目录

译序

本书目前在github上由laruence(http://www.laruence.com)和walu(http://www.walu.cc)两位大牛组织翻译. 该翻译项目地址为: https://github.com/walu/phpbook

本书在github上的地址: https://github.com/goosman-lei/php-eae

未来本书将可能部分合并到phpbook项目中, 同时保留一份独立版本.

原书名: <Extending and Embedding PHP>
原作者: Sara Golemon
译者: goosman.lei(雷果国)
译者Email: lgg860911@yahoo.com.cn
译者Blog: http://blog.csdn.net/lgg201

权利声明

此译本前世今生

最早是11年初看这本书, 当时C语言以及*NIX系统编程功力不足, 看到第11章左右就停工了, 译者看英文书籍有一个习惯就是按照自己的理解原文翻译记录电子笔记.

停工后译者一直在补充C语言, 网络以及*NIX系统编程方面的基本功, 12年底在github看到了本书的翻译项目, 遂准备github上的译本为辅助材料再学习一遍此书, 过程中发现该项目翻译章节有所缺失, 且叙述风格偏俏皮, 不合译者口味.

因此, 补充该项目中关于线程安全层缺失的一部分后, 自己重又拿起原著阅读, 并重新对照翻译出这份译本.

由于译者自身英语水平的问题(大学英语平均40分左右), 以及自身技术功力不足的问题, 译本中所述内存可能与原著存在偏颇, 请读者选择阅读时, 尽量以原著为主, 以此译本为辅.

如果读者与译者一样, 在英文阅读上有困难, 可以学习译者的阅读方法: 搭建好完善的开发环境, 从原著中不能理解的部分, 提取关键字, 通过搜索引擎分析出所述知识点(如果直接能看出当然就不需要了), 通过自己编写代码实现并验证的方式去阅读学习.

此译本核心目的是让那些和我一样英文阅读有障碍, 又想学习php模块开发的同学, 将它作为自己啃原著时的一份参考, 提高学习效率.

译本使用的环境

操作系统:

```
Linux linux.centos 2.6.32-279.5.2.el6.x86_64 #1 SMP Fri Aug 24 01:07:11 UTC 2012 x86_64 x86_64
x86_64 GNU/Linux
```

php源代码版本:

```
php-5.4.9
```

php源代码编译参数:

```
'./configure'  '--disable-all' '--prefix=/usr/local/php-dev' '--enable-embed' '--enable-debug'
'--enable-maintainer-zts' '--enable-fpm' '--with-readline' '--enable-sockets' '--with-pear' '--
enable-xml' '--enable-libxml' '--enable-tokenizer'
```

php编译安装后的版本信息:

```
PHP 5.4.9 (cli) (built: Dec 25 2012 15:12:52) (DEBUG)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2012 Zend Technologies
```

php的生命周期

在常见的webserver环境中, 你不能直接启动php解释器; 一般是启动apache或其他webserver, 由它们加载php处理需要处理的脚本(请求的.php文档).

一切都从sapi开始

尽管看起来有所不同, 但实际上CLI的行为和web方式一致. 在命令行中键入php命令将启动"命令行sapi", 它实际上就像一个设计用于服务单请求的迷你版webserver. 当脚本运行完成后, 这个迷你的php-webserver终止并返回控制给shell.

启动和终止

这里的启动和终止过程分为两个独立的启动阶段和两个独立的终止阶段. 一个周期用于php解释器整体执行所需结构和值的初始化设置, 它们在sapi生命周期中持久存在. 另一个则仅服务于单页面请求, 生命周期短暂一些.

初始化启动在所有的请求发生之前, php调用每个扩展的MINIT(模块初始化)方法. 这里, 扩展可能会定义常量, 定义类, 注册资源, 流, 过滤处理器等所有将要被请求脚本所使用的资源. 所有这些都有一个特性, 就是它们被设计跨所有请求存在, 也可以称为"持久".

常见的MINIT方法如下:

```
/* 初始化myextension模块
 * 这在sapi启动后将立即发生
 */
PHP_MINIT_FUNCTION(myextension)
{
    /* 全局: 第12章 */

#ifdef ZTS
    ts_allocate_id(&myextension_globals_id,
        sizeof(php_myextension_globals),
        (ts_allocate_ctor) myextension_globals_ctor,
        (ts_allocate_dtor) myextension_globals_dtor);
#else
    myextension_globals_ctor(&myextension_globals TSRMLS_CC);
#endif


    /* REGISTER_INI_ENTRIES() 指向一个全局的结构, 我们将在第13章"INI设置"中学习 */
    REGISTER_INI_ENTRIES();


    /* 等价于define('MYEXT_MEANING', 42); */
    REGISTER_LONG_CONSTANT("MYEXT_MEANING", 42, CONST_CS | CONST_PERSISTENT);
    /* 等价于define('MYEXT_FOO', 'bar'); */
    REGISTER_STRING_CONSTANT("MYEXT_FOO", "bar", CONST_CS | CONST_PERSISTENT);


    /* 资源: 第9章 */
    le_myresource = zend_register_list_destructors_ex(
                    php_myext_myresource_dtor, NULL,
                    "My Resource Type", module_number);
    le_myresource_persist = zend_register_list_destructors_ex(
                    NULL, php_myext_myresource_dtor,
                    "My Resource Type", module_number);


    /* 流过滤器: 第16章 */
    if (FAILURE == php_stream_filter_register_factory("myfilter",
```

```
                    &php_myextension_filter_factory TSRMLS_CC)) {
        return FAILURE;
    }


    /* 流包装器: 第15章 */
    if (FAILURE == php_register_url_stream_wrapper ("myproto",
                    &php_myextension_stream_wrapper TSRMLS_CC)) {
        return FAILURE;
    }


    /* 自动全局变量: 第12章 */
#ifdef ZEND_ENGINE_2
    if (zend_register_auto_global("_MYEXTENSION", sizeof("_MYEXTENSION") - 1,
                                                NULL TSRMLS_CC) == FAILURE) {
        return FAILURE;
    }
    zend_auto_global_disable_jit ("_MYEXTENSION", sizeof("_MYEXTENSION") - 1
                                                    TSRMLS_CC);
#else
    if (zend_register_auto_global("_MYEXTENSION", sizeof("_MYEXTENSION") - 1
                                                    TSRMLS_CC) == FAILURE) {
        return FAILURE;
    }
#endif
    return SUCCESS;
}
```

在一个请求到达时, php会安装一个操作环境, 该环境包含符号表(变量存储), 并且会同步每个目录的配置值. php接着遍历所有的扩展, 这一次调用每个扩展的RINIT(请求初始化)方法. 这里, 扩展可能充值全局变量到默认值, 预置变量到脚本的符号表, 或执行其他的任务比如记录页面请求日志到文件. RINIT之于所有脚本请求就像auto_prepend_file指令一样.

RINIT方法的写法如下:

```
/* 每个页面请求开始之前执行
 */
PHP_RINIT_FUNCTION(myextension)
{
    zval *myext_autoglobal;


    /* 初始化MINIT函数中定义的自动全局变量为空数组. 这等价于$_MYEXTENSION = array(); */
    ALLOC_INIT_ZVAL(myext_autoglobal);
    array_init(myext_autoglobal);
    zend_hash_add(&EG(symbol_table), "_MYEXTENSION", sizeof("_MYEXTENSION") - 1,
                                (void**)&myext_autoglobal, sizeof(zval*), NULL);


    return SUCCESS;
}
```

在一个请求完成处理后(到达脚本文件末尾或调用了die()/exit()语句), php通过调用每个扩展的RSHUTDOWN(请求终止)开始清理过程. 就像RINIT对应于auto_prepend_file, RSHUTDOWN可以类比auto_append_file指令. 在RSHUTDOWN和auto_append_file之间, 最重要的不同是: 无论如何, RSHUTDOWN总会被执行, 而用户空间脚本的die()/exit()调用会跳过所有的auto_append_file.

在符号表和其他资源释放之前所需要做的最后一件事旧是RSHUTDOWN. 在所有的RSHUTDOWN方法完成后, 符号表中的所有变量都会被立即unset(),

在此期间, 所有非持久化资源和对象的析构器都将被调用去优雅的释放资源.

```
/* 每个页面请求结束后调用 */
```

```
PHP_RSHUTDOWN_FUNCTION(myextension)
{
    zval **myext_autoglobal;


    if (zend_hash_find(&EG(symbol_table), "_MYEXTENSION", sizeof("_MYEXTENSION"),
                                        (void**)&myext_autoglobal) == SUCCESS) {
        /* 做一些对$_MYEXTENSION数组的值有意义的处理 */
        php_myextension_handle_values(myext_autoglobal TSRMLS_CC);
    }
    return SUCCESS;
}
```

最后, 当所有的请求都被满足(完成处理)后, webserver或其他sapi就开始准备终止, php循环执行每个扩展的MSHUTDOWN(模块终止)方法. 这是MINIT周期内, 扩展最后一次卸载处理器和释放持久化分配的内存的机会.

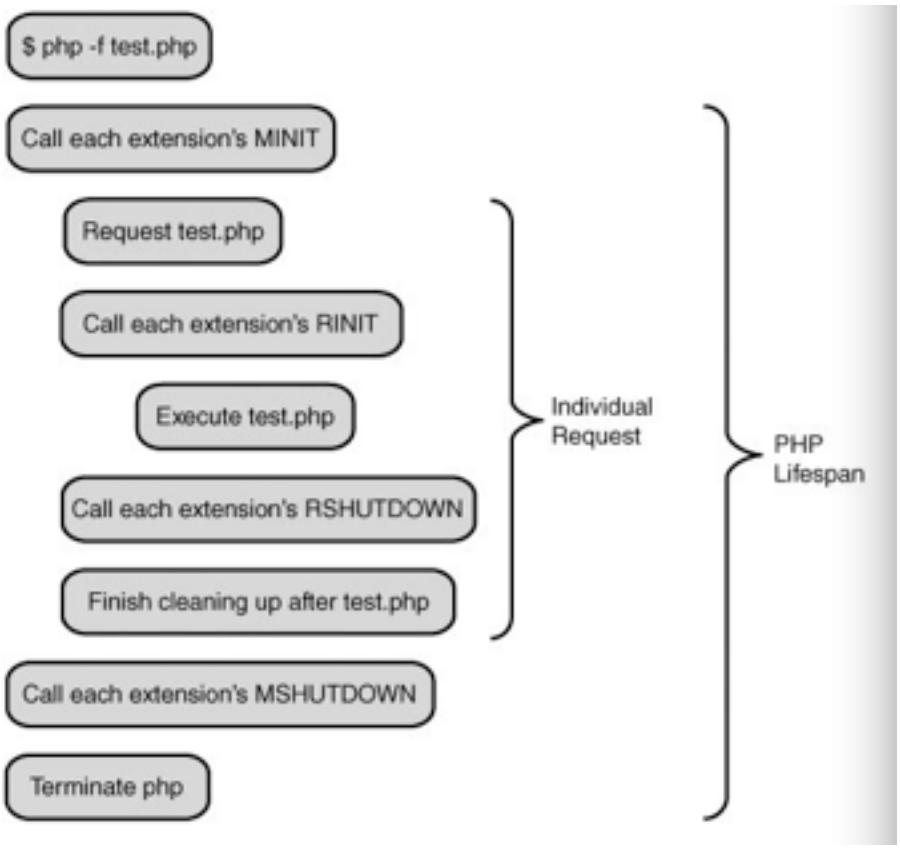
```
/* 这个模块正在被卸载, 常量和函数将被自动的卸载, 持久化资源, 类, 流处理器必须手动的卸载. */
PHP_MSHUTDOWN_FUNCTION(myextension)
{
    UNREGISTER_INI_ENTRIES();
    php_unregister_url_stream_wrapper ("myproto" TSRMLS_CC);
    php_stream_filter_unregister_factory ("myfilter" TSRMLS_CC);
    return SUCCESS;
}
```

生命周期

每个php实例, 无论从init脚本启动还是从命令行启动, 接下来都是上一节讲到的一系列请求/模块的初始化/终止事件, 以及脚本自身的执行. 每个启动和终止阶段会被执行多少次? 以什么频率执行? 都依赖于所使用的sapi. 下面讨论4种最常见的sapi: cli/cgi, 多进程模块, 多线程模块, 嵌入式.

cli生命周期

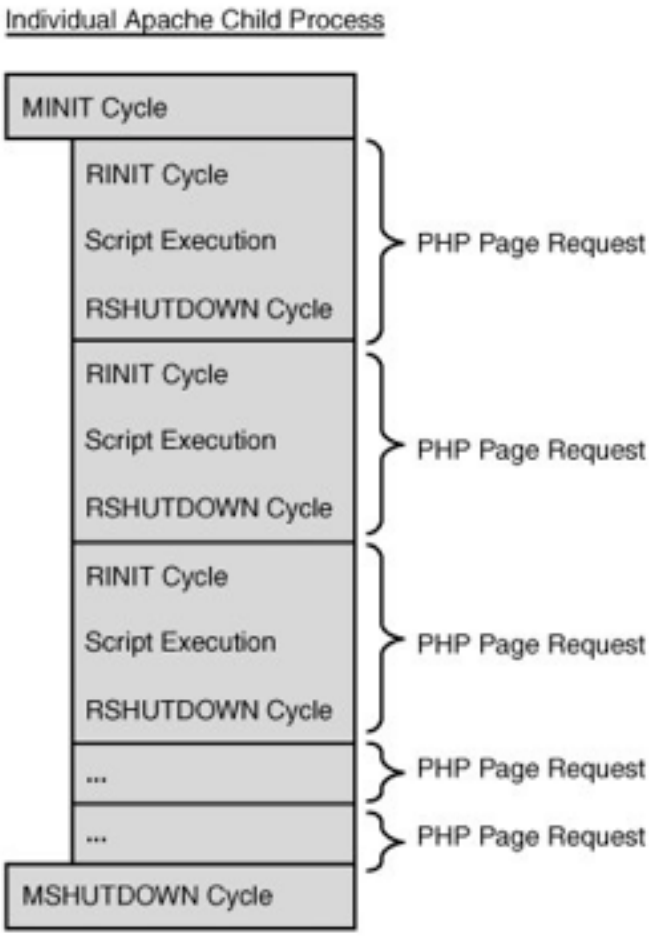
cli(和cgi)sapi在它的单请求生命周期中相当特殊, 因为此时整个php的生命周期只有一个请求. 不过, 麻雀虽小五脏俱全, 前面讲的各个阶段仍然会全部执行. 下图是从命令行调用php解释器处理test.php脚本的处理过程:



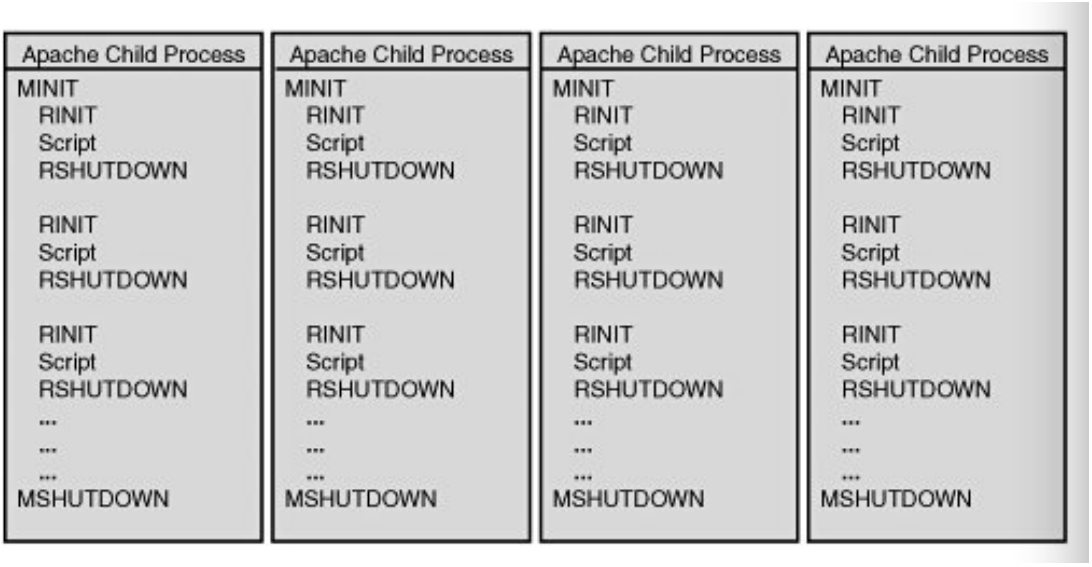
多进程生命周期

php最常见的用法是将php构建为apache 1或使用pre-fork MPM的apache 2的apxs模块, 将php嵌入到webserver中. 还有一些其他webserver也是属于这一类的, 本书后面将它们统称为"多进程模块".

将它们称为多进程模块是因为当apache启动时, 它会立即fork出一些子进程, 每个都有自己的独立的进程空间, 互相之间独立. 在一个子进程中, php实例的生命周期就像下图所示一样. 这里唯一的变化是多个请求被夹在在单个的MINIT/MSHUTDOWN对中:

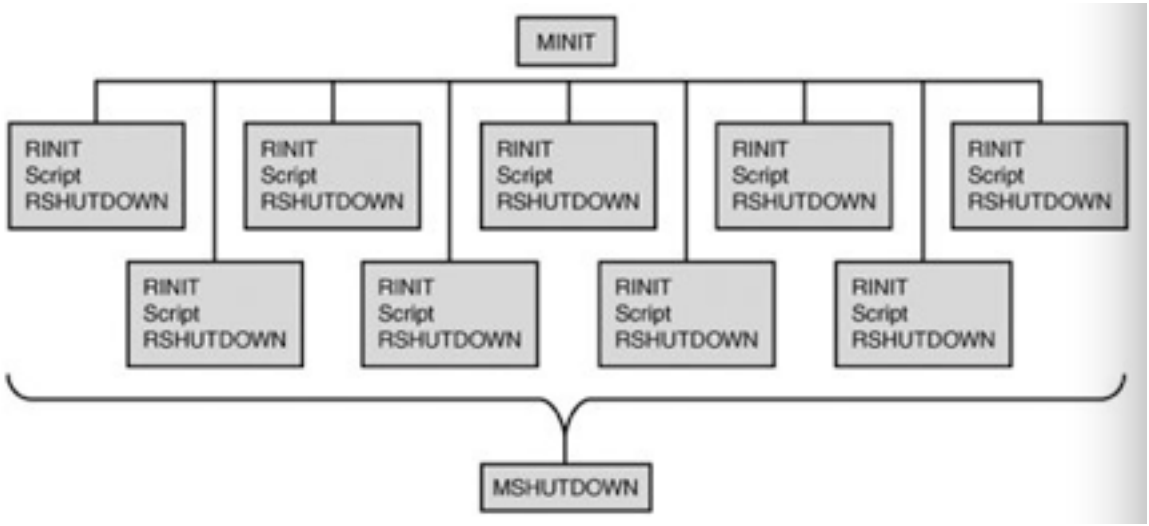


这种模式不允许任意子进程知道其他子进程拥有的数据, 不过它允许子进程死亡并被替换, 而不影响其他子进程的稳定性. 下图战士了一个apache实例中的多个进程以及它们对MINIT, RINIT, RSHUTDOWN, MSHUTDOWN方法的调用.



多线程生命周期

随着发展, php逐渐的被一些webserver以多线程方式使用, 比如IIS的isapi接口, apapche 2的worker mpm. 在多线程webserver中永远都只有一个进程在运行, 但是在进程空间中有多个线程同时执行. 这样做能降低一些负载, 包括避免了MINIT/MSHUTDOWN的重复调用, 真正的全局数据只被分配和初始化一次, 潜在的打开了多个请求的信息共享之门. 下图展示了apache 2这样的多线程webserver上运行php时的进程状态:



嵌入式生命周期

回顾前面, 嵌入式sapi只是sapi的另外一种实现, 它还是遵循和cli, apxs, isapi接口一致的规则, 因此很容易猜到请求的生命周期遵循相同的基本路径: 模块初始化 => 请求初始化 => 请求 => 请求终止 => 模块终止. 实际上, 嵌入式sapi和它的同族一样遵循着这些步骤.

让嵌入式sapi变得特殊的是它可能被当做一个整个请求的一部分被潜入到多个脚本片段中. 多数情况下控制会在php和调用应用之间多次来回的传递.

虽然一个嵌入请求可能由一个或多个代码元素组成, 但嵌入式应用还是受和webserver一样的请求隔离影响. 为了处理两个或多个并行的嵌入环境, 你的应用要么像apache 1去fork, 要么旧像apache 2线程化. 尝试在单个非线程进程空间中处理两个独立的请求环境将产生不可预料的结果, 这肯定是你不期望的.

Zend线程安全

当php还在幼儿期的时候, 它作为一个单进程cgi运行, 并没有线程安全的概念, 因为没有比单个请求存活更久的进程空间. 内部变量可以在全局作用域中定义, 访问, 修改, 只要初始化没有问题就不会产生严重后果. 任何没有被正确清理的资源都会在cgi进程终止时被释放.

后来, php嵌入了多进程webserver, 比如apache. 给定的内部变量仍然可以定义在全局并且可以通过在每个请求启动时正确的初始化, 终止时去做适当的清理工作来做到安全访问, 因为在一个进程空间中同时只会有一个请求. 这个时候, 增加了每个请求的内存管理, 以放置资源泄露的增长失去控制.

单进程多线程webserver出现后, 就需要一种对全局数据处理的新的方法. 最后这作为新的一层TSRM(线程安全资源管理)

线程安全Vs.废线程安全定义

在一个简单的非线程应用中, 你可能很喜欢定义全局变量, 将它们放在你的源代码的顶部. 编译器会在你的程序的数据段分配内存块保存信息.

在多线程应用中, 每个线程需要它自己的数据元素, 需要为每个线程分配独立的内存块. 一个给定线程在它需要访问自己的数据时需要能够正确的访问到自己的这个内存块.

线程安全数据池

在一个扩展的MINIT阶段, 扩展可以调用ts_allocate_id()一次或多次告诉TSRM层它需要多少数据空间, TSRM接收到通知后, 将总的运行数据空间增大请求的字节数, 并返回一个新的唯一的标识, 标记线程数据池的数据段部分.

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
int sample_globals_id;
PHP_MINIT_FUNCTION(sample)
{
    ts_allocate_id(&sample_globals_id,
        sizeof(php_sample_globals),
        (ts_allocate_ctor) php_sample_globals_ctor,
        (ts_allocate_dtor) php_sample_globals_dtor);
    return SUCCESS;
}
```

当一个请求需要访问数据段的时候，扩展从TSRM层请求当前线程的资源池，以ts_allocate_id()返回的资源ID来获取偏移量。

换句话说，在代码流中，你可能会在前面所说的MINIT语句中碰到SAMPLE_G(sampleint) = 5;这样的语句。在线程安全的构建下，这个语句通过一些宏扩展如下:

```
(((php_sample_globals*)(*((void ***)tsrm_ls))[sample_globals_id-1])->sampleint = 5;
```

如果你看不懂上面的转换也不用沮丧，它已经很好的封装在PHPAPI中了，以至于许多开发者都不需要知道它怎样工作的。

当不在线程环境时

因为在PHP的线程安全构建中访问全局资源涉及到在线程数据池查找对应的偏移量，这是一些额外的负载，结果就是它比对应的非线程方式（直接从编译期已经计算好的真实的全局变量地址中取出数据）慢一些。

考虑上面的例子，这一次在非线程构建下:

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
php_sample_globals sample_globals;
PHP_MINIT_FUNCTION(sample)
{
    php_sample_globals_ctor(&sample_globals TSRMLS_CC);
    return SUCCESS;
}
```

首先注意到的是这里并没有定义一个int型的标识去引用全局的结构定义，只是简单的在进程的全局空间定义了一个结构体。也就是说SAMPLE_G(sampleint) = 5;展开后就是sample_globals.sampleint = 5; 简单，快速，高效。

非线程构建还有进程隔离的优势，这样给定的请求碰到完全出乎意料的情况时，它也不会影响其他进程，即便是产生段错误也不会导致整个webserver瘫痪。实际上，Apache的MaxRequestsPerChild指令就是设计用来提升这个特性的，它经常性的有目的性的kill掉子进程并产生新的子进程，来避免某些可能由于进程长时间运行“累积”而来的问题（比如内存泄露）。

访问全局变量

在创建一个扩展时，你并不知道它最终的运行环境是否是线程安全的。幸运的是，你要使用的标准包含文件集合中已经包含了条件定义的ZTS预处理标记。当PHP因为SAPI需要或通过enable-maintainer-zts选项安装等原因以线程安全方式构建时，这个值会被自动的定义，并可以用一组#ifdef ZTS这样的指令集去测试它的值。

就像你前面看到的，只有在PHP以线程安全方式编译时，才会存在线程安全池，只有线程安全池存在时，才会真的在线程安全池中分配空间。这就是为什么前面的例子包裹在ZTS检查中的原因，非线程方式供非线程构建使用。

在本章前面PHP_MINIT_FUNCTION(myextension)的例子中，你可以看到#ifdef ZTS被用作条件调用正确的全局初始代码。对于ZTS模式它使用ts_allocate_id()弹出myextension_globals_id变量，而非ZTS模式只是直接调用myextension_globals的初始化方法。这两个变量已经在你的扩展源文件中使用Zend宏：DECLARE_MODULE_GLOBALS(myextension)声明，它将自动的处理对ZTS的测试并依赖构建的ZTS模式选择正确的方式声明。

在访问这些全局变量的时候，你需要使用前面给出的自定义宏SAMPLE_G()。在第12章，你将学习到怎样设计这个宏以使它可以依赖ZTS模式自动展开。

即便你不需要线程也要考虑线程

正常的PHP构建默认是关闭线程安全的，只有在被构建的sapi明确需要线程安全或线程安全在./configure阶段显式的打开时，才会以线程安全方式构建。

给出了全局查找的速度问题和进程隔离的缺点后，你可能会疑惑为什么明明不需要还有人故意打开它呢？这是因为，多数情况下，扩展和SAPI的开发者认为你是线程安全开关的操作者，这样做可以很大程度上确保新代码可以在所有环境中正常运行。

当线程安全启用时，一个名为tsrm_ls的特殊指针被增加到了很多的内部函数原型中。这个指针允许PHP区分不同线程的数据。回想一下本章前面ZTS模式下的SAMPLE_G()宏函数中就使用了它。没有它，正在执行的函数就不知道查找和设置哪个线程的符号表；不知道应该执行哪个脚本，引擎也完全无法跟踪它的内部寄存器。这个指针保留了线程处理的所有页面请求。

这个可选的指针参数通过下面一组定义包含到原型中。当ZTS禁用时，这些定义都被展开为空；当ZTS开启时，它们展开如下：

```
#define TSRMLS_D     void ***tsrm_ls
#define TSRMLS_DC     , void ***tsrm_ls
#define TSRMLS_C     tsrm_ls
#define TSRMLS_CC     , tsrm_ls
```

非ZTS构建对下面的代码看到的是两个参数：int, char *。在ZTS构建下，原型则包含三个参数：int, char *, void ***。当你的程序调用这个函数时，只有在ZTS启用时才需要传递第三个参数。下面代码的第二行展示了宏的展开：

```
int php_myext_action(int action_id, char *message TSRMLS_DC);
php_myext_action(42, "The meaning of life" TSRMLS_CC);
```

通过在函数调用中包含这个特殊的变量，php_myext_action就可以使用tsrm_ls的值和MYEXT_G()宏函数一起访问它的线程特有全局数据。在非ZTS构建上，tsrm_ls将不可用，但是这是ok的，因为此时MYEXT_G()宏函数以及其他类似的宏都不会使用它。

现在考虑，你在一个新的扩展上工作，并且有下面的函数，它可以在你本地使用CLI SAPI的构建上正常工作，并且即便使用apache 1的apxs SAPI编译也可以正常工作：

```
static int php_myext_isset(char *varname, int varname_len)
{
    zval **dummy;


    if (zend_hash_find(EG(active_symbol_table),
        varname, varname_len + 1,
        (void**)&dummy) == SUCCESS) {
        /* Variable exists */
        return 1;
    } else {
        /* Undefined variable */
        return 0;
    }
}
```

所有的一切看起来都工作正常，你打包这个扩展发送给他人构建并运行在生产服务器上。让你气馁的是，对方报告扩展编译失败。

事实上它们使用了Apache 2.0的线程模式，因此它们的php构建启用了ZTS。当编译期碰到你使用的EG()宏函数时，它尝试在本地空间查找tsrm_ls没有找到，因为你并没有定义它并且没有在你的函数中传递。

修复这个问题非常简单；只需要在php_myext_isset()的定义上增加TSRMLS_DC，并在每行调用它的地方增加TSRMLS_CC。不幸的是，现在对方已经有点不信任你的扩展质量了，这样就会推迟你的演示周期。这种问题越早解决越好。

现在有了enable-maintainer-zts指令。通过在./configure时增加该指令来构建php，你的构建将自动的包含ZTS，哪怕你当前的SAPI（比如CLI）不需要它。打开这个开关，你可以避免这些常见的不应该出现的错误。

注意：在PHP4中，enable-maintainer-zts标记等价的名字是enable-experimental-zts；请确认使用你的php版本对应的正确标记。

寻回丢失的tsrm_ls

有时，我们需要在一个函数中使用tsrm_ls指针，但却不能传递它。通常这是因为你的扩展作为某个使用回调的库的接口，它并没有提供返回抽象指针的地方。考虑下面的代码片段：

```
void php_myext_event_callback(int eventtype, char *message)
{
    zval *event;


    /* $event = array('event'=>$eventtype,
                      'message'=>$message) */
    MAKE_STD_ZVAL(event);
    array_init(event);
    add_assoc_long(event, "type", eventtype);
    add_assoc_string(event, "message", message, 1);


    /* $eventlog[] = $event; */
    add_next_index_zval(EXT_G(eventlog), event);
}
PHP_FUNCTION(myext_startloop)
{
    /* The eventlib_loopme() function,
     * exported by an external library,
     * waits for an event to happen,
```

```
     * then dispatches it to the
     * callback handler specified.
     */
    eventlib_loopme(php_myext_event_callback);
}
```

虽然你可能不完全理解这段代码，但你应该注意到了回调函数中使用了EXT_G()宏函数，我们知道在线程安全构建下它需要tsrm_ls指针。修改函数原型并不好也不应该这样做，因为外部的库并不知道php的线程安全模型。那这种情况下怎样让tsrm_ls可用呢?

解决方案是前面提到的名为TSRMLS_FETCH()的Zend宏函数。将它放到代码片段的顶部，这个宏将执行给予当前线程上下文的查找，并定义本地的tsrm_ls指针拷贝。

这个宏可以在任何地方使用并且不用通过函数调用传递tsrm_ls，尽管这看起来很诱人，但是，要注意到这一点：TSRMLS_FETCH调用需要一定的处理时间。这在单次迭代中并不明显，但是随着你的线程数增多，随着你调用TSRMLS_FETCH()的点的增多，你的扩展就会显现出这个瓶颈。因此，请谨慎的使用它。

注意：为了和c++编译器兼容，请确保将TSRMLS_FETCH()和所有变量定义放在给定块作用域的顶部（任何其他语句之前）。因为TSRMLS_FETCH()宏自身有多种不同的解析方式，因此最好将它作为变量定义的最后一行。

小结

本章中主要是对后续章节将要解释的各种概念的一个概览. 你还应该对整件事建立了基础的认识, 它不只是要构建扩展, 还有幕后的Zend引擎和TSRM层, 它们将使你在将php嵌入到你的应用时获利.

变量的里里外外

每种编程语言共有的一个特性是存储和取回信息; php也不例外. 虽然许多语言要求所有的变量都要在使用之前被定义, 并且它们的类型信息是固定的, 然而php允许程序员在使用的时候创建变量, 并且可以存储任意类型语言能够表达的信息. 并且还可以在需要的时候自动的转换变量类型.

因为你已经使用过用户空间的php, 因此你应该知道这个概念是"弱类型". 本章, 你将看到这些信息在php的父语言----c(C的类型是严格的)中是怎样编码的.

当然, 数据的编码只是一半工作. 为了保持对所有这些信息片的跟踪, 每个变量还需要一个标签和一个容器. 从用户空间角度来看, 你可以把它们看做是变量名和作用域的概念.

数据类型

php中的数据存储单位是zval, 也称作Zend Value. 它是一个只有4个成员的结构体, 在Zend/zend.h中定义, 格式如下:

```
typedef struct _zval_struct {
    zval_value    value;
    zend_uint     refcount;
    zend_uchar    type;
    zend_uchar    is_ref;
} zval;
```

我们可以凭直觉猜想到这些成员中多数的基础存储类型: unsigned integer的refcount, unsigned character的type和is_ref. 而value成员实际上是一个定义为union的结构, 在php5中, 它定义如下:

```
typedef union _zvalue_value {
    long                    lval;
    double                  dval;
    struct {
        char     *val;
        int      len;
    }                       str;
    HashTable               *ht;
    zend_object_value       obj;
} zvalue_value;
```

union允许Zend使用一个单一的, 统一的结构来将许多不同类型的数据存储到一个php变量中.

zend当前定义了下表列出的8种数据类型:

类型值	目的
IS_NULL	这个类型自动的赋值给未初始化的变量, 直到它第一次被使用. 也可以在用户空间使用内建的NULL常量进行显式的赋值. 这个变量类型提供了一种特殊的"没有数据"的类型, 它和布尔的FALSE以及整型的0有所不同.
IS_BOOL	布尔变量可以有两种可能状态中的一种, TRUE/FALSE. 用户空间控制结构if/while/ternary/for等中间的条件表达式在评估时都会隐式的转换为布尔类型.

类型值	目的
IS_DOUBLE	浮点数据类型, 使用主机系统的signed double数据类型. 浮点数并不是以精确的精度存储的; 而是用一个公式表示值的小数部分的有限精度(译注: 浮点数被表示为3部分: 符号, 尾数--小数部分, 指数. 浮点数的值 = 符号 * 尾数 * 2 ^ 指数----来自BSD Library Functions Manual: float(3)). 这种计数法允许计算机存储很大范围的值(正数或负数): 用8字节就可以表示2.225*10^(-308)到1.798*10^(308)范围内的数字. 不幸的是它评估的数字实际的十进制并不能总是像二进制分数一样干净的存储. 例如, 十进制表达式0.5转换为二进制的精确值是0.1, 然而十进制的0.8转换为二进制则是无限循环的0.1100110011..., 当它转换回十进制时, 因为无法存储被丢弃的二进制位将无法恢复. 类似的可以想一下将1/3转换为十进制的0.333333, 两个值非常相近, 但是它不精确, 因为3 * 0.333333并不等于1.0. 这个不精确常常会在计算机上处理浮点数时让人迷惑.(这些范围限制通常是基于32位平台的; 不同的系统范围可能不同)
IS_STRING	php中最常见的数据类型是字符串, 它的存储方式符合有经验的C程序员的预期. 分配一块足够大去保存字符串中所有的字节/字符的内存, 并将指向该字符串的指针保存在宿主zval中. 值得注意的是php字符串的长度总是显式的在zval结构中指出. 这就允许字符串包含NULL字节而不被截断. 关于php字符串的这一方面, 我们往后称为"二进制安全"因为这样做使得它可以安全的包含任意类型的二进制数据. 需要注意的是为一个php字符串分配的内存总量总是最小化的: 长度加1. 最后的一个字节存放终止的NULL字符, 因此不关心二进制安全的函数可以直接传递字符串指针.
IS_ARRAY	数组是一种特殊目的的变量, 它唯一的功能就是组织其他变量. 不像C中的数组概念, php的数组并不是单一类型数据的向量(比如zval arrayofzvals[];). 实际上, php的数组是一个复杂的数据桶集合, 它的内部是一个HashTable. 每个HashTable元素(桶)包含两个相应的信息片: 标签和数据. 在php数组的应用场景中, 标签就是关联数组的key或数值下表, 数据就是key指向的变量(zval)
IS_OBJECT	对象拥有数组的多元素数据存储, 此外还增加了方法, 访问修饰符, 作用域常量, 特殊的事件处理器. 作为一个扩展开发者, 构建在php4和php5中等价的面向对象代码是一个很大的挑战, 因为在Zend引擎1(php4)和Zend引擎2(php5)之间, 内部的对象模型有非常大的变更.

类型值	目的
IS_RESOURCE	有一些数据类型并不能简单的映射到用户空间. 比如, stdio的FILE指针或libmysqlclient的连接句柄, 它们不能被简单的映射为标量值的数组, 那样做它们就失去了意义. 为了保护用户空间脚本编写者不去处理这些问题, php提供了一个泛华的资源数据类型. 资源类型的实现细节我们将在第9章"资源数据类型"中涉及, 现在我们只需要知道有这么个东西就好了.

上表中的IS_*常量被存储在zval结构的type元素中, 用来确定在测试变量的值时应该查看value元素中的哪个部分.

最明显的检查一个数据的类型的方法如下代码:

```
void describe_zval(zval *foo)
{
    if (foo->type == IS_NULL) {
        php_printf("The variable is NULL");
    } else {
        php_printf("The variable is of type %d", foo->type);
    }
}
```

显而易见, 但是是错的.

好吧, 没有错, 但确实不是首选做法. Zend头文件包含了很多的zval访问宏, 它们是作者期望在测试zval数据时使用的方式. 这样做主要的原因是避免在引擎的api变更后产生不兼容问题, 不过从另一方面来看这样做还会使得代码更加易读. 下面是相同功能的代码段, 这一次使用了Z_TYPE_P()宏:

```
void describe_zval(zval *foo)
{
    if (Z_TYPE_P(foo) == IS_NULL) {
        php_printf("The variable is NULL");
    } else {
        php_printf("The variable is of type %d",
                           Z_TYPE_P(foo));
    }
}
```

这个宏的_P后缀标识传递的参数应该是一级间访的指针. 还有另外两个宏Z_TYPE()和Z_TYPE_PP(), 它们期望的参数类型是zval(非指针)和zval **(两级间访指针).

注意

在这个例子中使用了一个特殊的输出函数php_printf(), 它被用于展示数据片. 这个函数语法上等同于stdio的printf函数; 不过它对webserver sapi有特殊的处理, 使用php的输出缓冲机制提升性能. 你将在第5章"你的第一个扩展"中更多的了解这个函数以及它的同族PHPWRITE().

数据值

和类型一样, zval的值也可以用3个一组的宏检查. 这些宏总是以Z_开始, 可选的以_P或_PP结尾, 具体依赖于它们的间访层级.

对于简单的标量类型, boolean, long, double, 宏简写为: BVAL, LVAL, DVAL.

```
void display_values(zval boolzv, zval *longpzv,
                zval **doubleppzv)
{
    if (Z_TYPE(boolzv) == IS_BOOL) {
```

```
        php_printf("The value of the boolean is: %s\n",
            Z_BVAL(boolzv) ? "true" : "false");
    }
    if (Z_TYPE_P(longpzv) == IS_LONG) {
        php_printf("The value of the long is: %ld\n",
            Z_LVAL_P(longpzv));
    }
    if (Z_TYPE_PP(doubleppzv) == IS_DOUBLE) {
        php_printf("The value of the double is: %f\n",
            Z_DVAL_PP(doubleppzv));
    }
}
```

由于字符串变量包含两个成员, 因此它有一对宏分别表示char *(STRVAL)和int(STRLEN)成员:

```
void display_string(zval *zstr)
{
    if (Z_TYPE_P(zstr) != IS_STRING) {
        php_printf("The wrong datatype was passed!\n");
        return;
    }
    PHPWRITE(Z_STRVAL_P(zstr), Z_STRLEN_P(zstr));
}
```

数组数据类型内部以HashTable *存储, 可以使用: Z_ARRVAL(zv), Z_ARRVAL_P(pzv), Z_ARRVAL_PP(ppzv)访问. 在阅读旧的php内核和pecl模块的代码时, 你可能会碰到HASH_OF()宏, 它期望一个zval *参数. 这个宏等价于Z_ARRVAL_P()宏, 不过, 这个用法已经废弃, 在新的代码中应该不再被使用.

对象的内部表示结构比较复杂, 它有较多的访问宏: OBJ_HANDLE返回处理标识, OBJ_HT返回处理器表, OBJCE用于类定义, OBJPROP用于属性的HahsTable, OBJ_HANDLER用于维护OBJ_HT表中的一个特殊处理器方法. 现在不要被这么多的对象访问宏吓到, 在第10章"php4对象"和第11章"php5对象"中它们的细节都会介绍.

在一个zval中, 资源数据类型被存储为一个简单的整型, 它可以通过RESVAL这一组宏来访问. 这个整型将被传递给zend_fetch_resource()函数在已注册资源列表中查找资源对象. 我们将在第9章深入讨论资源数据类型.

数据的创建

现在你知道了怎样从一个zval中取出数据, 是时候创建一些自己的数据了. 虽然zval可以作为一个直接变量定义在函数的顶部, 这使得变量的数据存储在本地, 为了让它离开这个函数到达用户空间就需要对其进行拷贝.

因为你大多数时候都是希望自己创建的zval到达用户空间, 因此你就需要分配一个块内存给它, 并且将它赋值给一个zval *指针. 与之前的"显而易见"的方案一样, 使用malloc(sizeof(zval))并不是正确的答案. 取而代之的是你要用另外一个Zend宏: MAKE_STD_ZVAL(pzv). 这个宏将会以一种优化的方式在其他zval附近为其分配内存, 自动的处理超出内存错误(下一章将会解释), 并初始化新zval的refcount和is_ref属性.

除了*MAKE_STD_ZVAL(),* 你可能还经常会碰到其他的*zval *创建宏,* 比如*ALLOC_INIT_ZVAL().* 这个宏和*MAKE_STD_ZVAL*唯一的区别是它会将*zval *的数据类型初始化为IS_NULL.*

一旦数据存储空间可用, 就可以向你的新zval中填充一些信息了. 在阅读了前面的数据存储部分后, 你可能准备使用Z_TYPE_P()和Z_SOMEVAL_P()宏去设置你的新变量. 我们来看看这个"显而易见"的方案是否正确?

同样, "显而易见"的并不正确!

Zend暴露了另外一组宏用来设置zval *的值. 下面就是这些新的宏和它们展开后你已经熟悉的格式:

```
ZVAL_NULL(pvz);                    Z_TYPE_P(pzv) = IS_NULL;
```

虽然这些宏相比使用更加直接的版本并没有节省什么, 但它的出现体现了完整性.

```
ZVAL_BOOL(pzv, b);                 Z_TYPE_P(pzv) = IS_BOOL;
                                   Z_BVAL_P(pzv) = b ? 1 : 0;
ZVAL_TRUE(pzv);                    ZVAL_BOOL(pzv, 1);
ZVAL_FALSE(pzv);                   ZVAL_BOOL(pzv, 0);
```

注意, 任何非0值提供给ZVAL_BOOL()都将产生一个真值. 当在内部代码中硬编码时, 使用1表示真值被认为是较好的实践. 宏ZVAL_TRUE()和ZVAL_FALSE()提供用来方便编码, 有时也会提升代码的可读性.

```
ZVAL_LONG(pzv, l);                 Z_TYPE_P(pzv) = IS_LONG;
                                   Z_LVAL_P(pzv) = l;
ZVAL_DOUBLE(pzv, d);               Z_TYPE_P(pzv) = IS_DOUBLE;
                                   Z_DVAL_P(pzv) = d;
```

基础的标量宏和它们自己一样简单. 设置zval的类型, 并给它赋一个数值.

```
ZVAL_STRINGL(pzv,str,len,dup);     Z_TYPE_P(pzv) = IS_STRING;
                                   Z_STRLEN_P(pzv) = len;
                                   if (dup) {
                                       Z_STRVAL_P(pzv) =
                                             estrndup(str, len + 1);
                                   } else {
                                      Z_STRVAL_P(pzv) = str;
                                   }
ZVAL_STRING(pzv, str, dup);        ZVAL _STRINGL(pzv, str,
                                                 strlen(str), dup);
```

这里, zval的创建就开始变得有趣了. 字符串就像数组, 对象, 资源一样, 需要分配额外的内存用于它们的数据存储. 在下一章你将继续探索内存管理的陷阱; 现在, 只需要注意, 当dup的值为1时, 将分配新的内存并拷贝字符串内容, 当dup的值为0时, 只是简单的将zval指向已经存在的字符串数据.

```
ZVAL_RESOURCE(pzv, res);           Z_TYPE_P(pzv) = IS_RESOURCE;
                                   Z_RESVAL_P(pzv) = res;
```

回顾前面, 资源在zval中只是存储了一个简单的整型, 它用于在Zend管理的资源表中查找. 因此ZVAL_RESOURCE()宏就很像ZVAL_LONG()宏, 但是, 使用不同的类型.

数据类型/值/创建回顾练习

```
static void eae_001_zval_dump_real(zval *z, int level) {
    HashTable    *ht;
    int          ret;
    char     *key;
    uint     index;
    zval     **pData;

    switch ( Z_TYPE_P(z) ) {
        case IS_NULL:
            php_printf("%*stype = null, refcount = %d%s\n", level * 4, "", Z_REFCOUNT_P(z),
Z_ISREF_P(z) ? ", is_ref " : "");
            break;
        case IS_BOOL:
            php_printf("%*stype = bool, refcount = %d%s, value = %s\n", level * 4, "",
Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", Z_BVAL_P(z) ? "true" : "false");
            break;
        case IS_LONG:
            php_printf("%*stype = long, refcount = %d%s, value = %ld\n", level * 4, "",
Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", Z_LVAL_P(z));
            break;
        case IS_STRING:
            php_printf("%*stype = string, refcount = %d%s, value = \"%s\", len = %d\n", level *
4, "", Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", Z_STRVAL_P(z), Z_STRLEN_P(z));
            break;
        case IS_DOUBLE:
```

```
            php_printf("%*stype = double, refcount = %d%s, value = %0.6f\n", level * 4, "",
Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", Z_DVAL_P(z));
            break;
        case IS_RESOURCE:
            php_printf("%*stype = resource, refcount = %d%s, resource_id = %d\n", level * 4,
"", Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", Z_RESVAL_P(z));
            break;
        case IS_ARRAY:
            ht       = Z_ARRVAL_P(z);

            zend_hash_internal_pointer_reset(ht);
            php_printf("%*stype = array, refcount = %d%s, value = %s\n", level * 4, "",
Z_REFCOUNT_P(z), Z_ISREF_P(z) ? ", is_ref " : "", HASH_KEY_NON_EXISTANT !=
zend_hash_has_more_elements(ht) ? "" : "empty");
            while ( HASH_KEY_NON_EXISTANT != (ret = zend_hash_get_current_key(ht, &key, &index,
0)) ) {
                if ( HASH_KEY_IS_STRING == ret ) {
                    php_printf("%*skey is string \"%s\"", (level + 1) * 4, "", key);
                } else if ( HASH_KEY_IS_LONG == ret ) {
                    php_printf("%*skey is long %d", (level + 1) * 4, "", index);
                }
                ret  = zend_hash_get_current_data(ht, &pData);
                eae_001_zval_dump_real(*pData, level + 1);
                zend_hash_move_forward(ht);
            }
            zend_hash_internal_pointer_end(Z_ARRVAL_P(z));
            break;
        case IS_OBJECT:
            php_printf("%*stype = object, refcount = %d%s\n", level * 4, "", Z_REFCOUNT_P(z),
Z_ISREF_P(z) ? ", is_ref " : "");
            break;
        default:
            php_printf("%*sunknown type, refcount = %d%s\n", level * 4, "", Z_REFCOUNT_P(z),
Z_ISREF_P(z) ? ", is_ref " : "");
            break;
    }
}

PHP_FUNCTION(eae_001_zval_dump)
{
    zval *z;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &z) == FAILURE) {
        return;
    }

    eae_001_zval_dump_real(z, 0);

    RETURN_NULL();
}

PHP_FUNCTION(eae_001_zval_make)
{
    zval *z;

    MAKE_STD_ZVAL(z);

    ZVAL_NULL(z);
    eae_001_zval_dump_real(z, 0);

    ZVAL_TRUE(z);
    eae_001_zval_dump_real(z, 0);

    ZVAL_FALSE(z);
```

```
    eae_001_zval_dump_real(z, 0);

    ZVAL_LONG(z, 100);
    eae_001_zval_dump_real(z, 0);

    ZVAL_DOUBLE(z, 100.0);
    eae_001_zval_dump_real(z, 0);

    ZVAL_STRING(z, "100", 0);
    eae_001_zval_dump_real(z, 0);
}
```

数据存储

你已经在用户空间一侧使用过php了, 因此你应该已经比较熟悉数组了. 我们可以将任意数量的php变量(zval)放入到一个容器(array)中, 并可以为它们指派数字或字符串格式的名字(标签----key)

如果不出意外, php脚本中的每个变量都应该可以在一个数组中找到. 当你创建变量时, 为它赋一个值, Zend把这个值放到被称为符号表的一个内部数组中.

有一个符号表定义了全局作用域, 它在请求启动后, 扩展的RINIT方法被调用之前初始化, 接着在脚本执行完成后, 后续的RSHUTDOWN方法被执行之前销毁.

当一个用户空间的函数或对象方法被调用时, 则分配一个新的符号表用于函数或方法的生命周期, 它被定义为激活的符号表. 如果当前脚本的执行不在函数或方法中, 则全局符号表被认为是激活的.

我们来看看globals结构的实现(在Zend/zend_globals.h中定义), 你会看到下面的两个元素定义:

```
struct _zend_execution_globals {
    ...
    HashTable symbol_table;
    HashTable *active_symbol_table;
    ...
};
```

symbol_table, 使用EG(symbol_table)访问, 它永远都是全局变量作用域, 和用户空间的$GLOBALS变量相似, 用于对应于php脚本的全局作用域. 实际上, $GLOBALS变量的内部就是对EG(symbol_table)上的一层包装.

另外一个元素active_symbol_table, 它的访问方法类似: EG(active_symbol_table), 表示此刻激活的变量作用域.

这里有一个需要注意的关键点, EG(symbol_table), 它不像你在php和zend api下工作时将遇到的几乎所有其他HashTable, 它是一个直接变量. 几乎所有的函数在HashTable上操作时都期望一个间访的HashTable *作为参数. 因此, 你在使用时需要在EG(symbol_table)前加取地址符(&).

考虑下面的代码块, 它们的功能是等价的

```
/* php实现 */
<?php $foo = 'bar'; ?>

/* C实现 */
{
    zval *fooval;

    MAKE_STD_ZVAL(fooval);
    ZVAL_STRING(fooval, "bar", 1);
    ZEND_SET_SYMBOL(EG(active_symbol_table), "foo", fooval);
}
```

首先, 使用MAKE_STD_ZVAL()分配一个新的zval, 它的值被初始化为字符串"bar". 接着是一个新的宏调用, 它的作用是将fooval这个zval增加到当前激活的符号表中, 设置的变

量名为"foo". 因为此刻并没有用户空间函数被激活, 因此EG(active_symbol_table) == &EG(symbol_table), 最终的含义就是这个变量被存储到了全局作用域中.

数据取回

为了从用户空间取回一个变量, 你需要在符号表的存储中查找. 下面的代码段展示了使用zend_hash_find()函数达成这个目的:

```
{
    zval **fooval;

    if (zend_hash_find(EG(active_symbol_table),
                       "foo", sizeof("foo"),
                       (void**)&fooval) == SUCCESS) {
        php_printf("Got the value of $foo!");
    } else {
        php_printf("$foo is not defined.");
    }
}
```

这个例子中有一点看起来有点奇怪. 为什么要把fooval定义为两级间访指针呢? 为什么sizeof()用于确定"foo"的长度呢? 为什么是&fooval? 哪一个被评估为zval ***, 转换为void **?如果你问了你自己所有上面3个问题, 请拍拍自己的后背.

首先, 要知道HashTable并不仅用于用户空间变量, 这一点很有价值. HashTable结构用途很广, 它被用在整个引擎中, 甚至它还能完美的存储非指针数据. HashTable的桶是定长的, 因此, 为了存储任意大小的数据, HashTable将分配一块内存用来放置被存储的数据. 对于变量而言, 被存储的是一个zval *, 因此HashTable的存储机制分配了一块足够保存一个指针的内存. HashTable的桶使用这个新的指针保存zval *的值, 因此在HashTable中被保存的是zval **. HashTable完全可以漂亮的存储一个完整的zval, 那为什么还要这样存储zval *呢? 具体原因我们将在下一章讨论.

在尝试取回数据的时候, HashTable仅知道有一个指针指向某个数据. 为了将指针弹出到调用函数的本地存储中, 调用函数自然就要取本地指针(变量)的地址, 结果就是一个未知类型的两级间访的指针变量(比如void **). 要知道你的未知类型在这里是zval *, 你可以看到把这种类型传递给zend_hash_find()时, 编译器会发现不同, 它知道是三级间访而不是两级. 这就是我们在前面加一个强制类型转换的目的, 用来抑制编译器的警告.

在前面的例子中使用sizeof()的原因是为了在"foo"常量用作变量的标签时包含它的终止NULL字节. 这里使用4的效果是等价的; 不过这比较危险, 因为对标签名的修改会影响它的长度, 现在这样做在标签名变更时比较容易查找需要修改的地方. (strlen("foo") + 1)也可以解决这个问题, 但是, 有些编译器并没有优化这一步, 结果产生的二进制文件最终执行时可能得到的是一个毫无意义的字符串长度, 拿它去循环可不是那么好玩的!

如果zend_hash_find()定位到了你要查找的项, 它就会将所请求数据第一次被增加到HashTable中时时分配的桶的指针地址弹出到所提供的指针(zend_hash_find()第4个参数)中, 同时返回一个SUCCESS整型常量. 如果zend_hash_find()不能定位到数据, 它就不会修改指针(zend_hash_find()第四个参数)而是返回整型常量FAILURE.

站在用户空间的角度看, 变量存储到符号表所返回的SUCCESS或FAILURE实际上就是变量是否已经设置(isset).

类型转换

现在你可以从符号表抓取变量, 那可能你就想对它们做些什么. 一种直接的事倍功半的方法是检查变量的类型, 并依赖类型执行特殊的动作. 就像下面代码中简单的switch语句就可以工作.

```
void display_zval(zval *value)
```

```
{
    switch (Z_TYPE_P(value)) {
        case IS_NULL:
            /* NULLs are echoed as nothing */
            break;
        case IS_BOOL:
            if (Z_BVAL_P(value)) {
                php_printf("1");
            }
            break;
        case IS_LONG:
            php_printf("%ld", Z_LVAL_P(value));
            break;
        case IS_DOUBLE:
            php_printf("%f", Z_DVAL_P(value));
            break;
        case IS_STRING:
            PHPWRITE(Z_STRVAL_P(value), Z_STRLEN_P(value));
            break;
        case IS_RESOURCE:
            php_printf("Resource #%ld", Z_RESVAL_P(value));
            break;
        case IS_ARRAY:
            php_printf("Array");
            break;
        case IS_OBJECT:
            php_printf("Object");
            break;
        default:
            /* Should never happen in practice,
             * but it's dangerous to make assumptions
             */
             php_printf("Unknown");
             break;
    }
}
```

是的, 简单, 正确. 对比前面<?php echo $value; ?>的例子, 不难猜想这种编码会使得代码不好管理. 幸运的是, 在脚本执行输出变量的行为时, 无论是扩展, 还是嵌入式环境, 引擎都使用了非常相似的里程. 使用Zend暴露的convert_to_*()函数族可以让这个例子变得很简单:

```
void display_zval(zval *value)
{
    convert_to_string(value);
    PHPWRITE(Z_STRVAL_P(value), Z_STRLEN_P(value));
}
```

你可能会猜到, 有很多这样的函数用于转换到大多数数据类型. 值得注意的是convert_to_resource(), 它没有意义, 因为资源类型的定义旧是不能映射到真实用户空间表示的值.

如果你担心convert_to_string()调用对传递给函数的zval的值的修改不可逆, 那说明你很棒. 在真正的代码段中, 这是典型的坏主意, 当然, 引擎在输出变量时并不是这样做的. 下一章你将会看到安全的使用转换函数的方法, 它会安全的修改值的内容, 而不会破坏它已有的内容.

小结

本章中你看到了php变量的内部表示. 你学习了区别类型, 设置和取回值, 将变量增加到符号表中以及将它们取回. 下一章你将在这些知识的基础之上, 学习怎样拷贝一个zval, 怎样在不需要的时候销毁它们, 最重要的而是, 怎样避免在不需要的时候产生拷贝.

你还将看到Zend的单请求内存管理层的一角, 了解了持久化和非持久化分配. 在下一章的结尾, 你旧有实力可以去创建一个工作的扩展并在上面用自己的代码做实验了.

内存管理

php和c最重要的区别就是是否控制内存指针.

内存

在php中, 设置一个字符串变量很简单: <?php $str = 'hello world'; ?>, 字符串可以自由的修改, 拷贝, 移动. 在C中, 则是另外一种方式, 虽然你可以简单的用静态字符串初始化: char *str = "hello world"; 但是这个字符串不能被修改, 因为它存在于代码段. 要创建一个可维护的字符串, 你需要分配一块内存, 并使用一个strdup()这样的函数将内容拷贝到其中.

```
{
   char *str;

   str = strdup("hello world");
   if (!str) {
       fprintf(stderr, "Unable to allocate memory!");
   }
}
```

传统的内存管理函数(malloc(), free(), strdup(), realloc(), calloc()等)不会被php的源代码直接使用, 本章将解释这么做的原因.

