



# Готовимся к переходу на PHP 5.3

**Александр Майоров**

## Обзор основных нововведений, доступных в PHP 5.3.

PHP 5.2 существует уже почти 2 года. В нем найдено несколько серьезных ошибок, которые не могут быть исправлены без потери бинарной совместимости. В разрабатываемом PHP 6 из-за перехода на Unicode перестанет работать большое количество наработанного кода, также для него было разработано много интересных дополнений и улучшений, которые не могут быть совместимы со старыми работами. Для плавного перехода с PHP 5.2 на PHP 6 было решено разработать промежуточную версию PHP 5.3, которая будет содержать большинство улучшений, разработанных для PHP 6, но также способная выполнять существующие скрипты без каких-либо изменений.

Команда разработчиков PHP 1 августа 2008 года с гордостью объявила первый альфа-релиз новой версии PHP 5.3, содержащей значительные изменения по сравнению с предыдущими релизами. Бинарные пакеты для операционной системы Windows будут доступны начиная со 2-й альфа-версии. Промежуточные снапшоты для всех платформ уже сейчас до-

ступны на <http://snaps.php.net>. PHP 5.3, как ожидается, будет отличаться большей стабильностью и производительностью, а также содержать новые функциональные расширения и синтаксические конструкции.

Цель альфа-версии состоит в том, чтобы стимулировать php-программистов не только активно участвовать в выявлении ошибок, но и в обеспечении того, чтобы все новые возможности были протестированы на практике. Также разработчикам важно ваше мнение о новых возможностях языка, которыми вы будете в дальнейшем пользоваться. Вы можете все это сделать через список рассылки либо в багтрекере на [php.net](http://php.net).

В этой статье описаны основные нововведения, которые нас ожидают в PHP 5.3. Вот краткий обзор наиболее важных из них:

- пространства имен;
- позднее статическое связывание;
- перегрузка статических методов;
- лямбда-функции и замыкания;
- новое расширение PHAR;
- поддержка SQLite3;
- циклическая сборка мусора;

- поддержка драйвера MySQLnd;
- синтаксические изменения;
- улучшена производительность;
- расширение системы конфигурирования;
- новые возможности SPL.

## Пространства имен

Если вы являетесь постоянным читателем нашего журнала, то вы должны иметь представление о пространствах времён из статьи в № 12 за 2007 год. Так что вы можете использовать эту статью как подробную инструкцию по данной теме. Для тех, кто не имел возможности прочесть этот материал, я проведу краткий обзор и опишу дополнительные возможности, которые были добавлены в этой версии.

Пространство имен (namespace) — это область определения переменных, констант и т. п., ограничивающая их область видимости. Оно предназначено для локализации имен идентификаторов и предотвращения конфликтов между ними. Namespace устраняют конфликты имен. Разные namespace могут использовать одно и то же имя для разных целей. Имя внутри про-

странства имеет единственный смысл. Также они делают имена короче. Имена, определенные в namespace имеют короткое (локальное) имя и уникальное длинное (квалифицированное) для использования за пределами namespace.

Один namespace может определяться в нескольких файлах. В namespace могут определяться:

- классы;
- интерфейсы;
- функции;
- константы;
- PHP-код.

В namespace не могут определяться:

- глобальные переменные.

PHP не поддерживает вложенных namespace. PHP поддерживает составные имена namespace. Приведу простой пример работы. Создадим 3 файла: myspace\_part1.php, myspace\_part2.php, index.php.

В файле myspace\_part1.php напомним следующий тестовый код:

```
--- myspace_part2.php ---
<?php
namespace My::Test::Space;

const TEST_ITEM = __NAMESPACE__;

class TestClass
{
    public static function getTestItem()
    {
        return TEST_ITEM;
    }
}

?>

--- myspace_part1.php ---
<?php
namespace My::Test::Space;

require 'myspace_part2.php';

function var_dump()
{
    $args = func_get_args();
    return call_user_func_array('::var_dump', $args);
}

?>

--- index.php ---
<?php
require 'myspace_part1.php';

$Std = new stdClass();
$Std->field_1 = 123;
$Std->field_2 = array(1,2,3);

My::Test::Space::var_dump($Std);

use My::Test::Space::TestClass;
use My::Test::Space as MTS;

$obj = new TestClass;
MTS::var_dump( $obj, MTS::TestClass::getTestItem() );

?>
```

Пример самодостаточен, чтобы на практике показать всю мощь namespace. Еще скажу пару слов про оператор «use», с помощью которого осуществляется импорт. Оператор «use» может импортировать:

- namespace;
- классы;
- интерфейсы.

