05. 深入浅出ODP框架 —— AP源码解析

作者：倪煜

AP框架是通过PHP扩展实现的一种PHP
MVC开发框架，也是ODP架构的核心技术。其作者是著名的鸟哥，对应的开源版本为Yaf。学习并了解其实现原理，即有助于很好的理解ODP开发模式，还能看到框架的实现细节，避免在使用过程中踩坑，也为进一步改进和优化AP框架奠定基础。同时还可以学习到优秀的PHP扩展开发的一些技巧。本文通过对AP的源码解析，深入理解其实现细节，并通过简单举例贯串框架主线，帮助读者深入浅出。

1. 前言

因为AP框架是PHP扩展实现的，所以学习AP源码必须要了解一些PHP内核原理，PHP扩展开发基础。由于PHP内核及扩展均为C语言实现，还需要具备一定的C基础。AP框架在公司内主要随ODP开发框架一起，文中某些例子是基于ODP的，所以希望读者是已经使用过ODP开发的，或者是了解过ODP开发的。本文假设读者已具备所需前驱知识，文章附录会附上相关知识的学习地址，方便不了解的同学进一步学习。

2. AP整体流程

这里引用AP官方的经典流程图如图1所示：

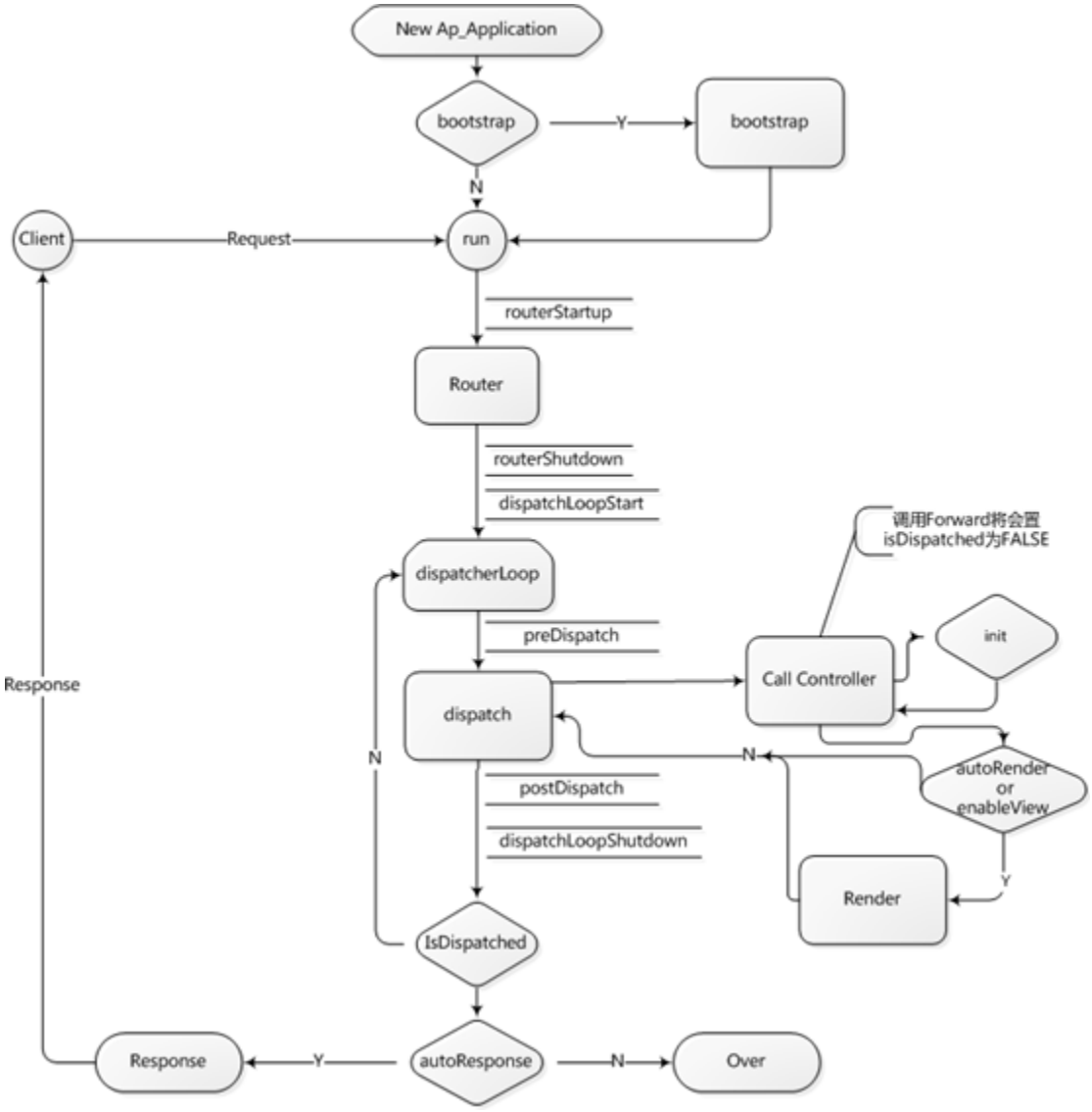


图1 AP流程图

对照官方手册时初看这个流程图时，可能对其中各个环节并没有很清晰的印象，但是如果结合源码一起的话，相信大家看过之后就会印象深刻。下面我就通过源码来分析上图中的各个环节。

我们在开发一个ODP的app时，通常的入口index文件内容会是如下：

```
$objApplication = Bd_Init::init();
$objResponse = $objApplication->bootstrap()->run();
```

非常简单的两句代码，包含了全部内容。

第一行完成了AP框架核心类Ap_Application的初始化

Bd_Init::init()
Bd_Init类跟AP框架没有关系，这里简单介绍一下，在其init方法中会new一个Ap_Application并将其返回

```
$app = new Ap_Application(array('ap' => $ap_conf));
```

