Capítulo 10. Zend_Db

Traduzido por Flávio Gomes da Silva Lisboa (versão 1.5 do Zend Framework)

Sumário

10.1. Zend_Db_Adapter

Zend_Db e suas classes relacionadas provêem uma interface de banco de dados SQL simples para Zend Framework. Zend_Db_Adapter é a classe básica que você usa para conectar sua aplicação PHP a um RDBMS[1]. Há diferentes classes adaptadoras para cada marca de RDBMS.

Os adaptadores Zend_Db criam uma ponte das extensões PHP específicas de um vendedor para uma interface comum, de modo a ajudar você a escrever aplicações PHP uma vez e distribuí-las com múltiplas marcas de RDBMS com pouco esforço.

A interface da classe adaptadora é similar a interface da extensão PHP Data Objects. Zend_Db provê classes adaptadoras de drivers PDO para as seguintes marcas de RDBMS:

- IBM DB2 e Informix Dynamic Server (IDS), usando a extensão PHP pdo_ibm
- MySQL, usando a extensão PHP pdo_mysql
- Microsoft SQL Server, usando a extensão PHP pdo_mssql
- Oracle, usando a extensão PHP pdo_oci
- PostgreSQL, usando a extensão PHP pdo_pgsql
- SQLite, usando a extensão PHP pdo_sqlite

Em adição, Zend_Db provê classes adaptadoras que utilizam extensões de bancos de dados PHP para as seguintes marcas de RDBMS:

- MySQL, usando a extensão PHP mysqli
- Oracle, usando a extensão PHP oci8
- IBM DB2, usando a extensão PHP ibm_db2
- Firebird/Interbase, usando a extensãp PHP php_interbase

Nota

Cada adaptador Zend_Db usa uma extensão PHP. Você deve ter a extensão PHP respectiva habilitada em seu ambiente PHP para usar um adaptador Zend_Db. Por exemplo, se você usar qualquer um dos adaptadores Zend_Db PDO, você precisa habilitar tanto a extensão PDO quanto o driver PDO para a marca de RDBMS que você usa.

1 Relational Database Management System: Sistema Gerenciador de Banco de Dados Relacional

10.1.1. Conectando a um Banco de Dados usando um Adaptador

Esta seção descreve como criar uma instância de um adaptador de banco de dados. Isso corresponde a fazer uma conexão ao seu servidor RDBMS de sua aplicação PHP.

10.1.1.1. Usando um Construtor do Adaptador Zend_Db

Você pode criar uma instância de um adaptador usando seu construtor. Um construtor de adaptador leva um argumento, que é um vetor de parâmetros usados para declarar a conexão

Exemplo 10.1. Usando um Construtor do Adaptador

```
<?php
require_once 'Zend/Db/Adapter/Pdo/Mysql.php';
$db = new Zend_Db_Adapter_Pdo_Mysql(array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

10.1.1.2. Usando Zend_Db Factory

Como uma alternativa a usar o construtor do adaptador diretamente, você pode criar uma instância de um adaptador usando o método estático `Zend_Db::factory()`. Esse método carrega dinamicamente o arquivo da classe adaptadora sob demanda, usando Zend_Loader::loadClass().

O primeiro argumento é um literal que representa o nome base da classe adaptadora. Por exemplo o literal 'Pdo_Mysql' corresponde a classe Zend_Db_Adapter_Pdo_Mysql. O segundo argumento pe o mesmo vetor de parâmetros que você teria dados ao construtor do adaptador.

Exemplo 10.2. Usando o método factory do adaptador

```
<?php
require_once 'Zend/Db.php';
// Carrega automaticamente a classe Zend_Db_Adapter_Pdo_Mysql e cria uma
instância dela.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

Se você criar sua própria classe que estende Zend_Db_Adapter_Abstract, mas você não nomear sua classe com o prefixo de pacote "Zend_Db_Adapter", você pode usar o método `factory()` para carregar seu adaptador se você especificar a parte condutora da classe adaptadora com a chave 'adapterNamespace' no vetor de parâmetros.

Exemplo 10.3. Usando o método factory do adaptador para uma classe adaptadora customizada

```
<?php
require_once 'Zend/Db.php';
// Carrega automaticamente a classe MyProject_Db_Adapter_Pdo_Mysql e cria uma
instância dela.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'             => '127.0.0.1',
    'username'         => 'webuser',
    'password'         => 'xxxxxxxx',
    'dbname'           => 'test',
    'adapterNamespace' => 'MyProject_Db_Adapter'
));
```

10.1.1.3. Usando Zend_Config com Zend_Db Factory

Opcionalmente, você pode especificar um argumento do método `factory()` como um objeto do tipo Zend_Config.

Se o primeiro argumento é um objeto de configuração, é esperado que ele contenha uma propriedade chamada `adapter`, contendo um literal definindo o nome base da classe adaptadora. Opcionalmente, o objeto pode conter uma propriedade chamada `params`, com subpropriedades correspondentes aos nomes de parâmetro do adaptador. Isso é usado somente se o segundo argumento do método `factory()` estiver ausente.

Exemplo 10.4. Usando o método factory do adaptador com um objeto

No exemplo abaixo, um objeto Zend_Config é criado a partir de um vetor. Você pode também carregar dados de um arquivo externo, por exemplo com Zend_Config_Ini ou Zend_Config_Xml.

```
<?php
require_once 'Zend/Config.php';
require_once 'Zend/Db.php';
$config = new Zend_Config(
    array(
        'database' => array(
            'adapter' => 'Mysqli',
            'params' => array(
                'dbname' => 'test',
                'username' => 'webuser',
                'password' => 'secret',
            )
        )
```

```
    )
);
$db = Zend_Db::factory($config->database);
```

O segundo argumento do método `factory()` pode ser um vetor associativo contendo entradas correspondens aos parâmetros do adaptador. Esse argumento é opcional. Se o primeiro argumento é do tipo Zend_Config, é assumido que ele contém todos os parâmetros, e o segundo argumento é ignorado.

10.1.1.4. Parâmetros do Adaptador

A lista abaixo explica parâmetros comuns reconhecidos pelas classes adaptadoras Zend_Db.

- **host**: um literal contendo um nome de hospedeiro ou um endereço IP do servidor de banco de dados. Se o banco de dados está rodando no mesmo hospedeiro que a aplicação PHP, você pode usar 'localhost' ou '127.0.0.1'.
- **username**: identificador de conta para autenticar uma conexão ao servidor RDBMS.
- **password**: credencial de senha da conta para autenticar uma conexão ao servidor RDBMS.
- **dbname**: nome da instância do banco de dados no servidor RDBMS.
- **port**: alguns servidores RDBMS podem aceitar conexões de rede em um número de porta especificado pelo administrador. O parâmetro port permite que você especifique a porta na qual sua aplicação PHP se conecta, de modo a casar com a porta configurada no servidor RDBMS.
- **options**: este parâmetro é um vetor associativo de opções que são genéricas para todas as classes Zend_Db_Adapter.
- **driver_options**: este parâmetro é uma vetor associativo de opções adicionais que são específicas para uma dada extensão de banco de dados. Um uso típico desse parâmetro é confitgurar atributos de um driver PDO.
- **adapterNamespace**: determina a parte inicial do nome da classe para o adaptador, ao invés de 'Zend_Db_Adapter'. Usa isso se você precisa usar o método `factory()` para carregar uma classe adaptadora de banco de dados não-Zend.

Exemplo 10.5. Passando a opção case-folding para o factory

Você pode especificar essa opção pela constante `Zend_Db::CASE_FOLDING`. Isso corresponde ao atributo `ATTR_CASE` nos drivers de banco de dados PDO e IBM DB2, ajustando a caixa de chaves literais em conjuntos de resultado de consultas. A opção leva os valores `Zend_Db::CASE_NATURAL` (o padrão), `Zend_Db::CASE_UPPER`, e `Zend_Db::CASE_LOWER`.

```php
<?php
$options = array(
    Zend_Db::CASE_FOLDING => Zend_Db::CASE_UPPER
);
$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);
$db = Zend_Db::factory('Db2', $params);
```

Exemplo 10.6. Passando a opção auto-quoting para o factory

Você pode especificar essa opção pela constante `Zend_Db::AUTO_QUOTE_IDENTIFIERS`. Se o valor é `true` (o padrão), identificadores como nomes de tabela, nomes de coluna, e mesmo apelidos são delimitados em toda sintaxe SQL gerada pelo objeto adaptador. Isso torna simples usar identificadores que contêm palavras chave, ou caracteres especiais. Se o valor é `false`, identificadores não são delimitador automaticamente. Se você precisar delimitar identificadores, você deve fazer por conta própria uso do método `quoteIdentifier()`.

```php
<?php
$options = array(
    Zend_Db::AUTO_QUOTE_IDENTIFIERS => false
);
$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);
$db = Zend_Db::factory('Pdo_Mysql', $params);
```

Exemplo 10.7. Passando opções de driver PDO para o factory

```php
<?php
$pdoParams = array(
    PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true
);
$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'driver_options' => $pdoParams
);
```

```
$db = Zend_Db::factory('Pdo_Mysql', $params);
echo $db->getConnection()->getAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY);
```

10.1.1.5. Gerenciando Conexões Ociosas

Criar uma instância de uma classe adaptadora não conecta imediatamente ao servidor RDBMS. O adaptador grava os parâmetros da conexão, e faz a conexão atual sob demanda, na primeira vez que você precisa executar uma consulta. Isso garante que a criação do objeto adaptador seja rápida e barata. Você pode criar uma instância de um adaptador mesmo se você não está certo de que precisará rodar quaisquer consultas de banco de dados durante a requisição atual que sua aplicação está servindo.

Se você precisar forçar o adaptador a se conectar ao RDBMS, use o método `getConnection()`. Esse método retorna um objeto para conexão como representado pela extensão de banco de dados PHP respectiva. Por exemplo, se você usar qualquer das classes adaptadoras para drivers PDO, então `getConnection()` retorna o objeto PDO, depois de iniciá-lo como uma conexão ativa para o banco de dados específico.

Pode ser útil forçar a conexão se você quiser capturar quaisquer exceções que forem lançadas como resultado de credenciais de conta inválidas, ou outra falha de conexão ao servidor RDBMS. Essas exceções não são lançadas até a conexão ser feita, assim ele pode ajudar a simplificar o código de sua aplicação se você manipular as exceções em um lugar, ao invés de no momento da primeira consulta contra o banco de dados.

Exemplo 10.8. Manipulando exceções de conexão

```
<?php
try {
    $db = Zend_Db::factory('Pdo_Mysql', $parameters);
    $db->getConnection();
} catch (Zend_Db_Adapter_Exception $e) {
    // talvez uma falha de credencial de login, talvez o RDBMS não esteja
rodando
} catch (Zend_Exception $e) {
    // talvez factory() falhou ao carregar a classe adaptadora especificada
}
```

10.1.2. O banco de dados de exemplo

Na documentação para classes Zend_Db, nós usamos um conjunto de tabelas simples para ilustrar o uso das classes e métodos. Essas tabelas de exemplo poderiam armazenar informações para rastrear bugs em um projeto de desenvolvimento de software. O banco de dados contém quatro tabelas:

- **accounts** armazena informação sobre cada usuário do banco de dados de rastreamento de

bug.

- **products** armazena informação sobre cada produto para o qual um bug pode ser logado.

- **bugs** armazena informação sobre bugs, incluindo o estado atual do bug, a pessoa que reportou o bug, a pessoa que é incumbida de corrigir o bug, e a pessoa que é incumbida de verificar a correção.

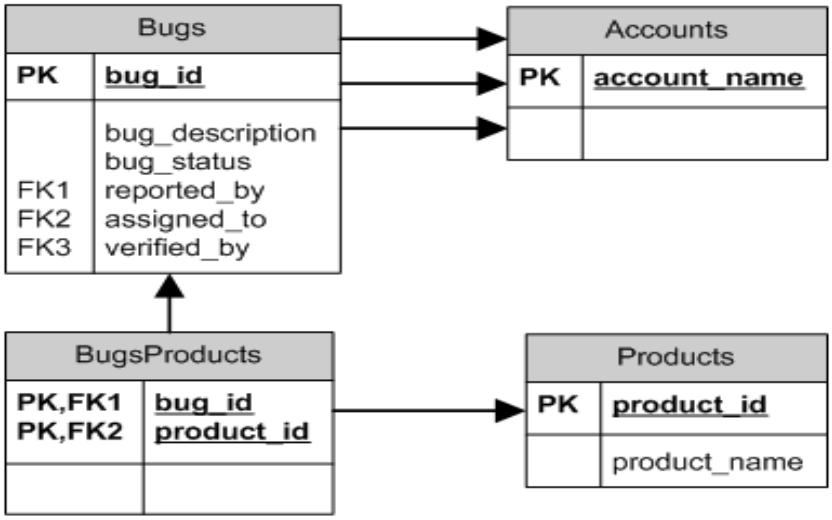
- **bugs_products** armazena um relacionamento entre bugs e produtos. Isso implementa um relacionamento muitos-pra-muitos, porque um dado bug pode ser relevante para múltiplos produtos, e na verdade um dado produto pode ter múltiplos bugs.

O seguinte pseudocódigo de linguagem de definição de dados SQL descreve as tabelas neste exemplo de banco de dados. Essas tabelas de exemplo são usadas extensivamente pelos testes unitários automatizados para Zend_Db.

```
CREATE TABLE accounts (
  account_name      VARCHAR(100) NOT NULL PRIMARY KEY
);
CREATE TABLE products (
  product_id        INTEGER NOT NULL PRIMARY KEY,
  product_name      VARCHAR(100)
);
CREATE TABLE bugs (
  bug_id            INTEGER NOT NULL PRIMARY KEY,
  bug_description   VARCHAR(100),
  bug_status        VARCHAR(20),
  reported_by       VARCHAR(100) REFERENCES accounts(account_name),
  assigned_to       VARCHAR(100) REFERENCES accounts(account_name),
  verified_by       VARCHAR(100) REFERENCES accounts(account_name)
);
CREATE TABLE bugs_products (
  bug_id            INTEGER NOT NULL REFERENCES bugs,
  product_id        INTEGER NOT NULL REFERENCES products,
  PRIMARY KEY       (bug_id, product_id)
);
```

Note também que a tabela `bugs` contém múltiplas referências de chave estrangeira para a tabela `accounts`. Cada uma das chaves estrangeiras pode referenciar uma linha diferente na tabela `accounts` para um dado bug.

O diagram abaixo ilustra o modelo de dados físico do banco de dados de exemplo.



10.1.3. Lendo Resultados de Consulta

Esta seção descreve métodos da classe adaptadora com os quais você pode rodar consultas SELECT e recuperar resultados de consulta.

10.1.3.1. Buscando um Objeto Rowset Completo

Você pode rodar uma consulta SQL SELECT e recuperar seu resultado em um passo usando o método `fetchAll()`.

O primeiro argumento para este método é um literal contendo uma declaração SELECT. De modo alternativo, o primeiro argumento pode ser um objeto da classe Zend_Db_Select. O adaptador automaticamente converte esse objeto para uma representação literal da declaração SELECT.

O segundo argumento para `fetchAll()` é um vetor de valores a serem substituídos pelos espaços reservados para parâmetros na declaração SQL.

Exemplo 10.9. Usando fetchAll()

```
<?php
$sql = 'SELECT * FROM bugs WHERE bug_id = ?';
$result = $db->fetchAll($sql, 2);
```

10.1.3.2. Alterando o Fetch Mode

Por padrão, `fetchAll()` retorna um vetor de linhas, cada uma das quais é um vetor associativo. As chaves do vetor associativo são as colunas ou apelidos de coluna definidos na consulta SELECT.

Você pode especificar um estilo diferente de resultados de busca usando o método

`setFetchMode()`. Os modos suportados são identificados por constantes:

- **Zend_Db::FETCH_ASSOC**: retorna dados em um vetor de vetores associativos. As chaves do vetor são nomes de coluna, como literais. Esse é o modo de busca padrão para classes Zend_Db_Adapter.

  Note que se sua lista de seleção contém mais de uma coluna com o mesmo nome, por exemplo se elas são de duas tabelas diferentes em um JOIN, pode haver somente uma entrada no vetor associativo para o nome dados. Se você usar o modo FETCH_ASSOC, você deve especificar apelidos de coluna em sua consulta SELECT para garantir que os nomes do resultado em chaves de vetor únicas.

  Por padrão, esses literais são retornados do mesmo jeito que o driver do banco de dados retorna. Isso é tipicamente a ortografia da coluna no servidor RDBMS. Você pode especificar a caixa para esses literais, usando a opção `Zend_Db::CASE_FOLDING`. Especifique isso quando instanciar o adaptador. Veja Exemplo 10.5, "Passando a opção case-folding para o factory".

- **Zend_Db::FETCH_NUM**: retorna dados em um vetor de vetores. Os vetores são indexados por inteiros, correspondentes a posição do respectivo campo na lista de seleção da consulta.

- **Zend_Db::FETCH_BOTH**: retorna dados em um vetor de vetores. As chaves do vetor são tanto literais como os usados no modo FETCH_ASSOC, quanto inteiros como os usados no modo FETCH_NUM. Note que o número de elementos no vetor é o dobro do que seria em um vetor se você usasse ou FETCH_ASSOC ou FETCH_NUM.

- **Zend_Db::FETCH_COLUMN**: retorna dados em um vetor de valores. O valor em cada vetor é o valor retornado por uma coluna do conjunto de resultado. Por padrão, essa é a primeira coluna, indexada por 0.

- **Zend_Db::FETCH_OBJ**: retorna dados em um vetor de objetos. A classe padrão é a classe interna PHP stdClass. Colunas do Objeto Rowset ficam disponíveis como propriedades públicas do objeto.

Exemplo 10.10. Usando setFetchMode()

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);
$result = $db->fetchAll('SELECT * FROM bugs WHERE bug_id = ?', 2);
// $result é um veto de objetos
echo $result[0]->bug_description;
```

10.1.3.3. Buscando um Objeto Rowset como um Vetor Associativo

O método `fetchAssoc()` retorna dados em um vetor de vetores associativos, independentemente

de qual valor você tenha configurado para o modo de busca.

Exemple 10.11. Usando fetchAssoc()

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);
$result = $db->fetchAssoc('SELECT * FROM bugs WHERE bug_id = ?', 2);
// $result é um vetor de vetores associativos, apesar do modo de busca
echo $result[0]['bug_description'];
```

10.1.3.4. Buscando uma Coluna Simples de um Objeto Rowset

O método `fetchCol()` retorna dados em um veto de valores, independentemente do valor que você tenha configurado para o modo de busca. Isso somente retorna a primeira coluna devolvida pela consulta. Quaisquer outras colunas retornadas pela consulta são descartadas. Se você precisa retornar uma outra coluna que não seja a primeira, veja Seção 10.2.3.4, “Buscando uma Coluna Simples de um Objeto Rowset”.

Exemplo 10.12. Usando fetchCol()

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);
$result = $db->fetchCol('SELECT bug_description, bug_id FROM bugs WHERE bug_id =
?', 2);
// contém bug_description; bug_id não é devolvido
echo $result[0];
```

10.1.3.5. Buscando Pares Chave-Valor de um Objeto Rowset

O método `fetchPairs()` retorna dados em um vetor de pares chave-valor, como um vetor associativo com uma entrada simples por linha. A chave desse vetor associativo é tomada da primeira coluna retornada pela consulta SELECT. O valor é tomado da segunda coluna retornada pela consulta SELECT. Quaisquer outras colunas retornadas pela consulta são descartadas.

Você deve projetar a consulta SELECT de modo que a primeira coluna retornada tenha valores únicos. Se há valores duplicados na primeira coluna, as entradas no vetor associativo serão sobrescritas.

Exemplo 10.13. Usando fetchPairs()

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);
```

```
$result = $db->fetchPairs('SELECT bug_id, bug_status FROM bugs');
echo $result[2];
```

10.1.3.6. Buscando uma Linha Simples de um Objeto Rowset

O método `fetchRow()` retorna dados usando o modo de busca atual, mas ele retorna somente a primeira linha do Objeto Rowset.

Exemplo 10.14. Usando fetchRow()

```
<?php
$db->setFetchMode(Zend_Db::FETCH_OBJ);
$result = $db->fetchRow('SELECT * FROM bugs WHERE bug_id = 2');
// note que $result é um simples objeto, não um vetor de objetos
echo $result->bug_description;
```

10.1.3.7. Buscando um Escalar Simples de um Objeto Rowset

O método `fetchOne()` é como uma combinação de `fetchRow()` com `fetchCol()`, na qual ele retorna dados somente para a primeira linha do Objeto Rowset, e retorna somente o valor da primeira coluna dessa linha. Portanto ele retorna somente um valor escalar simples, não um vetor ou um objeto.

Exemplo 10.15. Usando fetchOne()

```
<?php
$result = $db->fetchOne('SELECT bug_status FROM bugs WHERE bug_id = 2');
// isto é um valor literal simples
echo $result;
```

10.1.4. Gravando Alterações no Banco de Dados

Você pode usar a classe adaptadora para gravar novos dados ou alterar dados existentes em seu banco de dados. Esta seção descreve métodos para fazer essas operações.

10.1.4.1. Incluindo Dados

Você pode adicionar novas linhas a tabela em seu banco de dados usando o método `insert()`. O

primeiro argumento é um literal que denomina a tabela, e o segundo argumento é um vetor associativo, mapeando os nomes de coluna para valores de dados.

Exemplo 10.16. Incluindo uma tabela

```
<?php
$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
$db->insert('bugs', $data);
```

As colunas que você excluir de um vetor de dados não são especificadas para o banco de dados. Portanto, elas seguirão as mesmas regras qeu uma declaração SQL INSERT segue: se a coluna tem uma cláusula DEFAULT, a coluna leva aquele valor na linha criada, caso contrário a coluna é deixada em um estado NULL.

Por padrão, os valores em seu vetor de dados são inseridos usando parâmetros. Isso reduz o risco de alguns tipos de alertas de segurança. Você não precisa aplicar cancelamento[2] ou citação[3] no vetor de dados.

Você poderia precisar que os valores em seu vetor de dados fossem tratados como expressões SQL, caso no qual elas não seriam citadas. Por padrão, todos os valores de dados passados como literais são tratados como literais. Para especificar que o valor está na expressão SQL e portanto não deve ser citado, passe o valor no vetor de dados como um objeto do tipo Zend_Db_Expr ao invés de um texto simples.

Exemplo 10.17. Incluindo expressões em uma tabela

```
<?php
$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
$db->insert('bugs', $data);
```

10.1.4.2. Recuperando um Valor Gerado

Algumas marcas de RDBMS suportam chaves primárias autoincrementais. Uma tabela definida

2 "cancelamento" foi a forma adotada para o termo "escaping" que significa anular o efeito de comando que caracteres especiais tem em conteúdos literais (strings) com o uso da barra invertida (\).

