Подробно об объектах и классах в PHP

<http://jpauli.github.io/2015/03/24/zoom-on-php-objects.html>

* [Разработка веб-сайтов](https://habrahabr.ru/hub/webdev/),
* [PHP](https://habrahabr.ru/hub/php/),
* [Блог компании Mail.Ru Group](https://habrahabr.ru/company/mailru/)
* Перевод

Сегодня объекты используются очень активно, хотя это трудно было предположить после выхода PHP 5 в 2005 году. Тогда я ещё мало что знал о возможностях этого языка. Пятую версию PHP сравнивали с предыдущей, четвёртой, и главным преимуществом нового релиза стала новая, очень мощная объектная модель. И сегодня, десять лет спустя, около 90% всего PHP-кода содержит объекты, не изменившиеся со времени PHP 5.0. Это убедительно говорит о том, какую роль сыграло внедрение объектной модели, неоднократно улучшавшейся на протяжении последующих лет. В этом посте я хотел бы рассказать о том, как всё устроено «под капотом». Чтобы люди понимали суть процессов — почему сделано так, а не иначе — и лучше, полнее использовали возможности языка. Также я затрону тему использования памяти объектами, в том числе в сравнении с эквивалентными массивами (когда это возможно).

Я буду рассказывать на примере версии PHP 5.4, и описываемые мной вещи справедливы для 5.5 и 5.6, потому что устройство объектной модели там почти не претерпело изменений. Обратите внимание, что в версии 5.3 всё не так хорошо с точки зрения возможностей и общей производительности.

В PHP 7, который пока ещё активно разрабатывается, объектная модель переработана не сильно, были внесены лишь незначительные изменения. Просто потому что всё и так хорошо работает, а лучшее — враг хорошего. Были добавлены возможности, не затрагивающие ядро, но здесь об этом речи не пойдёт.

В качестве демонстрации начну с синтетических бенчмарков:

class Foo {

 public $a = "foobarstring";

 public $b;

 public $c = ['some', 'values'];

}

for ($i=0; $i<1000; $i++) {

 $m = memory\_get\_usage();

 ${'var'.$i} = new Foo;

 echo memory\_get\_usage() - $m"\n";

}

Здесь объявляется простой класс с тремя атрибутами, а затем в цикле создаётся 1000 объектов этого класса. Обратите внимание, как в этом примере используется память: при создании объекта класса Foo и переменной для его хранения выделяется 262 байт динамической памяти PHP.

Давайте заменим объект на эквивалентный массив:

for ($i=0; $i<1000; $i++) {

 $m = memory\_get\_usage();

 ${'var'.$i} = [['some', 'values'], null, 'foobarstring'];

 echo memory\_get\_usage() - $m . "\n";

}

В данном случае используются те же элементы: сам массив, null и строковая переменная foobarstring. Вот только потребляется уже 1160 байт памяти, что в 4,4 раза больше.

Вот ещё один пример:

$class = <<<'CL'

class Foo {

 public $a = "foobarstring";

 public $b;

 public $c = ['some', 'values'];

}

CL;

echo memory\_get\_usage() . "\n";

eval($class);

echo memory\_get\_usage() . "\n";

Поскольку класс декларируется во время компиляции, то для декларирования и измерения используемой памяти (с помощью диспетчера памяти PHP) мы используем оператор eval(). При этом никакие объекты в данном коде не создаются. Объём задействованной памяти (diff memory) составляет 2216 байт.

Теперь давайте разберём, как всё это устроено в недрах PHP, подкрепив теорией практические наблюдения.

Всё начинается с классов

Внутри PHP класс представляется с помощью структуры zend\_class\_entry:

struct \_zend\_class\_entry {

 char type;

 const char \*name;

 zend\_uint name\_length;

 struct \_zend\_class\_entry \*parent;

 int refcount;

 zend\_uint ce\_flags;

 HashTable function\_table;

 HashTable properties\_info;

 zval \*\*default\_properties\_table;

 zval \*\*default\_static\_members\_table;

 zval \*\*static\_members\_table;

 HashTable constants\_table;

 int default\_properties\_count;

 int default\_static\_members\_count;

 union \_zend\_function \*constructor;

 union \_zend\_function \*destructor;

 union \_zend\_function \*clone;

 union \_zend\_function \*\_\_get;

 union \_zend\_function \*\_\_set;

 union \_zend\_function \*\_\_unset;

 union \_zend\_function \*\_\_isset;

 union \_zend\_function \*\_\_call;

 union \_zend\_function \*\_\_callstatic;

 union \_zend\_function \*\_\_tostring;

 union \_zend\_function \*serialize\_func;

 union \_zend\_function \*unserialize\_func;

 zend\_class\_iterator\_funcs iterator\_funcs;

 */\* handlers \*/*

 zend\_object\_value (\*create\_object)(zend\_class\_entry \*class\_type TSRMLS\_DC);

 zend\_object\_iterator \*(\*get\_iterator)(zend\_class\_entry \*ce, zval \*object, int by\_ref TSRMLS\_DC);

 int (\*interface\_gets\_implemented)(zend\_class\_entry \*iface, zend\_class\_entry \*class\_type TSRMLS\_DC); */\* a class implements this interface \*/*

 union \_zend\_function \*(\*get\_static\_method)(zend\_class\_entry \*ce, char\* method, int method\_len TSRMLS\_DC);

 */\* serializer callbacks \*/*

 int (\*serialize)(zval \*object, unsigned char \*\*buffer, zend\_uint \*buf\_len, zend\_serialize\_data \*data TSRMLS\_DC);

 int (\*unserialize)(zval \*\*object, zend\_class\_entry \*ce, const unsigned char \*buf, zend\_uint buf\_len, zend\_unserialize\_data \*data TSRMLS\_DC);

 zend\_class\_entry \*\*interfaces;

 zend\_uint num\_interfaces;

 zend\_class\_entry \*\*traits;

 zend\_uint num\_traits;

 zend\_trait\_alias \*\*trait\_aliases;

 zend\_trait\_precedence \*\*trait\_precedences;

 union {

 struct {

 const char \*filename;

 zend\_uint line\_start;

 zend\_uint line\_end;

 const char \*doc\_comment;

 zend\_uint doc\_comment\_len;

 } user;

 struct {

 const struct \_zend\_function\_entry \*builtin\_functions;

 struct \_zend\_module\_entry \*module;

 } internal;

 } info;

};

Размер структуры, исходя из модели LP64, составляет **568 байт**. То есть каждый раз, когда PHP декларирует класс, он вынужден создавать zend\_class\_entry, используя только для этого более половины килобайта динамической памяти. Конечно, дело этим не ограничивается: как вы заметили, структура содержит немало указателей, которые тоже надо разместить в памяти. То есть сами по себе классы потребляют памяти гораздо больше, чем все создаваемые из них впоследствии объекты.

Помимо прочего, классы содержат атрибуты (статические и динамические), а также методы. Всё это тоже требует памяти. Что касается методов, то здесь сложно вычислить точную зависимость, но одно верно: чем больше тело метода, тем больше его OPArray, а значит, тем больше памяти он потребляет. Добавьте к этому статические переменные, которые могут быть объявлены в методе. Далее идут атрибуты, позже они тоже будут размещены в памяти. Объём зависит от их значений по умолчанию: целочисленные займут немного, а вот большой статический массив съест немало памяти.

Важно знать ещё об одном моменте, связанном с zend\_class\_entry — о PHP-комментариях. Они также известны как аннотации. Это строковые переменные (в языке С — буферы char\*), которые тоже надо разместить в памяти. Для языка С, не использующего Unicode, в отличие от PHP, правило очень простое: один символ = один байт. Чем больше у вас в классе аннотаций, тем больше памяти будет использовано после парсинга.

У zend\_class\_entry поле doc\_comment содержит аннотации класса. У методов и атрибутов тоже есть такое поле.

Пользовательские и внутренние классы

Пользовательский класс — это класс, заданный с помощью PHP, а внутренний класс задаётся либо благодаря внедрению исходного кода в сам PHP, либо с помощью расширения. Самое большое различие между этими двумя видами классов заключается в том, что пользовательские классы оперируют памятью, выделяемой по запросу, а внутренние — «постоянной» памятью.

