——高性能PHP应用开发之

第10讲 数据库基本操作

目录

CONTENTS

1 数据库迁移和填充

2 原生SQL查询和查询构造器

3 Eloquent ORM

目录

CONTENTS

1 数据库迁移和填充

2 原生SQL查询和查询构造器

3 Eloquent ORM

数据库迁移

数据库的迁移和填充作为开发数据库的辅助手段可以极大地提高开发效率，使得对数据库的管理和控制变得简单。在实际程序开发过程中，经常采取迭代的开发方式，数据库架构也会不断变化。数据库迁移实际上可以看做是数据库的版本控制，通过在Laravel框架下建立数据库迁移文件，可以很容易地实现数据库的维护更新。而数据库填充可以通过PHP代码文件控制数据库中数据内容，方便程序的测试。

数据库迁移本质上通过定义一个统一的接口来实现数据库架构的创建和维护。在Laravel框架中，通过**schema门面**很轻松地构建数据库表结构（使用代码创建）；当后续数据库发生改变时，只需要更新数据库表结构代码，执行几个Artisan命令即可实现数据库迁移。

Laravel 的Schema门面提供了与数据库系统无关的创建和操纵表的支持，在 Laravel 所支持的所有数据库系统中提供一致的、优雅的、平滑的API。

数据库迁移

■ **数据库迁移流程：**

① 创建迁移数据表文件

② 设置数据表结构

③ 执行数据库迁移

创建迁移表文件

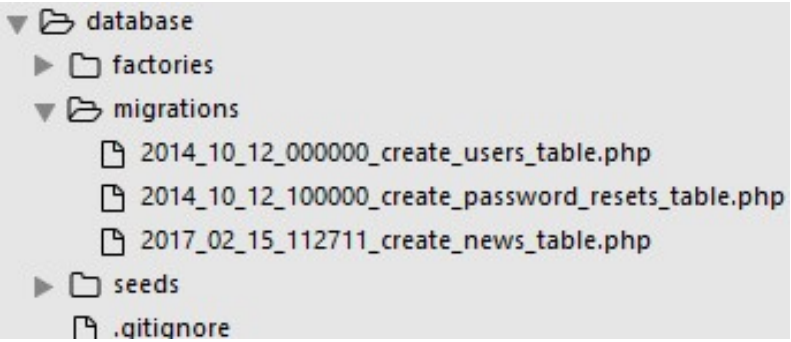
■ 使用 Artisan 命令 **make:migration** 来创建一个数据表迁移文件

✓ 新创建的数据表迁移文件位于 /database/migrations/ 目录下，每个迁移文件名都包含时间戳从而允许程序判断其先后顺序。

✓ 可以添加几个命令参数：

- --table：表明使用已经存在的某一个数据表（修改数据表结构）
- --create：表明创建新的数据表（默认）
- --path：迁移文件的保存目录

```
> php artisan make:migration create_news_table
Created Migration: 2017_02_15_112711_create_news_table
```



创建迁移表文件

- 新创建好的数据表迁移文件中包含两个方法：
  - ✓ up( )：运行迁移；用于新增表、列或者索引到数据库
  - ✓ down( )：撤销迁移；up方法的反操作

```
class CreateNewsTable extends Migration
{
    /**
    public function up()
    {
        //
    }

    /**
    public function down()
    {
        //
    }
}
```

设置数据表结构

- 创建表：使用**Schema门面上的create方法**来创建新的数据表
- 删除表：使用Schema门面的 **drop 方法**， 写在 迁移文件的 down( )方法 内
- 创建字段：使用Blueprint实例对象的**指定方法创建指定数据类型**的字段
- 删除字段：使用Blueprint实例对象的 **dropColumn**( ) 方法 删除某个或某些字段

注意数据表的名字为 **迁移表名称 + s**

```php
public function up() {
    Schema::create('newss', function(Blueprint $table) {
        $table->increments('id');    // id字段
        $table->string('title');     // varchar类型
        $table->text('content');     // text类型
        $table->timestamps();        // 添加created_at和updated_at列
    });
}
```

```php
public function down() {
    Schema::drop('newss');
}
```

执行数据库迁移

- 要运行应用中所有未执行的迁移，可以使用 Artisan 命令提供的**migrate**方法；即把当前对数据表迁移文件所做的修改，写入到真实的数据库中（**首先配置好数据库连接信息**）。

```
> php artisan migrate
Migrated: 2017_02_15_112711_create_news_table
```

