PHPInside
электронный журнал для веб-разработчиков
Ноябрь '04 № 8

PHPInside Art Contest

Результаты конкурса (часть 1)

Содержание

Обратная связь

Вступление:
Андрей Олищук

Номер который сейчас лежит перед вами является в своем роде экспериментом, на который мы пошли, чтобы сделать журнал более интересным и насыщенным информацией. Здесь мы опробуем новую рубрику «Форум», в которой соберем реальные вопросы и ответы из реальных форумов о PHP. «Донорами» этой рубрики стали два сообщества: http://phpclub.ru и http://php.com.ua.

Также, в этом номере мы предлагаем вашему вниманию врезки с общим заголовком «Тем временем...», которые в сжатом виде сообщат полезную информацию и насытят журнал дополнительным содержанием. Необходимо заметить, что темы этих врезок не связаны с темами статей, на полях которых они публикуются.

Особенностью данного номера является его содержимое – статьи, которые победили в проведенном конкурсе «PHP Inside Art Contest». Всего в нем участвовали 22 материала и жюри определило трех победителей, которые заслужили призы от http://www.arbatek.ru, http://linuxcenter.ru, http://weblancer.net. Сейчас мы публикуем эти три статьи (в разделе «В фокусе») и статью Антона Калмыкова, которая не оценивалась членами жюри в силу своей специфики не укладывающейся в концепцию конкурса, но получила очень хорошие отклики.

Конечно, конкурс принес немало хороших и нужных статей, однако мы решили не публиковать их все сразу, а разместить в нескольких номерах и смешать с другими материалами, поэтому надеемся порадовать вас еще и в следующем номере. А пока, пишите нам свои отзывы и участвуйте в выпуске новых номеров, тем более, что теперь мы публикуем не только большие статьи, но и краткие мини-статьи и советы. Спасибо за внимание!

Команда номера

Авторы и переводчики
Владислав Зараковский
Артем Сидоренко
Виктор Кулаков
Данил Миронов
Антон Калмыков
Александр Бутюгин [BArS]

Редакционная коллегия
Александр Смирнов
Александр Войцеховский
Андрей Олищук [nw]
Антон Чаплыгин
Дмитрий Попов
Елена Тесля

Выпуск номера
Андрей Олищук [nw]
Антон Чаплыгин
Денис Зенькович

Контактные данные
http://phpinside.ru
nw@phpinside.net

В фокусе

Ускоряемся вместе с Propel

Автор:
Владислав Зараковский

Новую, пятую версию PHP компания Zend позиционирует как платформу для приложений уровня предприятия. В связи с этим возросла необходимость в средствах ускоряющих разработку приложений. Об одном из таких средств, представляющих собой ORM (Object Relational Mapping) для PHP и пойдет речь в этой статье.

В последнее время наблюдается тенденция переноса на PHP проектов, изначально созданных на других языках (обычно Java). С выходом пятой версии PHP этот процесс упростился, поскольку в языке наконец-то реализованы отсутствовавшие ранее возможности: обработка исключений и полноценная объектная модель.

Хотелось бы отметить несколько проектов, родившихся в сообществе разработчиков на языке Java и перенесенных в PHP. Например, Phing – система сборки проектов на основе конфигурационных файлов в формате XML (основан на Apache Ant). PHPUnit – среда для выполнения модульных тестов (реализует возможности JUnit и JUni-t-our**)**. В этой статье мы с вами рассмотрим Propel - еще один проект, перенесенный из Java в PHP и позволяющий упростить разработку приложений.

Введение

Propel представляет собой реализацию слоя Object Persistence для PHP5. Подобные средства также называют ORM (Object Relational Mapping) или DAO (Data Access Objects).

Разработчики Propel решили «не изобретать велосипед» и взяли за основу проверенный временем проект Apache Torque (ORM для Java). Конечно же, существуют и другие реализации ORM на PHP с различными возможностями. Вот некоторые из них:

- EasyORM
- PEAR::DB_DataObject
- Metastorage

Сами разработчики объясняют причину создания Propel недостатками присущими другим решениям. Так, например EasyORM жестко привязан к MySQL.

PEAR::DB_DataObject использует для абстрактного доступа к базе данных PEAR::DB, не имеющий средств для получения схемы данных по уже существующей БД (так называемый reverse-engineering). Использование ini-файлов для описания структуры данных затрудняет его интеграцию с другими технологиями. Кроме того, он вынуждает использовать обработку ошибок в стиле PEAR, что мягко говоря странно в PHP5 с его встроенной поддержкой обработки исключений.

Самое, пожалуй, развитое решение, Metastorage требует глубокого изучения своего собственного, основанного на XML языка MetaL со своим циклом обучения, и не особенно упрощает некоторые вещи наподобие запросов по внешним ключам.

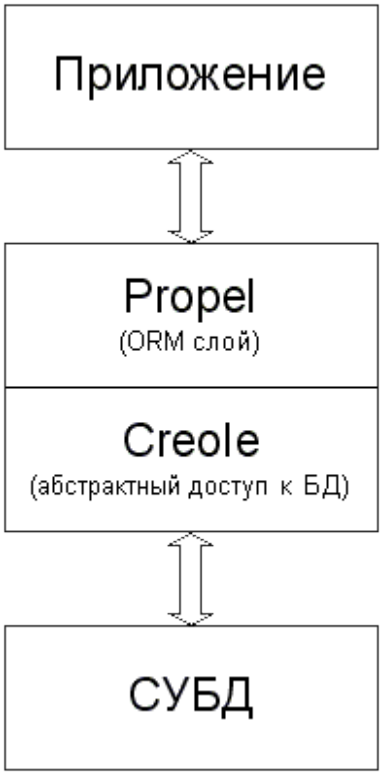
Отбросим в сторону рассуждения о том, какое из решений лучше. Выбор каждый сможет сделать сам, в зависимости от конкретных требований. Давайте познакомимся поближе с Propel.

Структура Propel

Propel состоит из двух компонентов:

- Генератора классов и SQL-файлов (каталог *generator*)
- Среды исполнения необходимой для использования сгенерированных классов (каталог *runtime*).

Приложение, использующее Propel можно представить как на врезке справа.



Как мы видим, Propel на самом деле состоит из двух слоев, точнее он использует второй слой (Creole) для абстрактного доступа к БД.

- **Creole** – обеспечивает абстрактный доступ к различным СУБД.
- **Propel** – выполняет отображение реляционной базы данных в объекты.

Далее в статье вы увидите, что Creole, как и Propel, также состоит из двух частей. Одна часть (*creole*) представляет собой базовый API, а вторая (*jargon*) предоставляет расширенные возможности.

Достоинства

- **Расширяемость.** Propel использует схему базы данных в формате XML, что позволяет легко интегрировать его с другими средствами, основанными на этом структурном языке. Например, с генератором форм на основе схемы данных.
- **Адаптируемость.** При генерации кода Propel создает базовые классы для доступа к данным и пустые подклассы (заглушки). Именно в эти классы-заглушки вы будете вносить новую функциональность. Если модель данных изменится, то перегенерация базовых классов не затронет напрямую ваш собственный код внутри классов-заглушек (хотя конечно вам придется исправить код, если он не соответствует новой модели данных).
- **Независимость от СУБД.** Можете использовать любую из поддерживаемых СУБД. Если в ней отсутствует какая-либо возможность, Propel будет ее эмулировать. В Propel также имеются средства для миграции. С помощью этих средств вы сможете в любой момент (!) перенести свое приложение с одной СУБД на другую.

- **Простота изучения.** Модель данных Propel интуитивно понятна, поскольку соответствует структуре БД. Сущности в Propel – таблицы, и одна таблица отображается в один класс.

Тем временем...

На платформе Windows, PHP может работать с приложениями через COM. Вот простой пример работы с MS Word.

```
<?php
$word = new COM("word.application")
print "Loaded Word, version
{$word->Version}\n";
$word->Visible = 0;
$word->Documents->Add();
$word->Selection->TypeText("Testing,
testing… 1,2,3");
$word->Documents[1]->SaveAs
("c:\some_tst.doc");
$word->Quit();     ?>
```

Пример использования

Propel крайне прост в использовании. Стандартная последовательность шагов при разработке нового проекта такова:

- Вы описываете структуру базы данных в формате XML (например, в редакторе XML Spy или, в перспективе, в визуальном редакторе наподобие DBDesigner4);
- Propel генерирует классы для доступа к данным, SQL-файлы для создания структуры БД и создает саму базу с помощью этих файлов;
- Вы используете полученные классы.

Описываем модель данных

Модель данных Propel простая и наглядная. Это позволяет быстро освоить описание структуры БД и положительно сказывается на производительности приложений. Для описания схемы данных используется язык XML. Рассмотрим простейший пример:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE database SYSTEM "../dtd/database.dtd">
<database name="bookstore">
  <table name="book">
    <column name="book_id"
      type="INTEGER"
      required="true"
      primaryKey="true"/>
    <column name="title"
      type="VARCHAR"
      size="50"
      required="true" />
  </table>
</database>
```

В этом примере описана база данных *bookstore* с единственной таблицей *book*. В таблице *book* имеются два поля:

- **book_id** – уникальный номер книги (INTEGER, обязательное, т.е. не NULL, первичный ключ)
- **title** – название книги (VARCHAR, максимальная длина 50 символов, обязательное)

Файл с описанием типа документа (*database.dtd*) позволяет выполнять проверку правильности созданной схемы любым XML-процессором. Как видим, ничего сложного тут нет. После описания схемы данных можно приступать к генерации кода.

Генерируем код

Для начала сделаем некоторые настройки. Генератору кода кроме схемы данных потребуются два файла:

- build.properties
- runtime-conf.xml

В первом вы должны указать настройки для вашего проекта, необходимые для генерации кода. Во втором – настройки, которые будут использоваться во время исполнения.

Вот как выглядит файл *build.properties* идущий в составе Propel вместе с примером *bookstore* (каталог *generator/projects/bookstore*):

```
# имя проекта
propel.project = bookstore

# используемая база данных
propel.database = sqlite

# URL для подключения к БД
propel.database.url = sqlite://localhost/./test/bookstore.db

# URL для создания БД (не должен включать имя БД)
# не требуется для SQLite, поскольку БД автоматически создается,
# если она не существует при обращении к ней
# propel.database.createUrl =

# имя пакета
# для пакета com.company.project.om классы будут сгенерированы в каталоге
# generator\projects\bookstore\build\classes\com\company\project\om
propel.targetPackage = bookstore
```

Файл *runtime-conf.xml* также можно найти в каталоге *bookstore*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
  <log>
    <ident>propel-bookstore</ident>
    <level>7</level>
  </log>
  <propel>
    <datasources default="bookstore">
      <datasource id="bookstore">
        <adapter>sqlite</adapter>
        <connection>
          <phptype>sqlite</phptype>
          <hostspec>localhost</hostspec>
          <database>./test/bookstore.db</database>
          <username></username>
          <password></password>
        </connection>
      </datasource>
    </datasources>
  </propel>
</config>
```

В элементе <*log*> задаются параметры для записи событий в журнал: строка-идентификатор <*ident*> и уровень отладочной информации <*level*>. Параметры самого Propel задаются внутри тэга <*propel*>. Допускается использовать для одного проекта несколько источников данных, каждый из которых описывается в элементе <*datasource*>. Используемый драйвер задан с помощью <*adapter*>, а информация о подключении в виде DSN указывается в элементе <*connection*>. Названия элементов говорят сами за себя, так что сделать нужные настройки не составит труда.

Со структурой файла разобрались, теперь из каталога *generator* запускаем процесс генерации кода:

```
phing -Dproject=bookstore
```

В случае успешного окончания процесса для приведенной выше схемы данных мы получим целый набор классов (для полной схемы из примера *bookstore* их будет гораздо больше):

- **BaseBook** – представляет строку из таблицы *book* (любители паттернов узнают *Row Data Gateway*)
- **BaseBookPeer** – содержит набор статических методов для выполнения запросов к таблице *book* (паттерн *Table Data Gateway*)
- **BookMap** – хранит информацию о структуре таблицы *book* (какие поля являются первичными ключами, внешними ключами и т.п.)

Кроме того, для классов *BaseBook* и *BaseBookPeer* создаются пустые подклассы *Book* и *BookPeer* соответственно, которые вы будете подключать, и использовать у себя в приложении. При необходимости расширить функциональность, вы должны вносить все изменения именно в них. Это позволит впоследствии сгенерировать заново классы *BaseBook* и *BaseBookPeer* при изменениях в модели данных, не затронув ваш собственный код внутри *Book* и *BookPeer*.

Пользуемся классами

Использовать сгенерированные классы в ваших программах очень просто:

```
<?php
// Инициализируем Propel
// Указываем путь к файлу с runtime-настройками нашего проекта.
Propel::init('bookstore-conf.php');

// подключаем нужные в этом файле классы
include_once 'bookstore\Book.php';
include_once 'bookstore\BookPeer.php';

// далее следует ваш код
?>
```

На этом закончим краткий обзор использования Propel и перейдем к конкретным примерам. Но сначала, конечно же, нужно установить Propel.

Установка

Для установки Propel потребуются

- PHP версии не ниже 5.0.0 (http://php.net)

  - Обязательно должна присутствовать command-line версия PHP для генерации классов и SQL-файлов с помощью Phing

  - Для работы Phing также требуется, чтобы PHP был скомпилирован с поддержкой libxml2 и xsl

- Любая поддерживаемая Propel СУБД. На сегодня это MySQL, MS SQL Server, PostgresSQL, SQLite (поддерживается в PHP5 по умолчанию) и Oracle, а также идет работа над драйвером MySQLi (для MySQL версии 4.1 и выше).
- Phing 2.0 (http://phing.info)
- Creole версии не ниже 1.0.0 (http://creole.phpdb.org)
- Пакет PEAR::Log (http://pear.php.net/package/Log)

Будем считать, что PHP, PEAR, PEAR_Log и какая-нибудь СУБД у вас уже установлены и сконфигурированы. Рассмотрим установку Propel.