释放分配的内存

内存管理在以前的所有平台上都以请求/释放的方式处理. 应用告诉它的上层(通常是操作系统)"我想要一些内存使用", 如果空间允许, 操作系统提供给程序, 并对提供出去的内存进行一个记录.

应用使用完内存后, 应该将内存还给OS以使其可以被分配给其他地方. 如果程序没有还回内存, OS就没有办法知道这段内存已经不再使用, 这样就无法分配给其他进程. 如果一块内存没有被释放, 并且拥有它的应用丢失了对它的句柄, 我们就称为"泄露", 因为已经没有人可以直接得到它了.

在典型的客户端应用中, 小的不频繁的泄露通常是可以容忍的, 因为进程会在一段时间后终止, 这样泄露的内存就会被OS回收. 并不是说OS很牛知道泄露的内存, 而是它知道为已经终止的进程分配的内存都不会再使用.

对于长时间运行的服务端守护进程, 包括apache这样的webserver, 进程被设计为运行很长周期, 通常是无限期的. 因此OS就无法干涉内存使用, 任何程度的泄露无论多小都可能累加到足够导致系统资源耗尽.

考虑用户空间的stristr()函数; 为了不区分大小写查找字符串, 它实际上为haystack和needle各创建了一份小写的拷贝, 接着执行普通的区分大小写的搜索去查找相关的偏移量. 在字符串的偏移量被定位后, haystack和needle字符串的小写版本都不会再使用了. 如果没有释放这些拷贝, 那么每个使用stristr()的脚本每次被调用的时候都会泄露一些内存. 最终, webserver进程会占用整个系统的内存, 但是却都没有使用.

完美的解决方案是编写良好的, 干净的, 一致的代码, 保证它们绝对正确. 不过在php解释器这样的环境中, 这只是解决方案的一半.

错误处理

为了提供从用户脚本的激活请求和所在的扩展函数中跳出的能力, 需要存在一种方法跳出整个激活请求. Zend引擎中的处理方式是在请求开始的地方设置一个跳出地址, 在所有的die()/exit()调用后, 或者碰到一些关键性错误(E_ERROR)时, 执行longjmp()转向到预先设置的跳出地址.

虽然这种跳出处理简化了程序流程, 但它存在一个问题: 资源清理代码(比如free()调用)会被跳过, 会因此带来泄露. 考虑下面简化的引擎处理函数调用的代码:

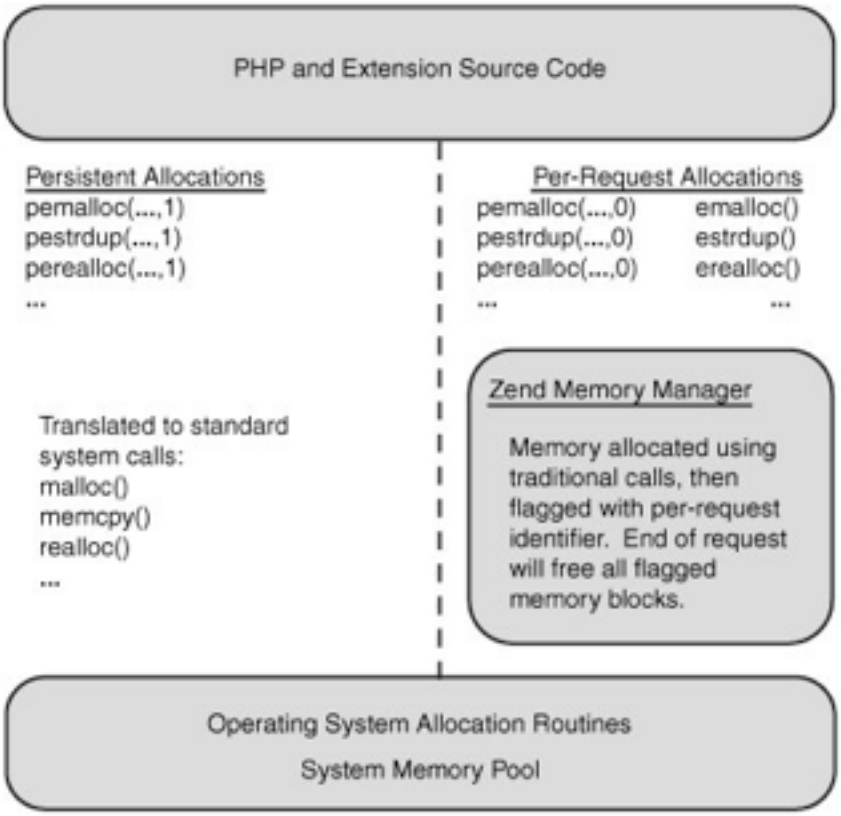
```
void call_function(const char *fname, int fname_len TSRMLS_DC)
{
    zend_function *fe;
    char *lcase_fname;
    /* php函数是大小写不敏感的, 为了简化在函数表中对它们的定位, 所有的函数名都隐式的翻译为小写 */
    lcase_fname = estrndup(fname, fname_len);
    zend_str_tolower(lcase_fname, fname_len);

    if (zend_hash_find(EG(function_table),
            lcase_fname, fname_len + 1, (void **)&fe) == FAILURE) {
        zend_execute(fe->op_array TSRMLS_CC);
    } else {
        php_error_docref(NULL TSRMLS_CC, E_ERROR,
                         "Call to undefined function: %s()", fname);
    }
    efree(lcase_fname);
}
```

当php_error_docref()一行执行到时, 内部的处理器看到错误级别是关键性的, 就调用longjmp()中断当前程序流, 离开call_function(), 这样就不能到达efree(lcase_fname)一行. 那你就可能会想, 把efree()行移动到php_error_docref()上面, 但是如果这个call_function()调用进入第一个条件分支呢(查找到了函数名, 正常执行)? 还有一点, fname自己是一个分配的字符串, 并且它在错误消息中被使用, 在使用完之前你不能释放它.

*php_error_docref()*函数是一个内部等价于*trigger_error()*. 第一个参数是一个可选的文档引用, 如果在*php.ini*中启用它将被追加到*docref.root*后面. 第三个参数可以是任意的*E_*族常量标记错误的严重程度. 第四个和后面的参数是符合*printf()*样式的格式串和可变参列表.

Zend内存管理

由于请求跳出(故障)产生的内存泄露的解决方案是Zend内存管理(ZendMM)层. 引擎的这一部分扮演了相当于操作系统通常扮演的角色, 分配内存给调用应用. 不同的是, 站在进程空间请求的认知角度, 它足够底层, 当请求die的时候, 它可以执行和OS在进程die时所做的相同的事情. 也就是说它会隐式的释放所有请求拥有的内存空间. 下图展示了在php进程中ZendMM和OS的关系:



除了提供隐式的内存清理, ZendMM还通过php.ini的设置memory_limit控制了每个请求的内存使用. 如果脚本尝试请求超过系统允许的, 或超过单进程内存限制剩余量的内存, ZendMM会自动的引发一个E_ERROR消息, 并开始跳出进程. 一个额外的好处是多数时候内存分配的结果不需要检查, 因为如果失败会立即longjmp()跳出到引擎的终止部分.

在php内部代码和OS真实的内存管理层之间hook的完成, 最复杂的是要求所有内部的内存分配要从一组函数中选择. 例如, 分配一个16字节的内存块不是使用malloc(16), php代码应该使用emalloc(16). 除了执行真正的内存分配任务, ZendMM还要标记内存块所绑定请求的相关信息, 以便在请求被故障跳出时, ZendMM可以隐式的释放它(分配的内存).

很多时候内存需要分配, 并使用超过单请求生命周期的时间. 这种分配我们称为持久化分配, 因为它们在请求结束后持久的存在, 可以使用传统的内存分配器执行分配, 因为它们不可以被ZendMM打上每个请求的信息. 有时, 只有在运行时才能知道特定的分配需要持久化还是不需要, 因此ZendMM暴露了一些帮助宏, 由它们来替代其他的内存分配函数, 但是在末尾增加了附加的参数来标记是否持久化.

如果你真的想要持久化的分配, 这个参数应该被设置为1, 这种情况下内存分配的请求将会传递给传统的malloc()族分配器. 如果运行时逻辑确定这个块不需要持久化 则这个参数被设置为0, 调用将会被转向到单请求内存分配器函数.

例如, pemalloc(buffer_len, 1)映射到malloc(buffer_len), 而pemalloc(buffer_len, 0)映射到emalloc(buffer_len), 如下:

```
#define in Zend/zend_alloc.h:

#define pemalloc(size, persistent) \
           ((persistent)?malloc(size): emalloc(size))
```

ZendMM提供的分配器函数列表如下, 并列出了它们对应的传统分配器.

传统分配器	php中的分配器
void *malloc(size_t count);	void *emalloc(size_t count);
	void *pemalloc(size_t count, char persistent);
void *calloc(size_t count);	void *ecalloc(size_t count);
	void *pecalloc(size_t count, char persistent);
void *realloc(void *ptr, size_t count);	void *erealloc(void *ptr, size_t count);
	void *perealloc(void *ptr, size_t count, char persistent);
void *strdup(void *ptr);	void *estrdup(void *ptr);
	void *pestrdup(void *ptr, char persistent);
void free(void *ptr);	void efree(void *ptr);
	void pefree(void *ptr, char persistent);

你可能注意到了, pefree要求传递持久化标记. 这是因为在pefree()调用时, 它并不知道ptr是否是持久分配的. 在废持久分配的指针上调用free()可能导致双重的free, 而在持久化的分配上调用efree()通常会导致段错误, 因为内存管理器会尝试查看管理信息, 而它不存在. 你的代码需要记住它分配的数据结构是不是持久化的.

除了核心的分配器外, ZendMM还增加了特殊的函数:

```
void *estrndup(void *ptr, int len);
```

它分配len + 1字节的内存, 并从ptr拷贝len个字节到新分配的块中. estrndup()的行为大致如下:

```
void *estrndup(void *ptr, int len)
{
    char *dst = emalloc(len + 1);
    memcpy(dst, ptr, len);
    dst[len] = 0;
    return dst;
}
```

终止NULL字节被悄悄的放到了缓冲区末尾, 这样做确保了所有使用estrndup()进行字符串赋值的函数不用担心将结果缓冲区传递给期望NULL终止字符串的函数(比如printf())时产生错误. 在使用estrndup()拷贝非字符串数据时, 这个最后一个字节将被浪费, 但是相比带来的方便, 这点小浪费就不算什么了.

```
void *safe_emalloc(size_t size, size_t count, size_t addtl);
void *safe_pemalloc(size_t size, size_t count, size_t addtl, char persistent);
```

这两个函数分配的内存大小是((size * count) + addtl)的结果. 你可能会问, "为什么要扩充这样一个函数? 为什么不是使用emalloc/pemalloc, 然后自己计算呢?" 理由来源于它的名字"安全". 尽管这种情况很少有可能发生, 但仍然是有可能的, 当计算的结果溢出所在主机平台的整型限制时, 结果会很糟糕. 可能导致分配负的字节数, 更糟的是分配一个正值的内存大小, 但却小于所请求的大小. safe_emalloc()通过检查整型溢出避免了这种类型的陷阱, 如果发生溢出, 它会显式的报告失败.

并不是所有的内存分配例程都有*p*副本. 例如, *pestrndup()*和*safe_pemalloc()*在*php 5.1*之前就不存在. 有时你需要在*ZendAPI*的这些不足上工作.

引用计数

在php这样长时间运行的多请求进程中谨慎的分配和释放内存非常重要, 但这只是一半工作. 为了让高并发的服务器更加高效, 每个请求需要使用尽可能少的内存, 最小化不需要的数据拷贝. 考虑下面的php代码片段:

```
<?php
    $a = 'Hello World';
    $b = $a;
    unset($a);
?>
```

在第一次调用后, 一个变量被创建, 它被赋予12字节的内存块, 保存了字符串"Hello world"以及结尾的NULL. 现在来看第二句: $b被设置为和$a相同的值, 接着$a被unset(释放)

如果php认为每个变量赋值都需要拷贝变量的内容, 那么在数据拷贝期间就需要额外的12字节拷贝重复的字符串, 以及额外的处理器负载. 在第三行出现的时候, 这种行为看起来就有些可笑了, 原来的变量被卸载使得数据的复制完全不需要. 现在我们更进一步想想当两个变量中被装载的是一个10MB文件的内容时, 会发生什么? 它需要20MB的内存, 然而只要10MB就足够了. 引擎真的会做这种无用功浪费这么多的时间和内存吗?

你知道php是很聪明的.

还记得吗? 在引擎中变量名和它的值是两个不同的概念. 它的值是自身是一个没有名字的zval *. 使用zend_hash_add()将它赋值给变量$a. 那么两个变量名指向相同的值可以吗?

```
{
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),
                                           &helloval, sizeof(zval*), NULL);
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),
                                           &helloval, sizeof(zval*), NULL);
}
```

此时, 在你检查$a或$b的时候, 你可以看到, 它们实际都包含了字符串"Hello World". 不幸的是, 接着来了第三行: unset($a);. 这种情况下, unset()并不知道$a指向的数据还被另外一个名字引用, 它只是释放掉内存. 任何后续对$b的访问都将查看已经被释放的内存空间, 这将导致引擎崩溃. 当然, 你并不希望引擎崩溃.

这通过zval的第三个成员: refcount解决. 当一个变量第一次被创建时, 它的refcount被初始化为1, 因为我们认为只有创建时的那个变量指向它. 当你的代码执行到将helloval赋值给$b时, 它需要将refcount增加到2, 因为这个值现在被两个变量"引用"

```
{
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),
                                           &helloval, sizeof(zval*), NULL);
    ZVAL_ADDREF(helloval);
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),
                                           &helloval, sizeof(zval*), NULL);
}
```

现在, 当unset()删除变量的$a拷贝时, 它通过refcount看到还有别人对这个数据感兴趣, 因此它只是将refcount减1, 其他什么事情都不做.

写时复制

通过引用计数节省内存是一个很好的主意, 但是当你只想修改其中一个变量时该怎么办呢? 考虑下面的代码片段:

```
<?php
    $a = 1;
    $b = $a;
    $b += 5;
?>
```

看上面代码的逻辑, 处理完后期望$a仍然等于1, 而$b等于6. 现在你知道, Zend为了最大化节省内存, 在第二行代码执行后$a和$b只想同一个zval, 那么到达第三行代码时会发生什么呢? $b也会被修改吗?

答案是Zend查看refcount, 看到它大于1, 就对它进行了隔离. Zend引擎中的隔离是破坏一个引用对, 它和你刚才看到的处理是对立的:

```
zval *get_var_and_separate(char *varname, int varname_len TSRMLS_DC)
{
    zval **varval, *varcopy;
    if (zend_hash_find(EG(active_symbol_table),
                       varname, varname_len + 1, (void**)&varval) == FAILURE) {
       /* 变量不存在 */
       return NULL;
   }
   if ((*varval)->refcount < 2) {
       /* 变量名只有一个引用, 不需要隔离 */
       return *varval;
   }
   /* 其他情况, 对zval *做一次浅拷贝 */
   MAKE_STD_ZVAL(varcopy);
   varcopy = *varval;
   /* 对zval *进行一次深拷贝 */
   zval_copy_ctor(varcopy);

   /* 破坏varname和varval之间的关系, 这一步会将varval的引用计数减小1 */
   zend_hash_del(EG(active_symbol_table), varname, varname_len + 1);

   /* 初始化新创建的值的引用计数, 并为新创建的值和varname建立关联 */
   varcopy->refcount = 1;
   varcopy->is_ref = 0;
   zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,
                                        &varcopy, sizeof(zval*), NULL);
   /* 返回新的zval * */
   return varcopy;
}
```

现在引擎就有了一个只被$b变量引用的zval *, 就可以将它转换为long, 并将它的值按照脚本请求增加5.

写时修改

引用计数的概念还创建了一种新的数据维护方式, 用户空间脚本将这种方式称为"引用". 考虑下面的用户空间代码片段:

```
<?php
    $a = 1;
    $b = &$a;
    $b += 5;
?>
```

凭借你在php方面的经验, 直觉上你可能认识到$a的值现在应该是6, 即便它被初始化为1并没有被(直接)修改过. 发生这种情况是因为在引擎将$b的值增加5的时候, 它注意到

$b是$a的一个引用, 它就说"对于我来说不隔离它的值就修改是没有问题的, 因为我原本就想要所有的引用变量都看到变更"

但是引擎怎么知道呢? 很简单, 它查看zval结构的最后一个元素: is_ref. 它只是一个简单的开关, 定义了zval是值还是用户空间中的引用. 在前面的代码片段中, 第一行执行后, 为$a创建的zval, refcount是1, is_ref是0, 因为它仅仅属于一个变量($a), 并没有其他变量的引用指向它. 第二行执行时, 这个zval的refcount增加到2, 但是此时, 因为脚本中增加了一个取地址符(&)标记它是引用传值, 因此将is_ref设置为1.

最后, 在第三行中, 引擎获得$b关联的zval, 检查是否需要隔离. 此时这个zval不会被隔离, 因为在前面我们没有包含的一段代码(如下). 在get_var_and_separate()中检查refcount的地方, 还有另外一个条件:

```
if ((*varval)->is_ref || (*varval)->refcount < 2) {
    /* varname只有在真的是引用方式, 或者只被一个变量引用时才会不发生隔离 */
    return *varval;
}
```

此时, 即便refcount为2, 隔离处理也会被短路, 因为这个值是引用传值的. 引擎可以自由的修改它而不用担心引用它的其他变量被意外修改.

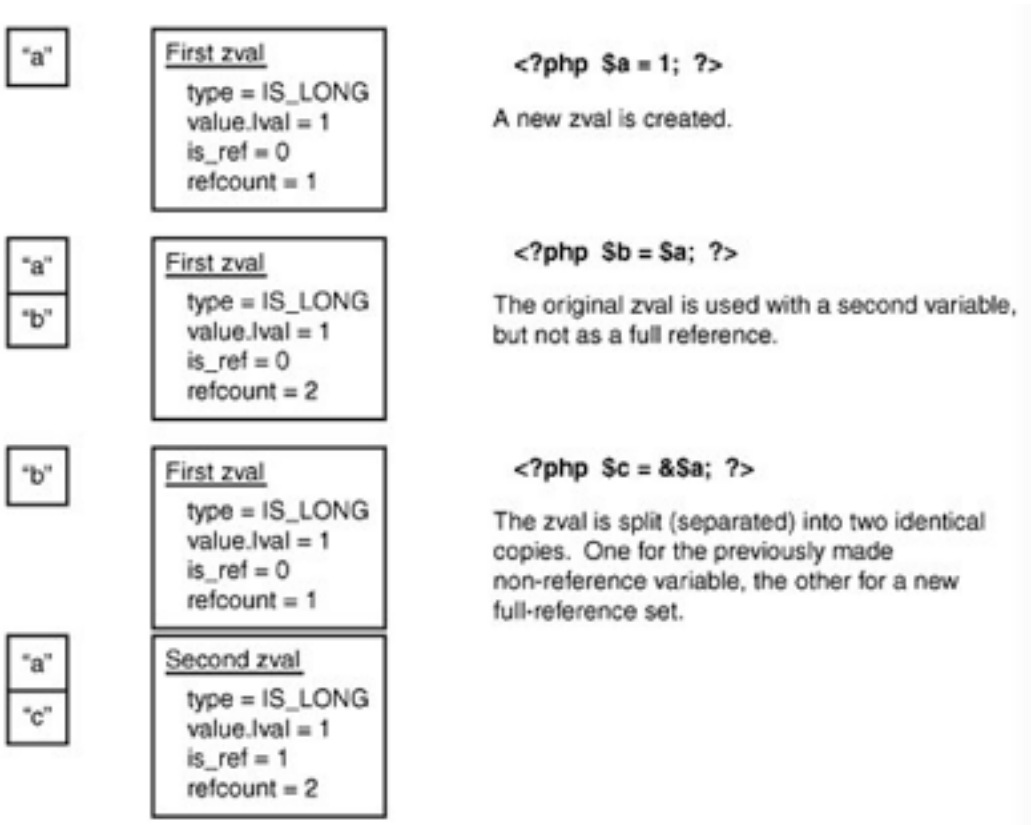
隔离的问题

对于这些拷贝和引用, 有一些组合是is_ref和refcount无法很好的处理的. 考虑下面的代码:

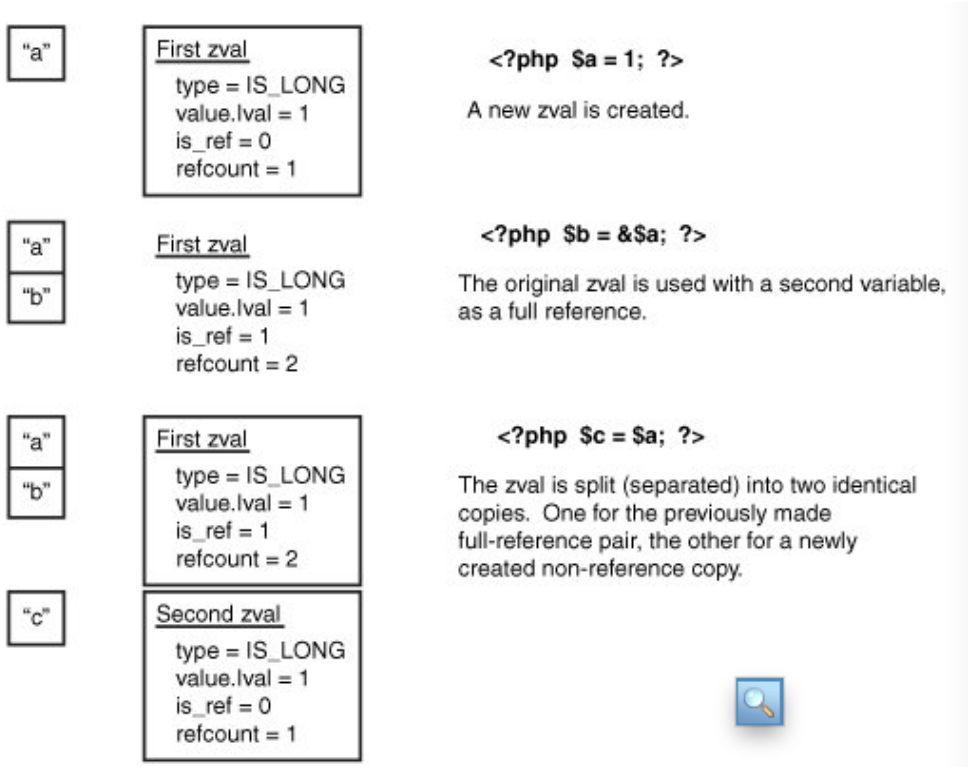
```
<?php
    $a = 1;
    $b = $a;
    $c = &$a;
?>
```

这里你有一个值需要被3个不同的变量关联, 两个是写时修改的引用方式, 另外一个是隔离的写时复制上下文. 仅仅使用is_ref和refcount怎样来描述这种关系呢?

答案是: 没有. 这种情况下, 值必须被复制到两个分离的zval *, 虽然两者包含相同的数据. 如下图:



类似的, 下面的代码块将导致相同的冲突, 并强制值隔离到一个拷贝中(如下图)



```
<?php
    $a = 1;
    $b = &$a;
    $c = $a;
?>
```

注意, 这里两种情况下, $b都和原来的zval对象关联, 因为在隔离发生的时候, 引擎不知道操作中涉及的第三个变量的名字.

小结

php是一种托管语言. 从用户空间一侧考虑, 小心的控制资源和内存就意味着更容易的原型涉及和更少的崩溃. 在你深入研究揭开引擎的面纱后, 就不能再有博彩心里, 而是对运行环境完整性的开发和维护负责.

安装构建环境

现在你可能至少已经有一个已安装的php, 并且已经使用它做基于web的应用开发了. 你可能已经从php.net下载了win32构建并运行在iis或windows版的apache上, 或者使用你的*nix(linux, bsd, 或其他遵循POSIX的发行)发行版的包管理系统安装了第三方创建的二进制.

构建php

除非你是下载源码包自己编译, 否则你肯定会错过一些知识点.

*nix工具

C开发者工具集中必不可少的第一个工具是C的编译器. 你的发行版中可能会默认包含一个, 如果幸运, 它就是gcc(GNU编译器集合). 你可以通过执行gcc version或cc version检查是否安装了编译器, 如果已经安装, 它会响应已安装的编译器版本信息.

如果你还没有安装编译器, 可以按照你使用的发行版官方指定的方式下载并安装gcc. 通常这就意味着要下载一个.rpm或.deb文件, 并执行一个命令去安装它. 这依赖于你使用的发行版版本, 你可以简单的使用下面命令去尝试安装: urpmi gcc, apt-get install gcc, pkg-add -r gcc, emerge gcc.

除了编译器, 你还需要下面的程序和工具: make, autoconf, automake, libtool. 这些工具同样可以用你所使用的发行版的包管理系统去安装, 和安装gcc时一样, 或者直接从gnu.org下载源码包自己编译.

推荐的版本是: libtool 1.4.3, autoconf 2.13, automake 1.4或1.5. 使用这些软件的更新的版本可能也能很好的工作, 但这些版本是经过长期使用验证的.

如果你计划使用CVS检出最新的php开发版代码, 还需要bison和flex去构造语言解释器. 和其他包一样, 这两个包可以使用你的发行版包管理系统安装, 或者从gnu.org下载源码自己编译.

如果你选择了CVS, 你还需要安装cvs客户端. 同样, 它也可能已经在你的发行版上安装, 或者你自己去下载编译. 和其他包不一样的是这个包你需要在cvshome.org下载.

Win32工具

译者不熟悉windows环境, 因此略过.

获取php源代码

下载php的时候, 你有集中选择. 首先, 如果你的发行版支持, 你可以使用apt-get source php5这样的命令去下载. 这种方式的有点在于你使用的发行版可能存在某些问题需要对php源代码进行修改, 从这里下载, 可以肯定这些问题已经被打补丁使得你的构建存在更少的问题.缺点在于大多数的发行版都会比php官方的发布延迟数周.

另外一个选择是首选项, 在www.php.net下载php-x.y.z.tar.gz(x.y.z是当前发布版本). 这些php发布版是经过全世界无数的php用户测试的, 并且是最新的.

你还可以从snaps.php.net下载快照包. 这个站点上, php版本库中所有源代码的最新版本会每隔几小时打包一次.php内核开发者的某些提交可能会导致它暂时不可用, 但是如果你在官方发布之前需要最新的php 6.0的特性, 这是最容易得到的地方.

最后, 你可以使用cvs直接获取到php内核开发团队所使用的开发版. 如果你只是要开发扩展和嵌入式程序, 相比使用官方发布包和获取快照,这不会有什么明显的好处. 但是如果你计划发布你的扩展或其他应用到CVS库, 熟悉检出过程还是很有用的.

译注: *php*目前已经使用*git*来管理代码库, 关于*cvs*检出不再赘述, 请访问*https://github.com/php/php-src*获取最新源码. 如果你想为*php*贡献代码, 可以查看该项目首页的介绍.

配置用于开发的php

第一章中我们讨论了, 无论你计划开发扩展还是潜入php的其他应用, 在构建开发者友好的php时有两个特殊的./configure开关你需要使用, 这两个开关应该和你构建php时使用的其他开关一起使用.

enable-debug

在php和zend源码树的某些关键函数上开启调试. 首先它启用了每个请求结束后的内存泄露报告.

回顾第三章"内存管理", ZendMM会隐式的释放每个请求分配的, 但在脚本结束之前没有被释放的内存. 通过在新开发的代码上运行一系列的回归测试用例, 泄露点可以很容易的暴露出来, 这样就可以在发布之间修补. 我们来看看下面的代码片段:

```
void show_value(int n)
{
    char *message = emalloc(1024);

    sprintf(message, "The value of n is %d\n", n);
    php_printf("%s", message);
}
```

如果这段愚蠢的代码在php请求执行过程中被执行, 它将泄露1024字节的内存. 一般情况下ZendMM会在脚本执行结束后释放它.

在enable-debug开启时, 就会为开发者提供定位问题的错误消息:

```
/cvs/php5/ext/sample/sample.c(33) :  Freeing 0x084504B8 (1024 bytes), script=-
=== Total 1 memory leaks detected ===
```

这个短小但完整的消息告诉你ZendMM在你弄脏了内存后它进行了清理, 并给出了泄露的内存块是在哪里分配的. 使用这个信息, 很容易定位问题, 打开文件, 找到对应的行, 在函数结束前适当的位置增加efree(message).

当然, 内存泄露并不是你会碰到的唯一难以追查的问题. 有时候, 问题是潜在的, 很少显现. 比如你通宵达旦的工作, 修改了很多的代码和源文件, 当所有事情做完后, 你自信的执行了make, 测试了一个简单的脚本, 接着就看到了下面的输出:

```
$ sapi/cli/php -r 'myext_samplefunc();'
Segmentation Fault
```

这只是表象, 那问题出在哪里呢? 查看你的myext_samplefunc()实现, 并没有显示出什么明显的线索, 使用gdb运行仅显示出一串未知的符号.

同样, enable-debug会帮到你. 通过在./configure时增加这个开关, 结果的php二进制将包含所有gdb以及其他core文件检查程序所需的调试符号, 这样可以显示出问题出在哪里.

使用这个选项重新构建, 通过gdb触发崩溃, 你现在可以看到下面的输出:

```
#0 0x1234567 php_myext_find_delimiter(str=0x1234567 "foo@#(FHVN)@\x98\xE0...",
                                      strlen=3, tsrm_ls=0x1234567)
    p = strchr(str, ',');
```

目标就变得清晰了. str字符串并不是NULL终止的, 后面的垃圾可以证明这一点, 而非二进制安全的函数使用了它. strchr()实现尝试从头到尾的扫描传入的str, 但由于没有终止NULL字节, 它到达了不属于它的内存, 这就导致了段错误. 我们可以使用memchr()和strlen参数来修复这个问题防止崩溃.

enable-mantainer-zts

这个选项强制php构建启用线程安全资源管理(TSRM)/Zend线程安全(ZTS)层. 这个开关会增加处理时的复杂度, 但是对于开发者而言, 你会发现这是一件好事情. 关于ZTS的详细介绍以及为什么在开发时要开启这个选项, 请参考第一章.

enable-embed

如果你要开发一个嵌入php的其他应用, 就需要另外一个非常重要的开关. 这个开关打开后就会构建出一个类似开启了with-apxs后构建出的mod_php5.so动态链接库: libphp5.so, 它可以用于将php嵌入到其他应用中.

在Unix上编译

现在你已经有了所有需要的工具, 下载了php源码包, 认识了所有需要的./configure开关, 是时候真正的编译php了.

这里假设你下载的是php-5.1.0.tar.gz, 放在了你的主目录, 你将使用下面的命令序列解包源码包, 并切换到解压出的源码目录:

```
[/home/sarag]$ tar -zxf php-5.1.0.tar.gz
[/home/sarag]$ cd php-5.1.0
```

如果你使用的不是gnu的tar, 命令可能需要略作修改:

```
[/home/sarag]$ gzip -d php-5.1.0.tar.gz | tar -xf -
```

现在, 用所需的开关和其他你想要开启或禁用的选项, 执行./configure命令:

```
[/home/sarag/php-5.1.0]$ ./configure enable-debug \
enable-maintainer-zts disable-cgi enable-cli \
disable-pear disable-xml disable-sqlite \
without-mysql enable-embed
```

在一段时间的处理后, 在你的屏幕上输出了很多的信息, 最终完成了./configure阶段. 接下来你就可以开始编译了:

```
[/home/sarag]$ make all install
```

现在, 站起来喝杯咖啡吧. 编译的时间在性能高的机器上可能需要几分钟, 在旧的486上甚至可能需要半个小时. 构建处理完成后, 你就拥有了一个正确配置, 功能完整, 可用于开发的php.

在Win32上编译

译者不熟悉windows环境, 因此略过.

小结

现在php已经以正确的选项安装了, 你已经准备好开发一个真实的, 有功能的扩展了. 后面的章节, 就开始剖析php扩展. 即便你只计划将php嵌入到你的应用中, 而不对语言做任何扩展, 你也应该阅读这些章节, 因为它详细解释了php的运行机制.

你的第一个扩展

每一个php扩展的构建至少需要两个文件: 一个configuration文件, 它告诉编译期要构建哪些文件以及需要什么外部的库, 还需要至少一个源文件, 它执行实际的工作.

剖析扩展

实际上, 通常会有第二个或第三个配置文件, 以及一个或多个头文件. 对于你的第一个扩展, 你需要添加每种类型的一个文件并使用它们工作.

配置文件

要开始了, 首先在你的php源代码目录树的ext/目录下创建名为sample的目录. 实际上这个新的目录可以放在任何地方, 但是为了在本章后面演示win32和静态构建选项, 我们还是先建立在源代码目录下吧.

下一步, 进入这个目录, 创建一个名为config.m4的文件, 键入下面内容:

```
PHP_ARG_ENABLE(sample,
  [Whether to enable the "sample" extension],
  [  enable-sample        Enable "sample" extension support])

if test $PHP_SAMPLE != "no"; then
  PHP_SUBST(SAMPLE_SHARED_LIBADD)
  PHP_NEW_EXTENSION(sample, sample.c, $ext_shared)
fi
```

这是./configure时能够调用enable-sample选项的最低要求.PHP_ARG_ENABLE的第二个参数将在./configure处理过程中到达这个扩展的配置文件时显示. 第三个参数将在终端用户执行./configure --help时显示为帮助信息.

有没有想过为什么有的扩展配置使用*enable-extname*, 而有的扩展则使用*with-extname?* 功能上两者没有区别. 实际上, *enable*表示启用这个特性不需要其他任何第三方库, 相比之下, *with*则表示要使用这个特性还有其他先决条件

现在, 你的*sample*扩展并不需要和其他库链接, 因此只需要使用*enable*版本. 在第*17*章"外部库"中, 我们将介绍使用*with*并指示编译器使用额外的*CFLAGS*和*LDFLAGS*设置.

如果终端用户使用enable-sample选项调用了./configure, 那么本地的环境变量$PHP_SAMPLE, 将被设置为yes. PHP_SUBST()是标准autoconf的AC_SUBST()宏的php修改版, 它在将扩展构建为共享模块时需要.

最后但并不是不重要的, PHP_NEW_EXTENSION()定义了模块并枚举了所有必须作为扩展的一部分编译的源文件. 如果需要多个文件, 它可以在第二个参数中使用空格分隔列举, 例如:

```
PHP_NEW_EXTENSION(sample, sample.c sample2.c sample3.c, $ext_shared)
```

最后一个参数是对应于PHP_SUBST(SAMPLE_SHARED_LIBADD)命令的, 在构建共享模块的时候同样需要它.

头文件

当使用C开发的时候, 将数据的类型定义放到外部的头文件中隔离起来, 由源文件包含是常见的做法. 尽管php并不要求这样, 但是这样做在模块增长到不能放到单个源文件时是有简化作用的.

在你的php_sample.h头文件中, 以下面内容开始:

```
#ifndef PHP_SAMPLE_H
/* 防止重复包含 */
#define PHP_SAMPLE_H

/* 定义扩展的属性 */
```

```
#define PHP_SAMPLE_EXTNAME    "sample"
#define PHP_SAMPLE_EXTVER    "1.0"

/* 在php源码树外面构建时引入配置选项 */
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

/* 包含php的标准头文件 */
#include "php.h"

/* 定义入口点符号, Zend在加载这个模块的时候使用 */
extern zend_module_entry sample_module_entry;
#define phpext_sample_ptr &sample_module_entry

#endif /* PHP_SAMPLE_H */
```

这个头文件完成了两个主要的任务: 如果扩展使用phpize工具构建(本书通常都使用这种方式), 那么HAVE_CONFG_H就是已定义的, 这样config.h就会被正常的包含进来. 无论扩展怎样编译, 都会从php源码树中包含php.h. 这个头文件中包含了php源码中访问大部分PHPAPI要使用的其他头文件.

接下来, 你的扩展使用的zend_module_entry结构定义为外部的, 这样当这个模块使用extension=xxx加载时, 就可以被Zend使用dlopen和dlsym()取到.

译注: 关于模块的加载过程, 请参考译者的一篇博客<从dl('xxx.so');函数分析PHP模块开发>(http://blog.csdn.net/lgg201/article/details/6584095)

头文件中还会包含一些预处理, 定义将在原文件中使用的信息.

源代码

最后, 最重要的你需要在sample.c中创建一个简单的源码骨架:

```
#include "php_sample.h"

zend_module_entry sample_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
     STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE_EXTNAME,
    NULL, /* Functions */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_SAMPLE
ZEND_GET_MODULE(sample)
#endif
```

就这样简单. 这三个文件是创建一个模块骨架所需的一切.但是, 它没有任何功能, 不过作为你在本节后面填充功能的模板是不错的选择. 不过先让我们看看究竟发生了什么.

开始的一行非常简单, 包含了你刚才创建的头文件, 通过扩展得到了php源码树中的其他内核头文件.

接下来, 创建你在头文件中定义的zend_module_entry结构. 你应该注意到了, zend_module_entry的第一个元素是一个条件表达式, 给予当前的ZEND_MODULE_API_NO定义. 这个API编号大概是php4.2.0的, 如果你确定你的扩展不

会安装在比这还古老的版本, 你可以砍掉#ifdef部分, 直接包含STANDARD_MODULE_HEADER元素.

考虑一下, 无论如何, 它都会在编译期耗费一点时间, 而不会对结果产生的二进制或处理需要的时间产生影响, 因此多数情况下最好直接砍掉这个条件. 这同样适用于下面的版本属性.

其他的6个元素现在初始设置为NULL; 你可以在它后面的注释中看到它的用途.

最后, 在最底下你可以看到每个可以编译为共享模块的php扩展都会有的一个公共元素. 这个简短的条件在你动态加载时由Zend增加一个引用. 不要关心它的细节, 你只需要保证它的存在, 否则下一节可能就无法工作了.

构建你的第一个扩展

现在你拥有了所有的文件, 是时候编译安装了. 相比编译主php二进制, 步骤上略有不同.

在*nix上构建

第一步是使用config.m4中的信息作为末班生成./configure脚本. 这可以运行在你安装主php二进制时附带安装的phpize程序来完成:

```
$ phpize
PHP Api Version: 20041225
Zend Module Api No: 20050617
Zend Extension Api No: 220050617
```

*Zend Extension Api No*前面多出来的*2*并不是印刷错误; 它对应于*Zend*引擎*2*这个版本号, 一般认为要保持这个*API*编号大于它对应的*ZE1*版本.

如果你此时查看当前目录 你会注意到比刚才的3个文件多了不少文件. phpize程序结合你扩展的config.m4文件以及从你的php构建中收集的信息和所有让编译发生所需的一切. 这意味着你不用纠缠在makefile和定位php头上面. php已经帮你做了这个工作.

下一步就简单了, 执行./configure. 这里你只要配置你的扩展, 因此你需要做的如下:

```
$ ./configure enable-sample
```

注意这里没有使用enable-debug和enable-maintainer-zts. 这是因为phpize已经将它们的值从主php构建中拿过来并应用到你的扩展的./configure脚本中了.

现在, 构建它! 和其他任何的包一样, 你只需要键入make, 生成的脚本文件就会处理剩下的事情.

构建处理完成后, 你会得到一个消息指出sample.so已经编译并放在了当前构建目录下一个名为"modules"的目录中.

在windows上构建

译者不熟悉windows平台, 因此略过.

将构建的扩展作为共享模块加载

在请求的时候为了让php找到这个模块, 需要将它放到php.ini中extension_dir设置的目录下. 默认的php.ini放在/usr/local/lib/php.ini; 不过这个默认值可能会因为包管理系统而不同. 检查php -i的输出可以看到你这个配置文件在哪里.

如果php.ini中的这个设置没有修改过, 它的值默认是"PHP_HOME/lib/php/extensions/debug-zts-20100525", 后面的debug-zts-20100525分别是是否启用调试, 是否启用zts, PHPAPI编号. 如果你还没有已加载的扩展, 或者说除了sample.so没有其他扩展, 可以将这个值修改到你make产生模块的路径. 否则, 直接将产生的sample.so拷贝到这个设置的目录下.(译注: php -i | grep extension_dir查找你的extension_dir设置)

在extension_dir指向正确的位置后, 有两种方式告诉php去加载你的模块. 第一种是在脚本中使用dl()函数:

```
<?php
    dl('sample.so');
    var_dump(get_loaded_modules());
?>
```

如果脚本没有显示sample已加载, 则表示有哪里出了问题. 查看输出上面的错误消息作为线索, 或者如果在php.ini中进行了相应的设置就参考error_log.

第二种方式, 也是更常用的方式, 在php.ini中使用extension指令指定要加载的模块. 这个指令在php.ini的设置中是比较特殊的, 可以多次以不同的值使用它. 因此如果你已经在php.ini中设置了一个扩展, 不要在同一行使用分隔符的方式列举, 而是插入新的一行: extension=sample.so. 此时你的php.ini看起来是这样的:

```
extension_dir=/usr/local/lib/php/modules/
extension=sample.so
```

现在你可以不使用dl()运行相同的脚本, 或者直接执行php -m命令, 就可以在已加载模块列表中看到sample了.

所有本章剩余以及以后章节的代码, 都假设你已经按照这里描述的方法加载了当前扩展. 如果你计划使用dl(), 请确认在测试脚本中加入加载的代码(dl()).

静态构建

在已加载模块列表中, 你可能注意到了一些在php.ini中并没有使用extension指令包含的模块. 这些模块是直接构建到php中的, 它们作为主php程序的一部分被编译进php中.

在*nix下静态构建

现在, 如果你现在进入到php源码树的根目录, 运行./configure --help, 你会看到虽然你的扩展和所有其他模块都在ext/目录下, 但它并没有作为一个选项列出. 这是因为, 此刻, ./configure脚本已经生成了, 而你的扩展并不知道. 要重新生成./configure并让它找到你的新扩展, 你需要做的只是执行一条命令:

```
$ ./buildconf
```

如果你使用产品发布版的php做开发, 你会发现./buildconf自己不能工作. 这种情况下, 你需要执行: ./buildconf --force来绕过对./configure命令的一些保护.

现在你执行./configure --help就可以看到--enable-sample是一个可用选项了. 此时, 你就可以重新执行./configure, 使用你原来构建主php时使用的所有选项, 外加--enable-sample, 这样构建出来的php二进制文件就是完整的, 包含你自己扩展的程序.

当然, 这样做还有点早. 你的扩展除了占用空间还应该做一些事情.

windows下静态构建

译者不熟悉windows环境, 因此略过.

功能函数

在用户空间和扩展代码之间最快捷的链接就是PHP_FUNCTION(). 首先在你的sample.c文件顶部, #include "php_sample.h"之后增加下面代码:

```
PHP_FUNCTION(sample_hello_world)
{
    php_printf("Hello World!\n");
}
```

PHP_FUNCTION()宏函数就像一个普通的C函数定义, 因为它按照下面方式展开:

```
#define PHP_FUNCTION(name) \
    void zif_##name(INTERNAL_FUNCTION_PARAMETERS)
```

这种情况下, 它就等价于:

```
void zif_sample_hello_world(zval *return_value,
    char return_value_used, zval *this_ptr TSRMLS_DC)
```

当然, 只定义函数还不够. 引擎需要知道函数的地址以及应该暴露给用户空间的函数名. 这通过下一个代码块完成, 你需要在PHP_FUNCTION()块后面增加:

```
static function_entry php_sample_functions[] = {
    PHP_FE(sample_hello_world,        NULL)
    { NULL, NULL, NULL }
};
```

php_sample_functions向量是一个简单的NULL结束向量, 它会随着你向扩展中增加功能而变大. 每个你要暴露出去的函数都需要在这个向量中给出说明. 展开PHP_FE()宏如下:

```
{ "sample_hello_world", zif_sample_hello_world, NULL},
```

这样提供了新函数的名字, 以及指向它的实现函数的指针. 这里第三个参数用于提供暗示信息, 比如某些参数需要引用传值. 在第7章"接受参数"你将看到这个特性.

现在, 你有了一个要暴露的函数列表, 但是仍然没有连接到引擎. 这通过对sample.c的最后一个修改完成, 将你的sample_module_entry结构体中的NULL, /* Functions */一行用php_sample_functions替换(请确保留下那个逗号).

现在, 通过前面介绍的方式重新构建, 使用php命令行的-r选项进行测试, -r允许你不用创建文件直接在命令行运行简单的代码片段:

```
$ php -r 'sample_hello_world();
```

如果一切OK的话, 你将会看到输出"Hello World!". 恭喜!!!

Zend内部函数

内部函数名前缀"zif_"是"Zend内部函数"的命名标准, 它用来避免可能的符号冲突. 比如, 用户空间的strlen()函数并没有实现为void strlen(INTERNAL_FUNCTION_PARAMETERS), 因为它会和C库的strlen冲突.

zif_的前缀也并不能完全避免名字冲突的问题. 因此, php提供了可以使用任意名字定义内部函数的宏: PHP_NAMED_FUNCTION(); 例如 PHP_NAMED_FUNCTION(zif_sample_hello_world)等同于前面使用的 PHP_FUNCTION(sample_hello_world)

当使用PHP_NAMED_FUNCTION定义实现时, 在function_entry向量中, 可以对应使用PHP_NAMED_FE()宏. 因此, 如果你定义了自己的函数 PHP_NAMED_FUNCTION(purplefunc), 就要使用 PHP_NAMED_FE(sample_hello_world, purplefunc, NULL), 而不是使用 PHP_FE(sample_hello_world, NULL).

我们可以在ext/standard/file.c中查看fopen()函数的实现, 它实际上使用 PHP_NAMED_FUNCTION(php_if_fopen)定义.从用户空间角度来看, 它并不关心函数是什么东西, 只是简单的调用fopen().

函数别名

有时一个函数可能会有不止一个名字. 回想一下, 普通的函数内部定义是用户空间函数名加上zif_前缀, 我们可以看到用PHP_NAMED_FE()宏可以很容易的创建这个可选映射.

```
PHP_FE(sample_hello_world,    NULL)
PHP_NAMED_FE(sample_hi,    zif_sample_hello_world,     NULL)
```

PHP_FE()宏将用户空间函数名sample_hello_world和 PHP_FUNCTION(sample_hello_world)展开而来的zif_sample_hello_world关联起来. PHP_NAMED_FE()宏则将用户空间函数名sample_hi和同一个内部实现关联起来.

现在, 假设Zend引擎发生了一个大的变更, 内部函数的前缀从zif_修改为pif_了. 这个时候, 你的扩展就不能工作了, 因为当到达PHP_NAMED_FE()时, 发现zif_sample_hello_world没有定义.

这种情况并不常见, 但非常麻烦, 可以使用PHP_FNAME()宏展开sample_hello_world避免这个问题:

```
PHP_NAMED_FE(sample_hi, PHP_FNAME(sample_hello_world), NULL)
```

这种情况下, 即便函数前缀被修改, 扩展的zend_function_entry也会使用宏扩展自动的更新.

现在, 你这样做已经可以工作了, 但是我们已经不需要这样做了. php暴露了另外一个宏, 专门设计用于创建函数别名. 前面的例子可以如下重写:

```
PHP_FALIAS(sample_hi, sample_hello_world, NULL)
```

实际上这是官方的创建函数别名的方法, 在php源码树中你时常会看到它.

小结

本章你创建了一个简单的可工作的php扩展, 并学习了在主要平台上构建它需要的步骤. 在以后的章节中, 你将会继续丰满这个扩展, 最终让它包含php的所有特性.

php源码树和它编译/构建在各个平台上依赖的工具经常会发生变化, 如果本章介绍的某些地方不能正常工作, 请参考php.net在线手册的Installation一掌, 查看你使用的版本的特殊需求.

返回值

用户空间函数利用return关键字向它的调用空间回传信息, 这一点和C语言的语法相同.

例如:

```
function sample_long() {
  return 42;
}
$bar = sample_long();
```

当sample_long()被调用时, 返回42并设置到$bar变量中. 在C语言中的等价代码如下:

```
int sample_long(void) {
  return 42;
}
void main(void) {
  int bar = sample_long();
}
```

当然, 在C语言中你总是知道被调用的函数是什么, 并且基于函数原型返回, 因此相应的你要定义返回结果存储的变量. 在php用户空间处理时, 变量类型是动态的, 转而依赖于第2章"变量的里里外外"中介绍的zval的类型.

return_value变量

你可能认为你的内部函数应该直接返回一个zval, 或者说分配一个zval的内存空间并如下返回zval *.

```
PHP_FUNCTION(sample_long_wrong)
{
    zval *retval;

    MAKE_STD_ZVAL(retval);
    ZVAL_LONG(retval, 42);

    return retval;
}
```

不幸的是, 这样做是不正确的. 并不是强制每个函数实现分配zval并返回它. 而是Zend引擎在函数调用之前预先分配这个空间. 接着将zval的类型初始化为IS_NULL, 并将值作为参数名return_value传递. 下面是正确的做法:

```
PHP_FUNCTION(sample_long)
{
    ZVAL_LONG(return_value, 42);
    return;
}
```

要注意的是PHP_FUNCTION()实现并不会直接返回任何值. 取而代之的是直接将恰当的数据弹出到return_value参数中, Zend引擎会在内部函数执行完成后处理它.

友情提示: ZVAL_LONG()宏是对多个赋值操作的一个封装:

```
Z_TYPE_P(return_value) = IS_LONG;
Z_LVAL_P(return_value) = 42;
```

或者更直接点:

```
return_value->type = IS_LONG;
return_value->value.lval = 42;
```

*return_value*的*is_ref*和*refcount*属性不应该被内部函数直接修改. 这些值由*Zend*引擎在调用你的函数时初始化并处理.

现在我们来看看这个特殊的函数, 将它增加到第5章"你的第一个扩展"中创建的sample扩展中. 只需要在sample_hello_world()函数下面添加这个函数, 并将sample_long()加入到php_sample_functions结构体中:

```
static function_entry php_sample_functions[] = {
```

```
    PHP_FE(sample_hello_world, NULL)
    PHP_FE(sample_long, NULL)
    { NULL, NULL, NULL }
};
```

现在我们就可以执行make重新构建扩展了.

如果一切OK, 可以运行php并测试新函数:

```
$ php -r 'var_dump(sample_long());
```

包装更紧凑的宏

在代码可读性和可维护性方面, ZVAL_*()宏中有重复的部分: return_value变量. 这种情况下, 将宏的ZVAL替换为RETVAL, 我们就可以在调用时省略return_value了.

前面的例子中, sample_long()的实现代码可以缩减到下面这样:

```
PHP_FUNCTION(sample_long)
{
    RETVAL_LONG(42);
    return;
}
```

下表列出了Zend引擎中RETVAL一族的宏. 除了两个特殊的, RETVAL宏除了删除了return_value参数之外, 和对应的ZVAL族宏相同.

普通的ZVAL宏	return_value专用宏
ZVAL_NULL(return_value)	RETVAL_NULL()
ZVAL_BOOL(return_value, bval)	RETVAL_BOOL(bval)
ZVAL_TRUE(return_value)	RETVAL_TRUE
ZVAL_FALSE(return_value)	RETVAL_FALSE
ZVAL_LONG(return_value, lval)	RETVAL_LONG(lval)
ZVAL_DOUBLE(return_value, dval)	RETVAL_DOUBLE(dval)
ZVAL_STRING(return_value, str, dup)	RETVAL_STRING(str, dup)
ZVAL_STRINGL(return_value, str, len, dup)	RETVAL_STRINGL(str, len, dup)
ZVAL_RESOURCE(return_value, rval)	RETVAL_RESOURCE(rval)

要注意到, *TRUE*和*FALSE*宏没有括号. 这是考虑到了*Zend/PHP*代码标准的偏差, 保留了主要的一种以保持向后兼容. 如果你构建扩展失败, 收到了错误消息*undefined macro RETVAL_TRUE()*, 请确认你是否在代码中写这两个宏时误写了括号.

通常, 在你的函数处理返回值的时候, 它已经准备好退出并将控制返回给调用作用域了. 由于这个原因, 为内部函数设计了另外一些宏用于返回: RETURN_*()族宏.

```
PHP_FUNCTION(sample_long)
{
    RETURN_LONG(42);
}
```

尽管看不到, 但这个函数在RETURN_LONG()宏调用完后的确会返回. 我们可以在函数末尾增加php_printf()进行测试:

```
PHP_FUNCTION(sample_long)
{
```

```
    RETURN_LONG(42);
    php_printf("I will never be reached.\n");
}
```

php_printf(), 如内容所描述的, 因为RETURN_LONG()调用隐式的结束了函数.

和RETVAL系列一样, 前面表中列出的每个简单类型都有对应的RETURN宏. 同样和RETVAL系列一样, RETURN_TRUE和RETURN_FALSE宏不使用括号.

更加复杂的类型, 比如对象和数组, 同样是通过return_value参数返回的; 然而, 它们天生就不能通过简单的宏创建. 即便是资源类型, 虽然它有RETVAL宏, 但是真正要返回资源类型还需要额外的工作. 你将在第8章到第11章看到怎样返回这些类型.

值得这么麻烦吗?

一个尚未使用的Zend内部函数特性是return_value_used参数. 考虑下面的用户空间代码:

```
function sample_array_range() {
    $ret = array();
    for($i = 0; $i < 1000; $i++) {
        $ret[] = $i;
    }
    return $ret;
}
sample_array_range();
```

因为sample_array_range()调用的时候并没有将结果存储到变量中, 这里创建使用的1000个元素的数组空间将被完全浪费. 当然, 这样调用sample_array_range()是愚蠢的, 但是没有很好的办法预知未来你又能怎么样呢?

虽然无法访问用户空间函数, 内部函数可以依赖于所有内部函数公共的return_value_used参数设置, 条件式的跳开这样的无意义行为.

```
PHP_FUNCTION(sample_array_range)
{
    if (return_value_used) {
        int i;
        /* 返回从0到999的数组 */
        array_init(return_value);
        for(i = 0; i < 1000; i++) {
            add_next_index_long(return_value, i);
        }
        return;
    } else {
        /* 提示错误 */
        php_error_docref(NULL TSRMLS_CC, E_NOTICE,
                "Static return-only function called without processing output");
        RETURN_NULL();
    }
}
```

要看到这个函数的操作, 只需要在你的sample.c源文件中增加这个函数, 并将它暴露在php_sample_functions结构中:

```
PHP_FE(sample_array_range, NULL)
```

译注: 关于用户空间不使用的函数返回值怎么处理, 可以阅读*Zend/zend_vm_execute.h*中的*zend_do_fcall_common_helper_SPEC*函数, 它在处理完内部函数调用后, 会检查该函数的返回值是否被使用, 如果不使用, 则进行了相应的释放.

返回引用值

从用户空间的php工作中你可能已经知道了, php函数还可以以引用方式返回值. 由于实现问题, 在php 5.1之前应该避免在内部函数中返回引用, 因为它不能工作. 考虑下面的用户空间代码片段:

```
function &sample_reference_a() {
```

```
    /* 如果全局空间没有变量$a, 就以初始值NULL创建它 */
    if (!isset($GLOBALS['a'])) {
        $GLOBALS['a'] = NULL;
    }
    return $GLOBALS['a'];
}
$a = 'Foo';
$b = sample_reference_a();
$b = 'Bar';
```

在这个代码片段中, 就像使用$b = &$GLOBALS['a];或者由于在全局空间, 使用$b = &$a; 将$b创建为$a的一个引用.

回顾第3章"内存管理", 在到达最后一行时, $a和$b实际上包含相同的值'Bar'. 现在我们看看内部实现这个函数:

```
#if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && \
                          PHP_MINOR_VERSION > 0)
PHP_FUNCTION(sample_reference_a)
{
    zval **a_ptr, *a;


    /* 从全局符号表查找变量$a */
    if (zend_hash_find(&EG(symbol_table), "a", sizeof("a"),
                                        (void**)&a_ptr) == SUCCESS) {
        a = *a_ptr;
    } else {
        /* 如果不存在$GLOBALS['a']则创建它 */
        ALLOC_INIT_ZVAL(a);
        zend_hash_add(&EG(symbol_table), "a", sizeof("a"), &a,
                                            sizeof(zval*), NULL);
    }
    /* 废弃旧的返回值 */
    zval_ptr_dtor(return_value_ptr);
    if (!a->is_ref && a->refcount > 1) {
        /* $a需要写时复制, 在使用之前, 必选先隔离 */
        zval *newa;
        MAKE_STD_ZVAL(newa);
        *newa = *a;
        zval_copy_ctor(newa);
        newa->is_ref = 0;
        newa->refcount = 1;
        zend_hash_update(&EG(symbol_table), "a", sizeof("a"), &newa,
                                               sizeof(zval*), NULL);
        a = newa;
    }
    /* 将新的返回值设置为引用方式并增加refcount */
    a->is_ref = 1;
    a->refcount++;
    *return_value_ptr = a;
}
#endif /* PHP >= 5.1.0 */
```

return_value_ptr参数是所有内部函数都会传递的另外一个公共参数, 它是zval **类型, 包含了指向return_value的指针. 通过在它上面调用zval_ptr_dtor(), 默认的return_value的zval *将被释放. 接着可以自由的选择一个新的zval *去替代它, 在这里选择了变量$a, 选择性的进行zval隔离后, 将它的is_ref设置为1, 设置到return_value_ptr中.

