Eloquent Model 关系

hanOne

#User Model 在Model里面 默认函数
public function phone()
{
return $this->hasOne('App\Phone');
}

默认的外键是模块的名称_id 比如user 表hasOne Task 那么 tasks 表有一个 column 叫 user_id 如果想自定义 就要第二个参数
第二个参数的决定就是子表对应于本表 id 的字段名称
#User Model
$this->hasOne('App\Phone', 'foreign_key')

user 表默认的对外的键是id 如果相反 分传入第三个参数 第二个参数的决定就是主表的primary_key
#User Model
$this->hasOne('App\Phone', "foreign_key", "local_key")

belongsTo

Task Model
public function user()
{
return $this->belongsTo('App\User');
}

如果本表对应的外键不是user_id 用来对应 User model时候 将要提供 第二个参数
#Task Model
public function user()
{
return $this->belongsTo('App\User',"foreign_key");
}

如果主表不是使用的id作为primary key 将提供第三个参数
public function user()
{
return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}

hasMany

belongsTo

hasMany（关键字 pivot）

belongsToMany

比如说user 和role 的关系那用第三张表来对应这种关系
users, roles, and role_user

如果想自定义中间表的名称 第二个参数就是了
return $this->belongsToMany('App\Role', "role_user");

如果向自定义中间表关于再张表的primary key 的名字
return $this->belongsToMany('App\Role', "role_user", 'user_id', 'role_id');
第二个参数是本表的user的foreign key
第四个参数是role 对应 的foreign key

获取中间键的 使用 pivot 这个属性是向动添加的

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

默认情况下只有user_id 和role_id 这两个 columns
如果想添加另外的属性
return $this->belongsToMany('App\Role')->withPivot('approved');

withTimestamps 将自动添加 created_at 和 updated_at
return $this->belongsToMany('App\Role')->withTimestamps();

进行额外的筛选 使用下面的方法
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);
return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);

自定义中间件model 使用下面的函数
中间件模块必须继承 Illuminate\Database\Eloquent\Relations\Pivot

```
public function users()
{
    return $this->belongsToMany('App\User')->using('App\UserRole');
}
```

hasManyThrough

简单的讲就是一张表belongsto 另外一张表 同时hasmany 第三张表

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

可以使用 $country->posts 来获取poste 相当牛逼
定义country model 的时候
public function posts()
{
return $this->hasManyThrough('App\Post', 'App\User');
}
第一个参数是我们最终想获得的model 第二个参数是中间的model

如果想指定 各个表中的primary key 可以使用下面的函数
return $this->hasManyThrough(
'App\Post', 'App\User',
'country_id', 'user_id', 'id'
);
第三个参数是 中间件Model 的foreign key 第四个参数是 目的表对应的foreign key 第五个参数是本表的primary key

多态（关键字morphTo 和 morphMany）

简单的将三张表 第三张表里面 包含一二两张表的内容，第二张表有一个字段用来区分属于一二两张表的哪一个

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments 中有posts 也有videos 的评论 laravel 称之为多态

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}
```

$post->comments 来获取comments 结果

$comment->commentable 来获取 到post 还是 video
The commentable relation on the Comment model will return either a Post or Video instance, depending on which type of model owns the comment.

默认情况下 comment table 对应的commentable_type 字段 会是 App\Post or App\Video
如果想自定义的名称 创建如下函数 注册在 APP\ServiceProvider 中
#APP\ServiceProvide
use Illuminate\Database\Eloquent\Relations\Relation;

```
Relation::morphMap([
    'posts' => App\Post::class,
    'videos' => App\Video',
]);
```

You may register the morphMap in the boot function of your AppServiceProvider or create a separate service provider if you with.

多态ManyToMany（关键字morphToMany，morphedByMany）

一个tag 包含多个 posts 或者 videos 或者再种都包含个（人理解）
文档如下 好像更倾向于 一个tag 把post 和 video 同时包括
For example, a blog Post and Video model could share a polymorphic relation to a Tag model. Using a many-to-many polymorphic relation allows you to have a single list of unique tags that are shared across blog posts and videos
但是这里用的是数两以后是个人理解是对的

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

