Как работать с PDO? Полное руководство.

Соединение
Обработка исключений
Выполнение запросов.
Подготовленные выражения
Подготовленные выражения. Множественное выполнение.
Получение данных. fetch()
Получение данных. fetchColumn()
Получение данных. fetchAll()
PDO и оператор LIKE
PDO и оператор LIMIT
PDO и оператор IN
PDO и имена полей/таблиц
PDO и запросы INSERT/UPDATE
PDO и ключевые слова
Комментарии (79)

Соединение

У PDO свой собственный способ соединения, называемый DSN. Плюс во время коннекта можно задать набор опций, некоторые из которых чрезвычайно полезны. Полный список можно найти здесь, но важными из них являются только несколько.

Пример правильного соединения:

```
    $host = '127.0.0.1';
    $db   = 'test';
    $user = 'root';
    $pass = '';
    $charset = 'utf8';

    $dsn = "mysql:host=$host;dbname=$db;charset=$charset";
    $opt = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => false,
    ];
    $pdo = new PDO($dsn, $user, $pass, $opt);
```

Что здесь происходит?

- в $dsn задается тип БД, с которым будем работать (mysql), хост, имя базы данных и чарсет.
- затем идут имя пользователя и пароль
- после которого задается массив опций, про который ни в одном из руководств не пишут.

При том что этот массив - чрезвычайно полезная, как уже говорилось выше, штука. Самое главное - режим выдачи ошибок надо задавать только в виде исключений.
- Во-первых, потому что во всех остальных режимах PDO не сообщает об ошибке ничего внятного,
- во-вторых, потому что исключение всегда содержит в себе незаменимый stack trace,
- в-третьих - исключения чрезвычайно удобно обрабатывать.

Плюс очень удобно задать FETCH_MODE по умолчанию, чтобы не писать его в КАЖДОМ запросе. Также здесь можно задавать режим pconnect-а, эмуляции подготовленных выражений и много других страшных слов.

В результате мы получаем переменную $pdo, с которой и работаем далее на протяжении всего скрипта.

Обработка исключений

Поскольку практически КАЖДЫЙ мануал по PDO считает своим долгом продемонстрировать в корне неверный способ обработки исключений, я должен сделать специальное пояснение.
Обычно пример выглядит как-то так

```
try {
    $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    die('Подключение не удалось: ' . $e->getMessage());
}
```

Так делать не надо НИКОГДА:
- во-первых, этот код избыточен: PHP покажет ошибку и так, безо всяких try..catch.
- во-вторых, этот код гораздо менее гибкий: он выводит ошибку ТОЛЬКО на экран, в то время как исключение улетит туда же, куда и все остальные ошибки: либо в лог файл, либо на экран, в зависимости от глобальных настроек.
- в-третьих, этот код лишает нас возможности обрабатывать все ошибки централизованно, в едином exception handler-е.

Поэтому использовать try..catch нужно только тогда, когда вы собираетесь ОБРАБОТАТЬ ошибку - то есть, совершить какое-то действие, связанное с ФАКТОМ ошибки - откатить транзакцию, например. Для того же, чтобы просто выдать сообщение об ошибке, try..catch использовать не нужно - PHP прекрасно справится сам

Выполнение запросов.

Для выполнения запросов можно пользоваться двумя методами.
Если в запрос не передаются никакие переменные, то можно воспользоваться функцией query(). Она выполнит запрос и вернёт специальный объект — PDO statement. Очень грубо можно его сравнить с mysql resource, который возвращала mysql_query(). Получить данные из этого объекта можно как традиционным образом, через while, так и через foreach(). Также можно попросить вернуть полученные данные в особом формате, о чем ниже.

```
$stmt = $pdo->query('SELECT name FROM users');
while ($row = $stmt->fetch())
{
    echo $row['name'] . "\n";
}
```

Подготовленные выражения

Если же в запрос передаётся хотя бы одна переменная, то этот запрос в обязательном порядке должен выполняться только через **подготовленные выражения**. Что это такое? Это обычный SQL запрос, в котором вместо переменной ставится специальный маркер - плейсхолдер. PDO поддерживает позиционные плейсхолдеры (?), для которых важен порядок передаваемых переменных, и именованные (:name), для которых порядок не важен. Примеры:

```
$sql = 'SELECT name FROM users WHERE email = ?';
$sql = 'SELECT name FROM users WHERE email = :email';
```

