TIPI

TIPI:深入理解PHP内核

www.php-internal.com RELEASE_2012-04-04_V0.7.3

获取新版本

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

第一章 准备工作和背景知识

千里之行，始于足下。

在开始进入PHP的内核实现之前，需要做一些准备工作，也需要了解一些背景知识。本章主要涉及PHP源码的获取，PHP源码的编译，从而得到我们的调试环境。

接下来，我们将简单描述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。

> 目前PHP6已经停止开发了，PHP6的设计初衷是向后不兼容以及Unicode支持等。目前很多特性已经在PHP5.3和PHP5.4中实现了:比如5.4中的traits，支持C#类似的getter&setter语法（目前处在实现阶段），基本类型的类型提示等。

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，首先需要下载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svn/git等版本控制软件，也可以使用svn/git来获取最新的源代码。

```
# git 官方地址
git clone https://git.php.net/repository/php-src.git
# 也可以访问github官方镜像
git clone git://github.com/php/php-src.git
cd php-src && git checkout origin PHP-5.3 # 签出5.3分支

# svn地址不变，不过不推荐从这里签出代码
cd ~
svn co http://svn.php.net/repository/php/php-src/branches/PHP_5_2 php-src-5.2
#5.2版本
svn co http://svn.php.net/repository/php/php-src/branches/PHP_5_3 php-src-5.3
#5.3版本
```

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，如果你想修复PHP的某个Bug或者提交新功能的话，有版本控制也会容易的多，更多信息可以参考附录：怎样为PHP做贡献。

> 目前PHP已经迁移到Git了，PHP的wiki上有关于 迁移到Git的说明，以及使用Git的流程 ~~在笔者编写这些内容的时候PHP版本控制是还基于SVN的，上面提到的github镜像地址目前已经没有同步更新了，由于把svn同步到git会对系统性能造成明显影响，加上社区还没有就到底是否迁移到git达成一致，所以也就停止了更新。目前很多开源软件都开始转向了分布式版本控制系统(DVCS)，例如Python语言在转向DVCS时对目前的分布式版本控制系统做了一个详细的对比，如果以前没有接触过，笔者强烈建议试试这些版本控制软件。~~现在Github的同步基本是实时的。所以习惯Github基本上可以把Github当做官方版本库了。

2.准备编译环境

在*nix环境下，需要安装编译构建环境。如果你用的是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。相关资料请自行查阅相关文档。如果你是从svn/git签出的代码则需要执行代码根目录的buildconf脚本以生成所需要的构建脚本。

```
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：

> 有的系统自带的autoconf程序版本会有Bug，可能导致扩展的配置无法更新，如果在执行./buildconf时 报错，可以更具出错信息安装合适版本的autoconf工具。

```
./configure --help # 查看可用参数
```

为了尽快得到可以测试的环境，我们仅编译一个最精简的PHP。通过执行 ./configure --disable-all来进行配置。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。

```
./configure --disable-all
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， 在sapi/cli目录里就已经有了php的可以执行文件. 执行一下命令：

```
./sapi/cli/php -v
```

-v参数表示输出版本号， 如果命令执行完后看到输出php版本信息则说明编译成功。 如果是make install的话可以执行$prefix/bin/php这个路径的php。 当然如果是安装在系统目录或者你的prefix目录在$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， 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。

推荐书籍和参考

- linuxsir.org的make介绍 http://www.linuxsir.org/main/doc/gnumake/GNUmake_v3.80-zh_CN_html/index.html
- 《Autotools A Practioner's Guide》

第二节 源码结构、阅读代码方法

PHP源码目录结构

俗话讲：重剑无锋， 大巧不工。PHP的源码在结构上非常清晰。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的测试脚本集合，包含PHP各项功能的测试文件
- **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，例如以下的这个测试脚本/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。phpt文件的编写详细信息可参考 附录E phpt文件的编写。如果你真的那么感兴趣，那么研究下$PHP_SRC/run-tests.php脚本的实现也是不错的选择。这个测试框架刚开始 由PHP的发明者Rasmus Lerdorf编写，后来进行了很多的改进。后面可能会引入并行测试的支持。

PHP源码阅读工具

使用VIM + Ctags

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，在文字编辑方面基本上无所不能。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。

推荐在Linux下编写代码的读者或多或少的试一试ctags。ctags支持非常多的语言，可以将源代码中的各种符号（如:函数、宏类等信息）抽取出来做上标记并保存到一个文件中，供其他文本编辑工具（VIM，EMACS等）进行检索。它保存的文件格式符合UNIX的哲学（小即是美），使用也比较简洁：

```
#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 /server/ ，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
#或者在vim中运行命令:
: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，这个文件的格式如下：

```
{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，第一列是符号名（如上例的EG宏），第二列是该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（可以是变量名之类）使用**CTRL+]**时VIM将尝试从tags文件中检索这个符号。如果找到则根据该符号所在的文件以及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。

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。

习惯这种浏览代码的方式之后，大家会感觉很方便的。不过若你不习惯使用VIM这类编辑器，也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。如果执行还是会出错，请执行下`ctags -v` 或着 `which ctags`确保你执行的是新安装的ctags。

使用IDE查看代码

如果不习惯使用VIM来看代码，也可以使用一些功能较丰富的IDE，比如Windows下可以使用Visual Studio 2010 Express。或者使用跨平台的Netbeans、Eclipse来查看代码，当然，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
```

对于一些搜索类型的操作，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的一些很常见的宏，或者有些对于才开始接触源码的读者比较难懂的代码。这些代码在PHP的源码中出现的频率极高，基本在每个模块都会他们的身影。本小节我们提取中间的一些进行说明。

1. "##"和"#"

宏是C/C++是非常强大，使用也很多的一个功能，有时用来实现类似函数内联的效果，或者将复杂的代码进行简单封装，提高可读性或可移植性等。在PHP的宏定义中经常使用双井号。下面对"##"及"#"进行详细介绍。

双井号(##)

在C语言的宏中，"##"被称为 **连接符**（concatenator），它是一种预处理运算符，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的值连接起来。以这种连接的方式以基础，多次使用这种宏形式，可以将它当作一个代码生成器，这样可以在一定程度上减少代码密度，我们也可以将它理解为一种代码重用的手段，间接地减少不小心所造成的错误。

单井号(#)

"#"是一种预处理运算符，它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循环

PHP源码中大量使用了宏操作， 比如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)
```

这段代码， 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。 在其他使用C/C++编写的程序中也会有很多这种编写宏的代码， 多行宏的这种格式已经是一种公认的编写方式了。 为什么在宏定义时需要使用do-while语句呢? 我们知道do-while循环语句是先执行循环体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)时由于条件肯定为false, 代码也肯定只执行一次， 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。

了解了do-while循环在宏中的作用， 再来看"空操作"的定义。由于PHP需要考虑到平台的移植性和不同的系统配置， 所以需要在某些时候把一些宏的操作定义为空操作。例如在sapi\thttpd\thttpd.c文件中的

VEC_FREE():

```
#ifdef SERIALIZE_HEADERS
    # define VEC_FREE() smart_str_free(&vec_str)
#else
    # define VEC_FREE() do {} while (0)
#endif
```

这里涉及到条件编译，在定义了SERIALIZE_HEADERS宏的时候将VEC_FREE()定义为如上的内容，而没有定义时，不需要做任何操作，所以后面的宏将VEC_FREE()定义为一个空操作，不做任何操作，通常这样来保证一致性，或者充分利用系统提供的功能。

有时也会使用如下的方式来定义“空操作”，这里的空操作和上面的还是不一样，例如很常见的Debug日志打印宏：

```
#ifdef DEBUG
#   define LOG_MSG printf
#else
#   define LOG_MSG(...)
#endif
```

在编译时如果定义了DEBUG则将LOG_MSG当做printf使用，而不需要调试，正式发布时则将LOG_MSG()宏定义为空，由于宏是在预编译阶段进行处理的，所以上面的宏相当于从代码中删除了。

上面提到了两种将宏定义为空的定义方式，看上去一样，实际上只要明白了宏都只是简单的代码替换就知道该如何选择了。

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（__LINE__）和文件名（__FILE__）。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中定义的宏，这系列宏主要的作用是解决线程安全所写的全局变量包裹宏，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对这个全局变量进行操作的。

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

不积跬步，无以至千里。

完成这章后，我们就可以开始我们的千里之行了，我们完成的第一步：起步。

在本章，我们开始搭建了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源码本身在功能上的实现。

好了，下一步我们从宏观上来看看PHP的实现：概览。

第二章 用户代码的执行

不识庐山真面目，只缘身在此山中。

PHP做为一种优秀的脚本语言，在当前的互联网应用中可谓风光无限。从简单的“Hello World!”到各种框架开发，架构设计，性能优化，到编写PHP扩展，PHP编程中涉及的知识结构和跨度蔚为可观。从这个角度上来看，学会PHP编程的语法可能并不困难，但如果想真正用好PHP，在不同的场景下发挥PHP最大的性能和效用，对PHP的理解到达熟悉和精通的程度，就不得不去了解PHP语言的实现，进一步理解PHP语法的本质，这确实是一件需要更多的精力和时间的事情。

PHP语言经过许多人多年的淬炼，性能不断优化，支持的语法现象与各种特性也越来越多。导致PHP内核的代码中，涉及知识面比较广泛，具体实现也非常复杂，从脚本的编译解析到执行以及和Web服务器等的配合，内存管理，语法实现等等。为了不过早陷入细节的沼泽，我们先从整体上来接触PHP的实现，先对PHP的整体结构，生命周期，PHP与其它容器（如Apache）的交互，PHP的整个执行过程等进行一个大概的了解，从而有一个整体的概念。

> 关于PHP是如何一步步从一个朴素的想法发展到今天的模样，可以了解一下PHP的历史：
> http://en.wikipedia.org/wiki/PHP

从宏观上来看，PHP内核的实现与世界上绝大多数的程序一样，接收输入数据，做相应处理然后输出（返回）结果。我们编写的代码就是PHP接收的输入数据，PHP内核对我们编写的代码进行解释和运算，最后返回相应的运算结果。然而，PHP与我们自己平时写的一般的C程序有所不同的是，我们的程序一般用来解决某个具体问题，而PHP本身实现了把用户的逻辑“翻译”为机器语言来执行的功能，这也是各种编译语言与承载具体业务逻辑的程序代码的一个明显区别。于是PHP就多出一个把用户代码“翻译”成具体操作的步骤：**词法分析、语法分析**

当用户代码输入给PHP内核去执行的时候，PHP内核会对PHP代码进行词法分析和语法分析，词法分析是把PHP代码分割成一个个的“单元”（TOKEN），语法分析则将这些“单元”转化为Zend Engine可执行的操作。然后PHP内部的Zend Engine对这些操作进行顺次的执行。Zend Engine是PHP内核的核心部分，负责最终操作的执行和结果的返回，可以理解成为PHP内核中的“发动机”。

于是PHP代码的执行过程可以简单描述为下图：

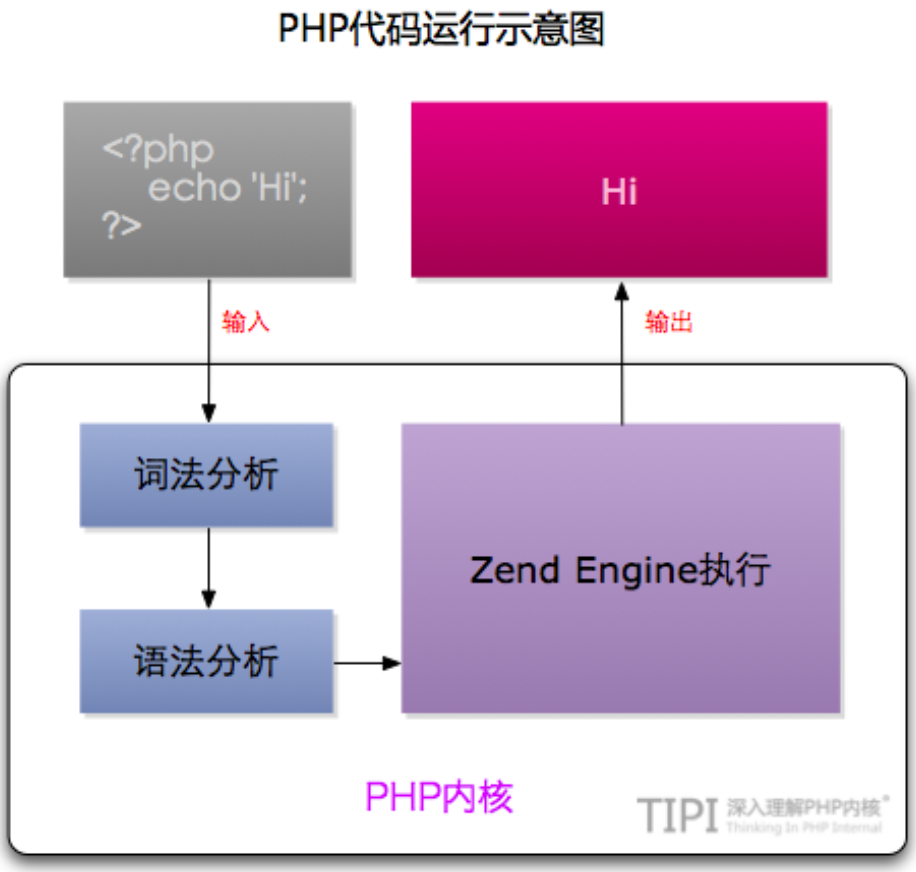


图2.1 单进程SAPI生命周期

接下来，本章会对图中的每一部分展开详细的讨论，主要包括以下内容：

1. PHP内部的生命周期
2. SAPI接口
3. 词法分析与语法分析
4. 什么是Opcodes

第一节 生命周期和Zend引擎

一切的开始: SAPI接口

SAPI(Server Application Programming Interface)指的是PHP具体应用的编程接口，就像PC一样，无论安装哪些操作系统，只要满足了PC的接口规范都可以在PC上正常运行，PHP脚本要执行有很多种方式，通过Web服务器，或者直接在命令行下，也可以嵌入在其他程序中。

通常，我们使用Apache或者Nginx这类Web服务器来测试PHP脚本，或者在命令行下通过PHP解释器程序来执行。脚本执行完后，Web服务器应答，浏览器显示应答信息，或者在命令行标准输出上显示内容。

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，实际上它们的工作流程是一样的。命令行参数传递给PHP解释器要执行的脚本，相当于通过url请求一个PHP页面。脚本执行完成后返回响应结果，只不过命令行的响应结果是显示在终端上。

脚本执行的开始都是以SAPI接口实现开始的。只是不同的SAPI接口实现会完成他们特定的工作，例如

Apache的mod_php SAPI实现需要初始化从Apache获取的一些信息，在输出内容是将内容返回给Apache，其他的SAPI实现也类似。

下面几个小节将对一些常见的SAPI实现进行更为深入的介绍。

开始和结束

PHP开始执行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：第一个过程是模块初始化阶段（MINIT），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)，该过程只进行一次。第二个过程是模块激活阶段（RINIT），该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模块激活（RINIT请求开始）。例如PHP注册了一些扩展模块，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。模块在实现时可以通过如下宏来实现这些回调函数：

```
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会调用所有模块的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回调函数类似：

```
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，一个在请求结束后停用模块(RSHUTDOWN，对应RINIT)，一个在SAPI生命周期结束（Web服务器退出或者命令行脚本执行完毕退出）时关闭模块(MSHUTDOWN，对应MINIT)。

```
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接口实现就完成了其生命周期。如图2.1所示：

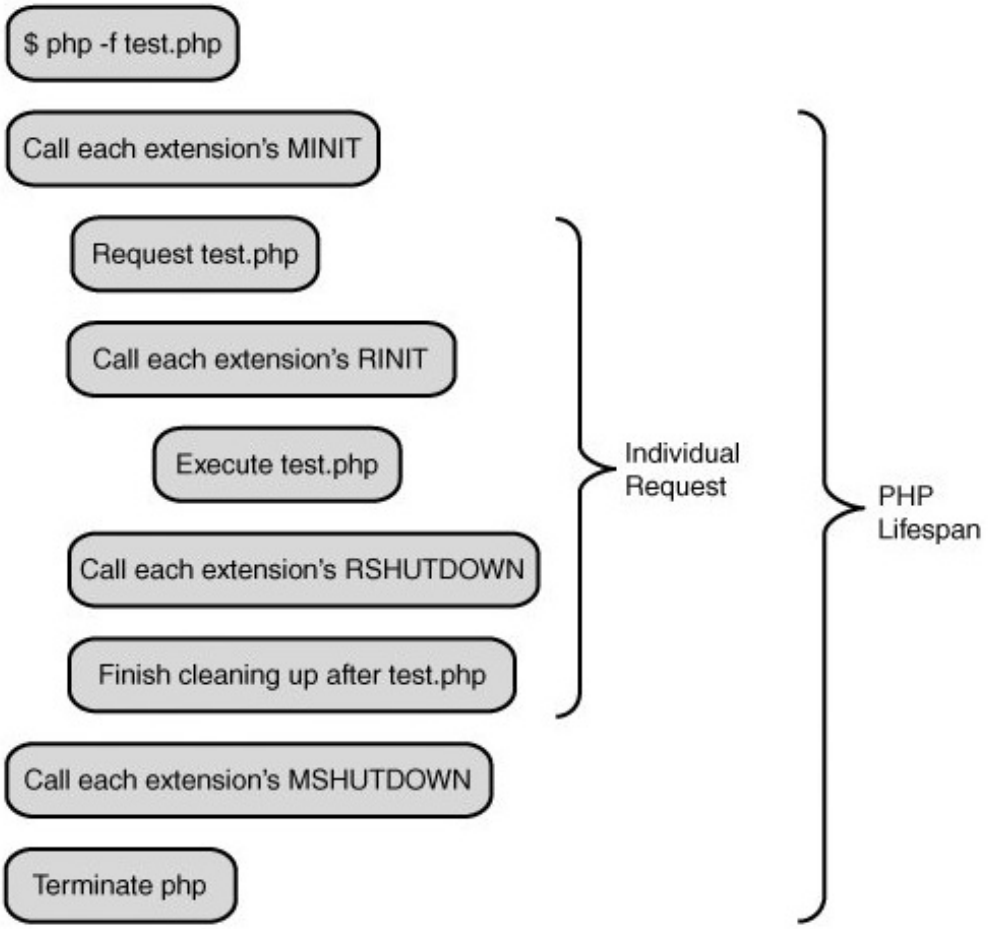


图2.1 单进程SAPI生命周期

如上的图是非常简单，也很好理解。只是在各个阶段之间PHP还做了许许多多的工作。这里做一些补充：

启动

在调用每个模块的模块初始化前，会有一个初始化的过程，它包括：

- **初始化若干全局变量**

这里的初始化全局变量大多数情况下是将其设置为NULL，有一些除外，比如设置zuf（zend_utility_functions），以zuf.printf_function = php_printf为例，这里的php_printf在zend_startup函数中会被赋值给zend_printf作为全局函数指针使用，而zend_printf函数通常会作为常规字符串输出使用，比如显示程序调用栈的debug_print_backtrace就是使用它打印相关信息。

- **初始化若干常量**

这里的常量是PHP自己的一些常量，这些常量要么是硬编码在程序中,比如PHP_VERSION，要么是写在配置头文件中，比如PEAR_EXTENSION_DIR，这些是写在config.w32.h文件中。

- **初始化Zend引擎和核心组件**

前面提到的zend_startup函数的作用就是初始化Zend引擎，这里的初始化操作包括内存管理初始化、全局使用的函数指针初始化（如前面所说的zend_printf等），对PHP源文件进行词法分析、语法分析、中间代码执行的函数指针的赋值，初始化若干HashTable（比如函数表，常量表等等），为ini文件解析做准

备， 为PHP源文件解析做准备， 注册内置函数（如strlen、define等）， 注册标准常量（如E_ALL、TRUE、NULL等）、注册GLOBALS全局变量等。

- **解析php.ini**

php_init_config函数的作用是读取php.ini文件， 设置配置参数， 加载zend扩展并注册PHP扩展函数。此函数分为如下几步： 初始化参数配置表， 调用当前模式下的ini初始化配置， 比如CLI模式下， 会做如下初始化：

```
INI_DEFAULT("report_zend_debug", "0");
INI_DEFAULT("display_errors", "1");
```

不过在其它模式下却没有这样的初始化操作。接下来会的各种操作都是查找ini文件：

1. 判断是否有php_ini_path_override， 在CLI模式下可以通过-c参数指定此路径（在php的命令参数中-c表示在指定的路径中查找ini文件）。
2. 如果没有php_ini_path_override， 判断php_ini_ignore是否为非空（忽略php.ini配置， 这里也就CLI模式下有用， 使用-n参数）。
3. 如果不忽略ini配置， 则开始处理php_ini_search_path（查找ini文件的路径）， 这些路径包括CWD(当前路径， 不过这种不适用CLI模式)、 执行脚本所在目录、环境变量PATH和PHPRC和配置文件中的PHP_CONFIG_FILE_PATH的值。
4. 在准备完查找路径后， PHP会判断现在的ini路径（php_ini_file_name）是否为文件和是否可打开。如果这里ini路径是文件并且可打开， 则会使用此文件， 也就是CLI模式下通过-c参数指定的ini文件的优先级是最高的， 其次是PHPRC指定的文件， 第三是在搜索路径中查找php-%sapi-module-name%.ini文件（如CLI模式下应该是查找php-cli.ini文件）， 最后才是搜索路径中查找php.ini文件。

- **全局操作函数的初始化**

php_startup_auto_globals函数会初始化在用户空间所使用频率很高的一些全局变量， 如：$_GET、$_POST、$_FILES等。 这里只是初始化， 所调用的zend_register_auto_global函数也只是将这些变量名添加到CG(auto_globals)这个变量表。

php_startup_sapi_content_types函数用来初始化SAPI对于不同类型内容的处理函数， 这里的处理函数包括POST数据默认处理函数、默认数据处理函数等。

- **初始化静态构建的模块和共享模块(MINIT)**

php_register_internal_extensions_func函数用来注册静态构建的模块， 也就是默认加载的模块， 我们可以将其认为为内置模块。在PHP5.3.0版本中内置的模块包括PHP标准扩展模块（/ext/standard/目录，这里是我们用的最频繁的函数， 比如字符串函数， 数学函数， 数组操作函数等等）， 日历扩展模块、FTP扩展模块、 session扩展模块等。这些内置模块并不是一成不变的， 在不同的PHP模板中， 由于不同时间的需求或其它影响因素会导致这些默认加载的模块会变化， 比如从代码中我们就可以看到mysql、xml等扩展模块曾经或将来会作为内置模块出现。

模块初始化会执行两个操作： 1. 将这些模块注册到已注册模块列表（module_registry）， 如果注册的模块已经注册过了， PHP会报Module XXX already loaded的错误。 1. 将每个模块中包含的函数注册到函数表（ CG(function_table) ）， 如果函数无法添加， 则会报 Unable to register functions, unable to load。

在注册了静态构建的模块后，PHP会注册附加的模块，不同的模式下可以加载不同的模块集，比如在CLI模式下是没有这些附加的模块的。

在内置模块和附加模块后，接下来是注册通过共享对象（比如DLL）和php.ini文件灵活配置的扩展。

在所有的模块都注册后，PHP会马上执行模块初始化操作（zend_startup_modules）。 它的整个过程就是依次遍历每个模块，调用每个模块的模块初始化函数， 也就是在本小节前面所说的用宏PHP_MINIT_FUNCTION包含的内容。

- **禁用函数和类**

php_disable_functions函数用来禁用PHP的一些函数。这些被禁用的函数来自PHP的配置文件的disable_functions变量。 其禁用的过程是调用zend_disable_function函数将指定的函数名从CG(function_table)函数表中删除。

php_disable_classes函数用来禁用PHP的一些类。这些被禁用的类来自PHP的配置文件的disable_classes变量。 其禁用的过程是调用zend_disable_class函数将指定的类名从CG(class_table)类表中删除。

ACTIVATION

在处理了文件相关的内容，PHP会调用php_request_startup做请求初始化操作。 请求初始化操作，除了图中显示的调用每个模块的请求初始化函数外，还做了较多的其它工作，其主要内容如下：

- **激活Zend引擎**

gc_reset函数用来重置垃圾收集机制，当然这是在PHP5.3之后才有的。

init_compiler函数用来初始化编译器，比如将编译过程中在放opcode的数组清空，准备编译时用来的数据结构等等。

init_executor函数用来初始化中间代码执行过程。 在编译过程中，函数列表、类列表等都存放在编译时的全局变量中， 在准备执行过程时，会将这些列表赋值给执行的全局变量中，如：EG(function_table) = CG(function_table); 中间代码执行是在PHP的执行虚拟栈中，初始化时这些栈等都会一起被初始化。 除了栈，还有存放变量的符号表(EG(symbol_table))会被初始化为50个元素的hashtable，存放对象的EG(objects_store)被初始化了1024个元素。 PHP的执行环境除了上面的一些变量外，还有错误处理，异常处理等等，这些都是在这里被初始化的。 通过php.ini配置的zend_extensions也是在这里被遍历调用activate函数。

- **激活SAPI**

sapi_activate函数用来初始化SG(sapi_headers)和SG(request_info)，并且针对HTTP请求的方法设置一些内容， 比如当请求方法为HEAD时，设置SG(request_info).headers_only=1； 此函数最重要的一个操作是处理请求的数据，其最终都会调用sapi_module.default_post_reader。 而sapi_module.default_post_reader在前面的模块初始化是通过php_startup_sapi_content_types函数注册了 默认处理函数为main/php_content_types.c文件中php_default_post_reader函数。 此函数会将POST的原始数据写入$HTTP_RAW_POST_DATA变量。

在处理了post数据后，PHP会通过sapi_module.read_cookies读取cookie的值， 在CLI模式下，此函数的实现为sapi_cli_read_cookies，而在函数体中却只有一个return NULL;

如果当前模式下有设置activate函数，则运行此函数，激活SAPI，在CLI模式下此函数指针被设置为NULL。

- **环境初始化**

这里的环境初始化是指在用户空间中需要用到的一些环境变量初始化，这里的环境包括服务器环境、请求数据环境等。实际到我们用到的变量，就是$_POST、$_GET、$_COOKIE、$_SERVER、$_ENV、$_FILES。和sapi_module.default_post_reader一样，sapi_module.treat_data的值也是在模块初始化时，通过php_startup_sapi_content_types函数注册了默认数据处理函数为main/php_variables.c文件中php_default_treat_data函数。

以$_COOKIE为例，php_default_treat_data函数会对依据分隔符，将所有的cookie拆分并赋值给对应的变量。

- **模块请求初始化**

PHP通过zend_activate_modules函数实现模块的请求初始化，也就是我们在图中看到Call each extension's RINIT。此函数通过遍历注册在module_registry变量中的所有模块，调用其RINIT方法实现模块的请求初始化操作。

运行

php_execute_script函数包含了运行PHP脚本的全部过程。

当一个PHP文件需要解析执行时，它可能会需要执行三个文件，其中包括一个前置执行文件、当前需要执行的主文件和一个后置执行文件。非当前的两个文件可以在php.ini文件通过auto_prepend_file参数和auto_append_file参数设置。如果将这两个参数设置为空，则禁用对应的执行文件。

对于需要解析执行的文件，通过zend_compile_file（compile_file函数）做词法分析、语法分析和中间代码生成操作，返回此文件的所有中间代码。如果解析的文件有生成有效的中间代码，则调用zend_execute（execute函数）执行中间代码。如果在执行过程中出现异常并且用户有定义对这些异常的处理，则调用这些异常处理函数。在所有的操作都处理完后，PHP通过EG(return_value_ptr_ptr)返回结果。

DEACTIVATION

PHP关闭请求的过程是一个若干个关闭操作的集合，这个集合存在于php_request_shutdown函数中。这个集合包括如下内容：

1. 调用所有通过register_shutdown_function()注册的函数。这些在关闭时调用的函数是在用户空间添加进来的。一个简单的例子，我们可以在脚本出错时调用一个统一的函数，给用户一个友好一些的页面，这个有点类似于网页中的404页面。
2. 执行所有可用的__destruct函数。这里的析构函数包括在对象池（EG(objects_store）中的所有对象的析构函数以及EG(symbol_table)中各个元素的析构方法。
3. 将所有的输出刷出去。
4. 发送HTTP应答头。这也是一个输出字符串的过程，只是这个字符串可能符合某些规范。
5. 遍历每个模块的关闭请求方法，执行模块的请求关闭操作，这就是我们在图中看到的Call each extension's RSHUTDOWN。
6. 销毁全局变量表（PG(http_globals)）的变量。

7. 通过zend_deactivate函数，关闭词法分析器、语法分析器和中间代码执行器。
8. 调用每个扩展的post-RSHUTDOWN函数。只是基本每个扩展的post_deactivate_func函数指针都是NULL。
9. 关闭SAPI，通过sapi_deactivate销毁SG(sapi_headers)、SG(request_info)等的内容。
10. 关闭流的包装器、关闭流的过滤器。
11. 关闭内存管理。
12. 重新设置最大执行时间

结束

最终到了要收尾的地方了。

- **flush**

sapi_flush将最后的内容刷新出去。其调用的是sapi_module.flush，在CLI模式下等价于fflush函数。

- **关闭Zend引擎**

zend_shutdown将关闭Zend引擎。

此时对应图中的流程，我们应该是执行每个模块的关闭模块操作。在这里只有一个zend_hash_graceful_reverse_destroy函数将module_registry销毁了。当然，它最终也是调用了关闭模块的方法的，其根源在于在初始化module_registry时就设置了这个hash表析构时调用ZEND_MODULE_DTOR宏。而ZEND_MODULE_DTOR宏对应的是module_destructor函数。在此函数中会调用模块的module_shutdown_func方法，即PHP_RSHUTDOWN_FUNCTION宏产生的那个函数。

在关闭所有的模块后，PHP继续销毁全局函数表，销毁全局类表、销售全局变量表等。通过zend_shutdown_extensions遍历zend_extensions所有元素，调用每个扩展的shutdown函数。

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：

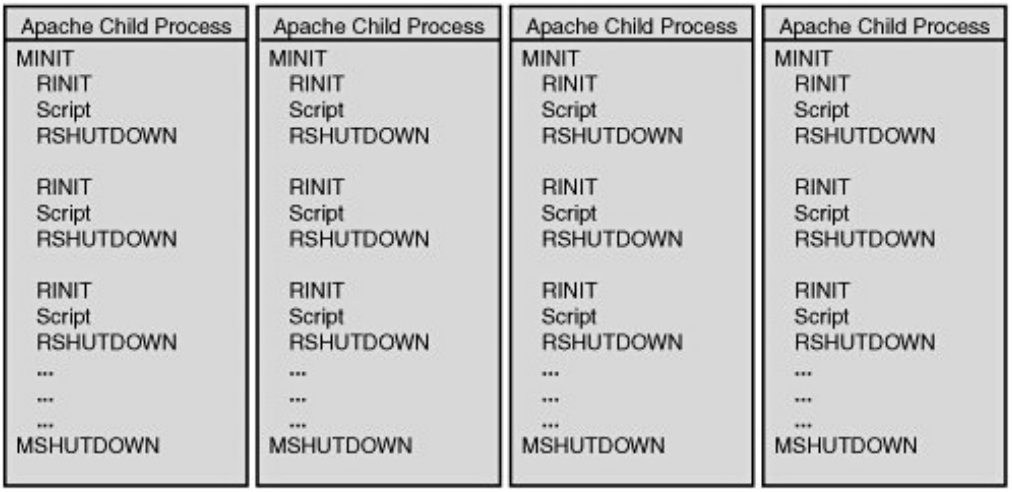


图2.2 多进程SAPI生命周期

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

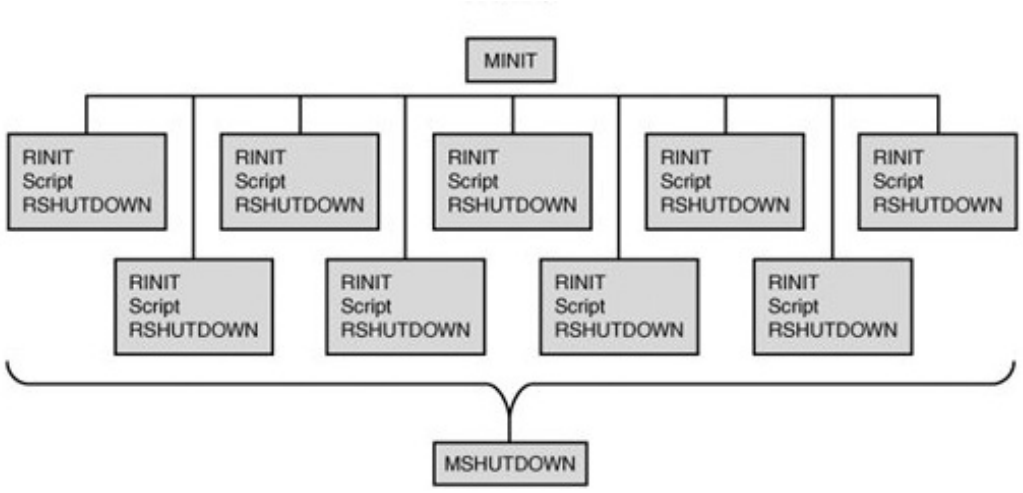


图2.3 多线程SAPI生命周期

Zend引擎

Zend引擎是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，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)。

目前PHP的实现和Zend引擎之间的关系非常紧密，甚至有些过于紧密了，例如很多PHP扩展都是使用的Zend API， 而Zend正是PHP语言本身的实现，PHP只是使用Zend这个内核来构建PHP语言的，而PHP扩展大都使用Zend API， 这就导致PHP的很多扩展和Zend引擎耦合在一起了，在笔者编写这本书的时候PHP核心开发者就提出将这种耦合解开，

目前PHP的受欢迎程度是毋庸置疑的，但凡流行的语言通常都会出现这个语言的其他实现版本， 这在

Java社区里就非常明显，目前已经有非常多基于JVM的语言了，例如IBM的Project Zero就实现了一个基于JVM的PHP实现，.NET也有类似的实现，通常他们这样做的原因无非是因为：他们喜欢这个语言，但又不想放弃原有的平台，或者对现有的语言实现不满意，处于性能或者语言特性等（HipHop就是这样诞生的）。

很多脚本语言中都会有语言扩展机制，PHP中的扩展通常是通过Pear库或者原生扩展，在Ruby中则这两者的界限不是很明显，他们甚至会提供两套实现，一个主要用于在无法编译的环境下使用，而在合适的环境则使用C实现的原生扩展，这样在效率和可移植性上都可以保证。目前这些为PHP编写的扩展通常都无法在其他的PHP实现中实现重用，HipHop的做法是对最为流行的扩展进行重写。如果PHP扩展能和ZendAPI解耦，则在其他语言中重用这些扩展也将更加容易了。

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，在其生命周期的各个阶段，一些与服务相关的操作都是通过SAPI接口实现。这些内置实现的物理位置在PHP源码的SAPI目录。这个目录存放了PHP对各个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SAPI实现都是一个_sapi_module_struct结构体变量。（SAPI接口）。在PHP的源码中，当需要调用服务器相关信息时，全部通过SAPI接口中对应方法调用实现，而这对应的方法在各个服务器抽象层实现时都会有各自的实现。

由于很多操作的通用性，有很大一部分的接口方法使用的是默认方法。

如图2.4所示，为SAPI的简单示意图。

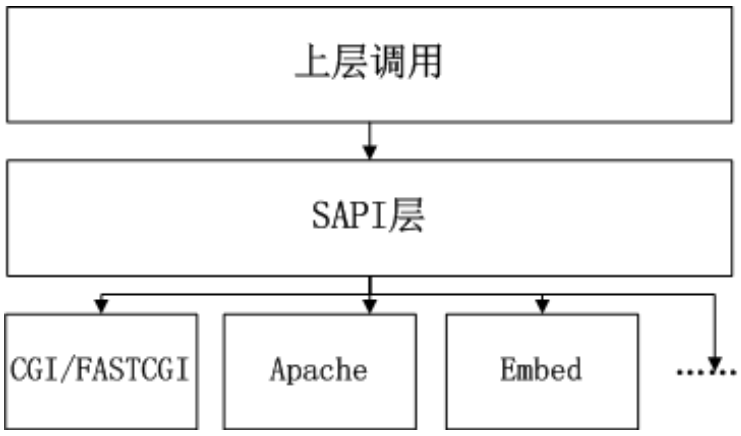


图2.4 SAPI的简单示意图

以cgi模式和apache2服务器为例，它们的启动方法如下：

```
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
```

这里的cgi_sapi_module是sapi_module_struct结构体的静态变量。 它的startup方法指向php_cgi_startup函数指针。在这个结构体中除了startup函数指针， 还有许多其它方法或字段。 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

    ...
    ...
};
```

其中一些函数指针的说明如下：

- startup 当SAPI初始化时， 首先会调用该函数。如果服务器处理多个请求时， 该函数只会调用一次。比如Apache的SAPI， 它是以mod_php5的Apache模块的形式加载到Apache中的， 在这个SAPI中， startup函数只在父进程中创建一次， 在其fork的子进程中不会调用。
- activate 此函数会在每个请求开始时调用， 它会再次初始化每个请求前的数据结构。
- deactivate 此函数会在每个请求结束时调用， 它用来确保所有的数据都， 以及释放在activate中初始化的数据结构。
- shutdown 关闭函数， 它用来释放所有的SAPI的数据结构、内存等。

- ub_write 不缓存的写操作(unbuffered write)，它是用来将PHP的数据输出给客户端，如在CLI模式下，其最终是调用fwrite实现向标准输出输出内容；在Apache模块中，它最终是调用Apache提供的方法rwrite。
- sapi_error 报告错误用，大多数的SAPI都是使用的PHP的默认实现php_error。
- flush 刷新输出，在CLI模式下通过使用C语言的库函数fflush实现，在php_mode5模式下，使用Apache的提供的函数函数rflush实现。
- read_cookie 在SAPI激活时，程序会调用此函数，并且将此函数获取的值赋值给SG(request_info).cookie_data。在CLI模式下，此函数会返回NULL。
- read_post 此函数和read_cookie一样也是在SAPI激活时调用，它与请求的方法相关，当请求的方法是POST时，程序会操作$_POST、$HTTP_RAW_POST_DATA等变量。
- send_header 发送头部信息，此方法一般的SAPI都会定制，其所不同的是，有些的会调服务器自带的（如Apache），有些的需要你自己实现（如 FastCGI）。

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的源码中实现了很多的实现，比如IIS的实现以及一些非主流的Web服务器实现，其文件结构如图2.5所示：

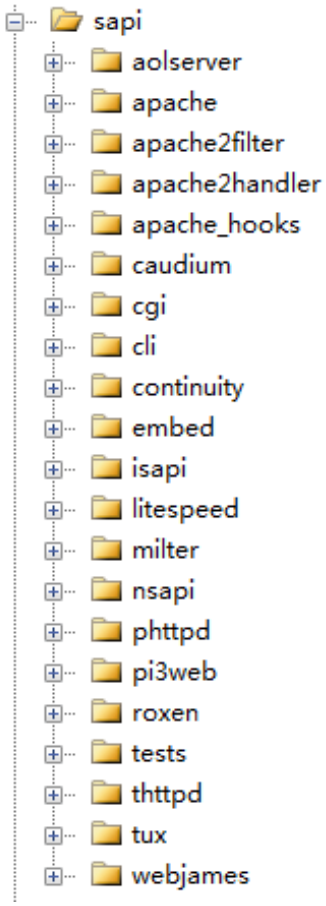


图2.5 SAPI文件结构图

> 目前PHP内置的很多SAPI实现都已不再维护或者变的有些非主流了，PHP社区目前正在考虑将一些SAPI移出代码库。社区对很多功能的考虑是除非真的非常必要，或者某些功能已近非常通用了，否则就在PECL库中，例如非常流行的APC缓存扩展将进入核心代码库中。

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。

这样的结构在PHP的源码中有多处使用，比如在PHP扩展开发中，每个扩展都需要定义一个zend_module_entry结构体。这个结构体的作用与sapi_module_struct结构体类似，都是一个类似模板方法模式的应用。在PHP的生命周期中如果需要调用某个扩展，其调用的方法都是zend_module_entry结构体中指定的方法，如在上一小节中提到的在执行各个扩展的请求初始化时，都是统一调用request_startup_func方法，而在每个扩展的定义时，都通过宏PHP_RINIT指定request_startup_func对应的函数。以VLD扩展为例：其请求初始化为PHP_RINIT(vld),与之对应在扩展中需要有这个函数的实现：

```
PHP_RINIT_FUNCTION(vld) {
}
```

所以，我们在写扩展时也需要实现扩展的这些接口，同样，当实现各服务器接口时也需要实现其对应的SAPI。

Apache模块

Apache是Apache软件基金会的一个开放源代码的Web服务器，可以在大多数电脑操作系统中运行，由于其跨平台和安全性被广泛使用，是最流行的Web服务器端软件之一。Apache支持许多特性，大部分通过模块扩展实现。常见的模块包括mod_auth（权限验证）、mod_ssl（SSL和TLS支持）mod_rewrite（URL重写）等。一些通用的语言也支持以Apache模块的方式与Apache集成。如Perl，Python，Tcl，和PHP等。

当PHP需要在Apache服务器下运行时，一般来说，它可以mod_php5模块的形式集成，此时mod_php5模块的作用是接收Apache传递过来的PHP文件请求，并处理这些请求，然后将处理后的结果返回给Apache。如果我们在Apache启动前在其配置文件中配置好了PHP模块（mod_php5），PHP模块通过注册apache2的ap_hook_post_config挂钩，在Apache启动的时候启动此模块以接受PHP文件的请求。

除了这种启动时的加载方式，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。

Apache是如何加载模块的呢？我们以前面提到的mod_php5模块为例。首先我们需要在Apache的配置文件httpd.conf中添加一行：

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

这里我们使用了LoadModule命令，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。

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。Apache的每一个模块都是以module结构体的形式存在， 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。这也就决定了我们的文件名和模块名是相同的。 通过之前指令中指定的路径找到相关的动态链接库文件后， 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module结构体中的magic字段实现的。 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 */
```

在php5_module定义的结构中，php_dir_cmds是模块定义的所有的指令集合，其定义的内容如下：

```
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}
};
```

这是mod_php5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，mod_php5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。

到这里, 我们知道了Apache加载mod_php5模块的整个过程, 可是这个过程与我们的SAPI有什么关系呢？ mod_php5也定义了属于Apache的sapi_module_struct结构:

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

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

NULL,                       /* activate */
NULL,                       /* deactivate */

php_apache_sapi_ub_write,           /* unbuffered write */
php_apache_sapi_flush,              /* flush */
php_apache_sapi_get_stat,           /* get uid */
php_apache_sapi_getenv,             /* getenv */

php_error,                  /* error handler */

php_apache_sapi_header_handler,         /* header handler */
php_apache_sapi_send_headers,           /* send headers handler */
NULL,                       /* send header handler */

php_apache_sapi_read_post,          /* read POST data */
php_apache_sapi_read_cookies,           /* read Cookies */

php_apache_sapi_register_variables,
php_apache_sapi_log_message,            /* Log message */
php_apache_sapi_get_request_time,       /* Request Time */
NULL,                       /* Child Terminate */

STANDARD_SAPI_MODULE_PROPERTIES
};
```

这些方法都专属于Apache服务器。以读取cookie为例, 当我们在Apache服务器环境下, 在PHP中调用读取Cookie时, 最终获取的数据的位置是在激活SAPI时。它所调用的方法是read_cookies。

```
SG(request_info).cookie_data = sapi_module.read_cookies(TSRMLS_C);
```

对于每一个服务器在加载时, 我们都指定了sapi_module, 而Apache的sapi_module是apache2_sapi_module。 其中对应read_cookies方法的是php_apache_sapi_read_cookies函数。

又如flush函数, 在ext/standard/basic_functions.c文件中, 其实现为sapi_flush：

```
SAPI_API int sapi_flush(TSRMLS_D)
{
    if (sapi_module.flush) {
        sapi_module.flush(SG(server_context));
        return SUCCESS;
    } else {
        return FAILURE;
    }
}
```

如果我们定义了此前服务器接口的flush函数，则直接调用flush对应的函数，返回成功，否则返回失败。对于我们当前的Apache模块，其实现为php_apache_sapi_flush函数，最终会调用Apache的ap_rflush，刷新apache的输出缓冲区。当然，flush的操作有时也不会生效，因为当PHP执行flush函数时，其所有的行为完全依赖于Apache的行为，而自身却做不了什么，比如启用了Apache的压缩功能，当没有达到预定的输出大小时，即使使用了flush函数，Apache也不会向客户端输出对应的内容。

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的内部实现。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生命周期可知：嵌入式PHP类似CLI，也是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。因为特定的应用由自己特殊的需求，只是在处理PHP脚本这个环节基本一致。

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，例如很多游戏中使用Lua语言作为粘合语言，或者作为扩展游戏的脚本语言，类似的，浏览器中的Javascript语言就是嵌入在浏览器中的。只是目前很少有应用将PHP作为嵌入语言来使用，PHP的强项目前还是在Web开发方面。

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。

以上是使用PHP的嵌入式方式开发的一个简单的PHP代码运行器，它的这些调用的方式都基于PHP本身的一些实现，而针对嵌入式的SAPI定义是非常简单的，没有Apache和CGI模式的复杂，或者说是相当简陋，这也是由其所在环境决定。在嵌入式的环境下，很多的网络协议所需要的方法都不再需要。如下所示，为嵌入式的模块定义。

```
sapi_module_struct php_embed_module = {
    "embed",                       /* name */
    "PHP Embedded Library",        /* pretty name */

    php_embed_startup,              /* startup */
    php_module_shutdown_wrapper,   /* shutdown */

    NULL,                          /* activate */
    php_embed_deactivate,           /* deactivate */

    php_embed_ub_write,             /* unbuffered write */
    php_embed_flush,                /* flush */
    NULL,                          /* get uid */
    NULL,                          /* getenv */

    php_error,                     /* error handler */

    NULL,                          /* header handler */
    NULL,                          /* send headers handler */
    php_embed_send_header,          /* send header handler */

    NULL,                          /* read POST data */
    php_embed_read_cookies,         /* read Cookies */

    php_embed_register_variables,   /* register server variables */
    php_embed_log_message,          /* Log message */
    NULL,                          /* Get request time */
    NULL,                          /* Child terminate */

    STANDARD_SAPI_MODULE_PROPERTIES
};
/* }}} */
```

在这个定义中我们看到了若干的NULl定义，在前面一小节中说到SAPI时，我们是以cookie的读取为例，在这里也有读取cookie的实现——php_embed_read_cookies函数，但是这个函数的实现是一个空指

针NULL。

而这里的flush实现与Apache的不同:

```
static void php_embed_flush(void *server_context)
{
    if (fflush(stdout)==EOF) {
        php_handle_aborted_connection();
    }
}
```

flush是直接调用fflush(stdout)，以达到清空stdout的缓存的目的。如果输出失败（fflush成功返回0，失败返回EOF），则调用php_handle_aborted_connection，进入中断处理程序。

参与资料

《Extending and Embedding PHP》

FastCGI

FastCGI简介

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。

FastCGI是Web服务器和处理程序之间通信的一种协议，是CGI的一种改进方案，FastCGI像是一个常驻(long-live)型的CGI，它可以一直执行，在请求到达时不会花费时间去fork一个进程来处理(这是CGI最为人诟病的fork-and-execute模式)。正是因为他只是一个通信协议，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，将CGI解释器进程保持在内存中，以此获得较高的性能。CGI程序反复加载是CGI性能低下的主要原因，如果CGI程序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。

一般情况下，FastCGI的整个工作流程是这样的：

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实现了Fastcgi协议，是一个TCP或UDP协议的服务器接受来自Web服务器的请求，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。随后就进入了PHP的生命周期：模块初始化，sapi初始化，处理PHP请求，模块关闭，sapi关闭等就构成了整个CGI的生命周期。

以TCP为例，在TCP的服务端，一般会执行这样几个操作步骤：

1. 调用socket函数创建一个TCP用的流式套接字；
2. 调用bind函数将服务器的本地地址与前面创建的套接字绑定；
3. 调用listen函数将新创建的套接字作为监听，等待客户端发起的连接，当客户端有多个连接连接到这个套接字时，可能需要排队处理；
4. 服务器进程调用accept函数进入阻塞状态，直到有客户进程调用connect函数而建立起一个连接；
5. 当与客户端创建连接后，服务器调用read_stream函数读取客户的请求；
6. 处理完数据后，服务器调用write函数向客户端发送应答。