另一个问题，为何能在入口文件处直接使用Bd_Init类呢，系统如何找到的。

是在 ./etc/ext/init.ini配置中有

```
auto_prepend_file =/home/users/niyu/odp/php/phplib/bd/Init.php
```

这样就会在index.php执行之前先加载Bd_init类了。

第二行实际就是完成了上面流程图中的所有环节，我们先简单分析一下上述流程。

在ODP开发框架中，通常一个典型app目录结构如下：

```
|-- actions
|   |-- api
|   |   `-- Sample.php
|   `-- Sample.php
|-- Bootstrap.php
|-- controllers
|   |-- Api.php
|   `-- Main.php
|-- library
|   `-- one
|       `-- Util.php
|-- models
|   |-- dao
|   |   `-- Sample.php
|   `-- service
|       |-- data
|       |   `-- Sample.php
|       `-- page
|           |-- SampleApi.php
|           `-- Sample.php
|-- script
|   `-- sampleScript.php
`-- test
```

我们看到app一级目录下有个Bootstrap.php文件，这个就是对应图1.1中的第一个环节，如果存在Bootstrap就会先执行该文件。该文件包含了一系列的初始化环节，并返回一个Ap_Application对象，紧接着调用了它的run方法，run里面包含了图中所有环节。run首先是调用路由，路由的主要目的其实就是找到controllers文件，该文件中记录着所有actions的地址，所以通过controllers加载action，而action就是真正业务逻辑的入口。在下面的dispatcher中会调用action的execute方法来调用下层业务逻辑，如果设置了autoRender在返回的时候会执行render方法。图中有六个双横线标出的环节，就是六个插件方法，用户可以自定义实现这几个方法，AP框架会在图中相应的步骤处调用对应的HOOK方法。

所以一次请求的过程就是通过路由找到action，然后执行其execute方法。下面分章节通过源码详细介绍AP中各个部分的实现过程。AP代码结构也很简单清晰，基本按照每个C文件去分析就可以了。

2. 核心功能模块

2.1 Ap_Application

application就是AP框架的核心类，我们先看其定义

```
/** {{{ AP_STARTUP_FUNCTION
*/
AP_STARTUP_FUNCTION(application) {
    zend_class_entry ce;
    AP_INIT_CLASS_ENTRY(ce, "Ap_Application", "Ap\\Application", ap_application_methods);

    ap_application_ce            = zend_register_internal_class_ex(&ce, NULL, NULL TSRMLS_CC);
    ap_application_ce->ce_flags |= ZEND_ACC_FINAL_CLASS;

    zend_declare_property_null(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_CONFIG),   ZEND_ACC_PROTECTED  TSRMl
    zend_declare_property_null(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_DISPATCHER), ZEND_ACC_PROTECTED
    zend_declare_property_null(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_APP),      ZEND_ACC_STATIC|ZEND_ACC_
    zend_declare_property_null(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_MODULES),  ZEND_ACC_PROTECTED TSRMLS

    zend_declare_property_bool(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_RUN),   0, ZEND_ACC_PROTECTED  TSRMl
    zend_declare_property_string(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_ENV), AP_G(environ), ZEND_ACC_PRO

    zend_declare_property_long(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_ERRNO), 0, ZEND_ACC_PROTECTED TSRMLS
    zend_declare_property_string(ap_application_ce, ZEND_STRL(AP_APPLICATION_PROPERTY_NAME_ERRMSG), "", ZEND_ACC_PROTECTED TS

    return SUCCESS;
} ? end AP_STARTUP_FUNCTION ?
/* }}} */
```

PHP内核方法
zend_class_entry ce;

PHP内核中对PHP类的实现是通过zend_class_entry结构实现的，所以可以把ce理解为类的通用结构

AP_INIT_CLASS_ENTRY(ce, "Ap_Application", "Ap\\Application", ap_application_methods);

相当于对ce初始化，指定一个类名称Ap_Application，指定类的成员方法列表 ap_application_methods

ap_application_ce   = zend_register_internal_class_ex(&ce, NULL, NULL TSRMLS_CC);

向PHP注册类，PHP中由class_table维护全局的类数组，可以简单理解为把类添加到这个数组中，这样就可以在PHP中找到这个类了。内核中有一组类似的注册函数，用来注册接口、类、子类、接口实现、抽象类等

ap_application_ce->ce_flags |= ZEND_ACC_FINAL_CLASS;

指定类属性，内核中有一组这样的属性标记来指定类的性质

```
/* method flags (types) */
#define ZEND_ACC_STATIC           0x01
#define ZEND_ACC_ABSTRACT         0x02
#define ZEND_ACC_FINAL            0x04
#define ZEND_ACC_IMPLEMENTED_ABSTRACT       0x08
/* class flags (types) */

//没有声明为抽象,但是内部有抽象方法
/* ZEND_ACC_IMPLICIT_ABSTRACT_CLASS is used for abstract classes (since it is set by any abstract method even
interfaces MAY have it set, too). */
//抽象，用abstract关键字定义为抽象类的
/* ZEND_ACC_EXPLICIT_ABSTRACT_CLASS denotes that a class was explicitly defined as abstract by using the
keyword. */

#define ZEND_ACC_IMPLICIT_ABSTRACT_CLASS    0x10
#define ZEND_ACC_EXPLICIT_ABSTRACT_CLASS    0x20
#define ZEND_ACC_FINAL_CLASS                0x40
#define ZEND_ACC_INTERFACE                  0x80
```

