php

Keyboard Shortcuts

?	This help
j	Next menu item
k	Previous menu item
g p	Previous man page
g n	Next man page
G	Scroll to bottom
g g	Scroll to top

PDO::prepare

(PHP 5 >= 5.1.0, PHP 7, PECL pdo >= 0.1.0)

PDO::prepare — Prepares a statement for execution and returns a statement object

Description ¶

public PDOStatement **PDO::prepare** (string `$statement` [, array `$driver_options` = array()])

Prepares an SQL statement to be executed by the PDOStatement::execute() method. The SQL statement can contain zero or more named (:name) or question mark (?) parameter markers for which real values will be substituted when the statement is executed. You cannot use both named and question mark parameter markers within the same SQL statement; pick one or the other parameter style. Use these parameters to bind any user-input, do not include the user-input directly in the query.

You must include a unique parameter marker for each value you wish to pass in to the statement when you call PDOStatement::execute(). You cannot use a named parameter marker of the same name more than once in a prepared statement, unless emulation mode is on.

> **Note**:
>
> Parameter markers can represent a complete data literal only. Neither part of literal, nor keyword, nor identifier, nor whatever arbitrary query part can be bound using parameters. For example, you cannot bind multiple values to a single parameter in the IN() clause of an SQL statement.

Calling **PDO::prepare()** and PDOStatement::execute() for statements that will be issued multiple times with different parameter values optimizes the performance of your application by allowing the driver to negotiate client and/or server side caching of the query plan and meta information, and helps to prevent SQL injection attacks by eliminating the need to manually quote the parameters.

PDO will emulate prepared statements/bound parameters for drivers that do not natively support them, and can also rewrite named or question mark style parameter markers to something more appropriate, if the driver supports one style but not the other.

Parameters ¶

`statement`

This must be a valid SQL statement template for the target database server.

`driver_options`

This array holds one or more key=>value pairs to set attribute values for the PDOStatement object that this method returns. You would most commonly use this to set the *PDO::ATTR_CURSOR* value to *PDO::CURSOR_SCROLL* to request a scrollable cursor. Some drivers have driver specific options that may be set at prepare-time.

Return Values ¶

If the database server successfully prepares the statement, **PDO::prepare()** returns a PDOStatement object. If the database server cannot successfully prepare the statement, **PDO::prepare()** returns **`FALSE`** or emits PDOException (depending on error handling).

Note:

Emulated prepared statements does not communicate with the database server so **PDO::prepare()** does not check the statement.

Examples ¶

Example #1 Prepare an SQL statement with named parameters

```
<?php
/* Execute a prepared statement by passing an array of values */
$sql = 'SELECT name, colour, calories
    FROM fruit
    WHERE calories < :calories AND colour = :colour';
$sth = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
$sth->execute(array(':calories' => 150, ':colour' => 'red'));
$red = $sth->fetchAll();
$sth->execute(array(':calories' => 175, ':colour' => 'yellow'));
$yellow = $sth->fetchAll();
?>
```

Example #2 Prepare an SQL statement with question mark parameters

```
<?php
/* Execute a prepared statement by passing an array of values */
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->execute(array(150, 'red'));
$red = $sth->fetchAll();
$sth->execute(array(175, 'yellow'));
```

```
$yellow = $sth->fetchAll();
?>
```

See Also ¶

- PDO::exec() - Execute an SQL statement and return the number of affected rows
- PDO::query() - Executes an SQL statement, returning a result set as a PDOStatement object
- PDOStatement::execute() - Executes a prepared statement

add a note

User Contributed Notes 23 notes

up
down
95
Anonymous ¶
3 years ago

```
To those wondering why adding quotes to around a placeholder is wrong, and why you can't use
placeholders for table or column names:

There is a common misconception about how the placeholders in prepared statements work: they are not
simply substituted in as (escaped) strings, and the resulting SQL executed. Instead, a DBMS asked to
"prepare" a statement comes up with a complete query plan for how it would execute that query,
including which tables and indexes it would use, which will be the same regardless of how you fill in
the placeholders.

The plan for "SELECT name FROM my_table WHERE id = :value" will be the same whatever you substitute
for ":value", but the seemingly similar "SELECT name FROM :table WHERE id = :value" cannot be
planned, because the DBMS has no idea what table you're actually going to select from.

Even when using "emulated prepares", PDO cannot let you use placeholders anywhere, because it would
have to work out what you meant: does "Select :foo From some_table" mean ":foo" is going to be a
column reference, or a literal string?

When your query is using a dynamic column reference, you should be explicitly white-listing the
columns you know to exist on the table, e.g. using a switch statement with an exception thrown in the
default: clause.
```

