Jakub Vrána

1001 TIPŮ A TRIKŮ PRO PHP

CD obsahuje
veškeré zdrojové kódy z knihy, instalaci Apache, MySQL a další užitečné nástroje

Nejlepší postupy a řešení pro vaše webové aplikace

- Využití AJAXu, návrhových vzorů, export a import dat
- Zabezpečení dat, ladění výkonu aplikace
- Práce se soubory, XML dokumenty, knihovnami a frameworky
- Regulární výrazy, zpracování formulářů a e-mailů

CPRESS

Jakub Vrána

1001 tipů a triků pro PHP

Computer Press
Brno
2012

1001 tipů a triků pro PHP

Jakub Vrána

Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-2940-1

Vydalo nakladatelství Computer Press v Brně roku 2012 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 16 097.

Dotisk 1. vydání.

ALBATROS MEDIA a.s.

Stručný obsah

Obsah

Webové aplikace 167

Úvod

Kromě toho, že se živím programováním, tak se také podílím na tvorbě oficiální dokumentace PHP a učím základy vytváření webových stránek na vysoké škole. Vedle toho píšu *blog php.vrana.cz* a vedu několik školení o tvorbě webových aplikací se zaměřením na PHP, MySQL a JavaScript. Ze všech těchto vstupů čerpám inspiraci, a proto doufám, že všechny tipy a triky v této knize budou pro čtenáře nějakým způsobem zajímavé. S vatovými triky následujícího typu se tedy v této knize rozhodně nesetkáte:

1. Věděli jste, že oblouk nakreslíte funkcí `imageArc`?
2. Věděli jste, že obdélník nakreslíte funkcí `imageRectangle`?
3. Věděli jste, že čáru nakreslíte funkcí `imageLine`?

Místo toho najdete třeba i tipy týkající se výkonnosti webových aplikací, verzovacích systémů nebo dokonce vývoje PHP. I to je totiž pro programátora v PHP důležité.

Kniha u čtenáře předpokládá znalost syntaxe PHP a schopnost dohledávat si informace v dokumentaci. Pokud tedy nějaký tip např. pojednává o zajímavém využití nějaké funkce, tak se nevysvětluje význam jednotlivých parametrů, protože ten je popsán v dokumentaci.

Kniha byla napsaná v době PHP 5.3, v době zrušení vývojové verze PHP 6, která měla zavést podporu pro Unicode. Zabývá se tedy verzemi PHP 5.3 a staršími.

Kompletní text knihy lze prohledávat pomocí hledání na webu *php.vrana.cz*.

Komu je kniha určena

Tipy jsou rozděleny do tří kategorií pro začátečníky, pokročilé a znalce. Rozdělení ale není ani tak podle náročnosti, jako spíš podle jejich potřebnosti.

Tipy pro začátečníky by měl znát i začínající programátor (i když někdy můžou být složitější).

Tipy pro pokročilé by měla znát většina programátorů PHP.

Bez znalosti tipů pro znalce se dá často obejít, i když samozřejmě můžou poskytnout nějakou zkratku nebo usnadnění práce.

Odlišení písma

V knize se používá tučné písmo pro názvy produktů, kurziva pro odborné termíny, neproporcionální písmo pro ukázky kódu. V ukázkách kódu jsou tučně klíčová slova a vestavěné funkce, kurzivou proměnné.

Poděkování

Chtěl bych poděkovat Davidu Grudlovi, který je autorem několika knihoven použitých v této knize a také spoluautorem kódu u několika tipů. Také mu děkuji za řadu cenných připomínek.

Doprovodné CD

Doprovodný disk obsahuje kromě zdrojových kódů také řadu odkazů na užitečné stránky a také několik užitečných nástrojů, jež vám programování v jazyce PHP výrazně usnadní nebo alespoň zpříjemní.

CD stačí vložit do počítače a rozhraní se spustí automaticky. Pokud nemáte automatické spouštění disků povoleno, vyhledejte na CD kořenový adresář a otevřete soubor spustit_CD.html.

Jestliže rozhraní CD otevřete v prohlížeči Internet Explorer, Opera nebo Google Chrome, budete z CD moci instalovat doprovodný software okamžitě. V případě jiných prohlížečů se zobrazí výzva k uložení instalačního souboru na pevný disk. V tomto případě doporučujeme spustit instalaci přímo z CD. Obsah CD najdete ve složce obsah.

Přehledný kód

1 Jak je to s velikostí písmen

V důsledku toho, že PHP převzalo řadu funkcí a obratů z jiných programovacích jazyků, nemá jednotné pravidlo pro používání velkých a malých písmen. Alespoň některá pravidla jsou ale společná:

- Proměnné a funkce je zvykem začínat malým písmenem.
- Názvy tříd obvykle začínají velkým písmenem.
- Konstanty se píšou celé velkými písmeny a slova se oddělují podtržítkem.

Nejednotnost tkví hlavně v oddělování slov. Většina vestavěných funkcí PHP používá pro oddělení slov podtržítko (např. `array_keys`), pokud to ovšem nejsou zkratky (jako např. `strPos`). Některé funkce se toho ale nedrží a slova nijak neoddělují (např. `imageCreate`).

Objektový kód obvykle slova odděluje zvětšením písmene (např. `PDO::errorCode`), i když ani to neplatí vždy (`mysqli::select_db`).

U funkcí a tříd PHP velikost písmen nerozlišuje, u proměnných ale ano. V této knize budeme pro oddělování slov používat zvětšení písmene, z důvodu lepší čitelnosti i u vestavěných funkcí PHP.

2 Jak je to s velikostí písmen u zkratek

Pokud je součástí identifikátoru (např. proměnné nebo funkce) nějaká zkratka, je vhodné ji zapisovat malými písmeny stejně jako ostatní slova. Jinak bude identifikátor hůře čitelný, obzvlášť v situaci, kdy by zkratek bylo více za sebou.

```
<?php
// dobře čitelné
createXmlHttpRequest($xmlHttp);

// špatně čitelné, proměnné by navíc měly začínat malým písmenem
createXMLHTTPRequest($XMLHTTP);
?>
```

3 Jak je to s velikostí písmen v URL

Adresu URL je zvykem zapisovat malými písmeny a slova oddělovat spojovníkem, stejný způsob pojmenování tedy volíme u skriptů přímo zobrazovaných na webu.

U souborů vkládaných do jiných skriptů odvíjíme název souboru od toho, co v souboru definujeme. Pokud to je např. třída, tak soubor můžeme pojmenovat stejně jako třídu, včetně velikosti písmen.

4 Diakritika v identifikátorech

PHP dovoluje v názvech proměnných a v dalších identifikátorech používat i znaky z horní poloviny ASCII tabulky. Pokud se tedy rozhodneme zadávat identifikátory česky, můžeme je psát i s diakritikou. V praxi se toho ale téměř nevyužívá, protože tento způsob zápisu nemusí podporovat všechny editory.

5 Diakritika v komentářích

Pokud se rozhodneme komentáře k programu psát česky, měli bychom zásadně používat diakritiku (háčky a čárky). Souvislejší texty bez diakritiky se totiž velmi špatně čtou. Zdrojové kódy se v poslední době obvykle píšou v kódování UTF-8.

6 Rozmístění tříd do souborů

Každá třída se obvykle umisťuje do samostatného souboru pojmenovaného podle jejího názvu. Toto pravidlo se občas porušuje u definic výjimek, které se buď definují hromadně v jednom souboru, nebo spolu s třídou, která je vyvolává.

7 Pojmenování rozhraní

Název rozhraní (`interface`) v některých projektech začíná znakem I, zdaleka to ale není pravidlem. Stejně jako třídy se rozhraní obvykle umisťují do souborů pojmenovaných podle jejich názvu.

8 Co to je značka Byte-Order-Mark

Soubory ve znakové sadě Unicode mohou začínat značkou *Byte-Order-Mark* (BOM). To je posloupnost bajtů na začátku souboru, která obsahuje informaci o použitém kódování. Platí to i pro kódování UTF-8, které lze používat pro vytváření PHP skriptů. Tuto značku ale PHP nijak neinterpretuje, takže ji není vhodné používat. Jinak může dojít k tomu, že např. funkce `header` skončí chybou, protože už skript odeslal nějaký výstup (neviditelnou značku BOM) nebo že skript odešle těchto značek více (pokud do sebe vkládáme více skriptů, které by měly značku uvedenou).

9 Čeština nebo angličtina?

Při programování bychom si měli vybrat nejen programovací jazyk (což je nevyhnutelné), ale i ten lidský, který budeme používat pro identifikátory a texty v programu. U identifikátorů, jako je název proměnných nebo funkcí, je obvykle přirozené používat angličtinu, protože i návazný kód používá angličtinu.

```
<?php
$xmlReader = new XmlReader; // působí přirozeně
$cteckaXml = new XmlReader; // působí krkolomně
?>
```

V některých případech se navíc používají ustálené obraty jako např. název metody `getConnection`. České varianty, ať už v podobě `getPripojeni` nebo třeba `dejPripojeni`, působí nepřirozeně.

V knize tedy budeme používat anglické identifikátory.

Poznámka: Stejné pravidlo lze použít i pro pojmenování souborů.

10 Jazyk textů a komentářů

U vypisovaných textů je situace daná tím, zda vytváříme jednojazyčnou nebo vícejazyčnou aplikaci. U jednojazyčné je přirozené psát texty rovnou ve výsledném jazyce, u vícejazyčné potom v angličtině (pokud je jedním z výsledných jazyků).

U komentářů záleží na tom, kdo s nimi bude pracovat. Pokud všichni autoři i čtenáři komentářů ovládají dostatečně dobře angličtinu, tak je vhodné psát komentáře právě v ní, aby byl jazyk v souladu s identifikátory. Pokud taková situace nenastane, tak je vhodné psát komentáře jazykem, který všichni zúčastnění ovládají nejlépe.

Vypisované texty a komentáře v knize budou tedy česky.

11 Pojmenování funkcí

Název funkcí a metod je vhodné začínat slovesem určujícím, co přesně vlastně funkce nebo metoda dělá. Např. `getConnection` je lepší název než samotné `connection`, protože je z názvu jasné, že jde o získání připojení, a ne např. o jeho nastavení. Slovesem je vhodné název funkce vždy začínat, `getConnection` je lepší než `connectionGet`. U neobjektového kódu je možné názvu funkce předřadit modul, kterého se týká (např. `imageCreate`), u metod objektů to není potřeba.

12 Název aplikace v identifikátorech

Budoucí verze PHP mohou používat nová klíčová slova nebo jiné identifikátory, které by mohly kolidovat s identifikátory použitými naší aplikací. V zájmu dopředné kompatibility bychom tedy v hlavním jmenném prostoru aplikace neměli používat příliš obecné identifikátory. Např. PHP 5.1 zavedlo třídu `Date` (ta byla po sporech nakonec přejmenována na `DateTime`). Stejně tak mohou nové identifikátory zavádět používané knihovny.

Předřazovat všem identifikátorům název aplikace ale také není zrovna nejšťastnější, protože to identifikátory prodlužuje a znepřehledňuje.

```
<?php
// v budoucnu může kolidovat
class Database {
}

// dlouhý a nepřehledný identifikátor
class MyAppDatabase {
}
```

```
?>
```

Čisté řešení tohoto problému nabízí jmenné prostory, které jsou k dispozici od PHP 5.3.

```
<?php
namespace MyApp;
class Database {
}
new Database; // použití zevnitř jmenného prostoru
new \MyApp\Database; // použití kdekoliv
?>
```

13 Velikost písmen v SQL dotazech

V SQL dotazech je zvykem klíčová slova uvádět velkými písmeny a vlastní identifikátory (především názvy tabulek a sloupců) malými písmeny. Slova se obvykle oddělují podtržítkem. Někdo se kloní k oddělování slov zvětšením písmene stejně jako v PHP kódu, s tím ale mohou být za určitých okolností problémy (např. MySQL v závislosti na konfiguraci může názvy tabulek převádět na malá písmena).

Pokud se u identifikátorů rozhodneme používat češtinu, je zvykem ji psát bez diakritiky, i když by si většina databázových serverů poradila i s ní. V komentářích tabulek a sloupců je naopak diakritiku vhodné používat.

14 Jak vypadá Spaghetti code

O špagetovém kódu mluvíme tehdy, když jsou jednotlivé části aplikace promíchané tak, že se v nich dá jen těžko vyznat. U webových aplikací je obzvlášť jednoduché takový kód vytvořit, protože se v nich používá řada jazyků zapisovaných v prostém textu:

- **JavaScript** je vhodné vyčlenit do samostatného souboru, který z HTML odkážeme značkou `<script src="">`. Přímo v HTML kódu je vhodné uvádět pouze inicializaci proměnných JavaScriptu, které jsou specifické pro daného uživatele.
- **CSS** je také vhodné dát do samostatného souboru a odkázat ho značkou `<link rel="stylesheet" href="">`. Provázání s HTML kódem zajišťují názvy značek, tříd a identifikátorů.
- **HTML** je vhodné nemíchat s řídicím PHP kódem, ale vyčlenit ho do samostatné šablony. Ta může využívat nějaký šablonovací systém nebo může být zapsaná také v PHP, vždy ale platí, že už slouží jen k vypsání dat.
- **SQL** je užitečné centralizovat do tzv. *modelu* (což je část aplikace zajišťující především získávání a ukládání dat), případně se bez něj díky ORM můžeme dokonce obejít.

Výhodou rozdělení aplikace do jednotlivých částí je jejich menší složitost a snazší udržovatelnost. Umožňuje to také větší specializaci vývojářů, kdy nehrozí, že HTML kodér poškodí práci PHP programátora.

Poznámka: Některé ukázky kódu v knize se této zásady nebudou kvůli jednoduchosti držet.

15 Jak na odsazování kódu

Bílé znaky jako mezery a konce řádků se sice v PHP až na výjimky ignorují, i tak je ale vhodné kvůli čitelnosti dodržovat pevná pravidla jejich zápisu. Každý příkaz se obvykle píše na jeden řádek – dlouhé příkazy lze rozdělit, ale nikdy bychom neměli na jeden řádek dávat příkazů více. Bloky kódu je zvykem odsazovat, otevírací závorka se umisťuje obvykle na konec předchozího řádku, u funkcí a tříd ji lze dát i na samostatný řádek.

Poznámka: V této knize se budeme zásad správného formátování kódu držet, otevírací složená závorka bude vždy na konci předchozího řádku.

