



ThinkPHP Framework 1.0.1

Developer Manual

ThinkPHP 1.0.1

开发人员指南

编写：ThinkPHP 文档组

最后更新：2008-02-02

第1部分 版权信息

发布本资料须遵守开放出版许可协议 1.0 或者更新版本。

未经版权所有者明确授权，禁止发行本文档及其被实质上修改的版本。

未经版权所有者事先授权，禁止将此作品及其衍生作品以标准（纸质）书籍形式发行。

如果有兴趣再发行或再版本手册的全部或部分内容，不论修改过与否，或者有任何问题，请联系版权所有者 liu21st@gmail.com。

对 ThinkPHP 有任何疑问或者建议，请进入官方网站[<http://thinkphp.cn>] 发布相关讨论。并在此感谢 ThinkPHP 团队的所有成员和所有关注和支持 ThinkPHP 的朋友。

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

文档规范约定：

请注意文档中相关信息提示以及各自代表的含义



表示提示信息



表示需要注意的事项



词语解释



给出的友好提示



可能存在的 BUG



需要引起注意，否则会导致问题



错误的操作

目 录:

| | | |
|--------|----------------------------|------------|
| 第 1 部分 | 版权信息..... | 2 |
| 第 2 部分 | 入门基础..... | 5 |
| 2.1 | ThinkPHP 是什么 | 5 |
| 2.2 | 功能分布 | 5 |
| 2.3 | 许可协议..... | 6 |
| 2.4 | 版本区别..... | 6 |
| 2.5 | 系统特色..... | 7 |
| 2.6 | 目录结构..... | 15 |
| 2.7 | 环境要求..... | 17 |
| 2.8 | 获取 ThinkPHP..... | 17 |
| 2.9 | ThinkPHP 比较..... | 错误! 未定义书签。 |
| 第 3 部分 | 构建应用..... | 18 |
| 3.1 | 示例一: Hello, ThinkPHP | 18 |
| 3.2 | 示例二: Blog..... | 22 |
| 第 4 部分 | 开发指南..... | 36 |
| 4.1 | 类库和函数库 | 37 |
| 4.2 | Action 控制器..... | 45 |
| 4.3 | ActiveRecord 基础..... | 60 |
| 4.4 | 更多的 ActiveRecord 特性..... | 75 |
| 4.5 | 模板输出 | 92 |
| 4.6 | AJAX 支持 | 96 |
| 4.7 | 异常处理..... | 98 |
| 4.8 | 日志和调试 | 99 |
| 4.9 | 文件上传..... | 102 |
| 4.10 | 权限控制..... | 108 |
| 4.11 | 插件机制..... | 112 |

| | | |
|--------|----------------|-----|
| 第 5 部分 | 模板指南..... | 112 |
| 5.1 | 模板概述..... | 113 |
| 5.2 | 模板标签..... | 115 |
| 5.3 | 模板变量..... | 115 |
| 5.4 | 模板注释..... | 118 |
| 5.5 | 公共模板..... | 118 |
| 5.6 | 布局模板..... | 119 |
| 5.7 | 标签库..... | 120 |
| 第 6 部分 | 附录..... | 128 |
| | 附录 A 发展历程..... | 128 |
| | 附录 B 系统常量..... | 129 |
| | 附录 C 配置参数..... | 131 |
| | 附录 D 文件列表..... | 140 |
| 第 7 部分 | FAQ..... | 142 |

第2部分 入门基础

2.1 ThinkPHP 是什么

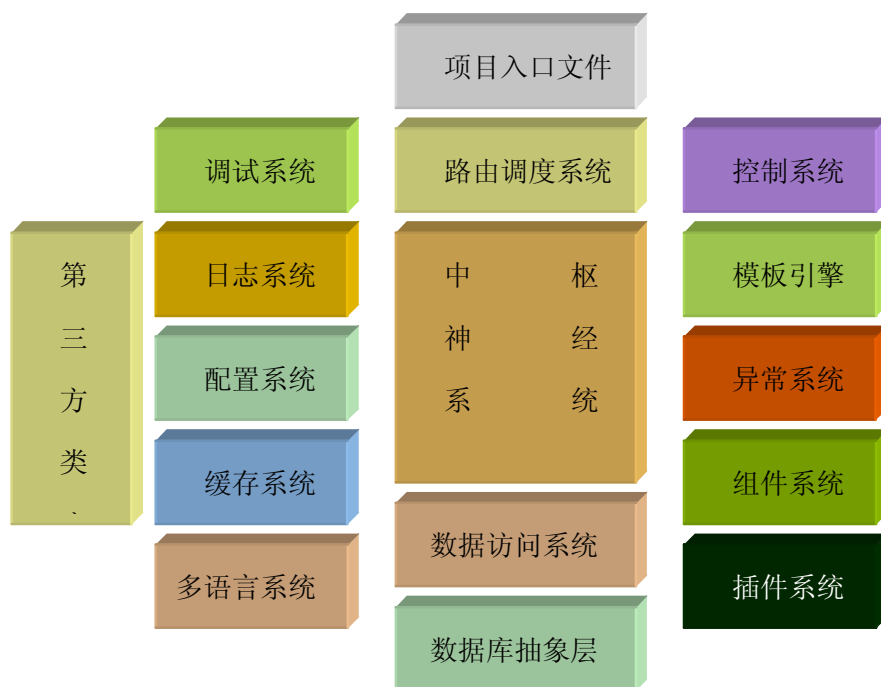
简单的说，ThinkPHP 是一个开源的快速、兼容而且简单的面向对象的轻量级 PHP 开发框架，遵循 Apache2 开源协议发布，是为了简化企业级应用开发和敏捷 WEB 应用开发而诞生的。借鉴了国外很多优秀的框架和模式，使用面向对象的开发结构和 MVC 模式，融合了 Struts 的 Action 思想和 JSP 的 TagLib（标签库）、RoR 的 ORM 映射和 ActiveRecord 模式，封装了 CURD 和一些常用操作，单一入口模式等，在模版引擎、缓存机制、认证机制和扩展性方面均有独特的表现。

使用 ThinkPHP，你可以更方便和快捷的开发和部署应用，当然不仅仅是企业级应用，任何 PHP 应用开发都可以从 ThinkPHP 的简单、兼容和快速的特性中受益。简洁、快速和实用是 ThinkPHP 发展秉承的宗旨，为此 ThinkPHP 会不断吸收和融入更好的技术以保证其新鲜和活力，提供 WEB 应用开发的最佳实践！

2.2 功能分布

ThinkPHP 不是你所想象的只是一个提供一组类库的框架，ThinkPHP 是一个具有你开发所需的全功能的开发平台，是一个有机组合体，是一个让你可以立刻开始编写应用逻辑的开发框架，并且具有很好的扩展性。

下面的图示可以看出 ThinkPHP 的功能分布。



2.3 许可协议

ThinkPHP 遵循 Apache2 开源许可协议发布,意味着你可以免费使用 ThinkPHP,甚至允许把你的 ThinkPHP 应用采用商业闭源发布。具体参考 Apache2 许可协议内容:

<http://www.apache.org/licenses/LICENSE-2.0>

2.4 版本区别

ThinkPHP 目前分为两个系列的版本:

0.* 系列版本属于 PHP4 兼容版本,可以运行在 PHP4.3.0 以上的环境,考虑到 PHP4 的生命周期已经结束,该版本目前最新版本 0.9.8 处于维护状态,基本不再更新。

 **1.*** 系列版本属于 PHP5 重构版本,只能运行在 PHP5 以上,这是响应 GoPHP5 后的重构版本,提供了更高的性能和更多的功能。

1.* 也是 ThinkPHP 以后的重点发展和更新版本。从 1.0.0 版本开始 ThinkPHP 提供了更加完善的文档支持。本文档的内容主要针对最新的 1.0.0 版本。

2.5 系统特色

ThinkPHP 框架最早是从 Struts 结构移植过来并做了改进，并且在后续的不断完善中增加了很多特性，同时也借鉴了国外很多优秀的框架和模式，使用面向对象的开发结构和 MVC 模式，包含了多项目的支持以及对企业级开发和部署的支持，融合了 Struts 的思想和 JSP 的 TagLib（标签库）、RoR 的 ORM 映射和 ActiveRecord 模式，封装了 CURD 和一些常用操作和单一入口模式等，并且在可维护性、安全性、可移植性以及可扩展性方面都有很好的表现。

作为一个整体开发解决方案，ThinkPHP 能够解决应用开发中的大多数需要，因为其自身包含了底层架构、兼容处理、基类库、数据库访问层、模板引擎、缓存机制、插件机制、角色认证、表单处理等常用的组件，并且对于跨版本、跨平台和跨数据库移植都比较方便。并且每个组件都是精心设计和完善的，应用开发过程仅仅需要关注您的业务逻辑。

下面是对这些特色的简单介绍，在后面会有更加详细的使用方法。

2.5.1 架构模式

和很多应用框架一样，ThinkPHP 也采用了 MVC 模式，应用控制器（核心控制器）和 Action 控制器都承担了控制器的角色，开发人员主要涉及到 Action 控制器的开发，而且相当简单，只是添加不同 Action 的业务逻辑控制，调用由应用控制器来负责。模型的定义由 Model 类来完成。系统的视图（模板）的实现是和框架无关的，做到了 100% 分离，可以独立预览和制作。内置的模板引擎给模板的制作带来了更方便有效的方法。

在原来的 Dao 模式的基础上，新版正式引入了 ActiveRecord 模式，并且对其做了增强，除了可以作为数据对象之外，还可以作为数据集对象，提供了最佳的开发体验。ThinkPHP 实现了 ActiveRecord 模式的 ORM 模型，采用了非标准的 ORM 模型：表映射到类，记录（集）映射到对象，字段属性映射到对象的虚拟属性，拥有领域对象的领域属性、领域方法和持久性方法。最大的特点就是使用方便，从而达到敏捷开发的目的。

2.5.2 基类库

框架内置有基类库支持，就连 ThinkPHP 框架本身的核心类都属于基类库的部分，基类库和框架是合为一体的。基类库包括核心类库包 Think 和扩展类库包 ORG，以及商业 Com 包。其中 ORG 包提供了很多

的实用类库。

ThinkPHP 的所有类库引用采用命名空间的方式，以保证类库的唯一性。在应用开发的时候不需要使用传统的 `include` 或者 `require` 指令，ThinkPHP 的类库导入方式和 Java 的 `Import` 机制类似，只要遵循命名规范，类库的导入和使用就显得方便有效，熟悉 .Net 的开发人员还可以使用 `using` 的别名，内建的导入缓存支持避免重复的文件导入。1.0.0 版本还增加了匹配导入和导入冲突的判断功能。

2.5.3 项目编译

新版引入了项目编译和核心编译机制，把项目运行过程的一些分散类库和文件编译成一个独立的运行缓存文件，最大程度减少文件的加载，提升整体性能。

2.5.4 数据访问

为了提供更好的效率和移植性，框架内置了抽象数据库访问层，支持 `Mysql`、`MsSQL`、`Sqlite`、`PgSQL`、`Firebird` 等多种数据库，并且引入了 `PDO` 的支持，还可以通过插件的方式增加需要的数据库驱动。

在应用开发中，我们无需访问具体的数据库驱动，而只需要通过数据库公共类库 `Db` 来访问底层数据库操作，在大多数情况下，数据库操作是封装在数据对象的领域方法之中。在使用 ThinkPHP 开发的过程中，只需要告诉系统数据库的连接信息就可以了。

2.5.5 SEO 支持

ThinkPHP 提供了对搜索引擎友好的充分支持，默认的 `Dispatch` 器会把所有的 `GET` 方式传值转换成 `PATHINFO` 形式，并且会把 `PATHINFO` 参数转换成 `GET` 数组变量，让您可以保持以前的传值和链接形式，转换的工作由框架来完成。您还可以关闭 `PATHINFO` 形式，甚至还支持自定义的 `PATHINFO` 格式，让你的 URL 富有创造性。

系统支持的 URL 模式包括普通模式、`PATHINFO` 模式和 `REWRITE` 模式，并且都提供路由支持。默认为 `PATHINFO` 模式，提供最好的用户体验和 SEO 支持

并且系统支持多种方式的 `PATHINFO` 地址，您可以根据应用的要求灵活地选用何种方式。

第一种：普通模式 参数没有顺序 `/m/module/a/action/id/1`

第二种：智能模式 自动识别模块和操作 `/module/action/id/1/` 或者 `/module,action,id,1/`

系统默认采用智能模式。还可以设置不同的分割符，

例如：把 PATH_DEPR 的值设置为 `:` 那么您的 URL 看起来会象下面的效果

`http://<serverIp>/module:action:id:1/`

PATHINFO 技术对以往的编程方式没有影响, 因为 GET 和 POST 方式传值依然有效, 因为系统对 PATHINFO 方式是自动处理, 会把类似 `?m=moudule&a=action&id=2` 的 URL 地址重新转换为 `/module/action/id/2` 或者你设置的格式。

不同的是在由于因为模拟了目录, 所以在模板页面中对当前目录下面的连接处理不当可能会导致失效。

如果你的系统不支持 PATHINFO 方式, 或者你依然不打算采用 PATHINFO 方式, 那么可以在项目配置文件中把 URL_MODEL 的值设置为 0 就可以使用传统模式了。

除此之外, 还有另外一种选择, 就是使用 PATHINFO 兼容模式, 通过设置 VAR_PATHINFO 变量, 系统就会自动判断, 例如, 下面的 URL

`http://<serverIp>/index.php?_info=/module/action/id/1/`

其中 `_info` 就是设置的兼容模式 PATHINFO 的 GET 变量, 该地址等效于

支持 PATHINFO 模式下面的 `http://<serverIp>/index.php/module/action/id/1/`

2.5.6 配置灵活

ThinkPHP 提供了灵活的配置功能, 采用最有效率的 PHP 返回数组方式定义, 支持惯例配置、项目配置、调试配置和模块配置, 并且会自动生成配置缓存文件, 无需重复解析的开销。对于有些简单的应用, 你无需配置任何配置文件, 而对于复杂的要求, 你还可以增加模块配置文件, 因为 ThinkPHP 的动态配置使得你在开发过程中灵活调整配置参数。

2.5.7 查询语言

查询操作是开发过程中最经常使用的, ThinkPHP 提供的 ORM 查询语言支持普通查询、统计查询、定位查询、SQL 查询, 配合新的动态查询和组合查询机制和更多的查询方法, 让你的查询操作随心所欲。并且新版的数据库中间层和模型之间的操作更加优化。

2.5.8 延迟加载

ThinkPHP 在进行数据库操作的时候具有延迟连接和延迟加载的特性, 也有的称为惰性加载, 可以最大程度避免了数据库的开销, 并做到按需获取, 提升了性能。系统在模型初始化的时候会创建数据库操

作对象，但是这个时候并不会马上进行数据库连接，而是会在第一次实际查询的时候才会进行连接。对于有些查询，还可以采用延迟加载，在执行查询操作的时候并不会立刻进行查询，而是等到需要使用查询返回数据的时候才进行查询。该特性对于查询大量数据的时候非常有用。

2.5.9 事务支持

ThinkPHP 提供了单数据库的事务支持，包括事务启动、提交和回滚操作。

2.5.10 脱管对象

新版 ThinkPHP 摆脱了需要手动定义 Vo 对象的麻烦，脱管对象使得系统可以自动获取数据表字段来动态创建属性，并且可以支持将脱管对象作为数据对象传递。

2.5.11 缓存机制

ThinkPHP 在数据缓存方面包括 SQL 查询缓存、数据对象缓存、Action 缓存、视图缓存、静态页面缓存以及浏览器缓存等多种机制，采用了包括文件方式、共享内存方式和数据库方式在内的多种方式进行缓存，通过插件方式还可以增加以后需要的缓存类，让应用开发可以选择更加适合自己的缓存方式，从而有效地提高应用执行效率。在有些操作中，系统会自动进行数据对象的缓存以提升性能。

2.5.12 模板引擎

ThinkPHP 内置了一个性能卓越的模板引擎，是一个使用了 XML 标签库技术的编译型模板引擎，支持两种类型的模板标签，使用了动态编译和缓存技术，而且支持自定义标签库，利用该模板引擎可以方便地定义模板文件，从而达到快速开发的目的。

ThinkPHP 内置模板引擎的模板标签有两种类型：第一种是普通标签，类似于 Smarty 的模板标签，在功能方面作了部分简化，增强了显示功能，弱化了逻辑控制功能；第二种是 XML 标签库形式，该模板技术是 ThinkPHP 特有的标签技术，有效地借鉴了 JSP 的标签库技术，在控制功能和各方面都比较强大，而且允许自定义标签库，是新版 ThinkPHP 系统引入和推荐的模板标签技术。两种标签方式的结合使用，可以让您的模板定义功能相当强大。

ThinkPHP 架构的设计中模板和程序完全分离，一套模板文件就是一个目录，模板是标准 html 文件（可

以配置成其它后缀，如.shtml，.xml 等)，可以单独预览。

系统会在需要显示的过程中动态加载模板文件，并编译该模板文件，生成一个模板缓存文件，下次会自动检测该模板是否变更过，如果在缓存有效期之内并且模板没有改动过的话，系统就不会再次重新编译模板，而是直接读取编译过的缓存模板文件，编译过的缓存模板文件一旦包含之后就可以直接显示变量的值。所以，缓存模板中的显示内容依然是动态的（除了个别系统特殊变量的输出采用静态输出，因为其值比较稳定），而并不是一个静态的输出，如果您的应用需要采用静态文件输出，请设置启用静态文件功能，系统会读取静态文件输出，并且还可以设置静态文件的有效期。

2.5.13 Ajax 支持

ThinkPHP 内置 ThinkAjax 类库，对 Ajax 提供良好支持。支持 HTML 事件绑定、表单提交、附件上传和定时执行等 Ajax 操作。支持 JSON 和 XML 方式返回客户端，当然，您也一样可以使用和扩展任何其他 Ajax 类库来进行 Ajax 操作。

2.5.14 语言支持

系统对国际化的支持比较好，支持语言包功能，您可以定制你需要的语言提示信息。系统在启动应用的时候会自动检测当前用户的浏览器接受的语言，并会尝试在语言文件目录下面寻找该语言包定义文件，如果不存在则会采用默认的语言文件。在检查到正确的语言后，系统会调用 `setlocale` 方法设置本地化支持（包括对时间、数字和货币的本地化支持）。

系统默认语言是简体中文，语言文件的定义支持包括 XML、数组、常量定义在内的多种方式，您可以方便地扩展其它语言包文件，每个项目有自己独立的语言包目录和文件，还可以定义每个模块的语言包。

2.5.15 自动编码

UTF-8 的支持和自动输出编码转换的实现让页面表现更加灵活，框架的文件全部采用 UTF-8 编码格式（为了更好的支持扩展，您以后的应用组件也应该遵循该规范），您可以配置输出的页面编码格式，如 gb2312 等（默认采用 UTF-8 输出）。系统根据配置文件中设置的编码格式自动对页面进行编码转换，支持 `iconv` 和 `mb_string` 两种方式，为了提高效率，如果系统的模板编码设置和输出编码设置项相同，则不会进行编码转换。ThinkPHP 可以设置模板编码、输出编码和数据库编码，并自动完成转换工作，

让你的应用不再受编码的苦恼。

2.5.16 表单处理

ThinkPHP 支持表单数据提交的自动校验和自动完成，让你的表单提交、验证和数据处理轻而易举，不再担心复杂的表单处理。

2.5.17 权限认证

ThinkPHP 框架内置了基于 RBAC 方式的权限认证机制，并且通过 ThinkPHP 框架的管理后台可以方便地进行授权节点（包括项目、模块和操作）的创建和授权操作，以及用户组的分配。

管理图示：



并且，新版的权限模块支持记录级别的权限控制，可以分别对某个数据表进行权限控制。

2.5.18 协作开发

ThinkPHP 框架以项目为目录管理，而且可以把系统目录和网站目录分离，完全可以做到更安全的部署应用，您可以把 ThinkPHP 系统目录放到非 WEB 访问目录下面，以保障应用的代码安全。新版类库的

设计更加满足企业级应用开发中的协作和分布式开发的需要。ThinkPHP 支持分布式的开发和团队合作开发，体现在可以实现项目、组件、模块的分布开发，对于调试工作没有影响，最后项目整体完成后再统一集成。每个项目包是一个独立的目录，因此，协作和集成都很容易完成。每台开发服务器上面只要有 ThinkPHP 框架本身的基类库和框架文件就可以进行应用开发和测试工作。

基于 ThinkPHP 框架进行应用开发，开发人员只需要关注应用类库，所有的后台编码全部都集中（并且大部分情况下只有）在应用类库上面，在进行测试的时候需要模板文件的配合。

同时，ThinkPHP 框架还为同时并发多个项目提供了良好的支持，主要表现在：

每个项目分配不同的入口文件，容易部署；

每个项目有单独的配置文件，相互独立，互不影响；

每个项目的缓存文件和静态文件相互独立，更改和调试互不影响；

每个项目可以单独定制各自的语言包，不会冲突；

每个项目有独立的类库目录，分工协作更加方便；

每个项目的系统日志有单独目录，互不影响；

2.5.19 异常处理

ThinkPHP 对异常处理的支持是完全的，无论是在 PHP4 或者 PHP5 环境下，您都可以使用 `Throw_exception` 方法来抛出异常（默认抛出 `ThinkException` 异常，你可以指定抛出的异常类型），框架集成了 `AppException` 方法和 `AppError` 方法来处理异常和错误，如果设置为非调试模式的话，抛出异常和错误的结果是定向到一个指定的错误页面，对于用户来说更友好些。

鉴于 PHP4 对于异常捕获的能力不够，所以在 PHP4 下面的异常捕获实现采用 `halt` 错误信息的方式来模拟，但是对于最终的效果是一样的，系统抛出异常后的页面显示信息 PHP5 和 PHP4 下效果一致。只是因为 PHP4 下面无法使用 `Try` 和 `Catch` 来自动捕获异常，需要手动判断抛出异常类。

系统的 `ThinkException` 类是所有异常类的基础类，其它模块的异常处理类应该在该基础类上扩展。默认 `Throw_exception` 方法只能抛出系统 `ThinkException` 异常，如果您要使用 `Throw_exception` 方法来抛出自己扩展的异常处理，请添加 `$type` 参数来指定需要抛出的异常类型，如：`throw_exception('用户信息错误','UserException')`。

2.5.20 系统日志

实现了简单的日志记录类, 通过 `Log::Write($errorStr)` 方法来记录系统日志, 包括系统异常和错误信息, 以及 SQL 记录, 日志文件分别对应为 `SystemErr.log`、`SystemOut.log` 和 `SystemSql.log`。您可以随时查看日志文件。

`SystemErr.log` 主要用于记录系统异常, 通常为抛出异常或者捕获严重错误后自动记录

`SystemOut.log` 主要用于调试信息和页面的一些非严重错误记录, 调试信息一般为 `systemOut` 方法写入。

`SystemSql.log` 主要是用于记录执行过程中的 SQL 语句和执行时间, 便于进行分析和优化。

在系统的调试模式中, 系统的所有异常和错误都会记录到系统日志中, 在应用实施后, 您可以关闭调试模式, 这样系统就不会自动完成日志记录, 除非你自己触发日志写入。

系统对项目单独记录日志, 所以查看的时候请注意定位到某个项目目录下。

如果您的应用组件需要记录特殊的日志, 也可以调用 (或者扩展) 该方法来完成。

2.5.21 系统调试

为了更好的适应开发过程的调试需要, **ThinkPHP** 增强了调试模式, 增加了项目的调试模式配置文件, 正式部署只需要关闭调试模式即可, 不需要更改其它项目参数。并且支持调试模式使用不同的数据库配置信息。除了本身可以借助一些开发工具进行调试外, **ThinkPHP** 还提供了一些调试手段, 包括日志分析、运行调试和页面 `Trace` 调试, 也提供了一些内置的调试函数和类库。而且, 在调试模式下面增加了页面 `Trace` 功能, 为实施页面监控和调试输出提供了必要的手段。在 `Trace` 信息里面可以清楚看到当前页面的请求信息、相关变量、SQL 语句、错误提示, 并且可以加入自定义的 `Trace` 信息。

2.5.22 项目部署

ThinkPHP 框架为企业级的部署和开发提供了方便。

首先, **ThinkPHP** 框架无需安装, 仅仅需要上传就可以了, **ThinkPHP** 系统目录可以传到服务器的任何位置, 例如 `C:/ThinkPHP` 或者 `/User/Local/ThinkPHP`, `WebApps` 目录传到你访问的 `WEB` 目录下面, 由于 **ThinkPHP** 系统的架构特点, 如果你有多个网站应用系统基于 **ThinkPHP** 框架构建, 那么你不需设置多个网站目录, 因为每个网站应用都采用单一入口 (也就是说一个网站就只有一个入口文件, 模板目录也是公用的, 而且并不会相互影响), 当然, 由于其它原因 (如由于不同域名的关系或者要放到

不同的服务器)您仍然可以分开多个目录存放,这个并没有影响。

其次,开发模式和部署模式的切换仅仅需要关闭调试模式就可以了,其它配置无需修改,让你的应用部署更加简单。框架是简单的,但是可以构建复杂的应用。

2.5.23 插件支持

ThinkPHP 可以满足大多数的应用需要,并且还可以支持插件方式对系统进行功能扩展。ThinkPHP 采用和 WordPress 类似的插件机制,通过插件功能,你可以实现替换模板引擎、增加数据库支持、实现 URL 路由控制、实现安全过滤和输出过滤、增加外挂模块和操作,而不用修改框架核心文件,轻松扩展,不再担心不断变化的复杂应用需求,同时也免去升级的不便。

2.6 目录结构

ThinkPHP 的目录结构非常清晰和容易部署。大致的目录结构如下,以项目为基础进行部署。

├──ThinkPHP 框架系统目录

| └─ThinkPHP.php 系统公共文件

| └─Common 公共文件目录

| └─Tpl 框架系统模版目录

| └─Lang 系统语言包目录

| └─Plugins 公共插件目录

| └─Lib 系统基类库目录

| └─Think 系统运行库(必须)

| └─Com 扩展类库包(非必须)

| └─ORG 扩展类库包(非必须)

|

└──App App 项目目录

| └─index.php 项目入口文件

| └─Cache 模版缓存目录

| └─Common 公共文件目录(非必须)

- | └ Conf 项目配置目录
- | └ Data 项目数据目录
- | └ Html 静态文件目录（非必须）
- | └ Plugins 插件目录（非必须）
- | └ Tpl 模版文件目录
- | └ Lang 语言包目录（非必须）
- | └ Logs 日志文件目录
- | └ Temp 数据缓存目录
- | └ Uploads 上传文件目录（非必须）
- | └ Lib 应用类库目录
 - | └ Action 控制器（模块）类目录
 - | └ Model Model 类文件目录
 - | ... 下面的应用目录可根据需要选择和定义
 - | └ Exception 异常类库目录
 - | └ Common 公共应用类目录
 - | └ Help 助手类目录
- |
- | ...更多项目目录（和 App 目录类似，每个项目采用独立目录，便于部署）
- |
- | └ Public 网站公共目录（多项目公用）
- | └ Js JS 类库目录（建议）
- | └ Images 公共图像目录（建议）
- | └ Uploads 公共上传目录（建议）

ThinkPHP 框架除了模板目录和网站入口文件必须放到 **WEB** 目录下之外，其它所有框架的文件和目录可以单独存放，不受限制，您需要做的仅仅是在首页文件中指定 ThinkPHP 框架的包含目录，我们建议您如果可能的话把 ThinkPHP 框架的目录包放到其它网站不能访问的目录下面，以保障应用的安全性。项目独立目录，方便部署和团队开发。每个项目有自身的配置文件、语言文件、插件文件和日志文件。如果在类 Linux 环境下面部署，需要对以下目录设置可写权限（这些目录仅仅针对项目目录，系统目

录无需设置任何可写权限，因为每个项目的模版缓存和数据缓存，以及日志文件都是独立的）。

项目目录下面的 **Cache**（模版缓存目录）、**Temp**（数据缓存目录）、**Conf**（项目配置目录，写入权限用于自动生成配置缓存和插件缓存文件）、**Logs**（日志文件目录）、如果设置了 **Uploads** 上传目录和 **Data** 数据目录的话也必须设置为可写。另外，如果设置了 **Public** 目录下面的 **Uploads** 目录作为公共上传目录，也需要设置可写权限。通常的设置都是设置目录属性为 **777**。



1.0.0 正式版本需要生成核心缓存和项目编译缓存文件，所以要求项目目录属性为可写。

一定要注意在 Linux 环境下面的文件大小写问题，否则会导致文件加载错误。

2.7 环境要求

ThinkPHP 可以支持 WIN/Unix 服务器环境，支持 PHP4.3.0 以上版本，完全兼容 PHP5。

支持 Mysql、MsSQL、Sqlite 等多种数据库，在 PHP5 下面可以支持 PDO，ThinkPHP 框架本身没有什么特别模块要求，具体的应用系统运行环境要求视开发所涉及的模块。

2.8 获取 ThinkPHP

获取 ThinkPHP 的方式很多，官方网站（<http://thinkphp.cn>）是最好的下载和文档获取来源。

下面提供 ThinkPHP 的更多相关资源：

SVN 地址：<http://thinkphp.googlecode.com/svn/trunk>

Google 项目地址：<http://code.google.com/p/thinkphp/>

SF 项目地址：<http://sourceforge.net/projects/thinkphp>

SF 官方站点：<http://thinkphp.sourceforge.net/>

第3部分 构建应用

3.1 示例一：Hello，ThinkPHP

3.1.1 创建项目

在进行项目开发之前，首先按照 ThinkPHP 的应用目录结构来创建好项目的目录架构。使用第三方开发的 TP 项目生成器(参考官方网站)可以直接生成项目目录，或者直接拷贝一个空的项目目录结构过来，然后再增加文件。针对 Hello 项目，只需要创建下面的目录结构就可以了：

```
Hello 项目目录
|----- index.php 项目入口文件
|----- Cache 模版缓存目录 (*)
|----- Conf 项目配置目录 (*)
|----- Tpl 项目模版文件目录
|----- Logs 项目日志文件目录 (*)
|----- Temp 数据缓存目录 (*)
|----- Lib 应用类库目录
|----- Action 控制器（模块）类目录
```

其中，(*) 的目录是需要设置写权限的。实际上，我们只需要创建好项目目录下面的 Conf、Lib、Tpl 目录就可以工作了，其它目录会在需要的时候自动生成。

3.1.2 入口文件

ThinkPHP 框架采用单一入口模式部署项目，所以，每个项目至少需要一个入口文件来执行项目。入口文件并不一定是指 index.php 文件，虽然通常都这么设置。一个项目的入口文件可以放置在 WEB 目录下面任何需要的位置，区别仅仅在于读取的 ThinkPHP 系统目录的路径就不同而已。

对于一个项目来说，入口文件其实没有做任何与项目本身相关的逻辑或者处理，而仅仅是实例化一个 ThinkPHP 项目实例而已，所以，建议不要在入口文件里面做过多的处理工作，以避免额外的维护工作量。一个入口文件的标准写法是（假设该入口文件是位于项目目录下面的 index.php 文件）：

```
// 常量定义

define('THINK_PATH', './ThinkPHP');

define('APP_NAME', 'Hello');

define('APP_PATH', '.');

// 加载框架公共入口文件

require(THINK_PATH."/ThinkPHP.php");

// 实例化一个网站应用实例

$app = new App();

// 执行应用程序

$app->run();
```

如果你把 ThinkPHP 所在路径添加到了 PHP 的 `include_path` 中，那么 `THINK_PATH` 可以无需定义。

`APP_NAME` 如果不定义，则默认为入口文件名。

无论你的项目有多么复杂或者多么简单，入口文件都是不变的，事实上，入口文件仅仅是实例化一个 WEB 应用而已，但是却必不可少。你可以为你的项目在多个位置定义多个入口文件，这并不违反框架的规则，而且也不影响项目的运行。

😊 再次强调，不要在项目入口文件里面添加任何逻辑处理代码。

3.1.3 项目配置

定义好了应用的入口文件后，我们可能需要给项目定义一个配置文件，但是对于简单的应用来说，我们甚至无需任何配置文件就可以执行。在这个应用里面，我们没有定义任何配置文件。

3.1.4 模块定义

在 ThinkPHP 框架中，访问是基于项目、模块和操作的概念，所以，在定义逻辑之前我们需要规划好应用项目的模块和操作，一个完整的 URL 访问地址应该是类似于：

`http://<serverName>/[<appName>]/<项目入口文件>/<moduleName>/<actionName>`

如果没有指定 `moduleName` 和 `actionName`，则默认采用 `Index` 模块和 `index` 操作（可以在项目配置文件中更改，关于项目的更加详细的配置会在以后说明）

针对这个简单的应用，通常我们使用 Index 模块的 index 操作就可以了。

因此，我们需要在 Lib 目录下面的 Action 子目录下面创建一个 IndexAction.class.php 文件来进行控制器定义。该文件就是 Index 的模块类定义文件。

打开 IndexAction.class.php 文件后添加类的定义代码

```
class IndexAction extends Action {  
  
} //end class
```

IndexAction 类通常都继承 Action 基础类，因为里面集成了模版操作和通用操作定义。

模块类里面的每个方法都可以表示一个操作，如果在当前类没有该方法，系统会自动在父类里面查找，如果依然没有找到，就会抛出异常。

下来，我们给 IndexAction 类添加 index 操作方法：

```
// 定义 index 操作  
  
Public function index() {  
  
    echo 'Hello,ThinkPHP!';  
  
}
```

操作方法的定义很简单，方法名就是操作名，并且必须是 Public 类型，否则操作无效。操作方法通常无需任何参数，因为操作的参数都是通过 GET、POST、SESSION 或者 Cookie 方式传入的，不会显式传入任何参数到操作方法里面。如果要增加参数，通常都应该使用默认参数。

这个时候，如果我们在 URL 里面执行<http://localhost/Hello/index.php>，就会看到输出了

Hello,ThinkPHP!

由于这个操作相当简单，以致于我们可以不需要使用模版，直接用 echo 来输出。如果需要使用模版功能，那么上面的操作方法就改成：

```
// 定义 index 操作  
  
Public function index()  
{  
  
    $this->assign('var','Hello,PHP!'); // 对模板变量赋值  
  
    $this->display(); // 渲染模板输出  
  
}
```

当我们再次企图通过 URL <http://localhost/Hello/index.php> 来访问的时候，我们发现系统提示了错误：模板不存在，这很正常，因为我们还没有给 index 操作定义模板。

3.1.5 模板输出

接下来，我们定义模版文件，模版文件位于 `Tpl` 目录下面，因为 **ThinkPHP** 支持多模版的切换，默认的模板主题是 `default`，所以我们先在 `Tpl` 目录下面创建一个 `default` 子目录，并且在 `default` 目录下面，我们用模块名称创建一个 `Index` 目录（注意需要和模块名一致，在 `linux` 环境下面注意大小写），在该目录下面创建一个 `index.html` 模版文件（文件名和操作名称对应，在 `Linux` 环境下面注意大小写）。模版文件中我们使用了标签输出变量，

```
<html><body>

{$var}

</body></html>
```

这里我们使用了 **ThinkPHP** 内置的模板引擎来输出，有关模板引擎的功能在后面会有更加详细的描述。

3.1.6 运行示例

保存后，我们直接在浏览器里面输入 `URL` 执行应用

`http://localhost/Hello/index.php` 就可以看到输出了 `Hello,ThinkPHP!`

使用下面的 `URL` 地址看到的是同样的结果

`http://localhost/Hello/index.php/Index/`

`http://localhost/Hello/index.php/Index/index/`

也许大家认为为了输出一个 `Hello,PHP` 要做这么多的工作，不值得，呵呵～其实 **ThinkPHP** 框架不管对于简单还是相当复杂的应用都是采用这样的步骤。我们可以看到，作为一个类似的简单应用，我们只需要创建一个 `Index` 控制器类，并且最多定义一个模板文件就完成了整个项目，而我们却可以通过框架底层实现了 `SEO` 功能和众多内置的功能。

下面是完整的控制器代码

```
class IndexAction extends Action {

    // 定义 index 操作

    function index()

    {

        $this->assign('var','Hello,PHP!');

        $this->display();

    }

}
```

```
}  
} //end class
```

3.2 示例二：Blog

示例二描述了如何使用 ThinkPHP 框架进行一个 Blog 应用的快速开发，这个示例我们涉及到的功能要比前面的例子多一些，但仍然是一个非常简单的示例，通过这个示例，我们可以了解如何定义配置文件、创建模型以及进行页面调试。一个应用的入口文件和控制器是必须的，其它过程都是可以根据项目需求可选的。

3.2.1 目录结构

我们按照下面的目录结构来创建 Blog 应用的目录，假设 ThinkPHP 系统目录和 Blog 目录在同级。

Blog 项目目录

- |----- index.php 项目入口文件
- |----- Cache 模版缓存目录 (*)
- |----- Common 公共文件目录 (可选)
- |----- Conf 项目配置目录 (*)
- |----- Data 项目数据目录 (*)
- |----- Html 静态文件目录 (*) (可选)
- |----- Tpl 项目模版文件目录
- |----- Logs 项目日志文件目录 (*)
- |----- Temp 数据缓存目录 (*)
- |----- Lib 应用类库目录
 - |----- Action 控制器 (模块) 类目录
 - |----- Model MModel 类文件目录

3.2.2 入口文件

和 Hello 应用一样，我们也需要给 Blog 项目创建一个入口文件。虽然应用不同，但是入口文件几乎是一模一样的，区别仅仅在于 APP_NAME 不同。另外，我们在 Blog 应用里面把入口文件放到网站的根目录下，所以 APP_PATH 有所区别。

```
// 常量定义

define('THINK_PATH', './ThinkPHP');

define('APP_NAME', 'Blog');

define('APP_PATH', './Blog');

// 加载框架公共入口文件

require(THINK_PATH."/ThinkPHP.php");

// 实例化一个网站应用实例

$app = new App();

// 执行应用程序

$app->run();
```

3.2.3 项目配置

我们对 Blog 应用做一些配置的调整，例如：加上数据库连接信息、启用调试模式等，项目配置文件有很多种定义方式（在后面会有关于项目配置的描述），但是在运行过程中会自动生成一个配置缓存文件，为了省事，我们在项目的 Conf 目录下面直接定义配置缓存文件_config.php，内容如下：

```
return array(

    'DEBUG_MODE'=>true,

    'DB_TYPE'=>'mysql',

    'DB_HOST'=>'localhost',

    'DB_NAME'=>'thinkphp',

    'DB_USER'=>'root',

    'DB_PWD'=>'admin',

    'DB_PORT'=>'3306',

    'DB_PREFIX'=>'think_',
```

```
'MODEL_CLASS_SUFFIX'=>',  
  
'DEFAULT_MODULE'=>'Blog',  
  
}
```

注意我们设置了 `MODEL_CLASS_SUFFIX` 配置参数为空，是为了方便我们的模型定义，系统默认的值是 `Model`，也就是说所有的模型类的命名有一个 `Model` 后缀，该设置是为了避免大型应用的类名冲突，因为这个示例比较简单，所以我们设置为空，可以省去命名中的后缀。

`DEFAULT_MODULE` 参数设置了项目的默认模块是 `Blog` 模块，而不是系统默认的 `Index` 模块，这使得我们在输入入口文件的时候，默认执行的是 `Blog` 模块而不是 `Index` 模块。

有些情况下我们还希望开发人员在测试的时候可以使用另外的测试数据库进行数据测试，ThinkPHP 支持给调试模式定义单独的配置文件。在这个例子里面我们增加了调试模式配置文件，并增加相应的测试数据库连接信息，同样，我们也是在项目的 `Conf` 目录下面创建一个 `_debug.php` 文件，内容如下：

```
return array(  
  
    'DB_TYPE'=>'mysql',  
  
    'DB_HOST'=>'localhost',  
  
    'DB_NAME'=>'test',  
  
    'DB_USER'=>'root',  
  
    'DB_PWD'=>'admin',  
  
    'DB_PORT'=>'3306',  
  
    'DB_PREFIX'=>'think_',  
  
}
```

注意，调试模式配置文件，只有当项目配置文件中启用了调试模式才有效。在调试配置文件里面，我们还可以设置在调试模式下面的不同参数。系统存在一个为调试模式下使用的调试配置文件，该文件位于 ThinkPHP 系统的 `Common` 目录下面，名称是 `debug.php`，如果你的项目需要定义额外的调试参数，可以在项目的调试配置文件里面添加。

 需要注意的是，1.*以上版本对 `DB_PREFIX` 参数的设置有所不同，必须带上后面的“_”符号。

3.2.4 模块规划

完成项目配置后，我们就要开始进行应用开发了。在这个 `Blog` 应用中，我们设计了下面的功能：

- ✧ 日志列表功能
- ✧ 日志维护（添加、编辑和删除日志）
- ✧ 类别管理
- ✧ 给日志添加评论
- ✧ 日志阅读和评论计数
- ✧ 生成静态日志查看

针对上面的功能，我们规划了下面几个模块和操作：

Blog 模块

- 列表操作 `index`
- 添加操作 `add` 和 `insert`
- 编辑操作 `edit` 和 `update`
- 删除操作 `delete`
- 评论操作 `comment`

Category 模块

- 列表操作 `index`
- 添加操作 `add` 和 `insert`
- 编辑操作 `edit` 和 `update`
- 删除操作 `delete`

其中，列表 `index`、添加 `add`、编辑 `edit` 操作需要模板页面。

3.2.5 模型定义

我们首先来定义各自的数据模型，在这个应用中我们使用了三个数据表，分别是 `blog`、`category` 和 `comment`，数据表的前缀是 `think`，DDL 如下：（示例使用的是 MySQL 数据库，如果是其它数据库，请做相应修改）

```
CREATE TABLE `think_blog` (  
    `id` int(11) unsigned NOT NULL auto_increment,  
    `category_id` smallint(5) unsigned NOT NULL,  
    `title` varchar(255) NOT NULL default "",  
    `content` longtext NOT NULL,
```

```
`create_at` int(11) unsigned NOT NULL default '0',  
`update_at` int(11) unsigned NOT NULL default '0',  
`read_count` mediumint(5) unsigned NOT NULL default '0',  
`comment_count` mediumint(5) unsigned NOT NULL default '0',  
PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;
```

```
CREATE TABLE `think_category` (  
  `id` mediumint(5) unsigned NOT NULL auto_increment,  
  `name` varchar(30) NOT NULL default '',  
  `title` varchar(50) NOT NULL default '',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;
```

```
CREATE TABLE `think_comment` (  
  `id` mediumint(5) unsigned NOT NULL auto_increment,  
  `blog_id` int(11) unsigned NOT NULL default '0',  
  `author` varchar(50) NOT NULL default '',  
  `ip` varchar(25) NOT NULL default '',  
  `content` text NOT NULL,  
  `create_at` int(11) unsigned NOT NULL default '0',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据表的主键我们使用了 id，并且设置了自动增长。

接下来，我们就创建需要的模型。

Blog 模型：位于 Lib\Model\Blog.class.php

```
Class Blog extends Model{  
  
}
```

Category 模型：位于 Lib\Model\Category.class.php

```
Class Category extends Model{  
  
}
```

Comment 模型：位于 Lib\Model\Comment.class.php

```
Class Comment extends Model{  
  
}
```

通常情况下在模型类里面我们什么都不需要设置，系统会在第一次运行的时候自动获取数据库的字段信息，并缓存起来，而且可以自动判断主键的名称，就这么简单。

3.2.6 辅助定义

表单验证是开发过程中经常需要使用的功能，如果你需要使用系统内置的自动验证和自动处理功能，可以在模型里面定义相关的规则。

例如，我们需要对 Blog 数据表的 title 和 content 字段进行验证，可以增加下面的规则：

```
var $_validate = array(  
  
    array('title','require','标题必须！'),  
  
    array('content','require','内容必须'),  
  
);
```

默认是使用正则表达式进行验证。

另外，我们需要在发表评论的时候，自动记录用户的 ip，这就需要定义自动处理规则，定义如下：

```
var $_auto = array(  
  
    array('ip', get_client_ip, 'ADD', 'function'),  
  
);
```

表示要在新增评论的时候自动对 ip 字段调用 get_client_ip 函数。

更多的关于自动验证和自动处理的说明请参考后面的章节。

3.2.7 定义视图

考虑到我们在显示 Blog 列表的时候可以使用视图来方便数据的查询操作，ThinkPHP 新版增加了视图模型功能，可以模拟数据库中的视图功能，并且该功能并不需要数据库的支持。下面我们定义一个视图模型来专门读取 blog 列表。视图模型文件和其它模型类文件一样，都放置项目类库目录的 Model

目录下面，这里的文件名应该是 `BlogView.class.php`。

```
Class BlogView extends Model{

    protected $viewModel = true;

    // 定义视图中的字段

    protected $viewFields = array(

        'Blog'=>array('id','title','create_at','category_id','read_count','comment_count'),

        'Category'=>array('title'=>'category'),

    );

    // 定义基础查询条件

    protected $viewCondition = array("Blog.category_id" => array('eqf',"Category.id"));

    // 定义视图主键名称

    Public function getPk() {

        return 'id';

    }

}
```

这是一个虚拟的视图模型，并不需要数据库支持视图功能。

`viewModel` 属性设置为 `true` 表示 `BlogView` 模型是一个视图模型，并且我们定义了该视图关联的数据模型 `Blog` 和 `Category`，并且对其中的一些字段做了映射（可以不是全部的字段），并且定义了关联查询的条件 `$viewCondition` 来避免查询过多的重复数据。事实上，系统还会自动给视图模型的查询加上 `group` 功能，通常是对视图的主键字段进行 `group`。

一般情况下，视图模型定义首先需要定义数据模型，就好比定义数据库的视图需要首先定义其中的数据表一样。但是 `ThinkPHP` 允许视图模型中的数据模型没有定义单独的模型类。也就是说 `BlogView` 类如果没有定义 `Blog` 模型类一样可以工作。

3.2.8 控制器定义

完成了模型类的定义后，我们就来定义 `Action` 控制器。

根据我们先前规划的模块和操作，我们需要增加两个 `Action` 控制器就可以了

```
Class BlogAction extends Action{

}
```

对于 Blog 模块来说,由于 Blog 名称和数据表的名称一致,所以我们可以直接使用系统的内置列表、查看、增加、编辑和删除操作方法,而无需自己定义。不过因为我们定义了 BlogView 视图模型,所以,这里我们可以自己来添加列表操作。

```
Public function index(){  
  
    $Blog = D("BlogView");  
  
    $list =    $Blog->findAll();  
  
    $this->assign("list",$list);  
  
}
```

由于是简单的例子,我们暂时不加入分页功能,而是获取全部的日志列表。以上的视图模型操作类似于定义了一个关联关系,在获取日志的时候同时获取关联的类型名称。

如果你有特殊的功能需要在操作里面实现,可以使用前置方法或者后置方法。在这个应用中,因为我们需要对阅读计数,并且要读取当前 blog 的评论,所以需要增加查看操作的前置方法 `_before_read`,例如:

```
Public Function _before_read(){  
  
    $Blog = D("Blog"); // 创建 Blog 对象实例  
  
    $id = $_GET['id']; // 当前查看的日志  
  
    $Blog->setInc('read_count',"id=".$id); // 阅读计数  
  
    // 获取评论  
  
    $Comment = D("Comment");  
  
    $comments = $Comment->findAll("blog_id=".$id);  
  
    // 模板变量赋值  
  
    $this->assign("comments",$comments);  
  
}
```

然后,我们还要增加一个评论操作,来写入评论数据。

```
Public Function comment(){  
  
    $Comment = D("Comment"); // 实例化 Comment 对象  
  
    $Comment->create(); // 使用表单提交的数据创建评论数据对象  
  
    $result = $Comment->add(); // 写入评论数据到数据库
```

```
If ($result){  
    // 如果评论写入成功 更新 Blog 的评论计数  
    $Blog = D("Blog");  
    $Blog->setInc("comment_count","id=".$Comment->blog_id);  
}  
}
```

Category 模块的操作完全属于自动化操作可以完成的范围，所以我们无需定义任何操作方法，直接定义一个空的 **CategoryAction** 控制器类就行了。

```
Class CategoryAction extends Action{  
}
```

或许大家会疑问，为什么我们不需要定义 **CommentAction** 控制器呢，因为在我们的 URL 规划里面，不需要有 **Comment** 模块的存在，只是在 **Blog** 模块里面有写入 **Comment** 的操作，如果我们需要对 **Comment** 有专门的管理，可能就需要定义 **CommentAction** 控制器了。原则上，控制器就表示存在相应的模块，不需要使用的模块就不需要定义控制器。

到目前为止，**Blog** 应用的逻辑部分我们已经全部完成了，你是不是觉得不太可能，连 **Blog** 的增加、编辑、查看和删除操作的逻辑我们都还没有写呢，没错，事实上这些 **CURD** 自动化操作已经有系统内置实现了，你根本无需额外的工作。

可惜的是，到目前为止 **Blog** 应用还无法运行，因为还差一个视图环节没有定义。

3.2.9 模板定义

Blog 应用的模板我们采用系统内置的 **ThinkTemplate** 模板引擎，因为无需任何插件就可以实现模板输出。为了保持页面的统一，我们使用了公共模板，把头部和尾部页面脱离出来。

模板的目录结构如下：

```
|----- Tpl 项目模版文件目录  
|----- default 默认主题目录  
|----- Public 公共模板文件目录  
|----- header.html 头部公共文件  
|----- footer.html 尾部公共文件  
|----- Blog Blog 模块模板文件目录
```

|----- index.html 列表操作模板文件
|----- add.html 新增操作模板文件
|----- edit.html 编辑操作模板文件
|----- Category Category 模块模板文件目录
|----- index.html 列表操作模板文件

下面是具体的模板文件定义

头部公共模板文件

文件名 Public/header.html

```
<html>

<HEAD>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<TITLE>{$title} </TITLE>

</HEAD>

<body >

<div id="header" >ThinkPHP Blog 示例 </div>
```

尾部公共模板文件

文件名 Public/footer.html

```
<div id="footer" >Powered by ThinkPHP {$Think.version} </div>

</BODY>

</HTML>
```

日志列表模板

文件名 Blog/index.html

```
<include file="Public:header" />

<h2>日志列表</h2>

<volist name="list" id="vo">

<div id="blog_{$vo.id}">
```

```
<h2>{$vo:title}</h2>

[ {$vo.create_at|toDate} {$vo:category}] <a href="__URL__/delete/id/{$vo.id}">删除</a>

<div>{$vo.content}</div>

</div>

</volist>

{$page}

<include file="Public:footer" />
```

注意：列表模板文件中的{\$vo:category} 其实是属于视图模型中的字段，而并非 Blog 数据表中的字段，这个就是视图模型的优势所在，免去了关联查询的定义和查询过程。

日志新增模板

文件名 Blog/add.html

```
<include file="Public:header" />

<h2>新增日志</h2>

<FORM METHOD=POST ACTION="__URL__/insert/">

  标题: <INPUT TYPE="text" NAME="title" value="">

  内容: <TEXTAREA NAME="content" ROWS="8" COLS="45"></TEXTAREA>

   <INPUT TYPE="text" NAME="verify">

  <INPUT TYPE="submit" value="保存日志">

</FORM>

<include file="Public:footer" />
```

日志查看模板

文件名 Blog/read.html

```
<include file="Public:header" />

<h2>查看日志</h2>

<div id="blog_{$vo.id}">

  <h2>{$vo.title}</h2>

  [ {$vo.create_at|toDate} {$vo:category}]
```



```
<div>{$vo.content}</div>

</div>

<div id="comments">

<volist name="comments" id="comment">

{$comment.create_at|toDate}

{$comment.content}

</volist>

<FORM METHOD=POST ACTION="__URL__/comment/">

<INPUT TYPE="hidden" NAME="blog_id" value="{ $vo.id}">

姓名: <INPUT TYPE="text" NAME="author">

评论: <TEXTAREA NAME="content" ROWS="8" COLS="45"></TEXTAREA>

 <INPUT TYPE="text" NAME="verify">

<INPUT TYPE="submit" value="发表评论">

</FORM>

</div>

<include file="Public:footer" />
```

日志编辑模板

文件名 Blog/edit.html

```
<include file="Public:header" />

<h2>编辑日志</h2>

<FORM METHOD=POST ACTION="__URL__/update/">

<INPUT TYPE="hidden" NAME="id" value="{ $vo.id}">

标题: <INPUT TYPE="text" NAME="title" value="{ $vo.title}">

内容: <TEXTAREA NAME="content" ROWS="8" COLS="45">{$vo.content}</TEXTAREA>

 <INPUT TYPE="text" NAME="verify">

<INPUT TYPE="submit" value="保存日志">

</FORM>

<include file="Public:footer" />
```

类别列表模板

文件名 Category/index.html

```
<include file="Public:header" />

<html:import type="js" file="Js.prototype" />
<html:import type="js" file="Js.mootools" />
<html:import type="js" file="Js.Ajax.ThinkAjax" />
<SCRIPT LANGUAGE="JavaScript">

<!--
Function add(){
ThinkAjax.send('__URL__/insert','ajax=1&title='+$F('title'),addComplete);
}

Function addComplete(data,status){
    if (status==1){
        $('categoryList').innerHTML += data+'<br/>';
    }
}

//-->
</SCRIPT>

<h2>类别列表</h2>

    新增类别： <INPUT TYPE="hidden" id="title" NAME="title" value=""> <INPUT TYPE="button"
onclick="add()" value="新增">

    <div id="categoryList">

        <volist name="list" id="vo">

            {$vo:title}<br/>

        </volist>

    </div>

<include file="Public:footer" />
```

在类别列表中，我们使用了 Ajax 方式增加类别

3.2.10 运行示例

模板定义完成后，我们直接在浏览器里面输入 URL 来检查 Blog 应用

http://localhost/Blog/index.php



3.2.11 错误调试

运行 Blog 示例后，我们可以在页面下面看到 Trace 信息，这是由于我们启用了调试模式，当前看到的是 ThinkPHP 内置的页面 Trace 输出信息，我们可以查看当前页面的请求信息、执行时间和错误提示。

页面Trace信息

```
当前页面 : /thinkvms/admin/index.php/Girl
通信协议 : HTTP/1.1
请求方法 : GET
请求时间 : 2007-09-01 20:33:40
压缩编码 : gzip, deflate
请求语言 : zh-cn
用户代理 : Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; Mozilla/4.0 (compatible; MSIE 6.0
SESSIONID : 4e2cc45a46d9079712021
hello1 : status
hello2 : ddddd
运行时间 : Process: 0.203s ( Load:0.047s Init:0.035s Exec:0.093s Template:0.028s ) | DB :9 queries
```

在开发过程中，我们可以自己来定制需要输出的 Trace 信息。

3.2.12 应用部署

应用开发完成之后的部署工作非常简单，基本上只需要上传然后适当做些配置修改就可以了。把整个 ThinkPHP 系统目录（包含了集成后的应用类库）拷贝或者上传到指定的目录，然后把项目首页文件和模板目录整个上传到 WEB 访问的指定目录下面（注意相关目录的权限）。

⚠ 注意：确保您的项目开发测试环境和应用部署环境一致，因为系统框架本身兼容没有问题，但是如果您的应用类库开发过程中没有考虑系统的兼容性问题，那么部署后的项目可能和测试环境有所区别。

并且关闭调试模式，把 `DEBUG_MODE` 置为 `False`

ThinkPHP 会自动切换到部署模式来运行应用，这个时候生效的就是项目配置文件。

第4部分 开发指南

通过以上的两个简单应用的构建，我们已经大致了解了 ThinkPHP 开发的过程，下面的部分，我们会比较全面的介绍 ThinkPHP 的类库和功能模块。

4.1 类库和函数库

4.1.1 系统基类库

ThinkPHP 框架通过基类库的概念把所有系统类库都集成在一起，包括 ThinkPHP 的核心类库。

目前的基类库分成 Think 核心类库、ORG 扩展类库、Com 扩展类库，在这主要介绍的是核心基类库，核心基类库的作用是完成框架的通用性开发而必须的基础类和常用工具类等，包含有：

Think.Core 核心类库包

Think.Db 数据库类库包

Think.Util 系统工具类库包

Think.Template 内置模板引擎类库包

Think.Exception 异常处理类库包

在后面的章节我们会陆续讲解或者涉及这些核心类库的使用。

4.1.2 函数库

ThinkPHP 内置了一个系统公共函数库，提供了一些系统需要的函数，系统函数库位于 ThinkPHP 的 Common 目录下面，名称为 functions.php。

每个项目可以定义自己的函数库，位于项目的 Common 目录下面的 common.php 文件。

如果项目在 Common 目录下面有定义自己的 common.php 文件，框架会在初始化的时候自动导入，而无需自己导入。

4.1.3 系统定义文件

在系统的 Common 目录下面有一个 defines.php 文件，是系统的定义文件，通常情况下不需要做任何修改。包含了一些系统常量定义和版本定义。

4.1.4 匹配导入

Import 方法是 ThinkPHP 内建的类库和文件导入方法，提供了方便和灵活的文件导入机制。例如：

```
Import("Think.util.ListIterator");
```

```
Import("App.Model.UserModel");
```

⚠ 注意：在 Unix 或者 Linux 主机下面是区别大小写的，所以在使用 **import** 方法或者 **using** 方法的时候要注意目录名和类库名称的大小写，否则会引入文件失败。

由于命名空间设计的局限性，在命名目录名称的时候不能使用 “.” 符号，否则会转换成 “/” 符号导致文件引入失败。

Import 方法具有缓存和检测机制，相同的文件不会重复导入，如果发现导入了不同的位置下面的同名类库文件，系统会提示冲突，例如：

```
Import("Think.Util.Array");
```

```
Import("App.Util.Array");
```

上面的情况导入会产生引入两个同名的 **Array.class.php** 类，即使实际上的类名可能不存在冲突，但是按照 **ThinkPHP** 的规范，类名和文件名是一致的，所以系统会抛出类名冲突的异常，并终止执行。

除了正常的导入操作为，还支持模式匹配导入和多文件导入。例如：

导入 **Think.Util** 目录下面的所有类文件

```
Import('Think.Util.*');
```

```
Import("App.Model.*");
```

注意：使用子目录引入的方式，如果目录下面文件较多会给系统带来较大的目录遍历开销。

对于 **Import** 方法，系统会自动识别导入类库文件的位置，**ThinkPHP** 的约定是 **Think**、**ORG**、**Com** 包的导入以系统基类库为相对起始目录，否则就认为是项目应用类库为起始目录，如果是其它情况的导入，需要指定 **baseUrl** 参数，也就是 **import** 方法的第二个参数。例如，要导入当前文件所在目录下面的 **RBAC/AccessDecisionManager.class.php** 文件：

```
Import('RBAC. AccessDecisionManager',dirname(__FILE__));
```

Import 方法具有智能的模式匹配导入机制，例如下面的例子使用了更高级的匹配模式导入：

```
Import('Think.*.Array');
```

```
Import('ORG.Util.Array*');
```

```
Import('ORG.Util.*Map');
```

```
Import("App.Util.?Tool");
```

ThinkPHP 的类文件命名规则是以.class.php 作为后缀, 所以默认的导入只会匹配.class.php 为后缀的文件, 如果你使用的第三方类库文件没有使用.class.php 作为类文件命名规范, 那么需要指定后缀, 例如, 导入 Com.Zend.Search 目录下面的所有类文件, 但不包括子目录, 并且这些类库文件的后缀是.php, 我们就需要指定 import 方法的第三个参数 ext:

```
Import('Com.Zend.Search.*','','.php');
```

为了方便类似的导入机制, 系统给第三方类库专门指定了一个 vendor 方法, 该方法默认的导入类文件的后缀是.php, 并且起始路径是 ThinkPHP 系统目录下面的 Vendor 目录。其它使用方法相同。

4.1.5 导入第三方类库

我们知道 ThinkPHP 的基类库都是以.class.php 为后缀的, 这是系统内置的一个约定, 当然也可以通过 import 的参数来控制, 为了更加方便引入其他框架和系统的类库, 系统增加了导入第三方类库的功能, 第三方类库统一放置在系统的 Vendor 目录下面, 并且使用 vendor 方法导入, 其参数和 import 方法是一致的, 只是默认的值有针对变化。

例如, 我们把 Zend 的 Filter\Dir.php 放到 Vendor 目录下面, 这个时候 Dir 文件的路径就是 Vendor\Zend\Filter\Dir.php, 我们使用 vendor 方法导入就是:

```
Vendor('Zend.Filter.Dir');
```

4.1.6 关于约定

在很多情况下, 你看到的关于 ThinkPHP 的一些规范只是系统内置的一些惯例和约定, 但是这些通常不是一成不变的, 在有些情况下, 你可以通过配置文件更改这些约定。但是遵守这些约定是 ThinkPHP 应用开发的良好基础, 能保证最大程度的兼容和移植。

在使用 ThinkPHP 框架进行应用开发的过程中, 我们做了如下的一些约定:

- 所有类库文件必须使用.class.php 作为文件后缀, 并且类名和文件名保持一致;
- 控制器的类名以 Action 为后缀 (可以配置)
- 模型的类名以 Model 为后缀 (可以配置)
- 数据库表名全部采用小写 (可定义)

4.1.7 如何配置

ThinkPHP 的项目配置一直以灵活著称，尤其在 0.9.8 版本发布后，又有了进一步的提高和完善。

ThinkPHP 在项目配置上面创造了自己独有的分层配置模式，其配置层次体现在：

惯例配置--> 项目配置--> 模块配置--> 操作（动态）配置

// 优先顺序从右到左（在没有生效的前提下）

1、多格式支持

ThinkPHP 框架对配置文件的定义支持很多格式，目前支持的有 PHP 数组、常量定义文件、INI 配置文件、XML 配置文件和 PHP 对象，以及数据库存储配置。项目配置文件统一使用 `_config.php` 作为文件名。

正式版本已经取消了多格式配置文件的支持，请使用 PHP 数组返回方式定义 `_config.php` 文件。

// PHP 数组方式定义，格式为：

```
<?php return array( 'DEBUG_MODE' => true,);?>
```

配置参数不区分大小写～无论大小写定义都会转换成小写。原则上，每个项目配置文件除了定义 ThinkPHP 所需要的配置参数之外，开发人员可以在里面添加项目需要的一些配置参数，系统必须的配置参数请查看后面的配置参数列表。

2、惯例配置

Rail 的重要思想就是惯例重于配置，新版的 ThinkPHP 吸收了这一特性，引入了惯例配置的支持。系统内置有一个惯例配置文件（位于 `Think\Common\convention.php`），按照大多数的使用对常用参数进行了默认配置。所以，对于应用项目的配置文件，往往只需要配置和惯例配置不同的或者新增的配置参数，如果你完全采用默认配置，甚至可以不需要定义任何配置文件。

3、模块配置

新版的 ThinkPHP 支持对某些参数进行动态配置，针对这一特性，ThinkPHP 还特别引入了模块配置文

件的支持，这其实也是动态配置的体现。模块配置文件的命名规则是

m_+模块名称+Config.php

4、操作（动态）配置

而在具体的 Action 方法里面，我们仍然可以对某些参数进行动态配置，主要是指那些还没有使用的参数。获取已经设置的参数值：

```
C('参数名称')
```

设置新的值：

```
C('参数名称','新的参数值');
```

模板支持

ThinkPHP 的配置参数的存取采用静态变量，以保证存取的速度。而在视图层还可以通过模板标签来直接显示配置参数的值，而无需进行模板赋值。

读取配置参数的标签用法如下：

```
{Think.config.参数名称}
```

4.1.8 ADSL 方法

ThinkPHP 为一些常用的操作定义了快捷方法，这些方法具有单字母的方法名，具有比较容易记忆的特点。非常有意思的是，这些快捷方法的字母包含了 ADSL 字母，所以我们称之为 ADSL 方法，但是并不局限于 ADSL 四个方法，包括下面的：

A 快速创建 Action 对象

```
$action = new UserAction();
```

可以简化为 `$action = A("User");`

而且，如果当前的 UserAction 类还没有引入的话，A 方法会自动引入。并且具有单例模式的支持，不会重复创建相同的 Action 对象。

D 快速创建模型数据对象

和 A 方法类似，只不过是快速创建数据对象

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

可以简化为 `$User = D("User");`

同样，如果 `UserModel` 类还没有加载，该方法会自动加载。还可以支持创建组件的数据对象。

例如，创建 `User` 组件的 `Info` 数据对象，可以使用

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

S 快速操作缓存方法

`S` 方法主要用于操作缓存方法，是一个集缓存设置和读取、删除操作于一体的方法。

例如：

```
// 用配置的缓存方式缓存 value 到 name 标识
```

```
S('name','value');
```

```
// 用内存的缓存方式缓存 value 到 name 标识，有效期 120 秒
```

```
S('name','value',120,'shmop');
```

```
// 获取缓存标 name 的值
```

```
S('name');
```

```
// 删除缓存标识 name
```

```
S('name',null);
```

`S` 方法隐藏了具体的缓存操作的细节，把看起来复杂的缓存操作变得轻而易举。

F 文件数据保存方法

`F` 方法主要用于项目的文件数据的写入、更改和删除，其工作机理和 `S` 方法是类似的，区别在于用途不同，数据保存的目录也不同，而且不能指定缓存方式，因为默认就是文件形式保存数据。

F 方法使用了 `var_export` 方法，所以只能支持简单数据类型，不支持对象的缓存。

L 快速操作语言变量

`L` 方法提供了多语言的支持，可以快速设置和获取语言定义。

例如：

设置名称为 `USER_INFO` 的语言变量

```
L('USER_INFO','用户信息');
```

```
// 获取 USER_INFO 的语言变量值
```

```
L('USER_INFO');
```

```
// 批量赋值
```

```
$array['USER_INFO'] = '用户信息';
```

```
$array['ERROR_INFO'] = '错误信息';
```

```
L($array);
```

注意：语言变量不区分大小写

C 快速操作配置变量

C 方法提供了对配置参数的设置、获取

设置名称为 USER_AUTH_ON 的配置参数

```
C('USER_AUTH_ON',true);
```

```
// 获取 USER_AUTH_ON 的变量值
```

```
C('USER_AUTH_ON');
```

```
// 批量赋值
```

```
$array['USER_AUTH_ON'] = true;
```

```
$array['USER_TIPS_MSG'] = '错误提示信息';
```

```
C($array);
```

注意：配置参数不区分大小写

4.1.9 多语言支持

ThinkPHP 内置多语言支持，如果你的应用涉及到国际化的支持，那么可以定义相关的语言包文件。任何字符串形式的输出，都可以定义语言常量。可以为项目定义不同的语言文件，框架的系统语言包目录在系统框架的 Lang 目录下面，每个语言都对应一个语言包文件，系统默认只有简体中文语言包文件 zh-cn.php，如果要增加繁体中文 zh-tw 或者英文 en，只要增加相应的文件。

语言包的使用由系统自动判断当前用户的浏览器支持语言来定位，如果找不到相关的语言包文件，会使用默认的语言。如果浏览器支持多种语言，那么取第一种支持语言。

ThinkPHP 的多语言支持已经相当完善了，可以满足应用的多语言需求。这里指的是模板多语言支持，

数据的多语言转换（翻译）不在这个范畴之内。ThinkPHP 具备语言包定义、自动识别、动态定义语言参数的功能。并且可以自动识别用户浏览器的语言，从而选择相应的语言包（如果有定义）。例如：

```
Throw_exception（'新增用户失败！'）；
```

我们在语言包里面增加了 `ADD_USER_ERROR` 语言配置变量后，在程序中的写法就要改为：

```
Throw_exception（L('ADD_USER_ERROR'））；
```

项目语言包文件位于项目的 `Lang` 目录下面，并且按照语言类别分子目录存放，在执行的时候系统会自动加载，无需手动加载。语言包文件可以按照模块来定义，每个模块单独定义语言包文件，文件名和模块名称相同，例如：

`Lang/zh-cn/user.php` 表示给 `User` 模块定义简体中文语言包文件

`Lang/zh-tw/user.php` 表示给 `User` 模块定义繁体中文语言包文件

语言子目录采用浏览器的语言命名(全部小写)定义，例如 `English (United States)` 可以使用 `en-us` 作为目录名。如果项目比较小，整个项目只有一个语言包文件，那可以定义 `common.php` 文件，而无需按照模块分开定义。

语言文件定义

ThinkPHP 语言文件定义采用返回数组方式：

```
return array(  
    'lan_define'=>'欢迎使用 ThinkPHP'  
);
```

要在程序里面设置语言定义的值，使用下面的方式：

```
L('define2','语言定义');
```

```
$value = L('define2');
```

如果要在模板中输出语言变量不需要在 `Action` 中赋值，可以直接使用模板引擎特殊标签来直接输出语言定义的值：

```
{Think.lang.define2}
```

上面的语言包是指项目的语言包，如果在提示信息的时候使用了框架底层的提示，那么还需要定义系

统的语言包，系统语言包目录位于 ThinkPHP 目录下面的 Lang 目录。

多语言的切换

ThinkPHP 的多语言支持是自动的，系统会根据用户的浏览器语言设置自动获取相关的语言包，如果找到匹配当前的语言包文件，就会自动加载进来，因此不需要你手动加载语言包文件在项目中。

如果你需要测试语言包的话，可以使用 `l` 参数（可以通过项目配置项 `VAR_LANGUAGE` 设置）来手动切换，进行效果测试。例如：

`http://ServerIP/AppName/Module/Action/?l=zh-tw` //切换到繁体中文

`http://ServerIP/AppName/Module/Action?l=en-us` //切换到英文

语言包切换后会自动保存 Cookie 记录，所以浏览其他页面的时候仍然可以保持上次选择的语言。

4.2 Action 控制器

4.2.1 项目编译

1.0.0 正式版本开始引入了新的项目编译特性，系统第一次运行的时候会在项目目录下面生成一个核心缓存文件 `~runtime.php` 和项目编译缓存文件 `~app.php`，所以在 Linux 环境下面需要对项目根目录设置可写权限，如果你担心这样给网站会带来安全隐患，可以在入口文件里面设置 `RUNTIME_PATH`，例如：

```
Defined('RUNTIME_PATH','./MyApp/runtime/');
```

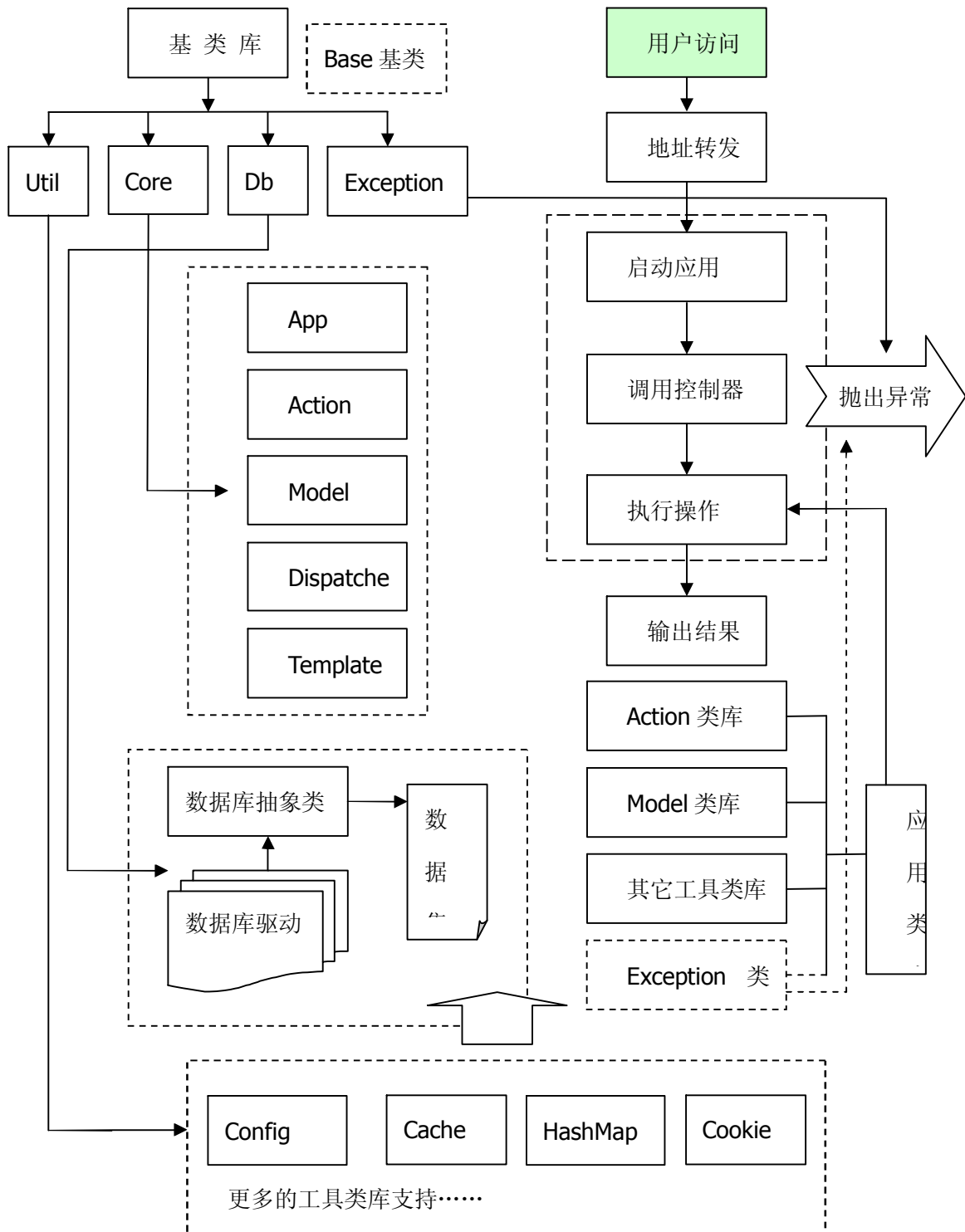
系统会自动把编译缓存文件放到该目录下面生成。

注意在调试模式下面不会生成项目编译缓存，但是依然会生成核心缓存。

如果在非调试模式下面，修改了配置文件或者项目公共文件，需要删除项目编译文件。

4.2.2 执行过程

ThinkPHP 采用模块和操作的方式来执行，任何一个 WEB 行为都可以认为是一个模块的某个操作，而模块的概念在 ThinkPHP 中的体现就是 Action 类，也类似于我们平常提到控制器的概念，而具体的操作就是 Action 类中的某个公共的方法。一个用户请求的执行过程如下：



首先，用户的请求会通过入口文件生成一个应用实例，应用控制器（我们称之为核心控制器）会管理整个用户执行的过程，并负责模块的调度和操作的执行，并且在最后销毁该应用实例。

4.2.3 模块和操作

ThinkPHP 里面会根据当前的 URL 来分析要执行的模块和操作。这个分析工作由 URL 调度器来实现，官方内置了 Dispatcher 来完成该调度。在 Dispatcher 调度器中，会根据

`http://servername/appName/moduleName/actionName/params`

来获取当前需要执行的项目（appName）、模块（moduleName）和操作（actionName），在某些情况下，appName 可以不需要（通常是网站的首页，因为项目名称可以在入口文件中指定，这种情况下，appName 就会被入口文件替代）

每个模块名称是一个 Action 文件，类似于我们平常所说的控制器，系统会自动寻找项目类库 Action 目录下面的相关类，如果没有找到，会尝试搜索应用目录下面的组件类中包含的模块类，如果依然没有，则抛出异常。

而 actionName 操作是首先判断是否存在 Action 类的公共方法，如果不存在则会继续寻找父类中的方法，如果依然不存在，就会寻找是否存在自动匹配的模版文件。如果存在模版文件，那么就直接渲染模版输出。

因此应用开发中的一个重要过程就是给不同的模块定义具体的操作。一个应用如果不需要和数据库交互的时候可以不需要定义模型类，但是必须定义 Action 控制器。

Action 控制器的定义非常简单，只要继承 Action 基础类就可以了，例如：

```
Class UserAction extends Action{  
  
}
```

你甚至不需要定义任何操作方法，就可以完成很多默认的操作，因为 Action 基础类已经为你定义了很多常用的操作方法。例如，我们可以执行（如果已经存在对应模板文件的情况）

`http://servername/index.php/User/`

`http://servername/index.php/User/add`

如果你需要增加或者重新定义自己的操作方法，增加一个方法就可以了，例如

```
Class UserAction extends Action{  
  
    // 定义一个 select 操作方法，注意操作方法不需要任何参数  
  
    Public function select(){  
  
        // select 操作方法的逻辑实现  
  
        // .....  
    }
```

```
$this->display(); // 输出模板页面  
}  
}
```

我们就可以执行<http://servername/index.php/User/select/>了，系统会自动定位当前操作的模板文件

4.2.4 URL 组成

我们在上面的执行过程里面看到的 URL 是大多数情况下，其实 ThinkPHP 支持三种 URL 模式，可以通过设置 URL_MODEL 参数来定义，包括普通模式、PATHINFO 和 REWRITE 模式。

普通模式 设置 URL_MODEL 为 0

采用传统的 URL 参数模式

<http://<serverName>/appName/?m=module&a=action&id=1>

PATHINFO 模式 设置 URL_MODEL 为 1

默认情况使用 URL_PATHINFO 模式，ThinkPHP 内置强大的 PATHINFO 支持，提供灵活和友好 URL 支持。

PATHINFO 模式还包括普通模式和智能模式两种：

普通模式 设置 PATH_MODEL 参数为 1

该模式下面 URL 参数没有顺序，例如

<http://<serverName>/appName/m/module/a/action/id/1>

<http://<serverName>/appName/a/action/id/1/m/module>

以上 URL 等效

智能模式 设置 PATH_MODEL 参数为 2 （系统默认的模式）

自动识别模块和操作，例如

<http://<serverName>/appName/module/action/id/1/> 或者

<http://<serverName>/appName/module,action,id,1/>

在智能模式下面，第一个参数会被解析成模块名称（或者路由名称，下面会有描述），第二个参数会被解析成操作（在第一个参数不是路由名称的前提下），后面的参数是显式传递的，而且必须成对出现，例如：

`http://<serverName>/appName/module/action/year/2008/month/09/day/21/`

其中参数之间的分割符号由 `PATH_DEPR` 参数设置，默认为 `"/"`，例如我们设置 `PATH_DEPR` 为 `^` 的话，就可以使用下面的 URL 访问

`http://<serverName>/appName/module^action^id^1/`

注意不要使用 `":"` 和 `"&"` 符号进行分割，该符号有特殊用途。

略加修改，就可以展示出富有诗意的 URL，呵呵～

如果想要简化 URL 的形式可以通过路由功能（后面会有描述），在系统不支持 `PATHINFO` 的情况下，一样可以使用 `PATHINFO` 模式的功能，只需要传入 `PATHINFO` 兼容模式获取变量 `VAR_PATHINFO`，默认值为 `s`，例如

`http://<serverName>/appName/?s=module/action/id/1/` 会执行和上面的 URL 等效的操作

并且也可以支持参数分割符号的定义，例如在 `PATH_DEPR` 为 `~` 的情况下，下面的 URL 有效

`http://<serverName>/appName/?s=module~action~id~1`

在 `PATH_INFO` 模式下面，会把相关参数转换成 `GET` 变量，以及并入 `REQUEST` 变量，因此不妨碍应用里面的以上变量获取。

REWRITE 模式 设置 `URL_MODEL` 为 2

该 URL 模式和 `PATHINFO` 模式功能一样，除了可以不需要在 URL 里面写入口文件，和可以定义 `.htaccess` 文件外。

例如，我们可以增加如下的 `.htaccess` 内容把所有操作都指向 `index.php` 文件。

```
<IfModule mod_rewrite.c>

RewriteEngine on

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php/$1 [QSA,PT,L]

</IfModule>
```

4.2.5 默认模块

如果使用 `http://<serverName>/index.php`，没有带任何模块和操作的参数，系统就会寻找默认模块和默认操作，通过 `DEFAULT_MODULE` 和 `DEFAULT_ACTION` 来定义，系统的默认模块设置是 `Index` 模块，默

认操作设置是 index 操作。也就是说

`http://<serverName>/index.php` 和

`http://<serverName>/index.php/Index` 以及

`http://<serverName>/index.php/Index/index` 等效。

4.2.6 模块伪装

ThinkPHP 支持模块和操作伪装，从而构造伪装的 URL 组成，该功能可以用于对已有的数据表定义成新的模块需要。

需要设置 `MODULE_REDIRECT` 参数，格式：

`Module1:TrueModuleName1,Module2:TrueModuleName2`

可以一次定义多个模块伪装，用逗号分割，伪装模块和真实模块名之间用冒号隔开。定义模块伪装后，伪装模块和真实模块名称都可以访问，例如：

假如定义了下面的参数，`'MODULE_REDIRECT'=>'User:Member,Info:Member_info'`，那么当我们在浏览器里面运行的时候，

`http://<serverName>/AppName/User/`和 `http://<serverName>/AppName/Member/`等效

`http://<serverName>/AppName/Info/`和 `http://<serverName>/AppName/Member_info/`等效

操作伪装的定义方式类似，设置 `ACTION_REDIRECT` 参数即可。

假如定义了下面的参数，`'ACTION_REDIRECT'=>'list:index'`，那么当我们在浏览器里面运行的时候，

`http://<serverName>/AppName/User/list/`和 `http://<serverName>/AppName/User/index/`等效

4.2.7 伪静态

系统支持伪静态 URL 设置，可以通过设置 `HTML_URL_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。例如，我们设置 `HTML_URL_SUFFIX` 为 `.shtml` 的话，我们可以把下面的 URL

`http://<serverName>/Blog/read/id/1`

变成

`http://<serverName>/Blog/read/id/1.shtml`

后者更具有静态页面的 URL 特征，但是具有和前面的 URL 相同的执行效果。

4.2.8 组件模块

ThinkPHP 允许把一系列模块和模型单独封装成一个组件，便于组件化和封装，这样的组件方式下面，Action 类和 Model 类的定义和操作方式不变，只是在外层增加了一个组件目录。例如，我们把 User、Info 和 UserType 三个模块，封装成一个 User 组件，应用目录结构变成：

```
|--- Lib 应用类库目录
    |--- User User 组件目录
        |----- Action 控制器（模块）类目录
        |----- Model MModel 类文件目录
```

对应的模板目录变成：

```
|----- Tpl 项目模版文件目录
|----- default 主题目录
    |----- User User 组件模版文件目录
        |----- User User 模块模版文件目录
        |----- Info Info 模块模版文件目录
        |----- UserType UserType 模块模版文件目录
```

并且，可以支持多层组件嵌套。

如果我们要访问 User 组件下面的 Info 模块的操作，可以使用下面的方式：

```
http://<serverName>/AppName/User:Info/list/
```

```
http://<serverName>/AppName/User:Info/add/
```

假如 User 组件下面还有子组件 Info，而 Info 组件下面有一个 Type 模块，那么可以用下面的方式访问：

```
http://<serverName>/AppName/User:Info:Type/list/
```

我们调用 Action 类和 Model 类的方式不变，依然使用 A("User")和 D("User")的方式，系统会自动定位组件下面的相关模块 Action 和 Model 对象。当然，为了提高检索效率，我们建议采用下面的方式：

A("User:Info") 和 D("User:Info") 来调用 User 组件下面的 Info 模块。

组件模块仅仅是为了部署方便，和原来的模块方式没有质的区别。

4.2.9 操作链

ThinkPHP 支持多操作的执行，默认情况下，系统会检测当前操作的前置和后置操作，如果存在就会按照顺序执行，例如，我们在 `UserAction` 类里面定义了 `_before_insert()` 和 `_after_insert()` 操作，那么执行 `User` 模块的 `insert` 操作的时候，会按照顺序执行下面的操作：

```
_before_insert
```

```
insert
```

```
_after_insert
```

特殊情况是，当前的 `add` 操作并没有定义操作方法，而是直接渲染模板文件，那么如果定义了 `_before_add` 和 `_after_add` 方法的话，依然会生效，也会按照这个顺序来执行 `add` 操作。真正有模板输出的可能仅仅是当前的 `add` 操作，前置和后置操作一般情况是没有任何输出的。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_`。

另外一种情况是，不通过前置和后置操作，而是使用操作链的方式，例如，我们访问下面的 URL：

```
http://<serverName>/appName/User/action1:action2:action3/
```

那么会依次执行 `UserAction` 的 `action1 action2 action3` 方法，并且当前操作名称是最后一个操作。在进行默认模板输出的时候会用到。如果确实需要在不同的操作方法中都进行输出，请确保在 `Action` 的 `display` 方法中指定需要渲染的模板文件名。否则，只能输出最后的操作模板。该模式下面，当前操作的前置和后置方法会失效。

4.2.10 路由功能

ThinkPHP 支持 URL 路由功能，要启用路由功能，需要设置 `ROUTER_ON` 参数为 `true`。开启路由功能后，系统会自动进行路由检测，如果在路由定义里面找到和当前 URL 匹配的路由名称，就会进行路由解析和重定向。路由功能需要定义路由定义文件，位于项目的配置目录下面，文件名为 `_routes.php`，定义格式：

```
Return Array(
```

```
// 第一种方式 常规路由
```

```
'RouteName'=>array('模块名称','操作名称','参数定义','额外参数'),
```

```
// 第二种方式 泛路由
```

```
'RouteName@'=>array(
    array('路由匹配正则','模块名称','操作名称','参数定义','额外参数'),
),
...更多的路由名称定义
)
```

或者

```
$_routes = Array(
    'RouteName'=>array('模块名称','操作名称','参数定义','额外参数'),
    'RouteName@'=>array(
        array('路由匹配正则','模块名称','操作名称','参数定义','额外参数'),
    ),
    ...更多的路由名称定义
)
```

系统在执行 Dispatch 解析的时候，会判断当前 URL 是否存在定义的路由名称，如果有就会按照定义的路由规则来进行 URL 解析。例如，我们启用了路由功能，并且定义了下面的一个路由规则：

```
'blog'=>array('Blog','index','year,month,day','userId=1&status=1')
```

那么我们在执行

`http://<serverName>/appName/blog/2007/9/15/`的时候就会实际执行 Blog 模块的 index 操作，后面的参数 `/2007/9/15/` 就会依次按照 `year/month/day` 来解析，并且会隐含传入 `userId=1` 和 `status=1` 两个参数。

另外一种路由参数的传入是

`http://<serverName>/appName/?r=blog&year=2007&month=9&day=15`，会执行上面相同的路由解析，该方式主要是提供不支持 PATHINFO 方式下面的兼容实现。其中 `r` 由 `VAR_ROUTER` 参数定义，默认是 `r`。

泛路由支持

新版引入了泛路由支持，提供了对同一个路由名称的多个规则的支持，使得 URL 的设置更加灵活，例如，我们对 Blog 路由名称需要多个规则的路由：

```
'Blog@'=>array(
    array('/^\\(\\d+)(\\p{\\d})?$/','Blog','read','id'),
    array('/^\\(\\d+)\\(\\d+)/','Blog','archive','year,month'),
)
```

),

第一个路由规则表示解析 Blog/123 这样的 URL 到 Blog 模块的 read 操作

第二个路由规则表示解析 Blog/2007/08 这样的 URL 到 Blog 模块的 archive 操作

泛路由的定义难度就在路由正则的定义上面，其它参数和常规路由的使用一致。

4.2.11 空操作

如果执行一个没有定义的操作，并且也没有定义任何对应的模板，那么当前操作就属于空操作。

ThinkPHP 对于空操作有一个约定，首先会判断当前 Action 是否定义了 `_empty` 操作方法，如果有则会调用该操作方法执行。否则如果是调试模式就会抛出操作不存在的异常，如果是部署模式，就会跳转到默认操作

4.2.12 特殊操作

通用前置和后置操作

在 ThinkPHP 中，你可以给任何一个操作定义前置和后置操作方法。

前置操作的定义就是在当前操作方法的前面添加 `_before_` 前缀，而后置操作的定义就是在当前操作方法的前面添加 `_after_` 前缀。

例如，我们已经给 UserAction 定义了一个 del 操作方法，我们可以定义：

`_before_del()` 在 del 操作之前执行

`_after_del()` 在 del 操作之后执行（del 操作正常执行完毕）

内置前置和后置操作

另外，如果使用系统内置的 insert 和 update 操作的话，会首先检查是否存在 `_operation` 方法，这个方法是系统在表单提交时候的默认前置操作。

`_operation` 方法没有任何参数，主要是用来检测表单的提交数据。

在 insert 或者 update 执行完毕后，系统会检查是否存在 `_trigger` 方法，这个方法是系统默认的数据更新触发器方法。

`_trigger` 方法只有一个参数，也就是要保存的 Vo 对象。

列表过滤方法

如果你使用系统内置的 `index` 方法，系统还会检查 `_filter` 方法，该方法负责对列表数据进行过滤，当你的列表需要针对不同的用户进行判断和过滤的时候，这个方法非常有效。

4.2.13 Action 缓存

ThinkPHP 内置了 Action 缓存功能，需要启用 Action 缓存功能，设置 Action 的 `userCache` 属性为 `true` 即可，例如：

```
$this->setCache(true);
```

然后设置 `_cacheAction` 属性，增加需要缓存的 Action 名称即可，例如：

对当前 Action 控制器的 `list` 和 `add` 操作使用 Action 缓存

```
$_cacheAction = array('list','add');
```

比较常用的方式是在 Action 控制器的初始化方法里面定义，例如：

```
function _initialize() {  
    $this->setCache(true);  
    $this->_cacheAction = array('index','add');  
    parent::_initialize();  
}
```

需要注意的是 Action 缓存是使用当前的 URL 地址作为标识来缓存的，所以对于相同的 Action 操作，如果后面的参数不同的话，是不同的缓存，例如：

`http://<serverName>/appName/User/read/id/1`

`http://<serverName>/appName/User/read/id/2`

同样是 `read` 操作，但是是分开缓存的。

4.2.14 静态缓存

ThinkPHP 内置了静态缓存的功能，并且支持静态缓存的规则定义。

要使用静态缓存功能，需要开启 `HTML_CACHE_ON` 参数，并且在项目配置目录下面增加静态缓存规则文件 `_htmls.php`，规则的定义方式如下：

```
Return Array(  
    'ActionName'=>array('静态规则','静态缓存有效期','附加规则'),
```

```
'ModuleName:ActionName'=>array('静态规则','静态缓存有效期','附加规则'),  
'*'=>array('静态规则','静态缓存有效期','附加规则'),  
...更多操作的静态规则  
)
```

静态缓存文件的根目录在 `HTML_PATH` 定义的路径下面，并且只有定义了静态规则的操作才会进行静态缓存，注意，静态规则的定义有三种方式，

第一种是定义全局的操作静态规则，例如定义所有的 `read` 操作的静态规则为

```
'read'=>array('{id}','60')
```

其中，`{id}` 表示取 `$_GET['id']` 为静态缓存文件名，第二个参数表示缓存 60 秒

第二种是定义某个模块的操作的静态规则，例如，我们需要定义 `Blog` 模块的 `read` 操作进行静态缓存

```
'Blog:read'=>array('{id}',-1)
```

第三种方式是定义全局的静态缓存规则，这个属于特殊情况下的使用，任何模块的操作都适用，例如

```
'*'=>array('{$_SERVER.REQUEST_URI|md5}'), 根据当前的 URL 进行缓存
```

静态规则的写法可以包括以下情况

使用系统变量 包括 `_GET _REQUEST _SERVER _SESSION _COOKIE`

格式：`{$_xxx|function}`

例如：`{$_GET.name}{$_SERVER.REQUEST_URI}`

框架变量 框架特定的变量

例如：`{:module}{:action}` 表示当前模块名和操作名

使用 `_GET` 变量

```
{var|function}
```

也就是说 `{id}` 其实等效于 `{$_GET.id}`

直接使用函数

```
{|function}
```

例如：`{|time}`

支持混合定义，例如我们可以定义一个静态规则为：

```
'{id},{name|md5}'
```

在`{}`之外的字符作为字符串对待，如果包含有`"/"`，会自动创建目录。

例如，定义下面的静态规则：

```
{:module}/{:action}_{id}
```

则会在静态目录下面创建模块名称的子目录，然后写入操作名_id.shtml 文件。

静态有效时间如果不定义，则会获取配置参数 `HTML_CACHE_TIME` 的设置值

附加规则通常用于对静态规则进行函数运算，例如

```
'read'=>array('Think{id},{name}','60','md5')
```

翻译后的静态规则是 `md5('Think'._GET['id'].'._GET['name']');`

页面静态化后读取的规则有两种方式，通过 `HTML_READ_TYPE` 设置：

一种是直接读取缓存文件输出（`readfile` 方式 `HTML_READ_TYPE` 为 0）这是系统默认的方式，属于隐含静态化，用户看到的 URL 地址是没有变化的。

另外一种方式是重定向到静态文件的方式（`HTML_READ_TYPE` 为 1），这种方式下面，用户可以看到 URL 的地址属于静态页面地址，比较直观。

4.2.15 浏览器缓存

可以在 ThinkPHP 里面启用浏览器缓存功能，设置 `LIMIT_RESFRESH_ON` 为 `True`，并且设置缓存有效时间 `LIMIT_REFLESH_TIMES`（单位为秒），该功能可以防止页面频繁刷新，系统会根据用户访问页面的 `$_SERVER['PHP_SELF']` 的值来进行缓存标识。

4.2.16 URL 组装

为了方便不同 URL 模式下面的跳转，系统包含了一个 `url` 方法用于 URL 的组装，并且自动根据当前的 URL 模式进行判断，而且支持参数和路由功能，如果设置了伪静态后缀的话，还可以自动生成伪静态的后缀。

4.2.17 操作重定向

ThinkPHP 支持操作的重定向操作，可以改变 URL 地址跳转。

例如：

```
// 跳转到 User 模块的 read 操作
```

```
$this->redirect('read/id/1','User');
```

4.2.18 隐含执行

ThinkPHP 支持隐含执行某个 Action 操作，并且可以指定项目、模块，以及延时执行。

例如：

执行 User 模块的 read 操作

```
$this->forward('read','User');
```

4.2.19 Session

ThinkPHP 对 Session 操作进行了静态封装，Session 类无需引入就可以使用，Session 的启动会在应用初始化自动执行，所以无需手动使用 Session::start() 来启动 Session。

最常用的操作方法示例：

```
// 检测 Session 变量是否存在
```

```
Session::is_set('name');
```

```
// 给 Session 变量赋值
```

```
Session::set('name','value');
```

```
// 获取 Session 变量
```

```
Session::get('name');
```

和 Session 相关的配置参数：

```
'SESSION_NAME'=>'ThinkID',    // 默认 Session_name  
'SESSION_PATH'=>'',          // 采用默认的 Session save path  
'SESSION_TYPE'=>'File',       // 默认 Session 类型 支持 DB 和 File  
'SESSION_EXPIRE'=>'300000',   // 默认 Session 有效期  
'SESSION_TABLE'=>'think_session', // 数据库 Session 方式表名  
'SESSION_CALLBACK'=>'',       // 反序列化对象的回调方法
```

其中 SESSION_NAME 参数需要注意，如果需要在不同的项目之间不共享传递 Session 的值，请设置不

同的值，否则请保留相同的默认值。

如果设置了相同的 `SESSION_NAME` 的值，但是又希望创建基于项目的私有 Session 空间，应该怎么处理呢？ThinkPHP 还支持以项目为 Session 空间的私有 Session 操作，以之前的常用操作为例，我们更改如下：

```
// 检测 Session 变量是否存在（当前项目有效）
```

```
Session::is_setLocal('name');
```

```
// 给 Session 变量赋值（当前项目有效）
```

```
Session::setLocal('name','value');
```

```
// 获取 Session 变量（当前项目有效）
```

```
Session::getLocal('name');
```

这样，和全局的 Session 操作就不会冲突，可以用于一些特殊情况的需要。

ThinkPHP 支持数据库方式的 Session 操作，设置 `SESSION_TYPE` 的值为 `DB` 就可以了，如果使用数据库方式，还要确保设置好 `SESSION_TABLE` 的值，并且导入下面的 DDL 到你的数据库（以 MySQL 为例子）：

```
CREATE TABLE `think_session` (  
    `id` int(11) unsigned NOT NULL auto_increment,  
    `session_id` varchar(255) NOT NULL,  
    `session_expires` int(11) NOT NULL,  
    `session_data` blob,  
    PRIMARY KEY (`id`)  
)
```

注意，Db Session 方式的数据库连接会采用项目的数据库配置信息进行连接。除了数据库方式外，还可以增加其它方式的 Session 保存机制，例如内存方式、Memcache 方式等，我们只要增加相应的过滤器就行了，使用 `session_set_save_handler` 方法，具体的方法定义参考 `Think.Util.Filter` 下面的 `FilterSessionDb.class.php` 文件的实现。

4.2.20Cookie

ThinkPHP 还封装了 Cookie 操作静态类，可以实现简单的 Cookie 操作，并且无需引入就可以直接使用。

和 Cookie 相关的设置参数有：

```
'COOKIE_EXPIRE'=>30000000000,          // Cookie 有效期  
'COOKIE_DOMAIN'=>$_SERVER['HTTP_HOST'], // Cookie 有效域名  
'COOKIE_PATH'=> '/',                    // Cookie 路径  
'COOKIE_PREFIX'=>'THINK_',              // Cookie 前缀 避免冲突
```

常用的操作方法:

// 检测 Cookie 变量是否存在

```
Cookie::is_set('name');
```

// 设置 Cookie 值

```
Cookie::set('name','value');
```

// 获取 Cookie 值

```
Cookie::get('name');
```

Set 方法还支持三个可选参数

\$expire 有效期

\$path Cookie 路径

\$domain Cookie 域名

如果没有设置则使用配置文件里面的相关配置的值。

4.3 ActiveRecord 基础

ThinkPHP 实现了 ActiveRecord 模式的 ORM 模型, 采用了非标准的 ORM 模型: 表映射到类, 记录(集)映射到对象, 字段属性映射到对象的虚拟属性。最大的特点就是使用方便, 从而达到敏捷开发的目的。开发过程中, 只需要定义好模型类就可以进行方便的数据操作了, 例如我们定义了一个 UserModel 类:

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

甚至无需增加任何属性和方法, 我们就可以进行下面的操作了。

```
$User = D("User"); // 实例化 User 对象
```

```
// 或者 $User = new UserModel();
```

```
$User->find(1); // 查找 id 为 1 的记录
```

```
$User->name = 'ThinkPHP'; // 把查找到的记录的名称字段修改为 ThinkPHP
```

```
$User->save(); // 保存修改的数据
```

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

```
$User->findAll();

foreach ($User as $user){

    echo $user->name;

}
```

内部的操作细节全部被隐藏了，包括数据库的连接。想一下是否觉得不可思议呢～

4.3.1 数据库访问层

业务逻辑一般都是对数据的访问和操作，为了兼容性考虑，ThinkPHP 框架没有使用 Adodb 或者 PDO（PHP5 才内置支持）作为抽象数据库访问层，而是内置实现了一个抽象数据库访问层，当然这个抽象层的实现上也参考了部分目前比较先进的抽象数据库访问层框架，并且会保持不断完善。

ThinkPHP 的抽象数据库访问层主要包含了数据库公共类、各类数据库驱动类和数据集对象。在实际应用开发过程中，你可能并不需要接触这些类库，只需要通过各自的 Model 对象来访问和操作数据库就可以。数据库操作类库的主要特点有：

支持事务和回滚（当然首先需要数据库本身支持），但是默认是自动提交

1. 在数据库调试模式下支持 SQL 语句的输出
2. 完全支持中文 UTF-8 字符集
3. 把查询操作和执行操作分开，便于返回不同的结果
4. 支持配置文件和 DSN 两种数据库配置读取方式
5. 支持对数据集的数组和对象两种方式返回
6. 支持多个数据库连接
7. 支持主从式分布式数据库

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

抽象数据库访问层的关键类库在于数据库的公共访问类 Db 类，这个类是操作数据库的底层接口，换句话说，Db 类不能够独立存在，必须有对应的数据库驱动类库，但是开发人员不能够跨过该类直接进行数据库操作的访问。目前系统支持的数据库包含有 MySQL、MySQLi、MsSQL、PgSQL、SQLite 和 Oracle，

以及 PDO 的支持，还可以通过插件方式增加自己需要的数据库驱动，开发人员也可以按照规范编写自己的数据库驱动（甚至包括众多的嵌入式数据库的驱动）。

4.3.2 表和主键

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

`UserModel` 表示 `User` 数据对象，其对应的数据表应该是

`think_user`（假设数据库的前缀定义是 `think_`）

`UserTypeModel` 对应的数据表是 `think_usertype`

`User_typeModel` 对应的数据表是 `think_user_type`

如果你的规则和系统的约定不符合，那么需要设置 `Model` 类的 `tableName` 属性，或者覆盖父类的 `getTableName` 方法。

在 ThinkPHP 的模型里面，有两个数据表名称的定义：

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

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

下面举个例子来加深理解：

例如，在数据库里面有一个 `think_categories` 表，而我们定义的模型类名称是 `CategoryModel`，按照系统的约定，这个模型的名称是 `Category`，对应的数据表名称应该是 `category`（全部小写），但是现在的数据表名称是 `categories`，因此我们就需要设置 `tableName` 属性来改变默认的规则（假设我们已经在配置文件里面定义了 `DB_PREFIX` 为 `think_`）。

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

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

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

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

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

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

```
$Model->getPk();
```

4.3.3 属性访问

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

最常用的访问方式是直接通过数据对象访问，例如

```
$User = D("User");  
$User->find(1);  
// 获取 name 属性的值  
echo $User->name;  
// 设置 name 属性的值  
$User->name = 'ThinkPHP';
```

还有一种属性的操作方式是通过返回一个数据对象的方式：


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

// 注意这里返回的 user 数据对象可能并不一定是对象，也可以是一个数组
// 系统默认的惯例配置返回的是一个数组参考 DATA_RESULT_TYPE 的配置值

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

// 如果 DATA_RESULT_TYPE 是 1 那么使用下面的方式操作
// 获取 name 属性的值
echo $user->name;

// 设置 name 属性的值
$user->name = 'ThinkPHP';

// 如果 DATA_RESULT_TYPE 是 0 那么使用下面的方式操作
// 获取 name 属性的值
echo $user['name'];

// 设置 name 属性的值
$user['name'] = 'ThinkPHP';
```

两种方式的属性区别是一个是对象的属性，一个是数组的索引名称。

那么如何获取当前数据对象的所有属性呢，有两个方法可以做到：

使用 `getDbFields` 方法来获取当前数据对象的属性数组

```
$User->getDbFields();

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

4.3.4 连接数据库

ThinkPHP 内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的 `Db` 类进行操作，而无需针对不同的数据库写不同的代码和底层实现，`Db` 类会自动调用相应的数据库适配器来

处理。目前的数据库包括 MySQL、MsSQL、PgSQL、Sqlite、Oracle 和 PDO 的支持，如果应用需要使用数据库，必须配置数据库连接信息，数据库的配置文件有多种定义方式：

第一种 在项目配置文件里面定义

```
Return array(  
'DB_TYPE'=> 'mysql',  
'DB_HOST'=> 'localhost',  
'DB_NAME'=>'web',  
'DB_USER'=>'root',  
'DB_PWD'=>'',  
'DB_PORT'=>'',  
..... 其它项目配置参数  
);
```

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

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

第二种 使用 DSN 方式在初始化 Db 类的时候传参数

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

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

第三种 使用数组传参数

```
$DSN = array(  
'dbms' => 'mysql',  
'username' => 'username',  
'password' => 'password',  
'hostname' => 'localhost',
```

```
'hostport' => '3306',  
  
'database' => 'dbname'  
  
);  
  
$db = new Db($DSN);
```

该方式也是用于手动连接数据库的情况，或者用于创建多个数据库连接。

第四种 在模型类里面定义

```
$connection = array(  
  
    'dbms'    => 'mysql',  
  
    'username' => 'username',  
  
    'password' => 'password',  
  
    'hostname' => 'localhost',  
  
    'hostport' => '3306',  
  
    'database' => 'dbname'  
  
);  
  
// 或者使用下面的定义
```

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

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

4.3.5 使用 PDO

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

我们以项目配置文件定义为例来说明：

```
Return array(  
  
    'DB_TYPE'=> 'pdo',  
  
    'DB_DSN'=>'mysql:host=localhost;dbname=think',  
  
    'DB_USER'=>'root',  
  
    'DB_PWD'=>'pwd',  
  
    ..... 其它项目配置参数
```

);

使用 PDO 方式的时候，要注意检查是否开启相关的 PDO 模块。

4.3.6 CURD

ThinkPHP 提供了灵活和方便的数据操作方法，不仅实现了对数据库操作的四大基本操作（CURD）：创建、读取、更新和删除的实现，还内置了很多实用的数据操作方法，提供了 ActiveRecord 模式的最佳体验。

Model 类将数据库操作统一为 CURD 和一个 SQL 查询方法，也就是

`_create` 新增（写入）数据

`_update` 更新（保存）数据

`_read` 读取（查询）数据

`_delete` 删除数据

`_query` SQL 查询

其它的所有方法基本上对你调用这些基础方法进行操作，但是我们无需关注这些细节，只需要按照提供的抽象方法来操作，我们来看看在 ThinkPHP 中是怎么进行数据操作的。

新建记录

```
// 实例化一个 User 模型对象
```

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

```
// 然后给数据对象赋值
```

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

```
$User->email = 'ThinkPHP@gmail.com';
```

```
// 然后就可以保存新建的 User 对象了
```

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

```
// 如果需要锁实例化模型对象的时候传入数据，可以使用
```

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

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

```
$User = new UserModel($data);
```

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

// 或者直接在 add 方法传入要新建的数据

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

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

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

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

如果你的主键是自动增长类型，不需要传入主键的值就可以新建数据，并且如果插入数据成功的话，Add 方法的返回值就是最新插入的主键值，可以直接获取。

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

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

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

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

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

```
$User->add(); // 新增表单提交的数据
```

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

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

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

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

```
// 从 User 数据对象创建新的 Member 数据对象
```

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

```
$Member->create($User);
```

支持新增多条记录

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

```
$data[0]['name'] = 'ThinkPHP';
```

```
$data[0]['email'] = 'ThinkPHP@gmail.com';
```

```
$data[1]['name'] = '流年';
```

```
$data[1]['email'] = 'liu21st@gmail.com';
```

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

在 MySQL 数据库下面，会自动使用一条 SQL 语句实现多数据的插入。

查询记录

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

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

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

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

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

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

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

遍历查询到的数据对象属性

```
foreach ($User as $key=>$val){
```

```
    echo($key.':'.$val);
```

```
}
```

// 或者进行相关的数据更改和保存操作

也可以用变量保存下来以便随时使用。

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

对于上面的查询条件，我们还可以使用 `getById` 来完成相同的查询

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

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

// 查询主键为 1、3、8 的记录集

```
$User->findAll('1,3,8');
```

```
// 遍历数据列表
```

```
foreach ($User as $vo){  
    dump($vo->name);  
}
```

更多的查询操作请参考后面章节的内容。

更新记录

了解了查询记录后，更新操作就显得非常简单了。

```
$User->find(1); // 查找主键为 1 的数据
```

```
$User->name = 'TOPThink'; // 修改数据对象
```

```
$User->save(); // 保存当前数据对象
```

```
// 还可以使用下面的方式更新
```

```
$User->score = '{score+1}'; // 对用户的积分加 1
```

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

如果不是使用数据对象的方式来保存，可以传入要保存的数据和条件

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

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

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

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

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

同样可以支持对字段的操作

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

```
// 或者改成下面的
```

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

删除记录

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

```
$User->delete(); // 删除查找到的记录
```

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

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

4.3.7 查询语言

正如上面我们看到的一样，ThinkPHP 大多数情况使用的都是对象查询，因为充分利用了 ORM 查询语言，了解查询条件的定义对使用对象查询非常有帮助，前面看到的例子我们都只是使用字符串作为查询条件，对于复杂的查询，或者从安全方面考虑，通常我们可以使用 **HashMap** 对象或者索引数组来传递查询条件。查询条件可以用于 **find**、**findAll** 等所有有查询条件的方法，下面是几种查询条件的定义：

普通查询

```
$condition = new HashMap();
```

```
// 查询 name 为 thinkphp 的记录
```

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

```
// 使用数组作为查询条件
```

```
$condition = Array();
```

```
$condition['name'] = 'thinkphp';
```

使用 **Map** 方式查询和使用数组查询的效果是相同的，并且是可以互换的。

条件查询

在查询条件里面，如果仅仅使用

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

查询条件应该是 `name = 'thinkphp'`

如果需要进行其它方式的条件判断，可以使用

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

这样，查询条件就变成 `name like 'thinkphp%'`

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

查询条件 `id > 100`

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

```
// 或者使用数组范围
```

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

上面表示的查询条件都是 id in (1,3,8)

支持的查询表达式有

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

区间查询

ThinkPHP 支持对某个字段的区间查询，例如：

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

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

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

组合查询

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

```
$map->put('id',array('neq',1));
```

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

// 现在的条件是 id !=1 and name like '%ok%'

```
$map->put('_logic','or');
```

// 现在的条件变为 id !=1 or name like '%ok%'

多字段查询

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

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

查询条件是 (id != 1) OR (name like 'aaa') OR (title like '%bbb%')

如果结合上面的几种方式，我们可以写出下面更加复杂的查询条件

```
$map->put('id',array('NOT IN','1,6,9'));
```

```
$map->put('name,email',array(array('like','thinkphp'),array('like','liu21st'),'or'));
```

以上查询条件变成：


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

4.3.8 查询缓存

ThinkPHP 内建查询缓存支持，可以包括对 SQL 查询缓存、数据对象缓存的缓存支持。对于同一个数据对象的多次查询并不会导致多次数据库的查询开销，系统内置会进行缓存判断。

4.3.9 统计查询

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

```
// 获取用户数
$userCount = $User->count();

// 获取用户的最大积分
$maxScore = $User->max('score');

// 获取积分大于 0 的用户的最小积分
$minScore = $User->min('score','score>0');

// 获取用户的平均积分
$avgScore = $User->avg('score');

// 统计用户的总成绩
$sumScore = $User->sum('score');
```

4.3.10 定位查询

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

```
// 返回符合条件的第 2 条记录
$user = $User->getN(2,'score>80','score desc');

还可以获取最后第二条记录
$user = $User->getN(-2,'score>80','score desc');

如果要查询第一条记录，还可以使用
$user = $User->first('score>80','score desc');
```

```
// 获取最后一条记录
$User->last('score>80','score desc');

// 获取积分最高的前 5 条记录
$User->top(5,'','score desc');
```

4.3.11 动态查询

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

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

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

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

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

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

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

例如，我们需要获取当前用户中积分大于 0，积分最高的前 5 位用户

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

而在另外一个频道，我们需要获取点击最多的前 10 位主播

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

4.3.12 SQL 查询

ThinkPHP 支持原生的 SQL 查询，在某些特殊的情况下可以满足应用的需要。

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，所以永远是返回的数据集对象或者惰性数据查询对象。而且可以支持查询缓存、延迟加载和事务锁（悲观锁）。

SQL 查询使用 query 方法

```
$list = $User->query("select id,name from think_user");
```

如果没有使用延迟加载，返回结果是 `ArrayObject` 对象，如果启用了延迟加载，那么返回的是 `ResultIterator` 对象。

4.3.13 事务支持

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

```
// 启动事务
$User->startTrans()

// 提交事务
$User->commit()

// 事务回滚
$User->rollback();
```

在有些对多个数据的批量操作中，已经内建了事务支持。

4.4 更多的 ActiveRecord 特性

4.4.1 分布式数据库支持

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

如果采用分布式数据库，定义数据库配置信息的方式如下：

```
// 在项目配置文件里面定义

Return array(

'DB_TYPE'=>'mysql', // 分布式数据库的类型必须相同

'DB_HOST'=>'192.168.0.1,192.168.0.2',

'DB_NAME'=>'thinkphp', // 如果相同可以不用定义多个

'DB_USER'=>'user1,user2',

'DB_PWD'=>'pwd1,pwd2',

'DB_PORT'=>'3306',

'DB_PREFIX'=>'think',

..... 其它项目配置参数
```

);

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

'DB PORT'=>'3306,3306' 和 'DB PORT'=>'3306' 等效

```
'DB_USER'=>'user1',
```

```
'DB_PWD'=>'pwd1',
```

和

```
'DB USER'=>'user1,user1',
```

```
'DB_PWD'=>'pwd1,pwd1',
```

等效。

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

```
'DB RW SEPARATE'=>true,
```

在读写分离的情况下，第一个数据库配置是主服务器的配置信息，负责写入数据，其它的都是从数据库的配置信息，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随机进行在从服务器中选择。主从数据库的数据同步工作不在框架实现。

4.4.2 多数据库连接支持

分布式数据库的配置信息是定义在配置文件里面的，所以一般情况下是无法更改的。另外使用分布式数据库连接有个不足，就是无法同时连接多个不同类型的数据库。如果你的应用需要在特殊的时候连接另外的数据库，这通常是在某些特别的数据获取情况下，那么可以尝试使用 **ThinkPHP** 的多数据库连接特性：包括相同类型的数据库和不同类型的数据库。

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

我们首先需要在模型类里面增加需要的数据库连接，例如：

```
// 我们在 UserModel 类增加多个数据库连接
```

```
// 首先定义额外的数据库连接信息
```

```
$myConnect1 = array(
```

'dbms' => 'mysql',

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

// 或者使用下面的定义

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

// 增加数据库连接 第二个参数表示连接的序号

// 注意内置的数据库连接序号是 0,所以额外的数据库连接序号应该从 1 开始

// 第三个参数表示是否是相同类型的数据库连接, 这里我们假设和项目配置采用相同的数据库类型

mysql

```
$User->D("User");
```

```
$User->addConnect($myConnect1,1,true);
```

// 如果需要增加另外的 mssql 数据库连接, 使用下面的方法

```
$User->addConnect($myConnect1,1,false);
```

// 可以同时增加多个数据库连接 myConnect2 和 myConnect3 的定义方式同 myConnect1

```
$User->addConnect($myConnect1,1,false);
```

```
$User->addConnect($myConnect2,2,false);
```

```
$User->addConnect($myConnect3,3,true);
```

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

```
$User->switchConnect($myConnect2);
```

switchConnect 方法会智能识别该连接是否是相同类型的连接

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

// 删除连接序号为 2 的数据库连接

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

4.4.3 数据验证

系统内置了数据对象的自动验证功能,而大多数情况下面,数据对象是由表单提交的\$_POST 数据创建。需要使用系统的自动验证功能,只需要在 Model 类里面定义\$ _validate 属性,是由多个验证因子组成的数组,支持的验证因子格式:

array(验证字段,验证规则,错误提示,验证条件,附加规则,验证时间)

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

验证规则 要进行验证的规则,需要结合附加规则

提示信息 用于验证失败后的提示信息定义

验证因子中上面三个参数必须定义,下面为可选参数。

验证条件

EXISTS_TO_VALIDATE 或者 0 存在字段就验证 (默认)

MUST_TO_VALIDATE 或者 1 必须验证

VALUE_TO_VALIDATE 或者 2 值不为空的时候验证

附加规则 配合验证规则使用,包括:

function 使用函数验证,前面定义的验证规则是一个函数名

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

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

equal 验证是否等于某个值,该值由前面的验证规则定义

in 验证是否在某个范围内,前面定义的验证规则必须是一个数组

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

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

如果采用正则进行验证,会调用系统内置的验证类进行验证操作,该验证类位于 **ORG.Text.Validation**,通过正则的方式对数据进行验证,并定义了一些常用的验证规则。包括:

require 字段必须

email 邮箱

url URL 地址

currency 货币

number 数字

这些验证规则可以直接使用。

验证时间:

all 全部情况下验证（默认）

add 新增数据时候验证

edit 编辑数据时候验证

示例:

```
var $_validate = array(
    array('verify','require','验证码必须！ '), //所有情况下用正则进行验证
    array(name,"','帐号名称已经存在！ ',0,'unique','add'), // 在新增的时候验证 name 字段是否唯一
    array('value',array(1,2,3),'值的范围不正确！ ',2,'in'), // 当值不为空的时候判断是否在一个范围内
    array('repassword','password','确认密码不正确',0,'confirm'), // 验证确认密码是否和密码一致
    array('password','checkPwd','密码格式不正确',0,'function'), // 自定义函数验证密码格式
);
```

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

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

4.4.4 数据处理

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

`$_auto` 属性是由多个填充因子组成的数组，填充因子定义格式：

`array(填充字段,填充内容,填充条件,附加规则)`

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

填充条件包括：

ADD 新增数据的时候处理（默认方式）

UPDATE 更新数据的时候处理

ALL 所有情况下都进行处理

附加规则包括：

function 使用函数

callback 回调方法

field 用其它字段填充

string 字符串（默认方式）

示例：

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

PS：该自动填充可能会覆盖表单提交项目。其目的是为了防止表单非法提交字段。

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

4.4.5 记录时间戳

通过 ThinkPHP 的自动数据处理功能，我们可以设置每次编辑的时候写入当前时间戳，例如：

```
array('mTime','time','UPDATE','function'),
```

其实，ThinkPHP 内置支持自动记录时间戳的功能，所以可以无需通过定义自动处理也可以完成时间戳

的自动写入功能。

在创建数据对象的时候 如果数据表中有 `autoCreateTimestamps` 定义的字段属性，那么会自动记录当前时间戳，默认的 `autoCreateTimestamps` 定义了 `create_at`、`create_on`、`cTime` 三个字段名称，默认以时间戳方式记录，如果需要以其它日期格式记录，可以设置 `autoTimeFormat` 属性，例如：

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

同样的，在数据对象保存的时候，会检测是否有 `autoUpdateTimestamps` 定义的字段属性，如果存在则会自动记录时间戳，默认的 `autoUpdateTimestamps` 定义了 `update_at`、`update_on`、`mTime` 三个字段名称。

4.4.6 回调方法

`_initialize` 初始化回调方法

`_before_validation` 前置验证回调

`_after_validation` 后置验证回调

`_before_operation` 前置数据处理回调

`_after_operation` 后置数据处理回调

除了上面的回调方法，`Model` 还包含了：

`_create` 写入数据

`_update` 更新数据

`_read` 查询数据

`_delete` 删除数据

`_query` SQL 查询

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

这些回调方法在系统的

如果前置方法返回 `false`，那么将会中止后面的操作。

4.4.7 延迟加载

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

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

// 第一种 使用 Model 类的 query 方法进行延迟加载，该方法主要针对 SQL 查询方式
// 第二个参数是是否缓存 第三个参数是是否采用延迟加载
$list = $User->query("select * from think_user",false,true);

// 当使用 foreach 进行遍历的时候才真正进行查询
foreach ($list as $data){
    dump($data);
}

// 第二种 使用 startLazy 和 stopLazy 方法进行延迟加载，可以用在 Model 类的所有查询方式
$User->startLazy();
$list = $User->findAll();
$User->stopLazy();
foreach ($list as $data){
    dump($data);
}

// 第三种 直接在方法后面使用 lazy 参数，进行延迟加载，这种方式略显麻烦
$User->findAll($condition,$fields,$order,$limit,$group,$having,$cache,$relation,$lazy)
```

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

4.4.8 聚合对象

我们可以把数据表中的某些属性进行数据封装，这样就把枯燥的数据表字段赋予更好的可读性。例如，在数据库有一个 User 表，字段定义如下：

```
CREATE TABLE `topthink_user` (
```

```
`id` int(11) unsigned NOT NULL auto_increment,  
`name` varchar(30) NOT NULL default '',  
`nickname` varchar(50) NOT NULL default '',  
`password` varchar(32) NOT NULL default '',  
`email` varchar(255) NOT NULL default '',  
`url` varchar(255) NOT NULL default '',  
`status` tinyint(1) unsigned NOT NULL default '0',  
`remark` varchar(255) NOT NULL,  
`city` varchar(50) NOT NULL,  
`qq` varchar(15) NOT NULL,  
PRIMARY KEY (`id`),  
)
```

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

id、name、password、Info（聚合对象属性，包括 nickname、email、url、city、qq）、status

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

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

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

我们需要在 UserModel 类里面增加聚合对象的定义

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

我们可以同时定义多个聚合对象，例如：

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

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

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

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

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

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

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

上面的操作，`msn` 属性就不会写入数据库。

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

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

于是，下面的操作

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

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

系统会自动把 `web` 属性转换成 `url` 字段名称写入数据库。

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

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

```
Class InfoModel extends Model{
```

```
    function _initialize(){
```

```
        // 表示该对象是复合对象
```

```
        $this->composite = true;
```

```
    }
```

```
}
```

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

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

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

4.4.9 静态模型

有些时候，数据表的数据一旦添加后就不再容易变化，我们更希望把这样的模型数据静态化，而不需要再次访问数据库。ThinkPHP 支持静态模型的概念，一旦把模型设置为静态，那么会在第一次初始化的时候获取数据表的全部数据，并生成缓存，以后不会再连接数据库。而只需要直接访问模型的 dataList 数据即可。

例如，我们把分类 CategoryModel 设置为静态模型

```
class CategoryModel extends Model  
  
    $staticModel = true;  
  
}
```

一旦初始化完成后，我们就可以用 S('Category')来获取 Category 数据表的数据了，并且如果我们以后再次访问 CategoryModel 对象的话，也可以直接获取到缓存的数据列表，但是这个时候已经不能再进行数据更改和其它数据库操作了。如果需要更新数据，请首先把 staticModel 设置为 false 再进行操作。

4.4.10 视图模型

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

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

```
class BlogViewModel extends Model  
  
{
```

```
var $viewModel = true;

var $viewFields = array(

    'Category'=>array('title'=>'categoryName'),

    'User'=>array('name'=>'userName'),

    'Blog'=>array('id','name','title'),

);

}
```

接下来，我们就可以对视图对象进行操作了

```
$Model = D("BlogView");
```

```
$Model->findAll();
```

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

```
$viewCondition = array(

    'Blog.categoryId'=>array('egf','Category.id'),

    'Blog.userId'=>array('egf','User.id'),

);
```

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

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

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

目前的视图模型仅仅针对查询操作，更多操作正在完善中～

4.4.11 单表继承

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

在数据库里面，我们只有一个 **User** 数据表，该数据表包括了我們需要的其它单表继承类型的所有字段，定义 **UserModel** 模型的时候我们和其它的一样，简单定义就行了。

```
UserModel extends Model{  
}
```

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

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

同样的方式，我们来定义 **Manager** 数据模型：

```
ManagerModel extends UserModel{  
    // 我们需要指定所对应的数据表名称  
    public function getTableName(){  
        return $this->tablePrefix.'user';  
    }  
    protected $_auto = array (  
        array('type','2','ADD'), // 写入数据的时候默认把 type 字段设置为 2 这里用来表示 Manager 类型  
    );
```

```
}  
  
$data['name'] = 'liu21st';  
  
$data['email'] = 'liu21st@gmail.com';  
  
$data['score'] = 80; // 对于 Member 类型我们需要记录 score 字段  
  
$Member->create($data);  
  
$Member->add();  
  
$data['name'] = 'thinkphp';  
$data['email'] = 'thinkphp@gmail.com';  
  
$data['dept'] = '开发部门'; // 对于 Manager 类型我们需要记录 dept 字段  
  
$Manager->create();  
  
$Manager->add();  
  
$Member->first(); // 查找第一个 Member 数据
```

4.4.12 表间关联

ThinkPHP 支持数据表的关联操作，目前支持的关联关系包括下面三种：

- 1、ONE_TO_ONE (包括 HAS_ONE 和 BELONGS_TO)
- 2、ONE_TO_MANY (包括 HAS_MANY 和 BELONGS_TO)
- 3、MANY_TO_MANY

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

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

```
// 每个用户有多个文章
```



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

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

默认规则

外键的默认规则是当前数据对象名称_id，例如：

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

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

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

多对多的中间表默认表规则如下：

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

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

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

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

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

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

4.4.13 锁机制

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

悲观锁（`Pessimistic Locking`）

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

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

```
// 启用悲观锁功能
```

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

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

乐观锁（ Optimistic Locking ）

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

ThinkPHP 也可以支持乐观锁机制，要启用乐观锁，只需要定义模型类的 `optimLock` 属性，并且在数据表字段里面增加相应的字段就可以自动启用乐观锁机制了。默认的 `optimLock` 属性是 `lock_version`，也就是说如果要在 `User` 表里面启用乐观锁机制，只需要在 `User` 表里面增加 `lock_version` 字段，如果有已经存在的其它字段作为乐观锁用途，可以修改模型类的 `optimLock` 属性即可。如果存在 `optimLock` 属性对应的字段，但是需要临时关闭乐观锁机制，把 `optimLock` 属性设置为 `false` 就可以了。

4.4.14 更新检测

如果提交对没有任何更改的数据进行更新，系统会进行检测，而不会实际进行数据库更新操作，避免多余的数据库操作。

4.4.15 文本字段

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

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

```
$blobFields = array('content');
```

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

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

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

4.5 模板输出

4.5.1 模板结构

在 ThinkPHP 框架中的项目模板文件目录是 `Tpl` 目录，在目录下面是模板主题目录，默认的模板名称是 `default`（可以在项目配置文件中设置），然后就是以每个模块名称命名的目录，存放模块对应的每个操作的模板文件，模板文件的后缀可以由项目配置文件设置。

ThinkPHP 的模板结构和系统是 100% 分离的，可以独立制作和预览，而没有任何依赖，除了模板标签的输出之外。模板的结构请遵循下面的规范，以确保模板读取正常。

| - `Tpl` 模板根目录（位于项目目录下面）

| -- `default` 主题目录（主题目录下面存放各个模块目录）

| ----- `Index` 模块目录（每个模块目录下面存放模块的操作模板文件）

| ----- `index.html` `index` 操作模板文件

| ----- `add.html` `add` 操作模板文件

| ----- `Layout` 布局模板目录

| ----- `Public` 公共目录（模板的外部资源文件引用通过该相对路径访问）

（以下目录可以自行修改，不影响模板读取）

| ----- `images` 建议图像目录

| ----- `css` 建议样式目录

| ----- `js` 建议脚本目录

例如

Tpl/default/user/index.html 当前项目的 user 模块中的 index 操作方法

Tpl/default/user/add.html 当前项目的 user 模块中的 add 操作方法

注意：只有需要显示的 Action 方法才需要定义模板文件

按照上面的模板结构，系统可以自动定位需要渲染的模板文件，而不需要手动指定。

4.5.2 定义模板

定义一个模板文件相当简单，例如：

```
<html>

<head>

<title>My Blog</title>

</head>

<body>

    <h1>Hello, {$name}! Welcome to my Blog!</h1>

</body>

</html>
```

模板文件的后缀可以通过 `TEMPLATE_SUFFIX` 设置，默认的设置就是 `.html`。所以，你可以把模板文件当作简单的 `Html` 文件一样编辑，并且可以直接预览。ThinkPHP 的模板文件根据你使用的模板引擎而有所区别，系统内置了一套模板引擎可以让你方便地通过标签来定义，可以完成包括变量输出、条件控制、循环迭代等复杂的功能，关于更加详细的标签使用参考模板指南。

4.5.3 如何渲染

最常用的模板渲染方式就是使用 `display` 方法，ThinkPHP 会自动定位当前操作的模板文件，所以 `display` 方法可以无需带任何参数，除非你需要指定另外的模板文件进行渲染。模板文件参数可以是一个完整的模板文件名，也可以是想要的操作名，例如：

假如当前操作方法是 `read`，当前模块是 `User`

`$this->display();` 表示渲染当前模块的 `read` 操作模板

`$this->display('edit');` 表示渲染当前模块的 `edit` 操作模板

`$this->display('Member:edit');` 表示渲染 Member 模块的 edit 操作模板

`$this->display('blue@User:edit');` 表示渲染 blue 模板主题的用户模块的 edit 操作模板

除了可以指定模板文件名之外，`display` 方法还可以指定输出的编码，如果没有指定，默认为配置文件设置的输出编码，例如，我们需要输出一个 GBK 的页面编码

```
$this->display('edit','gbk');
```

`display` 方法通常用于输出页面模板，`fetch` 方法可以获取要渲染的模板内容而不进行输出，而是返回要输出的内容，再进行其它的操作，其它参数的使用和 `display` 方法一致，例如：

```
$content = $this->fetch();
```

4.5.4 变量赋值

无论你是否使用内置的模板引擎，模板变量的赋值方式是统一的。

模板变量赋值通过 `assign` 方法，只有赋值的模板变量才能在模板文件中输出。在进行模板变量赋值的时候，可以对任何变量进行赋值，由模板标签来决定输出何种类型的，例如：

```
$this->assign('vo',$vo);
```

```
$this->display();
```

并且，赋值具有智能化和批量赋值，默认情况下第一个参数是要在模板中输出的变量名称，而第二个参数是变量的值，但是如果没有指定第二个参数，那么会对第一个参数进行判断，如果是索引数组，则自动进行批量赋值。例如：

```
$tmpl = array();
```

```
$tmpl['var1'] = 'value1';
```

```
$tmpl['var2'] = 'value2';
```

```
$this->assign($tmpl);
```

```
$this->display();
```

上面代码会自动赋值两个模板变量 `var1` 和 `var2`，用来在模板文件中输出。

作为安全性考虑，没有赋值的模板变量是不能用于输出的，如果使用内置的模板引擎，有些特殊的系统变量可以不需要通过赋值就可以直接在模板文件中输出，详细请参考模板引擎指南。

4.5.5 布局模板

普通的模板必须是每个模块的每个操作都要单独定义，哪怕是相同的表现，只是数据不同，布局模板可以为类似的模板提供统一的定义，然后在控制器里面定义好模板的不同部分进行输出。当然，即使不借助布局模板，也一样可以完成我们想要的功能，只不过布局模板提供了一个规范，而且支持特殊的标签可以直接读取其它模块的操作模板。

例如，User 模块的 `add` 和 `edit` 操作都必须定义单独的模板文件，假如你的 `add` 操作和 `edit` 操作的界面非常相似，你可以公用一个统一的模板文件来渲染输出。假如我们统一使用 `edit` 模板文件，可能在 `add` 操作的 Action 定义会采用下面的方式输出：

```
$this->display('edit');
```

那么，这样在执行 `add` 操作的时候，会读取 `edit.html` 模板文件输出，这样就起到了模板共用的目的。

如果使用布局模板，可以完成更强大的公共模板功能，例如，我们可以这样定义一个页面的布局模板：

```
<html>

<title>{$title}</title>

<body>

{$header}

{$main}

{$footer}

</body>

</html>
```

布局模板文件位于 `layout` 目录下面，和模块目录同级，要渲染一个布局模板，使用

```
$this->display('layout::list');
```

或者 `$this->layout('list');`

布局模板除了定义公共变量外，还可以引入其它模块的操作模板，例如

```
<html>

<title>{$title}</title>

<body>
```



```
<layout name="User:header" />
```

```
<layout name="User:main" />
```

```
<layout name="User:footer" />
```

```
</body>
```

```
</html>
```

更多的模板功能请参考模板引擎章节的说明。

4.5.6 使用第三方模板

ThinkPHP 默认的方式是使用 ThinkTemplate 模板引擎作为系统的模板引擎支持，你可以修改 `TMPL_ENGINE_TYPE` 配置参数类使用第三方的模板引擎，默认为 Think，官方已经提供了包括 Smarty、TemplateLite、EaseTemplate、DzTemplate 模板引擎在内的其它模板引擎插件，你还可以自己扩展自己需要的模板引擎支持。

以 Smarty 模板引擎的使用为例，我们在

如果你是一个模板引擎的反对者，还可以设置 `TMPL_ENGINE_TYPE` 为 PHP，就可以免去模板引擎的编译和解析的麻烦。事实上，内置的 ThinkTemplate 也支持直接在模板里面使用 PHP 代码。无论使用何种模板引擎，模板变量的赋值方式不变，转换工作由模板引擎插件来完成。

4.6 AJAX 支持

系统内置了 Ajax 的提交和数据返回支持，并且内置了一个 ThinkAjax 的 AJAX 操作的 JS 类库。

使用 ThinkAjax 类库进行 Ajax 提交非常方便，支持 HTML 事件绑定、表单提交、定时执行等 Ajax 操作。

支持 JSON 和 XML 方式返回客户端，当然，您也一样可以使用和扩展任何其他 Ajax 类库来进行 Ajax 操作，包括 JQuery 和 Prototype 等比较典型的 JS 类库包。

这里我们只介绍下 ThinkAjax 类库的使用

要使用 ThinkAjax 类库，需要加载相关的 JS 类库：

```
<script language='JavaScript' src='/Js/prototype.js'></script>
```

```
<script language='JavaScript' src='/Js/mootools.js'></script>
```

```
<script language='JavaScript' src='/Js/ThinkAjax.js'></script>
```


// 发送 Ajax 请求

send(提交 URL,提交参数,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新显示效果)

// 发送表单 Ajax 操作

sendForm(表单对象或者 ID,提交 URL,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新显示效果)

// 绑定 Ajax 操作到 HTML 元素和事件

bind(源 HTML 元素或者 ID,绑定事件,提交 URL,提交参数,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新效果)

// 页面加载时候进行 Ajax 操作

load(提交 URL,提交参数,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新效果)

// 定时执行 Ajax 操作

repeat(提交 URL,提交参数,执行间隔,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新效果)

// 延时执行 Ajax 操作

time(提交 URL,提交参数,延时时间,返回响应方法,结果显示 DIV 对象或者 ID,提示信息,更新效果)

示例:

发送 Ajax 请求

```
ThinkAjax.send('__URL__/insert','ajax=1&title=ThinkPHP',addComplete);
```

提交表单 Form1 的数据

```
ThinkAjax.sendForm('Form1','__URL__/insert',addComplete)
```

绑定 HTML 元素的 AJAX 操作

```
ThinkAjax.bind('CategoryId','mouseover','__URL__/over');
```

在发送请求到后台之后，AJAX 数据返回我们使用 Action 类的 AjaxReturn 方法，该方法支持三个参数，包括

AjaxReturn(\$data,\$info,\$status,\$type)

其中\$**data** 表示返回客户端的数据，如果是 JSON 方式返回数据，会自动进行 JSON 编码后返回，如果是 XML 方式，就会自动进行 XML 编码后返回。

\$**info** 是返回的提示信息

`$status` 是返回的状态信息

`$type` 指定返回的数据类型，默认是 JSON 格式数据

例如：

```
$this->ajaxReturn($data,'更新成功',1);
```

在后台使用 `AjaxReturn` 方法返回数据到客户端后，我们可以定义返回响应方法来处理返回的数据，例如：

```
ThinkAjax.send('__URL__/_insert','ajax=1&title=ThinkPHP',addComplete);
```

定义 `addComplete` 数据返回的响应方法

```
function addComplete(data,status){  
    if (status==1) Alert(data.title);  
}
```

响应处理方法有三个参数，分别是返回数据 `data`、返回状态 `status` 和返回信息 `info`，可以根据需要来处理。

4.7 异常处理

ThinkPHP 定义了 `throw_exception` 方法，用于手动抛出异常。ThinkPHP 的 `throw_exception` 方法兼容了 PHP4 和 PHP5 的异常处理机制，因此，无论是 PHP4 和 PHP5 环境下面，都使用 `throw_exception` 方法手动抛出异常，当然，如果你使用的是 PHP5 以上版本，同样也可以使用 `throw` 关键字来抛出异常。

`throw_exception` 方法支持三个参数：

`$msg` 异常信息，必须

`$type` 异常类型，即异常类的名称，默认是系统异常基础类 `ThinkException`

`$code` 异常代码 默认为 0

如果指定的异常类型不存在，系统自动调用 `halt` 方法直接输出异常信息文字，而不输出异常详细信息。

下面是一些使用例子：

```
throw_exception('新增失败');
```

```
throw_exception('信息录入错误','InfoException');
```

异常页面的模板是可以修改的，通过设置 `EXCEPTION_TMPL_FILE` 配置参数来修改系统默认的异常模板

文件，如果没有定义，则采用系统内置的异常模板文件，该模板文件位于 `Think.Exception` 目录下面的 `ThinkException.tpl.php` 文件。

另外一种方式是配置 `ERROR_PAGE` 参数，把所有异常和错误都指向一个统一页面，从而避免让用户看到异常信息，通常在部署模式下面使用。

如果需要，我们建议在项目的类库目录下面增加 `Exception` 目录用于专门存放异常类库，以更加精确地定位异常。

4.8 日志和调试

4.8.1 日志处理

ThinkPHP 内置了日志处理功能，包括系统异常和错误和调试信息，以及 SQL 记录，日志文件分别对应为 `WEB_LOG_ERROR`、`WEB_LOG_DEBUG` 和 `SQL_LOG_DEBUG` 三种类型，对应的日志文件名称为：

`systemErr.log` 主要用于 `WEB_LOG_ERROR` 类型日志

用于记录系统异常，通常为抛出异常或者捕获严重错误后自动记录

`systemOut.log` 主要用于 `WEB_LOG_DEBUG` 日志类型

用于记录调试信息和页面的一些非严重错误记录，调试信息一般为 `system_out` 方法写入。

`systemSql.log` 主要是用于 `SQL_LOG_DEBUG` 日志类型

记录执行过程中的 SQL 语句和执行时间，便于进行分析和优化。

日志文件的写入统一使用 `Log::Write($message,$type,$file)` 静态方法

`$message` 是要记录的日志信息

`$type` 就是日志类型

`$file` 日志文件位置和名称，该参数可以改变系统默认的日志文件命名。

设置 `WEB_LOG_RECORD` 为 `true` 就可以启用日志记录功能，日志文件的命名规则是前面增加日期前缀，原则上是一天的同类型的日志记录在一个文件里面，您可以随时查看日志文件，例如：

`07_09_21_systemOut.log` // 2007 年 9 月 21 日的错误日志文件

`07_12_03_systemSql.log` // 2007 年 12 月 3 日的 SQL 日志文件

`07_02_03_systemSql.log` // 2007 年 2 月 3 日的异常日志文件

可以设置 `LOG_FILE_SIZE` 参数来限制日志文件的大小，超过大小的日志会形成备份文件。备份文件的格式是在当前文件名前面加上备份的时间戳，例如：

`1189571417-07_09_12_systemSql.log` 备份的 SQL 日志文件

在系统的调试模式中，系统的所有异常和错误都会记录到系统日志中，在正式部署应用后，您可以关闭调试模式，这样系统就不会自动完成日志记录，除非你自己触发日志写入。

系统对项目单独记录日志，所以查看的时候请注意定位到某个项目目录下。

如果您的应用组件需要记录特殊的日志，也可以调用（或者扩展）该方法来完成。

4.8.2 调试模式

系统支持部署模式和调试（开发）模式，在开发模式的情况下，可以定义调试模式配置文件，和部署模式下面的项目配置文件区别开来。并且，做到调试模式的配置参数由项目自己来配置，如果没有配置项目的调试模式配置文件，那么系统会有一个默认的调试模式配置参数，例如启用日志记录、模板缓存有效期缩短、关闭数据库缓存、记录 SQL 日志、显示运行信息、显示页面 Trace 信息等。调试通过后，你只需要关闭项目配置文件里面的调试模式参数，就可以正式部署到生产环境了。并且，调试模式下面还可以定义不同的数据库配置参数，来和部署环境的数据库进行区分，方便测试。

要启用调试模式，请在项目配置文件里面设置 `DEBUG_MODE` 为 `true`，正式部署后设置该参数为 `false` 就行了。

每个项目可以定义自己的调试配置文件，位于项目配置目录下面，名称为 `_debug.php`，和项目配置文件相同的配置项可以无需配置，只需要单独定义不同的配置参数即可，如果要加入调试模式下面的数据库配置，使用下面的方式：

Return array(

`'DB_TYPE'=>'pdo',`

`'DB_DSN'=>'mysql:host=localhost;dbname=thinkPHP',`

`'DB_USER'=>'root',`

`'DB_PWD'=>'admin',`

`'DB_PORT'=>'3306',`

`'DB_PREFIX'=>'think',`

..... 其它配置参数

```
}
```

项目配置文件里面同样可以用这样的方式定义数据库配置信息，可以省去在入口文件里面加载数据库配置文件。

4.8.3 调试方法

除了本身可以借助一些开发工具进行调试外，ThinkPHP 还提供了一些内置的调试函数和类库。

`halt($msg)` //输出错误信息，并中止执行

`system_out($msg)` //输出调试信息到日志文件

`dump($var, $echo=true, $label=null)` //输出变量信息

`debug_start($label=')` //记录调试开始时间

`debug_end($label=')` //输出调试范围运行时间（相同 label 属于一个调试范围）

`get_include_contents($filename)` //获取载入文件的内容

更高级的调试方法是使用 Debug 类

`Debug::mark($name);` // 标记一个调试位置

`Debug::useTime($start,$end);` // 返回区间所用的时间

`Debug::useMemory($start,$end);` // 返回区间所用的内存

4.8.4 页面 Trace

页面 Trace 功能是新版增加的一个用于开发调试的辅助手段。可以，实时看到当前页面的操作的请求信息、运行情况、SQL 执行、错误提示等，启用调试模式的话，页面 Trace 功能会默认开启（除非在项目的调试配置文件中关闭），并且系统默认的 Trace 信息包括：当前页面、请求方法、通信协议、请求时间、用户代理、会话 ID、运行情况、输出编码、模板编译、SQL 记录、错误记录。如果需要扩展自己的 Trace 信息，有下面几种方式：

第一种方式：在当前项目的配置目录下面定义 `_trace.php` 文件，返回数组方式的定义，例如：

```
return array(
    '当前页面'=>$_SERVER['PHP_SELF'],
    '通信协议'=>$_SERVER['SERVER_PROTOCOL'],...);
```

在显示页面 **Trace** 信息的时候会把这个部分定义的信息追加到系统默认的信息之后，这种方式通常用于 **Trace** 项目的公共信息。

第二种方式：在 **Action** 方法里面使用 **trace** 方法来增加 **Trace** 信息，该部分可以用于系统的开发阶段调试。例如：

```
$this->trace('执行时间',$runTime);  
$this->trace('Name 的值',$name);  
$this->trace('GET 变量',dump($_GET,false));
```

4.9 文件上传

4.9.1 上传概述

上传类使用 **ORG** 类库包中的 **Net.UpdateFile** 类，**ThinkPHP** 内置的 **Action** 操作里面（主要是 **insert** 和 **update** 操作，其他操作可以相应实现）实现了自动识别是否存在文件上传，如果存在会自动进行处理。而上传类要做的仅仅是文件上传的过程，其他功能需要依赖系统类库或者相应类库。系统在 **Action** 类里面对文件上传设置了很多灵活的参数以便进行更细致的控制。下面我们通过几种常用的例子分别来描述下如何使用 **UploadFile** 类。最新版本的上传类包含的功能如下（有些功能需要结合 **ThinkPHP** 系统其他类库）：

- ✧ 基本上传功能
- ✧ 批量上传
- ✧ 自动生成图片缩略图
- ✧ 自定义参数上传
- ✧ 上传检测
- ✧ 支持覆盖方式上传
- ✧ 支持上传类型、附件大小、上传路径定义
- ✧ 采用 **IFrame** 方式的 **Ajax** 上传机制支持
- ✧ 支持上传文件命名规则
- ✧ 支持对上传文件的 **Hash** 规则
- ✧ 可设置是否保存附件数据到数据库

✧ 支持附件的版本功能

4.9.2 基本上传功能

基本上，在 ThinkPHP 中简单的上传功能无需进行特别处理，而全部有内置操作实现了。要做的仅仅是在表单中添加文件上传框和设置 `enctype="multipart/form-data"` 属性即可。当然，这和框架的架构和数据结构有关，因为 ThinkPHP 的上传数据表是单独的，上传文件数据表中有两个关键的用于记录对应数据的字段：`module` 和 `recordId`，其实 `module` 也就是某个数据表，而 `recordId` 也就是该数据表对应的数据 ID。在其他任何需要上传的数据表中可以方便地查询到属于自己的附件列表，就是采用这种机制和结构，令得 ThinkPHP 的上传变得简化了。

下面就是实现代码：

```
<form METHOD=POST action="__URL__/_upload/" enctype="multipart/form-data" >
<INPUT TYPE="text" NAME="name" >
<INPUT TYPE="text" NAME="email" >
<INPUT TYPE="file" name="photo">
<INPUT TYPE="submit" value="保存">
</form>
```

上面的表单，在保存用户数据的同时包括了一个照片文件上传，使用普通方式提交到后台后，系统会自动会把用户数据保存在用户数据表中，而把上传的文件保存到附件数据表，并记录了对应的用户数据表的名称和编号。系统的 `Action` 类内置了一个 `upload` 操作方法，用于上传附件操作。

系统的附件数据表结构如下，可以作为设计的参考：

```
CREATE TABLE `thinkphp_attach` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(255) NOT NULL,
  `type` varchar(255) default NULL,
  `size` varchar(20) NOT NULL,
  `extension` varchar(20) NOT NULL,
  `savepath` varchar(255) NOT NULL,
  `savename` varchar(255) NOT NULL,
```

```
`module` varchar(100) NOT NULL,  
`recordId` int(11) NOT NULL,  
`userId` int(11) unsigned default NULL,  
`uploadTime` int(11) unsigned default NULL,  
`downCount` mediumint(9) unsigned default NULL,  
`hash` varchar(32) NOT NULL,  
`version` int(6) unsigned,  
`remark` varchar(255) NOT NULL,  
`verify` varchar(32) NOT NULL,  
PRIMARY KEY `id`  
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

下次取得数据的时候，使用下面的方式获取属于该记录的附件列表：

```
//读取附件信息  
$Attach = D('Attach');  
$list = $Attach->findAll("module='User' and recordId='$id'");  
//模板变量赋值  
$this->assign("attachs",$list);
```

也可以获取某个用户的上传附件信息

```
$Attach->findAll("userId=".$userId);
```

4.9.3 批量上传

ThinkPHP 上传类支持多文件上传，而这些仅仅是在客户端增加多个文件上传框而已，后台会自动获取所有的文件上传，并一一进行上传和保存数据操作，并且过滤无效的上传。批量上传的一个例子：

假设用户往自己的图片库里面添加多个图片：

```
<form METHOD=POST action="__URL__/_action/" enctype="multipart/form-data" >  
<INPUT TYPE="file" name="photo1" >  
<INPUT TYPE="file" name="photo2" >
```



```
<INPUT TYPE="file" name="photo3" >  
<INPUT TYPE="submit" value="上传图片" >  
</form>
```

另外，批量上传的表单还可以使用下面的方式，系统能够自动识别，和上面的方式等效。

```
<INPUT TYPE="file" name="photo[]" >
```

上传文件的个数并无限制，ThinkPHP 管理后台还实现了一个动态增加文件上传的功能。通过该方式可以方便地进行多文件批量上传。



4.9.4 Ajax 文件上传

通过简单的参数设置就可以把文件上传改装成 AJAX 方式（Iframe 实现方式），而你要做的仅仅是添加下面代码：

```
<form METHOD=POST id="upload" action="__URL__/upload/" enctype="multipart/form-data" target="ajaxUpload">  
<iframe name="ajaxUpload" src="" frameborder="0" SCROLLING="no" style="display:none"></iframe>  
<INPUT TYPE="hidden" name="ajax" value="1">  
<INPUT TYPE="hidden" name="_uploadFormId" value="upload">  
<INPUT TYPE="hidden" name="_uploadFileResult" value="result">  
<INPUT TYPE="hidden" name="_uploadResponse" value="uploadComplete">  
</form>
```

`_uploadFormId` 用于设置上传表单 id，用于在上传成功后重置表单，避免重复上传。在 `_uploadFileResult` 变量中设置返回提示的层 id，在 `_uploadResponse` 参数中设置文件上传返回数据的处理方法。该方法返回两个参数：id 和 name，如果有多文件上传，使用逗号分割多个返回值。ThinkPHP 框架的 Action 类中的 `ajaxUploadResult` 方法对 Ajax 文件上传的信息返回提供支持。

例如，第一个例子上传后希望更新照片，使用下面的方法定义：

```
1. function uploadComplete(id,name){  
2.     $('photo').innerHTML = '<IMG SRC="__PUBLIC__/_Images/user/' + name + '" class="shadow" BORDER=  
    "0" ALT="" align="left">';  
3. }
```

下面的示例是 AJAX 文件上传的实现画面，左边图片会上传成功后自动更新。



4.9.5 自动生成缩略图

如果希望在上传过程自动为图片文件生成缩略图，ThinkPHP 的 `UploadFile` 类也可以轻松实现，而且不需要你多特殊添加缩略图处理代码。要做的也仅仅是在客户端添加如下参数：

```
// 设置是否需要生成图片缩略图，仅对图片上传有效  
<INPUT TYPE="hidden" name="_uploadImgThumb" value="1">  
// 生成缩略图的最大宽度  
<INPUT TYPE="hidden" name="_uploadThumbMaxWidth" value="45">  
// 生成缩略图的最大高度
```

```
<INPUT TYPE="hidden" name="_uploadThumbMaxHeight" value="45">
```

设置后系统在上传后会自动生成相同格式的缩略图。系统默认的缩略图路径是上传文件所在目录，并且在文件中后面添加_thumb 以标识缩略图文件。缩略图路径可以在项目配置文件中配置。

生成多缩略图

ThinkPHP 支持对上传的图片生成多缩略图，TOPTThink 社区的头像功能就是多缩略图功能的例子，使用起来也非常简单。下面的代码是 TOPTThink 社区上传头像的部分缩略图代码：

```
<INPUT TYPE="hidden" name="_uploadImgThumb" value="1">
<INPUT TYPE="hidden" name="_uploadThumbSuffix" value="_big,_small,_min">
<INPUT TYPE="hidden" name="_uploadThumbMaxWidth" value="75,32,16">
<INPUT TYPE="hidden" name="_uploadThumbMaxHeight" value="75,32,16">
```

上面的例子表示生成三个大小的缩略图，并规定了缩略图文件名后面添加的后缀，和三种缩略图的宽高尺寸。

4.9.6 更多设置

ThinkPHP 在 Action 来中还提供了和 UploadFile 类的上传设置接口，方便在客户端进行更多的参数设置进行上传控制。

下面列举下主要的参数，更多的参数可以参考框架的 Action 类中的_upload 方法。

```
// 设置覆盖方式上传
<INPUT TYPE="hidden" name="_uploadReplace" value="1">

// 设置允许上传文件类型
<INPUT TYPE="hidden" name="_uploadFileType" value="jpg,gif,png,swf" >

// 上传文件保存目录，要注意设置可写权限
<INPUT TYPE="hidden" name="_uploadSavePath" value="/Public/Images/user/" >

// 上传文件名命名规则，支持函数，例如 time uniqid com_create_guid 系统默认设置为 uniqid 保证上传文件名不会重复，如果不存在设置函数，则使用规则字符串作为上传文件名

// 如果上传规则为空 则表示直接使用上传的文件名保存
<INPUT TYPE="hidden" name="_uploadSaveRule" value="time">
```

```
// 设置上传文件大小
<INPUT TYPE="hidden" name="_uploadFileSize" value="20480" >

// 设置是否保存附件信息到数据库 以下设置都是基于数据库方式保存附件信息
<INPUT TYPE="hidden" name="_uploadRecord" value="1">

// 设置上传数据表，默认的上传数据记录在当前模块表中
<INPUT TYPE="hidden" name="_uploadFileTable" value="user">

// 设置上传文件对应的数据编号，通常不用设置，除非特别需要
<INPUT TYPE="hidden" name="_uploadRecordId" value="">

// 设置上传用户 id，通常不用设置，系统自动获取当前登录用户编号
<INPUT TYPE="hidden" name="_uploadUserId" value="{ $user.id }">

// 设置附件 id，通常用于覆盖模式下的对一条记录的多个附件进行替换
<INPUT TYPE="hidden" name="_uploadFileId" value="120">

// 设置是否记录附件版本，通常用于覆盖模式下的附件版本记录
<INPUT TYPE="hidden" name="_uploadFileVersion" value="1">
```

4.10 权限控制

4.10.1 权限概述

企业级的应用是离不开安全保护的，ThinkPHP 以基于 Spring 的 Acegi 安全系统作为参考原型，并做了简化，以适合目前的 ThinkPHP 结构，提供了一个多层的、可定制的安全体系来为应用开发提供安全控制。安全体系中主要有：

安全拦截器

认证管理器

决策访问管理器

运行身份管理器

安全拦截器

安全拦截器就好比一道道门，在系统的安全防护系统中可能存在很多不同的安全控制环节，一旦某个环节你未通过安全体系认证，那么安全拦截器就会实施拦截。

认证管理器

防护体系的第一道门就是认证管理器，认证管理器负责决定你是谁，一般它通过验证你的主体（通常是一个用户名）和你的凭证（通常是一个密码），或者更多的资料来做到。更简单的说，认证管理器验证你的身份是否在安全防护体系授权范围之内。

访问决策管理

虽然通过了认证管理器的身份验证，但是并不代表你可以在系统里面肆意妄为，因为你还需要通过访问决策管理这道门。访问决策管理器对用户进行授权，通过考虑你的身份认证信息和与受保护资源关联的安全属性决定是否可以进入系统的某个模块，和进行某项操作。

例如，安全规则规定只有主管才允许访问某个模块，而你并没有被授予主管权限，那么安全拦截器会拦截你的访问操作。

决策访问管理器不能单独运行，必须首先依赖认证管理器进行身份确认，因此，在加载访问决策过滤器的时候已经包含了认证管理器和决策访问管理器。

为了满足应用的不同需要，ThinkPHP 在进行访问决策管理的时候采用两种模式：登录模式和即时模式。

登录模式，系统在用户登录的时候读取改用户所具备的授权信息到 **Session**，下次不再重新获取授权信息。也就是说即使管理员对该用户进行了权限修改，用户也必须在下次登录后才能生效。

即时模式就是为了解决上面的问题，在每次访问系统的模块或者操作时候，进行即使验证该用户是否具有该模块和操作的授权，从更高程度上保障了系统的安全。

运行身份管理器

运行身份管理器的用处在大多数应用系统中是有限的，例如某个操作和模块需要多个身份的安全需求，运行身份管理器可以用另一个身份替换你目前的身份，从而允许你访问应用系统内部更深处的受保护对象。这一层安全体系目前的 **RBAC** 插件中尚未实现。

4.10.2 使用 RBAC 组件

要启用 RBAC 组件，请在项目配置文件中设置

USER_AUTH_ON 为 True

并设置认证类型 USER_AUTH_TYPE

1 普通认证（认证一次）

2 高级认证（实时认证）

不设置默认为 1

认证识别号 USER_AUTH_KEY 是用于检查用户是否经过身份认证的标识，一旦用户经过系统认证，系统会把该用户编号保存在\$_SESSION 中

为了满足应用系统的需要，RBAC 组件中可以设置

REQUIRE_AUTH_MODULE 需要认证的模块

NOT_AUTH_MODULE 无需认证的模块

多个模块之间用逗号分割

如果某个模块需要认证，但是用户还没有经过身份认证，就会跳转到

USER_AUTH_GATEWAY 认证网关，例如 /Public/login

验证地址就是： 项目入口文件 URL 地址/Public/login

假设认证网关的验证操作地址是/Public/CheckLogin，可以在 public 模块的 checkLogin 操作中采用如下方式进行认证：

```
<?php
// 生成认证 Map 条件
// 这里使用用户名、密码和状态的方式进行认证

$map = new HashMap();

$map->put("name",$_POST['name']);

$map->put("password",$_POST['password']);

$map->put("status",1);

$authInfo = RBAC::authenticate($map);

if(false === $authInfo) {
```

```
$this->assign('error','登录失败，请检查用户名和密码是否有误！');  
}  
  
// 设置认证识别号  
Session::set(USER_AUTH_KEY,$authInfo->id);  
  
//获取并保存用户访问权限列表  
RBAC::saveAccessList();  
  
// 登录成功，页面跳转  
$this->assign("message",'登录成功！');  
  
$this->assign("jumpUrl",'http://www.topthink.com.cn');  
}  
  
$this->forward();  
  
?>
```

RBAC 组件的常用方法:

1、委托身份认证方法

`RBAC::authenticate($map,$model='User',$provider=USER_AUTH_PROVIDER)`

方法是静态方法，支持三个参数，其中第一个认证条件\$map 是必须的，可以灵活地控制需要认证的字段。

第二个参数是进行认证的 Model 类，默认是 UserModel 类

第三个参数是委托方式 由 USER_AUTH_PROVIDER 设置委托认证管理器的委托方式，目前支持的是 DaoAuthenticionProvider 通过数据库进行认证。

在应用系统的开发过程中，只需要设置相关的配置项和添加上面的认证方法，其他的认证和决策访问就由 RBAC 组件的 AccessDecision 方法自动完成了。

系统会在执行某个模块的操作时候，首先判断该模块是否需要认证，如果需要认证并且已经登录，就会获取当前用户的权限列表判断是否具有当前模块的当前操作权限，并进行相应的提示。

接下来就是在框架总后台设置相关项目的模块和操作权限了。

首先，在总管理后台（ThinkPHP 自带的 Admin 项目）的节点管理添加相关项目、模块和操作，作为权限管理的节点。

如果需要设置公共的操作，可以使用 **Public** 模块，所有属于 **Public** 模块的操作对所有模块都有效。

添加完成项目管理节点后，就在权限管理里面对某个用户组设置相关权限（包括项目权限、模块权限和操作权限）

以后需要授权就把用户添加到某个权限组就可以了，同一个用户可以属于多个权限组。

授权和认证功能涉及到四个数据表，**DB_PREFIX** 为配置文件中设置的数据库前缀

DB_PREFIX_group 权限组表

DB_PREFIX_groupuser 组-用户关联表

DB_PREFIX_access 访问权限表

DB_PREFIX_node 权限节点表

2、获取用户的权限

```
RBAC::getAccessList(userId)
```

3、保存用户的权限信息

```
RBAC::saveAccessList($userId)
```

4、获取记录的授权信息

```
RBAC::getRecordAccessList($authId=null,$module="")
```

除了进行用户认证和模块操作的授权外，新版的 **RBAC** 组件还支持对记录的授权，可以设置数据表的某些记录的访问权限。使用 **RBAC::getRecordAccessList()**就可以了

4.11 插件机制

ThinkPHP 支持插件机制，包括模板引擎插件、过滤插件等。

第5部分 模板指南

之所以把模板的使用单独讲述，是因为模板标签的用法其实是相对独立的，但是内容又比较多，所以单独出来。ThinkPHP 框架内置了一个性能卓越的模板引擎 **ThinkTemplate**，内置的模版引擎提供了一定

程度的功能,虽然卓越但是需要一个熟悉和掌握的过程,如果你已经熟悉了另外一种模版引擎的使用,而目前的项目又不允许你花更多的时间来学习内置的模版引擎,没有关系,ThinkPHP 框架允许你使用第三方的模版引擎。目前官方已经提供了 Smarty 模版引擎的插件,已经有人给 ThinkPHP 开发了 TemplateLite、EaseTempalte 和 DzTemplate 模版引擎插件。而且对于自己熟悉的模版引擎来说,非常容易扩展类似的插件。如果你没有使用内置的模板引擎,则可以跳过本章内容。

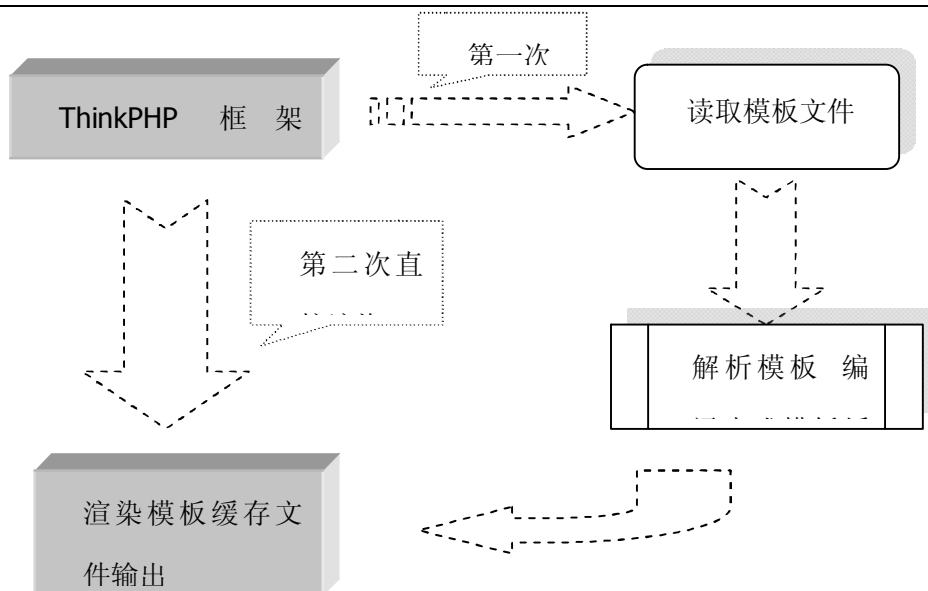
使用何种模板引擎,请设置 `TMPL_ENGINE_TYPE` 配置参数。

5.1 模板概述

ThinkTemplate 是一个使用了 XML 标签库技术的编译型模板引擎,支持两种类型的模板标签,使用了动态编译和缓存技术,而且支持自定义标签库。其特点包括:

- 1、支持 XML 标签库和普通标签的混合定义;
- 2、编译一次,下次直接运行而无需重新编译;
- 3、模板文件更新后,自动更新模板缓存;
- 4、自动加载当前操作的模板缓存文件,无需指定模板文件名称;
- 5、支持编码转换和 Content-Type 更换;
- 6、模板变量输出前缀支持,避免变量名称冲突;
- 7、模板常量替换,无需设置模板变量;
- 8、支持变量组合调节器和格式化功能;
- 9、支持替换其它模板引擎插件使用;
- 10、支持获取模板页面内容

ThinkTemplate 模板引擎的运作过程如图所示:



如果在 ThinkPHP 框架中使用的话，无需创建 `ThinkTemplate` 对象，`Action` 类会自动创建，只需要赋值并输出就行了。

```
$this->assign('vo',$vo);
```

```
$this->display();
```

ThinkPHP 内置模板引擎的模板标签有两种类型：第一种是普通标签，类似于 `Smarty` 的模板标签，在功能方面作了部分简化，增强了显示功能，弱化了逻辑控制功能；第二种是 `XML` 标签库形式，该模板技术是新版（0.8 版本以上才支持）新增的标签技术，有效地借鉴了 `Java` 的标签库技术，在控制功能和各方面都比较强大，而且允许自定义标签库，是新版 ThinkPHP 系统引入和推荐的模板标签技术。两种标签方式的结合使用，可以让您的模板定义功能相当强大。

ThinkPHP 架构的设计中模板和程序完全分离，一套模板文件就是一个目录，模板是标准 `html` 文件（可以配置成其它后缀，如 `.shtml`，`.xml` 等），可以单独预览。

由于使用了模板动态缓存技术，在您第一次运行某个模块的某个操作时候，对应的模板文件就会被缓存，下次读取的时候，无论是模板文件修改或者是缓存文件被删除，系统都会重新生成缓存文件。您还可以设置模板缓存的有效时间间隔，如每隔 10 分钟重新读取模板文件。模板动态缓存只是让您免

去每次重复编译模板的时间。

5.2 模板标签

模板文件可以包含普通模板标签和 XML 模板标签，并且完全支持直接使用 PHP 语句，如果你使用其它模板引擎作为页面输出，那么请遵守相关的模板标签规定。

ThinkPHP 系统的普通模板标签默认以 { 和 } 作为开始和结束标识，并且在开始标记紧跟标签的定义，如果之间有空格或者换行则被视为非模板标签直接输出。

例如：{\$name} {\$vo.name} {\$vo['name']|strtoupper} 都属于普通模板标签

要更改普通模板的起始标签和结束标签，请使用下面的配置参数：

TMPL_L_DELIM 模板引擎普通标签开始标记

TMPL_R_DELIM 模板引擎普通标签结束标记

普通模板标签主要用于模板变量输出、模板注释和公共模板包含。如果要使用其它功能，请使用 XML 模板标签，ThinkPHP 包含了一个基于 XML 和 TagLib 技术的模板标签，包含了普通模板有的功能，并且有一些方面的增强和补充，更重要的一点是新的标签库模板技术更加具有扩展性。新的 TagLib 标签库具有命名空间功能，ThinkPHP 框架内置了两个小型的标签库实现：CX 和 Html。

如果你觉得 XML 标签无法在正在使用的编辑器里面无法编辑，还可以更改 XML 标签库的起始和结束标签，请修改下面的配置参数：

TAGLIB_BEGIN 标签库标签开始标签

TAGLIB_END 标签库标签结束标记

XML 模板标签可以用于模板变量输出、文件包含、模板注释、条件控制、循环输出等功能，而且完全可以自己扩展功能。

5.3 模板变量

内置模板引擎对模板变量的支持相当充分，除了可以输出正常变量外，还可以输出特殊变量和对输出变量使用函数定义（并支持多个函数）。

模板变量的标签格式为

格式：{\$varname|function1|function2=arg1,arg2,### }

说明：

{ 和 \$ 符号之间不能有空格，后面参数的空格就没有问题

###表示模板变量本身的参数位置

支持多个函数，函数之间支持空格

支持函数屏蔽功能，在配置文件中可以配置禁止使用的函数列表

支持变量缓存功能，重复变量字串不多次解析

使用例子：

```
{webTitle|md5|strtoupper|substr=0,3 }
```

```
{number|number_format=2 }
```

如果在应用 Action 中需要输出某个变量，使用下面的方法：

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

系统只会输出设定的变量，其它变量不会输出，一定程度上保证了变量的安全性。

系统支持输出数组和对象属性，无论要输出的模板变量是数组还是对象，都可以用下列方式输出：

```
{user.name}
```

系统会自动判断要输出的变量，如果是多维数组或者多层对象属性的输出，请使用下面的定义方式：

```
{user['sub']['name']}
```

```
{user:sub:name}
```

如果要同时输出多个模板变量，可以使用下面的方式：

```
$array = array();
```

```
$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。

除了常规变量的输出外，模板引擎还支持系统变量和系统常量、以及系统特殊变量的输出。它们的输

出不需要对模板变量赋值。

系统变量（依然支持函数使用和大小写、空格），以 Think.打头，如

```
{Think.server.script_name } //取得$_SERVER 变量  
{Think.session.session_id|md5 } // 获取$_SESSION 变量  
{Think.get.pageNumber } //获取$_GET 变量  
{Think.cookie.name } //获取$_COOKIE 变量
```

系统常量

```
{Think.const.__FILE__}  
{Think.const.MODULE_NAME }
```

特殊变量，由 ThinkPHP 系统定义的常量

```
{Think.version } //版本  
{Think.now } //现在时间  
{Think.template|basename } //模板页面  
{Think.LDELIM } //模板标签起始符号  
{Think.RDELIM } //模板标签结束符号
```

配置参数

```
{Think.config.db_charset}
```

语言变量

```
{Think.lang.page_error}
```

我们还给一些常用的变量输出定义了快捷标签，他们分别是：

```
{:function(...)} 执行方法并输出返回值  
{~function} 执行方法不输出  
{@var} 输出 Session 变量
```

`{&var}` 输出配置参数

`{%var}` 输出语言变量

`{.var}` 输出 GET 变量

`{^var}` 输出 POST 变量

`{*var}` 输出常量

5.4 模板注释

模板支持注释功能，该注释文字在最终页面不会显示，仅供模板制作人员参考和识别。

格式：`{/* 注释内容 */}` 或 `{// 注释内容 }`

说明：在显示页面的时候不会显示模板注释，仅供模板制作的时候参考。

注意{和注释标记之间不能有空格。

5.5 公共模板

当页面需要包含公共文件的时候，可以通过下面的模板标签

格式：`{ include:Filename }`

说明：Filename 表示公共文件的名称（不包含后缀，因为模板文件后缀为可配置），Filename 默认在当前目录下寻找，但是完全支持相对路径访问，例如，下面的格式都是正确的。`{ include:header }` 和 `{ include:../public/header }`。该标签可以出现在模板页面的任何位置，也就是说可以包含除了头部文件和尾部文件之外的一些公共文件，就看网站页面的设计了。建议的方式是首先让美工写在一个页面文件中，然后调试完毕后在分成多个文件来引用，因为一旦使用引用标签后，模板页面就不能直接浏览到实际的效果了。加载公共模板文件后，模板引擎会重新对该页面中的模板标签进行解析，有意思的是你还可以在公共模板中再次包含公共文件，但是一定要注意不能循环包含。例如，

在 header.html 文件中包含了 menu 文件

```
{include:menu}
```


在 index.html 文件中则包含了 header 和 footer

```
{include:header}
```

这里是首页的内容

```
{include:footer}
```

在访问 `index` 操作方法的时候，模板首先读取 `index` 文件，并开始解析 `include:header`，在解析 `header` 文件的过程中又遇到 `include:menu` 标签，又开始解析，解析完成后再解析 `include:footer` 标签，在经过几层的嵌套包含解析后 `index` 文件最终被解析成一个缓存模板文件。

 **注意：**由于模板解析的特点，从入口模板开始解析，如果公共模板有所更改，模板引擎并不会重新编译模板，除非缓存已经过期。如果遇到比较大的更改，您可以尝试把模块的缓存目录清空，系统就会重新编译，并解析到最新的公共文件了。

5.6 布局模板

普通的模板必须是每个模块的每个操作都要单独定义，哪怕是相同的表现，只是数据不同，布局模板可以为类似的模板提供统一的定义，然后在控制器里面定义好模板的不同部分进行输出。当然，即使不借助布局模板，也一样可以完成我们想要的功能，只不过布局模板提供了一个规范，而且支持特殊的标签可以直接读取其它模块的操作模板。

例如，`User` 模块的 `add` 和 `edit` 操作都必须定义单独的模板文件，假如你的 `add` 操作和 `edit` 操作的界面非常相似，你可以公用一个统一的模板文件来渲染输出。假如我们统一使用 `edit` 模板文件，可能在 `add` 操作的 `Action` 定义会采用下面的方式输出：

```
$this->display('edit');
```

那么，这样在执行 `add` 操作的时候，会读取 `edit.html` 模板文件输出，这样就起到到了模板共用的目的。

如果使用布局模板，可以完成更强大的公共模板功能，例如，我们可以这样定义一个页面的布局模板：

```
<html>

<title>{$title}</title>

<body>

{$header}

{$main}

{$footer}

</body>

</html>
```

布局模板文件位于 `layout` 目录下面，和模块目录同级，要渲染一个布局模板，使用

```
$this->display('layout::list');
```

或者 `$this->layout('list');`

布局模板除了定义公共变量外，还可以引入其它模块的操作模板，例如

```
<html>

<title>{$title}</title>

<body>

<layout name="User:header" />

<layout name="User:main" />

<layout name="User:footer" />

</body>

</html>
```

5.7 标签库

ThinkPHP 包含了一个基于 XML 和 TagLib 技术的模板标签。基本上，新的标签技术和简单标签互为补充，包含了普通模板有的功能，并且有了一些方面的增强和补充，更重要的一点是新的标签库模板技术更加具有扩展性。新的 TagLib 标签库具有命名空间功能，ThinkPHP 框架内置了两个小型的标签库实现：CX 和 Html。

5.7.1 如何使用标签库

要在模板页面中使用 TagLib 标签库功能，需要在开始时候使用 `taglib` 标签导入需要使用的标签，防止以后标签库大量扩展后增加解析工作量，用法如下：

```
<tagLib name='cx,html' />
```

引入标签库后，就可以使用标签库定义的标签来定义模板了，例如：

//可以使用下面的模板标签定义。

```
<cx:var name='user' property='name' />
```

```
<cx:present name='user.name' ></cx:present>
```



```
<html:select options='name' selected='value' />
```

标签库使用的时候忽略大小写，因此下面的方式一样有效：

```
<CX:VAR NAME='user' property='name' />
```

实际上，ThinkPHP 框架模板引擎会默认加载 CX 标签库，所以下面的方式效果相同：

```
<tagLib name='html' />
```

并且，默认加载的 CX 库可以不使用 CX 命名空间前缀，也就是说

```
<var name='user' property='name' />
```

```
//等效于 <cx:var name='user' property='name' />
```

要使用 Html 标签库，需要添加 html 命名前缀，如

```
<html:link href='/path/to/common.js' />
```

```
//这是一个复杂的 DataGrid 组件的标签定义
```

```
<html:list id='checkList' style='list' name='action' checkbox='true' action='true' datasource='list' show='id:编号|8%,name:名称,title:显示名,status|getStatus:状态|8%' actionlist='edit:编辑,del:删除,forbid|resume:禁用|恢复' />
```

5.7.2 标签的混合使用

在附录的标签指南里面会详细描述每个标签的使用方法。

ThinkPHP 模板引擎的两种标签定义方式，其实，搭配使用两种定义方式能够带来更大的效率。

例如，用

```
<volist id='user' name='userList' >
```

```
{ $user.name }
```

```
{ $user.age }
```

```
{ $user.email }
```

```
</volist>
```

的方式似乎更加简单。

对于不太复杂的变量输出，建议多采用 `{ $var }` 方式，因为单纯从易用性方面而言，这种方式最简洁，而且功能也比较完善。

5.7.3 CX 标签库

CX 标签库主要用于输出 ThinkPHP 框架的变量、文件包含、条件判断、循环控制等结构。

主要有：

include 包含文件 支持的标签属性有 `file`

示例：

包含公共模块的 `header` 模板

```
<include file='Public:header' />
```

包含当前模块的 `edit` 模板

```
<include file='edit' />
```

包含 `blue` 模板主题下面的 `User` 模块的 `add` 模板

```
<include file='blue@User:add' />
```

comment 模板注释 无标签属性

```
<comment> 模板注释 </comment>
```

iterate 迭代因子输出

支持的标签属性有 `id` | `name` | `offset` | `length` | `empty`

```
<iterate name='list' id='vo'>
```

```
{ $vo.name }
```

```
</iterate>
```

volist 数据对象列表输出，循环内可以结合 `vo` 标签

支持的标签属性有 `id` | `name` | `offset` | `length` | `empty`

```
<volist name='list' id='vo'>
```

```
{ $vo.name }
```

```
</volist>
```

多重循环

```
<volist name='list' id='vo'>
```

```
<sublist name='vo.sub' id='sub'>
```

```
{${sub.name}}
```

```
</sublist>
```

```
</volist>
```

equal 判断是否相同

notequal 判断是否不同

支持的标签属性有 name | value

例如:

```
<equal name='vo.name' value='1'>输出</equal>
```

```
<notequal name='vo[\'name\']|strtoupper' value='PIC'>输出</notequal>
```

其它比较标签还有

eq neq gt egt lt elt

```
<gt name='vo.name' value='1'></gt>
```

present 判断是否定义

notpresent 判断是否没有定义

支持的标签属性有 name

```
<present name='vo'>输出</present>
```

foreach 循环标签

```
<foreach name='list' item='User' key='key' >
```

```
{${User.name}}
```

```
</foreach>
```

多重循环

```
<foreach name='list' id='vo'>

<subeach name='vo.sub' id='sub'>

{$sub.name}

</subeach>

</foreach>
```

If 条件判断标签

支持的标签属性有 condition

```
<if conditon='$vo eq 1'>

输出 1

<elseif condition='$vo neq 2' />

输出 2

<else/>

输出 3

</if>
```

switch case default 标签

例如:

```
<switch name='vo.name'>

<case value='1'>输出 1</case>

<case value='2'>输出 2</case>

<default/>输出 3

</switch>
```

literal 标签

literal 标签用于原样输出任何代码段，而不会受变量替换和其它标签的硬性

例如，下面的代码中，包含在 literal 标签中的代码是会被解析的，保留原样输出。

```
<literal>

<switch name='vo.name'>
```

```
<case value='1'>输出 1</case>
```

```
<case value='2'>输出 2</case>
```

```
<default/>输出 3
```

```
</switch>
```

```
</literal>
```

如果担心你的代码被解析，就可以用该方法。

php 标签

php 标签用于执行 php 语句和代码段

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

5.7.4 Html 标签库

Html 标签库主要用于实现一些 Html 标记的动态生成和变量封装，主要有：

select 动态生成 select 列表

checkbox 动态生成 checkbox

radio 动态生成 radio

link 动态加载 js 或者 css 文件

imageLink 带有链接的图片

imageBtn 图片按钮

mulitSelect 多选组件

list DataGrid 组件

5.7.5 标签扩展

任何一个模板引擎的功能都不是为你量身定制的，具有一个良好的可扩展机制也是模板引擎的另外一个考量，Smarty 采用的是插件方法来实现扩展，ThinkTemplate 由于采用了标签库技术，比 Smarty 提供了更为强大的定制功能，和 Java 的 TagLibs 一样可以支持自定义标签库和标签，每个 XML 标签都有

独立的解析方法，所以可以根据标签库的定义规则来增加和修改标签解析规则。在 **ThinkTemplate** 中标签库的体现是采用 XML 命名空间的方式，例如：

```
<cx:volist id="list"></cx:volist>
```

每个命名空间都有一个对应的标签库 XML 定义文件，并且还包含有一个用于解析该标签库的类文件。系统默认对 **cx** 标签库进行支持，所以在定义 **cx** 标签库的标签时候，可以省略 XML 的命名空间前缀。当系统中存在很多的标签库的时候，每次编译都会加载所有的标签库解析文件，这样会造成一种浪费，因为很多情况，我们可能只是使用其中的一个或者二个标签库。所以，我们还必须在模板页面实现标签库引入功能，来告诉模板引擎当前模板页面需要哪些标签库的支持，从而加载需要的解析类。在 **ThinkTemplate** 中，使用 **tagLib** 标签来实现这一功能，例如：`<tagLib name="html,cx" />` 表示导入 **html** 和 **cx** 两个标签库的支持。如果没有定义，那么默认只是加载 **cx** 标签库。

利用标签库的特性，我们可以非常方便地扩展自己需要的标签，**ThinkTemplate** 正是采用这种机制来内置集成了一些常用的 **HTML** 组件标签，例如：

```
<html:list id="checkList" name="user" style="list" checkbox="true" action="true" datasource="list" show="id: 编号 |10%,title: 标题 :edit,cTime|toDate='Y-m-d h#i#s': 评论日期 ,status|getStatus: 状态 " actionlist="forbid|resume:禁用|恢复,edit:编辑" />
```

使用上面的自定义 XML 标签定义了一个 **DataGrid** 组件，省去了复杂的 **Html** 代码，在模板第一次执行的时候，模板引擎会把上面的组件标签解析成 **PHP** 和 **Html** 结合的代码，生成缓存文件。**ThinkTemplate** 中包含的 **Html** 标签库中封装了很多有价值的 **Html** 组件。

系统已经支持的标签库包括 **CX** 标签库和 **HTML** 标签库，你还可以根据自己需要扩展或者增加标签库。标签库由定义文件和解析类构成。每个标签库存在一个 XML 定义文件，用来定义标签库中的标签和属性。并且一个标签库文件对应一个标签库解析类，每个标签就是解析类中的一个方法。

例如，**CX** 标签库的定义文件是 **cx.xml** 位于 **ThinkTemplate/Template/Tags/** 目录下面，而 **cx** 标签库解析类文件是位于 **ThinkTemplate/Template/TagLib/** 目录下面的 **TagLibCx.class.php** 文件，每个标签的解析方法就是 **TagLibCx** 类的一个方法，为了不和系统的关键字冲突，所以在方法名前加上了 **_** 前缀，因此，假如要定义 **Cx:Var** 的标签解析，就需要定义一个 **_var** 方法。

标签库解析类的作用其实就是把某个标签定义解析成为有效的模版文件(可以包括 **PHP** 语句或者 **HTML** 标签)。扩展标签库需要添加标签库定义 XML 文件和标签库解析类

标签库定义的 XML 文件格式如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<taglib>

<shortname>cx</shortname>

<tag>

<name>include</name>

<bodycontent>empty</bodycontent>

<attribute>

<name>file</name>

<required>true</required>

</attribute>

</tag>

</taglib>
```

标签库的名称和文件名一致，每个 **tag** 标签对定义了标签库中的一个标签，例如，下面的代码定义了 **cx** 标签库的 **include** 标签：

```
<tag>

<name>include</name>

<bodycontent>empty</bodycontent>

<attribute>

<name>file</name>

<required>true</required>

</attribute>

</tag>
```

Bodycontent 属性表示该标签是否属于闭合标签，然后

```
<attribute>

<name>file</name>

<required>true</required>

</attribute>
```

定义了其中的一个 **file** 属性，**required** 表示该属性是否为必须定义属性。

然后,我们看解析类的定义,每个标签的解析方法在定义的时候需要添加“_”前缀,可以传入两个参数,属性字符串和内容字符串(对于非闭合标签)。下面是一个解析方法的定义:

```
function _include($attr)
{
    $tag    = $this->parseXmlAttr($attr,'include');
    $file   = $tag['file'];
    if(is_file($file)) {
        $parseStr = file_get_contents($file);
        return $this->tpl->parse($parseStr);
    } else {
        return $this->tpl->parseInclude($file);
    }
}
```

必须通过 `return` 返回标签的解析输出,在标签解析类中可以调用模板类的实例。

第6部分 附录

附录 A 发展历程

ThinkPHP 的诞生和发展历程

[2006 年 1 月 15 日] 6Path 0.6.0 版本发布, FCS 雏形版本

[2006 年 1 月 26 日] 6Path 更名为 FCS

全称为 Fast, Compatible & Simple OOP PHP Framework

[2006 年 2 月 12 日] (元宵节) 发布 FCS 0.6.1 版本, Google 讨论组成立

同期 FCS 第一版网站 labs.liu21st.com (已关闭) 开通

[2006 年 3 月 15 日] FCS 0.7.0 版本发布

[2006 年 3 月 23 日] FCS QQ 群成立

[2006 年 5 月 1 日] FCS 第二版网站 <http://fcs.org.cn> (已关闭) 开通

- [2006 年 5 月 7 日] FCS 0.8 版本发布
- [2006 年 9 月 28 日] FCS 官方 BLOG 开通
- [2006 年 10 月 25 日] FCS 0.9.0 版本发布
- [2006 年 12 月 25 日] SF 项目和 Google 网站 ThinkPHP 项目申请完成
- [2007 年 1 月 1 日] FCS 正式更名为 ThinkPHP
- [2007 年 1 月 8 日] ThinkPHP 0.9.5 版发布 同期 第三版官方网站 <http://ThinkPHP.cn> 开通
- [2007 年 2 月 21 日] TOPThink 社区暨新版 ThinkPHP 官方网站开通，并提供社区支持
- [2007 年 2 月 25 日] 发布 ThinkPHP 0.9.6 版本，完成 FCS 到 ThinkPHP 的正式迁移
- [2007 年 3 月 9 日] ThinkPHP 在线手册开通
- [2007 年 4 月 29 日] ThinkPHP 发布 0.9.7 版本
- [2007 年 5 月 24 日] ThinkPHP 和 PHPChina 合作推广
- [2007 年 6 月 4 日] ThinkPHP 和 PHP 开发资源网合作
- [2007 年 7 月 1 日] ThinkPHP 发布 0.9.8 版本
- [2007 年 10 月 15 日] ThinkPHP 发布 1.0.0RC1 版本，完成 PHP5 的重构
- [2007 年 12 月 15 日] ThinkPHP 发布 1.0.0 正式版本

附录 B 系统常量

THINK_PATH // ThinkPHP 系统目录

APP_PATH // 当前项目目录

APP_NAME // 当前项目名称

MODULE_NAME // 当前模块名称

ACTION_NAME // 当前操作名称

TMPL_PATH // 项目模版目录

LIB_PATH // 项目类库目录

CACHE_PATH // 项目模版缓存目录

CONFIG_PATH // 项目配置文件目录

LOG_PATH // 项目日志文件目录

LANG_PATH // 项目语言文件目录

TEMP_PATH //项目临时文件目录

PLUGIN_PATH // 项目插件文件目录

VENDOR_PATH // 第三方类库目录

DATA_PATH // 项目数据文件目录

IS_APACHE // 是否属于 Apache

IS_IIS //是否属于 IIS

IS_WIN //是否属于 Windows 环境

IS_LINUX //是否属于 Linux 环境

IS_FREEBSD //是否属于 FreeBSD 环境

NOW_TIME // 当前时间戳

MEMORY_LIMIT_ON // 是否有内存使用限制

OUTPUT_GZIP_ON // 是否开启输出压缩

MAGIC_QUOTES_GPC // MAGIC_QUOTES_GPC

THINK_VERSION //ThinkPHP 版本号

LANG_SET // 浏览器语言

TEMPLATE_NAME //当前模版名称

TEMPLATE_PATH //当前模版路径

__ROOT__ // 网站根目录地址

__APP__ // 当前项目（入口文件）地址

__URL__ // 当前模块地址

__ACTION__ // 当前操作地址

__SELF__ // 当前 URL 地址

TMPL_FILE_NAME //当前操作的默认模版名（含路径）

WEB_PUBLIC_URL //网站公共目录

APP_PUBLIC_URL //项目公共模版目录

预定义常量

WEB_LOG_ERROR=0 // 错误日志类型

WEB_LOG_DEBUG=1 // 调试日志类型

SQL_LOG_DEBUG=2 // SQL 日志类型
SYSTEM_LOG=0 // 系统方式记录日志
MAIL_LOG=1 // 邮件方式记录日志
TCP_LOG=2 // TCP 方式记录日志
FILE_LOG=3 // 文件方式记录日志
DATA_TYPE_OBJ=1 // 对象方式返回
DATA_TYPE_ARRAY=0 // 数组方式返回
URL_COMMON=0 // 普通模式 URL
URL_PATHINFO=1 // PATHINFO URL
URL_REWRITE=2 // REWRITE URL
HAS_ONE=1 // HAS_ONE 关联定义
BELONGS_TO=2 // BELONGS_TO 关联定义
HAS_MANY=3 // HAS_MANY 关联定义
MANY_TO_MANY=4 // MANY_TO_MANY 关联定义
EXISTS_TO_VALIDATE = 0 // 表单存在字段则验证
MUST_TO_VALIDATE = 1 // 必须验证
VALUE_TO_VALIDATE = 2 // 表单值不为空则验证

附录 C 配置参数

这里列出了系统内置的惯例配置中的配置参数，所有参数在没有生效之前都可以在项目配置文件或者模块配置文件中被覆盖，这里只是列出了默认的惯例设置，并不代表你的应用设置。

DISPATCH_ON=true

是否启用 Dispatcher

DISPATCH_NAME = 'Think'

默认的 Dispatcher 名称

URL_MODEL=1

URL 模式： 0 普通模式 1 PATHINFO 2 REWRITE

默认为 PATHINFO 模式，提供最好的用户体验和 SEO 支持

PATH_MODEL=2

// PATHINFO 模式

// 普通模式 1 参数没有顺序/m/module/a/action/id/1

// 智能模式 2 自动识别模块和操作/module/action/id/1/ 或者 /module,action,id,1/...

默认采用智能模式

PATH_DEPR='/'

PATHINFO 参数之间分割号

ROUTER_ON=true

启用路由判断

/ 日志设置 */*

WEB_LOG_RECORD=false

默认不记录日志

LOG_FILE_SIZE=2097152

日志文件大小限制

/ 插件设置 */*

THINK_PLUGIN_ON=false

默认不启用插件机制

/ 防刷新设置 */*

LIMIT_RESFLESH_ON=false

默认关闭防刷新机制

LIMIT_REFLESH_TIMES=3

页面防刷新时间 默认 3 秒

/ 错误设置 */*

DEBUG_MODE=false

调试模式默认关闭

ERROR_MESSAGE='您浏览的页面暂时发生了错误！请稍后再试～'

错误显示信息 非调试模式有效

ERROR_PAGE=""

错误定向页面

/* 系统变量设置 */

VAR_PATHINFO='s'

PATHINFO 兼容模式获取变量例如 ?s=/module/action/id/1 后面的参数取决于 PATH_MODEL 和 PATH_DEPR

VAR_MODULE='m'

默认模块获取变量

VAR_ACTION='a'

默认操作获取变量

VAR_ROUTER='r'

默认路由获取变量

VAR_FILE='f'

默认文件变量

VAR_PAGE='p'

默认分页跳转变量

VAR_LANGUAGE='l'

默认语言切换变量

VAR_TEMPLATE='t'

默认模板切换变量

VAR_AJAX_SUBMIT='ajax'

默认的 AJAX 提交变量

/* 模块和操作设置 */

DEFAULT_MODULE='Index'

默认模块名称

DEFAULT_ACTION='index'

默认操作名称

/ 模板设置 */*

TMPL_SWITCH_ON = true

默认启用多模板机制

TMPL_CACHE_ON=true

默认开启模板缓存

TMPL_CACHE_TIME=-1

模板缓存有效期 -1 永久 单位为秒

DEFAULT_TEMPLATE='default'

默认模板风格名称

TEMPLATE_SUFFIX='.html'

默认模板文件后缀

CACHFILE_SUFFIX='.php'

默认模板缓存后缀

TEMPLATE_CHARSET='utf-8'

模板模板编码

OUTPUT_CHARSET='utf-8'

默认输出编码

/ 模型设置 */*

CONTR_CLASS_PREFIX=""

控制器类名前缀

CONTR_CLASS_SUFFIX='Action'

控制器类名后缀，默认为 Action

ACTION_PREFIX=""

操作方法前缀

ACTION_SUFFIX=""

操作方法后缀

MODEL_CLASS_PREFIX=""

模型类前缀

MODEL_CLASS_SUFFIX='Model'

模型类后缀，默认为 Model

/ 静态缓存设置 */*

HTML_FILE_SUFFIX='.shtml'

默认静态文件后缀

HTML_CACHE_ON=false

默认关闭静态缓存

HTML_CACHE_TIME=60

静态缓存有效期

HTML_READ_TYPE=1

静态缓存读取方式 0 readfile 1 redirect

HTML_URL_SUFFIX='.shtml'

伪静态后缀设置

/ 语言时区设置 */*

LANG_SWITCH_ON = false

默认关闭多语言功能

DEFAULT_LANGUAGE='zh-cn'

默认语言

TIME_ZONE='PRC'

默认时区

/ 用户认证设置 */*

USER_AUTH_ON=false

默认不启用用户认证

USER_AUTH_TYPE=1

默认认证类型 1 登录认证 2 实时认证

USER_AUTH_KEY='authId'

用户认证 SESSION 标记

AUTH_PWD_ENCODER='md5'

用户认证密码加密方式

USER_AUTH_PROVIDER='DaoAuthenticitionProvider'

默认认证委托器

USER_AUTH_GATEWAY='/Public/login'

默认认证网关

NOT_AUTH_MODULE='Public'

默认无需认证模块

REQUIRE_AUTH_MODULE=""

默认需要认证模块

/* SESSION 设置 */

SESSION_NAME='ThinkID'

默认 Session_name 如果需要不同项目共享 SESSION 可以设置相同

SESSION_PATH=""

采用默认的 Session save path

SESSION_TYPE='File'

默认 Session 类型 支持 DB 和 File

SESSION_EXPIRE='300000'

默认 Session 有效期

SESSION_TABLE='think_session'

数据库 Session 方式表名

SESSION_CALLBACK=""

反序列化对象的回调方法

/* 数据库设置 */

DB_CHARSET='utf8'

数据库编码默认采用 utf8

DB_DEPLOY_TYPE=0

数据库部署方式 0 集中式（单一服务器） 1 分布式（主从服务器）

DB_CACHE_ON=false

默认关闭数据库缓存

DB_CACHE_TIME=60

数据库缓存有效期

DB_CACHE_MAX=5000

数据库缓存最多记录

SQL_DEBUG_LOG=false

记录 SQL 语句到日志文件

/* 数据缓存设置 */

DATA_CACHE_ON=false

默认关闭数据缓存

DATA_CACHE_TIME=-1

数据缓存有效期

DATA_CACHE_MAX=5000

数据缓存最多记录

DATA_CACHE_COMPRESS=false

数据缓存是否压缩缓存

DATA_CACHE_CHECK=false

数据缓存是否校验缓存

DATA_CACHE_TYPE='File'

数据缓存类型 支持 File Db Apc Memcache Shmop Sqlite Xcache Apachenote Eaccelerator

DATA_CACHE_TABLE='think_cache'

数据缓存表 当使用数据库缓存方式时有效

CACHE_SERIAL_HEADER="<?php\n//"

文件缓存开始标记，当缓存方式为 File 有效

CACHE_SERIAL_FOOTER="\n?". ">"

文件缓存结束标记，当缓存方式为 File 有效

SHARE_MEM_SIZE=1048576

共享内存分配大小，当缓存方式为 Shmop 有效

/ 运行时间设置 */*

SHOW_RUN_TIME=false

运行时间显示

SHOW_ADV_TIME=false

显示详细的运行时间

SHOW_DB_TIMES=false

显示数据库查询和写入次数

SHOW_CACHE_TIMES=false

显示缓存操作次数

SHOW_USE_MEM=false

显示内存开销

SHOW_PAGE_TRACE=false

显示页面 Trace 信息 由 Trace 文件定义和 Action 操作赋值

/ 模板引擎设置 */*

TMPL_ENGINE_TYPE='Think'

默认模板引擎 以下设置仅对使用 Think 模板引擎有效

TMPL_DENY_FUNC_LIST='echo,exit'

模板引擎禁用函数

TMPL_L_DELIM='{'

模板引擎普通标签开始标记

TMPL_R_DELIM='}'

模板引擎普通标签结束标记

TAGLIB_BEGIN='<'

标签库标签开始标记

TAGLIB_END='>'

标签库标签结束标记

/ Cookie 设置 */*

COOKIE_EXPIRE=3600

Cookie 有效期

COOKIE_DOMAIN=""

Cookie 有效域名

COOKIE_PATH='/'

Cookie 路径

COOKIE_PREFIX='THINK_'

Cookie 前缀 避免冲突

/ 分页设置 */*

PAGE_NUMBERS=5

分页显示页数

LIST_NUMBERS=20

分页每页显示记录数

/ 数据格式设置 */*

AJAX_RETURN_TYPE='JSON'

AJAX 数据返回格式 JSON XML ...

DATA_RESULT_TYPE=0

默认数据返回格式 1 对象 0 数组

`/* 其它设置 */`

AUTO_LOAD_PATH='Think.Util.'

`__autoload` 的路径设置 当前项目的 Model 和 Action 类会自动加载，无需设置 注意搜索顺序

CALLBACK_LOAD_PATH=""

反序列化对象时自动加载的路径设置

UPLOAD_FILE_RULE='uniqid'

文件上传命名规则，例如 `time`、`uniqid`、`com_create_guid` 等，支持自定义函数，仅适用于内置的 `UploadFile` 类

LIKE_MATCH_FIELDS='title|content|value|remark|company|address'

数据库查询的时候需要进行模糊匹配的字段，当对包含这些字段的数据表进行查询的时候，会自动使用 `LIKE '%value%'` 方式的模糊查询

附录 D 文件列表

`Common/convention.php` 惯例配置文件

`Common/debug.php` 默认调试配置文件

`Common/defines.php` 系统定义文件

`Common/functions.php` 系统函数文件

`Lang/zh-cn.php` 系统简体中文语言包文件

`Tpl/PageTrace.tpl.php` 页面 Trace 模板文件

`Tpl/ThinkException.tpl.php` 异常页面模板文件

`Think.Core.Base` 系统基类

`Think.Core.App` 系统应用程序类

`Think.Core.Dispatcher` 系统 Dispatcher 类

`Think.Core.View` 系统视图类

`Think.Core.Action` Action 控制器基础类

`Think.Core.Model` 系统模型基础类

`Think.Db.Db` 系统数据库公共类

`Think.Db.ResultIterator` 数据集 Iterator 类

`Think.Db.Driver.*` 数据库驱动类库

Think.Exception.ThinkException 系统异常基类

Think.Template.ThinkTemplate ThinkTemplate 模型引擎类

Think.Template.TagLib 标签库基类

Think.Template.TagLib.* 标签库解析类

Think.Template.Tags.* 标签库 XML 定义文件

Think.Util.Cache 缓存类

Think.Util.Cache.* 缓存驱动类

Think.Util.Session Session 管理类

Think.Util.Cookie Cookie 管理类

Think.Util.Debug 调试类

Think.Util.Filter 过滤器类

Think.Util.Filter.FilterDbSession 数据库 Session 方式过滤器类

Think.Util.Log 日志处理类

Think.Util.HtmlCache 静态缓存处理类

ORG.Util.Page 分页处理类

ORG.Util.Image 图像处理类

ORG.Util.Stack 堆栈实现类

ORG.Util.ArrayList ArrayList 实现类

ORG.Util.HashMap HashMap 实现类

ORG.Date.Date 日期处理类

ORG.Net.Http Http 处理类

ORG.Net.UploadFile 文件上传类

ORG.Net.IpLocation IP 查询类

ORG.Text.Validation 验证类

ORG.RBAC.RBAC RBAC 组件类

ORG.RBAC.ProviderManager 认证委托管理器类

ORG.RBAC.Provider.* 认证委托方式类

ORG.RBAC.AccessDecisionManager 访问决策管理器类

第7部分 FAQ