

ModelHelper

对laravel Eloquent OMR 的一层带缓存的封装

封装的目的

1. 在laravel的使用过程中，发现很多对model层缓存的透明化封装，而这些封装主要是对所有的sql语句进行了缓存，如果有数据的更新或者删除，则对数据进行了全部删除，做的好一些的，就是对表级数据，进行了删除。

在实际使用过程中，特别是web类，面向用户的操作，更多的只是简单的select操作，如果我们将这些简单的select单条查询存入缓存，讲连表操作改成 select一张表中的数据list，然后foreach 该list，循环取另一张表的info类型，我们可以避免80%以上的连表操作。

但是，这种做法，对缓存控制要求就很高了，对于缓存脏数据的清理，我们希望更精准，谁脏了，就干掉谁，而不是批量处理的做法，封装了此扩展。

2. 自动缓存包含精准缓存的缓存删除操作。

什么是原子化缓存？

对应于原子化操作的定义，对一条不可在细分数据的缓存就是原子化缓存，通俗点说，就是对表的行数据的缓存

主要功能

1. 自动原子化缓存（行数据缓存）
2. 自动分页缓存
3. hasOne/hasMany 时，外键自动缓存
4. 常用方法封装

用法

1.安装

1. composer require hbclare/model-helper:dev-master
2. 修改config/app.php,
将'Eloquent' => 'Illuminate\Database\Eloquent\Model',
改为
'Eloquent' => 'Hbclare\ModelHelper\Model',
3. 在porviders里面，增加 'Hbclare\ModelHelper\ModelHelperServiceProvider',

2.artisan方法，生成程序

假设我们有一个user表

字段	说明
id	主键id
passport	登陆账号
password	登陆密码
nickname	昵称
...	...

```
php artisan make:eachmodel Models/User
```

会在/app/Models 下生成User的model类

```
php artisan make:repository Repository/UserRepository
```

会在/app/Repository 下生成UserRepository类

同时会在 /app/RepositoryInterface 目录下，生成UserRepositoryInterface类

3.在model中，缓存的使用

```
<?php
#Game.php
namespace App\Models\Desktop;
use Hbclare\ModelHelper\Model;

class Game extends Model
{
    //设置表名
    protected $table = 'game';
```

```

//设置修改字段
protected $fillable = [
    'game_category',
    'game_name',
    'game_license_key',
    'reviewed_status',
    'active_begin_data',
    'active_end_data',
];

public function __construct(array $attributes = [])
{
    parent::__construct($attributes);
    //开启自动原子缓存,设置缓存10小时
    $this->startAutoEachCache(600);
    //开启自动分页缓存, 实验方法, 暂时默认设置5分钟
    $this->startAutoPageCache(5);
    //改后, 需要处理的缓存
    $this->setAfterUpdateFlushKey(
        ['gameinfo_by_name_{game_name}']
    );
    //删后, 需要处理的缓存
    $this->setAfterDeleteFlushKey(['gameinfo_by_name_{game_name}']);
    //新增后, 需要处理的缓存
    $this->setAfterInsertFlushKey(['getGameList*']);
    //增删改, 都需要处理的缓存
    $this->setFlushKeys(['gameinfo_by_name_{game_name}']);
}

public function hasManyImage()
{
    return $this->hasMany('App\Models\CMGameImg', 'game_id');
}

public function getGameInfo($id){
    return $this->findOne($id);
}

public function changeGame($id, $game_name){
    return $this->saveInfo(['id'=>$id, 'game_name'=>$game_name]);
}

...
}

```

```

<?php
#GameImg.php
namespace App\Models;

use Hbclare\ModelHelper\Model;

class GameImg extends Model {
    //设置表名
    protected $table = 'game_img';

    //设置修改字段
    protected $fillable = [
        'game_id',
        'game_name',
        'game_img_url',
    ];
    //设置外键
    protected $foreignKeyArr = [
        'game_id',
    ];
    //关闭自动更新时间戳
    #public $timestamps = false;

    public function __construct(array $attributes = [])
    {
        parent::__construct($attributes);
        $this->startAutoEachCache(120);#设置120分钟缓存
    }

    public function belongsToGame()
    {
        return $this->belongsTo(CMGame::class, 'game_id', 'id');
    }
}

```

1. 执行 `$game->getGameInfo` 就会自动生成 `autoCache_tableName_primaryName_{id}` 的缓存
2. 执行 `$game->changeGame` 就会删除 `autoCache_tableName_primaryName_{id}`这个缓存
3. 执行 `$game->getGameInfo(1)->hasManyImage`,会自动生成, `'autoForeignCache_'.table_name.''.foreignKey_name.*. {foreignKeyValue};`
4. 执行 `$gameImg update,delete`操作, 都会删除相关缓存
5. 缓存优先级 `setCacheKeys > autoForeignCache > autoCache`

