵循以下原则：

ngx_module_t 类型的结构体命名为“ngx_模块的类型_模块的功能描述_module”，也就是该模块本身的名称，比如 ngx_module_t ngx_http_hello_nginx_module。

ngx_command_t 类型的结构体数组命名为“ngx_模块的类型_模块的功能描述_commands”，比如 ngx_command_t ngx_http_hello_nginx_commands[]。

模块上下文结构体命名为“ngx_模块的类型_模块的功能描述_module_ctx”，也就是该模块本身的名称加“_ctx”后缀，比如 ngx_http_module_t ngx_http_hello_nginx_module_ctx。

其他变量的命名遵循 C 语言程序开发相关的命名原则即可。

16.3 模块的编译与安装

完成“Hello_Nginx”模块源码的编写后，我们就可以将其加入到 Nginx 服务器中了。Nginx 服务器不支持模块的热启用，所有的模块都是在 Nginx 服务器程序编译时添加进来的。核心模块和标准模块是自动加入的，而可选模块和第三方模块需要在编译配置时手动指定。

添加第三方模块时，在重新编译 Nginx 之前还要对模块目录下的 config 文件中添加以下内容：

```
ngx_addon_name=ngx_hello_nginx_module
HTTP_AUX_FILTER_MODULES="$HTTP_MODULES ngx_hello_nginx_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_hello_nginx_module.c"
```

$ngx_addon_dir 将接收 configure 命令中--add-module 参数的值。

接下来就可以重新编译 Nginx 服务器源码了，唯一需要注意的是，在编译配置时，我们手动指定

加入我们自己编写的“Hello_Nginx”模块：

```
./configure --add-module=/home/ngx_http_hello_nginx_module ……
```

--add-module 参数的值为我们放置“Hello_Nginx”模块源码以及 config 文件的目录的完整路径。在编译过程中，编译器将读取 config 文件中的内容，找到“Hello_Nginx”模块源码的主文件，然后将其编译为.o 文件，再复制到当前编译目录中的 objs 目录中，编译完成后，“Hello_Nginx”模块就已经被插入到 Nginx 服务器中了。

注意

许多人习惯于在 Windows 平台上编写源码，然后将源码文件复制到 Linux 平台上。但是采用这种方式编写的 Nginx 第三方模块可能在编译时出现问题，一般只提示“needed by nginx”，没有其他有用信息。遇到这种情况，可能是由于从 Windows 平台上复制过来的 Nginx 第三方模块的源码文件编码格式不正确导致的。我们可以使用 Linux 平台上的 dos2unix 工具将所有源码文件和 config 文件进行一下转换，然后重新 configure，再进行 make 就没有问题了。

Nginx 服务器源码重新编译完成以后，我们就可以在配置文件中对“Hello_Nginx”模块进行配置了。比如，笔者基于第 2 章中的基础配置实例，对 myserver1（端口 8081）虚拟主机进行如下的配置：

```
server {
    # 配置监听端口和主机名称（基于名称）
   listen      8081;
    server_name  myServer1;
   # 配置请求处理日志存放路径
    access_log   /myweb/server1/log/access.log;
    # 配置错误页面
    error_page  404    /404.html;
    # 配置处理/server1/location1 请求的 location
    location /server1/location1 {
        root  /myweb;
        index  index.svr1-loc1.htm;
    }
    # 配置处理/server1/location2 请求的 location
    location /server1/location2 {
        root  /myweb;
        index  index.svr1-loc2.htm;
    }
   # 配置“Hello_Nginx”模块
   location / {
        hello_nginx  “ This is my world!”;
   }
}
```

进入新编译好的 Nginx 安装目录，使用以下命令使 Nginx 服务器应用新的配置文件：

```
./sbin/nginx -s reload
```

之后我们从客户端浏览器访问“myserver1:8081/helllo_nginx”时，就可以看到“Hello_Nginx”模块将“Hello, my new nginx! This is my world!”打印到浏览器界面上了。这样，我们的“Hello_Nginx”

模块就生效了。

16.4 本章小结

本章我们简单地学习了 Nginx 服务器第三方模块的编程，通过一个简单的模块编程实例，我们对 Nginx 服务器中各种模块的模块结构有了一个比较清晰的认识。一个模块使用 ngx_module_t 结构组织该模块的所有数据资源，使用 ngx_command_t 结构保存该模块在 Nginx 配置文件中可以使用的指令及指令的使用方法，使用模块上下文结构保存 Nginx 在对配置文件解析之前、解析过程中、解析完成后等不同时机应该触发的预定义操作。这三个结构体构成了 Nginx 服务器模块的基本结构，无论多么复杂的模块都是以它们为中心完成任务的。