3 "citação" aqui, tradução do termo "quoting", quer dizer destacado com o uso de caracteres delimitadores, geralmente apóstrofos.

desse modo gera um valor de chave primária automaticamente durante um INSERT de uma nova linha. O valor de retorno do método `insert()` não é o último ID inserido, porque a tabela pode não ter uma coluna autoincremental. Ao invés disso, o valor de retorno é o numero de linhas afetadas (geralmente 1).

Se sua tabela é definida com uma chave primária autoincremental, você pode chamar o método `lastInsertId()` depois da inclusão. Esse método retorna o último valor gerado no escopo da conexão atual de banco de dados.

Exemplo 10.18. Usando lastInsertId() para uma chave autoincremental

```
<?php
$db->insert('bugs', $data);
// retorna o último valor gerado por uma coluna autoincremental.
$id = $db->lastInsertId();
```

Algumas marcas de banco de dados suportam um objeto seqüêncial, que gera valores únicos para servir como chaves primárias. Para suportar seqüências, o método `lastInsertId()` aceita dois argumentos literais opcionais. Esses argumentos nomeiam a tabela e a coluna, assumindo que você tem seguido a convenção de que uma seqüência é nomeada usando os nomes de tabela e coluna para os quais a seqüência gera valores, e um sufixo "_seq". Isso é baseado na convenção usada pelo PostgreSQL quando nomeia seqüências para colunas SERIAL. Por exemplo, uma tabela "bugs" com coluna de chave primária "bug_id" usaria uma seqüência chamada "bugs_bug_id_seq".

Exemplo 10.19. Usando lastInsertId() para uma seqüência

```
<?php
$db->insert('bugs', $data);
// retorna o último valor gerado pela seqüência 'bugs_bug_id_seq'.
$id = $db->lastInsertId('bugs', 'bug_id');
// de forma alternativa, retorna o valor gerado pela seqüência 'bugs_seq'.
$id = $db->lastInsertId('bugs');
```

Se o nome do seu objeto de seqüência não segue essa convenção de nomeação, use em seu lugar o método `lastSequenceId()`. Esse método leva um argumento literal simples, nomeando a seqüência literalmente.

Exemplo 10.20. Usando lastSequenceId()

```
<?php
$db->insert('bugs', $data);
// retorna o último valor gerado pela seqüência 'bugs_id_gen'.
$id = $db->lastSequenceId('bugs_id_gen');
```

Para marcas de RDBMS que não suportam seqüências, incluindo MySQL, Microsoft SQL Server, e SQLite, os argumentos para o método lastInsertId() são ignorados, e o valor retornado é o valor gerado mais recentemente para qualquer tabela por operações INSERT durante a conexão atual. Para essas marcas de RDBMS, o método lastSequenceId sempre retorna `null`.

Por que não usar "SELECT MAX(id) FROM table"?

Algumas vezes essa consulta retorna o valor de chave inserido mais recentemente na tabela. Entretanto, essa técnica não é segura para usar em um ambiente onde múltiplos clientes estão inserindo registros em um banco de dados. É possível, e portanto é possível de acontecer eventualmente, que outro cliente insira outra linha no instante emtre a inserção executada por sua aplicação cliente e sua consulta para o valor de MAX(id). Assim o valor retornado não identifica a linha que você inseriu, identifica a linha inserida por algum outro cliente. Não há modo de saber quando isso aconteceu.

Usar um modo de isolamento de transação forte tal como “repeatable read” pode mitigar esse risco, mas algumas marcas de RDBMS não suportam o isolamento de transação requerido para isso, ou sua aplicação pode usar um isolamento de transação baixo por projeto.

Além disso, usar uma expressão como "MAX(id)+1" para gerar um novo valor para uma chave primária não é seguro, porque dois clientes poderiam fazer essa consulta simultaneamente, e então ambos usam o mesmo valor calculado para sua próxima operação INSERT.

Todas as marcas de RDBMS fornecem mecanismos para gerar valores únicos, e retornam o último valor gerado. Esses mecanismos necessariamente trabalham fora do escopo de isolamento de transação, assim não há chance de dois clientes gerarem o mesmo valor e não há chance que o valor gerado por outro cliente possa ser reportado para sua conexão de cliente como o último valor gerado.

10.1.4.3. Atualizando Dados

Você pode atualizar dados em uma tabela usando o método `update()` de um adaptador. Esse método leva três argumentos: o primeiro é o nome da tabela; o segundo é um vetor associativo que mapeia colunas a serem alteradas para novos valores para associar a essas colunas.

Os valores no vetor de dados são tratados como literais. Veja Seção 10.1.4.1, “Incluindo Dados” para informações sobre como usar expressões SQL no vetor de dados.

O terceiro argumento é um literal, contendo uma expressão SQL que é usada como critério para as linhas a serem alteradas. Os valores e identificadores neste argumento não são citados nem sofrem cancelamento. Você é responsável por garantir que qualquer conteúdo dinâmico seja interpolado nesse literal de forma segura. Veja Seção 10.1.5, “Citando Valores e Identificadores” para métodos que ajudam você a fazer isso.

O valor de retorno é o número de linhas afetadas pela operação de atualização.

Exemplo 10.21. Atualizando linhas

```php
<?php
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);
$n = $db->update('bugs', $data, 'bug_id = 2');
```

Se você omitir o terceiro argumento, então todas as linhas na tabela do banco de dados são atualizadas com os valores especificados no vetor de dados.

Se você fornecer um vetor de literais como terceiro argumento, esses literais serão unidos como termos em uma expressão separada por operadores `AND`.

Exemplo 10.22. Atualizando linhas usando um vetor de expressões

```php
<?php
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);
$where[] = "reported_by = 'goofy'";
$where[] = "bug_status = 'OPEN'";
$n = $db->update('bugs', $data, $where);
// A SQL resultante é:
//  UPDATE "bugs" SET "update_on" = '2007-03-23', "bug_status" = 'FIXED'
//  WHERE ("reported_by" = 'goofy') AND ("bug_status" = 'OPEN')
```

10.1.4.4. Apagando Dados

Você pode apagar linhas de uma tabela de banco de dados usando o método `delete()`. Esse método leva dois argumentos: o primeiro é um literal que nomeia a tabela.

O segundo argumento é um literal contendo uma expressão SQL que é usada como critério para as linhas a serem apagadas. Os valores e identificadores nesse argumento não são citados nem sofrem cancelamento. Você é responsável por garantir que qualquer contéudo dinâmico seja interpolado nesse literal de modo seguro. Veja Seção 10.1.5, “Colando Valores e Identificadores Citados” para métodos que ajudam você a fazer isso.

O valor de retorno é o número de linhas afetadas pela operação de apagamento.

Exemplo 10.23. Apagando linhas

```php
<?php
$n = $db->delete('bugs', 'bug_id = 3');
```

Se você omitir o segundo argumento, o resultado é que todas as linhas no banco de dados são apagadas.

Se você fornecer um vetor de literais como segundo argumento, esses literais serão unidos como termos em uma expressão separada por operadores `AND`.

10.1.5. Citando Valores e Identificadores

Quando você forma consultas SQL, freqüentemente ocorre o caso em que você precisa incluir os valores de variáveis PHP em expressões SQL. Isso é um risco, porque se o valor no literal PHP contiver certos símbolos, tais como o símbolo de apóstrofo, isso poderá resultar em uma SQL inválida. Por exemplo, observe o desbalanceamento de caracteres de apóstrofo na seguinte consulta:

```
$name = "O'Reilly";
$sql = "SELECT * FROM bugs WHERE reported_by = '$name'";
echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O'Reilly'
```

O pior é o risco de que tais equívocos de código possam ser explorados deliberadamente por uma pessoa que está tentando manipular a função de sua aplicação web. Se eles podem especificar o valor de uma variável PHP através do uso de uma parâmetro HTTP ou outro mecanismo, eles podem ser capaz de fazer suas consultas SQL fazerem coisas que você não pretendia para elas, tais como retornar dados para pessoas as quais não devem ter privilégio de leitura. Essa é uma técnica séria e muito difundida, conhecida como "SQL Injection" (veja http://en.wikipedia.org/wiki/SQL_Injection).

A classe Zend_Db Adapter provê funções convenientes para ajudar você a reduzir vulnerabilidades a ataques SQL Injection em seu código PHP. A solução é aplicar cancelamento em caracteres especiais tais como apóstrofos em valores PHP antes que eles sejam interpolados em seus literais SQL. Isso protege contra manipulação acidental e deliberada de literais SQL por variáveis PHP que contém caracteres especiais.

10.1.5.1. Usando `quote()`

O método `quote()` aceita um simples argumento, um valor literal escalar. Ele retorna o valor com caracteres especiais em escaping de uma forma apropriada para o RDBMS que você está usando, e cercado por delimitadores de valor literal. O delimitador de valor literal padrão SQL é o apóstrofo (`'`).

Exemplo 10.24. Usando quote()

```
<?php
$name = $db->quote("O'Reilly");
echo $name;
// 'O\'Reilly'
$sql = "SELECT * FROM bugs WHERE reported_by = $name";
```

```
echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

Note que o valor de retorno de `quote()` inclui os delimitadores de apóstrofo cercando o literal. Isso é diferente de algumas funções que realizam escaping de caracteres especiais mas não adicionam os delimitadores de apóstrofo, por exemplo mysql_real_escape_string().

Valores podem precisar ser cercados ou não por apóstrofos, de acordo com o contexto de tipo de dados SQL no qual eles são usados. Por instância, em algumas marcas de RDBMS, um valor inteiro não deve ser cercado por apóstrofos como um literal se ele é comparado com uma coluna ou expressão do tipo inteiro. Em outras palavras, a linha a seguir é um erro em algumas implementações de SQL, assumindo que `intColumn` tenha um tipo de dados `INTEGER`.

```
SELECT * FROM atable WHERE intColumn = '123'
```

Você pode usar o segundo argumento opcional do método `quote()` para aplicar apóstrofos seletivamente para o tipo de dados SQL que você especificar.

Exemplo 10.25. Usando quote() com um tipo SQL

```
<?php
$value = '1234';
$sql = 'SELECT * FROM atable WHERE intColumn = '
     . $db->quote($value, 'INTEGER');
```

Cada classe Zend_Db_Adapter codificou os nomes de tipos de dados SQL numéricos para a respectiva marca de RDBMS. Você pode também usar as constantes `Zend_Db::INT_TYPE`, `Zend_Db::BIGINT_TYPE`, e `Zend_Db::FLOAT_TYPE` para escrever código em modo mais independente de RDBMS.

Zend_Db_Table especifica tipos SQL para `quote()` automaticamente quando gera consultas SQL que fazem referência a colunas chave de tabela.

10.1.5.2. Usando `quoteInto()`

O uso mais típico de apóstrofos é interpolar uma variável PHP em uma expressão ou declaração SQL. Você pode usar o método `quoteInto()` para fazer isso em um passo. Esse método leva dois argumentos: o primeiro argumento é um literal contendo um símbolo de espaço reservado para parâmetro (?), e o segundo argumento é o valor ou variável PHP que deve ser substituída pelo símbolo de espaço reservado.

O símbolo de espaço reservado é o mesmo símbolo usado por muitas marcas de RDBMS para

parâmetros posicionais, mas o método `quoteInto()` só emula parâmetros de consulta. O métodos simplesmente interpola o valor em um literal, aplica cancelamento a caracteres especiais, e aplica apóstrofos ao redor do mesmo. Parâmetros de consulta verdadeiros mantém a separação entre o literal SQL e os parâmetros como a declaração é analisada gramaticalmente no servidor RDBMS.

Exemplo 10.26. Usando quoteInto()

```
<?php
$sql = $db->quoteInto("SELECT * FROM bugs WHERE reported_by = ?", "O'Reilly");
echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

Você pode usar o terceiro parâmetro opcional de `quoteInto()` para especificar o tipo de dados SQL. Tipos de dados numéricos não são cercados por apóstrofos, e outros tipos são cercados.

Exemplo 10.27. Usando quoteInto() com um tipo SQL

```
<?php
$sql = $db->quoteInto("SELECT * FROM bugs WHERE bug_id = ?", '1234', 'INTEGER');
echo $sql;
// SELECT * FROM bugs WHERE reported_by = 1234
```

10.1.5.3. Usando `quoteIdentifier()`

Valores não são a única parte da sintaxe SQL que pode precisar ser variável. Se você usa variáveis PHP para nomear tabelas, colunas, ou outros identificadores em suas declarações SQL, você pode precisar colocar apóstrofos nesses literais também. Por padrão, identificadores SL tem regras de sintaxe iguais ao PHP e a maioria das linguagens de programação. Por exemplo, identificadores não devem conter espaços, certos caracteres especiais ou pontuação, ou caracteres internacionais. Também certas palavras são reservadas para a sintaxe SQL, e não devem ser usadas como identificadores.

Entretanto, SQL tem uma característica chamada *identificadores delimitados*, que permite amplas escolhas para a grafia de identificadores. Se você envolver um identificador SQÇ nos tipos apropriados de apóstrofos, você pode usar identificadores com grafiass que deveriam ser inválidas sem os apóstrofos. Identificadores delimitados podem conter espaços, pontuação, ou caracteres internacionais. Você também pode usar palavras reservadas SQL se quiser envolvê-los.

O método `quoteIdentifier()` trabalha com `quote()`, mas aplica o delimitador de identificação de caracteres para o literal de acordo com o tipo de adaptador. Por exemplo, SQL padrão usa aspas (“) para delimitadores de identificador, e a maioria das marcas de RDBMS usam esse símbolo. MySQL usa crase (`) por padrão. O método `quoteIdentifier()` também aplica escaping de caracteres especiais de dentro do argumento literal.

Exemple 10.28. Usando quoteIdentifier()

```
<?php
// nós podemos ter uma tabela cujo nome é uma palavra-reservada SQL
$tableName = $db->quoteIdentifier("order");
$sql = "SELECT * FROM $tableName";
echo $sql
// SELECT * FROM "order"
```

Identificadores delimitados SQL são sensíveis a caixa, exceto identificadores não cercados por apóstrofos. Portanto, se você usa identificadores delimitados, você deve usar a grafia do identificador exatamente como está armazenado em seu esquema, incluindo a caixa das letras.

Na maioria dos casos onde SQL é gerado de dentro das classes Zend_Db, padrão é que todos os identificadores sejam delimitados automaticamente. Você pode mudar esse comportamento com a opção `Zend_Db::AUTO_QUOTE_IDENTIFIERS`. Especifique isso quando instanciar o adaptador. Veja Exemplo 10.6, “Passando a opção auto-quoting para o factory”.

10.1.6. Controlando Transações de Banco de Dados

Bancos de dados definem transações como unidades lógicas de trabalho que podem ser submetidas ou descartadas como uma alteração simples, mesmo que elas operem sobre múltiplas tabelas. Todas as consultas a um banco de dados são executadas dentro do contexto de uma transação, mesmo se o driver do banco de dados os gerencia implicitamente. Isso é chamado modo de *auto-commit*, no qual o driver de banco de dados cria uma transação para cada declaração que você executa, e efetiva essa transação depois que sua declaração SQL tenha sido executada. Por padrão, todas as classes adaptadoras operam em modo de auto-commit.

Alternativamente, você pode especificar o início e a resolução de uma transação, e assim controlar quantas consultas SQL são incluídas em um grupo simples que é efetivado (ou descartado) como uma operação simples. Use o método `beginTransaction()` para iniciar uma transação. Declarações SQL subseqüentes serão executadas no contexto da mesma transação até que você as resolva explicitamente.

Para resolver a transação, use o método `commit()` ou `rollBack()`. O método `commit()` marca alterações feitas enquanto sua transação é efetivada, o que significa que os efeitos dessas alterações serão exibidos em consultas rodando em outras transações.

O método `rollBack()` faz o oposto: ele descarta as alterações feitas durante sua transação. As mudanças são efetivamente desfeitas, e o estado dos dados retorna a como ele era antes de você começar sua transação. Entretanto, descartar sua transação não tem efeito sobre mudanças feitas por outras transações rodando concorrentemente.

Depois que você resolve essa transação, `Zend_Db_Adapter` retorna ao modo auto-commit até você chamar `beginTransaction()` novamente.

Exemplo 10.29. Gerenciando uma transação para garantir consistência

```
<?php
// Inicie uma transação explicitamente.
$db->beginTransaction();
try {
    // Tentativa de executar uma ou mais consultas:
    $db->query(...);
    $db->query(...);
    $db->query(...);
    // Se tudo deu certo, efetiva a transação e todas as mudanças
    // são efetivadas de uma vez.
    $db->commit();
} catch (Exception $e) {
    // Se qualquer uma das consultas falhar e lançar uma exceção,
    // nós queremos descartar a transação inteira, revertendo
    // as mudanças feitas na transação, mesmo que essas tenham dado certo.
    // Assim, todas as mudanças são efetivadas juntas, ou nenhuma é efetivada.
    $db->rollBack();
    echo $e->getMessage();
}
```

10.1.7. Listando e Descrevendo Tabelas

O método `listTables()` retorna um vetor de literais, denominando todas as tabelas no banco de dados atual.

O método `describeTable()` retorna uma vetor associativo de metadados sobre uma tabela. Especifique o nome da tabela como um literal no primeiro argumento para esse método. O segundo argumento é opcional, e nomeia o esquema no qual a tabela existe.

As chaves do vetor associativo retornado são as colunas da tabela. Os valores correspondentes para cada coluna também é um vetor associativo, com as seguintes chaves e valores:

Tabela 10.1. Campos de metadados retornados por describeTable()

Chave	Tipo	Descrição
SCHEMA_NAME	(string)	Nome do esquema de banco de dados no qual a tabela existe.
TABLE_NAME	(string)	Nome da tabela para o qual essa coluna pertence
COLUMN_NAME	(string)	Nome da coluna.
COLUMN_POSITION	(integer)	Posição ordinal da coluna na tabela.
DATA_TYPE	(string)	Nome que o RDBMS dá para o tipo de dados da coluna
DEFAULT	(string)	Valor padrão para a coluna, se houver.
NULLABLE	(boolean	True se a coluna aceita SQL NULLs, e false se a coluna tem uma

Chave	Tipo	Descrição
	)	restrição NOT NULL.
LENGTH	(integer)	Comprimento ou tamanho da coluna como reportado pelo RDBMS.
SCALE	(integer)	Escala do tipo SQL NUMERIC ou DECIMAL.
PRECISION	(integer)	Precisão do tipo SQL NUMERIC ou DECIMAL.
UNSIGNED	(boolean)	True se um tipo baseado em inteiro é reportado como UNSIGNED.
PRIMARY	(boolean)	True se a coluna é parte da chave primária dessa tabela.
PRIMARY_POSITION	(integer)	Posição ordinal (baseada em 1) da coluna na chave primária.
IDENTITY	(boolean)	True se a coluna usa um valor autogerado.

Se não existe tabela que coincida com nome de tabela e com o nome de esquema opcional especificado, então `describeTable()` retorna um vetor vazio.

10.1.8. Fechando um Conexão

Normalmente não é necessário fechar uma conexão de banco de dados. PHP automaticamente elimina todos os recursos e finaliza uma requisição. Extensões de bancos de dados são desenhadas para fechar a conexão assim que a referência para o objeto de recurso seja eliminada.

Entretanto, se você tiver um script PHP de longa duração que inicia muitas conexões de banco de dados, você pode precisar fechar a conexão, para evitar a exaustão da capacidade de seu servidor RDBMS. Você pode usar o método `closeConnection()` do adaptador para fechar explicitamente a conexão de banco de dados subjacente.

Exemplo 10.30. Fechando uma conexão de banco de dados

```
<?php
$db->closeConnection();
```

Zend_Db suporta conexões persistentes?

O uso de conexões persistentes não é suportado ou mesmo encorajado em Zend_Db.

Usar conexões persistentes pode causar um excesso de conexões no servidor RDBMS, o que causa mais problemas do que qualquer ganho de performance que você possa ter adquirido pela redução da sobrecarga de criar conexões.

Conexões de banco de dados tem estado. Isto é, alguns objetos no servidor RDBMS existem no escopo da sessão. Exemplos são bloqueios, variáveis de usuário, tabelas temporárias, e informação sobre as consultas executadas mais recentemente, tais como linhas afetadas, e a último valor de id gerado. Se você usar conexões persistentes, sua aplicação poderia acessar dados inválidos ou privilegiados que foram criados em uma requisição PHP anterior.

10.1.9. Rodando Outras Declarações de Banco de Dados

Pode haver casos em que você precise acessar o objeto de conexão diretamente, como fornecido pela extensão de banco de dados PHP. Algumas dessas extensões podem oferecer características que não são cobertas por métodos da classe Zend_Db_Adapter_Abstract.

Por exemplo, todas as declarações SQL que rodam por Zend_Db são preparadas, então executadas. Entretanto, algumas características de bancos de dados são incompatíveis com declarações preparadas. Declarações DDL como CREATE e ALTER não podem ser preparados em MySQL. Declarações SQL também não se beneficiam do MySQL Query Cache, anterior ao MySQL 5.1.17.

A maioria das extensões de bancos de dados fornecem um método para executar declarações SQL sem a preparação dos mesmos. Por exemplo, em PDO, esse método é `exec()`. Você pode acessar o objeto de conexão na extensão PHP diretamente usando getConnection().

Exemplo 10.31. Rodando uma declaração não-preparada em um adaptador PDO

```
<?php
$result = $db->getConnection()->exec('DROP TABLE bugs');
```

De modo similar, você pode acessar outros métodos ou propriedades que são específicas para extensões de bancos de dados PHP. Esteja atento, entretanto, que ao fazer isso você pode restringir sua aplicação à interface fornecida pela extensão para uma marca específica de RDBMS.

Em versões futuras de Zend_Db, haverá oportunidades de adicionar pontos de entrada de método para funcionalidades que são comuns às extensões de banco de dados suportadas pelo PHP. Isso não afetará a compatibilidade retrógrada.

10.1.10. Notas sobre Adaptadores Específicos

Esta seção lista diferenças entre as classes adaptadoras com as quais você deve ficar atento.

10.1.10.1. IBM DB2

- Especique esse adaptador pelo método factory() com o nome 'Db2'.
- Esse adaptador usa a extensão PHP ibm_db2.

- IBM DB2 suporta tanto chaves seqüenciais quando autoincrementais. Portanto os argumentos para o `lastInsertId()` são opcionais. Se você não der nenhum argumento, o adaptador retorna o último valor gerado para uma chave autoincremental. Se você der argumentos, o adaptador retorna o último valor gerado pela seqüencia nomeada de acordo com a convenção '*table_column*_seq'.

10.1.10.2. MySQLi

- Especifique esse adaptador pelo método `factory()` com o nome 'Mysqli'.
- Esse adaptador utiliza a extensão PHP mysqli.
- MySQL não suporta seqüencias, assim `lastInsertId()` ignorada seus argumentos e sempre retorna o último valor gerado por uma chave de autoincremento. O método `lastSequenceId()` retorna `null`.

