第 5 章

◂ 数据库操作层 ▸

作为 MVC 中三大组成部分之一的模型层，重要程度不言而喻，无论是最终数据的持久化或者说业务逻辑组织，都是由 model 层来完成的。需要说明的是本书中的数据库操作层也叫 DAO（Data Access Object）层，只用来进行底层数据库操作（如增删改查），并不涉及业务上的处理。DAO 层的出现有利于将业务逻辑和底层数据库操作分离，便于代码解耦以及后期维护。而模型层是相对比较高级的一层，通过将数据库字段映射为 PHP 的类属性来实现，使用模型操作数据库时，其实并不需要写 SQL 相关的代码，一般当作普通对象实例化操作即可，可以屏蔽底层数据库的差异，让数据库操作像类操作一样简单易用。

ThinkPHP 5 中的数据库操作层实现大致和 ThinkPHP 3.2 一致，基于驱动类设计，可以在不更改代码的情况下平滑切换数据库，不得不说，这一点做的确实精彩！开发应用时刚开始可能会直接使用 MySQL 数据库，待项目做大之后可能就会考虑 SQLServer、Oracle 之类的数据库了，这时直接修改配置即可切换数据库，是不是很方便呢？

5.1 数据库配置

ThinkPHP 5 中数据库配置支持方式比较多，本书只列举两种常用的，防止读者在实际应用中不知道该如何选择何种配置。

- database.php 定义

database.php 默认在 application/database.php 文件中，推荐配置如下：

```
return [
    // 数据库类型
    'type'           => 'mysql',
    // 服务器地址
    'hostname'       => '127.0.0.1',
    // 数据库名
    'database'       => 'thinkphp',
    // 用户名
    'username'       => 'root',
    // 密码
```

```
    'password'        => 'root',
    // 端口
    'hostport'        => 3306,
    // 连接 dsn
    'dsn'             => '',
    // 数据库连接参数
    'params'          => [
        PDO::ATTR_EMULATE_PREPARES => 0,
        PDO::ATTR_ERRMODE          => PDO::ERRMODE_EXCEPTION
    ],
    // 数据库编码默认采用 utf8
    'charset'         => 'utf8mb4',
    // 数据库表前缀
    'prefix'          => 'think_',
    // 数据库调试模式
    'debug'           => true,
    // 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)
    'deploy'          => 0,
    // 数据库读写是否分离主从式有效
    'rw_separate'     => false,
    // 读写分离后主服务器数量
    'master_num'      => 1,
    // 指定从服务器序号
    'slave_no'        => '',
    // 自动读取主库数据
    'read_master'     => false,
    // 是否严格检查字段是否存在
    'fields_strict'   => true,
    // 数据集返回类型
    'resultset_type'  => 'array',
    // 自动写入时间戳字段
    'auto_timestamp'  => false,
    // 时间字段取出后的默认时间格式
    'datetime_format' => 'Y-m-d H:i:s',
    // 是否需要进行 SQL 性能分析
    'sql_explain'     => false,
];
```

- 模型定义

有时候应用开发中会使用到多个数据库，这时如果手动选择数据库实际上是不太方便的，好在 ThinkPHP 5 允许我们在模型声明中指定数据库连接，比如有如下模型：

```
class User extends Model {
```

```
    protected $connection = 'user';
}
```

当我们在使用 User 模型时，系统会自动读取 user 连接定义来连接数据库。

5.2 基本操作

数据库操作离不开 CURD（Create/Update/Read/Delete，俗称增删改查）。ThinkPHP 5 的 DAO 基本操作如下：

上文中提到过 DAO 层是相对比较底层的，所以需要手写 SQL。ThinkPHP 5 中 Db 类负责底层 SQL 操作，该类会自动读取默认的数据库连接信息，当然你也可以手动指定数据库配置来在特定数据库执行 SQL 语句。

（1）添加数据

```
Db::execute('INSERT INTO user (username, password) VALUES
(?,?)',['admin',md5('111111')]);
```

（2）更新数据

```
Db::execute(`UPDATE user SET password=? WHERE
username=?`,[md5('123456'),'admin']);
```

（3）删除数据

```
Db::execute('DELETE FROM user WHERE username=?',['admin']);
```

（4）查找数据

```
Db::query('SELECT * FROM user WHERE username=?',['admin']);
```

在上面的示例中，不知道各位读者有没有发现不同？其实数据库操作主要有两种：数据操作和数据查询。这两种操作的返回结果是不同的，数据查询中返回的是数据列表，而数据操作中返回的是受影响行数，所以在使用时需要区分 execute 和 query。

SQL 语句中的“?”是占位符，是为了解决 SQL 注入问题而出现的，早期 PHP 开发者使用 mysql_系列函数操作数据库时都是手写 SQL，有极大的安全风险。而使用 SQL 占位符之后可以规避这种风险，使用也很简单，“?”的顺序和后面数组参数的顺序一一对应。

（5）在特定数据库指定操作

上文中提到了 Db 类使用默认的数据库连接来操作，而如果想使用其他库的话需要传入到 config 方法中，代码如下：

```
Db::config($connection)->query('SELECT * FROM user');
```

5.3 使用查询构造器

在 5.2 节的内容中演示了如何通过原生 SQL 操作数据库，在实际的开发过程中，这种操作用得比较少，原因是手写原生 SQL 不太方便，容易写错，而 ThinkPHP 5 为了方便开发者提供了查询构造器。

查询构造器使用 Builder（建造者模式，设计模式的一种）设计，而建造者模式最大的特点就是支持链式调用。如下示例就是链式操作的一种：

```
A::b()->c()->d();
```