第 17 章

Nginx 在动态网站建设中的应用实例

Nginx 服务器本身对流行的 JSP、PHP、Perl 等动态页面的支持并不完整，但是它可以通过反向代理完成对动态页面的请求，而其本身可以提供高效的负载均衡、代理缓存等功能。本章我们结合不同类型动态网站建设的实际环境，介绍 Nginx 服务器的配置方案，为大家的实际工作提供参考。

在本章中，我们可以学习到以下内容：

- Nginx 服务器在 JSP 网站建设中的应用。
- Nginx 服务器在 PHP 网站建设中的应用。
- Nginx 服务器在 Perl 网站建设中的应用。

17.1 Nginx 在 JSP 网站建设中的应用

在 JSP 网站的建设中，Nginx 服务器可以作为反向代理服务器，利用反向代理功能将客户端请求发送到 Tomcat、Apache 等后端服务器完成动态页面的请求处理。当然，在应用过程中，对于少量的静态网页，Nginx 服务器可以直接处理。在高并发访问的情形下，Nginx 服务器不仅可以完成反向代理的功能，更能够起到负载均衡的作用。

17.1.1 环境描述

1. Linux 平台。
2. Nginx 服务器用做代理服务器。
3. Tomcat 用做 JSP 程序运行服务器。

图 17.1 示意了 Nginx 服务器在 JSP 网站建设中的应用。

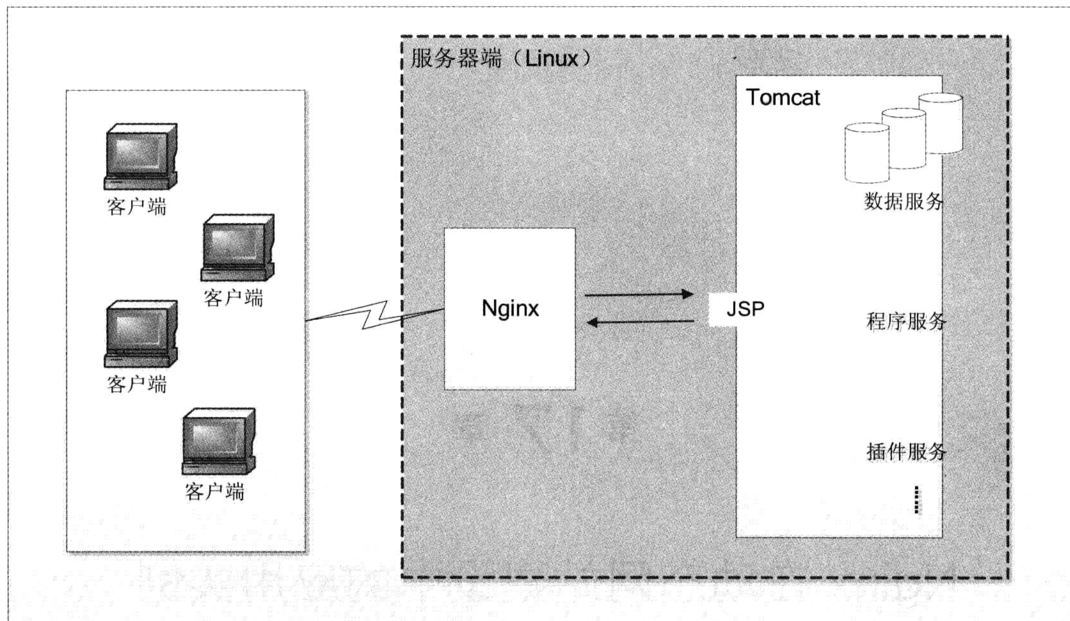

图 17.1　Nginx + JSP 网站结构示意图

17.1.2　特别模块说明

1. 支持正则表达式

为了确保能在 Nginx 服务器中使用正则表达式进行更灵活的配置，安装之前需要确定系统是否安装有 PCRE（PerlCompatibleRegularExpressions）包 pcre-devel。

在 configure 配置过程中如果因为没有找到 pcre 库报类似下面的错误：

```
./configure:error: the HTTP rewrite module requires the PCRE library. You can either
disable the module by using --without-http_rewrite_module option, or install the PCRE
library into the system, or build the PCRE library statically from the source with
nginx by using --with-pcre = <path> option.
```

