YII 框架源码分析

百度联盟事业部——黄银锋

目　录

1、 引言 3
1.1、Yii 简介 3
1.2、本文内容与结构 3
2、组件化与模块化 4
2.1、框架加载和运行流程 4
2.2、YiiBase 静态类 5
2.3、组件 6
2.4、模块 9
2.5、App 应用 10
2.6、WebApp 应用 11
3、系统组件 13
3.1、日志路由组件 13
3.2、Url 管理组件 15
3.3、异常处理组件 17
3.4、Cache 组件 17
3.5、角访问控制组件 19
3.6、全局状态组件 21
4、控制器层 23
4.1、Action 23
4.2、Filter 24
4.3、Action 与 Filter 的执行流程 26
4.4、访问控制过滤器 27
5、模型层 30
5.1、DAO 层 30
5.1.1、数据库连接组件 30
5.1.2、事务对象 31
5.1.3、Command 对象 31
5.2、元数据与 Command 构造器 32
5.2.1、表结构查询 32
5.2.2、查询条件对象 33
5.2.3、Command 构造器 33
5.3、ORM(ActiveRecord) 34
5.3.1、表的元数据信息 34
5.3.2、单表 ORM 34
5.3.3、多表 ORM 36
5.3.4、CModel 与 CValidator 37
6、视图层 38
6.1、视图渲染流程 38
6.2、Widget 39
6.3、客户端脚本组件 40

百度联盟事业部 huangyinfeng@baidu.com

1、引言

1.1、Yii 简介

Yii 的作者是美籍华人“薛强”，他原是 Prado 核心开发成员之一。2008 年薛强另起炉灶，开发了 Yii 框架，于 2008 年 12 月 3 日发布了 Yii1.0 版本。

Yii 是目前比较优秀的 PHP 框架之一，它的支持的特性包括：MVC、DAO/ActiveRecord、I18N/L10N、caching、AJAX 支持、用户认证和基于角色的访问控制、脚手架、输入验证、部件、事件、主题化以及 Web 服务等。

Yii 的很多思想参考了其它一些比较优秀的 Web 框架(我们写东西时是不是也喜欢参考别人的？有木有？嘿嘿，都喜欢站在别人的肩膀上干活！)，下面是一个简短的列表：

框架名称	参考思想
Prado	基于组件和事件驱动编程模式、数据库抽象层、模块化的应用架构、国际化和本地化等
Ruby on Rails	配置思想、基于 Active Record 的 ORM
jQuery	集成了 jQuery
Symfony	过滤设计和插件架构
Joomla	模块化设计和信息翻译方案

1.2、本文内容与结构

本文对 Yii1.1.8 版本的源代码进行了深入的分析，本文的内容与结构为：

组件化与模块化：对 Yii 的基于组件和事件驱动编程模式的基础类(CComponent)进行分析；对组件化和模块化的工作原理进行分析；对 WebApp 应用创建 Controller 流程等进行分析。

系统组件：对 Yii 框架自带的重要组件进行分析，主要包括：日志路由组件、Url 管理组件、异常处理组件、Cache 组件、基于角色的访问控制组件等。

控制器层：控制器主要包含 Action 和 Filter，对 Action 与 Filter 的工作原理进行分析。

模型层：对 DAO 层、元数据和 Command 构造器、ORM 的原理进行分析

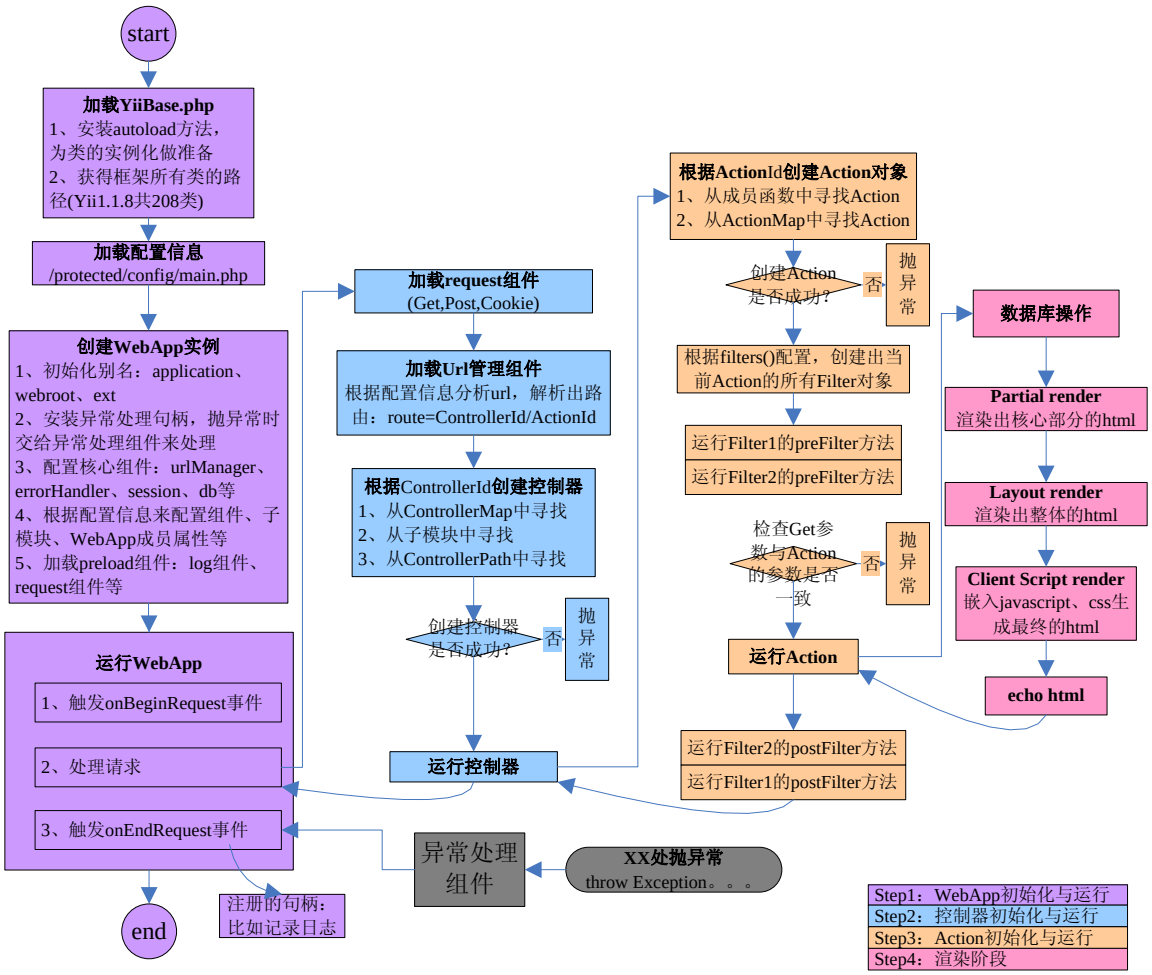
视图层：对视图层的渲染过程、Widget 和客户端脚本组件等进行分析

本文档中的错误或不妥之处在所难免，殷切希望本文档的读者给予批评指正！

百度联盟事业部 huangyinfeng@baidu.com

2、组件化与模块化

2.1、框架加载和运行流程



Yii 框架加载和运行流程共分 4 个阶段(也许看着有点吓人，木有关系，我们先知道一个大概)：

Step1：WebApp 初始化与运行

1.1、 加载 YiiBase.php，安装 autoload 方法；加载用户的配置文件；

1.2、 创建 WebApp 应用，并对 App 进行初始化，加载部分组件，最后执行 WebApp

Step2：控制器初始化与运行

2.1、 加载 request 组件，加载 Url 管理组件，获得路由信息 route=ControllerId/ActionId

2.2、 创建出控制器实例，并运行控制器

Step3：控制器初始化与运行

3.1、 根据路由创建出 Action

3.2、 根据配置，创建出该 Action 的 Filter；

3.3、 执行 Filter 和 Action

Step4：渲染阶段

4.1、 渲染部分视图和渲染布局视图

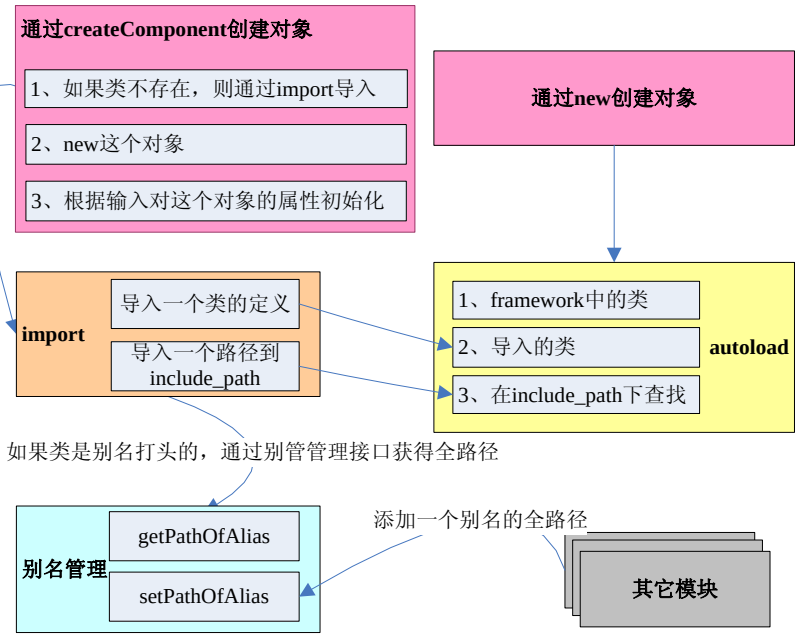
4.2、 渲染注册的 javascript 和 css

2.2、YiiBase 静态类

YiiBase 为 YII 框架的运行提供了公共的基础功能：别名管理与对象创建管理。

在创建一个 php 的对象时，需要先 include 这个类的定义文件，然后再 new 这个对象。在不同环境下(开发环境/测试环境/线上环境)，apache 的 webroot 路径的配置可能不一样，所以这个类的定义文件的全路径就会不同，Yii 框架通过 YiiBase 的别名管理来解决了这个问题。

在创建对象时，需要导入对应类的定义，经常需要使用这 5 个函数：include()、include_once()、require()、require_once()、set_include_path()。Yii 通过使用 YiiBase::import()来统一解决这个问题。下图描述了 YiiBase 提供“别名管理与对象创建管理”的工作原理：



首先看别名管理，它是通过为某个文件夹(一个文件夹往往对应一个模块)起一个别名，在 YII 框架中可以使用这个别名来替代这个文件夹的全路径，比如：system 别名代表的是框架 /home/work/yii/framework 的路径，所以可以使用 system.base.CApplication 代表 /home/work/yii/framework/base/CApplication.php 文件的路径。当然在应用层(我们)的代码中也可以通过 Yii::setPathOfAlias 来注册别名。

一般情况下我们使用绝对路径或者相对路径来进行文件引用，当然这 2 种情况都有弊端。绝对路径：当我们的代码部署到测试环境或者线上环境的时候需要大量修改被 include 文件的路径；相对路径：当某些模块的文件夹的位置发生调整(改名)的时候，所有的相对路径都需要修改。而使用别名的方式只需要改一处：注册别名的时候，即 Yii::setPathOfAlias()。从而将文件夹的变动而导致的代码改动集中到一处完成。

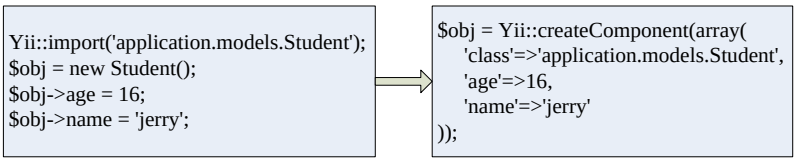
再看 import 功能：a、导入一个类的定义，从而可以创建该类的对象；b、将某个文件夹加入到 include_path，从而可以直接 include 这个文件下的所有文件。Yii::import 相当于如下 5 个函数的统一：include()、include_once()、require()、require_once()、set_include_path()。而且一般情况下速度会比这些函数更快。当然 Yii::import 支持别名的功能，从而可以解决路径变动带来的麻烦。

最后看一下对象的创建，在 YII 框架中有 2 中方法创建对象：1、使用 new 关键字；2、

使用 Yii::createComponent 方法。

当使用 new 关键字创建对象时，autoload 会分 3 步来寻找对应类的定义：a、判断是否为 framework 中的类(framework 的所有类和这个类的全路径都保存在 YiiBase 的一个成员变量中，就相当于整个框架都 import 了)；2、判断是否使用 Yii::import 导入了这个类，对于非框架的类，我们在创建这个类的对象时需要先 import 这个类的定义；3、从 include_path 目录下查找以这个类名字命名的 php 脚本，所以在开发的时候类名尽量与文件名保存一致，这样我们导入(import)包含这个文件的文件夹就行了，从而无需把这个文件夹中的每个文件都导入一遍。

当使用 Yii::createComponent 方法创建对象时，它提供了比 new 关键字更多的功能：a、通过这个类的全路径别名来指定类的位置和类名(类名必须与文件名一致)，当这个类还没有导入的时候，会根据全路径来自动导入这个类的定义；2、对创建出来的对象的成员变量进行赋值。即如下图描述，原来要写 3 行以上的代码，现在一行代码就可以搞定(write less, do more)。



2.3、组件

CComponent 类就是组件，它为整个框架的组件编程和事件驱动编程提供了基础，YII 框架中的大部分类都将 CComponent 类作为基类。CComponent 类为它的子类提供 3 个特性：

1、成员变量扩展

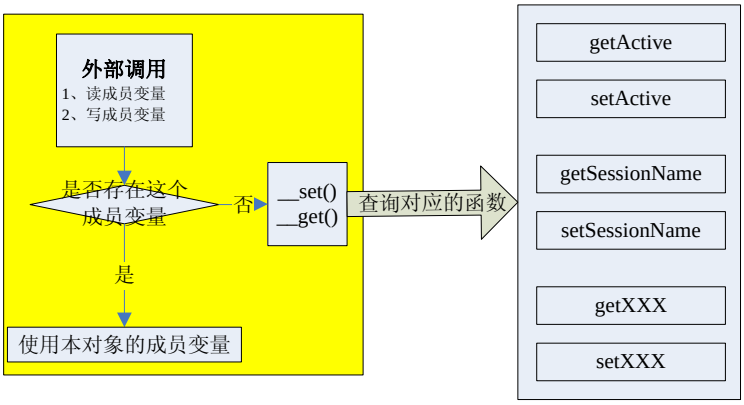
通过定义两个成员函数（getXXX/setXXX）来定义一个成员变量，比如：

```
public function getText() {…}
public function setText {…}
```