Это означает, что когда PHP заканчивает обработку текущего HTTP-запроса, он убирает из памяти и уничтожает все пользовательские классы, готовясь к обработке следующего запроса. Этот подход известен под названием «архитектура без разделения ресурсов» (the share nothing architecture). Так было заложено в PHP с самого начала, и изменять это пока не планируется.

Итак, каждый раз при формировании запроса и парсинге классов происходит выделение памяти для них. После использования класса уничтожается всё, что с ним связано. Так что обязательно используйте все объявленные классы, в противном случае будет теряться память. Применяйте автозагрузчики, они задерживают парсинг/объявление во время выполнения, когда PHP нужно задействовать класс. Несмотря на замедление выполнения, автозагрузчик позволяет грамотно использовать память, поскольку он не будет запущен, пока действительно не возникнет потребность в классе.

С внутренними классами всё иначе. Они размещаются в памяти постоянно, вне зависимости от того, использовали их или нет. То есть они уничтожаются только тогда, когда прекращается работа самого PHP — после завершения обработки всех запросов (подразумеваются веб SAPI, например, PHP-FPM). Поэтому внутренние классы более эффективны, чем пользовательские (в конце запроса уничтожаются только статические атрибуты, больше ничего).

if (EG(full\_tables\_cleanup)) {

 zend\_hash\_reverse\_apply(EG(function\_table), (apply\_func\_t) clean\_non\_persistent\_function\_full TSRMLS\_CC);

 zend\_hash\_reverse\_apply(EG(class\_table), (apply\_func\_t) clean\_non\_persistent\_class\_full TSRMLS\_CC);

} else {

 zend\_hash\_reverse\_apply(EG(function\_table), (apply\_func\_t) clean\_non\_persistent\_function TSRMLS\_CC);

 zend\_hash\_reverse\_apply(EG(class\_table), (apply\_func\_t) clean\_non\_persistent\_class TSRMLS\_CC);

}

static int clean\_non\_persistent\_class(zend\_class\_entry \*\*ce TSRMLS\_DC)

{

 return ((\*ce)->type == ZEND\_INTERNAL\_CLASS) ? ZEND\_HASH\_APPLY\_STOP : ZEND\_HASH\_APPLY\_REMOVE;

}

Обратите внимание, что даже при кешировании опкодов, как OPCache, создание и уничтожение класса осуществляется при каждом запросе, как и в случае с пользовательскими классами. OPCache просто ускоряет оба этих процесса.

Как вы заметили, если активировать много PHP-расширений, каждое из которых объявляет много классов, но при этом использовать лишь небольшое их количество, то теряется память. Помните, что PHP-расширения объявляют классы во время запуска PHP, даже если в последующих запросах эти классы использоваться не будут. Поэтому не рекомендуется держать расширения активными, если они не применяются в данный момент, иначе вы будете терять память. Особенно если эти расширения объявляют много классов — хотя они могут забить память и чем-нибудь другим.

Классы, интерфейсы или трейты — без разницы

Для управления классами, интерфейсами и трейтами в PHP используется одна и та же структура — zend\_class\_entry. И как вы уже видели, эта структура весьма громоздка. Иногда разработчики объявляют в коде интерфейсы, чтобы иметь возможность использовать свои названия в catch-блоках. Это позволяет ловить только определённый вид исключений. Например, так:

interface BarException { }

class MyException extends Exception implements BarException { }

try {

 $foo->bar():

} catch (BarException $e) { }

Не слишком хорошо, что здесь используется 912 байт всего лишь для декларирования интерфейса BarException.

$class = <<<'CL'

interface Bar { }

CL;

$m = memory\_get\_usage();

eval($class);

echo memory\_get\_usage() - $m . "\n"; */\* 912 bytes \*/*

Не хочу сказать, что это плохо или глупо, я не пытаюсь никого и ничто обвинять. Просто обращаю ваше внимание на этот момент. С точки зрения внутренней структуры PHP, классы, интерфейсы и трейты используются совершенно одинаково. В интерфейс нельзя добавить атрибуты, парсер или компилятор просто не позволят этого сделать. Однако структура zend\_class\_entry никуда не девается, просто ряд полей, включая static\_members\_table, не будут размещёнными в памяти указателями. Объявление класса, эквивалентного трейта или эквивалентного интерфейса потребует одинакового объёма памяти, поскольку все они используют одну и ту же структуру.

Привязка класса

Многие разработчики не вспоминают о привязке класса, пока не начинают задавать вопросом, а как же всё устроено на самом деле. Привязку класса можно описать как «процесс, в ходе которого сам класс и все связанные с ним данные подготавливаются для полноценного использования разработчиком». Этот процесс очень прост и не требует много ресурсов, если речь идёт о каком-то одном классе, не дополняющем другой, не использующем трейты и не внедряющим интерфейс. Процесс привязки для таких классов полностью протекает во время компиляции, а в ходе выполнения ресурсы на это уже не тратятся. Обратите внимание, что речь шла привязке класса, задекларированного пользователем. Для внутренних классов тот же самый процесс выполняется, когда классы зарегистрированы ядром или расширениями PHP, как раз перед запуском пользовательских скриптов — и делается это лишь один раз за всё время работы PHP.

Всё сильно усложняется, если речь заходит о внедрении интерфейсов или наследовании классов. Тогда в ходе привязки класса у родительских и дочерних объектов (будь то классы или интерфейсы) копируется абсолютно все.

*/\* Single class \*/*

case ZEND\_DECLARE\_CLASS:

 if (do\_bind\_class(CG(active\_op\_array), opline, CG(class\_table), 1 TSRMLS\_CC) == NULL) {

 return;

 }

 table = CG(class\_table);

 break;

В случае простого объявления класса мы запускаем do\_bind\_class(). Эта функция всего лишь регистрирует полностью определённый класс в таблице классов с целью дальнейшего использования во время выполнения, а также осуществляет проверку на возможные абстрактные методы:

void zend\_verify\_abstract\_class(zend\_class\_entry \*ce TSRMLS\_DC)

{

 zend\_abstract\_info ai;

 if ((ce->ce\_flags & ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS) && !(ce->ce\_flags & ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS)) {

 memset(&ai, 0, sizeof(ai));

 zend\_hash\_apply\_with\_argument(&ce->function\_table, (apply\_func\_arg\_t) zend\_verify\_abstract\_class\_function, &ai TSRMLS\_CC);

 if (ai.cnt) {

 zend\_error(E\_ERROR, "Class %s contains %d abstract method%s and must therefore be declared abstract or implement the remaining methods (" MAX\_ABSTRACT\_INFO\_FMT MAX\_ABSTRACT\_INFO\_FMT MAX\_ABSTRACT\_INFO\_FMT ")",

 ce->name, ai.cnt,

 ai.cnt > 1 ? "s" : "",

 DISPLAY\_ABSTRACT\_FN(0),

 DISPLAY\_ABSTRACT\_FN(1),

 DISPLAY\_ABSTRACT\_FN(2)

 );

 }

 }

}

Тут добавить нечего, простой случай.

При привязке класса, внедряющего интерфейс, нужно осуществить следующие действия:

* Проверить, не объявлен ли уже интерфейс.
* Проверить, действительно ли нужный класс является классом, а не самим интерфейсом (как говорилось выше, с точки зрения внутренней структуры они устроены одинаково).
* Скопировать константы из интерфейса в класс, проверяя на наличие возможных коллизий.
* Скопировать методы из интерфейса в класс, проверяя на наличие возможных коллизий и несоответствий в декларировании (например, превращая в дочернем классе методы интерфейса в статические).
* Добавить интерфейс и все возможные материнские интерфейсы к списку интерфейсов, внедряемых классом.

Под «копированием» подразумевается не полное глубокое копирование. Для констант, атрибутов и функций по очереди ведется пересчет, сколько сущностей в памяти их использует.