其实实现起来也不难，只要在 b/c/d 方法执行结束后 return 当前类实例即可。

查询构造器的基本使用（以 CURD 为例）在后面介绍。

5.3.1 添加数据

- 添加一条数据

```
Db::table('user')->insert(['username'=>'admin','password'=>md5('111111')]);
```

- 添加多条数据

```
Db::table('user')->insertAll([
    ['username'=>'admin','password'=>md5('111111')],
    ['username'=>'admin1','password'=>md5('123456')]
]);
```

5.3.2 更新数据

- 根据指定条件更新数据

```
Db::table('user')->where('username','admin')->update(['password'=>md5('123456')]);
```

- 待更新数据中包含主键更新数据

```
Db::table('user')->update(['admin_id'=>1,'password'=>md5('123456')]);
```

- 更新指定字段

```
Db::table('user')->where('username','admin')->setField('password',md5('111111'));
```

- 自增一个字段（比如文章增加点击量）

```
Db::table('article')->where('article_id',1)->setInc('hit',1);
```

- 自减一个变量（比如扣除库存）

```
Db::table('user')->where('goods)Id,1)->setDec('stock',1);
```

5.3.3 查询数据

- 查询一条数据

```
Db::table('user')->where('username','admin')->find();
```

- 查询多条数据

```
Db::table('user')->where('sex','male')->select();
```

- 查询指定记录的某个字段值

```
Db::table('user')->where('username','admin')->value('last_login'
);
```

- 查询某一列数据

```
Db::table('user')->where('sex','male')->column('user_id');
```

- 批量查询

当数据库数据比较多而服务器内存有限制时可以使用，通过循环来降低资源占用：

```
Db::table('user')->where('sex','male')->chunk(100,function($user
s){
    print_r($users);
},'user_id','desc');
```

- JSON 查询

MySQL5.7 中新增了 JSON 类型，作用和 MongoDB 类似，可以存储非结构化数据。ThinkPHP 5 同样也支持 JSON 类型的查询中。在如下示例中，params 为 JSON 类型：

```
Db::table('user')->where('params$.phone','13333333333')->find();
```

请注意 params 后的$号。

5.3.4 删除数据

- 根据主键删除

```
Db::table('user')->delete([1,2,3]);
```

- 根据 where 删除

```
Db::table('user')->where('username','admin')->delete();
```

5.4 查询语法

5.4.1　查询表达式和查询方法

大部分数据操作中都会应用到查询，不管是 Select、Update 还是 Delete。Update 或 Delete 需要更新或删除指定条件的行。ThinkPHP 为我们提供了强大而又简便的查询语法。

查询语法由查询表达式以及查询方法组成，比如：

```
    Db::table('user')->where('username','admin')->where('created_at'
,'>',1500000000);
```

可以看到 username='admin'和 created_at > 1500000000 为查询表达式。两个表达式通过 AND 连接，表示两个条件必须同时满足才会被查询到。

ThinkPHP 提供了 where 和 whereOr 来进行查询表达式的连接，where 通过 AND 连接，whereOr 通过 OR 连接。

ThinkPHP 的查询表达式基本是 SQL 标准表达式，与 SQL 对应关系如表 5-1 所示。

表 5-1　ThinkPHP 查询表达式

ThinkPHP 表达式	SQL 表达式	说明
=	=	等于
<>	<>	不等于
>	>	大于
>=	>=	大于等于
<	<	小于
<=	<=	小于等于
between/not between	between/not between	区间查询
in/not in	in/not in	列表查询
null/not null	is null/is not null	NULL 查询
exists/not exists	is exists/is not exists	存在性查询
like	like	模糊查询
exp	-	表达式查询（ThinkPHP 特有）

where 和 whereOr 接收 3 个参数，其中第 3 个为可选参数。

当传入 2 个参数时，如下代码所示：

```
where('username','admin');
```

可以理解为 username='admin'。

当传入的第 2 个参数为 null 时，如下代码所示：

```
where('username',null);
```

可以理解为 username is null。注意，MySQL 没有字段名=null 这种语法。

当传入 3 个参数时，如下代码所示：

```
where('age','>',18);
```

可以理解为 age > 18。

可以看到传入 2 个参数时实际上操作符为=/is（null/exists 等操作符）。

5.4.2 查询表达式示例

本节只简单介绍一下复杂一点的表达式，简单的操作符各位读者可以对照表格测试。

- between

```
where('age','between',[18,24]);
```

查询年龄从 18（含）到 24（含）的数据。

- in

```
where('role','in',['admin','super_admin']);
```

查询管理员和超级管理员角色。

- like

```
where('name','like','%张三%');
```

查询姓名包含'张三'的数据。

- exp

```
where('age','exp','between 18 and 24');
```

查询年龄从 18（含）到 24（含）的数据，表达式查询可以应对复杂情况下的查询，但使用时需要小心，容易发生 SQL 注入风险。

5.5 连贯操作

当你看到如下代码是否会感到很神奇呢？

```
Db::table('user')->field('id,username')->where('username','admin
')->order('id desc')->limit(10)->select();
```

这个在 ThinkPHP 中被称为连贯操作，在执行最后的方法前之间的方法都可以继续调用查询方法。实现原理其实很简单：

```
class Db {
    field() {
        // … 选择字段
        return $this;
    }
    where() {
        // …查询条件
        return $this;
    }
}
```