这样就相当于定义了一个 text 成员变量，可以这样调用

```
$a=new CComponent;
$a=$component->text;        // 等价于$a=$component->getText();
$component->text='abc';     // 等价于$component->setText('abc');
```

CComponent 是通过魔术方法__get 和__set 来实现“成员变量扩展”特性的，如果对类本身不存在的成员变量进行操作时，php 会调用这个类的__get 和__set 方法来进行处理。CComponent 利用这两个魔术方法实现了“成员变量扩展”特性。下图描述了一个 CComponent 的子类，它增加了 active 和 sessionName 两个成员变量，该图描述了对于这两个成员变量的调用流程。



面向对象编程中直接定义一个成员变量就可以了，为什么 CComponent 要通过定义 2 个函数来实现一个成员变量呢？一个主要得原因是需要对成员变量进行“延时加载”，一般情况下类的成员变量是在构造函数或者初始化函数进行统一赋值，但是在一次 web 请求的处理过程中不是每个成员变量都会被使用，比如 App 类中定义了两个成员变量：$cache 和$db（$cache 是一个缓存对象，$db 是一个数据库链接对象），这两个对象在 App 类初始化的时候创建，但是一个 web 网站的有些页面，它内容可以通过缓存获取，那么数据库链接对象其实就不需要创建。如果将 App 定义为 CComponent 的子类，在 App 类中定义两个方法：getCache/getDb，这样就可以做到第一次使用 db 成员变量的时候，才调用 getDb 函数来进行数据库链接的初始化，从而实现延时加载——即在第一次使用时进行初始化。虽然延时加载会增加一次函数调用，但是可以减少不必要的成员变量的初始化（总体上其实是提升了网站的访问速度），而且可以使得我们的代码更加易维护、易扩展。

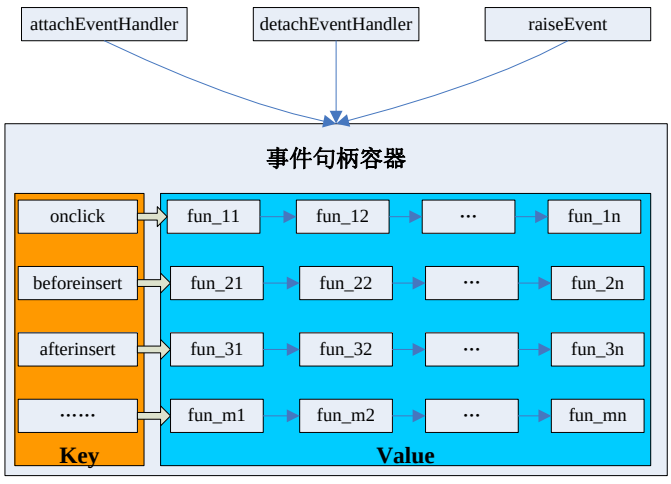
延时加载应该是“成员变量扩展”特性的最重要的用途，当然这个特性还会有其它用途，想一想，当你操作一个成员变量的时候，你其实是在调用 getXXX 和 setXXX 成员函数，你是在调用一段代码！

2、事件模型

事件模型就是设计模式中的“观察者模式”：当对象的状态发生了变化，那么这个对象可以将该事件通知其它对象。

为了使用事件模型，需要实现这三个步骤：1、定义事件；2、注册事件句柄；3、触发事件。

CComponent 的子类通过定义一个以 on 打头的成员函数来定义一个事件，比如：public function onClick(){...}，接着通过调用 attachEventHandler 成员函数来注册事件句柄（可以注册多个事件句柄），最后通过调用 raiseEvent 来触发事件。



CComponent 类使用一个私有的成员变量来保存事件以及处理该事件的所有句柄，该成员变量可以看作一个 hash 表，hash 表的 key 是事件的名称，hash 表的 value 是事件处理函数链表。

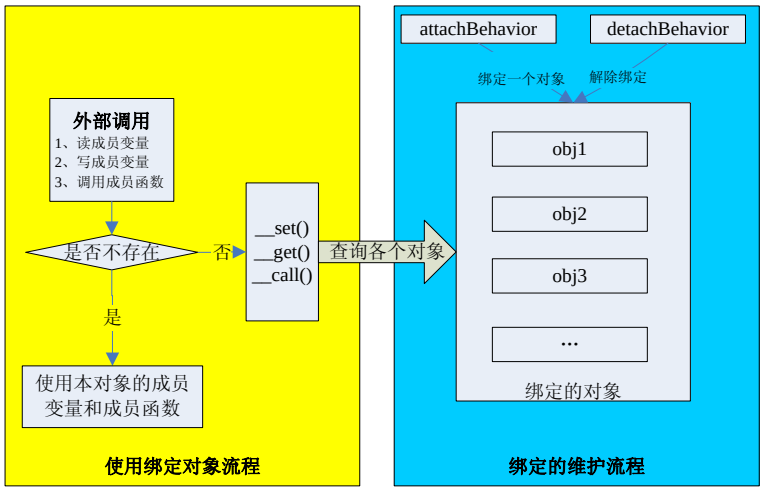
3、行为类绑定

有两种办法可以对类添加特性：1、直接修改这个类的代码：添加一些成员函数和成员变量；2、派生：通过子类来扩展。很明显第二种方法更加易维护、易扩展。如果需要对一个类添加多个特性（多人在不同时期），那么需要进行多级派生，这显然加大了维护成本。

CComponent 使用一种特殊的方式对类信息扩展——行为类绑定。行为类是 CBehavior 类的一个子类，CComponent 可以将一个或者多个 CBehavior 类的成员函数和成员变量添加到自己身上，并且在不需要的时候卸载掉某些 CBehavior 类。下面是一个简单的例子：

```
//计算器类
class Calculator extends CBehavior
{
    public function add($x, $y) { return $x + $y; }
    public function sub($x, $y) { return $x - $y; }
    ...
}
$comp = new CComponent();
//为我的类添加计算器功能
$comp->attachbehavior('calculator', new Calculator());
$comp->add(2, 5);
$comp->sub(2, 5);
```

CComponent 通过__get、__set 和__call 这 3 个魔术方法来实现“行为类绑定”这个特性，当调用 CComponent 类不存在的成员变量和成员方法的时候，CComponent 类会通过这三个魔法方法在“动态绑定的行为对象”上进行查找。即将不存在的成员变量和成员方法路由到“动态绑定对象”上。



可以用 3 句话来总结 CComponent 类的特性：

1、 更好的配置一个对象，当设置对象的成员变量的时候，其实是运行一段代码；

2、 更好的监听一个对象，当对象的内部状态发生变化的时候，其它对象可以得到通知；

3、 更好的扩展一个对象，可以给一个对象增加成员变量和成员函数，还能监听这个对象的状态。

2.4、模块

模块是整个系统中一些相对独立的程序单元，完成一个相对独立的软件功能。比如 Yii 自带的 gii 模块，它实现了在线代码生成的功能。CModule 是所有模块类的基类，它有 3 部分组成：

a、基本属性(模块 id，模块路径等)；

b、组件，这是模块的核心组成部分，模块可以看成这些组件的容器；

c、子模块，这为模块提供了扩展性，比如一个模块做大了，可以拆成多个子模块(每个子模块也是有这 3 部分组成，是一个递归结构)。

下图是模块与它的成员之间的包含关系图：

下表列出了 CModule 各个组成部分：

3 部分	详细成员	说明
基本属性 (用户对整个模块的全局性的东西进行配置)	id	模块的 id
	parentModule	父模块
	basePath	当前模块的路径
	modulePath	子模块的路径
	params	模块的参数
	preload	需要预先加载的组件 id
	behaviors	绑定的行为类
	aliases	新增加的别名，添加到 YiiBase 的别名管理中
	import	需要包含的文件或者路径
组件 (这是模块的核心组成部分)	components	数组类型，数组的每个成员描述了一个组件
子模块 (这为模块提供了扩展性)	modules	数组类型，数组的每个成员描述了一个模块，每个模块也是有这 3 部分组成，是递归结构

可以非常方便的对模块的这 3 个组成部分进行初始化：使用一个数组进行配置，数组的 key 是需要配置的属性，value 就是需要配置的值，下图是一个例子，为什么会如此方面的进行配置呢？因为 CModule 继承自 CComponent 类，所以在对成员属性进行配置的时候，其实是在运行一段代码，即一个成员函数。

```
array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',//模块的路径
    'preload'=>array('log'),//需要预先加载日志组件
    'import'=>array('application.models.*', 'application.components.*',),//需要include的路径
    //组件的配置
    'components'=>array(
        'user'=>array(//用户组件的配置
            'allowAutoLogin'=>true
            ),
```

```
        'log'=>array(//日志组件的配置
            'class'=>'CLogRouter',
            'routes'=>array(array('class'=>'CWebLogRoute','levels'=>'trace, profile'))
            )
    ),
    //模块的配置
    'modules'=>array(
        'gii'=>array(//自动生成代码模块的配置
            'class'=>'system.gii.GiiModule',
            'password'=>'123456'
        ),
    ),
);
```

2.5、App 应用

应用是指请求处理中的执行上下文。它的主要任务是分析用户请求并将其分派到合适的控制器中以作进一步处理。它同时作为服务中心，维护应用级别的配置。鉴于此，应用也叫做“前端控制器”。

Yii 使用 CApplication 类用来表示 App 应用，CApplication 继承自 CModule，它在父类基础上做了 3 方面的扩展：1、增加一个 run 方法；2、添加了若干成员属性；3、添加了若干组件。

run 方法的作用相当于 C 语言的 main 函数，是整个程序开始运行的入口，内部调用虚函数 processRequest 来处理每个请求，CApplication 有 2 个子类：CWebApplication 和 CConsoleApplication，它们都实现了该方法。在处理每个请求的开始和结束分别发起了 onBeginRequest 和 onEndRequest 事件，用于通知监听的观察者。复习一下“Yii 框架加载和运行流程”图，从中可以找到该方法在整个流程中所起的作用。

添加的成员变量、成员函数和组件见下表：

类别	名称	说明
成员变量	name	应用的名称
	charset	应用的编码集，默认为 UTF-8
	sourceLanguage	编码所使用的语言和区域 id 号，这在开发多语言时需要，默认为 UTF-8
	language	app 要求的语言和区域 id 号，默认为 sourceLanguage
	runtimePath	运行时的路径，比如全局的状态会保存到这个路径下，默认为 application.runtime
	extensionPath	放第三方扩展的路径，默认为 application.ext
	timezone	获取或者设置时区
	locale	本地化对象，用于对时间、数字等的本地化
	globalsate	全局状态数组，该数组会被持久化(通过 statePersister 实现)
组件	coreMessages	对框架层内容进行翻译，支持多语言
	messages	对应用层内容进行翻译，支持多语言

	db	数据库组件
	errorHandler	异常处理组件，该组件与 App 配合来处理所有的异常
	securityManager	安全管理组件
	statePersister	状态持久化组件
	urlManager	url 管理组件
	request	请求组件
	format	格式化组件

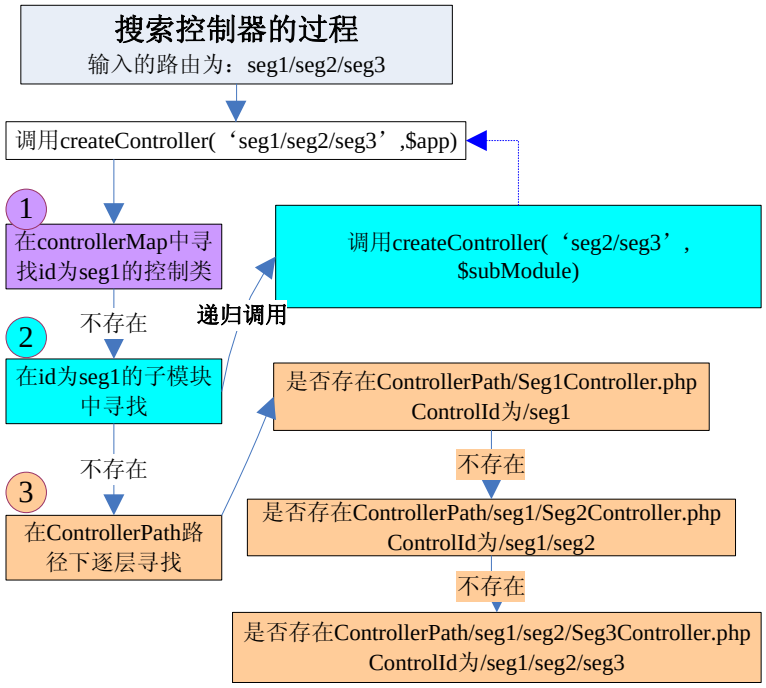
2.6、WebApp 应用

每个 web 请求都由 WebApp 应用来处理，即 WebApp 应用为 http 请求的处理提供了运行的环境。WebApp 应用就是 CWebApplication 类，它的最主要工作是根据 url 中的路由来创建对于的控制类，下图描述了控制器创建的过程，主要由 3 步组成：

1、在成员变量 controllerMap 中查找，判断是否有对应的 Controller，controllerMap 的优先级最高

2、在子模块中中查找，判断是否有对应的 Controller

3、在 ControllerPath 及其子文件夹中查找



添加的重要的成员变量、成员函数和组件见下表：

类别	名称	说明
成员变量	defaultController	默认的控制类，如果没有指定控制器，则使用该控制器
	layout	默认的布局，如果控制器没有指定布局，则使用该布局
	controllerMap	控制器映射表，给某些特定的路由指定控制器
	theme	设置主题
	controller	当前的控制器对象
	controllerPath	控制器文件的路径

	ViewPath	视图层文件的路径，默认为 protected/views/
	SystemViewPath	系统视图文件的路径，默认为 protected/views/system/
	LayoutPath	布局文件的路径，默认为 protected/views/layouts/
组件	session	session 组件
	assetManager	资源管理组件，用于发布私有的 js、css 和 image
	user	用户组件，用户登录等
	themeManager	主题组件
	authManager	权限组件，实现了基于角色的权限控制
	clientScript	客户端脚本管理组件，管理 js 和 css 代码

