第三章 数据库详解

3.1 数据库配置与连接

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。

3.1.1 配置文件

在全局或者应用配置目录（不清楚配置目录位置的话参考配置章节）下面的`database.php`中（后面统称为数据库配置文件）配置下面的数据库参数：

```
return [
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 数据库类型
            'type'        => 'mysql',
            // 服务器地址
            'hostname'    => '127.0.0.1',
            // 数据库名
            'database'    => 'thinkphp',
            // 数据库用户名
            'username'    => 'root',
            // 数据库密码
            'password'    => '',
            // 数据库连接端口
            'hostport'    => '',
            // 数据库连接参数
            'params'      => [],
            // 数据库编码默认采用utf8
            'charset'     => 'utf8',
            // 数据库表前缀
            'prefix'      => 'think_',
        ],
    ],
];
// default配置用于设置默认使用的数据库连接配置。
// connections配置具体的数据库连接信息，default配置参数定义的连接配置必须要存在。

// type参数用于指定数据库类型
type                数据库
mysql               MySQL
sqlite              SqLite
pgsql               PostgreSQL
sqlsrv              SqlServer
mongo               MongoDb
oracle              Oracle
```

3.1.2 切换连接

我们可以在数据库配置文件中定义多个连接信息

```
return [
```

```
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 数据库类型
            'type'        => 'mysql',
            // 服务器地址
            'hostname'    => '127.0.0.1',
            // 数据库名
            'database'    => 'thinkphp',
            // 数据库用户名
            'username'    => 'root',
            // 数据库密码
            'password'    => '',
            // 数据库连接端口
            'hostport'    => '',
            // 数据库连接参数
            'params'      => [],
            // 数据库编码默认采用utf8
            'charset'     => 'utf8',
            // 数据库表前缀
            'prefix'      => 'think_',
        ],
        'demo'    =>    [
            // 数据库类型
            'type'        => 'mysql',
            // 服务器地址
            'hostname'    => '127.0.0.1',
            // 数据库名
            'database'    => 'demo',
            // 数据库用户名
            'username'    => 'root',
            // 数据库密码
            'password'    => '',
            // 数据库连接端口
            'hostport'    => '',
            // 数据库连接参数
            'params'      => [],
            // 数据库编码默认采用utf8
            'charset'     => 'utf8',
            // 数据库表前缀
            'prefix'      => 'think_',
        ],
    ],
];

// 我们可以调用Db::connect方法动态配置数据库连接信息，例如：
\think\facade\Db::connect('demo')
    ->table('user')
    ->find();
// 注意： connect方法必须在查询的最开始调用，而且必须紧跟着调用查询方法，否则可能会导致部分查询失效或者依然使用默认的数据库连接。
```

模型定义

如果某个模型类里面定义了`connection`属性的话，则该模型操作的时候会自动按照给定的数据库配置进行连接，而不是配置文件中设置的默认连接信息，例如：

```php
<?php
namespace app\index\model;

use think\Model;

class User extends Model
{
    protected $connection = 'demo';
}
```

3.2 数据库基本操作

3.2.1 查询数据

查询单个数据

```php
// 查询单个数据使用find方法：
Db::name('user')->where('id', 1)->find();

// 如果你设置的表前缀是 think_ 最终生成的SQL语句是：
SELECT * FROM `think_user` WHERE  `id` = 1 LIMIT 1
// 注意： find方法查询结果不存在，返回 null，否则返回结果数组
```

查询数据集

```php
// 查询多个数据（数据集）使用select方法：
Db::name('think_user')->where('status', 1)->select();

// select 方法查询结果是一个数据集对象，如果需要转换为数组可以使用
Db::name('think_user')->where('status', 1)->select()->toArray();
```

值和列的查询

```php
// 查询某个字段的值可以用
Db::name('think_user')->where('id', 1)->value('name'); // value 方法查询结果不存在，返回 null

// 查询某一列的值可以用

Db::table('think_user')->where('status',1)->column('name'); // 返回数组
Db::table('think_user')->where('status',1)->column('name', 'id'); // 指定id字段的值作为索引

// 如果要返回完整数据，并且添加一个索引值的话，可以使用
指定id字段的值作为索引 返回所有数据
Db::table('think_user')->where('status',1)->column('*','id');
```

游标查询

如果你需要处理大量的数据，可以使用新版提供的游标查询功能，该查询方式利用了PHP的生成器特性，可以大幅减少大量数据查询的内存开销问题。

```php
$cursor = Db::table('user')->where('status', 1)->cursor();
foreach($cursor as $user){
    echo $user['name'];
}
```

3.2.2 添加数据

```php
// 使用 insert 方法向数据库提交数据
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->insert($data); // insert 方法添加数据成功返回添加成功的条数，通常情况返回 1
// 如果你的数据表里面没有foo或者bar字段，那么就会抛出异常。
```

添加多条数据

```php
// 添加多条数据直接向 Db 类的 insertAll 方法传入需要添加的数据（通常是二维数组）即可。
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
];
Db::name('user')->insertAll($data); // insertAll方法添加数据成功返回添加成功的条数
```

3.2.3 更新数据

使用`update`方法

```php
Db::name('user')
    ->where('id', 1)
    ->update(['name' => 'thinkphp']); // update方法返回影响数据的条数，没修改任何数据返回 0
// 支持使用data方法传入要更新的数据
Db::name('user')
    ->where('id', 1)
    ->data(['name' => 'thinkphp'])
    ->update();
```

自增/自减

```php
// 可以使用inc/dec方法自增或自减一个字段的值（ 如不加第二个参数，默认步长为1）。
// score 字段加 1
Db::name('think_user')
    ->where('id', 1)
    ->inc('score')
    ->update();

// score 字段加 5
Db::name('think_user')
    ->where('id', 1)
    ->inc('score', 5)
    ->update();

// score 字段减 1
Db::name('think_user')
```

```
    ->where('id', 1)
    ->dec('score')
    ->update();

// score 字段减 5
Db::name('think_user')
    ->where('id', 1)
    ->dec('score', 5)
    ->update();
```

3.2.3 删除数据

```
// 根据主键删除
Db::name('think_user')->delete(1);
Db::name('think_user')->delete([1,2,3]);

// 条件删除
Db::name('think_user')->where('id',1)->delete();
Db::name('think_user')->where('id','<',10)->delete();

// 如果不带任何条件调用delete方法会提示错误，如果你确实需要删除所有数据，可以使用
Db::name('user')->delete(true);
```

一般情况下，业务数据不建议真实删除数据，系统提供了软删除机制（模型中使用软删除更为方便）。

```
// 软删除数据 使用delete_time字段标记删除
Db::name('user')
    ->where('id', 1)
    ->useSoftDelete('delete_time',time())
    ->delete();
// useSoftDelete方法表示使用软删除，并且指定软删除字段为delete_time，写入数据为当前的时间戳。
```

3.2.4 查询表达式

查询表达式支持大部分的SQL查询语法，也是`ThinkPHP`查询语言的精髓，查询表达式的使用格式：

```
where('字段名','查询表达式','查询条件');
```

除了where方法外，还可以支持whereOr，用法是一样的。为了更加方便查询，大多数的查询表达式都提供了快捷查询方法。
表达式不分大小写，支持的查询表达式有下面几种：

表达式	含义	快捷查询方法
=	等于	
<>	不等于	
>	大于	
>=	大于等于	
<	小于	
<=	小于等于	
[NOT] LIKE	模糊查询	whereLike/whereNotLike
[NOT] BETWEEN	（不在）区间查询	whereBetween/whereNotBetween

```
[NOT] IN                          （不在）IN 查询
 whereIn/whereNotIn
[NOT] NULL                         查询字段是否（不）是NULL
whereNull/whereNotNull
[NOT] EXISTS                      EXISTS查询
whereExists/whereNotExists
[NOT] REGEXP                       正则（不）匹配查询（仅支持Mysql）
[NOT] BETWEEM TIME                 时间区间比较
 whereBetweenTime
> TIME                            大于某个时间                          whereTime
< TIME                            小于某个时间                          whereTime
>= TIME                            大于等于某个时间                     whereTime
<= TIME                           小于等于某个时间                      whereTime
EXP                               表达式查询，支持SQL语法               whereExp
find in set                       FIND_IN_SET查询                       whereFindInSet
```

以上是详细的查询表达式，下面为大家演示几个常用的查询表达式。

小于（<）

```
Db::name('user')->where('id', '<', 100)->select();

// 最终生成的sql语句是
SELECT * FROM `think_user` WHERE  `id` < 100
```

[NOT] LIKE： 同sql的LIKE

```
Db::name('user')->where('name', 'like', 'thinkphp%')->select();

// 最终生成的SQL语句是：
SELECT * FROM `think_user` WHERE  `name` LIKE 'thinkphp%'
```

[NOT] BETWEEN ： 同sql的[not] between

```
Db::name('user')->where('id','between',[1,8])->select();

// 最终生成的SQL语句都是：
SELECT * FROM `think_user` WHERE  `id` BETWEEN 1 AND 8
```

3.3 链式操作详解

数据库提供的链式操作方法，可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的CURD操作（原生查询不支持链式操作）。
使用也比较简单，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，代码如下：

```
Db::table('think_user')
    ->where('status',1)
    ->order('create_time')
    ->limit(10)
    ->select();
```

这里的`where`、`order`和`limit`方法就被称之为链式操作方法，除了`select`方法必须放到最后一个外（因为`select`方法并不是链式操作方法），链式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：

```
Db::table('think_user')
    ->order('create_time')
    ->limit(10)
    ->where('status',1)
    ->select();
```

系统支持的链式操作方法包含：

连贯操作	作用	支持的参数类型
where*	用于AND查询	字符串、数组和对象
whereOr*	用于OR查询	字符串、数组和对象
whereTime*	用于时间日期的快捷查询	字符串
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
field*	用于定义要查询的字段（支持字段排除）	字符串和数组
order*	用于对结果排序	字符串和数组
limit	用于限制查询结果数量	字符串和数字
page	用于查询分页（内部会转换成limit）	字符串和数字
group	用于对查询的group支持	字符串
having	用于对查询的having支持	字符串
join*	用于对查询的join支持	字符串和数组
union*	用于对查询的union支持	字符串、数组和对象
view*	用于视图查询	字符串、数组
distinct	用于查询的distinct支持	布尔值
lock	用于数据库的锁机制	布尔值
cache	用于查询缓存	支持多个参数
with*	用于关联预载入	字符串、数组
bind*	用于数据绑定操作	数组或多个参数
comment	用于SQL注释	字符串
force	用于数据集的强制索引	字符串
master	用于设置主服务器读取数据	布尔值
strict	用于设置是否严格检测字段名是否存在	布尔值
sequence	用于设置Pgsql的自增序列名	字符串
failException	用于设置没有查询到数据是否抛出异常	布尔值
partition	用于设置分区信息	数组 字符串
replace	用于设置使用REPLACE方式写入	布尔值
extra	用于设置额外查询规则	字符串

duplicate	用于设置DUPLCATE信息	数组 字符串

以上是详细的链式操作方法，下面为大家介绍常用的链式操作方法。

where方法：

`where`方法的用法是ThinkPHP查询语言的精髓，也是ThinkPHP `ORM`的重要组成部分和亮点所在，可以完成包括普通查询、表达式查询、快捷查询、区间查询、组合查询在内的查询操作。`where`方法的参数支持的变量类型包括字符串、数组和闭包。

```
// 查询表达式的使用方式
Db::table('think_user')
    ->where('id','>',1)
    ->where('name','thinkphp')
    ->select();

// 传入数组作为查询条件
Db::table('think_user')->where([
    ['name','=','thinkphp'],
    ['status','=',1]
])->select();
```

alias 方法

用于设置当前数据表的别名，便于使用其他的连贯操作例如join方法等。

```
// 为当前数据表设置别名
Db::table('think_user')
->alias('a')
->join('think_dept b ','b.user_id= a.id')
->select();

// 可以传入数组批量设置数据表以及别名
Db::table('think_user')
->alias(['think_user'=>'user','think_dept'=>'dept'])
->join('think_dept','dept.user_id= user.id')
->select();
```

field 方法

方法主要作用是标识要返回或者操作的字段，可以用于查询和写入操作。

```
Db::table('user')->field('id,nickname as name')->select(); // 可以为字段设置别名
```

order 方法

方法用于对操作的结果排序或者优先级限制。

```
// 为一个字段排序
Db::table('user')
->where('status', 1)
->order('id', 'desc')
->limit(5)
->select();

// 支持使用数组对多个字段的排序
Db::table('user')
->where('status', 1)
->order(['order','id'=>'desc'])
->limit(5)
->select();
```

3.4 高级查询

3.4.1 聚合查询

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，包括：

```
方法                            说明
count                           统计数量，参数是要统计的字段名（可选）
max                             获取最大值，参数是要统计的字段名（必须）
min                             获取最小值，参数是要统计的字段名（必须）
avg                             获取平均值，参数是要统计的字段名（必须）
sum                             获取总分，参数是要统计的字段名（必须）

// 统计用户数
Db::table('think_user')->count();

// 统计最大积分
Db::table('think_user')->max('score');

// 获取用户的平均积分
Db::table('think_user')->avg('score');
```

3.4.2 分页查询

ThinkPHP内置了分页实现，要给数据添加分页输出功能变得非常简单，可以直接在Db类查询的时候调用paginate方法：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = Db::name('user')->where('status',1)->order('id', 'desc')->paginate(10);

// 渲染模板输出
return view('index', ['list' => $list]);

模板文件中分页输出代码如下：

<div>
<ul>
{volist name='list' id='user'}
    <li> {$user.nickname}</li>
{/volist}
```

```
</ul>
</div>
{$list|raw}
```

3.4.3 时间查询

框架内置了常用的时间查询方法，并且可以自动识别时间字段的类型，所以无论采用什么类型的时间字段，都可以统一使用本章的时间查询用法。

```
// whereTime方法提供了日期和时间字段的快捷查询，示例如下：
// 大于某个时间
Db::name('user')
    ->whereTime('birthday', '>=', '1970-10-1')
    ->select();
// 小于某个时间
Db::name('user')
    ->whereTime('birthday', '<', '2000-10-1')
    ->select();
// 时间区间查询
Db::name('user')
    ->whereTime('birthday', 'between', ['1970-10-1', '2000-10-1'])
    ->select();
// 不在某个时间区间
Db::name('user')
    ->whereTime('birthday', 'not between', ['1970-10-1', '2000-10-1'])
    ->select();

// 查询2018年注册的用户
Db::name('user')
    ->whereYear('create_time', '2018')
    ->select();

// 查询2018年6月注册的用户
Db::name('user')
    ->whereMonth('create_time', '2018-06')
    ->select();

// 查询指定某天开始的一周数据 查询2019-1-1到2019-1-7的注册用户
Db::name('user')
    ->whereWeek('create_time', '2019-1-1')
    ->select();

// 查询2018年6月1日注册的用户
Db::name('user')
    ->whereDay('create_time', '2018-06-01')
    ->select();
```

3.4.4 JSON 字段

如果你的`user`表有一个`info`字段是`JSON`类型的（或者说你存储的是JSON格式，但并非是要JSON字段类型），你可以使用下面的方式操作数据。

```
// JSON 数据写入
$user['name'] = 'thinkphp';
$user['info'] = [
    'email'    => 'thinkphp@qq.com',
```

```
    'nickname' => '流年',
];
Db::name('user')
    ->json(['info'])
    ->insert($user);

// JSON数据查询
$user = Db::name('user')
    ->json(['info'])
    ->where('info->nickname','ThinkPHP')
    ->find();
dump($user);

// JSON数据更新
$data['info->nickname'] = 'ThinkPHP';
Db::name('user')
    ->json(['info'])
    ->where('id',1)
    ->update($data);
```

3.4.5 原生查询

`Db`类支持原生`SQL`查询操作，主要包括下面两个方法：

```
// query方法  query方法用于执行SQL查询操作，和select方法一样返回查询结果数据集（数组）。
Db::query("select * from think_user where status=1");

// execute方法 execute用于更新和写入数据的sql操作，如果数据非法或者查询错误则返回false，否则返回影响的记录数。
Db::execute("update think_user set name='thinkphp' where status=1");
```

3.5 查询事件和获取器

3.5.1 查询事件

数据库操作的回调也称为查询事件，是针对数据库的CURD操作而设计的回调方法，主要包括：

```
事件                            描述
before_select                    select查询前回调
before_find                      find查询前回调
after_insert                     insert操作成功后回调
after_update                     update操作成功后回调
after_delete                     delete操作成功后回调


// 使用下面的方法注册数据库查询事件
\think\facade\Db::event('before_select', function ($query) {
    // 事件处理
    return $result;
});
```

3.5.2 获取器

获取可以对从数据库获取的数据进行修改

```
// 获取器方法支持传入两个参数，第一个参数是当前字段的值，第二个参数是所有的数据。
Db::name('user')->withAttr('name', function($value, $data) {
    return strtolower($value);
})->select(); // 查询的数据集数据中的name字段的值会统一进行小写转换。

注意：withAttr方法可以多次调用，对多个字段定义获取器。
```

3.6 事务操作和监听SQL

3.6.1 事务操作

使用事务处理的话，需要数据库引擎支持事务处理。比如 `MySQL` 的 `MyISAM` 不支持事务处理，需要使用 `InnoDB` 引擎。

```
// 最简单的方式是使用 transaction 方法操作数据库事务，当闭包中的代码发生异常会自动回滚，例如：
Db::transaction(function () {
    Db::table('think_user')->find(1);
    Db::table('think_user')->delete(1);
});
```

3.6.2 监听SQL

如果开启数据库SQL监听的话，你可以对数据库执行的任何SQL操作进行监听，使用如下方法：

```
Db::listen(function ($sql, $time, $master) {
    // 记录SQL
    echo $sql . ' [' . $time . 's] ' . ($master ? 'master' : 'slave');
});
```

3.7 数据集

数据库的查询结果默认返回数据集对象。

```
// 获取数据集
$users = Db::name('user')->select();
// 遍历数据集
foreach($users as $user){
    echo $user['name'];
    echo $user['id'];
}

注意：返回的数据集对象是think\Collection，提供了和数组无差别用法，并且另外封装了一些额外的方法。
在模型中进行数据集查询，全部返回数据集对象，但使用的是think\model\Collection类（继承think\Collection），但用法是一致的。
```

可以直接使用数组的方式操作数据集对象，例如：

```php
// 获取数据集
$users = Db::name('user')->select();
// 直接操作第一个元素
$item  = $users[0];
// 获取数据集记录数
$count = count($users);
// 遍历数据集
foreach($users as $user){
    echo $user['name'];
    echo $user['id'];
}
```

需要注意的是，如果要判断数据集是否为空，不能直接使用`empty`判断，而必须使用数据集对象的`isEmpty`方法判断，例如：

```php
$users = Db::name('user')->select();
if($users->isEmpty()){
    echo '数据集为空';
}
```

`Collection`类包含了下列主要方法：

方法	描述
isEmpty	是否为空
toArray	转换为数组
all	所有数据
merge	合并其它数据
diff	比较数组，返回差集
flip	交换数据中的键和值
intersect	比较数组，返回交集
keys	返回数据中的所有键名
pop	删除数据中的最后一个元素
shift	删除数据中的第一个元素
unshift	在数据开头插入一个元素
push	在结尾插入一个元素
reduce	通过使用用户自定义函数，以字符串返回数组
reverse	数据倒序重排
chunk	数据分隔为多个数据块

各位同学，以上就是数据库的全部内容。