则可以按照提示使用--with-pcre 参数手动指定 pcre 库的目录位置。

2. 支持实时显示 Nginx 运行状况

为了能够实时监控 Nginx 服务器的运行状况，可以在 Nginx 编译配置时将任务状态模块 http_stub_status_modul 编译进去，使用的参数为--with-http_stub_status_module。

17.1.3　配置方案

Nginx 配置文件的内容，其代码如下：

```
user  nobody  nobody;
worker_processes  2;
error_log  /usr/local/nginx/logs/nginx_error.log;
pid  /usr/local/nginx/nginx.pid;
events
```

```
{
    use  epoll;
    worker_connections  51200;
}
http
{
    include  mime.types;
    default_type  application/octet-stream;
    keepalive_timeout  60;
    #配置 gzip 功能
    gzip  on;
    gzip_min_length  1k;
    gzip_buffers  4  8k;
    gzip_http_version  1.1;
    gzip_types  text/plain  application/x-javascript  text/css  text/html  application/xml;
    server
    {
        listen  80;
        server_name  localhost;
        index  index.jsp  index.html  index.htm;
        root  /usr/local/www;
        #首页是 index.jsp 时，不能自动解析，需要使用 proxy_pass 指令
        location  /  {
            root  /usr/local/www;
            index  index.jsp;
            proxy_pass  http://jsphost:port;
            access_log  off;
            # 启用 Nginx 服务器的 Status 状态监控模块
            stub_status  on;
        }
        #用匹配扩展名的方式匹配静态文件
        location ~ *\.(htm|html|gif|jpg|jpeg|png|bmp|ico|css|js|zip|java|jar|txt| txt)$
        {
            root  /usr/local/www;
            #设置静态文件缓存的过期时间
            expires  24h;
        }
        #用匹配目录的方式匹配静态目录
        location ~^ /(images|common|download|css|js)/
        {
            root  /usr/local/www;
            #设置静态文件缓存的过期时间
            expires  30d;
        }
        #用匹配扩展名的方式匹配动态文件
        location  ~*  \.(jsp|do|action)$
```

```
        {
            root  /usr/local/www;
            index  index.jsp;
            #加载proxy.conf配置文件，用于测试JSP访问
            include  /usr/local/nginx/conf/proxy.conf;
            proxy_pass  http://jsphost:port;
            proxy_set_header  X-Real-IP  $remote_addr;
        }
    }
}
```

/usr/local/nginx/conf/proxy.conf 文件的内容如下：

```
#proxy.conf
proxy_redirect  off;
proxy_set_header  Host  $host;
proxy_set_header  X-Real-IP  $remote_addr;
proxy_set_header  X-Forwarded-For  $proxy_add_x_forwarded_for;
client_max_body_size  10m;
client_body_buffer_size  128k;
proxy_connect_timeout  90;
proxy_send_timeout  90;
proxy_read_timeout  90;
proxy_buffers  324k;
```

17.2 Nginx 在 PHP 网站建设中的应用

同样，在 PHP 网站的建设中，Nginx 服务器也可以作为反向代理服务器，利用反向代理功能将客户端请求发送到 Tomcat、Apache 等后端服务器完成动态页面的请求处理。对于少量的静态网页，Nginx 服务器可以实现快速缓存，在高并发访问的情形下，它可以起到负载均衡的作用。本节的例子中有一处与上一节不同，就是 PHP 使用 FastCGI 模式。

17.2.1 环境描述

1. Linux 平台。

2. Nginx 服务器用做代理服务器。

3. Tomcat 用做 PHP 程序运行服务器。

在整体架构上，PHP 网站使用 Nginx 作为前端服务器，这与 JSP 网站是基本相同的。图 17.2 示意了 Nginx 服务器在 PHP 网站建设中的应用。

