Введение в PHP

Введение в PHP 1
Простейший сценарий 1
Скалярные переменные 2
Массивы 3
Хэши 3
Условные операторы 4
Операторы циклов 5
Функции, определяемые пользователем (пользовательские) 6
Инструкции require и include 10
Регулярные выражения 11
Строковые функции 15
PHP и HTTP: headers 19
PHP и HTTP: cookie 21
Сессии 23
Работа с файлами 25

Простейший сценарий

В PHP код сценария должен находиться между открывающим тегом <? и закрывающим тегом ?>. Расширение файлов должно быть – php, phtml, phtm, при этом файлы могут находиться в любой директории на сервере. Напишем простейший сценарий, который выводит строку «Hello world!» полужирным начертанием:

```
<?
echo "<b>Hello world!</b>"
?>
```
Hello world!

Функция echo служит для вывода информации на экран (в броузер). В данном примере и – это обычные теги html. В принципе, данный сценарий может выглядеть как

```
<b>Hello world!</b>
```
Hello world!

Таким образом, без указания специальных тегов (<? и ?>), сценарии, написанные на языке PHP являются обычными html-документами.

Примечание

Курсивом в конце примера скрипта выделяется результат работы данного скрипта.

Комментарии в PHP

Однострочные комментарии в PHP начинаются с символа # или пары символов //. PHP игнорирует весь текст, идущий от этих символов до конца строки. Также поддерживаются C-подобные комментарии, начинающиеся с /* и заканчивающиеся */. Эти комментарии могут занимать больше, чем одну строку:

```
<?
// это комментарий
# это тоже комментарий
/* а этот комментарий состоит из нескольких строк,
здесь можно писать все, что угодно */
?>
```

Скалярные переменные

Все скалярные переменные в PHP должны начинаться с символа **$**, например:

```
$foo = 5;
```

В данном случае, переменной $foo присвоили числовое значение 5.

Важно
Имена скалярных переменных чувствительны к регистру: имя $foo – это не то же самое, что имя $Foo.

Работа со строками

В скалярных переменных можно хранить как числа, так и строки:

```
$foo = "Hello";
```

Для объединения строк используется оператор конкатенации, в роли которого используется точка (.):

```
$foo = "Hello ";
$biz = "there\n";
echo $foo . $biz;
```
Hello there

Строки могут задаваться с помощью одинарных или двойных кавычек:

```
$foo = "Hello";
$biz = 'there';
```

Между этими способами есть различие. Если строка ограничена двойными кавычками, то PHP вычисляет переменные, встречающиеся в данной строке. Если строка ограничена одинарными кавычками, то PHP выводит данную строку как есть.

В строках, ограниченных двойными кавычками, можно использовать escape-последовательности, управляющие их форматированием и позволяющие задавать символы, которые иначе записать не удастся. Например, чтобы внести в текст двойную кавычку, можно использовать последовательность символов \":

```
echo "I said \"Hello\".";
```

Подстановка переменных (интерполяция строк)

При использовании в строке, заключенной в двойные кавычки, имена переменных, PHP подставляет вместо них значения присвоенные переменным. Например, если есть переменная $text, в которой храниться слово Hello:

```
$text = "Hello";
```

то можно использовать её имя в теле строки, и PHP подставит Hello вместо имени переменной:

```
$text = "Hello";
echo "PHP says: $text!\n";
```
PHP says: Hello!

Однако если заключить тело строки в одинарные кавычки (апострофы), то PHP не будет выполнять интерполяцию:

```
$text = "Hello";
echo 'PHP says: $text!';
```
PHP says: $text!

Если в строке идет переменная, а следом сразу следует слово, то для обозначения переменной её следует заключать в фигурные скобки:

```
$text = "un";
echo "Don't be ${text}happy.";
```
Don't be unhappy.

Массивы

Массивы в PHP задаются следующим образом:

```
$a = array (1, 2, 3);
```

Таким образом, был создан массив, состоящий из трех элементов. Чтобы ссылаться на отдельные элементы массива, следует указать индекс элемента в квадратных скобках. Следует обратить внимание, что индексы для массивов отсчитываются от нуля:

```
echo $a[1];
```
2

Общее количество элементов в массиве определяется с помощью функции count():

```
$total=count($a);
echo $total;
```
3

Хэши

Хэш-таблицы (хэшированные таблицы, или просто хэши), называются также ассоциативными массивами для доступа к отдельным элементам данных, используют не индексы, а ключи. Хеши в PHP задаются следующим образом:

```
$a = array ("fruit"=>"apple", "sandwich"=>"hamburger", "drink"=>"bubbly");
```

Теперь можно использовать эти данные, применив для доступа к ним ключ

```
echo $a["sandwich"];
```
hamburger

Можно создавать новые элементы в массиве:

```
$a["car"] = "Ford";
```

И менять значения уже существующих элементов:

```
$a["fruit"] = "orange";
```

Условные операторы

Условный оператор if-else

Условный оператор if записывается в формате:

```
if(логическое_выражение)
        инструкция 1;
else
     инструкция_2 ;
```

Действие его следующее: если логическое_выражение истинно, то выполняется инструкция_1, а иначе – инструкция_2. Как и в любом другом языке, конструкция else может опускаться, в этом случае при получении должного значения просто ничего не делается. Пример:

```
if($a>=1&&$b<=10) echo "Все ОК";
else echo "Неверное значение в переменной!";
```

Если инструкция_1 или инструкция_2 должны состоять из нескольких команд, то они, как всегда, заключаются в фигурные скобки. Например:

```
if($a>$b) { print "a больше b"; c=$b; }
elseif($a==$b) { print "a равно b"; $c=$a; }
else { print "a меньше b"; $c=$a; }
```

Конструкция if-else имеет еще один альтернативный синтаксис:

```
if(логическое_выражение):
        команды;
elseif(другое_логическое_выражение):
        другие команды;
else:
        иначе команды;
endif
```

Конструкция switch-case

Часто вместо нескольких расположенных подряд инструкций if-else целесообразно воспользоваться специальной конструкцией switch-case:

```
switch(выражение) {
     case значение1: команды1; [break;]
     case значение2: команды2; [break;]
     . . .
     case значениеN: командыN; [break;]
     [default: команды_по_умолчанию; [break]]
}
```

Делает она следующее: вычисляет значение выражения (пусть оно равно, например, V), а затем пытается найти строку, начинающуюся с case V:. Если такая строка обнаружена, выполняются команды, расположенные сразу после нее (причем на все последующие операторы case что_то внимание не обращается, как будто их нет, а код после них остается без изменения). Если же найти такую строку не удалось, выполняются команды после default (когда они заданы).

Следует обратить внимание на операторы break (которые условно заключены в квадратные скобки, чтобы подчеркнуть их необязательность), добавленные после каждой строки команд, кроме последней (для которой можно было бы тоже указать break, что не имело бы смысла). Если бы не они, то при равенстве V=значение1 сработали бы не только команды1, но и все нижележащие.

Операторы циклов

Цикл с предусловием while

Эта конструкция унаследована непосредственно от Си. Ее предназначение – цикличное выполнение команд в теле цикла, включающее предварительную проверку, нужно ли это делать (истинно ли логическое выражение в заголовке). Если не нужно (выражение ложно), то конструкция заканчивает свою работу, иначе выполняет очередную итерацию и начинает все сначала. Выглядит цикл так:

```
while(логическое_выражение) инструкция;
```

где, как обычно, логическое_выражение – логическое выражение, а инструкция – простая или составная инструкция тела цикла. (Очевидно, что внутри последнего должны производиться какие-то манипуляции, которые будут иногда изменять значение логического выражения, иначе оператор зациклится. Это может быть, например, простое увеличение некоего счетчика, участвующего в выражении, на единицу.) Если выражение с самого начала ложно, то цикл не выполнится ни разу. Например:

```
$i=1; $p=1;
while($i<32) { echo $p," ";
$p=$p*2; // можно было бы написать $p*=2
$i=$i+l; // можно было бы написать $i+=l или даже $i++
}
```

Данный пример выводит все степени двойки до 31-й включительно.