ZEND\_API void zend\_do\_implement\_interface(zend\_class\_entry \*ce, zend\_class\_entry \*iface TSRMLS\_DC)

{

 */\* ... ... \*/*

 } else {

 if (ce->num\_interfaces >= current\_iface\_num) {

 if (ce->type == ZEND\_INTERNAL\_CLASS) {

 ce->interfaces = (zend\_class\_entry \*\*) realloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (++current\_iface\_num));

 } else {

 ce->interfaces = (zend\_class\_entry \*\*) erealloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (++current\_iface\_num));

 }

 }

 ce->interfaces[ce->num\_interfaces++] = iface;

 zend\_hash\_merge\_ex(&ce->constants\_table, &iface->constants\_table, (copy\_ctor\_func\_t) zval\_add\_ref, sizeof(zval \*), (merge\_checker\_func\_t) do\_inherit\_constant\_check, iface);

 zend\_hash\_merge\_ex(&ce->function\_table, &iface->function\_table, (copy\_ctor\_func\_t) do\_inherit\_method, sizeof(zend\_function), (merge\_checker\_func\_t) do\_inherit\_method\_check, ce);

 do\_implement\_interface(ce, iface TSRMLS\_CC);

 zend\_do\_inherit\_interfaces(ce, iface TSRMLS\_CC);

 }

}

Обратите внимание на разницу между внутренними и пользовательскими классами. Первые для распределения памяти будут использовать realloc(), вторые — erealloc(). realloc() распределяет «постоянную» память, а erealloc() оперирует памятью, «выделяемой по запросу».

Вы можете видеть, что, когда объединяются две константные таблицы (интерфейс-1 и класс-1), они делают это с помощью колбека zval\_add\_ref. Он не копирует константы из одной таблицы в другую, а расшаривает их указатели, просто добавляя количество референсов.

Для каждой из таблиц функций (методов) используется do\_inherit\_method:

static void do\_inherit\_method(zend\_function \*function)

{

 function\_add\_ref(function);

}

ZEND\_API void function\_add\_ref(zend\_function \*function)

{

 if (function->type == ZEND\_USER\_FUNCTION) {

 zend\_op\_array \*op\_array = &function->op\_array;

 (\*op\_array->refcount)++;

 if (op\_array->static\_variables) {

 HashTable \*static\_variables = op\_array->static\_variables;

 zval \*tmp\_zval;

 ALLOC\_HASHTABLE(op\_array->static\_variables);

 zend\_hash\_init(op\_array->static\_variables, zend\_hash\_num\_elements(static\_variables), NULL, ZVAL\_PTR\_DTOR, 0);

 zend\_hash\_copy(op\_array->static\_variables, static\_variables, (copy\_ctor\_func\_t) zval\_add\_ref, (void \*) &tmp\_zval, sizeof(zval \*));

 }

 op\_array->run\_time\_cache = NULL;

 }

}

К OPArray функции добавлен refcount, а также с помощью zval\_add\_ref скопированы все возможные статические переменные, объявленные в функции (здесь это метод). Таким образом, для всего процесса копирования нужно немало вычислительных ресурсов, потому что здесь задействовано много циклов и проверок. Но памяти задействуется немного. К сожалению, сегодня привязка интерфейса полностью протекает во время выполнения, и вы будете это чувствовать при каждом запросе. Возможно, скоро разработчики это изменят.

Что касается наследования, то здесь, в принципе, всё то же самое, что и при внедрении интерфейса. Только вовлечено ещё больше «участников». Но хочу отметить, что если PHP уже знает о классе, то привязка осуществляется во время компилирования, а если не знает — то во время выполнения. Так что лучше объявлять так:

*/\* good \*/*

class A { }

class B extends A { }

вместо:

*/\* bad \*/*

class B extends A { }

class A { }

Кстати, рутинная процедура привязки класса может привести к очень странному поведению:

*/\* это работает \*/*

class B extends A { }

class A { }

*/\* а это нет \*/*

Fatal error: Class 'B' not found \*/

class C extends B { }

class B extends A { }

class A { }

В первом варианте привязка класса В отложена на время выполнения, потому что когда компилятор доходит до объявления этого класса, он ещё ничего не знает о классе А. Когда начинается выполнение, то привязка класса А происходит без вопросов, потому что он уже скомпилирован, будучи одиночным классом. Во втором случае всё иначе. Привязка класса С отложена на время выполнения, потому что компилятор ещё ничего не знает о В, пытаясь скомпилировать его. Но когда во время выполнения начинается привязка класса С, то он ищет В, который не существует, поскольку не скомпилирован по причине того, что В является дополнением. Вылетает сообщение “Class B doesn’t exist”.

Объекты

Итак, теперь мы знаем, что:

* Классы занимают много памяти.
* Внутренние классы гораздо лучше оптимизированы по сравнению с пользовательскими, потому что последние должны быть созданы и уничтожены при каждом запросе. Внутренние классы существуют постоянно.
* Классы, интерфейсы и трейты используют одни и те же структуру и процедуры, различия очень малы.
* Во время наследования или объявления процесс привязки сильно и долго нагружает процессор, но памяти задействуется немного, поскольку многие вещи не дуплицируются, а используются совместно. Кроме того, лучше запускать привязку классов во время компиляции.

Теперь поговорим об объектах. В первой главе показано, что создание «классического» объекта («классического» пользовательского класса) потребовало очень мало памяти, около 200 байт. Всё дело в классе. Дальнейшая компиляция класса тоже потребляет память, но это к лучшему, потому что для создания одиночного объекта требуется меньше байт. По сути, объект представляет собой крохотный набор из крохотных структур.

Управление методами объекта

На уровне движка методы и функции являются одним и тем же — структурой zend\_function\_structure. Различаются лишь названия. Методы компилируются и добавляются к атрибуту function\_table в zend\_class\_entry. Поэтому во время выполнения представлен каждый метод, это лишь вопрос перевода указателя на исполнение.

typedef union \_zend\_function {

 zend\_uchar type;

 struct {

 zend\_uchar type;

 const char \*function\_name;

 zend\_class\_entry \*scope;

 zend\_uint fn\_flags;

 union \_zend\_function \*prototype;

 zend\_uint num\_args;

 zend\_uint required\_num\_args;

 zend\_arg\_info \*arg\_info;

 } common;

 zend\_op\_array op\_array;

 zend\_internal\_function internal\_function;

} zend\_function;

Когда объект пытается вызвать метод, то движок по умолчанию ищет в таблице значений функций класса этого объекта. Если метод не существует, то вызывается \_\_call(). Также проверяется видимость — public/protected/private — в зависимости от чего предпринимаются следующие действия:

static union \_zend\_function \*zend\_std\_get\_method(zval \*\*object\_ptr, char \*method\_name, int method\_len, const zend\_literal \*key TSRMLS\_DC)

{

 zend\_function \*fbc;

 zval \*object = \*object\_ptr;

 zend\_object \*zobj = Z\_OBJ\_P(object);

 ulong hash\_value;

 char \*lc\_method\_name;

 ALLOCA\_FLAG(use\_heap)

 if (EXPECTED(key != NULL)) {

 lc\_method\_name = Z\_STRVAL(key->constant);

 hash\_value = key->hash\_value;

 } else {

 lc\_method\_name = do\_alloca(method\_len+1, use\_heap);

 zend\_str\_tolower\_copy(lc\_method\_name, method\_name, method\_len);

 hash\_value = zend\_hash\_func(lc\_method\_name, method\_len+1);

 }

 */\* If the method is not found \*/*

 if (UNEXPECTED(zend\_hash\_quick\_find(&zobj->ce->function\_table, lc\_method\_name, method\_len+1, hash\_value, (void \*\*)&fbc) == FAILURE)) {

 if (UNEXPECTED(!key)) {

 free\_alloca(lc\_method\_name, use\_heap);

 }

 if (zobj->ce->\_\_call) { */\* if the class has got a \_\_call() handler \*/*

 return zend\_get\_user\_call\_function(zobj->ce, method\_name, method\_len); */\* call the \_\_call() handler \*/*

 } else {

 return NULL; */\* else return NULL, which will likely lead to a fatal error : method not found \*/*

 }

 }

 */\* Check access level \*/*

 if (fbc->op\_array.fn\_flags & ZEND\_ACC\_PRIVATE) {

 zend\_function \*updated\_fbc;

 updated\_fbc = zend\_check\_private\_int(fbc, Z\_OBJ\_HANDLER\_P(object, get\_class\_entry)(object TSRMLS\_CC), lc\_method\_name, method\_len, hash\_value TSRMLS\_CC);

 if (EXPECTED(updated\_fbc != NULL)) {

 fbc = updated\_fbc;