16 Znak pro odsazování kódu

O znacích používaných pro odsazování kódu se vedlo mnoho bezvýsledných debat. Někdo používá dvě mezery, někdo čtyři, někdo trvá na osmi mezerách. Osobně používám tabulátor z těchto důvodů:

- Každý programátorský editor dovoluje nastavit jeho zobrazovanou šířku. Ostatní editory ho obvykle zobrazují jako osm mezer.
- Odsazení je otázkou jednoho stisknutí klávesy v libovolných podmínkách.
- V uloženém souboru zabírá tabulátor jen jeden bajt.

17 Odsazování konstrukce switch

Formátování konstrukce `switch` je ošemetné, protože vytváří dvě úrovně. Někdo doporučuje zarovnávat `case` na stejnou úroveň jako `switch`, někdo o úroveň dál. Pokud je ve větvích `case` jen jeden příkaz, tak se mi zdá nejpřehlednější ho uvést rovnou na řádek `case`:

```
<?php
function numbers($number) {
    switch ($number) {
        case 0: return "nula";
        case 1: return "jedna";
        case 2: return "dva";
        default: return null;
    }
}
?>
```

18 Odsazování kódu v kombinaci s HTML

Pokud kombinujeme PHP kód s HTML, tak může být správné odsazování oříškem. Můžeme odsazovat buď pouze PHP kód, nebo pouze HTML, nejpřehlednější je ale asi odsazování obojího.

```
<!-- Odsazování pouze PHP -->
<?php foreach ($data as $row) { ?>
    <tr>
    <td><?php echo htmlSpecialChars($row["name"]); ?></td>
```

```
    </tr>
<?php } ?>

<!-- Odsazování pouze HTML -->
<?php foreach ($data as $row) { ?>
<tr>
    <td><?php echo htmlSpecialChars($row["name"]); ?></td>
</tr>
<?php } ?>

<!-- Kombinace odsazování -->
<?php foreach ($data as $row) { ?>
    <tr>
        <td><?php echo htmlSpecialChars($row["name"]); ?></td>
    </tr>
<?php } ?>
```

Situace se ovšem zkomplikuje, když chceme v bloku PHP kódu vypsat neuzavřenou značku, kterou uzavřeme později. Tam kombinace odsazování příliš dobře nefunguje.

19 Zarovnávání operátorů

S formátováním kódu není vhodné to přehánět. Někteří programátoři se vyžívají třeba v tom, že při více přiřazeních pod sebou zarovnávají operátor přiřazení:

```
<?php
$int   = 0;
$s     = "";
$array = array();
?>
```

S takovýmto odsazováním je spousta práce, která čitelnost žádným zásadním způsobem nezlepší. Problém nastane hlavně v případě, kdy chceme přiřadit další proměnnou s delším názvem – pak musíme zarovnání všude opravit.

Zcela tragicky takovéto zarovnání dopadne v případě, kdy se na zdrojový kód podíváme s proporcionálním písmem, které někteří programátoři používají (protože je čitelnější a vejde se ho na obrazovku víc).

```
<?php
$int   = 0;
$s     = "";
$array = array();
?>
```

Obrázek 1. Zobrazení kódu proporcionálním písmem v editoru SciTE

20 Odřádkování před operátorem

Pokud by jeden příkaz vytvořil příliš dlouhý řádek, lze ho rozdělit na více řádků. Odřádkování je vhodné udělat před operátorem a pokračování příkazu odsadit:

```
<?php
$mail
```

```
    ->setFrom("alice@example.com")
    ->addTo($email)
    ->setSubject("Pokusná zpráva")
;
?>
```

21 Jak na konce řádků

Na Windows se řádky ukončují znaky CR+LF (v PHP řetězci se zapisuje jako `\r\n`), na Linuxu pouze znakem LF (`\n`). PHP koncům řádek nepřikládá zvláštní význam, takže je mu jedno, co použijeme. Programátorské editory si poradí s oběma druhy konců řádků, primitivní textové editory (např. **Notepad** na Windows) vyžadují nativní konce řádků.

Kvůli menší velikosti se obvykle používá znak LF, každopádně je vhodné v celém souboru používat stejný znak.

22 Konec řádku na konci souboru

Poslední řádek souboru je vhodné ukončit koncem řádku (klávesa `Enter`). Usnadňuje to přidávání dalších příkazů a zpřehledňuje to rozdíly mezi verzemi, které něco doplňují na konec souboru.

23 Ukončení posledního prvku pole

Za posledním prvkem v definici pole je možné v PHP uvést čárku stejně jako za ostatními prvky. Velmi užitečné je to hlavně u definice složitějších polí, kde jednotlivé prvky napíšeme na samostatné řádky. Zjednodušuje se tím budoucí přidávání nebo prohazování prvků.

```
<?php
$array = array(
    "první",
    "druhý",
    "třetí",
);
?>
```

24 Pořadí parametrů funkcí

Při vytváření funkce s více než jedním parametrem bychom se měli zamyslet nad tím, v jakém pořadí budeme parametry předávat. U funkcí, které s něčím manipulují, je zvykem v prvním parametru předávat právě tuto hodnotu. Pokud funkce manipulují s objektem, tak je obvykle lepší funkci změnit na metodu tohoto objektu. Další parametry se obvykle uvádí od těch přesnějších po ty vágnější (nejdříve např. předáme ID a až potom název). Volitelné parametry uvádíme pochopitelně až na konci, jinak to v PHP ani nejde.

Funkce by každopádně neměly mít parametrů příliš mnoho, ukázkou z tohoto pohledu nevhodně navržené funkce je `setCookie`, která má 7 parametrů.

25 Pojmenování parametrů funkcí

PHP na rozdíl od některých jiných programovacích jazyků nedovoluje při volání funkce pojmenovat předávané parametry. Všechny parametry je potřeba předat v pořadí, které funkce definuje. Pokud má funkce více podobných parametrů za sebou, je vhodné si hodnotu parametru zapsat do proměnné s popisným názvem.

```
<?php
// nepřehledné
$sqlite->singleQuery($query, true, false);

// přehlednější
$firstRowOnly = true;
$decodeBinary = false;
$sqlite->singleQuery($query, $firstRowOnly, $decodeBinary);

// třetí možnost
$sqlite->singleQuery($query,
    $firstRowOnly = true,
    $decodeBinary = false
);

// využití komentáře
$sqlite->singleQuery($query,
    true, // firstRowOnly
    false // decodeBinary
);

// dá se využít i toho, že se řetězec převede na pravdivostní hodnotu
$sqlite->singleQuery($query, "firstRowOnly", false);
?>
```

Dobrý editor zdrojového kódu nám samozřejmě význam parametru odhalí, ale při prohlížení kódu nemusí být editor k dispozici (ukázky kódu v článcích, prohlížení kódu z repozitáře, atd.).

26 Závorky kolem ternárního operátoru

Kolem ternárního operátoru (neboli podmíněného výrazu) je vhodné dělat závorky, což zlepší jeho čitelnost:

```
<?php
$a = ($m ? f() : ($n ? g() : $o));
?>
```

Ještě přehlednější je v takovýchto případech použít řídicí konstrukci `if`.

27 Použít apostrofy nebo uvozovky?

PHP dovoluje pro zápis řetězců používat uvozovky nebo apostrofy. Uvnitř uvozovek lze používat proměnné a speciální znaky, uvnitř apostrofů ne. Při výběru zápisu bychom

se měli řídit hlavně přehledností. Pokud vypisovaný kód obsahuje uvozovky, tak bývá přehlednější ho uzavřít do apostrofů, jinak budeme v této knize používat uvozovky.

```
<?php
echo '<option value="">(vyberte)</option>';
echo "<div>$html</div>\n";
?>
```

28 Řídicí konstrukce s bloky kódu

Pokud za řídicími konstrukcemi jako `if`, `while` nebo `foreach` uvedeme jenom jeden příkaz, nemusíme ho uzavírat do bloku kódu (složených závorek). I tak je vhodné to ale udělat, protože se tím zlepší přehlednost kódu a usnadní se jeho další modifikace (přidání příkazu).

```
<?php
// přehledný a snadno rozšiřitelný kód
if ($valid) {
    f();
}

// funkce se zavolá vždy, protože se vykoná pouze prázdný příkaz (;)
if ($valid); // chyba
    f();
?>
```

29 Volání funkce před její definicí

PHP dovoluje použít funkci v kódu dříve, než je definovaná. Definice funkce totiž vzniká ve fázi kompilace skriptu, ale zavolá se až ve fázi spuštění. Lepší je ale obvykle funkci nejprve definovat a pak teprve zavolat. Pokud bychom se totiž v budoucnu rozhodli funkci přesunout do vloženého souboru, tak přestane fungovat (vkládané soubory se totiž kompilují až při spuštění hlavního skriptu).

```
<?php
// bude fungovat
function defineFirst() {
}
defineFirst();

// bude fungovat
runFirst();
function runFirst() {
}

// bude fungovat
include "defineFirst.inc.php";
defineFirst();

// nebude fungovat
runFirst();
include "runFirst.inc.php";
?>
```

Do souboru s definicí funkcí je lepší nedávat žádný globální kód. Když toto pravidlo, např. u jednoduchého skriptu spouštěného z příkazové řádky, porušíme a v jednom souboru nadefinujeme funkce i globální kód, tak je zvykem nejprve uvádět definice konstant a funkcí a pak teprve spouštěný kód.

30 V jakém pořadí definovat metody

Na pořadí definic metod stejně jako u funkcí nezáleží, kvůli čitelnosti je ale vhodné dodržovat určitá pravidla. Pořadí definic ve třídě může být takovéto:

- Nejprve vlastnosti v pořadí `public`, `protected`, `private`.
- Konstruktor, destruktor a další magické metody.
- Další metody seřazené opět podle viditelnosti a podle logických celků.

31 Funkce vypisující data

Ve většině programovacích jazyků je lepší se vyhnout vytváření funkcí, které přímo vypisují nějaká data. Co kdybychom pak výstup chtěli použít někde jinde, třeba ho umístit do e-mailu? Místo toho proto funkce obvykle vrací řetězec, který vypíše až volající.

V PHP se velmi pohodlně kombinuje textový výstup s programovým kódem a jakýkoliv výstup lze snadno zachytit. Proto lze ospravedlnit i vytváření funkcí, které něco přímo vypisují. Pokud bychom náhodou s výstupem chtěli udělat něco jiného, můžeme použít *output buffering*:

```
<?php
ob_start();
phpInfo();
$phpInfo = ob_get_clean();
?>
```

32 Postradatelné proměnné a funkce

Pokud výsledek nějakého výpočtu používáme na více místech, je vhodné si ho přiřadit do proměnné. Pokud stejnou posloupnost příkazů používáme víckrát, je vhodné je uzavřít do funkce a volat tuto funkci. Když více funkcí pracuje se stejným objektem, je vhodné je uzavřít do třídy. Tyto zásady vedou k vytváření přehledného, výkonného a snadno udržitelného kódu.

Co když ale výsledek výpočtu používáme jen na jednom místě, co když funkci voláme jen jednou? Vyplatí se proměnnou nebo funkci vytvářet v takovém případě? Z pohledu běhu programu to je zbytečné, výkonnost to zcela nepatrně zhorší, ale program to obvykle zpřehlední – kus kódu tak vlastně dostane své jméno. Porovnejme následující tři ukázky kódu:

```
<?php
// nejefektivnější, ale nepřehledný kód
echo simplexml_load_file(mysql_result(mysql_query("
```

```
    SELECT filename
    FROM xml
    WHERE id = $id
"), 0))->title;

// pojmenování jednotlivých výpočtů
$result = mysql_query("SELECT filename FROM xml WHERE id = $id");
$filename = mysql_result($result, 0);
$xml = simplexml_load_file($filename);
echo $xml->title;

// pojmenování celé operace
function getTitleFromXml($id) {
    $result = mysql_query("SELECT filename FROM xml WHERE id = $id");
    $filename = mysql_result($result, 0);
    $xml = simplexml_load_file($filename);
    return $xml->title;
}
echo getTitleFromXml($id);
?>
```

Postradatelné proměnné a funkce jsou vlastně způsob dokumentování kódu. Pokud si tedy nějaký blok kódu zaslouží komentář, může ho zpřehlednit také vyčlenění do funkce nebo pojmenování vypočtených hodnot.

Běhové prostředí

33 Použití PHP pro webové aplikace

Nejběžnější použití PHP je pro tvorbu webových aplikací. K tomu je potřeba provozovat webový server (např. **Apache** nebo **IIS**), se kterým lze PHP propojit třemi základními způsoby:

- *CGI* je nejzákladnější způsob, kdy se pro zpracování každého skriptu spouští vlastní interpret PHP. V praxi se tento způsob v podstatě nepoužívá, protože je velmi pomalý.
- *Modul webového serveru* je nejběžnější způsob propojení PHP s webovým serverem Apache. Díky těsné vazbě na webový server jde o velmi výkonné řešení.
- *FastCGI* nechává interpret PHP spuštěný na pozadí a předává mu skripty, které je potřeba zpracovat. Díky tomu je toto řešení srovnatelně rychlé jako modul webového serveru, díky volnější vazbě s webovým serverem ale navíc umožňuje skripty spouštět pod jejich vlastníkem (v modulu všechny skripty běží pod společným uživatelem). To je užitečné hlavně na sdíleném webhostingu.

34 Použití PHP z příkazové řádky

PHP lze spouštět i z příkazové řádky. Pro toto použití se používá termín *CLI* (*Command Line Interface*). Hodí se hlavně pro spouštění testů nebo třeba pravidelně spouštěných úloh.

35 Syntaktická kontrola skriptu

Syntaktickou správnost skriptu lze ověřit bez jeho spuštění příkazem `php -l`. Tuto kontrolu je vhodné provádět před jakoukoliv publikací skriptu (ať už třeba uložením do verzovacího systému nebo nahráním na server) nejlépe automaticky. Pokud totiž syntaktická kontrola neprojde, tak se skript vůbec nespustí.

Poznámka: V PHP 5 existovala také funkce `php_check_syntax`, v PHP 5.0.5 ale byla odstraněna, protože zpřístupnila funkce a třídy definované v kontrolovaném souboru.

36 Jiná konfigurace při spuštění z příkazové řádky

Při spuštění z příkazové řádky lze nastavit odlišný konfigurační soubor nebo samotnou konfigurační direktivu. Zajišťuje to parametr `-c` resp. `-d`.

```
# celý konfigurační soubor
php -c php.ini file.php

# jedna nebo více konfiguračních direktiv
php -d safe_mode=0 file.php
```

37 Spuštění PHP kódu bez vytváření souboru


pokročilý

Pro spuštění PHP kódu přímo z příkazové řádky bez vytváření souboru lze použít parametr `-r`. Tomu se předává samotný PHP kód, který se vyhodnocuje už v kontextu PHP značek, ty se tedy neuvádí.

```
php -r 'phpcredits(CREDITS_DOCS);'
```

Velký pozor je potřeba dát na ošetřování znaků, které mají speciální význam jak na příkazové řádce, tak v PHP kódu.

