深入理解PHP内核

Francis

Published
with GitBook

Table of Contents

深入理解**PHP**内核

深入理解PHP内核是一本有关PHP内部实现的开源书籍。

从环境准备到代码实现，从实现过程到细节延展，从变量、函数、对象到内存、Zend虚拟机…… 如此种种，道尽PHP之风流。

本博主根据网络资源重新精心整理。如有不足，欢迎提出建议。

Email:francissoung@126.com

Blog:Francis Soung

> **FrancisSoung**

第一章 准备工作和背景知识

第一章 准备工作和背景知识

千里之行，始于足下。

在开始理解PHP的实现之前，需要做一些准备工作，也需要了解一些背景知识。本章涉及PHP源码的下载， PHP源码的简单编译，从而得到我们的调试环境。

接下来，我们将简单描述整个PHP源码的结构以及在*nix环境和Windows环境下如何阅读源码。 最后我们介绍在阅读PHP源码过程中经常会遇到的一些语句。

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

下面，我们将介绍源码阅读环境的搭建。

第一节 环境搭建

在开始学习PHP实现之前， 我们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)

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。

```
#svn
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

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

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

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git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。

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。

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：

```
./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 -n -v
```

命令中的-n参数表示不使用php.ini文件，-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的话则执行$prefix/bin/php这个路径的php。

当然如果是安装在系统目录或者你的prefix目录在$PATH环境变量里的话，直接执行php就行了。

在只进行make而不make install时，只是编译为可执行二进制文件，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

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，例如一下的这个测试脚本/tests/basic/001。phpt：

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

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

php测试使用了proc_open()函数，这样就可以保证测试脚本和被测试脚本之间能隔离开。

PHP源码阅读方法 使用VIM + Ctags查看源码

通常在Linux或其他*Nix环境我们都使用VIM作为代码编辑工具，在纯命令终端下，它几乎是无可替代的。

它具有非常强大的扩展机制，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。

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。

"##"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。

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。

#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。

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
        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

        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
        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
