ThinkPHP 入门系列


# ThinkPHP Framework 1.0
# DataAccess Manual

# ThinkPHP 1.0
# 数据操作指南

编写：ThinkPHP 文档组

最后更新：2008-06-13

# 目录

# 1 版权信息



对 ThinkPHP 有任何疑问或者建议，请进入官方论坛［ http://bbs.thinkphp.cn ］发布相关讨论。

并在此感谢 ThinkPHP 团队的所有成员和所有关注和支持 ThinkPHP 的朋友。

有关 ThinkPHP 项目及本文档的最新资料，请及时访问 ThinkPHP 项目主站 http://thinkphp.cn 。

# 2 数据操作

## 2.1 数据库访问层

业务逻辑一般都是对数据的访问和操作，ThinkPHP 内置实现了一个抽象数据库访问层，实现上参考了部分目前比较先进的抽象数据库访问层框架，并且会保持不断完善。

ThinkPHP 的抽象数据库访问层主要包含了数据库公共类、数据库驱动类和一些辅助类。在实际应用开发过程中，你可能并不需要接触这些类库，只需要通过各自的 Model 对象来访问和操作数据库就可以。

数据库操作类库的主要特点有：

1. 支持事务和回滚（当然首先需要数据库本身支持），默认是自动提交
2. 支持调试模式
3. 完全支持中文 UTF-8 字符集
4. 把查询操作和执行操作分开，便于返回不同的结果
5. 支持配置文件和 DSN 两种数据库配置读取方式
6. 支持对数据集的数组和对象两种方式返回
7. 支持多个数据库连接
8. 支持主从式分布式数据库

数据库类库中最重要就是公共类 Db 的实现，除了引导各种数据库驱动之外，它还提供了数据库抽象访问的一些基本方法，包括 CURD 的 SQL 指令的抽象访问。查询的结果返回数组，并统一通过 Model 数据对象来访问，最终的目的是为了让应用操作数据库更加方便有效。

抽象数据库访问层的关键类库在于数据库的公共访问类 Db 类，这个类是操作数据库的底层接口，换句话说，Db 类不能够独立存在，必须有对应的数据库驱动类库，但是开发人员不能够跨过该类直接进

行数据库操作的访问。目前系统支持的数据库包含有 MySQL、MySQLi、MsSQL、PgSQL、SqLite 和 Oracle，以及 PDO 的支持，还可以通过插件方式增加自己需要的数据库驱动，开发人员也可以按照规范编写自己的数据库驱动（甚至包括众多的嵌入式数据库的驱动）。

## 2.2 表和模型的命名

当我们创建一个 UserModel 类的时候，其实已经遵循了系统的约定。ThinkPHP 要求数据库的表名和模型类的命名遵循一定的规范，首先数据库的表名和字段全部采用小写形式，模型类的命名规则是除去表前缀的数据表名称，并且首字母大写，然后加上模型类的后缀定义，例如：

UserModel 表示 User 数据对象，（假设数据库的前缀定义是 think_）其对应的数据表应该是 think_user

UserTypeModel 对应的数据表是 think_usertype

User_typeModel 对应的数据表是 think_user_type

如果你的规则和系统的约定不符合，那么需要设置 Model 类的 tableName 属性。

在 ThinkPHP 的模型里面，有两个数据表名称的定义：

1、**tableName** 不包含表前后缀的数据表名称，一般情况下默认和模型名称相同，只有当你的表名和当前的模型类的名称不同的时候才需要定义。

2、**trueTableName** 包含前后缀的数据表名称，也就是数据库中的实际表名，该名称无需设置，只有当上面的规则都不适用的情况下才需要设置。

下面举个例子来加深理解：

例如，在数据库里面有一个 think_categories 表，而我们定义的模型类名称是 CategoryModel，按照系统的约定，这个模型的名称是 Category，对应的数据表名称应该是 think_category（全部小写），但是

现在的数据表名称是 think_categories，因此我们就需要设置 tableName 属性来改变默认的规则（假设我们已经在配置文件里面定义了 DB_PREFIX 为 think_ ）。

```
protected $tableName = 'categories';
// 注意这个属性的定义不需要加表的前缀 think_
```

而对于另外一种特殊情况，数据库中有一个表（top_depts）的前缀和其它表前缀不同，不是 think_ 而是 top_，这个时候我们就需要定义 trueTableName 属性了

```
protected $trueTableName = 'top_depts';
// 注意 trueTableName 需要完整的表名定义
```

另外，我们来了解下表后缀的含义。表后缀通常情况下用处不大，因为这个表的设计有关。但是个别情况下也是有用，例如，我们在定义数据表的时候统一采用复数形式定义，下面是我们设计的几个表名 think_users、think_categories、think_blogs ,我们定义的模型类分别是 UserModel 、CategoryModel 、BlogModel，按照上面的方式，我们必须给每个模型类定义 tableName 属性。其实我们可以通过设置表后缀的方式来实现相同的效果，我们可以设置 DB_SUFFIX 配置参数为 s，那么系统在获取真实的表名的时候就会自动加上这个定义的表后缀，我们就不必给每个模型类定义 tableName 属性了，而只是对 categories 这样的复数情况单独定义 trueTableName 属性就可以了。

## 2.3 获取字段

我们在 UserModel 类里面根本没有定义任何 User 表的字段信息 ,但是系统是如何做到属性对应数据表的字段呢？这是因为 ThinkPHP 可以在运行时动态获取数据表的字段信息（确切的说，是在第一次运行的时候，而且只需要一次，以后会永久缓存字段信息，除非删除），包括数据表的主键字段和是否自动增长等等，如果需要显式获取当前数据表的字段信息，可以使用 getDbFields 方法来获取。如果你在开发过程中修改了数据表的字段信息，可能需要清空 Temp 目录下面的缓存文件，让系统重新获取更新

的数据表字段信息。

也可以在模型类里面手动定义数据表字段的名称，可以避免 IO 加载的效率开销，例如：

```
class UserModel extends Model{
  protected $fields = array(
  'id',
  'username',
  'email',
  'age',
  '_pk'=>'id',
  '_autoInc'=>true
  )
}
```

其中_pk 表示主键字段名称 _autoInc 表示主键是否自动增长类型

1.0.3 以上版本增加了数据库字段缓存的开关 可以通过设置 DB_FIELDS_CACHE 参数来关闭字段自动缓存，如果在开发的时候经常变动数据库的结构，而不希望进行数据表的字段缓存，可以在项目配置文件中增加如下配置：

```
'DB_FIELDS_CACHE'=>false
```

ThinkPHP 的默认约定每个数据表的主键名采用统一的 id 作为标识，并且是自动增长类型的。系统会自动识别当前操作的数据表的字段信息和主键名称，所以即使你的主键不是 id，也无需进行额外的设置，系统会自动识别。要在外部获取当前数据对象的主键名称，请使用下面的方法：

```
$Model = getPk();
```

## 2.4 属性访问

因为 Model 对象本身就属于数据对象，所以属性的访问就显得非常直观和简单。

最常用的访问方式是直接通过数据对象访问，例如

$User = D(“User”);

$User->find(1);

// 获取 name 属性的值

```
echo $User->name;

// 设置 name 属性的值

$User->name = 'ThinkPHP';
```