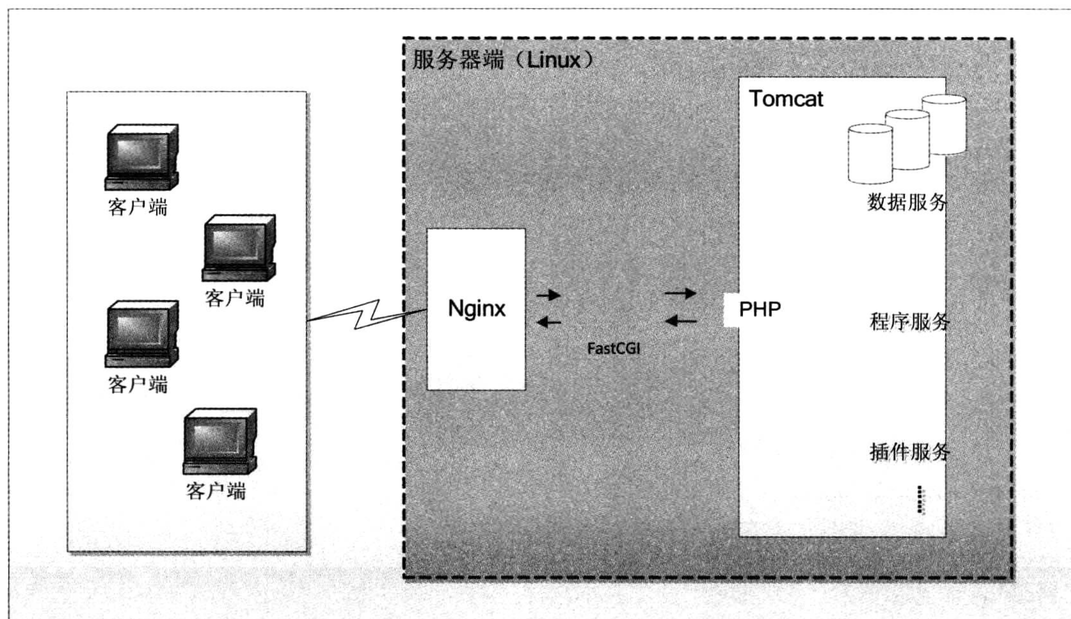

图 17.2　Nginx + PHP 网站结构示意图

17.2.2　特别模块说明

1. 支持 FastCGI 模式

对于 CGI（Common Gateway Interface，公共网关接口），可以理解其为 HTTP 服务器与其他主机上运行的程序进行通信的接口，其程序运行在 HTTP 服务器上。CGI 可以用任何一种语言编写，具有语言无关的特性。

CGI 主要用在下面的情况。较早期的 Web 服务器一般只处理静态的请求，如果碰到动态请求，服务器会根据这次请求的内容，生成一个新进程来运行外部的脚本程序获取数据，然后把处理完的数据返回给服务器，最后服务器把结果发送给用户，而刚才建立的进程也随之退出。如果下次用户还请求该动态脚本，那么 Web 服务器又再次生成一个新进程，重复上面的过程。这样的模式被称为 fork-and-execute 模式。

从上面的描述可以看出，当出现大量动态请求时，CGI 每次都必须生成新的工作进程，这样它的工作效率就会降低，而其对系统资源的占用也相当高。为了解决这一问题，就出现了 FastCGI 模式。

可以将 FastCGI 理解为一个常驻在系统中的 CGI，被激活以后，它不会每次都花费时间去生成新的进程。该模式还有一个优点，就是支持分布式的运算。FastCGI 模式目前支持的语言非常多，有 PHP、C/C++、Java、Perl、Tcl、Python、SmallTalk、Ruby 等。Apache、IIS、Lighttpd、Nginx 等流行的 Web 服务器上支持 FastCGI 模式。