```

```
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
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

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

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

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
        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

        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。

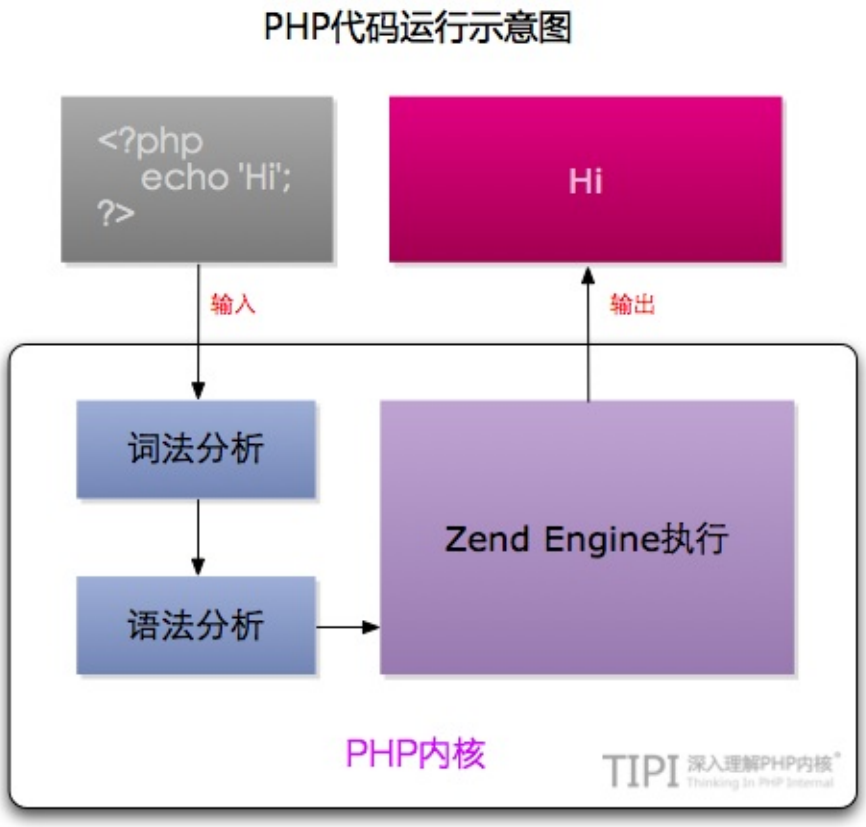
PHP语言经过许多人多年的淬炼，性能不断优化，支持的语法现象与各种特性也越来越多。导致PHP内核的代码中，涉及知识面比较广泛，具体实现也非常复杂， 从脚本的编译解析到执行以及和Web服务器等的配合，内存管理，语法实现等等。 为了不过早陷入细节的沼泽，我们先从整体上来接触PHP的实现， 先对PHP的整体结构，生命周期，PHP与其它容器（如Apache）的交互， PHP的整个执行过程等进行一个大概的了解，从而有一个整体的概念。

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：



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(RSHUWDOWN，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：

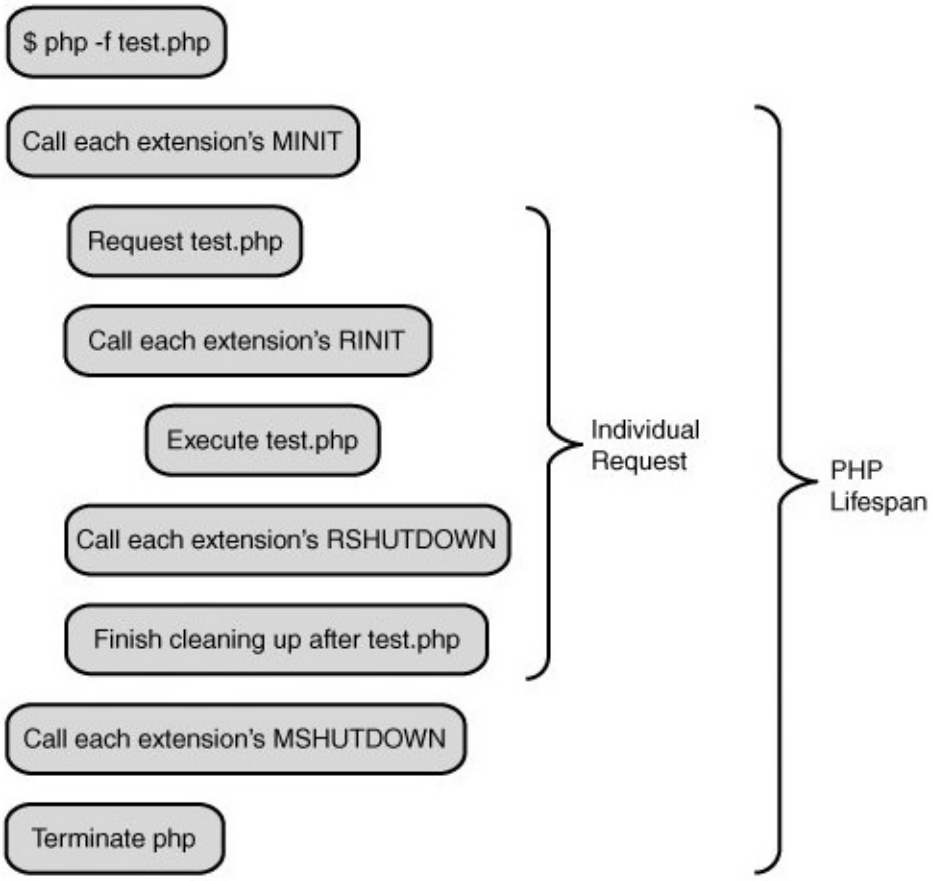


图2.1 单进程SAPI生命周期

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：

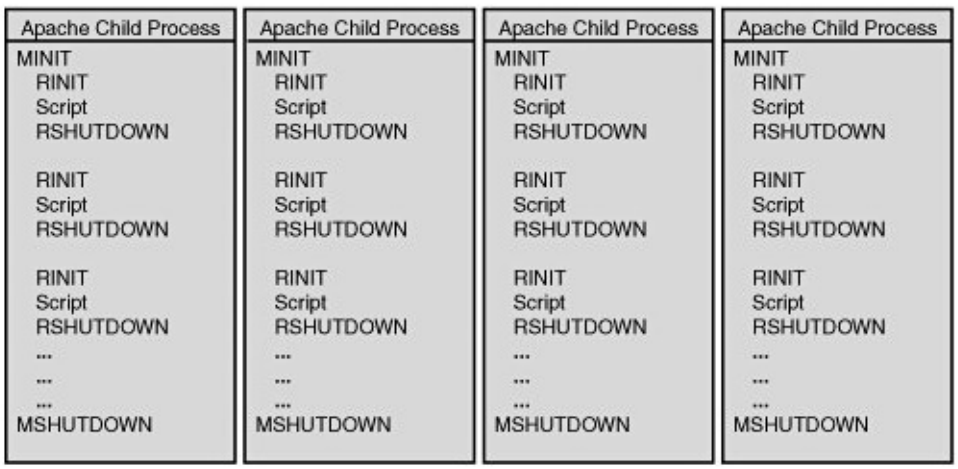


图2.2 多进程SAPI生命周期

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

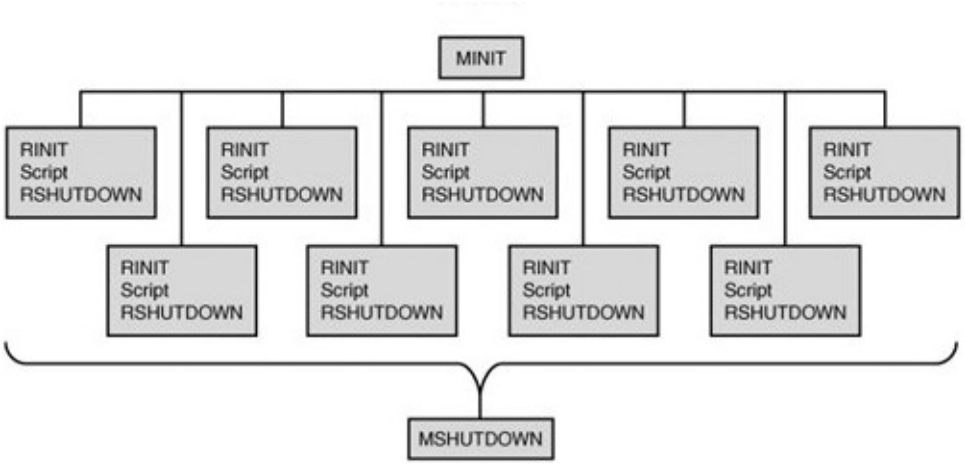