Цикл с постусловием do-while

В отличие от цикла while, этот цикл проверяет значение выражения не до, а после каждого прохода. Таким образом, тело цикла выполняется хотя бы один раз. Выглядит оператор так:

```
do {
        команды;
} while(логическое_выражение);
```

После очередной итерации проверяется, истинно ли логическое_выражение, и, если это так, управление передается вновь на начало цикла, в противном случае цикл обрывается.

Цикл for

Формат конструкции такой:

```
for(инициализирующие_команды; условие_цикла; команды_после_прохода)
тело цикла;
```

Работает он следующим образом. Как только управление доходит до цикла, первым делом выполняются операторы, включенные в инициализирующие_команды (слева направо). Эти команды перечисляются там через запятую, например:

```
for($i=0,$j=10,$k="Test"; ......)
```

Затем начинается итерация. Первым делом проверяется, выполняется ли условие_цикла (как в конструкции while). Если да, то все в порядке, и цикл продолжается. Иначе осуществляется выход из конструкции. Например:

```
// прибавляем по одной точке
for($i=0,$j=0,$k="Test"; $i<10; .....) $k.=".";
```

Предположим, что тело цикла проработало одну итерацию. После этого вступают в действие команды_после_прохода (их формат тот же, что и у инициализирующих операторов). Например:

```
for($i=0,$j=0,$k="Points"; $i<100; $j++,$i+=$j) $k=$k.".";
```

Функции, определяемые пользователем (пользовательские)

Функция может быть определена с использованием такого синтаксиса:

```
function foo ($arg_1, $arg_2, ..., $arg_n)
{
        echo "Пример \n";
        return $retval;
}
```

Внутри функции может появляться любой правильный код PHP, даже другие функции и определения классов. В PHP (версии 4 и выше) нет такого требования, чтобы функции были определены до обращения к ним. В PHP 4 можно задавать переменное количество аргументов функции, а также поддерживаются аргументы по умолчанию.

Аргументы функции

Информация может передаваться в функцию через список аргументов, который является списком разделённых запятыми переменных и/или констант.

PHP поддерживает разбор аргументов по значению (по умолчанию), разбор по ссылке и значения по умолчанию.

Значения аргументов по умолчанию

Функция может определить значения по умолчанию в стиле C++ для скалярных аргументов:

```
function makecoffee ($type = "cappuccino") {
    return "Making a cup of $type.\n";
}
echo makecoffee ();
echo makecoffee ("espresso");

// Вывод будет таким:
// Making a cup of cappuccino.
// Making a cup of espresso.
```

Значение по умолчанию обязано быть константным выражением, а не (например) переменной или членом класса.

Обратите внимание, что, при использовании аргументов по умолчанию, любые значения по умолчанию должны находиться справа от любых значений не по умолчанию; иначе что-нибудь может работать не так, как ожидалось. Рассмотрим следующий фрагмент кода:

```
function makeyogurt ($type = "acidophilus", $flavour) {
    return "Making a bowl of $type $flavour.\n";
}
echo makeyogurt ("raspberry");   // не будет работать так, как ожидается
```

Вывод этого примера:

```
Warning!: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/php3test/functest.html on line 41
Making a bowl of raspberry.
```

Теперь сравним с этим:

```
function makeyogurt ($flavour, $type = "acidophilus") {
    return "Making a bowl of $type $flavour.\n";
}
echo makeyogurt ("raspberry");   // работает как положено

//На выводе этого примера:
//Making a bowl of acidophilus raspberry.
```

Списки аргументов переменного размера

В PHP 4 имеется поддержка списков аргументов переменного размера/variable-length в пользовательских функциях. Это довольно легко делается функциями func_num_args(), func_get_arg() и func_get_args() (Описание этих функций вы можете найти здесь – http://web.php.net.ua/manual/php4/index.functions.html).

Специального синтаксиса не нужно, и списки аргументов могут по-прежнему явно предоставляться в определениях функций и будут вести себя как обычно.

Возвращаемые значения

Значения из функций возвращаются с помощью необязательного оператора return. Может быть возвращён любой тип, в том числе список и объект. Этот оператор немедленно останавливает выполнение функции и передаёт управление обратно на строчку, с которой функция была вызвана.

```
function square ($num) {
    return $num * $num;
}
echo square (4);
// выводит '16'
```

Вы можете возвратить из функции несколько значений, но сходные результаты можно получить путём возвращения списка.

```
function small_numbers() {
    return array (0, 1, 2);
}
list ($zero, $one, $two) = small_numbers();
```

Области видимости переменных. Локальные и глобальные переменные

В приводимых выше примерах мы рассматривали аргументы функции как некие временные объекты, которые создаются в момент вызова и исчезают после окончания функции. Например:

```
$a=100; // Глобальная переменная, равная 100
function Test($a) {
        echo $a; // выводим значение параметра $a
        // Этот параметр не имеет к глобальной $a никакого отношения!
        $a++; // изменяется только локальная копия значения, переданного в $a
}
Test(1); // выводит 1
echo $a; // выводит 100 - глобальная $a, конечно, не изменилась
```

В действительности такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции. Вот пример:

```
function Silly() {
        $i=rand(); // записывает в $i случайное число
        echo $i; // выводит его на экран
        // Эта $i не имеет к глобальной $i никакого отношения!
}
for($i=0; $i!=10; $i++) Silly ();
```

Здесь переменная $i в функции будет не той переменной $i, которая используется в программе для организации цикла. Поэтому, собственно, цикл и проработает только 10 «витков», напечатав 10 случайных чисел (а не будет крутиться долго и упорно, пока «в рулетке» функции rand( ) не выпадет 10).

Собственно говоря, это даже хорошо. Действительно, мало ли какие имена переменных использует функция для своих личных целей... Какое до этого дело программе (которая вообще может быть написана другим человеком)?

Однако, разумеется, в PHP есть способ, посредством которого функции могут добраться и до любой глобальной переменной в программе (не считая, конечно, передачи параметра по ссылке). Однако для этого они должны проделать определенные действия, а именно: до первого использования в своем теле внешней переменной объявить ее «глобальной»:

```
function Silly()
{
     global $i;
     $i=rand();
     echo $i;
}
for($i=0; $i!=10; $i++) Silly();
```

Вот теперь-то переменная $i будет везде едина: что в функции, что во внешнем цикле (для последнего это приведет к немедленному его "зацикливанию", во всяком случае, на ближайшие несколько минут, пока rand( ) не выкинет 9). А вот еще один пример, который показывает удобство использования глобальных переменных внутри функции:

```
$Monthes[1]="Январь";
$Monthes[2]="Февраль";

. . . и т. д.
$Monthes[12]="Декабрь";

function GetMonthName ($n) {
     global $Monthes;
     return $Monthes [$n];
}
echo GetMonthName (2); // выводит "Февраль"
```

Согласитесь, массив $Monthes, содержащий названия месяцев, довольно объемист. Поэтому описывать его прямо в функции было бы, мягко говоря, неудобно. В то же время функция GetMonthName( ) представляет собой довольно приемлемое средство для приведения номера месяца

к его словесному эквиваленту (что может потребоваться во многих программах). Она имеет единственный и понятный параметр: это номер месяца.