Nginx 服务器的标准 HTTP 模块默认包含对 FastCGI 模式的支持。为了使用 FastCGI，在编译安装 PHP 时，要启用 FastCGI 模式。

17.2.3 配置方案

Nginx 配置文件的内容，其代码如下：

```
user  nobody  nobody;
worker_processes  2;
error_log  /usr/local/nginx/logs/nginx_error.log;
pid  /usr/local/nginx/nginx.pid;
events
{
   use  epoll;
   worker_connections  51200;
}
http
{
   include  mime.types;
   default_type  application/octet-stream;
   keepalive_timeout  60;
   #配置 gzip 功能
   gzip  on;
   gzip_min_length  1k;
   gzip_buffers  4  8k;
   gzip_http_version  1.1;
   gzip_types  text/plain  application/x-javascript  text/css  text/html
application/xml;
   #配置 FastCGI 功能
       #设置 Nginx 与 FastCGI 服务器建立连接的超时时间
   fastcgi_connect_timeout  200;
       #设置 Nginx 向 FastCGI 发送请求后等待的超时时间
   fastcgi_send_timeout  200;
       #设置 Nginx 向 FastCGI 发送请求后等待的超时时间
   fastcgi_read_timeout  200;
       #设置 Nginx 从 FastCGI 读取应答数据时等待的超时时间
   fastcgi_buffer_size  64k;
       #设置每次从 FastCGI 读取应答数据使用的缓存区大小
   fastcgi_buffers 4 64k;
       #设置每次将来自 FastCGI 的应答数据写入本地临时文件时数据块的大小上限
   fastcgi_temp_file_write_size  128k;
   server
   {
      listen  80;
      server_name  localhost;
      index  index.jsp  index.html  index.htm;
      root  /usr/local/www;
      location  ~  .*\.(php|php5)?$
      {
         #添加 FastCGI 的配置文件
```

```
            include  fcgi.conf;
            fastcgi_pass    fastcgi_IP:port;
            fastcgi_index   index.php;
        }
        #用匹配扩展名的方式匹配静态文件
        location ~ *\.(htm|html|gif|jpg|jpeg|png|bmp|ico|css|js|zip|java|jar|txt|txt)$
        {
            root  /usr/local/www;
            expires  24h;
        }
        #用匹配目录的方式匹配静态目录
        location ~^ /(images|common|download|css|js)/
        {
            root  /usr/local/www;
            expires  30d;
        }
    }
}
```

/usr/local/nginx/conf/fcgi.conf 文件的内容如下：

```
fastcgi_paramGATEWAY_INTERFACECGI/1.1;
fastcgi_paramSERVER_SOFTWAREnginx;
fastcgi_paramQUERY_STRING$query_string;
fastcgi_paramREQUEST_METHOD$request_method;
fastcgi_paramCONTENT_TYPE$content_type;
fastcgi_paramCONTENT_LENGTH$content_length;
fastcgi_paramSCRIPT_FILENAME$document_root$fastcgi_script_name;
fastcgi_paramSCRIPT_NAME$fastcgi_script_name;
fastcgi_paramREQUEST_URI$request_uri;
fastcgi_paramDOCUMENT_URI$document_uri;
fastcgi_paramDOCUMENT_ROOT$document_root;
fastcgi_paramSERVER_PROTOCOL$server_protocol;
fastcgi_paramREMOTE_ADDR$remote_addr;
fastcgi_paramREMOTE_PORT$remote_port;
fastcgi_paramSERVER_ADDR$server_addr;
fastcgi_paramSERVER_PORT$server_port;
fastcgi_paramSERVER_NAME$server_name;
#PHPonly,requiredifPHPwasbuiltwith--enable-force-cgi-redirect
#fastcgi_paramREDIRECT_STATUS200;
```

17.3　Nginx+Perl 脚本在网站建设中的应用

作为比较古老的一种脚本语言，曾经大量的 Web 应用都使用 Perl 语言编写，其灵活的操作和强大的功能给许多开发者留下了很好的印象，但也是因为它的灵活性和冗余的语法，导致许多程序的源码难以阅读和理解，给维护带来了巨大的困难，因此到目前为止，使用 Perl 语言编写 Web 应用的案例已

经不多了。不过，Perl 语言仍然在部分应用中发挥着巨大的作用，Nginx 服务器对 Perl 脚本的支持也是一大亮点，所以，我们还是有必要列举一个简单的配置实例来展示一下 Nginx 服务器对 Perl 语言的支持。在这个实例中，Perl 语言更像是被当作一种 CGI 来使用。

17.3.1　环境描述

1. Linux 平台。

2. Nginx 服务器用做 Web 服务器。

3. Perl 接口库（CGI）。

该类网站的结构实际上很简单，由于 Nginx 服务器可以通过指定的模块支持 Perl 脚本的解析，因此可以将 Perl 语言当作 CGI 来调用，从后端服务器中获取数据，最终解析出网页结果。Nginx 服务器在其中扮演的角色仍然可以被认为是反向代理服务器。如图 17.3 所示为该框架的示意图。