Сначала нужно установить Phing и Creole. Проще всего это сделать, используя дистрибутивы, предназначенные для установки с помощью инсталлятора пакетов PEAR. Если вы скачали дистрибутивы Phing и Creole себе на диск, то их можно установить так:

```
pear install phing-2.0.0-pear.tgz

pear install creole-1.0.0-pear.tgz
pear install jargon-1.0.0-pear.tgz
```

Последние 2 строки установят Creole, который, как вы видите, состоит из двух отдельных архивов. Первый обеспечивает базовую функциональность для абстрактного доступа к данным, а второй – расширенные возможности.

Кроме того, вы можете запустить установку прямо из Интернета:

```
pear install http://phing.info/pear/phing-current.tgz
pear install http://creole.phpdb.org/pear/creole-current.tgz
pear install http://creole.phpdb.org/pear/jargon-current.tgz
```

Теперь скачиваем с сайта http://propel.phpdb.org дистрибутив Propel и распаковываем его в какой-нибудь каталог, например *C:\PHP\propel*.

После чего к переменной *include_path* в файле *php.ini* нужно добавить путь к runtime-классам Propel:

```
include_path=".;C:\PHP\PEAR;C:\PHP\propel\runtime\classes";
```

Для правильной работы Propel также потребуется установить значения нескольких переменных в файле *php.ini*

```
ze1_compatibility_mode = off
magic_quotes_gpc       = off
magic_quotes_sybase    = off
```

Ну и наконец найдите файл *build.properties-sample* в каталоге *C:\PHP\propel\generator* и создайте его копию с именем *build.properties*. Вот и все. Propel готов к использованию.

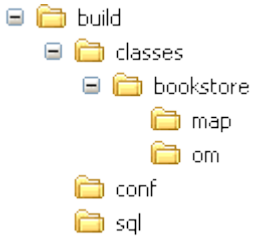
Пример базы данных

Вместе с Propel поставляется схема базы данных *bookstore* (*generator\projects\bookstore\schema.xml*). Все дальнейшие примеры в статье будут использовать эту базу, поэтому для начала сгенерируем для нее код. В дальнейших объяснениях пути указаны относительно корневого каталога Propel.

Зайдите в каталог *generator* и выполните следующую команду:

```
phing -Dproject=bookstore
```

Если все прошло успешно, то в каталоге *generator\projects\bookstore\build* появится все необходимое для работы с базой *bookstore*. Рассмотрим структуру каталога *build* (на врезке справа).



В каталоге *classes\bookstore* содержатся классы, которые вы будете подключать в свои проекты. В подкаталогах *map* и *om* содержатся базовые классы, сгенерированные Propel.

В каталоге *conf* вы найдете файл *bookstore-conf.php*, сгенерированный на основе *runtime-conf.xml*, который не используется во время выполнения. Для повышения производительности вместо него к проекту подключается *bookstore-conf.php* с минимальными затратами времени.

В каталоге *sql* содержится файл *schema.sql*, предназначенный для создания базы данных. Этим мы сейчас и займемся. Для создания базы используйте:

```
#phing -Dproject=bookstore -Dtarget=create-db
phing -Dproject=bookstore -Dtarget=insert-sql
```

Первая строка создает пустую базу и закомментирована, поскольку не нужна для SQLite (эта СУБД указана по умолчанию в файле *build.properties*) и выполнится с ошибками. Однако в остальных случаях ее нужно будет выполнить. Вторая строка создает структуру базы данных с помощью файла *schema.sql*. Конечно, вы можете создать базу и импортировать в нее файл *schema.sql* средствами самой СУБД, но так значительно проще.

Итак, для нашего примера будет создана база данных bookstore (файл *generator\test\bookstore.db*). Теперь, перед началом работы с базой можно проверить, что все работает правильно. Специально для этого предназначен тест *bookstore-test.php* в каталоге *generator\test*. Для запуска теста выполните следующую команду:

```
php -qC test\bookstore-test.php
```

Перед выполнением этой команды в базе данных не должно быть никаких записей, иначе вы увидите сообщения об ошибках. Если все тесты прошли с отметкой [OK], значит все в порядке, и можно приступать непосредственно к работе с базой.

Хранение объектов

Вставка (INSERT)

Добавление объектов в базу сводится к созданию объекта соответствующего класса и вызову метода save().

```
// создаем объект класса Author
$a = new Author();
$a->setFirstName("Leo");
$a->setLastName("Tolstoy");

// сохраняем объект (добавляет 1 строку в таблицу author)
$a->save();

// теперь известно значение ключевого поля
$author_id = $a->getAuthorId();
```

В приведенном примере вызов метода save() приводит к записи в базу одной строки. Однако для связанных таблиц вызов метода save() может приводить к созданию нескольких записей. Подобное поведение называется каскадными вставками.

```
// автор книги
$author = new Author();
$author->setFirstName("John");
$author->setLastName("Smith");
// издатель книги
$publisher = new Publisher();
$publisher->setName("Penguin");
// конкретная книга
$book = new Book();
$book->setISBN("111-2222-3333");
$book->setTitle("Fake Title");
// связываем книгу с ее автором и издателем
$book->setPublisher($publisher);
$book->setAuthor($author);
// сохранение book приводит также к сохранению объектов author и
publisher!
$book->save(); // добавляет 3 записи в базу
```

Последняя строка в этом примере вставляет в базу 3 записи, т.к. автоматически сохраняются связанные объекты $author и $publisher. Очень удобно, не так ли?

Обновление (UPDATE)

Для обновления объектов вы читаете объект из базы, меняете некоторые его поля, затем вызываете метод save().

```
// получаем запись по первичному ключу
$obj = AuthorPeer::retrieveByPK(1);
// обновляем поле
$obj->setFirstName($obj->getFirstName() . "-modiified");
// сохраняем измененный объект
$obj->save();

// вызов save() приводит к выполнению примерно такого запроса:
// UPDATE author
//    SET author.FIRST_NAME = '".$obj->getFirstName()."-modified'
//    WHERE author.AUTHOR_ID = 1
```

Удаление (DELETE)

Для удаления вы можете использовать метод delete() конкретного объекта или метод doDelete() Peer-класса. Метод delete() удаляет объект по первичному ключу и устанавливает объект в состояние “delete”. После этого вы все еще можете читать поля такого объекта, но не сможете его сохранить.

```
$author = AuthorPeer::retrieveByPk(1);
$author->delete();
// соответствующий запрос:
// DELETE FROM author WHERE author.AUTHOR_ID = 1
```

При использовании метода Peer-класса doDelete() вы должны указать критерий, по которому необходимо производить удаление, так что за один раз можно удалить сразу несколько строк из таблицы.

```
$crit = new Criteria();
$crit->add(AuthorPeer::FIRST_NAME, 'A%', Criteria::LIKE);
AuthorPeer::doDelete($crit);
// соответствующий запрос:
// DELETE FROM author WHERE author.FIRST_NAME LIKE 'A%'
```

В этом примере удаляются все записи из таблицы author, в которых имя автора начинается с буквы ‘A’.

Запомните, что кроме каскадных вставок существуют также каскадные удаления. Для того чтобы они заработали достаточно указать атрибут onDelete=”cascade” для тега <foreign-key> в схеме данных. Все остальное Propel сделает сам, как и в случае с каскадными вставками.

Поиск объектов

Поиск по первичному ключу

Проще всего найти объект, если известен его первичный ключ:

```
// в случае таблицы book с первичным ключом по одной колонке book_id
$book = BookPeer::retrieveByPK(1);
```

Если первичный ключ состоит из нескольких частей, то все ненамного сложнее. В этом случае вместо единственного значения в функцию retrieveByPK() нужно передать массив.

```
// в таблице book_author_xref (нет в примере bookstore)
// первичный ключ состоит из book_id и author_id

// последовательность ключей в массиве должна соответствовать
// последовательности определения полей в файле schema.xml
$obj = BookAuthorXrefPeer::retrieveByPK(array(1,2));
```

Поиск по критерию

Для поиска записей, удовлетворяющих некоторому критерию нужно использовать объект Criteria совместно с методом Peer-класса doSelect(). С помощью объекта Criteria вы сможете сконструировать практически любой запрос к базе данных. Метод doSelect() возвратит вам массив объектов соответствующего типа.

```
$c = new Criteria();
$c->add(AuthorPeer::FIRST_NAME, "Karl");
$c->add(AuthorPeer::LAST_NAME, "Marx", Criteria::NOT_EQUAL);

$results = AuthorPeer::doSelect($c);
foreach($results as $author) {
    print $author->getLastName() . " " . $author->getFirstName() . "\n";
}
```

В этом примере находятся все авторы, которых зовут Karl но фамилия которых не Marx.

```
// Поиск авторов, которых зовут "Leo" ИЛИ фамилия которых
// "Tolstoy", "Dostoevsky", или "Bakhtin"
$c = new Criteria();
$cton1 = $c->getNewCriterion(AuthorPeer::FIRST_NAME, "Leo");
$cton2 = $c->getNewCriterion(AuthorPeer::LAST_NAME,
                array("Tolstoy", "Dostoevsky", "Bakhtin"), Criteria::IN);
// комбинируем критерии по OR
$cton1->addOr($cton2);
// а дальше как обычно
$c->add($cton1);
```

Комбинируем критерии

Напрямую комбинировать объекты Criteria нельзя, вместо этого нужно использовать объекты Criterion.

Несколько условий для одного поля

Propel хранит критерии в хеш-таблице для оптимизации производительности, поэтому задать несколько условий для одного поля несколько труднее.

Это неправильный вариант использования:

```
$c = new Criteria();
$c->add(AuthorPeer::FIRST_NAME, "Leo%", Criteria::LIKE);
$c->add(AuthorPeer::FIRST_NAME, "Leonardo", Criteria::NOT_EQUAL);
```

Второй вызов add() просто переопределит условие, добавленное первым add() поэтому найдутся только те записи, где имя не равно "Leonardo". Вот правильный способ задать несколько условий для поля:

```
$c = new Criteria();
$criterion = $c->getNewCriterion(AuthorPeer::FIRST_NAME, "Leo%",
Criteria::LIKE);
$criterion->addOr($c->getNewCriterion(AuthorPeer::FIRST_NAME, "Leonardo",
Criteria::NOT_EQUAL));
$c->add($criterion);
```

Другие аспекты запроса

Класс Criteria позволяет также управлять числом возвращаемых записей, чувствительностью к регистру, сортировкой записей и т.д.

```
$c = new Criteria();
$c->add(AuthorPeer::FIRST_NAME, "max");
$c->setLimit(10);
$c->addAscendingOrderByColumn(AuthorPeer::LAST_NAME);
$c->setIgnoreCase(true);
```

В этом примере создается объект Criteria для поиска не более 10 авторов с именем “max” (регистро-независимый поиск) отсортированных по фамилии.

Расширенные возможности

Объект Criteria может также использоваться для написания кода, который вместо создания объектов будет возвращать только необходимую информацию, например, в массиве. Следующий код возвращает список ISBN кодов по заданному критерию:

```
class BookPeer extends BaseBookPeer {
    function getJustISBN(Criteria $c) {
         $c->clearSelectColumns()->addSelectColumn(ISBN);
         $rs = BasePeer::doSelect($c);
         $isbns = array();
         while($rs->next()) {
             $isbns[] = $rs->get(0);
        }
        return $isbns;
    }
}
```

Использование SQL

Класс Criteria подходит для создания большинства запросов, но не претендует на возможность создания абсолютно любого запроса. Для построения очень сложных запросов или повышения производительности иногда лучше использовать SQL. Рекомендуется размещать код, использующий SQL в заглушках базовых Peer-классов.

```
class BookPeer extends BaseBookPeer {
    function getUnreviewedBooks() {
        $con = Propel::getConnection(DATABASE_NAME);
     // используйте CROSS JOIN, если БД не поддерживает вложенные запросы
        $sql = "SELECT books.* FROM books WHERE ".
        "NOT EXISTS (SELECT id FROM review WHERE book_id = book.id)";
        $stmt = $con->createStatement();
        $rs = $stmt->executeQuery($sql, ResultSet::FETCHMODE_NUM);

        return parent::populateObjects($rs);
    }
}
```

Приведенный выше код возвращает список книг без обзора. В этом примере есть одна проблема. Что будет, если поменяются имена полей в базе данных? Код перестанет работать! К счастью подобную зависимость легко устранить. Для этого в Peer-классах имеются специальные константы, хранящие имя таблицы и имена полей. С использованием этих констант запрос из предыдущего примера примет такой вид:

```
$sql = "SELECT b.* FROM ".BookPeer::TABLE_NAME." b WHERE ".
       "NOT EXISTS (SELECT r.".ReviewPeer::ID.
       " FROM ".ReviewPeer::TABLE_NAME." r ".
       " WHERE r.".ReviewPeer::BOOK_ID." = b.".BookPeer::ID.")";
```

Конечно, запрос стал менее читабельный, но теперь он не зависит от названия полей в таблице. При этом, однако, все еще сохраняется зависимость от последовательности полей в XML схеме, для устранения которой можно использовать Peer::addSelectColumns (Criteria) и Criteria->getSelectColumns().

Выборка больших объемов данных

В некоторых случаях нежелательно возвращать результат запроса в виде массива объектов. Например, обработка больших запросов подобным образом требует много памяти.