Он не может импортировать:

- функции;
- константы;
- переменные.

В момент импорта можно сделать переименование пространства имен для более удобного использования. Оператор «use» действует только на текущий файл. Данный оператор сам не подгружает никаких файлов. Очень удобно организовывать подключение библиотек через функцию автозагрузки классов.

```
<?php

function __autoload($libName)
{
    require_once ( str_replace('::', '/', $libName) .
        '.php' );
}

//require_once 'My/Test/Space/testClass.php';
use My::Test::Space::TestClass;

$obj = new TestClass;

?>
```

## Позднее статическое связывание (Late Static Binding)

Начиная с версии 5.3.0 PHP реализует возможность, называемую позднее статическое связывание, которая может быть использована для управления статическими методами при наследовании. Для начала приведу небольшой пример кода, на основе которого и разьясню суть проблемы. Есть 2 класса:

```
<?php

class TestClass_A
{
    public static function foo()
    {
        printf("\n %s \n", __CLASS__);
    }

    public static function bar()
    {
        self::foo();
    }
}

class TestClass_B extends TestClass_A
{
    public static function foo()
    {
        printf("\n %s \n", __CLASS__);
    }
}

TestClass_B::bar();

?>
```

В результате выполнения данного кода будет вызван метод TestClass\_A::foo(), хотя мы ожидали бы, что будет вызван метод класса TestClass\_B. Таким образом, не существовало инструмента для точного определения, какой статический метод должен быть вызван при наследовании. Для решения

данной проблемы был введен класс `static`, с помощью которого можно осуществить позднее статическое связывание. Тогда, переписав наш код следующим образом:

```
<?php
class TestClass_A
{
    public static function foo()
    {
        printf("\n %s \n", __CLASS__);
    }

    public static function bar()
    {
        // self::foo();
        static::foo();
    }
}

class TestClass_B extends TestClass_A
{
    public static function foo()
    {
        printf("\n %s \n", __CLASS__);
    }
}

TestClass_B::bar();

?>
```

мы получим то, что хотели, а именно вызов метода `TestClass_B::foo()`. Благодаря этому нововведению мы получили возможность написать абстрактный класс `Singleton`.

```
<?php
abstract class Singleton
{
    protected function __construct() {}
    protected function __clone() {}

    public function getInstance()
    {
        if ( is_null(static::$Instance) )
            static::$Instance = new static;
        return static::$Instance;
    }
}

class TestSingleton_A extends Singleton
{
    protected static $Instance = NULL;
    public $field = 123;
}

class TestSingleton_B extends Singleton
{
    protected static $Instance = NULL;
    public $field = 456;
}

var_dump( TestSingleton_B::getInstance()->field );
var_dump( TestSingleton_A::getInstance()->field );

?>
```

## Перегрузка статических методов

В PHP 5.3 появился «магический» метод `__callStatic()`, позволяющий перехватывать обращение к статическим методам класса.

```
<?php
class TestClass_A
{
    public static function __callStatic($name, $argv)
    {

```

```
        var_dump( __METHOD__, $name, $argv);
    }
}

TestClass_A:: foo('arguments');

?>
```

Данный «магический» метод по сути эквивалентен методу `_call()`, с той лишь разницей, что он обязательно должен быть объявлен как статический. Уверен, что применение данной функции найдется во многих областях и шаблонах проектирования. Здесь происходит вызов несуществующего статического метода `TestClass_A:: foo()`, обращение к которому мы перехватили с помощью магического метода `TestClass_A:: __callStatic()`. В этом методе мы можем сделать нужные нам обработки и вызвать нужные нам функции.

## Лямбда-функции и замыкания

Чем действительно порадовали нас разработчики PHP 5.3, так это поддержкой лямбда-функций, за что им особое большое спасибо! Лямбда-функция – это анонимная функция, создающаяся каждый раз при вызове в определенном контексте приложения. Данный тип функций создан в первую очередь для реализации таких алгоритмов, как «замыкания» и создания функций высшего порядка.