图2.3 多线程SAPI生命周期

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。

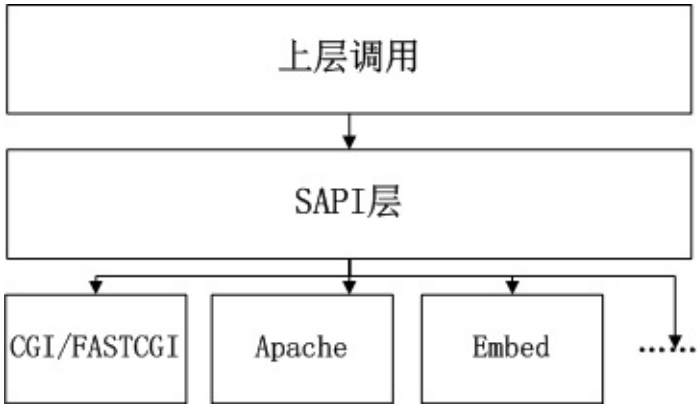


图2.4 SAPI的简单示意图

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：

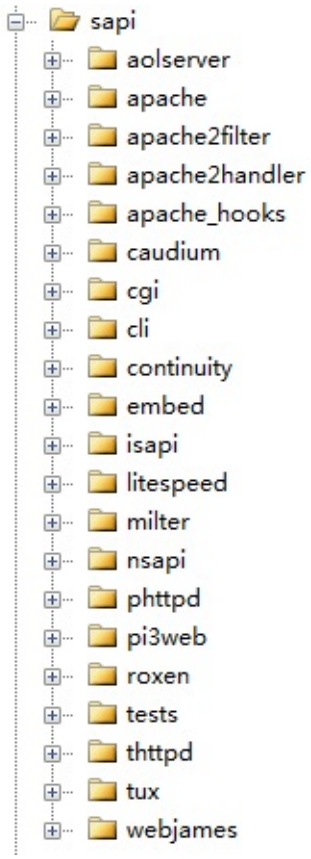


图2.5 SAPI文件结构图

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。

第三节 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*modphp5.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》

第四节 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》

第五节 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：

- Web Server启动时载入FastCGI进程管理器（IIS ISAPI或Apache Module)
- 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。 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。
- 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：

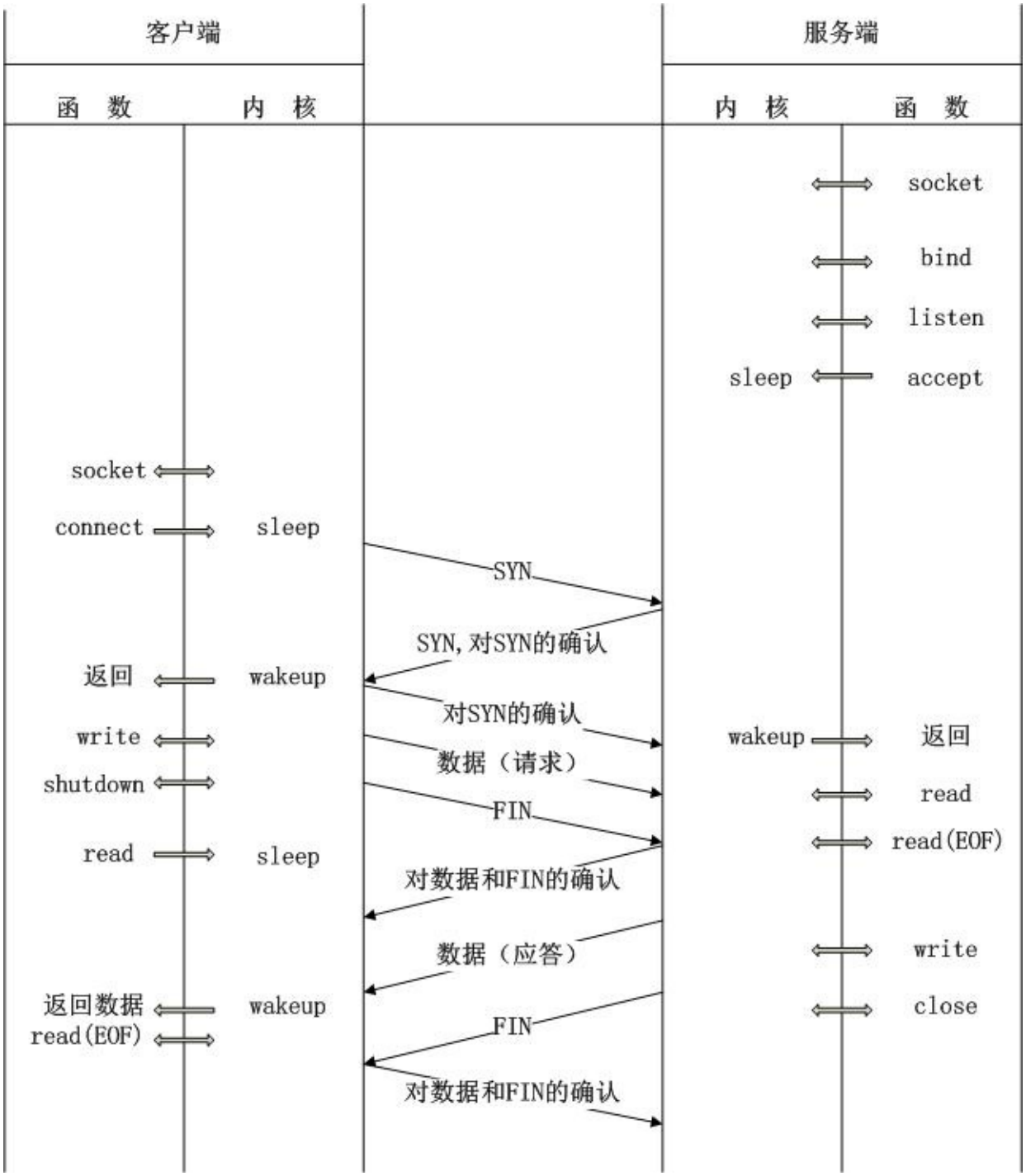


图2.6 TCP上客户-服务器事务的时序

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：

将代码切分为一个个的标记(token)。 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);
                case 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->op

     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。