3、系统组件

3.1、日志路由组件

每个 Web 系统在运行的过程中都需要记录日志，日志可以记录到文件或数据库中，在开发阶段可以把日志直接输出到页面得底部，这样可以加快开发速度。Yii 在日志处理上做了如下 2 点重要工作：

1、每个 Http 请求，可能需要记录多条日志(数据库更新日志/与其它系统交互日志)。比如某次 Http 请求要记录 18 条日志，我们是每记一条日志都写一次硬盘(即写 18 硬盘)呢，还是在请求快结束的时候一次性写硬盘？很显然，先把这些日志保存在一个 php 的数组中，在请求快结束的时候，把数组中的所有日志一次性写硬盘速度要快一些。

2、每条日志可以从 2 个维度来进行分类：日志的严重级别、日志的业务逻辑。用下表来描述“百度创意专家”产品的日志在这 2 个维度上的情况：

业务逻辑 / 严重级别	数据库日志	用户中心接口日志	Drmc 接口日志	Memcache 日志
trace				
info				
profile				
warning				
error				

按业务逻辑分为：数据库操作日志、用户中心接口日志、Drmc 接口日志、Memcache 更新日志等等。

按照严重级别分为：trace、info、profile、warning、error。

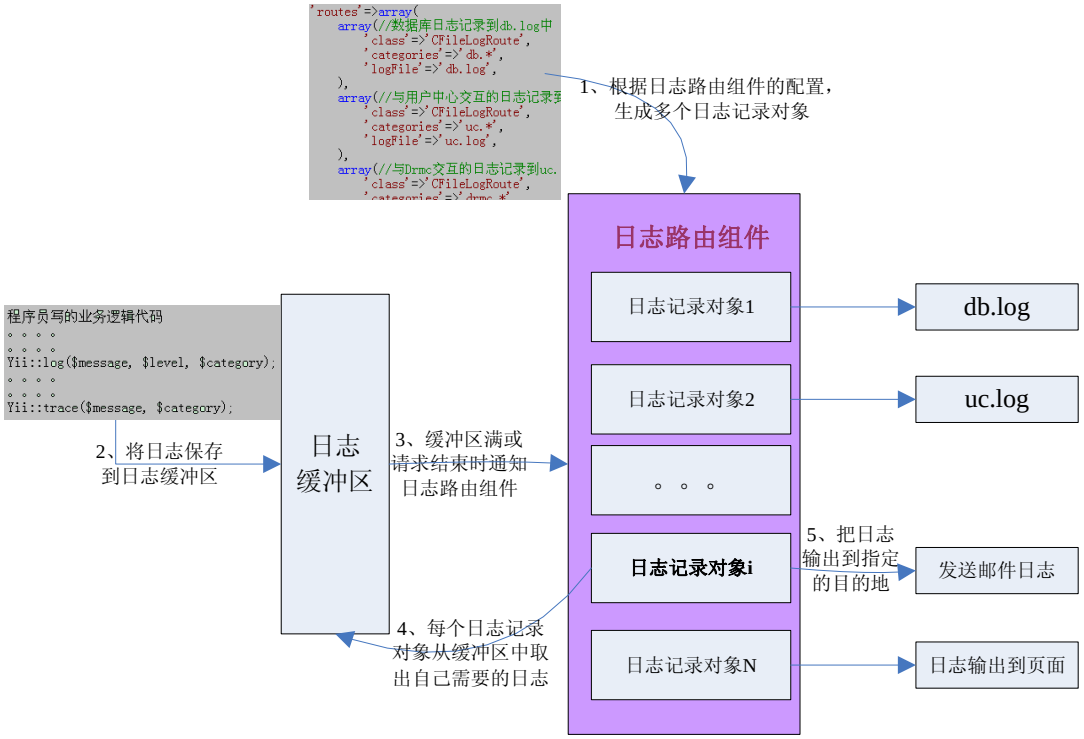
我们可能希望把不同业务逻辑(数据库日志、与其它系统交互的日志)的日志记录到不同的文件中，这样可以分门别类的查看。因为 error 日志一般比较严重，所以我们可能还希望把所有的 error 记录到一个单独的文件中或者 mongodb 中。Yii 中的日志路由组件可以将不同类别的日志路由到不同的目的地(文件、数据库、邮箱和页面)，利用它可以非常方便维护和管理日志。

如下是一个日志路由组件的配置，该配置将不同业务逻辑的日志记录到不同的文件中，把错误日志单独记录到 error.log 文件中，把严重的日志直接发邮件，在开发过程还将日志输出到页面上，加快了开发速度。具体配置如下：

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(//数据库日志记录到db.log中
            'class'=>'CFileLogRoute',
            'categories'=>'db.*',
            'logFile'=>'db.log',
        ),
        array(//与用户中心交互的日志记录到uc.log中
            'class'=>'CFileLogRoute',
            'categories'=>'uc.*',
```

```
            'logFile'=>'uc.log',
        ),
        array(//与Drmc交互的日志记录到uc.log中
            'class'=>'CFileLogRoute',
            'categories'=>'drmc.*',
            'logFile'=>'drmc.log',
        ),
        array(//所有的错误日志记录到error.log中
            'class'=>'CFileLogRoute',
            'levels'=>'error',
            'logFile'=>'error.log',
        ),
        array(//因为用户中心很重要，所有的用户中心错误日志需要离开发邮件
            'class'=>'CEmailLogRoute',
            'categories'=>'uc.*',
            'levels'=>'error',
            'emails'=>'admaker@baidu.com',
        ),
        array(//开发过程中，把所有的日志直接打印到页面底部，这样就不需要登录服务器看日志了
            'class'=>'CWebLogRoute'
            'levels'=>'trace,info,profile,warning,error',
        ),
)
```

通过上面的代码可以知道，Yii 的日志记录是通过配置数组驱动的，接下来对 Yii 中日志处理进行深入的分析，下图描述 Yii 中日志处理的流程和原理：



一次 Http 请求的过程中，记录日志的处理流程分如下 5 个阶段：

Step1：根据日志路由器的配置信息，生成各个日志记录对象，即 CFileLogRoute、CEmailLogRoute 和 CWebLogRoute 的对象，日志路由组件统一管理这些对象；

Step2：程序员调用写日志的接口(见下表)，来记录日志，所有的日志都是暂时保存在一个 php 的数组缓冲区中；

Step3：当缓冲区满的时候或请求处理结束的时候，会触发以个 Flush 事件，通知日志路由组件来取日志，这里使用的就是是观察者模式；

Step4：每个日志记录对象分别取出自己需要的日志，比如数据库的日志、用户中心交互日志、error 级别的日志等，即各取所需；

Step5：每个日志记录对象分别保存自己的日志。CFileLogRoute 对象把日志保存到文件中；CEmailLogRoute 对日志进行发送邮件；CWebLogRoute 把日志输出到 web 页面上。

Yii 提供了如下 4 个接口用于记录日志：

接口名称	用途
Yii::log($msg,$level=CLogger::LEVEL_INFO,$category='application')	记录日志
Yii::trace($msg,$category='application')	记录调试日志
Yii::beginProfile($token,$category='application')	记录 profile 开始时刻
Yii::endProfile($token,$category='application')	记录 profile 结束时刻

3.2、Url 管理组件

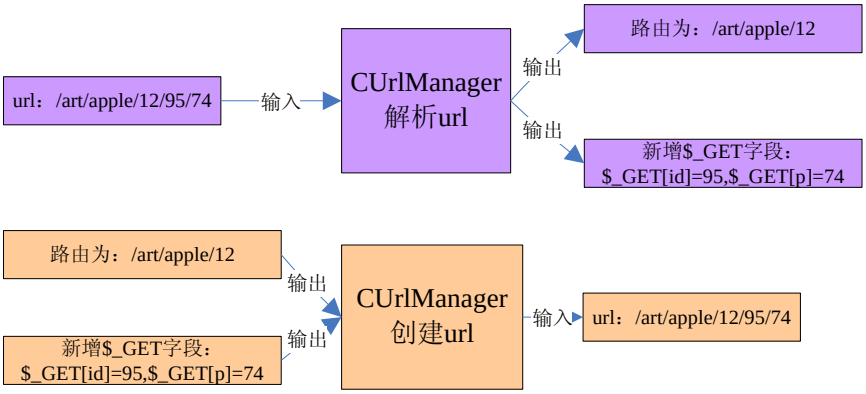
url 管理组件主要提供 2 个功能：

1、根据用户输入的 url，解析出处理这个请求的**路由**——由哪个 Controller 的哪个 Action 来处理，同时将 url 中的部分参数添加到$_GET 参数中。在每个 web 框架中都需要一个这样的组件来进行路由分发的工作。

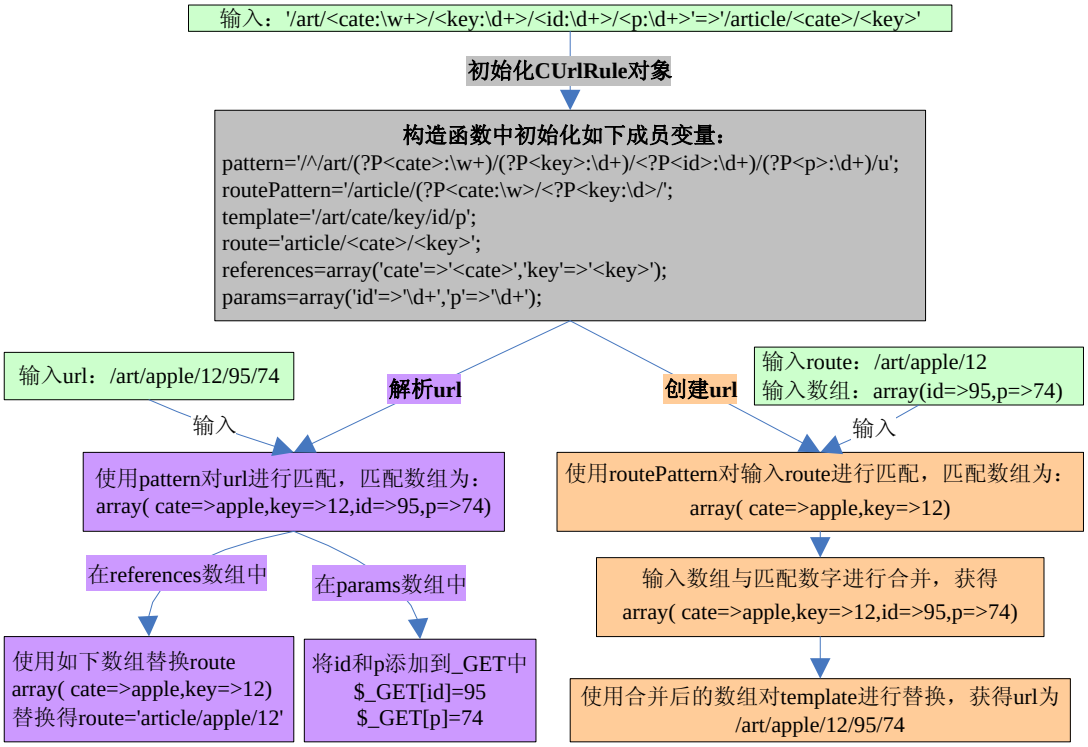
2、根据路由和参数数组来**创建 url**。在视图层可以对 url 进行硬编码，即直接写死 url 地址，但是这往往缺乏灵活性，为后期的维护带来成本。