 } else {

 if (zobj->ce->\_\_call) {

 fbc = zend\_get\_user\_call\_function(zobj->ce, method\_name, method\_len);

 } else {

 zend\_error\_noreturn(E\_ERROR, "Call to %s method %s::%s() from context '%s'", zend\_visibility\_string(fbc->common.fn\_flags), ZEND\_FN\_SCOPE\_NAME(fbc), method\_name, EG(scope) ? EG(scope)->name : "");

 }

 }

 } else {

 */\* ... ... \*/*

}

Вы могли заметить интересную вещь, посмотрите на первые строки:

if (EXPECTED(key != NULL)) {

 lc\_method\_name = Z\_STRVAL(key->constant);

 hash\_value = key->hash\_value;

 } else {

 lc\_method\_name = do\_alloca(method\_len+1, use\_heap);

 */\* Create a zend\_copy\_str\_tolower(dest, src, src\_length); \*/*

 zend\_str\_tolower\_copy(lc\_method\_name, method\_name, method\_len);

 hash\_value = zend\_hash\_func(lc\_method\_name, method\_len+1);

 }

Это проявление невосприимчивости PHP к разным регистрам. Система сначала должна привести каждую функцию к нижнему регистру (zend\_str\_tolower\_copy()), прежде чем вызывать её. Не совсем каждую, а те, где присутствует оператор if. Переменная key предотвращает исполнение функции, переводящей в нижний регистр (часть с else) — это часть оптимизации PHP, осуществлённой в версии 5.4. Если вызов метода не динамический, то компилятор уже вычислил key, и во время выполнения тратится меньше ресурсов.

class Foo { public function BAR() { } }

$a = new Foo;

$b = 'bar';

$a->bar(); */\* static call : good \*/*

$a->$b(); */\* dynamic call : bad \*/*

Во время компиляции функции/метода происходит немедленный перевод в нижний регистр. Вышеприведённая функция BAR() превращается в bar() компилятором при добавлении метода таблице классов и функций.

В приведённом примере первый вызов статический: компилятор вычислил key для строковой “bar”, а когда приходит время вызова метода, ему нужно делать меньше работы. Второй вызов уже динамический, компилятор ничего не знает о “$b”, не может вычислить key для вызова метода. Затем, во время выполнения, нам придётся перевести строковую в нижний регистр и вычислить её хеш (zend\_hash\_func()), что не лучшим образом сказывается на производительности.

Что касается \_\_call(), то она не настолько сильно снижает производительность. Тем не менее, в этом случае тратится больше ресурсов, чем при вызове существующей функции.

Управление атрибутами объекта

Вот что происходит:



Как видите, когда создаётся несколько объектов одного класса, движок перенаправляет каждый атрибут на тот же указатель, что и в случае с атрибутами класса. На протяжении своей жизни класс хранит не только свои, статические, атрибуты, но также и атрибуты объектов. В случае с внутренними классами — в течение всего времени работы PHP. Создание объекта не подразумевает создания его атрибутов, так что это довольно быстрый и экономичный подход. Только когда объект собирается поменять один из своих атрибутов, движок создаёт для этого новый, предполагая, что вы меняете атрибут $a объекта Foo #2:



Так что, создавая объект, мы «всего лишь» создаём структуру zend\_object весом 32 байта:

typedef struct \_zend\_object {

 zend\_class\_entry \*ce;

 HashTable \*properties;

 zval \*\*properties\_table;

 HashTable \*guards; */\* protects from \_\_get/\_\_set ... recursion \*/*

} zend\_object;

Эта структура добавляется к хранилищу объектов. А им, в свою очередь, является структура zend\_object\_store. Это глобальный реестр объектов движка Zend — место, где собираются все объекты и хранятся в одном экземпляре:

ZEND\_API zend\_object\_value zend\_objects\_new(zend\_object \*\*object, zend\_class\_entry \*class\_type TSRMLS\_DC)

{

 zend\_object\_value retval;

 \*object = emalloc(sizeof(zend\_object));

 (\*object)->ce = class\_type;

 (\*object)->properties = NULL;

 (\*object)->properties\_table = NULL;

 (\*object)->guards = NULL;

 */\* Add the object into the store \*/*

 retval.handle = zend\_objects\_store\_put(\*object, (zend\_objects\_store\_dtor\_t) zend\_objects\_destroy\_object, (zend\_objects\_free\_object\_storage\_t) zend\_objects\_free\_object\_storage, NULL TSRMLS\_CC);

 retval.handlers = &std\_object\_handlers;

 return retval;

}

Далее движок создаёт вектор признаков нашего объекта:

ZEND\_API void object\_properties\_init(zend\_object \*object, zend\_class\_entry \*class\_type)

{

 int i;

 if (class\_type->default\_properties\_count) {

 object->properties\_table = emalloc(sizeof(zval\*) \* class\_type->default\_properties\_count);

 for (i = 0; i < class\_type->default\_properties\_count; i++) {

 object->properties\_table[i] = class\_type->default\_properties\_table[i];

 if (class\_type->default\_properties\_table[i]) {

#if ZTS

 ALLOC\_ZVAL( object->properties\_table[i]);

 MAKE\_COPY\_ZVAL(&class\_type->default\_properties\_table[i], object->properties\_table[i]);

#else

 Z\_ADDREF\_P(object->properties\_table[i]);

#endif

 }

 }

 object->properties = NULL;

 }

}

Как видите, мы разместили в памяти таблицу/вектор (как в языке С) для zval\*, основанный на объявленных свойствах класса объекта. В случае непоточнобезопасного PHP мы просто добавляем к признаку refcount, а если используется поточнобезопасный Zend (ZTS, Zend thread safety), то нужно полностью скопировать zval. Это один из многочисленных примеров, подтверждающих низкую производительность и высокую ресурсоёмкость режима ZTS по сравнению с не ZTS PHP.

Вероятно, у вас возникли два вопроса:

* Чем отличаются properties\_table и properties в структуре zend\_object?
* Если мы поместили атрибуты нашего объекта в С-вектор, то как вернуть их обратно? Каждый раз просматривать вектор (что снижает производительность)?

Ответ на оба вопроса даёт zend\_property\_info.

typedef struct \_zend\_property\_info {

 zend\_uint flags;

 const char \*name;

 int name\_length;

 ulong h;

 int offset;

 const char \*doc\_comment;

 int doc\_comment\_len;

 zend\_class\_entry \*ce;

} zend\_property\_info;

Каждый **объявленный** атрибут (свойство) нашего объекта имеет соответствующую информацию о свойстве, добавляемую в поле property\_info в zend\_class\_entry. Делается это во время компиляции объявленных в классе атрибутов:

class Foo

{

 public $a = 'foo';

 protected $b;

 private $c;

}

struct \_zend\_class\_entry {

 */\* ... ... \*/*

 HashTable function\_table;

 HashTable properties\_info; */\* here are the properties infos about $a, $b and $c \*/*

 zval \*\*default\_properties\_table; */\* and here, we'll find $a, $b and $c with their default values \*/*

 int default\_properties\_count; */\* this will have the value of 3 : 3 properties \*/*

 */\* ... ... \*/*

Properties\_infos представляет собой таблицу, сообщающую объекту о существовании запрашиваемого атрибута. И если он существует, то передаёт его индексный номер в массиве object->properties. Потом мы проверяем видимость и доступ к scope (public/protected/private).

Если же атрибут не существует и нам нужно записать в него, то можно попытаться вызвать \_\_set(). В случае неудачи создаём динамический атрибут, который будет храниться в поле object->property\_table.

property\_info = zend\_get\_property\_info\_quick(zobj->ce, member, (zobj->ce->\_\_set != NULL), key TSRMLS\_CC);

if (EXPECTED(property\_info != NULL) &&

 ((EXPECTED((property\_info->flags & ZEND\_ACC\_STATIC) == 0) &&

 property\_info->offset >= 0) ?

 (zobj->properties ?