字段 (5)

id	int(10) unsigned	否	<auto_increment>
title	varchar(255)	否	
content	text	否	
created_at	timestamp	是	<空>
updated_at	timestamp	是	<空>

- 回滚迁移：取消上一次（或上几次）的迁移

✓ 回滚上一次迁移：php artisan migrate:rollback

✓ 回滚上几次的迁移：php artisan migrate:rollback --step=5

✓ 回滚所有迁移（重置库）：php artisan migrate:reset

✓ 回滚所有迁移并重新运行迁移：php artisan migrate:refresh

数据库填充

Laravel 使用填充类和测试数据提供了一个简单方法来填充数据到数据库。所有的填充类都位于database/seeds目录。安装完 Laravel 后，会默认提供一个DatabaseSeeder类。从这个类中，你可以使用call方法来运行其他填充类，从而允许你控制填充顺序。

- 编写填充：要生成一个填充器，可以通过 Artisan 命令**make:seeder**。所有框架生成的填充器都位于database/seeders目录。
  - ✓ 一个填充器类默认只包含一个方法：run。当Artisan命令**db:seed**运行时该方法被调用。
  - ✓ 在run方法中，可以插入任何你想插入数据库的数据，你可以使用查询构建器手动插入数据，也可以使用 Eloquent 模型工厂。
- 运行填充：db:seed命令运行可以用来运行其它填充器类的DatabaseSeeder类，但是，你也可以使用--class 选项来指定你想要运行的独立的填充器类。

目录

CONTENTS

1 数据库迁移和填充

2 原生SQL查询和查询构造器

3 Eloquent ORM

配置数据库连接

在Laravel中建立数据库连接非常简单，只需要修改配置文件，设置数据库连接配置项即可。但是需要注意的是，在Laravel中关于数据库配置信息有两个位置：一是在/config/database.php中，另一个在 /.env 文件中。

- **/config/database.php文件**：设置框架所采用的数据库类型，数据库连接配置信息，框架会自动从该文件中加载配置信息。注意，该文件会自动读取 .env文件中的配置信息。

```
'default' => env('DB_CONNECTION', 'mysql'),
```

```
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
```

配置数据库连接

- **/.env 文件**：环境相关的配置文件。数据库连接信息，不建议直接修改/config/database.php文件，而应该在 .env文件中设置数据库连接信息；这里设置的信息将会自动覆盖 /config/database.php 文件中的对应信息。

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=blog
DB_USERNAME=root
DB_PASSWORD=root
```

执行原生SQL查询

配置好数据库连接后，就可以使用**DB门面**来运行查询。DB门面为每种查询提供了相应方法：**select, update, insert, delete和statement**。

- select：执行SQL查询，以数组形式返回指定SQL语句的结果集。
- update：更新数据库中已存在的记录，该方法返回受更新语句影响的行数。
- insert：执行数据库插入操作，若插入成功，返回true；否则返回false
- delete：用于删除数据库中已存在的记录，返回被删除的行数。
- 有些数据库语句不返回任何值，可以使用DB门面的statement方法。

```
Route::get('news/add', function() {
    $r = Db::insert('insert into newss(title, content) values(?,?)', ['news_title'
        , 'news_content']);
    dd($r);
});
```

查询构造器

数据库查询构建器提供了一个方便的接口用于创建和执行数据库查询。查询构建器可以用于执行应用中大部分数据库操作，并且能够在支持的所有数据库系统上工作。

Laravel 查询构建器使用 PDO 参数绑定来避免 SQL 注入攻击，不再需要过滤传递到绑定的字符串。

查询构造器

数据库查询构建器提供了一个方便的接口用于创建和执行数据库查询。查询构建器可以用于执行应用中大部分数据库操作，并且能够在支持的所有数据库系统上工作。

Laravel 查询构建器使用 PDO 参数绑定来避免 SQL 注入攻击，不再需要过滤传递到绑定的字符串。

■ **获取查询构造器**：使用**Db门面的 table静态方法**

```
static Builder table(string $table, string $connection = null)
```

✓ $table表示待操作的数据表名称。

✓ $connection表示需要操作的数据库连接对象；默认为空，表示使用配置文件中的默认数据库连接对象。

```
$builder = Db::table('newss');
dd($builder);
```

获取结果集

■ 获取表中指定条件的**所有记录**：**get( )**方法