```
array(
 'components'=>array(
     'urlFormat'=>'path',
     'rules'=>array(
         '/art/<cate:\w+>/<key:\d+>/<id:\d+>/<p:\d+>'=>'article/<cate>/<key>',
         'post/<id:\d+>/<title:.*?>'=>'post/view',
         '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
       ),
   ),
);
```

如上是一个 url 管理组件的配置，一共有 3 条规则。下图以第一条规则为例，说明了 url 解析和 url 创建的 2 个功能。对于每个路由规则，CUrlManager 都会创建一个 CUrlRule 对象来处理这条规则对应的这个 2 个功能，所以说有一条规则就会有几个 CUrlRule 对象。所以 CUrlRule 才是 url 管理的核心所在，接下来分析 CUrlRule 的工作原理。



每条 url 路由规则由一个 CUrlRule 对象来进行处理，接下来以如下路由规则为例：'/art/<cate:\w+>/<key:\d+>/<id:\d+>/<p:\d+>'=>'article/<cate>/<key>'，说明 url 解析和 url 创建的处理过程。每个 CUrlRule 对象处理 url 的过程可以分为 3 个阶段：



1、 初始化 CUrlRule 对象

在 CUrlRule 对象的构造函数中，会初始化 6 个重要的成员变量：

成员变量名称	用途
pattern	用于对 url 进行匹配的正则表达式，在解析 url 阶段使用
routePattern	用于对路由进行匹配的正则表达式，在创建 url 阶段使用
template	记录 url 由哪些字段组成，是创建 url 的模板，在创建 url 阶段，是要将这些字段填上值，就可以得到需要的 url 了
route	路由路径的格式
references	路由路径中哪些字段来源与输入的 url，在解析 url 阶段使用
params	url 中哪些字段需要添加到$_GET 数字中去，在解析 url 阶段使用

2、 解析 url

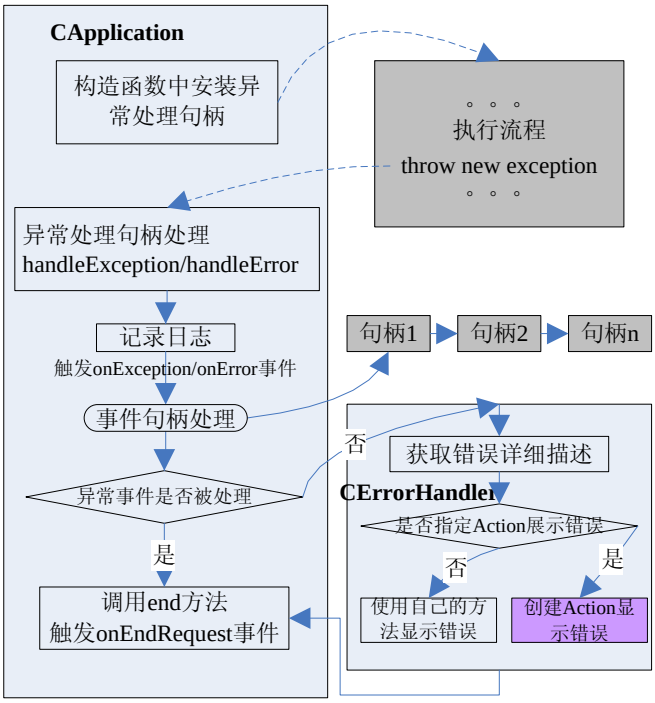
解析 url 的工作分 3 步走：a、根据 pattern 规则，解析出 url 中的各个字段；b、根据 references 对路由中的引用字段进行替换；c、将 params 中指定的字段添加到$_GET 数组中

3、 创建 url

创建 url 的工作分 3 步走：a、根据 routePattern 规则，解析出输入的路由中各个字段；b、将输入的参数数组和上一步解析的数组进行合并；c、用合并后的数组对 template 进行替换

3.3、异常处理组件

异常处理组件与 CApplication 一起配合来处理所有异常(未捕获的)。



通过上图可以看出，CApplication 将它的 handleException/handleError 方法注册为事件处理句柄，即 CApplication 得到所有的异常，然后将它交给异常处理组件处理。

异常处理最主要的工作是给浏览器端展示异常信息，一般都是将异常交给某个 Action 来展示：如果是正常请求，就返回一个异常页面；如果是 ajax 请求，就返回一个 json，由浏览器端的 javascript 对 json 进行展示。

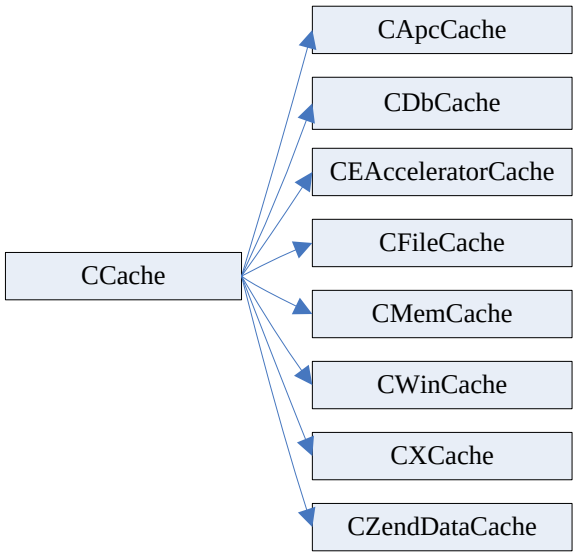
3.4、Cache 组件

使用缓存可以很好的提升 web 系统的性能，常用的缓存有：memcache、apc 和 redis 等，Yii 定义了 CCache 类，它为访问各种缓存设定了统一的接口。

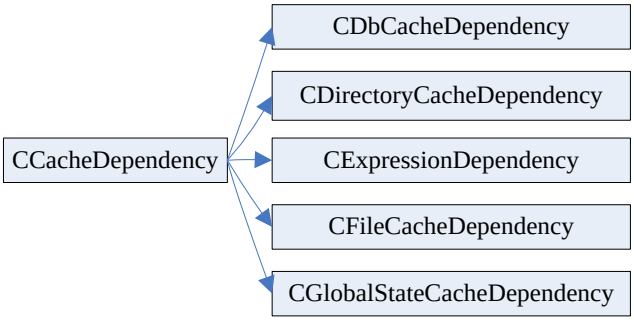
接口名	用途
get()	从缓存中读一条数据
mget()	从缓存中读多条数据
set()	往缓存中写一条数据

add()	往缓存中添加一条数据
delete()	从缓存中删除一条数据
flush()	清空缓存

如下图，Yii 使用 CCache 的子类来表示缓存组件，比如：CApcCache 表示 apc 缓存，CMemCache 表示 memcache 缓存。



默认情况下，缓存数据的失效依赖于缓存设定的时间，但是缓存数据也可以依赖于其它条件而失效。我们将一个依赖关系表现为一个 CCacheDependency 或其子类的实例。当调用 set()时，我们连同要缓存的数据将其一同传入。如下图，Yii 内置了 5 种依赖关系类。



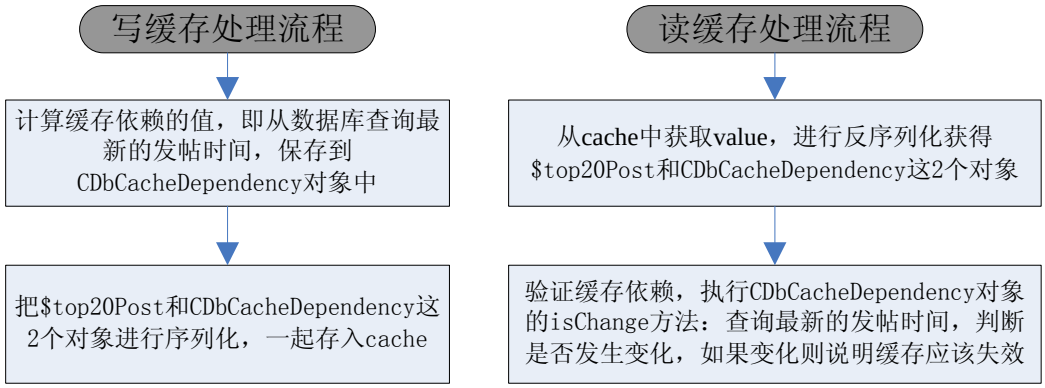
下面用一个例子讲解缓存和缓存依赖的用法和原理。比如我们有一个论坛，论坛首页有一个最新帖子区(显示最新的 20 个帖子)，即只要用户发表帖子，那么刷新首页就可以立刻看到他发表的帖子，不能有延时。保存帖子的表名为 Post，发帖时间为 createTime 字段，如下显示了获取最新帖子的主要代码：

```
array(
    'components'=>array(
        'cache'=>array(
            'class'=>'CMemCache',//配置缓存，使用memcache进行缓冲
            'servers'=>array(array('host'=>'127.0.0.1', 'port'=>11211)),
        ),),
);

$top20Post = Yii::app()->cache->get('top20Post');//从cache中读数据
if($top20Post==false){
```

```
    $top20Post = Yii::app()->db->createCommand('select * from Post order by createTime desc limit
20')->queryAll();//从数据库中查询
    $dependcy = new CDbCacheDependency('select max(createTime) from Post');//创建缓存依赖，依赖于最新发帖时间
    Yii::app()->cache->set('top20Post', $top20Post, 600, $dependcy);//往cache中写数据
}
```

从上面的代码可以看出，首先对 cache 配置，这里使用的是 memcache，接着从 cache 中取数据，如果 cache 已经失效，则将从数据库中获取的数据重新保存到 cache 中，缓存依赖使用的最新的发帖时间。

接下来分析一下写 cache 和读 cache 两种操作的原理，缓存依赖其实就是在写缓存的时候获取一下最新发帖时间，然后在读缓存的时候再获取一下最新发帖时间，判断这 2 个时间是否有变化，如果不相等，就可以说明缓存失效了。



3.5、角访问控制组件

基于角色的访问控制(Role-Based Access Control)提供了一种简单而又强大的集中访问控制机制，Yii 通过 CAuthManager 组件实现了分等级的 RBAC 机制。

在 Yii 的 RBAC 中，一个最基本的概念是“授权项目”(authorization item)。一个授权项目就是做某件事的权限(例如新帖发布，用户管理)。根据其权限的粒度，授权项目可分为 3 种类型：操作(operations)、任务(tasks)和角色(roles)。一个角色由若干任务组成，一个任务由若干操作组成，而一个操作就是一个许可，不可再分，即角色的粒度最粗，操作的粒度最细。例如，我们有一个系统，它有一个管理员角色，它由帖子管理和用户管理任务组成。用户管理任务可以包含创建用户，修改用户和删除用户操作组成。为保持灵活性，Yii 还允许一个角色包含其他角色或操作，一个任务可以包含其他操作，一个操作可以包括其他操作。

授权项目的名字必须是唯一的，一个授权项目可能与一个业务规则关联(bizRule)。业务规则是一段 PHP 代码，在进行验证授权项目的访问权限检查时，会被执行。仅在执行返回为 true 时，用户才会被视为拥有此授权项目所代表的权限许可。例如，当定义一个 updatePost(更新帖子)操作时，我们可以添加一个检查当前用户 ID 是否与此帖子的作者 ID 相同的业务规则，这样，只有作者自己才有更新帖子的权限。

通过授权项目，我们可以构建一个授权等级体系。在等级体系中，如果项目 A 由另外的项目 B 组成（或者说 A 继承了 B 所代表的权限），则 A 就是 B 的父项目。一个授权项目可以有多个子项目，也可以有多个父项目。因此，授权等级体系是一个偏序图(partial-order graph)结构而不是一种树状结构。在这种等级体系中，角色项目位于最顶层，操作项目位于最底层，而任务项目位于两者之间。

一旦有了授权等级体系，我们就可以将此体系中的角色分配给用户。而一个用户一旦被赋予一个角色，他就会拥有此角色所代表的权限。例如，如果我们赋予一个用户管理员的角色，他就会拥有管理员的权限，包括帖子管理和用户管理(以及相应的操作，例如创建用户)。

定义授权等级体总共分三步：定义授权项目，建立授权项目之间的关系，还要分配角色给用户。authManager 应用组件提供了用于完成这三项任务的一系列 API。

Step1：要定义一个授权项目，可调用下列方法之一，具体取决于项目的类型：

CAuthManager::createRole

CAuthManager::createTask

CAuthManager::createOperation

建立授权项目之后，我们就可以调用下列方法建立授权项目之间的关系：

CAuthManager::addItemChild

CAuthManager::removeItemChild

CAuthItem::addChild

CAuthItem::removeChild

最后，我们调用下列方法将角色分配给用户。

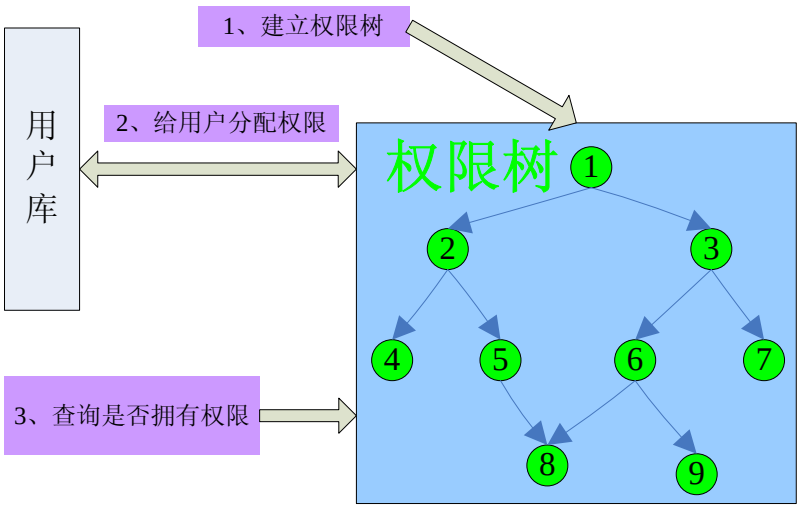
CAuthManager::assign

CAuthManager::revoke

下面的代码演示了使用 Yii 提供的 API 构建一个授权体系的例子：

```
$auth=Yii::app()->authManager;

$auth->createOperation('createPost','create a post');
$auth->createOperation('readPost','read a post');
$auth->createOperation('updatePost','update a post');
$auth->createOperation('deletePost','delete a post');

$bizRule='return Yii::app()->user->id==$params["post"]->authID;';
$task=$auth->createTask('updateOwnPost','update a post by author himself',$bizRule);
$task->addChild('updatePost');

$role=$auth->createRole('reader');
$role->addChild('readPost');

$role=$auth->createRole('author');
$role->addChild('reader');
$role->addChild('createPost');
$role->addChild('updateOwnPost');

$role=$auth->createRole('editor');
$role->addChild('reader');
$role->addChild('updatePost');

$role=$auth->createRole('admin');
$role->addChild('editor');
$role->addChild('author');
```

```
$role->addChild('deletePost');