TCP上客户-服务器事务的时序如图2.6所示：

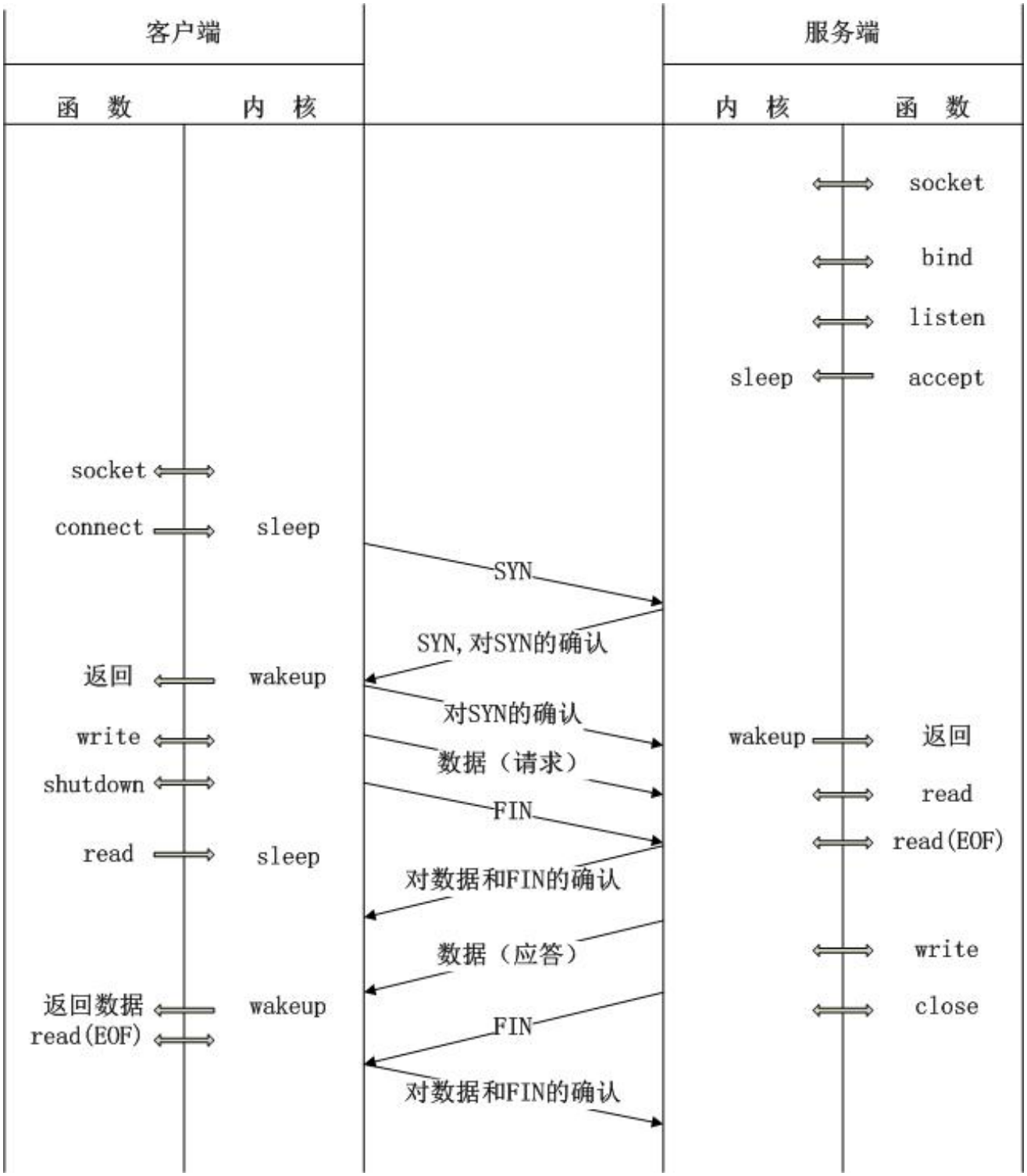


图2.6 TCP上客户-服务器事务的时序

PHP的CGI实现从cgi_main.c文件的main函数开始，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。对比TCP的流程，我们查看PHP对TCP协议的实现，虽然PHP本身也实现了这些流程，但是在main函数中一些过程被封装成一个函数实现。对应TCP的操作流程，PHP首先会执行创建socket，绑定套接字，创建监听：

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

在fastcgi.c文件中，fcgi_listen函数主要用于创建、绑定socket并开始监听，它走完了前面所列TCP流

程的前三个阶段,

```
    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) {
        ...
    }
```

当服务端初始化完成后，进程调用accept函数进入阻塞状态，在main函数中我们看到如下代码：

```
    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;
                    ...
            }
```

如上的代码是一个生成子进程，并等待用户请求。在fcgi_accept_request函数中，程序会调用accept函数阻塞新创建的进程。当用户的请求到达时，fcgi_accept_request函数会判断是否处理用户的请求，其中会过滤某些连接请求，忽略受限制客户的请求，如果程序受理用户的请求，它将分析请求的信息，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);

}
```

如上对应服务器端读取用户的请求数据。

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。

在处理完用户的请求后，服务器端将返回信息给客户端，此时在main函数中调用的是fcgi_finish_request(&request, 1); fcgi_finish_request函数定义在fastcgi.c文件中，其代码如下：

```
int fcgi_finish_request(fcgi_request *req, int force_close)
{
int ret = 1;

if (req->fd >= 0) {
    if (!req->closed) {
        ret = fcgi_flush(req, 1);
        req->closed = 1;
    }
    fcgi_close(req, force_close, 1);
}
return ret;
}
```

如上，当socket处于打开状态，并且请求未关闭，则会将执行后的结果刷到客户端，并将请求的关闭设置为真。将数据刷到客户端的程序调用的是fcgi_flush函数。在此函数中，关键是在于答应头的构造和写操作。程序的写操作是调用的safe_write函数，而safe_write函数中对于最终的写操作针对win和linux环境做了区分，在Win32下，如果是TCP连接则用send函数，如果是非TCP则和非win环境一样使用write函数。如下代码：

```
#ifdef _WIN32
if (!req->tcp) {
    ret = write(req->fd, ((char*)buf)+n, count-n);
} else {
    ret = send(req->fd, ((char*)buf)+n, count-n, 0);
    if (ret <= 0) {
            errno = WSAGetLastError();
    }
}
#else
ret = write(req->fd, ((char*)buf)+n, count-n);
#endif
```

在发送了请求的应答后，服务器端将会执行关闭操作，仅限于CGI本身的关闭，程序执行的是fcgi_close函数。fcgi_close函数在前面提的fcgi_finish_request函数中，在请求应答完后执行。同样，对于win平台和非win平台有不同的处理。其中对于非win平台调用的是write函数。

以上是一个TCP服务器端实现的简单说明。这只是我们PHP的CGI模式的基础，在这个基础上PHP增加了更多的功能。在前面的章节中我们提到了每个SAPI都有一个专属于它们自己的sapi_module_struct结构：cgi_sapi_module，其代码定义如下：

```
/* {{{ sapi_module_struct cgi_sapi_module
 */
static sapi_module_struct cgi_sapi_module = {
"cgi-fcgi",                     /* name */
```

```
"CGI/FastCGI",                  /* pretty name */

php_cgi_startup,                /* startup */
php_module_shutdown_wrapper,    /* shutdown */

sapi_cgi_activate,              /* activate */
sapi_cgi_deactivate,            /* deactivate */

sapi_cgibin_ub_write,           /* unbuffered write */
sapi_cgibin_flush,              /* flush */
NULL,                           /* get uid */
sapi_cgibin_getenv,             /* getenv */

php_error,                      /* error handler */

NULL,                           /* header handler */
sapi_cgi_send_headers,          /* send headers handler */
NULL,                           /* send header handler */

sapi_cgi_read_post,             /* read POST data */
sapi_cgi_read_cookies,          /* read Cookies */

sapi_cgi_register_variables,    /* register server variables */
sapi_cgi_log_message,           /* Log message */
NULL,                           /* Get request time */
NULL,                           /* Child terminate */

STANDARD_SAPI_MODULE_PROPERTIES
};
/* }}} */
```

同样，以读取cookie为例，当我们在CGI环境下，在PHP中调用读取Cookie时，最终获取的数据的位置是在激活SAPI时。它所调用的方法是read_cookies。由SAPI实现来实现获取cookie，这样各个不同的SAPI就能根据自己的需要来实现一些依赖环境的方法。

```
SG(request_info).cookie_data = sapi_module.read_cookies(TSRMLS_C);
```

所有使用PHP的场合都需要定义自己的SAPI，例如在第一小节的Apache模块方式中，sapi_module是apache2_sapi_module，其对应read_cookies方法的是php_apache_sapi_read_cookies函数，而在我们这里，读取cookie的函数是sapi_cgi_read_cookies。从sapi_module结构可以看出flush对应的是sapi_cli_flush，在win或非win下，flush对应的操作不同，在win下，如果输出缓存失败，则会和嵌入式的处理一样，调用php_handle_aborted_connection进入中断处理程序，而其它情况则是没有任何处理程序。这个区别通过cli_win.c中的PHP_CLI_WIN32_NO_CONSOLE控制。

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

广义而言， 语言是一套采用共同符号、表达方式与处理规则。就编程语言而言， 编程语言也是特定规则的符号， 用来传达特定的信息， 自然语言是人与人之间沟通的渠道， 而编程语言则是机器之间， 人与机器之间的沟通渠道。人有非常复杂的语言能力， 语言本身也在不断的进化， 人之间能够理解复杂的语言规则， 而计算机并没有这么复杂的系统， 它们只能接受指令执行操作， 编程语言则是机器和人(准确说是程序员)之间的桥梁， 编程语言的作用就是将语言的特定符号和处理规则进行翻译， 由编程语言来处理这些规则，

目前有非常多的编程语言， 不管是静态语言还是动态语言都有固定的工作需要做： 将代码编译为目标指令， 而编译过程就是根据语言的语法规则来进行翻译， 我们可以选择手动对代码进行解析， 但这是一个非常枯燥而容易出错的工作， 尤其是对于一个完备的编程语言而言， 由此就出现了像lex/yacc这类的编译器生成器。

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， 也就是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。

下面以php中echo语句的编译为例：echo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的语法规则如下：

```
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。在上例中当匹配到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，其他的语法规则也类似。这和C语言或者Java的编译器类似，不过GCC等编译器时将代码编译为机器码，Java编译器将代码编译为字节码。

> 更多关于lex/yacc的内容请参考Yacc 与Lex 快速入门

下面将介绍PHP中的opcode。

opcode

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的opcode就是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

在PHP实现内部，opcode由如下的结构体表示：

```
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。

例如如下代码是在编译器遇到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。这样在最终执行这条opcode的时候， Zend引擎能获取到足够的信息以便输出内容。

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。虚拟机相关章节的详细介绍。

> PHP有三种方式来进行opcode的处理:CALL，SWITCH和GOTO，PHP默认使用CALL的方式，也就是函数调用的方式，由于opcode执行是每个PHP程序频繁需要进行的操作，可以使用SWITCH或者GOTO的方式来分发，通常GOTO的效率相对会高一些，不过效率是否提高依赖于不同的CPU。

opcode处理函数查找

从上一小节读者可以了解到opcode在PHP内部的实现，那怎么找到某个opcode的处理函数呢？为了方便读者在追踪代码的过程中找到各种opcode对应的处理函数实现，下面介绍几种方法。

> 从PHP5.1开始，PHP对opcode的分发方式可以用户自定义，分为CALL，SWITCH和GOTO三种类型。默认使用的CALL的方式，本文也应用于这种方式。有关Zend虚拟机的介绍请阅读后面相关内容。

Debug法

在学习研究PHP内核的过程中，经常通过opcode来查看代码的执行顺序，opcode的执行由在文件Zend/zend_vm_execute.h中的execute函数执行。

```
ZEND_API void execute(zend_op_array *op_array TSRMLS_DC)
{
...
zend_vm_enter:
....
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;
            }
}
...
}
```

在执行的过程中，EX(opline)->handler（展开后为 *execute_data->opline->handler）存储了处理当前操作的函数指针。 使用gdb调试， 在execute函数处增加断电， 使用p命令可以打印出类似这样的结果：

```
(gdb) p *execute_data->opline->handler
$1 = {int (zend_execute_data *)} 0x10041f394 <ZEND_NOP_SPEC_HANDLER>
```

这样就可以方便的知道当前要执行的处理函数了， 这种debug的方法。这种方法比较麻烦， 需要使用gdb来调试。

计算法

在PHP内部有一个函数用来快速的返回特定opcode对应的opcode处理函数指针：zend_vm_get_opcode_handler()函数：

```
static opcode_handler_t
zend_vm_get_opcode_handler(zend_uchar opcode, zend_op* op)
{
        static const int zend_vm_decode[] = {
            _UNUSED_CODE, /* 0                */
            _CONST_CODE,  /* 1 = IS_CONST     */
            _TMP_CODE,    /* 2 = IS_TMP_VAR   */
            _UNUSED_CODE, /* 3                */
            _VAR_CODE,    /* 4 = IS_VAR       */
            _UNUSED_CODE, /* 5                */
            _UNUSED_CODE, /* 6                */
            _UNUSED_CODE, /* 7                */
            _UNUSED_CODE, /* 8 = IS_UNUSED    */
            _UNUSED_CODE, /* 9                */
            _UNUSED_CODE, /* 10               */
            _UNUSED_CODE, /* 11               */
            _UNUSED_CODE, /* 12               */
            _UNUSED_CODE, /* 13               */
            _UNUSED_CODE, /* 14               */
            _UNUSED_CODE, /* 15               */
            _CV_CODE      /* 16 = IS_CV       */
        };
        return zend_opcode_handlers[
            opcode * 25 + zend_vm_decode[op->op1.op_type] * 5
                    + zend_vm_decode[op->op2.op_type]];
}
```

由上面的代码可以看到， opcode到php内部函数指针的查找是由下面的公式来进行的：

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

然后将其计算的数值作为索引到zend_init_opcodes_handlers数组中进行查找。 不过这个数组实在是太大了， 有3851个元素， 手动查找和计算都比较麻烦。

命名查找法

上面的两种方法其实都是比较麻烦的， 在定位某一opcode的实现执行代码的过程中， 都不得不对程序进行执行或者计算中间值。而在追踪的过程中， 笔者发现处理函数名称是有一定规则的。 这里以函数调用的opcode为例， 调用某函数的opcode及其对应在php内核中实现的处理函数如下：

```
//函数调用：
DO_FCALL  ==>  ZEND_DO_FCALL_SPEC_CONST_HANDLER

//变量赋值：
ASSIGN     =>      ZEND_ASSIGN_SPEC_VAR_CONST_HANDLER
                   ZEND_ASSIGN_SPEC_VAR_TMP_HANDLER
                   ZEND_ASSIGN_SPEC_VAR_VAR_HANDLER
                   ZEND_ASSIGN_SPEC_VAR_CV_HANDLER
//变量加法：
ASSIGN_SUB =>   ZEND_ASSIGN_SUB_SPEC_VAR_CONST_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_VAR_TMP_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_VAR_VAR_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_VAR_UNUSED_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_VAR_CV_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_UNUSED_CONST_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_UNUSED_TMP_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_UNUSED_VAR_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_UNUSED_UNUSED_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_UNUSED_CV_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_CV_CONST_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_CV_TMP_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_CV_VAR_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_CV_UNUSED_HANDLER,
                    ZEND_ASSIGN_SUB_SPEC_CV_CV_HANDLER,
```

在上面的命名就会发现， 其实处理函数的命名是有以下规律的：

```
ZEND_[opcode]_SPEC_(变量类型1)_(变量类型2)_HANDLER
```

这里的变量类型1和变量类型2是可选的， 如果同时存在， 那就是左值和右值， 归纳有下几类： VAR TMP CV UNUSED CONST 这样可以根据相关的执行场景来判定。

日志记录法

这种方法是上面**计算法**的升级， 同时也是比较精准的方式。在**zend_vm_get_opcode_handler** 方法中添加以下代码：

```
static opcode_handler_t
zend_vm_get_opcode_handler(zend_uchar opcode, zend_op* op)
{
        static const int zend_vm_decode[] = {
            _UNUSED_CODE, /* 0              */
            _CONST_CODE,  /* 1 = IS_CONST   */
            _TMP_CODE,    /* 2 = IS_TMP_VAR */
```

```
            _UNUSED_CODE, /* 3                */
            _VAR_CODE,    /* 4 = IS_VAR       */
            _UNUSED_CODE, /* 5                */
            _UNUSED_CODE, /* 6                */
            _UNUSED_CODE, /* 7                */
            _UNUSED_CODE, /* 8 = IS_UNUSED    */
            _UNUSED_CODE, /* 9                */
            _UNUSED_CODE, /* 10               */
            _UNUSED_CODE, /* 11               */
            _UNUSED_CODE, /* 12               */
            _UNUSED_CODE, /* 13               */
            _UNUSED_CODE, /* 14               */
            _UNUSED_CODE, /* 15               */
            _CV_CODE      /* 16 = IS_CV       */
        };

    //很显然，我们把opcode和相对应的写到了/tmp/php.log文件中
    int op_index;
    op_index = opcode * 25 + zend_vm_decode[op->op1.op_type] * 5 +
zend_vm_decode[op->op2.op_type];

    FILE *stream;
    if((stream = fopen("/tmp/php.log", "a+")) != NULL){
        fprintf(stream, "opcode: %d , zend_opcode_handlers_index:%d\n",
opcode, op_index);
    }
    fclose(stream);


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

然后，就可以在**/tmp/php.log**文件中生成类似如下结果:

```
opcode: 38 , zend_opcode_handlers_index:970
```

前面的数字是opcode的，我们可以这里查到： http://php.net/manual/en/internals2.opcodes.list.php 后面的数字是static const opcode_handler_t labels[] 索引，里面对应了处理函数的名称，对应源码文件是：Zend/zend_vm_execute.h （第30077行左右）。这是一个超大的数组，php5.3.4中有3851个元素，在上面的例子里，看样子我们要数到第970个了，当然，有很多种方法来避免人工去计算，这里就不多介绍了。

第四节 小结

本章对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过程了解了PHP脚本是怎样被执行的。比如opcode的编译和执行等。

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

世界上唯一不变的就是变化。

现代编程语言中的基本元素主要有：变量，流程控制接口，函数等等。我能能否不使用变量来编写程序呢？这显然是可以的，例如：

```
<?php
echo "Hello TIPI Readers";
```

这个程序很简单，输出一个字符串内容。

就和我们仅仅使用二进制也能编程一样，不使用变量也能完成大部分的工作，不使用变量我们的程序将丧失极大的灵活性，变量可以让我们将值存储起来，以便在程序的其他地方使用，或者通过计算保存新的值。变量具有三个基本特性：

1. **名称** 变量的标示符。就像小狗一样，主人可能会给这些小狗起个喜欢的名称。变量命名上，PHP继承了Perl的语法风格，变量以美元符号开始，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。PHP同时还支持复合变量，也就是类似$$a的变量，它会进行两次的解释。这给PHP带来了非常灵活的动态特性。

2. **类型** 变量的类型，就像小狗的品种，不同的小狗血统可能会不一样，有的聪明，有的会购物等等。在很多静态语言中，变量在定义时就指定了，在程序运行过程中都不允许进行变更，那如果你有一只能随便指定品种的小狗会不会很拉风呢;-) PHP就是这样，属于弱类型语言，可以随便赋予它任何类型的值。

3. **值内容。** 这是标示所代表的具体内容。这就像是实实在在的小狗的这个实物。你可以给任何一条小狗起名为：小七，在编程语言中也是如此，你可以给变量赋予它 所能表示范围的值。不过在同一时间，变量只能有一个值。

   PHP中组成变量名的字母可以是英文字母 a-z，A-Z，还可以是 ASCII 字符从 127 到 255（0x7f-0xff）。变量名是区分大小写的。

除了变量本身，在PHP中我们经常会接触到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本身的变量低层存储结构以及弱类型系统的实现，以及这些类型之间的相互转换等。

先看一段PHP代码：

```
<?php
$foo = 10;
$bar = 20;

function change() {
    global $foo;
    $bar = 0;
    $foo++;
}

change();
echo $foo, ' ', $bar;
```

运行代码会输出**11 20**。可是为什么会有这样的输出呢？变量在PHP的内部是如何实现的呢？ 变量的作用域又是怎么实现的呢？ 这是本章将对围绕**变量**这个主题展开讨论，下面我们从最基本的变量实现开始。

> 不是所有编程语言中的变量的值都可以改变的。想想我们学过的数学中的变量。 他们的值也是不可改变的。例如： x + y = 10; 变量x和y的值是不能发生变化的。 在某个具体场景，也就是某个方程式中只有表示特定的值，变量的值不能改变的好处是： 这样就能尽可能少的产生副作用， 在Erlang语言中就是如此，它是一门函数式编程语言，非常值得学习。

第一节 变量的结构和类型

前言中提到变量的三个基本特性，其中的有一个特性为变量的类型，变量都有特定的类型， 如：字符串、数组、对象等等。编程语言的类型系统可以分为强类型和弱类型两种：

强类型语言是一旦某个变量被申明为某个类型的变量，则在程序运行过程中，该不能将该变量的类型以外的值赋予给它 （当然并不完全如此，这可能会涉及到类型的转换，后面的小节会有相应介绍），C/C++/Java等语言就属于这类。

PHP及Ruby，JavaScript等脚本语言属于弱类型语言：一个变量可以表示任意的数据类型。

PHP之所以成为一个简单而强大的语言，很大一部分的原因是它拥有弱类型的变量。 但是有些时候这也是一把双刃剑，使用不当也会带来一些问题。就像仪器一样，越是功能强大， 出现错误的可能性也就越大。

在官方的PHP实现内部，所有变量使用同一种数据结构(zval)来保存，而这个结构同时表示PHP中的各种数据类型。 它不仅仅包含变量的值，也包含变量的类型。这就是PHP弱类型的核心。

那zval结构具体是如何实现弱类型的呢，下面我们一起来揭开面纱。

一. PHP变量类型及存储结构

PHP在声明或使用变量的时候，并不需要显式指明其数据类型。

PHP是弱类型语言，这并不表示PHP没有类型，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*

官方PHP是用C实现的，而C是强类型的语言，那这是怎么实现PHP中的弱类型的呢？

1. 变量存储结构

变量的值存储到以下所示zval结构体中。 zval结构体定义在Zend/zend.h文件，其结构如下：

```
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;
};
```

PHP使用这个结构来存储变量的所有数据。和其他编译性静态语言不同， PHP在存储变量时将PHP用户空间。 的变量类型也保存在同一个结构体中。这样我们就能通过这些信息获取到变量的类型。

zval结构体中有四个字段， 其含义分别为：

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	存储变量的值	
type	变量具体的类型	

> 在PHP5.3之后， 引入了新的垃圾收集机制， 引用计数和引用的字段名改为refcount__gc和is_ref__gc。在此之前为refcount和is__ref。

而变量的值则存储在另外一个结构体zvalue_value中。值存储见下面的介绍。

> PHP用户空间指的在PHP语言这一层面， 而本书中大部分地方都在探讨PHP的实现。 这些实现可以理解为内核空间。由于PHP使用C实现， 而这个空间的范畴就会限制在C语言。 而PHP用户空间则会受限于PHP语法及功能提供的范畴之内。
> 例如有些PHP扩展会提供一些PHP函数或者类， 这就是向PHP用户空间导出了方法或类。

2.变量类型:

zval结构体的type字段就是实现弱类型最关键的字段了， type的值可以为： IS_NULL、IS_BOOL、IS_LONG、IS_DOUBLE、IS_STRING、IS_ARRAY、IS_OBJECT和IS_RESOURCE 之一。 从字面上就很好理解， 他们只是类型的唯一标示， 根据类型的不同将不同的值存储到value字段。 除此之外， 和他们定义在一起的类型还有IS_CONSTANT和IS_CONSTANT_ARRAY。

这和我们设计数据库时的做法类似， 为了避免重复设计类似的表， 使用一个标示字段来记录不同类型的数据。

二.变量的值存储

前面提到变量的值存储在zvalue_value联合体中， 结构体定义如下：

```
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;
```

这里使用联合体而不是用结构体是出于空间利用率的考虑，因为一个变量同时只能属于一种类型。 如果使用结构体的话将会不必要的浪费空间，而PHP中的所有逻辑都围绕变量来进行的，这样的话， 内存浪费将是十分大的。这种做法成本小但收益非常大。

各种类型的数据会使用不同的方法来进行变量值的存储，其对应赋值方式如下：

- 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。
float	ZVAL_DOUBLE	`Z_TYPE_P(z)=IS_BOOL/LONG;  Z_LVAL_P(z)=((b)!=0);`
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String

字符串的类型标示和其他数据类型一样，不过在存储字符串时多了一个字符串长度的字段。

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

C中字符串是以\0结尾的字符数组，这里多存储了字符串的长度，这和我们在设计数据库时增加的冗余字段异曲同工。 因为要实时获取到字符串的长度的时间复杂度是O(n)，而字符串的操作在PHP中是非常频繁的，这样能避免重复计算字符串的长度， 这能节省大量的时间，是空间换时间的做法。
这么看在PHP中strlen()函数可以在常数时间内获取到字符串的长度。 计算机语言中字符串的操作都非常之多，所以大部分高级语言中都会存储字符串的长度。

- 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， 数组的值存储在zvalue_value.ht字段中，它是一个HashTable类型的数据。 PHP的数组使用哈希表来存储关联数据。哈希表是一种高效的键值对存储结构。PHP的哈希表实现中使用了两个数据结构HashTable和Bucket。 PHP所有的工作都由哈希表实现，在下节HashTable中将进行哈希表基本概念的介绍以及PHP的哈希表实现。

- 对象Object

在面向对象语言中，我们能自己定义自己需要的数据类型，包括类的属性，方法等数据。而对象则是类的一个具体实现。对象有自身的状态和所能完成的操作。

PHP的对象是一种复合型的数据，使用一种zend_object_value的结构体来存放。其定义如下：

```
typedef struct _zend_object_value {
    zend_object_handle handle;  //  unsigned int类型，EG(objects_store).object_buckets的索引
    zend_object_handlers *handlers;
} zend_object_value;
```

PHP的对象只有在运行时才会被创建，前面的章节介绍了EG宏，这是一个全局结构体用于保存在运行时的数据。其中就包括了用来保存所有被创建的对象的对象池，EG(objects_store)，而object对象值内容的zend_object_handle域就是当前 对象在对象池中所在的索引，handlers字段则是将对象进行操作时的处理函数保存起来。这个结构体及对象相关的类的结构_zend_class_entry，将在第五章作详细介绍。

PHP的弱变量容器的实现方式是兼容并包的形式体现，针对每种类型的变量都有其对应的标记和存储空间。使用强类型的语言在效率上通常会比弱类型高，因为很多信息能在运行之前就能确定，这也能帮助排除程序错误。而这带来的问题是编写代码相对会受制约。

PHP主要的用途是作为Web开发语言，在普通的Web应用中瓶颈通常在业务和数据访问这一层。不过在大型应用下语言也会是一个关键因素。facebook因此就使用了自己的php实现。将PHP编译为C++代码来提高性能。不过facebook的hiphop并不是完整的php实现，由于它是直接将php编译为C++，有一些PHP的动态特性比如eval结构就无法实现。当然非要实现也是有方法的，hiphop不实现应该也是做了一个权衡。

哈希表(HashTable)

按图索骥。

PHP中使用最为频繁的数据类型非字符串和数组莫属，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，这里列举一下HashTable实现中出现的基本概念。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所在位置的函数。
- 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

链接法通过使用一个链表来保存slot值的方式来解决冲突，也就是当不同的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，这样哈希表就退化成了一个链表，这样的话操作链表的时间复杂度则成了O(n)，这样哈希表的性能优势就没有了，所以选择一个合适的哈希函数是最为关键的。

由于目前大部分的编程语言的哈希表实现都是开源的，大部分语言的哈希算法都是公开的算法，虽然目前的哈希算法都能良好的将key进行比较均匀的分布，而这个假使的前提是key是随机的，正是由于算法的确定性，这就导致了别有用心的黑客能利用已知算法的可确定性来构造一些特殊的key，让这些key都映射到 同一个槽位导致哈希表退化成单链表，导致程序的性能急剧下降，从而造成一些应用的吞吐能力急剧下降，尤其是对于高并发的应用影响很大，通过大量类似的请求可以让服务器遭受DoS(服务拒绝攻击)，这个问题一直就存在着，只是最近才被各个语言重视起来。

哈希冲突攻击利用的哈希表最根本的弱点是：**开源算法和哈希实现的确定性以及可预测性**，这样攻击者才可以利用特殊构造的key来进行攻击。要解决这个问题的方法则是让攻击者无法轻易构造 能够进行攻击的key序列。

> 在笔者编写这节内容的时候PHP语言也采取了相应的措施来防止这类的攻击，PHP采用的是一种 治标不治本的做法: 限制用户提交数据字段数量 这样可以避免大部分的攻击，不过应用程序通常会有很多的数据输入方式，比如，SOAP，REST等等，比如很多应用都会接受用户传入的JSON字符串，在执行json_decode()的时候也可能会遭受攻击。所以最根本的解决方法是让哈希表的碰撞key序列无法轻易的构造，目前PHP中还没有引入不增加额外的复杂性情况下的完美解决方案。

目前PHP中HashTable的哈希冲突解决方法就是链接法。

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。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， 在查找时也使用同样的策律来进行。

由于开放寻址法处理冲突的时候占用的是其他槽位的空间,这可能会导致后续的key在插入的时候更加容易出现 哈希冲突， 所以采用开放寻址法的哈希表的装载因子不能太高， 否则容易出现性能下降。

> *装载因子*是哈希表保存的元素数量和哈希表容量的比， 通常采用链接法解决冲突的哈希表的装载 因子最好不要大于1， 而采用开放寻址法的哈希表最好不要大于0.5。

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， 也就是前面所提到的的链接法。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，它的效果并不好，在实际场景下不会使用这种哈希算法，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。

有兴趣的读者可以运行本小节实现的哈希表实现,在输出日志中将看到很多的哈希冲突, 这是本例中使用的哈希算法过于简单造成的.

操作接口的实现

为了操作哈希表，实现了如下几个操作接口函数：

```
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);
```

下面以初始化、插入和获取操作函数为例：

```
int hash_init(HashTable *ht)
{
    ht->size        = HASH_TABLE_INIT_SIZE;
    ht->elem_num    = 0;
    ht->buckets     = (Bucket **)calloc(ht->size, sizeof(Bucket *));

    if(ht->buckets == NULL) return FAILED;

    LOG_MSG("[init]\tsize: %i\n", ht->size);

    return SUCCESS;
```

```
}
```

初始化的主要工作是为哈希表申请存储空间，函数中使用calloc函数的目的是确保 数据存储的槽为都初始化为0，以便后续在插入和查找时确认该槽为是否被占用。

```
int hash_insert(HashTable *ht, char *key, void *value)
{
    // check if we need to resize the hashtable
    resize_hash_table_if_needed(ht);

    int index = HASH_INDEX(ht, key);

    Bucket *org_bucket = ht->buckets[index];
    Bucket *tmp_bucket = org_bucket;

    // check if the key exits already
    while(tmp_bucket)
    {
        if(strcmp(key, tmp_bucket->key) == 0)
        {
            LOG_MSG("[update]\tkey: %s\n", key);
            tmp_bucket->value = value;

            return SUCCESS;
        }

        tmp_bucket = tmp_bucket->next;
    }

    Bucket *bucket = (Bucket *)malloc(sizeof(Bucket));

    bucket->key   = key;
    bucket->value = value;
    bucket->next  = NULL;

    ht->elem_num += 1;

    if(org_bucket != NULL)
    {
        LOG_MSG("[collision]\tindex:%d key:%s\n", index, key);
        bucket->next = org_bucket;
    }

    ht->buckets[index]= bucket;

    LOG_MSG("[insert]\tindex:%d key:%s\tht(num:%d)\n",
        index, key, 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。

由于在插入过程中可能会导致哈希表的元素个数表较多，如果超过了哈希表的容量，则说明肯定会出现碰撞，出现碰撞则会导致哈希表的性能下降，为此如果出现元素容量达到容量则需要进行扩容。由于所有的key都进行了哈希，扩容后哈希表不能简单的扩容，而需要重新将原有已插入的预算插入到新的容器中。

```
static void resize_hash_table_if_needed(HashTable *ht)
{
    if(ht->size - ht->elem_num < 1)
    {
        hash_resize(ht);
    }
}

static int hash_resize(HashTable *ht)
{
    // double the size
    int org_size = ht->size;
    ht->size = ht->size * 2;
    ht->elem_num = 0;

    LOG_MSG("[resize]\torg size: %i\tnew size: %i\n", org_size, ht->size);

    Bucket **buckets = (Bucket **)calloc(ht->size, sizeof(Bucket **));

    Bucket **org_buckets = ht->buckets;
    ht->buckets = buckets;

    int i = 0;
    for(i=0; i < org_size; ++i)
    {
        Bucket *cur = org_buckets[i];
        Bucket *tmp;
        while(cur)
        {
            // rehash: insert again
            hash_insert(ht, cur->key, cur->value);

            // free the org bucket, but not the element
            tmp = cur;
            cur = cur->next;
            free(tmp);
        }
    }
    free(org_buckets);

    LOG_MSG("[resize] done\n");

    return SUCCESS;
}
```

哈希表的扩容首先申请一块新的内存，大小为原来的2倍，然后重新将元素插入到哈希表中，读者会发现扩容的操作的代价为O(n)，不过这个问题不大，因为只有在到达哈希表容量的时候才会进行。

在查找时也使用插入同样的策略，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) goto failed;

    while(bucket)
    {
        if(strcmp(bucket->key, key) == 0)
        {
```

```
            LOG_MSG("[lookup]\t found %s\tindex:%i value: %p\n",
                key, index, bucket->value);
            *result = bucket->value;

            return SUCCESS;
        }

        bucket = bucket->next;
    }

failed:
    LOG_MSG("[lookup]\t key:%s\tfailed\t\n", key);
    return FAILED;
}
```

PHP中数组是基于哈希表实现的，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，这样是无法根据插入的先后顺序获取到这些元素的，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。具体内容在后一小节PHP中的HashTable中进行详细说明。上面的例子就是PHP中实现的一个精简版。

> 本小节的HashTable实例完整代码可以在$TIPI_ROOT/book/sample/chapt03/03-01-01-hashtable目录中找到。或者在github上浏览:
> https://github.com/reeze/tipi/tree/master/book/sample/chapt03/03-01-01-hashtable

参考文献

- 《Data.Structures.and.Algorithm.Analysis.in.C》
- 《算法导论: 第二版》

PHP的哈希表实现

上一节已经介绍了哈希表的基本原理并实现了一个基本的哈希表，而在实际项目中，对哈希表的需求远不止那么简单。对性能，灵活性都有不同的要求。下面我们看看PHP中的哈希表是怎么实现的。

PHP的哈希实现

PHP内核中的哈希表是十分重要的数据结构，PHP的大部分的语言特性都是基于哈希表实现的，例如：变量的作用域、函数表、类的属性、方法等，Zend引擎内部的很多数据都是保存在哈希表中的。

数据结构及说明

上一节提到PHP中的哈希表是使用拉链法来解决冲突的，具体点讲就是使用链表来存储哈希到同一个槽位的数据，Zend为了保存数据之间的关系使用了双向列表来链接元素。

哈希表结构

PHP中的哈希表实现在Zend/zend_hash.c中，还是按照上一小节的方式，先看看PHP实现中的数据结构，PHP使用如下两个数据结构来实现哈希表，HashTable结构体用于保存整个哈希表需要的基本信息，而Bucket结构体用于保存具体的数据内容，如下：

```
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; // 下一个数字索引的位置
    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;     // 在删除元素时执行的回调函数，用于资源的释放
    zend_bool persistent;        //指出了Bucket内存分配的方式。如果persisient为TRUE，则使用操作系统本身的内存分配函数为Bucket分配内存，否则使用PHP的内存分配函数。
    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;
```

nTableSize字段用于标示哈希表的容量，哈希表的初始容量最小为8。首先看看哈希表的初始化函数：

```
ZEND_API int _zend_hash_init(HashTable *ht, uint nSize, hash_func_t pHashFunction,
                    dtor_func_t pDestructor, zend_bool persistent ZEND_FILE_LINE_DC)
{
    uint i = 3;
    //...
    if (nSize >= 0x80000000) {
        /* prevent overflow */
        ht->nTableSize = 0x80000000;
    } else {
        while ((1U << i) < nSize) {
            i++;
        }
        ht->nTableSize = 1 << i;
    }
    // ...
    ht->nTableMask = ht->nTableSize - 1;

    /* Uses ecalloc() so that Bucket* == NULL */
    if (persistent) {
        tmp = (Bucket **) calloc(ht->nTableSize, sizeof(Bucket *));
        if (!tmp) {
            return FAILURE;
        }
        ht->arBuckets = tmp;
    } else {
        tmp = (Bucket **) ecalloc_rel(ht->nTableSize, sizeof(Bucket *));
        if (tmp) {
            ht->arBuckets = tmp;
        }
    }
```

```
    return SUCCESS;
}
```

例如如果设置初始大小为10， 则上面的算法将会将大小调整为16。也就是始终将大小调整为接近初始大小的 2的整数次方。

为什么会做这样的调整呢？我们先看看HashTable将哈希值映射到槽位的方法， 上一小节我们使用了取模的方式来将哈希值 映射到槽位， 例如大小为8的哈希表， 哈希值为100， 则映射的槽位索引为: 100 % 8 = 4， 由于索引通常从0开始， 所以槽位的索引值为3， 在PHP中使用如下的方式计算索引：

```
h = zend_inline_hash_func(arKey, nKeyLength);
nIndex = h & ht->nTableMask;
```

从上面的_zend_hash_init()函数中可知， ht->nTableMask的大小为ht->nTableSize -1。 这里使用&操作而不是使用取模， 这是因为是相对来说取模操作的消耗和按位与的操作大很多。

> mask的作用就是将哈希值映射到槽位所能存储的索引范围内。 例如：某个key的索引值是21， 哈希表的大小为8， 则mask为7， 则求与时的二进制表示为： 10101 & 111 = 101 也就是十进制的5。 因为2的整数次方-1的二进制比较特殊：后面N位的值都是1， 这样比较容易能将值进行映射， 如果是普通数字进行了二进制与之后会影响哈希值的结果。那么哈希函数计算的值的平均分布就可能出现影响。

设置好哈希表大小之后就需要为哈希表申请存储数据的空间了， 如上面初始化的代码， 根据是否需要持久保存而条用了不同的内存申请方法， 是需要需要持久体现的是在前面PHP生命周期里介绍的： 持久内容能在多个请求之间可访问， 而如果是非持久存储则会在请求结束时释放占用的空间。 具体内容将在内存管理章节中进行介绍。

HashTable中的nNumOfElements字段很好理解， 每插入一个元素或者unset删掉元素时会更新这个字段。 这样在进行count()函数统计数组元素个数时就能快速的返回。

nNextFreeElement字段非常有用。先看一段PHP代码：

```
<?php
$a = array(10 => 'Hello');
$a[] = 'TIPI';
var_dump($a);

// ouput
array(2) {
  [10]=>
  string(5) "Hello"
  [11]=>
  string(5) "TIPI"
}
```

PHP中可以不指定索引值向数组中添加元素， 这时将默认使用数字作为索引， 和C语言中的枚举类似， 而这个元素的索引到底是多少就由nNextFreeElement字段决定了。 如果数组中存在了数字key， 则会默认使用最新使用的key + 1， 例如上例中已经存在了10作为key的元素， 这样新插入的默认索引就为11了。

下面看看保存哈希表数据的槽位数据结构体：

```
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;    // 整个哈希表该元素的上一个元素
    struct bucket *pNext;        // 存放在同一个hash Bucket内的下一个元素
    struct bucket *pLast;        // 同一个哈希bucket的上一个元素
    // 保存当前值所对于的key字符串，这个字段只能定义在最后，实现变长结构体
    char arKey[1];
} Bucket;
```

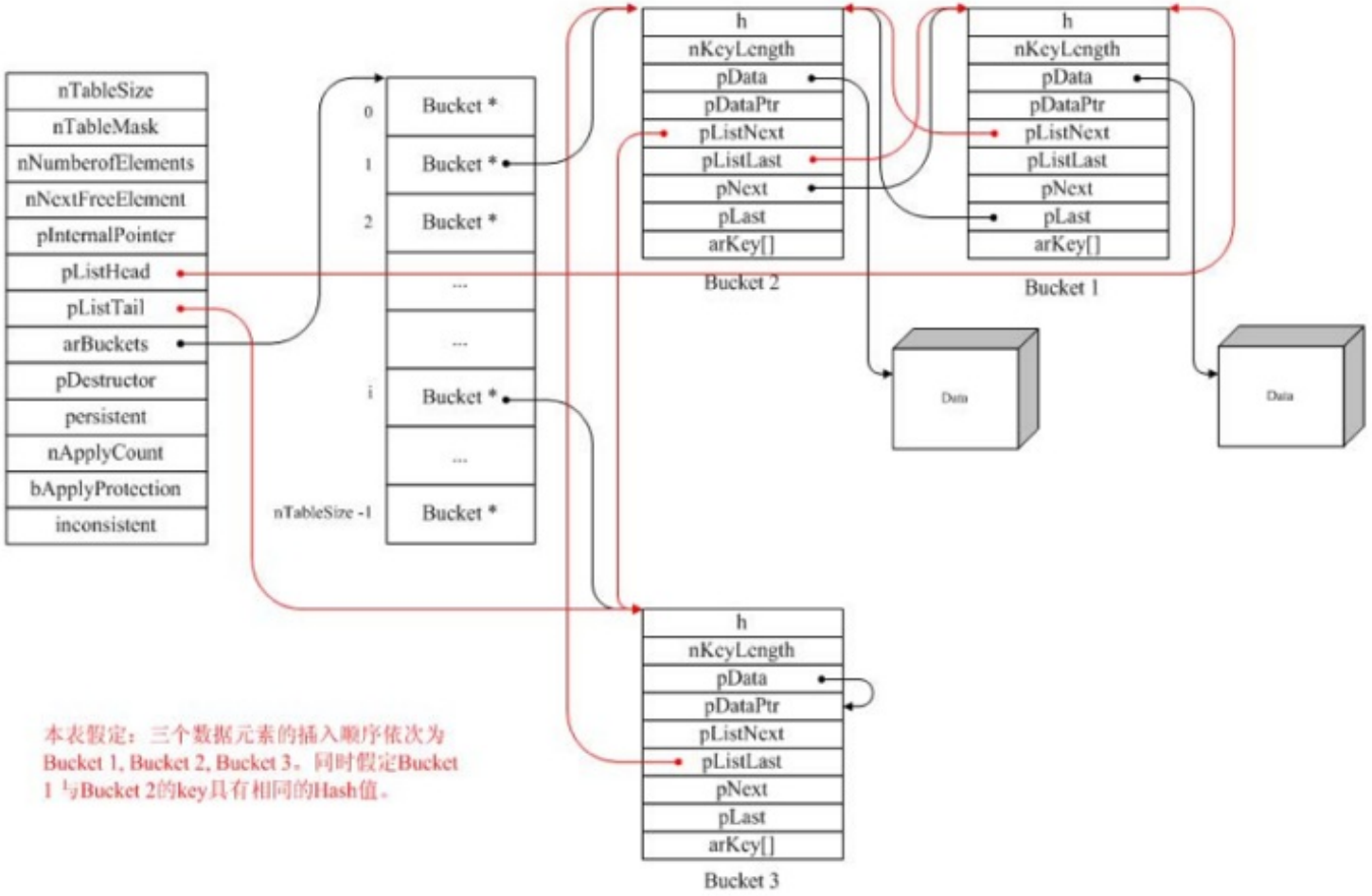
如上面各字段的注释。h字段保存哈希表key哈希后的值。这里保存的哈希值而不是在哈希表中的索引值， 这是因为索引值和哈希表的容量有直接关系，如果哈希表扩容了，那么这些索引还得重新进行哈希在进行索引映射， 这也是一种优化手段。 在PHP中可以使用字符串或者数字作为数组的索引。 数字索引直接就可以作为哈希表的索引，数字也无需进行哈希处理。h字段后面的nKeyLength字段是作为key长度的标示， 如果索引是数字的话，则nKeyLength为0。在PHP数组中如果索引字符串可以被转换成数字也会被转换成数字索引。 **所以在PHP中例如'10'，'11'这类的字符索引和数字索引10， 11没有区别。**

上面结构体的最后一个字段用来保存key的字符串，而这个字段却申明为只有一个字符的数组， 其实这里是一种长见的变长结构体，主要的目的是增加灵活性。 以下为哈希表插入新元素时申请空间的代码

```
p = (Bucket *) pemalloc(sizeof(Bucket) - 1 + nKeyLength, ht->persistent);
if (!p) {
    return FAILURE;
}
memcpy(p->arKey, arKey, nKeyLength);
```

如代码， 申请的空间大小加上了字符串key的长度，然后把key拷贝到新申请的空间里。 在后面比如需要进行hash查找的时候就需要对比key这样就可以通过对比p->arKey和查找的key是否一样来进行数据的查找。申请空间的大小-1是因为结构体内本身的那个字节还是可以使用的。

在PHP5.4中将这个字段定义成const char* arKey类型了。



Zend引擎哈希表结构和关系

上图来源于网络。

- Bucket结构体维护了两个双向链表，pNext和pLast指针分别指向本槽位所在的链表的关系。
- 而pListNext和pListLast指针指向的则是整个哈希表所有的数据之间的链接关系。HashTable结构体中的pListHead和pListTail则维护整个哈希表的头元素指针和最后一个元素的指针。

PHP中数组的操作函数非常多，例如：array_shift()和array_pop()函数，分别从数组的头部和尾部弹出元素。哈希表中保存了头部和尾部指针，这样在执行这些操作时就能在常数时间内找到目标。PHP中还有一些使用的相对不那么多的数组操作函数：next()，prev()等的循环中，哈希表的另外一个指针就能发挥作用了：pInternalPointer，这个用于保存当前哈希表内部的指针。这在循环时就非常有用。

如图中左下角的假设，假设依次插入了Bucket1，Bucket2，Bucket3三个元素：

1. 插入Bucket1时，哈希表为空，经过哈希后定位到索引为1的槽位。此时的1槽位只有一个元素Bucket1。其中Bucket1的pData或者pDataPtr指向的是Bucket1所存储的数据。此时由于没有链接关系。pNext，pLast，pListNext，pListLast指针均为空。同时在HashTable结构体中也保存了整个哈希表的第一个元素指针，和最后一个元素指针，此时HashTable的pListHead和pListTail指针均指向Bucket1。
2. 插入Bucket2时，由于Bucket2的key和Bucket1的key出现冲突，此时将Bucket2放在双链表的前面。由于Bucket2后插入并置于链表的前端，此时Bucket2.pNext指向Bucket1，由于Bucket2后插入。Bucket1.pListNext指向Bucket2，这时Bucket2就是哈希表的最后一个元素，这是HashTable.pListTail指向Bucket2。

3. 插入Bucket3，该key没有哈希到槽位1，这时Bucket2.pListNext指向Bucket3，因为Bucket3后插入。同时HashTable.pListTail改为指向Bucket3。

简单来说就是哈希表的Bucket结构维护了哈希表中插入元素的先后顺序，哈希表结构维护了整个哈希表的头和尾。在操作哈希表的过程中始终保持预算之间的关系。

哈希表的操作接口

和上一节类似，将简单介绍PHP哈希表的操作接口实现。提供了如下几类操作接口：

- 初始化操作，例如zend_hash_init()函数，用于初始化哈希表接口，分配空间等。
- 查找，插入，删除和更新操作接口，这是比较常规的操作。
- 迭代和循环，这类的接口用于循环对哈希表进行操作。
- 复制，排序，倒置和销毁等操作。

本小节选取其中的插入操作进行介绍。在PHP中不管是对数组的添加操作（zend_hash_add），还是对数组的更新操作（zend_hash_update），其最终都是调用_zend_hash_add_or_update函数完成，这在面向对象编程中相当于两个公有方法和一个公共的私有方法的结构，以实现一定程度上的代码复用。