 ((variable\_ptr = (zval\*\*)zobj->properties\_table[property\_info->offset]) != NULL) :

 (\*(variable\_ptr = &zobj->properties\_table[property\_info->offset]) != NULL)) :

 (EXPECTED(zobj->properties != NULL) &&

 EXPECTED(zend\_hash\_quick\_find(zobj->properties, property\_info->name, property\_info->name\_length+1, property\_info->h, (void \*\*) &variable\_ptr) == SUCCESS)))) {

*/\* ... ... \*/*

} else {

 zend\_guard \*guard = NULL;

 if (zobj->ce->\_\_set && */\* class has a \_\_set() ? \*/*

 zend\_get\_property\_guard(zobj, property\_info, member, &guard) == SUCCESS &&

 !guard->in\_set) {

 Z\_ADDREF\_P(object);

 if (PZVAL\_IS\_REF(object)) {

 SEPARATE\_ZVAL(&object);

 }

 guard->in\_set = 1; */\* prevent circular setting \*/*

 if (zend\_std\_call\_setter(object, member, value TSRMLS\_CC) != SUCCESS) { */\* call \_\_set() \*/*

 }

 guard->in\_set = 0;

 zval\_ptr\_dtor(&object);

 */\* ... ... \*/*

Пока вы не пишете в объект, его потребление памяти не меняется. После записи он занимает уже больше места (пока не будет уничтожен), поскольку содержит все записанные в него атрибуты.

Объекты, ведущие себя как ссылки благодаря хранилищу объектов

Объекты не являются ссылками. Это демонстрируется на маленьком скрипте:

function foo($var) {

 $var = 42;

}

$o = new MyClass;

foo($o);

var\_dump($o); */\* this is still an object, not the integer 42 \*/*

Все сейчас скажут, что «в PHP 5 объекты являются ссылками», об этом упоминает даже официальный мануал. Технически это совершенно неверно. Тем не менее, объекты могут вести себя так же, как и ссылки. Например, когда вы передаёте переменную, являющуюся объектом функции, эта функция может модифицировать тот же объект.

Так происходит потому, что zval, передаваемый в виде функции, передаёт не сам объект, а его уникальный идентификатор, используемый для поиска в общем хранилище объектов. А результат получается тот же самый. Можно разместить в памяти три разных zval, и все они могут содержать один и тот же дескриптор объекта.

object(MyClass)#1 (0) { } /\* #1 is the object handle (number), it is unique \*/



Zend\_object\_store обеспечивает однократное занесение объектов в память. Единственный способ записать в хранилище заключается в создании нового объекта с ключевым словом new, функцией unserialize(), reflection API или ключевым словом clone. Никакие другие операции не позволят дуплицировать или создать новый объект в хранилище.

typedef struct \_zend\_objects\_store {

 zend\_object\_store\_bucket \*object\_buckets;

 zend\_uint top;

 zend\_uint size;

 int free\_list\_head;

} zend\_objects\_store;

typedef struct \_zend\_object\_store\_bucket {

 zend\_bool destructor\_called;

 zend\_bool valid;

 zend\_uchar apply\_count;

 union \_store\_bucket {

 struct \_store\_object {

 void \*object;

 zend\_objects\_store\_dtor\_t dtor;

 zend\_objects\_free\_object\_storage\_t free\_storage;

 zend\_objects\_store\_clone\_t clone;

 const zend\_object\_handlers \*handlers;

 zend\_uint refcount;

 gc\_root\_buffer \*buffered;

 } obj;

 struct {

 int next;

 } free\_list;

 } bucket;

} zend\_object\_store\_bucket;

Что такое $this?

Понять устройство $this не так уж сложно, но с этим инструментом связаны куски кода в нескольких местах движка: в компиляторе, в коде получения переменных во время выполнения и т.д. $thisпоявляется и исчезает по мере необходимости, автоматически присваивая себе текущий объект — в общем, «волшебная» штука. А внутренний код прекрасно позволяет ей управлять.

Во-первых, компилятор не позволит записывать в $this. Для этого он проверят каждое осуществляемое вами присваивание, и если обнаруживает присвоение $this, то возникает фатальная ошибка.

*/\* ... ... \*/*

 if (opline\_is\_fetch\_this(last\_op TSRMLS\_CC)) {

 zend\_error(E\_COMPILE\_ERROR, "Cannot re-assign $this");

}

*/\* ... ... \*/*

static zend\_bool opline\_is\_fetch\_this(const zend\_op \*opline TSRMLS\_DC)

{

 if ((opline->opcode == ZEND\_FETCH\_W) && (opline->op1\_type == IS\_CONST)

 && (Z\_TYPE(CONSTANT(opline->op1.constant)) == IS\_STRING)

 && ((opline->extended\_value & ZEND\_FETCH\_STATIC\_MEMBER) != ZEND\_FETCH\_STATIC\_MEMBER)

 && (Z\_HASH\_P(&CONSTANT(opline->op1.constant)) == THIS\_HASHVAL)

 && (Z\_STRLEN(CONSTANT(opline->op1.constant)) == (sizeof("this")-1))

 && !memcmp(Z\_STRVAL(CONSTANT(opline->op1.constant)), "this", sizeof("this"))) {

 return 1;

 } else {

 return 0;

 }

}

Как управляется $this? Его использование возможно только внутри метода, во время вызова которого компилятор генерирует OPCode INIT\_METHOD\_CALL. Движок знает, кто вызывает метод, в случае с $a->foo() это $a. После чего извлекается значение $a и сохраняется в общем пространстве. Далее происходит вызов метода с помощью OPCode DO\_FCALL. На этом этапе снова извлекается сохранённое значение (объект вызывает метод) и присваивается глобальному внутреннему $this-указателю — EG(This).

if (fbc->type == ZEND\_USER\_FUNCTION || fbc->common.scope) {

 should\_change\_scope = 1;

 EX(current\_this) = EG(This);

 EX(current\_scope) = EG(scope);

 EX(current\_called\_scope) = EG(called\_scope);

 EG(This) = EX(object); */\* fetch the object prepared in previous INIT\_METHOD opcode and affect it to EG(This) \*/*

 EG(scope) = (fbc->type == ZEND\_USER\_FUNCTION || !EX(object)) ? fbc->common.scope : NULL;

 EG(called\_scope) = EX(call)->called\_scope;

}

Теперь, когда метод вызван, если в его теле вы используете $this для действия с переменной или вызова метода (например, $this->a = 8), то это приведёт к OPCode ZEND\_ASSIGN\_OBJ, который, в свою очередь, обратно извлечёт $this из EG(This).

static zend\_always\_inline zval \*\*\_get\_obj\_zval\_ptr\_ptr\_unused(TSRMLS\_D)

{

 if (EXPECTED(EG(This) != NULL)) {

 return &EG(This);

 } else {

 zend\_error\_noreturn(E\_ERROR, "Using $this when not in object context");

 return NULL;

 }

}

В том случае, если вы использовали $this для вызова метода (например, $this->foo()) или передали другому вызову функции ($this->foo($this);), то движок попытается извлечь $this из текущей символьной таблицы, как он это делает для каждой стандартной переменной. Но здесь осуществляется специальная подготовка в ходе создания кадра стека текущей функции:

if (op\_array->this\_var != -1 && EG(This)) {

 Z\_ADDREF\_P(EG(This));

 if (!EG(active\_symbol\_table)) {

 EX\_CV(op\_array->this\_var) = (zval \*\*) EX\_CV\_NUM(execute\_data, op\_array->last\_var + op\_array->this\_var);

 \*EX\_CV(op\_array->this\_var) = EG(This);

 } else {

 if (zend\_hash\_add(EG(active\_symbol\_table), "this", sizeof("this"), &EG(This), sizeof(zval \*), (void \*\*) EX\_CV\_NUM(execute\_data, op\_array->this\_var))==FAILURE) {

 Z\_DELREF\_P(EG(This));

 }

 }

}

Когда мы вызываем метод, движок изменяет область видимости:

if (fbc->type == ZEND\_USER\_FUNCTION || fbc->common.scope) {

 */\* ... ... \*/*

 EG(scope) = (fbc->type == ZEND\_USER\_FUNCTION || !EX(object)) ? fbc->common.scope : NULL;