可以看到每个连贯方法返回的都是$this 对象，保证了后续调用，有兴趣的读者可以将这个模式应用到其他开发活动当中。该模式在设计模式中被称为建造者模式。

ThinkPHP 支持的连贯操作如表 5-2 所示。

表 5-2　ThinkPHP 支持的连贯操作

操作名	说明
table	指定要操作的数据表名称
alias	给数据表定义别名
field	查询指定字段，可多次调用
order/orderRaw	查询结果排序，可多次调用
limit	限定结果集长度
group	分组查询
having	筛选结果集
join	关联查询，可多次调用
union	联合查询，可多次调用

（续表）

操作名	说明
view	视图查询
distinct	查询非重复数据
relation	关联查询，可多次调用
page	分页查询（框架实现，非 SQL 语法）
lock	数据库锁
cache	缓存查询（框架实现，非 SQL 语法）
with	关联查询预处理，可多次调用
bind	数据绑定，一般配合占位符
strict	是否严格检测字段名存在性
master	读写分离环境下从主服务器读取数据
failException	未查询到数据时是否抛出异常
partition	数据库分表查询（框架实现，非 SQL 语法）

5.6 连贯操作示例

5.6.1 table

- 一般使用

```
Db::table('user')->find();
// SELECT * FROM `user` LIMIT 1;
```

- 使用表前缀（假设表前缀为 think_）

```
Db::table('__USER__')->find();
// SELECT * FROM `think_user` LIMIT 1;
```

- 指定数据库名

```
Db::table('think.user')->find();
// SELECT * FROM `think`.`user` LIMIT 1;
```

5.6.2　alias

```
    Db::table('__USER__')->alias(['think_user'=>'user','think_post'=
>'post'])->join(['think_user'=>'user'],'post.user_id=user.user_id')
->select();
    // SELECT * FROM `think_user` `user` INNER JOIN `think_post` `post`
ON `post`.`user_id`=`user`.`user_id`
```

5.6.3　field

- 一般使用

```
    Db::table('user')->field(['username','password'])->find();
    // SELECT `username`,`password` FROM `user` LIMIT 1;
```

- 字段别名

```
    Db::table('user')->field(['nickname'=>'realname'])->find();
    // SELECT `nickname` as `realname` FROM `user` LIMIT 1;
```

- 使用 SQL 表达式（一般用于统计查询，当然，所有 SQL 表达式都支持）

```
    Db::table('user')->field(['SUM(amount)'=>'amount'])->find();
    // SELECT SUM(`amount`) as `amount` FROM `user` LIMIT 1;
```

- 查询字段排除（一般用来排除 TEXT 类型的大字段）

```
    Db::table('article')->field('content',true)->find();
    // SELECT `article_id`,`title`,`desc` FROM `article` LIMIT 1;
```

- 安全写入

```
    Db::table('user')->field(['email','phone'])->insert($data);
```

不管客户端提交什么样的数据，ThinkPHP 只会接收 email 和 phone 两个字段，防止修改其他敏感字段。

5.6.4　order/orderRaw

- 一般使用

```
    Db::table('user')->order(['age'=>'desc','user_id'=>'desc'])->sel
ect();
    // SELECT * FROM `user` ORDER BY `age` DESC, `user_id` DESC;
```

- 使用表达式（常见于乱序查询）

```
    Db::table('user')->orderRaw('RAND()')->select();
    // SELECT * FROM `user` ORDER BY RAND();
```

5.6.5 limit

- 一般使用

```
Db::table('user')->limit(10)->select();
// SELECT * FROM `user` LIMIT 10;
```

- 指定起始行

```
Db::table('user')->limit(100,100)->select();
// SELECT * FROM `user` LIMIT 100,100;
```

- 写入数据时限定

```
Db::table('user')->where('sex','female')->limit(1)->delete();
DELETE FROM `user` WHERE `sex`='female' LIMIT 1;
```

5.6.6 group

```
Db::table('exam`)->field(['user_id','SUM(score)'=>'score'])->gro
up('user_id')->select();
// SELECT `user_id`,SUM(`score`) `score` FROM `exam` GROUP BY
`user_id`;
```

5.6.7 having

```
Db::table('exam`)->field(['user id','SUM(score)'=>'score'])->gro
up('user id')->having('score>=60')->select();
// SELECT `user id`,SUM(`score`) `score` FROM `exam` GROUP BY
`user_id` HAVING `score`>=60;
```

5.6.8 join

join 方法原型：

```
join($join [,$condition=null [, $type='INNER']])
```

type 支持 INNER/LEFT/RIGHT/FULL。

- 一般使用

```
Db::table('think user')->join(['think post','think post.user id=
think user.user id'])->select();
// SELECT * FROM `think user` INNERT JOIN `think post` ON
`think_post`.`user_id`=`think_user`.`user_id`;
```

- 多表关联

```
Db::table('think user')->alias('user')->join([
['think article article','article.user id=user.user id'],
['think comment comment','comment.user id=user.user id']
])->select();
```

```
    // SELECT * FROM `think user` `user` INNER JOIN `think article`
`article` ON `article`.`user id`=`user`.`user id` INNER JOIN
`think_comment` `comment` ON `comment`.`user_id`=`user`.`user_id`
```

5.6.9　union

- 字符串/数组方式

```
    Db::table('user')->field(['name'])->union([
    'SELECT name FROM user1',
    'SELECT name FROM user2'
    ])->select();
    // SELECT `name` FROM `user` UNION SELECT `name` FROM `user1` UNION
SELECT `name` FROM `user2`;
```

- 闭包方（不了解闭包的可以查看 PHP 官方文档 http://www.php.net/manual/zh/functions.anonymous.php）

```
    Db::table('user')->field(['name'])->union(function($query){
    $query->table('user1')->field(['name']);
    })->union(function($query){
    $query->table('user2')->field(['name']);
    })->select();
    // SELECT `name` FROM `user` UNION SELECT `name` FROM `user1` UNION
SELECT `name` FROM `user2`;
```