```
Collection get(array $columns = array('*'))
```

✓ 参数表示待获取结果集的字段列表，默认为所有字段

✓ get( )方法返回结果为Collection对象，该对象可以使用 foreach 循环处理

```
$results = Db::table('newss')->get();
foreach ($results as $row) {
    echo $row->title;
}
```

■ 获取表中指定条件的**一条记录**：**first( )**方法

```
stdClass|null first(array $columns = array('*'))
```

✓ 返回记录的stdClass对象形式，不能使用数组形式访问

```
$row = Db::table('newss')->first();
echo $row->title;
```

where查询

■ 使用查询构建器上的**where方法**可以添加where子句到查询中。

```
$this where(string|array|Closure $column, string $operator = null,
mixed $value = null, string $boolean = 'and')
```

- ✓ 参数1表示 条件字段名；
- ✓ 参数2表示 任意一个数据库系统支持的操作符（如 >、<=等）；
- ✓ 参数3表示 该字段要比较的值；
- ✓ 参数4表示 多个where条件之间的逻辑关系（默认为 and 关系）；
- ✓ where( ) 方法返回 $this 对象（即查询构造器对象），以方便后续链式调用。

```
$results = Db::table('newss')
    ->where('id', '>', 2)
    ->where('title', 'like', 'news%')
    ->get();
dd($results);
```

其它常用方法

- orderBy( )：通过给定字段对结果集进行排序
- groupBy( )：对结果集进行分组
- join( )：连接查询
- select( )：选择指定的字段
- count( )：返回指定条件的结果集个数
- take( )和limit( )：限定结果集个数
- insert( )：插入记录
- update( )：更新记录
- delete( )：删除记录
- getQueryLog( )：获取最后执行的SQL语句
- 更多方法：https://laravel.com/api/master/Illuminate/Database/Query/Builder.html

数据库事务

- 要在一个数据库事务中运行一连串操作，可以使用DB门面的**transaction**方法，如果事务闭包中抛出异常，事务将会自动回滚。如果闭包执行成功，事务将会自动提交。

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
});
```

- 手动使用事务

✓ 开启事务：DB::beginTransaction();

✓ 回滚事务：DB::rollBack();

✓ 提交事务：DB::commit();

目录

CONTENTS

1 数据库迁移和填充

2 原生SQL查询和查询构造器

3 **Eloquent ORM**

Eloquent ORM

Laravel 自带的 Eloquent ORM 提供了一个美观、简单的与数据库交互的 ActiveRecord 实现，每张数据表都对应一个与该表进行交互的“模型”，模型允许你在表中进行数据查询，以及插入、更新、删除等操作。

■ Eloquent模型继承自 Illuminate\Database\Eloquent\Model类，可以放在 **App 目录下的任意目录**中。

✓ 模型文件可以位于 App目录下，也可以位于其任意子目录下，只要 Composer 能够自动加载即可。

✓ 使用Artisan命令创建模型类：php artisan **make:model** 模型类名

- 附加参数 --migration（-m）、--controller（-c）、--resource（-r）

```
D:\WWW\Laravel\Cms>php artisan make:model Goods -m -c -r
Model created successfully.
Created Migration: 2017_03_07_123359_create_goods_table
Controller created successfully.
```

Eloquent模型约定

■ Eloquent模型类中可以添加几个重要属性：

✓ $table 属性：指明当前模型类操作的数据表名称

- 若未指定该属性，则使用 **模型类名 + s** 形式表示数据表名

✓ $primaryKey 属性：指明当前数据表的主键名称，默认为 id

✓ $incrementing 属性：主键字段是否为自增类型，默认为 true

✓ $timestamps 属性：模型类是否自动修饰 created_at 和 updated_at 字段

- 在数据库迁移中，使用 $table->timestamps( ); 自动添加这两个字段

✓ $dates 属性：数据表结构变异字段数组（软删除）

- 使用软删除时，在迁移表结构中，使用 $table->softDeletes( ); 添加 deleted_at 字段
- 判断给定模型实例是否被软删除，可以使用trashed方法

✓ 其它属性：Illuminate\Database\Eloquent\Model 类源代码

```php
// 当前模型类对应的数据表名
protected $table = "goods";
// 当前数据表主键字段
protected $primaryKey = 'id';
// 是否自动更新created_at和updated_at字段
public $timestamps  = true;
// 数据表结构变异字段（软删除）
protected $dates = ['deleted_at'];
```

