Zephir Documentation

Выпуск 0.10.9

Zephir Team

мая 13, 2018

Оглавление

Глава 1

Содержание

1.1 Добро пожаловать!

Встречайте Zephir, открытый, высокоуровневый, специализированный язык разработанный для быстрого и удобного создания расширений для PHP с упором на типизацию и безопасное управление памятью.

1.1.1 Некоторые особенности

Основные особенности Zephir:

Типизация	динамическая/статическа
Память	Указатели и ручное выделение памяти запрещены
Компиляция	Перед исполнением
Управление памятью	Свой сборщик мусора

1.1.2 Попробуйте

Этот код регистрирует класс с методом, который оставляет в строке только буквы:

```
namespace MyLibrary;

/**
 * Filter
 */
class Filter
{
    /**
     * Filters a string returning its alpha characters
     */
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    public function alpha(string str)
    {
        char ch; string filtered = "";

        for ch in str {
           if (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') {
              let filtered .= ch;
           }
        }

        return filtered;
    }
}
```

А теперь используем этот класс в PHP:

```
<?php

$filter = new MyLibrary\Filter();
echo $filter->alpha("01he#l.lo?/1"); // выведет hello
```

1.2 Почему Zephir?

Сегодня PHP приложения должны сбалансировать ряд проблем, в том числе стабильности, производительности и функциональности. Каждое PHP приложение основано на наборе общих компонентов, которые также являются базой для большинства приложений.

Эти общие компоненты библиотек/фреймворков или их сочетание. После установки фреймворки меняются редко, являясь основой приложения они должны быть очень функциональным, а также очень быстрыми.

Получение быстрых и надежных библиотек может быть затруднено из-за высоких уровней абстракции, которые, как правило, реализуются в них. Учитывая условие, что базовые библиотеки или фреймворки меняются редко, есть возможность создавать расширения, которые обеспечивают эту функциональность используя компиляцию для улучшение производительности и использования ресурсов.

С Zephir, вы можете реализовать объектно-ориентированные библиотеки/фреймворки/приложения, которые могут быть использованы в PHP, экономя важные секунды, которые могут сделать ваше приложение быстрее, улучшая его потребительские свойства.

1.2.1 Если вы PHP программист. . .

PHP является одним из самых популярных языков, используемых для разработки веб-приложений. Динамически типизированные и интерпретируемые языки, как PHP предлагают очень высокую скорость разработки благодаря их гибкости.

Начиная с версии 4, а затем 5, PHP основан на реализации движка Zend Engine. Это виртуальная машина, которая выполняет PHP код из представления в байт-коде. Zend Engine практически присутствует в каждой установке PHP в мире, с Zephir, вы можете создавать расширения для PHP, работающих под Zend Engine.

Zephir основывается на PHP, поэтому они, очевидно, имеют много общего, однако; они имеют существенные различия, которые придают Zephir свою собственную индивидуальность. Например, Zephir

является более строгим, и это может быть сделать вас менее производительным по сравнению с PHP из-за шага компиляции.

1.2.2 Если вы C программист...

C является одним из самых мощных и популярных языков когда-либо созданных. На самом деле, PHP написан на C, это является одной из причин, почему расширения PHP доступны для него. C дает вам свободу для управления памятью, использования типы низкого уровня и даже встроенные ассемблерные процедуры.

Тем не менее, разработка больших приложений на C может занять гораздо больше времени, чем ожидалось, по сравнению с PHP или Zephir и некоторые ошибки может быть сложно найти, если вы не являетесь опытным разработчиком.

Zephir был разработан, чтобы быть безопасным, поэтому он не реализует указатели или ручное управление памятью, так что если вы программируете на C, вы будете чувствовать, что Zephir менее мощный, но более дружественным, чем C.

1.2.3 Компиляция против интерпретации

Компиляция обычно замедляет разработку; вам потребуется немного больше терпения, чтобы компилировать ваш код перед его запуском. Соответственно, интерпретация ведет к снижению производительности в пользу скорости разработки. В некоторых случаях, не существует какой-либо заметной разницы между скоростью интерпретируемого и скомпилированного кода.

Zephir требует компиляции кода, однако, его функциональность используется с PHP, который интерпретируется.

После компиляции кода не нужно делать ее снова, однако, интерпретируемый код интерпретируется каждый раз, когда он запускается. Разработчик может решить, какие части приложения должны быть написаны на Zephir и какие нет.

1.2.4 Статическая типизация против динамической типизации

Вообще говоря, в статически типизированном языке, переменная связана с конкретным типом всю свою жизнь. Её тип не может быть изменен, и он может ссылаться только на типо-совместимые экземпляры и операции. Языки, типа C/C ++ были реализованы со схемой:

```
int a = 0;
a = "hello"; // не позволяется
```

В динамической типизации, тип связан со значением, а не переменной. Таким образом, переменная может ссылаться на тип значения, а затем переназначить позже на значение несвязанного типа. Javascript/PHP - примеры динамического типизированного языка:

```
var a = 0;
a = "hello"; // позволяется
```

Несмотря на свои преимущества производительности, динамические Языки могут быть не лучший выбор для всех приложений, в частности, для очень больших кодовых баз и высокопроизводительных приложений.

Оптимизация производительности динамического языка, такого как PHP, более сложна, чем для статического языка, такого как C. На статическом языке оптимизаторы могут использовать информацию

типов для принятия решений. В динамическом языке для оптимизатора доступно меньше таких ключей, что затрудняет выбор вариантов оптимизации.

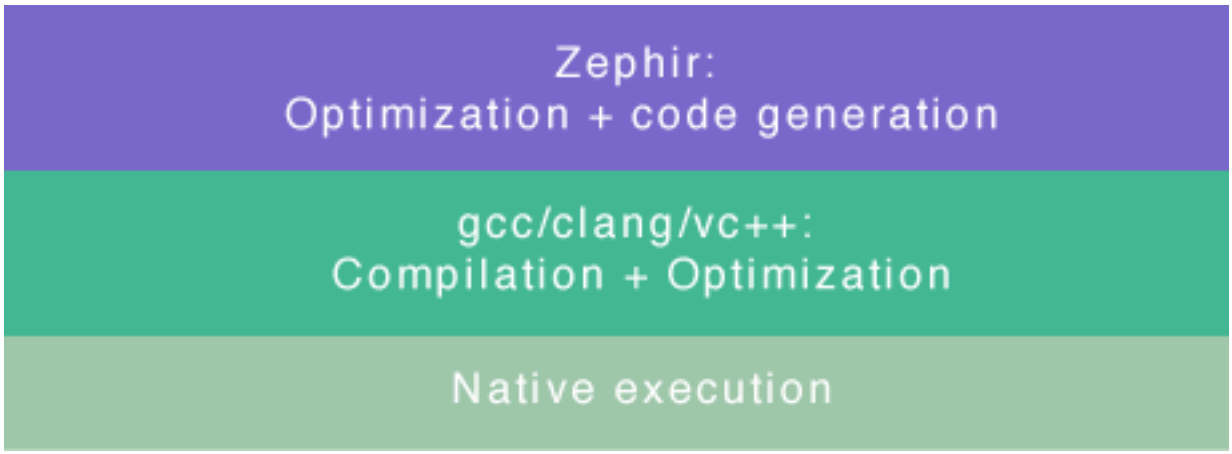
Хотя последние достижения в оптимизации динамических языков являются многообещающими (например, компиляция JIT), они отстают от уровня техники для статических языков. Таким образом, если вам требуется очень высокая производительность, статические языки, вероятно, более безопасный выбор.

Еще одно небольшое преимущество статических языков - дополнительная проверка компилятора. Компилятор не может найти логические ошибки, которые намного важнее, но компилятор может заранее обнаружить ошибки, которые в динамическом языке могут быть найдены только во время выполнения.

Zephir статически и динамически типизируется, позволяя вам использовать преимущества с обеих сторон, где это возможно.

1.2.5 Схема компиляции

Zephir предлагает генерацию собственного кода (в настоящее время через компиляцию в C), компилятор, подобный gcc/clang/vc ++, оптимизирует и компилирует код до машинного кода. На следующем графике показано, как работает этот процесс:



В дополнение к тому, что предоставляет Zephir, Со временем, в компиляторе были реализованы и назрели ряд оптимизаций, которые улучшают производительность скомпилированных приложений:

- GCC оптимизации
- LLVM проходы
- Visual C/C++ оптимизации

1.2.6 Защита кода

В некоторых случаях компиляция не приводит к существенному повышению производительности, это может быть связано с тем, что узкое место расположено в области ввода-вывода приложения (что вполне вероятно), а не в вычислении/памяти. Тем не менее, компиляция кода может также обеспечить определенный уровень интеллектуальной защиты для вашего приложения. Благодаря Zephir, производящему конечные двоичные файлы, вы также получаете возможность скрывать код для пользователей или клиентов.

1.2.7 Заключение

Zephir был создан не для замены PHP или C, а вместо этого, мы считаем, что это дополнение к ним, что позволяет разработчикам приступить к компиляции кода и статической типизации. Zephir - это попытка приобщиться к хорошим вещам из мира C и PHP, для возможности ускорения приложений.