4.可以对较复杂的Model层操作，快速的进行缓存，同时对于该缓存key的设计，可以自动的进行缓存更新操作

A. key通配处理：

1. 如 `gameinfo_by_name_{game_name}`,括号中的`game_name`,对应的就是表中的字段名字，如果条件中有 `game_name=lol` 这样的条件，就会将key变成 `gameinfo_by_name_lol`,并触发缓存动作（写缓存，或者更新缓存）
2. 一个key里面可以包含多个`{}`,如: `user_info_type_{type}id{id}`
3. 使用的时候要注意，如果通配符字段未完全匹配（必须是配置`game_name='lol'`，且所有通配符字段必须匹配上），则该条缓存key不触发任何条件
4. 在reids的驱动下，可以支持`*`,`?`等通配符的支持
5. 注意，`*`,`?`和`{}`,不能组合使用
6. 不带通配符时，100%匹配，eg：`gameinfoList`

B. 通配符触发情况

1. `model->setCacheKeys(key);`
 - 仅支持传入字符串，如: `gameinfo_by_name_{game_name}`
 - 在紧接着的一个`$model->get()`中，触发缓存是否缓存，匹配则缓存，不匹配，则抛弃。不带通配符的时候，100%匹配。
 - `setCacheKeys` 缓存优先级，高于`auto`缓存
 - 匹配条件从`where`中获取
 - 匹配条件再从`columns`中获取
 - `where` 和 `columns` 分别匹配，不混合匹配
 - 注意，该方法不能处理分页缓存，如果需要分页缓存，可以直接在在`model`中，开启自动分页缓存配置`$this->startAutoPageCache(5);`
2. `$model->setAfterUpdateFlushKey([]);`
 - 传值支持字符串或者数组，eg: `['gameinfo_by_name_{game_name}']` or `'gameinfo_by_name_{game_name}'`
 - 每次，在执行完`update`操作，都会尝试匹配，如果匹配成功，则触发删除该缓存操作
 - 匹配条件从`where`中获取
 - 匹配条件再从`columns`中获取
 - `where` 和 `columns` 分别匹配，不混合匹配
3. `$model->setAfterDeleteFlushKey([]);`
 - 传值支持字符串或者数组，eg: `['gameinfo_by_name_{game_name}']` or `'gameinfo_by_name_{game_name}'`
 - 每次，在执行`delete`后操作（注意，这里实际执行步骤，在`delete`真正执行之前），都会尝试匹配，如果匹配成功，则触发删除该缓存操作
 - 匹配条件从`where`中获取

4. `$model->setAfterInsertFlushKey([]);`

- 传值支持字符串或者数组，eg: ['gameinfoList'] or 'gameinfoList'
- 每次，在执行Inserter后操作，都会尝试匹配，如果匹配成功，则触发删除该缓存操作。不带通配符的时候，100%匹配。
- 匹配条件从columns中获取

5. *model* -> *setFlushKeys*([flushData]);

- 传值支持字符串或者数组，eg: ['gameinfoList'] or 'gameinfoList'
- 相当于分给 setAfterUpdateFlushKey, setAfterDeleteFlushKey, setAfterInsertFlushKey写入了[\$flushData]值

5. 提供常用sql封装方法，方便代码书写

A. `getOne()`：通过主键查找数据，如果开启原子化缓存会自动处理缓存（测试通过）

```
$model->getOne($id);
```

B. `getOne`：通过条件查找数据（测试通过）

```
param array $where eg:['id'=>1,'file'=>2]
```

```
param array $orderBy eg : ['id'=>'desc']
```

```
$model->findOne(['username'=>'testname'], ['age'=>'desc']);
```

C. `getListUpgraded`：得到列表数据，建议使用

param array \$where 查询条件 eg: ['name'=>'test']

param array \$predicate 断言条件

eg :

```
[
    'groupBy'=>['name'], #groupBy 的 value,支持字符串 'name' , 或者数组 [ 'name', 'sex' ]
    'having'=>['name'], #必须是数组[$column, $operator = null, $value = null, $boolean = 'and'
    'orderBy'=>['name'], # orderBy 的 value为数组 ['id'=>'desc']
    'skip'=>0, #起始值, 从0开始
    'take'=>20, #获得的条数
]
```

param string/array fields 查询字段, 字符串或者数组

```
$where = [
    'client_type' => $clientType,
    'real_time' => ['between', [$startDay, $endDay]]
];
if( !empty($provCode) && 'all' != $provCode){
    $where['prov_code'] = $provCode;
}
if( !empty($cityCode) ){
    $where['city_code'] = $cityCode;
}

$fields = [
    DB::raw('DATE_FORMAT(real_time, \'%Y-%m-%d\') as show_date') ,
    DB::raw('count(id) as kdump_num'),
    DB::raw('count(DISTINCT gid) as kdump_gid_num')
];
$predicate = [
    'groupBy' => 'show_date'
];
return $this->KDumpModels->getListUpgraded($where, $predicate, $fields);
```

D. getList: 得到列表数据, 不建议使用

向下兼容方法, 不建议时间, 具体使用方法, 请看源码

E. getPagedListUpgraded:获得分页数据高级方法，建议使用

param array \$where 查询条件 eg: ['name'=>'test']

param int \$pageNum 每页展示条数

param array \$predicate 断言条件，同 getListUpgraded

param string/array fields 查询字段，字符串或者数组

F. getPagedList:获得分页数据,不建议使用

G. saveInfo:保存/修改方法

param array saveInfo 需要保存或者修改的值，自动判断主键，自动清理缓存

H. delCleanCache: 条件删除，同时删除自动生成的缓存

执行流程是，先查找出id，再按照id删除

I. del:根据条件删除数据

直接删除