 */\* ... ... \*/*

}

EG(scope) относится к типу zend\_class\_entry. Это класс, которому принадлежит запрашиваемый вами метод. И он будет использоваться для любой операции с объектом, которую вы будете выполнять в теле метода после проверки видимости движком:

static zend\_always\_inline int zend\_verify\_property\_access(zend\_property\_info \*property\_info, zend\_class\_entry \*ce TSRMLS\_DC)

{

 switch (property\_info->flags & ZEND\_ACC\_PPP\_MASK) {

 case ZEND\_ACC\_PUBLIC:

 return 1;

 case ZEND\_ACC\_PROTECTED:

 return zend\_check\_protected(property\_info->ce, EG(scope));

 case ZEND\_ACC\_PRIVATE:

 if ((ce==EG(scope) || property\_info->ce == EG(scope)) && EG(scope)) {

 return 1;

 } else {

 return 0;

 }

 break;

 }

 return 0;

}

Вот так можно получать доступ к приватным членам объектов, не принадлежащим вам, но являющимся дочерними по отношению к вашей текущей области видимости:

class A

{

 private $a;

 public function foo(A $obj)

 {

 $this->a = 'foo';

 $obj->a = 'bar'; */\* yes, this is possible \*/*

 }

}

$a = new A;

$b = new A;

$a->foo($b);

Эта особенность стала причиной большого количества баг-репортов от разработчиков. Но так устроена объектная модель в PHP — на самом деле, мы задаём область видимости на основе не объекта, а класса. В случае с нашим классом “Foo”, вы можете работать с любым приватным Foo любого другого Foo, как показано выше.

О деструкторе

Деструкторы опасны, не полагайтесь на них, поскольку PHP их не вызывает даже в случае фатальной ошибки:

class Foo { public function \_\_destruct() { echo "byebye foo"; } }

$f = new Foo;

thisfunctiondoesntexist();

*/\* fatal error, function not found, the Foo's destructor is NOT run \*/*

А что насчёт порядка вызова деструкторов в том случае, если они всё-таки вызываются? Ответ хорошо виден в коде:

void shutdown\_destructors(TSRMLS\_D)

{

 zend\_try {

 int symbols;

 do {

 symbols = zend\_hash\_num\_elements(&EG(symbol\_table));

 zend\_hash\_reverse\_apply(&EG(symbol\_table), (apply\_func\_t) zval\_call\_destructor TSRMLS\_CC);

 } while (symbols != zend\_hash\_num\_elements(&EG(symbol\_table)));

 zend\_objects\_store\_call\_destructors(&EG(objects\_store) TSRMLS\_CC);

 } zend\_catch {

 */\* if we couldn't destruct cleanly, mark all objects as destructed anyway \*/*

 zend\_objects\_store\_mark\_destructed(&EG(objects\_store) TSRMLS\_CC);

 } zend\_end\_try();

}

static int zval\_call\_destructor(zval \*\*zv TSRMLS\_DC)

{

 if (Z\_TYPE\_PP(zv) == IS\_OBJECT && Z\_REFCOUNT\_PP(zv) == 1) {

 return ZEND\_HASH\_APPLY\_REMOVE;

 } else {

 return ZEND\_HASH\_APPLY\_KEEP;

 }

}

Здесь продемонстрированы три стадии вызова деструктора:

* Цикличный просмотр глобальной символьной таблицы в обратном направлении и вызов деструкторов для объектов, у которых refcount = 1.
* Затем направление цикличности меняется, а деструкторы вызываются уже для всех остальных объектов, с refcount > 1.
* Если на одном из предыдущих этапов возникает проблема, то вызов оставшихся деструкторов прерывается.

К чему это приводит:

class Foo { public function \_\_destruct() { var\_dump("destroyed Foo"); } }
class Bar { public function \_\_destruct() { var\_dump("destroyed Bar"); } }

Пример первый:

$a = new Foo;

$b = new Bar;

"destroyed Bar"

"destroyed Foo"

Тот же пример:

$a = new Bar;

$b = new Foo;

"destroyed Foo"

"destroyed Bar"

Пример второй:

$a = new Bar;

$b = new Foo;

$c = $b; */\* increment $b's object refcount \*/*

"destroyed Bar"

"destroyed Foo"

Пример третий:

class Foo { public function \_\_destruct() { var\_dump("destroyed Foo"); die();} } */\* notice the die() here \*/*

class Bar { public function \_\_destruct() { var\_dump("destroyed Bar"); } }

$a = new Foo;

$a2 = $a;

$b = new Bar;

$b2 = $b;

destroyed Foo

Эта процедура была выбрана не просто так. Но если она вас не устраивает, то лучше уничтожайте свои объекты самостоятельно. Это единственный способ контролировать вызовы \_\_destruct(). Если вы предоставите PHP делать это за вас, то не возмущайтесь потом результатами его работы. У вас всегда есть возможность уничтожать свои объекты вручную, чтобы полностью контролировать очерёдность.

PHP не вызывает деструкторы в случае возникновения какой-либо фатальной ошибки. Дело в том, что в этом случае Zend работает нестабильно, а вызов деструкторов приводит к выполнению пользовательского кода, который может получить доступ к ошибочным указателям и, в результате, к падению PHP. Уж лучше сохранять стабильность системы — поэтому вызов деструкторов и блокируется. Возможно, в PHP 7 что-то и поменяется.

Что касается рекурсий, то в PHP они слабо защищены, да и относится это только к \_\_get() и \_\_set(). Если вы уничтожаете свой объект где-то в стековом кадре деструктора, то окажетесь в бесконечном рекурсивном цикле, который сожрёт все ресурсы вашего стека процесса (обычно 8 кбайт, ulimit –s) и сломает PHP.

class Foo

{

 public function \_\_destruct() { new Foo; } */\* you will crash \*/*

}

Суммируя вышесказанное: не доверяйте деструкторам критически важный код, например, управление механизмом блокировки (lock mechanism), поскольку PHP может и не вызвать деструктор или вызвать его в неконтролируемой последовательности. Если всё-таки важный код обрабатывается деструктором, то как минимум самостоятельно контролируйте жизненный цикл объектов. PHP вызовет деструктор, когда refcount вашего объекта упадёт до нуля, а это значит, что объект больше не используется и его можно безопасно уничтожить.

Заключение

Надеюсь, теперь вам многое стало понятнее в повседневной работе с объектами. Они не потребляют много памяти, а их реализация на уровне движка хорошо оптимизирована. Старайтесь использовать грамотно спроектированный автозагрузчик для улучшения использования памяти. Объявляйте классы в порядке логического наследования, и если превратите самые сложные из них в С-расширения, то сможете оптимизировать многие процессы, и даже ещё больше увеличить общую производительность подобных классов.

# Объектно-ориентированный PHP: специально для начинающих

В этом уроке вы ознакомитесь с основами объектно-ориентированного программирования в PHP. Вы узнаете о принципах ООП вообще и научитесь писать простенькие скрипты на PHP.

Добро пожаловать в первый из серии уроков по ООП на PHP! Ознакомившись со всеми уроками данной серии, вы узнаете об основных принципах и концепциях ООП и научитесь быстро и легко создавать полезные приложения на PHP.

В этом уроке я начну вводить вас в курс дела и расскажу вам об основных понятиях ООП. Вы узнаете:

* что такое ООП
* как ООП поможет вам создавать лучшие PHP скрипты
* некоторые основные понятия, такие как классы, объекты, методы, переменные класса
* с чего начать написание PHP скрипта

Вы готовы погрузиться в мир объектов PHP? Тогда вперед!

## Что такое объектно-ориентированное программирование?

Если вы когда-то создавали собственные функции в PHP и использовали их, то вы использовали такой стиль программирования, как процедурный. В процедурном программировании вы обычно создаете структуры данных - числа, строки, массивы и т.д. - для хранения каких-то данных, а затем обрабатываете эти структуры специальными функциями, которые манипулируют этими данными.

Объектно-ориентированное программирование, или ООП, пошло вперед, так как здесь мы храним структуры данных и функции, их обрабатывающие, в одной сущности, называемой объектом. Вместо того, чтобы обработать данные какой-либо функцией, вы загружаете эти данные в объект, а затем вызываете его методы для манипулирования ими и получаете желаемый результат.