zend_declare_property_null

一系列定义类成员变量的函数

我们再看一下类成员函数的声明：

```
/** {{{ ap_application_methods
*/
zend_function_entry ap_application_methods[] = {
    PHP_ME(ap_application, __construct,     ap_application_construct_arginfo, ZEND_ACC_PUBLIC|ZEND_ACC_CTOR)
    PHP_ME(ap_application, run,             ap_application_run_arginfo,         ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, execute,     ap_application_execute_arginfo,     ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, app,             ap_application_app_arginfo,         ZEND_ACC_PUBLIC|ZEND_ACC_STATIC)
    AP_ME(ap_application_environ, "environ", ap_application_environ_arginfo,    ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, bootstrap,       ap_application_bootstrap_arginfo,   ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getConfig,       ap_application_getconfig_arginfo,   ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getModules,      ap_application_getmodule_arginfo,   ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getDispatcher,   ap_application_getdispatch_arginfo,ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, setAppDirectory,ap_application_setappdir_arginfo,  ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getAppDirectory,ap_application_void_arginfo, ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getLastErrorNo, ap_application_void_arginfo, ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, getLastErrorMsg,ap_application_void_arginfo, ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, clearLastError, ap_application_void_arginfo, ZEND_ACC_PUBLIC)
    PHP_ME(ap_application, __destruct,      NULL, ZEND_ACC_PUBLIC|ZEND_ACC_DTOR)
    PHP_ME(ap_application, __clone,     NULL, ZEND_ACC_PRIVATE|ZEND_ACC_CLONE)
    PHP_ME(ap_application, __sleep,     NULL, ZEND_ACC_PRIVATE)
    PHP_ME(ap_application, __wakeup,        NULL, ZEND_ACC_PRIVATE)
    {NULL, NULL, NULL}
};
/* }}} */
```

注意红框中的两个方法，对应的就是入口文件中执行的两个方法

$objResponse = $objApplication->bootstrap()->run();

因此我们这里只需要看这两个函数就OK了。

- 1 . PHP_METHOD(ap_application, bootstrap)

```
len = spprintf(&bootstrap_path, 0, "%s%c%s.%s", AP_G(directory), DEFAULT_SLASH, AP_DEFAULT_BOOTSTRAP, AP_G(ext));
//#define AP_DEFAULT_BOOTSTRAP          "Bootstrap"
```

bootstrap_path就是定位到Bootstap文件，可以看出必须是app_directory/Bootstrap.php这个文件，大小写都不能错。
Bootstrap继承至Ap_Bootstrap_Abstract，一段典型的Bootstrap文件通常如下：

```
class Bootstrap extends Ap_Bootstrap_Abstract{
        public function _initRoute(Ap_Dispatcher $dispatcher) {
                //在这里注册自己的路由协议,默认使用static路由
        }

        public function _initPlugin(Ap_Dispatcher $dispatcher) {
        //注册saf插件
        $objPlugin = new Saf_ApUserPlugin();
        $dispatcher->registerPlugin($objPlugin);
    }
```

Ap_Bootstrap_Abstract是一个没有方法的抽象类。

```
/** {{{ AP_STARTUP_FUNCTION
*/
AP_STARTUP_FUNCTION(bootstrap) {
    zend_class_entry ce;
    AP_INIT_CLASS_ENTRY(ce, "Ap_Bootstrap_Abstract",  "Ap\\Bootstrap_Abstract", ap_bootstrap_methods);
    ap_bootstrap_ce = zend_register_internal_class_ex(&ce, NULL, NULL TSRMLS_CC);
    ap_bootstrap_ce->ce_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
    return SUCCESS;
}
/* }}} */
```

Bootstrap类实现了一系列的_init开头的方法，在PHP_METHOD(ap_application, bootstrap)中

```
methods      = &((*ce)->function_table);//Bootstrap类中实现的所有方法，也就是那些_init开头的方法

for(zend_hash_internal_pointer_reset(methods);//遍历bootstrap中定义的方法
//内核中对array的实现是通过hash_table实现的，内核中会看到到处使用hash_table的地方，可以简单理解为数组的操作
zend_hash_has_more_elements(methods) == SUCCESS;
zend_hash_move_forward(methods)) {
    char *func;
    uint len;
    ulong idx;

    //取出方法名，赋值给func
    zend_hash_get_current_key_ex(methods, &func, &len, &idx, 0, NULL);
    /* cann't use ZEND_STRL in strncasecmp, it cause a compile failed in VS2009 */
    if (strncasecmp(func, AP_BOOTSTRAP_INITFUNC_PREFIX, sizeof(AP_BOOTSTRAP_INITFUNC_PREFIX)-1)) {
        continue;
    }//比较函数func是否以_init开头
    //调用所有以_init开头的函数，入参统一为dispatcher
    zend_call_method(&bootstrap, *ce, NULL, func, len - 1, NULL, 1, dispatcher, NULL TSRMLS_CC);
    /** an uncaught exception threw in function call */
    if (EG(exception)) {
        zval_ptr_dtor(&bootstrap);
        RETURN_FALSE;
    }
}
```

上面代码片段加了注释，很容易看出Bootstrap的功能就是按你顶一个的_init开头的方法顺序，依次调用，且入参都为dispatcher。dispatcher类后文会详细介绍。

bootstrap方法最后会RETVAL_ZVAL(self, 1, 0);，也就是返回application自身，下面看最重要的run方法

- 2 . PHP_METHOD(ap_application, run)

run主要就是调用了

```
if ((response = ap_dispatcher_dispatch(dispatcher TSRMLS_CC))) {
    RETURN_ZVAL(response, 1, 1);
}
```

主要分两大部分：

1. 路由

```
/* route request */
if (!ap_request_is_routed(request TSRMLS_CC)) {
    AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_ROUTESTARTUP, request, response);
    AP_EXCEPTION_HANDLE(dispatcher, request, response);
    if (!ap_dispatcher_route(dispatcher, request TSRMLS_CC)) {
        ap_trigger_error(AP_ERR_ROUTE_FAILED TSRMLS_CC, "Routing request failed");
        AP_EXCEPTION_HANDLE_NORET(dispatcher, request, response);
        zval_ptr_dtor(&response);
        return NULL;
    }
    ap_dispatcher_fix_default(dispatcher, request TSRMLS_CC);
    AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_ROUTESHUTDOWN, request, response);
    AP_EXCEPTION_HANDLE(dispatcher, request, response);
    (void)ap_request_set_routed(request, 1 TSRMLS_CC);
} else {
    ap_dispatcher_fix_default(dispatcher, request TSRMLS_CC);
}

AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_LOOPSTARTUP, request, response);
AP_EXCEPTION_HANDLE(dispatcher, request, response);
```