ZEND_API int _zend_hash_add_or_update(HashTable *ht, const char *arKey, uint nKeyLength, void *pData, uint nDataSize, void **pDest, int flag ZEND_FILE_LINE_DC) { //...省略变量初始化和nKeyLength <=0 的异常处理

```
h = zend_inline_hash_func(arKey, nKeyLength);
nIndex = h & ht->nTableMask;

p = ht->arBuckets[nIndex];
while (p != NULL) {
    if ((p->h == h) && (p->nKeyLength == nKeyLength)) {
        if (!memcmp(p->arKey, arKey, nKeyLength)) { //  更新操作
            if (flag & HASH_ADD) {
                return FAILURE;
            }
            HANDLE_BLOCK_INTERRUPTIONS();

            //..省略debug输出
            if (ht->pDestructor) {
                ht->pDestructor(p->pData);
            }
            UPDATE_DATA(ht, p, pData, nDataSize);
            if (pDest) {
                *pDest = p->pData;
            }
            HANDLE_UNBLOCK_INTERRUPTIONS();
            return SUCCESS;
        }
    }
    p = p->pNext;
}

p = (Bucket *) pemalloc(sizeof(Bucket) - 1 + nKeyLength, ht->persistent);
if (!p) {
    return FAILURE;
}
memcpy(p->arKey, arKey, nKeyLength);
p->nKeyLength = nKeyLength;
```

```
INIT_DATA(ht, p, pData, nDataSize);
p->h = h;
CONNECT_TO_BUCKET_DLLIST(p, ht->arBuckets[nIndex]); //Bucket双向链表操作
if (pDest) {
    *pDest = p->pData;
}

HANDLE_BLOCK_INTERRUPTIONS();
CONNECT_TO_GLOBAL_DLLIST(p, ht);     // 将新的Bucket元素添加到数组的链接表的最后面
ht->arBuckets[nIndex] = p;
HANDLE_UNBLOCK_INTERRUPTIONS();

ht->nNumOfElements++;
ZEND_HASH_IF_FULL_DO_RESIZE(ht);         /*  如果此时数组的容量满了，则对其进行扩容。
*/
return SUCCESS;
```

}

整个写入或更新的操作流程如下：

1. 生成hash值，通过与nTableMask执行与操作，获取在arBuckets数组中的Bucket。
2. 如果Bucket中已经存在元素，则遍历整个Bucket，查找是否存在相同的key值元素，如果有并且是update调用，则执行update数据操作。
3. 创建新的Bucket元素，初始化数据，并将新元素添加到当前hash值对应的Bucket链表的最前面（CONNECT_TO_BUCKET_DLLIST）。
4. 将新的Bucket元素添加到数组的链接表的最后面（CONNECT_TO_GLOBAL_DLLIST）。
5. 将元素个数加1，如果此时数组的容量满了，则对其进行扩容。这里的判断是依据nNumOfElements和nTableSize的大小。 如果nNumOfElements > nTableSize则会调用zend_hash_do_resize以2X的方式扩容（nTableSize << 1）。

哈希表的性能

其他语言中的HashTable实现

Ruby使用的st库，Ruby中的两种hash实现

参考资料

http://nikic.github.com/2012/03/28/Understanding-PHPs-internal-array-implementation.html

链表简介

Zend引擎中实现了很多基本的数据结构，这些接口贯穿PHP和Zend引擎的始末，这些数据结构以及相应的操作接口都可以作为通用的接口来使用。本小节再简单描述一下

在Zend引擎中HashTable的使用非常频繁，这得益于他良好的查找性能，如果读者看过 前一小节会知

道哈希表会预先分配内容以提高性能，而很多时候数据规模不会很大，固然使用哈希表能提高查询性能，但是某些场景下并不会对数据进行随机查找，这时使用哈希表就有点浪费了。

Zend引擎中的链表是双链表，通过双链表的任意节点都能方便的对链表进行遍历。

Zend引擎的哈希表实现是哈希表和双链表的混合实现，这也是为了方便哈希表的遍历。

链表的实现很简单，通常只需要三个关键元素：

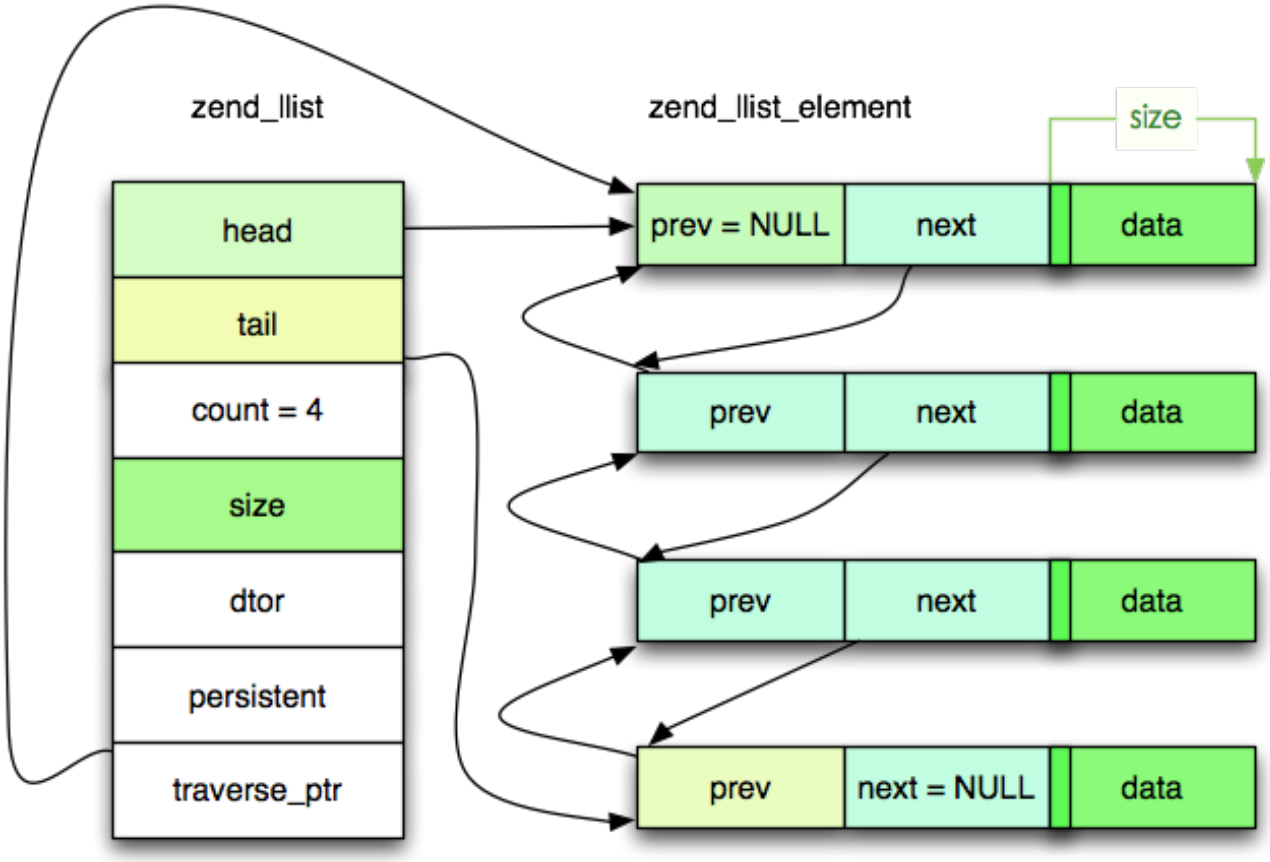
1. 指向上个元素的指针
2. 指向下个元素的指针
3. 数据容器

Zend引擎的实现也很简单，如下两个是核心的数据接口，第一个是元素节点，第二个是链表容器。

```
typedef struct _zend_llist_element {
    struct _zend_llist_element *next;
    struct _zend_llist_element *prev;
    char data[1]; /* Needs to always be last in the struct */
} zend_llist_element;

typedef struct _zend_llist {
    zend_llist_element *head;
    zend_llist_element *tail;
    size_t count;
    size_t size;
    llist_dtor_func_t dtor;
    unsigned char persistent;
    zend_llist_element *traverse_ptr;
} zend_llist;
```

节点元素只含有前面提到的3个元素，第三个字段data和哈希表的实现一样，是一个柔性结构体。



Zend zend_llist结构

如上图所示，data字段的空间并不是只有一个字节，我们先看看元素插入的实现：

```
ZEND_API void zend_llist_add_element(zend_llist *l, void *element)
{
    zend_llist_element *tmp = pemalloc(sizeof(zend_llist_element)+l->size-1, l->persistent);

    tmp->prev = l->tail;
    tmp->next = NULL;
    if (l->tail) {
        l->tail->next = tmp;
    } else {
        l->head = tmp;
    }
    l->tail = tmp;
    memcpy(tmp->data, element, l->size);

    ++l->count;
}
```

如方法第一行所示，申请空间是额外申请了`l->size - 1`的空间。`l->size`是在链表创建是指定的，`zend_llist_element`结构体最后那个字段的注释提到这个字段必须放到最后也是这个原因，例如curl扩展中的例子：`zend_llist_init(&(*ch)->to_free->slist, sizeof(struct curl_slist), (llist_dtor_func_t) curl_free_slist, 0);`，`size`指的是要插入元素的空间大小，这样不同的链表就可以插入不同大小的元素了。

为了提高性能增加了链表头和尾节点地址，以及链表中元素的个数。

最后的traverse_ptr 字段是为了方便在遍历过程中记录当前链表的内部指针，和哈希表中的:`Bucket *pInternalPointer;`字段一个作用。

操作接口

操作接口比较简单，本文不打算介绍接口的使用，这里简单说一下PHP源代码中的一个小的约定，

如下为基本的链表遍历操作接口：

```
/* traversal */
ZEND_API void *zend_llist_get_first_ex(zend_llist *l, zend_llist_position
*pos);
ZEND_API void *zend_llist_get_last_ex(zend_llist *l, zend_llist_position *pos);
ZEND_API void *zend_llist_get_next_ex(zend_llist *l, zend_llist_position *pos);
ZEND_API void *zend_llist_get_prev_ex(zend_llist *l, zend_llist_position *pos);

#define zend_llist_get_first(l) zend_llist_get_first_ex(l, NULL)
#define zend_llist_get_last(l) zend_llist_get_last_ex(l, NULL)
#define zend_llist_get_next(l) zend_llist_get_next_ex(l, NULL)
#define zend_llist_get_prev(l) zend_llist_get_prev_ex(l, NULL)
```

一般情况下我们遍历只需要使用后面的那组宏定义函数即可，如果不想要改变链表内部指针，可以主动传递当前指针所指向的位置。

PHP中很多的函数都会有`*_ex()`以及不带ex两个版本的函数，这主要是为了方便使用，和上面的代码一样，ex版本的通常是一个功能较全或者可选参数较多的版本，而在代码中很多地方默认的参数值都一样，为了方便使用，再封装一个普通版本。

这里之所以使用宏而不是定义另一个函数是为了避免函数调用带来的消耗，不过有的情况下还要进行其他的操作，也是会再定义一个新的函数的。

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';
}
//$var = ^_^;    //语法错误
$var = constant("^_^");
```

通过defined函数测试表示，^_^这个常量已经定义好，这样的常量无法直接调用，只能使用constant()方法来获取到，否则在语法解析时会报错，因为它不是一个合法的标示符。

除了CONST_CS标记，常量的flags字段通常还可以用CONST_PERSISTENT和CONST_CT_SUBST。

CONST_PERSISTENT表示这个常量需要持久化。这里的持久化内存申请时的持久化是一个概念，非持久常量会在请求结束时释放该常量，如果读者还不清楚PHP的生命周期，可以参考，PHP生命周期这一小节，也就是说，如果是非持久常量，会在RSHUTDOWN阶段就该常量释放，否则只会在MSHUTDOWN阶段 将内存释放，在用户空间，也就是用户定义的常量都是非持久化的，通常扩展和内核定义的常量 会设置为持久化，因为如果常量被释放了，而下次请求又需要使用这个常量，该常量就必须 在请求时初始化一次，而对于常量这些不变的量来说就是个没有意义的重复计算。

在PHP，只有标量才能被定义为常量，而在内核C代码中，一些字符串，数字等作为代码的一部分，并且他们被定义成PHP内核中的常量。这些常量属于静态对象，被给定了一个绝对地址，当释放这些常量时，我们并不需要将这些静态的内存释放掉，从而也就有了我们这里的CONST_PERSISTENT标记。

CONST_CT_SUBST我们看注释可以知道其表示Allow compile-time substitution（在编译时可被替换）。在PHP内核中这些常量包括：TRUE、FALSE、NULL、ZEND_THREAD_SAFE和ZEND_DEBUG_BUILD五个。

标准常量的初始化

通过define()函数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常量：如错误报告级别E_ALL, E_WARNING等常量就有点不同了。这些是PHP内置定义的常量，他们属于标准常量。

在Zend引擎启动后，会执行如下的标准常量注册操作。**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()是一个宏，用于注册一个长整形数字的常量，因为C是强类型 语言，不同类型的数据等分别处理，以上的宏展开到下面这个函数。

```
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);
}
```

代码很容易理解，前面看到注册内置常量都是用了CONST_PERSISTENT标志位，也就是说，这些常量都是持久化常量。

魔术常量

PHP提供了大量的预定义常量，有一些是内置的，也有一些是扩展提供的，只有在加载了这些扩展库时才会出现。

不过PHP中有七个魔术常量，他们的值其实是变化的，它们的值随着它们在代码中的位置改变而改变。 所以称他们为魔术常量。例如 __LINE__ 的值就依赖于它在脚本中所处的行来决定。 这些特殊的常量不区分大小写。在手册中这几个变量的简单说明如下：

几个 PHP 的“魔术常量”

```
名称                          | 说明
------------------------------ | --------------------------------
\_\_LINE\_\_                    | 文件中的当前行号
\_\_FILE\_\_                    | 文件的完整路径和文件名。如果用在被包含文件中，则返回被包含的文件名。自 PHP 4.0.2 起，\_\_FILE\_\_ 总是包含一个绝对路径（如果是符号连接，则是解析后的绝对路径），而在此之前的版本有时会包含一个相对路径。
\_\_DIR\_\_                     | 文件所在的目录。如果用在被包括文件中，则返回被包括的文件所在的目录。它等价于 dirname(\_\_FILE\_\_)。除非是根目录，否则     目录中名不包括末尾的斜杠。（PHP 5.3.0中新增）
\_\_FUNCTION\_\_                | 函数名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该函数被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写>     字母的
\_\_CLASS\_\_                   | 类的名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该类被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的
\_\_METHOD\_\_                  | 类的方法名（PHP 5.0.0 新加）。返回该方法被定义时的名字（区分大小写）。
\_\_NAMESPACE\_\_               | 当前命名空间的名称（大小写敏感）。这个常量是在编译
```

时定义的（PHP 5.3.0 新增）

PHP中的一些比较*魔术*的变量或者标示都习惯使用下划线来进行区分， 所以在编写PHP代码时也尽量不要定义双下线开头的常量。

PHP内核会在词法解析时将这些常量的内容赋值进行替换， 而不是在运行时进行分析。 如下PHP代码：

```
<?PHP
echo __LINE__;
function demo() {
    echo __FUNCTION__;
}
demo();
```

PHP已经在词法解析时将这些常量换成了对应的值， 以上的代码可以看成如下的PHP代码：

```
<?PHP
echo 2;
function demo() {
    echo "demo";
}
demo();
```

如果我们使用VLD扩展查看以上的两段代码生成的中间代码， 你会发现其结果是一样的。

前面我们有说PHP是在词法分析时做的赋值替换操作， 以__FUNCTION__为例， 在Zend/zend_language_scanner.l文件中， __FUNCTION__是一个需要分析垢元标记（token）：

```
<ST_IN_SCRIPTING>"__FUNCTION__" {
    char *func_name = NULL;

    if (CG(active_op_array)) {
        func_name = CG(active_op_array)->function_name;
    }

    if (!func_name) {
        func_name = "";
    }
    zendlval->value.str.len = strlen(func_name);
    zendlval->value.str.val = estrndup(func_name, zendlval->value.str.len);
    zendlval->type = IS_STRING;
    return T_FUNC_C;
}
```

就是这里， 当当前中间代码处于一个函数中时， 则将当前函数名赋值给zendlval(也就是token T_FUNC_C的值内容)， 如果没有， 则将空字符串赋值给zendlval(因此在顶级作用域名中直接打印__FUNCTION__会输出空格)。 这个值在语法解析时会直接赋值给返回值。这样我们就在生成的中间代码中看到了这些常量的位置都已经赋值好了。

和__FUNCTION__类似， 在其附近的位置， 上面表格中的其它常量也进行了类似的操作。

前面有个比较特殊的地方， 当func_name不存在时， __FUNCTION__被替换成空字符串， 你可能会想， 怎么会有变量名不存在的方法呢， 这里并不是匿名方法， 匿名方法

的function_name 并不是空的， 而是:"{closure}", 有兴趣的读者可以去代码找找在那里给定义了。
这里涉及PHP字节码的编译， 在PHP中， 一个函数或者一个方法会变编译成一个opcode array opcode array的function name字段标示的就是这个函数或方法的名称， 同时一段普通的代码也会被当成一个完整实体被编译成一段opcode array， 只不过没有函数名称。
在PHP5.4中增加了对于trait类的常量定义：__TRAIT__。

这些常量其实相当于一个常量模板， 或者说是一个占位符， 在词法解析时这些模板或占位符就被替换成实际的值

第三节 预定义变量

在PHP脚本执行的时候， 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， 而我们用得非常多的在全局范围内有效的变量却与这些用户全局变量不同。例如:$_GET， $_POST， $_SERVER， $_FILES等变量， 我们并没有在程序中定义这些变量， 并且这些变量也同样保存在符号表中， 从这些表象我们不难得出结论：PHP是在脚本运行之前就将这些特殊的变量加入到了符号表。

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
```

php_request_startup函数在PHP的生命周期中属于请求初始化阶段， 即每个请求都会执行这个函数。因此， 对于每个用户请求， 其用到的这些预定义的全局变量都会不同。 $GLOVALS的关键点在于zend_hash_update函数的调用， 它将变量名为GLOBALS的变量注册到EG(symbol_table)中，EG(symbol_table)是一个HashTable的结构， 用来存放顶层作用域的变量。 通过这个操作， GLOBAL变量与其它顶层的变量一样都会注册到了变量表， 也可以和其它变量一样直接访问了。这在下面将要提到的$_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。

以文件上传中获取文件的信息为例（假设在Apache服务器环境下）：我们首先创建一个静态页面test.html，其内容如下所示：

```
<form name="upload" action="upload_test.php" method="POST"
enctype="multipart/form-data">
    <input type="hidden" value="1024" name="MAX_FILE_SIZE" />
    请选择文件:<input name="ufile" type="file" />
    <input type="submit" value="提 交" />
</form>
```

当我们在页面中选择点击提交按钮时，浏览器会将数据提交给服务器。通过Filddle我们可以看到其提

交的请求头如下：

```
POST http://localhost/test/upload_test.php HTTP/1.1
Host: localhost
Connection: keep-alive
Content-Length: 1347
Cache-Control: max-age=0
Origin: http://localhost
User-Agent: //省略若干
Content-Type: multipart/form-data; boundary=----
WebKitFormBoundaryBq7AMhcljN14rJrU

// 上面的是关键
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost/test/test.html
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3

// 以下为POST提交的内容

------WebKitFormBoundaryBq7AMhcljN14rJrU
Content-Disposition: form-data; name="MAX_FILE_SIZE"

10240
------WebKitFormBoundaryBq7AMhcljN14rJrU
Content-Disposition: form-data; name="ufile"; filename="logo.png"
Content-Type: image/png //这里就是我们想要的文件类型

//以下为文件内容
```

如果我们在upload_test.php文件中打印$_FILES，可以看到上传文件类型为image/png。对应上面的请求头，image/png在文件内容输出的前面的Content-Type字段中。基本上我们知道了上传的文件类型是浏览器自己识别，直接以文件的Content-Type字段传递给服务器。如果有多个文件上传，就会有多个boundary分隔文件内容，形成多个POST内容块。那么这些内容在PHP中是如何解析的呢？

当客户端发起文件提交请求时，Apache会将所接收到的内容转交给mod_php5模块。当PHP接收到请求后，首先会调用sapi_activate，在此函数中程序会根据请求的方法处理数据，如示例中POST方法，其调用过程如下：

```
if(!strcmp(SG(request_info).request_method, "POST")
&& (SG(request_info).content_type)) {
    /* HTTP POST -> may contain form data to be read into variables
    depending on content type given
    */
    sapi_read_post_data(TSRMLS_C);
}
```

sapi_read_post_data在main/SAPI.c中实现，它会根据POST内容的Content-Type类型来选择处理POST内容的方法。

```
if (zend_hash_find(&SG(known_post_content_types), content_type,
content_type_length+1, (void **) &post_entry) == SUCCESS) {
    /* found one, register it for use */
    SG(request_info).post_entry = post_entry;
    post_reader_func = post_entry->post_reader;
}
```

以上代码的关键在于SG(known_post_content_types)变量， 此变更是在SAPI启动时初始化全局变量时被一起初始化的， 其基本过程如下：

```
sapi_startup
sapi_globals_ctor(&sapi_globals);
php_setup_sapi_content_types(TSRMLS_C);
sapi_register_post_entries(php_post_entries TSRMLS_CC);
```

这里的的php_post_entries定义在main/php_content_types.c文件。如下：

```
/* {{{ php_post_entries[]
*/
static sapi_post_entry php_post_entries[] = {
{ DEFAULT_POST_CONTENT_TYPE, sizeof(DEFAULT_POST_CONTENT_TYPE)-1,
sapi_read_standard_form_data, php_std_post_handler },
{ MULTIPART_CONTENT_TYPE, sizeof(MULTIPART_CONTENT_TYPE)-1, NULL,
rfc1867_post_handler },
{ NULL, 0, NULL, NULL }
};
/* }}} */

#define MULTIPART_CONTENT_TYPE "multipart/form-data"

#define DEFAULT_POST_CONTENT_TYPE "application/x-www-form-urlencoded"
```

如上所示的MULTIPART_CONTENT_TYPE（multipart/form-data）所对应的rfc1867_post_handler方法就是处理$_FILES的核心函数， 其定义在main/rfc1867.c文件：SAPI_API SAPI_POST_HANDLER_FUNC(rfc1867_post_handler) 后面获取Content-Type的过程就比较简单了：

- 通过multipart_buffer_eof控制循环， 遍历所有的multipart部分
- 通过multipart_buffer_headers获取multipart部分的头部信息
- 通过php_mime_get_hdr_value(header, “Content-Type”)获取类型
- 通过register_http_post_files_variable(lbuf, cd, http_post_files, 0 TSRMLS_CC); 将数据写到$_FILES变量。

main/rfc1867.c

```
SAPI_API SAPI_POST_HANDLER_FUNC(rfc1867_post_handler)
{

//若干省略
    while (!multipart_buffer_eof(mbuff TSRMLS_CC)){
        if (!multipart_buffer_headers(mbuff, &header TSRMLS_CC)) {
        goto fileupload_done;
    }
//若干省略
    /* Possible Content-Type: */
    if (cancel_upload || !(cd = php_mime_get_hdr_value(header, "Content-
Type"))) {
        cd = "";
    } else {
    /* fix for Opera 6.01 */
        s = strchr(cd, ';');
        if (s != NULL) {
```

```
                *s = '\0';
            }
        }
//若干省略
        /* Add $foo[type] */
        if (is_arr_upload) {
                snprintf(lbuf, llen, "%s[type][%s]", abuf, array_index);
        } else {
            snprintf(lbuf, llen, "%s[type]", param);
        }
        register_http_post_files_variable(lbuf, cd, http_post_files, 0 TSRMLS_CC);
        //若干省略
        }
}
```

其它的$_FILES中的size、name等字段，其实现过程与type类似。

预定义变量的获取

在某个局部函数中使用类似于$GLOBALS变量这样的预定义变量，如果在此函数中有改变的它们的值的话，这些变量在其它局部函数调用时会发现也会同步变化。为什么呢？是否是这些变量存放在一个集中存储的地方？从PHP中间代码的执行来看，这些变量是存储在一个集中的地方：EG(symbol_table)。

在模块初始化时，$GLOBALS在zend_startup函数中通过调用zend_register_auto_global将GLOBALS注册为预定义变量。$_GET、$_POST等在php_startup_auto_globals函数中通过zend_register_auto_global将_GET、_POST等注册为预定义变量。

在通过$获取变量时，PHP内核都会通过这些变量名区分是否为全局变量（ZEND_FETCH_GLOBAL），其调用的判断函数为zend_is_auto_global，这个过程是在生成中间代码过程中实现的。如果是ZEND_FETCH_GLOBAL或ZEND_FETCH_GLOBAL_LOCK(global语句后的效果)，则在获取获取变量表时(zend_get_target_symbol_table)，直接返回EG(symbol_table)。则这些变量的所有操作都会在全局变量表进行。

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和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，因为除非明确unset释放，在程序运行过程中始终存在。
- 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。

最常见的是静态局部变量及静态成员变量。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。如果某个局部变量定义为静态的， 则它的值不会在函数调用结束后释放， 而是继续保留变量的值。

在本小节将介绍静态局部变量， 有关静态成员变量的内容将在类与对象章节进行介绍。

先看看如下局部变量的使用：

```
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变量的值在改变后函数继续执行还能访问到， $i变量就像是只有函数t()才能访问到的一个全局变量。 那PHP是怎么实现的呢？

static是PHP的关键字， 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); }

;
```

语法分析的过程中如果匹配到相应的模式则会进行相应的处理动作， 通常是进行opcode的编译。 在本例中的static关键字匹配中， 是由函数zend_do_fetch_static_variable处理的。

3. 生成opcode中间代码

zend_do_fetch_static_variable函数的作用就是生成opcode，定义如下：

```
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

opcode的编译阶段完成后就开始opcode的执行了。在Zend/zend_vm_opcodes.h文件中包含所有opcode的宏定义，这些宏丙没有特殊含义，只是作为opcode的唯一标示，包含本例中相关的如下两个宏的定义：

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

前面第二章 [脚本的执行一节][from-op-code-to-handler]介绍了根据opcode查找到相应处理函数的方法。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 {
        // 取符号表，这里我们取的是EG(active_op_array)->static_variables
        target_symbol_table = zend_get_target_symbol_table(opline, EX(Ts),
type, varname TSRMLS_CC);
        ...//   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。它的作用是获取当前正在执行的目标符号表， 而在函数执行时当前的op_array则是函数体本身，先看看zend_op_array的结构。

```
struct _zend_op_array {
    /* Common elements */
    zend_uchar type;
    char *function_name;
    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;
    zend_uint last, size;

    /* static variables support */
    HashTable *static_variables;

    zend_op *start_op;
    int backpatch_count;

    zend_uint this_var;
    // ...
}
```

由上可以看到zend_op_array中包含function_name字段，也就是当前函数的名称。 再看看获取当前符号标的函数：

```
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函数中我们返回的是EG(active_op_array)->static_variables。也就是当前函数的的静态变量哈希表。每次执行时都会从该符号表中查找相应的值，由于op_array在程序执行时始终存在。所有对静态符号表中数值的修改会继续保留，下次函数执行时继续从该符号表获取信息。也就是说Zend为每个函数(准确的说是zend_op_array)分配了一个私有的符号表来保存该函数的静态变量。

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也并不严格检查数据类型。不过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。

要使用类型提示，只要在方法（或函数）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。

一个数组的类型提示示例：

```
function array_print(Array $arr) {
    print_r($arr);
}

array_print(1);
```

以上的这段代码有一点问题，它触发了我们这次所介绍的类型提示，这段代码在PHP5.1之后的版本执行，会报错如下：

```
Catchable fatal error: Argument 1 passed to array_print() must be an array,
integer given, called in  ...
```

当我们把函数参数中的整形变量变为数组时，程序会正常运行，调用print_r函数输出数组。那么这个类型提示是如何实现的呢？不管是在类中的方法，还是我们调用的函数，都是使用function关键字作为其

声明的标记， 而类型提示的实现是与函数的声明相关的， 在声明时就已经确定了参数的类型是哪些， 但是需要在调用时才会显示出来。 这里， 我们从两个方面说明类型提示的实现：

1. 参数声明时的类型提示
2. 函数或方法调用时的类型提示

将刚才的那个例子修改一下：

```
function array_print(Array $arr = 1) {
    print_r($arr);
}

array_print(array(1));
```

这段代码与前面的那个示例相比， 函数的参数设置了一个默认值， 但是这个默认值是一个整形变量， 它与参数给定的类型提示Array不一样， 因此， 当我们运行这段代码时会很快看到程序会报错如下：

```
Fatal error: Default value for parameters with array type hint
can only be an array or NULL
```

为什么为很快看到报错呢？ 因为默认值的检测过程发生在成中间代码生成阶段， 与运行时的报错不同， 它还没有生成中间代码， 也没有执行中间代码的过程。 在Zend/zend_language_parser.y文件中， 我们找到函数的参数列表在编译时都会调用zend_do_receive_arg函数。 而在这个函数的参数列表中， 第5个参数（ znode *class_type）与我们这节所要表述的类型提示密切相关。 这个参数的作用是声明类型提示中的类型， 这里的类型有三种：

1. 空， 即没有类型提示
2. 类名， 用户定义或PHP自定义的类、接口等
3. 数组， 编译期间对应的token是T_ARRAY， 即Array字符串

在zend_do_receive_arg函数中， 针对class_type参数做了一系列的操作， 基本上是针对上面列出的三种类型， 其中对于类名， 程序并没有判断这个类是否存在， 即使你使用了一个不存在的类名， 程序在报错时， 显示的也会是实参所给的对象并不是给定类的实例。

以上是声明类型提示的过程以及在声明过程中对参数默认值的判断过程， 下面我们看下在函数或方法调用时类型提示的实现。

从上面的声明过程我们知道PHP在编译类型提示的相关代码时调用的是Zend/zend_complie.c文件中的zend_do_receive_arg函数， 在这个函数中将类型提示的判断的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) { //  数组
        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：

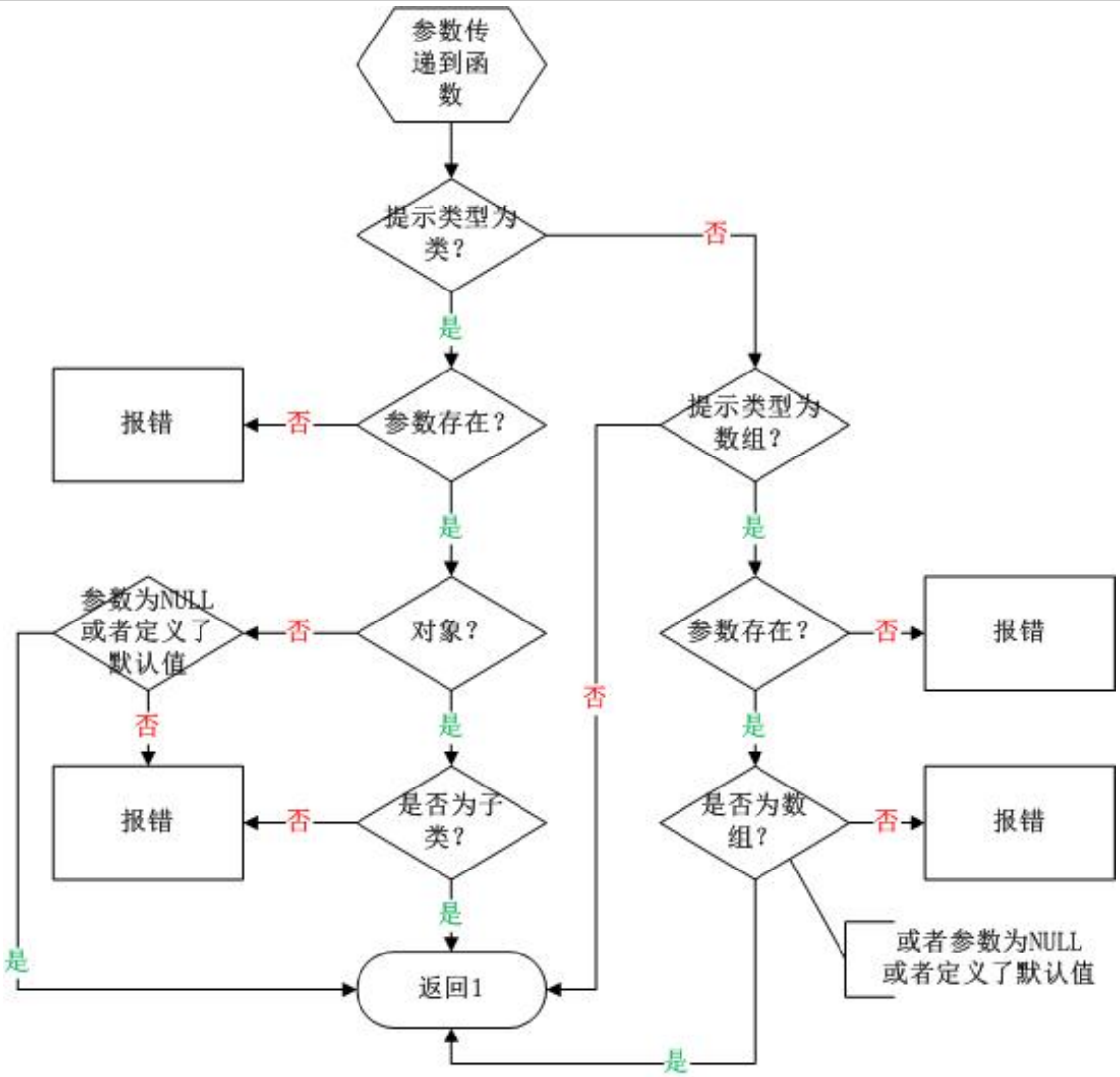


图3.1 类型提示判断流程图

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;
}
```

在上面的代码中，我们可以找到前面的报错信息中的一些关键字Argument、 passed to、called in等。 这就是我们在调用函数或方法时类型提示显示错误信息的最终执行位置。

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， 直接对其赋值就可以使用， 这是如何实现的？

变量的声明和赋值

在PHP中没有对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。在赋值操作的同时已经进行声明操作。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**。（找到这个函数的方法之一：$a为CV，10为CONST，操作为ASSIGN。其他方法可以参见附：找到Opcode具体实现） 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。
参见Zend/zend_complie.c文件中的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。

```
static zend_always_inline zval **_get_zval_ptr_ptr_cv(const znode *node, const
temp_variable *Ts, int type TSRMLS_DC)
{
    zval ***ptr = &CV_OF(node->u.var);

    if (UNEXPECTED(*ptr == NULL)) {
        return _get_zval_cv_lookup(ptr, node->u.var, type TSRMLS_CC);
    }
    return *ptr;
}

// 函数中的CV_OF宏定义
#define CV_OF(i)      (EG(current_execute_data)->CVs[i])
```

_get_zval_ptr_ptr_cv函数程序会先判断变量是否存在于EX(CVs)，如果存在则直接返回，否则调用_get_zval_cv_lookup，通过HastTable操作在EG(active_symbol_table)表中查找变量。虽然HashTable的查找操作已经比较快了，但是与原始的数组操作相比还是不在一个数量级。这就是CV类型变量的性能优化点所在。CV以数组的方式缓存变量所在HashTable的值，以取得对变量更快的访问速度。

如果变量不在EX(CVs)中，程序会调用_get_zval_cv_lookup。从而最后的调用顺序为：[_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

这种情形描述起来比较抽象，如下面的示例：

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

xdebug_debug_zval('a');

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

试想，如果我们来做这个**$b = &$a;**的底层实现，我们可能会这样做：

- 判断左值是不是已经被引用过了;
- 左值已经被引用，则不改变左值的引用计数，将右值赋与左值;

事实上，ZE也是用同样的方法来实现，其代码如下：

```
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：

上面的例子的输出结果：

```
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，应该是什么都不会发生吗？看一个示例：

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

看上去真的像是什么都没有发生，左值的引用计数还是1，值仍是10 。然而在这个赋值过程中，$a的引用计数经历了一次加一和一次减一的操作。如以下代码：

```
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

用一个PHP的示例来描述一下这种情况：

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

这里的**$c = $a;**的操作就是我们所示的第三种情况。对于这种情况，ZEND内核直接创建一个新的zval容器，左值的值为右值，并且左值的引用计数为1。也就是说，这种情形$c不会与$a指向同一个zval。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;
```

在这个例子中，若将 **$c = $a;** 换成 **$c = &$a;**，$a，$b和$c三个变量的引用计数会发生什么变化？

将 **$b = &$a**; 换成 **$b = $a;** 呢？

大家可以将答案回复在下面：）

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

这种情形如下面的例子：

```
$a = 10;
```

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

这时，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，左值的引用计数为大于0，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;
$b = $a;
$va = 20;
$vb = &$va;

$a = $va;
```

最后一个操作就是我们的情况五。使用xdebug看引用计数发现，最终$a变量的引用计数为1，$va变量的引用计数为2，并且$va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，印证了我们之前的例子$a变量的结果。

除上述五种情况之外，**zend_assign_to_variable**函数还对全部的临时变量做了处理。变量赋值的各种操作全部由此函数完成。

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。

一个简洁的例子：

```
$a = 10;
unset($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) { //  删除
HashTable元素
        zend_execute_data *ex = execute_data;

        do {
            int i;

            if (ex->op_array) {
                for (i = 0; i < ex->op_array->last_var; i++) {
                    if (ex->op_array->vars[i].hash_value == hash_value &&
                        ex->op_array->vars[i].name_len == varname-
>value.str.len &&
                        !memcmp(ex->op_array->vars[i].name, varname-
>value.str.val, varname->value.str.len)) {
                        ex->CVs[i] = NULL; // 置空EX(CVs)
                        break;
                    }
                }
            }
            ex = ex->prev_execute_data;
        } while (ex && ex->symbol_table == target_symbol_table);
    }
```

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。如果对HashTable的元素删除操作成功，程序还会对EX(CVs)内存储的值进行清空操作。以缓存机制来解释，在删除原始数据后，程序也会删除相对应的缓存内容，以免用户获取到赃数据。

> 变量的销毁还涉及到垃圾回收机制（GC），请参见相关第六章内容 关于HashTable的操作请参考 [<< 哈希表(HashTable) >>]。

变量的作用域

变量的作用域是变量的一个作用范围，在这个范围内变量为可见的，即可以访问该变量的代码区域，相反，如果不在这个范围内，变量是不可见的，无法被调用。（全局变量可以将作用范围看作为整个程序）如下面的例子：（会输出什么样的结果呢？）

```php
<?php
    $foo = 'tipi';
    function variable_scope(){
        $foo = 'foo';
        print $foo ;
        print $bar ;
    }
```

由此可见，变量的作用域是一个很基础的概念，在变量的实现中比较重要。

全局变量与局部变量

变量按作用域类型分为：全局变量和局部变量。**全局变量**是在整个程序中任何地方随意调用的变量，在PHP中，全局变量的“全局化”使用gloal语句来实现。相对于全局变量，**局部变量**的作用域是程序中的部分代码（如函数中），而不是程序的全部。

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。

> 变量的作用域决定其生命周期吗？程序运行到变量作用域范围之外，就会将变量进行销毁吗？
> 如果你知道答案，可以回复在下面。

对于不同作用域的变量，如果存在冲突情况，就像上面的例子中，全局变量中有一个名为$bar的变量，在局部变量中也存在一个名为$bar的变量，此时如何区分呢？

对于全局变量，Zend引擎有一个_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，如果程序运行于某个函数中，Zend引擎会遍历查询与其对应的active_symbol_table，而每个函数的active_symbol_table是相对独立的，由此而实现的作用域的独立。

展开来看，如果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。因为每个函数调用开始时都会重新初始化EG(active_symbol_table)为NULL，在这个函数的所有opcode的执行过程中这个全局变量会一直存在，并且所有的局部变量修改都是在它上面操作完成的，如前面的赋值操作等。而此时，其他函数中的symbol_table会存放在栈中，将当前函数执行完并返回时，程序会将之前保存的zend_execute_data恢复，从而其他函数中的变量也就不会被找到，局部变量的作用域就是以这种方式来实现的。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;
```

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global语句来将变量“挪”到函数独立的*active_symbol_table中，即变量的跨域操作。（关于global的详细解释，见下一小节）

> 在PHP的源码中，EX宏经常出现，它的作用是获取结构体zend_execute_data的字段值，它的实现是：
> #define EX(element) execute_data->element

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是弱类型的动态语言，在前面的章节中我们已经介绍了PHP的变量都存放在一个名为ZVAL的容器中， ZVAL包含了变量的类型和各种类型变量的值。 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变量之间的数据类型转换有两种: 隐式和显式转换。

隐式类型转换

隐式类型转换也被称为自动类型转换，是指不需要程序员书写代码，由编程语言自动完成的类型转换。 在PHP中，我们经常遇到的隐式转换有：

1. **直接的变量赋值操作**

在PHP中，直接对变量的赋值操作是隐式类型转换最简单的方式，也是我们最常见的一种方式，或许我们已经习以为常，从而没有感觉到变量的变化。 在直接赋值的操作中，变量的数据类型由赋予的值决定，即左值的数据类型由右值的数据类型决定。 比如，当把一个字符串类型的数据赋值给变量时，不管该变量以前是什么类型的变量，此时该变量就是一个字符串类型的变量。 看一段代码：

```
$string = "To love someone sincerely means to love all the people,  to love the world and life,  too."
$integer = 10;
$string = $integer;
```

上面的代码，当执行完第三行代码，$string变量的类型就是一个整形了。 通过VLD扩展可以查到第三次赋值操作的中间代码及操作数的类型，再找到赋值的最后实现为**zend_assign_to_variable**函数。 这在前面的小节中已经详细介绍过了。我们这个例子是很简单的一种赋值，在源码中是直接将$string的ZVAL容器的指针指向$integer变量指向的指针， 并将$integer的引用计数加1。这个操作在本质上改变了$string变量的内容，而原有的变量内容则被垃圾收集机制回收。关于赋值的具体细节，请返回上一节查看。

2. **运算式结果对变量的赋值操作** 我们常说的隐式类型转换是将一个表达式的结果赋值给一个变量，在运算的过程中发生了隐式的类型转换。 这种类型转换不仅仅在PHP语言，在其它众多的语言中也有见到，这是我们常规意义上的隐式类型转换。 这种类型转换又分为两种情况：

- 表达式的操作数为同一数据类型 这种情况的作用以上面的直接变量的类型转换是同一种情况，只是此时右值变成了表达式的运算结果。
- 表达式的操作数不为同的数据类型 这种情况的类型转换发生在表达式的运算符的计算过程中，在源码中也就是发生在运行符的实现过程中。

看一个字符串和整数的隐式数据类型转换:

```
<?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说明隐式转换的实现:

脚本执行的时候字符串的连接操作是通过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:
            // ...省略
        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:
                // ... 省略
        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。在表达式计算完成后，表达式最后会有一个结果， 这个结果的数据类型就是整个表达式的数据类型。当执行赋值操作时，如果再有数据类型的转换发生， 则是直接变量赋值的数据类型转换了。

显式类型转换(强制类型转换)

在前面介绍了隐式类型转换，在我们的日常编码过程也会小心的使用这种转换， 这种不可见的操作可能与我们想象中的不一样，如整形和浮点数之间的转换。 当我们是一定需要某个数据类型的变量时，可以使用强制的数据类型转换，这样在代码的可读性等方面都会好些。 在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一个我们比较少见的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，这只是一个类型的变化，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。

除了上面提到的与C语言很像，在其它语言中也经常见到的强制数据转换，PHP中有一个极具PHP特色的强制类型转换。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。在ext/standard/type.c文件中找到settype的实现源码：

```
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;
}
```

这个极具PHP特色的强制类型转换就是这个函数，而这个函数是作为一个代理方法存在，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，这和前面的强制类型转换在本质上是一样的。

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，还包括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 代码都可以编写在函数内部，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，它们都是由标准扩展提供的；如我们经常用到的isset、empty、eval等函数，这些结构被称之为语言结构。还有一些函数需要和特定的PHP扩展模块一起编译并开启，否则无法使用。也就是有些扩展是可选的。

> 标准函数的实现存放在ext/standard扩展目录中。

匿名函数

有时我们的一代代码并不需要为它指定一个名称，而只需要它完成特定的工作，匿名函数的作用是为了扩大函数的使用功能，在PHP 5.3以前，传递函数回调的方式，我们只有两种选择：

- 字符串的函数名
- 使用create_function创建的返回

在PHP5.3以后，我们多了一个选择--Closure。在实现上PHP 5.3中对匿名函数的支持，采用的是把要保持的外部变量，做为Closure对象的”Static属性”来实现的，关于如何实现我们将在后面的章节介绍。

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（echo等）

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

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。也就是说PHP将内置的函数和 用户定义的函数分别保存。

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， 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， 并不一定显式的有返回值， 在PHP的实现中， 即使没有显式的返回， PHP内核也会“帮你“返回NULL。

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;    /* 如用户自定义则为 #define ZEND_USER_FUNCTION 2
                           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;      // 作为方法时的访问类型等，如ZEND_ACC_STATIC等
        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扩展中查看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>>

函数间的转换

在函数调用的执行代码中我们会看到这样一些强制转换：

```
EX(function_state).function = (zend_function *) op_array;
```

或者：

```
EG(active_op_array) = (zend_op_array *) EX(function_state).function;
```

这些不同结构间的强制转换是如何进行的呢？

首先我们来看zend_function的结构，在Zend/zend_compile.h文件中，其定义如下：

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

    struct {
        zend_uchar type;  /* never used */
        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;
    } common;

    zend_op_array op_array;
    zend_internal_function internal_function;
} zend_function;
```

这是一个联合体，我们来温习一下联合体的一些特性。联合体的所有成员变量共享内存中的一块内存，在某个时刻只能有一个成员使用这块内存，并且当使用某一个成员时，其仅能按照它的类型和内存大小修改对应的内存空间。我们来看看一个例子：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    typedef  union _utype
    {
        int i;
        char ch[2];
    } utype;

    utype a;

    a.i = 10;
    a.ch[0] = '1';
    a.ch[1] = '1';

    printf("a.i= %d a.ch=%s",a.i, a.ch);
    getchar();

    return (EXIT_SUCCESS);
}
```

程序输出：a.i= 12593 a.ch=11 当修改ch的值时，它会依据自己的规则覆盖i字段对应的内存空间。

'1'对应的ASCII码值是49，二进制为00110001，当ch字段的两个元素都为'1'时，此时内存中存储的二进制为 00110001 00110001 转成十进制，其值为12593。

回过头来看zend_function的结构，它也是一个联合体，第一个字段为type，在common中第一个字段也为type，并且其后面注释为/* Never used*/，此处的type字段的作用就是为第一个字段的type留下内存空间。并且不让其它字段干扰了第一个字段。我们再看zend_op_array的结构：

```
struct _zend_op_array {
    /* 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 */

    zend_bool done_pass_two;
    ....//  其它字段
}
```

这里的字段集和common的一样，于是在将zend_function转化成zend_op_array时并不会产生影响，这种转变是双向的。

再看zend_internal_function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;
```

同样存在公共元素，和common结构体一样，我们可以将zend_function结构强制转化成zend_internal_function结构，并且这种转变是双向的。

总的来说zend_internal_function，zend_function，zend_op_array这三种结构在一定程序上存在公共的元素，于是这些元素以联合体的形式共享内存，并且在执行过程中对于一个函数，这三种结构对应的字段在值上都是一样的，于是可以在一些结构间发生完美的强制类型转换。可以转换的列表如下：

- zend_function可以与zend_op_array互换
- zend_function可以与zend_internal_function互换

但是一个zend_op_array结构转换成zend_function是不能再次转变成zend_internal_function结构的，反之亦然。

其实zend_function就是一个混合的数据结构，这种结构在一定程序上节省了内存空间。

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、内部函数的参数和参数的传递：

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。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
- 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：

- | - 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。

参数的传递

在PHP的运行过程中，如果函数有参数，当执行参数传递时，所传递参数的引用计数会发生变化。 如和Xdebug的作者Derick Rethans在其文章 php variables中的示例的类似代码：

```
function do_something($s) {
```

```
        xdebug_debug_zval('s');
        $s = 100;
        return $s;
}

$a = 1111;
$b = do_something($a);
echo $b;
```

如果你安装了xdebug，此时会输出s变量的refcount为3，如果使用debug_zval_dump，会输出4。因为此内部函数调用也对refcount执行了加1操作。这里的三个引用计数分别是：

- function stack中的引用
- function symbol table中引用
- 原变量$a的引用。

这个函数符号表只有用户定义的函数才需要，内置和扩展里的函数不需要此符号表。debug_zval_dump()是内置函数，并不需要符号表，所以只增加了1。xdebug_debug_zval()传递的是变量名字符串，所以没有增加refcount。

每个PHP脚本都有自己专属的全局符号表，而每个用户自定义的函数也有自己的符号表，这个符号表用来存储在这个函数作用域下的属于它自己的变量。当调用每个用户自定义的函数时，都会为这个函数创建一个符号表，当这个函数返回时都会释放这个符号表。

当执行一个拥有参数的用户自定义的函数时，其实它相当于赋值一个操作，即$s = $a; 只是这个赋值操作的引用计数会执行两次，除了给函数自定义的符号表，还有一个是给函数栈。

参数的传递的第一步是SEND_VAR操作，这一步操作是在函数调用这一层级，如示例的PHP代码通过VLD生成的中间代码:

```
compiled vars:  !0 = $a, !1 = $b
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------------
-
-
   2     0  >   EXT_STMT
         1      NOP
   7     2      EXT_STMT
         3      ASSIGN                                                 !0,
1111
   8     4      EXT_STMT
         5      EXT_FCALL_BEGIN
         6      SEND_VAR                                               !0
         7      DO_FCALL                                    1          'demo'
         8      EXT_FCALL_END
         9      ASSIGN                                                 !1, $1
   9    10      EXT_STMT
        11      ECHO                                                   !1
        12    > RETURN                                                 1

branch: #  0; line:     2-    9; sop:     0; eop:    12
path #1: 0,
Function demo:
```

函数调用是DO_FCALL，在此中间代码之前有一个SEND_VAR操作，此操作的作用是将实参传递给函数，并且将它添加到函数栈中。最终调用的具体代码参见zend_send_by_var_helper_SPEC_CV函数，

在此函数中执行了引用计数加1（Z_ADDREF_P）操作和函数栈入栈操作（zend_vm_stack_push）。

与第一步的SEND操作对应，第二步是RECV操作。RECV操作和SEND_VAR操作不同，它是归属于当前函数的操作，仅为此函数服务。它的作用是接收SEND过来的变量，并将它们添加到当前函数的符号表。示例函数生成的中间代码如下：

```
compiled vars:  !0 = $s
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------------
-
-
   2     0  >   EXT_NOP
         1      RECV                                                    1
   3     2      EXT_STMT
         3      ASSIGN                                                  !0, 10
   4     4      EXT_STMT
         5    > RETURN                                                  !0
   5     6*     EXT_STMT
         7*   > RETURN                                                  null

branch: #  0; line:     2-    5; sop:     0; eop:     7
```

参数和普通局部变量一样，都需要进行操作，都需要保存在符号表（或CVs里，不过查找一般都是直接从变量变量数组里查找的）。如果函数只是需要读这个变量，如果我们将这个变量复制一份给当前函数使用的话，在内存使用和性能方面都会有问题，而现在的方案却避免了这个问题，如我们的示例：使用类似于赋值的操作，将原变量的引用计数加一，将有变化时才将原变量引用计数减一，并新建变量。其最终调用是ZEND_RECV_SPEC_HANDLER。

参数的压栈操作用户自定义的函数和内置函数都需要，而RECV操作仅用户自定义函数需要。

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标示符，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

$array = array(1, 2, 3, 4);
array_walk($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

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

面向对象是一种编程范式，它将对象作为程序的基本单元，将程序和数据封装起来，以此来提高程序的重用性、灵活性和可扩展性。

目前很多语言都支持面向对象编程，既然对象对象是一种范式，其实这就和具体的编程语言没有直接关系，只不过很多语言将这个范式作为语言的基本元素，使用C语言也能够进行面向对象编程。

面向对象的程序设计中包含：

1. 类。类是具体事物的抽象。通常类定义了事物的属性和所能完成的工作。有一点需要注意，并不是所有的面向对象编程语言的类都具有class这个明确的实体。例如Javascript就不是基于类的。Javascript中的类(Function)也具有类定义的特性。这也印证了面向对象只是一种编程范式。
2. 对象。对象是类的实例。对象是具体的。
3. 方法。方法是类定义对象可以做的事情。
4. 继承性。继承是类的具体化，子类是比具备更多特性和行为的类。面向对象是对现实世界的一个抽象。在很多时候的关系并不一定是继承关系。能在一定程序上实现代码的重用。
5. 封装性、抽象性。封装性能实现的复杂性隐藏，减少出错的可能。

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。面向过程将一个个功能封装，以一种模块化的思想解决问题。

面向对象听起来很美，但是现实中的编程语言中很少有纯粹的面向对象的语言，处于性能或者程序员的开发习惯，通常的编程语言都同时支持两种变编程方式。

PHP就是如此，从PHP4起开始支持面向对象编程。但PHP4的面向对象支持不太完善。从PHP5起，PHP引入了新的对象模型（Object Model），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中对象是按引用传递的，即对象进行赋值和操作的时候是按引用（reference）传递的，而不是整个对象的拷贝。

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

面向对象编程中我们的编程都是围绕类和对象进行的。那在PHP内部类是怎么实现的呢？它的内存布局以及存储是怎么样的呢？继承、封装和多态又是怎么实现的呢？

类的结构

首先我们看看类是什么。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。

类和函数类似，PHP内置及PHP扩展均可以实现自己的内部类，也可以由用户使用PHP代码进行定义。当然我们在编写代码时通常是自己定义。

使用上，我们使用class关键字进行定义，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，里面是类的实体，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。这些结构在Zend引擎内部是如何实现的？类的方法、成员变量是如何存储的？访问控制，静态成员是如何标记的？

首先，我们看看类的内部存储结构:

```
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;// 类本身所具有的静态变量
    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 l type=2 l fn_flags=258	• function_name=__construct l type=2 l fn_flags=8448 • function_name=iMethod l type=2 l fn_flags=65800 • function_name=_access l type=2 l fn_flags=66560 • function_name=access l type=2 l 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。分别为一下宏的定义，也就是说用户定义的类和模块或者内置的类也是保存在这个结构里的:

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

对于父类和接口，都是保存在**struct _zend_class_entry**结构体中。这表示接口也是以类的形式存储，而实现是一样的，并且在继承等操作时有与类操作的不同的处理。常规的成员方法存放在函数结构体的哈希表中，而魔术方法则单独保存。如在类定义中的 union _zend_function *constructor; 定义就是类的构造魔术方法，它是以函数的形式存在于类结构中，并且与常规的方法分隔开来了。在初始化时，这些魔术方法都会被设置为NULL。

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)。他们分别对应的类型在内核中为:

- 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，类还包含有另外两种类型没有加abstract关键字的抽象类和接口：

- 没有加abstract关键字的抽象类，它对应的type=ZEND_ACC_IMPLICIT_ABSTRACT_CLASS。由于在class前面没有abstract关键字，在语法分析时并没有分析出来这是一个抽象类，但是由于类中拥有抽象方法，在函数注册时判断成员函数是抽象方法或继承类中的成员方法是抽象方法时，会将这个类设置为此种抽象类类型。
- 接口，其type=ZEND_ACC_INTERFACE。接口类型的区分是在interface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。 这个错误会在运行生成中间的代码时触发。 此错误的判断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...

