ThinkPHP®

3.2.3

快速入门

快速入门 1：基础

快速入门（一）：基础

简介

ThinkPHP是一个快速、简单的基于MVC和面向对象的轻量级PHP开发框架，遵循Apache2开源协议发布，从诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简的代码的同时，尤其注重开发体验和易用性，并且拥有众多的原创功能和特性，为WEB应用开发提供了强有力的支持。

> 本快速入门教程针对3.2.3最新版本制作，虽然大部分特性3.2版本同样存在，但是强烈建议你使用3.2.3版本来学习本入门教程。

下载

ThinkPHP最新版本可以在官方网站（http://thinkphp.cn/down/framework.html）下载。

最新的3.2.3版本下载地址：http://www.thinkphp.cn/down/610.html。

如果你希望保持最新的更新，可以通过github获取当前最新的版本（完整版）。
Git获取地址列表（你可以选择一个最快的地址）：

- Github： https://github.com/liu21st/thinkphp
- Oschina： http://git.oschina.net/liu21st/thinkphp.git
- Code： https://code.csdn.net/topthink2011/ThinkPHP
- Coding： https://git.coding.net/liu21st/thinkphp.git

目录结构

把下载后的压缩文件解压到你的WEB目录（或者任何子目录都可以），框架的目录结构为：

```
www  WEB部署目录（或者子目录）
├─index.php       入口文件
├─README.md       README文件
├─composer.json   Composer定义文件
├─Application     应用目录
├─Public          资源文件目录
└─ThinkPHP        框架目录
```

3.2版本开始相比之前的版本自带了一个完整的应用目录结构（`Application`）和默认的应用入口文件（`index.php`），开发人员可以在这个基础之上灵活调整（目录名称和位置）。其中，`Application`和`Public`目录下面都是空的，而`README.md`和`composer.josn`文件仅用于说明，实际部署的时候可以

删除。

其中，ThinkPHP为框架核心目录，其目录结构如下：

```
├─ThinkPHP 框架系统目录（可以部署在非web目录下面）
│  ├─Common       核心公共函数目录
│  ├─Conf         核心配置目录
│  ├─Lang         核心语言包目录
│  ├─Library      框架类库目录
│  │  ├─Think     核心Think类库包目录
│  │  ├─Behavior  行为类库目录
│  │  ├─Org       Org类库包目录
│  │  ├─Vendor    第三方类库目录
│  │  ├─ ...      更多类库目录
│  ├─Mode         框架应用模式目录
│  ├─Tpl          系统模板目录
│  ├─LICENSE.txt  框架授权协议文件
│  ├─logo.png     框架LOGO文件
│  ├─README.txt   框架README文件
│  └─index.php    框架入口文件
```

> 框架核心目录的结构无需改变，但框架的目录名称（ThinkPHP）可以在应用入口文件中随意更改。

入口文件

在开始之前，你需要一个Web服务器和PHP运行环境，如果你暂时还没有，我们推荐使用集成开发环境WAMPServer（是一个集成了Apache、PHP和MySQL的开发套件，而且支持多个PHP版本、MySQL版本和Apache版本的切换）来使用ThinkPHP进行本地开发和测试。

3.2版本开始框架自带了一个应用入口文件，默认内容如下：

```
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

这段代码的作用就是定义应用目录和加载ThinkPHP框架的入口文件，这是所有基于ThinkPHP开发应用的第一步。

然后，在浏览器中访问运行后我们会看到欢迎页面：

欢迎使用 ThinkPHP！

版本 V3.2.3

当你看到这个欢迎页面的时候，系统已经在 Application 目录下面自动生成了公共模块 Common 、默认模块 Home 和 Runtime 运行时目录，如下所示：

```
Application
├─Common        应用公共模块
│  ├─Common     应用公共函数目录
│  └─Conf       应用公共配置文件目录
├─Home          默认生成的Home模块
│  ├─Conf       模块配置文件目录
│  ├─Common     模块函数公共目录
│  ├─Controller 模块控制器目录
│  ├─Model      模块模型目录
│  └─View       模块视图文件目录
├─Runtime       运行时目录
│  ├─Cache      模版缓存目录
│  ├─Data       数据目录
│  ├─Logs       日志目录
│  └─Temp       缓存目录模块设计
```

3.2版本采用模块化的设计架构，下面是一个典型的模块目录结构，每个模块可以方便的卸载和部署，并且支持公共模块（Runtime目录非模块目录）。

```
Application      默认应用目录（可以设置）
├─Common         公共模块（不能直接访问）
├─Home           前台模块
├─Admin          后台模块
├─...            其他更多模块
├─Runtime        默认运行时目录（可以设置）每个模块是相对独立的，其目录结构如下：
├─Module         模块目录
│  ├─Conf        配置文件目录
│  ├─Common      公共函数目录
│  ├─Controller  控制器目录
│  ├─Model       模型目录
│  ├─Logic       逻辑目录（可选）
│  ├─Service     服务目录（可选）
│  ... 更多分层目录可选
│  └─View        视图目录
```

由于采用多层的MVC机制，除了Conf和Common目录外，每个模块下面的目录结构可以根据需要灵活设置和添加，所以并不拘泥于上面展现的目录。

如果我要添加新的模块，有没有快速生成模块目录结构的办法呢？只需要在入口文件中添加如下定义（假设要生成Admin模块）：

```
define('APP_PATH','./Application/');
// 绑定入口文件到Admin模块访问
define('BIND_MODULE','Admin');
require './ThinkPHP/ThinkPHP.php';
```

BIND_MODULE常量定义表示绑定入口文件到某个模块，由于并不存在Admin模块，所以会在第一次访问的时候自动生成。重新访问入口文件后，就会再次看到欢迎页面，这个时候在Application下面已经自动生成了Admin模块及其目录结构。

> 注意：生成以后，你需要删除（或者注释掉）刚才添加的那段常量定义才能正常访问Home模块，否则就只能访问Admin模块（因为应用入口中已经绑定了Admin模块）。

```
define('APP_PATH','./Application/');
// 注释掉绑定模块的定义
// define('BIND_MODULE','Admin');
require './ThinkPHP/ThinkPHP.php';
```

有些情况下，我们需要更改应用目录、运行时目录和框架的位置，那么可以修改入口文件如下：

```
// 定义应用目录
define('APP_PATH','./Apps/');
// 定义运行时目录
define('RUNTIME_PATH','./Runtime/');
// 更名框架目录名称，并载入框架入口文件
require './Think/ThinkPHP.php';
```

这样最终的应用目录结构如下：

```
www  WEB部署目录（或者子目录）
├─index.php       应用入口文件
├─Apps            应用目录
├─Public          资源文件目录
├─Runtime         运行时目录
└─Think           框架目录
```

调试模式

ThinkPHP支持调试模式，默认情况下是运行在部署模式下面。部署模式下面性能优先，并且尽可能少地抛出错误信息，调试模式则以除错方便优先，关闭任何缓存，而且尽可能多的抛出错误信息，所以对性能有一定的影响。

部署模式采用了项目编译机制，第一次运行会对核心和项目相关文件进行编译缓存，由于编译后会影响开发过程中对配置文件、函数文件和数据库修改的生效（除非你修改后手动清空Runtime下面的缓存文件）。因此为了避免以上问题，我们强烈建议新手在使用ThinkPHP开发的过程中使用调试模式，这样可以更好的获取错误提示和避免一些不必要的问题和烦恼。

开启调试模式很简单，我们只需要在入口文件的开头加上一行常量定义代码：

```
define('APP_DEBUG', true); // 开启调试模式
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

开发完成后，我们实际进行项目部署的时候，删除这行常量定义代码即可，或者改成：

```
define('APP_DEBUG',false); // 关闭调试模式
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

> 为了安全考虑，避免泄露你的服务器WEB目录信息等资料，一定记得正式部署的时候关闭调试模式。

配置

每个应用模块都有独立的配置文件（位于模块目录的 Conf/config.php ），配置文件的定义格式支

持**PHP/JSON/YAML/INI/XML**等方式，默认采用PHP数组定义，例如：

```
// 配置文件
return array(
    '配置参数' => '配置值',
    // 更多配置参数
    //...
);
```

如果你需要为各个模块定义公共的配置文件，可以在公共模块中定义（通常位于`Common/Conf/config.php`），定义格式是一样。

一旦有需要，我们就可以在配置文件中添加相关配置项目。通常我们提到的添加配置项目，就是指在项目配置文件中添加：

```
'配置参数'=>'配置值',
```

配置值可以支持包括字符串、数字、布尔值和数组在内的数据，通常我们建议配置参数均使用大写定义。

如果有需要，我们还可以为项目定义其他类型的配置文件，如果要使用其他格式的配置文件，可以在入口文件中定义**CONF_EXT**常量即可，例如：

```
define('CONF_EXT','.ini');
```

这样，模块的配置文件就变成了`Conf/config.ini`，定义格式如下：

```
DEFAULT_MODULE = Index ;默认模块
URL_MODEL      = 2 ;URL模式
```

更多的配置定义请参考后续的内容。

控制器

需要为每个控制器定义一个控制器类，控制器类的命名规范是：
控制器名+Controller.class.php（模块名采用驼峰法并且首字母大写）

系统的默认控制器是Index，对应的控制器就是模块目录下面的`Controller/IndexController.class.php`，类名和文件名一致。默认操作是index，也就是控制器的一个public方法。初次生成项目目录结构的时候，系统已经默认生成了一个默认控制器（就是之前看到的欢迎页面），我们把index方法改成下面的代码：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        echo 'hello,thinkphp!';
    }
}
```

再次访问入口文件的时候，在浏览器中看到默认的欢迎页面已经改成如下输出：

`hello,thinkphp!`

可以为操作方法定义参数，例如：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello($name='thinkphp'){
        echo 'hello,'.$name.'!';
    }
}
```

当我们带name参数访问入口文件地址（例如 `http://localhost/index.php/home/index/hello/name/baby` ）的时候，在浏览器中可以看到如下输出：

`hello,baby!`

一个模块可以包括多个操作方法，但如果你的操作方法是protected或者private类型的话，是无法直接通过URL访问到该操作的。

我们修改Index控制器类的方法如下：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello(){
        echo 'hello,thinkphp!';
    }

    public function test(){
        echo '这是一个测试方法!';
    }

    protected function hello2(){
        echo '只是protected方法!';
    }

    private function hello3(){
        echo '这是private方法!';
    }
}
```

当我们访问hello2和hello3操作方法后的结果都会显示非法操作：

URL请求

ThinkPHP采用单一入口模式访问应用，对应用的所有请求都定向到应用的入口文件，系统会从URL参数中解析当前请求的模块、控制器和操作，下面是一个标准的URL访问格式：
`http://serverName/index.php/模块/控制器/操作`

如果我们直接访问入口文件的话，由于URL中没有模块、控制器和操作，因此系统会访问默认模块（Home）下面的默认控制器（Index）的默认操作（index），因此下面的访问是等效的：

```
http://serverName/index.php
http://serverName/index.php/Home/Index/index
```

这种URL模式就是系统默认的PATHINFO模式，不同的URL模式获取模块和操作的方法不同，ThinkPHP支持的URL模式有四种：**普通模式、PATHINFO、REWRITE和兼容模式。**

1 普通模式

普通模式也就是使用传统的GET传参方式来指定当前访问的模块、控制器和操作，例如：

```
http://localhost/?m=home&c=index&a=hello&name=thinkphp
```

m参数表示模块，c表示控制器，a表示操作（当然，这些参数名是可以配置的），后面的表示其他GET参数。
默认值可以不传，因此下面的URL访问是和上面的等效：

```
http://localhost/?a=hello&name=thinkphp
```

2 PATHINFO模式
PATHINFO模式是系统的默认URL模式，提供了最好的SEO支持，系统内部已经做了环境的兼容处理，所以能够支持大多数的主机环境。
对应上面的URL模式，PATHINFO模式下面的URL访问地址是：

```
http://localhost/index.php/home/index/hello/name/thinkphp/
```

PATHINFO地址的前三个参数分别表示模块/控制器/操作。

PATHINFO模式下面，也可以用普通模式的参数方式传入参数，例如：

```
http://localhost/index.php/home/index/hello?name=thinkphp
```

PATHINFO模式下面，URL参数分隔符是可定制的，例如，通过下面的配置：

```
'URL_PATHINFO_DEPR'=>'-',// 更改PATHINFO参数分隔符
```

我们可以支持下面的URL访问：

```
http://localhost/index.php/home-index-hello-name-thinkphp
```

3 REWRITE模式
REWRITE模式是在PATHINFO模式的基础上添加了重写规则的支持，可以去掉URL地址里面的入口文件index.php，但是需要额外配置WEB服务器的重写规则。
如果是Apache则需要在入口文件的同级添加.htaccess文件，内容如下：

```
<IfModule mod_rewrite.c>
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php/$1 [QSA,PT,L]
</IfModule>
```

接下来就可以使用下面的URL地址访问了

`http://localhost/home/index/hello/name/thinkphp/`

4 兼容模式

兼容模式是用于不支持PATHINFO的特殊环境，URL地址是：

`http://localhost/?s=/home/index/hello/name/thinkphp`

> 兼容模式配合Web服务器重写规则的定义，可以达到和REWRITE模式一样的URL效果。

视图

ThinkPHP内置了一个编译型模板引擎，也支持原生的PHP模板，并且还提供了包括Smarty在内的模板引擎驱动。和Smarty不同，ThinkPHP在渲染模板的时候如果不指定模板，则会采用系统默认的定位规则，其定义规范默认是模块目录下面的 `View/控制器名/操作名.html` ，所以，Index模块的hello操作的默认模板文件位于Home模块目录下面的 `View/Index/hello.html` ，我们添加模板内容如下：

```
<html>
<head>
<title>hello {$name}</title>
</head>
<body>
    hello, {$name}!
</body>
</html>
```

要输出视图，必须在控制器方法中进行模板渲染输出操作，例如：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello($name='thinkphp'){
        $this->assign('name',$name);
        $this->display();
    }
}
```

display方法中我们没有指定任何模板，所以按照系统默认的规则输出了Index/hello.html模板文件。

接下来，我们在浏览器访问输出：

`hello,thinkphp!`

读取数据

在开始之前，我们首先在数据库thinkphp中创建一个think_data数据表（以mysql数据库为例）：

```
CREATE TABLE IF NOT EXISTS `think_data`(
`id`int(8)unsigned NOT NULL AUTO_INCREMENT,
`data` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 ;
INSERT INTO `think_data`(`id`,`data`) VALUES
(1,'thinkphp'),
(2,'php'),
(3,'framework');
```

如果我们需要读取数据库中的数据，就需要在模块配置文件中添加数据库连接信息如下：

```
// 添加数据库配置信息
'DB_TYPE'=>'mysql',// 数据库类型
'DB_HOST'=>'127.0.0.1',// 服务器地址
'DB_NAME'=>'thinkphp',// 数据库名
'DB_USER'=>'root',// 用户名
'DB_PWD'=>'',// 密码
'DB_PORT'=>3306,// 端口
'DB_PREFIX'=>'think_',// 数据库表前缀
'DB_CHARSET'=>'utf8',// 数据库字符集
```

接下来，我们修改下控制器方法，添加读取数据的代码：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    public function index(){
        $Data     = M('Data');// 实例化Data数据模型
        $result     = $Data->find(1);
        $this->assign('result',$result);
        $this->display();
    }
}
```

这里用到了M函数，是ThinkPHP内置的实例化模型的方法，而且用M方法实例化模型不需要创建对应的模型类，你可以理解为M方法是直接在操作底层的Model类，而Model类具备基本的CURD操作方法。

M('Data') 实例化后，就可以对 think_data 数据表（ think_ 是我们在项目配置文件中定义的数据表前缀）进行操作（包括CURD）了，M函数的用法还有很多，我们以后会深入了解。

定义好控制器后，我们修改模板文件，添加数据输出标签如下：

```
<html>
<head>
<title></title>
</head>
<body>
{$result.id}--{$result.data}
</body>
</html>
```

模板标签的用法和Smarty类似，就是用于输出数据的字段，这里就表示输出 think_data 表的id和data字段的值。

我们访问会输出：

```
1--thinkphp
```

如果发生错误，检查你是否开启了调试模式或者清空Runtime目录下面的缓存文件。

如果你看到了上面的输出结果，那么恭喜你已经拿到了入门ThinkPHP的钥匙！

总结

本篇我们学习了ThinkPHP的目录结构、URL模式，如何创建项目的入口文件和开启调试模式，以及控制器、模板和模型的基础认识，后面会继续了解对数据的CURD操作。

快速入门 2：CURD

快速入门（二）：CURD

上一篇中，我们了解了ThinkPHP的基础部分，以及如何创建一个控制器和模板，并知道了M方法的用法，本篇将会讲解下数据的CURD操作，探索下更多的数据操作。

CURD

CURD是一个数据库技术中的缩写词，一般的项目开发的各种参数的基本功能都是CURD。它代表 创建（Create）、 更新（Update）、 读取（Read）和 删除（Delete） 操作。CURD 定义了用于处理数据的基本原子操作。之所以将CURD 提升到一个技术难题的高度是因为完成一个涉及在多个数据库系统中进行CURD操作的汇总相关的活动，其性能可能会随数据关系的变化而有非常大的差异。