Чтобы выполнить такой запрос, сначала его надо подготовить с помощью функции prepare(). Она также возвращает PDO statement, но ещё без данных. Чтобы их получить, надо исполнить этот запрос, предварительно передав в него переменные. Передать можно двумя способами:
Чаще всего можно просто выполнить метод execute(), передав ему массив с переменными:

```
$stmt = $pdo->prepare('SELECT name FROM users WHERE email = ?');
$stmt->execute(array($email));

$stmt = $pdo->prepare('SELECT name FROM users WHERE email = :email');
$stmt->execute(array('email' => $email));
```

Как видно, в случае именованных плейсхолдеров в execute() должен передаваться массив, в

котором ключи должны совпадать с именами плейсхолдеров.

Иногда, очень редко, может потребоваться второй способ, когда переменные сначала привязывают к запросу по одной, с помощью bindValue() / bindParam(), а потом только исполняют. В этом случае в execute() ничего не передается.
Используя этот метод, всегда следует предпочесть bindValue()? поскольку поведение bindParam() не очевидно для новичков и будет приводить к проблемам.

После этого можно использовать PDO statement теми же способами, что и выше. Например, через foreach:

```
$stmt = $pdo->prepare('SELECT name FROM users WHERE email = ?');
$stmt->execute([$_GET['email']]);
foreach ($stmt as $row)
{
    echo $row['name'] . "\n";
}
```

ВАЖНО: Подготовленные выражения - основная причина использовать PDO, поскольку это **единственный безопасный способ** выполнения SQL запросов, в которых участвуют переменные.

Подготовленные выражения. Множественное выполнение.

Также prepare() / execute() могут использоваться для многократного выполнения единожды подготовленного запроса с разными наборами данных. На практике это бывает нужно чрезвычайно редко, и особого прироста в скорости не приносит. Но на случай, если понадобится делать много однотипных запросов, то можно писать так:

```
$data = array(
1 => 1000,
5 => 300,
9 => 200,
);

$stmt = $pdo->prepare('UPDATE users SET bonus = bonus + ? WHERE id = ?');
foreach ($data as $id => $bonus)
{
    $stmt->execute([$bonus,$id]);
}
```

Здесь мы один раз подготавливаем запрос, а затем много раз выполняем.

Получение данных. fetch()

Мы уже выше познакомились с методом fetch(), который служит для последовательного получения строк из БД. Этот метод является аналогом функции mysq_fetch_array() и ей подобных, но действует по-другому: вместо множества функций здесь используется одна, но ее поведение задается переданным параметром. В подробностях об этих параметрах будет написано позже, а в качестве краткой рекомендации посоветую применять fetch() в режиме `FETCH_LAZY`:

```
$stmt = $pdo->prepare('SELECT name FROM users WHERE email = ?');
$stmt->execute([$_GET['email']]);
while ($row = $stmt->fetch(PDO::FETCH_LAZY))
{
    echo $row[0] . "\n";
    echo $row['name'] . "\n";
    echo $row->name . "\n";
}
```

В этом режиме не тратится лишняя память, и к тому же к колонкам можно обращаться любым из трех способов - через индекс, имя, или свойство.

Получение данных. fetchColumn()

Также у PDO statement есть функция-хелпер для получения значения единственной колонки. Очень удобно, если мы запрашиваем только одно поле - в этом случае значительно сокращается количество писанины:

```
$stmt = $pdo->prepare("SELECT name FROM table WHERE id=?");
$stmt->execute(array($id));
$name = $stmt->fetchColumn();
```

Получение данных. fetchAll()

Но самой интересной функцией, с самым большим функционалом, является fetchAll(). Именно она делает PDO высокоуровневой библиотекой для работы с БД, а не просто низкоуровневым драйвером.

fetchAll() возвращает массив, который состоит из всех строк, которые вернул запрос. Из чего можно сделать два вывода:
1. Эту функцию не стоит применять тогда, когда запрос возвращает много данных. В таком случае лучше использовать традиционный цикл с fetch()
2. Поскольку в современных PHP приложениях данные никогда не выводятся сразу по получении, а передаются для этого в шаблон, fetchAll() становится просто незаменимой, позволяя не писать циклы вручную, и тем самым сократить количество кода.