Чаще всего объекты, создаваемые с помощью ООП, отражают реальные сущности. Например, если вы создаете форум для своего сайта, вам следовало бы создать объект Member, который будет хранить информацию о каждом участнике форума (имя, логин, электронный адрес, пароль и др.), а также методы, которые будут обрабатывать эту информацию (регистрация, авторизация, выход из системы, бан и т.д.).

## Зачем использовать ООП?

Процедурный и объектно-ориентированный - это два разных способа сделать одно и то же. Нельзя сказать, что один из них лучше другого - каждый пишет, как ему нравится, так что вы даже можете легко комбинировать эти два подхода в одном скрипте.

Однако, вот некоторые преимущества ООП для разработчиков:

* Легче отражать реальные ситуации: как я отметил выше, объекты отражают реальные сущности - люди, товары, карточки, статьи в блогах и др. Это во многом упрощает задачу, когда вы только начинаете проектировать свое приложение, так как назначение каждого объекта, как и цель отношений между объектами, будет ясно и понятно.
* Легче писать модульные программы: ООП предполагает написание модулей. Разделяя код на модули, вам будет легче им управлять, дебажить и расширять его.
* Легче писать код, который будет использоваться много раз: написание кода, который можно будет использовать не один раз, сэкономит время при написании приложения, и со временем вы даже можете создать целую библиотеку такого рода модулей, которые вы сможете использовать во многих приложениях. С помощью ООП становится сравнительно легче писать такой код, так как структуры данных и функции инкапсулируются в единственный объект, который можно использовать любое количество раз.

## Некоторые основные понятия



Перед тем, как начать писать скрипты, необходимо хорошо разобраться с такими понятиями, как класс, объект, переменная класса и метод.

### Классы

Класс - это каркас для объекта. Это кусок кода, который определяет:

* Типы данных, которые будут содержать созданные объекты класса
* Функции, которые будут содержать эти объекты.

Когда вы создаете приложение на ООП, вы обычно будете создавать несколько классов, которые будут представлять различные типы сущностей вашего приложения. Например, для создания форума вы можете создать классы Forum, Topic, Post и Member.

### Объекты

Объект - это переменная специального типа, которая создается через класс. Он содержит действительные данные и функции для манипулирования ими. Вы можете создавать сколько угодно объектов от одного единственного класса. Каждая функция объекта не зависит от другого объекта, даже если они созданы от одного и того же класса.

Для сравнения с реальными сущностями:

* Класс - это каркас для автомобиля: он определяет, как автомобиль будет выглядеть и действовать, но это все же абстрактная сущность
* Объект - это настоящий автомобиль, созданный из каркаса: у него есть настоящие свойства (например, скорость) и поведение (например, ускорение или торможение).

На заметку: Объект часто называют сущностью класса, а процесс создания объекта класса - реализацией.

### Переменные класса

Значения данных, которые хранятся в том или ином объекте, записываются в специальные переменные, называемые переменными класса. Переменные класса тесно связаны с его объектом. Несмотря на то что все объекты класса имеют одни и те же переменные, их значения могут отличаться.

### Методы

Функции, определяемые в классе и применяемые для объектов этого класса, называются методами. Они не во многом отличаются от обычных функций - вы можете передавать им значения, они могут содержать локальные переменные и возвращать значения. Однако, методы чаще работают с переменными объекта. К примеру, метод login() для авторизации пользователей в вашем форуме может устанавливать значение переменной класса loggedIn в true.

## Как создать класс в PHP?

Теперь, когда вы уже знаете, что такое классы, методы, переменные класса и объекты, пришло время создать пару классов и объектов в коде PHP.

Для начала посмотрим, как собственно нужно создавать класс. В принципе, скрипт по созданию класса выглядит так:

|  |  |
| --- | --- |
| 1 | class ClassName |
| 2 | { |

|  |  |
| --- | --- |
| 3 |   // (определение класса) |
| 4 | } |

К примеру, если вы создаете класс Member для вашего форума, вы напишите так:

|  |  |
| --- | --- |
| 1 | class Member |
| 2 | { |

|  |  |
| --- | --- |
| 3 |   // (определение класса) |
| 4 | } |

Это достаточно просто. Естественно, класс этот ничего не сделает, пока вы не добавите в него переменные и методы. Тем не менее, приведенный выше код создает валидный класс на PHP, который можно использовать.

Правило хорошего тона: каждый класс помещайте в отдельный файл с названием, совпадающим с именем класса. Например, поместите класс Member в файл Member.php и храните его в папке, допустим, classes.

## Как создавать объекты в PHP?

Создать объект можно с помощью ключевого слова new:

|  |  |
| --- | --- |
| 1 | new ClassName() |

Этот код создаст объект класса ClassName. Вам впоследствии понадобится использовать этот объект, поэтому его нужно хранить в переменной. Например, создадим объект класса Member и сохраним его в переменной $member:

|  |  |
| --- | --- |
| 1 | $member = new Member(); |

Мы также можем создать еще один объект того же класса:

|  |  |
| --- | --- |
| 1 | $member2 = new Member(); |

Несмотря на то что мы создали эти два объекта от одного и того же класса, переменные $member и $member2 не зависят друг от друга.

## Создаем переменные класса



Теперь, когда мы уже умеем создавать классы и объекты классов, давайте посмотрим, как создавать переменные класса. Есть 3 идентификатора доступа для переменных класса, которые можно добавлять в класс:

* Открытые переменные класса (public): доступны - т.е. их можно прочитать и/или изменять - в любом месте скрипта, независимо от того, где находится этот код - внутри класса или за его пределами
* Частные переменные класса (private): доступны только методам класса. Лучше всего делать переменные класса именно частными, чтобы отделить объекты от остальной части кода.
* Защищенные переменные класса (protected): доступны методам собственного класса, а также методам наследуемых классов (мы поговорим о наследовании позже).

Чтобы создать переменную класса, напишите ключевое слово public, private или protected, а затем введите имя переменной:

|  |  |
| --- | --- |
| 1 | class ClassName |
| 2 | { |

|  |  |
| --- | --- |
| 3 |   public $propertyName; |
| 4 |   private $propertyName; |

|  |  |
| --- | --- |
| 5 |   protected $propertyName; |
| 6 | } |

Давайте добавим переменную класса public нашему классу Member для хранения имени пользователя:

|  |  |
| --- | --- |
| 1 | class Member |
| 2 | { |

|  |  |
| --- | --- |
| 3 |   public $username = ""; |
| 4 | } |

Обратите внимание на то, что мы инициализировали нашу переменную класса, его значение - пустая строка, “”. Это значит, что при создании нового пользователя значение его имени по умолчанию будет равняться пустой строке. Так же, как и в случае с обычными переменными в PHP, переменные класса не обязательно инициализировать, но лучше все-таки не лениться. Если вы не инициализируете переменную класса, то по умолчанию ее значение равно null.

## Доступ к переменным класса

Для получения доступа к переменной того или иного объекта используется оператор ->:

|  |  |
| --- | --- |
| 1 | $object->propertyName |

Давайте попробуем. Напишем скрипт, который объявляет класс Member и переменную класса, создает объект этого класса, а затем задает значение переменной класса и выводит его на экран:

|  |  |
| --- | --- |
| 01 | <?php |
| 02 |    |

|  |  |
| --- | --- |
| 03 | class Member |
| 04 | { |

|  |  |
| --- | --- |
| 05 |   public $username = ""; |
| 06 | } |

|  |  |
| --- | --- |
| 07 |    |
| 08 | $member = new Member(); |

|  |  |
| --- | --- |
| 09 | $member->username = "Fred"; |
| 10 | echo $member->username;  // Выведет "Fred" |

|  |  |
| --- | --- |
| 11 |    |
| 12 | ?> |

Запустите данный код, он выведет на экран строку “Fred”, значение переменной класса $member->username. Как видите, вы оперируете переменной объекта так же, как обычной переменной - вы можете задать ей значение и прочитать его.