Статические переменные

Видимо, чтобы не отставать от других языков, создатели PHP предусмотрели еще один вид переменных, кроме локальных и глобальных, – статические. Работают они точно так же, как и в Си. Рассмотрим следующий пример:

```
function Silly() {
        static $a=0;
        echo $a;
        $a++;
}
for($i=0; $i<10; Si++) Silly ();
```

После запуска будет выведена строка 0123456789. Если убрать слово static, то будет выведено: 0000000000. Это и понятно, ведь переменная $a стала локальной, и ей при каждом вызове функции присваивается одно и то же значение – 0.

Итак, конструкция static говорит компилятору о том, что уничтожать указанную переменную для нашей функции между вызовами не надо. В то же время присваивание $a=0 сработает только один раз, а именно – при самом первом обращении к функции.

Функции переменных

PHP поддерживает концепцию функций переменных (variable functions) – динамические вызовы функций, имена которых определяются во время выполнения программы. Хотя в большинстве web-приложений можно обойтись и без функций-переменных, они значительно сокращают объем и сложность программного кода, а также часто снимают необходимость в условных командах if.

Вызов функции-переменной представляет собой имя переменной, за которым следует пара круглых скобок. В круглых скобках могут перечисляться параметры (однако присутствие параметров не обязательно). Обобщенный синтаксис функции-переменной:

```
$имя_функции( );
```

Следующий пример демонстрирует эту непривычную, но полезную возможность. Допустим, программа выводит разную информацию в зависимости от языка, выбранного пользователем. В нашем примере для простоты используются приветственные сообщения для англо- и русскоязычных пользователей:

Переменная $language используется для выполнения функции-переменной (в приведенном примере – russian()).

```
// Приветствие на русском языке,
function russian( ) {
        print "Добро пожаловать.";
}
// Приветствие на английском языке
function english( ) {
        print "Welcome.";
}

// Выбор русского языка
$language = "russian":

// Выполнить функцию-переменную
$language( );
```

Этот пример демонстрирует интересную концепцию функций-переменных и наглядно показывает, что функции-переменные способствуют уменьшению объема программного кода. Если бы не эта возможность, функцию пришлось бы выбирать командой if или switch; это привело бы к заметному увеличению объема программного кода и риску появления дополнительных ошибок при кодировании.

Инструкции require и include

require

Эта инструкция позволяет разбить текст программы на несколько файлов. Ее формат такой:

```
require имя_файла;
```

При запуске (именно при запуске, а не при исполнении!) программы интерпретатор просто заменит инструкцию на содержимое файла имя_файла (этот файл может также содержать сценарий на PHP, обрамленный, как обычно, тэгами <? и ?>). Причем сделает он это только один раз (в отличие от include, который рассматривается ниже): а именно, непосредственно перед запуском программы. Это бывает довольно удобно для включения в вывод сценария всяких «шапок» с HTML-кодом. Например:

Файл header.htm
```
<html>
<head><title>Title! </title></head>
<body bgcolor=yellow>
```

Файл footer.htm
```
&copy;My company, 2006. </body></html>
```

Файл script.php
```
<?
require "header.htm" ;
. . . работает сценарий и выводит само тело документа
require "footer.htm";
?>
```

Безусловно, это лучше, чем включать весь HTML-код в сам сценарий вместе с инструкциями программы.

include

Эта инструкция практически идентична require, за исключением того, что включаемый файл вставляется сценарий не перед его выполнением, а прямо во время.

Например, пусть есть 10 текстовых файлов с именами file0.php, file1.php и так далее до file9.php, содержимое которых просто десятичные цифры 0, 1 ... 9 (по одной цифре в каждом файле). Запустим такую программу:

```
for($i=0; $i<10; $i++) {
    include "file$i.php";
}
```

В результате получим вывод, состоящий из 10 цифр: "0123456789". Из этого можно заключить, что каждый из данных файлов был включен по одному разу прямо во время выполнения цикла.

Регулярные выражения

Регулярные выражения лежат в основе всех современных технологий поиска по шаблону. Регулярное выражение представляет собой последовательность простых и служебных символов, описывающих искомый текст. Иногда регулярные выражения бывают простыми и понятными (например, слово dog), но часто в них присутствуют служебные символы, обладающие особым смыслом в синтаксисе регулярных выражений, – например, <(?)>.*<\/.?>.

В PHP существуют два семейства функций, каждое из которых относится к определенному типу регулярных выражений: в стиле POSIX или в стиле Perl. Каждый тип регулярных выражений обладает собственным синтаксисом. Ниже будут приведены лишь основные сведения о регулярных выражениях типа POSIX.

Синтаксис регулярных выражений(POSIX)

Структура регулярных выражений POSIX чем-то напоминает структуру типичных математических выражений – различные элементы (операторы) объединяются друг с другом и образуют более сложные выражения.

Простейшее регулярное выражение совпадает с одним литеральным символом – например, выражение g совпадает в таких строках, как g, haggle и bag. Выражение, полученное при объединении нескольких литеральных символов, совпадает по тем же правилам – например, последовательность gan совпадает в любой строке, содержащей эти символы (например, gang, organize или Reagan).

Оператор | (вертикальная черта) проверяет совпадение одной из нескольких альтернатив. Например, регулярное выражение php | zend проверяет строку на наличие php или zend.

Квадратные скобки

Квадратные скобки ([ ]) имеют особый смысл в контексте регулярных выражений – они означают "любой символ из перечисленных в скобках". В отличие от регулярного выражения php, которое совпадает во всех строках, содержащих литеральный текст php, выражение [php] совпадает в любой строке, содержащей символы p или h. Квадратные скобки играют важную роль при работе с регулярными выражениями, поскольку в процессе поиска часто возникает задача поиска символов из заданного интервала. Ниже перечислены некоторые часто используемые интервалы:

- [0-9] – совпадает с любой десятичной цифрой от 0 до 9;
- [a-z] – совпадает с любым символом нижнего регистра от a до z;
- [A-Z] – совпадает с любым символом верхнего регистра от A до Z;
- [a -Z] – совпадает с любым символом нижнего или верхнего регистра от a до Z.

Конечно, перечисленные выше интервалы всего лишь демонстрируют общий принцип. Например, вы можете воспользоваться интервалом [0-3] для обозначения любой десятичной цифры от 0 до 3 или интервалом [b-v] для обозначения любого символа нижнего регистра от b до v. Короче говоря, интервалы определяются совершенно произвольно.

Квантификаторы

Существует особый класс служебных символов, обозначающих количество повторений отдельного символа или конструкции, заключенной в квадратные скобки. Эти служебные символы (+, * и {...}) называются квантификаторами. Принцип их действия проще всего пояснить на примерах:

- p+ означает один или несколько символов p, стоящих подряд;
- p* означает ноль и более символов p, стоящих подряд;
- p? означает ноль или один символ p;
- p{2} означает два символа p, стоящих подряд;
- p{2,3} означает от двух до трех символов p, стоящих подряд;
- p{2,} означает два и более символов p, стоящих подряд.

Прочие служебные символы

Служебные символы $ и ^ совпадают не с символами, а с определенными позициями в строке. Например, выражение p$ означает строку, которая завершается символом p, а выражение ^p – строку, начинающуюся с символа p.

- Конструкция [^a-zA-Z] совпадает с любым символом, не входящим в указаные интервалы (a-z и A-Z).
- Служебный символ . (точка) означает "любой символ". Например, выражение p.p совпадает с символом p, за которым следует произвольный символ, после чего опять следует символ p.

Объединение служебных символов приводит к появлению более сложных выражений. Рассмотрим несколько примеров:

- ^.{2}$ – любая строка, содержащая ровно два символа;
- (.*) – произвольная последовательность символов, заключенная между и (вероятно, тегами HTML для вывода жирного текста);
- p(hp)* – символ p, за которым следует ноль и более экземпляров последовательности hp (например, phphphp).

