Ausgabe 4.12 Juni/Juli www.phpmagazin.de S&S

PHPmagazin

PHPmagazin

Deutschland 9,80 € Österreich 10,80 € | Schweiz 19,20 sFr
Niederlande 11,25 € | Luxemburg 11,25 €

CodeIgniter
Die große Serie

SPDY
HTTP war gestern

Zend_Date
Zeitzonen in PHP

MongoDB
Geo Data und NoSQL

PHP 5.4
Alles zum aktuellen Release

OAuth2
Den Standard vorgestellt

PDF-Bibliotheken
Der große Überblick

4 195649 609809 04

Verbesserte Performance des @-Operators

Shut up already

Der @-Operator stellt eine schnelle und einfache Variante dar, um Fehlermeldungen zu unterdrücken. Mit ihm einher geht immer auch eine Geschwindigkeitseinbuße, die in PHP 5.4 nun verringert wurde. Unterdrückte Fehlermeldungen tauchen anschließend aber in keiner Log-Datei mehr auf, was bedeutet, dass eine allzu umfangreiche Nutzung des Operators zu einer erschwerten Fehlersuche und -behebung führt. Dieser Artikel beleuchtet die Vor- und Nachteile des @-Operators, warum man ihn vermeiden sollte, wenn nicht unbedingt nötig, und wie groß der Unterschied zwischen PHP 5.3, PHP 5.4 und einer Nichtbenutzung wirklich ist.

von Michael Kliewe

Stellt man das @ in PHP vor einen Ausdruck, werden alle Fehlermeldungen, die von diesem Ausdruck erzeugt werden, unterdrückt. Ein Ausdruck kann zum Beispiel eine Variablenbenutzung oder ein Funktionsaufruf sein, häufig wird er bei *include()*-Aufrufen eingesetzt. Warum das meistens keine gute Idee ist, zeige ich später noch. In PHP 5.4 wurden nun die Geschwindigkeitseinbußen etwas verringert, die mit dieser Art der Fehlerunterdrückung einhergehen. Im Changelog steht: *Improved performance of @ (silence) operator*.

Der so genannte Fehler-Kontroll-Operator

Der @-Operator, auch Silent-, Silence- oder Shutup-Operator genannt, unterdrückt Fehlermeldungen, die in der entsprechenden Zeile bzw. den aufgerufenen Untermethoden auftreten. Auf den ersten Blick macht das natürlich Sinn, denn man möchte seinen Webseitenbesuchern schließlich keine Fehlermeldungen präsentieren, doch das Problem sollte besser anders gelöst werden. So unterdrückte Fehlermeldungen sind nämlich in keiner Log-Datei enthalten, sodass eine Fehlersuche für den Entwickler sehr nervenaufreibend werden kann. Besser ist es, den Fehler nicht zu unterdrücken damit er weggeloggt wird, aber die Ausgabe an den Browser zu deaktivieren (*display_errors = 0*). Man stelle sich sonst ein Projekt vor, in dem vor jeder zehnten Zeile der @-Operator steht, und dann bricht das Script plötzlich ohne Fehlermeldungen ab. Wie soll man da den Fehler finden?

Ich habe schon Code gesehen, in dem der Operator sehr häufig genutzt wurde. Vor jedem *fopen()* und *file_get_contents()*, bei dem auf eine lokale Datei zugegriffen wurde, war aus Prinzip ein @. Stattdessen hätte man besser vorher mit *file_exists()* prüfen sollen, ob die Datei existiert, oder noch besser mit *is_readable()* nachschauen sollen, ob die Datei existiert und lesbar ist.