```
class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
video 一样的定义方法
```

```
class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

$post->tags
$tag->posts

获取相关对象方法 可以使用 query builder 的方法

$comments = App\Post::find(1)->comments;

如果想进一步进行筛选
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();

query 关系对象对象获取方法 关键字

has、withCount ，
whereHas、
whereDoesntHave、
doesntHave

```
// Retrieve all posts that have at least one comment...
$posts = App\Post::has('comments')->get();
```

相反的 $posts = App\Post::doesntHave('comments')->get();

```
// Retrieve all posts that have three or more comments...
$posts = App\Post::has('comments', '>=', 3)->get();
```

```
// Retrieve all posts that have at least one comment with votes...
$posts = App\Post::has('comments.votes')->get();
```

使用whereHas and orWhereHas 加入query builder 筛选

```
// Retrieve all posts with at least one comment containing words like foo%
$posts = App\Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

```
$posts = App\Post::whereDoesntHave('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

单纯的计算数量 使用withCount 函数 在array 中获取 (tables) _count 来获取数量

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

自定义添加counts 方法
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

```
echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

eager loading（基礎,中文不知道怎么翻译）主要是用来解决 query效益的问题

```
Author hasMany books
book belongsto Author

$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

如果有25本书的话将会执行26次这个效率太低

```
# eager loading to reduce this operation to just 2 queries
使用了with 关键字
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}

select * from books
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

可以一次 eager 多个relationships tables
$books = App\Book::with('author', 'publisher')->get();

```
$books = App\Book::with('author.contactor')->get();
以eager loading 的一个嵌套中加载了 author 和 author.contactor
```

```
with 里面再进行筛选
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

如果book 已经被写出来 用来判断一些条件 是否要 load related models
$books = App\Book::all();
($someCondition) {
    $books->load('author', 'publisher');
# 这里只载了两个关系
}

如果还要加入筛选的话
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);

添加关系 save()其实不止包含hasmany 这个比较通用

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment);
```

```
$post = App\Post::find(1);
$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

```
$post = App\Post::find(1);
$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Before using the create method, be sure to review the documentation on attribute mass assignment.

belongsto 添加关系

```
#user belongsTo account
$account = App\Account::find(10);
$user->account()->associate($account);
$user->save();
```

When removing a belongsTo relationship, you may use the dissociate method

```
$user->account()->dissociate();
$user->save();
```

ManyToMany 关系的添加 关键字attach、detach

user 拥有很多roles，一个role 包含很多users
attach 方法

```
$user = App\User::find(1);
$user->roles()->attach($roleId);
第二个参数用来更新中间表格
$user->roles()->attach($roleId, ['expires' => $expires]);
```

The detach method will remove the appropriate record out of the intermediate table

```
$user->roles()->detach($roleId);
// Detach all roles from the user...
$user->roles()->detach();
```

```
方便起见 也接受一个数组作为参数
$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

只有sync参数里面包含的id保存 其他的都被从中间表格中移除

```
$user->roles()->sync([1, 2, 3]);

$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

```
$user->roles()
    ->syncWithoutDetaching([1, 2, 3]); #sync相反只有不包含 1，2，3
```

```
$user->roles()->toggle([1, 2, 3]); 假如1，2，3 是attached 那么就变成 detached 如果是detached 就变成attached
```

App\User::find(1)->roles()->save($role, ['expires' => $expires]); 和上面 hasmany的save 方法一样 第二个参数用来更新中间表

```
#更新中间表格数据（attribute）
This method accepts the pivot record foreign key and an array of attributes to update
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

touch parent timestamps

就是在子表中更新数据之后想更新父表 updated_date 字段数据

```
class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Now, when you update a Comment, the owning Post will have its updated_at column updated as well, making it more convenient to know when to invalidate a cache of the Post model