$auth->assign('reader','readerA');
$auth->assign('author','authorB');
$auth->assign('editor','editorC');
$auth->assign('admin','adminD');
//检查权限
checkAccess('deletePost');
```



基于角色的访问控制可以使用上图来描述整个使用的过程，共分 3 步走

1、定义权限树中的各个节点：操作(operations)、任务(tasks)和角色(roles)，创建出它们之间的包含关系，即创建出权限树

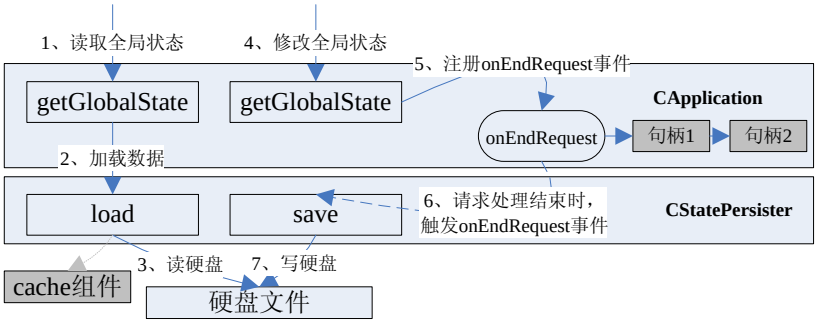
2、给用户分配权限，每个用户可以拥有权限树中的 0 个到对个节点所代表的权限

3、验证用户是否拥护每个权限，为了加快验证速度，该组件对权限验证做了优化。比如用户 Jerry 拥有权限节点 3 和 4，现在要验证用户是否拥有节点 9 所代表的权限。Yii 的权限组件不是遍历所有节点 3 和 4 的所有孩子节点来进行验证的，而是遍历所有节点有的父节点来进行验证，即只要遍历节点 9、6、3 这三个节点就行了，这比遍历节点 3 和 4 的所有孩子节点来速度要快。

3.6、全局状态组件

该组件(CStatePersister)用于存储持久化信息，这些信息会被序列化后保存到 application.runtime.state.bin 文件中，该组件提供了 2 个接口：

接口名	用途
load()	从硬盘(cache 组件)中读取文件信息，反序列化
save()	将信息进行序列化，然后保存为硬盘文件



CApplication 使用该组件来进行持续存储，通过上图大致可以描述它们之间的交互过程，通过调用 app 的 getGlobalState 和 setGlobalState 可以对全局状态进行操作。第 2 步加载数据的时候可以从 cache 中加载，这是对硬盘文件的缓存，可以加快 load 数据的速度。第 4 步修改全局状态，不会触发立刻写硬盘，所有的修改操作都是在 onEndRequest 事件触发的时候一起写硬盘的，即在请求快退出的时候写硬盘，从而加快的保存的速度。

CStatePersister 通过硬盘文件对数据进行持久化，如果是多台服务器就需要对硬盘文件进行同步。所以如果是多台服务器，应该将持久化的数据保存到一个中心点上，比如数据库，对 CStatePersister 进行派生，实现数据库的读写操作即可。

4、控制器层

4.1、Action

每个 http 请求最后都是交给一个 Action 来处理，“Url 路由组件”会将请求路由到某个控制器的某个 Action，每个 Action 都有一个唯一的名字——ActionID。一般会把将业务逻辑相近的 Action 放在同一个 Controller 中，它们一般使用相同的页面布局(layout)和模型层。有 2 种定义 Action 的方法：

1、外部 Action 对象：定义单独的 Action 类，以 CAction 为基类

2、内部 Action 对象：在 Controller 中定义以 action 打头的成员函数

Yii 使用 CAction 为这 2 种方式定义的 Action 封装了一致的接口——即这 2 种方法定义的 Action 都存在一个 run 方法：

1、对于定义单独的 Action 类——外部 Action，重写父类的 run 方法即可

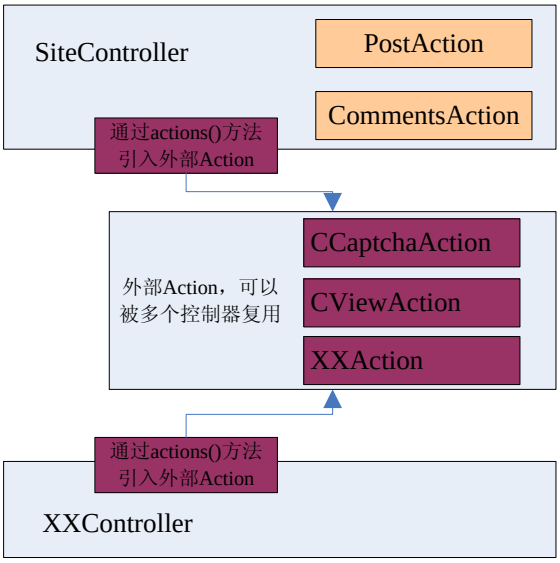
2、对于定义以 action 打头的成员函数——内部 Action，Yii 使用 CInlineAction 做了一层代理，CInlineAction 的 run 方法最后会调用的控制器的以 action 打头的成员函数。

这样的话，控制器的 Action 就全部对象化了，即每个 Action 都是一个对象(内部 Action 对象和外部 Action 对象)。每个控制器通过 actions 方法来制定使用那些外部 Action 对象，并给它们分配唯一的 ActionID，所以外部 Action 可以被多个 Controller 复用。下面以一个例子来说明如何定义内部 Action 和外面 Action，以及调用时的处理流程：

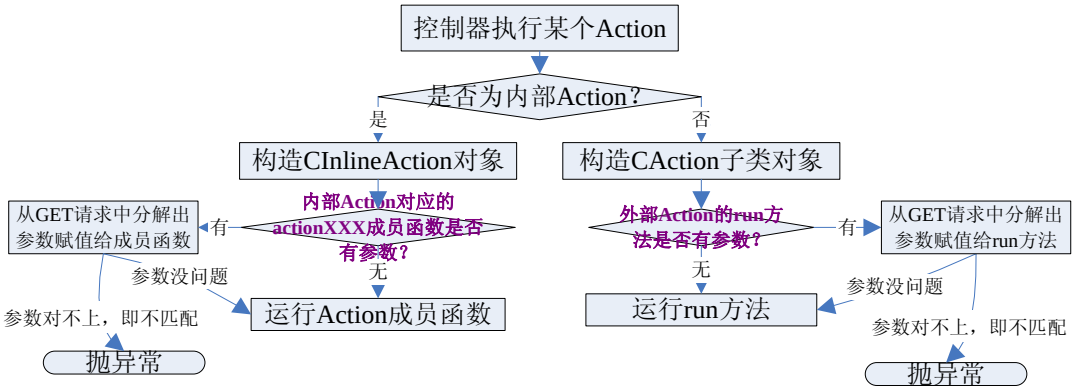
```
class CCaptchaAction extends CAction{//验证码Action
    public function run(){
        ...
        $this->renderImage($this->getVerifyCode());//显示验证码图片
    }
}
class SiteController extends CController{
    public function actions(){//引用外部Action
        return array(
        'captcha'=>'CCaptchaAction',//验证码Action
        'about'=>'CViewAction'      //线上静态页面的Action
        );
    }
    public function actionPost($type, $page=1){//显示帖子的Action
        ...
        $this->render('post', ...);
    }
    public function actionComments($cate, $type){//显示回帖的Action
        ...
        $this->render('comments', ...);
    }
}
```

SiteController 控制器一共定义了 4 个 Action，2 个内部 Action，2 个外部 Action。

使用 actions 方法就可以方便的引入外部 Action，这为 Action 的复用提供了很好的基础。可以使用下图来描述 SiteController 与 Action 之间的关系：



Ok，至此已经说明了控制器与 Action 之间的结构关系，接下来对 Action 对象的 run 方法进行分析，即一个 http 请求过来控制器如何运行 Action 的 run 方法的，大部分处理逻辑是在 CAction 的 runWithParams 和 runWithParamsInternal 两个成员函数中，具体分析见下图：



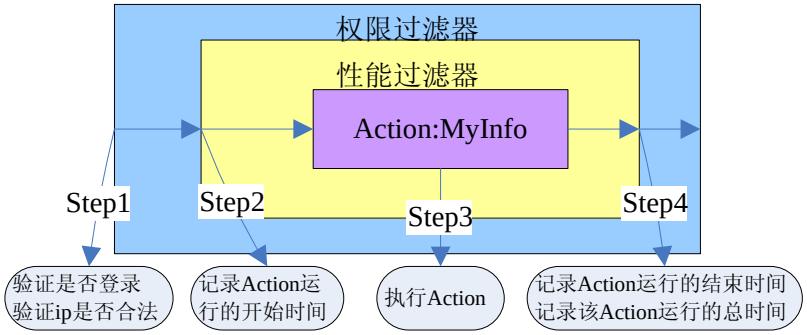
4.2、Filter

Filter(过滤器)与 Action 之间的关系可以比喻成房子和围墙的关系，执行某个 Action 则可以比喻成一个人要去这个房子里取东西，那么他要做 3 个工作：1、翻入围墙；2、进入房间取东西；3、翻出围墙。每个 Action 可以配置多个过滤器，那么这个人就需要翻入和翻出多道围墙。

通过上面的比喻，就基本理解过滤器的用途和情景了。过滤器其实是两段代码：一部分代码在 Action 之前执行；一部分代码在 Action 之后执行。在控制器中的 filter()成员函数中配置各个 Action 需要哪些过滤器——即配置过滤器列表，过滤器的执行顺序就是它们出现在过滤器列表中的顺序。过滤器可以阻止 Action 及后面其他过滤器的执行，一个 Action 可以有多个过滤器。

下面举一个例子，比如只允许北京的登录用户访问“我的个人信息”页面，同时需要记录这个页面的访问性能，即记录这个 Action 的执行时间。对于这种需求，需要定义 2 个过

滤器：1、权限过滤器——用于验证用户访问权限；2、性能过滤器——用于测量控制器执行所用的时间。下图说明的过滤器和 Action 执行的先后顺序。



有 2 中方法来定义过滤器，即内部过滤器和外部过滤器，接下来分别讲解。

内部过滤器：定义为一个控制器类的成员函数，方法名必须以 filter 开头。下面定义一个“ajax 过滤器”，即只允许 ajax 请求才能访问的 Action——filterAjaxOnly()，那么这个过滤器的名称就是 ajaxOnly，如下是这个过滤器的具体定义

```
public function filterAjaxOnly($filterChain){
    if($_SERVER['HTTP_X_REQUESTED_WITH']==='XMLHttpRequest')
        $filterChain->run();//调用$filterChain->run()让后面的过滤器与Action继续执行
    else
        throw new CHttpException(400,'这不是Ajax请求');//到此为止，直接返回
}
```

外部过滤器：定义一个 CFilter 子类，实现 preFilter 和 postFilter 两个成员函数，preFilter 成员函数中的代码在 Action 之前执行，postFilter 成员函数中的代码在 Action 之后执行。下面定义一个“性能过滤器”，用于记录相应 Action 执行使用的时间，具体代码如下，

```
class PerformanceFilter extends CFilter{
    private $_startTime;

    protected function preFilter($filterChain){//Action执行之前运行的代码
        $this->_startTime=time();     //记录Action执行的开始时间
        return true;                  //如果返回false，那么Action就不会被运行
    }

    protected function postFilter($filterChain){//Action执行之后运行的代码
        $spendTime=time()-$this->_startTime;//计算Action执行的总时间
        echo "{$filterChain->action->id} spend $spendTime second\n";//输出Action的执行时间
    }
}
```

为了统一处理内部过滤器和外部过滤器，Yii 使用 CInlineFilter 做了一层代理，CInlineFilter 的 filter 方法会调用的控制器的以 filter 打头的成员函数，即 filterAjaxOnly()。

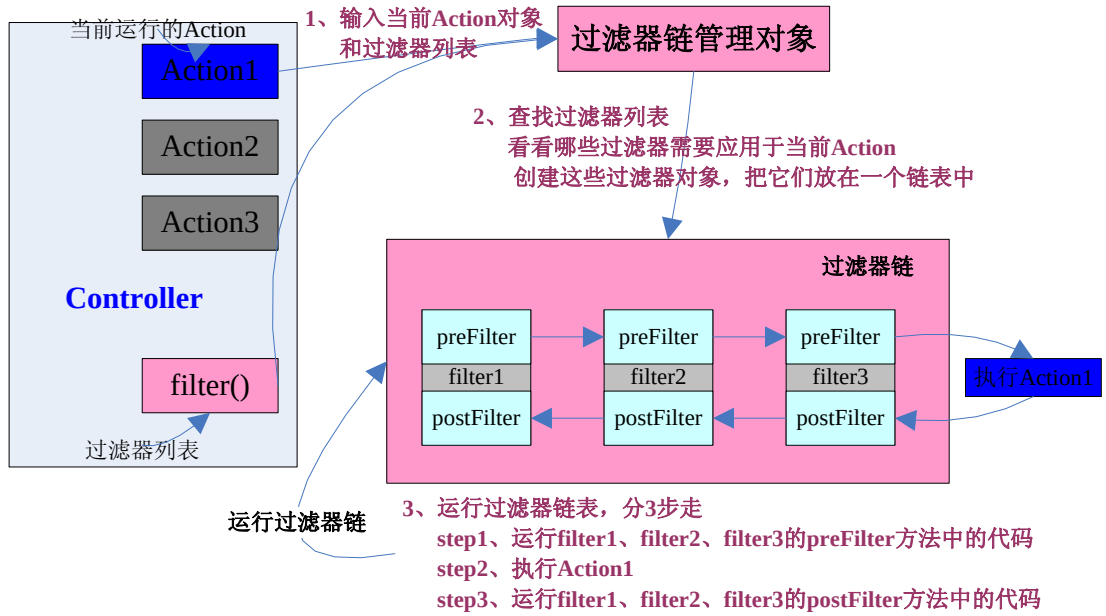
Ok，如上讲解了过滤器的用途和过滤器的定义，接下来讲解：1、如何给各个 Action 配置过滤器；2、在运行 Action 的过程中，过滤器对象是如何被创建和运行的。

如何给各个 Action 配置过滤器呢？只要在控制器中配置一下过滤器列表即可，如下所示，通过 filter()成员函数来配置过滤器列表。对于内部过滤器，使用字符串进行配置；对于外部过滤器，使用数组进行配置。“+”表示这个过滤器只应用于加号后面的 Action；“-”表

示这个过滤器只应用于除减号后面的 Action 以外的所有 Action。