Гораздо лучше с помощью итератора перемещаться по результату запроса и работать с отдельным объектом, используя метод hydrate().

```
// выборка всех записей из таблицы book
$rs = BookPeer::doSelectRS(new Criteria());

// используем методы Creole для перемещения по результату запроса
while($rs->next()) {
    $book = new Book();
    $book->hydrate($rs);

    // далее что-нибудь делаем с очередным объектом book
}
```

Типы полей Propel

Типы полей Propel напрямую отображаются в типы полей Creole. Типы полей Creole в свою очередь представляют собой упрощенную версию типов JDBC. Тип поля задается в атрибуте *type* тэга *<column>* в схеме данных. В зависимости от возможностей конкретной СУБД типы полей Propel могут отображаться в различные типы.

Например, если в СУБД отсутствует встроенная поддержка булевых типов, то может использоваться поле BIT, INTEGER или SMALLINT. В таблице 1 приведены все типы, поддерживаемые Propel.

Тип поля	**Описание**
BOOLEAN	Булево значение (TRUE/FALSE)

TINYINT, INTEGER, BIGINT, DOUBLE, DECIMAL, FLOAT, REAL	Числовое значение. Можно использовать атрибуты *size* и *scale* для задания числа десятичных знаков до запятой и после
CHAR, VARCHAR, LONGVARCHAR	Строковое значение. Для задания размеров поля используется атрибут *size* (игнорируется для типа LONGVARCHAR). Все строковые значения возвращаются с удаленными лидирующими и конечными пробелами.
CLOB, BLOB	LOB предназначены для хранения больших объектов в базе. Результат запроса для этих типов возвращается в виде объектов *creole.util.Clob* или *creole.util.Blob*
DATE, TIME, TIMESTAMP	Представляет дату и время

Табл. 1. Типы данных в Propel.

Допустим в схеме данных (*schema.xml*) генератор встречает такую запись:

```
<column name="title" required="true" type="VARCHAR" size="255"/>
```

Тогда в сгенерированном SQL-файле появится запись, создающая поле title типа VARCHAR с максимальной длиной 255 символов.

Нюансы работы с LOB, DATE, TIME, TIMESTAMP

Рассмотрим пример использования поля типа BLOB. Создадим простую таблицу для хранения фотографий:

```
<table name="lob_test">
  <column name="photo" type="BLOB"/>
</table>
```

Посмотрим, как работать с таким полем.

```
// выбираем одну строку из таблицы
$lob = LobTestPeer::doSelectOne(new Criteria());

// запишем содержимое BLOB-поля в переменную
// так не получится:
// $bits = $lob->getPhoto();

// нужно явно вызывать __toString() т.к. в финальной версии PHP5
// разработчики отключили автоматический вызов __toString()
$bits = $lob->getPhoto()->__toString();

// то же самое можно сделать так:
$bits = $lob->getPhoto()->getContents();

// записываем фотографию в файл
$lob->getPhoto()->writeToFile('/tmp/photo.gif');

// или просто выводим в буфер
$lob->getPhoto()->dump();
```

Как видите, не сложнее чем работа с любыми другими полями. То же самое касается даты и времени. Для этих полей дополнительно вы можете указывать формат возвращаемого значения.

```
<table name="date_test">
  <column name="birth_date" type="DATE"/>
  <column name="birth_time" type="TIME"/>
  <column name="updated" type="TIMESTAMP"/>
</table>
```

По умолчанию результат для таких полей возвращается в соответствии с настройками локализации системы, которые можно поменять функцией PHP *setlocale()*, но можно задавать и собственный формат.

```
// вывод зависит от настроек локализации системы
print $obj->getBirthDate() . "\n";
print $obj->getBirthTime() . "\n";
print $obj->getUpdate() . "\n";
```

В системе с английской локализацией (en_US) вывод будет таким:

```
03/02/1998
12:34:23 AM
2003-12-21 18:32:01
```

В системе с немецкой локализацией (de_DE) вывод будет таким:

```
02.03.1998
00:34:23
2003-12-21 18:32:01
```

Можно указать желаемый формат вывода напрямую.

```
print $obj->getBirthDate('n/j') . "\n";
print $obj->getBirthTime('%X') . "\n";
print $obj->getUpdate(null) . "\n";   // UNIX timestamp
```

В результате выполнения этого примера вы увидите:

```
3/2
12:34:23 AM
1072049521
```

Реляционные отношения

Propel поддерживает базовые отношения вида «один-к-одному». Отношения «многие-ко-многим» реализуются с помощью вспомогательных таблиц в модели данных. Отношения между таблицами задаются с помощью внешних ключей (foreign keys).

```
<table name="book">
  <column name="book_id" type="INTEGER" required="true"
primaryKey="true"/>
  <column name="title" type="VARCHAR" size="100" required="true"/>
  <column name="author_id" type="INTEGER" required="true"/>
  <foreign-key foreignTable="author">
    <reference
      local="author_id"
      foreign="author_id"/>
  </foreign-key>
</table>
<table name="author">
  <column name="author_id" type="INTEGER" required="true"
primaryKey="true"/>
  <column name="fullname" type="VARCHAR" size="40" required="true"/>
</table>
```

Выборка связанных объектов

Для приведенной схемы данных можно использовать Book->getAuthor() чтобы получить автора книги по внешнему ключу таблицы *book*.

```
$books = BookPeer::doSelect(new Criteria());
foreach ($books as $book) {
    $author = $book->getAuthor();
}
// запросы к базе выглядят так:
// SELECT * FROM book
// SELECT * FROM author WHERE author_id = $book->getAuthorId()
```

Кроме того, Propel генерирует методы для получения информации о книге и авторе одним запросом:

```
$books = BookPeer::doSelectJoinAuthor(new Criteria());
foreach ($books as $book) {
    $author = $book->getAuthor();
}
// запрос примет такой вид:
//SELECT* FROM book INNER JOIN author ON author.author_id =book.author_id
```

Чтобы ограничить число методов в общедоступном API класса, все такие методы объявлены protected в базовом Peer-классе. Поэтому, вам придется сначала создать в классе-заглушке реализацию этих методов с модификатором public.

```
class BookPeer {
    public function doSelectJoinAuthor(Criteria $c) {
        return parent::doSelectJoinAuthor($c);
    }
}
```

Отношения «многие-ко-многим»

Допустим, мы хотим связать книги с людьми, которые их читают. Для этого опишем вспомогательную таблицу *book_reader_ref*.

```
<table name="book_reader_ref">
  <column name="book_id" type="INTEGER" required="true"
primaryKey="true"/>
  <column name="reader_id" type="INTEGER" required="true"
primaryKey="true"/>
  <foreign-key foreignTable="book">
    <reference
      local="book_id"
      foreign="book_id"/>
  </foreign-key>
  <foreign-key foreignTable="reader">
    <reference
      local="reader_id"
      foreign="reader_id"/>
  </foreign-key>
</table>
```

Пример кода для выборки связанных значений:

```
$books = BookPeer::doSelect(new Criteria());

// для каждой книги получим всех читателей
foreach ($books as $book) {
    $readers = $book->getBookReaderRefsJoinReader();
}
// SQL-запросы:
// SELECT * FROM book
//
// SELECT * FROM book_reader_ref
//   INNER JOIN reader ON reader.reader_id = book_reader_ref.reader_id
//     WHERE book_reader_ref.book_id = $book->getBookId()
```

Что еще нужно знать?

Статья представляет собой всего лишь краткий обзор возможностей Propel. Не был рассмотрен подробно файл *schema.xml* из примера *bookstore*, поскольку он достаточно большой. Однако разобраться с ним не составит труда, и будет полезно для понимания, как создавать собственные схемы данных. Полное описание всех возможностей схемы вы найдете в официальной документации.

Там же вы найдете главу, посвященную управлению наследованием сгенерированных классов из файла *schema.xml*.

Не рассмотрено в статье использование MapBuilder-классов, представляющих информацию о структуре базы данных. Если такая информация вам вдруг потребуется, в документации есть небольшой пример на эту тему.

Описание ключей генератора вы также найдете в официальном руководстве.

В Propel также имеется интересная возможность задавать в схеме правила проверки полей (элементы <*validator*> и <*rule*>) на минимальную и максимальную длину/значение поля, на соответствие некоторому шаблону и просто на присутствие данных для этого поля. Для каждого правила задается сообщение, возвращаемое в случае, если условие не выполнено (атрибут *message* для тэга <*rule*>).

Заключение

Propel позволяет рационально использовать время программиста. Вместо размышлений о том, как написать очередной SQL-запрос, или повторного написания, не меняющегося от проекта к проекту кода, можно сосредоточиться на предметной области. Выигрыш в скорости создания приложений с использованием Propel значительный, а потери производительности практически незаметны.

Документирование кода, или как сделать разработку удобней?

Автор:
Артем Сидоренко

Перед всеми разработчиками, работающими над проектами, рано или поздно встает вопрос о документировании кода. Более того, при работе в команде это просто необходимо, дабы другому разработчику не приходилось искать в чужом коде описания функций, классов и прочего, а сразу использовать чужой код. В связи с этим хотел бы немного осветить этот вопрос.

Как-то раз, изучая документацию pear.php.net по одному из классов, обратил внимание на строку "*Documentation generated on Sun, 22 Aug 2004 09:10:05 -0400 by phpDocumentor 1.2.3.*". Заинтересовался, проследовал по ссылке, почитал. Пришел к выводу, что это поможет мне сэкономить немало времени, так как на составление сопроводительной документации я тратил всегда немало времени.

Что такое phpDocumentor?

Это такая утилита, которая в настоящее время стала стандартом для документации PHP-кода. Работает она следующим образом.

Вы указываете путь, где лежат файлы вашего проекта, расширения файлов для анализа и прочие настройки, запускаете. Она анализирует код, и на выходе получается полная документация по коду. PhpDocumentor написан на PHP, и существует две возможности работы с ним: через коммандную строку и через веб-интерфейс. Более подробная информация доступна в справке phpDocumentor'a. Также, дабы каждый раз не указывать все параметры для создания документации, существуют конфигурационные файлы, подробнее они будут рассмотрены в конце статьи.

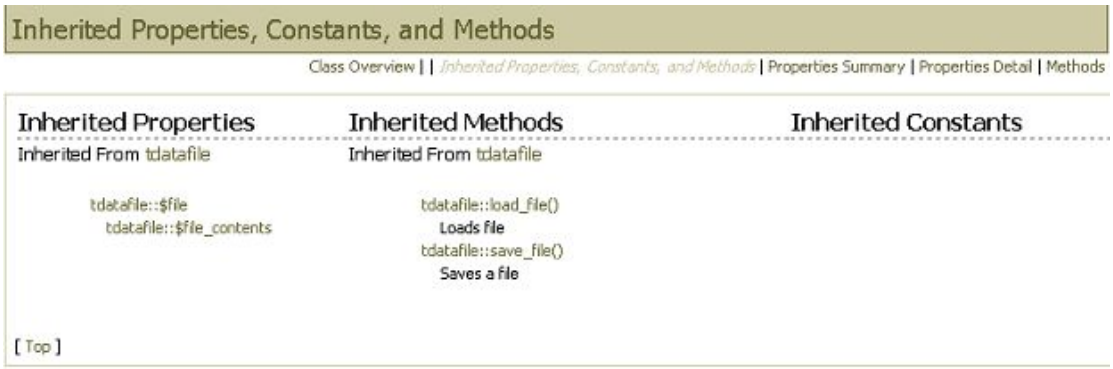


Рис.1. Пример сгенерированой документации.

Как это работает?

В каждом файле вашего проекта, а также перед каждым значимым элементом, таким как функция, константа и проч., вы делаете специальный комментарий, с которым phpDocumentor и работает.

Рассмотрим на примере комментария для файла.

```
<?php
/**
 * Config Module
 * For working with our config files.
 * Config files have INI-files structure.
 * @package syslybs
 * @author Everybody <everybody@something.com
 * @version 1 , 31.10.2004
 * @since engine v.1.1
 * @copyright (c) by Everybody
 */
```

Первая строка *"Config module"* означает имя, или предназначение данного файла. Обратите внимание на точку в конце, без нее следующая строка будет восприниматься как продолжение первой. Чтобы этого избежать, ставится точка, либо между первой и второй строкой необходимо вставить пустую строку. Вторая строка *"For working with our config files."* - описание данного файла. Может быть многострочным.

- **"@package"** означает принадлежность данного файла, к какому либо пакету, либо части вашего проекта (многие части состоят ведь более чем из одного файла);
- **"@author"** указывается автор, разработчик. В "<>" скобках можно указать e-mail адрес для связи. Он впоследствии будет преобразован в ссылку;
- **"@version"** информация о версии;
- **"@since"** с какого момента данный файл появился, или доступен для использования в проекте;
- ***"@copyright"*** копирайт.

Вот что в итоге из этого получится:

Рис. 2. Результат работы.

Комментирование функций

```
<?php
/**
 * Test function
 * Used for everything
 * Config files have INI-files structure.
 * @param integer $test our test parameter
 * @return string
 */
 function test($test) {
    if ($tcot==123) return 'abc';
    else return 'cba';
 } //\\test
```

Первые две строки комментария аналогичны – это описание. “@param” описывает параметр функции. Ситаксис следующий:

```
@param type name desc
```

- **type** – тип переменной;
- ***name*** – соответственное ее имя;
- ***desc*** – описание.