- union all 方式

```
    Db::table('user')->field(['name'])->union(['SELECT name FROM
user1', true])->select();
    // SELECT `name` FROM `user` UNION ALL SELECT `name` FROM `user1`
UNION SELECT `name` FROM `user2`;
```

5.6.10　distinct

```
    Db::table('user')->distinct(true)->field(['username'])->select();
    // SELECT DISTINCT `username` FROM `user`
```

5.6.11　page

page 是 ThinkPHP 框架实现的一个方法，用来简化 limit 方法的计算。

```
    Db::table('user')->page(1,10)->select();
    // SELECT * FROM `user` LIMIT 0,10
```

5.6.12　lock

为了保证高并发条件下数据写入一致性，SQL 提供了锁机制。锁可分为共享锁和独占锁，对于锁的说明有兴趣的读者可以在网上查阅相关资源。

```
    Db::table('user')->where('user_id',1)->lock(true)->find();
    // SELECT * FROM `user` WHERE `user_id`=1 FOR UPDATE
```

`FOR UPDATE`是 MySQL 的锁语法，只有成功取得锁的客户端才能操作该数据。

lock 方法支持传入 SQL 表达式来满足一定特定环境下的锁要求：

```
    Db::table('user')->where('user_id',1)->lock('lock in share
mode')->find();
    // SELECT * FROM `user` WHERE `user_id`=1 LOCK IN SHARE MODE
```

5.6.13 cache

由 ThinkPHP 框架实现的方法，在缓存有效期内直接返回缓存数据，多用于 CMS 系统内文章查询的缓存。

- 一般使用

```
    Db::table('user')->cache(60)->find();
    // 缓存一分钟
```

- 指定缓存 key（方便外部调用）

```
    Db::table('user')->cache('tmp_user',60)->find();
    Cache::get('tmp_user'); // 读取缓存
```

- 缓存清除（使用主键更新数据时无须指定缓存 key）

```
    Db::table('user')->update(['user_id'=>1,'name'=>'demo']);
```

- 缓存清除（手动指定 key）

```
    Db::table('user')->cache('tmp_user')->where('user_id',1)->update
(['name'=>'demo']);
```

5.6.14 relation

将在关联模型中进行详细介绍，本节暂时略过。

以上操作符都是实际项目中非常常见的操作符，在连贯操作表格中出现的其他操作符在实际操作中用得比较少，有兴趣的读者可以参考官方文档。

5.7 查询事件与 SQL 调试

5.7.1 查询事件

ThinkPHP 5 新增部分，能够允许我们在执行数据库操作前后进行一些事件监听和操作。

- before_select
- before_find
- after_insert
- after_update
- after_delete

使用闭包来注册事件监听器：

```
Query::event('after_delete',function($options,$query){
// 数据删除后调用
});
```

5.7.2　SQL 调试

通过调用 Db::listen 方法来监听 SQL 语句、执行时间、explain 执行计划等。

```
Db::listen(function($sql,$time,$explain,$isMaster){
});
```

5.7.3　事务

当需要同时操作多表且需要保证其一致性时，需要使用事务操作。ThinkPHP 5 的事务操作也是基于闭包来操作的。

```
Db::transaction(function(){
Db::table('user')->insert($data);
Db::table('user_profile')->insert($otherData);
});
```

当闭包函数抛出异常时事务会自动回滚，无异常时事务自动提交，解决了以往手动捕获异常回滚事务的问题。

5.7.4　调用存储过程或函数

使用 Db::query 传入原生 SQL 查询即可，支持参数绑定，比如如下查询：

```
Db::query('call demo_query(?)',[1]);
```

当然在实际开发过程中不推荐存储过程，原因如下：

（1）不利于迁移（迁移数据时容易遗漏存储过程的迁移）。
（2）跨数据软件的兼容问题（每个 DBMS 支持的存储过程语法有差异性）。

第 6 章

◂ 模型层 ▸

模型层（Model）是对 DAO 层的上层包装，基于对象关系映射来使得数据库操作像对象操作一样简单方便。

6.1 模型定义

```
namespace app\index\model;

use think\Model;

class User extends Model {
protected $pk = 'user_id'; // 主键，框架默认自动识别，也可以手动指定
protected $table = 'think_user'; // 指定数据表
protected $connection = 'db2'; // 指定数据库连接
}
```

数据表识别规则，表前缀+大驼峰，遇到下划线时将首字母大写，示例如下：

（1）数据表前缀 think_

```
User: think_user
UserArticle: think_user_article
```

（2）数据表为空

```
User: user
UserArticle: user_article
```

接下来基于 CURD 来介绍采用模型的方式操作数据库。

6.2 插入数据

（1）对象方式

```
$user = new User();
```

```
$user->username = 'demo';
$user->password = md5('111111');
$user->email = 'demo@demo.com';
$user->created_at = time();
$user->save;
```

（2）数组方式

```
$user = new User();
$user->data([
    'username' => 'demo',
    'password' => md5('111111'),
    'created_at' => time()
]);
$user->save();
```

6.3 更新数据

（1）使用查询条件修改

```
$user = new User();
$user->save([
    'password' => md5('123456')
],[
    'username' => 'demo'
]);
```

（2）基于对象修改

```
$user = User::get(['username' => 'demo']);
$user->password = md5('111111');
$user->save();
```

6.4 批量更新（只支持主键）

```
$user = new User();
$user->saveAll([
    ['user_id' => 1, 'password' => md5('111111')],
    ['user_id' => 2, 'password' => md5('123456')]
]);
```