up
down
42
Simon Le Pine ¶
3 years ago

```
Hi All,

First time posting to php.net, a little nervous.

After a bunch of searching I've learned 2 things about prepared statements:
1.) It fails if you enclose in a single quote (')
This fails: "SELECT * FROM users WHERE email=':email'"
This works: "SELECT * FROM users WHERE email=:email"
2.) You cannot search with a prepared statement
This fails: "SELECT * FROM users WHERE :search=:email"
This succeeds: "SELECT * FROM users WHERE $search=:email"

In my case I allow the user to enter their username or email, determine which they've entered and set
$search to "username" or "email". As this value is not entered by the user there is no potential for
SQL injection and thus safe to use as I have done.

Hope that saves someone else from a lot of searching.
```

up
down
36
daniel dot egeberg at gmail dot com ¶
7 years ago

```
You can also pass an array of values to PDOStatement::execute(). This is also secured against SQL
injection. You don't necessarily have to use bindParam() or bindValue().
```

up
down
15
public at grik dot net ¶
4 years ago

```
With PDO_MYSQL you need to remember about the PDO::ATTR_EMULATE_PREPARES option.

The default value is TRUE, like
$dbh->setAttribute(PDO::ATTR_EMULATE_PREPARES,true);

This means that no prepared statement is created with $dbh->prepare() call. With exec() call PDO
replaces the placeholders with values itself and sends MySQL a generic query string.

The first consequence is that the call  $dbh->prepare('garbage');
reports no error. You will get an SQL error during the $dbh->exec() call.
The second one is the SQL injection risk in special cases, like using a placeholder for the table
name.

The reason for emulation is a poor performance of MySQL with prepared statements. Emulation works
significantly faster.
```

up
down
29
admin at wdfa dot co dot uk ¶
7 years ago

```
Note on the SQL injection properties of prepared statements.

Prepared statements only project you from SQL injection IF you use the bindParam or bindValue option.

For example if you have a table called users with two fields, username and email and someone updates
their username you might run

UPDATE `users` SET `user`='$var'

where $var would be the user submitted text.

Now if you did
<?php
$a=new PDO("mysql:host=localhost;dbname=database;","root","");
$b=$a->prepare("UPDATE `users` SET user='$var'");
```

```
$b->execute();
?>

and the user had entered  User', email='test for a test the injection would occur and the email would
be updated to test as well as the user being updated to User.

Using bindParam as follows
<?php
$var="User', email='test";
$a=new PDO("mysql:host=localhost;dbname=database;","root","");
$b=$a->prepare("UPDATE `users` SET user=:var");
$b->bindParam(":var",$var);
$b->execute();
?>

The sql would be escaped and update the username to User', email='test'
```

up
down
17

bg at enativ dot com ¶

2 years ago

```
if you run queries in a loop, don't include $pdo->prepare() inside the loop, it will save you some
resources (and time).

prepare statement inside loop:
for($i=0; $i<1000; $i++) {
    $rs = $pdo->prepare("SELECT `id` FROM `admins` WHERE `groupID` = :groupID AND `id` <> :id");
    $rs->execute([':groupID' => $group, ':id' => $id]);
}

// took 0.066626071929932 microseconds

prepare statement outside loop:
$rs = $pdo->prepare("SELECT `id` FROM `admins` WHERE `groupID` = :groupID AND `id` <> :id");
for($i=0; $i<1000; $i++) {
    $rs->execute([':groupID' => $group, ':id' => $id]);
}

// took 0.064448118209839 microseconds

for 1,000 (simple) queries it took 0.002 microseconds less.
not much, but it worth mention.
```

up
down
11

pbakhuis ¶

2 years ago

```
Noteworthy in my opinion is that if you prepare a statement but do not bind a value to the markers it
will insert null by default. e.g.
<?php
/** @var PDO $db */
$prep = $db->prepare('INSERT INTO item(title, link) VALUES(:title, :link)');
$prep->execute();
```

```
?>
Will attempt to insert null, null into the item table.
```

up
down
5
orrd101 at gmail dot com ¶
4 years ago

```
Don't just automatically use prepare() for all of your queries.

If you are only submitting one query, using PDO::query() with PDO::quote() is much faster (about 3x
faster in my test results with MySQL).  A prepared query is only faster if you are submitting
thousands of identical queries at once (with different data).

If you Google for performance comparisons you will find that this is generally consistently the case,
or you can write some code and do your own comparison for your particular configuration and query
scenario. But generally PDO::query() will always be faster except when submitting a large number of
identical queries.  Prepared queries do have the advantage of escaping the data for you, so you have
to be sure to use quote() when using query().
```

up
down
4
roth at egotec dot com ¶
10 years ago

```
Attention using MySQL and prepared statements.
Using a placeholder multiple times inside a statement doesn't work. PDO just translates the first
occurance und leaves the second one as is.

select id,name from demo_de where name LIKE :name OR name=:name

You have to use

select id,name from demo_de where name LIKE :name OR name=:name2

and bind name two times. I don't know if other databases (for example Oracle or MSSQL) support
multiple occurances. If that's the fact, then the PDO behaviour for MySQL should be changed.
```

up
down
1
Robin ¶
6 years ago

```
Use prepared statements to ensure integrity of binary data during storage and retrieval.
Escaping/quoting by f.e. sqlite_escape_string() or PDO::quote() is NOT suited for binary data - only
for strings of text.

A simple test verifies perfect storage and retrieval with prepared statements:

<?php

$num_values = 10000;

$db = new pdo( 'sqlite::memory:' );

$db->exec( 'CREATE TABLE data (binary BLOB(512));' );
```

```
// generate plenty of troublesome, binary data
for( $i = 0; $i < $num_values; $i++ )
{
    for( $val = null, $c = 0; $c < 512/16; $c++ )
        $val .= md5( mt_rand(), true );
    @$binary[] = $val;
}

// insert each value by prepared statement
for( $i = 0; $i < $num_values; $i++ )
    $db->prepare( 'INSERT INTO data VALUES (?);' )->execute( array($binary[$i]) );

// fetch the entire row
$data = $db->query( 'SELECT binary FROM data;' )->fetchAll( PDO::FETCH_COLUMN );

// compare with original array, noting any mismatch
for( $i = 0; $i < $num_values; $i++ )
    if( $data[$i] != $binary[$i] ) echo "[$i] mismatch\n";

$db = null;

?>
```

up
down
0
php dot chaska at xoxy dot net ¶
3 years ago

```
Note that for Postgres, even though Postgres does support prepared statements, PHP's PDO driver NEVER
sends the prepared statement to the Postgres server in advance of the call to PDO::execute().

Therefore, PDO::prepare() will never throw an error for things like faulty SQL syntax.

It also means the server will not parse and plan the SQL until the first time PDO::execute() is
called, which may or may not adversely affect your optimization plans.
```

up
down
-1
Stan ¶
9 years ago

```
Using prepared SELECT statements on a MySQL database prior to MySQL 5.1.17 can lead to SERIOUS
performance degradation.

Quote from http://dev.mysql.com/doc/refman/5.1/en/query-cache.html :

>> The query cache is not used for server-side prepared statements before MySQL 5.1.17 <<

The MySQL query cache buffers complete query results and is used to satisfy repeated identical
queries if the underlying tables do not change in the meantime - just what happens all the time in a
typical web application. It speeds up queries by a several hundred to a several thousand percent.

Obviously, it doesn't make much sense to give up query caching for the relatively small performance
benefit of prepared statements (i.e. the DBMS not having to parse and optimize the same query
multiple times) - so using PDO->query() for SELECT statements is probably the better choice i you're
connecting to MySQL < 5.1.17.
```

up
down
-1
***johniskew* ¶**
9 years ago

If you need to create variable sql statements in a prepare statement...for example you may need to construct a sql query with zero, one, two, etc numbers of arguments...here is a way to do it without a lot of if/else statements needed to glue the sql together:

```
<?php

    public function matchCriteria($field1=null,$field2=null,$field3=null) {
        $db=DB::conn();
        $sql=array();
        $paramArray=array();
        if(!empty($field1)) {
            $sql[]='field1=?';
            $paramArray[]=$field1;
        }
        if(!empty($field2)) {
            $sql[]='field2=?';
            $paramArray[]=$field2;
        }
        if(!empty($field3)) {
            $sql[]='field3=?';
            $paramArray[]=$field3;
        }
        $rs=$db->prepare('SELECT * FROM mytable'.(count($paramArray)>0 ? ' WHERE '.join(' AND ',$sql)
: ''));
        $result=$rs->execute($paramArray);
        if($result) {
            return $rs;
        }
        return false;
    }

?>
```

up
down
-1
***ak_9jsz* ¶**
8 years ago

Using cursors doesn't work with SQLite 3.5.9. I get an error message when it gets to the execute() method.

Some of you might be saying "duh!" but i was surprised to see TRIGGER support in SQLite, so i had to try. :)

I wanted to use Absolute referencing on a Scrollable cursor and i only wanted one column of data. So i used this instead of a cursor.

```
<?php

$dbo = new PDO('sqlite:tdb');
```

```
$sql = 'SELECT F1, F2 FROM tblA WHERE F1 <> "A";';
$res = $dbo->prepare($sql);
$res->execute();
$resColumn = $res->fetchAll(PDO::FETCH_COLUMN, 0);

for($r=0;$r<=3;$r++)
    echo 'Row '. $r . ' returned: ' . $resColumn[$r] . "\n";

$dbo = null;
$res = null;
?>
```

up
down
-2
jesse dot chisholm at gmail dot com ¶
2 years ago

```
@Simon Le Pine

Be aware that:

$search = "user";
$sth = db->prepare("SELECT * FROM users WHERE $search=:email");

and

$search = "email";
$sth = db->prepare("SELECT * FROM users WHERE $search=:email");

will produce two totally different prepared statements.

Doing this _will not work_:

$search = "user";
$sth = db->prepare("SELECT * FROM users WHERE $search=:email");
$sth->execute(array(email=>"yada"));
$search = "email";
$sth->execute(array(email=>"yada@ya.da"));
```

up
down
-2
Hayley Watson ¶
3 years ago

It is possible to prepare in advance several statements against a single connection. As long as that connection remains open the statements can be executed and fetched from as often as you like in any order; their "prepare-execute-fetch" steps can be interleaved in whichever way is best.

So if you're likely to be using several statements often (perhaps within a loop of transactions), you may like to consider preparing all the statements you'll be using up front.

up
down
-4
richard at codevanilla.com ¶
7 years ago

```
beware
PDO will emulate prepared statements/bound parameters for drivers that do not natively support them,
and can also rewrite named or question mark style parameter markers to something more appropriate, if
the driver supports one style but not the other.

This includes mySQL it seems so

<?php
try{
        $sth1 = $this->db1->prepare($t1, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));

        }
        catch(PDOException $e){
                return $this->pack('dbError', $e->getMessage());
        }
?>

does not and so will not throw the exception if your SQL is wrong.

You will need to check that $sth1 is not null.
```

up
down
-3
william dot clarke at gmail dot com ¶
10 years ago

```
Surely if you want to use prepared statements that way you should use the syntax in the second
example:

eg.

instead of:
select id,name from demo_de where name LIKE :name OR name=:name

use:
select id,name from demo_de where name LIKE ? OR name=?

I believe you are supposed to either use distinct named parameters (name, name1) OR anonymous
parameters (?s)
```

up
down
-6
Kjetil H ¶
3 years ago

```
Please note that the correct internal method signature is:
<?php public function prepare ($statement, $driver_options = array()) ?>

and NOT:
<?php public function prepare ($statement, array $driver_options = array()) ?>.

Redeclaring the method using the latter method signature throws a Stricts Standards error.
```

up
down
-9
pascal dot buguet at laposte dot net ¶

6 years ago

```
PDO::CURSOR_SCROLL is ok with MSS.
You must install SQL Server Driver for PHP 2.0 CTP2 : SQLSRV20.EXE
and  the native client "Microsoft SQL Server 2008 R2 Native Client" : sqlncli.msi.
```

up
down
-8
chatelain dot cedric dot pro at gmail dot com ¶
1 year ago

```
you can't use CREATE DATABASE with prepared statement.

$sql = $conn->prepare("DROP DATABASE IF EXISTS :dbname ;",
                    array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
            $sql->execute(array(':dbname' => $dbname));

This will not work.
Anyone has an explanation ?
```

up
down
-4
sgirard at rossprint dot com ¶
7 years ago

```
Maybe everyone else already knows this but...

If you have a routine that prepares/executes many insert or update statements for a sqlite db then
you may want to make use of the pdo transactions.

On some old hardware my query set went from 12 seconds to 1/3-1/2 second.

-sean
```

up
down
-4
www.onphp5.com ¶
9 years ago

```
Please note that the statement regarding driver_options is misleading:

"This array holds one or more key=>value pairs to set attribute values for the PDOStatement object
that this method returns. You would most commonly use this to set the PDO::ATTR_CURSOR value to
PDO::CURSOR_SCROLL to request a scrollable cursor. Some drivers have driver specific options that may
be set at prepare-time"

From this you might think that scrollable cursors work for all databases, but they don't! Check out
this bug report:
http://bugs.php.net/bug.php?id=34625
```

add a note

- PDO
  - beginTransaction
  - commit
  - __construct
  - errorCode
  - errorInfo
  - exec