Введение в PDO

> «PDO – PHP Data Objects – это прослойка, которая предлагает универсальный способ работы с несколькими базами данных.»

Заботу об особенностях синтаксиса различных СУБД она оставляет разработчику, но делает процесс переключения между платформами гораздо менее болезненным. Нередко для этого требуется лишь изменить строку подключения к базе данных.

Эта статья написана для людей, которые пользуются mysql и mysqli, чтобы помочь им в переходе на более мощный и гибкий PDO.

Поддержка СУБД

Это расширение может поддерживать любую систему управления базами данных, для которой существует PDO-драйвер. На момент написания статьи доступны следующие драйвера:

- PDO_CUBRID (CUBRID)
- PDO_DBLIB (FreeTDS / Microsoft SQL Server / Sybase)
- PDO_FIREBIRD (Firebird/Interbase 6)
- PDO_IBM (IBM DB2)
- PDO_INFORMIX (IBM Informix Dynamic Server)
- PDO_MYSQL (MySQL 3.x/4.x/5.x)
- PDO_OCI (Oracle Call Interface)
- PDO_ODBC (ODBC v3 (IBM DB2, unixODBC and win32 ODBC))
- PDO_PGSQL (PostgreSQL)
- PDO_SQLITE (SQLite 3 and SQLite 2)
- PDO_SQLSRV (Microsoft SQL Server)
- PDO_4D (4D)

Впрочем, не все из них есть на вашем сервере. Увидеть список доступных драйверов можно так:

```
print_r(PDO::getAvailableDrivers());
```

Подключение

Способы подключения к разным СУБД могут незначительно отличаться. Ниже приведены примеры подключения к наиболее популярным из них. Можно заметить, что первые три имеют идентичный синтаксис, в отличие от SQLite.

```
try {
    # MS SQL Server и Sybase через PDO_DBLIB
    $DBH = new PDO("mssql:host=$host;dbname=$dbname", $user, $pass);
    $DBH = new PDO("sybase:host=$host;dbname=$dbname", $user, $pass);

    # MySQL через PDO_MYSQL
    $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
```

```
    # SQLite
    $DBH = new PDO("sqlite:my/database/path/database.db");
}   catch(PDOException $e) {
    echo $e->getMessage();
}
```

Пожалуйста, обратите внимание на блок try/catch – всегда стоит оборачивать в него все свои PDO-операции и использовать механизм исключений (об этом чуть дальше).

$DBH расшифровывается как «database handle» и будет использоваться на протяжении всей статьи.

Закрыть любое подключение можно путем переопределения его переменной в null.

```
# закрывает подключение
$DBH = null;
```

Больше информации по теме отличительных опций разных СУБД и методах подключения к ним можно найти на php.net.

Исключения и PDO

PDO умеет выбрасывать исключения при ошибках, поэтому все должно находиться в блоке try/catch. Сразу после создания подключения, PDO можно перевести в любой из трех режимов ошибок:

```
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING );
$DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

Но стоит заметить, что ошибка при попытке соединения будет всегда вызывать исключение.

PDO::ERRMODE_SILENT

Это режим по умолчанию. Примерно то же самое вы, скорее всего, используете для отлавливания ошибок в расширениях mysql и mysqli. Следующие два режима больше подходят для DRYпрограммирования.

PDO::ERRMODE_WARNING

Этот режим вызовет стандартный Warning и позволит скрипту продолжить выполнение. Удобен при отладке.

PDO::ERRMODE_EXCEPTION

В большинстве ситуаций этот тип контроля выполнения скрипта предпочтителен. Он выбрасывает исключение, что позволяет вам ловко обрабатывать ошибки и скрывать щепетильную информацию. Как, например, тут:

```
# подключаемся к базе данных  try {
    $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
    $DBH->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

    # Черт! Набрал DELECT вместо SELECT!
    $DBH->prepare('DELECT name FROM people')->execute();
}   catch(PDOException $e) {
    echo "Хьюстон, у нас проблемы.";
```

```
    file_put_contents('PDOErrors.txt', $e->getMessage(), FILE_APPEND);
}
```

В SQL-выражении есть синтаксическая ошибка, которая вызовет исключение. Мы можем записать детали ошибки в лог-файл и человеческим языком намекнуть пользователю, что что-то случилось.

Insert и Update

Вставка новых и обновление существующих данных являются одними из наиболее частых операций с БД. В случае с PDO этот процесс обычно состоит из двух шагов. (В следующей секции все относится как к UPDATE, так и INSERT)

Тривиальный пример вставки новых данных:

```
# STH означает "Statement Handle"
$STH = $DBH->prepare("INSERT INTO folks ( first_name ) values ( 'Cathy' )");
$STH->execute();
```

Вообще-то можно сделать то же самое одним методом exec(), но двухшаговый способ дает все преимущества prepared statements. Они помогают в защите от SQL-инъекций, поэтому имеет смысл их использовать даже при однократном запросе.

Prepared Statements

> Использование prepared statements укрепляет защиту от SQL-инъекций.

Prepared statement — это заранее скомпилированное SQL-выражение, которое может быть многократно выполнено путем отправки серверу лишь различных наборов данных. Дополнительным преимуществом является невозможность провести SQL-инъекцию через данные, используемые в placeholder'ах.

Ниже находятся три примера prepared statements.

```
# без placeholders - дверь SQL-инъекциям открыта!
$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values ($name, $addr, $city)
");
    # безымянные placeholders
$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values (?, ?, ?)");
    # именные placeholders
$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values (:name, :addr, :city)
");
```

Первый пример здесь лишь для сравнения, его стоит избегать. Разница между безымянными и именными placeholder'ами в том, как вы будете передавать данные в prepared statements.

Безымянные placeholder'ы

```
# назначаем переменные каждому placeholder, с индексами от 1 до 3
$STH->bindParam(1, $name);
```

```
$STH->bindParam(2, $addr);
$STH->bindParam(3, $city);
    # вставляем одну строку
$name = "Daniel"
$addr = "1 Wicked Way";
$city = "Arlington Heights";
$STH->execute();
    # вставляем еще одну строку, уже с другими данными
$name = "Steve"
$addr = "5 Circle Drive";
$city = "Schaumburg";
$STH->execute();
```

Здесь два шага. На первом мы назначаем всем placeholder'ам переменные (строки 2-4). Затем назначаем этим переменным значения и выполняем запрос. Чтобы послать новый набор данных, просто измените значения переменных и выполните запрос еще раз.

Если в вашем SQL-выражении много параметров, то назначать каждому по переменной весьма неудобно. В таких случаях можно хранить данные в массиве и передавать его:

```
# набор данных, которые мы будем вставлять
$data = array('Cathy', '9 Dark and Twisty Road', 'Cardiff');

$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values (?, ?, ?)");
$STH->execute($data);
```

$data[0] вставится на место первого placeholder'а, $data[1] — на место второго, и т.д. Но будьте внимательны: если ваши индексы сбиты, это работать не будет.

Именные placeholder'ы

```
# первым аргументом является имя placeholder'а# его принято начинать с двоеточия# хотя работает и без них
$STH->bindParam(':name', $name);
```

Здесь тоже можно передавать массив, но он должен быть ассоциативным. В роли ключей должны выступать, как можно догадаться, имена placeholder'ов.

```
# данные, которые мы вставляем
$data = array( 'name' => 'Cathy', 'addr' => '9 Dark and Twisty', 'city' => 'Cardiff' );

$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values (:name, :addr, :city)
");
$STH->execute($data);
```

Одним из удобств использования именных placeholder'ов является возможность вставки объектов напрямую в базу данных, если названия свойств совпадают с именами параметров. Вставку данных, к примеру, вы можете выполнить так:

```
# класс для простенького объекта  class person {
    public $name;
    public $addr;
    public $city;

    function __construct($n,$a,$c) {
        $this->name = $n;
        $this->addr = $a;
        $this->city = $c;
    }
    # так далее...
}
```

```
$cathy = new person('Cathy','9 Dark and Twisty','Cardiff');
  # а тут самое интересное
$STH = $DBH->prepare("INSERT INTO folks (name, addr, city) values (:name, :addr, :city)
");
$STH->execute((array)$cathy);
```

Преобразование объекта в массив при execute() приводит к тому, что свойства считаются ключами массива.

Выборка данных

Данные можно получить с помощью метода ->fetch(). Перед его вызовом желательно явно указать, в каком виде они вам требуются. Есть несколько вариантов:

- **PDO::FETCH_ASSOC:** возвращает массив с названиями столбцов в виде ключей
- **PDO::FETCH_BOTH (по умолчанию):** возвращает массив с индексами как в виде названий стобцов, так и их порядковых номеров
- **PDO::FETCH_BOUND:** присваивает значения столбцов соответствующим переменным, заданным с помощью метода ->bindColumn()
- **PDO::FETCH_CLASS:** присваивает значения столбцов соответствующим свойствам указанного класса. Если для какого-то столбца свойства нет, оно будет создано
- **PDO::FETCH_INTO:** обновляет существующий экземпляр указанного класса
- **PDO::FETCH_LAZY:** объединяет в себе PDO::FETCH_BOTH и PDO::FETCH_OBJ
- **PDO::FETCH_NUM:** возвращает массив с ключами в виде порядковых номеров столбцов
- **PDO::FETCH_OBJ:** возвращает анонимный объект со свойствами, соответствующими именам столбцов

На практике вам обычно хватит трех: FETCH_ASSOC, FETCH_CLASS, и FETCH_OBJ. Чтобы задать формат данных, используется следующий синтаксис:

```
$STH->setFetchMode(PDO::FETCH_ASSOC);
```

Также можно задать его напрямую при вызове метода ->fetch().

FETCH_ASSOC

При этом формате создается ассоциативный массив с названиями столбцов в виде индексов. Он должен быть знаком тем, кто использует расширения mysql/mysqli.

```
# поскольку это обычный запрос без placeholder'ов,# можно сразу использовать метод quer
y()
$STH = $DBH->query('SELECT name, addr, city from folks');
  # устанавливаем режим выборки
$STH->setFetchMode(PDO::FETCH_ASSOC);
  while($row = $STH->fetch()) {
    echo $row['name'] . "\n";
    echo $row['addr'] . "\n";
    echo $row['city'] . "\n";
}
```

Цикл while() переберет весь результат запроса.

FETCH_OBJ

Данный тип получения данных создает экземпляр класса std для каждой строки.

```
# создаем запрос
$STH = $DBH->query('SELECT name, addr, city from folks');
# выбираем режим выборки
$STH->setFetchMode(PDO::FETCH_OBJ);
# выводим результатwhile ($row = $STH->fetch()) {
    echo $row->name . "\n";
    echo $row->addr . "\n";
    echo $row->city . "\n";
}
```

FETCH_CLASS

При использовании fetch_class данные заносятся в экземпляры указанного класса. При этом значения назначаются свойствам объекта ДО вызова конструктора. Если свойства с именами, соответствующими названиям столбцов, не существуют, они будут созданы автоматически (с областью видимости public).

Если ваши данные нуждаются в обязательной обработке сразу после их получения из базы данных, ее можно реализовать в конструкторе класса.

Для примера возьмем ситуацию, когда вам нужно скрыть часть адреса проживания человека.

```
class secret_person {
    public $name;
    public $addr;
    public $city;
    public $other_data;

    function __construct($other = '') {
        $this->addr = preg_replace('/[a-z]/', 'x', $this->addr);
        $this->other_data = $other;
    }
}
```

При создании объекта все латинские буквы в нижнем регистре должны замениться на x. Проверим:

```
$STH = $DBH->query('SELECT name, addr, city from folks');
$STH->setFetchMode(PDO::FETCH_CLASS, 'secret_person');
 while($obj = $STH->fetch()) {
    echo $obj->addr;
}
```

Если в базе данных адрес выглядит как ’5 Rosebud’, то на выходе получится ’5 Rxxxxxx’.

Конечно, иногда будет требоваться, чтобы конструктор вызывался ПЕРЕД присваиванием значений. PDO такое тоже позволяет.

```
$STH->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'secret_person');
```

Теперь, когда вы дополнили предыдущий пример дополнительной опцией (PDO::FETCH_PROPS_LATE), адрес видоизменяться не будет, так как после записи значений

ничего не происходит.

Наконец, при необходимости можно передавать конструктору аргументы прямо при создании объекта:

```
$STH->setFetchMode(PDO::FETCH_CLASS, 'secret_person', array('stuff'));
```

Можно даже передавать разные аргументы каждому объекту:

```
$i = 0;  while($rowObj =  $STH->fetch(PDO::FETCH_CLASS, 'secret_person', array($i)))
{
    // что-то делаем
    $i++;
}
```

Другие полезные методы

```
$DBH->lastInsertId();
```

Метод ->lastInsertId() возвращает id последней вставленной записи. Стоит заметить, что он всегда вызывается у объекта базы данных (в статье он именуется $DBH), а не объекта с выражением ($STH).

```
$DBH->exec('DELETE FROM folks WHERE 1');
$DBH->exec("SET time_zone = '-8:00'");
```

Метод ->exec() используется для операций, которые не возвращают никаких данных, кроме количества затронутых ими записей.

```
$safe = $DBH->quote($unsafe);
```

Метод ->quote() ставит кавычки в строковых данных таким образом, что их становится безопасно использовать в запросах. Пригодится, если вы не используете prepared statements.

```
$rows_affected = $STH->rowCount();
```

Метод ->rowCount() возвращает количество записей, которые поучаствовали в операции.
К сожалению, эта функция отказывалась работать с SELECT-запросами вплоть до PHP 5.1.6. Если обновить версию PHP не представляется возможным, количество записей можно получить так:

```
$sql = "SELECT COUNT(*) FROM folks";  if ($STH = $DBH->query($sql)) {
    # проверяем количество записей
    if ($STH->fetchColumn() > 0) {
            # делаем здесь полноценную выборку, потому что данные найдены!
    }
    else {
            # выводим сообщение о том, что удовлетворяющих запросу данных не найдено
    }
}
```