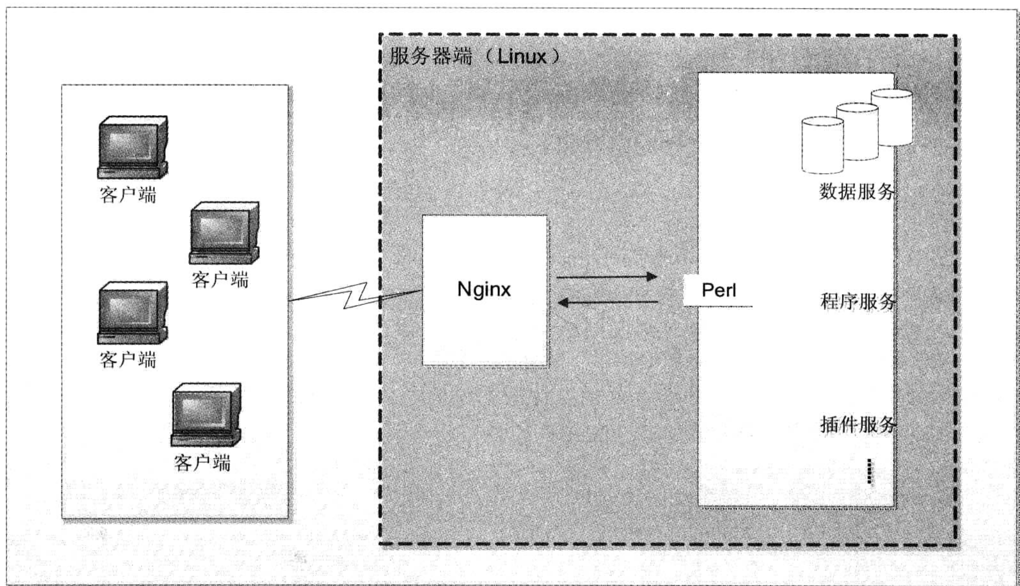

图 17.3　Nginx + Perl 网站结构示意图

17.3.2　特别模块说明

支持 Perl 模块

Nginx 服务器支持 Perl 脚本语言需要用到 Perl 模块，该模块是可选的 HTTP 模块，需要在编译配置 Nginx 服务器源码时指定启用，使用的参数为--with-http_perl_module。

17.3.3　配置方案

Nginx 配置文件的内容，其代码如下：

```
user  nobody  nobody;
worker_processes  2;
error_log  /usr/local/nginx/logs/nginx_error.log;
```

```
pid  /usr/local/nginx/nginx.pid;
events
{
   use  epoll;
   worker_connections  51200;
}
http{
   include  mime.types;
   default_type  application/octet-stream;
   keepalive_timeout  60;
   #配置 gzip 功能
   gzip  on;
   gzip_min_length  1k;
   gzip_buffers  4  8k;
   gzip_http_version  1.0;
   gzip_comp_level  2;
   gzip_types  text/plain  text/plain  application/x-javascript  text/css  application/ xml;
   gzip_vary  on;
   #配置对 Perl（CGI）的支持
   perl_modules   /perlcgi;        # Perl(CGI)的目录
   perl_require  perlcgi.pm;       # Perl(CGI)的入口
   server{
      listen80;
      server_name  myweb;
      #此处配置通过 Perl(CGI)访问
      set  $APPLICATION_PATH  '/perl_app/';     #web 服务的路径
      set  $CONTEXT_PATH  '/perl';   #客户端的访问 URL，如：http://myweb/perl
      location  /perl  {
         perlAction::execute;
      }
   }
}
```

在配置文件中，我们配置了 Perl(CGI)的相关文件和路径，它的目录结构是这样的：

```
DIR/perlcgi/Perlcgi.pm
DIR/perl_app/lib
DIR/perl_app/mywebdir
```

其中，*DIR* 指任意 Nginx 服务器有权限访问的目录路径，*mywebdir* 指我们的网站存放的路径。

Perlcgi.pm 文件的内容如下：

```
package Perlcgi;
sub doExecute {
   return -1;
}
# 请求控制器
sub execute {
   my $r = shift;
```

```
    my $content_path     = $r->variable('CONTEXT_PATH');
    my $application_path = $r->variable('APPLICATION_PATH');
    #加载库
    unshift @INC, $application_path;
    unshift @INC, "$application_path/lib";
    # 判断 $CONTEXT_PATH 与 uri 是否对应
    my $context_path_pos = index( $r->uri, $content_path );
    if ( $context_path_pos != 0 && $context_path_pos != 1 ) {
        http_error( $r, 500,
            'CONTEXT_PATH: "' . $content_path . '" not in ' . $r->uri );
        return OK;
    }
    # 替换uri "context_path/xxx/xxx"成"/xxx/xxx"
    my $uri = substr( $r->uri, $context_path_pos + length($content_path) );
    # 替换uri "/xxx/xxx"成"xxx/xxx"
    if ( index( $uri, "/" ) == 0 ) {
        $uri = substr( $uri, 1 );
    }
    eval {
        equire("$application_path/$uri.pm");
        eval( $uri . '::doExecute($r)' );
    };
    if ($@) {
        http_error( $r, 500, $@ );
    }
    $r->rflush;
    return OK;
}
sub http_error {
    my ( $r, $status, $message ) = @_;
    $r->status($status);
    $r->send_http_header("text/html");
    $r->print( '<H2>Nginx perl module error:</H2>' . '<HR/>' . $message );
}
1;
```