Иногда требуется найти служебные символы в строках вместо того, чтобы использовать их в описанном специальном контексте. Для этого служебные символы экранируются обратной косой чертой (\). Например, для поиска денежной суммы в долларах можно воспользоваться выражением \$[0-9]+, то есть "знак доллара, за которым следует одна или несколько десятичных цифр". Обратите внимание на обратную косую черту перед $. Возможными совпадениями для этого регулярного выражения являются $42, $560 и $3.

Стандартные интервальные выражения (символьные классы)

Для удобства программирования в стандарте POSIX были определены некоторые стандартные интервальные выражения, также называемые символьными классами (character classes). Символьный класс определяет один символ из заданного интервала – например, букву алфавита или цифру:

- [[:alpha:]] – алфавитный символ (aA-zZ);
- [[:digit:]] – цифра (0-9);
- [[:alnum:]] – алфавитный символ (aA-zZ) или цифра (0-9);
- [[:space:]] – пропуски (символы новой строки, табуляции и т. д.).

Функции PHP для работы с регулярными выражениями(POSIX-совместимые)

В настоящее время PHP поддерживает семь функций поиска с использованием регулярных выражений в стиле POSIX:

- ereg( );
- ereg_replace( );
- eregi( );
- eregi_replace( );
- split( );
- spliti( );
- sql_regcase( ).

Описания этих функций приведены ниже.

ereg( )

Функция ereg( ) ищет в заданной строке совпадение для шаблона. Если совпадение найдено, возвращается TRUE, в противном случае возвращается FALSE. Синтаксис функции ereg( ):

```
int ereg (string шаблон, string строка [, array совпадения])
```

Поиск производится с учетом регистра алфавитных символов. Пример использования ereg( ) для поиска в строках доменов .com:

```
$is_com = ereg("(\.)(com$)", $email);
// Функция возвращает TRUE, если $email завершается символами ".com"
// В частности, поиск будет успешным для строк
// "www.wjgilmore.com" и "someemail@apress.com"
```

Обратите внимание: из-за присутствия служебного символа $ регулярное выражение совпадает только в том случае, если строка завершается символами .com. Например, оно совпадет в строке "www.apress.com", но не совпадет в строке "www.apress.com/catalog".

Необязательный параметр совпадения содержит массив совпадений для всех подвыражений, заключенных в регулярном выражении в круглые скобки. Ниже показано, как при помощи этого массива разделить URL на несколько сегментов. Вывод элементов массива $regs:

```
$url = "http://www.apress.com";
// Разделить $url на три компонента: "http://www". "apress" и "com"
$www_url = ereg("^(http://www)\.([[:alnum:]]+\.([[:alnum:]]+)\.(com$)", $url, $regs);
if ($www_url) : // Если переменная $www_url содержит URL
        echo $regs[0]; // Вся строка "http://www.apress.com"
        echo "<br>";
        echo $regs[1]; // "http://www"
        echo "<br>";
        echo $regs[2]; // "apress"
        echo "<br>";
        echo $regs[3]; // "com"
endif;
```

При выполнении данного сценария будет получен следующий результат:

```
http://www.apress.com
http://www
apress
com
```

ereg_replace( )

Функция ereg_replace( ) ищет в заданной строке совпадение для шаблона и заменяет его новым фрагментом. Синтаксис функции ereg_replace( ):

string ereg_replace (string шаблон, string замена, string строке)

Функция ereg_replace( ) работает по тому же принципу, что и ereg( ), но ее возможности расширены от простого поиска до поиска с заменой. После выполнения замены функция возвращает модифицированную строку. Если совпадения отсутствуют, строка остается в прежнем состоянии. Функция ereg_replace( ), как и ereg( ), учитывает регистр символов. Ниже приведен простой пример, демонстрирующий применение этой функции:

```
$copy_date = "Copyright 1999";
$copy_date = ereg_replace("([0-9]+)", "2000", $copy_date);
print $copy_date; // Выводится строка "Copyright 2000"
```

У средств поиска с заменой в языке PHP имеется одна интересная возможность – возможность использования обратных ссылок на части основного выражения, заключенные в круглые скобки. Обратные ссылки похожи на элементы необязательного параметра-массива совпадения функции ereg( ) за одним исключением: обратные ссылки записываются в виде \0, \1, \2 и т. д., где \0 соответствует всей строке, \1 – успешному совпадению первого подвыражения и т. д. Выражение может содержать до 9 обратных ссылок. В следующем примере все ссылки на URL в тексте заменяются работающими гиперссылками:

```
$url = "Apress (http://www.apress.com)";
$url = ereg_replace("http://(([A-Za-z0-9.\-])*)", "<a href=\"\\0\">\\0</a>", $url);
print $url;
// Выводится строка:
// Apress (<a href="http://www.apress.com">http://www.apress.com</a>)
```

eregi( )

Функция eregi( ) ищет в заданной строке совпадение для шаблона. Синтаксис функции eregi( ):

int eregi (string шаблон, string строка [, array совпадения])

Поиск производится без учета регистра алфавитных символов. Функция eregi( ) особенно удобна при проверке правильности введенных строк (например, паролей). Использование функции eregi( ) продемонстрировано в следующем примере:

```
$password = "abc";
if (! eregi("[[:alnum:]]{8,10}", $password)) :
print "Invalid password! Passwords must be from 8 through 10 characters in length.";
endif;
// В результате выполнения этого фрагмента выводится сообщение об ошибке.
// поскольку длина строки "abc" не входит в разрешенный интервал
// от 8 до 10 символов.
```

eregi_replace( )

Функция eregi_replace( ) работает точно так же, как ereg_replace( ), за одним исключением: поиск производится без учета регистра символов. Синтаксис функции ereg_replace( ):

string eregi_replace (string шаблон, string замена, string строка)

split( )

Функция split( ) разбивает строку на элементы, границы которых определяются по заданному шаблону. Синтаксис функции split( ):

array split (string шаблон, string строка [, int порог])

Необязательный параметр порог определяет максимальное количество элементов, на которые делится строка слева направо. Если шаблон содержит алфавитные символы, функция split( ) работает с учетом регистра символов. Следующий пример демонстрирует использование функции split( ) для разбиения канонического IP-адреса на триплеты:

```
$ip = "123.345.789.000"; // Канонический IP-адрес
$iparr = split ("\.", $ip); // Поскольку точка является служебным символом.
// ее необходимо экранировать.
print "$iparr[0] <br>"; // Выводит "123"
print "$iparr[1] <br>"; // Выводит "456"
print "$iparr[2] <br>"; // Выводит "789"
print "$iparr[3] <br>"; // Выводит "000"
```

spliti( )

Функция spliti( ) работает точно так же, как ее прототип split( ), за одним исключением: она не учитывает регистра символов. Синтаксис функции spliti( ):

array spliti (string шаблон, string строка [, int порог])

Разумеется, регистр символов важен лишь в том случае, если шаблон содержит алфавитные символы. Для других символов выполнение spliti( ) полностью аналогично split( ).

sql_regcase( )

Вспомогательная функция sql_regcase( ) заключает каждый символ входной строки в квадратные скобки и добавляет к нему парный символ. Синтаксис функции sql_regcase( ):

string sql_regcase (string строка)

Строковые функции

Дополнение и сжатие строк

В процессе форматирования часто возникает необходимость в изменении длины строки посредством дополнения или удаления символов. В PHP существует несколько функций, предназначенных для решения этой задачи.

chop( )

Функция chop( ) возвращает строку после удаления из нее завершающих пропусков и символов новой строки. Синтаксис функции chop( ):

string chop(string строка)

В следующем примере функция chop( ) удаляет лишние символы новой строки:

```
$header = "Table of Contents\n\n";
$header = chop($header);
// $header = "Table of Contents"
```

trim( )

Функция trim( ) удаляет псе пропуски с обоих краев строки и возвращает полученную строку. Синтаксис функции trim( ):

```
string trim (string строка)
```

К числу удаляемых пропусков относятся и специальные символы \n, \r, \t, \v и \0.