10.1.10.3. Oracle

- Especifique esse adaptador pelo método `factory()` com o nome 'Oracle'.
- Esse adaptador usa a extensão PHP oci8.
- Oracle não suporta chaves autoincrementais, assim você deve especificar o nome de uma seqüencia para `lastInsertId()` ou `lastSequenceId()`.
- A extensão Oracle nã suporta parâmetros posicionais. Você deve usar parâmetros nomeados.
- Atualmente a opção `Zend_Db::CASE_FOLDING` não é suportada pelo adaptador Oracle. Para usar esta opção com Oracle, você deve usar o adaptador PDO OCI.

10.1.10.4. PDO para IBM DB2 e Informix Dynamic Server (IDS)

- Especifique esse adaptador para o método `factory()` com o nome 'Pdo_Ibm'.
- Esse adaptador usa as extensões PHP pdo e pdo_ibm.
- Você deve usar pelo menos a extensão PDO_IBM versão 1.2.2. Se você tem um versão mais recente dessa extensão, você deve atualizar a extensão PDO_IBM da PECL.

10.1.10.5. PDO Microsoft SQL Server

- Especifique esse adaptador para o método `factory()` com o nome 'Pdo_Mssql'.
- Esse adaptador usa as extensões PHP pdo e pdo_mssql.
- Microsoft SQL Server não suporta seqüencias, assim `lastInsertId()` ignora seus

argumentos e sempre retorna o último valor gerado para uma chave autoincremental. O método `lastSequenceId()` retornal `null`.

- Zend_Db_Adapter_Pdo_Mssql configura `QUOTED_IDENTIFIER ON` imediatamente depois de conectar em um banco de dados SQL Server. Ela faz o driver usar o símbolo delimitador identificador padrão SQL (") ao invés da sintaxe proprietária square-brackets (colchetes) que o SQL Server usa para identificadores de delimitação.

- Você pode especificar `pdoType` como uma chave no vetor de opções. O valor pode ser "mssql" (o padrão), "dblib", "freetds", ou "sybase". Essa opção afeta o prefixo DSN que o adaptador usa quando constrói o literal DSN. Tanto "freetds" quanto "sybase" implicam em um prefixo de "sybase:", o qual é usado para o conjunto de bibliotecas FreeTDS. Veja também http://www.php.net/manual/en/ref.pdo-dblib.connection.php para mais informações sobre os prefixos DSN usados nesse driver.

10.1.10.6. PDO MySQL

- Especifique esse adaptador para o método `factory()` com o nome 'Pdo_Mysql'.

- Esse adaptador usa as extensões PHP pdo e pdo_mysql.

- MySQL não suporta seqëncias, assim `lastInsertId()` ignora seus argumentos e sempre retorna o último valor gerado para uma chave autoincremental. O método `lastSequenceId()` retorna `null`.

10.1.10.7. PDO Oracle

- Especifique esse adaptador para o método `factory()` com o nome 'Pdo_Oci'.

- Esse adaptador usa as extensões PHP pdo e pdo_oci.

- Oracle não suporta chaves autoincrementais, assim você deve especificar o nome de uma seqëncia para `lastInsertId()` ou `lastSequenceId()`.

10.1.10.8. PDO PostgreSQL

- Especifique esse adaptador para o metódo `factory()` com o nome 'Pdo_Pgsql'.

- Esse adaptador usa as extensões PHP pdo e pdo_pgsql.

- PostgreSQL suporta tanto seqüencias quanto chaves autoincrementais. Portanto os argumentos para `lastInsertId()` são opcionais. Se você não fornecer argumentos, o adaptador retorna o último valor gerador para uma chave autoincremental. Se você fornecer argumentos, o adaptador retornará o último valor gerado pela seqëncia nomeada de acordo com a convenção '*table_column*_seq'.

10.1.10.9. PDO SQLite

- Especifique esse adaptador para o método `factory()` com o nome 'Pdo_Sqlite'.
- Esse adaptador usa as extensões PHP pdo e pdo_sqlite.
- SQLite não suporta seqüencias, assim `lastInsertId()` ignora seus argumentos e sempre retorna o último valor gerado para uma chave autoincremental. O método `lastSequenceId()` retorna `null`.
- Para conectar ao banco de dados SQLite2, especifique `'dsnprefix'=>'sqlite2'` no vetor de parâmetros quando criar uma instância do adaptador Pdo_Sqlite.
- Para conectar-se a um banco de dados SQLite em memória, especifique `'dbname'=>':memory:'` no vetor de parâmetros quando criar uma instância do adaptador Pdo_Sqlite.
- Versões anteriores do driver SQLite para PHP não parecem suportar os comandos PRAGMA[4] necessários para garantir que nome de coluna curos serão usados nos conjuntos de resultado. Se você tiver problemas em que seus conjuntos de resultado são devolvidos com chaves do formulário "tablename.columnname" quando você faz uma consulta de junção, então você deve atualizar para a versão atual do PHP.

10.1.10.10. Firebird/Interbase

- Esse adaptador usa a extensão PHP php_interbase.
- Firebird/interbase não suporta chaves autoincrementais, assim você deve especificar o nome da seqüencia para `lastInsertId()` ou `lastSequenceId()`.
- Atualmente a opção `Zend_Db::CASE_FOLDING` não é suportada pelo adaptador Firebird/interbase. Identificadores sem apóstrofos são automaticamente devolvidos em caixa alta.

10.2. Zend_Db_Statement

Em adição a métodos convenientes tais como `fetchAll()` e `insert()` documentados em Seção 10.1, “Zend_Db_Adapter”, você pode usar um objeto de declaração para ganhar mais opções para rodar consultas e encontrar conjuntos de resultado. Essa seção descreve como obter uma instância de um objeto de declaração, e como usar seus métodos.

Zend_Db_Statement é baseado no objeto PDOStatement object da extensão PHP Data Objects.

4 Declaração SQL especial usada para modificar a operação da biblioteca SQLite ou para consultar a biblioteca para dados internos que não sejam de tabelas.

10.2.1. Criando uma Declaração

Tipicamente um objeto de declaração é devolvido pelo método `query()` da classe adaptadora. Esse método é um modo geral para preparar qualquer declaração SQL. O primeiro argumento é um literal contendo uma declaração SQL. O segundo argumento opcional é um vetor de valores para ser combinado aos espaços reservados para parâmetros no literal SQL.

Exemplo 10.32. Criando um objeto declaração SQL com query()

```
<?php
$stmt = $db->query('SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?
',
                array('goofy', 'FIXED'));
```

O objeto declaração corresponde à declaração SQL que foi preparada, e executada uma vez com os valores associados especificados. Se a declaração foi uma consulta SELECT ou outro tipo de declaração que retorne um Objeto Rowset, ele ela estará pronta agora para buscar resultados.

Você pode criar uma declaração com seu construtor, mas isso é o uso menos típico. Não há método factory para criar este objeto, assim você precisa carregar a classe de declaração específica e chamar seu construtor. Passe o objeto adaptador como primeiro argumento, e um literal contendo uma declaração SQL como segundo argumento. A declaração é preparada, mas não executada.

Exemplo 10.33. Usando um construtor de declaração SQL

```
<?php
require_once 'Zend/Db/Statement/Mysqli.php';
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';
$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
```

10.2.2. Executando uma Declaração

Você precisa executar um objeto de declaração se você criá-lo usando seu construtor, ou se você quiser executar o mesmo argumento múltiplas vezes. Use o método `execute()` do objeto de declaração. O argumento simples é um vetor de valores a ser combinado aos espaços reservados para parâmetros na declaração.

Se você usar *parâmetros posicionais*, ou aqueles que são marcados com um símbolo marcador de interrogação (`?`), passe os valores de combinação em um vetor simples.

Exemplo 10.34. Executando uma declaração com parâmetros posicionais

```php
<?php
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';
$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
$stmt->execute(array('goofy', 'FIXED'));
```

Se você usar *parâmetros nomeados*, ou aqueles que são indicados por um identificador literal precedido por um caráter de dois pontos (:), passe os valores de combinação em um vetor associativo. As chaves deste vetor devem casar com os nomes de parâmetro.

Exemplo 10.35. Executando uma declaração com parâmetros nomeados

```php
<?php
$sql = 'SELECT * FROM bugs WHERE reported_by = :reporter AND bug_status = :statu
s';
$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
$stmt->execute(array(':reporter' => 'goofy', ':status' => 'FIXED'));
```

Declarações PDO suportam tanto parâmetros posicionais quanto parâmetros nomeados, mas não ambos os tipos em uma declaração SQL simples. Algumas das classes Zend_Db_Statement para extensões não-PDO podem suportar somente um tipo de parâmetro ou outro.

10.2.3. Buscando Resultados de uma Declaração `SELECT`

Você pode chamar métodos a partir do objeto de declaração para recuperar linhas das declarações SQL que produzem conjuntos de resultados. SELECT, SHOW, DESCRIBE e EXPLAIN são exemplos de declarações que produzem um Objeto Rowset. INSERT, UPDATE, e DELETE são exemplos de declarações que não produzem um Objeto Rowset. Você pode executar as últimas declarações SQL que usaram Zend_Db_Statement, mas você não pode chamar métodos para buscar linhas de resultados das primeiras.

10.2.3.1. Buscando uma Linha Simples de um Objeto Rowset

Para recuperar uma linha do Objeto Rowset, use o método `fetch()` do objeto de declaração. Todos os três argumetnos desse método são opcionais:

- **Fetch style** é o primeiro argumento. Ele controla a estrutura na qual a linha é devolvida. Veja Seção 10.1.3.2, “Alterando o Fetch Mode” para uma descrição dos valores válidos e os correspondentes formatos de dados.

- **Cursor orientation** é o segundo argumento. O padrão é Zend_Db::FETCH_ORI_NEXT, que simplesmente significa que cada chamada a `fetch()` retorna a próxima linha no Objeto Rowset, na ordem retornada pelo RDBMS.

- **Offset** é o terceiro argumento. Se a orientação do cursor for Zend_Db::FETCH_ORI_ABS, então o número de deslocamento é o número ordinal da linha a ser retornada. Se a orientação do cursos for Zend_Db::FETCH_ORI_REL, então o número de deslocamento é relativo à posição do cursor antes de `fetch()` ser chamado.

`fetch()` retorna `false` se todas as linhas do Objeto Rowset forem localizadas.

Exemplo 10.36. Usando fetch() em um loop

```
<?php
$stmt = $db->query('SELECT * FROM bugs');
while ($row = $stmt->fetch()) {
    echo $row['bug_description'];
}
```

Veja também PDOStatement::fetch().

10.2.3.2. Buscando um Objeto Rowset Completo

Para recuperar todas as linhas de cum Objeto Rowset em um passo, use o método `fetchAll()`. Isso é equivalente a chamar o método `fetch()` em um loop e retornar todas as linhas em um vetor. O método `fetchAll()` aceita dois argumentos. O primeiro é o estilo de busca, como descrito acima, e o segundo indica o número de colunas a serem retornadas, quando o estilo de busca é Zend_Db::FETCH_COLUMN.

Exemplo 10.37. Usando fetchAll()

```
<?php
$stmt = $db->query('SELECT * FROM bugs');
$rows = $stmt->fetchAll();
echo $rows[0]['bug_description'];
```

Veja também PDOStatement::fetchAll().

10.2.3.3. Alterando o Fetch Mode

Por padrão, o objeto de declaração retorna linhas do Objeto Rowset como vetores associativos, que mapeiam nomes colunas para valores de colunas. Você pode especificar um formato diferente para a classe de declaração retornar linhas, exatamente como você pode fazer na classe adaptadora. Você pode usar o método `setFetchMode()` do objeto de declaração para especificar o fetch mode. Espeficique o fetch mode usando as constantes da classe Zend_Db FETCH_ASSOC, FETCH_NUM, FETCH_BOTH, FETCH_COLUMN, e FETCH_OBJ. Veja a Seção 10.1.3.2,

"Alterando o Fetch Mode" para mais informações sobre esses modos. Chamadas subseqüentes para os métodos de declaração `fetch()` ou `fetchAll()` usam o fetch mode que você especificar.

Exemplo 10.38. Configurando o fetch mode

```
<?php
$stmt = $db->query('SELECT * FROM bugs');
$stmt->setFetchMode(Zend_Db::FETCH_NUM);
$rows = $stmt->fetchAll();
echo $rows[0][0];
```

Veja também PDOStatement::setFetchMode().

10.2.3.4. Buscando uma Coluna Simples de um Objeto Rowset

Para retornar uma coluna simples da próxima linha de um Objeto Rowset, use `fetchColumn()`. O argumento opcional é o índice inteiro da coluna, e seu padrão é 0. Esse método retorna um valor escalar, ou `false` se todas as linhas do Objeto Rowset tiverem sido buscadas.

Note que esse método opera de forma diferente do método`fetchCol()` da classe adaptadora. O método `fetchColumn()` de uma declaração retorna um valor simples de uma linha. O método `fetchCol()` de um adaptador retorna um vetor de valores, obtido da primeira coluna de todas as linhas do Objeto Rowset.

Exemplo 10.39. Usando fetchColumn()

```
<?php
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');
$bug_status = $stmt->fetchColumn(2);
```

Veja também PDOStatement::fetchColumn().

10.2.3.5. Buscando uma Linha como um Objeto

Para recuperar uma linha de um Objeto Rowset estruturado como um objeto, use o `fetchObject()`. Esse método leva dois argumentos opcionais. O primeiro argumento é um literal que denomina o nome da classe do objeto a ser retornado; o padrão é 'stdClass'. O segundo argumento é um vetor de valores que serão passados para o construtor daquela classe.

Exemplo 10.40. Usando fetchObject()

```php
<?php
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');
$obj = $stmt->fetchObject();
echo $obj->bug_description;
```

Veja também PDOStatement::fetchObject().

10.3. Zend_Db_Profiler[5]

10.3.1. Introdução

`Zend_Db_Profiler` pode ser habilitado para permitir o perfilar de consultas. Perfis incluem as consultas processadas pelo adaptador assim como o tempo decorrido para rodar as consultas, permitindo a inspeção das consultas que foram executadas sem precusar de código de depuração extra para as classes. Uso avançado também permite ao desenvolvedor filtrar quais consultas serão perfiladas.

Habilite o profiler passando um diretiva para o construtor do adaptador, ou pedindo ao adaptador para habilitá-lo mais tarde.

```php
<?php
require_once 'Zend/Db.php';
$params = array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
    'profiler' => true  // liga o profiler; configura para false para
desabilitar (padrão)
);
$db = Zend_Db::factory('PDO_MYSQL', $params);
// desliga o profiler:
$db->getProfiler()->setEnabled(false);
// liga o profiler:
$db->getProfiler()->setEnabled(true);
```

O valor da opção '`profiler`' é flexível. É interpretado de modo diferente dependendo de seu tipo. De modo mais freqüente, você deve usar um valor booleano simples, mas outros tipos habilitam você a customizar o comportamento do profiler.

Um argumento booleano configura o profiler a ser habilitado se seu valor é `true` value, o desabilitado se `false`. A classe profiler é a classe profiler padrão do adaptador, `Zend_Db_Profiler`.

5 O termo em inglês para criador de perfis não será traduzido, mas suas variações sim.

```
$params['profiler'] = true;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

Uma instância de um objeto profiler faz o adaptador usar esse objeto. O tipo de objeto deve ser `Zend_Db_Profiler` ou uma subclasse do mesmo. A habilitação do profiler é feita separadamente.

```
$profiler = MyProject_Db_Profiler();
$profiler->setEnabled(true);
$params['profiler'] = $profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

O argumento pode ser um vetor associativo contendo qualquer ou todas as chaves '`enabled`', '`instance`', e '`class`'. As chaves '`enabled`' e '`instance`' correspondem ao booleano e tipos de instância documentados acima. A chave '`class`' é usada para nomear uma classe a ser usada por um profiler customizado. A classe deve ser `Zend_Db_Profiler` ou uma subclasse. A classe é instanciada sem nenhum argumento de construtor. A opção '`class`' é ignorada quando a opção '`instance`' é fornecida.

```
$params['profiler'] = array(
    'enabled' => true,
    'class'   => 'MyProject_Db_Profiler'
);
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

Finalmente, o argumento pode ser um objeto do tipo `Zend_Config` contendo propriedades, que são tratadas como as chaves de vetor descritas acima. Por exemplo, um arquivo “config.ini” pode conter os seguintes dados:

```
[main]
db.profiler.class   = "MyProject_Db_Profiler"
db.profiler.enabled = true
```

Essa configuração ode ser aplicada pelo seguinte código PHP:

```
$config = new Zend_Config_Ini('config.ini', 'main');
$params['profiler'] = $config->db->profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

A propriedade '`instance`' pode ser usada como no código seguinte:

```
$profiler = new MyProject_Db_Profiler();
$profiler->setEnabled(true);
$configData = array(
    'instance' => $profiler
    );
$config = new Zend_Config($configData);
$params['profiler'] = $config;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

10.3.2. Usando o Profiler

Em qualquer ponto, pegue o profiler usando o método `getProfiler()` do adaptador:

```
<?php
$profiler = $db->getProfiler();
```

Isso retorna uma instância de objeto `Zend_Db_Profiler`. Com essa instância, o desenvolvedor pode examinar suas consultas usando uma variedade de métodos:

- `getTotalNumQueries()` retorna o número total de consultas que foram perfiladas.
- `getTotalElapsedSecs()` retorna o número total de segundos transcorridos para todas as consultas perfiladas.
- `getQueryProfiles()` retorna um vetor de todos os perfis de consulta.
- `getLastQueryProfile()` retorna o último (mais recente) perfil de consulta, não obstante a consulta tenha ou não terminado. (se não tiver, a hora de término será nula)
- `clear()` limpa quaisquer perfis de consulta passados da pilha.

O valor de retorno de `getLastQueryProfile()` e os elementos individuais de `getQueryProfiles()` são objetos `Zend_Db_Profiler_Query`, que provêem a habilidade de inspecionar as consultas individuais por elas mesmas:

- `getQuery()` retorna o texto SQL da consulta. O texto SQL de uma declaração preparada com parâmetros é o texto no momento em que a consulta for preparada, assim ela contém espaços reservados para valores de parâmetro, não os valores usados quando a declaração é executada.
- `getQueryParams()` retorna um vetor de valores de parâmetro usados quando executar uma consulta preparada. Isso inclui tanto parâmetros de combinação quanto argumentos para o método `execute()` de declaração. As chaves do vetor são índices de parmatro posicionais (baseados em 1) ou nomeados (string).
- `getElapsedSecs()` retorna o número de segundo que a consulta levou para rodar.

A informação que o `Zend_Db_Profiler` provê é útil para perfilar gargalos em aplicações, e para depurar consultas que tenham rodado. Por exemplo, para ver a consulta exata que foi rodada por último:

```
<?php
$query = $profiler->getLastQueryProfile();
echo $query->getQuery();
```

Talvez um página seja gerada muito lentamente; use o profiler para determinar primeiro o número de segundos total de todas as consultas, e então caminhe através das consultas para encontra a que roda por mais tempo:

```
<?php
$totalTime    = $profiler->getTotalElapsedSecs();
$queryCount   = $profiler->getTotalNumQueries();
$longestTime  = 0;
$longestQuery = null;
foreach ($profiler->getQueryProfiles() as $query) {
    if ($query->getElapsedSecs() > $longestTime) {
        $longestTime  = $query->getElapsedSecs();
        $longestQuery = $query->getQuery();
    }
}
echo 'Executed ' . $queryCount . ' queries in ' . $totalTime . ' seconds' . "\n"
;
echo 'Average query length: ' . $totalTime / $queryCount . ' seconds' . "\n";
echo 'Queries per second: ' . $queryCount / $totalTime . "\n";
echo 'Longest query length: ' . $longestTime . "\n";
echo "Longest query: \n" . $longestQuery . "\n";
```

10.3.3. Uso Avançado do Profiler

Em adição à inspeção de consultas, o profiler também permite ao desenvolvedor filtrar quais consultas conseguem ser perfiladas. Os seguintes métodos operam sobre uma instância de `Zend_Db_Profiler`:

10.3.3.1. Filtro pelo tempo de consulta transcorrido

`setFilterElapsedSecs()` permite ao desenvolvedor configurar um tempo de consulta mínimo antes que uma consulta seja perfilada. Para remover o filtro, passe para o método um valor nulo.

```
<?php
// Only profile queries that take at least 5 seconds:
$profiler->setFilterElapsedSecs(5);
// Profile all queries regardless of length:
$profiler->setFilterElapsedSecs(null);
```

10.3.3.2. Filtro pelo tipo de consulta

`setFilterQueryType()` permite ao desenvolvedor configurar quais tipos de consulta devem ser perfiladas; para múltiplos tipos de perfis, use o operador booleano OR entre eles. Tipos de consulta são definidiso como as seguintes constantes `Zend_Db_Profiler`:

- `Zend_Db_Profiler::CONNECT`: operações de conexão, ou seleção de um banco de dados.

- `Zend_Db_Profiler::QUERY`: consultas gerais a banco de dados que não se encaixem em outros tipos.
- `Zend_Db_Profiler::INSERT`: qualquer consulta que adicione novos dados ao banco, geralmente SQL INSERT.
- `Zend_Db_Profiler::UPDATE`: qualquer consulta que atualize quaisquer dados existentes, normalmente SQL UPDATE.
- `Zend_Db_Profiler::DELETE`: qualquer consulta que apague dados existentes, geralmente SQL DELETE.
- `Zend_Db_Profiler::SELECT`: qualquer consulta que recupere dados existentes, geralmente SELECT.
- `Zend_Db_Profiler::TRANSACTION`: qualquer operação transacional, tal como start transaction, commit, ou rollback.

Assim como com acontece com `setFilterElapsedSecs()`, você pode remover quaisquer filtros existentes pela passagem de `null` como argumento solo.

```
<?php
// Perfila somente consultas SELECT
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT);
// Perfila consultas SELECT, INSERT, e UPDATE
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT | Zend_Db_Profile
r::INSERT | Zend_Db_Profiler::UPDATE);
// Perfila consultas DELETE (assim podemos imaginar por que os dados continuam
desaparecendo)
$profiler->setFilterQueryType(Zend_Db_Profiler::DELETE);
// Remove todos os filtros
$profiler->setFilterQueryType(null);
```

10.3.3.3. Recuperar perfis por tipo de consulta

Usar `setFilterQueryType()` pode abreviar a geração de perfis. Entretanto, algumas vezes, pode ser mais útil manter todos os perfis, mas ver somente aqueles que forem necessários em um dado momento. Outra característica de `getQueryProfiles()` é que ele pode fazer essa filtragem dinâmica, pela passagem de um tipo de consulta (ou combinação lógica de tipos de consulta) como seu primeiro argumento; veja Seção 10.3.3.2, “Filtro pelo tipo de consulta” para uma lista completa das constantes de tipo de consulta.