Eloquent读取数据

- Eloquent使用方式：
  - ✓ **静态**形式：　　类名::方法名
  - ✓ **对象**形式：　　对象名->方法名( )
- 获取多条记录：**all( )、get( )、find(主键数组)**
  - ✓ 返回结果为 Collection类型，可以使用 foreach 遍历循环
- 获取单条记录：**find(主键)、first( )**
- 获取字段值：value( )
- 获取聚合：count( )、sum( )、max( )、min( )、avg( )等

```php
// 获取所有记录
$results = Goods::all();
// 获取所有记录
$results = Goods::get();
// 获取指定条件的所有记录
$results = Goods::where('id', '>', 3)->all();
// 获取指定主键列表的所有记录
$results = Goods::find([1, 3, 5]);
// 获取指定主键的单条记录
$row = Goods::find(3);
// 获取指定条件的单条记录
$row = Goods::where('title', 'test')->first();
// 获取字段值
$value = Goods::find(3)->value('title');
// 获取结果集个数
$count = Goods::all()->count();
```

Eloquent增加数据

- 增加数据
  - ✓ **save**( )方法：创建模型对象，直接为属性赋值，然后save即可
  - ✓ **insert**( )方法：传入数组形式数据
    - 注意，此方法不能自动生成 timestamps 字段
  - ✓ **create**( )方法：根据表单参数，生成模型对象，然后save即可
    - $fillable 和 $guarded属性

```php
// 增加数据
$goods = new Goods();
$goods->title = 'test';
$goods->content = 'test test';
$goods->add_time = date('Y-m-d H:i:s');
$goods->save();
```

```php
Goods::insert([
    [
        'title' =>  't1',
        'content'   =>  't1 t1',
    ],
    [
        'title' =>  't2',
        'content'   =>  't1 t2',
    ],
]);
```

Eloquent修改数据

- 修改数据
  - ✓ **save**( )方法：获取模型对象，修改属性值，然后save即可
  - ✓ **update**( )方法：传入数组形式数据（更新后的数据），注意，传入where条件

```php
// 修改数据
$goods = Goods::find(3);
$goods->title = 'change id = 3';
$goods->save();
```

```php
// 修改数据
Goods::where('id', '=', 3)
    ->update([
        'title' =>  't1',
        'content'   =>  't1 t1',
    ]);
```

Eloquent删除数据

- 删除数据
  - ✓ **delete**( )方法：根据where条件删除指定记录
  - ✓ **destroy**( )方法：删除主键所限定的记录
  - ✓ 软删除问题
    - 先在数据表中建立 deleted_at 字段
    - 在模型类中引入 Traits： SoftDeletes
    - 模型类中添加 $dates 属性
    - 执行删除即可

```php
// 删除数据
Goods::where('title', '=', 't1')
    ->delete();

// 删除数据
Goods::destroy([1, 3, 5]);
```

多表关联关系

- 多表关联关系

✓ 一对一关系：每一个 Member 具有一条 MemberInfo

- Member **hasOne** MemberInfo
- MemberInfo **belongsTo** Member

✓ 一对多关系：每一个 Member 发表了多条 Msg

- Member **hasMany** Msgs
- Msg **belongsTo** Member

✓ 多对多关系：每一个 Member 具有多个 Role，每一种 Role 具有多个 Member

- Member **belongsToMany** Roles
- Role **belongsToMany** Members

一对一关系的实现

- 模型类中实现一对一关系
  - ✓ hasOne：父表 hasOne 子表
  - ✓ belongsTo：子表 belongsTo 父表

```php
class Member extends Model
{
    public function memberInfo()
    {
        return $this->hasOne('App\MemberInfo');
    }
}
class MemberInfo extends Model
{
    public function member()
    {
        return $this->belongsTo('App\Member');
    }
}
```

- 获取关联数据

```php
// 从Member表中获取MemberInfo
$row = Member::find(5);      // Member表中记录
dd($row->memberInfo->email);    // 获取关联表字段
```

Member表

字段	意义
id	主键
name	用户名

MemberInfo表

字段	意义
id	主键
member_id	关联字段

一对多关系的实现

- 模型类中实现一对多关系
  - ✓ hasMany：父表 hasMany 子表
  - ✓ belongsTo：子表 belongsTo 父表

- 获取关联数据

```php
// 从Member表中获取Msgs
$row = Member::find(1);
foreach ($row->msgs as $msg) {
    dump($msg->title);
}
```

```php
// 从Msg获取Member
$row = Msg::find(5);
dd($row->member->name);
```

```php
class Member extends Model
{
    public function msgs()
    {
        return $this->hasMany('App\Msg');
    }
}

class Msg extends Model
{
    public function member()
    {
        return $this->belongsTo('App\Member');
    }
}
```