Функция ltrim( ) удаляет все пропуски и специальные символы с левого края строки и возвращает полученную строку. Синтаксис функции ltrim( ):

```
string ltrim (string строка)
```

Функция удаляет те же специальные символы, что и функция trim( ).

Определение длины строки

Длину строки в символах можно определить при помощи функции strlen( ). Синтаксис функции strlen( ):

```
int strlen (string строка)
```

Следующий пример демонстрирует определение длины строки функцией strlen( ):

```
$string = "hello";
$length = strlen($string);
// $length = 5
```

Разделение и объединение строк

explode( )

Функция explode( ) делит строку на элементы и возвращает эти элементы в виде массива. Синтаксис функции explode( ):

```
array explode (string разделитель, string строка [, int порог])
```

Разбиение происходит по каждому экземпляру разделителя, причем количество полученных фрагментов может ограничиваться необязательным параметром порог. Разделение строки функцией explode( ) продемонстрировано в следующем примере:

```
$info = "wilson | baseball | indians";
$user = explode(" | ", $info);
// $user[0] = "wilson";
// $user[1] = "baseball";
// $user[2] = "Indians";
```

Функция explode( ) практически идентична функции регулярных выражений POSIX split( ), описанной выше. Главное различие заключается в том, что передача регулярных выражений в параметрах допускается только при вызове split( ).

implode( )

Если функция explode( ) разделяет строку на элементы массива, то ее двойник – функция implode( ) – объединяет массив в строку. Синтаксис функции implode( ):

```
string implode (string разделитель, array фрагменты)
```

Формирование строки из массива продемонстрировано в следующем примере:

```
$ohio_cities = array ("Columbus", "Youngstown", "Cleveland", "Cincinnati");
$city_string = implode(" | ", $ohio_cities);
// $city_string = "Columbus | Youngstown | Cleveland | Cincinnati";
```

У implode( ) имеется псевдоним – функция join( ).

Обработка строковых данных без применения регулярных выражений

str_replace( )

Функция str_replace( ) ищет в строке все вхождения заданной подстроки и заменяет их новой подстрокой. Синтаксис функции str_replace( ):

```
string str_replace (string подстрока, string замена, string строка)
```

Функция substr_replace( ), описанная ниже в этом разделе, позволяет провести заме ну лишь в определенной части строки. Ниже показано, как функция str_replace( ) используется для проведения глобальной замены в строке.

Если подстрока ни разу не встречается в строке, исходная строка не изменяется:

```
$favorite_food = "My favorite foods are ice cream and chicken wings";
$favorite_food = str_replace("chicken wings", "pizza", $favorite_food);
// $favorite_food = "My favorite foods are ice cream and pizza"
```

Преобразование текста в HTML

Быстрое преобразование простого текста к формату web-браузера – весьма распространенная задача. В ее решении вам помогут функции, описанные в этом разделе.

nl2br( )

Функция nl2br( ) заменяет все символы новой строки (\n) эквивалентными конструкциями HTML
. Синтаксис функции nl2br( ):

```
string nl2br (string строка)
```

Символы новой строки могут быть как видимыми (то есть явно включенными в строку), так и невидимыми (например, введенными в редакторе). В следующем примере текстовая строка преобразуется в формат HTML посредством замены символов \n разрывами строк:

```
// Текстовая строка, отображаемая в редакторе.
$text_recipe = "
Party Sauce recipe:
1 can stewed tomatoes
3 tablespoons fresh lemon juice
Stir together, server cold.";
// Преобразовать символы новой строки в <br>
$html_recipe = nl2br($text_recipe);
```

При последующем выводе $html_recipe браузеру будет передан следующий текст в формате HTML:

```
Party Sauce recipe:<br>
1 can stewed tomatoes<br>
3 tablespoons fresh lemon juice<br>
Stir together, server cold.<br>
```

htmlspecialchars( )

Функция htmlspecialchars( ) заменяет некоторые символы, имеющие особый смысл в контексте HTML, эквивалентными конструкциями HTML. Синтаксис функции htmlspecialchars( ):

```
string htmlspecialchars (string строка)
```

Функция html special chars( ) в настоящее время преобразует следующие символы:

& преобразуется в &,
" преобразуется в ",
< преобразуется в <,
> преобразуется в >.

В частности, эта функция позволяет предотвратить ввод пользователями разметки HTML в интерактивных web-приложениях (например, в электронных форумах). Ошибки, допущенные в разметке HTML, могут привести к тому, что вся страница будет формироваться неправильно. Впрочем, у этой задачи существует и более эффективное решение – полностью удалить теги из строки функцией strip_tags( ).

Следующий пример демонстрирует удаление потенциально опасных символов функцией htmlspeclalchars( ):

```
$user_input = "I just can't get <enough> of PHP & those fabulous cooking recipes!";
$conv_input = htmlspecialchars($user_input);
// $conv_input = "I just can't &lt;&lt;enough&gt;&gt; of PHP &amp those fabulous cooking recipes!";
```

Если функция htmlspecialchars( ) используется в сочетании с nl2br( ), то последнюю следует вызывать после htmlspecialchars( ). В противном случае конструкции, сгенерированные при вызове nl2br( ), преобразуются в видимые символы.

Преобразование HTML в простой текст

Иногда возникает необходимость преобразовать файл в формате HTML в простой текст. Функции, описанные ниже, помогут вам в решении этой задачи.

strip_tags( )

Функция strip_tags( ) удаляет из строки все теги HTML и PHP, оставляя в ней только текст. Синтаксис функции strip_tags( ):

```
string strip_tags (string строка [, string разрешенные_теги])
```

Необязательный параметр разрешенные_теги позволяет указать теги, которые должны пропускаться в процессе удаления.

Ниже приведен пример удаления из строки всех тегов HTML функцией strip_tags( ):

```
$user_input = "I just <b>love</b> PHP and <i>gourment</i> recipes!";
$stripped_input = strip_tags($user_input);
// $stripped_input = "I just love PHP and gourmet recipes!";
```

В следующем примере удаляются не все, а лишь некоторые теги:

```
$user_input = "I <b>love</b> to <a href = \"http://www.eating.com\">eat</a>!";
$strip_input = strip_tags ($user_input, "<a>");
// $strip_input = "I love to <a href = "http://www.eating.com">eat</a>!";
```

Удаление тегов из текста также производится функцией fgetss( ), описанной в главе 7.

PHP и HTTP: headers

PHP, будучи языком вебпрограммирования, поддерживает реализацию механизма отправки заголовков HTTP.

В соответствии со спецификацией HTTP, этот протокол поддерживает передачу служебной информации от сервера к броузеру, оформленной в виде специальных заголовков.

Таким образом, HTTP headers – это средство общения сервера с удаленным клиентом. Каждый заголовок обычно состоит из одиночной линии ASCII текста с именем и значением. Сами заголовки никак не отображаются в окне броузера, но зачастую могут сильно изменить отображение сопутствующего документа.

Механизм отправки HTTP заголовков в PHP

Механизм отправки заголовков в PHP представлен функцией header(). Особенность протокола HTTP заключается в том, что заголовок должен быть отправлен до посылки других данных, поэтому функция должна быть вызвана в самом начале документа и должна выглядеть следующим образом:

```
header("HTTP заголовок", необязательный параметр replace);
```

Опциональный параметр replace может принимать значения типа bool (true или false) и указывает на то, должен ли быть замещен предыдущий заголовок подобного типа, либо добавить данный заголовок к уже существующему.