如果现在编译运行这段代码, 无论如何你会得到一个段错误. 为了使它可以工作, 你需要在php_sample.h中增加下面的代码:

```
#if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && \
                          PHP_MINOR_VERSION > 0)
static
    ZEND_BEGIN_ARG_INFO_EX(php_sample_retref_arginfo, 0, 1, 0)
```

```
    ZEND_END_ARG_INFO ()
#endif /* PHP >= 5.1.0 */
```

译注: 译者使用的*php-5.4.9*中, *ZEND_BEGIN_ARG_INFO_EX*的宏定义中已经包含了*static*修饰符, 因此本书示例中相应的需要进行修改, 请读者在阅读过程中注意这一点.

接着, 在php_sample_functions中声明你的函数时使用这个结构:

```
#if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && \
                        PHP_MINOR_VERSION > 0)
    PHP_FE(sample_reference_a, php_sample_retref_arginfo)
#endif /* PHP >= 5.1.0 */
```

这个结构你将在本章后面详细学习, 用来向Zend引擎提供函数调用的重要暗示信息. 这里它告诉Zend引擎return_value需要被覆写, 应该从return_value_ptr中得到正确的地址. 如果没有这个暗示, Zend引擎会简单的在return_value_ptr中设置NULL, 这可能使得在执行到zval_ptr_dtor()时程序崩溃.

这段代码每一个片段都包裹在*#if*块中, 它指示编译器只有在*PHP*版本大于等于*5.1*时才启用这个支持. 如果没有这些条件指令, 这个扩展将不能在*php4*上编译*(*因为在*return_value_ptr*中包含的一些元素不存在*)*, 在*php5.0*中不能提供正确的功能*(*有一个*bug*导致返回的引用被以值的方式拷贝*)*

引用方式返回值

使用return(语法)结构将值和变量回传给调用方是没有问题的, 但是, 有时你需要从一个函数返回多个值. 你可以使用数组(我们将在第8章"使用数组和哈希表工作")达到这个目的, 或者你可以使用参数栈返回值.

调用时引用传值

一种简单的引用传递变量方式是要求调用时在参数变量名前使用取地址符(&), 如下用户空间代码:

```
function sample_byref_calltime($a) {
    $a .= ' (modified by ref!)';
}
$foo = 'I am a string';
sample_byref_calltime(&$foo);
echo $foo;
```

参数变量名前的取地址符(&)使得发送给函数的是$foo实际的zval, 而不是它的内容拷贝.这就使得函数可以通过传递的这个参数修改这个值来返回信息. 如果调用sample_byref_calltime()时没有在$foo前面使用取地址符(&), 则函数内的修改并不会影响原来的变量.

在C层面重演这个行为并不需要特殊的编码. 在你的sample.c源码文件中sample_long()后面创建下面的函数:

```
PHP_FUNCTION(sample_byref_calltime)
{
    zval *a;
    int addtl_len = sizeof(" (modified by ref!)") - 1;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &a) == FAILURE) {
        RETURN_NULL();
    }
    if (!a->is_ref) {
        /* 不是以引用方式传值则不做任何事离开函数 */
        return;
    }
    /* 确保变量是字符串 */
    convert_to_string(a);
    /* 扩大a的缓冲区以使可以保存要追加的数据 */
    Z_STRVAL_P(a) = erealloc(Z_STRVAL_P(a),
        Z_STRLEN_P(a) + addtl_len + 1);
    memcpy(Z_STRVAL_P(a) + Z_STRLEN_P(a),
```

```
    " (modified by ref!)", addtl_len + 1);
    Z_STRLEN_P(a) += addtl_len;
}
```

和往常一样, 这个函数需要增加到php_sample_functions结构中.

```
PHP_FE(sample_byref_calltime,        NULL)
```

译注: 运行时引用传值在译者使用的版本*(php-5.4.9 ZendEngine 2.4.0)*中已经被完全废弃. 早前的版本可以在*php.ini*中使用*allow_call_time_pass_reference*指令启用. 测试时请注意版本问题.

编译期引用传值

更常用的方式是编译期引用传值. 这里函数的参数定义为只能以引用方式使用, 传递常量或其他中间值(比如函数调用的结果)将导致错误, 因为那样函数就没有地方可以存储结果值去回传了. 用户空间的编译期引用传值代码如下:

```
function sample_byref_compiletime(&$a) {
    $a .= ' (modified by ref!)';
}
$foo = 'I am a string';
sample_byref_compiletime($foo);
echo $foo;
```

如你所见, 这和调用时引用传值的差别仅在于取地址符的位置不同. 在C的层面上去看这个函数时, 函数代码上是完全相同的. 唯一的区别在于php_sample_functions块中对函数的声明:

```
PHP_FE(sample_byref_compiletime, php_sample_byref_arginfo)
```

php_sample_byref_arginfo是一个常量结构, 你需要在使用之前先定义它.

实际上, 编译期引用传值中, 对*is_ref*检查的代码可以删除, 因为总能保证它是引用方式的. 不过这里留着它也不会有什么危害.

在Zend引擎1(php4)中, 这个结构是一个简单的char *列表, 第一个元素指定了长度, 接下来是描述每个函数参数的标记集合.

```
static unsigned char php_sample_byref_arginfo[] =
                                { 1, BYREF_FORCE };
```

这里, 1表示向量只包含了一个参数的信息. 后面的元素就顺次描述参数特有的标记信息, 第二个元素描述第一个参数. 如果涉及到第二个或第三个参数, 对应的就需要在这个向量中增加第三个和第四个元素. 参数信息的可选值如下表:

标记类型	含义
BYREF_NONE	这个参数永远都不允许引用传值. 尝试使用调用时引用传值将被忽略, 参数仍然会被拷贝.
BYREF_FORCE	参数永远都是引用传值, 无论怎样调用. 这等价于在用户空间函数定义时使用取地址符(&)
BYREF_ALLOW	参数是否引用传值取决于调用时的语义. 这等价于普通的用户空间函数定义.
BYREF_FORCE_REST	当前参数以及后面所有参数都将应用BYREF_FORCE. 这个标记只能作为列表的最后一个标记. 在BYREF_FORCE_REST后面放置其他标记可能导致未定义行为.

在Zend引擎2(php 5+)中, 你将使用一种更加可扩展的结构, 它包含类更多的信息, 比如最小和最大参数要求, 类型暗示, 是否强制引用等.

首先, 参数信息结构使用两个宏中的一个定义. 较简单的一个是ZEND_BEGIN_ARG_INFO(), 它需要两个参数:

```
ZEND_BEGIN_ARG_INFO(name, pass_rest_by_reference)
```

name非常简单, 就是扩展中其他地方使用这个结构时的名字, 当前这个例子我们使用的名字是: php_sample_byref_arginfo

pass_rest_by_reference的含义和BYREF_FORCE_REST用在Zend引擎1的参数信息向量最后一个元素时一致. 如果这个参数设置为1, 所有没有在结构中显式描述的参数都被认为是编译期引用传值的参数.

还有一种可选的开始宏, 它引入了两个Zend引擎1没有的新选项, 它是ZEND_BEGIN_ARG_INFO_EX():

```
ZEND_BEGIN_ARG_INFO_EX(name, pass_rest_by_reference, return_reference,
                       required_num_args)
```

当然, name和pass_rest_by_reference和前面所说的是相同的含义. 本章前面也提到了, return_reference是告诉Zend你的函数需要用自己的zval覆写return_value_ptr.

最后一个参数, required_num_args, 它是另外一种类型的暗示, 用来告诉Zend在某种被认为是不完整的调用时完全的跳过函数调用.

在你拥有了一个恰当的开始宏后, 接下来就可以是0个或多个ZEND_ARG_*INFO元素. 这些宏的类型和用法如下表.

宏	用途
ZEND_ARG_PASS_INFO(by_ref)	by_ref和所有后面的宏一样是一个二选一的选项, 它用来标识是否对应的参数应该被强制为引用传值. 设置这个选项为1等同于在Zend引擎1中使用BYREF_FORCE.
ZEND_ARG_INFO(by_ref, name)	这个宏提供了一个附加的name属性, 用于内部生成的错误消息和反射API. 它应该被设置成一些非加密的帮助信息.
ZEND_ARG_ARRAY_INFO(by_ref, name, allow_null)	这两个宏提供了内部函数的参数类型暗示, 用来描述参数只能是数组或某个特定类型的实例. 将allow_null设置为非0值将允许调用时在放置array/object的地方传递NULL值.
ZEND_ARG_OBJ_INFO(by_ref, name, classname, allow_null)	

最后, 所有使用Zend引擎2的宏设置的参数信息结构必须使用ZEND_END_ARG_INFO()结束. 对于你的sample函数, 你需要选择一个如下的结构:

```
ZEND_BEGIN_ARG_INFO(php_sample_byref_arginfo, 0)
    ZEND_ARG_PASS_INFO(1)
ZEND_END_ARG_INFO()
```

为了让扩展兼容Zend引擎1和2, 需要使用#ifdef语句为两者均定义arg_info结构:

```
#ifdef ZEND_ENGINE_2
static
    ZEND_BEGIN_ARG_INFO(php_sample_byref_arginfo, 0)
        ZEND_ARG_PASS_INFO(1)
    ZEND_END_ARG_INFO()
#else /* ZE 1 */
static unsigned char php_sample_byref_arginfo[] =
                                { 1, BYREF_FORCE };
#endif
```

注意, 这些代码片段是集中在一起的, 现在是时候创建一个真正的编译期引用传值实现了. 首先, 我们将为Zend引擎1和2定义的php_sample_byref_arginfo块放到头文件php_sample.h中.

接下来, 可以有两种选择, 一种是将PHP_FUNCTION(sample_byref_calltime)拷贝一份, 并重命名为PHP_FUNCTION(sample_byref_compiletime), 接着在php_sample_functions中增加一行PHP_FE(sample_byref_compiletime, php_sample_byref_arginfo)

这种方式简单移动, 并且在一段时候后修改时, 更加不容易产生混淆. 因为这里只是示例代码, 因此, 我们可以稍微放松点, 使用你在上一章学的PHP_FALIAS()避免代码重复.

这样, 就不是赋值PHP_FUNCTION(sample_byref_calltime), 而是在php_sample_functions中直接增加一行:

```
PHP_FALIAS(sample_byref_compiletime, sample_byref_calltime,
    php_sample_byref_arginfo)
```

回顾第5章, 这将创建一个名为sample_byref_compiletime()的用户空间函数, 它对应的内部实现是sample_byref_calltime()的代码. php_sample_byref_arginfo是这个版本的特殊之处.

小结

本章你看到了怎样从一个内部函数返回值, 包括直接返回值和引用方式返回, 以及通过参数栈引用返回. 此外还简单了解了Zend引擎2的参数类型暗示结构zend_arg_info.

下一章你将会继续探究接受基本的zval参数以及使用zend_parse_parameters()强大的类型戏法.

接受参数

除了几个"预览"的例外, 你迄今处理的扩展函数都很简单, 只有返回. 然而, 多数函数并非只有一个目的. 你通常会传递一些参数, 并希望接收到基于值和其他附加处理的有用的响应.

zend_parse_parameters()的自动类型转换

和上一章你看到的返回值一样, 参数的值也是围绕着对zval引用的间访展开的. 获取这些zval*的值最简单的方法就是使用zend_parse_parameters()函数.

调用zend_parse_parameters()几乎总是以ZEND_NUM_ARGS()宏接着是无所不在的TSRMLS_CC. ZEND_NUM_ARGS()从名字上可以猜到, 它返回int型的实际传递的参数个数. 由于zend_parse_parameters()内部工作的方法, 你可能不需要直接了解这个值, 因此现在只需要传递它.

zend_parse_parameters()的下一个参数是格式串参数, 它是由Zend引擎支持的基础类型描述字符组成的字符序列, 用来描述要接受的函数参数. 下表是基础的类型字符:

类型字符	用户空间数据类型
b	Boolean
l	Integer
d	Floating point
s	String
r	Resource
a	Array
o	Object instance
O	Object instance of a specified type
z	Non-specific zval
Z	Dereferenced non-specific zval

zend_parse_parameters()剩下的参数依赖于你的格式串中所指定的类型描述. 对于简单类型, 直接解引用为C语言的基础类型. 例如, long数据类型如下解出:

```
PHP_FUNCTION(sample_getlong)
{
    long foo;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                        "l", &foo) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("The integer value of the parameter you "
             "passed is: %ld\n", foo);
    RETURN_TRUE;
}
```

尽管*integer*和*long*的存储空间通常是一样大的, 但它们并不完全一致. 尝试将一个*int*类型的数据解引用到*long *参数可能会带来不期望的结果, 尤其是在*64*位平台上更加容易出错. 因此, 请严格使用下表中列出的数据类型.

类型	C语言中的对应数据类型
b	zend_bool
l	long
d	double
s	char *, int
r	zval *
a	zval *
o	zval *
O	zval *, zend_class_entry *
z	zval *
Z	zval *

注意, 所有其他的复杂类型实际上都解析为简单的zval. 这样做的原因和不使用RETURN_*()宏返回复杂数据类型一样, 都是受限于无法真正的模拟C空间中的这些结构. zend_parse_parameters()能为你的函数所做的, 是确保你接收到的zval *是正确的类型. 如果需要, 它甚至会执行隐式的类型转换, 比如将数组转换为stdClass的对象.

s和O类型需要单独说明, 因为它们一次调用需要两个参数. 在第10章"php4对象"和第11章"php5对象"中你将更进一部的了解O. 对于s这个类型, 我们对第5章"你的第一个扩展"的sample_hello_world()函数进行一次扩展, 让它可以跟指定的人名打招呼.

```
function sample_hello_world($name) {
    echo "Hello $name!\n";
}
/* 在C中, 你将使用zend_parse_parameters()函数接受一个字符串 */
PHP_FUNCTION(sample_hello_world)
{
    char *name;
    int name_len;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
                        &name, &name_len) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("Hello ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

*zend_parse_parameters()*函数可能由于函数传递的参数太少不能满足格式串, 或者因为其中某个参数不能转换为请求的类型而失败. 这种情况下, 它将自动的输出错误消息, 因此你的扩展不需要这样做.

要请求超过1个参数, 就需要扩充格式串, 包括其他字符, 并将zend_parse_parameters()调用的后面其他参数压栈. 参数和它们在用户空间函数定义的行为一致, 从左向右解析.

```
function sample_hello_world($name, $greeting) {
    echo "Hello $greeting $name!\n";
}
sample_hello_world('John Smith', 'Mr.');
Or:
PHP_FUNCTION(sample_hello_world)
{
    char *name;
    int name_len;
    char *greeting;
    int greeting_len;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",
      &name, &name_len, &greeting, &greeting_len) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("Hello ");
    PHPWRITE(greeting, greeting_len);
    php_printf(" ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

除了基础类型, 还有3个元字符用于修改参数处理方式. 如下表所示:

类型修改符	含义
|	接下来是可选参数了. 当指定它时, 所有之前的参数都被认为是必须的, 所有后续的参数都被认为是可选的.
!	!之前的一个修饰符对应的参数如果是NULL, 提供的内部变量将被设置为真实的NULL指针.
/	/之前的一个修饰符对应的参数指定为写时拷贝, 它将自动的隔离到新的zval(is_ref = 0, refcount = 1)

可选参数

我们再来看一看修订版的sample_hello_world()示例, 下一步是增加一个可选的$greeting参数:

```
function sample_hello_world($name, $greeting='Mr./Ms.') {
    echo "Hello $greeting $name!\n";
}
```

sample_hello_world()现在可以只用$name参数调用, 也可以同时使用两个参数调用.

```
sample_hello_world('Ginger Rogers','Ms.');
sample_hello_world('Fred Astaire');
```

当不传递第二个参数时, 使用默认值. 在C语言实现中, 可选参数也是以类似的方式指定.

要完成这个功能, 我们需要使用zend_parse_parameters()格式串中的管道符(|). 管道符左侧的参数从调用栈解析, 所有管道符右边的参数如果在调用栈中没有提供, 则不会被修改(zend_parse_parameters()格式串后面对应的参数). 例如:

```
PHP_FUNCTION(sample_hello_world)
{
    char *name;
    int name_len;
    char *greeting = "Mr./Mrs.";
    int greeting_len = sizeof("Mr./Mrs.") - 1;

    /* 如果调用时没有传递第二个参数, 则greeting和greeting_len保持不变. */
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s|s",
      &name, &name_len, &greeting, &greeting_len) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("Hello ");
    PHPWRITE(greeting, greeting_len);
    php_printf(" ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

除非调用时提供了可选参数的值, 否则不会修改它的初始值, 因此, 为可选参数设置初始的默认值非常重要. 多数情况下, 它的初始默认值是NULL/0, 不过有时候, 比如上面的例子, 默认的是有意义的其他值.

IS_NULL Vs. NULL

每个zval, 即便是非常简单的IS_NULL类型, 都会占用一块很小的内存空间. 从而, 它就需要一些(CPU)时钟周期去分配内存空间, 初始化值, 最后认为不再需要它的时候释放它.

对于很多函数, 在调用空间使用NULL参数只是标记参数不重要, 因此这个处理就没有意义. 幸运的是zend_parse_parameters()允许参数被标记为"允许NULL", 如果在一个格式描述字符后加上感叹号(!), 则表示对应参数如果传递了NULL, 则将zend_parse_parameters()调用时对应的参数设置为真正的NULL指针. 考虑下面两段代码, 一个有这个修饰符, 一个没有:

```
PHP_FUNCTION(sample_arg_fullnull)
{
    zval *val;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z",
                                  &val) == FAILURE) {
        RETURN_NULL();
    }
    if (Z_TYPE_P(val) == IS_NULL) {
        val = php_sample_make_defaultval(TSRMLS_C);
    }
...
PHP_FUNCTION(sample_arg_nullok)
{
    zval *val;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z!",
                                  &val) == FAILURE) {
        RETURN_NULL();
    }
    if (!val) {
        val = php_sample_make_defaultval(TSRMLS_C);
    }
...
```

这两个版本的代码其实没什么不同, 前者表面上看起来需要更多的处理时间. 通常, 这个特性并不是很有用, 但最好还是知道有这么回事.

强制隔离

当一个变量传递到一个函数中后, 无论是否是引用传值, 它的refcount至少是2; 一个是变量自身, 另外一个是传递给函数的拷贝. 因此在修改zval之前(如果直接在参数传递进来的zval上), 将它从它所属的非引用集合中分离出来非常重要.

如果没有"/"格式修饰符, 这将是一个单调重复的工作. 这个格式修饰符自动的隔离所有写时拷贝的引用传值参数, 这样, 你的函数中就可以为所欲为了. 和NULL标记一样, 这个修饰符放在它要修饰的格式描述字符后面. 同样和NULL标记一样, 直到你真的有使用它的地方, 你可能才知道你需要这个特性.

zend_get_arguments()

如果你正在设计的代码计划在非常旧的php版本上工作, 或者你有一个函数, 它只需要zval *, 就可以考虑使用zend_get_parameters()的API调用.

zend_get_parameters()调用与它对应的新版本有一点不同. 首先, 它不会自动执行类型转换; 而是所有期望的参数都是zval *数据类型. 下面是zend_get_parameters()的最简单用法:

```
PHP_FUNCTION(sample_onearg)
{
    zval *firstarg;
    if (zend_get_parameters(ZEND_NUM_ARGS(), 1, &firstarg)
                                    == FAILURE) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "Expected at least 1 parameter.");
        RETURN_NULL();
    }
    /* Do something with firstarg... */
}
```

其次, 你可能已经注意到, 它需要手动处理错误消息, zend_get_parameters()并不会在失败的时候输出错误文本. 它对可选参数的处理也很落后. 如果你让它抓取4个参数, 最好至少给它提供4个参数, 否则可能返回FAILURE.

最后, 它不像parse, 这个get变种会自动的隔离所有写时复制的引用集合. 如果你想要跳过自动隔离, 就要使用它的兄弟接口: zend_get_parameters_ex().

除了不隔离写时复制集合, zend_get_parameters_ex()还有一个不同点是返回zval **指针而不是直接的zval *. 它们的差别同样可能直到你使用的时候才知道需要它们, 不过它们最终的使用非常相似:

```
PHP_FUNCTION(sample_onearg)
{
    zval **firstarg;
    if (zend_get_parameters_ex(1, &firstarg) == FAILURE) {
        WRONG_PARAM_COUNT;
    }
    /* Do something with firstarg... */
}
```

要注意的是*_ex*版本不需要*ZEND_NUM_ARGS()*参数. 这是因为增加*_ex*的版本时已经比较晚了, 当时*Zend*引擎已经不需要这个参数了.

在这个例子中, 你还使用了*WRONG_PARAM_COUNT*宏, 它用来处理*E_WARNING*错误消息的显示已经自动的离开函数.

处理任意数目的参数

还有两个zend_get_parameters()族的函数, 用于解出zval *和zval **指针集合, 它们适用于有很多参数或运行时才知道参数个数的引用场景.

考虑var_dump()函数, 它用来展示传递给它的任意数量的变量的内容:

```
PHP_FUNCTION(var_dump)
{
    int i, argc = ZEND_NUM_ARGS();
    zval ***args;


    args = (zval ***)safe_emalloc(argc, sizeof(zval **), 0);
    if (ZEND_NUM_ARGS() == 0 ||
        zend_get_parameters_array_ex(argc, args) == FAILURE) {
        efree(args);
        WRONG_PARAM_COUNT;
    }
    for (i=0; i<argc; i++) {
        php_var_dump(args[i], 1 TSRMLS_CC);
    }
    efree(args);
}
```

这里, var_dump()预分配了一个传给函数的参数个数大小的zval **指针向量. 接着使用zend_get_parameters_array_ex()一次性将参数摊入到该向量中. 你可能会猜到, 存在这个函数的另外一个版本: zend_get_parameters_array(), 它们仅有一点不同: 自动隔离, 返回zval *而不是zval **, 在第一个参数上要求传递ZEND_NUM_ARGS().

参数信息和类型暗示

在上一章已经简短的向你介绍了使用Zend引擎2的 参数信息结构进行类型暗示的概念. 我们应该记得, 这个特性是针对ZE2(Zend引擎2)的, 对于php4的ZE1(Zend引擎1)不可用.

我们重新从ZE2的参数信息结构开始. 每个参数信息都使用ZEND_BEGIN_ARG_INFO()或ZEND_BEGIN_ARG_INFO_EX()宏开始, 接着是0个或多个ZEND_ARG_*INFO()行, 最后以ZEND_END_ARG_INFO()调用结束.

这些宏的定义和基本使用可以在刚刚结束的第6章"返回值"的编译期引用传值一节中找到.

假设你想要重新实现count()函数, 你可能需要创建下面这样一个函数:

```
PHP_FUNCTION(sample_count_array)
{
    zval *arr;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a",
                                  &arr) == FAILURE) {
        RETURN_NULL();
    }
    RETURN_LONG(zend_hash_num_elements(Z_ARRVAL_P(arr)));
}
```

zend_parse_parameters()会做很多工作, 以保证传递给你的函数是一个真实的数组. 但是, 如果你使用zend_get_parameters()函数或某个它的兄弟函数, 你就需要自己在函数中做类型检查. 当然, 你也可以使用类型暗示! 通过定义下面这样一个arg_info结构:

```
static
    ZEND_BEGIN_ARG_INFO(php_sample_array_arginfo, 0)
        ZEND_ARG_ARRAY_INFO(0, "arr", 0)
    ZEND_END_ARG_INFO()
```

并在你的php_sample_function结构中声明该函数时使用它:

```
PHP_FE(sample_count_array, php_sample_array_arginfo)
```

这样就将类型检查的工作交给了Zend引擎. 同时你给了你的参数一个名字(arr), 它可以在产生错误消息时被使用, 这样在使用你API的脚本发生错误时, 更加容易跟踪问题.

在第6章第一次介绍参数信息结构时, 你可能注意到了对象, 也可以使用ARG_INFO宏进行类型暗示. 只需要在参数名后增加一个额外的参数来说明类型名(类名)

```
static
    ZEND_BEGIN_ARG_INFO(php_sample_class_arginfo, 0)
        ZEND_ARG_OBJECT_INFO(1, "obj", "stdClass", 0)
    ZEND_END_ARG_INFO()
```

要注意到这里的第一个参数(by_ref)被设置为1. 通常来说这个参数对对象来说并不是很重要, 因为ZE2中所有的对象默认都是引用方式的, 对它们的拷贝必须显式的通过clone来实现. 在函数调用内部强制clone可以做到, 但是它和强制引用完全不同.

因为当设置了zend.ze1_compatiblity_mode标记时, 你可能关心ZEND_ARG_OBJECT_INFO一行的by_ref设置. 这种特殊情况下, 对象可能仍然传递的是一个拷贝而不是引用. 因为你在处理对象的时候, 可能需要的是一个真正的引用, 因此设置这个标记你就不用担心这一方面的影响.

不要忘记了数组和对象的参数信息宏中有一个*allow_null*选项. 关于允许*NULL*的细节请参考前一章的编译期引用传值一节.

当然, 使用使用参数信息进行类型暗示只在ZE2中支持, 如果你想让你看的扩展兼容php4, 需要使用zend_get_parameters(), 这样就只能将类型验证放到函数内部, 手动的通过测试Z_TYPE_P(value)或使用第2章看到的convert_to_type()方法进行自动类型转换来完成.

总结

现在你的手头可能已经有点脏乱了, 和用户空间通信的功能代码通过简单的输入/输出函数实现. 已经比较深入的了解了zval的引用计数系统, 并学习了控制变量传递到你的内部函数方法和时机.

下一章将开始学习数组数据类型, 并了解用户空间的数组表示怎样映射到内部的HashTable实现. 此外还将看到一大批可选的Zend以及php api函数, 它们用于操纵这些复杂的结构体.

在数组和哈希表上工作

在C语言中, 有两种不同的基础方法用来在一个结构体中存储任意数量的独立数据元素. 两种方法都有赞成者和反对者.

向量 Vs. 链表

应用的编写通常基于特定类型数据的特性的选择, 需要存储多少数据, 以及需要多快速度的检索. 为了能够有对等的认知, 我们先来看看简单的看看这些存储机制.

向量

向量是一块连续的内存空间, 它们包含的数据有规律的间隔. 向量最常见的例子就是字符串变量(char *或char []), 它包含了一个接着一个的字符(字节)序列.

```
char foo[4] = "bar";
```

这里, foo[0]包含了字符'b'; 紧接着, 你将在foo[1]中找到字符'a', 最后在foo[3]中是一个空字符'\0'.

将指向其他结构的指针存储到向量中的用法几乎是无所不在的, 比如在上一章, 使用zend_get_parameters_array_ex()函数时, 就使用了一个zval的向量. 那里, 我们看到var_dump()定义了一个zval ***的函数变量, 接着为它分配空间用来存储zval **指针(最终的数据来自zend_get_parameters_ex()调用)

```
zval ***args = safe_emalloc(ZEND_NUM_ARGS(), sizeof(zval**), 0);
```

和访问字符串中的数组一样, var_dump()实现中使用args[i]依次传递每个zval **元素到内部函数php_var_dump().

向量最大的优点在于运行时单个元素的访问速度. args[i]这样的变量引用, 可以很快的计算出它的数据地址(args + i * sizeof(args[0]). 这个索引结构的空间分配和释放是在单次, 高效的调用中完成的.

链表

另外一种常见的存储数据的方式是链表. 对于链表而言, 每个数据元素都是一个至少有两个属性的结构体: 一个指向链表中的下一个节点, 一个则是实际的数据. 考虑下面假设的数据结构:

```
typedef struct _namelist namelist;
struct {
    struct namelist *next;
    char *name;
} _namelist;
```

使用这个数据结构的引用需要定义一个变量:

```
static namelist *people;
```

链表中的第一个名字可以通过检查people变量的name属性得到: people->name; 第二个名字则访问next属性: people->next->name, 依此类推: people->next->next->name等等, 直到next为NULL表示链表中已经没有其他名字了. 更常见的用法是使用循环迭代链表:

```
void name_show(namelist *p)
{
    while (p) {
        printf("Name: %s\n", p->name);
        p = p->next;
    }
}
```

这种链表非常适合于FIFO的链式结构, 新的数据被追加到链表的末尾, 从另外一端线性的消耗数据:

```
static namelist *people = NULL, *last_person = NULL;
```

```
void name_add(namelist *person)
{
    person->next = NULL;
    if (!last_person) {
        /* 链表中没有数据 */
        people = last_person = person;
        return;
    }
    /* 向链表末尾增加新的数据 */
    last_person->next = person;

    /* 更新链表尾指针 */
    last_person = person;
}
namelist *name_pop(void)
{
    namelist *first_person = people;
    if (people) {
    people = people->next;
    }
    return first_person;
}
```

新的namelist结构可以从这个链表中多次插入或弹出, 而不用调整结构的大小或在某些位置间块拷贝元素.

前面你看到的链表只是一个单链表, 虽然它有一些有趣的特性, 但它有致命的缺点. 给出链表中一项的指针, 将它从链中剪切出来并确保前面的元素正确的链接上下一个元素就变得比较困难.

为了知道它的前一个元素, 就需要遍历整个链表直到找到一个元素的next指针指向要被删除的元素. 对于大的链表, 这可能需要可观的CPU时间. 一个简单的相对廉价的解决方案是双链表.

对于双链表而言, 每个元素增加了一个指针元素, 它指向链表中的前一个元素:

```
typedef struct _namelist namelist;
struct {
    namelist *next, *prev;
    char *name;
} _namelist;
```

一个元素被添加到双链表的时候, 这两个指针相应的都被更新:

```
void name_add(namelist *person)
{
    person->next = NULL;
    if (!last_person) {
        /* 链表中没有元素 */
        people = last_person = person;
        person->prev = NULL;
        return;
    }
    /* 在链表尾增加一个新元素 */
    last_person ->next = person;
    person->prev = last_person;

    /* 更新链表尾指针 */
    last_person = person;
}
```

迄今为止, 你还没有看到这种数据结构的任何优势, 但是现在你可以想想, 给出people链表中间的一条任意的namelist记录, 怎样删除它. 对于单链表, 你需要这样做:

```
void name_remove(namelist *person)
{
    namelist *p;
    if (person == people) {
        /* 要删除链表头指针 */
```

```
        people = person->next;
        if (last_person == person) {
            /* 要删除的节点同时还是尾指针 */
            last_person = NULL;
        }
        return;
    }
    /* 搜索要删除节点的前一个节点 */
    p = people;
    while (p) {
        if (p->next == person) {
            /* 删除 */
            p->next = person->next;
            if (last_person == person) {
                /* 要删除的节点是头指针 */
                last_person = p;
            }
            return;
        }
        p = p->next;
    }
    /* 链表中没有找到对应元素 */
}
```

现在和双链表的代码比较一下:

```
void name_remove(namelist *person)
{
    if (people == person) {
        people = person->next;
    }
    if (last_person == person) {
        last_person = person->prev;
    }
    if (person->prev) {

        person->prev->next = person->next;
    }
    if (person->next) {
        person->next->prev = person->prev;
    }
}
```

不是很长, 也没有循环, 从链表中删除一个元素只需要简单的执行条件语句中的重新赋值语句. 与此过程相逆的过程就可以同样高效的将元素插入到链表的任意点.

最好的是HashTable

虽然在你的应用中你完全可以使用向量或链表, 但有另外一种集合数据类型, 最终你可能会更多的使用: HashTable.

HashTable是一种特殊的双链表, 它增加了前面看到的向量方式的高效查找. HashTable在Zend引擎和php内核中使用的非常多, 整个ZendAPI都子例程都主要在处理这些结构.

如你在第2章"变量的里里外外"中所见, 所有的用户空间变量都存储在一个zval *指针的HashTable中. 后面章节中你可以看到Zend引擎使用HashTable存储用户空间函数, 类, 资源, 自动全局标记以及其他结构.

回顾第2章, Zend引擎的HashTable可以原文存储任意大小的任意数据片. 比如, 函数存储了完整的结构. 自动全局变量只是很少几个字节的元素, 然而其他的结构, 比如php5的类定义则只是简单的存储了指针.

本章后面我们将学习构成Zend Hash API的函数调用, 你可以在你的扩展中使用这些函数.

Zend Hash API

Zend Hash API被分为几个基本的打雷, 除了几个特殊的, 这些函数通常都返回SUCCESS或FAILURE.

创建

每个HashTable都通过一个公用的构造器初始化:

```
int zend_hash_init(HashTable *ht, uint nSize,
    hash_func_t pHashFunction,
    dtor_func_t pDestructor, zend_bool persistent)
```

ht是一个指向HashTable变量的指针, 它可以定义为直接值形式, 也可以通过emalloc()/pemalloc()动态分配, 或者更常见的是使用ALLOC_HASHTABLE(ht). ALLOC_HASHTABLE()宏使用了一个特定内存池的预分配块来降低内存分配所需的时间, 相比于ht = emalloc(sizeof(HashTable));它通常是首选.

nSize应该被设置为HashTable期望存储的最大元素个数. 如果向这个HashTable中尝试增加多于这个数的元素, 它将会自动增长, 不过有一点需要注意的是, 这里Zend重建整个新扩展的HashTable的索引的过程需要耗费不少的处理时间. 如果nSize不是2的幂, 它将被按照下面公式扩展为下一个2的幂:

```
nSize = pow(2, ceil(log(nSize, 2)));
```

pHashFunction是旧版本Zend引擎的遗留参数, 它不在使用, 因此这个值应该被设置为NULL. 在早期的Zend引擎中, 这个值指向一个用以替换标准的DJBX33A(一种常见的抗碰撞哈希算法, 用来将任意字符串key转换到可重演的整型值)的可选哈希算法.

pDestructor指向当从HashTable删除元素时应该被调用的函数, 比如当使用zend_hash_del()删除或使用zend_hash_update()替换. 析构器函数的原型如下:

```
void method_name(void *pElement);
```

pElement指向指向要从HashTable中删除的元素.

最后一个选项是persistent, 它只是一个简单的标记, 引擎会直接传递给在第3章"内存管理"中学习的pemalloc()函数. 所有需要保持跨请求可用的HashTable都必须设置这个标记, 并且必须调用pemalloc()分配.

这个方法的使用在所有php请求周期开始的时候都可以看到: EG(symbol_table)全局变量的初始化:

```
zend_hash_init(&EG(symbol_table), 50, NULL, ZVAL_PTR_DTOR, 0);
```

这里, 你可以看到, 当从符号表删除一个元素时, 比如可能是对unset($foo)的处理; 在HashTable中存储的zval *指针都会被发送给zval_ptr_dtor()(ZVAL_PTR_DTOR展开就是它.).

因为50并不是2的幂, 因此实际初始化的全局符号表是2的下一个幂64.

填充

有4个主要的函数用于插入和更新HashTable的数据:

```
int zend_hash_add(HashTable *ht, char *arKey, uint nKeyLen,
                void **pData, uint nDataSize, void *pDest);

int zend_hash_update(HashTable *ht, char *arKey, uint nKeyLen,
                void *pData, uint nDataSize, void **pDest);
int zend_hash_index_update(HashTable *ht, ulong h,
                void *pData, uint nDataSize, void **pDest);
int zend_hash_next_index_insert(HashTable *ht,
                void *pData, uint nDataSize, void **pDest);
```

这里的前两个函数用于新增关联索引数据, 比如$foo['bar'] = 'baz';对应的C语言代码如下:

```
zend_hash_add(fooHashTbl, "bar", sizeof("bar"), &barZval, sizeof(zval*), NULL);
```

zend_hash_add()和zend_hash_update()唯一的区别是如果key存在, zend_hash_add()将会失败.

接下来的两个函数以类似的方式处理数值索引的HashTable. 这两行之间的区别在于是否指定索引 或者说是否自动赋值为下一个可用索引.

如果需要存储使用zend_hash_next_index_insert()插入的元素的索引值, 可以调用zend_hash_next_free_element()函数获得:

```
ulong nextid = zend_hash_next_free_element(ht);
zend_hash_index_update(ht, nextid, &data, sizeof(data), NULL);
```

对于上面这些插入和更新函数, 如果给pDest传递了值, 则pDest指向的void *数据元素将被填充为指向被拷贝数据的指针. 这个参数和你已经见到过的zend_hash_find()的pData参数是相同的用法(也会有相同的结果).

译注*:* 下面的例子及输出可能对理解*pDest*有帮助

```
/* 拷贝自Zend/zend_hash.c */
void zend_hash_display_string(const HashTable *ht)
{
    Bucket *p;
    uint i;

    if (UNEXPECTED(ht->nNumOfElements == 0)) {
        zend_output_debug_string(0, "The hash is empty");
        return;
    }
    for (i = 0; i < ht->nTableSize; i++) {
        p = ht->arBuckets[i];
        while (p != NULL) {
            zend_output_debug_string(0, "%s[0x%lX] <==> %s", p->arKey, p->h, (char *)p->pData);
            p = p->pNext;
        }
    }

    p = ht->pListTail;
    while (p != NULL) {
        zend_output_debug_string(0, "%s[hash = 0x%lX, pointer = %p] <==> %s[pointer = %p]", p-
>arKey, p->h, p->arKey, (char *)p->pData, p->pData);
        p = p->pListLast;
    }
}
PHP_FUNCTION(sample_ht)
{
    HashTable   *ht0;
    char        *key;
    char        *value;
    void        *pDest;

    key     = emalloc(16);
    value   = emalloc(32);

    ALLOC_HASHTABLE(ht0);
    zend_hash_init(ht0, 50, NULL, NULL, 0);

    strcpy(key, "ABCDEFG");
    strcpy(value, "0123456789");

    printf("key: %p %s\n", key, key);
    printf("value: %p %s\n", value, value);

    zend_hash_add(ht0, key, 8, value, 11, &pDest);

    printf("pDest: %p\n", pDest);
```

```
    zend_hash_display_string(ht0);

    zend_hash_destroy(ht0);
    FREE_HASHTABLE(ht0);

    efree(value);
    efree(key);

    RETURN_NULL();
}
```

译注: 在*sample.c*以及*php_sample.h*中增加对应的*php_sample_functions*条目及声明, 重新编译这个扩展. 执行下面命令:

```
php -d extension=sample.so -r 'sample_ht();'
```

译注: 得到如下输出

```
key: 0x7feef4d17bd8 ABCDEFG
value: 0x7feef4d15aa0 0123456789
pDest: 0x7feef4d17da0
ABCDEFG[0x1AE58CF22D2E61] <==> 0123456789
ABCDEFG[hash = 0x1AE58CF22D2E61, pointer = 0x7feef4d17d38] <==> 0123456789[pointer =
0x7feef4d17da0]
```

找回

因为HashTable有两种不同的方式组织索引, 因此就相应的有两种方法提取数据:

```
int zend_hash_find(HashTable *ht, char *arKey, uint nKeyLength,
                                        void **pData);
int zend_hash_index_find(HashTable *ht, ulong h, void **pData);
```

你可能已经猜到了, 第一种用来维护关联索引的数组, 第二种用于数字索引. 回顾第2章, 当数据被增加到HashTable时, 为它分配一块新的内存并将数据拷贝到其中; 当提取数据的时候, 这个数据指针将被返回. 下面的代码片段向HashTable增加了data1, 接着在程序的末尾提取它, *data2包含了和*data1相同的内容, 虽然它们指向不同的内存地址.

```
void hash_sample(HashTable *ht, sample_data *data1)

{
   sample_data *data2;
   ulong targetID = zend_hash_next_free_element(ht);
   if (zend_hash_index_update(ht, targetID,
           data1, sizeof(sample_data), NULL) == FAILURE) {
       /* 应该不会发生 */
       return;
   }
   if(zend_hash_index_find(ht, targetID, (void **)&data2) == FAILURE) {
       /* 同样不太可能, 因为我们只是增加了一个元素 */
       return;
   }
   /* data1 != data2, however *data1 == *data2 */
}
```

通常, 取回存储的数据和检查它是否存在一样重要; 有两个函数用于检查是否存在:

```
int zend_hash_exists(HashTable *ht, char *arKey, uint nKeyLen);
int zend_hash_index_exists(HashTable *ht, ulong h);
```

这两个函数并不会返回SUCCESS/FAILURE, 而是返回1标识请求的key/index存在, 0标识不存在, 下面代码片段的执行等价于isset($foo):

```
if (zend_hash_exists(EG(active_symbol_table),
                              "foo", sizeof("foo"))) {
    /* $foo is set */
} else {
    /* $foo does not exist */
}
```

快速的填充和取回

```
ulong zend_get_hash_value(char *arKey, uint nKeyLen);
```

在相同的关联key上执行多次操作时, 可以先使用zend_get_hash_value()计算出哈希值. 它的结果可以被传递给一组"快速"的函数, 它们的行为与对应的非快速版本一致, 但是使用预先计算好的哈希值, 而不是每次重新计算.

```
int zend_hash_quick_add(HashTable *ht,
    char *arKey, uint nKeyLen, ulong hashval,
    void *pData, uint nDataSize, void **pDest);
int zend_hash_quick_update(HashTable *ht,
    char *arKey, uint nKeyLen, ulong hashval,
    void *pData, uint nDataSize, void **pDest);
int zend_hash_quick_find(HashTable *ht,
    char *arKey, uint nKeyLen, ulong hashval, void **pData);
int zend_hash_quick_exists(HashTable *ht,
    char *arKey, uint nKeyLen, ulong hashval);
```

奇怪的是没有zend_hash_quick_del(). 下面的代码段从hta(zval *的HashTable)拷贝一个特定的元素到htb, 它演示了"快速"版本的哈希函数使用:

```
void php_sample_hash_copy(HashTable *hta, HashTable *htb,
                    char *arKey, uint nKeyLen TSRMLS_DC)
{
    ulong hashval = zend_get_hash_value(arKey, nKeyLen);
    zval **copyval;

    if (zend_hash_quick_find(hta, arKey, nKeyLen,
                hashval, (void**)&copyval) == FAILURE) {
        /* arKey不存在 */
        return;
    }
    /* zval现在同时被另外一个哈希表持有引用 */
    (*copyval)->refcount++;
    zend_hash_quick_update(htb, arKey, nKeyLen, hashval,
                copyval, sizeof(zval*), NULL);
}
```

译注: 下面的例子是译者对上面例子的修改, 应用在数组上, 对外暴露了用户空间接口.

```
/* php_sample.h中定义的arg info */
#ifdef ZEND_ENGINE_2
ZEND_BEGIN_ARG_INFO(sample_array_copy_arginfo, 0)
    ZEND_ARG_ARRAY_INFO(1, "a", 0)
    ZEND_ARG_ARRAY_INFO(1, "b", 0)
    ZEND_ARG_PASS_INFO(0)
ZEND_END_ARG_INFO()
#else
static unsigned char    sample_array_copy_arginfo[] =
    {3, BYREF_FORCE, BYREF_FORCE, 0};
#endif
PHP_FUNCTION(sample_array_copy);

/* sample.c中在php_sample_functions中增加的对外暴露接口说明 */
PHP_FE(sample_array_copy, sample_array_copy_arginfo)

/* 函数逻辑实现 */
PHP_FUNCTION(sample_array_copy)
{
    zval    *a1, *a2, **z;
    char    *key;
    int     key_len;
    ulong   h;

    if ( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "aas", &a1, &a2, &key, &key_len) ==
FAILURE ) {
```

```
        RETURN_FALSE;
    }

    h   = zend_get_hash_value(key, key_len + 1);

    if ( zend_hash_quick_find(Z_ARRVAL_P(a1), key, key_len + 1, h, (void **)&z) == FAILURE ) {
        RETURN_FALSE;
    }

    Z_SET_REFCOUNT_PP(z, Z_REFCOUNT_PP(z) + 1);
    Z_SET_ISREF_PP(z); /* 这里设置为引用类型, 读者可以注释这一行比较结果, 增强对变量引用的理解. */
    zend_hash_quick_update(Z_ARRVAL_P(a2), key, key_len + 1, h, z, sizeof(zval *), NULL);

    RETURN_TRUE;
}
```

拷贝和合并

前面的任务是从一个HashTable拷贝一个元素到另一个HashTable, 这是很常见的, 并且通常都是批量去做. 为了避免重复的取回和设置值的循环操作, 有3个帮助函数:

```
typedef void (*copy_ctor_func_t)(void *pElement);
void zend_hash_copy(HashTable *target, HashTable *source,
            copy_ctor_func_t pCopyConstructor,
            void *tmp, uint size);
```

source中的每个元素都会被拷贝到target中, 接着通过pCopyConstructor函数处理. 对于用户空间数组变量这样的HashTable, 这里提供了增加引用计数的机会, 因此当zval *从一个HashTable中移除的时候, 它并不会被提前销毁. 如果在目标HashTable中已经存在了相同的元素, 将使用新元素覆盖. 其他已有的没有被覆盖的元素也不会被隐式的移除.

tmp应该是一个指针, 它指向的内存区域将被zend_hash_copy()函数在执行过程中作为临时空间使用. php 4.0.3之后, 这个临时空间不再使用. 如果确认你的扩展不会在4.0.3之前的php中使用, 就将它设置为NULL.

size是每个成员元素所占的字节数. 对于用户空间变量Hash的情况, 它应该是sizeof(zval *).

```
void zend_hash_merge(HashTable *target, HashTable *source,
            copy_ctor_func_t pCopyConstructor,
            void *tmp, uint size, int overwrite);
```

zend_hash_merge()与zend_hash_copy()唯一的不同在于最后的overwrite参数. 当将它设置为非0值时, zend_hash_merge()的行为和zend_hash_copy()一致. 当它设置为0时, 跳过已经存在的元素.

```
typedef zend_bool (*merge_checker_func_t)(HashTable *target_ht,
    void *source_data, zend_hash_key *hash_key, void *pParam);
void zend_hash_merge_ex(HashTable *target, HashTable *source,
            copy_ctor_func_t pCopyConstructor, uint size,
            merge_checker_func_t pMergeSource, void *pParam);
```

这一组函数中的最后一个, 允许使用一个合并检查函数有选择的拷贝. 下面的例子展示了zend_hash_merge_ex()用于仅拷贝源HashTable中关联索引成员的例子:

```
zend_bool associative_only(HashTable *ht, void *pData,
            zend_hash_key *hash_key, void *pParam)
{
    /* True if there's a key, false if there's not */
    return (hash_key->arKey && hash_key->nKeyLength);
}
void merge_associative(HashTable *target, HashTable *source)
{
    zend_hash_merge_ex(target, source, zval_add_ref,
                sizeof(zval*), associative_only, NULL);
}
```

使用Hash Apply迭代

就像用户空间一样, 有多种方式去迭代数据集合. 首先, 最简单的方法就是类似于用户空间的foreach()结构, 使用回调系统. 系统涉及两个部分, 一部分是你要编写的回调函数, 它扮演的角色相当于foreach循环内嵌的代码, 另一部分则是对3个Hash应用API函数的调用.

```
typedef int (*apply_func_t)(void *pDest TSRMLS_DC);
void zend_hash_apply(HashTable *ht,
        apply_func_t apply_func TSRMLS_DC);
```

Hash apply族函数中最简单的格式是通过迭代ht, 将当前迭代到的元素指针作为参数pDest传递, 调用apply_func.

```
typedef int (*apply_func_arg_t)(void *pDest,
                            void *argument TSRMLS_DC);
void zend_hash_apply_with_argument(HashTable *ht,
        apply_func_arg_t apply_func, void *data TSRMLS_DC);
```

下一种Hash apply的格式是与迭代元素一起传递另外一个参数. 这通常用于多目的的Hash apply函数, 它的行为依赖于额外的参数而不同.

回调函数并不关心使用哪个迭代函数, 它只有3种可能的返回值:

常量	含义
ZEND_HASH_APPLY_KEEP	返回这个值将完成当前循环, 并继续迭代HashTable中的下一个值. 这等价于在foreach()控制块中执行continue;
ZEND_HASH_APPLY_STOP	返回这个值将中断迭代, 等价于在foreach()控制块中执行break;
ZEND_HASH_APPLY_REMOVE	类似于ZEND_HASH_APPLY_KEEP, 这个返回值将跳到下一次迭代. 不过, 这个返回值同时会导致从目标HashTable中删除当前元素.

下面是一个简单的用户空间foreach()循环:

```
<?php
foreach($arr as $val) {
    echo "The value is: $val\n";
}
?>
```

它被翻译成对应的C代码如下:

```
int php_sample_print_zval(zval **val TSRMLS_DC)
{
    /* 复制一份zval, 使得原来的结构不被破坏 */
    zval tmpcopy = **val;

    zval_copy_ctor(&tmpcopy);
    /* 重置引用计数并进行类型转换 */
    INIT_PZVAL(&tmpcopy);
    convert_to_string(&tmpcopy);
    /* 输出 */

    php_printf("The value is: ");
    PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
    php_printf("\n");
    /* 释放拷贝 */
```

```
    zval_dtor(&tmpcopy);
    /* 继续下一个 */
    return ZEND_HASH_APPLY_KEEP;
}
```

我们使用下面的函数进行迭代:

```
zend_hash_apply(arrht, php_sample_print_zval TSRMLS_CC);
```

虽然函数的调用只使用一级间访, 但它定义的参数仍然是*zval* **, 这是因为变量在*HashTable*中存储时, 实际上只拷贝了*zval* 的指针, 而*HashTable*自身并没有触及*zval*的内容. 如果还不清楚为什么这样做, 请参考第*2*章.

```
typedef int (*apply_func_args_t)(void *pDest,
        int num_args, va_list args, zend_hash_key *hash_key);
void zend_hash_apply_with_arguments(HashTable *ht,
        apply_func_args_t apply_func, int numargs, ...);
```

为了在循环过程中和值一起接受key, 就必须使用zend_hash_apply()的第三种格式. 例如, 扩展上面的理智, 支持key的输出:

```
<?php
foreach($arr as $key => $val) {
    echo "The value of $key is: $val\n";
}
?>
```

当前的迭代回调无法处理$key的获取. 切换到zend_hash_apply_with_arguments(), 回调函数的原型和实现修改如下:

```
int php_sample_print_zval_and_key(zval **val,
        int num_args, va_list args, zend_hash_key *hash_key)
{
    /* 复制zval以使原来的内容不被破坏 */
    zval tmpcopy = **val;
    /* 输出函数需要tsrm_ls */
    TSRMLS_FETCH();

    zval_copy_ctor(&tmpcopy);
    /* 重置引用计数并进行类型转换 */
    INIT_PZVAL(&tmpcopy);
    convert_to_string(&tmpcopy);
    /* 输出 */
    php_printf("The value of ");
    if (hash_key->nKeyLength) {
        /* 关联类型的key */
        PHPWRITE(hash_key->arKey, hash_key->nKeyLength);
    } else {
        /* 数值key */
        php_printf("%ld", hash_key->h);
    }
    php_printf(" is: ");
    PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
    php_printf("\n");
    /* 释放拷贝 */
    zval_dtor(&tmpcopy);
    /* 继续 */
    return ZEND_HASH_APPLY_KEEP;
}
```

译注*:* 译者使用的*php-5.4.9*中不需要*TSRMLS_FETCH()*一行, 回调原型中已经定义了*TSRMLS_DC.*

使用下面的函数调用进行迭代:

```
zend_hash_apply_with_arguments(arrht,
                    php_sample_print_zval_and_key, 0);
```

这个示例比较特殊, 不需要传递参数*;* 对于从*va_list args*中提取可变参数, 请参考*POSIX*文档的*va_start(), va_arg(), va_end().*

注意用于测试一个*key*是否是关联类型的, 使用的是*nKeyLength*, 而不是*arKey*. 这是因为在*Zend HashTable*的实现中, 可能会在*arKey*中遗留数据. 同时, *nKeyLength*还可以安全的处理空字符串的*key*(比如*$foo[''] = 'bar';*), 因为*nKeyLength*包含了末尾的*NULL*字节.

向前推移的迭代

我们也可以不使用回调进行HashTable的迭代. 此时, 你就需要记得HashTable中一个常常被忽略的概念: 内部指针.

在用户空间, 函数reset(), key(), current(), next(), prev(), each(), end()可以用于访问数组内的元素, 它们依赖于一个不可访问的"当前"位置.

```
<?php
    $arr = array('a'=>1, 'b'=>2, 'c'=>3);
    reset($arr);
    while (list($key, $val) = each($arr)) {
        /* Do something with $key and $val */
    }
    reset($arr);
    $firstkey = key($arr);
    $firstval = current($arr);
    $bval = next($arr);
    $cval = next($arr);
?>
```

这些函数都是对同名的Zend Hash API函数的封装.

```
/* reset() */
void zend_hash_internal_pointer_reset(HashTable *ht);
/* key() */
int zend_hash_get_current_key(HashTable *ht,
        char **strIdx, unit *strIdxLen,
        ulong *numIdx, zend_bool duplicate);
/* current() */
int zend_hash_get_current_data(HashTable *ht, void **pData);
/* next()/each() */
int zend_hash_move_forward(HashTable *ht);
/* prev() */
int zend_hash_move_backwards(HashTable *ht);
/* end() */
void zend_hash_internal_pointer_end(HashTable *ht);
/* Other... */
int zend_hash_get_current_key_type(HashTable *ht);
int zend_hash_has_more_elements(HashTable *ht);
```

*next(), prev(), end()*三个用户空间语句实际上映射到的是内部的向前/向后移动, 接着调用*zend_hash_get_current_data(). each()*执行和*next()*相同的步骤, 但是同时调用*zend_hash_get_current_key()*并返回.

通过向前移动的方式实现的迭代实际上和foreach()循环更加相似, 下面是对前面print_zval_and_key示例的再次实现:

```
void php_sample_print_var_hash(HashTable *arrht)
{

    for(zend_hash_internal_pointer_reset(arrht);
    zend_hash_has_more_elements(arrht) == SUCCESS;
    zend_hash_move_forward(arrht)) {
        char *key;
        uint keylen;
        ulong idx;
        int type;
        zval **ppzval, tmpcopy;

        type = zend_hash_get_current_key_ex(arrht, &key, &keylen,
                                            &idx, 0, NULL);
        if (zend_hash_get_current_data(arrht, (void**)&ppzval) == FAILURE) {
```

```
            /* 应该永远不会失败, 因为key是已知存在的. */
            continue;
        }
        /* 复制zval以使原来的内容不被破坏 */
        tmpcopy = **ppzval;
        zval_copy_ctor(&tmpcopy);
        /* 重置引用计数, 并进行类型转换 */
        INIT_PZVAL(&tmpcopy);
        convert_to_string(&tmpcopy);
        /* 输出 */
        php_printf("The value of ");
        if (type == HASH_KEY_IS_STRING) {
            /* 关联类型, 输出字符串key. */
            /* 译注: 这里传递给PHPWRITE的keylen应该要减1才合适, 因为HashTable中的key长度包含
             * 末尾的NULL字节, 而正常的php字符串长度不包含这个NULL字节, 不过这里打印通常不会有
             * 问题, 因为NULL字节一般打印出是空的 */
            PHPWRITE(key, keylen);
        } else {
            /* 数值key */
            php_printf("%ld", idx);
        }
        php_printf(" is: ");
        PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
        php_printf("\n");
        /* 释放拷贝 */
        zval_dtor(&tmpcopy);
    }
}
```

这个代码片段对你来说应该是比较熟悉的了. 没有接触过的是zend_hash_get_current_key()的返回值. 调用时, 这个函数可能返回下表中3个返回值之一:

常量	含义
HASH_KEY_IS_STRING	当前元素是关联索引的; 因此, 指向元素key名字的指针将会被设置到strIdx中, 它的长度被设置到stdIdxLen中. 如果指定了duplicate标记, key的值将在设置到strIdx之前使用estrndup()复制一份. 这样做, 调用方就需要显式的释放这个复制出来的字符串.
HASH_KEY_IS_LONG	当前元素是数值索引的, 索引的数值将被设置到numIdx中
HASH_KEY_NON_EXISTANT	内部指针到达了HashTable内容的末尾. 此刻已经没有其他key或数据可用了.

保留内部指针

在迭代HashTable时, 尤其是当它包含用户空间变量时, 少数情况下会碰到循环引用或者说自交的循环. 如果一个迭代上下文的循环开始后, HashTable的内部指针被调整, 接着内部启动了对同一个HashTable的迭代循环, 它就会擦掉原有的当前内部指针位置, 内部的迭代将导致外部的迭代被异常终止.

对于使用zend_hash_apply样式的实现以及自定义的向前移动的用法, 均可以通过外部的HashPosition变量的方式来解决这个问题.

前面列出的zend_hash_*()函数均有对应的zend_hash_*_ex()实现, 它们可以接受一个HashPosition类型的参数. 因为HashPosition变量很少在短生命周期的循环之外使用, 因此将它定义为直接变量就足够了. 接着可以取地址进行使用, 如下示例:

```
void php_sample_print_var_hash(HashTable *arrht)
{
    HashPosition pos;
    for(zend_hash_internal_pointer_reset_ex(arrht, &pos);
    zend_hash_has_more_elements_ex(arrht, &pos) == SUCCESS;
    zend_hash_move_forward_ex(arrht, &pos)) {
        char *key;
        uint keylen;
        ulong idx;
        int type;

        zval **ppzval, tmpcopy;

        type = zend_hash_get_current_key_ex(arrht,
                                &key, &keylen,
                                &idx, 0, &pos);
        if (zend_hash_get_current_data_ex(arrht,
                    (void**)&ppzval, &pos) == FAILURE) {
            /* 应该永远不会失败, 因为key已知是存在的 */
            continue;
        }
        /* 复制zval防止原来的内容被破坏 */
        tmpcopy = **ppzval;
        zval_copy_ctor(&tmpcopy);
        /* 重置引用计数并进行类型转换 */
        INIT_PZVAL(&tmpcopy);
        convert_to_string(&tmpcopy);
        /* 输出 */
        php_printf("The value of ");
        if (type == HASH_KEY_IS_STRING) {
            /* 关联方式的字符串key */
            PHPWRITE(key, keylen);
        } else {
            /* 数值key */
            php_printf("%ld", idx);
        }
        php_printf(" is: ");
        PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
        php_printf("\n");
        /* 释放拷贝 */
        zval_dtor(&tmpcopy);
    }
}
```

通过这些轻微的修改, HashTable真正的内部指针将被保留, 它就可以保持为刚刚进入函数时的状态不变. 在用户空间变量的HashTable(数组)上工作时, 这些额外的步骤很可能就是决定脚本执行结果是否与预期一致的关键点.

析构

你需要关注的析构函数只有4个. 前两个用于从一个HashTable中移除单个元素:

```
int zend_hash_del(HashTable *ht, char *arKey, uint nKeyLen);
int zend_hash_index_del(HashTable *ht, ulong h);
```

你应该可以猜到, 这里体现了HashTable独立的索引设计, 它为关联和数值方式的索引元素分别提供了删除函数. 两者均应该返回SUCCESS或FAILURE.

回顾前面, 当一个元素从HashTable中移除时, HashTable的析构函数将被调用, 传递的参数是指向元素的指针.