38 Interaktivní vyhodnocování kódu


pokročilý

Parametrem `-a` příkazového řádku lze spustit tzv. interaktivní režim, kdy jsou příkazy vyhodnocovány bezprostředně po jejich zapsání. Tento režim se dá použít především pro různé pokusy, osobně ale preferuji vytvoření souboru s testovaným kódem a jeho běžné spuštění, protože se v něm snáze dělají změny.

39 Vstup a výstup z příkazového řádku


pokročilý

Při spuštění z příkazového řádku můžeme pracovat se standardním vstupem, standardním výstupem a chybovým výstupem pomocí konstant `STDIN`, `STDOUT` a `STDERR`. Tyto prostředky ani není vhodné zavírat.

```
<?php
// načtení celého standardního vstupu
$input = stream_get_contents(STDIN);
?>
```

Pokud bychom s těmito vstupy a výstupy chtěli pracovat i mimo příkazový řádek, můžeme si otevřít virtuální soubory `php://stdin`, `php://stdout` a `php://stderr`, obvykle to ale není potřeba.

40 Chyba skriptu spuštěného z příkazového řádku


začátečník

Pokud program z příkazového řádku nemůže pokračovat (třeba proto, že chybí povinný parametr), tak je vhodné skript ukončit nenulovým stavovým kódem. To zajistí příkaz `exit` s číselným parametrem. Ostatní programy příkazového řádku pak podle tohoto stavového kódu mohou poznat, jestli skript uspěl nebo ne.

```
<?php
if ($argc != 2) {
    echo "Použití: php example.php parametr\n";
    exit(1);
}
?>
```

Při spuštění externího programu z PHP skriptu můžeme stavový kód získat např. v druhém parametru funkce `system`.

Poznámka: Příkaz `die` je *aliasem* příkazu `exit`.

41 Použití chybového výstupu

Pokud program příkazového řádku kromě běžného výstupu vypisuje také chybová hlášení, je vhodné je vypisovat na samostatný chybový výstup. Při běžném spuštění se sice oba výstupy (běžný i chybový) zobrazí společně na obrazovce, při přesměrování výstupu je možné je ale rozdělit.

```
<?php
if (!is_readable($filename)) {
    // nevhodné řešení
    echo "Soubor $filename nelze přečíst.\n";

    // lepší řešení
    fWrite(STDERR, "Soubor $filename nelze přečíst.\n");
}
?>
```

42 Načtení vstupu z webové služby

Pokud skriptu pošleme data z formuláře metodou POST, tak je PHP rozebere do proměnné `$_POST`. Když ale skript dostane data v jiném formátu, než který používají formuláře (např. XML z webové služby nebo požadavku AJAX), tak potřebujeme načíst jejich původní podobu.

K načtení dat odeslaných metodou POST můžeme použít virtuální soubor `php://input`. Za tímto účelem lze použít i proměnnou `$HTTP_RAW_POST_DATA`, její naplňování se ale dá zakázat konfigurační direktivou `always_populate_raw_post_data`.

```
<?php
$xml = simplexml_load_file("php://input");
?>
```

Poznámka: Ani jeden zdroj dat není k dispozici, pokud jsme z formuláře dovolili odeslání souboru (typ `multipart/form-data`).

43 Zápis na výstup z webové aplikace

Pokud chceme nějaká data zapsat na výstup, můžeme k tomu použít kromě prostého vypsání příkazem `echo` i virtuální soubor `php://output`. Hodí se to v situaci, kdy s výstupem potřebujeme pracovat jako se souborem.

```
<?php
// vypsání binárního objektu databáze Oracle
$ociLob->export("php://output");
?>
```

44 Načtení konfigurace PHP


pokročilý

PHP hledá konfigurační soubor na několika místech:

1. V adresáři určeném konfigurační direktivou `PHPIniDir` webového serveru Apache.
2. Podle proměnné prostředí `PHPRC`.
3. Na Windows se cesta hledá také v registrech v klíči `HKLM\SOFTWARE\PHP\IniFilePath` (za větví `PHP` může následovat ještě číslo verze ve formátu `x.y.z`, `x.y` nebo `x`).
4. Aktuální pracovní adresář.
5. Adresář webového serveru.
6. Nakonec se soubor hledá v adresáři `C:\Windows` na Windows, jinde podle nastavení kompilační volby `--with-config-file-path`.

V těchto adresářích se hledá nejprve soubor `php-SAPI.ini`, kde SAPI je kód běhového prostředí (např. `apache` nebo `cli`), a potom soubor `php.ini`. Použitý soubor nám sdělí funkce `phpInfo`.

45 Detekce běhového prostředí


znalec

Skript můžeme napsat tak, aby se dal spustit z různých běhových prostředí, typicky přes webový server a z příkazové řádky. Pro zjištění běhového prostředí můžeme použít konstantu `PHP_SAPI` (nebo funkci `php_sapi_name`), při spuštění z příkazové řádky má hodnotu `"cli"`.

Podle běhového prostředí můžeme upravit výstup.

```
<?php
if (PHP_SAPI == "cli") {
    echo "$error\n";
} else {
    echo "<p>" . htmlSpecialChars($error) . "</p>\n";
}
?>
```

Poznámka: Funkce `phpInfo` je ukázkou vestavěné funkce, která se chová různě v různých prostředích. Při spuštění přes webový server vypíše úhlednou tabulku, při spuštění z příkazové řádky pouze jednoduchý textový výstup.

46 Definice konfiguračního souboru


začátečník

S PHP se distribuují konfigurační soubory `php.ini-development` a `php.ini-production`, které se dají použít jako základ konfigurace pro vývojové a produkční prostředí. Tyto soubory jsou velmi obsáhlé, obsahují bohaté komentáře a nastavují většinu konfiguračních direktiv včetně těch, které aplikace nastavit nepotřebuje, případně odpovídají výchozí hodnotě. Konfigurační soubor se značně zpřehlední, pokud se v něm nastaví jen direktivy nutné pro provoz aplikací na daném serveru. Volitelně je možné nenastavovat ani direktivy na jejich výchozí hodnotu, ta se ale v budoucích verzích PHP může změnit, takže pokud na nich aplikace závisí, tak je lepší je explicitně uvést.

47 Pravidelné spouštění úloh

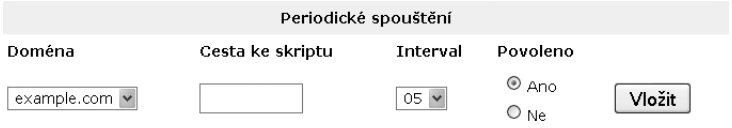
Pro pravidelné spouštění úloh se obvykle používá aplikace **cron**. Ta dovoluje v daný čas a ve stanoveném intervalu spouštět určené programy. Používá se např. pro pravidelný import dat nebo naopak pro mazání zastaralých dat.

Spouštěný program by v případě úspěchu neměl nic vypisovat, protože jakýkoliv výstup se považuje za chybu a odešle se e-mailem uživateli.

Poznámka: Na Windows lze k pravidelnému spouštění využít ovládací panel Naplánované úlohy.

48 Pravidelné spouštění úloh z webu

Někdy může být výhodné spouštět pravidelné úlohy nikoliv z příkazového řádku, ale přímo v kontextu webové aplikace. Může to být potřeba kvůli oprávněním nebo kvůli zajištění totožné konfigurace. Tomuto způsobu spouštění se obvykle říká **WebCron** a některé webhostingy ani jiný způsob pravidelného spouštění nenabízejí.



Obrázek 2. Formulář pro pravidelné spouštění úloh aplikace WebControl

49 Aktuální adresář při spuštění skriptu

Při spuštění PHP skriptu z příkazové řádky (obzvlášť ze služby **cron**) je běžné, že aktuální pracovní adresář není nastaven na adresář, ve kterém je umístěn skript. Relativní cesty se vyhodnocují vzhledem k aktuálnímu pracovnímu adresáři, takže když takovéto cesty ve skriptu používáme, nemusí skript při spuštění z příkazového řádku pracovat.

Spolehlivější je tedy definovat všechny cesty jako absolutní. Můžeme k tomu využít cestu k momentálně zpracovávanému skriptu, která je vždy uložena v „konstantě“ `__FILE__`. Adresář skriptu můžeme z této konstanty získat funkcí `dirName`, od PHP 5.3 se dá použít také přímo konstanta `__DIR__`.

Poznámka: Není vhodné pracovní adresář měnit funkcí `chDir`, protože navazující programy mohou počítat s tím, že jsou spuštěny z daného adresáře.

50 Co je to PHP-GTK

Projekt **PHP-GTK** se dá použít pro vývoj desktopových aplikací v PHP s využitím grafického prostředí GTK. To se používá hlavně na Linuxu, je ale k dispozici i pro Windows nebo OS X. Desktopové aplikace mají proti webovým tu výhodu, že nemusí

ze serveru přenášet uživatelské rozhraní, a to má také bohatší výrazové prostředky. Webové aplikace mají zase tu výhodu, že se dají spustit odkudkoliv a díky AJAXu a dalším technologiím nabízí solidně rychlé a bohaté prostředí také. PHP se proto pro vývoj desktopových aplikací v praxi příliš nepoužívá.

```
<?php
$window = new GtkWindow;
$window->set_title("Ahoj světe!");
$window->connect_simple('destroy', array('Gtk', 'main_quit'));

$label = new GtkLabel("Jen jsme chtěli říct\r\n'Ahoj světe!'");
$window->add($label);

$window->show_all();
Gtk::main();
?>
```

51 Jak na PHP pro Javu

Kromě standardní implementace PHP vytvořené v jazyce C existuje i verze pro Javu. Aplikace, která dovoluje PHP skripty spouštět v prostředí Javy, se jmenuje **Quercus** a využívá se především pro spouštění PHP skriptů v prostředí **Google App Engine**, které dovoluje provozovat vlastní aplikace na serverech firmy Google a snadno je škálovat.

52 Jak na PHP pro .NET

PHP skripty lze spouštět také v rámci frameworku **.NET**. Využít se k tomu dá kompilátor **Phalanger**, jehož hlavní vývojáři pochází z České republiky. Toto běhové prostředí se dá použít především pro vytváření desktopových aplikací, které mohou využívat plnou sílu frameworku **.NET**, např. včetně komunikace se zvukovou kartou.

Toto prostředí se dá použít také pro vytváření **Silverlight** aplikací v PHP.

Instalace a konfigurace

53 Jak zjistit verzi PHP

Kód je vhodné psát tak, aby fungoval ve všech podporovaných verzích PHP. Použitou verzi můžeme zjistit z konstanty `PHP_VERSION` (případně z funkce `phpVersion`), od PHP 5.2.7 můžeme použít také konstanty `PHP_MAJOR_VERSION`, `PHP_MINOR_VERSION`, `PHP_RELEASE_VERSION` a `PHP_EXTRA_VERSION`, kde jsou jednotlivé části čísla verze k dispozici samostatně. Od PHP 5.2.7 je k dispozici také konstanta `PHP_VERSION_ID` vyjadřující verzi jako celé číslo (např. 50207).

Verze v PHP mají tvar `5.3.2RC1` a je potřeba je porovnávat funkcí `version_compare`.

54 Aktuální verze PHP

Téměř každá nová verze PHP opravuje řadu bezpečnostních chyb, je proto vhodné používat vždy aktuální verze. Aktuální verze ale není obvykle jen jedna, protože vývojáři udržují více větví PHP po delší dobu. V jeden okamžik tak mohou být aktuální např. verze 5.3.2 i 5.2.13. Linuxové distribuce mohou navíc udržovat bezpečnost i starších verzí, takže ve stejný okamžik může být z pohledu bezpečnosti aktuální i verze 5.2.6-1+lenny8.

55 Nastavení konfigurace PHP

PHP dovoluje konfiguraci nastavit na několika místech. Vedle vlastního konfiguračního souboru `php.ini` ji lze nastavit v konfiguraci webového serveru – tyto dva způsoby vyžadují pro načtení konfigurace restart webového serveru. Webový server může dovolit nastavit konfiguraci i pro adresář (typicky v souboru `.htaccess`), tento způsob může být ale zakázaný. A konečně některé konfigurační direktivy lze nastavit i funkcí `ini_set`, ta ale na některých hostinzích může být také zakázaná. Proto existují i funkce, které vybrané konfigurační direktivy nastavují přímo (např. `session_set_cookie_params`).

Možnost nastavení jednotlivých konfiguračních direktiv je popsaná v manuálu PHP.

56 Konfigurace pro adresář

Pokud PHP skript běží jako modul webového serveru Apache s povolenou konfigurační direktivou `AllowOverride`, tak je možné v souboru `.htaccess` definovat konfiguraci specifickou pro aktuální adresář a jeho podadresáře. Vedle konfigurace webového serveru lze pomocí direktiv `php_value` a `php_flag` nastavovat i konfiguraci PHP. Nejčastěji se v tomto souboru nastavují konfigurační direktivy, které se projevují ještě před spuštěním skriptu jako `magic_quotes_gpc` nebo `register_globals`, ale lze nastavit i většinu ostatních konfiguračních direktiv (ty, které mají měnitelnost `PHP_INI_PERDIR`).

```
php_flag magic_quotes_gpc Off
php_value error_reporting 6135
```

Poznámka: Ve webovém serveru IIS slouží ke stejnému účelu soubor `Web.config`.

57 Centrální konfigurace pro adresář

Povolení souboru `.htaccess` je velmi praktické, ale vede k jistému zpomalení, protože webový server při každém požadavku prochází všechny adresáře od aktuálního až po kořenový a hledá v něm tento soubor. Odlišná konfigurace pro jednotlivé adresáře serveru se proto dá nastavit i v konfiguračním souboru Apache pomocí direktivy `<Directory>`.

V konfiguračním souboru `php.ini` se dá nastavit odlišná konfigurace pro jednotlivé adresáře v sekci `[PATH]`. Tuto možnost lze použít při spuštění PHP přes *FastCGI* a je k dispozici od PHP 5.3.

Poznámka: Nevýhodou centrální konfigurace může být to, že k jejímu načtení musíme restartovat server.

58 Vynucení konfigurace

Pokud nechceme, aby uživatelé přepsali hodnotu nějaké konfigurační direktivy, můžeme ji nastavit v konfiguraci webového serveru Apache pomocí direktiv `php_admin_value` a `php_admin_flag`. Takto nastavené hodnoty nelze přepsat v souboru `.htaccess` ani funkcí `ini_set`.