2. 分发

```
do {
    AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_PREDISPATCH, request, response);
    if (!ap_dispatcher_handle(dispatcher, request, response, view TSRMLS_CC)) {
        AP_EXCEPTION_HANDLE(dispatcher, request, response);
        zval_ptr_dtor(&response);
        return NULL;
    }
    ap_dispatcher_fix_default(dispatcher, request TSRMLS_CC);
    AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_POSTDISPATCH, request, response);
    AP_EXCEPTION_HANDLE(dispatcher, request, response);
} while (--nesting > 0 && !ap_request_is_dispatched(request TSRMLS_CC));

AP_PLUGIN_HANDLE(plugins, AP_PLUGIN_HOOK_LOOPSHUTDOWN, request, response);
AP_EXCEPTION_HANDLE(dispatcher, request, response);
```

第一个高亮的方法ap_dispatcher_route，对应的就是图1.1中的route环节，简单说就是通过解析URL获得对应的action类入口，这部分内容放在后面专门介绍。

我们再注意一下六个标黄的宏，其对应的就是图1.1中六个插件HOOK方法

```
#define AP_PLUGIN_HOOK_ROUTESTARTUP          "routerstartup"
#define AP_PLUGIN_HOOK_ROUTESHUTDOWN         "routershutdown"
#define AP_PLUGIN_HOOK_LOOPSTARTUP           "dispatchloopstartup"
#define AP_PLUGIN_HOOK_PREDISPATCH           "predispatch"
#define AP_PLUGIN_HOOK_POSTDISPATCH          "postdispatch"
#define AP_PLUGIN_HOOK_LOOPSHUTDOWN          "dispatchloopshutdown"
#define AP_PLUGIN_HOOK_PRERESPONSE           "preresponse"

//宏定义如下：
#define AP_PLUGIN_HANDLE(p, n, request, response) \
    do { \
        if (!ZVAL_IS_NULL(p)) { \
            zval **_t_plugin;\
            for(zend_hash_internal_pointer_reset(Z_ARRVAL_P(p));\
            zend_hash_has_more_elements(Z_ARRVAL_P(p)) == SUCCESS;\
            zend_hash_move_forward(Z_ARRVAL_P(p))) {\
                if (zend_hash_get_current_data(Z_ARRVAL_P(p), (void**)&_t_plugin) == SUCCESS) {\
                    if (zend_hash_exists(&(Z_OBJCE_PP(_t_plugin)->function_table), n, sizeof(n))) {\
                        zend_call_method_with_2_params(_t_plugin, Z_OBJCE_PP(_t_plugin), NULL, n, NULL, request, response);\
                    }\
                }\
            }\
        }\
    } while(0)
```

上面的代码可以看出插件就是在上述六个地方，调用注册的插件。宏定义的方法就是遍历plugins集合，也就是宏里的p，调用名称为n的方法，也就是那六个方法名之一。所以你可以注册多个插件，插件的实现也很简单，只要继承Ap_Plugin_Abstract就行，就可以在不同的环节中执行自定义行为，按注册时的顺序调用各个插件的hook方法。

```
/** {{{ AP_STARTUP_FUNCTION
*/
AP_STARTUP_FUNCTION(plugin) {
    zend_class_entry ce;
    AP_INIT_CLASS_ENTRY(ce, "Ap_Plugin_Abstract", "Ap\\Plugin_Abstract", namespace_switch(ap_plugin_methods));
    ap_plugin_ce = zend_register_internal_class_ex(&ce, NULL, NULL TSRMLS_CC);
    ap_plugin_ce->ce_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
    return SUCCESS;
}
/* }}} */
/** {{{ ap_plugin_methods
*/
zend_function_entry ap_plugin_methods[] = {
    PHP_ME(ap_plugin, routerStartup,        plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, routerShutdown,       plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, dispatchLoopStartup,  plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, dispatchLoopShutdown, plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, preDispatch,          plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, postDispatch,         plugin_arg, ZEND_ACC_PUBLIC)
    PHP_ME(ap_plugin, preResponse,          plugin_arg, ZEND_ACC_PUBLIC)
    {NULL, NULL, NULL}
};
//六个方法入参都是一样的：
/** {{{ ARG_INFO
 */
ZEND_BEGIN_ARG_INFO_EX(plugin_arg, 0, 0, 2)
    ZEND_ARG_OBJ_INFO(0, request, Ap_Request_Abstract, 0)
    ZEND_ARG_OBJ_INFO(0, response, Ap_Response_Abstract, 0)
ZEND_END_ARG_INFO()
//两个入参，request，response
//所以插件可以做很多事，也很灵活，但由于可以注册多个插件，而且HOOK调用时是遍历调用的，就有可能出现覆盖的结果
```

ODP框架中的saf框架就是通过AP插件的形式实现部分功能的。

第二个高亮的函数ap_dispatcher_handle就是最主要的逻辑处理了

ap_dispatcher_get_controller

```
ce = ap_dispatcher_get_controller(app_dir, Z_STRVAL_P(module),Z_STRVAL_P(controller), Z_STRLEN_P(controller),
is_def_module TSRMLS_CC);
//找到对应的controller类

directory_len = spprintf(&directory, 0, "%s%c%s", app_dir, DEFAULT_SLASH, AP_CONTROLLER_DIRECTORY_NAME);
//找到controller类文件的目录，例如：odp/app/cashdesk/controllers
//#define AP_CONTROLLER_DIRECTORY_NAME        "controllers"

class_len = spprintf(&class, 0, "%s%s%s", "Controller", AP_G(name_separator), controller);
//拼装类名，例如：class Controller_Main extends Ap_Controller_Abstract

class_lowercase = zend_str_tolower_dup(class, class_len);
//类名转小写

ap_internal_autoload(controller,len, &directory TSRMLS_CC)
//加载类文件，这里的controller参数被直接当成目录directory下的文件名了，所以controller文件需要区分大小写，要注意。
```