## Добавление методов в класс



Что скажете насчет создания методов? Как я ранее упоминал, методы - это обычные функции, являющиеся частью класса. Так что вы, возможно, не удивитесь тому, что и создаются они с помощью того же ключевого слова function. Единственное отличие от создания обычных функций заключается в том, что вы также можете добавить один из идентификаторов доступа (public, private, protected) в ее объявлении. В этом методы схожи с переменными класса:

|  |  |
| --- | --- |
| 01 | class ClassName |
| 02 | { |

|  |  |
| --- | --- |
| 03 |   public function methodName() { |
| 04 |     // (код) |

|  |  |
| --- | --- |
| 05 |   } |
| 06 |   |

|  |  |
| --- | --- |
| 07 |   private function methodName() { |
| 08 |     // (код) |

|  |  |
| --- | --- |
| 09 |   } |
| 10 |   |

|  |  |
| --- | --- |
| 11 |   protected function methodName() { |
| 12 |     // (код) |

|  |  |
| --- | --- |
| 13 |   } |
| 14 | } |

На заметку: так же, как и в случае с переменными класса, методы public могут быть вызваны откуда угодно, методы private могут вызываться только в пределах класса, а методы protected - из самого класса и его наследника.

Давайте попробуем добавить в наш класс некоторые методы и переменные класса:

* переменная класса private $loggedIn для идентификации пользователя, т.е. зашел он или нет,
* метод login(), который будет осуществлять вход на форум, устанавливая значение переменной класса $loggedIn в true,
* метод logout(), который будет осуществлять выход из форума, устанавливая значение переменной класса $loggedIn в false,
* метод isLoggedIn(), который будет возвращать значение переменной класса $loggedIn.

Вот наш код:

|  |  |
| --- | --- |
| 01 | <?php |
| 02 |    |

|  |  |
| --- | --- |
| 03 | class Member |
| 04 | { |

|  |  |
| --- | --- |
| 05 |   public $username = ""; |
| 06 |   private $loggedIn = false; |

|  |  |
| --- | --- |
| 07 |    |
| 08 |   public function login() { |

|  |  |
| --- | --- |
| 09 |     $this->loggedIn = true; |
| 10 |   } |

|  |  |
| --- | --- |
| 11 |    |
| 12 |   public function logout() { |

|  |  |
| --- | --- |
| 13 |     $this->loggedIn = false; |
| 14 |   } |

|  |  |
| --- | --- |
| 15 |    |
| 16 |   public function isLoggedIn() { |

|  |  |
| --- | --- |
| 17 |     return $this->loggedIn; |
| 18 |   } |

|  |  |
| --- | --- |
| 19 | } |
| 20 |    |

|  |  |
| --- | --- |
| 21 | ?> |

Вы наверное заметили, что мы использовали новое ключевое слово $this. В контексте методов объекта специальная переменная $this ссылается на сам объект. Используя $this в методе объекта, метод может получить доступ к любой переменной класса и методу объекта.

Например, метод login() может получить доступ к переменной класса $loggedIn объекта через $this->loggedIn.

Кстати, наша переменная класса - частная (private), поэтому ее нельзя вызывать из любой части скрипта, а только из методов login(), logout() и isLoggedIn(). Это хороший подход, так как внутренняя часть объекта (например, то, как именно записывается, авторизовался ли пользователь или нет) находится отдельно от остального кода. По возможности старайтесь использовать именно переменные класса private, чтобы ваши объекты были автономными, мобильными и защищенными.

На заметку: переменная класса $username в нашем примере - public. Я это сделал только для того, чтобы продемонстрировать, как можно получать доступ к переменным класса объекта. В реальных проектах скорее нужно сделать эту переменную частной и создать специальные переменные класса public для задания значений имени пользователя, если это необходимо.

## Использование методов

Чтобы вызвать метод объекта, воспользуйтесь оператором ->, с которым вы уже успели подружиться.

|  |  |
| --- | --- |
| 1 | $object->methodName() |

Это работает как и вызов обычной функции. Вы можете передать аргументы в скобках (если конечно он принимает какие-то аргументы), вызов метода также может возвращать определенные значения, которые вы затем можете использовать.

Мы добавили некоторые методы в наш класс, теперь давайте попробуем их применить для чего-то:

|  |  |
| --- | --- |
| 01 | <?php |
| 02 |    |

|  |  |
| --- | --- |
| 03 | class Member |
| 04 | { |

|  |  |
| --- | --- |
| 05 |   public $username = ""; |
| 06 |   private $loggedIn = false; |

|  |  |
| --- | --- |
| 07 |    |
| 08 |   public function login() { |

|  |  |
| --- | --- |
| 09 |     $this->loggedIn = true; |
| 10 |   } |

|  |  |
| --- | --- |
| 11 |    |
| 12 |   public function logout() { |

|  |  |
| --- | --- |
| 13 |     $this->loggedIn = false; |
| 14 |   } |

|  |  |
| --- | --- |
| 15 |    |
| 16 |   public function isLoggedIn() { |

|  |  |
| --- | --- |
| 17 |     return $this->loggedIn; |
| 18 |   } |

|  |  |
| --- | --- |
| 19 | } |
| 20 |    |

|  |  |
| --- | --- |
| 21 | $member = new Member(); |
| 22 | $member->username = "Fred"; |

|  |  |
| --- | --- |
| 23 | echo $member->username . " is " . ( $member->isLoggedIn() ? "logged in" : "logged out" ) . "<br>"; |
| 24 | $member->login(); |

|  |  |
| --- | --- |
| 25 | echo $member->username . " is " . ( $member->isLoggedIn() ? "logged in" : "logged out" ) . "<br>"; |
| 26 | $member->logout(); |

|  |  |
| --- | --- |
| 27 | echo $member->username . " is " . ( $member->isLoggedIn() ? "logged in" : "logged out" ) . "<br>"; |
| 28 |    |

|  |  |
| --- | --- |
| 29 | ?> |

Данный скрипт отобразит следующее:

|  |  |
| --- | --- |
| 1 | Fred is logged out |
| 2 | Fred is logged in |

|  |  |
| --- | --- |
| 3 | Fred is logged out |

Вот, как он работает:

1. После описания класса Member мы создали его объект и сохранили в переменной $member. Также мы дали переменной класса $username данного объекта значение “Fred”.
2. Затем мы вызвали метод $member->isLoggedIn() для того, чтобы определить, залогинился ли пользователь или нет. Данный метод просто-напросто возвращает значение переменной класса $loggedIn. Так как значение по умолчанию этой переменной класса - false, значит результатом вызова $member->isLoggedIn() будет ложь, поэтому отобразится сообщение "Fred is logged out".
3. Затем вызовем метод login(). Он установит в true значение переменной класса $loggedIn.
4. Теперь, при вызове метода $member->isLoggedIn() вернется истина, и выведется сообщение "Fred is logged in".
5. Вызовем метод logout(), который устанавливает в false значение свойства $loggedIn.
6. В третий раз вызовем метод $member->isLoggedIn(). Сейчас он вернет false, потому что значение свойства $loggedIn опять установлено в ложь. Так, снова выведется сообщение "Fred is logged out".

На заметку: на случай, если вы в первые увидели такое: ?:, - это тернарный оператор. Это упрощенная версия блоков if … else. Узнать о такого рода операторах можно [здесь](http://php.net/manual/en/language.operators.comparison.php).

## Выводы

В этом уроке вы познакомились с основами ООП в PHP. Вы узнали о таких вещах, как:

* что такое ООП и почему его полезно применять
* понятия классов, объектов, переменных класса и методов
* как создавать классы и объекты
* как создавать и использовать переменные классов
* понятия идентификаторов доступа public, private, protected
* как создавать и применять методы классов

Вы уже много о чем узнали и еще много чему научитесь в следующих уроках. Тем не менее, если вы отработали хорошенько все примеры, приведенные мной, у вас есть крепкая основа. Можете приступать к созданию приложений на ООП.

До новых встреч!

Данный урок подготовлен для вас командой сайта [ruseller.com](http://ruseller.com/)
Источник урока: [www.elated.com/articles/object-oriented-php-for-absolute-beginners/](https://ruseller.com/www.elated.com/articles/object-oriented-php-for-absolute-beginners/)
Перевел: Станислав Протасевич
Урок создан: 25 Июня 2011
Просмотров: 100003
[Правила перепечатки](https://ruseller.com/page.php?doc=reprint)