59 Použití konstant v konfiguraci PHP

Některé konfigurační direktivy PHP přijímají číselnou hodnotu, která se může vyjádřit jako bitová maska konstant. Typickým zástupcem takové direktivy je `error_reporting`. Konstanty můžeme samozřejmě používat při volání funkce `ini_set`, dají se použít také v souboru `php.ini`, nelze je ale použít v konfiguračních souborech Apache (typicky hlavně `.htaccess`), kde je nutné použít číslo.

60 Použití jednotek v konfiguraci PHP

Direktivy ovlivňující množství zabrané paměti nebo místa na disku dovolují za číslem uvést jednotku. Ta může být `K` (1024 bajtů), `M` (1024 KB) nebo `G` (1024 MB). Velikost písmen se nerozlišuje a jednotky lze použít i při nastavení z konfiguračních souborů webového serveru.

Funkce `ini_get` vrací hodnotu tak, jak byla zadaná.

61 Pravdivostní hodnoty konfiguračních direktiv


pokročilý

Pravdivostní konfigurační direktivy mohou být v konfiguračních souborech webového souboru **Apache** nastaveny pomocí `php_flag`. Nastavení pomocí `php_value` bude také fungovat, ale funkce `ini_get` bude vracet řetězec. Pokud tedy ve skriptu chceme spolehlivě zjistit hodnotu pravdivostní konfigurační direktivy, musíme zohlednit všechny hodnoty, které se vyhodnotí jako `true`.

```
<?php
/** Zjištění pravdivostní hodnoty konfigurační direktivy
* @param string
* @return bool
*/
function getIniBool($ini) {
    $value = strToLower(ini_get($ini));
    return (intVal($value)
        || $value == "on"
        || $value == "true"
        || $value == "yes"
    );
}
?>
```

62 Otevírání vzdálených souborů


pokročilý

Konfigurační direktivou `allow_url_fopen` můžeme v PHP povolit otevíraní vzdálených souborů přístupných protokolem HTTP a několika dalšími protokoly.

```
<?php
file_get_contents("http://www.example.com");
?>
```

Pokud je tato direktiva vypnutá, můžeme ke stažení vzdálených souborů použít extenzi **cURL**.


Poznámka: Některé ukázky v této knize předpokládají zapnutí této direktivy.

63 Vkládání vzdálených souborů


pokročilý

Možnost vkládání vzdálených souborů konstrukcemi `include` a spol. ovlivňuje od PHP 5.2 direktiva `allow_url_include`, která je ve výchozím nastavení vypnutá (ve starších verzích se k tomu používala direktiva `allow_url_fopen`). Vzhledem k tomu, že prakticky nedává smysl spouštět vzdálené soubory jako PHP kód, není důvod tuto direktivu zapínat.

I tak je ale nezbytné vkládané soubory kontrolovat, protože útočník může pro útok využít soubor na disku serveru.

64 HTML nebo XHTML?

Webové stránky se obvykle posílají v jazyce HTML nebo v jeho alternativním zápisu XHTML. Ten je o něco přísnější – vyžaduje například uzavírání všech značek a obalování všech hodnot atributů do uvozovek nebo apostrofů. Stránky v jazyce XHTML by se ideálně měly posílat s typem `application/xhtml+xml`, k jehož odeslání lze použít HTTP hlavičku `Content-Type`. Tento typ ale dělá potíže starším verzím **Internet Exploreru**, proto se obvykle neposílá. Prohlížeče pak pracují s XHTML dokumentem jako s HTML.

Výběr varianty jazyka musíme sdělit i některým funkcím, např. `nl2br`.

Poznámka: Tato kniha uvádí ukázky kódu v jazyce XHTML.

65 Výběr kódování

Pro kódování českých znaků se tradičně používají kódování `windows-1250` (používané hlavně na Windows), `iso-8859-2` (používané hlavně v Linuxu) a `utf-8` (způsob kódování znakové sady Unicode). První dvě zmíněná kódování ukládají každý znak do jednoho bajtu a dají se použít pro středoevropské jazyky. Třetí uvedené kódování je univerzální a dovoluje uložit prakticky všechny znaky, které se na světě používají. Využívá k tomu proměnlivý počet bajtů na znak – písmena bez diakritiky se ukládají do jednoho bajtu, písmena s diakritikou do dvou bajtů a například čínské znaky do tří bajtů. České texty uložené v kódování UTF-8 jsou proto mírně delší než v jednobajtových kódováních, v HTML dokumentu představuje nárůst asi jen 4 %.

Vzhledem k univerzálnosti je kódování UTF-8 nejlepší současná volba.

66 Nastavení kódování

V PHP se výstupní kódování nastavuje konfigurační direktivou `default_charset` a mělo by být vždy nastaveno, protože jinak kódování určí prohlížeč automatickou detekcí, která nemusí být vždy spolehlivá.

Přímo ze skriptu můžeme kódování určit funkcí `header("Content-Type: text/html; charset=utf-8")`. Obdobným způsobem bychom měli kódování určit i u exportů, např. RSS.

Kromě toho bychom měli kódování specifikovat i v HTML značce `<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />`. To je důležité pro případ, kdy si uživatel uloží kopii stránky do souboru – prohlížeče v tom případě uloží pouze samotný HTML kód bez hlaviček.

67 Předefinování řetězcových funkcí

Extenze **MBstring** dovoluje pomocí konfigurační direktivy `mbstring.func_overload` předefinovat řetězcové funkce tak, aby respektovaly kódování nastavené touto extenzí. Tomuto předefinování je ale obvykle lepší se vyhnout, protože knihovny používané aplikací se mohou spoléhat na standardní chování řetězcových funkcí.

68 Jak na hlášení chyb

Pomocí konfigurační direktivy `error_reporting` nebo stejnojmenné funkce můžeme ovlivnit, o jakých chybách chceme být informováni. Chyby se dají rozdělit do tří základních skupin:

- Fatální chyby – `E_ERROR`, `E_USER_ERROR`
- Varování – `E_WARNING`, `E_USER_WARNING`
- Poznámky – `E_NOTICE`, `E_USER_NOTICE`

Fatální chyby způsobí ukončení běhu programu, varování nás upozorňují na jasné chyby, které ale neukončí skript, a poznámky nás upozorňují na obraty, které můžou být chybou, ale nemusí. Poznámky upozorňují především na práci s neinicializovanými proměnnými. Takové proměnné mají v PHP hodnotu `null`, ale zapomenutí inicializace proměnné může vést k těžko odhalitelné chybě (např. překlep).

Důsledná inicializace proměnných nicméně může kód znepřehlednit. Představme si, že chceme do košíku (reprezentovaného session proměnnou) přidat jeden kus nějakého produktu. V PHP k tomu můžeme použít následující kód:

```
<?php
$_SESSION["basket"][$id_produkt]++;
?>
```

Kód je přehledný a jednoznačný, pokud ale produkt v košíku ještě nebude, tak PHP vygeneruje poznámku. Aby se tato poznámka nevygenerovala, je potřeba kód přepsat:

```
<?php
if (isset($_SESSION["basket"][$id_produkt])) {
    $_SESSION["basket"][$id_produkt]++;
} else {
    $_SESSION["basket"][$id_produkt] = 1;
}
?>
```

Takto zapsaný kód je mnohem delší a méně přehledný. Příklady v této knize jsou nicméně napsány tak, aby žádné chyby včetně poznámek negenerovaly, mohou se ale spoléhat na existenci proměnných, které používají.

Poznámka: Existují i další chybové úrovně (např. `E_STRICT` nebo `E_RECOVERABLE_ERROR`) a skoro každá větší verze PHP nějakou novou zavádí.

69 Práce s neinicializovanými proměnnými

Upozornění na práci s neinicializovanými proměnnými je hlavním úkolem chybové úrovně `E_NOTICE` (poznámky), PHP to ale bohužel dělá špatně ze dvou důvodů:

1. Na práci s neinicializovanou proměnnou jsme upozorněni až v době spuštění kódu, což vzhledem k bezpečnostním rizikům tohoto přístupu může být pozdě.

2. O některých chybných obratech se nedozvíme (např. přidání prvku do neinicializovaného pole).

Pro kontrolu práce s proměnnými bez obou neduhů lze použít program **php-initialized**, který skript zkontroluje bez jeho spuštění. Autorem tohoto programu je autor této knihy.

70 Proměnná s chybovou hláškou


pokročilý

Konfigurační direktivou `track_errors` můžeme dosáhnout toho, že se poslední vyvolaná chyba uloží do proměnné *$php_errormsg*. Tuto proměnnou můžeme využít např. pro zaznamenání chyby.

Od PHP 5.2 lze použít také funkci `error_get_last`, která o chybě vrací i další informace (typ chyby a soubor a řádek, kde k chybě došlo).

71 Využití obsluhy chyby pro získání chybové hlášky


pokročilý

Pokud se nemůžeme spolehnout na zapnutí konfigurační direktivy `track_errors` a nechceme ovlivňovat okolní prostředí jejím povolením (třeba v kódu knihovny), tak můžeme pro zachycení chybové hlášky použít vlastní obsluhu chyb:

```
<?php
class Error {
    private $error = "";

    function _errorHandler($level, $error) {
        $this->error = $error;
    }

    function trigger() {
        set_error_handler(array($this, '_errorHandler'));
        // zde bude zavolání kódu, který může obsahovat chybu
        pg_connect("");
        restore_error_handler();
        echo "$this->error\n";
    }
}

$error = new Error;
$error->trigger();
?>
```

72 HTML kód v chybách


pokročilý

Konfigurační direktiva `html_errors` způsobuje, že se chybové zprávy vytváří ošetřené pro HTML. To je velmi praktické při jejich vypisování, ale nepraktické při jakékoliv jiné manipulaci s nimi. Pokud nechceme ovlivňovat okolní prostředí vypnutím této direktivy, můžeme HTML ošetřování zneutralizovat:

```
<?php
$error = $php_errormsg;
if (ini_get("html_errors")) {
    $error = html_entity_decode(strip_tags($error));
}
// pokud nás navíc nezajímá ani funkce, ve které došlo k chybě
$error = preg_replace('~^[^:]*: ~', '', $error);
?>
```

73 Chyby v SQL dotazech

PHP ve výchozím nastavení nezobrazuje chyby v SQL dotazech. Funkce jako `mysql_query` a podobné v případě chyby pouze vrátí `false`. Konfigurační direktiva `mysql.trace_mode` se dá použít k automatickému vytváření PHP varování z SQL chyb.

Tato konfigurační direktiva ale zároveň vyvolá varování při kompletním procházení tabulek (což může být důležité kvůli výkonnosti) a pro výsledky neuvolněné funkcí `mysql_free_result` (to je důležité kvůli šetření paměti). První uvedená vlastnost je navíc realizována voláním příkazu `EXPLAIN` pro každý `SELECT`, což znemožňuje používat obraty jako `SELECT FOUND_ROWS()`.

V praxi se tato konfigurační direktiva téměř nepoužívá, a pokud chceme chyby v SQL příkazech hlásit, je vhodné si za tím účelem napsat vlastní funkci:

```
<?php
/** Položení MySQL dotazu s nahlášením chyby
* @param string
* @param resource mysql link
* @return resource mysql result
* @throws Exception v případě chyby v dotazu
*/
function mysqlQueryError($sql, $connection = null) {
    $args = func_get_args();
    $return = call_user_func_array("mysql_query", $args);
    if (!$return) {
        array_shift($args); // funkcím předáme pouze $connection
        $message = call_user_func_array("mysql_error", $args);
        $code = call_user_func_array("mysql_errno", $args);
        throw new Exception($message, $code);
    }
    return $return;
}
?>
```

Poznámka: Čistší řešení nabízí extenze MySQLi, kde lze ze základní třídy vytvořit potomka, který může chyby hlásit požadovaným způsobem.

74 Instalace PHP na vlastním počítači

Pokud chceme na vlastním počítači provozovat PHP aplikaci, můžeme si samostatně nainstalovat webový server, PHP a databázový server. Tato volba je obvyklá na Linuxu,

kde jsou aplikace distribuovány formou balíků nebo zdrojových kódů, můžeme ji ale využít i na Windows.

Druhou možností je využít nějakou aplikaci, která všechny produkty nainstaluje společně a navíc přidá služby pro jejich snadnou správu a konfiguraci. Mezi nejlepší takové balíky patří **XAMPP**, který je k dispozici pro Windows, Linux i další platformy. Pro Windows lze použít také **WampServer** nebo **VertrigoServ**.

75 Instalace PHP na IIS

PHP se nejčastěji používá spolu s webovým serverem **Apache**, srovnatelně výkonné je ale zhruba od verze 5.2 také pod serverem **IIS** na Windows. K jeho instalaci pod tímto serverem lze použít binární soubory PHP – od verze 5.3 se používá verze zkompilovaná jako *VC9 Non Thread Safe* (pro Apache se používá verze *VC6 Thread Safe*).

76 Instalace PHP a aplikací na IIS

K instalaci PHP pod webovým serverem **IIS** můžeme využít také **Microsoft Web Platform**. Ta umožňuje instalovat na server **IIS** webové aplikace napsané v různých programovacích jazycích včetně PHP. Kromě instalace aplikací se stará také o instalaci jejich závislostí, např. databázový server, frameworky nebo právě PHP. K instalaci PHP můžeme tedy použít i tuto platformu.

77 Jaká je licence PHP

PHP je k dispozici pod licencí *PHP License*. Ta se vztahuje na redistribuci zdrojových a binárních kódů PHP a na vytváření jeho odvozenin. Pokud bychom tedy chtěli vytvořit např. verzi PHP, která nebude obsahovat datový typ `resource`, tak bychom tuto licenci museli dodržet. Licence je poměrně velkorysá, pouze nedovoluje použít „PHP“ v názvu odvozeného produktu.

Na skripty vytvořené pro PHP se tato licence nevztahuje, a můžeme je tedy šířit s libovolnou licencí. Neplatilo by to pouze v situaci, kdy bychom je šířili jako jeden celek spolu s PHP.

78 Jaká je licence MySQL

MySQL je možné získat buď s licencí GPL nebo s komerční licencí. Licence GPL vyžaduje, aby program distribuovaný spolu s MySQL používal opět tuto licenci. To především znamená, že musíme dát k dispozici jeho zdrojový kód. Komerční licence to nevyžaduje.

Pokud svoji aplikaci pouze provozujeme na webovém serveru, tak k distribuci nedochází, a zdrojový kód tedy k dispozici dávat nemusíme. V tom případě můžeme použít MySQL s GPL licencí nezávisle na tom, zda jde o komerční nebo nekomerční aplikaci.