以上的错误程序判断定义在 **zend_do_begin_class_declaration** 函数。 与self关键字一样， 还有parent， static两个关键字的判断在同一个地方。 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， 报我们之前提到的类重复声明错误， 只是这个判断在编译开启时是不会生效的。

类相关的各个结构均保存在**struct _zend_class_entry** 结构体中。这些具体的类别在语法分析过程中进行区分。 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。

第二节 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结构， 接口抽象类等类型的实现方式。 在本小节， 我们将介绍类的成员变量和成员方法。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， 只是这些变量都归属于某个类， 并且给这些变量是有访问控制的。 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， 是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，它可能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，并和成员方法处于同一层次。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;
}
```

类的结构在PHP内核中的存储方式我们已经在上一小节介绍过了。现在，我们要讨论类的成员变量的存储方式。假如我们需要直接访问这个变量，整个访问过程是什么？当然，以这个示例来说，访问这个成员变量是通过对象来访问，关于对象的相关知识我们将在后面的小节作详细的介绍。

当我们用VLD扩展查看以上代码生成的中间代码时，我们发现，并没有相关的中间代码输出。这是因为成员变量在编译时已经注册到了类的结构中，那注册的过程是什么？成员变量注册的位置在哪？

我们从上一小节知道，在编译时类的声明编译会调用zend_do_begin_class_declaration函数。此函数用来初始化类的基本信息，其中包括类的成员变量。其调用顺序为：[zend_do_begin_class_declaration] --> [zend_initialize_class_data] --> [zend_hash_init_ex]

```
zend_hash_init_ex(&ce->default_properties, 0, NULL, zval_ptr_dtor_func,
persistent_hashes, 0);
```

因为类的成员变量是保存在HashTable中，所以，其数据的初始化使用zend_hash_init_ex函数来进行。

在声明类的时候初始化了类的成员变量所在的HashTable，之后如果有新的成员变量声明时，在编译时 **zend_do_declare_property**。函数首先检查成员变量不允许的一些情况：

- 接口中不允许使用成员变量
- 成员变量不能拥有抽象属性
- 不能声明成员变量为final
- 不能重复声明属性

如果在上面的PHP代码中的类定义中，给成员变量前面添加final关键字：

```
class Tipi {
    public final $var;
}
```

运行程序将报错：Fatal error: Cannot declare property Tipi::$var final, the final modifier is allowed only for methods and classes in .. 这个错误由zend_do_declare_property函数抛出：

```
if (access_type & ZEND_ACC_FINAL) {
    zend_error(E_COMPILE_ERROR, "Cannot declare property %s::$%s final, the
final modifier is allowed only for methods and classes",
               CG(active_class_entry)->name, var_name-
>u.constant.value.str.val);
}
```

在定义检查没有问题之后，函数会进行成员变量的初始化操作。

```
ALLOC_ZVAL(property);    //  分配内存

if (value) {     //  成员变量有初始化数据
    *property = value->u.constant;
} else {
    INIT_PZVAL(property);
    Z_TYPE_P(property) = IS_NULL;
}
```

在初始化过程中，程序会先分配内存，如果这个成员变量有初始化的数据，则将数据直接赋值给该属性，否则初始化ZVAL，并将其类型设置为IS_NULL。在初始化过程完成后，程序通过调用 **zend_declare_property_ex** 函数将此成员变量添加到指定的类结构中。

以上为成员变量的初始化和注册成员变量的过程，常规的成员变量最后都会注册到类的 **default_properties** 字段。在我们平时的工作中，可能会用不到上面所说的这些过程，但是我们可能会使用get_class_vars()函数来查看类的成员变量。此函数返回由类的默认属性组成的关联数组，这个数组的元素以 varname => value 的形式存在。其实现核心代码如下：

```
if (zend_lookup_class(class_name, class_name_len, &pce TSRMLS_CC) == FAILURE) {
    RETURN_FALSE;
} else {
    array_init(return_value);
    zend_update_class_constants(*pce TSRMLS_CC);
    add_class_vars(*pce, &(*pce)->default_properties, return_value TSRMLS_CC);
    add_class_vars(*pce, CE_STATIC_MEMBERS(*pce), return_value TSRMLS_CC);
}
```

首先调用zend_lookup_class函数查找名为class_name的类，并将赋值给pce变量。这个查找的过程最核心是一个HashTable的查找函数zend_hash_quick_find，它会查找EG(class_table)。判断类是否存在，如果存在则直接返回。如果不存在，则需要判断是否可以自动加载，如果可以自动加载，则会加载类后再返回。如果不能找到类，则返回FALSE。如果找到了类，则初始化返回的数组，更新类的静态成员变量，添加类的成员变量到返回的数组。这里针对类的静态成员变量有一个更新的过程，关于这个过程我们在下面有关于静态成员变量中做相关介绍。

静态成员变量

类的静态成员变量是所有实例共用的，它归属于这个类，因此它也叫做类变量。在PHP的类结构中，类本身的静态变量存放在类结构的 **default_static_members** 字段中。

与普通成员变量不同，类变量可以直接通过类名调用，这也体现其称作类变量的特别。一个PHP示例：

```
class Tipi {
    public static $var = 10;
}

Tipi::$var;
```

这是一个简单的类，它仅包括一个公有的静态变量$var。通过VLD扩展查看其生成的中间代码：

```
function name:  (null)
number of ops:  6
compiled vars:  !0 = $var
line     # *  op                                fetch          ext  return  operands
---------------------------------------------------------------------------------
-
-
   2     0  >   EXT_STMT
         1      NOP
   6     2      EXT_STMT
         3      ZEND_FETCH_CLASS                                  :1      'Tipi'
         4      FETCH_R                      static member                'var'
         5    > RETURN                                                    1

branch: #  0; line:     2-    6; sop:     0; eop:     5
path #1: 0,
Class Tipi: [no user functions]
```

这段生成的中间代码仅与Tipi::$var;这段调用对应，它与前面的类定义没有多大关系。根据前面的内容和VLD生成的内容，我们可以知道PHP代码：Tipi::$var; 生成的中间代码包括ZEND_FETCH_CLASS和FETCH_R。这里只是一个静态变量的调用，但是它却生成了两个中间代码，什么原因呢？很直白的解释：我们要调用一个类的静态变量，当然要先找到这个类，然后再获取这个类的变量。从PHP源码来看，这是由于在编译时其调用了zend_do_fetch_static_member函数，而在此函数中又调用了zend_do_fetch_class函数，从而会生成ZEND_FETCH_CLASS中间代码。它所对应的执行函数为**ZEND_FETCH_CLASS_SPEC_CONST_HANDLER**。此函数会调用zend_fetch_class函数（Zend/zend_execute_API.c）。而zend_fetch_class函数最终也会调用 **zend_lookup_class_ex** 函数查找类，这与前面的查找方式一样。

找到了类，接着应该就是查找类的静态成员变量，其最终调用的函数为：zend_std_get_static_property。这里由于第二个参数的类型为 ZEND_FETCH_STATIC_MEMBER。这个函数最后是从 **static_members** 字段中查找对应的值返回。而在查找前会和前面一样，执行zend_update_class_constants函数，从而更新此类的所有静态成员变量，其程序流程如图5.1所示：

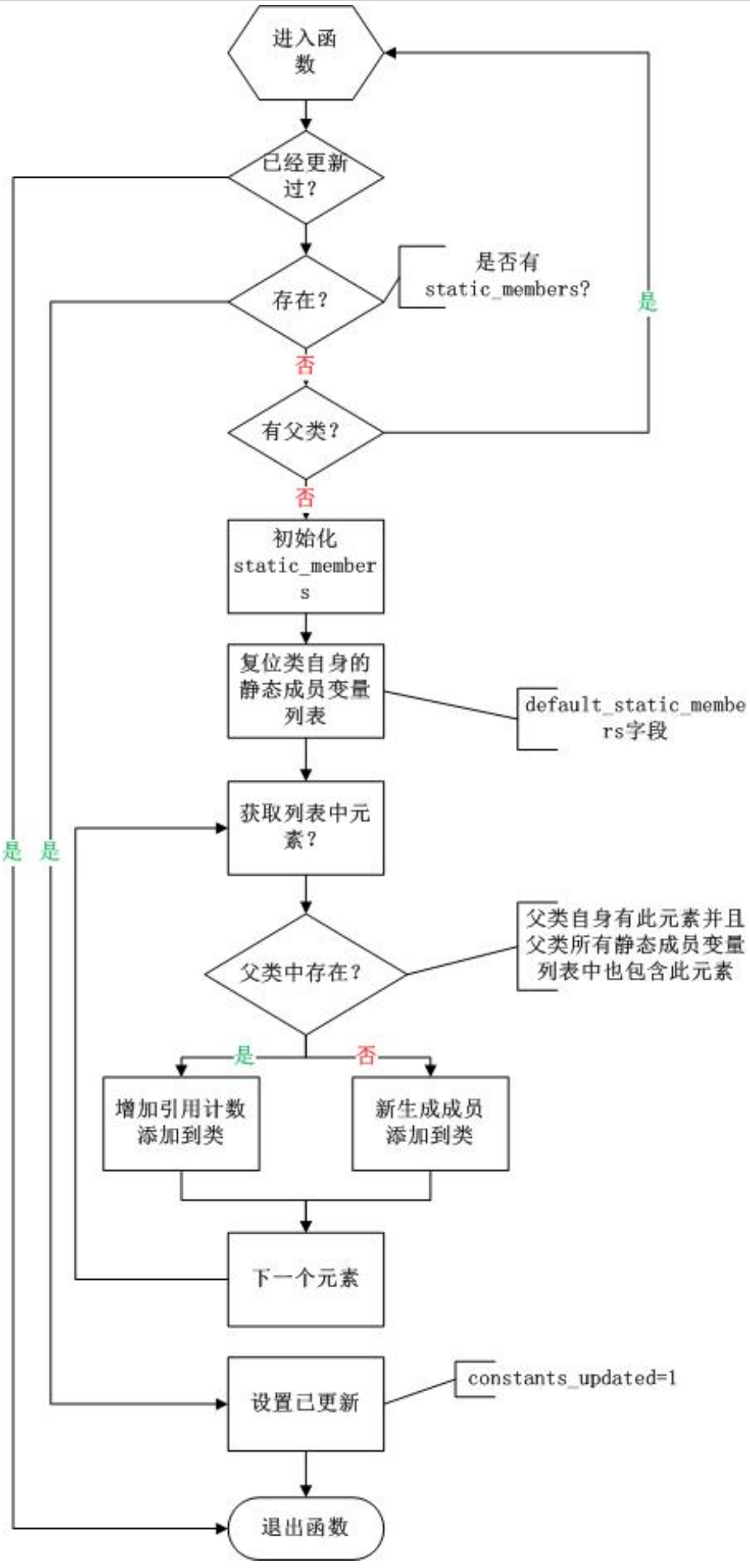


图5.1 静态变量更新流程图

成员方法

成员方法从本质上来讲也是一种函数，所以其存储结构也和常规函数一样，存储在zend_function结构体中。对于一个类的多个成员方法，它是以HashTable的数据结构存储了多个zend_function结构体。和前面的成员变量一样，在类声明时成员方法也通过调用zend_initialize_class_data方法，初始化了整个方法列表所在的HashTable。在类中我们如果要定义一个成员方法，格式如下：

```
class Tipi{
    public function t() {
        echo 1;
    }
}
```

除去访问控制关键字，一个成员方法和常规函数是一样的，从语法解析中调用的函数一样（都是zend_do_begin_function_declaration函数），但是其调用的参数有一些不同，第三个参数is_method，成员方法的赋值为1，表示它作为成员方法的属性。在这个函数中会有一系统的编译判断，比如在接口中不能声明私有的成员方法。看这样一段代码：

```
interface Ifce {
   private function method();
}
```

如果直接运行，程序会报错：Fatal error: Access type for interface method Ifce::method() must be omitted in 这段代码对应到zend_do_begin_function_declaration函数中的代码，如下：

```
if (is_method) {
    if (CG(active_class_entry)->ce_flags & ZEND_ACC_INTERFACE) {
        if ((Z_LVAL(fn_flags_znode->u.constant) &
~(ZEND_ACC_STATIC|ZEND_ACC_PUBLIC))) {
            zend_error(E_COMPILE_ERROR, "Access type for interface method
%s::%s() must be omitted",
                CG(active_class_entry)->name, function_name-
>u.constant.value.str.val);
        }
        Z_LVAL(fn_flags_znode->u.constant) |= ZEND_ACC_ABSTRACT; /* propagates
to the rest of the parser */
    }
    fn_flags = Z_LVAL(fn_flags_znode->u.constant); /* must be done *after* the
above check */
} else {
    fn_flags = 0;
}
```

在此程序判断后，程序将方法直接添加到类结构的function_talbe字段，在此之后，又是若干的编译检测。比如接口的一些魔术方法不能被设置为非公有，不能被设置为static，如__call()、__callStatic()、__get()等。如果在接口中设置了静态方法，如下定义的一个接口：

```
interface ifce {
    public static function __get();
}
```

若运行这段代码，则会显示Warning：Warning: The magic method __get() must have public visibility and cannot be static in

这段编译检测在zend_do_begin_function_declaration函数中对应的源码如下：

```
if (CG(active_class_entry)->ce_flags & ZEND_ACC_INTERFACE) {
        if ((name_len == sizeof(ZEND_CALL_FUNC_NAME)-1) && (!memcmp(lcname,
ZEND_CALL_FUNC_NAME, sizeof(ZEND_CALL_FUNC_NAME)-1))) {
            if (fn_flags & ((ZEND_ACC_PPP_MASK | ZEND_ACC_STATIC) ^
ZEND_ACC_PUBLIC)) {
                zend_error(E_WARNING, "The magic method __call() must have
public visibility and cannot be static");
            }
        } else if() {    //  其它魔术方法的编译检测
        }
}
```

同样，对于类中的这些魔术方法，也有同样的限制，如果在类中定义了静态的魔术方法，则显示警告。如下代码

```
class Tipi {
    public static function __get($var) {

    }
}
```

运行这段代码，则会显示： Warning: The magic method __get() must have public visibility and cannot be static in

与成员变量一样，成员方法也有一个返回所有成员方法的函数--get_class_methods()。 此函数返回由指定的类中定义的方法名所组成的数组。 从 PHP 4.0.6 开始，可以指定对象本身来代替指定的类名。 它属于PHP内建函数，整个程序流程就是一个遍历类成员方法列表，判断是否为符合条件的方法， 如果是，则将这个方法作为一个元素添加到返回数组中。

静态成员方法

类的静态成员方法通常也叫做类方法。 与静态成员变量不同，静态成员方法与成员方法都存储在类结构的function_table 字段。

类的静态成员方法可以通过类名直接访问。

```
class Tipi{
    public static function t() {
        echo 1;
    }
}

Tipi::t();
```

以上的代码在VLD扩展下生成的部分中间代码如如下：

```
number of ops:  8
compiled vars:  none
line     # *  op                                   fetch          ext  return  operands
-------------------------------------------------------------------------------
--
```

```
   2     0  >   EXT_STMT
         1      NOP
   8     2      EXT_STMT
         3      ZEND_INIT_STATIC_METHOD_CALL
'Tipi','t'
         4      EXT_FCALL_BEGIN
         5      DO_FCALL_BY_NAME                                0
         6      EXT_FCALL_END
   9     7    > RETURN                                                  1

branch: #  0; line:     2-    9; sop:     0; eop:     7
path #1: 0,
Class Tipi:
Function t:
Finding entry points
Branch analysis from position: 0
```

从以上的内容可以看出整个静态成员方法的调用是一个先查找方法，再调用的过程。而对于调用操作，对应的中间代码为 ZEND_INIT_STATIC_METHOD_CALL。由于类名和方法名都是常量，于是我们可以知道中间代码对应的函数是 ZEND_INIT_STATIC_METHOD_CALL_SPEC_CONST_CONST_HANDLER。在这个函数中，它会首先调用zend_fetch_class函数，通过类名在EG(class_table)中查找类，然后再执行静态方法的获取方法。

```
if (ce->get_static_method) {
    EX(fbc) = ce->get_static_method(ce, function_name_strval,
function_name_strlen TSRMLS_CC);
} else {
    EX(fbc) = zend_std_get_static_method(ce, function_name_strval,
function_name_strlen TSRMLS_CC);
}
```

如果类结构中的get_static_method方法存在，则调用此方法，如果不存在，则调用zend_std_get_static_method。在PHP的源码中get_static_method方法一般都是NULL，这里我们重点查看zend_std_get_static_method函数。此函数会查找ce->function_table列表，在查找到方法后检查方法的访问控制权限，如果不允许访问，则报错，否则返回函数结构体。关于访问控制，我们在后面的小节中说明。

静态方法和实例方法的小漏洞

细心的读者应该注意到前面提到静态方法和实例方法都是保存在类结构体zend_class_entry.function_table中，那这样的话，Zend引擎在调用的时候是怎么区分这两类方法的，比如我们静态调用实例方法或者实例调用静态方法会怎么样呢？

可能一般人不会这么做，不过笔者有一次错误的这样调用了，而代码没有出现任何问题，在review代码的时候意外发现笔者像实例方法那样调用的静态方法，而什么问题都没有发生(没有报错)。在理论上这种情况是不应发生的，类似这这样的情况在PHP中是非常的多的，例如前面提到的create_function方法返回的伪匿名方法，后面介绍访问控制时还会介绍访问控制的一些瑕疵，PHP在现实中通常采用Quick and Dirty的方式来实现功能和解决问题，这一点和Ruby完整的面向对象形成鲜明的对比。我们先看一个例子：

```
<?php

error_reporting(E_ALL);
```

```
class A {
    public static function staticFunc() {
        echo "static";
    }

    public function instanceFunc() {
        echo "instance";
    }
}

A::instanceFunc(); // instance
$a = new A();
$a->staticFunc();  // static
```

上面的代码静态的调用了实例方法，程序输出了instance，实例调用静态方法也会正确输出static，这说明这两种方法本质上并没有却别。唯一不同的是他们被调用的上下文环境，例如通过实例方法调用方法则上下文中将会有$this这个特殊变量，而在静态调用中将无法使用$this变量。

不过实际上Zend引擎是考虑过这个问题的，将error_reporting的级别增加E_STRICT，将会出出现E_STRICT错误:

```
Strict Standards: Non-static method A::instanceFunc() should not be called
statically
```

这只是不建议将实例方法静态调用，而对于实例调用静态方法没有出现E_STRICT错误，有人说：某些事情可以做并不代表我们要这样做。

PHP在实现新功能时通常采用渐进的方式，保证兼容性，在具体实现上通常采用打补丁的方式，这样就造成有些”边界“情况没有照顾到。

第三节 访问控制的实现

面向对象的三大特性(封装、继承、多态)，其中封装是一个非常重要的特性。封装隐藏了对象内部的细节和实现，使对象能够集中而完整的描述并对应一个具体的事物，只提供对外的访问接口，这样可以在不改变接口的前提下改变实现细节，而且能使对象自我完备。除此之外，封装还可以增强安全性和简化编程。在面向对象的语言中一般是通过访问控制来实现封装的特性。PHP提供了public、protected及private三个层次访问控制。这和其他面向对象的语言中对应的关键字语义一样。这几个关键字都用于修饰类的成员:

- private 用于禁止除类本身以外(包括继承也属于非类本身)对成员的访问，用于隐藏类的内部数据和实现。
- protected 用于禁止除本类以及继承该类的类以外的任何访问。同样用于封装类的实现，同时给予类一定的扩展能力，因为子类还是可以访问到这些成员。
- public 最好理解，被public修饰的成员可以被任意的访问。

> 如果没有设置访问控制关键字，则类的成员方法和成员变量会被设置成默认的 public。

这三个关键字在语法解析时分别对应三种访问控制的标记：

```
member_modifier:
```

```
    T_PUBLIC                { Z_LVAL($$.u.constant) = ZEND_ACC_PUBLIC; }
|   T_PROTECTED             { Z_LVAL($$.u.constant) = ZEND_ACC_PROTECTED; }
|   T_PRIVATE               { Z_LVAL($$.u.constant) = ZEND_ACC_PRIVATE; }
```

这三种访问控制的标记是PHP内核中定义的三个常量，在Zend/zend_compile.h中，其定义如下：

```
#define ZEND_ACC_PUBLIC      0x100
#define ZEND_ACC_PROTECTED   0x200
#define ZEND_ACC_PRIVATE     0x400
#define ZEND_ACC_PPP_MASK   (ZEND_ACC_PUBLIC | ZEND_ACC_PROTECTED |
ZEND_ACC_PRIVATE)
```

我们经常使用16进制的数字表标示状态，例如上面的访问控制常量，0x100使用二进制表示就为 0001 0000 0000 0x200为0010 0000 0000 0x400为0100 0000 0000 我们通过二进制的某个位来表示特定的意义，至于为什么ZEND_ACC_PUBLIC这几个常量后面多两个0，这是因为0x01和0x10已经被占用了，使用和其他不同意义的常量值不一样的值可以避免误用。通过简单的二进制&即可的除某个数值是否表示特定的意义，例如:某个常量为0011 0000 0000，这个数值和 0001 0000 0000 做&，如果结果为0则说明这个位上的值不为1，在上面的例子中就是这个访问控制不具有public的级别。当然PHP中不允许使用多个访问控制修饰符修饰同一个成员。这种处理方式在很多语言中都很常见。

在前面有提到当我们没有给成员方法或成员变量设置访问控制时，其默认值为public。与常规的访问控制实现一样，也是在语法解析阶段进行的。

```
method_modifiers:
        /* empty */
        { Z_LVAL($$.u.constant) = ZEND_ACC_PUBLIC; }
    |   non_empty_member_modifiers          { $$ = $1;
        if (!(Z_LVAL($$.u.constant) & ZEND_ACC_PPP_MASK))
        { Z_LVAL($$.u.constant) |= ZEND_ACC_PUBLIC; } }
;
```

虽然是在语法解析时就已经设置了访问控制，但其最终还是要存储在相关结构中。在上面的语法解析过程中，访问控制已经存储在编译节点中，在编译具体的类成员时会传递给相关的结构。此变量会作为一个参数传递给生成中间代码的函数。如在解析成员方法时，PHP内核是通过调用zend_do_begin_function_declaration 函数实现，此函数的第五个参数表示访问控制，在具体的代码中，

```
// ...省略
fn_flags = Z_LVAL(fn_flags_znode->u.constant);
// ... 省略

op_array.fn_flags |= fn_flags;
//  ...省略
```

如此，就将访问控制的相关参数传递给了将要执行的中间代码。假如我们先现在有下面一段代码：

```
class Tipi{
    private static function t() {
        echo 1;
    }
}

Tipi::t();
```

这个还是上一小节中我们说明静态成员方法的示例，只是，这里我们将其访问控制从public变成了private。执行这段代码会报错：Fatal error: Call to private method Tipi::t() from context '' in...

根据前一节的内容我们知道，如果要执行一个静态成员变量需要先获得类，再获得类的方法，最后执行访方法。而是否有访问权限的检测的实现过程在获取类的方法过程中，即在zend_std_get_static_method函数中。此函数在获取了类的方法后，会执行访问控制的检查过程。

```
if (fbc->op_array.fn_flags & ZEND_ACC_PUBLIC) {
    //公有方法，可以访问
} else if (fbc->op_array.fn_flags & ZEND_ACC_PRIVATE) {
    //  私有方法，报错
} else if ((fbc->common.fn_flags & ZEND_ACC_PROTECTED)) {
    //  保护方法，报错
}
```

> 见前面有关访问控制常量的讨论，这是使用的是 fbc->op_array.fn_flags & ZEND_ACC_PUBLIC 而不是使用==来判断访问控制类型，通过这种方式，op_array.fn_flags中可以保存不止访问控制的信息，所以flag使用的是复数。

对于成员函数来说，其对于访问控制存储在函数结构体中的fn_flags字段中，不管是函数本身的common结构体中的fn_flags，还是函数包含所有中间代码的代码集合op_array中的fn_flags。

访问控制的小漏洞

先看一个小例子吧：

```
<?php

class A {
    private $money = 10000;
    public function doSth($anotherA) {
        $anotherA->money = 10000000000;
    }

    public function getMoney() {
        return $this->money;
    }
}

$b = new A();
echo $b->getMoney(); // 10000

$a = new A();
$a->doSth($b);
echo $b->getMoney(); // 10000000000;
```

在$a变量的doSth()方法中我们直接修改了$b变量的私有成员money，当然我们不太可能这样写代码，从封装的角度来看，这也是不应该的行为，从PHP实现的角度来看，这并不是一个功能，在其他语言中并不是这样表现的。这也是PHP面向对象不纯粹的表现之一。

下面我们从实现上面来看看是什么造就了这样的行为。以下函数为验证某个属性能否被访问的验证方法：

```
static int zend_verify_property_access(zend_property_info *property_info,
zend_class_entry *ce TSRMLS_DC) /* {{{ */
{
    switch (property_info->flags & ZEND_ACC_PPP_MASK) {
        case ZEND_ACC_PUBLIC:
            return 1;
        case ZEND_ACC_PROTECTED:
            return zend_check_protected(property_info->ce, EG(scope));
        case ZEND_ACC_PRIVATE:
            if ((ce==EG(scope) || property_info->ce == EG(scope)) && EG(scope))
{
                return 1;
            } else {
                return 0;
            }
            break;
    }
    return 0;
}
```

在doSth()方法中，我们要访问$b对象的属性money，这是Zend引擎检查我们能否访问$b对象的这个属性，这是Zend赢取获取$b对象的类，以及要访问的属性信息，首先要看看这个属性是否为public，公开的话直接访问就好了。如果是protected的则继续调用zend_check_protected()函数检查，因为涉及到该类的父类，这里不继续跟这个函数了，看看是private的情况下是什么情况，在函数doSth()执行的时候，这时的EG(scope)指向的正是类A，ce变量值得就是变量$b的类，而$b的类就是类A，这样检查就判断成功返回1，也就表示可以访问。

至于成员函数的检查规则类似，就留给读者自己去探索了。

第四节 类的继承，多态及抽象类

面向对象的三大特性(封装、继承、多态)，在前一小节介绍了封装，这一小节我们将介绍继承和多态的实现。

继承

继承是一种关联类的层次模型，它可以建立类之间的关系，并实现代码重用，方便系统扩展。继承提供了一种明确表述共性的方法，是一个新类从现有的类中派生的过程。继承产生的新类继承了原始类的特性，新类称为原始类的派生类（或子类），而原始类称为新类的基类（或父类）。派生类可以从基类那里继承方法和变量，并且新类可以重载或增加新的方法，使之满足自己的定制化的需要。

PHP中使用extends关键字来进行类的继承，一个类只能继承一个父类。被继承的成员方法和成员变量可以使用同名的方法或变量重写，如果需要访问父类的成员方法或变量可以 使用特殊类parent来进行。

PHP内核将类的继承实现放在了"编译阶段"，因此使用VLD生成中间代码时会发现并没有关于继承的相关信息。通过对extends关键字的词法分析和语法分析，在Zend/zend_complie.c文件中 找到继承实现的编译函数zend_do_inheritance()。其调用顺序如下: [zend_do_early_binding] --> [do_bind_inherited_class()] --> [zend_do_inheritance()]

```
ZEND_API void zend_do_inheritance(zend_class_entry *ce, zend_class_entry
```

```
*parent_ce TSRMLS_DC)
{
    //  ...省略  报错处理 接口不能从类继承, final类不能继承

    //  ...省略 序列化函数和反序列化函数 如果当前类没有, 则取父类的

    /* Inherit interfaces */
    zend_do_inherit_interfaces(ce, parent_ce TSRMLS_CC);

    /* Inherit properties */
    zend_hash_merge(&ce->default_properties, &parent_ce->default_properties,
(void (*)(void *)) zval_add_ref, NULL, sizeof(zval *), 0);
    if (parent_ce->type != ce->type) {
        /* User class extends internal class */
        zend_update_class_constants(parent_ce  TSRMLS_CC);
        zend_hash_apply_with_arguments(CE_STATIC_MEMBERS(parent_ce) TSRMLS_CC,
(apply_func_args_t)inherit_static_prop, 1, &ce->default_static_members);
    } else {
        zend_hash_apply_with_arguments(&parent_ce->default_static_members
TSRMLS_CC, (apply_func_args_t)inherit_static_prop, 1, &ce-
>default_static_members);
    }
    zend_hash_merge_ex(&ce->properties_info, &parent_ce->properties_info,
(copy_ctor_func_t) (ce->type & ZEND_INTERNAL_CLASS ?
zend_duplicate_property_info_internal : zend_duplicate_property_info),
sizeof(zend_property_info), (merge_checker_func_t)
do_inherit_property_access_check, ce);

    zend_hash_merge(&ce->constants_table, &parent_ce->constants_table, (void
(*)(void *)) zval_add_ref, NULL, sizeof(zval *), 0);
    zend_hash_merge_ex(&ce->function_table, &parent_ce->function_table,
(copy_ctor_func_t) do_inherit_method, sizeof(zend_function),
(merge_checker_func_t) do_inherit_method_check, ce);
    do_inherit_parent_constructor(ce);

    if (ce->ce_flags & ZEND_ACC_IMPLICIT_ABSTRACT_CLASS && ce->type ==
ZEND_INTERNAL_CLASS) {
        ce->ce_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
    } else if (!(ce->ce_flags & ZEND_ACC_IMPLEMENT_INTERFACES)) {
        /* The verification will be done in runtime by
ZEND_VERIFY_ABSTRACT_CLASS */
        zend_verify_abstract_class(ce TSRMLS_CC);
    }
}
```

整个继承的过程是以类结构为中心，当继承发生时，程序会先处理所有的接口。接口继承调用了zend_do_inherit_interfaces函数 此函数会遍历所有的接口列表，将接口写入到类结构的interfaces字段，并增加num_interfaces的计数统计。 在接口继承后，程序会合并类的成员变量、属性、常量、函数等，这些都是HashTable的merge操作。

在继承过程中，除了常规的函数合并后，还有魔法方法的合并，其调用的函数为do_inherit_parent_constructor(ce)。 此函数实现魔术方法继承，如果子类中没有相关的魔术方法，则继承父类的对应方法。如下所示的PHP代码为子类没构造函数的情况

```
class Base {
    public function __construct() {
        echo 'Base __construct<br />';
    }
}
```

```
class Foo extends Base {

}

$foo = new Foo();
```

在PHP函数中运行，会输出：Base __construct

这显然继承了父类的构造方法，如果子类有自己的构造方法，并且需要调用父类的构造方法时 需要在子类的构造方法中调用父类的构造方法，PHP不会自动调用。

当说到继承，就不得不提到访问控制。继承在不同的访问控制权限下有不同的表现。 以成员方法为例，我们可以使用private和protected访问修饰符来控制需要继承的内容。

- private 如果一个成员被指定为private，它将不能被继承。实际上在PHP中这个方法会被继承下来，只是无法访问。
- protected 如果一个成员被指定为protected，它将在类外不可见，可以被继承。

在继承中访问控制的实现是在合并函数时实现，其实现函数为do_inherit_method_check。 在此函数中，如果子类没有父类中定义的方法，则所有的此类方法都会被继承，包括私有访问控制权限的方法。

看一个PHP的示例：

```
class Base {
    private function privateMethod() {
    }
}

class Child extends Base{
    public function publicMethod() {
    }
}

$c = new Child();

if (method_exists($c, 'privateMethod')) {
    echo 1;
}else{
    echo 0;
}
```

这段代码会输出1，至此，我们可以证明：**在PHP中，对于私有方法，在继承时是可以被继承下来的。**

多态

多态是继数据抽象和继承后的第三个特性。顾名思义，多态即多种形态，相同方法调用实现不同的实现方式。 多态关注一个接口或基类，在编程时不必担心一个对象所属于的具体类。在面向对象的原则中 里氏代换原则（Liskov Substitution Principle，LSP），依赖倒转原则（dependence inversion principle，DIP）等 都依赖于多态特性。而我们在平常工作中也会经常用到。

```
interface Animal {
    public function run();
}
```

```
class Dog implements Animal {
    public function run() {
        echo 'dog run';
    }
}

class  Cat implements Animal{
    public function run() {
        echo 'cat run';
    }
}

class Context {
    private $_animal;

    public function __construct(Animal $animal) {
        $this->_animal = $animal;
    }

    public function run() {
        $this->_animal->run();
    }
}

$dog = new Dog();
$context = new Context($dog);
$context->run();

$cat = new Cat();
$context = new Context();
$context->run();
```

上面是策略模式示例性的简单实现。对于不同的动物，其跑的方式不一样，当在环境中跑的时候，根据所传递进来的动物执行相对应的跑操作。多态是一种编程的思想，但对于不同的语言，其实现也不同。对于PHP的程序实现来说，关键点在于类型提示的实现。而类型提示是PHP5之后才有的特性。在此之前，PHP本身就具有多态特性。

<< 第三章 第五节 类型提示的实现 >>已经说明了类型提示的实现，只是对于 对象的判断没有做深入的探讨。它已经指出对于类的类型提示实现函数为zend_verify_arg_type。在此函数中，关于对象的关键代码如下：

```
if (Z_TYPE_P(arg) == IS_OBJECT) {
    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);
    }
}
```

第一步，判断参数是否为对象，使用宏Z_TYPE_P，如果是转二步，否则跳到其它情况处理

第二步，获取类的类型验证信息，调用了zend_verify_arg_class_kind函数，此函数位于Zend/zend_execute.c文件中，它会通过zend_fetch_class函数获取类信息，根据类的类型判断是否为接口，返回字符串"implement interface"或"be an instance of"

第三步，判断是否为指定类的实例，调用的函数是instanceof_function。此函数首先会遍历实例所在

类的所有接口， 递归调用其本身， 判断实例的接口是否为指定类的实例， 如果是， 则直接返回1， 如果不是， 在非仅接口的情况下， 循环遍历所有的父类， 判断父类与指定的类是否相等， 如果相等返回1， 当函数执行完时仍没有找到， 则返回0， 表示不是类的实例。 instanceof_function函数的代码如下：

```
ZEND_API zend_bool instanceof_function_ex(const zend_class_entry *instance_ce,
const zend_class_entry *ce, zend_bool interfaces_only TSRMLS_DC) /* {{{ */
{
    zend_uint i;

    for (i=0; i<instance_ce->num_interfaces; i++) { //  递归遍历所有的接口
        if (instanceof_function(instance_ce->interfaces[i], ce TSRMLS_CC)) {
            return 1;
        }
    }
    if (!interfaces_only) {
        while (instance_ce) {    //  遍历所有的父类
            if (instance_ce == ce) {
                return 1;
            }
            instance_ce = instance_ce->parent;
        }
    }

    return 0;
}
```

第四步， 如果不是指定类的实例， 程序会调用zend_verify_arg_error报错， 此函数最终会调用zend_error函数显示错误。

接口的实现

前面的PHP示例中有用到接口， 而且在多态中， 接口是一个不得不提的概念。接口是一些方法特征的集合， 是一种逻辑上的抽象， 它没有方法的实现， 因此这些方法可以在不同的地方被实现， 可以有相同的名字而具有完全不同的行为。

而PHP内核对类和接口一视同仁， 它们的内部结构一样。这点在前面的类型提示实现中也有看到， 不管是接口还是类， 调用instanceof_function函数时传入的参数和计算过程中使用的变量都是zend_class_entry类型。

<< 第一节 类的结构和实现 >>中已经对于类的类型做了说明， 在语法解析时， PHP内核已经设置了其type=ZEND_ACC_INTERFACE,

```
interface_entry:
    T_INTERFACE      { $$.u.opline_num = CG(zend_lineno);
              $$.u.EA.type = ZEND_ACC_INTERFACE; }
;
```

而在声明类的函数zend_do_begin_class_declaration中， 通过下列语句， 将语法解析的类的类型赋值给类的ce_flags字段。

```
new_class_entry->ce_flags |= class_token->u.EA.type;
```

类结构的ce_flags字段的作用是标记类的类型。

接口与类除了在ce_flags字段不同外，在其它一些字段的表现上也不一样，如继承时，类只能继承一个父类，却可以实现多个接口。二者在类的结构中存储在不同的字段，类的继承由于是一对一的关系，则每个类都有一个parent字段。而接口实现是一个一对多的关系，每个类都会有一个二维指针存放接口的列表，还有一个存储接口数的字段num_interfaces。

接口也可以和类一样实现继承，并且只能是一个接口继承另一个接口。一个类可以实现多个接口，接口在编译时调用zend_do_implement_interface函数，zend_do_implement_interface函数会合并接口中的常量列表和方法列表操作，这就是接口中不能有变量却可以有常量的实现原因。在接口继承的过程中有对当前类的接口中是否存在同样接口的判断操作，如果已经存在了同样的接口，则此接口继承将不会执行。

抽象类

抽象类是相对于具体类来说的，抽象类仅提供一个类的部分实现。抽象类可以有实例变量，构造方法等。抽象类可以同时拥有抽象方法和具体方法。一般来说，抽象类代表一个抽象的概念，它提供了一个继承的 出发点，理想情况下，所有的类都需要从抽象类继承而来。而具体类则不同，具体类可以实例化。由于抽象类不可以实例化，因此所有抽象类应该都是作为继承的父类的。

在PHP中，抽象类是被abstract关键字修饰的类，或者类没有被声明为abstract，但是在类中存在抽象成员的类。对于这两种情况，PHP内核作了区分，类的结构体zend_class_entry.ce_flags中保存了这些信息，二者对应的值为ZEND_ACC_EXPLICIT_ABSTRACT_CLASS和ZEND_ACC_IMPLICIT_ABSTRACT_CLASS，这两个值在前面的第一节已经做了介绍。

标记类为抽象类或标记成员方法为抽象方法的确认阶段是语法解析阶段。标记为抽象类与标记为接口等的过程一样。而通过标记成员方法为抽象方法来确认一个类为抽象类则是在声明函数时实现的。从第四章中我们知道编译时声明函数会调用 zend_do_begin_function_declaration函数。在此函数中有如下代码：

```
if (fn_flags & ZEND_ACC_ABSTRACT) {
        CG(active_class_entry)->ce_flags |= ZEND_ACC_IMPLICIT_ABSTRACT_CLASS;
    }
}
```

若函数为抽象函数，则设置类的ce_flags为ZEND_ACC_IMPLICIT_ABSTRACT_CLASS，从而将这个类设置为抽象类。

抽象类，接口，普通类都是保存在zend_class_entry结构体中，他们只通过一个标志字段来区分，抽象类和接口还有一个共性：无法实例化。那我们看看Zend在那里限制的。要实例化一个对象我们只能使用new关键字来进行。下面是执行new是进行的操作:

```
static int ZEND_FASTCALL  ZEND_NEW_SPEC_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    zend_op *opline = EX(opline);
    zval *object_zval;
    zend_function *constructor;

    if (EX_T(opline->op1.u.var).class_entry->ce_flags &
(ZEND_ACC_INTERFACE|ZEND_ACC_IMPLICIT_ABSTRACT_CLASS|ZEND_ACC_EXPLICIT_ABSTRACT
_CLASS)) {
```

```
        char *class_type;

        if (EX_T(opline->op1.u.var).class_entry->ce_flags & ZEND_ACC_INTERFACE)
{
            class_type = "interface";
        } else {
            class_type = "abstract class";
        }
        zend_error_noreturn(E_ERROR, "Cannot instantiate %s %s", class_type,
EX_T(opline->op1.u.var).class_entry->name);
    }
    // ...
}
```

代码很好理解，进行了简单的判断，如果为抽象类、隐式抽象类或者接口都无法进行实例化操作。

类的继承、多态、封装，以及访问控制，接口，抽象类等都是基于类的结构实现的，因为这几个类型只有个别的特性的差异，其他基本一致。如果要真正理解这些特性，需要更多的关注类的结构，基础往往很重要，而在程序，数据结构就是程序的基础。

第五节 魔术方法，延迟绑定及静态成员

PHP中有一些特殊的函数和方法，这些函数和方法相比普通方法的特殊之处在于: 用户代码通常不会主动调用，而是在特定的时机会被PHP自动调用。在PHP中通常以"__"打头的方法都作为魔术方法，所以通常不要定义以"__"开头的函数或方法。例如:__autoload()函数，通常我们不会手动调用这个函数，而如果在代码中访问某个未定义的方法，如过已经定义了__autoload()函数，此时PHP将会尝试调用__autoload()函数，例如在类的定义中如果定义了__construct()方法，在初始化类的实例时将会调用这个方法，同理还有__destuct()方法，详细内容请参考PHP手册。

魔术函数和魔术方法

前面提到魔术函数和魔术方法的特殊之处在于这些方法(在这里把函数和方法统称方法)的调用时机是在某些特定的场景才会被触发，这些方法可以理解为一些事件监听方法，在事件触发时才会执行。

魔术方法

根据前面的介绍，魔术方法就是在类的某些场景下触发的一些监听方法。这些方法需要在类定义中进行定义，在存储上魔术方法自然存储于类中，而类在PHP内部是一个**_zend_class_entry**结构体，与普通方法一样，只不过这些类不是存储在类的函数表，而是直接存储在类结构体中:

- 在**_zend_class_entry**结构体中的存储位置不同;
- 由ZendVM自动分情境进行调用;
- 不是必须的，按需定义，自动调用

从以上三个方面可以发现，关于魔术变量的关键理解，主要集中在两个方面：**一，定义在哪里; 二，如何判断其存在并进行调用。**

首先，魔术变量的存储在**_zend_class_entry**中的代码如下：（完整的**_zend_class_entry**代码见本

章第一节）

```
struct _zend_class_entry {
    ...
    //构造方法 __construct
    union _zend_function *constructor;
    //析构方法 __destruct
    union _zend_function *destructor;
    //克隆方法 __clone
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__unset;
    union _zend_function *__isset;
    union _zend_function *__call;
    union _zend_function *__callstatic;
    union _zend_function *__tostring;
    //序列化
    union _zend_function *serialize_func;
    //反序列化
    union _zend_function *unserialize_func;
    ...
}
```

这段代码明确的在对象内部定义了不同的指针来保存各种魔术变量。 关于Zend VM对魔术方法的调用机制， 由于每种方法的调用情境不同， 笔者在这里也分开进行分析。

__construct

__construct构造方法， 在对象创建时被自动调用。 与其它很多语言（如JAVA）不同的是， 在PHP中， 构造方法并没有使用”与类定义同名“的约定方式， 而是单独用魔术方法来实现。 **__construct**方法的调用入口是new关键字对应的ZEND_NEW_SPEC_HANDLER函数。 Zend VM在初始化对象的时候， 使用了new关键字， 对其OPCODE进行分析后， 使用GDB可以得到下面的堆栈信息：