“@return” описывает тип результата, возвращаемый функцией. Результат:

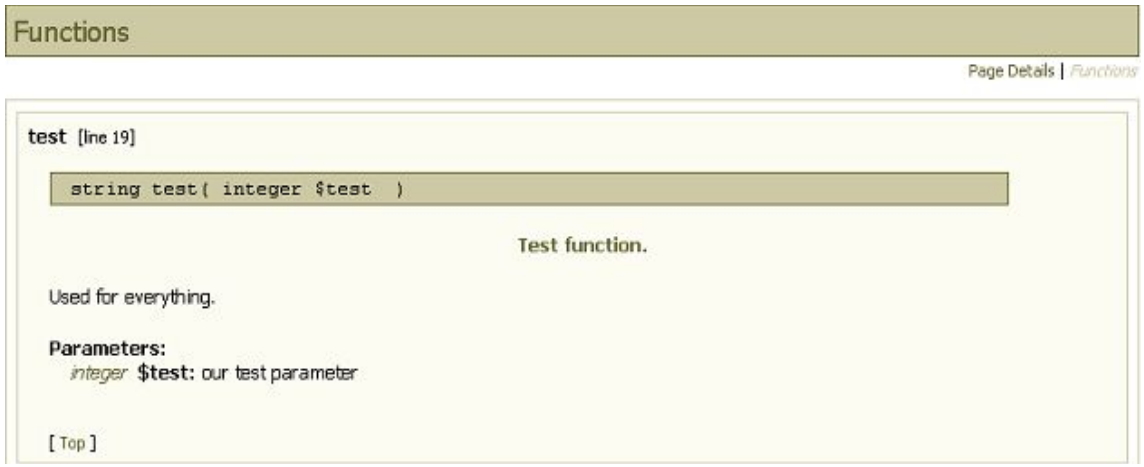


Рис. 3. Результат документирования функции.

Комментрование констант

Просто описание константы:

```
/** test constant */
define ('test','everything');
```

В результате:

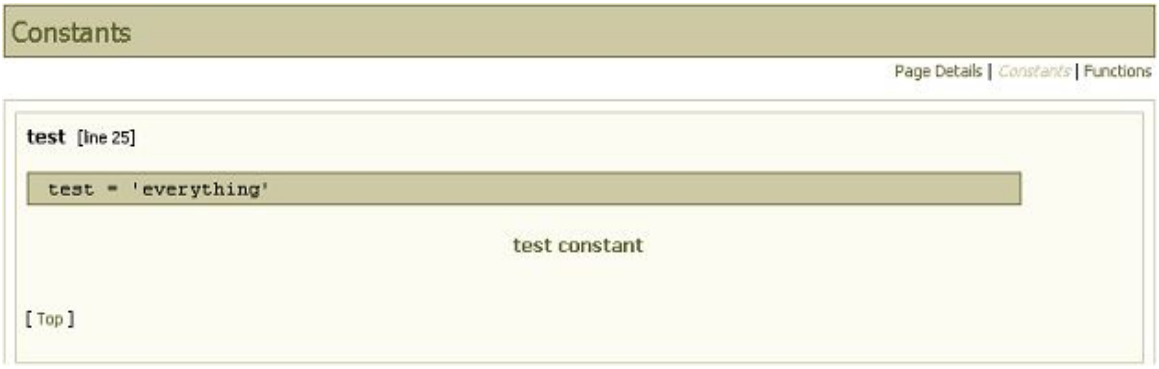


Рис. 4. Результат документирования константы.

Комментрование классов

Один из самых важных моментов. Потому как включает в себя все предыдущие.

Перед самим описанием класса ставится комментарий, аналогичный тому, который мы применяли в начале файла, т.е. Указывается принадлежность к пакету, разработчик, версия и т.д.

Можно этот комментарий не ставить, тогда информация будет взята из заголовка файла, в котором хранится класс.

```
class ttest {
    /**
    * A variable
    * @var array
    */
    var $a = array();

    /**
    * Test function
    * Used for everething
    * @param integer $test our test parameter
    * @return string
    */
    function test($test) {
        if ($test==123) return 'abc';
        else return 'cba';
    }//\\test
}//\\ttest
```

Думаю что, вам уже понятно как проводится описание свойства “a”. Сначала описание, затем при помощи ключевого слова *“@var”* задается тип данного свойства. С описанием метода действует все тоже самое, что и при описании функции.

В результате:

Рис. 5. Документирование класса.

Несколько общих моментов

Бывает, что метод, функция, свойство, константа существуют лишь для внутреннего использования, естественно, что остальным разработчикам, которые будут просто использовать вашу работу, знать о них ни к чему. Именно для этого существует ключевое слово *"@access*", т.е. если вы хотите указать что данную функцию, метод и т.д. не надо отображать в документации, вы должны указать *"@access private"*.

Внимание! Если в конфугарации phpDocumentor'a указана соответствующая опция, то данные "скрытые" элементы также будут обработаны и включены в документацию. Это позволяет вести два вида документации: для внутренного использования, и публичную.

Конфигурация phpDocumentor'a

Рассмотрим устройство конфигурационных файлов. Полный текст я приводоить не буду, просто дам описание и значение ключей.

- **`title`** – заголовок, название документации, проекта
- **`hidden`** – парсить также скрытые файлы, начинающиеся с точки (.htaccess etc.)

- **`parseprivate`** – включать ли в документацию скрытую информацию(например отмеченную как *“@access private”*)
- **`defaultpackagename`** – имя пакета “по-умолчанию”, т.е. открывающегося при открытии документации.
- **`target`** – путь, куда сохранять готовую документацию
- **`readmeinstallchangelog`** – имена файлов, которые также содержат какую-либо информацию, такие как readme, install, changelog
- **`directory`** – путь, где находяться файлы с кодом
- **`ignore`** – шаблоны имен файлов, которые надо игнорировать при составлении документации
- **`output`** – вид генерируемой документации, т.е. HTML, XML, CHM, также при HTML выбираются дополнительные параметры, как шаблоны
- **`sourcecode`** – включать ли в документацию исходный код файлов?

Надеюсь, что данный материал был полезен, и в будущем не раз поможет вам.

Данная статья ни в коем случае не претендует на изложение полных возможностей phpDocumentor'a, а призвана дать примерное описание и общий обзор возможностей.

Более подробная документация доступна на официальной странице, www.phpdoc.org.

Введение в стандартную библиотеку PHP 5

Автор:
Гарри Фукс
(Harry Fuecks)
Перевод:
Виктор Кулаков

Большая часть шумихи вокруг PHP5 сосредоточилась на его новом объектно-ориентированном синтаксисе, а также возможностях сравнимых с языком Java. А пока эта суета продолжалась, расширение с многообещающим названием «Standard PHP Library» (SPL, стандартная библиотека PHP) потихоньку проложило свой путь в самую сердцевину дистрибутива PHP5.

Хотя работа над этой библиотекой все еще продолжается, средства, предложенные в текущей версии Стандартной Библиотеки PHP, значительно увеличивают шансы разработчиков приложений на языке PHP достичь согласия кое в чем (при этом увеличивая шансы повторного использования кода). Она может также сделать интерфейс вашего хитро устроенного класса очень простым для использования остальными программистами, поскольку библиотека SPL позволяет «перегружать» (overload) базовый синтаксис PHP и делает объекты похожими на обычные массивы в PHP.

В этой статье предлагается введение в функциональные возможности, доступные с помощью расширения SPL и PHP5, и достаточное для начала работы количество примеров. Будьте внимательны: используется синтаксис PHP5. Если вы хотите ухватить основные особенности PHP5, можете посмотреть обзор PHP5 [1] на сайте SitePoint.

Содержание статьи:

- Введение в SPL: а о чём это вообще?
- Зацикливание цикла: кто сказал «Итератор»?
- Итерации для каждого из нас: фактор «вау»;
- Итерации с помощью SPL: шаг за шагом;
- Поклонение Древу: краткий курс классов и интерфейсов SPL;
- Объекты как массивы: так проще для вашего веб-дизайнера;
- Великое дело: почему вам это нравится.

Не забудьте скачать все исходные тексты этой статьи [2] для самостоятельного использования.

Введение в SPL

Расширение «Standard PHP Library» — это расширение PHP, разработанное Маркусом Бёргером (Marcus Boerger) [3], которое, (как сказано в руководстве [4]), «есть коллекция интерфейсов и классов, которые предназначены для решения стандартных проблем». Как часть основного дистрибутива PHP5, оно должно быть «всегда включено».

Если вы хорошо знакомы с PHP4, то вы знаете, что есть несколько сфер, где вечно изобретается велосипед (почти в каждом новом PHP-проекте).

Стандартизация некоторых из этих фундаментальных приемов будет самой короткой дорогой к тому, чтобы разработчики на PHP пели в унисон, и увеличивает шансы на то, что мы сможем повторно использовать код из Проекта А в проекте Б.

Сегодня расширение SPL обращается к единственному подмножеству проблем: Итераторы (повторители в циклах). Реализацию итераторов в SPL делает интересной не только то, что они задают стандарт для всех при использовании PHP5, но и то, что они «перегружают» (overload) определенные части синтаксиса PHP, такие, как конструкция foreach и базовый синтаксис массивов, делая их более легкими для работы с объектами ваших классов.

Зацикливание цикла

Итак, что есть итератор? В нашем контексте, итератор — это «шаблон проектирования» в программировании, выявленный «Бандой четырех» (Gamma E., Helm R., Johnson R., Vlissides J.) в их потрясающей книге “Шаблоны проектирования” (Design Patterns: Elements of Reusable Object Oriented Software) [5]

Задачей Итератора является «предоставить объекту, который проходит через некие сложные структуры, абстрагирование от предположений о реализации этих структур». Как все общие определения, точное значение этой фразы может показаться не очень понятным на первый взгляд. Под «сложными структурами» мы понимаем все то, через что мы можем «проходить в цикле», например, строки в выборке из базы данных, список файлов в каталоге или каждую новую строку в текстовом файле. Используя «обычный» PHP, вы можете использовать следующий код для цикла по результатам SQL-запроса:

```
// Выборка из "сложной структуры"
$result = mysql_query("SELECT * FROM users");

// Применяем итерацию к этой структуре
while ( $row = mysql_fetch_array($result) ) {
    // Здесь поместим обработку очередной записи
}
```

Чтобы прочитать содержимое каталога, вы можете использовать:

```
// Выборка из "сложной структуры"
$dh = opendir('/home/harryf/files');

// Применяем итерацию к этой структуре
while ( $file = readdir($dh) ) {
    // Здесь поместим обработку очередного файла
}
```

А чтобы прочитать содержимое файла, вы можете использовать:

```
// Выборка из "сложной структуры"
$fh = fopen("/home/hfuecks/files/results.txt", "r");
// Применяем итерацию к этой структуре
while (!feof($fh)) {
    $line = fgets($fh);
    // Здесь поместим обработку очередной строки
}
```

Видно, что приведенные примеры очень похожи. Хотя каждый работает со своим типом ресурса, и использует функции PHP, специфические для этого ресурса, заклинание будет очень простым: извлечь ресурс, пройти в цикле по всему содержимому.

Если бы удалось как-то «абстрагироваться» от специфических функций PHP в приведенных примерах и использовать вместо них некий общий интерфейс, стало бы возможным сделать работу прохода по данным в цикле выглядящей одинаково, независимо от типа использованного ресурса. Если избавиться от необходимости модифицировать цикл для различных источников данных, станет возможным использовать код, расположенный в теле цикла (например, функцию, создающую список HTML), где угодно.

Вот тут-то и появляется итератор. Он определяет абстрактный интерфейс для использования в вашем коде. Специфические реализации итератора заботятся о каждом типе структур данных, с которыми вы работаете. А код, использующий этот итератор, освобождается от заботы о деталях.

Такова базовая теория итераторов. Если вы хотите знать больше, то отправные точки найдете на сайтах C2 Wiki [6] и Wikipedia [7]. Дополнительная пища для размышлений может быть найдена на сайте phpPatterns в статье Iterator Pattern [8] и в статье The PHP Anthology - Volume II, Applications [9].

Итерации для каждого из нас

Итак, что же восхитительного в итераторах SPL? Ну, если вы написали больше, чем пару строк кода на PHP, скорее всего, вы сталкивались с оператором foreach [10] («для каждого»), который используется для облегчения перебора элементов массива:

```
// Список цветов
$colors = array (
    'красный',
    'зеленый',
    'синий',
    );

foreach ( $colors as $color ) {
    echo $color.'<br>';
}
```

Неплохо бы сделать так, чтобы все циклы были такими же простыми, независимо от того, что мы перебираем в цикле.

Как насчет этого?

```
<?php
// Волшебный класс... (объяснения чуть позже)
class DirectoryReader extends DirectoryIterator {

  function __construct($path) {
    parent::__construct($path);
  }

  function current() {
    return parent::getFileName();
  }

  function valid() {
    if ( parent::valid() ) {
      if ( !parent::isFile() ) {
        parent::next();
        return $this->valid();
      }
      return TRUE;
    }
    return FALSE;
  }

  function rewind() {
    parent::rewind();
  }
}

// Создаем "читателя" каталога для текущего каталога
$Reader = new DirectoryReader('./');

// Перебираем файлы в этом каталоге ?!?
foreach ( $Reader as $Item ) {
     echo $Item.'<br>';
}
?>
```

Листинг: directoryreader.php

Если вы на мгновение проигнорируете сам класс и взглянете на последние несколько строк листинга, вы увидите, что здесь использован объект DirectoryReader прямо в цикле foreach. Мы извлекаем значения из него без необходимости обращаться к каким-либо его методам!

Если вы соблюдаете определенные правила (к которым мы скоро перейдем), расширение SPL позволит проходить в цикле по итерациям по вашим собственным классам (где это возможно) абсолютно одинаковым способом.

Фактически в приведенных примерах мы ныряем в омут прямо с головой! Давайте отойдем на пару шагов, чтобы я смог объяснить, что же на самом деле здесь происходит.

Итерации с помощью SPL

Теперь, когда аппетит разыгрался, пора предупредить, что в руководстве по PHP в настоящий момент отсутствуют возможности, необходимые, чтобы полностью документировать расширение SPL. Это связано прежде всего с документированием основных функций и отсутствием понятных средств для того, чтобы полностью описать что-либо, как встроенный класс; интерфейсы даже не упоминаются.

Вместо этого, вам надо посмотреть на сгенерированную документацию [11], которую поддерживает Маркус (Marcus), и вытащить исходные тексты под CVS [12].