79 Nahrávání extenzí

Funkčnost PHP lze rozšiřovat pomocí řady existujících extenzí. Ty dovolí z PHP pracovat třeba s obrázky nebo s FTP. Ostatně i běžně používaná práce s databázemi je zajištěna extenzemi. Pro nahrání extenze můžeme použít konfigurační direktivu `extension`, kterou lze zapsat pouze do souboru `php.ini` (a ne např. do konfigurace webového serveru). Pro nahrání extenze se dá použít také funkce `dl`, ta je ale k dispozici jen při spuštění z příkazového řádku (navíc se dá zakázat pomocí `enable_dl`).

Poznámka: Pro nahrání extenzí rozšiřujících jádro PHP (jako jsou např. akcelerátory nebo debuggery) se používá konfigurační direktiva `zend_extension` (ve starších verzích PHP rozdělená podle způsobu kompilace).

80 Název souboru s extenzí

Skript spouštěný z příkazového řádku si může nahrát potřebné extenze funkcí `dl`. Problém je ten, že se soubory s extenzemi na různých operačních systémech jmenují různě. Na Windows začínají `php_` a mají koncovku `.dll`, na Linuxu mají koncovku `.so`. Správná koncovka je uložena v konstantě `PHP_SHLIB_SUFFIX`. Můžeme si proto napsat funkci, která tento rozdíl skryje:

```
<?php
/** Nahrání extenze podle názvu
* @param string
* @return int
*/
function loadExtension($name) {
    $filename = "$name." . PHP_SHLIB_SUFFIX;
    if (PHP_SHLIB_SUFFIX == "dll") {
        $filename = "php_$filename";
    }
    return dl($filename);
}
?>
```

Poznámka: Nahrané extenze můžeme otestovat funkcí `extension_loaded`.

81 Vývojové a produkční prostředí

Aplikaci bychom měli vždy vyvíjet na veřejně nedostupném místě (obvykle na vlastním počítači) a teprve po jejím úplném odladění bychom ji měli umístit na cílové místo. Těmto dvěma prostředím se říká vývojové a produkční.

Ve vývojovém prostředí je vhodné být ihned informován o všech chybách, ke kterým v programu dojde. Jejich zobrazení zajistí konfigurační direktiva `display_errors`. V produkčním prostředí je naopak vhodné všechny chyby touto direktivou skrýt. Jejich zobrazení totiž jednak působí velice amatérsky, jednak z nich mohou uniknout citlivé informace (např. přihlašovací údaje k databázovému serveru).

V produkčním prostředí je naopak vhodné zapnout logování chyb direktivou `log_errors`. Umístění logu můžeme ovlivnit direktivou `error_log`. V produkčním prostředí by měl log zůstávat pokud možno prázdný, což můžeme ověřit třeba tím, že si ho v případě neprázdnosti necháme zasílat.

Ve vývojovém prostředí je možné logování chyb zapnout taky, protože pokud se chyba zobrazí např. uvnitř hodnoty HTML atributu, můžeme ji snadno přehlédnout.

82 Odkazy z chybových hlášek

Chybové hlášky generované PHP obsahují odkazy do dokumentace. URL dokumentace můžeme nastavit direktivou `docref_root`, koncovku pomocí `docref_ext`:

```
docref_root = http://www.php.net/manual/en/
docref_ext = .php
```

Poznámka: Vypnutím volby `html_errors` můžeme vytváření odkazů úplně zrušit.

83 Konfigurace aplikace specifická pro server

Na vývojovém a produkčním serveru můžeme používat odlišnou konfiguraci, což můžeme ve skriptu zohlednit:

```
<?php
if ($_SERVER["SERVER_ADDR"] == "127.0.0.1") {
    // vývojový server
    define("DB_SERVER", "localhost");
    define("DB_USER", "root");
    define("DB_PASSWORD", "");
} else {
    // produkční server
    define("DB_SERVER", "db.example.com");
    define("DB_USER", "web");
    define("DB_PASSWORD", "GwqfUcEO");
}
?>
```

Skripty se velmi zjednoduší, když na vývojovém i produkčním serveru budeme používat stejnou konfiguraci. Nic nám např. nebrání si na vývojovém serveru vytvořit uživatele databáze se stejným jménem a heslem (a se stejnými právy) jako na ostrém serveru. S názvem počítače to je o něco složitější – ten si v souboru `etc/hosts` můžeme namapovat na vývojový databázový server, to má nicméně své nevýhody (např. aplikaci nespustíme bez tohoto nastavení).

84 Povolení ladicího režimu

Podle IP adresy serveru můžeme zjistit, zda aplikaci spouštíme na lokálním počítači nebo na veřejně dostupném serveru. Podle toho můžeme povolit ladicí režim.

```
<?php
define("DEBUG", $_SERVER["SERVER_ADDR"] == "127.0.0.1");
?>
```

Pokud je veřejně dostupný server sdílený, tak by měl poslouchat pouze na veřejné IP adrese, aby si jiný uživatel nemohl vytvořit požadavek na lokální počítač, a tím ladicí režim zapnout. U webového serveru **Apache** k tomu slouží nastavení `Listen`.

Poznámka: Lokálních adres je více, i když 127.0.0.1 je nejběžnější. Knihovna Nette Debug nabízí detekci na základě kompletního seznamu.

85 Nevhodné povolení ladicího režimu

Při nastavování ladicího režimu by nás mohlo napadnout orientovat se místo IP adresy podle názvu serveru:

```
<?php
// nepoužívat - skrývá nebezpečí
define("DEBUG", $_SERVER["SERVER_NAME"] == "localhost");
?>
```

Tento způsob detekce je ale lepší nepoužívat, protože si útočník může snadno určit (obvykle souborem `etc/hosts`), že jeho `localhost` míří na náš server, a tím podstrčit hodnotu proměnné *`$_SERVER`*`["SERVER_NAME"]`.

86 Registrace proměnných zvenku

Konfigurační direktiva `variables_order` slouží k určení vnějších zdrojů, pro které PHP vytvoří superglobální proměnné. Jde o řetězec sestavený z těchto znaků:

- `E` – proměnné prostředí (*`$_ENV`*).
- `G` – parametry URL (*`$_GET`*).
- `P` – data odeslaná metodou POST (*`$_POST`*).
- `C` – cookies (*`$_COOKIE`*).
- `S` – proměnné serveru (*`$_SERVER`*).

Pokud je zapnutá konfigurační direktiva `register_globals`, tak `variables_order` určuje také rozsah a pořadí registrovaných globálních proměnných (pozdější hodnoty přepisují dřívější).

Poznámka: Na registraci globálních proměnných bychom se neměli spoléhat (a `register_globals` je tedy lepší vypnout), takže na pořadí znaků v této direktivě by nám nemělo záležet. Většina webových aplikací se neobejde bez nastavení `variables_order = GPCS`.

87 Pořadí proměnných zvenku

Do pole *$_REQUEST* se ukládají data odeslaná uživatelem v pořadí určeném direktivou `request_order`, která je k dispozici od PHP 5.3 (ve starších verzích se k tomuto účelu používala direktiva `variables_order`). Hodnota `PG` tedy určí, že pole nebude obsahovat cookies a GET proměnné přepíšou POST proměnné stejného názvu.

Poznámka: Pole *$_REQUEST* je lepší nepoužívat, protože může obsahovat data z jiného zdroje, než očekáváme. Na obsahu této direktivy by nám tedy nemělo záležet.

88 Automatické ošetřování vstupu od uživatele

Pokud data od uživatele posíláme do databáze, musíme je ošetřit tak, aby nemohla ovlivnit samotný dotaz, což by bylo významné bezpečnostní riziko. Protože na to začínající programátoři často zapomínají, byla do PHP zavedena konfigurační direktiva `magic_quotes_gpc`, která se o ošetření postará automaticky. Takto ošetřená data lze ale použít jen v některých případech:

- Data se mohou použít pouze uvnitř SQL řetězce uzavřeného do apostrofů nebo uvozovek.
- Musíme používat jednobajtové kódování nebo UTF-8.
- Ošetření je plně kompatibilní pouze s databázovým serverem MySQL.

Důležité je zmínit, že pokud bychom data chtěli použít v jakémkoliv jiném kontextu (např. je vypsat do stránky), musíme je zase od-ošetřit funkcí `stripSlashes`.

Každá aplikace, která chce s daty od uživatele nějakým způsobem pracovat, musí vědět, jak je tato direktiva nastavena, případně si ji musí sama nastavit. To ale nejde udělat přímo ze skriptu, protože data se ošetřují ještě před jeho spuštěním.

Dá se říci, že tato konfigurační direktiva znehodnotí data od uživatele pro jedno konkrétní použití, a proto byla v PHP 5.3 označena jako zastaralá a z budoucích verzí bude odstraněna. Příklady v této knize počítají s tím, že je vypnuta.

89 Vypnutí automatického ošetřování vstupu od uživatele

Pokud má aplikace pracovat s daty od uživatele a chce být nezávislá na nastavení konfigurační direktivy `magic_quotes_gpc`, může si ošetření sama odstranit. K tomu ale nejde použít běžná funkce `ini_set`, protože při spuštění skriptu jsou data již ošetřena. Je proto nutné to udělat vlastním skriptem:

```
<?php
// odstranění ošetření dat způsobeného direktivou magic_quotes_gpc
if (get_magic_quotes_gpc()) {
    $process = array(&$_GET, &$_POST, &$_COOKIE, &$_REQUEST);
    while (list($key, $val) = each($process)) {
        foreach ($val as $k => $v) {
            unset($process[$key][$k]);
            if (is_array($v)) {
```

```
                $process[$key][stripSlashes($k)] = $v;
                $process[] = &$process[$key][stripSlashes($k)];
            } else {
                $process[$key][stripSlashes($k)] = stripSlashes($v);
            }
        }
    }
    unset($process);
}
?>
```

Poznámka: Automatické ošetřování pole *$_FILES* se v různých verzích PHP chová různě.

90 Automatické ošetření načítaných dat

Vedle dat od uživatele je možné automaticky ošetřovat i data načítaná z jiných zdrojů. Slouží k tomu konfigurační direktiva `magic_quotes_runtime`, ale k jejímu použití není rozumné opodstatnění. Pokud aplikace nemůže nastavení této direktivy ovlivnit konfigurací, může ji sama vypnout funkcí `set_magic_quotes_runtime` (ta je ovšem označena jako zastaralá).

91 Automatické zahájení session

Pomocí konfigurační direktivy `session.auto_start` lze zapnout automatické otevření session. Ve skriptech pak není nutné volat funkci `session_start` a session proměnné lze rovnou používat.

Ve většině případů je ale lepší tuto direktivu nezapínat, protože session nemusí být vždy potřeba a někdy je také potřeba ji ve skriptu nakonfigurovat. Otevření session konfigurační direktivou do nich také znemožňuje ukládat vlastní objekty, protože definice tříd se načte až později.

92 Platnost session proměnných

Session proměnné mají ve výchozím nastavení velmi omezenou platnost ovlivněnou dvěma faktory.

1. Jednak to je platnost session cookie nastavená direktivou `session.cookie_lifetime`, která má ve výchozím nastavení hodnotu 0, což znamená „do zavření prohlížeče“. To je obvykle vyhovující.
2. Druhá direktiva ovlivňující platnost session proměnných je `session.gc_maxlifetime` určující, jak stará session data se už mohou smazat. Tato direktiva má ve výchozím nastavení hodnotu 1440 sekund, což je jen 24 minut. Důležité je, že tuto direktivu nemůžeme samotnou nastavit jen pro část aplikace (např. administrační rozhraní), protože stará session data může smazat kterýkoliv skript. Tuto direktivu tedy musíme nastavit vždy stejně všem částem aplikace, které ukládají data do stejného adresáře (`session.save_path`).

Poznámka: K automatickému mazání starých session dat dochází jen při výchozím způsobu ukládání session proměnných do souboru. Pokud tento způsob změníme, tak si můžeme způsob mazání starých dat zajistit sami.

93 Mazání starých session dat

Při každém zavolání funkce `session_start` se PHP s pravděpodobností `session.gc_probability` / `session.gc_divisor` pokusí smazat stará session data. Za stará se považují ta, která nebyla použita po dobu `session.gc_maxlifetime` sekund.

K mazání tedy může dojít při spuštění libovolného skriptu, který používá session proměnné. Pokud je session dat hodně, tak ho tato operace může výrazně zdržet. Pro zlepšení odezvy serveru je tedy možné nastavit direktivu `session.gc_probability` na nulu a mazání starých session dat provádět nezávislou úlohou (nejspíš spouštěnou službou **cron**).

94 Přidání session identifikátoru do URL

Pomocí direktivy `session.use_trans_sid` lze dosáhnout automatického přidávání session identifikátoru u všech interních odkazů na stránce. Ty se do URL přidají v situaci, kdy PHP neobdrží session identifikátor z cookie. K tomu dojde i při prvním zobrazení stránky uživatelem (protože tehdy se session cookie teprve odešle, ale ještě není přijata).

Vzhledem k tomu, že se do URL přidává z pohledu uživatele nesmyslný identifikátor, tak je lepší tuto direktivu vypnout, pokud můžeme vyžadovat zapnuté cookies. Pokud skriptem odesíláme něco jiného než HTML stránku zobrazovanou v prohlížeči (např. export dat nebo obrázek), je nutné tuto direktivu vypnout, protože může vést k poškození dat.

Předávání session identifikátoru v URL může také znamenat bezpečnostní riziko.

Poznámka: Na adresy posílané funkcí `header` (nejčastěji `Location`) se tato direktiva nevztahuje a session identifikátor je do nich nutné přidat ručně.

95 Odmítnutí session identifikátoru z URL

Pokud po uživatelích můžeme požadovat zapnuté cookies, je vhodné zapnout konfigurační direktivu `session.use_only_cookies`, která způsobí ignorování session identifikátoru z URL. Její zapnutí ztíží útok *Session Fixation*, kdy útočník vnucuje uživateli session identifikátor nejčastěji právě pomocí URL.

96 Konstanta se session identifikátorem

Při zahájení session (obvykle zavoláním funkce `session_start`) vytvoří PHP konstantu `SID`. Pokud od uživatele přišla cookie se session identifikátorem, tak je tato konstanta nastavena na prázdný řetězec, jinak obsahuje název a hodnotu session identifikátoru.

Konstantu tedy můžeme použít např. při přesměrování, kde se session identifikátor automaticky nedoplňuje.

```
<?php
session_start();
$sid = (SID ? "?" . SID : "");
header("Location: http://$_SERVER[SERVER_NAME]/$sid");
?>
```

Poznámka: Bez použití této konstanty se můžeme obejít, pokud po uživatelích vyžadujeme zapnuté cookies.