1.3 Введение в Zephir

Zephir - это язык, который обращается к основным потребностям разработчика PHP, пытающегося писать и компилировать код, который может быть выполнен PHP. Он динамически/статически типизирован, некоторые его функции могут быть знакомы разработчикам PHP.

Имя Zephir является сокращение слов Zend Engine/PHP/Intermediate. Хотя это говорит о том, что произношение должно быть зефир, создатели Zephir фактически произносят его zaefire.

1.3.1 Hello World!

Каждый язык имеет свой собственный образец «Hello World!», в Zephir этот вводный пример демонстрирует некоторые важные особенности этого языка.

Код в Zephir должны быть помещены в классы. Этот язык предназначен для создания объектно-ориентированных библиотек/фреймворков, поэтому использование кода без класса не допускается. Кроме того, пространство имен необходимо:

```
namespace Test;

/**
 * This is a sample class
 */
class Hello
{
    /**
     * This is a sample method
     */
    public function say()
    {
        echo "Hello World!";
    }
}
```

После компиляции этого класса он создает следующий код, который прозрачно компилируется gcc/clang/vc++:

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_test.h"
#include "test.h"

#include "kernel/main.h"

/**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 * This is a sample class
 */
ZEPHIR_INIT_CLASS(Test_Hello) {
    ZEPHIR_REGISTER_CLASS(Test, Hello, hello, test_hello_method_entry, 0);
    return SUCCESS;
}

/**
 * This is a sample method
 */
PHP_METHOD(Test_Hello, say) {
    php_printf("%s", "Hello World!");
}
```

На самом деле, не ожидается, что разработчик, который использует Zephir должен знать или даже понимать C, однако, если у вас есть опыт работы с компиляторами, PHP внутренностей или самого языка C, это обеспечит более четкий сценарий разработчику при работе с Zephir.

1.3.2 Понимание Zephir

В следующих примерах мы опишем достаточно подробностей, чтобы вы поняли, что происходит. Цель состоит в том, чтобы дать вам понять, что такое программирование в Zephir. Мы рассмотрим детали функций в последующих главах.

Следующий пример очень прост, он реализует класс и метод с небольшой программой, которая проверяет типы массива

Рассмотрим код подробно, чтобы мы могли начать изучать синтаксис Zephir. В нескольких строках кода много деталей! Мы объясним общие идеи здесь:

```
namespace Test;

/**
 * MyTest (test/mytest.zep)
 */
class MyTest
{
    public function someMethod()
    {
        /* Переменные должны быть объявлены */
        var myArray;
        int i = 0, length;

        /* Создать массив */
        let myArray = ["hello", 0, 100.25, false, null];

        /* Подсчитайте массив в 'INT' переменной */
        let length = count(myArray);

        /* Вывод типов переменных */
        while i < length {
            echo typeof myArray[i], "\n";
            let i++;
        }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return myArray;
    }
}
```

В методе в первых строках ключевые слова «var» и «int» используются для объявления переменной в локальной области. Каждая переменная, используемая в методе, должна быть объявлена с соответствующим типом. Эта декларация не является обязательным, это помогает компилятору сообщать вам о неверном вводе переменных или об использовании переменных из сферы их применения, который обычно заканчивается ошибки во время выполнения.

Динамические переменные объявляются с помощью ключевого слова «var». Эти переменные могут быть назначены и переназначены для разных типов. С другой стороны, у нас есть «i» и «length» целочисленные статические типизированные переменные, которые могут иметь только значения этого типа во всем исполнении программы.

В отличие от PHP, вам не нужно ставить знак доллара ($) перед именами переменных.

Zephir следует тем же соглашениям о комментариях, что и Java, C #, C ++ и т. Д. //Комментарий идет до конца строки, в то время как /*комментарий*/ может пересекать границы строки.

Переменные по умолчанию неизменяемы, это означает, что Zephir ожидает, что большинство переменных остаются неизменными. Переменные, которые сохраняют свое первоначальное значение может быть оптимизирована компилятором статических констант. Когда значение переменной должно быть изменено, должно быть использовано ключевое слово „let“:

```
/* Создать массив */
let myArray = ["hello", 0, 100.25, false, null];
```

По умолчанию, массивы являются динамическими, как в PHP, они могут содержать значения различных типов. Функции из PHP пользовательского уровня можно вызывать в коде Zephir, в примере, вызывается функция „Count“, компилятор может выполняя оптимизацию избежать этого вызова, потому что он уже знает размер массива:

```
/* Подсчитать массив в 'INT' переменной */
let length = count(myArray);
```

Скобки в операторах управления потоком данных не являются обязательными, вы можете использовать их, если вам так больше нравится.

```
while i < length {
    echo typeof myArray[i], "\n";
    let i++;
}
```

PHP работает только с динамическими переменными, методы всегда возвращают динамические переменные, это означает, что если возвращается статическая типизированная переменная, со стороны PHP вы получите динамическую переменную, которая может быть использована в PHP-коде. Обратите внимание, что память автоматически управляется компилятором, так что вам не нужно выделять или освобождать память, как и в C, работает аналогичным образом, как и PHP.

1.4 Установка

1.4.1 Системные требования

Чтобы собрать расширение под PHP и использовать Zephir нужно:

- gcc >= 4.x/clang >= 3.x
- re2c 0.13 or later
- gnu make >=3.81
- autoconf >=2.31
- automake >=1.14
- libpcre3
- php development headers and tools

На Ubuntu эти пакеты можно поставить так:

```
$ sudo apt-get update
$ sudo apt-get install git gcc make re2c php7.0 php7.0-json php7.0-dev libpcre3-dev
```

Так как Zephir написан на PHP, вам нужно установить последнюю версию PHP. PHP должен быть доступен из консоли:

```
$ php -v
PHP 7.0.8 (cli) (built: Jun 26 2016 00:59:31) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
        with Zend OPcache v7.0.8, Copyright (c) 1999-2016, by Zend Technologies
```

Также проверьте, доступны ли dev-инструменты для сборки расширений:

```
$ phpize -v
Configuring for:
PHP Api Version:         20151012
Zend Module Api No:      20151012
Zend Extension Api No:   320151012
```

1.4.2 Установка Zephir

Склонируйте репозиторий Zephir:

```
$ git clone https://github.com/phalcon/zephir
$ cd zephir
```

Чтобы скомпилировать и установить Zephir выполните следующие команды:

```
$ cd zephir
$ ./install -c
```

1.4.3 Протестируйте Zephir

Проверьте, доступен ли Zephir из любой директории командой:

```
$ zephir help
```

1.5 Урок

Zephir и эта книга предназначены для разработчиков PHP, которые хотят создавать C-расширения с меньшей сложностью.

Мы предполагаем, что вы знакомы с одним или несколькими языками программирования. Мы проводим параллели с функциями в PHP, C, Javascript и других языках. Если вы знаете какой-либо из этих языков, мы укажем на похожие функции в Zephir, а также много новых или отличий.

1.5.1 Проверка установки

Если вы успешно установили Zephir, вы должны быть в состоянии выполнить следующую команду в своей консоли:

```
$ zephir help
```

Если все в порядке, на вашем экране должна появиться следующая справка:

```
 _____               __    _
/__  /  ___  ____  / /_  (_)____
  / /  / _ \/ __ \/ __ \/ / ___/
 / /__/  __/ /_/ / / / / / /
/____/\___/ .___/_/ /_/_/_/
         /_/

Zephir version 0.10.9a-dev

Usage:
    command [options]

Available commands:
    stubs               Generates extension PHP stubs
    install             Installs the extension (requires root password)
    version             Shows the Zephir version
    compile             Compile a Zephir extension
    api [--theme-path=/path][--output-directory=/path][--theme-options={json}|/path]Generates a␣
↪HTML API
    init [namespace]    Initializes a Zephir extension
    fullclean           Cleans the generated object files in compilation
    builddev            Generate/Compile/Install a Zephir extension in development mode
    clean               Cleans the generated object files in compilation
    generate            Generates C code from the Zephir code
    help                Displays this help
    build               Generate/Compile/Install a Zephir extension

Options:
    -f([a-z0-9\-]+)     Enables compiler optimizations
    -fno-([a-z0-9\-]+)  Disables compiler optimizations
    -w([a-z0-9\-]+)     Turns a warning on
    -W([a-z0-9\-]+)     Turns a warning off
```

1.5.2 Каркас расширения

Первое, что нам нужно сделать, это сгенерировать скелет расширения, это предоставит нашему расширению базовую структуру, которую мы должны начать работать. В нашем случае мы создадим расширение под названием «utils»:

```
$ zephir init utils
```

После этого в текущем рабочем каталоге создается каталог с именем «utils»:

```
utils/
   ext/
   utils/
```

Каталог «ext/» (внутри utils) содержит код, который будет использоваться компилятором для создания расширения. Другой созданный каталог - «utils», этот каталог имеет то же самое, что и наше расширение. Мы разместим код Zephir в этом каталоге.

Нам нужно изменить рабочий каталог на «utils», чтобы начать компилировать наш код:

```
$ cd utils
$ ls
ext/ utils/ config.json
```

В листинге каталога также будет отображаться файл с именем «config.json». Этот файл содержит параметры конфигурации, которые мы можем использовать для изменения поведения Zephir и/или этого расширения.

1.5.3 Добавление нашего первого класса

Zephir предназначен для создания объектно-ориентированных расширений. Чтобы начать разработку функциональности, нам нужно добавить наш первый класс к расширению.

Как и во многих языках/инструментах, первое, что мы хотим сделать, это увидеть «hello world», сгенерированный Zephir, и проверить, что все в порядке. Итак, наш первый класс будет называться «Utils\Greeting» и он содержит метод печати «hello world!».

Код для этого класса должен быть помещен в «utils/utils/greeting.zep»:

```
namespace Utils;

class Greeting
{

    public static function say()
    {
        echo "hello world!";
    }

}
```

Теперь нам нужно сказать Zephir, что наш проект должен быть скомпилирован и расширение сгенерировано:

```
$ zephir build
```

Изначально и только в первый раз выполняется ряд внутренних команд, создающих необходимый код и конфигурации, чтобы экспортировать этот класс в расширение PHP, если все пойдет хорошо, вы увидите следующее сообщение в конце вывода:

```
...
Extension installed!
Add extension=utils.so to your php.ini
Don't forget to restart your web server
```

На этом этапе вполне вероятно, что вам потребуется указать пароль root, чтобы установить расширение. Наконец, расширение должно быть добавлено в php.ini для загрузки PHP. Это достигается добавлением директивы инициализации: extension=utils.so к нему.

1.5.4 Первоначальное тестирование

Теперь, когда расширение было добавлено в ваш php.ini, проверьте, правильно ли загружается расширение, выполнив следующее:

```
$ php -m
[PHP Modules]
Core
date
libxml
pcre
Reflection
session
SPL
standard
tokenizer
utils
xdebug
xml
```

Расширения «utils» должны быть частью вывода, указывающего, что расширение было загружено правильно. Теперь давайте посмотрим на наш «hello world», непосредственно выполняемый PHP. Для этого вы можете создать простой PHP-файл, вызывающий статический метод, который мы только что создали:

```
<?php

echo Utils\Greeting::say(), "\n";
```

Поздравляем !, у вас есть первое расширение, работающее на PHP.

1.5.5 Удобные класс

Класс «hello world» был хорош, чтобы проверить, правильная ли наша среда, теперь давайте создадим еще несколько полезных классов.

Первый полезный класс, который мы добавим к этому расширению, предоставит пользователям средства фильтрации. Этот класс называется «Utils\Filter», и его код должен быть помещен в «utils/utils/filter.zep»:

Основным скелетом этого класса является следующее:

```
namespace Utils;

class Filter
{

}
```

Класс содержит методы фильтрации, которые помогают пользователям фильтровать нежелательные символы из строк. Первый метод называется «alpha», и его целью является отфильтровать только те символы, которые являются основными буквами ascii. Для начала, мы просто пройдем через строковую печать каждого байта в стандартный вывод:

```
namespace Utils;

class Filter
{

    public function alpha(string str)
    {
        char ch;

        for ch in str {
            echo ch, "\n";
        }

    }

}
```

При вызове этого метода:

```
<?php

$f = new Utils\Filter();
$f->alpha("hello");
```

Ты увидишь:

```
h
e
l
l
o
```

Проверка каждого символа в строке проста, теперь мы можем просто создать другую строку с правильными отфильтрованными символами:

```
class Filter
{

    public function alpha(string str) -> string
    {
        char ch; string filtered = "";

        for ch in str {
            if (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') {
                let filtered .= ch;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            }
        }

        return filtered;
    }
}
```

Полный метод может быть проверен, как и прежде:

```
<?php

$f = new Utils\Filter();
echo $f->alpha("$he$0213'121lo."); // prints "hello"
```

В следующем скринкасте вы можете посмотреть, как создать расширение, объясненное в этом уроке:

1.5.6 Заключение

Это очень простой учебник, и, как вы можете видеть, легко начать создание расширений с помощью Zephir. Мы приглашаем вас продолжить чтение руководства, чтобы вы могли ознакомиться с дополнительными функциями, которые предлагает Zephir!

1.6 Базовый синтаксис

В этой главе мы обсудим организацию файлов и пространств имен, объявления переменных, разные синтаксические соглашения и несколько других концепций.

1.6.1 Организация кода в файлах и пространствах имен

В PHP вы можете поместить код в любой файл без определенной структуры. В Zephir каждый файл должен содержать класс (и только один класс). Каждый класс должен иметь пространство имен, а структура каталогов должна соответствовать именам используемых классов и имен.

Например, для следующей структуры классы в каждом файле должны быть:

```
mylibrary/
        router/
                exception.zep # MyLibrary\Router\Exception
        router.zep # MyLibrary\Router
```

Класс в mylibrary/router.zep:

```
namespace MyLibrary;

class Router
{

}
```

Класс в mylibrary/router/exception.zep:

```
namespace MyLibrary\Router;

class Exception extends \Exception
{

}
```

Zephir поднимет исключение компилятора, если файл или класс не находится в ожидаемом файле или наоборот.

1.6.2 Разделение инструкций

Возможно, вы уже заметили, что в примерах кода в предыдущей главе было очень мало точек с запятой. Вы можете использовать точки с запятой для разделения операторов и выражений, как в Java, C / C ++, PHP и подобных языках:

```
myObject->myMethod(1, 2, 3); echo "world";
```

1.6.3 Комментарии

Zephir поддерживает комментарии „C“/“C++“, это одна строка комментариев с // и многострочными комментариями с /* ... */:

```
// Это комментарий в одной строке

/**
 * Многострочный комментарий
 */
```

В большинстве языков комментарии - это просто текст, игнорируемый компилятором/интерпретатором. В Zephir многострочные комментарии также используются в качестве док-блоков, и они экспортируются в сгенерированный код, поэтому они являются частью языка !.

Если док-блок не находится там, где ожидается, компилятор выдаст исключение.

1.6.4 Объявления переменных

В Zephir все переменные, используемые в заданной области видимости, должны быть объявлены. Этот процесс предоставляет компилятору важную информацию для выполнения оптимизаций и проверок. Переменные должны быть уникальными идентификаторами, и они не могут быть зарезервированными словами.

```
//Объявление переменных для одного и того же типа в одной команде
var a, b, c;

//Объявление каждой переменной в разных строках
var a;
var b;
var c;
```

Переменные могут дополнительно иметь начальное совместимое значение по умолчанию, вы можете присвоить новое значение переменной так часто, как вы хотите:

```
//Объявление переменных со значениями по умолчанию
var a = "hello", b = 0, c = 1.0;
int d = 50; bool some = true;
```

Имена переменных чувствительны к регистру, следующие переменные различаются:

```
//Различные переменные
var somevalue, someValue, SomeValue;
```

1.6.5 Область переменной

Все объявленные переменные локально охвачены методом, в котором они были объявлены:

```
namespace Test;

class MyClass
{

    public function someMethod1()
    {
        int a = 1, b = 2;
        return a + b;
    }

    public function someMethod2()
    {
        int a = 3, b = 4;
        return a + b;
    }

}
```

1.6.6 Супер-глобальные переменные

Zephir не поддерживает глобальные переменные, доступ к глобальным переменным из пользовательского домена PHP недопустим. Тем не менее, вы можете получить доступ к супер-глобальным объектам PHP следующим образом:

```
//Получение значения от _POST
let price = _POST["price"];

//Чтение значения из _SERVER
let requestMethod = _SERVER["REQUEST_METHOD"];
```

1.6.7 Локальная таблица символов

Каждый метод или контекст в PHP имеет таблицу символов, которая позволяет писать переменные очень динамичным способом:

```
<?php

$b = 100;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
$a = "b";
echo $$a; // prints 100
```

Zephir не реализует эту функцию, так как все переменные скомпилированы до низкоуровневых переменных и не существует способа узнать, какие переменные существуют в определенном контексте. Если вы хотите создать переменную в текущей таблице символов PHP, вы можете использовать следующий синтаксис:

```
//Установить переменную $name в PHP
let {"name"} = "hello";

//Установить переменную $price в PHP
let name = "price";
let {name} = 10.2;
```

1.7 Типы

Zephir динамически и статически типизируется. В этой главе мы выделим поддерживаемые типы и их поведение:

1.7.1 Динамический тип

Динамические переменные точно так, как в PHP, они могут быть назначены и переназначены в различные типы без ограничений.

Динамическая переменная должна быть объявлена с помощью служебного слова „var“, поведение с ней идентична как и PHP:

```
var a, b, c;

// Инициализировать переменные
let a = "hello", b = false;

// Change their values
let a = 10, b = "140";

// Выполнять операции между ними
let c = a + b;
```

Всего существует 8 типов:

Тип	Описание
boolean	Логическое выражение выражает истинное значение. Он может быть „true“ или „false“.
integer	Целые числа. Размер целого числа зависит от платформы.
float/double	Числа с плавающей запятой. Размер зависит от платформы.
string	Строка представляет собой последовательность символов, где символ совпадает с байтом.
array	Массив - это упорядоченная карта. Карта - это тип, который связывает значения с ключами
object	Абстракция объекта, как в PHP
resource	Ресурс содержит ссылку на внешний ресурс
null	Специальное значение NULL представляет переменную без значения

Подробнее об этих типах можно узнать в PHP manual

Boolean

Логическое выражение выражает истинное значение. Он может быть „true“ или „false“:

```
var a = false, b = true;
```

Integer

Целые числа. Размер целого числа зависит от платформы, хотя максимальное значение около двух миллиардов является обычным значением (это 32 бита, подписанного). 64-битные платформы обычно имеют максимальное значение около 9E18. PHP не поддерживает целые числа без знака, поэтому у Zephir есть это ограничение тоже:

```
var a = 5, b = 10050;
```

Целочисленное переполнение

В отличие от PHP, Zephir не проверяет автоматически переполнение целочисленного типа, как в C, если вы выполняете операции, которые могут возвращать большое число, вы можете использовать такие типы, как «unsigned long» или «float» для их хранения:

```
unsigned long my_number = 2147483648;
```

Float/Double

Числа с плавающей запятой (также известные как «floats», «doubles», или «real numbers»). Литералы с плавающей запятой представляют собой выражения из нуля или более цифр, за которыми следует точка (.), За которой следуют ноль или более цифр. Размер float зависит от платформы, хотя максимум ~ 1.8e308 с точностью примерно 14 десятичных цифр является общим значением (64-битный формат IEEE).

```
var number = 5.0, b = 0.014;
```

Числа с плавающей запятой имеют ограниченную точность. Хотя это зависит от системы, как PHP, Zephir использует формат двойной точности IEEE 754, который даст максимальную относительную ошибку из-за округления порядка 1.11e-16.

String

Строка представляет собой последовательность символов, где символ совпадает с байтом. Как PHP, Zephir поддерживает только 256-символьный набор и, следовательно, не предлагает поддержку Unicode.

```
var today = "friday";
```

В Zephir строковые литералы могут указываться только с помощью двойных кавычек (как в C), одинарные кавычки зарезервированы для символов.

В строках поддерживаются следующие escape-последовательности:

Последовательность	Описание
\t	Горизонтальная табуляция
\n	Перевод строки
\r	Возврат каретки
\ \	Обратная косая черта
"	Двойная кавычка

```
var today = "\tfriday\n\r",
    tomorrow = "\tsaturday";
```

В Zephir строки не поддерживают парсинг переменных, как в PHP, вместо этого вы можете использовать конкатенацию:

```
var name = "peter";

echo "hello: " . name;
```

Arrays

Реализация массива в Zephir в основном такая же, как и в PHP: упорядоченные карты, оптимизированные для различных целей; Его можно рассматривать как массив, список (вектор), хэш-таблицу (реализацию карты), словарь, коллекцию, стек, очередь и, возможно, больше. В качестве значений массива могут выступать другие массивы, деревья и многомерные массивы также возможны.

Синтаксис для определения массивов немного отличается от PHP:

```
//Квадратные фигурные скобки должны использоваться для определения массивов
let myArray = [1, 2, 3];

//Двойной двоеточие необходимо использовать для определения ключей хэшей
let myHash = ["first": 1, "second": 2, "third": 3];
```

В качестве ключей могут использоваться только long и string значения:

```
let myHash = [0: "first", 1: true, 2: null];
let myHash = ["first": 7.0, "second": "some string", "third": false];
```

Objects

Zephir позволяет создавать экземпляры, манипулировать, вызывать методы, читать константы классов и т.д. Из объектов PHP:

```
let myObject = new stdClass(),
    myObject->someProperty = "my value";
```

1.7.2 Static Types

Статическая типизация позволяет разработчику объявлять и использовать некоторые типы переменных, доступные в C. Переменные не могут изменить их тип, как только они объявлены как динамические типы. Тем не менее, они позволяют компилятору делать лучшую оптимизационную работу. Поддерживаются следующие типы:

Тип	Описание
boolean	Логическое выражение. Оно может быть либо „true“ или „false“.
integer	Целые числа со знаком. Минимум 16 бит.
unsigned integer	Целочисленные без знака. Минимум 16 бит.
char	Наименьшая адресная единица машины, которая может содержать базовый символ.
unsigned char	Тот же размер, что и char, но гарантированно без знака.
long	Длинное целое со знаком. Минимум 32 бита.
unsigned long	То же, что и long, но без знака.
float/double	Тип с плавающей запятой двойной точности. Размер зависит от платформы.
string	Строка представляет собой последовательность символов, где символ совпадает с байтом.
array	Структура, которая может использоваться в качестве хеша, карты, словаря, коллекции, стека и т. д.

Boolean

Логическое выражение выражает истинное значение. Он может быть „true“ или „false“. В отличие от динамического поведения статические логические типы остаются логическими (true или false), а не тем значением, что им присваевается:

```
boolean a;

let a = true,
    a = 100, // автоматически переводится в true
    a = null, // автоматически переводится в false
    a = "hello"; // выдает исключение компилятора
```

Integer/Unsigned Integer

Целочисленные значения подобны целочисленному элементу в динамических значениях. Значения, присвоенные целым переменным, остаются целыми:

```
int a;

let a = 50,
    a = -70,
    a = 100.25, // автоматически переводится в 100
    a = null, // автоматически переводится в 0
    a = false, // автоматически переводится в 0
    a = "hello"; // выдает исключение компилятора
```

Целые числа без знака подобны целым числам, но они не имеют знака, это означает, что вы не можете хранить отрицательные числа в таких переменных:

```
let a = 50,
    a = -70, // автоматически переводится в 70
    a = 100.25, // автоматически переводится в 100
    a = null, // автоматически переводится в 0
    a = false, // автоматически переводится в 0
    a = "hello"; // выдает исключение компилятора
```

Целые числа без знака в два раза больше стандартных целых чисел, присваивать целые числа без знака целым может означать потерю данных:

```
uint a, int b;

let a = 2147483648,
    b = a, // возможная потеря данных
```

Long/Unsigned Long

Long переменные в два раза больше, чем integer переменные, поэтому они могут хранить большие числа. В качестве значений целых чисел, назначенных длинным переменным, автоматически присваивается этот тип:

```
long a;

let a = 50,
    a = -70,
    a = 100.25, // автоматически переводится в 100
    a = null, // автоматически переводится в 0
    a = false, // автоматически переводится в 0
    a = "hello"; // выдает исключение компилятора
```

Unsigned long похожи на long, но они не имеют знака, это означает, что вы не можете хранить отрицательные числа в таких переменных:

```
let a = 50,
    a = -70, // автоматически переводится в 70
    a = 100.25, // аавтоматически переводится в 100
    a = null, // автоматически переводится в 0
    a = false, // автоматически переводится в 0
    a = "hello"; // выдает исключение компилятора
```

Unsigned longs в два раза больше, чем стандартные longs, назначить unsigned longs longs может означать потерю данных:

```
ulong a, long b;

let a = 4294967296,
    b = a, // возможная потеря данных
```

Char/Unsigned Char

Переменная Char - наименьшая адресуемая единица машины, которая может содержать базовый символ из набора. Каждая переменная „char“ представляет один символ в строке:

```
char ch, string name = "peter";

let ch = name[2]; // сохраняет 't'
let ch = 'Z'; // литералы char должны быть заключены в простые кавычки
```

String

Строка представляет собой последовательность символов, где символ совпадает с байтом. Как и в PHP, он поддерживает только 256-символьный набор и, следовательно, не поддерживает использование Юникода.

Когда переменная объявлена как строка, она никогда не меняет тип:

```
string a;

let a = "",
    a = "hello", // строковые литералы должны быть заключены в двойные кавычки
    a = 'A', //  преобразуется в строку "A"
    a = null; // автоматически переводится в ""
```

1.8 Операторы

Операторы в Zephir похожи на их аналоги в PHP и также себя ведут.

1.8.1 Арифметические операторы.

Поддерживаемые операторы:

Оператор	Пример
Приведение к отрицательному	-a
Сложение	a + b
Вычитание	a - b
Умножение	a * b
Деление	a / b
Модуль	a % b

1.8.2 Операторы сравнения

Операции сравнения зависят от типа сравниваемых переменных, например если оба операнда динамические (var), результат будет таким же как и в PHP:

a == b	Равенство	TRUE если a равно b после приведения типов.
a === b	Идентичность	TRUE если a равно b, и операнды одного типа.
a != b	Не равны	TRUE если a не равно b после приведения типов.
a <> b	Не равны	TRUE если a не равно b после приведения типов.
a !== b	Не идентичны	TRUE если a не равно b, или операнды разных типов.
a < b	Меньше	TRUE если a строго меньше b.
a > b	Больше	TRUE если a строго больше b.
a <= b	Меньше, или равно	TRUE если a меньше, или равно b.
a >= b	Больше, или равно	TRUE если a строго больше, или равно b.

Пример:

```
if a == b {
    return 0;
} else {
    if a < b {
        return -1;
    } else {
        return 1;
    }
}
```

1.8.3 Логические опреаторы

Поддерживаемые операторы:

Операция	Пример
И	a && b
Или	a || b
Отрицание	!a

Пример:

```
if a && b || !c {
    return -1;
}
return 1;
```

1.8.4 Побитовые операторы

Поддерживаемые операторы:

Операция	Пример
И	a & b
Или	a | b
Исключающее или	a ^ b
Отрицание	~a
Сдвиг влево	a << b
Сдвиг вправо	a >> b

Пример:

```
if a & SOME_FLAG {
    echo "has some flag";
}
```

Вы узнаете больше о сравнении динамических переменных в php документации.

1.8.5 Тернарный оператор

Zephir поддерживает тернарный оператор, как в C или PHP.

```
let b = a == 1 ? "x" : "y"; // в b будет присвоен "x", если a равно 1 в противном случае "y"
```

1.8.6 Специальные операторы

Поддерживаемые операторы:

Empty

Empty позволяет узнать пусто ли выражение. Под „пусто“ подразумевается выражение возвращающее

- null
- пустую строку
- пустой массив

```
let someVar = "";
if empty someVar {
    echo "is empty!";
}

let someVar = "hello";
if !empty someVar {
    echo "is not empty!";
}
```

Isset

Проверяет, существует ли индекс у массива, или свойство у объекта:

```
let someArray = ["a": 1, "b": 2, "c": 3];
if isset someArray["b"] { // проверим, есть ли у массива индекс "b"
    echo "yes, it has an index 'b'\n";
}
```

Использование „isset“ возможно в return-конструкциях:

```
return isset this->{someProperty};
```

Учтите, что „isset“ в Zephir работает скорее как array_key_exists в PHP. То есть оператор вернет true даже если значение равно null.

Fetch

Оператор „fetch“ создан для сокращения популярной в PHP конструкции:

```
<?php

if (isset($myArray[$key])) {
    $value = $myArray[$key];
    echo $value;
}
```

В Zephir тот же код будет можно написать так:

```
if fetch value, myArray[key] {
    echo value;
}
```

„Fetch“ вернет true, если в массиве есть что-то по ключу „key“ и тогда в „value“ будет присвоенно значение.

Подсказка Типа

Zephir всегда пытается проверить, реализует ли объект методы и свойства, вызываемые/доступные для переменной, которая выводится как объект:

```
let o = new MyObject();

// Zephir проверяет, реализован ли «myMethod» в MyObject
o->myMethod();
```

Однако из-за динамизма, унаследованного от PHP, иногда нелегко узнать класс объекта, поэтому Zephir не может эффективно создавать отчеты об ошибках. Подсказка типа сообщает компилятору, какой класс связан с динамической переменной, позволяющей компилятору выполнять больше проверок компиляции:

```
// Сообщает компилятору, что "o"
// является экземпляром класса MyClass
let o = <MyClass> this->_myObject;
o->myMethod();
```

Подсказки прогнозирования ветвлений

Что такое прогнозирование ветвлений? Для подробного описания это понятия обратитесь к статье Игоря Островского или описанию на Wikipedia. В окружениях, где производительность является очень важной составляющей, может оказаться полезным использование подсказок при прогнозировании ветвлений.

Рассмотрим следующий пример:

```
let allPaths = [];
for path in this->_paths {
    if path->isAllowed() == false {
        throw new App\Exception("Some error message here");
    } else {
        let allPaths[] = path;
    }
}
```

Авторы кода, приведенного выше, заранее знают, что условие, которое выбрасывает исключение, вряд ли произойдет. Это означает, что в 99.9% случаев наш метод выполняет эту проверку в холостую – условие вероятно никогда не будет оцениваться как истинное. Но для процессора, обычно, это сложно понять, поэтому мы могли бы ввести здесь подсказку:

```
let allPaths = [];
for path in this->_paths {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    if unlikely path->isAllowed() == false {
        throw new App\Exception("Some error message here");
    } else {
        let allPaths[] = path;
    }
}
```

1.9 Массивы

Работа с массивом в Zephir работает так же как и в PHP arrays. Массивы реализованы через hash table.

1.9.1 Объявление переменных массива

Переменная тип массив может быть обьявлена с помощью ключевых слов „var“ or „array“:

```
var a = []; // массив, с возможностю переопределения типа
array b = []; // массив, без возможностю переопределения типа
```

1.9.2 Создание массивов

Массив создается заключая его элементы в квадратные скобки:

```
//Создание пустого массива
let elements = [];

//Создание массива с элементами
let elements = [1, 3, 4];

//Создание массива с элементами разных типов
let elements = ["first", 2, true];

//Создане многомерного массива
let elements = [[0, 1], [4, 5], [2, 3]];
```

Как и в PHP, поддерживаются простые(списки) и ассоциативные массивы:

```
//Создание массива с строковыми ключами
let elements = ["foo": "bar", "bar": "foo"];

//Создание массива с числоовыми ключами
let elements = [4: "bar", 8: "foo"];

//Создание массива со смешанными ключами, строковые и числовые
let elements = [4: "bar", "foo": 8];
```

1.9.3 Обновление массивов

Массивы обновляются так же, как в PHP, используя квадратные скобки:

```
//Обновление массива с строковым ключем
let elements["foo"] = "bar";

//Обновление массива с числовым ключем
let elements[0] = "bar";

//Обновление многомерного массива
let elements[0]["foo"] = "bar";
let elements["foo"][0] = "bar";
```

1.9.4 Добавление элементов

Элементы могут быть добавлены в конце массива следующим образом:

```
//Добавление элемента в массив
let elements[] = "bar";
```

1.9.5 Чтение элементов из массивов

Можно прочитать элементы массива следующим образом:

```
//Получение элемента используя строковый ключ "foo"
let foo = elements["foo"];

//Получение элемента используя числовой ключ 0
let foo = elements[0];
```

1.10 Классы и объекты

Zephir позволяет создавать только классы, однако также становится возможным использовать исключения, вмето фатальных ошибок, или предупреждений.

1.10.1 Классы

Каждый файл в Zephir должны содержать класс или интерфейс (причем только один). Структура класса крайне схожа с структурой обьявления его в PHP:

```
namespace Test;

/**
 * Это пример класса
 */
class MyClass
{

}
```

1.10.2 Реализация методов

Ключевое слово «function» декларирует новый метод. Методы имеют те же модификаторы для разрешения видимости, что и PHP, однако Zephir требует явно его указывать.

```
namespace Test;

class MyClass
{

    public function myPublicMethod()
    {
        // ...
    }

    protected function myProtectedMethod()
    {
        // ...
    }

    private function myPrivateMethod()
    {
        // ...
    }
}
```

Методы могут принимать как обязательные так и необязательные параметры:

```
namespace Test;

class MyClass
{

    /**
     * Все параметры обязательные
     */
    public function doSum1(a, b)
    {
        return a + b;
    }

    /**
     * Обязателен только 'a', 'b' не обязателен и имеет значение по умолчанию
     */
    public function doSum2(a, b=3)
    {
        return a + b;
    }

    /**
     * Оба параметра не обязательны
     */
    public function doSum3(a=1, b=2)
    {
        return a + b;
    }

    /**
```

(continues on next page)

(продолжение с предыдущей страницы)

```
     * Параметры обязательны, и они должны быть целочисленными
     */
    public function doSum4(int a, int b)
    {
        return a + b;
    }

    /**
     * Параметры обязательны, целочисленны и имеют значения по умолчанию
     */
    public function doSum4(int a=4, int b=2)
    {
        return a + b;
    }

}
```

Поддерживаемые видимости метода (Инкапсуляция)

- Открытый (public): Методы с модификатором «public» экспортируются в расширение, это значит, что этими методами можно пользоваться также, как и самому расширению.
- Защищенный (protected): Методы с модификатором «protected» экспортируются в расширение, это значит, что этими методами можно пользоваться также, как и самому расширению. Однако, защищенные методы могут быть вызваны либо внутри класса, либо наследником класса.
- Закрытый (private): Методы с модификатором «private» не экспортируются в расширение, это значит, что эти методы может использовать только класс, в котором метод реализован.

Поддерживаемые модификаторы

- Устаревший (deprecated): Методы с модификатором «deprecated» бросают ошибку (E_DEPRECATED) где они вызываются.

Сокращения для геттеров/сеттеров

Как в C#, вы можете использовать сокращения для get/set/toString методов. Это значит, что вы можете создавать геттеры и сеттеры для свойств не явно.

Рассмотрим как выглядит код без сокращений:

```
namespace Test;

class MyClass
{
    protected myProperty;

    protected someProperty = 10;

    public function setMyProperty(myProperty)
    {
        let this->myProperty = myProperty;
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    public function getMyProperty()
    {
        return this->myProperty;
    }

    public function setSomeProperty(someProperty)
    {
        let this->someProperty = someProperty;
    }

    public function getSomeProperty()
    {
        return this->someProperty;
    }

    public function __toString()
    {
        return this->myProperty;
    }

 }
```

Вы можете написать тот же код, используя сокращения:

```
namespace App;

class MyClass
{
    protected myProperty {
        set, get, toString
    };

    protected someProperty = 10 {
        set, get
    };

}
```

Zephir сгенерирует реальные методы, но вам не прийдется писать их самостоятельно.

Тип возвращаемого значения

Методы классов и интерфейсов могут объявлять тип возвращаемого значения. Это поможет компилятору подсказать вам о ошибках в расширении. Рассмотрим пример:

```
namespace App;

class MyClass
{
    public function getSomeData() -> string
    {
        // здесь будет сгенерирована ошибка
        // потому что возвращаемое значение (boolean)
        // не соответсвует ранее объявленному типу
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return false;
    }

    public function getSomeOther() -> <App\MyInterface>
    {
        // здесь будет сгенерирована ошибка
        // если возвращаемый объект не реализует
        // ожидаемый компилитором интерфейс
        return new App\MyObject;
    }

    public function process()
    {
        var myObject;

        // the type-hint will tell the compiler that
        // myObject is an instance of a class
        // that implement App\MyInterface
        let myObject = this->getSomeOther();

        // the compiler will check if App\MyInterface
        // implements a method called "someMethod"
        echo myObject->someMethod();
    }

}
```

Метод может содержать более одного возвращаемого типа. Для объявления нескольких типов возвращаемых значений, разделите их оператором „|“.

```
namespace App;

class MyClass
{
    public function getSomeData(a) -> string|bool
    {
        if a == false {
            return false;
        }
        return "error";
    }

}
```

Возвращаемый тип: void

Методы также могут быть помечены как „void“. Это значит, что метод не можете вернуть ничего.

```
public function setConnection(connection) -> void
{
    let this->_connection = connection;
}
```

Почему это полезно? Потому что если компилятор обнаружит, что программа использует возвращаемое значение, то сгенерирует исключение.

```
let myDb = db->setConnection(connection);
myDb->execute("SELECT * FROM robots"); // тут сгенерируется исключение
```

Строгие/Приводимые Типы Параметров

В Zephir вы можете определить тип для каждого параметра в методе. По умолчанию все типизированные аргументы приводимы. Это зачит, что если значение не соответсвует ожидаемому типу, Zephir сгенерирует код для его приведения к ожидаемому.

```
public function filterText(string text, boolean escape=false)
{
    //...
}
```

Этот метод будет работать так:

```
<?php

$o->filterText(1111, 1); // OK
$o->filterText("some text", null); // OK
$o->filterText(null, true); // OK
$o->filterText("some text", true); // OK
$o->filterText(array(1, 2, 3), true); // Ошибка
```

Однако, передача не правильного типа часто может вести к багам. Вы можете запретить автоматическое приведение, используя строгие типы:

```
public function filterText(string! text, boolean escape=false)
{
    //...
}
```

Теперь большинство вызовов сгенерируют исключения благодяря неправильному переданному типу:

```
<?php

$o->filterText(1111, 1); // Ошибка
$o->filterText("some text", null); // OK
$o->filterText(null, true); // Ошибка
$o->filterText("some text", true); // OK
$o->filterText(array(1, 2, 3), true); // Ошибка
```

Определяя какие параметры строгие, а какие приводимые, вы можете контролировать поведение так, этого хочет.

Read-Only Параметры

Используя ключевое слово „const“ вы можете объявить, что параметр только для чтения. Такие параметры не могут быть изменены внутри метода:

```
namespace App;

class MyClass
{
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    // "a" только для чтения
    public function getSomeData(const string a)
    {
        // компилятор сгенерирует ошибку
        let a = "hello";
    }
}
```

Когда параметр объявлен только для чтения, компилятор может безопасно создавать оптимизации над этими переменными.

1.10.3 Реализация свойств

Переменные-члены класса называются «properties». По умолчанию они действуют как свойства PHP. Свойства экспортируются в расширение PHP и являются видимыми из кода PHP. Свойства реализуют обычные модификаторы видимости, доступные в PHP, явно задавать модификатор видимости обязательно в Zephir:

```
namespace Test;

class MyClass
{

    public myProperty1;

    protected myProperty2;

    private myProperty3;

}
```

В пределах методов класса нестатические свойства могут быть доступны с помощью -> (Оператор объекта): this->property (где property - это имя свойства):

```
namespace Test;

class MyClass
{

    protected myProperty;

    public function setMyProperty(var myProperty)
    {
        let this->myProperty = myProperty;
    }

    public function getMyProperty()
    {
        return this->myProperty;
    }

}
```

Свойства могут иметь литерально совместимые значения по умолчанию. Эти значения должны быть в состоянии быть оценены во время компиляции и не должны зависеть от информации времени вы-

полнения для оценки:

```
namespace Test;

class MyClass
{

    protected myProperty1 = null;
    protected myProperty2 = false;
    protected myProperty3 = 2.0;
    protected myProperty4 = 5;
    protected myProperty5 = "my value";

}
```

Обновление свойств

Свойства можно обновить, обратившись к ним с помощью оператора „->“:

```
let this->myProperty = 100;
```

Zephir проверяет, что свойства существуют, когда программа обращается к ним, если свойство не объявлено, вы получите исключение компилятора:

```
CompilerException: Property '_optionsx' is not defined on class
'App\MyClass' in /Users/scott/cphalcon/phalcon/cache/backend.zep on line 62

      let this->_optionsx = options;
      ------------^
```

Если вы хотите избежать валидации компилятора или просто создать свойство динамически, вы можете заключить имя свойства, используя строковые кавычки:

```
let this->{"myProperty"} = 100;
```

Вы также можете использовать простую переменную для обновления свойства, имя свойства будет взято из переменной:

```
let someProperty = "myProperty";
let this->{someProperty} = 100;
```

Чтение свойств

Свойства можно прочитать, обратившись к ним с помощью оператора „->“:

```
echo this->myProperty;
```

Как и при обновлении, свойства могут динамически читаться следующим образом:

```
// Избежание проверки компилятора или чтение динамического пользовательского свойства
echo this->{"myProperty"};

//Чтение с использованием имени переменной
let someProperty = "myProperty";
echo this->{someProperty}
```

1.10.4 Константы классов

Класс может содержать константы класса, которые остаются неизменными и неизменными после компиляции расширения. Константы класса экспортируются в расширение PHP, что позволяет им использовать их с PHP.

```
namespace Test;

class MyClass
{

    const MYCONSTANT1 = false;
    const MYCONSTANT2 = 1.0;

}
```

К константам класса можно получить доступ, используя имя класса и статический оператор (: :):

```
namespace Test;

class MyClass
{

    const MYCONSTANT1 = false;
    const MYCONSTANT2 = 1.0;

    public function someMethod()
    {
        return MyClass::MYCONSTANT1;
    }

}
```

1.10.5 Взов Методов

Методы можно вызывать, используя оператор объекта (->), как в PHP:

```
namespace Test;

class MyClass
{

    protected function _someHiddenMethod(a, b)
    {
        return a - b;
    }

    public function someMethod(c, d)
    {
        return this->_someHiddenMethod(c, d);
    }

}
```

Статические методы должны вызываться с использованием статического оператора (: :):

```
namespace Test;

class MyClass
{

    protected static function _someHiddenMethod(a, b)
    {
        return a - b;
    }

    public static function someMethod(c, d)
    {
        return self::_someHiddenMethod(c, d);
    }

}
```

Вы можете вызвать методы динамическим способом следующим образом:

```
namespace Test;

class MyClass
{
    protected adapter;

    public function setAdapter(var adapter)
    {
        let this->adapter = adapter;
    }

    public function someMethod(var methodName)
    {
        return this->adapter->{methodName}();
    }

}
```

1.11 Встроенные методы

Как упоминалось ранее, Zephir способствует объектно-ориентированному программированию, переменные, относящиеся к статическим типам, также могут обрабатываться как объекты.

Сравните эти два метода:

```
public function binaryToHex(string! s) -> string
{
    var o = "", n; char ch;

    for ch in range(0, strlen(s)) {
        let n = sprintf("%X", ch);
        if strlen(n) < 2 {
            let o .= "0" . n;
        } else {
            let o .= n;
        }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    }
    return o;
}
```

и:

```
public function binaryToHex(string! s) -> string
{
    var o = "", n; char ch;

    for ch in range(0, s->length()) {
        let n = ch->toHex();
        if n->length() < 2 {
            let o .= "0" . n;
        } else {
            let o .= n;
        }
    }
    return o;
}
```

Оба они имеют одинаковую функциональность, а во втором - объектно-ориентированное программирование. Вызывающие методы для статических типизированных переменных не оказывают никакого влияния на производительность, поскольку Zephir внутренне преобразует код из объектно-ориентированной версии в процедурную версию.

1.11.1 String

Доступны следующие встроенные методы строки:

ООП	Процедурный	Описание
s->length()	strlen(s)	Получить длину строки
s->trim()	trim(s)	Удалять пробелы (или другие символы) из начала и конца строки
s->index(«foo»)	strpos(s, «foo»)	Найти позицию первого вхождения подстроки в строке
s->lower()	strtolower(s)	Строка в нижнем регистре
s->upper()	strtoupper(s)	Строка в верхнем регистре

1.11.2 Array

Доступны следующие встроенные методы массива:

ООП	Процедурный	Описание
a->join(» «)	join(» «, a)	Объединение элементов массива в строку
a->reverse()	array_reverse(a)	Возвращает массив с элементами в обратном порядке

1.11.3 Char

Доступны следующие встроенные методы char:

ООП	Процедурный
ch->toHex()	sprintf(«%X», ch)

1.11.4 Integer

Доступны следующие встроенные методы целочисленного типа:

ООП	Процедурный
i->abs()	abs(i)

1.12 Управляющие структуры

Zephir реализует упрощенный набор структур управления, присутствующих в подобных языках, таких как C, PHP и т.д.

1.12.1 Условные

Оператор If

Оператор „if“ оценивает выражение, выполняет свой блок, если оценка true. Фигурные скобки обязательны, „if“ может иметь необязательное предложение „else“, несколько конструкций „if“/“else“ могут быть соединены вместе:

```
if false {
    echo "false?";
} else {
    if true {
        echo "true!";
    } else {
        echo "neither true nor false";
    }
}
```

Вы также можете использовать „elseif“:

```
if a > 100 {
    echo "слишком много";
} elseif a < 0 {
    echo "слишком мало";
} elseif a == 50 {
    echo "идеально!";
} else {
    echo "сойдет";
}
```

Скобки в оцениваемом выражении необязательны:

```
if a < 0 { return -1; } else { if a > 0 { return 1; } }
```

Оператор Switch

Оператор „switch“ сравнивает выражение с рядом предопределенных значений литералов, и выполняет соответствующий блок „case“ или в случае неудачи, выполняет блок „default“:

```
switch count(items) {
    case 1:
    case 3:
        echo "odd items";
        break;
    case 2:
    case 4:
        echo "even items";
        break;
    default:
        echo "unknown items";
}
```

1.12.2 Циклы

Цикл While

„While“ обозначает цикл, который выполняет итерацию, пока его заданное условие принимает значение true:

```
let counter = 5;
while counter {
    let counter -= 1;
}
```

Цикл Loop

В дополнение к „while“, „loop“ может использоваться для создания бесконечных циклов:

```
let n = 40;
loop {
    let n -= 2;
    if n % 5 == 0 { break; }
    echo x, "\n";
}
```

Цикл For

Цикл „for“ является структурой управления, которая позволяет перебирать массивы или строки:

```
for item in ["a", "b", "c", "d"] {
    echo item, "\n";
}
```

Ключи в хэшах можно получить следующим образом:

```
let items = ["a": 1, "b": 2, "c": 3, "d": 4];

for key, value in items {
    echo key, " ", value, "\n";
}
```

Цикл „for“ также может быть проинструктирован об обходе массива или строки в обратном порядке:

```
let items = [1, 2, 3, 4, 5];

for value in reverse items {
    echo value, "\n";
}
```

Цикл „for“ может использоваться для перемещения по строковым переменным:

```
string language = "zephir"; char ch;

for ch in language {
    echo "[", ch ,"]";
}
```

В обратном порядке:

```
string language = "zephir"; char ch;

for ch in reverse language {
    echo "[", ch ,"]";
}
```

Стандартный „for“, который проходит диапазон целочисленных значений, можно записать следующим образом:

```
for i in range(1, 10) {
    echo i, "\n";
}
```

Оператор break

„break“ ends execution of the current „while“, „for“ or „loop“ statements: „break“ завершает выполнение текущих операторов „while“, „for“ или „loop“:

```
for item in ["a", "b", "c", "d"] {
    if item == "c" {
        break; // exit the for
    }
    echo item, "\n";
}
```

Оператор continue

„continue“ используется внутри структур цикла, чтобы пропустить оставшуюся часть текущей итерации цикла и продолжить выполнение при оценке условия, а затем в начале следующей итерации.

```
let a = 5;
while a > 0 {
    let a--;
    if a == 3 {
        continue;
    }
    echo a, "\n";
}
```

1.12.3 Оператор require

Инструкция „require“ динамически включает и оценивает указанный PHP-файл. Обратите внимание, что файлы, включенные через Zephir, интерпретируются Zend Engine как обычные PHP-файлы. „Require“ не позволяет включать другие файлы Zephir во время выполнения.

```
if file_exists(path) {
    require path;
}
```

1.12.4 Оператор let

Оператор „let“ используется для изменения переменных, свойств и массивов. Переменные по умолчанию неизменяемые, и эта команда делает их изменяемыми:

```
let name = "Tony";           // simple variable
let this->name = "Tony";     // object property
let data["name"] = "Tony";   // array index
let self::_name = "Tony";    // static property
```

Также эта инструкция должна использоваться для увеличения/уменьшения переменных:

```
let number++;           // increment simple variable
let number--;           // decrement simple variable
let this->number++;     // increment object property
let this->number--;     // decrement object property
```

Множественные изменения могут быть выполнены в единственной операции „let“:

```
let price = 1.00, realPrice = price, status = false;
```

1.13 Вызов функций

PHP имеет богатую библиотеку функций, которые вы можете использовать в своих расширениях. Чтобы вызвать функцию PHP, вы просто используете ее как обычно в своем коде Zephir:

```
namespace MyLibrary;

class Encoder
{

    public function encode(var text)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    {
        if strlen(text) != 0 {
            return base64_encode(text);
        }
        return false;
    }
}
```

Вы также можете вызывать функции, которые, как ожидается, существуют в пользовательском окружении PHP, но не встроены в PHP:

```
namespace MyLibrary;

class Encoder
{

    public function encode(var text)
    {
        if strlen(text) != 0 {
            if function_exists("my_custom_encoder") {
                return my_custom_encoder(text);
            } else {
                return base64_encode(text);
            }
        }
        return false;
    }
}
```

Обратите внимание, что все функции PHP только получают и возвращают динамические переменные. Если вы передаете переменную статического типа в качестве параметра, то в качестве моста для вызова функции будет создана временная динамическая переменная:

```
namespace MyLibrary;

class Encoder
{

    public function encode(string text)
    {
        if strlen(text) != 0 {
            // Неявная динамическая переменная создается,
            // чтобы передать переменную статического типа
            // 'text' в качестве параметра
            return base64_encode(text);
        }
        return false;
    }
}
```

Аналогично, функции возвращают динамические значения, которые не могут быть напрямую назначены статическим переменным без соответствующего приведения:

```
namespace MyLibrary;

class Encoder
```

(continues on next page)

(продолжение с предыдущей страницы)

```
{

    public function encode(string text)
    {
        string encoded = "";

        if strlen(text) != 0 {
            let encoded = (string) base64_encode(text);
            return '(' . encoded . ')';
        }
        return false;
    }
}
```

Zephir предоставляет вам возможность динамически вызывать функции, такие как:

```
namespace MyLibrary;

class Encoder
{

    public function encode(var callback, string text)
    {
        return {callback}(text);
    }
}
```

1.14 Пользовательские оптимизаторы

В большинстве распространенных функций в Zephir используются внутренние оптимизаторы. „optimizer“ работает как перехватчик для вызовов функций. „optimizer“ заменяет вызов функции в пользовательском пространстве PHP прямыми C-вызовами, которые выполняются быстрее и имеют более низкие накладные расходы, повышающие производительность.

Чтобы создать оптимизатор, вам нужно создать класс в каталоге „optimizers“, необходимо использовать следующее соглашение:

Функция в Zephir	Название класса оптимизатора	Путь оптимизатора	Функция в C
calculate_pi	CalculatePiOptimizer	optimizers/CalculatePiOptimizer.php	my_calculate_pi

Это основная структура для „optimizer“:

```
<?php

class CalculatePiOptimizer extends OptimizerAbstract
{

    public function optimize(array $expression, Call $call, CompilationContext $context)
    {
        //...
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
}
```

Реализация оптимизаторов в большой степени зависит от типа кода, который вы хотите сгенерировать. В нашем примере мы собираемся заменить вызов этой функции вызовом c-функции. В Zephir код, используемый для вызова этой функции:

```
let pi = calculate_pi(1000);
```

Таким образом, оптимизатор будет ожидать только один параметр, мы должны подтвердить это, чтобы избежать проблем позже:

```
<?php

public function optimize(array $expression, Call $call, CompilationContext $context)
{

    if (!isset($expression['parameters'])) {
        throw new CompilerException("'calculate_pi' requires one parameter");
    }

    if (count($expression['parameters']) < 2) {
        throw new CompilerException("'calculate_pi' requires one parameter");
    }

    //...
}
```

Есть только что вызванные функции и они не возвращают никакого значения, наша функция возвращает значение, которое является вычисленным значением PI. Поэтому мы должны знать, что тип переменной, используемой для получения этого вычисленного значения, - OK:

```
<?php

public function optimize(array $expression, Call $call, CompilationContext $context)
{

    if (!isset($expression['parameters'])) {
        throw new CompilerException("'calculate_pi' requires one parameter");
    }

    if (count($expression['parameters']) < 2) {
        throw new CompilerException("'calculate_pi' requires one parameter");
    }

    /**
     * Обработка возвращаемого символа
     */
    $call->processExpectedReturn($context);

    $symbolVariable = $call->getSymbolVariable();
    if ($symbolVariable->isNotDouble()) {
        throw new CompilerException("Calculated PI values only can be stored in double variables",
↪$expression);
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    //...
}
```

Мы проверяем, будет ли возвращаемое значение храниться в типе переменной „double“, если не выбрано исключение компилятора.

Следующее, что нам нужно сделать, это обработать параметры, переданные функции:

```
<?php

$resolvedParams = $call->getReadOnlyResolvedParams($expression['parameters'], $context,
↪$expression);
```

Как хорошая практика в Zephir важна для создания функций, которые не изменяют их параметры, если вы изменяете переданные параметры, Zephir нужно будет распределить память для переданных констант, и вам придется использовать getResolvedParams вместо getReadOnlyResolvedParams.

Код, возвращаемый этими методами, является допустимым C-кодом, который может использоваться в принтере кода для генерации вызова c-функции:

```
<?php

//Generate the C-code
return new CompiledExpression('double', 'calculate_pi( ' . $resolvedParams[0] . ')', $expression);
```

Все оптимизаторы должны возвращать экземпляр CompiledExpression, это сообщит компилятору тип, возвращаемый кодом и связанным с ним C-кодом.

Полный код оптимизатора:

```
<?php

class CalculatePiOptimizer extends OptimizerAbstract
{

    public function optimize(array $expression, Call $call, CompilationContext $context)
    {

        if (!isset($expression['parameters'])) {
            throw new CompilerException("'calculate_pi' requires one parameter");
        }

        if (count($expression['parameters']) < 2) {
            throw new CompilerException("'calculate_pi' requires one parameter");
        }

        /**
         * Обработка возвращаемого символа
         */
        $call->processExpectedReturn($context);

        $symbolVariable = $call->getSymbolVariable();
        if ($symbolVariable->isNotDouble()) {
            throw new CompilerException("Calculated PI values only can be stored in double␣
↪variables", $expression);
        }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        $resolvedParams = $call->getReadOnlyResolvedParams($expression['parameters'], $context,
↪$expression);

        return new CompiledExpression('double', 'my_calculate_pi( ' . $resolvedParams[0] . ')',
↪$expression);
    }

}
```

1.15 Конфигурационный файл

Каждое расширение Zephir имеет файл конфигурации, называемый config.json. Этот файл читается Zephir каждый раз, когда вы создаете или создаете расширение, и это позволяет разработчику изменять расширение или поведение компилятора.

Этот файл использует формат JSON в качестве формата конфигурации:

```
{
    "namespace": "test",
    "name": "Test Extension",
    "description": "My amazing extension",
    "author": "Tony Hawk",
    "version": "1.2.0"
}
```

Параметры, определенные в этом файле, переопределяют любые заводские настройки, предоставляемые Zephir.

Поддерживаются следующие параметры:

1.15.1 namespace

Пространство имен расширения - это должен быть простой идентификатор, соответствующий регулярному выражению: [a-zA-Z0-9_]+:

```
{
    "namespace": "test"
}
```

1.15.2 name

Имя расширения может содержать только символы ascii:

```
{
    "namespace": "test"
}
```

1.15.3 description

Расширение описание, любой текст, описывающий расширение:

```
{
    "description": "My amazing extension"
}
```

1.15.4 author

Компания, разработчик, учреждение и т.д., Которые разработали расширение:

```
{
    "author": "Tony Hawk"
}
```

1.15.5 version

Версия расширения должна следовать регулярному выражению: [0-9]+.[0-9]+.[0-9]+:

```
{
    "version": "1.2.0"
}
```

1.15.6 warnings

Предупреждения компилятора включены или отключены в текущем проекте:

```
{
    "warnings": {
        "unused-variable": true,
        "unused-variable-external": false,
        "possible-wrong-parameter": true,
        "possible-wrong-parameter-undefined": false,
        "nonexistent-function": true,
        "nonexistent-class": true
    }
}
```

1.15.7 optimizations

Оптимизация компилятора включена или отключена в текущем проекте:

```
{
    "optimizations": {
        "static-type-inference": true,
        "static-type-inference-second-pass": true,
        "local-context-pass": false
    }
}
```

1.15.8 globals

Доступные для расширения переменные globals. Для получения дополнительной информации см. Главу *extension globals*.

```
{
    "globals": {
        "my_setting_1": {
            "type": "bool",
            "default": true
        },
        "my_setting_2": {
            "type": "int",
            "default": 10
        }
    }
}
```

1.15.9 info

phpinfo() Для получения дополнительной информации см. Главу *phpinfo()*.

```
{
    "info": [
        {
            "header": ["Directive", "Value"],
            "rows": [
                ["setting1", "value1"],
                ["setting2", "value2"]
            ]
        }
    ]
}
```

1.15.10 extra-cflags

Любые дополнительные флаги, которые вы хотите добавить в процесс компиляции:

```
{
    "extra-cflags": "-I/usr/local/Cellar/libevent/2.0.21_1/include"
}
```

1.15.11 extra-libs

Любые дополнительные библиотеки, которые вы хотите добавить в процесс компиляции:

```
{
    "extra-libs": "-L/usr/local/Cellar/libevent/2.0.21_1/lib -levent"
}
```

1.15.12 package-dependencies

Объявление библиотечных зависимостей (проверка версии по `pkg-config`)

```
{
    "package-dependencies": {
        "openssl": "*",
        "libpng": ">= 0.1.0",
        "protobuf": "<= 2.6.1"
    }
}
```

Оператор версии поддерживает `=`, `>=`, `<=`, and `*`

1.16 Глобальные параметры рассширения

Расширения PHP предоставляют способ определения глобальных переменных внутри расширения. Чтение/запись глобальных данных должны быть быстрее, чем любые другие глобальные механизмы (например, статические члены). Глобальные значения расширений можно использовать для настройки параметров конфигурации, которые изменяют поведение вашей библиотеки.

В Zephir расширения globals ограничены простыми скалярными типами типа int/bool/double/char и т.д. Здесь не допускаются сложные типы, такие как строки/массивы/объекты/ресурсы.

Вы можете включить глобальные расширения, добавив следующую структуру в config.json:

```
{
    //...
    "globals": {
        "allow_some_feature": {
            "type": "bool",
            "default": true
        },
        "number_times": {
            "type": "int",
            "default": 10
        },
        "some_component.my_setting_1": {
            "type": "bool",
            "default": true
        },
        "some_component.my_setting_2": {
            "type": "int",
            "default": 100
        }
    }
}
```

Каждая глобальная секция имеет следующую структуру:

```
"<global-name>": {
    "type": "<some-valid-type>",
    "default": <some-compatible-default-value>
}
```

Составные глобальные переменные имеют следующую структуру:

```
"<namespace>.<global-name>": {
    "type": "<some-valid-type>",
    "default": <some-compatible-default-value>
}
```

Внутри любого метода вы можете читать/писать глобальные расширения с помощью встроенных функций globals_get/globals_set:

```
globals_set("allow_some_feature", true);
let someFeature = globals_get("allow_some_feature");
```

Если вы хотите изменить эти глобальные переменные из PHP, хорошим вариантом является включение метода, направленного на это:

```
namespace Test;

class MyOptions
{

    public static function setOptions(array options)
    {
        boolean someOption, anotherOption;

        if fetch someOption, options["some_option"] {
            globals_set("some_option", someOption);
        }

        if fetch anotherOption, options["another_option"] {
            globals_set("another_option", anotherOption);
        }
    }
}
```

Расширения globals не могут быть динамически доступны, так как C-код, сгенерированный оптимизаторами globals_get / globals_set, должен быть разрешен во время компиляции:

```
let myOption = "someOption";

// Будет генерировать исключение компилятора
let someOption = globals_get(myOption);
```

1.17 Phpinfo() секции

Как и большинство расширений, Zephir расширения могут отображать информацию при выводе phpinfo(). Обычно эта информация относится к директивам, данным окружения и т.п.

По умолчанию каждое Zephir расширение добавляет базовую таблицу в вывод phpinfo() отображающую версию расширения.

Вы можете добавить больше директив добавив следующую конфигурацию в файл config.json:

```
"info": [
    {
        "header": ["Directive", "Value"],
        "rows": [
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            ["setting1", "value1"],
            ["setting2", "value2"]
        ]
    },
    {
        "header": ["Directive", "Value"],
        "rows": [
            ["setting3", "value3"],
            ["setting4", "value4"]
        ]
    }
]
```

Эта информация будет отображена следующим образом:

test

Test Extension	enabled
Version	1.0.0

Directive	Value
setting1	value1
setting2	value2

Directive	Value
setting3	value3
setting4	value4

1.18 License

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Глава 2

Другие форматы

- PDF

Алфавитный указатель

I