Функцией высшего порядка называется функция, принимающая другие функции в качестве параметров или возвращающая функциональные значения. В PHP была уже возможность создавать динамически анонимные функции с помощью функции `create_function()`. Но данная функция возвращает имя созданной функции в виде строки. Пример такой приведен ниже:

```
<?php
$lambda = create_function(" ", 'print __FUNCTION__ . "\n";');
$lambda();

call_user_func($lambda);

?>
```

В примере мы создаем динамическую функцию без имени. Точнее, мы не задаем ей имя, но PHP присвоит ей имя в стиле «`lambdaN`», где `N` – порядковый номер анонимной функции. К такой функции мы всегда можем обращаться только через переменную, которой мы присвоили результат выполнения `create_function()`. Данная функция будет занимать ресурсы точно так же, как и обычная пользовательская функция. Эта запись не является настоящей реализацией лямбда-функции. В PHP 5.3 добавили более менее нормальную поддержку лямбда-выражений, которая выглядит следующим образом:

```
<?php
$lambda = function () {
    { echo 'Test function: ' . __FUNCTION__ . "\n"; };
};

var_dump($lambda);

$lambda();

call_user_func($lambda);

?>
```

Во-первых, обратите внимание на обязательную точку с запятой, без которой будет сгенерирована синтаксическая ошибка. Запись функции теперь является выражением, которое обязательно должно закрываться знаком завершения выражения. Лямбда-выражение является объектом класса Closure. Также константа `__FUNCTION__` внутри контекста лямбда-функций не доступна. Поэтому в нашем примере вызов функции выведет всего лишь надпись «Test function: ». При использовании лямбда-выражений в классах есть небольшие особенности. Например есть класс:

```
<?php

class TestClass_A
{
    public $lambda_foo;

    function __construct()
    {
        $this->lambda_foo = function() { return __CLASS__; };
    }
}

$obj = new TestClass_A();
echo $obj->lambda_foo();

?>
```

Данный код приведет к ошибке, так как метода `TestClass_A::lambda_foo()` не существует. Что же делать? Для таких случаев у объекта типа Closure есть метод `__invoke()`, который вызывает функцию. Поэтому вызов придется осуществлять таким образом:

```
<?php

echo $obj->lambda_foo->__invoke();

?>
```

Может, не особо красиво, но что мешает нам сделать вызов лямбда-функций через перехват обращений с помощью метода `__call()`? Да, собственно, ничто.

```
<?php

class TestClass_A
{
    public $lambda_foo;

    function __construct()
    {
        $this->lambda_foo = function() { return __CLASS__; };
    }

    function __call($name, $argv)
    {
        if ( isset($this->$name) && $this->$name instanceof Closure )
            return $this->$name->__invoke($argv);
    }
}

$obj = new TestClass_A();
echo $obj->lambda_foo();

?>
```

Но вообще такое должно встречаться редко. Не стоит увлекаться и использовать лямбда-выражения везде, где попало. Раз мы заговорили о магическом методе `__invoke()`, который был добавлен, то стоит его разобрать

чуть более подробно, но после того как закончим с лямбда-выражениями.

Мы уже достаточно успели поговорить о лямбда-функциях и разобрать несколько примеров, но я до сих пор не показал полный синтаксис записи. Шаблон записи лямбда-выражения выглядит так:

```
function & (parameters) use (lexical vars) { body };
```

Вы обратили внимание на оператор `use`? Да, точно такой же оператор используется для импорта пространств имен. В данном контексте он используется для импорта внешних переменных, относительно лямбда-функции внутри выражения. Лучше всего показать на примере:

```
<?php

class TestClass_A
{
    function foo_method($x, $y)
    {
        $i = 8;
        $n = 32;

        $closure =
            static function ($x, $y) use ($i, $n)
            {
                return ($x+$i) * ($y+$n);
            };

        return $closure($x, $y);
    }
}

$obj = new TestClass_A();
echo $obj->foo_method(3, 2);

?>
```

В данном случае `$x` и `$y` передаются как аргументы функции, а вот `$i` и `$n` импортируются в нее. Импортированные переменные становятся доступными внутри выражения, как если бы они были частью лямбда-функции. Кстати, вышеприведенный пример реализует такой алгоритм высшего порядка, как замыкание, о которых мы поговорим далее. Также надо сказать про префикс `static`. Вы можете создавать статические лямбда-выражения, точно так же, как статические переменные внутри функций.