```
#0  ZEND_NEW_SPEC_HANDLER (execute_data=0x100d00080) at zend_vm_execute.h:461
#1  0x000000010041c1f0 in execute (op_array=0x100a1fd60) at
zend_vm_execute.h:107
#2  0x00000001003e9394 in zend_execute_scripts (type=8, retval=0x0,
file_count=3) at /Volumes/DEV/C/php-5.3.4/Zend/zend.c:1194
#3  0x0000000100368031 in php_execute_script (primary_file=0x7fff5fbff890) at
/Volumes/DEV/C/php-5.3.4/main/main.c:2265
#4  0x00000001004d4b5c in main (argc=2, argv=0x7fff5fbffa30) at
/Volumes/DEV/C/php-5.3.4/sapi/cli/php_cli.c:1193
```

上面的椎栈信息清晰显示了new关键的调用过程， 可以发现new关键字对应了ZEND_NEW_SPEC_HANDLER的处理函数， 在ZEND_NEW_SPEC_HANDLER中， Zend VM使用下面的代码来获取对象是否定义了**__construct**方法：

```
...
constructor = Z_OBJ_HT_P(object_zval)->get_constructor(object_zval TSRMLS_CC);
if (constructor == NULL){
    ...
} else {
    ...
}
```

```
//get_constructor的实现
ZEND_API union _zend_function *zend_std_get_constructor(zval *object TSRMLS_DC)
{
    zend_object *zobj = Z_OBJ_P(object);
    zend_function *constructor = zobj->ce->constructor;

    if(constructor){ ... } else { ...}
    ...
}
```

从上面的代码可以看出ZendVM通过读取**zend_object->ce->constructor**的值来判断对象是不是定义的构造函数。

Z_OBJ_P(zval); Z_OBJ_P宏将一个zval类型变量构造为zend_object类型。

在判断了**__construct**魔术变量存在之后，ZEND_NEW_SPEC_HANDLER中对当前EX(called_scope)进行了重新赋值， 使ZEND_VM_NEXT_OPCODE();将opline指针指向__construct方法的op_array，开始执行__construct魔术方法

```
    [c]
    EX(object) = object_zval;
    EX(fbc) = constructor;
    EX(called_scope) = EX_T(opline->op1.u.var).class_entry;
    ZEND_VM_NEXT_OPCODE();
```

__destruct

__destruct是析构方法，运行于对象被显示销毁或者脚本关闭时，一般被用于释放占用的资源。**__destruct**的调用涉及到垃圾回收机制，在第七章中会有更详尽的介绍。 本文笔者只针对**__destruct**调用机制进行分析，其调用堆栈信息如下：

```
//省略部分内存地址信息后的堆栈：
#0  zend_call_function () at /..//php-5.3.4/Zend/zend_execute_API.c:767
#1  zend_call_method () at /..//php-5.3.4/Zend/zend_interfaces.c:97
#2  zend_objects_destroy_object () at /..//php-5.3.4/Zend/zend_objects.c:112
#3  zend_objects_store_del_ref_by_handle_ex () at /..//php-
5.3.4/Zend/zend_objects_API.c:206
#4  zend_objects_store_del_ref () at /..//php-5.3.4/Zend/zend_objects_API.c:172
#5  _zval_dtor_func () at /..//php-5.3.4/Zend/zend_variables.c:52
#6  _zval_dtor () at zend_variables.h:35
#7  _zval_ptr_dtor () at /..//php-5.3.4/Zend/zend_execute_API.c:443
#8  _zval_ptr_dtor_wrapper () at /..//php-5.3.4/Zend/zend_variables.c:189
#9  zend_hash_apply_deleter () at /..//php-5.3.4/Zend/zend_hash.c:614
#10 zend_hash_reverse_apply () at /..//php-5.3.4/Zend/zend_hash.c:763
#11 shutdown_destructors () at /..//php-5.3.4/Zend/zend_execute_API.c:226
#12 zend_call_destructors () at /..//php-5.3.4/Zend/zend.c:874
#13 php_request_shutdown () at /..//php-5.3.4/main/main.c:1587
#14 main () at /..//php-5.3.4/sapi/cli/php_cli.c:1374
```

__destruct方法存在与否是在**zend_objects_destroy_object**函数中进行判断的。 在脚本执行结果时，ZendVM在**php_request_shutdown**阶段会将对象池中的对象一一销毁， 这时如果某对象定义了**__destruct**魔术方法，此方法便会被执行。

在**zend_objects_destroy_object**中，与**__construct**一样，ZendVM判断**zend_object->ce->destructor**是否为空，如果不为空，则调用**zend_call_method**执行**__destruct**析构方法。进入**__destruct**的方式与**__construct**不同的是，**__destruct**的执行方式是由ZendVM直接调用**zend_call_function**来执行。

__call与__callStatic

- **__call**：在对对象不存在的方法进行调用时自动执行;
- **__callStatic**：在对对象不存在的静态方法进行调用时自动执行;

__call与**__callStatic**的调用机制几乎完全相同，关于函数的执行已经在上一章中提到，用户对函数的调用是由**zend_do_fcall_common_helper_SPEC()**方法进行处理的。

__call：

经过**[ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER]-> [zend_do_fcall_common_helper_SPEC]-> [zend_std_call_user_call]-> [zend_call_method]->[zend_call_function]** 调用，经过**zend_do_fcall_common_helper_SPEC**的分发，最终使用**zend_call_function**来执行**__call**。

__callStatic：

经过**[ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER]-> [zend_do_fcall_common_helper_SPEC]-> [zend_std_callstatic_user_call]-> [zend_call_method]->[zend_call_function]** 调用，经过**zend_do_fcall_common_helper_SPEC**的分发，最终使用**zend_call_function**来执行**__callStatic**。

其他魔术方法

PHP中还有很多种魔术方法，它们的处理方式基本与上面类似，运行时执行与否取决的判断根据，最终都是**_zend_class_entry**结构体中对应的指针是否为空。这里列出它们的底层实现函数：

魔术方法	对应处理函数	所在源文件
__set	zend_std_call_setter()	Zend/zend_object_handlers.c
__get	zend_std_call_getter()	Zend/zend_object_handlers.c
__isset	zend_std_call_issetter()	Zend/zend_object_handlers.c
__unset	zend_std_call_unsetter()	Zend/zend_object_handlers.c
__sleep	php_var_serialize_intern()	ext/standard/var.c
__wakeup	php_var_unserialize()	ext/standard/var_unserializer.c
__toString	zend_std_cast_object_tostring()	Zend/zend_object_handlers.c
__invoke	ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER()	Zend/zend_vm_execute.h

__set_state	php_var_export_ex()	ext/standard/var.c
__clone	ZEND_CLONE_SPEC_CV_HANDLER()	Zend/zend_vm_execute.h

延迟绑定

在PHP手册中，对延迟绑定有以下定义。

> 从PHP 5.3.0开始，PHP增加了一个叫做后期静态绑定的功能，用于在继承范围内引用静态调用的类。该功能从语言内部角度考虑被命名为“后期静态绑定”。“后期绑定”的意思是说，static::不再被解析为定义当前方法所在的类，而是在实际运行时计算的。也可以称之为”静态绑定“，因为它可以用于（但不限于）静态方法的调用。

延迟绑定的实现关键在于static关键字，如果以static调用静态方法，则在语法解析时：

```
function_call:
...//省略若干其它情况的函数调用
|   class_name T_PAAMAYIM_NEKUDOTAYIM T_STRING '(' { $4.u.opline_num =
zend_do_begin_class_member_function_call(&$1, &$3 TSRMLS_CC); }
        function_call_parameter_list
        ')' { zend_do_end_function_call($4.u.opline_num?NULL:&$3, &$$, &$6,
$4.u.opline_num, $4.u.opline_num TSRMLS_CC);
zend_do_extended_fcall_end(TSRMLS_C);}
...//省略若干其它情况的函数调用

class_name:
    T_STATIC { $$.op_type = IS_CONST; ZVAL_STRINGL(&$$.u.constant, "static",
sizeof("static")-1, 1);}
```

如上所示，static将以第一个参数(class_name)传递给zend_do_begin_class_member_function_call函数。此时class_name的op_type字段为IS_CONST，但是通过zend_get_class_fetch_type获取此类的类型为ZEND_FETCH_CLASS_STATIC。这个类型作为操作的extended_value字段存在，此字段在后面执行获取类的中间代码ZEND_FETCH_CLASS（ZEND_FETCH_CLASS_SPEC_CONST_HANDLER）时，将作为第三个参数(fetch_type)传递给获取类名的最终执行函数zend_fetch_class。

```
EX_T(opline->result.u.var).class_entry =
zend_fetch_class(Z_STRVAL_P(class_name),
    Z_STRLEN_P(class_name), opline->extended_value TSRMLS_CC);
```

至于在后面如何执行，请查看下一小节：第六节 PHP保留类及特殊类

第六节 PHP保留类及特殊类

在面向对象语言中，都会内置一些语言内置提供的基本功能类，比如JavaScript中的Array，Number等类，PHP中也有很多这种类，比如Directory，stdClass，Exception等类，同时一些标准扩展比如PDO等扩展中也会定义一些类，PHP中类是不允许重复定义的，所以在编写代码时不允许定义已经存在的类。

同时PHP中有一些特殊的类：self，static和parent，相信读者对这self和parent都比较熟悉了，而static特殊类是PHP5.3才引入的。

PHP中的static关键字非常多义：

- 在函数体内的修饰变量的static关键字用于定义静态局部变量。
- 用于修饰类成员函数和成员变量时用于声明静态成员。
- (PHP5.3)在作用域解析符(::)前又表示静态延迟绑定的特殊类。

这个关键字修饰的意义都表示"静态"，在PHP手册中提到self，parent和static这几个关键字，但实际上除了static是关键字以外，其他两个均不是关键字，在手册的关键字列表中也没有这两个关键字，要验证这一点很简单:

```
<?php
var_dump(self); // ->  string(4) "self"
```

上面的代码并没有报错，如果你把error_reporting(E_ALL)打开，就能看到实际是什么情况了: 运行这段代码会出现“ Notice: Use of undefined constant self - assumed 'self'“，也就是说PHP把self当成一个普通常量了，尝试未定义的常量会把产量本身当成一个字符串，例如上例的”self"，不过同时会出一个NOTICE，这就是说self这个标示符并没有什么特殊的。

```
<?php
define('self', "stdClass");
echo self; // stdClass
```

> 不同语言中的关键字的意义会有些区别，Wikipedia上的解释是： 具有特殊含义的标示符或者单词，从这个意义上说$this也算是一个关键字，但在PHP的关键字列表中并没有。 PHP的关键字和C/C++一样属于保留字(关键字)，关键字用于表示特定的语法形式，例如函数定义，流程控制等结构。 这些关键字有他们的特定的使用场景，而上面提到的self和parent并没有这样的限制。

self，parent，static类

前面已经说过self的特殊性。self是一个特殊类，它指向当前类，但只有在类定义内部才有效，但也并不一定指向类本身这个特殊类，比如前面的代码，如果放在类方法体内运行，echo self; 还是会输出常量self的值，而不是当前类，它不止要求在类的定义内部，还要求在类的上下文环境，比如 new self()的时候，这时self就指向当前类，或者self::$static_varible，self::CONSTANT类似的作用域解析符号(::)，这时的self才会作为指向本身的类而存在。

同理parent也和self类似。下面先看看在在类的环境下的编译吧 $PHP_SRC/Zend/zend_language_parser.y:

```
class_name_reference:
        class_name                      { zend_do_fetch_class(&$$, &$1
TSRMLS_CC); }
    |   dynamic_class_name_reference    { zend_do_end_variable_parse(&$1,
BP_VAR_R, 0 TSRMLS_CC); zend_do_fetch_class(&$$, &$1 TSRMLS_CC); }
;
```

在需要获取类名时会执行zend_do_fetch_class()函数：

```
void zend do fetch class(znode *result, znode *class name TSRMLS DC) /* {{{ */
```

```
{
    // ...
    opline->opcode = ZEND_FETCH_CLASS;
    if (class_name->op_type == IS_CONST) {
        int fetch_type;

        fetch_type = zend_get_class_fetch_type(class_name-
>u.constant.value.str.val, class_name->u.constant.value.str.len);
        switch (fetch_type) {
            case ZEND_FETCH_CLASS_SELF:
            case ZEND_FETCH_CLASS_PARENT:
            case ZEND_FETCH_CLASS_STATIC:
                SET_UNUSED(opline->op2);
                opline->extended_value = fetch_type;
                zval_dtor(&class_name->u.constant);
                break;
            default:
                zend_resolve_class_name(class_name, &opline->extended_value, 0
TSRMLS_CC);
                opline->op2 = *class_name;
                break;
        }
    } else {
        opline->op2 = *class_name;
    }
    // ...
}
```

上面省略了一些无关的代码，重点关注fetch_type变量。这是通过zend_get_class_fetch_type()函数获取到的。

```
int zend_get_class_fetch_type(const char *class_name, uint class_name_len) /*
{{{ */
{
    if ((class_name_len == sizeof("self")-1) &&
        !memcmp(class_name, "self", sizeof("self")-1)) {
        return ZEND_FETCH_CLASS_SELF;
    } else if ((class_name_len == sizeof("parent")-1) &&
        !memcmp(class_name, "parent", sizeof("parent")-1)) {
        return ZEND_FETCH_CLASS_PARENT;
    } else if ((class_name_len == sizeof("static")-1) &&
        !memcmp(class_name, "static", sizeof("static")-1)) {
        return ZEND_FETCH_CLASS_STATIC;
    } else {
        return ZEND_FETCH_CLASS_DEFAULT;
    }
}
```

前面的代码是Zend引擎编译类相关操作的代码，下面就到执行阶段了，self，parent等类的指向会在执行时进行获取，找到执行opcode为ZEND_FETCH_CLASS的执行函数:

```
zend_class_entry *zend_fetch_class(const char *class_name, uint class_name_len,
int fetch_type TSRMLS_DC) /* {{{ */
{
    zend_class_entry **pce;
    int use_autoload = (fetch_type & ZEND_FETCH_CLASS_NO_AUTOLOAD) == 0;
    int silent       = (fetch_type & ZEND_FETCH_CLASS_SILENT) != 0;

    fetch_type &= ZEND_FETCH_CLASS_MASK;
```

```
check_fetch_type:
    switch (fetch_type) {
        case ZEND_FETCH_CLASS_SELF:
            if (!EG(scope)) {
                zend_error(E_ERROR, "Cannot access self:: when no class scope
is active");
            }
            return EG(scope);
        case ZEND_FETCH_CLASS_PARENT:
            if (!EG(scope)) {
                zend_error(E_ERROR, "Cannot access parent:: when no class scope
is active");
            }
            if (!EG(scope)->parent) {
                zend_error(E_ERROR, "Cannot access parent:: when current class
scope has no parent");
            }
            return EG(scope)->parent;
        case ZEND_FETCH_CLASS_STATIC:
            if (!EG(called_scope)) {
                zend_error(E_ERROR, "Cannot access static:: when no class scope
is active");
            }
            return EG(called_scope);
        case ZEND_FETCH_CLASS_AUTO: {
                fetch_type = zend_get_class_fetch_type(class_name,
class_name_len);
                if (fetch_type!=ZEND_FETCH_CLASS_DEFAULT) {
                    goto check_fetch_type;
                }
            }
            break;
    }

    if (zend_lookup_class_ex(class_name, class_name_len, use_autoload, &pce
TSRMLS_CC) == FAILURE) {
        if (use_autoload) {
            if (!silent && !EG(exception)) {
                if (fetch_type == ZEND_FETCH_CLASS_INTERFACE) {
                    zend_error(E_ERROR, "Interface '%s' not found",
class_name);
                } else {
                    zend_error(E_ERROR, "Class '%s' not found", class_name);
                }
                }
            }
        }
        return NULL;
    }
    return *pce;
}
```

从这个函数就能看出端倪了，当需要获取self类的时候，则将EG(scope)类返回，而EG(scope)指向的正是当前类。如果时parent类的话则从去EG(scope)->parent也就是当前类的父类，而static获取的时EG(called_scope)，分别说说EG宏的这几个字段，前面已经介绍过EG宏，它可以展开为如下这个结构体:

```
struct _zend_executor_globals {
    // ...
    zend_class_entry *scope;
    zend_class_entry *called_scope; /* Scope of the calling class */
    // ...
}
```

```
struct _zend_class_entry {
    char type;
    char *name;
    zend_uint name_length;
    struct _zend_class_entry *parent;
}
#define struct _zend_class_entry zend_class_entry
```

其中的zend_class_entry就是PHP中类的内部结构表示，zend_class_entry有一个parent字段，也就是该类的父类。在EG结构体中的中called_scope会在执行过程中将当前执行的类赋值给called_scope，例如如下代码:

```
<?php
class A {
    public static funcA() {
        static::funcB();
    }
}

class B {
    public static funcB() {
        echo  "B::funcB()";
    }
}

B::funcA();
```

代码B::funcA()执行的时候，实际执行的是B的父类A中定义的funcA函数，A::funcA()执行时当前的类(scope)指向的是类A，而这个方法是从B类开始调用的，called_scope指向的是类B，static特殊类指向的正是called_scope，也就是当前类(触发方法调用的类)，这也是延迟绑定的原理。

第七节 对象

对象是我们可以进行研究的任何事物，世间万物都可以看作对象。它不仅可以表示我们可以看到的具体事物，也可以表示那些我们看不见的事件等。对象是一个实体，它具有状态，一般我们用变量来表示，同时它也可以具有操作行为，一般用方法来表示，对象就是对象状态和对象行为的集合体。

在之前我们很多次的说到类，对于对象来说，具有相同或相似性质的对象的抽象就是类。因此，对象的抽象是类，类的具体化就是对象，我们常常也说对象是类的实例。从对象的表现形式来看，它和一般的数据类型在形式上十分相似，但是它们在本质是不同的。对象拥有方法，对象间的通信是通过方法调用，以一种消息传递的方式进行。而我们常说的面向对象编程(OOP)使得对象具有交互能力的主要模型就是消息传递模型。对象是消息传递的主体，它可以接收，也可以拒绝外界发来的消息。

这一小节，我们从源码结构来看看PHP实现对象的方法以及其消息传递的方式。

对象的结构

在第三章<< 第一节 变量的内部结构 >>中提到：对象在PHP中是使用一种zend_object_value的结构体来存储。

```
typedef struct _zend_object_value {
    zend_object_handle handle;
        //  unsigned int类型, EG(objects_store).object_buckets的索引
    zend_object_handlers *handlers;
} zend_object_value;
```

PHP内核会将所有的对象存放在一个对象列表容器中，这个列表容器是保存在EG(objects_store)里的一个全局变量。上面的handle字段就是这个列表中object_buckets的索引。当我们需要在PHP中存储对象的时候，PHP内核会根据handle索引从对象列表中获取相对应的对象。而获取的对象有其独立的结构，如下代码所示：

```
typedef struct _zend_object {
    zend_class_entry *ce;
    HashTable *properties;
    HashTable *guards; /* protects from __get/__set ... recursion */
} zend_object;
```

ce是存储该对象的类结构，properties是一个HashTable，用来存放对象的属性。

在zend_object_value结构体中除了索引字段外还有一个包含对象处理方法的字段：handlers。它的类型是zend_object_handlers，我们可以在Zend/zend_object_handlers.h文件中找到它的定义。这是一个包含了多个指针函数的结构体，这些指针函数包括对对象属性的操作，对对象方法的操作，克隆等。此字段会在对象创建的时候初始化。

对象的创建

在PHP代码中，对象的创建是通过关键字 **new** 进行的。从此关键字出发，我们遍历词法分析，语法分析和编译成中间代码等过程，得到其最后执行的函数为 **ZEND_NEW_SPEC_HANDLER** 。

ZEND_NEW_SPEC_HANDLER函数首先会判断对象所对应的类是否为可实例化的类，即判断类的ce_flags是否与ZEND_ACC_INTERFACE、ZEND_ACC_IMPLICIT_ABSTRACT_CLASS或ZEND_ACC_EXPLICIT_ABSTRACT_CLASS有交集，即判断类是否为接口或抽象类。

> 此处的抽象类包括直接声明的抽象类或因为包含了抽象方法而被声明的抽象类

在类的类型判断完成后，如果一切正常，程序会给需要创建的对象存放的ZVAL容器分配内存。然后调用object_init_ex方法初始化类，其调用顺序为：[object_init_ex()] --> [_object_init_ex()] --> [_object_and_properties_init()]

在_object_and_properties_init函数中，程序会执行前面提到的类的类型的判断，然后更新类的静态变量等信息（在这前面的章节有说明），更新完成后，程序会设置zval的类型为IS_OBJECT。

```
Z_TYPE_P(arg) = IS_OBJECT;
```

在设置了类型之后，程序会执行zend_object类型的对象的初始化工作，此时调用的函数是zend_objects_new。

```
ZEND_API zend_object_value zend_objects_new(zend_object **object,
zend_class_entry *class_type TSRMLS_DC)
{
```

```
    zend_object_value retval;

    *object = emalloc(sizeof(zend_object));
    (*object)->ce = class_type;
    retval.handle = zend_objects_store_put(*object, (zend_objects_store_dtor_t)
zend_objects_destroy_object, (zend_objects_free_object_storage_t)
zend_objects_free_object_storage, NULL TSRMLS_CC);
    retval.handlers = &std_object_handlers;
    (*object)->guards = NULL;
    return retval;
}
```

zend_objects_new函数会初始化对象自身的相关信息，包括对象归属于的类，对象实体的存储索引，对象的相关处理函数。在这里将对象放入对象池中的函数为zend_objects_store_put。

在将对象放入对象池，返回对象的存放索引后，程序设置对象的处理函数为标准对象处理函数：std_object_handlers。其位于Zend/zend_object_handles.c文件中。

对象池

这里针对对象，我们引入一个新的概念--对象池。我们将PHP内核在运行中存储所有对象的列表称之为对象池，即EG(objects_store)。这个对象池的作用是存储PHP中间代码运行阶段所有生成的对象，这个思想有点类似于我们做数据库表设计时，当一个实例与另一个实体存在一对多的关系时，将多的那一端对应的实体提取出来存储在一个独立的表一样。这样做的好处有两个，一个是可以对象复用，另一个是节省内存，特别是在对象很大，并且我们不需要用到对象的所有信息时。对象池的存储结构为zend_objects_store结构体，如下；

```
typedef struct _zend_objects_store {
    zend_object_store_bucket *object_buckets;
    zend_uint top;
    zend_uint size;
    int free_list_head;
} zend_objects_store;

typedef struct _zend_object_store_bucket {
    zend_bool destructor_called;
    zend_bool valid;
    union _store_bucket {
        struct _store_object {
            void *object;
            zend_objects_store_dtor_t dtor;
            zend_objects_free_object_storage_t free_storage;
            zend_objects_store_clone_t clone;
            const zend_object_handlers *handlers;
            zend_uint refcount;
            gc_root_buffer *buffered;
        } obj;
        struct {
            int next;
        } free_list;
    } bucket;
} zend_object_store_bucket;
```

针对对象池，PHP内核有一套对象操作API，位于Zend/zend_objects_API.c文件，其列表如下：

- zend_objects_store_init 对象池初始化操作，它的执行阶段是请求初始化阶段，执行顺序是：

[php_request_startup] --> [php_start_sapi] --> [zend_activate] --> [init_executor] 初始化时，它会分配1024个zend_object_store_bucket给对象池。

- zend_objects_store_destroy 销毁对象池，调用efree释放内存
- zend_objects_store_mark_destructed 标记所有对象已经调用了析构函数
- zend_objects_store_free_object_storage 释放存储的对象
- zend_objects_store_put 对象的添加API，在此函数中，程序会执行单个bucket的初始化操作
- zend_objects_store_get_refcount 获取对象池中对象的引用计数
- zend_objects_store_add_ref 对象的引用计数加1，传入值为对象
- zend_objects_store_add_ref_by_handle 通过handle查找对象，并将其引用计数加1
- zend_objects_store_del_ref 对象的引用计数减1，传入值为对象
- zend_objects_store_del_ref_by_handle_ex 通过handle查找对象，并将其引用计数减1，对于引用计数为1的对象有清除处理
- zend_objects_store_clone_obj 对象克隆API，构造一个新的bucket，并将新的对象添加到对象池
- zend_object_store_get_object 获取对象池中bucket中的对象，传入值为对象
- zend_object_store_get_object_by_handle 获取对象池中bucket中的对象，传入值为索引值

成员变量

从前面的对象结构来看，对象的成员变量存储在properties参数中。并且每个对象都会有一套标准的操作函数，如果需要获取成员变量，对象最后调用的是read_property，其对应的标准函数为zend_std_read_property; 如果需要设置成员变量，对象最后调用的是write_property，其对应的标准函数zend_std_write_property。这些函数都是可以定制的，如果有不同的需求，可以通过设置对应的函数指针替换。如在dom扩展中，它的变量的获取函数和设置函数都是定制的。

```
/* {{{ PHP_MINIT_FUNCTION(dom) */
PHP_MINIT_FUNCTION(dom)
{
    zend_class_entry ce;

    memcpy(&dom_object_handlers, zend_get_std_object_handlers(),
sizeof(zend_object_handlers));
    dom_object_handlers.read_property = dom_read_property;
    dom_object_handlers.write_property = dom_write_property;
    //  ...省略
}
```

以上是dom扩展的模块初始化函数的部分内容，在这里，它替换了对象的read_property方法等。

这里我们以标准的操作函数为例说明成员变量的读取和获取。成员变量的获取最终调用的是zend_std_read_property函数。这个函数的流程是这样的：

- 第一步，获取对象的属性，如果存在，转第二步；如果没有相关属性，转第三步
- 第二步，从对象的properties查找是否存在与名称对应的属性存在，如果存在返回结果，如果不存在，转第三步
- 第三步，如果存在__get魔术方法，则调用此方法获取变量，如果不存在，转第四步
- 第四步，如果type=BP_VAR_IS，返回 &EG(uninitialized_zval_ptr)，否则报错

成员变量的设置最终调用的是zend_std_write_property函数。整个执行流程如下：

- 第一步，获取对象的属性，如果存在，转第二步；如果没有相关属性，转第四步
- 第二步，从对象的properties查找是否存在与名称对应的属性存在，如果存在，转第三步，如果不存在，转第四步
- 第三步，如果已有的值和需要设置的值相同，则不执行任何操作，否则执行变量赋值操作，此处的变量赋值操作和常规的变量赋值类似，有一些区别，这里只处理了是否引用的问题
- 第四步，如果存在__set魔术方法，则调用此方法设置变量，如果不存在，转第五步
- 第五步，如果成员变量一直没有被设置过，则直接将此变量添加到对象的properties字段所在HashTable中。

成员方法

成员方法又包括常规的成员方法和魔术方法。魔术方法在前面的第五小节已经介绍过了，这里就不再赘述。在对象的标准函数中并没有成员方法的调用函数，默认情况下设置为NULL。在SPL扩展中，有此函数的调用设置，如下代码：

```
PHP_MINIT_FUNCTION(spl_iterators)
{
    // ...省略
    memcpy(&spl_handlers_dual_it, zend_get_std_object_handlers(),
sizeof(zend_object_handlers));
    spl_handlers_dual_it.get_method = spl_dual_it_get_method;
    /*spl_handlers_dual_it.call_method = spl_dual_it_call_method;*/
    spl_handlers_dual_it.clone_obj = NULL;

    // ...省略
}
```

以下面的PHP代码为例，我们看看成员方法的调用过程：

```
class Tipi {
    public function t() {
        echo 'tipi';
    }
}

$obj = new Tipi();
$obj->t();
```

这是一个简单的类实现，它仅有一个成员方法叫t。创建一个此类的实例，将其赋值给变量$obj，通过这个对象变量执行其成员方法。使用VLD扩展查看其生成的中间代码，可以知道其过程分为初始化成员方法的调用，执行方法两个过程。初始化成员方法的调用对应的中间代码为ZEND_INIT_METHOD_CALL，从我们的调用方式（一个为CV，一个为CONST）可知其对应的执行函数为**ZEND_INIT_METHOD_CALL_SPEC_CV_CONST_HANDLER** 此函数的调用流程如下：

- 第一步，处理调用的方法名，获取其值，并做检验处理：如果不是字符串，则报错
- 第二步，如果第一个操作数是对象，则转第三步，否则报错 Call to a member function t on a non-object
- 第三步，调用对象的get_method函数获取成员方法
- 第四步，其它处理，包括静态方法，this变量等。

而get_method函数一般是指标准实现中的get_method函数，其对应的具体函数为Zend/zend_object_handlers.c文件中zend_std_get_method函数。zend_std_get_method函数的流程如下：

- 第一步，从zobj->ce->function_table中查找是否存在需要调用的函数，如果不存在，转第二步，如果存在，转第三步
- 第二步，如果__call函数存在，则调用zend_get_user_call_function函数获取并返回，如果不存在，则返回NULL
- 第三步，检查方法的访问控制，如果为私有函数，转第四步，否则转第五步
- 第四步，如果为同一个类或父类和这个方法在同一个作用域范围，则返回此方法，否则判断__call函数是否存在，存在则调用此函数，否则报错
- 第五步，处理函数重载及访问控制为protected的情况。转第六步
- 第六步，返回fbc

在获得了函数的信息后，下面的操作就是执行了，关于函数的执行在第四章已经介绍过了。

第八节 命名空间

命名空间概述

在维基百科中，对命名空间的定义是：命名空间（英语：Namespace）表示标识符（identifier）的上下文（context）。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。在编程语言中，命名空间是一种特殊的作用域，它包含了处于该作用域内的标识符，且本身也用一个标识符来表示，这样便将一系列在逻辑上相关的标识符用一个标识符组织了起来。函数和类的作用域可被视作隐式命名空间，它们和可见性、可访问性和对象生命周期不可分割的联系在一起。

命名空间可以看作是一种封装事物的方法，同时也可以看作是组织代码结构的一种形式，在很多语言中都可以见到这种抽象概念和组织形式。在PHP中，命名空间用来解决在编写类库或应用程序时创建可重用的代码如类或函数时碰到的两类问题：

1. 用户编写的代码与PHP内部的类/函数/常量或第三方类/函数/常量之间的名字冲突。
2. 为很长的标识符名称(通常是为了缓解第一类问题而定义的)创建一个别名（或简短）的名称，提高源代码的可读性。

PHP从5.3.0版本开始支持命名空间特性。看一个定义和使用命名空间的示例：

```
<?php
namespace tipi;
class Exception {
    public static $var = 'think in php internal';
}

const E_ALL = "E_ALL IN Tipi";

function strlen(){
    echo 'strlen in tipi';
}

echo Exception::$var;
```

```
echo strlen(Exception::$var);
```

如上所示，定义了命名空间tipi，在这个命名空间内定义了一个Exception类，一个E_ALL常量和一个函数strlen。这些类、常量和函数PHP默认已经实现。假如没有这个命名空间，声明这些类、常量或函数时会报函数重复声明或类重复声明的错误，并且常量的定义也不会成功。

从PHP语言来看，命名空间通过 **namespace** 关键字定义，在命名空间内，可以包括任何合法的PHP代码，但是它的影响范围仅限于类、常量和函数。从语法上来讲，PHP支持在一个文件中定义多个命名空间，但是不推荐这种代码组织方式。当需要将全局的非命名空间中的代码与命名空间中的代码组合在一起，全局代码必须用一个不带名称的 namespace 语句加上大括号括起来。

此时，思考一下，在PHP内核中，命名空间的定义是如何实现的呢？当在多个命名空间中存在多个相同的函数或类时，如何区分？命名空间内的函数如何调用？

命名空间的定义

命名空间在PHP中的实现方案比较简单，不管是函数，类或者常量，在声明的过程中都将命名空间与定义的函数名以\合并起来，作为函数名或类名存储在其对应的容器中。如上面示例中的Exception类，最后存储的类名是tipi\Exception。对于整个PHP实现的架构来说，这种实现方案的代价和对整个代码结构的调整都是最小的。

下面我们以Exception类为例说明整个命名空间的实现。命名空间实现的关键字是namespace，从此关键字开始我们可以找到在编译时处理此关键字的函数为 **zend_do_begin_namespace**。在此函数中，关键是在对CG(current_namespace)的赋值操作，这个值在后面类声明或函数等声明时都会有用到。

在前面我们讲过，类声明的实现在编译时会调用Zend/zend_complie.c文件中的zend_do_begin_class_declaration函数，在此函数中对于命名空间的处理代码如下：

```
if (CG(current_namespace)) {
    /* Prefix class name with name of current namespace */
    znode tmp;

    tmp.u.constant = *CG(current_namespace);
    zval_copy_ctor(&tmp.u.constant);
    zend_do_build_namespace_name(&tmp, &tmp, class_name TSRMLS_CC);
    class_name = &tmp;
    efree(lcname);
    lcname = zend_str_tolower_dup(Z_STRVAL(class_name->u.constant),
Z_STRLEN(class_name->u.constant));
}
```

这段代码的作用是如果当前存在命名空间，则给类名加上命名空间的前缀，如前面提到示例中的tipi\Exception类，添加tipi\的操作就是在这里执行的。在zend_do_build_namespace_name函数中最终会调用zend_do_build_full_name函数实现类名的合并。在函数和常量的声明中存在同样的名称合并操作。这也是命名空间仅对类、常量和函数有效的原因。

使用命名空间

以函数调用为例，当需要调用函数时，会调用zend_do_begin_function_call函数。在此函数中，当使

用到命名空间时会检查函数名，其调用的函数为zend_resolve_non_class_name。 在zend_resolve_non_class_name函数中会根据类型作出判断并返回相关结果：

1. 完全限定名称的函数： 程序首先会做此判断，其判断的依据是第一个字符是否为"\"，这种情况下，在解析时会直接返回。 如类似于\strlen这样以\开头的全局调用或类似于前面定义的\tipi\Exception调用。
2. 所有的非限定名称和限定名称（非完全限定名称）：根据当前的导入规则 程序判断是否为别名，并从编译期间存储别名的HashTable中取出对应的命名空间名称，将其与现有的函数名合并。 关于别名的存储及生成在后面的内容中会说明，
3. 在命名空间内部： 所有的没有根据导入规则转换的限定名称均会在其前面加上当前的命名空间名称。最后判断是否在当前命名空间， 最终程序都会返回一个合并了命名空间的函数名。

别名/导入

允许通过别名引用或导入外部的完全限定名称，是命名空间的一个重要特征。 这有点类似于在类 unix 文件系统中可以创建对其它的文件或目录的符号连接。 PHP 命名空间支持 有两种使用别名或导入方式：为类名称使用别名，或为命名空间名称使用别名。

PHP不支持导入函数或常量

在PHP中，别名是通过操作符 use 来实现的。从而我们可以从源码中找到编译时调用的函数是zend_do_use。 别名在编译为中间代码过程中存放在CG(current_import)中，这是一个HashTable。zend_do_use整个函数的实现基本上是一个查找，判断是否错误，最后写入到HashTable的过程。 其中针对命名空间和类名都有导入的处理过程，而对于常量和函数来说却没有， 这就是PHP不支持导入函数或常量的根本原因所在。

第九节 小结

编程语言的实现是一门非常复杂的工程。包括语法实现，对象模型。流程结构等等的设计和实现。 在面向对象语言中对象模型是尤为重要的，PHP的对象模型比较常规。和Java/C++类似。

本章从面向对象的概念开始，依次介绍了类在PHP内核中的内存表示。 以及抽象类，接口，final类在实现级别上的异同。随后介绍了类的成员变量， 成员方法的内部存储方式。面向对象中的继承，封装和抽象的实现。

PHP中有一些特殊的成员方法，称为魔术方法，本章也介绍了这些魔术的方法的特殊之处和实现原理。

第六章 内存管理

内存是计算机非常关键的部件之一，是暂时存储程序以及数据的空间，CPU只有有限的寄存器可以用于 存储计算数据，而大部分的数据都是存储在内存中的，程序运行都是在内存中进行的。和CPU计算能力一样，内存也是决定计算效率的一个关键部分。

计算中的资源中主要包含：CPU计算能力，内存资源以及I/O。现代计算机为了充分利用资源，而出现了多任务操作系统，通过进程调度来共享CPU计算资源，通过虚拟存储来分享内存存储能力。本章的内存管理中不会介绍操作系统级别的虚拟存储技术，而是关注在应用层面：如何高效的利用有限的内存资源。

目前除了使用C/C++等这类的低层编程语言以外，很多编程语言都将内存管理移到了语言之后，例如Java, 各种脚本语言:PHP/Python/Ruby等等，程序手动维护内存的成本非常大，而这些脚本语言或新型语言都专注于特定领域，这样能将程序员从内存管理中解放出来专注于业务的实现。虽然程序员不需要手动维护内存，而在程序运行过程中内存的使用还是要进行管理的，内存管理的工作也就编程语言实现程序员的工作了。

内存管理的主要工作是尽可能**高效的利用内存**。

内存的使用操作包括申请内存，销毁内存，修改内存的大小等。如果申请了内存在使用完后没有及时释放则可能会造成内存泄露，如果这种情况出现在常驻程序中，久而久之，程序会把机器的内存耗光。所以对于类似于PHP这样没有低层内存管理的语言来说，内存管理是其至关重要的一个模块，它在很大程序上决定了程序的执行效率，

在PHP层面来看，定义的变量、类、函数等等实体在运行过程中都会涉及到内存的申请和释放，例如变量可能会在超出作用域后会进行销毁，在计算过程中会产生的临时数据等都会有内存操作，像类对象，函数定义等数据则会在请求结束之后才会被释放。在这过程中合适申请内存合适释放内存就比较关键了。PHP从开始就有一套属于自己的内存管理机制，在5.3之前使用的是经典的引用计数技术，但引用技术存在一定的技术缺陷，在PHP5.3之后，引入了新的垃圾回收机制，至此，PHP的内存管理机制更加完善。

本章将介绍PHP语言实现中的内存管理技术实现。

第一节 内存管理概述

从某个意义上讲，资源总是有限的，计算机资源也是如此，衡量一个计算机处理能力的指标有很多，根据不同的应用需要也会有不同的指标，比如3D游戏对显卡的性能有要求，而Web服务器对吞吐量及响应时间有要求，通常CPU、内存及硬盘的读取和计算速度具有决定性的作用，在同一时刻这些资源是有限的，正是因为有限我们才需要合理的利用他们。

操作系统的内存管理

当计算机的电源被打开之后，不管你使用的是什么操作系统，这些软件可能已经在使用内存了。这是由计算机的结构决定的，操作系统也是一种软件，只不过它是比较特殊的软件，管理计算机的所有资源，普通应用程序和操作系统的关系有点像老师和学生，老师通常管理一切，而学生的行为会受到老师或学校规定的限制，例如普通应用程序无法直接访问物理内存或者其他硬件资源。

操作系统直接管理着内存，所以操作系统也需要进行内存管理，内存管理是如此之重要，计算机中通常都有内存管理单元(MMU) 用于处理CPU对内存的访问。

应用层的内存管理

由于计算机的内存由操作系统进行管理，所以普通应用程序是无法直接对内存进行访问的，应用程序只能向操作系统申请内存，通常的应用也是这么做的，在需要的时候通过类似malloc之类的库函数 向操作系统申请内存，在一些对性能要求较高的应用场景下是需要频繁的使用和释放内存的，比如Web服务器，编程语言等，由于向操作系统申请内存空间会引发系统调用，系统调用和普通的应用层函数调用性能差别非常大，因为系统调用会将CPU从用户态切换到内核，因为涉及到物理内存的操作，只有操作系统才能进行，而这种切换的成本是非常大的，如果频繁的在内核态和用户态之间切换会产生性能问题。

鉴于系统调用的开销，一些对性能有要求的应用通常会自己在用户态进行内存管理，例如第一次申请稍大的内存留着备用，而使用完释放的内存并不是马上归还给操作系统，可以将内存进行复用，这样可以避免多次的内存申请和释放所带来的性能消耗。

PHP不需要显式的对内存进行管理，这些工作都由Zend引擎进行管理了。PHP内部有一个内存管理体系，它会自动将不再使用的内存垃圾进行释放，这部分的内容后面的小节会介绍到。

PHP中内存相关的功能特性

可能有很多的读者碰到过类似下面的错误吧:

```
Fatal error: Allowed memory size of X bytes exhausted (tried to allocate Y
bytes)
```

这个错误的信息很明确，PHP已经达到了允许使用的最大内存了，通常上来说这很有可能是我们的程序编写的有些问题。比如：一次性读取超大的文件到内存中，或者出现超大的数组，或者在大循环中的没有及时是放掉不再使用的变量，这些都有可能会造成内存占用过大而被终止。

PHP默认的最大内存使用大小是32M, 如果你真的需要使用超过32M的内存可以修改php.ini配置文件的如下配置:

```
memory_limit = 32M
```

如果你无法修改php配置文件，如果你的PHP环境没有禁用ini_set()函数，也可以动态的修改最大的内存占用大小:

```
<?php
ini_set("memory_limit", "128M");
```

既然我们能动态的调整最大的内存占用，那我们是否有办法获取目前的内存占用情况呢？答案是肯定的。

1. memory_get_usage()，这个函数的作用是获取 目前PHP脚本所用的内存大小。
2. memory_get_peak_usage()，这个函数的作用返回 当前脚本到目前位置所占用的内存峰值，这样就

可能获取到目前的脚本的内存需求情况。

单就PHP用户空间提供的功能来说，我们似乎无法控制内存的使用，只能被动的获取内存的占用情况，这样的话我们学习内存管理有什么用呢？

前面的章节有介绍到引用计数，函数表，符号表，常量表等。当我们明白这些信息都会占用内存的时候，我们可以有意的避免不必要的浪费内存，比如我们在项目中通常会使用autoload来避免一次性把不一定会使用的类 包含进来，而这些信息是会占用内存的，如果我们及时把不再使用的变量unset掉之后*可能*会释放掉它所占用的空间，

> 前面之所以会说把变量unset掉时候*可能*会把它释放掉的原因是：在PHP中为了避免不必要的内存复制，采用了引用计数和写时复制的技术，所以这里unset只是将引用关系打破，如果还有其他变量指向该内存，它所占用的内存还是不会被释放的。
> 当然这还有一种情况：出现循环引用，这个就得靠gc来处理了，内存不会当时就是放，只有在gc环节才会被释放。

后面的章节主要介绍PHP在运行时的内存使用和管理细节。这也能帮助我们写出更为内存友好的PHP代码。

第二节 PHP中的内存管理

在前面的小节中我们介绍了内存管理一般会包括以下内容：

1. 是否有足够的内存供我们的程序使用；
2. 如何从足够可用的内存中获取部分内存；
3. 对于使用后的内存，是否可以将其销毁并将其重新分配给其它程序使用。

与此对应，PHP的内容管理也包含这样的内容，只是这些内容在ZEND内核中是以宏的形式作为接口提供给外部使用。后面两个操作分别对应emalloc宏，efree宏，而第一个操作可以根据emalloc宏返回结果检测。

PHP的内存管理可以被看作是分层（hierarchical）的。它分为三层：存储层（storage）、堆层（heap）和接口层（emalloc/efree）。存储层通过 malloc()、mmap() 等函数向系统真正的申请内存，并通过 free() 函数释放所申请的内存。存储层通常申请的内存块都比较大，这里申请的内存大并不是指storage层结构所需要的内存大，只是堆层通过调用存储层的分配方法时，其以大块大块的方式申请的内存，存储层的作用是将内存分配的方式对堆层透明化。如图6.1所示，PHP内存管理器。PHP在存储层共有4种内存分配方案: malloc，win32，mmap_anon，mmap_zero，默认使用malloc分配内存，如果设置了ZEND_WIN32宏，则为windows版本，调用HeapAlloc分配内存，剩下两种内存方案为匿名内存映射，并且PHP的内存方案可以通过设置环境变量来修改。

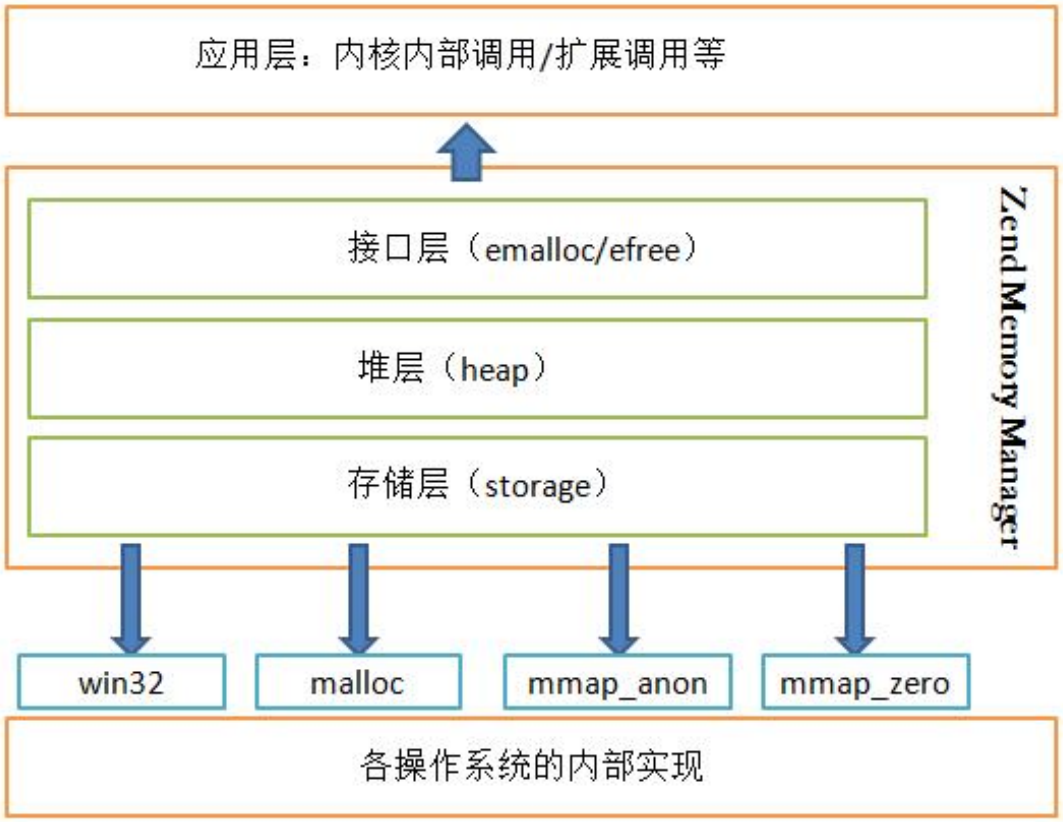


图6.1 PHP内存管理器

首先我们看下接口层的实现，接口层是一些宏定义，如下：

```
/* Standard wrapper macros */
#define emalloc(size)                           _emalloc((size) ZEND_FILE_LINE_CC
ZEND_FILE_LINE_EMPTY_CC)
#define safe_emalloc(nmemb, size, offset)    _safe_emalloc((nmemb), (size),
(offset) ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define efree(ptr)                              _efree((ptr) ZEND_FILE_LINE_CC
ZEND_FILE_LINE_EMPTY_CC)
#define ecalloc(nmemb, size)                    _ecalloc((nmemb), (size)
ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define erealloc(ptr, size)                     _erealloc((ptr), (size), 0
ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define safe_erealloc(ptr, nmemb, size, offset) _safe_erealloc((ptr), (nmemb),
(size), (offset) ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define erealloc_recoverable(ptr, size)      _erealloc((ptr), (size), 1
ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define estrdup(s)                              _estrdup((s) ZEND_FILE_LINE_CC
ZEND_FILE_LINE_EMPTY_CC)
#define estrndup(s, length)                     _estrndup((s), (length)
ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define zend_mem_block_size(ptr)                _zend_mem_block_size((ptr)
TSRMLS_CC ZEND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
```

这里为什么没有直接调用函数？因为这些宏相当于一个接口层或中间层，定义了一个高层次的接口，使得调用更加容易 它隔离了外部调用和PHP内存管理的内部实现，实现了一种松耦合关系。虽然PHP不限制这些函数的使用，但是官方文档还是建议使用这些宏。这里的接口层有点门面模式(facade模式)的味道。

在接口层下面是PHP内存管理的核心实现，我们称之为heap层。这个层控制整个PHP内存管理的过

程，首先我们看这个层的结构：

```
/* mm block type */
typedef struct _zend_mm_block_info {
    size_t _size;    /* block的大小*/
    size_t _prev;    /* 计算前一个块有用到*/
} zend_mm_block_info;


typedef struct _zend_mm_block {
    zend_mm_block_info info;
} zend_mm_block;

typedef struct _zend_mm_small_free_block {  /* 双向链表 */
    zend_mm_block_info info;
    struct _zend_mm_free_block *prev_free_block;     /* 前一个块 */
    struct _zend_mm_free_block *next_free_block;     /* 后一个块 */
} zend_mm_small_free_block; /* 小的空闲块*/

typedef struct _zend_mm_free_block {     /* 双向链表 + 树结构 */
    zend_mm_block_info info;
    struct _zend_mm_free_block *prev_free_block;     /* 前一个块 */
    struct _zend_mm_free_block *next_free_block;     /* 后一个块 */

    struct _zend_mm_free_block **parent;     /* 父结点 */
    struct _zend_mm_free_block *child[2];    /* 两个子结点*/
} zend_mm_free_block;



struct _zend_mm_heap {
    int                 use_zend_alloc; /* 是否使用zend内存管理器 */
    void               *(*_malloc)(size_t); /* 内存分配函数*/
    void                (*_free)(void*);     /* 内存释放函数*/
    void               *(*_realloc)(void*, size_t);
    size_t              free_bitmap;     /* 小块空闲内存标识 */
    size_t              large_free_bitmap;   /* 大块空闲内存标识*/
    size_t              block_size;      /* 一次内存分配的段大小，即
ZEND_MM_SEG_SIZE指定的大小，默认为ZEND_MM_SEG_SIZE    (256 * 1024)*/
    size_t              compact_size;    /* 压缩操作边界值，为ZEND_MM_COMPACT指定大
小，默认为 2 * 1024 * 1024*/
    zend_mm_segment    *segments_list;   /* 段指针列表 */
    zend_mm_storage    *storage;     /* 所调用的存储层 */
    size_t              real_size;   /* 堆的真实大小 */
    size_t              real_peak;   /* 堆真实大小的峰值 */
    size_t              limit;   /* 堆的内存边界 */
    size_t              size;    /* 堆大小 */
    size_t              peak;    /* 堆大小的峰值*/
    size_t              reserve_size;    /* 备用堆大小*/
    void               *reserve;     /* 备用堆 */
    int                 overflow;    /* 内存溢出数*/
    int                 internal;
#if ZEND_MM_CACHE
    unsigned int        cached; /* 已缓存大小 */
    zend_mm_free_block *cache[ZEND_MM_NUM_BUCKETS]; /* 缓存数组/
#endif
    zend_mm_free_block *free_buckets[ZEND_MM_NUM_BUCKETS*2];     /* 小块内存数组，
相当索引的角色 */
    zend_mm_free_block *large_free_buckets[ZEND_MM_NUM_BUCKETS];     /* 大块内存
数组，相当索引的角色 */
    zend_mm_free_block *rest_buckets[2];     /* 剩余内存数组*/

};
```

当初始化内存管理时，调用函数是zend_mm_startup。它会初始化storage层的分配方案，初始化段大小，压缩边界值，并调用zend_mm_startup_ex()初始化堆层。这里的分配方案就是图6.1所示的四种方案，它对应的环境变量名为：ZEND_MM_MEM_TYPE。这里的初始化的段大小可以通过ZEND_MM_SEG_SIZE设置，如果没设置这个环境变量，程序中默认为256 * 1024。这个值存储在_zend_mm_heap结构的block_size字段中，将来在维护的三个列表中都没有可用的内存中，会参考这个值的大小来申请内存的大小。

PHP中的内存管理主要工作就是维护三个列表：小块内存列表（free_buckets）、大块内存列表（large_free_buckets）和剩余内存列表（rest_buckets）。看到bucket这个单词是不是很熟悉？在前面我们介绍HashTable时，这就是一个重要的角色，它作为HashTable中的一个单元角色。在这里，每个bucket也对应一定大小的内存块列表，这样的列表都包含双向链表的实现。

我们可以把维护的前面两个表看作是两个HashTable，那么，每个HashTable都会有自己的hash函数。首先我们来看free_buckets列表，这个列表用来存储小块的内存分配，其hash函数为：

```
#define ZEND_MM_BUCKET_INDEX(true_size)
((true_size>>ZEND_MM_ALIGNMENT_LOG2)-
(ZEND_MM_ALIGNED_MIN_HEADER_SIZE>>ZEND_MM_ALIGNMENT_LOG2))
```

假设ZEND_MM_ALIGNMENT为8（如果没有特殊说明，本章的ZEND_MM_ALIGNMENT的值都为8），则ZEND_MM_ALIGNED_MIN_HEADER_SIZE=16，若此时true_size=256，则((256>>3)-(16>>3))= 30。当ZEND_MM_BUCKET_INDEX宏出现时，ZEND_MM_SMALL_SIZE宏一般也会同时出现，ZEND_MM_SMALL_SIZE宏的作用是判断所申请的内存大小是否为小块的内存，在上面的示例中，小于272Byte的内存为小块内存，则index最多只能为31，这样就保证了free_buckets不会出现数组溢出的情况。

在内存管理初始化时，PHP内核对初始化free_buckets列表。从heap的定义我们可知free_buckets是一个数组指针，其存储的本质是指向zend_mm_free_block结构体的指针。开始时这些指针都没有指向具体的元素，只是一个简单的指针空间。free_buckets列表在实际使用过程中只存储指针，这些指针以两个为一对（即数组从0开始，两个为一对），分别存储一个个双向链表的头尾指针。其结构如图6.2所示。

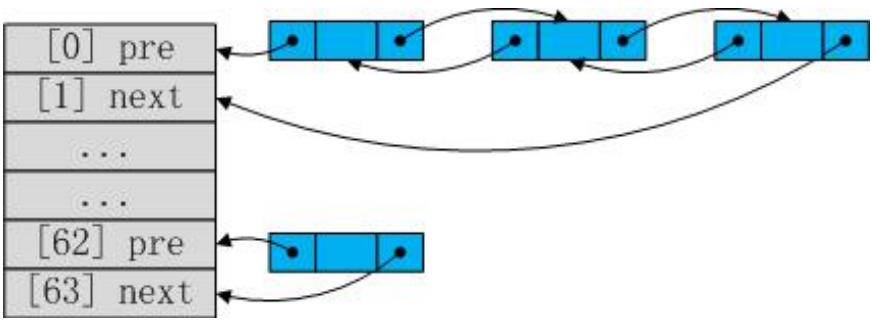


图6.2 free_buckets列表结构

对于free_buckets列表位置的获取，关键在于ZEND_MM_SMALL_FREE_BUCKET宏，宏代码如下：

```
#define ZEND_MM_SMALL_FREE_BUCKET(heap, index) \
(zend_mm_free_block*) ((char*)&heap->free_buckets[index * 2] + \
    sizeof(zend_mm_free_block*) * 2 - \
    sizeof(zend_mm_small_free_block))
```

仔细看这个宏实现，发现在它的计算过程是取free_buckets列表的偶数位的内存地址加上 两个指针的

内存大小并减去zend_mm_small_free_block结构所占空间的大小。 而zend_mm_free_block结构和zend_mm_small_free_block结构的差距在于两个指针。 据此计算过程可知， ZEND_MM_SMALL_FREE_BUCKET宏会获取free_buckets列表 index对应双向链表的第一个zend_mm_free_block的prev_free_block指向的位置。 free_buckets的计算仅仅与prev_free_block指针和next_free_block指针相关， 所以free_buckets列表也仅仅需要存储这两个指针。

那么， 这个数组在最开始是怎样的呢？ 在初始化函数zend_mm_init中free_buckets与large_free_buckts列表一起被初始化。 如下代码：

```
p = ZEND_MM_SMALL_FREE_BUCKET(heap, 0);
for (i = 0; i < ZEND_MM_NUM_BUCKETS; i++) {
    p->next_free_block = p;
    p->prev_free_block = p;
    p = (zend_mm_free_block*)((char*)p + sizeof(zend_mm_free_block*) * 2);
    heap->large_free_buckets[i] = NULL;
}
```

对于free_buckets列表来说， 在循环中， 偶数位的元素（索引从0开始）将其next_free_block和prev_free_block都指向自己， 以i=0为例， free_buckets的第一个元素(free_buckets[0])存储的是第二个元素(free_buckets[1])的地址， 第二个元素存储的是第一个元素的地址。 此时将可能会想一个问题， 在整个free_buckets列表没有内容时， ZEND_MM_SMALL_FREE_BUCKET在获取第一个zend_mm_free_block时， 此zend_mm_free_block的next_free_block元素和prev_free_block元素却分别指向free_buckets[0]和free_buckets[1]。

在整个循环初始化过程中都没有free_buckets数组的下标操作， 它的移动是通过地址操作， 以加两个sizeof(zend_mm_free_block*)实现， 这里的sizeof(zend_mm_free_block*)是获取指针的大小。比如现在是在下标为0的元素的位置， 加上两个指针的值后， 指针会指向下标为2的地址空间， 从而实现数组元素的向后移动， 也就是zend_mm_free_block->next_free_block和zend_mm_free_block->prev_free_block位置的后移。 这种不存储zend_mm_free_block数组， 仅存储其指针的方式不可不说精妙。虽然在理解上有一些困难， 但是节省了内存。

free_buckets列表使用free_bitmap标记是否该双向链表已经使用过时有用。 当有新的元素需要插入到列表时， 需要先根据块的大小查找index， 查找到index后， 在此index对应的双向链表的头部插入新的元素。

free_buckets列表的作用是存储小块内存， 而与之对应的large_free_buckets列表的作用是存储大块的内存， 虽然large_free_buckets列表也类似于一个hash表， 但是这个与前面的free_buckets列表一些区别。 它是一个集成了数组， 树型结构和双向链表三种数据结构的混合体。 我们先看其数组结构， 数组是一个hash映射， 其hash函数为：

```
#define ZEND_MM_LARGE_BUCKET_INDEX(S) zend_mm_high_bit(S)


static inline unsigned int zend_mm_high_bit(size_t _size)
{

..//省略若干不同环境的实现
    unsigned int n = 0;
    while (_size != 0) {
        _size = _size >> 1;
        n++;
    }
```

```
    return n-1;
}
```

这个hash函数用来计算size中最高位的1的比特位是多少，这点从其函数名就可以看出。假设此时size为512Byte，则这段内存会放在large_free_buckets列表，512的二进制码为1000000000，则zend_mm_high_bit(512)计算的值为9，则其对应的列表index为9。关于右移操作，这里有一点说明：

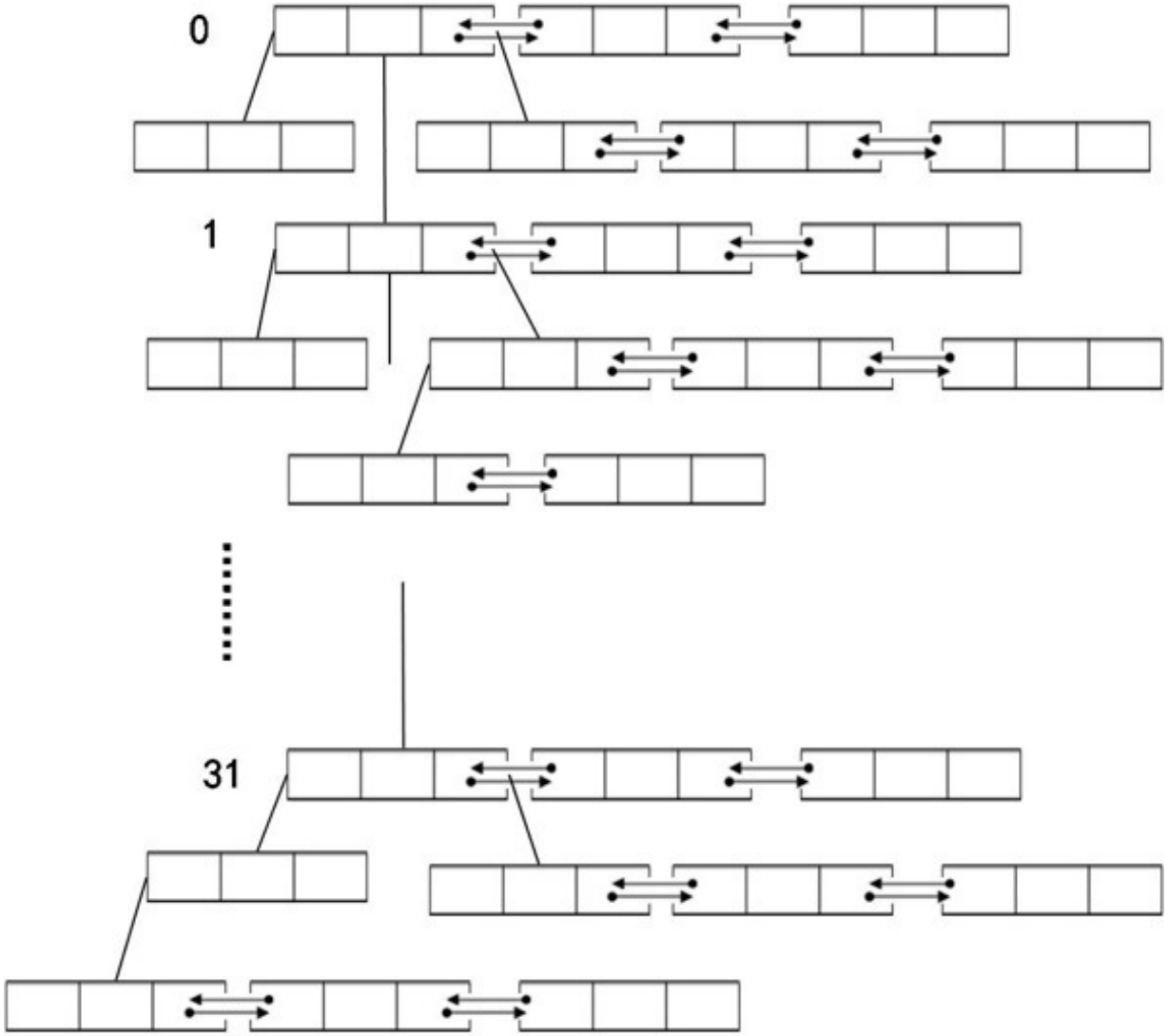
> 一般来说，右移分为逻辑右移和算术右移。逻辑位移在在左端补K个0，算术右移在左端补K个最高有效位的值。C语言标准没有明确定义应该使用哪种方式。对于无符号数据，右移必须是逻辑的。对于有符号的数据，则二者都可以。但是，现实中都会默认为算术右移。

以上的zend_mm_high_bit函数实现是节选的最后C语言部分（如果对汇编不了解的话，看这部分会比较容易一些）。但是它却是最后一种选择，在其它环境中，如x86的处理中可以使用汇编语言BSR达到这段代码的目的，这样的速度会更快一些。这个汇编语句是BSR（Bit Scan Reverse），BSR被称为逆向位扫描指令。它使用方法为：BSF dest,src，它的作用是从源操作数的的最高位向低位搜索，将遇到的第一个“1”所在的位序号存入目标寄存器。

我们通过一次列表的元素插入操作来理解列表的结果。首先确定当前需要内存所在的数组元素位置，然后查找此内存大小所在的位置。这个查找行为是发生在树型结构中，而树型结构的位置与内存的大小有关。其查找过程如下：

- 第一步 通过索引获取树型结构第一个结点并作为当前结点，如果第一个结点为空，则将内存放到第一个元素的结点位置，返回，否则转第二步
- 第二步 从当前结点出发，查找下一个结点，并将其作为当前结点
- 第三步 判断当前结点内存的大小与需要分配的内存大小是否一样 如果大小一样则以双向链表的结构将新的元素添加到结点元素的后面第一个元素的位置。否则转四步
- 第四步 判断当前结点是否为空，如果为空，则占据结点位置，结束查找，否则第二步。

从以上的过程我们可以画出large_free_buckets列表的结构如图6.3所示：



PHP内存管理large_free_buckets列表结构图

图6.3 large_free_buckets列表结构

从内存分配的过程中可以看出，内存块查找判断顺序依次是小块内存列表，大块内存列表，剩余内存列表。在heap结构中，剩余内存列表对应rest_buckets字段，这是一个包含两个元素的数组，并且也是一个双向链表队列，其中rest_buckets[0]为队列的头，rest_buckets[1]为队列的尾。而我们常用的插入和查找操作是针对第一个元素，即heap->rest_buckets[0]，当然，这是一个双向链表队列，队列的头和尾并没有很明显的区别。它们仅仅是作为一种认知上的区分。在添加内存时，如果所需要的内存块的大小大于初始化时设置的ZEND_MM_SEG_SIZE的值（在heap结构中为block_size字段）与ZEND_MM_ALIGNED_SEGMENT_SIZE(等于8)和ZEND_MM_ALIGNED_HEADER_SIZE(等于8)的和的差，则会将新生成的块插入 rest_buckts所在的双向链表中，这个操作和前面的双向链表操作一样，都是从”队列头“插入新的元素。此列表的结构和free_bucket类似，只是这个列表所在的数组没有那么多元素，也没有相应的hash函数。

在heap层下面是存储层，存储层的作用是将内存分配的方式对堆层透明化，实现存储层和heap层的分离。在PHP的源码中有注释显示相关代码为"Storage Manager"。存储层的主要结构代码如下：

```
/* Heaps with user defined storage */
typedef struct _zend_mm_storage zend_mm_storage;

typedef struct _zend_mm_segment {
```

```
    size_t     size;
    struct _zend_mm_segment *next_segment;
} zend_mm_segment;

typedef struct _zend_mm_mem_handlers {
    const char *name;
    zend_mm_storage* (*init)(void *params);     //     初始化函数
    void (*dtor)(zend_mm_storage *storage);     //     析构函数
    void (*compact)(zend_mm_storage *storage);
    zend_mm_segment* (*_alloc)(zend_mm_storage *storage, size_t size);     //
内存分配函数
    zend_mm_segment* (*_realloc)(zend_mm_storage *storage, zend_mm_segment
*ptr, size_t size);     //     重新分配内存函数
    void (*_free)(zend_mm_storage *storage, zend_mm_segment *ptr);     //     释
放内存函数
} zend_mm_mem_handlers;

struct _zend_mm_storage {
    const zend_mm_mem_handlers *handlers;     //     处理函数集
    void *data;
};
```

以上代码的关键在于存储层处理函数的结构体，对于不同的内存分配方案，所不同的就是内存分配的处理函数。其中以name字段标识不同的分配方案。在图6.1中，我们可以看到PHP在存储层共有4种内存分配方案: malloc，win32，mmap_anon，mmap_zero默认使用malloc分配内存，如果设置了ZEND_WIN32宏，则为windows版本，调用HeapAlloc分配内存，剩下两种内存方案为匿名内存映射，并且PHP的内存方案可以通过设置变量来修改。其官方说明如下：

```
The Zend MM can be tweaked using ZEND_MM_MEM_TYPE and ZEND_MM_SEG_SIZE
environment
variables. Default values are “malloc” and “256K”. Dependent on target system
you
can also use “mmap_anon”, “mmap_zero” and “win32” storage managers.
```

在代码中，对于这4种内存分配方案，分别对应实现了zend_mm_mem_handlers中的各个处理函数。配合代码的简单说明如下：

```
/* 使用mmap内存映射函数分配内存 写入时拷贝的私有映射，并且匿名映射，映射区不与任何文件关
联。*/
# define ZEND_MM_MEM_MMAP_ANON_DSC {"mmap_anon", zend_mm_mem_dummy_init,
zend_mm_mem_dummy_dtor, zend_mm_mem_dummy_compact, zend_mm_mem_mmap_anon_alloc,
zend_mm_mem_mmap_realloc, zend_mm_mem_mmap_free}

/* 使用mmap内存映射函数分配内存 写入时拷贝的私有映射，并且映射到/dev/zero。*/
# define ZEND_MM_MEM_MMAP_ZERO_DSC {"mmap_zero", zend_mm_mem_mmap_zero_init,
zend_mm_mem_mmap_zero_dtor, zend_mm_mem_dummy_compact,
zend_mm_mem_mmap_zero_alloc, zend_mm_mem_mmap_realloc, zend_mm_mem_mmap_free}

/* 使用HeapAlloc分配内存 windows版本 关于这点，注释中写的是VirtualAlloc() to
allocate memory，实际在程序中使用的是HeapAlloc*/
# define ZEND_MM_MEM_WIN32_DSC {"win32", zend_mm_mem_win32_init,
zend_mm_mem_win32_dtor, zend_mm_mem_win32_compact, zend_mm_mem_win32_alloc,
zend_mm_mem_win32_realloc, zend_mm_mem_win32_free}

/* 使用malloc分配内存 默认为此种分配 如果有加ZEND_WIN32宏，则使用win32的分配方案*/
# define ZEND_MM_MEM_MALLOC_DSC {"malloc", zend_mm_mem_dummy_init,
zend_mm_mem_dummy_dtor, zend_mm_mem_dummy_compact, zend_mm_mem_malloc_alloc,
zend_mm_mem_malloc_realloc, zend_mm_mem_malloc_free}
```

```
static const zend_mm_mem_handlers mem_handlers[] = {
#ifdef HAVE_MEM_WIN32
    ZEND_MM_MEM_WIN32_DSC,
#endif
#ifdef HAVE_MEM_MALLOC
    ZEND_MM_MEM_MALLOC_DSC,
#endif
#ifdef HAVE_MEM_MMAP_ANON
    ZEND_MM_MEM_MMAP_ANON_DSC,
#endif
#ifdef HAVE_MEM_MMAP_ZERO
    ZEND_MM_MEM_MMAP_ZERO_DSC,
#endif
    {NULL, NULL, NULL, NULL, NULL, NULL}
};
```

假设我们使用的是win32内存方案，则在PHP编译时，编译器会选择将ZEND_MM_MEM_WIN32_DSC宏所代码的所有处理函数赋值给mem_handlers。在之后我们调用内存分配时，将会使用此数组中对应的相关函数。当然，在指定环境变量 USE_ZEND_ALLOC 时，可用于允许在运行时选择 malloc 或 emalloc 内存分配。使用 malloc-type 内存分配将允许外部调试器观察内存使用情况，而 emalloc 分配将使用 Zend 内存管理器抽象，要求进行内部调试。

第三节 内存使用：申请和销毁

内存的申请

通过前一小节我们可以知道，PHP底层对内存的管理，围绕着小块内存列表（free_buckets）、大块内存列表（large_free_buckets）和 剩余内存列表（rest_buckets）三个列表来分层进行的。ZendMM向系统进行的内存申请，并不是有需要时向系统即时申请，而是由ZendMM的最底层（heap层）先向系统申请一大块的内存，通过对上面三种列表的填充，建立一个类似于内存池的管理机制。在程序运行需要使用内存的时候，ZendMM会在内存池中分配相应的内存供使用。这样做的好处是避免了PHP向系统频繁的内存申请操作，如下面的代码：

```
<?php
$tipi = "o_o\n";
echo $tipi;
?>
```

这是一个简单的php程序，但通过对emalloc的调用计数，发现对内存的请求有数百次之多，当然这非常容易解释，因为PHP脚本的执行，需要大量的环境变量以及内部变量的定义，这些定义本身都是需要在内存中进行存储的。

> 在编写PHP的扩展时，推荐使用emalloc来代替malloc，其实也就是使用PHP的ZendMM来代替 手动直接调用系统级的内存管理。（除非，你自己知道自己在做什么。）

那么在上面这个小程序的执行过程中，ZendMM是如何使用自身的heap层存储空间的呢？经过对源码的追踪我们可以找到：

```
    ZEND_ASSIGN_SPEC_CV_CONST_HANDLER (......)
-> ALLOC_ZVAL(......)
```

```
-> ZEND_FAST_ALLOC(......)
-> emalloc (......)
-> _emalloc(......)
-> _zend_mm_alloc_int(.....)
```

void *_emalloc 实现了对内存的申请操作，在_emalloc的处理过程中，对是否使用ZendMM进行了判断，如果heap层没有使用ZendMM来管理，就直接使用_zend_mm_heap结构中定义的_malloc函数进行内存的分配；（我们通过上节可以知道，这里的_malloc可以是malloc，win32，mmap_anon，mmap_zero中的一种）；

就目前所知，不使用ZendMM进行内存管理，唯一的用途是打开enable-debug开关后，可以更方便的追踪内存的使用情况。所以，在这里我们关注ZendMM使用_zend_mm_alloc_int函数进行内存分配:

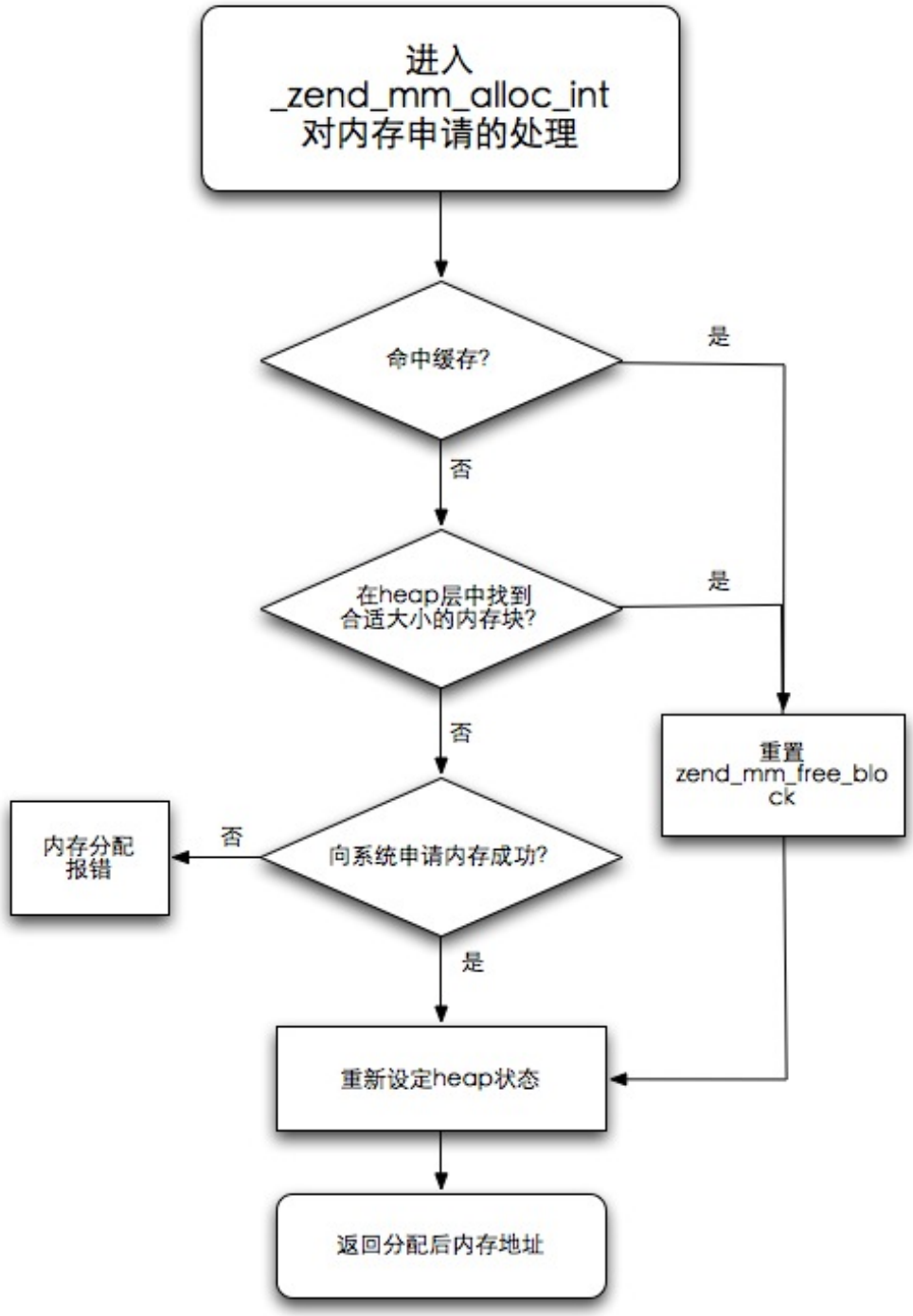


图6.1 PHP内存管理器

结合上图，再加上内存分配之前的验证，ZendMM对内存分配的处理主要有以下步骤：

1. 内存检查。 对要申请的内存大小进行检查， 如果太大（超出memory_limit则报 Out of Memory）；
2. 如果命中缓存， 使用fastcache得到内存块(详见第五节)， 然后直接进行第5步;
3. 在ZendMM管理的heap层存储中搜索合适大小的内存块, 在这一步骤ZendMM通过与ZEND_MM_MAX_SMALL_SIZE进行大小比较， 把内存请求分为两种类型： large和small。small类型的的请求会先使用zend_mm_low_bit函数 在mm_heap中的free_buckets中查找， 未找到则使用与large类型相同的方式： 使用zend_mm_search_large_block函数在“大块”内存（_zend_mm_heap->large_free_buckets）中进行查找。 如果还没有可以满足大小需求的内存， 最后在rest_buckets中进行查找。 也就是说， 内存的分配是在三种列表中小到大进行的。 找到可以使用的block后， 进行第5步;
4. 如果经过第3步的查找还没有找到可以使用的资源（请求的内存过大）， 需要使用ZEND_MM_STORAGE_ALLOC函数向系统再申请一块内存（大小至少为ZEND_MM_SEG_SIZE）， 然后直接将对齐后的地址分配给本次请求。跳到第6步;
5. 使用zend_mm_remove_from_free_list函数将已经使用block节点在zend_mm_free_block中移除;
6. 内存分配完毕， 对zend_mm_heap结构中的各种标识型变量进行维护， 包括large_free_buckets, peak， size等;
7. 返回分配的内存地址;

从上面的分配可以看出， PHP对内存的分配， 是结合PHP的用途来设计的， PHP一般用于web应用程序的数据支持， 单个脚本的运行周期一般比较短（最多达到秒级）， 内存大块整块的申请， 自主进行小块的分配， 没有进行比较复杂的不相临地址的空闲内存合并， 而是集中再次向系统请求。 这样做的好处就是运行速度会更快， 缺点是随着程序的运行时间的变长， 内存的使用情况会“越来越多”（PHP5.2及更早版本）。 所以PHP5.3之前的版本并不适合做为守护进程长期运行。 （当然， 可以有其他方法解决， 而且在PHP5.3中引入了新的GC机制， 详见下一小节）

内存的销毁

ZendMM在内存销毁的处理上采用与内存申请相同的策略， 当程序unset一个变量或者是其他的释放行为时， ZendMM并不会直接立刻将内存交回给系统， 而是只在自身维护的内存池中将其重新标识为可用, 按照内存的大小整理到上面所说的三种列表（small,large,free）之中， 以备下次内存申请时使用。

关于变量销毁的处理， 还涉及较多的其他操作， 请参看变量的创建和销毁

内存销毁的最终实现函数是**_efree**。在**_efree**中， 内存的销毁首先要进行是否放回cache的判断。 如果内存的大小满足ZEND_MM_SMALL_SIZE并且cache还没有超过系统设置的ZEND_MM_CACHE_SIZE， 那么， 当前内存块zend_mm_block就会被放回mm_heap->cache中。 如果内存块没有被放回cache， 则使用下面的代码进行处理：

```
zend_mm_block *mm_block; //要销毁的内存块
zend_mm_block *next_block;
...
next_block = ZEND_MM_BLOCK_AT(mm_block, size);
if (ZEND_MM_IS_FREE_BLOCK(next_block)) {
    zend_mm_remove_from_free_list(heap, (zend_mm_free_block *) next_block);
    size += ZEND_MM_FREE_BLOCK_SIZE(next_block);
}
if (ZEND_MM_PREV_BLOCK_IS_FREE(mm_block)) {
    mm_block = ZEND_MM_PREV_BLOCK(mm_block);
    zend_mm_remove_from_free_list(heap, (zend_mm_free_block *) mm_block);
```

```
    size += ZEND_MM_FREE_BLOCK_SIZE(mm_block);
}
if (ZEND_MM_IS_FIRST_BLOCK(mm_block) &&
    ZEND_MM_IS_GUARD_BLOCK(ZEND_MM_BLOCK_AT(mm_block, size))) {
    zend_mm_del_segment(heap, (zend_mm_segment *) ((char *)mm_block -
ZEND_MM_ALIGNED_SEGMENT_SIZE));
} else {
    ZEND_MM_BLOCK(mm_block, ZEND_MM_FREE_BLOCK, size);
    zend_mm_add_to_free_list(heap, (zend_mm_free_block *) mm_block);
}
```

这段代码逻辑比较清晰，主要是根据当前要销毁的内存块**mm_block**在**zend_mm_heap** 双向链表中所处的位置进行不同的操作。如果下一个节点还是free的内存，则将下一个节点合并; 如果上一相邻节点内存块为free，则合并到上一个节点; 如果只是普通节点，刚使用 **zend_mm_add_to_free_list**或者**zend_mm_del_segment** 进行回收。

就这样，ZendMM将内存块以整理收回到zend_mm_heap的方式，回收到内存池中。 程序使用的所有内存，将在进程结束时统一交还给系统。

在内存的销毁过程中，还涉及到引用计数和垃圾回收（GC），将在下一小节进行讨论。

第四节 垃圾回收

垃圾回收机制是一种动态存储分配方案。它会自动释放程序不再需要的已分配的内存块。 自动回收内存的过程叫垃圾收集。垃圾回收机制可以让程序员不必过分关心程序内存分配，从而将更多的精力投入到业务逻辑。 在现在的流行各种语言当中，垃圾回收机制是新一代语言所共有的特征，如Python、PHP、Eiffel、C#、Ruby等都使用了垃圾回收机制。 虽然垃圾回收是现在比较流行的做法，但是它的年纪已经不小了。早在20世纪60年代MIT开发的Lisp系统中就已经有了它的身影，但是由于当时技术条件不成熟，从而使得垃圾回收机制成了一个看起来很美的技术，直到20世纪90年代Java的出现，垃圾回收机制才被广泛应用。

PHP也在语言层实现了内存的动态管理，这在前面的章节中已经有了详细的说明，内存的动态管理将开发人员从繁琐的内存管理中解救出来。与此配套，PHP也提供了语言层的垃圾回收机制，让程序员不必过分关心程序内存分配。

在PHP5.3版本之前，PHP只有简单的基于引用计数的垃圾回收，当一个变量的引用计数变为0时，PHP将在内存中销毁这个变量，只是这里的垃圾并不能称之为垃圾。 并且PHP在一个生命周期结束后就会释放此进程/线程所点的内容，这种方式决定了PHP在前期不需要过多考虑内存的泄露问题。 但是随着PHP的发展，PHP开发者的增加以及其所承载的业务范围的扩大，在PHP5.3中引入了更加完善的垃圾回收机制。 新的垃圾回收机制解决了无法处理循环的引用内存泄漏问题。PHP5.3中的垃圾回收机制使用了文章引用计数系统中的同步周期回收(Concurrent Cycle Collection in Reference Counted Systems) 中的同步算法。关于这个算法的介绍我们就不再赘述，在PHP的官方文档有图文并茂的介绍：回收周期(Collecting Cycles)。

在本小节，我们从PHP的垃圾回收机制的结构出发，结合其算法介绍PHP5.3垃圾回收机制的实现。

新的垃圾回收

如前面所说，在PHP中，主要的内存管理手段是引用计数，引入垃圾收集机制的目的是为了打破引用

计数中的循环引用，从而防止因为这个而产生的内存泄露。垃圾收集机制基于PHP的动态内存管理而存在。PHP5.3为引入垃圾收集机制，在变量存储的基本结构上有一些变动，如下所示：

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

与PHP5.3之前的版本相比，引用计数字段refcount和是否引用字段is_ref都在其后面添加了__gc以用于新的的垃圾回收机制。在PHP的源码风格中，大量的宏是一个非常鲜明的特点。这些宏相当于一个接口层，它屏蔽了接口层以下的一些底层实现，如，ALLOC_ZVAL宏，这个宏在PHP5.3之前是直接调用PHP的内存管理分配函数emalloc分配内存，所分配的内存大小由变量的类型等大小决定。在引入垃圾回收机制后，ALLOC_ZVAL宏直接采用新的垃圾回收单元结构，所分配的大小都是一样的，全部是zval_gc_info结构体所占内存大小，并且在分配内存后，初始化这个结构体的垃圾回收机制。如下代码：

```
/* The following macroses override macroses from zend_alloc.h */
#undef  ALLOC_ZVAL
#define ALLOC_ZVAL(z)                                           \
    do {                                                        \
        (z) = (zval*)emalloc(sizeof(zval_gc_info));             \
        GC_ZVAL_INIT(z);                                        \
    } while (0)
```

zend_gc.h文件在zend.h的749行被引用：#include “zend_gc.h” 从而替换覆盖了在237行引用的zend_alloc.h文件中的ALLOC_ZVAL等宏 在新的的宏中，关键性的改变是对所分配内存大小和分配内容的改变，在以前纯粹的内存分配中添加了垃圾收集机制的内容，所有的内容都包括在zval_gc_info结构体中：

```
typedef struct _zval_gc_info {
    zval z;
    union {
        gc_root_buffer       *buffered;
        struct _zval_gc_info *next;
    } u;
} zval_gc_info;
```

对于任何一个ZVAL容器存储的变量，分配了一个zval结构，这个结构确保其和以zval变量分配的内存的开始对齐，从而在zval_gc_info类型指针的强制转换时，其可以作为zval使用。在zval字段后面有一个联合体:u。u包括gc_root_buffer结构的buffered字段和zval_gc_info结构的next字段。这两个字段一个是表示垃圾收集机制缓存的根结点，一个是zval_gc_info列表的下一个结点，垃圾收集机制缓存的结点无论是作为根结点，还是列表结点，都可以在这里体现。ALLOC_ZVAL在分配了内存后会调用GC_ZVAL_INIT用来初始化替代了zval的zval_gc_info，它会把zval_gc_info中的成员u的buffered字段设置成NULL，此字段仅在将其放入垃圾回收缓冲区时才会有值，否则会一直是NULL。由于PHP中所有的变量都是以zval变量的形式存在，这里以zval_gc_info替换zval，从而成功实现垃圾收集机制在原有系统中的集成。

PHP的垃圾回收机制在PHP5.3中默认为开启，但是我们可以通过配置文件直接设置为禁用，其对应的配置字段为：zend.enable_gc。在php.ini文件中默认是没有这个字段的，如果我们需要禁用此功能，则在php.ini中添加zend.enable_gc=0或zend.enable_gc=off。除了修改php.ini配置zend.enable_gc，也可以

通过调用gc_enable()/gc_disable()函数来打开/关闭垃圾回收机制。 这些函数的调用效果与修改配置项来打开或关闭垃圾回收机制的效果是一样的。 除了这两个函数PHP提供了gc_collect_cycles()函数可以在根缓冲区还没满时强制执行周期回收。 与垃圾回收机制是否开启在PHP源码中有一些相关的操作和字段。在zend.c文件中有如下代码：

```
static ZEND_INI_MH(OnUpdateGCEnabled) /* {{{ */
{
    OnUpdateBool(entry, new_value, new_value_length, mh_arg1, mh_arg2, mh_arg3,
stage TSRMLS_CC);

    if (GC_G(gc_enabled)) {
        gc_init(TSRMLS_C);
    }

    return SUCCESS;
}
/* }}} */

ZEND_INI_BEGIN()
    ZEND_INI_ENTRY("error_reporting",               NULL,        ZEND_INI_ALL,
OnUpdateErrorReporting)
    STD_ZEND_INI_BOOLEAN("zend.enable_gc",              "1",     ZEND_INI_ALL,
OnUpdateGCEnabled,      gc_enabled,      zend_gc_globals,        gc_globals)
#ifdef ZEND_MULTIBYTE
    STD_ZEND_INI_BOOLEAN("detect_unicode", "1", ZEND_INI_ALL, OnUpdateBool,
detect_unicode, zend_compiler_globals, compiler_globals)
#endif
ZEND_INI_END()
```

zend.enable_gc对应的操作函数为ZEND_INI_MH(OnUpdateGCEnabled)，如果开启了垃圾回收机制， 即GC_G(gc_enabled)为真， 则会调用gc_init函数执行垃圾回收机制的初始化操作。 gc_init函数在zend/zend_gc.c 121行， 此函数会判断是否开启垃圾回收机制， 如果开启， 则初始化整个机制，即直接调用malloc给整个缓存列表分配10000个gc_root_buffer内存空间。 这里的10000是硬编码在代码中的， 以宏GC_ROOT_BUFFER_MAX_ENTRIES存在， 如果需要修改这个值， 则需要修改源码， 重新编译PHP。 gc_init函数在预分配内存后调用gc_reset函数重置整个机制用到的一些全局变量， 如设置gc运行的次数统计(gc_runs)和gc中垃圾的个数(collected)为0， 设置双向链表头结点的上一个结点和下一个结点指向自己等。除了这种提的一些用于垃圾回收机制的全局变量， 还有其它一些使用较多的变量， 部分说明如下：

```
typedef struct _zend_gc_globals {
    zend_bool         gc_enabled;    /* 是否开启垃圾收集机制 */
    zend_bool         gc_active;     /* 是否正在进行 */

    gc_root_buffer    *buf;                /* 预分配的缓冲区数组，默认为
10000 (preallocated arrays of buffers)     */
    gc_root_buffer    roots;               /* 列表的根结点 (list of possible roots
of cycles)  */
    gc_root_buffer    *unused;             /* 没有使用过的缓冲区列表(list of unused
buffers)            */
    gc_root_buffer    *first_unused;       /* 指向第一个没有使用过的缓冲区结点
 (pointer to first unused buffer)     */
    gc_root_buffer    *last_unused;        /* 指向最后一个没有使用过的缓冲区结点，此处
为标记结束用(pointer to last unused buffer)     */

    zval_gc_info      *zval_to_free;       /* 将要释放的zval变量的临时列表
 (temporaryt list of zvals to free)  */
    zval_gc_info      *free_list;          /* 临时变量，需要释放的列表开头 */
```

```
    zval_gc_info      *next_to_free;      /* 临时变量，下一个将要释放的变量位置*/

    zend_uint gc_runs;  /* gc运行的次数统计 */
    zend_uint collected;    /* gc中垃圾的个数 */

    // 省略...
```

}

当我们使用一个unset操作想清除这个变量所占的内存时（可能只是引用计数减一），会从当前符号的哈希表中删除变量名对应的项，在所有的操作执行完后，并对从符号表中删除的项调用一个析构函数，临时变量会调用zval_dtor，一般的变量会调用zval_ptr_dtor。

> 当然我们无法在PHP的函数集中找到unset函数，因为它是一种语言结构。其对应的中间代码为ZEND_UNSET，在Zend/zend_vm_execute.h文件中你可以找到与它相关的实现。

zval_ptr_dtor并不是一个函数，只是一个长得有点像函数的宏。在Zend/zend_variables.h文件中，这个宏指向函数_zval_ptr_dtor。在Zend/zend_execute_API.c 424行，函数相关代码如下：

```
ZEND_API void _zval_ptr_dtor(zval **zval_ptr ZEND_FILE_LINE_DC) /* {{{ */
{
#if DEBUG_ZEND>=2
    printf("Reducing refcount for %x (%x): %d->%d\n", *zval_ptr, zval_ptr,
Z_REFCOUNT_PP(zval_ptr), Z_REFCOUNT_PP(zval_ptr) - 1);
#endif
    Z_DELREF_PP(zval_ptr);
    if (Z_REFCOUNT_PP(zval_ptr) == 0) {
        TSRMLS_FETCH();

        if (*zval_ptr != &EG(uninitialized_zval)) {
            GC_REMOVE_ZVAL_FROM_BUFFER(*zval_ptr);
            zval_dtor(*zval_ptr);
            efree_rel(*zval_ptr);
        }
    } else {
        TSRMLS_FETCH();

        if (Z_REFCOUNT_PP(zval_ptr) == 1) {
            Z_UNSET_ISREF_PP(zval_ptr);
        }

        GC_ZVAL_CHECK_POSSIBLE_ROOT(*zval_ptr);
    }
}
/* }}} */
```

从代码我们可以很清晰的看出这个zval的析构过程，关于引用计数字段做了以下两个操作：

- 如果变量的引用计数为1，即减一后引用计数为0，直接清除变量。如果当前变量如果被缓存，则需要清除缓存
- 如果变量的引用计数大于1，即减一后引用计数大于0，则将变量放入垃圾列表。如果变更存在引用，则去掉其引用。

将变量放入垃圾列表的操作是GC_ZVAL_CHECK_POSSIBLE_ROOT，这也是一个宏，其对应函数gc_zval_check_possible_root，但是此函数仅对数组和对象执行垃圾回收操作。对于数组和对象变量，它会调用gc_zval_possible_root函数。

```
ZEND_API void gc_zval_possible_root(zval *zv TSRMLS_DC)
{
    if (UNEXPECTED(GC_G(free_list) != NULL &&
                   GC_ZVAL_ADDRESS(zv) != NULL &&
                   GC_ZVAL_GET_COLOR(zv) == GC_BLACK) &&
                   (GC_ZVAL_ADDRESS(zv) < GC_G(buf) ||
                    GC_ZVAL_ADDRESS(zv) >= GC_G(last_unused))) {
        /* The given zval is a garbage that is going to be deleted by
         * currently running GC */
        return;
    }

    if (zv->type == IS_OBJECT) {
        GC_ZOBJ_CHECK_POSSIBLE_ROOT(zv);
        return;
    }

    GC_BENCH_INC(zval_possible_root);

    if (GC_ZVAL_GET_COLOR(zv) != GC_PURPLE) {
        GC_ZVAL_SET_PURPLE(zv);

        if (!GC_ZVAL_ADDRESS(zv)) {
            gc_root_buffer *newRoot = GC_G(unused);

            if (newRoot) {
                GC_G(unused) = newRoot->prev;
            } else if (GC_G(first_unused) != GC_G(last_unused)) {
                newRoot = GC_G(first_unused);
                GC_G(first_unused)++;
            } else {
                if (!GC_G(gc_enabled)) {
                    GC_ZVAL_SET_BLACK(zv);
                    return;
                }
                zv->refcount__gc++;
                gc_collect_cycles(TSRMLS_C);
                zv->refcount__gc--;
                newRoot = GC_G(unused);
                if (!newRoot) {
                    return;
                }
                GC_ZVAL_SET_PURPLE(zv);
                GC_G(unused) = newRoot->prev;
            }

            newRoot->next = GC_G(roots).next;
            newRoot->prev = &GC_G(roots);
            GC_G(roots).next->prev = newRoot;
            GC_G(roots).next = newRoot;

            GC_ZVAL_SET_ADDRESS(zv, newRoot);

            newRoot->handle = 0;
            newRoot->u.pz = zv;

            GC_BENCH_INC(zval_buffered);
            GC_BENCH_INC(root_buf_length);
            GC_BENCH_PEAK(root_buf_peak, root_buf_length);
        }
    }
}
```

在前面说到gc_zval_check_possible_root函数仅对数组和对象执行垃圾回收操作，然而在gc_zval_possible_root函数中，针对对象类型的变量会去调用GC_ZOBJ_CHECK_POSSIBLE_ROOT宏。而对于其它的可用于垃圾回收的机制的变量类型其调用过程如下：

- 检查zval结点信息是否已经放入到结点缓冲区，如果已经放入到结点缓冲区，则直接返回，这样可以优化其性能。然后处理对象结点，直接返回，不再执行后面的操作
- 判断结点是否已经被标记为紫色，如果为紫色则不再添加到结点缓冲区，此处在于保证一个结点只执行一次添加到缓冲区的操作。
- 将结点的颜色标记为紫色，表示此结点已经添加到缓冲区，下次不用再做添加
- 找出新的结点的位置，如果缓冲区满了，则执行垃圾回收操作。
- 将新的结点添加到缓冲区所在的双向链表。

在gc_zval_possible_root函数中，当缓冲区满时，程序调用gc_collect_cycles函数，执行垃圾回收操作。其中最关键的几步就是：

- 第628行 此处为其官方文档中算法的步骤 B，算法使用深度优先搜索查找所有可能的根，找到后将每个变量容器中的引用计数减1，为确保不会对同一个变量容器减两次“1”，用灰色标记已减过1的。
- 第629行 这是算法的步骤 C，算法再一次对每个根节点使用深度优先搜索，检查每个变量容器的引用计数。如果引用计数是 0，变量容器用白色来标记。如果引用次数大于0，则恢复在这个点上使用深度优先搜索而将引用计数减1的操作（即引用计数加1），然后将它们重新用黑色标记。
- 第630行 算法的最后一步 D，算法遍历根缓冲区以从那里删除变量容器根(zval roots)，同时，检查是否有在上一步中被白色标记的变量容器。每个被白色标记的变量容器都被清除。在[gc_collect_cycles() -> gc_collect_roots() -> zval_collect_white()]中我们可以看到，对于白色标记的结点会被添加到全局变量zval_to_free列表中。此列表在后面的操作中有用到。

PHP的垃圾回收机制在执行过程中以四种颜色标记状态。

- GC_WHITE 白色表示垃圾
- GC_PURPLE 紫色表示已放入缓冲区
- GC_GREY 灰色表示已经进行了一次refcount的减一操作
- GC_BLACK 黑色是默认颜色，正常

相关的标记以及操作代码如下：

```
#define GC_COLOR  0x03

#define GC_BLACK  0x00
#define GC_WHITE  0x01
#define GC_GREY   0x02
#define GC_PURPLE 0x03

#define GC_ADDRESS(v) \
    ((gc_root_buffer*)(((zend_uintptr_t)(v)) & ~GC_COLOR))
#define GC_SET_ADDRESS(v, a) \
    (v) = ((gc_root_buffer*)((((zend_uintptr_t)(v)) & GC_COLOR) |
((zend_uintptr_t)(a))))
#define GC_GET_COLOR(v) \
    (((zend_uintptr_t)(v)) & GC_COLOR)
#define GC_SET_COLOR(v, c) \
    (v) = ((gc_root_buffer*)((((zend_uintptr_t)(v)) & ~GC_COLOR) | (c)))
#define GC_SET_BLACK(v) \
    (v) = ((gc_root_buffer*)(((zend_uintptr_t)(v)) & ~GC_COLOR))
#define GC_SET_PURPLE(v) \
```