ap_dispatcher_handle主逻辑

```
action      = zend_read_property(request_ce, request, ZEND_STRL(AP_REQUEST_PROPERTY_NAME_ACTION), 1
TSRMLS_CC);
action_lower = zend_str_tolower_dup(Z_STRVAL_P(action), Z_STRLEN_P(action));
//获取action名称

func_name_len = spprintf(&func_name,  0, "%s%s", action_lower, "action");
//拼接一个action函数名。会先查看controller中有没有带action的函数

if (zend_hash_find(&((ce)->function_table), func_name, func_name_len + 1, (void **)&fptr) == SUCCESS) {
    //先看controller中有没有这个function，有的话就执行该函数，如indexaction：

else if ((ce = ap_dispatcher_get_action(app_dir, icontroller,
    Z_STRVAL_P(module), is_def_module, Z_STRVAL_P(action), Z_STRLEN_P(action) TSRMLS_CC))
    && (zend_hash_find(&(ce)->function_table, AP_ACTION_EXECUTOR_NAME,
    sizeof(AP_ACTION_EXECUTOR_NAME), (void **)&fptr) == SUCCESS)) {

//这里第一步ap_dispatcher_get_action拿到action，然后查看action中是否有execute的成员函数。有的话就调用execute，也就是我们的业务主体逻辑。
    //到这里基本就执行完了一次请求，我们所有的业务逻辑都是通过execute入口的，在execute中调用PS层等等
    //#define AP_ACTION_EXECUTOR_NAME        "execute"
```

上面ap_dispatcher_get_action就是为了找到action，我们看一下它的实现：

```
actions_map = zend_read_property(Z_OBJCE_P(controller), controller,
ZEND_STRL(AP_CONTROLLER_PROPERTY_NAME_ACTIONS), 1 TSRMLS_CC);
//获取contraller中的actions变量。    #define AP_CONTROLLER_PROPERTY_NAME_ACTIONS    "actions"
```

例如：

```
public $actions = array(
'cashier_wireless_direct' => 'actions/cashier/Wireless.php',
    'cashier_wireless_direct_nologin' => 'actions/cashier/Wireless.php',
    'cashier_wireless_reservable' => 'actions/cashier/Wireless.php',
    'cashier_wireless_transfer' => 'actions/cashier/Wireless.php',
    'cashier_wireless_charge' => 'actions/cashier/Wireless.php',
);
```

上面的actions_map就是对应的这个自己定义的数组。

```
if (zend_hash_find(Z_ARRVAL_P(actions_map), action, len + 1, (void **)&ppaction) == SUCCESS) {
//在数组中找到对应的key（action名称），并把值（后面的文件路径）赋给ppaction。也就是得到了action类所在的文件了

action_path_len = spprintf(&action_path, 0, "%s%c%s", app_dir, DEFAULT_SLASH, Z_STRVAL_PP(ppaction));
if (ap_loader_import(action_path, action_path_len, 0 TSRMLS_CC)) {
   //拼装action文件完整路径，并加载。ap_loader_import是AP的自动加载功能，后面会单独介绍ap_loader
```

拼装action类名：

```
char *action_upper = estrndup(action, len);//action就是从路由环节解析出的action名称
*(action_upper) = toupper(*action_upper);//首字母大写
class_len = spprintf(&class, 0, "%s%s%s", "Action", AP_G(name_separator), action_upper);//加上Action前缀
class_lowercase = zend_str_tolower_dup(class, class_len);//全转小写，用来查找类
if (zend_hash_find(EG(class_table), class_lowercase, class_len + 1, (void **) &ce) == SUCCESS) {
   //在class_table中找到该类，并赋值给ce
if (instanceof_function(*ce, ap_action_ce TSRMLS_CC)) {//如果是ap_action_ce类型的，则返回ce
    efree(class);
    return *ce;
```

上面return后就获得了action类，同时也看到action的路径是按照你所填写的映射中地址加载，但是类名却是action的名称拼接的，所以虽然类文件不需要按照AP的标准路径设定，但是类名必须和action一致。有同学在这个环节可能会因为action的特殊性出现找不到类的问题。

2.2 路由Ap_Route

前面run流程中的路由模块就是利用内置路由策略或者你自定义的路由策略，通过解析URL获取contraller、action、model的名称。

这里分为两层的概念，一层我称之为路由器，在路由器下有对应的路由策略或者路由协议，官方手册中介绍了六种内置路由协议的功能。

```
/** {{{ int ap_dispatcher_route(ap_dispatcher_t *dispatcher, ap_request_t *request TSRMLS_DC)
 */
int ap_dispatcher_route(ap_dispatcher_t *dispatcher, ap_request_t *request TSRMLS_DC) {
    zend_class_entry *router_ce;
    ap_router_t *router = zend_read_property(ap_dispatcher_ce, dispatcher,
ZEND_STRL(AP_DISPATCHER_PROPERTY_NAME_ROUTER), 1 TSRMLS_CC);
    if (IS_OBJECT == Z_TYPE_P(router)) {
        if ((router_ce = Z_OBJCE_P(router)) == ap_router_ce) {
            /* use built-in router */
            ap_router_route(router, request TSRMLS_CC);//调用内置路由器
        } else {
            /* user custom router */
            zval *ret = zend_call_method_with_1_params(&router, router_ce, NULL, "route", &ret, request);//自定义路由器
            if (Z_TYPE_P(ret) == IS_BOOL && Z_BVAL_P(ret) == 0) {
                ap_trigger_error(AP_ERR_ROUTE_FAILED TSRMLS_CC, "Routing request faild");
                return 0;
            }
        }
        return 1;
    }
    return 0;
}
/* }}} */
```