Oder bei jeder Division, bei der der Divisor auch *0* sein könnte, wurde einfach ein @ gesetzt, um die Fehlermeldung „Division by zero“ zu unterdrücken. Das ist weder professionell noch hilft es dabei, mögliche Bugs zu finden. Es gibt jedoch Beispiele, in denen der Silence-Operator Sinn macht, beispielsweise *stream_socket_client()*, *mysql_connect()* oder *imap_open()*. Oder auch *fopen()* bei der Benutzung von Wrappern, bei denen kein *file_exists()* möglich ist. Falls es dort Verbindungsprobleme gibt, wird eine Fehlermeldung ausgegeben, und man kann die Verbindung nicht vorher mit *file_exists()* oder ähnlichen Prüfungen sinnvoll testen. Hier macht es also eventuell Sinn, die Fehlermeldung zu unterdrücken und danach zu prüfen, ob die Verbindung erfolgreich war, indem der Rückgabewert auf '*false*' überprüft wird. Die aufgetretene Fehlermeldung kann man dann im Falle der IMAP-Verbindung mittels *imap_errors()*, *imap_alerts()* bzw. *imap_last_error()* abfragen. Falls '*track_errors*' aktiviert ist, kann man den letzten Fehler auch in der Variable *$php_errormsg* finden. Falls nun aber zum Beispiel die Funktion *imap_open()* auf dem System nicht verfügbar ist oder man sich beim Funktionsnamen vertippt hat, bekommt man keine Fehlermeldung. Das Script bricht einfach so ab, und dann ist es zeitaufwändig, alle möglichen Vorkommnisse von @ zu überprüfen, um herauszufinden, wo denn nun der Fehler aufgetreten sein könnte. Der @-Operator bewirkt dasselbe, als wenn man manuell vor dem Ausdruck das *error_reporting* temporär auf *0* und nach dem Ausdruck wieder auf den alten Wert zurücksetzt.

Weniger Code

Manche Entwickler rechtfertigen die Nutzung damit, dass sie „wissen, was sie tun“. Ein einfaches Beispiel aus vielen Projekten:

```
<?php

if (isset($_GET['param']) && $_GET['param'] === 'a') {
    echo 'money';
}

if (@$_GET['param'] === 'a') {
    echo 'money';
}
```

Nach meinem Kenntnisstand verhalten sich beide *if*-Abfragen gleich, man könnte also die kürzere Variante wählen. Hier ein weiteres Beispiel mit der gleichen Struktur, nur wird hier eine Variable gegen NULL geprüft:

```
<?php

if (isset($var) && $var === null) {
    echo 'money';
}

if (@$var === null) {
    echo 'money';
}
```

Wenn hier die Variable *$var* nicht existiert, ist die erste Bedingung wegen dem *isset()* nicht erfüllt. Die Bedingung in der zweiten *if*-Abfrage jedoch ist wahr, und damit wird '*money*' ausgegeben. Wer das nicht weiß oder nicht testet, kann schnell schwer zu findende Bugs einbauen. Das ist vergleichbar mit *empty()* und *isset()*, bis auf einen Extremfall verhalten sich beide exakt gleich. Und irgendwann tritt genau dieser Fall dann ein.

Die Scream Extension

Um unterdrückte Fehler auf Entwicklersystemen leichter zu finden, gibt es Möglichkeiten, den @-Operator zu deaktivieren. Mit der PHP-Erweiterung Scream, zu finden im PECL Repository [1], wird der Code einfach ausgeführt, als wäre der @-Operator nicht da. Installiert wird es je nach Betriebssystem etwas unterschiedlich, hier am Beispiel eines Debian-Systems:

```
sudo apt-get install php-pear php5-dev scream-0.1.0
```

In der *php.ini* muss man dann die Extension eventuell noch laden und kann dann die Funktionalität aktivieren:

```
extension=scream.so;
scream.enabled=1
```

Möchte man es je nach Projekt oder Umgebung aktivieren, geht es auch via *ini_set()*:

```
ini_set('scream.enabled', true);
```

Unter Windows ist das Ganze nicht so einfach, ich habe keine fertige DLL gefunden.