97 Vytváření session identifikátoru

PHP vytváří session identifikátory z pseudonáhodné posloupnosti. Aby se vývoj této posloupnosti nedal snadno odhadnout, je vhodné nastavit konfigurační direktivu `session.entropy_file`, která do posloupnosti přidává zdroj náhody z určeného souboru. Na Linuxu lze použít `/dev/random` nebo `/dev/urandom`. Spolu s touto direktivou je potřeba také nastavit `session.entropy_length`.

Poznámka: Vhodné je také nezveřejňovat výstup funkce `uniqId` s druhým parametrem, která používá stejnou pseudonáhodnou posloupnost.

98 Oddělovač parametrů URL

Konfigurační direktivou `arg_separator.input` lze nastavit oddělovače parametrů v adrese. Některé servery používají jako oddělovač středník, protože výchozí znak `&` má zvláštní význam i v HTML, takže se např. odkaz s více parametry musí zapisovat jako `<a href="?a=1&amp;b=5">`. Nicméně vzhledem k tomu, že při odesílání formulářů prohlížeče vždy generují `&`, je toto nastavení spíše na škodu, protože vznikají duplicity.

Direktiva `arg_separator.output` se používá při generování odkazů, např. při zapnuté volbě `session.use_trans_sid` nebo ve funkci `http_build_query`. Výchozí hodnota této konfigurační direktivy je `&`, což je problematické kvůli tomu, že se používá při výpisu do HTML kódu. Můžeme ji proto změnit na `&`, pak ale výstup funkce `http_build_query` bez třetího parametru (který dovoluje oddělovač explicitně určit) můžeme použít jen v HTML kódu bez dalšího ošetření.

Poznámka: Jako oddělovač v direktivě `arg_separator.input` lze nastavit i více znaků, to ale není moc vhodné, protože pak mají dvě různé adresy (např. `?a=1&b=5` a `?a=1;b=5`) stejný význam.

99 Nastavení limitů PHP

PHP obsahuje v rámci ochrany před zahlcením serveru několik paměťových a časových limitů. Časové limity jsou dva:

- `max_execution_time` – maximální čistá doba běhu skriptu je ve výchozím nastavení 30 s, u dlouhotrvajících skriptů je vhodné ji zvýšit nebo úplně vypnout. Lze změnit také funkcí `set_time_limit`.
- `max_input_time` – maximální čas pro zpracování vstupních dat, ve výchozím nastavení je vypnut a obvykle ho není potřeba zapínat.

Paměťové limity jsou tři:

- `memory_limit` – celkové množství paměti, kterou může skript zabrat. Od PHP 5.2.1 je tato direktiva nastavena na přijatelných 128 MB.
- `post_max_size` – maximální velikost dat přenášených metodou POST. Výchozí hodnotu 8 MB může být potřeba zvětšit pro nahrání většího množství souborů najednou.
- `upload_max_filesize` – maximální velikost nahrávaného souboru. Výchozí hodnota 2 MB může být pro zpracování větších souborů nedostatečná.

100 Nastavení omezení paměti

Někdy dopředu nevíme, kolik paměti bude skript potřebovat, můžeme to ale zjistit při běhu skriptu. Tehdy se dá využít toho, že direktiva `memory_limit` se dá nastavit i při běhu skriptu:

```
<?php
// momentální zabraná paměť + požadovaná velikost + rezerva
ini_set("memory_limit", memory_get_usage() + $size + 8000000);
?>
```

101 Co je to bezpečný režim

Direktiva `safe_mode` je určena pro zlepšení zabezpečení serveru, kde provozuje skripty více uživatelů. Direktiva má tyto základní funkce:

1. Souborovým funkcím dovolí pracovat jen s těmi soubory, které mají stejného vlastníka jako spuštěný skript nebo leží v adresáři s tímto vlastníkem.
2. Pomocí direktivy `safe_mode_exec_dir` dovoluje omezit programy spouštěné z příkazové řádky.
3. Nedovoluje změnit maximální dobu běhu skriptu a množství maximálně zabrané paměti.
4. Do parametru `realm` HTTP autentizace přidává ID vlastníka skriptu.

První bod je nejvýznamnější a také nejproblematičtější. Pokud si totiž soubory vytvářené skriptem ukládáme do adresářů vytvořených skriptem (např. strukturovaná cache), tak k nim kvůli `safe_mode` nebudeme mít přístup. Naopak `safe_mode` dovoluje skriptům vytvořeným pomocí PHP pracovat se soubory vytvořenými skripty jiných uživatelů.

Poznámka: Direktiva `safe_mode` byla v PHP 5.3 označena jako zastaralá a z budoucích verzí bude odstraněna.

102 Omezení adresáře skriptů

Direktivou `open_basedir` od sebe můžeme oddělit uživatele sdíleného webhostingu. Tato direktiva nám dovoluje omezit adresář, do kterého bude mít spuštěný skript přístup. Pokud tuto direktivu nastavíme odlišně např. pro každou doménu, tak skripty běžící na této doméně nebudou moci pracovat s cizími soubory.

Poznámka: Adresář může být nastaven i relativně (např. jako `.`), to je ale nevhodné, protože funkcí `chDir` se dá pracovní adresář snadno změnit.

103 Umístění dočasných souborů

Na sdíleném hostingu bychom měli každému uživateli nastavit vlastní adresář pro nahrávané soubory a ukládaní session proměnných. Řídí to direktivy `upload_tmp_dir` a `session.save_path` a ideálně by měly mířit do adresáře nedostupného z webu, ale v rámci `open_basedir`.

104 Jak skrýt přítomnost PHP

Vypnutím direktivy `expose_php` můžeme skrýt informaci o tom, že na serveru běží PHP. Tato informace spolu s verzí PHP se jinak předává v hlavičce `Server` nebo `X-Powered-By`. Informaci o verzi PHP mohou využít útočníci pro zjištění bezpečnostních slabin serveru, pravda je ale taková, že útok proti starším verzím PHP můžou vyzkoušet i bez znalosti verze. Lepší je proto používat aktuální verzi PHP a informaci o přítomnosti PHP neskrývat.

Poznámka: Vypnutí této direktivy odstraní také obrázek z výsledku funkce `phpInfo`, protože podle něj by šla přítomnost PHP také zjistit.

105 Vložení souboru do všech skriptů

Pomocí direktiv `auto_prepend_file` a `auto_append_file` můžeme automaticky vložit soubor na začátek resp. konec každého skriptu. Můžeme to použít v situaci, kdy všechny skripty využívají společnou knihovnu, kterou pak nemusíme ručně vkládat do všech skriptů.

Obvykle se nicméně tyto konfigurační direktivy nepoužívají, protože je lepší navrhovat aplikace tak, aby běžely pokud možno nezávisle na konfiguraci. Použít je nicméně můžeme tehdy, když chceme aplikaci třetí strany předřadit vlastní kód (např. ověření oprávnění ke spuštění aplikace). Díky těmto direktivám totiž nemusíme měnit zdrojový kód aplikace, což usnadňuje její aktualizaci.

106 Konfigurace e-mailu

Pro odesílání e-mailů pod Windows používá PHP přímo protokol SMTP. Adresu serveru pro odesílání zpráv je proto potřeba nastavit konfigurační direktivou `SMTP` (na rozdíl od ostatních konfiguračních direktiv se píše velkými písmeny).

V ostatních operačních systémech se používá pro odesílání zpráv program příkazového řádku, který je možno nastavit konfigurační direktivou `sendmail_path`.

107 Jak nastavit časové pásmo

Při použití funkcí `date`, `strToTime` a podobných se zohledňuje časové pásmo. To dokáže PHP detekovat z proměnné prostředí `TZ`, tento způsob je ale nespolehlivý. Od PHP 5.1 lze proto časové pásmo nastavit funkcí `date_default_timezone_set` nebo konfigurační direktivou `date.timezone` a bez jejího nastavení vyvolá PHP při použití datové funkce varování. V České republice lze tuto direktivu nastavit na hodnotu `Europe/Prague`.

Poznámka: Funkce `gmDate` pracuje vždy s centrálním časem, a nastavení časového pásma tedy nepotřebuje.

108 Konfigurace MySQL

Pokud se ze skriptů připojujeme k databázi MySQL, můžeme přihlašovací údaje definovat v konfiguračním souboru. Slouží k tomu direktivy `mysql.default_host`, `mysql.default_user` a `mysql.default_password` při použití extenze **MySQL**, resp. `mysqli.default_host`, `mysqli.default_user` a `mysqli.default_pw` při použití **MySQLi**. Takovéto direktivy nabízí i některé další databázové extenze.

Výhoda spočívá v tom, že ve skriptech potom přihlašovací údaje už nemusíme uvádět:

```
<?php
mysql_connect();
mysql_select_db("test");
?>
```

Poznámka: Takto zadané přihlašovací údaje se zobrazují např. ve výstupu funkce `phpInfo`.

109 Co je to MySQLnd

PHP s databázovým serverem MySQL komunikuje tradičně pomocí knihovny **libmysql**, ale od verze PHP 5.3 dovoluje využít také knihovnu **MySQLnd** (*native driver*). Knihovna **libmysql** je obecná a kromě PHP ji využívá také řada dalších projektů, což je její hlavní nevýhoda. Data si tato knihovna totiž ukládá do vlastních struktur, z kterých je PHP musí převádět, což způsobuje dvojnásobné paměťové nároky a stojí čas vynaložený na konverzi.

Knihovna **MySQLnd** je napsaná na míru PHP a data ukládá přímo v interním formátu, což má pozitivní vliv na výkon. Kromě toho také nabízí funkce pro získání statistik jako např. `mysqli_get_client_stats`.

MySQLnd se zapíná při kompilaci PHP, od PHP 5.3 je ve výchozím nastavení zapnuté.

110 Zastaralé direktivy

Některé konfigurační direktivy PHP byly označené jako zastaralé, nebo už byly z PHP dokonce odstraněny. Tyto konfigurační direktivy je tedy lepší vůbec nepoužívat. Jedná se zejména o následující direktivy:

- `zend.ze1_compatibility_mode`
- `magic_quotes_gpc`
- `magic_quotes_runtime`
- `magic_quotes_sybase`
- `register_globals`
- `register_long_arrays`
- `safe_mode`
- `allow_call_time_pass_reference`
- `enable_dl`

111 Jaký použít oddělovač adresářů

PHP dovoluje jako oddělovač adresářů a souborů použít na všech platformách lomítko i zpětné lomítko. Obvykle se používá běžné lomítko. Existuje také konstanta `DIRECTORY_SEPARATOR` obsahující nativní oddělovač, tu bychom mohli použít např. při volání externích programů (např. u příkazu `include` je ale použití této konstanty zbytečné).

112 Oddělovač cest

Funkce `set_include_path` dovoluje nastavit více cest, ve kterých se budou hledat vkládané soubory. Cesty se na Linuxu oddělují dvojtečkou, na Windows středníkem. Správný oddělovač lze získat z konstanty `PATH_SEPARATOR`.

```
<?php
$include = dirName(__FILE__) . "/include";
set_include_path($include . PATH_SEPARATOR . get_include_path());
?>
```

113 Vypsání XML značky

Konfigurační direktivou `short_open_tag` se dá povolit zahajování PHP kódu značkou `<?`. To ovšem koliduje se značkou `<?xml`, která se používá pro zahájení XML dokumentů. Pro vypsání této značky je tedy lepší použít PHP kód:

```
<?php
echo "<?xml version='1.0' encoding='UTF-8' ?>\n";
?>
```

Poznámka: Značku lze vypsat také trikem – `<<?php ?>?xml version='1.0' encoding= 'UTF-8' ?>`.

114 Použití konstant pro vlastní konfiguraci

Pokud má mít naše aplikace nějakou globální konfiguraci (určující např. výchozí počet vypisovaných záznamů), je obvykle vhodné ji zapsat do konstant. Ty jsou na rozdíl od proměnných neměnné a automaticky viditelné ve funkcích. Zapisují se obvykle velkými písmeny se slovy oddělenými podtržítky:

```
<?php
// dá se změnit a není přímo vidět ve funkcích
$defaultLimit = 30;

// lepší
define("DEFAULT_LIMIT", 30);
?>
```

115 Konstanty bez rozlišení velikosti písmen

Pomocí třetího parametru funkce `define` lze vytvořit konstanty, u kterých se nebude rozlišovat velikost písmen. Takové konstanty je ale obvykle lepší nevytvářet, protože vedou k psaní nejednotného kódu.

Užitečné to je pouze u výsadních konstant PHP, jako jsou `null`, `true` a `false`.

116 Definice absolutní cesty

Pokud si potřebujeme definovat absolutní cestu k aplikaci v souborovém systému, je vhodné ji odvodit z cesty k aktuálnímu souboru. Jinak totiž aplikace téměř jistě přestane fungovat při přesunu na jiný hosting, protože používané cesty nejsou ustálené.

```
<?php
// nepoužívat
define("APPLICATION_PATH", "/var/www/example.com/htdocs");

// lepší
define("APPLICATION_PATH", dirName(__FILE__));

// možné od PHP 5.3
define("APPLICATION_PATH", __DIR__);
?>
```

Poznámka: Cestu lze většinou zapsat i relativně (např. pomocí odkazu na nadřazený adresář `../`), to může ale způsobit problémy při spuštění skriptu z jiného adresáře (např. na příkazovém řádku nebo v obsluze funkce `register_shutdown_function`).

117 Definice absolutního URL


pokročilý

Pokud naše aplikace potřebuje pracovat s absolutním URL (např. při přesměrování nebo při posílání odkazů do aplikace e-mailem), je možné ho odvodit z aktuálního URL. To umožní aplikaci snadno testovat na vývojovém počítači nebo ji přesunout na jinou adresu:

```
<?php
// neumožňuje provoz jinde
define("ABSOLUTE_URL", "http://www.example.com/");

// lepší
define("ABSOLUTE_URL", "http://$_SERVER[SERVER_NAME]"
    . dirName($_SERVER["SCRIPT_NAME"])
);
?>
```

Poznámka: Zcela obecná aplikace by měla nejprve kontrolovat proměnnou `HTTP_HOST`, protože `SERVER_NAME` nemusí být vždy nastaveno. Pokud aplikace může běžet na protokolu HTTPS, mělo by to být v absolutním URL také zohledněno.

118 Kde je uložena cesta k sobě


pokročilý

Proměnná `$_SERVER["PHP_SELF"]` obsahuje cestu zobrazené stránky až do případného otazníku. Cestu ke spuštěnému skriptu obsahuje proměnná `$_SERVER["SCRIPT_NAME"]`. Tyto dvě proměnné se liší v případě přepisu URL nebo při použití tzv. *path info*.