这段代码是路由环节的入口，dispatcher初始化时会创建内置路由器，这里只涉及路由器概念，上面的自定义并不是自定义路由协议，而是你可以重新写一个路由器，我们通常在项目中自定义路由协议就可以了，没有必要自己实现一个路由器。而且框架中其实也是写死了内置路由器，没有给你set自定义路由器的接口。

```
int ap_router_route(ap_router_t *router, ap_request_t *request TSRMLS_DC) {
    zval         *routers, *ret;
    ap_route_t    **route;
    HashTable     *ht;
    routers = zend_read_property(ap_router_ce, router, ZEND_STRL(AP_ROUTER_PROPERTY_NAME_ROUTERS), 1
TSRMLS_CC);
    ht = Z_ARRVAL_P(routers);
    for(zend_hash_internal_pointer_end(ht);
        zend_hash_has_more_elements(ht) == SUCCESS;
        zend_hash_move_backwards(ht)) {
        if (zend_hash_get_current_data(ht, (void**)&route) == FAILURE) {
            continue;
        }
        zend_call_method_with_1_params(route, Z_OBJCE_PP(route), NULL, "route", &ret, request);
        if (IS_BOOL != Z_TYPE_P(ret) || !Z_BVAL_P(ret)) {
            zval_ptr_dtor(&ret);
            continue;
        } else {
            char *key;
            uint len = 0;
            ulong idx = 0;
            switch(zend_hash_get_current_key_ex(ht, &key, &len, &idx, 0, NULL)) {
                case HASH_KEY_IS_LONG:
                    zend_update_property_long(ap_router_ce, router,
ZEND_STRL(AP_ROUTER_PROPERTY_NAME_CURRENT_ROUTE), idx TSRMLS_CC);
                    break;
                case HASH_KEY_IS_STRING:
                    if (len) {
                        zend_update_property_string(ap_router_ce, router,
ZEND_STRL(AP_ROUTER_PROPERTY_NAME_CURRENT_ROUTE), key TSRMLS_CC);
                    }
                    break;
            }
            ap_request_set_routed(request, 1 TSRMLS_CC);
            zval_ptr_dtor(&ret);
            break;
        }
    }
    return 1;
}
```

上面这段代码很简单，所以就贴出完整的实现。这里简单解释一下：

首先拿到所有的路由协议routers（你可以添加多层路由协议，类似于多重插件），for循环依次调用路由协议的“route”方法，成功则记下当前生效的这个路由协议的索引位置，并设置request为已路由。不成功则继续调用下一个路由协议。

路由器的实现类是：

```
/** {{{ ap_router_methods
 */
zend_function_entry ap_router_methods[] = {
    PHP_ME(ap_router, __construct, NULL, ZEND_ACC_PUBLIC|ZEND_ACC_CTOR)
    PHP_ME(ap_router, addRoute,  NULL, ZEND_ACC_PUBLIC)
    PHP_ME(ap_router, addConfig, NULL, ZEND_ACC_PUBLIC)
    PHP_ME(ap_router, route,     NULL, ZEND_ACC_PUBLIC)
    PHP_ME(ap_router, getRoute,  NULL, ZEND_ACC_PUBLIC)
    PHP_ME(ap_router, getRoutes, NULL, ZEND_ACC_PUBLIC)
    PHP_ME(ap_router, getCurrentRoute, NULL, ZEND_ACC_PUBLIC)
    {NULL, NULL, NULL}
};
/* }}} */
/** {{{ AP_STARTUP_FUNCTION
 */
AP_STARTUP_FUNCTION(router) {
    zend_class_entry ce;
    AP_INIT_CLASS_ENTRY(ce, "Ap_Router", "Ap\\Router", ap_router_methods);
    ap_router_ce = zend_register_internal_class_ex(&ce, NULL, NULL TSRMLS_CC);
    ap_router_ce->ce_flags |= ZEND_ACC_FINAL_CLASS;
    zend_declare_property_null(ap_router_ce, ZEND_STRL(AP_ROUTER_PROPERTY_NAME_ROUTERS),
ZEND_ACC_PROTECTED TSRMLS_CC);
    zend_declare_property_null(ap_router_ce, ZEND_STRL(AP_ROUTER_PROPERTY_NAME_CURRENT_ROUTE),
ZEND_ACC_PROTECTED TSRMLS_CC);
    AP_STARTUP(route);
    return SUCCESS;
}
/* }}} */
```

你可以通过addRoute方法向路由器注册自己的路由协议

在其构造函数中有：

```
if (!AP_G(default_route)) {
static_route:
    MAKE_STD_ZVAL(route);
    object_init_ex(route, ap_route_static_ce);
```

表明默认的路由协议是ap_route_static_ce静态路由。

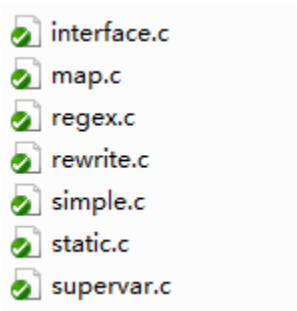
路由协议需要实现协议接口

```
/** {{{ ap_route_methods
 */
zend_function_entry ap_route_methods[] = {
    PHP_ABSTRACT_ME(ap_route, route, ap_route_route_arginfo)
    {NULL, NULL, NULL}
};
/* }}} */
/** {{{ AP_STARTUP_FUNCTION
 */
AP_STARTUP_FUNCTION(route) {
    zend_class_entry ce;
    AP_INIT_CLASS_ENTRY(ce, "Ap_Route_Interface", "Ap\\Route_Interface", ap_route_methods);
    ap_route_ce = zend_register_internal_interface(&ce TSRMLS_CC);
    AP_STARTUP(route_static);
    AP_STARTUP(route_simple);
    AP_STARTUP(route_supervar);
    AP_STARTUP(route_rewrite);
    AP_STARTUP(route_regex);
    AP_STARTUP(route_map);
    return SUCCESS;
}
/* }}} */
```

ap_route_ce就是协议接口，千万别看眼花了，这个和前面的ap_router_ce不是一个东西。代码中用router表示路由器，route表示路由协议，很容易看错。