Xdebug Scream

Seit Xdebug 2.1.0 gibt es dort auch die Scream-Funktionalität, und Xdebug ist schon auf den meisten Entwicklermaschinen installiert, sodass die Aktivierung nur noch eine Zeile in der *php.ini* bzw. ein *ini_set()*-Aufruf ist:

```
xdebug.scream = 1
ini_set('xdebug.scream', true);
```

Suchen und Finden

Soll ein vorhandenes Projekt auf Vorkommen des @-Operators durchsucht werden, ist eine einfache Suche nach dem Zeichen @ meist keine gute Idee. Je nachdem wie groß das Projekt ist, wird man tausende von PHP-Doc, Kommentaren und Annotations finden. Hilfreicher ist da beispielsweise der PHP_Codesniffer [2] mit seinem *NoSilencedErrorsSniff*.

Die Benchmark-Skripte

Wie groß ist also der Geschwindigkeitsverlust, mit dem man bei der Verwendung zu rechnen hat? Dazu habe ich zwei kleine Testskripte, die verschiedene Problemursachen betreffen. Zuerst möchten wir prüfen, wie sich ein erfolgreicher bzw. fehlschlagender *include()*-Aufruf auswirkt. Dazu wird der in Listing 1 folgende Schnipsel einmal mit existierender (aber leerer) *money.php* aufgerufen und einmal mit fehlender Datei.

Der zweite Test betrifft das häufig bei Programmieranfängern auftretende Problem der vergessenen String-Begrenzungen ' bzw. ". Falls sie vergessen werden und die entsprechende Konstante nicht existiert, ist PHP so schlau und nimmt an, dass eine Zeichenkette gemeint war. Eine Notice wird ausgegeben: „Notice: Use of undefined constant bar – assumed '*bar*'“. Das Script sieht

Listing 1

```
<?php

$start = microtime(true);
for ($i=0; $i<1000000; $i++) {
    @include 'money.php';
}
echo microtime(true)-$start;
echo "\n-----\n";

$start = microtime(true);
for ($i=0; $i<1000000; $i++) {
    include 'money.php';
}
echo microtime(true)-$start;
```

Listing 2

```
<?php

$start = microtime(true);
for ($i=0; $i<10000000; $i++) {
    @$foo = bar;
}
echo microtime(true)-$start;

$start = microtime(true);
for ($i=0; $i<10000000; $i++) {
    $foo = bar;
}
echo microtime(true)-$start;
```

wie in Listing 2 aus. Einmal wird dieses Script mit diesem Fehler ausgeführt – und einmal mit eingefügten Hochkommata.

Die Benchmark-Ergebnisse

Wenn es keine Fehler gibt, dann verliert man durch das Aktivieren der Fehlerunterdrückung im *include()*-Test nur sehr wenig, häufig weniger als 10 Prozent. Anders sieht es im zweiten Test aus, da läuft das Script plötzlich fünfmal so lang. Im Fehlerfall ist die Verwendung des Operators auch deutlicher messbar. Wenn man den Operator hinzufügt, ist das langsamer als die Nichtnutzung, allerdings nur wenn die auftretenden Fehler nicht in eine Log-Datei geschrieben werden. In diesen Extremfällen mit vielen Fehlern bremst das Logging der Millionen Fehlermeldungen extrem aus. Abhängig ob es sich um eine schnelle SSD oder eine „normale langsame“ Festplatte handelt, variiert die Laufzeit sehr stark. Bei massiv auftretenden Fehlern ist es also das Beste, den Fehler zu beheben bzw. zu vermeiden und dann ohne @-Zeichen zu arbeiten. Die Log-Datei kann man ja aktiv lassen, wenn im Normalfall kein Fehler auftritt. Da es hier pro Vorkommen nur um Nanosekunden geht, müssen die Schleifenkörper sehr oft ausgeführt werden. Falls es also nicht exzessiv in oft laufenden Schleifen benutzt wird, reden wir hier von nicht merkbaren Einbußen, so genannter Micro-Optimierung. Ein wirklich greifendes Gegenargument ist Performance gegen die Nutzung also nicht, da sind erschwertes Debugging und schwerer lesbarer Code deutlich wichtiger. Generell sieht man auch, dass PHP 5.4 in allen Tests schneller ist als 5.3, teilweise sogar sehr deutlich.