6.5 删除数据

（1）基于对象删除

```
$user = User::get(['username' => 'demo']);
$user->delete();
```

（2）基于主键删除

```
User::destroy(1); // 删除一个
User::destroy([1,2,3]); // 删除多个
```

（3）条件删除

```
User::where('user_id',1)->delete();
```

6.6 查询数据

（1）主键查询

```
$user = User::get(1);
```

（2）指定字段查询（数组方式）

```
$user = User::get(['username' => 'demo']);
```

（3）where 查询（类似 ThinkPHP 3.2）

```
$user = User::where('user_id',1)->find();
```

（4）闭包查询

```
$user = User::get(function($q){
    $q->where('user_id',1);
});
```

（5）指定字段查询（通过 PHP 魔术方法自动识别字段）

```
$user = User::geByUsername('demo');
```

6.7 批量查询

（1）基于主键的批量查询

```
$users = User::get([1,2,3]);
```

（2）基于字段的批量查询

```
$users = User::get(['sex' => 'female']);
```

（3）基于条件的查询

```
$users = User::where('sex','female')
->page(1,10)
->order(['user_id'=>'desc'])
->select();
```

6.8 聚合查询

ThinkPHP 5 目前支持 count/max/min/avg/sum 聚合查询，下面演示其中一种：

```
$avgScore = Score::where('score','>=',60)->avg('score');
// 计算及格人的平均分
```

6.9 get/set

get/set 方法用来覆盖 ThinkPHP 5 处理动态字段的方法，比如数据库有 username 字段但是模型类并没有定义 username 属性，然而我们可以使用$user->username 这种代码，实际上就是 ThinkPHP 5 接管了 PHP 的魔术方法__get 和__set。

get 方法声明如下：

```
get 属性名 Attr($value,$data);
// $value 为当前属性的值，也就是$data[属性名]
// $data 为当前模型对应的数据数组
// 当属性名不存在时，忽略$value，直接从$data 取值即可
```

如下代码声明一个状态说明属性（数据库只有 status）：

```
namespace app\index\model;

use think\User;

class User extends Model {
    public function getStatusDesc($value, $data) {
    return [0=>'正常',-1=>'被封禁'][$data['status']];
}
}
```

```
echo $user->status_desc;
```

（1）获取所有读取器的值

读取器可以让我们在读取到原始数据之后进行进一步处理再返回给调用层。单个属性的读取是惰性求值（访问时才调用读取器方法，刚从数据库查出来不会主动调用）。如果需要读取所有读取器的值，需要使用 toArray 方法。例如（接上例）：

```
print_r($user->toArray());
```

（2）获取所有数据库值

有个尴尬的事情是当我们使用了读取器后，某些情况下也需要访问数据库值，就需要使用到 getData 方法了。例如：

```
$status = $user->getData('status'); // 返回数据库中保存的 status 值
$data = $user->getData(); // 返回数据库该行记录的数组
```

set 方法声明与 get 类似：

```
public function set 属性名 Attr($value, $data);
```

但是 set 要求属性名存在于数据库中。

比如以下代码将提交的用户名首字母大写后再写入数据库：

```
public function setUsername($value, $data) {
    return ucwords($value);
}
```

6.10 自动时间戳处理

以往写入数据时时间戳都需要手动给字段赋值，而 ThinkPHP 5 已经自动帮你完成这一步，开发者只需要定义相关配置即可。

（1）配置文件方式。在 database.php 中添加：

```
'auto_timestamp' => true
```

（2）模型定义。在具体模型文件中添加：

```
'protected $autoWriteTimestamp = true;'
```

auto_timestamp 取值为 true/'datetime'/'timestamp'，分别对应 int/datetime/timestamp 数据库类型。

默认的时间戳字段为 create_time 和 update_time，像上文中的 created_at 字段需要在模型文件中做如下定义：

```
class User extends Model {
```

```
    protected $createTime = 'created_at';
    protected $updateTime = 'updated_at';
}
```

如果只需要 createTime 而不需要 updateTime 字段（适用于数据不更新的情况），那么将 $updateTime 属性置为 false 即可。

需要注意的是，由于时间戳字段 ThinkPHP 5 内置了读取器，所以取值的时候会变成配置文件中'datetime_format'定义的格式，一般为'Y-m-d H:i:s'，如果不需要该配置，将'datetime_format'配置为 false 即可。

6.11 只读字段

为了避免数据操作时更新到原本不该更新的字段，比如将用户表的用户名给更新了，需要使用到只读字段，很简单，在模型中定义$readonly 即可。

```
class User extends Model {
    protected $readonly = ['username'];
}
```

6.12 软删除

该功能和 Laravel 的类似，通过在数据表中添加一个 deleted_at 的字段来标记删除，当值为 null 时该数据未删除，当值不为 null 时标记为删除时间来保护一些重要数据可以在需要的时候恢复。

使用方式与 Laravel 一致，基于 PHP 的 trait 实现，对于 PHP trait 不明白的读者可以查看官网文档（http://php.net/traits）。

```
class User extends Model {
    use SoftDelete;
    protected $deleteTime = 'deleted_at';
}
```

当我们调用 destroy 方法或者 delete 方法时默认就是软删除，如果需要硬删除（从数据库删掉），需要传入额外参数：

```
User::destroy([1,2,3],true);
$user = User::get(1);
$user->delete(true);
```

查询时默认不包括软删除数据，如果需要包含软删除数据，请使用如下查询：

```
User::withTrashed()->select();
User::withTrashed()->find();
```