路由协议很简单，就是实现route接口，AP框架会预加载六个内置的协议，对应的代码路径：

php-ap_1-1-5-0_PD_BL\routes



代码逻辑比较简单，这里不再详细介绍了，大家可以自己翻阅一下上面路径下的代码。每个路由的功能都是为了设置module，controller，action的名称

```
if (module != NULL) {
    zend_update_property_string(ap_request_ce, request, ZEND_STRL(AP_REQUEST_PROPERTY_NAME_MODULE),
    efree(module);
}
if (controller != NULL) {
    zend_update_property_string(ap_request_ce, request, ZEND_STRL(AP_REQUEST_PROPERTY_NAME_CONTROLLI
    efree(controller);
}

if (action != NULL) {
    zend_update_property_string(ap_request_ce, request, ZEND_STRL(AP_REQUEST_PROPERTY_NAME_ACTION),
    efree(action);
}
```

2.3 自动加载ap_loader

了解过ODP开发的同学都知道在AP框架中类是自动加载的，不需要提前require，只有在使用时才会自动加载，你只要按照规定的路径放置就行，下面我们介绍一下ap_loader加载器

AP主要通过spl_autoload方式实现的，这是php标准库中的一个方法。

spl_autoload
spl_autoload解决将不同开发者的类拦截器函数都注册到自动加载函数的hashtable中。spl实现自动加载的机制是维护一个hashtable，里面存储有具有自动加载功能的各个函数。当触发自动加载机制时，zend会在遍历执行这个hashtable里面的函数，直到成功加载类或加载失败后返回。

当需要使用自动加载功能时，使用函数spl_autoload_register()或spl_autoload_register('autoloadfuncitonname')，无参的spl_autoload_register()会默认加载spl_autoload()函数，该函数功能有限，只能在inlcude_path中搜索指定扩展名的类库。有参的spl_autoload_register()默认不再加载spl_autoload()函数。

可以通过spl_autoload_functions()查看当前自动加载hashtable中的函数，该函数返回一个数组。注意，使用spl_autoload时，系统会忽略拦截器__autoload，除非显式地使用spl_autoload_register('__autoload')将其加入hashtable。

利用spl_autoload_register把Ap_Loader的自动加载器注册进去，实现ap加载类文件的规则。

ap_loader_instance初始化加载器的入口

```
MAKE_STD_ZVAL(instance);
object_init_ex(instance, ap_loader_ce);
//分配内存，并初始化为ap_loader_ce类型，这样就创建了该类

if(!ap_loader_register(instance TSRMLS_CC)) {
    return NULL;
}
//把自己的加载函数注册到spl中
```

再进一步看一下ap_loader_register

ap_loader_register

```
int ap_loader_register(ap_loader_t *loader TSRMLS_DC)
。。。。。。
    MAKE_STD_ZVAL(method);
    ZVAL_STRING(method, AP_AUTOLOAD_FUNC_NAME, 1);
    zend_hash_next_index_insert(Z_ARRVAL_P(autoload), &loader, sizeof(ap_loader_t *), NULL);
    zend_hash_next_index_insert(Z_ARRVAL_P(autoload), &method, sizeof(zval *), NULL);
    MAKE_STD_ZVAL(function);
    ZVAL_STRING(function, AP_SPL_AUTOLOAD_REGISTER_NAME, 0);
。。。。。。
//这里有两个定义：
//#define AP_SPL_AUTOLOAD_REGISTER_NAME         "spl_autoload_register"
//#define AP_AUTOLOAD_FUNC_NAME                 "autoload"
//这是将loader 类的"autoload"方法，调用"spl_autoload_register"方法，也就是注册一个autoload方法。
//所以自动加载的真正入口就是autoload方法
```

autoload方法：

PHP_METHOD(ap_loader, autoload)

```
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &class_name, &class_name_len) == FAILURE) {
    return;
}
//取出你要加载的类名
if (strncmp(class_name, AP_LOADER_RESERVERD, AP_LOADER_LEN_RESERVERD) == 0) {
    php_error_docref(NULL TSRMLS_CC, E_WARNING, "You should not use '%s' as class name prefix",
AP_LOADER_RESERVERD);
}
//自己的类不可以用Ap_开头
//#define AP_LOADER_RESERVERD                "Ap_"
//#define AP_LOADER_LEN_RESERVERD            3
//Ap框架对于MVC的目录划分是固定的，就是在这里做检查和区分的
if (ap_loader_is_category(class_name, class_name_len, AP_LOADER_MODEL, AP_LOADER_LEN_MODEL TSRMLS_CC)) {
    /* this is a model class */
    spprintf(&directory, 0, "%s/%s", app_directory, AP_MODEL_DIRECTORY_NAME);
    file_name_len = class_name_len - separator_len - AP_LOADER_LEN_MODEL;
    if (AP_G(name_suffix)) {
        file_name = estrndup(class_name, file_name_len);
    } else {
        file_name = estrdup(class_name + AP_LOADER_LEN_MODEL + separator_len);
    }
    break;
}
if (ap_loader_is_category(class_name, class_name_len, AP_LOADER_PLUGIN, AP_LOADER_LEN_PLUGIN TSRMLS_CC))
{
    /* this is a plugin class */
    spprintf(&directory, 0, "%s/%s", app_directory, AP_PLUGIN_DIRECTORY_NAME);
    file_name_len = class_name_len - separator_len - AP_LOADER_LEN_PLUGIN;
    if (AP_G(name_suffix)) {
        file_name = estrndup(class_name, file_name_len);
    } else {
        file_name = estrdup(class_name + AP_LOADER_LEN_PLUGIN + separator_len);
    }
    break;
}
if (ap_loader_is_category(class_name, class_name_len, AP_LOADER_CONTROLLER, AP_LOADER_LEN_CONTROLLER
TSRMLS_CC)) {
    /* this is a controller class */
    spprintf(&directory, 0, "%s/%s", app_directory, AP_CONTROLLER_DIRECTORY_NAME);
    file_name_len = class_name_len - separator_len - AP_LOADER_LEN_CONTROLLER;
    if (AP_G(name_suffix)) {
        file_name = estrndup(class_name, file_name_len);
    } else {
        file_name = estrdup(class_name + AP_LOADER_LEN_CONTROLLER + separator_len);
    }
    break;
}
//这几段相似的逻辑是根据类名判断所属的模块，比如以"Controller"为标识的，就把类文件的查找路劲设为#define
AP_CONTROLLER_DIRECTORY_NAME "controllers"。
```