Der eigene Error Handler

Wer sich etwas intensiver mit dem Thema Fehlerbehandlung, Logging, Unterdrückung und Vermeidung selbiger beschäftigt, wird auch darauf stoßen, dass es möglich ist, in PHP einen eigenen Error Handler festzulegen, der im Fehlerfall aufgerufen wird. Ein Error Handler ist eine selbst geschriebene Funktion, die im Fehlerfall (Notice, Warning etc.) aufgerufen wird, und die sich dann um die Behandlung kümmern kann. Dort kann man dann ein eigenes Logging implementieren, nur auf bestimmte Fehler reagieren, das Script kontrolliert beenden oder den eingebauten PHP Error Handler zur Weiterbehandlung aufrufen. Das ist ein sehr flexibles Werkzeug, um nach eigenen Wünschen auf Fehler reagieren zu können.

Fazit

Bevor Fehler unterdrückt werden, sollte man ihnen auf den Grund gehen und sie lieber vermeiden, damit behält man eine leere Log-Datei und erhält im Zweifelsfall aussagekräftige Fehlermeldungen. Nicht selten deuten Notices und Warnings auf fehlerhaften bzw. unprofessionellen Code hin, es gibt eigentlich keine Ausflüchte die Probleme nicht zu lösen, und sich einzureden „wenn ich den Fehler nicht mehr sehe, ist er auch nicht mehr da“. Wenn es sich nicht vermeiden lässt (z. B. *mysql_connect()*), sollte man unbedingt den Rückgabewert und eventuell vorhandene Fehlervariablen (*$php_errormsg*) und -funktionen (*mysql_error()*) prüfen. Im PHP Manual gibt es natürlich auch eine Seite zum @-Operator [3] und dem ganzen Thema der Fehlerbehandlung und Protokollierung [4]. Happy Coding, Logging and Debugging!

> Wenn es sich nicht vermeiden lässt, sollte man unbedingt den Rückgabewert prüfen.

Michael Kliewe hat Informatik an der Universität Paderborn studiert und arbeitet nun als Programmierer bei mail.de in Gütersloh. In seiner Freizeit betreibt er einen der größten deutschen PHP-Blogs unter http://www.phpgangsta.de.

Links & Literatur

[1] http://pecl.php.net/package/scream
[2] http://pear.php.net/package/PHP_CodeSniffer
[3] http://www.php.net/manual/en/language.operators.errorcontrol.php
[4] http://www.php.net/manual/de/book.errorfunc.php (Kopfzeile)

	ohne @	mit @
PHP 5.3.10	11.1 s	12,2 s
PHP 5.4.0 RC7	10,9 s	11,6 s

Tabelle 1: 1 000 000 Schleifendurchläufe „include()“ mit existierender Datei

	ohne @	mit @	ohne @/log_errors=0
PHP 5.3.10	62.6 s (403 MB Log)	7.7 s	7.2 s
PHP 5.4.0 RC7	54,2 s (405 MB Log)	7,3s	7,0s

Tabelle 2: 1 000 000 Schleifendurchläufe „include()“ mit nicht existierender Datei

	ohne @	mit @
PHP 5.3.10	0.6 s	2.8 s
PHP 5.4.0 RC7	0,3 s	1,3 s

Tabelle 3: 10 000 000 Schleifendurchläufe Variablenzuweisung gültiger String

	ohne @	mit @	ohne @/log_errors=0
PHP 5.3.10	50s-282 s (1,6 GB Log)	5,5 s	3,1 s
PHP 5.4.0 RC7	48s-276 s (1,6 GB Log)	3,8 s	2,8 s

Tabelle 4: 10 000 000 Schleifendurchläufe Variablenzuweisung ungültiger String