还有一种属性的操作方式是通过返回一个数据对象的方式：

```
$User = D("User");

// 注意这里返回的 user 数据对象可能并不一定是对象，也可以是一个数组

// 系统默认的惯例配置返回的是一个数组参考 DATA_RESULT_TYPE 的配置值

$user = $User->find(1);

// 如果 DATA_RESULT_TYPE 是 1 那么使用下面的方式操作

// 获取 name 属性的值

echo $user->name;

// 设置 name 属性的值

$user->name = 'ThinkPHP';

// 如果 DATA_RESULT_TYPE 是 0 那么使用下面的方式操作

// 获取 name 属性的值

echo $user['name'];

// 设置 name 属性的值

$user['name'] = 'ThinkPHP';
```

两种方式的属性区别是一个是对象的属性，一个是数组的索引名称。

那么如何获取当前数据对象的所有属性呢，有两个方法可以做到：

使用 getDbFields 方法来获取当前数据对象的属性数组

```
$User->getDbFields();
```

```
// 第二种方式可以直接遍历数据字段，该方式仅限于在没有任何数据操作之前
foreach ($User as $field){
    echo($field);
}
```

## 2.5 连接数据库

ThinkPHP 内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的 Db 类进行操作，而无需针对不同的数据库写不同的代码和底层实现，Db 类会自动调用相应的数据库适配器来处理。目前的数据库包括 MySQL、PgSQL、Sqlite 和 PDO 的支持，如果应用需要使用数据库，必须配置数据库连接信息，数据库的配置文件有多种定义方式：

第一种 在项目配置文件里面定义

```
Return array(
'DB_TYPE'=> 'mysql',
'DB_HOST'=> 'localhost',
'DB_NAME'=>'thinkphp',
'DB_USER'=>'root',
'DB_PWD'=>'',
'DB_PORT'=>'3306',
'DB_PREFIX'=>'think_',
// 其他项目配置参数.....
);
```

系统推荐使用该种方式，因为一般一个项目的数据库访问配置是相同的。该方法系统在连接数据库的时候会自动获取，无需手动连接。

可以对每个项目定义不同的数据库连接信息，还可以在调试配置文件里面定义调试数据库的配置信息，如果在项目配置文件和调试模式配置文件里面同时定义了数据库连接信息，那么在调试模式下面后者生效，部署模式下面前者生效。

第二种 使用 DSN 方式在初始化 Db 类的时候传参数

```
$db_dsn = “mysql://username:passwd@localhost:3306/DbName”;
$db = new Db($db_dsn);
```

该方式主要用于在控制器里面自己手动连接数据库的情况，或者用于创建多个数据库连接。

第三种 使用数组传参数

```
$DSN = array(
'dbms'     => 'mysql',
'username' => 'username',
'password' => 'password',
'hostname' => 'localhost',
'hostport' => '3306',
'database' => 'dbname'
);
```

```
$db = new Db($DSN);
```

该方式也是用于手动连接数据库的情况，或者用于创建多个数据库连接。

第四种 在模型类里面定义

```
protected $connection = array(
'dbms'     => 'mysql',
'username' => 'username',
```

```
'password' => 'password',
'hostname' => 'localhost',
'hostport' => '3306',
'database' => 'dbname'
);
```

// 或者使用下面的定义

```
protected $connection = ”mysql://username:passwd@localhost:3306/DbName”;
```

如果在某个模型类里面定义了 connection 属性，则在实例化模型对象的时候，会使用该数据库连接信息进行数据库连接。通常用于某些数据表位于当前数据库连接之外的其它数据库。

ThinkPHP 并不是在一开始就会连接数据库，而是在有数据查询操作的时候才会去连接数据库。额外的情况是，在系统第一次操作模型的时候，框架会自动连接数据库获取相关模型类的数据字段信息，并缓存下来。

## 2.6 使用 PDO

ThinkPHP 支持 PDO 方式，如果要使用 PDO 方式连接数据库，可以参考下面的设置。

我们以项目配置文件定义为例来说明：

```
Return array(
'DB_TYPE'=> 'pdo',
// 注意 DSN 的配置针对不同的数据库有所区别 请参考 PHP 手册 PDO 类库部分
'DB_DSN'=> 'mysql:host=localhost;dbname=think',
'DB_USER'=>'root',
'DB_PWD'=>'',
'DB_PREFIX'=>'think_',
// 其他项目配置参数.....
);
```

使用 PDO 方式的时候，要注意检查是否开启相关的 PDO 模块。

## 2.7 ActiveRecord

ThinkPHP 实现了 ActiveRecords 模式的 ORM 模型 采用了非标准的 ORM 模型 表映射到类 记录( 集 ) 映射到对象 ,字段属性映射到对象的虚拟属性。最大的特点就是使用方便 ,从而达到敏捷开发的目的。

开发过程中 只需要定义好模型类就可以进行方便的数据操作了 例如我们定义了一个 UserModel 类：

```
class UserModel extends Model{
}
```

甚至无需增加任何属性和方法，我们就可以进行下面的操作了。

```
$User = D("User"); // 实例化 User 对象
// 或者 $User = new UserModel();
$User->find(1); // 查找 id 为 1 的记录
$User->name = 'ThinkPHP'; // 把查找到的记录的名称字段修改为 ThinkPHP
$User->save(); // 保存修改的数据
```

比 ActiveRecord 模式更加高级的是，ThinkPHP 可以把记录集映射到对象，例如

```
$User->findAll();
foreach ($User as $user){
        echo $user->name;// 可以结合配置来使用 $user['name']
}
```

ThinkPHP 提供了灵活和方便的数据操作方法，不仅实现了对数据库操作的四个基本操作（CURD）：创建、读取、更新和删除的实现，还内置了很多实用的数据操作方法，提供了 ActiveRecords 模式的最佳体验。

**新建记录**

```
// 实例化一个 User 模型对象
$User = new UserModel();
// 然后给数据对象赋值
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
// 然后就可以保存新建的 User 对象了
$User->add();
// 如果需要锁实例化模型对象的时候传入数据，可以使用
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User = new UserModel($data);
$User->add();
// 或者直接在 add 方法传入要新建的数据
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User = new UserModel();
$User->add($data);
```

如果你的主键是自动增长类型，不需要传入主键的值就可以新建数据，并且如果插入数据成功的话，

Add 方法的返回值就是最新插入的主键值，可以直接获取。

```
$insertId = $User->add($data);
```

一般情况下，应用中的数据对象不太可能通过手动赋值的方式写入，而是有个数据对象的创建过程。

ThinkPHP 提供了一个 create 方法来创建数据对象，然后进行其它的新增或者编辑操作。

```
$User = D("User");
$User->create(); // 创建 User 数据对象，默认通过表单提交的数据进行创建
```

```
$User->add(); // 新增表单提交的数据
```

Create 方法支持从其它方式创建数据对象，例如，从其它的数据对象，或者数组等

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->create($data);
// 从 User 数据对象创建新的 Member 数据对象
$Member = D("Member");
$Member->create($User);
```

支持新增多条记录

```
$User = new UserModel();
$data[0]['name'] = 'ThinkPHP';
$data[0]['email'] = 'ThinkPHP@gmail.com';
$data[1]['name'] = '流年';
$data[1]['email'] = 'liu21st@gmail.com';
$User->addAll($data);
```

在 MySql 数据库下面，会自动使用一条 SQL 语句实现多数据的插入。

**查询记录**

读取数据库的记录我觉得是数据库操作中的最有意思的一件事情了，写过文本数据库的人都知道，保存和删除数据不难（无非是规范和效率问题），难在可以通过各种方式来查找需要的数据。ThinkPHP 通过各种努力，让数据库的查询操作变得轻而易举，也让 ThinkPHP 变得富有内涵。

ThinkPHP 有一个非常明确的约定，就是单个数据查询和多个数据查询的方法是分开的，或者你会认为有时候自己也不知道要查询的数据是单个还是多个，但是有一点是明确的，你需要的是返回一个数据还是希望返回的是一个数据集。因为对两种类型的返回数据的操作方式是截然不同的，无论何种方式的返回，我们都可以直接在模型对象里面操作，当然也一样可以作为数据传递给你需要的变量。

先举个最简单的例子，假如我们要查询主键为 8 的某个用户记录，我们可以使用下面的一些方法：

```
$User->find(8);
```

这个作为查询语言来说是最为直观的，如果查询成功，查询的结果直接保存在当前的数据对象中，在进行下一次查询操作之前，我们都可以提取，例如获取查询的结果数据：

```
$name = $User->name;
$email = $User->email;
```

遍历查询到的数据对象属性

```
foreach ($User as $key=>$val){
        echo($key.':'.$val);
}
```

// 或者进行相关的数据更改和保存操作

也可以用变量保存下来以便随时使用。

```
$user = $User->find(8);
```

对于上面的查询条件，我们还可以使用 getById 来完成相同的查询

```
$User->getById(8);
```

需要注意的是，对于 find 方法来说，即使查询结果有多条记录，也只会返回符合条件的第一条记录，如果要返回符合要求的所有记录，请使用 findAll 方法。

// 查询主键为 1、3、8 的记录集

```
$User->findAll('1,3,8');
// 遍历数据列表
foreach ($User as $vo){
        dump($vo->name);
}
```

更多的查询操作请参考后面章节的内容。

**更新记录**

了解了查询记录后，更新操作就显得非常简单了。

```
$User->find(1); // 查找主键为 1 的数据

$User->name = 'TOPThink'; // 修改数据对象

$User->save(); // 保存当前数据对象

// 还可以使用下面的方式更新

$User->score = '(score+1)'; // 对用户的积分加 1

$User->save();
```

如果不是使用数据对象的方式来保存，可以传入要保存的数据和条件

```
$data['id'] = 1;

$data['name'] = 'TopThink';

$User->save($data);
```

除了 save 方法外，你还可以使用 setField 方法来更新特定字段的值，例如：

```
$User->setField('name','TopThink','id=1');
```

同样可以支持对字段的操作

```
$User->setField('score','(score+1)','id=1');

// 或者改成下面的

$User->setInc('score','id=1');
```

**删除记录**

```
$User->find(2);

$User->delete(); // 删除查找到的记录

$User->delete('5,6'); // 删除主键为 5、6 的数据

$User->deleteAll(); // 删除查询出来的所有数据
```

**注意事项**

在 ThinkPHP 里面 AR 模式和普通模式结合起来使用，根据实际的项目需要。因此有时候，也不要拘泥于 AR 模式的操作~

## 2.8 普通查询

ThinkPHP 大多数情况使用的都是对象查询，因为充分利用了 ORM 查询语言，了解查询条件的定义对使用对象查询非常有帮助，对于复杂的查询，或者从安全方面考虑，通常我们可以使用 HashMap 对象或者索引数组来传递查询条件。查询条件可以用于 find、findAll 等所有有查询条件的方法，下面是几种查询条件的定义：

**普通查询**

```
// 注意在使用 HashMap 之前需要手动导入
$condition = new HashMap();
// 查询 name 为 thinkphp 的记录
$condition->put('name','thinkphp');
// 或者使用下面的方式赋值
$condition->name = 'thinkphp';
// 使用数组作为查询条件
$condition = Array();
$condition['name'] = 'thinkphp';
```

使用 Map 方式查询和使用数组查询的效果是相同的，并且是可以互换的。

## 2.9 条件查询

在查询条件里面，如果仅仅使用

```
$map->put('name','thinkphp');
```

查询条件应该是 name = 'thinkphp'

如果需要进行其它方式的条件判断，可以使用

```
$map->put('name',array('like','thinkphp%'));
```

这样，查询条件就变成 name like 'thinkphp%'

```
$map->put('id',array('gt',100));
```

表示的查询条件就是 id > 100

```
$map->put('id',array('in','1,3,8'));

// 或者使用数组范围

$map->put('id',array('in',array(1,3,8)));
```

上面表示的查询条件都是 id in (1,3,8)

支持的查询表达式有

EQ|NEQ|GT|EGT|LT|ELT|LIKE|BETWEEN|IN|NOT IN

## 2.10 区间查询

ThinkPHP 支持对某个字段的区间查询，例如：

```
$map->put('id',array(1,10)); // id >=1 and id<=10

$map->put('id',array('10','3','or')); //id >= 10 or id <=3

$map->put('id',array(array('neq',6),array('gt',3),'and')); // id != 6 a

nd id > 3
```

## 2.11 组合查询

如果进行多字段查询，那么字段之间的逻辑关系是 逻辑与 AND，但是用下面的规则可以更改默认的逻辑判断，例如下面的查询条件：

```
$map->put('id',array('neq',1));
$map->put('name','ok');
// 现在的条件是 id !=1 and name like '%ok%'
$map->put('_logic','or');
// 现在的条件变为 id !=1 or name like '%ok%'
```

## 2.12 多字段查询

ThinkPHP 还支持直接对进行多字段查询的方法 ,可以简化查询表达式和完成最复杂的查询方法 ,例如：

```
$map->put('id,name,title',array(array('neq',1),array('like','%aaa'),array('like','%bbb%'),'or'));
```

查询条件是

```
( id != 1) OR ( name like '%aaa') OR ( title like '%bbb%')
```

如果结合上面的几种方式，我们可以写出下面更加复杂的查询条件

```
$map->put('id',array('NOT IN','1,6,9'));
$map->put('name,title',array(array('like','%aaa'),array('like','%bbb%'),'or'));
```

以上查询条件变成：

```
( id NOT IN(1,6,9) ) AND ( ( name like '%aaa') OR ( title like '%bbb%') )
```

## 2.13 统计查询

在应用中我们经常会用到一些统计数据，例如当前所有（或者满足某些条件）的用户数、所有用户的最大积分、用户的平均成绩等等，ThinkPHP 为这些统计操作提供了一系列的内置方法。

```
// 获取用户数
$userCount = $User->count();
// 获取用户的最大积分
$maxScore = $User->max('score');
// 获取积分大于 0 的用户的最小积分
$minScore = $User->min('score','score>0');
// 获取用户的平均积分
$avgScore = $User->avg('score');
// 统计用户的总成绩
$sumScore = $User->sum('score');
```

## 2.14 定位查询

ThinkPHP 支持定位查询，可以使用 getN 方法直接返回查询结果中的某个位置的记录。例如：

```
// 返回符合条件的第 2 条记录
$User->getN(2,'score>80','score desc');
还可以获取最后第二条记录
$User->getN(-2,'score>80','score desc');
如果要查询第一条记录，还可以使用
$User->first('score>80','score desc');
// 获取最后一条记录
```

```
$User->last('score>80','score desc');

// 获取积分最高的前 5 条记录

$User->top(5,'','score desc');
```

## 2.15 动态查询

借助 PHP5 语言的特性，ThinkPHP 实现了动态查询。该查询方式针对数据表的字段进行查询。例如，User 对象拥有 id,name,email,address 等属性，那么我们就可以使用下面的查询方法来直接根据某个属性来查询符号条件的记录。

```
$user = $User->getByName('liu21st');
```

上面的查询会转化为$User->getBy('name','liu21st')的查询语言来执行

```
$user = $User->getByEmail('liu21st@gmail.com');

$user = $User->getByAddress('中国深圳');
```

暂时不支持多数据字段的动态查询方法，请使用 find 方法和 findAll 方法进行查询。

ThinkPHP 还提供了另外一种动态查询方式，就是获取符合条件的前 N 条记录

例如，我们需要获取当前用户中积分大于 0，积分最高的前 5 位用户

```
$User->top5('score>0','*','score desc');
```

而在另外一个频道，我们需要获取点击最多的前 10 位主播

```
$Master->top10('','*','visit desc');
```

## 2.16 SQL 查询

ThinkPHP 内置的 ORM 和 ActiveRecord 模式实现了方便的数据存取操作，而且 1.0.3 新版增加的连贯操作功能更是让这个数据操作更加清晰，但是 ThinkPHP 仍然保留了原生的 SQL 查询和执行操作支持，为了满足复杂查询的需要和一些特殊的数据操作，SQL 查询的返回值因为是直接返回的 Db 类的查询结果，没有做任何的处理。而且可以支持查询缓存、延迟加载和事务锁（悲观锁）。主要包括下面两个

方法：

1、query 方法

query 方法是用于 sql 查询操作，和 findall 一样返回数据集，但是不会写入模型类的 dataList 属性，也不会进行数据关联操作，例如：

```
$Model = new Model() // 实例化一个 model 对象 没有对应任何数据表
$Model->query("select * from think_user where status=1");
```

2、execute 方法

用于更新和写入数据的 sql 操作，返回影响的记录数，例如：

```
$Model = new Model() // 实例化一个 model 对象 没有对应任何数据表
$Model->execute("update think_user set name='thinkPHP' where status=1");
```

关于原生 SQL 操作的一点补充

通常使用原生 SQL 需要手动加上当前要查询的表名，如果你的表名以后会变化的话，那么就需要修改每个原生 SQL 查询的 sql 语句了，针对这个情况，TP 还提供了一个小的技巧来帮助解决这个问题。

例如：

```
$model = D("User");
$model->query('select * from __TABLE__ where status>1');
```

我们这里使用了__TABLE__ 这样一个字符串，系统在解析的时候会自动替换成当前模型对应的表名，这样就可以做到即使模型对应的表名有所变化，仍然不用修改原生的 sql 语句。这种方式适合于实际的 Model 类，如果直接实例化 Model 类的话则无效了。

## 2.17 事务支持

ThinkPHP 提供了单数据库的事务支持，如果要在应用逻辑中使用事务，可以参考下面的方法：

```
// 启动事务
```

```
$User->startTrans()
// 启动事务
$User->commit()
// 事务回滚
$User->rollback()
```

## 2.18 延迟查询

延迟加载是由数据库抽象层底层支持的，Db 类内置了 lazyQuery 方法来提供延迟加载支持，原则上，任何查询都可以使用延迟加载。和普通查询返回一个 ArrayObject 对象不同的是，延迟加载返回的是一个 ResultIterator 对象，只有在遍历的时候才真正进行查询操作，例如下面的例子：

```
$User = D("User");
// 第一种 使用 Model 类的 query 方法进行延迟加载，该方法主要针对 SQl 查询方式
// 第二个参数是是否缓存 第三个参数是是否采用延迟加载
$list = $User->query("select * from think_user",false,true);
// 当使用 foreach 进行遍历的时候才真正进行查询
foreach ($list as $data){
   dump($data);
}
// 第二种 使用 startLazy 和 stopLazy 方法进行延迟加载，可以用在 Model 类的所有查询方式
$User->startLazy();
$list = $User->findAll();
$User->stopLazy();
foreach ($list as $data){
dump($data);
}
```

// 第三种 直接在方法后面使用 lazy 参数，进行延迟加载，这种方式略显麻烦

$User->findAll($condition,$fields,$order,$limit,$group,$having,$cache,$relation,$lazy)

第四种 使用连贯操作的 lazy 方法，例如

$User->lazy(true)->findAll();

注意：对单个数据的查询使用延迟加载没有实际意义，如果确实需要可以通过 query 查询方法。

## 2.19 表间关联

ThinkPHP 支持数据表的关联操作，目前支持的关联关系包括下面三种：

1、ONE_TO_ONE ( 包括 HAS_ONE 和 BELONGS_TO )

2、ONE_TO_MANY( 包括 HAS_MANY 和 BELONGS_TO)

3、MANY_TO_MANY

无论何种关联关系，我们统一在_link 属性里面定义，例如，我们在 UserModel 类里面定义关联关系如下：

```
var $_link = array(
    // 每个用户都有一个个人档案
    array(  'mapping_type'=>HAS_ONE,
            'class_name'=>'Profile',
            'foreign_key'=>'userId',
            'mapping_name'=>'profile',
           ),

   // 每个用户有多个文章
    array(   'mapping_type'=>HAS_MANY,
            'class_name'=>'Article',
```

```
        'foreign_key'=>'userId',
        'mapping_name'=>'articles',
        'mapping_order'=>'cTime desc'),
    // 每个用户都属于一个部门
    array(  'mapping_type'=>BELONGS_TO,
        'class_name'=>'Dept',
        'foreign_key'=>'userId',
        'mapping_name'=>'dept'),
    // 每个用户可以属于多个组，每个组可以有多个用户
    array(   'mapping_type'=>MANY_TO_MANY,
        'class_name'=>'Group',
        'mapping_name'=>'groups',
        'foreign_key'=>'userId',
        'relation_foreign_key'=>'goupId',
        'relation_table'=>'think_gourpUser')
    );
```

在实际的开发过程中，关联关系的定义可以简化，只有 mapping_type 和 class_name 是必须的，其它的参数都是可选的，如果没有设置，系统会有默认的约定规则。

默认规则

外键的默认规则是当前数据对象名称_id，例如：

UserModel 对应的可能是表 think_user （注意：think 只是一个表前缀，可以随意配置）

那么 think_user 表的外键默认为 user_id，如果不是，就必须在定义表间映射关系的时候定义 foreign_key 。

同样的道理，对于多对多的情况，relation_foreign_key 的规则相同。

多对多的中间表默认表规则如下：

如果 think_user 和 think_group 存在一个对应的中间表，默认的表名应该是

如果是由 group 来操作关联表，中间表应该是 think_group_user，如果是从 user 表来操作，那么应该是 think_user_group，也就是说，多对多关联的设置，必须有一个 Model 类里面需要显式定义中间表，否则双向操作会出错。

中间表无需另外的 id 主键（但是这并不影响中间表的操作），通常只是由 user_id 和 group_id 构成。

另外一个潜规则，因为表间映射关系可以定义 condition 属性，通常都是通过外键来获取关联数据，但是如果你定义了 condition 属性，那么就会重新使用 condition 的条件来获取关联表的数据。

在 HAS_MANY 和 MANY_TO_MANY 情况下，可以使用 mapping_order 和 mapping_limit 进行记录的排序和获取部分数据。HAS_ONE 和 BELONGS_TO 永远都只会返回最多一条记录。

另外除了 MANY_TO_MANY 之外，其他映射关系可以使用 mapping_fields 来返回你需要的关联表字段。

## 2.20 锁机制

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒种，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓的“锁”，即给我们选定的目标数据上锁，使其无法被其他程序修改。 ThinkPHP 支持两种锁机制：即通常所说的“悲观锁（Pessimistic Locking）”和“乐观锁（Optimistic Locking）”。

悲观锁（Pessimistic Locking）

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。 通常是使用 for update 子句来实现悲观锁机制。

ThinkPHP 支持悲观锁机制，默认情况下，是关闭悲观锁功能的，要在查询和更新的时候启用悲观锁功能，可以通过下面的方式：

```
// 启用悲观锁功能
$User->startLock();
$User->save($data);
// 关闭悲观锁功能
$User->stopLock();
```

乐观锁（ Optimistic Locking ）

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。 如一个金融系统，当某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时( 如更改用户帐户余额 ),如果采用悲观锁机制 ,也就意味着整个操作过程中( 从操作员读出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间 ), 数据库记录始终处于加锁状态，可以想见，如果面对几百上千个并发，这样的情况将导致怎样的后果。乐观锁机制在一定程度上解决了这个问题。乐观锁 ,大多是基于数据版本( Version )记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。

ThinkPHP 也可以支持乐观锁机制，要启用乐观锁，只需要定义模型类的 optimLock 属性，并且在数据表字段里面增加相应的字段就可以自动启用乐观锁机制了。默认的 optimLock 属性是 lock_version，也就是说如果要在 User 表里面启用乐观锁机制，只需要在 User 表里面增加 lock_version 字段，如果有已经存在的其它字段作为乐观锁用途，可以修改模型类的 optimLock 属性即可，例如，下面的定义把 User 表的 version 字段作为乐观锁字段。

```
Class UserModel extends Model{
  protected $optimLock = 'version';
}
```

如果存在 optimLock 属性对应的字段，但是需要临时关闭乐观锁机制，把 optimLock 属性设置为 false 就可以了。

## 2.21 文本字段

ThinkPHP 支持数据模型中的个别字段采用文本方式存储，这些字段就称为文本字段，通常可以用于某些 Text 或者 Blob 字段，或者是经常更新的数据表字段。

要使用文本字段非常简单，只要在模型里面定义 blobFields 属性就行了。例如，我们需要对 Blog 模型的 content 字段使用文本字段，那么就可以使用下面的定义：

Protected $blobFields = array('content');

系统在查询和写入数据库的时候会自动检测文本字段，并且支持多个字段的定义，手动获取文件字段的方式是：

$Model-> getBlobFields($data,$field);

需要注意的是：对于定义的文本字段并不需要数据库有对应的字段，完全是另外的。而且，暂时不支持对文本字段的搜索功能。

## 2.22 连贯操作

ThinkPHP 从 1.0.3 版本开始，增加了模型类的连贯操作支持，可以有效的提高数据存取的代码清晰度和开发效率。使用方面也比较简单，看下面的一个例子，

假如我们现在要查询一个 User 表的满足状态为 1 的前 10 条记录，并希望按照用户的创建时间排序

如果按照旧版的写法，我们的代码如下：

```
$User->findAll('status=1','*','create_time',10);
```

如果使用连贯操作的话，代码改成：

```
$User->where('status=1')->order('create_time')->limit(10)->findAll();
```

虽然看起来代码量增加了一些，但是很明显，代码清晰了很多，也不会混淆。

除了 findAll 方法必须放到最后一个外，其他的连贯操作的方法调用顺序没有先后，例如，下面的代码和上面的等效：

```
$User->order('create_time')->where('status=1')->limit(10)->findAll();
```

是否连贯操作仅仅是为了不用在 findAll 方法里面写过多的参数和记住这些的顺序呢？事实上 并非如此，连贯操作虽然没有改变之前的方法调用，但是引入了一个更大的灵活性，以往的方式如果要在当前的模型里面操作其他的数据表的话，必须使用原生的 SQL 查询方法，因为 findAll 等查询方法是没有表名参数的，而是由系统自动获取当前模型对应的数据表，这样的做法是否合理，暂且不提，但是并不能保证没有这样的需求，我需要偶尔操作下其他的数据表，抑或其他的数据库，而连贯操作引入了一个 table 方法，可以轻松的改变当前操作的数据表名称，同样的例子，我们上面的查询要求查询另外一个数据库中的 User 表，我们把代码修改为：

```
$User->where('status=1')->table('`other_db`.`think_user`')->order('create_time')->limit(10)->findAll();
```

当然，这只是举个例子，你还要保证你有相关的数据库的权限。

原则上说，连贯操作可以包含之前 findall 方法的所有参数，总结如下：

where 查询条件

table 数据表名称

field 要查询的字段名

order 排序

limit 结果限制

group group 支持

having having 支持

join join 支持

cache 是否缓存查询结果

lazy 是否惰性查询

lock 是否锁查询

sql 直接写 sql 语句

连贯操作的最后查询方法除了 findAll 之外，还可以使用 getN find topN 甚至包括 save 方法

例如：

```
$User->where('id=1')->field('id,name,email')->find();
$User->where('status=1')->order('create_time')->top8();
```

## 2.23 聚合模型

聚合模型是一种虚拟模型，用于把数据表的字段抽象话，更加容易理解和操作。

我们可以把数据表中的某些属性进行数据封装 ,这样就把枯燥的数据表字段赋予更好的可读性。例如，

在数据库有一个 User 表，字段定义如下：

```
CREATE TABLE `topthink_user` (
`id` int(11) unsigned NOT NULL auto_increment,
```

```
`name` varchar(30) NOT NULL default '',
`nickname` varchar(50) NOT NULL default '',
`password` varchar(32) NOT NULL default '',
`email` varchar(255) NOT NULL default '',
`url` varchar(255) NOT NULL default '',
`status` tinyint(1) unsigned NOT NULL default '0',
`remark` varchar(255) NOT NULL,
`city` varchar(50) NOT NULL,
`qq` varchar(15) NOT NULL,
PRIMARY KEY (`id`)
)
```

我们把 User 表中的有关用户信息的字段封装成一个 Info 对象（或者数组），例如，我们希望 Info 对象包括 nickname、email、url、city、qq 这些字段属性，这样，看起来的 User 属性变成 id、name、password、Info（聚合对象属性，包括 nickname、email、url、city、qq）、status

所以在进行数据操作的时候，我们只需要把 Info 对象赋值传递到 User 对象就可以完成 Info 对象所封装的属性的数据写入。这个 Info 对象就称为聚合对象，或者组合对象。在 ThinkPHP 里面，这个聚合对象并不一定要定义 Model，因为可以动态的创建这个聚合对象，例如：

```
$User = D("User");
$User->Info=array('nickname'=>'ThinkPHP','email'=>'ThinkPHP@gmail.com';
$User->add();
```

这里就使用了 Info 聚合对象来完成数据写入操作，在写入数据库之前，ThinkPHP 会自动把聚合对象的属性值转换成 User 对象的属性来完成数据写入。当然，要使用聚合对象，我们还要对 UserModel 进行一些定义，确保系统可以识别该属性是聚合对象属性还是普通属性。

我们需要在 UserModel 类里面增加聚合对象的定义

```
$aggregation = array('Info');
```

我们可以同时定义多个聚合对象，例如：

```
$aggregation = array('Info','Log');
```

默认情况下，对聚合对象的属性是不受限制，而是由应用代码把握，为了避免无效的字段写入数据库而导致错误，我们可以限制属性，通过使用下面的方式定义：

```
$aggregation = array(array('Info','nickname,email'));
```

这样一来，聚合对象的属性就只能包括 nickname 和 email 属性，其它属性的值将无效，如：

```
$User = D("User");
$User->Info=array('nickname'=>'ThinkPHP','msn'=>'ThinkPHP@gmail.com');
$User->add();
```

上面的操作，msn 属性就不会写入数据库。

有些情况下，我们可能需要把数据表的字段映射成我们想要的属性名称，例如，我想把 url 字段映射成 web 属性定义，但是不想修改数据表的定义，我们可以进行下面的定义

```
$aggregation = array(array('Info',array('nickname','email','web'=>'url'
)));
```

于是，下面的操作

```
$User->Info=array('nickname'=>'ThinkPHP','web'=>'http://thinkphp.cn');
$User->add();
```

系统会自动把 web 属性转换成 url 字段名称写入数据库。

我们还可以单独定义一个 Info 模型对象，来更好的配合 User 对象完成一些复杂操作

在定义 Info 对象的时候，我们要注意，这个对象是一个虚拟对象，并没有对应数据库的任何数据表，所以有些特殊的属性要定义，例如：

```
Class InfoModel extends Model{
function _initialize(){
// 表示该对象是复合对象
$this->composite = true;
```

```
}
}
```

定义了 composite 为 true 后，系统就不会对 Info 对象进行数据库初始化操作，然后我们就可以进行下面的操作：

```
$User = D("User");
$Info = D("Info");
$Info->nickname = "ThinkPHP";
$Info->email = 'ThinkPHP@gmail.com';
$User->Info=$Info;
$User->add();
```

## 2.24 视图模型

ThinkPHP 在 ORM 模型里面模拟实现了数据库的视图模型，该功能可以用于多表查询。

要定义视图对象，需要设置 viewModel 为 true，然后设置 viewFields 属性即可，例如下面的例子，我们设置了一个 BlogView 模型对象，其中包括了 Blog 模型的 id、name、title 和 User 模型的 name，以及 Category 模型的 title 字段，我们通过创建 BlogView 模型来快速读取一个包含了 User 名称和类别名称的 Blog 记录（集）。

```
class BlogViewModel extends Model
{
        var $viewModel = true;
        var $viewFields = array(
                'Category'=>array('title'=>'categoryName'),
                'User'=>array('name'=>'userName'),
                'Blog'=>array('id','name','title'),
                );
}
```

接下来，我们就可以对视图对象进行操作了

```
$Model = D("BlogView");
$Model->findAll();
```

因为视图查询返回的结果有时候存在重复数据，我们可以通过定义 viewConditon 属性来设置查询的基础条件，例如

```
$viewCondition = array(
'Blog.categoryId'=>array('eqf','Category.id'),
'Blog.userId'=>array('eqf','User.id'),
);
```

注意 eqf 指的是后面的条件不是字符串，而是 SQL 字段操作，这样定义后，在进行其它查询的时候，会自动带上这个基础条件。视图模型的查询操作尽可能使用 Map 对象或者数组方式，否则就需要手动添加基础条件。例如

```
$Model->findAll('Blog.categoryId=Category.id AND Blog.userId=User.id AN
D Blog.id in (1,3,6)');
```

视图模型的定义一般需要先单独定义其中的模型类，但是这并不是必须的，如果没有定义其中的模型类，系统会默认按照系统的规则进行数据表的定位。

目前的视图模型仅仅针对查询操作

## 2.25 单表继承

ThinkPHP 支持单表继承方式，可以用来简化数据库的设计，借助单表继承，我们可以使用一个数据表来定义多个数据模型，区别在于使用一个存在一个附加的字段来记录所属的类型。借助数据对象的自动处理功能，我们可以在创建单表继承的不同的数据对象的时候自动写入所属的类型。例如：

在数据库里面，我们只有一个 User 数据表，该数据表包括了我们需要的其它单表继承类型的所有字段，

定义 UserModel 模型的时候我们和其它的一样，简单定义就行了。

```
UserModel extends Model{
}
```

对于其它的数据类型，我们在定义的时候只是需要继承相关的模型类即可，并且为了自动写入我们需要的类型字段，我们可以定义自动处理机制，例如，我们对 MemberModel 模型定义如下：

```
MemberModel extends UserModel{
   // 我们需要指定所对应的数据表名称
   Protected $tableName = 'user';
protected $_auto = array (
array('type','1','ADD'),   // 写入数据的时候默认把 type 字段设置为 1 这里用来表示 Member 类型
);
// 为了保证查询的时候只是查询 Member 类型的记录，我们定义查询回调方法
Public function _before_read(&$condition){
   // 注意这里是参考写法，具体可能还要对 condition 进行类型判断
   $condition->put('type','1');
}
}
```

同样的方式，我们来定义 Manager 数据模型：

```
ManagerModel extends UserModel{
   // 我们需要指定所对应的数据表名称
Protected $tableName = 'user';
protected $_auto = array (
array('type','2','ADD'),   // 写入数据的时候默认把 type 字段设置为 2 这里用来表示 Manager 类型
);
}
$data['name'] = 'liu21st';
```

$data['email'] = 'liu21st@gmail.com';

$data['score'] = 80; // 对于 Member 类型我们需要记录 score 字段

$Member->create($data);

$Member->add();

$data['name'] = 'thinkphp;

$data['email'] = 'thinkphp@gmail.com';

$data['dept'] = '开发部门'; // 对于 Manager 类型我们需要记录 dept 字段

$Manager->create();

$Manager->add();

$Member->first(); // 查找第一个 Member 数据

## 2.26 字段映射

ThinkPHP 的字段映射功能可以让你在表单中隐藏真正的数据表字段，而不用担心放弃 TP 的自动创建表单对象的功能，假设我们的 User 表里面有 username 和 email 字段，我们需要映射成另外的字段，定义方式如下：

```
Class UserModel extends Model{
  protected $_map = array(
    'username'  =>'name',
    'email'     =>'mail',

  );
}
```

这样，在表单里面就可以直接使用 name 和 mail 名称作为表单数据提交了。在保存的时候会字段转换成定义的字段映射。

## 2.27 数据验证

系统内置了数据对象的自动验证功能，而大多数情况下面，数据对象是由表单提交的$_POST 数据创

建。

需要使用系统的自动验证功能，只需要在 Model 类里面定义$_validate 属性，是由多个验证因子组成的数组，支持的验证因子格式：

```
array(验证字段,验证规则,错误提示,验证条件,附加规则,验证时间)
```

验证字段就是定义需要验证的表单字段，这个字段不一定是数据库字段，也可以是表单的一些辅助字段，例如确认密码和验证码等等。

验证规则 要进行验证的规则，需要结合附加规则

提示信息 用于验证失败后的提示信息定义

验证因子中上面三个参数必须定义，下面为可选参数。

验证条件

EXISTS_TO_VAILIDATE 或者 0 存在字段就验证 （默认）

MUST_TO_VALIDATE 或者 1 必须验证

VALUE_TO_VAILIDATE 或者 2 值不为空的时候验证

附加规则 配合验证规则使用，包括：

function 使用函数验证，前面定义的验证规则是一个函数名

callback 使用方法验证，前面定义的验证规则是一个当前 Model 类的方法

confirm 验证表单中的两个字段是否相同，前面定义的验证规则是一个字段名

equal 验证是否等于某个值，该值由前面的验证规则定义

in 验证是否在某个范围内，前面定义的验证规则必须是一个数组

unique 验证是否唯一，系统会根据字段目前的值查询数据库来判断是否存在相同的值

regex 使用正则进行验证，表示前面定义的验证规则是一个正则表达式（默认）

如果采用正则进行验证 ,会调用系统内置的验证类进行验证操作 ,该验证类位于 ORG.Text.Validation，

通过正则的方式对数据进行验证，并定义了一些常用的验证规则。包括：

require 字段必须

email 邮箱

url URL 地址

currency 货币

number 数字

这些验证规则可以直接使用。

验证时间:

all 全部情况下验证（默认）

add 新增数据时候验证

edit 编辑数据时候验证

示例：

```
var $_validate          =          array(

array('verify','require','验证码必须！'), //所有情况下用正则进行验证

array(name,'','帐号名称已经存在！',0,'unique','add'), // 在新增的时候验证 name
字段是否唯一

array('value',array(1,2,3),'值的范围不正确！',2,'in'), // 当值不为空的时候判断
是否在一个范围内

array('repassword','password','确认密码不正确',0,'confirm'), // 验证确认密
```

```
码是否和密码一致
array('password','checkPwd','密码格式不正确',0,'function'), // 自定义函数验
证密码格式
);
```

当使用系统的 create 方法创建数据对象的时候会自动进行数据验证操作，代码示例：

```
$User = D("User");
$vo = $User->create();
if (!$vo){
// 如果创建失败 表示验证没有通过 输出错误提示信息
$this->error($User->getError());
}
```

## 2.28 数据填充

在 Model 类定义 $_auto 属性，可以完成数据自动处理功能，用来处理默认值和其他系统写入字段。

$_auto 属性是由多个填充因子组成的数组，填充因子定义格式：

```
array(填充字段,填充内容,填充条件,附加规则)
```

填充字段就是需要进行处理的表单字段，这个字段不一定是数据库字段，也可以是表单的一些辅助字段，例如确认密码和验证码等等。

填充条件包括：

ADD 新增数据的时候处理（默认方式）

UPDATE 更新数据的时候处理

ALL 所有情况下都进行处理

附加规则包括：

function 使用函数

callback 回调方法

field 用其它字段填充

string 字符串（默认方式）

示例：

```
var $_auto = array (
array('status','1','ADD'),  // 默认把 status 字段设置为 1
array('password','md5','ADD','function') // 对 password 字段在新增的时候使
md5 函数处理
array('name','getName','ADD','callback') // 对 name 字段在新增的时候回
getName 方法
array('mTime','time','UPDATE','function'), // 对mTime字段在编辑的时候写入当
前时间戳
);
```

PS：该自动填充可能会覆盖表单提交项目。其目的是为了防止表单非法提交字段。

使用 Model 类的 create 方法创建数据对象的时候会自动进行表单数据处理

## 2.29 分布式数据库

ThinkPHP 的模型支持主从式数据库的连接，配置 DB_DEPLOY_TYPE 为 1 可以采用分布式数据库支持。

如果采用分布式数据库，定义数据库配置信息的方式如下：

```
// 在项目配置文件里面定义
Return array(
'DB_TYPE'=> 'mysql', // 分布式数据库的类型必须相同
'DB_HOST'=> '192.168.0.1,192.168.0.2',
'DB_NAME'=>'thinkphp', // 如果相同可以不用定义多个
'DB_USER'=>'user1,user2',
'DB_PWD'=>'pwd1,pwd2',
'DB_PORT'=>'3306',
'DB_PREFIX'=>'think_',
…… 其它项目配置参数
);
```

连接的数据库个数取决于 DB_HOST 定义的数量，所以即使是两个相同的 IP 也需要重复定义，但是其他的参数如果存在相同的可以不用重复定义，例如：

```
'DB_PORT'=>'3306,3306' 和 'DB_PORT'=>'3306' 等效
'DB_USER'=>'user1',
'DB_PWD'=>'pwd1',
和
'DB_USER'=>'user1,user1',
'DB_PWD'=>'pwd1,pwd1',
```

等效。

还可以设置分布式数据库的读写是否分离，默认的情况下读写不分离，也就是每台服务器都可以进行读写操作，对于主从式数据库而言，需要设置读写分离，通过下面的设置就可以：

```
'DB_RW_SEPARATE'=>true,
```

在读写分离的情况下，第一个数据库配置是主服务器的配置信息，负责写入数据，其它的都是从数据库的配置信息，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随

机进行在从服务器中选择。

**注意事项**

主从数据库的数据同步工作不在框架实现，需要数据库考虑自身的同步或者复制机制。

## 2.30 多数据库连接

分布式数据库的配置信息是定义在配置文件里面的，所以一般情况下是无法更改的。另外使用分布式数据库有个不足，就是无法同时连接多个不同类型的数据库。

**多数据库支持**

如果你的应用需要在特殊的时候连接多个数据库 那么可以尝试使用 ThinkPHP 的多数据库连接特性：包括相同类型的数据库和不同类型的数据库。

注意：所谓的相同类型数据库的定义是指和项目配置文件或者模型的数据库连接的数据库类型相同。

我们首先需要在模型类里面增加需要的数据库连接，例如：

我们在 UserModel 类增加多个数据库连接

首先定义额外的数据库连接信息

```
$myConnect1 = array(
'dbms'     => 'mysql',
'username' => 'username',
'password' => 'password',
'hostname' => 'localhost',
'hostport' => '3306',
'database' => 'dbname'
);
```

或者使用下面的定义

```
$myConnect1 = 'mysql://username:passwd@localhost:3306/DbName';
```

定义之后就可以进行动态的增加和切换数据库了。

```
$User->D("User");
// 增加数据库连接 第二个参数表示连接的序号
// 注意内置的数据库连接序号是 0,所以额外的数据库连接序号应该从 1 开始
// 第三个参数表示是否是相同类型的数据库连接，这里我们假设和项目配置采用相同的数据库类型 mysql
$User->addConnect($myConnect1,1,true);
// 如果需要增加另外的 mssql 数据库连接，使用下面的方法
$User->addConnect($myConnect1,1,false);
// 可以同时增加多个数据库连接 myConnect2 和 myConnect3 的定义方式同 myConnect1
$User->addConnect($myConnect1,1,false);
$User->addConnect($myConnect2,2,false);
$User->addConnect($myConnect3,3,true);
```

这样在 UserModel 里面就同时存在了 4 个数据库（加上项目配置里面定义的）连接。那么我们如何使用这些不同的数据库连接呢？ThinkPHP 采用了灵活的切换机制，由应用来控制不同的数据库连接。例如，我们需要在其中一个应用里面用到$myConnect2 这个数据库连接，那么用下面的方法切换即可：