PHP_METHOD(ap_loader, autoload)

```
if (AP_G(st_compatible) && (strncmp(class_name, AP_LOADER_DAO, AP_LOADER_LEN_DAO) == 0
 || strncmp(class_name, AP_LOADER_SERVICE, AP_LOADER_LEN_SERVICE) == 0)) {
    /* this is a model class */
    spprintf(&directory, 0, "%s/%s", app_directory, AP_MODEL_DIRECTORY_NAME);
 }
//Dao层和Service层都会先被定位到#define AP_MODEL_DIRECTORY_NAME "models"目录中。
```

if (!AP_G(use_spl_autoload))

这是一个控制开关，如果在ap配置中设置为0，意思是加载失败不返回false，只打印一个错误日志。如果设置为1，则RETURN_FALSE。

这两种有什么区别呢，在spl_autoload机制中前面提到过，是可以注册多个自动加载方法的，如果第一个方法返回false，spl会自动调用后一个注册的加载方法，直到遍历完所有的加载方法。如果不返回false，则自动加载结束。这里会涉及到php执行的性能问题，遍历加载器是很耗时的事情。

当定位好主目录后调用ap_internal_autoload加载类：

ap_internal_autoload

```
smart_str buf = {0};//这里会存储最终获取的文件加载路径

smart_str_appendl(&buf,*directory, strlen(*directory));//把路径添加到buf中

//把下划线 _ 拆分成路径名
p = file_name;
q = p;
while (1) {
    while(++q && *q != '_' && *q != '\0');
    if (*q != '\0') {
        seg_len     = q - p;
        seg         = estrndup(p, seg_len);
        smart_str_appendl(&buf, seg, seg_len);
        efree(seg);
        smart_str_appendc(&buf, DEFAULT_SLASH);
        p = q + 1;
    } else {
        break;
    }
}

if (AP_G(lowcase_path)) {
    /* all path of library is lowercase */
    zend_str_tolower(buf.c + directory_len, buf.len - directory_len);
}//转小写

//添加.php后缀
smart_str_appendl(&buf, p, strlen(p));
smart_str_appendc(&buf, '.');
smart_str_appendl(&buf, ext, strlen(ext));

status = ap_loader_import(buf.c, buf.len, 0 TSRMLS_CC);
```

加载类文件，import的任务很简单

1. op_array = zend_compile_file(&file_handle, ZEND_INCLUDE TSRMLS_CC);//编译

2. zend_execute(op_array TSRMLS_CC); //执行

smart_str
smart字符串是PHP内核中常用的字符串类，并且定义了一系列高效的操作函数

```
typedef struct {
    char *c;
    size_t len;
    size_t a;
} smart_str;

#define smart_str_appendl(dest, src, len) \
    smart_str_appendl_ex((dest), (src), (len), 0)

#define smart_str_appendl_ex(dest, src, nlen, what) do {            \
    register size_t __nl;                                           \
    smart_str *__dest = (smart_str *) (dest);                       \
                                                                    \
    smart_str_alloc4(__dest, (nlen), (what), __nl);                 \
    memcpy(__dest->c + __dest->len, (src), (nlen));                 \
    __dest->len = __nl;                                             \
} while (0)

#define smart_str_alloc4(d, n, what, newlen) do {                   \
    if (!(d)->c) {                                                  \
        (d)->len = 0;                                               \
        newlen = (n);                                               \
        (d)->a = newlen < SMART_STR_START_SIZE                      \
              ? SMART_STR_START_SIZE                                \
              : newlen + SMART_STR_PREALLOC;                        \
        SMART_STR_DO_REALLOC(d, what);                              \
    } else {                                                        \
        newlen = (d)->len + (n);                                    \
        if (newlen >= (d)->a) {                                     \
            (d)->a = newlen + SMART_STR_PREALLOC;                   \
            SMART_STR_DO_REALLOC(d, what);                          \
        }                                                           \
    }                                                               \
} while (0)
```

上面展开了一个常用的宏，其主要好处是包含一个预分配机制，并通过len表示内容长度，a表示内存大小。

上面几章基本上把AP框架的主要代码都深入地分析了一遍，希望以此做引导，帮助大家更容易的去阅读AP源码。最后再通过一个简单的实例贯串一下AP框架处理流程，达到深入并浅出的效果。

3. 实例应用

举一个简单的例子，比如我们请求一个http://localhost/newapp/sample?id=1

第一个newapp会被认为是app名称，作为app入口的根目录，首先调用路由协议解析出action名是sample，根据前面解释的，先找到controllers目录下的controller文件

```
|-- controllers
| |-- Api.php
| `-- Main.php
```

这里找到了Main.php。（为何是这个文件，注意Bootstrap文件内容）

```
class Controller_Main extends Ap_Controller_Abstract {
        public $actions = array(
                'sample' => 'actions/Sample.php',
        );
}
```

找到了对应的名叫sample的action，直接加载后面路径指定的文件。

```
class Action_Sample extends Ap_Action_Abstract {

        public function execute() {
                //1. check if user is login as needed
                $arrUserinfo = Saf_SmartMain::getUserInfo();
                if (empty($arrUserinfo)) {
                //ouput error
        }
```

action类名是由sample拼装出来的，只需要实现execute方法，框架就会自动调用这个方法。

这样就完成了一次请求。

附录

AP源码路径：https://svn.baidu.com/inf/odp/tags/php-ap/php-ap_1-1-5-0_PD_BL