只查询软删除数据：

```
User::onlyTrashed()->select();
User::onlyTrashed()->find();
```

6.13 自动完成

自动完成跟 ThinkPHP 3.2 相比变化不算太大，编码方式有变更而已，思路不变。ThinkPHP 5 依旧支持 auto/insert/update 三种场景，auto 包含 insert/update。

自动完成用来在模型保存的时候自动写入数据，与修改器不同的是，修改器需要手动赋值，只不过赋值的时候我们可以做一下处理，而自动完成不需要手动赋值。

自动完成的代码示例如下：

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected $auto = [];
    protected $insert = ['created ip','created ua'];
    protected $update = ['login ip', 'login at'];

    protected function setCreatedIpAttr() {
        return request()->ip();
}

protected function setCreatedUaAttr() {
        return request()->header('user-agent');
}

protected function setLoginIpAttr() {
        return request()->ip();
}

protected function setLoginAtAttr() {
        return time();
}
}
```

在插入数据时系统会自动填充 created_ip 和 created_ua 字段，在更新数据时系统会自动填充 login_ip 和 login_at 字段。

自动完成和修改器的区别在于：修改器需要手动赋值，自动完成不需要！

6.14 数据类型自动转换

由于 PHP 是弱类型语言，因此容易引发一些问题，比如客户端提交的表单是数字，但是 PHP 接收到的是字符串，或者说客户端提交的是 JSON 数组，存储到数据库需要手动 json_encode 一下。ThinkPHP 5 的数据类型自动转化就是为了解决该问题而产生的，该功能同 Laravel 模型的 cast 相似，通过配置式的定义来替换硬编码。

ThinkPHP 5 支持的数据转换类型如表 6-1 所示（PDO:: ATTR_EMULATE_PREPARES 为 true 时数据库总会返回字符串形式的数据，哪怕字段定义是其他类型）。

表 6-1　ThinkPHP 5 支持的数据转换类型

ThinkPHP 类型	转换操作
integer	写入/读取时自动转换为整数型
float	写入/读取时自动转换为浮点型
boolean	写入/读取时自动转换为布尔型
array	写入时转换为 JSON，读取时转换为 array
object	写入时转换为 JSON，读取时转换为 stdClass
serialize	写入时调用 serialize 转换为字符串，读取时调用 unserialize 转换为转换前类型
json	写入时调用 json_encode，读取时调用 json_decode
timestamp	写入时调用 strtotime，读取时调用 date，格式默认为"Y-m-d H:i:s"，通过模型$dateFormat 属性自定义

自动类型转换示例如下：

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected $type = [
        'status' => 'integer',
        'balance' => 'float',
        'data' => 'json'
];
}
```

```
$user = new User();
$user->status = '1';
$user->balance = '1.2';
$user->data = ['name'=>1];
$user->save();
var_dump($user->status, $user->balance, $user->data);
// int(1) float(1.2) array(size=1) 'name'=>int(1)
```

6.15 快捷查询

模型层可以将常用的或者复杂的查询定义为快捷查询来提高代码复用率。快捷查询定义如下：

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected function scopeMale($query) {
        $query->where('sex','male');
}

protected function scopeAdult($query) {
        $query->where('age','>=',18);
}
}
```

调用代码如下：

```
User::scope('male')->select(); // 查找所有男性
User::scope('adult')->select(); // 查找所有成年人
User::scope('adult,age')->select(); // 查找所有成年男性
```

6.16 全局查询条件

全局查询条件用来解决查询的数据需要有基础条件的情形，比如之前讲过的软删除就要求任何查询默认包含 deleted_at 为空的条件（手动查询被删除数据的除外）。这时就需要全局查询条件来解决问题，否则要在每个查询中手动添加一个 deleted_at 为空的查询条件。

全局查询条件代码和快捷查询类似，代码如下：

```
namespace app\index\model;
```

```
use think\Model;

class User extends Model {
    protected function base($query) {
        $query->where('deleted_at', null);
}
}
```

之后使用任何 User 模型的查询都会自动添加 deleted_at 为空的约束，如果需要显示关闭该约束，可以使用如下代码：

```
User::useGlobalScope(false)->select(); // 关闭全局查询条件
User::useGlobalScope(true)->select(); // 开启全局查询条件
```

6.17 模型事件

模型事件是在通过模型写入数据时触发的事件，使用 DAO 层操作数据不会触发。ThinkPHP 5 支持的模型事件如下：

- before_insert：插入前。
- after_insert：插入后。
- before_update：数据更新前。
- after_update：数据更新后。
- before_write：写入前。
- after_write：写入后。
- before_delete：删除前。
- after_delete：删除后。

通过模型的静态方法 init 进行注册，注册代码如下：

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected function init() {
        User::beforeInsert(function($user){
        if($user->age<=0) {
            return false;
}
        return true;
```

```
        });
    }
}
```

所有 before_*事件回调函数中返回 false，将导致后续代码不会继续执行！

6.18 关联模型

关系型数据库最重要的就是实体的划分以及关系的确定，好的关联关系可以让数据表减少冗余，加快查询速度。ThinkPHP 5 对关联模型的支持非常完善，可以让开发者使用很少的代码实现强大的关联功能。

关联关系有一对一、一对多、多对多。

6.18.1 一对一关联

一对一关联比较好理解，比如每个用户都会有一个钱包，那么用户和钱包之间就是一对一关系。

ThinkPHP 5 使用 hasOne 来定义一对一关联，hasOne 原型如下：

```
hasOne('模型类名','外键名','主键名','JOIN 类型='INNER'')
```

模型定义如下：

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected function wallet() {
        return $this->hasOne('Wallet','wallet_id','wallet_id');
    }
}
$user = User::get(1);
echo $user->wallet->balance; // 输出钱包余额
$user->wallet->save(['balance'=>1]);//保存钱包余额
```

6.18.2 一对一关联模型数据操作

使用关联模型后，数据保存也是关联式的，不用再手动进行关联数据保存。ThinkPHP 5 使用 together 方法进行关联数据的操作。

还是以上面的钱包为例：

- 新增数据

```
$user = new User();
$user->realname = 'demo';
$wallet = new Wallet();
$wallet->balance = 100;
$user->wallet = $wallet;
$user->together('wallet')->save();
```

- 更新数据

```
$user = User::get(1);
$user->realname = '姓名';
$user->wallet->balance = 200;
$user->together('wallet')->save();
```

- 删除数据

```
$user = User::get(1);
$user->together('wallet')->delete();
```

6.18.3　一对一从属关联

从属关联属于特殊的关联关系，钱包属于用户这种是一对一的，而文章从属于用户是多对一的，即多篇文章可以从属于一个用户。

ThinkPHP 使用 belongsTo 定义从属关系，belongsTo 原型如下：

```
belongsTo('模型类名','外键名','关联表主键名','join 类型='INNER'')
```

模型定义如下（默认外键是表名*_id*，如下例子外键默认为 *user_id*）：

```
namespace app\index\model;

use think\Model;

class Wallet extends Model {
    protected function user() {
        return $this->belongsTo('User');
    }
}
$wallet = Wallet::get(['user_id'=>1]);
echo $wallet->user->realname;// 打印钱包所有者姓名
```

6.18.4　一对多关联

一对多关联也比较常见，比如一个用户有多篇文章，每篇文章只可能属于一个用户。

ThinkPHP 5 通过 hasMany 定义一对多关联。hasMany 原型如下：

```
hasMany('模型类名','外键名','主键名');
```

示例代码如下:

```
namespace app\index\model;

use think\Model;

class User extends Model {
    protected function articles(){
        return $this->hasMany('Article')
    }
}
$user = User::get(1);
print_r($user->articles); // 读取用户所有文章
print_r($user->articles()->where('pubdate',date('Y-m-d'))->selec
t());// 查看当天该用户发布的文章
```

6.18.5 一对多关联模型数据操作

与一对一关联类似,需要先查询或者新建主模型,然后保存从属模型数据,示例代码如下:

```
$user = User::get(1);
$user->articles()->save(['title'=>'demo']); // 单个保存
$user->articles()->saveAll([
    ['title'=>'demo1'],
    ['title'=>'demo2']
]);
```

6.18.6 一对多从属关联

一对多从属关联和一对一从属关联一致，这里不再举例说明，可以参考前面小节的例子。

6.18.7 多对多关联

多对多关联直接用语言表述可能有点难以理解，请看以下例子。

部门和员工的关系是一个部门可以有多个员工，一个员工也可以在多个部门（虽然现实中很少这样）。这时的数据表结构如下所示。

- 部门表（部门 ID，部门名称）
- 员工表（员工 ID，员工姓名）
- 员工所在部门表（员工 ID，部门 ID）

如果没有员工所在的部门表，那么这个多对多关联是无法实现的。假设员工表有个部门 ID，这时只能查到一个员工仅有的一个部门，与一个员工在多个部门的需求不符。

所以多对多的结论就是：两个模型通过中间表才能实现多对多关联。明白了这个，接下来的内容就比较简单了。ThinkPHP 5 的多对多关联也是基于该理论设计的，只不过 ThinkPHP 5 使用的是中间模型，而上文使用的是中间表，原理是一致的。

ThinkPHP 5 使用 belongsToMany 方法来实现多对多定义，belongsToMany 方法原型如下：

```
belongsToMany('关联模型类','中间表|关联模型','外键','关联键');
```

比如上文中部门表（department）和员工（member）的关联代码如下：

```
namespace app\index\model;

use think\Model;

class Department extends Model {
    public function members() {
        return $this->belongsToMany('Member','department_member',
'member_id','department_id');
    }
}
$department = Department::get(1); // 获取一个部门
print_r($department->members); // 读取该部门所有员工
foreach($department->members as $member) {
    print_r($member->pivot); // 获取中间表(department_member)数据
}
```

6.18.8　多对多模型数据操作

- 完全新增关联数据（中间表无数据，被关联表也无数据）

```
$department = Department::get(1);
$department->members()->save(['name'=>'张三']);
$department->members()->saveAll([
    ['name'=>'张三'],
['name'=>'李四']
]);
```

- 被关联表有数据（比如有员工），中间表没数据（员工未关联到部门）

```
$department = Department::get(1);
$department->members()->attach(1); // 将 ID 为 1 的员工关联到 ID 为 1 的部门
$department->members()->detach(1); // 将 ID 为 1 的员工取消部门 ID 为 1 的关联
$department->members()->attach([1,2,3]); // 批量关联到 ID 为 1 的部门
$department->members()->detach([1,2,3]); // 批量解除关联
```

6.18.9 多对多从属关联

在多对多关联关系中，关联表和被关联表地位一致，都是通过中间表关联对方，所以定义也类似。以上文中部门与员工为例，示例代码如下：