```
void zend_hash_clean(HashTable *ht);
```

要完全清空HashTable时, 最快的方式是调用zend_hash_clean(), 它将迭代所有的元素调用zend_hash_del():

```
void zend_hash_destroy(HashTable *ht);
```

通常, 清理HashTable时, 你会希望将它整个都清理掉. 调用zend_hash_destroy()将会执行zend_hash_clean()的所有步骤, 同时还会释放zend_hash_init()分配的其他结构.

下面的代码演示了一个完整的HashTable生命周期:

```
int sample_strvec_handler(int argc, char **argv TSRMLS_DC)
{
    HashTable *ht;
    /* 分配一块内存用于HashTable结构 */
    ALLOC_HASHTABLE(ht);
    /* 初始化HashTable的内部状态 */
    if (zend_hash_init(ht, argc, NULL,
                       ZVAL_PTR_DTOR, 0) == FAILURE) {
        FREE_HASHTABLE(ht);
        return FAILURE;
    }
    /* 将传入的字符串数组, 顺次以字符串的zval *放入到HashTable中 */
    while (argc) {
        zval *value;
        MAKE_STD_ZVAL(value);
        ZVAL_STRING(value, argv[argc], 1);
        argv++;
        if (zend_hash_next_index_insert(ht, (void**)&value,
                            sizeof(zval*)) == FAILURE) {
            /* 添加失败则静默的跳过 */
            zval_ptr_dtor(&value);
        }
    }
    /* 执行一些其他工作(业务) */
    process_hashtable(ht);
    /* 销毁HashTable, 释放所有需要释放的zval */
    zend_hash_destroy(ht);

    /* 释放HashTable自身 */
    FREE_HASHTABLE(ht);
    return SUCCESS;
}
```

排序, 比较

在Zend Hash API中还存在其他一些回调. 第一个是用来处理同一个HashTable中两个元素或者不同HashTable相同位置元素的比较的:

```
typedef int (*compare_func_t)(void *a, void *b TSRMLS_DC);
```

就像用户空间的usort()回调一样, 这个函数期望你使用自己的逻辑比较两个值a和b, 返回-1表示a小于b, 返回1表示b小于a, 返回0表示两者相等.

```
int zend_hash_minmax(HashTable *ht, compare_func_t compar,
                     int flag, void **pData TSRMLS_DC);
```

使用这个回调的最简单的API函数是zend_hash_minmax(), 顾名思义, 它将基于多次对比较回调的调用, 最终返回HashTable的最大值/最小值元素. flag为0时返回最小值, flag非0时返回最大值.

下面的例子中, 对已注册的用户空间函数以函数名排序, 并返回(函数名)最小和最大的函数(大小写不敏感):

```
int fname_compare(zend_function *a, zend_function *b TSRMLS_DC)
{
    return strcasecmp(a->common.function_name, b->common.function_name);
}
void php_sample_funcname_sort(TSRMLS_D)
{
```

```
    zend_function *fe;
    if (zend_hash_minmax(EG(function_table), fname_compare,
                0, (void **)&fe) == SUCCESS) {
        php_printf("Min function: %s\n", fe->common.function_name);
    }
    if (zend_hash_minmax(EG(function_table), fname_compare,
                1, (void **)&fe) == SUCCESS) {
        php_printf("Max function: %s\n", fe->common.function_name);
    }
}
```

译注: 原书中的示例在译者的环境*(php-5.4.9)*中不能运行, 经过跟踪检查, 发现*zend_hash_minmax*传递给*fname_compare*的两个参数类型是*Bucket ***, 而非这里的*zend_function **, 为了避免读者疑惑, 下面给出译者修改后的示例供参考.

```
static int sample_fname_compare(Bucket **p1, Bucket **p2 TSRMLS_DC) {
    zend_function   *zf1, *zf2;    zf1 = (zend_function *)(*p1)->pData;
    zf2 = (zend_function *)(*p2)->pData;
    return strcasecmp(zf1->common.function_name, zf2->common.function_name);
}
PHP_FUNCTION(sample_funcname_sort)
{
    zend_function   *zf;

    if ( zend_hash_minmax(EG(function_table), (compare_func_t)sample_fname_compare, 0, (void
**)&zf TSRMLS_CC) == SUCCESS )
        php_printf("Min function: %s\n", zf->common.function_name);    if
( zend_hash_minmax(EG(function_table), (compare_func_t)sample_fname_compare, 1, (void **)&zf
TSRMLS_CC) == SUCCESS )
        php_printf("Max function: %s\n", zf->common.function_name);

    RETURN_TRUE;
}
```

哈希比较函数还会用于zend_hash_compare()中, 它会评估两个HashTable中的每个元素进行比较. 如果hta大于htb, 返回1, 如果htb大于hta, 返回-1, 如果两者相等, 返回0.

```
int zend_hash_compare(HashTable *hta, HashTable *htb,
        compare_func_t compar, zend_bool ordered TSRMLS_DC);
```

这个函数首先会比较两个HashTable的元素个数. 如果其中一个元素个数多于另外一个, 则直接认为它比另外一个大, 快速返回.

接下来, 循环遍历hta. 如果设置了ordered标记, 它将hta的第一个元素和htb的第一个元素的key长度进行比较, 接着使用memcmp()二进制安全的比较key内容. 如果key相等, 则使用提供的compar回调函数比较两个元素的值.

如果没有设置ordered标记, 则遍历hta得到一个元素后, 从htb中查找key/index相等的元素, 如果存在, 对它们的值调用传入的compar回调函数, 否则, 则认为hta比htb大, 直接返回1.

如果上面的处理结束后, hta和htb一致都被认为是相等的, 则从hta中遍历下一个元素重复上面过程, 直到找到不同, 或者所有的元素耗尽, 此时认为它们相等返回0.

这一族的回调函数中第二个是排序函数:

```
typedef void (*sort_func_t)(void **Buckets, size_t numBuckets,
            size_t sizBucket, compare_func_t comp TSRMLS_DC);
```

这个回调将被触发一次, 它以向量方式接受HashTable中所有Bucket(元素)的指针. 这些Bucket可以在向量内部按照排序函数自己的逻辑(与是否使用比较回调无关)进行交换. 实际上, sizBucket总是等于sizeof(Bucket *)

除非你计划实现自己的冒泡或其他排序算法, 否则不需要自己实现排序函数. php内核中已经有一个预定义的排序函数: zend_qsort, 它可以作为zend_hash_sort()的回调函数, 这样, 你就只需要实现比较函数.

```
int zend_hash_sort(HashTable *ht, sort_func_t sort_func,
        compare_func_t compare_func, int renumber TSRMLS_DC);
```

zend_hash_sort()的最后一个参数被设置后, 将会导致在排序后, 原来的关联key以及数值下表都被按照排序结果重置为数值索引. 用户空间的sort()实现就以下面的方式使用了zend_hash_sort():

```
zend_hash_sort(target_hash, zend_qsort,
                        array_data_compare, 1 TSRMLS_CC);
```

不过, array_data_compare只是一个简单的compare_func_t实现, 它只是依据HashTable中zval *的值进行排序.

zval *数组API

你在开发php扩展时, 95%以上的HashTable引用都是用于存储和检索用户空间变量的. 反过来说, 你的多数HashTable自身都将被包装在zval中.

简单的数组创建

为了辅助这些常见的HashTable的创建和操作, PHP API暴露了一些简单的宏和辅助函数, 我们从array_init(zval *arrval)开始看. 这个函数分配了一个HashTable, 以适用于用户空间变量哈希的参数调用zend_hash_init(), 并将新创建的结构设置到zval *中.

这里不需要特殊的析构函数, 因为在zval最后一个refcount失去后, 通过调用zval_dtor()/zval_ptr_dtor(), 引擎会自动的调用zend_hash_destroy()和FREE_HASHTABLE().

联合array_init()方法和第6章"返回值"中已经学习的从函数返回值的技术:

```
PHP_FUNCTION(sample_array)
{
    array_init(return_value);
}
```

因为return_value是一个预分配的zval *, 因此不需要在它上面做其他工作. 并且由于它唯一的引用就是你的函数返回, 因此不要担心它的清理.

简单的数组构造

和所有的HashTable一样, 你需要迭代增加元素来构造数组. 由于用户空间变量的特殊性, 你需要回到你已经知道的C语言中的基础数据类型. 有3种格式的函数: add_assoc_*(), add_index_*(), add_next_index_*(), 对于已知的ZVAL_*(), RETVAL_*(), RETURN_*()宏所支持的每种数据类型, 都有对应的这3种格式的函数. 例如:

```
add_assoc_long(zval *arrval, char *key, long lval);
add_index_long(zval *arrval, ulong idx, long lval);
add_next_index_long(zval *arrval, long lval);
```

每种情况中, 数组zval *都是第一个参数, 接着是关联key名或数值下标, 或者对于next_index变种来说, 两者都不需要. 最后是数据元素自身, 最终它将被包装为一个新分配的zval *, 并使用zend_hash_update(), zend_hash_index_update(), zend_hash_next_index_insert()增加到数组中.

add_assoc_*()函数变种以及它们的函数原型如下. 其他两种格式则将assoc替换为index或next_index, 并对应调整key/index参数即可.

```
add_assoc_null(zval *aval, char *key);
add_assoc_bool(zval *aval, char *key, zend_bool bval);
add_assoc_long(zval *aval, char *key, long lval);
add_assoc_double(zval *aval, char *key, double dval);
add_assoc_string(zval *aval, char *key, char *strval, int dup);
add_assoc_stringl(zval *aval, char *key,
                    char *strval, uint strlen, int dup);
add_assoc_zval(zval *aval, char *key, zval *value);
```

这些函数的最后一个版本允许你自己准备一个任意类型(包括资源, 对象, 数组)的zval, 将它增加到数组中. 现在尝试在你的sample_array()函数中做一些额外的工作.

```
PHP_FUNCTION(sample_array)
{
    zval *subarray;

    array_init(return_value);
    /* 增加一些标量值 */
    add_assoc_long(return_value, "life", 42);
    add_index_bool(return_value, 123, 1);
    add_next_index_double(return_value, 3.1415926535);
    /* 增加一个静态字符串, 由php去复制 */
    add_next_index_string(return_value, "Foo", 1);
    /* 手动复制的字符串 */
    add_next_index_string(return_value, estrdup("Bar"), 0);

    /* 创建一个子数组 */
    MAKE_STD_ZVAL(subarray);
    array_init(subarray);
    /* 增加一些数值 */
    add_next_index_long(subarray, 1);
    add_next_index_long(subarray, 20);
    add_next_index_long(subarray, 300);
    /* 将子数组放入到父数组中 */
    add_index_zval(return_value, 444, subarray);
}
```

如果在这个函数的返回值上调用var_dump()将得到下面输出:

```
$ php -r 'var_dump(sample_array());'
array(6) {
  ["life"]=>
  int(42)
  [123]=>
  bool(true)
  [124]=>
  float(3.1415926535)
  [125]=>
  string(3) "Foo"
  [126]=>
  string(3) "Bar"
  [444]=>
  array(3) {
    [0]=>
    int(1)
    [1]=>
    int(20)
    [2]=>
    int(300)
  }
}
```

这些add_*()函数还可以用于简单对象的内部公共属性. 在第10章"php 4对象"中我们可以看到它们.

小结

你已经花费了一些时间学习了很长的一章, 本章介绍了Zend引擎和php内核中仅次于zval *的通用数据结构. 本章还比较了不同的数据存储机制, 并介绍了很多未来将多次使用的API.

现在你已经有了足够的积累, 可以实现一些相当一部分标准扩展了. 后面的几章将完成剩余的zval数据类型(资源和对象)的学习.

资源数据类型

迄今为止, 你都是工作在非常基础的用户空间数据类型上, 字符串, 数值, TRUE/FALSE等值. 即便上一章你已经开始接触数组了, 但也只是收集这些基础数据类型的数组.

复杂的结构体

现实世界中, 你通常需要在更加复杂的数据集合下工作, 通常涉及到晦涩的结构体指针. 一个常见的晦涩的结构体指针示例就是stdio的文件描述符, 即便是在C语言中也只是一个指针.

```
#include <stdio.h>
int main(void)
{
    FILE *fd;
    fd = fopen("/home/jdoe/.plan", "r");
    fclose(fd);
    return 0;
}
```

stdio的文件描述符和其他多数文件描述符一致, 都像是一个书签. 你扩展的调用应用仅需要在feof(), fread(), fwrite(), fclose()这样的实现函数调用时传递这个值. 有时, 这个书签必须是用户空间代码可访问的; 因此, 就需要在标准的php变量或者说zval *中有表示它的方法.

这里就需要一种新的数据类型. RESOURCE数据类型在zval *中存储一个简单的整型值, 使用作为已注册资源的索引用来查找. 资源条目包含了资源索引所表示的内部数据类型, 以及存储资源数据的指针等信息.

定义资源类型

为了使注册的资源条目所包含的资源信息更加明确, 需要定义资源的类型. 首先在你的sample.c中已有的函数实现下增加下面的代码片段

```
static int le_sample_descriptor;
PHP_MINIT_FUNCTION(sample)
{
    le_sample_descriptor = zend_register_list_destructors_ex(
                NULL, NULL, PHP_SAMPLE_DESCRIPTOR_RES_NAME,
                module_number);
    return SUCCESS;
}
```

接下来, 滚动到你的代码文件末尾, 修改sample_module_entry结构体, 将NULL, /* MINIT */一行替换为下面的内容. 就像你给这个结构中增加函数列表结构时一样, 你需要确认在这一行末尾保留一个逗号.

```
PHP_MINIT(sample), /* MINIT */
```

最后, 你需要在php_sample.h中定义PHP_SAMPLE_DESCRIPTOR_RES_NAME, 将下面的代码放到你的其他常量定义下面:

```
#define PHP_SAMPLE_DESCRIPTOR_RES_NAME "File Descriptor"
```

PHP_MINIT_FUNCTION()代表第1章"PHP生命周期"中介绍的4个特殊的启动和终止操作中的第一个, 关于生命周期, 在第12章"启动, 终止以及之间的几个关键点"和第13章"INI设置"中还将深入讨论.

这里需要知道的非常重要的一点是MINIT函数在你的扩展第一次加载时执行一次, 它会在所有请求到达之前被执行. 这里我们利用这个机会注册了析构函数, 不过它们是NULL值, 不过在通过一个唯一整型ID足以知道一个资源类型时, 你很快就会修改它.

注册资源

现在引擎已经知道了你要存储一些资源数据, 是时候给用户空间的代码一种方式去产生实际的资源了. 要做到这一点, 需要如下重新实现fopen()命令:

```
PHP_FUNCTION(sample_fopen)
{
    FILE *fp;
    char *filename, *mode;
    int filename_len, mode_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",
                        &filename, &filename_len,
                        &mode, &mode_len) == FAILURE) {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Invalid filename or mode length");
        RETURN_FALSE;
    }
    fp = fopen(filename, mode);
    if (!fp) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Unable to open %s using mode %s",
                filename, mode);
        RETURN_FALSE;
    }
    ZEND_REGISTER_RESOURCE(return_value, fp,
                               le_sample_descriptor);
}
```

为了让编译器知道什么是*FILE **, 你需要包含*stdio.h*. 这可以放在*sample.c*中, 但是为了本章后面部分做准备, 我还是要求你放到*php_sample.h*中.

如果你对前面的章节付出了努力, 最后一行前面的所有内容都应该可以读懂. 这一行代码执行的任务是将fp指针存储到资源的索引中, 将它和MINIT中定义的类型关联起来, 并存储一个可用于查找的key到return_value中.

如果需要存储多于一个指针的值, 或者存储一个直接量, 则必须新分配一段内存用来存储数据, 接着将指向这段内存的指针注册为资源.

译注*:*

1. 资源数据类型的注册实际上是在*list_destructors(Zend/zend_list.c*中定义的静态全局变量*HashTable)*中插入一个新构建的*zend_rsrc_list_dtors_entry*结构体, 这个结构体描述了这个资源类型的信息.

2. 资源数据的注册*(ZEND_REGISTER_RESOURCE)*实际上是在*EG(regular_list)*中使用*zend_hash_next_free_element()*得到下一个数值下标, 作为资源的*ID*, 并将传入的资源指针*(*封装为*zend_rsrc_list_entry*结构体*)*存储到*EG(regular_list)*中这个下标对应的元素中.

*3. EG(regular_list)*的初始化是在请求初始化阶段完成的, 通过跟踪代码, 可以看到其函数调用流程如下*: php_request_startup(main/main.c) --> zend_active(Zend/zend.c) --> init_compiler(Zend/zend_compile.c) --> zend_init_rsrc_list(Zend/zend_list.c).* 通过观察*zend_init_rsrc_list()*函数可以看出*EG(regular_list)*的析构函数是*list_entry_destructor(Zend/zend_list.c).* 而*list_entry_destructor()*的逻辑是从*list_destructors(*上面第一步所述的静态全局变量*)*中查找要释放的资源对象类型的信息, 接着按照注册资源类型时所指定的析构器进行析构.

4. 按照上面几点, 可以很容易理清本章前面所述内容. 首先注册一个资源类型, 这个资源类型中包含了诸如所属模块编号, 析构器句柄这样的信息. 接着, 在创建具体的资源对象时, 将资源对象和资源类型做了一个关联.

释放资源

现在你已经有办法附加内部数据块到用户空间. 因为大多数你附加到用户空间的资源变量都需要在某个时刻去清理(这里是调用fclose()), 因此你可能需要一个匹配的sample_fclose()函数接受资源变量, 处理它的销毁并从注册的资源列表(EG(regular_list))中删除它.

如果变量被简单的unset()会怎么样呢? 没有到原来的FILE *指针的引用, 就没有办法去fclose()它, 它就会保持打开状态直到php进程终止. 因为单进程将服务多个请求, 这可能需要很长时间.

答案就是你传递给zend_register_list_destructors_ex的NULL指针. 顾名思义, 你注册的是析构函数. 第一个指针指向的函数在一个请求生命周期内注册资源的最后一个引用被破坏时调用. 实际上就是我们所说的在已存储的资源变量上调用unset().

传递给zend_register_list_destructors_ex的第二个指针指向另外一个回调函数, 它用于持久化资源, 当一个进程或线程终止时被调用. 本章后面将会介绍持久化资源.

现在我们来定义第一个析构函数. 将下面的代码放到你的PHP_MINIT_FUNCTION上面:

```
static void php_sample_descriptor_dtor(
                    zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    FILE *fp = (FILE*)rsrc->ptr;
    fclose(fp);
}
```

下一步是将zend_register_list_destructors_ex调用中的第一个NULL替换为php_sample_destriptor_dtor:

```
le_sample_descriptor = zend_register_list_destructors_ex(
        php_sample_descriptor_dtor, NULL,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME, module_number);
```

现在, 当变量被赋值为sample_fopen()注册的资源值时, 当变量通过unset()或到达函数结束隐式的结束其生命周期时, 将自动的调用fclose()释放FILE *指针. 不再需要sample_fclose()的实现了.

```
<?php
  $fp = sample_fopen("/home/jdoe/notes.txt", "r");
  unset($fp);
?>
```

当unset($fp)被调用时, 引擎会自动的调用php_sample_descriptor_dtor去处理资源的清理.

资源解码

创建资源仅仅是第一步, 因为书签的作用只是让你可以回到原来的那一页. 这里是另外一个函数:

```
PHP_FUNCTION(sample_fwrite)
{
    FILE *fp;
    zval *file_resource;
    char *data;
    int data_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",
            &file_resource, &data, &data_len) == FAILURE ) {
        RETURN_NULL();
    }
    /* 使用zval *验证资源类型, 并从注册资源表中取回它的指针 */
    ZEND_FETCH_RESOURCE(fp, FILE*, &file_resource, -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor);
    /* 写数据并返回实际写入到文件的字节数 */
```

```
    RETURN_LONG(fwrite(data, 1, data_len, fp));
}
```

在zend_parse_parameters()中使用"r"格式描述符相对比较新, 不过, 在你阅读完第7章"接受参数"后应该可以理解. 这里真正新鲜的是ZEND_FETCH_RESOURCE()的使用.

展开ZEND_FETCH_RESOURCE()宏, 代码如下:

```
#define ZEND_FETCH_RESOURCE(rsrc, rsrc_type, passed_id,
           default_id, resource_type_name, resource_type)
    rsrc = (rsrc_type) zend_fetch_resource(passed_id TSRMLS_CC,
                    default_id, resource_type_name, NULL,
                    1, resource_type);
    ZEND_VERIFY_RESOURCE(rsrc);
```

套用当前示例则如下:

```
fp = (FILE*) zend_fetch_resource(&file_descriptor TSRMLS_CC, -1,
                    PHP_SAMPLE_DESCRIPTOR_RES_NAME, NULL,
                    1, le_sample_descriptor);
if (!fp) {
    RETURN_FALSE;
}
```

就像上一章学习的zend_hash_find()函数一样, zend_fetch_resource()实际上是使用索引在一个HashTable集合中找出之前存储的数据. 与zend_hash_find()的不同在于这个函数执行了额外的数据完整性检查, 比如确保资源表中的条目是正确的资源类型.

现在, 你请求的zend_fetch_resource()是和在le_sample_descriptor中存储的资源类型匹配的. 如果提供的资源ID不存在, 或者是不正确的类型, zend_fetch_resource()将返回NULL, 并自动的产生一个错误.

通过在ZEND_FETCH_RESOURCE()宏内部包含ZEND_VERIFY_RESOURCE()宏, 函数实现可以自动的返回, 使得函数自身的代码可以聚焦条件正确时对资源数据值的处理上. 现在你的函数得到了原来的FILE *指针, 直接和普通程序一样调用内部的fwrite()函数.

为了避免*zend_fetch_resource()*在失败时产生错误, 可以将*resource_type_name*参数传递为*NULL*. 由于无法产生有意义的错误消息, *zend_fetch_resoure()*将会静默的失败.

还有一种将资源变量ID翻译成所存储的资源指针的方法是使用zend_list_find()函数:

```
PHP_FUNCTION(sample_fwrite)
{
    FILE *fp;
    zval *file_resource;
    char *data;
    int data_len, rsrc_type;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",
            &file_resource, &data, &data_len) == FAILURE ) {
        RETURN_NULL();
    }
    fp = (FILE*)zend_list_find(Z_RESVAL_P(file_resource),
                                    &rsrc_type);
    if (!fp || rsrc_type != le_sample_descriptor) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                        "Invalid resource provided");
        RETURN_FALSE;
    }
    RETURN_LONG(fwrite(data, 1, data_len, fp));
}
```

虽然对于一般的C语言背景程序员, 这种方式更加容易理解, 但它相比ZEND_FETCH_RESOURCE()更加冗长. 你可以根据自己的编码风格选择合适的方法, 但是还是希望你可以去看看php内核中的其他扩展, 更多的还是使用了ZEND_FETCH_RESOURCE()宏.

强制析构

前面你看到了使用unset()让一个变量结束其生命周期可以触发资源的析构, 并导致其下的资源被以你注册的析构函数清理. 现在想想一个资源变量被拷贝到了其他变量中:

```
<?php
  $fp = sample_fopen("/home/jdoe/world_domination.log", "a");
  $evil_log = $fp;
  unset($fp);
?>
```

此时, $fp并不是注册资源的唯一引用, 因此该资源并没有结束它的生命周期, 不会被释放. 这表示$evil_log仍然可以写. 当你真正的需要一个资源不再被使用时, 为了避免四处找寻引用它的代码, 就需要一个sample_fclose()实现:

```
PHP_FUNCTION(sample_fclose)
{
    FILE *fp;
    zval *file_resource;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r",
                        &file_resource) == FAILURE ) {
        RETURN_NULL();
    }
    /* 虽然并不需要真的取回FILE *资源, 但执行这个宏可以去检查我们关闭资源类型是否正确 */
    ZEND_FETCH_RESOURCE(fp, FILE*, &file_resource, -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor);
    /* 强制资源进入自解模式 */
    zend_hash_index_del(&EG(regular_list),
                    Z_RESVAL_P(file_resource));
    RETURN_TRUE;
}
```

这个删除方法更有力的说明了资源变量是注册在一个全局的HashTable中的. 使用资源ID作为索引在regular_list中查找并删除这个资源条目是很简单的. 虽然其他的HashTable直接操作函数, 比如zend_hash_index_find()/zend_hash_next_index_insert()可以用来替代FETCH和REGISTER宏, 但是这种做法是不鼓励的, 因为这可能使得Zend API在发生变更时影响已有的扩展.

和用户空间的HashTable变量(数组)一样, EG(regular_list)这个HashTable有一个自动的dtor函数, 每当一条记录被移除或覆盖时都会调用该函数. 这个方法会检查你的资源类型, 调用在MINIT中调用zend_register_list_destructors_ex()提供的析构函数.

在php内核和Zend引擎中, 你可以看到很多地方在现在这种情况时使用的是zend_list_delete(), 而不是zend_hash_index_del(). 这是因为zend_list_delete()中有对引用计数的维护, 这一点你将在本章后面看到.

持久化资源

对于存储资源变量的复杂数据类型通常需要可观的内存分配, CPU时间, 或网络通信去初始化. 对于每个调用都需要重新建立的资源类型, 比如数据库连接, 让它们可以在多个请求之间共享是很有用的.

内存分配

通过前面章节的学习我们知道, emalloc()以及它的同族函数是在php中分配内存时的首选, 因为它们能够做到系统的malloc()函数所不能的垃圾回收, 使得在脚本意外终止时通过它们分配的内存可以被回收. 如果一个持久化的资源要跨请求逗留, 这样的垃圾回收很显然不是一件好事.

想象一下, 现在还需要和FILE *指针一起保存打开文件的文件名. 现在, 你就需要在php_sample.h中创建一个自定义结构体来保存这个联合信息:

```
typedef struct _php_sample_descriptor_data {
    char *filename;
    FILE *fp;
} php_sample_descriptor_data;
```

sample.c中所有你处理文件资源的代码都需要修改:

```
static void php_sample_descriptor_dtor(
                  zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    php_sample_descriptor_data *fdata =
                (php_sample_descriptor_data*)rsrc->ptr;
    fclose(fdata->fp);
    efree(fdata->filename);
    efree(fdata);
}
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode;
    int filename_len, mode_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",
                        &filename, &filename_len,
                        &mode, &mode_len) == FAILURE) {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Invalid filename or mode length");
        RETURN_FALSE;
    }
    fp = fopen(filename, mode);
    if (!fp) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Unable to open %s using mode %s",
                filename, mode);
        RETURN_FALSE;
    }
    fdata = emalloc(sizeof(php_sample_descriptor_data));
    fdata->fp = fp;
    fdata->filename = estrndup(filename, filename_len);
    ZEND_REGISTER_RESOURCE(return_value, fdata,
                               le_sample_descriptor);
}
PHP_FUNCTION(sample_fwrite)
{
    php_sample_descriptor_data *fdata;
    zval *file_resource;
    char *data;
    int data_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",
            &file_resource, &data, &data_len) == FAILURE ) {
        RETURN_NULL();
    }
    ZEND_FETCH_RESOURCE(fdata, php_sample_descriptor_data*,
        &file_resource, -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor);
    RETURN_LONG(fwrite(data, 1, data_len, fdata->fp));
}
```

从技术角度来说, *sample_fclose()*可以不用修改, 因为它并不会真的直接处理资源数据. 如果你有信心, 可以自己去更新它.

迄今为止, 一切都是完美的, 因为你仍然只是注册了一个非持久化的描述符资源. 此时, 可以增加一个新的函数去获取已经打开的资源的文件名.

```
PHP_FUNCTION(sample_fname)
```

```
{
    php_sample_descriptor_data *fdata;
    zval *file_resource;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r",
            &file_resource) == FAILURE ) {
        RETURN_NULL();
    }
    ZEND_FETCH_RESOURCE(fdata, php_sample_descriptor_data*,
        &file_resource, -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor);
    RETURN_STRING(fdata->filename, 1);
}
```

然而, 在你开始注册持久化版本的描述符资源时, 问题很快就会显现.

延后析构

你已经能够看到了, 非持久化资源一旦所有持有该资源ID引用的变量都被unset()或结束其生命周期, 它们都会从EG(regular_list)(它是包含所有每个请求注册的资源的HashTable)中被移除.

本章后面你将看到的持久化资源, 也存储在一个HashTable中: EG(persistent_list). 它跟EG(regular_list)有所不同, 使用的索引是关联形式的, 元素不会在请求结束后自动的从HashTable中移除. EG(persistent_list)中的条目只有通过手动调用zend_hash_del()或在线程/进程完全终止(通常是在webserver停止时)时才会被移除.

与EG(regular_list)类似, EG(persistent_list)也有自己的dtor函数. 类似于regular_list, 这个函数也是使用资源类型查找对应的析构函数并调用. 但这里它调用的是调用zend_register_list_destructors_ex()注册资源类型时提供的第二个参数.

实际上, 持久化和非持久化资源注册为两种完全分开的类型是为了避免非持久化析构代码在本应为持久化的资源上再调用一次. 具体依赖于你的实现, 你可以选择在同一个类型中组合非持久化和持久化两种析构函数. 现在, 在sample.c中增加另外一个静态的int变量用于新的持久化资源:

```
static int le_sample_descriptor_persist;
```

接着扩充你的MINIT函数, 增加一个资源注册, 使用新的用于持久化分配结构的dtor函数:

```
static void php_sample_descriptor_dtor_persistent(
                    zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    php_sample_descriptor_data *fdata =
                (php_sample_descriptor_data*)rsrc->ptr;
    fclose(fdata->fp);
    pefree(fdata->filename, 1);
    pefree(fdata, 1);
}
PHP_MINIT_FUNCTION(sample)
{
    le_sample_descriptor =     zend_register_list_destructors_ex(
            php_sample_descriptor_dtor, NULL,
            PHP_SAMPLE_DESCRIPTOR_RES_NAME, module_number);
    le_sample_descriptor_persist =
                        zend_register_list_destructors_ex(
            NULL, php_sample_descriptor_dtor_persistent,
            PHP_SAMPLE_DESCRIPTOR_RES_NAME, module_number);
    return SUCCESS;
}
```

通过给这两个资源类型相同的名字, 它们的不同对于终端用户就是透明的. 在内部, 只有一种在请求清理过程会调用php_sample_descriptor_dtor; 另外一个, 你马上会看到, 它将和webserver的进程或线程保持相同的生命周期.

持久化注册

现在相应的清理函数已经到位了, 是时候创建一些可用的资源结构了. 通常会使用两个独立的函数, 在内部映射到同一个实现上, 但是这可能会使得已经很混杂的主题更加混乱, 所以我们这里只是在sample_fopen()中增加一个布尔类型的参数来完成这件事.

```
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode;
    int filename_len, mode_len;
    zend_bool persist = 0;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,"ss|b",
                &filename, &filename_len, &mode, &mode_len,
                &persist) == FAILURE) {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Invalid filename or mode length");
        RETURN_FALSE;
    }
    fp = fopen(filename, mode);
    if (!fp) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Unable to open %s using mode %s",
                filename, mode);
        RETURN_FALSE;
    }
    if (!persist) {
        fdata = emalloc(sizeof(php_sample_descriptor_data));
        fdata->filename = estrndup(filename, filename_len);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,
                                le_sample_descriptor);
    } else {
        list_entry le;
        char *hash_key;
        int hash_key_len;

        fdata =pemalloc(sizeof(php_sample_descriptor_data),1);
        fdata->filename = pemalloc(filename_len + 1, 1);
        memcpy(data->filename, filename, filename_len + 1);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,
                        le_sample_descriptor_persist);

        /* 在persistent_list中保存一份拷贝 */
        le.type = le_sample_descriptor_persist;
        le.ptr = fdata;
        hash_key_len = spprintf(&hash_key, 0,
                "sample_descriptor:%s:%s", filename, mode);
        zend_hash_update(&EG(persistent_list),
            hash_key, hash_key_len + 1,
            (void*)&le, sizeof(list_entry), NULL);
        efree(hash_key);
    }
}
```

这个函数的核心部分现在你应该已经很熟悉了. 打开一个文件, 将它的名字存储到新分配的内存中, 将它注册为请求特有的资源ID并设置到return_value中. 这一次新的知识点是第二部分, 但它也并不完全陌生.

这里, 你实际上做的事情和ZEND_REGISTER_RESOURCE()所做的基本一致; 不过, 这里不再是获取一个数值索引放到每个请求特有的列表(EG(regular_list))中, 而是赋值给了一个关联key(可以使用它在未来的请求中重新获取资源), 将它放到了持久化列表中, 这个持久化列表(EG(persistent_list))并不会在每个请求结束后被清理.

当这样的一个持久化描述符资源结束其生命周期时, EG(regular_list)的dtor函数将会检查已注册的析构器列表, 发现le_sample_descriptor_persist的(非持久化)析构器为NULL, 因此不做任何事(即不进行释放操作). 这使得FILE *指针和它的char *名字字符串可以在下一个请求中安全的使用.

当资源最终从EG(persistent_list)中移除时(由于进程或线程终止, 或者由于你的扩展有意的移除), 引擎会查找持久化析构器. 由于这个资源类型定义了持久化析构器, 因此它将会被正确的调用pefree()释放原来由pemalloc()分配的内存.

重用

将一个资源条目的拷贝放到persistent_list中, 除了延长执行时间, 占用内存以及文件锁资源, 不会有任何好处, 除非你在后续的请求中以某种方式重用它.

这就是hash_key的来由. 当sample_fopen()被调用时, 无论是持久化或非持久化方式, 你的函数都可以使用请求的文件名和模式参数重新创建hash_key, 并在打开文件之前尝试从persistent_list中使用hash_key查找该资源.

```
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode, *hash_key;
    int filename_len, mode_len, hash_key_len;
    zend_bool persist = 0;
    list_entry *existing_file;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,"ss|b",
                &filename, &filename_len, &mode, &mode_len,
                &persist) == FAILURE) {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Invalid filename or mode length");
        RETURN_FALSE;
    }
    /* 尝试查找已经打开的文件 */
    hash_key_len = spprintf(&hash_key, 0,
            "sample_descriptor:%s:%s", filename, mode);
    if (zend_hash_find(&EG(persistent_list), hash_key,
            hash_key_len + 1, (void **)&existing_file) == SUCCESS) {
        /* There's already a file open, return that! */
        ZEND_REGISTER_RESOURCE(return_value,
            existing_file->ptr, le_sample_descriptor_persist);
        efree(hash_key);
        return;
    }
    fp = fopen(filename, mode);
    if (!fp) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                "Unable to open %s using mode %s",
                filename, mode);
        RETURN_FALSE;
    }
    if (!persist) {
        fdata = emalloc(sizeof(php_sample_descriptor_data));
        fdata->filename = estrndup(filename, filename_len);
```

```
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,
                               le_sample_descriptor);
    } else {
        list_entry le;
        fdata =pemalloc(sizeof(php_sample_descriptor_data),1);
        fdata->filename = pemalloc(filename_len + 1, 1);
        memcpy(data->filename, filename, filename_len + 1);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,
                       le_sample_descriptor_persist);
        /* 存储一份拷贝到persistent_list */
        le.type = le_sample_descriptor_persist;
        le.ptr = fdata;
        /* hash_key现在已经创建了 */
        zend_hash_update(&EG(persistent_list),
            hash_key, hash_key_len + 1,
            (void*)&le, sizeof(list_entry), NULL);
    }
    efree(hash_key);
}
```

因为所有的扩展都使用同一个持久化HashTable存储它们的资源, 因此选择唯一的可复现的hash_key非常重要. sample_fopen()中使用了一种常见的方式: 使用扩展和资源类型名字作为前缀, 接着是创建的资源的关键信息.

活性检查和提前离开

尽管你打开一个文件并无限期的保持打开是安全的, 但是对于其他资源类型则不然, 尤其是远程网络资源可能会变得不可用, 尤其是在请求间长时间不使用时.

因此在取回一个持久化资源时, 对它的可用性检查就非常重要. 如果资源不再可用, 就必须从持久化列表中移除, 并且应该继续以没有找到已分配资源(持久化)的逻辑执行.

下面的假想代码块在持久化列表中的套接字上执行了一个活性检查:

```
if (zend_hash_find(&EG(persistent_list), hash_key,
        hash_key_len + 1, (void**)&socket) == SUCCESS) {
    if (php_sample_socket_is_alive(socket->ptr)) {
        ZEND_REGISTER_RESOURCE(return_value,
                    socket->ptr, le_sample_socket);
        return;
    }
    zend_hash_del(&EG(persistent_list),
                          hash_key, hash_key_len + 1);
}
```

如你所见, 这里所做的只是在运行时手动的从持久化资源列表中移除. 这个行为会触发调用zend_register_list_destructors_ex()注册的持久化dtor函数. 在这段代码完成后, 函数所处的状态和没有从持久化列表中找到资源时的状态一致.

未知类型的取回

此刻你可以创建文件描述符资源, 将它们持久化存储, 并可以透明的获取它们, 但是你试试用sample_fwrite()函数使用你的持久化资源对象? 很无奈, 它不能工作. 回顾一下, 数值ID怎样转换成资源指针:

```
ZEND_FETCH_RESOURCE(fdata, php_sample_descriptor_data*,
    &file_resource, -1, PHP_SAMPLE_DESCRIPTOR_RES_NAME,
    le_sample_descriptor);
```

le_sample_descriptor明确指定了类型名, 因此资源的类型将被验证. 这样做就可以确保你在希望得到php_sample_descriptor_data *结构的资源时, 不会得到mysql_connection_handler *或其他类型的资源. 但这对于混合匹配类型来说就不是一件好事. 我们知道, 在le_sample_descriptor和le_sample_descriptor_persist两种资源类型中

存储了相同的数据结构, 这样做是为了保证用户空间的简单性, 因此, 理想的情况是sample_fwrite()可以公平的接受两种类型.

这可以通过ZEND_FETCH_RESOURCE()的兄弟宏: ZEND_FETCH_RESOURCE2()来解决. 这两个宏唯一的不同是后者允许指定两种资源类型. 这样, 我们就可以对上面的代码进行修改:

```
ZEND_FETCH_RESOURCE2(fdata, php_sample_descriptor_data*,
    &file_resource, -1, PHP_SAMPLE_DESCRIPTOR_RES_NAME,
    le_sample_descriptor, le_sample_descriptor_persist);
```

现在, file_resource中包含的资源ID就可以指向持久化以及非持久化的Sample Descriptor资源了, 并且它们都能够通过验证.

要允许多个资源类型需要使用原生的zend_fetch_resource()实现. 回顾前面, ZEND_FETCH_RESOURCE()宏展开如下:

```
fp = (FILE*) zend_fetch_resource(&file_descriptor TSRMLS_CC, -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME, NULL,
    1, le_sample_descriptor);
ZEND_VERIFY_RESOURCE(fp);
```

类似的, ZEND_FETCH_RESOURCE2()宏展开后也使用了相同的原生函数:

```
fp = (FILE*) zend_fetch_resource(&file_descriptor TSRMLS_CC, -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME, NULL,
    2, le_sample_descriptor, le_sample_descriptor_persist);
ZEND_VERIFY_RESOURCE(fp);
```

看到规律了吗? zend_fetch_resource()第6个以及后面的参数的含义是"我将要匹配N种可能的资源类型, 它们分别是...", 因此, 如果要匹配第三种资源类型(比如: le_sample_othertype), 就可以如下编码:

```
fp = (FILE*) zend_fetch_resource(&file_descriptor TSRMLS_CC, -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME, NULL,
    3, le_sample_descriptor, le_sample_descriptor_persist,
    le_sample_othertype);
ZEND_VERIFY_RESOURCE(fp);
```

如果要四个, 就依此类推.

译注*:* 译者使用的*php-5.4.9*下, 原著的示例不能正常使用, 因此贴出译者自己环境下可编译的代码, 需要的读者可以参考这个示例.

```
PHP_FUNCTION(sample_fopen)
{
    sample_descriptor_data_t    *sddp;
    FILE                        *fp;
    char                        *filename, *mode;
    int                         filename_len, mode_len;
    zend_bool                   persist = 1;
    char                        *hash_key;
    int                         hash_key_len;
    int                         rsrc_l;
    zend_rsrc_list_entry        *le_p;

    if ( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss|b",
            &filename, &filename_len,
            &mode, &mode_len, &persist) == FAILURE ) {
        RETURN_NULL();
    }

    if ( !filename_len || !mode_len ) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Invalid filename or mode length");
        RETURN_FALSE;
    }

    hash_key_len    = spprintf(&hash_key, 0, "sample_descriptor:%s:%s", filename, mode);
    if ( zend_hash_find(&EG(persistent_list), hash_key, hash_key_len + 1, (void **)&le_p) ==
SUCCESS ) {
```

```
        rsrc_l  = ZEND_REGISTER_RESOURCE(return_value, le_p->ptr, le_sample_descriptor_persist);
    } else {
        fp  = fopen(filename, mode);
        if ( !fp ) {
            efree(hash_key);
            php_error_docref(NULL TSRMLS_CC, E_WARNING, "Unable to open %s using mode %s",
                    filename, mode);
            RETURN_FALSE;
        }

        sddp        = pemalloc(sizeof(sample_descriptor_data_t), persist);
        sddp->fname = pestrdup(filename, persist);
        sddp->fp    = fp;
        rsrc_l      = ZEND_REGISTER_RESOURCE(return_value, sddp, persist ?
le_sample_descriptor_persist : le_sample_descriptor);
        if ( persist ) {
            zend_rsrc_list_entry    le;

            le.type = le_sample_descriptor_persist;
            le.ptr  = sddp;
            zend_hash_update(&EG(persistent_list), hash_key, hash_key_len + 1, (void *)&le,
sizeof(zend_rsrc_list_entry), NULL);
        }
    }
    efree(hash_key);
    RETURN_RESOURCE(rsrc_l);
}

PHP_FUNCTION(sample_fwrite)
{
    sample_descriptor_data_t    *sddp;
    zval                        *file_resource;
    char                        *data;
    int                         data_len;

    if ( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",
            &file_resource, &data, &data_len) == FAILURE ) {
        RETURN_FALSE;
    }
    ZEND_FETCH_RESOURCE2(sddp, sample_descriptor_data_t *, &file_resource, -1,
        SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor, le_sample_descriptor_persist);
#if ZEND_DEBUG
php_printf("FILE * pointer: %p\n", sddp->fp);
#endif
    RETURN_LONG(fwrite(data, 1, data_len, sddp->fp));
}

PHP_FUNCTION(sample_fclose)
{
    sample_descriptor_data_t    *sddp;
    zval                        *file_resource;

    if ( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &file_resource) == FAILURE ) {
        RETURN_FALSE;
    }

    ZEND_FETCH_RESOURCE2(sddp, sample_descriptor_data_t *, &file_resource, -1,
SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor, le_sample_descriptor_persist);
    zend_hash_index_del(&EG(regular_list), Z_RESVAL_P(file_resource));

    RETURN_TRUE;
}

PHP_FUNCTION(sample_fname)
```

```
{
    sample_descriptor_data_t    *sddp;
    zval    *file_resource;

    if ( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &file_resource) == FAILURE ) {
        RETURN_FALSE;
    }

    ZEND_FETCH_RESOURCE2(sddp, sample_descriptor_data_t *, &file_resource, -1,
SAMPLE_DESCRIPTOR_RES_NAME, le_sample_descriptor, le_sample_descriptor_persist);
    RETURN_STRING(sddp->fname, 1);
}
```

其他引用计数器

和用户空间变量类似, 已注册资源也有引用计数. 这里, 引用计数指有多少容器结构知道这个资源ID.

现在我们已经知道, 当用户空间变量(zval *)的类型是IS_RESOURCE时, 它并不会真正的持有任何结构的指针, 只是简单的保存一个HashTable的索引值, 通过这个索引值可以在EG(regular_list) HashTable中查找到真正的资源指针.

当一个资源第一次被创建时, 比如通过调用sample_fopen(), 它被放到一个zval *容器中, 并将它的refcount初始化为1, 因为只有一个变量持有它.

```
$a = sample_fopen('notes.txt', 'r');
/* var->refcount = 1, rsrc->refcount = 1 */
```

如果变量被拷贝, 通过第3章"内存管理"的学习可以知道, 并不会创建新的zval *. 而是两个变量共享同一个写时复制的zval *. 这种情况下, zval *的refcount被增加到2; 然而, 此时资源的refcount值仍然为1, 因为它仅被一个zval *持有.

```
$b = $a;
/* var->refcount = 2, rsrc->refcount = 1 */
```

当这两个变量中的一个被unset()时, zval *的refcount减小, 但是它并不会被真的销毁, 因为还有其他变量仍然指向它.

```
unset($b);
/* var->refcount = 1, rsrc->refcount = 1 */
```

现在你还应该知道, 混合引用赋值和写时复制将强制隔离并拷贝到新的zval *中. 当发生这件事时, 资源的引用计数将会增加, 因为它现在被两个zval *持有.

```
$b = $a;
$c = &$a;
/* bvar->refcount = 1, bvar->is_ref = 0
   acvar->refcount = 2, acvar->is_ref = 1
   rsrc->refcount = 2 */
```

现在, 卸载$b将会完全释放它的zval *, 将rsrc->refcount修改为1. 卸载$a或$c但不两者都卸载则不会减小资源的refcount, 因为它们的zval *(acvar)实际上还是存在的. 直到所有三个变量(涉及到两个zval *)都被unset()后, 资源的refcount才会减小到0, 它的析构函数才会被触发.

小结

使用本章涉及的主题, 你就可以开始应用php著名的粘合性了. 资源数据类型使得你的扩展可以很容易的将第三方库的透明指针这样的抽象概念, 连接到用户空间脚本语言中, 使得php更加强大.

接下来两章你将深入php词法中最后但很重要的数据类型. 你将首先探究简单的基于Zend引擎1的类, 接着就要把它迁移到更强大的Zend引擎2中.

php4的对象

曾几何时, 在很早的版本中, php还不支持任何的面向对象编程语法. 在php4中引入了Zend引擎(ZE1), 出现了几个新的特性, 其中就包括对象数据类型.

php对象类型的演化

第一次的面向对象编程(OOP)支持仅实现了对象关联的语义. 用一个php内核开发者的话来说就是"php4的对象只是将一个数组和一些方法绑定到了一起". 它就是现在你要研究的php对象.

Zend引擎(ZE2)的第二个大版本发布是在php5中, 在php的OOP实现中引入了一些新的特性. 例如, 属性和方法可以使用访问修饰符标记它们在你的类定义外面的可见性, 函数的重载可以用来定义内部语言结构的自定义行为, 在多个类的调用链之间可以使用接口实施API标准化. 在你学习到第11章"php5对象"时, 你将通过在php5的类定义中实现这些特性来建立对这些知识的认知.

实现类

在进入OOP的世界之前, 我们需要轻装上阵. 因此, 请将你的扩展恢复到第5章"你的第一个扩展"中刚刚搭建好的骨架形态.

为了和你原有的习作独立, 你可以将这个版本命名为sample2. 将下面的三个文件放入到你php源代码的ext/sample2目录下:

config.m4

```
PHP_ARG_ENABLE(sample2,
  [Whether to enable the "sample2" extension],
  [  enable-sample2       Enable "sample2" extension support])

if test $PHP_SAMPLE2 != "no"; then
  PHP_SUBST(SAMPLE2_SHARED_LIBADD)
  PHP_NEW_EXTENSION(sample2, sample2.c, $ext_shared)
fi
```

php_saple2.h

```
#ifndef PHP_SAMPLE2_H
/* Prevent double inclusion */
#define PHP_SAMPLE2_H

/* Define Extension Properties */
#define PHP_SAMPLE2_EXTNAME    "sample2"
#define PHP_SAMPLE2_EXTVER    "1.0"

/* Import configure options
   when building outside of
   the PHP source tree */
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

/* Include PHP Standard Header */
#include "php.h"

/* Define the entry point symbol
 * Zend will use when loading this module
 */
extern zend_module_entry sample2_module_entry;
```

```
#define phpext_sample2_ptr &sample2_module_entry

#endif /* PHP_SAMPLE2_H */
```

sample2.c

```
#include "php_sample2.h"

static function_entry php_sample2_functions[] = {
    { NULL, NULL, NULL }
};

PHP_MINIT_FUNCTION(sample2)
{
    return SUCCESS;
}

zend_module_entry sample2_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE2_EXTNAME,
    php_sample2_functions,
    PHP_MINIT(sample2),
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE2_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_SAMPLE2
ZEND_GET_MODULE(sample2)
#endif
```

现在, 就像在第5章时一样, 你可以执行phpize, ./configure, make去构建你的sample2.so扩展模块.

你之前的*config.w32*做与这里给出的*config.m4*一样的修改也可以正常工作.

定义类条目

在用户空间中, 定义一个类如下:

```
<?php
class Sample2_FirstClass {
}
?>
```

毫无疑问, 你会猜到, 在扩展中实现它还是有一点难度的. 首先, 你需要在你的源代码文件中, 像上一章定义int le_sample_descriptor一样, 定义一个zend_class_entry指针:

```
zend_class_entry *php_sample2_firstclass_entry;
```

现在, 就可以在MINIT函数中初始化并注册类了.

```
PHP_MINIT_FUNCTION(sample2)
{
    zend_class_entry ce; /* 临时变量 */

    /* 注册类 */
    INIT_CLASS_ENTRY(ce, "Sample2_FirstClass", NULL);
    php_sample2_firstclass_entry =
            zend_register_internal_class(&ce TSRMLS_CC);

    return SUCCESS;
```

```
}
```

构建这个扩展, 测试get_declared_classes(), 将会看到Sample2_FirstClass现在在用户空间可用了.

定义方法的实现

此刻, 你实现的只是一个stdClass, 当然它是可用的. 但实际上你是希望你的类可以做一些事情的.

要达成这个目的, 你就需要回到第5章学到的另外一个知识点了. 将传递给INIT_CLASS_ENTRY()的NULL参数替换为php_sample2_firstclass_functions, 并直接在MINIT函数上面如下定义这个结构:

```
static function_entry php_sample2_firstclass_functions[] = {
    { NULL, NULL, NULL }
};
```

看起来熟悉吗? 当然. 这和你原来定义过程函数的结构相同. 甚至, 设置这个结构的方式也很相似:

```
PHP_NAMED_FE(method1, PHP_FN(Sample2_FirstClass_method1), NULL)
```

当然, 你也可以选用PHP_FE(method1, NULL). 不过回顾一下第5章, 这样做期望找到的函数实现的名字是zif_method1, 它可能潜在的回合其他的method1()实现冲突. 为了函数的名字空间安全, 我们将类名作为方法名的前缀.

PHP_FALIAS(method1, Sample2_FirstClass_method1, NULL)的格式也是可以的; 但它有点不直观, 你以后回过头来看代码的时候可能会产生疑问"为什么当时没有使用PHP_FE()?"

现在, 你已经将一个函数列表附加到类的定义上了, 是时候定义一些方法了. 在php_sample2_firstclass_functions结构上面创建下面的函数:

```
PHP_FUNCTION(Sample2_FirstClass_countProps)
{
    RETURN_LONG(zend_hash_num_elements(Z_OBJPROP_P(getThis())));
}
```

相应的, 在它的函数列表中增加一条PHP_NAMED_FE()条目:

```
static function_entry php_sample2_firstclass_functions[] = {
    PHP_NAMED_FE(countprops,
            PHP_FN(Sample2_FirstClass_countProps), NULL)
    { NULL, NULL, NULL }
};
```

要注意, 这里暴露给用户空间的函数名是全部小写的.为了确保方法和函数名都是大小写不敏感的, 就要求内部函数给出全部小写的名字.

这里唯一的新元素就是getThis(), 在所有的php版本中, 它都会被解析为一个宏, 展开是this_ptr. this_ptr从本质上来说就和用户空间对象方法中的$this含义相同. 如果没有可用的对象实例, 比如方法被静态化调用, 则getThis()返回NULL.

对象方法的数据返回语义和过程函数一致, 参数接受以及arg_info都是同一套东西.