DIR/perl_app/lib 目录中有两个文件：分别是 Log.pm 和 Request.pm，它们的内容分别是：

```
package Log;
sub info {
    my ($msg) = @_;
    open LOG, '>> /log_dir/log_file ';
    print LOG $msg;
    close LOG;
}
1;
package Request;
sub params {
```

```
    ($r) = @_;
   my %param = {};
   map { $param{$1} = $2 if /(.*?)=(.*)/; } split /\&/, $r;
   return %param;
}
1;
```

通过以上设置，我们就可以在 DIR/perl_app/mywebdir 目录下编写自己的 Perl 动态网页了。注意，在动态网页中使用 Perl 脚本时使用 use　Request 将刚才 lib 目录下用于处理请求的库函数应用进来。

17.4　本章小结

本章重点介绍了 Nginx 服务器在动态网站建设过程中的应用，分别列举了在 JSP、PHP 动态网站中，Nginx 服务器作为代理服务器的配置方法。JSP+Nginx 的配置方案是 Nginx 服务器在动态网站方面应用的一般配置方案，其着眼于对动态数据的代理请求处理和对静态数据的缓存处理；PHP+Nginx 的配置方案着眼于 Nginx 服务器对 FastCGI 的支持。本章还列举了在 Perl 动态网站中 Nginx 服务器的应用配置，那里的 Nginx 服务器实际上也是用作代理服务器，通过 Perl 脚本请求动态数据，Perl 脚本则相当于 CGI。

第 18 章

Nginx 经典应用——LNAMP

在本章中，我们主要介绍一种经典的 Web 服务器架构——LNAMP。该架构可以应用于大多数的动态网站的建设和 Web 应用发布站点的建设，其支持 Web 服务应该具备的大多数功能，并且运行稳定，性能也能够满足一般站点的需求。

我们将在本章中学习到以下内容：

- LNAMP 架构的基本组成。
- 手工部署 LNAMP。
- 自动部署 LNAMP 的方法。

18.1 LNAMP 概述

LNAMP 是指由 Linux 平台、Nginx 服务器、Apache 服务器、MySQL 数据库、PHP 为主要组成部分的 Web 应用架构，其中 Linux 平台是该架构的实际执行环境，Nginx 服务器实现反向代理和负载均衡功能，Apache 服务器作为 Web 服务器支持 Web 程序运行，MySQL 数据库实现了对站点数据的存储和管理，PHP 作为服务器脚本语言实现数据的获取和组织。LNAMP 的主要特点是将 Nginx 服务器和 Apache 服务器组合使用。

在第 1 章中我们介绍过 Apache 服务器，它是重量级的 Web 服务器，功能强大而且性能稳定，是 Web 服务器用户的首选。但 Apache 服务器有一种严重的缺陷就是它每建立一个网络连接就会创建一个进程，在较高并发访问的情形下，这对服务器硬件资源的要求实在太高。而在资源有限的情况下，Apache 服务器又不能达到最佳性能或最大资源利用率。

Nginx 作为轻量级 Web 服务器，同时具备完备的功能和稳定的性能，尤其在处理静态 Web 资源的

能力上更是十分出色。同时 Nginx 也可以作为负载均衡服务器、反向代理服务器，其消耗资源低，转发性能高。然而，Nginx 服务器不能支持 PHP、JSP 等动态脚本语言，因此不能独立支撑动态站点的建设。

我们看到，Nginx 服务器和 Apache 服务器各有优缺点，如果将它们组合在一起发挥各自的优势之处，就可以形成功能完整、性能稳定的 Web 应用运行环境。由于 Nginx 在处理静态内容上占有巨大优势，因而其适合于放在前端处理静态资源，同时 Nginx 服务器可以对高并发访问情况下的大量客户端请求提供均衡服务，从而保证整个系统的稳定性；Apache 服务器作为后端服务器，只负责处理动态内容就可以了，这样也就降低了 Apache 服务器对硬件资源的压力。