> CURD在具体的应用中并非一定使用create、update 、read和delete字样的方法，但是他们完成的功能是一致的。例如，ThinkPHP就是使用add、save、select和delete方法表示模型的CURD操作。

创建数据

大多数情况下，CURD的Create操作通常会通过表单来提交数据，首先，我们在Home模块的View/Form目录下面创建一个add.html 模板文件，内容为：

```
<FORM method="post" action="__URL__/insert">
标题：<INPUT type="text" name="title"><br/>
内容：<TEXTAREA name="content" rows="5" cols="45"></TEXTAREA><br/>
<INPUT type="submit" value="提交">
</FORM>
```

然后，我们还需要在Home模块的Controller目录下面创建一个FormController.class.php文件，暂时只需要定义FormController类，不需要添加任何操作方法，代码如下：

```
<?php
namespace Home\Controller;
use Think\Controller;
class FormController extends Controller{
}
```

接下来，访问
http://localhost/app/index.php/home/Form/add

就可以看到表单页面了，我们并没有在控制器里面定义add操作方法，但是很显然，访问是正常的。因为

ThinkPHP在没有找到对应操作方法的情况下，会检查是否存在对应的模板文件，由于我们有对应的add模板文件，所以控制器就直接渲染该模板文件输出了。所以说对于没有任何实际逻辑的操作方法，我们只需要直接定义对应的模板文件就行了。

我们可以看到，在表单中定义了提交地址是到Form控制器的insert操作，为了处理表单提交数据，我们需要在FormController类中添加insert操作方法，如下：

```
<?php
namespace Home\Controller;
use Think\Controller;
class FormController extends Controller{
    public function insert(){
        $Form    =    D('Form');
        if($Form->create()) {
            $result =    $Form->add();
            if($result) {
                $this->success('数据添加成功！');
            }else{
                $this->error('数据添加错误！');
            }
        }else{
            $this->error($Form->getError());
        }
    }
}
```

如果你的主键是自增类型的话，add方法的返回值就是该主键的值。不是自增主键的话，返回值表示插入数据的个数。如果返回false则表示写入出错。

模型

为了方便测试，我们首先在数据库中创建一个think_form表：

```
CREATE TABLE IF NOT EXISTS `think_form` (
  `id` smallint(4) unsigned NOT NULL AUTO_INCREMENT,
  `title` varchar(255) NOT NULL,
  `content` varchar(255) NOT NULL,
  `create_time` int(11) unsigned NOT NULL,
  PRIMARY KEY (`id`)
 ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 ;
```

我们在insert操作方法中用了D函数，和M函数不同，D函数需要有对应的模型类，下面我们就来创建模型类。

模型类的定义规范是：

```
模型名+Model.class.php（模型名的定义采用驼峰法并且首字母大写）
```

我们在Home模块的Model目录下面创建FormModel.class.php文件，添加代码如下：

```
<?php
namespace Home\Model;
use Think\Model;
class FormModel extends Model {
    // 定义自动验证
    protected $_validate    =    array(
        array('title','require','标题必须'),
        );
    // 定义自动完成
    protected $_auto    =    array(
        array('create_time','time',1,'function'),
        );
}
```

主要是用于表单的自动验证和自动完成，具体用法我们会用另外的篇幅单独讲述，这里暂时先略过。我们只要了解的是，如果使用D函数实例化模型类，一般需要对应一个数据模型类，而且create方法会自动把表单提交的数据进行自动验证和自动完成（如果有定义的话），如果自动验证失败，就可以通过模型的getError方法获取验证提示信息，如果验证通过，就表示数据对象已经成功创建，但目前只是保存在内存中，直到我们调用add方法写入数据到数据库。这样就完成了一个完整的Create操作，所以可以看到ThinkPHP在创建数据的过程中使用了两步：

- 第一步，create方法创建数据对象
- 第二步，使用add方法把当前的数据对象写入数据库

当然，你完全可以跨过第一步，直接进行第二步，但是这样的预处理有几个优势：

1. 无论表单有多复杂，create方法都可以用一行代码轻松创建数据对象；
2. 在写入数据之前，可以对数据进行验证和补充；

其实create方法还有很多的功能操作，目的只有一个，确保写入数据库的数据安全和有效。

我们来验证下表单提交的效果，当我们不输入标题就直接提交表单的话，系统会给出标题必须这样的提示信息。

标题必须！

页面自动 跳转 等待时间： 2

当我们顺利提交表单后，会看到写入数据表的数据中的 create_time 字段已经有值了，这就是通过模型的自动完成写入的。

如果你的数据完全是内部操作写入而不是通过表单的话（也就是说可以充分信任数据的安全），那么可以直接使用add方法，如：

```
$Form = D('Form');
$data['title'] = 'ThinkPHP';
$data['content'] = '表单内容';
$Form->add($data);
```

也可以支持对象方式操作：

```
$Form = D('Form');
$Form->title = 'ThinkPHP';
$Form->content = '表单内容';
$Form->add();
```

对象方式操作的时候，add方法无需传入数据，会自动识别当前的数据对象赋值。

读取数据

当我们成功写入数据后，就可以进行数据读取操作了。在前面一篇中，我们已经知道可以用select方法获取数据集，这里我们来通过find方法获取一个单一数据，定义read操作方法如下：

```
public function read($id=0){
    $Form    =    M('Form');
    // 读取数据
    $data =    $Form->find($id);
    if($data) {
        $this->assign('data',$data);// 模板变量赋值
    }else{
        $this->error('数据错误');
    }
    $this->display();
}
```

read操作方法有一个参数$id，表示我们可以接受URL里面的id变量（后面我们会在变量章节详细描述。这里之所以用M方法而没有用D方法，是因为find方法是基础模型类Model中的方法，所以没有必要浪费开销去实例化FormModel类（即使已经定义了FormModel类）。我们通常采用find方法读取某个数据，这里使用了AR模式来操作，所以没有传入查询条件，find($id) 表示读取主键为$id值的数据，find方法的返回值是一个如下格式的数组：

```
array(
    'id'        =>  5,
    'title'     =>  '测试标题',
    'content'   =>  '测试内容',
    'status'    =>  1,
)
```

然后我们可以在模板中输出数据，添加一个read模板文件，

```
<table>
 <tr>
    <td>id:</td>
    <td>{$data.id}</td>
 </tr>
 <tr>
    <td>标题：</td>
    <td>{$data.title}</td>
 </tr>
 <tr>
    <td>内容：</td>
    <td>{$data.content}</td>
 </tr>
 </table>
```

完成后，我们就可以访问
`http://localhost/app/index.php/home/Form/read/id/1`

来查看了。

如果你只需要查询某个字段的值，还可以使用getField方法，例如：

```
$Form = M("Form");
 // 获取标题
$title = $Form->where('id=3')->getField('title');
```

上面的用法表示获取id值为3的数据的title字段值。其实getField方法有很多用法，但是获取某个字段的值是getField方法最常规的用法。
查询操作是最常用的操作，尤其是涉及到复杂的查询条件，我们会在查询语言一章对查询进行更加详细的讲解。

更新数据

在成功写入并读取数据之后，我们就可以对数据进行编辑操作了，首先我们添加一个编辑表单的模板文件edit.html，如下：

```
<FORM method="post" action="__URL__/update">
标题：<INPUT type="text" name="title" value="{$vo.title}"><br/>
内容：<TEXTAREA name="content" rows="5" cols="45">{$vo.content}</TEXTAREA><br/>
<INPUT type="hidden" name="id" value="{$vo.id}">
<INPUT type="submit" value="提交">
</FORM>
```

编辑模板不同于新增表单，需要对模板进行变量赋值，所以，我们这次需要在FormController类添加两个操作方法：

```
public function edit($id=0){
    $Form    =    M('Form');
    $this->assign('vo',$Form->find($id));
    $this->display();
 }
 public function update(){
    $Form    =    D('Form');
    if($Form->create()) {
        $result =    $Form->save();
        if($result) {
            $this->success('操作成功！');
        }else{
            $this->error('写入错误！');
        }
    }else{
        $this->error($Form->getError());
    }
 }
```

完成后，我们就可以访问
`http://localhost/app/index.php/home/Form/edit/id/1`

数据的更新操作在ThinkPHP使用save方法，可以看到，我们同样可以使用create方法创建表单提交的数据，而save方法则会自动把当前的数据对象更新到数据库，而更新的条件其实就是表的主键，这就是我们在编辑页面要把主键的值作为隐藏字段一起提交的原因。
如果更新操作不依赖表单的提交的话，就可以写成：

```
$Form = M("Form");
 // 要修改的数据对象属性赋值
$data['id'] = 5;
$data['title'] = 'ThinkPHP';
$data['content'] = 'ThinkPHP3.2.3版本发布';
$Form->save($data); // 根据条件保存修改的数据
```

save方法会自动识别数据对象中的主键字段，并作为更新条件。当然，你也可以显式的传入更新条件：

```
$Form = M("Form");
 // 要修改的数据对象属性赋值
$data['title'] = 'ThinkPHP';
$data['content'] = 'ThinkPHP3.2.3版本发布';
$Form->where('id=5')->save($data); // 根据条件保存修改的数据
```

也可以改成对象方式来操作：

```
$Form = M("Form");
 // 要修改的数据对象属性赋值
$Form->title = 'ThinkPHP';
$Form->content = 'ThinkPHP3.2.3版本发布';
$Form->where('id=5')->save(); // 根据条件保存修改的数据
```

数据对象赋值的方式，save方法无需传入数据，会自动识别。
save方法的返回值是影响的记录数，如果返回false则表示更新出错。

有些时候，我们只需要修改某个字段的值，就可以使用setField方法，而不需要每次都调用save方法。

```
$Form = M("Form");
 // 更改title值
$Form->where('id=5')->setField('title','ThinkPHP');
```

对于统计字段，系统还提供了更加方便的setInc和setDec方法。
例如：

```
$User = M("User"); // 实例化User对象
$User->where('id=5')->setInc('score',3); // 用户的积分加3
$User->where('id=5')->setInc('score'); // 用户的积分加1
$User->where('id=5')->setDec('score',5); // 用户的积分减5
$User->where('id=5')->setDec('score'); // 用户的积分减1
```

删除数据

删除数据很简单，只需要调用delete方法，例如：

```
$Form = M('Form');
$Form->delete(5);
```

表示删除主键为5的数据，delete方法可以删除单个数据，也可以删除多个数据，这取决于删除条件，例如：

```
$User = M("User"); // 实例化User对象
$User->where('id=5')->delete(); // 删除id为5的用户数据
$User->delete('1,2,5'); // 删除主键为1,2和5的用户数据
$User->where('status=0')->delete(); // 删除所有状态为0的用户数据
```

delete方法的返回值是删除的记录数，如果返回值是false则表示SQL出错，返回值如果为0表示没有删除任何数据。

总结

现在，你已经基本掌握了ThinkPHP的CURD操作了，并且学会了使用ThinkPHP的create、add、save和delete方法，还有两个对字段操作的getField和setField方法。下一篇，我们会更深入的了解下如何使用ThinkPHP提供的查询语言。

快速入门 3：查询语言

快速入门（三）：查询语言

上一篇中我们掌握了基本的数据CURD方法，但更多的情况下面，由于业务逻辑的差异，CURD操作往往不是那么简单，尤其是复杂的业务逻辑下面，这也是ActiveRecord模式的不足之处。ThinkPHP的查询语言配合连贯操作可以很好解决复杂的业务逻辑需求，本篇我们就首先来深入了解下框架的查询语言。

介绍

ThinkPHP内置了非常灵活的查询方法，可以快速的进行数据查询操作，查询条件可以用于读取、更新和删除等操作，主要涉及到where方法等连贯操作即可，无论是采用什么数据库，你几乎采用一样的查询方法（个别数据库例如Mongo在表达式查询方面会有所差异），系统帮你解决了不同数据库的差异性，因此我们把框架的这一查询方式称之为查询语言。查询语言也是ThinkPHP框架的ORM亮点，让查询操作更加简单易懂。下面来一一讲解查询语言的内涵。

查询方式

ThinkPHP可以支持直接使用字符串作为查询条件，但是大多数情况推荐使用索引数组或者对象来作为查询条件，因为会更加安全。

一、使用字符串作为查询条件

这是最传统的方式，但是安全性不高，例如：

```
$User = M("User"); // 实例化User对象
$User->where('type=1 AND status=1')->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE type=1 AND status=1
```

采用字符串查询的时候，我们可以配合使用新版提供的字符串条件的安全预处理机制，暂且不再细说。

二、使用数组作为查询条件

这种方式是最常用的查询方式，例如：

```
$User = M("User"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['status'] = 1;
 // 把查询条件传入查询方法
$User->where($condition)->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE `name`='thinkphp' AND status=1
```

如果进行多字段查询，那么字段之间的默认逻辑关系是 逻辑与 AND，但是用下面的规则可以更改默认的逻辑判断，通过使用 _logic 定义查询逻辑：

```
$User = M("User"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['account'] = 'thinkphp';
$condition['_logic'] = 'OR';
 // 把查询条件传入查询方法
$User->where($condition)->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE `name`='thinkphp' OR `account`='thinkphp'
```

三、使用对象方式来查询

这里以stdClass内置对象为例：

```
$User = M("User"); // 实例化User对象
 // 定义查询条件
$condition = new stdClass();
$condition->name = 'thinkphp';
$condition->status= 1;
$User->where($condition)->select();
```

最后生成的SQL语句和上面一样

```
SELECT * FROM think_user WHERE `name`='thinkphp' AND status=1
```

使用对象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，大多数情况下，我们建议采用数组方式更加高效。

表达式查询

上面的查询条件仅仅是一个简单的相等判断，可以使用查询表达式支持更多的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：

```
$map['字段名'] = array('表达式','查询条件');
```

表达式不分大小写，支持的查询表达式有下面几种，分别表示的含义是：

表达式	含义
EQ	等于（=）
NEQ	不等于（<>）
GT	大于（>）
EGT	大于等于（>=）
LT	小于（<）
ELT	小于等于（<=）
LIKE	模糊查询
[NOT] BETWEEN	（不在）区间查询
[NOT] IN	（不在）IN 查询
EXP	表达式查询，支持SQL语法

示例如下：
EQ ：等于（=）
例如：

```
$map['id']  = array('eq',100);
```

和下面的查询等效

```
$map['id']  = 100;
```

表示的查询条件就是 `id = 100`

NEQ： 不等于（<>）
例如：

```
$map['id']  = array('neq',100);
```

表示的查询条件就是 `id <> 100`

GT：大于（>）

例如：

```
$map['id']  = array('gt',100);
```

表示的查询条件就是 `id > 100`

EGT：大于等于（>=）
例如：

```
$map['id']  = array('egt',100);
```

表示的查询条件就是 `id >= 100`

LT：小于（<）
例如：

```
$map['id']  = array('lt',100);
```

表示的查询条件就是 `id < 100`

ELT： 小于等于（<=）
例如：

```
$map['id']  = array('elt',100);
```

表示的查询条件就是 `id <= 100`

[NOT] LIKE： 同sql的LIKE
例如：

```
$map['name'] = array('like','thinkphp%');
```

查询条件就变成 `name like 'thinkphp%'`
如果配置了DB_LIKE_FIELDS参数的话，某些字段也会自动进行模糊查询。例如设置了：

```
'DB_LIKE_FIELDS'=>'title|content'
```

的话，使用

```
$map['title'] = 'thinkphp';
```

查询条件就会变成 `title like '%thinkphp%'`

支持数组方式，例如

```
$map['a'] =array('like',array('%thinkphp%','%tp'),'OR');
$map['b'] =array('notlike',array('%thinkphp%','%tp'),'AND');
```

生成的查询条件就是：

```
(a like '%thinkphp%' OR a like '%tp') AND (b not like '%thinkphp%' AND b not like '%tp')
```

[NOT] BETWEEN ：同sql的[not] between， 查询条件支持字符串或者数组，例如：

```
$map['id']  = array('between','1,8');
```

和下面的等效：

```
$map['id']  = array('between',array('1','8'));
```

查询条件就变成 `id BETWEEN 1 AND 8`

[NOT] IN： 同sql的[not] in ，查询条件支持字符串或者数组，例如：

```
$map['id']  = array('not in','1,5,8');
```

和下面的等效：

```
$map['id']  = array('not in',array('1','5','8'));
```

查询条件就变成 `id NOT IN (1,5, 8)`

EXP：表达式，支持更复杂的查询情况
例如：

```
$map['id']  = array('in','1,3,8');
```

可以改成：

```
$map['id']  = array('exp',' IN (1,3,8) ');
```

exp查询的条件不会被当成字符串，所以后面的查询条件可以使用任何SQL支持的语法，包括使用函数和字段名称。查询表达式不仅可用于查询条件，也可以用于数据更新，例如：

```
$User = M("User"); // 实例化User对象
 // 要修改的数据对象属性赋值
$data['name'] = 'ThinkPHP';
$data['score'] = array('exp','score+1');// 用户的积分加1
$User->where('id=5')->save($data); // 根据条件保存修改的数据
```