Member表

字段	意义
id	主键
name	用户名

Msg表

字段	意义
id	主键
member_id	关联字段

多对多关系的实现

- 模型类中实现多对多关系
- ✓ belongsToMany：A表 belongsToMany B表
- ✓ belongsToMany：B表 belongsToMany A表

```php
class Member extends Model
{
    public function roles()
    {
        return $this->belongsToMany('App\Role', 'role_member');
    }
}

class Role extends Model
{
    public function members()
    {
        return $this->belongsToMany('App\Member', 'role_member');
    }
}
```

Member表

字段	意义
id	主键
name	用户名

Role表

字段	意义
id	主键
name	角色名

role_member表

字段	意义
role_id	角色id
member_id	用户id

多对多关系的实现

- 模型类中实现多对多关系
- 获取关联数据

```php
// 获取Member所从属的Roles
$row = Member::find(5);
foreach ($row->roles as $role) {
    dump($role->name);
}
```

```php
// 获取Role所包含的Members
$row = Role::find(1);
foreach ($row->members as $member) {
    dump($member->name);
}
```

关联关系的插入/修改/删除

- hasMany关系插入数据
  - ✓ save( )方法：为子表添加一条记录
  - ✓ saveMany( )方法：为子表添加多条记录
  - ✓ create( )方法：通过数组参数为子表添加记录
- belongsTo关系插入数据
  - ✓ associate( )方法：插入父表数据时，在子表中插入外键
  - ✓ dissociate( )方法：取消关联，在子表中删除外键
- belongsToMany插入数据
  - ✓ attach( )方法：建立关联关系
  - ✓ detach( )方法：取消关联关系

```php
// Member发表了多条Msgs
$member = Member::find(5);
$member->msgs()->saveMany([
    new Msg(['title'=>'mt1', 'content'=>'c1']),
    new Msg(['title'=>'mt2', 'content'=>'c2']),
]);
```

```php
// 修改Msg的member_id
$member = Member::find(3);
$msg = Msg::find(10);
$msg->member()->associate($member);
$msg->save();
```

```php
// 建立belongsToMany关联关系
$member = Member::find(8);
$member->roles()->attach([1, 2]);
```

```php
// 取消belongsToMany关联关系
$member = Member::find(8);
$member->roles()->detach();
```

Collection对象

Eloquent 返回的所有的包含多条记录的结果集都是Illuminate\Database\Eloquent\Collection 对象的实例，包括通过 get 方法或者通过访问关联关系获取的结果。Collection集合实现了迭代器接口，允许你像PHP数组一样对其进行循环。

```php
$users = App\User::where('active', 1)->get();
foreach ($users as $user) {
    echo $user->name;
}
```

另一方面，集合使用直观的接口提供了各种映射/简化操作，因此比数组更加强大。

```php
$names = $users->reject(function ($user) {
    return $user->active === false;
})->map(function ($user) {
    return $user->name;
});
```

路由模型绑定

注入模型ID到路由或控制器动作时，通常需要查询数据库才能获取相应的模型数据。Laravel 路由模型绑定让注入模型实例到路由变得简单，例如，你可以将匹配给定 ID 的整个 User 类实例注入到路由中，而不是直接注入用户 ID。

Laravel 会自动解析定义在路由或控制器动作（变量名匹配路由片段）中的 Eloquent 模型类型声明。

```
Route::get('api/users/{user}', function (App\User $user) {
    return $user->email;
});
```

在这个例子中，由于类型声明了 Eloquent 模型 App\User，对应的变量名 $user 会匹配路由片段中的{user}，这样，Laravel 会自动注入与请求 URI 中传入的 ID 对应的用户模型实例。

如果数据库中找不到对应的模型实例，会自动生成 HTTP 404 响应。

序列化

- Collection集合转化为数组或JSON
  - ✓ 转化为数组：toArray( )
  - ✓ 转化为JsON：toJson( )
- 序列化时指定白名单或黑名单
  - ✓ $hidden属性：黑名单，序列化时不会被转化
  - ✓ $visible属性：白名单，只有指定的字段会被序列化

感谢聆听！

THANK YOU FOR YOUR ATTENTION