```
class PostController extends CController{
    ......
    public function filters(){//配置过滤器列表
        return array(
     'ajaxOnly+delete,modify',//只有delete和modify两个Action需要配置ajax过滤器
     'postOnly+edit,create',  //只有edit和create两个Action需要配置post过滤器
     array(
         'PerformanceFilter-edit,create',//除了edit和create以外的所有Action都要统计执行时间
         'unit'=>'second',//性能过滤器的共有属性赋值
          ),
      );
    }
}
```

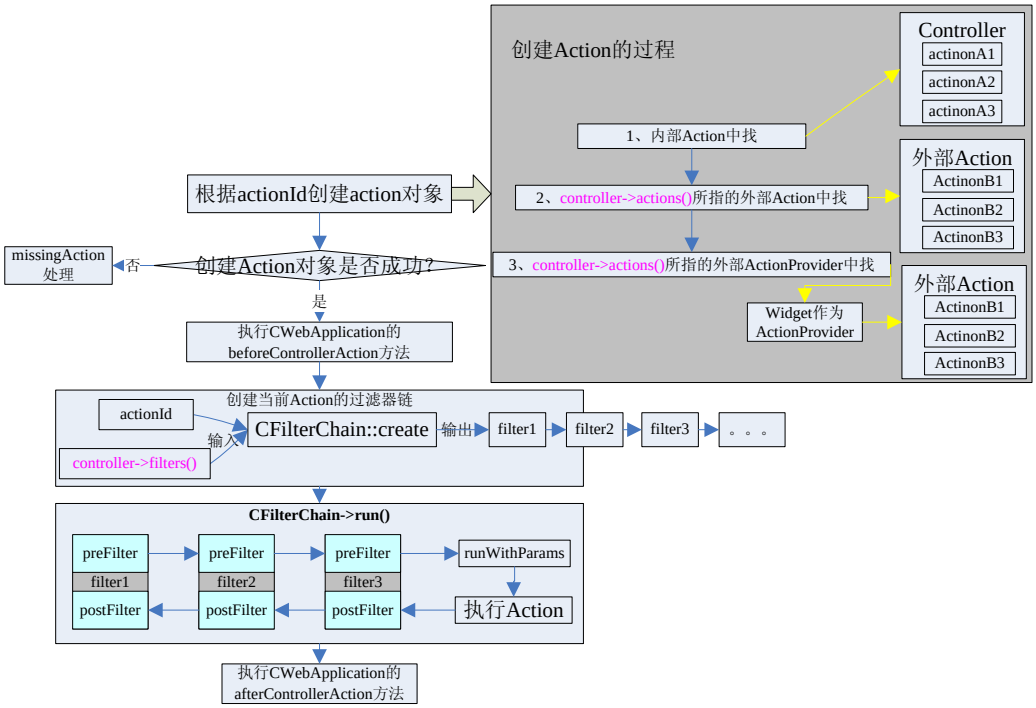
在运行 Action 的过程中，过滤器对象是如何被创建和运行的呢？下图就是答案。



过滤器链管理对象(CFilterChain)起了关键的作用：1、根据控制器提供的过滤器列表创建出应用于当前 Action 的过滤器链，主要就是理解“+”和“-”；2、CFilterChain 提供了一个 run 成员函数，运行过滤器链的时候，run 函数会取出第一个过滤器，第一个过滤器执行完 preFilter 函数后，又会调用 CFilterChain 的 run 函数，此时 run 函数会取出第 2 个过滤器，等等。具体就是上图的调用流程，其实就是形成一个函数调用栈，好比是本节刚开始讲的翻围墙的比喻。

4.3、Action 与 Filter 的执行流程

前两节分表讲了 Action 和 Filter 的定义与使用，它们是控制器的核心，下面把它们串起来看一下，具体可以用下图描述：



主要的步骤可以描述如下：

1、 根据控制器拿到的 ActionID 来创建 Action 对象，ActionID 由 Url 管理组件提供

2、 根据上一步创建的 Action 对象和控制器提供的过滤器列表，创建出应用于当前 Action 的所有过滤器对象，把这些对象放入一个链表

3、 用函数调用栈的方式执行各个过滤器对象和 Action；即执行顺序为：
filter1->preFilter()→filter2->preFilter()→filter3->preFilter()→Action->run()→filter3->postFilter()→filter2->postFilter()→filter1->postFilter()

4.4、访问控制过滤器

访问控制过滤器(CAccessControlFilter)是对控制器的各个 Action 的访问权限进行控制，通过控制器的 accessRules()成员函数可以对各个 Action 的权限进行配置。CAccessControlFilter 支持如下几个访问权限的控制

1、 用户是否登陆进行过滤，对用户名进行过滤
另外：*任何用户、?匿名用户、@验证通过的用户

2、 对用户所应有的角色进行过滤，即基于角色的权限控制

3、 对 ip 段进行过滤

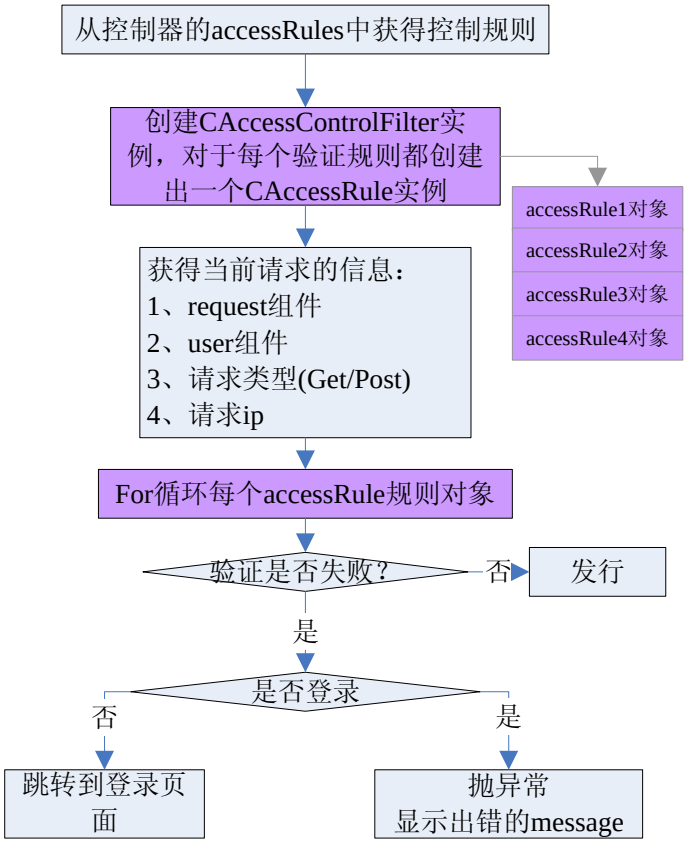
4、 对请求的类型进行过滤，比如 GET、POST

5、 对 php 表达式进行过滤，比如 expression'=>'!$user->isGuest && $user->level==2'

比如对于 modifyAction，只有管理员角色的用户才能访问，并且 ip 段必须在 60.68.2.205 网段，该配置具体如下：

```
public function accessRules()
{
    return array(
        array('allow',
```

```
            'actions'=>array('modify'),
            'roles'=>array('admin'),
            'ips'=>array('60.28.205.*'),
        ),
        array('deny',
            'actions'=>array('modify'),
            'users'=>array('*'),
            'message'=>'需要管理员身份才能访问'
        ),
    );
}
```



上图是 CAccessControlFilter 进行权限控制的处理流程图，整个处理流程主要分为 4 步：

Step1：首先创建出访问控制过滤器对象，再根据控制器的 accessRules()成员函数提供的规则，创建出一堆 accessRule 规则对象，即每条规则对应一个对象；

Step2：获取请求获得当前请求的信息，主要是如下 4 个信息

a、request 组件

b、user 组件

c、请求类型(Get/Post)

d、请求的 ip

Step3：For 循环各个 accessRule 规则对象，即对每个规则单独验证，只要有一个规则拒绝，那么后面的规则就不会再验证。每个规则都会验证各种配置的每个字段：用户信息、角色、

ip、php 表达式等；

Step4：某一个 accessRule 规则对象验证失败，如果用户没有登录则跳转到登录页面，如果已经登录，则抛出异常，显示验证失败规则所对应的 message 字段。

5、模型层

模型层用于数据的查询和更新等操作，主要与数据库打交道，Yii 的模型层可以分为 3 层。

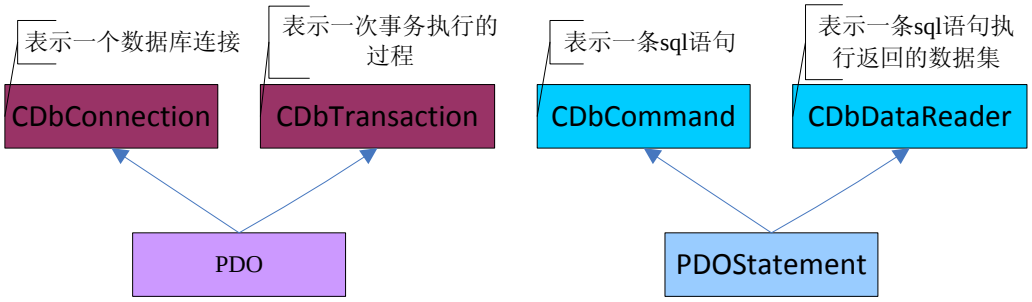
1、DAO 层：对于数据库的直接操作，Yii 使用 php 的 PDO，所以可以支持大多数类型的数据库

2、元数据与 Command 构造器层：通过 CDbSchema 可以获得各个表的结构信息，通过 Command 构造器直接构造出 Command 对象

3、ORM 层：使用 AvtiveRecord 技术实现了对象关系映射，这一层的实现依赖于上两层提供的功能

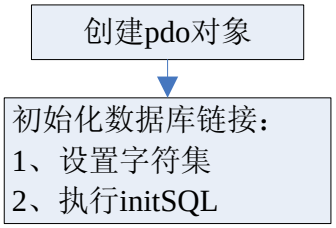
5.1、DAO 层

Yii 的 DAO 层是对 PHP 的 PDO 层的简单封装，PDO 层提供了 2 个类：PDO 和 PDOStatement，Yii 的 DAO 层使用 4 个类对查询的逻辑进行了分离：CDbConnection、CDbTransaction、CDbCommand 和 CDbDataReader，它们的用途可以使用下图来描述：



5.1.1、数据库连接组件

CDbConnection 表示一个数据库的链接，一般将它定义为框架的组件来使用。通过设置它的成员属性 active 可以启动数据库链接，启动链接主要做如下 2 个工作：



重要成员变量和成员函数有：

charset	链接数据使用的字符集(SET NAMES utf8)
createCommand()	创建一个 Command 对象
beginTransaction()	开始一个事务，新建一个事务对象

getCurrentTransaction()	获得当前的事务对象
getSchema()	获得元数据对象
getCommandBuilder()	获得 CommandBuilder 对象
getPdoInstance()	获得原生的 pdo 对象
getLastInsertID()	获得刚插入列的自增主键

5.1.2、事务对象

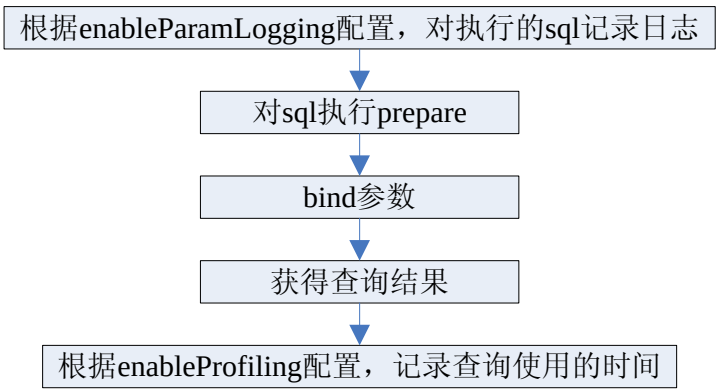
CDbTransaction 表示一个事务的执行对象，一般是由 CDbConnection 的 beginTransaction 成员函数创建，生命周期为一次事务的执行，重要成员变量和成员函数有：

commit()	事务提交
rollback()	事务回滚

5.1.3、Command 对象

CDbCommand 表示一个命令的执行对象，一般是由 CDbConnection 的 createCommand 成员函数创建，生命周期为一次命令的执行。所有的 sql 语句执行都是使用 prepare 模式，这可以提高反复执行的 sql 语句效率，而对于大部分 web 系统，sql 语句一般是固定的并且是多次运行的，所以可以提高模型层的执行速度。

下图分析了内部查询函数 queryInternal 的执行流程，CDbCommand 对于所有的 sql 都进行 prepare 操作，主要是为了提供相似 sql 的执行速度。对每次执行的 sql 语句所花费的时间进行记录。



重要成员变量和成员函数有：

bindParam()	对参数进行绑定
bindValue()	对参数进行绑定
execute()	执行一条 sql
query()	查询，返回 CDbDataReader 对象
queryAll()	查询，返回所有行的数据
queryRow()	查询，返回一行的数据
queryScalar()	查询，返回第一列的数据

queryColumn()	查询，返回第一行第一列的数据

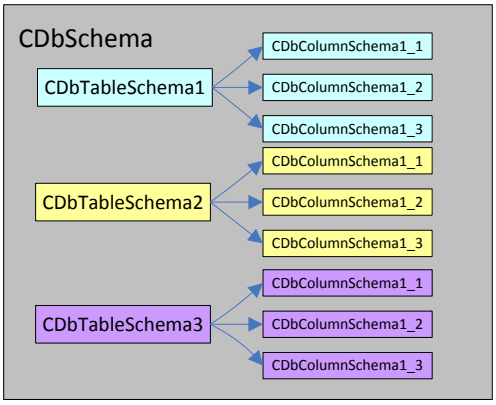
CDbCommand 还提供了构建 sql 的成员函数：select()，from()，join()，where()等等，使用这些函数就无需直接写 sql 语句了，CDbCommand 使用 buildQuery()来构建出最终的 sql 语句。个人认为这些成员函数的用处不大，因为已经很底层了，直接写 sql 就可以了，没有必要使用这些函数来构建 sql 语句。

5.2、元数据与 Command 构造器

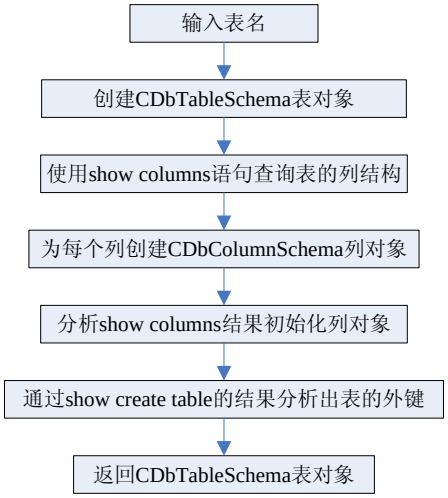
5.2.1、表结构查询

CDbSchema 用于获取各个表的元数据信息：表的各个字段、每个字段的类型、主键信息和外键信息等。

CDbSchema 为 ORM 提供了表的结构信息，CDbSchema 使用 show create table、show columns、show tables 语句来获得表的元数据信息，最终将表结构存储在 3 个类中：CDbSchema、CDbTableSchema、CDbColumnSchema；这 3 个类是一种包含的关系：CDbSchema 代表一个 db，CDbSchema 包含多个 table(CDbTableSchema)，每个 CDbTableSchema 又包含多个 column(CDbColumnSchema)。以下图为例，当前数据库一共有 3 张表组成，每个表又有 3 个列组成：



下图分析了 loadTable()函数的工作流程，该函数输入是 table name，返回是这个表的结构对象——即返回一个 CDbTableSchema 对象：



5.2.2、查询条件对象

CDbCriteria 代表一个查询条件，用来是表示 sql 语句的各个组成部分：select、where、order、group、having 等等，即使用一个对象来代表一条 sql 语句的查询条件，从而使用该对象就可以设定数据库查询和操作的范围。

CDbCriteria 提供了多个成员函数来对查询条件进行设置和修改，多个查询条件也可以进行合并。

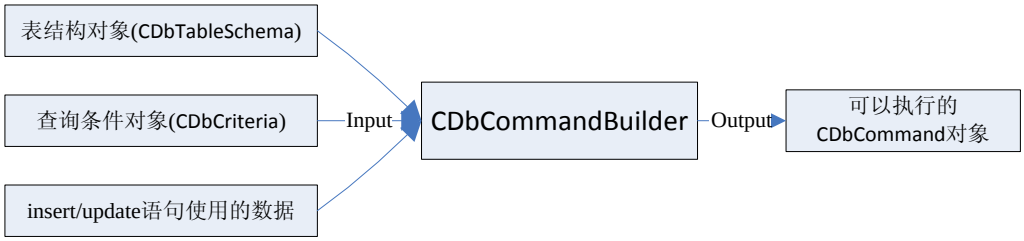
重要成员变量和成员函数有：

select	sql 语句中的 select 语法
condition	sql 语句中的 where 语法
limit	sql 语句中的 limit 语法
offset	sql 语句中的 offset 语法
order	sql 语句中的 order 语法
group	sql 语句中的 group 语法
join	sql 语句中的 join 语法
having	sql 语句中的 having 语法
with	查询关联对象
together	是否拼接成一条 sql 执行，默认为 true
scopes	查询使用的名字空间
addCondition()	添加一个查询条件
addInCondition()	添加 in 语法的查询条件
addColumnCondition()	添加多列匹配的查询条件
compare()	添加比较大小的查询条件
addBetweenCondition()	添加 between 查询条件
mergeWith()	与其它查询对象进行合并

5.2.3、Command 构造器

CDbCommandBuilder 主要有 2 个用途：创建 CDbCommand 对象；创建 CDbCriteria 对象。

创建 CDbCommand 对象功能：输入为表结构对象(CDbTableSchema)、查询条件对象(CDbCriteria)和数据库需要更新的数据，CDbCommand 根据这三个输入拼接出满足需要的 sql，最后创建并返回一个 CDbCommand 对象。



创建 CDbCriteria 对象：输入为表结构对象(CDbTableSchema)和查询使用的数据，

CDbCommand 根据这两个输入拼接出满足需要的查询条件对象或者查询条件语句。

CDbCommandBuilder 为 ActiveRecord 层提供了很好的支持。

5.3、ORM(ActiveRecord)

5.3.1、表的元数据信息

CActiveRelation 表示 ER 图中表与表之间的关系，在 Yii 的 AR 模型中通过 CActiveRelation 对象来找到关联对象。Yii 定义了 5 种关联关系：

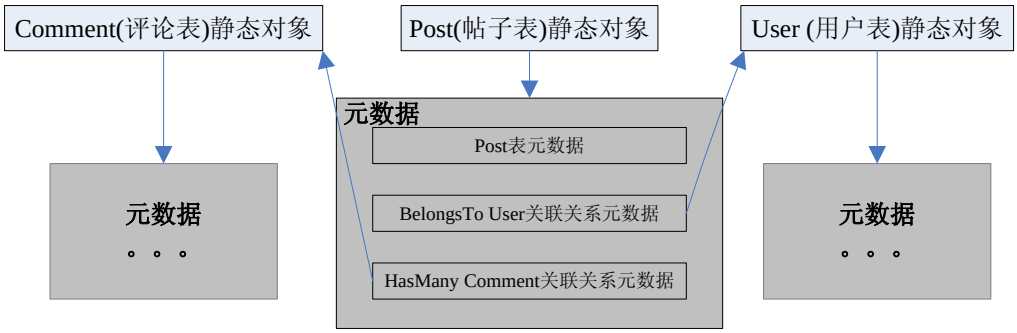
CBelongsToRelation	N:1 关系
CHasOneRelation	1:1 关系
CHasManyRelation	1:N 关系
CManyManyRelation	N:M 关系
CStatRelation	统计关系

CActiveRecordMetaData 表示一个表的元数据信息，它有 2 部分组成：当前表的元数据信息(CDbTableSchema)、与其它表的关联关系元数据(CActiveRelation)。当对当前单表进行更新和操作的时候使用当前表的元数据信息，当对当前表以及关联表进行联合查询的时候使用关联关系元数据(CActiveRelation)。

5.3.2、单表 ORM

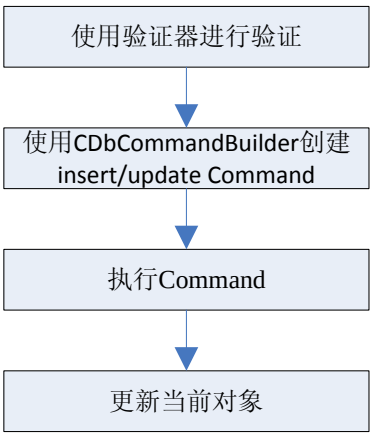
Yii 使用 ActiveRecord 来实现 ORM，一个 CActiveRecord 子类(简称 AR 类)代表一张表，表的列在 AR 类中体现为类的属性，一个 AR 实例则表示表中的一行。 常见的 CRUD 操作作为 AR 的方法实现。因此，可以以一种更加面向对象的方式访问数据。

每个 AR 类都存在一个静态对象，比如 通过 Post::model()可以取到该静态对象，该静态对象主要用于存储表的“元数据”，如下图，静态对象在首次创建的时候会初始化元数据，该元数据会被 AR 类的所有实例使用，即在内存中只存在一份元数据对象(CActiveRecordMetaData)。

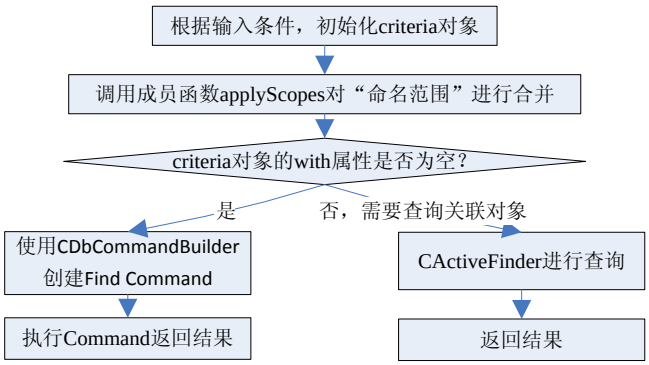


接下来分析对于单表的更新和查找的处理流程，对于关联表的查询会在 CActiveFinder

中进行分析。

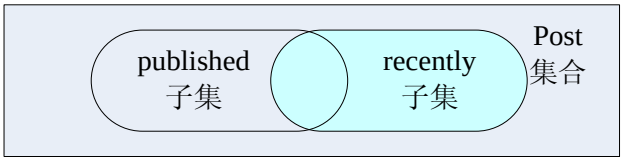


上图 save 成员函数的保存流程。首先对要保存的各个字段进行合法性验证，接着通过 CommandBuilder 创建相应的 Command，最后执行。可以看出，关键是 Command 的合成主要的工作由 CommandBuilder 完成的。



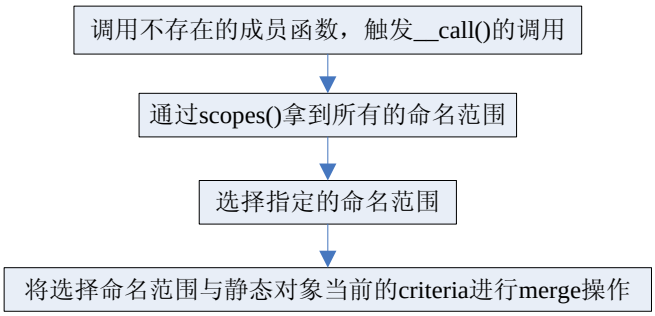
上图分析了 find 系列成员函数的查询流程。第一步将输入条件进行合成为一个 criteria 查询对象；第二步对“命名范围”进行合并，从而形成一个新的查询对象；接下来兵分两路，如果需要查询关联对象，则通过 CActiveFinder 进行查询，如果只是单表查询则通过 CommandBuilder 来创建相应的查询 Command。

“命名范围”表示当前表中记录的一个子集，它最初想法来源于 Ruby on Rails。一个 Post 帖子表为例，我们可以定义 2 个命名范围：1、通过审核的帖子，即 status=1；2、最新发表的帖子，即按最近发表的 5 个帖子。如果将整个 Post 表看成一个集合的话，那么前 2 个命名范围就是它的子集；如下图：




```
public function scopes(){
    return array(
        'published'=>array('condition'=>'status=1',),
         'recently'=>array('order'=>'create_time DESC', 'limit'=>5,),
    );
}
$posts=Post::model()->published()->recently()->findAll();//查找
Post::model()->published()->recently()->delete/update();//删除或者更新
```

使用 scopes 成员函数来定义“命名范围”，如上图。在进行查询、删除、更新之前可以使用命名范围来设定范围，多个命名范围可以链接使用，接下分析命名范围的工作原理。



上图分析了命名范围的工作原理，所有的命名范围最终会与静态对象当前的 criteria 进行合并，从而形成一个更加严格的查询条件。在进行查询或者更新操作的是时候，通过调用成员函数 applyScopes 来获得命名范围合并后的查询条件 criteria。

5.3.3、多表 ORM

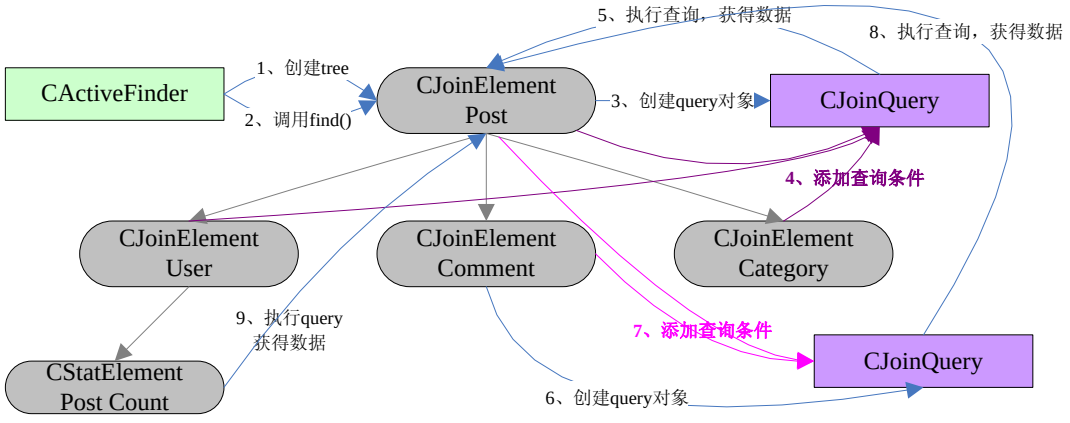
多表 ORM 查询的算法复杂度是 Yii 框架中最大的(不要怕，用多表 ORM 的时候只要知道大概的流程就行)，因为多表的 ORM 映射本身就很复杂，多表 ORM 需要考虑 5 种 ActiveRelation，其次为了优化查询速度，在表做联合的时候还需要考虑使用哪个外键可以使得查询速度更快。

下图是多表 ORM 查询的一个处理流程，需要使用 4 个类：CActiveFinder、CJoinElement、CStatElement、CJoinQuery。

CActiveFinder：查询的发起者，它负责创建查询树和查询查询树

CJoinElement：查询树有多个 CJoinElement 和 CStatElement 组成，给树的每个结点表示一个表结构，因为 CStatElement 表示统计信息，所以它只能作为叶子结点

CJoinQuery：对树中多个结点构建联合查询，即构建 join 语句。执行联合查询语句，将查询的结果返回给 CJoinElement 的根结点



整个查询共分 9 个步骤：

Step1：首先由 CActiveFinder 构建查询结点树，根据模型层提供的 relation()成员函数提供的表之间关系构成查询树，可以通过 with 语法指定多久关联关系。树根代表的是当前查询的表结构，树中其它结点表示的是与树根相关联的其它表。

Step2：CActiveFinder 向查询树发起 find 操作，查询树的树根会先找出是以一对多关系的直接孩子结点(这样可以有效利用索引)，先对这些孩子进行联合查询

Step3：创建出一个 CJoinQuery 对象，该对象用于查询，将树根的查询条件添加到 CJoinQuery 对象中

Step4：遍历 Step2 中的一对多关系的孩子，把他们的查询条件添加到 CJoinQuery 对象中，次数 CJoinQuery 已经获得的所有的查询条件

Step5：对上 2 步添加进 CJoinQuery 对象的节点构建联合查询语句，执行这条语句，将执行的结果保存到树根节点中

Step6：遍历树根剩余的直接孩子，剩下的孩子就是多对一或者多对多的关系了。此时创建一个新的 CJoinQuery 对象

Step7：将当前节点的查询条件和树根节点的查询添加到 CJoinQuery 对象中

Step8：构建 Step7 添加的查询条件，对着 2 个节点构建联合查询语句，执行这条语句，将执行的结果保存到树根节点中

Step9：最后还剩下一个统计关系节点，因为树根节点所对应的表数据已经全部查询出来，所以 CStatElement 可以直接从树根节点中获得数据，构建 group by 的 sql 语句来直接执行，就无需使用 CJoinQuery 了，执行构建 group by 语句，将执行的结果保存到树根节点中

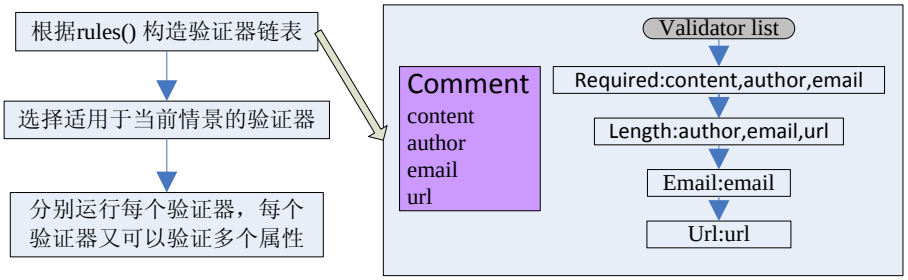
Ok，通过如上 9 步查询可以获得 ORM 关联表的全部信息，其中前 5 步用于查询一对多的关联表，Step6 到 Step8 用于查询多对一和多对多关系表，Step9 则一步用于查询统计关系表。

5.3.4、CModel 与 CValidator

验证器用于对 CModel 的各个属性进行验证，从而保证存储数据的合法性。我们以回帖表 Comment 的模型层为例，它的验证规则定义如下：