Предупреждаю, что расширение SPL быстро изменяется, активно разрабатывается и расширяется. Код в этой статье проверен под PHP 5.0.1, но если вы будете читать его через некоторое, достаточно удаленное от момента написания статьи время, то можете обнаружить, что часть кода устарела.

Расширение SPL определяет иерархию классов и интерфейсов [13]. Некоторые из них будут уже загружены в вашу инсталляцию PHP5 (смотрите, что делают get_declared_classes() [14]). Они соответствуют определениям интерфейсов и классов, определенным здесь [15] и здесь [16] (файлы PHP находящихся там, возможно исчезнут, как только Маркус найдет время переписать их на Си).

Некоторые классы, находящиеся в каталоге примеров [17] (с расширением .inc) также составляют часть иерархии, но не загружаются по умолчанию; если вы хотите использовать их, вам надо убедиться, что их копии находятся в пути поиска PHP (include_path). Больше примеров классов может быть найдено в тестах [18], а независимые примеры могут быть найдены на сайте http://www.wiki.cc/php/PHP5#Iterators [19].

Хотя количество классов и интерфейсов в иерархии может быть на первый взгляд устрашающим, не надо впадать в панику! Базовое использование итераторов требует только одного интерфейса. Если идеи интерфейсов для вас нова, взгляните на эту дискуссию по интерфейсам [20] на сайте SitePoint.

Мы приведем список всех предварительно загруженных классов и интерфейсов в этой статье позже (в разделе «Поклонение Древу»), чтобы вы просматривали его в свое удовольствие.

Когда вы начнете улавливать, что вам предлагается, вы поймете, какую грандиозную работу выполнил Маркус по решению самых частых проблем, связанным с циклами в PHP. Жить станет легче...

Давайте вернемся к примеру с DirectoryReader. Как это получилось, что мы смогли перебрать значения моего объекта DirectoryReader, используя foreach? Источник волшебства — класс-предок DirectoryIterator [21], который реализует определенный в библиотеке SPL интерфейс, называемый Iterator [22].

Любой класс, который применяет интерфейс Iterator, может быть использован в цикле foreach (обратите внимание на статью в которой объясняется, как это работает [23] с точки зрения внутренностей PHP). Интерфейс Iterator определен следующим образом:

```
interface Iterator extends Traversable {

    /**
    * "Перемотаем" Iterator к началу, к первому элементу.
    * Похоже на функцию reset() для массивов в PHP
    * @return void
    */
    function rewind();

    /**
    * Вернем текущий элемент.
    * Похоже на функцию current() для массивов в PHP
    * @return mixed текущий элемент из коллекции
    */
    function current();

    /**
    * Вернем идентификационный ключ текущего элемента.
    * Похоже на функцию key() для массивов в PHP
    * @return mixed либо целое, либо строка
    */
    function key();

    /**
    * Передвинемся вперед на следующий элемент.
    * Похоже на функцию next() для массивов в PHP
    * @return void
    */
    function next();

    /**
    * Проверим, существует ли текущий элемент после
    * обращения к rewind() или next().
    * Используется для проверки, дошли ли мы в цикле итераций
    * до конца коллекции
    * @return boolean FALSE если больше не к чему применять итерации
    */
    function valid();

}
```

Заметьте, что расширение SPL регистрирует интерфейс `Traversable` (т.е. «умеющий обходить в цикле»), от которого `Iterator` наследуется (inherit) интерпретатором Zend, чтобы позволить использование `foreach`. Этот `Traversable interface` реализован не непосредственно в интерпретаторе PHP, а в других встроенных классах PHP (в настоящее время этим занимается расширение SimpleXML; возможно, что это будет делать расширение SQLite, но сейчас оно обращается непосредственно к Zend API). Чтобы реализовать этот интерфейс, ваш класс должен обеспечить все определенные выше методы.

Чтобы показать, как это работает, мы начнем с изобретения велосипеда — реализации итератора для обычных массивов языка PHP.

Это как будто бессмысленное упражнение поможет понять, как работает итератор — без углубления в специфические детали, в которых легко заблудиться. Сначала мы определим класс, управляющий итерацией:

```
/**
* Итератор для обычных массивов PHP. Изобретение велосипеда
*
* Обратите внимание на конструкцию "implements Iterator" - ВАЖНО!
*/
class ArrayReloaded implements Iterator {
    /**  Обычный массив PHP, к которому будем применять итерации */
  private $array = array();
    /** Флаг для отслеживания конца массива */
  private $valid = FALSE;
    /**
    * Конструктор
    * @param array Обычный массив PHP,
    * к которому будем применять итерации
    */
  function __construct($array) {
    $this->array = $array;
  }
    /**
    * Перемещаем "указатель" массива на первый элемент
    * PHP's reset() возвращает FALSE если массив не имеет элементов
    */
  function rewind(){
    $this->valid = (FALSE !== reset($this->array));
  }
    /** Возвращает текущий элемент массива */
  function current(){
    return current($this->array);
  }

    /** Возвращает ключ текущего элемента массива
    */
  function key(){
    return key($this->array);
  }
    /**
    * Продвинуться вперед на один элемент
    * PHP's next() возвращает FALSE если массив не имеет элементов
    */
  function next(){
    $this->valid = (FALSE !== next($this->array));
  }
    /**
    * Текущий элемент существует?
    */
  function valid(){
    return $this->valid;
  }
}
```

Листинг: arrayreloaded.php

Обратите внимание на конструкцию «implements Iterator» в начале. Она говорит, что мы соглашаемся выполнить «контракт» с итератором и обеспечить все требуемые методы.

Класс затем обеспечит реализацию каждого метода, выполнит необходимую работу, используя обычные функции работы с массивами в PHP (комментарии объясняют детали).

Есть пара моментов в конструкции итератора (Iterator), о которых стоит помнить, когда вы пишете свой код. Методы current() и key() в Итераторе могут вызываться несколько раз за время одной итерации цикла, поэтому будьте внимательны, чтобы, вызывая их, не изменить состояние Итератора. В данном случае это не проблема, но при работе с файлами, например, может возникнуть соблазн использовать fgets() [24] внутри метода current(), а это передвинет указатель позиции в файле.

В других случаях помните, что метод valid() должен показывать, что существует именно текущий (current) элемент, но не следующий (next) элемент. Это означает, что при прохождении цикла в Итераторе, мы в действительности продвигаемся на один элемент за конец коллекции и обнаруживаем этот факт только тогда, когда вызываем метод valid(). Обычно будут существовать методы next() (следующий) и rewind() (перемотать), которые реально передвигают итератор и заботятся о проверке существования очередного элемента. Теперь мы можем использовать этот класс следующим образом:

```
// Создаем объект-итератор
$colors = new ArrayReloaded(array ('красный','зеленый','синий',));

// Начать итерации!
foreach ( $colors as $color ) {
  echo $color."<br>";
}
```

Это же очень просто! За кулисами оператор foreach вызывает методы, которые мы определили, начиная с rewind(). Затем, поскольку метод valid() возвращает TRUE, он вызывает метод current (), чтобы занести значение в переменную $color, и метод next(), чтобы передвинуть итератор вперед на один элемент. Как и обычно, в foreach мы можем присваивать переменным значения, возвращаемые методом key():

```
// Показать и ключи тоже
foreach ( $colors as $key => $color ) {
  echo "$key: $color<br>";
}
```

Конечно, никто не заставляет нас использовать именно `foreach`. Мы можем вызывать методы непосредственно из нашего кода, например, так:

```
// Сбросить итератор (foreach делает это автоматически)
$colors->rewind();
while ( $colors->valid() ) { // Повторять пока есть элементы
    echo $colors->key().": ".$colors->current()."<br>";
    $colors->next();
}
```

Этот пример должен помочь вам увидеть, что foreach на самом деле делает с вашим объектом.

Заметьте, что простейший тест производительности показал, что вызов методов напрямую действует быстрее, чем использование foreach, поскольку последний добавляет дополнительный уровень перенаправления, который должен быть обработан во время исполнения скрипта интерпретатором PHP.

Поклонение Древу

Теперь вы знаете, как написать простейший итератор. Пора перечислить интерфейсы и классы, встроенные в расширение SPL, чтобы вы знали, чем они занимаются. В будущем список может измениться, но сейчас он отражает то, что есть в наличии.

Интерфейсы

- **Traversable** [25]: как упомянуто выше, это итераторный интерфейс к внутренностям PHP. Игнорируйте его, если только вы не пишете собственное расширение PHP.
- **Iterator** [26]: как вы уже видели, этот интерфейс определяет базовые методы для того, чтобы производить итерации через коллекцию.
- **IteratorAggregate** [27]: если вы намерены реализовать этот интерфейс отдельно от вашего объекта «коллекции», использование итератора Aggregate позволит вам делегировать работу по итерации в отдельный класс, не теряя возможности использовать эту коллекцию внутри цикла foreach.
- **RecursiveIterator** [28]: этот интерфейс определяет методы для применения итераций к иерархическим структурам данных.
- **SeekableIterator** [29]: этот интерфейс определяет метод для поиска по коллекции, которой управляет данный итератор.
- **ArrayAccess** [30]: это другой пример «волшебного» интерфейса в интерпретаторе Zend. Его применение позволит вам обращаться к объектам, как к массивам с обычным синтаксисом массивов PHP (подробнее об этом — дальше).

Классы

- **ArrayIterator** [31]: этот итератор может управлять и обычными массивами PHP, и публичными свойствами объектов (перебирая их последовательно).
- **ArrayObject** [32]: этот оператор унифицирует массивы и объекты, позволяя вам применять итерации к ним и используя синтаксис массивов для доступа к их содержимому. См. «Объекты как массивы» далее (мы создадим собственный класс с похожим поведением).
- **FilterIterator** [33]: это абстрактный класс, который может быть расширен для фильтрации элементов, к которым применяются итерации (возможно, удаляя нежелательные элементы для поиска).

- **ParentIterator** [34]: когда вы используете ResursiveIterator, ParentIterator позволяет вам отфильтровывать элементы, у которых нет «потомков». Если, например, у вас есть CMS, в которой документы могут быть помещены в любом месте дерева категорий, итератор ParentIterator позволит рекурсивно обходить дерево, но показывать только узлы «категории», пропуская документы, которые появляются под каждой категорией.
- **LimitIterator** [35]: этот класс позволяет вам определить ранг элементов для применения итераций, указывая начальное значение ключа и указывая количество элементов, которые надо пройти, начиная с данной точки. Это та же концепция, что и условие LIMIT в MySQL.
- **CachingIterator** [36]: этот класс управляет другими итераторами (которые вы подаете на его конструктор). Он позволяет вам проверить, имеет ли внутренний итератор еще элементы, используя метод hasNext() перед тем, как действительно продвинутся вперед методом next(). Лично я не на 100% уверен в имени, возможно LookAheadIterator будет более корректно?
- **CachingRecursiveIterator** [37]: это в основном тот же CachingIterator, но позволяющий итерации над иерархическими структурами данных.
- **DirectoryIterator** [38]: чтобы применять итерации к каталогу файловой системы, этот оператор предоставляет набор полезных методов, таких как isFile() и isDot(), которые сохраняют массу сил программисту.
- **RecursiveDirectoryIterator** [39]: этот класс позволяет применять итерации к структуре каталогов, так, что вы можете заходить в подкаталоги.
- **SimpleXMLIterator** [40]: делает простой SimpleXML [41] еще проще! Лучшие примеры могут быть найдены в тестах SPL: tests [42] — смотрите файлы, начинающиеся с «sxe_*»
- **RecursiveIteratorIterator** [43]: этот класс помогает вам выполнять интересную работу по превращению иерархической структуры данных в плоскую, так, что вы можете организовать по ней цикл с помощью одного оператора foreach, в то же время сохраняя данные об иерархии. Этот класс может быть очень полезен, например, для отображения древовидных меню.

Чтобы увидеть это в действии, попробуйте использовать DirectoryTreeIterator [44] (который расширяет RecursiveIterator), например, так:

```
$DirTree = new DirectoryTreeIterator('/some/directory');
foreach ($DirTree as $node) {
    echo "$node\n";
}
```

Итак, мы подвели итог встроенным классам и интерфейсам, которые определены сегодня в расширении SPL.

Объекты как массивы

Вы уже знаете, как применение интерфейса итератора позволяет «перегружать» конструкцию foreach. Расширение SPL имеет в запасе еще несколько сюрпризов, хотя бы, например, интерфейс ArrayAccess. Применение этого интерфейса в классе позволяет вам трактовать объекты класса как массивы с точки зрения синтаксиса PHP.

Вот пример:

```
/** Класс может быть использован как массив */
class Article implements ArrayAccess {

  public $title;
  public $author;
  public $category;

  function __construct($title,$author,$category) {
    $this->title = $title;
    $this->author = $author;
    $this->category = $category;
  }
  /**
  * Определено интерфейсом ArrayAccess interface
  * Присвоить значение заданному ключом элементу,
  * напр. $A['title'] = 'foo';
  * @param mixed key - ключ (строка или целое)
  * @param mixed value - значение
  * @return void
  */
  function offsetSet($key, $value) {
    if ( array_key_exists($key,get_object_vars($this)) ) {
      $this->{$key} = $value;
    }
}
  /**
  * Определено интерфейсом ArrayAccess interface
  * Возвращает значение, заданное ключом, напр. $A['title'];
  * @param mixed key - ключ (целое или строка)
  * @return mixed value - значение
  */
  function offsetGet($key) {
    if ( array_key_exists($key,get_object_vars($this)) ) {
      return $this->{$key};
    }
 }
  /**
  * Определено интерфейсом ArrayAccess interface
  * Сбрасывает значение, заданное ключом, напр.  unset($A['title']);
  * @param mixed key - ключ (целое или строка)
  * @return void
  */

  function offsetUnset($key) {
    if ( array_key_exists($key,get_object_vars($this)) ) {
      unset($this->{$key});
    }
}
```

Продолжение на следующей странице

```
/**
 * Определено интерфейсом ArrayAccess interface
 * Проверяет, существует ли значение, заданное ключом,
 * например isset($A['title'])
 * @param mixed key - ключ (целое или строка)
 * @return boolean
 */
 function offsetExists($offset) {
  return array_key_exists($offset,get_object_vars($this));
 }
}
```

Листинг: arrayaccess1.php (Окончание. Начало на предыдущей странице)

Четыре метода, начинающихся с «offset» определены с помощью интерфейса ArrayAccess [45], который мы применили.