快捷查询

采用快捷查询方式，可以进一步简化查询条件的写法，例如：

一、实现不同字段相同的查询条件

```
$User = M("User"); // 实例化User对象
$map['name|title'] = 'thinkphp';
 // 把查询条件传入查询方法
$User->where($map)->select();
```

查询条件就变成

`name= 'thinkphp' OR title = 'thinkphp'`

二、实现不同字段不同的查询条件

```
$User = M("User"); // 实例化User对象
$map['status&title'] =array('1','thinkphp','_multi'=>true);
 // 把查询条件传入查询方法
$User->where($map)->select();
```

'_multi'=>true必须加在数组的最后，表示当前是多条件匹配，这样查询条件就变成

`status= 1 AND title = 'thinkphp'`

，查询字段支持更多的，例如：

```
$map['status&score&title'] =array('1',array('gt','0'),'thinkphp','_multi'=>true);
```

查询条件就变成

`status= 1 AND score >0 AND  title = 'thinkphp'`

> 注意：快捷查询方式中“|”和“&”不能同时使用。

区间查询

ThinkPHP支持对某个字段的区间查询，例如：

```
$map['id'] = array(array('gt',1),array('lt',10)) ;
```

得到的查询条件是：

```
(`id` > 1) AND (`id` < 10)
```

```
$map['id'] = array(array('gt',3),array('lt',10), 'or') ;
```

得到的查询条件是：

```
(`id` > 3) OR (`id` < 10)
```

```
$map['id']  = array(array('neq',6),array('gt',3),'and');
```

得到的查询条件是：

```
(`id` != 6) AND (`id` > 3)
```

最后一个可以是AND、 OR或者 XOR运算符，如果不写，默认是AND运算。

区间查询的条件可以支持普通查询的所有表达式，也就是说类似LIKE、GT和EXP这样的表达式都可以支持。另外区间查询还可以支持更多的条件，只要是针对一个字段的条件都可以写到一起，例如：

```
$map['name']  = array(array('like','%a%'), array('like','%b%'), array('like','%c%'), 'ThinkPHP','or');
```

最后的查询条件是：

```
(`name` LIKE '%a%') OR (`name` LIKE '%b%') OR (`name` LIKE '%c%') OR (`name` = 'ThinkPHP')
```

组合查询

组合查询的主体还是采用数组方式查询，只是加入了一些特殊的查询支持，包括字符串模式查询（_string）、复合查询（_complex）、请求字符串查询（_query），混合查询中的特殊查询每次查询只能定义一个，由于采用数组的索引方式，索引相同的特殊查询会被覆盖。

一、字符串模式查询（采用_string 作为查询条件）

数组条件还可以和字符串条件混合使用，例如：

```
$User = M("User"); // 实例化User对象
$map['id'] = array('neq',1);
$map['name'] = 'ok';
$map['_string'] = 'status=1 AND score>10';
$User->where($map)->select();
```

最后得到的查询条件就成了：

```
( `id` != 1 ) AND ( `name` = 'ok' ) AND ( status=1 AND score>10 )
```

二、请求字符串查询方式

请求字符串查询是一种类似于URL传参的方式，可以支持简单的条件相等判断。

```
$map['id'] = array('gt','100');
$map['_query'] = 'status=1&score=100&_logic=or';
```

得到的查询条件是：

```
`id`>100 AND (`status` = '1' OR `score` = '100')
```

三、复合查询

复合查询相当于封装了一个新的查询条件，然后并入原来的查询条件之中，所以可以完成比较复杂的查询条件组装。
例如：

```
$where['name']  = array('like', '%thinkphp%');
$where['title']  = array('like','%thinkphp%');
$where['_logic'] = 'or';
$map['_complex'] = $where;
$map['id']  = array('gt',1);
```

查询条件是

```
( id > 1) AND ( ( name like '%thinkphp%') OR ( title like '%thinkphp%') )
```

复合查询使用了_complex作为子查询条件来定义，配合之前的查询方式，可以非常灵活的制定更加复杂的查询条件。
很多查询方式可以相互转换，例如上面的查询条件可以改成：

```
$where['id'] = array('gt',1);
$where['_string'] = ' (name like "%thinkphp%")  OR ( title like "%thinkphp") ';
```

最后生成的SQL语句是一致的。

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，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）

用法示例：

```
$User = M("User"); // 实例化User对象
// 获取用户数：
$userCount = $User->count();
// 或者根据字段统计：
$userCount = $User->count("id");
// 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->where('score>0')->min('score');
// 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');
```

并且所有的统计查询均支持连贯操作的使用。

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。主要包括下面两个方法：

1、query方法

query	执行SQL查询操作
用法	query($sql,$parse=false)
参数	sql（必须）：要查询的SQL语句 parse（可选）：是否需要解析SQL
返回值	如果数据非法或者查询错误则返回false，否则返回查询结果数据集（同select方法）

使用示例：

```
$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");
```

> 如果你当前采用了分布式数据库，并且设置了读写分离的话，query方法始终是在读服务器执行，因此query方法对应的都是读操作，而不管你的SQL语句是什么。

2、execute方法

execute	用于更新和写入数据的sql操作
用法	execute($sql,$parse=false)
参数	sql（必须）：要执行的SQL语句 parse（可选）：是否需要解析SQL
返回值	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

使用示例：

```
$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");
```

如果你当前采用了分布式数据库，并且设置了读写分离的话，execute方法始终是在写服务器执行，因此execute方法对应的都是写操作，而不管你的SQL语句是什么。

动态查询

借助PHP5语言的特性，ThinkPHP实现了动态查询，核心模型的动态查询方法包括下面几种：

方法名	说明	举例
getBy	根据字段的值查询数据	例如，getByName,getByEmail
getFieldBy	根据字段查询并返回某个字段的值	例如，getFieldByName

一、getBy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合条件的记录。

```
$user = $User->getByName('liu21st');
$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select方法进行查询。

二、getFieldBy动态查询

针对某个字段查询并返回某个字段的值，例如

```
$userId = $User->getFieldByName('liu21st','id');
```

表示根据用户的name获取用户的id值。

子查询

子查询有两种使用方式：

1、使用select方法

当select方法的参数为false的时候，表示不进行查询只是返回构建SQL，例如：

```
// 首先构造子查询SQL
$subQuery = $model->field('id,name')->table('tablename')->group('field')->where($where)->order('status')->select(false);
```

当select方法传入false参数的时候，表示不执行当前查询，而只是生成查询SQL。

2、使用buildSql方法

```
$subQuery = $model->field('id,name')->table('tablename')->group('field')->where($where)->order('status')->buildSql();
```

调用buildSql方法后不会进行实际的查询操作，而只是生成该次查询的SQL语句（为了避免混淆，会在SQL两边加上括号），然后我们直接在后续的查询中直接调用。

```
// 利用子查询进行查询
$model->table($subQuery.' a')->where()->order()->select()
```

构造的子查询SQL可用于ThinkPHP的连贯操作方法，例如table where等。

总结

本篇主要帮助我们了解如何进行数据的查询，包括简单查询、表达式查询、快捷查询、区间查询、统计查询，以及如何进行子查询操作。后面我们还会详细了解如何使用连贯操作进行更复杂的CURD操作。

快速入门 4：连贯操作

快速入门（四）：连贯操作

上一篇我们详细描述了查询语言的用法，但是查询语言仅仅解决了查询或者操作条件的问题，更多的配合还需要使用模型提供的连贯操作方法。

介绍

连贯操作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，也是ThinkPHP的ORM中的一个亮点。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：

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

这里的where、order和limit方法就称之为连贯操作方法，除了select方法必须放到最后一个外（因为select方法并不是连贯操作方法），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')->limit(10)->where('status=1')->select();
```

其实不仅仅是查询方法可以使用连贯操作，包括所有的CURD方法都可以使用，例如：

```
$User->where('id=1')->field('id,name,email')->find();
$User->where('status=1 and id=1')->delete();
```

连贯操作仅在当次查询或者操作有效，完成后会自动清空连贯操作的所有传值（有个别特殊的连贯操作会记录当前的传值，如cache连贯操作）。简而言之，连贯操作的结果不会带入以后的查询。
系统支持的连贯操作方法有：

方法	作用	支持的参数类型
where*	用于查询或者更新条件的定义	字符串、数组和对象
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
data	用于新增或者更新数据之前的数据对象赋值	数组和对象
field	用于定义要查询的字段（支持字段排除）	字符串和数组

方法	作用	支持的参数类型
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
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（以后在缓存部分再详细描述）
relation	用于关联查询（需要关联模型扩展支持）	字符串
validate	用于数据自动验证	数组
auto	用于数据自动完成	数组
filter	用于数据过滤	字符串
scope*	用于命名范围	字符串、数组
bind*	用于数据绑定操作	数组或多个参数
token	用于令牌验证	布尔值
comment	用于SQL注释	字符串
index	用于数据集的强制索引	字符串
strict	用于数据入库的严格检测	布尔值

所有的连贯操作都返回当前的模型实例对象，其中带*标识的表示支持多次调用。

用法

由于连贯操作的使用往往涉及到多个方法的联合使用，下面大概介绍下各个连贯操作的基本用法：

WHERE

where	用于查询或者更新条件的定义
用法	where($where)
参数	where（必须）：查询或者操作条件，支持字符串、数组和对象
返回值	当前模型实例

where	用于查询或者更新条件的定义

> 备注 如果不调用where方法，默认不会执行更新和删除操作

Where方法是使用最多的连贯操作方法，更详细的用法请参考：快速入门（3）查询语言。

TABLE

table	定义要操作的数据表名称，动态改变当前操作的数据表名称，需要写数据表的全名，包含前缀，可以使用别名和跨库操作
用法	table($table)
参数	table（必须）：数据表名称，支持操作多个表，支持字符串、数组和对象
返回值	当前模型实例

> 备注:如果不调用table方法，会自动获取模型对应或者定义的数据表

用法示例：

```
$Model->Table('think_user user')->where('status>1')->select();
```

也可以在table方法中跨库操作，例如：

```
$Model->Table('db_name.think_user user')->where('status>1')->select();
```

Table方法的参数支持字符串和数组，数组方式的用法：

```
$Model->Table(array('think_user'=>'user','think_group'=>'group'))->where('status>1')->select();
```

使用数组方式定义的优势是可以避免因为表名和关键字冲突而出错的情况。

一般情况下，无需调用table方法，默认会自动获取当前模型对应或者定义的数据表。

DATA

data	可以用于新增或者保存数据之前的数据对象赋值
用法	data($data)
参数	data（必须）：数据，支持数组和对象
返回值	当前模型实例

> 备注：如果不调用data方法，则会取当前的数据对象或者传入add和save的数据

使用示例：

```
$Model->data($data)->add();
$Model->data($data)->where('id=3')->save();
```

Data方法的参数支持对象和数组，如果是对象会自动转换成数组。如果不定义data方法赋值，也可以使用create方法或者手动给数据对象赋值的方式。

模型的data方法除了创建数据对象之外，还可以读取当前的数据对象，
例如：

```
$this->find(3);
$data = $this->data();
```

FIELD

field	用于定义要查询的字段
用法	field($field,$except=false)
参数	field（必须）：字段名，支持字符串和数组，支持指定字段别名；如果为true则表示显式或者数据表的所有字段。except（可选）：是否排除，默认为false，如果为true表示定义的字段为数据表中排除field参数定义之外的所有字段。
返回值	当前模型实例

> 备注：如果不调用field方法，则默认返回所有字段，和field（'*'）等效

使用示例：

```
$Model->field('id,nickname as name')->select();
$Model->field(array('id','nickname'=>'name'))->select();
```

如果不调用field方法或者field方法传入参数为空的话，和使用field（'*'）是等效的。
如果需要显式的传入所有的字段，可以使用下面的方法：

```
$Model->field(true)->select();
```

但是我们更建议只获取需要显式的字段名，或者采用字段排除方式来定义，例如：

```
$Model->field('status',true)->select();
```

表示获取除了status之外的所有字段。

ORDER

order	用于对操作结果排序
用法	order($order)
参数	order（必须）：排序的字段名，支持字符串和数组，支持多个字段排序
返回值	当前模型实例

> 备注：如果不调用order方法，按照数据库的默认规则

使用示例：

```
order('id desc')
```

排序方法支持对多个字段的排序

```
order('status desc,id asc')
```

order方法的参数支持字符串和数组，数组的用法如下：

```
order(array('status'=>'desc','id'))
```

LIMIT

limit	用于定义要查询的结果限制（支持所有的数据库类型）
用法	limit($limit)
参数	limit（必须）：限制数量，支持字符串
返回值	当前模型实例

> 备注：如果不调用limit方法，则表示没有限制

我们知道不同的数据库类型的limit用法是不尽相同的，但是在ThinkPHP的用法里面始终是统一的方法，也就是limit('offset,length')，无论是Mysql、SqlServer还是Oracle数据库，都是这样使用，系统的数据库驱动类会负责解决这个差异化。

使用示例：

```
limit('1,10')
```

也可以用下面的两个参数的写法，是等效的：

```
limit(1,10)
如果使用
limit('10')
等效于
limit('0,10')
```

PAGE

page	用于定义要查询的数据分页
用法	page($page)
参数	page（必须）：分页，支持字符串
返回值	当前模型实例

Page操作方法可以更加快速的进行分页查询。
Page方法的用法和limit方法类似，格式为：

```
Page('page[,listRows]')
```

Page表示当前的页数，listRows表示每页显示的记录数。例如：

```
Page('2,10')
```

表示每页显示10条记录的情况下面，获取第2页的数据。
listRow如果不写的话，会读取limit('length') 的值，例如：

```
limit(25)->page(3);
```

表示每页显示25条记录的情况下面，获取第3页的数据。
如果limit也没有设置的话，则默认为每页显示20条记录。
page方法也支持传入二个参数，例如：

```
$this->page(5,25)->select();
```

和之前的用法

```
$this->page('5,25')->select();
```

等效。

GROUP

group	用于数据库的group查询支持
用法	group($group)
参数	group（必须）：group的字段名，支持字符串
返回值	当前模型实例

使用示例：

```
group('user_id')
```

> Group方法的参数只支持字符串

HAVING

having	用于数据库的having查询支持
用法	having($having)
参数	having（必须）：having，支持字符串
返回值	当前模型实例

使用示例：

```
having('user_id>0')
```

> having方法的参数只支持字符串

JOIN

join	用于数据库的join查询支持
用法	join($join)
参数	join（必须）：join操作，支持字符串和数组
返回值	当前模型实例

> 备注：join方法支持多次调用

使用示例：

```
$Model->join(' work ON artist.id = work.artist_id')->join('card ON artist.card_id = card.id')->select();
```

默认采用JOIN （等同于 INNER JOIN）方式，如果需要用其他的JOIN方式，可以改成

```
$Model->join('RIGHT JOIN work ON artist.id = work.artist_id')->select();
```

如果join方法的参数用数组的话，只能使用一次join方法，并且不能和字符串方式混合使用。
例如：

```
join(array(' work ON artist.id = work.artist_id','card ON artist.card_id = card.id'))
```

UNION

union	**用于数据库的union查询支持**
用法	union($union,$all=false)
参数	union（必须）：union操作，支持字符串、数组和对象 all（可选）：是否采用UNION ALL操作，默认为false
返回值	当前模型实例

> 备注：Union方法支持多次调用
> 使用示例：

```
$Model->field('name')
    ->table('think_user_0')
    ->union('SELECT name FROM think_user_1')
    ->union('SELECT name FROM think_user_2')
    ->select();
```

数组用法：

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('field'=>'name','table'=>'think_user_1'))
    ->union(array('field'=>'name','table'=>'think_user_2'))
    ->select();
```

或者

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('SELECT name FROM think_user_1','SELECT name FROM think_user_2'))
    ->select();
```

支持UNION ALL 操作，例如：

```
$Model->field('name')
    ->table('think_user_0')
    ->union('SELECT name FROM think_user_1',true)
    ->union('SELECT name FROM think_user_2',true)
    ->select();
```

或者

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('SELECT name FROM think_user_1','SELECT name FROM think_user_2'),true)
    ->select();
```

每个union方法相当于一个独立的SELECT语句。

> 注意：UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

DISTINCT

distinct	**查询数据的时候进行唯一过滤**
用法	distinct($distinct)
参数	distinct（必须）：是否采用distinct，支持布尔值
返回值	当前模型实例

使用示例：

```
$Model->Distinct(true)->field('name')->select();
```

LOCK

lock	**用于查询或者写入锁定**
用法	lock($lock)
参数	lock（必须）：是否需要锁定，支持布尔值

lock	用于查询或者写入锁定
返回值	当前模型实例

> 备注：join方法支持多次调用

Lock方法是用于数据库的锁机制，如果在查询或者执行操作的时候使用：

```
lock(true)
```

就会自动在生成的SQL语句最后加上 FOR UPDATE或者FOR UPDATE NOWAIT（Oracle数据库）。

VALIDATE

validate	用于数据的自动验证
用法	validate($validate)
参数	validate（必须）：自动验证定义
返回值	当前模型实例

> 备注：只能和create方法配合使用

validate方法用于数据的自动验证，我们会在数据验证部分详细描述。

AUTO

auto	用于数据自动完成
用法	auto($auto)
参数	auto（必须）：定义自动完成
返回值	当前模型实例

> 备注：auto方法只能配合create方法使用

auto方法用于数据的自动完成操作，具体使用我们会在数据自动完成部分描述。

SCOPE

scope	用于模型的命名范围
用法	scope($scope)
参数	scope（必须）：命名范围定义
返回值	当前模型实例

> 备注：scope方法其实是连贯操作的预定义

scope方法的具体用法可以参考：3.1的新特性 命名范围

FILTER

filter	**用于数据的安全过滤**
用法	filter($filter)
参数	filter（必须）：过滤方法名
返回值	当前模型实例

> 备注：filter方法一般用于写入和更新操作
> filter方法用于对数据对象的安全过滤，例如：

```
$Model->data($data)->filter('strip_tags')->add();
```

目前filter方法不支持多个方法的过滤。

总结

连贯操作为我们的数据操作带来了很大的便捷之处，并且只要SQL可以实现的操作，基本上都可以用ThinkPHP的连贯操作来实现，并且不用考虑数据库之间的表达差异，具有可移植性。后面会和大家讲解如何操作和获取变量。

快速入门 5：变量

快速入门（五）：变量

本篇我们来学习如何在ThinkPHP中使用变量和对变量进行过滤。

在Web开发过程中，我们经常需要获取系统变量或者用户提交的数据，这些变量数据错综复杂，而且一不小心就容易引起安全隐患，但是如果利用好ThinkPHP提供的变量获取功能，就可以轻松的获取和驾驭变量了。

获取变量

虽然你仍然可以在开发过程中使用传统方式获取各种系统变量，例如：

```
$id    =  $_GET['id']; // 获取get变量
$name  =  $_POST['name'];  // 获取post变量
$value =  $_SESSION['var']; // 获取session变量
$name  =  $_COOKIE['name']; // 获取cookie变量
$file  =  $_SERVER['PHP_SELF']; // 获取server变量
```

但是我们不建议直接使用传统方式获取，因为没有统一的安全处理机制，后期如果调整的话，改起来会比较麻烦。所以，更好的方式是在框架中统一使用I函数进行变量获取和过滤。

I方法是ThinkPHP用于更加方便和安全的获取系统输入变量，可以用于任何地方，用法格式如下：

```
I('变量类型.变量名/修饰符',['默认值'],['过滤方法'],['额外数据源'])
```

变量类型是指请求方式或者输入类型，包括：

变量类型	含义
get	获取GET参数
post	获取POST参数
param	自动判断请求类型获取GET、POST或者PUT参数
request	获取REQUEST 参数
put	获取PUT 参数
session	获取 $_SESSION 参数
cookie	获取 $_COOKIE 参数
server	获取 $_SERVER 参数
globals	获取 $GLOBALS参数

变量类型	含义
path	获取 PATHINFO模式的URL参数
data	获取 其他类型的参数，需要配合额外数据源参数

> 注意：变量类型不区分大小写。
> 变量名则严格区分大小写。
> 默认值和过滤方法均属于可选参数。

我们以GET变量类型为例，说明下I方法的使用：

```
echo I('get.id'); // 相当于 $_GET['id']
echo I('get.name'); // 相当于 $_GET['name']
```

支持默认值：

```
echo I('get.id',0); // 如果不存在$_GET['id'] 则返回0
echo I('get.name',''); // 如果不存在$_GET['name'] 则返回空字符串
```

采用方法过滤：

```
// 采用htmlspecialchars方法对$_GET['name'] 进行过滤，如果不存在则返回空字符串
echo I('get.name','','htmlspecialchars');
```

支持直接获取整个变量类型，例如：

```
// 获取整个$_GET 数组
I('get.');
```

用同样的方式，我们可以获取post或者其他输入类型的变量，例如：

```
I('post.name','','htmlspecialchars'); // 采用htmlspecialchars方法对$_POST['name'] 进行过滤，如果不存在则返回空字符串
I('session.user_id',0); // 获取$_SESSION['user_id'] 如果不存在则默认为0
I('cookie.'); // 获取整个 $_COOKIE 数组
I('server.REQUEST_METHOD'); // 获取 $_SERVER['REQUEST_METHOD']
```

param变量类型是框架特有的支持自动判断当前请求类型的变量获取方式，例如：

`echo I('param.id');`

如果当前请求类型是GET，那么等效于 $_GET['id']，如果当前请求类型是POST或者PUT，那么相当于获取 $_POST['id'] 或者 PUT参数id。

由于param类型是I函数默认获取的变量类型，因此事实上param变量类型的写法可以简化为：

```
I('id'); // 等同于 I('param.id')
I('name'); // 等同于 I('param.name')
```

path类型变量可以用于获取PATHINFO方式的URL参数（必须是PATHINFO模式参数有效，无论是GET还是POST方式都有效），例如：
当前访问URL地址是
`http://serverName/index.php/New/2013/06/01`

那么我们可以通过

```
echo I('path.1'); // 输出2013
echo I('path.2'); // 输出06
echo I('path.3'); // 输出01
```

变量过滤

如果你没有在调用I函数的时候指定过滤方法的话，系统会采用默认的过滤机制（由DEFAULT_FILTER配置），事实上，该参数的默认设置是：

```
// 系统默认的变量过滤机制
'DEFAULT_FILTER'        => 'htmlspecialchars'
```

也就说，I方法的所有获取变量如果没有设置过滤方法的话都会进行htmlspecialchars过滤，那么：

```
// 等同于 htmlspecialchars($_GET['name'])
I('get.name');
```

同样，该参数也可以设置支持多个过滤，例如：

```
'DEFAULT_FILTER'        => 'strip_tags,htmlspecialchars'
```

设置后，我们在使用：

```
// 等同于 htmlspecialchars(strip_tags($_GET['name']))
I('get.name');
```

如果我们在使用I方法的时候 指定了过滤方法，那么就会忽略DEFAULT_FILTER的设置，例如：

```
// 等同于 strip_tags($_GET['name'])
echo I('get.name','','strip_tags');
```

I方法的第三个参数如果传入函数名，则表示调用该函数对变量进行过滤并返回（在变量是数组的情况下自动使用 array_map 进行过滤处理），否则会调用PHP内置的 filter_var 方法进行过滤处理，例如：

```
I('post.email','',FILTER_VALIDATE_EMAIL);
```

表示 会对 $_POST['email'] 进行 格式验证，如果不符合要求的话，返回空字符串。
（关于更多的验证格式，可以参考 官方手册的 filter_var 用法。）
或者可以用下面的字符标识方式：

```
I('post.email','','email');
```

可以支持的过滤名称必须是 filter_list 方法中的有效值（不同的服务器环境可能有所不同），可能支持的包括：

```
int
boolean
float
validate_regexp
validate_url
validate_email
validate_ip
string
stripped
encoded
special_chars
unsafe_raw
email
url
number_int
number_float
magic_quotes
callback
```

也可以支持正则匹配过滤，例如：

```
// 采用正则表达式进行变量过滤
I('get.name','','/^[A-Za-z]+$/');
I('get.id',0,'/^\d+$/');
```

如果正则匹配不通过的话，则返回默认值。

在有些特殊的情况下，我们不希望进行任何过滤，即使**DEFAULT_FILTER**已经有所设置，可以使用：

```
// 下面两种方式都不采用任何过滤方法
I('get.name','','');
I('get.id','',false);
```

一旦过滤参数设置为空字符串或者false，即表示不再进行任何的过滤。

变量修饰符

I函数支持对变量使用修饰符功能，可以更好的过滤变量。

用法如下：

I('变量类型.变量名/修饰符');

例如：

```
I('get.id/d');
I('post.name/s');
I('post.ids/a');
```

可以使用的修饰符包括：

修饰符	作用
s	强制转换为字符串类型
d	强制转换为整形类型
b	强制转换为布尔类型
a	强制转换为数组类型
f	强制转换为浮点类型

快速入门 6：路由

快速入门（六）：路由

ThinkPHP框架对URL有一定的规范，所以如果你希望定制你的URL格式的话，就需要好好了解下内置的路由功能了，它能让你的URL变得更简洁和有内涵。

路由定义

路由定义一般包括三个配置参数：

参数	描述
URL_ROUTER_ON	开启路由，设置为true后路由规则定义生效
URL_ROUTE_RULES	路由规则定义
URL_MAP_RULES	静态路由（URL映射）定义

要使用路由功能，前提是你的URL支持PATH_INFO（或者兼容URL模式也可以，采用普通URL模式的情况下不支持路由功能），并且在应用（或者模块）配置文件中开启路由：

```
'URL_ROUTER_ON'   => true, //开启路由
```

然后就是配置路由规则了，使用**URL_ROUTE_RULES**参数进行配置，配置格式是一个数组，每个元素都代表一个路由规则，例如：

```
'URL_ROUTE_RULES'=>array(
    'news/:year/:month/:day' => array('News/archive', 'status=1'),
    'news/:id'               => 'News/read',
    'news/read/:id'          => '/news/:1',
 ),
```

系统会按定义的顺序依次匹配路由规则，一旦匹配到的话，就会定位到路由定义中的模块、控制器和操作方法去执行（可以传入其他的参数），并且后面的规则不会继续匹配。

路由规则一般是配置在模块的配置文件中（`模块名/Conf/config.php`），这样的话路由规则仅针对当前模块，并且URL中匹配的路由不能省略当前模块名（除了默认模块外）。如果需要定义全局的路由规则，就需要把路由配置定义在公共模块的配置文件中（`Common/Conf/config.php`），但全局路由定义的路由规则中必须包含明确的模块。

路由规则的定义方式如下：

```
'路由表达式'=>'路由地址和额外参数'
```

路由表达式

路由表达式包括规则路由和正则路由的定义表达式，只能使用字符串。

表达式	示例
正则表达式	/^blog\/(\d+)$/
规则表达式	blog/:id

正则表达式

路由表达式支持的正则定义必须以“/”开头，否则就视为规则表达式。也就是说如果采用

```
'#^blog\/(\d+)$#'
```

方式定义的正则表达式不会被支持，而会被认为是规则表达式进行解析，从而无法正确匹配。

```
'/^new\/(\d{4})\/(\d{2})$/' => 'News/achive?year=:1&month=:2',
```

对于正则表达式中的每个变量（即正则规则中的子模式）部分，如果需要在后面的路由地址中引用，可以采用:1、:2这样的方式，序号就是子模式的序号。
更多的关于如何定义正则表达式就不在本文的描述范畴了。

规则表达式

规则路由比正则路由更方便定义和容易理解，规则表达式通常包含静态地址和动态地址，或者两种地址的结合，例如下面都属于有效的规则表达式：

```
'my'        =>'Member/myinfo', // 静态地址路由 类似于之前版本的简单路由
'blog/:id'    =>'Blog/read', // 静态地址和动态地址结合
'new/:year/:month/:day'=>'News/read', // 静态地址和动态地址结合
':user/:blog_id'=>'Blog/read',// 全动态地址
```

规则表达式的定义以“/”为参数分割符（无论你的URL_PATHINFO_DEPR设置是什么，请确保在定义规则表达式的时候统一使用“/”进行URL参数分割）。
每个参数中以“:”开头的参数都表示动态参数，并且会自动对应一个GET参数，例如 `:id` 表示该处匹配到的参数可以使用$_GET['id']方式获取， `:year :month :day` 则分别对应 `$_GET['year'] $_GET['month'] $_GET['day']` 。

数字约束

支持对变量的类型检测，但仅仅支持数字类型的约束定义，例如

```
'blog/:id\d'=>'Blog/read',
```

表示只会匹配数字参数，如果你需要更加多的变量类型检测，请使用正则表达式定义来解决。

规则排除
非数字变量支持简单的排除功能，主要是起到避免解析混淆的作用，例如：

```
'news/:cate^add-edit-delete'=>'News/category'
```

因为规则定义的局限性，恰巧我们的路由规则里面的news和实际的news模块是相同的命名，而:cate并不能自动区分当前URL里面的动态参数是实际的操作名还是路由变量，所以为了避免混淆，我们需要对路由变量cate进行一些排除以帮助我们进行更精确的路由匹配，格式^add|edit|delete表示，匹配除了add edit 和delete之外的所有字符串，我们建议更好的方式还是改进你的路由规则，避免路由规则和模块同名的情况存在，例如

```
'new/:cate'=>'News/category'
```

就可以更简单的定义路由规则了。

完全匹配
规则匹配检测的时候只是对URL从头开始匹配，只要URL地址包含了定义的路由规则就会匹配成功，如果希望完全匹配，可以使用$符号，例如：

```
'new/:cate$'=> 'News/category',
```

`http://serverName/index.php/new/info`

会匹配成功
而
`http://serverName/index.php/new/info/2`

则不会匹配成功
如果是采用

```
'new/:cate'=> 'News/category',
```

方式定义的话，则两种方式的URL访问都可以匹配成功。

路由地址和参数

路由地址和参数表示前面的路由表达式最终需要路由到的地址并且允许隐式传入URL里面没有的一些参数，这里允许使用字符串或者数组方式定义，支持下面6种方式定义：

定义方式	定义格式
方式1：路由到内部地址（字符串）	'[分组/模块/操作]?参数1=值1&参数2=值2...'
方式2：路由到内部地址（数组）参数采用字符串方式	array('[分组/模块/操作]','参数1=值1&参数2=值2...')
方式3：路由到内部地址（数组）参数采用数组方式	array('[分组/模块/操作]',array('参数1'=>'值1','参数2'=>'值2'...))
方式4：路由到外部地址（字符串）301重定向	'外部地址'
方式5：路由到外部地址（数组）可以指定重定向代码	array('外部地址','重定向代码')
方式6：闭包函数	function($name){ echo 'Hello,'.$name;}

如果路由地址以“/”或者“http”开头则会认为是一个重定向地址或者外部地址，例如：

```
'blog/:id'=>'/blog/read/id/:1'
```

和

```
'blog/:id'=>'blog/read/'
```

虽然都是路由到同一个地址，但是前者采用的是301重定向的方式路由跳转，这种方式的好处是URL可以比较随意（包括可以在URL里面传入更多的非标准格式的参数），而后者只是支持模块和操作地址。

举个例子，如果我们希望 avatar/123 重定向到 /member/avatar/id/123_small 的话，只能使用：

```
'avatar/:id'=>'/member/avatar/id/:1_small'
```

> 路由地址采用重定向地址的话，如果要引用动态变量，也是采用:1、:2 的方式。

采用重定向到外部地址通常对网站改版后的URL迁移过程非常有用，例如：

```
'blog/:id'=>'http://blog.thinkphp.cn/read/:1'
```

表示当前网站（可能是http://thinkphp.cn）的 blog/123 地址会直接重定向到 http://blog.thinkphp.cn/read/123。

在路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到），支持“ 参数1=值1&参数2=值2 ”或者

array('参数1'=>'值1','参数2'=>'值2'...) 这样的写法，可以参考不同的定义方式选择。例如：

```
'blog/:id'=>'blog/read/?status=1&app_id=5',
'blog/:id'=>array('blog/read/?status=1&app_id=5'),
'blog/:id'=>array('blog/read/','status=1&app_id=5'),
'blog/:id'=>array('blog/read/',array('status'=>1,'app_id'=>5)),
```

上面的路由规则定义中额外参数的传值方式都是等效的。 status 和 app_id 参数都是URL里面不存在的，属于隐式传值，当然并不一定需要用到，只是在需要的时候可以使用。

闭包支持

我们可以使用闭包的方式定义一些特殊需求的路由，而不需要执行控制器的操作方法了，例如：

```
'URL_ROUTE_RULES'=>array(
    'test'        => function(){ echo 'just test'; },
    'hello/:name' => function($name){
            echo 'Hello,'.$name;
        }
 )
```

参数传递

闭包定义的参数传递在规则路由和正则路由的两种情况下有所区别。

规则路由的参数传递比较简单：

```
'hello/:name' =>
    function($name){
        echo 'Hello,'.$name;
    }
```

规则路由中定义的动态变量的名称 就是闭包函数中的参数名称，不分次序。 因此，如果我们访问的URL地址是： http://serverName/Home/hello/thinkphp
则浏览器输出的结果是： Hello,thinkphp

如果多个参数可以使用：

```
'blog/:year/:month' =>
    function($year,$month){
        echo 'year='.$year.'&month='.$month;
    }
```

如果是正则路由的话，闭包函数中的参数就以正则中出现的参数次序来传递，例如：

```
'/^new\/(\d{4})\/(\d{2})$/' =>
    function($year,$month){
        echo 'year='.$year.'&month='.$month;
    }
```

如果我们访问： `http://serverName/Home/new/2013/03` 浏览器输出结果是：
`year=2013&month=03`

静态路由

静态路由其实属于规则路由的静态简化版（又称为URL映射），路由定义中不包含动态参数，静态路由不需要遍历路由规则而是直接定位，因此效率较高，但作用也有限。
如果我们定义了下面的静态路由

```
'URL_ROUTER_ON'   => true,
'URL_MAP_RULES'=>array(
    'new/top' => 'news/index?type=top'
)
```

> 注意：为了不影响动态路由的遍历效率，静态路由采用URL_MAP_RULES定义和动态路由区分开来

定义之后，如果我们访问： `http://serverName/Home/new/top`
其实是访问： `http://serverName/Home/news/index/type/top`
静态路由是完整匹配，所以如果访问： `http://serverName/Home/new/top/var/test`
尽管前面也有new/top，但并不会被匹配到news/index/type/top。
静态路由定义不受URL后缀影响，例如： `http://serverName/Home/new/top.html` 也可以正常访问。

> 静态路由的路由地址 只支持字符串，格式：[控制器/操作?]参数1=值1&参数2=值2

总结

通过本篇的学习，我们大概掌握了如何定义路由，后面我们将会学习如何定义视图和模板赋值。

快速入门 7：视图

快速入门（七）：视图

在了解了控制器和模型操作后，我们开始熟悉视图部分，ThinkPHP中的视图主要就是指模板文件和模板引擎，本篇首先了解下模板定义以及如何进行模板赋值并渲染输出的。

模板定义

每个模块的模板文件是独立的，为了对模板文件更加有效的管理，ThinkPHP对模板文件进行目录划分，默认的模板文件定义规则是：

```
视图目录/[模板主题/]控制器名/操作名+模板后缀
```

默认的视图目录是模块的View目录（模块可以有多个视图文件目录，这取决于你的应用需要），框架的默认视图文件后缀是.html。

> 大多数情况下你不需要主题功能，因此新版模板主题默认是空（表示不启用模板主题功能）。

一般情况下，模板文件都在模块的视图目录下面，并且是以模块下面的控制器名为目录，然后是每个控制器的具体操作模板文件，例如：
User控制器的add操作对应的模板文件就应该是：`./Application/Home/View/User/add.html`

如果你的默认视图层不是View，例如：

```
// 设置默认的视图层名称
'DEFAULT_V_LAYER'      =>  'Template',
```

那么，对应的模板文件就变成了：`./Application/Home/Template/User/add.html`。

模板文件的默认后缀的情况是.html，也可以通过 **TMPL_TEMPLATE_SUFFIX** 来配置成其他的。例如，我们可以配置：

```
'TMPL_TEMPLATE_SUFFIX'=>'.tpl'
```

定义后，User控制器的add操作 对应的模板文件就变成是： `./Application/Home/View/User/add.tpl`

如果觉得目录结构太深，可以通过设置 **TMPL_FILE_DEPR** 参数来配置简化模板的目录层次，例如设置：

```
'TMPL_FILE_DEPR'=>'_'
```

默认的模板文件就变成了：./Application/Home/View/User_add.html

如果需要，允许把模板目录设置到模块目录之外，有两种方式：

一、改变所有模块的模板文件目录

可以通过设置TMPL_PATH常量来改变所有模块的模板目录所在，例如：

```
define('TMPL_PATH','./Template/');
```

原来的 ./Application/Home/View/User/add.html 变成了 ./Template/Home/User/add.html 。

> 注意TMPL_PATH常量最后使用“/”符号结尾。

二、改变某个模块的模板文件目录

我们可以在模块配置文件中设置**VIEW_PATH**参数单独定义某个模块的视图目录，例如：

```
'VIEW_PATH'=>'./Theme/'
```

把当前模块的视图目录指定到最外层的Theme目录下面，而不是放到当前模块的View目录下面。 原来的 ./Application/Home/View/User/add.html 变成了 ./Theme/User/add.html 。

> 注意：如果同时定义了TMPL_PATH常量和VIEW_PATH设置参数，那么以当前模块的VIEW_PATH参数设置优先。

模板渲染

渲染模板输出最常用的是使用display方法，调用格式：

```
display('[模板文件]'[,'字符编码'][,'输出类型'])
```

模板文件的写法支持下面几种：

用法	描述
不带任何参数	自动定位当前操作的模板文件
[模块@][控制器:][操作]	常用写法，支持跨模块 模板主题可以和theme方法配合
完整的模板文件名	直接使用完整的模板文件名（包括模板后缀）

下面是一个最典型的用法，不带任何参数：

```
// 不带任何参数 自动定位当前操作的模板文件
$this->display();
```

表示系统会按照默认规则自动定位模板文件，其规则是：

- 如果当前没有启用模板主题则定位到：`当前模块/默认视图目录/当前控制器/当前操作.html` ；
- 如果有启用模板主题则定位到：`当前模块/默认视图目录/当前主题/当前控制器/当前操作.html` ；
- 如果有更改TMPL_FILE_DEPR设置（假设 'TMPL_FILE_DEPR'=>'_'）的话，则上面的自动定位规则变成：`当前模块/默认视图目录/当前控制器_当前操作.html` 和 `当前模块/默认视图目录/当前主题/当前控制器_当前操作.html`。

所以通常display方法无需带任何参数即可输出对应的模板，这是模板输出的最简单的用法。

通常默认的视图目录是View
如果没有按照模板定义规则来定义模板文件（或者需要调用其他控制器下面的某个模板），可以使用：

```
// 指定模板输出
// 表示调用当前控制器下面的edit模板
$this->display('edit');
```

或者指定控制器

```
// 表示调用Member控制器下面的read模板
$this->display('Member:read');
```

如果我们使用了模板主题功能，那么也可以支持跨主题调用，使用：

```
// 调用blue主题下面的User控制器的edit模板
$this->theme('blue')->display('User:edit');
```

渲染输出不需要写模板文件的路径和后缀，确切地说，这里面的控制器和操作并不一定需要有实际对应的控制器和操作，只是一个目录名称和文件名称而已，例如，你的项目里面可能根本没有Public控制器，更没有Public控制器的menu操作，但是一样可以使用

```
$this->display('Public:menu');
```

输出这个模板文件。

display方法支持在渲染输出的时候指定输出编码和类型，例如，可以指定编码和类型：

```
// 输出XML页面类型（配合你的应用需求可以输出很多类型）
$this->display('read', 'utf-8', 'text/xml');
```

事情总有特例，如果的模板目录是自定义的，或者根本不需要按模块进行分目录存放，那么默认的display渲染规则就不能处理，这个时候，我们就需要使用另外一种方式来应对，直接传入模板文件名即可，例如：

```
$this->display('./Template/Public/menu.html');
```

这种方式需要指定模板路径和后缀，这里的Template/Public目录是位于当前项目入口文件位置下面（当然如果需要也可以使用绝对路径）。

如果是其他的后缀文件，也支持直接输出，例如：

```
$this->display('./Template/Public/menu.tpl');
```

只要 ./Template/Public/menu.tpl 是一个实际存在的模板文件。

> 要注意模板文件位置是相对于项目的入口文件，而不是模板目录。

如果需要获取渲染模板的输出内容而不是直接输出，可以使用fetch方法。

fetch方法的用法除了不需要指定输出编码和类型外其它和display基本一致，格式：

```
fetch('模板文件')
```

模板文件的调用方法和display方法完全一样，区别就在于fetch方法渲染后不是直接输出，而是返回渲染后的内容，例如：

```
$content = $this->fetch('Member:edit');
```

使用fetch方法获取渲染内容后，你可以进行过滤和替换等操作，或者用于对输出的复杂需求。

渲染内容

如果你没有定义任何模板文件，或者把模板内容存储到数据库中的话，你就需要使用show方法来渲染输出了，show方法的调用格式：

```
show('渲染内容'[,'字符编码'][,'输出类型'])
```

例如，

```
$this->show($content);
// 也可以指定编码和类型
$this->show($content, 'utf-8', 'text/xml');
```

> 注意：show方法中的内容也可以支持模板解析。

模板赋值

如果要在模板中输出变量，必须在在控制器中把变量传递给模板，系统提供了assign方法对模板变量赋值，无论何种变量类型都统一使用assign赋值。

```
$this->assign('name',$value);
```

assign方法必须在display和show方法之前调用，并且系统只会输出设定的变量，其它变量不会输出（系统变量例外），一定程度上保证了变量的安全性。

> 系统变量可以通过特殊的标签输出，无需赋值模板变量

赋值后，就可以在模板文件中输出变量了，如果使用的是内置模板的话，就可以这样输出：

```
{$name}
```

如果要同时输出多个模板变量，可以使用下面的方式：

```
$array['name']    =    'thinkphp';
$array['email']   =    'liu21st@gmail.com';
$array['phone']   =    '12335678';
$this->assign($array);
```

这样，就可以在模板文件中同时输出name、email和phone三个变量。

模板变量的输出根据不同的模板引擎有不同的方法，我们在后面会专门讲解内置模板引擎的用法。如果你使用的是PHP本身作为模板引擎的话 ，就可以直接在模板文件里面输出了：

```
<?php echo $name.'['.$email.''.$phone.']';?>
```

如果采用内置的模板引擎，可以使用：

```
{$name} [ {$email} {$phone} ]
```

输出同样的内容。

总结

通过本篇的学习，我们大概掌握了如何定义模板文件和进行模板渲染输出，以及如何赋值模板变量，后面我们将会学习如何在模板文件中使用标签来简化你的书写。

快速入门 8：变量输出

快速入门（八）：变量输出

在上一章我们了解了如何通过assign方法把变量赋值到模板变量，这一篇我们来详细了解下如何在模板中使用标签输出模板变量。

注意，本篇的描述仅针对使用内部模板引擎的情况，如果你使用了Smarty或者其他模板引擎，请参考其相关的变量输出语法。

变量输出

变量输出（**这里主要是指标量类型的输出**）的方法很简单，例如，在控制器中我们给模板变量赋值：

```
$name = 'ThinkPHP';
$this->assign('name',$name);
$this->display();
```

然后就可以在模板中使用：

```
Hello,{$name}！
```

模板编译后的结果就是：

```
Hello,<?php echo($name);?>！
```

这样，运行的时候就会在模板中显示：

```
Hello,ThinkPHP！
```

注意模板标签的 { 和 $ 之间不能有任何的空格，否则标签无效。所以，下面的标签

```
Hello,{ $name}！
```

将不会正常输出name变量，而是直接保持不变输出：

```
Hello,{ $name}！
```

普通标签默认开始标记是 { ，结束标记是 } 。也可以通过设置 TMPL_L_DELIM 和 TMPL_R_DELIM 进行更改。例如，我们在项目配置文件中定义：

```
'TMPL_L_DELIM'=>'<{',
'TMPL_R_DELIM'=>'}>',
```

那么，上面的变量输出标签就应该改成：

```
Hello,<{$name}> !
```

后面的内容我们都以默认的标签定义来说明。
模板标签的变量输出根据变量类型有所区别，刚才我们输出的是字符串变量，如果是数组变量，

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'thinkphp@qq.com';
$this->assign('data',$data);
```

那么，在模板中我们可以用下面的方式输出：

```
Name：{$data.name}
Email：{$data.email}
```

或者用下面的方式也是有效：

```
Name：{$data['name']}
Email：{$data['email']}
```

当我们要输出多维数组的时候，往往要采用后面一种方式。
如果data变量是一个对象（并且包含有name和email两个属性），那么可以用下面的方式输出：

```
Name：{$data:name}
Email：{$data:email}
```

或者

```
Name：{$data->name}
Email：{$data->email}
```

系统变量

普通的模板变量需要首先赋值后才能在模板中输出，但是系统变量则不需要，可以直接在模板中输出，系统变量的输出通常以`{$Think`打头，例如：

```
{$Think.server.script_name} // 输出$_SERVER['SCRIPT_NAME']变量
{$Think.session.user_id} // 输出$_SESSION['user_id']变量
{$Think.get.pageNumber} // 输出$_GET['pageNumber']变量
{$Think.cookie.name}  // 输出$_COOKIE['name']变量
```

支持输出 $_SERVER、$_ENV、 $_POST、 $_GET、 $_REQUEST、$_SESSION和 $_COOKIE 变量。

还可以输出常量

```
{$Think.const.MODULE_NAME}
```

或者直接使用

```
{$Think.MODULE_NAME}
```

输出配置参数使用：

```
{$Think.config.db_charset}
{$Think.config.url_model}
```

输出语言变量可以使用：

```
{$Think.lang.page_error}
{$Think.lang.var_error}
```

使用函数

我们往往需要对模板输出变量使用函数，可以使用：

```
{$data.name|md5}
```

编译后的结果是：

```
<?php echo (md5($data['name'])); ?>
```

如果函数有多个参数需要调用，则使用：

```
{$create_time|date="y-m-d",###}
```

表示date函数传入两个参数，每个参数用逗号分割，这里第一个参数是y-m-d，第二个参数是前面要输出的create_time变量，因为该变量是第二个参数，因此需要用 ### 标识变量位置，编译后的结果是：

```
<?php echo (date("y-m-d",$create_time)); ?>
```

如果前面输出的变量在后面定义的函数的第一个参数，则可以直接使用：

```
{$data.name|substr=0,3}
```

表示输出

```
<?php echo (substr($data['name'],0,3)); ?>
```

虽然也可以使用：

```
{$data.name|substr=###,0,3}
```

但完全没用这个必要。

还可以支持多个函数过滤，多个函数之间用“|”分割即可，例如：

```
{$name|md5|strtoupper|substr=0,3}
```

编译后的结果是：

```
<?php echo (substr(strtoupper(md5($name)),0,3)); ?>
```

函数会按照从左到右的顺序依次调用。
如果你觉得这样写起来比较麻烦，也可以直接这样写：

```
{:substr(strtoupper(md5($name)),0,3)}
```

默认值

我们可以给变量输出提供默认值，例如：

```
{$user.nickname|default="这家伙很懒，什么也没留下"}
```

对系统变量依然可以支持默认值输出，例如：

```
{$Think.get.name|default="名称为空"}
```

默认值和函数可以同时使用，例如：

```
{$Think.get.name|getName|default="名称为空"}
```

使用运算符

我们可以对模板输出使用运算符，包括对“+” “–” “*” “/”和“%”的支持。

例如：

运算符	使用示例
+	{$a+$b}
-	{$a-$b}
*	{$a*$b}
/	{$a/$b}
%	{$a%$b}
++	{$a++} 或 {++$a}
--	{$a--} 或 {--$a}
综合运算	{$a+$b*10+$c}

在使用运算符的时候，不再支持点语法和常规的函数用法，例如：

```
{$user.score+10} //错误的
{$user['score']+10} //正确的
{$user['score']*$user['level']} //正确的
{$user['score']|myFun*10} //错误的
{$user['score']+myFun($user['level'])} //正确的
```

三元运算

模板可以支持三元运算符，例如：

```
{$status?'正常':'错误'}
{$info['status']?$info['msg']:$info['error']}
```

三元运算符中暂时不支持点语法，因此下面的写法是错误的：

```
{$info.status?$info.msg:$info.error}
```

总结

通过本篇的学习，我们掌握了如何在模板文件中输出变量和使用函数、默认值和运算符，下一篇我们将会了解如何进行模板变量的循环、判断等控制输出，以及导入其他公共模板。

快速入门 9：循环和控制输出

快速入门（九）：循环和控制输出

在上一章我们了解了如何把变量输出到模板中，这一篇我们来进一步了解如何进行模板数据的循环和控制输出。

注意，本篇的描述仅针对使用内部模板引擎的情况，如果你使用了Smarty或者其他模板引擎，请参考其相关的变量输出语法。

循环输出

循环输出主要是使用volist和foreach标签输出。

VOLIST

volist标签通常用于查询数据集（select方法）的结果输出，通常模型的select方法返回的结果是一个二维数组，可以直接使用volist标签进行输出。

在控制器中首先对模版赋值：

```
$User = M('User');
$list = $User->limit(10)->select();
$this->assign('list',$list);
```

在模版定义如下，循环输出用户的编号和姓名：

```
<volist name="list" id="vo">
{$vo.id}:{$vo.name}<br/>
</volist>
```

Volist标签的 name属性 表示模板赋值的变量名称，因此不可随意在模板文件中改变。 id 表示当前的循环变量，可以随意指定，但确保不要和name属性冲突，例如：

```
<volist name="list" id="data">
{$data.id}:{$data.name}<br/>
</volist>
```

支持输出查询结果中的部分数据，例如输出其中的第5～15条记录

```
<volist name="list" id="vo" offset="5" length='10'>
{$vo.name}
</volist>
```

输出偶数记录

```
<volist name="list" id="vo" mod="2" >
<eq name="mod" value="1">{$vo.name}</eq>
</volist>
```

Mod属性还用于控制一定记录的换行，例如：

```
<volist name="list" id="vo" mod="5" >
{$vo.name}
<eq name="mod" value="4"><br/></eq>
</volist>
```

为空的时候输出提示：

```
<volist name="list" id="vo" empty="暂时没有数据" >
{$vo.id}|{$vo.name}
</volist>
```

empty属性不支持直接传入html语法，但可以支持变量输出，例如：

```
$this->assign('empty','<span class="empty">没有数据</span>');
$this->assign('list',$list);
```

然后在模板中使用：

```
<volist name="list" id="vo" empty="$empty" >
{$vo.id}|{$vo.name}
</volist>
```

模板中可以直接使用函数设定数据集，而不需要在控制器中给模板变量赋值传入数据集变量，如：

```
<volist name=":fun('arg')" id="vo">
{$vo.name}
</volist>
```

FOREACH

除了volist标签之外，还可以使用foreach标签，foreach标签类似与volist标签，只是更加简单，没有太多额外的属性，例如：

```
<foreach name="list" item="vo">
   {$vo.id}:{$vo.name}
</foreach>
```

name表示数据源 item表示循环变量。

foreach标签还可以输出一维数组，例如：

```
<foreach name="list" item="vo" >
   {$key}|{$vo}
</foreach>
```

条件输出

条件输出最常用的包括switch、if以及eq等比较标签。

SWITCH

用法：

```
<switch name="变量" >
<case value="值1" break="0或1">输出内容1</case>
<case value="值2">输出内容2</case>
<default />默认情况
</switch>
```

使用方法如下：

```
<switch name="User.level">
   <case value="1">value1</case>
   <case value="2">value2</case>
   <default />default
</switch>
```

其中name属性可以使用函数以及系统变量，例如：

```
<switch name="Think.get.userId|abs">
   <case value="1">admin</case>
   <default />default
</switch>
```

对于case的value属性可以支持多个条件的判断，使用”|”进行分割，例如：

```
<switch name="Think.get.type">
    <case value="gif|png|jpg">图像格式</case>
    <default />其他格式
</switch>
```

表示如果$_GET["type"] 是gif、png或者jpg的话，就判断为图像格式。

Case标签还有一个break属性，表示是否需要break，默认是会自动添加break，如果不要break，可以使用：

```
<switch name="Think.get.userId|abs">
    <case value="1" break="0">admin</case>
    <case value="2">admin</case>
    <default />default
</switch>
```

也可以对case的value属性使用变量，例如：

```
<switch name="User.userId">
    <case value="$adminId">admin</case>
    <case value="$memberId">member</case>
    <default />default
</switch>
```

比较标签

比较标签用于简单的变量比较，复杂的判断条件可以用if标签替换，比较标签是一组标签的集合，基本上用法都一致，如下：

```
<比较标签 name="变量" value="值">
内容
</比较标签>
```

系统支持的比较标签以及所表示的含义分别是：

标签	含义
eq或者 equal	等于
neq 或者notequal	不等于
gt	大于
egt	大于等于
lt	小于
elt	小于等于

标签	含义
heq	恒等于
nheq	不恒等于

他们的用法基本是一致的，区别在于判断的条件不同，并且所有的比较标签都可以和else标签一起使用。

例如，要求name变量的值等于value就输出，可以使用：

```
<eq name="name" value="value">value</eq>
```

或者

```
<equal name="name" value="value">value</equal>
```

也可以支持和else标签混合使用：

```
<eq name="name" value="value">
相等
<else/>
不相等
</eq>
```

当 name变量的值大于5就输出

```
<gt name="name" value="5">value</gt>
```

当name变量的值不小于5就输出

```
<egt name="name" value="5">value</egt>
```

比较标签中的变量可以支持对象的属性或者数组，甚至可以是系统变量，例如：
当vo对象的属性（或者数组，或者自动判断）等于5就输出

```
<eq name="vo.name" value="5">
{$vo.name}
</eq>
```

当vo对象的属性等于5就输出

```
<eq name="vo:name" value="5">
{$vo.name}
</eq>
```

当$vo['name']等于5就输出

```
<eq name="vo['name']" value="5">
{$vo.name}
</eq>
```

而且还可以支持对变量使用函数
当vo对象的属性值的字符串长度等于5就输出

```
<eq name="vo:name|strlen" value="5">{$vo.name}</eq>
```

变量名可以支持系统变量的方式，例如：

```
<eq name="Think.get.name" value="value">相等<else/>不相等</eq>
```

通常比较标签的值是一个字符串或者数字，如果需要使用变量，只需要在前面添加“$”标志：
当vo对象的属性等于$a就输出

```
<eq name="vo:name" value="$a">{$vo.name}</eq>
```

范围判断标签

范围判断标签包括 in 、 notin 、 between 和 notbetween 四个标签，都用于判断变量是否中某个范围。

IN和NOTIN

用法：
假设我们中控制器中给id赋值为1：

```
$id =    1;
$this->assign('id',$id);
```

我们可以使用in标签来判断模板变量是否在某个范围内，例如：

```
<in name="id" value="1,2,3">
id在范围内
</in>
```

最后会输出： id在范围内 。

如果判断不在某个范围内，可以使用：

```
<notin name="id" value="1,2,3">
id不在范围内
</notin>
```

可以把上面两个标签合并成为：

```
<in name="id" value="1,2,3">
id在范围内
<else/>
id不在范围内
</in>
```

name属性还可以支持直接判断系统变量，例如：

```
<in name="Think.get.id" value="1,2,3">
$_GET['id'] 在范围内
</in>
```

value属性也可以使用变量，例如：

```
<in name="id" value="$range">
id在范围内
</in>
```

$range变量可以是数组，也可以是以逗号分隔的字符串。

BETWEEN 和 NOTBETWEEN

可以使用between标签来判断变量是否在某个区间范围内，可以使用：

```
<between name="id" value="1,10">
输出内容1
</between>
```

同样，可以使用notbetween标签来判断变量不在某个范围内：

```
<notbetween name="id" value="1,10">
输出内容2
</notbetween>
```

也可以使用else标签把两个用法合并，例如：

```
<between name="id" value="1,10">
输出内容1
<else/>
输出内容2
</between>
```

当使用between标签的时候，value只需要一个区间范围，也就是只支持两个值，后面的值无效，例如

```
<between name="id" value="1,3,10">
输出内容1
</between>
```

实际判断的范围区间是 `1~3` ，而不是 `1~10` ，也可以支持字符串判断，例如：

```
<between name="id" value="A,Z">
输出内容1
</between>
```

name属性可以直接使用系统变量，例如：

```
<between name="Think.post.id" value="1,5">
输出内容1
</between>
```

value属性也可以使用变量，例如：

```
<between name="id" value="$range">
输出内容1
</between>
```

赋值判断标签

可以使用present、empty等标签进行赋值判断输出。

present标签用于判断某个变量是否已经定义，用法：

```
<present name="name">
name已经赋值
<else />
name还没有赋值
</present>
```

name属性可以直接使用系统变量，例如：

```
<present name="Think.get.name">
$_GET['name']已经赋值
</present>
```

empty标签用于判断某个变量是否为空，用法：

name为空

name不为空

name属性可以直接使用系统变量，例如：

```
<empty name="Think.get.name">
$_GET['name']为空值
</empty>
```

DEFINED标签用于判断某个常量是否有定义，用法如下：

```
<defined name="NAME">
NAME常量已经定义
<else />
NAME常量未定义
</defined>
```

原生代码

Php代码可以和标签在模板文件中混合使用，可以在模板文件里面书写任意的PHP语句代码 ，包括下面两种方式：

第一种：使用php标签

例如：

```
<php>echo 'Hello,world!';</php>
```

我们建议需要使用PHP代码的时候尽量采用php标签，因为原生的PHP语法可能会被配置禁用而导致解析错误。

第二种：使用原生php代码

```
<?php echo 'Hello,world!'; ?>
```

总结

本章我们讲述了模板中进行循环和控制输出，下一篇我们还会讲述如何使用公共模板和布局模板来简化模板文件的定义。

快速入门 10：公共模板和模板布局

快速入门（十）：公共模板和模板布局

我们学习了模板的输出后，就会发现很多应用存在大量的模板文件，如何简化模板文件的定义和公共调用就成了关键，ThinkPHP的模板引擎内置了公共模板和布局模板功能支持，可以方便的规划和公用你的模板文件。

公共模板

在当前模版文件中包含其他公用的模版文件使用include标签，标签用法：

```
<include file='模版表达式或者模版文件1,模版表达式或者模版文件2,...' />
```

使用模版表达式

模版表达式的定义规则为：

```
模块@主题/控制器/操作
```

例如：

```
<include file="Public/header" /> // 包含头部模版header
<include file="Public/menu" /> // 包含菜单模版menu
<include file="Blue/Public/menu" /> // 包含blue主题下面的menu模版
```

可以一次包含多个模版，例如：

```
<include file="Public/header,Public/menu" />
```

> 注意，包含模版文件并不会自动调用控制器的方法，也就是说包含的其他模版文件中的变量赋值需要在当前操作中完成。

使用模版文件

可以直接包含一个模版文件名（包含完整路径），例如：

```
<include file="./Application/Home/View/default/Public/header.html" />
```

传入参数

无论你使用什么方式包含外部模板，Include标签支持在包含文件的同时传入参数，例如，下面的例子我们在包含header模板的时候传入了title和keywords变量：

```
<include file="Public/header" title="ThinkPHP框架" keywords="开源WEB开发框架" />
```

就可以在包含的header.html文件里面使用title和keywords变量，如下：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>[title]</title>
<meta name="keywords" content="[keywords]" />
</head>
```

注意：如果外部模板有所更改，模板引擎并不会重新编译模板，除非在调试模式下或者缓存已经过期。如果部署模式下修改了包含的外部模板文件后，需要把模块的缓存目录清空，否则无法生效。

模板布局

有三种布局模板的支持方式：

第一种方式：全局配置方式

这种方式仅需在项目配置文件中添加相关的布局模板配置，就可以简单实现模板布局功能，比较适用于全站使用相同布局的情况，需要配置开启 LAYOUT_ON 参数（默认不开启），并且设置布局入口文件名 LAYOUT_NAME （默认为layout）。

```
'LAYOUT_ON'=>true,
'LAYOUT_NAME'=>'layout',
```

开启LAYOUT_ON后，我们的模板渲染流程就有所变化，例如：

```
<?php
namespace Home\Controller;
use Think\Controller;
Class UserController extends Controller {
    Public function add() {
        $this->display('add');
    }
}
```

在不开启 LAYOUT_ON 布局模板之前，会直接渲染 Home/View/User/add.html 模板文件,开启之后首先会渲染 Home/View/layout.html 模板，布局模板的写法和其他模板的写法类似，本身也可以支持所有

的模板标签以及包含文件，区别在于有一个特定的输出替换变量{__CONTENT__}，例如，下面是一个典型的layout.html模板的写法：

```
<include file="Public:header" />
{__CONTENT__}
<include file="Public:footer" />
```

读取layout模板之后，会再解析Home/View/User/add.html模板文件，并把解析后的内容替换到layout布局模板文件的{__CONTENT__}特定字符串。

采用这种布局方式的情况下，一旦Home/View/User/add.html模板文件或者Home/View/layout.html布局模板文件发生修改，都会导致模板重新编译。
如果需要指定其他位置的布局模板，可以使用：

```
'LAYOUT_NAME'=>'Layout/layoutname',
```

就表示采用Home/View/Layout/layoutname.html作为布局模板。

如果某些页面不需要使用布局模板功能，可以在模板文件开头加上{__NOLAYOUT__}字符串。
如果上面的Home/View/User/add.html模板文件里面包含有{__NOLAYOUT__}，则即使当前开启布局模板，也不会进行布局模板解析。

第二种方式：模板标签方式

这种布局模板不需要在配置文件中设置任何参数，也不需要开启**LAYOUT_ON**，直接在模板文件中指定布局模板即可，相关的布局模板调整也在模板中进行。

以前面的输出模板为例，这种方式的入口还是在Home/View/User/add.html模板，但是我们可以修改下add模板文件的内容，在头部增加下面的布局标签（记得首先关闭前面的LAYOUT_ON设置，否则可能出现布局循环）：

```
<layout name="layout" />
```

表示当前模板文件需要使用Home/View/layout.html布局模板文件，而布局模板文件的写法和上面第一种方式是一样的。当渲染Home/View/User/add.html模板文件的时候，如果读取到layout标签，则会把当前模板的解析内容替换到layout布局模板的{__CONTENT__}特定字符串。

一个模板文件中只能使用一个布局模板，如果模板文件中没有使用任何layout标签则表示当前模板不使用任何布局。
如果需要使用其他的布局模板，可以改变layout的name属性，例如：

```
<layout name="newlayout" />
```

还可以在layout标签里面指定要替换的特定字符串：

```
<layout name="Layout/newlayout"  replace="{__REPLACE__}" />
```

由于所有include标签引入的文件都支持layout标签，所以，我们可以借助layout标签和include标签相结合的方式实现布局模板的嵌套。例如，上面的例子

```
<include file="Public:header" />
<div id="main" class="main" >
{__CONTENT__}
</div>
<include file="Public:footer" />
```

在引入的header和footer模板文件中也可以添加layout标签，例如header模板文件的开头添加如下标签：

```
<layout name="menu" />
```

这样就实现了在头部模板中引用了menu布局模板。
也可以采用两种布局方式的结合，可以实现更加复杂的模板布局以及嵌套功能。

第三种方式：使用layout控制模板布局

使用内置的layout方法可以更灵活的在程序中控制模板输出的布局功能，尤其适用于局部需要布局或者关闭布局的情况，这种方式也不需要在配置文件中开启LAYOUT_ON。例如：

```
<?php
namespace Home\Controller;
use Think\Controller;
Class UserController extends Controller {
    Public function add() {
        layout(true);
        $this->display('add');
    }
}
```

表示当前的模板输出启用了布局模板，并且采用默认的layout布局模板。

如果当前输出需要使用不同的布局模板，可以动态的指定布局模板名称，例如：

```
<?php
namespace Home\Controller;
use Think\Controller;
Class UserController extends Controller {
    Public function add() {
        layout('Layout/newlayout');
        $this->display('add');
    }
}
```

或者使用layout方法动态关闭当前模板的布局功能（这种用法可以配合第一种布局方式，例如全局配置已经开启了布局，可以在某个页面单独关闭）：

```
<?php
namespace Home\Controller;
use Think\Controller;
Class UserController extends Controller {
    Public function add() {
        layout(false); // 临时关闭当前模板的布局功能
        $this->display('add');
    }
}
```

三种模板布局方式中，第一种和第三种是在程序中配置实现模板布局，第二种方式则是单纯通过模板标签在模板中使用布局。具体选择什么方式，需要根据项目的实际情况来了。

总结

本章主要讲述了如何在模板中调用外部公共模板和使用模板布局简化模板定义，下一篇开始我们会讲述控制器的一些高级特性。

快速入门 11：Action参数绑定

快速入门（十一）：Action参数绑定

从本章开始，我们陆续为大家介绍下控制器的一些高级特性用法，让你更深入了解ThinkPHP控制器的独特功能。

Action参数绑定

在之前的内容中，涉及的控制器操作方法都是没有任何参数的，其实ThinkPHP可以支持操作方法的参数绑定功能。

Action参数绑定是通过直接绑定URL地址中的变量作为操作方法的参数，可以简化方法的定义甚至路由的解析,其原理是把URL中的参数（不包括模块、控制器和操作名）和控制器的操作方法中的参数（按变量名或者变量顺序）进行绑定。

按变量名绑定

默认的参数绑定方式是按照变量名进行绑定，例如，我们给Blog控制器定义了两个操作方法read和archive方法，由于read操作需要指定一个id参数，archive方法需要指定年份（year）和月份（month）两个参数，那么我们可以如下定义：

```
<?php
namespace Home\Controller;
use Think\Controller;
class BlogController extends Controller{
    public function read($id){
        echo 'id='.$id;
    }

    public function archive($year='2013',$month='01'){
        echo 'year='.$year.'&month='.$month;
    }
}
```

> 注意这里的操作方法并没有具体的业务逻辑，只是简单的示范。

URL的访问地址分别是：

```
http://serverName/index.php/Home/Blog/read/id/5
http://serverName/index.php/Home/Blog/archive/year/2013/month/11
```

两个URL地址中的id参数和year和month参数会自动和read操作方法以及archive操作方法的同名参数绑

定。

> 变量名绑定不一定由访问URL决定，路由地址也能起到相同的作用

输出的结果依次是：

```
id=5
year=2013&month=11
```

按照变量名进行参数绑定的参数必须和URL中传入的变量名称一致，但是参数顺序不需要一致。也就是说

```
http://serverName/index.php/Home/Blog/archive/month/11/year/2013
```

和上面的访问结果是一致的，URL中的参数顺序和操作方法中的参数顺序都可以随意调整，关键是确保参数名称一致即可。

如果使用下面的URL地址进行访问，参数绑定仍然有效：

```
http://serverName/index.php?s=/Home/Blog/read/id/5
http://serverName/index.php?s=/Home/Blog/archive/year/2013/month/11
http://serverName/index.php?c=Blog&a=read&id=5
http://serverName/index.php?c=Blog&a=archive&year=2013&month=11
```

如果用户访问的URL地址是（至于为什么会这么访问暂且不提）：

```
http://serverName/index.php/Home/Blog/read/
```

那么会抛出下面的异常提示： `参数错误:id`

报错的原因很简单，因为在执行read操作方法的时候，id参数是必须传入参数的，但是方法无法从URL地址中获取正确的id参数信息。由于我们不能相信用户的任何输入，因此建议你给read方法的id参数添加默认值，例如：

```
    public function read($id=0){
        echo 'id='.$id;
    }
```

这样，当我们访问 `http://serverName/index.php/Home/Blog/read/` 的时候 就会输出

```
id=0
```

当我们访问 `http://serverName/index.php/Home/Blog/archive/` 的时候，输出：

```
year=2013&month=01
```

> 始终给操作方法的参数定义默认值是一个避免报错的好办法

按变量顺序绑定

第二种方式是按照变量的顺序绑定，这种情况下URL地址中的参数顺序非常重要，不能随意调整。要按照变量顺序进行绑定，必须先设置 URL_PARAMS_BIND_TYPE 为1：

```
'URL_PARAMS_BIND_TYPE'  =>  1, // 设置参数绑定按照变量顺序绑定
```

操作方法的定义不需要改变，URL的访问地址分别改成：

```
http://serverName/index.php/Home/Blog/read/5
http://serverName/index.php/Home/Blog/archive/2013/11
```

输出的结果依次是：

```
id=5
year=2013&month=11
```

这个时候如果改成

```
http://serverName/index.php/Home/Blog/archive/11/2013
```

输出的结果就变成了：

```
year=11&month=2013
```

显然就有问题了，所以不能随意调整参数在URL中的传递顺序，要确保和你的操作方法定义顺序一致。

> 可以看到，这种参数绑定的效果有点类似于简单的规则路由。

按变量顺序绑定的方式目前仅对PATHINFO地址有效，所以下面的URL访问参数绑定会失效：

```
http://serverName/index.php?c=Blog&a=read&id=5
http://serverName/index.php?c=Blog&a=archive&year=2013&month=11
```

但是，兼容模式URL地址访问依然有效：

```
http://serverName/index.php?s=/Home/Blog/read/5
http://serverName/index.php?s=/Home/Blog/archive/2013/11
```

如果你的操作方法定义都不带任何参数或者不希望使用该功能的话，可以关闭参数绑定功能：

```
'URL_PARAMS_BIND'       =>  false
```

快速入门 12：空操作和空控制器

快速入门（十二）：空操作和空控制器

本章的内容是讲解下ThinkPHP控制器的两个实用特性：空操作和空控制器，通常可以用来实现一些错误页面和URL优化。

空操作

空操作是指系统在找不到请求的操作方法的时候，会定位到当前控制器的空操作（ _empty ）方法来执行。

例如，下面我们用空操作功能来实现一个城市切换的功能。 我们只需要给CityController类定义一个 _empty 方法：

```
<?php
namespace Home\Controller;
use Think\Controller;
class CityController extends Controller{
    public function _empty($name){
        //把所有城市的操作解析到city方法
        $this->city($name);
    }
    //注意 city方法 本身是 protected 方法
    protected function city($name){
        //和$name这个城市相关的处理
         echo '当前城市' . $name;
    }
}
```

接下来，我们就可以在浏览器里面输入

```
http://serverName/index.php/Home/City/beijing/
http://serverName/index.php/Home/City/shanghai/
http://serverName/index.php/Home/City/shenzhen/
```

由于City控制器并没有定义beijing、shanghai或者shenzhen操作方法，因此系统会定位到空操作方法 _empty 中去解析， _empty 方法的参数就是当前URL里面的操作名，因此会看到依次输出的结果是：

```
当前城市:beijing
当前城市:shanghai
当前城市:shenzhen
```

> 注意：空操作方法仅在你的控制器类继承系统的 Think\Controller 类才有效。

空控制器

空控制器的概念是指当系统找不到请求的控制器名称的时候，系统会尝试定位空控制器(EmptyController)。

现在我们把前面的需求进一步，把URL由原来的

http://serverName/index.php/Home/City/shanghai/
变成

http://serverName/index.php/Home/shanghai/

这样更加简单的方式，如果按照传统的模式，我们必须给每个城市定义一个控制器类，然后在每个控制器类的index方法里面进行处理。可是如果使用空控制器功能，这个问题就可以迎刃而解了。

我们可以给项目定义一个EmptyController类

```
<?php
namespace Home\Controller;
use Think\Controller;
class EmptyController extends Controller{
    public function index(){
        //根据当前控制器名来判断要执行那个城市的操作
        $cityName = CONTROLLER_NAME;
        $this->city($cityName);
    }
    //注意 city方法 本身是 protected 方法
    protected function city($name){
        //和$name这个城市相关的处理
         echo '当前城市' . $name;
    }
}
```

接下来，我们就可以在浏览器里面输入

```
http://serverName/index.php/Home/beijing/
http://serverName/index.php/Home/shanghai/
http://serverName/index.php/Home/shenzhen/
```

由于系统并不存在beijing、shanghai或者shenzhen控制器，因此会定位到空控制器（ EmptyController ）去执行，会看到依次输出的结果是：

```
当前城市:beijing
当前城市:shanghai
当前城市:shenzhen
```

空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

快速入门 13：初始化、前置和后置操作

快速入门（十三）：初始化、前置和后置操作

本章的内容讲解了如何在ThinkPHP控制器的操作方法调用之前或者之后做一些额外的操作，涉及到的知识点包括初始化操作、前置和后置操作。

初始化操作

如果要在控制器的任何操作方法之前都执行某个方法的话，可以使用下面的方式：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    // 初始化方法
    public function _initialize(){
        echo 'initialize<br/>';
    }
    public function index(){
        echo 'index';
    }

    public function hello(){
        echo 'hello';
    }
}
```

如果我们访问 `http://serverName/index.php/Home/index/index`

结果会输出

```
initialize
index
```

如果我们访问 `http://serverName/index.php/Home/index/hello`

结果会输出

```
initialize
hello
```

可以看出，无论是执行index操作还是hello操作 都会首先执行 `_initialize` 操作方法。

如果把 `_initialize` 操作方法定义到一个公共的控制器类里面的话，那么所有的控制器操作方法之前都会执

行。

前置和后置操作

_initialize 方法是调用所有操作方法之前都会执行，前置和后置操作则是针对某个特定的操作方法而言。

如果当前访问的操作是存在（必须是实际在控制器中定义过）的，系统会检测当前操作是否具有前置和后置操作，如果存在就会按照顺序执行，前置和后置操作的方法名是在要执行的方法前面加 _before_ 和 _after_ ，例如：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    //前置操作方法
    public function _before_index(){
        echo 'before<br/>';
    }
    public function index(){
        echo 'index<br/>';
    }
    //后置操作方法
    public function _after_index(){
        echo 'after';
    }
}
```

如果我们访问 http://serverName/index.php

结果会输出

```
before
index
after
```

对于任何操作方法我们都可以按照这样的规则来定义前置和后置方法。

如果在操作方法里面使用了exit或者error方法的话 有可能不会再执行后置方法了，例如：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    //前置操作方法
    public function _before_index(){
        echo 'before<br/>';
    }
    public function index(){
        echo 'index<br/>';
        exit;
    }
    //后置操作方法
    public function _after_index(){
        echo 'after';
    }
}
```

如果我们再次访问结果会输出

```
before
index
```

> 除了初始化、前置和后置操作之外，我们还可以在控制器以外的地方对操作方法进行扩展，这个以后会在行为扩展部分描述。

快速入门 14：页面跳转和重定向

快速入门（十四）：页面跳转和重定向

本章内容讲解了如何在ThinkPHP中使用页面跳转和重定向功能。

页面跳转

系统的 Think\Controller 类内置了两个页面跳转方法error和success，分别用于错误（提示）跳转和成功（提示）跳转。两个方法都会输出一个提示信息页面，然后自动跳转到指定的地址。下面是一个简单的例子：

```
$New = M('New'); //实例化New对象
$result = $New->add($data);
if($result){
    // 成功后跳转到新闻列表页面
    $this->success('新增成功，即将返回列表页面', '/New/index');
} else {
    // 错误页面的默认跳转页面是返回前一页，通常不需要设置
    $this->error('新增失败');
}
```

success和error方法有三个参数，分别是提示信息、跳转地址和跳转页面等待时间（秒），除了第一个参数外其他都是可选的。

提示信息：成功或者错误信息字符串。

跳转地址：页面跳转地址是可选的，success方法的默认跳转地址是 $_SERVER["HTTP_REFERER"] ，error方法的默认跳转地址是 javascript:history.back(-1); 。

等待时间：默认的等待时间success方法是1秒，error方法是3秒。

success和error方法都可以对应的模板，默认两个方法对应的模板是框架自带的跳转模板 dispatch_jump.tpl：

```
//默认错误跳转对应的模板文件
'TMPL_ACTION_ERROR' => THINK_PATH . 'Tpl/dispatch_jump.tpl',
//默认成功跳转对应的模板文件
'TMPL_ACTION_SUCCESS' => THINK_PATH . 'Tpl/dispatch_jump.tpl',
```

success方法默认页面显示如下：

新增成功，即将返回列表页面

页面自动 跳转 等待时间： 1

error方法默认页面显示如下：

新增失败

页面自动 跳转 等待时间： 3

你可以重新定义跳转模板，通常建议直接放到项目目录下面（下面采用公共模块的模板作为项目统一的跳转模板）：

```
//默认错误跳转对应的模板文件
'TMPL_ACTION_ERROR' => 'Common@Public/error',
//默认成功跳转对应的模板文件
'TMPL_ACTION_SUCCESS' => 'Common@Public/success',
```

模板文件可以使用模板标签，并且可以使用下面的模板变量：

变量	含义
$message	页面成功提示信息
$error	页面错误提示信息
$waitSecond	跳转等待时间 单位为秒
$jumpUrl	跳转页面地址

重定向

如果不需要提示页面，ThinkPHP还可以实现直接重定向操作，`Think\Controller` 类提供了redirect方法

实现页面的重定向功能。

重定向到操作

```
redirect('重定向操作地址（一般为[控制器/操作]）','参数（字符串或者数组）','重定向等待时间（秒）','重定向提示信息')
```

例如：

```
$New = M('New'); //实例化New对象
$result = $New->add($data);
if($result){
    // 停留5秒后跳转到New模块的category操作，并且显示页面跳转中字样
    $this->redirect('New/category', 'cate_id=2&status=1', 5,'页面跳转中...');
} else {
    // 错误页面
    $this->redirect('New/error');
}
```

可以传入参数和设置重定向的等待时间，甚至给出等待的提示信息：

> 注意：重定向后会改变当前的URL地址。

重定向到URL

如果你仅仅是想重定向要一个指定的URL地址，而不是到控制器的操作方法，可以直接使用redirect函数重定向，例如：

```
$New = M('New'); //实例化New对象
$result = $New->add($data);
if($result){
    //重定向到指定的URL地址
    redirect('/New/category/cate_id/2', 5, '页面跳转中...');
}
```

> redirect函数的第一个参数是要跳转的实际URL地址。

快速入门 15：页面请求和AJAX

快速入门（十五）：页面请求和AJAX

本章内容描述了如何判断页面请求类型和AJAX数据返回。

判断请求类型

在很多情况下面，我们需要判断当前操作的请求类型是GET 、POST 、PUT或 DELETE，一方面可以针对请求类型作出不同的逻辑处理，另外一方面有些情况下面需要验证安全性，过滤不安全的请求。

系统内置了一些常量用于判断请求类型，包括：

常量	说明
IS_GET	判断是否是GET方式提交
IS_POST	判断是否是POST方式提交
IS_PUT	判断是否是PUT方式提交
IS_DELETE	判断是否是DELETE方式提交
IS_AJAX	判断是否是AJAX提交
REQUEST_METHOD	当前提交类型

使用举例如下：

```
class UserController extends Controller{
    public function update(){
        if (IS_POST){
            $User = M('User');
            $User->create();
            $User->save();
            $this->success('保存完成');
        }else{
            $this->error('非法请求');
        }
    }
}
```

> 个别情况下判断AJAX请求的时候，你可能需要在表单里面添加一个隐藏域，告诉后台属于ajax方式提交，默认的隐藏域名称是ajax（可以通过VAR_AJAX_SUBMIT配置），如果是JQUERY类库的话，则无需添加任何隐藏域即可自动判断。

AJAX返回

ThinkPHP可以很好的支持AJAX请求，系统的`\Think\Controller`类提供了`ajaxReturn`方法用于AJAX调用后返回数据给客户端。并且支持`JSON`、`JSONP`、`XML`和`EVAL`四种方式给客户端接受数据，并且支持配置其他方式的数据格式返回。

ajaxReturn方法调用示例：

```
$data = 'ok';
$this->ajaxReturn($data);
```

支持返回数组数据：

```
$data['status']  = 1;
$data['content'] = 'content';
$this->ajaxReturn($data);
```

默认配置采用JSON格式返回数据（通过配置**DEFAULT_AJAX_RETURN**进行设置），我们可以指定格式返回，例如：

```
// 指定XML格式返回数据
$data['status']  = 1;
$data['content'] = 'content';
$this->ajaxReturn($data,'xml');
```

返回数据data可以支持字符串、数字和数组、对象，返回客户端的时候根据不同的返回格式进行编码后传输。如果是JSON/JSONP格式，会自动编码成JSON字符串，如果是XML方式，会自动编码成XML字符串，如果是EVAL方式的话，只会输出字符串data数据。

> JSON和JSONP虽然只有一个字母的差别，但其实他们根本不是一回事儿：JSON是一种数据交换格式，而JSONP是一种非官方跨域数据交互协议。一个是描述信息的格式，一个是信息传递的约定方法。

默认的JSONP格式返回的处理方法是`jsonpReturn`，如果你采用不同的方法，可以设置：

```
'DEFAULT_JSONP_HANDLER' =>  'myJsonpReturn', // 默认JSONP格式返回的处理方法
```

或者直接在页面中用callback参数来指定。

快速入门 16：伪静态

快速入门（十六）：伪静态

URL伪静态通常是为了满足更好的SEO效果，ThinkPHP支持伪静态URL设置，可以通过设置 `URL_HTML_SUFFIX` 参数随意在URL的最后增加你想要的静态后缀，而不会影响当前操作的正常执行。

单个URL后缀

默认情况下，伪静态的设置为 `html` ，因此下面的URL访问是等效的：

```
http://serverName/Home/Blog/index
http://serverName/Home/Blog/index.html
```

但后者更具有静态页面的URL特征，并且不会影响原来参数的使用。

但如果我们访问

```
http://serverName/Home/Blog/index.xml
```

则会提示出错。

除非我们设置了：

```
'URL_HTML_SUFFIX'=>'xml'
```

全后缀支持

如果我们设置伪静态后缀为空，则可以支持所有的静态后缀访问，并且会记录当前的伪静态后缀到常量 `__EXT__` ，但不会影响正常的页面访问。

```
'URL_HTML_SUFFIX'=>''
```

设置后，下面的URL访问都有效：

```
http://serverName/Home/blog/index.html
http://serverName/Home/blog/index.shtml
http://serverName/Home/blog/index.xml
http://serverName/Home/blog/index.pdf
```

可以通过常量 __EXT__ 判断当前访问的后缀，例如：

```
if('pdf'==__EXT__){
  // 输出PDF文档
}elseif('xml'==__EXT__){
  // 输出XML格式文档
}
```

多个后缀支持

如果希望仅支持设置的多个伪静态后缀访问，可以设置如下：

```
// 多个伪静态后缀设置 用|分割
'URL_HTML_SUFFIX' => 'html|shtml|xml'
```

那么，当访问 http://serverName/Home/blog/index.pdf 的时候会报系统错误。

禁止访问后缀

可以设置禁止访问的URL后缀，例如：

```
'URL_DENY_SUFFIX' => 'pdf|ico|png|gif|jpg', // URL禁止访问的后缀设置
```

如果访问 http://serverName/Home/blog/index.pdf 就会直接返回404错误。

> **注意：**
> URL_DENY_SUFFIX 的优先级比 URL_HTML_SUFFIX 要高。

快速入门 17：操作绑定到类

快速入门（十七）：操作绑定到类

如果你的应用规模比较大，每个操作方法彼此相对独立，那么就可以尝试下操作绑定到类的功能。

定义

系统提供了把每个操作方法定位到一个类的功能，可以让你的开发工作更细化，可以设置参数**ACTION_BIND_CLASS**，例如：

```
'ACTION_BIND_CLASS'    =>    True,
```

设置后，我们的控制器定义有所改变，以URL访问为 http://serverName/Home/Index/index 为例，原来的控制器文件定义位置为：

```
Application/Home/Controller/IndexController.class.php
```

控制器类的定义如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    public function index(){
        echo '执行Index控制器的index操作';
    }
}
```

可以看到，实际上我们调用的是 Home\Controller\IndexController 类的**index**方法。

设置操作绑定到类以后，控制器文件位置改为：

```
Application/Home/Controller/Index/index.class.php
```

控制器类的定义如下：

```
namespace Home\Controller\Index;
use Think\Controller;
class index extends Controller{
    public function run(){
        echo '执行Index控制器的index操作';
    }
}
```

现在，我们调用的其实是 `Home\Controller\Index\index` 类的**run**方法。

> 注意：操作方法类的命名空间比之前要多了一个控制器名称，这个地方很容易忽略。

run方法依旧可以支持传入参数和进行Action参数绑定操作。

```
namespace Home\Controller\Index;
use Think\Controller;
class index extends Controller{
    public function run($name=''){
        echo 'Hello,'.$name.'!';
    }
}
```

我们访问

```
http://serverName/Home/Index/index/name/thinkphp
```

可以看到输出结果为：

Hello，thinkphp!

前置和后置操作

当设置操作方法绑定到类后，前置和后置操作的定义有所改变，只需要在类里面定义 `_before_run` 和 `_after_run` 方法即可，例如：

```
namespace Home\Controller\Index;
use Think\Controller;
class index extends Controller{
    public function _before_run(){
        echo 'before_'.ACTION_NAME;
    }

    public function run(){
        echo '执行Index控制器的index操作';
    }

    public function _after_run(){
        echo 'after_'.ACTION_NAME;
    }
}
```

空控制器

操作方法绑定到类后，一样可以支持空控制器，我们可以创建 `Application/Home/Controller/_empty` 目录，即表示如果找不到当前的控制器的话，会到_empty控制器目录下面定位操作方法。

例如，我们访问了URL地址 `http://serverName/Home/Test/index`,但并不存在 `Application/Home/Controller/Test` 目录，但是有定义 `Application/Home/Controller/_empty` 目录。

并且我们有定义：

```
Application/Home/Controller/_empty/index.class.php
```

控制器定义如下：

```
namespace Home\Controller\_empty;
use Think\Controller;
class index extends Controller{
    public function run(){
        echo '执行'CONTROLLER_NAME.'控制器的'.ACTION_NAME.'操作';
    }
}
```

访问 `http://serverName/Home/Test/index` 后 输出结果显示：

```
执行Test控制器的index操作
```

空操作

操作绑定到类后，我们依然可以实现空操作方法，我们只要定义一个 Home\Controller\Index_empty 类，就可以支持Index控制器的空操作访问，例如： 控制器定义如下：

```
namespace Home\Controller\Index;
use Think\Controller;
class _empty extends Controller{
    public function run(){
        echo '执行Index控制器的'.ACTION_NAME.'操作';
    }
}
```

当我们访问 http://serverName/Home/Index/test 后 输出结果显示：

```
执行Index控制器的test操作
```

快速入门 18：多层控制器

快速入门（十八）：多层控制器

ThinkPHP支持多层业务控制器的支持，给中大型应用提供了方便。

定义多层控制器

我们通常所了解的控制器其实是Controller控制器类，而且大多数也是继承了核心的Think\Controller类，由于该类控制器是通过URL访问请求后调用的，因此也称之为访问控制器，事实上，ThinkPHP可以支持更多的控制器分层，多层控制器的定义完全取决于项目的需求，例如我们可以分为业务控制器和事件控制器：

> Home\Controller\UserController //用于用户的业务逻辑控制和调度
> Home\Event\UserEvent //用于用户的事件响应操作

```
├─Controller 访问控制器
│  ├─UserController.class.php
│  ...
├─Event 事件控制器
│  ├─UserEvent.class.php
│  ...
```

一个标准的访问控制器定义如下：

```
namespace Home\Controller;
class UserController extend Think\Controller {
    // 默认操作方法
    public function index(){
        //...
    }

    // 用户注册操作方法
    public function register(){
        //...
    }
}
```

> 注：访问控制器的名称并非一定是Controller，而是通过DEFAULT_C_LAYER设置的，默认设置是Controller。

访问控制器负责外部的交互响应，通过URL请求调用，例如：

```
http://serverName/Home/User/index
http://serverName/Home/User/register
```

而事件控制器负责内部的事件响应，并且只能在内部调用，所以是和外部隔离的。

> 确切的说，所有访问控制器之外的分层控制器都只能内部实例化调用。

```
namespace Home\Event;
class UserEvent {
    // 用户登录事件
    public function login(){
        echo 'login event';
    }

    // 用户登出事件
    public function logout(){
        echo 'logout event';
    }
}
```

> 如果是定义其他的控制器层，则不一定必须要继承系统的Controller类或其子类，通常需要输出模版的时候才需要继承Controller类。

调用多层控制器

访问控制器是通过URL请求调用，访问控制器之外的分层控制器都只能内部调用，调用多层控制器可以通过两种方式：

直接实例化

```
namespace Home\Controller;
class UserController extend Think\Controller {
    // 默认操作方法
    public function index(){
        // 触发事件
        $event = new \Home\Event\UserEvent();
        $event->login();
    }

}
```

A函数实例化

```
namespace Home\Controller;
class UserController extend Think\Controller {
    // 默认操作方法
    public function index(){
        // 触发事件
        $event = A('User','Event');
        $event->login();
        // 或者直接使用
        // R('User/login','','Event');
    }
}
```

Widget实例

Widget类的实现可以作为分层控制器的另外一个典型实例。

举个例子，我们在页面中实现一个分类菜单的Widget，首先我们要定义一个Widget控制器层MenuWidget，如下：

```
namespace Home\Widget;
class MenuWidget extends Think\Controller {
    public function index(){
        echo 'menuWidget';
    }
 }
```

类文件位于 `Home/Widget/MenuWidget.class.php` 。

然后，我们在需要显示分类菜单的模版中通过W方法调用这个Widget。

```
{~W('Menu/index')}
```

执行后的输出结果是： menuWidget

如果需要在调用Widget的时候传入参数，可以这样定义：

```
namespace Home\Widget;
class MenuWidget extends Think\Controller {
    public function index($id,$name){
        echo $id.':'.$name;
    }
 }
```

在需要显示分类菜单的模版中添加如下的Widget调用代码如下：

```
{~W('Menu/index',array(5,'thinkphp'))}
```

则会输出 5:thinkphp

来一个复杂一点的例子：

```
namespace Home\Widget;
class MenuWidget extends Think\Controller {
    public function index(){
        $menu = M('Cate')->getField('id,title');
        $this->assign('menu',$menu);
        $this->display('Widget/menu');
    }
}
```

CateWiget类渲染了一个模版文件 `Home/View/Widget/menu.html`，
在menu.html模版文件中的用法：

```
<foreach name="menu" item="title">
{$key}:{$title}
</foreach>
```

快速入门 19：自动验证

快速入门（十九）：自动验证

ThinkPHP的模型可以支持数据自动验证功能，用于验证提交到数据库的数据的有效性和安全性。

> 该自动验证为后端验证，并非前端验证。

可以支持两种方式的自动验证，包括静态验证和动态验证。

静态验证

静态验证是指把验证的规则直接定义在模型类里面，也就是说，这种验证必须有自定义的模型类存在，假设我们的用户注册表单如下：

```
<form method="post" action="/user/register">
   <input type="text" name="name" >
   <input type="text" name="email" >
   <input type="password" name="password" >
   <input type="password" name="repassword" >
   <img height="42" src="/user/verify" >
   <input name="verify" type="text" maxlength="4">
</form>
```

我们需要对用户注册表单进行一些相关的验证操作，包括：

> 用户名必须；
> 验证码用户名是否已经存在；
> 如果输入邮箱则验证邮箱格式是否正确；
> 验证密码是否为指定长度；
> 确认密码是否和密码一致；

按照上述的验证规则，我们定义好自动验证规则，为UserModel类添加 `$_validate` 属性定义如下：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
   protected $_validate = array(
     array('name','require','用户名必须！'), // 用户名必须
     array('name','','帐号名称已经存在！',1,'unique',1), // 验证用户名是否已经存在
     array('email','email','Email格式错误！',2), // 如果输入则验证Email格式是否正确
     array('password','6,30','密码长度不正确',0,'length'), // 验证密码是否在指定长度范围
     array('repassword','password','确认密码不一致',0,'confirm'), // 验证确认密码是否和密码一致
   );
}
```

定义好自动验证规则后，我们就可以在控制器使用create方法进行自动验证了。

```
$User = D("User"); // 实例化User对象
if (!$User->create()){
     // 如果创建失败 表示验证没有通过 输出错误提示信息
     exit($User->getError());
}else{
     // 验证通过 可以进行其他数据操作
     $User->add();
}
```

在进行自动验证的时候，系统会对定义好的验证规则进行依次验证。对于某个字段也可以进行多次规则验证，如果某一条验证规则没有通过，则会报错，getError方法返回的错误信息（字符串）就是对应字段的验证规则里面的错误提示信息。

验证规则

验证规则的定义格式为：

```
array(验证字段1,验证规则,错误提示,[验证条件,附加规则,验证时间]),
```

验证字段 （必须）

需要验证的表单字段名称，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。有个别验证规则和字段无关的情况下，验证字段是可以随意设置的，例如expire有效期规则是和表单字段无关的。如果定义了字段映射的话，这里的验证字段名称应该是实际的数据表字段而不是表单字段。

验证规则 （必须）

要进行验证的规则，需要结合附加规则，如果在使用正则验证的附加规则情况下，系统还内置了一些常用正则验证的规则，可以直接作为验证规则使用，包括：require 字段必须、email 邮箱、url URL地址、currency 货币、number 数字。

提示信息（必须）

用于验证失败后的提示信息定义

验证条件（可选）

包含下面几种情况：

- self::EXISTS_VALIDATE 或者0 存在字段就验证（默认）
- self::MUST_VALIDATE 或者1 必须验证
- self::VALUE_VALIDATE或者2 值不为空的时候验证

附加规则（可选）

配合验证规则使用，包括下面一些规则：

规则	说明
regex	正则验证，定义的验证规则是一个正则表达式（默认）
function	函数验证，定义的验证规则是一个函数名
callback	方法验证，定义的验证规则是当前模型类的一个方法
confirm	验证表单中的两个字段是否相同，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
notequal	验证是否不等于某个值，该值由前面的验证规则定义（3.1.2版本新增）
in	验证是否在某个范围内，定义的验证规则可以是一个数组或者逗号分割的字符串
notin	验证是否不在某个范围内，定义的验证规则可以是一个数组或者逗号分割的字符串（3.1.2版本新增）
length	验证长度，定义的验证规则可以是一个数字（表示固定长度）或者数字范围（例如3,12 表示长度从3到12的范围）
between	验证范围，定义的验证规则表示范围，可以使用字符串或者数组，例如1,31或者array(1,31)
notbetween	验证不在某个范围，定义的验证规则表示范围，可以使用字符串或者数组（3.1.2版本新增）
expire	验证是否在有效期，定义的验证规则表示时间范围，可以到时间，例如可以使用2012-1-15,2013-1-15 表示当前提交有效期在2012-1-15到2013-1-15之间，也可以使用时间戳定义
ip_allow	验证IP是否允许，定义的验证规则表示允许的IP地址列表，用逗号分隔，例如201.12.2.5,201.12.2.6
ip_deny	验证IP是否禁止，定义的验证规则表示禁止的ip地址列表，用逗号分隔，例如201.12.2.5,201.12.2.6

规则	说明
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，当表单数据中包含主键字段时unique不可用于判断主键字段本身

验证时间（可选）

- self::MODEL_INSERT或者1新增数据时候验证
- self::MODEL_UPDATE或者2编辑数据时候验证
- self::MODEL_BOTH或者3全部情况下验证（默认）

这里的验证时间需要注意，并非只有这三种情况，你可以根据业务需要增加其他的验证时间。

动态验证

如果采用动态验证的方式，因为不依赖模型类的定义，就比较灵活，可以根据不同的需要，在操作同一个模型的时候使用不同的验证规则，而且可以直接使用M函数操作模型类，例如上面的静态验证方式可以改为：

```
$rules = array(
    array('name','require','用户名必须！'), // 用户名必须
    array('name','','帐号名称已经存在！',1,'unique',1), // 验证用户名是否已经存在
    array('email','email','Email格式错误！',2), // 如果输入则验证Email格式是否正确
    array('password','6,30','密码长度不正确',0,'length'), // 验证密码是否在指定长度范围
    array('repassword','password','确认密码不一致',0,'confirm'), // 验证确认密码是否和密码一致
);

$User = M("User"); // 实例化User对象
if (!$User->validate($rules)->create()){
    // 如果创建失败 表示验证没有通过 输出错误提示信息
    exit($User->getError());
}else{
    // 验证通过 可以进行其他数据操作
    $User->add();
}
```

快速入门 20：自动完成

快速入门（二十）：自动完成

自动完成是ThinkPHP提供用来完成数据自动处理和过滤的方法，使用create方法创建数据对象的时候会自动完成数据处理。

规则定义

自动完成通常用来完成默认字段写入，安全字段过滤以及业务逻辑的自动处理等，和自动验证的定义方式类似，自动完成的定义也支持静态定义和动态定义两种方式。

1. 静态方式：在模型类里面通过$_auto属性定义处理规则。
2. 动态方式：使用模型类的auto方法动态创建自动处理规则。

两种方式的定义规则都采用：

```
array(
    array(完成字段1,完成规则,[完成条件,附加规则]),
    array(完成字段2,完成规则,[完成条件,附加规则]),
    ......
);
```

说明

完成字段（必须）

需要进行处理的数据表实际字段名称。

完成规则（必须）

需要处理的规则，配合附加规则完成。

完成时间（可选）

设置自动完成的时间，包括：

设置	说明
self::MODEL_INSERT或者1	新增数据的时候处理（默认）
self::MODEL_UPDATE或者2	更新数据的时候处理
self::MODEL_BOTH或者3	所有情况都进行处理

附加规则（可选）

设置自动完成的附加规则，包括：

规则	说明
function	使用函数，表示填充的内容是一个函数名
callback	回调方法 ，表示填充的内容是一个当前模型的方法
field	用其它字段填充，表示填充的内容是一个其他字段的值
string	字符串（默认方式）
ignore	为空则忽略（3.1.2新增）

预先在模型类里面定义好自动完成的规则，我们称之为静态定义。例如，我们在模型类定义 _auto 属性：

```
namespace Home\Model;

use Think\Model;

class UserModel extends Model{
    protected $_auto = array (
        array('status','1'),  // 新增的时候把status字段设置为1
        array('password','md5',3,'function') , // 对password字段在新增和编辑的时候使md5函数处理
        array('name','getName',3,'callback'), // 对name字段在新增和编辑的时候回调getName方法
        array('update_time','time',2,'function'), // 对update_time字段在更新的时候写入当前时间戳
    );
}
```

然后，就可以在使用create方法创建数据对象的时候自动处理：

```
$User = D("User"); // 实例化User对象
if (!$User->create()){ // 创建数据对象
    // 如果创建失败 表示验证没有通过 输出错误提示信息
    exit($User->getError());
}else{
    // 验证通过 写入新增数据
    $User->add();
}
```

如果你没有定义任何自动验证规则的话，则不需要判断create方法的返回值：

```
$User = D("User"); // 实例化User对象
$User->create(); // 生成数据对象
$User->add(); // 新增用户数据
```

或者更简单的使用：

```
$User = D("User"); // 实例化User对象
$User->create(); // 生成数据对象
$User->add(); // 写入数据
```

create方法默认情况下是根据表单提交的post数据生成数据对象，我们也可以根据其他的数据源来生成数据对象，你也可以明确指定当前创建的数据对象自动处理的时间是新增还是编辑数据，例如：

```
$User = D("User"); // 实例化User对象
$userData = getUserData(); // 通过方法获取用户数据
$User->create($userData,2); // 根据userData数据创建数据对象，并指定为更新数据
$User->add();
```

create方法的第二个参数就用于指定自动完成规则中的完成时间，也就是说create方法的自动处理规则只会处理符合完成时间的自动完成规则。 create方法在创建数据的时候，已经自动过滤了非数据表字段数据信息，因此不需要担心表单会提交其他的非法字段信息而导致数据对象写入出错，甚至还可以自动过滤不希望用户在表单提交的字段信息（详见字段合法性过滤）。

3.1.2版本开始新增了ignore完成规则，这一规则表示某个字段如果留空的话则忽略，通常可用于修改用户资料时候密码的输入，定义如下：

```
array('password','',2,'ignore')
```

表示password字段编辑的时候留空则忽略。

动态完成

除了静态定义之外，我们也可以采用动态设置自动完成规则的方式来解决不同的处理规则。

```
$rules = array (
    array('status','1'),  // 新增的时候把status字段设置为1
    array('password','md5',3,'function') , // 对password字段在新增和编辑的时候使md5函数处理
    array('update_time','time',2,'function'), // 对update_time字段在更新的时候写入当前时间戳
);
$User = M('User');
$User->auto($rules)->create();
$User->add();
```

修改数据对象

在使用create方法创建好数据对象之后，此时的数据对象保存在内存中，因此仍然可以操作数据对象，包括修改或者增加数据对象的值，例如：

```
$User = D("User"); // 实例化User对象
$User->create(); // 生成数据对象
$User->status = 2; // 修改数据对象的status属性
$User->register_time = NOW_TIME; // 增加register_time属性
$User->add(); // 新增用户数据
```

一旦调用了add方法（或者save方法），创建在内存中的数据对象就会失效，如果希望创建好的数据对象在后面的数据处理中再次调用，可以保存数据对象先，例如：

```
$User = D("User"); // 实例化User对象
$data = $User->create(); // 保存生成的数据对象
$User->add();
```

不过要记得，如果你修改了内存中的数据对象并不会自动更新保存的数据对象，因此下面的用法是错误的：

```
$User = D("User"); // 实例化User对象
$data = $User->create(); // 保存生成的数据对象
$User->status = 2; // 修改数据对象的status属性
$User->register_time = NOW_TIME; // 增加register_time属性
$User->add($data);
```

上面的代码我们修改了数据对象，但是仍然写入的是之前保存的数据对象，因此对数据对象的更改操作将会无效。