Cestu včetně případných parametrů obsahuje proměnná `$_SERVER["REQUEST_URI"]`, ta se může hodit třeba v případě, kdy chceme nastavit cookie platnou jen pro momentálně zobrazenou stránku. Řetězec se samotnými parametry je k dispozici v proměnné `$_SERVER["QUERY_STRING"]`, tato proměnná je ale potřeba zřídka, protože z parametrů vytvoří PHP pole `$_GET`.

119 Instalace vlastní aplikace


znalec

Pokud vytváříme aplikaci určenou pro instalaci na více počítačů, můžeme vytvořit jednoduchý instalační skript, který cílovým uživatelům umožní nastavení konfigurace (např. přihlašovací údaje k databázovému serveru). Tento skript může vytvořený konfigurační soubor buď rovnou uložit, nebo nabídnout ke stažení.

Z pohledu bezpečnosti je důležité, aby se skript nedal spustit v běžící aplikaci, protože by mohl začít např. přihlašovací údaje posílat na jiný server. Hlavní aplikace proto může kontrolovat, zda je instalační skript smazán:

```
<?php
if (file_exists("config.php")) {
    echo "Před spuštěním aplikace nejrpve smažte instalační skript.";
    exit;
}
?>
```

Syntaxe jazyka

120 Možnosti uzavírání PHP kódu

Pro uzavření PHP kódu je možno v PHP použít až čtyři různé zápisy:

1. `<?php ?>`
2. `<? ?>`
3. `<% %>`
4. `<script language="php"> </script>`

Druhý způsob se dá zakázat konfigurační direktivou `short_open_tag`, třetí pomocí `asp_tags`, čtvrtý zase vypadá spíše jako klientský skript. Nejlepší je tedy vždy používat první způsob zápisu.

121 Vynechání koncové značky

Pokud obsah souboru končí PHP kódem, není nutné kód ukončovat značkou `?>`. Kromě ušetřeného psaní to má tu výhodu, že pokud by soubor měl na konci nějaké zapomenuté bílé znaky (mezery, tabulátory, konce řádků), tak se při uvedení koncové značky vypíšou. To může vadit například při odesílání HTTP hlaviček, které je potřeba poslat dříve než jakýkoliv textový výstup.

122 Konec řádku za koncovou značkou

Znak konce řádku za koncovou značkou `?>` se nevypisuje. To obvykle vede k lepší přehlednosti výsledného HTML kódu, ale je na to potřeba dát pozor v případě, kdy další řádek začíná textem. V tom případě se tento text připojí k předchozímu řádku. Řešit se to dá vložením dvou konců řádek nebo mezery.

123 Ukončení příkazu

Příkaz se v PHP ukončuje středníkem, před značkou `?>` je ale nepovinný. Vynechání středníku je nicméně vhodné se až na výjimky vyhnout, protože znesnadňuje budoucí přidání příkazu na konec bloku PHP kódu.

124 Ignorování zbytku souboru

Od PHP 5.1 je možné z pohledu parseru ignorovat zbytek souboru od určitého místa. Toto místo se definuje zavoláním funkce `__halt_compiler`. Tento obrat se dá využít v případě, kdy chceme do souboru zahrnout i data, se kterými skript pracuje, a vytvořit tak např. samorozbalitelný archiv. Bajtovou pozici zakončení zpracování PHP získáme konstantou `__COMPILER_HALT_OFFSET__`.

125 Ukončení příkazu pro ignorování zbytku souboru

U funkce `__halt_compiler` se dá využít toho, že příkaz lze v PHP ukončit značkou `?>`. Za touto značkou navíc může být znak konce řádku, který se ignoruje.

Nepřehledný způsob:

```
<?php
__halt_compiler();textová data
```

Přehledný způsob:

```
<?php
__halt_compiler() // středník chybí záměrně
?>
textová data
```

Textové editory totiž funkci `__halt_compiler` obvykle neznají a při ukončení příkazu středníkem by zbytek souboru zvýrazňovaly také jako PHP kód.

126 Druhy komentářů

PHP dovoluje komentáře zapsat třemi možnými způsoby:

```
<?php
// jednořádkový komentář

# jednořádkový komentář

/*
víceřádkový komentář
*/
?>
```

Jednořádkové komentáře jsou vhodné k popisu částí kódu, víceřádkové komentáře se hodí především pro dokumentační komentáře. Dají se použít také pro dočasné skrytí bloku kódu, nelze je ale vnořovat do sebe. Pro dočasné skrytí kódu můžeme také každý řádek uvodit komentářem #, který jinde nebudeme používat. Některé programátorské editory mají klávesovou zkratku, která dovoluje tento komentář přidat nebo odebrat ze všech označených řádků.

127 Ukončení jednořádkového komentáře

Jednořádkový komentář se ukončuje buď koncem řádku, nebo značkou `?>`. To se hodí obzvlášť u jednořádkových bloků PHP kódu: `<?php echo $count; // vypsání počtu ?>`.

128 Dokumentační komentáře

Speciálním druhem komentářů jsou dokumentační komentáře. Ty se uvozují znaky `/**`, a jak už název napovídá, mohou kromě popisu kódu sloužit i k vygenerování dokumentace. K tomu se nejčastěji používá program **phpDocumentor**. Kromě toho dokážou s dokumentačními komentáři pracovat některá pokročilá vývojová prostředí (např. **Eclipse**) a používat je pro nápovědu u funkcí, metod, tříd a dalších prvků kódu.

Dokumentační komentář se píše před dokumentovaný kód a každý řádek bloku musí začínat hvězdičkou. Kromě běžného textu lze v komentáři pomocí znaku `@` uvádět značky, které mají zvláštní význam.

129 Značky dokumentačních komentářů

Řada značek dokumentačních komentářů je již zastaralá. Např. v PHP 5 nemá smysl používat značku `@abstract`, protože abstraktní metody a třídy lze určit přímo pomocí klíčového slova `abstract`. Stejně tak u vlastností a metod není potřeba používat značku `@access` ovlivňující viditelnost, ta ale může mít využití u globálních funkcí. V praxi se tak používají především tyto značky:

```
<?php
/**
* @author    jméno <email>
* @copyright jméno datum
* @license   URL název (licence)
* @link      URL
* @package   název (balíčku)
* @param     typ popis (parametru funkce nebo metody)
* @return    typ popis (návratové hodnoty)
* @throws    třída (vyvolávané výjimky)
* @var       typ popis (proměnné nebo vlastnosti)
*/
?>
```

Značky se v dokumentačním komentáři mohou opakovat, platí to třeba i pro informace o vyvolávaných výjimkách.

130 Jak na „Nutno dodělat“

Někdy si v kódu potřebujeme poznačit informaci o tom, že je něco potřeba ještě dodělat. Můžeme k tomu použít buď dokumentační komentář `@todo`, nebo obyčejný komentář začínající textem `TODO`. Některá pokročilá vývojová prostředí z těchto komentářů dokážou vytvořit seznam úkolů.

```
<?php
mysql_query($query); // TODO: předat identifikátor připojení
?>
```

131 Zpracování dokumentačních komentářů

Kromě existujících aplikací pro zpracování dokumentačních komentářů je můžeme zpracovat i sami. Od PHP 5 je vrací funkce `token_get_all`, ale především jsou k dispozici při reflexi. PHP se nestará o vnitřní zpracování dokumentačního komentáře (především získání jednotlivých značek), o to už se pomocí řetězcových operací musíme postarat sami.

132 Jak na anotace

Dokumentační komentáře se dají použít i k ovlivnění chování kódu. Pokud si do dokumentačního komentáře poznamenáme nějakou speciální značku, může ji okolní kód brát v úvahu. Dobrým příkladem využití může být např. označení databázového typu v knihovně ORM. Tento způsob ovlivnění nějaké funkčnosti může být při používání velmi pohodlný, ne každému ale vyhovuje – někomu přijde změna chování programu pomocí anotací až příliš magická.

Pro zpracování anotací můžeme použít reflexi, je to ale poměrně pracné. Raději proto nejspíš sáhneme po hotové knihovně, vhodná je např. **Nette Annotations**.

133 Co je operátor identity

Většina porovnávacích operátorů si nevšímá datového typu a pracuje jen s hodnotou operandů. Pokud se datové typy liší, jsou podle definovaných pravidel sjednoceny. To může být na škodu například u funkce `strPos`, která vrací v případě neúspěchu hodnotu `false` a v případě nalezení řetězce index jeho pozice počínaje nulou.

```
<?php
if (strPos($string, "a") === false) {
    echo "Řetězec nebyl nalezen.\n";
}
?>
```

Operátor identity je vhodné používat ve většině případů místo prostého porovnání, protože to pomůže najít hůře odhalitelné chyby.

134 Jaký použít operátor nerovnosti

V PHP jsou dva operátory pro zjištění nerovnosti dvou výrazů: `!=` a `<>`. Protože PHP vychází především z jazyka C, kde se používá operátor `!=`, je zvykem nerovnost zkoumat právě tímto operátorem i v PHP, druhý operátor má ale zcela totožnou funkci.

Poznámka: Operátor `!==` uspěje při rozdílnosti typu nebo hodnoty.

135 Pořadí porovnávání

Operátor = slouží v PHP k přiřazení hodnoty, a nikoliv k porovnání jako v některých jiných programovacích jazycích. Přiřazení je ale také výraz, který se může vyhodnotit např. v podmínce, takže zápis `if ($a = 5)` je korektní, i když neudělá to, co autor nejspíš očekával (přiřadí do proměnné `$a` hodnotu 5 a tuto hodnotu vyhodnotí jako pravdu).

Pokud se někomu operátory pletou třeba v důsledku přechodu z jiného programovacího jazyka, je možné proměnnou uvádět až na druhém místě porovnání, protože zápis `if (5 = $a)` způsobí syntaktickou chybu.

Obecně vzato je ale přehlednější uvádět nejprve výraz, který porovnáváme, a teprve potom hodnotu, se kterou porovnáváme.

136 Co je to ternární operátor

Operátor `? :` se jmenuje ternární podle toho, že pracuje se třemi operandy. Pokud by takových operátorů bylo více, tak bychom mu mohli říkat podmíněný výraz. Operátor vyhodnotí výraz před otazníkem, a pokud je splněn, vrátí výraz za otazníkem, jinak výraz za dvojtečkou. Je tedy jakousi výrazovou obdobou podmíněného příkazu `if else`.

Operátor je vhodné využívat pouze v případě, kdy jsou všechny tři výrazy dostatečně krátké, jinak je nepřehledný.

137 Zkrácený ternární operátor

Od PHP 5.3 je možno v ternárním operátoru vynechat prostřední část. Místo ní se potom použije testovaný výraz:

```
<?php
echo ($a ?: 0); // totéž jako ($a ? $a : 0)
// pokud proměnná $a není nastavena, dojde k chybě E_NOTICE
?>
```

Hodí se to hlavně v případě, kdy je testovaný výraz složitý a obsahuje např. volání funkce.

Poznámka: Pojmenování ternární operátor je odvozeno z počtu tří operandů, se kterými tento operátor pracuje. Při vynechání prostřední části jde vlastně o binární operátor a přesnější by bylo nazývat ho zkrácený podmíněný výraz.

138 Jak používat operátor plus

Operátor plus se v PHP používá ke sčítání čísel, pro spojení řetězců se používá operátor tečka. Pokud se pokusíme operátor plus použít na dva řetězce, tak se převedou na číslo a sečtou.

Jinak se operátor plus chová s poli, kde provede jejich sjednocení.

139 Pořadí násobení a dělení


pokročilý

Násobení a dělení celých čísel je vhodné dělat v takovém pořadí, aby mezivýsledek zůstal celým číslem v přípustném rozsahu. Jednak jsou celočíselné operace mírně rychlejší, jednak v tom případě nemůže dojít k zaokrouhlovacím chybám.

```
<?php
var_dump(1 / 2 * 2); // float(1)
var_dump(2 * 1 / 2); // int(1)
?>
```

140 Bitový posun


znalec

V PHP jsou mírně netradičně zpracovány operátory bitového posunu. Operátor << posouvá bity doleva a zprava doplňuje nuly. V nejvýznamnějším bitu se uchovává informace o znaménku, takže tento operátor může znaménko změnit.

Operátor >> posouvá bity doprava, ale informaci o znaménku zachovává. Zleva se tedy doplňuje u záporných čísel jednička a u kladných nula.

```
<?php
var_dump(PHP_INT_MAX << 1); // vznikne záporné číslo
var_dump(-PHP_INT_MAX >> 1); // číslo zůstane záporné
?>
```

141 Obvyklý bitový posun


znalec

Při bitovém posunu doprava je zvykem zleva doplňovat nuly. Takový operátor v PHP bohužel k dispozici není, můžeme si ale napsat vlastní funkci, která se o to postará:

```
<?php
/** Bitový posun doprava s doplňováním nul
* @param int
* @param int
* @return int
*/
function rightShift($number, $shift) {
    return ($number >> $shift) & (PHP_INT_MAX >> ($shift - 1));
}
?>
```

142 Bitová negace


začátečník

Operátor ~ se používá pro obrácení hodnoty bitů v čísle. V PHP se nejčastěji používá spolu s konstantami.

```
<?php
// nastaví úroveň chyb na všechny kromě poznámek
error_reporting(E_ALL & ~E_NOTICE);
?>
```

Poznámka: Pro zapnutí nebo vypnutí určitého druhu chyb se někdy používá zápis `E_ALL ^ E_NOTICE` (operátor bitový XOR). Tento zápis se ale spoléhá na to, že konstanta `E_ALL` danou chybu obsahuje nebo naopak neobsahuje – její hodnota se ale v průběhu verzí měnila, takže je lepší se tomuto zápisu vyhnout.

143 Definice konstant

Konstanty slouží pro přiřazení symbolického názvu nějaké neměnné hodnotě. Na rozdíl od proměnných nelze jejich hodnotu změnit a nejde je ani zrušit.

```
<?php
define("NOW", time()); // vytvoření konstanty
const ERROR = 'error'; // alternativní vytvoření od PHP 5.3
?>
```

144 Použití konstant

Pokud chceme přistoupit k definované konstantě, stačí uvést její název.

```
<?php
if (defined("NOW")) { // test existence
    echo NOW; // použití
}
?>
```

Kromě toho lze použít také funkci `constant`, což se hodí v případě, kdy název konstanty dynamicky sestavujeme.

```
<?php
$input = '_POST';
constant("FILTER$input"); // dynamicky sestavený název konstanty
?>
```

145 Nedefinované konstanty

Pokud se v PHP pokusíme přistoupit k nedefinované konstantě, tak se vyvolá chyba úrovně `E_NOTICE` a vrátí se místo ní řetězec s identifikátorem.