Заметьте, что мы использовали пару трюков времени исполнения на PHP, чтобы облегчить себе жизнь, таких, как проверка, что переменные объекта существуют:

```
function offsetSet($key, $value) {
    if ( array_key_exists($key,get_object_vars($this)) ) {
```

Мы также обращались к ним косвенно, используя переменную, в которой хранится их имя:

```
$this->{$key} = $value;
```

Этот пример станет интересным, когда вы увидите, как этот класс теперь используется:

```
// Создаем объект
$A = new Article('SPL Rocks','Joe Bloggs', 'PHP');

// Проверяем, на что это похоже
echo 'Initial State:<pre>';
print_r($A);
echo '</pre>';

// Меняем заголовок, используя синтаксис массива
$A['title'] = 'SPL _really_ rocks';

// Пытаемся занести значение в несуществующее свойство (игнорируется)
$A['not found'] = 1;

// Сбрасываем свойство 'author'
unset($A['author']);
// Снова проверяем, на что это похоже
echo 'Final State:<pre>';
print_r($A);
echo '</pre>';
```

Кроме первой строки, в которой мы создаем объект, код является синтаксически правильным обращением к обычному массиву PHP. Вот вывод этого скрипта:

```
Initial State:

Article Object
(
    [title] => SPL Rocks
    [author] => Joe Bloggs
    [category] => PHP
)

Final State:

Article Object
(
    [title] => SPL _really_ rocks
    [category] => PHP
)
```

Заметьте, что мы можем добавить логику, чтобы работать с данными по мере их чтения, изменив метод offsetGet() таким образом:

```
function offsetGet($key) {
    if ( array_key_exists($key,get_object_vars($this)) ) {
      return strtolower($this->{$key});
    }
  }
```

Это переведет все буквы в нижний регистр.

Чтобы сделать объект доступным для итераций, используя foreach или другим способом, мы можем теперь воспользоваться преимуществами класса ArrayIterator [46], в сочетании с интерфейсом IteratorAggregate [47].

Как было уже упомянуто, интерфейс IteratorAggregate используется, когда вы не хотите встраивать логику итератора в объект, который содержит данные, к которым вы хотите применять итерации. Это может быть очень полезно — отделять логику, но, что более интересно, — позволяет повторно использовать существующие итераторы.

Сначала модифицируем первую строку в классе Article, чтобы определить применение интерфейса:

```
class Article implements ArrayAccess, IteratorAggregate {
```

Теперь нам надо добавить один дополнительный метод: `getIterator()`.

Метод возвращает объект, используемый для итерации:

```
/**
  * Определено интерфейсом IteratorAggregate
  * Возвращает итератор для этого объекта,
  * для использования в foreach
  * @return ArrayIterator
  */
  function getIterator() {
    return new ArrayIterator($this);
  }
```

Сделав это, мы проходим в цикле по свойствам, определенным в классе:

```
$A = new Article('SPL Rocks','Joe Bloggs', 'PHP');

// Перебираем в цикле (getIterator будет вызван автоматически)
echo 'Looping with foreach:<pre>';
foreach ( $A as $field => $value ) {
   echo "$field : $value<br>";
}
echo '</pre>';

// Получить размер итератора (посмотреть, сколько свойств осталось)
echo "Object has ".sizeof($A->getIterator())." elements";
```

Листинг: arrayaccess2.php

Вот что он показывает:

```
Looping with foreach:

title : SPL Rocks
author : Joe Bloggs
category : PHP

Object has 3 elements
```

Обратите внимание, что мы могли бы также применить функцию count [48] к объекту, чтобы определить, сколько в нем элементов. Это позволило бы нам использовать другие операторы цикла для вызова методов итератора:

```
$size = count($A);
for($i = 0; $i < $size; $i++ ) {
    echo $A[$i]."\n";
}
```

Что пока не работает, так это применение функций для работы с массивами в PHP [49] к нашим объектам.

Вы получите жалобы программы, что это не массивы. Однако если вы не используете проверки типа is_array() [50], то сможете повторно использовать любую часть вашего кода, который был написан для работы с массивами.

Великое дело

Надеемся, что у вас нарастает ощущение, что расширение SPL — нечто великое с точки зрения облегчения жизни PHP-программистов. То, что SPL предлагает уже сейчас, достойно восхищения, и надо отдать должное Маркусу Бёргеру за то, что он сделал это возможным. Это также устраняет некоторые сомнения, которые имелись по поводу интерфейсов в PHP5, доказывая их полезность в качестве механизма для изложения контракта между интерпретатором PHP и кодом, написанным на PHP, позволяя сильно изменять семантику конструкций PHP.

Возможно, самым важным аспектом действия SPL сегодня, является то, что она поощряет использование стандартов, во-первых, определяя набор прикладных интерфейсов API, которые «всегда включены» в PHP5 (поэтому почему бы их не использовать?), и, во-вторых, предлагает дополнительную «морковку», которой является возможность перегружать синтаксис PHP и таких операторов, как foreach.

При условии, что все мы согласимся использовать классы и интерфейсы, предоставляемые SPL, проекты могут начать на нем базироваться. Например, рассмотрим HTML_TreeMenu [51], библиотеку PEAR для генерации древовидных Javascript-меню на HTML. Сейчас она оставляет много забот разработчикам. Например, чтобы создать дерево из структуры каталогов с помощью HTML_TreeMenu сегодня требуется:

```
require_once 'HTML/TreeMenu.php';
$map_dir = 'c:/windows';
$menu  = new HTML_TreeMenu('menuLayer', 'images', '_self');
$menu->addItem(recurseDir($map_dir));
function &recurseDir($path) {
    if (!$dir = opendir($path)) {
        return false;
    }
    $files = array();
    $node = &new HTML_TreeNode(basename($path), basename($path), 'folder.gif');

    while (($file = readdir($dir)) !== false) {
        if ($file != '.' && $file != '..') {
            if (@is_dir("$path/$file")) {
                $addnode = &recurseDir("$path/$file");
            } else {
                $addnode = &new HTML_TreeNode($file, $file, 'document2.png');
            }
            $node->addItem($addnode);
        }
    }
    closedir($dir);
    return $node;
}
echo $menu->printMenu();
```

Другими словами, нам оставлена подготовка данных в правильном порядке и построение дерева. Вместо этого, HTML_Treemenu могло бы предоставить механизм, с помощью которого мы бы могли регистрировать структуру данных, затем расстаться с ней, чтобы HTML_Treemenu провела итерации для нас. Упомянутый пример может быть сокращен до следующего:

```
require_once 'HTML/TreeMenu.php';
$map_dir = 'c:/windows';
$menu  = new HTML_TreeMenu('menuLayer', 'images', '_self');

// Register the tree data structure
$menu->registerTree(new new RecursiveDirectoryIterator($map_dir);

echo $menu->printMenu();
```

Если под рукой нет итератора RecursiveIterator, который решит вашу проблему, вы можете всегда реализовать свой собственный, оставляя библиотеке HTML_Treemenu воспользоваться преимуществами технологии type hints [52], чтобы удостовериться, что вы передаете ей то, что ей нужно.

Еще стандартов, пожалуйста!

Вопрос стоит так: что еще можно стандартизировать? В конце концов, расширение названо «Стандартная библиотека PHP». После короткой переписки по электронной почте с Маркусом Бёргером, я счастлив сообщить всем несколько идей, которыми он планирует заняться в будущем (в зависимости от наличия собственного времени и поддержки добровольцев):

Стандартные исключения, такие как RuntimeException и OutOfBoundsException (нечто, эквивалентное встроенным исключениям [53] языка Python, где исключения названы по проблеме, которую они сигнализируют).

Некоторые идеи из Design by Contract TM [54] (которые, вероятно, введут новые языковые конструкции, такие, как requires() и ensures(), плюс, ассоциированные исключения).

Больше реализаций шаблонов проектирования (где возможно). Например, шаблон Observer (наблюдатель) [55] может стать особенно хорошим кандидатом, но лучше посмотреть, что именно ответил Маркус: «Проблема шаблона Observer в том, как сделать его без множественного наследования. Объекты, за которыми наблюдают, требует контейнера Наблюдателей. Но у нас есть только интерфейсы. К тому же, как поступать с объектами, которые должны действовать, как Наблюдатели для различных Наблюдаемых? Должен ли я передавать возникающего Наблюдаемого? И, если так, как они себя будут идентифицировать? Так что здесь есть над чем подумать. И есть сферы, где мне нужно больше информации и описанных случаев использования.»

Некоторые идеи из dependency injection [56] (шаблон проектирования для управления объектами, которые зависят от других объектов). Для этого надо сделать новый рефлексивный API более доступным.

Если вы хотите узнать больше или пообщаться с Маркусом напрямую, то место, где его можно поймать — это Международная конференция по PHP [57] в ноябре 2004 во Франкфурте, Германия, где он будет делать доклад по SPL.

Свое я сказал. Дальше — итерируйте!

Ссылки

[1] http://www.sitepoint.com/article/coming-soon-webserver-near/
[2] http://www.sitepoint.com/examples/php5/php5stdlib.zip
[3] http://marcus-boerger.de/
[4] http://www.php.net/spl
[5] http://www.amazon.com/exec/obidos/ASIN/0201633612/
[6] http://c2.com/cgi/wiki?IteratorPattern
[7] http://en.wikipedia.org/wiki/Iterator
[8] http://www.phppatterns.com/index.php/article/articleview/50/1/1/
[9] http://www.sitepoint.com/books/phpant1/
[10] http://www.php.net/foreach
[11] http://www.php.net/~helly/php/ext/spl/
[12] http://cvs.php.net/pecl/spl/
[13] http://www.php.net/~helly/php/ext/spl/hierarchy.html
[14] http://www.php.net/get_declared_classes
[15] http://cvs.php.net/co.php/pecl/spl/spl.php
[16] http://cvs.php.net/pecl/spl/internal/
[17] http://cvs.php.net/pecl/spl/examples/
[18] http://cvs.php.net/pecl/spl/tests/
[19] http://www.wiki.cc/php/PHP5#Iterators
[20] http://www.sitepoint.com/article/coming-soon-webserver-near/7
[21] http://www.php.net/~helly/php/ext/spl/classDirectoryIterator.html
[22] http://www.php.net/~helly/php/ext/spl/interfaceIterator.html
[23] http://lxr.php.net/source/ZendEngine2/zend_iterators.h
[24] http://www.php.net/fgets
[25] http://www.php.net/~helly/php/ext/spl/interfaceTraversable.html
[26] http://www.php.net/~helly/php/ext/spl/interfaceIterator.html
[27] http://www.php.net/~helly/php/ext/spl/interfaceIteratorAggregate.html
[28] http://www.php.net/~helly/php/ext/spl/interfaceRecursiveIterator.html
[29] http://www.php.net/~helly/php/ext/spl/interfaceSeekableIterator.html
[30] http://www.php.net/~helly/php/ext/spl/interfaceArrayAccess.html
[31] http://www.php.net/~helly/php/ext/spl/classArrayIterator.html
[32] http://www.php.net/~helly/php/ext/spl/classArrayObject.html
[33] http://www.php.net/~helly/php/ext/spl/classFilterIterator.html
[34] http://www.php.net/~helly/php/ext/spl/classParentIterator.html
[35] http://www.php.net/~helly/php/ext/spl/classLimitIetrator.html
[36] http://www.php.net/~helly/php/ext/spl/classCachingIterator.html
[37] http://www.php.net/~helly/php/ext/spl/classCachingRecursiveIterator.html
[38] http://www.php.net/~helly/php/ext/spl/classDirectoryIterator.html
[39] http://www.php.net/~helly/php/ext/spl/classRecursiveDirectoryIterator.html
[40] http://www.php.net/~helly/php/ext/spl/classSimpleXMLIterator.html

[41] http://www.php.net/simplexml
[42] http://cvs.php.net/pecl/spl/tests/
[43] http://www.php.net/~helly/php/ext/spl/classRecursiveIteratorIterator.html
[44] http://cvs.php.net/co.php/pecl/spl/examples/directorytreeiterator.inc
[45] http://www.php.net/~helly/php/ext/spl/interfaceArrayAccess.html
[46] http://www.php.net/~helly/php/ext/spl/classArrayIterator.html
[47] http://www.php.net/~helly/php/ext/spl/interfaceIteratorAggregate.html
[48] http://www.php.net/count
[49] http://www.php.net/array
[50] http://www.php.net/is_array
[51] http://pear.php.net/package/HTML_TreeMenu
[52] http://www.sitepoint.com/article/coming-soon-webserver-near/11
[53] http://docs.python.org/lib/module-exceptions.html
[54] http://en.wikipedia.org/wiki/Design_by_contract
[55] http://c2.com/cgi/wiki?ObserverPattern
[56] http://www.martinfowler.com/articles/injection.html
[57] http://www.phpconference.de/2004/session-php5_en.php

Об авторе

Гарри Фукс работает в сфере корпоративных информационных технологий с 1994, со всеми, от начинающих до крупнейших компаний, вошедших в рейтинг «Fortune 100».

За пределами офиса он занимается проектом phpPatterns: это сайт, посвященный проектированию программ на PHP с целью поднять уровень стандартов PHP-разработок. Он также поддерживает раздел Dynamically Typed: SitePoint's PHP blog.