Получение простого массива.
Вызванная без параметров, эта функция возвращает обычный индексированный массив, в котором лежат строки из бд, в формате, который задан в FETCH_MODE по умолчанию. Константы PDO::FETCH_NUM, PDO::FETCH_ASSOC, PDO::FETCH_OBJ могут менять формат на лету.

Получение колонки.
Иногда бывает нужно получить простой одномерный массив, запросив единственное поле из кучи строк. Для этого используется режим PDO::FETCH_COLUMN

```
$data = $pdo->query('SELECT name FROM users')->fetchAll(PDO::FETCH_COLUMN);
array (
  0 => 'John',
  1 => 'Mike',
  2 => 'Mary',
  3 => 'Kathy',
)
```

Получение пар ключ-значение.
Также востребованный формат, когда желательно получить ту же колонку, но индексированную не числами, а одним из полей. За это отвечает константа PDO::FETCH_KEY_PAIR.

```
$data = $pdo->query('SELECT id, name FROM users')->fetchAll(PDO::FETCH_KEY_PAIR);
array (
  104 => 'John',
  110 => 'Mike',
  120 => 'Mary',
  121 => 'Kathy',
)
```

Следует помнить, что первой в колонкой надо обязательно выбирать уникальное поле.

Получение всех строк, индексированных полем.
Также часто бывает нужно получить все строки из БД, но также индексированные не числами, а уникальным полем. Это делает константа PDO::FETCH_UNIQUE

```
$data = $pdo->query('SELECT * FROM users')->fetchAll(PDO::FETCH_UNIQUE);
array (
  104 => array (
    'name' => 'John',
    'car' => 'Toyota',
  ),
  110 => array (
    'name' => 'Mike',
    'car' => 'Ford',
  ),
  120 => array (
    'name' => 'Mary',
    'car' => 'Mazda',
  ),
  121 => array (
    'name' => 'Kathy',
    'car' => 'Mazda',
  ),
)
```

Следует помнить, что первой в колонкой надо обязательно выбирать уникальное поле.

Всего различных режимов получения данных в PDO больше полутора десятков. Плюс ещё их можно комбинировать! Но это уже тема для отдельной статьи.

PDO и оператор LIKE

Работая с подготовленными выражениями, следует понимать, что плейсхолдер может заменять только строку или число. Ни ключевое слово, ни идентификатор, ни часть строки или набор строк через плейсхолдер подставить нельзя. Поэтому для LIKE надо сначала подготовить строку поиска целиком, а потом ее подставлять в запрос:

```
$name = "%$name%";
$stm  = $pdo->prepare("SELECT * FROM table WHERE name LIKE ?");
$stm->execute(array($name));
$data = $stm->fetchAll();
```

PDO и оператор LIMIT

Здесь есть один нюанс.
Когда PDO работает в режиме эмуляции, все данные, которые были переданы напрямую в execute(), форматируются как строки. То есть, эскейпятся и обрамляются кавычками.
Поэтому `LIMIT ?,?` превращается в `LIMIT '10', '10'` и очевидным образом вызывает ошибку синтаксиса.

Соответственно, есть 2 решения:
- либо отключить режим эмуляции

```
$conn->setAttribute( PDO::ATTR_EMULATE_PREPARES, false );
```

Либо биндить эти цифры через bindValue, принудительно выставляя им тип PDO::PARAM_INT.

```
$stm = $pdo->prepare('SELECT * FROM table LIMIT ?, ?');
$stm->bindValue(1, $limit_from, PDO::PARAM_INT);
$stm->bindValue(2, $per_page, PDO::PARAM_INT);
$stm->execute();
$data = $stm->fetchAll();
```

PDO и оператор IN

Как уже говорилось выше, плейсхолдер может представлять только строку или число.
Подставить набор данных на место одного плейсхолдера не получится.
Поэтому для IN придется изворачиваться, динамически формируя две переменные:
- набор вопросиков через запятую по числу элементов в IN()
- и массив данных для подстановки.

В отличие от mysqli кода получается не очень много, но все равно - сразу не сообразишь:

```php
$arr = array(1,2,3);
$in  = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM table WHERE column IN ($in)";
$stm = $db->prepare($sql);
$stm->execute($arr);
$data = $stm->fetchAll();
```

PDO и имена полей/таблиц

Ну, вы поняли. Тут тоже всё плохо. PDO не предоставляет вообще никаких средств для работы с идентификаторами, и их надо форматировать по-старинке, вручную (или посмотреть, все-таки, в сторону SafeMysql, в которой этот, как и многие другие вопросы, решены просто и элегантно).
Следует помнить, что правила форматирования идентификаторов отличаются для разных БД.

В mysql для ручного форматирования идентификатора необходимо выполнить два действия:
- заключить его в обратные одинарные кавычки (backticks, "`").
- проискейпить эти символы внутри идентификатора внутри путём удвоения.

```php
$field = "`".str_replace("`","``",$_GET['field'])."`";
$sql   = "SELECT * FROM `table` ORDER BY $field";
```

Однако, здесь есть один нюанс. Одного форматирования может быть недостаточно. приведенный выше код гарантирует нас от классической инъекции, но в некоторых случаях враг все равно может записать что-то нежелательное, если мы бездумно подставляем имена полей и таблиц прямиком в запрос. К примеру, есть в таблице users поле admin. Если входящие имена полей не фильтровать, то в это поле, при автоматическом формировании запроса из POST-а, любой дурак запишет любую гадость.

Поэтому имена таблиц и полей, приходящие от юзера, желательно проверять на допустимость, как в приведённом ниже примере

PDO и запросы INSERT/UPDATE

Любой код для вставки, который можно увидеть в многочисленных туториалах, навевает тоску и желание убиться апстену. Многокилометровые построения с повторением одних и тех же имен - в идексах $_POST-а, в именах переменных, в именах полей в запросе, в именах плейсхолдеров в запросе, в именах плейсходеров и именах переменных при привязке.
Глядя на этот код, хочется кого-нибудь убить, или, по крайней мере, сделать его немного короче.

Это можно сделать, если принять соглашение, по которому имена полей в форме будут соответствовать именам полей в таблице. Тогда эти имена можно будет перечислить только один раз (в целях защиты от подмены, о которой говорилось выше), и использовать небольшую функцию-хелпер для сборки запроса, которая, в силу особенностей mysql, годится как для INSERT, так и UPDATE запросов:

```php
function pdoSet($allowed, &$values, $source = array()) {
  $set = '';
  $values = array();
  if (!$source) $source = &$_POST;
  foreach ($allowed as $field) {
    if (isset($source[$field])) {
      $set.="`".str_replace("`","``",$field)."`". "=:$field, ";
      $values[$field] = $source[$field];
    }
  }
```

```
  return substr($set, 0, -2);
}
```

Соответственно, для вставки код будет

```
$allowed = array("name","surname","email"); // allowed fields
$sql = "INSERT INTO users SET ".pdoSet($allowed,$values);
$stm = $dbh->prepare($sql);
$stm->execute($values);
```

А для апдейта - такой:

```
$allowed = array("name","surname","email","password"); // allowed fields
$_POST['password'] = MD5($_POST['login'].$_POST['password']);
$sql = "UPDATE users SET ".pdoSet($allowed,$values)." WHERE id = :id";
$stm = $dbh->prepare($sql);
$values["id"] = $_POST['id'];
$stm->execute($values);
```

Не слишком эффектно, но зато очень эффективно. Напомню, кстати, что если использовать Класс для безопасной и удобной работы с MySQL, то это всё делается в две строчки.

PDO и ключевые слова

Здесь кроме фильтрации ничего придумать невозможно. поэтому тупо прогонять все не прописанные в запросе напрямую операторы через белый список:

```
$dirs = array("ASC","DESC");
$key  = array_search($_GET['dir'],$dirs));
$dir  = $orders[$key];
$sql  = "SELECT * FROM `table` ORDER BY $field $dir";
```