## Замыкания

Суть замыканий в том, что многие функции можно параметризовать целыми алгоритмами. И очень часто такие алгоритмы можно оформить в виде функций, которые передаются в качестве аргумента. Они, как правило, имеют небольшой размер и необходимы только для маленького участка кода, где они применяются. Отсюда возникает необходимость в синтаксисе, позволяющем описывать такие функции прямо по месту использования, не засоряя пространство имен. Следствием такой необходимости были придуманы лямбда-выражения, о которых мы уже говорили, и такой паттерн функционального программирования, как «замыкания».

Замыкание (англ. closure) – процедура, которая ссылается на свободные переменные в своём лексическом контексте. Замыкание, так же как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе. Также можно сказать, что за-

мыкание – это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. В записи это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции с новыми ссылками на переменные внешней функции. Замыкание связывает код функции с её лексическим окружением (местом, в котором она определена в коде). Лексические переменные замыкания отличаются от глобальных переменных тем, что они не занимают глобальное пространство имён. От переменных в объектах они отличаются тем, что привязаны к функциям, а не объектам.

```
<?php
$array = array(1, 2, 3, 4, 5, 6, 7, 8, 9);

$array = array_map( function($element) {
    return $element < 5 ? $element /10 :
        $element *10; } , $array );

var_dump($array);

?>
```

Согласитесь, что такая запись намного удобней, чем написание отдельной функции, особенно если учесть, что данный участок кода должен выполняться всего один раз на протяжении работы всей программы. Ради такого случая засорять программу реализацией отдельной функции нет смысла. К тому же после отработки данного кода ресурсы не будут засорены, так как никакой функции не будет висеть в памяти.

## Магический метод \_\_invoke()

Данный метод позволяет работать с объектом как с функцией. Сразу приведу пример:

```
<?php
class TestClass_A
{
    function __invoke($i)
    {
        return $i * 16;
    }
}

$obj = new TestClass_A();
echo $obj(8);

?>
```

Теперь в арсенале программистов ООП-щиков появился еще один новый инструмент для реализации новых интересных алгоритмов. Появилась реальная возможность создавать функции-объекты, а это расширяет объектно-ориентированные возможности языка.

## Поддержка драйвера MySQLnd

В PHP5.3 добавлен ряд расширений, одним из которых является MySQLnd. Хотя данное новшество не является PHP-расширением в полном смысле этого слова, так как не предоставляет каких-то новых функций, доступных для PHP-программиста. Оно является всего лишь заме-

ной библиотеки libmysql, оптимизированной специально для PHP. MySQLnd использует менеджер памяти Zend Engine, таким образом не требуется копирование данных из библиотеки libmysql в PHP. Оно использует значение директивы memory\_limit, что позволяет контролировать выделяемую память. MySQLnd использует потоки PHP, что делает более быстрыми постоянные соединения. Также данное расширение содержит серию оптимизаций для более быстрого получения результатов. Кроме того, MySQLnd позволяет собирать статистику, что может пригодиться в дальнейшем для оптимизации ваших запросов. MySQLnd используется как для работы mysql, так и для mysqli-библиотек.

Вопрос о том, какая библиотека будет использоваться libmysql или mysqlnd, решается на стадии компиляции PHP. Многими любимым PDO на данный момент не поддерживает работу через MySQLnd. Как заявил Дмитрий Стогов на седьмой международной конференции PHPCONF 2008, поддержку PDO реализовывать не будут, но не исключают, что могут найтись энтузиасты, которые смогут осуществить это и написать расширение. Может быть, этим человеком станете вы, уважаемый читатель.

## Новое расширение Phar

В PHP 5.3 появилось новое расширение, по принципу аналогичное концепции JAR-технологии Java. Phar-архив прежде всего придуман для приложений-инсталляторов, а также для удобного распространения целого PHP-приложения или библиотеки в одном файле. В отличие от JAR-архивов, Phar-архивы не нуждаются ни в специальной среде исполнения, ни во внешних средствах для запуска процесса или PHP. Phar-архив используется так же, как и обычные PHP-файлы. Например, чтобы вывести в браузер изображение из архива, достаточно написать:

```
<?php
header('Content-type: image/jpeg');
echo file_get_contents('phar:///path/to/my/test.phar/
    resources/image.jpg');

?>
```

Как видите, для подключения используется специальный стрим-враппер 'phar:'. Давайте теперь попробуем на практике создать Phar-архив. Вообще надо сказать, что специальных архиваторов нет. Но это не проблема. Создать такой архив несложно самостоятельно, но есть одно но. По умолчанию работа с Phar-архивами разрешена только на чтение. А чтобы создать архив, нужно разрешить запись. Для этого надо в php.ini добавить всего одну строчку:

```
[phar]
phar.readonly=0
```

Если этого не сделать, ваш скрипт будет все время генерить исключения. Теперь напомним небольшой код:

```
<?php
$DIR = __DIR__;

try {
    $Phar = new Phar('test.phar', 0);

    $Phar["/test.php"] =
    <<<'NOWDOC'
```

```
<?php

echo 'This is test phar archive.';

NOWDOC;

$fp = fopen('img.jpg', 'rb');
$Phar["/img.jpg"] = fread($fp, filesize('img.jpg'));
fclose($fp);

} catch (Exception $e) {
    echo $e->getMessage();
}

?>
```

Данный код демонстрирует, как в архив добавляется PHP-код из строки и один графический файл. После того как отработает скрипт, будет создан файл test.phar. Проверим, что внутри, делаем простой тест:

```
<?php

$DIR = __DIR__;

include "phar://$DIR/test.phar/test.php";

?>
```

и

```
<?php

$DIR = __DIR__;

header('Content-type: image/jpeg');
echo file_get_contents("phar://$DIR/test.phar/img.jpg");

?>
```

Как видите, ничего сложного. Мы создали исполняемый php-архив. Данный архив изначально сжат по алгоритму tar. Вы всегда можете сжать данные архивы с помощью ZIP. Добавьте в код следующую запись:

```
<?php

$Phar = $Phar->convertToExecutable(Phar::TAR, Phar::GZ);

?>
```

Если создать в архиве файл с именем index.php, то он будет запускаться автоматически. Достаточно, например, в консоли написать

```
php test.phar
```

либо сделать подключение файла через include:

```
<?php

$DIR = __DIR__;

include "phar://$DIR/test.phar";

?>
```

Более подробную информацию вы можете получить по адресу: <http://ru2.php.net/manual/ru/book.phar.php>.

## Синтаксические изменения

В самом синтаксисе языка также есть некоторые изменения. Все эти изменения можно отнести к термину «синтак-

сический сахар», но, тем не менее, они приятны. Ведь все мы любим сладкое. Итак, что нам на десерт приготовили разработчики PHP? Давайте посмотрим. Первое – это новый синтаксис старого оператора HEREDOC, который называется NOWDOC.

### NOWDOC

В PHP существует синтаксис HEREDOC, который называется «встроенный документ». Обычная форма записи такого оператора выглядит так:

```
<?php

$var_1 = 123;
$var_2 = 456;

echo <<<HEREDOC

Example:
$var_1+$var_2

HEREDOC;

// Example:
// prints 123+456

?>
```

Новый синтаксис NOWDOC показан в следующем примере:

```
<?php

echo <<<'NOWDOC'

Example:
$var_1+$var_2

NOWDOC;

// Example:
// $var_1+$var_2

echo <<<"NOWDOC"

Example:
$var_1+$var_2

NOWDOC;

// Example:
// printd 123+456

?>
```

Суть нового синтаксиса в том, что теперь можно контролировать поведение многострочных вставок. Если раньше они интерпретировались так же, как двойные кавычки, то теперь можно переключить поведение, чтобы данный блок интерпретировался как одинарные. Делается это добавлением одинарных кавычек к открывающему маркеру блока: <<<'NOWDOC'. Без указания кавычек данный блок будет вести себя так же, как и раньше. Для совместимости была добавлена возможность указывать поведение синтаксиса с помощью двойных кавычек, хотя это не обязательно.

### Оператор безусловного перехода GOTO

Данный оператор реализован для поддержки программно-генерируемого кода. Но так уж повелось, что присутствие или отсутствие оператора GOTO в языке программирова-

ния всегда вызывает жаркие дебаты среди сторонников «правильного стиля» программирования. Всегда считалось признаком «хорошего тона» неиспользование данного оператора. Я не являюсь сторонником одной из этих «коалиций», хотя считаю оператор GOTO неприемлемым в серьезном программировании. Тем не менее хочу сказать, что действительно есть места, где использование оператора goto выглядит вполне логично.

Начинающие программисты, пришедшие в PHP из других языков, постоянно задаются одними и теми же вопросами. Например, такими: как выйти из многовложенного цикла? Средствами структурного программирования такой выход можно достигнуть, используя оператор break(n), но он не всегда подходит для поставленных целей. Чаще такие алгоритмы реализуются, добавляя флаги в циклы, которые проверяются при каждой итерации, что очень громоздко и непроизводительно. Процессоры не любят ветвлений, и каждая лишняя проверка съедает очень много тактов, особенно на нерегулярных переходах, которые невозможно предсказать.

Пример такого цикла может быть следующим:

```
<?php

$exit = false;

for ($i=0; $i<100; $i++)
{
    // ...

    for ($n=0; $n<100; $n++)
    {
        // ...

        for ($x=0; $x<100; $x++)
        {
            // ...

            if ( $x == 8 && 4 == $n && $i == 2 )
            {
                $exit = true;
                break;
            }

            // ...
        }

        if ( $exit ) break;

        // ...
    }

    if ( $exit ) break;

    // ...
}

echo "Exit from loop: $i, $n, $x";

?>
```

То есть, вместо того чтобы использовать GOTO и выйти из циклов сразу, пришлось завести еще одну переменную \$exit и добавить проверки условий. Оператор GOTO в такой ситуации выглядит намного лучше. Это простой пример, но в реальности он может содержать очень много кода, и циклов может быть не три, а много больше. Кроме того, введение еще одной переменной дает возможность где-нибудь допустить ошибку, например, в ее инициализации или при проверке. Опять же, программисту придется лазить

по тексту и смотреть, зачем была нужна эта переменная.

Красивое быстрое решение для данного примера состоит в использовании горячо критикуемого GOTO, который обвиняют в неструктурности и в «идеологической неправильности». Наш пример с использованием данного оператора будет выглядеть следующим образом:

```
<?php

for ($i=0; $i<100; $i++)
{
    // ...

    for ($n=0; $n<100; $n++)
    {
        // ...

        for ($x=0; $x<100; $x++)
        {
            // ...

            if ( $x == 8 && 4 == $n && $i == 2 )
                goto L_exit;

            // ...
        }

        // ...
    }

    // ...
}

L_exit:
echo "Exit from loop: $i, $n, $x";

?>
```

Как видите, данный код намного лучше предыдущего. Оператор GOTO имеет полное право на существование, потому что существуют ситуации, в которых он может значительно улучшить программный код. Но надо помнить, что таких случаев должно быть немного и программа должна проектироваться так, чтобы не возникало необходимости в данном операторе. Хотя оператор goto и может облегчить задачу, при злоупотреблении им программа превращается в спагетти, и ее становится совершенно невозможно нормально отлаживать, так как непонятно, как мы попали в этот блок кода и откуда был совершен переход сюда.

Данный оператор предназначен в первую очередь для реализации таких алгоритмов, как «конечные автоматы». В PHP у него есть также ограничение, в отличие от других языков.

Переход с помощью GOTO запрещен внутрь цикла или оператора switch.

### Сокращенный тернарный оператор

Еще одним синтаксическим сахаром нас «порадовали» разработчики. Это сокращенная запись тернарного оператора «?:». Честно говоря, не совсем вижу очевидные преимущества данной записи, хотя данной фишке можно будет придумать применение. Например, такое:

```
<?php

define('IF_DEBUG', false);

// ...
```

```
function print_debug_info()
{
    print "Deug ...";
}

// ...

IF_DEBUG ?: print_debug_info();

?>
```

Правда, здесь теряется очевидность логики, так как, чтобы отработала функция `print_debug_info()`, константа `IF_DEBUG` должна принимать значение `FALSE`. Сокращенная запись по сути эквивалентна следующему выражению: «`$a ? : $b === $a ? $a : $b`». Надо сказать сразу про небольшое отличие от полной записи оператора. Следующий код показывает различия в поведении:

```
<?php

$condition = 0;

$result_1 =
$result_2 = 'True';

$result_1 = $condition ? : 'False';
$result_2 = $condition ? $result_2 : 'False';

var_dump($result_1, $result_2);

?>
```

Результат данного кода вроде бы очевиден. Обе переменные будут содержать строку `False`. Далее мы переключаем наш флаг в 1 и смотрим снова результат кода. На этот раз переменная `$result_1` будет равна 1, `$result_2` будет равна `True`. Как видите, в качестве положительного результата сокращенный оператор возвращает результат условия. Кстати, также не вижу особой полезности данного оператора, так как сокращенный вариант условий можно было писать, используя логические операторы. Мы можем написать так:

```
<?php

$a = 1;
$r2 = ,True`;

$a || $r2 = ,False`;

var_dump($r1, $r2);

?>
```

Правда, эта запись уже подходит под определение не тривиального синтаксиса (о котором было подробно написано в №10 за 2007 год). Тем не менее, сокращенная запись тернарного оператора может быть востребована в некоторых случаях.

## Константа `__DIR__`

Частенько приходится определять константу с именем текущей директории. Как правило, это делается следующим образом:

```
<?php

define('__DIR__', dirname(__FILE__));
```

```
printf("%s \n", __DIR__);

?>
```

Теперь в этом нет необходимости, так как разработчиками PHP была добавлена константа `__DIR__`. По сути дела данная константа эквивалентна записи «`__DIR__ == dirname(__FILE__)`», однако она имеет ряд преимуществ. Одно из таких преимуществ данной константы заключается в том, что она вычисляется движком Zend Engine на стадии компиляции скрипта и соответственно потребляет меньше ресурсов, чем вышеописанный вариант. Во-вторых, она вычисляется для каждого скрипта, в котором была запрошена, в отличие от нашего варианта.

## Сборщик мусора и производительность

В PHP 5.3 появился сборщик мусора, который распознает и уничтожает циклические структуры. Хотя сборщик мусора был в PHP практически с самого начала, эффективность его работы оставляла желать лучшего. Он был построен на счетчике ссылок и не распознавал циклических структур. Типичным примером такой структуры является массив, один из элементов которого является ссылкой на самого себя:

```
<?php

$a = array();
$a[0] = &$a;
unset ($a);

?>
```

Даже когда мы делали уничтожение переменной, значение счетчика ссылок не уменьшалось и не доходило до нуля. Это соответственно приводило к тому, что переменная оставалась висеть в памяти до окончания работы скрипта, занимая системные ресурсы. Сборщик мусора может запускаться как автоматически, так и явно через вызов функции `gc_collect_cycles()`. Данная функция возвращает количество уничтоженных циклических структур. Автоматический запуск сборщика мусора можно отключать функцией `gc_disable()` и включать функцией `gc_enable()` соответственно.

```
<?php

gc_disable();

$a = array();
$a[0] = &$a;
unset ($a);

printf("%d \n", gc_collect_cycles());

gc_enable();

?>
```

Также выросла производительность интерпретатора по сравнению с предыдущими версиями.

## Расширение системы конфигурирования

В новой версии был изменен механизм обработки файла настроек. Теперь есть возможность задавать разные установки для разных директорий.

Например, у нас на 1-м сервере располагаются сразу девелоп-хост для разработки программного обеспечения и продакшн-хост, где выкладывается конечная версия продукта для пользователей. На девелоп-версии мы включаем вывод всех сообщений об ошибках, а на продакшн версии, естественно, отключаем вывод сообщений о каких-либо ошибках. Обычно это делается добавлением установки директивы `error_reporting` в скриптах или через вызов одноименной функции. К примеру, часто делают так: в `php.ini` директиву `error_reporting` выставляют в ноль, а в девелоп-версии скриптов в самом начале пишут «`error_reporting( E_ALL | E_STRICT )`». Не так уж это и сложно, но ведь это простой пример, а на практике может быть намного больше различных настроек, и не все они могут быть выставлены в пользовательских скриптах. Теперь появилась возможность указывать настройки для каждой директории в отдельности. Просто в `php.ini` пишем:

```
[PATH=/home/ develop_host/www]
error_reporting = E_ALL | E_STRICT

[PATH=/home/production_host/www]
error_reporting = 0
```

Все. При обращении к скриптам в той или иной директории PHP будет применять соответствующие настройки. Также есть возможность задавать разные установки для разных виртуальных хостов. Мы можем предыдущий пример записать как:

```
[PATH=develop.my-super-startup.ru]
error_reporting = E_ALL | E_STRICT

[PATH= my-super-startup.ru]
error_reporting = 0
```

В итоге мы привязали настройки к виртуальным хостам, а не к директориям, что позволит нам переносить проекты из директории в директорию, но при этом не менять настройки в `php.ini`.

Более подробно о данном нововведении можно прочитать на <http://ru.php.net/ini.sections>.

## Новые возможности SPL

SPL – стандартная библиотека PHP (Standard PHP Library). Представляет собой набор интерфейсов и классов, которые предназначены для решения стандартных задач и реализуют функционал для эффективного доступа к данным, интерфейсам и классам. Эта библиотека позволяет не отвлекаться на рутину по изобретению «велосипедов», так как многие алгоритмы в программировании, можно сказать, стандартизированы. SPL предоставляет программисту доступ к эффективной реализации наиболее распространенных алгоритмов и структур данных. В SPL у PHP 5.3 появился новый функционал и новые структуры данных, такие как `heap`, `stack`, `queue` и `linked list`. Много рассказывать про данные структуры не буду, так как даже начинающим программистам они должны быть известны, хотя бы из других языков программирования. Тем не менее я покажу пару примеров работы. Самый простой, с чего начнем, – это стек. Думаю, нет смысла рассказывать, что стек – это такая структура, в которой элемент, пришедший первым, будет извлечен из нее последним:

```
<?php

$Stack = new SplStack();

$Stack->push('a');
$Stack->push('b');
$Stack->push('c');

printf("%s \n", $Stack->pop() );
printf("%s \n", $Stack->pop() );
printf("%s \n", $Stack->pop() );

?>
```

На выходе получим: c,b,a. Более сложные структуры данных, такие как `SplPriorityQueue`, позволяют формировать структуры, отсортированные по приоритету.

```
<?php

$PQueue = new SplPriorityQueue();
$PQueue->setExtractFlags( SplPriorityQueue::EXTR_DATA );

$PQueue->insert('последний элемент',1);
$PQueue->insert('первый элемент', 3);
$PQueue->insert('средний элемент', 2);

printf("Верхний элемент: %s \n", $PQueue->top() );
printf("%s \n", $PQueue->extract() );
printf("%s \n", $PQueue->extract() );
printf("%s \n", $PQueue->extract() );


?>
```

На выходе получим:

```
Приоритетный элемент: первый элемент
первый элемент
средний элемент
последний элемент
```

Более подробно о данных расширениях вы можете прочитать на <http://ru2.php.net/manual/ru/book.spl.php>.

## The end...

Рассказывать о всех нововведениях можно очень долго. Но главное то, что, по заверениям разработчиков, PHP 5.3 от PHP 6 будет отличаться только поддержкой юникода. А значит, если вы научитесь работать с PHP 5.3, у вас не будет проблем с переходом на шестую версию. Да, это, конечно, произойдет не сейчас, но это произойдет. Жизнь не стоит на месте, как и прогресс. Надо всегда быть на гребне волны и смотреть в будущее. Всем известна простая истина: кто осведомлен – тот вооружен! Вооружайтесь знаниями сейчас, чтобы быть среди лидеров веб-разработки. Язык PHP «растет» и развивается, и мы, девелоперы, также развиваемся вместе с ним. Добавление таких инструментов в язык, как пространства имен и элементы функционального программирования, не говоря уже об объектно-ориентированных инструментах, позволяет наконец-то относиться к PHP как к серьезному языку программирования. Конечно, это еще пока только альфа-версия языка, но релиз не за горами. Стабильная версия PHP 5.3 ожидается уже к концу октября 2008 года. Так что в ваших интересах начать освоение новых возможностей «старого» инструмента. 

1. <http://ru.php.net/manual/ru/book.spl.php>.
2. <http://wiki.php.net/doc/scratchpad/upgrade/53>.
3. <http://ru2.php.net/manual/ru/book.spl.php>.
4. <http://ru.php.net/ini.sections>.