```
    namespace app\index\model;

    use think\Model;

    class Member extends Model {
        public function departments() {
        return $this->belongsToMany('Department','department_member',
'department_id','member_id');
    }
    }
```

数据操作代码类似，这里不再赘述。

6.18.10 不定类型关联模型

本小节标题可能一眼看不明白，不过没关系，还是那句话，抛开生硬的理论介绍，直接以举例开始。不用理会本小节标题，明白其中意思即可达到本小节学习的目的。

假设我们在设计一个支付系统，需要考虑的是支付通道往往是确定的，比如微信支付、支付宝支付、银联等，本例只考虑一种，以微信支付为例。

首先，系统有一张订单总表，所有与微信支付有关的订单数据都存放在这张表里，方便和微信支付对账（因为入口、出口统一，这就是支付网关的一个作用）。但是我们系统可能有很多类型的订单，比如购买商城物品、购买会员服务等。总订单表就需要一个类型字段来指明该订单具体是什么类型以及对应类型的表标识键数据。以下是示例的表结构：

- 总订单表（订单 ID、订单类型、订单类型对应的 ID、订单名称、订单金额、下单时间、支付时间、微信支付数据等）
- 商城订单表（商城订单 ID、订单名称、订单金额、总订单表 ID）
- 会员服务表（会员 ID、VIP 登记、生效时间、到期时间、总订单表 ID）

当我们购买了商城订单时，总订单表会写入一条数据，订单类型为商城订单，订单类型对应的 ID 为商城订单 ID。

当我们购买了会员服务时，总订单表会写入一条数据，订单类型为会员服务，订单类型对应 ID 为会员 ID。

看到这里，相信有的读者应该明白了一点东西，那就是总订单每条记录关联的表是不定的，有时候是商城订单表，有时候是会员服务表。

本章节前面讲过的内容中关联表和被关联表都是确定的，每条记录关联的数据类型也是确定的，所以本小节名称才定为了不定类型关联模型。

以往查询这种数据都需要循环查询、效率极低，但是需求是要实现的，有时候只能牺牲性能保证需求，增加缓存不能从根本上解决问题。好在 ThinkPHP 5 已经为我们内置了这一种关联操作。

该关联分为一对一关联和一对多关联，上文中的商城订单为一对一关联（每个商城订单表在总订单表中只有一条记录），而像一般内容发布系统中评论和被评论内容就是一对多关联（一篇内容可以有多条评论）。

假设上文的总订单表为主表，商城订单和会员服务为副表，ThinkPHP 5 中副表使用 morphMany 和 morphOne 方法关联主表，主表使用 morphTo 声明关联键。

morphMany 方法原型如下：

```
morphMany('主模型','类型字段定义'[,'关联结果类型'])
```

- 主模型在本例中为总订单表，也就是 Order。
- 类型字段定义有两种方式：字符串（定义的字符串*_type* 为类型，定义的字符串*_id* 为类型对应的 ID），数组（['类型字段','类型 ID 字段']）。
- 关联结果默认为从表对应模型，也可以使用其他模型类名。

morphOne 和 morphMany 原型类似。

morphTo 方法原型如下：

```
morphTo('类型字段定义'[,'类型与模型映射关系'])
```

- 类型字段定义需要与 morphMany 或 morphOne 中类型字段定义中相对应。
- 类型与模型映射关系在默认情况下，框架会使用副表模型名作为类型识别键，可以通过数组定义来覆盖配置。

以上面的订单系统为例，使用 ThinkPHP 5 的不定关联类型模型来实现：

```
// 定义商城订单对应的总订单
namespace app\index\model;

use think\Model;

class MallOrder extends Model {
    public function order() {
        return $this->morphOne('Order','item');
    }
}
```

根据上文中类型字段定义规则，示例代码使用的是字符串，那么主订单 order 表中 *item_type* 对应类型（商城订单类型），*item_id* 对应类型 ID（商城订单表 ID）。

```
// 定义主表模型类
namespace app\index\model;
```

```
use think\Model;

class Order extends Model {
    public function item() {
        return $this->morphTo('item',[
            'mall' => MallOrder::class,
            'vip' => VipOrder::class,
        ]);
    }
}
$mallOrder = MallOrder::get(1); // 读取商城订单
print_r($mallOrder->order); // 读取商城订单对应的总订单

$order = Order::get(1); // 读取总订单
print_r($order->item); // 读取具体类型订单，$order->item 有可能为商城订单模型，有可能为会员服务订单模型，取决于总订单中该数据的 item_type 字段
```

一对多使用与一对一类似，只不过副表需要使用 morphMany 来代替 morphOne。

6.18.11 关联数据一次查询优化

请看以下示例代码：

```
$users = User::where('user_id',[1,2,3])->select();
foreach($users as $user) {
    print_r($user->wallet);
}
```

答案是 4 次，第 1 次查询用户列表，然后循环 3 次读取用户钱包。

这样的代码效率是最低的，却是很多开发者喜欢用的，因为省事，否则需要提取所有的用户 ID 数组再额外查询一次，虽然减少了查询次数，但是逻辑复杂了一点。

为了解决该场景，ThinkPHP 5 提供了关联数据一次查询优化的功能，将原本需要开发者手动提取 ID 再查询关联表的操作封装起来。开发者只要多调用一个函数即可。

仍然以本节开头的内容为例，使用关联数据一次查询优化后的代码：

```
$users = User::with('wallet')
->where('user_id',[1,2,3])
->select();
foreach($users as $user) {
    print_r($user->wallet);
}
```

上面的代码改动不大，但是对于效率的提升是非常明显的，特别是数据量多的时候，由 N+1 次查询变为了 2 次查询。

with 函数可以通过传入数组的形式同时载入多个关联模型，也可以通过语法来载入嵌套数据。

```
// 提前加载用户资料和钱包数据
User::with(['profile','wallet'])->select([1,2,3]);
// 提前加载钱包数据和钱包对应流水记录
User::with('wallet.water')->select([1,2,3]);
// 提前加载钱包数据以及对应流水记录和钱包对应提现记录
User::with(['wallet'=>['water','withdrawal']])->select([1,2,3]);
```