基于以上的原因，就出现了 Nginx+Apache 这样的 Web 应用架构，再将 MySQL 数据库整合到其中，使用 PHP 建站，全部部署在 Linux 平台上发布，这样就形成了一套完整的 LNAMP。

18.2　手动部署和配置

手动部署一套 LNAMP 是一个比较漫长的过程，因为这其中涉及 Nginx 服务器、Apache 服务器、MySQL 服务器的各自配置和关联配置，同时它们要支持 PHP，并且要尽可能体现对静态资源和动态资源的优化处理。在配置过程中，大家还要熟悉一些 Linux 平台上的基本操作。我们下面分步讲解 LNAMP 的部署。

18.2.1　环境准备

我们首先准备一套 Linux 平台，推荐 CentOS 或者 RedHat。为了避免部署过程中出现不必要的错误，建议大家将系统中原有的与 MySQL、PHP 和 Apache 相关的组件全部删除。由于 Apache 服务器可能使用的实际安装包为 httpd，如果是这样，请删除 httpd 相关的组件。另外，笔者在这里使用的各主要安装包的版本是 CentOS 6、PHP 5.2.14、Nginx 1.2.3、Apache 2.2（httpd-2.2.17）和 MySQL 5.1.62。

1. 检查和安装必要组件

请确保系统中包含以下组件：

gcc　gcc-c++　gcc-g77　flex　bison　tar

libtool　libtool-libs　kernel-devel　autoconf

libjpeg　libjpeg-devel　libpng　libpng-devel

libtiff　libtiff-devel　gettext　gettext-devel

freetype　freetype-devel　libxml2　libxml2-devel　zlib　zlib-devel

file　glib2　glib2-devel　bzip2 diff*　openldap-devel

bzip2-devel　ncurses　ncurses-devel　curl　curl-devel　e2fsprogs

e2fsprogs-devel　krb5　krb5-devel　libidn　libidn-devel

openssl　openssl-devel　vim-minimal　unzip

如果系统中缺少相关的组件，在联网的状态下可以使用下列命令安装相关组件：

```
#yum -y install name
```

其中，*name* 就是待安装组件的名称。

注意

在使用 yum 命令时，为了能够获取最快的网络下载速度，可以先运行以下命令对下载源进行更新：

```
#yum -y install yum-fastestmirror
```

2. 检查和安装 PHP 5.2.x 所需的支持库

请确保安装了以下 PHP 所需的支持库：

libiconv-1.13.1

libevent-1.4.14b

libmcrypt-2.5.8

mhash-0.9.9.9

mcrypt-2.6.8

如果系统中缺少相关的库，在联网的状态下可以使用以下命令下载源码包：

```
#wget  -c  URL
```

其中，*URL* 表示源码地址。

当然我们也可以直接从浏览器查找和下载相关的源码包。关于这些库文件的编译和安装和一般 Linux 程序的源码编译安装基本一样，只有 libmcrypt-2.5.8 需要大家注意，它的安装过程如下：

```
#tar  zxvf  libmcrypt-2.5.8.tar.gz
#cd libmcrypt-2.5.8/
#./configure
#make
#make install
#/sbin/ldconfig
#cd libltdl/
#./configure --enable-ltdl-install
#make
#make install
```

18.2.2　安装和配置 MySQL

MySQL 5.1.62 源码包的下载地址为 http://mysql.proserve.nl/Downloads/mysql-5.1/mysql-5.1.62.tar.gz。下载解压后可以进行安装。有关 MySQL 数据库的安装和配置超出本书的讨论范围，因此不在这里详述，请大家参阅 MySQL 的官方文档。安装和配置完成后使用以下命令启用数据库服务：

```
#service mysql restart
```

这里有一点需要说明的是，Google 开发了一套开源的 TCMalloc 库可以提高 MySQL 在高并发访问情况下的性能，大家可以在安装 MySQL 之前先编译和安装该库。该库最新版为 2.0，源码下载地址为 http://code.google.com/gperftools/downloads/detail?name=gperftools-2.0.tar.gz。具体的安装说明请大家参阅压缩包内的官方文档。

安装完成 TCMalloc 库后，需要进行一些配置：