В отношении функции header() часто применяется функция headers_sent(), которая в качестве результата возвращает true в случае успешной отправки заголовка и false в обратном случае.

Рассмотрим наиболее используемые HTTP заголовки.

Cache-control

"Cache-control: " значение

Заголовок управления кешированием страниц. Вообще, данная функция является одной из самых распространенных в использовании заголовков.

Данный заголовок может быть использован со следующими значениями:

- no-cashe – Запрет кеширования. Используется в часто обновляемых страницах и страницах с динамическим содержанием. Его дейсвтие подобно META тегу "Pragma: no-cache".
- public – Разрешение кеширования страницы как локальным клиентом, так и прокси-сервером.
- private – Разрешение кеширования только локальным клиентом.
- max-age – Разрешение использования кешированного документа в течение заданного времени в секундах.

```
header("Cache-control: private, max-age = 3600")
// Кеширование локальными клиентами и использование в течение 1 часа
```

Expires

"Expires: " HTTP-date

Устанавливает дату и время, после которого документ считается устаревшим. Дата должна указываться в следующем формате (на английском языке):

День недели (сокр.) число (2 цифры) Месяц (сокр.) год часы:минуты:секунды GMT

Например, Fri, 09 Jan 2002 12:00:00 GMT

Текущее время в этом формате возвращает функция gmdate() в следующем виде:

```
echo gmdate("D, d M Y H:i:s")."GMT";
```

Возможно использование данного HTTP заголовка для запрета кеширования. Для этого необходимо указать прошедшую дату.

Last-Modified.

"Last-Modified: " HTTP-date

Указывает дату последнего изменения документа. Дата должна задаваться в том же формате, что и в случае с заголовком Expires. Данный заголовок можно не использовать для динамических страниц, так как многие серверы (например, Apache) для таких страниц сами выставляют дату модификации.

Возможно сделать страницу всегда обновленной:

```
header("Last-Modified: ".gmdate("D, d M Y H:i:s")." GMT");
```

Location

"Location :" абсолютный URL

Полезный заголовок, который перенаправляет броузер на указанный адрес. Его действие сравнимо с META тегом Refresh:

```
<META HTTP-EQUIV="Refresh" CONTENT="0; URL=someURL">
```

Например, этот заголовок может быть использован так:

```
if ($login != $admin_login) header("Location: http://www.server.com/login.php");
else header("Location: http://www.server.com/admin.php?login=$login");

if (!headers_sent())
exit("Произошла ошибка! Пройдите <a  href='http://www.server.com/login.php'>авторизацию</a> заново");
```

Выше были рассмотрены наиболее полезные и самые часто используемые HTTP заголовки. Полный список смотрите на странице http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

PHP и HTTP: cookie

Выше была рассмотрена взаимосвязь протокола HTTP и языка PHP на уровне HTTP заголовков. Ниже пойдет речь об ещё одном специфическим HTTP заголовком – cookie.

Что такое cookies?

Дело в том, что в процессе развития www-технологий и внедрения языков программирования в Интернет, перед разработчиками программ возникла очень серьезная проблема – как сохранять результаты выполнения алгоритма для каждого конкретно взятого пользователя на долгое время? Сам по себе протокол HTTP не имеет возможности фиксирования результатов программных процессов. Использование сессий также не является решением проблемы, так как их действие прекращается сразу после разрыва соединения с сервером.

Проблема разрешилась с внедрением механизма cookies (то есть, в переводе с английского, – «печенье»). Cookies обладают замечательным свойством – они сохраняются на винчестере пользователя и могут храниться там практически неограниченное время.

По своей сути cookies – это обычные текстовые файлы, хранящиеся в специальной директории, используемой броузером (обычно эта папка называется Temporary Internet Files), и вы можете увидеть их, зайдя в эту директорию (быстрый доступ к ней для броузера IE осуществляется через пункты меню Сервис → Свойства обозревателя → Временные файлы Интернета → Настройка → Просмотр файлов).

Реализация механизма cookies в PHP

Реализация механизма cookies представлена единственной функцией setcookie(). Как и в случае с HTTP заголовками, эта функция должна быть вызвана до отправки каких-либо данных удаленному клиенту, не допускаются даже "пустые" символы, то есть пробел, символы перевода строки и так далее.

Функция имеет следующий синтаксис:

```
setcookie(имя куки, значение, срок годности, информация о пути, домен, защищенность)
```

Все параметры, кроме имени cookie, являются необязательными. Если cookie посылается только с этим параметром, то она сразу же уничтожается удаленным клиентом, поэтому сам по себе этот параметр не несет информационной нагрузки. Полнофункциональной cookie делают два следующих параметра: значение, заложенное в куке, и время, до которого эта cookie может быть использована.

Значением, которое несет cookie, может быть любая строка ASCII символов. Например, можно установить cookie с именем и фамилией посетителя, которые он до этого ввел в поле формы.

```
$data = $name."||".$surname;
setcookie("username", $data);
```

Заметьте, что отсылаемые данные должны быть оформлены в виде строки, попытка прочитать отосланный ранее массив значений ни к чему не приведет.

Cookie, установленная в вышеуказанном примере, будет уничтожена сразу после закрытия броузера пользователем, так как по умолчанию срок жизни cookie устанавливается в ноль. Чтобы изменить этот порядок, необходимо указать третий параметр expire. Определение этого параметра можно произвести двумя способами:

Задать относительный срок действия с помощью функции time(), к которой прибавляется время в секундах для хранения cookie. Например, чтобы определить cookie на два часа необходимо написать:

```
setcookie("test 1", "это тестовая куки", time() + 3600 * 2);
// 3600 – количество секунд в часе
```

Второй способ – задание абсолютного срока истечения годности cookie. Он устанавливается с помощью функции mktime(), которая возвращает конкретную дату удаления куки. Если необходимо задать срок жизни cookie до полуночи 1 сентября 2003 года, то следует определить cookie так:

```
setcooikie("test 2", "куки с абсолютной датой удаления", mktime(0, 0, 0, 9, 1, 2003);
```

Необязательный параметр пути ограничивает область действия cookie в пределах определенных директорий. Причем в эту область входят все пути, начинающиеся со значения в этом параметре. Например:

```
setcookie("test 3, "", 0, "/mus");
```

Мы установили куку, пропустив параметры значения и времени и определив область действия всеми путями, начинающимися со строки "/mus", то есть сюда входят и директория "/music/", и "/museums/". Чтобы однозначно определить путь, необходимо завершить путь слешем. То есть для ограничения действия куки каталогом "/mus", необходимо было написать в параметре "/mus/".

Следующим опциональным параметром является параметр определения действия cookie в пределах указанного домена. Причем значению этого параметра "someserver.com" соответствует только сайт с адресом http://someserver.com, а значению ".someserver.com" соответствуют уже и http://someserver.com, и http://mail.someserver.com, и http://my-someserver.com, то есть все домены, кончающиеся данной строкой.

Последний параметр функции setcookie() указывает на то, что данная cookie должна быть послана через защищенное соединение (HTTPS). Этот параметр необходим при установке cookie с конфеденциальными данными.

```
setcookie("my_cookie", $value, time() + 3600 * 24 * 5, "/", ".myphp.dem.ru", 1);
```

Чтение cookie

Обращение к установленной cookie идет через ее имя. Например, продолжая пример выше, прочесть cookie можно следующим образом:

```
echo "У вас сохранены следующие данные:<br>";
echo $my_cookie;
```

Обращение к данным, сохраненным в cookie, также может происходить через массив $HTTP_COOKIE_VARS. Он схож с другими подобными массивами, такими как $HTTP_POST_VARS и другими, и содержит все значения, прочтенные из cookie.

Удаление cookie

Удаление cookie производится отправкой новой cookie с именем удаляемой и с установкой даты в прошлом. Например:

```
setcookie("my_cookie","",1);
echo "Cookie с именем my_cookie были удалены";
```

В данном случае "1" – это первая секунда 1 января 1970 г., т.е. дата в прошлом.

Сессии

Рано или поздно практически перед каждым вебмастером встает проблема передачи данных сквозь несколько страниц. Как же сделать так, чтобы пользователь, бродя по сайту, не "терял" однажды введенных данных?

Один из таких способов – использование скрытых элементов форм «hidden». На каждой странице сайта мы размещаем эти элементы, внося в них с помощью PHP значения и передавая эти значения далее, другой странице. Конечно, такой способ вполне возможен, но он нерационален. Представляете, сколько таких элементов надо вставить на сайте объемом, например, 50 страниц.

Но в PHP реализован очень удобный и функциональный механизм работы с сессиями. Он позволяет сохранять любые данные, связанные с пользователем, и использовать их на протяжении всего времени нахождения пользователя на данном сайте.

Реализация механизма сессий в PHP

Любая сессия открывается с помощью функции session_start( ), создающей специальный служебный файл с именем, соответствующим ID сессии, в который впоследствии будут записаны все данные, связанные с текущей сессией. Место размещения этих файлов зависит от настроек PHP. Так что если вы используете в своих скриптах сессии, не забывайте иногда подчищать директорию с этими временными файлами, так как там со временем может накопиться солидное количество ненужных файлов.

Также эта функция используется для продолжения текущей сессии. Таким образом, она должна быть вызвана на каждой странице, использующей данные текущей сессии.

В PHP предусмотрено два способа передачи ID сессии (сокращенно SID):

- Через метод GET.
  Тогда посетитель будет видеть в своем броузере адресную строку следующего типа:
  http://server.com/main.php?PHPSESSID=bdd95bcd4e1e2ef5ec57fc83a69bba86
- Через Cookie.
  Здесь, соответственно, посетитель не будет видеть признаков существования сессии, SID передается через Cookie.

Следующий шаг в работе с сессиями – запись данных в сессию. Для работы с данными, сохраненными при работе с сессиями, используется суперглобальный массив $_SESSION.

Регистрация данных в сессию должна выглядеть примерно следующим образом:

```
session_start();
$_SESSION['name'] = "Вася Пупкин";
```

Теперь на любой странице данного сайта можно обратиться к посетителю по имени:

```
session_start();
echo $_SESSION['name'];
```

Для удаление какого-либо ставшего уже ненужным элементом массива $_SESSION необходимо использовать функцию unset( ). Иногда бывает полезно – например, в том случае, если вы регистрируете в сессию большое количество переменных, чтобы не перезагружать файл текущей сессии, можно удалить оттуда уже ненужные значения.

Дополнительные функции работы с сессиями

session_id

Нередко при работе с сессиями требуется определить ее ID. Этим занимается функция с соответствующим названием session_id(), которая в качестве результата возвращает ID текущей сессии.

session_name

Бывают случаи, когда становится очень неудобным использовать ID сессии, например, из-за его громозкости и ненаглядности, так как id сессии вида 7542b069d57510a99eaeb31391b15cbf нам практически ничего не скажет. В этом случае более разумным становится использование функции session_name, которая может выполнять две роли. Во-первых, она может возвращать имя текущей сессии (по умолчанию – PHPSESSID). В этом случае ее следует использовать без аргументов. Во-вторых, эта функция может устанавливать имя текущей сессии. Рассмотрим пример:

```
session_start();
echo session_name();
session_name("MySession");
echo session_name();
```

Данный пример выведет:

```
PHPSESSID
MySession
```

Безусловно, такие названия сессии воспринимаются намного лучше, чем страшные ID.

session_destroy

Завершает работу сессии функция session_destroy( ). Она уничтожает файл, связанный с текущей сессией, что является очень удобным. Но здесь возникает проблема: часто мы не знаем, где именно необходимо уничтожить сессиию. Например, если данные сессии используется страницами всего сайта, то мы не можем уничтожить сессию на определенной странице, так как не знаем, какая именно страница будет последней просмотренной страницей посетителем на сайте. Поэтому использование данной функци возможно лишь в том случае, если мы заранее знаем, на какой именно странице действие сессии должно прекратиться.

Работа с файлами

Проверка существования и размера файла

Прежде чем пытаться работать с файлом, желательно убедиться в том, что он существует. Для решения этой задачи обычно используются две функции:

file_exists( ) и is_file( ).

file_exists( )

Функция file_exists( ) проверяет, существует ли заданный файл. Если файл существует, функция возвращает TRUE, в противном случае возвращается FALSE. Синтаксис функции file_exists( ):

```
bool file_exists(string файл)
```

Пример проверки существования файла:

```
$filename="file.txt";
if (! file_exists ($filename)) :
print "File $filename does not exist!";
endif;
```

is_file( )

Функция is_file( ) проверяет существование заданного файла и возможность выполнения с ним операций чтения/записи. В сущности, is_file( ) представляет собой более надежную версию file_exists( ), которая проверяет не только факт существования файла, но и то, поддерживает ли он чтение и запись данных:

```
bool is_file(string файл)
```

Следующий пример показывает, как убедиться в существовании файла и возможности выполнения операций с ним:

```
$file = "somefile.txt";
if (is_file($file)) :
print "The file $file is valid and exists!";
else :
print "The file $file does not exist or it is not a valid file!";
endif;
```

Убедившись в том, что нужный файл существует и с ним можно выполнять различные операции чтения/записи, можно переходить к следующему шагу – открытию файла.

filesize( )

Функция filesize( ) возвращает размер (в байтах) файла с заданным именем или FALSE в случае ошибки. Синтаксис функции filesize( ):

```
int filesize(string имя_файла)
```

Предположим, вы хотите определить размер файла pastry.txt. Для получения нужной информации можно воспользоваться функцией filesize( ):

```
$fs = filesize("pastry.txt");
print "Pastry.txt is $fs bytes.";
```

Выводится следующий результат:

```
Pastry.txt is 179 bytes.
```

basename( )

Функция basename( ) выделяет имя файла из переданного полного имени. Синтаксис функции basename( ):

```
string basename(string полное_имя)
```

Выделение базового имени файла из полного имени происходит следующим образом:

```
$path = "/usr/local/phppower/htdocs/index.php";
$file = basename($path); // $file = "index.php"
```

Фактически эта функция удаляет из полного имени путь и оставляет только имя файла.

chmod( )

Функция chmod( ) изменяет разрешения файла с заданным именем. Синтаксис функции chmod( ):

```
int chmod (string имя_файла, int разрешения)
```

Разрешения задаются в восьмеричной системе. Например:

```
chmod("data_file.txt", 0766);
```

Открытие файла

Работа с файлами в PHP разделяется на три этапа. Сначала файл открывается в нужном режиме, при этом возвращается некое целое число, служащее идентификатором открытого файла (дескриптор файла). Затем настает очередь команд работы с файлом (чтение или запись, или и то и другое), причем они "привязаны" уже к дескриптору файла, а не к его имени. После этого файл лучше всего закрыть (хотя это можно и не делать, поскольку PHP автоматически закрывает все файлы по завершении сценария).

```
fopen($filename, $mode, $use_include_path=false)
```

Открывает файл с именем $filename в режиме $mode и возвращает дескриптор открытого файла. Если операция "провалилась", то, как это принято, fopen() возвращает false. Необязательный параметр $use_include_path говорит PHP о том, что, если задано относительное имя файла, его следует искать также и в списке путей, используемом инструкциями include и require. Обычно этот параметр не используют.

Параметр $mode может принимать следующие значения:

- r – файл открывается только для чтения. Если файла не существует, вызов регистрирует ошибку. После удачного открытия указатель файла устанавливается на его первый байт, т. е. на начало;

- r+ – файл открывается одновременно на чтение и запись. Указатель текущей позиции устанавливается на его первый байт. Как и для режима r, если файла не существует, возвращается false. Следует отметить, что если в момент записи указатель файла установлен где-то в середине файла, то данные запишутся прямо поверх уже имеющихся, а не "раздвинут" их, при необходимости увеличив размер файла.

- w – создает новый пустой файл. Если на момент вызова уже был файл с таким именем, то он предварительно уничтожается. В случае неверно заданного имени файла вызов, как нетрудно догадаться, "проваливается";

- w+ – аналогичен r+, но если файла изначально не существовало, создает его. После этого с файлом можно работать как в режиме чтения, так и записи. Если файл существовал до момента вызова, его содержимое удаляется;

- a – открывает существующий файл в режиме записи, и при этом сдвигает указатель текущей позиции за последний байт файла. Этот режим полезен, если требуется что-то дописать в конец уже имеющегося файла. Как водится, вызов неуспешен в случае отсутствия файла;

- a+ – открывает файл в режиме чтения и записи, указатель файла устанавливается на конец файла, при этом содержимое файла не уничтожается. Отличается от a тем, что если файла изначально не существовало, то он создается. Этот режим полезен, если вам нужно что-то дописать в файл (например, в журнал), но вы не знаете, создан ли уже такой файл;

Закрытие файла

После работы файл лучше всего закрыть (хотя, на самом деле это делается автоматически при завершении сценария, но лучше все же не искушать судьбу):

```
fclose(int $fp)
```

Закрывает файл, открытый предварительно функцией fopen(). Возвращает false, если файл закрыть не удалось (например, что-то с ним случилось или же разорвалась связь с удаленным хостом). В противном случае возвращает значение "истина".

Чтение и запись

Для каждого открытого файла система хранит определенную величину, которая называется текущей позицией ввода-вывода, или указатель файла. Функции чтения и записи файлов работают именно с этой позицией. А именно, функции чтения читают блок данных, начиная с этой позиции, а функции записи – записывают, также отсчитывая от нее. Если указатель файла установлен за последним байтом и осуществляется запись, то файл автоматически увеличивается в размере. Есть также функции для установки этой самой позиции в любое место файла.

После того как файл успешно открыт, из него (при помощи дескриптора файла) можно читать, а также, при соответствующем режиме открытия, писать. Обмен данными осуществляется через обыкновенные строки и, что важнее всего, начиная с позиции указателя файла.

Блочные чтение/запись

fread($f, $numbytes)

Читает из файла $f $numbytes символов и возвращает строку этих символов. После чтения указатель файла продвигается к следующим после прочитанного блока позициям (это происходит и для всех остальных функций). Разумеется, если $numbytes больше, чем можно прочитать из файла (например, раньше достигается конец файла), возвращается то, что удалось считать. Этот прием можно использовать, если нужно считать в строку файл целиком. Для этого следует задать в $numbytes очень большое число (например, сто тысяч).

fwrite($f, $st)

Записывает в файл $f все содержимое строки $st. Эта функция составляет пару для fread(), действуя "в обратном направлении".

Построчные чтение/запись

fgets($f, $length)

Читает из файла одну строку, заканчивающуюся символом новой строки \n. Этот символ также считывается и включается в результат. Если строка в файле занимает больше $length-1 байтов, то возвращаются только ее $length-1 символов.

fputs($f, $st)

Эта функция – полный аналог fwrite().

Чтение файла в массив

Функция file( ) загружает все содержимое файла в индексируемый массив. Каждый элемент массива соответствует одной строке файла. Синтаксис функции file ( ):

array file (string файл [, int включение_пути])

Если необязательный третий параметр включение_пути равен 1, то путь к файлу определяется по отношению к каталогу включаемых файлов, указанному в файле php.ini.

Копирование и переименование файлов

К числу других полезных системных функций, которые могут выполняться в сценариях PHP, относятся копирование и переименование файлов на сервере. Эти операции выполняются двумя функциями: copy( ) и rename( ).

copy( )

Копирование файлов осуществляется функцией PHP copy( ):

int copy (string источник, string приемник)

Функция copy( ) пытается скопировать файл источник в файл приемник; в случае успеха возвращается TRUE, а при неудаче – FALSE. Если файл приемник не существует, функция copy( )

создает его. Следующий пример показывает, как создать резервную копию файла при помощи функции copy( ):

```
$data_file = "data.txt";
copy($data_file, $data_file.'.bak') or die("Could not copy $data_file");
```

rename( )

Функция rename( ) переименовывает файл. В случае успеха возвращается TRUE, а при неудаче – FALSE. Синтаксис функции rename( ):

```
bool rename (string старое_имя, string новое_имя)
```

Пример переименования файла функцией rename( ):

```
$data_file = "data.txt";
rename($data_file, $data_file.'.old') or die ("Could not rename $data_file");
```

Удаление файлов

unlink( )

Функция unlink( ) удаляет файл с заданным именем. Синтаксис:

```
int unlink (string файл)
```

Если вы работаете с PHP в системе Windows, при использовании этой функции иногда возникают проблемы. В этом случае можно воспользоваться функцией system( ) и удалить файл командой DOS del:

```
system ("del filename.txt");
```

Работа с каталогами

Функции PHP позволяют просматривать содержимое каталогов и перемещаться по ним

dirname( )

Функция dirname( ) дополняет basename( ) – она извлекает путь из полного имени файла. Синтаксис функции dirname( ):

```
string dirname (string путь)
```

Пример использования dirname( ) для извлечения пути из полного имени:

```
$path = "/usr/locla/phppower/htdocs/index.php";
$file = dirname($path); // $file = "/usr/local/phppower/htdocs"
```

Функция dirname( ) иногда используется в сочетании с переменной $SCRIPT_FILENAME для получения полного пути к сценарию, из которого выполняется команда:

```
$dir = dirname($SCRIPT_FILENAME);
```

is_dir( )

Функция is_dir( ) проверяет, является ли файл с заданным именем каталогом:

```
bool is_dir (string имя_файла)
```

mkdir()

Функция mkdir( ) создает новый каталог. Синтаксис функции mkdir( ):

```
int mkdir (string путь, int режим)
```

Параметр путь определяет путь для создания нового каталога. Не забудьте завершить параметр именем нового каталога! Параметр режим определяет разрешения, назначаемые созданному каталогу.

opendir( )

Подобно тому как функция fopen( ) открывает манипулятор для работы с заданным файлом, функция opendir( ) открывает манипулятор для работы с каталогом. Синтаксис функции opendir( ):

```
int opendir (string путь)
```

closedir( )

Функция closedir( ) закрывает манипулятор каталога, переданный в качестве параметра. Синтаксис функции closedir( ):

```
void closedir(int манипулятор_каталога)
```

readdir( )

Функция readdir( ) возвращает очередной элемент заданного каталога. Синтаксис:

```
string readdir(int манипулятор_каталога)
```

С помощью этой функции можно легко вывести список всех файлов и подкаталогов, находящихся в текущем каталоге:

```
$dh = opendir('.');
while ($file = readdir($dh)) :
print "$file <br>";
endwhile;
closedir($dh);
```

chdir( )

Функция chdir( ) осуществляет переход в каталог, заданный параметром. Синтаксис функции chdir( ):

```
int chdir (string каталог)
```