```
PHP_FUNCTION(Sample2_FirstClass_sayHello)
{
    char *name;
    int name_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
                        &name, &name_len) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("Hello");
    PHPWRITE(name, name_len);
    php_printf("!\nYou called an object method!\n");
    RETURN_TRUE;
}
```

构造器

你的类构造器可以和其他的普通类方法一样实现, 它的命名遵循也遵循相同的规则. 特别之处在于你需要将构造器命名为类名. 其他两个ZE1魔术方法__sleep()和__wakeup()也可以以这种方式实现.

继承

php4中, 内部对象之间的继承是不完善的, 最好避免使用. 如果你确实必须继承其他对象, 需要复制下面的ZE1代码:

```
void php_sample2_inherit_from_class(zend_class_entry *ce,
                        zend_class_entry *parent_ce) {
    zend_hash_merge(&ce->function_table,
            &parent_ce->function_table, (void (*)(void *))function_add_ref,
            NULL, sizeof(zval*), 0);
    ce->parent = parent_ce;
    if (!ce->handle_property_get) {
        ce->handle_property_get =
                parent_ce->handle_property_get;
    }
    if (!ce->handle_property_set) {
        ce->handle_property_set =
                parent_ce->handle_property_set;
    }
    if (!ce->handle_function_call) {
        ce->handle_function_call =
                parent_ce->handle_function_call;
    }
    if (!zend_hash_exists(&ce->function_table,
                ce->name, ce->name_length + 1)) {
        zend_function *fe;
        if (zend_hash_find(&parent_ce->function_table,
                parent_ce->name, parent_ce->name_length + 1,
                (void**)fe) == SUCCESS) {
            zend_hash_update(&ce->function_table,
                ce->name, ce->name_length + 1,
                fe, sizeof(zend_function), NULL);
            function_add_ref(fe);
        }
    }
}
```

定义这样一个函数, 你就可以在MINIT中zend_register_internal_class下面对其进行调用:

```
INIT_CLASS_ENTRY(ce, "Sample2_FirstClass", NULL);
/* 假定php_saple2_ancestor是一个已经注册的zend_class_entry */
php_sample2_firstclass_entry =
        zend_register_internal_class(&ce TSRMLS_CC);
php_sample2_inherit_from_class(php_sample2_firstclass_entry
                        ,php_sample2_ancestor);
```

尽管这种方式的继承可以工作, 但还是应该避免*ZE1*中的继承, 因为它并没有设计内部对象的继承处理. 对于*php*中的多数*OOP*实践, *ZE2*和它修订的对象模型是健壮的, 鼓励所有的*OOP*相关任务都直接使用它来处理.

使用实例工作

和其它用户空间变量一样, 对象存储在zval *容器中. 在ZE1中, zval *包含了一个HashTable *用于保存属性, 以及一个zend_class_entry *指针, 指向类的定义. 在ZE2中, 这些值被一个句柄表替代, 增加了一个数值的对象ID, 它和资源ID的用法类似.

很幸运, ZE1和ZE2的这些差异被第2章"变量的里里外外"中介绍的Z_*()族宏隐藏了, 因此在你的扩展中不需要关心这些. 下表10.1列出了两个ZE1的宏, 与非OOP的相关宏一致, 它们也有对应的_P和_PP版本, 用来处理一级或两级间访.

宏	含义
Z_OBJPROP(zv)	获取内建属性的HashTable *
Z_OBJCE(zv)	获取关联的zend_class_entry *

创建实例

大部分时间, 你的扩展都不需要自己创建实例. 而是用户空间调用new关键字创建实例并调用你的类构造器.

但你还是有可能需要创建实例, 比如在工厂方法中, ZEND_API中的object_init_ex(zval *val, zend_class_entry *ce)函数可以用于将对象实例初始化到变量中.

要注意, object_init_ex()函数并不会调用构造器. 当在内部函数中实例化对象时, 构造器必须手动调用. 下面的过程函数重演了new关键字的功能逻辑:

```
PHP_FUNCTION(sample2_new)
{
    int argc = ZEND_NUM_ARGS();
    zval ***argv = safe_emalloc(sizeof(zval**), argc, 0);
    zend_class_entry *ce;
    if (argc == 0 ||
        zend_get_parameters_array_ex(argc, argv) == FAILURE) {
        efree(argv);
        WRONG_PARAM_COUNT;
    }
    /* 第一个参数是类名 */
    SEPARATE_ZVAL(argv[0]);
    convert_to_string(*argv[0]);
    /* 类名存储为小写 */
    php_strtolower(Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]));
    if (zend_hash_find(EG(class_table),
            Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]) + 1,
            (void**)&ce) == FAILURE) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "Class %s does not exist.",
            Z_STRVAL_PP(argv[0]));
        zval_ptr_dtor(argv[0]);
        efree(argv);
        RETURN_FALSE;
    }
    object_init_ex(return_value, ce);
    /* 如果有构造器则调用, 额外的参数将传递给构造器 */
    if (zend_hash_exists(&ce->function_table,
            Z_STRVAL_PP(argv[0]),Z_STRLEN_PP(argv[0]) + 1)) {
        /* 对象有构造器 */
        zval *ctor, *dummy = NULL;

        /* 构造器名字是类名 */
        MAKE_STD_ZVAL(ctor);
        array_init(ctor);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
        if (call_user_function_ex(&ce->function_table,
```

```
                NULL, ctor,
                &dummy, /* 不关心返回值 */
                argc - 1, argv + 1, /* 参数 */
                0, NULL TSRMLS_CC) == FAILURE) {
            php_error_docref(NULL TSRMLS_CC, E_WARNING,
                 "Unable to call constructor");
        }
        if (dummy) {
            zval_ptr_dtor(&dummy);
        }
        zval_ptr_dtor(&ctor);
    }
    zval_ptr_dtor(argv[0]);
    efree(argv);
}
```

不要忘了在php_sample2_functions中增加一个引用. 它是你的扩展的过程函数列表, 而不是类方法的列表. 为了使用php_strtolower()函数, 还需要增加#include "ext/standard/php_string.h".

这个函数是目前你实现的最复杂的一个, 其中有几个全新的特性. 首先就是SEPARATE_ZVAL(), 实际上它的功能你已经实现过很多次, 利用zval_copy_ctor()赋值值到一个临时的结构体, 避免修改原始的内容. 不过它是一个宏版本的封装.

php_strtolower()用于将类名转换为小写, 这样做是为了达到php类名和函数名不区分大小写的目的. 这只是附录B中列出的众多PHPAPI工具函数的其中一个.

EG(class_table)是一个全局变量, 所有的zend_class_entry定义都注册到它里面. 要注意的是在ZE1(php4)中这个HashTable存储了一级间访的zend_class_entry *结构体. 而在ZE2(php5)中, 它被存储为两级间访. 这应该不会是一个问题, 因为对这个HashTable的直接访问并不常见, 但知道这一点总归是有好处的.

call_user_function_ex()是你将在第20章"高级嵌入式"中看到的ZENDAPI调用的一部分. 这里你将从zend_get_parameters_ex()接收到的zval **参数栈第一个元素拿走, 这样做就是为了原封不动的将剩余的参数传递给构造器.

译注: 原著中的代码在译者的环境*(php-5.4.9)*中不能运行, 需要将*zend_class_entry *ce*修改为二级间访. 下面给出译者测试通过的代码.

```
PHP_FUNCTION(sample_new)
{
    int                 argc    = ZEND_NUM_ARGS();
    zval                ***argv = safe_emalloc(sizeof(zval **), argc, 0);
    zend_class_entry    **ce;       /* 译注: 这里在译者的环境(php-5.4.9)是二级间访 */

    /* 数组方式读取所有传入参数 */
    if ( argc == 0 ||
            zend_get_parameters_array_ex(argc, argv) == FAILURE ) {
        efree(argv);
        WRONG_PARAM_COUNT;
    }

    /* 隔离第一个参数(隔离为了使下面的类型转换不影响原始数据) */
    SEPARATE_ZVAL(argv[0]);
    /* 将第一个参数转换为字符串类型, 并转为小写(因为php的类名是不区分大小写的) */
    convert_to_string(*argv[0]);
    php_strtolower(Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]));
    /* 在类的HashTable中查找提供的类是否存在, 如果存在, ce中就得到了对应的zend_class_entry * */
    if ( zend_hash_find(EG(class_table), Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]) + 1, (void
**)&ce) == FAILURE ) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Class %s does not exist.",
Z_STRVAL_PP(argv[0]));
        zval_ptr_dtor(argv[0]);
        efree(argv);
        RETURN_FALSE;
```

```
    }

    /* 将返回值初始化为查找到的类的对象 */
    object_init_ex(return_value, *ce);
    /* 检查类是否有构造器 */
    if ( zend_hash_exists(&(*ce)->function_table, Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]) +
1) ) {
        zval    *ctor, *dummy = NULL;

        /* 将ctor构造为一个数组, 对应的用户空间形式为: array(argv[0], argv[0]),
         * 实际上对应于用户空间调用类的静态方法时$funcname的参数形式:
         * array(类名, 方法名)
         */
        MAKE_STD_ZVAL(ctor);
        array_init(ctor);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
        /* 调用函数 */
        if ( call_user_function_ex(&(*ce)->function_table, NULL, ctor, &dummy, argc - 1, argv +
1, 0, NULL TSRMLS_CC) == FAILURE ) {
            php_error_docref(NULL TSRMLS_CC, E_WARNING, "Unable to call constructor");
        }
        /* 如果有返回值直接析构丢弃 */
        if ( dummy ) {
            zval_ptr_dtor(&dummy);
        }
        /* 析构掉临时使用(用来描述所调用方法名)的数组 */
        zval_ptr_dtor(&ctor);
    }
    /* 析构临时隔离出来的第一个参数(类名) */
    zval_ptr_dtor(argv[0]);
    /* 释放实参列表空间 */
    efree(argv);
}
```

接受实例

有时你的函数或方法需要接受用户空间的对象参数. 对于这种目的, zend_parse_parameters()提供了两种格式的修饰符. 第一种是o(小写字母o), 它将验证传递的参数是否是对象, 并将它设置到传递的zval **中. 下面是这种方式的一个简单的用户空间函数示例, 它返回传入对象的类名.

```
PHP_FUNCTION(sample2_class_getname)
{
    zval *objvar;
    zend_class_entry *objce;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "o",
                              &objvar) == FAILURE) {
        RETURN_NULL();
    }
    objce = Z_OBJCE_P(objvar);
    RETURN_STRINGL(objce->name, objce->name_length, 1);
}
```

第二种修饰符是O(大写字母O), 它不仅允许zend_parse_parameters()验证zval *的类型, 还可以验证所传递对象的类. 要做到这一点, 就需要传递一个zval **容易以及一个zend_class_entry *用来验证, 比如下面的实现就期望传入的是Sample2_FirstClass类的实例:

```
PHP_FUNCTION(sample2_reload)
{
    zval *objvar;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "O",
        &objvar, php_sample2_firstclass_entry) == FAILURE) {
        RETURN_NULL();
    }
    /* 调用假想的"reload"函数 */
    RETURN_BOOL(php_sample2_fc_reload(objvar TSRMLS_CC));
}
```

访问属性

你已经看到了, 类方法可以通过getThis()获取到当前对象实例. 将这个宏的结果或其它包含对象实例的zval *与Z_OBJPROP_P()宏组合, 得到的HashTable *就包含了该对象的所有属性.

对象的属性列表是一个包含zval *的HashTable *, 它只是另外一种放在特殊位置的用户空间变量列表. 和使用zend_hash_find(EG(active_symbol_table), ...)从当前作用域获取变量一样, 你也可以使用第8章"在数组和HashTable上工作"中学习的zend_hash-API去获取或设置对象的属性.

例如, 假设在变量rcvdclass这个zval *中包含的是Sample2_FirstClass的实例, 下面的代码块就可以从它的标准属性HashTable中取到属性foo.

```
zval **fooval;
if (zend_hash_find(Z_OBJPROP_P(rcvdclass),
        "foo", sizeof("foo"), (void**)&fooval) == FAILURE) {
    /* $rcvdclass->foo doesn't exist */
    return;
}
```

要向属性表中增加元素, 则是这个过程的逆向过程, 调用zend_hash_add()去增加元素, 或者也可以将第8章介绍数组时介绍的add_assoc_*()族函数的assoc替换为property来处理对象.

下面的构造器函数为Sample2_FirstClass的实例提供了一些预先设置的默认属性:

```
 PHP_NAMED_FUNCTION(php_sample2_fc_ctor)
{
    /* 为了简洁, 同时演示函数名可以是任意的, 这里实现的函数名并不是类名 */
    zval *objvar = getThis();

    if (!objvar) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                         "Constructor called statically!");
        RETURN_FALSE;
    }

    add_property_long(objvar, "life", 42);
    add_property_double(objvar, "pi", 3.1415926535);
    /* 构造器的返回值会被忽略(请回顾前面构造器的例子) */
}
```

现在可以通过php_sample2_firstclass_functions列表将它连接到对象的构造器:

```
PHP_NAMED_FE(sample2_firstclass, php_sample2_fc_ctor, NULL)
```

译注: 由于前面的*sample_new()*工厂函数在*call_user_function_ex()*调用构造器时使用的是静态方法的调用格式, 因此, 如果是使用这个工厂函数触发的构造器调用, *getThis()*就不会有期望的结果. 因此译者对例子进行了相应的修改, 读者如果在这块遇到问题可以参考译者的代码.

```
PHP_FUNCTION(sample_new)
{
    int                 argc    = ZEND_NUM_ARGS();
    zval                ***argv = safe_emalloc(sizeof(zval **), argc, 0);
    zend_class_entry    **ce;       /* 译注: 这里在译者的环境(php-5.4.9)是二级间访 */

    /* 数组方式读取所有传入参数 */
    if ( argc == 0 ||
            zend_get_parameters_array_ex(argc, argv) == FAILURE ) {
```

```
        efree(argv);
        WRONG_PARAM_COUNT;
    }

    /* 隔离第一个参数(隔离为了使下面的类型转换不影响原始数据) */
    SEPARATE_ZVAL(argv[0]);
    /* 将第一个参数转换为字符串类型, 并转为小写(因为php的类名是不区分大小写的) */
    convert_to_string(*argv[0]);
    php_strtolower(Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]));
    /* 在类的HashTable中查找提供的类是否存在, 如果存在, ce中就得到了对应的zend_class_entry * */
    if ( zend_hash_find(EG(class_table), Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]) + 1, (void
**)&ce) == FAILURE ) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "Class %s does not exist.",
Z_STRVAL_PP(argv[0]));
        zval_ptr_dtor(argv[0]);
        efree(argv);
        RETURN_FALSE;
    }

    /* 将返回值初始化为查找到的类的对象 */
    object_init_ex(return_value, *ce);
    /* 检查类是否有构造器 */
    if ( zend_hash_exists(&(*ce)->function_table, Z_STRVAL_PP(argv[0]), Z_STRLEN_PP(argv[0]) +
1) ) {
#define DYNAMIC_CONSTRUCTOR
#ifndef DYNAMIC_CONSTRUCTOR
        zval    *ctor;
#endif
        zval    *dummy = NULL;

#ifndef DYNAMIC_CONSTRUCTOR
        /* 将ctor构造为一个数组, 对应的用户空间形式为: array(argv[0], argv[0]),
         * 实际上对应于用户空间调用类的静态方法时$funcname的参数形式:
         * array(类名, 方法名)
         */
        MAKE_STD_ZVAL(ctor);
        array_init(ctor);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
        zval_add_ref(argv[0]);
        add_next_index_zval(ctor, *argv[0]);
#endif
        /* 调用函数 */
        if ( call_user_function_ex(&(*ce)->function_table,
#ifndef DYNAMIC_CONSTRUCTOR
                NULL, ctor,
#else
                &return_value, *argv[0],
#endif
                &dummy, argc - 1, argv + 1, 0, NULL TSRMLS_CC) == FAILURE ) {
            php_error_docref(NULL TSRMLS_CC, E_WARNING, "Unable to call constructor");
        }
        /* 如果有返回值直接析构丢弃 */
        if ( dummy ) {
            zval_ptr_dtor(&dummy);
        }
#ifndef DYNAMIC_CONSTRUCTOR
        /* 析构掉临时使用(用来描述所调用方法名)的数组 */
        zval_ptr_dtor(&ctor);
#endif
    }
    /* 析构临时隔离出来的第一个参数(类名) */
    zval_ptr_dtor(argv[0]);
    /* 释放实参列表空间 */
```

```
    efree(argv);
}
```

译注: 现在, 就可以用函数中是否定义*DYNAMIC_CONSTRUCTOR*这个宏来切换构造器的调用方式, 以方便读者理解.

小结

尽管ZE1/php4提供的类功能最好少用, 但是由于当前php4在产品环境下还是广泛使用的, 因此做这个兼容还是有好处的. 本章涉及的技术可以让你灵活的编写各种功能的代码, 它们现在可以编译运行, 并且未来也将继续可以工作.

下一章, 你将看到php5中真正的面向对象, 如果你想要OOP, 从中你就可以得到升级的理由, 并且, 升级后你肯定再也不愿回头.

php5对象

将php5的对象和它的先辈php4对象进行比较实在有些不公平, 不过php5对象使用的API函数还是遵循php4的API构建的. 如果你已经阅读了第10章"php4对象", 你将会对本章内容多少有些熟悉. 在开始本章之前, 可以像第10章开始时一样, 重命名扩展为sample3并清理多余的代码, 只保留扩展的骨架代码.

进化史

在php5对象变量中有两个关键的组件. 第一个是一个数值的标识, 它和第9章"资源数据类型"中介绍的数值资源ID非常相似, 扮演了一个用来在对应表中查找对象实例的key的角色. 在这个实例表中的元素包含了到zend_class_entry的引用以及内部的属性表.

第二个元素是对象变量的句柄表, 使用它可以自定义Zend引擎对实例的处理方式. 在本章后面你将看到这个句柄表.

zend_class_entry

类条目是你在用户空间定义的类的内部表示. 正如你在前一章所见, 这个结构通过调用INIT_CLASS_ENTRY()初始化, 参数为类名和它的函数表. 接着在MINIT阶段使用zend_register_internal_class()注册.

```
zend_class_entry *php_sample3_sc_entry;
#define PHP_SAMPLE3_SC_NAME "Sample3_SecondClass"
static function_entry php_sample3_sc_functions[] = {
    { NULL, NULL, NULL }
};

PHP_MINIT_FUNCTION(sample3)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce, PHP_SAMPLE3_SC_NAME,
                             php_sample3_sc_functions);
    php_sample3_sc_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

方法

如果你已经阅读了上一章, 你可能就会想"到现在为止看起来几乎一样啊?", 到现在为止, 你是对的. 现在我们开始定义一些对象方法. 你将开始看到一些非常确定的并且大受欢迎的不同.

```
PHP_METHOD(Sample3_SecondClass, helloWorld)
{
    php_printf("Hello World\n");
}
```

在Zend引擎2中引入了PHP_METHOD()宏, 它是对PHP_FUNCTION()宏的封装, 将类名和方法名联合起来, 不用像php4中手动定义方法名了. 通过使用这个宏, 在扩展中你的代码和其他维护者的代码的名字空间解析规范就保持一致了.

定义

定义一个方法的实现, 和其他函数一样, 只不过是将它连接到类的函数表中. 除了用于实现的PHP_METHOD()宏, 还有一些新的宏可以用在函数列表的定义中.

- PHP_ME(classname, methodname, arg_info, flags)

PHP_ME()相比于第5章"你的第一个扩展"中介绍的PHP_FE()宏, 增加了一个classname参数, 以及末尾的一个flags参数(用来提供public, protected, private, static等访问控制, 以及abstract和其他一些选项). 比如要定义helloWorld方法, 就可以如下定义:

```
PHP_ME(Sample3_SecondClass,helloWorld,NULL,ZEND_ACC_PUBLIC)
```

- PHP_MALIAS(classname, name, alias, arg_info, flags)

和PHP_FALIAS()宏很像, 这个宏允许你给alias参数描述的方法(同一个类中的)实现提供一个name指定的新名字. 例如, 要复制你的helloWorld方法则可以如下定义

```
PHP_MALIAS(Sample3_SecondClass, sayHi, helloWorld,
                           NULL, ZEND_ACC_PUBLIC)
```

- PHP_ABSTRACT_ME(classname, methodname, arg_info)

内部类中的抽象方法很像用户空间的抽象方法. 在父类中它只是一个占位符, 期望它的子类提供真正的实现. 你将在接口一节中使用这个宏, 接口是一种特殊的class_entry.

- PHP_ME_MAPPING(methodname, functionname, arg_info)

最后一种方法定义的宏是针对同时暴露OOP和非OOP接口的扩展(比如mysqli既有过程化的mysqli_query(), 也有面向对象的MySQLite::query(), 它们都使用了相同的实现.)的. 假定你已经有了一个过程化函数, 比如第5章写的sample_hello_world(), 你就可以使用这个宏以下面的方式将它附加为一个类的方法(要注意, 映射的方法总是public, 非static, 非final的):

```
PHP_ME_MAPPING(hello, sample_hello_world, NULL)
```

现在为止, 你看到的方法定义都使用了ZEND_ACC_PUBLIC作为它的flags参数. 实际上, 这个值可以是下面两张表的任意值的位域运算组合, 并且它还可以和本章后面"特殊方法"一节中要介绍的一个特殊方法标记使用位域运算组合.

类型标记	含义
ZEND_ACC_STATIC	方法可以静态调用. 实际上, 这就表示, 方法如果通过实例调用, $this或者更确切的说this_ptr, 并不会被设置到实例作用域中
ZEND_ACC_ABSTRACT	方法并不是真正的实现. 当前方法应该在被直接调用之前被子类覆写.
ZEND_ACC_FINAL	方法不能被子类覆写

可见性标记	含义
ZEND_ACC_PUBLIC	可以在对象外任何作用域调用. 这和php4方法的可见性是一样的
ZEND_ACC_PROTECTED	只能在类中或者它的子类中调用
ZEND_ACC_PRIVATE	只能在类中调用

比如, 由于你前面定义的Sample3_SecondClass::helloWorld()方法不需要对象实例, 你就可以将它的定义从简单的ZEND_ACC_PUBLIC修改为ZEND_ACC_PUBLIC | ZEND_ACC_STATIC, 这样引擎知道了就不会去提供(实例)了.

魔术方法

除了ZE1的魔术方法外, ZE2新增了很多魔术方法, 如下表(或者可以在http://www.php.net/language.oop5.magic中找到)

方法	用法
__construct(...)	可选的自动调用的对象构造器(之前定义的是和类名一致的方法). 如果__construct()和classname()两种实现都存在, 在实例化的过程中, 将优先调用__construct()
__destruct()	当实例离开作用域, 或者请求整个终止, 都将导致隐式的调用实例的__destruct()方法去处理一些清理工作, 比如关闭文件或网络句柄.
__clone()	默认情况下, 所有的实例都是真正的引用传值. 在php5中, 要想真正的拷贝一个对象实例, 就要使用clone关键字. 当在一个对象实例上调用clone关键字时, __clone()方法就会隐含的被执行, 它允许对象去复制一些需要的内部资源数据.
__toString()	在用文本表示一个对象时, 比如当直接在对象上使用echo或print语句时, __toString()方法将自动的被引擎调用. 类如果实现这个魔术方法, 应该返回一个包含描述对象的当前状态的字符串.
__get($var)	如果脚本中请求一个对象不可见的属性(不存在或者由于访问控制导致不可见)时, __get()魔术方法将被调用, 唯一的参数是所请求的属性名. 实现可以使用它自己的内部逻辑去确定最合理的返回值返回.
__set($var, $value)	和__get()很像, __set()提供了与之相反的能力, 它用来处理赋值给对象的不可见属性时的逻辑.__set()的实现可以选择隐式的在标准属性表中创建这些变量, 以其他存储机制设置值, 或者直接抛出错误并丢弃值.
__call($fname, $args)	调用对象的未定义方法时可以通过使用__call()魔术方法实现漂亮的处理. 这个方法接受两个参数: 被调用的方法名, 包含调用时传递的所有实参的数值索引的数组.
__isset($varname)	php5.1.0之后, isset($obj->prop)的调用不仅是检查$obj中是否有prop这个属性, 它还会调用$obj中定义的__isset()方法, 动态的评估尝试使用动态的__get()和__set()方法是否能成功读写属性

方法	用法
__unset($varname)	类似于__isset(), php 5.1.0为unset()函数引入了一个简单的OOP接口, 它可以用于对象属性, 虽然这个属性可能在对象的标准属性表中并不存在, 但它可能对于__get()和__set()的动态属性空间是有意义的, 因此引入__unset()来解决这个问题.

还有其他的魔术方法功能, 它们可以通过某些接口来使用, 比如*ArrayAccess*接口以及一些*SPL*接口.

在一个内部对象的实现中, 每个这样的"魔术方法"都可以和其他方法一样实现, 只要在对象的方法列表中正确的定义PHP_ME()以及PUBLIC访问修饰符即可.对于 __get(), __set(), __call(), __isset()以及__unset(), 它们要求传递参数, 你必须定义恰当的arg_info结构来指出方法需要一个或两个参数. 下面的代码片段展示了这些木梳函数的arg_info和它们对应的PHP_ME()条目:

```
static
    ZEND_BEGIN_ARG_INFO_EX(php_sample3_one_arg, 0, 0, 1)
    ZEND_END_ARG_INFO()
static
    ZEND_BEGIN_ARG_INFO_EX(php_sample3_two_args, 0, 0, 2)
    ZEND_END_ARG_INFO()
static function_entry php_sample3_sc_functions[] = {
    PHP_ME(Sample3_SecondClass, __construct, NULL,
                        ZEND_ACC_PUBLIC|ZEND_ACC_CTOR)
    PHP_ME(Sample3_SecondClass, __destruct, NULL,
                        ZEND_ACC_PUBLIC|ZEND_ACC_DTOR)
    PHP_ME(Sample3_SecondClass, __clone, NULL,
                        ZEND_ACC_PUBLIC|ZEND_ACC_CLONE)
    PHP_ME(Sample3_SecondClass, __toString, NULL,
                        ZEND_ACC_PUBLIC)
    PHP_ME(Sample3_SecondClass, __get, php_sample3_one_arg,
                        ZEND_ACC_PUBLIC)
    PHP_ME(Sample3_SecondClass, __set, php_sample3_two_args,
                        ZEND_ACC_PUBLIC)
    PHP_ME(Sample3_SecondClass, __call, php_sample3_two_args,
                        ZEND_ACC_PUBLIC)
    PHP_ME(Sample3_SecondClass, __isset, php_sample3_one_arg,
                        ZEND_ACC_PUBLIC)
    PHP_ME(Sample3_SecondClass, __unset, php_sample3_one_arg,
                        ZEND_ACC_PUBLIC)
    { NULL, NULL, NULL }
};
```

要注意__construct, __destruct, __clone使用位域运算符增加了额外的常量. 这三个访问修饰符对于方法而言是特殊的, 它们不能被用于其他地方.

属性

php5中对象属性的访问控制与方法的可见性有所不同. 在标准属性表中定义一个公开属性时, 就像你通常期望的, 你可以使用zend_hash_add()或add_property_*()族函数.

对于受保护的和私有的属性, 则需要使用新的ZEND_API函数:

```
void zend_mangle_property_name(char **dest, int *dest_length,
                        char *class, int class_length,
                        char *prop, int prop_length,
                        int persistent)
```

这个函数会分配一块新的内存, 构造一个"\0classname\0propname"格式的字符串. 如果类名是特定的类名, 比如Sample3_SecondClass, 则属性的可见性为private, 只能在Sample3_SecondClass对象实例内部可见.

如果类名指定为*, 则属性的可见性是protected, 它可以被对象实例所属类的所有祖先和后辈访问. 实际上, 属性可以以下面方式增加到对象上:

```
void php_sample3_addprops(zval *objvar)
{
    char *propname;
    int propname_len;
    /* public */
    add_property_long(objvar, "Chapter", 11);
    /* protected */
    zend_mangle_property_name(&propname, &propname_len,
        "*", 1, "Title", sizeof("Title")-1, 0);
    add_property_string_ex(objvar, propname, propname_len,
        "PHP5 Objects", 1 TSRMLS_CC);
    efree(propname);
    /* Private */
    zend_mangle_property_name(&propname, &propname_len,
        "Sample3_SecondClass",sizeof("Sample3_SecondClass")-1,
        "Section", sizeof("Section")-1, 0);
    add_property_string_ex(objvar, propname, propname_len,
        "Properties", 1 TSRMLS_CC);
    efree(propname);
}
```

通过_ex()版的add_property_*()族函数, 可以明确标记属性名的长度. 这是需要的, 因为在protected和private属性名中会包含NULL字节, 而strlen()认为NULL字节是字符串终止标记, 这样将导致属性名被认为是空. 要注意的是_ex()版本的add_property_*()函数还要求显式的传递TSRMLS_CC. 而通常它是通过宏扩展隐式的传递的.

定义类常量和定义类属性非常相似. 两者的关键不同点在于它们的持久性, 因为属性的生命周期是伴随的实例的, 它发生在请求中, 而常量是和类定义在一起的, 只能在MINIT阶段定义.

由于标准的zval *维护宏的函数假定了非持久性, 所以你需要手动写不少代码. 考虑下面的函数:

```
void php_sample3_register_constants(zend_class_entry *ce)
{
    zval *constval;


    /* 基本的标量值可以使用z_*()去设置它们的值 */
    constval = pemalloc(sizeof(zval), 1);
    INIT_PZVAL(constval);
    ZVAL_DOUBLE(constval, 2.7182818284);
    zend_hash_add(&ce->constants_table, "E", sizeof("E"),
                  (void*)&constval, sizeof(zval*), NULL);

    /* 字符串需要额外的空间分配 */
    constval = pemalloc(sizeof(zval), 1);
    INIT_PZVAL(constval);
    Z_TYPE_P(constval) = IS_STRING;
    Z_STRLEN_P(constval) = sizeof("Hello World") - 1;
    Z_STRVAL_P(constval) = pemalloc(Z_STRLEN_P(constval)+1, 1);
    memcpy(Z_STRVAL_P(constval), "Hello World",
                            Z_STRLEN_P(constval) + 1);
    zend_hash_add(&ce->constants_table,
                  "GREETING", sizeof("GREETING"),
                  (void*)&constval, sizeof(zval*), NULL);
```

```
    /* Objects, Arrays, and Resources can't be constants */
}
PHP_MINIT_FUNCTION(sample3)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce, PHP_SAMPLE3_SC_NAME,
                            php_sample3_sc_functions);
    php_sample3_sc_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_register_constants(php_sample3_sc_entry);
    return SUCCESS;
}
```

在这之下, 这些类常量就可以访问了, 分别是: Sample3_SecondClass::E和Sample3_SecondClass::GREETING.

接口

接口的定义和类的定义除了几个差异外基本一致. 首先是所有的方法都定义为抽象的, 这可以通过PHP_ABSTRACT_ME()宏来完成.

```
static function_entry php_sample3_iface_methods[] = {
    PHP_ABSTRACT_ME(Sample3_Interface, workerOne, NULL)
    PHP_ABSTRACT_ME(Sample3_Interface, workerTwo, NULL)
    PHP_ABSTRACT_ME(Sample3_Interface, workerThree, NULL)
    { NULL, NULL, NULL }
};
```

由于这些方法是抽象的, 所以不需要实现. 接下来的第二个差异就是注册. 和一个实际的类注册类似, 首先调用INIT_CLASS_ENTRY和zend_register_internal_class.

当类(zend_class_entry)可用时, 最后一部就是标记这个类是接口, 实现方法如下:

```
zend_class_entry *php_sample3_iface_entry;
PHP_MINIT_FUNCTION(sample3)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce, "Sample3_Interface",
                        php_sample3_iface_methods);
    php_sample3_iface_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_iface_entry->ce_flags|= ZEND_ACC_INTERFACE;
```

实现接口

假设你想让Sample3_SecondClass这个类实现Sample3_Interface这个接口, 就需要实现这个接口定义的所有抽象方法:

```
PHP_METHOD(Sample3_SecondClass,workerOne)
{
    php_printf("Working Hard.\n");
}
PHP_METHOD(Sample3_SecondClass,workerTwo)
{
    php_printf("Hardly Working.\n");
}
PHP_METHOD(Sample3_SecondClass,workerThree)
{
    php_printf("Going wee-wee-wee all the way home.\n");
}
```

接着在php_sample3_sc_functions列表中定义它们:

```
PHP_ME(Sample3_SecondClass,workerOne,NULL,ZEND_ACC_PUBLIC)
PHP_ME(Sample3_SecondClass,workerTwo,NULL,ZEND_ACC_PUBLIC)
PHP_ME(Sample3_SecondClass,workerThree,NULL,ZEND_ACC_PUBLIC)
```

最后, 定义你新注册的类实现php_sample3_iface_entry接口:

```
PHP_MINIT_FUNCTION(sample3)
```

```
{
    zend_class_entry ce;
    /* 注册接口 */
    INIT_CLASS_ENTRY(ce, "Sample3_Interface",
                        php_sample3_iface_methods);

    php_sample3_iface_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_iface_entry->ce_flags|= ZEND_ACC_INTERFACE;
    /* 注册实现接口的类 */
    INIT_CLASS_ENTRY(ce, PHP_SAMPLE3_SC_NAME,
                            php_sample3_sc_functions);
    php_sample3_sc_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_register_constants(php_sample3_sc_entry);
    /* 声明实现关系 */
    zend_class_implements(php_sample3_sc_entry TSRMLS_CC,
                1, php_sample3_iface_entry);
    return SUCCESS;
}
```

如果Sample3_SecondClass实现了其他接口, 比如ArrayAccess, 就需要将对应的类(zend_class_entry)作为附加参数增加到zend_class_implements()调用中, 并将现在传递为数字1的参数值相应的增大为2:

```
zend_class_implements(php_sample3_sc_entry TSRMLS_CC,
            2, php_sample3_iface_entry, php_other_interface_entry);
```

句柄

ZE2并没有把所有的对象实例看做是相同的, 它为每个对象实例关联了句柄表. 当在一个对象上执行特定的操作时, 引擎调用执行对象的句柄表中自定义的行为.

标准句柄

默认情况下, 每个对象都被赋予了std_object_handlers这个内建句柄表. std_object_handlers中对应的句柄方法以及它们的行为定义如下:

- void add_ref(zval *object TSRMLS_DC)

当对象值的refcount增加时被调用, 比如, 当一个对象变量赋值给新的变量时. add_ref和del_ref函数的默认行为都是调整内部对象存储的refcount.

- void del_ref(zval *object TSRMLS_DC)

和add_ref类似, 这个方法也在修改refcount时调用, 通常是在unset()对象变量时发生的.

- zend_object_value clone_obj(zval *object TSRMLS_DC)

用于利用已有的对象实例创建一个新的实例. 默认行为是创建一个新的对象实例, 将它和原来的句柄表关联, 拷贝属性表, 如果该对象的类定义了__clone()方法, 则调用它让新的对象执行一些附加的复制工作.

- zval *read_property(zval *obj, zval *prop, int type TSRMLS_DC)
- void write_property(zval *obj, zval *prop, zval *value TSRMLS_DC)

在用户空间尝试以$obj->prop方式访问, 去读写对象的属性时, read_property/write_property对应的被调用. 默认的处理是首先在标准属性表中查找属性. 如果属性没有定义, 则检查是否存在__get()或__set()魔术方法, 如果有则调用该方法.

- zval **get_property_ptr_ptr(zval *obj, zval *value TSRMLS_DC)

get_property_ptr_ptr()是read_property()的一个变种, 它的含义是允许调用作用域直接将当前的zval *替换为新的. 默认的行为是返回标准属性表中该属性的指针地址. 如果不存在, 并且没有__get()/__set()魔术方法, 则隐式创建并返回指针. 如果存在__get()或

__set()方法, 则导致这个句柄失败, 使得引擎转而依靠单独的read_property和write_property调用.

- zval *read_dimension(zval *obj, zval *idx, int type TSRMLS_DC)
- void write_dimension(zval *obj, zval *idx, zval *value TSRMLS_DC)

read_dimension()和write_dimension()类似于对应的read_property()和write_property(); 不过它们在使用$obj['idx']方式将对象作为数组访问时被触发. 如果对象的类没有实现ArrayAccess接口, 默认的行为是触发一个错误; 否则它就会调用魔术方法offsetget($idx)或offsetset($idx, $value).

- zval *get(zval *obj TSRMLS_DC)
- void set(zval *obj, zval *value TSRMLS_DC)

在设置或取回对象的值时, 则会在对象上调用get()或set()方法. 对象自身作为第一个参数被传递. 对于set, 新的值作为第二个参数传递; 实际上, 这些方法被用于算数运算中. 这些操作没有默认处理器.

- int has_property(zval *obj, zval *prop, int chk_type TSRMLS_DC)

当在一个对象属性上调用isset()时, 这个句柄被调用. 默认情况下标准的处理器会检查prop指定的属性名, 在php 5.1.0中如果没有找到这个属性, 并且定义了__isset()方法, 则会调用这个方法. chk_type参数的值如果是2则仅需要属性存在, 如果chk_type值为0, 则必须存在并且不能是IS_NULL的值, 如果chk_type值为1, 则属性必须存在并且必须是非FALSE的值. 注意: 在php 5.0.x中, chk_type的含义和has_dimension的chk_type一致.

- int has_dimension(zval *obj, zval *idx, int chk_type TSRMLS_DC)

当将对象看做数组调用isset()时(比如isset($obj['idx'])), 使用这个处理器. 默认的标准处理器会检查对象是否实现了ArrayAccess接口, 如果实现了, 则调用offsetexists($idx)方法. 如果没有找到(指调用offsetexists()), 则和没有实现offsetexists()方法一样, 返回0. 否则, 如果chk_type为0, 直接返回true(1). chk_type为1标识它必须调用对象的offsetget($idx)方法并测试返回值, 检查值是非FALSE才返回TRUE(1).

- void unset_property(zval *obj, zval *prop TSRMLS_DC)
- void unset_dimension(zval *obj, zval *idx TSRMLS_DC)

这两个方法在尝试卸载对象属性时(或将对象以数组方式应用调用unset()时)被调用. unset_property()处理器要么从标准属性表删除属性(如果存在), 要么就尝试调用实现的__unset($prop)方法(php 5.1.0中), unset_dimension()则在类实现了ArrayAccess时, 调用offsetunset($idx)方法.

- HashTable *get_properties(zval *obj TSRMLS_DC)

当内部函数使用Z_OBJPROP()宏从标准属性表中读取属性时, 实际上是调用了这个处理器. php对象的默认处理器是解开并返回Z_OBJ_P(object)->properties, 它是真正的标准属性表.

- union _zend_function *get_method(zval **obj_ptr char *method_name, int methodname_len TSRMLS_DC)

这个处理器在解析类的function_table中的对象方法时被调用. 如果在主的function_table中不存在方法, 则默认的处理器返回一个指向对对象的__call($name, $args)方法包装的zend_function *指针.

- int call_method(char *method, INTERNAL_FUNCTION_PARAMETERS)

定义为ZEND_OVERLOADED_FUNCTION类型的函数将以call_method处理器的方式执行. 默认情况下, 这个处理器是未定义的.

- union _zend_function *get_constructor(zval *obj TSRMLS_DC)

类似于get_method()处理器, 这个处理器返回一个对对应对象方法的引用. 类的zend_class_entry中构造器是特殊方式存储的, 这使得它比较特殊. 对这个方法的重写非常少见.

- zend_class_entry *get_class_entry(zval *obj TSRMLS_DC)

和get_constructor()类似, 这个处理器也很少被重写. 它的目的是将一个对象实例映射回它原来的类定义.

- int get_class_name(zval *object, char **name zend_uint *len, int parent TSRMLS_DC)

get_class_entry()就是get_class_name()其中的一步, 在得到对象的zend_object后, 它将对象的类名或它的父类名(这依赖于参数parent的值)复制一份返回. 返回的类名拷贝必须使用非持久化存储(emalloc()).

- int compare_objects(zval *obj1, zval *obj2 TSRMLS_DC)

当比较操作符(比如: ==, !=, <=, <, >, >=)用在两个对象上时, 在操作数(参与比较的两个对象)上调用compare_objects()就是这个工作的第一部分. 它的返回值通常是1, 0, -1, 分别代表大于, 等于, 小于.默认 情况下, 对象是基于它们的标准属性表比较的, 使用的比较规则和第8章"在数组和HashTable上工作"中学习的数组比较规则一样.

- int cast_object(zval *src, zval *dst, int type, int should_free TSRMLS_DC)

当尝试将对象转换为其他数据类型时, 会触发这个处理器. 如果将should_free设置为非0值, zval_dtor()将会在dst上调用, 首先释放内部的资源. 总之, 处理器应该尝试将src中的对象表示为dst给出的zval *的类型中. 这个处理器默认是未定义的, 但当有它的时候, 应该返回SUCCESS或FAILURE.

- int count_elements(zval *obj, long *count TSRMLS_DC)

实现了数组访问的对象应该定义这个处理器, 它将设置当前的元素数量到count中并返回SUCCESS. 如果当前实例没有实现数组访问, 则它应该返回FAILURE, 以使引擎回头去检查标准属性表.

译注: 上面的句柄表和译者使用的*php-5.4.9*中已经不完全一致, 读者在学习这一部分的时候, 可以参考*Zend/zend_object_handlers.c*中最下面的标准处理器句柄表.

魔术方法第二部分

使用前面看到的对象句柄表的自定义版本, 可以让内部类提供与在用户空间基于对象或类的__xxx()魔术方法相比, 相同或更多的能力.将这些自定义的句柄设置到对象实例上首先要求创建一个新的句柄表. 因为你通常不会覆写所有的句柄, 因此首先将标准句柄表拷贝到你的自定义句柄表中再去覆写你想要修改的句柄就很有意义了:

```
static zend_object_handlers php_sample3_obj_handlers;
int php_sample3_has_dimension(zval *obj, zval *idx,
                        int chk_type TSRMLS_DC)
{
    /* 仅在php版本>=1.0时使用 */
    if (chk_type == 0) {
       /* 重新映射chk_type的值 */
       chk_type = 2;
    }
    /* 当chk_type值为1时保持不变. 接着使用标准的hash_property方法执行逻辑 */
    return php_sample3_obj_handlers.has_property(obj,
                            idx, chk_type TSRMLS_CC);
}
PHP_MINIT_FUNCTION(sample3)
{
    zend_class_entry ce;
    zend_object_handlers *h = &php_sample3_obj_handlers;


    /* 注册接口 */
```

```
    INIT_CLASS_ENTRY(ce, "Sample3_Interface",
                        php_sample3_iface_methods);
    php_sample3_iface_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_iface_entry->ce_flags = ZEND_ACC_INTERFACE;
    /* 注册SecondClass类 */
    INIT_CLASS_ENTRY(ce, PHP_SAMPLE3_SC_NAME,
                            php_sample3_sc_functions);
    php_sample3_sc_entry =
                zend_register_internal_class(&ce TSRMLS_CC);
    php_sample3_register_constants(php_sample3_sc_entry);


    /* 实现AbstractClass接口 */
    zend_class_implements(php_sample3_sc_entry TSRMLS_CC,
                1, php_sample3_iface_entry);


    /* 创建自定义句柄表 */
    php_sample3_obj_handlers = *zend_get_std_object_handlers();


    /* 这个句柄表的目的是让$obj['foo']的行为等价于$obj->foo */
    h->read_dimension = h->read_property;
    h->write_dimension = h->write_property;
    h->unset_dimension = h->unset_property;
#if PHP_MAJOR_VERSION > 5 || \
            (PHP_MAJOR_VERSION == 5 && PHP_MINOR_VERSION > 0)
    /* php-5.1.0中, has_property和has_dimension的chk_type含义不同, 为使它们行为一致, 自己包装一个函数 */
    h->has_dimension = php_sample3_has_dimension;

#else
    /* php 5.0.x的has_property和has_dimension行为一致 */
    h->has_dimension = h->has_property;
#endif


    return SUCCESS;
}
```

要将这个句柄表应用到对象上, 你有两种选择. 最简单也是最具代表性的就是实现一个构造器方法, 并在其中重新赋值变量的句柄表.

```
PHP_METHOD(Sample3_SecondClass,__construct)
{
    zval *objptr = getThis();


    if (!objptr) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
                    "Constructor called statically!");
        RETURN_FALSE;
    }
    /* 执行正常的构造器任务... */
    /* 覆写句柄表 */
    Z_OBJ_HT_P(objptr) = &php_sample3_obj_handlers;
}
```

当构造器返回时, 对象就有了新的句柄表以及对应的自定义行为. 还有一种更加受欢迎的方法是覆写类的对象创建函数.

```
zend_object_value php_sample3_sc_create(zend_class_entry *ce
                                        TSRMLS_DC)
{
    zend_object *object;
    zend_object_value retval;
```

```
    /* 返回Zend创建的对象 */
    retval = zend_objects_new(&object, ce TSRMLS_CC);
    /* 覆写create_object时, 属性表必须手动初始化 */
    ALLOC_HASHTABLE(object->properties);
    zend_hash_init(object->properties, 0, NULL,
                                    ZVAL_PTR_DTOR, 0);
    /* 覆写默认句柄表 */
    retval.handlers = &php_sample3_obj_handlers;
    /* 这里可能会执行其他对象初始化工作 */
    return retval;
}
```

这样就可以在MINIT阶段注册类(zend_class_entry)之后直接将自定义句柄表附加上去.

```
INIT_CLASS_ENTRY(ce, PHP_SAMPLE3_SC_NAME,
                        php_sample3_sc_functions);
php_sample3_sc_entry =
            zend_register_internal_class(&ce TSRMLS_CC);
php_sample3_sc_entry->create_object= php_sample3_sc_create;
php_sample3_register_constants(php_sample3_sc_entry);
zend_class_implements(php_sample3_sc_entry TSRMLS_CC,
            1, php_sample3_iface_entry);
```

这两种方法唯一可预见的不同是它们发生的时机不同. 引擎在碰到new Sample3_SecondClass后会在处理构造器及它的参数之前调用create_object. 通常, 你计划覆盖的各个点使用的方法(create_object Vs. __construct)应该一致.

译注*: php-5.4.9*中, *xxx_property/xxx_dimension*这一组句柄的原型是不一致的, 因此, 按照原著中的示例, 直接将*xxx_property/xxx_dimension*进行映射已经不能工作, 要完成上面的功能, 需要对*4*个句柄均包装一个函数去映射. 由于译者没有详细跟踪具体在哪一个版本发生了这些改变, 因此这里不给出译者测试的示例*(*没有做兼容性处理检查*)*, 如果读者碰到这个问题, 请检查自己所使用*php*版本中两组句柄原型的差异并进行相应修正.

小结

毋庸置疑, php5/ZE2的对象模型比它的前辈php4/ZE1中的对象模型更加复杂. 在看完本章中介绍的所有特性和实现细节后, 你可能已经被它的所包含的信息量搞得手足无措. 幸运的是, php中在OOP之上有一层可以让你选择你的任务所需的部分而不关心其他部分. 找到复杂性之上一个舒适的层级开始工作, 剩下的都会顺起来的.

现在已经看完了所有的php内部数据类型, 是时候回到之前的主题了: 请求生命周期. 接下来的两章, 将在你的扩展中使用线程安全全局变量增加内部状态, 定义自定义的ini设置, 定义常量, 以及向使用你扩展的用户空间脚本提供超级全局变量.

启动, 终止, 以及其中的一些点

在本书中, 你已经多次使用MINIT函数在php加载你扩展的共享库时执行初始化任务. 在第1章"php的生命周期"中, 你还学习了其他三个启动/终止函数, 与MINIT对应的是MSHUTDOWN, 另外还有一对RINIT/RSHUTDOWN方法在每个页面请求启动和终止时被调用.

生命周期

除了这四个直接链接到模块结构的函数外, 还有两个函数仅用于线程环境, 用来处理每个线程的启动和终止, 以及它们使用的似有存储空间.开始之前, 首先将你的php扩展骨架程序拷贝到php源码树的ext/sample4下. 代码如下

config.m4

```
PHP_ARG_ENABLE(sample4,
  [Whether to enable the "sample4" extension],
  [  enable-sample4       Enable "sample4" extension support])

if test $PHP_SAMPLE4 != "no"; then
  PHP_SUBST(SAMPLE4_SHARED_LIBADD)
  PHP_NEW_EXTENSION(sample4, sample4.c, $ext_shared)
fi
```

php_sample4.h

```
#ifndef PHP_SAMPLE4_H
/* Prevent double inclusion */
#define PHP_SAMPLE4_H

/* Define Extension Properties */
#define PHP_SAMPLE4_EXTNAME    "sample4"
#define PHP_SAMPLE4_EXTVER    "1.0"

/* Import configure options
   when building outside of
   the PHP source tree */
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

/* Include PHP Standard Header */
#include "php.h"

/* Define the entry point symbol
 * Zend will use when loading this module
 */
extern zend_module_entry sample4_module_entry;
#define phpext_sample4_ptr &sample4_module_entry

#endif /* PHP_SAMPLE4_H */
```

sample4.c

```
#include "php_sample4.h"
#include "ext/standard/info.h"

static function_entry php_sample4_functions[] = {
    { NULL, NULL, NULL }
};

PHP_MINIT_FUNCTION(sample4)
```

```
{
    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(sample4)
{
    return SUCCESS;
}

PHP_RINIT_FUNCTION(sample4)
{
    return SUCCESS;
}

PHP_RSHUTDOWN_FUNCTION(sample4)
{
    return SUCCESS;
}
PHP_MINFO_FUNCTION(sample4)
{
}

zend_module_entry sample4_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE4_EXTNAME,
    php_sample4_functions,
    PHP_MINIT(sample4),
    PHP_MSHUTDOWN(sample4),
    PHP_RINIT(sample4),
    PHP_RSHUTDOWN(sample4),
    PHP_MINFO(sample4),
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE4_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_SAMPLE4
ZEND_GET_MODULE(sample4)
#endif
```

要注意, 每个启动和终止函数在退出时都返回SUCCESS. 如果这些函数中某个返回FAILURE, 引擎就认为这个过程失败并中断php的执行.

模块生命周期

在前面章节已经多次使用, 因此MINIT对你来说应该已经很熟悉了. 它在模块第一次加载到进程空间时触发, 对于单请求sapi比如CLI和CGI, 或者多线程sapi比如apache2-worker, 它都只执行一次, 因为不涉及到fork.

对于多进程sapi, 比如apache1, apache2-prefork, 通过它们的mod_php实例for了多个webserver进程. 每个mod_php实例都必须加载自己的扩展模块, 因此MINIT将被执行多次, 不过对于每个进程, 它仍然只执行一次.

当模块被卸载时, MSHUTDOWN方法被调用, 此时模块的所有资源(比如持久化内存块)都将被释放, 返回给操作系统.

引擎端的特性, 比如类, 资源ID, 流包装和过滤器, 用户空间全局变量, php.ini中的指令这些公共的资源都是在模块的INIT和SHUTDOWN阶段被分配和释放的.

理论上来说, 你可以不用在*MSHUTDOWN*阶段做资源释放的工作, 把它留给*OS*去做隐式的内存和文件释放. 不过在*apache 1.3*中使用你的扩展时, 你会发现一个有趣的现象, *apache*将加载

mod_php, 在进程中运行*MINIT,* 接着立即卸载*mod_php,* 触发*MSHUTDOWN*方法, 接着再次加载它. 如果没有正确的*MSHUTDOWN*阶段, 在*MINIT*阶段初始分配的资源就将泄露.

线程生命周期

在多线程sapi中, 有时需要为每个线程分配它自己独立的资源, 或跟踪它自己的单请求计数器.对于这些特殊情况, 存在一组每个线程的钩子, 允许在线程启动和终止时执行它们. 典型的情况是当apache2-worker这样的sapi启动时, 它将会产生一打或更多的线程去处理并发请求.

任何在多请求间共享, 在同一进程中不同线程有不能访问的资源, 都是在线程的构造器和析构器中分配和释放的. 比如这可能包括EG(persistent_list)HashTable中的持久化资源, 因为它们通常包括网络或文件资源, 需要考虑指令间它们的状态一致性.

请求生命周期

最后一个也是最短的生命周期是请求生命周期, 在这个周期内, 你的扩展可能会去初始化默认的用户空间变量, 或初始化内部状态跟踪信息. 因为这些方法在每个页面请求都被调用, 因此要尽可能的保证这些处理和内存分配可以执行的足够快.

通过MINFO对外暴露模块信息

除非你计划只有很少人使用你的扩展, 并且并没有计划修改API, 否则你就需要能够告诉用户空间一些关于扩展自身的信息. 比如, 是否所有的环境和版本特有特性都可用? 它编译的外部库的版本是什么? 是否有网站或邮件地址可以让你扩展的用户在需要时寻求帮助?

如果你曾经看过phpinfo()或php -i的输出, 你就会注意到, 所有这些信息都被组织到一种良好格式, 易于解析的输出中. 你的扩展可以很简单的在这些内容中增加自己的信息, 只需要在你的模块中增加一个MINFO()函数即可:

```
PHP_MINFO_FUNCTION(sample4)
{
    php_info_print_table_start();
    php_info_print_table_row(2, "Sample4 Module", "enabled");
    php_info_print_table_row(2, "version", PHP_SAMPLE4_EXTVER);
    php_info_print_table_end();
}
```

通过使用这些包装函数, 你的模块信息将在从webserver sapi(比如cgi, iis, apache等)输出时自动的包装为HTML标签, 而在使用cli时输出为普通文本. 为了使得构建时你的扩展中可以使用这些函数原型, 你需要#include "ext/standard/info.h".

下面是这个头文件中可用的php_info_*()族函数.

- char *php_info_html_esc(char *str TSRMLS_DC)

用户空间htmlentites()函数的底层实现php_escape_html_entities()的一个包装. 返回的字符串是用emalloc()分配的, 使用后必须显式的使用efree()释放.

- void php_info_print_table_start(void)
- void php_info_print_table_end(void)

输出html表格的开始/结束标签. html输出禁用时, 比如在cli中, 它将在start中输出换行符, end中不输出任何内容.

- void php_info_print_table_header(int cols, ...)
- void php_info_print_table_colspan_header(int cols, char *header)

输出一行表头. 第一个版本为每个可变参输出一个<th></th>, 内容是cols后面的字符串参数. clospan版的则只输出一个<th></th>, 并给它指定colspan属性.

- void php_info_print_table_row(int cols, ...)

- void php_info_print_table_row_ex(int cols, char *class, ...)

这两个版本都为每个可变参输出一个<td></td>. 两者的不同在于前者将为其设置class="v"属性, 而后者则允许调用者指定自己的类名用于自定义格式. 没有打开HTML格式输出时, 由于只是文本输出, 两者的差异就不复存在了.

- void php_info_print_box_start(int flag)
- void php_info_print_box_end()

这两个函数只是简单的输出一个表格(<tr class="h"><td>, </td></tr>)的开始和结束. 如果给定的flag值非0, 则使用class="h", 否则使用class="v". 使用非html输出时, 标记为0将导致在star中输出一个换行符, 此时这两个函数不会在产生其他任何输出.

- void php_info_print_hr(void)

这个函数在html启用时输出<hr />标签, 或者, 当没有启用html输出时, 输出31个下划线, 并在前后各输出两个换行符.

在MINFO中通常可以使用PHPWRITE()和php_printf(), 但手动输出内容时应该注意它需要依赖于当前的SAPI期望输出文本还是html. 可以通过测试全局的sapi_module结构体的phpinfo_as_text属性来确认这一点:

```
PHP_MINFO_FUNCTION(sample4)
{
    php_info_print_table_start();
    php_info_print_table_row(2, "Sample4 Module", "enabled");
    php_info_print_table_row(2, "version", PHP_SAMPLE4_EXTVER);
    if (sapi_module.phpinfo_as_text) {
        /* No HTML for you */
        php_info_print_table_row(2, "By",
            "Example Technologies\nhttp://www.example.com");
    } else {
        /* HTMLified version */
        php_printf("<tr>"
            "<td class=\"v\">By</td>"
            "<td class=\"v\">"
            "<a href=\"http://www.example.com\""
            " alt=\"Example Technologies\">"
            "<img src=\"http://www.example.com/logo.png\" />"

            "</a></td></tr>");
    }
    php_info_print_table_end();
}
```

常量

向用户空间脚本暴露信息更好的方法是使用扩展定义脚本可以在运行时访问的常量, 并可以通过这些常量改变扩展的某些行为. 在用户空间中, 我们使用define()函数定义常量; 内部, 则是和它非常相似的REGISTER_*_CONSTANT()一族的宏.

多数常量是你想要它们在所有脚本中初始化为相同值的数据. 它们是在MINIT函数中定义的.

```
PHP_MINIT_FUNCTION(sample4)
{
    REGISTER_STRING_CONSTANT("SAMPLE4_VERSION",
            PHP_SAMPLE4_EXTVER, CONST_CS | CONST_PERSISTENT);

    return SUCCESS;
}
```

这个宏的第一个参数是要暴露给用户空间的常量名. 在这个例子中, 用户空间就可以执行echo SAMPLE4_VERSION; 得到输出1.0. 这里有一点要特别注意, REGISTER_*_CONSTANT()一族的宏使用了sizeof()调用去确定常量名的长度. 也就是说

只能使用字面量值. 如果使用char *变量则会导致不正确的结果(sizeof(char *)在32位平台上通常是4, 而不是真正字符串的长度).

下一个参数是常量的值. 多数情况下, 它只需要一个参数, 不过, 对于STRINGL版本, 你还需要一个参数去指定长度. 在注册字符串常量时, 字符串的值并不会拷贝到常量中, 只是引用它. 也就是说需要在持久化内存中为其分配空间, 并在对应的SHUTDOWN阶段释放它们.

最后一个参数是一个有两个可选值的位域操作结果. CONST_CS标记说明该常量大小写敏感. 对于用户空间定义的常量以及几乎所有的内部常量来说, 这都是默认行为. 只有极少数的情况, 比如trUE, FALSE, NULL, 在注册时省略了这个标记用以说明它们是不区分大小写的.

注册常量时的第二个标记是持久化标记. 当在MINIT中定义常量时, 它们必须被构建为跨请求的持久化常量. 但是, 如果在请求中定义常量, 比如在RINIT中, 你可能就需要省略这个标记以允许引擎在请求结束时销毁该常量了.

下面是4个可用的常量注册宏的原型. 一定要记住, 名字参数必须是字符串字面量而不能是char *变量:

```
REGISTER_LONG_CONSTANT(char *name, long lval, int flags)
REGISTER_DOUBLE_CONSTANT(char *name, double dval, int flags)
REGISTER_STRING_CONSTANT(char *name, char *value, int flags)
REGISTER_STRINGL_CONSTANT(char *name,
                    char *value, int value_len, int flags)
```

如果字符串必须从变量名初始化, 比如在循环中, 你可以使用如下的函数调用(上面的宏就是映射到这些函数中的):

```
void zend_register_long_constant(char *name, uint name_len,
        long lval, int flags, int module_number TSRMLS_DC)
void zend_register_double_constant(char *name, uint name_len,
        double dval, int flags, int module_number TSRMLS_DC)
void zend_register_string_constant(char *name, uint name_len,
        char *strval, int flags, int module_number TSRMLS_DC)
void zend_register_stringl_constant(char *name, uint name_len,
        char *strval, uint strlen, int flags,
        int module_number TSRMLS_DC)
```

此时, 名字参数的长度可以直接由调用作用域提供. 你应该注意到, 这次就必须显式的传递TSRMLS_CC参数了, 并且, 这里还引入了另外一个参数.

module_number是在你的扩展被加载或被卸载时传递给你的信息. 你不用关心它的值, 只需要传递它就可以了. 在MINIT和RINIT函数原型中都提供了它, 因此, 在你定义常量的时候, 它就是可用的. 下面是函数版的常量注册例子:

```
PHP_MINIT_FUNCTION(sample4)
{
    register_string_constant("SAMPLE4_VERSION",
        sizeof("SAMPLE4_VERSION"),
        PHP_SAMPLE4_EXTVER,
        CONST_CS | CONST_PERSISTENT,
        module_number TSRMLS_CC);

    return SUCCESS;
}
```

要注意当sizeof()用于确定SAMPLE4_VERSION的长度时, 这里并没有减1. 常量的名字是包含它的终止NULL的. 如果使用strlen()确定长度, 要记得给结果加1以使其包含终止的NULL.

除了数组和对象, 其他的类型都可以被注册, 但是因为在ZEND API中不存在这些类型的宏或函数, 你就需要手动的定义常量. 按照下面的范本, 仅需要在使用时修改去创建恰当类型的zval *即可:

```
void php_sample4_register_boolean_constant(char *name, uint len,
```

```
    zend_bool bval, int flags, int module_number TSRMLS_DC)
{
    zend_constant c;

    ZVAL_BOOL(&c.value, bval);
    c.flags = CONST_CS | CONST_PERSISTENT;
    c.name = zend_strndup(name, len - 1);
    c.name_len = len;
    c.module_number = module_number;
    zend_register_constant(&c TSRMLS_CC);
}
```

扩展的全局空间

如果可以保证任何时刻一个进程中只有一个php脚本在执行, 你的扩展就可以随意的定义全局变量并去访问它们, 因为已知在opcode执行过程中不会有其他脚本被执行. 对于非线程sapi, 这是可行的, 因为所有的进程空间中都只能同时执行一个代码路径.

然而在线程sapi中, 可能会有两个或更多的线程同时读或更糟糕的情况是同时写相同的值. 为了解决这个问题, 就引入了一个扩展的全局空间概念, 它为每个扩展的数据提供一个唯一的数据存储桶.

定义扩展的全局空间

要给你的扩展申请一块存储的桶, 首先就需要在php_sample4.h上的一个标准结构体中定义所有你的全局变量. 比如, 假设你的扩展要保存一个计数器, 保持对某个方法在请求内被调用次数的跟踪, 你就需要定义一个结构体包含一个unsigned long:

```
ZEND_BEGIN_MODULE_GLOBALS(sample4)
    unsigned long counter;
ZEND_END_MODULE_GLOBALS(sample4)
```

ZEND_BEGIN_MODULE_GLOBALS和ZEND_END_MODULE_GLOBALS宏为扩展全局变量结构的定义提供了统一的框架. 如果你看过这个块的展开形式, 就可以很容易的理解它了:

```
typedef struct _zend_sample4_globals {
    unsigned long counter;
} zend_sample4_globals;
```

你可以像在其他的C语言结构体中增加成员一样, 为它增加其他成员. 现在, 你有了存储桶的(数据结构)定义, 接下来要做的就是声明一个这个类型的变量, 你需要在扩展的sample4.c文件中, #include "php_sample4.h"语句下一行声明它:

```
ZEND_DECLARE_MODULE_GLOBALS(sample4);
```

它将根据是否启用了线程安全, 被解析为两种不同的格式. 对于非线程安全构建, 比如apache1, apache2-prefork, cgi, cli以等等, 它是直接在真正的全局作用域声明了一个zend_sample4_globals结构体的直接值:

```
zend_sample4_globals sample4_globals;
```

这和你在其他单线程应用中声明的全局变量没有什么差异. 计数器的值直接通过sample4_globals.counter访问. 而对于线程安全构建, 则是另外一种处理, 它只是声明了一个整型值, 以后它将扮演到真实数据的引用的角色:

```
int sample4_globals_id;
```

设置这个ID就代表声明你的扩展全局变量到引擎中. 通过提供的信息, 引擎将在每个新的线程产生时分配一块内存 专门用于线程服务请求时的似有存储空间. 在你的MINIT函数中增加下面的代码块:

```
#ifdef ZTS
    ts_allocate_id(&sample4_globals_id,
                sizeof(zend_sample4_globals),
                NULL, NULL);
#endif
```

注意, 这个语句被包裹在一个ifdef中, 以放置在没有启用Zend线程安全(ZTS)时执行它. 这是因为sample4_globals_id只在线程环境下才会被声明, 非线程环境的构建则使用的是sample4_globals变量的直接值.

每个线程的初始化和终止

在非线程构建中, 你的zend_sample4_globals结构体在一个进程中只有一份拷贝. 你可以给它设置初始值或在MINIT或RINIT中为其分配资源, 进行初始化, 在MSHUTDOWN和RSHUTDOWN阶段如果需要, 则进行相应的释放.

然而, 对于线程构建, 每次一个新的线程产生时, 都会分配一个新的结构体. 实际上, 这在webserver启动时可能会发生很多次, 而在webserver进程的整个生命周期中, 这可能会发生成百上千次. 为了知道怎样初始化和终止你的扩展全局空间, 引擎需要执行一些回调函数. 这就是上面的例子中你传递给ts_allocate_id()的NULL参数; 在你的MINIT函数上面增加下面的两个函数:

```
static void php_sample4_globals_ctor(
            zend_sample4_globals *sample4_globals TSRMLS_DC)
{
    /* 在线程产生时初始化一个新的zend_sample4_globals结构体 */
    sample4_globals->counter = 0;
}
static void php_sample4_globals_dtor(
            zend_sample4_globals *sample4_globals TSRMLS_DC)
{
    /* 在初始化阶段分配的各种资源, 都在这里释放 */
}
```

接着, 在启动和终止时使用这些函数:

```
PHP_MINIT_FUNCTION(sample4)
{
    REGISTER_STRING_CONSTANT("SAMPLE4_VERSION",
            PHP_SAMPLE4_EXTVER, CONST_CS | CONST_PERSISTENT);
#ifdef ZTS
    ts_allocate_id(&sample4_globals_id,
                sizeof(zend_sample4_globals),
                (ts_allocate_ctor)php_sample4_globals_ctor,
                (ts_allocate_dtor)php_sample4_globals_dtor);
#else
    php_sample4_globals_ctor(&sample4_globals TSRMLS_CC);
#endif
    return SUCCESS;
}
PHP_MSHUTDOWN_FUNCTION(sample4)
{
#ifndef ZTS
    php_sample4_globals_dtor(&sample4_globals TSRMLS_CC);
#endif
    return SUCCESS;
}
```

要注意, 在没有开启ZTS时, ctor和dtor函数是手动调用的. 不要忘记: 非线程环境也需要初始化和终止.

你可能奇怪为什么在*php_sample4_globals_ctor()*和*php_sample4_globals_dtor()*中直接使用了*TSRMLS_CC*. 如果你认为"这完全不需要, 它在*ZTS*禁用时解析出来是空的内容, 并且由*#ifndef*指令, 我们知道*ZTS*是被禁用的, 你的观点绝对正确. 声明中的相关的*TSRMLS_DC*指令仅用于保证代码的一致性. 从积极的一面考虑, 如果*ZEND API*修改这些值使得在非*ZTS*构建中也有有效内容时, 你的代码就不需要修改就做好了相应的调整.

访问扩展的全局空间

现在你的扩展有了一个全局变量集合, 你可以开始在你的代码中访问它们了. 在非ZTS模式中这很简单, 只需要访问进程全局作用域的sample4_globals变量的相关成员即可, 比如, 下面的用户空间函数增加了你前面定义的计数器并返回它的当前值:

```
PHP_FUNCTION(sample4_counter)
{
    RETURN_LONG(++sample4_globals.counter);
}
```

很简单很容易. 不幸的是, 这种方式在线程环境的PHP构建中不能工作. 这种情况下你就需要做更多的工作. 下面是使用ZTS语义的该函数返回语句:

```
RETURN_LONG(++TSRMG(sample4_globals_id,
               zend_sample4_globals*, counter));
```

TSRMG()宏需要你已经传递的TSRMLS_CC参数, 它会从当前线程池的资源结构中查找需要的数据. 这里, 它使用sample4_globals_id索引映射到内存池中你扩展的全局结构体的位置, 最终, 使用数据类型映射的元素名得到结构体中的偏移量. 因为你并不知道运行时你的扩展是否使用ZTS模式, 因此, 你需要让你的代码适应两种情况. 要做到这一点, 就需要按照下面方式重写该函数:

```
PHP_FUNCTION(sample4_counter)
{
#ifdef ZTS
    RETURN_LONG(++TSRMG(sample4_globals_id, \
                    zend_sample4_globals*, counter));
#else /* non-ZTS */
    RETURN_LONG(++sample4_globals.counter);
#endif
}
```

看起来不舒服? 是的, 如果你所有的代码都基于这样的ifdef指令去处理线程安全的全局访问, 它看起来可能比Perl还糟糕! 这就是为什么在所有的PECL扩展中都使用了一个抽象的宏来封装全局访问的原因. 在你的php_sample4.h文件中进行如下定义:

```
#ifdef ZTS
#include "TSRM.h"
#define SAMPLE4_G(v)    TSRMG(sample4_globals_id,
                         zend_sample4_globals*, v)
#else
#define SAMPLE4_G(v)    (sample4_globals.v)
#endif
```

这样, 就可以让你访问扩展全局空间时变得简单易懂:

```
PHP_FUNCTION(sample4_counter)
{
    RETURN_LONG(++SAMPLE4_G(counter));
}
```

这个宏给你一种似曾相识的感觉吗? 应该是这样的. 它和你已经使用过的EG(symbol_table)以及EG(active_symbol_table)是仙童的概念和实践. 在阅读php源码树中其他部分以及其他扩展时, 你会经常碰到这种宏. 下表列出了常用的全局访问宏:

访问宏	关联数据
EG()	执行全局空间. 这个结构体主要用于引擎内部对当前请求的状态跟踪. 这个全局空间中可以找到符号表, 函数表, 类表, 常量表, 资源表等.

访问宏	关联数据
CG()	核心全局空间. 主要被Zend引擎在脚本编译和内核底层执行过程中使用. 在你的扩展中一般很少直接测试这些值.
PG()	php全局空间. 多数"核心的"php.ini指令映射到php全局变量结构体中的一个或多个元素. 比如: PG(register_globals), PG(safe_mode)以及PG(memory_limit)
FG()	文件全局空间. 多数文件I/O或流相关的全局变量被装入到这个结构通过标准扩展暴露.

用户空间超级全局变量

用户空间有它自己的完全无关的全局概念. 在用户空间, 有一种特殊的全局变量被称为超级全局变量. 这种特殊的用户空间变量包括$_GET, $_POST, $_FILE等等, 在全局作用域, 函数或方法内部都可以等同本地作用域进行访问.

这是由于超级全局变量的解析方式造成的, 它们必须在脚本编译之前定义. 这就意味着在普通的脚本中不能定义其他超级全局变量. 不过, 在扩展中, 可以在请求接收到之前去将变量名定义为超级全局变量.

扩展定义超级全局变量的一个基本示例是ext/session, 它在session_start()和session_write_close()或脚本结束之间, 使用$_SESSION超级全局变量存储会话信息. 为了将$_SESSION定义为超级全局变量, session扩展在MINIT函数中执行了一次下面的语句:

```
PHP_MINIT_FUNCTION(session)
{
    zend_register_auto_global("_SESSION",
                         sizeof("_SESSION") - 1,
                         NULL TSRMLS_CC);
    return SUCCESS;
}
```

注意, 第二个参数, 变量名的长度, 使用了sizeof() - 1, 因此不包含终止NULL. 这和之前你看到的多数内部调用不同, 因此, 在定义自己的超级全局变量时要格外小心这一点.

zend_register_auto_global()函数在Zend引擎2中的原型如下:

```
int zend_register_auto_global(char *name, uint name_len,
    zend_auto_global_callback auto_global_callback TSRMLS_DC)
```

在Zend引擎1中, auto_global_callback参数并不存在. 为了让你的扩展兼容php4, 就需要在MINIT函数中通过#ifdef块去选择性的执行不同的调用, 定义$_SAMPLE4超级全局变量.

```
PHP_MINIT_FUNCTION(sample4)
{
    zend_register_auto_global("_SAMPLE4", sizeof("_SAMPLE4") - 1
#ifdef ZEND_ENGINE_2
                         , NULL
#endif
                         TSRMLS_CC);
    return SUCCESS;
}
```

自动全局回调

ZE2中zend_register_auto_global()的auto_global_callback参数是一个指向自定义函数的指针, 该函数在编译阶段用户空间脚本碰到你的超级全局变量时将被触发. 实际上, 它可以用于在当前脚本没有访问超级全局变量时避免繁杂的初始化处理. 考虑下面的代码:

```
zend_bool php_sample4_autoglobal_callback(char *name,
                           uint name_len TSRMLS_DC)
{
    zval *sample4_val;
    int i;

    MAKE_STD_ZVAL(sample4_val);
    array_init(sample4_val);
    for(i = 0; i < 10000; i++) {
        add_next_index_long(sample4_val, i);
    }
    ZEND_SET_SYMBOL(&EG(symbol_table), "_SAMPLE4",
                                    sample4_val);
    return 0;
}
PHP_MINIT_FUNCTION(sample4)
{
    zend_register_auto_global("_SAMPLE4", sizeof("_SAMPLE4") - 1
#ifdef ZEND_ENGINE_2
                         , php_sample4_autoglobal_callback
#endif
                         TSRMLS_CC);
    return SUCCESS;
}
```

php_sample4_autoglobal_callback()所做的工作代表的是对内存和CPU时间的耗费, 如果$_SAMPLE4没有被访问, 则这些资源都将被浪费. 在Zend引擎2中, 只有当脚本被编译时发现某个地方访问了$_SAMPLE4才会调用php_sample4_autoglobal_callback()函数. 注意, 一旦数组初始化完成并增加到请求的符号表后, 函数就返回0值. 这样就解除了请求中后续对该超级全局变量访问时的回调, 以确保对$_SAMPLE4的多次访问不会导致对该回调函数的多次调用. 如果你的扩展需要在每次碰到该超级全局变量时都执行回调函数, 只需要让回调函数返回真值(非0)使得超级全局变量回调函数不被解除即可.

不幸的是, 现在的设计和php4/zend引擎1冲突, 因为旧的引擎并不支持自动全局回调. 这种情况下, 你就需要在每次脚本启动时, 无论是否使用了变量都去初始化. 要这样做, 直接在RINIT函数中调用你上面编写的回调函数即可:

```
PHP_RINIT_FUNCTION(sample4)
{
#ifndef ZEND_ENGINE_2
    php_sample4_autoglobal_callback("_SAMPLE4",
                              sizeof("_SAMPLE4") - 1,
                              TSRMLS_CC);
#endif
    return SUCCESS;
}
```

小结

通过本章的学习, 你认识了一些新的但是已经熟悉的概念, 包括内部的线程安全全局变量, 怎样向用户空间暴露诸如常量, 预初始化变量, 超级全局变量等信息. 下一章, 你将学会怎样定义和解析php.ini中的指令, 并将它们和你已经设置的内部线程安全的全局结构关联起来.

INI设置

和上一章你看到的超级全局变量以及持久化常量一样, php.ini值必须在扩展的MINIT代码块中定义. 然而, 和其他特性不同的是, INI选项的定义仅仅由简单的启动/终止线组成.

```
PHP_MINIT_FUNCTION(sample4)
{
    REGISTER_INI_ENTRIES();
    return SUCCESS;
}
PHP_MSHUTDOWN_FUNCTION(sample4)
{
    UNREGISTER_INI_ENTRIES();
    return SUCCESS;
}
```

定义并访问INI设置

INI指令自身是在源码文件中MINIT函数上面, 使用下面的宏完全独立的定义的, 在这两个宏之间可以定义一个或多个INI指令:

```
PHP_INI_BEIGN()
PHP_INI_END()
```

这两个宏函数和ZEND_BEGIN_MODULE_GLOBALS()/ ZEND_END_MODULE_GLOBALS()异曲同工. 不过这里不是typdef一个结构体, 而是对静态数据实例定义的框架组织:

```
static zend_ini_entry ini_entries[] = {
{0,0,NULL,0,NULL,NULL,NULL,NULL,NULL,0,NULL,0,0,NULL} };
```

如你所见, 它定义了一个zend_ini_entry值的向量, 以一条空的记录结束. 这和你在前面看到的静态向量function_entry的定义一致.

简单的INI设置

现在, 你已经有一个INI结构体用于定义INI指令, 以及引擎注册/卸载INI设置的机制, 因此我们可以真正的去为你的扩展定义一些INI指令了. 假设你的扩展暴露了一个打招呼的函数, 就像第5章"你的第一个扩展"中一样, 不过, 你想让打招呼的话可以自定义:

```
PHP_FUNCTION(sample4_hello_world)
{
    php_printf("Hello World!\n");
}
```

最简单最直接的方式就是定义一个INI指令, 并给它一个默认值"Hello world!":

```
#include "php_ini.h"
PHP_INI_BEGIN()
    PHP_INI_ENTRY("sample4.greeting", "Hello World",
                                    PHP_INI_ALL, NULL)
PHP_INI_END()
```

你可能已经猜到了, 这个宏的前两个参数表示INI指令的名字和它的默认值. 第三个参数用来确定引擎是否允许这个INI指令被修改(这将涉及到本章后面要介绍的访问级别问题). 最后一个参数是一个回调函数, 它将在每次INI指令的值发生变化时被调用. 你将在修改事件一节看到这个参数的细节.

译注: 如果你和译者一样遇到结果和原著结果预期不一致时, 请在测试时, 在你的*MINIT()*函数中增加一句*"REGISTER_INI_ENTRIES();"*调用, 并确保该调用在你的*MINIT*中分配全局空间之后执行.

现在你的INI设置已经定义, 只需要在你的打招呼函数中使用就可以了.

```
PHP_FUNCTION(sample4_hello_world)
{
    const char *greeting = INI_STR("sample4.greeting");
```

```
    php_printf("%s\n", greeting);
}
```

一定要注意, char *的值是引擎所有的, 一定不要修改. 正因为这样, 所以将你本地用来临时存储INI设置值的变量定义为const修饰. 当然, 并不是所有的INI值都是字符串; 还有其他的宏用来获取整型, 浮点型以及布尔型的值:

```
long lval = INI_INT("sample4.intval");
double dval = INI_FLT("sample4.fltval");
zend_bool bval = INI_BOOL("sample4.boolval");
```

通常你想要知道的是INI设置的当前值; 不过, 作为补充, 存在几个宏可以用来读取未经修改的INI设置值:

```
const char *strval = INI_ORIG_STR("sample4.stringval");
long lval = INI_ORIG_INT("sample4.intval");
double dval = INI_ORIG_FLT("sample4.fltval");
zend_bool bval = INI_ORIG_BOOL("sample4.boolval");
```

这个例子中, *INI*指令的名字*"sample4.greeting"*增加了扩展名作为前缀, 这样来保证不会和其他扩展暴露的*INI*指令名字冲突. 对于私有的扩展来说, 这个前缀不是必须的, 但是对于商业化或开源发布的公开扩展还是鼓励这样做的.

访问级别

对于INI指令, 开始总是有一个默认值. 多数情况下, 理想是保持默认值不变; 然而, 对于某些特殊的环境或者脚本内特定的动作, 这些值可能需要被修改. 如下表所示, INI指令的值可能在下面3个点被修改:

访问级别	含义
SYSTEM	位于php.ini中, 或者apache的httpd.conf配置文件中<Directory>和<VirtualHost>指令外部, 影响引擎的启动阶段, 可以认为是INI设置的"全局"值.
PERDIR	位于Apache的httpd.conf配置文件中<Directory>和<VirtualHost>指令中, 或者请求脚本所在目录或虚拟主机下的.htaccess文件以及其他apache在处理请求之前其他地方设置的INI指令.
USER	一旦脚本开始执行, 就只能通过调用用户空间函数ini_set()去修改INI设置了.

某些设置如果可以在任何地方被修改就没有多大意义了, 比如safe_mode, 如果可以在任何地方去修改, 那么恶意脚本的作者就可以很简单的去禁用safe_mode, 接着去读或修改本不允许操作的文件.

类似的, 某些非安全相关的指令比如register_globals或magic_quotes_gpc, 在脚本中不能被修改, 因为, 在脚本执行时, 它所影响的事情已经发生过了.

这些指令的访问控制是通过PHP_INI_ENTRY()的第三个参数完成的. 在你前面例子中, 使用了PHP_INI_ALL, 它的定义是一个位域操作: PHP_INI_SYSTEM | PHP_INI_PERDIR | PHP_INI_USER.

对于register_globals和magic_quotes_gpc这样的指令, 定义的访问级别为PHP_INI_SYSTEM | PHP_INI_PERDIR. 排除了PHP_INI_USER将导致以这个名字调用ini_set()时最终会失败.

现在, 你可能已经猜到, safe_mode和open_basedir这样的指令应该仅被定义为PHP_INI_SYSTEM. 这样的设置就确保了只有系统管理员可以修改这些值, 因为只有它们可以访问修改php.ini或httpd.conf文件中的配置.

修改事件

当INI指令被修改时, 无论是通过ini_set()函数还是某个perdir指令的处理, 引擎都会为其测试OnModify回调. 修改处理器可以使用ZEND_INI_MH()宏定义, 并通过在OnModify参数上传递函数名附加到INI指令上:

```
ZEND_INI_MH(php_sample4_modify_greeting)
{
    if (new_value_length == 0) {
        return FAILURE;
    }
    return SUCCESS;
}
PHP_INI_BEGIN()
    PHP_INI_ENTRY("sample4.greeting", "Hello World",
            PHP_INI_ALL, php_sample4_modify_greeting)
PHP_INI_END()
```

通过在new_value_length为0时返回FAILURE, 这个修改处理器禁止将greeting设置为空字符串. ZEND_INI_MH()宏产生的整个原型如下:

```
int php_sample4_modify_greeting(zend_ini_entry *entry,
    char *new_value, uint new_value_length,
    void *mh_arg1, void *mh_arg2, void *mh_arg3,
    int stage TSRMLS_DC);
```

各个参数的含义见下表:

参数名	含义
entry	指向引擎真实存储的INI指令项. 这个结构体提供了当前值, 原始值, 所属模块, 以及其他一些下面代码(zend_ini_entry结构体结构)列出的信息
new_value	要被设置的值. 如果处理器返回SUCCESS, 这个值将被设置到entry->value, 同时如果entry->orig_value当前没有设置, 则将当前值设置到entry->orig_value中, 并设置entry->modified标记. 这个字符串的长度通过new_value_length传递.
mh_arg1, 2, 3	这3个指针对应INI指令定义时给出的数据指针(zend_ini_entry结构体中的3个同名成员). 实际上, 这几个值是引擎内部处理使用的, 你不需要关心它们.
stage	ZEND_INI_STAGE_系列的5个值之一: STARTUP, SHUTDOWN, ACTIVATE, DEACTIVATE, RUNTIME. 这些常量对应于MINIT, MSHUTDOWN, RINIT, RSHUTDOWN, 以及活动脚本执行.

核心结构体: zend_ini_entry

```
struct _zend_ini_entry {
    int module_number;
```

```
    int modifiable;
    char *name;
    uint name_length;
    ZEND_INI_MH((*on_modify));
    void *mh_arg1;
    void *mh_arg2;
    void *mh_arg3;

    char *value;
    uint value_length;

    char *orig_value;
    uint orig_value_length;
    int modified;

    void ZEND_INI_DISP(*displayer);
};
```

展示INI设置

在上一章, 你看到了MINFO函数以及相关的指令用于展示扩展的信息. 由于扩展暴露INI指令是很常见的, 因此引擎提供了一个公共的宏可以放置到PHP_MINFO_FUNCTION()中用于展示INI指令信息.

```
PHP_MINFO_FUNCTION(sample4)
{
    DISPLAY_INI_ENTRIES();
}
```

这个宏将迭代PHP_INI_BEGIN()和PHP_INI_END()宏之间定义的INI指令集和, 在一个3列的表格中展示它们的INI指令名, 原始值(全局的), 以及当前值(经过PERDIR指令或ini_set()调用修改后)

默认情况下, 所有的指令都直接以其字符串形式输出. 对于某些指令, 比如布尔值以及用于语法高亮的颜色值, 则在展示处理时应用了其他格式. 这些格式是通过每个INI设置的显示处理器处理的, 它和你看到的OnModify一样是一个动态的回调函数指针.

显示处理器可以使用PHP_INI_ENTRY()宏的扩展版指定, 它接受一个额外的参数. 如果设置为NULL, 则使用展示字符串值的处理器作为默认处理器:

```
PHP_INI_ENTRY_EX("sample4.greeting", "Hello World", PHP_INI_ALL,
    php_sample4_modify_greeting, php_sample4_display_greeting)
```

显然, 需要在INI设置定义之前声明这个函数. 和OnModify回调函数一样, 这可以通过一个包装宏以及少量编码完成:

```
#include "SAPI.h" /* needed for sapi_module */
PHP_INI_DISP(php_sample4_display_greeting)
{
    const char *value = ini_entry->value;

    /* 选择合适的当前值或原始值 */
    if (type == ZEND_INI_DISPLAY_ORIG &&
        ini_entry->modified) {
        value = ini_entry->orig_value;
    }

    /* 使得打招呼的字符串粗体显示(当以HTML方式输出时) */
    if (sapi_module.phpinfo_as_text) {
        php_printf("%s", value);
    } else {
        php_printf("<b>%s</b>", value);
    }
}
```

绑定到扩展的全局空间

所有的INI指令都在Zend引擎内有一块存储空间, 可以用以跟踪脚本内的变更并进行请求外部的全局设置维护. 在这块存储空间中, 所有的INI指令都以字符串值存储. 你已经知道了, 这些值可以使用INI_INT(), INI_FLT(), INI_BOOL()等宏函数, 很简单的翻译成其他的标量类型.

这个查找和转换过程由于两个原因非常低效: 首先, 每次一个INI的值在获取时, 它必须通过名字在一个HashTable中进行定位. 这种查找方式对于仅在运行时编译的用户空间脚本而言是没有问题的, 但是对于已编译的机器代码源, 运行时做这个工作就毫无意义.

每次请求标量值的时候都需要将底层的字符串值转换到标量值是非常低效的. 因此我们使用你已经学习过的线程安全全局空间作为存储媒介, 每次INI指令值变更时更新它即可. 这样, 所有访问INI指令的代码都只需要查找你的线程安全全局空间结构体中的某个指针即可, 这样就获得了编译期优化的优点.

在你的php_sample4.h文件MODULE_GLOBALS结构体中增加const char *greeting; 接着更新sample4.c中的下面两个方法:

```
ZEND_INI_MH(php_sample4_modify_greeting)
{
    /* Disallow empty greetings */
    if (new_value_length == 0) {
        return FAILURE;
    }
    SAMPLE4_G(greeting) = new_value;
    return SUCCESS;
}
PHP_FUNCTION(sample4_hello_world)
{
    php_printf("%s\n", SAMPLE4_G(greeting));
}
```

由于这是对INI访问的一种非常常见的优化方式, 因此引擎暴露了一组专门处理INI指令到全局变量的绑定宏:

```
STD_PHP_INI_ENTRY_EX("sample4.greeting", "Hello World",
    PHP_INI_ALL, OnUpdateStringUnempty, greeting,
    zend_sample4_globals, sample4_globals,
    php_sample4_display_greeting)
```

这个宏执行和上面你自己的php_sample4_modify_greeting相同的工作, 但它不需要OnModify回调. 取而代之的是, 它使用了一个泛化的修改回调OnUpdateStringUnempty, 以及信息应该存储的空间. 如果要允许空的greeting指令值, 你可以直接指定OnUpdateString替代OnUpdateStringUnempty.

类似的, INI指令也可以绑定long, double, zend_bool的标量值. 在你的php_sample4.h中MODULE_GLOBALS结构体上增加几个字段:

```
long mylong;
double mydouble;
zend_bool mybool;
```

现在在你的PHP_INI_BEGIN()/PHP_INI_END()代码块中使用STD_PHP_INI_ENTRY()宏创建新的INI指令, 它和对应的_EX版本的宏的区别只是显示处理器以及绑定到的值不同.

```
?STD_PHP_INI_ENTRY("sample4.longval", "123",
    PHP_INI_ALL, OnUpdateLong, mylong,
    zend_sample4_globals, sample4_globals)
STD_PHP_INI_ENTRY("sample4.doubleval", "123.456",
    PHP_INI_ALL, OnUpdateDouble, mydouble,
    zend_sample4_globals, sample4_globals)
STD_PHP_INI_ENTRY("sample4.boolval", "1",
    PHP_INI_ALL, OnUpdateBool, mybool,
```

```
    zend_sample4_globals, sample4_globals)
```

这里要注意, 如果调用了DISPLAY_INI_ENTRIES(), 布尔类型的INI指令"sample4.boolval"将和其他设置一样, 被显示为它的字符串值; 然而, 首选的布尔值指令应该被显示为"on"或"off". 要使用这些更加表意的显示, 你可以使用STD_PHP_INI_ENTRY_EX()宏并创建显示处理器, 或者使用另外一个宏:

```
STD_PHP_INI_BOOLEAN("sample4.boolval", "1",
    PHP_INI_ALL, OnUpdateBool, mybool,
    zend_sample4_globals *, sample4_globals)
```

这个特定类型的宏是布尔类型特有的, 它提供的是将布尔值转换为"on"/"off"值的显示处理器.

小结

在本章, 你了解了php语言中最古老的特性之一的实现, 它也是阻碍php可移植的罪魁. 对于每个新的INI设置, 都会使得编写可移植代码变得更加复杂. 使用这些特性要非常慎重, 因为扩展以后时钟都要使用它了. 并且, 在使用时要注意不同系统间的行为一致性, 以免在维护时出现不可预期的状况.

接下来的三张, 我们将深入到流API, 开始使用流的实现层和包装操作, 上下文, 过滤器等.

访问流

PHP用户空间中所有的文件I/O处理都是通过php 4.3引入的php流包装层处理的. 在内部, 扩展代码可以选择使用stdio或posix文件处理和本地文件系统或伯克利域套接字进行通信, 或者也可以调用和用户空间流I/O相同的API.

流的概览

通常, 直接的文件描述符相比调用流包装层消耗更少的CPU和内存; 然而, 这样会将实现某个特定协议的所有工作都堆积到作为扩展开发者的你身上. 通过挂钩到流包装层, 你的扩展代码可以透明的使用各种内建的流包装, 比如HTTP, FTP, 以及它们对应的SSL版本, 另外还有gzip和bzip2压缩包装. 通过include特定的PEAR或PECL模块, 你的代码还可以访问其他协议, 比如SSH2, WebDav, 甚至是Gopher!

本章将介绍内部基于流工作的基础API. 后面到第16章"有趣的流"中, 我们将看到诸如应用过滤器, 使用上下文选项和参数等高级概念.

打开流

尽管是一个统一的API, 但实际上依赖于所需的流的类型, 有四种不同的路径去打开一个流. 从用户空间角度来看, 这四种不同的类别如下(函数列表只代表示例, 不是完整列表):

```
<?php
    /* fopen包装
     * 操作文件/URI方式指定远程文件类资源 */
    $fp = fopen($url, $mode);
    $data = file_get_contents($url);
    file_put_contents($url, $data);
    $lines = file($url);

    /* 传输
     * 基于套接字的顺序I/O */
    $fp = fsockopen($host, $port);
    $fp = stream_socket_client($uri);
    $fp = stream_socket_server($uri, $options);

    /* 目录流 */
    $dir = opendir($url);
    $files = scandir($url);
    $obj = dir($url);

    /* "特殊"的流 */
    $fp = tmpfile();
    $fp = popen($cmd);
    proc_open($cmd, $pipes);
?>
```

无论你打开的是什么类型的流, 它们都存储在一个公共的结构体php_stream中.

fopen包装

我们首先从实现fopen()函数开始. 现在你应该已经对创建扩展骨架很熟悉了, 如果还不熟悉, 请回到第5章"你的第一个扩展"复习一下, 下面是我们实现的fopen()函数:

```
PHP_FUNCTION(sample5_fopen)
{
    php_stream *stream;
    char *path, *mode;
    int path_len, mode_len;
    int options = ENFORCE_SAFE_MODE | REPORT_ERRORS;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",
            &path, &path_len, &mode, &mode_len) == FAILURE) {
        return;
    }
    stream = php_stream_open_wrapper(path, mode, options, NULL);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

php_stream_open_wrapper()的目的应该是完全绕过底层. path指定要读写文件名或URL, 读写行为依赖于mode的值.

options是位域的标记值集合, 这里是设置为下面介绍的一组固定值:

USE_PATH	将php.ini文件中的include_path应用到相对路径上. 内建函数fopen()在指定第三个参数为TRUE时将会设置这个选项.
STREAM_USE_URL	设置这个选项后, 将只能打开远端URL. 对于php://, file://, zlib://, bzip2://这些URL包装器并不认为它们是远端URL.
ENFORCE_SAFE_MODE	尽管这个常量这样命名, 但实际上设置这个选项后仅仅是启用了安全模式(php.ini文件中的safe_mode指令)的强制检查. 如果没有设置这个选项将导致跳过safe_mode的检查(不论INI设置中safe_mode如何设置)
REPORT_ERRORS	在指定的资源打开过程中碰到错误时, 如果设置了这个选项则将产生错误报告.
STREAM_MUST_SEEK	对于某些流, 比如套接字, 是不可以seek的(随机访问); 这类文件句柄, 只有在特定情况下才可以seek. 如果调用作用域指定这个选项, 并且包装器检测到它不能保证可以seek, 将会拒绝打开这个流.
STREAM_WILL_CAST	如果调用作用域要求流可以被转换到stdio或posix文件描述符, 则应该给open_wrapper函数传递这个选项, 以保证在I/O操作发生之前就失败
STREAM_ONLY_GET_HEADERS	标识只需要从流中请求元数据. 实际上这是用于http包装器, 获取http_response_headers全局变量而不真正的抓取远程文件内容.
STREAM_DISABLE_OPEN_BASEDIR	类似safe_mode检查, 不设置这个选项则会检查INI设置open_basedir, 如果指定这个选项则可以绕过这个默认的检查
STREAM_OPEN_PERSISTENT	告知流包装层, 所有内部分配的空间都采用持久化分配, 并将关联的资源注册到持久化列表中.
IGNORE_PATH	如果不指定, 则搜索默认的包含路径. 多数URL包装器都忽略这个选项.
IGNORE_URL	提供这个选项时, 流包装层只打开本地文件. 所有的is_url包装器都将被忽略.

最后的NULL参数是char **类型, 它最初是用来设置匹配路径, 如果path指向普通文件URL, 则去掉file://部分, 保留直接的文件路径用于传统的文件名操作. 这个参数仅仅是以前引擎内部处理使用的.

此外, 还有php_stream_open_wrapper()的一个扩展版本:

```
php_stream *php_stream_open_wrapper_ex(char *path,
           char *mode, int options, char **opened_path,
           php_stream_context *context);
```

最后一个参数context允许附加的控制, 并可以得到包装器内的通知. 你将在第16章看到这个参数的细节.

传输层包装

尽管传输流和fopen包装流是相同的组件组成的, 但它的注册策略和其他的流不同. 从某种程度上来说, 这是因为用户空间对它们的访问方式的不同造成的, 它们需要实现基于套接字的其他因子.

从扩展开发者角度来看, 打开传输流的过程是相同的. 下面是对fsockopen()的实现:

```
PHP_FUNCTION(sample5_fsockopen)
{
    php_stream *stream;
    char *host, *transport, *errstr = NULL;
    int host_len, transport_len, implicit_tcp = 1, errcode = 0;
    long port = 0;
    int options = ENFORCE_SAFE_MODE;
int flags = STREAM_XPORT_CLIENT | STREAM_XPORT_CONNECT;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s|l",
                        &host, &host_len, &port) == FAILURE) {
        return;
    }
    if (port) {
        int implicit_tcp = 1;
        if (strstr(host, "://")) {
                /* A protocol was specified,
                 * no need to fall back on tcp:// */
            implicit_tcp = 0;
        }
        transport_len = spprintf(&transport, 0, "%s%s:%d",
                implicit_tcp ? "tcp://" : "", host, port);
    } else {
        /* When port isn't specified
         * we can safely assume that a protocol was
         * (e.g. unix:// or udg://) */
        transport = host;
        transport_len = host_len;
    }
    stream = php_stream_xport_create(transport, transport_len,
                            options, flags,
                            NULL, NULL, NULL, &errstr, &errcode);
    if (transport != host) {
        efree(transport);
    }
    if (errstr) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "[%d] %s",
                              errcode, errstr);
        efree(errstr);
    }
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

这个函数的基础构造和前面的fopen示例是一样的. 不同在于host和端口号使用不同的参数指定, 接着为了给出一个传输流URL就必须将它们合并到一起. 在产生了一个有意

义的"路径:后, 将它传递给php_stream_xport_create()函数, 方式和fopen()使用的php_stream_open_wrapper()API一样. php_stream_xport_create()的原型如下:

```
php_stream *php_stream_xport_create(char *xport, int xport_len,
                    int options, int flags,
                    const char *persistent_id,
                                    struct timeval *timeout,
                                    php_stream_context *context,
                                    char **errstr, int *errcode);
```

每个参数的含义如下:

xport	基于URI的传输描述符. 对于基于inet的套接字流, 它可以是tcp://127.0.0.1:80, udp://10.0.0.1:53, ssl://169.254.13.24:445等. 此外, UNIX域传输协议unix:///path/to/socket, udg:///path/to/dgramsocket等都是合法的. xport_len指定了xport的长度, 因此xport是二进制安全的.
options	这个值是由前面php_stream_open_wrapper()中介绍的选项通过按位或组成的值.
flags	由STREAM_XPORT_CLIENT或STREAM_XPORT_SERVER之一与下面另外一张表中将列出的STREAM_XPORT_*常量通过按位或组合得到的值.
persistent_id	如果请求的传输流需要在请求间持久化, 调用作用域可以提供一个key名字描述连接. 指定这个值为NULL创建非持久化连接; 指定为唯一的字符串值将尝试首先从持久化池中查找已有的传输流, 或者没有找到时就创建一个新的持久化流.
timeout	在超时返回失败之前连接的尝试时间. 如果这个值传递为NULL则使用php.ini中指定的默认超时值. 这个参数对服务端传输流没有意义.
errstr	如果在选定的套接字上创建, 连接, 绑定或监听时发生错误, 这里传递的char *引用值将被设置为一个描述发生错误原因的字符串. errstr初始应该指向的是NULL; 如果在返回时它被设置了值, 则调用作用域有责任去释放这个字符串相关的内存.
errcode	通过errstr返回的错误消息对应的数值错误代码.

php_stream_xport_create()的flags参数中使用了STREAM_XPORT_*一族常量定义如下:

STREAM_XPORT_CLIENT	本地端将通过传输层和远程资源建立连接. 这个标记通常和STREAM_XPORT_CONNECT或STREAM_XPORT_CONNECT_ASYNC联合使用.
STREAM_XPORT_SERVER	本地端将通过传输层accept连接. 这个标记通常和STREAM_XPORT_BIND以及STREAM_XPORT_LISTEN一起使用.
STREAM_XPORT_CONNECT	用以说明建立远程资源连接是传输流创建的一部分. 在创建客户端传输流时省略这个标记是合法的, 但是这样做就要求手动的调用php_stream_xport_connect().
STREAM_XPORT_CONNECT_ASYNC	尝试连接到远程资源, 但不阻塞.
STREAM_XPORT_BIND	将传输流绑定到本地资源. 用在服务端传输流时, 这将使得accept连接的传输流准备端口, 路径或特定的端点标识符等信息.
STREAM_XPORT_LISTEN	在已绑定的传输流端点上监听到来的连接. 这通

常用于基于流的传输协议, 比如: tcp://, ssl://, unix://.

目录访问

fopen包装器支持目录访问, 比如file://和ftp://, 还有第三种流打开函数也可以用于目录访问, 下面是对opendir()的实现:

```
PHP_FUNCTION(sample5_opendir)
{
    php_stream *stream;
    char *path;
    int path_len, options = ENFORCE_SAFE_MODE | REPORT_ERRORS;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
                                  &path, &path_len) == FAILURE) {
        return;
    }
    stream = php_stream_opendir(path, options, NULL);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

同样的, 也可以为某个特定目录打开一个流, 比如本地文件系统的目录名或支持目录访问的URL格式资源. 这里我们又看到了options参数, 它和原来的含义一样, 第三个参数NULL原型是php_stream_context类型.

在目录流打开后, 和文件以及传输流一样, 返回给用户空间.

特殊流

还有一些特殊类型的流不能归类到fopen/transport/directory中. 它们中每一个都有自己独有的API:

```
php_stream *php_stream_fopen_tmpfile(void);
php_stream *php_stream_fopen_temporary_file(const char *dir,
                          const char *pfx, char **opened_path);
```

创建一个可seek的缓冲区流用于读写. 在关闭时, 这个流使用的所有临时资源, 包括所有的缓冲区(无论是在内存还是磁盘), 都将被释放. 使用这一组API中的后一个函数, 允许临时文件被以特定的格式命名放到指定路径. 这些内部API调用被用户空间的tmpfile()函数隐藏.

```
php_stream *php_stream_fopen_from_fd(int fd,
               const char *mode, const char *persistent_id);
php_stream *php_stream_fopen_from_file(FILE *file,
               const char *mode);
php_stream *php_stream_fopen_from_pipe(FILE *file,
            const char *mode);
```

这3个API方法接受已经打开的FILE *资源或文件描述符ID, 使用流API的某种操作包装. fd格式的接口不会搜索匹配你前面看到过的fopen函数打开的资源, 但是它会注册持久化的资源, 后续的fopen可以使用到这个持久化资源.

访问流

在你打开一个流之后, 就可以在它上面执行I/O操作了. 使用那种协议包装API创建了流并不重要, 它们都使用相同的访问API.

读

流的读写可以使用下面的API函数组合完成, 它们多数都是遵循POSIX I/O中对应的API规范的:

```
int php_stream_getc(php_stream *stream);
```

从数据流中接收一个字符. 如果流上再没有数据, 则返回EOF.

```
size_t php_stream_read(php_stream *stream, char *buf, size_t count);
```

从指定流中读取指定字节的数据. buf必须预分配至少count字节的内存空间. 这个函数将返回从数据流实际读到缓冲区中的数据字节数.

*php_stream_read()*不同于其他的流读取函数. 如果使用的流不是普通文件流, 哪怕数据流中有超过请求字节数的数据, 并且当前也可以返回, 它也只会调用过一次底层流实现的*read*函数. 这是为了兼容基于包*(*比如*UDP)*的协议的这种做法.

```
char *php_stream_get_line(php_stream *stream, char *buf,
                          size_t maxlen, size_t *returned_len);
char *php_stream_gets(php_stream *stream, char *buf,
                          size_t maxlen);
```

这两个函数从stream中读取最多maxlen个字符, 直到碰到换行符或流结束. buf可以是一个指向预分配的至少maxlen字节的内存空间的指针, 也可以是NULL, 当它是NULL时, 则会自动的创建一个动态大小的缓冲区, 用从流中实际读出的数据填充, 成功后函数返回指向缓冲区的指针, 失败则返回NULL. 如果returned_len传递了非NULL值, 则在返回时它将被设置为实际从流中读取的字节数.

```
char *php_stream_get_record(php_stream *stream,
        size_t maxlen, size_t *returned_len,
        char *delim, size_t delim_len
        TSRMLS_DC);
```

和php_stream_get_line()类似, 这个函数将读取最多maxlen, 或到达EOF/行结束第一次出现的位置. 但是它也有和php_stream_get_line()的不同指出, 这个函数允许指定任意的停止读取标记.

读取目录项

从php流中读取目录项和上面从普通文件中读取普通数据相同. 这些数据放到了固定大小的dirents块中. 内部的php_stream_dirent结构体如下, 它与POSIX定义的dirent结构体一致:

```
typedef struct _php_stream_dirent {
    char d_name[MAXPATHLEN];
} php_stream_dirent;
```

实际上你可以直接使用php_stream_read()函数读取数据到这个结构体中:

```
{
   struct dirent entry;
   if (php_stream_read(stream, (char*)&entry, sizeof(entry))
                               == sizeof(entry)) {
       /* 成功从目录流中读取到一项 */
       php_printf("File: %s\n", entry.d_name);
   }
}
```

由于从目录流中读取是很常见的操作, php流包装层暴露了一个API, 它将记录大小的检查和类型转换处理封装到了一次调用中:

```
php_stream_dirent *php_stream_readdir(php_stream *dirstream,
                            php_stream_dirent *entry);
```

如果成功读取到目录项, 则传入的entry指针将被返回, 否则返回NULL标识错误. 使用这个为目录流特殊构建的函数而不是直接从目录流读取非常重要, 这样做未来流API改变时就不至于和你的代码冲突.

写

和读类似, 向流中写数据只需要传递一个缓冲区和缓冲区长度给流.

```
size_t php_stream_write(php_stream *stream, char *buf,
                                                    size_t count);
size_t php_stream_write_string(php_stream *stream, char *stf);
```

write_string的版本实际上是一个提供便利的宏, 它允许写一个NULL终止的字符串,而不用显式的提供长度. 返回的是实际写到流中的字节数. 要特别小心的是尝试写大数据的时候可能导致流阻塞, 比如套接字流, 而如果流被标记为非阻塞, 则实际写入的数据量可能会小于传递给函数的期望大小.

```
int php_stream_putc(php_stream *stream, int c);
int php_stream_puts(php_string *stream, char *buf);
```

还有一种选择是, 使用php_stream_putc()和php_stream_puts()写入一个字符或一个字符串到流中. 要注意, php_stream_puts()不同于php_stream_write_string(), 虽然它们的原型看起来是一样的, 但是php_stream_puts()会在写出buf中的数据后自动的追加一个换行符.

```
size_t php_stream_printf(php_stream *stream TSRMLS_DC,
                                    const char *format, ...);
```

功能和格式上都类似于fprintf(), 这个API调用允许在写的同时构造字符串而不用去创建临时缓冲区构造数据. 这里我们能够看到的一个明显的不同是它需要TSRMLS_CC宏来保证线程安全.

随机访问, 查看文件偏移量以及缓存的flush

基于文件的流, 以及另外几种流是可以随机访问的. 也就是说, 在流的一个位置读取了一些数据之后, 文件指针可以向前或向后移动, 以非线性顺序读取其他部分.

如果你的流应用代码预测到底层的流支持随机访问, 在打开的时候就应该传递STREAM_MUST_SEEK选项. 对于那些原本就可随机访问的流来说, 这通常不会有什么影响, 因为流本身就是可随机访问的. 而对于那些原本不可随机访问的流, 比如网络I/O或线性访问文件比如FIFO管道, 这个暗示可以让调用程序有机会在流的数据被消耗掉之前, 优雅的失败.

在可随机访问的流资源上工作时, 下面的函数可用来将文件指针移动到任意位置:

```
int php_stream_seek(php_stream *stream, off_t offset, int whence);
int php_stream_rewind(php_stream *stream);
```

offset是相对于whence表示的流位置的偏移字节数, whence的可选值及含义如下:

SEEK_SET offset相对于文件开始位置. php_stream_rewind()API调用实际上是一个宏, 展开后是php_stream_seek(stream, 0, SEEK_SET), 表示移动到文件开始位置偏移0字节处. 当使用SEEK_SET时, 如果offset传递负值被认为是错误的, 将会导致未定义行为. 指定的位置超过流的末尾也是未定义的, 不过结果通常是一个错误或文件被扩大以满足指定的偏移量.

SEEK_CUR offset相对于文件的当前偏移量. 调用php_stream_seek(steram, offset, SEEK_CUR)一般来说等价于php_stream_seek(stream, php_stream_tell() + offset, SEEK_SET);

SEEK_END offset是相对于当前的EOF位置的. 负值的offset表示在EOF之前的位置, 正值和SEEK_SET中描述的是相同的语义, 可能在某些流实现上可以工作.

```
int php_stream_rewinddir(php_stream *dirstream);
```

在目录流上随机访问时, 只有php_stream_rewinddir()函数可用. 使用php_stream_seek()函数将导致未定义行为. 所有的随机访问一族函数返回0标识成功或者-1标识失败.

```
off_t php_stream_tell(php_stream *stream);
```

如你之前所见, php_stream_tell()将返回当前的文件偏移量.

```
int php_stream_flush(php_stream *stream);
```

调用flush()函数将强制将流过滤器此类内部缓冲区中的数据输出到最终的资源中. 在流被关闭时, flush()函数将自动调用, 并且大多数无过滤流资源虽然不进行任何内部缓冲, 但也需要flush. 显式的调用这个函数很少见, 并且通常也是不需要的.

```
int php_stream_stat(php_stream *stream, php_stream_statbuf *ssb);
```

调用php_stream_stat()可以获取到流实例的其他信息, 它的行为类似于fstat()函数. 实际上, php_stream_statbuf结构体现在仅包含一个元素: struct statbuf sb; 因此, php_stream_stat()调用可以如下面例子一样, 直接用传统的fstat()操作替代, 它只是将posix的stat操作翻译成流兼容的:

```
int php_sample4_fd_is_fifo(int fd)
{
    struct statbuf sb;
    fstat(fd, &sb);
    return S_ISFIFO(sb.st_mode);
}
int php_sample4_stream_is_fifo(php_stream *stream)
{
    php_stream_statbuf ssb;
    php_stream_stat(stream, &ssb);
    return S_ISFIFO(ssb.sb.st_mode);
}
```

关闭

所有流的关闭都是通过php_stream_free()函数处理的, 它的原型如下:

```
int php_stream_free(php_stream *stream, int options);
```

这个函数中的options参数允许的值是PHP_STREAM_FREE_xxx一族常量的按位或的结果, 这一族常量定义如下(下面省略PHP_STREAM_FREE_前缀):

CALL_DTOR	流实现的析构器应该被调用. 这里提供了一个时机对特定的流进行显式释放.
RELEASE_STREAM	释放为php_stream结构体分配的内存
PRESERVE_HANDLE	指示流的析构器不要关闭它的底层描述符句柄
RSRC_DTOR	流包装层内部管理资源列表的垃圾回收
PERSISTENT	作用在持久化流上时, 它的行为将是永久的而不局限于当前请求.
CLOSE	CALL_DTOR和RELEASE_STREAM的联合. 这是关闭非持久化流的一般选项.
CLOSE_CASTED	CLOSE和PRESERVE_HANDLE的联合.
CLOSE_PERSISTENT	CLOSE和PERSISTENT的联合. 这是永久关闭持久化流的一般选项.

实际上, 你并不需要直接调用php_stream_free()函数. 而是在关闭流时使用下面两个宏的某个替代:

```
#define php_stream_close(stream) \
    php_stream_free((stream), PHP_STREAM_FREE_CLOSE)
#define php_stream_pclose(stream) \
    php_stream_free((stream), PHP_STREAM_FREE_CLOSE_PERSISTENT)
```

通过zval交换流

因为流通常映射到zval上, 反之亦然, 因此提供了一组宏用来简化操作, 并统一编码(格式):

```
#define php_stream_to_zval(stream, pzval) \
    ZVAL_RESOURCE((pzval), (stream)->rsrc_id);
```

要注意, 这里并没有调用ZEND_REGISTER_RESOURCE(). 这是因为当流打开的时候, 已经自动的注册为资源了, 这样就可以利用到引擎内建的垃圾回收和shutdown系统的优点. 使用这个宏而不是尝试手动的将流注册为新的资源ID是非常重要的; 这样做的最终结果是导致流被关闭两次以及引擎崩溃.

```
#define php_stream_from_zval(stream, ppzval) \
    ZEND_FETCH_RESOURCE2((stream), php_stream*, (ppzval), \
```

```
    -1, "stream", php_file_le_stream(), php_file_le_pstream())
#define php_stream_from_zval_no_verify(stream, ppzval) \
    (stream) = (php_stream*)zend_fetch_resource((ppzval) \
    TSRMLS_CC, -1, "stream", NULL, 2, \
    php_file_le_stream(), php_file_le_pstream())
```

从传入的zval *中取回php_stream *有一个类似的宏. 可以看出, 这个宏只是对资源获取函数(第9章"资源数据类型")的一个简单封装. 请回顾ZEND_FETCH_RESOURCE2()宏, 第一个宏php_stream_from_zval()就是对它的包装, 如果资源类型不匹配, 它将抛出一个警告并尝试从函数实现中返回. 如果你只是想从传入的zval *中获取一个php_stream *, 而不希望有自动的错误处理, 就需要使用php_stream_from_zval_no_verify()并且需要手动的检查结果值.

静态资源操作

一个基于流的原子操作并不需要实际的实例. 下面这些API仅仅使用URL执行这样的操作:

```
int php_stream_stat_path(char *path, php_stream_statbuf *ssb);
```

和前面的php_stream_stat()类似, 这个函数提供了一个对POSIX的stat()函数协议依赖的包装. 要注意, 并不是所有的协议都支持URL记法, 并且即便支持也可能不能报告出statbuf结构体中的所有成员值. 一定要检查php_stream_stat_path()失败时的返回值, 0标识成功, 要知道, 不支持的元素返回时其值将是默认的0.

```
int php_stream_stat_path_ex(char *path, int flags,
        php_stream_statbuf *ssb, php_stream_context *context);
```

这个php_stream_url_stat()的扩展版本允许传递另外两个参数. 第一个是flags, 它的值可以是下面的PHP_STERAM_URL_STAT_*(下面省略PHP_STREAM_URL_STAT_前缀)一族常量的按位或的结果. 还有一个是context参数, 它在其他的一些流函数中也有出现, 我们将在第16章去详细学习.

LINK　　原始的php_stream_stat_path()对于符号链接或目录将会进行解析直到碰到协议定义的结束资源. 传递PHP_STREAM_URL_STAT_LINK标记将导致php_stream_stat_path()返回请求资源的信息而不会进行符号链接的解析. (译注: 我们可以这样理解, 没有这个标记, 底层使用stat(), 如果有这个标记, 底层使用lstat(), 关于stat()和lstat()的区别, 请查看*nix手册)

QUIET　　默认情况下, 如果在执行URL的stat操作过程中碰到错误, 包括文件未找到错误, 都将通过php的错误处理机制触发. 传递QUIET标记可以使得php_stream_stat_path()返回而不报告错误.

```
int php_stream_mkdir(char *path, int mode, int options,
                           php_stream_context *context);
int php_stream_rmdir(char *path, int options,
                           php_stream_context *context);
```

创建和删除目录也会如你期望的工作. 这里的options参数和前面的php_stream_open_wrapper()函数的同名参数含义一致. 对于php_stream_mkdir(), 还有一个参数mode用于指定一个八进制的值表明读写执行权限.

小结

本章中你接触了一些基于流的I/O的内部表象. 下一章将演示做呢样实现自己的协议包装, 甚至是定义自己的流类型.

实现流

php的流最强力的特性之一是它可以访问众多数据源: 普通文件, 压缩文件, 网络透明通道, 加密网络, 命名管道以及域套接字, 它们对于用户空间以及内部都是统一的API.

php流的表象之下

对于给定的流实例, 比如文件流和网络流, 它们的不同在于上一章你使用的流创建函数返回的php_stream结构体中的ops成员.

```
typedef struct _php_stream {
    ...
    php_stream_ops *ops;
    ...
} php_stream;
```

php_stream_ops结构体定义的是一个函数指针集合以及一个描述标记.

```
typedef struct _php_stream_ops {
    size_t (*write)(php_stream *stream, const char *buf,
                            size_t count TSRMLS_DC);
    size_t (*read)(php_stream *stream, char *buf,
                            size_t count TSRMLS_DC);
    int    (*close)(php_stream *stream, int close_handle
                            TSRMLS_DC);
    int    (*flush)(php_stream *stream TSRMLS_DC);

    const char *label;

    int (*seek)(php_stream *stream, off_t offset, int whence,
                            off_t *newoffset TSRMLS_DC);
    int (*cast)(php_stream *stream, int castas, void **ret
                            TSRMLS_DC);
    int (*stat)(php_stream *stream, php_stream_statbuf *ssb
                            TSRMLS_DC);
    int (*set_option)(php_stream *stream, int option,int value,
                            void *ptrparam TSRMLS_DC);
} php_stream_ops;
```

当流访问函数比如php_stream_read()被调用时, 流包装层实际上解析调用了stream->ops中对应的函数, 这样实际调用的就是当前流类型特有的read实现. 比如, 普通文件的流ops结构体中的read函数实现如下(实际的该实现比下面的示例复杂一点):

```
size_t php_stdio_read(php_stream *stream, char *buf,
                                size_t count TSRMLS_DC)
{
    php_stdio_stream_data *data =
                (php_stdio_stream_data*)stream->abstract;
    return read(data->fd, buf, count);
}
```

而compress.zlib流使用的ops结构体中则read则指向的是如下的函数:

```
size_t php_zlib_read(php_stream *stream, char *buf,
                                size_t count TSRMLS_DC)
{
    struct php_gz_stream_data_t *data =
            (struct php_gz_stream_data_t *) stream->abstract;

    return gzread(data->gz_file, buf, count);
}
```

这里第一点需要注意的是ops结构体指向的函数指针常常是对数据源真正的读取函数的一个瘦代理. 在上面两个例子中, 标准I/O流使用posix的read()函数, 而zlib流使用的是libz的gzread()函数.

你可能还注意到了, 这里使用了stream->abstract元素. 这是流实现的一个便利指针, 它可以被用于获取各种相关的捆绑信息. 在上面的例子中, 指向自定义结构体的指针, 用于存储底层read函数要使用的文件描述符.

还有一件你可能注意到的事情是php_stream_ops结构体中的每个函数都期望一个已有的流实例, 但是怎样得到实例呢? abstract成员是怎样设置的以及什么时候流指示使用哪个ops结构体? 答案就在你在上一章使用过的第一个打开流的函数(php_stream_open_wrapper())中.

当这个函数被调用时, php的流包装层尝试基于传递的URL中的scheme://部分确定请求的是什么协议. 这样它就可以在已注册的php包装器中查找对应的php_stream_wrapper项. 每个php_stream_wrapper结构体都可以取到自己的ops元素, 它指向一个php_stream_wrapper_ops结构体:

```
typedef struct _php_stream_wrapper_ops {
    php_stream *(*stream_opener)(php_stream_wrapper *wrapper,
                        char *filename, char *mode,
                        int options, char **opened_path,
                        php_stream_context *context
                        STREAMS_DC TSRMLS_DC);
    int (*stream_closer)(php_stream_wrapper *wrapper,
                        php_stream *stream TSRMLS_DC);
    int (*stream_stat)(php_stream_wrapper *wrapper,
                        php_stream *stream,
                        php_stream_statbuf *ssb
                        TSRMLS_DC);
    int (*url_stat)(php_stream_wrapper *wrapper,
                        char *url, int flags,
                        php_stream_statbuf *ssb,
                        php_stream_context *context
                        TSRMLS_DC);
    php_stream *(*dir_opener)(php_stream_wrapper *wrapper,
                        char *filename, char *mode,
                        int options, char **opened_path,
                        php_stream_context *context
                        STREAMS_DC TSRMLS_DC);

    const char *label;

    int (*unlink)(php_stream_wrapper *wrapper, char *url,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);

    int (*rename)(php_stream_wrapper *wrapper,
                        char *url_from, char *url_to,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);


    int (*stream_mkdir)(php_stream_wrapper *wrapper,
                        char *url, int mode, int options,
                        php_stream_context *context
                        TSRMLS_DC);
    int (*stream_rmdir)(php_stream_wrapper *wrapper, char *url,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);
} php_stream_wrapper_ops;
```

这里, 流包装层调用wrapper->ops->stream_opener(), 它将执行包装器特有的操作创建流实例, 赋值恰当的php_stream_ops结构体, 绑定相关的抽象数据.

dir_opener()函数和stream_opener()提供相同的基础服务; 不过, 它是对php_stream_opendir()这个API调用的响应, 并且通常会绑定一个不同的php_stream_ops结构体到返回的实例. stat()和close()函数在这一层上是重复的, 这样做是为了给包装器的这些操作增加协议特有的逻辑.

其他的函数则允许执行静态流操作而不用实际的创建流实例. 回顾这些流API调用, 它们并不实际返回php_stream对象, 你马上就会看到它们的细节.

尽管在*php 4.3*中引入流包装层时, *url_stat*在内部作为一个包装器的*ops*函数存在, 但直到*php 5.0*它才开始被使用. 此外, 最后的*3*个函数, *rename(), stream_mkdir()*以及*stream_rmdir()*一直到*php 5.0*才引入, 在这个版本之前, 它们并不在包装器的*ops*结构中.

包装器操作

除了url_stat()函数, 包装器操作中在const char *label元素之前的每个操作都可以用于激活的流实例上. 每个函数的意义如下:

stream_opener()	实例化一个流实例. 当某个用户空间的fopen()函数被调用时, 这个函数将被调用. 这个函数返回的php_stream实例是fopen()函数返回的文件资源句柄的内部表示. 集成函数比如file(), file_get_contents(), file_put_contents(), readfile()等等, 在请求包装资源时, 都使用这个包装器ops.
stream_closer()	当一个流实例结束其生命周期时这个函数被调用. stream_opener()时分配的所有资源都应该在这个函数中被释放.
stream_stat()	类似于用户空间的fstat()函数, 这个函数应该填充ssb结构体(实际上只包含一个struct statbuf sb结构体成员),
dir_opener()	和stream_opener()行为一致, 不过它是调用opendir()一族的用户空间函数时被调用的. 目录流使用的底层流实现和文件流遵循相同的规则; 不过目录流只需要返回包含在打开的目录中找到的文件名的记录, 它的大小为struct dirent这个结构体的大小.

静态包装器操作

包装器操作函数中的其他函数是在URI路径上执行原子操作, 具体取决于包装器协议. 在php4.3的php_stream_wrapper_ops结构体中只有url_stat()和unlink(); 其他的方式是到php 5.0后才定义的, 编码时应该适时的使用#ifdef块说明.

url_stat()	stat()族函数使用, 返回文件元数据, 比如访问授权, 大小, 类型; 以及访问, 修改,创建时间. 尽管这个函数是在php 4.3引入流包装层时出现在php_stream_wrapper_ops结构体中的, 但直到php 5.0才被用户空间的stat()函数使用.
unlink()	和posix文件系统的同名函数语义相同, 它执行文件删除. 如果对于当前的包装器删除没有意义, 比如内建的http://包装器, 这个函数应该被定义为NULL, 以便内核去引发适当的错误消息.
rename()	当用户空间的rename()函数的参数$from和$to参数指向的是相同的底层包装器实现, php则将这个重命名请求分发到包装器的rename函数.
mkdir() & rmdir()	这两个函数直接映射到对应的用户空间函数.

实现一个包装器

为了演示包装器和流操作的内部工作原理, 我们需要重新实现php手册的stream_wrapper_register()一页示例中的var:///包装器.

此刻, 首先从下面功能完整的变量流包装实现开始. 构建他, 并开始检查每一块的工作原理.

译注: 为了方便大家阅读, 对代码的注释进行了适量补充调整, 此外, 由于*phpapi*的调整, 原著中的代码不能直接在译者使用的*php-5.4.10*中运行, 进行了适当的修改. 因此下面代码结构可能和原著略有不同, 请参考阅读.*(*下面*opendir*的例子也进行了相应的修改*)*

config.m4

```
PHP_ARG_ENABLE(varstream,whether to enable varstream support,
[  enable-varstream      Enable varstream support])

if test "$PHP_VARSTREAM" = "yes"; then
  AC_DEFINE(HAVE_VARSTREAM,1,[Whether you want varstream])
  PHP_NEW_EXTENSION(varstream, varstream.c, $ext_shared)
fi
```

php_varstream.h

```
#ifndef PHP_VARSTREAM_H
#define PHP_VARSTREAM_H

extern zend_module_entry varstream_module_entry;
#define phpext_varstream_ptr &varstream_module_entry

#ifdef PHP_WIN32
#   define PHP_VARSTREAM_API __declspec(dllexport)
#elif defined(__GNUC__) && __GNUC__ >= 4
#   define PHP_VARSTREAM_API __attribute__ ((visibility("default")))
#else
#   define PHP_VARSTREAM_API
#endif

#ifdef ZTS
#include "TSRM.h"
#endif

PHP_MINIT_FUNCTION(varstream);
PHP_MSHUTDOWN_FUNCTION(varstream);

#define PHP_VARSTREAM_WRAPPER       "var"
#define PHP_VARSTREAM_STREAMTYPE    "varstream"

/* 变量流的抽象数据结构 */
typedef struct _php_varstream_data {
    off_t   position;
    char    *varname;
    int     varname_len;
} php_varstream_data;

#ifdef ZTS
#define VARSTREAM_G(v) TSRMG(varstream_globals_id, zend_varstream_globals *, v)
#else
#define VARSTREAM_G(v) (varstream_globals.v)
#endif

#endif
```

varstream.c

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
```

```
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/url.h"
#include "php_varstream.h"

static size_t php_varstream_write(php_stream *stream,
                const char *buf, size_t count TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var;
    size_t newlen;

    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), data->varname,
            data->varname_len + 1,(void**)&var) == FAILURE) {
        /* 变量不存在, 直接创建一个字符串类型的变量, 并保存新传递进来的内容 */
       zval *newval;
       MAKE_STD_ZVAL(newval);
       ZVAL_STRINGL(newval, buf, count, 1);
       /* 将新的zval *放到变量中 */
       zend_hash_add(&EG(symbol_table), data->varname,
           data->varname_len + 1, (void*)&newval,
           sizeof(zval*), NULL);
       return count;
    }
    /* 如果需要, 让变量可写. 这里实际上处理的是写时复制 */
    SEPARATE_ZVAL_IF_NOT_REF(var);
    /* 转换为字符串类型 */
    convert_to_string_ex(var);
    /* 重置偏移量(译注: 相比于正常的文件系统, 这里的处理实际上不支持文件末尾的空洞创建, 读者如果熟悉*nix文件系统,
应该了解译者所说, 否则请略过) */
    if (data->position > Z_STRLEN_PP(var)) {
        data->position = Z_STRLEN_PP(var);
    }
    /* 计算新的字符串长度 */
    newlen = data->position + count;
    if (newlen < Z_STRLEN_PP(var)) {
        /* 总长度不变 */
        newlen = Z_STRLEN_PP(var);
    } else if (newlen > Z_STRLEN_PP(var)) {
        /* 重新调整缓冲区大小以保存新内容 */
        Z_STRVAL_PP(var) =erealloc(Z_STRVAL_PP(var),newlen+1);
        /* 更新字符串长度 */
        Z_STRLEN_PP(var) = newlen;
        /* 确保字符串NULL终止 */
        Z_STRVAL_PP(var)[newlen] = 0;
    }
    /* 将数据写入到变量中 */
    memcpy(Z_STRVAL_PP(var) + data->position, buf, count);
    data->position += count;

    return count;
}

static size_t php_varstream_read(php_stream *stream,
                char *buf, size_t count TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var, copyval;
    int got_copied = 0;
    size_t toread = count;

    if (zend_hash_find(&EG(symbol_table), data->varname,
        data->varname_len + 1, (void**)&var) == FAILURE) {
```

```
        /* 变量不存在, 读不到数据, 返回0字节长度 */
        return 0;
    }
    copyval = **var;
    if (Z_TYPE(copyval) != IS_STRING) {
        /* 对于非字符串类型变量, 创建一个副本进行读, 这样对于只读的变量, 就不会改变其原始类型 */
        zval_copy_ctor(&copyval);
        INIT_PZVAL(&copyval);
        got_copied = 1;
    }
    if (data->position > Z_STRLEN(copyval)) {
        data->position = Z_STRLEN(copyval);
    }
    if ((Z_STRLEN(copyval) - data->position) < toread) {
        /* 防止读取到变量可用缓冲区外的内容 */
        toread = Z_STRLEN(copyval) - data->position;
    }
    /* 设置缓冲区 */
    memcpy(buf, Z_STRVAL(copyval) + data->position, toread);
    data->position += toread;

    /* 如果创建了副本, 则释放副本 */
    if (got_copied) {
        zval_dtor(&copyval);
    }

    /* 返回设置到缓冲区的字节数 */
    return toread;
}

static int php_varstream_closer(php_stream *stream,
                            int close_handle TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;

    /* 释放内部结构避免泄露 */
    efree(data->varname);
    efree(data);

    return 0;
}

static int php_varstream_flush(php_stream *stream TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var;

    /* 根据不同情况, 重置偏移量 */
    if (zend_hash_find(&EG(symbol_table), data->varname,
                   data->varname_len + 1, (void**)&var)
                   == SUCCESS) {
        if (Z_TYPE_PP(var) == IS_STRING) {
            data->position = Z_STRLEN_PP(var);
        } else {
            zval copyval = **var;
            zval_copy_ctor(&copyval);
            convert_to_string(&copyval);
            data->position = Z_STRLEN(copyval);
            zval_dtor(&copyval);
        }
    } else {
        data->position = 0;
    }
```

```
    return 0;
}

static int php_varstream_seek(php_stream *stream, off_t offset,
                  int whence, off_t *newoffset TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;

    switch (whence) {
        case SEEK_SET:
            data->position = offset;
            break;
        case SEEK_CUR:
            data->position += offset;
            break;
        case SEEK_END:
        {
            zval **var;
           size_t curlen = 0;

           if (zend_hash_find(&EG(symbol_table),
                   data->varname,    data->varname_len + 1,
                   (void**)&var) == SUCCESS) {
              if (Z_TYPE_PP(var) == IS_STRING) {
                  curlen = Z_STRLEN_PP(var);
              } else {
                  zval copyval = **var;
                  zval_copy_ctor(&copyval);
                  convert_to_string(&copyval);
                  curlen = Z_STRLEN(copyval);
                  zval_dtor(&copyval);
              }
           }

           data->position = curlen + offset;
           break;
       }
    }

    /* 防止随机访问指针移动到缓冲区开始位置之前 */
    if (data->position < 0) {
        data->position = 0;
    }

    if (newoffset) {
        *newoffset = data->position;
    }

    return 0;
}

static php_stream_ops php_varstream_ops = {
    php_varstream_write,
    php_varstream_read,
    php_varstream_closer,
    php_varstream_flush,
    PHP_VARSTREAM_STREAMTYPE,
    php_varstream_seek,
    NULL, /* cast */
    NULL, /* stat */
    NULL, /* set_option */
};

/* Define the wrapper operations */
```

```
static php_stream *php_varstream_opener(
            php_stream_wrapper *wrapper,
            char *filename, char *mode, int options,
            char **opened_path, php_stream_context *context
            STREAMS_DC TSRMLS_DC)
{
    php_varstream_data *data;
    php_url *url;

    if (options & STREAM_OPEN_PERSISTENT) {
        /* 按照变量流的定义, 是不能持久化的
         * 因为变量在请求结束后将被释放
         */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unable to open %s persistently",
                                    filename);
        return NULL;
    }

    /* 标准URL解析: scheme://user:pass@host:port/path?query#fragment */
    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing URL");
        return NULL;
    }
    /* 检查是否有变量流URL必须的元素host, 以及scheme是否是var */
    if (!url->host || (url->host[0] == 0) ||
        strcasecmp("var", url->scheme) != 0) {
        /* Bad URL or wrong wrapper */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Invalid URL, must be in the form: "
                    "var://variablename");
        php_url_free(url);
        return NULL;
    }

    /* 创建一个数据结构保存协议信息(变量流协议重要是变量名, 变量名长度, 当前偏移量) */
    data = emalloc(sizeof(php_varstream_data));
    data->position = 0;
    data->varname_len = strlen(url->host);
    data->varname = estrndup(url->host, data->varname_len + 1);
    /* 释放前面解析出来的url占用的内存 */
    php_url_free(url);

    /* 实例化一个流, 为其赋予恰当的流ops, 绑定抽象数据 */
    return php_stream_alloc(&php_varstream_ops, data, 0, mode);
}

static php_stream_wrapper_ops php_varstream_wrapper_ops = {
    php_varstream_opener, /* 调用php_stream_open_wrapper(sprintf("%s://xxx",
PHP_VARSTREAM_WRAPPER))时执行 */
    NULL, /* stream_close */
    NULL, /* stream_stat */
    NULL, /* url_stat */
    NULL, /* dir_opener */
    PHP_VARSTREAM_WRAPPER,
    NULL, /* unlink */
#if PHP_MAJOR_VERSION >= 5
    /* PHP >= 5.0 only */
    NULL, /* rename */
    NULL, /* mkdir */
    NULL, /* rmdir */
#endif
```

```
};

static php_stream_wrapper php_varstream_wrapper = {
    &php_varstream_wrapper_ops,
    NULL, /* abstract */
    0, /* is_url */
};

PHP_MINIT_FUNCTION(varstream)
{
    /* 注册流包装器:
     * 1. 检查流包装器名字是否正确(符合这个正则: /^[a-zA-Z0-9+.-]+$/)
     * 2. 将传入的php_varstream_wrapper增加到url_stream_wrappers_hash这个HashTable中, key为
PHP_VARSTREAM_WRAPPER
     */
    if (php_register_url_stream_wrapper(PHP_VARSTREAM_WRAPPER,
            &php_varstream_wrapper TSRMLS_CC)==FAILURE) {
        return FAILURE;
    }
    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(varstream)
{
    /* 卸载流包装器: 从url_stream_wrappers_hash中删除 */
    if (php_unregister_url_stream_wrapper(PHP_VARSTREAM_WRAPPER
                                TSRMLS_CC) == FAILURE) {
        return FAILURE;
    }
    return SUCCESS;
}

zend_module_entry varstream_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    "varstream",
    NULL,
    PHP_MINIT(varstream),
    PHP_MSHUTDOWN(varstream),
    NULL,
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    "0.1",
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_VARSTREAM
ZEND_GET_MODULE(varstream)
#endif
```

在构建加载扩展后, php就可以处理以var://开始的URL的请求, 它的行为和手册中用户空间实现的行为一致.

内部实现

首先你注意到的可能是这个扩展完全没有暴露用户空间函数. 它所做的只是在MINIT函数中调用了一个核心PHPAPI的钩子, 将var协议和我们定义的包装器关联起来:

```
static php_stream_wrapper php_varstream_wrapper = {
    &php_varstream_wrapper_ops,
    NULL, /* abstract */
    0, /* is_url */
```

```
}
```

很明显, 最重要的元素就是ops, 它提供了访问特定流包装器的创建以及检查函数. 你可以安全的忽略abstract属性, 它仅在运行时使用, 在初始化定义时, 它只是作为一个占位符. 第三个元素is_url, 它告诉php在使用这个包装器时是否考虑php.ini中的allow_url_fopen选项. 如果这个值非0, 并且将allow_url_fopen设置为false, 则这个包装器不能被脚本使用.

在本章前面你已经知道, 调用用户空间函数比如fopen将通过这个包装器的ops元素得到php_varstream_wrapper_ops, 这样去调用流的打开函数php_varstream_opener.

这个函数的第一块代码检查是否请求持久化的流:

```
if (options & STREAM_OPEN_PERSISTENT) {
```

对于很多包装器这样的请求是合法的. 然而目前的情况这个行为没有意义. 一方面用户空间变量的定义就是临时的, 另一方面, varstream的实例化代价很低, 这就使得持久化的优势很小.

像流包装层报告错误很简单, 只需要返回一个NULL值而不是流实例即可. 流包装层透出到用户空间的失败消息并不会说明具体的错误, 只是说明不能打开URL. 要想给开发者暴露更多的错误信息, 可以在返回之前使用php_stream_wrapper_log_error()函数.

```
php_stream_wrapper_log_error(wrapper, options
    TSRMLS_CC, "Unable to open %s persistently",
                                filename);
return NULL;
```

URL解析

实例化varstream的下一步需要一个人类可读的URL, 将它分块放入到一个易管理的结构体中. 幸运的是它使用了和用户空间url_parse()函数相同的机制. 如果URL成功解析, 将会分配一个php_url结构体并设置合适的值. 如果在URL中没有某些值, 在返回的php_url中对应的将被设置为NULL. 这个结构体必须在离开php_varstream_opener函数之前被显式释放, 否则它的内存将会泄露:

```
typedef struct php_url {
    /* scheme://user:pass@host:port/path?query#fragment */
    char *scheme;
    char *user;
    char *pass;
    char *host;
    unsigned short port;
    char *path;
    char *query;
    char *fragment;
} php_url;
```

最后, varstream包装器创建了一个数据结构, 保存了流指向的变量名, 读取时的当前位置. 这个结构体将在流的读取和写入函数中用于获取变量, 并且将在流结束使用时由php_varstream_close函数释放.

opendir()

读写变量内容的实现可以再次进行扩展. 这里可以加入一个新的特性, 允许使用目录函数读取数组中的key. 在你的php_varstream_wrapper_ops结构体之前增加下面的代码:

```
static size_t php_varstream_readdir(php_stream *stream,
                char *buf, size_t count TSRMLS_DC)
{
    php_stream_dirent *ent = (php_stream_dirent*)buf;
    php_varstream_dirdata *data = stream->abstract;
    char *key;
    int type, key_len;
    long idx;
```

```
    /* 查找数组中的key */
    type = zend_hash_get_current_key_ex(Z_ARRVAL_P(data->arr),
                    &key, &key_len, &idx, 0, &(data->pos));

    /* 字符串key */
    if (type == HASH_KEY_IS_STRING) {
        if (key_len >= sizeof(ent->d_name)) {
            /* truncate long keys to maximum length */
            key_len = sizeof(ent->d_name) - 1;
        }
        /* 设置到目录结构上 */
        memcpy(ent->d_name, key, key_len);
        ent->d_name[key_len] = 0;
    /* 数值key */
    } else if (type == HASH_KEY_IS_LONG) {
        /* 设置到目录结构上 */
        snprintf(ent->d_name, sizeof(ent->d_name), "%ld",idx);
    } else {
        /* 迭代结束 */
        return 0;
    }
    /* 移动数组指针(位置记录到流的抽象结构中) */
    zend_hash_move_forward_ex(Z_ARRVAL_P(data->arr),
                                        &data->pos);
    return sizeof(php_stream_dirent);
}

static int php_varstream_closedir(php_stream *stream,
                            int close_handle TSRMLS_DC)
{
    php_varstream_dirdata *data = stream->abstract;

    zval_ptr_dtor(&(data->arr));
    efree(data);
    return 0;
}

static int php_varstream_dirseek(php_stream *stream,
                    off_t offset, int whence,
                    off_t *newoffset TSRMLS_DC)
{
    php_varstream_dirdata *data = stream->abstract;

    if (whence == SEEK_SET && offset == 0) {
        /* 重置数组指针 */
        zend_hash_internal_pointer_reset_ex(
                    Z_ARRVAL_P(data->arr), &(data->pos));
        if (newoffset) {
            *newoffset = 0;
        }
        return 0;
    }
    /* 不支持其他类型的随机访问 */
    return -1;
}

static php_stream_ops php_varstream_dirops = {
    NULL, /* write */
    php_varstream_readdir,
    php_varstream_closedir,
    NULL, /* flush */
    PHP_VARSTREAM_DIRSTREAMTYPE,
    php_varstream_dirseek,
```

```
    NULL, /* cast */
    NULL, /* stat */
    NULL, /* set_option */
};

static php_stream *php_varstream_opendir(
           php_stream_wrapper *wrapper,
           char *filename, char *mode, int options,
           char **opened_path, php_stream_context *context
           STREAMS_DC TSRMLS_DC)
{
    php_varstream_dirdata *data;
    php_url *url;
    zval **var;

    /* 不支持持久化流 */
    if (options & STREAM_OPEN_PERSISTENT) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Unable to open %s persistently",
                filename);
        return NULL;
    }

    /* 解析URL */
    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Unexpected error parsing URL");
        return NULL;
    }
    /* 检查请求URL的正确性 */
    if (!url->host || (url->host[0] == 0) ||
            strcasecmp("var", url->scheme) != 0) {
        /* Bad URL or wrong wrapper */
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Invalid URL, must be in the form: "
                "var://variablename");
        php_url_free(url);
        return NULL;
    }

    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), url->host,
                strlen(url->host) + 1, (void**)&var) == FAILURE) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Variable $%s not found", url->host);
        php_url_free(url);
        return NULL;
    }

    /* 检查变量类型 */
    if (Z_TYPE_PP(var) != IS_ARRAY) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "$%s is not an array", url->host);
        php_url_free(url);
        return NULL;
    }
    /* 释放前面分配的URL结构 */
    php_url_free(url);

    /* 分配抽象数据结构 */
    data = emalloc(sizeof(php_varstream_dirdata));
    if ( Z_ISREF_PP(var) && Z_REFCOUNT_PP(var) > 1) {
        /* 全拷贝 */
```

```
        MAKE_STD_ZVAL(data->arr);
        *(data->arr) = **var;
        zval_copy_ctor(data->arr);
        INIT_PZVAL(data->arr);
    } else {
        /* 写时拷贝 */
        data->arr = *var;
        Z_SET_REFCOUNT_P(data->arr, Z_REFCOUNT_P(data->arr) + 1);
    }
    /* 重置数组指针 */
    zend_hash_internal_pointer_reset_ex(Z_ARRVAL_P(data->arr),
            &data->pos);
    return php_stream_alloc(&php_varstream_dirops,data,0,mode);
}
```

现在, 将你的php_varstream_wrapper_ops结构体中的dir_opener的NULL替换成你的php_varstream_opendir函数. 最后, 将下面新定义的类型放入到你的php_varstream.h文件的php_varstream_data定义下面:

```
#define PHP_VARSTREAM_DIRSTREAMTYPE    "varstream directory"
typedef struct _php_varstream_dirdata {
    zval *arr;
    HashPosition pos;
} php_varstream_dirdata;
```

在你基于fopen()实现的varstream包装器中, 你直接使用持久变量名, 每次执行读写操作时从符号表中获取变量. 而这里, opendir()的实现中获取变量时处理了变量不存在或者类型错误的异常. 你还有一个数组变量的拷贝, 这就说明原数组的改变并不会影响后续的readdir()调用的结果. 原来存储变量名的方式也可以正常工作, 这里只是给出另外一种选择作为演示示例.

由于目录访问是基于成块的目录条目, 而不是字符, 因此这里需要一套独立的流操作. 这个版本中, write没有意义, 因此保持它为NULL. read的实现使用zend_hash_get_current_key_ex()函数将数组映射到目录名. 而随机访问也只是对SEEK_SET有效, 用来响应rewinddir()跳转到数组开始位置.

实际上, 目录流并没有使用*SEEK_CUR, SEEK_END*, 或者除了*0*之外的偏移量. 在实现目录流操作时, 最好还是涉及你的函数能以某种方式处理这些情况, 以使得在流包装层变化时能够适应其目录随机访问.

操纵

5个静态包装器操作中的4个用来处理不是基于I/O的流资源操作. 你已经看到过它们并了解它们的原型; 现在我们看看varstream包装器框架中它们的实现:

unlink

在你的wrapper_ops结构体中增加下面的函数, 它可以让unlink()通过varstream包装器, 拥有和unset()一样的行为:

```
static int php_varstream_unlink(php_stream_wrapper *wrapper,
                        char *filename, int options,
                        php_stream_context *context
                        TSRMLS_DC)
{
    php_url *url;

    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing URL");
        return -1;
    }
```

```
    if (!url->host || (url->host[0] == 0) ||
        strcasecmp("var", url->scheme) != 0) {
        /* URL不合法 */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Invalid URL, must be in the form: "
                    "var://variablename");
        php_url_free(url);
        return -1;
    }

    /* 从符号表删除变量 */
    //zend_hash_del(&EG(symbol_table), url->host, strlen(url->host) + 1);
    zend_delete_global_variable(url->host, strlen(url->host) + 1 TSRMLS_CC);

    php_url_free(url);
    return 0;
}
```

这个函数的编码量和php_varstream_opener差不多. 唯一的不同在于这里你需要传递变量名给zend_hash_del()去删除变量.

译注: 译者的*php-5.4.10*环境中, 使用*unlink()*删除变量后, 在用户空间再次读取该变量名的值会导致*core dump.* 因此上面代码中译者进行了修正, 删除变量时使用了*zend_delete_global_variable()*, 请读者参考阅读*zend_delete_global_variable()*函数源代码, 考虑为什么直接用*zend_hash_del()*删除, 会导致*core dump.* 下面是译者测试用的用户空间代码:

```
<?php
$fp = fopen('var://hello', 'r');
fwrite($fp, 'world');
var_dump($hello);
unlink('var://hello');
$a  = $hello;
```

这个函数的代码量应该和php_varstream_opener差不多. 唯一的不同是这里是传递变量名给zend_hash_del()去删除变量.

rename, mkdir, rmdir

为了一致性, 下面给出rename, mkdir, rmdir函数的实现:

```
static int php_varstream_rename(php_stream_wrapper *wrapper,
        char *url_from, char *url_to, int options,
        php_stream_context *context TSRMLS_DC)
{
    php_url *from, *to;
    zval **var;

    /* 来源URL解析 */
    from = php_url_parse(url_from);
    if (!from) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing source");
        return -1;
    }
    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), from->host,
                strlen(from->host) + 1,
                (void**)&var) == FAILURE) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "$%s does not exist", from->host);
        php_url_free(from);
        return -1;
    }
    /* 目标URL解析 */
    to = php_url_parse(url_to);
    if (!to) {
        php_stream_wrapper_log_error(wrapper, options
```

```
            TSRMLS_CC, "Unexpected error parsing dest");
        php_url_free(from);
        return -1;
    }
    /* 变量的改名 */
    Z_SET_REFCOUNT_PP(var, Z_REFCOUNT_PP(var) + 1);
    zend_hash_update(&EG(symbol_table), to->host,
                strlen(to->host) + 1, (void*)var,
                sizeof(zval*), NULL);
    zend_hash_del(&EG(symbol_table), from->host,
                strlen(from->host) + 1);
    php_url_free(from);
    php_url_free(to);
    return 0;
}

static int php_varstream_mkdir(php_stream_wrapper *wrapper,
                char *url_from, int mode, int options,
                php_stream_context *context TSRMLS_DC)
{
    php_url *url;

    /* URL解析 */
    url = php_url_parse(url_from);
    if (!url) {
       php_stream_wrapper_log_error(wrapper, options
           TSRMLS_CC, "Unexpected error parsing URL");
       return -1;
    }

    /* 检查变量是否存在 */
    if (zend_hash_exists(&EG(symbol_table), url->host,
                    strlen(url->host) + 1)) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "$%s already exists", url->host);
        php_url_free(url);
        return -1;
    }
    /* EG(uninitialized_zval_ptr)通常是IS_NULL的zval *, 引用计数无限大 */
    zend_hash_add(&EG(symbol_table), url->host,
            strlen(url->host) + 1,
            (void*)&EG(uninitialized_zval_ptr),
            sizeof(zval*), NULL);
    php_url_free(url);
    return 0;
}

static int php_varstream_rmdir(php_stream_wrapper *wrapper,
                char *url, int options,
                php_stream_context *context TSRMLS_DC)
{
    /* 行为等价于unlink() */
    wrapper->wops->unlink(wrapper, url, options,
                                context TSRMLS_CC);
}
```

检查

并不是所有的流操作都涉及到资源的操纵. 有时候也需要查看活动的流在某个时刻的状态, 或检查潜在可打开的资源的状态.

这一节流和包装器的ops函数都是在相同的数据结构php_stream_statbuf上工作的, 它只有一个元素: posix标准的struct statbuf. 当本节的某个函数被调用时, 将尝试填充尽可能多的statbuf元素的成员.

stat

如果设置, 当请求激活流实例的信息时, 将会调用wrapper->ops->stream_stat(). 如果没有设置, 则对应的stream->ops->stat()将会被调用. 无论哪个函数被调用, 都应该尽可能多的向返回的statbuf结构体ssb->sb中填充尽可能多流实例的有用信息. 在普通文件I/O的用法中, 它对应fstat()的标准I/O调用.

url_stat

在流实例外部调用wrapper->ops->url_stat()取到流资源的元数据. 通常来说, 符号链接和重定向都应该被解析, 直到找到一个真正的资源, 对其通过stat()系统调用这样的机制读取统计信息. url_stat的flags参数允许是下面PHP_STREAM_URL_STAT_*系列的常量值(省略PHP_STREAM_URL_STAT_前缀):

LINK　　不解析符号链接和重定向. 而是报告它碰到的第一个节点的信息, 无论是连接还是真正的资源.

QUIET　　不报告错误. 注意, 这和许多其他流函数中的REPORT_ERRORS逻辑恰恰相反.

小结

无论是暴露远程网络I/O还是本地数据源的流资源, 都允许你的扩展在核心数据上挂在操纵函数的钩子, 避免重新实现单调的描述符管理和I/O缓冲区工作. 这使得它在用户空间环境中更加有用, 更加强大.

下一章将通过对过滤器和上下文的学习结束流包装层的学习, 过滤器和上下文可以用于选择默认的流行为, 甚至过程中修改数据.

有趣的流

php常被提起的一个特性是流上下文. 这个可选的参数甚至在用户空间大多数流创建相关的函数中都可用, 它作为一个泛化的框架用于向给定包装器或流实现传入/传出额外的信息.

上下文

每个流的上下文包含两种内部消息类型. 首先最常用的是上下文选项. 这些值被安排在上下文中一个二维数组中, 通常用于改变流包装器的初始化行为. 还有一种则是上下文参数, 它对于包装器是未知的, 当前提供了一种方式用于在流包装层内部的事件通知.

```
php_stream_context *php_stream_context_alloc(void);
```

通过这个API调用可以创建一个上下文, 它将分配一些存储空间并初始化用于保存上下文选项和参数的HashTable. 还会自动的注册为一个请求终止后将被清理的资源.

设置选项

设置上下文选项的内部API和用户空间的API是等同的:

```
int php_stream_context_set_option(php_stream_context *context,
            const char *wrappername, const char *optionname,
            zval *optionvalue);
```

下面是用户空间的原型:

```
bool stream_context_set_option(resource $context,
            string $wrapper, string $optionname,
            mixed $value);
```

它们的不同仅仅是用户空间和内部需要的数据类型不同.下面的例子就是使用这两个API调用, 通过内建包装器发起一个HTTP请求, 并通过一个上下文选项覆写了user_agent设置.

```
php_stream  *php_varstream_get_homepage(const char *alt_user_agent TSRMLS_DC)
{
    php_stream_context  *context;
    zval    tmpval;

    context = php_stream_context_alloc(TSRMLS_C);
    ZVAL_STRING(&tmpval, alt_user_agent, 0);
    php_stream_context_set_option(context, "http", "user_agent", &tmpval);
    return php_stream_open_wrapper_ex("http://www.php.net", "rb", REPORT_ERRORS |
ENFORCE_SAFE_MODE, NULL, context);
}
```

译者使用的*php-5.4.10*中*php_stream_context_alloc()*增加了线程安全控制, 因此相应的对例子进行了修改, 请读者测试时注意.

这里要注意的是*tmpval*并没有分配任何持久性的存储空间, 它的字符串值是通过复制设置的. *php_stream_context_set_option()*会自动的对传入的*zval*内容进行一次拷贝.

取回选项

用于取回上下文选项的API调用正好是对应的设置API的镜像:

```
int php_stream_context_get_option(php_stream_context *context,
            const char *wrappername, const char *optionname,
            zval ***optionvalue);
```

回顾前面, 上下文选项存储在一个嵌套的HashTable中, 当从一个HashTable中取回值时, 一般的方法是传递一个指向zval **的指针给zend_hash_find(). 当然, 由于php_stream_context_get_option()是zend_hash_find()的一个特殊代理, 它们的语义是相同的.

下面是内建的http包装器使用php_stream_context_get_option()设置user_agent的简化版示例:

```
zval **ua_zval;
char *user_agent = "PHP/5.1.0";
if (context &&
    php_stream_context_get_option(context, "http",
                "user_agent", &ua_zval) == SUCCESS &&
                Z_TYPE_PP(ua_zval) == IS_STRING) {
    user_agent = Z_STRVAL_PP(ua_zval);
}
```

这种情况下, 非字符串值将会被丢弃, 因为对用户代理字符串而言, 数值是没有意义的. 其他的上下文选项, 比如max_redirects, 则需要数字值, 由于在字符串的zval中存储数字值并不通用, 所以需要执行一个类型转换以使设置合法.

不幸的是这些变量是上下文拥有的, 因此它们不能直接转换; 而需要首先进行隔离再进行转换, 最终如果需要还要进行销毁:

```
long max_redirects = 20;
zval **tmpzval;
if (context &&
    php_stream_context_get_option(context, "http",
            "max_redirects", &tmpzval) == SUCCESS) {
    if (Z_TYPE_PP(tmpzval) == IS_LONG) {
        max_redirects = Z_LVAL_PP(tmpzval);
    } else {
        zval copyval = **tmpzval;
        zval_copy_ctor(&copyval);
        convert_to_long(&copyval);
        max_redirects = Z_LVAL(copyval);
        zval_dtor(&copyval);
    }
}
```

实际上, 在这个例子中, *zval_dtor()*并不是必须的. *IS_LONG*的变量并不需要*zval*容器之外的存储空间, 因此*zval_dtor()*实际上不会有真正的操作. 在这个例子中包含它是为了完整性考虑, 对于字符串, 数组, 对象, 资源以及未来可能的其他类型, 就需要这个调用了.

参数

虽然用户空间API中看起来参数和上下文选项是类似的, 但实际上在语言内部的php_stream_context结构体中它们被定义为不同的成员.

目前只支持一个上下文参数: 通知器. php_stream_context结构体中的这个元素可以指向下面的php_stream_notifier结构体:

```
typedef struct {
    php_stream_notification_func func;
    void (*dtor)(php_stream_notifier *notifier);
    void *ptr;
    int mask;
    size_t progress, progress_max;
} php_stream_notifier;
```

当将一个php_stream_notifier结构体赋值给context->notifier时, 它将提供一个回调函数func, 在特定的流上发生下表中的PHP_STREAM_NOTIFY_*代码表示的事件时被触发. 每个事件将会对应下面第二张表中的PHP_STREAM_NOTIFY_SEVERITY_*的级别:

事件代码	含义
RESOLVE	主机地址解析完成. 多数基于套接字的包装器将在连接之前执行这个查询.

事件代码	含义
CONNECT	套接字流连接到远程资源完成.
AUTH_REQUIRED	请求的资源不可用, 原因是访问控制以及缺失授权
MIME_TYPE_IS	远程资源的mime-type不可用
FILE_SIZE_IS	远程资源当前可用大小
REDIRECTED	原来的URL请求导致重定向到其他位置
PROGRESS	由于额外数据的传输导致php_stream_notifier结构体的progress以及(可能的)progress_max元素被更新(进度信息, 请参考php手册curl_setopt的CURLOPT_PROGRESSFUNCTION和CURLOPT_NOPROGRESS选项)
COMPLETED	流上没有更多的可用数据
FAILURE	请求的URL资源不成功或未完成
AUTH_RESULT	远程系统已经处理了授权认证

安全码	
INFO	信息更新. 等价于一个E_NOTICE错误
WARN	小的错误条件. 等价于一个E_WARNING错误
ERR	中断错误条件. 等价于一个E_ERROR错误.

通知器实现提供了一个便利指针*ptr用于存放额外数据. 这个指针指向的空间必须在上下文析构时被释放, 因此必须指定一个dtor函数, 在上下文的最后一个引用离开它的作用域时调用这个dtor进行释放.

mask元素允许事件触发限定特定的安全级别. 如果发生的事件没有包含在mask中, 则通知器函数不会被触发.

最后两个元素progress和progress_max可以由流实现设置, 然而, 通知器函数应该避免使用这两个值, 除非它接收到PHP_STREAM_NOTIFY_PROGRESS或PHP_STREAM_NOTIFY_FILE_SIZE_IS事件通知.

下面是一个php_stream_notification_func()回调原型的示例:

```
void php_sample6_notifier(php_stream_context *context,
        int notifycode, int severity, char *xmsg, int xcode,
        size_t bytes_sofar, size_t bytes_max,
        void *ptr TSRMLS_DC)
{
    if (notifycode != PHP_STREAM_NOTIFY_FAILURE) {
        /* 忽略所有通知 */
        return;
```

```
    }
    if (severity == PHP_STREAM_NOTIFY_SEVERITY_ERR) {
        /* 分发到错误处理函数 */
        php_sample6_theskyisfalling(context, xcode, xmsg);
        return;
    } else if (severity == PHP_STREAM_NOTIFY_SEVERITY_WARN) {
        /* 日志记录潜在问题 */
        php_sample6_logstrangeevent(context, xcode, xmsg);
        return;
    }
}
```

默认上下文

在php5.0中, 当用户空间的流创建函数被调用时, 如果没有传递上下文参数, 请求一般会使用默认的上下文. 这个上下文变量存储在文件全局结构中: FG(default_context), 并且它可以和其他所有的php_stream_context变量一样访问. 当在用户空间脚本执行流的创建时, 更好的方式是允许用户指定一个上下文或者至少指定一个默认的上下文. 将用户空间的zval *解码得到php_stream_context可以使用php_steram_context_from_zval()宏完成, 比如下面改编自第14章"访问流"的例子:

```
PHP_FUNCTION(sample6_fopen)
{
    php_stream *stream;
    char *path, *mode;
    int path_len, mode_len;
    int options = ENFORCE_SAFE_MODE | REPORT_ERRORS;
    zend_bool use_include_path = 0;
    zval *zcontext = NULL;
    php_stream_context *context;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
            "ss|br", &path, &path_len, &mode, &mode_len,
                &use_include_path, &zcontext) == FAILURE) {
        return;
    }
    context = php_stream_context_from_zval(zcontext, 0);
    if (use_include_path) {
        options |= PHP_FILE_USE_INCLUDE_PATH;
    }
    stream = php_stream_open_wrapper_ex(path, mode, options,
                                    NULL, context);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

如果zcontext包含一个用户空间的上下文资源, 通过ZEND_FETCH_RESOURCE()调用获取到它关联的指针设置到context中. 否则, 如果zcontext为NULL并且php_stream_context_from_zval()的第二个参数设置为非0值, 这个宏则直接返回NULL. 这个例子以及几乎所有的核心流创建的用户空间函数中, 第二个参数都被设置为0, 此时将使用FG(default_context)的值.

过滤器

过滤器作为读写操作的流内容传输过程中的附加阶段. 要注意的是直到php 4.3中才加入了流过滤器, 在php 5.0对流过滤器的API设计做过较大的调整. 本章的内容遵循的是php 5的流过滤器规范.

在流上应用已有的过滤器

在一个打开的流上应用一个已有的过滤器只需要几行代码即可:

```
php_stream *php_sample6_fopen_read_ucase(const char *path
                                         TSRMLS_DC) {
    php_stream_filter *filter;
    php_stream *stream;

    stream = php_stream_open_wrapper_ex(path, "r",
                        REPORT_ERRORS | ENFORCE_SAFE_MODE,
                        NULL, FG(default_context));
    if (!stream) {
        return NULL;
    }

    filter = php_stream_filter_create("string.toupper", NULL,
                                         0 TSRMLS_CC);
    if (!filter) {
        php_stream_close(stream);
        return NULL;
    }
    php_stream_filter_append(&stream->readfilters, filter);

    return stream;
}
```

首先来看看这里引入的API函数以及它的兄弟函数:

```
php_stream_filter *php_stream_filter_create(
                const char *filtername, zval *filterparams,
                int persistent TSRMLS_DC);
void php_stream_filter_prepend(php_stream_filter_chain *chain,
                php_stream_filter *filter);
void php_stream_filter_append(php_stream_filter_chain *chain,
                php_stream_filter *filter);
```

php_stream_filter_create()的filterparams参数和用户空间对应的stream_filter_append()和stream_filter_prepend()函数的同名参数含义一致. 要注意, 所有传递到php_stream_filter_create()的zval *数据都不是过滤器所拥有的. 它们只是在过滤器创建期间被借用而已, 因此在调用作用域分配传入的所有内存空间都要手动释放.

如果过滤器要被应用到一个持久化流, 则必须设置persistent参数为非0值. 如果你不确认你要应用过滤器的流是否持久化的, 则可以使用php_stream_is_persistent()宏进行检查, 它只接受一个php_stream *类型的参数.

如在前面例子中看到的, 流过滤器被隔离到两个独立的链条中. 一个用于写操作中对php_stream_write()调用响应时的stream->ops->write()调用之前. 另外一个用于读操作中对stream->ops->read()取回的所有数据进行处理.

在这个例子中你使用&stream->readfilters指示读的链条. 如果你想要在写的链条上应用一个过滤器, 则可以使用&stream->writefilters.

定义一个过滤器实现

注册过滤器实现和注册包装器遵循相同的基础规则. 第一步是在MINIT阶段向php中引入你的过滤器, 与之匹配的是在MSHUTDOWN阶段移除它. 下面是需要调用的API原型以及两个注册过滤器工厂的示例:

```
int php_stream_filter_register_factory(
            const char *filterpattern,
            php_stream_filter_factory *factory TSRMLS_DC);
int php_stream_filter_unregister_factory(
            const char *filterpattern TSRMLS_DC);
```

```
PHP_MINIT_FUNCTION(sample6)
{
    php_stream_filter_register_factory("sample6",
            &php_sample6_sample6_factory TSRMLS_CC);
    php_stream_filter_register_factory("sample.*",
            &php_sample6_samples_factory TSRMLS_CC);
    return SUCCESS;
}
PHP_MSHUTDOWN_FUNCTION(sample6)
{
    php_stream_filter_unregister_factory("sample6" TSRMLS_CC);
    php_stream_filter_unregister_factory("sample.*"
                                         TSRMLS_CC);
    return SUCCESS;
}
```

这里注册的第一个工厂定义了一个具体的过滤器名sample6; 第二个则利用了流包装层内部的基本匹配规则. 为了进行演示, 下面的用户空间代码, 每行都将尝试通过不同的名字实例化php_sample6_samples_factory.

```
<?php
    stream_filter_append(STDERR, 'sample.one');
    stream_filter_append(STDERR, 'sample.3');
    stream_filter_append(STDERR, 'sample.filter.thingymabob');
    stream_filter_append(STDERR, 'sample.whatever');
?>
```

php_sample6_samples_factory的定义如下面代码, 你可以将这些代码放到你的MINIT块上面:

```
#include "ext/standard/php_string.h"

typedef struct {
    char    is_persistent;
    char    *tr_from;
    char    *tr_to;
    int     tr_len;
} php_sample6_filter_data;

/* 过滤逻辑 */
static php_stream_filter_status_t php_sample6_filter(
        php_stream *stream, php_stream_filter *thisfilter,
        php_stream_bucket_brigade *buckets_in,
        php_stream_bucket_brigade *buckets_out,
        size_t *bytes_consumed, int flags TSRMLS_DC)
{
    php_stream_bucket       *bucket;
    php_sample6_filter_data *data       = thisfilter->abstract;
    size_t                  consumed    = 0;

    while ( buckets_in->head ) {
        bucket      = php_stream_bucket_make_writeable(buckets_in->head TSRMLS_CC);
        php_strtr(bucket->buf, bucket->buflen, data->tr_from, data->tr_to, data->tr_len);
        consumed    += bucket->buflen;
        php_stream_bucket_append(buckets_out, bucket TSRMLS_CC);
    }
    if ( bytes_consumed ) {
        *bytes_consumed = consumed;
    }
    return PSFS_PASS_ON;
}

/* 过滤器的释放 */
static void php_sample6_filter_dtor(php_stream_filter *thisfilter TSRMLS_DC)
{
    php_sample6_filter_data *data   = thisfilter->abstract;
```

```
    pefree(data, data->is_persistent);
}

/* 流过滤器操作表 */
static php_stream_filter_ops php_sample6_filter_ops = {
    php_sample6_filter,
    php_sample6_filter_dtor,
    "sample.*",
};

/* 字符翻译使用的表 */
#define PHP_SAMPLE6_ALPHA_UCASE     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define PHP_SAMPLE6_ALPHA_LCASE     "abcdefghijklmnopqrstuvwxyz"
#define PHP_SAMPLE6_ROT13_UCASE     "NOPQRSTUVWXYZABCDEFGHIJKLM"
#define PHP_SAMPLE6_ROT13_LCASE     "nopqrstuvwxyzabcdefghijklm"

/* 创建流过滤器实例的过程 */
static php_stream_filter *php_sample6_filter_create(
        const char *name, zval *param, int persistent TSRMLS_DC)
{
    php_sample6_filter_data *data;
    char                    *subname;

    /* 安全性检查 */
    if ( strlen(name) < sizeof("sample.") || strncmp(name, "sample.", sizeof("sample.") - 1) ) {
        return NULL;
    }

    /* 分配流过滤器数据 */
    data    = pemalloc(sizeof(php_sample6_filter_data), persistent);

    if ( !data ) {
        return NULL;
    }

    /* 设置持久性 */
    data->is_persistent = persistent;

    /* 根据调用时的名字, 对过滤器数据进行适当初始化 */
    subname = (char *)name + sizeof("sample.") - 1;
    if ( strcmp(subname, "ucase") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_LCASE;
        data->tr_to     = PHP_SAMPLE6_ALPHA_UCASE;
    } else if ( strcmp(subname, "lcase") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_UCASE;
        data->tr_to     = PHP_SAMPLE6_ALPHA_LCASE;
    } else if ( strcmp(subname, "rot13") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_LCASE
                          PHP_SAMPLE6_ALPHA_UCASE;;
        data->tr_to     = PHP_SAMPLE6_ROT13_LCASE
                          PHP_SAMPLE6_ROT13_UCASE;
    } else {
        /* 不支持 */
        pefree(data, persistent);
        return NULL;
    }

    /* 节省未来使用时每次的计算 */
    data->tr_len    = strlen(data->tr_from);

    /* 分配一个php_stream_filter结构并按指定参数初始化 */
    return php_stream_filter_alloc(&php_sample6_filter_ops, data, persistent);
}
```

```
/* 流过滤器工厂, 用于创建流过滤器实例(php_stream_filter_append/prepend的时候) */
static php_stream_filter_factory php_sample6_samples_factory = {
    php_sample6_filter_create
};
```

译注: 下面是译者对整个流程的分析

1. MINIT阶段的register操作将在stream_filters_hash这个HashTable中注册一个php_stream_filter_factory结构, 它只有一个成员create_filter, 用来创建过滤器实例.

2. 用户空间代码stream_filter_append(STDERR, 'sapmple.one');在内部的实现是apply_filter_to_stream()函数(ext/standard/streamsfuncs.c中), 这里有两步操作, 首先创建过滤器, 然后将过滤器按照参数追加到流的readfilters/writefilters相应链中;

2.1 创建过滤器(php_stream_filter_create()): 首先直接按照传入的名字精确的从stream_filters_hash(或FG(stream_filters))中查找, 如果没有, 从右向左替换句点后面的内容为星号"*"进行查找, 直到找到注册的过滤器工厂或错误返回. 一旦找到注册的过滤器工厂, 就调用它的create_filter成员, 创建流过滤器实例.

2.2 直接按照参数描述放入流的readfilters/writefilters相应位置.

3. 用户向该流进行写入或读取操作时(以写为例): 此时内部将调用_php_stream_write(), 在这个函数中, 如果流的writefilters非空, 则调用流过滤器的fops->filter()执行过滤, 并根据返回状态做相应处理.

4. 当流的生命周期结束, 流被释放的时候, 将会检查流的readfilters/writefilters是否为空, 如果非空, 相应的调用php_stream_filter_remove()进行释放, 其中就调用了fops->fdtor对流过滤器进行释放.

上一章我们已经熟悉了流包装器的实现, 你可能能够识别这里的基本结构. 工厂函数(php_sample6_samples_filter_create)被调用分配一个过滤器实例, 并赋值给一个操作集合和抽象数据. 这上面的例子中, 你的工厂为所有的过滤器类型赋值了相同的ops结构, 但使用了不同的初始化数据.

调用作用域将得到这里分配的过滤器, 并将它赋值给流的readfilters链或writefilters链. 接着, 当流的读/写操作被调用时, 过滤器链将数据放入到一个或多个php_stream_bucket结构体, 并将这些bucket组织到一个队列php_stream_bucket_brigade中传递给过滤器.

这里, 你的过滤器实现是前面的php_sample6_filter, 它取出输入队列bucket中的数据, 使用php_sample6_filter_create中确定的字符表执行字符串翻译, 并将修改后的bucket放入到输出队列.

由于这个过滤器的实现并没有其他内部缓冲, 因此几乎不可能出错, 因此它总是返回PSFS_PASS_ON, 告诉流包装层有数据被过滤器存放到了输出队列中. 如果过滤器执行了内部缓冲消耗了所有的输入数据而没有产生输出, 就需要返回PSFS_FEED_ME标识过滤器循环周期在没有其他输入数据时暂时停止. 如果过滤器碰到了关键性的错误, 它应该返回PSFS_ERR_FATAL, 它将指示流包装层, 过滤器链处于不稳定状态. 这将导致流被关闭.

用于维护bucket和bucket队列的API函数如下:

```
php_stream_bucket *php_stream_bucket_new(php_stream *stream,
                      char *buf, size_t buflen, int own_buf,
                      int buf_persistent TSRMLS_DC);
```

创建一个php_stream_bucket用于存放到输出队列. 如果own_buf被设置为非0值, 流包装层可以并且通常都会修改它的内容或在某些点释放分配的内存. buf_persistent的非0值标识buf使用的内存是否持久分配的:

```
int php_stream_bucket_split(php_stream_bucket *in,
        php_stream_bucket **left, php_stream_bucket **right,
        size_t length TSRMLS_DC);
```

这个函数将in这个bucket的内容分离到两个独立的bucket对象中. left这个bucket将包含in中的前length个字符, 而right则包含剩下的所有字符.

```
void php_stream_bucket_delref(php_stream_bucket *bucket
                                                TSRMLS_DC);
void php_stream_bucket_addref(php_stream_bucket *bucket);
```

Bucket使用和zval以及资源相同的引用计数系统. 通常, 一个bucket仅属于一个上下文, 也就是它依附的队列.

```
void php_stream_bucket_prepend(
                    php_stream_bucket_brigade *brigade,
                    php_stream_bucket *bucket TSRMLS_DC);
void php_stream_bucket_append(
        php_stream_bucket_brigade *brigade,
        php_stream_bucket *bucket TSRMLS_DC);
```

这两个函数扮演了过滤器子系统的苦力, 用于附加bucket到队列的开始(prepend)或末尾(append)

```
void php_stream_bucket_unlink(php_stream_bucket *bucket
                                                TSRMLS_DC);
```

在过滤器逻辑应用处理完成后, 旧的bucket必须使用这个函数从它的输入队列删除(unlink).

```
php_stream_bucket *php_stream_bucket_make_writeable(
        php_stream_bucket *bucket TSRMLS_DC);
```

将一个bucket从它所依附的队列中移除, 并且如果需要, 赋值bucket->buf的内部缓冲区, 这样就使得它的内容可修改. 在某些情况下, 比如当输入bucket的引用计数大于1时, 返回的bucket将会是不同的实例, 而不是传入的实例. 因此, 我们要保证在调用作用域使用的是返回的bucket, 而不是传入的bucket.

小结

过滤器和上下文可以让普通的流类型行为被修改, 或通过INI设置影响整个请求, 而不需要直接的代码修改. 使用本章设计的计数, 你可以使你自己的包装器实现更加强大, 并且可以对其他包装器产生的数据进行改变.

接下来, 我们将离开PHPAPI背后的工作, 回到php构建系统的机制, 产生更加复杂的扩展链接到其他应用, 找到更加容易的方法, 使用工具集处理重复的工作.

配置和链接

所有前面示例中的代码, 都是你曾经在php用户空间编写过代码的C语言的独立版本. 如果你做的项目需要和php扩展进行粘合, 那么你就至少需要链接一个外部库.

autoconf

在一个简单的应用中, 你可能已经在你的Makefile中增加了下面这样的CFLAGS和LDFLAGS.

```
CFLAGS = ${CFLAGS} -I/usr/local/foobar/include
LDFLAGS = ${LDFLAGS} -lfoobar -L/usr/local/foobar/lib
```

想要构建你的应用却没有libfoobar的人, 或将libfoobar安装到其他位置的人, 将会得到一个处理过的错误消息, 用于帮助他找到错误原因.

在过去十年开发的多数开发源代码软件(OSS)以及PHP都利用了一个实用工具autoconf, 通过一些简单的宏来生成复杂的configure脚本. 这个产生的脚本会执行查找依赖库已经头文件是否安装的工作. 基于这些信息, 一个包可以自定义构建代码行, 或在编译的时间被浪费之前提供一个有意义的错误消息.

在构建php扩展时, 无论你是否计划公开发布, 都需要利用这个autoconf机制. 即便你对autoconf已经很熟悉了, 也请花几分钟时间阅读本章, php中引入了一些一般安装的autoconf没有的自定义宏.

和传统的autoconf步骤(集中的configure.in文件包含了包的所有配置宏)不同, php只是用configure.in管理许多位域源码树下小的config.m4脚本的协调, 包括各个扩展, SAPI, 核心自身, 以及ZendEngine.

你已经在前面的章节看到了一个简单版本的config.m4. 接下来, 我们将在这个文件中增加其他的autoconf语法, 让你的扩展可以收集到更多的配置时信息.

库的查找

config.m4脚本最多是用于检查依赖库是否已安装. 扩展比如mysql, ldap, gmp以及其他设计为php用户空间和c库实现的其他功能之间的粘合层的扩展. 如果它们的依赖库没有安装, 或者安装的版本太旧, 要么会编译错误, 要么会导致产生的二进制无法运行.

头文件扫描

对依赖库扫描中最简单的一步就是检查你的脚本中的包含文件, 它们将在链接时使用. 下面的代码尝试在一些常见位置查找zlib.h:

```
PHP_ARG_WITH(zlib,[for zlib Support]
[  with-zlib              Include ZLIB Support])

if test "$PHP_ZLIB" != "no"; then
  for i in /usr /usr/local /opt; do
    if test -f $i/include/zlib/zlib.h; then
      ZLIB_DIR=$i
    fi
  done

  if test -z "$ZLIB_DIR"; then
    AC_MSG_ERROR([zlib not installed (http://www.zlib.org)])
  fi

  PHP_ADD_LIBRARY_WITH_PATH(z,$ZLIB_DIR/lib, ZLIB_SHARED_LIBADD)
  PHP_ADD_INCLUDE($ZLIB_DIR/include)

  AC_MSG_RESULT([found in $ZLIB_DIR])
```

```
  AC_DEFINE(HAVE_ZLIB,1,[libz found and included])

  PHP_NEW_EXTENSION(zlib, zlib.c, $ext_shared)
  PHP_SUBST(ZLIB_SHARED_LIBADD)
fi
```

config.m4文件很明显比你迄今为止使用的要大. 幸运的是, 它的语法非常的简单易懂并且如果你熟悉bash脚本, 对它也就不会陌生.

文件和第5章"你的第一个扩展"中第一次出现的一样, 都是以PHP_ARG_WITH()宏开始. 这个宏的行为和你用过的PHP_ARG_ENABLE()宏一样, 不过它将导致./configure时的选项是--with-extname/--without-extname而不再是--enable-extname/--disable-extname.

回顾这些宏, 它们的功能是等同的, 不同仅在于是否让终端用户给你的包一些暗示. 你可以在自己创建的私有扩展上使用任意一种方式. 不过, 如果你计划公开发布, 那就应该知道php正式的编码标准, 它指出enable/disable用于哪些不需要链接外部库的扩展, with/without则反之.

由于我们这里假设的扩展将链接zlib库, 因此你的config.m4脚本要以查找扩展源代码中将包含的zlib.h头文件. 这通过检查一些标准位置/usr, /usr/local, /opt中include/zlib目录下的zlib.h完成对其下两个目录的定位.

如果找到了zlib.h, 则将基路径设置到临时变量ZLIB_DIR中. 一旦循环完成, config.m4脚本检查ZLIB_DIR是否包含内容来确定是否找到了zlib.h. 如果没有找到, 则产生一个有意义的错误让用户知道./configure不能继续.

此刻, 脚本假设头文件存在, 对应的库也必须存在, 因此在下面的两行使用它修改构建环境, 最终增加-lz -L$ZLIB_DIR/lib到LDFLAGS以及-I$ZLIB_DIR/include到CFLAGS.

最终, 输出一个确认消息指示zlib安装已经找到, 并且在编译期间使用它的路径. config.m4的其他部分从前面部分的学习中你应该已经熟悉了. 为config.h定义一个#define, 定义扩展并指定它的源代码文件, 同时标识一个变量完成将扩展附加到构建系统的工作.

功能测试

迄今为止, 这个config.m4示例指示查找了需要的头文件. 尽管这已经够用了, 但它仍然不能确保产生的二进制正确的进行链接, 因为可能不存在匹配的库文件, 或者版本不正确.

最简单的检查zlib.h对应的libz.so库文件是否存在的方式就是检查文件是否存在:

```
if ! test -f $ZLIB_DIR/lib/libz.so; then
  AC_MSG_ERROR([zlib.h found, but libz.so not present!])
fi
```

当然, 这仅仅是问题的一面. 如果安装了其他的同名库但和你要查找的库不兼容怎么办呢? 确保你的扩展可以成功编译的最好方式是测试找到的库实际编译所需的内容. 要这样做就需要在config.m4中PHP_ADD_LIBRARY_WITH_PATH调用之前加入下面代码:

```
PHP_CHECK_LIBRARY(z, deflateInit,,[
  AC_MSG_ERROR([Invalid zlib extension, gzInit() not found])
],-L$ZLIB_DIR/lib)
```

这个工具宏将展开输出一个完整的程序, ./configure将尝试编译它. 如果编译成功, 表示第二个参数定义的符号在第一个参数指定的库中存在. 成功后, 第三个参数中指定的autoconf脚本将会执行; 失败后, 第四个参数中指定的autoconf脚本将执行. 在这个例子中, 第三个参数为空, 因为没有消息就是最好的消息(译注: 应该是unix哲学之一), 第五个参数也就是左后一个参数, 用于指定额外的编译器和链接器标记, 这里, 使用-L致命了一个额外的用于查找库的路径.

可选功能

那么现在你已经有正确的库和头文件了, 但依赖的是所安装库的哪个版本呢? 你可能需要某些功能或排斥某些功能. 由于这种类型的变更通常涉及到某些特定入口点的增加或删除, 因此可以重用PHP_CHECK_LIBRARY()宏来检查库的某些能力.

```
PHP_CHECK_LIBRARY(z, gzgets,[
  AC_DEFINE(HAVE_ZLIB_GETS,1,[Having gzgets indicates zlib >= 1.0.9])
],[
  AC_MSG_WARN([zlib < 1.0.9 installed, gzgets() will not be available])
],-L$ZLIB_DIR/lib)
```

测试实际行为

可能知道某个符号存在也还不能确保你的代码正确编译; 某些库的特定版本可能存在bug需要运行一些测试代码进行检查.

AC_TRY_RUN()宏可以编译一个小的源代码文件为可执行程序并执行. 依赖于传回给./configure的返回代码, 你的脚本可以设置可选的#define语句或直接输出消息(比如如果bug导致不能工作则提示升级)安全退出. 考虑下面的代码(摘自ext/standard/config.m4):

```
AC_TRY_RUN([
#include <math.h>

double somefn(double n) {
  return floor(n*pow(10,2) + 0.5);
}
int main() {
  return somefn(0.045)/10.0 != 0.5;
}
],[
  PHP_ROUND_FUZZ=0.5
  AC_MSG_RESULT(yes)
],[
  PHP_ROUND_FUZZ=0.50000000001
  AC_MSG_RESULT(no)
],[
  PHP_ROUND_FUZZ=0.50000000001
  AC_MSG_RESULT(cross compile)
])
AC_DEFINE_UNQUOTED(PHP_ROUND_FUZZ, $PHP_ROUND_FUZZ,
                   [Is double precision imprecise?])
```

你可以看到, AC_TRY_RUN()的第一个参数是一块C语言代码, 它将被编译执行. 如果这段代码的退出代码是0, 位于第二个参数的autoconf脚本将被执行, 这种情况标识round()函数和期望一样工作, 返回0.5.

如果代码块返回非0值, 位域第三个参数的autoconf脚本将被执行. 第四个参数(最后一个)在php交叉编译时使用. 这种情况下, 尝试运行示例代码是没有意义的, 因为目标平台不同于扩展编译时使用的平台.

强制模块依赖

在php 5.1中, 扩展之间的内部依赖是可以强制性的. 由于扩展可以静态构建到php中, 也可以构建为共享对象动态加载, 因此强制依赖需要在两个地方实现.

配置时模块依赖

第一个位置是你在本章课程中刚刚看到的config.m4文件中. 你可以使用PHP_ADD_EXTENSION_DEP(extname, depname[, optional])宏标识extname这个扩展依赖于depname这个扩展. 当extname以静态方式构建到php中时, ./configure脚本将使用

这一行代码确认depname必须首先初始化. optional参数是一个标记, 用来标识depname如果也是静态构建的, 应该在extname之前加载, 不过它并不是必须的依赖.

这个宏的一个使用示例是pdo驱动, 比如pdo_mysql是可预知依赖于pdo扩展的:

```
ifdef([PHP_ADD_EXTENDION_DEP],
[
  PHP_ADD_EXTENSION_DEP(pdo_mysql, pdo)
])
```

要注意PHP_ADD_EXTENSION_DEP()宏被包裹到一个ifdef()结构中. 这是因为pdo和它的驱动在编译大于或等于5.0版本的php时都是存在的, 然而PHP_ADD_EXTENSION_DEP()宏是直到5.1.0版本才出现的.

运行时模块依赖

另外一个你需要注册依赖的地方是zend_module_entry结构体中. 考虑下面第5章中你定义的zend_module_entry结构体:

```
zend_module_entry sample_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE_EXTNAME,
    php_sample_functions,
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};
```

增加运行时模块依赖信息就需要对STANDARD_MOUDLE_HEADER部分进行一些小修改:

```
zend_module_entry sample_module_entry = {
#if ZEND_MODULE_API_NO >= 220050617
    STANDARD_MODULE_HEADER_EX, NULL,
    php_sample_deps,
#elif ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE_EXTNAME,
    php_sample_functions,

    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};
```

现在, 如果ZEND_MODULE_API_NO高于php 5.1.0 beta发布版, 则STANDARD_MODULE_HEADER(译注: 这里原著笔误为STANDARD_MODULE_PROPERTIES)将被替换为略微复杂的结构, 它将包含一个指向模块依赖信息的引用.

这个目标结构体可以在你的zend_module_entry结构体上面定义如下:

```
#if ZEND_MODULE_API_NO >= 220050617
```

```
static zend_module_dep php_sample_deps[] = {
    ZEND_MODULE_REQUIRED("zlib")
    {NULL,NULL,NULL}
};
#endif
```

和zend_function_entry向量类似, 这个列表可以有多项依赖, 按照顺序进行检查. 如果尝试加载某个依赖模块未满足, Zend将会中断加载, 报告不满足依赖的名字, 这样, 终端用户就可以通过首先加载其他模块来解决问题.

Windows方言

由于译者对*windows*环境不熟悉, 因此略过本节.

小结

如果你的扩展将在未知或不可控制的环境构建, 让它足够聪明以应付奇怪的环境就非常重要. 使用php提供的unix和windows上强有力的脚本能力, 你应该可以检测到麻烦并在未知的管理员需要电话求助之前给于她一个解决方案.

现在你已经有使用php api从头建立php扩展的基础能力了, 你可以准备学习一下使用php提供的扩展开发工具把自己从繁重的重复劳动中解放出来了, 使用它们可以快速, 准确的建立新扩展的原型.

扩展生成

毫无疑问你已经注意到, 每个php扩展都包含一些非常公共的并且非常单调的结构和文件. 当开始一个新扩展开发的时候, 如果这些公共的结构已经存在, 我们只用考虑填充功能代码是很有意义的. 为此, 在php中包含了一个简单但是很有用的shell脚本.

ext_skel

切换到你的php源代码树下ext/目录中, 执行下面的命令:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ ./ext_skel extname=sample7
```

稍等便可, 输出一些文本, 你将看到下面的这些输出:

```
To use your new extension, you will have to execute the following steps:

1.  $ cd ..
2.  $ vi ext/sample7/config.m4
3.  $ ./buildconf
4.  $ ./configure [with|enable]-sample7
5.  $ make
6.  $ ./php -f ext/sample7/sample7.php
7.  $ vi ext/sample7/sample7.c
8.  $ make

Repeat steps 3-6 until you are satisfied with ext/sample7/config.m4 and
step 6 confirms that your module is compiled into PHP. Then, start writing
code and repeat the last two steps as often as necessary.
```

此刻观察ext/sample7目录, 你将看到在第5章"你的第一个扩展"中你编写的扩展骨架代码的注释版本. 只是现在你还不能编译它; 不过只需要对config.m4做少许修改就可以让它工作了, 这样你就可以避免第5章中你所做的大部分工作.

生成函数原型

如果你要编写一个对第三方库的包装扩展, 那么你就已经有了一个函数原型及基本行为的机器刻度版本的描述(头文件), 通过传递一个额外的参数给./ext_skel, 它将自动的扫描你的头文件并创建对应于接口的简单PHP_FUCNTION()块. 下面是使用./ext_skel指令解析zlib头:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ ./ext_skel extname=sample8 \
proto=/usr/local/include/zlib/zlib.h
```

现在在ext/sample8/sample8.c中, 你就可以看到许多PHP_FUNCTION()定义, 每个zlib函数对应一个. 要注意, 骨架生成程序会对某些未知资源类型产生警告消息. 你需要对这些函数特别注意, 并且为了将这些内部的复杂结构体和用户空间可访问的变量关联起来, 可能会需要使用你在第9章"资源数据类型"中学到的知识.

PECL_Gen

还有一种更加完善但也更加复杂的代码生成器: PECL_Gen, 可以在PECL(http://pecl.php.net)中找到它, 使用pear install PECL_Gen命令可以安装它.

译者注: PECL_Gen已经迁移为CodeGen_PECL(http://pear.php.net/package/CodeGen_PECL). 本章涉及代码测试使用CodeGen_PECL的版本信息为: "php 1.1.3, Copyright (c) 2003-2006 Hartmut Holzgraefe", 如果您的环境使用有问题, 请参考译序中译者的环境配置.

一旦安装完成, 它就可以像ext_skel一样运行, 接受相同的输入参数, 产生大致相同的输出, 或者如果提供了一个完整的xml定义文件, 则产生一个更加健壮和完整可编译版本的扩展. PECL_Gen并不会节省你编写扩展核心功能的时间; 而是提供一种可选的方式高效的生成扩展骨架代码.

specfile.xml

下面是最简单的扩展定义文件:

```
<?xml version="1.0" encoding="utf-8" ?>
<extension name="sample9">
 <functions>
  <function name="sample9_hello_world" role="public">
   <code>
<![CDATA[

    php_printf("Hello World!");
]]>
   </code>
  </function>
 </functions>
</extension>
```

译注: 请注意, 译者使用的原著中第一行少了后面的问号, 导致不能使用, 加上就*OK*.

通过PECL_Gen命令运行这个文件:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ pecl-gen specfile.xml
```

则会产生一个名为sample9的扩展, 并暴露一个用户空间函数sample9_hello_world().

关于扩展

除了你已经熟悉的功能文件, PECL_Gen还会产生一个package.xml文件 它可以用于pear安装. 如果你计划发布包到PECL库, 或者哪怕你只是想要使用pear包系统交付内容, 有这个文件都会很有用.

总之, 你可以在PECL_Gen的specfile.xml中指定多数package.xml文件的元素.

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension name="sample9">
    <summary>Extension 9 generated by PECL_Gen</summary>
    <description>Another sample of PHP Extension Writing</description>
    <maintainers>
        <maintainer>
            <name>John D. Bookreader</name>
            <email>jdb@example.com</email>
            <role>lead</role>
        </maintainer>
    </maintainers>
    <release>
        <version>0.1</version>
        <date>2006-01-01</date>
        <state>beta</state>
        <notes>Initial Release</notes>
    </release>
    ...
</extension>
```

当PECL_Gen创建扩展时, 这些信息将被翻译到最终的package.xml文件中.

依赖

如你在第17章"配置和链接"中所见, 依赖可以扫描出来用于config.m4和config.w32文件. PECL_Gen可以使用<deps>定义各种类型的依赖完成扫描工作. 默认情况下, 列在<deps>标签下的依赖会同时应用到Unix和win32构建中, 除非显式的是否用platform属性指定某个目标

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension name="sample9">
    ...
    <deps platform="unix">
        <! UNIX specific dependencies >
```

```
    </deps>
    <deps platform="win32">
        <! Win32 specific dependencies >
    </deps>
    <deps platform="all">
        <! Dependencies that apply to all platforms >
    </deps>
    ...
</extension>
```

with

通常, 扩展在配置时使用--enable-extname样式的配置选项. 通过增加一个或多个<with>标签到<deps>块中, 则不仅配置选项被修改为--with-extname, 而且同时需要扫描头文件:

```
<deps platform="unix">
    <with defaults="/usr:/usr/local:/opt"
        testfile="include/zlib/zlib.h">zlib headers</with>
</deps>
```

库

必须的库也列在<deps>下, 使用<lib>标签.

```
<deps platform="all">
    <lib name="ssleay" platform="win32"/>
    <lib name="crypto" platform="unix"/>
    <lib name="z" platform="unix" function="inflate"/>
</deps>
```

在前面两个例子中, 只是检查了库是否存在; 第三个例子中, 库将被真实的加载并扫描以确认inflate()函数是否定义.

尽管*<deps>*标签实际已经命名了目标平台, 但*<lib>*标签也有一个*platform*属性可以覆盖*<deps>*标签的*platform*设置. 当它们混合使用的时候要格外小心.

<header>

此外, 需要包含的文件也可以通过在<deps>块中使用<header>标签在你的代码中追加一个#include指令列表. 要强制某个头先包含, 可以在<header>标签上增加属性prepend="yes". 和<lib>依赖类似, <header>也可以严格限制平台:

```
<deps>
    <header name="sys/types.h" platform="unix" prepend="yes"/>
    <header name="zlib/zlib.h"/>
</deps>
```

译注*:* 经测试*,* 译者的环境*<header>*标签不支持*platform*属性*.*

常量

用户空间常量使用<constants>块中的一个或多个<constant>标签定义. 每个标签需要一个name和一个value属性, 以及一个值必须是int, float, string之一的type属性.

```
    <constants>
        <constant name="SAMPLE9_APINO" type="int" value="20060101"/>
        <constant name="SAMPLE9_VERSION" type="float" value="1.0"/>
        <constant name="SAMPLE9_AUTHOR" type="string" value="John Doe"/>
    </constants>
```

全局变量

线程安全全局变量的定义方式几乎相同. 唯一的不同在于type参数需要使用C语言原型而不是php用户空间描述. 一旦定义并构建, 全局变量就可以使用第12章"启动, 终止, 以及其中的一些点"中学习的EXTNAME_G(global_name)的宏用法进行访问. 在这里, value

属性表示变量在请求启动时的默认值. 要注意在specfile.xml中这个默认值只能指定为简单的标量数值. 字符串和其他复杂结构应该在RINIT阶段手动设置.

```
    <globals>
        <global name="greeting" type="char *"/>
        <global name="greeting_was_issued" type="zend_bool" value="1"/>
    </globals>
```

INI选项

要绑定线程安全的全局变量到php.ini设置, 则需要使用<phpini>标签而不是<globa>. 这个标签需要两个额外的参数: onupdate="updatemethod"标识INI的修改应该怎样处理, access="mode"和第13章"INI设置"中介绍的模式含义相同, "mode"值可以是: all, user, perdir, system.

```
    <globals>
        <phpini name="mysetting" type="int" value="42" onupdate="OnUpdateLong" access="all"/>
    </globals>
```

函数

你已经看到了最基本的函数定义; 不过, <function>标签在PECL_Gen的specfile中实际上支持两种不同类型的函数.

两个版本都支持你已经在<extension>级别上使用过的<summary>和<description>属性; 两种类型都必须的元素是<code>标签, 它包含了将要被放入你的源代码文件中的原文C语言代码.

role="public"

如你所想, 所有定义为public角色的函数都将包装恰当的PHP_FUNCTION()头和花括号, 对应到扩展的函数表向量中的条目.

除了其他函数支持的标签, public类型还允许指定一个<proto>标签. 这个标签的格式应该匹配php在线手册中的原型展示, 它将被文档生成器解析.

```
    <functions>
        <function role="public" name="sample9_greet_me">
            <summary>Greet a person by name</summary>
            <description>Accept a name parameter as a string and say hello to that person.
Returns TRUE.</description>
            <proto>bool sample9_greet_me(string name)</proto>
            <code>
            <![CDATA[
            char *name;
            int name_len;

            if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
                        &name, &name_len) == FAILURE) {
                return;
            }

            php_printf("Hello ");
            PHPWRITE(name, name_len);
            php_printf("!\n");
            RETURN_TRUE;
            ]]>
            </code>
        </function>
    </functions>
```

role="internal"

内部函数涉及5个zend_module_entry函数: MINIT, MSHUTDOWN, RINIT, RSHUTDOWN, MINFO. 如果指定的名字不是这5个之一将会产生pecl-gen无法处理的错误.

```
<functions>
    <function role="internal" name="MINFO">
        <code>
        <![CDATA[
        php_info_print_table_start();
        php_info_print_table_header(2, "Column1", "Column2");
        php_info_print_table_end();
        ]]>
        </code>
    </function>
</functions>
```

自定义代码

所有其他需要存在于你的扩展中的代码都可以使用<code>标签包含. 要放置任意代码到你的目标文件extname.c中, 使用role="code"; 或者说使用role="header"将代码放到目标文件php_extname.h中. 默认情况下, 代码将放到代码或头文件的底部, 除非指定了position="top"属性.

```
<code role="header" position="bottom">
<![CDATA[
typedef struct _php_sample9_data {
    long val;
} php_sample9_data;
]]>
</code>
<code role="code" position="top">
<![CDATA[
static php_sample9_data *php_sample9_data_ctor(long value)
{
    php_sample9_data *ret;
    ret = emalloc(sizeof(php_sample9_data));
    ret->val = value;
    return ret;
}
]]>
</code>
```

译注: 译者的环境中不支持原著中*<code>*标签的*name*属性.

小结

使用本章讨论的工具, 你就可以快速的开发php扩展, 并且让你的代码相比手写更加不容易产生bug. 现在是时候转向将php嵌入到其他项目了. 剩下的章节中, 你将利用php环境和强大的php引擎为你的已有项目增加脚本能力, 使它可以为你的客户提供更多更有用的功能.

设置宿主环境

现在你已经了解了PHPAPI的世界, 并可以使用zval以及语言内部扩展机制执行很多工作了, 是时候转移目标用它做它最擅长的事情了: 解释脚本代码.

嵌入式SAPI

回顾介绍中, php构建了一个层级系统. 最高层是提供用户空间函数和类库的所有扩展. 同时, 其下是服务API(SAPI)层, 它扮演了webserver(比如apache, iis以及命令行接口cli)的接口.

在这许多sapi实现中有一个特殊的sapi就是嵌入式sapi. 当这个sapi实现被构建时, 将会创建一个包含所有你已知的php和zend api函数以及变量的库对象, 这个库对象还包含一些额外的帮助函数和宏, 用以简化外部程序的调用.

生成嵌入式api的库和头文件和其他sapi的编译所执行的动作相同. 只需要传递--enable-embed到./configure命令中即可. 和以前一样, 使用--enable-debug对于错误报告和跟踪很有帮助.

你可能还需要打开--enable-maintainer-zts, 当然, 理由你已经耳熟能详了, 它将帮助你注意到代码的错误, 不过, 这里还有其他原因. 假设某个时刻, 你有多个应用使用php嵌入库执行脚本任务; 其中一个应用是简单的短生命周期的, 它并没有使用线程, 因此为了效率你可能想要关闭ZTS.

现在假设第二个应用使用了线程, 比如webserver, 每个线程需要跟踪自己的请求上下文. 如果ZTS被关闭, 则只有第一个应用可以使用这个库; 然而, 如果打开ZTS, 则两个应用都可以在自己的进程空间使用同一个共享对象.

当然, 你也可以同时构建两个版本, 并给它们不同的名字, 但是这相比于在不需要ZTS时包括ZTS带来的很小的效率影响更多的问题.

默认情况下, 嵌入式库将构建为libphp5.so共享对象, 或者在windows下的动态链接库, 不过, 它也可能使用可选的static关键字(--enable-embed=static)被构建为静态库.

构建为静态库的版本避免了ZTS/非ZTS的问题, 以及潜在的可能在一个系统中有多个php版本的情况. 风险在于这就意味着你的结果应用二进制将显著变大, 它将承载整个ZendEngine和PHP框架, 因此, 选择的时候就需要慎重的考虑你是否需要的是一个相对更小的库.

无论你选择那种构建方式, 一旦你执行make install, libphp5都将被拷贝到你的./configure指定的PREFIX目录下的lib/目录中. 此外还会在PREFIX/include/php/sapi/embed目录下放入名为php_embed.h的头文件, 以及你在使用php嵌入式库编译程序时需要的其他几个重要的头文件.

构建并编译一个宿主应用

究其本质而言, 库只是一个没有目的的代码集合. 为了让它工作, 你需要用以嵌入php的应用. 首先, 我们来封装一个非常简单的应用, 它启动Zend引擎并初始化PHP处理一个请求, 接着就回头进行资源的清理.

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

int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc,argv)
    PHP_EMBED_END_BLOCK()

    return 0;
}
```

由于这涉及到了很多头文件, 构建实际上需要的时间要长于这么小的代码片段通常需要的时间. 如果你使用了不同于默认路径(/usr/local)的PREFIX, 请确认以下面的方式指定路径:

```
gcc -I /usr/local/php-dev/include/php/ \
    -I /usr/local/php-dev/include/php/main/ \
    -I /usr/local/php-dev/include/php/Zend/ \
    -I /usr/local/php-dev/include/php/TSRM/ \
    -lphp5 \
    -o embed1
    embed1.c
```

由于这个命令每次输入都很麻烦, 你可能更原意用一个简单的Makefile替代:

```
CC = gcc
CFLAGS = -c \
    -I /usr/local/php-dev/include/php/ \
    -I /usr/local/php-dev/include/php/main/ \
    -I /usr/local/php-dev/include/php/Zend/ \
    -I /usr/local/php-dev/include/php/TSRM/ \
    -Wall -g
LDFLAGS = -lphp5

all: embed1.c
    $(CC) -o embed1.o embed1.c $(CFLAGS)
    $(CC) -o embed1 embed1.o $(LDFLAGS)
```

这个*Makefile*和前面提供的命令有一些重要的区别. 首先, 它用*-Wall*开关打开了编译期的警告, 并且用*-g*打开了调试信息. 此外它将编译和链接两个阶段分为了两个独立的阶段, 这样在后期增加更多源文件的时候就相对容易. 请自己重新组着这个*Makefile,* 不过这里用于对齐的是*Tab(*水平制表符*)*而不是空格.

现在, 你对embed1.c源文件做修改后, 只需要执行一个make命令就可以构建出新的embed1可执行程序了.

通过嵌入包装重新创建cli

现在php已经可以在你的应用中访问了, 是时候让它做一些事情了. 本章剩下的核心就是围绕着在这个测试应用框架中重新创建cli sapi展开的.

很简单, cli二进制程序最基础的功能就是在命令行指定一个脚本的名字, 由php对其解释执行. 用下面的代码替换你的embed1.c的内容就在你的应用中实现了cli.

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

int main(int argc, char *argv[]) {
    zend_file_handle    script;

    /* 基本的参数检查 */
    if ( argc <= 1 ) {
        fprintf(stderr, "Usage: %s <filename.php> <arguments>\n", argv[0]);
        return -1;
    }

    /* 设置一个文件处理结构 */
    script.type             = ZEND_HANDLE_FP;
    script.filename         = argv[1];
    script.opened_path      = NULL;
    script.free_filename    = 0;
    if ( !(script.handle.fp = fopen(script.filename, "rb")) ) {
        fprintf(stderr, "Unable to open: %s\n", argv[1]);
        return -1;
    }

    /* 在将命令行参数注册给php时(php中的$argv/$argc), 忽略第一个命令行参数, 因为它对php脚本无意义 */
```

```
    argc --;
    argv ++;

    PHP_EMBED_START_BLOCK(argc, argv)
        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()

    return 0;
}
```

译注: 原著中的代码在译者的环境不能直接运行, 上面的代码是经过修改的.

当然, 你需要一个文件测试它, 创建一个小的php脚本, 命名为test.php, 在命令行使用你的embed程序执行它:

```
$ ./embed1 test.php
```

如果你给命令行传递了其他参数, 你可以在你的php脚本中使用$_SERVER['argc']/$_SERVER['argv']看到它们.

你可能注意到了, 在*PHP_EMBED_START_BLOCK()*和*PHP_EMBED_END_BLOCK()*之间的代码是缩进的. 这个细节是因为这两个宏实际上构成了一个*C*语言的代码块作用域. 也就是说*PHP_EMBED_START_BLOCK()*包含一个打开的花括号*"{"*, 在*PHP_EMBED_END_BLOCK()*中则有与之对应的关闭花括号*"}"*. 这样做非常重要的一个问题是它们不能被放入到独立的启动/终止函数中. 下一章你将看到这个问题的解决方案.

老技术新用

在PHP_EMBED_START_BLOCK()被调用后, 你的应用处于一个php请求周期的开始位置, 相当于RINIT回调函数完成以后. 此刻你就可以和前面一样执行php_execute_script()命令, 或者其他任意合法的, 可以在PHP_FUNCTION()或RINIT()块中出现的php/Zend API指令.

设置初始变量

第2章"变量的里里外外"中介绍了操纵符号表的概念, 第5至18章则介绍了怎样通过用户空间脚本调用内部函数使用这些技术. 到这里这些处理也并没有发生变化, 虽然这里并没有激活的用户空间脚本, 但是你的包装应用仍然可以操纵符号表. 将你的PHP_EMBED_START_BLOCK()/PHP_EMBED_END_BLOCK()代码块替换为下面的代码:

```
    PHP_EMBED_START_BLOCK(argc, argv)
        zval    *type;

        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(&EG(symbol_table), "type", type);

        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()
```

现在使用make重新构建embed1, 并用下面的测试脚本进行测试:

```
<?php
    var_dump($type);
?>
```

当然, 这个简单的概念可以很容易的扩展为填充这个类型信息到$_SERVER超级全局变量数组中.

```
    PHP_EMBED_START_BLOCK(argc, argv)
        zval    **SERVER_PP, *type;

        /* 注册$_SERVER超级全局变量 */
        zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
        /* 查找$_SERVER超级全局变量 */
```

```
        zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP) ;

        /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);

        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()
```

译注: 译者的环境中代码运行到*zend_hash_find()*处*$_SERVER*尚未注册, 经过跟踪, 发现它是直到编译用户空间代码的时候, 发现用户空间使用了*$_SERVER*变量才进行的注册. 因此, 上面的代码中增加了*zend_is_auto_global_quick()*的调用, 通过这个调用将完成对*$_SERVER*的注册.

覆写INI选项

在第13章"INI设置"中, 有一部分是讲INI修改处理器的, 在那里看到的是INI阶段的处理. PHP_EMBED_START_BLOCK()宏则将这些代码放到了运行时阶段. 也就是说这个时候修改某些设置(比如register_globals/magic_quotes_gpc)已经有点迟了.

不过在内部访问也没有什么不好. 所谓的"管理设置"比如safe_mode在这个略迟的阶段可以使用下面的zend_alter_ini_entry()命令打开或关闭:

```
int zend_alter_ini_entry(char *name, uint name_length,
                         char *new_value, uint new_value_length,
                         int modify_type, int stage);
```

name, new_value以及它们对应的长度参数的含义正如你所预期的: 修改名为name的INI设置的值为new_value. 要注意name_length包含了末尾的NULL字节, 然而new_value_length则不包含; 然而, 无论如何, 两个字符串都必须是NULL终止的.

modify_type则提供简化的访问控制检查. 回顾每个INI设置都有一个modifiable属性, 它是PHP_INI_SYSTEM, PHP_INI_PERDIR, PHP_INI_USER等常量的组合值. 当使用zend_alter_ini_entry()修改INI设置时, modify_type参数必须包含至少一个INI设置的modifiable属性值.

用户空间的ini_set()函数通过传递PHP_INI_USER利用了这个特性, 也就是说只有modifiable属性包含PHP_INI_USER标记的INI设置才能使用这个函数修改. 当在你的嵌入式应用中使用这个API调用时, 你可以通过传递PHP_INI_ALL标记短路这个访问控制系统, 它将包含所有的INI访问级别.

stage必须对应于Zend Engine的当前状态; 对于这些简单的嵌入式示例, 总是PHP_INI_STAGE_RUNTIME. 如果这是一个扩展或更高端的嵌入式应用, 你可能就需要将这个值设置为PHP_INI_STAGE_STARTUP或PHP_INI_STAGE_ACTIVE.

下面是扩展embed1.c源文件, 让它在执行脚本文件之前强制开启safe_mode.

```
    PHP_EMBED_START_BLOCK(argc, argv)
        zval    **SERVER_PP, *type;

        /* 不论php.ini中如何设置都强制开启safe_mode */
        zend_alter_ini_entry("safe_mode", sizeof("safe_mode"), "1", sizeof("1") - 1, PHP_INI_ALL,
PHP_INI_STAGE_RUNTIME);

        /* 注册$_SERVER超级全局变量 */
        zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
        /* 查找$_SERVER超级全局变量 */
        zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP) ;

        /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);

        php_execute_script(&script TSRMLS_CC);
```

```
    PHP_EMBED_END_BLOCK()
```

定义附加的超级全局变量

在第12章"启动, 终止, 以及其中的一些点"中, 你知道了用户空间全局变量以及超级全局变量可以在启动(MINIT)阶段定义. 同样, 本章介绍的嵌入式直接跳过了启动阶段, 处于运行时状态. 和覆写INI一样, 这并不会显得太迟.

超级全局变量的定义实际上只需要在脚本编译之前定义即可, 并且在php的进程生命周期中它只应该出现一次. 在扩展中的正常情况下, MINIT是唯一可以保证这些条件的地方.

由于你的包装应用现在是在控制中的, 因此可以保证定义用户空间自动全局变量的这些点位于真正编译脚本源文件的php_execute_script()命令之前. 我们定义一个$_EMBED超级全局变量并给它设置一个初始值来进行测试:

```
    PHP_EMBED_START_BLOCK(argc, argv)
        zval    **SERVER_PP, *type, *EMBED, *foo;

        /* 在全局作用域创建$_EMBED数组 */
        ALLOC_INIT_ZVAL(EMBED);
        array_init(EMBED);
        ZEND_SET_SYMBOL(&EG(symbol_table), "_EMBED", EMBED);

        /* $_EMBED['foo'] = 'Bar'; */
        ALLOC_INIT_ZVAL(foo);
        ZVAL_STRING(foo, "Bar", 1);
        add_assoc_zval_ex(EMBED, "foo", sizeof("foo"), foo);

        /* 注册超级全局变量$_EMBED */
        zend_register_auto_global("_EMBED", sizeof("_EMBED")
#ifdef ZEND_ENGINE_2
             , 1, NULL TSRMLS_CC);
#else
             , 1 TSRMLS_CC);
#endif

        /* 不论php.ini中如何设置都强制开启safe_mode */
        zend_alter_ini_entry("safe_mode", sizeof("safe_mode"), "1", sizeof("1") - 1, PHP_INI_ALL,
PHP_INI_STAGE_RUNTIME);

        /* 注册$_SERVER超级全局变量 */
        zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
        /* 查找$_SERVER超级全局变量 */
        zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP) ;

        /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);

        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()
```

要记住, Zend Engine 2(php 5.0或更高)使用了不同的zend_register_auto_global()元婴, 因此你需要用前面讲php 4兼容时候讲过的#ifdef. 如果你不关心旧版本php的兼容性, 则可以丢弃这些指令让代码变得更加整洁.

小结

如你所见, 将完整的Zend Engine和PHP语言嵌入到你的应用中相比如扩展新功能来说工作量要少. 由于它们共享相同的基础API, 我们可以学习尝试让其他实例可访问.

通过本章的学习, 你了解了最简单的嵌入式脚本代码格式, 同时还有all-in-one的宏PHP_EBED_START_BLOCK()和PHP_EMBED_END_BLOCK(). 下一章你将回到这些宏的层的使用, 利用它们将php和你的宿主系统结合起来.

高级嵌入式

php的嵌入式能够提供的可不仅仅是同步的加载和执行脚本. 通过理解php的执行模块各个部分是怎样组合的, 甚至给出一个脚本还可以回调到你的宿主应用中. 本章将涉及SAPI层提供的I/O钩子带来的好处, 展开你已经从前面的主题中获取到信息的执行模块进行学习.

回调到php中

除了加载外部的脚本, 和你在上一章看到的类似, 你的php嵌入式应用, 下面将实现一个类似于用户空间eval()的命令.

```
int zend_eval_string(char *str, zval *retval_ptr,
                     char *string_name TSRMLS_DC)
```

这里, str是实际要执行的php脚本代码, 而string_name是一个与执行关联的任意描述信息. 如果发生错误, php会将这个描述信息作为错误输出中的"文件名". retval_ptr, 你应该已经猜到了, 它将被设置为所传递代码产生的返回值. 试试用下面的代码创建新的项目吧.

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

int main(int argc, char *argv[]) {
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_eval_string("echo 'Hello World!';", NULL, "Simple Hello World App" TSRMLS_CC);
    PHP_EMBED_END_BLOCK()

    return 0;
}
```

现在使用命令或第19章"设置宿主环境"构建它(将Makefile中或命令中的embed1替换为embed2)

备选方案: 脚本文件的包含

可以预见的是, 这使得编译和执行外部脚本文件远比之前的方法更加容易, 因为你的应用可以将原本复杂的打开/准备/执行的执行序列, 以这种简化但功能更加强大的设计替代:

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

int main(int argc, char *argv[]) {
    char    *filename;

    if ( argc <= 1 ) {
        fprintf(stderr, "Usage: %s <filename.php> <arguments>\n", argv[1]);
        return -1;
    }

    filename    = argv[1];

    /* 忽略第0个参数 */
    argc --;
    argv ++;

    PHP_EMBED_START_BLOCK(argc, argv)
        char    *include_script;

        spprintf(&include_script, 0, "include '%s';", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    PHP_EMBED_END_BLOCK()

    return 0;
```

```
}
```

注意: 这种特殊的方法必须接受一个缺点, 如果文件名包含单引号, 将导致解析错误. 不过这可以通过使用*ext/standard/php_string.h*中的*php_addslashes()API*调用解决. 花一些时间去阅读这个文件以及附录中的*API*参考, 你会发现很多的特性, 它们可以让你避免在以后重造轮子.

调用用户空间函数

如你看到的加载和执行脚本文件, 在内部有两种方式调用用户空间函数. 现在最明显的可能是重用zend_eval_string(), 将函数名和所有它的参数组织到一个庞大的字符串中, 然后收集返回值.

```
    PHP_EMBED_START_BLOCK(argc, argv)
        char    *command;
        zval    retval;

        spprintf(&command, 0, "nl2br('%s');", argv[1]);
        zend_eval_string(command, &retval, "nl2br() execution" TSRMLS_CC);
        efree(command);
        printf("out: %s\n", Z_STRVAL(retval));
        zval_dtor(&retval);
    PHP_EMBED_END_BLOCK()
```

和前面的include很像, 这个方法有一个致命的缺陷: 如果输入参数paramin(译者给出的例子中是argv[1])给出一个错误的数据, 函数将会失败, 或者更糟糕的是导致无法预期的结果. 解决方案是永远都避免编译代码的运行时片段, 并直接使用call_user_function()API调用函数.

```
int call_user_function(HashTable *function_table, zval **object_pp,
                       zval *function_name, zval *retval_ptr,
                       zend_uint param_count, zval *params[] TSRMLS_DC);
```

实际上从引擎外部调用时, function_table总是EG(function_table). 如果调用一个对象或类方法, object_pp需要是IS_OBJECT类型的调用实例zval, 或者对于类的静态调用则是IS_STRING的值. function_name通常是IS_STRING的值, 包含要调用的函数名, 但是它也可以是IS_ARRAY, 第0个元素包含一个对象或类名, 第1个元素包含方法名.

这个函数调用的结果是向传入的retval_ptr指向的zval设置返回值. param_count和params扮演了argc/argv的角色. 也就是说, params[0]包含所传递的第一个参数, params[param_count - 1]包含了所传递的最后一个参数.

下面是用这种方法重新实现上面的例子:

```
    PHP_EMBED_START_BLOCK(argc, argv)
        zval    *args[1];
        zval    retval, str, funcname;

        ZVAL_STRING(&funcname, "nl2br", 0);
        args[0] = &str;
        ZVAL_STRINGL(args[0], "HELLO WORLD!", sizeof("HELLO WORLD!"), 1);
        call_user_function(EG(function_table), NULL, &funcname, &retval, 1, args TSRMLS_CC);

        printf("out: %s\n", Z_STRVAL(retval));

        zval_dtor(args[0]);
        zval_dtor(&retval);
    PHP_EMBED_END_BLOCK()
```

尽管代码看起来比较长, 但是工作量会显著降低, 因为这里没有要编译的中间代码, 传递的数据不需要复制, 每个参数都已经在Zend兼容的结构体中. 同时, 要记得原来的例子中在字符串中包含单引号时会有潜在的错误. 而这个版本没有这个问题.

错误处理

当发生错误时, 比如脚本解析错误, php将会进入到bailout模式. 在你已经看到的简单的嵌入式例子中, 这表示它将直接跳到PHP_EMBED_END_BLOCK()宏, 并且绕过所有这个块中的剩余代码. 由于多数潜入php解释器的应用, 目的并不只是为了执行php代码, 因此避免由于php脚本的故障导致整个应用崩溃是有意义的.

有一种方式可以将所有的执行限制到一个非常小的START/END块中, 这样发生崩溃就只影响当前块. 这种方式的缺点是每个START/END块函数都是独立的PHP请求. 因此比如下面START/END块, 虽然从语法逻辑上来看两个块是协同工作的, 但实际上它们之间是不共享公共作用域的.

```
int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_eval_string("$a = 1;", NULL, "Script Block 1");
    PHP_EMBED_END_BLOCK()
    PHP_EMBED_START_BLOCK(argc, argv)
        /* 将打印出"NULL", 因为变量$a在这个请求中并没有定义. */
        zend_eval_string("var_dump($a);", NULL, "Script Block 2");
    PHP_EMBED_END_BLOCK()
    return 0;
}
```

还有一种解决方法是将两个zend_eval_string()调用使用Zend特有的伪语言结构zend_try, zend_catch, zend_end_try进行隔离. 使用这些结构, 你的应用就可以按照想要的方式处理错误. 考虑下面的代码:

```
int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_try {
            /* 尝试执行一些可能失败的代码 */
            zend_eval_string("$1a = 1;", NULL, "Script Block 1a");
        } zend_catch {
            /* 发生错误, 则尝试执行另外一部分代码(一般错误的补救或报告等行为) */
            zend_eval_string("$a = 1;", NULL, "Script Block 1");
        } zend_end_try();
        /* 这里将显示"NULL", 因为变量$a在这个请求中没有定义. */
        zend_eval_string("var_dump($a);", NULL, "Script Block 2");
    PHP_EMBED_END_BLOCK()
    return 0;
}
```

在这个示例的第二个版本中, zend_try块中将发生解析错误, 但它只影响自己的代码块, 同时在zend_catch块中使用了一段好的代码对错误进行了处理. 同样你也可以尝试自己给var_dump()部分也加上这些块.

译注*:* 这里对*zend_try/zend_catch/zend_end_try*解释的不是很清楚, 因此做以下补充说明. 读者阅读这一部分内容需要首先了解*sigsetjmp()/siglongjmp()*的机制*(*可以参考*<Unix*环境高级编程*>*第*10*章第*15*节*).*

相关的定义如下*:*

```
#ifdef HAVE_SIGSETJMP#   define SETJMP(a) sigsetjmp(a, 0)
#   define LONGJMP(a,b) siglongjmp(a, b)
#   define JMP_BUF sigjmp_buf
#else
#   define SETJMP(a) setjmp(a)
#   define LONGJMP(a,b) longjmp(a, b)
#   define JMP_BUF jmp_buf
#endif

#define zend_try                                                    \
    {                                                               \
```

```
        JMP_BUF *__orig_bailout = EG(bailout);                  \
        JMP_BUF __bailout;                                      \
                                                                \
        EG(bailout) = &__bailout;                               \
        if (SETJMP(__bailout)==0) {
#define zend_catch                                              \
        } else {                                                \
            EG(bailout) = __orig_bailout;
#define zend_end_try()                                          \
        }                                                       \
        EG(bailout) = __orig_bailout;                           \
    }
```

*zend_try {}*代码块中的代码是在一个*if*语句中的, 这个*if*的条件是*SETJMP(__bailout) == 0, SETJMP()*是在当前程序执行的点设置一个可回溯的点*(*保存了当前执行上下文和环境*), SETJMP()*的返回比较特殊, 它有两种返回*: 1)* 直接返回, 此时返回值为*0; 2)* 调用*LONGJMP()*返回到对应*__bailout*当时调用*SETJMP()*的位置, 此时返回值非*0.*

基于上面的论述, 可以看出, 当*zend_try*的代码块中调用了*LONGJMP()*的时候, 程序将回到*if (SETJMP(__bailout) == 0)*的位置开始执行, 并且它的返回值为*-1*, 因此, 进入到对应的*else*语句块, 也就是*zend_catch*语句块的代码.

*zend_end_try()*则只是一个结尾的花括号.

*php*中的这个伪语言结构正式这种方式实现的异常处理机制, 在系统的关键点调用*zend_bailout()(*在*Zend/zend.h*中定义*)*即可.

本例中, 译者增加了*zend_bailout()*调用, 演示了这个伪语言结构的使用.

初始化php

迄今为止, 你看到的PHP_EMBED_START_BLOCK()和PHP_EMBED_END_BLOCK()宏都用于启动, 执行, 终止一个紧凑的院子的php请求. 这样做的优点是任何导致php bailout的错误顶多影响到PHP_EMBED_END_BLOCK()宏之内的当前作用域. 通过将你的代码执行放入到这两个宏之间的小块中, php的错误就不会影响到你的整个应用.

你刚才已经看到了, 这种短小精悍的方法主要的缺点在于每次你建立一个新的START/END块的时候, 都需要创建一个新的请求, 新的符号表, 因此就失去了所有的持久性语义.

要想同时得到两种优点(持久化和错误处理), 就需要将START和END宏分解为它们各自的组件(译注: 如果不明白可以参考这两个宏的定义). 下面是本章开始给出的embed2.c程序, 这一次, 我们对它进行了分解:

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

int main(int argc, char *argv[])
{
#ifdef ZTS
    void ***tsrm_ls;
#endif

    php_embed_init(argc, argv PTSRMLS_CC);
    zend_first_try {
        zend_eval_string("echo 'Hello World!';", NULL,
                         "Embed 2 Eval'd string" TSRMLS_CC);
    } zend_end_try();
    php_embed_shutdown(TSRMLS_C);

    return 0;
}
```

它执行和之前一样的代码, 只是这一次你可以看到打开和关闭的括号包裹了你的代码, 而不是无法分开的START和END块. 将php_embed_init()放到你应用的开始, 将

php_embed_shutdown()放到末尾, 你的应用就得到了一个持久的单请求生命周期, 它还可以使用zend_first_try {} zend_end_try(); 结构捕获所有可能导致你整个包装应用跳出末尾的PHP_EMBED_END_BLOCK()宏的致命错误.

这个时候要注意, *zend_first_try,* 而不是*zend_try.* 在*TRy/catch*块外围使用*zend_first_try*非常重要, 因为*zend_first_try*执行了一些额外的步骤*(*清理*EG(bailout))*

为了看看真实世界环境的这种方法的应用, 我们将本章前面一些的例子的启动和终止处理进行了抽象:

```
#include <sapi/embed/php_embed.h>
#ifdef ZTS
    void ***tsrm_ls;
#endif
static void startup_php(void)
{
    /* Create "dummy" argc/argv to hide the arguments
     * meant for our actual application */
    int argc = 1;
    char *argv[2] = { "embed4", NULL };
    php_embed_init(argc, argv PTSRMLS_CC);
}
static void shutdown_php(void)
{
    php_embed_shutdown(TSRMLS_C);
}
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]);
    shutdown_php();
    return 0;
}
```

类似的概念也可以应用到处理任意代码的执行以及其他任务. 只需要确认在最外部的容器上使用zend_first_try, 则里面的每个容器上使用zend_try即可.

覆写INI_SYSTEM和INI_PERDIR选项

在上一章中, 你曾经使用zend_alter_ini_setting()修改过一些php的ini选项. 由于samp/embed直接将你的脚本推入了运行时模式, 因此许多重要的INI选项在控制返回到你的应用时并没有被修改. 为了修改这些值, 就需要在主引擎启动之后而请求启动之前执行代码.

有一种方式是拷贝php_embed_init()的内容到你的应用中, 在你的本地拷贝中做必要的修改, 接着使用你修改后的版本替代它. 当然这种方式可能会有问题.

首先也是最重要的, 你实际已经对别人的部分代码做了分支, 然而可能别人还会向其中添加新的代码. 现在, 你就不再是只维护自己的应用了, 还需要保持分支出来的代码和主分支保持一致. 幸运的是, 还有几种更简单的方法:

覆写默认的php.ini文件

因为嵌入式和其他的php sapi实现一样都是sapi, 它通过一个sapi_module_struct挂入到引擎中. 嵌入式SAPI定义并设置了这个结构体的一个实例, 你的应用可以在调用php_embed_init()之前访问它.

在这个结构体中, 有一个名为php_ini_path_override的char *类型字段. 为了让嵌入的请求使用你的可选文件扩展php和Zend, 只需要在调用php_embed_init()之前将这个字段设置为NULL终止的字符串. 下面是embed4.c中修改版的startup_php()函数:

```
static void startup_php(void)
{
    /* Create "dummy" argc/argv to hide the arguments
     * meant for our actual application */
    int argc = 1;
    char *argv[2] = { "embed4", NULL };

    php_embed_module.php_ini_path_override = "/etc/php_embed4.ini";
    php_embed_init(argc, argv PTSRMLS_CC);
}
```

这就使得每个使用嵌入库的应用可以保持自定义, 而不用将自己的配置暴露给别人. 相反, 如果你想要你的应用不使用php.ini, 只需要设置php_embed_module的php_ini_ignore字段, 这样所有的设置都将使用内建的默认值, 除非由你的应用手动进行修改.

覆写嵌入启动

sapi_module_struct结构还包含一些回调函数, 下面是其中4个在PHP启动和终止阶段比较有用的回调:

```
/* From main/SAPI.h */
typedef struct _sapi_module_struct {

    ...
    int (*startup)(struct _sapi_module_struct *sapi_module);
    int (*shutdown)(struct _sapi_module_struct *sapi_module);
    int (*activate)(TSRMLS_D);
    int (*deactivate)(TSRMLS_D);
    ...
} sapi_module_struct;
```

这些方法的名字熟悉吗? 它们对应于扩展的MINIT, MSHUTDOWN, RINIT, RSHUTDOWN, 并且和对应在扩展生命周期中的阶段一致. 要利用这些钩子, 可以如下修改embed4中的startup_php()函数:

```
static int (*original_embed_startup)(struct _sapi_module_struct *sapi_module);

static int embed4_startup_callback(struct _sapi_module_struct *sapi_module)
{
    /* 首先调用原来的启动回调, 否则环境未就绪 */
    if (original_embed_startup(sapi_module) == FAILURE) {
        /* 这里可以做应用的失败处理 */
        return FAILURE;
    }
    /* 调用原来的embed_startup实际上让我们进入到ACTIVATE阶段而不是STARTUP阶段,
     * 但是我们仍然可以修改多数INI_SYSTEM和INI_PERDIR选项.
     */
    zend_alter_ini_entry("max_execution_time", sizeof("max_execution_time"),
                "15", sizeof("15") - 1, PHP_INI_SYSTEM, PHP_INI_STAGE_ACTIVATE);
    zend_alter_ini_entry("safe_mode", sizeof("safe_mode"),
                "1", sizeof("1") - 1, PHP_INI_SYSTEM, PHP_INI_STAGE_ACTIVATE);
    return SUCCESS;
}
```

```
static void startup_php(void)
{
    /* 创建假的argc/argv, 隐藏应用实际的参数 */
    int argc = 1;
    char *argv[2] = { "embed4", NULL };

    /* 使用我们自己的启动函数覆写标准的启动方法, 但是保留了原来的指针, 因此它仍然能够被调用到 */
    original_embed_startup = php_embed_module.startup;
    php_embed_module.startup = embed4_startup_callback;

    php_embed_init(argc, argv PTSRMLS_CC);
}
```

使用safe_mode, open_basedir这样的选项, 以及其他用以限制独立脚本行为的选项, 可以让你的应用更加安全可靠.

捕获输出

除非你开发的是非常简单的控制台应用, 否则你应该不希望php脚本代码产生的输出直接被扔到激活的终端上. 捕获这些输出和你刚才用以覆写启动处理器的方法类似.

在sapi_module_struct中还有一些有用的回调:

```
typedef struct _sapi_module_struct {
    ...
    int (*ub_write)(const char *str, unsigned int str_length TSRMLS_DC);
    void (*flush)(void *server_context);
    void (*sapi_error)(int type, const char *error_msg, ...);
    void (*log_message)(char *message);
    ...
} sapi_module_struct;
```

标准输出: ub_write

所有用户空间的echo和print语句产生的输出, 以及其他内部通过php_printf()或PHPWRITE()产生的输出, 最终都将被发送到激活的SAPI的ub_write()方法. 默认情况, 嵌入式SAPI直接将这些数据交给stdout管道, 而不关心你的应用的输出策略.

假设你的应用想要把所有的输出都发送到一个独立的控制台窗口; 你可能需要实现一个类似于下面伪代码块所描述的回调:

```
static int embed4_ub_write(const char *str, unsigned int str_length TSRMLS_DC)
{
    output_string_to_window(CONSOLE_WINDOW_ID, str, str_length);
    return str_length;
}
```

要让这个函数能够处理php产生的内容, 你需要在调用php_embed_init()之前对php_embed_module结构做适当的修改:

```
php_embed_module.ub_write = embed4_ub_write;
```

注意: 哪怕你决定你的应用不需要*php*产生的输出, 也必须为*ub_write*设置一个回调. 将它的值设置为*NULL*将导致引擎崩溃, 当然, 你的应用也不能幸免.

缓冲输出: Flush

你的应用可能会使用缓冲php产生的输出进行优化, sapi层提供了一个回调用以通知你的应用"现在请发送你的缓冲区数据", 你的应用并没有义务去实施这个通知; 不过, 由于这个信息通常是由于足够的理由(比如到达请求结束位置)才产生的, 听从这个意见并不会有什么坏处.

下面的这对回调函数, 以256字节缓冲区缓冲数据由引擎安排执行flush.

```
char buffer[256];
int buffer_pos = 0;
static int embed4_ubwrite(const char *str, unsigned int str_length TSRMLS_DC)
```

```
{
    char *s = str;
    char *d = buffer + buffer_pos;
    int consumed = 0;
    /* 缓冲区够用, 直接追加到缓冲区后面 */
    if (str_length < (256 - buffer_pos)) {
        memcpy(d, s, str_length);
        buffer_pos += str_length;
        return str_length;
    }
    consumed = 256 - buffer_pos;
    memcpy(d, s, consumed);
    embed4_output_chunk(buffer, 256);
    str_length -= consumed;
    s += consumed;
    /* 消耗整个传入的块 */
    while (str_length >= 256) {
        embed4_output_chunk(s, 256);
        s += 256;
        consumed += 256;
    }
    /* 重置缓冲区头指针内容 */
    memcpy(buffer, s, str_length);
    buffer_pos = str_length;
    consumed += str_length;
    return consumed;
}
static void embed4_flush(void *server_context)
{
    if (buffer_pos < 0) {
        /* 输出缓冲区中剩下的内容 */
        embed4_output_chunk(buffer, buffer_pos);
        buffer_pos = 0;
    }
}
```

在startup_php()中增加下面的代码, 这个基础的缓冲机制就绪了:

```
php_embed_module.ub_write = embed4_ub_write;
php_embed_module.flush = embed4_flush;
```

标准错误: log_message

在启用了log_errors INI设置时, 在启动或执行脚本时如果碰到错误, 将激活log_message回调. 默认的php错误处理程序会在处理显示(这里是调用log_message回调)之前, 格式化这些错误消息, 使其称为整齐的, 人类可读的内容.

关于log_message回调, 这里你需要注意的第一件事是它并不包含长度参数, 因此它并不是二进制安全的. 也就是说, 它只是按照NULL终止来处理字符串末尾.

使用它来做错误报告通常不会有什么问题, 实际上, 它可以用于在错误消息的呈现上做更多的事情. 默认情况下, sapi/embed将会通过这个简单的内建回调, 发送这些错误消息到标准错误管道:

```
static void php_embed_log_message(char *message)
{
    fprintf (stderr, "%s\n", message);
}
```

如果你想发送这些消息到日志文件, 则可以使用下面的版本替代:

```
static void embed4_log_message(char *message)
{
    FILE *log;
    log = fopen("/var/log/embed4.log", "a");
    fprintf (log, "%s\n", message);
    fclose(log);
}
```

特殊错误: sapi_error

少数特殊情况的错误属于某个sapi, 因此将绕过php的主错误处理程序. 这些错误一般是由于使用不当造成的, 比如非web应用不应该使用header()函数, 上传文件到控制台应用程序等.

由于这些情况都离你所开发的sapi/embed应用非常遥远, 因此最好保持这个回调为空. 不过, 如果你非要坚持去捕获每种类型错误的源, 也只需要实现一个回调函数, 并在调用php_embed_init()之前覆写它就可以了.

同时扩展和嵌入

在你的应用中运行php代码固然不错, 但是此刻, php执行环境仍然和你的主应用是隔离的, 它们并没有在真正意义上的一个层级进行交互.

现在你应该对php扩展的开发以及构建启用方面比较熟悉了. 你也已经有完成了嵌入工作的例程, 这样就省去了这份工作. 将扩展代码植入到嵌入式应用中的工作量要比标准扩展小. 下面是一个新的嵌入式项目:

```
#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;
}
```

现在, 你可以定义function_entry向量, 启动/终止函数, 定义类, 以及所有你想增加的东西. 现在, 它和你使用用户空间的dl()命令加载这个扩展库一样, 在这一个命令中Zend将自动的处理所有的钩子并对你的模块进行注册, 就绪等待使用.(译注: startup_php()中调用zend_startup_module(&php_mymod_module_entry)进行了模块注册)

小结

本章你看了一些上一章的一些简单的嵌入式示例进行了扩展, 你已经可以将php放入到各种非线程应用了. 现在你已经掌握了扩展和嵌入式的基础, 并且可以在zval, 类, 资源, HashTable上工作了, 你已经可以真正开始一个真正的项目了.

在剩下的附录中, 你将看到php, zend以及其他扩展暴露的很多API函数. 你将会看到一些常用的代码片段以及近几年数以百计的开源PECL项目, 它们都可以作为你未来项目的参考.

后记

本书的翻译从2012年12月25日开始, 历时两个月, 于2013年2月19日完成译稿. 我自己在翻译的过程中也收获颇多.

剩下的附录部分由于API变化特性, 不再翻译, 读者可在学习工作中多参考源代码.

感谢上天让我可以从事软件开发的工作.

感谢 Rasmus Lerdorf, Zeev Suraski, Andi Gutmans, Sara Golemon以及所有无私奉献的大牛们.