```
$User->switchConnect(2);
```

switchConnect 方法会智能识别该连接是否是相同类型的连接

如果需要删除之前动态添加的连接，可以使用 delConnect 方法，例如：

```
// 删除连接序号为 2 的数据库连接
$User->delConnect(2);
```

## 2.31 自动时间戳

因为 ThinkPHP 的自动完成和自动验证功能知道的比较多，所以很多人第一想法就是使用 TP 的自动完成来进行时间戳的自动写入或者更新，其实关于时间戳的问题使用自动完成功能有点大材小用了，呵呵~其实自动写入时间戳完全可以不用通过自动完成来实现，TP 内置了对时间戳的自动写入和更新支持。

你只需要在模型里面定义需要自动写入和更新的时间戳字段名称就行了，使用数组方式定义：

```
protected $autoCreateTimestamps = array('create_time');
protected $autoUpdateTimestamps = array('update_time');
```

autoCreateTimestamps 属性是值新增数据的时候会自动写入的时间字段名称

autoUpdateTimestamps 属性是值更新数据的时候会自动更新的时间字段名称

都是采用数组的方式，系统默认定义了一些日期字段如下：

```
protected $autoCreateTimestamps = array('create_at','create_on','cTime');
protected $autoUpdateTimestamps = array('update_at','update_on','mTime');
```

定义完成之后，就可以使用模型的 create 方法的时候来自动完成时间戳的写入。

例如：

```
class TopicModel extends Model{
protected $autoCreateTimestamps = array('create_time');
protected $autoUpdateTimestamps = array('update_time');
}
```

然后我们在 Action 类里面就可以通过 create 方法来完成时间戳的自动写入了。

```
class TopicAction extends Action{
//  定义 Topic 数据写入操作
public function insert(){
     $Topic = D("Topic");
```

```
        if($Topic->create()){
            dump($Topic->create_time);
            // 这里可以输出当前的时间戳：1205292755
            $Topic->add();
        }
}
//  定义 Topic 数据更新操作
public function update(){
        $Topic = D("Topic");
        if($Topic->create()){
            dump($Topic->update_time);
            // 这里可以输出当前的时间戳：1205292755
            $Topic->save();
        }
}
}
```

默认的格式是时间戳格式 如果需要写入其他格式的日期和时间 还可以定义$autoTimeFormat 属性，

例如：

```
protected $autoTimeFormat = 'Y-m-d H:i:s';
```

这样就会转换成相应的日期格式写入，格式化方法同 date 方法的 format 参数。

如果我们上面的 TopicModel 里面增加了下面的定义

```
protected $autoTimeFormat = 'Y年m月d日 H:i:s';
```

那么 Action 里面的

```
dump($Topic->create_time);
// 这里可以输出当前的日期：2008 年 4 月 28 日 10:02:12
```

## 2.32 回调方法

模型类支持回调方法，包括：

_initialize 初始化回调方法

_before_validation 前置验证回调

_after_validation 后置验证回调

_before_operation 前置数据处理回调

_after_operation 后置数据处理回调

除了上面的回调方法，Model 还包含了：

_create 写入数据

_update 更新数据

_read 查询数据

_delete 删除数据

_query SQL 查询

ThinkPHP 的 CURD 操作最终大多都调用这 5 个方法，系统支持对这五个方法使用回调方法，每个方法都有前置和后置调用，分别定义相关的 _before 和 _after 方法就可以了，例如可以对_update 方法使用 _before_update 和 _after_update 回调方法。

如果前置方法返回 false，那么将会中止后面的操作。

## 2.33 调试技巧

为了方便使用 ThinkPHP 在开发过程中快速解决遇到的问题，框架具有很多特性可以方便进行调试处理。

**页面 Trace**

页面 Trace 功能提供了很好的 SQL 调试方法，而且可以很直观的在当前页面查看到 SQL 语句信息，我们只需要设置参数：

```
'SHOW_PAGE_TRACE' => TRUE,
```

这样，我们可以在页面看到类似下面的信息：

SQL 记录 ：1 条 SQL

[ 08-05-07 19:20:23 ] RunTime:0.000241s SQL = SELECT * FROM think_blog

**开启 SQL 日志**

页面 Trace 信息虽然直观，但是只能显示当前页面的 SQL 执行信息，如果你使用了 Ajax 操作，那么后台的执行 SQL 可能就不会显示，我们只能通过查看 SQL 日志了。可以在配置参数里面开启 SQL 日志记录：

```
'SQL_DEBUG_LOG' => TRUE,
```

我们可以在 Logs 目录下面的 sql 日志文件里面看到：

[ 08-05-05 12:44:45 ]
RunTime:0.000285s SQL = SELECT * FROM think_form WHERE id in (1,2,4)

这样的记录信息，可以分析具体执行的 SQL 语句是否有问题，以及执行时间，便于优化。

使用 **getLastSql** 方法

如果你没有开启 SQL 日志，也没有使用页面 Trace 信息显示功能，那么依然可以调试可能出现错误的 SQL 语句，我们可以在查询方法的后面使用 getLastSql 方法来查看最后一次执行的 SQL 语句。便于分析错误可能的原因。

```
$User->find(8);
echo $User->getLastSql()
```

**使用调试配置文件**

单独定义调试配置文件，可以给你的项目设置最合理的参数。而且你还可以给调试模式增加单独的数据库连接。

我们可以在项目的 Conf 目录下面增加调试配置文件 config.php，内容如下：

```
<?php
return array(
// 定义数据库连接信息
'DB_TYPE'=> 'mysql',
'DB_HOST'=> 'localhost',
'DB_NAME'=>'test_db',
'DB_USER'=>'root',
'DB_PWD'=>'',
'DB_PORT'=>'3306',
'DB_PREFIX'=>'think_',
'SQL_DEBUG_LOG' => TRUE, // 开启 SQl 日志记录
'SHOW_RUN_TIME' => TRUE, // 开启运行时间显示
'SHOW_ADV_TIME' => TRUE, // 显示详细运行时间信息
'SHOW_DB_TIMES' => TRUE, // 显示数据库查询和写入次数
'SHOW_RUN_TIME' => TRUE, // 开启运行时间显示
);
?>
```

## 2.34 查询技巧

**字段别名**

可以在查询的字段里面使用别名，例如

以 MySQL 为例，我们可以使用

```
$User->where('status=1')
     ->field('id,email,username as `name`')
     ->limit(10)
     ->findAll();
```

可以生成下面的 SQL 语句

Select id,email,username as `name` from think_user limit 10 where 1

注意在 mysql 下面一定在自己添加`符号，否则会自动添加`符号导致不正常。

**使用 JOIN**

假如有二个表，第一个表为 Comment 表，第二个表为 Reply 表，现在做一个左连接，MYSQL 上测试

语句为：

select a.*,b.* from comment a left join reply b on a.id=b.cid

那么可以使用下面的方式来写查询

```
$Blog->table('comment a')
         ->join('reply b on a.id=b.cid')
         ->field('a.*,b.*')
         ->order('id desc')
         ->limit('8')
         ->findall();
```

还可以支持 right join 等方式，定义方式是通过数组参数传入 join 方式

```
$Blog->table('comment a')
     ->join(array('right','reply b on a.id=b.cid'))
     ->field('a.*,b.*')
     ->order('id desc')
     ->limit('8')
```

```
    ->findall();
```