```
    (v) = ((gc_root_buffer*)(((zend_uintptr_t)(v)) | GC_PURPLE))
```

以上的这种以位来标记状态的方式在PHP的源码中使用频率较高，如内存管理等都有用到，这是一种比较高效及节省的方案。但是在我们做数据库设计时可能对于字段不能使用这种方式，应该是以一种更加直观，更加具有可读性的方式实现。

第五节 内存管理中的缓存

在维基百科中有这样一段描述：**凡是位于速度相差较大的两种硬件之间的，用于协调两者数据传输速度差异的结构，均可称之为Cache。**从最初始的处理器与内存间的Cache开始，都是为了让数据访问的速度适应CPU的处理速度，其基于的原理是内存中“程序执行与数据访问的局域性行为”。同样PHP内存管理中的缓存也是基于“程序执行与数据访问的局域性行为”的原理。引入缓存，就是为了减少小块内存块的查询次数，为最近访问的数据提供更快的访问方式。

PHP将缓存添加到内存管理机制中做了如下一些操作：

- 标识缓存和缓存的大小限制，即何时使用缓存，在某些情况下可以以最少的修改禁用掉缓存
- 缓存的存储结构，即缓存的存放位置、结构和存放的逻辑
- 初始化缓存
- 获取缓存中内容
- 写入缓存
- 释放缓存或者清空缓存列表

首先我们看标识缓存和缓存的大小限制，在PHP内核中，是否使用缓存的标识是宏ZEND_MM_CACHE（Zend/zend_alloc.c 400行），缓存的大小限制与size_t结构大小有关，假设size_t占4位，则默认情况下，PHP内核给PHP内存管理的限制是128K(32 * 4 * 1024)。如下所示代码：

```
#define ZEND_MM_NUM_BUCKETS (sizeof(size_t) << 3)

#define ZEND_MM_CACHE 1
#define ZEND_MM_CACHE_SIZE (ZEND_MM_NUM_BUCKETS * 4 * 1024)
```

如果在某些应用下需要禁用缓存，则将ZEND_MM_CACHE宏设置为0，重新编译PHP即可。为了实现这个一处修改所有地方都生效的功能，则在每个需要调用缓存的地方在编译时都会判断ZEND_MM_CACHE是否定义为1。

如果我们启用了缓存，则在堆层结构中增加了两个字段：

```
struct _zend_mm_heap {

#if ZEND_MM_CACHE
    unsigned int        cached; //  已缓存元素使用内存的总大小
    zend_mm_free_block *cache[ZEND_MM_NUM_BUCKETS]; //  存放被缓存的块
#endif
```

如上所示，cached表示已缓存元素使用内存的总大小，zend_mm_free_block结构的数组装载被缓存的块。在初始化内存管理时，会调用zend_mm_init函数。在这个函数中，当缓存启用时会初始化上面所说的两个字段，如下所示：

```
#if ZEND_MM_CACHE
    heap->cached = 0;
    memset(heap->cache, 0, sizeof(heap->cache));
#endif
```

程序会初始化已缓存元素的总大小为0，并给存放缓存块的数组分配内存。初始化之后，如果外部调用需要PHP内核分配内存，此时可能会调用缓存，之所以是可能是因为它有一个前提条件，即所有的缓存都只用于小于的内存块的申请。所谓小块的内存块是其真实大小小于ZEND_MM_MAX_SMALL_SIZE(272)的。比如，在缓存启用的情况下，我们申请一个100Byte的内存块，则PHP内核会首先判断其真实大小，并进入小块内存分配的流程，在此流程中程序会先判断对应大小的块索引是否存在，如果存在则直接从缓存中返回，否则继续走常规的分配流程。

当用户释放内存块空间时，程序最终会调用_zend_mm_free_int函数。在此函数中，如果启用了缓存并且所释放的是小块内存，并且已分配的缓存大小小于缓存限制大小时，程序会将释放的块放到缓存列表中。如下代码

```
#if ZEND_MM_CACHE
    if (EXPECTED(ZEND_MM_SMALL_SIZE(size)) && EXPECTED(heap->cached <
ZEND_MM_CACHE_SIZE)) {
        size_t index = ZEND_MM_BUCKET_INDEX(size);
        zend_mm_free_block **cache = &heap->cache[index];

        ((zend_mm_free_block*)mm_block)->prev_free_block = *cache;
        *cache = (zend_mm_free_block*)mm_block;
        heap->cached += size;
        ZEND_MM_SET_MAGIC(mm_block, MEM_BLOCK_CACHED);
#if ZEND_MM_CACHE_STAT
        if (++heap->cache_stat[index].count > heap-
>cache_stat[index].max_count) {
            heap->cache_stat[index].max_count = heap->cache_stat[index].count;
        }
#endif
        return;
    }
#endif
```

当堆的内存溢出时，程序会调用zend_mm_free_cache释放缓存中。整个释放的过程是一个遍历数组，对于每个数组的元素程序都遍历其所在链表中在自己之前的元素，执行合并内存操作，减少堆结构中缓存计量数字。具体实现参见Zend/zend_alloc.c的909行。

在上面的一些零碎的代码块中我们有看到在ZEND_MM_CACHE宏出现时经常会出现ZEND_MM_CACHE_STAT宏。这个宏是标记是否启用缓存统计功能，默认情况下为不启用。缓存统计功能也有对应的存储结构，在分配，释放缓存中的值时，缓存统计功能都会有相应的实现。

第六节 写时复制（Copy On Write）

在开始之前，我们可以先看一段简单的代码：

```
<?php    //例一
    $foo = 1;
    $bar = $foo;
    echo $foo + $bar;
?>
```

执行这段代码，会打印出数字2。从内存的角度来分析一下这段代码“可能”是这样执行的：分配一块内存给foo变量，里面存储一个1；再分配一块内存给bar变量，也存一个1，最后计算出结果输出。事实上，我们发现foo和bar变量因为值相同，完全可以使用同一块内存，这样，内存的使用就节省了一个1，并且，还省去了分配内存和管理内存地址的计算开销。没错，很多涉及到内存管理的系统，都实现了这种相同值共享内存的策略：**写时复制**

很多时候，我们会因为一些术语而对其概念产生莫测高深的恐惧，而其实，他们的基本原理往往非常简单。本小节将介绍PHP中写时复制这种策略的实现：

> 写时复制（Copy on Write，也缩写为COW)的应用场景非常多，比如Linux中对进程复制中内存使用的优化，在各种编程语言中，如C++的STL等等中均有类似的应用。COW是常用的优化手段，可以归类于：资源延迟分配。只有在真正需要使用资源时才占用资源，写时复制通常能减少资源的占用。

注：为节省篇幅，下文将统一使用COW来表示“写时复制”；

推迟内存复制的优化

正如前面所说，PHP中的COW可以简单描述为：如果通过赋值的方式赋值给变量时不会申请新内存来存放 新变量所保存的值，而是简单的通过一个计数器来共用内存，只有在其中的一个引用指向变量的 值发生变化时才申请新空间来保存值内容以减少对内存的占用。在很多场景下PHP都COW进行内存的优化。比如：变量的多次赋值、函数参数传递，并在函数体内修改实参等。

下面让我们看一个查看内存的例子，可以更容易看到COW在内存使用优化方面的明显作用：

```
<?php  //例二
$j = 1;
        var_dump(memory_get_usage());

$tipi = array_fill(0, 100000, 'php-internal');
        var_dump(memory_get_usage());

$tipi_copy = $tipi;
        var_dump(memory_get_usage());

foreach($tipi_copy as $i){
    $j += count($i);
}
        var_dump(memory_get_usage());

//-----执行结果-----
$ php t.php
int(630904)
int(10479840)
int(10479944)
int(10480040)
```

上面的代码比较典型的突出了COW的作用，在数组变量`$tipi`被赋值给`$tipi_copy`时，内存的使用并没有立刻增加一半，在循环遍历数`$tipi_copy`时也没有发生显著变化，在这里`$tipi_copy`和`$tipi`变量的数据共同指向同一块内存，而没有复制。

也就是说，即使我们不使用引用，一个变量被赋值后，只要我们不改变变量的值，也不会新申请内存用来存放数据。据此我们很容易就可以想到一些COW可以非常有效的控制内存使用的场景：只是使用变量进行计算而很少对其进行修改操作，如函数参数的传递，大数组的复制等等等不需要改变变量值的情形。

复制分离变化的值

多个相同值的变量共用同一块内存的确节省了内存空间，但变量的值是会发生变化的，如果在上面的例子中，指向同一内存的值发生了变化（或者可能发生变化），就需要将变化的值“分离”出去，这个“分离”的操作，就是“复制”。

在PHP中，Zend引擎为了区别同一个zval地址是否被多个变量共享，引入了ref_count和is_ref两个变量进行标识：

ref_count和**is_ref**是定义于zval结构体中（见第一章第一小节）
is_ref标识是不是用户使用 & 的强制引用；
ref_count是引用计数，用于标识此zval被多少个变量引用，即COW的自动引用，为0时会被销毁；
关于这两个变量的更多内容，跳转阅读：第三章第六节：变量的赋值和销毁的实现。
注：由此可见，$a=$b; 与 $a=&$b; 在PHP对内存的使用上没有区别（值不变化时）；

下面我们把**例二**稍做变化：如果$copy的值发生了变化，会发生什么？：

```
<?php //例三
//$tipi = array_fill(0, 3, 'php-internal');
//这里不再使用array_fill来填充，为什么？
$tipi[0] = 'php-internal';
$tipi[1] = 'php-internal';
$tipi[2] = 'php-internal';
var_dump(memory_get_usage());

$copy = $tipi;
xdebug_debug_zval('tipi', 'copy');
var_dump(memory_get_usage());

$copy[0] = 'php-internal';
xdebug_debug_zval('tipi', 'copy');
var_dump(memory_get_usage());

//-----执行结果-----
$ php t.php
int(629384)
tipi: (refcount=2, is_ref=0)=array (0 => (refcount=1, is_ref=0)='php-internal',
                                    1 => (refcount=1, is_ref=0)='php-internal',
                                    2 => (refcount=1, is_ref=0)='php-internal')
copy: (refcount=2, is_ref=0)=array (0 => (refcount=1, is_ref=0)='php-internal',
                                    1 => (refcount=1, is_ref=0)='php-internal',
                                    2 => (refcount=1, is_ref=0)='php-internal')
int(629512)
tipi: (refcount=1, is_ref=0)=array (0 => (refcount=1, is_ref=0)='php-internal',
                                    1 => (refcount=2, is_ref=0)='php-internal',
                                    2 => (refcount=2, is_ref=0)='php-internal')
copy: (refcount=1, is_ref=0)=array (0 => (refcount=1, is_ref=0)='php-internal',
                                    1 => (refcount=2, is_ref=0)='php-internal',
                                    2 => (refcount=2, is_ref=0)='php-internal')
```

```
int(630088)
```

在这个例子中，我们可以发现以下特点：

1. $copy = $tipi；这种基本的赋值操作会触发COW的内存“共享”，不会产生内存复制；
2. COW的粒度为zval结构，由PHP中变量全部基于zval，所以COW的作用范围是全部的变量，而对于zval结构体组成的集合（如数组和对象等），在需要复制内存时，将复杂对象分解为最小粒度来处理。这样可以使内存中复杂对象中某一部分做修改时，不必将该对象的所有元素全部“分离复制”出一份内存拷贝；

   array_fill()填充数组时也采用了COW的策略，可能会影响对本例的演示，感兴趣的读者可以 阅读：$PHP_SRC/ext/standard/array.c中PHP_FUNCTION(array_fill)的实现。

   xdebug_debug_zval()是xdebug扩展中的一个函数，用于输出变量在zend内部的引用信息。如果你没有安装xdebug扩展，也可以使用debug_zval_dump()来代替。参考：http://www.php.net/manual/zh/function.debug-zval-dump.php

实现写时复制

看完上面的三个例子，相信大家也可以了解到PHP中COW的实现原理：PHP中的COW基于引用计数**ref_count**和**is_ref**实现，多一个变量指针，就将**ref_count**加1，反之减去1，减到0就销毁；同理，多一个强制引用&,就将**is_ref**加1，反之减去1。

这里有一个比较典型的例子：

```
<?php  //例四
    $foo = 1;
    xdebug_debug_zval('foo');
    $bar = $foo;
    xdebug_debug_zval('foo');
    $bar = 2;
    xdebug_debug_zval('foo');
?>
//-----执行结果-----
foo: (refcount=1, is_ref=0)=1
foo: (refcount=2, is_ref=0)=1
foo: (refcount=1, is_ref=0)=1
```

经过前面对变量章节的介绍，我们知道当$foo被赋值时，$foo变量的值的只由$foo变量指向。当$foo的值被赋给$bar时，PHP并没有将内存复制一份交给$bar，而是把$foo和$bar指向同一个地址。同时引用计数增加1，也就是新的2。随后，我们更改了$bar的值，这时如果直接需该$bar变量指向的内存，则$foo的值也会跟着改变。这不是我们想要的结果。于是，PHP内核将内存复制出来一份，并将其值更新为赋值的：2（这个操作也称为变量分离操作），同时原$foo变量指向的内存只有$foo指向，所以引用计数更新为：refcount=1。

看上去很简单，但由于**&**运算符的存在，实际的情形要复杂的多。见下面的例子：

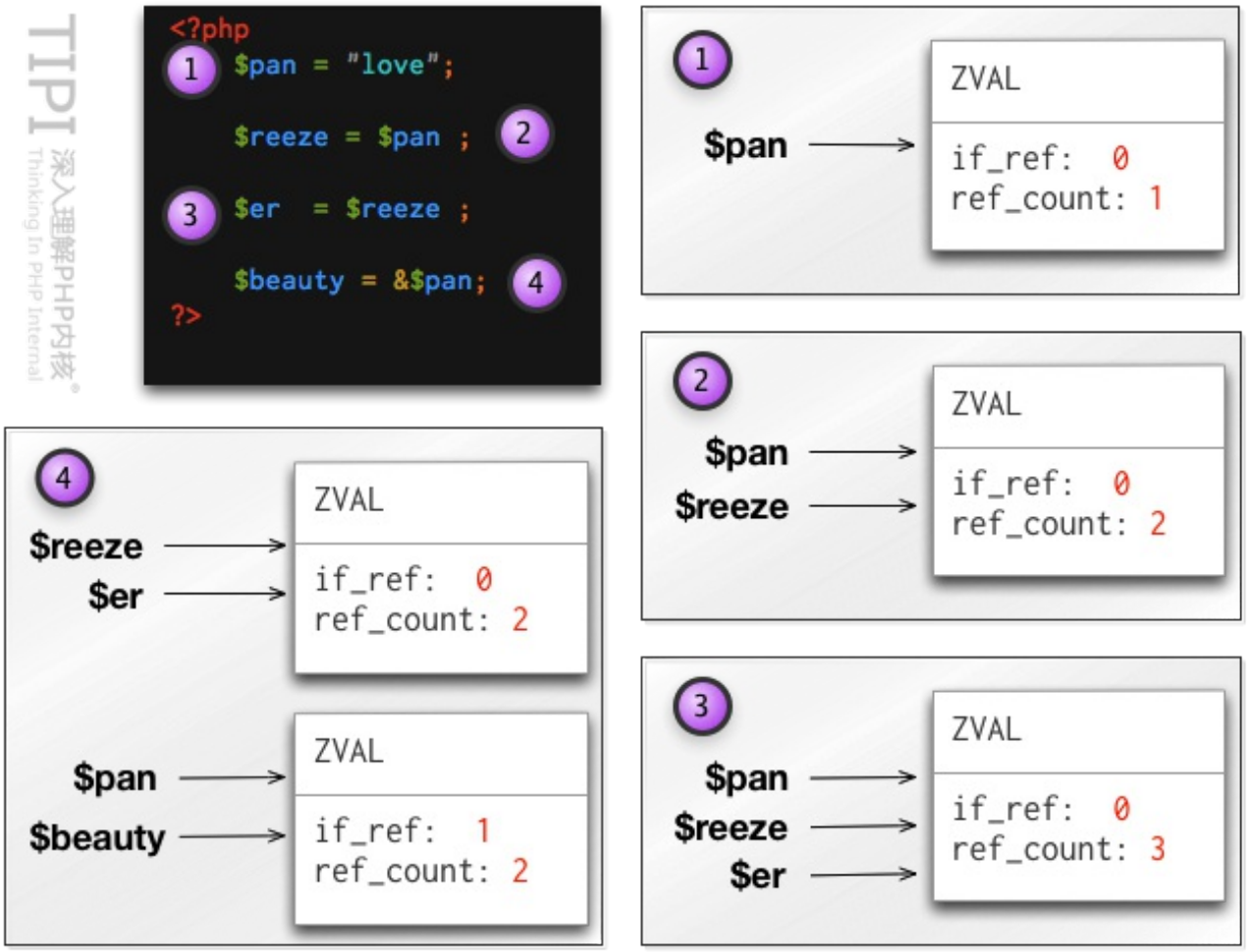


图6.6 &操作符引起的内存复制分离

从这个例子可以看出PHP对&运算符的一个容易出问题的处理：当 $beauty=&$pan; 时， 两个变量本质上都变成了引用类型， 导致看上去的普通变量$pan, 在某些内部处理中与&$pan行为相同， 尤其是在数组元素中使用引用变量， 很容易引发问题。（见最后的例子）

PHP的大多数工作都是进行文本处理， 而变量是载体， 不同类型的变量的使用贯穿着PHP的生命周期， 变量的COW策略也就体现了Zend引擎对变量及其内存处理， 具体可以参阅源码文件相关的内容：

```
Zend/zend_execute.c
========================================
    zend_assign_to_variable_reference();
    zend_assign_to_variable();
    zend_assign_to_object();
    zend_assign_to_variable();

//以及下列宏定义的使用
Zend/zend.h
========================================
    #define Z_REFCOUNT(z)             Z_REFCOUNT_P(&(z))
    #define Z_SET_REFCOUNT(z, rc)         Z_SET_REFCOUNT_P(&(z), rc)
    #define Z_ADDREF(z)           Z_ADDREF_P(&(z))
    #define Z_DELREF(z)           Z_DELREF_P(&(z))
    #define Z_ISREF(z)            Z_ISREF_P(&(z))
    #define Z_SET_ISREF(z)            Z_SET_ISREF_P(&(z))
    #define Z_UNSET_ISREF(z)          Z_UNSET_ISREF_P(&(z))
    #define Z_SET_ISREF_TO(z, isref)      Z_SET_ISREF_TO_P(&(z), isref)
```

最后，请慎用引用&

引用和前面提到的变量的引用计数和PHP中的引用并不是同一个东西，引用和C语言中的指针的类似，他们都可以通过不同的标示访问到同样的内容，但是PHP的引用则只是简单的变量别名，没有C指令的灵活性和限制。

PHP中有非常多让人觉得意外的行为，有些因为历史原因，不能破坏兼容性而选择 暂时不修复，或者有的使用场景比较少。在PHP中只能尽量的避开这些陷阱。例如下面这个例子。

由于引用操作符会导致PHP的COW策略优化，所以使用引用也需要对引用的行为有明确的认识才不至于误用，避免带来一些比较难以理解的的Bug。如果您认为您已经足够了解了PHP中的引用，可以尝试解释下面这个例子：

```
<?php
$foo['love'] = 1;
$bar  = &$foo['love'];
$tipi = $foo;
$tipi['love'] = '2';
echo $foo['love'];
```

这个例子最后会输出 2，大家会非常惊讶于$tipi怎么会影响到$foo，`$bar`变量的引用操作，将$foo['love']污染变成了引用，从而Zend没有 对`$tipi['love']`的修改产生内存的复制分离。

第七节 小结

我们平常在讨论算法时会讲到空间复杂度，一般来说这里的空间复杂度是指所占内存的大小。这就突显了内存管理在我们编程过程中的重要性。从某种意见上来说内存也属于缓存的一种，它的作用就是将硬盘或其它较慢存储介质中的数据更快的提供给处理器（或处理器缓存）。

PHP内核以接口的方式提供了内存管理，将内存管理对PHP内核的其它模块透明，从而提供更加高效的内存管理，减少内存碎片。在本章，我们从内存管理概述开始，介绍了内存管理的意义及必要性，然后从PHP内存管理的整体结构、内存管理宏的具体实现等方面做了详细的说明。并在第四小节详细介绍了PHP5.3才引入的垃圾收集机制，之后介绍了内存管理中的缓存优化，虽然PHP有实现缓存的统计功能，但是在默认情况下是关闭的，最后我们以写时复制这样一个特性结束了本章。

虽然PHP内核提供了内存管理机制，但是我们也可以通过环境变量设置绕过内存管理直接使用某些系统级的内存管理函数。这适用于调试或一些特定的应用场景，一般情况下，我们还是使用PHP内核替我们实现的内存管理吧。

下一章，我们将介绍PHP的虚拟机。

第七章 Zend虚拟机

在前面的章节中，我们了解到一个PHP文件在服务器端的执行过程包括以下两个大的过程：

1. 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。

在第二步中，词法分析、语法分析，编译中间代码，执行中间代码等各个部分统称为Zend虚拟机。 与Java、C#等编译型语言相比，PHP少了一个手动编译的过程，它们无需编译即可运行，我们称其为解释性语言。 Java有自己的Java虚拟机，它在多个平台上实现统一语言； C#有自己的.NET虚拟机，它在单一平台实现多种语言； PHP跟他们一样，也有属于自己的Zend虚拟机。它们在本质是相同的，它们都是抽象的计算机。 这些虚拟机都是在某种较底层的语言上抽象出另外一种语言，有自己的指令集，有自己的内存管理体系。 它们最终都会将抽象级别较高的语言实现转化为抽象级别较低的语言实现， 并且实现其它辅助功能，如内存管理，垃圾回收等机制， 以减少程序员在具体实现上的工作，从而可以将更多的时间和精力投入到业务逻辑中。 从抽象层次看，Zend虚拟机比Java等语言更高级一些， 这里的高级不是说功能更强大或效率更高， 简单点说，Zend虚拟机离真正的机器实现更远一些。 最近这些年，语言的发展只是不断的抽象，不断的远离机器，没有根本性的变化。

本章，我们从虚拟机的前世今生讲起，叙述Zend虚拟机的实现原理，关键的数据结构， 并其中穿插一个关于语法实现的示例和源码加密解密的过程说明。

第一节 Zend虚拟机概述

在wiki中虚拟机的定义是： 虚拟机（Virtual Machine）， 在计算机科学中的体系结构里，是指一种特殊的软件， 他可以在计算机平台和终端用户之间创建一种环境，而终端用户则是基于这个软件所创建的环境来操作软件。 在计算机科学中，虚拟机是指可以像真实机器一样运行程序的计算机的软件实现。

虚拟机是一种抽象的计算机，它有自己的指令集，有自己的内存管理体系。 在此类虚拟机上实现的语言比较低抽象层次的语言更加明了，更加简单易学。

Zend虚拟机核心实现代码

为了方便读者对Zend引擎的实现有个全面的感觉，下面列出涉及到Zend引擎实现的核心代码文件功能参考。

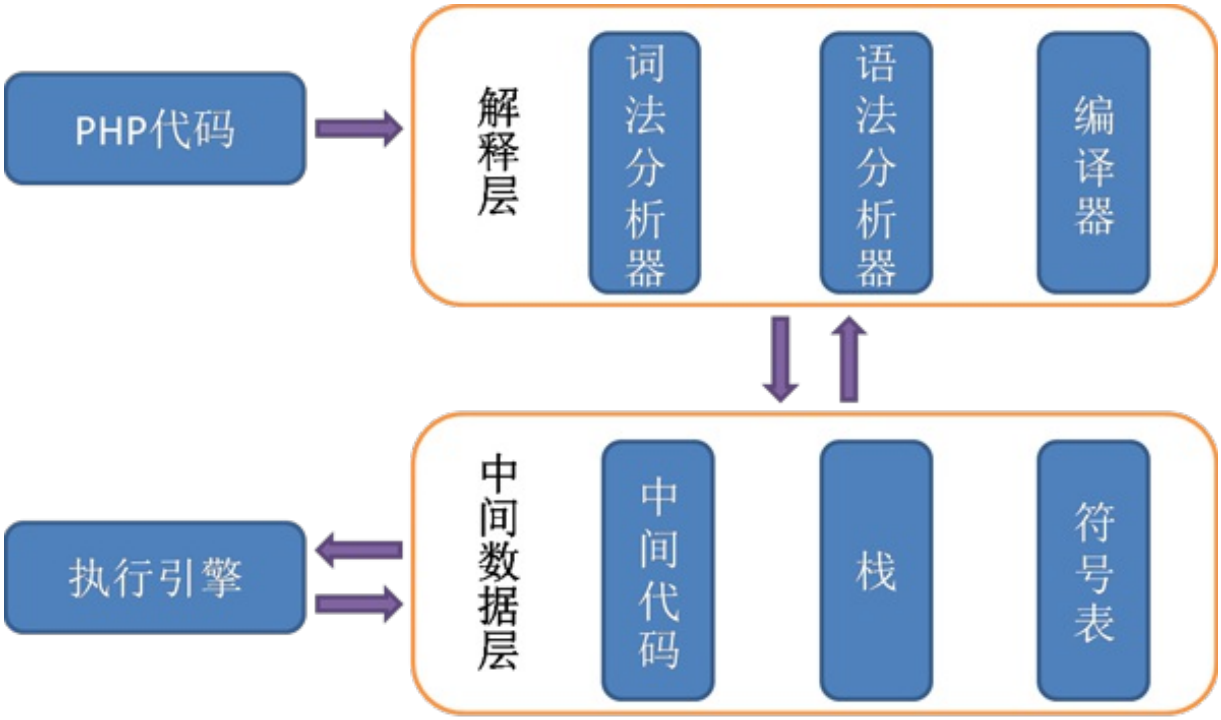
Zend引擎的核心文件都在$PHP_SRC/Zend/目录下面。不过最为核心的文件只有如下几个：

1. PHP语法实现
    - Zend/zend_language_scanner.l
    - Zend/zend_language_parser.y
2. Opcode编译
    - Zend/zend_compile.c

3. 执行引擎
    - Zend/zend_vm_*
    - Zend/zend_execute.c

Zend虚拟机体系结构

从概念层将Zend虚拟机的实现进行抽象，我们可以将Zend虚拟机的体系结构分为：解释层、执行引擎、中间数据层，如图7.1所示：



Zend 虚拟机体系结构图

图7.1 Zend虚拟机体系结构图

当一段PHP代码进入Zend虚拟机，它会被执行两步操作:编译和执行。对于一个解释性语言来说，这是一个创造性的举动，但是，现在的实现并不彻底。现在当PHP代码进入Zend虚拟机后，它虽然会被执行这两步操作，但是这两步操作对于一个常规的执行过程来说却是连续的，也就是说它并没有转变成和Java这种编译型语言一样：生成一个中间文件存放编译后的结果。如果每次执行这样的操作，对于PHP脚本的性能来说是一个极大的损失。虽然有类似于APC，eAccelerator等缓存解决方案。但是其本质上是没有变化的，并且不能将两个步骤分离，各自发展壮大。

解释层

解释层是Zend虚拟机执行编译过程的位置。它包括词法解析、语法解析和编译生成中间代码三个部分。词法分析就是将我们要执行的PHP源文件，去掉空格，去掉注释，切分为一个个的标记(token)，并且处理程序的层级结构(hierarchical structure)。

语法分析就是将接受的标记(token)序列，根据定义的语法规则，来执行一些动作，Zend虚拟机现在使用的Bison使用巴科斯范式(BNF)来描述语法。编译生成中间代码是根据语法解析的结果对照Zend虚拟机制定的opcode生成中间代码，在PHP5.3.1中，Zend虚拟机支持135条指令（见

Zend/zend_vm_opcodes.h文件）， 无论是简单的输出语句还是程序复杂的递归调用，Zend虚拟机最终都会将所有我们编写的PHP代码转化成这135条指令的序列， 之后在执行引擎中按顺序执行。

中间数据层

当Zend虚拟机执行一个PHP代码时，它需要内存来存储许多东西， 比如，中间代码，PHP自带的函数列表，用户定义的函数列表，PHP自带的类，用户自定义的类， 常量，程序创建的对象，传递给函数或方法的参数，返回值，局部变量以及一些运算的中间结果等。 我们把这些所有的存放数据的地方称为中间数据层。

如果PHP以mod扩展的方式依附于Apache2服务器运行，中间数据层的部分数据可能会被多个线程共享，如果PHP自带的函数列表等。 如果只考虑单个进程的方式，当一个进程被创建时它就会被加载PHP自带的各种函数列表，类列表，常量列表等。 当解释层将PHP代码编译完成后，各种用户自定义的函数，类或常量会添加到之前的列表中， 只是这些函数在其自身的结构中某些字段的赋值是不一样的。

当执行引擎执行生成的中间代码时，会在Zend虚拟机的栈中添加一个新的执行中间数据结构（zend_execute_data）， 它包括当前执行过程的活动符号列表的快照、一些局部变量等。

执行引擎

Zend虚拟机的执行引擎是一个非常简单的实现，它只是依据中间代码序列（EX(opline)）,一步一步调用对应的方法执行。 在执行引擎中没并有类似于PC寄存器一样的变量存放下一条指令，当Zend虚拟机执行到某条指令时，当它所有的任务都执行完了， 这条指令会自己调用下一条指令，即将序列的指针向前移动一个位置，从而执行下一条指令，并且在最后执行return语句，如此反复。 这在本质上是一个函数嵌套调用。

回到开头的问题， PHP通过词法分析、语法分析和中间代码生成三个步骤后，PHP文件就会被解析成PHP的中间代码opcode。 生成的中间代码与实际的PHP代码之间并没有完全的一一对应关系。只是针对用户所给的PHP代码和PHP的语法规则和一些内部约定生成中间代码， 并且这些中间代码还需要依靠一些全局变量中转数据和关联。至于生成的中间代码的执行过程是依据中间代码的顺利， 依赖于执行过程中的全局变量，一步步执行。当然，在遇到一些函数跳转也会发生偏移，但是最终还是会回到偏移点。

第二节 语法的实现

世上没有无缘无故的爱，也没有无缘无故的恨。

语言从广义上来讲是人们进行沟通交流的各种表达符号。每种语言都有专属于自己的符号，表达方式和规则。 就编程语言来说，它也是由特定的符号，特定的表达方式和规则组成。 语言的作用是沟通，不管是自然语言，还是编程语言，它们的区别在于自然语言是人与人之间沟通的工具， 而编程语言是人与机器之间的沟通渠道。相对于自然语言， 编程语言的历史还非常短， 虽然编程语言是站在历史巨人的基础上创建的，但是它还很小，还是一个小孩。 它只能按编程人员所给的指令翻译成对应的机器可以识别的语言。它就相当于一个转化工具， 将人们的知识或者业务逻辑转化成机器码（机器的语言）， 让其执行对应的的操作。 而这些指令是一些规则， 一些约定， 这些规则约定都是由编程语言来处理。

就PHP语言来说，它也是一组符合一定规则的约定的指令。 在编程人员将自己的想法以PHP语言实现

后，通过PHP的虚拟机将这些PHP指令转变成C语言（可以理解为更底层的一种指令集）指令，而C语言又会转变成汇编语言，最后汇编语言将根据处理器的规则转变成机器码执行。这是一个更高层次抽象的不断具体化，不断细化的过程。

在这一章，我们讨论PHP虚拟机是如何将PHP语言转化成C语言。从一种语言到另一种语言的转化称之为编译，这两种语言分别可以称之为源语言和目标语言。这种编译过程通过发生在目标语言比源语言更低级（或者说更底层）。语言转化的编译过程是由编译器来完成，编码器通常被分为一系列的过程：词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成等。前面几个阶段（词法分析、语法分析和语义分析）的作用是分析源程序，我们可以称之为编译器的前端。后面的几个阶段（中间代码生成、代码优化和目标代码生成）的作用是构造目标程序，我们可以称之为编译器的后端。一种语言被称为编译类语言，一般是由于在程序执行之前有一个翻译的过程，其中关键点是有一个形式上完全不同的等价程序生成。而PHP之所以被称为解释类语言，就是因为并没有这样的一个程序生成，它生成的是中间代码，这只是PHP的一种内部数据结构。

在本章我们会介绍PHP编译器的前端的两个阶段，语法分析、语法分析；后端的一个阶段，中间代码生成。在第一节我们介绍PHP的词法分析过程及其用到的工具re2c，第二节我们介绍在词法分析后的语法分析过程，第三节我们以PHP的一个简单语法实现作为本章的结束。

词法解析

在前面我们提到语言转化的编译过程一般分为词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成等六个阶段。不管是编译型语言还是解释型语言，扫描（词法分析）总是将程序转化成目标语言的第一步。词法分析的作用就是将整个源程序分解成一个一个的单词，这样做可以在一定程度上减少后面分析工作需要处理的个体数量，为语法分析等做准备。除了拆分工作，更多的时候它还承担着清洗源程序的过程，比如清除空格，清除注释等。词法分析作为编译过程的第一步，在业界已经有多种成熟工具，如PHP在开始使用的是Flex，之后改为re2c，MySQL的词法分析使用的Flex，除此之外还有作为UNIX系统标准词法分析器的Lex等。这些工具都会读进一个代表词法分析器规则的输入字符串流，然后输出以C语言实做的词法分析器源代码。这里我们只介绍PHP的现版词法分析器，re2c。

re2c是一个扫描器制作工具，可以创建非常快速灵活的扫描器。它可以产生高效代码，基于C语言，可以支持C/C++代码。与其它类似的扫描器不同，它偏重于为正则表达式产生高效代码（和他的名字一样）。因此，这比传统的词法分析器有更广泛的应用范围。你可以在sourceforge.net获取源码。

PHP在最开始的词法解析器是使用的是Flex，后来改为使用re2c。在源码目录下的Zend/zend_language_scanner.l 文件是re2c的规则文件，如果需要修改该规则文件需要安装re2c才能重新编译，生成新的规则文件。

re2c调用方式：

```
re2c [-bdefFghisuvVw1] [-o output] [-c [-t header]] file
```

我们通过一个简单的例子来看下re2c。如下是一个简单的扫描器，它的作用是判断所给的字符串是数字/小写字母/大小字母。当然，这里没有做一些输入错误判断等异常操作处理。示例如下：

```
#include <stdio.h>

char *scan(char *p){
```

```
#define YYCTYPE char
#define YYCURSOR p
#define YYLIMIT p
#define YYMARKER q
#define YYFILL(n)
    /*!re2c
      [0-9]+ {return "number";}
      [a-z]+ {return "lower";}
      [A-Z]+ {return "upper";}
      [^] {return "unkown";}
     */
}

int main(int argc, char* argv[])
{
    printf("%s\n", scan(argv[1]));

    return 0;
}
```

如果你是在ubuntu环境下，可以执行下面的命令生成可执行文件。

```
re2c -o a.c a.l
gcc a.c -o a
chmod +x a
./a 1000
```

此时程序会输出number。

我们解释一下我们用到的几个re2c约定的宏。

- YYCTYPE 用于保存输入符号的类型，通常为char型和unsigned char型
- YYCURSOR 指向当前输入标记，-当开始时，它指向当前标记的第一个字符，当结束时，它指向下一个标记的第一个字符
- YYFILL(n) 当生成的代码需要重新加载缓存的标记时，则会调用YYFILL(n)。
- YYLIMIT 缓存的最后一个字符，生成的代码会反复比较YYCURSOR和YYLIMIT，以确定是否需要重新填充缓冲区。

参照如上几个标识的说明，可以较清楚的理解生成的a.c文件，当然，re2c不会仅仅只有上面代码所显示的标记，这只是一个简单示例，更多的标识说明和帮助信息请移步 re2c帮助文档：http://re2c.org/manual.html。

我们回过头来看PHP的词法规则文件zend_language_scanner.l。你会发现前面的简单示例与它最大的区别在于每个规则前面都会有一个条件表达式。

> NOTE re2c中条件表达式相关的宏为YYSETCONDITION和YYGETCONDITION，分别表示设置条件范围和获取条件范围。在PHP的词法规则中共有10种，其全部在zend_language_scanner_def.h文件中。此文件并非手写，而是re2c自动生成的。如果需要生成和使用条件表达式，在编译成c时需要添加-c 和-t参数。

在PHP的词法解析中，它有一个全局变量:language_scanner_globals，此变量为一结构体，记录当前re2c解析的状态，文件信息，解析过程信息等。它在zend_language_scanner.l文件中直接定义如下：

```
#ifdef ZTS
```

```
ZEND_API ts_rsrc_id language_scanner_globals_id;
#else
ZEND_API zend_php_scanner_globals language_scanner_globals;
#endif
```

在zend_language_scanner.l文件中写的C代码在使用re2c生成C代码时会直接复制到新生成的C代码文件中。 这个变量贯穿了PHP词法解析的全过程， 并且一些re2c的实现也依赖于此， 比如前面说到的条件表达式的存储及获取， 就需要此变量的协助， 我们看这两个宏在PHP词法中的定义：

```
//  存在于zend_language_scanner.l文件中
#define YYGETCONDITION()  SCNG(yy_state)
#define YYSETCONDITION(s) SCNG(yy_state) = s
#define SCNG    LANG_SCNG

//  存在于zend_globals_macros.h文件中
# define LANG_SCNG(v) (language_scanner_globals.v)
```

结合前面的全局变量和条件表达式宏的定义， 我们可以知道PHP的词法解析是通过全局变量在一次解析过程中存在。 那么这个条件表达式具体是怎么使用的呢？我们看下面一个例子。这是一个可以识别为结束， 识别字符， 数字等的简单字符串识别器。它使用了re2c的条件表达式， 代码如下：

```
#include <stdio.h>
#include "demo_def.h"
#include "demo.h"

Scanner scanner_globals;

#define YYCTYPE char
#define YYFILL(n)
#define STATE(name)  yyc##name
#define BEGIN(state) YYSETCONDITION(STATE(state))
#define LANG_SCNG(v) (scanner_globals.v)
#define SCNG    LANG_SCNG

#define YYGETCONDITION()  SCNG(yy_state)
#define YYSETCONDITION(s) SCNG(yy_state) = s
#define YYCURSOR  SCNG(yy_cursor)
#define YYLIMIT   SCNG(yy_limit)
#define YYMARKER  SCNG(yy_marker)

int scan(){
    /*!re2c

      <INITIAL>"<?php" {BEGIN(ST_IN_SCRIPTING); return T_BEGIN;}
      <ST_IN_SCRIPTING>[0-9]+ {return T_NUMBER;}
      <ST_IN_SCRIPTING>[ \n\t\r]+ {return T_WHITESPACE;}
      <ST_IN_SCRIPTING>"exit" { return T_EXIT; }
      <ST_IN_SCRIPTING>[a-z]+ {return T_LOWER_CHAR;}
      <ST_IN_SCRIPTING>[A-Z]+ {return T_UPPER_CHAR;}
      <ST_IN_SCRIPTING>"?>" {return T_END;}

      <ST_IN_SCRIPTING>[^] {return T_UNKNOWN;}
      <*>[^] {return T_INPUT_ERROR;}
     */
}

void print_token(int token) {
    switch (token) {
        case T_BEGIN: printf("%s\n", "begin");break;
```

```
        case T_NUMBER: printf("%s\n", "number");break;
        case T_LOWER_CHAR: printf("%s\n", "lower char");break;
        case T_UPPER_CHAR: printf("%s\n", "upper char");break;
        case T_EXIT: printf("%s\n", "exit");break;
        case T_UNKNOWN: printf("%s\n", "unknown");break;
        case T_INPUT_ERROR: printf("%s\n", "input error");break;
        case T_END: printf("%s\n", "end");break;
    }
}

int main(int argc, char* argv[])
{
    int token;
    BEGIN(INITIAL); //  全局初始化，需要放在scan调用之前
    scanner_globals.yy_cursor = argv[1];     //将输入的第一个参数作为要解析的字符串

    while(token = scan()) {
        if (token == T_INPUT_ERROR) {
            printf("%s\n", "input error");
            break;
        }
        if (token == T_END) {
            printf("%s\n", "end");
            break;
        }
        print_token(token);
    }

    return 0;
}
```

和前面的简单示例一样，如果你是在linux环境下，可以使用如下命令生成可执行文件

```
re2c -o demo.c -c -t demo_def.h demo.l
gcc demo.c -o demo -g
chmod +x demo
```

在使用re2c生成C代码时我们使用了-c -t demo_def.h参数，这表示我们使用了条件表达式模式，生成条件的定义头文件。main函数中，在调用scan函数之前我们需要初始化条件状态，将其设置为INITIAL状态。然后在扫描过程中会直接识别出INITIAL状态，然后匹配<?php字符串识别为开始，如果开始不为<?php，则输出input error。在扫描的正常流程中，当扫描出<?php后，while循环继续向下走，此时会再次调用scan函数，当前条件状态为ST_IN_SCRIPTING，此时会跳过INITIAL状态，直接匹配<ST_IN_SCRIPTING>状态后的规则。如果所有的<ST_IN_SCRIPTING>后的规则都无法匹配，输出unkwon。这只是一个简单的识别示例，但是它是从PHP的词法扫描器中抽离出来的，其实现过程和原理类似。

那么这种条件状态是如何实现的呢？我们查看demo.c文件，发现在scan函数开始后有一个跳转语句：

```
int scan(){

#line 25 "demo.c"
{
    YYCTYPE yych;
    switch (YYGETCONDITION()) {
    case yycINITIAL: goto yyc_INITIAL;
    case yycST_IN_SCRIPTING: goto yyc_ST_IN_SCRIPTING;
    }
...
```

```
}
```

在zend_language_scanner.c文件的lex_scan函数中也有类型的跳转过程，只是过程相对这里来说if语句多一些，复杂一些。这就是re2c条件表达式的实现原理。

语法分析

Bison是一种通用目的的分析器生成器。它将LALR(1)上下文无关文法的描述转化成分析该文法的C程序。使用它可以生成解释器，编译器，协议实现等多种程序。Bison向上兼容Yacc，所有书写正确的Yacc语法都应该可以不加修改地在Bison下工作。它不但与Yacc兼容还具有许多Yacc不具备的特性。

Bison分析器文件是定义了名为yyparse并且实现了某个语法的函数的C代码。这个函数并不是一个可以完成所有的语法分析任务的C程序。除此这外我们还必须提供额外的一些函数：如词法分析器、分析器报告错误时调用的错误报告函数等等。我们知道一个完整的C程序必须以名为main的函数开头，如果我们要生成一个可执行文件，并且要运行语法解析器，那么我们就需要有main函数，并且在某个地方直接或间接调用yyparse，否则语法分析器永远都不会运行。

先看下bison的示例：逆波兰记号计算器

```
%{
#define YYSTYPE double
#include <stdio.h>
#include <math.h>
#include <ctype.h>
int yylex (void);
void yyerror (char const *);
%}

%token NUM

%%
input:    /* empty */
     | input line
    ;

line:     '\n'
    | exp '\n'      { printf ("\t%.10g\n", $1); }
;

exp:      NUM           { $$ = $1;           }
   | exp exp '+'   { $$ = $1 + $2;       }
    | exp exp '-'   { $$ = $1 - $2;       }
    | exp exp '*'   { $$ = $1 * $2;       }
    | exp exp '/'   { $$ = $1 / $2;       }
     /* Exponentiation */
    | exp exp '^'   { $$ = pow($1, $2); }
    /* Unary minus    */
    | exp 'n'       { $$ = -$1;           }
;
%%

#include <ctype.h>

int yylex (void) {
       int c;

/* Skip white space.  */
```

```
        while ((c = getchar ()) == ' ' || c == '\t') ;

/* Process numbers.  */
        if (c == '.' || isdigit (c)) {
        ungetc (c, stdin);
        scanf ("%lf", &yylval);
        return NUM;
      }

        /* Return end-of-input.  */
        if (c == EOF) return 0;

        /* Return a single char.  */
        return c;
}

void yyerror (char const *s) {
    fprintf (stderr, "%s\n", s);
}

int main (void) {
    return yyparse ();
}
```

我们先看下运行的效果：

```
bison demo.y
gcc -o test -lm test.tab.c
chmod +x test
./test
```

gcc命令需要添加-lm参数。因为头文件仅对接口进行描述，但头文件不是负责进行符号解析的实体。此时需要告诉编译器应该使用哪个函数库来完成对符号的解析。 GCC的命令参数中，-l参数就是用来指定程序要链接的库，-l参数紧接着就是库名，这里我们在-l后面接的是m，即数学库，他的库名是m，他的库文件名是libm.so。

这是一个逆波兰记号计算器的示例，在命令行中输入 3 7 + 回车，输出10

一般来说，使用Bison设计语言的流程，从语法描述到编写一个编译器或者解释器,有三个步骤:

- 以Bison可识别的格式正式地描述语法。对每一个语法规则，描述当这个规则被识别时相应的执行动作，动作由C语句序列。即我们在示例中看到的%%和%%这间的内容。
- 描述编写一个词法分析器处理输入并将记号传递给语法分析器（即yylex函数一定要存在）。词法分析器既可是手工编写的C代码, 也可以由lex产生，后面我们会讨论如何将re2c与bison结合使用。上面的示例中是直接手工编写C代码实现一个命令行读取内容的词法分析器。
- 编写一个调用Bison产生的分析器的控制函数，在示例中是main函数直接调用。编写错误报告函数（即yyerror函数）。

将这些源代码转换成可执行程序，需要按以下步骤进行：

- 按语法运行Bison产生分析器。对应示例中的命令，bison demo.y
- 同其它源代码一样编译Bison输出的代码，链接目标文件以产生最终的产品。即对应示例中的命令 gcc -o test -lm test.tab.c

我们可以将整个Bison语法文件划分为四个部分。 这三个部分的划分通过%%',%{' 和`%}'符号实现。

一般来说，Bison语法文件结构如下：

```
%{
这里可以用来定义在动作中使用类型和变量，或者使用预处理器命令在那里来定义宏，或者使用
#include包含需要的文件。
如在示例中我们声明了YYSTYPE，包含了头文件math.h等，还声明了词法分析器yylex和错误打印程序
yyerror。
%}

Bison 的一些声明
在这里声明终结符和非终结符以及操作符的优先级和各种符号语义值的各种类型
如示例中的%token  NUM。我们在PHP的源码中可以看到更多的类型和符号声明，如%left，%right的
使用

%%
在这里定义如何从每一个非终结符的部分构建其整体的语法规则。
%%

这里存放附加的内容
这里就比较自由了，你可以放任何你想放的代码。
在开始声明的函数，如yylex等，经常是在这里实现的，我们的示例就是这么搞的。
```

我们在前面介绍了PHP是使用re2c作为词法分析器，那么PHP是如何将re2c与bison集成在一起的呢？我们以一个从PHP源码中剥离出来的示例来说明整个过程。这个示例的功能与上一小节的示例类似，作用都是识别输入参数中的字符串类型。本示例是在其基础上添加了语法解析过程。首先我们看这个示例的语法文件：demo.y

```
%{
#include <stdio.h>
#include "demo_scanner.h"
extern int yylex(znode *zendlval);
void yyerror(char const *);

#define YYSTYPE znode    //关键点一，znode定义在demo_scanner.h
%}

%pure_parser     //  关键点二

%token T_BEGIN
%token T_NUMBER
%token T_LOWER_CHAR
%token T_UPPER_CHAR
%token T_EXIT
%token T_UNKNOWN
%token T_INPUT_ERROR
%token T_END
%token T_WHITESPACE

%%

begin: T_BEGIN {printf("begin:\ntoken=%d\n", $1.op_type);}
     | begin variable {
        printf("token=%d ", $2.op_type);
        if ($2.constant.value.str.len > 0) {
            printf("text=%s", $2.constant.value.str.val);
        }
        printf("\n");
}

variable: T_NUMBER {$$ = $1;}
|T_LOWER_CHAR {$$ = $1;}
```

```
|T_UPPER_CHAR {$$ = $1;}
|T_EXIT {$$ = $1;}
|T_UNKNOWN {$$ = $1;}
|T_INPUT_ERROR {$$ = $1;}
|T_END {$$ = $1;}
|T_WHITESPACE {$$ = $1;}

%%

void yyerror(char const *s) {
    printf("%s\n", s);
}
```

这个语法文件有两个关键点：

1、znode是复制PHP源码中的znode， 只是这里我们只保留了两个字段， 其结构如下：

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

typedef struct _zval_struct {
    /* Variable information */
    zvalue_value value;     /* value */
    int type;    /* active type */
}zval;

typedef struct _znode {
    int op_type;
    zval constant;
}znode;
```