```
public function rules(){
    return array(
        array('content, author, email', 'required'),
        //帖子内容、作者、邮箱不能为空
        array('author, email, url', 'length', 'max'=>128),
        //作者、邮箱、URL的长度最大为128
        array('email','email'),
        //邮箱必须合法
        array('url','url'),
        //url地址必须合法
    );
}
```

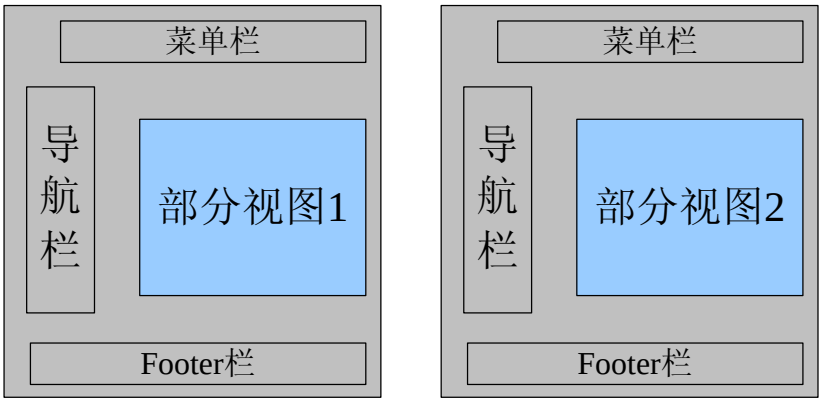
CModel 的 validate()成员函数的出来流程为：



6、视图层

6.1、视图渲染流程

在 MVC 架构中，View 主要是用于展示信息的。Yii 中的视图层文件由 2 部分组成：布局视图、部分视图。web 系统的大部分页面都存在相同的元素：logo、菜单、foot 栏等，我们把这些相同的元素组成的视图文件称为布局视图，一般 web 系统需要 2 个布局，即前台布局和后台布局，前台布局是给用户看的，后台布局是给管理员看的。每个页面所独有的部分视图称为部分视图



可以使用上图进行描述，我们将菜单栏、导航栏和 Footer 栏放到布局文件中，即所有页面复用一个布局文件，然后每个页面(Action)有各自的部分视图文件。

接下来看一下视图文件的存放路径。WebApp 可以配置视图文件路径和布局文件路径同时还会指定一个默认的布局文件；每个 Controller 的视图文件存放在 WebApp 指定的视图路径下，以 Controller 的名字职位后缀，Controller 还可以指定自己使用哪个布局文件。

WebApp 成员属性	**说明**
viewPath	用于指定视图文件路径，所有的视图文件必须在这个文件下 默认 protected/views
layoutPath	用于指定布局文件路径，所有的布局文件必须在这个文件下 默认 protected/views/layouts，该路径下有：main.php、column.php
viewPath	用于指定系统视图文件路径，默认 protected/views/system
layout	指定默认使用的布局文件，默认为 main

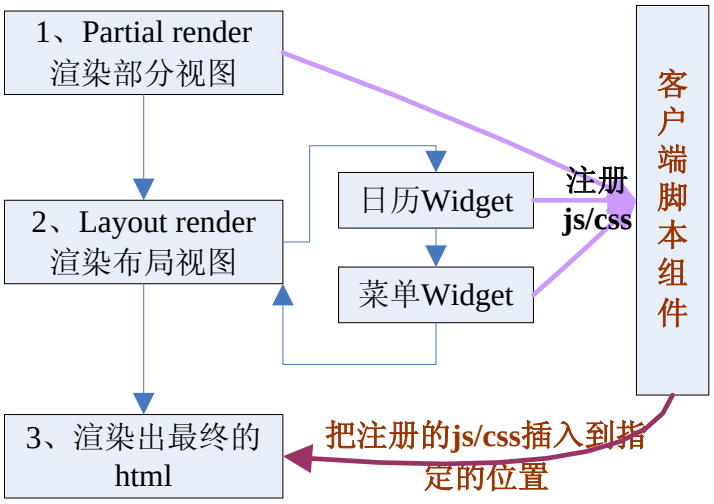
比如当前正在执行 PostController 的 modifyAction，PostController 指定使用 column 布局，那么这个请求所使用的布局文件为 protected/views/layouts/column.php，视图文件为 protected/views/post/modify.php。

视图层中还有 2 个重要的概念：客户端脚本组件、Widget。

客户端脚本组件：该组件用于管理客户端脚本(javascript 和 css)，可以通过该组件向视图中添加 javascript 和 css，客户端脚本组件统一管理这些代码，在页面输出的最后一步对客户端脚本(javascript 和 css)进行渲染。

Widget：又称小物件，通过 Widget 可以对页面进行模块化，Widget 可以看成是一个没有布局的控制器。通过 Widget 可以把公用的页面元素进行复用，比如：Menu Widget、列表

Widget、表格 Widget、分页 Widget 等等。



视图层的渲染分 3 个步骤完成：

Step1：渲染部分视图，即渲染每个页面各自特有的视图片断；

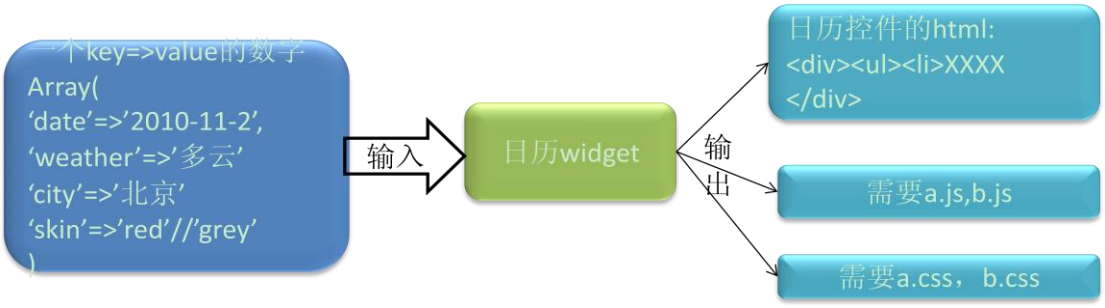
Step2：将渲染布局视图，即即渲染每个页面共有的页面元素，同时将 Step1 的结果插入到布局视图中。在 Step1 和 Step2 中，可能还需要渲染 Widget，比如日历 Widget、菜单 Widget 等。这 2 个步骤中可以注册自己使用了哪些 js 和 css；

Step3：渲染 js 和 css。将前 2 步注册的 js 和 css 添加到 html 页面的制定位置。

6.2、Widget

在 windows(MFC,Delphi,游戏)开发过程中，有很多小控件(下拉菜单/按钮/日历/人物)可以使用，不需要从头开发。小物件(Cwidget) 的设计思想与其类似，主要是可以为了提升视图层的开发速度，它将页面看成是有多个可以复用的控件组成，从而提高了页面控件的复用性和可维护性。

下面说一个日历组件的例子，日历组件的输入是一个数组，如下图，输出分 2 部分：html 片断；向 clientScript 注册这个日历需要使用的 js 和 css



如下是两种创建 Widget 的方法：

```
<?php $this->beginWidget('path.to.WidgetClass'); ?>
...可能会由小物件获取的内容主体...
<?php $this->endWidget(); ?>
```

或者

```
<?php $this->widget('path.to.WidgetClass'); ?>
```

Yii 自带了 20 个左右的常用 widget，开源社区目前也贡献了 100 多个 widget。小物件可以配置多套皮肤(国庆用红色的，清明用灰色的)。

6.3、客户端脚本组件

大部分页面需要使用 js 和 css，一般情况下我们是直接在页面中引用 js 文件或者直接写 js 代码。客户端脚本组件(CClientScript)用于管理视图层的 JavaScript 和 CSS 脚本，可以根据方便的管理和维护 js 和 css：

1、 注册 js、css 文件

通过 ClientScript 组件的提供的注册接口来向 html 代码中注册 js 和 css 文件，这样的话不用直接修改模板文件，为 widget 的开发提供了很好的基础——视图层模块化。

2、 注册 js、css 代码

我们希望在 dom ready 的时候，对某些 dom 绑定时候；那么需要在 html 页面最下面或者在某个 js 文件中写这个代码。通过 ClientScript 组件的提供的注册接口来向 html 代码中注册 js 代码，目前可以知道这个代码嵌入到什么地方(domready/onload)。

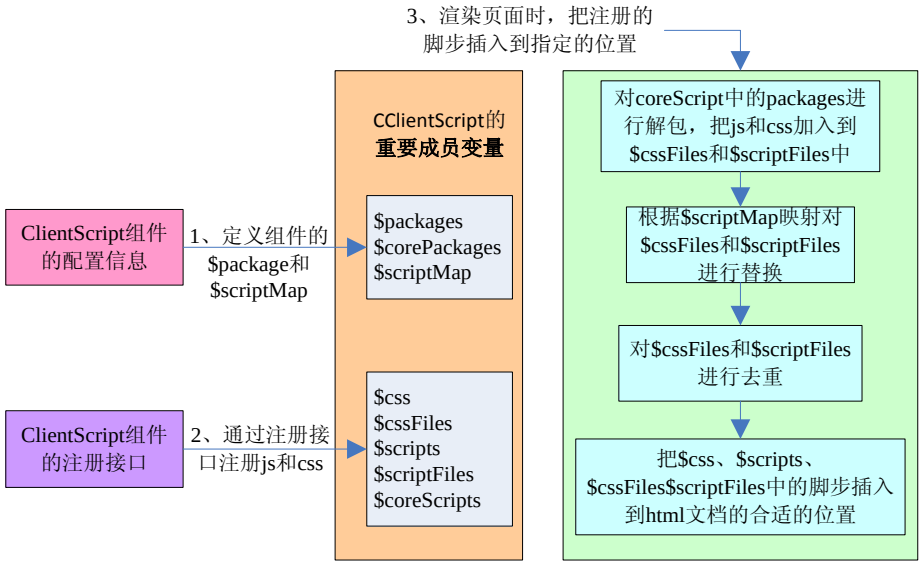
3、 解决 Js、css 代码依赖、代码合并、代码压缩、代码版本控制

比如有 2000 个页面中使用的 edit.js 这个文件，而它又使用了 jquery.cookie.js，那么这 2000 个页面需要同时引用这 2 个 js，如果有一天 edit.js 使用了 jquery.tab.js 中的功能，那么需要在这 2000 个页面中都增加对 jquery.tab.js 的引用，**老大，2000 个啊！**对于代码合并、代码压缩、代码加版本号都是是类似的，一旦发生修改，需要修改大量的模板页面。Yii 通过 ClientScript 组件的包依赖数组和 scriptMap 数组就可以解决这些问题，简单来说就是只要改一下这 2 个 php 的数组即可。

Ok，下面我们先看一下 ClientScript 组件的重要的成员属性和成员函数，主要归为 3 类：初始化成员属性、脚本保存成员属性、注册接口。初始化成员属性用于配置 package 和脚本映射；脚本保存成员属性用于记录注册了哪些脚本，这些脚本分别注册到什么位置；注册接口用于外部调用，将脚本提交给 ClientScript 组件。

初始化成员属性	$packages	保存用户脚本(js/css)包和包依赖关系
	$corePackages	保存框架脚本包和包依赖关系
	$scriptMap	对脚本进行映射，从而实现脚本的压缩、合并、版本控制
保存脚本的成员属性	$css	保存 css 代码片断
	$cssFiles	保存 css 文件的 url
	$scripts	保存 js 代码片断
	$scriptFiles	保存 js 文件的 url
	$coreScripts	保存使用了哪些 package
注册接口	registerCss($id,$css,$media)	注册 css 代码
	registerCssFile($url,$media)	注册 css 文件
	registerScript($id,$script,$position)	注册 js 代码
	registerScriptFile($url,$position)	注册 js 文件
	registerPackage($name)	注册 package
	registerCoreScript($name)	注册 package

ClientScript 组件在使用的过程中分三步走：对组件进行配置、调用注册接口注册脚本、渲染插入脚本。可以使用下图来形象的描述，接下来详细分析每一步的工作。



Step1、对 ClientScript 组件进行配置，下面是该组件的配置数组

```
'clientScript'=>array(
    'class'=>'CClientScript',
    'packages'=>array(//用户的js、css代码包结构
        'edit'=>array(
            'js'=>array('edit.js'),
            'depends'=>array('yiitab'),
        ),
        'favor'=>array(
            'js'=>array('favor.js'),
            'depends'=>array('jquery'),
        )
    ),
    'corePackages'=>array(//框架的js、css代码包结构
        'jquery'=>array(
            'js'=>array(YII_DEBUG ? 'jquery.js' : 'jquery.min.js'),
        ),
        'yii'=>array(
            'js'=>array('jquery.yii.js'),
            'depends'=>array('jquery'),
        ),
        'yiitab'=>array(
            'js'=>array('jquery.yiitab.js'),
            'depends'=>array('jquery'),
        )
        ...
```

```
    ),
    'scriptMap'=>array(//对js代码进行映射
        'edit.js'=>'edit.js?version=1.0',//代码加版本号
        'favor.js'=>'favor.min.js',//代码压缩
        '*.js'=>'common.min.js?version=1.1',//代码合并、压缩、加版本号
    )
)
```

如上配置数组定义了 packages 和 corePackage 进行赋值，通过 scriptMap 可以对 js 和 css 进行压缩、合并、加版本信息

Step2、调用注册接口，注册脚本

```
//注册favor package
Yii::app()->clientScript->registerPackage('favor');
//将edit.js注册到页面的最下部
Yii::app()->clientScript->registerScriptFile('edit.js', CClientScript::POS_END);
//在页面onload的时候执行一段js代码
Yii::app()->clientScript->registerScript(
    'id_load',
    'alert("the page is load!")',
    CClientScript::POS_LOAD
);
//在dom ready的时候执行一段js代码(dom ready事件要早于onload事件)
Yii::app()->clientScript->registerScript(
    'id_ready',
    'alert("dom is ready!")',
    CClientScript::POS_READY
);
```

javascript 可以注册到 html 中的 5 个位置，具体见下表

注册的位置	说明
POS_HEAD	把 js 文件或代码注册到<title>标签之前
POS_BEGIN	把 js 文件或代码注册到<body>标签之后
POS_END	把 js 文件或代码注册到</body>标签之前
POS_LOAD	在触发 onload 事件时，执行执行注册的 js 代码
POS_READY	在 dom ready 的时候，执行执行注册的 js 代码

Step3、渲染页面时，将注册的 js、css 插入到指定的位置。从注册的 package 中获得 js 和 css；对 js 和 css 进行 map 映射，最后在输出的 html 文档中进行正则匹配，嵌入 js 和 css 代码，如上两步最后输出的 html 文档为：

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <meta name="language" content="en" />
    <script type="text/javascript" src="/common.min.js?version=1.1"></script>
    <script type="text/javascript" src="/favor.min.js"></script>
    <title>ClientScript组件学习</title>
</head>
<body>
<div>
    ....
</div>
<script type="text/javascript" src="'edit.js?version=1.0"></script>
<script type="text/javascript">
/*<![CDATA[*/
jQuery(function($) {
alert("dom is ready!")
});
jQuery(window).load(function() {
alert("the page is load!")
});
/*]]>*/
</script>
</body>
```

registerPackage('favor')

registerScriptFile('edit.js',ClientScript::POS_END)

registerScript('id_ready','alert("dom is ready!")',CClientScript::POS_READY)

registerScript('id_load','alert("the page is load!")',CClientScript::POS_LOAD)