Оригинал статьи находится по адресу: http://www.sitepoint.com/print/php5-standard-library

Идеи

Safe_mode в PHP или какими средства защиты быть не должны

Автор:
Илья Альшанетский (Ilia Alshanetsky)
Перевод:
Данил Миронов

Как-то раз несколько лет назад разработчики PHP задумали решить проблему, которая их не касалась, и создали директиву конфигурации, названную safe_mode. Всем, кто не знаком с этим дивным изобретением, поясняю: изначально эта директива задумывалась для ограничения пользователей в доступе к не принадлежащим им файлам. Предполагалось, что эта директива запретит доступ к чужим файлам в разделяемой среде сервера, поскольку чаще всего PHP работает как модуль Apache и в этом своём качестве имеет право на чтение всех файлов, доступных веб-серверу, независимо от владельца. Когда включается safe_mode, производится uid/gid-сверка (user id и group id): берётся uid/gid файла/директории и сравнивается с uid/gid скрипта, который пытается получить доступ к нему. Если и первое, и второе совпадает, то операция с файлом проходит как обычно, во всех остальных случаях операция не проходит.

Теоретически это очень простой выход из положения при решении проблемы, все остальные методы связаны с очень большими потерями, например, запуск PHP в режиме CGI, все скрипты при этом будут исполняться с uid/gid пользователя. Однако практически все такие "выходы из положения" ведут к возникновению совершенно непредвиденных трудностей, и вы убеждаетесь, что все временные решения только лишь способствуют росту числа проблем.

Итак, в данной статье предлагается подробно рассмотреть проблемы safe_mode и, будем надеяться, показать способы их решения.

Первая проблема, даже, скорее всего, просто баг, - это то, что каждый раз, когда PHP или библиотеки, используемые его расширениями, пытаются получить доступ к файлу, safe_mode требует проведения uid/gid-проверки. Реализовать это несложно, кроме того, для этого существуют упаковщики, но с момента появления safe_mode, каждый новый релиз PHP содержит исправления по поводу пропущенных uid/gid-проверок. Конечно, это редко касается каких-либо популярных функций или расширений, но факт остаётся фактом: многие функции могут обойти это ограничение. Изучение расширений в репозитории PECL как раз и выявило, что многие из них подобных проверок не делают. Надо заметить, что некоторые из этих расширений очень даже популярны, и я видел их на многих хостах.

Это означает только одно: провайдеры, которые притворяются (ну да, мы все знаем, что они говорят о safe_mode), что safe_mode действительно работает, вводят самих себя в заблуждение, если, конечно, они не проводят аудит PHP-расширений, работающих на их серверах.

Провайдеры, которые притворяются, что safe_mode действительно работает, вводят самих себя в заблуждение, если, конечно, они не проводят аудит PHP-расширений, работающих на их серверах

Вторая, и, пожалуй, самая важная проблема заключается в реализации safe_mode и в том, какое действие она оказывает на работу пользователей.

Представим себе следующую ситуацию: у вас есть скрипт, генерирующий какую-то часто посещаемую страницу, значит, некоторые её части можно безболезненно закэшировать. Самое простое, что разработчики могут здесь сделать – это заранее сгенерировать эти данные, и хранить их в файле, и файл этот потом include-ить, экономя тем самым кучу времени на подготовку контента. Чудненько, под safe_mode создание такого файла пройдёт просто на отлично, однако вот доступ к нему осуществить не удастся.

Причина этой неприятности в том, что владельцем скрипта являетесь вы, и ваш uid/gid не совпадает с uid/gid у файла, потому что он был создан с принадлежащим веб-серверу (то есть, с uid/gid сервера), под которым работает PHP. Вот так safe_mode ограничивает вас в получении доступа к заранее подготовленным файлам, которые вы подгрузили через ftp или ssh, вы можете только добавлять [append] к нему (или обрезать [truncate] и добавлять) записи. Такое положение дел практически каждую файловую операцию превращает в откровенную проблему. Но к счастью для некоторых частей вашего тела, существует один обходной манёвр, он же и ставит под вопрос саму концепцию "безопасности", предлагаемой safe_mode.

Вы можете подгрузить [upload] все PHP-скрипты, которые будут совершать файловые операции, через другой скрипт, и таким образом, у нужных скриптов будет владелец и группа веб-сервера. И скрипт обретёт возможность редактировать/создавать/удалять файлы и директории, поскольку все участники операций теперь принадлежат веб-серверу. Однако сначала вам необходимо будет подготовить директории, куда будут писать эти подгруженные скрипты, поскольку уже созданные директории не имеют правильные uid/gid (их владельцем являетесь вы). И, в конечном счёте, достаточно сложная процедура сводится к такой банальности, как несколько файловых операций, выгладит всё примерно так:

1. Создание скрипта со страничкой для загрузки файлов
2. Подгрузка "рабочих" скриптов
3. Создание директорий, в которые рабочие скрипты будут проводить запись.

Некоторые провайдеры пытаются как-то бороться с этой проблемой, регулярно выставляя правильных владельцев (chown) для пользовательских файлов и директорий, однако это не только очень большая нагрузка для серверов с большим количеством файлов, но и неясная ситуация в период между загрузкой новых файлов и повальными chown-ами от провайдера.

Теперь интересная часть сказки: если какой-либо пользователь применяет описанный мной обходной манёвр против safe_mode, или просто пишет через PHP в файлы, которые впоследствии будут доступны с веб-сервера (например, создают статические html-странички), то все остальные пользователи могут читать или даже писать (если позволяют файловые разрешения) в эти файлы, подгрузив соответствующий PHP-скрипт через веб-сервер.

Тем временем...

В PHP можно манипулировать данными, полученными из внешних программ. Это делается с помощью функции exec(). Например, используем программу Date.

```
<?php
exec('date', $output);
print implode("\n", $output);
?>
```

Не используйте регулярные выражения там, где в этом нет нужды. PHP имеет много отличных функций для работы со строками. Пример:

Плохо:
```
$new = ereg_replace("-","_",$str);
```
Хорошо:
```
$new= str_replace("-","_",$str);
```

Плохо:
```
preg_match ('/(\..*?)$/',$str,$reg);
```
Хорошо:
```
substr($str,strrpos($str,'.'));
```

Если вы имеете возможность конфигурировать ваш веб-сервер apache, то для вас может быть полезной возможность устанавливать директиву conf.httpd ErrorDocument. Например:

```
ErrorDocument 404 /error.php
```

Таким образом, если посетитель сайта запросит несуществующую страничку (ошибка 404), веб-сервер переадресует его на скрипт error.php.

Эта возможность может быть полезной не только для адаптации сообщения об ошибке под фирменный дизайн, но и для того, чтобы направить псетителя на ближайшую по смыслу страничку вашего сайта.

Полезные серверные переменные:
REDIRECT_URL
REDIRECT_ERROR_NOTES

Третья проблема safe_mode состоит в том, что он накладывает дополнительные ограничения "безопасности ради", как отключение shell_exec и обратной кавычки [backtic]. В режиме safe_mode принудительно игнорируются все аргументы функций exec(), system(), и им подобных, последние при этом становятся практически бесполезными, особенно если нужно выполнить какую-либо команду с аргументами. С другой стороны, ничто не мешает сообразительному взломщику подгрузить скрипт или какой-либо исполняемый файл в "защищённую" директорию и запустить оный с разрешениями веб-сервера.

Я больше всего не люблю в safe_mode, то, что он вовсе ничего не защищает, создавая при этом иллюзию безопасности, да и она при ближайшем рассмотрении рассеивается

Даже если функции запуска программ отключены, то человек с включенным воображением сможет запустить любые скрипты по своему выбору через cron (а это очень даже распространенная услуга у провайдеров). И поскольку почти ни один провайдер не озаботился запретом всеобщего доступа на чтение из директорий других пользователей, вы можете читать чужие файлы. Более того, поскольку многие объекты могут иметь достаточные разрешения для того, что бы вы могли редактировать/удалять и даже хранить файлы на аккаунтах других пользователей. Неплохой манёвр, если провайдер выделяет мало дискового пространства :).

Но, наверное, что я больше всего не люблю в safe_mode, так это то, что он вовсе ничего не защищает, создавая при этом иллюзию безопасности, да и она при ближайшем рассмотрении рассеивается.

Более или менее соображающий взломщик без труда может обойти все эти препятствия. И если не через PHP, то через mod_perl, mod_python или даже через CGI-скрипт или бинарник, поскольку большинство провайдеров не ограничиваются предоставлением только PHP.

Обманчивая природа safe_mode вводит в заблуждение админов, которые идут по пути наименьшего сопротивления и просто включают safe_mode, там, где необходимы по-настоящему эффективные мероприятия

Кроме того, разработчики на других языках предпочитают программировать на привычном им языке, и не пытаются решить проблемы безопасности, которые надо решать на уровне серверного ПО или конфигурации сервера.

Обманчивая природа safe_mode вводит в заблуждение админов, которые идут по пути наименьшего сопротивления и просто включают safe_mode, там, где необходимы по-настоящему эффективные мероприятия. А система и пользовательские данные доступны практически любому начинающему взломщику.

Все эти проблемы, в конечном счёте, дают очень нехороший эффект: пользователи, столкнувшиеся с ними, начинают заваливать техподдержку провайдера письмами и вопросами, а это стоит денег провайдеру, а пользователям, чьи приложения зачастую неправильно работают под safe_mode, такая забота приносит много разочарований.

Что и побуждает их в конце концов перейти к провайдеру, который safe_mode не включил. PHP-разработчику тоже жизнь мёдом не покажется: постоянное домогательство со стороны админов и заказчиков, которым нужно, чтобы всё работало, причём правильно, в режиме safe_mode.

Тогда где же стоит искать решения проблемы с доступам к файлам на разделяемых хостах? Конечно, лучшее решение – это каждому пользователю предоставить виртуальный сервер, где он и будет администровать.

Этот способ создаёт для каждого пользователя систему отдельную и закрытую от соседей по серверу. Кроме того, виртуальные сервера показали себя как очень эффективное средство в отношении ресурсопотребления, поскольку пользователи склонны запускать одно и то же ПО, то есть память используется эффективно.

Ещё одно средство: использование CGI или более эффективного Fast-CGI в связке с правильными разрешениями (umask), которые не будут давать пользовательским файлам права на чтение любых других файлов на сервере и выставлять группу Apache в качестве группы-владельца. Средство попроще - это назначить всем пользовательским директориям разрешения 711, запретив, таким образом, листинг файлов в этих директориях, - это сильно затруднит несанкционированный доступ к файлам.

Ещё одно средство можно найти в самом PHP, называется оно open_basedir. Эта INI-директива позволяет вам ограничить доступ конкретному пользователю к ряду директорий.

Оригинал статьи: http://ilia.ws/archives/18_PHPs_safe_mode_or_how_not_to_implement_security.html

Трудно быть богом

Автор:
Антон Калмыков

«Привет, привет, любимый мой читатель», - банальная фраза, но она очень важна. Важна, для тебя и, в то же время, очень важна для меня, потому что эта фраза устанавливает «контакт», без которого наш последующий диалог просто теряет весь смысл. Ты программист? Нет? Да? Не можешь сказать? Сомневаешься? Ты в раздере, ты в творчестве! Ты на пути и я тоже. И разницы нет, прошел ли я больше тебя, или ты уже в дебрях, а я еще только карабкаюсь на гору. Просто мы идем - каждый своим путем. У нас много общего и много разного, ЛИЧНОГО, ВЫСТРАДАННОГО, ЛЮБИМОГО, и, каждый из нас будет стоять на своем, если чувствует, что прав!

Начинать любое дело всегда очень трудно. «Трудность» подкрадывается незаметно и в самый неподходящий момент. И не важно, кем ты на этот момент являешься: новичком, или профи. Перед обеими категориями специалистов, как это ни странно, встают одни и те же жизненные проблемы. Проблемы «нужности», профессионализма, заплаты, удовлетворенности, и.т.д. и.т.п. Мир не меняется, меняемся мы.

Но что же заставляет нас «искать приключения», стремиться к недостижимому и, в то же время, не терять самоуважения и веры в себя? Почему тебе и мне просто жизненно необходимо доказывать всем окружающим, что «ты чего-то стоишь» независимо от зарплаты и положения в обществе, зачем мы стремимся к известности, почему так важно заработать имя и что же это дает? На эти вопросы я постараюсь ответить в следующий раз. Я хочу открыть этот цикл статей самой животрепещущей, на мой взгляд, темой: «Кто же мы – программисты?» Я искренне надеюсь, что еще не утомил тебя своими сантиментами, но что-то подсказывает мне, что в глубине своей души, ты уже задавал себе этот вопрос, и тоже хочешь понять: «Кто же я – программист?»

Давай вспомним свою первую программу. Что ты писал или что ты писала? Алгоритм шифрования? Гениальную игру? Банально, но я уверен, что путь любого программиста начинается с программы «Hello world». Затем эта программа становится еще круче. Мы с трепетом добавляем ввод переменной «name» (кто с «клавы», а кто и с dialog box, а самые гениальные с input stream), и балдеем от того, что теперь наша программа узнает нас по имени. Нам нужна власть. Мы пьем ее сначала из «Hello world», затем из чувства иллюзорности этой власти и гораздо позже из того, что ты властен над IT миром и одновременно никто в нем.

Я чувствую врыв негодования. Никто! Да, никто! Чем дальше в лес, тем «толще чаща знаний». Откровение ошарашивает, пора делать выбор. Готовы ли мы его сделать. Это так страшно! Но это иллюзия! Мы вольны выбирать новые пути. Это доступно всем: новичкам и профессионалам. Скучно писать одно и то же. Скучно обсуждать одни и те же темы и «меряться письками».

Тем временем...

Для чего необходимы постоянные соединения (persistent connections) с базами данных?

Они позволяют избегать траты системных ресурсов на переподключения к БД (особенно это актуально, если данный процесс медленный), однако необходимо учитывать, что на поддержку постоянного подключения так же требуются ресурсы.