这里我们同样也复制了PHP的zval结构， 但是我们也只取了关于整型， 浮点型和字符串型的结构。op_type用于记录操作的类型， constant记录分析过程获取的数据。 一般来说， 在一个简单的程序中， 对所有的语言结构的语义值使用同一个数据类型就足够用了。比如在前一小节的逆波兰记号计算器示例就只有double类型。 而且Bison默认是对于所有语义值使用int类型。如果要指明其它的类型， 可以像我们示例一样将YYSTYPE定义成一个宏:

```
#define YYSTYPE znode
```

2、%pure_parser 在Bison中声明%pure_parse表明你要产生一个可重入(reentrant)的分析器。默认情况下Bison调用的词法分析函数名为yylex， 并且其参数为void， 如果定义了YYLEX_PARAM， 则使用YYLEX_PARAM为参数， 这种情况我们可以在Bison生成的.c文件中发现其是使用#ifdef实现。

如果声明了%pure_parser， 通信变量yylval和yylloc则变为yyparse函数中的局部变量， 变量yynerrs也变为在yyparse中的局部变量， 而yyparse自己的调用方式并没有改变。比如在我们的示例中我们声明了可重入， 并且使用zval类型的变更作为yylex函数的第一个参数， 则在生成的.c文件中， 我们可以看到yylval的类型变成

一个可重入(reentrant)程序是在执行过程中不变更的程序；换句话说,它全部由纯

(pure)(只读)代码构成。 当可异步执行的时候， 可重入特性非常重要。例如， 从一个句柄调用不可重入程序可能是不安全的。 在带有多线程控制的系统中， 一个非可重入程序必须只能被互锁(interlocks)调用。

通过声明可重入函数和使用znode参数， 我们可以记录分析过程中获取的值和词法分析过程产生的token。 在yyparse调用过程中会调用yylex函数， 在本示例中的yylex函数是借助re2c生成的。 在demo_scanner.l文件中定义了词法的规则。大部分规则是借用了上一小节的示例， 在此基础上我们增加了新的yylex函数， 并且将zendlval作为通信变量， 把词法分析过程中的字符串和token传递回来。 而与此相关的增加的操作为：

```
SCNG(yy_text) = YYCURSOR;   //  记录当前字符串所在位置
/*!re2c
  <!*> {yyleng = YYCURSOR - SCNG(yy_text);} //  记录字符串长度
```

main函数发生了一些改变：

```
int main(int argc, char* argv[])
{
    BEGIN(INITIAL); //  全局初始化，需要放在scan调用之前
    scanner_globals.yy_cursor = argv[1];    //将输入的第一个参数作为要解析的字符串

    yyparse();
    return 0;
}
```

在新的main函数中， 我们新增加了yyparse函数的调用， 此函数在执行过程中会自动调用yylex函数。

如果需要运行这个程序， 则需要执行下面的命令：

```
re2c -o demo_scanner.c -c -t demo_scanner_def.h demo_scanner.l
bison -d demo.y
gcc -o t demo.tab.c demo_scanner.c
chmod +x t
./t "<?php tipi2011"
```

在前面我们以一个小的示例和从PHP源码中剥离出来的示例简单说明了bison的入门和bison与re2c的结合。 当我们用gdb工具Debug PHP的执行流程中编译PHP代码过程如下：

```
#0  lex_scan (zendlval=0xbfffccbc) at Zend/zend_language_scanner.c:841
#1  0x082bab51 in zendlex (zendlval=0xbfffccb8)
    at /home/martin/project/c/phpsrc/Zend/zend_compile.c:4930
#2  0x082a43be in zendparse ()
    at /home/martin/project/c/phpsrc/Zend/zend_language_parser.c:3280
#3  0x082b040f in compile_file (file_handle=0xbffff2b0, type=8)
    at Zend/zend_language_scanner.l:343
#4  0x08186d15 in phar_compile_file (file_handle=0xbffff2b0, type=8)
    at /home/martin/project/c/phpsrc/ext/phar/phar.c:3390
#5  0x082d234f in zend_execute_scripts (type=8, retval=0x0, file_count=3)
    at /home/martin/project/c/phpsrc/Zend/zend.c:1186
#6  0x08281b70 in php_execute_script (primary_file=0xbffff2b0)
    at /home/martin/project/c/phpsrc/main/main.c:2225
#7  0x08351b97 in main (argc=4, argv=0xbffff424)
    at /home/martin/project/c/phpsrc/sapi/cli/php_cli.c:1190
```

在PHP源码中，词法分析器的最终是调用re2c规则定义的lex_scan函数，而提供给Bison的函数则为zendlex。而yyparse被zendparse代替。

实现自己的语法

经过前面对r2ec以及Bison的介绍，熟悉了PHP语法的实现，我们来动手自己实现一个语法吧。也就是对Zend引擎语法层面的实现。以此来对Zend引擎有更多的了解。

编程语言和社会语言一样都是会慢慢演进的，不同的语种就像我们的不同国家的语言一样，他们各有各的特点，语言通常也能反映出一个群体的特质，不同语言的社区氛围和文化也都会有 很大的差异，和现实生活一样，我们也需要尽可能的去接触不同的文化，来开阔自己的视野和 思维方式，所以我们也建议多学习不同的编程语言。

在这里简单提一下PHP语言的演进，PHP的语法继承自Perl的语法，这一点和自然语言也很类似，语言之间会互相影响，比如PHP5开始完善的面向对象机制，已经PHP5.4中增加的命名空间以及闭包等等功能。

PHP是个开源项目，它的发展是由社区来决定的，它也是开放的，如果你有想要改进它的愿望都可以加入到这个社区当中，当然也不是谁都可以改变PHP，重大改进都需要由社区确定，只有有限的人 具有对代码库的修改权限，如果你发现了PHP的Bug可以去http://bugs.php.net提交Bug, 如果同时你也找到了Bug的原因那么你也可以同时附上对Bug的修复补丁，然后在PHP邮件组中进行一些 讨论，如果没有问题那么有权限的成员就可以将你的补丁合并进入相应的版本内，更多内容可以参考 附录D　怎样为PHP共享自己的力量。

在本小节中将要实现一个对PHP本身语言的一个“需求”：返回变量的名称。用一小段代码简单描述一下这个需求：

```
[php]
<?php
$demo = 'tipi';
echo var_name($demo);    //执行结果，输出： demo
?>
```

经过前面的章节，我们了解到，一种PHP语法的内部实现，主要经历了以下步骤：

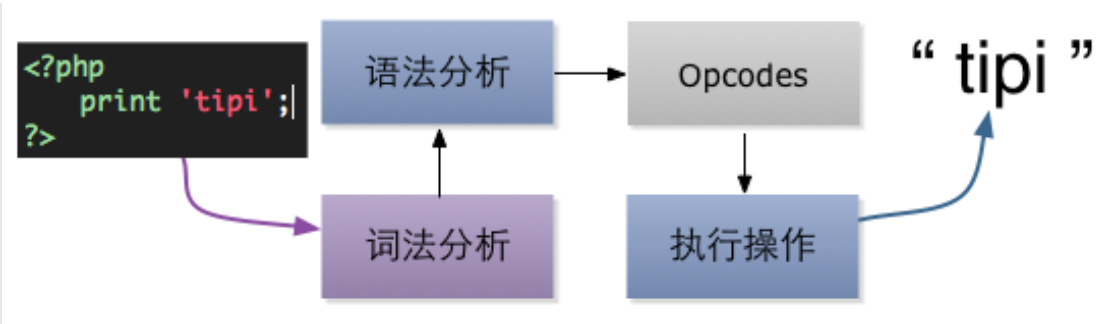


图7.2 Zend Opcodes执行

即:词法分析 => 语法分析 => opcode编译 => 执行

由此，我们还是要从词法和语法分析着手。

词法分析与语法分析

熟悉编译原理的朋友应该比较熟悉这两个概念，简而言之，就是在要运行的程序中，根据原来设定好的“关键字”（Tokens），将每条程序指令解释成为可以由语言解释器理解的操作。

在PHP中，可以使用token_get_all()函数来查看一段PHP代码生成的Tokens。

PHP的词法分析和语法分析的实现分别位于Zend目录下的zend_language_scanner.l和zend_language_parser.y 文件，使用r2ec&flex来编译。我们要做的，就是在PHP原有的词法和语法分析中，加入新的Token,在zend_language_scanner.l中 加入以下内容：

```
"var_name" {
    return T_VARIABLE_NAME;
}
```

也就是在此法分析阶段遇到var_name这个字符串的时候会被标记为我们定义的T_VARIABLE_NAME token。

同样，在 zend_language_parser.y 也需要加入对这个token的处理，通常是进行响应的逻辑处理。我们要实现的语法和PHP内置的echo print结构类似，所以我们把这个处理放到 internal_functions_in_yacc 规则里面：

```
| T_VARIABLE_NAME '(' T_VARIABLE ')' { zend_do_variable_name(&$$, &$3
TSRMLS_CC); }
| T_VARIABLE_NAME T_VARIABLE { zend_do_variable_name(&$$, &$2 TSRMLS_CC); }
```

上面的两条规则分别对于类似：

```
<?php
echo var_name($varname);
echo var_name $varname;
```

的两种调用方式，和include() require() 类似。

大家可以很容易理解第一行的定义，如果发现T_VARIABLE_NAME + (+ 变量 +), 则使用zend_do_variable_name来处理，&$$是当前表达式的返回值，&$3表示第三个表达式的值，也就是T_VARIABLE，也就是一个通常的变量定义。这样就是把变量相关的信息传递进zend_do_variable_name()函数中进行处理。在这里是获取变量的名称，然后进行opcode编译。

opcode编译

在开始之前需要向大家介绍一下PHP opcode的定义及执行。opcode在PHP中通常是一个数字唯一标示，在PHP中目前对每个opcode对应的执行方法的分发提供了3种方式:

首先，我们在Zend/zend_vm_opcodes.h 为我们的新opcode 加入一个宏定义：

```
#define ZEND_VARIABLE_NAME 154
```

这个数字要求在0-255之间，并且不能与现有opcode重复。

第二步，在Zend/zend_compile.c中加入我们对OPCODE的处理，也就是将代码操作转化为op_array放入到opline中：

```
void zend_do_variable_name(znode *result, znode *variable TSRMLS_DC) /* {{{ */
{
    // 生成一条zend_op
    zend_op *opline = get_next_op(CG(active_op_array) TSRMLS_CC);

    // 因为我们需要有返回值，并且返回值只作为中间值.所以就是一个临时变量
    opline->result.op_type = IS_TMP_VAR;
    opline->result.u.var = get_temporary_variable(CG(active_op_array));

    opline->opcode = ZEND_VARIABLE_NAME;
    opline->op1 = *variable;

    // 我们只需要一个操作数就好了
    SET_UNUSED(opline->op2);
    *result = opline->result;
}
```

这样，我们就完成了对opcode的编译。

内部处理逻辑的编写

经过在上面两个步骤中，我们已经完成了自定义PHP语法的语法规则定义，opcode编译。最后的工作，就是定义如何处理自定义的opcode，以及编写具体的代码逻辑。在前面关于如何找到opcode具体实现的小节，我们提到 Zend/zend_vm_execute.h中的zend_vm_get_opcode_handler()函数。这个函数就是用来获取opcode的执行函数。

这个对应的关系，是根据一个公式来进行了，目的是将不同的参数类型分开，对应到多个处理函数，公式是这样的：

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

从这个公式我们可以看出，最终的处理函数是与参数类型有关，根据计算，我们要满足所有类型的映射，尽管我们可以可以使用同一函数进行处理，于是，我们在zend_opcode_handlers这个数组的结尾，加上25个相同的函数定义：

```
void zend_init_opcodes_handlers(void)
{
    static const opcode_handler_t labels[] = {
    ....
    ZEND_VARIABLE_NAME_HANDLER,
    ....
    ZEND_VARIABLE_NAME_HANDLER
}
```

如果我们不想支持某类型的数据，只需要将类型代入公式计算出的数字做为索引，使opcode_handler_t中相应的项为：ZEND_NULL_HANDLER

最后， 我们在Zend/zend_vm_def.h 中增加相应的处理函数。

和对语法的修改一样， opcode处理函数也不是直接修改Zend/zend_vm_execute.h文件的， 这是因为PHP提供了3种opcode分发的机制： 1. CALL 函数调用的方式分发 1. SWITCH 使用SWITCH case 进行分发 1. GOTO 使用goto语句进行分发 之所以提供3中方式主要是从性能出发的， 可能在不同的CPU上这几种调用方式的效率并不一样。 默认采用的是CALL

回到编写返回变量名的具体实现， 在Zend/zend_vm_def.h中增加如下：

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

    // PHP中所有的变量在内部都是存储在zval结构中的.
    zval *result = &EX_T(opline->result.u.var).tmp_var;

    // 把变量的名字赋给临时返回值
    Z_STRVAL(*result) = estrndup(opline->op1.u.constant.value.str.val, opline-
>op1.u.constant.value.str.len);
    Z_STRLEN(*result) = opline->op1.u.constant.value.str.len;
    Z_TYPE(EX_T(opline->result.u.var).tmp_var) = IS_STRING;

    ZEND_VM_NEXT_OPCODE();
}
```

进行完上面的修改之后， 我们要删除r2ec&flex已经编译好的原文件， 即删除Zend/zend_language*.c文件以使新的语法规则生效。 这样我们再次对PHP源码进行make时， 会自动生成新的编译好的语法规则处理程序， 不过， 编译环境要安装有lex&yacc和re2c。

从上面的步骤可以看出， php语法的扩展并不困难， 而真正的难点在于如何在当前zend内核框架基础上进行的具体功能的实现， 以及到底应该实现什么语法。关于语法的改进通常也是一个漫长的过程， 要修改语言的语法通常需要：

1. 提出需求， 并说明该语法的作用， 以及具体的应用场景， 这个语法带来的好处
2. 大家讨论这个需求是否合理， 实现啊起来是否有困难， 对现有的语法是否造成影响
3. 如果大部分人都认可这个需求最好， 那么提出该需求的人可以自己来实现， 并让大家review， 如果没有问题则就可以进入版本库了。
4. 如果比较有争议， 那可能需要进行投票了。

更多内容请参考附录：怎么样为PHP做贡献小节。

第三节 中间代码的执行

在<< 第二章第三小节 PHP脚本的执行 -- opcode >>中， 我们对opcode进行了一个简略的说明。 这一小节我们讲这些中间代码在Zend虚拟机中是如何被执行的。

假如我们现在使用的是CLI模式， 直接在SAPI/cli/php_cli.c文件中找到main函数， 默认情况下PHP的CLI模式的行为模式为PHP_MODE_STANDARD。 此行为模式中PHP内核会调用php_execute_script(&file_handle TSRMLS_CC);来执行PHP文件。 顺着这条执行的线路， 可以看到一个PHP文件在经过词法分析， 语法分析， 编译后生成中间代码的过程：

```
EG(active_op_array) = zend_compile_file(file_handle, type TSRMLS_CC);
```

在销毁了文件所在的handler后，如果存在中间代码，则PHP虚拟机将通过以下代码执行中间代码：

```
zend_execute(EG(active_op_array) TSRMLS_CC);
```

如果你是使用VS查看源码的话，将光标移到zend_execute并直接按F12，你会发现zend_execute的定义跳转到了一个指针函数的声明(Zend/zend_execute_API.c)。

```
ZEND_API void (*zend_execute)(zend_op_array *op_array TSRMLS_DC);
```

这是一个全局的函数指针，它的作用就是执行PHP代码文件解析完的转成的zend_op_array。和zend_execute相同的还有一个zedn_execute_internal函数，它用来执行内部函数。在PHP内核启动时(zend_startup)时，这个全局函数指针将会指向execute函数。注意函数指针前面的修饰符ZEND_API，这是ZendAPI的一部分。在zend_execute函数指针赋值时，还有PHP的中间代码编译函数zend_compile_file（文件形式）和zend_compile_string(字符串形式)。

```
zend_compile_file = compile_file;
zend_compile_string = compile_string;
zend_execute = execute;
zend_execute_internal = NULL;
zend_throw_exception_hook = NULL;
```

这几个全局的函数指针均只调用了系统默认实现的几个函数，比如compile_file和compile_string函数，他们都是以全局函数指针存在，这种实现方式在PHP内核中比比皆是，其优势在于更低的耦合度，甚至可以定制这些函数。比如在APC等opcode优化扩展中就是通过替换系统默认的zend_compile_file函数指针为自己的函数指针my_compile_file，并且在my_compile_file中增加缓存等功能。

到这里我们找到了中间代码执行的最终函数：execute(Zend/zend_vm_execure.h)。在这个函数中所有的中间代码的执行最终都会调用handler。这个handler是什么呢？

```
if ((ret = EX(opline)->handler(execute_data TSRMLS_CC)) > 0) {
}
```

这里的handler是一个函数指针，它指向执行该opcode时调用的处理函数。此时我们需要看看handler函数指针是如何被设置的。在前面我们有提到和execute一起设置的全局指针函数：zend_compile_string。它的作用是编译字符串为中间代码。在Zend/zend_language_scanner.c文件中有compile_string函数的实现。在此函数中，当解析完中间代码后，一般情况下，它会执行pass_two(Zend/zend_opcode.c)函数。pass_two这个函数，从其命名上真有点看不出其意义是什么。但是我们关注的是在函数内部，它遍历整个中间代码集合，调用ZEND_VM_SET_OPCODE_HANDLER(opline);为每个中间代码设置处理函数。ZEND_VM_SET_OPCODE_HANDLER是zend_vm_set_opcode_handler函数的接口宏，zend_vm_set_opcode_handler函数定义在Zend/zend_vm_execute.h文件。其代码如下：

```
static opcode_handler_t zend_vm_get_opcode_handler(zend_uchar opcode, zend_op*
op)
{
        static const int zend_vm_decode[] = {
            _UNUSED_CODE, /* 0                */
```

```
                _CONST_CODE,   /* 1 = IS_CONST    */
                _TMP_CODE,     /* 2 = IS_TMP_VAR */
                _UNUSED_CODE, /* 3                */
                _VAR_CODE,     /* 4 = IS_VAR      */
                _UNUSED_CODE, /* 5                */
                _UNUSED_CODE, /* 6                */
                _UNUSED_CODE, /* 7                */
                _UNUSED_CODE, /* 8 = IS_UNUSED   */
                _UNUSED_CODE, /* 9                */
                _UNUSED_CODE, /* 10               */
                _UNUSED_CODE, /* 11               */
                _UNUSED_CODE, /* 12               */
                _UNUSED_CODE, /* 13               */
                _UNUSED_CODE, /* 14               */
                _UNUSED_CODE, /* 15               */
                _CV_CODE       /* 16 = IS_CV      */
        };
        return zend_opcode_handlers[opcode * 25
                + zend_vm_decode[op->op1.op_type] * 5
                + zend_vm_decode[op->op2.op_type]];
}

ZEND_API void zend_vm_set_opcode_handler(zend_op* op)
{
    op->handler = zend_vm_get_opcode_handler(zend_user_opcodes[op->opcode],
op);
}
```

在前面章节<< 第二章第三小节 -- opcode处理函数查找 >>中介绍了四种查找opcode处理函数的方法， 而根据其本质实现查找也在其中， 只是这种方法对于计算机来说比较容易识别， 而对于自然人来说却不太友好。 比如一个简单的A + B的加法运算， 如果你想用这种方法查找其中间代码的实现位置的话， 首先你需要知道中间代码的代表的值， 然后知道第一个表达式和第二个表达式结果的类型所代表的值， 然后计算得到一个数值的结果， 然后从数组zend_opcode_handlers找这个位置， 位置所在的函数就是中间代码的函数。 这对阅读代码的速度没有好处， 但是在开始阅读代码的时候根据代码的逻辑走这样一个流程却是大有好处。

回到正题。 handler所指向的方法基本都存在于Zend/zend_vm_execute.h文件文件。 知道了handler的由来， 我们就知道每个opcode调用handler指针函数时最终调用的位置。

在opcode的处理函数执行完它的本职工作后， 常规的opcode都会在函数的最后面添加一句：ZEND_VM_NEXT_OPCODE();。 这是一个宏， 它的作用是将当前的opcode指针指向下一条opcode， 并且返回0。如下代码：

```
#define ZEND_VM_NEXT_OPCODE() \
CHECK_SYMBOL_TABLES() \
EX(opline)++; \
ZEND_VM_CONTINUE()

#define ZEND_VM_CONTINUE()    return 0
```

在execute函数中， 处理函数的执行是在一个while(1)循环作用范围中。如下：

```
while (1) {
        int ret;
#ifdef ZEND_WIN32
        if (EG(timed_out)) {
```

```
            zend_timeout(0);
        }
#endif

        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;
            }
        }

    }
```

前面说到每个中间代码在执行完后都会将中间代码的指针指向下一条指令，并且返回0。当返回0时，while 循环中的if语句都不满足条件，从而使得中间代码可以继续执行下去。正是这个while(1)的循环使得PHP内核中的opcode可以从第一条执行到最后一条，当然这中间也有一些函数的跳转或类方法的执行等。

以上是一条中间代码的执行，那么对于函数的递归调用，PHP内核是如何处理的呢？看如下一段PHP代码：

```
function t($c) {
    echo $c, "\n";
    if ($c > 2) {
            return ;
    }
    t($c + 1);
}
t(1);
```

这是一个简单的递归调用函数实现，它递归调用了两次，这个递归调用是如何进行的呢？我们知道函数的调用所在的中间代码最终是调用zend_do_fcall_common_helper_SPEC(Zend/zend_vm_execute.h)。在此函数中有如下一段：

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

前面提到zend_execute API可能会被覆盖，这里就进行了简单的判断，如果扩展覆盖了opcode执行函数，则进行特殊的逻辑处理。

上一段代码中的ZEND_VM_ENTER()定义在Zend/zend_vm_execute.h的开头，如下：

```
#define ZEND_VM_CONTINUE()    return 0
#define ZEND_VM_RETURN()      return 1
```

```
#define ZEND_VM_ENTER()         return 2
#define ZEND_VM_LEAVE()         return 3
```

这些在中间代码的执行函数中都有用到，这里的ZEND_VM_ENTER()表示return 2。在前面的内容中我们有说到在调用了EX(opline)->handler(execute_data TSRMLS_CC))后会将返回值赋值给ret。然后根据ret判断下一步操作，这里的递归函数是返回2，于是下一步操作是：

```
op_array = EG(active_op_array);
goto zend_vm_enter;
```

这里将EG(active_op_array)的值赋给op_array后，直接跳转到execute函数的定义的zend_vm_enter标签，此时的EG(active_op_array)的值已经在zend_do_fcall_common_helper_SPEC中被换成了当前函数的中间代码集合，其实现代码为：

```
if (EX(function_state).function->type == ZEND_USER_FUNCTION) {  //  用户自定义的函数
            EX(original_return_value) = EG(return_value_ptr_ptr);
            EG(active_symbol_table) = NULL;
            EG(active_op_array) = &EX(function_state).function->op_array;    //  将当前活动的中间代码指针指向用户自定义函数的中间代码数组
            EG(return_value_ptr_ptr) = NULL;
```

当内核执行完用户自定义的函数后，怎么返回之前的中间代码代码主干路径呢？这是由于在execute函数中初始化数据时已经将当前的路径记录在EX(op_array)中了（EX(op_array) = op_array;）当用户函数返回时程序会将之前保存的路径重新恢复到EG(active_op_array)中（EG(active_op_array) = EX(op_array);）。可能此时你会问如果函数没有返回呢？这种情况在用户自定义的函数中不会发生的，就算是你没有写return语句，PHP内核也会自动给加上一个return语句，这在第四章 << 第四章 函数的实现 » 第二节 函数的定义，传参及返回值 » 函数的返回值 >>已经有说明过。

整个调用路径如下图所示：

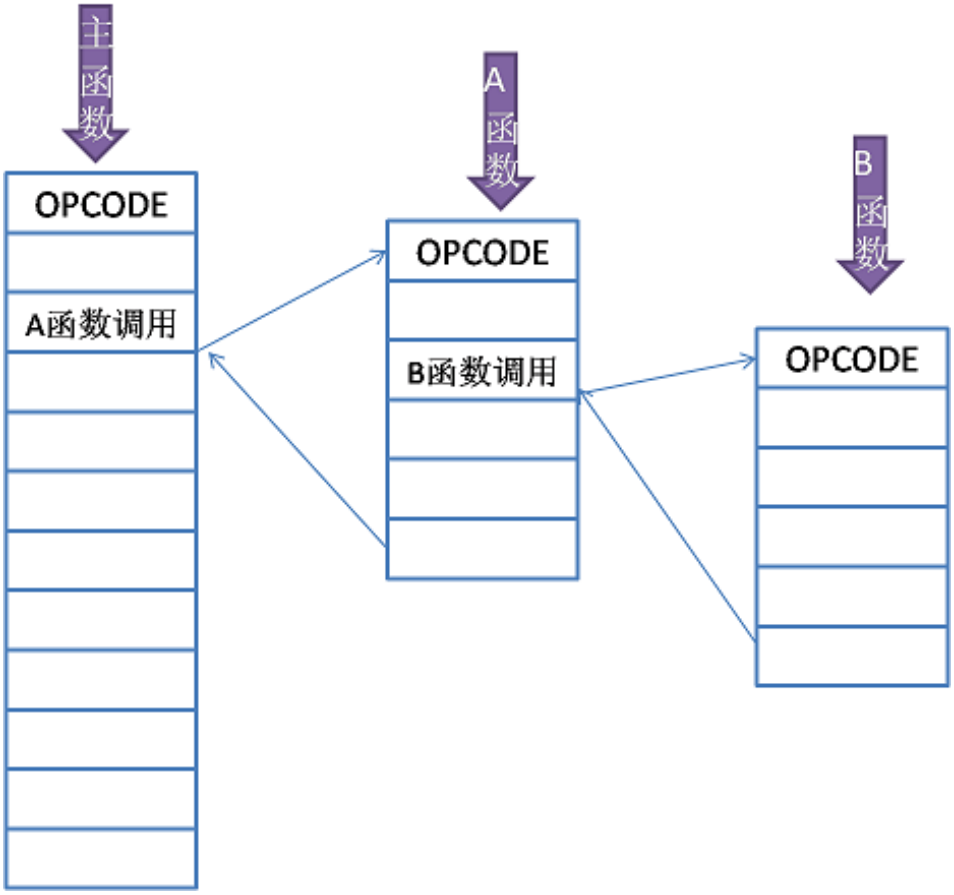


图7.2 Zend中间代码调用路径图

以上是opcode的执行过程，与过程相比，过程中的数据会更加重要，那么在执行过程中的核心数据结构有哪些呢？在Zend/zend_vm_execute.h文件中的execute函数实现中，zend_execute_data类型的execute_data变量贯穿整个中间代码的执行过程，其在调用时并没有直接使用execute_data，而是使用EX宏代替，其定义在Zend/zend_compile.h文件中，如下：

```
#define EX(element) execute_data.element
```

因此我们在execute函数或在opcode的实现函数中会看到EX(fbc)，EX(object)等宏调用，它们是调用函数局部变量execute_data的元素：execute_data.fbc和execute_data.object。execute_data不仅仅只有fbc、object等元素，它包含了执行过程中的中间代码，上一次执行的函数，函数执行的当前作用域，类等信息。其结构如下：

```
typedef struct _zend_execute_data zend_execute_data;

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;  /* 当前执行的中间代码 */
    zval *object;
    union _temp_variable *Ts;
    zval ***CVs;
    HashTable *symbol_table; /* 符号表 */
    struct _zend_execute_data *prev_execute_data;    /* 前一条中间代码执行的环境*/
    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;
};
```

在前面的中间代码执行过程中有介绍：中间代码的执行最终是通过EX(opline)->handler(execute_data TSRMLS_CC)来调用最终的中间代码程序。 在这里会将主管中间代码执行的execute函数中初始化好的execture_data传递给执行程序。

zend_execute_data结构体部分字段说明如下：

- opline字段：struct _zend_op类型， 当前执行的中间代码
- op_array字段： zend_op_array类型， 当前执行的中间代码队列
- fbc字段：zend_function类型， 已调用的函数
- called_scope字段：zend_class_entry类型， 当前调用对象作用域， 常用操作是EX(called_scope) = Z_OBJCE_P(EX(object))， 即将刚刚调用的对象赋值给它。
- symbol_table字段： 符号表， 存放局部变量， 这在前面的<< 第六节 变量的生命周期 » 变量的作用域 >>有过说明。 在execute_data初始时， EX(symbol_table) = EG(active_symbol_table);
- prev_execute_data字段：前一条中间代码执行的中间数据， 用于函数调用等操作的运行环境恢复。

在execute函数中初始化时， 会调用zend_vm_stack_alloc函数分配内存。 这是一个栈的分配操作， 对于一段PHP代码的上下文环境， 它存在于这样一个分配的空间作放置中间数据用， 并作为栈顶元素。 当有其它上下文环境的切换（如函数调用）， 此时会有一个新的元素生成， 上一个上下文环境会被新的元素压下去， 新的上下文环境所在的元素作为栈顶元素存在。

在zend_vm_stack_alloc函数中我们可以看到一些PHP内核中的优化。 比如在分配时， 这里会存在一个最小分配单元， 在zend_vm_stack_extend函数中， 分配的最小单位是ZEND_VM_STACK_PAGE_SIZE((64 * 1024) - 64)， 这样可以在一定范围内控制内存碎片的大小。 又比如判断栈元素是否为空， 在PHP5.3.1之前版本(如5.3.0)是通过第四个元素elelments与top的位置比较来实现， 而从PHP5.3.1版本开始， struct _zend_vm_stack结构就没有第四个元素， 直接通过在当前地址上增加整个结构体的长度与top的地址比较实现。 两个版本结构代码及比较代码如下：

```
// PHP5.3.0
struct _zend_vm_stack {
    void **top;
    void **end;
    zend_vm_stack prev;
    void *elements[1];
};

if (UNEXPECTED(EG(argument_stack)->top == EG(argument_stack)->elements)) {
}

//  PHP5.3.1
struct _zend_vm_stack {
    void **top;
    void **end;
    zend_vm_stack prev;
};

if (UNEXPECTED(EG(argument_stack)->top ==
ZEND_VM_STACK_ELEMETS(EG(argument_stack)))) {
}
```

```
#define ZEND_VM_STACK_ELEMETS(stack) \
((void**)(((char*)(stack)) + ZEND_MM_ALIGNED_SIZE(sizeof(struct
_zend_vm_stack))))
```

当一个上下文环境结束其生命周期后，如果回收这段内存呢？还是以函数为例，我们在前面的函数章节中<< 函数的返回 >>中我们知道每个函数都会有一个函数返回，即使没有在函数的实现中定义，也会默认返回一个NULL。以ZEND_RETURN_SPEC_CONST_HANDLER实现为例，在函数的返回最后都会调用一个函数**zend_leave_helper_SPEC**。

在zend_leave_helper_SPEC函数中，对于执行过程中的函数处理有几个关键点：

- 上下文环境的切换：这里的关键代码是：EG(current_execute_data) = EX(prev_execute_data);。EX(prev_execute_data)用于保留当前函数调用前的上下文环境，从而达到恢复和切换的目的。
- 当前上下文环境所占用内存空间的释放：这里的关键代码是：zend_vm_stack_free(execute_data TSRMLS_CC);。zend_vm_stack_free函数的实现存在于Zend/zend_execute.h文件，它的作用就是释放栈元素所占用的内存。
- 返回到之前的中间代码执行路径中：这里的关键代码是：ZEND_VM_LEAVE();。我们从zend_vm_execute.h文件的开始部分就知道ZEND_VM_LEAVE宏的效果是返回3。在执行中间代码的while循环当中，当ret=3时，这个执行过程就会恢复之前上下文环境，继续执行。

第四节 PHP代码的加密解密

PHP语言作为脚本语言的一种，由于不需要进行编译，所以通常PHP程序的分发都是直接发布源代码。对于一些开源软件来说，这并没有什么问题，因为它本来就希望有更多的人阅读代码，希望有更多的人参与进来，而对于商业代码来说，这却是一个不太好的消息，不管是从商业秘密，还是从对公司产权的保护来说却是一个问题，基于此，从而引出了对PHP代码的加密和解密的议题。例如国内的Discuz论坛程序在开源之前要运行是必须安装Zend Optimizer的，Zend官方的代码加密软件是Zend Guard，可以用来加密和混淆PHP代码，这样分发出去的代码就可以避免直接分发源代码，不过加密后的代码是无法直接运行的，在运行时还需要一个解密的模块来运行加密后的程序，要运行Zend Guard加密后的代码需要安装Zend Optimizer(PHP5.2之前的版本)，或者安装Zend Guard Loader(PHP5.3版本)扩展才能运行。

加密的本质

本质上程序在运行时都是在执行机器码，而基于虚拟机的语言的加密通常也是加密到这个级别，也就是说PHP加密后的程序在执行之前都会解密成opcode来执行。

PHP在执行之前有一个编译的环节，编译的结果是opcode，然后由Zend虚拟机执行，从这里看如果只要将源代码加密，然后在执行之前将代码解密即可。

从这里看，只要代码能被解密为opcode，那么总有可能反编译出来源代码，其他的语言中也是类似，比如objdump程序能将二进制程序反汇编出来，.NET、Java的程序也是一样，都有一些反编译的程序，不过通常这些厂商同时还会 附带代码混淆的工具，经过混淆的代码可读性极差，很多人都留意过Gmail等网站 经过混淆的JS代码吧，他们阅读起来非常困难，经过混淆的代码即使反编译出来，读者也很难通过代码分析出代码中的逻辑，这样也就极大的增加了应用的安全性。

简单的代码加密解密实战

根据前文的介绍，作为实例，本文将编写一个简单的代码加密扩展用于对PHP代码的加密，我们只需要能把源码加密，简单通过浏览源代码的方法无法获取到源代码那我们的目标就达到了，为了能正确执行加密后的代码，我们还需要另一个模块：解密模块。

简单的思路是把所有的PHP文件代码进行加密，同时另存为同名的PHP文件，这是一种很简单的做法，只是为了防止源代码赤裸裸的暴露在代码中。

加密也有很多种做法，第一种简单的方法可以简单的把源码本身进行一些可逆加密，这样我们可以在运行之前把真实的源码反解出来执行，不过这种方式存在一种问题，只要知道了加密算法我们就可以把代码给解出来，采用这种方式唯一能做的就是尽量 增加加密的复杂度，既然正式的代码在运行之前会被转化成PHP源代码，通过hack的方式是可以 完完整整的获得PHP源码的，保密的效果就很有限了。

因为Zend引擎最终执行的是opcode，那么我们只要保证能解密出opcode则能满足需求，我们只要简单的将opcode进行简单的序列化或者像Zend Guard那样进行混淆，在运行之前将opcode还原，那么源代码的信息就不存在了，这样我们就能保证源代码的安全，而不至于泄露。

加密

前面提到加密的目的就是为了防止轻易获取程序源码的一种手段，对于PHP来说，将源码编译为opcode已经能达到目的了，因为PHP引擎最终都是需要执行opcode的。虽然可以将加密进一步，但是如果需要修改Zend引擎，那么成本就有点大了，因为需要修改 Zend引擎了，而这是无法通过简单的扩展机制来实现了，所以解密的成本也会变的太大，也就没有实际意义了。

在本例中为了方便，代码的加密和解密实现均实现在同一个模块中。

熟悉PHP的同学可能会发现，这种加密方式和opcode缓存本质上没有太大差别，opcode缓存的工作是将 源码编译为opcode然后缓存起来，在执行的时候绕过编译直接执行opcode，的确是没错的。这里唯一的区别是： opcode缓存是动态透明的，而加密后我们要做的是分发加密后的代码。这么说我们是不是可以直接将 APC之类的缓存扩展进行改造就可以了，其实理论上是可以的。不过这两者的定位还是有差别的：加密的目的是为了减少源码被分析破解的可能，而缓存只是为了提高程序运行的速度。

解密

本例中的代码其实并没有进行加密，相对源代码来说，opcode编译本身也可以算做一种加密了，因为毕竟通过阅读opcode来理解程序的逻辑还是比较困难的。

第五节 小结

Zend引擎作为PHP的核心，它的作用尤为重要，Zend引擎实现了PHP的语法以及语言扩展机制，前面的章节已经陆陆续续介绍了Zend引擎的一些内容,本章则详细介绍了Zend虚拟机的实现机制。

本章完整的介绍了词法分析和语法分析的细节内容，同时完整的实现了一个PHP语法结构，用以在运行时获取到变量的名称。随后介绍了PHP代码的分发和安全，以此引出PHP代码的加密和解密，并实现了

一个简单的扩展来实现代码的加密。

附录A: PHP及Zend API

附录B PHP的历史

附录C VLD扩展使用指南

VLD(Vulcan Logic Dumper)是一个挂钩在Zend引擎下， 并且输出PHP脚本生成的中间代码（执行单元）的扩展。 它可以在一定程序上查看Zend引擎内部的一些实现原理， 是我们学习PHP源码的必备良器。它的作者是Derick Rethans, 除了VLD扩展， 我们常用的XDebug扩展的也有该牛人的身影。

VLD扩展是一个开源的项目， 在这里可以下载到最新的版本， 虽然最新版本的更新也是一年前的事了。作者没有提供编译好的扩展， Win下使用VC6.0编译生成dll文件， 可以看笔者的一篇文章(使用VC6.0生成VLD扩展)。

*nix系统下直接configue,make,make install生成。如果遇到问题， 请自行Google之。

看一个简单的例子,假如存在t.php文件， 其内容如下：

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

在命令行下使用VLD扩展显示信息。

```
php -dvld.active=1 t.php
```

-dvld.active=1表示激活VLD扩展， 使用VLD扩展输出中间代码， 此命令在CMD中输出信息为：

```
Branch analysis from position: 0
Return found
filename:       D:\work\xampp\xampp\php\t.php
function name:  (null)
number of ops:  5
compiled vars:  !0 = $a
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------
--
   2     0  >   EXT_STMT
         1      ASSIGN                                                 !0, 10
   3     2      EXT_STMT
         3      ECHO                                                   !0
   4     4    > RETURN                                                 1

branch: #  0; line:     2-    4; sop:     0; eop:     4
path #1: 0,
10
```

如上为VLD输出的PHP代码生成的中间代码的信息， 说明如下：

- Branch analysis from position 这条信息多在分析数组时使用。
- Return found 是否返回， 这个基本上有都有。
- filename 分析的文件名
- function name 函数名， 针对每个函数VLD都会生成一段如上的独立的信息， 这里显示当前函数的名称
- number of ops 生成的操作数
- compiled vars 编译期间的变量， 这些变量是在PHP5后添加的， 它是一个缓存优化。这样的变量在

PHP源码中以IS_CV标记。

- op list 生成的中间代码的变量列表

使用-dvld.active参数输出的是VLD默认设置，如果想看更加详细的内容。可以使用-dvld.verbosity参数。

```
php -dvld.active=1 -dvld.verbosity=3 t.php
```

-dvld.verbosity=3或更大的值的效果都是一样的，它们是VLD在当前版本可以显示的最详细的信息了，包括各个中间代码的操作数等。显示结果如下：

```
Finding entry points
Branch analysis from position: 0
Add 0
Add 1
Add 2
Add 3
Add 4
Return found
filename:       D:\work\xampp\xampp\php\t.php
function name:  (null)
number of ops:  5
compiled vars:  !0 = $a
line     # *  op                           fetch          ext  return  operands
---------------------------------------------------------------------------------
-
-
   2     0  >   EXT_STMT                                               RES[
IS_UNUSED  ]         OP1[  IS_UNUSED  ] OP2[  IS_UNUSED  ]
         1      ASSIGN
OP1[IS_CV !0 ] OP2[ ,  IS_CONST (0) 10 ]
   3     2      EXT_STMT                                               RES[
IS_UNUSED  ]         OP1[  IS_UNUSED  ] OP2[  IS_UNUSED  ]
         3      ECHO
OP1[IS_CV !0 ]
         4    > RETURN
OP1[IS_CONST (0) 1 ]

branch: #  0; line:     2-    3; sop:     0; eop:     4
path #1: 0,
10
```

以上的信息与没有加-dvld.verbosity=3的输出相比，多了Add 字段，还有中间代码的操作数的类型，如IS_CV，IS_CONST等。PHP代码中的$a = 10; 其中10的类型为IS_CONST，$a作为一个编译期间的一个缓存变量存在，其类型为IS_CV。

如果我们只是想要看输出的中间代码，并不想执行这段PHP代码，可以使用-dvld.execute=0来禁用代码的执行。

```
php -dvld.active=1 -dvld.execute=0 t.php
```

运行这个命令，你会发现这与最开始的输出有一点点不同，它没有输出10。除了直接在屏幕上输出以外，VLD扩展还支持输出.dot文件，如下的命令：

```
php -dvld.active=1 -dvld.save_dir='D:\tmp' -dvld.save_paths=1 -
```

```
dvld.dump_paths=1 t.php
```

以上的命令的意思是将生成的中间代码的一些信息输出在D:/tmp/paths.dot文件中。 -dvld.save_dir指定文件输出的路径， -dvld.save_paths控制是否输出文件， -dvld.dump_paths控制输出的内容， 现在只有0和1两种情况。 输出的文件名已经在程序中硬编码为paths.dot。这三个参数是相互依赖的关系， 一般都会同时出现。

总结一下， VLD扩展的参数列表：

- -dvld.active 是否在执行PHP时激活VLD挂钩， 默认为0， 表示禁用。可以使用-dvld.active=1启用。
- -dvld.skip_prepend 是否跳过php.ini配置文件中auto_prepend_file指定的文件， 默认为0， 即不跳过包含的文件， 显示这些包含的文件中的代码所生成的中间代码。此参数生效有一个前提条件：-dvld.execute=0
- -dvld.skip_append 是否跳过php.ini配置文件中auto_append_file指定的文件， 默认为0， 即不跳过包含的文件， 显示这些包含的文件中的代码所生成的中间代码。此参数生效有一个前提条件：-dvld.execute=0
- -dvld.execute 是否执行这段PHP脚本， 默认值为1， 表示执行。可以使用-dvld.execute=0， 表示只显示中间代码， 不执行生成的中间代码。
- -dvld.format 是否以自定义的格式显示， 默认为0， 表示否。可以使用-dvld.format=1， 表示以自己定义的格式显示。这里自定义的格式输出是以-dvld.col_sep指定的参数间隔
- -dvld.col_sep 在-dvld.format参数启用时此函数才会有效， 默认为 "\t"。
- -dvld.verbosity 是否显示更详细的信息， 默认为1， 其值可以为0， 1， 2， 3 其实比0小的也可以， 只是效果和0一样， 比如0.1之类， 但是负数除外， 负数和效果和3的效果一样 比3大的值也是可以的， 只是效果和3一样。
- -dvld.save_dir 指定文件输出的路径， 默认路径为/tmp。
- -dvld.save_paths 控制是否输出文件， 默认为0， 表示不输出文件
- -dvld.dump_paths 控制输出的内容， 现在只有0和1两种情况， 默认为1， 输出内容

附录D 怎样为PHP贡献

既然你在阅读本书，那说明你也是对PHP很感兴趣的读者，在窥探到PHP内部实现之后 或许也蠢蠢欲动想要共享自己的力量。下面进行一些简单的说明。

很多人以为为PHP做贡献(contribute)只是简单的为PHP提交补丁，其实在广义上来说，为PHP做贡献有很多种方式，这包含不限于： - 宣传和参与PHP的讨论 - 发现和报告或者修复PHP的bug - 编写和翻译PHP手册 - 编写PHP相关的书籍 - 写PHP相关技术的博客 - 为PHP增加新功能 - 编写和贡献PHP扩展或者库

所以很可能大部分的读者目前已经是在为PHP做贡献了。只不过如果你是本书的读者，可能更想为PHP-Runtime做贡献，比如：修复PHP代码的bug，提交功能改进。我们可能根据自己的特长来为PHP做贡献，如果你英语好，那么翻译手册将会是你的强项。如果你的C比较好，那么可以为PHP修改bug，如果你对PHP语言的语法或者功能有改进想法，你可以提交改进方法，当然如果你能将该功能实现出来那更好不过了。

下面介绍一下，为PHP做贡献的方方面面。

沟通方式

邮件组

IRC

报告和修复Bug

原则

修复

贡献功能

RFC

贡献PECL扩展

贡献Pear库

Composer

改进和增加文档

代码权限申请

关于Credit

附录E phpt测试文件说明

phpt文件用于PHP的自动化测试，这是PHP用自己来测试自己的测试数据用例文件。测试脚本通过执行PHP源码根目录下的run-tests.php，读取phpt文件执行测试。

phpt文件包含 TEST，FILE，EXPECT 等多个段落的文件。在各个段落中，TEST、FILE、EXPECT 是基本的段落，每个测试脚本都必须至少包括这三个段落。其中：

- TEST段可以用来填写测试用例的名字。
- FILE段是一个 PHP 脚本实现的测试用例。
- EXPECT段则是测试用例的期待值。

在这三个基本段落之外，还有多个段落，如作为用例输入的GET、POST、COOKIE等，此类字段最终会赋值给$env变量。比如，cookie存放在$env['HTTP_COOKIE']，$env变量将作为用例中脚本的执行环境。一些主要段落说明如下表所示：

PHP测试脚本中的段落说明

段落名	填充内容	备注
TEST	测试用例名称	必填段落
FILE	测试脚本语句	必填段落。用PHP语言书写的脚本语句。其执行的结果将与 EXPECT* 段的期待结果做对比。
ARGS	FILE 段的输入参数	选填段落
SKIPIF	跳过这个测试的条件	选填段落
POST	传入测试脚本的 POST 变量	选填段落。如果使用POST段，建议配合使用SKIPIF段
GET	传入测试脚本的 GET 变量	选填段落。如果使用GET段，建议配合使用SKIPIF段。
POST_RAW	传入测试脚本的POST内容的原生值	选填段落。比如在做文件上传测试时就需要使用此字段来模拟HTTP的POST请求。
COOKIE	传入测试脚本的COOKIE的值	选填段落。最常见的是将PHPSESSID的值传入。
INI	应用于测试脚本的 ini 设置	选填段落。例如 foo=bar 。其值可通过函数 ini_get(string name_entry) 获得。

ENV	应用于测试脚本的环境设置	选填段落。例如做gzip测试，则需要设置环境HTTP_ACCEPT_ENCODING=gzip。
EXPECT	测试脚本的预期结果 相当于测试文件的结果	必填段落
EXPECTF	测试脚本的预期结果	选填段落。可用函数 sscanf() 中的格式表达预期结果 EXPECT 段的变体
EXPECTREGEX	测试脚本的正则预期结果	选填段落。以正则的方式包含多个预期结果，是预期结果EXPECT段的一种变体。
EXPECTHEADERS	测试脚本的预期头部内容	选填段落.测试脚本期待HTTP头部返回，是预期结果EXPECT段的另一种格式。验证过程中会按头部的字段一一比对测试，比如zlib扩展中，如果开启zlib.output_compression，则在EXPECTHEADERS中包含Content-Encoding: gzip作为预期结果。

phpt文件只是用例文件，它还需要一个控制器来调用这些文件，以实现整个测试过程。PHP的测试控制器文件是源码根目录下的run-tests.php文件。此文件的作用是根据传入的参数，分析用例相关数据，执行测试过程。其大概过程如下：

1. 分析输入的命令行，根据参数配置相关参数，初始化各种信息。
2. 分析用例输入参数，获取需要执行的用例文件列表。PHP支持指定单文件用例执行，支持多文件用例执行，支持* .phpt多用例执行，支持* .phpt简化版本*多用例执行（相当于*.phpt）。
3. 遍历用例文件列表，执行每一个用例。对于每个用例，PHP会具体解析测试脚本中各个段落的含义，清除所有上次测试的记录与设置将准备此次的测试环境，并把各种中间文件和日志文件准备好，然后用环境变量 TEST_PHP_EXECUTABLE 指定的 PHP 可执行对象运行实际的测试语句。最后将运行后的结果和测试脚本中的预期结果（EXPECT*段）进行比较，如果比较结果一致，则测试通过；如果不一致，则测试失败，最后将结果信息一一记录到用户设置的日志文件中。
4. 生成测试结果。

这仅仅是执行的过程，除此之外，还有若干准备和清理工作，如，对上次测试遗留下的环境的清理，本次测试所必须的环境变量的读取与设置，对测试参数的解析，测试脚本名的解析，各种输出文件的准备等等

以测试脚本/tests/basic/001.phpt为例：

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

这个用例脚本只包含必填的三项。测试控制器会执行--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，如果不一致，则测试不通过，最终这个用例的测试结果会汇总会所有的测试结果集中。

附录F PHP5.4.0新功能升级解析

本篇主要从两个角度对PHP5.4的一些更新进行说明，并同时尽可能的解释做出这些变更 的具体原因。也就解释What & Why。本片并不会介绍所有的变更，只针对比较大或者 对我们的开发有影响的一些特性及变更进行说明。

新特性