Toto chování bychom neměli využívat jednak kvůli vyvolané chybě, a jednak kvůli tomu, že při dodatečné definici konstanty přestane kód fungovat.

146 Konstanty obsahující pole

V PHP lze definovat pouze konstanty obsahující skalární hodnoty a `null`, nelze tedy vytvořit konstantu obsahující pole nebo objekt. Tento nedostatek se obchází různými způsoby:

```
<?php
// nelze
define("DATABASE", array(
    "server" => "localhost",
    "login" => "ODBC",
    "password" => "",
));

// musíme volat funkci unserialize při každém použití konstanty
define("DATABASE", serialize(array(
    "server" => "localhost",
    "login" => "ODBC",
    "password" => "",
)));

// definice globální proměnné místo konstanty
$DATABASE = array(
    "server" => "localhost",
    "login" => "ODBC",
    "password" => "",
);
// použití: $GLOBALS["DATABASE"]["login"]

// definice více konstant
define("DATABASE_SERVER", "localhost");
define("DATABASE_LOGIN", "ODBC");
define("DATABASE_PASSWORD", "");

// použití třídních konstant
class Database {
    const SERVER = "localhost";
    const LOGIN = "ODBC";
    const PASSWORD = "";
}
// použití: Database::LOGIN
?>
```

Každý přístup má své výhody a nevýhody, obvykle je ale nejjednodušší prostě definovat více konstant.

147 Jak uložit návratovou hodnotu funkce

Příkaz `return` použitý uvnitř funkce způsobí její ukončení a předání návratové hodnoty. Někdy si ale potřebujeme návratovou hodnotu jen zapamatovat a funkci nechat pokračovat (např. proto, aby po sobě mohla uklidit). Pro volbu názvu proměnné obsahující návratovou hodnotu lze využít toho, že názvy proměnných nekolidují s jinými identifikátory PHP, a lze ji pojmenovat prostě *`$return`*.

148 Interní funkce s proměnným počtem parametrů

Řada PHP funkcí přijímá proměnný počet parametrů, nepovinné parametry ale nemusí mít definovanou výchozí hodnotu. Funkce se totiž interně rozhodují podle počtu předaných parametrů.

```
<?php
// první funkce vyvolá chybu, druhá použije poslední připojení
mysql_query($query, null);
mysql_query($query);
?>
```

Implementaci těchto funkcí si můžeme představit takto:

```
<?php
function mysqlQuery($query, $connection = null) {
    if (func_num_args() < 2) {
        // získání posledního připojení do proměnné $connection
    }
}
?>
```

Znalost tohoto chování je důležitá při vytváření vyšších vrstev nad těmito funkcemi.

149 Předefinování funkce

PHP samo o sobě nenabízí možnost předefinování funkce. Jakmile je funkce jednou definovaná, tak pokus o definici funkce se stejným názvem skončí chybou. Tuto schopnost nabízí pouze extenze **APD** a **Runkit** v podobě funkcí `override_function`, resp. `runkit_function_redefine`. Kromě experimentů je ale lepší se této vlastnosti vyhnout a funkce definovat vždy jen jednou.

Poznámka: Přejmenování umožňují funkce `rename_function`, resp. `runkit_function_rename`.

150 Definice výjimky

Pro výjimky musíme použít objekt třídy `Exception`, kterému můžeme určit chybovou hlášku a číselný kód chyby. Při zachytávání výjimek v bloku `catch` se ale nerozhoduje podle kódu chyby, ale podle třídy výjimky. Pro vlastní výjimky je tedy zcela běžné definovat vlastní třídu. Ta musí být potomkem třídy `Exception`, ale nemusí ji nijak funkčně rozšiřovat.

```
<?php
class DbException extends Exception {
}
?>
```

151 Rozšíření výjimky

Při definici třídy výjimky si často vystačíme s metodami nabízenými třídou `Exception`. Pokud ale třeba výjimce chceme přiřadit nečíselný chybový kód vrácený nižší vrstvou, nic nám v tom nebrání. Praktické je v tom případě přepsat hlavně konstruktor.

```
<?php
class ServiceException extends Exception {
```

```
    /** Konstruktor
    * @param string
    * @param string
    */
    function __construct($message, $code) {
        parent::__construct($message, 0);
        $this->code = $code;
    }
}

try {
    throw new ServiceException("Zpráva", "kód");
} catch (Exception $e) {
    echo $e->getCode() . "\n";
}
?>
```

152 Probublání výjimky


pokročilý

Pokud blokem `catch` zachytíme výjimku a teprve tehdy zjistíme, že ji nedokážeme zpracovat (např. podle chybového kódu), můžeme ji poslat ke zpracování nadřazenému bloku. Výjimku není třeba znovu vytvářet, ale stačí vyvolat tu zachycenou:

```
<?php
try {
    throw new Exception("Probublání", 1);
} catch (Exception $e) {
    if ($e->getCode() == 1) { // takovou výjimku neumíme zpracovat
        throw $e;
    }
    // tady bude běžná obsluha chyby
}
?>
```

153 Eskalace výjimky


znalec

Chyby způsobené nějakou námi vyvíjenou knihovnou můžeme chtít reprezentovat jednotnou výjimkou. Někdy se ale může hodit vědět i to, jaká výjimka chybu původně způsobila. Od PHP 5.3 to lze vyřešit předáním třetího parametru konstruktoru výjimky, pomocí kterého lze předat předchozí výjimku:

```
<?php
class LibraryException extends Exception {
}

try {
    throw new Exception("Vyvolání obecné chyby");
} catch (Exception $e) {
    throw new LibraryException("Při komunikaci došlo k chybě", 0, $e);
}
?>
```

154 Překlad chyb na výjimky


pokročilý

Jazyku PHP se někdy vyčítá, že sice obsahuje mechanizmus výjimek, sám ho ale až na několik málo případů nepoužívá a místo toho využívá vlastní mechanizmus chyb. Nicméně je velmi snadné překládat na výjimky všechny chyby, ke kterým dojde v interních funkcích PHP. Stačí použít funkci `set_error_handler` pro definici vlastní obsluhy chyb a v ní chybu na výjimky přeložit. Od PHP 5.1 pro tento druh výjimky dokonce existuje vlastní třída.

```
<?php
/** Funkce pro set_error_handler překládající chyby na výjimky
* @param int
* @param string
* @param string
* @param int
* @throws ErrorException
*/
function exceptionErrorHandler($level, $message, $file, $line) {
    if (error_reporting() & $level) {
        throw new ErrorException($message, 0, $level, $file, $line);
    }
}
set_error_handler("exceptionErrorHandler");
strPos("abc", "a", 4); // vyvolá výjimku
?>
```


Poznámka: Před PHP 5.3 obsahuje takto vyvolaná výjimka nesprávný *stack trace*.

155 Typy výjimek v PHP


pokročilý

Kromě třídy `Exception`, která je předkem všech výjimek, a třídy `ErrorException`, která je určena k překladu chyb na výjimky, definuje další standardní výjimky extenze **SPL**. Užitečné mohou být především následující výjimky:

- `LogicException` a její potomci `InvalidArgumentException` a `LengthException`.
- `RuntimeException` a její potomci `OutOfBoundsException` a `UnexpectedValueException`.

156 Zobrazení typu výjimky


pokročilý

Pokud v bloku `catch` chceme vypsat typ výjimky, tak můžeme využít toho, že výjimka je v PHP vždy tvořena objektem, takže můžeme získat její třídu pomocí funkce `get_class`.

```
<?php
try {
    throw new UnexpectedValueException("Nečekaná hodnota");
} catch (Exception $e) {
    echo get_class($e) . ": " . $e->getMessage() . "\n";
}
?>
```

Poznámka: Takovouto obsluhu výjimek je vhodné používat třeba při hlášení chyb v PHP. Pro běžné otestování třídy výjimky lze použít operátor `instanceof`, kromě toho lze objekt výjimky také přímo vypsat.

157 Obsluha nezachycené výjimky

Pokud dojde k výjimce, kterou program nezachytí, skončí fatální chybou. Toto chování lze upravit zavoláním funkce `set_exception_handler` definující *callback*, který se v takovém případě zavolá. Funkci můžeme použít třeba pro odeslání informace o výjimce e-mailem.

```
<?php
function handleException($exception) {
    mail("admin@example.com", "Vyjimka", $exception->getMessage());
}
set_exception_handler('handleException');
throw new Exception("Nezachycená výjimka");
?>
```

Poznámka: Skončením funkce je ukončen i celý program.

158 Skrytí chyb

V PHP je k dispozici operátor `@` (zavináč), který skryje všechny chyby vyvolané následujícím příkazem. Použití tohoto operátoru bychom se ale měli v naprosté většině případů vyhnout. Pokud např. funkce vyvolá chybu v případě, že jí předáme nesprávné parametry, je mnohem vhodnější tyto parametry zkontrolovat před zavoláním samotné funkce. Operátor `@` totiž může skrýt i chyby, se kterými nepočítáme a které se potom špatně odhalují (včetně fatálních). Použití tohoto operátoru také aplikaci zpomaluje.

Operátor bychom neměli používat ani v případě, kdy vznik chyby nedokážeme ovlivnit (např. funkce pro připojení k databázi vyvolá chybu v případě, kdy se k databázovému serveru nedá připojit). O takovéto chybě je vhodné samozřejmě uživatele informovat vlastní zprávou:

```
<?php
if (!mysql_connect()) {
    echo "Nepodařilo se připojit k databázovému serveru.\n";
}
?>
```

Skrytí systémové chyby na produkčním serveru zajistíme vypnutím konfigurační direktivy `display_errors`. Logování těchto chyb naopak povolíme direktivou `log_errors`. V logu se pak dozvíme, kdy připojení k databázi nefungovalo.

Poznámka: Vypnutí všech chyb funkcí `error_reporting(0)` má stejné nevýhody jako použití operátoru `@`.

159 Zobrazení všech chyb

Pokud je nějaká knihovna napsaná s hojným využitím operátoru `@` a my v ní potřebujeme dohledat chyby, můžeme zapnout extenzi **scream**, která operátor pro skrytí chyby potlačí.

160 Úklid po chybě

Pokud dojde k nějaké chybě, je vhodné po sobě uklidit. Od PHP 5.3 se k tomu dá využít operátor `goto`:

```
<?php
function fillFileGoto($filename) {
    $fp = fOpen($filename, "w");
    // první část kódu
    if ($error1) {
        goto clean;
    }
    // druhá část kódu
    if ($error2) {
        goto clean;
    }
    return $fp;
    clean:
    fClose($fp);
    unlink($filename);
    return false;
}
?>
```

Ve starších verzích lze použít cyklus `do` s jedním průchodem, kód je ale o něco málo nepřehlednější:

```
<?php
function fillFileBreak($filename) {
    $fp = fOpen($filename, "w");
    do {
        // první část kódu
        if ($chyba1) {
            break;
        }
        // druhá část kódu
        if ($chyba2) {
            break;
        }
        return $fp;
    } while (false);
    fClose($fp);
    unlink($filename);
    return false;
}
?>
```

Nejčistší je použití výjimek, s těmi je ale spojena jistá časová režie, která může být u časově kritických operací na škodu:

```
<?php
class FillFileException extends Exception {
}

function fillFileThrow($filename) {
    $fp = fOpen($filename, "w");
    try {
        // první část kódu
        if ($chyba1) {
            throw new FillFileException;
        }
        // druhá část kódu
        if ($chyba2) {
            throw new FillFileException;
        }
        return $fp;
    } catch (FillFileException $e) {
        fClose($fp);
        unlink($filename);
        return false;
    }
}
?>
```

Jazykové konstrukce

161 Univerzálnost cyklu for

Je dobré si uvědomit, že příkaz `for` má univerzální syntaxi a dá se použít i pro složitější věci než jen pro předem známý počet průchodů. Jednotlivé výrazy znamenají `(inicializace; podmínka ukončení; inkrementace)`. Inicializace se provádí před prvním průchodem, podmínka se vyhodnocuje před každým průchodem a inkrementuje se na konci každého průchodu.

```
<?php
// delší řešení
$queue->first();
while (!$queue->last()) {
    // zde bude kód cyklu
    $queue->next();
}

// využití for cyklu
for ($queue->first(); !$queue->last(); $queue->next()) {
    // zde bude kód cyklu
}
?>
```

162 Došli jsme na konec cyklu?

Pokud při iteraci předčasně vyskočíme za určité podmínky z cyklu, můžeme chtít následně zjistit, zda se cyklus dokončil. To se dá poznat podle hodnoty proměnné, přes kterou iterujeme:

```
<?php
for ($i=0; $i < $count; $i++) {
    if (!processData($i)) {
        break;
    }
}
if ($i == $count) {
    // došli jsme na konec cyklu
}
?>
```

163 Počet průchodů cyklem

Pokud si u podmíněného cyklu `while` potřebujeme počítat počet průchodů, můžeme využít obecnost cyklu `for`:

```
<?php
for ($count = 1; $row = $result->fetchArray(); $count++) {
    // v proměnné $count je uložen počet průchodů počítaný od 1
}
?>
```

164 Jak použít operátor čárka


pokročilý

Především ve `for` cyklu se používá operátor čárka, který jednoduše vyhodnotí více výrazů:

```
<?php
for ($i=0, $x->first(); $i < 10 && $x->valid(); $i++, $x->next()) {
    // prochází zároveň přes proměnné $i a $x
}
?>
```

165 Alternativní syntaxe řídicích konstrukcí


pokročilý

Většina jazykových konstrukcí PHP umožňuje zápis v alternativní syntaxi:

```
<?php
if ($condition):
    // kód
endif;
?>
```

Tato syntaxe může být přehlednější především v situaci, kdy se hodně kombinuje PHP a HTML kód. Sám ji ale nepoužívám, protože programátorské editory umožňují snadno přeskakovat mezi otevírací a uzavírací závorkou, takže klasická syntaxe dovoluje lepší orientaci v kódu.

166 Řetězení podmíněného příkazu


začátečník

Kvůli alternativní syntaxi je v PHP povoleno psaní častého obratu `else if` také jako `elseif`. Důvodem je, aby po takto zřetězených příkazech nebylo potřeba uvádět několikrát `endif`. Tento obrat lze ale použít i v obvyklé syntaxi:

```
<?php
if ($a == 2) {
} elseif ($b == 3) {
}
?>
```

167 Vyvolání chyby v podmíněném příkazu


začátečník

Pokud má podmíněný příkaz vyvolat chybu (např. vrácením hodnoty `false` nebo vyvoláním výjimky), je vhodné podmínku zkonstruovat tak, aby nebylo vůbec potřeba uvádět blok `else`: