Skihat 开发手册

Skihat Team

1.0 Beta

2014/7/17

目录

第一部分：了解 Skihat MVC 框架

第 1 章：Skihat 简介

1.1、Skihat 是什么

Skihat 是一个基于 PHP 5.3 + 的高效 MVC 框架，提供安全、日志、事务、缓存、数据库访问、主题等诸多特性。遵循自然、简洁的设计理念、强调实用性、鼓励模块化开发、遵循习惯优于配置的编程理念。

Skihat 遵循 Apache 2 开源协议，任何个人或公司都允许在尊重原作者版权的基础上开发、修改和发布，由 Skihat 开发的企业或个人应用，同时鼓励开发者为 Skihat 提供高质量的代码和各种扩展组件。

1.2、Skihat 的功能和特性

作为一个 PHP MVC 应用框架，Skihat 希望能满足应用的日常开发需求，为此提供相应的功能和特性。

- **微核模式设计**

Skihat 类是框架的核心类，提供动态配置、调试、优先文件服务、ioc 和 i18n 五个基础服务，其它服务都是基于这五个核心服务进行构建。

- **分层设计**

自上而下将整个应用分为“应用层”、“应用服务层”和“内核服务层”。为了提高应用的性能和便于扩展，Skihat 采用白盒模型作为分层的实现方式。

- **Web MVC**

采用常见的单入口 MVC 模式设计，所有请求都由 app/publics/index.php 文件统一处理。

- **模块化开发**

允许将不同的业务划分到不同的模块中，再由模块提供具体的实现，框架从路由、分发、控制器、视图和模型等多个方面提供支持。

- **组件化的开发与设计**

采用组件化的开发与设计，将不同的功能封装到不同的组件中，由组件提供对外一致性服务。

- **ActiveRecord**

采用标准 ActiveRecord 模式，将数据表封装成模型类，静态方法提供多条记录的操作，实例方法提供对单条记录的操作（目前仅支持 MySql 数据库）。

- **视图/主题**

采用视图模板的方式提供用户的响应内容，允许设置不同的模板主题，支持布局模板、子视图、助手和表单助手等。

- **路由器支持**

支持普通路由、重写、phpinfo 和 restful 四种路由样式，并提供相应的支持。允许根据模块、包和数据的不同分别设置不同的子域名。

- **安全性设计**

Skihat 的所有源代码都是由.inc 或.stp 作为文件扩展名，同时源代码默认部署在 Web 根目录之外，禁止外部直接访问，数据访问采用 PDO 占位符，防止 SQL 注入。

- **数据读写分离**

支持数据库访问的读写分离，允许读写分别由不同的服务器提供。

- **事务处理**

允许将事务由统一的组件提供支持，内部自动完成提交或回滚。

- **缓存支持**

允许在同一应用中使用多种缓存，允许在控制器、视图和子视图的自动缓存支持。

- **日志记录**

  允许在同一应用中使用多种日志记录器，并由统一的接口提供支持。

1.3、Skihat 的环境需求

作为 PHP MVC 框架，支持任何满足 PHP 环境的 Windows 或 Linux 操作系统，可以运行于 IIS、Nginx、Apache 在内的多种 Web 服务器，只需要满足以下特定要求：

- PHP 5.3 +、php_mbstring、json、php_pdo、php_pdo_mysql；
- MySql 5.0 +

1.4、获取 Skihat 最新版

为了保证 Skihat 的安全性和完整性，建议通过以下渠道获取源代码：

- 从官方网站 http://skihat.cn 下载最新稳定版；
- 从 https://github.com/skihat-team/framework 下载最新开发版；

1.5、许可协议

Skihat 遵循 Apache 2 开源协议发布 http://www.apache.org/licenses/LICENSE-2.0。

第 2 章：Skihat 结构设计

2.1、Skihat 命名规范

- **目录**

遵循 camel 小写规范的复数形式，即单词小写、单词间使用下划线作为分隔符，同时采用复数形式。例如：controllers、models 等。

- **文件名**

遵循 camel 小写规范，所有 PHP 代码文件都采用.inc 扩展名，模板文件采用.stp 作为扩展名，全站只有 index.php 一个 php 扩展文件。例如：core.inc、routers.inc。

采用.inc 作为扩展名主要是出于安全考虑，防止客户端对框架源代码文件的直接访问。

- **全局函数**

全局函数同原始 PHP 函数保持一致，采用 camel 小写规范。例如：array_key_pop、array_key_value。

- **类名/接口**

类和接口都采用 Pascal 命名规范，即单词首字母大写，例如：Router、Model。

接口与类相比单词前还需要加 I 字母（从.NET 引入的不良习惯）。例如：IRouterRule、IRouterStyle。

注意：Skihat 并不要求类名与文件名完全一致，但文件名一定要具有引导作用。例如：router_styles.inc 中包含多个路由样式类的声明。

- **类/接口成员**

Skihat 中类和接口的成员都采用 camel 大写规范，即除第一个单词首字母小写外，其余单词首字母大写，例如：User::fetchAll，$user->save。此外，非公有成员前面建议加下划线。例如：

  - protected $_conf;

- protected function _invokeRender();

■ **请求参数名称**

Skihat 建议请求参数名称采用 camel 小写命名规范，即单词全部小写，单词之间使用”_”分隔。例如：http://www.example.com/?controller=users&action=edit。

■ **数据表**

数据表默认采用模型名称的 camel 小写规范。例如：SkxUser => skx_user。

■ **数据表字段**

数据表字段没有特殊规定，建议采用 camel 小写规范。

■ **常量**

在 Skihat 中常量分为全局常量和类常量两种：

- 全局常量：全局常量使用"SKIHAT_"作为前缀，单词全部大写，单词间使用下划线分隔。例如：SKIHAT_PATH_LIBRARY、SKIHAT_PARAM_CONTROLLER。
- 类常量：类常量同样采用单词全大写，同全局常量相比不需要"SKIHAT_"前缀。例如：Database::SYTAX_TALBE、Database::SYTAX_LINKS。

注意：所有的全局常量都在文件 app/boots/starup.inc 中声明。

2.2、Skihat 目录结构

解压 skihat_lasted.zip 后，包含 app、data、docs、skihat、testes 和 vendor 六个目录，下面对各个目录进行详细的说明。

app 目录

app 目录是应用程序的开发目录，存放应用服务层的代码和实现，目录常量 SKIHAT_PATH_APP，允许包含以下子目录：

- app/appservs：应用服务目录，保存与应用相关的类的子类实现；
- app/boots：启动目录，包含应用启动所需的启动文件和配置信息；

- app/controllers：控制器目录，存储应用控制器信息；
- app/langs：语言目录，支持 i18n 信息；
- app/models：模型目录，存储模型代码；
- app/modules：模块目录，存储应用模块代码，常量 SKIHAT_PATH_APP_MODULES；
- app/publics：公共访问目录，Web 服务器绑定根目录，包含 index.php 应用入口文件，目录常量 SKIHAT_PATH_APP_PUBLICS；
- app/services：服务目录，封装内部的服务接口；
- app/themes：主题目录，存储主题模板；
- app/views：视图目录，存储应用视图代码；

data 目录

data 目录是应用存储数据的临时目录，需要创建和修改文件的权限，目录常量 SKIHAT_PATH_DATA，包含以下子目录：

- data/caches：缓存文件临时存储目录，常量 SKIHAT_PATH_DATA_CACHES；
- data/downloads：下载临时存储目录，常量 SKIHAT_PATH_DATA_DOWNLOADS；
- data/loggers：日志目录，保存网站运行的日志文件，目录常量 SKIHAT_PATH_DATA_LOGGERS；
- data/runtime：默认存放 Skihat 运行时缓存文件；

docs 目录

用于存储应用程序中产生的开发文档，没有预设子目录。

skihat 目录

Skihat 目录用于存储 Skihat 库文件代码，目录常量 SKIHAT_PATH_LIBRARY，包含以下子目录：

- skihat/applications：前端控制器组件目录，提供应用分发所需的服务；
- skihat/appservs：应用服务目录，提供应用开发所需的类的声明；
- skihat/boots：应用启动目录，保存应用启动的默认实现文件；
- skihat/controllers：控制器组件目录，提供控制器的相关类声明；
- skihat/models：模型组件目录，提供模型类的相关类声明；

- skihat/routers：路由器组件目录，提供路由相关类声明；
- skihat/views：视图组件目录，提供视图接口和主题类相关声明；

除此以外，还有一个非常特殊的 kernels 目录，提供内核服务层的相关代码和组件：

- kernels/caches：缓存组件目录，提供应用缓存支持；
- kernels/databases：数据库组件目录，提供应用数据库访问支持（目前仅支持 MySql）；
- kernels/loggers：日志组件目录，提供日志记录服务；
- kernels/securities：安全组件目录，提供应用安全管理支持；
- kernels/transactions：事务组件目录，提供一致性事务处理；
- kernels/medias：资源组件目录，提供文件上传服务；

testes 目录

测试代码目录，在 testes/units 中包含 Skihat 的全部单元测试代码，testes/example 中包含本书中的大部分示例代码。

vendor 目录

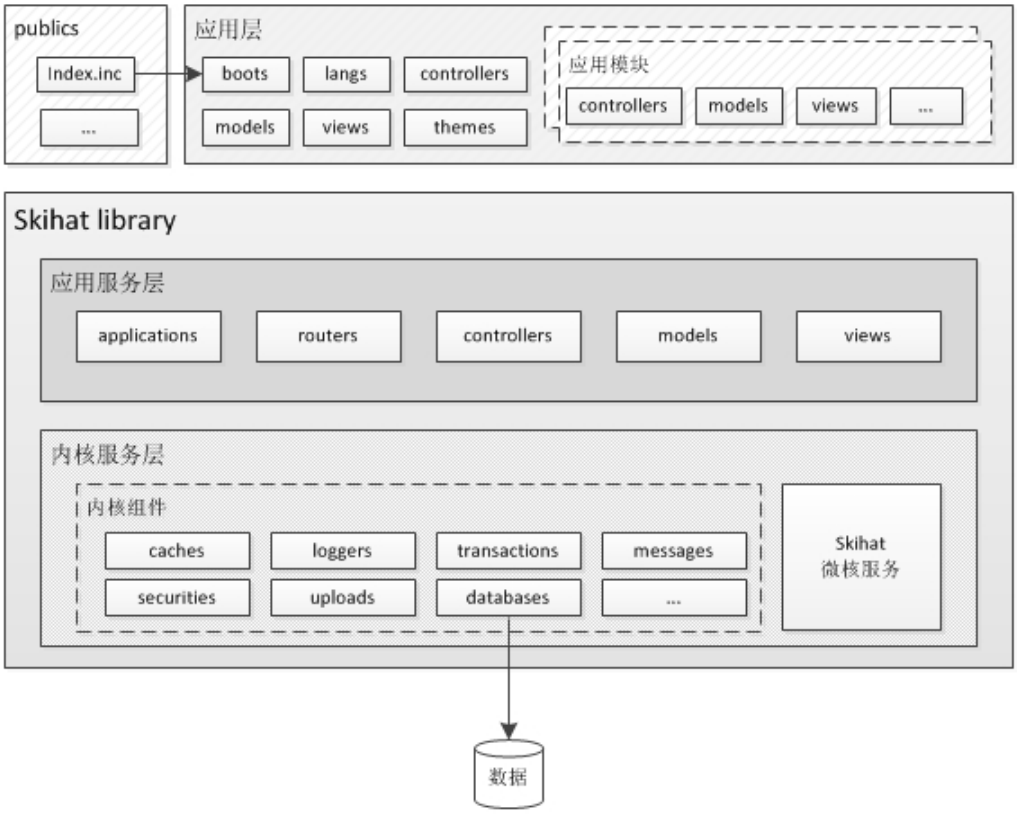
外部依赖库目录，在 skihat 核心中不包含任何的外部依赖，因此该目录为空，开发中可以根据需要引用外部库文件，目录常量 SKIHAT_PATH_VENDOR。

2.3、Skihat 架构模式

这一部分讲解 Skihat 中采用的一些设计模式和设计方法，对于不了解设计模式的开发者可能会理解比较困难。

如果不能很好的理解这一部分的内容也没有什么关系，这一部分的内容只需要了解。

2.3.1、Skihat 架构图



图：2-1 Skihat 架构图

Skihat 整体采用分层架构模式（白盒）和组件工厂模式，对外部的依赖由内核服务层组件提供封装，并且提供一致性访问。

2.3.2、Skihat 分层模式

Skihat 采用分层模式作为核心的架构模式，整个框架自上而下分为：

- 应用层：根据具体的业务逻辑进行开发和实现，通常开发者只需要关注这一部分的代码，目录 app 存储应用层代码。
- 应用服务层：提供应用层开发所需的服务，主要由一组 MVC 组件集构成，目录 skihat 存储应用服务层代码。
- 内核服务层：内核服务层的主要作用是解耦外部依赖，将外部依赖封装为统一接口，提供对内的一致性服务，目录 skihat/kernels 存储内核服务层代码。

为了提高性能和扩展性，Skihat 分层架构采用白盒模式，即允许"应用层"直接调用"内核服务层"的组件。

2.3.3、组件与工厂模式

Skihat 完全采用组件的方式封装各种服务，将不同的服务封装在不同的组件，例如：缓存、日志、安全等。

为了保持一致性，大部分的组件都是采用以下方式实现：

- 每一个组件都有一个接口对象，外部通过接口对象获取一致性服务；
- 在组件内部使用“工厂模式”＋“提供者引擎”，提供服务的不同解决方案；

例如：在缓存组件中 Cache 类作为外部访问的统一接口，提供一致性访问；内部则使用 ICacheEngine 接口提供不同的缓存解决方案。

2.3.4、MVC 单入口模式

MVC 模式是实现 Web 处理的一种常见模式，将应用分为 Model、View 和 Controller 三个部分。在结合 Web 开发特点和其它 MVC 框架的基础上，Skihat 应用服务层 MVC 由以下五个组件构成：

- 前端控制器：接收应用请求，将请求交由路由组件处理，并根据从路由组件中生成的请求参数，将请求分发到不同的模块和控制器，同时还提供错误的全局处理；
- 路由器：在 Web 开发中，路由器是 MVC 中不可或缺的一个组件，能够根据路由配置信息，将外部请求转换为内部应用所需的参数；
- 控制器：控制器提供外部请求的响应服务，调用模型组件完成业务处理，并将处理的结果交由视图返回给客户端；
- 模型：使用 ActiveRecord 模式封装的领域对象，根据业务需要执行相关的处理；
- 视图：使用模板将处理结果返回给客户端；

整个 Skihat 的 MVC 采用单入口模式响应所有的请求，即 app/publics/index.php 作为响应的 php 文件。

2.3.5、微核模式

微核模式是指将核心服务封装在一个单独的类、组件或服务中，再通过核心服务扩展出新的服务，很多成功的产品都是采用该模式进行设计。

在 Skihat 中核心服务被封装在 Skihat（skihat/kernels/skihat.inc）类中，提供以下五个服务：

- 动态配置服务：为应用提供新的配置信息；
- 动态调试服务：为应用提供整体调试服务；
- 优先文件服务：为应用提供文件加载服务；
- Ioc 依赖倒置：使用配置信息创建实例和单例对象的管理；
- i18n 国际化：提供国际化支持；

其它服务都是依赖于 Skihat 类所提供的服务扩展出新的服务。

2.3.6、模块化开发

在 Skihat 中，建议采用模块的方式来实现相关的业务和需求，为了支持模块化开发，在 Skihat 内部通过以下三个方面提供支持：

- 模块分发：允许指定访问的模块，并根据模块参数进行分发操作；
- 路由器：允许根据模块、包划分不同的子域名，允许为不同的子域名采用不同的路由样式；
- 包：允许在模块内部提供不同功能的包，面向不同的需求；

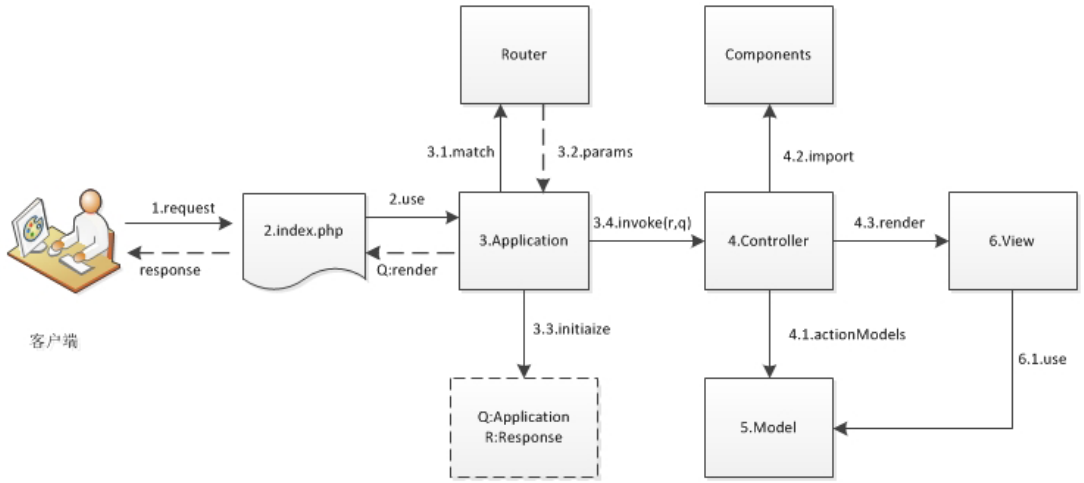
使用模块化开发的最大优势在于复用性和扩展性。

2.3.7、代码隔离原则

绑定 Skihat 网站，建议将 app/publics 绑定为 Web 根目录，这样做的好处在于通过禁止 publics 目录以外的文件访问和限制，提供应用程序的安全性，目前很多 PHP 框架都采用这一机制来提高安全性。

2.4、Skihat 的 MVC 访问流程

2.4.1、Skihat MVC 流程图



图：2-2 Skihat MVC 流程图

Skihat 的 MVC 在整体上采用单入口模式＋前端控制器、分发器作为主的结构。

2.4.2、index.php 文件

index.php 文件提供单一入口，所有的请求都由 index.php 文件进行处理，在 index.php 内部调用框架的初始化代码，引用 Application 前端控制器类，再将流程的控制权交由 Application 类来进行处理。

2.4.3、Application 类

Application 类是 Skihat 的前端控制器类，主要有四个方面的职责：

- 调用 Router 组件生成完整请求参数；
- 初始化 ApplicationRequest 和 ApplicationResponse 类；
- 根据请求参数将请求分发到 Controller 实例的 invoke 方法中；
- 将控制器的处理结果所回给请求客户端；

除此以外，Application 类还负责全局的错误和异常处理，因此如果需要修改默认错误或异常处理，只需要重写 Application 的对应方法。

2.4.4、Controller 类

Controller 类，是具体处理客户响应的类，在 Controller 中使用模型和各种组件（主要是内核服务组件）来完成业务处理，并将处理的结果由 View 类处理。

2.4.5、View 类

在 View 类中，调用模板信息生成各种响应格式并填充 ApplicationResponse 类，再由 ApplicationResponse 类输出响应的结果。

2.4.6、Skihat 流程总结

总体上来讲 Skihat 的 MVC 实现与大多数的 MVC 框架所采用的流程基本相同，只是在部分设计方法上有所区别，对于有过 MVC 开发经验的开发者还是比较容量理解。

2.5、Skihat 的完整调用流程

2.5.1、Skihat 完整调用流程图

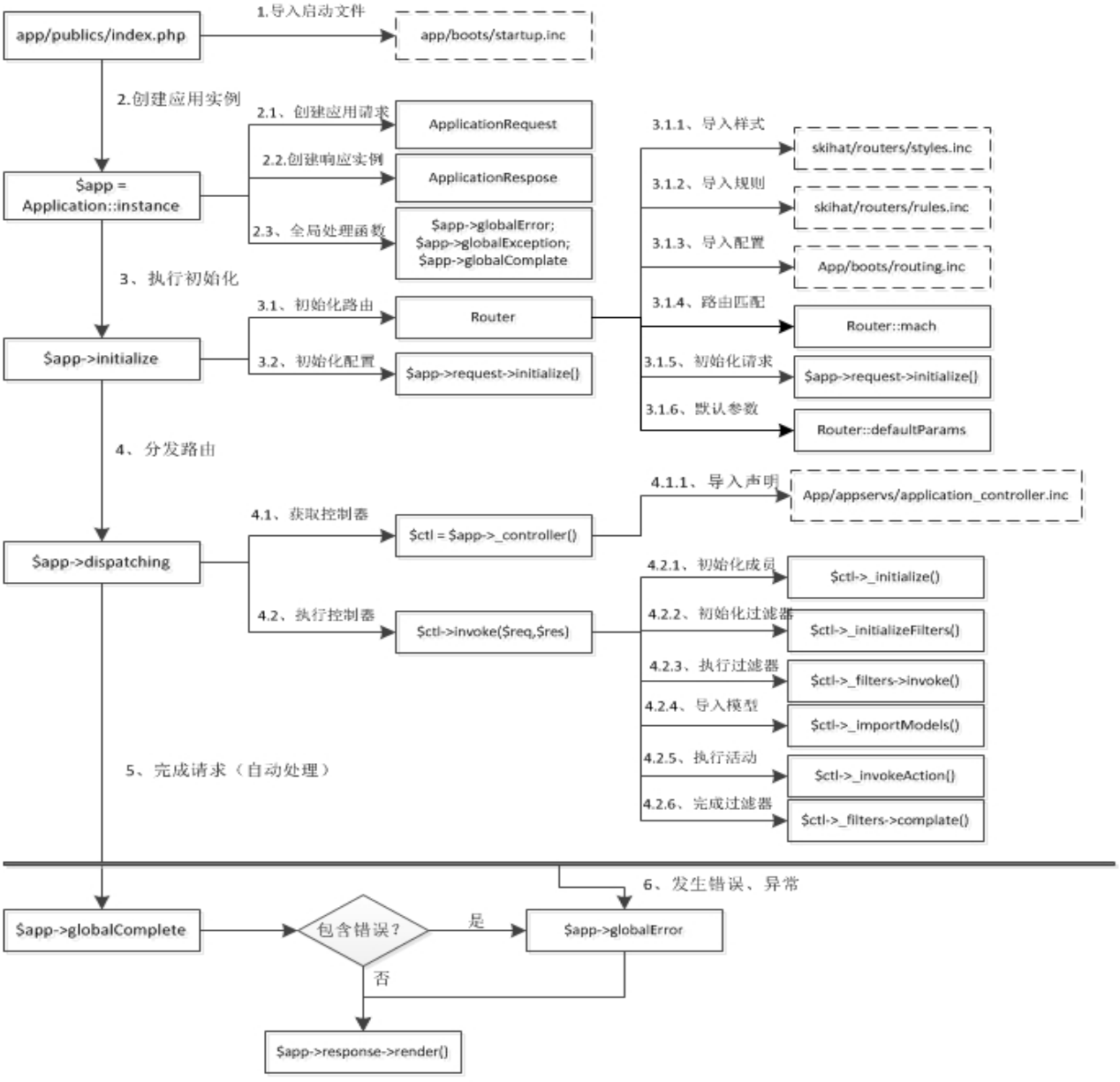


图 20-3：Skihat 完整流程图 (大图 http://skihat.cn/docs/stack.jpg)

其中的部分内容，在 MVC 流程一节中已有讲解，完整流程图比 MVC 流程图更为详细。

2.5.2、boots/startup.inc

访问 index.php 后，首先导入的文件就是 boots/startup.inc，在 startup.inc 中主要处理以下工作：

- 声明全局常量，可以根据实际情况进行调整；
- 引用内核文件：skihat/kernels/core.inc 和 skihat/kernels/skihat.inc；
- 配置运行环境 Session、Cookie、日志、错误提示等；

startup.inc 是一个非常重要的文件，决定了程序运行的基础，开发中可以根据面要重新调整相关的设置，以满足环境或部署的需要。

2.5.3、boots/config.inc

在$app->initialize 中会自动调用 boots/config.inc 文件，在文件中包含全局运行的动态配置信息，包含类似以下信息：

- Skihat::write(‘kernels/databases’,array(‘xxxxx’));
- Skihat::write(‘kernels/loggers’,array(‘xxx’));

为了满足部署需求，我们也可以根据实际需要增加或修改配置信息，详情请查看各个组件的配置节点。

2.5.4、boots/routing.inc

一起进行初始化的还有 routing.inc 文件，在文件中包含路由配置信息，包含以下类似代码：

- Router::domain(‘xxxxx’)->style(‘xxxx’)->rule(‘xxx’);

这些信息同样可以根据需要进行更新和调整，详情请查看路由组件。

2.5.5、内核服务组件

在整个流程图中没有包含对内核组件的引用，这是因为这些组件都是在 Controller 内部根据需要进行加载和使用。

2.6、SKIHAT 常量声明

为了便于统一管理和维护，在 Skihat 中所有全局常量都在 app/boots/startup.inc 文件中声明，所有常量都是以 SKIHAT_作为前缀。

2.6.1、调试常量

常用 SKIHAT_DEBUG 声明当前应用的调试级别，允许使用以下值：

- 0：运行环境，不开启任何调试服务（产品环境）；
- 1：缓存模式，开启缓存服务，启动调试服务；
- 2：完全调试模式，不开启缓存功能，启动调试服务；

2.6.2、文件常量

文件常量用于文件查找和获取的常量声明：

- SKIHAT_DEFAULT_EXT：默认文件扩展名，默认值为".inc"。
- SKIHAT_CLASS_SEPARATOR：默认类分隔符，默认值为"#"。
- SKIHAT_PACKAGE_SEPARATOR：默认目录（包）分隔符，默认值为"."。

2.6.3、目录常量（SKIHAT_PATH_XXX）

目录常量用于声明常用的目录路径，通常这些目录都包含在根目录下，可以根据实际需要调整相应的值，适应不同的部署需求。

根目录（SKIHAT_PATH_ROOT）

SKIHAT_PATH_ROOT：根目录，默认情况下，其它路径基于根路径。

应用目录（SKIHAT_PATH_APP）

- SKIHAT_PATH_APP：应用目录，默认值为 app 目录；
- SKIHAT_PATH_APP_MODULE：应用模块目录，默认值为 app/modules 目录；

- SKIHAT_PATH_APP_PUBLICS：应用公共目录，默认值为 app/publics 目录；
  SKIHAT_PATH_APP_PUBLICS_UPLOADS：应用上传目录，默认为 app/publics/uploads 目录；

数据目录（SKIHAT_PATH_DATA）

- SKIHAT_PATH_DATA：运行数据目录，存放临时文件，默认为 data；
- SKIHAT_PATH_DATA_CACHES：缓存目录，存放文件缓存信息，默认为 data/caches；
- SKIHAT_PATH_DATA_DOWNLOADS：下载目录存放下载信息，默认为 data/downloads；
- SKIHAT_PATH_LOGGERS：日志目录存放日志信息，默认为 data/loggers 目录；

库目录（SKIHAT_PATH_LIBRARY）

- SKIHAT_PATH_LIBRARY：Skihat 库文件存放路径，默认值为 skihat 目录。

引用目录（SKIHAT_PATH_LIBRARY）

- SKIHAT_PATH_VENDOR：外部引用库文件路径，默认值为 vendor 目录。

注意：Skihat 所有的目录都不包含最后一个"/"信息。

2.6.4、参数常量（SKIHAT_PARAM_XXXX）

在 Skihat 中，参数包含分发的具体模块、包、控制器和活动的信息，使用自定义参数时，避免与框架的参数名称相同。

请求参数常量（SKIHAT_PARAM_XXXX）

- SKIHAT_PARAM_MODULE：模块参数，决定请求的模块，默认名称为 module。
- SKIHAT_PARAM_PACKAGE：包参数，决定请求的包，默认值名称为 package。
- SKIHAT_PARAM_CONTROLLER：控制器参数，默认名称为 controller。
- SKIHAT_PARAM_ACTION：活动名称参数，默认名称为 action。
- SKIHAT_PARAM_ID：编号参数，默认名称为 id。
- SKIHAT_PARMA_FORMAT：请求格式参数，默认名称为 fmt。

除此以外，还包含三个特殊的参数名称：

- SKIHAT_PARAM_REWRITE：重写参数，从 Web 服务器的重写值，默认名称为 url。
- SKIHAT_PARAM_ANCHOR：锚参数，决定 URL 的锚名称，默认为"#"。
- SKIHAT_PARAM_METHOD：方法参数，指定请求所使用的参数，默认为__method__。

注意：在实际过程中建议不要修改这些默认名称值，防止代码 URL 发生错误。

请求参数默认值（SKIHAT_PARAM_DEFAULT_XXX）

参数默认值，是指定参数不存在时的默认情况，Skihat 的参数默认值有：

- SKIHAT_PARAM_DEFAULT_MODULE：默认模块，默认值为 false。
- SKIHAT_PARAM_DEFAULT_PACKAGE：默认包，默认为 false。
- SKIHAT_PARAM_DEFAULT_CONTROLLER：默认控制器，默认值为 index。
- SKIHAT_PARAM_DEFAULT_ACTION：默认活动，默认值为 index。
- SKIHAT_PARAM_DEFAULT_FORMAT：默认格式，默认值为 html。

分页参数常量（SKIHAT_PARAM_PAGINATE）

分页参数，提供分页的参数名称，允许在 startup.inc 中重新设置，包含以下参数：

- SKIHAT_PARAM_PAGINATE：当前页参数，默认名称为 page。
- SKIHAT_PARAM_PAGINATE_SIZE：分页大小参数，默认名称为 page_size。
- SKIHAT_PARAM_PAGINATE_SORT：分页排序参数，默认名称为 page_sort。
- SKIHAT_PARAM_PAGINATE_DIR：分页排序方式名称，默认名称为 page_dir。

2.6.4、IOC 配置常量（SKIHAT_IOC_XXXX）

IOC 配置常量决定 Skihat::ioc 创建时所需的关键字，包含以下四个常量：

- SKIHAT_IOC_CLASS：类常量，指创建的类信息，默认值为__class。
- SKIHAT_IOC_PATHS：目录常量，指定类所在的文件目录信息，默认值为__paths。
- SKIHAT_IOC_EXT：文件扩展名，指定创建类文件的扩展名信息，默认值为__ext。
- SKIHAT_IOC_INITIAL：IOC 初始化常量，指定初始化方法信息，

更多信息请查看 Skihat::ioc 方法。

2.6.5、国际化常量（SKIHAT_I18N_XXX）

默认语言常量（SKIHAT_I18N_LANG）

- SKIHAT_I18N_LANG：默认语言常量，默认值为 zh-CN；

格式化常量（SKIHAT_I18N_XXX）

- SKIHAT_I18N_DATE_TIME：日期时间格式化字符串常量，默认值为 Y-m-d H:i:s。
- SKIHAT_I18N_DATE：日期格式化常量，默认值为 Y-m-d。
- SKIHAT_I18N_TIME：时间格式化常量，默认值为 H:i:s。
- SKIHAT_I18N_WEEK：周格式化常量，默认值为 l。
- SKIHAT_I18N_CURRENCY：货币格式化常量，默认为￥.2f。

2.6.6、消息通知常量 SKIHAT_MESSAGE_XXX

通知消息是进行客户交互的重要方式，在 Skihat 中允许指定以下通知消息：

- SKIHAT_MESSAGE_INFO：消息通知，默认值为 info。
- SKIHAT_MESSAGE_ERROR：错误通知，默认值为 error。
- SKIHAT_MESSAGE_WARNING：警告通知，默认值为 warning。

除此外还有两个消息值表示执行的结果：

- SKIHAT_MESSAGE_SUCCESS：成功标志，默认值为 success。
- SKIHAT_MESSAGE_FAIL：失败标志，默认值为 fail。

2.6.7、其它常量

- SKIHAT_CURRENT_DOMAIN：域名常量，默认为$_SERVER[‘HTTP_HOST’];

第 3 章：Skihat 应用开发

讲解框架最简单的办法，就是使用框架开发一个简单的应用程序，通过示例来了解框架的开发流程。

- 示例文件位置：testes/examples

3.1、准备工作

3.1.1、需求列表

开发一个简单的留言板，满足以下需求：

- 在前台展示最新 5 条留言记录并允许提交新留言；
- 后台允许对查看留言信息，并且删除已发布的留言;
- 分别使用不同的域名，对前后台进行访问；

3.1.2、数据库设计

为了示例的简洁性，仅仅包含一个数据表 guestbook，创建代码如下：

 # 文件：docs/schema.sql

```
CREATE DATABASE guestbooks character set 'UTF8';
USE guestbooks;
CREATE TABLE guest_book (
    `id`          INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `user`        VARCHAR(32) NOT NULL,
    `email`       VARCHAR(64) NOT NULL,
    `created`     INT NOT NULL,
    `content`     VARCHAR(256) NOT NULL
);

INSERT INTO guest_book(`user`,`email`,`created`,`content`)
VALUES('skihat','skihat-team@outlook.com’,unix_timestamp(),'skihat framework'),
        ('php framework’,’ skihat-example@outlook.com’,unix_timestamp(),'example framework');
```

3.1.3、建立测试环境

建立测试环境需要经过以下几个步骤：

1. 从 http://skihat.cn 中下载 skihat 的最新稳定版压缩包 skihat-lasted.zip；
2. 将解压后的内容上传到 web 服务器上，如果是本地服务器则复制到相应的目录；
3. 在 Web 服务器上配置新的站点，注意将站点的根目录绑定到 app/publics 目录下；
4. 如果 Web 服务器绑到有具体的域名信息，则将为本地 hosts 绑定站点域名；
5. 通过浏览器访问，如果显示 skihat 的欢迎信息，则表示环境建立成功。

注意：不同 Web 服务器的站点创建方式各不相同，网上有大量相关的资料，这里就不一一进行说明；

使用 hosts 文件进行跳转是为了不进行 DNS 设置，在开发 Web 应用时经常都需要进行设置，对应的文件地址是：

- Windows： C:\Windows\System32\drivers\etc\hosts；
- linux/mac：/etc/hosts

文件的格式为：

127.0.0.1　　www.example.com

服务器 IP 地址　　网站域名

设置完成后，可以使用 ping 命令查看设置的结果，如果有响应就表示设置成功。例如：ping www.example.com 如果返回 hosts 设置的结果表示设置成功。

3.2、创建模型类

同一些框架不需要声明模型不同，在 Skihat 中要求所有的模型必须声明，创建文件 app/models/guest_book.inc，并录入以下 PHP 代码：

```
class GuestBook extends ApplicationModel {
}
```

模型类必须继承自 ApplicationModel 类，同时数据表的名称默认为类名的 camel 小写命名规范（可以使用__config 方法重写）。

注意： Skihat 建议将每个模型类存放独立的文件中，文件名采用类的 camel 小写命名规范， 文件存放在 app/models 目录下（模块的模型存放的模块的 models 目录下）。

3.3、编写控制器

控制器是响应客户端请求的最小组织单位，在一个控制器中允许包含多个 action（活动），每个 action 对应客户一种类型的请求。例如：默认请求 Skihat 时的响应代码就是由 index_controller.inc 文件下的 IndexController->indexAction 方法提供响应。

创建文件 app/controllers/guestbooks_controller.inc 并修改代码如下：

```
class IndexController extends ApplicationController {
    public function indexAction() {
            # 将数据的响应结果设置为视图变量
            $this['guestbooks'] = GuestBook::fetchAll();
    }

    # 重载父类方法，加载当前模型所需要模型
    public function actionModels() {
        # 使用 import 方法加载模型实例。
        Skihat::import('models.guest_book', SKIHAT_PATH_APP);
    }
}
```

为了 Skihat 框架能够正确解析到控制器，控制器必须满足以下规范：

- Skihat 控制器文件中只允许包含一个控制器；
- 控制器文件名必须以_controller 为后缀；
- 类名必须是文件名的 Pascal 命名规范；
- 活动方法的名称必须以 Action 为后缀，以区分普通方法；

例如：index_controller.inc => IndexController、users_controller.inc => UsersController。

3.4、编写视图模板

Skihat 没有采用第三方的模板插件，也没有为模板设计专门的标签，而仅仅是普通的 php 代码，这是为了提高运行效率和减少学习难度。

创建 app/views/guestbooks/index.stp 文件，编写代码如下：

```
<!doctype>
 <html lang="zh-cn”>"
    <head><title>Guest books</title>
        <meta http-equiv="content-type" content="text/html;charset=utf-8" />
    </head>
    <body>
        <h1>Guest books</h1>
        <?php $guestbooks = $this[ ‘ guestbooks']->page(0,1)->order('created desc')->fetchObjects(); ?>
        <ul>
        <?php foreach ($guestbooks as $guestbook) { ?>
            <li> [<?php echo $guestbook->user; ?>]
<?php echo $guestbook->content; ?></li>
        <?php } ?>
        </ul>
        <a href="/?controller=guestbooks&action=write">发布留言</a>"
    </body>
</html>
```

通常控制器对应 views 下的目录，活动对应文件名称，模板的扩展名使用.stp（skihat template）。

3.5、配置数据库

直接访问网站将会产生 500 错误，表示服务器内部发生错误，这是因为我们还没有进行设置数据库的连接信息，打开 app/boots/config.inc，分别配置以以下三个值：

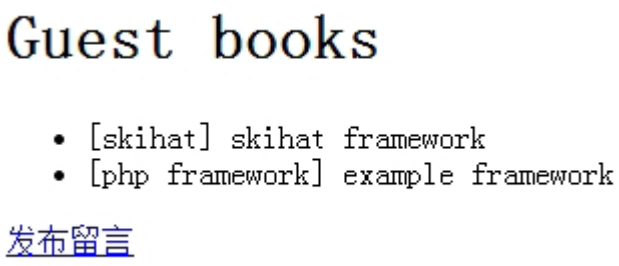
- host：MySql 服务器地址，本机使用 127.0.0.1；
- dbname：数据库名称 ，使用 guestbooks；
- user：数据库访问账户，root 表示超级账户；

■ pass：数据库密码，请使用实际的值：

```
Skihat::write(‘kernels/databases’,array (
        ‘default’  => array(
            SKIHAT_IOC_CLASS =>  ‘kernels.databases.engines.mysql_pdo_engine’,
             ‘conf’ =>
‘mysql:host=127.0.0.1;dbname=test;&user=root&pass=root&charset=utf8’
        )
);
```

在数据库内部使用 pdo 方式实现，配置的参数间使用&分隔，第一个&前面的部分作为 dns 配置值，在 mysql_pdo_engine 中需要指定用户名 user、密码 pass 和 charset 编码格式三个参数。

执行请求：http://www.example.com/?controller=guestbooks，将会显示执行的结果。



3.6、发布留言

3.6.1、Skihat 请求参数与控制器之间的对应关系

前面的示例中，仅对于 guestbooks 控制器的默认活动进行了编写，而在实际过程中我们经常都需要开发多个控制器和活动，这就需要了解请求和控制器、活动之间的对应关系：

示例：

访问地址：http://www.example.com/?controller=users&action=edit

处理活动：app/controllers/users_controller.inc(IndexController-> editAction)

访问地址：http://www.example.com/?action=edit

处理活动：app/controllers/index_controller.inc(IndexController->editAction)

由示例可见，最后决定执行那一个控制器和活动是由 controller 和 action 参数所决定。如果没有指定 controller 和 action 参数，将会使用默认值 index 替换。

例如：

访问 地址：http://www.example.com

处理活动：app/controllers/index_controller.inc(IndexController-> indexAction)。

3.6.2、编写 editAction 方法

打开 app/controllers/guestbooks_controller.inc 增加以下方法：

```
public function writeAction() {
    if ( $this->isPost()) {
        $guestbook = new Guestbook($this->form('guestbook'));
        if ($guestbook->save()) {
            $this->text('发布留言成功');
        } else {
            $this->text('发布留言失败，请检查发布内容');
        }
    }
}
```

代码中，isPost 方法判断当前请求是否为 POST 请求，text 方法用于直接输出文本。

3.6.3、编写 write.stp 视图模板

编写好控制器代码后，还需要有视图模板的支持，创建 app/views/guestbooks/write.stp，并编写以下代码：

```
<!doctype>
<html lang="zh-cn">
    <head>
        <title>发布留言</title>
            <meta http-equiv="content-type" content="text/html;charset=utf-8" />
```

```
    </head>
    <body>
      <h1>发布留言</h1>
     <form action="/?controller=guestbooks&action=write" method="post">
        <p><label for="user">用户：</label>
            <input type="text" name="guestbook[user]" id="user" />
        </p>
        <p><label for="email">邮件：</label>
            <input type="email" name="guestbook[email]" id="email" />
        </p>
        <p><label for="content">内容：</label>
            <textarea name="guestbook[content]" id="content"></textarea>
        </p>
        <p>
            <input type="submit" value="提交" />
            <a href="/">返回</a>
        </p>
    </form>
    </body>
</html>
```

3.6.4、测试执行结果

访问：http://www.example.com/?action=write，录入发布的留言信息并提交，如果没有错误发布后将会提示留言成功，退回到首页将会显示最新的留言内容。

提交后如果成功将显示提交留言信息成功。

3.7、编写后台管理程序

3.7.1、包(package）参数

前面的内容主要讲解控制器和活动两个重要参数，这两个参数决定响应的控制器和活动方法，而在 Skihat 中还有一个非常重要的 package（包）参数，package 参数决定调用那一个目录下的控制器：

示例：

访问地址：http://www.example.com/?controller=index&action=index

执行活动：app/controllers/index_controller (IndexController->indexAction)

访问地址：http://www.example.com/?package=admins&controller=index

执行活动：app/controllers/admins/index_controller(IndexController->indexAction)

实际上 package 参数不仅仅允许单级目录，还允许指定多级目录：

示例：

访问地下：http://www.example.com/?package=admins.reports

执行活动：app/controllers/admins/reports/index_controller(IndexController->indexAction)

因为有包参数，在 Skihat 中建议根据不同需求开发不同的包来满足应用需求。例如：后台管理功能、用户中心等都可以使用包来进行组合。

3.7.2、编写控制器

在 controllers 目录中创建 admins 目录，并且创建文件 guestbooks_controller.inc，编写以下代码：

```
class GuestbooksController extends ApplicationController {
    public function indexAction() {
        $this['guestbooks'] = Guestbook::fetchAll();
    }
```

```
    public function deleteAction() {
        if ( $id = $this->query('id')) {
            if (Guestbook::deleteAll(intval($id))) {
                $this->text('删除留言成功');
            } else {
                $this->text('删除留言失败');
            }
        }
    }

    public function actionModels() {
        parent::actionModels();
        Skihat::import('models.guest_book',SKIHAT_PATH_APP);
    }
}
```

3.7.3、编写视图模板

因为控制器属于包，因此创建视图模板时也需要加入包的目录信息，创建文件app/views/admins/guestbooks/index.stp，编写以下代码：

```
<!doctype>
<html lang="zh-cn">
    <head>
        <title>Guest books</title>
            <meta http-equiv="content-type" content="text/html;charset=utf-8" />
    </head>
    <body>
    <h1>Guest books Administrators</h1>
    <?php $guestbooks = $this['guestbooks']->fetchObjects(); ?>
    <ul>
        <?php foreach ($guestbooks as $guestbook) { ?>
        <li>[<?php echo $guestbook->user; ?>]<?php echo $guestbook->content; ?>
            [<a
href="/?package=admins&controller=guestbooks&action=delete&id=<?php echo
$guestbook->id>">删除</a>]
```

```
            </li>
            <?php } ?>
        </ul>
        </body>
</html>
```

3.7.4、测试执行结果

访问：http://www.example.com/?package=admins&controller=guestbooks 显示后台管理界面，执行删除后将提示删除成功信息，退回查看会看到删除的内容已不存在。

执行删除，提示“删除留言成功”。

注意：为了示例的简单性，这里没有使用 POST 方法执行删除功能，正式开发时应当检查当前请求是否为 POST 方法。

3.8、配置子域名

Skihat 的路由器非常方便，允许根据不同的模块、包、数据将划分成不同的子域名。

默认路由配置

打开 app/boots/routing.inc，默认代码如下：

```
Router::domain(SKIHAT_CURRENT_DOMAIN)
        ->rule(‘#NormalRouterRule’);
```

在默认配置中包含一个域名信息，在该域名下使用普通路由样式和使用普通路由规则。

路由设置方法

在设置路由时，使用以下方法进行设置：

- Router::domain 方法：指定采用的域名，同一个应用允许指定多个域名信息。
- Router->style 方法：指定该域名下面的路由样式，允许 Normal、Rewrite、Phpinfo 和 Restful 四种样式。
- Router->rule 方法：指定该域名下的路由规则，同一域名允许指定多个规则。

配置子域名

在 Skihat 中，建议开发明确域名信息，以检查程序的完整性，打开 app/boots/routing.inc 文件，并编写以下内容：

```
Router::domain('www.example.com')
        ->rule(array(SKIHAT_IOC_CLASS => '#NormalRouterRule',
                        SKIHAT_PARAM_PACKAGE => array(false,'/^admins/')));

Router::domain('admins.example.com')
    ->style('#NormalRouterRule')
    ->rule(array(SKIHAT_IOC_CLASS => '#NormalRouterRule',
                    SKIHAT_PARAM_PACKAGE => array('admins','admins',true)));
```

绑定 Web 服务器域名

重新配置 web 服务器配置，同样将域名绑定到示例的 app/publics 目录下，更新 hosts 文件的设置，将 admins 的值增加到 hosts 文件中，可能设置如下：

- 127.0.0.1 admins.example.com

使用命令符 ping admins.example.com 查看设置是否正确；

检查执行结果

访问 http://admins.example.com/?controller=guestbooks，将显示以下结果：

访问：http://www.example.com/?package=admins&controller=guestbooks，将提示路由规则错误，禁止访问。

3.9、配置示例

本书中的所有示例代码都存放在 testes/examples 目录，执行 test/examples/docs/目录下的 schema.sql 文件，将 web 目录绑定到 testes/examples/app/publics 目录，配置好数据库设置就可以正常使用了。

docs/apache.conf 中包含 Mac 默认 Apache 的配置，可以根据需要修改适应 Apache 服务器；

其它环境的配置请参考网上的相关内容。

3.10、总结

在整个应用中基本完成了对于一个数据表的 CRUD 操作，主要的目的在于对 Skihat 的开发有了一个基本的了解和认识，因此没有使用太过的特性，后面将会对其它的部分做更加详细的说明。

更多的使用技巧可以下载功能更完成的 skihat-fruit 示例。

第二部分：核心服务对象

第 4 章：Skihat 微核类

Skihat 类是框架的核心类、是内核服务的基础，提供动态配置、调试、优先文件、ioc 和 i18n 五个基础服务，其它服务都依赖于这五个核心服务。

4.1、Skihat 类声明

```
# 文件 skihat/kernels/skihat.inc
abstract class Skihat {
    protected function __construct();                    # 禁止实例化
    public static function register(array $paths);       # 注册全局路径
    # 动态配置服务
    public static function write($name,$value);              # 写入动态配置值
    public static function read($name,$default = false);     # 读取动态配置值
    # 动态调试服务
    public static function debug($name,$message);                    # 写入调试
    public static function debugLine($name = false,$file = false);    # 格式化调试输出
    # 优先文件服务：定位、导入和类引用
    public static function locate($file,$paths = false,$ext = SKIHAT_DEFAULT_EXT);
    public static function import($file,$paths = false,$ext = SKIHAT_DEFAULT_EXT);
    public static function using($class,$paths = false,$ext = SKIHAT_DEFAULT_EXT);
    # ioc 创建服务
    public static function ioc($conf,$paths = false,$ext = SKIHAT_DEFAULT_EXT);
    public static function singleton($className,$paths=false,$ext=SKIHAT_FILE_EXT);
    # i18n 国际化服务
    public static function i18n($name,$default = ‘’,$lang = SKIHAT_I18N_LANG);
    # 其它方法
    public static function version();                # 返回版本信息。
}
```

4.2、动态配置服务

动态配置是指在程序的运行过程中，使用执行代码来完成配置信息。与使用配置文件相比具有以下优势和不足：

- 优势：允许在任何位置更新和修改配置的值，允许配置任何类型的值，扩展性好；
- 不足：动态配置的实质就是 PHP 代码，对于一般人的使用来说会比较复杂；

在实际使用过程中，无论使用那种配置都需要有配置的说明文件，因此实际上设置的复杂度相差不大，而动态配置扩展性更好。

4.2.1、Skihat::write 写入配置

- 定义：Skihat::write($name,$value）

根据$name 和$value 参数设置和修改配置信息，如果$name 已存在则覆盖原有值。

示例：

```
Skihat::write（‘kernels/databases’,array（‘default’）);        # 设置值
Skihat::write（‘kernels/databases’,array（‘example’）);        # 设置新的值，覆盖原有值
```

4.2.2、Skihat::read 读取配置

- 定义：Skihat::read($name,$default = false)

根据$name 读取配置值，如果$name 指定的配置信息不存在，返回$default 值。

示例：

```
Skihat::write（‘kernels/databases’,’default’);
Skihat::read（‘kernels/databases’);                    # default
Skihat::read（‘kernels/caches’);                       # false
Skihat::read（‘kernels/caches’,’example’);             # example
```

4.2.3、app/boots/config.inc 全局配置文件

config.inc 是一个特殊的文件，Skihat 运行时会自动导入该文件，因此当需要进行全局

配置时，只需使用 Skihat::write 方法写入新的值，默认文件如下：

```
Skihat::write(‘kernels/caches’,array(
        ‘default’  => array(
            SKIHAT_IOC_CLASS =>  ‘kernels.caches.engines.file_cache_engine’,
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
            ‘paths’ => SKIHAT_PATH_DATA_CACHE),
        ‘runtime’  => array(
            SKIHAT_IOC_CLASS =>  ‘kernels.caches.engine.file_cache_engine’,
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
             ‘paths’  => SKIHAT_PATH_DATA_RUNTIME)
));

Skihat::write(‘kernels.databases’,array(
        ‘default’  => array(
            SKIHAT_IOC_CLASS =>  ‘kernels.databases.engines.mysql_pod_engine’,
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
‘conf’=>‘mysql:host=127.0.0.1;dbname=example;&user=root&pass=123456&charset=utf8’
        )
));
```

如果需要新的配置值，只需要在 config.inc 文件后面使用 Skihat::write 方法写入新的配置。如果需要引用其它文件作为配置的值，还可以与 Skihat::import 方法配合：

```
# boots/other.inc
return array(‘name’  =>  ‘xxx’,’name1’  =>  ‘xxx1’);

# 使用配置文件
Skihat::write(‘example’,Skihat::import(‘boots.other’,SKIHAT_PATH_APP));
Skihat::read(‘example’);  # return array(’name’  =>  ‘xxxx’,’name1’  =>  ‘xxx1’)
```

4.3、动态调试服务

动态调试服务提供在运行时 Skihat 的内部运行状态，主要是各个组件的运行状态，所有的 Skihat 核心组件都支持动态调试服务，开启动态调试必须指定常量值：

- SKIHAT_DEBUG > 0

每一次运行的记录都会被自动记录在 data/loggers/debugs.log 文件中。

4.3.1、Skihat::debug 写入调试信息

- 定义：Skihat::debug($name,$message);

  根据$name 和$message 增加调试信息，允许根据$name 记录多个值。

示例：

```
Skihat::debug(‘databases’,’SELECT * FROM users’);    # 记录执行了 SQL 命令；
Skihat::debug(‘databases’,’SELECT * FROM roles’);    # 记录执行了 SQL 命令；
```

4.3.2、Skihat::debugLine 调试信息格式化输出

- 定义：Skihat::debugLine($name = false,$file = false);

  根据$name 和$file 参数格式化调试信息并返回，如果$file 参数不为空，表示将结果保存到$file 指定文件中。

示例：

```
 # 打印调试信息
echo Skihat::debugLine(‘databases’);

# 将 databases 调试保存到 database.log 文件中。
Skihat::debugLine(‘databases’,’databases.log’);
```

注意：为了保证安全 Skihat 要求$file 不能使用绝对路径，所有的$file 都将记录到 SKIHAT_DATA_LOGGERS 目录下。

4.3.3、内核服务组件的调试名称

在 Skihat 中，内核组件使用目录名称作为调试的名称，这样做的好处在于便于理解和防止命名冲突，已知的名称有：

- kernels/caches：缓存组件
- kernels/transactions：事务组件
- kernels/securities：安全组件
- kernels/databases：数据库组件
- kernels/medias：资源组件

如果开发自定义组件，请遵循以上原则。

4.4、优先文件服务

优先文件服务是 Skihat 框架提供的文件查找和引用服务，目的就是简化文件操作。为了使用方便和消除平台的依赖，使用"."替换目录分隔符，例如：

- kernels.caches.engines.file_cache_engine => kernels/caches/engines/file_cache_engine

4.4.1、文件优先级是什么

文件优先级是指当进行文件操作时，根据查找路径的先后顺序，优先找到前面路径的文件 。例如：在 app 和 skihat 目录下都存在文件 boots/startup.inc，那么如果指定路径时 app 在前，那么 app/boots/startup.inc 将会被优先返回，这就是文件优先原则。

例如：

Skihat::locate(‘boots.startup’,array(SKIHAT_PATH_APP,SKIHAT_PATH_LIBRARY));

根据目录的先后顺序，将返回 SKIHAT_PATH_APP/boots/startup.inc 文件。

在 Skihat 内部很多部分都通过优先文件服务提供在不更改 SKIHAT 库文件的基础上，自定义库代码的功能。

4.4.2、Skihat::register 注册全局路径

在 Skihat 框架中有一个全局路径的设置，当使用优先文件服务时，如果没有指定路径时，Skihat 类会使用全局注册路径进行优先级的查找，优先路径使用 Skihat::register 方法进行设置 。

■ 定义：Skihat::register(array $paths);

根据$paths 参数注册全局文件路径，注意路径中不包含不需要’/‘结局符。

示例：

Skihat::register(array(SKIHAT_PATH_APP,SKIHAT_PATH_LIBRARY));

通常不需要直接调用 Skihat::register 方法，因为在 app/boots/startup.inc 文件中会自动设置全局路径：

```
Skihat::register(array (
    SKIHAT_PATH_APP_MODULE,
    SKIHAT_PATH_APP,
    SKIHAT_PATH_LIBRARY,
    SKIHAT_PATH_VENDOR
));
```

注册或修改全局路径的设置，只需要修改 startup.inc 文件。

4.4.3、Skihat::locate 文件定位服务

■ 定义：Skihat::locate($file,$paths = false,$ext = SKIHAT_DEFAULT_EXT)

根据$file,$paths 和$ext 参数查找文件，如果指定文件存在则返回文件的完整路径，否则返回 false，如果$paths 参数为 false，表示使用全局查找路径。

示例：

```
# 在全局路径中查找 kernels/caches/cache.inc 文件
Skihat::locate(‘kernels.caches.cache’);

# 从 SKIHAT_PATH_LIBRARY 中查找；
Skihat::locate(‘kernels.caches.cache’,SKIHAT_PATH_LIBRARY);

# 从 APP 和 LIBARY 目录中查找；
Skihat::locate(‘kernels.caches.cache’,array(SKIHAT_PATH_APP,SKIHAT_PATH_LIBRARY));
```

以上示例都会返回 SKIHAT_PATH_LIBRARY/kernels/caches/cache.inc 文件。

4.4.4、Skihat::import 文件导入服务

- 定义：Skihat::import($file,$paths = false,$ext = SKIHAT_DEFAULT_EXT);

根据$file，$paths 和$ext 参数查找并引用文件并返回文件返回值，如果文件已引用将直接返回值。如果$paths 为空则表示使用全局路径。

示例：

```
# 将文件 kernels/caches/cache.inc 导入当前运行环境
Skihat::import(‘kernels.caches.cache’);

# 从 SKIHAT_PATH_LIBRARY 中进行查找。
Skihat::import(‘kernels.caches.cache’,SKIHAT_PATH_LIBRARY);

# 从 APP 和 LIBRARY 目录中查找文件并导入当前环境。
Skihat::import(‘kernels.caches.cache’,array(SKIHAT_PATH_APP,SKIHAT_PATH_LIBRARY));
```

使用 import 方法导入文件时有以下两个特点：

- 唯一性：只要 file 和$ext 的参数相同，则进行二次导入时不会重新导入；
- 返回值：如果被导入文件中包含 return 返回值，则返回值为 return 值，否则值为 1；

文件导入的性能问题

在使用优先文件服务时，建议直接指定$paths 值，这是因为在 Skihat 内部会使用 is_file 函数判断文件是否存在，如果每次都从全局路径中查找，将会调用大量的 is_file 影响性能，同时引用也不直观。

错误处理

如果导入的文件文件不存在，将引发 FileNotFoundException 异常。

4.4.5、Skihat::using 类型引用服务

- 定义：Skihat::using($class,$paths = false,$ext = false)

根据$class，$paths 和$ext 参数引用类并返回类名称，如果指定类还未引入将自动引入类。

示例：

如果类型 IndexController 未导入，自动引用 controllers.index_controller 文件。

Skihat::using(‘controllers.index_controller’,SKIHAT_PATH_APP);

如果类 NormalRouterStyle 未导入，自动引用 routers.router_styles.inc 文件。

Skihat::using(‘routers.router_styles#NormalRouterStyle’,SKIHAT_PATH_LIBRARY);

注意：using 方法的返回值为引用的类型名称，示例返回类型为 IndexController 和 NormalRouterStyle。

4.4.5.1、使用$class 参数

Skihat 类中，允许直接通过文件名和指定类名两种方式引用类：

- 直接使用文件名：直接使用文件名，类名为文件名的 Pascal 命名格式；

  例如：Skihat::using(‘controllers.index_controller’);

- 使用#指定类名：使用#指定类名，用于类名与文件名不匹配时；

  例如：Skihat::using(‘routers.router_styles#NormalRouterStyle’);

这两个部分的使用，在前面的示例中已有说明，这里就不在详细介绍。

4.4.5.2、错误处理

如果引用类时指定文件不存在将引发 FileNotFoundException 异常，如果类型不存在将引发 TypeNotFoundException 异常。

4.6、Skihat::ioc 实例创建服务

Ioc 服务是 Skihat 框架提供的一种简化类实例创建的服务，允许根据配置信息实例化类。

- 定义：Skihat::ioc($conf, $paths = false,$ext = SKIHAT_DEFAULT_EXT)

根据$conf，$paths 和$ext 参数创建实例对象并返回。在 ioc 方法内部会自动查找和引用文件，$paths 和$ext 参数分别用于查找的路径和文件扩展名。

4.6.1、使用字符$conf 参数

4.6.1.1、使用文件名

如果文件名和类名相同(类名是文件名的 pascal 命名格式)，则可以直接使用文件名实例化对象。

示例：

```
# 自动引用 controllers.index_controller.inc 文件，并且创建 IndexController 类实例。
Skihat::ioc(‘controllers.index_controller’,SKIHAT_PATH_APP);
```

4.6.1.2、使用#分隔类名

如果文件名与类名不相同，允许在$conf 参数中使用#分隔符指定类名。

示例：

```
# 自动引用 routers.router_styles.inc 文件，并创建 NormalRouterStyle 类实例。
Skihat::ioc(‘routers.router_styles#NormalRouterStyle’,SKIHAT_PATH_LIBRARY);
```

4.6.1.3、仅仅使用#分隔符

有些时候如果我们已经明确了解类已经被引用，那么使用 ioc 方法时则不需要指定文件名，直接使用#指定类名。

示例：

```
# 创建 NormalRouterStyle 类实例
```

Skihat::ioc(‘#NormalRouterStyle’);

4.6.2、使用数组$conf 参数

数组参数能够提供比字符串更多的选项和信息，数组索引提供参数的选项。

4.6.2.1、使用 SKIHAT_IOC_CLASS 常量

使用数组参数时必须指定 SKIHAT_IOC_CLASS 常量，用于指定文件名和类的信息，使用方式与"字符$conf 参数"相同。

示例：

```
Skihat::ioc(array(SKIHAT_IOC_CLASS => ‘routers.router_styles#NormalRouterStyle’));
Skihat::ioc(array(SKIHAT_IOC_CLASS => ‘controllers.index_controller’));
Skihat::ioc(array(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’));
```

以上示例展示了与"字符$conf 参数"相同的服务。

4.6.2.2、使用 SKIHAT_IOC_PATHS 常量

SKIHAT_IOC_PATHS 常量，提供 Skihat::ioc 中类的路径信息，允许使用字符串或数组参数值。

示例：

```
Skihat::ioc(array(SKIHAT_IOC_CLASS =>‘controllers.index_controller’,
                    SKIHAT_IOC_PATHS => SKIHAT_PATH_APP));
Skihat::ioc(array(SKIHAT_IOC_CLASS => ‘controllers.not_found_controller’,
                    SKIHAT_IOC_PATHS => array(
                        SKIHAT_PATH_APP,SKIHAT_PATH_LIBRARY));
```

同样如果指定路径为数组参数，那么创建时也具体文件优先级特性。

注意：如果同时提供 SKIHAT_IOC_PATHS 和$paths 参数，那么将会优先采用

SKIHAT_IOC_PATHS 路径。

4.6.2.3、使用 SKIHAT_IOC_EXT 常量

SKIHAT_IOC_EXT 常量，提供类所在文件的扩展名，默认为 SKIHAT_DEFAULT_EXT，通常不需要指定该值。

4.6.2.4、使用 SKIHAT_IOC_INITIAL 常量

很多情况下，当类完成实例化后，还需要调用初始化方法来完成类的完全初始化，在 Skihat::ioc 中这个功能就是由 SKIHAT_IOC_INITIAL 常量来完成，SKIHAT_IOC_INITIAL 的值表示初始化实例的方法。

示例：

```
Skihat::ioc(array(
    SKIHAT_IOC_CLASS =>  ‘kernels.caches.engines.memcache_engine’,
    SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
    SKIHAT_IOC_INITIAL =>  ‘connect’
));
```

实例化方法后，Skihat::ioc 方法会自动调用 MemcacheEngine 实例的 connect 方法完成初始化。

注意：SKIHAT_IOC_INITIAL 指定的方法，必须允许空参数调用。

4.6.2.5、使用属性键，设置实例属性值

除了 SKIHAT_IOC_XXX 特殊键外，Skihat::ioc 会将其它键会被识别为实例的成员变量，这就使用 ioc 方法能够非常灵活的创建对象实例。

示例：

```
# 文件：app/services/order_service.inc
class OrderService {
    public $conf = false;
```

```
}

$service = Skihat::ioc(array(
    SKIHAT_IOC_CLASS => ‘services.order_service’,
    SKIHAT_IOC_PATHS => SKIHAT_PATH_APP,
    ‘conf’ => 123
));
echo $service->conf;      # 123
```

4.6.3、IOC 与动态配置

Skihat 大部分的内核组件的运行都与 Ioc 相关，这是由于 Skihat 所采用的设计所决定的，在各个组件内部都声明有相关的接口，但如何创建这些接口的实例对象，则是由 Ioc 配置来完成。

示例：

```
Skihat::write(‘kernels/caches’,array(
    ‘default’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.caches.engines.memcache_engine’,
        SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY
    )
));
```

这里的 default 的键的值，就是一个 Ioc 的$conf 参数的值，允许配置满足要求的任意值。

不仅仅是组件的配置实际上路由也是由 ioc 配置来完成的：

```
Router::domain(‘www.example.com’)
    ->style(‘#NormalRouterStyle’)          # 指定类名
    ->rule(‘#NormalRouterRule’);           # 指定类名，允许是数组信息或其它满足
```

4.6.4、Skihat::ioc 错误处理

执行 ioc 方法时，可能会引发以下两种类别的异常：

- FileNotFoundException：如果类的文件没有找到；
- TypeNotFoundException：如果指定的类不存在；

4.6.5、Skihat::singleton 单例方法

单例方法用于管理类的单个实例对象，调用时如果类型不存在，则创建类型的实例，存在则直接返回上一次创建的结果。

- 定义：public static function singleton($className,$paths = false,$ext = SKIHAT_FILE_EXT)

单例方法中的$className 只允许使用 ioc 的字符串参数，不允许使用数组参数。

示例：

```
$service = Skihat::singleton('orders.services.order_service',
                                SKIHAT_PATH_APP_MODULE);
```

示例中，创建 orders.services.order_service 类的实例，多次创建时将会返回第一次创建的实例。

4.7、Skihat::i18n 国际化服务

国际化是一个开发时经常需要解决的问题，在 Skihat 中提供了一套简单有效的方式来解决国际化问题。

- 定义：Skihat::i18n($name,$default = ‘’,$lang = SKIHAT_I18N_LANG)

根据$name 和$lang 参数获取国际化字符串，如果指定字符串不存在则返回默认字符串。

4.7.1、语言文件规范

在 Skihat 中使用文件和关键字作为资源引用的标识，同时语言文件时必须返回包含键值的数组。

示例：

```
# app/langs/default.inc
```

```
return array(
    ‘example’ => ‘Hello Skihat’
);
```

声明 default 语言文件，包含 example 关键字声明。

注意：语言文件必须在 app/langs 目录。

4.7.2、读取语言文件的值

根据语言文件规范，引用一个语言的具体值，需要了解文件位置和关键字，在 Skihat 中使用常量 SKIHAT_PACKAGE_SEPARATOR 作为文件和关键字的分隔符，前一部分表示文件后，一部分表示关键字。

示例：

```
Skihat::i18n(‘default.example’);      # Hello Skihat
```

4.7.3、使用 default 参数

如果指定的语言文件不存在，或指定的关键字不存在将返回$default 的值。

示例：

```
Skihat::i18n(‘default.name’);                # 返回值： ‘’
Skihat::i18n(‘default.name’,’World’);        # 返回值 ： world
Skihat::i18n(‘order.name’);                  # 返回值： ‘’
```

4.7.4、使用$lang 参数

默认情况下$lang 参数的值为 SKIHAT_I18N_LANG，在查找语言文件时会按以下方式进行查找（SKIHAT_I18N_LANG = zh-CN)：

- app/langs/zh-CN/default.inc
- app/langs/default.inc

查找时会优先在$lang 指定的目录中进行查找语言文件，如果查找失败再查找 langs 目录下的语言文件。

示例：

```
# app/langs/zh-CN/default.inc
return array(‘example’ => ‘你好 Skihat’);
Skihat::i18n(‘default.example’);     # 你好 Skihat
```

通过示例可以看出，通过修改 SKIHAT_I18N_LANG 值并且增加新的语言文件，就能实现各种不同的语言支持。

4.8、Skihat::version 返回版本

- 定义：Skihat::version()

返回当前 Skihat 的版本信息，版本信息由二位构成，分别表示主版本和次版本。

示例：

```
Skihat.version();          # 返回值 ： 1.0 beta
```

第 5 章：Skihat 基础类

除了 Skihat 微核类之外，skihat.inc 文件中还有 IocFactory 和 CollectionBase 两个基础类。

5.1、IocFactory 工厂类

IocFactory 是一个组件辅助类，提供针对 ioc 配置的实例管理服务，在 Skihat 中的大部分内核组件，都依赖 IocFactory 进行实例的管理。

5.1.1、IocFactory 类声明

```
# 文件：skihat/kernels/skihat.inc
class IocFactory {
    public function __construct(array $conf);        # 初始化实例集
    public function instance($name);                 # 返回管理实例成员
    public function instances();                     # 返回管理实例名称集合
}
```

5.1.2、IocFactory->__construct 初始化实例值

- 定义：public function __construct(array $conf)

使用$conf 数组初始化实例时，数组的键为实例的名称，值为 Skihat::ioc 的$conf 参数值，提供实例创建的具体信息。

示例：

```
$factory = new IocFactory( array (
    ‘default’ => ‘kernels.caches.engines.file_cache_engine’,
    ‘memecache’ => ‘kernels.cache.engines.memcache_engine’
));
```

初始化 IocFacotry 实例，包含 default 和 memcache 两个管理实例。

5.1.3、IocFacotry->instance 获取实例

- 定义：public function instance($name)

使用 instance 方法返回实例，如果实例还没有创建则调用 Skihat::ioc 方法创建实例，否则直接返回原有的实例。

示例：

```
# 第一次调用，创建 FileCacheEngine 实例并返回。
$factory->instance(‘default’);

# 第二次调用，返回前一次创建的 FileCacheEngine 实例。
$factory->instance(‘default’);
```

错误处理

获取实例时，如果$name 指定的实例不存在，将引发 ConfigException 异常。

5.1.4、IocFactory->instances 获取名称集

- 定义：public function instances()

返回全部的实例配置名称。

示例：

```
# array(‘default’,’memcache’);
$factory->instances();
```

5.1.5、IocFactory 与动态配置

IocFacotry 在组件中，通常与动态配置配合使用，从而获取动态管理和创建实例的能力。

示例：

```
Skihat::write (‘kernels/caches’, array(
        ‘default’ => ‘kernels.caches.engines.file_cache_engine’,
        ‘memcache’ => ‘kernels.caches.engines.memcache_engine’
));
```

这就是一个典型的 IocFacotry 配置，在缓存组件内部通过以下方式使用配置值：

```
$factory = new IocFactory( Skihat::read (‘kernels/caches’,array());
```

通过这种方式缓存组件就获取得了缓存引擎的动态配置能力。

5.2、CollectionBase 类

CollectionBase 类是一个基础类，封装对于 ArrayAccess 接口的基本服务，简化 ArrayAccess 接口类的实现。

5.2.1、CollectionBase 类声明

```
# 文件： skihat/kernels/skihat.inc
class CollectionBase implement ArrayAccess {
    public function offsetExists($name);
    public function offsetGet($name)
    public function offsetSet($name,$value);
    public function offsetUnset($name);

    protected $_vars = array();
}
```

子类使用时，所有通过 ArrayAccess 设置的值，都保存在$_vars 变量中。

示例：

```
class Controller extends CollectionBase {}
```

第 6 章：Skihat 核心函数

Skihat 核心函数是对 PHP 函数库的扩展，全部存放在 skihat/kernels/core.inc 文件中，在框架初始化时自动加载，因此可以在程序的任意位置调用。

6.1、单词命名规范转换函数

6.1.1、camel_upper 函数

- 定义：function camel_upper($words)

将单词$words 转换为 camel 大写规范，即第一个单词首字母小写，其余单词首字母大写。

例如：

camel_upper（‘hello_world’）; # 返回值 ：helloWorld

camel_upper（‘HelloWorld’）; # 返回值 ：helloWorld

6.1.2、camel_lower 函数

- 定义：function camel_lower ($words)

将单词$words 转换为 camel 小写规范，即全部单词小写，单词间使用下划线分隔。

例如：

camel_lower（‘HelloWorld’）; # 返回值：hello_world

camel_lower（‘helloWorld’）; # 返回值：hello_world

camel_lower（‘_hello_world’）; # 返回值：hello_world

6.1.3、pascal 函数

- 定义：function pascal($words)

将单词$words 转换为 pascal 规范，即全部单词首字母大写。

例如：

pascal(‘helloWorld’); # 返回值 ：HelloWorld

pascal(‘hello_world’); # 返回值 ：HelloWorld

6.2、数组扩展函数

6.2.1、array_key_pop 函数

- 定义：function array_key_pop(array &$array,$key,$default = false)

根据$array 索引$key 删除值，并返回删除$key 的值，如果$key 不存在则返回$default 参数值。

示例：

$input = array(‘name’ => ‘skihat’);

array_key_pop($input,’name’); # 返回值：skihat,$input = array()

array_key_pop($input,’age’); # false,$input = array(‘name’ => ‘skihat);

array_key_pop($input,’age’,’man’); # man,$input = array(‘name’ => ‘skihat’);

6.2.2、array_key_value 函数

- 定义：function array_key_value(array &$array,$key,$default = false)

根据数组$array 索引$key 返回值，如果指定$key 的索引不存在，则返回$default 值。

示例：

$input = array(‘name’ => ‘skihat’);

array_key_value($input,’name’); # skihat,$input = array(‘name’ => ‘skihat’);

array_key_value($input,’age’); # false,$input = array(‘name’ => ‘skihat’);

array_key_value($input,’age’,’man’); # man,$input = array(‘name’ => ‘skihat’);

6.2.3、array_key_filter 函数

- 定义：function array_key_filter(array &$array,$callback)

根据$array 和$callback 执行数组过滤，并返回过滤的结果，回调方法签名为 function method($key)。

示例：

```
$input = array(1,2,’name’ => ‘skihat’);
$input = array_key_filter($input,’is_numeric’);     # input = array(‘name’=>‘skihat’);
```

6.2.4、array_append 函数

- 定义：function array_append(array &$array,array $join)

将 join 数组的值，追加到$input 后面，并返回新的结果。

示例：

```
$input = array(1,2,3);
$join = array(4,5,6);

array_append($input,$join);  # input = array(1,2,3,4,5,6);
```

6.2.5、array_join_str 函数

- 定义：function array_join_str(array &$arr, $key, $join)

为指定数组的值附加字符串，如果指定$key 不存在，则设置附加字符串为数组$key 的值。

示例：

```
$array = array('name' => 'chenJ’);

# $array = array('name' => 'chenJ programmer)
array_join_str($array,'name',' programmer');
```

```
 # $array = array('name' => 'chenJ programmer','age' => '18');
array_join_str($array,'age','18’ );
```

6.2.6、array_html_attrs 函数

- 定义：function array_html_attrs (array $arr)

  将$arr 数组生成为 html 属性标签，KEY 将生成为属性名称、VALUE 生成为属性值。

示例：

```
# name="emp_id" class="header"
array_html_attrs(array(‘name’  => 'emp_id','class' => 'header'));
```

注意：属性值将会自动调用 htmlspecialchars 进行过滤处理。

6.3、客户端请求函数

6.3.1、client_method 函数

- 定义：function client_method()

  返回客户端的请求方法，因为 http 不允许 put、delete 等方法，因此在 client_method 方法中会检查 SKIHAT_PARAM_METHOD 参数值，并返回。

示例：

```
GET /index.php      # client_method => get
POST /index.php    # client_method => post

# client_method =>  ‘put’;
POST /index.php ($_POST  =  array (SKIHAT_PARAM_METHOD =>  ‘put’)
```

6.3.2、client_url 函数

- 定义：function client_url()

  返回客户端请求的完整 url 地址。

示例：

client_url() -> http://www.example.com/index.php?action=edit

GET /index.php?action=edit

6.3.3、client_refer 函数

- 定义：function client_refer()

  返回客户请求的引用地址。

示例：

GET /index.php?action=index

client_refer -> http://www.example.com/index.php?action=index

GET /index.php?action=edit

6.3.4、client_script_name 函数

- 定义：function client_script_name()

  返回请求的脚本文件名称。

示例：

GET /index.php?action=index # client_script_name -> index.php

6.3.5、client_address 函数

- 定义：function client_address()

  返回客户请求的 IP 地址。

6.3.6、client_browser 函数

- 定义：function client_browser()

  返回客户端请求的浏览器信息。

6.3.7、client_languages 函数

- 定义：function client_languages()

  返回客户端请求的语言建议数组。

6.4、其它函数

6.4.1、enable_cached 函数

- 定义：function enable_cached()

  返回一个布求值，判断当前环境是否允许开启缓存。

6.4.2、full_path 函数

- 定义：function full_path()

  根据调用参数，生成完整路径信息。

示例：

full_path(‘app’,’controllers’); # app/controllers

6.4.3、full_file 函数

- 定义：function full_file()

  根据调用参数，生成满足 skihat 需求的文件调用格式。

示例：

full_file(‘views’,’index’,’edit’); # views.index.edit

6.4.4、not_null 函数

- 定义：function not_null($val)

  非空值判断，如果$val 为非空值则返回 true，否则返回 false。

示例：

not_null (false); # false

not_null (‘’); # false

not_null(0); # false

6.4.5、not_numeric 函数

- 定义：function not_numeric($val)

  非数值判断，如果$val 非数值则返回 true，否则返回 false。

示例：

not_numeric(1); # true

not_numeric(‘abc’); # false

第三部分：应用服务层

第 7 章：前端控制器组件（Applications)

前端控制器组件是整个 MVC 模型的入口，负责将客户的请求分发到控制器中，并提供相应的支持。

7.1、ApplicationBase 类

ApplicationBase 类是前端控制器的基类，提供请求的分发和处理，通常我们不直接使用 ApplicationBase，而是使用子类 Application。

7.1.1、ApplicationBase 类声明

```
# 文件： skihat/applications/application_base.inc
class ApplicationBase {
    public function initialize();          # 初始化
    public function dispatching();         # 分发
    # 全局处理方法
    public function globalError($errno,$errstr,$errfile,$errline);      # 全局错误处理
    public function globalException($ex);                               # 全局异常处理
    public function globalComplete();                                   # 全局完成处理
    # 静态方法
    public static function instance();          # 获取唯一实例
    public static function request();           # 获取请求实例
    public static function response();          # 获取响应实例
}
```

7.1.2、ApplicationBase->initialize 初始化前端控制器

- 定义：public function initialize ()

初始化前端控制器，执行初始化方法。在方法内部将初始化路由组件、应用请求和响应实例，并调用全局配置文件。

7.1.2、ApplicationBase->dispatching 分发客户请求

- 定义：public function dispatching ()

  根据请求参数，将请求分发到控制器中，并调用控制器的 invoke 方法，执行控制器。

7.1.3、ApplicationBase 全局处理函数

全局处理函数提供针对 PHP 默认行为的处理，由 initialize 方法，自动绑定到运行环境中。

7.1.3.1、ApplicationBase->globalError 全局错误处理

- 定义：public function globalError($errno,$errstr,$errfile,$errline);

  在 Skihat 应用执行过程中，如果发生错误，将调用 globalError 方法进行全局处理。

7.1.3.2、ApplicationBase->globalException 全局异常处理

- 定义：public function globalException($ex)

  在 Skihat 应用执行过程中，如果发生任何异常并且没有捕捉，将调用 globalException 方法进行全局处理。

7.1.3.3、ApplicationBase->globalComplete 全局完成处理

- 定义：public function globalComplete()

  当 Skihat 应用以任何情况结束时，都将执行 globalComplete 方法。

7.1.4、ApplicationBase 静态方法

ApplicationBase 类使用单例模式构造，可以直接通过静态方法访问唯一实例和重要的

成员变量。

7.1.4.1、ApplicationBase::instance 获取唯一实例

- 定义：public static function instance()

  返回 ApplicationBase 类的唯一实例。

7.1.4.2、Application::request 获取应用请求

- 定义：public static function request()

  返回当前 ApplicationRequest 实例，表示应用的请求信息。

7.1.4.3、ApplicationBase::response 获取应用响应

- 定义：public static function response()

  返回当前 ApplicationResponse 实例，表示应用的响应信息。

7.2、Application 类

Application 类是 ApplicationBase 类的子类，虽然在 appservs 目录中声明，但也是属于前端组件的一部分。

7.2.1、默认 Application 类

默认 Application 类，没有进行任何操作，直接继承自 ApplicationBase 类。

```
# 文件 skihat/appservs/application.inc
Skihat::import(‘applications.application_base’,SKIHAT_PATH_LIBRARY);

class Application extends ApplicationBase {

}
```

7.2.2、自定义 Application 类

根据 Skihat 提供的优先文件原则，Skihat 框架允许创建 app/appservs/application.inc 文件，替换默认 Application 类声明。

示例：

```
# 文件：app/appservs/application.inc
Skihat::import(‘applications.application_base’,SKIHAT_PATH_LIBRARY)

class Application extends ApplicationBase {
    public function initialize() {
        parent::initialize();

        # do something
    }
}
```

在自定义 Application 类中，允许对默认的行为进行重写或修改，而不用修改库所提供的默认实例。

注意：在 app/appservs/application.inc 文件中必须声明 Application 类，否则将引发 TypeNotFoundException 异常。

7.2.3、获取全局实例

通常调用 ApplicationBase 提供的表态方法 instance、request 和 response，允许在任何位置获取应用实例、请求和响应对象的实例。

- Application::instance：获取 Application 实例；
- Application::response ：取 ApplicationResponse 实例；
- Application::request：获取 ApplicationRequest 实例；

7.3、ApplicationRequest 类

ApplicationRequest 类封装了客户端的请求信息，包括$_GET、$_POST 和$_FILES 变量信息，并提供了适当的扩展。

7.3.1、ApplicationRequest 类声明

```
# 文件：skihat/applications/application_request.inc
class ApplicationRequest {
    public function initialize($get,$post,$files);  # 初始化实例

    # GET 参数查询方法
    public function query($name,$default = false);    # 获取查询字符串($_GET)的值
    public function queryAll();                       # 返回所有查询字符串

    # POST 参数查询方法
    public function form($name,$default = false);     # 获取表单（$_POST）参数值
    public function file($name);                      # 获取文件($_FILES) 参数值

    # 预定查询方法，返回模块、包、控制器、活动、格式。
    public function module($default = SKIHAT_PARAM_DEFAULT_MODULE);
    public function package($default = SKIHAT_PARAM_DEFAULT_PACKAGE);
    public function controller($default = SKIHAT_PARAM_DEFAULT_CONTROLLER);
    public function action($default = SKIHAT_PARAM_DEFAULT_ACTION);
    public function fmt($default = SKIHAT_PARAM_DEFAULT_FORMAT);

    # 返回请求头信息
    public static function(header,$default = false);    # 返回请求头信息
}
```

7.3.2、访问 ApplicationRequest 实例

通常我们不需要自己创建 ApplicationRequest 类的实例，而是访问由 ApplicationBase 类创建的实例 Application::request 方法。

示例：

```
$request = Application::request();
$name = $request->query(‘name);
```

7.3.2.1、在控制器中访问实例

如果在控制器活动中使用应用请求实例，直接使用控制器的$request 成员变量。

示例：

```
# app/controllers/request_controller.inc
public function indexAction() {
    $name =  $this->request->query(’name’);
    $this->text($name);
}
```

7.3.2.2、在视图模板中访问实例

在视图模板中使用应用请求，可以直接使用局部变量$request。

示例：

```
# app/controllers/request/template.stp
<?php echo $request->query(‘name’); ?>
```

7.3.3、GET 查询参数

ApplicationRequest 的 GET 参数，并不与$_GET 变量完成一致，这是因为在 Application 中，使用路由进行了参数的转换操作。

7.3.3.1、ApplicationRequest->query 获取单个查询字符串的值

- 定义：function query($name,$default = false)

根据$name 返回查询字符串的值，如果参数$name 指定的查询字符串不存在，则返回$default 值。

示例：

```
# app/controllers/request_controller.inc
public function queryAction() {
    $this->text($this->request->query(‘name’));
}
```

在活动方法中，在控制器中查询请求参数通常不直接使用$request->query 方法，而是使用控制器 query 方法，简化使用方式。

示例：

```
public function queryAction() {
    $this->text($this->query(‘name’));
}
```

了解更多内容，请查看控制器 query 方法。

7.3.3.2、ApplicationRequest->queryAll 获取全部查询字符串的值

- 定义：public function queryAll()

  返回当前请求的全部参数值。

示例：

```
# app/controllers/request_controller.inc
public function queryAllAction() {
    $this->autoRender = false;
    print_r($this->request->queryAll());
}
```

7.3.4、POST 表单参数

从 POST 表单提交的参数包含表单参数和文件参数，分别使用 form 和 file 方法。

7.3.4.1、ApplicationRequest->form 获取表单值

- 定义：function form($name,$default = false);

根据$name 和$default 获取表单值，如果指定$name 值不存在，则返回$default 值。

示例：

```
# app/controllers/request_controller.inc
public function formAction() {
    if ($this->isPost()) {
        $this->text($this->request->form('name'));
    }
}
```

与 query 方法类似，在控制器中通常也不直接使用$request 的 form 方法，而是使用控制器的 form 方法，简化使用方式。

示例：

```
# app/controllers/request_controller.inc
public function formAction() {
    if ($this->isPost()) {
        $this->text($this->form('name'));
     }
}
```

7.3.4.2、ApplicationRequest->file 获取上传文件值

- 定义：public function file($name)

根据$name 返回提交的表单文件数据，如果数据不存在返回 false。

例如：

```
# app/controllers/request/file.stp
public function fileAction() {
    if ($this->isPost()) {
        print_r($this->request->file('file'));
    }
```

```
}
```

与 query 类似，在控制器内部通常也不直接使用$request 的 file 方法，而是使用 file 方法，简化使用方式。

示例：

```
# app/controllers/request/file.stp
public function fileAction() {
    if ($this->isPost()) {
        print_r($this->file('file'));
     }
}
```

上传文件数组

上传文件数组时需要注意，ApplicationRequest 内部会对文件数组进行处理，以使用于控制器中使用。

示例：

```
# app/controllers/request_controller.inc
public function multipleAction() {
    if ($this->isPost()) {
        print_r($this->file('file'));
    }
}
```

使用 ApplicationRequest 的 file 方法时，转换后返回值如下：

```
array(‘file’ => array(
    array(‘name’ => ‘example.gif’,’size’ => 1000,
            ’type’ => ‘gif’,’tmp_name’ => ‘1.tmp’,’error’ => 0)
    array(‘name’ => ‘ab.gif’,’size’ => 2000,
        ’type’ => ‘gif’,’tmp_name’ => ‘2.tmp’,’error’ => 0)
));
```

7.3.5、ApplicationRequest::header 获取自定义头

- 定义：public static function header($name,$default = false)

  返回请求自定义$name 的请求信息，如果$name 不存在则返回$default 值。

7.3.6、获取常用请求参数

使用 module、package、controller、action 和 fmt 方法用于获取常见的请求参数。

7.4、ApplicationResponse 类

ApplicationResponse 类提供应用的响应服务，在 Skihat 中严格要求 ApplicationResponse 作为响应的唯一方法，ApplicationResponse 类同要也不需要创建，而是由 ApplicationBase 类负责创建和实例化。

7.4.1、ApplicationResponse 类定义

```
# 文件：skihat/applications/application_response.inc
class ApplicationResponse {
    # 自定义响应头
    public function header($name,value);        # 设置自定义响应头
    public function allHeaders();               # 返回全部自定义响应头

    # 客户端缓存方法
    public function enableCache($minute = 30, $lasted = false); # 开启客户端缓存
    public function disableCache();                             # 禁止客户端缓存

    # 设置响应
    public function statusCode($statusCode = null);      # 设置 HTML 响应码
    public function contentType($contentType = null);    # 设置响应类型
    public function contentBody($contentBody = null);    # 设置响应内容

    # 输出和复制实例
    public function render();                            # 输出响应内容
```

```
    public function copy(ApplicationResponse $response);      # 复制内容
}
```

7.4.2、访问 ApplicationResponse 实例

同 ApplicationRequest 一样，访问 ApplicationResponse 也可以在以下位置访问：

- 控制器：使用$this->response 成员变量引用实例；
- 视图模板：使用$response 局部变量引用实例；
- 其它位置：使用 Application::response 静态方法引用实例；

更多内容请查看 ApplicationRequest 类。

7.4.3、自定义 header 响应内容方法

自定义响应头是在响应实例内部使用 header 方法直接输出。

7.4.3.1、ApplicationResponse->header 设置自定义响应头

- 定义：public function header($name,$value)

根据$name 和$value 设置响应的头信息，在 Skihat 中设置自定义响应头（内部使用 header 函数输出）。

7.4.3.2、ApplicationResponse->allHeaders 获取全部自定义响应头

- 定义：public function allHeaders()

返回全部已定义的头信息。

7.4.3、缓存方法

7.4.3.1、ApplicationResponse->enableCache 开启客户端缓存

- 定义：public function enableCache($minute = 30,$lasted = false)

根据$minute 和$lasted 开启客户端缓存选项，$minute 表示缓存时间（分），$lasted 表示

缓存的最后更新时间。

示例：

```
# app/controllers/response_controller.inc
public function cacheAction() {
    $this->response->enableCache(30);
}
```

7.4.3.2、ApplicationResponse->disableCache 禁止客户端缓存

- 定义：public function disableCache()

  禁止开启客户端缓存选项，默认情况下 Skihat 客户端没有开启客户端缓存。

7.4.4、响应内容方法

7.4.4.1、ApplicationResponse->statusCode 设置 HTML 响应码

- 定义：public function statusCode($statusCode = null)

  根据$statusCode 设置 HTTP 响应状态码，如果$statusCode 为空值，表示返回当前状态码，默认状态码为 HTTP 200。

在控制器中使用 statusCode 方法，使用控制器的$response 成员变量：

示例：

```
# app/controllers/response_controller.inc
public function statusAction() {
    $this->autoRender = false;
    $this->response->statusCode(404);
}
```

- **使用控制器 error 方法设置错误码**

  如果返回错误代码响应，可以在控制器中使用 error 方法，同使用$response 成员变量相比，使用 error 方法更加简洁和方便。

示例：

```
# app/controllers/response_controller.inc
public function errorAction() {
    $this->error(404);
}
```

- **在视图模板中设置状态码**

  除了在控制器中使用状态码外，还经常需要在模板中使用状态码的设置。

示例：

```
# skihat/views/__errors/404.stp
$response->statusCode(404);
<html>
    <head><title>404 Not Found</title></head>
    <body>
        <h1>404 Not Found Page</h1>
        <p>Sorry not fund page</p>
    </body>
</html>
```

7.4.4.2、ApplicationResponse->contentType 设置响应类型

- 定义：public function contentType($contentType = null)

  设置或获取响应内容类型，如果$contentType 为空，则表示返回当前值，默认响应类型为 text/html。

同 statusCode 类似，控制器中使用$response 成员变量：

示例：

```
# app/controllers/response_controller.inc
public function jsonAction() {
    $this->autoRender = false;
    $this->response->contentType(‘application/json’);
    $this->response->contentBody(‘[name:skihat]’);
```

```
}
```

在视图模板中设置响应类型，直接使用$response 局部变量。

示例：

```
# skihat/views/__ctltpls/json.stp
$response->contentType(‘json/application');
echo $this[‘json'];
```

7.4.4.3、ApplicationResponse->contentBody 设置响应内容

- 定义：public function contentBody($body = null)

获取或设置响应内容，如果$contentBody 为非空值，则表示仅仅获取响应内容值。通常我们不直接使用 contentBody 而是由控制器的 render 方法内部调用，当然我们也可以直接设置值。

示例：

```
# app/controllers/response_controller.inc
public function jsonAction() {
    $this->autoRender = false;
    $this->response->contentType(‘application/json’);
    $this->response->contentBody(‘[name:skihat]’);
}
```

7.4.5、输出和复制方法

7.4.5.1、ApplicationResponse->render 输出响应结果

- 定义： public function render()

将响应内容输出到客户端，通常我们不直接使用，Application 的 globalComplete 方法自动调用。

7.4.5.2、ApplicationResponse->copy 复制实例

- 定义：public function copy(ApplicationResponse $response)

  根据$response 执行响应内容复制信息。

第 8 章：控制器组件（Controllers）

控制器组件是 MVC 中非常重要的部分，负责处理客户端的请求和响应请求结果，控制器组件主要由控制器和活动过滤器两个部分组成。

8.1、Controller 控制器类

在一个应用中允许声明多个控制器，并且在每一个控制器中允许声明多个活动，控制器的名称允许相同，但必须属于不同的模块或包。

8.1.1、Controller 类定义

```
# 文件：skihat/controllers/controller.inc
class Controller extends CollectionBase {
    # 常量
    const DEFAULT_VIEW_CLASS = ‘views.theme’;      # 默认视图类
    const DEFAULT_ERROR_TEMPLATE = ‘error’;      # 默认错误模板

    # 公有实例
    public $request = null;              # ApplicationRequest 请求实例
    public $response = null;             # ApplicationResponse 响应实例
    public $autoRender = false;          # 自动绘制选项

    # 请求查询方法
    public function query($name,$default = false);    # 获取查询参数
    public function form($name,$default = false);     # 获取表单参数
    public function file($name);                      # 获取表单文件参数
    public function isGet();             # 返回布尔值，表示是否为 GET 请求；
    public function isPost();            # 返回布尔值，表示是否为 POST 请求；

    # 控制器执行入口方法
    public function invoke(ApplicationRequest $request,ApplicationResponse $response);
```

```
        # 视图模方法
        public function render($options = array());  # 绘制视图模板
        public function actionView();                # 活动视图声明

        # 视图模板响应方法
        public function message($title,$sign = SKIHAT_MESSAGE_INFO,$actions = array(),$message = ‘’);
        public function redirect($url,$delay = 0);   # 执行客户端跳转
        public function text($text);                 # 使用文本内容响应客户端
        public function json($json);                 # 使用 json 内容响应客户端
        public function error($code);                # 使用错误代码响应客户端

        # 活动内部方法
        public function actionFilters();             # 活动过滤器声明
        public function actionModels();              # 活动模型声明
        public function actionBlank();               # 空白活动方法
    }
```

8.1.2、声明控制器

8.1.2.1、声明控制器

控制器是响应客户请求的最小组织单元，为了保证 Application 能够正确的分发控制器内部的活动中，在 Skihat 中控制器必须严格遵循以下规范：

- 控制器目录：控制器文件必须存放在 app/controllers 或 app/modules/xxxx/controllers 目录内（xxxx 为模块名称）；
- 控制器包：在 controllers 目录内部允许子目录，这些子目录被称为包，包允许多级嵌套，理论上嵌套层级没有限制；
- 控制器文件名：控制器文件必须以"控制器名称＋_controller.inc"命名。例如：index_controller.inc；
- 控制器类名：控制器类必须是文件名的 pascal 命名规范。例如：index_controller.inc -> IndexController；
- 控制器命名冲突：Skihat 允许控制器具有相同的名称，但必须在不同的模块或不同的包中；
- 控制器父类：原则上控制器类都是直接或间接继承自 Controller 类；

模块相关控制器，请查看模块化开发。

示例：

```
# app/controllers/guestbooks_controller.inc
class IndexController extends ApplicationController {
}

# app/controllers/admins/guestbooks_controller.inc
class IndexController extends ApplicationController {
}
```

注意：每个控制器文件内部仅允许声明一个控制器类。

8.1.2.2、声明活动

在一个控制器内部允许声明多个活动，每个活动负责处理一种用户请求，活动声明必须遵循以下规范：

- 活动方法：活动方法命名，必须采用以 Action 为后缀，例如：indexAction、editAction；
- 方法参数：活动方法只能使用空参数，例如：indexAction()、editAction()

示例：

```
#   app/controllers/guestbooks_controller.inc
class GuestbooksController extends ApplicationController {
    public function indexAction() {
        # do something
    }
}
```

注意：Skihat 的活动方法与访问级别无关，建议采用公有方法。

8.1.3、访问控制器

从 Web 客户端到 Skihat 内部控制器，有四个非常特殊的参数，它们决定最终调用的控

制器和活动：

- SKIHAT_PARAM_MODULE：模块参数，如果模块为空值，将会调用 app/controllers 目录下的控制器；
- SKIHAT_PARAM_PACKAGE：包参数，如果包参数不为空，将会调用 controllers 的子目录内部，如果为多级子目录，使用"."分隔目录（参考 Skihat 优先文件）；
- SKIHAT_PARAM_CONTROLLER：控制器参数，指定控制器名称，默认值为 index；
- SKIHAT_PARAM_ACTION：活动参数，指定活动名称，默认值为 index；

在 Skihat 中建议参数名称都使用 camel 命名规范，在 Skihat 内部会根据需要自动转换为所需的命名规范。

示例：

地址：http://www.example.com/
调用：app/controllers/index_controller.inc(IndexController->index)

地址：http://www.example.com/?package=admins&controller=users&action=edit
调用：app/controllers/admins/users_controller.inc(UsersController->edit)

地址：http://www.example.com/?package=admins.reprots
调用：app/controllers/admins/reports/users_controller.inc (UsersController->index)

地址：http://www.example.com/?controller=security_roles&action=assign_role
调用：app/controllers/security_roles_controller.inc(SecurityRoleController->assignRole)

注意：通过示例可以看出，在 Skihat 内部同样遵循 Skihat 的命名规范。

8.1.4、获取请求参数

8.1.4.1、Controller->request 成员变量

- 定义：public $request = null;

提供控制器对全局 ApplicationRequest 实例的访问，我们可以通过它获取当前应用的完整请求信息。

8.1.4.2、Controller->query 查询参数值

- 定义：public function query($name,$default = false)

根据$name 和$default 获取查询字符串的值，如果$name 指定值不存在，则返回$default 值。

示例：

```
# 文件：app/controllers/admins/guestbooks_controller.inc
public function deleteAction() {
    if ($id = $this->query('id')) {
        if (Guestbook::deleteAll(intval($id))) {
            $this->text('删除留言成功');
        } else {
            $this->text('删除留言失败');
        }
    }
}
```

使用 query 方法是为了简化对$request->query 方法的调用，内部依然会调用$request->query 方法。

8.1.4.3、Controller->form 表单参数值

- 定义：public function form($name,$default = false)

根据$name 和$default 获取提交表单的值，如果$name 指定的值不存在，则返回$default 值。

示例：

```
# 文件：app/controllers/guestbooks_controller.inc
public function writeAction() {
    if ( $this->isPost()) {
        $guestbook = new Guestbook($this->form('guestbook'));
    if ($guestbook->save()) {
        $this->text('发布留言成功');
    } else {
        $this->text('发布留言失败，请检查发布内容');
```

```
        }
    }
```

虽然也可以直接采用$_POST 全局变量，但建议采用 form 方法，这样更符合 Skihat 开发规范。

8.1.4.4、Controller->file 获取表单文件

- 定义： public function file($name)

  根据$name 获取提交文件字段值，如果指定$name 字段不存在则返回 false。

示例：

```
# 文件：app/controllers/request_controller.inc
public function fileAction() {
    if ($this->isPost()) {
                print_r($this->file('file'));
        }
}
```

8.1.4.5、Controller->isGet GET 请求判断

- 定义：public function isGet()

  返回一个布尔值，表示当前请求是否为 GET 方法，如果是则返回 true，否则返回 false。

8.1.4.6、Controller->isPost POST 请求判断

- 定义：public function isPost()

  返回一个布尔值，表示当前请求是否为 POST 方法，如果是则返回 true，否则返回 false。

正确使用 isPost 方法

在使用时根据 HTTP 协议规范，查询数据使用 GET 命令，更新数据使用 POST 命令，因此需要更新数据时必须使用 isPost 方法判断是否为 POST 命令。

8.1.5、控制器与视图模板

在 Skihat 中所有的客户端响应结果都是由视图模板生成。通常控制器将视图所需的变量或处理的结果传递给视图对象，再由视图对象调用模板文件来生成响应的结果，并填充到 AppicationResponse 实例中。

Skihat 默认采用 Theme 类为视图类，因此这里主要讲解 Theme 与控制器之间的操作。

8.1.5.1、视图模板文件

当访问请求后，控制器会在 app/views 目录中，根据以下方式查找模板文件：

- SKIHAT_PARAM_CONTROLLER/SKIHAT_PARAM_ACTION.stp
- SKIHAT_PARAM_PACKAGE/SKIHAT_PARAM_CONTROLLER/SKIHAT_PARAM_ACTION.stp

例如：

http://www.example.com/?controller=users&action=edit
模板文件：app/views/users/edit.stp

http://www.example.com/?package=admins&controller=users&action=edit
模板文件：app/views/admins/users/edit.stp

http://www.example.com/?package=admins.reports&controller=users
模板文件：app/views/admins/reports/users/index.stp

主题模板文件

Theme 类包含一个非常特殊的成员变量$theme，用于指定当前视图所采用的主题，使用$theme 成员变量后，Theme 类会首先从 app/themes 目录下查找对应的文件，如果指定文件不存在则会重新在普通模板中查找，查找规范如下：

- $theme/SKIHAT_PARAM_CONTROLLER/SKIHAT_PARAM_ACTION.stp
- $theme/SKIHAT_PARAM_PACKAGE/SKIHAT_PARAM_CONTROLLER/SKIHAT_PARAM_ACTION.stp

示例：

```
# http://www.example.com/?controller=users&action=edit
模板文件：app/themes/defaults/users/edit.stp

# http://www.example.com/?package=admins&controller=users&action=edit
模板文件：app/themes/defaults/admins/users/edit.stp

# http://www.example.com/?package=admins.reports&controller=users&action=edit
模板文件：app/themes/defaults/admins/reports/users/edit.stp
```

注意：主题名称为 defaults，Theme 类默认主题为 false。

8.1.5.2、ArrayAccess 设置视图变量

在控制器的处理过程中，经常需要向视图模板传递各种变量，在 Skihat 中利用 ArrayAccess 接口来完成:

- offsetExists：返回一个布尔值，判断是否包含指定名称的视图变量；
- offsetGet：返回一个视图变量，指定变量不存在则返回 null；
- offsetSet：设置一个视图变量，如果变量已存在则覆盖原有值；
- offsetUnset：删除已存在的视图变量；

开发过程中，最常使用的是 offsetSet 和 offsetGet 方法设置和获取值。

示例：

```
# app/controllers/guestbooks_controller.inc
public function indexAction() {
    $this['guestbooks'] = GuestBook::fetchAll();
}
```

模板中使用视图变量

在模板中使用$this 变量引用当前模型实例，在视图实例中同样使用 ArrayAccess 方法获取或设置视图变量的值。

示例：

```
# app/views/guestbooks/index.stp
<?php $guestbooks = $this['guestbooks']->page(0,1)->
                        order('created desc')->fetchObjects(); ?>
```

在视图模板中，变量的名称与控制器中一致。

8.1.5.3、Controller->$autoRender 自动绘制

- 定义：public $autoRender = false

自动绘制选项，如果值为 true 表示活动方法执行完成后，将自动调用模板填充响应内容，false 表示不调用模板进行填充，默认值为 true。

通常不需要进行任何操作控制就能正确调用模板信息，这是因为默认 autoRender 变量值为 true，如果不需要自动模板可以将 autoRender 属性设置为 false。

示例：

```
# app/controllers/request_controller.inc
public function queryAllAction() {
    $this->autoRender = false;
    print_r($this->request->queryAll());
}
```

注意：调用控制器 render、text、json、error、redirect 方法会自动设置 autoRender 为 false 值，因此默认模板不会被填充。

8.1.5.4、Controller->render 手动绘制模板

- 定义：public function render(array $options = array())

根据$options 输出控制器的执行结果，如果$options 为字符串表示指定活动选项。

通常我们不需要手动调用 render 方法，但如果模板的显示没有按默认规定的执行，就可以使用 render 方法手动控制。

- **默认$options 参数**

当我们不传递任何参数时，Theme 类会自动为我们生成一组参数选项，包括以下内容：

- SKIHAT_PARAM_MODULE => $request->module()
- SKIHAT_PARAM_PACKAGE => $request->package()
- SKIHAT_PARAM_CONTROLLER => $request->controller()
- SKIHAT_PARAM_ACTION => $request->action()
- IView::OPT_PASS => false
- IView::OPT_TEMPLATE => false
- IView::OPT_BASE_PATHS => false

这一组选项决定了 Theme 类查找模板文件的路径，我们也可以根据实际的需要设置新的值覆盖默认值（$request 表示当前请求实例）。

- **使用$options 参数**

使用$options 参数非常简单，只需要使用对应的键替换默认选项。

示例：

```
# app/controllers/guestbooks_controller.inc
public function editAction()  {
    $this->render(array(SKIHAT_PARAM_ACTION =>  ‘write’));
}
```

- **使用$options 字符串参数**

如果直接使用字符串选项，Skihat 会自动转换为 SKIHAT_PARAM_ACTION 选项。

示例：

```
# app/controllers/guestbooks_controller.inc
public function newAction() {
        $this->render('write');
}
```

- **使用 pass 选项**

pass 选项是一个非常重要的参数，用于解决模板内部的分类问题。

假设我们需要开发一个产品分类的列表，默认情况下我们只需要创建一个视图模板，但

很多时候我们希望能够根据分类展示不同的视图模板（比如：色调原因，推广原因等），这时候就可以使用 pass 选项。

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function indexAction() {
        $cat = $this->query('cat', 0);
        $this->render(array('pass' => $cat));
}
```

访问地址：http://www.example.com/?controller=ctlview&cat=2

根据以上代码，Theme 会按以下方式查找模板文件：

- app/views/categories/1/index.stp
- app/views/categories/index.stp

查找模板文件时，如果 1/show.stp 模板不存在则会使用 show.stp 模板，因此不用担心需要为每一种分类都创建模板，只需要为特殊的分类创建。

数组 pass 参数，pass 参数允许使用数组参数指定多层个层次：

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function subcatAction() {
        $cat = $this->query('cat', 0);
        $sub = $this->query('sub', 0);
        $this->render(array('pass' => array($cat, $sub)));
    }
```

- 访问地址：http://www.example.com?controller=ctlview&action=subcat&cat=1&sub=2
- 模板文件：
  - app/views/categories/1/2/subcat.stp
  - app/views/categories/1/subcat.stp
  - app/views/categories/subcat.stp

- **使用$options 参数选项 template**

通常 Theme 会根据控制器和活动查找路径，但如果需要直接指定路径文件，就可以使

用 template 选项指定模板文件。

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function templateAction() {
    $this->render(array('template' =>  'ctlview.1.2.subcat'));
}
```

- 访问地址：http://www.example.com/?controller=ctlview&action=template
- 模板文件：app/views/categories/1/2/subcat.stp

通过示例可以看出指定 template 属性时允许使用"."指定子目录，同时会从模板文件的根目录开始查找。

- **使用$options 参数选项 paths**

通常情况下我们不需要使用 OPT_BASE_PATHS 指定模板的基本路径，但如果模板的路径不是标准路径，则可以使用 OPT_BASE_PATHS 指定模板基本路径。

示例：

```
public function indexAction() {
    $this->render(array('paths' => '/www/www.example.com/vhosts/'));
}
```

使用 paths 选项后，模板会从/www/www.example.com/vhosts/'目录下进行查找。

8.1.5.5、Controller->actionView 自定义视图类

- 定义：public function actionView()

返回一个 Ioc 配置，提供视图类型配置信息（默认为 Theme)。

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function actionView() {
```

```
    return array(SKIHAT_IOC_CLASS => self::DEFAULT_VIEW_CLASS,
                'theme' => 'defaults' );
}
```

如果需要在全局设置视图信息，可以直接在 ApplicationController 类中进行设置。

示例：

```
# 文件：app/appservs/application_controller.inc
public function actionView() {
    return array(SKIHAT_IOC_CLASS => self::DEFAULT_VIEW_CLASS,
                ’theme’ => ‘site’);
}
```

8.1.6、视图模板响应方法

视图模板响应方法是一组为了简化响应内容输出而设计的方法，执行时会自动调用相关的模板自动完成。

8.1.6.1、Controller->message 设置响应消息

- 定义：public function message($title,$sign = SKIHAT_MESSAGE_INFO,$actions = array(),$message = ‘’);

经常我们都需要将处理的结果展示到响应的结果中，在 Skihat 中使用控制器的 message 方法实现。

示例：

```
# app/controllers/ctlview_controller.inc
public function messageAction() {
    $this->message(‘显示提示信息’,SKIHAT_MESSAGE_SUCCESS);
    $this->redirect(‘result’);
}

# app/views/ctlview/message.stp
$this->helpers(‘core’);
```

_text_messge($this);

设置好消息后，在 Skihat 的视图模板中，还需要使用_text_message 方法输出消息内容。

注意：这里的视图模板为 result.stp，而不是 message.stp 也能够正常输出，这是因为控制器消息的能够跨越活动进行展示，并且仅仅会展示一次。

- **$sign 参数**

$sign 参数指定响应的标识信息，允许使用 SKIHAT_MESSAGE_XXX 作为响应代码，也可以使用自定义的字符串。

- **$actions 参数**

$actions 参数用于指定相关活动，即当前响应消息的相关性操作活动，格式如下：

array ($url => $title,$url1 => $title1);

_text_message 方法会自动生成相关活动的显示方式。

- **$message 参数**

$message 消息用于显示当前响应消息的更多说明信息。

8.1.6.2、Controller->redirect 执行客户端跳转

- 定义： public function redirect($url,$delay = 0)

根据$url 和$delay 参数进行客户端跳转响应，如果$url 参数为字符串表示跳转到当前控制器的指定活动。

示例：

```
# app/controller/ctlview_controller.inc
public function messageAction() {
    $this->message(‘显示提示信息’,SKIHAT_MESSAGE_SUCCESS);
    $this->redirect(‘result’);
}
```

直接跳转到 result 活动中，通常使用是会在执行特定操作完成后再进行跳转。

- **数组$url 参数**

如果$url 为数组参数，则在$url 内部将会调用 Router::url 方法生成完整的 url 参数，并跳转到生成后的 url 地址。

- **字符串$url 参数**

如果$url 为字符串参数，则在$url 内部将会生成为 array(SKIHAT_PARAM_ACTION => $url)，指定跳转到当前控制器的 url 地址。

- **完整$url 字符串参数**

如果$url 以 http 或 https 作为前缀，则 Skihat 认为这是完整 URL 地址，将直接跳转到$url 参数。

- **$delay 参数**

设置跳转的延迟时间。

- **正确使用 redirect 方法**

redirect 方法不是一个中断方法，活动内部的其它代码将会继续执行，因此使用后，请调用 return 跳出活动的执行。

示例：

```
# app/controllers/ctlview_controller.inc
public function redirectAction() {
    $this->redirect(‘index’);
    # 这里的代码将会继续执行
    die(‘ok’);
}
```

关于数组$url 的更多的信息，请查看 Router::url 方法。

8.1.6.3、Controller->text 使用文本内容响应客户端

- 定义：public function text($text);

  使用文本$text 响应客户请求。

通常在活动中将使用对应的模板来输出响应的结果，但如果只需要简单文本作为响应内容就可以直接使用 text 方法。

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function textAction () {
    $this->text(‘Hello Skihat’);
}
```

8.1.6.4、Controller->json 使用 json 格式响应客户端

- 定义：public function json($json);

  使用$json 格式数据响应客户端请求。同 text 方法类似，只是返回的内容为 json 格式数据：

示例：

```
# 文件：app/controllers/ctlview_controller.inc
public function jsonAction() {
    $this->json(json_encode(array(‘name’ => ‘Hello Skihat’)));
}
```

注意：使用 json 响应时，$json 参数必须是已转换的 json 格式数据 。

8.1.6.5、Controller->jsonp 使用 jsonp 格式响应跨站请求

- 定义：public function jsonp($json,$contentType = ‘application/json’);

  使用$json 和$contentType 响应客户段 jsonp 响应，json 表示响应的内容，$contentType 表示响应的格式。

在 Skihat 中允许将不同的模块或包划分为不同的域名，因此经常都需要进行跨域内容的访问。通常浏览器根据同源原则，不允许进行跨域的内容访问，但 javascript 可以跳过这一限制，在 IE6 中就有非标准的 jsonp 允许进行跨域内容的访问。

示例：

```
# app/controllers/ctlview_controller.inc
public function jsonpAction() {
    $this->jsonp(json_encode(array(‘name’ => ‘Hello Skihat’));
}
```

虽然 jsonp 的默认响应格式为 json，但也允许使用非常 json 格式响应。

示例：

```
# app/controllers/ctlview_controller.inc
public function jsonpTextAction() {
    $this->jsonp(‘Hello Skihat’,’html/text’);
}
```

基本上常见的 javascript 框架都提供了对 jsonp 的请求支持，下面是 jquery 的访问方式：

示例：

```
$.getJSON(‘http://www.example.com/?controller=ctlview&action=jsonp&callback=?'),{},
    function(data) {
        // 这里的 data 是已经处理好的 json 格式数据，允许直接使用。
    }
);
```

注意：callback=?是必须的参数，这是由 jquery 框架所要求，更多内容请参数网上的 jsonp 的相关内容。

8.1.6.6、Controller->error　使用错误码响应客户端

- 定义：public function error($errorCode);

  根据$errorCode 提示错误信息。如果程序执行过程中需要使用 HTTP 错误码响应客户，

就可以直接使用 error 方法。

示例：

```
# 文件：app/controller/ctlview_controller.inc
public function errorAction() {
    if ($error = $this->query('errorCode', 404)) {
        $this->error($error);
    }
}
```

$code 错误码与 HTTP 协议相同，常用的包括：

- 400：请求错误；
- 401：未授权；
- 403：禁止访问；
- 404：文件不存在；
- 500：服务器内部错误；

同 redirect 类似，error 也应当使用 return 跳出活动代码的执行。

8.1.7、活动定制方法

活动定制方法允许对控制器的一些默认行为重新进行定义，这组方法都是以 action 为前缀。

8.1.7.1、Controller->actionFilters 活动过滤器声明

- 定义：public function actionFilters();

返回当前控制器，活动过滤器集合（IOC 配置数组，默认为空），详情请查看活动过滤器。

8.1.7.2、Controller->actionModels 活动模型声明

- 定义：public function actionModels();

actionModels 方法用于导入活动模型信息，在控制器中导入模型，可以直接在活动内部

导入或重写 actionModels 方法导入所需的模型。

示例：

```
# app/controllers/guestbooks_controller.inc
public function actionModels() {
    # 注意：必须调用 parent::actionModels
    parent::actionModels();
    Skihat::import(‘models.guest_book’,SKIHAT_PATH_APP);
}
```

注意：在 Skihat 中不能在控制器文件头部直接导入模型文件，将引发 ApplicationModel 未声明错误消息。

8.1.7.3、Controller->actionBlank 空白活动方法

- 定义：public function actionBlank()

如果当前请求活动在控制器中未找到，那么控制器会自动调用 actionBlank 方法，默认 actionBlank 方法执行 404 错误提示，如果需要改写默认行为可以重写 actionBlank 方法：

示例：

```
class IndexController extends ApplicationController {
    public function indexAction() {
    }

    public function actionBlank() {
        $this->text(’error’);
    }
}

# http://www.example.com/?controller=index&action=edit
执行结果：error 字符串提示
```

8.1.8、Controller->invoke 方法

- 定义：public function invoke(ApplicationRequest $request,ApplicationResponse $response)
  使用$request 和$response 参数，执行控制器。

当 ApplicationBase 执行分发时，并不直接发送到活动的 action 方法，而是调用 invoke 方法，再由 invoke 方法执行内部的活动执行，因此如果需要修改默认的活动流程可以重写 invoke 方法。

示例：

```
class IndexController extends ApplicationController {
    public function invoke(ApplicationRequest $request,ApplicationResponse $response) {
        $this->text(‘invoke’);
    }
}

http://www.example.com/?controller=index&action=edit    # 响应 invoke
http://www.example.com/?controller=index  # 响应 invoke
```

8.2.、IActionFilter 活动过滤器

活动过滤器是一组实现了特殊接口的类，这些类能够为控制器提供额外的服务和功能，Skihat 框架提供的过滤器都存放在 skihat/controllers/filters 目录下。

8.2.1、IActionFilter 接口声明

```
interface IActionFilter    {
    function initialize(Controller $context); # 实始化过滤器。
    function invoke($actionMethod);        # 活动方法回调函数
    function render(array &$options);      # 绘制方法回调函数
    function complete();                        # 请求执行完整回调函数
}
```

IActionFilter 接口非常简单，详细说明如下：

- initialize：使用$controller 初始化 IActionFilter 接口实例。
- invoke：使用$actionMethod 参数，执行活动处理前回调，如果返回值为 true 表示继续执行，false 表示中断执行。
- render：使用$options 参数，执行模板渲染前回调，如果返回 true 表示继续执行，false 表示中断执行。
- complete：控制器执行完成方法，当控制器执行完成后将回调本方法。

8.2.2、在控制器中声明和访问过滤器

在控制器中声明过滤器非常简单，只需要重写 Controller->actionFilters 方法，并返回 Ioc 配置数组。

示例：

```
# app/controllers/cache_controller.inc
public function actionFilters() {
    return array(
        ‘cache’  => array(
                SKIHAT_IOC_CLASS =>  ‘controllers.filters.cache_filter’,
                    SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY
        );
    );
}
```

声明过滤器后，在活动方法中，我们可以直接使用成员变量的方式来访问过滤器的实例。

示例：

```
public function indexAction() {
    $this->cache->xxx =  ‘xxxx’;  # 注意 cache 与 Ioc 数组的 key 相同。
}
```

注意：因为过滤器实例的访问采用成员变量的方式，因此设置 actionFilters 的 Ioc 数组时要防止 key 值与控制器的成员变量相冲突。

8.2.3、CacheFilter 缓存过滤器

使用 CacheFilter 过滤器能够为活动的 GET 请求提供缓存支持，当第一次请求活动后，活动的执行结果将会自动存储到缓存引擎中，后面的访问都将从缓存引擎中读取信息。

示例：

```
# app/controllers/cache_controller.inc
class CacheController extends ApplicationController {
    public function indexAction() {
        $this->text('Time:', date(SKIHAT_I18N_DATE_TIME));
    }

    public function actionFilters() {
        return array( 'cache' => array(
                    SKIHAT_IOC_CLASS => 'controllers.filters.cache_filter’,
                    SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY
                )
            );
     }
}
# 访问：http://www.example.com/?controller=cache
```

执行方法后，第二次访问时将会前面的缓存响应结果，如果失败请检查 data/caches 目录是具有写入权限。

在 CacheFilter 中包含三个重要的成员言变量：

- actionMethods：活动方法名称，指定那些活动方法将会被缓存，默认不包含任何活动方法，”*”表示缓存全部活动。
- engine：引擎名称，指定采用那个缓存引擎提供的缓存服务，默认为 default；
- expire：缓冲有效期（秒），默认为 1 个小时；

使用时可以根据实际的需要在 Ioc 配置中设置相应的值，更多信息请查看内核缓存组件。

8.2.4、RequestFilter 请求过滤器

在 Skihat 的 Theme 类中，允许指定 OPT_PASS 参数来使用不同的模板，而 RequestFilter 过滤器用于处理两个特殊参数的 pass 值。

- client_method：客户请求方法；
- SKIHAT_PARAM_FORMAT：客户请求数据格式；

示例：

```
public function actionFilters() {
    return array('requestFilter' => array(
        SKIHAT_IOC_CLASS => 'controllers.filters.request_filter',
        SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBARY
    ));
}
```

请求：http://www.example.com/index.php?fmt=ajax

将会选择以下模板：

- index/ajax/index.stp
- index/index.stp

如果执行 POST 请求，则还会加下 POST 参数：

- index/ajax/post/index.stp
- index/ajax/index.stp
- index/index.stp

如果在方法中也指定有参数，参数会自动合并到后面。

示例

```
public function indexAction() {
    $this->render('pass' => 1);
}
```

将会选择以下模板执行：

- index/ajax/1/index.stp
- index/ajax/index.stp
- index/index.stp

8.2.5、PaginateFilter 分页支持

分页是开发 Web 应用经常都需要实现的功能，在 Skihat 中通过 PaginateFilter 和 ModelFetch 类之间的配合使用来实现。

示例：

```
class PaginateController extends ApplicationController {
    public function indexAction() {
        $this[‘guestbooks’] = GuestBook::fetchAll();
    }

    public function actionModels() {
        parent::actionModels();
        Skihat::import(‘models.guest_book’,SKIHAT_PATH_PARAM);
    }

    public function actionFilters() {
        return array(‘paginate’ => array(
                SKIHAT_IOC_CLASS => ‘controllers.filters.paginate_filter’,
                SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
                ‘var’ => ‘guestbooks’
    }
}
```

在 Skihat 中有一组与分页相关的请求参数：

- SKIHAT_PARAM_PAGINATE：当前页，默认参数名称为 page；
- SKIHAT_PARAM_PAGINATE_SIZE：分页大小，默认参数名称为 page_size；
- SKIHAT_PARAM_PAGINATE_SORT：分页排序字段，默认参数名称为 page_sort；
- SKIHAT_PARAM_PAGINATE_DIR：排序方式(asc、desc)，默认参数名称为 page_dir；

当访问：http://www.example.com/?controller=paginate&page=2&page_size=20&page_sort＝name 后，Skihat 会自动为当前请求进行分页处理。

通常在实际开发中，PaginateFilter 还会与 paginate 分页助手配合使用，详细内容请查看分页助手。

除了 var 成员变量外，在 PaginateFilter 类中，还有以下三个成员变量：

- size：默认分页大小；
- sort：默认排序字段；
- dir：默认排序方式；

当年请求参数与默认参数发生冲突时，优先采用请求参数的值。

8.2.6、SecurityFilter 安全过滤器

安全过滤器提供当前应用的安全服务支持，自动判断当前控制器是否允许访问和提供其它的方案支持服务。

8.2.6.1、SecurityFilter 类声明

```
class SecurityFilter implements IActionFilter {
    public $authenticates = false;          # 指定需要身份验证的活动方法。
    public $authorizates = false;           # 指定需要安全授权的活动方法。
    public $authentication = 'default';     # 指定身份验证名称。
    public $authorization   = 'default';    # 指定安全授权名称。

    public function $defaultUrl = '/';      # 身份验证完成后默认跳转路径。
    public function $loginUrl = '/';        # 身份验证失败后的登陆 URL 地址。

    # 自定义方法
    public function identity();             # 返回当前用户身份信息；
    public authenticate($identity,$autoRedirect = true);    # 使用$identity 执行身份验证
    public authorizate($sro,$rule = 'sum');      # 执行安全授权检查

    public function register($identity);    # 注册身份验证信息
    public function unregister();           # 注销身份验证信息；

    public function autoRedirect();         # 执行自动跳转；

    # IActionFitler 方法
    public function initialize(Controller $controller);
    public function invoke($actionMethod);
    public function render(array &$options);
```

```
    public function complete();
}
```

SecurityFilter 类是在控制器中对于安全组件的补充和扩展，相关方法都与 Security 组件紧密联系。

8.2.6.2、使用 SecurityFilter 类

使用 SecurityFilter 类非常简单，只需要控制器中声明相关信息。

示例：

```
class IndexController extends ApplicationController {
    public function actionFilters() {
        'securityFilter' => array(
            SKIHAT_IOC_CLASS => 'controllers.filters.security_filter',
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
            'loginUrl' => '/?action=login',
            'defaultUrl' => '/',
            'authentications' => array('indexAction','editAction','deleteAction'),
            'authorizations' => array('indexAction','editAction','deleteAction')
    }
}
```

如果是一组控制器都需要声明安全控制，可以直接在 application_controller.inc 中声明父类，在父类中进行管理。

8.2.6.3、身份验证

在 SecurityFilter 中支持两种方式的身份验证，authenticate 方法和 reigster 方法自定义身份标识。

在 SecurityFilter 中，当访问需要身份验证才能访问的活动时，SecurityFilter 会自动进行判断，如果还没有进行身份验证就会自动将当前请求转换到 loginUrl 指定的地址；如果当前活动已验证，则会自动为当前控制器设置 identity 标识。

示例：

```
    public function indexAction() {
        print_r($this['identity']);            # 当前的身份信息。
    }
```

因为 identity 设置视图变量，因此在视图中也可以直接使用 identity 访问当前用户信息。

示例：

```
# index.stp
<?php print_r($this[‘identity’]); ?>
```

8.2.6.3.1、authenticate 方法

- 定义：public function authenticate($identity,$autoRedirect = true);

根据$identity 和$autoRedirect 方法执行验证，如果验证成功将返回 true。

在 authenticate 方法内容会自动调用 Security::authenticate 方法进行身份验证，如果验证成功将会通过$_SESSION 保存当前用户的标识信息。

示例：

```
public function loginAction() {
    if ($this->isPost()) {
        if (!$this->SecurityFilter->authenticate($this->form(‘user’)) {
            $this->message(‘用户名或密码错误’,SKIHAT_MESSAGE_FAIL);
        }
    }
}
```

$autoRedirect 参数

$autoRedirect 参数表示是否执行自动跳转，当我们访问需要进行身份检查的活动时，如果活动不允许访问，SecurityFilter 组件会自动记录访问的地址。

当 autoRedirect 参数为 true 时，会自动跳转到最后访问的需要身份验证的地址，如果没有最后访问的地址，SecurityFilter 方法会自动访问 defaultUrl 变量指定的地址。

8.2.6.3.2、register 绑定自定义身份

随着越来越多的开放平台的加入，很多时候授权工作并不是应用自身完成，而是由开方平台来完成，这时候就可以使用 register 方法将验证后的 identity 保存到 SecurityFilter 中。

- 定义：public function register($identity);

将$identity 指定的信息，保存为用户标识信息。

示例：

```
public function weiboAction() {
    if ($user = User::fetch(array(‘sns’ => ‘xxxxx’),array(),User::FETCH_OBJ)) {
```

```
            $this->securityFilter->register($user);
            $this->SecurityFilter->autoRedirect();
        } else {
            $this->message('授权失败!',SKIHAT_MESSAGE_FAIL);
        }
    }
```

注意：在 authenticate 方法中会自动调用 register 方法保存验证的用户标识。

9.2.6. 3.3、identity 方法

- 定义：public function identity()

返回当前验证用户的信息，如果还没有完成验证将返回 false。

9.2.6.3.4、unRegister 方法

- 定义：public function unRegister();

从 SecurityFilter 中注销用户标识信息。

示例：

```
public function logoutAction() {
    $this->SecurityFilter->unRegister();
}
```

9.2.6.3.5、authenticates 变量

$authenticates 变量用于指定需要进行身份验证的活动名称，允许指定以下值：

- false：表示不执行验证；
- array：指定需要验证的活动方法名称，例如：array('indexAction');
- *：表示验证全部活动信息。

默认 authenticates 变量的值为 false，表示不进行验证。

9.2.6.3.6、authentication 变量

$authentication 变量用于指定使用的验证器名称，默认为 false，我们也可以直接设置不同的值。

示例：

```
public function actionFilters() {
    return array('securityFilter' => array(
            SKIHAT_IOC_CLASS => 'controllers.filters.security_filter',
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
            'authentication'    => 'ucenter'
    ));
}
```

8.2.6.4、权限授权检查

除了进行身份验证外，在 SecurityFilter 控制器中，还包含一组用于进行权限检查的方法，允许进行自动检查和手动检查。

8.2.6.4.1、自动授权检查

在 SecurityFilter 中有一个特殊的变量 authorizates，用于指定需要进行授权检查的活动，允许使用以下值：

- false：不进行授权检查；
- array：允许指定多个需要检查的活动，例如：array('indexAction')；
- *：全部都需要检查。

默认值为 false，表示不进行任何检查。

当前设置 authenticates 不为 false 时，SecurityFilter 会自动生成一组 sro，并调用 Security 组件进行判断。

- $model-$controller-action
- $model-$controller
- $model

当进行判断时，sro 的值都大于 1，则允许继续执行，否则将会返回 403 禁止访问页。

8.2.6.2.4.2、手动授权检查

- 定义：public function authorize($sro,$rule = 'sum')

根据$sro 和$rule 进行授权检查并返回结果。

示例：

```
public function indexAction() {
    if ($this->SecurityFilter->authorize('audit') <= 0) {
        $this->error(403);
    }
}
```

8.2.6.2.4.3、authorization 变量

$authorization 变量用于指定使用的授权方式，默认使用 default，也可以根据实际的需要指定其它的授权方式。

示例：

```
public function actionFilters() {
    return array('securityFilter' => array(
            SKIHAT_IOC_CLASS => 'controllers.filters.security_filter',
            SKIHAT_ICO_PATHS => SKIHAT_PATH_LIBRARY,
            'authorization'          => 'ucenter'
    ));
}
```

8.2.7、自定义过滤器

自定义过滤器非常简单，只需要实现 IActionFilter 接口并提供相应的服务。

示例：

```
# 文件：app/controllers/filters/call_filter.inc
class CallFilter implement IActionFilter {
    public function initialize(Controller $context) {
        echo  'initialize;';
    }

    public function invoke($actionMethod) {
        echo  'actionMethod:',$actionMethod,';';
        return true;
    }

    public function render(array &$options) {
        echo  'render;';
        return true;
    }
```

```
    public function complete() {
        echo ‘complete;’;
    }
}

# 文件：app/controllers/call_filter_controller.inc
class CallFilterController extends ApplicationController {
    public function indexAction() {
        $this->autoRender = false;
        echo ‘indexAction;’;
    }

    public function actionFilters(){
        return array(‘call’ => ‘controllers.filters.action_call_filter’);
    }
}
```

执行：http://www.example.com/?controller=call_filter 将会依次输出：

1. initialize
2. invoke:indexAction
3. render
4. indexAction
5. complete

8.3、Skihat 特殊控制器

8.3.1、not_found_controller 控制器不存在异常

NotFoundController 控制器是一个特殊的控制器，如果请求的控制器不存在，ApplicationBase 会自动将请求转发到 NotFoundController 控制器。

示例：

```
# 访问：http://www.example.com/?controller=not_exists
```

not_exists_controller 控制器不存在，执行 not_found_controller 控制器。

8.3.1.1、默认 NotFoundController 控制器

默认 NotFoundController 在文件 skihat/controllers/not_found_controller.inc 中声明，默认 NotFoundController 执行使用 404 错误码显示错误。

```
class NotFoundController extends ApplicationController {
    public function invoke(ApplicationRequest $request,ApplicationResponse $response) {
        # do something
        $this->error(404);
    }
}
```

8.3.1.2、自定义 NotFoundController 控制器

自定义 NotFoundController 非常简单，只需要声明 app/controllers/not_found_controller.inc 文件，并提供 NotFoundController.inc 类声明。

示例：

```
class NotFoundController extends ApplicationController {
    public function invoke(ApplicationRequest $request,ApplicationResponse $response) {
        # do something
    }
}
```

注意：如果控制器文件存在，但文件内部的控制器声明不存在，将引发 TypeNotFoundException，而不是执行 NotFoundController 控制器。

8.3.2、error_controller 错误控制器

同 NotFoundController 一样，ErrorController 也一个特殊控制器，当 Skihat 执行过程中发生任何错误时，Skihat 将自动调用 ErrorController 控制器。

8.3.2.1、默认 ErrorController 控制器

默认 ErrorController 在文件 skihat/controllers/error_controller.inc 中声明，默认 ErrorController 执行 500 错误提示。

```
class ErrorController extends ApplicationController {
    public $ex = null;
    public function invoke(ApplicationRequest $request,AppicationResponse $response) {
        # do something
        $this->error(500);
    }
}
```

注意：ErrorController 必须包含成员变量 ex，表示当前发生错误的异常信息，ApplicationBase 类会自己将异常设置给 ex 成员变量。

8.3.2.2、自定义 ErrorController 控制器

自定义 ErrorController 控制器非常简单，只需要声明 app/controllers/error_controller.inc 文件。

示例：

```
class ErrorController extends ApplicationController {
    public function invoke(ApplicationRequest $request,ApplicationResponse $response) {
        # do something
    }
}
```

第 9 章：模型组件（Models）

模型组件提供数据的访问和业务处理，在 Skihat 中采用标准 ActiveRecord 模式实现模型类。

9.1、模型类的声明

9.1.1、Model 类声明

```
class Model implements ArrayAccess,IModelMetaSupport {
    # 自定义配置信息
    public static function __config();
    # 模型初始和状态
    public function __construct($props = array(),
                                $forceUpdate = FALSE, $forceLoad = FALSE);
    public function isNew();        # 当前模型是否为创建；
    public function isUpdate();     # 当前模型是否为更新；
    public function isDelete();     # 当前模型是否为删除；

    # 获取和设置属性值；
    public function key();                          # 获取主键值
    public function __get($name);                   # 获取属性值
    public function __set($name,$value);            # 设置属性值
    public function offsetExists($name);            # 判断是否存在属性值
    public function offsetGet($name);               # 返回属性值
    public function offsetSet($name,$value);        # 设置属性值
    public function offsetUnset($name);             # 删除属性值
    public function changeProps(array $props);      # 指定多属性值
    public function originalProp($name);            # 返回原始属性值

    public function originalProps();        # 返回全部属性值
    public function currentProps();         # 返回当前属性值
    public function json();                 # 返回 json 格式值
```

```
        # 更新和保存数据
        public function save(array syntax = array());              # 保存值
        public function delete(array $syntax = array());           # 删除当前字模型
        public function cancel();                                  # 取消更新值；

        # 验证和错误处理
        public function validate($onRule = false,array &$syntax);  # 验证值
        public function addError($prop,$error);       # 附加字段错误信息
        public function error($prop);                 # 返回字段错误信息
        public function allErrors();                  # 返回所有错误信息
        public function clearErrors($error = array()); # 清除错误信息

        # 模型配置值
        public static function &meta();           # 返回 ModelMeta 描述信息
        public static function primaryKey();      # 返回主键名称
        public static function database();        # 返回数据库


        # 创建多条记录
        public static function createAll(array $fields,array $syntax    = array());
        # 更新多条记录
        public static function updateAll($fields,$conditions,array $syntax = array());
        # 删除多条记录
        public static function deleteAll($conditions,array $syntax = array());
        # 查询记录
        public static function fetchAll($conditions = array(),array $syntax = array());
        # 查找单条记录
        public static function fetch($conditions = array(),array $syntax = array());
        # 查找单条记录
        public static function field($field,$conditions,$syntax = array());

        # 数据命令查询、执行命令
        public static function query($cmd,array $args = array(),
                $fetchStyle = self::FETCH_STYLE_ASSOC);
        public static function execute($cmd,array $args = array());
    }
```

通过类的完整声明可以看出，Model 类提供了非常多的模型功能。

9.1.2、IModelMetaSupport 接口声明

```
interface IModelMetaSupport extends IDataSyntaxSupport {
    const CONFIG_METHOD = ‘__config’;          # 模型元数据配置方法;
    # 默认值
    const DEFAULT_PRIMARY_KEY = ‘id’;          # 默认主键名称;
    const DEFAULT_PRIMARY_INCREMENT = true;    # 默认主键自增值;
    # 数据库配置声明
    const META_DATABASE = ‘database’;          # 数据库声明;
    const META_TABLE    = self::SYNTAX_TABLE;     # 数据表声明;
    const META_TABLE_ALIAS = self::SYNTAX_TABLE_ALIAS;    # 数据表别名;

    # 字段常量配置声明
    const META_PRIMARY_KEY = ‘primaryKey’;                  # 主键名称;
    const META_PRIMARY_INCREMENT = ‘primaryIncrement’;  # 主键自增;
    const META_DEFAULTS = ‘defaults’;                       # 默认字段;
    const META_READONLY = ‘readonly’;                       # 只读字段;
    const META_VIRTUAL = ‘virtual’;                         # 虚拟字段声明;

    # 关联配置声明
    const META_LINKS = ‘links’;                     #  模型关系声明
    const META_LINKS_CLASS = ‘class’;               #  模型关系类声明;
    const META_LINKS_TYPE = ‘type’;                 # 模型关系类型声明;
    const META_LINKS_ON = self::SYNTAX_JOIN_ON;     # 模型关系条件声明;
    const META_LINK_FOREIGN = ‘foreign’;            #  模型外键声明;
    const META_LINK_TABLE = ‘linkTable’;            # 模型关系关联表声明;
    # 模型关系表外键表声明;
    const META_LINK_TABLE_FOREIGN = ‘linkForeign’;

    # 规则配置声明
    const META_RULES = ‘rules’;          # 模型规则声明;
    const META_RULES_MESSAGE = ‘ruleMessages’;  # 模型验证规则声明;

    # 模型关系类别声明
    const HAS_ONE = ‘hasOne’;                   #  一对一关系;
```

```
    const HAS_MANY = ‘hasMany’;                    #  一对多关系;
    const BELONGS_TO = ‘belongsTo’;                #  多对一关系。
    const HAS_AND_BELONGS = ‘hasAndBelongs"; # 多对多关系

    # 模型配置方法，用于返回模型的配设信息
    static function __config();
}
```

IModelMetaSupport 接口用于声明模型的配置主键，包含的__config 表态方法用于返回模型的配置信息，后面将会对配置主键进行详细的讲解。

9.1.3、IDataSyntaxSupport 接口声明

```
interface IDataSyntaxSupport {
    # 表格语法
    const SYNTAX_TABLE = ‘table’;            # 数据表
    const SYNTAX_TABLE_ALIAS = ‘alias’;  # 数据表别名
    # 条件语法
    const SYNTAX_FIELDS = ‘fields’;          # 数据字段
    const SYNTAX_WHERE = ‘where’;          # 数据条件
    const SYNTAX_ORDER = ‘order’;      #  排序方式
    const SYNTAX_PAGE = ‘page’;        # 分页信息

    #  关联关系语法
    const SYNTAX_LINK = ‘links’;         # 关联关系
    const SYNTAX_JOIN = ‘joins’;         # 连接关系
    const SYNTAX_JOIN_ON = ‘on’;       # 连接条件
    const SYNTAX_JOIN_TYPE = ‘joinType’; # 连接方式

    # 数据查询方式
    const FETCH_STYLE_ASSOC = 2;         # 字段名称索引数组
    const FETCH_STYLE_NUM = 3;           # 字段数字索引数组
```

```
    const FETCH_STYLE_BOTH = 4;          # 字段名称/索引数组
    const FETCH_STYLE_OBJ = 5;           # 对象数组
    const FETCH_STYLE_COLUMN = 7;        # 列值
    const FETCH_STYLE_GROUP = 65536;     # 分组索引值
    const FETCH_STYLE_MODEL = 1000;      # 模型格式数据
}
```

IDataSyntaxSupport 接口并不是模型组件的一部分，而是内核服务层数据库组件的一部分，但因为两者的紧密联系因此在这里简单说明。

9.2、模型类定义

在 Skihat 中定义模型类非常简单，只需要声明并继承自 ApplicationModel 类（虽然了可以直接继承自 Model，但为了提供全局处理建议继承自 ApplicationModel 类）。

9.2.1、声明模型类

虽然模型类允许声明在任意目录中，但 Skihat 建议存放在 app/models 目录下，同时与有些 MVC 框架不需要声明模型不同，Skihat 中的模型类必须声明。

示例：

```
# app/models/guest_book.inc
class GuestBook extends ApplicationModel {
}
```

虽然可以在同一个文件中声明多个模型类，但建议保持一个模型文件包含一个模型类。

9.2.2、默认数据表名称

默认情况下，Skihat 采用模型名称的 camel 小写规范作为数据表的名称。

示例：

UcenterUser => ucenter_user

GuestBook => guest_book

User => user

9.2.3、自定义数据表名称

如果需要自定义表名称，可以重写模型的__config 静态方法，并设置 META_TABLE。

示例：

```
# app/models/my_guest_book.inc
class MyGuestBook extends ApplicationModel {
        public static function __config() {
            return array(
                    self::META_TABLE => 'guest_book'
            );
        }
}
```

__config 方法是模型的特殊方法，提供模型的自定义配置信息，后面我们将看到更多配置。

9.2.4、表别名

在 Skihat 模型中，除了数据表名字外还有表的别名信息，使用表别名的主要目的是为了解决数据库自引用关系，默认模型类的别名与类名相同。

示例：

```
# app/models/guest_book.inc
class GuestBook extends ApplicationModel {
    # 表的别名为 GuestBook
}
```

如果我们希望重新设定表的别名，可以使用__config 方法进行配置。

示例：

```
# 文件：app/models/my_guest_book.inc
class MyGuestBook extends ApplicationModel {
    public static function __config() {
        return array(self::META_TABLE_ALIAS => ‘GuestBook’);
    }
}
```

META_TABLE_ALIAS 主键值用于指定表的别名，除非需要自引用关系，否则 Skihat 不建议修改默认别名。

9.2.5、配置数据库

为了完成模型提供的服务，在模型内部使用内核数据库组件（kernels/databases）服务，默认情况下模型类将使用 default 数据引擎提供的服务。

示例：

```
# 文件：app/models/guest_book.inc
class GuestBook extends ApplicationModel {}
```

如果我们不使用默认数据库引擎，而是使用自定义的名称，可以使用 META_DATABASE 关键字指定数据库引擎。

示例：

```
# 文件：app/models/my_guest_book.inc
class MyGuestBooks extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_DATABASE => ‘guest_books’
        );
    }
}
```

使用的数据库必须是在 app/boots/config.inc 中提供有配置信息的，否则将引发 ioc 创建

错误。

示例：

```
# 文件：app/boots/config.inc
Skihat::write('kernels/databases',array(
        'default' => array(
            SKIHAT_IOC_CLASS => 'kernels.databases.engines.mysql_pdo_engine’,
'conf' => 'mysql:host=127.0.0.1;dbname=guestbooks&user=root&pass=killer&charset=utf8’
        ),
        'guest_books' => array(
            SKIHAT_IOC_CLASS => 'kernels.databases.engines.mysql_pdo_engine’,
'conf' => 'mysql:host=127.0.0.1;dbname=guestbooks&user=root&pass=killer&charset=utf8’ )
        )
);
```

数据库组件的更多配置信息，请查看数据库组件。

9.3、Model->__construct 构造模型实例

- 定义：public function __construct($props = array(),
  $forceUpdate = false,$forceLoad = false)

使用$props、$forceUpdate 和$forceLoad 创建 Model 实例。

9.3.1、使用$props 数组参数

如果$props 为数组类型参数，表示使用$props 初始化模型属性，数组键表示属性名称，数组值表示属值的值。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->isNew(); # 返回 true
```

9.3.2、使用$props 数值参数

如果$props 为数值类型，表示使用主键实始化实例，模型会自动根据主键值返回数据并填充当前模型。

示例：

```
$guestBook = new GuestBook(1);
$guestBook->isUpdate();          # 返回 true
```

注意：如果主键值在数据库中不存则，将引发 ModelNotFoundException 异常。

9.3.3、使用$forceUpdate 参数

默认情况下，使用$props 数组参数表示创建新的模型实例，但有时我们希望能够进行更新操作。

比如当需要编辑信息时，这时候就可以指定$forceUpdate 参数，强制模型更新状态。

示例：

```
$guestBook = new GustBook(array(‘id’ => 2,
                ’user’ => ‘guest’,’content’ => ‘留言内容’),true);
$guestBook->isUpdate();          # 返回：true
```

注意：指定$forceUpdate 参数为 true 时，必须在$props 中指定主键的值，否则将引发 ModelNotFoundException 异常。

9.3.4、使用$forceLoad 参数

如果在更新数据时，想要将数据库中原有的属性也读取出来，就可以使用$forceLoad 参数。

示例：

```
$guestBook = new GuestBook(array(‘id’ = 2,’user’ => ‘guest’,
                                ’content’ => ‘留言内容’),true,true);
```

```
$guestBook->isUpdate();          # true
$guestBook->created;             # 1928319283
```

读取数据后，新设置的属性值将会覆盖原有的值，没有设置的属性值保持不变。

注意：Skihat 建议更新模型时，除非所有模型属性由 POST 提交，否则建议指定 forceUpdate 参数为 true。

9.3.5、使用请求参数构造模型实例

通常我们不直接使用数组构造模型实例，而是使用请求参数来填充模型实例。

示例：

```
# 使用查询参数
$guestBook = new GuestBook($this->query(‘id’));

# 从表单中获取值
$guestBook = new GuestBook($this->form(‘guest_book’),true,true);

# 文件：app/views/guestbooks/write.stp
<input type="text" name="guest_book[user]" />
<input type="text" name="guest_book[content]" />
```

9.4、访问模型属性

9.4.1、Model->key 返回模型主键值

- 定义：public function key()

返回当前模型的主键值，如果主键值为空则返回 null。

虽然直接使用属性名称也能访问主键的值，但 key 方法会自动根据主键的名称获取主键的值，因此是主键访问最快捷的方式。

示例：

```
$guestBook = new GuestBook(array(‘id’ => 12,’user’ => ‘guest’));
$guestBook->key();              # 返回 12
```

默认主键

在 Skihat 中，默认主键名称为 id，并且采用自增类型。

自定义主键

在__config 方法中，使用 META_PRIMARY 和 META_PRIMARY_INCREMENT 指定主键的相关属性：

- META_PRIMARY：指定主键名称，默认为 id；
- META_PRIMARY_INCREMENT：主键是否为自增，默认为 true；

使用时，如果与默认值不符，只需要设置相应的键和值。

示例：

```
public static function __config() {
    return array(
        self::META_PRIMARY => ‘uid’,
        self::META_INCREMENT_PRIMARY => false
    );
}
```

注意：如果 META_INCREMENT_PRIMARY 为 false，则需要创建时自己指定主键的值。

多字段主键

Skihat 不支持多字段主键。

9.4.2、Model->__get && Model->__set：使用"->“访问单个属性值

为了便于处理模型属性，Model 类重写__set 和__get 魔术方法，允许使用"->"运算符直接访问属性值。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->name = ‘guest 1’;          # 设置属性值
$guestBook->newProp = ‘new prop’;     # 设置新的属性值

# 获取属性值
echo $guestBook->name;                 #  获取属性值
echo $guestBook->newProp;              # new prop
echo $guestBook->not_exists;           # null;
```

- **无效属性的值**

当读取或写入的属性无效时，模型类按以下原则进行处理：

- ✧ 获取属性时，如果属性值不存在则返回 null 值。
- ✧ 设置属性值时，模型实例会直接将值设置给模型，而不会判断是否为模型数据表中声明属性。

9.4.3、Model->ArrayAccess：使用 ArrayAccess 接口访问属性

除了使用魔术方法，在 Model 中还能通过使用 ArrayAccess 接口访问属性值：

- offsetExists：判断属性是否存在；
- offsetGet：获取属性的值；
- offsetSet：设置属性的值；
- offsetUnset：删除属性值；

通过接口可以看出，则使用魔术方法相比，使用 ArrayAccess 接口提供了更多的属性操作功能。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’));
$guestBook[‘user’] = ‘admin’;
$guestBook[‘newProp’] = ‘new prop’;

# 获取属性值
echo $guestBook[‘name’];               # admin
```

```
echo $guestBook[‘newProp’];              # new prop
echo $guestBook[‘not_exists’];           # null

# 删除属性
unset($guestBook[‘newProp’]);

# 判断属性是否存在
isset($guestBook[‘not_exists’]);      # false
```

无效属性

无效属性的处理原则同__get 和__set 方法相同。

9.4.4、Model->changeProps 更新多个属性值

- 定义：public function changeProps(array $props)

使用$props 批量更新属性值，主键表示属性名称，值表示属性值。

示例：

```
$guestBook = new GuestBook(‘user’ => ‘guest’,’content’=> ‘留言内容’));
$guestBook->changeProps(array(‘user’=>‘admin’,’content’ => ‘新的留言内容’));
```

9.4.5、访问原始属性值

为了减少对数据库的重复访问，在模型内部保存有当前和原始两组属性值。当前属性值会随属性的设置而变化，但原始属性值却不会发生变化与构建实例时保持一致：

- 使用$props 数组参数：与实始化$props 参数保存一致；
- 使用$props 数值参数：与数据库中的原始值保持一致；
- 使用 fetch 方法：原始属性与查找字段值保持一致；
- 使用 save 方法后：原始属性与当前属性保持一致；

使用原始时，是为了在需要获取初始化值时，不需要再进行数据库读取操作，提供运行效率。

9.4.5.1、Model->originalProp 返回单个原始属性值

- 定义：public function originalProp($name)

获取$prop 属性的原始值，如果$prop 指定属性不存将返回 null 值。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->user = ‘admin’;
$guestBook->originalProp(‘user’);        # 返回 guest
```

9.4.5.2、Model->originalProps 返回全部原始属性值

- 定义：public function originalProps()

返回全部属性的初始化值，类型为数组类型。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->changeProps(array(‘user’=> ‘admin’,’content’ => ‘新的留言内容’));

# 返回：array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->originalProps();
```

9.4.6、Model->currentProps 返回全部当前属性值

- 定义：public function currentProps()

返回全部属性的当前值，类型为数组类型。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->changeProps(array(‘user’=>‘admin’,’content’ => ‘新的留言内容’));

# 返回：array(‘user’ => ‘admin’,’content’ => ‘新的留言内容’));
$guestBook->currentProps();
```

9.4.7、Model->cancel 取消字段更新

- 定义：public function cancel()

返回到创建或上一次 save 时的状态，cancel 的核心其实就是使用原始值替换当前的值。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->changeProps(array(‘user’ => ‘admin’,’content’ =>‘新的留言内容’));
$guestBook->cancel();

# 返回值：array(‘user’ => ‘guest’,’content’ => ‘留言内容’))
$guestBook->currentProps();
```

9.4.8、Model->json 返回 json 格式数据

- 定义：public function json()

返回当前对象的 json 格式数据，注意返回的当前属性值。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));

# 返回值： {user:guest,content:留言内容}
$guestBook->json();
```

9.5、更新实例数据

9.5.1、Model->save 保存模型实例

- 定义：public function save(array $syntax = array())

根据$syntax 参数，将数据保存到数据库中，成功返回 true，失败返回 false。

9.5.1.1、isNew 新增数据

如果模型的状态为 isNew，那么执行 save 方法时，将会执行数据库增加操作。

示例：

```
$guestBook = new GuestBook(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));
$guestBook->save();
$guestBook->key();              # 获取 key 值
$guestBook->isUpdate();         # true
```

如果模型主键为自增类型，则创建完成后可以使用 key 方法返回新增加的主键值，保存成功后模型的状态会转换为 isUpdate。

9.5.1.2、isUpdate 更新数据

如果模型的状态为 isUpdate，那么执行 save 方法时，会将当前的数据保存到数据库中。

示例：

```
$guestBook = new GuestBook(array(‘id’ => 12,’user’=> ‘admins’,
                                 ’content’ => ‘新的内容’));
$guestBook->save();
```

使用$syntax 参数

使用$syntax 参数是为了能够在更新时，提供更多的选项，允许使用以下语法声明：

- IDataSupport::SYNTAX_FIELDS：使用数组参数指定更新字段；
- IDataSupport::SYNTAX_WHERE：附加额外的条件限制，并且仅仅允许使用字段条件；

示例：

```
$guestBook = new GuestBook($this->form(‘guest_book’),true);

# 保存信息，仅仅更新 user 和 content 字段的值，并且创建时间必须为 1 小时内的。
$guestBook->save(array(
    Model::SYNTAX_FIELDS => array(‘user’,’content’)
    Model::SYNTAX_WHERE => array(‘GuestBook.created > ‘ => time() - 3600)
));
```

注意：这里没有使用 IDataSupport 接口常量，而是使用 Model 类常量，建议指定时使用 Model 常量，减少对数据库组件的依赖。

9.5.2、Model->delete 删除模型实例

- 定义：public function delete(array $syntax = array())

根据$syntax 参数从数据库中删除当前模型，删除成功返回 true，失败返回 false。

删除方法只允许删除状态为 isUpdate 的模型数据，其它状态将引发 ModelException 异常。

示例：

```
$guestBook = GuestBook::find(12);
$guestBook->delete();
$guestBook->isDelete();        # true
```

完成删除后，模型的 isDelete 状态为 true，表示当前模型已删除。

使用$syntax 参数

在 delete 方法中允许指定以下语法参数：

- IDataSupport::SYNTAX_WHERE：附加额外的条件限制，但仅允许使用字段条件；
- IDataSupport::SYNTAX_LINK：附加额外的关联关系，更多信息请参考模型关联；

示例：

```
$guestBook = new GuestBook(12);

# 如果留言的创建时间是一个小时内的删除成功，否则删除失败。
guestBook->delete(array(
    Model::SYNTAX_WHERE => array('GuestBook.created >'  => time() - 3600)
));
```

注意：注意删除方法基于主键的值，如果主键值为空，删除将失败。

9.6、查询数据

9.6.1、Model::fetch 查询单个实例

- 定义：public static function fetch($condition,$syntax = array(),
  $fetchStyle = self::FETCH_STYLE_MODEL)

使用$conditions 和$syntax 查询单个模型记录，如果有多个记录将什么满足条件则返回第一条记录。

示例：

```
$guestBook = GuestBook::fetch(12);              # 主键查询条件
```

使用$conditions 参数

请查 9.10、查询条件

使用$syntax 参数

在 fetch 方法中，支持以下语法参数：

- IDataSupport::SYNTAX_FIELDS：指定查询的字段值，允许使用字符和数组参数；
- IDataSupport::SYNTAX_ORDER：指定排序字段，允许使用字符串和数组参数；
- IDataSupport::SYNTAX_LINK：指定查询的关联模型，更多信息请查看模型关系；

示例：

```
$guestBook = GuestBook::fetch(12,array(
    Model::SYNTAX_FIELDS => ‘GuestBook.user,GuestBook.content’));

$guestBook = GuestBook::fetch(12,array(
    Model::SYNTAX_FIELDS => array(‘GuestBook.user’,’GuestBook.content’));

# 使用排序
$guestBook = GuestBook::fetch(array(GuestBook.user’ => ‘guest’),array(
                        Model::SYNTAX_ORDER => ‘GuestBook.id desc’));
```

```
$guestBook = GuestBook::fetch(array(‘GuestBook.user’ => ‘guest’),array(
    Model::SYNTAX_ORDER => array(‘GuestBook.id’ => ‘desc’)
));
```

注意：数组 order 参数，索引表示字段名称，值表示排序方式。

使用$fetchStyle 参数

$fetchStyle 参数表示返回参数的数据格式，支持以下格式：

- FETCH_STYLE_ASSOC：列名数组格式，列为 key。
- FETCH_STYLE_NUM：数值索引数组，没有列名；
- FETCH_STYLE_BOTH：列表+数值索引两者；
- FETCH_STYLE_OBJ：对象格式，列名表示属性名；
- FETCH_STYLE_COLUMN ：单列数值；
- FETCH_STYLE_GROUP：分组格式数组；
- FETCH_STYLE_MODEL：格式格式数据；

默认$fetchStyle 参数返回模型值，如果需要返回不同的格式，可以指定值：

示例：

```
$guestBook = GuestBook::fetch(1,Model::FETCH_STYLE_OBJ);
```

9.6.2、Model::fetchAll 查询多条记录

- 定义：public static function fetchAll($conditions,$syntax)

使用$conditions 和$syntax 查询多条记录，并返回模型查询对象。

示例：

```
$guestBooks = GuestBook::fetchAll();        #   返回全部数据
```

完成查询后$guestBooks 并不是一个普通的数组，而是一个 ModelFetch 类的实例，更多信息请查看 ModelFetch 类。

使用$conditions 条件参数

请参考 9.10、查询条件。

9.6.3、ModelFetch 类

ModelFetch 类用于查询多条记录，ModelFetch 实例允许我们执行更多的选项和操作，并且返回不同的数据类型。

9.6.3.1、ModelFetch 类声明

```
class ModelFetch {
    # 实始化实例
    public function __construct($modelName,array $syntax);

    #   设置查询语法
    public function fields($fields);            # 设置查询的字段
    public function links($joins);              # 设置关联数据
    public function where($where);              # 设置查询条件
    public function order($order);              # 设置排序方式
    public function page($page,$size = self::DEFAULT_PAGE_SIZE);   # 设置分页

    # 获取数据
    public function fetchModels();      # 返回模型数组
    public function fetchRows();        # 返回二维数组
    public function fetchObjects();     # 返回对象数组
    public function fetchJson();        # 返回 json 格式数据
    public function fetchCount();       # 返回记录数量

    # 附加属性值
    public function addExtra($name,$val);       #   附加属性
    public function removeExtra($name);         # 移除附加属性
    public function extra($name);               # 获取附加属性值
}
```

9.6.3.2、ModelFetch->__construct 实例化 ModelFetch 类

通常不需要直接调用__construct 方法实始化 ModelFetch 实例，而是调用 Model::fetchAll

方法返回 ModelFetch 类的实例。

示例：

```
$guestBooks = GuestBook::fetchAll();
```

9.6.3.3、设置查询语法

ModelFetch->where 设置条件语法

当调用 Model::fetchAll 方法后，$conditions 参数会自动设置给 ModelFetch 类实例，如果需要重新设置可以调用 where 方法。

- 定义：public function where($where)

根据$where 设置或移除检索条件，如果$where 为空值表示移除查询条件。

示例：

```
$guestBooks = GuestBook::fetchAll(array(‘GuestBook.user’ => ‘admin’));
$guestBooks->where(false);              # 移除条件
$guestBooks->where(array(‘GuestBook.user’ => ‘guest’));        # 重新设置条件
```

关于条件的更多信息，请查看”9.10、查询条件"。

ModelFetch->fields 设置查询字段

- 定义：public function fields($fields)

根据$fields 设置或移除检索列表的值，如果$fields 参数为空，表示移除字段设置。

示例：

```
$guestBooks = GuestBook::fetchAll();

# 使用字符串参数
$guestBooks->fields(‘GuestBook.user,GuestBook.id,GuestBook.created’);

# 使用数组参数
$guestBooks->fields(array(‘GuestBook.user’,’GuestBook.id’,’GuestBook.created’)));

$guestBooks->fields(false);        # 移除字段设置
```

ModelFetch->links 设置关联模型

- 定义：public function links($links);

根据$joins 设置或移除关联对象，$links 为空表示移除法，更多信息请查看模型关联。

ModelFetch->order 设置排序方式

- 定义：public function order($order)

根据$order 设置或移除排序方式，如果$order 为空，表示移除排序。

示例：

```
$guestBooks = GuestBooks::fetchAll();
$guestBooks->order(‘GuestBook.created desc’);                    # 使用字符串排序
$guestBooks->order(array(‘GuestBook.created’ => ‘desc’)); # 使用数组参数排序
$guestBooks->order(false);                                                  # 移除排序
```

ModelFetch->page 设置分页

- 定义：public function page($page,$size = self::DEFAULT_PAGE_SIZE)

根据$page 和$size 参数设置分页信息，设置完成后将返回指定范围内的数据。

示例：

```
$guestBooks = GuestBook::fetchAll();
$guestBooks->page(2,10);              # 设置第二页、每页 10 条记录；
$guestBooks->page(false);             # 多除分页
```

9.6.3.4、ModelFetch->fetchXXX 获取数据

ModelFetch 类设置好查询条件后，并不会执行数据库访问，而是需要使用 fetchXXX 执行数据获取后，才会进行数据库的访问。

ModelFetch->fetchModels 返回模型数据

- 定义：public function fetchModels()

根据当前查询命令，返回模型的二维数组集合。

示例：

```
$guestBooks = GuestBook::fetchAll();

#返回值：array(GuestBook(xxx),GuestBook(xxx));
$books = $guestBook->fetchModels();
```

ModelFetch->fetchObjects 返回对象数据

- 定义：public function fetchObjects()

根据当前查询命令，返回对象的二维数组集合。

示例：

```
$guestBooks = GuestBook::fetchAll();

# 返回值：array(StdClass(xxxx),StdClass(xxx));
$books = $guestBook->fetchObjects();
```

返回对象数据类型时，将根据字段的名称生成对象属性。

ModelFetch->fetchRows 返回数据二维数组

- 定义：public function fetchRows()

根据当前查询命令，返回数据的二维数组集合。

示例：

```
$guestBooks = GuestBook::fetchAll();

# 返回值：array(array(’name’ => xxx),array(’name’ => xx));
$books = $guestBook->fetchRows();
```

返回数组值时，字段名称为键值，值为数组值。

ModelFetch->fetchJson 返回数据的 Json 格式

- 定义：public function fetchJson()

根据当前查询命令，返回数据 Json 格式。

示例：

```
$guestBooks = GuestBook::fetchAll();
$books = $guestBook->fetchJson();          # 返回值：[{name:xxx},{name:xxx}];
```

ModelFetch->fetchCount 返回数据的数量

- 定义：public function fetchCount()

根据当前查询条件，返回查询的总数。

示例：

```
$guestBooks = GuestBook:fetchAll();
$guestBooks->fetchCount();              # 返回值：可能是 12 或其它。
```

9.6.3.5、附加属性方法

ModelFetch->addExtra 附加属性

- 定义：public function addExtra($name,$val)

根据$name 和$val 值附加新的属性值。

ModelFetch->removeExtra 移除附加属性

- 定义：public function removeExtra($name)

根据$name 移除附加属性。

ModelFetch->extra 获取附加属性

- 定义：public function extra($name)

根据$name 获取附加属性值，如果指定的属性没有添加，则返回 null。

使用附加方法

ModelFetch 附加方法，是一个非常有用的功能允许动态的为 ModelFetch 实例增加新的属性值。

在视图分页过滤器就中就是使用附加方法实现的分页控制：

- 在控制器中：使用 PaginateFilter 设置分页信息，并附加给 ModelFetch 类实例；
- 视图模型中：使用分页助手获取分页信息，并生成 HTML 代码；

更多信息查看 PaginateFilter 分页过滤器和分页助手。

9.6.3.6、在控制器中使用 ModelFetch 类

在 Skihat，通过控制器向视图模板传递模型集合时，通常不直接传递内容，而是传递 ModelFetch 类的实例。

例如：

```
# 文件：app/controllers/guestbooks_controller.inc
 public function indexAction() {
      $this[‘guest_books’] = GuestBook::fetchAll();
 }
```

采用这种方式有两个优点： 一是允许过滤器对 ModelFetch 实例进行二次处理； 二是允许在视图模板中，根据需要加载数据，实现按需加载。

9.7.4、其它查询方法

9.7.4.1、查询单个行的值

- 定义：public static function fetchField($field,$conditions,array $syntax = array())

根据$field、$conditions 和$syntax 查询单个字段的值。使用 fetchField 查询时无论有多少个记录，都将会返回第一行的第一列的值。

示例：

```
$count = User::fetch(‘count(*)’);          #  获取记录总数；
$sum = Order::fetch(‘sum(amount)’);  # 获取合计总数；
```

9.7.4.2、查询是否唯一字段值

- 定义：public function fetchUnique($field)

根据$field 字段，返回当前字段是否为数据库中的唯一值。

9.7.4.3、唯一值验证证方法

- 定义：function unique_validator(Model $model,$field)

虽然 unique_validator 方法不是 Model 的内部方法，用于判断模型的指定属性是否为数据库中的唯一值。例如：判断是用户名或邮件是否唯一。

使用方法请查看模型验证。

9.7、批量操作

9.7.1、Model::createAll 创建模型

- 定义：public static function createAll(array $fields, array $syntax = array())

根据$fields 和$syntax 创建模型记录值，并返回创建的数量。同实例化保存数据相比使用 createAll 可以不用构建实例，就能完成数据的创建任务。

示例：

GuestBook::createAll(array(‘user’ => ‘guest’,’content’ => ‘留言内容’));

创建数据时，key 表示数据字段名称，value 更新插入的值。

使用$syntax 参数

在 createAll 方法中允许以下语法参数：

- IDataSupport::SYNTAX_FIELDS：使用数组参数指定创建的字段值。

示例：

```
GuestBook::createAll($this->form(‘guest_books’),array(
    Model::SYNTAX_FIELDS => array(‘user’,’content’)
));
```

9.7.2、Model::createAllMultiple 创建多个模型

- 定义：public static function createAllMultiple(array $fields,array $values,
  array $syntax = array());

根据$fields、$values 和$syntax 创建多个模型记录，并返回创建的数量。

示例：

```
GuestBook::createAll(array(‘user’,’content’),array(
    array('guest’,’留言内容’),
    array(‘admin’,’留言内容’)
));
```

注意：$values 必须为二维数组。

9.7.3、Model::updateAll 更新多个模型

- 定义：public static function updateAll($fields,$conditions,array $syntax = array());

根据$fields、$conditions 和$syntax 更新多条记录，并返回更新的数量。

示例：

```
# 更新全部的用户名为 admins。
GuestBook::updateAll(array(‘GuestBook.name’ => ‘admins’));
```

9.7.3.1、使用$fields 参数

使用$fields 字符串（SQL 命令字段）

如果$fields 的类型为字符中，Skihat 将会认为这里是 SQL 命令字段，不进行任何操作直接生成为 UPDATE 的 SET 命令。

示例：

```
# UPDATE SET GuestBook.name = ‘admins’ WHERE GuestBook.id = 12;
GuestBook::updateAll(‘GuestBook.name = \’admins\’,12);
```

注意：使用 SQL 字段命令时，需要特别注意安全问题，防止拼接命令产生 SQL 注入安全。

使用$fields 字符串参数（SQL 命令参数字段）

经常在更新时都需要指定外部变量的值，这时可以使用字符串参数。使用方法是指定数组参数，第一个索引指定 SQL 命令，后面的参数表示参数的值。

示例：

UPDATE SET GuestBook.name=? WEHRE GuestBook.id = 12

GuestBook::updateAll(array(‘GuestBook.name=?’,’admins’),12);

通过示例可以看出，参数使用”?”号占位符，后面的参数与占位符之间一一对应。

使用$fields 字段参数

使用包含指定字段的$fields 数组，表示使用$fields 字段更新。

示例：

UPDATE SET GuestBook.name = ? AND GuestBook.content = ? WHERE GuestBook.id = ?

GuestBook::updateAll(array(‘GuestBook.name’ => ‘admin’,’GuestBook.content’ => ‘新内容),12);

9.7.3.2、使用$conditions 参数

请查看条件查询参数。

9.7.3.3、使用$syntax 语法参数

在 updateAll 中允许指定以下语法参数：

- IDataSupport::SYNTAX_FIELDS：使用字段参数时，指定需要更新字段。
- IDataSupport::SYNTAX_LINK：指定关联关系，更多请参数模型关联；

示例：

```
# 仅仅更新 GuestBook.name 的参数值。
GuestBook::updateAll(array(‘GuestBook.name’ => ‘admin’,
                           ’GuestBook.content’ => ‘新留言内容'),array(
    Model::SYNTAX_FIELDS => array(‘GuestBook.name’)
));
```

9.7.4、Model::deleteAll 删除多个模型记录

- 定义：public static function deleteAll($conditions,$syntax = array());

根据$conditions 和$syntax 删除多条记录，并返回影响的行。

示例：

```
GuestBook::deleteAll(array(‘GuestBook.user’ => ‘guest’));
```

9.7.4.1、使用$conditions 参数

请查看条件查询参数。

9.7.4.2、使用$syntax 参数

在 deleteAll 中允许使以下语法参数：

- IDataSupport::SYNTAX_LINK：指定关联关系；

更多内容请查看模型关联。

9.8、模型验证

9.8.1、配置模型验证

为了保证数据的有效性，在 Skihat 模型中允许使用验证功能，使用验证功能，使用以下两个配置键：

- META_RULE：使用数组参数，指定验证规则；
- META_RULE_MESSAGES：使用数组参数，指定验证消息提示；

示例：

```
class GuestBook extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_RULE => array(
                # 创建规则
                self::CREATE_RULES => array(
                    ‘字段’ => ‘验证规则’,          # 指定单个验证规则
                    ‘字端’ => array(‘验证规则’ => ‘规则配置值’)
                ),

                # 更新规则
                self::UPDATE_RULES => array(
                )
            ),

            self::META_RULE_MESSAGE => array (
                ‘字段’ => ‘错误消息’,    # 指定验证错误消息
                ‘字段’ => array(‘验证规则’ => ‘错误消息’)
            )
        );
    }
}
```

指定验证规则时，需要首先指定验证规则名称，在 Model 中内置两个规则名称：

- CREATE_RULES：创建规则，当创建的模型保存时使用本规则；
- UPDATE_RULES：更新规则，当更新的模型保存时使用本规则；

如果还有其它的业务需要实现，也可以声明新的验证规则。

9.8.2、配置字段验证规则

9.8.2.1、验证规则命名规范

验证规则是由验证方法提供的，验证方法的名称规则为"验证名称_validator"，允许以下两种方法签名：

- 定义：function xxxx_validator(&$input,$field,$options)　# 包含选项参数
- 定义：function xxxx_validator(&$input,$field);　# 不依赖于选项参数

9.8.2.2、Skihat 内置验证方法

在 Skihat 中，大部分的验证方法都是在 skihat/kernels/validators.inc 文件中声明，包含以下验证方法：

- function required_validator(&$input, $field)　# 非空值验证
- function numeric_validator(&$input, $field)　# 数值类型验证
- function telephone_validator(&$input, $field)　# 电话号码验证，支持坐机、手机、400 电话
- function email_validator(&$input, $field)　# 邮件验证
- function url_validator(&$input, $field)　# URL 验证
- function ip_validator(&$input, $field)　# IP 地址验证
- function strmin_validator(&$input, $field, $length = 10)　# 最小字串验证
- function strmax_validator(&$input, $field, $length = 32)　# 最大字符串
- function strlen_validator(&$input, $field, $range = array())　# 字符长度范围验证
- function nummax_validator(&$input, $field, $max = 10000)　# 最大数值验证
- function nummin_validator(&$input, $field, $min = 10)　# 最小数值验证
- function numrang_validator(&$input, $field, $range = array())　# 数值范围验证
- function compare_validator(&$input, $field, $compare = ‘’)　# 字段比较验证
- function regex_validator(&$input, $field, $pattern = '*’)　# 正则规则验证
- function enum_validator(&$input, $field, $options = array())　# 枚举验证

更多信息，请查看验证组件。

9.8.2.3、为字段配置单个验证规则

为字段配置单个验证规则，只需要直接为字段值中指定验证的规则的名称。

示例：

self::CREATE_RULES => array(‘name’ => ‘required’);

注意：指定验证规则时不需要指定_validator 方法后缀。

9.8.2.4、为字段配置多个验证规则

为字段配置多个验证规则，需要使用数组格式，数据的索引表示验证规则名称，值表示验证参数。

示例：

为$name 字段指定两个验证规则 required 和 strmin，其中 strmin 的最小值为 10。

self::CREATE_RULES => array(‘name’ => array(‘required’ => true,’strmin’ => 10));

注意：指定多个验证规则时，先指定的规则会优先执行。

9.8.3、配置字段验证消息

当验证规则验证失败时，模型将会为自己指定错误消息，错误消息默认为 Field error，我们也可以使用 META_RULE_MESSAGE 手动指定错误消息。

9.8.3.1、为字段配置统一错误提示

为字段指定统一错误提示，只需要要将字段值的值设置为错误提示的内容，当验证失败时，任何验证都显示一致的错误提示消息。

示例：

self::META_RULE_MESSAGE => array(’name’ => ‘用户名不能为空，并且长度必

须大于 10’));

9.8.3.2、为每一种错误分别指定错误提示

如果需要分别为每一种错误分别显示错误，则需要使用索引指定验证器名称，值表示表示验证消息。

示例：

```
self::META_RULE_MESSAGE => array(‘name’ => array(
    ‘required’ => ‘用户名不能为空’,
    ’strmin’ => ‘用户名长度必大于 10’
))
```

9.8.4、使用验证功能

使用验证功能非常简单，当我们执行模型的 save 方法后，就会自动根据当前状态，调用 CREATE_RULES 或 UPDATE_RULES 规则自动进行验证，如果验证失败，save 方法将会返回 false。

示例：

```
class GuestBook extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_RULE => array(
                self::CREATE_RULE => array(‘user’ => ‘required’),
                self::UPDATE_RULE => array(‘user’ => ‘required’)),
            self::META_RULE_MESSAGE => array(
                ‘name’ => ‘用户信息不能为空'
        )
    }
}

$guestBook = new GuestBook();
$guestBook->save();    # 返回 false，执行 CREATE_RULES 验证。
```

9.8.5、错误消息

当模型执行验证后，验证的错误消息会自动设置到模型错误消息中，我们可以通过一组方法来操作模型的错误消息：

- error：返回单个字段的错误消息，如果没有则返回 false。
- allErrors()：返回模型的全部错误消息。
- addError：附加新的模型错误消息；
- clearErrors：清除模型的错误消息；

当执行模型保存后，如果保存失败，可以使用以下方式获取错误消息：

示例：

```
$guestBook->save();
$guestBook->error(‘name’);   # 返回：用户信息不能为空
$guestBook->allErrors();          # 返回：array (‘name’ => ‘用户信息不能为空’)
```

如果需重新清除或重新初始化错误消息。

示例：

```
# 重新设置错误消息
$guestBook->clearErrors(array(’name’ => ‘用户名不能为空’));

# 清除全部错误消息
$guestBook->clearErrors();
```

9.8.6、验证回调方法

虽然 Skihat 提供的验证方法已经很完善，但也不能保证满足所有的业务规则，这时候可以使用回调方法增加自定义代码。

- 定义：protected function _validateBefore($onRule,&$syntax)

验证前回调方法，如果返回值为 false 表示验证失败。

示例：

```
protected function _validateBefore($onRule,&$syntax) {
```

```
        if ($this[‘user’]  !=  ‘admin’) {
            $this->error(‘user’,’用户名不为管理员，不允许更新’);
            return false;
        }
    }

    $guestBook = new GuestBook(array(‘user’  =>  ‘guest’));
    $guestBook->save();          # false
    $guestBook->allErrors();     array(‘user’,’用户名不为管理员，不允许更新’);
```

9.8.7、自定义验证方法

自定义验证方法非常简单，只需要满足验证方法的命名规则，并且引用到当前环境中。

示例：

```
function admin_validator(&$input,$field) {
    return $input[$field] ==  ‘admins’;
}

# 配置
self::CREATE_RULES => array(‘user’  =>  ‘admin’)
```

9.9、使用模型关联

在 Skihat 模型中，允许为模型指定关联关系，允许一对一、一对多、多对多、多对一四种关系。

9.9.1、配置模型关系

模型关系，使用 META_LINKS 关键字进行配置：

```
class User extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_LINKS => array(
                ‘关系名称’  => array(
```

```
                    self::META_LINKS_CLASS => ‘xxx’,
                    self::META_LINKS_TYPE => ‘xxx’,
                    self::META_LINKS_XXX = ‘xxxx’,
                )
            ));
        }
    }
```

建议关键名称使用 camel 大写规范全名关系，并且不要与已有的字段名称发生冲突。

在关系配置中，允许包含以下配置信息：

- META_LINKS_CLASS：指定关联模型的类信息，使用 Ioc 字符串配设置（必须值）；
- META_LINKS_TYPE：指定模型的关系的类型（默认关系为 HAS_ONE）；
- META_LINKS_ON：关系表的连接条件 XXX JOIN ON META_LINK_ON，非必须值；
- META_LINKS_FOREIGN：关系外键表的字段名称，非必须值；
- META_LINKS_TABLE：多对多中间表名称，仅仅用于多对多关系；
- META_LINKS_TABLE_FOREIGN：多对多中间表键名称，仅仅用于多对多关系；

模型关系允许使用以下类型：

- Model::HAS_ONE：一对一关系；
- Model::HAS_MANY：一对多关系；
- Model::BELONGS_TO：多对一关系；
- Model::HAS_AND_BELONGS 多对多关系；

后面会对各种关系进行详细的说明。

9.9.2、HasOne（一对一）关系

HasOne 表示了这样一种关系，即一个模型实例拥有另一个模型的单个实例（在数据库中表现为一对一关系）。

9.9.2.1、配置 HasOne 关系

在 HasOne 关系中，必须指定 META_LINKS_CLASS，同时还允许使用 META_LINKS_ON 和 META_LINKS_FOREIGN 两个配设置。

示例：

```
class Article extends ApplicationModel {
    public static function __config() {
        return array(‘content’ => array(
            self::META_LINKS = array (
                ‘content’ => array (
                    self::META_LINKS_CLASS => ‘models.article_content’,
                    self::META_LINKS_TYPE  = self::HAS_ONE, # 指定模型关系。
                    self::META_LINKS_FOREIGN = ‘article_id’
                )
            )
        ));
    }
}

class ArticleContent extends ApplicationModel {}
```

数据表的结构

```
CREATE TABLE article (
    `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
);

CREATE TALBE article_content(
    `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `article_id` INT NOT NULL
);
```

通过示例可以看出，META_LINKS_FOREIGN 指定的是子表的外键值，执行查询时将生成以下内容：

FORM article as Article LEFT JOIN article_content as ArticleContent ON Article.id = ArticleContent.article_id

默认外键名称

如果没有指定 META_LINKS_FOREIGN，那么 Skihat 默认采用"模型表名称+_id"作为外键值。

示例：

```
Article(article） => article_id;                # 默认表名称
Article(article_base) => article_base_id;       # 指定表名称
```

使用 META_LINKS_ON 配置

通常 Skihat 会自动生成关联的 ON 命令，但如果我们需要手动指定，使用 META_LINKS_ON 配置。

示例：

```
public static function __config() {
    return array(‘content’ => array(
        self::META_LINKS = array (
            ‘content’ => array (
                self::META_LINKS_CLASS => ‘models.article_content’,
                self::META_LINKS_TYPE  = self::HAS_ONE,
                self::META_LINKS_ON =  ‘Article.id = ArticleContent.article_id'
            )
        )
    ));
}
```

当 META_LINKS_ON 和 META_LINKS_FOREIGN 同时指定时，Skihat 优先使用 META_LINKS_ON 配置值。

9.9.2.2、使用 HasOne 关系成员变量

在 Skihat 中，使用关系成员变量非常简单，直接使用"->"或 ArrayAccess 的方式。

示例：

```
$article = Article::find(12);
$article->content;              # 使用 HasOne 的 content 关系，返回 ArticleContent 实例；
```

$article[‘content’];　　# 使用 HasOne 的 content 关系，返回 ArticleContent 实例；

9.9.2.3、加载 HasOne 成员数据

为了加载 HasOne 的成员数据，Skihat 中允许使用同步和异常两种方式完成加载。

同步加载

同步加载是指，在读取模型数据时同步加载 HasOne 关系的数据，减少数据读取操作，实现同步加载需要使用 SYNTAX_LINK 语法命令。

示例：

```
# 同步加载
$article = Article::find(1,array(Model::SYNTAX_LINK => ‘content’));
# 同步加载
$articles = Article::findAll(array(),array(Model::SYNTAX_LINK => ‘content’));

# 使用 ModelFetch::links 方法指定
$fetch = Article::findAll();
$fetch->links(‘content’);
```

异步加载

如果没有使用 SYNTAX_LINK 语法同步加载，默认采用异步加载，异步加载只有获取关系成员时才查询数据。

示例：

```
$article = Article::find(1);
$article->content;　　# 异步加载
```

9.9.2.4、正确使用 HasOne 和 BelongsTo 关系

HasOne 和 BelongsTo 获取数据非常类似，都是返回关系模型的实例，但两者在数据库中有非常大的区别。

- Article -> ArticleContent（父子关系 -> HasOne）
- ArticleContent->Article　(子父关系 -> BelongsTo）

虽然是推荐的模型关系，但有时也可能希望将"子父关系"表现为 HasOne 关系，例如：部门与部门经理的问题。

示例：

```
CREATE TABLE employee (
    `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY
);

CREATE TABLE department (
    `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `manager`    INT NOT NULL
);
```

要解决这种问题，只需要手动指定 META_LINKS_ON 修改默认声明：

示例：

```
class Department extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_LINKS => array(
                ‘managerEmployee’ => array (
                    self::META_LINKS_CLASS => ‘models.employee’,
                    self::META_LINKS_TYPE => self::HAS_ONE,
                    # 指定新的 ON，覆盖默认值 。
                    self::META_LINKS_ON => ‘Employee.id = Department.id’
                )
            )
        );
    }
}

$department = new Department(1);
$department->managerEmployee;      # 获取经理的信息
```

9.9.3、HasMany（一对多）关系

一对多关系表示一个模型的实例包含另一个模型的多个实例（数据库父子关系）。

9.9.3.1、配置 HasMany 关系

配置 HasMany 与配置 HasOne 的方式相似，只需要将 META_LINKS_TYPE 指定为 HAS_MANY。

示例：

```
class Group extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_LINKS => array (
                ‘users’ => array(
                    self::META_LINKS_CLASS => ‘models.user’,
                    self::META_LINKS_TYPE => self::HAS_MANY,
                    self::META_LINKS_FOREIGN => ‘group_id’
                )
            )
        );
    }
}
```

9.9.3.2、使用 HasMany 关系成员

使用 HasMany 关系同样使用"->"或 ArrayAccess。

示例：

```
$group = new Group(12);
$group->users;      # 返回 ModelFetch
```

注意：HasMany 返回的成员变量为 ModelFetch 类型。

9.9.3.3、加载 HasMany 成员数据

加载 HasMany 成员数据只允许使用异步加载，不能使用同步加载。

9.9.4、BelongsTo（多对一）关系

BelongsTo 表示一种属于关系，例如：用户属于一个管理组（数据表子父关系）。

9.9.4.1、配置 BelongsTo 关系

配置 BelongsTo 大致方法与 HasOne 类似，但 META_LINKS_FOREIGN 有本质的区别。

示例：

```
class User extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_LINKS => array(
                ‘group’  => array(
                    self::META_LINKS_CLASS =>  ‘models.group’,
                    self::META_LINKS_TYPE => self::BELONGS_TO,
                    # 注意 group_id 是 user 表的字段；
                    self::META_LINKS_FOREIGN =>  ‘group_id’
                )
            )
        );
    }
}

CREATE TABLE user (
    `id`  INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `group_id` INT NULL
);

CREATE TABLE group (
    `id`  INT NOT NULL AUTO_INCREMENT PRIMARY KEY
);
```

9.9.4.2、使用 BelongsTo 关系成员

使用 BelongsTo 成员同样使用"->"或 ArrayAccess。

示例：

```
$user = new User(12);
$user->group;        # 返回 Group 实例。
```

9.9.4.3、加载 BelongsTo 数据成员

加载 BelongsTo 允许使用同步和异常两种模式进行加载，具体方法请参考加载 HasOne 数据。

9.9.5、使用 HasAndBelongs（多对多）关系

多对多关系是最复杂的一个关系，表示一个模型实例包含另一个模型的多个实例，反过来也是。

虽然与 HasMany 一样都是有另一个模型的多个实例，但 HasAndBelongs 需要有中间表来存储这种多对多的关系。

9.9.5.1、配置 HasAndBelongs 关系

除了前面使用的配置信息外，还有 META_LINKS_TABLE 和 META_LINKS_TABLE_FOREIGN 两个特别的配置。

示例：

```
CREATE TABLE user ( `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE role (`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE user_role (`id`,`role_id`,`user_id`);                    # 中间表

class User extends ApplicationModel {
    public static function __config() {
```

```
            return array(
                    self::META_LINKS => array(
                        ‘roles’  => array(
                            self::META_LINKS_CLASS =>  ‘models.role’,
                            self::META_LINKS_TYPE => self::HAS_AND_BELONGS,
                            self::META_LINKS_TABLE =>  ‘user_role’,
                            self::META_LINKS_FOREIGN =>  ‘user_id’,
                            self::META_LINKS_TABLE_FOREIGN =>  ‘role_id’
                    )
                )
            );
        }
    }

    class Role extends ApplicationModel {
        public static function __config() {
            return array(
                    self::META_LINKS => array(
                        ‘users’  => array(
                            self::META_LINKS_CLASS =>  ‘models.user’,
                            self::META_LINKS_TYPE => self::HAS_AND_BELONGS’,
                            self::META_LINKS_TABLE =>  ‘user_role’,
                            self::META_LINKS_FOREIGN =>  ‘role_id’,
                            self::META_LINKS_FOREIGN =>  ‘user_id'
                        )
                )
             );
        }
    }
```

META_LINKS_TABLE 配置

META_LINKS_TABLE：用于指定中间表的名称，默认值为根据模型表的名称排序生使用"_"号联接。

- User(user)、Role(role) => role_user

META_LINKS_TABLE_FOREIGN 配置

META_LINKS_TABLE_FOREIGN 用于指定关联表在中间表的中的外键字段名称，默认使用关联类的表名称+"_id"。

- User(user) 、Role(role) => User(role_id),Role(user_id)

注意：两者的配置值的值是关联模型的外键值。

9.9.5.2、使用 HasAndBelongs 关系成员

使用 HasAndBelongs 成员，同样使用"->"或 ArrayAccess。

示例：

```
$user = new User(1);
$user->roles;              # 返回 HasAndBelongsModelFetch 实例
```

- **HasAndBelongsModelFetch** 类声明

```
class HasAndBelongsModelFetch extends ModelFetch {
    public function add($models);          # 增加关联模型、使用关联模型主键数组
    public function remove($models);       # 删除关联模型、使用关联模型主键数组
    public function replace($models);      # 重新指定关联模型、使用模型主键或数组
}
```

HasAndBelongsModelFetch 直接继承自 ModelFetch，因此可以直接使用 ModelFetch 的各种方法来设置语法参数或获取值。

- **增加关联模型**

增加关联模型非常简单，只需要调用 add 方法。

示例：

```
$user = new User(12);
$user->roles->add(15);              # 15 为角色编号
$user->roles->add(array(13,14));    # 13,14 为角色编号
```

■ **移除关联模型**

移除关联模型非常简单，只需要调用 remove 方法。

示例：

```
$user = new User(12);
$user->roles->remove(15);                 # 15 为角色编号
$user->roles->remove(array(13,14));    # 13,14 为角色编号
```

■ **替换关联模型**

替换关联模型是指使用一组新的值来替换原来的值，使用 replace 方法。

示例：

```
$user = new User(12);
$user->roles->replace(15);                 # 15 为角色编号
$user->roles->replace(array(12,13));    # 指定角色编号为 14、13
```

9.9.5.3、加载 HasAndBelongs 成员数据

在 HasAndBelongs 关系中，只能使用异步加载，不能进行同步加载。

9.9.6、删除关联模型数据

通常删除模型时，仅仅会删除模型自身的数据，但 Skihat 允许使用关联方式删除关联模型。

示例：

```
$user = new User(12);
# 删除用户相关的中间表信息；
$user->delete(array(Model::SYNTAX_LINK => ‘roles’));
# 删除时将用户组一起删除；
$user->delete(array(Model::SYNTAX_LINKS => array(‘roles’,’group’));

# 删除时将用户组一起删除；
```

```
User::deleteAll(12,array(Model::SYNTAX_LINK => ‘roles’));
User::deleteAll(12,array(Model::SYNTAX_LINK => array(‘roles’,’group’));
```

9.9.7、使用自引用关系

自引用关系是指关系模型就是模型自己，例如：每一员工都一个经理，但经理自身也一个员工。

示例：

```
CREATE TABLE employee (
    `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `manager_id` INT NOT NULL
);

class Employee extends ApplicationModel {
    public static function __config() {
        return array(
                self::META_LINKS => array (
                    ‘manager’ => array(
                        self::META_LINKS_CLASS => ‘models.employee’,
                        self::META_LINKS_TYPE => self::BELONGS_TO,
                        self::META_LINKS_FOREIGN => ‘manager_id’,
                        self::SYNTAX_TABLE_ALIAS => ‘Manager’
                    )
                )
        );
    }
}
```

使用自引用的关键就是使用 SYNTAX_TABLE_ALIAS 指定关系表的别名。

9.9.8、完整字段名称

通常查询、更新或删除时不需要使用完整的字段名称，但如果包含的关系关系就需要使

用完整字段名称，防止字段冲突。

示例：

```
$users = User::fetchAll(array(‘User.group’ => 1),array(‘links’ => ‘Group’);
```

9.10、模型特殊字段

9.10.1、默认字段

在 Skihat 模型中，允许为模型指定字段的默认值，实例化时如果默认字段没有指定将使用默认值，配置 META_DEFAULTS 使用数组值指定模型默认值。

示例：

```
class GuestBook extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_DEFAULTS => array(‘user’ => ‘guest’)
        );
    }
}

# 使用默认值
$guestBook = new GuestBook();
$guestBook->user;       # guest;

# 覆盖默认值
$guestBook = new GuestBook(array(‘user’ => ‘admin’));
$guestBook->user; # admin
```

9.10.2、只读字段

如果在执行 save 方法时，不允许部分字段执行更新，那么就可以声明为只读字段。配置 META_READONLY 使用数组指定模型只读字段。

示例：

```
class GuestBook extends ApplicationModel {
    public static function __config() {
        return array(
            self::META_READONLY => array(‘user’)
        );
    }
}

$guestBook = new GuestBook($this->form(‘guest_book’),true);
$guestBook->user = ‘admins’;
$guestBook->save();                    # 不会执行 user 字段的更新
```

注意：如果需要更新只读字段，只能够使用 updateAll 方法。

9.10.3、虚拟字段

虚拟字段允许使用自定义方法，提供设置字段的值，使用时能够像正常属性一样使用。配置 META_VIRTUAL 使用数组值指定模型虚拟字段。

示例：

```
class GuestBook extends ApplicationModel {
    public static function __config() {
            return array(
                self::META_VIRTUAL => array(‘full_content’)
        );
    }

    # 方法名称必须与虚拟字段的名称相同
    public function full_content($content = null) {
        if (is_null($content)) {        # 返回值
            return $this[‘user’] . ‘:’ . $this[‘content’];
        } else {  # 设置值
             list($user,$content) = explode(‘:’,$content);
```

```
                $this[‘user’] = $user;
                $this[‘content’] = $content;
            }
        }
    }
```

9.10.4、created 字段

created 字段是一个非常特殊的字段，表示模型的创建时间。

示例：

```
$guestBook = new GuestBook($this->form(‘guest_book’));
$guestBook->save();                # guestBook->created 为当前保存时间。
```

注意：created 字段仅对实例的 save 方法有效，createAll 必须手动设置。

9.10.5、modified 字段

模型 modified 字段与 created 类似，表示模型的最后更新时间。

示例：

```
$guestBook = new GuestBook(12);
$guestBook->save();          # guestBook->modified 为当前保存时间。
```

注意：modified 字段仅对实例的 save 方法有效，updateAll 方法必须手动设置。

9.10、数据库方法

9.10.1、Model::database 返回数据库

- 定义：public static function database()

  返回当前模型对象的数据库引擎实例。

9.10.2、Model::query 查询数据库

- 定义：public static function query($cmd,array $args = array(),
  $fetchStyle = self::FETCH_STYLE_ASSOC)

  使用$cmd 和$args 进行查询操作，结果结果为二维数组，更多信息请查看数据库查询。

9.10.3、Model::execute 执行数据库命令

- 定义：public static function execute($cmd,array $args = array())

  使用$cmd、$args 执行命令，并返回影响到的行，更多信息请查看数据库命令。

9.11、模型查询条件

在模型中无论是查询、更新和删除数据，我们经常都会使用到条件参数$conditions，Skihat 对于查询条件的支持非常强大，支持主键、字符串、字符串参数和数组字段四种类型的条件。

9.11.1、主键条件

如果$conditions 参数为数值，则表示该条件为主键条件，表示主键的值=$conditons 设置的值。

示例：

```
# 查询主键值为 1 的 GuestBook 实例；
$guestBook = GuestBook::fetch(1);

# 删除主键值为 1 的 GuestBook 实例；
GuestBook::deleteAll(1);

# 更新主键为 1 的留言板的用户名；
GuestBook::updateAll(array(‘user’ => ‘admin’),1);
```

9.11.2、字符串条件（SQL 命令）

字符串参数条件也被称为 SQL 命令条件，表示条件直接处理为 SQL 的 WHERE 命令，Skihat 不会进行任何处理，直接交由数据库处理条件。

示例：

```
$guestBooks = GuestBook::fetchAll(‘GuestBook.created < ’ . time() - 3600);
GuestBook::deleteAll(‘GuestBook.created < ‘ . time() - 3600);
GuestBook::updateAll(array(‘user’ => ‘admin’),’GuestBook.created < ‘ . time() - 3600);
```

注意：使用字符串条件时要特别注意安全问题，防止 SQL 注入，如果有外部参数建议使用"字符串参数条件"。

9.11.3、字符串参数条件（SQL 命令参数）

SQL 命令参数是一个数组类型的参数，数组的第一个元素表示 SQL 命令，后面的值表示参数值，需要注意参数的对应关系。

示例：

```
$guestBooks = GuestBook::fetchAll(array(‘GuestBook.user = ?’,’guest’));
GuestBook::deleteAll(array(‘GuestBook.user = ?’,’guest’));
GuestBook::updateAll(array(‘user’ => ‘admin’),array(‘GuestBook.user = ?’,’guest’));
```

使用 SQL 命令参数，最大的优势在于能够非常方便的生成复杂的查询条件，同时还能防止 SQL 注入安全。

9.11.4、数组字段条件

当条件指定为数据类型，并且条件中都包含 Key 值时，表示该条件为数组字段条件。

示例：

```
$guestBooks = GuestBook::fetchAll(array(‘GuestBook.user’ => ‘guest’));
GuestBook::deleteAll(array(‘GuestBook.user’ => ‘guest’));
GuestBook::updateAll(array(‘user’ => ‘admin’),array(‘GuestBook.user’ => ‘guest’));
```

在数组字段条件中，数组的 KEY 值将生成条件的字段部分，数组的值部分将生成条件值，各个参数之间使用 AND 命令合并在一起。

示例：

```
# GuestBook.user = ‘guest’ AND GuestBook.created < 13918281
GuestBook::fetchAll(array(‘GuestBook.user’ => ‘guest’,
                          ‘GuestBook.created < ‘ => time() - 3600));
```

9.11.4.1、非等于条件

默认情况下字段和值之间使用"＝"作为条件运算符，如果需要修改默认运行符只需要在 key 中指定值：

示例：

```
$guestBooks = GuestBook::fetchAll(array(‘GuestBook.created < ‘ => time() - 3600));
```

注意：字段和运行符之间至少包含一个空格。

9.11.4.2、使用 in 运行符

如果字段的条件值为数组，Skihat 会认为这是一个 in 运行符的条件判断，那么 Skihat 会默认采用 in 运行符。

示例：

```
# GuestBook.user in (‘guest’,’admin’);
$guestBooks = GuestBook::fetchAll(array(‘GuestBook.user’ => array(‘guest’,’admin’));
```

9.11.4.3、NormalParameter 类

为了保证安全，默认情况下数组字段中的参数值都被 PDO 参数化，但有些时候我们希望不进行参数化，而是直接转换为 SQL 命令，这就需要使用 NormalParameter 类。

示例：

```
# User.group IN SELECT id FROM group WHERE name like ‘%admin%’;
$users = User::fetchAll(‘User.group IN ‘ => new NormalParameter(‘SELECT id FROM group WHERE name like \’%admin%\’));
```

注意：使用 NormalParameter 参数值时，需要特别注意安全问题，防止 SQL 注入安全。

9.11.5、正确使用条件字段

在 Skihat 生成 SQL 查询命令时会自动生成表的别名，默认情况下表的别名与类名相同，为防止命名冲突，建议使用完整字段名称。

示例：

```
# 查询错误，name 为数据库关键字
$users = User::fetchAll(‘name = \’guest\’);
$users = User::fetchAll(‘User.name = \’guest\’);
```

第 10 章：视图组件（View）

视图作为 MVC 的重要组成部分，Skihat 使用视图模板填充请求的响应内容。

10.1、视图接口

10.1.1、IView 接口声明

为了提高扩展性，在 Skihat 使用 IView 接口作为视图必须提供的基础服务。

```
interface IView implements ArrayAccess {
    # 常量声明
    const TEMPLATE_EXT = ‘.stp’;            # 视图的默认文件扩展名；
    const OPT_TEMPLATE = ’ template’;       # 视图模板选项
    const OPT_PASS  = ‘pass’;               # pass 选项参数

    # 缓存选项
    const OPT_CACHE = ‘cache’;                  # 缓存
    const OPT_CACHE_ENGINE = ‘cacheEngine’;  # 缓存引擎
    const OPT_CACHE_EXPIRE = ‘cacheExpire’;  # 缓存过期时间

    # 方法声明
    function initialize(Controller controller,array $options);      # 实始化
    function assignAll(array $vars);                                # 指定视图变量
    function message($message = null);                              # 获取或指定视图消息
    function inflate();                                             # 填充视图模板
}
```

通常我们不需要直接调用 IView 的接口方法，而是由 Controller 内部自动完成调用。

10.1.2、Theme 类声明

Theme 类是 Skihat 框架提供的内置视图模板实现类，允许使用布局、引用模板、视图助手等特性。

```
class Theme implements IView {
    # 模板主题名称
    public $theme = false;
    # 实例方法
    public function layout($layout);                    # 指定布局文件
    public function helpers();                          # 导入视图助手
    # 指定视图占位符
    public function place($name,$default);
    # 模板填充方法
    public function inflateText($text,$place = false);
    public function inflateProc($proc,array $args = array(),$options = array());
    public function inflateFile($file,$options = array());
    public function beginPlace($place);
    public function endPlace();
}
```

10.2、视图模板文件

10.2.1、什么是视图模板文件

视图模板是一个简单的 PHP 文件，用于生成视图响应所需的内容。在 Theme 类中为了保证效率和方便学习，没有编写自定义的模板语言直接采用 php 代码。

示例：

```
# views/template/index.stp
<?php echo ‘Hello Skihat Theme’; ?>
```

10.2.2.、创建模板文件

一个模板文件的路径由模板路径和文件两个部分组成，模板路径表示从那些目录中查找，而文件路径指定找那个文件。

10.2.2.1、模板路径

默认模板路径

使用 Theme 类时，默认会在以下目录查找模板文件：

- app/views：应用模板路径；
- skihat/views：Skihat 库模板路径；

利用优先文件原则，查找时会优先查找 app/views 目录下的模板。

主题 Theme->theme 模板路径

如果在使用时指定 Theme->theme 成员变量，并且 Theme->theme 的值为 defaults，则查找目录为：

- app/themes/defaults：主题模板；
- app/views：应用模板；
- skihat/views：Skihat 库模板路径；

在控制器中指定 theme，请使用 Controller->actionView 方法。

10.2.2.2、文件路径

默认文件路径是由当前请求决定，具体情况请查看控制器与视图。

10.3、模板文件局部变量

在任何一个模板文件中允许使用$this、$request 和$response 三个局部变量：

- $this：当前 Theme 对象的实例，可以通过$this 调用局 Theme 的成员方法、属性；
- $request：当前请求的 ApplicationRequest 实例对象；
- $response：当前响应的 ApplicationResponse 对象实例；

10.3.1、使用 ArrayAccess 访问视图变量

访问从控制器传递的视图变量，使用 ArrayAccess 接口，获取变量值。

示例：

```
# app/controllers/index_controller.inc
public function indexAction() {
    $this[‘hello’] = ‘Hello Skihat’;
}

# views/index/index.stp
<?php echo $this[‘hello’]; ?> # 注意：键必须与控制器中对应。
```

如果访问的变量不存在，则返回 null 值。

10.3.2、使用$response 变量

通过使用$response 变量，我们可以在视图中非常方便的设置响应的信息。

示例：

```
# 文件：skihat/views/__errors/404.stp
<?php $response->statusCode(‘404 Not Found Page'); ?>
```

10.3、使用布局文件

10.4.1、什么是布局文件

在一个网站中，通常大部分的页面都会被设置为几个固定的布局，在 Skihat 中允许将这些固定的布局由模板文件来指定，这些文件被称为布局文件。

10.4.2、使用布局文件

10.4.2.1、布局文件目录

为了保证 Skihat 会查找到布局文件，布局文件必须声明在以下目录：

- app/views/__layouts：应用布局文件目录；
- skihat/views/__layouts：Skihat 框架布局文件目录；

同模板文件目录类似，如果指定 Theme->theme，那么将允许在主题中声明布局文件。

- app/themes/defaults/__layouts：主题布局文件路径；
- app/views/__layouts： 应用布局文件路径；
- skihat/views/__layouts：Skihat 框架布局文件目录；

defaults 表示主题的名称。

10.4.2.2、创建布局文件

布局文件与普通 Skihat 视图模板没有任何区别，只需要在需要由模板文件填充的位置指定占位符。

示例：

```
# app/views/__layouts/default.stp
<?php echo ‘layout file’; ?>
```

10.4.2.3、使用 Theme->layout 在模板中指定布局

- 定义：public function layout($layout);

在模板文件中使用布局文件非常简单，只需要使用$this 关键字指定布局的名称。

示例：

```
# app/views/template/layout.stp
<?php $this->layout(‘default’); ?>
```

10.4.2.4、使用 Theme->place 声明占位符

- 定义：public function place($name,$default = '')

使用$name 和$default 参数布局文件中声明占位符。

在一个布局文件中允许声明一个或多个布局占位符。在视图模板文件中允许使用内容来去替换占位符的默认值。

示例：

```
# app/views/__layouts/place.stp
<html lang="zh-CN">
    <head><title><?php $this->place('title','Skihat example'); ?></title></head>
    <body><?php $this->place('body','body content'); ?></body>
</html>
```

在 place.stp 布局文件中声明了二个占位符 title 和 body。

10.4.2.5、使用 beginPlace 和 endPlace 方法填充占位符

- 定义：public function beginPlace($place);

  使用$place 填充布局文件的占位符内容，endPlace 表示内容结束。

示例：

```
# app/views/template/place.stp
<?php $this->layout('place'); ?>
<?php $this->beginPlace('title'); ?>
        Hello Skihat Example
<?php $this->endPlace(); ?>

<?php # beginPlace 和 endPlace 之间不应当有任何输出代码 ?>
<?php $this->beginPlace('body'); ?>
    <h1>Hello Skihat Example</h1>
<?php $this->endPlace(); ?>
```

执行结果：

```
<html lang="zh-cn">
    <head><title>Hello Skihat Example</title></head>
    <body>Hello Skihat Example</body>
</html>
```

注意：在使用 beginPlace 和 endPlace 方法时必须成对出现，同时在两者之外不应当有任何代码。

10.5、使用模板填充方法

为了便于生成模板信息，在 Theme 类中，声明有一组 inflateXXX 的方法，帮助生成模板的内容，这一组方法都使用$this 变量访问。

10.5.1、Theme->inflateText 文本填充方法

- 定义：public function inflateText($text,$place = false)

根据$text 和$place 参数，填充文本内容。

通常使用 inflateText 方法，会直接将文件内容输出到当前模板，与使用 echo 方法效果相同。

示例：

```
# 文件：app/views/template/inflate_text.stp
$this->inflateText(‘Hello Skihat inflate text’);
```

执行结果：

```
Hello Skihat inflate text
```

使用 inflateText 方法更重要的作用是，为了布局占位符提供文本内容。

示例：

```
# app/views/template/inflate_text_place.stp
<?php $this->layout(‘place’); ?>
<?php $this->inflateText(‘Hello Skihat inflate text’,’body’); ?>
```

执行结果：

```
<html lang="zh-cn">
    <head><title>Skihat Example</title></head>
    <body>Hello Skihat inflate text<hr/></body>
</html>
```

10.5.2、Theme->inflateProc 函数填充

- 定义：public function inflatePlace($proc,array $args = array(),$options = array())

根据$proc,$args 和$options 使用自定义方法填充视图模板。

示例：

```
# app/views/template/inflate_proc.stp
<?php function hello($text) { echo ‘Hello ‘,$text; } ?>
<?php $this->inflateProc(‘hello’,array(‘Skihat’)); ?>
```

执行结果：

```
Hello Skihat
```

10.5.2.1、使用匿名函数

使用 inflateProc 时，还可以直接使用匿名回调函数。

示例：

```
# app/views/template/inflate_proc_noname.stp
<?php $this->inflateProc(function ($text) {
        echo 'Hello ', $text;
}, array('Skihat')); ?>
```

执行结果：

```
Hello Skihat
```

10.5.2.2、使用布局占位符

使用字符串参数选项

如果$options 类型为字符串，表示使用方法填充布局占位符中相应的内容。

示例：

```
# app/views/template/inflate_proc_place.stp
<?php $this->layout('place'); ?>
```

```
<?php $this->inflateProc(function($text) { echo 'Hello ',$text; },array('Skihat'),'title'); ?>
```

执行结果：

```
<html lang="zh-cn">
    <head><title>Hello Skihat</title></head>
    <body>body content</body>
</html>
```

使用 Theme::OPT_PLACE 选项

如果还有更多选项，使用使用数组选项，使用 Theme::OPT_PLACE 指定占位符。

示例：

```
# app/views/template/inflate_proc_opt_place.stp
<?php $this->inflateProc(‘hello’,array(’Skihat’), array(self::OPT_PLACE => ’title’));
```

执行结果：

```
<html lang="zh-cn">
    <head><title>Hello Skihat</title></head>
    <body>body content</body>
</html>
```

10.5.2.3、使用 Theme::OPT_CACHE 选项

使用 Theme::OPT_CACHE 选项值，可以为填充方法指定缓存信息，使用缓存后，内容会首先从缓存中获取，如果缓存的值不存在才会调用方法生成内容。

示例：

```
# app/views/template/inflate_proc_cache.stp
<?php $this->inflateProc(function ($text) {
    echo 'Hello ', $text, ' time:', date(SKIHAT_I18N_DATE_TIME);
}, array('Hello'), array(
        Theme::OPT_CACHE => 'hello-skihat'
));
```

Theme::OPT_CACHE 参数用于指定缓存的名称，如果多个模板中使用相同方法和参数填充，可以设定相同的 OPT_CACHE 值。

除了 Theme::OPT_CACHE 参数外，还有以下两个缓存选项：

- Theme::OPT_CACHE_ENGINE：指定缓存引擎，默认为 default。
- Theme::OPT_CACHE_EXPIRE：缓存过期时间，默认为 3600 秒。

10.5.3、Theme->inflateFile 文件填充

- 定义：public function inflateFile($file,$options = array())

使用$file 和$options 参数填充视图模板，$file 指定为模板的名称。

10.5.3.1、使用 inflateFile 方法

inflateFile 方法用于将另一个模板的内容，输出到当前的模板中，并允许指定占位符和缓存信息。

示例：

```
# 文件：app/views/elements/title.stp
<?php echo ‘Hello Skihat’; ?>

# 文件：app/views/template/inflate_file.stp
<?php $this->inflateFile(‘elements.title’); ?>
```

执行结果：

```
Hello Skihat
```

注意：inflateFile 中的$file 以模板路径为基础进行文件查找，文件名不需要指定扩展名，如果包含子目录使用”."分隔目录。

10.5.3.2、指定布局占位符选项

- **使用字符串选项**

如果选项信息为字符串，表示指定布局模板的占位符。

示例：

```
# 文件：app/views/template/inflate_file_place.stp
<?php $this->layout('place'); ?>
<?php $this->inflateFile('elements.title', 'title'); ?>
```

执行结果

```
<html lang=“zh-CN”>
    <head><title>Hello Skihat</head>
    <body>body content</body>
</html>
```

- **使用 Theme::OPT_PLACE 选项**

Theme::OPT_PLACE 的使用方式与 inflateProc 方法相同。

10.5.3.3、使用 Theme::OPT_CACHE 缓存选项

使用缓存选项的方式与 inflateProc 相同，请参考 inflateProc 方法。

10.6、使用视图助手

视图助手是用于生成模板内容的一组帮助函数或类，帮助我们简化模板的编写。

10.6.1、引用视图助手

- 定义：public function helpers()

根据参数引用视图助手，将视图助手方法导入到当前模板参数中。

示例：

```
# app/views/template/helper.stp
<?php $this->helpers(‘core’); ?>
```

或

<?php $this->helpers(‘core’,’form’);　　# 使用多个助手

使用名称为 core 的视图助手，视图助手的名称就是助手文件的名称（core => core.inc），视图助手的文件扩展名为.inc。

10.6.2、视图助手查找路径

当没有指定主题时，helpers 方法会在以下目录中进行查找：

- app/views/__helpers：应用自定义视图助手；
- skihat/views/__helpers：Skihat 库提供的视图助手；

如果指定主题后，helpers 方法会在以下目录中查找：

- app/themes/defaults/__helpers　　# defaults 为主题名称
- app/views/__helpers　　# 应用自定义视图助手
- skihat/views/__helpers　　# skihat 库提供的视图助手

根据以上规则，如果需要替换默认的助手方法，只需要在指定优先目录中创建相同的文件，并提示相同的方法即可。

10.6.3、skihat/views/__helpers/core.inc 核心助手方法

在 core.inc 文件中声明了一组核心的助手方法。

10.6.3.1、flash_message 函数

- 定义：function flash_message(IView $view)

将消息转换为闪存消息（一次性消息），通常我们不直接使用本方法。

10.6.3.2、text_message 函数

- 定义：function text_message(IView $view, $name = 'flash')

根据$view 提示构建文本消息字符串，并返回值，如果想要直接输出请使用_text_message

方法。

示例：

```
public function editAction() {
    if ($this->isPost()) {
        $this[‘guest_book’] = new GuestBook($this->form(‘guest_book’),true,true);

        if ($this[‘guest_book’]->save()) {
            $this->message('更新留言内容成功’);
            $this->redirect(‘index’);
            return;
        }
    }
}

# app/views/guestbooks/index.stp
<?php $this->helpers(‘core’); ?>
<?php _text_message($this); ?>
```

10.6.3.3、h 函数

- 定义： function h($text)

使用 htmlspecialchars 方法处理$text 参数，如果直接输出请使用_h 方法。

示例：

```
h(‘<div >Hello World</div>’);   # &lt;div&gt;Hello World&lt;/div&gt;
```

10.6.3.4、hs 函数

- 定义：function hs($text, $len, $start = 0)

使用 htmlspecialchars 格式化$text 文本，并且指定为$len，从$start 开始，如果直接输出请使用__hs。

示例：

```
hs(‘Hello World’,3);        # Hel
hs(‘Hello World’,3,2);      # llo
```

在 hs 内部使用 mb_substr 进行文本的截取。

10.6.3.5、url 函数

- 定义：function url($url = array())

根据$url 生成 URL 地址，$url 参数允许是字符串或数组，字符串表示 action 参数，直接输出请使用_url 函数。

示例：

url（‘index’）;　　# http://www.example.com/?action=index

url(array(SKIHAT_PARAM_ACTION => ‘index’));　# 同上

注意：在 url 函数内部将调用 Router::url 方法生成结果，更多内容请参考 Router::url 方法。

10.6.3.6、link_to 函数

- 定义：function link_to($text, $url = array(), $options = array())

根据$text、$url 和$options 生成 A 标签 HTML，如果直接输出使用_link_to 函数。

示例：

<a href="http://www.example.com/?action=edit&id=1" title="编辑">点击<a>

link_to(‘点击’,array(‘action’ => ‘edit’,’id’ => 1),array(‘title’ => ‘编辑’));

生成时$url 参数内部将调用 Router::url，而$options 则会直接生成 HTML 属性。

10.6.3.7、link_image 函数

- 定义：function link_image($image, $url = array(), $options = array())

使用$image、$url 和$options 生成带图片的 A 标签 HTML，直接输出请使用_link_image 函数。

示例：

#
link_image(array(‘src’ => ‘/images/edit’),array(‘action’ => ‘edit’));

10.6.3.8、mail_to 函数

- 定义：function mail_to($email, $options = array())

使用$email 生成带 mailto 的 A 标签 HTML，直接输出使用 _mail_to 函数。

示例：

# test@example.com
mail_to(‘test@example.com’,array(‘title’ => ‘点击’));

10.6.3.9、i18n 函数

- 定义：function i18n($name)

根据$name 和格式参数输出 i18n 国际化内容，直接输出使用_i18n 函数。

示例：

i18n(‘core:hello’,’Skihat’); # hello Skihat

如果指定的参数值大于 1 个，后面的参数将作为格式化参数，i18n 的更多内容请参考 Skihat::i18n 函数。

10.6.3.10、nice_date 函数

- 定义：function nice_date($date, $format = SKIHAT_I18N_DATE)

根据$date 返回格式化日期字符串，直接输出使用_nice_date 函数。

示例：

nice_date(time()); # 2013-08-12

10.6.3.11、nice_date_time 函数

- 定义：function nice_date_time($date, $format = SKIHAT_I18N_DATE_TIME)

根据$date 返回格式化时间字符串。

示例：

nice_date_time(time()); 2013-08-12 11:23:13

10.6.3.12、nice_time 函数

- 定义：function nice_time($time, $format = SKIHAT_I18N_TIME)

根据$time 返回格式化时间字符串，如果直接输出使用_nice_time 函数。

10.6.3.13、nice_week 函数

- 定义：function nice_week($date, $format = SKIHAT_I18N_WEEK)

根据$date 返回时间的周信息，如果直接输出使用_nice_week 函数。

10.6.3.14、nice_last_time 函数

- 定义：function nice_last_time($date)

根据$date 返回过去的时间字符串的友好形式，输出输出请使用_nice_last_time 函数。

示例：

```
nice_last_time(time() - 3)          # 3 秒钟前
nice_last_time(time() - 3700)       # 1 个小时前
```

10.6.4、skihat/views/__helpers/form.inc 表单助手

表单助手提供生成表单各个部分的服务，内部主要包含 FormBuilder 和 Form 两个类。

10.6.4.1、类声明

```
class FormBuilder {
```

```
    public function __construct(IView $context）;    # 初始化 FormBuilder 类
    public function begin(array $attrs = array());    # <form xx=xx>生成表单开始标志；
    public function end();                            # </form>

    public function label($field,$attrs = array());    # 生成 label 标签
    public function hidden($field,$attrs = array());   # 生成隐藏域

    # 文本录入框
    public function text($field,$attrs = array());         # 生成文本录入框
    public function textarea($field,$attrs = array());     # 文本域录入框
    public function email($field,$attrs = array());        # 生成邮件录入框
    public function password($field,$attrs = array());     # 生成密码录入框
    public function url($field,$attrs = array());          # 生成 URL 录入框

    # 数值录入框
    public function number($field,$attrs = array());       # 生成数值录入框
    public function range($field,$attrs = array());        # 生成范围录入框
    public function file($field,$attrs = array());         # 生成文件录入框
    public function datetime($field,$attrs = array());     # 生成日期时间录入框
    public function date($field,$attrs = array());         # 生成日期录入框
    public function time($field,$attrs = array());         # 生成时间录入框
    public function month($field,$attrs = array());        # 月录入框
    public function week($field,$attrs = array());         # 周录入框

    # 选项录入框
    public function checkbox($field,$attrs = array());     # 复选框
    public function radio($field,$attrs = array());        # 单选框
    public function select($field,$attrs = array());       # 下接录入框
    public function help($field,$attrs = array());         # 生成帮助信息
}
```

FormBuilder 类的录入框会基于 HTML 5 生成相关的内容，录入框也是按 HTML 5 规范进行设计。

```
class Form extends FormBuilder {}
```

Form 类与 FormBuilder 类的方法完全相同，不同在于 Form 类会将生成的表单 HTML

结果直接输出到模板。

10.6.4.2、使用表单类

使用表单类非常简单，只需要引用表单助手构建实例。

示例：

```
# app/views/template/form.stp
<?php $this->helpers('form'); ?>
<?php $form = new Form($this); ?>
```

10.6.4.3、Form->begin 生成表单标签

- 定义：public function begin($array $attrs = array());

根据$attrs 生成表单开始标志,$attrs 会自动生成为表单属性。

示例：

```
# <form action => "xxx" method="post">
<?php $form->begin(array(‘method’ => ‘post’,’action’ => url(‘edit’)); ?>
```

如果没有指定表单 action，表单将自动使用当前 URL 地址。如果需要结束表单声明，则直接使用$form->end 方法。

示例：

```
# </form>
<?php $form->end(); ?>
```

10.6.4.4、表单录入参数

所有表单录入框都包含$field 和$options 两个参数，两个参数的具体用法为：

- $field 参数：指定录入框填充值的视图变量和属性，格式为：变量名称.变量属性；
- $options 参数：如果没有特别说明$options 会生成为对应标签的 HTML 属性；

示例：

```
public function editAction() {
    $this[‘guest_book’] = GuestBook::find($this->query(‘id’));
}

# app/views/template/form.stp
<?php $form = new Form($this); ?>
<?php $form->text(‘guest_book.user’); ?>
```

执行结果：

```
<input type="text" value="guest" name=‘guest_book[user]’  id = "guest_book_user" />
```

text 方法中，guest_book.user 表示使用视图变量 guest_book 的 user 属性，填充当前录入框的值，如果 guest_book 不存在或 user 属性值不存在，则不会生成 value 选项。

10.6.4.5、Form->label 生成 label 标签

- 定义：public function label($field,$attrs = array());

根据$field 和$attrs 生成 HTML 表单 label 标签。

- **使用字符串参数**

使用字符串参数，直接表示标签的标题信息。

示例：

```
# <label id="guest_book_user_label" for="guest_book_user">用户名：</label>
$form->label(‘guest_book.user’,’用户名:’);
```

- **使用数组参数**

如果使用数组参数，title 表示指定的标题信息，其它信息生成 html 标签属性。

示例：

```
# <label id="guest_book_user_label" for="guest_book_user" class="label">用户名：</label>
$form->label(‘guest_book.user’,array(‘title’ => ‘用户名:’,’class’ => ‘label’));
```

10.6.4.6、Form->hidden 生成 hidden 隐藏域

- 定义：public function hidden($field,$attrs = array());

根据$field 和$attrs 生成 HTML 表单 input hidden 标签。

示例：

<input type="hidden" value="1" name="guest_book[id]" />

$form->hidden(‘guest_book.id’);

10.6.4.7、表单录入框

为了满足不同的表单录入需求，表单助手声明了一系列的方法，这些方法包括：

- public function text($field,$attrs = array()); # 文本录入框
- public function textarea($field,$attrs = array()); # 文本域录入框
- public function email($field,$attrs = array()); # 邮件录入框
- public function password($field,$attrs = array()); # 密码录入框
- public function url($field,$attrs = array()); # URL 录入框
- public function number($field,$attrs = array()); # 数值录入框
- public function range($field,$attrs = array()); # 范围录入框
- public function file($field,$attrs = array()); # 文件录入框
- public function datetime($field,$attrs = array()); # 日期时间录入框
- public function date($field,$attrs = array()); # 日期录入框
- public function time($field,$attrs = array()); # 时间录入框
- public function month($field,$attrs = array()); # 月录入框
- public function week($field,$attrs = array()); # 周录入框

这一组方法的使用方式相同，使用$field 指定视图变量和属性，$attrs 指定 HTML 选项值。

示例：

指定录入值为 guest.

$form->input(‘guest_book.user’,’guest’);

$form->input(‘guest_book.user’,array(‘value’ => ‘guest’));

如果没有指定 value 值，表单助手会从 guestbook 变量中获取 user 属性，作为 value 值。

10.6.4.8、Form->checkbox 生成复选框

- 定义：public function checkbox($field,$attrs = array());

根据$field 和$attrs 生成复选录入框，checkbox 标签不同于其它录入框，因为 checkbox 指定的值是表示返回的值，但 checkbox 是否返回值是由 checked 属性来判断的。

示例：

```
<label><?php $this->checkbox(‘nuke_user.allow’,1); ?>允许访问</label>
```

通常 checkbox 方法会自动根据视图变量的属性判断是否指定 checked 属性，如果需要手动指定请使用数组选项：

示例：

```
<label><?php $this->checkbox(‘nuke_user.allow’,
                array(‘value’ => 1,’checked’ => true)); ?>允许访问</label>
```

还有一个问题，在 checkbox 录入框返回值时，如果没有选中则指定的名称的值不会 post 传回服务器，这是由于 HTTP 协议造成的，因此 checkbox 方法在生成是会自动生成一个 hidden 标签。

- <input type="hidden" value="0" name="nuke_user[allow]" />

10.6.4.9、Form->radio 生成单选框

- 定义：public function radio($field,$attrs = array());

根据$field 和$attrs 生成单选框，radio 不同于其它录入框，因为 radio 标签的 value 属性是指定值，但指定是否选中确是使用 checked 属性。

示例：

```
<label><?php $this->radio(‘guest_book.user’,’admins’); ?>管理员</label>
<label><?php $this->radio(‘guest_book.user’,’guest’); ?>宾客</label>
```

通常表单助手会自动根据视图变量判断是否添加 checked 属性，如果需要手动指定可以使用数组选项。

示例：

```
<label><?php $this->radio(‘guest_book.user’,
        array(‘value’ => ‘admins’,’checked’ => true)); ?>管理员</label>
<label><?php $this->radio(‘guest_book.user’,array(‘value’ => ‘guest’)); ?>宾客</label>
```

10.6.4.10、Form->select 下拉列表

- 定义：public function select($field,$options = array())

  根据$field 和$options 生成 HTML 下拉标签。

示例：

```
#<select name="guest_book[user]" id="guest_book_user">
    <option>请选择管理员</option>
    <option value="admin">管理员</option>
    <option value="guest">宾客</option>
</select>
$form->select(‘guest_book.user’,array(
    ‘options’ => array(‘admin’ => ‘管理员’,‘guest’ => ‘宾客’),
    ‘empty’ => ‘请选管理员'
));
```

同录入框相同，如果指定的 guest_book 的 user 属性，则 select 方法会自动设置值。

特殊$options 选项

在 select 方法中，指定 options 允许有以下特殊选项：

- options：指定选项值，默认为空；
- empty：指定空值文本，默认为空；

10.6.4.11、Form->help 生成帮助信息

- 定义：public function help($field,$attrs = array());

根据$field 和$attrs 生成帮助标签。

示例：

```
# <em>用户名不能为空<em>
$form->help(‘guest_book.user’,’用户名不能为空!’);

# <span>用户名不能为空!</span>
$form->help(‘guest_book.user’,array(‘elem’ => ‘span’,
                                    ’title’ => ‘用户名不能为空!’));
```

10.6.4.12、模型错误信息

当执行模型的更新操作，如果因为内部字段验证失败而产生的错误，Form 表单会自动进行处理，具体处理为：

- 录入框：录入框自动增加 input-error 类选择器；
- help：帮助提示，使用错误消息替换默认的帮助方法；

示例：

```
public function editAction() {
    if ($this->isPost()) {
        # 使用$this[‘guest_book’] 防止提交失败后数据失效
        $this[‘guest_book’] = new GuestBook($this->form(‘guest_book’),true,true);
        if ($this->save()) {
            $this->message(‘保存成功’,SKIHAT_MESSAGE_SUCCESS);
            $this->redirect(‘index’);
            return;
        }
    } else {
        $this[‘guest_book’] = GuestBook::find($this->query(‘id’));
    }
}

# app/views/guestbooks/edit.stp
<?php $this->helpers(‘core’,’form’); ?>
<?php $form = new Form($this); ?>
```

```
<?php $form->begin(); ?>
    <p>
        <?php $form->label(‘guest_book.user’,’用户名:’); ?>
        <?php $form->text(‘guest_book.user’); ?>
        <?php $form->help(‘guest_book.user’,‘用户名不能为空’);
    </p>
    <p>
        <?php $form->label(‘guest_book.content’,’留言内容:’); ?>
        <?php $form->textarea(‘guest_book.content’); ?>
    </p>
<?php $form->end(); ?>
```

注意：因为表单助手默认会生成与变量名称相一致的录入框名称，因此在控制器中也是使用 guest_book 视图变量。

10.6.5、skihat/views/__helpers/paginate.inc 分页助手

分页助手帮助我们生成模型列表的分页信息，要支持分页，必须在 Skihat 控制器中进行相关的设置。

- 控制器：使用 paginate_filter 过滤请求参数，更多信息请查看控制器；
- 模型：使用 ModelFetch 类提供设置当前页选项；
- 视图：提供 paginate.inc 分页助手，生成分页信息；

10.6.5.1、使用分页过滤器（PaginateFilter）

```
class PaginateController extends ApplicationController {
    public function helperAction() {
        $this[‘guestbooks’] = GuestBooks::fetchAll();
    }

    public function actionFilters() {
        return array(
            ‘paginate’ => array(
            SKIHAT_IOC_CLASS => ‘controllers.filters.paginate_filter’
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
            ‘var’ => ‘guestbooks’# 必须与视图变量 guest_books 对应
          )
```

```
            );
        }
    }
```

10.6.5.2、paginate_url 函数

- 定义：function paginate_url($page, $ruleName = false)

根据$page 和$ruleName 生成分页 URL 地址，直接输出使用_paginate_url 函数。

10.6.5.3、paginate_link 函数

- 定义： function paginate_link($title, $page, $ruleName = false)

根据$title、$page 和$ruleName 生成 HTML 分页连接，直接输出使用_paginate_link 函数。

10.6.5.4、paginate_numbers 函数

- 定义： function paginate_numbers(ModelFetch $fetch, array $options = array())

根据$fetch 和$options 生成标准数字分页导航，直接输出使用_paginate_numbers 函数。

示例：

```
# app/views/paginate/helper.stp
_paginate_number($this[‘guest_book’]);
```

执行结果：

```
<ul class="pagination”>
    <li class="first"><span>&lt;&lt;</span></li>
    <li class="prov"><span>&lt;</span></li>
    <li class="active"><span>1</span></li>
    <li class="active”><a href=“#">2</a></li>
    <li class="next"><a href=“#">&gt;</a></li>
    <li class="last"><a href=“#">&gt;&gt;</a></li>
</ul>
```

在 paginate_numbers 函数中，允许使用以下三个选项：

- class：指定分页 ul 的 class 属性（默认为 pagination）；
- numbers：指定显示的分页数量（默认为 9）；
- rule：指定使用的 url 参数规则（默认为无）；

10.6.5.5、paginate_sort_link 函数

- 定义：function paginate_sort_link($title, $sort, $dir = false, $ruleName = false)

根据$title、$sort、$dir 和$ruleName 生成排序连接，直接输出使用 paginate_sort_link 函数。

示例：

创建排序连接，标题使用"创建时间"，默认采用 GuestBook.created 的反向排序。

_paginate_sort_link(‘创建时间’,’GuestBook.created’,’desc’);

10.6.5.6、paginate_navigator 函数

- 定义：function paginate_navigator(ModelFetch $fetch, $options = array())

根据$fetch 和$options 生成分页导航(上一页、下一页)，直接输出使用_paginate_navigator 函数。

示例：

_paginate_navigator($this[‘guest_books’]);

在 paginate_navigator 中允许使用以下选项：

- prov-title：上一页标题。
- next-title：下一页标题；
- rule：路由规则名称；
- class：连接标签 class 属性；

10.6.6、自定义视图助手

自定义助手非常简单，不需要继承任何类或引用文件，只需要在视图助手目录下创建自

定义助手文件。

示例：

```
# app/views/__helpers/custom.inc
function hello($text) {
    return ‘Hello ‘ . $text;
}

#Skihat 默认会使用下划线标识出直接输出版。
function _hello($text) { echo hello($text); }
```

使用自定义助手与使用 Skihat 提供的助手一样，也是使用 helpers 方法引用。

示例：

```
# app/views/template/custom_helpers.stp
<?php $this->helpers(‘custom’); ?>
<?php echo hello(‘World’); ?>
```

执行结果：

```
Hello World
```

10.7、自定义功能模板

在控制器中，很多功能最后都是由模板生成内容来完成的，这些内容包括：

- Controller->error：输出错误消息；
- Controller->redirect：执行客户端跳转；
- Controller->text：使用文本输出；
- Controller->json：使用 json 格式输出；
- Controller->jsonp：使用 jsonp 输出跨站点响应。

总结起来就是分为两个部分，控制器功能部分和错误页。

10.7.1、控制器功能部分

控制器功能的模板都是声明在视图路径的__ctlfuns 目录中：

- json.stp：使用 Controller->json 生成 json 格式数据；
- text.stp：使用 Controller->text 生成文本响应内容；
- redirect.stp：使用 Controller->redirect 生成客户段跳转内容；

如果我们想要修改 Skihat 所提供的默认行为，就可以直接在 app/views/__ctlfuns 目录下定义相同的文件。

10.7.2、错误页

当控制器调用$this->error 方法后，Skihat 就会调用视图路径下__errors 的错误模板来生成响应的内容，文件名与错误代码相同：

- 400.stp：客户段错误代码 Controller->error(400)；
- 404.stp：指定页未找到错误 Controller->error(404)；
- 500.stp：发生内部错误 Controller->error(500)；
- error.stp：默认错误页，如果指定错误码的模板没有找到则使用 error.stp；

同控制器方法相同，如果我们需要修改默认的错误信息，只需要在 app/views/__errors 目录下定义相同的文件。

第 11 章：路由组件（Routers）

路由是 Skihat MVC 的重要组件，负责将外部请求参数转换为内部参数集，同时 Skihat 的路由还是双向路由，即允许将内部参数转换为外部请求地址。

11.1、路由类声明

整个路由组件由路由器（Router）、路由样式接口(IRouterStyle)和路由规则接口(IRouterRule)组成。

- Router：路由器用于封装路由内部服务，提供外部访问接口;
- IRouterStyle：根据反向参数集，生成完整 URL 地址；
- IRouterRule：根据路由规则生成将请求参数转换为完整参数集（正向/反向转换）;

11.1.1、Router 路由器类声明

```
class Router {
    # 常量声明
    const FULL_URL = ‘fullUrl’;          # 完整参数配配
    const RULE_NAME = ‘ruleName’;        # 路由规则

    # 静态方法
    public static function domain($domain);                    # 声明子域名

    # 根据$params 匹配完整参数集
    public static function match(array $params);
    # 根据$params 反向生成 URL 地下
    public static function url(array $params);
    # 设置默认参数集
    public static function defaultParams(array $basicParams,array $fullParams);
    # 实例方法
    public function __construct($domain);        # 初始化路由器
    public function style($style);               # 设置路由样式
    public function rule($rule);                 # 附加路由样式
```

```
}
```

11.1.2、IRouterStyle 路由样式接口声明

```
interface IRuleStyle {
    const DEFAUT_SCRIPT = ‘index.php’;      # 默认请求脚本
    function fullUrl(array $params = array());     # 根据$params 生成完整 URL 地址
}
```

11.1.3、IRouterRule 路由规则接口声明

```
interface IRouterRule {
    function parse(array $params = array());          # 根据$params 转换参数集
    function reverse(array $params = array());        # 根据$params 反转参数集
}
```

11.2、Router 路由器

路由器是路由组件的外部访问接口，帮助我们访问路由组件的相关服务。

11.2.1、Router::domain 声明域名

- 定义：public static function domain($domain)

根据$domain 返回该域名下的路由实例。

在 Skihat 中允许将同一应用部署到多个子域名，并且提供不同的 URL 样式和重写规则。为了满足这一需求 Skihat 使用 domain 方法配置域名信息。

示例：

```
Router::domain(‘admin.example.com’);      # 配置 admin.example.com 子域名
Router::domain(‘www.example.com’);        # 配置 www.example.com 子域名
```

完成域名的配设置后，Skihat 内部会生成域名相关的 Router 实例，管理域名的 URL 样

式和重写规则，Router->style 和 Router->rule 将会提供更多的说明。

11.2.2、Router->style 设置路由样式

- 定义：public function style($style)

根据$style 指定当前路由样式的 Ioc 配置信息，并返回路由实例。

正确设置域名后，style 方法用于指定该域名的 URL 样式，在 Skihat 中支持普通（NormalRouterStyle）、重写(RewriteRouterRule)和 PathInfo(PathinfoRouterRule)三种路由样式，默认使用普通样式。

示例：

```
# 设置 admin.example.com 使用普通样式
Router::domain(‘admin.example.com’)->style(‘#NormalRouterStyle’);
或
$router = Router::domain(‘admin.example.com’);
$router->style(‘#NormalRouterStyle’);
```

注意：$style 为 Ioc 配设置，更多样式请查看 IRouterStyle 路由样式。

11.2.3、Router->rule 附加路由规则

- 定义：public function rule($rule,$names = false)

根据$rule 和$names 增加路由规则，如果名称为 false 表示匿名规则。

rule 方法声明附加域名的规则，同一域名允许附加多个规则，Skihat 支持普通（NormalRouterRule）、正则规则(RegexRouterRule）和 Restful 规则（RestfulRouterRule）。

示例：

```
# 为 admin.example.com 附加普通规则
Router::domain(‘admin.example.com’)->rule(‘#NormalRouterRule’);
```

或

$router = Router::domain(‘admin.example.com’);

$router->rule(‘#NormalRouterRule’);

同一个域名附加多个路由规则是为了满足域名的不同请求形式，更多信息请查看具体规则的配置。

注意：虽然在同一域名下允许包含多种类型的路由规则，但为了防止相互冲突建议采用一种类型的规则。

11.2.4、Router::url 反向生成 URL 地址

- 定义： public static function Router::url(array $params = array())

根据$url 生成完整 URL 地址并返回，如果地址生成失败返回 false。

在路由组件中 match 和 url 方法是一组正好相反的方法，两者都以路由的 Url 样式和规则为基础，进行参数的匹配和 URL 地址的生成。

示例：

Router::domain(‘admin.example.com’)->rule(‘#NormalRouterRule’);

/index.php?controller=index&action=edit

Router::url(array(‘action’ => ‘edit’,’controller’ => ‘index’));

基本参数集

Skihat 在生成 URL 地址时非常聪明，不需要提供完整的 URL 地址就能生成完整的 URL 地址。

示例：

地址：http://admin.example.com/?controller=users&action=edit

执行：Router::url(array(‘action’ => ‘index’));

结果：http://admin.example.com/?controller=users&action=index

通常自动提供这种方式的处理，是因为在 Router 内部有一组默认的基本参数集，在生成 URL 时，会首先合并这一组参数集：

- SKIHAT_PARAM_MODULE：模块参数，默认参数；
- SKIHAT_PARAM_PACKGE：包参数，默认参数；
- SKIHAT_PARAM_CONTROLLER：控制器参数，默认参数；
- SKIHAT_PARAM_ACTION：活动参数，默认参数；

在每一次请求中并不是五个默认参数都会有效，Skihat 会根据请求自动进行判断。

完整参数集

同基础参数集相对应的是完整参数集，完整参数集会将当前的全部请求参数作为默认参数，使用时需要使用 FULL_URL 常量选项。

示例：

地址：http://admin.example.com/?controller=users&action=edit&ac=1&id=2

执行：Router::url(array(‘action’ => ‘index’,Router::RULL_URL => true));

结果：http://admin.example.com?controller=users&action=index&ac=1&id=2

使用完整参数能够非常方便的解决查询的状态保存，例如：在用户列表页执行删除命令，为了保证删除后依赖在正确的分页位置上，就可以使用完整参数，保存查询的条件。

示例：

```
# users/index.stp
<a href="<?php echo Router::url(array(‘action’ => ‘edit’,
                Router::FULL_URL => true)); ?>">删除</a>

# app/controllers/users_controller.inc
public function deleteAction() {
    if (User::deleteAll($this->query(‘id’))) {
        $this->redirect(array(Router::FULL_URL => true));
    }
}
```

11.2.5、使用$names 声明命名规则

因为 Skihat 路由组件支持子域名配置，这就需要解决跨域名 URL 地址生成问题，在 Skihat 中这个问题主要是使用命名规则来解决的，$names 参数允许指定单个名称也可以指

定多个名称。

示例：

```
# 指定命名规则为’admins’
Router::domain(‘admin.example.com’)
    ->style(array(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’,
                  ‘domain’ => ‘http://admin.example.com'))
    ->rule(‘#NormalRouterRule’,’admins’);
或
Router::domain(‘admin.example.com’)
    ->style(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’,
        ’domain’ => ‘http://admin.example.com')
    ->rule(‘#NormalRouterRule’,array(‘admins’));
```

不仅可以为一个规则指定多个名称，还可以将多个规则指定为相同名称，形成命名组，执行时会根据命名组依次判断。

示例：

```
Router::domain(‘admin.example.com’)
    ->style(array(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’,
                  ’domain’ => ‘http://admin.example.com'))
    ->rule(‘#NormalRouterRule’,’admins’)
    ->rule(‘#NormalRouterRule’,’admins’);
```

注意：使用命名规则时必须指定样式的 domain 属性，否则生成的 URL 地址将没有域名的访问地址，更多信息请查看 IRouterStyle 接口。

使用命名规则

在 Router 中包含 RULE_NAME 参数，这个参数就是用于生成 URL 地址时指定规则名称。

示例：

```
# 访问：http://www.example.com/?controller=index&action=edit
Router::url(array(‘action’ => ‘edit’,’controller’ => ‘users’,
```

Router::RULE_NAME => ‘admins'));

结果：http://admin.example.com/?controller=users&action=edit

如果没有使用 RULE_NAME 执行结果为：

Router::url(array(‘action’ => ‘edit’,’controller’ => ‘users’));

http://www.example.com/?controller=users&action=edit

11.2.6、其它方法

Router::match 和 Router::defaultParams 通常外部不直接进行调用，都是由 Skihat 内部进行调用。

11.3、IRouterRule 路由规则

路由规则用于声明参数的内外转换，能够将外部请求参数与内部处理参数的双向转换。Skihat 中支持普通、正则和 Restful 三种路由规则，三种规则都继承自 AbstractRouterRule 类。

11.3.1、AbstractRouterRule 类

11.3.1.1、AbstractRouterRule 类声明

```
class AbstractRouterRule implement IRouterRule {
    # 常量声明
    const SYNTAX_NAME_VALIDATOR = '[a-z_]+’;              # 参数名称验证规则
    const NORMAL_NAME_VALIDATOR = '[a-z0-9_\-.]+’;       # 普通名称验证规则
    const NUMBER_VALIDATOR = '\d+’;                       # 数字验证规则
    # IRouterRule 接口方法
    public function parse(array $params = array());          # 参数转换
    public function reverse(array $params = array());        # 参数反转
    # 参数设置
    public __set($name,$value);                              # 设置参数
}
```

11.3.1.2、配置参数规则

在 AbstractRouterRule 类中，使用__set 方法设置参数的各种规则和限制，支持以下方式：

示例：

```
$rule = new NormalRouterRule();
$rule->参数名称 = ‘参数默认值’;
$rule->参数名称 = array(‘参数默认值’);
$rule->参数名称 = array(‘参数默认值’,’参数验证值’);
$rule->参数名称 = array(‘参数默认值’,’参数验证值’,’必选参数’);
```

注意：这里所说的参数，主要是指字符串请求参数，不包含 POST 提交参数。

路由规则使用 Router->rule 方法使用 Ioc 配置。

示例：

```
Router::domain(‘www.example.com’)
->rule(array(
        SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
        SKIHAT_PARAM_CONTROLLER => ‘index’,
        SKIHAT_PARAM_ACTION => array(‘index’)
));
```

11.3.1.3、默认值参数

默认参数值是指当指定参数不存在时，所使用的默认值。

示例：

```
Router::domain(‘www.example.com’)
->rule(array(
        SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
        ‘cat’ => 1,
        SKIHAT_PARAM_CONTROLLER => ‘users’)
));

# 请求：http://www.example.com/
请求参数集：array (‘controller’ => ‘users’,’cat’ => 1);
```

Skihat 预先设定有五个默认值：

- SKIHAT_PARAM_DEFAULT_MODULE：默认模块参数值，默认为 false
- SKIHAT_PARAM_DEFAULT_PACKAGE：默认包参数值，默认为 false
- SKIHAT_PARAM_DEFAULT_CONTROLLER：默认控制器名称值，默认为 index
- SKIHAT_PARAM_DEFAULT_ACTION：默认活动名称，默认为 index
- SKIHAT_PARAM_DEFAULT_FMT：默认数据格式参数，默认为 html

如果需要修改这些默认值，可以在路由规则中指定新的默认值替换原有值（请参考示例）。

11.3.1.4、验证参数规则

参数不仅仅允许有默认值，还允许使用正则表达式规范参数的数据格式：

示例：

```
Router::domain(‘www.example.com’)
->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
        SKIHAT_PARAM_CONTROLLER => ‘users’,
        SKIHAT_PARAM_ID => array(false,’\d+’))
->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’
        SKIHAT_CONTROLLER => ‘roles’,
        SKIHAT_PARAM_ID => array(false,’[a-z]+’)
));

# http://www.example.com/?id=abc
请求参数：array(‘id’ => ‘abc’,’controller’ => ‘roles’);
```

当路由规则判断参数值时，如果参数的值不满足条件规则验证失败，Router 会自动调用下一个规则进行判断，并返回判断结果。

为了保证 Skihat 的正确运行，Skihat 在路由规则中自动增加以下规则：

- SKIHAT_PARAM_CONTROLLER => self::SYNTAX_NAME_VALIDATOR,
- SKIHAT_PARAM_PACKAGE => self::NORMAL_NAME_VALIDATOR,
- SKIHAT_PARAM_MODULE => self::NORMAL_NAME_VALIDATOR,
- SKIHAT_PARAM_ACTION => self::SYNTAX_NAME_VALIDATOR,
- SKIHAT_PARAM_ID => self::NUMBER_VALIDATOR

如果需要修改默认规则，请使用新的配置值覆盖原有值。

11.3.1.5、参数必选

必选参数表示参数的值为必须值，即在判断时必须为非空值：

示例：

```
Router::domain(‘www.example.com’)
->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
        SKIHAT_PARAM_ACTION => array(‘approve’),
        SKIHAT_PARAM_CONTROLLER => array(false,false,true))
->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
        SKIHAT_PARAM_ACTION => ‘edit’));
```

请求：http://www.example.com/?id=12
参数集：array(‘id’ => 12,’action’ => ‘edit’,’controller’ => ‘index’)

请求：http://www.example.com/?controller=users&id=12
参数集：array(‘id’ => 12,’controller’ => ‘users’)

当路由规则判断参数值时，如果必须参数不存在规则验证失败，Router 会自动调用下一个规则进行判断，并返回判断结果。

11.3.2、NormalRouterRule 类

NormalRouterRule 类是最简单的路由规则，转换时 NormalRouterRule 类只会进行简单的转换，不会进行任何操作。

示例：

```
Router::domain(‘www.example.com’)
->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’));
```

根据以上配置，执行以下操作

http://www.example.com/?controller=users&action=edit&page=1
参数：array(‘controller’ => ‘users’,’action’ => ‘edit’,’page’ => 1)

Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’,’page’ => 1));
http://www.example.om/?controller=users&action=edit&page=1

11.3.2.1、参数默认值

如果在 NormalRouterRule 中声明默认参数，将产生以下行为：

- 请求参数：如果参数中不包含指定的默认参数，则使用默认参数的值，否则使用请求的参数值；
- URL 生成：如果指定的参数值与默认参数相同，则不生成指定的请求参数；

示例：

```
Router::domain(‘www.example.com’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#NormalRouterRule’,
                SKIHAT_PARAM_CONTROLLER => ‘users’));

# 参数同下
http://www.example.com/

# Url 同上
Router::url(array(‘controller’ => ‘users’))
```

参数验证规则和参数必选与 AbstractRouterRule 中介绍的方法完全相同。

11.3.3、RegexRouterRule 类

正则路由规则类，使用表达式来解析和生成路由参数，通常与 Web 服务器提供的重写功能配合使用。

11.3.3.1、使用 RegexRouterRule->express 声明表达式

- 定义：public $express = false

$express 成员变量用于提供判断所需要正则表达式，默认值为 false。

示例：

```
Router::domain(‘www.example.com’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
                    ‘express’ => ‘:controller/:action’
    ));
```

在 express 中使用"分号+参数名称"的方式指定参数名称，参数名称使用[a-z_]作为验证规范。

11.3.3.2、使用 Web 重写规范，生成友好 URL 地址

正则表达式使用参数 url 为解析的基础，通过 express 表达式从 url 参数中解析出完整的参数值。

示例：

```
# 参数集合：array(‘controller’ => ‘users’,’action’ => ‘edit’)
http://www.example.com/?url=users/edit

# url 地址：/?url=users/edit
Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’));
```

虽然直接使用 RegexRouterRule 解决了参数表达式的问题，但依然不是友好 URL 地址，生成友好 URL 地址必须与 Web 服务器提供的重写功能，将请求的地址转换为 url 参数。

示例：

```
# app/publics/.htaccess (Apache 重写规范)
<IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]
</IfModule>
```

为了能够生成正确的 URL，还需要使用 RewriteRegexStyle 样式指定为重写样式。

示例：

```
Router::domain(‘www.example.com’)
    ->style(‘#RewriteRouterStyle’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
                ‘express’ => ‘:controller/:action’
    ));
```

正确设置后，就可以通过友好 URL 访问应用。

示例：

```
# 参数集合同下：
http://www.example.com/users/edit

# URL 地址同上：
Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’));
```

11.3.3.3、使用可选参数

可选参数是指定当请求的 URL 地址不完整时，依然可以转换出正确的参数设置，在 express 表达式中使用中括号表示：

示例：

```
Router::domain(‘www.example.com’)
    ->style(‘#RewriteRouterStyle’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
                ‘express’ => ‘:controller/:action[.:fmt],
                ‘fmt’ => ‘html’));
```

正确设置可选参数后，就可以通过更短的 URL 地址来访问应用。

示例：

```
# 参数集合 ：array(‘controller’ => ‘users’,’action’ => ‘edit’)
http://www.example.com/users/edit
```

参数集合：array(‘controller’ => ‘users’,’action’ => ‘edit’,’fmt’ => ‘json’)
http://www.example.com/users/edit.json

反向生成 URL 地址时，如果可选参数的内部值与默认值完全相同，将生成最简单地址。

示例：

```
# /users/edit
Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’,’fmt’ => ‘html’))

# /users/edit.json
Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’,’fmt’ => ‘json’))
```

注意：中括号表示的部分为可选部分，可选部分内部同样允许声明参数，并且内部参数通常都需要声明默认值。

指定多个可选参数

在同一个 express 表达式中，还允许声明多个可选参数：

示例：

```
Router::domain(‘www.example.com’)
    ->style(‘#RewriteRouterStyle’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
                ‘express’ => ‘:controller/[:action][.:fmt],
                ‘action’ => ‘index’,’fmt’ => ‘html’));
```

访问结果

```
#   array(‘controller’=> ‘users’,’action’ => ‘index’,’fmt’ = > ‘html’)
http://www.example.com/users/

# array(‘controller’ =>’users’,’action’ => ‘edit’,’fmt’ => ‘html’)
http://www.example.com/users/edit

# array(‘controller’ => ‘users’,’action’ => ‘edit’,’fmt’ => ‘json’)
```

http://www.example.com/users/edit.json

反向生成 URL 时，如果可选参数的值与默认参数相同，则会生成最短路径，否则生成完整路径。

示例：

```
# http://www.example.com/users/
Router::url(array(‘controller’ => ‘users’,’action’ => ‘index’,’fmt’ => ‘html’))

# http://www.example.com/users/edit.html
Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’,’fmt’ => ‘html’));

# http://www.example.com/users/edit.json
Router::url(array(‘controller’ => ‘users’,’action’ => ‘index’,’fmt’ => ‘json’));
```

11.3.3.4、使用 more 参数

虽然在 express 中能够指定固定的参数，但是在很多时候可能会使用更多的参数，这时 RegexRouterRule 默认会转换为普通请求参数：

示例：

```
# 参数同下
http://www.example.com/users/edit?name=skihat

# URL 同上
Router::rule(array(‘controller’ => ‘users’,’action’ => ‘edit’,’name’ => ‘skihat’));
```

这种情况，可以使用 RegexRouterRule 提供的一个 more 参数：

示例：

```
Router::domain(‘www.example.com’)->style(‘#RewriteRouterStyle’)
    ->rule(array(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
                ‘express’ => ‘:controller/:action[-:more][.:fmt]));
```

使用 more 参数后，在 more 参数内部使用"-"分隔参数的名称和值。

示例：

```
 # 参数值同下
http://www.example.com/users/edit-name-skihat.html

# URL 地址同上
Router::url(array(‘controller’ => ‘users’,’ action’ => ‘edit’,’ name’ => ‘skihat’));
```

11.3.3.5、配置域名首页参数

当配置主页时，可以直接指定 express = /，当访问根目录时，直接使用/指定的配置信息：

示例：

```
Router::domain(‘www.example.com’)->style(‘#RewriteRouterStyle’)
    ->style(SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
        ’express’ => ‘/ ‘,’
        controller’ => array(‘index’,’ index),’action’ => array(‘index’,’ index’));
```

注意：为了防止生成 URL 地址时发生错误，主页的参数必须指定参数验证值。

11.3.3.6、参数默认值、参数验证值和必填参数

参数默认值、验证值和必须值与 AbstractRouterRule 中方法相同。

11.3.4、RestfulRouterRule 类

还未实现

11.4、IRouterRule 路由样式

路由样式解决如何将内部参数转换为 URL 地址，在 Skihat 中支持 NormalRouterStyle、RewriteRouterStyle 和 PhpinfoRouterStyle 三种 URL 样式。三个类都继承自 AbstractRouterStyle 类。

11.4.1、AbstractRouterStyle 类

11.4.1.1、AbstractRouterStyle 类声明

```
class AbstractRouterStyle implement IRouterStyle {
    public $domain = false;                                  # 完整域名地址
    public function fullUrl(array $params = array());        # 生成完整 URL 地址
}
```

11.4.1.2、AbstractRouterStyle->domain 成员变量

默认设置下，路由样式生成的 URL 地址将不带域名信息，但如果指定 domain 变量将生成完整的域名信息。

示例：

```
Router::domain(‘www.example.com’)
    ->style(‘#NormalRouterStyle’)
    ->rule(‘#NormalRouterRule’);
Router::url(array(‘action’ => ‘edit’));      # /?action=edit

# 指定包含 domain 变量的值
Router::domain(‘www.example.com’)
    ->style(array(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’,
                  ’domain’ => ‘http://www.example.com'))
    ->rule(‘#NormalRouterRule’);
Router::url(array(‘action’ => ‘edit’));      # http://www.example.com/?action=edit
```

11.4.2、NormalRouterStyle 类

前面的例子中都是使用 NormalRouterStyle 类作为示例，这是因为 NormalRouterStyle 类最简单，NormalRouterStyle 类声明如下：

```
class NormalRouterStyle extends AbstractRouterStyle {
    public function $script = self::DEFAULT_SCRIPT;     # index.php
}
```

script 成员变量用于指定脚本文件的名称，默认值为 index.php，如果需要指定新的执行脚本名称可以重新指定值。

示例：

```
Router::domain(‘www.example.com’)
    ->style(array(SKIHAT_IOC_CLASS => ‘#NormalRouterStyle’,
                  ’script’ => ‘admin.php’))
    ->rule(‘#NormalRouterRule’);
Router::url(array(‘action’ => ‘edit’));          # /admin.php?action=edit
```

11.4.3、RewriteRouterStyle 类

RewriteRouterStyle 类通常与 RegexRouterRule 类配合使用，用于生成满足重写规范的 URL 地址，示例请查看 RegexRouterRule。

11.4.4、PhpinfoRouterStyle 类

PhpinfoRouterStyle 类用于生成满足 Phpinfo 要求的地址信息，通常用于不支持重写的 Web 服务器，PhpinfoRouterStyle 类声明如下：

```
class PhpinfoRouterStyle extends AbstractRouterStyle {
    public function $script = self::DEFAULT_SCRIPT;          # index.php
}
```

同 NormalRouterStyle 类一样，script 用于指定根目录的脚本文件，使用方式也同 NormalRouterStyle 类一致。

示例：

```
Router::domain(‘www.example.com’)
    ->style(‘#PhpinfoRouterStyle’)
    ->rule(array(
        SKIHAT_IOC_CLASS => ‘#RegexRouterRule’,
        ’express’ => ‘:controller/:action’)
```

);

Router::url(array(‘controller’ => ‘users’,’action’ => ‘edit’)); # /index.php/users/edit

11.5、app/boots/routing.inc 路由配置文件

app/boots/routing.inc 文件用于配置路由的相关信息，默认配置如下：

示例：

```
Router::domain(SKIHAT_CURRENT_DOMAIN)
    ->style(‘#NormalRouterStyle’)
    ->rule(‘#NormalRouterRule’);
```

默认配置中，为了保存各种 Web 服务器的正常运行使用了最简单的方式，更多配置查看 Skihat 应用部署。

第四部分：内核服务组件

第 12 章：缓存组件

12.1、缓存接口声明

```
abstract class Cache {
    # 常量声明
    const DEFAULT_EXPIRE = 1800;                # 默认过期时间 1800 秒
    const DEFAULT_ENGINE = ‘default’;           # 默认缓存引擎
    const RUNTIME_ENGINE = ‘runtime’;           # 运行时缓存引擎
    const CONFIG_NAME = ‘kernels/caches’;       # 配置名称

    # 缓存接口
    public static function initialize();   # 初始化节点
    public static function engines();      # 返回全部引擎名称
    public static function engine($name); # 根据$name 返回引擎

    # 缓存方法
    public static function write($name,$value,$expire = self::DEFAULT_EXPIRE,
                                 $engine = self::DEFAULT_ENGINE);
    public static function read($name,$engine = self::DEFAULT_ENGINE);
    public static function delete($name,$engine = self::DEFAULT_ENGINE);
    public static function flush($name,$engine = self::DEFAULT_ENGIEN);
}
```

在缓存组件中允许同时使用多种不同类型的缓存，这些缓存使用统一的缓存引擎接口进行封装。

```
interface ICacheEngine {
    function write($name,$value);      # 写入缓存值
    function read($name);              # 读取缓存值
    function delete($name);            # 删除缓存值
    function flush();                  # 清空缓存值
```

```
}
```

12.2、引用缓存组件

缓存组件不需手动调用，当 SKIHAT_DEBUG 为大于 1 时就会自动启用缓存组件，同时可以使用以下方法判断当前是否支持缓存。

- enable_cached()

如果当前环境支持缓存则调用缓存组件，否则就不调用。

示例：

```
if (enable_cached()) {
    Cache::write(‘xxx’,’xxx’,’fff’);
}
```

12.3、Cache 缓存类

Cache 缓存类是 Cache 组件的服务接口类，提供缓存的外部服务接口。

12.3.1、使用 Cache 类配置

Cache 类配置非常简单，使用 Skihat::write 方法写入 kernels/caches 节点 ioc 配置值。

示例：

```
# app/boots/config.inc
Skihat::write(‘kernels/caches’,array(
    ‘default’ => ‘kernels.caches.engines.file_cache_engine',
    ‘runtime’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.caches.engines.memcache_engine’)
));
```

这里的 default 和 runtime 都需要配置 ICacheEngine 接口类，而这里的 default 和 runtime 则是指定引擎的名称，在 Cache 类的缓存方法组中的 engine 就是指这里的 default 和 runtime

名称。

注意：在 Skihat 中必须指定两个缓存引擎的一个是 default 一个是 runtime，runtime 由 Skihat 框架内部使用。

12.3.2、Cache::write 写入缓存值

- 定义：public function write($name,$value,$expire = self::DEFAULT_EXPIRE,
  $engine = self::DEFAULT_ENGINE)

根据$name，$value，$engine 和$expire 参数写入缓存信息。

示例：

```
 # 写入默认缓存
Cache::write(‘SQL’,’SELECT * FROM roles’);

#  写入默认缓存，有效期 1 个小时
Cache::write(‘SQL’,’SELECT * FROM roles’,3600);

# 写入名为 runtime 的缓存引擎，有效期 1 个小时
Cache::write(‘SQL’,’SELECT * FROM groups’,3600,’runtime’);
```

12.3.2、Cache::read 读取缓存值

- 定义：public function read($name,$engine = self::DEFAULT_ENGIEN);

根据$name 和$engine 读取缓存值，如果读取失败返回 false。

示例：

```
Cache::read(‘SQL’);                 # SELECT * FROM roles
Cache::read(‘SQL’,’runtime’);   # SELECT * FROM groups;
Cache::read(‘not-exists’);          # false
```

12.3.3、Cache::delete 删除缓存值

- 定义：public function delete($name,$engine = self::DEFAULT_ENGINE)

根据$name 和$engine 删除缓存的值。

示例：

```
Cache::delete(‘SQL’);                  # 删除默认引擎的 SQL
Cache::delete(‘SQL’,’runtime’); # 删除运行时缓存 SQL
```

12.3.4、Cache::flush 清空缓存值

- 定义：public function flush($engine = self::DEFAULT_ENGINE)

根据$engine 清空缓存内容。

示例：

```
Cache::flush();                        # 清空默认
Cache::flush(‘runtime’);               # 清空 runtime 缓存内容
```

12.3.5、其它缓存方法

12.3.5.1、Cache::engines

- 定义：public function engines()

返回全部的缓存引擎。

示例：

```
Cache::engines();  array(‘default’,’runtime’)
```

12.3.5.2、Cache::engine 方法

- 定义：public function engine($name)

根据$name 返回引擎 ICacheEngine 实例。

示例：

```
$engine = Cache::engine(‘default’);
```

12.4、缓存引擎类

12.4.1、FileCacheEngine 文件缓存引擎类

文件缓存引擎是通过将缓存信息存储到文件中，读取时直接读取文件的内容来提供缓存服务。

FileCacheEngine 类声明

```
class FileCacheEngine implement ICacheEngine {
    const FILE_EXT = ‘.tmp’;
    public $prefix = ‘default-cache’;  # 文件前缀名称

    # 缓存文件存放路径，默认为 data/caches/
    public $paths = SKIHAT_PATH_DATA_CACHES;

    # ICacheEngine 接口方法
    public function write($name,$value,$expire = Cache::DEFAULT_EXPIRE);
    public function read($name);
    public function delete($name);
    public function flush();
}
```

使用 FileCacheEngine 类

通常我们不直接使用 FileCacheEngine 类，而是将 FileCacheEngine 配置到 Cache 类，再通过 Cache 类提供的接口进行访问。

示例：

```
# app/boots/config.inc
Skihat::write(‘kernels/caches’,array(
    ‘default’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.caches.engines.file_cache_engine’,
```

```
                SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
                ‘paths’ => SKIHAT_PATH_DATA_CACHES . ‘/default’,
                ‘prefix’ => ‘def-cache’)
));
```

使用配置的实例名称：Cache::write(’SQL’,’SELECT * FROM roles’,3600,’default’)。

12.4.2、MemcacheEngine ： Memcache 缓存

使用 Memcache 服务提供对缓存的支持，因此 PHP 环境必须支持 Memcache 扩展。

MemcacheEngine 类声明

```
class MemcacheEngine implement ICacheEngine {
    # Memcache 配置成员
    public $host = ‘127.0.0.1’;              # memcache 服务器地址
    public $port = 11211;                    # memcache 服务端口
    public $flag = MEMCACHE_COMPRESSED;

    public function connect();               # 实始化
    public function write($name,$value);    # 写入缓存
    public function read($name);             # 读取缓存
    public function delete($name);           # 删除缓存
    public function flush();                 # 清空缓存内容
}
```

使用 MemcacheEngine 类

通常我们不直接使用 MemcacheEngine 类，而是通过 Cache 类使用，使用之间必须先进行配置。

示例：

```
Skihat::write(‘kernels/caches’,array(
    ‘default’ => array(
            SKIHAT_IOC_CLASS => ‘kernels.caches.engines.memcache_engine’,
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
            ‘host’ => ‘192.168.1.199’,
```

```
            'initial' => 'connect' # 必须将 initial 配置为 connect 方法。
    )));
```

使用 Cache::write('SQL','SELECT * FROM roles',3600,'default') 写入配置值。

12.5、ApcCacheEngine APC 缓存

使用 APC 缓存服务接口提供缓存支持，因此必须安装 APC 扩展组件时才能使用 ApcCacheEngine 类。

ApcCacheEngine 类声明

```
class ApcCacheEngine implement ICacheEngine {
    public function write($name,$value);    # 写入缓存
    public function read($name);            # 读取缓存
    public function delete($name);          # 删除缓存
    public function flush();                # 清空缓存内容
}
```

使用 ApcCacheEngine 类

通常我们不直接使用 ApcCacheEngine 类，而是通过 Cache 类使用，使用之间必须先进行配置。

示例：

```
Skihat::write('kernels/caches',array(
    'default' => array(
            SKIHAT_IOC_CLASS => 'kernels.caches.engines.apc_cache_engine',
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY
)));
```

使用 Cache::write('SQL','SELECT * FROM roles',3600,'default') 写入配置值。

12.6、XCacheEngine Xcache 缓存支持

使用 Xcache 缓存服务接口提供缓存支持，因此必须安装 Xcache 扩展组件时才能使用 XcacheEngine 类。

XcacheEngine 类声明

```
class XcacheEngine implement ICacheEngine {
    public function write($name,$value);        # 写入缓存
    public function read($name);                # 读取缓存
    public function delete($name);              # 删除缓存
    public function flush();                    # 清空缓存内容
}
```

使用 XcacheEngine 类

通常我们不直接使用 XcacheEngine 类，而是通过 Cache 类使用，使用之间必须先进行配置。

示例：

```
Skihat::write(‘kernels/caches’,array(
        ‘default’ => array(
                SKIHAT_IOC_CLASS => ‘kernels.caches.engines.xcache_engine’,
                SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY
)));
```

使用 Cache::write(‘SQL’,’SELECT * FROM roles’,3600,’default’) 写入配置值。

第 13 章：Loggers 日志组件

日志组件提供日志的记录服务，使用 ILoggerEngine 接口支持在同一应用中提供多种日志记录方式。

13.1、Loggers 组件声明

```
abstract class Logger {
    const CONFIG_NAME = ‘kernels/loggers’;          # 配置节点
    const DEFAULT_ENGINE = ‘default’;               # 默认引擎
    const DEFAULT_USER = ‘admin’;                   # 默认用户

    public static function initialize();            # 自动调用，初始化组件服务
    public static function engines();               # 返回全部引擎名称集
    public static function engine($name);           # 根据引擎名称返回引擎类

    # 记录日志信息
    public static function write($title,$message,$user = self::DEFAUT_USER,
                                 $engine = self::DEFAULT_ENGINE);
}

interface ILoggerEngine {
    # 记录日志内容
    function write($title,$message,$user = Logger ::DEFAULT_USER);
}
```

13.2、配置和使用日志组件

使用日志组件，必须使用 Skihat::write 配置日志引擎。

示例：

```
Skihat::write(‘kernels/loggers’,array(
```

‘default’ => ‘kernels.loggers.engines.file_logger_engine’
));

同时还必须在使用之前引用日志组件。

示例：

Skihat::import(‘kernels.loggers.logger’,SKIHAT_PATH_LIBRARY);

13.3、Logger 日志接口类

Logger 类是日志组件的外部接口类，提供对外的日志组件访问服务。

13.3.1、Logger::write 记录日志

- 定义：public static function write($title,$message,$user = self::DEFAULT_USER,
  $engine = self::DEFAULT_ENGINE)

  根据$title、$message、$user 和$engine 记录日志信息。

示例：

```
public function editAction() {
    Skihat::import(‘kernels.loggers.logger’,SKIHAT_PATH_LIBRARY);
    # do something

    Logger::write(‘更新会员内容’);      # 记录默认的日志信息
}
```

如果需要将日志记录到不同的引擎，请指定$engine 参数名称。

13.3.2、Logger::engines 返回全部引擎

- 定义：public static function engines()

  返回配置声明中的所有引擎集合。

示例：

Logger::engines();　　# array(‘default’);

13.3.3、Logger::engine 获取日志引擎实例

- 定义：public static function engine($name)

根据$name 返回日志引擎实例。

示例：

Logger::engine(‘default’);　　# FileLoggerEngine

13.4、FileLoggerEngine 文件日志引擎

文件日志引擎提供将日志信息，记录到文件的服务，使用时允许指定记录到不同的文件和设置记录的模板。

13.4.1、FileLoggerEngine 类声明

```
class FileLoggerEngine implement ILoggerEngine {
    public function $template = self::DEFAULT_TEMPLATE;      # 记录模板
    public function $fileName = self::DEFAULT_FILE;           # 文件名称 run.log

    # 执行记录
    public function write($title,$message,$user = Logger::DEFAULT_USER);
}
```

13.4.2、使用 FileLoggerEngine 类

通常我们不直接使用 FileLoggerEngine 类，而是通过 Logger 的配置间接访问：

示例：

Skihat::write(‘kernels/loggers’,array(

```
‘default’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.loggers.engines.file_logger_engine’,
        SKIHAT_IOC_PATH => SKIHAT_PATH_LIBRARY,
        ‘fileName’ => ‘rtime.log’ # 配置文件名称
)));
```

直接使用 Logger::write 方法会将内容记录到 data/loggers/rtime.log 文件中。

13.4.3、日志文件

为了保证安全，FileLoggerEngine 日志文件都存放在 SKIHAT_PATH_DATA_LOGGERS 目录下，因此指定的文件不应当包含目录名称。

13.4.4、日志模板

日志模板表示记录日志内容的记录形式，在日志模板中支持以下特殊标识：

- {:title}：日志标题；
- {:user}：日志用户名称；
- {:message}：日志消息；
- {:datetime}：日志的发生时间；

示例：

```
$template = "--------------------------------\r\n “ +
        “title:{:title}\r\n message:{:message}\r\n” +
        “user:{:user}\r\n datetime:{:datetime}"
```

注意：如果需要换行的话，必须使用双引号引用字符串。

第 14 章：Media 资源组件

Media 组件用于浏览和上传网站的资源文件，在一个应用中允许指定多个 IMediaEngine 实例，分别对应不同的资源。

14.1、Media 组件接口声明

Media 组件由 Media 类和 IMediaEngine 组成，Media 提供外部访问接口，IMediaEngine 接口提供实际的处理。

14.1.1、Media 类声明

```
abstract class Media {
    # 常量配置
    const DEFAULT_ENGINE = 'default’;
    const CONFIG_NAME = 'kernels/medias’;

    # Media 组件方法
    public static function initialize()
    public static function engines() ;
    public static function engine($name) ;

    # Media 组件接口方法
    public static function files($path = false, $sort = false,
                                $engine = self::DEFAULT_ENGINE);
    public static function upload($file, $path = false, $engine = self::DEFAULT_ENGINE);
}
```

14.1.2、IMediaEngine 接口

```
interface IMediaEngine {
    # 接口常量，表示文件的元数据信息
    const SCHEMA_FILE_NAME = 'file_name’;  # 文件名称
    const SCHEMA_FILE_SIZE = 'file_size’;      # 文件大小
```

```
        const SCHEMA_FILE_TYPE = 'file_type’;      # 文件类型
        const SCHEMA_FILE_PATH = 'dir_path’;       # 文件路径
        const SCHEMA_IS_DIR = 'is_dir’;            # 是否为目录
        const SCHEMA_HAS_FILE = 'has_file’;        # 是否包含文件
        const SCHEMA_IS_PHOTO = 'is_photo’;        # 是否为图片
        const SCHEMA_DATE_TIME = 'datetime’;       # 最后更新时间

        # 接口方法
        function files($path = '', $sort = 'name');
        function upload($file, $path = '');
    }
```

14.2、Media 组件接口类

负责 Media 组件的外部访问接口，提供外部访问所需的服务。

14.2.1、Media 组件配置

使用 Media 组件，必须使用 Skihat::write 配置资源引擎。

示例：

```
Skihat::write(‘kernels/medias’,array(
        ‘default’  => array(
            SKIHAT_IOC_CLASS =>  ‘kernels.medias.engines.file_media_engine’,
            SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY)
));
```

在使用 Media 组件前还必须使用 Skihat::import 方法将组件导入当前环境。

示例：

```
Skihat::import(‘kernels.medias.engines.file_media_engine’,SKIHAT_PATH_LIBRARY);
```

14.2.2、Media::files 返回文件列表

- 定义：public static function files($path = false,$sort = false,$engine =

self::DEFAULT_ENGINE);

根据$path、$sort 和$engine 返回文件列表数组，$path 表示文件的相对路径，$sort 表示排序方式，$engine 指定使用的资源引擎。

示例：

Media::files();

14.2.3、Media::upload 上传文件

- 定义：public static function upload($files,$path = false,$engine = self::DEFAULT_ENGINE);

根据$files、$path 和$engine 上传文件，$files 表示上传文件的信息，$path 表示文件存储的相对路径、$engine 表示使用的资源引擎，上传完成后返回文件的访问地址。

示例：

```
public function editAction() {
    $photoUrl = Media::upload($this->file('photo'));
}
```

注意：$files 参数支持单个文件传或多个文件上传。

14.2.4、Media::engines 返回资源引擎集合

- 定义：public static function engines()

根据当前配置返回所有的资源引擎集合。

示例：

```
Media::engines();  # return array('default');
```

14.2.5、Media::engine 返回资源引擎实例

- 定义：public static function engine($name)

根据$name 返回资源引擎实例。

示例：

```
Media::engine(‘default’);        # FileMediaEngine
```

14.3、FileMediaEngine 类

FileMediaEngine 类，支持将服务器的任意目录作为资源管理目录。

14.3.1、FileMediaEngine 类声明

```
class FileMediaEngine implement IMediaEngine {
    const DEFAULT_HOST_URL   =  ‘/uploads’;      # 默认主机 URL 地址
    const DEFAULT_MAX_SIZE = 20000;              # 默认上传大小

    # 允许上传的文件类型
    public $allowUploadTypes = array('png' => 'image/png’,'
                                    gif' => 'image/gif','jpg' => 'image/jpeg’);
    # 允许上传的文件大小
    public $allowUploadSize = self::DEFAULT_MAX_SIZE;
    # 允许查看的文件类型
    public $allowViews = array(‘png’,’gif’,’jpg’);
    # 主机管理的路径
    public $hostPath = SKIHAT_PATH_APP_PUBLICS_UPLOADS;
    # 默认的主机 URL 地址。
    public $hostUrl     = self::DEFAULT_HOST_URL;

    # IMediaEngine 接口方法
    public function files($path =  ‘’,$sort =  ‘name’);     # 获取文件列表
    public function upload($file,$path =  ‘’);              # 上传文件$file
}
```

14.3.2、FileMediaEngine 类使用

通常不直接使用 FileMediaEngine 类，而是通过使用 Media 类的组件接口方法，首先将

FileMediaEngine 配置到 Media 类。

示例：

```
Skihat::write(‘kernels/medias’,array(
    ‘default’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.medias.engines.file_media_engine’,
        SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
        ‘hostPath’ => ‘/home/www/uploads.example.com/',
        ‘hostUrl’ => ‘http://www.uploads.example.com')
));
```

注意：这里的 hostUrl 必须配置为 Web 主机，如果不单独配置为主机则可以使用相对路径。

第 15 章：Message 消息组件

略

第 16 章：Transactions 事务组件

Transaction 负责进行统一的事务处理，虽然也可以直接使用数据库提供的事务处理机制，但使用 Trsnaction 接口允许将多个支持事务的组件统一的提交、回滚操作。

16.1、Transactions 组件声明

Trsnactions 组件由 Trsnaction 类和 ITransactionObserver 接口组成。

16.1.1、Transaction 类

```
abstract class Transaction {
    # 观察者管理方法
    public static function register(ITransactionObserver $observer);
```

```
        public static function unregister(ITransactionObserver $observer);
        public static function hasRegister(ITransactionObserver $observer);
        # 事务管理方法
        public static function begin();             # 开始事务
        public static function comit();             # 提交事务
        public static function rollback();          # 回滚事务
        public static function inTransaction();   # 返回布尔值表示是否包含在事务中
    }

    interface ITransactionObserver {
        function begin();    # 开始事务
        function comit();    # 提交事务;
        function rollback(); # 回滚事务
    }
```

16.2、使用 Transaction 组件

使用 Transaction 组件非常简单，只需要调用事务管理方法执行相应的事务操作。

示例：

```
    class OrderService {
        public function submit($order,$items) {
            try {
                Transaction::begin();
                …
                Trsnaction::commit();
            }
            catch(Exception $ex) {
                Transaction::rollback();
            }
        }
    }
```

注意：Transaction 不支持嵌套事务。

16.3、ITransactionObserver 接口

Transactions 组件采用观察者模式，由 Transaction 管理 ITransactionObserver 对象的实例，

现在需要的时候执行相应的操作。

- begin：当开始事务时执行的方法；
- commit：当提交事务时执行的方法；
- rollback：当回滚事务时执行的方法；

ITransactionObserver 接口不需要我们手动调用，而是由 Transaction 类自动调用。

在 Skihat 中，只有数据库的 PdoEngine 类实现了 ITransactionObserver 接口，所以我们可以直接使用 Transaction 管理数据库的事务。

16.4、实现 ITransactionObserver 接口

实现 ITransactionObserver 接口非常简单，只需要实现三个对应的方法，并且在初始化和结束时执行相应的管理方法。

示例：

```
Skihat::import('kernels.transactions.transaction',SKIHAT_PATH_LIBRARY);
class MyTransactionObserver implements ITransactionObserver {
    public function _construct() {
        Transaction::register($this);  # 将自己注册到 Transaction 类
    }

    public function begin() {
        echo 'my transaction begin';
    }

    public function commit() {
        echo 'my transaction commit';
    }

    public function rollback() {
        echo 'my transaction bollback';
    }

    public function ~destruct() {
        Transaction::unregister($this);      # 将自己从 Transaction 中注销。
    }
}
```

第 17 章：Database 数据库组件

Database 数据库组件是用于提供外部数据库访问的接口，是 Skihat 中最难理解和最复杂的组件之一。

17.1、Database 组件声明

Database 组件由 IDataSupport、IDataEngine 和 Database 类组成，还包含 SqlBuilder 类生成需要执行的 SQL 代码。

17.1.1、Database 类

Database 类是 Databases 组件的接口类，主要负责创建数据库引擎实例。

```
abstract class Database {
    const CONFIG_NAME = ‘kernels/databases’;    # 配置节点名称
    const DEFAULT_ENGINE = ‘default’;           # 默认引擎

    public static function initialize();        # 初始化组件方法；
    public static function engine($name = self::DEFAULT_ENGINE);
    public static function engines();           # 返回全部引擎名称；
}
```

17.1.2、IDataSupport 接口

IDataSupport 接口提供数据库访问所需的支持，包含相关的数据库语法和查询结果的常量声明。

```
interface IDataSupport {
    # 语法常量
    const SYNTAX_FROM = ‘from’;       # 表格常量
    const SYNTAX_TABLE = ‘table’;     # 数据表常量
    const SYNTAX_TABLE_ALIAS = ‘alias’; # 数据表别名
    const SYNTAX_FIELDS = ‘fields’;     # 数据字段访问
    const SYNTAX_WHERE = ‘where’;       # 数据查询条件
```

```
    const SYNTAX_ORDER = 'order';          # 数据排序方式
    const SYNTAX_PAGE  = 'page';           # 数据分页信息
    const SYNTAX_LINK  = 'links';          # 数据关联关系
    const SYNTAX_JOIN  = 'joins';          # 数据表连接关系
    const SYNTAX_JOIN_ON   = 'on';         # 数据连接方式
    const SYNTAX_JOIN_TYPE  = 'joinType';   # 数据连接类型
    const SYNTA_GROUP  = 'group';          # 数据分组

    # 返回值常量
    const FETCH_STYLE_ASSOC = 2;   # 数组 key 返回值
    const FETCH_STYLE_NUM = 3;     # 数组索引返回值
    const FETCH_STYLE_BOTH = 4;    # 数组，包含 key 和索引
    const FETCH_STYLE_OBJ = 5;     # 对象返回值
    const FETCH_STYLE_GROUP = 65536;  # 分组返回值
    const FETCH_STYLE_MODEL = 1000;   # 模型返回值;
}
```

整个语法部分与 SQL 中的各个部分相对应，而返回值类型常量，则与 PDO 数据访问的返回值相对象。

17.1.3、IDatabaseEngine 接口

IDataEngine 接口是真正执行数据库访问的接口，继承自 IDataSupport 接口。

```
interface IDatabaseEngine extends IDataSupport {
    # Schema 常量
    const SCHEMA_FIELD = 'field';
    const SCHEMA_TYPE = 'type';
    const SCHEMA_SIZE  = 'size';
    const SCHEMA_NULL = 'nul';
    const SCHEMA_DEFAULT = 'default';
    const SCHEMA_FORMATTER = 'formatter';

    # 数据访问方法
    function create(array $fields,array $syntax);
    function createMultiple(array $fields,array $values,array $syntax);
    function update($fields,array $syntax);
    function delete(array $syntax);
    function fetchAll(array $syntax,$fetchStyle = self::FETCH_STYLE_ASSOC);
    function fetchLastID();
    function fetchSchema($table);
    # 命令方法
    function query($cmd,array $args = array(),$fetchStyle=self::FETCH_STYLE_ASSOC);
    function execute($cmd,array $args = array());
```

}

通过声明可以看出，对于数据库的任何操作都可以通过语法数组来完成。

示例：

```
$engine = Databsae::engine(‘default’);
$users = $engine->fetchAll(array(‘table’ => ‘users’)); # 返回 table 的所有数据。
```

17.1.4、SqlBuilder 类

SqlBuilder 类将语法和参数解析为可以执行的 SQL 命令的服务，在实际使用时我们必须从 SqlBuilder 继承自己的类提供特殊数据库的 SQL 命令。

```
abstract class SqlBuilder    implements IDataSupport {
    public function create(array $fields,array $syntax);
    public function createMultiple(4array $fields,array $values,array$sytnax);
    public function update($fields,array $syntax);
    public function delete(array $syntax);
    public function fetch(array $syntax,$options = array());
    public abstract function safeName($name);
}
```

通过声明可以看出，所有的方法基本都与 IDatabaseEngine 相对应。

17.2、使用 Database 组件

通常我们不直接使用 Database 组件所提供的服务，而是由模型内部调用。

17.3、使用 MySqlPdoEngine

目前在 Skihat 中只提供了针对于 MySql 访问的 MySqlPdoEngine 类。MySqlPdoEngine 类继承自 PdoEngine，PdoEngine 实现了所有的 IDatabsaeEngine 方法，并且提供了两个重要的配置参数：

- conf：访问连接配置；
- wconf：写入连接配置；

在 MySqlPdoEngine 中允许分别配置读取字符串，字符串的格式如下：

- mysql:host={host};dbname={dbname}&user={user}&pass={pass}&charset={charset}

这里的 host、dbname、user、pass 和 charset 分别指定：

- host：主机 ip 地址；
- dbname：数据库名称；
- user：用户名；
- pass：访问密码；
- charset：字符集，utf8；

示例：

```
Skihat::write (‘kernels/databases’,array (
    ‘default’ => array(
        SKIHAT_IOC_CLASS => ‘kernels.databases.engines.mysql_pdo_engine’,
        SKIHAT_IOC_PATHS => SKIHAT_PATH_LIBRARY,
        ‘conf’ =>
       ‘mysql:host=127.0.0.1;dbname=example&user=root&pass=123321&charset=utf8’
));
```

更多信息，请查看 Model 模型参考。

第 18 章：Security 安全组件

Security 负责应用的安全检查，由 Security 接口类，IAuthentication 和 IAuthorization 接口三个部分构成。

18.1、Security 组件接口声明

18.1.1、Security 接口类

Security 是安全组件的接口方法，外部通过 Security 直接访问安全的相关服务。

```
abstract class Security {
    const DEBUG_NAME = ‘kernels/securities’;      # 安全调试名称；
```

```
    const CONFIG_NAME = ‘kernels/securities’;     # 配置节点名称；
    const CONFIG_AUTHENTICATIONS = ‘authentications’;      # 认证节点配置；
    const CONFIG_AUTHORIZATIONS = ‘authorizations’;        # 授权节点配置；
    const DEFAULT_AUTHENTICATION = ‘default’;              # 默身份验证名称；
    const DEFAULT_AUTHORIZATION    = ‘default’;            # 默认授权名称；

    public static function initialize();              # 初始化方法，自动调用；
    # 身份验证方法，验证成功返回身份标识，失败返回 false。
    public static function authenticate($identity,
                $authName = self::DEFAULT_AUTHENTICATION);
    # 授权方法，返回授权结果。
    public static function authorizate($identity,$sro,$rule = IAuthorization::RULE_SUM,
                $authName = self::DEFAULT_AUTHORIZATION);
}
```

18.1.2、IAuthentication 接口

IAuthentication 接口提供用户的身份验证，可以根据自己的需要编写相关的实现。

```
class IAuthentication {
        function authenticate($identity);
}
```

IAuthentication 接口只有一个 authenticate 方法，通过用户的标识信息，判断用户是否验证成功，验证成功则返回系统的用户身份标识，否则返回 false 表示验证失败。

18.1.3、IAuthorization 接口

IAuthentication 接口提供用户的资源验证授权。

```
class IAuthorization {
        const RULE_SUM = ‘sum’;        # 合计规则；
        const RULE_MIN = ‘min’;        # 最小值规则；
        const RULE_MAX = ‘max’;        # 最大值规则；
        const RULE_RANGE = ‘range’;  # 包含、集合规则；

        function authorize($identity,$sro,$rule = self::RULE_SUM);
}
```

在 Skihat 中授权主要由三个基本组成部分：

- 身份：由 IAuthentication 验证返回的身份证明；
- SRO：Security Resource Object 安全资源管理对象，需要进行认证的项目；
- 规则：规则指定想要获取的授权值；

这其中最容易理解的是身份，不太好理解的是 SRO 和规则。

SRO

安全资源管理对象，在不同的权限设计体系中有不同的名称，例如：菜单项目、界面或者其它名称。Skihat 中广义的解释是程序中需要控制的资源。例如：需要对菜单项目进行权限控制，那么这个菜单就应当有一个对应的 SRO 标识；对财务审核的金额有权限上的控制，那么 SRO 就表示这个金额的控制标识。

规则

经常在设计权限时，不是权限的简单获取，而是需要一组权限进行相加、合并、取最大值、最小值等操作。例如：在一个典型的 RBAC（基于角色的访问控制）设计中，一个用户通常被允许有多个角色，每个角色设置有多个 SRO 对象，当获取用户的某一个 SRO 资源的控制时，经常就会有多个值，这就需要有一个规则来判断如何处理这些值的结果。

在 Skihat 中，将这一组规则简化为四种规则：

- 最小值：从全部 SRO 值中选取最小的值。例如：满足条件的最低分数；
- 最大值：从全部 SRO 值中选择最大的值。例如：允许审核的最大金额控制。
- 合计值：通常在 Skihat 中判断是否允许执行某一操作时使用合计值，这样的好处在于允许实现，否定优先原则。
- 合并值：将 SRO 的值，全部以数组形式返回。例如：一个编辑允许编辑多个栏目。

18.2、实现安全接口

在 Skihat 中没有标准的 Security 安全接口的实现，但如果需要实现自己的身份验证和权限控制也非常简单。

示例：

```
class MyAuthentication impletement IAuthentication {
    public function authentication($identity) {
```

```
            if ($identity[‘name’] == ‘skihat-team’ && $identity[‘password’] == ‘123321’)) {
                return array(‘id’ => 123321,‘name’ => ‘skihat-team’);
            }

            return false;
        }
    }

    class MyAuthorization implement IAuthorization {
        public function authorizate($identity,$sro,$rule = self::RULE_SUM) {
            if (!$identity) return false;

            if ($identity[‘id’] == 123321 && $sro == ‘audit’) {
                return 1
            }

            return -1;
        }
    }
```

在示例中非常简单，没用调用数据库和模型对象，在实际使用中需要根据自己的业务需求实现相关的处理。

18.3、Security 组件配设置

编写好自己的接口对象后，还不能直接使用，还需要先将相关的信息配置到应用中。

```
# app/boots/config.inc
    Skihat::write (‘kernels/securities’, array (
        ‘authentications’ => array (‘default’ => ‘#MyAuthentication’),
        ‘authorizations’ => array (‘default’ => ‘#MyAuthorization’)
    ));
```

如果有多个验证或授权实现接口，可以在 authentications 和 authorizations 节点中进行配设置。

配置完成后，就可以直接调用 Security 的相关方法执行处理。

```
$identity = Security::authenticate (array (‘name’=> ‘skihat-team’,’password’ => ‘123321’));
$allow = Security::authorizate ($identity,’audit’);
```

18.4、配合 SecurityFilter 类

通常使用安全组件时，还需要与 SecurityFilter 安全过滤器配合使用，SecurityFilter 会根据设置自己判断当前控制器是否允许匿名访问，更多信息请参考：SecurityFilter 类。

第 19 章：validators.inc 验证组件

验证组件提供对数据的验证服务，由 validate_validators 方法和一组验证器方法组成。

19.1、validate_validators 验证方法

- 定义：function validate_validators($input,array $validators,array $message= array());

根据$validators 和$messages 对$array 参数进行数据验证，并返回验证错误消息。$input 参数必须为数组或支持 ArrayAccess 接口的对象；

验证参数规则

$validators 参数支持以下规则：

- 方式一：直接指定验证器 array('name' => 'required','email' => 'email');
- 方式二：使用数组参数，指定多个验证器 array('name' => array('required' => true),'email' = array('email' => true));
- 方式三：使用数组参数，指定验证器参数 array('name' => array('strlen' => array(10,20),'required' => true));

错误消息规则

同时$message 参数使用以下规则：

- 方式一：指定字段的错误消息 array('name' => '名称不能为空，并且必须大于 10');
- 方式二：为每种验证方式分别设置错误消息 array('name' => array('required' => '名称不能为空.','strmin' => '名称长度必须大于 10'));

使用时只需要正确传递相关参数，并检查返回值。

示例：

```
$validators = array(‘name’ => array(
                    ‘required’ => true,’strmin’ => 6,’strmax’ => 16))
$messages = array(‘name’ => array(‘required’ => ‘名称不能为空',
                    ‘strmin’ => ‘名称必须大于 6 个字符’,
```

'strmax' => '名称必须大于 6 个字符'));

使用验证规则：

返回值 ：# array();

validate_validators(array('name' => 'skihat'),$validators,$messages);

返回值 ： array('name' => '名称必须大于 6 个字符');

validate_validators(array('name' => 'hello',$validators,$messages);

19.2、验证方法声明规范

验证方法非常简单，只需要遵循 camel 小写命名规范，并且使用_validator 作为函数后缀（例如：required_validator），同时必须按以下参数规则：

- xxxx_validator(&$input,$field) ：指定验证参数和验证字段；
- xxxx_validator(&$input,$field,$options)：指定验证参数、验证字段和选项值；

19.3、Skihat 提供的验证方法

19.3.1、required_validator 非空验证

- 定义：function numeric_validator(&$input, $field)

  非空值验证器，如果验证字段的值非空，则返回 true，否则返回 false。

19.3.2、numeric_validator 数值验证器

- 定义：function numeric_validator(&$input, $field)

  数值验证器，如果验证字段的值为字值，则返回 true，否则返回 false。

19.3.3、telephone_validator 电话号码验证器

- 定义：function telephone_validator(&$input, $field)

  电话号码验证器，如果验证字段值为电话号码，则返回 true，否则返回 false。

支持以下格式：

- ✧ 座机号码：xxx-xxxxxxxxx/xxxx-xxxxxxx；
- ✧ 手机号码：xxxxxxxxxxx；
- ✧ 400 号码：400-xxx-xxxxxx；

19.3.4、email_validator 邮件验证器

- 定义：function email_validator(&$input, $field)

  邮件验证器，如果验证字段值为邮箱则返回 true，否则返回 false。

19.3.5、url_validator URL 验证器

- 定义：function url_validator(&$input, $field)

  Url 验证器，如果验证字段的值为 URL 地址，则返回 true，否则返回 false。

19.3.6、ip_validator IP 验证器

- 定义：function ip_validator(&$input, $field)

  IP 验证器，如果验证字段的值为 IP 地址，则返回 true，否则返回 false。

19.3.6、strmin_validator 字符串最大长度验证器

- 定义：function strmin_validator(&$input, $field, $length = 10)

  字符串最小长度验证器，如果验证字段值长度大于或等于$length，则返回 true，否则返回 false。

19.3.7、strmax_validator 字符串最大长度验证器

- 定义： function strmax_validator(&$input, $field, $length = 32)

  字符串最大长度验证器，如果验证字段长度小于或等于$length，则返回 true，否则返回

false。

19.3.8、required_validator 非空验证

- 定义：function strlen_validator(&$input, $field, $range = array())

字符串长度范围验证器，如果验证字段长度在$range 指定的范围内，则返回 true，否则返回 false。指定$range 时，第一个值表示最小长度，第二个值表示最大长度。

示例：

strlen_validator($input,’name’,array(16,32));

19.3.9、nummax_validator 数值最大验证器

- 定义：function nummax_validator(&$input, $field, $max = 10000)

数值最大验证器，如果验证字段值小于或等于$max，则返回 true，否则返回 false。

19.3.10、nummin_validator 数值最小验证器

- 定义：function nummin_validator(&$input, $field, $min = 10)

数值最小验证器，如果验证字段值大于或等于$min，则返回 true，否则返回 true。

19.3.11、numrang_validator 数值范围验证器

- 定义：function numrang_validator(&$input, $field, $range = array())

数值范围验证器，如果验证字段值在$range 范围内，则返回 true，否则返回 false。指定$range 参数时，第一个值表示最小值，第二个表示最大值。

示例：

numrange_validator($input,’age’,array(18,60));

19.3.12、compare_validator 比较验证器

- 定义：function compare_validator(&$input, $field, $compare = '')

比较验证器，如果验证字段值与$compare 字段值相同，则返回 true，否则返回 false。

19.3.13、regex_validator 正则验证器

- 定义：function regex_validator(&$input, $field, $pattern = '*')

正则验证器，如果验证字段值符合正则$pattern，则返回 true，否则返回 false。

19.3.14、enum_validator 枚举验证器

- 定义：function enum_validator(&$input, $field, $options = array())

枚举验证器，如果验证字段值包含在$$options 中，则返回 true，否则返回 false。

19.3.15、自定义验证器

自定义验证器非常简单，只需要根据格式进行相关的声明和处理。

示例：

```
function abc_validator(&$input,$field) {
    if (!isset($input[$field])) return true;
    return $input[$field] == 'abc';
}
```

示例中，如果对象的字段值为 abc 则验证成功，否则返回验证失败。

第五部分：其它应用实践

第 20 章：Skihat 模块化开发

模块是 Skihat 组织应用和功能的一种方法，允许将完成特定功能和服务的代码织在一起形成模块。

20.1、模块目录

声明模块非常简单，只需要在 app/modules（SKIHAT_PATH_APP_MODULES）目录中，创建新的文件夹，文件夹的名称就是模块的名称。

示例：

app/modules/nuke：nuke 模块；

app/modules/archives：archives 模块；

注意：为了简化模块开发，不允许内部模块即模块中包含模块。

20.2、目录结构

在每一个模块内部允许允许声明以下目录：

- controllers：模块控制器目录；
- extends：扩展目录，允许提供对架构或其它目录的扩展；
- langs：模块语言包目录，允许声明不同的语言包；
- models：模块模型目录，允许声明模块内部的模型；
- services：模块提供的服务目录，允许声明模块的服务类，提供外部访问的接口；
- views：模块视图目录，提供模块的视图显示；

从整个目录来看，基本上就是一个 app 目录的缩小版本。

20.3、模块控制器

声明模块控制器，只需要在 controllers 目录下创建对应的控制器，规范与普通控制器一致。

示例：

```
# app/modules/nuke/index_controller.inc
class IndexController extends ApplicationController {
    public function indexAction() {
        $this->text(‘Hello Nuke’);
    }
}
```

访问模块控制器需要使用 SKIHAT_PARAM_MODULE 指定访问的模块：

- http://www.example.com/?module=nuke&controller=index

同普通控制器一样，在模块控制器内容也允许指定包。

示例：

```
# app/modules/nuke/admins/index_controller.inc
class IndexController extends ApplicationController {
    public function indexAction() {
        $this->text('Hello Nuke admins');
    }
}
```

访问地址：http://www.example.com/?module=nuke&package=admins&controller=index

20.4、模块模型

声明模块模型，只需要在模块对应的 models 目录下创建模型文件，规范与普通模型文件相同。

示例：

```
# app/modules/nuke/models/nuke_user.inc
class NukeUser extends ApplicationModel {
}
```

如果需要在控制器中引用模型，只需要在使用 SKIHAT_PATH_APP_MODULES 目录。

示例：

```
# app/modules/nuke/admins/index_controller.inc
public function actionModels() {
    parent::actionModels();
    # 加载自定义模型
    Skihat::import('nuke.models.nuke_user',SKIHAT_PATH_APP_MODULES);
}
```

如果模型关系中包含其它模块的模型，只需要使用完整路径。

示例：

```
# app/modules/archives/models/archive.inc
public static function __config() {
    return array(
        self::META_LINKS => array(
                    'admin'  => array(
                    SKIHAT_LINKS_CLASS =>  'nuke.models.nuke_user',
                    SKIHAT_LINKS_TYPE => self::BELONGS_TO,
                    SKIHAT_LINKS_FOREIGN => 'admin_user')
                )
        )
    );
}
```

注意：为了保证模型不冲突，Skihat 建议模块模型使用模块作为模型的前缀，例如：nuke_user.inc。

20.5、模块视图

20.5.1、创建模块视图模板

如果模块没有指定主题信息，则模块视图只允许声明在模块的 views 目录：

- app/modules/nuke/views # nuke 为模块名称
- app/modules/views/nuke # nuke 为模块名称

如果模块指定主题信息，则可以将模板声明在以下目录：

- app/themes/defaults/nuke # defaults 为主题名称、nuke 为模块名称
- app/themes/views/nuke # nuke 为模块名称
- app/modules/nuke/views # nuke 为模块名称

20.5.2、模块视图模板布局路径

模块视图模板的布局，允许出现在以下目录中：

- app/themes/defaults/__layouts（指定主题 defaults）
- app/views/__layouts　　# 应用模板布局
- app/modules/nuke/__layouts　# 其中 nuke 为模块名称

20.5.3、模块视图助手路径

模块允许使用以下目录的视图助手：

- app/themes/defaults/__helpers　　# default 为主题名称（声明主题）
- app/modules/nuke/views/__helpers　# nuke 为模块名称
- app/views/__helpers　　# 应用助手
- skihat/views/__helpers　　# Skihat 库视图助手

20.6、模块服务

在 Skihat 中，不建议将缓存、安全、日志等功能，直接在模型中使用，而是封装成模块服务。

示例：

```
# app/modules/nuke/services/nuke_service.inc
class NukeService {
    public function fetchUsers() {
        if (cache_enabled()) {
            $users = Cache::read(‘nuke-users’);
            if (!$users ) {
                $users = NukeUser::fetchAll()->fetchObjects();
                Cache::write(‘nuke-users’,$users);
            }
            return $users;
        }
        return NukeUser::fetchAll()->fetchObjects();
    }
```

```
}
```

服务不要求继承自任何类，同时建议服务以_service 文件后缀，文件名与类对应。使用服务类时，只需要在控制器方法中引用或使用 ioc 方法创建。

示例：

```
public function serviceAction() {
    $service = Skihat::ioc(‘nuke.services.nuke_service’,SKIHAT_PATH_MODULE);
    $this[‘user’] = $service->fetchUsers();
}
```

20.7、extends 目录

extends 目录用于声明对其它 Skihat 库或者其它模块的护展，例如：在 nuke 模块中开发一个日志程序，并且提供对 Skihat 库的扩展。

示例：

```
# app/modules/nuke/extends/nuke_logger_engine.inc
class NukeLoggerEngine implement ILoggerEngine {
    public function write($title,$message,$user = Logger::DEFAULT_USER) {
        Skihat::import(‘nuke.models.nuke_logger’,SKIHAT_PATH_APP_MODULES);
        NukeLogger::write($title,$message,$user);
    }
}
```

使用这些扩展，只需要修改配置文件：

示例：

```
# app/boots/config.inc
Skihat::write(‘kernels/loggers’,array(
    ‘default’ => array(
        SKIHAT_IOC_CLASS => ‘nuke/extends/nuke_logger_engine’,
        SKIHAT_IOC_PATH  => SKIHAT_PATH_APP_MODULES
    )
));
```

这样调用 Logger::write 时就将调用 NukeLoggerEngine 扩展引。

20.8、langs 目录

langs 目录用于保存模模块的语言信息，更多主参考 Skihat::i18n 国际化服务。

第 21 章：Skihat 错误处理

21.1、Skihat 的错误原则

在 Skihat 中，使用 trigger_error 和 throw 两者相结合的方式处理错误：

- trigger_error：如果不影响程序的继续执行使用 trigger_error 显示错误消息；
- throws：如果错误导致程序不能继续执行，则使用 throws 抛出异常信息；

21.1.2、trigger_error 函数

- 定义：function trigger_error($error_message,$error_types);

  根据$error_message 和$error_types 提示错误显示信息。

trigger_error 函数是由 PHP 提供的错误处理函数，PHP 会自动根据错误处理的配置，判断 PHP 的执行或中断，更多信息请参考 PHP 参考手册。

21.1.2、throws 抛出异常

如果错误会影响到程序的继续执行，Skihat 建议使用 throws 抛出异常，由开发人员自定义处理方式。throws 异常，请参考 PHP 语法。

21.2、Skihat 错误处理环境设置

21.2.1、SKIHAT_DEBUG 常量设置

在 boots/startup.inc 中，最先声明的就是 SKIHAT_DEBUG 常量，表示当前程序的调试信息，允许以下值：

- 0：关闭调试：运行模式，关闭调试信息；
- 1：启用缓存进行调试：运行调试模式，但开启缓存支持；
- 2：开启全部调试：关闭缓存，执行调试；

如果需要判断当前环境是否为调试模式，直接检查 SKIHAT_DEBUG 的值。

示例：

```
if (SKIHAT_DEBUG) {
    # do something
}
```

如果判断当前环境是否允许缓存，直接使用 cache_enable 方法。

- 定义：function cache_enable()

返回一个布尔值，表示当前环境是否允许缓存服务。

21.2.2、全局错误设置

在 app/boots/startup.inc 文件中，包含处理错误的全局设置。

```
ini_set ('log_errors', 'on');
ini_set ('error_log', SKIHAT_PATH_DATA_LOGGERS. '/errors.log');
 if (SKIHAT_DEBUG) {
    error_reporting (E_ALL);
    ini_set('display_errors', 'on');
} else {
    error_reporting(E_ERROR);
    ini_set('display_errors', 'on');
}
```

在全局错误设置中，默认开启错误日志记录，并将错误记录到 data/loggers/errors.log 文件中。如果需要重新设置错误的处理，可以直接修改 startup.inc 的代码。

21.3、全局错误处理

21.3.1、Skihat 的全局处理函数

Skihat 的全局错误处理函数是由 ApplicationBase 类提供，分别为：

- globalError：全局错误处理函数，当发生任何错误时将调用本方法；
- globalException：全局异常处理函数，当发生的异常没有处理时将会调用本方法；
- globalComplete：全局完成函数，当完成响应时将调用本方法；

三个方法都是实例方法。

虽然三个方法能够解决大部分的全局错误处理，但需要注意三个方法是在 ApplicationBase 的 intialize 方法中，分别调用各自的设置方法进行的绑定。因此，如果错误发生在 ApplicationBase::initialize 方法之前，则不能由这三个方法处理。

21.3.2、Skihat 的默认错误处理方式

当发生错误后，ApplicationBase 类会按以下方式进行处理：

- 将请求跳转到 ErrorController 类中，则 ErrorController 类处理。
- ErrorController 类将调用 Controller->error(500)方法。

在 views/__errors/500.stp 中，如果当前环境为调试环境，500 将会显示错误信息，并显示错误的详细信息。如果当前环境为运行环境，只会显示 500 错误。

21.3.3、自定义 Skihat 错误处理

自定义 Skihat 错误有两种方式：

- 自定义模板：如果只想错误提示更多人性化，可以直接编写 app/views/__errors/500.stp，编写自定义错误模板；
- 自定义错误控器：如果需要增强错误控制器，可以直接声明 app/controllers/error_controller.inc，提供自定义错误处理控制器；
- 自定义全局处理：如果需要自定义错误的全局处理，重写 app->globalXXX 方法。

具体的编写方式在控制器、模板的相关内容中都有说明。

21.4、Skihat 异常类

Skihat 框架以 SkihatException 异常类为基础，任何 Skihat 框架的异常都是继承自该异常。

SkihatException ：框架异常基础类，任何框架异常都继承自该异常。

- ConfigException：配置异常，当发生配置错误时引发本异常。
- TypeNotFoundException：类型未找到异常，如果类型未找到将引发本异常。
- FileNotFoundException：文件未找到异常，如果需要查找的文件不存在，引发本异常。
- CacheException：缓存异常，任何缓存发生错误，引发本异常。
- DatabaseException：数据库异常，如果数据库组件发生错误，引发本异常。
  - SyntaxException：语法错误，如果数据库解析语法时发生错误，引发本异常，继承自 DatabaseException。
- LoggerException：日志异常，如果日志发生错误，引发本异常。
- MediaException：资源异常，如果资源发生错误，引发本异常。
- MessageException：消息异常，如果通知消息发生错误，引发异常。
- SecurityException：安全异常，如果发生安全错误，引发安全异常。
- TransactionException：事务异常，如果发生事务错误，引发异常。
- ModelException：模型异常，如果模型发生错误，引发模型异常。
  - ModelNotFoundException：如果查找模型发生错误，引发本异常，继承自 ModelException。
- ViewException：视图异常，如果视图发生错误，引发本异常。
- RouterException：路由异常，如果发生路由错误，引发本异常。

如果需要声明新的异常类，请继承自 SkihatException 类，或具体的子类。

第 22 章：Skihat 安全建议

任何 Web 应用程序都不能保证绝对的安全，安全问题已经是 Web 开发过程中面临的最大问题之一，Web 安全来自以下三个方面：

- 操作系统安全
- Web 服务器安全
- Web 应用安全

在这三个方面的安全层次中，越靠近底层相对更加安全，而发生安全问题结果也越严重。

虽然操作系统和 Web 服务器也存在各种安全问题，但通常最容易发生安全问题的是 Web 应用，这是因为 Web 应用通常由不同的公司，不发的开发人员开发，很多开发人员对于 Web 安全没有基本的认识，因此更容易发生安全问题。

Web 安全问题要完全熟悉，需要一本重重的书籍，这里只讲解常见的安全问题和 Skihat 框架的特殊安全问题。

22.1、SQL 注入安全

在 Skihat 中，通常不会发生 SQL 安全注入问题，这是因为在数据库组件内部，将大部分的 SQL 代码使用 PDO 参数的方式来执行。但有几种情况可能会产生 SQL 注入问题：

- SQL 命令条件：使用 SQL 命令作为查询条件时。
- NormalParameter：使用 NormalParameter 设置参数值。

22.1.1、SQL 命令条件

SQL 条件命令是直接将 SQL 条件作为参数传递给数据库，再由数据库执行。

示例：

```
GuestBook::fetchAll (‘GuestBook.user = \’admins\’’);
```

直接这样的操作，不会产生 SQL 演注入攻击，但如果 admins 的值，来自于外部的请求则会产生攻击。

示例：

```
$user = $this->query (‘user”);
GuestBook::fetchAll (‘Guestbook.user = \” . $user . ‘\”);
```

使用 SQL 参数命令

如果使用 SQL 查询有参数信息，建议使用 SQL 参数命令条件。

示例：

```
$user = $this->query (‘user’);
GuestBook::fetchAll (‘Guestbook.user = ?’, ‘%’ . $user . ‘$’);
```

采用这种方式，会将$user 作为参数传递给数据库执行，而不是简单的字符串拼接。

使用类型转换函数

除此以外，如果明确数据类型，那么建议采用相关的类型转换方法，将外部参数先进行转换。

示例：

```
$start = $this->query(‘start’);
GuestBook::fetch (‘Guestbook.created <    ?’,intval ($start));
```

注意：如果类型为字符串，使用 addslashes 进行字符串预转换。

22.1.2、NormalParameter

NormalParameter 是一个特殊参数类型，更多信息请查看 NormalParameter 类。

22.2、模型字段注入

虽然 Skihat 的模型简化了应用的开发，但如果没有正确使用同样会有安全问题。

示例：

```
# 字段：id、name、group、description。
class AdminUser extends ApplicationModel {}

# editAction
$this->user = new AdminUser ($this->form (‘user’),true);
if ($this->user->save()) {
    $this->message(‘更新成功’);
    Return;
}

# edit.stp
<form action=”#” method=”POST”>
    <p><label>名字：<input type=”user[name]” /></p>
    <p><label>描述：<input type=”user[description]” /></p>
    <input type=”submit” value=”submit” />
</form>
```

在示例中，group 字段用于表示管理员的权限，普通管理员只允许更新名称和说明，不允许自己所在的组，因此在 edit.stp 中仅仅包含 name 和 description 的信息。

这看起来没有什么问题，但如果攻击者使用工具将 group 的值提交到服务器，那么当前管理员就能为自己指定任何管理员所在组的权限。

要解决这一安全问题，主要有两种方法：

- 使用 META_READONLY：指定只读字段，只读字段的值在 save 方法中不会被更新到数据库中。
- 使用 SYNTAX_FIELDS：指定更新的字段值，指定更新字段后，更新字段之外的值不会被保存。

示例：

```
# editAction
$this->user = new AdminUser ($this->form('user'),true);
If ($this->save (array(Model::SYNTAX_FIELDS => array('user','description')){
    # do something
}
```

Skihat 建议在保存任何模型数据时，都指定需要保存的字段名称。

22.3、XSS 攻击

XSS(跨站脚本攻击)，虽然不会对服务器造成损失，但会严重影响到访问的客户的安全。为了避免受到 XSS 攻击，建议采用以下方式：

- 对于客户直接上传的 HTML，使用 strip_tags 方法过滤不必要的 HTML 标志。
- 对于非 HTML 代码，输出时使用_h 方法转换为普通文本。
- 在模型验证中增加录入有效性检查，防止录入不满足要求的字段值。

通常以上方法，能够防止大部分的 XSS 攻击。

22.4、文件上传安全

上传功能是开发过程中经常都需要使用的功能，同时上传功能往往也是最容易发生安全问题的功能，这主要是因为没有能够正确的检查上传的内容和上传的值。在 Skihat 中建议采用以下方式防止文件上传安全。

- 使用 Meida 组件实现上传，Media 组件中，会对上传的文件类型、大小和扩展名等进行详细的检查；
- 使用不同的域名，将上传文件指定到不同的域名下，同时该域名仅允许执行读取操作，不允许执行其它任何操作；

22.5、CRSF 攻击

防止 CRSF 攻击的有效手段就是使用表单令牌，在 Skihat 中使用表单令牌非常简单，只需要在控制器中使用 FromFilter，同时在视图中使用 FormBuilder 类生成表单，更多信息请

查看 FormFilter。

22.6、目录安全建议

为了保证更加安全的应用，建议采用以下安全模式部署目录。

目录	权限
app	只读
app/publics	只读
app/publics/uploads	读写
data	读写
skihat	只读
vendor	只读

全站只允许 app/publics/index.php 一个文件允许执行权限，同时将 Web 服务器的网站目录设置为 app/publics。

第 23 章：Skihat 性能优化

23.1、关闭调试环境

优化过程中，最简单也是最有效的第一条就是关闭调试信息，调试信息会生成一些额外的执行代码和日志记录。

关闭调试代码非常简单，指定 SKIHAT_DEBUG 常量的值为 0。

示例：

```
# app/boots/startup.inc
define (‘SKIHAT_DEBUG’,0);
```

23.2、正确使用缓存

在 Skihat 中，在控制器和视图模板中都允许开启缓存功能，通过使用缓存功能，能够减少对于数据库的访问，从而提高应用的响应速度。

23.2.1、使用 CacheFilter 缓存过滤器

使用 CacheFilter 缓存过滤器，允许将活动方法的响应内容直接输出到缓存中，当再次访问请求时，直接从缓存中读取，减少数据库的读取和视图填充过程。

因为 CacheFilter 的特性，因此非常适合于只读内容的缓存，具体使用信息请查看 CacheFilter 类。

23.2.2、在视图中使用缓存选项

在视图方法中，inflateFile 和 inflateProc 方法都允许指定缓存信息。当多个网页需要访问同一个视图模板时，可以在使用 inflateFile 方法时，指定相同的缓存名称，从而避免每次都需要生成相同的模板内容。

如何在 inflateFile 和 inflateProc 中，使用缓存请查看 Theme 类。

23.2.3、使用缓存组件自定义缓存内容

直接使用缓存组件，可以将任何内容直接缓存到缓存组件中，从而减少数据库的访问或进行大量的运算。Cache 组件使用，请查看 Cache 类。

注意：并不是任何内容都适用于使用缓存组件缓存值，通常数据库的访问结合或需要进行大量运算才能获取的内容比较适合于使用缓存组件。

23.3、使用代码缓存器

代码缓存器的作用是减少从 PHP 代码到 opcode 之间的编译时间，通常使用 opcode 之后，性能会有 20% - 80%之间的提升。

目前流行的代码缓存器包括以下几种：

- XCache
- APC
- eAccelerator

具体每一种的配置请查看网上的相关内容。

23.4、减少使用路由转换

Skihat 的路由功能非常强大，但与之相对应的是将会产生大量的运算工作，来保证路由的正确执行。因此，当程序开发完成后，整个部署 URL 已经完成后，应当减少通过路由器来生成和转换地址。

要完成这一工作，需要在二个部分进行处理：

- 使用 Web 重写规则，减少路由转换；
- 使用视图助手方法生成 URL 地址，减少路由地址的转换；

在一个中等规模的网站应用中，可能会有上百个路由规则，而通常一个网页又有上百个 URL 地址，所以要完整的生成一个网页，可能就需要进行上万次的路由运算。

23.5、客户端优化

客户端优化也是性能优化的一个重要部分，据说 Facebook 通过优化客户端提高了 40% 的响应时间。

优化客户端有很多方法，比较常见的有以下几种：

- CSS/Javascritp 文件打包压缩：将多个文件的内容打包在一起可以减少客户端对于服务端的请求次数；使用文件压缩工具，能够文件压缩到最小尺寸（这就是为什么有 jquery.js 和 jquery.min.js）。
- 将 Javascript 放置到 HTML 结尾：通常 javascritp 用于提供动作，因此不需要最先就加载和执行。
- 使用 CSS Sprite 优化 CSS：CSS Sprite 是将多个背景或 CSS 图标样式由一个文件来提供，这样的好处在于能够减少 CSS 引用的外部文件数据，从而减少服务端的请求数量。

除此以外，还可以使用 YSlow 查看前端优化的更多参考。