```
<?php
// Recupera somente perfis de consulta SELECT
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT);
// Recupera somente perfis de consulsta SELECT, INSERT, e UPDATE
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT | Zend_Db_Profi
ler::INSERT | Zend_Db_Profiler::UPDATE);
// Recupera perfis de consulta DELETE (assim podemos imaginar por que os dados
```

```
continuam desaparecendo)
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::DELETE);
```

10.4. Zend_Db_Select

10.4.1. Resumo do Objeto Select

O objeto Zend_Db_Select representa uma declaração de consulta SQL `SELECT`. A classe tem métodos para adicionar partes individuais à consulta. Você pode especificar algumas partes da consulta usando métodos PHP e estruturas de dados, e a classe forma a sintaxe SQL correta para você. Depois de construir a consulta, você opode executá-la como se você a tivesse escrito como um literal.

O valor oferecido por Zend_Db_Select inclui:

- Métodos orientados a objeto para especificar consultas SQL de uma forma passo-a-passo;
- Abstração independente de banco de dados de algumas partes da consulta SQL;
- Citação automática de identificadores de metadados (com apóstrofos na maioria dos casos), para suportar identificadores contendo palavras reservadas SQL e caracteres especiais;
- Citação de identificadores e valores, para ajudar a reduzir o risco de ataques de SQL injection.

Usar Zend_Db_Select não é obrigatório. Para consultas SELECT muito simples, é geralmente mais simples especificar a consulta SQL inteira como um literal e executá-la usando métodos do adaptador como `query()` ou `fetchAll()`. Usar Zend_Db_Select é útil se você precisar montar uma consulta SELECT proceduralmente , ou baseado em condições lógicas em sua aplicação.

10.4.2. Criando um Objeto Select

Você pode criar uma instância de um objeto Zend_Db_Select usando o método `select()` de um objeto Zend_Db_Adapter_Abstract.

Exemplo 10.41. Exemplo de um método select() de um adaptador de banco de dados

```
<?php
$db = Zend_Db::factory( ...options... );
$select = $db->select();
```

Outro modo de criar um objeto Zend_Db_Select é com seu construtor, especificando o adaptador do banco de dados como um argumento.

Exemplo 10.42. Exemplo de criação de um novo objeto Select

```
<?php
$db = Zend_Db::factory( ...options... );
$select = new Zend_Db_Select($db);
```

10.4.3. Construindo consultas Select

Quando construir a consulta, você pode adicionar cláusulas da consulta, uma a uma. Há um método separado para cada cláusula do objeto Zend_Db_Select.

Exemplo 10.43. Exemplo de uso de métodos para adicionar cláusulas

```
<?php
// Cria o objeto Zend_Db_Select
$select = $db->select();
// Adiciona uma cláusula FROM
$select->from( ...specify table and columns... )
// Adiciona uma cláusula WHERE
$select->where( ...specify search criteria... )
// Adicione uma cláusula ORDER BY
$select->order( ...specify sorting criteria... );
```

Você também pode usar a maioria dos métodos do objeto Zend_Db_Select com uma interface fluente conveniente. Uma interface fluente significa que cada método retorna uma referência para o objeto sobre o qual foi chamado, assim você pode imediatamente chmara outro método.

Exemplo 10.44. Exemplo de uso da interface fluente

```
<?php
$select = $db->select()
    ->from( ...especifique tabela e colunas... )
    ->where( ...especifique critéio de busca... )
    ->order( ...especifique critério de ordenação... );
```

Os exemplos nesta seção mostram o uso da interface fluente, mas você pode usar a intreface não-fluente em todos os casos. É freqüentemente necessário usara interface não-fluente, por exemplo, se sua aplicação precisa executar alguma lógica antes de adicionar uma cláusula à consulta.

10.4.3.1. Adicionando uma cláusula FROM

Especifique a tabela para essa consulta usando o método `from()`. Você pode especificar o nome da tabela como um literal simples. Zend_Db_Select aplica identificação por citação no nome da tabela, assim você pode usar caracteres especiais.

Exemplo 10.45. Exemplo do método from()

```
<?php
// Constrói esta consulta:
//   SELECT *
//   FROM "products"
$select = $db->select()
    ->from( 'products' );
```

Você pode também especificar o nome de correlação (algumas vezes chamado de “apelido da tabela”) para uma tabela. Ao invés de um literal simples, use um vetor associativo para mapear o nome de correlação com o nome da tabela. Em outras cláusulas da consulta SQL, use o nome de correlação. Se sua consulta agrupa mais de uma tabela, Zend_Db_Select gera nomes de correlação únicos baseados nos nomes da tabelas, para todas as tabelas cujos nomes de correlação não forem especificados.

Exemplo 10.46. Exemplo de especificação de um nome de correlação de tabela

```
<?php
// Constrói esta consulta:
//   SELECT p.*
//   FROM "products" AS p
$select = $db->select()
    ->from( array('p' => 'products') );
```

Algumas marcas de RDBMS suportam um especificador de esquema líder para uma tabela. Você pode especificar o nome da tabela como "`schemaName.tableName`", onde Zend_Db_Select delimita com apóstrofos cada parte individualmente, ou você pode especificar o nome do esquema separadamente. Um nome de esquema especificado no nome da tabela tem precedência sobre um esquema fornecido separadamente no evento em que ambos são fornecidos.

Exemplo 10.47. Exemplo de especificação de um nome de esquema

```
<?php
// Constrói esta consulta:
//   SELECT *
//   FROM "myschema"."products"
$select = $db->select()
    ->from( 'myschema.products' );
```

```
// or
$select = $db->select()
    ->from('products', '*', 'myschema');
```

10.4.3.2. Adicionando Colunas

No segundo argumento do método `from()`, Você pode especificar as colunas a serem selecionadas da respectiva tabela. Se você não especificar coluna alguma, o padrão é "`*`", o curinga SQL para “todas as colunas”.

Você pode listar as colunas em um simples vetor de literais, ou como um mapeamento associativo de nomes de coluna. Se você tem somente uma coluna para consultar, e você não precisa especificar um apelido de coluna, você pode listá-la como um literal puro ao invés de um vetor.

Se você dá um vetor vazio como argumento de colunas, nenhuma coluna da respectiva tabela será incluída no conjunto de resultado. Veja um exemplo de código relacionado na seção sobre o método `join()`.

Você pode especificar o nome da coluna como "`correlationName.columnName`". Zend_Db_Select cerca cada parte com apóstrofos individualmente. Se você não especificar um nome de correlação para uma coluna, ele usará o nome de correlação para a tabela nomeado no método `from()` atual.

Exemplo 10.48. Exemplos de especificação de colunas

```
<?php
// Constrói esta consulta:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'product_name'));
// Constrói a mesma consulta, especificando os nomes de correlação:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('p.product_id', 'p.product_name'));
// Constrói esta consulta com um apelido para uma coluna:
//   SELECT p."product_id" AS prodno, p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('prodno' => 'product_id', 'product_name'));
```

10.4.3.3. Adicionando Colunas de Expressão

Colunas em consultas SQL são algumas vezes expressões, não simplesmente nomes de colunas de uma tabela. Expressões não devem ter nomes de correlação ou apóstrofos aplicados. Se seu literal de coluna contém parênteses, Zend_Db_Select reconhece-o como uma expressão.

Você também pode criar um objeto do tipo Zend_Db_Expr explicitamente, para prevenir um literal de ser tratado como um nome de coluna. Zend_Db_Expr é uma classe mínima que contém um literal simples. Zend_Db_Select reconhece objetos do tipo Zend_Db_Expr e converte-os de volta para literais, mas não aplica qualquer alterações, tais como inserção de apóstrofos ou nomes de correlação.

Nota

Usar Zend_Db_Expr para nomes de coluna não é necessário se sua expressão de coluna contiver parênteses; Zend_Db_Select reconhece parênteses e trata o literal como uma expressão, pulando a inserção de parênteses e nomes de correlação.

Exemplo 10.49. Exemplos de especificação de colunas contendo expressões

```
<?php
// Constrói esta consulta:
//   SELECT p."product_id", LOWER(product_name)
//   FROM "products" AS p
// Uma expressão com parênteses implicitamente torna-se
// uma Zend_Db_Expr.
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'LOWER(product_name)'));
// Constrói esta consulta:
//   SELECT p."product_id", (p.cost * 1.08) AS cost_plus_tax
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'cost_plus_tax' => '(p.cost * 1.08)'));
// Constrói esta consulta usando Zend_Db_Expr explicitamente:
//   SELECT p."product_id", p.cost * 1.08 AS cost_plus_tax
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'cost_plus_tax' => new Zend_Db_Expr('p.cost * 1.08')
));
```

Nos casos acima, Zend_Db_Select não altera o literal para aplicar nomes de correlação ou apóstrofos identificadores. Se essas mudanças forem necessárias para resolver ambigüidade, você deve fazer as mudanças manualmente no literal.

Se seus nomes de coluna são palavras-chave SQL ou contêm caracteres especiais, você deve usar o método de adaptador `quoteIdentifier()` e interpolar o resultado dentro de um literal. O método `quoteIdentifier()` usa apóstrofos SQL para delimitar o identificador, o que torna claro que ele é um identificador para uma tabela ou uma coluna, e não qualquer outra parte da

sintaxe SQL.

Seu código fica mais independente da base de dados se você usar o método `quoteIdentifier()` ao invés de digitar apóstrofos literalmente em seu literal, porque algumas marcas de RDBMS usam símbolos não padronizados para identificadores de apóstrofo. O método `quoteIdentifier()` é projetado para usar os símbolos de apóstrofos apropriados baseados no tipo de adaptador. O método `quoteIdentifier()` também anula os efeitos de quaisquer caracteres de citação que apareçam dentro do próprio nome do identificador.

Exemplo 10.50. Exemplos de colunas citadas em uma expressão

```
<?php
// Constrói esta consultaq, citando um nome de coluna especial "from" na
expressão:
//   SELECT p."from" + 10 AS origin
//   FROM "products" AS p
$select = $db->select()
    ->from(array('p' => 'products'),
        array('origin' => '(p.' . $db->quoteIdentifier('from') . ' + 10)'));
```

10.4.3.4. Adicionando Outra Tabela à Consulta com JOIN

Muitas consultas úteis envolvem o uso de `JOIN` para combinar linhas de múltiplas tabelas. Você pode adicionar tabelas a uma consulta Zend_Db_Select usando o método `join()`. Usar esse método é similar ao método `from()`, exceto pelo fato de que você pode especificar também uma condição de junção na maioria dos casos.

Exemplo 10.51. Exemplo do método join()

```
<?php
// Constrói essa consulta:
//   SELECT p."product_id", p."product_name", l.*
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'product_name'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id');
```

O segundo argumento para `join()` é um literal que representa a condição de junção. Essa é uma expressão que declara os critérios pelos quais linhas em uma tabela casarão com a outra tabela. Você pode usar nomes de correlação nessa expressão.

Nota

Citação não é aplicada à expressão que você especifica para a condição de join; se você tem nomes de coluna que precisam ser citados, você deve usar `quoteIdentifier()` para formatar o literal para a condição de junção.

O terceiro argumento para `join()` é um vetor de nomes de coluna, como o usado no método `from()`. O padrão é `"*"`, suporta nomes de correlação, expressões, e Zend_Db_Expr do mesmo modo que o vetor de nomes de colunas no método `from()`.

Para não selecionar coluna alguma de uma tabela, use um vetor vazio para a lista de colunas. Esse uso funciona no método `from()` também, mas tipicamente você irá querer colunas da tabela primária em suas consultas, desde que você não deseje coluna alguma de uma tabela combinada.

Exemplo 10.52. Exemplo de não especificação de colunas

```
<?php
// Constrói essa consulta:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id', 'product_name'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id',
        array() ); // empty list of columns
```

Note o `array()` vazio no exemplo acima no lugar de uma lista de colunas de uma tabela combinada.

SQL tem diversos tipos de junções. Veja a lista abaixo para os métodos que suportam diferentes tipos de junção em Zend_Db_Select.

- **INNER JOIN** com os métodos `join(table, join, [columns])` oo `joinInner(table, join, [columns])`.

  Esse pode ser o tipo de junção mais comum. Linhas de cada tabela são comparadas usando a condição de junção que você especificar. O Objeto Rowset inclui somente as linhas que satisfazem a condição de junção. O Objeto Rowset pode ser vazio se nenhuma linha satisfaz essa condição.

  Todas as marcas de RDBMS suportam esse tipo de junção.

- **LEFT JOIN** com o método `joinLeft(table, condition, [columns])`.

  Todas as linhas da tabela à esquerda do operando são incluídas, casando com linhas da tabela à direita do operando, e as colunas da tabela à direita do operando são preenchidas

com NULLs se não houver casamento com a tabela da esquerda.

Todas as marcas de RDBMS suportam esse tipo de junção.

- **RIGHT JOIN** com o método `joinRight(table, condition, [columns])`.

  Right outer join é o complemento de left outer join. Todas as linhas da tabela à direita do operando são incluídas, casando com linhas da tabela à esquerda do operando, e as colunas da tabela à esquerda do operando são preenchidas com NULLs se não houver casamento com a tabela da direita.

  Algumas marcas de RDBMS não suportam esse tipo de junção, mas em geral qualquer right join pode ser representado como um left join pela reversão da ordem das tabelas.

- **FULL JOIN** com o método `joinFull(table, condition, [columns])`.

  Uma full outer join é como uma combinação de uma left outer join e uma right outer join. Todas as linhas de ambas as tabelas são incluídas, emparelhadas umas com as outroas na mesma linha do Objeto Rowset se elas satisfizerem a condição de junção, e caso contrário emparelhados com NULLs no lugar de colunas da outra tabela.

  Algumas marcas de RDBMS não suportam esse tipo de junção.

- **CROSS JOIN** com o método `joinCross(table, [columns])`.

  Uma cross join é um produto Cartesiano. Cada linha na primeira tabela é casada com cada linha na segunda tabela. Portanto o número de linhas no Objeto Rowset é igual ao produto do número de linhas em cada tabela. Você pode filtrar o Objeto Rowset usando condições em uma cláusula WHERE; desse modo uma cross join é similar a velha sintaxe join SQL-89.

  O método `joinCross()` não tem parâmetro para especificar a condição de junção. Algumas marcas de bancos de dados não suportam esse tipo de junção.

- **NATURAL JOIN** com o método `joinNatural(table, [columns])`.

  Uma natural join compara qualquer coluna(s) que apareça com o mesmo nome em ambas as tabelas. A comparação é a igualdade de todas as colunas; comparar as colunas usando desigualdade não é uma natural join. Somente natural inner joins são suportadas por esta API, embora SQL permita natural outer joins de qualquer forma.

  O método `joinNatural()` não tem parâmetros para especificar a condição de junção.

Em adição a esses métodos de junção, você pode simplificar suas consultas pelo uso de métodos JoinUsing. Ao invés de fornecer uma condição completa para sua junção, você simplesmente passa o nome da coluna sobre a qual se dará a junção e o objeto Zend_Db_Select completa a condição para você.

Exemplo 10.53. Exemplo do uso do método joinUsing()

```php
<?php
// Constrói esta consulta:
//   SELECT *
//   FROM "table1"
//   JOIN "table2"
//   ON "table1".column1 = "table2".column1
//   WHERE column2 = 'foo'
$select = $db->select()
    ->from('table1')
    ->joinUsing('table2', 'column1')
    ->where('column2 = ?', 'foo');
```

Cada um dos métodos de junção aplicáveis no componente Zend_Db_Select tem um método 'using' correspondente.

- `joinUsing(table, join, [columns])` e `joinInnerUsing(table, join, [columns])`
- `joinLeftUsing(table, join, [columns])`
- `joinRightUsing(table, join, [columns])`
- `joinFullUsing(table, join, [columns])`

10.4.3.5. Adicionando uma Cláusula WHERE

Você pode especificar critérios para restringir linhas do Objeto Rowset usando o método `where()`. O primeiro argumento desse método é uma expressão SQL, e essa expressão é utilizada em uma cláusula SQL `WHERE` na consulta.

Exemplo 10.54. Exemplo do método where()

```php
<?php
// Constrói essa consulta:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE price > 100.00
$select = $db->select()
    ->from(
        'products',
        array('product_id', 'product_name', 'price'))
    ->where('price > 100.00');
```

Nota

Nenhuma citação é aplicada para expressões dadas aos métodos `where()` ou `orWhere()`. Se você não tiver nomes de colunas que precisam ser citados, você deve usar

`quoteIdentifier()` para formar o literal para a condição.

O segundo argumento para o método `where()` é opcional. É um valor para substituir na expressão. Zend_Db_Select cita o valor e substitui-o por um símbolo de interrogação ("`?`") na expressão.

Esse método aceita somente um parâmetro. Se você tiver uma expressão na qual precise substituir múltiplas variáveis, você deve formatar o literal manualmente, interpolando variáveis e executar a citação por conta própria.

Exemplo 10.55. Exemplo de um parâmetro no método where()

```
<?php
// Constrói esta consulta:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)
$minimumPrice = 100;
$select = $db->select()
    ->from(
        'products',
        array('product_id', 'product_name', 'price'))
    ->where('price > ?', $minimumPrice);
```

Você pode invocar o método `where()` múltiplas vezes sobre o mesmo objeto Zend_Db_Select. A consulta resultante combina os múltiplos termos junto usando `AND` entre eles.

Exemplo 10.56. Exemplo de múltiplos métodos where()

```
<?php
// Constrói esta consulta:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)
//     AND (price < 500.00)
$minimumPrice = 100;
$maximumPrice = 500;
$select = $db->select()
    ->from('products',
        array('product_id', 'product_name', 'price'))
    ->where('price > ?', $minimumPrice)
    ->where('price < ?', $maximumPrice);
```

Se você precisar combinar termos juntos usando `OR`, use o método `orWhere()`. Esse método é usado do mesmo modo que o método `where()`, exceto que o termo especificado é precedido por `OR`, ao invés de `AND`.

Exemplo 10.57. Exemplo do método orWhere()

```
<?php
// Constrói esta consulta:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00)
//     OR (price > 500.00)
$minimumPrice = 100;
$maximumPrice = 500;
$select = $db->select()
    ->from('products',
        array('product_id', 'product_name', 'price'))
    ->where('price < ?', $minimumPrice)
    ->orWhere('price > ?', $maximumPrice);
```

Zend_Db_Select automaticamente coloca parêntereses ao redor de cada expressão que você especificar usando os métodos `where()` ou `orWhere()`. Isso ajuda a garantir que a precedência do operador booleano não provoque resultados inesperados.

Exemplo 10.58. Exemplo de inserção de parênteses em expressões booleanas

```
<?php
// Constrói esta consulta:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00 OR price > 500.00)
//     AND (product_name = 'Apple')
$minimumPrice = 100;
$maximumPrice = 500;
$prod = 'Apple';
$select = $db->select()
    ->from('products',
        array('product_id', 'product_name', 'price'))
    ->where("price < $minimumPrice OR price > $maximumPrice")
    ->where('product_name = ?', $prod);
```

No exemplo acima, os resultados deveriam ser completamente diferentes sem os parênteses, porque `AND` tem precedência mais alta que `OR`. Zend_Db_Select aplica os parênteses de modo que o efeito seja que cada expressão em chamadas sucessivas a `where()` se feche mais rigidamente do que o `AND` que combina as expressões.

10.4.3.6. Adicionando uma cláusula GROUP BY

Em SQL, a cláusula `GROUP BY` permite que você reduza as linhas de um Objeto Rowset de uma consulta para uma linha por valor único encontrado na(s) coluna(s) nomeada(s) na cláusula `GROUP BY`.

Em Zend_Db_Select, você pode especificar a(s) coluna(s) a serem usadas para calcular os grupos de linhas usando o método `group()`. O argumento para esse método é uma coluna ou um vetor de colunas a serem utilizados na cláusula `GROUP BY`.

Exemplo 10.59. Exemplo do método group()

```
<?php
// Constrói esta consulta:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id',
        array('line_items_per_product' => 'COUNT(*)'))
    ->group('p.product_id');
```

Como o vetor de colunas no método `from()`, você pode usar nomes de correlação nos literais de nomes de colunas, e a coluna é citada como um identificador a menos que o literal contenha parênteses ou seja um objeto do tipo Zend_Db_Expr.

10.4.3.7. Adicionando uma Cláusula HAVING

Em SQL, a cláusula `HAVING` aplica uma condição de restrição sobre um grupo de linhas. Isso é similar a como a cláusula `WHERE` aplica uma condição de restrição sobre as linhas. Mas as duas cláusulas são diferentes porque as condições `WHERE` são aplicadas antes que os grupos sejam definidos, já que as condições `HAVING` são aplicadas depois que os grupos são definidos.

Em Zend_Db_Select, você pode especificar condições para restringir grupos usando o método `having()`. Seu uso é similar ao do método `where()`. O primeiro argumento é um literal contendo uma expressão SQL. O segundo argumento opcional é um valor que é usado para substituir um espaço reservado para parâmetro posicional na expressão SQL. Expressões dadas em múltiplas invocações do método `having()` são combinadas usando o operador booleano `AND`, ou o operador `OR` se você usar o método `orHaving()`.

Exemplo 10.60. Exemplo do método having()

```php
<?php
// Constrói esta consulta:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   HAVING line_items_per_product > 10
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id'))
    ->join(array('l' => 'line_items'),
        'p.product_id = l.product_id',
        array('line_items_per_product' => 'COUNT(*)'))
    ->group('p.product_id')
    ->having('line_items_per_product > 10');
```

Nota

Nenhuma citação é aplicada para expressões dadas aos métodos `having()` ou `orHaving()`. Se você tiver nomes de colunas que precisam ser citados, você deve usar `quoteIdentifier()` para formatar o literal para a condição.

10.4.3.8. Adicionando uma Cláusula ORDER BY

Em SQL, a cláusula `ORDER BY` especifica uma ou mais colunas ou expressões pelas quais o Objeto Rowset de uma consulta é ordenado. Se múltiplas colunas são listadas, as colunas secundárias são usadas para resolver camadas; a ordem de classificação é determinada pelas colunas secundárias se as colunas precedentes contiverem valores idênticos. A classificação padrão é do menor valor para o maior. Você pode também classificar pelo maior valor ao menos para uma dada coluna na lista pela especificação da palavra-chave `DESC` depois da coluna.

Em Zend_Db_Select, você pode usar o método `order()` para especificar uma coluna ou vetor de colunas pelas quais classificar. Cada elemento de um vetor é um literal nomeando uma coluna, opcionalmente com a palavra-chave `ASC` `DESC` seguindo-o, separada por um espaço.

Como nos métodos `from()` e `group()`, nomes de coluna são citados como identificadores, a menos que eles contenham parênteses ou sejam objetos do tipo Zend_Db_Expr.

Exemple 10.61. Exemplo do método order()

```php
<?php
// Constrói esta consulta:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   ORDER BY "line_items_per_product" DESC, "product_id"
$select = $db->select()
    ->from(array('p' => 'products'),
        array('product_id'))
    ->join(array('l' => 'line_items'),
```

```
        'p.product_id = l.product_id',
        array('line_items_per_product' => 'COUNT(*)'))
    ->group('p.product_id')
    ->order(array('line_items_per_product DESC', 'product_id'));
```

10.4.3.9. Adicionando uma Clásula LIMIT

Algumas marcas de RDBMS estendem SQL com uma cláusula de consulta conhecida como cláusula `LIMIT`. Essa cláusula reduz o número de linhas no Objeto Rowset para um número que você especificar. Você pode também especificar um salto no número de linhas antes de iniciar a saída. Essa característica torna fácil tomar um subconjunto de um Objeto Rowset, por exemplo quando exibir resultados de consultas em páginas progressivas da saída.

Em Zend_Db_Select, você pode usar o método `limit()` para especificar a contagem de linhas e o número de linhas a saltar. O primeiro argumento para esse método é a contagem desejada de linhas. O segundo argumento é o número de linhas a saltar.

Exemplo 10.62. Exemplo do método limit()

```
<?php
// Constrói esta consulta:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20
$select = $db->select()
    ->from(array('p' => 'products'), array('product_id', 'product_name'))
    ->limit(10, 20);
```

Nota

A sintaxe `LIMIT` não é suportada por todas as marcas de RDBMS. Alguns RDBMS requerem sintaxes diferentes para suportar funcionalidade similar. Cada classe Zend_Db_Adapter_Abstract inclui um método para produzir SQL apropriada para aquele RDBMS.

Use o método `limitPage()` para um modo alternativo de especificar contagem de linhas e segmento. Esse método permite que você limite o Objeto Rowset para um de uma série de subconjuntos de linhas de comprimento fixo originários do Objeto Rowset total da consulta. Em outras palavras, você especifica o comprimento de uma “página” de resultados, e o número ordinal da página simples de resultados que você deseja que consulta retorne. O número da página é o primeiro argumento do método `limitPage()`, e o comprimento da página é o segundo argumento. Ambos os argumentos são requeridos; eles não tem valores padrão.

Exemplo 10.63. Exemplo do método limitPage()

```php
<?php
// Constrói esta consulta:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20
$select = $db->select()
    ->from(array('p' => 'products'), array('product_id', 'product_name'))
    ->limitPage(2, 10);
```

10.4.3.10. Adicionando o Modificador de Consulta DISTINCT

O método `distinct()` habilita você a adicionar a palavra-chave `DISTINCT` à sua consulta SQL.

Exemplo 10.64. Exemplo do método distinct()

```php
<?php
// Constrói esta consulta:
//   SELECT DISTINCT p."product_name"
//   FROM "products" AS p
$select = $db->select()
    ->distinct()
    ->from(array('p' => 'products'), 'product_name');
```

10.4.3.11. Adicionando o Modificador de Consulta FOR UPDATE

O método `forUpdate()` habilita você a adicionar o modificador `FOR UPDATE` à sua consulta SQL.

Exemplo 10.65. Exemplo do método forUpdate()

```php
<?php
// Constrói esta consulta:
//   SELECT FOR UPDATE p.*
//   FROM "products" AS p
$select = $db->select()
    ->forUpdate()
    ->from(array('p' => 'products'));
```

10.4.4. Executando Consultas Select

Esta seção descreve como executar a consulta representada por um objeto Zend_Db_Select.

10.4.4.1. Executando Consultas Select do Db Adapter

Você pode executar a consulta representada pelo objeto Zend_Db_Select passando-o como primeiro argumento para o método `query()` de um objeto Zend_Db_Adapter_Abstract. Use os objetos Zend_Db_Select ao invés de um literal de consulta.

O método `query()` retorna um objeto do tipo Zend_Db_Statement ou PDOStatement, dependendo do tipo de adaptador.

Exemplio 10.66. Exemplo de uso do método query() do adaptador do banco de dados

```
<?php
$select = $db->select()
  ->from('products');
$stmt = $db->query($select);
$result = $stmt->fetchAll();
```

10.4.4.2. Executando Consultas Select a partir de um Objeto

Como uma alterantiva para o uso do método `query()` do objeto adaptador, você pode usar o método `query()` do objeto Zend_Db_Select. Ambos os métodos retornam um objeto do tipo Zend_Db_Statement ou PDOStatement, dependendo do tipo de adaptador.

Exemplo 10.67. Exemplo de uso do método de consulta do objeto Select

```
<?php
$select = $db->select()
    ->from('products');
$stmt = $select->query();
$result = $stmt->fetchAll();
```

10.4.4.3. Convertendo um Objeto Select para um Literal SQL

Se você precisar acessar uma representação literal da consulta SQL correspondente ao objeto Zend_Db_Select, use o método `__toString()`.

Exemplo 10.68. Exemplo do método __toString()

```
<?php
$select = $db->select()
    ->from('products');
$sql = $select->__toString();
```

```
echo "$sql\n";
// A saí9da é o literal:
//   SELECT * FROM "products"
```

10.4.5. Outros métodos

Esta seção descreve outros métodos da classe Zend_Db_Select que não são cobertos acima: `getPart()` e `reset()`.

10.4.5.1. Recuperando Partes do Objeto Select

O método `getPart()` retorna uma representação de uma parte de sua consulta SQL. Por exemplo, você pode usar ese método para retornar o vetor de expressões para a cláusula `WHERE`, ou o vetor de colunas (ou expressões de coluna) que estão na lista `SELECT`, ou os valores da contagem e segmento para a cláusula `LIMIT`.

O valor de retorno não é um literal contendo um fragmento de sintaxe SQL. O valor de retorno é uma representação interna, que é tipicamente uma estrutura de vetor contendo valores e expressões. Cada parte da consulta tem uma estrutura diferente.

O argumento simples para o método `getPart()` é um literal que identifica qual parte da consulta retornar. Por exemplo, o literal `'from'` identifica a parte do objeto Select que armazena informações sobre as tabelas na cláusula `FROM`, incluindo tabelas unidas.

Uma classe Zend_Db_Select define constantes que você pode usar como partes da consulta SQL. Você pode usar essas definições de constantes, ou os valores literais.

Tabela 10.2. Constantes usadas por getPart() reset()

Constante	Valor literal
`Zend_Db_Select::DISTINCT`	`'distinct'`
`Zend_Db_Select::FOR_UPDATE`	`'forupdate'`
`Zend_Db_Select::COLUMNS`	`'columns'`
`Zend_Db_Select::FROM`	`'from'`
`Zend_Db_Select::WHERE`	`'where'`
`Zend_Db_Select::GROUP`	`'group'`
`Zend_Db_Select::HAVING`	`'having'`
`Zend_Db_Select::ORDER`	`'order'`
`Zend_Db_Select::LIMIT_COUNT`	`'limitcount'`
`Zend_Db_Select::LIMIT_OFFSET`	`'limitoffset'`

Exemplo 10.69. Exemplo do método getPart()

```
<?php
$select = $db->select()
    ->from('products')
    ->order('product_id');
// Você pode usar um literal para especificar a parte
$orderData = $select->getPart( 'order' );
// Você pode usar uma constante para especificar a mesma parte
$orderData = $select->getPart( Zend_Db_Select::ORDER );
// O valor de retorno pode ser uma estrutura de vetor, não um literal
// Cada parte tem uma estrutura diferente.
print_r( $orderData );
```

10.4.5.2. Recompondo Partes do Objeto Select

O método `reset()` habilita você a limpar uma parte especificada da consulta SQL, ou enão limpar todas as partes da consulta SQL se você omitir o argumento.

O argumento simples é opcional. Você pode especificar a parte da consulta a ser limpa, usando os mesmos literais que você utilizou no argumento do método `getPart()`. A parte da consulta que você especificar será recomposta para um estado padrão.

Se você omitir o parâmetro, `reset()` mudará todas as partes da consulta para seus estados padrão. Isso faz o objeto Zend_Db_Select equivalente a um novo objeto, ainda que você tenha instanciado exatamente ele.

Exemplo 10.70. Exemplo do método reset()

```
<?php
// Constrói esta consulta:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_name"
$select = $db->select()
    ->from(array('p' => 'products')
    ->order('product_name');
// Requsito alterado, ao invés disso ordena por colunas diferentes:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_id"
// Limpa uma parte assim podemos redefini-la
$select->reset( Zend_Db_Select::ORDER );
// E especifica uma coluna diferente
$select->order('product_id');
// Limpa todas as partes da consulta
$select->reset();
```

10.5. Zend_Db_Table

10.5.1. Introdução à Classe Table

A classe Zend_Db_Table é uma interface orientada a objetos para tabelas de bancos de dados. Ela fornece métodos para muitas operações comuns sobre tabelas. A classe base é estensível, assim você pode adicionar lógica customizada.

A solução Zend_Db_Table é uma implementação do padrão Table Data Gateway. A solução também inclui uma classe que implementa o padrão Row Data Gateway.

10.5.2. Definindo uma Classe Table

Para cada tabela em seu banco de dados que você quer acessar, defina uma classe que estenda Zend_Db_Table_Abstract.

10.5.2.1. Definindo o Nome e o Esquema da Tabela

Declare a tabela do banco de dados para a qual essa classe é definida, usando a variável protegida `$_name`. Ela é um literal, e deve conter o nome da tabela digitado como aparece no banco de dados.

Exemplo 10.71. Declarando uma classe de tabela com nome explícito de tabela

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
}
```

Se você não especificar o nome da tabela, o padrão é o nome da classe. Se você confiar no padrão, o nome da classe deve casar a digitação do nome da tabela como ele aparece no banco de dados.

Exemplo 10.72. Declarando uma classe Table com nome de tabela implícito

```
<?php
class bugs extends Zend_Db_Table_Abstract
{
    // nome da tabela casa com nome da classe
}
```

Você também pode declarar o esquema para a tabela, ou com a variável protegida `$_schema`, ou com o esquema colocado para o nome da tabela na propriedade `$_name`. Qualquer esquema especificado com a propriedade `$_name` tem precedência sobre um esquema especificado com a propriedade `$_schema`. Em algumas marcas de RDBMS o termo para esquema é “database” ou “tablespace”, mas é usado de forma similar.

Exemplo 10.73. Declarando uma classe Table com esquema

```
<?php
// Primeira alternativa
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_schema = 'bug_db';
    protected $_name   = 'bugs';
}
// Segunda alternativa:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_db.bugs';
}
// Se os esquemas são especificados tanto em $_name quanto $_schema,
// o especificado em $_name tem preferência:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name   = 'bug_db.bugs';
    protected $_schema = 'ignored';
}
```

Os nomes de tabela e esquemas podem também ser especificados via diretivas de configuração do construtor, o que sobrescreve quaisquer valores padrão especificados com as propriedades `$_name` e `$_schema`. Uma especificação de esquema dada com a diretiva `name` sobrescreve qualquer valor fornecido com a opção `schema`.

Exemplo 10.74. Declarando nomes de tabela e esquema na instanciação

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
}
// Primeira alterantiva:
$tableBugs = new Bugs(array('name' => 'bugs', 'schema' => 'bug_db'));
// Segunda alternativa:
$tableBugs = new Bugs(array('name' => 'bug_db.bugs');
// Se esquemas são especificados tanto em 'name' quanto 'schema',
// o especificado em 'name' tem preferência:
$tableBugs = new Bugs(array('name' => 'bug_db.bugs', 'schema' => 'ignored');
```

Se você não especificar o nome do esquema, o padrão é o esquema para o qual sua instância de adaptador de banco de dados está conectada.

10.5.2.2. Definindo a Chave Primária da Tabela

Cada tabela deve ter uma chave primária. Você pode declarar a coluna para a chave primária usando a variável protegida `$_primary`. Isso é ou um literal que nomeia a coluna simples para a chave primária, ou então um vetor de nomes de coluna se sua chave primária é uma chave composta.

Exemplo 10.75. Exemplo de especificação de chave primária

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_primary = 'bug_id';
}
```

Se você não especificar a chave primária, Zend_Db_Table_Abstract tenta descobrir a chave primária baseada na informação fornecida pelo método `describeTable()`.

Nota

Cada classe Table deve conhecer quais colunas podem ser usadas para endereçar linhas de forma única. Se não houver colunas de chave primária especificadas na definição da classe tabela ou nos argumentos do construtor da tabela, ou descobertas nos metadados de tabela fornecidos por `describeTable()`, então a tabela não pode ser usada com Zend_Db_Table.

10.5.2.3. Sobrescrevendo Métodos de Configuração de Table

Quando você cria uma instância de uma classe Table, o construtor chama um conjunto de método protegidos que inicializam metadados para a tabela. Você pode estender qualquer um desses métodos para definir metadados explicitamente. Lembre de chamar o méodo de mesmo nome na classe mãe ao fim de seu método.

Exemplo 10.76. Exemplo de sobrescrita do método _setupTableName()

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        $this->_name = 'bugs';
        parent::_setupTableName();
    }
```

```
}
```

Os métodos de configuração que você pode sobrescrever são os seguintes:

- `_setupDatabaseAdapter()` verifica se um adaptador foi fornecido; obtém um adaptador do registro se for necessário. Pela sobrescrita desse método, você pode configurar um adaptador de banco de dados da outra fonte.

- `_setupTableName()` define como padrão para o nome de tabela o nome da classe. Pela sobrescrita desse método, você pode configurar o nome da tabela antes desse comportamento padrão rodar.

- `_setupMetadata()` configura o esquema se o nome da tabela contiver o padrão "schema.table"; chame `describeTable()` para obter informações de metadados; define como padrão para o vetor `$_cols` as colunas reportadas por `describeTable()`. Pela sobrescrita desse método, você pode especificar as colunas.

- `_setupPrimaryKey()` define como padrão para as colunas de chave primária aquelas reportadas por `describeTable()`; verifica se as colunas de chave primária estão incluídas no vetor `$_cols`. Pela sobrecrita desse método, você pode especificar as colunas de chave primária.

10.5.2.4. Table initialization

Se a lógica específica da aplicação precisa ser inicializada quando uma classe Table é construída, você pode decidir-se por mover suas tarefas para o método `init()`, o qual é chamado depois que todos os metadados de Table foram processados. Isso é recomendável sobre o método `__construct` se você não precisar alterar os metadados de qualquer modo programático.

Exemplo 10.77. Exemplo de uso do método init()

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_observer;
    protected function init()
    {
        $this->_observer = new MyObserverClass();
    }
}
```

10.5.3. Criando uma Instância de Table

Antes de usar uma classe Table, crie uma instância usando seu construtor. O argumento do construtor é um vetor de opções. A opção mais importante para um construtor Table é a instância do adaptador do banco de dados, representando um conexão ativa com o RDBMS. Há três modos de especificar o adaptador de banco de dados para uma classe Table, e esses três modos são descritos abaixo:

10.5.3.1. Especificando um Adaptador de Banco de Dados

O primeiro modo de fornecer um adaptador de banco de dados para uma classe Table é passando-o como um objeto do tipo Zend_Db_Adapter_Abstract no vetor de opções, identificado pela chave `'db'`.

Exemplo 10.78. Exemplo de construção de Table usando um objeto Adapter

```
<?php
$db = Zend_Db::factory('PDO_MYSQL', $options);
$table = new Bugs(array('db' => $db));
```

10.5.3.2. Configurando um Adaptador de Banco de Dados Padrão

O segundo modo de fornecer um adaptador de banco de dados para uma classe Table é declarando um objeto do tipo Zend_Db_Adapter_Abstract para ser um adaptador de banco de dados padrão para todas as instâncias subseqüentes de Tables em sua aplicação. Você pode fazer isso com o método estático `Zend_Db_Table_Abstract::setDefaultAdapter()`. O argumento pe um objeto do tipo Zend_Db_Adapter_Abstract.

Exemplo 10.79. Exemplo de construção de um Table usando o Adaptador Padrão

```
<?php
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Db_Table_Abstract::setDefaultAdapter($db);
// Later...
$table = new Bugs();
```

Isso pode ser conveniente para criar o objeto adaptador do banco de dados em um local central de sua aplicação, tal como o bootstrap, e então armazená-lo como o adaptador padrão. Isso dá a você um meio de garantir que a instância do adaptador seja a mesma durante toda a sua aplicação. Entretanto, a configuração de um adaptador padrão está limitada a uma única instância de adaptador.

10.5.3.3. Armazenando um Adaptador de Banco de Dados no Registro

O terceiro modo de fornecer um adaptador de banco de dados para um classe Table é passando um literal no vetor de opções, também identificado pela chave `'db'`. O literal é usado como uma chave para a instância estática Zend_Registry, onde a entrada naquela chave é um objeto do tipo Zend_Db_Adapter_Abstract.

Exemplo 10.80. Exemplo de construção de um Table usando a chave Registry

```
<?php
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Registry::set('my_db', $db);
// Mais tarde...
$table = new Bugs(array('db' => 'my_db'));
```

Tal configuração do adaptador padrão dá a você meios de garantir que a mesma instância de adaptador seja usada durante sua aplicação. Usar o registro é mais flexível, porque você pode armazenar mais que um adaptador por instância. Uma instância de um dado adaptador é específica para certas marcas de RDBMS e instâncias de banco de dados. Se sua aplicação precisa de acesso para múltiplos bancos de dados ou mesmo múltiplas marcas de banco de dados, então você precisa usar múltiplos adaptadores.

10.5.4. Incluindo Linhas em uma Tabela

Você pode usar o ojeto Table para inserir linhas em um tabela de banco de dados na qual o objeto Table está baseada. Use o método `insert()` de seu objeto Table. O argumento é um vetor associativo, mapeando nomes de colunas para valores.

Exemplo 10.81. Exemplo de inclusão em um Table

```
<?php
$table = new Bugs();
$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
$table->insert($data);
```

Por padrão, os valores em seu vetor de dados são inseridos como valores literais, usando parâmetros. Se você não precisar que eles sejam tratados como expressões SQL, você deve certificar-se de que eles são distintos de literais puros. Use um objeto do tipo Zend_Db_Expr para fazer isso.

Exemplo 10.82. Exemplo de inclusão de expressões em um Table

```
<?php
$table = new Bugs();
$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
```

Nos exemplos de inserção de linhas acima, é assumido que a tabela tem uma chave primária autoincremental. Esse é o comportamento padrão de Zend_Db_Table_Abstract, mas há outros tipos de chaves primárias. As seções seguintes descrevem como suportar dferentes tipos de chaves primárias

10.5.4.1. Usar um Table com uma Chave de Auto-incremental

Uma chave primária auto-incremental gera um valor inteiro único se você omitir a coluna de chave primária de sua declaração SQL `INSERT`.

Em Zend_Db_Table_Abstract, se você definir a variável protegida `$_sequence` para ser o valor booleano `true`, então a classe assume que a tabela tem uma chave primária auto-incremental.

Exemplo 10.83. Exemplo de declaração de um Table com chave primária autoincremental

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    // Esse é o padrão na classe Zend_Db_Table_Abstract class;
    // você não precisa definir isso.
    protected $_sequence = true;
}
```

MySQL, Microsoft SQL Server, e SQLite são exemplos de marcas de RDBMS que suportam chaves primárias autoincrementais.

PostgreSQL tem uma notação `SERIAL` que implicitamente define uma seqüencia baseada na tabela e no nome da coluna, e usa a seqüencia para gerar valores de chave para novas linhas. IBM DB2 tem uma notação `IDENTITY` que funciona de forma similar. Se você usa uma dessas notações, trate sua classe Zend_Db_Table como tendo uma coluna autoincremental que diz respeito à declaração do membro `$_sequence` como `true`.

10.5.4.2. Usando um Table com um Sequence

Uma seqüencia é um objeto de banco de dados que gera um valor único, que pode ser usado como um valor de chave primária em um ou mais tables of the database.

Se você definir `$_sequence` como sendo um literal, então Zend_Db_Table_Abstract assume que o literal nomeia um objeto de seqüencia no banco de dados. A seqüencia é invocada para gerar um novo valor, e esse valor é usado na operação `INSERT`.

Exemplo 10.84. Exemplo de declaração de um Table com uma seqüencia

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_sequence = 'bug_sequence';
}
```

Oracle, PostgreSQL, e IBM DB2 são exemplos de marcas de RDBMS brands que suportam objetos de seqüencia no banco de dados.

PostgreSQL e IBM DB2 tem também sintaxe que define seqüencias implicitamente e associa-as com colunas. Se você usar essa notação, trata a tabela como tendo uma coluna de chave autoincremental. Defina o nome da seqüencia como um literal somente em casos onde você invocaria a seqüencia explicitamente para obter o próximo valor de chave.

10.5.4.3. Uando um Table com uma Chave Natural

Algumas tabelas tem um chave natural. Isso quer dizer que o dado não é automaticamente gerado pela tabela ou pela seqüencia. Você deve especificar o valor para a chave primária nesse caso.

Se você definir que `$_sequence` seja o valor booleano `false`, então Zend_Db_Table_Abstract assume que a tabela tem uma chave primária natural. Você deve fornecer valores para as colunas de chave primária no vetor de dados ao método `insert()`, caso contrário esse método lança uma Zend_Db_Table_Exception.

Exemplo 10.85. Exemplo de declaração de um Table com uma chave natural

```
<?php
class BugStatus extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_status';
    protected $_sequence = false;
}
```

Nota

Todas as marcas de RDBMS suportam tabelas com chaves naturais. Exemplos de tabelas que são freqüentemente declaradas como tendo chaves naturais são tabelas de busca, tabelas de intersecção em relacionamentos muios-para-muitos, ou a maioria das tabelas com chaves primárias compostas.

10.5.5. Atualizando Linhas em um Table

Você pode atualizar linhas em uma tabela de banco de dados usando o método `update` da classe Table. Esse método leva dois argumentos: um vetor associativo de colunas a serem alteradas e novos valores a serem associados a essas colunas; e uma expressão SQL que é usada em uma cláusula cláusula `WHERE` .

Exemplo 10.86. Exemplo de atualização de dados em um Table

```
<?php
$table = new Bugs();
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);
$where = $table->getAdapter()->quoteInto('bug_id = ?', 1234);
$table->update($data, $where);
```

Uma vez que o método `update()` representa o método `update()` do adaptador de banco de dados, o segundo argumento pode ser um vetor de expressões SQL. As expressões são combinadas como termos booleanos usando um operador `AND`.

Nota

Os valores e identificadores na expressão SQL não são citados para você. Se você tem os valores e identificadores que requerem citação, você é responsável por fazer isso. Use os métodos `quote()`, `quoteInto()`, e `quoteIdentifier()` do adaptador do banco de dados.

10.5.6. Excluindo Linhas de um Table

Você pode apagar linhas de uma tabela de banco de dados usando o método `delete()`. Esse método leva um argumento, o qual é uma expressão SQL que é usada em um cláusula `WHERE`, como critério para as linhas a serem apagadas.

Exemplo 10.87. Exemplo de exclusão de um Table

```
<?php
$table = new Bugs();
```

```
$where = $table->getAdapter()->quoteInto('bug_id = ?', 1235);
$table->delete($where);
```

O segundo argumento pode ser um vetor de expressões SQL. As expressões são combinadas como termos booleanos usando um operador `AND`.

Uma vez que o método `delete()` da tabela representa o método `delete()` do adaptador de banco de dados, o segundo argumento pode ser um vetor de expressões SQL. As expressões são combinadas como termos booleanos usando um operador `AND`.

Nota

OS valores e identificadores na expressão SQL não são citados para você. Se você tem valores ou identificadores que requerem citação, você é responsável por fazer isso. Se você tiver valores ou identificadores que requerem citação, você é responsável por fazer isso. Use os métodos `quote()`, `quoteInto()`, e `quoteIdentifier()` do adaptador de banco de dados.

10.5.7. Procurando Linhas pela Chave Primária

Você pode consultar uma tabela do banco de daods procurando por linhas que casem valores específicos na chave primária, usando o método `find()`. O primeiro argumento desse método é ou um valor simples ou um vetor de valores, para casar contra a chave primária da tabela.

Exemplo 10.88. Exemplo de busca de linhas por valores de chave primária

```
<?php
$table = new Bugs();
// Procura uma linha simples
// Returns a Rowset
$rows = $table->find(1234);
// Find multiple rows
// Also returns a Rowset
$rows = $table->find(array(1234, 5678));
```

Se você especificar um valor simples, o método retorna no máximo uma única linha, porque uma chave primária não pode ter valores duplicados e há no máximo uma linha na tabela do banco de dados que casa com o valor que você especificar. Se você especificar múltiplos valores em um vetor, o método retorna tantas linhas quanto o número de valores distintos que você especificar.

O método `find()` pode retornar menos linhas do que o número de valores que você especificar para a chave primária, se algum dos valores não casar com quaisquer linhas na tabela do banco de dados. O método pode até retornar zero linhas. Como o número de linhas retornadas é variável, o método `find()` retorna um objeto do tipo `Zend_Db_Table_Rowset_Abstract`.

Se a chave primária é uma chave composta, isso é, consiste de múltiplas colunas, você pode especificar as colunas adicionais como argumentos adicionais para o método `find()`. Você deve fornecer tantos argumentos quantos forem o número de colunas na chave primária da tabela.

Para procurar múltiplas linhas de uma tabela com uma chave primária composta, forneça um vetor para cada um dos argumentos. Todos esses vetores devem ter o mesmo número de elementos. Os valores em cada vetor são formados em tuplas ordenadas; por exemplo, o primeiro elemento em todos os argumetnos de vetor define o primeiro valor da chave primária composta, então o segundo elemento de todos os vetores define o segundo valor da chave primária composta, e assim por diante.

Exemplo 10.89. Exemplo de busca de linhas por valores de chave primária composta

Chamar `find()` abaixo para casar múltiplas linhas pode casar duas linhas no banco de dados. A primeira linha deve ter valor de chave primária (1234, 'ABC'), e a segunda linha deve ter valor de chave primária (5678, 'DEF').

```
<?php
class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';
    protected $_primary = array('bug_id', 'product_id');
}
$table = new BugsProducts();
// Procura uma linha simples com uma chave primária.
// Retorna um Rowset
$rows = $table->find(1234, 'ABC');
// Procura múltiplas linhas com chaves primárias compostas
// Também retorna um Rowset
$rows = $table->find(array(1234, 5678), array('ABC', 'DEF'));
```

10.5.8. Consultando um Conjunto de Linhas

10.5.8.1. API Select

Advertência

A API para operações de busca foi modificada para permitir que um objeto `Zend_Db_Table_Select` modificasse a consulta. Entretanto, o uso desaprovado dos métodos `fetchRow()` e `fetchAll()` continuará a funcionar sem modificação.

As seguintes declarações são todas legais e funcionalmente idênticas, entretanto é recomendável atualizar seu código para levar vantagem do novo uso quando possível.

```
// Buscando um  rowset
    $rows = $table->fetchAll('bug_status = "NEW"', 'bug_id ASC', 10, 0);
    $rows = $table->fetchAll($table->select()->where('bug_status = ?', 'NEW')
```

```
                                                    ->order('bug_id ASC')
                                                    ->limit(10, 0));
// Buscando uma linha simples
$row = $table->fetchRow('bug_status = "NEW"', 'bug_id ASC');
$row = $table->fetchRow($table->select()->where('bug_status = ?', 'NEW')
                                                   ->order('bug_id ASC'));
```

O objeto `Zend_Db_Table_Select` é uma extensão do objeto `Zend_Db_Select` que aplica restrições específicas à consulta. As agregações e restrições são:

- Você *pode* decidir retornar um subconjunto de colunas dentro de uma consulta fetchRow ou fetchAll. Isso pode fornecer benefícios de otimização quando retornar um conjunto muito grande de resultados para todas as colunas não for desejável.

- Você *pode* especificar colunas que avaliam expressões de dentro da tabela selecionada. Entretanto isso significará que a linha ou conjunto de linhas retornado será somente leitura e não poderá ser usado para operações save(). Um objeto `Zend_Db_Table_Row` com status readOnly irá lançar uma exceção se uma operação `save()` for tentada.

- Você *pode* permitir cláusulas JOIN sobre um select para permitir buscas em múltiplas tabelas.

- Você *não pode* especificar que colunas de uma tabela afetada por JOIN serão retornadas em uma linha/conjunto de linhas. Fazer isso irá provacar um erro PHP. Isso foi feito para garantir que a integridade `Zend_Db_Table` `fosse` `mantida`., isto é, um `Zend_Db_Table_Row` deve somente referenciar colunas derivadas de sua tabela mãe.

Example 10.90. Uso simples

```
<?php
$table = new Bugs();
$select = $table->select();
$select->where('bug_status = ?', 'NEW');
$rows = $table->fetchAll($select);
```

Interfaces fluentes são implementadas através do componente, assim isso pode ser reescrito de uma forma mais abreviada.

Exemplo 10.91. Exemplo de interface fluente

```
<?php
$table = new Bugs();
$rows = $table->fetchAll($table->select()->where('bug_status = ?', 'NEW'));
```

10.5.8.2. Buscando um conjunto de linhas

Você pode consultar um conjunto de linhas usando qualquer critério que não sejam os valores da chave primária, usando o método `fetchAll()` da classe Table. Esse método retorna um objeto do tipo `Zend_Db_Table_Rowset_Abstract`.

Exemplo 10.92. Exemplo de busca de linhas por uma expressão

```
<?php
$table = new Bugs();
$select = $table->select()->where('bug_status = ?', 'NEW');
$rows = $table->fetchAll($select);
```

Você pode também passar um critério de classificação em uma cláusula `ORDER BY`, assim como contagem e valores inteirtos de segmento, usados para fazer a consulta retornar um subconjunto específico de linhas. Esses valores são usados em uma cláusula `LIMIT`, ou em lógica equivalente para marcas de RDBMS que não suportam a sintaxe `LIMIT`.

Exemplo 10.93. Exemplo de busca de linhas por uma expressão

```
<?php
$table = new Bugs();
$order  = 'bug_id';
// Retorna da 21ª a 30ª linhas
$count  = 10;
$offset = 20;
$select = $table->select()->where(array('bug_status = ?' => 'NEW'))
                          ->order($order)
                          ->limit($count, $offset);
$rows = $table->fetchAll($select);
```

Todos os argumentos acima são opcionais. Se você omitir a cláusula ORDER, o Objeto Rowset inclui linhas da tabela em uma ordem imprevisível. Se a cláusula LIMIT não estiver configurada, você recuperará cada linha na tabela que casar com a cláusula WHERE.

10.5.8.3. Uso avançado

Para requisições mais específicas e otimizadas, você pode querer limitar o número de colunas retornadas em uma linha/conjunto de linhas. Isso pode ser conseguido pela passagem da cláusula FROM para o objeto select. O primeiro argumento na cláusula FROM é idêntico ao que o objeto Zend_Db_Select usa com a adição de ser capaz de passar uma instância de Zend_Db_Table_Abstract e determinar automaticamente o nome da tabela.

Exemplo 10.94. Retornar colunas específicas

```php
<?php
$table = new Bugs();
$select = $table->select();
$select->from($table, array('bug_id', 'bug_description'))
       ->where('bug_status = ?', 'NEW');
$rows = $table->fetchAll($select);
```

Importante

O conjunto de linhas contêm linhas que ainda são 'válidas' – elas simplesmente contêm um subconjunto de uma tabela. Se um método save() for chamado sobre uma linha parcial então somente os campos disponíveis serão modificados.

Você pode também especificar expressões dentro da cláusula FROM e tê-las retornadas como linhas/conjuntos de linhas. Nesse exemplo retornaremos linhas da tabela bugs que mostram uma agregação do número de novos bugs reportados por indivíduos. Note a cláusula GROUP. A coluna 'count' ficará disponível para avaliação da linha e poderá ser acessada como se fosse parte do esquema.

Exemplo 10.95. Recuperando expressões como colunas

```php
<?php
$table = new Bugs();
$select = $table->select();
$select->from($table, array('COUNT(reported_by) as `count`', 'reported_by'))
       ->where('bug_status = ?', 'NEW')
       ->group('reported_by');
$rows = $table->fetchAll($select);
```

Você também pode usar um campo de busca como parte de sua consulta para refinar suas operações de busca. Neste exemplo, a tabela accounts é consultada como parte de uma pesquisa por todos os novos bugs reportados por 'Bob'.

Exemplo 10.96. Usando uma tabela de busca para refinar os resultados de fetchAll()

```php
<?php
$table = new Bugs();
$select = $table->select();
$select->where('bug_status = ?', 'NEW')
       ->join('accounts', 'accounts.account_id = bugs.reported_by')
       ->where('accounts.account_name = ?', 'Bob');
$rows = $table->fetchAll($select);
```

A classe `Zend_Db_Table_Select` é usada primariamente para restringir e validar de modo que

ela possa forçar os critérios para uma consulta SELECT legal. Entretanto podem haver certos casos onde você requeira a flexibilidade do componente Zend_Db_Table_Row e não requeira uma linha gravável ou apagável. Para esse caso de uso específico, é possível recuperar uma linha/conjunto de linhas pela passagem de um valor falso para setIntegrityCheck. A linha/conjunto de linhas resultante será retornada como uma linha 'bloqueada' (significando que save(), delete() e quaisquer métodos configuradores de campo lançarão uma exceção).

Exemplo 10.97. Removendo a verificação de integridade sobre Zend_Db_Table_Select para permitir linhas unidas (JOINed)

```
<?php
$table = new Bugs();
$select = $table->select()->setIntegrityCheck(false);
$select->where('bug_status = ?', 'NEW')
       ->join('accounts', 'accounts.account_id = bugs.reported_by', 'account_nam
e')
       ->where('accounts.account_name = ?', 'Bob');
$rows = $table->fetchAll($select);
```

10.5.9. Consultando uma Linha Simples

Você pode consultar uma linha simples usando critérios similares ao usados pelo método `fetchAll()`.

Exemplo 10.98. Exemplo de busca de uma linha simples por uma expressão

```
<?php
$table = new Bugs();
$select  = $table->select()->where('bug_status = ?', 'NEW')
                           ->order('bug_id');
$row = $table->fetchRow($where);
```

Esse método retorna um objeto do tipo Zend_Db_Table_Row_Abstract. Se o critério de busca que você especificou não casa com linha alguma na tabela do banco de dados, então `fetchRow()` retorna o valor `null` do PHP.

10.5.10. Recuperando Informações de Metadados de Tabelas

A classe Zend_Db_Table_Abstract fornece algumas informações sobre seus metadados. O método `info()` retorna uma estrutura de vetor com informações sobre a tabela, suas colunas e chave primária, e outros metadados.

Exemplo 10.99. Exemplo de obtenção do nome da tabela

```php
<?php
$table = new Bugs();
$info = $table->info();
echo "The table name is " . $info['name'] . "\n";
```

As chaves do vetor retornado pelo método `info()` são descritas abaixo:

- **name** => o nome da tabela.
- **cols** => um vetor, nomeando a(s) coluna(s) da tabela.
- **primary** => um vetor, nomeando a(s) coluna(s) na chave primária.
- **metadata** => um vetor associativo, mapeando nomes de coluna para informações sobre as colunas. Isso é a informação retornada pelo método `describeTable()`.
- **rowClass** => o nome da classe concreta usada pelos objetos Row retornados pelos métodos dessa instância de tabela. Seu padrão é Zend_Db_Table_Row.
- **rowsetClass** => o nome da classe concreta usada para objetos Rowset retornados por métodos dessa instância de tabela. Seu padrão é Zend_Db_Table_Rowset.
- **referenceMap** => um vetor associativo, com informações sobre referências dessa tabela para qualquer tabela mãe. Veja Seção 10.8.2, "Definindo Relacionamentos".
- **dependentTables** => um vetor de nomes de classe de tabelas que referenciam essa tabela. Veja Seção 10.8.2, "Definindo Relacionamentos".
- **schema** => o nome do esquema (ou banco de dados ou tablespace) para essa tabela.

10.5.11. Cacheando Metadados de Tabela

Por padrão, `Zend_Db_Table_Abstract` consulta os bancos de dados subjacentes para metadados de tabela sobre instanciação de um objeto tabela. Isto é, quando um novo objeto tabela é criado, o comportamento padrão do objeto é buscar os metadados da tabela a partir do banco de dados usando o método `describeTable()` do adaptador.

Em algumas circunstâncias, particularmente quando muitos objetos tabela são instanciados contra o mesmo banco de dados, consultar o banco de dados por metadados de tabela para cada instância pode ser indesejável fo ponto de vista de performance. Em tais casos, usuários podem se beneficiar pelo cacheamento de metadados de tabela recuperados a partir do banco de dados.

Há dois modos primários nos quais um usuário pode levar vantagem do cacheamento de metadados de tabela:

- **Chamando Zend_Db_Table_Abstract::setDefaultMetadataCache()** - Isso permite que

um desenvolvedor configure uma vez o objeto de cache padrão a ser usado para todas as classes de tabela.

- **Configurando Zend_Db_Table_Abstract::__construct()** - Isso permite que um desenvolvedor configure o objeto de cache a ser usado para uma instância de classe de tabela particular.

Em ambos os casos, a especificação do cache deve ser ou `null` (isto é, sem uso de cache) ou uma instância de Zend_Cache_Core. Os métodos podem ser usados em conjunção quando é desejável ter tanto um cache de metados padrão e a habilidade de mudar o cache para objetos de tabela individuais.

Exemplo 10.100. Uando um Cache de Metados Padrão para todos os Objetos Table

O código seguinte demonstra como configurar um cache de metadados padrão para ser usado para todos os objetos de tabela:

```
<?php
// Primeiro, configura o Cache
require_once 'Zend/Cache.php';
$frontendOptions = array(
    'automatic_serialization' => true
    );
$backendOptions  = array(
    'cacheDir'                => 'cacheDir'
    );
$cache = Zend_Cache::factory('Core', 'File', $frontendOptions, $backendOptions);
// Depois, configura o cache a ser usado com todos os objetos de tabela
require_once 'Zend/Db/Table/Abstract.php';
Zend_Db_Table_Abstract::setDefaultMetadataCache($cache);
// Uma classe de tabela também é necessária
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}
// Cada instância de Bugs usa agora o cache de metadados padrão
$bugs = new Bugs();
```

Exemplo 10.101. Usando um Cache de Metadados para um Objeto Tabele Específico

O código seguinte demonstra como configurar um cache de metadados para uma instância específica de um objeto tabela:

```
<?php
// Primeiro, configura o Cache
require_once 'Zend/Cache.php';
$frontendOptions = array(
    'automatic_serialization' => true
    );
$backendOptions  = array(
```

```
    'cacheDir'                => 'cacheDir'
    );
$cache = Zend_Cache::factory('Core', 'File', $frontendOptions, $backendOptions);
// Uma classe tabela também é necessária
require_once 'Zend/Db/Table/Abstract.php';
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}
// Configura uma instância mediante instanciação
$bugs = new Bugs(array('metadataCache' => $cache));
```

Serialização Automática com o Cache Frontend

Uma vez que a informação retornada do método describeTable() do adaptador é um vetor, garanta que a opção `automatic_serialization` seja configurada para `true` para o frontend `Zend_Cache_Core`.

Embora o exemplo acima use `Zend_Cache_Backend_File`, desenvolvedores podem usar qualquer backend cache que seja apropriado para a situação. Por favor, veja Zend_Cache para mais informações.

10.5.12. Customizando e Estendendo uma Classe Table

10.5.12.1. Usando Classes Row ou Rowset Customizadas

Por padrão, métodos da classe Table retornam conjuntos de linhas em instâncias da classe concreta Zend_Db_Table_Rowset, e um conjunto de linhas contêm uma coleção de instâncias da classe concreta Zend_Db_Table_Row. Você pode especificar classes alternativas para usar que não sejam essas, mas ele deve uma classe que estenda Zend_Db_Table_Rowset_Abstract e Zend_Db_Table_Row_Abstract, respectivamente.

Você pode especificar as classes Row e Rowset usando o vetor de opções do construtor, nas chaves `'rowClass'` e `'rowsetClass'` respectivamente. Especifique os nomes das classes usando literais.

Exemplo 10.102. Exemplo de especificação de classes Row e Rowset

```
<?php
class My_Row extends Zend_Db_Table_Row_Abstract
{
    ...
}
class My_Rowset extends Zend_Db_Table_Rowset_Abstract
{
    ...
}
$table = new Bugs(
    array(
        'rowClass'     => 'My_Row',
```

```
        'rowsetClass' => 'My_Rowset'
    )
);
$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')
// Retorna um objeto do tipo My_Rowset,
// contendo um vetor de objetos do tipo My_Row.
$rows = $table->fetchAll($where);
```

Você pode alterar as classes especificando-as com os métodos `setRowClass()` e `setRowsetClass()`. Isso se aplica a linhas e conjuntos de linhas criados subseqüentemente; isso não altera a classe de qualquer objeto row ou rowset que você tenha criado previamente.

Exemplo 10.103. Exemplo de alteração de classes Row e Rowset

```
<?php
$table = new Bugs();
$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')
// Retorna um objeto do tipo Zend_Db_Table_Rowset
// contendo um vetor de objetos do tipo Zend_Db_Table_Row.
$rowsStandard = $table->fetchAll($where);
$table->setRowClass('My_Row');
$table->setRowsetClass('My_Rowset');
// Retorna um objeto do tipo My_Rowset,
// contendo um vetor de objetos do tipo My_Row.
$rowsCustom = $table->fetchAll($where);
// O objeto $rowsStandard ainda existe, e ele é imutável.
```

Para mais informações sobre as classes Row e Rowset, veja Seção 10.6, “Zend_Db_Table_Row” e Seção 10.7, “Zend_Db_Table_Rowset”.

10.5.12.2. Definindo Lógica Customizada para Insert, Update e Delete

Você pode sobrescrever os métodos `insert()` e `update()` em suas classes Table. Isso dá a você a oportunidade de implementar código customizado que é executado antes da realização da operação do banco de dados. Certifique-se de chamar o método da classe mãe quando você o fizer.

Exemplo 10.104. Lógica customizada para gerenciar timestamps

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    public function insert(array $data)
    {
        // adiciona uma timestamp
        if (empty($data['created_on'])) {
            $data['created_on'] = time();
```

```
        }
        return parent::insert($data);
    }
    public function update(array $data, $where)
    {
        // adiciona uma timestamp
        if (empty($data['updated_on'])) {
            $data['updated_on'] = time();
        }
        return parent::update($data, $where);
    }
}
```

Você também pode sobrescrever o método `delete()`.

10.5.12.3. Defina Métodos de Busca Customizados em Zend_Db_Table

Você pode implementar métodos de consulta customizados em suas classes Table, se você necessidade freqüente de fazer consultas contra essa tabela com critérios específicos. A maioria das consultas pode ser escrita usando `fetchAll()`, mas isso requer que você duplique código para formar as condições da consulta se você precisar rodar a consulta em diversos lugares de sua aplicação. Portanto pode ser conveniente implementar um método na classe Table class que execute consultas usadas com freqüencia contra a tabela.

Exemplo 10.105. Método Customizado para encontrar bugs por status

```
<?php
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    public function findByStatus($status)
    {
        $where = $this->getAdapter()->quoteInto('bug_status = ?', $status);
        return $this->fetchAll($where, 'bug_id');
    }
}
```

10.5.12.4. Defina Inflection em Zend_Db_Table

Algumas pessoas preferem que o nome da classe de tabela case com um nome de tabela no RDBMS através do uso de uma transformação literal chamada *inflection*.

Por exemplo, se seu nome de classe de tabela for "`BugsProducts`", seria casado com a tabela física no banco de dados chamada "`bugs_products`," se você omitir a declaração explícita da propriedade de classe `$_name`. Nesse mapeamento de inflection, o nome da classe digitado no

formato "CamelCase" seria transformado para caixa baixa, e as palavras seriam separadas com um underscore.

Você pode especificar o nome da tabela do banco de dados independentemente do nome da classe pela declaração do nome da tabela com a propriedade de classe `$_name` em cada uma de suas classes de tabela.

Zend_Db_Table_Abstract não executa inflection para mapear o nome de classe para o nome de tabela. Se você omitir a declaração de `$_name` em sua classe de tabela, a classe mapeia para uma tabela do banco de dados que casa com a digitação exata do nome da classe.

É inapropriado transformar identificadores do banco de dados, porque isso pode levar À ambigüidade ou fazer alguns identificadores inacessíveis. Usar os identificadores SQL exatamente como eles aparecem no banco de dados torna Zend_Db_Table_Abstract tanto mais simples quanto mais flexível.

Se você preferir usar inflection, então você deve implementar a transformação por conta própria, pela sobrescrita do método `_setupTableName()` em suas classes Table. Um modo de fazer isso é definir uma classe abstrata que estenda Zend_Db_Table_Abstract, e então o resto de suas tabelas estende sua nova classe abstrata.

Exemplo 10.106. Exemplo de uma classe de tabela abstrata que implementa inflection

```
<?php
abstract class MyAbstractTable extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        if (!$this->_name) {
            $this->_name = myCustomInflector(get_class($this));
        }
        parent::_setupTableName();
    }
}
class BugsProducts extends MyAbstractTable
{
}
```

Você é responsável por escrever as funções que executarão transformação de inflection. Zend Framework não provê tal função.

10.6. Zend_Db_Table_Row

10.6.1. Introdução

Zend_Db_Table_Row é uma classe que contém um linha individual de um objeto Zend_Db_Table.

Quando você roda uma consulta contra uma classe Table, o resultado é retornado em um conjunto de objetos Zend_Db_Table_Row. Você pode também usar esse objeto para criar novas linhas e adicioná-las â tabela do banco de dados.

Zend_Db_Table_Row é uma implementação do padrão Row Data Gateway.

10.6.2. Buscando uma Linha

Zend_Db_Table_Abstract fornece os métodos `find()` e `fetchAll()`, cada qual retornando um objeto do tipo Zend_Db_Table_Rowset, e o método `fetchRow()`, que retorna um objeto do tipo Zend_Db_Table_Row.

Exemplo 10.107. Exemplo de busca de uma linha

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
```

Um objeto Zend_Db_Table_Rowset contém uma coleção de objetos Zend_Db_Table_Row. Veja Seção 10.7, “Zend_Db_Table_Rowset”.

Exemplo 10.108. Exemplo de leitura de uma linha em um conjunto de linhas

```
<?php
$bugs = new Bugs();
$rowset = $bugs->fetchAll($bugs->select()->where('bug_status = ?', 1));
$row = $rowset->current();
```

10.6.2.1. Lendo valores de coluna de uma linha

Zend_Db_Table_Row_Abstract fornece métodos acessores de modo que você possa referenciar colunas na linha como propriedades de objeto.

Exemplo 10.109. Exemplo de leitura de uma coluna em uma linha

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
// Echo the value of the bug_description column
echo $row->bug_description;
```

Nota

Versões anteriores de Zend_Db_Table_Row mapeiam esses acessores de coluna para nomes de coluna do banco de dados usando uma transformação literal chamada *inflection*.

Atualmente, Zend_Db_Table_Row não implementa inflection. Nomes de propriedades acessadas precisam casar a digitação dos nomes de coluna como eles aparecem em seu banco de dados.

10.6.2.2. Recuperando Dados de Linha como um Vetor

Você pode acessar os dados da linha como um vetor usando o método `toArray()` do objeto Row. Isso retorna um vetor associativo dos nomes de coluna para os valores de coluna.

Exemplo 10.110. Exemplo de uso do método toArray()

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
// Obtém o vetor associativo coluna/valor do objeto Row
$rowArray = $row->toArray();
// Agora usa-o como um vetor normal
foreach ($rowArray as $column => $value) {
    echo "Column: $column\n";
    echo "Value:  $value\n";
}
```

O vetor retornado de `toArray()` não é atualizável. Você pode modificar os valores no vetor assim como com qualquer vetor, mas você não pode gravar alterações desse vetor diretamente para o banco de dados.

10.6.2.3. Buscando dados de tabelas relacionadas

A classe Zend_Db_Table_Row_Abstract fornece métodos para buscar linhas e conjuntos de linhas de tabelas relacionadas. Veja Seção 10.8, “Relacionamentos Zend_Db_Table” para mais informações sobre relacionamentos de tabela.

10.6.3. Escrevendo linhas em um banco de dados

10.6.3.1. Alterando valores de coluna em uma linha

Você pode configurar valores de coluna individuais usando acessores de coluna, de modo similar a como as colunas são lidas como propriedades de objeto no exemplo acima.

Usando um acessor de coluna para configurar um valor altera o valor da coluna do objeto row em sua aplicação, mas não efetiva a mudança no banco de dados ainda. Você pode fazer isso com o método `save()`.

Exemplo 10.111. Exemplo de alteração de coluna em uma linha

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
// Altera o valor de uma ou mais colunas
$row->bug_status = 'FIXED';
// Atualiza a linha no banco de dados com novos valores
$row->save();
```

10.6.3.2. Incluindo uma nova linha

Você pode criar uma nova linha para uma tabela dada com o método `createRow()` da classe de tabela. Você pode acessar campos dessa linha com a interface orientada a objetos, mas a linha não é armazenada no banco de dados até que você chame o método `save()`.

Exemplo 10.112. Exemplo de criação de uma nova linha para a tabela

```
<?php
$bugs = new Bugs();
$newRow = $bugs->createRow();
// Configura valores de coluna da forma apropriada para sua aplicação
$newRow->bug_description = '...description...';
$newRow->bug_status = 'NEW';
// Inclui a nova linha no banco de dados
$newRow->save();
```

O argumento opcional para o método createRow() é um vetor associativo, com o qual você popula campos da nova linha.

Exemplo 10.113. Exemplo de população de uma nova linha para uma tabela

```
<?php
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);
$bugs = new Bugs();
$newRow = $bugs->createRow($data);
// Inclu´r a nova linha no banco de dados
$newRow->save();
```

Nota

O método `createRow()` foi chamado `fetchNew()` nos últimos lançamentos de Zend_Db_Table. Você é encorajado a usar o novo nome do método, embora o nome antigo continue a funcionar por motivos de compatibilidade inversa.

10.6.3.3. Alterando valores em múltiplas colunas

Zend_Db_Table_Row_Abstract fornece o método `setFromArray()` para habilitar você a configurar diversas colunas em uma linha simples de uma única vez, especificadas em um vetor associativo que maperia os nomes de coluna para os valores. Você pode achar esse método conveniente para configurar valores tanto apra novas linhas quanto para linhas que você precise atualizar.

Exemplo 10.114. Exemplo de uso de setFromArray() para configurar valores em uma nova linha

```
<?php
$bugs = new Bugs();
$newRow = $bugs->createRow();
// Dados são rearranjados em um vetor associativo
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);
// Configura todos os valores de coluna de uma vez
$newRow->setFromArray($data);
// Inclui a nova linha no banco de dados
$newRow->save();
```

10.6.3.4. Apagando uma linha

Você pode chamar o método `delete()` sobre um objeto Row. Isso apaga linhas no banco de dados que casam com a chave primária no objeto Row.

Exemplo 10.115. Exemplo de remoção de uma linha

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');
// Apaga essa linha
$row->delete();
```

Você não tem de chamar `save()` para aplicara a remoção; isso é executado contra o banco de dados imediatamente.

10.6.4. Serializando e desserializando linhas

É freqüentemente conveniente gravar os conteúdos de uma linha de banco de dados para serem usados mais tarde. *Serialização* é o nome para a operação que converte um objeto em um formato que é fácil de gravar em um armazenamento offline (por exemplo, um arquivo). Objetos do tipoZend_Db_Table_Row_Abstract são serializáveis.

10.6.4.1. Serializando uma Linha

Simplesmente use a função PHP `serialize()` para criar um literal contendo uma representação linear do argumento, no caso, do objeto Row.

Exemplo 10.116. Exemplo de serialização de uma linha

```
<?php
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');
// Converte objeto para forma serializada
$serializedRow = serialize($row);
// Agora você escreve $serializedRow para um arquivo, etc.
```

10.6.4.2. Desserializando Dados de Linha

Use a função PHP `unserialize()` para restaurar um tipo string contendo uma representação literal de um objeto. A função retorna o objeto original.

Note que o objeto Row retornado está em um estado *disconnected*. Você pode ler o objeto Row e suas propriedades, mas não pode mudar valores na linha ou executar outros métodos que requeiram um conexão com a base de dados (por exemplo, consultas contra tabelas relacionadas).

Exemplo 10.117. Exemplo de desserialização de uma linha serializada

```
<?php
$rowClone = unserialize($serializedRow);
// Agora você pode usar propriedades do objeto, mas somente leitura
echo $rowClone->bug_description;
```

Por que objetos Row desserializam em um estado desconectado?
Um objeto serializado é um literal que é legível para qualquer um que o possua. Isso pode ser

um risco de segurança para armazenar parâmetros tais como contas e senhas de banco de dados em texto puro, sem encriptação de texto no literal serializado. Você não desejaria armazenar tais dados para um arquivo de texto que não está protegido, ou enviá-lo em um email ou outro meio que é facilmente lido por potenciais atacantes. O leitor do objeto serializado não deveria ser capaz de usá-lo para obter acesso ao seu banco de dados sem conhecer credenciais válidas.

10.6.4.3. Reativando uma Linha como Dado Ativo

Você pode reativar uma linha desconectada, usando o método `setTable()`. O argumento para esse método é um objeto válido do tipo Zend_Db_Table_Abstract, o qual você cria. Criar um objeto Table requer uma conexão ativa para o banco de dados, assim por meio de reassociação do objeto Table com o Row, o objeto Row obtém acesso ao banco de dados. Subseqüentemente, você altera valores no objeto Row e grava as mudanças no banco de dados.

Exemplo 10.118. Exemplo de reativação de uma linha

```
<?php
$rowClone = unserialize($serializedRow);
$bugs = new Bugs();
// Reconecta a linha à tabela, e
// então a uma conexão de banco de dados ativa
$rowClone->setTable($bugs);
// Agora você pode fazer mudanças para a linha e gravá-las
$rowClone->bug_status = 'FIXED';
$rowClone->save();
```

10.6.5. Estendendo a classe Row

Zend_Db_Table_Row é a classe concreta padrão que estende Zend_Db_Table_Row_Abstract. Você pode definir sua própria classe concreta para instâncias de Row estendendo Zend_Db_Table_Row_Abstract. Para usar sua nova classe Row para armazenar resultados de consultas de Table, especifique a classe Row customizada pelo nome ou no membro protegido `$_rowClass` de uma classe Table, ou no argumento vetor do construtor de um objeto Table.

Exemplo 10.119. Especificando uma classe Row customizada

```
<?php
class MyRow extends Zend_Db_Table_Row_Abstract
{
    // ...customizações
}
// Especifique uma Row customizada para ser usada como padrão
// em todas as instâncias de uma classe Table.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
```

```
    protected $_rowClass = 'MyRow';
}
// Ou especifique uma Row customizada para ser usada em uma
// instância de uma classe Table.
$bugs = new Bugs(array('rowClass' => 'MyRow'));
```

10.6.5.1. Inicialiação de Row

Se a lógica específica da aplicação precisa ser inicializada quando uma linha é construída, você pode optar por mover suas tarefas para o método `init()`, o qual é chamado depois que todos os metadados de linha tem sido processados. É recomendado usar o método `__construct` se você não precisar alterar os metadados de qualquer modo programático.

Exemplo 10.120. Exemplo de uso do método init()

```
<?php
class MyApplicationRow extends Zend_Db_Table_Row_Abstract
{
    protected $_role;
    protected function init()
    {
        $this->_role = new MyRoleClass();
    }
}
```

10.6.5.2. Definindo Lógica Customizada para Insert, Update, e Delete em Zend_Db_Table_Row

A classe Row chama métodos protegidos `_insert()`, `_update()`, e `_delete()` antes de executar as operações correspondentes `INSERT`, `UPDATE`, e `DELETE`. Você pode adicionar lógica a esses métodos em sua subclasse Row customizada.

Se você precisar fazer lógica customizada em uma tabela específica, e a lógica customizada deva ocorrer para cada operação nessa tabela, pode fazer mais sentido implementar seu código customizado nos métodos `insert()`, `update()` e `delete()` de sua classe Table. Entretanto, algumas vezes pode ser necessário fazer lógica customizada na classe Row.

Abaixo estão alguns exemplos de casos onde pode fazer sentido implementar lógica customizada em uma classe ao invés de na classe Table:

Exemplo 10.121. Exemplo de uma lógica customizada em uma classe Row

A lógica customizada pode não ser aplicável em todos os cassos de operações na respectiva Table. Você pode fornecer lógica customizada sobre demanda implementando-a em uma classe Row e

criando uma instância da classe Tabela com essa classe Row customizada que foi especificada. Caso contrário, a Table usará a classe Row padrão.

Você precisa de dados de operações nessa tabela para gravar a operação em um objeto Zend_Log, mas somente se a configuração da aplicação tiver habilitado esse comportamento.

```
<?php
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data, "INSERT: $this->_tableClass",
false));
    }
}
// $loggingEnabled é um exemplo de propriedade que depende
// da configuração de sua aplicação
if ($loggingEnabled) {
    $bugs = new Bugs(array('rowClass' => 'MyLoggingRow'));
} else {
    $bugs = new Bugs();
}
```

Exemplo 10.122. Exemplo de uma classe Row que efetua log de dados de inclusão para múltiplas tabelas

A lógica customizada pode ser comum para múltiplas tabelas. Ao invés de implementar a mesma lógica customizada para cada uma de suas classes Table, você pode implementar o código para tais ações na definição de uma classe Row, e usar essa classe Row em cada uma de suas classes Table.

Nesse exemplo, o código de logging é idêntico em todas as classes de tabela.

```
<?php
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data, "INSERT: $this->_tableClass",
false));
    }
}
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyLoggingRow';
}
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowClass = 'MyLoggingRow';
```

```
}
```

10.6.5.3. Defina Inflection em Zend_Db_Table_Row

Algumas pessoas preferem que o nome da classe de tabela case com o nome de tabela no RDBMS pelo uso de uma transformação literal chamada *inflection*.

Classes Zend_Db não implementam inflection por padrão. Veja Seção 10.5.12.4, "Defina Inflection em Zend_Db_Table" para uma explicação dessa diretriz.

Se você preferir usar inflection, então deve implementar a transformação por sua conta, pela sobrescrita do método `_transformColumn()` em uma classe customizada Row, e usar essa classe Row customizada quando você executar consultas contra sua classe Table.

Exemplo 10.123. Exemplo de definição de uma transformação inflection

Isso permite que você use uma versão "inflected" do nome da coluna nos acessores. A classe Row usa o método `_transformColumn()` para alterar o nome que você usa para o nome da coluna nativo na tabela do banco de dados.

```
<?php
class MyInflectedRow extends Zend_Db_Table_Row_Abstract
{
    protected function _transformColumn($columnName)
    {
        $nativeColumnName = myCustomInflector($columnName);
        return $nativeColumnName;
    }
}
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyInflectedRow';
}
$bugs = new Bugs();
$row = $bugs->fetchNew();
// Use nomes de coluna camelcase, e confie na
// função de transformação para alterá-lo para a
// representação nativa.
$row->bugDescription = 'New description';
```

Você é responsável por escrever as funções para executar transformação de inflection. Zend Framework não fornece tais funções.

10.7. Zend_Db_Table_Rowset

10.7.1. Introdução

Quando você roda uma consulta contra uma classe Table usando os métodos `find()` ou `fetchAll()`, o resultado é retornado em um objeto do tipo `Zend_Db_Table_Rowset_Abstract`. Um Rowset contém uma coleção de objetos descendentes de `Zend_Db_Table_Row_Abstract`. Você pode iterar através do Rowset e acessar objetos Row individualmente, lendo ou modificando dados nos Rows.

10.7.2. Buscando um Rowset

`Zend_Db_Table_Abstract` fornece os métodos `find()` e `fetchAll()`, cada um dos quais retorna um objeto do tipo `Zend_Db_Table_Rowset_Abstract`.

Exemplo 10.124. Exemplo de busca de um conjunto de linhas

```
<?php
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_status = 'NEW'");
```

10.7.3. Recuperando Linhas de um Conjunto de Linhas

O objeto Rowset por si próprio é geralmente menos interessante do que os objetos Rows que ele contém. Esta seção ilustra como obter Rows que compreendem o Rowset.

Uma consulta legítima retorna zero linhas quando nenhuma linha no banco de ados casa com as condições de consulta. Portanto uma objeto Rowset podem conter zero objetos Row. Uma vez que `Zend_Db_Table_Rowset_Abstract` implementa a interface `Countable`, você pode usar `count()` para determinar o número de Rows no Rowset.

Exemplo 10.125. Contando Rows em um Rowset

```
<?php
$rowset   = $bugs->fetchAll("bug_status = 'FIXED'");
$rowCount = count($rowset);
if ($rowCount > 0) {
    echo "found $rowCount rows";
} else {
    echo 'no rows matched the query';
}
```

Exemplo 10.126. Lendo um objeto Row simples de um Rowset

O modo mais simples de acessar um objeto Row de um Rowset é usar o método `current()`. Isso é particularmente apropriado quando o Rowset contém exatamente um objeto Row.

```
<?php
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_id = 1");
$row    = $rowset->current();
```

Se o Rowset contém zero linhas, `current()` retorna o valor PHP `null`.

Exemplo 10.127. Iterando atravpes de um Rowset

Objetos descendentes de `Zend_Db_Table_Rowset_Abstract` implementam a interface `SeekableIterator`, e que significa que você pode iterar através deles usando a construção `foreach`. Cada valor que você recuperar desse modo é um objeto `Zend_Db_Table_Row_Abstract` que corresponde a um registro da tabela.

```
<?php
$bugs = new Bugs();
// busca todos os registros da tabela
$rowset = $bugs->fetchAll();
foreach ($rowset as $row) {
    // exibe 'Zend_Db_Table_Row' ou similar
    echo get_class($row) . "\n";
    // lê uma coluna na linha
    $status = $row->bug_status;
    // modifica uma coluna na linha atual
    $row->assigned_to = 'mmouse';
    // grava a mudança no banco de dados
    $row->save();
}
```

Exemplo 10.128. Procurando uma posição conhecida em um objeto Rowset

`SeekableIterator` permite que você procure por uma posição para a qual você gostaria que o iterador saltasse. Simplesmente use o método `seek()` para isso. Passe um inteiro representando o número do objeto Row que você gostaria que seu Rowset apontasse como próximo, não esqueça que o índice começa em 0. Se o índice estiver errado, quer dizer, não existe, uma exceção será lançada. Você deve usar `count()` para verificar o número de resultados antes de procurar por uma posição.

```
<?php
$bugs = new Bugs();
// busca todos os registros da tabela
$rowset = $bugs->fetchAll();
// leva o iterador para o nono elemento (zero é um elemento) :
$rowset->seek(8);
// recupera-o
$row9 = $rowset->current();
// e usa-o
$row9->assigned_to = 'mmouse';
$row9->save();
}
```

`getRow()` permite que você obtenha uma linha específica do objeto Rowset, sabendo sua posição; não esqueça entretanto que as posições começam com índice zero. O primeiro parâmetro para `getRow()` é um inteiro para a posição pedida. O segundo parâmetro opcional é um booleano; ele diz ao iterador Rowset se deve procurar essa posição na mesma hora, ou não (o padrão é false). Esse método retorna um objeto Zend_Db_Table_Row por padrão. Se a posição requerida não existir, uma exceção será lançada. Aqui está um exemplo:

```
<?php
$bugs = new Bugs();
// busca todos os registros da tabela
$rowset = $bugs->fetchAll();
// recupera o nono elemento imediatamente :
$row9->getRow(8);
// e usa-o :
$row9->assigned_to = 'mmouse';
$row9->save();
}
```

Depois que você tenha acessado um objeto Row individualmente, você pode manipular o objeto usando os métodos descritos na Seção 10.6, “Zend_Db_Table_Row”.

10.7.4. Recuperando um objeto Rowset como um Vetor

Você pode acessar todos os dados no objeto Rowset como um vetor usando o método `toArray()` do objeto Rowset. Ele retorna um vetor contendo uma entrada por linha. Cada entrada é um vetor associativo com chaves que correspondem a nomes de coluna e elementos que correspondem aos respectivos valores de coluna.

Exemplo 10.129. Usando toArray()

```
<?php
$bugs    = new Bugs();
$rowset = $bugs->fetchAll();
$rowsetArray = $rowset->toArray();
```

```
$rowCount = 1;
foreach ($rowsetArray as $rowArray) {
    echo "row #$rowCount:\n";
    foreach ($rowArray as $column => $value) {
        echo "\t$column => $value\n";
    }
    ++$rowCount;
    echo "\n";
}
```

O vetor retornado por `toArray()` não é atualizável. Isto é, você pode modificar os valores no vetor como faz com qualquer vetor, mas as mudanças no vetor não serão propagadas para o banco de dados.

10.7.5. Serializando e Desserializando um objeto Rowset

Objetos do tipo `Zend_Db_Table_Rowset_Abstract` são serializéveis. Em um modo similar a serializar um objeto Row individual, você pode serializar um objeto Rowset e desserializá-lo mais tarde.

Exemplo 10.130. Serializando um objeto Rowset

Simplesmente use a função PHP `serialize()` para criar um literal contendo uma representação literal do argumento, o objeto Rowset.

```
<?php
$bugs    = new Bugs();
$rowset = $bugs->fetchAll();
// Converte objeto para um formato serializado
$serializedRowset = serialize($rowset);
// Agora você pode gravar $serializedRowset em um arquivo, etc.
```

Exemplo 10.131. Desserializando um objeto Rowset Serializado

Use a função PHP `unserialize()` para restaurar um literal contendo uma representação literal de um objeto. A função retorna o objeto original.

Note que o objeto Rowset retornado está em um estado *disconnected*. Você pode iterar através do objeto Rowset e ler os objetos Row e suas propriedades, mas você não pode alterar os valores nos objetos Rows ou executar outros métodos que requeiram uma conexão com o banco de dados (por exemplo, consultas contra tabelas relacionadas).

```
<?php
$rowsetDisconnected = unserialize($serializedRowset);
// Agora você pode usar métodos e propriedade do objeto, mas somente leitura
```

```
$row = $rowsetDisconnected->current();
echo $row->bug_description;
```

Por que Rowsets desserializa em um estado desconectado?

Um ojeto serializado é um literal que é legível para qualquer um que o possuir. Isso pode ser um risco de segurança para armazenar parâmetros tais como conta e senha de banco de dados em texto puro e sem encriptação no literal serializado. Você não iria querer armazenar dados em um arquivo de texto que não está protegido, ou enviar um e-mail ou outro meio que é facilmente lido por potenciais atacantes. O leitor do objeto serializado não deve ser capaz de usá-lo para obter acesso ao seu banco de dados sem conhecer credenciais válidas.

Você pode reativar um objeto Rowset desconectado usando o método `setTable()`. O argumento para esse método é um objeto válido do tipo `Zend_Db_Table_Abstract`, que você cria. Criar um objeto Table requer uma conexão ativa ao banco de dados, assim pela reassociação do objeto Table com o Rowset, o Rowset obtém acesso ao banco de dados. Subseqüentemente, você pode alterar valores nos objetos Row contidos no Rowset e gravar as mudanças no banco de dados.

Exemplo 10.132. Reativando um objeto Rowset como Dado Ativo

```
<?php
$rowset = unserialize($serializedRowset);
$bugs = new Bugs();
// Reconecta o conjunto de linhas para uma tabela, e
// então para uma conexão de banco de dados ativa
$rowset->setTable($bugs);
$row = $rowset->current();
// Agora você pode fazer alterações na linha e gravá-las
$row->bug_status = 'FIXED';
$row->save();
```

Reativar um objeto Rowset com `setTable()` também reativa todos os objetos Row contidos nele.

10.7.6. Estendendo a classe Rowset

Você pode usar uma classe concreta alternativa para instâncias de Rowsets estendendo Zend_Db_Table_Rowset_Abstract. Especifique a classe Rowset customizada por nome ou no membro protegido `$_rowsetClass` de uma classe Table, ou no argumento vetor da construção de um objeto Table.

Exemplo 10.133. Especificando uma classe Rowset customizada

```
<?php
class MyRowset extends Zend_Db_Table_Rowset_Abstract
{
```

```
    // ...customizações
}
// Especifique uma classe Rowset customizada para ser usada como padrão
// em todas as instâncias de uma classe Table.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowsetClass = 'MyRowset';
}
// Ou especifique uma classe Rowset customizada para ser usada em
// uma instância da classe Table.
$bugs = new Bugs(array('rowsetClass' => 'MyRowset'));
```

Tipicamente, a classe concreta padrão Zend_Db_Rowset é suficiente para a maioria dos usos. Entretanto, você pode achar útil adicionar nova lógica a um Rowset, especifica para uma tabela da . Por exemplo, um novo método poderia calcular um agregado sobre todos os objetos Rows no Rowset.

Exemplo 10.134. Exemplo de classe Rowset class com um novo método

```
<?php
class MyBugsRowset extends Zend_Db_Table_Rowset_Abstract
{
    /**
     * Encontre o objeto Row no Rowset atual com o
     * maior valor em sua coluna 'updated_at'.
     */
    public function getLatestUpdatedRow()
    {
        $max_updated_at = 0;
        $latestRow = null;
        foreach ($this as $row) {
            if ($row->updated_at > $max_updated_at) {
                $latestRow = $row;
            }
        }
        return $latestRow;
    }
}
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowsetClass = 'MyBugsRowset';
}
```

10.8. Relacionamentos Zend_Db_Table

10.8.1. Introdução

Tabelas tem relacionamentos umas com as outros em um banco de dados relacional. Uma entidade

em uma tabela pode ser ligada a uma ou mais entidades em outra tabela pelo uso de restrições de integridade referencial definidas no esquema do banco de dados.

A classe Zend_Db_Table_Row tem métodos para consultar linhas relacionadas em outras tabelas.

10.8.2. Definindo Relacionamentos

Defina classe para uma de suas tabelas, estendendo a classe abstrata Zend_Db_Table_Abstract, como descrito em Seção 10.5.2, "Definindo uma Classe Table". Também veja Seção 10.1.2, "O banco de dados de exempo" para uma descrição do exemplo do banco de dados para o qual o código de exemplo foi projetado.

Abaixo, estão as definições de classe PHP para essas tabelas:

```
<?php
class Accounts extends Zend_Db_Table_Abstract
{
    protected $_name            = 'accounts';
    protected $_dependentTables = array('Bugs');
}
class Products extends Zend_Db_Table_Abstract
{
    protected $_name            = 'products';
    protected $_dependentTables = array('BugsProducts');
}
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name            = 'bugs';
    protected $_dependentTables = array('BugsProducts');
    protected $_referenceMap    = array(
        'Reporter' => array(
            'columns'           => 'reported_by',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Engineer' => array(
            'columns'           => 'assigned_to',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Verifier' => array(
            'columns'           => array('verified_by'),
            'refTableClass'     => 'Accounts',
            'refColumns'        => array('account_name')
        )
    );
}
class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';
    protected $_referenceMap    = array(
        'Bug' => array(
            'columns'           => array('bug_id'),
            'refTableClass'     => 'Bugs',
            'refColumns'        => array('bug_id')
        ),
```

```
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id')
        )
    );
}
```

Se você usar Zend_Db_Table para emular operações UPDATE e DELETE em cascata, declare o vetor `$_dependentTables` na classe para a tabela mãe. Liste o nome da classe para cada tabela dependente. Use o nome da classe, não o nome físico da tabela SQL.

Nota

Pule a declaração de `$_dependentTables` se você usar restrições de integridade referencial no servidor RDBMS para implementar operações de cascata. Veja Seção 10.8.6, "Operações de Escrita em Cascata" para mais informações.

Declare o vetor `$_referenceMap` na classe para cada tabela dependente. Esse é um vetor associativo de "regras" de referência. Um regra de referência identifica qual tabela é a tabela mãe no relacionamento, e também lista quais colunas na tabela dependente referenciam quais colunas na tabela mãe.

A chave regra é um literal usado como um índice para o vetor `$_referenceMap`. Essa chave regra é usada para identificar cada referência de relacionamento. Escolha um nome descritivo para essa chave regra. É melhor usar um literal que pode ser parte de um nome de método, como você verá mais tarde.

No exemplo de código PHP abaixo, as chaves regra na tabela Bugs são: `'Reporter'`, `'Engineer'`, `'Verifier'`, e `'Product'`.

O valor de cada entrada rule no vetor `$_referenceMap` também é um vetor associativo. Os elementos dessa entrada de regra são descritos abaixo:

- **columns** => Um literal ou um vetor de literais definindo o(s) nome(s) de coluna de chave estrangeira na tabela dependente.

  É comum que seja uma coluna simples, mas algumas tabelas tem chaves multicoluna.

- **refTableClass** => O nome da classe da tabela mãe. Use o nome da classe, não o nome físico da tabela SQL.

  É comum para uma tabela dependente ter somente uma referência para sua tabela mãe, mas algumas tabelas tem múltiplas referências para a mesma tabela mãe. No banco de dados de exemplo, há uma referência da tabela `bugs` para a tabela `products`, mas três referências da tabela `bugs` para a tabela `accounts`. Coloque cada referência em uma entrada separada no vetor `$_referenceMap`.

- **refColumns** => Um literal ou um vetor de literais definindo o(s) nome(s) de coluna de chave prim´ria na tabela mãe.

É comum que seja uma coluna simples, mas algumas tabelas tem chaves multicoluna. Se a referência usar uma chave multicoluna, a ordem de colunas na entrada `'columns'` deve casar com a ordem de colunas na entrada `'refColumns'`.

É opcional especificar esse elemento. Se você não especificar `refColumns`, a(s) coluna(s) reportada(s) como colunas de chave primária da tabela mãe são usadas por padrão.

- **onDelete** => A regra para uma ação ser executada se uma linha for apagada na tabela mãe. Veja Seção 10.8.6, "Operações de Escrita em Cascata" para mais informações.

- **onUpdate** => A regra para uma ação ser executada se valores nas colunas de chave primária são atualizados na tabela mãe. Veja Seção 10.8.6, "Operações de Escrita em Cascata" para mais informações.

10.8.3. Buscando um objeto Rowset Dependente

Se você tiver um objeto Row como o resultado de uma consulta sobre uma tabela mãe, você pode buscar linhas de tabelas dependentes que referenciam a linha atual. Use o método:

```
<
$row->findDependentRowset($table, [$rule]);
```

Esse método retorna um objeto Zend_Db_Table_Rowset_Abstract, contendo um conjunto de linhas da tabela dependente `$table` que referencie a linha identificada pelo objeto `$row`.

O primeiro argumento `$table` pode ser um literal que especifica a tabela dependente por seu nome de classe. Você também pode especificar a tabela dependente pelo uso de um objeto da classe de tabela.

Exemplo 10.135. Buscando um objeto Rowset Dependente

Esse exempo mostra a obtenção de um objeto Row da tabela `Accounts`, e procura o `Bugs` reportado por essa conta.

```
<?php
$accountsTable      = new Accounts();
$accountsRowset     = $accountsTable->find(1234);
$user1234           = $accountsRowset->current();
$bugsReportedByUser = $user1234->findDependentRowset('Bugs');
```

O segundo argumento `$rule` é opcional. É um literal que nomeia a chave regra no vetor `$_referenceMap` da classe de tabela dependente. Se você não especifica uma regra, a primeira regra no vetor que referencia a tabela mãe é usada. Se você precisar usar uma regra outra que não a primeira, você precisa especificar a chave.

No exemplo de código acima, a chave regra não foi especificada, assim a regra usada por padrão é a primeira que casa com a tabela mãe. Essa é a regra `'Reporter'`.

Exemplo 10.136. Buscando um objeto Rowset Dependente por uma Regra Específica

Esse exemplo mostra a obtenção de um objeto Row da tabela `Accounts`, e procura o `Bugs` associado para ser corrigido pelo usuário dessa conta. A chave regra que corresponde a essa referência de relacionamento nesse exemplo é `'Engineer'`.

```
<?php
$accountsTable      = new Accounts();
$accountsRowset     = $accountsTable->find(1234);
$user1234           = $accountsRowset->current();
$bugsAssignedToUser = $user1234->findDependentRowset('Bugs', 'Engineer');
```

Você pode também adicionar critérios, ordenando e limitando para que seus relacionamentos usem o objeto selecionado da linha mãe.

Exemplo 10.137. Buscando um objeto Rowset Dependente usando um Zend_Db_Table_Select

Esse exemplo mostra a obtenção de um objeto Row da tabela `Accounts`, e procura o `Bugs` associado para ser corrigido pelo usuário dessa conta, limitado somente a 3 linhas e ordenado por nome.

```
<?php
$accountsTable      = new Accounts();
$accountsRowset     = $accountsTable->find(1234);
$user1234           = $accountsRowset->current();
$select             = $accountsTable->select()->order('name ASC')
                                              ->limit(3);
$bugsAssignedToUser = $user1234->findDependentRowset('Bugs', 'Engineer', $selec
t);
```

Alternativamente, você pode consultar linhas de uma tabela dependente usando um mecanismo especial chamado "método mágico". Zend_Db_Table_Row_Abstract invoca o método: `findDependentRowset('<TableClass>', '<Rule>')` se você invoca um método no objeto Row que case com um dos seguintes padrões:

- `$row->find<TableClass>()`
- `$row->find<TableClass>By<Rule>()`

Nos padrões acima, `<TableClass>` e `<Rule>` são literais que correspondem ao nome da classe da tabela dependente, e à chave de regra da tabela dependente que referencia a tabela mãe.

Nota

Alguns frameworks de aplicação, tais como Ruby on Rails, usam um mecanismo chamado "inflection" para permitir a digitação de identificadores de alteração dependendo do uso. Por simplicidade, Zend_Db_Table_Row não fornece qualquer mecanismo de inflection. A identidade da tabela e a chave de regra nomeada na chamada de método devem casar exatamente com a digitação da classe e chave de regra.

Exemplo 10.138. Buscando Rowsets Dependentes usando o Método Mágico

Esse exemplo mostra a busca de Rowsets dependentes equivalentes a esses nos exemplos anteriores. Nesse caso, a aplicação usa a invocação do método mágico ao invés de especificar a tabela e regra como literais.

```
<?php
$accountsTable     = new Accounts();
$accountsRowset    = $accountsTable->find(1234);
$user1234          = $accountsRowset->current();
// Usa a regra de referência padrão
$bugsReportedBy    = $user1234->findBugs();
// Especifica a regra de referência
$bugsAssignedTo    = $user1234->findBugsByEngineer();
```

10.8.4. Buscando uma Linha Mãe

Se você tem um objeto Row como resultado de uma consulta sobre uma tabela dependente, você pode buscar a linha na mãe para a qual a dependente se refere. Use o método:

```
<
$row->findParentRow($table, [$rule]);
```

Deve sempre haver exatamente uma linha na tabela mãe referenciada por uma linha dependente, portanto esse método retorna um objeto Row, não um objeto Rowset.

O primeiro argumento `$table` pode ser um literal que especifica a tabela mãe por seu nome de classe. Você também pode especificar a tabela mãe pelo uso de um objeto dessa classe de tabela.

Exemplo 10.139. Buscando a Linha Mãe

Esse exemplo mostra a obtenção de um objeto Row da tabela `Bugs` (por exemplo um desses bugs com status 'NEW'), e a busca da linha na tabela `Accounts` para o usuário que reportou o bug.

```
<?php
$bugsTable          = new Bugs();
$bugsRowset         = $bugsTable->fetchAll(array('bug_status = ?' => 'NEW'));
$bug1               = $bugsRowset->current();
$reporter           = $bug1->findParentRow('Accounts');
```

O segundo argumento `$rule` é opcional. É um literal que nomeia a chave de regra no vetor `$_referenceMap` da classe de tabela dependente. Se você não especificar uma regra, a primeira regra no vetor que referenciar a tabela mãe será usada. Se você precisar usar uma regra outra que não a primeira, você precisa especificar a chave.

No exemplo acima, a chave de regra não foi especificada, assim a regra usada por padrão é a primeria que casa com a tabela mãe. Essa é a regra `'Reporter'`.

Exemplo 10.140. Buscando uma Linha Mãe por uma Regra Específica

Esse exemplo mostra a obtenção de um objeto Row da tabela `Bugs`, e a busca da conta para o engenheiro associado corrigir o bug. O literal da chave de regra que corresponde a essa referência de relacionamento nesse exemplo é `'Engineer'`.

```
<?php
$bugsTable          = new Bugs();
$bugsRowset         = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1               = $bugsRowset->current();
$engineer           = $bug1->findParentRow('Accounts', 'Engineer');
```

Alternatively, you can query rows from a parent table using a "magic method". Zend_Db_Table_Row_Abstract invokes the method: `findParentRow('<TableClass>', '<Rule>')` if you invoke a method on the Row object matching either of the following patterns:

- `$row->findParent<TableClass>([Zend_Db_Table_Select $select])`
- `$row->findParent<TableClass>By<Rule>([Zend_Db_Table_Select $select])`

Nos padrões acima, `<TableClass>` e `<Rule>` são literais que correspondem ao nome de classe da tabela mãe, e a chave de regra da tabela dependente que referencia a tabela mãe.

Nota

A identidade de tabela e a chave de regra nomeadas na chamada do método devem casar exatamente com a digitação da classe e chave de regra.

Exemplo 10.141. Buscando a Linha Mãe usando o Método Mágico

Esse exemplo mostra a busca de objetos Rows mãe equivalentes a esses nos exemplos anteriores. Nesse caso, a aplicação usa a invocação do método mágico ao invés de especificar a tabela e regra como literais.

```php
<?php
$bugsTable          = new Bugs();
$bugsRowset         = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1               = $bugsRowset->current();
// Usa a regra de referência padrão
$reporter           = $bug1->findParentAccounts();
// Especifica a regra de referência
$engineer           = $bug1->findParentAccountsByEngineer();
```

10.8.5. Buscando um objeto Rowset através de um Relacionamento Muitos-para-muitos

Se você tem um objeto Row como resultado de um consulta sobre uma tabela em um relacionamento muitos-para-muitos (para propósitos de exemplo, chama essa a tabela "original"), você pode buscar linhas correspondentes na outra tabela (chame essa a tabela "de destino") via por meio de uma tabela de intersecção. Use o método:

```
<
$row->findManyToManyRowset($table, $intersectionTable, [$rule1, [$rule2, [Zend_D
b_Table_Select $select]]]);
```

Esse método retorna um objeto Zend_Db_Table_Rowset_Abstract contendo linhas da tabela `$table`, satisfazendo o relacionamento muitos-para-muitos. O objeto Row atual da tabela de origem é usado para buscar linhas na tabela de intersecção, e isso é juntado à tabela de destino.

O primeiro argumento `$table` pode ser um literao que especificar a tabela "de destino" em um relacionamento muitos-para-muitos por seu nome de classe. Você pode também especificar a tabela de destino pelo uso de um objeto da mesma classe.

Example 10.142. Fetching a Rowset with the Many-to-many Method

Esse exemplo mostra a obtenção de um objeto Row da tabela original `Bugs`, e a busca de linha da tabela de destino `Products`, representando produtos relacionados ao bug.

```php
<?php
$bugsTable          = new Bugs();
$bugsRowset         = $bugsTable->find(1234);
$bug1234            = $bugsRowset->current();
$productsRowset     = $bug1234->findManyToManyRowset('Products', 'BugsProducts');
```

O terceiro o quarto argumentos `$rule1` e `$rule2` são opcionais. Esses são literais que nomeiam a chave de regra no vetor `$_referenceMap` da tabela de interseção. A cahve `$rule1` nomeia a regra para o relacionamento da tabela de intersecção com a tabela original. Nesse exemplo, esse é

relacionamento de `BugsProducts` para `Bugs`.

A chave `$rule2` nomeia a regra para o relacionamento da tabela de intersecção para a tabela de destino. Nesse exemplo, esse é o relacionamento de `Bugs` para `Products`.

Similarmente aos métodos para buscar linhas mães e dependentes, se você não especificar uma regra, o método usa a primeira regra no vetor `$_referenceMap` que casa com a tabela no relacionamento. Se você precisar usar uma regra outra que não a primeira, você precisa especificar a chave.

No exemplo de código acima, a chave de regra não foi especificada, assim as regras usadas por padrçoa foream as primeiras que casaram. Nesse caso, `$rule1` é `'Reporter'` e `$rule2` é `'Product'`.

Exemplo 10.143. Buscando um objeto Rowset com um Método Muitos-para-muitos por uma Regra Específica

Esse exemplo mostra a obtenção de tabela original `Bugs`, e a busca de linhas da tabela de destino `Products`, representando produtos relacionados a esse bug.

```
<?php
$bugsTable        = new Bugs();
$bugsRowset       = $bugsTable->find(1234);
$bug1234          = $bugsRowset->current();
$productsRowset   = $bug1234->findManyToManyRowset('Products', 'BugsProducts', '
Bug');
```

Alternativamente, você pode consultar linhas da tabela de destino em um relacionamento muitos-para-muitos usando um "método mágico". Zend_Db_Table_Row_Abstract invoca o método: `findManyToManyRowset('<TableClass>', '<IntersectionTableClass>', '<Rule1>', '<Rule2>')` se você invoca um método casando qualquer um dos seguintes padrões:

- `$row->find<TableClass>Via<IntersectionTableClass> ([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1> ([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1>And<Rule2> ([Zend_Db_Table_Select $select])`

Nos padrões acima, `<TableClass>` e `<IntersectionTableClass>` são literais que correspondem aos nomes de classe da tabela de destino e a tabela de intersecção, respectivamente. `<Rule1>` e `<Rule2>` são literais que correspondem às chaves de regra na tabela de intersecção

que referencia a tabela de origem e a tabela de destino, respectivamente.

Nota

As identidades da tabela e as chaves de regra nomeadas na chamada de método devem casar exatamente com a digitação da classe e chave de regra.

Exemplo 10.144. Buscando objetos Rowsets usando o Método Mágico Muitos-para-muitos

Esse exemplo mostra a busca de linhas na tabela de destino de um relacionamento muitos-para-muitos representando produtos relacionados a um dado bug.

```
<?php
$bugsTable        = new Bugs();
$bugsRowset       = $bugsTable->find(1234);
$bug1234          = $bugsRowset->current();
// Usa a regra de referência padrão
$products          = $bug1234->findProductsViaBugsProducts();
// Especifica a regra de referência
$products          = $bug1234->findProductsViaBugsProductsByBug();
```

10.8.6. Operações de Escrita em Cascata

Declare DRI no banco de dados:

Declarara operações em cascata Zend_Db_Table é algo pretendido somente para RDBMS que não suportam integridade referencial relativa (DRI).

Por exemplo, se você usar a máquina de armazenamento MyISAM storage, ou SQLite, essas soluções não suportam DRI. Você pode achar útil declarar operações me cascasta.

Se seu RDBMS implementa DRI e as cláusulas `ON DELETE` e `ON UPDATE`, você deve declarar essas cláusulas em seu esquema de banco de dados, ao invés de usar a característica de cascata em Zend_Db_Table. Declarar regras DRI em cascata é melhor para a performance, consistência e integridade.

O mais importrante, não declare operações em cascata tanto no RDBMS quanto em sua classe Zend_Db_Table.

Você pode declarar operações em cascata para executar contra uma tabela dependente quando você aplica um `UPDATE` e `DELETE` em uma linha de uma tabela mãe.

Exemplo 10.145. Exemplo de um Delete em Cascata

Esse exemplo mostra a exclusão de uma linha na tabela `Products`, que é configurada para excluir automaticamente linhas dependentes na tabela `Bugs`.

```php
<?php
$productsTable  = new Products();
$productsRowset = $productsTable->find(1234);
$product1234    = $productsRowset->current();
$product1234->delete();
// Automaticamente cascateria para a tabela Bugs
// e exclui linhas dependentes.
```

Similarmente, se você usa `UPDATE` para alterar o valor de uma chave primária na tabela mãe, você pode querer que o valor em chaves estrangeiras de tabelas dependentes seja automaticamente atualizado para casar com o novo valor, assim tais referências são mantidadas atualizadas.

Geralmente não é necessário atualizar o valor de uma chave primária que foi gerada por uma seqüência ou outro mecanismo. Mas se você usar uma *natural key* que possa alterar o valor ocasionamente, é mais comum que você aplique atualizações em cascata em tabelas dependentes.

Para declarar um relacionamento em cascata na classe Zend_Db_Table, edite as regras no `$_referenceMap`. Configure as chaves do vetor associativo `'onDelete'` e `'onUpdate'` para o literal 'cascade' (ou a constante `self::CASCADE`). Antes que uma linha seja excluída d a tabela mãe, ou que os valores de suas chaves primárias sejam atualizados, quaisquer linhas na tabela dependente que se refira a tabela mãe será apagada ou atualizada primeira.

Exemplo 10.146. Exemplo de Declaração de Operações em Cascata Declaration of Cascading Operations

No exemplo abaixo, linhas na tabela `Bugs` são excluídas automaticamente se a linha na tabela `Products` para o qual elas se referem foi excluida. O elemento `'onDelete'` do mapa de referência é configurado para `self::CASCADE`.

Nenhuma atualização em cascata é feita no exemplo abaixo se o valor da chave primária na classe mãe é alterado. O elemento `'onUpdate'` da entrada do mapa de referência é `self::RESTRICT`. Você pode obter o mesmo resultado usando o valor `self::NO_ACTION`, ou pela omissão da entrada `'onUpdate'`.

```php
<?php
class BugsProducts extends Zend_Db_Table_Abstract
{
    ...
    protected $_referenceMap    = array(
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id'),
            'onDelete'          => self::CASCADE,
            'onUpdate'          => self::RESTRICT
        ),
        ...
    );
}
```

10.8.6.1. Notas de Consideração sobre Operações em Cascata

Operações em cascata invocadas pelo Zend_Db_Table não são atômicas.

Isso significa que se seu banco de dados implementa e força restrições de integridade referencial, um `UPDATE` em cascata executado pela classe Zend_Db_Table entra em conflito com a restrição e resulta em uma violação da integridade referencial. Você pode usar `UPDATE` em cascata em Zend_Db_Table *somente* se seu banco de dados não forçar essa restrição de integridade referencial.

`DELETE` em cascata sofre menos do problema de violações de integridade referencial. Você pode excluir linhas dependentes como ações não atômicas antes de deixar todo mundo entrar.

Entretanto, tanto para `UPDATE` quanto para `DELETE`, mudar o banco de dados em um modo não atômico também cria o risco de que eesse outro usuário de banco de dados possa ver os dados em um estado de inconsistência. Por exemplo, se você excluir uma linha e todas as suas linhas dependentes, há uma pequena chance de que outro programa cliente de banco de dados possa consultar o banco de dados depois que você excluiru as linhas dependentes, mas antes que você exclua a linha mãe. Esse programa cliente pode ver a linha mãe sem nenhuma linha dependente, e assume que esse seja o estado pretendido dos dados. Não há modo para que o cliente saba que sua consulta lê o banco de dados no meio de uma alteração.

Esse questão da mudança não atômica pode ser mitigada pelo uso de transações para isolar suas alterações. Mas algumas marcas de RDBMS não suportam transações, ou permitem que clientes leiam alterações “sujas” que ainda não tenham sido submetidas.

Operações em cascata no Zend_Db_Table são invocadas por Zend_Db_Table.

Exclusões e atualizações em cascata definidas em suas classes são aplicaadas se você executar os métodos `save()` or `delete()` na classe Row. Entretanto, se você atualizar ou excluir dados usando outra interface, tal como um ferramenta de consulta ou outra aplicação, as operações em cascata não são aplicadas. Mesmo quando usamos os métodos `update()` e `delete()` na classe Zend_Db_Adapter class, operações em cascata definidas em suas classes Zend_Db_Table não são executadas.

Sem Cascateamento de `INSERT`.

Não há suporte para um `INSERT` em cascata. Você deve inverter uma linha em uma tabela mãe em uma única operação, e inserir linha(s) para uma tabela dependente em uma operação separada.