TIPI

TIPI:深入理解PHP内核

www.php-internal.com RELEASE_2011-04-01_V0.5.2

获取新版本

reeze http://reeze.cn
er http://www.zhanger.com
phppan http://www.phppan.com

第一章 准备工作和背景知识

我们在深入理解PHP的实现之前，需要做一些准备以及了解一些背景知识。本章首先说明PHP源码的下载，PHP的源码的简单编译，从而得到我们的调试环境。 接下来，我们将简单描述整个PHP源码的结构以及在*nix环境和Windows环境下如何阅读源码。 最后我们介绍在阅读PHP源码过程中经常会遇到的一些语句。

如果你没有接触过PHP，或者对PHP的历史不太了解，我们推荐你先移步百度百科 PHP， 这里有PHP非常详细的历史介绍，它包括PHP的诞生，PHP的发展，PHP的应用，PHP现有三大版本的介绍以及对于PHP6的展望等。

下面，我们将介绍源码阅读环境的搭建。

第一节 环境搭建

在开始学习PHP实现之前, 我们首先需要一个实验和学习的环境. 下面介绍一下怎样在*nix环境下准备和搭建PHP环境. (*nix指的是类Unix环境,比如各种Linux发行版,FreeBSD, OpenSolaris, Mac OS X等操作系统)

1.获取PHP源码

为了学习PHP的实现，我们需要下载PHP的源代码。下载源码首选是去PHP官方网站 http://php.net/downloads.php下载, 如果你喜欢使用类似svn/git这些版本控制软件,还可以使用svn/git来获取最新的源代码。

```
#svn
cd ~
svn co http://svn.php.net/repository/php/php-src/branches/PHP_5_2 php-src-5.2
#php5.2版本
svn co http://svn.php.net/repository/php/php-src/branches/PHP_5_3 php-src-5.3
#php5.3版本

#git
cd ~
git clone git://github.com/php/php-src.git php-src
```

笔者比较喜欢用版本控制软件签出代码,这样做的好处是能看到PHP每次修改的内容及日志信息, 如果自己修改了其中的某些内容也能快速的查看到.(当然你还可以试着建立属于自己的源代码分支)

2.准备编译环境

在*nix环境下，首先需要gcc编译构建环境. 如果你是用的是Ubuntu或者是用apt做为包管理的系统,可以通过如下命令快速安装.

```
sudo apt-get install build-essential
```

如果你使用的是Mac OS，则需要安装Xcode。Xcode可以在Mac OS X的安装盘中找到，如果你有Apple ID的话，也可以登陆苹果开发者网站http://developer.apple.com/下载。

3. 编译

下一步就可以开始编译了，本文只简单介绍基本的编译过程，不包含apache的PHP支持以及mysql等模块的编译。相关资料请百度或google之。 假设源代码下载到了~/php-src的目录中，执行buildconf命令以生成所需要的Makefile文件

```
cd ~/php-src
./buildconf
```

执行完以后就可以开始configure了, configure有很多的参数, 比如指定安装目录, 是否开启相关模块等选项

```
./configure --help # 查看可用参数
```

为了尽快得到可以测试的环境,我们就不加其他参数了.直接执行./configure就可以了. 以后如果需要其他功能可以重新编译. 如果configure命令出现错误,可能是缺少PHP所依赖的库,各个系统的环境可能不一样.出现错误可根据出错信息上网搜索. 直到完成configure. configure完成后我们就可以开始编译了.

```
make
```

在*nix下编译过程序的读者应该都熟悉经典的configure make, make install吧. 执行make之后是否需要make install就取决于你了. 如果install的话 最好在configure的时候是用prefix参数指定安装目录, 不建议安装到系统目录, 避免和系统原有的PHP版本冲突.

在make 完以后，~/php-src目录里就已经有了php的可以执行文件. 执行一下命令:

```
cd ~/php-src
./sapi/cli/php -v
```

如果看到输出php版本信息则说明编译成功. 如果是make install的话则执行 $prefix/bin/php 这个路径的php, 当然如果是安装在系统目录或者你的prefix 目录在$PATH环境变量里的话,直接执行php就行了.

> 在只进行make而不make install时，只是编译为可执行二进制文件，而不会进行任何安装动作(你懂的)，所以在终端下执行的php-cli所在路径就是php-src/sapi/cli/php。

后续的学习中可能会需要重复configure make 或者 make && make install 这几个步骤。

第二节 源码结构、阅读代码方法

PHP源码目录结构

俗话讲：大巧不工。PHP的源码在结构上非常清晰甚至简单。下面先简单介绍一下PHP源码的目录结构。

- **根目录:/** 这个目录包含的东西比较多, 主要包含一些说明文件以及设计方案. 其实项目中的这些README文件是非常值得阅读的例如:
  - /README.PHP4-TO-PHP5-THIN-CHANGES 这个文件就详细列举了PHP4和PHP5的一些差异.
  - 还有有一个比较重要的文件/CODING_STANDARDS, 如果要想写PHP扩展的话,这个文件一定要阅读一下, 不管你个人的代码风格是什么样, 怎么样使用缩进和花括号, 既然来到了这样一个团体里就应该去适应这样的规范, 这样在阅读代码或者别人阅读你的 代码是都会更轻松.
- **build** 顾名思义, 这里主要放置一些和源码编译相关的一些文件，比如开始构建之前的buildconf脚本等文件,还有一些检查环境的脚本等.
- **ext** 官方扩展目录，包括了绝大多数PHP的函数的定义和实现，如array系列，pdo系列，spl系列等函数的实现，都在这个目录中。个人写的扩展在测试时也可以放到这个目录，方便测试和调试。
- **main** 这里存放的就是PHP最为核心的文件了,主要实现PHP的基本设施, 这里和Zend引擎不一样,Zend引擎主要实现语言最核心的语言运行环境.
- **Zend** Zend引擎的实现目录,比如脚本的词法语法解析, opcode的执行以及扩展机制的实现等等.
- **pear** “PHP 扩展与应用仓库”, 包含PEAR的核心文件.
- **sapi** 包含了各种服务器抽象层的代码，例如apache的mod_php,cgi, fastcgi以及fpm等等接口.
- **TSRM** PHP的线程安全是构建在TSRM库之上的, PHP实现中常见的*G宏通常是对TSRM的封装, TSRM(Thread Safe Resource Manager)线程安全资源管理器.
- **tests** PHP的测试比较有意思,它使用PHP来测试PHP, 测试php脚本在/run-tests.php, 这个脚本读取tests目录中phpt文件.

读者可以打开这些看看, php定义了一套简单的规则来测试, 例如一下的这个测试脚本/tests/basic/001.phpt:

```
--TEST--
Trivial "Hello World" test
--FILE--
<?php echo "Hello World"?>
--EXPECT--
Hello World
```

这段测试脚本很容易看懂, 执行--FILE--下面的PHP文件,如果最终的输出是--EXPECT--所期望的结果则表示这个测试通过, 可能会有读者会想,如果测试的脚本 不小心触发Fatal Error,或者抛出未被捕获的异常了,因为如果在同一个进程中执行,测试就会停止,后面的测试也将无法执行, php中有很多将脚本隔离的方法比如: system(), exec()等函数, 这样可以使用主测试进程服务调度被测脚本和检测测试结果, 通过这些外部调用执行测试. php测试使用了proc_open()函数, 这样就可以保证测试脚本和被测试脚本之间能隔离开.

- **win32** 这个目录主要包括Windows平台相关的一些实现, 比如sokcet的实现在Windows下和*Nix平台就不太一样, 同时也包括了Windows下编译PHP相关的脚本。

PHP源码阅读方法

使用VIM + Ctags查看源码

通常在Linux或其他*Nix环境我们都使用VIM作为代码编辑工具, 在纯命令终端下，它几乎是无可替代的, 它具有非常强大的扩展机制,基本是无所不能了. 不过Emacs用户请不要激动, 笔者还没有真正使用Emacs, 虽然我知道它甚至可以煮咖啡, 还是等笔者有时间了或许会试试煮杯咖啡边喝边写.

在Linux下编写过代码的读者应该或多或少都试过ctags吧. ctags支持非常多的语言,可以将源代码中的各种符号,比如:函数、宏类等信息抽取出来做上标记。保存到一个文件中. 它保存的文件格式其实也挺简单,比如符合UNIX的哲学, 使用起来也很简单:

```
#在PHP源码目录(假定为/server/php-src)执行:
$ cd /server/php-src
$ ctags -R

#小技巧：在当前目录生成的tags文件中使用的是相对路径，
#若改用 ctags -R /full-path-to-php-src/ ，可以生成包含完整路径的ctags，就可以随意放到任意文件夹中了。

#在~/.vimrc中添加:
set tags+=/server/php-src/tags
```

上面代码会在/sever/php-src目录下生成一个名为tags的文件, 这个文件的格式很简单:

```
{tagname}<Tab>{tagfile}<Tab>{tagaddress}

EG  Zend/zend_globals_macros.h  /^# define EG(/;"   d
```

它的每行是上面的这样一个格式, 第一个就是符号名, 例如上例的EG宏, 这个符号的文件位置以及这个符号所在的位置. VIM可以读取tags文件,当我们在符号上**CTRL+]**时VIM将尝试从tags文件中寻找这个符号, 如果找到了则根据该符号所在的文件已经该符号的位置打开该文件, 并将光标定位到符号定义所在的位置. 这样我们就能快速的寻找到符号的定义了.

使用 **Ctrl+]** 就可以自动跳转至定义，**Ctrl+t** 可以返回上一次查看位置。这样就可以快速的在代码之间游动了.

这种浏览方式用习惯了还是很方便的. 不过如果你不习惯使用VIM这类编辑器,可以看看下面介绍的IDE.

> 如果你使用的Mac OS X, 运行ctags程序可能会出错, 因为Mac OS X自带的ctags程序有些问题, 所以需要自己下载安装ctags, 笔者推荐使用homebrew来安装.

使用IDE查看代码

如果不习惯使用VIM来看代码,那在可以使用一些功能较丰富IDE，比如Windows下可以使用Visual Studio 2010. 或者使用跨平台的Netbeans, 或者Eclipse来看代码,这些工具都相对较重量级一些,不过这些工具不管是调试还是查看代码都相对较方便,

在Eclipse及Netbeans下查看符号定义的方式通常是将鼠标移到符号上,同时按住**CTRL**,然后单击,将会跳转到符号定义的位置.

而如果使用VS的话, 在win32目录下已经存在了可以直接打开的工程文件，如果由于版本原因无法打开，可以在此源码目录上新建一个基于现有文件的Win32 Console Application工程。 常用快捷键：

```
F12 转到定义
CTRL + F12转到声明

F3: 查找下一个
Shift+F3: 查找上一个

Ctrl+G: 转到指定行

CTRL + -向后定位
CTRL + SHIFT + -向前定位
```

对于一些搜索类的操作，可以考虑使用editplus或其它文本编辑工具进行，这样的搜索速度相对来说会快一些。 如果使用editplus进行搜索，一般是选择 【搜索】 中的 【在文件中查找...】

第三节 常用代码

在PHP的源码中经常会看到一些宏或一些对于刚开始看源码的童鞋比较纠结的代码。这里提取中间的一些进行说明。

1. 关于"##"和"#"

在PHP的宏定义中，最常见的要数双井号。

双井号(##)

在C语言的宏中，"##"被称为 **连接符**（concatenator），用来把两个语言符号(Token)组合成单个语言符号。 这里的语言符号不一定是宏的变量。并且双井号不能作为第一个或最后一个元素存在。如下所示源码:

```
#define PHP_FUNCTION            ZEND_FUNCTION
#define ZEND_FUNCTION(name)             ZEND_NAMED_FUNCTION(ZEND_FN(name))
#define ZEND_FN(name) zif_##name
#define ZEND_NAMED_FUNCTION(name)       void name(INTERNAL_FUNCTION_PARAMETERS)
#define INTERNAL_FUNCTION_PARAMETERS int ht, zval *return_value, zval
**return_value_ptr, \
zval *this_ptr, int return_value_used TSRMLS_DC

PHP_FUNCTION(count);

//  预处理器处理以后, PHP_FUCNTION(count);就展开为如下代码
void zif_count(int ht, zval *return_value, zval **return_value_ptr,
        zval *this_ptr, int return_value_used TSRMLS_DC)
```

宏ZEND_FN(name)中有一个"##"，它的作用一如之前所说，是一个连接符，将zif和宏的变量name得值连接起来。

单井号(#)

"#"的功能是将其后面的宏参数进行 **字符串化操作** ，简单说就是在对它所引用的宏变量通过替换后在其左右各加上一个双引号， 用比较官方的话说就是将语言符号(Token)转化为字符串。 例如:

```
#define STR(x) #x

int main(int argc char** argv)
{
    printf("%s\n", STR(It's a long string)); // 输出 It's a long str
    return 0;
}
```

如前文所说，It's a long string 是宏STR的参数，在展开后被包裹成一个字符串了。所以printf函数能直接输出这个字符串， 当然这个使用场景并不是很适合，因为这种用法并没有实际的意义，实际中在宏中可能会包裹其他的逻辑，比如对字符串进行封装等等。

2. 关于宏定义中的do-while

如下所示为PHP5.3新增加的垃圾收集机制中的一段代码:

```
#define ALLOC_ZVAL(z)                                       \
do {                                                        \
    (z) = (zval*)emalloc(sizeof(zval_gc_info));             \
    GC_ZVAL_INIT(z);                                        \
} while (0)
```

如上所示的代码，在宏定义中使用了 **do{ }while(0)** 语句格式。如果我们搜索整个PHP的源码目录，会发现这样的语句还有很多。 那为什么在宏定义时需要使用do-while语句呢? 我们知道do-while循环语句是先执行再判断条件是否成立, 所以说至少会执行一次。 当使用do{ }while(0)时代码肯定只执行一次, 肯定只执行一次的代码为什么要放在do-while语句里呢? 这种方式适用于宏定义中存在多语句的情况。 如下所示代码:

```
#define TEST(a, b)  a++;b++;

if (expr)
    TEST(a, b);
else
    do_else();
```

代码进行预处理后，会变成:

```
if (expr)
    a++;b++;
else
    do_else();
```

这样if-else的结构就被破坏了: if后面有两个语句, 这样是无法编译通过的, 那为什么非要do-while而不是简单的用{}括起来呢. 这样也能保证if后面只有一个语句。例如上面的例子,在调用宏TEST的时候后面加了一个分号, 虽然这个分号可有可无, 但是出于习惯我们一般都会写上. 那如果是把宏里的代码用{}括起来,加上最后的那个分号. 还是不能通过编译. 所以一般的多表达式宏定义中都采用do-while(0)的方式.

3. #line 预处理

```
#line 838 "Zend/zend_language_scanner.c"
```

#line预处理用于改变当前的行号和文件名。 如上所示代码，将当前的行号改变为838,文件名Zend/zend_language_scanner.c 它的作用体现在编译器的编写中，我们知道编译器对C 源码编译过程中会产生一些中间文件，通过这条指令，可以保证文件名是固定的，不会被这些中间文件代替，有利于进行调试分析。

4.PHP中的全局变量宏

在PHP代码中经常能看到一些类似PG(), EG()之类的函数, 他们都是PHP中的一些宏，这些宏主要的作用是解决线程安全所写的全局变量包裹宏， 例如$PHP_SRC/main/php_globals.h文件中就包含了很多这类的宏, 例如PG这个PHP的核心全局变量的宏。 如下所示代码为其定义。

```
#ifdef ZTS   // 编译时开启了线程安全则使用线程安全库
# define PG(v) TSRMG(core_globals_id, php_core_globals *, v)
extern PHPAPI int core_globals_id;
#else
# define PG(v) (core_globals.v) // 否则这其实就是一个普通的全局变量
extern ZEND_API struct _php_core_globals core_globals;
#endif
```

如上，ZTS是线程安全的标记，这个在以后的章节会详细介绍，这里就不再说明。下面简单说说，PHP运行时的一些全局参数， 这个全局变量为如下的一个结构体，各字段的意义如字段后的注释:

```
struct _php_core_globals {
        zend_bool magic_quotes_gpc; //  是否对输入的GET/POST/Cookie数据使用自动字符串转义。
        zend_bool magic_quotes_runtime; //是否对运行时从外部资源产生的数据使用自动字符串转义
        zend_bool magic_quotes_sybase;  //   是否采用Sybase形式的自动字符串转义

        zend_bool safe_mode;    //  是否启用安全模式

        zend_bool allow_call_time_pass_reference;   //是否强迫在函数调用时按引用传递参数
        zend_bool implicit_flush;   //是否要求PHP输出层在每个输出块之后自动刷新数据

        long output_buffering;  //输出缓冲区大小(字节)

        char *safe_mode_include_dir;    //在安全模式下，该组目录和其子目录下的文件被包含时，将跳过UID/GID检查。
        zend_bool safe_mode_gid;    //在安全模式下，默认在访问文件时会做UID比较检查
        zend_bool sql_safe_mode;
        zend_bool enable_dl;    //是否允许使用dl()函数。dl()函数仅在将PHP作为apache模块安装时才有效。

        char *output_handler;   // 将所有脚本的输出重定向到一个输出处理函数。

        char *unserialize_callback_func;    // 如果解序列化处理器需要实例化一个未定义的类，这里指定的回调函数将以该未定义类的名字作为参数被unserialize()调用，
        long serialize_precision;   //将浮点型和双精度型数据序列化存储时的精度(有效位
```

```
数)。

        char *safe_mode_exec_dir;    //在安全模式下，只有该目录下的可执行程序才允许被
执行系统程序的函数执行。

        long memory_limit;  //一个脚本所能够申请到的最大内存字节数(可以使用K和M作为单
位)。
        long max_input_time;    // 每个脚本解析输入数据(POST, GET, upload)的最大允
许时间(秒)。

        zend_bool track_errors; //是否在变量$php_errormsg中保存最近一个错误或警告消
息。
        zend_bool display_errors;    //是否将错误信息作为输出的一部分显示。
        zend_bool display_startup_errors;    //是否显示PHP启动时的错误。
        zend_bool log_errors;    // 是否在日志文件里记录错误，具体在哪里记录取决于
error_log指令
        long      log_errors_max_len;    //设置错误日志中附加的与错误信息相关联的错误
源的最大长度。
        zend_bool ignore_repeated_errors;    //    记录错误日志时是否忽略重复的错误信
息。
        zend_bool ignore_repeated_source;    //是否在忽略重复的错误信息时忽略重复的错
误源。
        zend_bool report_memleaks;  //是否报告内存泄漏。
        char *error_log;    //将错误日志记录到哪个文件中。

        char *doc_root; //PHP的"根目录"。
        char *user_dir; //告诉php在使用 /~username 打开脚本时到哪个目录下去找
        char *include_path; //指定一组目录用于require(), include(),
fopen_with_path()函数寻找文件。
        char *open_basedir; // 将PHP允许操作的所有文件(包括文件自身)都限制在此组目录
列表下。
        char *extension_dir;    //存放扩展库(模块)的目录，也就是PHP用来寻找动态扩展模
块的目录。

        char *upload_tmp_dir;    // 文件上传时存放文件的临时目录
        long upload_max_filesize;    // 允许上传的文件的最大尺寸。

        char *error_append_string;  // 用于错误信息后输出的字符串
        char *error_prepend_string; //用于错误信息前输出的字符串

        char *auto_prepend_file;    //指定在主文件之前自动解析的文件名。
        char *auto_append_file; //指定在主文件之后自动解析的文件名。

        arg_separators arg_separator;    //PHP所产生的URL中用来分隔参数的分隔符。

        char *variables_order;  // PHP注册 Environment, GET, POST, Cookie,
Server 变量的顺序。

        HashTable rfc1867_protected_variables;  //  RFC1867保护的变量名，在
main/rfc1867.c文件中有用到此变量

        short connection_status;    //  连接状态，有三个状态，正常，中断，超时
        short ignore_user_abort;    //  是否即使在用户中止请求后也坚持完成整个请求。

        unsigned char header_is_being_sent; //  是否头信息正在发送

        zend_llist tick_functions;  //  仅在main目录下的php_ticks.c文件中有用到，
此处定义的函数在register_tick_function等函数中有用到。

        zval *http_globals[6];  // 存放GET、POST、SERVER等信息

        zend_bool expose_php;   //  是否展示php的信息

        zend_bool register_globals; //  是否将 E, G, P, C, S 变量注册为全局变量。
```

```
        zend_bool register_long_arrays; //    是否启用旧式的长式数组(HTTP_*_VARS)。
        zend_bool register_argc_argv;   //   是否声明$argv和$argc全局变量(包含用GET
方法的信息)。
        zend_bool auto_globals_jit; //   是否仅在使用到$_SERVER和$_ENV变量时才创建
(而不是在脚本一启动时就自动创建)。

        zend_bool y2k_compliance;    //是否强制打开2000年适应(可能在非Y2K适应的浏览器
中导致问题)。

        char *docref_root;  // 如果打开了html_errors指令，PHP将会在出错信息上显示超
连接，
        char *docref_ext;    //指定文件的扩展名(必须含有'.')。

        zend_bool html_errors;  //否在出错信息中使用HTML标记。
        zend_bool xmlrpc_errors;

        long xmlrpc_error_number;

        zend_bool activated_auto_globals[8];

        zend_bool modules_activated;     //   是否已经激活模块
        zend_bool file_uploads; //是否允许HTTP文件上传。
        zend_bool during_request_startup;    //是否在请求初始化过程中
        zend_bool allow_url_fopen;  //是否允许打开远程文件
        zend_bool always_populate_raw_post_data;     //是否总是生成
$HTTP_RAW_POST_DATA变量(原始POST数据)。
        zend_bool report_zend_debug;     //   是否打开zend debug，仅在main/main.c文
件中有使用。

        int last_error_type;     //   最后的错误类型
        char *last_error_message;    //   最后的错误信息
        char *last_error_file;  //   最后的错误文件
        int  last_error_lineno; //   最后的错误行

        char *disable_functions;     //该指令接受一个用逗号分隔的函数名列表，以禁用特
定的函数。
        char *disable_classes;  //该指令接受一个用逗号分隔的类名列表，以禁用特定的
类。
        zend_bool allow_url_include;     //是否允许include/require远程文件。
        zend_bool exit_on_timeout;  //   超时则退出
#ifdef PHP_WIN32
        zend_bool com_initialized;
#endif
        long max_input_nesting_level;    //最大的嵌套层数
        zend_bool in_user_include;  //是否在用户包含空间

        char *user_ini_filename;     //   用户的ini文件名
        long user_ini_cache_ttl;     //   ini缓存过期限制

        char *request_order;     //   优先级比variables_order高，在request变量生成时
用到，个人觉得是历史遗留问题

        zend_bool mail_x_header;     //   仅在ext/standard/mail.c文件中使用，
        char *mail_log;

        zend_bool in_error_log;
};
```

上面的字段很大一部分是与php.ini文件中的配置项对应的。 在PHP启动并读取php.ini文件时就会对这些字段进行赋值， 而用户空间的ini_get()及ini_set()函数操作的一些配置也是访问的这个全局变量。

在PHP代码的其他地方也存在很多类似的宏，这些宏和PG宏一样，都是为了将线程安全进行封装，同

时通过约定的 **G** 命名来表明这是全局的， 一般都是个缩写，因为这些全局变量在代码的各处都会使用到，这也算是减少了键盘输入。我们都应该尽可能的懒不是么?

如果你阅读过一些PHP扩展话应该也见过类似的宏，这也算是一种代码规范，在编写扩展时全局变量最好也使用这种方式命名和包裹， 因为我们不能对用户的PHP编译条件做任何假设。

第四节 小结

在本章，我们知道了如何搭建PHP源码阅读环境，探讨了PHP的源码结构和阅读PHP源码的方法，并且对于一些常用的代码有了一定的了解。 我们希望所有的这些能为源码阅读减轻一些难度，可以更好的关注PHP源码本身的实现。

第二章 概览

在详细阅读源码之前我们先对PHP的整体结构，生命周期，PHP与其它服务器的交互，PHP的整个执行过程等进行一个大概的了解，先有一个整体的概念。

下面，我们先介绍PHP的生命周期。

第一节 生命周期和Zend引擎

一切的开始: SAPI接口

通常我们编写PHP Web程序都是通过Apache或者Nginx这类Web服务器来测试脚本. 或者在命令行下通过php程序来执行PHP脚本. 执行完脚本后，服务器应答，浏览器显示应答信息,或者在命令结束后在标准输出显示内容. 我们很少关心PHP解释器在哪里. 虽然通过Web服务器和命令行程序执行 脚本看起来很不一样. 实际上她们的工作是一样的. 命令行程序和Web程序类似, 命令行参数传递给要执行的脚本,相当于通过url 请求一个PHP页面. 脚本戳里完成后返回响应结果,只不过命令行响应的结果是显示在终端上. 脚本执行的开始都是通过SAPI接口 进行的. 下面几个小节将对一些常见的SAPI进行更为深入的介绍.

开始和结束

PHP通过SAPI开始以后会经过两个主要的阶段. 处理请求之前的开始阶段和请求之后的结束阶段. 开始阶段有两个过程, 第一个是在 整个SAPI生命周期内(例如Apache启动以后的整个生命周期内或者命令行程序整个执行过程中)的开始阶段(MINIT),该阶段只进行一次. 第二个过程 发生在请求阶段,例如通过url请求某个页面.则在每次请求之前都会进行初始化过程(RINIT请求开始). 例如PHP注册了一些PHP模块,则在MINIT阶段会回调所有模块的MINIT函数. 模块在这个阶段可以进行一些初始化工作,例如注册常量, 定义 模块使用的类等等. 一般的模块回调函数

```
PHP_MINIT_FUNCTION(myphpextension)
{
    // 注册常量或者类等初始化操作
    return SUCCESS;
}
```

请求到达之后PHP初始化执行脚本的基本环境,例如创建一个执行环境,包括保存PHP运行过程中变量名称和变量值内容的符号表. 以及当前所有 的函数以及类等信息的符号表. 然后PHP会调用所有模块RINIT函数, 在这个阶段各个模块也可以执行一些相关的操作, 模块的RINIT函数和MINIT函数类似

```
PHP_RINIT_FUNCTION(myphpextension)
{
    // 例如记录请求开始时间
    // 随后在请求结束的时候记录结束时间.这样我们就能够记录下处理请求所花费的时间了
    return SUCCESS;
}
```

请求处理完后就进入了结束阶段, 一般脚本执行到末尾或者通过调用exit()或者die()函数,PHP都将进入

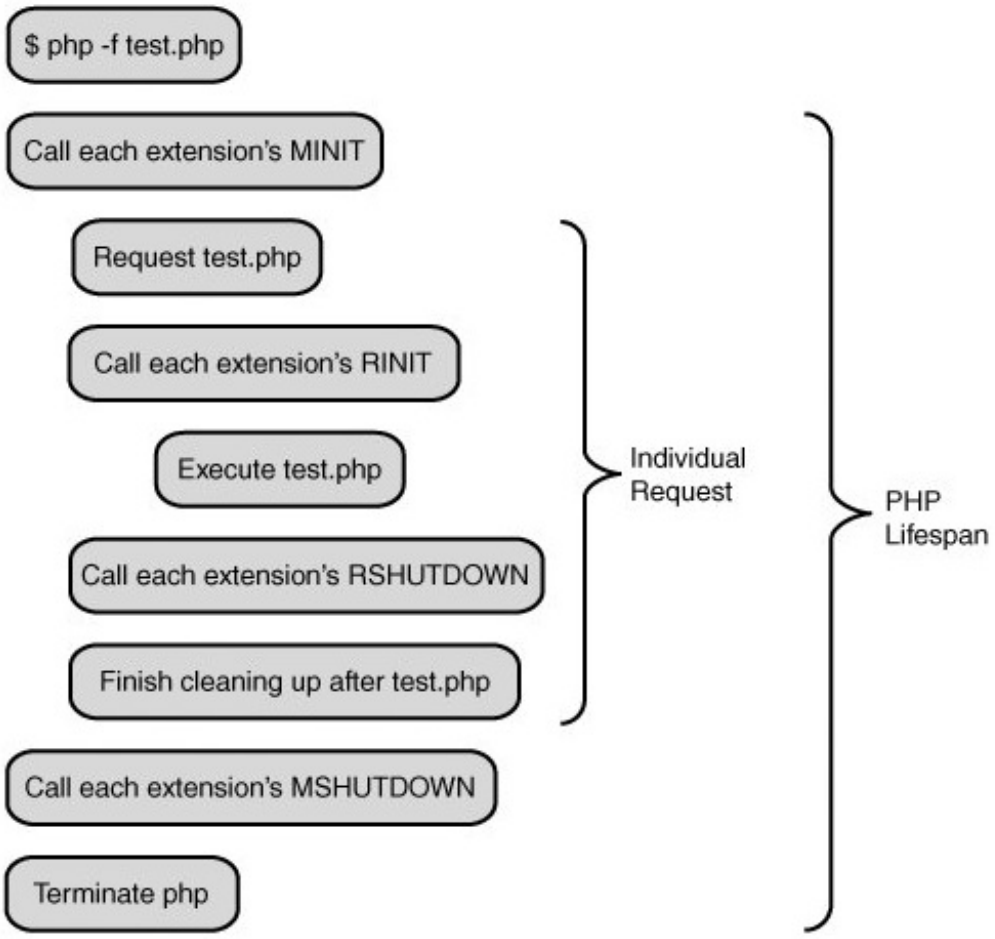
结束阶段. 和开始阶段对应,结束阶段也分为 两个环节,一个在请求结束后(RSHUWDOWN),一个在SAPI生命周期结束时(MSHUTDOWN).

```
PHP_RSHUTDOWN_FUNCTION(myphpextension)
{
    // 例如记录请求结束时间, 并把相应的信息写入到日至文件中.
    return SUCCESS;
}
```

想要了解扩展开发的相关内容,请参考第十三章 扩展开发

单进程SAPI生命周期

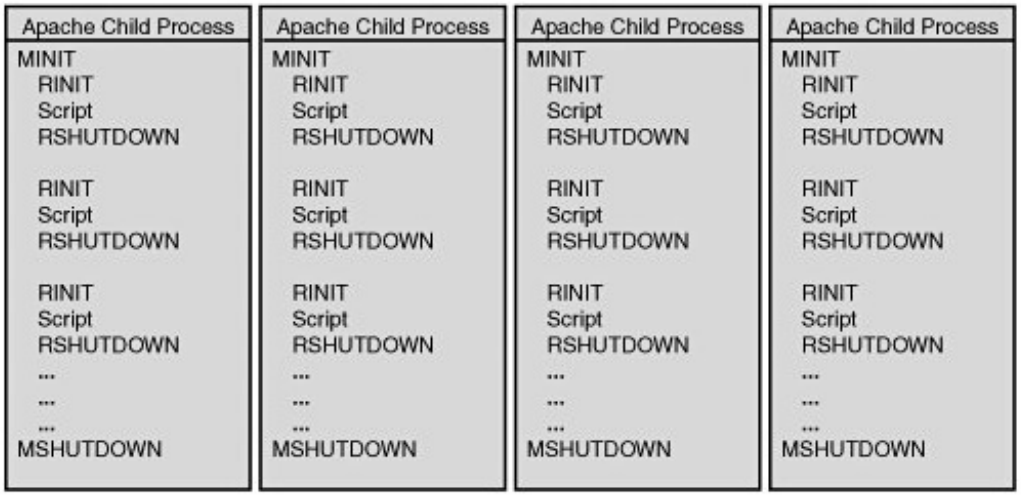
CLI/CGI模式的PHP属于单进程的SAPI模式. 这类的请求在处理一次请求后就关闭. 也就是只会经过如下几个环节 开始 - 请求开始 - 请求关闭 - 结束 SAPI 接口就完成了其生命周期。如图2.1所示:



多进程SAPI生命周期

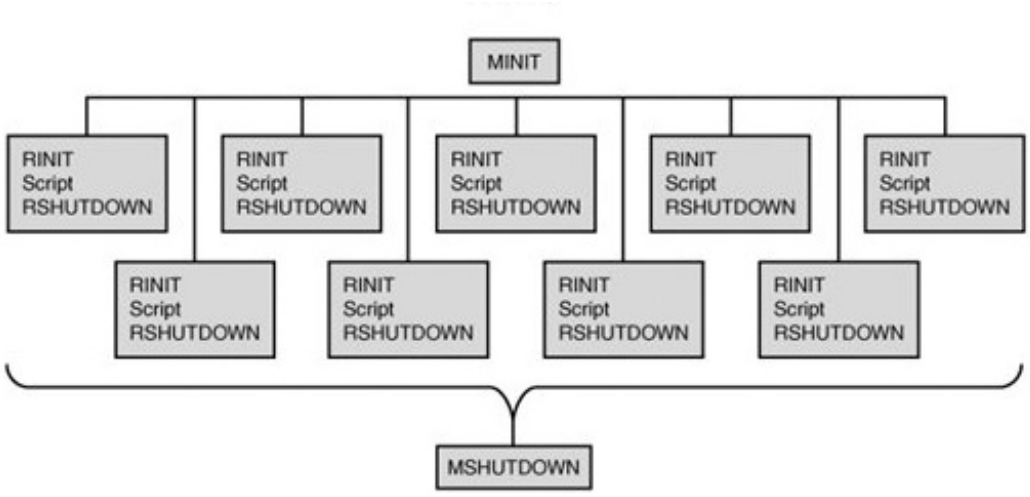
通常PHP是编译为apache的一个模块来处理PHP请求. Apache一般会采用多进程模式, Apache启动后会fork出多个子进程, 每个进程的内存空间独立, 每个子进程都会经过开始和结束环节,不过每个进程的开始阶段只在进程fork出来以来后进行, 在整个进程的生命周期内可能会处理多个请求. 只有 在Apache关闭或者进程被结束之后才会进行关闭阶段,在这两个阶段之间会随着每个请求重复请求开始-请求关闭的环节。如图

2.2所示:



多线程的SAPI生命周期

多线程模式和多进程中的某个进程类似,不同的是在整个进程的生命周期内会**并行**的重复着 请求开始-请求关闭的环节



Zend引擎

Zend引擎作为PHP实现的核心,提供了语言实现上的基础设施.例如: PHP的语法实现, 脚本的编译运行环境, 扩展机制以及内存管理等, 可以说Zend引擎是PHP的核心, 当然这里的PHP指的是官方的PHP实现(除了官方的实现,目前比较知名的有facebook的hiphop实现,不过到目前为止,PHP还没有一个标准的语言规范),而PHP则提供了请求处理和其他Web服务器的接口(SAPI).

参考文献

Extending and Embedding PHP

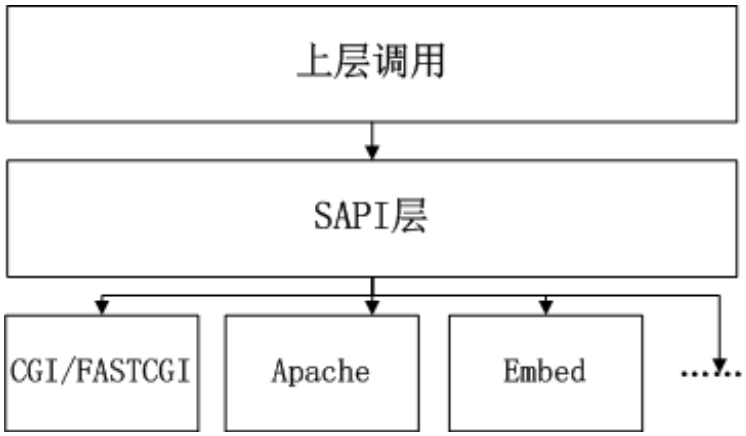
第二节 SAPI概述

前一小节介绍了PHP的生命周期, 所有的请求都是通过SAPI接口实现的. 在源码的SAPI目录存放了PHP对各种服务器抽象层的代码，例如命令行程序的实现, Apache的mod_php模块实现以及fastcgi的实现等等.

在各个服务器抽象层之间遵守着相同的约定，这里我们称之为SAPI接口。每个服务器都需要实现属于自己的_sapi_module_struct结构体。 在PHP的其它模块中，当需要调用服务器相关信息时，全部通过这个结构体的对应方法调用实现，而这对应的方法在各服务器抽象层实现时都会有各自的实现。

其实有很大一部分的方法使用的是默认方法。

如图2.4所示，为SAPI的简单示意图。



如cgi模式和apache2服务器中的启动方法：

```
cgi_sapi_module.startup(&cgi_sapi_module)    //  cgi模式 cgi/cgi_main.c文件

apache2_sapi_module.startup(&apache2_sapi_module);
 //  apache2服务器  apache2handler/sapi_apache2.c文件
```

除了startup方法，sapi_module_struct结构还有许多其它方法。其部分定义如下：

```
struct _sapi_module_struct {
    char *name;          //  名字（标识用）
    char *pretty_name;   //  更好理解的名字（自己翻译的）

    int (*startup)(struct _sapi_module_struct *sapi_module);     //  启动函数
    int (*shutdown)(struct _sapi_module_struct *sapi_module);    //  关闭方法

    int (*activate)(TSRMLS_D);  // 激活
    int (*deactivate)(TSRMLS_D);     //  停用

    int (*ub_write)(const char *str, unsigned int str_length TSRMLS_DC);
     //  不缓存的写操作(unbuffered write)
    void (*flush)(void *server_context);     //  flush
    struct stat *(*get_stat)(TSRMLS_D);      //  get uid
    char *(*getenv)(char *name, size_t name_len TSRMLS_DC); //  getenv

    void (*sapi_error)(int type, const char *error_msg, ...);    /* error
handler */

    int (*header_handler)(sapi_header_struct *sapi_header, sapi_header_op_enum
op,
        sapi_headers_struct *sapi_headers TSRMLS_DC);    /* header handler */
```

```
     /* send headers handler */
    int (*send_headers)(sapi_headers_struct *sapi_headers TSRMLS_DC);

    void (*send_header)(sapi_header_struct *sapi_header,
            void *server_context TSRMLS_DC);   /* send header handler */

    int (*read_post)(char *buffer, uint count_bytes TSRMLS_DC); /* read POST
data */
    char *(*read_cookies)(TSRMLS_D);    /* read Cookies */

    /* register server variables */
    void (*register_server_variables)(zval *track_vars_array TSRMLS_DC);

    void (*log_message)(char *message);     /* Log message */
    time_t (*get_request_time)(TSRMLS_D);   /* Request Time */
    void (*terminate_process)(TSRMLS_D);    /* Child Terminate */

    char *php_ini_path_override;    //  覆盖的ini路径

    ...
    ...
};
```

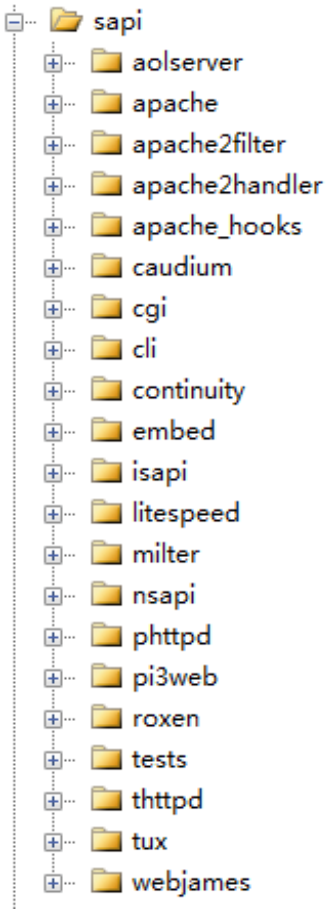
以上的这些结构在各服务器的接口实现中都有定义。如apache2的定义：

```
static sapi_module_struct apache2_sapi_module = {
    "apache2handler",
    "Apache 2.0 Handler",

    php_apache2_startup,                /* startup */
    php_module_shutdown_wrapper,            /* shutdown */

    ...
}
```

在PHP的源码，SAPI存在多个服务接口，其文件结构如图2.5所示：



整个SAPI类似于一个面向对象中的模板方法模式的应用。SAPI.c和SAPI.h文件所包含的一些函数就是模板方法模式中的抽象模板，各个服务器对于sapi_module的定义及相关实现则是一个个具体的模板。只是这里没有继承。

Apache模块

为了让PHP在让Apache服务器上可以运行,我们通常的做法是编译一个Apache的PHP模块(mod_php5)，让此模块来处理PHP文件的所有请求. 当在配置中配置好PHP模块后，PHP模块通过注册apache2的ap_hook_post_config挂钩, 在Apache启动的时候启动mod_php5模块以接受PHP文件的请求.

PHP模块是如何加载到Apache中的?

Apache模块加载机制简介

Apache的模块可以在运行的时候动态装载，这意味着对服务器可以进行功能扩展而不需要重新对源代码进行编译，甚至根本不需要停止服务器。 我们所需要做的仅仅是给服务器发送信号HUP或者AP_SIG_GRACEFUL通知服务器重新载入模块。 但是在动态加载之前，我们需要将模块编译成为动态链接库。此时的动态加载就是加载动态链接库。
Apache中对动态链接库的处理是通过模块mod_so来完成的，因此mod_so模块不能被动态加载， 它只能被静态编译进Apache的核心。这意味着它是随着Apache一起启动的。
比如我们要加载PHP模块，那么首先我们需要在httpd.conf文件中添加一行:

```
LoadModule php5_module modules/mod_php5.so
```

该命令的第一个参数是模块的名称，名称可以在模块实现的源码中找到。第二个选项是该模块所处的路径。 如果需要在服务器运行时加载模块，可以通过发送信号HUP或者AP_SIG_GRACEFUL给服务器，一旦接受到该信号，Apache将重新装载模块，而不需要重新启动服务器。

下面我们以PHP模块的加载为例，分析Apache的模块加载过程。在配置文件中添加了所上所示的指令后，Apache在加载模块时会根据模块名查找模块并加载， 对于每一个模块，Apache必须保证其文件名是以“mod_”开始的，如php的mod_php5.c。如果命名格式不对，Apache将认为此模块不合法。 module结构的name属性在最后是通过宏STANDARD20_MODULE_STUFF以__FILE__体现。 关于这点可以在后面介绍mod_php5模块时有看到。 通过之前指令中指定的路径找到相关的动态链接库文件，Apache通过内部的函数获取动态链接库中的内容，并将模块的内容加载到内存中的指定变量中。

在真正激活模块之前，Apache会检查所加载的模块是否为真正的Apache模块，这个检测是通过检查magic字段进行的。 而magic字段是通过宏STANDARD20_MODULE_STUFF体现，在这个宏中magic的值为MODULE_MAGIC_COOKIE，MODULE_MAGIC_COOKIE定义如下：

```
#define MODULE_MAGIC_COOKIE 0x41503232UL /* "AP22" */
```

最后Apache会调用相关函数(ap_add_loaded_module)将模块激活，此处的激活就是将模块放入相应的链表中(ap_top_modules链表：ap_top_modules链表用来保存Apache中所有的被激活的模块，包括默认的激活模块和激活的第三方模块。)

Apache加载的是PHP模块，那么这个模块是如何实现的呢到我们以Apache2的mod_php5模块为例进行说明。

Apache2的mod_php5模块说明

Apache2的mod_php5模块包括sapi/apache2handler和sapi/apache2filter两个目录 在□apache2_handle/mod_php5.c文件中，模块定义的相关代码如下：

```
AP_MODULE_DECLARE_DATA module php5_module = {
    STANDARD20_MODULE_STUFF,
        /* 宏，包括版本，小版本，模块索引，模块名，下一个模块指针等信息，其中模块名以__FILE__体现 */
    create_php_config,      /* create per-directory config structure */
    merge_php_config,       /* merge per-directory config structures */
    NULL,                   /* create per-server config structure */
    NULL,                   /* merge per-server config structures */
    php_dir_cmds,           /* 模块定义的所有的指令 */
    php_ap2_register_hook
        /* 注册钩子，此函数通过ap_hoo_开头的函数在一次请求处理过程中对于指定的步骤注册钩子 */
};
```

它所对应的是apache的module结构，module的结构定义如下：

```
typedef struct module_struct module;
struct module_struct {
    int version;
    int minor_version;
    int module_index;
```

```
    const char *name;
    void *dynamic_load_handle;
    struct module_struct *next;
    unsigned long magic;
    void (*rewrite_args) (process_rec *process);
    void *(*create_dir_config) (apr_pool_t *p, char *dir);
    void *(*merge_dir_config) (apr_pool_t *p, void *base_conf, void *new_conf);
    void *(*create_server_config) (apr_pool_t *p, server_rec *s);
    void *(*merge_server_config) (apr_pool_t *p, void *base_conf, void
*new_conf);
    const command_rec *cmds;
    void (*register_hooks) (apr_pool_t *p);
}
```

上面的模块结构与我们在mod_php5.c中所看到的结构有一点不同，这是由于STANDARD20_MODULE_STUFF的原因，这个宏它包含了前面8个字段的定义。STANDARD20_MODULE_STUFF宏的定义如下：

```
/** Use this in all standard modules */
#define STANDARD20_MODULE_STUFF MODULE_MAGIC_NUMBER_MAJOR, \
                MODULE_MAGIC_NUMBER_MINOR, \
                -1, \
                __FILE__, \
                NULL, \
                NULL, \
                MODULE_MAGIC_COOKIE, \
                                NULL      /* rewrite args spot */
```

php_dir_cmds所定义的内容如下：

```
const command_rec php_dir_cmds[] =
{
    AP_INIT_TAKE2("php_value", php_apache_value_handler, NULL,
        OR_OPTIONS, "PHP Value Modifier"),
    AP_INIT_TAKE2("php_flag", php_apache_flag_handler, NULL,
        OR_OPTIONS, "PHP Flag Modifier"),
    AP_INIT_TAKE2("php_admin_value", php_apache_admin_value_handler,
        NULL, ACCESS_CONF|RSRC_CONF, "PHP Value Modifier (Admin)"),
    AP_INIT_TAKE2("php_admin_flag", php_apache_admin_flag_handler,
        NULL, ACCESS_CONF|RSRC_CONF, "PHP Flag Modifier (Admin)"),
    AP_INIT_TAKE1("PHPINIDir", php_apache_phpini_set, NULL,
        RSRC_CONF, "Directory containing the php.ini file"),
    {NULL}
};
```

以上为php模块定义的指令表。它实际上是一个command_rec结构的数组。 当Apache遇到指令的时候将逐一遍历各个模块中的指令表，查找是否有哪个模块能够处理该指令， 如果找到，则调用相应的处理函数，如果所有指令表中的模块都不能处理该指令，那么将报错。 如上可见，php模块仅提供php_value等5个指令。

php_ap2_register_hook函数的定义如下：

```
void php_ap2_register_hook(apr_pool_t *p)
{
    ap_hook_pre_config(php_pre_config, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_post_config(php_apache_server_startup, NULL, NULL,
APR_HOOK_MIDDLE);
```

```
    ap_hook_handler(php_handler, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_child_init(php_apache_child_init, NULL, NULL, APR_HOOK_MIDDLE);
}
```

以上代码声明了pre_config,post_config,handler和child_init 4个挂钩以及对应的处理函数。 其中pre_config,post_config,child_init是启动挂钩，它们在服务器启动时调用。 handler挂钩是请求挂钩，它在服务器处理请求时调用。其中在post_config挂钩中启动php。 它通过php_apache_server_startup函数实现。php_apache_server_startup函数通过调用sapi_startup启动sapi, 并通过调用php_apache2_startup来注册sapi module struct（此结构在本节开头中有说明）, 最后调用php_module_startup来初始化PHP, 其中又会初始化ZEND引擎,以及填充zend_module_struct中 的treat_data成员(通过php_startup_sapi_content_types)等。

Apache的运行过程

Apache的运行分为启动阶段和运行阶段。 在启动阶段，Apache为了获得系统资源最大的使用权限，将以特权用户root（*nix系统）或超级管理员Administrator(Windows系统)完成启动，并且整个过程处于一个单进程单线程的环境中，。 这个阶段包括配置文件解析(如http.conf文件)、模块加载(如mod_php,mod_perl)和系统资源初始化（例如日志文件、共享内存段、数据库连接等）等工作。

Apache的启动阶段执行了大量的初始化操作，并且将许多比较慢或者花费比较高的操作都集中在这个阶段完成，以减少了后面处理请求服务的压力。

在运行阶段，Apache主要工作是处理用户的服务请求。 在这个阶段，Apache放弃特权用户级别，使用普通权限，这主要是基于安全性的考虑，防止由于代码的缺陷引起的安全漏洞。 Apache对HTTP的请求可以分为连接、处理和断开连接三个大的阶段。同时也可以分为11个小的阶段，依次为： Post-Read-Request，URI Translation，Header Parsing，Access Control，Authentication，Authorization，MIME Type Checking，FixUp，Response，Logging，CleanUp

Apache Hook机制

Apache 的Hook机制是指：Apache 允许模块(包括内部模块和外部模块，例如mod_php5.so,mod_perl.so等)将自定义的函数注入到请求处理循环中。换句话说，模块可以在 Apache的任何一个处理阶段中挂接(Hook)上自己的处理函数，从而参与Apache的请求处理过程。 mod_php5.so/php5apache2.dll就是将所包含的自定义函数，通过Hook机制注入到Apache中，在Apache处理流程的各个阶段负责处理php请求。 关于Hook机制在Windows系统开发也经常遇到，在Windows开发既有系统级的钩子，又有应用级的钩子。

以上介绍了apache的加载机制，hook机制，apache的运行过程以及php5模块的相关知识，下面简单的说明在查看源码中的一些常用对象。

Apache常用对象

在说到Apache的常用对象时，我们不得不先说下httpd.h文件。httpd.h文件包含了Apache的所有模块都需要的核心API。 它定义了许多系统常量。但是更重要的是它包含了下面一些对象的定义。

request_rec对象

当一个客户端请求到达Apache时，就会创建一个request_rec对象，当Apache处理完一个请求后，与这个请求对应的request_rec对象也会随之被释放。 request_rec对象包括与一个HTTP请求相关的所有数据，并且还包含一些Apache自己要用到的状态和客户端的内部字段。

server_rec对象

server_rec定义了一个逻辑上的WEB服务器。如果有定义虚拟主机，每一个虚拟主机拥有自己的server_rec对象。 server_rec对象在Apache启动时创建，当整个httpd关闭时才会被释放。 它包括服务器名称，连接信息，日志信息，针对服务器的配置，事务处理相关信息等 server_rec对象是继request_rec对象之后第二重要的对象。

conn_rec对象

conn_rec对象是TCP连接在Apache的内部体现。 它在客户端连接到服务器时创建，在连接断开时释放。

参考资料

《The Apache Modules Book--Application Development with Apache》

嵌入式

从第一章中对PHP源码目录结构的介绍以及PHP生命周期章节中可以看出，嵌入式PHP类似CLI,是SAPI接口的另一种实现。 一般情况下，它的一个请求的生命周期也会和其它的SAPI一样：模块初始化=>请求初始化=>处理请求=>关闭请求=>关闭模块。当然，这只是理想情况。 但是嵌入式PHP的请求可能包括一段或多段代码，这就导致了嵌入式PHP的特殊性。

对于嵌入式PHP或许我们了解比较少，或者说根本用不到，甚至在网上相关的资料也不多， 所幸我们在《Extending and Embedding PHP》这本书上找到了一些资料。 这一小节，我们从这本书的一个示例说起，介绍PHP对于嵌入式PHP的支持以及PHP为嵌入式提供了哪些接口或功能。 首先我们看下所要用到的示例源码：

```
#include <sapi/embed/php_embed.h>
#ifdef ZTS
    void ***tsrm_ls;
#endif
/* Extension bits */
zend_module_entry php_mymod_module_entry = {
    STANDARD_MODULE_HEADER,
    "mymod", /* extension name */
    NULL, /* function entries */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
    "1.0", /* version */
    STANDARD_MODULE_PROPERTIES
};
/* Embedded bits */
static void startup_php(void)
{
```

```
    int argc = 1;
    char *argv[2] = { "embed5", NULL };
    php_embed_init(argc, argv PTSRMLS_CC);
    zend_startup_module(&php_mymod_module_entry);
}
static void execute_php(char *filename)
{
    zend_first_try {
        char *include_script;
        spprintf(&include_script, 0, "include '%s'", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    } zend_end_try();
}
int main(int argc, char *argv[])
{
    if (argc <= 1) {
        printf("Usage: embed4 scriptfile";);
        return -1;
    }
    startup_php();
    execute_php(argv[1]);
    php_embed_shutdown(TSRMLS_CC);
    return 0;
}
```

以上的代码可以在《Extending and Embedding PHP》在第20章找到（原始代码有一个符号错误，有兴趣的童鞋可以去围观下）。 上面的代码是一个嵌入式PHP运行器（我们权当其为运行器吧），在这个运行器上我们可以运行PHP代码。 这段代码包括了对于PHP嵌入式支持的声明，启动嵌入式PHP运行环境，运行PHP代码，关闭嵌入式PHP运行环境。 下面我们就这段代码分析PHP对于嵌入式的支持做了哪些工作。 首先看下第一行：

```
#include <sapi/embed/php_embed.h>
```

在sapi目录下的embed目录是PHP对于嵌入式的抽象层所在。在这里有我们所要用到的函数或宏定义。如示例中所使用的php_embed_init，php_embed_shutdown等函数。

第2到4行：

```
 #ifdef ZTS
    void ***tsrm_ls;
#endif
```

ZTS是Zend Thread Safety的简写，与这个相关的有一个TSRM（线程安全资源管理）的东东，这个后面的章节会有详细介绍，这里就不再作阐述。

第6到17行：

```
 zend_module_entry php_mymod_module_entry = {
    STANDARD_MODULE_HEADER,
    "mymod", /* extension name */
    NULL, /* function entries */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
```

```
    NULL, /* MINFO */
    "1.0", /* version */
    STANDARD_MODULE_PROPERTIES
};
```

以上PHP内部的模块结构声明，此处对于模块初始化，请求初始化等函数指针均为NULL, 也就是模块在初始化及请求开始结束等事件发生的时候不执行任何操作。不过这些操作在sapi/embed/php_embed.c文件中的php_embed_shutdown等函数中有体现。 关于模块结构的定义在zend/zend_modules.h中。

startup_php函数:

```
static void startup_php(void)
{
    int argc = 1;
    char *argv[2] = { "embed5", NULL };
    php_embed_init(argc, argv PTSRMLS_CC);
    zend_startup_module(&php_mymod_module_entry);
}
```

这个函数调用了两个函数php_embed_init和zend_startup_module完成初始化工作。 php_embed_init函数定义在sapi/embed/php_embed.c文件中。它完成了PHP对于嵌入式的初始化支持。zend_startup_module函数是PHP的内部API函数，它的作用是注册定义的模块，这里是注册mymod模块。 这个注册过程仅仅是将所定义的zend_module_entry结构添加到注册模块列表中。

execute_php函数:

```
static void execute_php(char *filename)
{
    zend_first_try {
        char *include_script;
        spprintf(&include_script, 0, "include '%s'", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    } zend_end_try();
}
```

从函数的名称来看，这个函数的功能是执行PHP代码的。 它通过调用sprrintf函数构造一个include语句，然后再调用zend_eval_string函数执行这个include语句。 zend_eval_string最终是调用zend_eval_stringl函数，这个函数是流程是一个编译PHP代码，生成zend_op_array类型数据，并执行opcode的过程。 这段程序相当于下面的这段php程序, 这段程序可以用php命令来执行，虽然下面这段程序没有实际意义，而通过嵌入式PHP中，你可以在一个用C实现的 系统中嵌入PHP, 然后用PHP来实现功能。

```
<?php
if($argc < 2) die("Usage: embed4 scriptfile");

include $argv[1];
```

main函数:

```
int main(int argc, char *argv[])
{
    if (argc <= 1) {
        printf("Usage: embed4 scriptfile";);
```

```
        return -1;
    }
    startup_php();
    execute_php(argv[1]);
    php_embed_shutdown(TSRMLS_CC);
    return 0;
}
```

这个函数是主函数，执行初始化操作，根据输入的参数执行PHP的include语句，最后执行关闭操作，返回。 其中php_embed_shutdown函数定义在sapi/embed/php_embed.c文件中。它完成了PHP对于嵌入式的关闭操作支持。包括请求关闭操作，模块关闭操作等。

其它宏

```
#define PHP_EMBED_START_BLOCK(x,y) { \
    php_embed_init(x, y); \
    zend_first_try {

#endif

#define PHP_EMBED_END_BLOCK() \
  } zend_catch { \
    /* int exit_status = EG(exit_status); */ \
  } zend_end_try(); \
  php_embed_shutdown(TSRMLS_C); \
}
```

如上两个宏可能会用到你的嵌入式PHP中，从代码中可以看出，它包含了在示例代码中的php_embed_init，zend_first_try，zend_end_try，php_embed_shutdown等 嵌入式PHP中常用的方法。□ 大量的使用宏也算是PHP源码的一大特色吧，

参与资料

《Extending and Embedding PHP》

FastCGI

FastCGI简介

什么是CGI

CGI全称是“通用网关接口”(Common Gateway Interface)， 它可以让一个客户端，从网页浏览器向执行在Web服务器上的程序，请求数据。 CGI描述了客户端和这个程序之间传输数据的一种标准。 CGI的一个目的是要独立于任何语言的，所以CGI可以用任何一种语言编写，只要这种语言具有标准输入、输出和环境变量。如php,perl,tcl等

什么是FastCGI

FastCGI像是一个常驻(long-live)型的CGI，它可以一直执行着，只要激活后，不会每次都要花费时间去fork一次(这是CGI最为人诟病的fork-and-execute 模式)。它还支持分布式的运算, 即 FastCGI 程序可以在网站服务器以外的主机上执行并且接受来自其它网站服务器来的请求。

FastCGI是语言无关的、可伸缩架构的CGI开放扩展，其主要行为是将CGI解释器进程保持在内存中并因此获得较高的性能。众所周知，CGI解释器的反复加载是CGI性能低下的主要原因，如果CGI解释器保持在内存中并接受FastCGI进程管理器调度，则可以提供良好的性能、伸缩性、Fail- Over特性等等。

FastCGI的工作原理

1. Web Server启动时载入FastCGI进程管理器（IIS ISAPI或Apache Module)
2. FastCGI进程管理器自身初始化，启动多个CGI解释器进程(可见多个php-cgi)并等待来自Web Server的连接。
3. 当客户端请求到达Web Server时，FastCGI进程管理器选择并连接到一个CGI解释器。Web server将CGI环境变量和标准输入发送到FastCGI子进程php-cgi。
4. FastCGI子进程完成处理后将标准输出和错误信息从同一连接返回Web Server。当FastCGI子进程关闭连接时，请求便告处理完成。FastCGI子进程接着等待并处理来自FastCGI进程管理器(运行在Web Server中)的下一个连接。 在CGI模式中，php-cgi在此便退出了。

PHP中的CGI实现

PHP的cgi实现本质是是以socket编程实现一个tcp或udp协议的服务器，当启动时，创建tcp/udp协议的服务器的socket监听，并接收相关请求进行处理。这只是请求的处理，在此基础上添加模块初始化，sapi初始化，模块关闭，sapi关闭等就构成了整个cgi的生命周期。 程序是从cgi_main.c文件的main函数开始，而在main函数中调用了定义在fastcgi.c文件中的初始化，监听等函数。我们从main函数开始，看看PHP对于fastcgi的实现。

这里将整个流程分为初始化操作，请求处理，关闭操作三个部分。 我们就整个流程进行简单的说明，并在其中穿插介绍一些用到的重要函数。

初始化操作

过程说明见代码注释

```
/* {{{ main
 */
int main(int argc, char *argv[])
{
...
sapi_startup(&cgi_sapi_module);
    //  1512行 启动sapi,调用sapi全局构造函数，初始化sapi_globals_struct结构体
... //  根据启动参数，初始化信息
```

```
if (cgi_sapi_module.startup(&cgi_sapi_module) == FAILURE) {
    //  模块初始化 调用php_cgi_startup方法
...
}

...
if (bindpath) {
    fcgi_fd = fcgi_listen(bindpath, 128);    //  实现socket监听，调用fcgi_init初始化
    ...
}

if (fastcgi) {
    ...
    /* library is already initialized, now init our request */
    fcgi_init_request(&request, fcgi_fd);    //  request内存分配，初始化变量
}
```

fcgi_listen函数主要用于创建、绑定socket并开始监听

```
    if ((listen_socket = socket(sa.sa.sa_family, SOCK_STREAM, 0)) < 0 ||
        ...
        bind(listen_socket, (struct sockaddr *) &sa, sock_len) < 0 ||
        listen(listen_socket, backlog) < 0) {
        ...
    }
```

请求处理操作流程

过程说明见代码注释

```
    while (parent) {
        do {
            pid = fork();    //  生成新的子进程
            switch (pid) {
            case 0: //  子进程
                parent = 0;

                /* don't catch our signals */
                sigaction(SIGTERM, &old_term, 0);    //  终止信号
                sigaction(SIGQUIT, &old_quit, 0);    //  终端退出符
                sigaction(SIGINT,  &old_int,  0);    //  终端中断符
                break;
                ...
                default:
                /* Fine */
                running++;
                break;
        } while (parent && (running < children));

    ...
        while (!fastcgi || fcgi_accept_request(&request) >= 0) {
        SG(server_context) = (void *) &request;
        init_request_info(TSRMLS_C);
        CG(interactive) = 0;
                    ...
            }
```

在fcgi_accept_request函数中，处理连接请求，忽略受限制客户的请求，调用fcgi_read_request函数（定义在fastcgi.c文件），分析请求的信息，将相关的变量写到对应的变量中。 其中在读取请求内容时调用了safe_read方法。如下所示： **[main() -> fcgi_accept_request() -> fcgi_read_request() -> safe_read()]**

```
static inline ssize_t safe_read(fcgi_request *req, const void *buf, size_t
count)
{
    size_t n = 0;
    do {
    ... //  省略  对win32的处理
        ret = read(req->fd, ((char*)buf)+n, count-n);   //  非win版本的读操作
    ... //  省略
    } while (n != count);

}
```

在请求初始化完成，读取请求完毕后，就该处理请求的PHP文件了。假设此次请求为PHP_MODE_STANDARD则会调用php_execute_script执行PHP文件。 在此函数中它先初始化此文件相关的一些内容，然后再调用zend_execute_scripts函数，对PHP文件进行词法分析和语法分析，生成中间代码， 并执行zend_execute函数，从而执行这些中间代码。关于整个脚本的执行请参见第三节 脚本的执行。

关闭操作流程

过程说明代码注释

```
...
php_request_shutdown((void *) 0);   //  php请求关闭函数
...
fcgi_shutdown();    //  fcgi的关闭 销毁fcgi_mgmt_vars变量
php_module_shutdown(TSRMLS_C);  //  模块关闭    清空sapi,关闭zend引擎 销毁内存，清
除垃圾等
sapi_shutdown();    //  sapi关闭  sapi全局变量关闭等
...
```

参考资料

以下为本篇文章对于一些定义引用的参考资料：
http://www.fastcgi.com/drupal/node/2
http://baike.baidu.com/view/641394.htm

第三节 PHP脚本的执行

在前面的章节介绍了PHP的生命周期，PHP的SAPI，SAPI处于PHP整个架构较上层，而真正脚本的执行主要由Zend引擎来完成, 这一小节我们介绍PHP脚本的执行。

目前编程语言可以分为两大类:

- 第一类是像C/C++, .NET, Java之类的编译型语言, 它们的共性是: 运行之前必须对源代码进行编译,然后运行编译后的目标文件.
- 第二类比如:PHP, Javascript, Ruby, Python这些解释型语言, 他们都无需经过编译即可"运行". 虽然可以理解为直接运行, 但它们并不是真的直接就被能被机器理解, 机器只能理解机器语言,那这些语言是怎么被执行的呢, 一般这些语言都需要一个**解释器**, 由解释器来执行这些源码, 实际上这些语言还是会经过编译环节, 只不过它们一般会在运行的时候实时进行编译. 为了效率,并不是所有语言在每次执行的时候都会重新编译一遍, 比如PHP的各种opcode缓存扩展(如APC, xcache, eAccelerator等),比如Python会将编译的中间文件保存成pyc/pyo文件,避免每次运行重新进行编译所带来的性能损失.

PHP的脚本的执行也需要一个解释器, 比如命令行下的php程序,或者apache的mod_php模块等等. 前一节提到了PHP的SAPI接口, 下面就以PHP命令行程序为例解释PHP脚本是怎么被执行的. 例如如下的这段PHP脚本:

```
<?php
$str = "Hello, Tipi!\n";
echo $str;
```

假设上面的代码保存在名为hello.php的文件中, 用PHP命令行程序执行这个脚本:

```
$ php ./hello.php
```

这段代码的输出显然是Hello, Tipi!, 那么在执行脚本的时候PHP/Zend都做了些什么呢? 这些语句是怎么样让php输出这段话的呢? 下面将一步一步的进行介绍.

程序的执行

1. 如上例中, 传递给php程序需要执行的文件, php程序完成基本的准备工作后启动PHP及Zend引擎, 加载注册的扩展模块.
2. 初始化完成后读取脚本文件,Zend引擎对脚本文件进行词法分析,语法分析. 然后编译成opcode 执行. 如过安装了apc之类的opcode缓存, 编译环节可能会被跳过而直接从缓存中读取opcode执行.

脚本的编译执行

PHP在读取到脚本文件后首先对代码进行词法分析, PHP的词法分析器是通过lex生成的, 词法规则文件在$PHP_SRC/Zend/zend_language_scanner.l, 这一阶段lex会会将源代码按照词法规则切分一个一个的标记(token). PHP中提供了一个函数token_get_all(), 该函数接收一个字符串参数, 返回一个按照词法规则切分好的数组. 例如将上面的php代码作为参数传递给这个函数:

```
<?php
$code =<<<PHP_CODE
<?php
$str = "Hello, Tipi\n";
echo $str;
PHP_CODE;
```

```
var_dump(token_get_all($code));
```

运行上面的脚本你将会看到一如下的输出

```
array (
  0 =>
  array (
    0 => 368,        // 脚本开始标记
    1 => '<?php      // 匹配到的字符串
',
    2 => 1,
  ),
  1 =>
  array (
    0 => 371,
    1 => ' ',
    2 => 2,
  ),
  2 => '=',
  3 =>
  array (
    0 => 371,
    1 => ' ',
    2 => 2,
  ),
  4 =>
  array (
    0 => 315,
    1 => '"Hello, Tipi
"',
    2 => 2,
  ),
  5 => ';',
  6 =>
  array (
    0 => 371,
    1 => '
',
    2 => 3,
  ),
  7 =>
  array (
    0 => 316,
    1 => 'echo',
    2 => 4,
  ),
  8 =>
  array (
    0 => 371,
    1 => ' ',
    2 => 4,
  ),
  9 => ';',
```

这也是Zend引擎词法分析做的事情,将代码切分为一个个的标记,然后使用语法分析器(PHP使用bison生成语法分析器, 规则见$PHP_SRC/Zend/zend_language_parser.y), bison根据规则进行相应的处理, 如果代码找不到匹配的规则,也就是语法错误时Zend引擎会停止,并输出错误信息. 比如缺少括号,或者不符合语法规则的情况都会在这个环节检查. 在匹配到相应的语法规则后,Zend引擎还会进行编译, 将代码编译为opcode, 完成后,Zend引擎会执行这些opcode, 在执行opcode的过程中还有可能会继续重复进行编译-执行,

例如执行eval,include/require等语句, 因为这些语句还会包含或者执行其他文件或者字符串中的脚本.

例如上例中的echo语句会编译为一条ZEND_ECHO指令, 执行过程中,该指令由C函数zend_print_variable(zval* z)执行,将传递进来的字符串打印出来. 为了方便理解, 本例中省去了一些细节,例如opcode指令和处理函数之间的映射关系等. 后面的章节将会详细介绍.

如果想直接查看生成的Opcode，可以使用php的vld扩展查看。扩展下载地址: http://pecl.php.net/package/vld。Win下需要自己编译生成dll文件。

有关PHP脚本编译执行的细节,请阅读后面有关词法分析,语法分析及opcode编译相关内容.

词法分析和语法分析

编程语言的编译器(compiler)或解释器(interpreter)一般包括两大部分:

1. 读取源程序, 并处理语言结构
2. 处理语言结构并生成目标程序

Lex和Yacc可以解决第一个问题. 第一个部分也可以分为两个部分:

1. 将代码切分为一个个的标记(token).
2. 处理程序的层级结构(hierarchical structure).

很多编程语言都使用lex/yacc或他们的变体(flex/bison)来作为语言的词法语法分析生成器, 比如PHP, Ruby, Python以及MySQL的SQL语言实现.

Lex和Yacc是Unix下的两个文本处理工具, 主要用于编写编译器, 也可以做其他用途,

- Lex(词法分析生成器:A Lexical Analyzer Generator)
- Yacc(Yet Another Compiler-Compiler)

Lex/Flex

Lex读取词法规则文件,生成词法分析器. 目前通常使用Flex以及Bison来完成同样的工作, Flex和lex之间并不兼容, Bison则是兼容Yacc的实现.

词法规则文件一般以.l作为扩展名, flex文件由三个部分组成, 三部分之间用%%分割

```
定义段
%%
规则段
%%
用户代码段
```

例如以下一个用于统计文件字符, 词以及行数的例子:

```
%option noyywrap
%{
int chars = 0;
int words = 0;
```

```
int lines = 0;
%}

%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n   { chars++; lines++; }
.    { chars++; }
%%

main(int argc, char **argv)
{
    if(argc > 1) {
        if(!(yyin = fopen(argv[1], "r"))) {
            perror(argv[1]);
            return (1);
        }
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
    }
}
```

该解释器读取文件内容, 根据规则段定义的规则进行处理, 规则后面大括号中包含的是动作, 也就时匹配到该规则程序执行的动作, 这个例子中的匹配动作时记录下文件的字符,词以及行数信息并打印出来. 其中的规则使用正则表达式描述.

回到PHP的实现, PHP以前使用的是flex, 后来PHP的词法解析改为使用re2c, $PHP_SRC/Zend/zend_language_scanner.l 文件是re2c的规则文件, 所以如果修改该规则文件需要安装re2c才能重新编译.

Yacc/Bison

> PHP在后续的版本中可能会使用Lemon作为语法分析器, Lemon是SQLite作者为SQLite中SQL所编写的词法分析器. Lemno具有线程安全以及可重入等特点, 也能提供更直观的错误提示信息.

Bison和Flex类似, 也是使用%%作为分界不过Bison接受的是标记(token)序列, 根据定义的语法规则,来执行一些动作, Bison使用巴科斯范式(BNF)来描述语法, php中echo语句可以接受多个参数, 这几个参数之间可以使用逗号分隔, 在PHP的语法规则:

```
echo_expr_list:
        echo_expr_list ',' expr { zend_do_echo(&$3 TSRMLS_CC); }
    |   expr                    { zend_do_echo(&$1 TSRMLS_CC); }
;
```

其中echo_expr_list规则为一个递归规则, 这样就允许接受多个表达式作为参数. 在每个规则后面有一段大括号起来的语句, 这个称为动作,在上面的例子当中echo时会执行zend_do_echo函数, 函数中的参数可能看起来比较奇怪, 其中的$3 表示前面规则的第三个定义,也就是expr这个表达式的值, zend_do_echo函数则根据表达式的信息编译opcode, php中其他的语法规则也是类似, 下面将介绍继续介绍PHP中的opcode.

Opcode

Opcode在wikipedia中的解释是: opcode是计算机指令中的一部分, 用于指定要执行的操作, 指令的格式和规范由处理器的指令规范指定. 除了指令本身以外通常还有指令所需要的操作数, 可能有的指令不需要显式的操作数. 这些操作数可能是寄存器中的值, 堆栈中的值, 某块内存的值或者IO端口中的值等等.

通常opcode还有另一种称谓: 字节码(byte codes). 例如Java虚拟机(JVM), .NET的通用中间语言(CIL: Common Intermeditate Language)等等.

PHP的opcode

PHP中的opcode则属于前面介绍中的后着, PHP是构建在Zend虚拟机(Zend VM)之上的. PHP中的opcode就相当于Zend虚拟机中的指令.

> 有关Zend虚拟机的介绍请阅读后面相关内容

opcode在PHP的内部实现表示如下:

```
struct _zend_op {
    opcode_handler_t handler; // 执行该opcode时调用的处理函数
    znode result;
    znode op1;
    znode op2;
    ulong extended_value;
    uint lineno;
    zend_uchar opcode;  // opcode代码
};
```

和CPU的指令类似, 有一个标示指令的opcode字段, 以及这个opcode所操作的操作数, PHP不像汇编那么底层, 在脚本实际执行的时候可能还需要其他更多的信息, extended_value字段就保存了这类信息, 其中的result域则是保存该指令执行完成后的结果.

例如如下代码时在编译器遇到print语句的时候进行编译的函数:

```
void zend_do_print(znode *result, const znode *arg TSRMLS_DC)
{
    zend_op *opline = get_next_op(CG(active_op_array) TSRMLS_CC);

    opline->result.op_type = IS_TMP_VAR;
    opline->result.u.var = get_temporary_variable(CG(active_op_array));
    opline->opcode = ZEND_PRINT;
    opline->op1 = *arg;
    SET_UNUSED(opline->op2);
    *result = opline->result;
}
```

这个函数新创建一条zend_op, 将返回值的类型设置为临时变量(IS_TMP_VAR), 并为临时变量申请空间, 随后指定opcode为ZEND_PRINT, 并将传递进来的参数赋值给 这条opcode的第一个操作数.

下面这个函数是在编译器遇到echo语句的时候进行编译的函数:

```
void zend_do_echo(const znode *arg TSRMLS_DC)
{
    zend_op *opline = get_next_op(CG(active_op_array) TSRMLS_CC);
```

```
    opline->opcode = ZEND_ECHO;
    opline->op1 = *arg;
    SET_UNUSED(opline->op2);
}
```

可以看到echo处理除了指定opcode以外,还将echo的参数传递给op1, 这里并没有设置opcode的result结果字段。 从这里我们也能看出print和echo的区别来, print有返回值,而echo没有, 这里的没有和返回null是不同的, 如果尝试将echo的值赋值给某个变量或者传递给函数都会出现语法错误.

PHP脚本编译为opcode保存在op_array中, 其内部存储的结构如下:

```
struct _zend_op_array {
    /* Common elements */
    zend_uchar type;
    char *function_name;  // 如果是用户定义的函数则, 这里将保存函数的名字
    zend_class_entry *scope;
    zend_uint fn_flags;
    union _zend_function *prototype;
    zend_uint num_args;
    zend_uint required_num_args;
    zend_arg_info *arg_info;
    zend_bool pass_rest_by_reference;
    unsigned char return_reference;
    /* END of common elements */

    zend_bool done_pass_two;

    zend_uint *refcount;

    zend_op *opcodes;  // opcode数组

    zend_uint last, size;

    zend_compiled_variable *vars;
    int last_var, size_var;

    // ...
}
```

如上面的注释, opcodes保存在这里, 在执行的时候由下面的execute函数执行:

```
ZEND_API void execute(zend_op_array *op_array TSRMLS_DC)
{
    // ... 循环执行op_array中的opcode或者执行其他op_array中的opcode
}
```

前面提到每条opcode都有一个opcode_handler_t的函数指针字段,用于执行该opcode, 这里并没有给没有指定处理函数, 那在执行的时候该由哪个函数来执行呢? 更多信息请参考Zend虚拟机相关章节的详细介绍.

第四节 小结

本章中对PHP进行了一个宏观的介绍, 从PHP的生命周期开始,介绍了各种SAPI实现以及它们的特殊性, 最后通过脚本的执行了解了PHP脚本是怎样被执行的.

下一章将从语言最基本的结构: 变量开始了解PHP.

第三章 变量及数据类型

变量，是指没有固定的值，是可以改变的。PHP 中的变量以一个美元符号开头，后面跟变量名。 一个有效的变量名由字母或者下划线开头，后面跟上任意数量的字母，数字，或者下划线。

变量名是区分大小写的。

在PHP中，我们经常会见到或用到的与变量相关的一些概念，比如常量，全局变量，静态变量以及类型转换等。 本章我们将介绍这些与变量相关的实现。其中包括PHP本身的变量结构以及弱类型的实现，常量，预定义变量，静态变量，类型提示，变量范围和类型转换等。

先看一段PHP代码:

```
$var1 = 10;
$var2 = 20;

function t() {
    global $var1;
    $var2 = 0;
    $var1++;
}

t();
echo $var1, ' ';
echo $var2;
```

我们直接运行会显示输出11 20。可是为什么会有这样的输出呢？在PHP的内部是如何实现的呢？ 这是本章将要说明的一个问题。下面我们从最基本的变量实现开始。

第一节 变量的结构

每门计算机语言都需要一些容器来保存变量数据。在一些语言当中，变量都有特定的类型，如字符串，数组，对象等等。比如C和Pascal就属于这种。 而PHP则没有这样的类型。在PHP中，一个变量在某一行是字符串，可能到下一行就变成了数字。 变量可以经常在不同的类型间轻易的转化，甚至是自动的转换。 PHP之所以成为一个简单并且强大的语言，很大一部分的原因是它拥有弱类型的变量。但是有些时候这也会带来一些问题。 在PHP内部，所有的变量都保存在zval结构中，也就是说，zval使用同一种结构存储了包括int、array、string等不同数据类型。 它不仅仅包含变量的值，也包含变量的类型。变量容器中包含一些Zend引擎用来区分是否引用的字段。同时它也包含这个值的引用计数。

那么，zval是如何做到的呢，下面我们一起来揭开面纱。

一.PHP变量类型在内核中的存储结构

PHP是一种弱类型的语言，这就意味着在声明或使用变量的时候，并不需要显式指明其数据类型。 但是，PHP是由C来实现的，大家都知道C对变量的类型管理是非常严格的，强类型的C是这样实现弱类型的PHP变量类型的:

1.在PHP中，存在8种变量类型，可以分为三类

- 标量类型： *boolean integer float(double) string*
- 复合类型： *array object*
- 特殊类型： *resource NULL*

在变量声明的开始，ZE判断用户变量的类型，并存入到以下zval结构体中。zval结构体定义在Zend/zend.h文件，其代码如下：

```
typedef struct _zval_struct zval;
...
struct _zval_struct {
    /* Variable information */
    zvalue_value value;     /* value */
    zend_uint refcount__gc;
    zend_uchar type;    /* active type */
    zend_uchar is_ref__gc;
};
```

2.初始化变量类型:

在上面的结构体中有四个值，其含义为：

属性名	含义	默认值
refcount__gc	表示引用计数	1
is_ref__gc	表示是否为引用	0
value	存储着变量的值信息	
type	记录变量的内部类型	

在PHP5.3之后，由于引入了垃圾收集机制，引用计数和是否为引用的属性名为refcount__gc和is_ref__gc。在此之前为refcount和is__ref。

在变量的初始化过程中，ZE会将变量的类型（type）值根据其变量类型置为：IS_NULL, IS_BOOL, IS_LONG, IS_DOUBLE, IS_STRING, IS_ARRAY, IS_OBJECT, IS_RESOURCE 之一。

PHP的实现中，如何判断变量是属于哪种类型的呢？(下节介绍)

二.变量的值在_zval_value中的存储

在上面大家不难发现，所有的php变量都是存储于zval结构中，其中变量值存储在zvalue_value联合体中：

```
typedef union _zvalue_value {
    long lval;                  /* long value */
    double dval;                /* double value */
    struct {
        char *val;
        int len;
```

```
    } str;
    HashTable *ht;              /* hash table value */
    zend_object_value obj;
} zvalue_value;
```

各种类型的数据会使用不同的方法来进行变量值的存储，其对应变量的赋值方式：

- 一般类型

变量类型	宏	
boolean integer	ZVAL_BOOL ZVAL_LONG	布尔型/整型的变量值存储于(zval).value.lval中,其类型也会以相应的IS_*进行存储。 `Z_TYPE_P(z)=IS_BOOL/LONG;  Z_LVAL_P(z)=((b)!=0);`
float□	ZVAL_DOUBLE	
null	ZVAL_NULL	NULL值的变量值不需要存储，只需要把(zval).type标为IS_NULL。 `Z_TYPE_P(z)=IS_NULL;`
resource	ZVAL_RESOURCE	资源类型的存储与其他一般变量无异，但其初始化及存取实现则不同。 `Z_TYPE_P(z) = IS_RESOURCE;  Z_LVAL_P(z) = l;`

- 字符串Sting

字符串类型的存储有别于上述一般类型，因为C中的字符串变量实际上是指向一个字符数组的头指针。所以，PHP在实现字符串变量时， 也采用指针的方式，在_zvalue_value数据结构中，存在下面的结构体内，其中*val就存储了指向字符串的指针，而len则存储了字符的长度。

```
struct {
    char *val;
    int len;
} str;
```

从这里可以看出strlen()函数是不会重新计算字符串长度的，只是返回str结构体中的len的值。

- 数组Array

数组是PHP中最常用，也是最强大变量类型，它可以存储其他类型的数据，而且提供各种内置操作函数。数组的存储相对于其他变量要复杂一些， 需要使用其它两种数据结构HashTable和Bucket。

```
typedef struct _hashtable {
    uint nTableSize;         // hash Bucket的大小,最小为8,以2x增长。
    uint nTableMask;         // nTableSize-1 ，  索引取值的优化
    uint nNumOfElements;     // hash Bucket中当前存在的元素个数, count()函数会直接返回此值
    ulong nNextFreeElement; // 标记hash Bucket当前索引数
    Bucket *pInternalPointer;    // 当前遍历的指针（foreach比for快的原因之一）
    Bucket *pListHead;           // 存储数组头元素指针
    Bucket *pListTail;           // 存储数组尾元素指针
```

```
    Bucket **arBuckets;           // 存储hash数组
    dtor_func_t pDestructor;
    zend_bool persistent;
    unsigned char nApplyCount; // 标记当前hash Bucket被递归访问的次数（防止多次递
归）
    zend_bool bApplyProtection;// 标记当前hash桶允许不允许多次访问，不允许时，最多只
能递归3次
#if ZEND_DEBUG
    int inconsistent;
#endif
} HashTable;

....

typedef struct bucket {
    ulong h;              //对char *key进行hash后的值,或者是用户指定的数字索引值
    uint nKeyLength;      //hash关键字的长度,如果数组索引为数字，此值为0
    void *pData;          //指向value，一般是用户数据的副本,如果是指针数据，则指向
pDataPtr
    void *pDataPtr;       //如果是指针数据,此值会指向真正的value,同时上面pData会指向此
值
    struct bucket *pListNext;    //整个hash表的下一元素
    struct bucket *pListLast;
    struct bucket *pNext;        //存放在同一个hash Bucket内的下一个元素
    struct bucket *pLast;
    char arKey[1];
    /*存储字符索引，此项必须放在最末尾，因为此处只字义了1个字节，存储的实际上是指向char
*key的值，
    这就意味着可以省去再赋值一次的消耗，而且，有时此值并不需要，所以同时还节省了空间。
    */
} Bucket;
```

从代码中不难发现，数组的存储是由_zval_struct ， _zvalue_value，HashTable，Bucket 共同完成的。上面的注释中标出了结构中的主要属性的作用。

- 对象Object

对象是一种复合型的数据，其需要存储较多元化的数据，如属性，方法，以及自身的一些性质。对象在PHP中是使用一种zend_object_value的结构体来存放。其代码如下：

```
typedef struct _zend_object_value {
    zend_object_handle handle;  //  unsigned int类型，是
    zend_object_handlers *handlers;
} zend_object_value;
```

handle字段是 EG(objects_store).object_buckets的索引，用来存取对应对象的相关数据。zend_object_handlers是一个包含许多方法指针的结构体。 关于这个结构体及对象相关的类的结构_zend_class_entry，将在第五章节作详细介绍。

哈希表(HashTable)

在PHP中字符串和数组使用最为频繁, PHP比较容易上手也得益于非常灵活的数组类型. 在开始详细介绍这些数据类型之前有必要介绍一下哈希表(HashTable). 哈希表是PHP实现中尤为关键的数据结构.

哈希表在实践中使用的非常广泛, 例如编译器通常会维护的一个符号表来保存标记, 很多高级语言中也

显式的支持哈希表, 哈希表通常提供查找(Search), 插入(Insert), 删除(Delete)等操作, 这些操作在最坏的情况下和链表的性能一样为O(n), 不过通常并不会这么坏, 合理设计的哈希算法能有效的避免这类情况, 通常哈希表的这些操作时间复杂度为O(1). 这也是它被钟爱的原因.

正是因为哈希表在使用上的便利性及效率上的表现, 目前大部分动态语言的实现中都使用了哈希表.

基本概念

为了方便读者阅读后面的内容, 这里提前列举一下HashTable中的基本概念. 哈希表是一种通过哈希函数,将特定的键映射到特定值的一种数据结构, 它维护键和值之间一一对应关系.

- 键(key): 用于操作数据的标示, 例如PHP数组中的索引, 或者字符串键等等.
- 槽(slot/bucket): 哈希表中用于保存数据的一个单元, 也就是数据真正存放的容器.
- 哈希函数(hash function): 将key映射(map)到数据应该存放的slot的索引的函数.
- 哈希冲突(hash collision): 哈希函数将两个不同的key映射到同一个索引的情况.

哈希表可以理解为一个数组的扩展或者关联数组, 数组使用数字下标来寻址, 如果关键字(key)的范围较小且是数字的话, 我们可以直接使用数组来完成哈希表, 而如果关键字范围太大, 如果直接使用数组我们需要为所有可能的key申请空间, 很多情况下这是不现实的. 即使空间足够, 空间利用率也会很低, 这并不理想. 同时键也可能并不是数字, 在PHP中尤为如此, 所以人们使用一种映射函数(哈希函数)来将key映射到特定的域中:

```
h(key) -> index
```

通过合理设计的哈希函数,我们就能将key映射到合适的范围, 因为我们的key空间可以很大(例如字符串key), 在映射到一个较小的空间中时可能会出现两个不同的key映射被到同一个index上的情况, 这就是我们所说的出现了冲突. 目前解决hash冲突的方法主要有两种: 链接法和开放寻址法.

冲突解决

链接法

链接法解决冲突的方式是使用一个链表来保存特定key所存放的值, 当不同的key映射到一个槽中的时候使用链表来保存这些值. 所以使用链接法是在最坏的情况下,也就是所有的key都映射到同一个槽中了, 而操作链表的时间复杂度为O(n). 所以选择一个合适的哈希函数是最为关键的.

目前PHP中HashTable的实现就是采用这种方式的.

开放寻址法

通常还有另外一种解决冲突的方法: 开放寻址法. 通常使用开放寻址法是槽本身直接存放数据, 在插入数据时如果key所映射到的索引已经有数据了, 这说明发生了冲突, 这是会寻找下一个槽, 如果该槽也被占用了则继续寻找下一个槽, 直到寻找到没有被占用的槽, 查找的时候也使用同样的策律.

哈希表的实现

在了解到哈希表的原理之后要实现一个哈希表也很容易, 主要需要完成的工作只有三点:

1. 实现哈希函数
2. 冲突的解决
3. 操作接口的实现

数据结构

首先我们需要一个容器来保存我们的哈希表, 哈希表需要保存的内容主要是保存进来的的数据, 同时为了方便的得知哈希表中存储的元素个数, 需要保存一个大小字段, 第二个需要的就是保存数据的容器了. 作为实例，下面将实现一个简易的哈希表。 基本的数据结构主要有两个，一个用于保存哈希表本身，另外一个就是用于实际保存数据的单链表了，定义如下:

```
typedef struct _Bucket
{
    char *key;
    void *value;
    struct _Bucket *next;

} Bucket;

typedef struct _HashTable
{
    int size;
    Bucket* buckets;
} HashTable;
```

上面的定义和PHP中的实现类似, 为了便于理解裁剪了大部分无关的细节, 在本节中为了简化, key的数据类型为字符串, 而存储的数据类型可以为任意类型.

Bucket结构体是一个单链表, 这是为了解决多个key哈希冲突的问题, 也就是前面所说的链接法. 当多个key映射到同一个index的时候将冲突的元素链接起来.

哈希函数实现

哈希函数需要尽可能的将不同的key映射到不同的槽(slot或者bucket)中, 首先我们采用一种最为简单的哈希算法实现: 将key字符串的所有字符加起来, 然后以结果对哈希表的大小取模，这样索引就能落在数组索引的范围之内了.

```
static int hash_str(char *key)
{
    int hash = 0;

    char *cur = key;

    while(*(cur++) != '\0') {
```

```
        hash += *cur;
    }

    return hash;
}

// 使用这个宏来求得key在哈希表中的索引
#define HASH_INDEX(ht, key) (hash_str((key)) % (ht)->size)
```

这个哈希算法比较简单, 例如PHP中使用的是称为DJBX33A算法, 这里列举了Mysql, OpenSSL等开源软件使用的哈希算法, 有兴趣的读者可以前往参考。

操作接口的实现

为了操作哈希表，实现了如下几个操作函数:

```
int hash_init(HashTable *ht);                                   // 初始化哈希表
int hash_lookup(HashTable *ht, char *key, void **result);       // 根据key查找内容
int hash_insert(HashTable *ht, char *key, void *value);         // 将内容插入到哈希表
中
int hash_remove(HashTable *ht, char *key);                      // 删除key所指向的内
容
int hash_destroy(HashTable *ht);
```

下面以插入和获取操作函数为例:

```
int hash_insert(HashTable *ht, char *key, void *value)
{
    // check if we need to resize the hashtable
    resize_hash_table_if_needed(ht);    // 哈希表不固定大小，当插入的内容快占满哈表
的存储空间
                                        // 将对哈希表进行扩容，以便容纳所有的元素

    int index = HASH_INDEX(ht, key);    // 找到key所映射到的索引

    Bucket *org_bucket = ht->buckets[index];
    Bucket *bucket = (Bucket *)malloc(sizeof(Bucket)); // 为新元素申请空间

    bucket->key   = strdup(key);
    // 将值内容保存进来， 这里只是简单的将指针指向要存储的内容，而没有将内容复制。
    bucket->value = value;

    LOG_MSG("Insert data p: %p\n", value);

    ht->elem_num += 1; // 记录一下现在哈希表中的元素个数

    if(org_bucket != NULL) { // 发生了碰撞，将新元素放置在链表的头部
        LOG_MSG("Index collision found with org hashtable: %p\n", org_bucket);
        bucket->next = org_bucket;
    }

    ht->buckets[index]= bucket;

    LOG_MSG("Element inserted at index %i, now we have: %i elements\n",
        index, ht->elem_num);

    return SUCCESS;
```

```
}
```

上面这个哈希表的插入操作比较简单，简单的以key做哈希，找到元素应该存储的位置，并检查该位置是否已经有了内容， 如果发生碰撞则将新元素链接到原有元素链表头部。在查找时也按照同样的策略，找到元素所在的位置，如果存在元素， 则将该链表的所有元素的key和要查找的key依次对比，直到找到一致的元素，否则说明该值没有匹配的内容。

```
int hash_lookup(HashTable *ht, char *key, void **result)
{
    int index = HASH_INDEX(ht, key);
    Bucket *bucket = ht->buckets[index];

    if(bucket == NULL) return FAILED;

    // 查找这个链表以便找到正确的元素，通常这个链表应该是只有一个元素的，也就不用多次
    // 循环。要保证这一点需要有一个合适的哈希算法，见前面相关哈希函数的链接。
    while(bucket)
    {
        if(strcmp(bucket->key, key) == 0)
        {
            LOG_MSG("HashTable found key in index: %i with  key: %s value:
%p\n",
                index, key, bucket->value);
            *result = bucket->value;
            return SUCCESS;
        }

        bucket = bucket->next;
    }

    LOG_MSG("HashTable lookup missed the key: %s\n", key);
    return FAILED;
}
```

我们都知道在PHP中数组是基于哈希表的，而在依次给数组添加元素时，元素之间是有先后顺序的，而这里的哈希表在物理位置上显然是接近平均分布的， 这样怎么能根据插入的先后顺序获取到这些元素的，其实在PHP的实现中Bucket结构体还维护了另一个指针字段来维护元素之间的关系。具体内容在后面的小节中说明。 上面的例子就是PHP中实现的一个精简版。

完整代码可以在$TIPI_ROOT/book/sample/chapt03/03-01-01-hashtable目录中找到.

PHP中的哈希表

PHP的哈希实现

PHP哈希标的优化(参考PHP core的改进)

其他语言中的HashTable实现

第二节 常量

常量，顾名思义是一个常态的量值。它与值只绑定一次，它的作用在于有肋于增加程序的可读性和可靠性。 在PHP中，常量的名字是一个简单值的标识符，在脚本执行期间该值不能改变。 和变量一样，常量默认为大小写敏感，但是按照我们的习惯常量标识符总是大写的。 常量名和其它任何 PHP 标签遵循同样的命名规则。合法的常量名以字母或下划线开始，后面跟着任何字母，数字或下划线。 在这一小节我们一起看下常量与我们常见的变量有啥区别，它在执行期间的不可改变的特性是如何实现的以及常量的定义过程。

首先看下常量与变量的区别，常量是在变量的zval结构的基础上添加了一额外的元素。如下所示为PHP中常量的内部结构。

常量的内部结构

```
typedef struct _zend_constant {
    zval value; /* zval结构，PHP内部变量的存储结构，在第一小节有说明 */
    int flags;  /* 常量的标记如 CONST_PERSISTENT | CONST_CS */
    char *name; /* 常量名称 */
    uint name_len;
    int module_number;  /* 模块号 */
} zend_constant;
```

在Zend/zend_constants.h文件的33行可以看到如上所示的结构定义。 在常量的结构中，除了与变量一样的zval结构，它还包括属于常量的标记，常量名以及常量所在的模块号。

在了解了常量的存储结构后，我们来看PHP常量的定义过程。一个例子。

```
define('TIPI', 'Thinking In PHP Internal');
```

这是一个很常规的常量定义过程，它使用了PHP的内置函数**define**。常量名为TIPI，值为一个字符串，存放在zval结构中。 从这个例子出发，我们看下define定义常量的过程实现。

define定义常量的过程

define是PHP的内置函数，在Zend/zend_builtin_functions.c文件中定义了此函数的实现。如下所示为部分源码：

```
/* {{{ proto bool define(string constant_name, mixed value, boolean
case_insensitive=false)
   Define a new constant */
ZEND_FUNCTION(define)
{
        if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sz|b", &name,
                &name_len, &val, &non_cs) == FAILURE) {
```

```
                return;
        }

        ... // 类常量定义 此处不做介绍

        ... // 值类型判断和处理

        c.value = *val;
        zval_copy_ctor(&c.value);
        if (val_free) {
                zval_ptr_dtor(&val_free);
        }
        c.flags = case_sensitive; /* non persistent */
        c.name = zend_strndup(name, name_len);
        c.name_len = name_len+1;
        c.module_number = PHP_USER_CONSTANT;
        if (zend_register_constant(&c TSRMLS_CC) == SUCCESS) {
                RETURN_TRUE;
        } else {
                RETURN_FALSE;
        }
}
/* }}} */
```

上面的代码已经对对象和类常量做了简化处理， 其实现基本上是一个将传递的参数传递给新建的zend_constant结构，并将这个结构体注册到常量列表中的过程。 关于大小写敏感，函数的第三个参数表示是否**大小不敏感**，默认为false（大小写敏感）。这个参数最后会赋值给zend_constant结构体的flags字段。其在函数中实现代码如下：

```
zend_bool non_cs = 0;    //  第三个参数的临时存储变量
int case_sensitive = CONST_CS;  //  是否大小写敏感，默认为1

if(non_cs) {    //  输入为真，大小写不敏感
    case_sensitive = 0;
}

c.flags = case_sensitive; //      赋值给结构体字段
```

从上面的define函数的实现来看，**PHP对于常量的名称在定义时其实是没有所谓的限制。**如下所示代码：

```
define('^_^', 'smile');

if (defined('^_^')) {
    echo 'yes';
}else{
    echo 'no';
}
```

通过defined函数测试表示，‘^_^’这个常量已经定义好，只是这样的常量无法调用。 因为在作为语法解析时会显示错误。在上面的代码中有用到一个判断常量是否定义的函数，下面我们看看这个函数是如何实现的。

defined判断常量是否设置

和define一样， defined的实现也在Zend/zend_builtin_functions.c文件， 其实现是一个读取参数变量，调用 zend_get_constant_ex函数获取常量的值来判断常量是否存在的过程。 而zend_get_constant_ex函数不仅包括了常规的常规的常量获取，还包括类常量的获取， 最后是通过zend_get_constant函数获取常量的值。在zend_get_constant函数中，基本上是通过下面的代码来获取常量的值。

```
zend_hash_find(EG(zend_constants), name, name_len+1, (void **) &c)
```

除此之外，只是调用这个函数之前和之后对name有一些特殊的处理。

标准常量的初始化

以上通过define定义的常量的模块编号都是PHP_USER_CONSTANT，这表示是用户定义的常量。 除此之外我们在平时使用较多的，如在显示所有级别错误报告时使用的E_ALL常量就有点不同了。 这里我们以cgi模式为例说明标准常量的定义过程。 整个调用顺序如下所示：
[php_cgi_startup() -> php_module_startup() -> zend_startup() -> zend_register_standard_constants()]

```
void zend_register_standard_constants(TSRMLS_D)
{
    ... //  若干常量以REGISTER_MAIN_LONG_CONSTANT设置,
    REGISTER_MAIN_LONG_CONSTANT("E_ALL", E_ALL, CONST_PERSISTENT | CONST_CS);
    ...
}
```

REGISTER_MAIN_LONG_CONSTANT宏展开是以zend_register_long_constant实现。zend_register_long_constant函数将常量中值的类型，值，名称及模块号赋值给新的zend_constant。 并调用zend_register_constant添加到全局的常量列表中。

[php_cgi_startup() -> php_module_startup() -> zend_startup() -> zend_register_standard_constants() -> zend_register_constant]

```
ZEND_API void zend_register_long_constant(const char *name, uint name_len,
        long lval, int flags, int module_number TSRMLS_DC)
{
    zend_constant c;

    c.value.type = IS_LONG;
    c.value.value.lval = lval;
    c.flags = flags;
    c.name = zend_strndup(name, name_len-1);
    c.name_len = name_len;
    c.module_number = module_number;
    zend_register_constant(&c TSRMLS_CC);
}
```

zend_register_constant函数首先根据常量中的c->flags判断是否区分大小写，如果不区分，则名字统一为小写，如果包含"\\"，也统一成小写。否则为定义的名字 然后将调用下面的语句将当前常量添加到EG(zend_constants)。EG(zend_constants)是一个HashTable（这在前面的章节中说明），下面的代码是

将常量添加到这个HashTable中。

```
zend_hash_add(EG(zend_constants), name, c->name_len, (void *) c,
        sizeof(zend_constant), NULL)==FAILURE)
```

在php_module_startup函数中，除了zend_startup函数中有注册标准的常量，它本身体通过宏REGISTER_MAIN_LONG_CONSTANT等注册了一些常量，如：PHP_VERSION，PHP_OS等。

关于接口和类中的常量我们将在后面的类所在章节中详细说明。

第三节 预定义变量

大家都知道PHP脚本在执行的时候用户全局变量(在用户空间显式定义的变量)会保存在一个HashTable数据类型的符号表(symbol_table)中, 在PHP中有一些比较特殊的全局变量例如: $_GET, $_POST, $_SERVER等变量, 我们自己并没有定义这样的一些变量, 那这些变量是从何而来的呢? 既然变量是保存在符号表中, 那PHP应该是在脚本运行之前就将这些特殊的变量加入到了符号表中了吧? 事实就是这样.

预定义变量$GLOBALS的初始化

我们以cgi模式为例说明$GLOBALS的初始化。 从cgi_main.c文件main函数开始。 整个调用顺序如下所示：

[main() -> php_request_startup() -> zend_activate() -> init_executor()]

```
... //  省略
zend_hash_init(&EG(symbol_table), 50, NULL, ZVAL_PTR_DTOR, 0);
{
    zval *globals;

    ALLOC_ZVAL(globals);
    Z_SET_REFCOUNT_P(globals, 1);
    Z_SET_ISREF_P(globals);
    Z_TYPE_P(globals) = IS_ARRAY;
    Z_ARRVAL_P(globals) = &EG(symbol_table);
    zend_hash_update(&EG(symbol_table), "GLOBALS", sizeof("GLOBALS"),
        &globals, sizeof(zval *), NULL);       //  添加全局变量GLOBALS
}
... //  省略
```

上面的代码的关键点zend_hash_update函数的调用，它将变量名为GLOBALS的变量注册到EG(symbol_table)中，EG(symbol_table)是一个HashTable的结构，用来存放所有的全局变量。 这在下面将要提到的$_GET等变量初始化时也会用到。

$_GET、$_POST等变量的初始化

$_GET、$_COOKIE、$_SERVER、$_ENV、$_FILES、$_REQUEST这六个变量都是通过如下的调用序列进行初始化。 **[main() -> php_request_startup() -> php_hash_environment()]**
在请求初始化时，通过调用 **php_hash_environment** 函数初始化以上的六个预定义的变量。如下所示为

php_hash_environment函数的代码。在代码之后我们以$_POST为例说明整个初始化的过程。

```
/* {{{ php_hash_environment
 */
int php_hash_environment(TSRMLS_D)
{
        char *p;
        unsigned char _gpc_flags[5] = {0, 0, 0, 0, 0};
        zend_bool jit_initialization = (PG(auto_globals_jit) &&
!PG(register_globals) && !PG(register_long_arrays));
        struct auto_global_record {
                char *name;
                uint name_len;
                char *long_name;
                uint long_name_len;
                zend_bool jit_initialization;
        } auto_global_records[] = {
                { "_POST", sizeof("_POST"), "HTTP_POST_VARS",
sizeof("HTTP_POST_VARS"), 0 },
                { "_GET", sizeof("_GET"), "HTTP_GET_VARS",
sizeof("HTTP_GET_VARS"), 0 },
                { "_COOKIE", sizeof("_COOKIE"), "HTTP_COOKIE_VARS",
sizeof("HTTP_COOKIE_VARS"), 0 },
                { "_SERVER", sizeof("_SERVER"), "HTTP_SERVER_VARS",
sizeof("HTTP_SERVER_VARS"), 1 },
                { "_ENV", sizeof("_ENV"), "HTTP_ENV_VARS",
sizeof("HTTP_ENV_VARS"), 1 },
                { "_FILES", sizeof("_FILES"), "HTTP_POST_FILES",
sizeof("HTTP_POST_FILES"), 0 },
        };
        size_t num_track_vars = sizeof(auto_global_records)/sizeof(struct
auto_global_record);
        size_t i;

        /* jit_initialization = 0; */
        for (i=0; i<num_track_vars; i++) {
                PG(http_globals)[i] = NULL;
        }

        for (p=PG(variables_order); p && *p; p++) {
                switch(*p) {
                        case 'p':
                        case 'P':
                                if (!_gpc_flags[0] && !SG(headers_sent) &&
SG(request_info).request_method && !strcasecmp(SG(request_info).request_method,
"POST")) {
                                        sapi_module.treat_data(PARSE_POST,
NULL, NULL TSRMLS_CC);   /* POST Data */
                                        _gpc_flags[0] = 1;
                                        if (PG(register_globals)) {

php_autoglobal_merge(&EG(symbol_table),
Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_POST]) TSRMLS_CC);
                                        }
                                }
                                break;
                        case 'c':
                        case 'C':
                                if (!_gpc_flags[1]) {
                                        sapi_module.treat_data(PARSE_COOKIE,
NULL, NULL TSRMLS_CC); /* Cookie Data */
                                        _gpc_flags[1] = 1;
                                        if (PG(register_globals)) {
```

```
php_autoglobal_merge(&EG(symbol_table),
Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_COOKIE]) TSRMLS_CC);
                                        }
                                }
                                break;
                        case 'g':
                        case 'G':
                                if (!_gpc_flags[2]) {
                                        sapi_module.treat_data(PARSE_GET, NULL,
NULL TSRMLS_CC);    /* GET Data */
                                        _gpc_flags[2] = 1;
                                        if (PG(register_globals)) {

php_autoglobal_merge(&EG(symbol_table),
Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_GET]) TSRMLS_CC);
                                        }
                                }
                                break;
                        case 'e':
                        case 'E':
                                if (!jit_initialization && !_gpc_flags[3]) {
                                        zend_auto_global_disable_jit("_ENV",
sizeof("_ENV")-1 TSRMLS_CC);
                                        php_auto_globals_create_env("_ENV",
sizeof("_ENV")-1 TSRMLS_CC);
                                        _gpc_flags[3] = 1;
                                        if (PG(register_globals)) {

php_autoglobal_merge(&EG(symbol_table),
Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_ENV]) TSRMLS_CC);
                                        }
                                }
                                break;
                        case 's':
                        case 'S':
                                if (!jit_initialization && !_gpc_flags[4]) {
                                        zend_auto_global_disable_jit("_SERVER",
sizeof("_SERVER")-1 TSRMLS_CC);

php_register_server_variables(TSRMLS_C);
                                        _gpc_flags[4] = 1;
                                        if (PG(register_globals)) {

php_autoglobal_merge(&EG(symbol_table),
Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_SERVER]) TSRMLS_CC);
                                        }
                                }
                                break;
                }
        }

        /* argv/argc support */
        if (PG(register_argc_argv)) {
                php_build_argv(SG(request_info).query_string,
PG(http_globals)[TRACK_VARS_SERVER] TSRMLS_CC);
        }

        for (i=0; i<num_track_vars; i++) {
                if (jit_initialization &&
auto_global_records[i].jit_initialization) {
                        continue;
                }
                if (!PG(http_globals)[i]) {
```

```
                        ALLOC_ZVAL(PG(http_globals)[i]);
                        array_init(PG(http_globals)[i]);
                        INIT_PZVAL(PG(http_globals)[i]);
                }

                Z_ADDREF_P(PG(http_globals)[i]);
                zend_hash_update(&EG(symbol_table),
auto_global_records[i].name, auto_global_records[i].name_len,
&PG(http_globals)[i], sizeof(zval *), NULL);
                if (PG(register_long_arrays)) {
                        zend_hash_update(&EG(symbol_table),
auto_global_records[i].long_name, auto_global_records[i].long_name_len,
&PG(http_globals)[i], sizeof(zval *), NULL);
                        Z_ADDREF_P(PG(http_globals)[i]);
                }
        }

        /* Create _REQUEST */
        if (!jit_initialization) {
                zend_auto_global_disable_jit("_REQUEST", sizeof("_REQUEST")-1
TSRMLS_CC);
                php_auto_globals_create_request("_REQUEST",
sizeof("_REQUEST")-1 TSRMLS_CC);
        }

        return SUCCESS;
}
```

以$_POST为例，首先以 **auto_global_record** 数组形式定义好将要初始化的变量的相关信息。 在变量初始化完成后，按照PG(variables_order)指定的顺序（在php.ini中指定），通过调用sapi_module.treat_data处理数据。

> 从PHP实现的架构设计看，treat_data函数在SAPI目录下不同的服务器应该有不同的实现，只是现在大部分都是使用的默认实现。

在treat_data后，如果打开了PG(register_globals)，则会调用php_autoglobal_merge将相关变量的值写到符号表。

以上的所有数据处理是一个赋值前的初始化行为。在此之后，通过遍历之前定义的结构体，调用zend_hash_update，将相关变量的值赋值给&EG(symbol_table)。 另外对于$_REQUEST有独立的处理方法。

第四节 静态变量

通常意义上静态变量是静态分配的，他们的生命周期和程序的生命周期一样，只有在程序退出时才结束期生命周期， 这和局部变量相反，有的语言中全局变量也是静态分配的。例如PHP中的全局变量以及Javascript中的全局变量。

静态变量可以分为：

- 静态全局变量，PHP中的全局变量也可以理解为静态全局变量
- 静态局部变量，也就是在函数内定义的静态变量，函数在执行时对变量的操作会保持到下一次函数被调用。
- 静态成员变量，这主要是在类中定义的静态变量，和实例变量相对应，静态成员变量可以在所有实

例中共享。

我们常见的应该是静态局部变量。局部变量只有在函数执行时才会存在。 通常，当一个函数执行完毕，它的局部变量的值就已经不存在，而且变量所占据的内存也被释放。 当下一次执行该过程时，它的所有局部变量将重新初始化。当把局部变量定义成静态的，从而保留变量的值。 在函数内部用 static 关键字声明一个或多个变量，其用法如下：

```
function t() {
    static $i = 0;
    $i++;
    echo $i, ' ';
}

t();
t();
t();
```

上面的程序会输出1 2 3。从这个示例可以看出，$i变量是独立于其它局部变量的。那么这个独立性是如何实现的，下面我们一起来探索静态变量的实现过程。

static是PHP语言的一个语句，我们需要从词法分析，语法分析，中间代码生成到执行中间代码这几个部分探讨整个实现过程。

1. 词法分析

首先查看 Zend/zend_language_scanner.l文件，搜索 static关键字。我们可以找到如下代码：

```
<ST_IN_SCRIPTING>"static" {
    return T_STATIC;
}
```

2. 语法分析

在词法分析找到token后，通过这个token，在Zend/zend_language_parser.y文件中查找。找到相关代码如下：

```
|   T_STATIC static_var_list ';'

static_var_list:
        static_var_list ',' T_VARIABLE { zend_do_fetch_static_variable(&$3,
NULL, ZEND_FETCH_STATIC TSRMLS_CC); }
    |   static_var_list ',' T_VARIABLE '=' static_scalar {
zend_do_fetch_static_variable(&$3, &$5, ZEND_FETCH_STATIC TSRMLS_CC); }
    |   T_VARIABLE  { zend_do_fetch_static_variable(&$1, NULL,
ZEND_FETCH_STATIC TSRMLS_CC); }
    |   T_VARIABLE '=' static_scalar { zend_do_fetch_static_variable(&$1, &$3,
ZEND_FETCH_STATIC TSRMLS_CC); }

;
```

从上面代码可知，PHP在解释static变量赋值生成中间是调用zend_do_fetch_static_variable函数。

3. 生成中间代码

调用zend_do_fetch_static_variable函数其实是生成中间代码的过程。其代码如下：

```
void zend_do_fetch_static_variable(znode *varname, const znode
        *static_assignment, int fetch_type TSRMLS_DC)
{
    zval *tmp;
    zend_op *opline;
    znode lval;
    znode result;

    ALLOC_ZVAL(tmp);


    if (static_assignment) {
        *tmp = static_assignment->u.constant;
    } else {
        INIT_ZVAL(*tmp);
    }
    if (!CG(active_op_array)->static_variables) {    /* 初始化此时的静态变量存放位
置 */
        ALLOC_HASHTABLE(CG(active_op_array)->static_variables);
        zend_hash_init(CG(active_op_array)->static_variables, 2, NULL,
ZVAL_PTR_DTOR, 0);
    }
    //  将新的静态变量放进来
    zend_hash_update(CG(active_op_array)->static_variables, varname-
>u.constant.value.str.val,
        varname->u.constant.value.str.len+1, &tmp, sizeof(zval *), NULL);

    ...//省略
    opline = get_next_op(CG(active_op_array) TSRMLS_CC);
    opline->opcode = (fetch_type == ZEND_FETCH_LEXICAL) ? ZEND_FETCH_R :
ZEND_FETCH_W;       /* 由于fetch_type=ZEND_FETCH_STATIC，程序会选择ZEND_FETCH_W*/
    opline->result.op_type = IS_VAR;
    opline->result.u.EA.type = 0;
    opline->result.u.var = get_temporary_variable(CG(active_op_array));
    opline->op1 = *varname;
    SET_UNUSED(opline->op2);
    opline->op2.u.EA.type = ZEND_FETCH_STATIC;  /* 这在中间代码执行时会有大用 */
    result = opline->result;

    if (varname->op_type == IS_CONST) {
        zval_copy_ctor(&varname->u.constant);
    }
    fetch_simple_variable(&lval, varname, 0 TSRMLS_CC); /* Relies on the fact
that the default fetch is BP_VAR_W */

    if (fetch_type == ZEND_FETCH_LEXICAL) {
        ...//省略
    } else {
        zend_do_assign_ref(NULL, &lval, &result TSRMLS_CC); //  赋值操作中间代码
生成
    }
    CG(active_op_array)->opcodes[CG(active_op_array)->last-1].result.u.EA.type
|= EXT_TYPE_UNUSED;

}
```

从上面的代码我们可知，在解释成中间代码时，静态变量是存放在CG(active_op_array)->static_variables中的。 并且生成的中间代码为：**ZEND_FETCH_W** 和 **ZEND_ASSIGN_REF** 。 其中ZEND_FETCH_W中间代码是在zend_do_fetch_static_variable中直接赋值，而ZEND_ASSIGN_REF中间代码是在zend_do_fetch_static_variable中调用zend_do_assign_ref生成的。

4. 执行中间代码

在生成完中间代码后，ZE会调用执行中间代码。而中间代码会先查看中间代码的编号。在Zend/zend_vm_opcodes.h文件中，这两个的定义如下：

```
#define ZEND_FETCH_W                            83
#define ZEND_ASSIGN_REF                         39
```

通过中间代码调用映射方法计算得此时ZEND_FETCH_W 对应的操作为ZEND_FETCH_W_SPEC_CV_HANDLER。其代码如下：

```
static int ZEND_FASTCALL
ZEND_FETCH_W_SPEC_CV_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    return zend_fetch_var_address_helper_SPEC_CV(BP_VAR_W,
ZEND_OPCODE_HANDLER_ARGS_PASSTHRU);
}

static int ZEND_FASTCALL zend_fetch_var_address_helper_SPEC_CV(int type,
ZEND_OPCODE_HANDLER_ARGS)
{
    ...//省略

    if (opline->op2.u.EA.type == ZEND_FETCH_STATIC_MEMBER) {
        retval = zend_std_get_static_property(EX_T(opline-
>op2.u.var).class_entry, Z_STRVAL_P(varname), Z_STRLEN_P(varname), 0
TSRMLS_CC);
    } else {
        target_symbol_table = zend_get_target_symbol_table(opline, EX(Ts),
type, varname TSRMLS_CC);    // 取符号表，这里我们取的是EG(active_op_array)-
>static_variables
        ...//   省略
        if (zend_hash_find(target_symbol_table, varname->value.str.val,
varname->value.str.len+1, (void **) &retval) == FAILURE) {
            switch (type) {
                ...//省略
                //  在前面的调用中我们知道type = case BP_VAR_W, 于是程序会走按case
BP_VAR_W的流程走。
                case BP_VAR_W: {
                        zval *new_zval = &EG(uninitialized_zval);

                        Z_ADDREF_P(new_zval);
                        zend_hash_update(target_symbol_table, varname-
>value.str.val, varname->value.str.len+1, &new_zval, sizeof(zval *), (void **)
&retval);
                        // 更新符号表，执行赋值操作
                    }
                    break;
                EMPTY_SWITCH_DEFAULT_CASE()
            }
        }
```

```
        switch (opline->op2.u.EA.type) {
            ...//省略
            case ZEND_FETCH_STATIC:
                zval_update_constant(retval, (void*) 1 TSRMLS_CC);
                break;
            case ZEND_FETCH_GLOBAL_LOCK:
                if (IS_CV == IS_VAR && !free_op1.var) {
                    PZVAL_LOCK(*EX_T(opline->op1.u.var).var.ptr_ptr);
                }
                break;
        }
    }

    ...//省略
}
```

在上面的代码中有一个关键的函数zend_get_target_symbol_table。它的作用是取目标符号表，如下为本次调用的部分代码实现。

```
static inline HashTable *zend_get_target_symbol_table(const zend_op *opline,
const temp_variable *Ts, int type, const zval *variable TSRMLS_DC)
{
    switch (opline->op2.u.EA.type) {
        ...//   省略
        case ZEND_FETCH_STATIC:
            if (!EG(active_op_array)->static_variables) {
                ALLOC_HASHTABLE(EG(active_op_array)->static_variables);
                zend_hash_init(EG(active_op_array)->static_variables, 2, NULL,
ZVAL_PTR_DTOR, 0);
            }
            return EG(active_op_array)->static_variables;
            break;
        EMPTY_SWITCH_DEFAULT_CASE()
    }
    return NULL;
}
```

在前面的zend_do_fetch_static_variable执行时，op2.u.EA.type的值为ZEND_FETCH_STATIC， 从而这zend_get_target_symbol_table函数中我们取EG(active_op_array)->static_variables的值返回。

从上面的实现可以看出静态变量在生成中间代码以及在执行时都是以一个HashTable类型的变量static_variables独立存在，这就是静态变量与其它变量不同所在。

第五节 类型提示的实现

PHP是弱类型语言，向方法传递参数时候一般也不太区分数据类型。 但是有时需要判断传递到方法中的参数，为此，PHP中提供了一些函数，来判断数据的类型。 比如is_numeric(),判断是否是一个数值或者可转换为数值的字符串，比如用于判断对象的类型运算符：instanceof。 instanceof 用来测定一个给定的对象是否来自指定的对象类。instanceof 运算符是 PHP 5 引进的。在此之前是使用的is_a()，不过现在已经不推荐使用。

为了避免对象类型不规范引起的问题，PHP5中引入了类型提示这个概念。在定义方法参数时，同时定义参数的对象类型。 如果在调用的时候，传入参数的类型与定义的参数类型不符，则会报错。这样就可以过滤对象的类型，或者说保证了数据的安全性。

PHP中的类型提示功能只能用于参数为对象的提示，而无法用于为整数，字串，浮点等类型提示。在PHP5.1之后，PHP支持对数组的类型提示。

要达到类型提示，只要在方法的对象型参数前加一个已存在的类的名称，当使用类型提示时，你不仅可以指定对象类型，还可以指定抽象类和接口。

下面我们从PHP的整个解释过程分析类型提示的源码实现过程。我们将从function开始词法分析， 在语法分析中找到类型提示的相关代码，然后跟踪到中间代码的实现。

由于类型提示符是作用于类的方法或函数，则其实现一定与function相关，因此我们可以从function语句起查找类型提示的实现。 为function在 Zend/zend_language_scanner.l文件中，我们可以找到function对应的token为 **T_FUNCTION** 。

在词法解析完成后，在Zend/zend_language_parser.y文件中查找T_FUNCTION，并查找对应的参数列表。 从整个过程我们可以看到关于类型提示的检测最后调用的是：

```
zend_do_receive_arg(ZEND_RECV, &tmp, &$$, NULL, &$1, &$2, 0 TSRMLS_CC);
```

在这个函数中其opcode被赋值为ZEND_RECV。根据opcode的映射计算规则得出其在执行时调用的是ZEND_RECV_SPEC_HANDLER。其代码如下：

```
static int ZEND_FASTCALL  ZEND_RECV_SPEC_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
        ...//省略
         if (param == NULL) {
                char *space;
                char *class_name = get_active_class_name(&space TSRMLS_CC);
                zend_execute_data *ptr = EX(prev_execute_data);

                if (zend_verify_arg_type((zend_function *) EG(active_op_array),
arg_num, NULL, opline->extended_value TSRMLS_CC)) {
                        ...//省略
                }
               ...//省略
         } else {
              ...//省略
                zend_verify_arg_type((zend_function *) EG(active_op_array),
arg_num, *param, opline->extended_value TSRMLS_CC);
              ...//省略
         }
       ...//省略
}
```

如上所示：在ZEND_RECV_SPEC_HANDLER中最后调用的是zend_verify_arg_type。其代码如下：

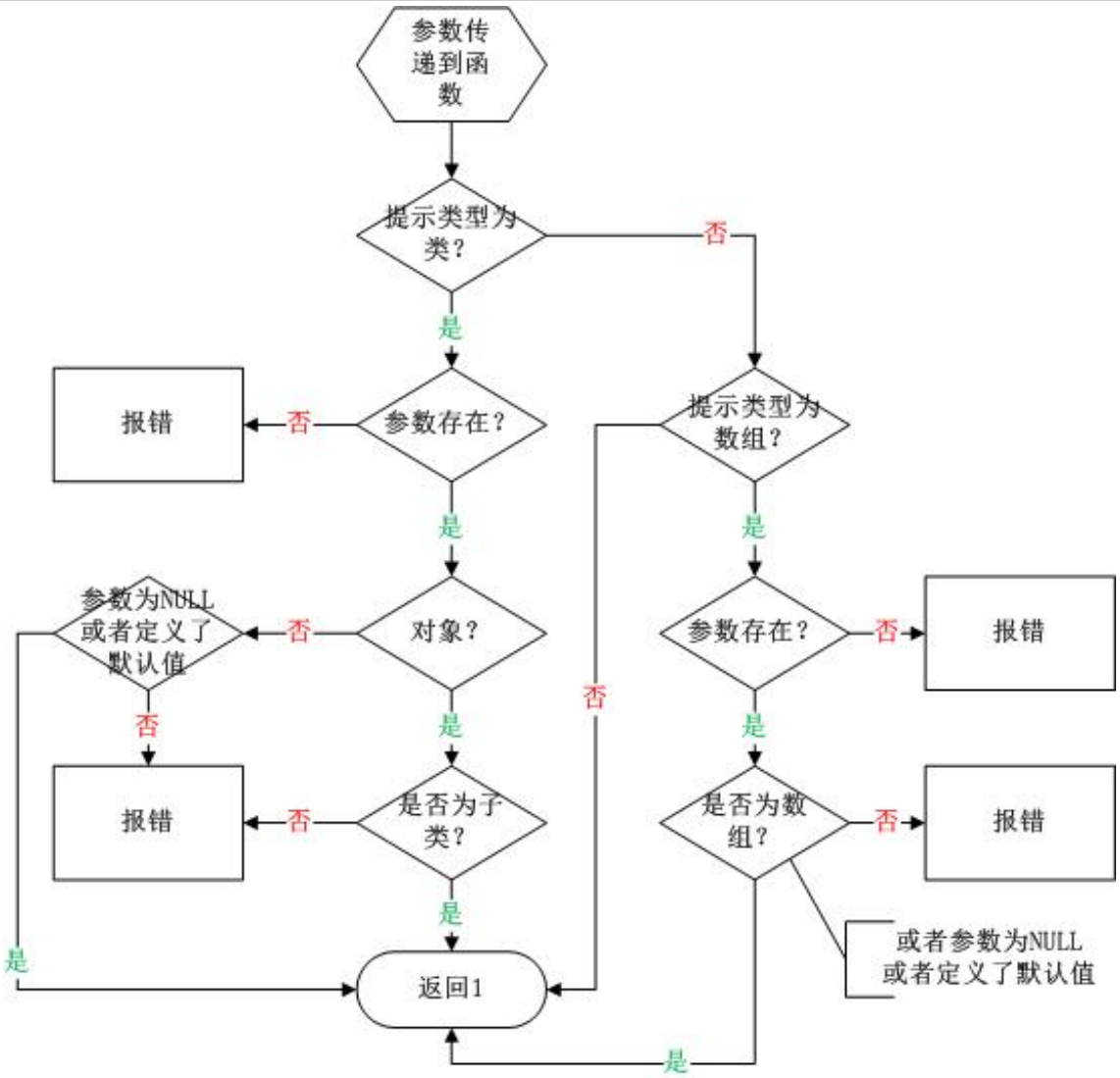
```
static inline int zend_verify_arg_type(zend_function *zf, zend_uint arg_num,
zval *arg, ulong fetch_type TSRMLS_DC)
{
    ...//省略

     if (cur_arg_info->class_name) {
         const char *class_name;

         if (!arg) {
             need_msg = zend_verify_arg_class_kind(cur_arg_info, fetch_type,
```

```
&class_name, &ce TSRMLS_CC);
            return zend_verify_arg_error(zf, arg_num, cur_arg_info, need_msg,
class_name, "none", "" TSRMLS_CC);
        }
        if (Z_TYPE_P(arg) == IS_OBJECT) { // 既然是类对象参数，传递的参数需要是对象
类型
            // 下面检查这个对象是否是参数提示类的实例对象，这里是允许传递子类实力对象
            need_msg = zend_verify_arg_class_kind(cur_arg_info, fetch_type,
&class_name, &ce TSRMLS_CC);
            if (!ce || !instanceof_function(Z_OBJCE_P(arg), ce TSRMLS_CC)) {
                return zend_verify_arg_error(zf, arg_num, cur_arg_info,
need_msg, class_name, "instance of ", Z_OBJCE_P(arg)->name TSRMLS_CC);
            }
        } else if (Z_TYPE_P(arg) != IS_NULL || !cur_arg_info->allow_null) { //
参数为NULL，也是可以通过检查的，
                                                                            //
如果函数定义了参数默认值，不传递参数调用也是可以通过检查的
            need_msg = zend_verify_arg_class_kind(cur_arg_info, fetch_type,
&class_name, &ce TSRMLS_CC);
            return zend_verify_arg_error(zf, arg_num, cur_arg_info, need_msg,
class_name, zend_zval_type_name(arg), "" TSRMLS_CC);
        }
    } else if (cur_arg_info->array_type_hint) {
        if (!arg) {
            return zend_verify_arg_error(zf, arg_num, cur_arg_info, "be an
array", "", "none", "" TSRMLS_CC);
        }
        if (Z_TYPE_P(arg) != IS_ARRAY && (Z_TYPE_P(arg) != IS_NULL ||
!cur_arg_info->allow_null)) {
            return zend_verify_arg_error(zf, arg_num, cur_arg_info, "be an
array", "", zend_zval_type_name(arg), "" TSRMLS_CC);
        }
    }
    return 1;
}
```

zend_verify_arg_type的整个流程如图3.1所示：



如果类型提示是类名，则判断传递进来的参数是否为对象，如果传递进来的是对象，则判断 如果类型提示报错，zend_verify_arg_type函数最后都会调用 zend_verify_arg_class_kind 生成报错信息，并且调用 zend_verify_arg_error 报错。如下所示代码：

```
static inline char * zend_verify_arg_class_kind(const zend_arg_info
*cur_arg_info, ulong fetch_type, const char **class_name, zend_class_entry
**pce TSRMLS_DC)
{
    *pce = zend_fetch_class(cur_arg_info->class_name, cur_arg_info-
>class_name_len, (fetch_type | ZEND_FETCH_CLASS_AUTO |
ZEND_FETCH_CLASS_NO_AUTOLOAD) TSRMLS_CC);

    *class_name = (*pce) ? (*pce)->name: cur_arg_info->class_name;
    if (*pce && (*pce)->ce_flags & ZEND_ACC_INTERFACE) {
        return "implement interface ";
    } else {
        return "be an instance of ";
    }
}


static inline int zend_verify_arg_error(const zend_function *zf, zend_uint
arg_num, const zend_arg_info *cur_arg_info, const char *need_msg, const char
*need_kind, const char *given_msg, char *given_kind TSRMLS_DC)
{
    zend_execute_data *ptr = EG(current_execute_data)->prev_execute_data;
    char *fname = zf->common.function_name;
```

```
    char *fsep;
    char *fclass;

    if (zf->common.scope) {
        fsep =  "::";
        fclass = zf->common.scope->name;
    } else {
        fsep =  "";
        fclass = "";
    }

    if (ptr && ptr->op_array) {
        zend_error(E_RECOVERABLE_ERROR, "Argument %d passed to %s%s%s() must
%s%s, %s%s given, called in %s on line %d and defined", arg_num, fclass, fsep,
fname, need_msg, need_kind, given_msg, given_kind, ptr->op_array->filename,
ptr->opline->lineno);
    } else {
        zend_error(E_RECOVERABLE_ERROR, "Argument %d passed to %s%s%s() must
%s%s, %s%s given", arg_num, fclass, fsep, fname, need_msg, need_kind,
given_msg, given_kind);
    }
    return 0;
}
```

第六节 变量的生命周期

通过前面章节的描述，我们已经知道了PHP中变量的存储方式－－所有的变量都保存在zval结构中。下面介绍一下PHP内核如何实现变量的定义方式以及作用域。

变量的生命周期

在ZE进行词法和语法的分析之后,生成具体的opcode,这些opcode最终被execute函数(Zend/zend_vm_execute.h:46)解释执行。在excute函数中，有以下代码:

```
while (1) {

    ...
    if ((ret = EX(opline)->handler(execute_data TSRMLS_CC)) > 0) {
        switch (ret) {
            case 1:
                EG(in_execution) = original_in_execution;
                return;
            case 2:
                op_array = EG(active_op_array);
                goto zend_vm_enter;
            case 3:
                execute_data = EG(current_execute_data);
            default:
                break;
        }
    }
    ...
}
```

这里的EX(opline)->handler(...)将op_array中的操作顺序执行，其中变量赋值操作在

ZEND_ASSIGN_SPEC_CV_CONST_HANDLER()函数中进行。
ZEND_ASSIGN_SPEC_CV_CONST_HANDLER中进行一些变量类型的判断并在内存中分配一个zval,然后将变量的值存储其中。变量名和指向这个zval的指针，则会存储于符号表内。
ZEND_ASSIGN_SPEC_CV_CONST_HANDLER的最后会调用ZEND_VM_NEXT_OPCODE()将op_array的指针移到下一条opline，这样就会形成循环执行的效果。

在ZE执行的过程中，有四个全局的变量，这些变量都是用于ZE运行时所需信息的存储：

```
//_zend_compiler_globals 编译时信息，包括函数表等
zend_compiler_globals    *compiler_globals;
//_zend_executor_globals 执行时信息
zend_executor_globals    *executor_globals;
//_php_core_globals 主要存储php.ini内的信息
php_core_globals         *core_globals;
//_sapi_globals_struct SAPI的信息
sapi_globals_struct      *sapi_globals;
```

在执行的过程中，变量名及指针主要存储于_zend_executor_globals的符号表中，_zend_executor_globals的结构这样的：

```
struct _zend_executor_globals {
...
/* symbol table cache */
HashTable *symtable_cache[SYMTABLE_CACHE_SIZE];
HashTable **symtable_cache_limit;
HashTable **symtable_cache_ptr;

zend_op **opline_ptr;

HashTable *active_symbol_table;  /* active symbol table */
HashTable symbol_table;      /* main symbol table */

HashTable included_files;    /* files already included */
...

}
```

在执行的过程中，active_symbol_table会根据执行的具体语句不断发生变化(详请见本节下半部分)，针对线程安全的EG宏就是用来取此变量中的值。 ZE将op_array执行完毕以后，HashTable会被FREE_HASHTABLE()释放掉。 如果程序使用了unset语句来主动消毁变量，则会调用ZEND_UNSET_VAR_SPEC_CV_HANDLER来将变量销毁，回收内存，这部分内存可以参考《第六章 内存管理》的内容。

变量的赋值和销毁

在强类型的语言当中，当使用一个变量之前，我们需要先声明这个变量。然而，对于PHP来说， 在使用一个变量时，我们不需要声明，也不需要初始化，直接用就可以了。那为什么可以直接用呢？ 下面我们看下原因。

变量的声明和赋值

在PHP中没有对于常规变量的声明操作，如果要使用一个变量,直接进行赋值操作即可。在赋值操作的时候已经将声明操作。 一个简单的赋值操作:

```
$a = 10;
```

使用VLD扩展查看其生成的中间代码为 **ASSIGN**. 依此，我们找到其执行的函数为 **ZEND_ASSIGN_SPEC_CV_CONST_HANDLER**. 之所以为这个函数是因为$a为CV,10为CONST，而我们进行的是一个ASSIGN操作。 CV是PHP在5.1后增加的一个在编译期的缓存。如我们在使用VLD查看上面的PHP代码生成的中间代码时会看到:

```
compiled vars:  !0 = $a
```

这个$a变量就是op_type为IS_CV的变量。

> IS_CV值的设置是在语法解析时进行的。具体见Zend/zend_complie.c文件 zend_do_end_variable_parse函数。

在这个函数中,获取这个赋值操作的左值和右值的代码为:

```
zval *value = &opline->op2.u.constant;
zval **variable_ptr_ptr = _get_zval_ptr_ptr_cv(&opline->op1, EX(Ts), BP_VAR_W
TSRMLS_CC);
```

由于右值为一个数值，我们可以理解为一个常量，则直接取操作数存储的constant字段，关于这个字段的说明将在后面的虚拟机章节说明。 左值是通过 _get_zval_ptr_ptr_cv函数获取zval值。这个函数最后的调用顺序为： [_get_zval_ptr_ptr_cv] --> [_get_zval_cv_lookup]

在_get_zval_cv_lookup函数中关键代码为:

```
zend_hash_quick_find(EG(active_symbol_table), cv->name, cv->name_len+1, cv-
>hash_value, (void **)ptr)
```

这是一个HashTable的查找函数，它的作用是从EG(active_symbol_table)中查找名称为cv->name的变量，并将这个值赋值给ptr。 最后，这个在符号表中找到的值将传递给ZEND_ASSIGN_SPEC_CV_CONST_HANDLER函数的variable_ptr_ptr变量。

以上是获取左值和右值的过程，在这步操作后将执行赋值操作的核心操作--赋值。赋值操作是通过调用zend_assign_to_variable函数实现。 在zend_assign_to_variable函数中，赋值操作分为好几种情况来处理，在程序中就是以几层的if语句体现。

情况一：赋值的左值存在引用，即zval变量中is_ref__gc字段不为0并且左值不等于右值

其在代码中的体现为:

```
if (PZVAL_IS_REF(variable_ptr)) {
    if (variable_ptr!=value) {
        zend_uint refcount = Z_REFCOUNT_P(variable_ptr);
```

```
        garbage = *variable_ptr;
        *variable_ptr = *value;
        Z_SET_REFCOUNT_P(variable_ptr, refcount);
        Z_SET_ISREF_P(variable_ptr);
        if (!is_tmp_var) {
            zendi_zval_copy_ctor(*variable_ptr);
        }
        zendi_zval_dtor(garbage);
        return variable_ptr;
    }
}
```

PZVAL_IS_REF(variable_ptr)判断is_ref__gc字段是否为0.在左值不等于右值的情况下执行操作。 所有指向这个zval容器的变量的值都变成了*value。并且引用计数的值不变。下面是这种情况的一个示例:

```
$a = 10;
$b = &$a;

xdebug_debug_zval('a');

$b = 20;
xdebug_debug_zval('a');
```

上面的程序输出:

```
a:
(refcount=2, is_ref=1),int 10
a:
(refcount=2, is_ref=1),int 20
```

情况二: 赋值的左值不存在引用，左值的引用计数为1，左值等于右值

在这种情况下，我们所看到的应该是什么都不会发生。看一个示例:

```
$a = 10;
$a = $a;
```

这个时候引用计数还是1，虽然没有变化，但是在内核中，变量的引用计数已经经历了一次加一和一次减一的操作。如下代码:

```
if (Z_DELREF_P(variable_ptr)==0) {  //  引用计数减一操作
        if (!is_tmp_var) {
            if (variable_ptr==value) {
                Z_ADDREF_P(variable_ptr);    //  引用计数加一操作
            }
...//省略
```

情况三: 赋值的左值不存在引用,左值的引用计数为1，右值存在引用

一个PHP的示例:

```
$a = 10;
$b = &$a;
$c = $a;
```

第三行的操作就是我们所示的第三种情况。其内核实现代码如下：

```
garbage = *variable_ptr;
*variable_ptr = *value;
INIT_PZVAL(variable_ptr);    //  初始化一个新的zval变量容器
zval_copy_ctor(variable_ptr);
zendi_zval_dtor(garbage);
return variable_ptr;
```

对于这种情况，ZEND内核直接创建一个新的zval容器，左值的值为右值，并且左值的引用计数为1。

情况四：赋值的左值不存在引用,左值的引用计数为1，右值不存在引用

一个PHP示例：

```
$a = 10;
$c = $a;
```

在这种情况下，右值的引用计数加上，一般情况下，会对左值进行垃圾收集操作，将其移入垃圾缓冲池。垃圾缓冲池的功能是在PHP5.3后才有的。 在PHP内核中的代码体现为：

```
Z_ADDREF_P(value);  //  引用计数加1
*variable_ptr_ptr = value;
if (variable_ptr != &EG(uninitialized_zval)) {
    GC_REMOVE_ZVAL_FROM_BUFFER(variable_ptr);    //  调用垃圾收集机制
    zval_dtor(variable_ptr);
    efree(variable_ptr);    //  释放变量内存空间
}
return value;
```

情况五：赋值的左值不存在引用,左值的引用计数为不为1，右值存在引用，并且引用计数大于0

一个演示这种情况的PHP示例：

```
$a = 10;
$va = 20;
$vb = &$va;

$a = $va;
```

最后一个操作就是我们的情况五。 使用xdebug看引用计数，我们可以知道，$a变量的引用计数为1，$va变量的引用计数为2，并且存在引用。 从源码层分析这个原因：

```
ALLOC_ZVAL(variable_ptr);    //  分配新的zval容器
```

```
*variable_ptr_ptr = variable_ptr;
*variable_ptr = *value;
zval_copy_ctor(variable_ptr);
Z_SET_REFCOUNT_P(variable_ptr, 1);  //  设置引用计数为1
```

从代码可以看出是新分配了一个zval容器，并设置了引用计数为1,这可以确认我们之前的例子中的$a变量的结果是正确，

除了这五种情况，函数中针对临时变量全部做了另外的处理。关于变量赋值的各种情况尽在这个函数。

变量的销毁

在PHP中销毁变量最常用的方法是使用unset函数。 unset函数并不是一个真正意义上的函数，它是一种语言结构。 在使用此函数时，它会根据变量的不同触发不同的操作。

一个简单的例子:

```
$a = 10;
unset($a);
```

使用VLD扩展查看其生成的中间代码:

```
compiled vars:  !0 = $a
line     # *  op                                fetch          ext  return  operands
---------------------------------------------------------------------------------
--
   2     0  >   EXT_STMT
         1      ASSIGN                                                   !0, 10
   3     2      EXT_STMT
         3      UNSET_VAR                                                !0
         4    > RETURN                                                   1
```

去掉关于赋值的中间代码，得到unset函数生成的中间代码为 **UNSET_VAR**,由于我们unse的是一个变量， 在Zend/zend_vm_execute.h文件中查找到其最终调用的执行中间代码的函数为: **ZEND_UNSET_VAR_SPEC_CV_HANDLER** 关键代码代码如下:

```
target_symbol_table = zend_get_target_symbol_table(opline, EX(Ts),
        BP_VAR_IS, varname TSRMLS_CC);
    if (zend_hash_quick_del(target_symbol_table, varname->value.str.val,
            varname->value.str.len+1, hash_value) == SUCCESS) {
        ...//省略
    }
```

程序会先获取目标符号表，这个符号表是一个HashTable,然后将我们需要unset掉的变量从这个HashTable中删除。 关于HashTable的操作请参考 << 哈希表(HashTable) >>。

变量的作用域

作用域是变量一个很重要的概念。变量的作用域就是语句的一个作用范围，在这个范围内变量为可见

的。 换句话说，变量的作用域是可以访问该变量的代码区域。我们常见的包含作用域概念的变量包括全局变量和局部变量。 变量的作用域与变量的生命周期有一定的联系，如在一个函数中定义的变量，这个变量的作用域从变量声明的时候开始到这个函数结束的时候。 这种变量我们称之为局部变量。它的生命周期开始于函数开始，结束于函数的调用完成之时。 对于不同作用域的变量，如果存在冲突情况，如全局变量中有一个名为$a的变量，在局部变量中也存在一个名为$a的变量， 此时如何区分呢？这个也是我们将要探讨的问题。

对于全局变量，ZEND内核有一个_zend_executor_globals结构，该结构中的symbol_table就是全局符号表， 其中保存了在顶层作用域中的变量。同样，函数或者对象的方法在被调用时会创建active_symbol_table来保存局部变量。 当程序在顶层中使用某个变量时，ZE就会在symbol_table中进行遍历，同理，每个函数也会有对应的active_symbol_table来供程序使用。

对于局部变量，如我们调用的一个函数中的变量，ZE使用_zend_execute_data来存储 某个单独的op_array（每个函数都会生成单独的op_array)执行过程中所需要的信息，它的结构如下：

```
struct _zend_execute_data {
    struct _zend_op *opline;
    zend_function_state function_state;
    zend_function *fbc; /* Function Being Called */
    zend_class_entry *called_scope;
    zend_op_array *op_array;
    zval *object;
    union _temp_variable *Ts;
    zval ***CVs;
    HashTable *symbol_table;
    struct _zend_execute_data *prev_execute_data;
    zval *old_error_reporting;
    zend_bool nested;
    zval **original_return_value;
    zend_class_entry *current_scope;
    zend_class_entry *current_called_scope;
    zval *current_this;
    zval *current_object;
    struct _zend_op *call_opline;
};
```

函数中的局部变量就存储在_zend_execute_data的symbol_table中，在执行当前函数的op_array时，全局zend_executor_globals中的*active_symbol_table会指向当前_zend_execute_data中的*symbol_table。 而此时，其他函数中的symbol_table不会出现在当前的active_symbol_table中，如此便实现了局部变量。 相关操作在 Zend/zend_vm_execute.h 文件中定义的execute函数中一目了然，如下所示代码：

```
zend_vm_enter:
/* Initialize execute_data */
execute_data = (zend_execute_data *)zend_vm_stack_alloc(
    sizeof(zend_execute_data) +
    sizeof(zval**) * op_array->last_var * (EG(active_symbol_table) ? 1 : 2) +
    sizeof(temp_variable) * op_array->T TSRMLS_CC);

EX(symbol_table) = EG(active_symbol_table);
EX(prev_execute_data) = EG(current_execute_data);
EG(current_execute_data) = execute_data;
```

EX宏的作用是取结构体zend_execute_data的字段值，如下所示代码：

```
#define EX(element) execute_data->element
```

所以，变量的作用域是使用不同的符号表来实现的，于是顶层的全局变量在函数内部使用时，需要先使用上一节中提到的global语句进行变量的跨域操作。

global语句

global语句的作用是定义全局变量, 例如如果想在函数内访问全局作用域内的变量则可以通过global声明来定义。 下面从语法解释开始分析.

1. 词法解析

查看 Zend/zend_language_scanner.l文件，搜索 global关键字。我们可以找到如下代码:

```
<ST_IN_SCRIPTING>"global" {
return T_GLOBAL;
}
```

2. 语法解析

在词法解析完后，获得了token,此时通过这个token，我们去Zend/zend_language_parser.y文件中查找。找到相关代码如下:

```
|   T_GLOBAL global_var_list ';'

global_var_list:
    global_var_list ',' global_var  { zend_do_fetch_global_variable(&$3, NULL,
ZEND_FETCH_GLOBAL_LOCK TSRMLS_CC); }
|   global_var                      { zend_do_fetch_global_variable(&$1, NULL,
ZEND_FETCH_GLOBAL_LOCK TSRMLS_CC); }
;
```

上面代码中的**$3**是指global_var（如果不清楚yacc的语法，可以查阅yacc入门类的文章。）

从上面的代码可以知道，对于全局变量的声明调用的是zend_do_fetch_global_variable函数，查找此函数的实现在Zend/zend_compile.c文件。

```
void zend_do_fetch_global_variable(znode *varname, const znode
*static_assignment, int fetch_type TSRMLS_DC)
{
        ...//省略
        opline->opcode = ZEND_FETCH_W;      /* the default mode must be Write,
since fetch_simple_variable() is used to define function arguments */
        opline->result.op_type = IS_VAR;
        opline->result.u.EA.type = 0;
        opline->result.u.var = get_temporary_variable(CG(active_op_array));
        opline->op1 = *varname;
        SET_UNUSED(opline->op2);
        opline->op2.u.EA.type = fetch_type;
        result = opline->result;

        ... // 省略
```

```
        fetch_simple_variable(&lval, varname, 0 TSRMLS_CC); /* Relies on the
fact that the default fetch is BP_VAR_W */

        zend_do_assign_ref(NULL, &lval, &result TSRMLS_CC);
        CG(active_op_array)->opcodes[CG(active_op_array)-
>last-1].result.u.EA.type |= EXT_TYPE_UNUSED;
}
/* }}} */
```

上面的代码确认了opcode为ZEND_FETCH_W外，还执行了zend_do_assign_ref函数。zend_do_assign_ref函数的实现如下：

```
void zend_do_assign_ref(znode *result, const znode *lvar, const znode *rvar
TSRMLS_DC) /* {{{ */
{
        zend_op *opline;

       ... //省略

        opline = get_next_op(CG(active_op_array) TSRMLS_CC);
        opline->opcode = ZEND_ASSIGN_REF;
       ...//省略
        if (result) {
                opline->result.op_type = IS_VAR;
                opline->result.u.EA.type = 0;
                opline->result.u.var =
get_temporary_variable(CG(active_op_array));
                *result = opline->result;
        } else {
                /* SET_UNUSED(opline->result); */
                opline->result.u.EA.type |= EXT_TYPE_UNUSED;
        }
        opline->op1 = *lvar;
        opline->op2 = *rvar;
}
```

从上面的zend_do_fetch_global_variable函数和zend_do_assign_ref函数的实现可以看出，使用global声明一个全局变量后，其执行了两步操作，ZEND_FETCH_W和ZEND_ASSIGN_REF。

3. 生成并执行中间代码

我们看下ZEND_FETCH_W的最后执行。从代码中我们可以知道：

- ZEND_FETCH_W = 83
- op->op1.op_type = 4
- op->op2.op_type = 0

而计算最后调用的方法在代码中的体现为：

```
zend_opcode_handlers[opcode * 25 + zend_vm_decode[op->op1.op_type] * 5 +
zend_vm_decode[op->op2.op_type]];
```

计算，最后调用ZEND_FETCH_W_SPEC_CV_HANDLER函数。即

```
static int ZEND_FASTCALL
ZEND_FETCH_W_SPEC_CV_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
```

```
{
        return zend_fetch_var_address_helper_SPEC_CV(BP_VAR_W,
ZEND_OPCODE_HANDLER_ARGS_PASSTHRU);
}
```

在zend_fetch_var_address_helper_SPEC_CV中调用如下代码获取符号表

```
target_symbol_table = zend_get_target_symbol_table(opline, EX(Ts), type,
varname TSRMLS_CC);
```

在zend_get_target_symbol_table函数的实现如下:

```
static inline HashTable *zend_get_target_symbol_table(const zend_op *opline,
const temp_variable *Ts, int type, const zval *variable TSRMLS_DC)
{
        switch (opline->op2.u.EA.type) {
                ... //  省略
                case ZEND_FETCH_GLOBAL:
                case ZEND_FETCH_GLOBAL_LOCK:
                        return &EG(symbol_table);
                        break;
               ...  //  省略
        }
        return NULL;
}
```

在前面语法分析过程中，程序传递的参数是 ZEND_FETCH_GLOBAL_LOCK，于是如上所示。我们取&EG(symbol_table);的值。这也是全局变量的存放位置。

如上就是整个global的解析过程。

第七节 数据类型转换

PHP中的变量不需要显式的数据类型定义, 可以给变量赋值任意类型的数据, PHP之间的数据类型转换有两种: 显式和隐式转换.

隐式类型转换(自动类型转换)

PHP中隐式数据类型转换很常见, 例如:

```
<?php
$a = 10;
$b = 'a string ';

echo $a . $b;
```

上面例子中字符串连接操作就存在自动数据类型转化, $a变量是数值类型, $b变量是字符串类型, 这里$b变量就是隐式(自动)的转换 为字符串类型了. 通常自动数据类型转换发生在特定的操作上下文中, 类似的还有求和操作"+". 具体的自动类型转换方式和特定的操作 有关. 下面就以字符串连接操作为例:

脚本执行的时候字符串的链接操作是通过Zend/zend_operators.c文件中的如下函数进行:

```
ZEND_API int concat_function(zval *result, zval *op1, zval *op2 TSRMLS_DC) /*
{{{ */
{
        zval op1_copy, op2_copy;
        int use_copy1 = 0, use_copy2 = 0;

        if (Z_TYPE_P(op1) != IS_STRING) {
                zend_make_printable_zval(op1, &op1_copy, &use_copy1);
        }
        if (Z_TYPE_P(op2) != IS_STRING) {
                zend_make_printable_zval(op2, &op2_copy, &use_copy2);
        }
        // 省略
}
```

可用看出如果字符串链接的两个操作数如果不是字符串的话则调用zend_make_printable_zval函数将操作数转换为"printable_zval"也就是字符串.

```
ZEND_API void zend_make_printable_zval(zval *expr, zval *expr_copy, int
*use_copy)
{
    if (Z_TYPE_P(expr)==IS_STRING) {
        *use_copy = 0;
        return;
    }
    switch (Z_TYPE_P(expr)) {
        case IS_NULL:
            Z_STRLEN_P(expr_copy) = 0;
            Z_STRVAL_P(expr_copy) = STR_EMPTY_ALLOC();
            break;
        case IS_BOOL:
            if (Z_LVAL_P(expr)) {
                Z_STRLEN_P(expr_copy) = 1;
                Z_STRVAL_P(expr_copy) = estrndup("1", 1);
            } else {
                Z_STRLEN_P(expr_copy) = 0;
                Z_STRVAL_P(expr_copy) = STR_EMPTY_ALLOC();
            }
            break;
        case IS_RESOURCE:
            Z_STRVAL_P(expr_copy) = (char *) emalloc(sizeof("Resource id #") -
1 + MAX_LENGTH_OF_LONG);
            Z_STRLEN_P(expr_copy) = sprintf(Z_STRVAL_P(expr_copy), "Resource id
#%ld", Z_LVAL_P(expr));
            break;
        case IS_ARRAY:
            Z_STRLEN_P(expr_copy) = sizeof("Array") - 1;
            Z_STRVAL_P(expr_copy) = estrndup("Array", Z_STRLEN_P(expr_copy));
            break;
        case IS_OBJECT:
            {
                TSRMLS_FETCH();

                if (Z_OBJ_HANDLER_P(expr, cast_object) && Z_OBJ_HANDLER_P(expr,
cast_object)(expr, expr_copy, IS_STRING TSRMLS_CC) == SUCCESS) {
                    break;
                }
                /* Standard PHP objects */
                if (Z_OBJ_HT_P(expr) == &std_object_handlers ||
!Z_OBJ_HANDLER_P(expr, cast_object)) {
```

```
                    if (zend_std_cast_object_tostring(expr, expr_copy,
IS_STRING TSRMLS_CC) == SUCCESS) {
                        break;
                    }
                }
                if (!Z_OBJ_HANDLER_P(expr, cast_object) &&
Z_OBJ_HANDLER_P(expr, get)) {
                    zval *z = Z_OBJ_HANDLER_P(expr, get)(expr TSRMLS_CC);

                    Z_ADDREF_P(z);
                    if (Z_TYPE_P(z) != IS_OBJECT) {
                        zend_make_printable_zval(z, expr_copy, use_copy);
                        if (*use_copy) {
                            zval_ptr_dtor(&z);
                        } else {
                            ZVAL_ZVAL(expr_copy, z, 0, 1);
                            *use_copy = 1;
                        }
                        return;
                    }
                    zval_ptr_dtor(&z);
                }
                zend_error(EG(exception) ? E_ERROR : E_RECOVERABLE_ERROR,
"Object of class %s could not be converted to string", Z_OBJCE_P(expr)->name);
                Z_STRLEN_P(expr_copy) = 0;
                Z_STRVAL_P(expr_copy) = STR_EMPTY_ALLOC();
            }
            break;
        case IS_DOUBLE:
            *expr_copy = *expr;
            zval_copy_ctor(expr_copy);
            zend_locale_sprintf_double(expr_copy ZEND_FILE_LINE_CC);
            break;
        default:
            *expr_copy = *expr;
            zval_copy_ctor(expr_copy);
            convert_to_string(expr_copy);
            break;
    }
    Z_TYPE_P(expr_copy) = IS_STRING;
    *use_copy = 1;
}
```

这个函数根据不同的变量类型来返回不同的字符串类型, 例如BOOL类型的数据返回0和1, 数组只是简单的返回Array等等, 类似其他类型的数据转换也是类型, 都是根据操作数的不同类型的转换为相应的目标类型.

显式类型转换(强制类型转换)

PHP中的强制类型转换和C中的非常像:

```
<?php
$double = 20.10;
echo (int)$double;
```

PHP中允许的强制类型有:

- (int), (integer) 转换为整型

- (bool), (boolean) 转换为布尔类型
- (float), (double) 转换为浮点类型
- (string) 转换为字符串
- (array) 转换为数组
- (object) 转换为对象
- (unset) 转换为NULL

在Zend/zend_operators.c中实现了转换为这些目标类型的实现函数convert_to_*系列函数, 读者自行查看这些函数即可, 这些数据类型转换类型中有unset类型转换:

```
ZEND_API void convert_to_null(zval *op) /* {{{ */
{
    if (Z_TYPE_P(op) == IS_OBJECT) {
        if (Z_OBJ_HT_P(op)->cast_object) {
            zval *org;
            TSRMLS_FETCH();

            ALLOC_ZVAL(org);
            *org = *op;
            if (Z_OBJ_HT_P(op)->cast_object(org, op, IS_NULL TSRMLS_CC) ==
SUCCESS) {
                zval_dtor(org);
                return;
            }
            *op = *org;
            FREE_ZVAL(org);
        }
    }

    zval_dtor(op);
    Z_TYPE_P(op) = IS_NULL;
}
```

转换为NULL非常简单, 对变量进行析构操作,然后将数据类型设为IS_NULL即可. 可能读者会好奇(unset)$a和unset($a)这两者有没有关系,其实并没有关系,前者是将 变量$a的类型变为NULL, 而后者是将这个变量释放, 释放后当前作用域内该变量及不存在了.

PHP的标准扩展中提供了两个有用的方法settype()以及gettype()方法, 前者可以动态的改变变量的数据类型, gettype()方法则是返回变量的数据类型.

```
PHP_FUNCTION(settype)
{
    zval **var;
    char *type;
    int type_len = 0;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "Zs", &var, &type,
&type_len) == FAILURE) {
        return;
    }

    if (!strcasecmp(type, "integer")) {
        convert_to_long(*var);
    } else if (!strcasecmp(type, "int")) {
        convert_to_long(*var);
    } else if (!strcasecmp(type, "float")) {
        convert_to_double(*var);
```

```
    } else if (!strcasecmp(type, "double")) { /* deprecated */
        convert_to_double(*var);
    } else if (!strcasecmp(type, "string")) {
        convert_to_string(*var);
    } else if (!strcasecmp(type, "array")) {
        convert_to_array(*var);
    } else if (!strcasecmp(type, "object")) {
        convert_to_object(*var);
    } else if (!strcasecmp(type, "bool")) {
        convert_to_boolean(*var);
    } else if (!strcasecmp(type, "boolean")) {
        convert_to_boolean(*var);
    } else if (!strcasecmp(type, "null")) {
        convert_to_null(*var);
    } else if (!strcasecmp(type, "resource")) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Cannot convert to resource
type");
        RETURN_FALSE;
    } else {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Invalid type");
        RETURN_FALSE;
    }
    RETVAL_TRUE;
}
```

这个函数主要作为一个代理方法, 具体的转换规则由各个类型的处理函数处理, 不管是自动还是强制类型转换,最终都会调用这些内部转换方法.

第八节 小结

在命令式程序语言中，程序的变量是程序语言对计算机的存储单元或一系列存储单元的抽象。 在PHP语言中，变量是对于C语言结构体和一系列结构体的抽象。但是追根究底，它也是对计算机的存储单元或一系列存储单元的抽象。

在这一章中，我们向您展示了PHP实现的内部结构，常量，预定义变量，静态变量等常见变量的实现，除此之外，还介绍了在PHP5之后才有的类型提示，变量的作用域以及类型的转换。

下一章将探索PHP对于函数的实现。

第四章 函数的实现

函数是一种可以在任何被需要的时候执行的代码块。它不仅仅包括用户自定义的函数，更多的是程序语言实现的库函数。

用户定义的函数

如下所示手册中的展示函数用途的伪代码

```
function foo($arg_1, $arg_2, ..., $arg_n) {
    echo "Example function.\n";
    return $retval;
}
```

任何有效的 PHP 代码都有可能出现在函数内部，甚至包括其它函数和类定义。

> 在 PHP 3 中，函数必须在被调用之前定义。而 PHP 4 则不再有这样的条件。除非函数如以下两个范例中有条件的定义。

内部函数

PHP 有很多标准的函数和结构。如我们常见的count,strpos,implode等函数，这些都是标准函数，它们都以标准扩展的形式存在； 如我们经常用到的isset,empty,eval等函数，或者它们应该称之为语言结构。还有一些函数需要和特定地 PHP 扩展模块一起编译，否则在使用它们的时候就会得到一个致命的“未定义函数”错误。 从源码层来看，标准的函数和标准的语言结构是分离。

> 标准函数的实现存放在ext/standard目录中。

匿名函数

匿名函数的作用就是扩大函数的使用功能，在PHP 5.3以前，传递Callback的方式，我们只有两种选择：

- 字符串的函数名
- 使用create_function的返回

在PHP5.3以后, 我们多了一个选择--Closure。但是PHP 5.3中对匿名函数的支持, 也仅仅是把要保持的外部变量, 做为Closure对象的”Static属性”,关于如何实现我们将在后面的章节介绍。 一个匿名函数的例子：

```
$func = function($str) {
            echo $str;
        };

$str = "tipi";
$func($str);
```

```
print_r($func);
```

变量函数

PHP 支持变量函数的概念。这意味着如果一个变量名后有圆括号，PHP 将寻找与变量的值同名的函数，并且将尝试执行它。除此之外，这个可以被用于实现回调函数，函数表等。 一个变量函数的简单例子：

```
$func = 'print_r';
$func('i am print_r function.');
```

变量函数不能用于语言结构，

下面我们将开始关注函数在PHP中具体实现，函数的内部结构，函数的调用，参数传递以及函数返回值等。

第一节 函数的内部结构

在PHP中，函数有自己的作用域，同时在其内部可以实现各种语句的执行，最后返回最终结果值。在PHP的源码中可以发现，PHP内核将函数分为以下类型：

```
#define ZEND_INTERNAL_FUNCTION             1
#define ZEND_USER_FUNCTION                 2
#define ZEND_OVERLOADED_FUNCTION           3
#define ZEND_EVAL_CODE                     4
#define ZEND_OVERLOADED_FUNCTION_TEMPORARY 5
```

其中的*ZEND_USER_FUNCTION*是用户函数，*ZEND_INTERNAL_FUNCTION*是内置的函数。 (PHP内部对不同种类的函数使用了不同的结构来进行实现?)

1.用户函数(ZEND_USER_FUNCTION)

用户自定义函数是非常常用的函数种类，如下面的代码，就定义了一个用户自定义的函数：

```
<?php

function tipi( $name ){
    $return = "Hi! " . $name;
    echo $return;
    return $return;
}

?>
```

这个示例中，对自定义函数传入了一个参数，并将其与*Hi!* 一起输出并做为返回值返回。从这个例子可以看出函数的基本特点：运行时声明、可以传参数、有值返回。 当然，有些函数只是进行一些操作，并不一定有返回值，而事实上，在PHP的实现中，即使没有返回值，PHP内核也会“帮你“加上一个NULL来做为

返回值。

通过 <<第六节 变量的作用域>> 可知，ZE在执行过程中，会将运行时信息存储于_zend_execute_data中：

```
struct _zend_execute_data {
    //...省略部分代码
    zend_function_state function_state;
    zend_function *fbc; /* Function Being Called */
    //...省略部分代码
};
```

在程序初始化的过程中，function_state也会进行初始化，function_state由两个部分组成：

```
typedef struct _zend_function_state {
    zend_function *function;
    void **arguments;
} zend_function_state;
```

**arguments是一个指向函数参数的指针，而函数体本身则存储于*function中， *function是一个zend_function结构体，它最终存储了用户自定义函数的一切信息，它的具体结构是这样的：

```
typedef union _zend_function {
    zend_uchar type;    /* MUST be the first element of this struct! */

    struct {
        zend_uchar type;  /* never used */
        char *function_name;    //函数名称
        zend_class_entry *scope; //函数所在的类作用域
        zend_uint fn_flags;     //函数类型，如用户自定义则为 #define
ZEND_USER_FUNCTION 2
        union _zend_function *prototype; //函数原型
        zend_uint num_args;     //参数数目
        zend_uint required_num_args; //需要的参数数目
        zend_arg_info *arg_info;  //参数信息指针
        zend_bool pass_rest_by_reference;
        unsigned char return_reference;  //返回值
    } common;

    zend_op_array op_array;   //函数中的操作
    zend_internal_function internal_function;
} zend_function;
```

*zend_function*的结构中的op_array存储了该函数中所有的操作，当函数被调用时，ZE就会将这个op_array中的opline一条条顺次执行，并将最后的返回值返回。 从VLD扩展的显示的关于函数的信息可以看出，函数的定义和执行是分开的，一个函数可以作为一个独立的运行单元而存在。

2.内部函数(ZEND_INTERNAL_FUNCTION)

ZEND_INTERNAL_FUNCTION函数是由扩展或者Zend/PHP内核提供的，用“C/C++”编写的，可以直接执行的函数。如下为内部函数的结构：

```
typedef struct _zend_internal_function {
```

```
    /* Common elements */
    zend_uchar type;
    char * function_name;
    zend_class_entry *scope;
    zend_uint fn_flags;
    union _zend_function *prototype;
    zend_uint num_args;
    zend_uint required_num_args;
    zend_arg_info *arg_info;
    zend_bool pass_rest_by_reference;
    unsigned char return_reference;
    /* END of common elements */

    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
    struct _zend_module_entry *module;
} zend_internal_function;
```

最常见的操作是在模块初始化时，ZE会遍历每个载入的扩展模块，然后将模块中function_entry中指明的每一个函数(module->functions)， 创建一个zend_internal_function结构， 并将其type设置为ZEND_INTERNAL_FUNCTION, 将这个结构填入全局的函数表(HashTable结构）；函数设置及注册过程见Zend/zend_API.c文件中的 **zend_register_functions**函数。这个函数除了处理函数，也处理类的方法，包括那些魔术方法。

内部函数的结构与用户自定义的函数结构基本类似，有一些不同，

- 调用方法，handler字段. 如果是ZEND_INTERNAL_FUNCTION， 那么ZE就调用zend_execute_internal,通过zend_internal_function.handler来执行这个函数。而用户自定义的函数需要生成中间代码，然后通过中间代码映射到相对就把方法调用。
- 内置函数在结构中多了一个module字段，表示属于哪个模块。不同的扩展其模块不同。
- type字段，在用户自定义的函数中，type字段几科无用，而内置函数中的type字段作为几种内部函数的区分。

3.变量函数

PHP 支持变量函数的概念。这意味着如果一个变量名后有圆括号，PHP 将寻找与变量的值同名的函数，并且将尝试执行它。 除此之外，这个可以被用于实现回调函数，函数表等。 对比使用变量函数和内部函数的调用:

变量函数$func

```
$func = 'print_r';
$func('i am print_r function.');
```

通过VLD来查看这段代码编译后的中间代码:

```
function name:  (null)
number of ops:  9
compiled vars:  !0 = $func
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------------
-
-
```

```
   2     0  >   EXT_STMT
         1      ASSIGN                                                 !0,
'print_r'
   3     2      EXT_STMT
         3      INIT_FCALL_BY_NAME                                     !0
         4      EXT_FCALL_BEGIN
         5      SEND_VAL
'i+am+print_r+function.'
         6      DO_FCALL_BY_NAME                             1
         7      EXT_FCALL_END
         8    > RETURN                                               1
```

内部函数print_r

```
print_r('i am print_r function.');
```

通过VLD来查看这段代码编译后的中间代码:

```
function name:  (null)
number of ops:  6
compiled vars:  none
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------
-
-
   2     0  >   EXT_STMT
         1      EXT_FCALL_BEGIN
         2      SEND_VAL
'i+am+print_r+function.'
         3      DO_FCALL                                     1
'print_r'
         4      EXT_FCALL_END
         5    > RETURN                                               1
```

对比发现，二者在调用的中间代码上存在一些区别。变量函数是DO_FCALL_BY_NAME，而内部函数是DO_FCALL。这在语法解析时就已经决定了， 见Zend/zend_complie.c文件的zend_do_end_function_call函数中部分代码:

```
if (!is_method && !is_dynamic_fcall && function_name->op_type==IS_CONST) {
        opline->opcode = ZEND_DO_FCALL;
        opline->op1 = *function_name;
        ZVAL_LONG(&opline->op2.u.constant,
zend_hash_func(Z_STRVAL(function_name->u.constant), Z_STRLEN(function_name-
>u.constant) + 1));
    } else {
        opline->opcode = ZEND_DO_FCALL_BY_NAME;
        SET_UNUSED(opline->op1);
    }
```

如果不是方法，并且不是动态调用，并且函数名为字符串常量，则其生成的中间代码为ZEND_DO_FCALL。其它情况则为ZEND_DO_FCALL_BY_NAME。 另外将变量函数作为回调函数，其处理过程在Zend/zend_complie.c文件的zend_do_pass_param函数中。 最终会体现在中间代码执行过程中的 **ZEND_SEND_VAL_SPEC_CONST_HANDLER** 等函数中。

4.匿名函数

匿名函数是一类不需要指定表示符, 而又可以被调用的函数或子例程, 匿名函数可以方便的作为参数传递给其他函数, 关于匿名函数的详细信息请阅读 <<第四节 匿名函数及闭包>>

第二节 函数的定义,传参及返回值

在本章开头部分，介绍了四种函数，而在本小节，我们从第一种函数：用户自定义的函数开始来认识函数。 本小节包括函数的定义，函数的参数传递和函数的返回值三个部分。下面我们将对每个部分做详细介绍。

函数的定义

在PHP中，用户函数的定义从function关键字开始。如下所示简单示例：

```
function foo($var) {
    echo $var;
}
```

这是一个非常简单的函数，它所实现的功能是定义一个函数，函数有一个参数，函数的内容是在标准输出端输出传递给它的参数变量的值。

函数的一切从function开始。我们从function开始函数定义的探索之旅。

词法分析

在 Zend/zend_language_scanner.l中我们找到如下所示的代码：

```
<ST_IN_SCRIPTING>"function" {
    return T_FUNCTION;
}
```

它所表示的含义是function将会生成T_FUNCTION标记。在获取这个标记后，我们开始语法分析。

语法分析

在 Zend/zend_language_parser.y文件中找到函数的声明过程标记如下：

```
function:
    T_FUNCTION { $$.u.opline_num = CG(zend_lineno); }
;

is_reference:
        /* empty */ { $$.op_type = ZEND_RETURN_VAL; }
    |   '&'         { $$.op_type = ZEND_RETURN_REF; }
;

unticked_function_declaration_statement:
        function is_reference T_STRING {
zend_do_begin_function_declaration(&$1, &$3, 0, $2.op_type, NULL TSRMLS_CC); }
            '(' parameter_list ')' '{' inner_statement_list '}' {
```

```
                zend_do_end_function_declaration(&$1 TSRMLS_CC); }
;
```

关注点在 function is_reference T_STRING，表示function关键字，是否引用，函数名。

T_FUNCTION标记只是用来定位函数的声明，表示这是一个函数，而更多的工作是与这个函数相关的东西，包括参数，返回值等。

生成中间代码

语法解析后，我们看到所执行编译函数为zend_do_begin_function_declaration。在Zend/zend_complie.c文件中找到其实现如下：

```
void zend_do_begin_function_declaration(znode *function_token, znode
*function_name,
 int is_method, int return_reference, znode *fn_flags_znode TSRMLS_DC) /* {{{
*/
{
    ...//省略
    function_token->u.op_array = CG(active_op_array);
    lcname = zend_str_tolower_dup(name, name_len);

    orig_interactive = CG(interactive);
    CG(interactive) = 0;
    init_op_array(&op_array, ZEND_USER_FUNCTION, INITIAL_OP_ARRAY_SIZE
TSRMLS_CC);
    CG(interactive) = orig_interactive;

     ...//省略

    if (is_method) {
        ...//省略 类方法 在后面的类章节介绍
    } else {
        zend_op *opline = get_next_op(CG(active_op_array) TSRMLS_CC);


        opline->opcode = ZEND_DECLARE_FUNCTION;
        opline->op1.op_type = IS_CONST;
        build_runtime_defined_function_key(&opline->op1.u.constant, lcname,
            name_len TSRMLS_CC);
        opline->op2.op_type = IS_CONST;
        opline->op2.u.constant.type = IS_STRING;
        opline->op2.u.constant.value.str.val = lcname;
        opline->op2.u.constant.value.str.len = name_len;
        Z_SET_REFCOUNT(opline->op2.u.constant, 1);
        opline->extended_value = ZEND_DECLARE_FUNCTION;
        zend_hash_update(CG(function_table), opline-
>op1.u.constant.value.str.val,
            opline->op1.u.constant.value.str.len, &op_array,
sizeof(zend_op_array),
             (void **) &CG(active_op_array));
    }

}
/* }}} */
```

生成的中间代码为 **ZEND_DECLARE_FUNCTION** ，根据这个中间代码及操作数对应的op_type。我们可以找到中间代码的执行函数为 **ZEND_DECLARE_FUNCTION_SPEC_HANDLER**。

在生成中间代码时，可以看到已经统一了函数名全部为小写，表示函数的名称不是区分大小写的。

为验证这个实现，我们看一段代码：

```
function T() {
    echo 1;
}

function t() {
    echo 2;
}
```

执行代码，可以看到屏幕上输出如下报错信息：

```
Fatal error: Cannot redeclare t() (previously declared in ...)
```

表示对于PHP来说T和t是同一个函数名。检验函数名是否重复，这个过程是在哪进行的呢？下面将要介绍的函数声明中间代码的执行过程包含了这个检查过程。

执行中间代码

在 Zend/zend_vm_execute.h 文件中找到 ZEND_DECLARE_FUNCTION中间代码对应的执行函数：ZEND_DECLARE_FUNCTION_SPEC_HANDLER。 此函数只调用了函数do_bind_function。其调用代码为：

```
do_bind_function(EX(opline), EG(function_table), 0);
```

在这个函数中将EX(opline)所指向的函数添加到EG(function_table)中，并判断是否已经存在相同名字的函数，如果存在则报错。 EG(function_table)用来存放执行过程中全部的函数信息，相当于函数的注册表。 它的结构是一个HashTable，所以在do_bind_function函数中添加新的函数使用的是HashTable的操作函数**zend_hash_add**

函数的参数

前一小节介绍了函数的定义，函数的定义只是一个将函数名注册到函数列表的过程，在了解了函数的定义后，我们来看看函数的参数。 这一小节将包括用户自定义函数的参数和内部函数的参数两部分，详细内容如下：

用户自定义函数的参数

在<<第三章第五小节 类型提示的实现>>中，我们对于参数的类型提示做了分析，这里我们在这一小节的基础上，进行一些更详细的说明。 在经过词语分析，语法分析后，我们知道对于函数的参数检查是通过**zend_do_receive_arg** 函数来实现的。在此函数中对于参数的关键代码如下：

```
CG(active_op_array)->arg_info = erealloc(CG(active_op_array)->arg_info,
        sizeof(zend_arg_info)*(CG(active_op_array)->num_args));
```

```
cur_arg_info = &CG(active_op_array)->arg_info[CG(active_op_array)->num_args-1];
cur_arg_info->name = estrndup(varname->u.constant.value.str.val,
        varname->u.constant.value.str.len);
cur_arg_info->name_len = varname->u.constant.value.str.len;
cur_arg_info->array_type_hint = 0;
cur_arg_info->allow_null = 1;
cur_arg_info->pass_by_reference = pass_by_reference;
cur_arg_info->class_name = NULL;
cur_arg_info->class_name_len = 0;
```

整个参数的传递是通过给中间代码的arg_info字段执行赋值操作完成。关键点是在arg_info字段。arg_info字段的结构如下：

```
typedef struct _zend_arg_info {
    const char *name;    /* 参数的名称*/
    zend_uint name_len;      /* 参数名称的长度*/
    const char *class_name; /* 类名 */
    zend_uint class_name_len;    /* 类名长度*/
    zend_bool array_type_hint;   /* 数组类型提示 */
    zend_bool allow_null;    /* 是否允许为NULL  */
    zend_bool pass_by_reference;     /*  是否引用传递 */
    zend_bool return_reference;
    int required_num_args;
} zend_arg_info;
```

参数的值传递和参数传递的区分是通过 **pass_by_reference**参数在生成中间代码时实现的。

对于参数的个数，中间代码中包含的arg_nums字段在每次执行 **zend_do_receive_arg×× 时都会加1.如下代码：

```
CG(active_op_array)->num_args++;
```

并且当前参数的索引为CG(active_op_array)->num_args-1 .如下代码：

```
cur_arg_info = &CG(active_op_array)->arg_info[CG(active_op_array)->num_args-1];
```

以上的分析是针对函数定义时的参数设置，这些参数是固定的。而在实际编写程序时可能我们会用到可变参数。此时我们会使用到函数 **func_num_args** 和 **func_get_args**。 它们是以内部函数存在。于是在 Zend\zend_builtin_functions.c 文件中找到这两个函数的实现。首先我们来看func_num_args函数的实现。其代码如下：

```
/* {{{ proto int func_num_args(void)
   Get the number of arguments that were passed to the function */
ZEND_FUNCTION(func_num_args)
{
    zend_execute_data *ex = EG(current_execute_data)->prev_execute_data;

    if (ex && ex->function_state.arguments) {
        RETURN_LONG((long)(zend_uintptr_t)*(ex->function_state.arguments));
    } else {
        zend_error(E_WARNING,
"func_num_args():  Called from the global scope - no function context");
        RETURN_LONG(-1);
    }
```

```
}
/* }}} */
```

在存在 ex->function_state.arguments的情况下，即函数调用时，返回ex->function_state.arguments转化后的值，否则显示错误并返回-1。 这里最关键的一点是EG(current_execute_data)。这个变量存放的是当前执行程序或函数的数据。此时我们需要取前一个执行程序的数据，为什么呢？ 因为这个函数的调用是在进入函数后执行的。函数的相关数据等都在之前执行过程中。于是调用的是：

```
zend_execute_data *ex = EG(current_execute_data)->prev_execute_data;
```

function_state等结构请参照本章第一小节。

在了解func_num_args函数的实现后，func_get_args函数的实现过程就简单了，它们的数据源是一样的， 只是前面返回的是长度，而这里返回了一个创建的数组。数组中存放的是从ex->function_state.arguments转化后的数据。

内部函数的参数

以上我们所说的都是用户自定义函数中对于参数的相关内容。下面我们开始讲解内部函数是如何传递参数的。以常见的count函数为例。其参数处理部分的代码如下：

```
/* {{{ proto int count(mixed var [, int mode])
   Count the number of elements in a variable (usually an array) */
PHP_FUNCTION(count)
{
    zval *array;
    long mode = COUNT_NORMAL;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z|l",
         &array, &mode) == FAILURE) {
        return;
    }
    ... //省略
}
```

这包括了两个操作：一个是取参数的个数，一个是解析参数列表。

取参数的个数

取参数的个数是通过ZEND_NUM_ARGS()宏来实现的。其定义如下：

```
#define ZEND_NUM_ARGS()     (ht)
```

PHP3 中使用的是宏 ARG_COUNT

ht是在 Zend/zend.h文件中定义的宏 **INTERNAL_FUNCTION_PARAMETERS** 中的ht，如下：

```
#define INTERNAL_FUNCTION_PARAMETERS int ht, zval *return_value,
zval **return_value_ptr, zval *this_ptr, int return_value_used TSRMLS_DC
```

解析参数列表

PHP内部函数在解析参数时使用的是 **zend_parse_parameters**。 它可以大大简化参数的接收处理工作，虽然它在处理可变参数时还有点弱。

其声明如下：

```
ZEND_API int zend_parse_parameters(int num_args TSRMLS_DC, char *type_spec,
...)
```

- 第一个参数num_args表明表示想要接收的参数个数，我们经常使用ZEND_NUM_ARGS() 来表示对传入的参数“有多少要多少”。
- 第二参数应该总是宏 TSRMLS_CC 。
- 第三个参数 type_spec 是一个字符串，用来指定我们所期待接收的各个参数的类型，有点类似于 printf 中指定输出格式的那个格式化字符串。
- 剩下的参数就是我们用来接收PHP参数值的变量的指针。

zend_parse_parameters() 在解析参数的同时会尽可能地转换参数类型，这样就可以确保我们总是能得到所期望的类型的变量。 任何一种标量类型都可以转换为另外一种标量类型，但是不能在标量类型与复杂类型（比如数组、对象和资源等）之间进行转换。 如果成功地解析和接收到了参数并且在转换期间也没出现错误，那么这个函数就会返回 SUCCESS，否则返回 FAILURE。 如果这个函数不能接收到所预期的参数个数或者不能成功转换参数类型时就会抛出一些错误信息.

第三个参数指定的各个参数类型列表如下所示：

- l - 长整形 x d - 双精度浮点类型
- s - 字符串 (也可能是空字节)和其长度
- b - 布尔型
- r - 资源, 保存在 zval*
- a - 数组, 保存在 zval*
- o - （任何类的）对象, 保存在 zval *
- O - （由class entry 指定的类的）对象, 保存在 zval *
- z - 实际的 zval*

除了各个参数类型，第三个参数还可以包含下面一些字符，它们的含义如下：

- l - 表明剩下的参数都是可选参数。如果用户没有传进来这些参数值，那么这些值就会被初始化成默认值。
- / - 表明参数解析函数将会对剩下的参数以 SEPARATE_ZVAL_IF_NOT_REF() 的方式来提供这个参数的一份拷贝，除非这些参数是一个引用。
- ! - 表明剩下的参数允许被设定为 NULL（仅用在 a、o、O、r和z身上）。如果用户传进来了一个 NULL 值，则存储该参数的变量将会设置为 NULL。

函数的返回值

在编程语言中，一个函数或一个方法一般都有返回值，但也存在不返回值的情况，此时，这些函数仅仅是处理一些事务， 没有返回，或者说没有明确的返回值，在pascal语言中它有一个专有的关键字

procedure 。 在PHP中，函数都有返回值，分两种情况，使用return语句明确的返回和没有return语句返回NULL。

return语句

当使用return语句时，PHP给用户自定义的函数返回指定类型的变量。 依旧我们查看源码的方式，对return 关键字进行词法分析和语法分析后，生成中间代码。 从 Zend/zend_language_parser.y文件中可以确认其生成中间代码调用的是 **zend_do_return** 函数。

```
void zend_do_return(znode *expr, int do_end_vparse TSRMLS_DC) /* {{{ */
{
    zend_op *opline;
    int start_op_number, end_op_number;

    if (do_end_vparse) {
        if (CG(active_op_array)->return_reference
                && !zend_is_function_or_method_call(expr)) {
            zend_do_end_variable_parse(expr, BP_VAR_W, 0 TSRMLS_CC);/* 处理返回引用 */
        } else {
            zend_do_end_variable_parse(expr, BP_VAR_R, 0 TSRMLS_CC);/* 处理常规变量返回 */
        }
    }

   ...// 省略  取其它中间代码操作

    opline->opcode = ZEND_RETURN;

    if (expr) {
        opline->op1 = *expr;

        if (do_end_vparse && zend_is_function_or_method_call(expr)) {
            opline->extended_value = ZEND_RETURNS_FUNCTION;
        }
    } else {
        opline->op1.op_type = IS_CONST;
        INIT_ZVAL(opline->op1.u.constant);
    }

    SET_UNUSED(opline->op2);
}
/* }}} */
```

生成中间代码为 **ZEND_RETURN** . 第一个操作数的类型在返回值为可用的表达式时，其类型为表达式的操作类型，否则类型为 IS_CONST。这在后续计算执行中间代码函数时有用到。 根据操作数的不同，ZEND_RETURN中间代码会执行ZEND_RETURN_SPEC_CONST_HANDLER,ZEND_RETURN_SPEC_TMP_HANDLER或ZEND_RETURN_SPEC_TMP_HANDLER。 这三个函数的执行流程基本类似，包括对一些错误的处理。这里我们以ZEND_RETURN_SPEC_CONST_HANDLER为例说明函数返回值的执行过程：

```
static int ZEND_FASTCALL
ZEND_RETURN_SPEC_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    zend_op *opline = EX(opline);
```

```
    zval *retval_ptr;
    zval **retval_ptr_ptr;


    if (EG(active_op_array)->return_reference == ZEND_RETURN_REF) {

        //  返回引用时不允许常量和临时变量
        if (IS_CONST == IS_CONST || IS_CONST == IS_TMP_VAR) {
            /* Not supposed to happen, but we'll allow it */
            zend_error(E_NOTICE, "Only variable references \
                should be returned by reference");
            goto return_by_value;
        }

        retval_ptr_ptr = NULL;  //  返回值

        if (IS_CONST == IS_VAR && !retval_ptr_ptr) {
            zend_error_noreturn(E_ERROR, "Cannot return string offsets by
reference");
        }

        if (IS_CONST == IS_VAR && !Z_ISREF_PP(retval_ptr_ptr)) {
            if (opline->extended_value == ZEND_RETURNS_FUNCTION &&
                EX_T(opline->op1.u.var).var.fcall_returned_reference) {
            } else if (EX_T(opline->op1.u.var).var.ptr_ptr ==
                    &EX_T(opline->op1.u.var).var.ptr) {
                if (IS_CONST == IS_VAR && !0) {
                    /* undo the effect of get_zval_ptr_ptr() */
                    PZVAL_LOCK(*retval_ptr_ptr);
                }
                zend_error(E_NOTICE, "Only variable references \
                 should be returned by reference");
                goto return_by_value;
            }
        }

        if (EG(return_value_ptr_ptr)) { //  返回引用
            SEPARATE_ZVAL_TO_MAKE_IS_REF(retval_ptr_ptr);   //  is_ref__gc设置为
1
            Z_ADDREF_PP(retval_ptr_ptr);    //  refcount__gc计数加1

            (*EG(return_value_ptr_ptr)) = (*retval_ptr_ptr);
        }
    } else {
return_by_value:

        retval_ptr = &opline->op1.u.constant;

        if (!EG(return_value_ptr_ptr)) {
            if (IS_CONST == IS_TMP_VAR) {

            }
        } else if (!0) { /* Not a temp var */
            if (IS_CONST == IS_CONST ||
                EG(active_op_array)->return_reference == ZEND_RETURN_REF ||
                (PZVAL_IS_REF(retval_ptr) && Z_REFCOUNT_P(retval_ptr) > 0)) {
                zval *ret;

                ALLOC_ZVAL(ret);
                INIT_PZVAL_COPY(ret, retval_ptr);   //  复制一份给返回值
                zval_copy_ctor(ret);
                *EG(return_value_ptr_ptr) = ret;
            } else {
                *EG(return_value_ptr_ptr) = retval_ptr; //  直接赋值
```

```
                Z_ADDREF_P(retval_ptr);
            }
        } else {
            zval *ret;

            ALLOC_ZVAL(ret);
            INIT_PZVAL_COPY(ret, retval_ptr);     //  复制一份给返回值
            *EG(return_value_ptr_ptr) = ret;
        }
    }

    return zend_leave_helper_SPEC(ZEND_OPCODE_HANDLER_ARGS_PASSTHRU);    //  返回前执行收尾工作
}
```

函数的返回值在程序执行时存储在 *EG(return_value_ptr_ptr)。ZE内核对值返回和引用返回作了区分， 并且在此基础上对常量，临时变量和其它类型的变量在返回时进行了不同的处理。在return执行完之前，ZE内核通过调用zend_leave_helper_SPEC函数，清除函数内部使用的变量等。 这也是ZE内核自动给函数加上NULL返回的原因之一。

没有return语句的函数

在PHP中，没有过程这个概念，只有没有返回值的函数。但是对于没有返回值的函数，PHP内核会“帮你“加上一个NULL来做为返回值。 这个“帮你”的操作也是在生成中间代码时进行的。在每个函数解析时都需要执行函数 **zend_do_end_function_declaration**， 在此函数中有一条语句:

```
zend_do_return(NULL, 0 TSRMLS_CC);
```

结合前面的内容，我们知道这条语句的作用就是返回NULL。这就是没有return语句的函数返回NULL的原因所在。

内部函数的返回值

内部函数的返回值都是通过一个名为 return_value 的变量传递的。 这个变量同时也是函数中的一个参数，在PHP_FUNCTION函数扩展开来后可以看到。 这个参数总是包含有一个事先申请好空间的 zval 容器，因此你可以直接访问其成员并对其进行修改而无需先对 return_value 执行一下 MAKE_STD_ZVAL 宏指令。 为了能够更方便从函数中返回结果，也为了省却直接访问 zval 容器内部结构的麻烦，ZEND 提供了一大套宏命令来完成相关的这些操作。 这些宏命令会自动设置好类型和数值。

从函数直接返回值的宏:

- RETURN_RESOURCE(resource) 返回一个资源。
- RETURN_BOOL(bool) 返回一个布尔值。
- RETURN_NULL() 返回一个空值。
- RETURN_LONG(long) 返回一个长整数。
- RETURN_DOUBLE(double) 返回一个双精度浮点数。
- RETURN_STRING(string, duplicate) 返回一个字符串。duplicate 表示这个字符是否使用 estrdup() 进行复制。

- RETURN_STRINGL(string, length, duplicate) 返回一个定长的字符串。其余跟 RETURN_STRING 相同。这个宏速度更快而且是二进制安全的。
- RETURN_EMPTY_STRING() 返回一个空字符串。
- RETURN_FALSE 返回一个布尔值假。
- RETURN_TRUE 返回一个布尔值真。

设置函数返回值的宏：

- RETVAL_RESOURCE(resource) 设定返回值为指定的一个资源。
- RETVAL_BOOL(bool) 设定返回值为指定的一个布尔值。
- RETVAL_NULL 设定返回值为空值
- RETVAL_LONG(long) 设定返回值为指定的一个长整数。
- RETVAL_DOUBLE(double) 设定返回值为指定的一个双精度浮点数。
- RETVAL_STRING(string, duplicate) 设定返回值为指定的一个字符串，duplicate 含义同 RETURN_STRING。
- RETVAL_STRINGL(string, length, duplicate) 设定返回值为指定的一个定长的字符串。其余跟 RETVAL_STRING 相同。这个宏速度更快而且是二进制安全的。
- RETVAL_EMPTY_STRING 设定返回值为空字符串。
- RETVAL_FALSE 设定返回值为布尔值假。
- RETVAL_TRUE 设定返回值为布尔值真。

如果需要返回的是像数组和对象这样的复杂类型的数据，那就需要先调用 array_init() 和 object_init()，也可以使用相应的 hash 函数直接操作 return_value。由于这些类型主要是由一些杂七杂八的东西构成，所以对它们就没有了相应的宏。

关于内部函数的return_value值是如何赋值给*EG(return_value_ptr_ptr)，函数的调用是如何进行的，请阅读下一小节 <<函数的调用和执行>>.

第三节 函数的调用和执行

前面小节中对函数的内部表示以及参数的传递，返回值都有了介绍，那函数是怎么被调用的呢？内置函数和用户定义函数在调用时会有什么不一样呢？ 下面将介绍函数调用和执行的过程。

函数的调用

函数被调用需要一些基本的信息，比如函数的名称，参数以及函数的定义(也就是最终函数是怎么执行的), 从我们开发者的角度来看， 定义了一个函数我们在执行的时候自然知道这个函数叫什么名字，以及调用的时候给传递了什么参数，以及函数是怎么执行的。 但是对于Zend引擎来说，它并不能像我们这样能“看懂”php源代码，他们需要对代码进行处理以后才能执行。我们还是从以下两个小例子开始：

```
<?php
    function foo(){
        echo "I'm foo!";
    }
    foo();
?>
```

下面我们先看一下其对应的opcodes:

```
function name:  (null)
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------
--
             DO_FCALL                                       0          'foo'
             NOP
           > RETURN                                                     1

function name:  foo
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------
--
   4     0  >   ECHO
'I%27m+foo%21'
   5     1    > RETURN                                                   null
```

上面是去除了一些枝节信息的的opcodes，可以看到执行时函数部分的opcodes是单独独立出来的，这点对于函数的执行特别重要，下面的部分会详细介绍。 现在，我们把焦点放到对foo函数的调用上面。调用foo的OPCODE是“DO_FCALL“， DO_FCALL进行函数调用操作时，ZE会在function_table中根据函数名（如前所述，这里的函数名经过str_tolower的处理，所以PHP的函数名大小写不敏感)查找函数的定义， 如果不存在， 则报出“Call to undefined function xxx()"的错误信息; 如果存在，就返回该函数zend_function□结构指针, 然后通过function.type的值来判断函数是内部函数还是用户定义的函数, 调用zend_execute_internal（zend_internal_function.handler）或者直接 调用zend_execute来执行这个函数包含的zend_op_array。

函数的执行

细心的读者可能会注意到上面opcodes里函数被调用的时候以及函数定义那都有个"function name:", 其实用户定义函数的执行与其他语句的执行并无区别， 在本质上看，其实函数中的php语句与函数外的php语句并无不同。函数体本身最大的区别，在于其执行环境的不同。 这个“执行环境”最重要的特征就是变量的作用域。大家都知道，函数内定义的变量在函数体外是无法直接使用的，反之也是一样。那么，在函数执行的时候， 进入函数前的环境信息是必须要保存的。在函数执行完毕后，这些环境信息也会被还原，使整个程序继续的执行下去。

内部函数的执行与用户函数不同。用户函数是php语句一条条“翻译”成op_line组成的一个op_array,而内部函数则是用C来实现的，因为执行环境也是C环境， 所以可以直接调用。如下面的例子:

```
[php]
<?php
    $foo = 'test';
    print_r($foo);
?>
```

对应的opcodes也很简单:

```
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------
--
   2     0  >   ASSIGN                                                 !0,
```

```
'test'
   3     1       SEND_VAR                                                !0
         2       DO_FCALL                                    1
'print_r'
   4     3     > RETURN                                                  1
```

可以看出，生成的opcodes中，内部函数和用户函数的处理都是由DO_FCALL来进行的。而在其具体实现的zend_do_fcall_common_helper_SPEC()中， 则对是否为内部函数进行了判断，如果是内部函数，则使用一个比较长的调用

```
((zend_internal_function *) EX(function_state).function)->handler(opline-
>extended_value, EX_T(opline->result.u.var).var.ptr,
EX(function_state).function->common       .return_reference?&EX_T(opline-
>result.u.var).var.ptr:NULL, EX(object), RETURN_VALUE_USED(opline) TSRMLS_CC);
```

上面这种方式的内部函数是在zend_execute_internal函数没有定义的情况下。而在而在Zend/zend.c文件的zend_startup函数中，

```
zend_execute_internal = NULL;
```

此函数确实被赋值为NULL。于是我们在if (!zend_execute_internal)判断时会成立，所以我们是执行那段很长的调用。 那么，这段很长的调用到底是什么呢？以我们常用的 **count**函数为例。在<<第一节 函数的内部结构>>中，我们知道内部函数所在的结构体中 有一个handler指针指向此函数需要调用的内部定义的C函数。这些内部函数在模块初始化时就以扩展的函数的形式加载到EG(function_table)。其调用顺序：

```
php_module_startup --> php_register_extensions -->
zend_register_internal_module
--> zend_register_module_ex --> zend_register_functions

zend_register_functions(NULL, module->functions, NULL, module->type TSRMLS_CC)
```

在standard扩展中。module的定义为：

```
zend_module_entry basic_functions_module = { /* {{{ */
    STANDARD_MODULE_HEADER_EX,
    NULL,
    standard_deps,
    "standard",                 /* extension name */
    basic_functions,            /* function list */
    ... //省略
}
```

从上面的代码可以看出，module->functions是指向basic_functions。在basic_functions.c文件中查找basic_functions的定义。

```
const zend_function_entry basic_functions[] = { /* {{{ */
    ...//   省略
    PHP_FE(count,
arginfo_count)
    ...//省略
}

#define PHP_FE          ZEND_FE
```

```
#define ZEND_FE(name, arg_info)                         ZEND_FENTRY(name,
ZEND_FN(name), arg_info, 0)
#define ZEND_FN(name) zif_##name
#define ZEND_FENTRY(zend_name, name, arg_info, flags)    { #zend_name, name,
arg_info, (zend_uint) (sizeof(arg_info)/sizeof(struct _zend_arg_info)-1), flags
},
```

综合上面的代码，count函数最后调用的函数名为zif_count，但是此函数对外的函数名还是为count。调用的函数名name以第二个元素存放在zend_function_entry结构体数组中。 对于zend_function_entry的结构

```
typedef struct _zend_function_entry {
    const char *fname;
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
    const struct _zend_arg_info *arg_info;
    zend_uint num_args;
    zend_uint flags;
} zend_function_entry;
```

第二个元素为handler。这也就是我们在执行内部函数时的调用方法。因此在执行时就会调用到对应的函数。

对于用户定义的函数，在zend_do_fcall_common_helper_SPEC()函数中，

```
if (EX(function_state).function->type == ZEND_USER_FUNCTION ||
    EX(function_state).function->common.scope) {
    should_change_scope = 1;
    EX(current_this) = EG(This);
    EX(current_scope) = EG(scope);
    EX(current_called_scope) = EG(called_scope);
    EG(This) = EX(object);
    EG(scope) = (EX(function_state).function->type == ZEND_USER_FUNCTION ||
!EX(object)) ? EX(function_state).function->common.scope : NULL;
    EG(called_scope) = EX(called_scope);
}
```

先将EG下的This,scope等暂时缓存起来（这些在后面会都恢复到此时缓存的数据）。在此之后，对于用户自定义的函数， 程序会依据zend_execute是否等于execute并且是否为异常来判断是返回，还是直接执行函数定义的op_array：

```
if (zend_execute == execute && !EG(exception)) {
        EX(call_opline) = opline;
        ZEND_VM_ENTER();
    } else {
        zend_execute(EG(active_op_array) TSRMLS_CC);
    }
```

而在Zend/zend.c文件的zend_startup函数中，已将zend_execute赋值为：

```
zend_execute = execute;
```

从而对于异常，程序会抛出异常；其它情况，程序会调用execute执行此函数中生成的opcodes。execute函数会遍历所传递给它的zend_op_array数组，以方式

```
ret = EX(opline)->handler(execute_data TSRMLS_CC)
```

调用每个opcode的处理函数。而execute_data在execute函数开始时就已经给其分配了空间，这就是这个函数的执行环境。

第四节 匿名函数及闭包

匿名函数在编程语言中出现的比较早,最早出现在Lisp语言中, 随后很多的编程语言都开始有这个功能了, 目前使用比较广泛的Javascript以及C#, PHP直到5.3才开始真正支持匿名函数, C++的新标准C++0x也开始支持了.

匿名函数是一类不需要指定表示符, 而又可以被调用的函数或子例程, 匿名函数可以方便的作为参数传递给其他函数, 最常见应用是作为回调函数.

闭包(Closure)

说到匿名函数, 就不得不提到闭包了, 闭包是词法闭包(Lexical Closure)的简称, 是引用了自由变量的函数, 这个被应用的自由变量将和这个函数一同存在, 即使离开了创建它的环境也一样, 所以闭包也可认为是有函数和与其相关引用组合而成的实体. 在一些语言中, 在函数内定义另一个函数的时候, 如果内部函数引用到外部函数的变量, 则可能产生闭包. 在运行外部函数时, 一个闭包就形成了.

这个词和匿名函数很容易被混用, 其实这是两个不同的概念, 这可能是因为很多语言实现匿名函数的时候允许形成闭包.

使用create_function()创建"匿名"函数

前面提到PHP5.3中才才开始正式支持匿名函数, 说到这里可能会有细心读者有意见了, 因为有个函数是可以生成匿名函数的: create_function函数, 在手册里可以查到这个函数在PHP4.1和PHP5中就有了, 这个函数通常也能作为匿名回调函数使用, 例如如下:

```php
<?php

$array = array(1, 2, 3, 4);
array_walk($array, create_function('$value', 'echo $value'));
```

这段代码只是将数组中的值依次输出, 当然也能做更多的事情. 那为什么这不算真正的匿名函数呢, 我们先看看这个函数的返回值, 这个函数返回一个字符串, 通常我们可以像下面这样调用一个函数:

```php
<?php

function a() {
    echo 'function a';
}

$a = 'a';
$a();
```

我们在实现回调函数的时候也可以采用这样的方式, 例如:

```
<?php

function do_something($callback) {
    // doing
    # ...

    // done
    $callback();
}
```

这样就能实现在函数do_something()执行完成之后调用$callback指定的函数. 回到create_function函数的返回值: 函数返回一个唯一的字符串函数名, 出现错误的话则返回FALSE. 这么说这个函数也只是动态的创建了一个函数, 而这个函数是**有函数名**的, 也就是说, 其实这并不是匿名的. 只是创建了一个全局唯一的函数而已.

```
<?php
$func = create_function('', 'echo "Function created dynamic";');
echo $func; // lambda_1

$func();    // Function created dynamic

$my_func = 'lambda_1';
$my_func(); // 不存在这个函数
lambda_1(); // 不存在这个函数
```

上面这段代码的前面很好理解, create_function就是这么用的, 后面指定函数名调用却失败了, 这就有些不好理解了, php是怎么保证这个函数是全局唯一的? lambda_1看起来也是一个很普通的函数明, 如果我们先定义一个叫做lambda_1的函数呢? 这里函数的返回字符串会是lambda_2, 它在创建函数的时候会检查是否这个函数是否存在知道找到合适的函数名, 但如果我们在create_function之后定义一个叫做lambda_1的函数会怎么样呢? 这样就出现函数重复定义的问题了, 这样的实现恐怕不是最好的方法, 实际上如果你真的定义了名为lambda_1的函数也是不会出现我所说的问题的. 这究竟是怎么回事呢? 上面代码的倒数2两行也说明了这个问题, 实际上并没有定义名为lambda_1的函数.

也就是说我们的lambda_1和create_function返回的lambda_1并不是一样的!? 怎么会这样呢? 那只能说明我们没有看到实质, 只看到了表面, 表面是我们在echo的时候输出了lambda_1, 而我们的lambda_1是我们自己敲入的. 我们还是使用debug_zval_dump函数来看看吧.

```
<?php
$func = create_function('', 'echo "Hello";');

$my_func_name = 'lambda_1';
debug_zval_dump($func);          // string(9) "lambda_1" refcount(2)
debug_zval_dump($my_func_name); // string(8) "lambda_1" refcount(2)
```

看出来了吧, 他们的长度居然不一样, 长度不一样, 所以我们调用的函数当然是不存在的, 我们还是直接看看create_function函数到底都做了些什么吧. 该实现见: $PHP_SRC/Zend/zend_builtin_functions.c

```
#define LAMBDA_TEMP_FUNCNAME    "__lambda_func"

ZEND_FUNCTION(create_function)
```

```
{
    // ... 省去无关代码
    function_name = (char *) emalloc(sizeof("0lambda_")+MAX_LENGTH_OF_LONG);
    function_name[0] = '\0';  // <--- 这里
    do {
        function_name_length = 1 + sprintf(function_name + 1, "lambda_%d",
++EG(lambda_count));
    } while (zend_hash_add(EG(function_table), function_name,
function_name_length+1, &new_function, sizeof(zend_function), NULL)==FAILURE);
    zend_hash_del(EG(function_table), LAMBDA_TEMP_FUNCNAME,
sizeof(LAMBDA_TEMP_FUNCNAME));
    RETURN_STRINGL(function_name, function_name_length, 0);
}
```

该函数在定义了一个函数之后, 给函数起了个名字, 它将函数名的第一个字符变为了'\0'也就是空字符, 然后在函数表中查找是否已经定义了这个函数, 如果已经有了则生成新的函数名, 第一个字符为空字符的定义方式比较特殊, 这样在用户代码中就无法定义出这样的函数了, 这样也就不存在命名冲突的问题了, 这也算是种取巧的做法了, 在了解到这个特殊的函数之后,我们其实还是可以调用到这个函数的, 只要我们在函数名前加一个空字符就可以了, chr()函数可以帮我们生成这样的字符串, 例如前面创建的函数可以通过如下的方式访问到:

```
<?php

$my_func = chr(0) . "lambda_1";
$my_func(); // Hello
```

这种创建"匿名函数"的方式有一些缺点:

1. 函数的定义是通过字符串动态eval的, 这就无法进行基本的语法检查;
2. 这类函数和普通函数没有本质区别, 无法实现闭包的效果.

真正的匿名函数

在PHP5.3引入的众多功能中, 除了匿名函数还有一个特性值得讲讲: 新引入的__invoke 魔幻方法.

__invoke魔幻方法

这个魔幻方法被调用的时机是: 当一个对象当做函数调用的时候, 如果对象定义了__invoke魔幻方法则这个函数会被调用, 这和C++中的操作符重载有些类似, 例如可以像下面这样使用:

```
<?php
class Callme {
    public function __invoke($phone_num) {
        echo "Hello: $num";
    }
}

$call = new Callme();
$call(13810688888); // "Hello: 13810688888
```

匿名函数的实现

前面介绍了将对象作为函数调用的方法, 聪明的你可能想到在PHP实现匿名函数的方法了, PHP中的匿名函数就的确是通过这种方式实现的. 我们先来验证一下:

```
<?php
$func = function() {
    echo "Hello, anonymous function";
}

echo gettype($func);    // object
echo get_class($func);  // Closure
```

原来匿名函数也只是一个普通的类而已. 熟悉Javascript的同学对匿名函数的使用方法很熟悉了, PHP也使用和Javascript类似的语法来定义, 匿名函数可以赋值给一个变量, 因为匿名函数其实是一个类实例, 所以能复制也是很容易理解的, 在Javascript中可以将一个匿名函数赋值给一个对象的属性, 例如:

```
var a = {};
a.call = function() {alert("called");}
a.call(); // alert called
```

这在Javascript中很常见, 但在PHP中这样并不可以, 给对象的属性复制是不能被调用的, 这样使用将会导致类寻找类中定义的方法, 在PHP中属性名和定义的方法名是可以重复的, 这是由PHP的类模型所决定的, 当然PHP在这方面是可以改进的, 后续的版本中可能会允许这样的调用, 这样的话就更容易灵活的实现一些功能了. 目前想要实现这样的效果也是有方法的: 使用另外一个魔幻方法__call(), 至于怎么实现就留给各位读者当做习题吧.

闭包的使用

PHP使用闭包(Closure)来实现匿名函数, 匿名函数最强大的功能也就在匿名函数所提供的一些动态特性以及闭包效果, 匿名函数在定义的时候如果需要使用作用域外的变量需要使用如下的语法来实现:

```
<?php
$name = 'TIPI Tea';
$func = function() use($name) {
    echo "Hello, $name";
}

$func(); // Hello TIPI Team
```

这个use语句看起来挺别扭的, 尤其是和Javascript比起来, 不过这也应该是PHP-Core综合考虑才使用的语法, 因为和Javascript的作用域不同, PHP在函数内定义的变量默认就是局部变量, 而在Javascript中则相反, 除了显式定义的才是局部变量, PHP在变异的时候则无法确定变量是局部变量还是上层作用域内的变量, 当然也可能有办法在编译时确定, 不过这样对于语言的效率和复杂性就有很大的影响.

这个语法比较直接, 如果需要访问上层作用域内的变量则需要使用use语句来申明, 这样也简单易读, 说到这里, 其实可以使用use来实现类似global语句的效果.

匿名函数在每次执行的时候都能访问到上层作用域内的变量, 这些变量在匿名函数被销毁之前始终保存

着自己的状态, 例如如下的例子:

```
<?php
function getCounter() {
    $i = 0;
    return function() use($i) { // 这里如果使用引用传入变量: use(&$i)
        echo ++$i;
    };
}

$counter = getCounter();
$counter(); // 1
$counter(); // 1
```

和Javascript中不同, 这里两次函数调用并没有使$i变量自增, 默认PHP是通过拷贝的方式传入上层变量进入匿名函数, 如果需要改变上层变量的值则需要通过引用的方式传递. 所以上面得代码没有输出1, 2而是1, 1.

闭包的实现

前面提到匿名函数是通过闭包来实现的, 现在我们开始看看闭包(类)是怎么实现的. 匿名函数和普通函数除了是否有变量名以外并没有区别, 闭包的实现代码在$PHP_SRC/Zend/zend_closure.c. 匿名函数"对象化"的问题已经通过Closure实现, 而对于匿名是怎么样访问到创建该匿名函数时的变量的呢?

例如如下这段代码:

```
<?php
$i=100;
$counter = function() use($i) {
    debug_zval_dump($i);
};

$counter();
```

通过VLD来查看这段编码编译什么样的opcode了

```
$ php -dvld.active=1 closure.php

vars:  !0 = $i, !1 = $counter
# *  op                            fetch          ext  return  operands
---------------------------------------------------------------------------
0  >   ASSIGN                                                   !0, 100
1      ZEND_DECLARE_LAMBDA_FUNCTION                             '%00%7Bclosure
2      ASSIGN                                                   !1, ~1
3      INIT_FCALL_BY_NAME                                       !1
4      DO_FCALL_BY_NAME                              0
5    > RETURN                                                   1

function name:  {closure}
number of ops:  5
compiled vars:  !0 = $i
line     # *  op                            fetch          ext  return  operands
---------------------------------------------------------------------------
-
  3     0  >   FETCH_R                        static              $0      'i'
```

```
        1      ASSIGN                                                   !0, $0
  4     2      SEND_VAR                                                 !0
        3      DO_FCALL                                      1
'debug_zval_dump'
  5     4    > RETURN                                                   null
```

上面根据情况去掉了一些无关的输出, 从上到下, 第1开始将100赋值给!0也就是变量$i, 随后执行ZEND_DECLARE_LAMBDA_FUNCTION, 那我们去相关的opcode执行函数中看看这里是怎么执行的, 这个opcode的处理函数位于$PHP_SRC/Zend/zend_vm_execute.h中:

```
static int ZEND_FASTCALL
ZEND_DECLARE_LAMBDA_FUNCTION_SPEC_CONST_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    zend_op *opline = EX(opline);
    zend_function *op_array;

    if (zend_hash_quick_find(EG(function_table), Z_STRVAL(opline-
>op1.u.constant), Z_STRLEN(opline->op1.u.constant), Z_LVAL(opline-
>op2.u.constant), (void *) &op_arra
y) == FAILURE ||
        op_array->type != ZEND_USER_FUNCTION) {
        zend_error_noreturn(E_ERROR, "Base lambda function for closure not
found");
    }

    zend_create_closure(&EX_T(opline->result.u.var).tmp_var, op_array
TSRMLS_CC);

    ZEND_VM_NEXT_OPCODE();
}
```

该函数调用了zend_create_closure()函数来创建一个闭包对象, 那我们继续看看位于$PHP_SRC/Zend/zend_closures.c的zend_create_closure()函数都做了些什么.

```
ZEND_API void zend_create_closure(zval *res, zend_function *func TSRMLS_DC)
{
    zend_closure *closure;

    object_init_ex(res, zend_ce_closure);

    closure = (zend_closure *)zend_object_store_get_object(res TSRMLS_CC);

    closure->func = *func;

    if (closure->func.type == ZEND_USER_FUNCTION) { // 如果是用户定义的匿名函数
        if (closure->func.op_array.static_variables) {
            HashTable *static_variables = closure-
>func.op_array.static_variables;

            // 为函数申请存储静态变量的哈希表空间
            ALLOC_HASHTABLE(closure->func.op_array.static_variables);
            zend_hash_init(closure->func.op_array.static_variables,
zend_hash_num_elements(static_variables), NULL, ZVAL_PTR_DTOR, 0);

            // 循环当前静态变量列表, 使用zval_copy_static_var方法处理
            zend_hash_apply_with_arguments(static_variables TSRMLS_CC,
(apply_func_args_t)zval_copy_static_var, 1, closure-
>func.op_array.static_variables);
        }
```

```
        (*closure->func.op_array.refcount)++;
    }

    closure->func.common.scope = NULL;
}
```

如上段代码注释中所说, 继续看看zval_copy_static_var()函数的实现:

```
static int zval_copy_static_var(zval **p TSRMLS_DC, int num_args, va_list args,
zend_hash_key *key) /* {{{ */
{
    HashTable *target = va_arg(args, HashTable*);
    zend_bool is_ref;

    // 只对通过use语句类型的静态变量进行取值操作, 否则匿名函数体内的静态变量也会影响到作
用域之外的变量
    if (Z_TYPE_PP(p) & (IS_LEXICAL_VAR|IS_LEXICAL_REF)) {
        is_ref = Z_TYPE_PP(p) & IS_LEXICAL_REF;

        if (!EG(active_symbol_table)) {
            zend_rebuild_symbol_table(TSRMLS_C);
        }
        // 如果当前作用域内没有这个变量
        if (zend_hash_quick_find(EG(active_symbol_table), key->arKey, key-
>nKeyLength, key->h, (void **) &p) == FAILURE) {
            if (is_ref) {
                zval *tmp;

                // 如果是引用变量, 则创建一个零时变量一边在匿名函数定义之后对该变量进行
操作
                ALLOC_INIT_ZVAL(tmp);
                Z_SET_ISREF_P(tmp);
                zend_hash_quick_add(EG(active_symbol_table), key->arKey, key-
>nKeyLength, key->h, &tmp, sizeof(zval*), (void**)&p);
            } else {
                // 如果不是引用则表示这个变量不存在
                p = &EG(uninitialized_zval_ptr);
                zend_error(E_NOTICE,"Undefined variable: %s", key->arKey);
            }
        } else {
            // 如果存在这个变量, 则根据是否是引用, 对变量进行引用或者复制
            if (is_ref) {
                SEPARATE_ZVAL_TO_MAKE_IS_REF(p);
            } else if (Z_ISREF_PP(p)) {
                SEPARATE_ZVAL(p);
            }
        }
    }
    if (zend_hash_quick_add(target, key->arKey, key->nKeyLength, key->h, p,
sizeof(zval*), NULL) == SUCCESS) {
        Z_ADDREF_PP(p);
    }
    return ZEND_HASH_APPLY_KEEP;
}
```

这个函数作为一个回调函数传递给zend_hash_apply_with_arguments()函数, 每次读取到hash表中的值之后由这个函数进行处理, 而这个函数对所有use语句定义的变量值赋值给这个匿名函数的静态变量, 这样匿名函数就能访问到use的变量了.

第五节 小结

本章从函数的内部结构开始, 介绍了在PHP中, 函数的各种不同内部实现, 例如用户定义的函数和模块提供的函数在内部的表示, 函数信息中通常包括了函数的名称, 所能接受的参数信息等, 用户定义的函数则会包括该函数编译好的op_array信息, 以便在函数执行的时候能将用户代码执行, 而内部函数则指向一个内部函数的结构, 内部函数最主要的信息是这个函数的C实现函数指针, 也就是在真正执行这个函数时所需要执行的C函数.

随后我们介绍了函数在定义以及传递参数方面的具体实现, 分别介绍了内部函数和用户函数的不同实现, 以及这两种不同类型的函数是怎么样将函数返回值返回的. 参数传递完成后, 函数是怎么执行的? 这就是第三小节主要介绍的内容了, 说到最后还介绍了PHP5.3中新引入的匿名函数的特性及它的内部实现.

第五章 类和面向对象

从我们接触PHP开始，我们最先遇到的是函数，数组操作函数，字符串操作函数，文件操作函数等等。 这些函数是我们使用PHP的基础，也是PHP自出生就支持的面向过程编程。面向过程是将一个个功能封装，以一种模块化的思想解决问题。 在PHP中除了面向过程的编程，还支持面向对象(Object Oriented,OO)的编程。这个从PHP4起就开始支持，只是这个时间段对面向对象的支持不太好。 从PHP5起，PHP引入了新的对象模型（Object Model），并且完全重写了 PHP 处理对象的方式， 增加了许多新特性，包括访问控制,抽象类和final类、类方法, 魔术方法, 接口, 对象克隆和类型提示等。 并且在近期发布的PHP5.3版本中，针对面向对象编程增加了命名空间、延迟静态绑定（Late Static Binding）以及增加了两个魔术方法__callStatic()和__invoke() PHP当中对象是按引用传递的，即每个包含对象的变量都持有对象的引用（reference），而不是整个对象的拷贝。 自此PHP对于面向对象的支持越发的成熟。

这一章我们从面向对象讲起，会说到PHP中的类，包括类的定义和实现、接口、抽象类以及与类相关的访问控制、对象和命名空间等。 除此之外也会从其存储的内部结构，类的单继承的实现，接口的多继承，以及魔法方法的实现等细微处着手分析类相关的方方面面。 首先我们来看第一小节--类的结构和实现。

第一节 类的结构和实现

在面向对象编程(OOP)中,我们最先接触的概念应该就是类了（不过在学习的过程中也会有人先接触对象这个概念的）. 在平常的工作中,我们经常需要写类,设计类等等. 那么,类是什么？在PHP中,类是以哪种方式存储的?

类的结构

首先,我们看下类是什么. 类是用户定义的一种抽象数据类型,它是现实世界中某些具有共性事物的抽象. 有时,我们也可以理解其为对象的类别. 类也可以看作是一种复合型的结构,其需要存储多元化的数据,如属性,方法,以及自身的一些性质等.

在PHP中,类的定义以class关键字开始,后面接类名,类名可以是任何非PHP保留字的名字. 在类名后面紧跟着一对花括号,这个里面就是类的实体了,它包括类所具有的属性,这些属性是对象的状态的抽象,其表现为PHP中支持的数据类型,也可以包括对象本身,通常我们称其为成员变量. 除了类的属性,类的实体中也包括类所具有的操作,这些操作是对象的行为的抽象,其表现为用操作名和实现该操作的方法,通常我们称其为成员方法或成员函数. 看一个PHP写的类示例的代码:

```
class ParentClass {
}

interface Ifce {
        public function iMethod();
}

final class Tipi extends ParentClass implements Ifce{
        public static $sa = 'aaa';
        const CA = 'bbb';
```

```
        public function __constrct() {
        }

        public function iMethod() {
        }

        private function _access() {
        }

        public static function access() {
        }
}
```

这里定义了一个父类ParentClass,一个接口Ifce,一个子类Tipi. 子类继承父类ParentClass, 实现接口Ifce,并且有一个静态变量$sa,一个类常量 CA,一个公用方法,一个私有方法和一个公用静态方法. 这些也许已经比较熟悉了,那么这些结构在Zend引擎内部是如何实现的? 类的这些方法、成员变量是如何存储的? 这些访问控制,静态成员是如何标记的?

首先,我们看下类的存储结构. 我们在PHP的源码中很容易找到类的结构存放在zend_class_entry结构体中,这个结构体在PHP源码中出现的频率很高.

```
    struct _zend_class_entry {
        char type;     // 类型: ZEND_INTERNAL_CLASS / ZEND_USER_CLASS
        char *name;// 类名称
        zend_uint name_length;                  // 即sizeof(name) - 1
        struct  _zend_class_entry *parent; // 继承的父类
        int  refcount;  // 引用数
        zend_bool constants_updated;

        zend_uint ce_flags; // ZEND_ACC_IMPLICIT_ABSTRACT_CLASS: 类存在abstract
方法
        // ZEND_ACC_EXPLICIT_ABSTRACT_CLASS: 在类名称前加了abstract关键字
        // ZEND_ACC_FINAL_CLASS
        // ZEND_ACC_INTERFACE
        HashTable function_table;       // 方法
        HashTable default_properties;           // 默认属性
        HashTable properties_info;      // 属性信息
        HashTable default_static_members;// 静态变量
        HashTable *static_members; // type == ZEND_USER_CLASS时,取
&default_static_members;
        // type == ZEND_INTERAL_CLASS时,设为NULL
        HashTable constants_table;      // 常量
        struct _zend_function_entry *builtin_functions;// 方法定义入口


        union _zend_function *constructor;
        union _zend_function *destructor;
        union _zend_function *clone;


        /* 魔术方法 */
        union _zend_function *__get;
        union _zend_function *__set;
        union _zend_function *__unset;
        union _zend_function *__isset;
        union _zend_function *__call;
        union _zend_function *__tostring;
        union _zend_function *serialize_func;
        union _zend_function *unserialize_func;
        zend_class_iterator_funcs iterator_funcs;// 迭代
```

```
        /* 类句柄 */
        zend_object_value (*create_object)(zend_class_entry *class_type
TSRMLS_DC);
        zend_object_iterator *(*get_iterator)(zend_class_entry *ce, zval
*object,
            intby_ref TSRMLS_DC);

        /* 类声明的接口 */
        int(*interface_gets_implemented)(zend_class_entry *iface,
                zend_class_entry *class_type TSRMLS_DC);


        /* 序列化回调函数指针 */
        int(*serialize)(zval *object, unsignedchar**buffer, zend_uint *buf_len,
                 zend_serialize_data *data TSRMLS_DC);
        int(*unserialize)(zval **object, zend_class_entry
*ce,constunsignedchar*buf,
                zend_uint buf_len, zend_unserialize_data *data TSRMLS_DC);


        zend_class_entry **interfaces;  //  类实现的接口
        zend_uint num_interfaces;   //  类实现的接口数


        char *filename; //  类的存放文件地址 绝对地址
        zend_uint line_start;   //  类定义的开始行
        zend_uint line_end; //  类定义的结束行
        char *doc_comment;
        zend_uint doc_comment_len;


        struct _zend_module_entry *module; // 类所在的模块入口:
EG(current_module)
    };
```

取上面这个结构的部分字段,我们分析文章最开始的那段PHP代码在内核中的表现. 如表5.1所示:

字段名	字段说明	ParentClass类	Ifce接口	Tipi类
name	类名	ParentClass	Ifce	Tipi
type	类别	2	2	2
parent	父类	空	空	ParentClass类
refcount	引用计数	1	1	2
ce_flags	类的类型	0	144	524352
function_table	函数列表	空	• function_name=iMethod | type=2 | fn_flags=258	• function_name=__construct | type=2 | fn_flags=8448 • function_name=iMethod | type=2 | fn_flags=65800 • function_name=_access | type=2 | fn_flags=66560 • function_name=access |

type=2 | fn_flags=257□

interfaces	接口列表	空	空	Ifce接口 接口数为1
filename□	存放文件地址	/tipi.php	/tipi.php	/ipi.php
line_start	类开始行数	15	18	22
line_end	类结束行数	16	20	38

类的结构中,type有两种类型,数字标记为1和2.在代码中体现为:

```
#define ZEND_INTERNAL_CLASS          1
#define ZEND_USER_CLASS              2
```

对于父类和接口,都是以 **struct _zend_class_entry** 存在. 常规的成员方法以HashTable的方式存放在函数结构体中,而魔术方法则单独存在.

类的实现

类的定义是以class关键字开始,在Zend/zend_language_scanner.l文件中,找到class对应的token为T_CLASS. 根据此token,在Zend/zend_language_parser.y文件中,找到编译时调用的函数:

```
unticked_class_declaration_statement:
        class_entry_type T_STRING extends_from
            { zend_do_begin_class_declaration(&$1, &$2, &$3 TSRMLS_CC); }
            implements_list
            '{'
                class_statement_list
            '}' { zend_do_end_class_declaration(&$1, &$2 TSRMLS_CC); }
    |   interface_entry T_STRING
            { zend_do_begin_class_declaration(&$1, &$2, NULL TSRMLS_CC); }
            interface_extends_list
            '{'
                class_statement_list
            '}' { zend_do_end_class_declaration(&$1, &$2 TSRMLS_CC); }
;


class_entry_type:
        T_CLASS         { $$.u.opline_num = CG(zend_lineno); $$.u.EA.type = 0;
}
    |   T_ABSTRACT T_CLASS { $$.u.opline_num = CG(zend_lineno); $$.u.EA.type =
ZEND_ACC_EXPLICIT_ABSTRACT_CLASS; }
    |   T_FINAL T_CLASS { $$.u.opline_num = CG(zend_lineno); $$.u.EA.type =
ZEND_ACC_FINAL_CLASS; }
;
```

上面的class_entry_type语法说明在语法分析阶段将类分为三种类型：常规类(T_CLASS),抽象类(T_ABSTRACT T_CLASS)和final类(T_FINAL T_CLASS). 这是以在类名前加不同的关键字来. 他们分别对□应的类型在内核中的体现为:

- 常规类(T_CLASS) 对应的type=0
- 抽象类(T_ABSTRACT T_CLASS) 对应type=ZEND_ACC_EXPLICIT_ABSTRACT_CLASS
- final类(T_FINAL T_CLASS) 对应type=ZEND_ACC_FINAL_CLASS□

除了上面的三种类型外,类还包含有另两种类型

- 另一种抽象类,它对应的type=ZEND_ACC_IMPLICIT_ABSTRACT_CLASS. 它在语法分析时并没有分析出来,因为这种类是由于其拥有抽象方法所产生的. 在PHP源码中,这个类别是在函数注册时判断是抽象方法或继承类时判断是抽象方法时设置的.
- 接口,其type=ZEND_ACC_INTERFACE.这个在接口关键字解析时设置,见interface_entry:对应的语法说明.

这五种类型在Zend/zend_complie.h文件中定义如下:

```
#define ZEND_ACC_IMPLICIT_ABSTRACT_CLASS    0x10
#define ZEND_ACC_EXPLICIT_ABSTRACT_CLASS    0x20
#define ZEND_ACC_FINAL_CLASS                0x40
#define ZEND_ACC_INTERFACE                  0x80
```

常规类为0,在这里没有定义,并且在程序也是直接赋值为0.

语法解析完后就可以知道一个类是抽象类还是final类,普通的类,又或者接口. 定义类时调用了□zend_do_begin_class_declaration和zend_do_end_class_declaration函数, 从这两个函数传入的参数,zend_do_begin_class_declaration函数用来处理类名,类的类别和父类, zend_do_end_class_declaration函数用来处理接口和类的中间代码 这两个函数在Zend/zend_complie.c文件中可以找到其实现.

在zend_do_begin_class_declaration中,首先会对传入的类名作一个转化,统一成小写,这也是为什么类名不区分大小的原因,如下代码

```
<?php
class TIPI {
}

class tipi {

}
```

运行时程序报错: Fatal error: Cannot redeclare class tipi. 这个报错是在运行生成中间的代码时显示的. 这个判断的过程在后面中间代码生成时说明.而关于类的名称的判断则是通过 **T_STRING** token, 在语法解析时做的判断, 但是这只能识别出类名是一个字符串. 假如类名为一些关键字, 如下代码:

```
class self {
}
```

运行, 程序会显示： Fatal error: Cannot use 'self' as class name as it is reserved in...

以上的这个程序判断也是在 **zend_do_begin_class_declaration** 函数中进行. 与self一样, 还有parent, static两个关键字.当这个函数执行完后，我们会得到类声明生成的中间代码为：**ZEND_DECLARE_CLASS** 。 当然，如果我们是声明内部类的话，则生成的中间代码为：**ZEND_DECLARE_INHERITED_CLASS**.

根据生成的中间代码，我们在Zend/zend_vm_execute.h文件中找到其对应的执行函数 **ZEND_DECLARE_CLASS_SPEC_HANDLER**。 这个函数通过调用 **do_bind_class** 函数将此类加入到EG(class_table) 。 在添加到列表的同时，也判断该类是否存在，如果存在，则添加失败，报我们之前出现过类重复声明错误，只是这个判断在编译期是不会生效的。

类的结构是以 **struct _zend_class_entry** 结构体为核心，类的实现是以class为中心作词法分析、语法分析等， 在这些过程中识别出类的类别，类的类名等，并将识别出来的结果存放到类的结构中。 下一节我们一起看看类所包含的成员变量和成员方法。

类的成员变量及方法

在上一小节，我们介绍了类的结构和声明过程，从而，我们知道了类的存储位置，类的类型设置等的实现方式。 在本小节，我们将介绍类的成员变量和方法。 首先，我们看一下，什么是成员变量，什么是成员方法。

类的成员变量在PHP中本质上是一个变量，只是在类中这些变量都归属于这个类，并且给这些变量都加上访问控制。 类的成员变量也称为成员属性，它是现实世界实体属性的抽象，可以用来描述对象状态的数据。 类的成员方法在PHP中本质上是一个函数，只是这个函数以类的方法存在，可能它是一个类方法也可能是一个实例方法， 并且在这些方法上都加上了类的访问控制。类的成员方法是现实世界实体行为的抽象，可以用来实现类的行为。

成员变量

在第三章介绍过变量，不过那些变量要么是定义在全局范围中，叫做全局变量，要么是定义在某个函数中，叫做局部变量。 成员变量是定义在类里面，并和方法处于同一层次。如下一个简单的PHP代码示例，定义了一个类，并且这个类有一个成员变量。

```
class Tipi {
    public $var;
}
```

这样一个类在PHP内核中是如何存储的呢？这个问题在上一小节已经有了说明。现在，我们要讨论的是这个成员变量是如何存储的。 假如我们需要直接访问这个变量，整个访问过程是如何的呢？

因为成员变量也是一样变量，

get_class_vars()

静态成员变量

因为类的静态成员变量是所有实例共用的，它也叫做类变量。

成员方法

get_class_methods()

静态成员方法

因为类的静态成员方法通常也叫做类方法。

附录A: PHP及Zend API