PHP позволяет получать статус соединения с клиентом. Существует три значения статуса соединения:

– 0 NORMAL
– 1 ABORETD
– 2 TIMEOUT

По умолчанию, выполнение скрипта прерывается, если нарушено соединение с клиентом (ABORTED). Если скрипт вышел за пределы отведенного ему времени, активируется флаг TIMEOUT. С помощью функции connection_status () можно контролировать логику скрипта в зависимости от активного флага соединения. Пример:

```
if(connection_status()==0)
{  //Соединение есть }
else {  //Соединение потеряно }
```

Подробнее см.
http://ru2.php.net/manual/ru/features.connection-handling.php

К этому приходят все, но иллюзия сильна! ТРУДНО СДЕЛАТЬ СВОЙ ВЫБОР! Но мы его делаем! Со скрипом, с проскальзыванием, с восторгом, депрессией и комплексами мы продолжаем творить свои программы и они начинают жить своей жизнью.

Мы боги внутри себя. Но как же это трудно признаться в том, что, порой, новичек лучше тебя знает ту или иную технологию, что кто-то более успешен, чем ты. И это тоже иллюзия, потому что программирование – это идеальная свобода самовыражения. Мы вольны творить то, что нам вдумается. Со знанием приходит понимание бессилия технологий перед интеллектом.

Все что может быть придумано одним человеком - может быть понято и использовано другим. Ты еще не хакер? Ты им будешь! Но это не цель, это просто ступенька на пути в неизвестность. Она так же будет пройдена. Так угодно нам - богам. Так устроен мир и боги тоже смертны. Трудно быть богом, для самого себя! Трудно выставить планку и ее перепрыгнуть. Но: «Суслика видишь? Нет? И я нет. А он есть!»

В погоне за «сусликом» мы создаем все более ужасных новых монстров, но они не способны его поймать, потому что суслик прост. Мы бежим за ним и приходим к простоте, к чистой кристальной простоте экстремального программирования, нам нужен адреналин, нам нужен экстрим. Час, день, месяц, год – время капает: кап, кап, кап. Усталость берет свое, но божек свербит: «Ты один такой! Ты единственный, гениальный и неповторимый! Пора быть собой!» И это тоже трудно, поскольку ответ на вопрос: «Кто я?», - мало кого приводит в чувство равновесия. Приходит время сбирать камни, но только для того, чтобы разбросать их снова. Собственным богом быть трудно.

Осознание получения удовольствия от творчества почему-то приходит только тогда, когда наступает творческий кризис. Ты еще пишешь как сумасшедший, испытывающий катарсис от процесса? Тогда ты счастлив! Ты всесилен и весь мир у твоих ног! Ты полон сил и планов на будущее. Ты уже завершил проект! Ты еще не веришь в его завершение, но он готов. Ты гений! Ты сделал это. «Люди! Я сделал это!», - кричишь ты. Кричи, объясняй, толкай, но будь готов к разочарованиям и потерям. Нет, не к потерям материальным, скорее к потерям моральным и психологическим. Ты потеряешь тот кайф от божественного всесилия и власти и захочешь вернуть его, но это тоже трудно. Пора что-то менять. Меняй! Ты бог. Но как же трудно что-то изменить в самом себе!

И, проговаривая свои мысли, я ловлю себя на том, что я начинаю сомневаться в себе. Так ли я одинок в этом? Эта статья – пробный камень в твою душу, - программер. Мне очень хочется верить, что я затронул хоть одну струнку в ней, а если так, то жду feedback в форум PHP клуба. Трудно быть богом, но «Не боги горшки обжигают». Быть или не быть следующей статье - решать тебе. Я жду и я надеюсь.

Тем временем...

Если у вас есть доступ к конфигурационному файлу php.ini, вы можете запретить выполнение любых PHP-функций с помощью директивы disable_functions. Пример:

```
disable_functions readfile,system
```

Если вы подключаете к скрипту внешние файлы (include, require...) следите за тем, чтобы они были недоступны посетителям сайта. Например, если вы в корневой директории веб-сервера создали файл db.inc и записали в него пароли и логины доступа к БД, то этот файл можно запросто скачать набрав www.yourserver.com/db.inc. Во избежание этого, храните включаемые файлы вне DOCUMENT_ROOT веб-сервера или в конфигурационном файле apache укажите:

```
<Files ~ "\.inc$">
Order allow,deny
Deny from all
</Files>
```

Если вы используете Smarty и в одном из PHP-файлов у вас определена константа, то ее значение можно подставить в шаблон с помощью массива $smarty.const. Пример:

PHP-файл:

```
define(PATH_ADMIN, "c:/web/admin");
```

Шаблон:

```
{$smarty.const.PATH_ADMIN}
```

Форум

В этом номере журнала, редакция решила пойти на некоторого рода эксперимент и «добавить жизни» в материалы, предлагаемые уважаемому читателю. Что значит «оживить», - возможно спросите вы? А это значит – добавить информацию не только от авторов статей и переводов, но и предложить на суд читателя немного переработанные топики из «живых» форумов о PHP. Просим вас откликнуться на это нововведение и высказать свое мнение – нужна ли эта рубрика в дальнейшем.

Как любое суждение в форуме, предлагаемые здесь решения могут быть спорны, но их предназначение не столько «учить с позиции Гуру», сколько предложить некоторую идею и рассказать о том, какие вопросы обсуждаются в форумах и какие предлагаются ответы (оговорюсь, что предлагаемый здесь список не является абсолютно исчерпывающим, есть вопросы более сложные и неоднозначные).

В подготовке рубрики для этого номера участвовали разработчики из двух ведущих отечественных PHP-сообществ: http://phpclub.ru и http://php.com.ua. Редакция благодарит всех тех, кто помогал нам подбирать и перерабатывать информацию и тех, кто активно участвует в работе двух вышеназванных сообществ – благодаря вам, любимая нами технология находит применение и любовь в наших странах.

По материалам:
http://phpclub.ru
http://php.com.ua

Как сравнить два файла (с точностью до байта)?

- Используй функцию strcmp(). вот ее описание:

```
int strcmp ( string str1, string str2)
```

Возвращает положительное число если str1 меньше, чем str2; отрицательное число если str1 больше, чем str2, и 0 если строки равны.

- Еще один вариант, возможно не самый быстрый:

```
$fd = fopen("1.txt",'r');
$fd2 = fopen("2.txt",'r');
$i = 1;
while(!feof($fd) or !feof($fd2))
{       $st1 = fgetc($fd);
        $st2 = fgetc($fd2);
        if ($st1 != $st2)
        { print "symbol $i is different!\n";
        }
        $i++;
}
```

- Еще вариант ....

file_get_contents - эта функция нормально работает с данными в бинарном формате, единственное - она считывает весь файл целиком сразу:

```
<?php
$st1 = file_get_contents("1.txt");
$st2 = file_get_contents("2.txt");
for($i=0;$i < strlen($st1);$i++)
{
        if ($st1[$i] != $st2[$i])
        {
                echo "symbol ",$i+1," is different!\n";
        }
}
?>
```

- Еще один из вариантов

```
if (md5(file_get_contents('1.txt'))!= md5(file_get_contents('2.txt'))) die
('Files are not identical');
```

Для более продвинутого анализа используйте Unix: man diff, cmp, sdiff, diff3.

- Чтобы узнать, равны ли два файла, можно использовать следующий способ:

```
if(sha1_file($file1) == sha1_file($file2)) { ... }
```

Как можно конвертировать строку UTF-8 в кириллицу win-1251?

- Если PHP собран с поддержкой iconv, то

```
<?php
echo iconv("UTF-8", "cp1251", "This is a test.");
?>
```

- Еще вариант:

```
function utf8win1251($s){
    $out = $c1 = "";
    $byte2=false;
    for ($c=0;$c<strlen($s);$c++){
        $i=ord($s[$c]);
        if ($i<=127) $out.=$s[$c];
        if ($byte2){
        $new_c2=($c1&3)*64+($i&63);
        $new_c1=($c1>>2)&5;
        $new_i=$new_c1*256+$new_c2;
        if ($new_i==1025){ $out_i=168; }
```

```
        else{
             if ($new_i==1105){
             $out_i=184; }
             else {
             $out_i=$new_i-848;
             }
         }
         $out.=chr($out_i);
         $byte2=false;
    }
    if (($i>>5)==6) {
    $c1=$i;
    $byte2=true;
    }
    }
return $out;
}
```

Как сгенерировать последовательность символов определенной длинны?

- Первый вариант:

```
<?php
    $strlen = 10; // длинна сгенерированной строки
    // массив с буквами которые должны быть в строке
    $chars = array("a","b","c","b","e","1","2","3","4","5","6");
    for ($i=0;$i<=$strlen;$i++)
        $str.= $chars[mt_rand(1,count($chars))-1];
        echo $str; // Результат
?>
```

- Приведенный выше вариант можно немного модифицировать с помощью range() и привести формирование исходного массива с буквами к форме: $chars=range(0..9)+range(a..z)+whatever
- Очевидно, что одной из целей решения данной задачи является генерация паролей. Что если необходимо сгенерировать не просто пароль, а пароль который проще запомнить? Здесь поможет генерация строки с чередованием гласных и согласных.

```
$passlen=6;
$vowels=array('a','e','i','o','u');
    //просто лень перечислять все согласные руками :-)
foreach(range('a','z') as $c) if (!in_array($c,$vowels)) $consonants[]=$c;
    while($i++ <= $passlen/2) $pass.=$consonants[array_rand($consonants)].
$vowels[array_rand($vowels)];
echo $pass;
```

- Про готовые решения можно прочитать тут:

  http://pear.php.net/manual/ru/package.text.text-password.types.php

http://pear.php.net/package/Text_Password

Вывод даты в виде «Понедельник, 26 ноября, 2001»

Так как скрипт выполняется на сервере, то и время он будет выдавать серверное. Для выдачи даты посетителя сайта, используйте JavaSCript.

- Читаем тут про дату
  http://www.php.net/manual/en/function.date.php
- Или вот пример кода:

```
<?php
$weekdays = array('Воскресенье', 'Понедельник', 'Вторник', 'Среда',
'Четверг', 'Пятница', 'Суббота');

$months = array('января', 'февраля', 'марта', 'апреля', 'мая', 'июня',
'июля', 'августа', 'сентября', 'октября', 'ноября', 'декабря');

$weekday = date(w);
$month = date(m) - 1;
echo date("$weekdays[$weekday], d $months[$month], Y");
?>
```

Чтение файла «без ошибок»

Что может быть проще, чем работа с файлом, спросите вы? Открыл функцией fopen и пиши-читай. Но многие забывают что файл может не существовать или быть недоступен для чтения или записи. Поэтому, работающая на первый взгляд программа, может в один момент стать неработающей, стоит только поменять права доступа к файлу. Мало того, многие ставят перед функцией fopen() оператор @ и лишают себя последней возможности вовремя среагировать на ошибку, а именно увидеть её в логах.

Вот один из примеров правильного чтения файла:

```
<php

$file = 'file.txt';
    if (is_readable($file) && $fp = fopen($fp,'r'))
    {
        while ($str = fgets($fp,1024))
        {
        print $str.'<br>';
        }
    }
    else
    { die('Файл не найден, или не доступен для чтения');
    }
?>
```

Этот код, можно использовать как шаблон, для любой программы, где происходит работа с файлами.

Вот краткий список функций, которые помогут вам безошибочно работать с файлами.

- **is_file** - проверяет, что указанный файл существует и это действительно файл;
- **is_dir** - проверяет, что указанный путь существует и это директория;
- **file_exists** - проверяет существование файла;
- **is_readable** - проверяет что файл существует и доступен для чтения;
- **is_writable** - проверяет что файл существует и доступен для записи;
- **is_uploaded_file** - проверяет что файл существует и был закружен по протоколу HTTP.

Обрезание текста до заданной длины

Очень часто в интнете встречается текст такого вида:

```
PHP (рекурсивный акроним для "PHP: Hypertext Preprocessor") это широко
распространённый Открытый ресурс - язык скриптинга ... можно внедрять в
HTML.
```

Этот текст обрезан до заданной длины. Причем, как видите, текст обрезан не с конца и даже не с середины. Как самостоятельно можно сделать нечто подобное? Это давольно легко. Потребуется лишь знания математики и начальные навыки программирования на PHP.

Итак, напишем функцию, которая бедет обрезать текст до заданой длины, вставляя вместо вырезаного текста, например, троеточие. Псевдокод (ориентированный на PHP):

```
Для начала нужно вырезать все теги из текста, чтобы не возникло ситуации,
когда в тексте останется незакрытый тег. В PHP все теги из текста вырезает
функция strip_tags.

Далее получаем длину строки.

Теперь нужно получить количество символов которое останется в левой части.
Для этого нужно применить формулу: left_len = max_len*percent/100, где
max_len - максимальная длина строки percent - смещение вырезаемой строки,
в процентах, от начала.

Получаем позицию символа, с которого начинается правая часть строки:
right_pos = str_len-(max_len-left_len), где str_len - длина исходной
строки.
Наконец, остается только, с помощью функции substr, получить левую часть
строки, начиная от нулевого символа, и заканчивая left_len, и правую
часть, начиная от right_pos и заканчивая str_len.
```

Вот, примерно такая функция может получиться:

```php
<?php

function str_cut($str,$max_len=50,$percent=50,$substitute='...')
{
$str = (strip_tags($str));
$str_len = strlen($str);
$max_len = $max_len-strlen($substitute);
    if ($str_len > $max_len && $max_len)
    {
    $lstr = substr($str,0,$max_len*$percent/100);
    $rstr = substr($str,$str_len-($max_len-strlen($lstr)),$str_len);
    return $lstr.$substitute.$rstr;
    }
return $str;
}

?>
```

Для заметок: