Yii 框架学习笔记

作者：许明

前言

笔记中的源代码基于 Yii1.1.22 版本。

下载：https://github.com/kaimixu/yii-book

联系：kaimix@126.com

目 录

第一章 请求的处理流程

1 项目入口脚本

yii 项目的入口脚本通过包含以下内容：

index.php:

```
// change the following paths if necessary
$yii=dirname(__FILE__).'/../../framework/yii.php';
$config=dirname(__FILE__).'/../config/main.php';

// remove the following lines when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);
// specify how many levels of call stack should be shown in each log message
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);

require_once($yii);
Yii::createWebApplication($config)->run();
```

首先包含 Yii 框架的引导文件 yii.php，根据配置文件 main.php 创建一个 Web 应用并运行。

YII_DEBUG 常量设置应用运行在生产或调试模式下。

YII_TRACE_LEVEL 常量设置调试模式下记录日志的堆栈的层级数。

2 类的自动加载

框架引导文件 yii.php 包含 Yii 类，首先判断 YiiBase 类是否已定义，如果未定义就包含 YiiBase.php 文件。

yii.php：

```
<?php
if(!class_exists('YiiBase', false))
	require(dirname(__FILE__).'/YiiBase.php');

class Yii extends YiiBase
{
}
```

YiiBase.php：

```
<?php

defined('YII_BEGIN_TIME') or define('YII_BEGIN_TIME',microtime(true));
defined('YII_DEBUG') or define('YII_DEBUG',false);
```

```
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',0);
defined('YII_ENABLE_EXCEPTION_HANDLER') or define('YII_ENABLE_EXCEPTION_HANDLER',true);
defined('YII_ENABLE_ERROR_HANDLER') or define('YII_ENABLE_ERROR_HANDLER',true);
defined('YII_PATH') or define('YII_PATH',dirname(__FILE__));
defined('YII_ZII_PATH') or define('YII_ZII_PATH',YII_PATH.DIRECTORY_SEPARATOR.'zii');

class YiiBase {...}
spl_autoload_register(array('YiiBase','autoload'));
require(YII_PATH.'/base/interfaces.php');
```

YiiBase.php 文件主要做了四件事情，1、定义常量，2、定义 YiiBase 类，3、调用 spl_autoload_register 函数注册 autoload 自动装载函数，4、加载 framework/base/interface.php 文件； 这里我们重点关注自动装载函数的注册。

YiiBase.php::autoload：

```
public static function autoload($className,$classMapOnly=false)
{
    // 通过向 autoloaderFilters 中添加 callback 可实现自定义 autoload
    // 如，在 index.php 文件中定义语句：Yii::$autoloaderFilters=['MyAutoload'];
    foreach (self::$autoloaderFilters as $filter)
    {
          // Yii::$autoloaderFilters = [['MyAutoloadClass', 'MyAutoloadHandler']];
        if (is_array($filter)
            && isset($filter[0]) && isset($filter[1])
            && is_string($filter[0]) && is_string($filter[1])
            && true === call_user_func(array($filter[0], $filter[1]), $className)
        )
        {
            return true;
        } // Yii::$autoloaderFilters = ['MyAutoload'];
        elseif (is_string($filter) && true === call_user_func($filter, $className))
        {
            return true;
        } // filter 是可调用的
        elseif (is_callable($filter) && true === $filter($className))
        {
            return true;
        }
    }

    // use include so that the error PHP file may appear
    if(isset(self::$classMap[$className])) // 类映射表中存通过 Yii::import 导入的文件，存在就包含文件
```

```
        include(self::$classMap[$className]);
    elseif(isset(self::$_coreClasses[$className])) //  检查核心类映射表中是否存在 className
        include(YII_PATH.self::$_coreClasses[$className]);
    elseif($classMapOnly) //  是否仅检查类映射表
        return false;
    else
    {
        // include class file relying on include_path
        if(strpos($className,'\\')===false)  //  未使用命名空间
        {
             // $enableIncludePath  默认等于 true
            if(self::$enableIncludePath===false)
            {
                 // Yii::import 函数会将_includePaths 设置为 include_path 配置项的值
                foreach(self::$_includePaths as $path)
                {
                    $classFile=$path.DIRECTORY_SEPARATOR.$className.'.php';
                    if(is_file($classFile))
                    {
                        include($classFile);
                        if(YII_DEBUG && basename(realpath($classFile))!==$className.'.php')
                            throw new CException(Yii::t('yii','Class name "{class}" does not match class file "{file}".', array('{class}'=>$className,   '{file}'=>$classFile,)));
                        break;
                    }
                }
            }
            else
                include($className.'.php');
        }
        else   // class name with namespace in PHP 5.3
        {    //  将命名空间转换成路径并 include 文件
            $namespace=str_replace('\\','.',ltrim($className,'\\'));
            //  将别名转换为路径
            if(($path=self::getPathOfAlias($namespace))!==false && is_file($path.'.php'))
                include($path.'.php');
            else
                return false;
        }
        //  检查文件是否 include 成功
        return class_exists($className,false) || interface_exists($className,false);
    }
    return true;
```

```
}
```

YiiBase::autoload()首先会通过 autoloaderFilters 数组提供的 callback 函数加载类，加载成功后直接返回，通过设置 autoloaderFilters 数组可跳过 yii 的自动装载机制。

接着检查类映射表中是否存在 className。

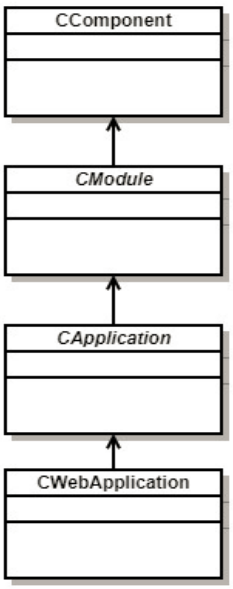
检查 className 是否为命名空间，如果是将 className 转换为路径并 include，否则在 include_path 配置项指定的目录下查找类文件。

3 创建应用实例并初始化

3.1 创建 Web 应用对象

```
Yii::createWebApplication($config) // index.php

// YiiBase.php
public static function createWebApplication($config=null)
{
    return self::createApplication('CWebApplication',$config);
}
public static function createApplication($class,$config=null)
{
    return new $class($config);
}
```

CWebApplication 类的继承关系如图所示：



CWebApplication 类没有自定义构造函数，当创建对象时默认会自动调用父类的构造函数。

CApplication::__construct：

```
public function __construct($config=null)
{
    Yii::setApplication($this);

    // set basePath as early as possible to avoid trouble
    // $config 是文件路径字符串，包含文件
    if(is_string($config))
        $config=require($config);
    if(isset($config['basePath']))
    {
        $this->setBasePath($config['basePath']);
        unset($config['basePath']);
    }
    else // 如果未设置 basePath 配置项默认为项目根目录下的 protected， protected/controller、
protected/model
        $this->setBasePath('protected');
    // 设置别名 application=$basePath
    Yii::setPathOfAlias('application',$this->getBasePath());
    Yii::setPathOfAlias('webroot',dirname($_SERVER['SCRIPT_FILENAME']));
    if(isset($config['extensionPath']))
    {
         // 设置别名 ext
        $this->setExtensionPath($config['extensionPath']);
        unset($config['extensionPath']);
    }
    else
        Yii::setPathOfAlias('ext',$this->getBasePath().DIRECTORY_SEPARATOR.'extensions');
    if(isset($config['aliases']))
    {
         // 自定义别名
        $this->setAliases($config['aliases']);
        unset($config['aliases']);
    }

    $this->preinit();
    // 初始化错误句柄
    $this->initSystemHandlers();
    // 调用 CWebApplication::registerCoreComponents 注册核心应用组件，如：db、log、messages、
urlManager 以及 request 等
    $this->registerCoreComponents();
```

```
    // 解析配置文件
    $this->configure($config);
    // 注册应用行为
    $this->attachBehaviors($this->behaviors);
    // 根据 preload 属性创建组件，默认情况下，组件注册后并不会被加载创建，preload 属性中的组件被会提前创建
    $this->preloadComponents();
    // 调用初始化函数, CWebApplication:创建 request 组件, CConsoleApplication:查找所有命令
    $this->init();
}
```

Yii::setApplication($this)将应用对象保存到静态变量中，后续可通过 Yii::app()访问。

调用 initSystemHandlers()函数设置默认的异常和错误处理程序，当设置异常处理程序后默认会将异常输出到浏览器，handleException 和 handleError 函数后面章节再分析。

CApplication::initSystemHandlers:

```
protected function initSystemHandlers()
{
     // YiiBase.php 文件定义了此宏，默认为 true
    if(YII_ENABLE_EXCEPTION_HANDLER)
        set_exception_handler(array($this,'handleException'));
         // YiiBase.php 文件定义了此宏，默认为 true
    if(YII_ENABLE_ERROR_HANDLER)
        set_error_handler(array($this,'handleError'),error_reporting());
}
```

3.2 解析配置项

除前面 unset 的配置项外，CApplication::configure($config)调用 CComponent::__set 魔术方法解析配置项，假设$config 内容如下：

```
array(
            'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
            'name'=>'My Web Application',

            // preloading 'log' component
            'preload'=>array('log'),

            // autoloading model and component classes
            'import'=>array(
```

```
                'application.models.*',
                'application.components.*',
        ),

        'onBeginRequest' => array('EventClass', 'EventMethod') ,

        // application components
        'components'=>array(...),

        // application-level parameters that can be accessed
        // using Yii::app()->params['paramName']
        'params'=>array(
                'pageSize' => 20,
        ),
);
```

我们来看看 yii 框架是如何解析配置项的，首先分析下 CComponent::__set 魔术方法：

CComponent.php:

```
public function __set($name,$value)
{
    $setter='set'.$name;
    // 检查类是否存在属性 seter 方法，如：setImport，如果存在直接调用 set 方法
    if(method_exists($this,$setter))
        return $this->$setter($value);
    // 注册以'on'开头的事件，如：onBeginRequest 事件，事件会被添加到链表的尾部
    elseif(strncasecmp($name,'on',2)===0 && method_exists($this,$name))
    {
        // duplicating getEventHandlers() here for performance
        $name=strtolower($name);
        if(!isset($this->_e[$name]))
            $this->_e[$name]=new CList;
        return $this->_e[$name]->add($value);
    }
    elseif(is_array($this->_m)) // _m 存放行为对象
    {
        // 遍历绑定的行为
        foreach($this->_m as $object)
        {
            // 行为类中是否存在属性或者 getter 函数
            if($object->getEnabled()&&(property_exists($object,$name) || $object->canSetProperty($name)))
                return $object->$name=$value;
        }
    }
```

```
    // 配置项无法解析时抛出异常
    if(method_exists($this,'get'.$name))
        throw new CException(Yii::t('yii','Property "{class}.{property}" is read only.',
            array('{class}'=>get_class($this), '{property}'=>$name)));
    else
        throw new CException(Yii::t('yii','Property "{class}.{property}" is not defined.',
            array('{class}'=>get_class($this), '{property}'=>$name)));
}
```

name 配置项设置应用程序名称，直接赋值给 CApplication 类中的 name 属性中，可通过 Yii::app()->name 访问。

preload 配置项

默认添加的组件只有用到时才创建，preload 配置项可提前创建组件，配置项直接赋值给 CModule 类中的 preload 属性中，preload=array(log)配置项创建 log 组件对象。

import 配置项

调用 CModule::setImport 方法导入 import 配置中的类或目录，当导入类时 import 方法跟 include 和 require 不一样，它更高效，导入(import)的类只有在第一次被引用时才真正包含进来。导入目录相当于将目录添加到 include 路径中，并支持多个目录导入，后导入的目录会添加在 include 路径的前面。导入类和目录时可使用别名，并且支持 namespace 格式，如：

```
application.components.GoogleMap     // 导入 GoogleMap 类
application\components\GoogleMap    // 导入 GoogleMap 类
application.components.*   // 导入目录
```

setImport 方法循环调用 Yii::import 导入类和路径，当形参 forceInclude 为 false 时 Yii::import 将导入的类存放到 classMap 属性中，类在第一次被引用时通过 autoloader 包含进来。

Yii::import:

```
public static function import($alias,$forceInclude=false)
{
    if(isset(self::$_imports[$alias]))   // previously imported
        return self::$_imports[$alias];

    if(class_exists($alias,false) || interface_exists($alias,false))
        return self::$_imports[$alias]=$alias;

    // 找最后一个'\'
    if(($pos=strrpos($alias,'\\'))!==false) // a class name in PHP 5.3 namespace format
```

```
    {
        // $alias="application\components\GoogleMap", $namespace="application.components"
        $namespace=str_replace('\\','.',ltrim(substr($alias,0,$pos),'\\'));
        // 通过别名查找路径
        if(($path=self::getPathOfAlias($namespace))!==false)
        {
            // 类文件的绝对路径
            $classFile=$path.DIRECTORY_SEPARATOR.substr($alias,$pos+1).'.php';
            if($forceInclude) //强制 include，此种情况 import 方法跟 include、require 没有区别
            {
                if(is_file($classFile))
                    require($classFile);
                else
                    throw new CException(Yii::t('yii','Alias "{alias}" is invalid. Make sure it points to an existing PHP file and the file is readable.',array('{alias}'=>$alias)));
                self::$_imports[$alias]=$alias;
            }
            else // 自动装载时会首先从$classMap 中查找，参考 YiiBase::autoload 方法
                self::$classMap[$alias]=$classFile;
            return $alias;
        }
        else
        {
            // try to autoload the class with an autoloader
            if (class_exists($alias,true))
                return self::$_imports[$alias]=$alias;
            else
                throw new CException(Yii::t('yii','Alias "{alias}" is invalid. Make sure it points to an existing directory or file.',
                    array('{alias}'=>$namespace)));
        }
    }

    if(($pos=strrpos($alias,'.'))===false)  // a simple class name
    {
        // try to autoload the class with an autoloader if $forceInclude is true
        if($forceInclude && (Yii::autoload($alias,true) || class_exists($alias,true)))
            self::$_imports[$alias]=$alias;
        return $alias;
    }

    $className=(string)substr($alias,$pos+1);
    $isClass=$className!=='*';
```

```
    // 如果类已加载，直接返回
    if($isClass && (class_exists($className,false) || interface_exists($className,false)))
        return self::$_imports[$alias]=$className;

    if(($path=self::getPathOfAlias($alias))!==false)
    {
        if($isClass)
        {
            if($forceInclude)
            {
                if(is_file($path.'.php'))
                    require($path.'.php');
                else
                    throw new CException(Yii::t('yii','Alias "{alias}" is invalid. Make sure it points to an existing PHP file and the file is readable.',array('{alias}'=>$alias)));
                self::$_imports[$alias]=$className;
            }
            else
                self::$classMap[$className]=$path.'.php';
            return $className;
        }
        else    // a directory
        {
            if(self::$_includePaths===null)
            {
                 // 去掉当前目录
                self::$_includePaths=array_unique(explode(PATH_SEPARATOR,get_include_path()));
                if(($pos=array_search('.',self::$_includePaths,true))!==false)
                    unset(self::$_includePaths[$pos]);
            }

            // 导入多个目录时，后导入的目录会添加在路径前面
            array_unshift(self::$_includePaths,$path);

            // 当设置 include_path 失败时置$enableIncludePath=false，这种情况下 autoload 时直接遍历 $_includePaths 查找文件
            if(self::$enableIncludePath&&
set_include_path('.'.PATH_SEPARATOR.implode(PATH_SEPARATOR,self::$_includePaths))===false)
                self::$enableIncludePath=false;

            return self::$_imports[$alias]=$path;
        }
```

```
    }
    else
        throw new CException(Yii::t('yii','Alias "{alias}" is invalid. Make sure it points to an existing directory or file.', array('{alias}'=>$alias)));
}
```

onBeginRequest 配置项

onBeginRequest 配置项注册一个 BeginRequest 事件，参考前面的 CComponent::__set 方法，事件通过 CList::add 方法被添加到事件链表的尾部；可注册多个相同的事件，事件回调函数按注册顺序被调用执行。

CList.php:

```
public function CList::add($item)
{
    $this->insertAt($this->_c,$item);
    return $this->_c-1;
}

public function insertAt($index,$item)
{
    if(!$this->_r)
    {
        if($index===$this->_c)
            $this->_d[$this->_c++]=$item; //  在尾部插入 item
        elseif($index>=0 && $index<$this->_c)
        {    //  在数组中间插入 item，例如：$ar=[1,3,5], array_splice($ar,1,0,array(2))，结果:$ar=[1,2,3,5]
            array_splice($this->_d,$index,0,array($item));
            $this->_c++;
        }
        else
            throw new CException(Yii::t('yii','List index "{index}" is out of bound.', array('{index}'=>$index)));
    }
    else
        throw new CException(Yii::t('yii','The list is read only.'));
}
```

components 配置项

components 配置项注册应用程序组件，在创建 CWebApplication 对象时已注册了核心组件，其中包括 urlManager 组件，此配置项会调用 CModule::setCompnents 方法添加组件:

CModule.php:

```
public function setComponents($components,$merge=true)
{
    foreach($components as $id=>$component)
        $this->setComponent($id,$component,$merge);
}

public function setComponent($id,$component,$merge=true)
{
     // 注销组件
    if($component===null)
    {   // _components 属性中存放已加载的组件
        unset($this->_components[$id]);
        return;
    }
    /**
     * CApplication 构造方法在调用 registerCoreComponents 方法注册核心组件时已注册 urlManager 组件
     * 假设：$components=array(
     *          ...,
     *          'urlManager'=>array(
     *                  'class'=>'CUrlManager',
     *          ),
     *          ...
     * );
     * 当调用 setComponent 方法时，参数 id='urlManager'，参数$component=['class' => 'CUrlManager'],
     * 语句 <code>$component instanceof IApplicationComponent</code>返回 false，下面的 elseif 代码段永远跳过
     */
    elseif($component instanceof IApplicationComponent)
    {
        $this->_components[$id]=$component;

        if(!$component->getIsInitialized())
            $component->init();

        return;
    } // 覆盖组件
    elseif(isset($this->_components[$id]))
    {
        if(isset($component['class']) && get_class($this->_components[$id])!==$component['class'])
        {
            unset($this->_components[$id]);
            $this->_componentConfig[$id]=$component; //we should ignore merge here
            return;
```

```
        }

        foreach($component as $key=>$value)
        {
            if($key!=='class')
                $this->_components[$id]->$key=$value;
        }
    }
    // 替换组件属性
    elseif(isset($this->_componentConfig[$id]['class'],$component['class'])
        && $this->_componentConfig[$id]['class']!==$component['class'])
    {
        $this->_componentConfig[$id]=$component; //we should ignore merge here
        return;
    }
    /** 合并组件属性
     *   $ar1 = ['class'=>'CUrlManager'];
     *   $ar2 = ['showScriptName'=>false,'urlFormat'=>'path','rules'=>[]];
     *
     * CMap::mergeArray 结果：
     * ['class'=>'CUrlManager','showScriptName'=>false,'urlFormat'=>'path','rules'=>[]]
     */
    if(isset($this->_componentConfig[$id]) && $merge)
     $this->_componentConfig[$id]=CMap::mergeArray($this->_componentConfig[$id],$component);
    else // 添加组件
        $this->_componentConfig[$id]=$component;
}
```

params 配置项

params 配置项用于定义 Application 级别的全局自定义参数，自定义的参数是大小写敏感的，参数通过 CModule:: setParams 方法被添加到 map 中，可通过 Yii::app()->params['paramName']访问。

3.3 加载 request 组件

在 CWebApplication::init 方法中加载 request 组件，这里提前加载的原因是响应接下来的 BeginRequest 事件。组件加载分为两步，第一步是调用 YiiBase::createComponent 方法创建组件对象：

YiiBase::createComponent:

```
public static function createComponent($config)
```

```
{
    $args = func_get_args();
    if(is_string($config))
    {
         // 保存类名，并将类属性赋空
        $type=$config;
        $config=array();
    }
    elseif(isset($config['class']))
    {
         // class 中存有类名
        $type=$config['class'];
        unset($config['class']);
    }
    else
        throw new CException(Yii::t('yii','Object configuration must be an array containing a "class" element.'));

    // 如果类未找到，就 include 类
    if(!class_exists($type,false))
        $type=Yii::import($type,true);

    // 形参个数大于 1，将参数传入给构造函数
    if(($n=func_num_args())>1)
    {
        if($n===2)
            $object=new $type($args[1]);
        elseif($n===3)
            $object=new $type($args[1],$args[2]);
        elseif($n===4)
            $object=new $type($args[1],$args[2],$args[3]);
        else
        {
            // 通过反射来创建类对象
            unset($args[0]);
            $class=new ReflectionClass($type);
            // Note: ReflectionClass::newInstanceArgs() is available for PHP 5.1.3+
            // $object=$class->newInstanceArgs($args);
            $object=call_user_func_array(array($class,'newInstance'),$args);
        }
    }
    else // 创建类对象
        $object=new $type;
```

```
    // 类属性赋值
    foreach($config as $key=>$value)
        $object->$key=$value;

    return $object;
}
```

第二步调用组件类 CHttpRequest::init 方法：

CHttpRequest.php:

```
public function init()
{
    // CApplicationComponent::init
    parent::init();
    $this->normalizeRequest();
}

protected function normalizeRequest()
{
    // normalize request
    if(version_compare(PHP_VERSION,'7.4.0','<'))
    {
        if(function_exists('get_magic_quotes_gpc') && get_magic_quotes_gpc())
        {
             // 调用 stripslashes 函数去掉$_GET、$_POST、$_REQUEST、$_COOKIE 数组 kv 值中
                  // 单引号（'）、双引号（"）、反斜线（\）与 NUL（NULL）字符前的反斜线
            if(isset($_GET))
                $_GET=$this->stripSlashes($_GET);
            if(isset($_POST))
                $_POST=$this->stripSlashes($_POST);
            if(isset($_REQUEST))
                $_REQUEST=$this->stripSlashes($_REQUEST);
            if(isset($_COOKIE))
                $_COOKIE=$this->stripSlashes($_COOKIE);
        }
    }

    // 开启 csrf 验证后，注册 BeginRequest 事件回调函数
    if($this->enableCsrfValidation)
        Yii::app()->attachEventHandler('onBeginRequest',array($this,'validateCsrfToken'));
}
```

4 运行应用实例

创建应用实例后调用 CApplication::run 方法运行实例。

CApplication::run:

```
public function run()
{
    // 处理 BeginRequest 事件
    if($this->hasEventHandler('onBeginRequest'))
        $this->onBeginRequest(new CEvent($this));
    // 注册 php 中止时执行的函数，在脚本执行完或调用 die、exit 后执行
    // 可多次调用 register_shutdown_function()，被注册的函数按照注册顺序依次执行
    register_shutdown_function(array($this,'end'),0,false);
    // 处理请求
    $this->processRequest();
    // 处理 EndRequest 事件
    if($this->hasEventHandler('onEndRequest'))
        $this->onEndRequest(new CEvent($this));
}
```

4.1 处理 onBeginRequest 事件

处理 onBeginRequest 事件的代码：

```
if($this->hasEventHandler('onBeginRequest'))
    $this->onBeginRequest(new CEvent($this));
```

先判断是否有注册事件句柄，如果有调用 CComponent::raiseEvent 方法按顺序执行回调函数。3.3 节在加载 request 组件时注册了 csrf 验证的回调函数，如下：

CHttpRequest::validateCsrfToken：

```
public function validateCsrfToken($event)
{
    if ($this->getIsPostRequest() ||
        $this->getIsPutRequest() ||
        $this->getIsPatchRequest() ||
        $this->getIsDeleteRequest())
    {
        $cookies=$this->getCookies();

        // 获取请求类型，默认为 GET 请求
```

```
        $method=$this->getRequestType();
        switch($method)
        {
            case 'POST':
                 // 从 post 表单中获取 token
                $maskedUserToken=$this->getPost($this->csrfTokenName);
            break;
            case 'PUT':
                $maskedUserToken=$this->getPut($this->csrfTokenName);
            break;
            case 'PATCH':
                $maskedUserToken=$this->getPatch($this->csrfTokenName);
            break;
            case 'DELETE':
                $maskedUserToken=$this->getDelete($this->csrfTokenName);
        }

        if (!empty($maskedUserToken) && $cookies->contains($this->csrfTokenName))
        {
            // 加载 securityManager 组件
            $securityManager=Yii::app()->getSecurityManager();
            // 获取 cookie 中的 token
            $maskedCookieToken=$cookies->itemAt($this->csrfTokenName)->value;
            // 解码 cookie 中的 token
            $cookieToken=$securityManager->unmaskToken($maskedCookieToken);
            // 解码表单中的 token
            $userToken=$securityManager->unmaskToken($maskedUserToken);
            // 比较两个 token 值是否相等
            $valid=$cookieToken===$userToken;
        }
        else // 提交的表单中 token 为空或者 cookie 中不存在 token，验证失败
            $valid = false;
        if (!$valid)
            throw new CHttpException(400,Yii::t('yii','The CSRF token could not be verified.'));
    }
}
```

CSRF 简称跨站请求伪造攻击，即携带用户的浏览器 cooke 向一个受信任的网站发起攻击者指定的请求，Yii 默认实现的防范 CSRF 攻击的原理是：在 cookie 中存留一个 token，在表单中嵌入一个隐藏项，储存跟 cookie 中相同的 token，请求提交表单时发送到服务端验证，比较 cookie 中的 token 是否跟表单提交的 token 一致，如果不一致就判定是受 csrf 攻击。

4.2 处理请求

CWebApplication::processRequest:

```
public function processRequest()
{
    // 用于服务维护模式，所有请求都交由 catchAllRequest 设置的路由接管
    if(is_array($this->catchAllRequest) && isset($this->catchAllRequest[0]))
    {
        $route=$this->catchAllRequest[0];
        /**
         * catchAllRequest 配置项格式：'catchAllRequest' => ['site/defend', 'arg1', 'arg2',...]
         * arg1、arg2 参数是可选的，下列 foearch 将参数设置为请求参数
         */
        foreach(array_splice($this->catchAllRequest,1) as $name=>$value)
            $_GET[$name]=$value;
    }
    else
        // 加载 UrlManager 组件，并获取 Request 组件，然后调用 parseUrl 方法解析 url
        $route=$this->getUrlManager()->parseUrl($this->getRequest());
    $this->runController($route);
}
```

4.2.1 加载 UrlManager 组件

加载组件首先创建组件对象再调用 init 方法

CUrlManager.php:

```
public function init()
{
    // 调用 CApplicationComponent::init 附加行为
    parent::init();
    // 处理 url 规则
    $this->processRules();
}

protected function processRules()
{
    /**
     * url 有 GET 和 PATH 两种风格：
         * GET 风格：/index.php?r=post/read&id=100
         * PATH 风格：/index.php/post/read/id/100
     */
```

```
    if(empty($this->rules) || $this->getUrlFormat()===self::GET_FORMAT)
        return;
     // 检查是否配置有 cache 组件
    if($this->cacheID!==false && ($cache=Yii::app()->getComponent($this->cacheID))!==null)
    {
        // 从缓存中取出 rules 并检查 rules 是否有变动
        $hash=md5(serialize($this->rules));
        if(($data=$cache->get(self::CACHE_KEY))!==false && isset($data[1]) && $data[1]===$hash)
        {
            $this->_rules=$data[0];
            return;
        }
    }
    // 循环处理每条 rule，保存每条 rule 的 CUrlRule 类对象
    foreach($this->rules as $pattern=>$route)
        $this->_rules[]=$this->createUrlRule($route,$pattern);
    if(isset($cache))
        $cache->set(self::CACHE_KEY,array($this->_rules,$hash)); // 缓存到 cache 中
}

protected function createUrlRule($route,$pattern)
{
    // 自定义的 url 规则，这里直接返回，在 parseUrl 方法中根据$route 创建组件
    if(is_array($route) && isset($route['class']))
        return $route;
    else
    {
         // 加载 CUrlRule 类
        $urlRuleClass=Yii::import($this->urlRuleClass,true);
        return new $urlRuleClass($route,$pattern); //创建类对象，并调用构造函数
    }
}
```

CUrlRule::__construct 方法将 pattern、route 解析成正则表达式并分别保存到 pattern 和 routePattern 属性中。

```
public function CUrlRule::__construct($route,$pattern)
{
    if(is_array($route))
    {
         /**
          * 数组格式，如：
          * [
```

```
            *       'post/index', //  必须是第一个数字索引元素
            *       'pattern' => 'posts',
            *       'urlSuffix' => '.json',
            * ]
            * 或
            * 'posts' => [
            *        'post/index',
            *        'urlSuffix' => '.json',
            * ]
            *
            * urlSuffix：开启 url 美化，可在 uri 后添加后缀，开启后以下 uri 相等：/posts, /posts.json
            * caseSensitive: 设置 pattern 匹配时是否大小写敏感
            * defaultParams：默认的 GET 请求参数，如：defaultParams=['name1'=>'value1']，可通过$_GET['name1']获取参数
            * matchValue: todo
            * verb: 设置此条 rule 仅用于匹配指定的 HTTP Method
            * parsingOnly: 此条 rule 是否仅用于请求解析
            */
        foreach(array('urlSuffix', 'caseSensitive', 'defaultParams', 'matchValue', 'verb', 'parsingOnly') as $name)
        {
            if(isset($route[$name]))
                $this->$name=$route[$name];
        }
        if(isset($route['pattern']))
            $pattern=$route['pattern'];
        //  第一个数字索引元素才是真正的 route
        $route=$route[0];
    }
    $this->route=trim($route,'/');

    //  转换成正则表达式格式
    $tr2['/']=$tr['/']='\\/';

    // pattern=>route: '<controller:\w+>/<action:\w+>'=>'<controller>/<action>'
    if(strpos($route,'<')!==false && preg_match_all('/<(\w+)>/',$route,$matches2))
    {
        foreach($matches2[1] as $name)
            $this->references[$name]="<$name>"; // references 中存放 route 引用 pattern 中的参数名称
    }

    // pattern 是否已 http 或 https 开头，$pattern 中可以包含"http://"？
    $this->hasHostInfo=!strncasecmp($pattern,'http://',7) || !strncasecmp($pattern,'https://',8);
```

```
// 是否设置了 http method，并且将 “，"或空白符分隔的字符串转换成数组
if($this->verb!==null)
    $this->verb=preg_split('/[\s,]+/',strtoupper($this->verb),-1,PREG_SPLIT_NO_EMPTY);

/**
 * pattern 格式如下：
 * 1：<controller:\w+><action:\w+>
 * 2：<controller\w+><action:\w+>
 * 3：<controller><action:\w+>
*/
if(preg_match_all('/<(\w+):?(.*?)?>/',$pattern,$matches))
{
    $tokens=array_combine($matches[1],$matches[2]);
    foreach($tokens as $name=>$value)
    {
        if($value==='')
            $value='[^\/]+'; // 默认包含除/外的任意字符
                // 命名一个名字为$name 的组，匹配规则满足$value
        $tr["<$name>"]="(?P<$name>$value)";
        if(isset($this->references[$name]))
            $tr2["<$name>"]=$tr["<$name>"];
        else // route 中没有引用 pattern 中的参数名，此参数将做为请求参数
            $this->params[$name]=$value;
    }
}
$p=rtrim($pattern,'*');
$this->append=$p!==$pattern;
$p=trim($p,'/');
/**
 * 替换前：<controller\w+><action:\w+>
 * 替换后：<controller><action>
 */
$this->template=preg_replace('/<(\w+):?.*?>/','<$1>',$p);
$p=$this->template;
// parsingOnly=false 此条 rule 可用于请求解析和 createUrl，=true 仅用于请求解析
if(!$this->parsingOnly)
    // 匹配非<>中的子串，并进行转义
    $p=preg_replace_callback('/(?<=^|>)[^<]+(?=<|$)/',array($this,'escapeRegexpSpecialChars'),$p);
$this->pattern='/^'.strtr($p,$tr).'\/';
if($this->append)
    $this->pattern.='/u';
else
    $this->pattern.='$/u';
```

```
    if($this->references!==array())
        $this->routePattern='/^'.strtr($this->route,$tr2).'$/u';

    if(YII_DEBUG && @preg_match($this->pattern,'test')===false)
        throw new CException(Yii::t('yii','The URL pattern "{pattern}" for route "{route}" is not a valid regular expression.',
            array('{route}'=>$route,'{pattern}'=>$pattern)));
}
```

4.2.2 解析请求

调用 CUrlManager::parseUrl()方法解析请求，GET 风格的 url 直接返回 routeVar 配置项的值(默认为 r)，

Path 风格会匹配 url 规则，未匹配到规则就返回 pathinfo。

CUrlManager::parseUrl:

```
public function parseUrl($request)
{
     // 判断是否为 path 风格
    if($this->getUrlFormat()===self::PATH_FORMAT)
    {
        /**
         * 例如，nginx 配置如下：
         * root /home/www/testyii/webroot
         * fastcgi_param  SCRIPT_FILENAME  /home/www/testyii/webroot/index.php
         * fastcgi_index  index.php
         *
         * url=http://www.example.com:80/site/index?a=1&b=2
         */
        // $rawPathInfo=/site/index
        $rawPathInfo=$request->getPathInfo();
        // urlSuffix 配置项设置后，开启美化 url，uri 可加后缀。removeUrlSuffix 方法去掉后缀
        // 例如：$rawPathInfo=/site/index.json, $pathInfo=/site/index
        $pathInfo=$this->removeUrlSuffix($rawPathInfo,$this->urlSuffix);
        foreach($this->_rules as $i=>$rule)
        {
            // CUrlManager::init()方法中只处理了非自定义的规则，这里先创建自定义规则类对象
            if(is_array($rule))
                $this->_rules[$i]=$rule=Yii::createComponent($rule);
            // 根据 rule 解析 url
            if(($r=$rule->parseUrl($this,$request,$pathInfo,$rawPathInfo))!==false)
                return isset($_GET[$this->routeVar]) ? $_GET[$this->routeVar] : $r;
        }
```

```
        // 请求未匹配到规则，且设置了 url 精确匹配抛异常
        if($this->useStrictParsing)
            throw new CHttpException(404,Yii::t('yii','Unable to resolve the request "{route}".',
                array('{route}'=>$pathInfo)));
        else
            return $pathInfo;
    }// GET 风格返回指定的请求参数值
    elseif(isset($_GET[$this->routeVar]))
        return $_GET[$this->routeVar];
    elseif(isset($_POST[$this->routeVar]))
        return $_POST[$this->routeVar];
    else
        return '';
}
```

CUrlRule::parseUrl:

```
public function parseUrl($manager,$request,$pathInfo,$rawPathInfo)
{
     // 根据 verb 配置项，判断当前请求 Method 是否匹配
    if($this->verb!==null && !in_array($request->getRequestType(), $this->verb, true))
        return false;

    // 根据 caseSensitive 配置项检查匹配时是否大小写敏感，正则表达式修饰 i 表示不区分大小写
    if($manager->caseSensitive && $this->caseSensitive===null || $this->caseSensitive)
        $case='';
    else
        $case='i';

    // 根据 urllSuffix 配置项，移除 uri 后缀
    if($this->urlSuffix!==null)
        $pathInfo=$manager->removeUrlSuffix($rawPathInfo,$this->urlSuffix);

    // URL suffix required, but not found in the requested URL
    if($manager->useStrictParsing && $pathInfo===$rawPathInfo)
    {
        // uri 精确匹配，当设置了 urlSuffix，但请求 uri 中不包含 Suffix 时返回 false
        $urlSuffix=$this->urlSuffix===null ? $manager->urlSuffix : $this->urlSuffix;
        if($urlSuffix!='' && $urlSuffix!=='/')
            return false;
    }

    if($this->hasHostInfo)
```

```
        $pathInfo=strtolower($request->getHostInfo()).rtrim('/'.$pathInfo,'/');

    $pathInfo.='/';

    // $this->pattern 中存储有 pattern 的正则表达式,
    if(preg_match($this->pattern.$case,$pathInfo,$matches))
    {
          // 填充默认的请求参数
        foreach($this->defaultParams as $name=>$value)
        {
            if(!isset($_GET[$name]))
                $_REQUEST[$name]=$_GET[$name]=$value;
        }
        $tr=array();
        foreach($matches as $key=>$value)
        {
            if(isset($this->references[$key]))
                  // references 中存放 route 引用 pattern 中的参数名称
                $tr[$this->references[$key]]=$value;
            elseif(isset($this->params[$key]))
                  // 将 route 没有引用的参数做为请求参数
                $_REQUEST[$key]=$_GET[$key]=$value;
        }
        if($pathInfo!==$matches[0]) // there're additional GET params
            $manager->parsePathInfo(ltrim(substr($pathInfo,strlen($matches[0])),'/'));
        if($this->routePattern!==null)
             // 用值替换 route 中的参数
            return strtr($this->route,$tr);
        else
            return $this->route;
    }
    else
        return false;
}
```

4.2.3 创建并运行控制器

首先根据 route 调用 CWebApplication::createController 方法按下列顺序生成控制器对象:

1、在 controllerMap 中查找控制器

2、判断 route 是否包含模块 id，如果包含，先加载模块再创建控制器

3、如果 route 等于空或'/'，就使用默认的控制器。

4、在 controllerPath 目录下查找控制器类文件并加载

5、根据控制器类名生成类对象

查找并创建控制器对象后，并调用 init()方法，然后调用 CController::run()方法运行 action。

CwebApplication.php:

```
public function runController($route)
{
     // 创建控制器类对象，以及解析 actionID
    if(($ca=$this->createController($route))!==null)
    {
        list($controller,$actionID)=$ca;
        $oldController=$this->_controller;
        $this->_controller=$controller;
        // 运行 action 前调用 controller 的 init 方法
        $controller->init();
        // 运行 action
        $controller->run($actionID);
        $this->_controller=$oldController;
    }
    else
        throw new CHttpException(404,Yii::t('yii','Unable to resolve the request "{route}".',
            array('{route}'=>$route===''?$this->defaultController:$route)));
}

public function createController($route,$owner=null)
{
    if($owner===null)
        $owner=$this;
    // $route=''或$route='/'或$route=数组
    if((array)$route===$route || ($route=trim($route,'/'))==='')
        $route=$owner->defaultController;
    $caseSensitive=$this->getUrlManager()->caseSensitive;

    $route.='/';
    while(($pos=strpos($route,'/'))!==false)
    {
         // 第一个反斜线前面的子串
        $id=substr($route,0,$pos);
        if(!preg_match('/^\w+$/',$id))
            return null;
        if(!$caseSensitive)
```

```
        $id=strtolower($id);
    // 第一个反斜线后面的子串
    $route=(string)substr($route,$pos+1);
    if(!isset($basePath))  // first segment
    {
         // 控制器 id 到控制器类名的映射，可通过 controllerMap 配置项赋值
        if(isset($owner->controllerMap[$id]))
        {
             // 返回控制器类对象以及 actionId
            return array(
                Yii::createComponent($owner->controllerMap[$id],$id,$owner===$this?null:$owner),
                $this->parseActionParams($route), // 解析 actionId
            );
        }

        // 第一个反斜线前面的子串是模块 id
        if(($module=$owner->getModule($id))!==null)
            return $this->createController($route,$module);

        // 获取控制器类存放的路径
        $basePath=$owner->getControllerPath();
        $controllerID='';
    }
    else
        $controllerID.='/';
    // 控制器类名，首字母大写，其它字母小写
    $className=ucfirst($id).'Controller';
    $classFile=$basePath.DIRECTORY_SEPARATOR.$className.'.php';

    if($owner->controllerNamespace!==null)
        $className=$owner->controllerNamespace.'\\'.str_replace('/','\\',$controllerID).$className;

    if(is_file($classFile))
    {
         // 类名是否存在
        if(!class_exists($className,false))
            require($classFile);
        // 类名存在，并且继承了 CController
        if(class_exists($className,false) && is_subclass_of($className,'CController'))
        {
             // 将 id 首字母转换成小写
            $id[0]=strtolower($id[0]);
            // 创建控制器类对象，以及解析 actionId
```

```
                return array(
                    new $className($controllerID.$id,$owner===$this?null:$owner),
                    $this->parseActionParams($route),
                );
            }
            return null;
        }
        $controllerID.=$id;
        $basePath.=DIRECTORY_SEPARATOR.$id;
    }
}
```

有两种方法定义一个 action，一是在控制类中定义一个以 action 单词为前缀的方法(简称内联 action)，如：

```
class PostController extends Controller
{
        public function beforeAction($action) {
                // more
                return true; // 返回 true 才会执行后续的 action，返回 false action 不被执行
        }
}
```

另一种是定义一个 action 类，如下定义一个新的 action 类方法(简称外部 action)：

```
class UpdateAction extends CAction
{
    public function run()
    {
        // place the action logic here
        return true; // 返回 true 才会执行后续的 action，返回 false action 不被执行
    }
}
```

为让控制器注意到这个动作，需按如下方式覆盖控制器类的 actions()：

```
class PostController extends CController
{
    public function actions()
    {
        return array(
            'edit'=>'application.controllers.post.UpdateAction',
        );
    }
}
```

过滤器可以定义为控制器类的方法，方法名以 filter 开头，如定义一个名为 accessControl 的过滤器：

```
public function filterAccessControl($filterChain)
{
    // 调用 $filterChain->run() 运行后续过滤器与 action 的执行。
}
```

也可以定义为一个 CFilter 或其子类的实例，如下定义了一个过滤器类：

```
class PerformanceFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // 动作被执行之前应用的逻辑
        return true; // 如果动作不应被执行，此处返回 false
    }

    protected function postFilter($filterChain)
    {
        // 动作执行之后应用的逻辑
    }
}
```

要对动作应用过滤器，我们需要覆盖 CController::filters() 方法。此方法应返回一个过滤器配置数组。例如：

```
class PostController extends CController
{
    ......
    public function filters()
    {
        return array(
            'postOnly + edit, create', // +号表示仅应用于 edit 和 create 动作
            array(
                // -号表示应用于除 edit 和 create 之外的其它任何动作
                'application.filters.PerformanceFilter - edit, create',
                'unit'=>'second',
            ),
        );
    }
}
```

调用 CController::run 方法运行 action，运行 action 前会先执行控制器类的 beforeAction 方法，运行 action 之后会执行控制器类的 afterAction 方法。

CController.php:

```
public function run($actionID)
{
    // 首先判断是否是内联 action，如果是创建一个 CInlineAction 对象，否则创建外部 action 类对象
    if(($action=$this->createAction($actionID))!==null)
    {
```

```
        if(($parent=$this->getModule())===null)
            $parent=Yii::app();
        // 调用模块的 beforeControllerAction 方法，如：CWebApplication::beforeControllerAction
        if($parent->beforeControllerAction($this,$action))
        {
             // 运行 action,在运行 action 前会处理控制器的过滤器
            $this->runActionWithFilters($action,$this->filters());
             // 调用模块的 afterControllerAction 方法，CWebApplication::afterControllerAction
            $parent->afterControllerAction($this,$action);
        }
    }
    else
        $this->missingAction($actionID);
}

public function runActionWithFilters($action,$filters)
{
    if(empty($filters))
        $this->runAction($action); // 控制器的过滤器为空直接运行 action
    else
    {
        $priorAction=$this->_action;
        $this->_action=$action;
        // 创建过滤器链，运行 action 前先执行控制器的过滤器
        CFilterChain::create($this,$action,$filters)->run();
        $this->_action=$priorAction;
    }
}

public function runAction($action)
{
    $priorAction=$this->_action;
    $this->_action=$action;
    // 调用控制器的 beforeAction 方法
    if($this->beforeAction($action))
    {
         // 运行 action
        if($action->runWithParams($this->getActionParams())===false)
            $this->invalidActionParams($action);
        else // 调用控制器的 afterAction 方法
            $this->afterAction($action);
    }
    $this->_action=$priorAction;
```

```
}
```

CFilterChain.php:

```
public static function create($controller,$action,$filters)
{
    $chain=new CFilterChain($controller,$action);

    $actionID=$action->getId();
    foreach($filters as $filter)
    {
         // 字符串格式
        if(is_string($filter))    // filterName [+|- action1 action2]
        {
            if(($pos=strpos($filter,'+'))!==false || ($pos=strpos($filter,'-'))!==false)
            {
                 // 查找 action 名
                $matched=preg_match("/\b{$actionID}\b/i",substr($filter,$pos+1))>0;
                if(($filter[$pos]==='+')===$matched) // + 表示过滤器能应用于 action
                    $filter=CInlineFilter::create($controller,trim(substr($filter,0,$pos)));
            }
            else // 不包含[+|-]，表示过滤器应用于所有 action
                $filter=CInlineFilter::create($controller,$filter);
        } // 数组格式
        elseif(is_array($filter))    // array('path.to.class [+|- action1, action2]','param1'=>'value1',...)
        {
            if(!isset($filter[0]))
                throw new CException(Yii::t('yii','The first element in a filter configuration must be the filter class.'));
            $filterClass=$filter[0]; // 第 0 个元素是类名
            unset($filter[0]);
            if(($pos=strpos($filterClass,'+'))!==false || ($pos=strpos($filterClass,'-'))!==false)
            {
                $matched=preg_match("/\b{$actionID}\b/i",substr($filterClass,$pos+1))>0;
                if(($filterClass[$pos]==='+')===$matched)
                    $filterClass=trim(substr($filterClass,0,$pos));
                else
                    continue;
            }
            $filter['class']=$filterClass;
            $filter=Yii::createComponent($filter);
        }

        if(is_object($filter))
```

```
        {
            $filter->init();
            $chain->add($filter); //  将过滤器按顺序添加到链表中
        }
    }
    return $chain;
}
public function run()
{
    if($this->offsetExists($this->filterIndex))
    {
        $filter=$this->itemAt($this->filterIndex++);
        Yii::trace('Runningfilter'.($filter instanceof CInlineFilter ? get_class($this->controller).
'.filter'.$filter->name.'()':get_class($filter).'.filter()'),'system.web.filters.CFilterChain');
       // CInlineFilter 类对象会直接调用控制器类的过滤器方法，CFilter 或直子类对象在 action 运行前后分
别执行 preFilter、postFilter
        $filter->filter($this);
    }
    else //  被执行控制器的过滤器，再执行 beforeAction，最后执行 action
        $this->controller->runAction($this->action);
}
```

4.3 处理 onEndRequest 事件

可在请求结束前注册 endRequest 事件处理请求的收尾工作，比如 log 组件在 endRequest 事件中将内存中的日志写到文件中。

5 小结

注意事项：

(1)、urlManager 组件的 url 规则太多时，最好将规则解析后通过 cache 组件存储到缓存(如:apc、yac、redis 等)或文件中，避免每次处理请求时都去解析 url 规则。

如下配置会将规则存储到文件中：

```
'components' => [
      'cache' => [
            'class' => 'CFileCache',
      ],
```

],

(2)、Yii::app()->end()和 exit 都可以显示终止当前请求，这两者存在差别。

Yii::app()->end()源代码如下：

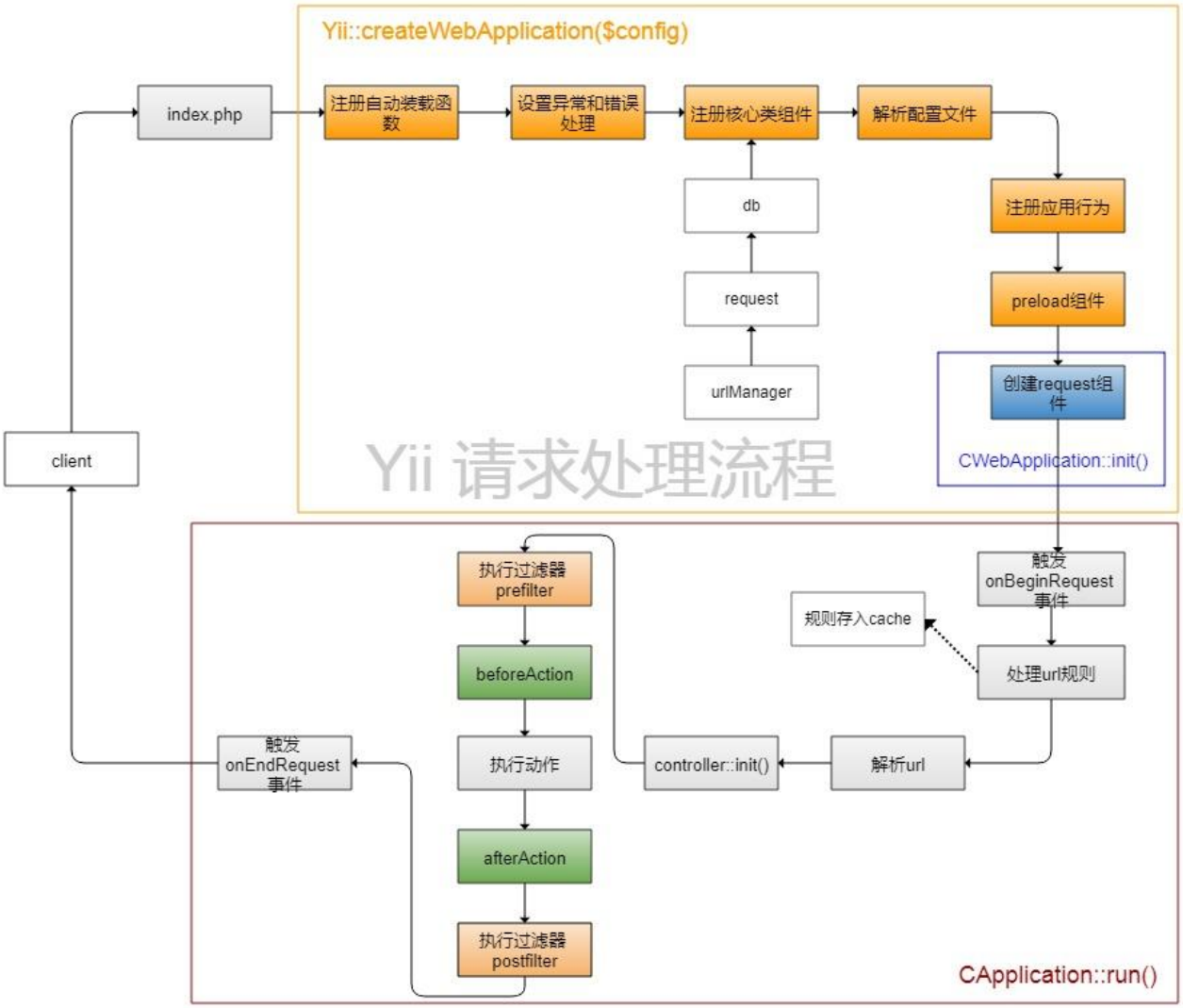
```
public function end($status=0,$exit=true)
{
    if($this->hasEventHandler('onEndRequest'))
        $this->onEndRequest(new CEvent($this));
    if($exit)
        exit($status);
}
```

在请求处理流程的 CApplication::run 方法中执行 register_shutdown_function(array($this,'end'),0,false)语句也注册了终止执行函数 end。

Yii::app()->end()方法在调用 exit 函数前会触发 onEndRequest 事件，而 exit 函数也会执行 CApplication::end()，也就是说 Yii::app()->end()触发了两次 onEndRequest 事件，而 exit 函数只触发一次。

(3)、相比 include、require，Yii::import 导入类更加高效，导入的类并不会真正被包含，直到第一次被引用。

请求处理流程图：


Yii::createWebApplication($config)
index.php
注册自动装载函数
设置异常和错误处理
注册核心类组件
解析配置文件
db
request
urlManager
注册应用行为
preload组件
创建request组件
CWebApplication::init()
Yii 请求处理流程
client
执行过滤器prefilter
beforeAction
执行动作
afterAction
执行过滤器postfilter
触发onEndRequest事件
controller::init()
解析url
处理url规则
规则存入cache
触发onBeginRequest事件
CApplication::run()

第二章 组件、事件与行为

1 组件

在项目的入口脚本 index.php 中 Yii::createWebApplication($config)语句创建的 CWebApplication 就是一个组件，组件是 Yii 的基石，Yii 应用是建立在组件之上的，自定义组件需继承 CComponent 或其子类，CComponent 是所有组件的基类，它定义了使用属性和事件的方式。

组件的属性可通过以下两种方法定义：

```
class MyComponent extends CComponent
{
    // 属性 prop1 定义为类公共成员变量
    public $prop1;
    // 属性 prop2 通过定义的 getter 和 setter 方法来赋值或获取值
    private $_prop2;

    public function __construct() {
    }

    // yii 框架在加载组件时，自动调用 init 方法
    public function init()
    {
    }

    public function setProp2($v)
    {
        $this->_prop2 = $v;
    }

    public function getProp2()
    {
        return $this->_prop2;
    }
}
```

组件通过 components 配置项注册，如：

```
'components'=> [
    'MyComponentId' => [
        'class' => 'application.components.MyComponent',
        'enable' => true,  // 开启或禁用组件
```

```
            'prop1' => 'type',
            'prop2' => false,
            ],
],
```

经过 components 配置项配置的组件，可通过如下语句访问组件：

```
$v = Yii::app()->MyComponentId->prop1;
```

以上语句访问 MyComponentId 组件的 prop1 属性，这条语句的原理是 Yii::app()获取应用程序对象，即 CWebApplication 类对象，然后访问 CWebApplication 类对象的 MyComponentId 属性，当没有此属性时调用__get 魔术方法，__get 方法调用 CModule::getComponent 方法获取组件类对象(第一次访问组件时，创建组件对象并调用组件的 init 方法)，再通过组件对象访问组件属性。CModule::__get 源代码如下：

```
public function __get($name)
{
    // 先判断是否有组件，然后再创建或获取组件类对象
    if($this->hasComponent($name))
        return $this->getComponent($name);
    else
        // 调用 CComponents::__get
        return parent::__get($name);
}
```

组件注册仅将组件缓存到内存中，第一次引用时才会加载组件类创建组件对象，并调用组件的 init 方法，如：Yii::app()->MyComponentId->prop2 = true 语句会调用 CModule::getComponent 方法创建组件：

CModule::getComponent:

```
public function getComponent($id,$createIfNull=true)
{
    // 组件已创建，直接返回
    if(isset($this->_components[$id]))
        return $this->_components[$id];
    elseif(isset($this->_componentConfig[$id]) && $createIfNull)
    {
        // setComponent 方法将组件缓存到_componentConfig 属性中
        $config=$this->_componentConfig[$id];
        // 可通过 enalbed 属性开启或禁用属性
        if(!isset($config['enabled']) || $config['enabled'])
        {
            Yii::trace("Loading \"$id\" application component",'system.CModule');
```

```
                unset($config['enabled']);
                // 创建组件对象
                $component=Yii::createComponent($config);
                // 调用组件的 init 方法
                $component->init();
                // 保留已创建的组件
                return $this->_components[$id]=$component;
            }
        }
}
```

YiiBase::createComponent:

```
public static function createComponent($config)
{
    $args = func_get_args();
    if(is_string($config))
    {
        // 保存类名，并将类属性赋空
        $type=$config;
        $config=array();
    }
    elseif(isset($config['class']))
    {
        // class 中存有类名
        $type=$config['class'];
        unset($config['class']);
    }
    else
        throw new CException(Yii::t('yii','Object configuration must be an array containing a "class" element.'));

    // 如果类未找到，就 include 类
    if(!class_exists($type,false))
        $type=Yii::import($type,true);

    // 形参个数大于 1，将参数传入给构造函数
    if(($n=func_num_args())>1)
    {
        if($n===2)
            $object=new $type($args[1]);
        elseif($n===3)
            $object=new $type($args[1],$args[2]);
        elseif($n===4)
```

```
            $object=new $type($args[1],$args[2],$args[3]);
        else
        {
            // 通过反射来创建类对象
            unset($args[0]);
            $class=new ReflectionClass($type);
            // Note: ReflectionClass::newInstanceArgs() is available for PHP 5.1.3+
            // $object=$class->newInstanceArgs($args);
            $object=call_user_func_array(array($class,'newInstance'),$args);
        }
    }
    else // 创建类对象
        $object=new $type;

    // 类属性赋值，属性直接赋值或调用 setter 方法
    foreach($config as $key=>$value)
        $object->$key=$value;

    return $object;
}
```

2 事件

yii 事件是异步执行的，事件用来解决一对多的依赖关系，当对象的状态发生改变时，所有依赖它的对象都能得到通知。自定义事件需要通过三步来实现：1、定义事件，2、注册事件句柄，3、触发事件。

事件名必须以'on'开头，并且不区分大小写，事件是基于组件的，定义事件的类必须继承 component 或其子类，如下代码定义了 MyComponent 组件的 onMyEvent 事件：

```
class MyComponent extends CComponent
{
    // 定义 onMyEvent 事件
    public function onMyEvent($event)
    {
        $this->raiseEvent('onMyEvent', $event);
    }

    // 事件句柄函数
    public static function myEventCb($event)
    {
```

```
    }
}
```

注册事件句柄，句柄是一个 PHP 回调函数，可使用如下回调函数之一：

- 字符串形式指定的 PHP 全局函数，如 'trim'；
- 对象名和方法名数组形式指定的对象方法，如 [$object, $method]；
- 类名和方法名数组形式指定的静态类方法，如 [$class, $method]；
- 匿名函数，如 function ($event) { ... }；

可以在项目配置文件中注册事件句柄，如：

```
components => [
    'MyComponentId' => [
        'class' => 'application.components.MyComponent',
        'onMyEvent' => ['MyComponent', ' myEventCb'],  // 注册事件句柄
    ],
]
```

也可以在函数中直接注册事件句柄，如：

```
class MyComponent extends CComponent
{
    public function __construct() {
        // 注册事件句柄,
        $this->onMyEvent = [$this, 'myEventCb'];
    }
}
```

事件触发可以通过以下语句来实现:

```
if(Yii::app()->MyComponentId->hasEventHandler('onMyEvent')) {
    Yii::app()->MyComponentId->onMyEvent(new CEvent($this));
}
```

事件的实现原理
采用配置项的方式注册事件句柄，在组件第一次被加载创建组件对象时，会调用如下代码：

```
public static function createComponent($config)
{
    ...
    foreach($config as $key=>$value)
        $object->$key=$value;
    ...
}
```

$object->$key=$value 语句会触发组件基类 CComponet 的__set 魔术方法，__set 方法将事件保存到$_e 属性中，如：

```
public function __set($name,$value)
{
    ...
    elseif(strncasecmp($name,'on',2)===0 && method_exists($this,$name))
    {
        // duplicating getEventHandlers() here for performance
        $name=strtolower($name);
        if(!isset($this->_e[$name]))
            $this->_e[$name]=new CList;
        return $this->_e[$name]->add($value);
    }
    ….
}
```

触发事件时先调用 CComponent::hasEventHandler 判断是否注册了事件句柄

```
public function hasEventHandler($name)
{
    $name=strtolower($name);
    return isset($this->_e[$name]) && $this->_e[$name]->getCount()>0;
}
```

然后调用 CComponent::raiseEvent 方法执行句柄函数，此方法按句柄注册顺序执行。

```
public function raiseEvent($name,$event)
{
    // 事件名不区别大小写
    $name=strtolower($name);
    if(isset($this->_e[$name]))
    {
        // 一个事件支持注册多个句柄，按注册顺序执行事件句柄
        foreach($this->_e[$name] as $handler)
        {
            if(is_string($handler))
                call_user_func($handler,$event);
            elseif(is_callable($handler,true))
            {
                if(is_array($handler))
                {
                    // an array: 0 - object, 1 - method name
                    list($object,$method)=$handler;
                    if(is_string($object)) // static method call
                        call_user_func($handler,$event);
                    elseif(method_exists($object,$method))
```

```
                    $object->$method($event);
                else
                    throw new CException(Yii::t('yii','Event "{class}.{event}" is attached with an invalid handler "{handler}"', array('{class}'=>get_class($this), '{event}'=>$name, '{handler}'=>$handler[1])));
                }
                else // PHP 5.3: anonymous function
                    call_user_func($handler,$event);
            }
            else
                throw new CException(Yii::t('yii','Event "{class}.{event}" is attached with an invalid handler "{handler}"', array('{class}'=>get_class($this), '{event}'=>$name, '{handler}'=>gettype($handler))));
            // stop further handling if param.handled is set true
            // 事件句柄置 handled=true，后面的事件句柄不再执行
            if(($event instanceof CEvent) && $event->handled)
                return;
        }
    }
    elseif(YII_DEBUG && !$this->hasEvent($name))
        throw new CException(Yii::t('yii','Event "{class}.{event}" is not defined.', array('{class}'=>get_class($this), '{event}'=>$name)));
}
```

事件句柄注册后想删除怎么办？CComponent 类提供了 detachEventHandler 方法用于删除已注册的句柄。

CComponent:: detachEventHandler:

```
public function detachEventHandler($name,$handler)
{
    if($this->hasEventHandler($name))
        // 从 list 中获取事件，然后调用 remove 方法移除事件句柄
        return $this->getEventHandlers($name)->remove($handler)!==false;
    else
        return false;
}
```

3 行为

3.1 行为类的使用

行为可以在不改变类的情况下丰富类的功能，php 中不支持多重继承，通过行为可以达到多重继承的效果，行为的定义需要实现 IBehavior 接口，一般情况下我们继承 CBehavior 类就可以了，如下代码实现了一个行

为类：

```
class MyBehavior extends CBehavior
{
    // 行为的属性 1
    public $bprop1 = 1;

    // 行为的属性 2，通过 getter 和 seter 来操作属性 2
    private $_bprop2;

    public function getBprop2()
    {
        return $this->_bprop2;
    }

    public function setBprop2($v)
    {
        $this->_bprop2 = $v;
    }

    // 行为的方法
    public function bfoo()
    {
        return 'bfoo';
    }

    // 覆盖父类的 events 方法，行为类附加到组件后，调用此方法绑定组件事件
    public function events()
    {
        return [
            'my_event' => 'methodName',
        ];
    }

    public function methodName($event)
    {
        $event->owner; // 附加的组件对象
    }
}
```

接下来调用 CComponent::attachBehavior 方法将行为附加到组件上，来扩充组件的功能：

```
class MyComponent extends CComponent
{
```

```
    // 在 MyComponent 的 init 方法中附加行为，也可在其它地方附加
    public function init()
    {
        // 附加行为类
        $this->attachBehavior('myBehavior', new MyBehavior);
        // 获取行为类的属性 1
        $this->bprop1;
        // 设置行为类的属性 2
        $this->bprop2 = 'example';
        // 调用行为类的 foo 方法
        $this->bfoo();
    }
}
```

可见，附加行为后，访问行为类属性或方法，跟访问自身类属性和方法一样。

CApplicationComponent 也是应用程序组件的基类，他继承 CComponent 并实现 IApplicationComponent 接口。

```
abstract class CApplicationComponent extends CComponent implements IApplicationComponent
{
    public $behaviors=array(); // 存放 behaviors 行为配置项值

    private $_initialized=false;

    public function init()
    {
        //  附加行为，行为可通过 behaviors 配置项配置
        $this->attachBehaviors($this->behaviors);
        $this->_initialized=true;
    }

    public function getIsInitialized()
    {
        return $this->_initialized;
    }
}
```

当组件类继承 CApplicationComponent 类后，可通过配置文件附加行为，假设我们的组件类如下：

```
class MyComponent extends CComponent
{
    public function init()
    {
        parent::init(); // 调用父类的 init 方法附加组件
        ...
```

```
    }
}
```

配置文件如下：

```
'components'=>[
    'MyComponentId' => [
        'class' => 'application.components.MyComponent',
        'behaviors' => [
            'myBehaviors' => 'MyBehavior', //  附加行为
        ],
    ],
]
```

行为附加到组件后，可通过行为的 events 方法将行为方法跟组件事件绑定，来实现行为观察或改变组件的执行流程。MyBehavior::events 方法中绑定了组件的 my_event 事件，当 my_event 事件触发后会调用 MyBehavior::methodName 方法。

3.2 行为的实现原理

行为类需要实现 IBehavior 接口，IBehavior 接口申明了 attach 等方法：

```
interface IBehavior
{
    //  附加行为到组件
    public function attach($component);
    //  解除行为的附加
    public function detach($component);
    //  判断行为是否使能
    public function getEnabled();
    //  使能行为
    public function setEnabled($value);
}
```

CBehavior 做为行为类的基类，其实现了 IBehavior 的接口：

```
class CBehavior extends CComponent implements IBehavior
{
    private $_enabled=false; //  行为默认禁止附加
    private $_owner; //  存放行为附加的组件对象

    //  子类可覆盖此方法实现行为方法绑定组件事件
    public function events()
    {
        return array();
```

```
}
// 附加行为,
public function attach($owner)
{
    $this->_enabled=true;
    $this->_owner=$owner; // 行为的拥有者对象
    $this->_attachEventHandlers(); // 绑定事件
}

// 解除行为的附加
public function detach($owner)
{
    foreach($this->events() as $event=>$handler)
        $owner->detachEventHandler($event,array($this,$handler));
    $this->_owner=null;
    $this->_enabled=false;
}

public function getOwner()
{
    return $this->_owner;
}

public function getEnabled()
{
    return $this->_enabled;
}

public function setEnabled($value)
{
    $value=(bool)$value;
    // $this->_owner 为 true 表示行为已经 attach
    if($this->_enabled!=$value && $this->_owner)
    {
        if($value) // 绑定行为事件
            $this->_attachEventHandlers();
        else
        {
            // 解除行为事件
            foreach($this->events() as $event=>$handler)
                $this->_owner->detachEventHandler($event,array($this,$handler));
        }
    }
```

```
        $this->_enabled=$value;
    }

    private function _attachEventHandlers()
    {
        $class=new ReflectionClass($this);
        foreach($this->events() as $event=>$handler)
        {
            // 事件绑定的方法必须是 public 申明的方法
            if($class->getMethod($handler)->isPublic())
                $this->_owner->attachEventHandler($event,array($this,$handler));
        }
    }
}
```

行为的附加

组件调用 CComponent:: attachBehavior 将行为附加，*CComponent* 类将行为对象保存到$_m 属性中。

```
public function attachBehavior($name,$behavior)
{
     // 形参$behavior 为配置数组时，先创建行为对象
    if(!($behavior instanceof IBehavior))
        $behavior=Yii::createComponent($behavior);
    $behavior->setEnabled(true); // 置_enabled=true
    $behavior->attach($this); // 附加行为
    return $this->_m[$name]=$behavior; // 将行为对象保存到$_m 属性中
}
```

CBehavior::attach 方法调用 CComponent::attachEventHandlers 方法将行为方法绑定到组件事件，

```
public function attachEventHandler($name,$handler)
{
     // 首先判断组件是否存在事件(事件保存在$_e 属性中)，如果有就添加事件句柄
    $this->getEventHandlers($name)->add($handler);
}
```

行为附加到组件后，为什么能直接访问行为的属性和方法，其原理是通过组件基类 CComponent 提供的__get、__set 和__call 魔术方法。

```
public function __get($name)
{
    $getter='get'.$name;
    if(method_exists($this,$getter)) // 判断类内是否存在 getter 方法
        return $this->$getter();
```

```
    // 事件
    elseif(strncasecmp($name,'on',2)===0 && method_exists($this,$name))
    {
        // duplicating getEventHandlers() here for performance
        $name=strtolower($name);
        if(!isset($this->_e[$name]))
            $this->_e[$name]=new CList;
        return $this->_e[$name];
    }
    elseif(isset($this->_m[$name])) // $_m 中放存附加的行为类对象
        return $this->_m[$name]; // 返回行为类对象
    elseif(is_array($this->_m))
    {
        foreach($this->_m as $object)
        {
            // 行为使能的前提下，检查那个行为类中存在属性或方法
            if($object->getEnabled()&&(property_exists($object,$name)|| $object->canGetProperty($name)))
                return $object->$name; //返回属性或调用方法
        }
    }
    throw new CException(Yii::t('yii','Property "{class}.{property}" is not defined.',
        array('{class}'=>get_class($this), '{property}'=>$name)));
}

public function __set($name,$value)
{
    $setter='set'.$name;
    // 检查类是否存在属性 set 方法，如：setImport，如果存在直接调用 set 方法
    if(method_exists($this,$setter))
        return $this->$setter($value);
    // 注册以'on'开头的事件，如：onBeginRequest 事件，事件会被添加到链表的尾部
    elseif(strncasecmp($name,'on',2)===0 && method_exists($this,$name))
    {
        // duplicating getEventHandlers() here for performance
        $name=strtolower($name);
        if(!isset($this->_e[$name]))
            $this->_e[$name]=new CList;
        return $this->_e[$name]->add($value);
    }
    elseif(is_array($this->_m)) // _m 存放行为对象
    {
        // 遍历绑定的行为
        foreach($this->_m as $object)
```

```
        {
            // 行为类中是否存在属性或者 getter 函数
            if($object->getEnabled()&&(property_exists($object,$name) || $object->canSetProperty($name)))
                return $object->$name=$value;
        }
    }
    // 配置项无法解析时抛出异常
    if(method_exists($this,'get'.$name))
        throw new CException(Yii::t('yii','Property "{class}.{property}" is read only.',
            array('{class}'=>get_class($this), '{property}'=>$name)));
    else
        throw new CException(Yii::t('yii','Property "{class}.{property}" is not defined.',
            array('{class}'=>get_class($this), '{property}'=>$name)));
}

public function __call($name,$parameters)
{
    if($this->_m!==null)
    {
        // 遍历所有行为对象
        foreach($this->_m as $object)
        {
            if($object->getEnabled() && method_exists($object,$name))
                return call_user_func_array(array($object,$name),$parameters);
        }
    }
    if(class_exists('Closure', false) && ($this->canGetProperty($name) || property_exists($this, $name)) &&
$this->$name instanceof Closure)
        return call_user_func_array($this->$name, $parameters);
 throw new CException(Yii::t('yii','{class} and its behaviors do not have a method or closure named
"{name}".',array('{class}'=>get_class($this), '{name}'=>$name)));
}
```

行为的解绑

调用 CComponent::detachBehaviors 函数解除组件的所有行为，或调用 CComponent::detachBehavior 解除单个行为。

```
public function detachBehaviors()
{
    if($this->_m!==null)
    {
        foreach($this->_m as $name=>$behavior)
```

```
            $this->detachBehavior($name);
        $this->_m=null;
    }
}
public function detachBehavior($name)
{
    if(isset($this->_m[$name]))
    {
        // 解绑行为的事件
        $this->_m[$name]->detach($this);
        $behavior=$this->_m[$name];
        // 将行为从$_m 中删除
        unset($this->_m[$name]);
        return $behavior;
    }
}
```

解除行为时调用 CBehavior:: detach 方法解除绑定的事件：

```
public function detach($owner)
{
    foreach($this->events() as $event=>$handler)
        $owner->detachEventHandler($event,array($this,$handler));
    $this->_owner=null;
    $this->_enabled=false;
}
```

而$owner->detachEventHandler 方法是这样的：

```
public function detachEventHandler($name,$handler)
{
    if($this->hasEventHandler($name))
        return $this->getEventHandlers($name)->remove($handler)!==false;
    else
        return false;
}
```

第三章 错误和日志处理

1 错误处理

1.1 异常处理

在 CApplication::__construct 函数中调用 set_exception_handler 注册了异常处理函数，代码如下：

```
if(YII_ENABLE_EXCEPTION_HANDLER)
    set_exception_handler(array($this,'handleException'));
```

可以通过将 YII_ENABLE_EXCEPTION_HANDLER 宏置为 false 来禁止框架注册异常处理函数，当在代码中抛出异常时，handleException 函数会接管执行流程：

CApplication:: handleException:

```
public function handleException($exception)
{
    // disable error capturing to avoid recursive errors
    // 还原到系统内置的错误处理函数
    restore_error_handler();
    restore_exception_handler();

    // get_class 获取抛出的异常对象类名
    $category='exception.'.get_class($exception);
    // CHttpException 是 http 异常基类
    if($exception instanceof CHttpException)
        $category.='.'.$exception->statusCode;
    // php <5.2 doesn't support string conversion auto-magically
        // 获取异常堆栈
    $message=$exception->__toString();
    if(isset($_SERVER['REQUEST_URI']))
        $message.="\nREQUEST_URI=".$_SERVER['REQUEST_URI'];
    if(isset($_SERVER['HTTP_REFERER']))
        $message.="\nHTTP_REFERER=".$_SERVER['HTTP_REFERER'];
    $message.="\n---";
    // 根据 log 组件配置，输出$message
    Yii::log($message,CLogger::LEVEL_ERROR,$category);

    try
    {
        $event=new CExceptionEvent($this,$exception);
        // 触发 onException 事件，框架默认没有注册事件句柄，可通过配置文件注册句柄
        $this->onException($event);
        if(!$event->handled) // handled 属性标识是否允许后续句柄执行
        {
```

```
            // try an error handler
            //  使用 errorHandler 组件来处理异常
            if(($handler=$this->getErrorHandler())!==null)
                $handler->handle($event);
            else
                $this->displayException($exception);
        }
    }
    catch(Exception $e)
    {
        $this->displayException($e);
    }

    try
    {
         //  触发 onEndRequest 事件
        $this->end(1);
    }
    catch(Exception $e)
    {
        // use the most primitive way to log error
        $msg = get_class($e).': '.$e->getMessage().' ('.$e->getFile().':'.$e->getLine().")\n";
        $msg .= $e->getTraceAsString()."\n";
        $msg .= "Previous exception:\n";
        $msg.=get_class($exception).':'.$exception->getMessage().'
                    ('.$exception->getFile().':'.$exception->getLine().")\n";
        $msg .= $exception->getTraceAsString()."\n";
        $msg .= '$_SERVER='.var_export($_SERVER,true);
        error_log($msg);
        exit(1);
    }
}
```

handleException 函数调用 errorHandler 组件的 handle 方法将异常输出到浏览器,

CErrorHandler::handle

```
public function handle($event)
{
    // set event as handled to prevent it from being handled by other event handlers
    $event->handled=true;

    //  等于 true 表示丢弃输出缓存区中的所有内容
    if($this->discardOutput)
    {
```

```
        ...
    }

    if($event instanceof CExceptionEvent)
        $this->handleException($event->exception); // 处理异常
    else // CErrorEvent
        $this->handleError($event); // 处理错误
}

protected function handleException($exception)
{
    $app=Yii::app();
    if($app instanceof CWebApplication)
    {
        // 获取抛出异常的文件名和行号
        ...

        if(!headers_sent())
        {
            $httpVersion=Yii::app()->request->getHttpVersion();
            header("HTTP/$httpVersion{$data['code']}".$this->getHttpHeader($data['code'],
                    get_class($exception)));
        }

        // 当 errorHandler 事件有配置 errorAction 属性时，就调用 errorAction 输出错误，否则调用框架的
错误处理函数输出
        $this->renderException();
    }
    else
        $app->displayException($exception);
}
```

由此可见，yii 的异常处理比较复杂，线上环境可以配置 errorAction 配置项来自定义输出异常信息。

1.2 错误处理

跟注册异常处理函数类似，错误处理函数的注册语句如下：

```
if(YII_ENABLE_ERROR_HANDLER)
    set_error_handler(array($this,'handleError'),error_reporting());
```

set_error_handler 函数参数 2 用于设置在那些错误类型下调用触发自定义错误处理函数，当没有设置此参

数时，自定义错误处理函数会在每个错误发生时都被调用，但以下级别的错误类型不能由自定义错误处理函数来处理：E_ERROR、E_PARSE、 E_CORE_ERROR、 E_CORE_WARNING、 E_COMPILE_ERROR、E_COMPILE_WARNING。

CComponent::handleError 方法跟 CComponent::handleException 类似，收集错误信息并输出到文件和浏览器

CComponent::handlerError:

```
public function handleError($code,$message,$file,$line)
{
      // $code 是错误级别
    if($code & error_reporting())
    {
        // disable error capturing to avoid recursive errors
                // 还原到系统内置的错误处理函数
        restore_error_handler();
        restore_exception_handler();

        $log="$message ($file:$line)\nStack trace:\n";
        $trace=debug_backtrace();
        // skip the first 3 stacks as they do not tell the error position
        if(count($trace)>3)
            $trace=array_slice($trace,3);
        foreach($trace as $i=>$t)
        {
            if(!isset($t['file']))
                $t['file']='unknown';
            if(!isset($t['line']))
                $t['line']=0;
            if(!isset($t['function']))
                $t['function']='unknown';
            $log.="#$i {$t['file']}({$t['line']}): ";
            if(isset($t['object']) && is_object($t['object']))
                $log.=get_class($t['object']).'->';
            $log.="{$t['function']}()\n";
        }
        if(isset($_SERVER['REQUEST_URI']))
            $log.='REQUEST_URI='.$_SERVER['REQUEST_URI'];
        // 根据 log 组件配置，输出 log
        Yii::log($log,CLogger::LEVEL_ERROR,'php');

        try
```

```
        {
            Yii::import('CErrorEvent',true);
            $event=new CErrorEvent($this,$code,$message,$file,$line);
                    // 触发 onError 事件，默认框架没有注册此事件句柄，可通过配置文件注册句柄
            $this->onError($event);
            if(!$event->handled) // handled 属性标识是否允许后续句柄执行
            {
                // try an error handler
                        // 使用 errorHandler 组件来处理异常
                if(($handler=$this->getErrorHandler())!==null)
                    $handler->handle($event);
                else
                    $this->displayError($code,$message,$file,$line);
            }
        }
        catch(Exception $e)
        {
            $this->displayException($e);
        }

        try
        {
                    // 触发 onEndRequest 事件
            $this->end(1);
        }
        catch(Exception $e)
        {
            // use the most primitive way to log error
            $msg = get_class($e).': '.$e->getMessage().' ('.$e->getFile().':'.$e->getLine().")\n";
            $msg .= $e->getTraceAsString()."\n";
            $msg .= "Previous error:\n";
            $msg .= $log."\n";
            $msg .= '$_SERVER='.var_export($_SERVER,true);
            error_log($msg);
            exit(1);
        }
    }
}
```

CErrorHandler.php:

```
public function handle($event)
{
```

```
    ...
    if($event instanceof CExceptionEvent)
        $this->handleException($event->exception);
    else // CErrorEvent
        $this->handleError($event); //  处理错误
}

protected function handleError($event)
{
    $trace=debug_backtrace();
    // skip the first 3 stacks as they do not tell the error position
    if(count($trace)>3)
        $trace=array_slice($trace,3);
    $traceString='';
    foreach($trace as $i=>$t)
    {
        if(!isset($t['file']))
            $trace[$i]['file']='unknown';

        if(!isset($t['line']))
            $trace[$i]['line']=0;

        if(!isset($t['function']))
            $trace[$i]['function']='unknown';

        $traceString.="#$i {$trace[$i]['file']}({$trace[$i]['line']}): ";
        if(isset($t['object']) && is_object($t['object']))
            $traceString.=get_class($t['object']).'->';
        $traceString.="{$trace[$i]['function']}()\n";

        unset($trace[$i]['object']);
    }

    $app=Yii::app();
    if($app instanceof CWebApplication)
    {
        switch($event->code)
        {
            case E_WARNING:
                $type = 'PHP warning';
                break;
            case E_NOTICE:
                $type = 'PHP notice';
```

```
                break;
            case E_USER_ERROR:
                $type = 'User error';
                break;
            case E_USER_WARNING:
                $type = 'User warning';
                break;
            case E_USER_NOTICE:
                $type = 'User notice';
                break;
            case E_RECOVERABLE_ERROR:
                $type = 'Recoverable error';
                break;
            default:
                $type = 'PHP error';
        }
        $this->_exception=null;
        $this->_error=array(
            'code'=>500,
            'type'=>$type,
            'message'=>$event->message,
            'file'=>$event->file,
            'line'=>$event->line,
            'trace'=>$traceString,
            'traces'=>$trace,
        );
        if(!headers_sent())
        {
            $httpVersion=Yii::app()->request->getHttpVersion();
            header("HTTP/$httpVersion 500 Internal Server Error");
        }

        // 当 errorHandler 事件有配置 errorAction 属性时，就调用 errorAction 输出错误，否则调用框架的
错误处理函数输出
        $this->renderError();
    }
    else
        $app->displayError($event->code,$event->message,$event->file,$event->line);
}
```

2 日志处理

Yii 提供了 Yii::log 和 Yii::trace 方法来记录日志，后者只能在 Debug 模式下才会记录信息。

Yii::log($message, $level, $category);
Yii::trace($message, $category);

记录日志时，$level 指日志的级别，如：

CLogger::LEVEL_TRACE: 用于在开发中 跟踪程序的执行流程

CLogger::LEVEL_INFO: 用于记录普通的信息

CLogger::LEVEL_WARNING: 用于警告（warning）信息

CLogger::LEVEL_ERROR: 用于致命错误（fatal error）信息

CLooger:: LEVEL_PROFILE: 指性能监控日志

$category 指日志的分类，类似 xxx.yyy.zzz 格式的，我们可以将其视为一个分类层级。 具体地，我们说 xxx 是 xxx.yyy 的父级，而 xxx.yyy 又是 xxx.yyy.zzz 的父级。这样我们就可以使用 xxx.* 表示分类 xxx 及其所有的子级和孙级分类。

Yii::log 和 Yii::trace 默认将日志保存在内存中，可以通过日志路由将日志路由到目的地(如 :输出到文件或发送到 email 中)。要想使用日志路由，需配置 log 组件并使用 preload 配置项预加载 log 组件。如：

```
[
    ......
    'preload'=>['log'],
    'components'=>[
        ......
        'log'=>[
            'class'=>'CLogRouter',
            'routes'=>[
                [
                    'class'=>'CFileLogRoute',
                    'levels'=>'info,warning',
                    'logFile'=>'app.log',
                ],
                [
                    'class'=>'CEmailLogRoute',
                    'levels'=>'error',
                    'categories'=>'system.*',
                    'except'=>'system.abc',
                    'emails'=>'admin@example.com',
```

```
                ],
            ],
        ],
    ],
]
```

上面配置了两个日志路由，第一个是 CFileLogRoute，它会将 info 和 warning 级别的日志保存到应用程序 runtime 目录下的 app.log 文件中。第二个是 CEmailLogRoute，它将等级为 error、分类以 system 开头且不等于 system.abc 的日志发送到指定的 email 地址中。

Yii 框架提供了以下几种日志路由：

CDbLogRoute: 将信息保存到数据库的表中。

CEmailLogRoute: 发送信息到指定的 Email 地址。

CFileLogRoute: 保存信息到应用程序 runtime 目录中的一个文件中。

CWebLogRoute: 将信息显示在当前页面的底部。

CsysLogRoute：将信息输出到 syslog 中。

CProfileLogRoute: 在页面的底部显示概述（profiling）信息。

2.1 日志路由的流程

日志路由是根据路由规则将日志路由到目的地，一条日志支持可以路由到多个目的地，日志发送到目的地后会从内存中清除，为避免频繁写入文件(或路由到其它目的地)，Yii 支持在内存中保存日志，满足一定条件后或请求结束前将日志写入文件(或路由到其它目的地)。

在框架的启动过程中，通过 preload 配置项预加载 log 组件时调用 CLogRouter::init()方法注册事件句柄。

CLogRouter::init():

```
public function init()
{
    parent::init(); // 调用父类 init 方法附加行为
    // 解析 log 路由规则
    foreach($this->_routes as $name=>$route)
    {
        // 创建日志路由对象
        $route=Yii::createComponent($route);
        // 调用路由对象的 init 方法
        $route->init();
        $this->_routes[$name]=$route;
```

```
    }
    // 注册 CLogger 组件的 onFlush 事件句柄
    Yii::getLogger()->attachEventHandler('onFlush',array($this,'collectLogs'));
    // 注册框架的 onEndRequest 事件句柄,processLogs 方法会触发 onFlush 事件
    Yii::app()->attachEventHandler('onEndRequest',array($this,'processLogs'));
}
```

从 Yii::log 函数开始分析 Yii 框架记录日志的流程，Yii::log 方法默认是 info 级别日志,在 Debug 模式下会输出当前堆栈。

Yii::log:

```
public static function log($msg,$level=CLogger::LEVEL_INFO,$category='application')
{
    if(self::$_logger===null)
        self::$_logger=new CLogger;
    // DEBUG 模式下打印堆栈
    if(YII_DEBUG && YII_TRACE_LEVEL>0 && $level!==CLogger::LEVEL_PROFILE)
    {
        $traces=debug_backtrace();
        $count=0;
        foreach($traces as $trace)
        {
            if(isset($trace['file'],$trace['line']) && strpos($trace['file'],YII_PATH)!==0)
            {
                $msg.="\nin ".$trace['file'].' ('.$trace['line'].')';
                // YII_TRACE_LEVEL 指示打印堆栈的层数
                if(++$count>=YII_TRACE_LEVEL)
                    break;
            }
        }
    }
    self::$_logger->log($msg,$level,$category);
}
```

CLogger:log()将日志缓存到内存中，并根据条件触发 onFlush 事件。日志是否被立即写入文件(或发送到其它媒介)中依赖 CLogger 的 autoFlush 和 autoDump 属性，当内存中的日志条数超过 autoFlush 并且 autoDump 等于 true 时日志才会真正写入文件(或发送到其它媒介)，否则仍然保存在内存中。

CLogger::log():

```
public function log($message,$level='info',$category='application')
{
    // 将日志缓存到内存中
```

```
    $this->_logs[]=array($message,$level,$category,microtime(true));
    $this->_logCount++;
    // autoFlush：指示达到多少条消息后自动 flush 日志，默认为 10000 条，值为 0 表示不自动 flush
    // _processing：是否正在 flush 日志，默认为 false
    if($this->autoFlush>0 && $this->_logCount>=$this->autoFlush && !$this->_processing)
    {
        $this->_processing=true;
        // autoDump：默认为 false，false：日志经过路由规则解析后仍保存在内存中，true：日志立即被写入路由后的媒介
        $this->flush($this->autoDump);
        $this->_processing=false;
    }
}

public function flush($dumpLogs=false)
{
    // 触发 onFlush 事件，当触发 onEndRequest 事件时 CLogrouter::processLogs 调用 flush 时$dumpLogs 等于 true
    $this->onFlush(new CEvent($this, array('dumpLogs'=>$dumpLogs)));
    // 清空内存中的日志
    $this->_logs=array();
    $this->_logCount=0;
}
```

CLogRouter:: collectLogs 是 onFlush 事件的句柄，它循环调用每个路由并根据 level、category、except 过滤日志，将满足条件的日志保存在内存，同一条日志可以被多个路由保存。

CLogRouter::collectLogs:

```
public function collectLogs($event)
{
    $logger=Yii::getLogger();
    $dumpLogs=isset($event->params['dumpLogs']) && $event->params['dumpLogs'];
    foreach($this->_routes as $route)
    {
        /* @var $route CLogRoute */
        // 调用每个日志路由的 collectLogs 函数
        if($route->enabled)
            $route->collectLogs($logger,$dumpLogs);
    }
}
```

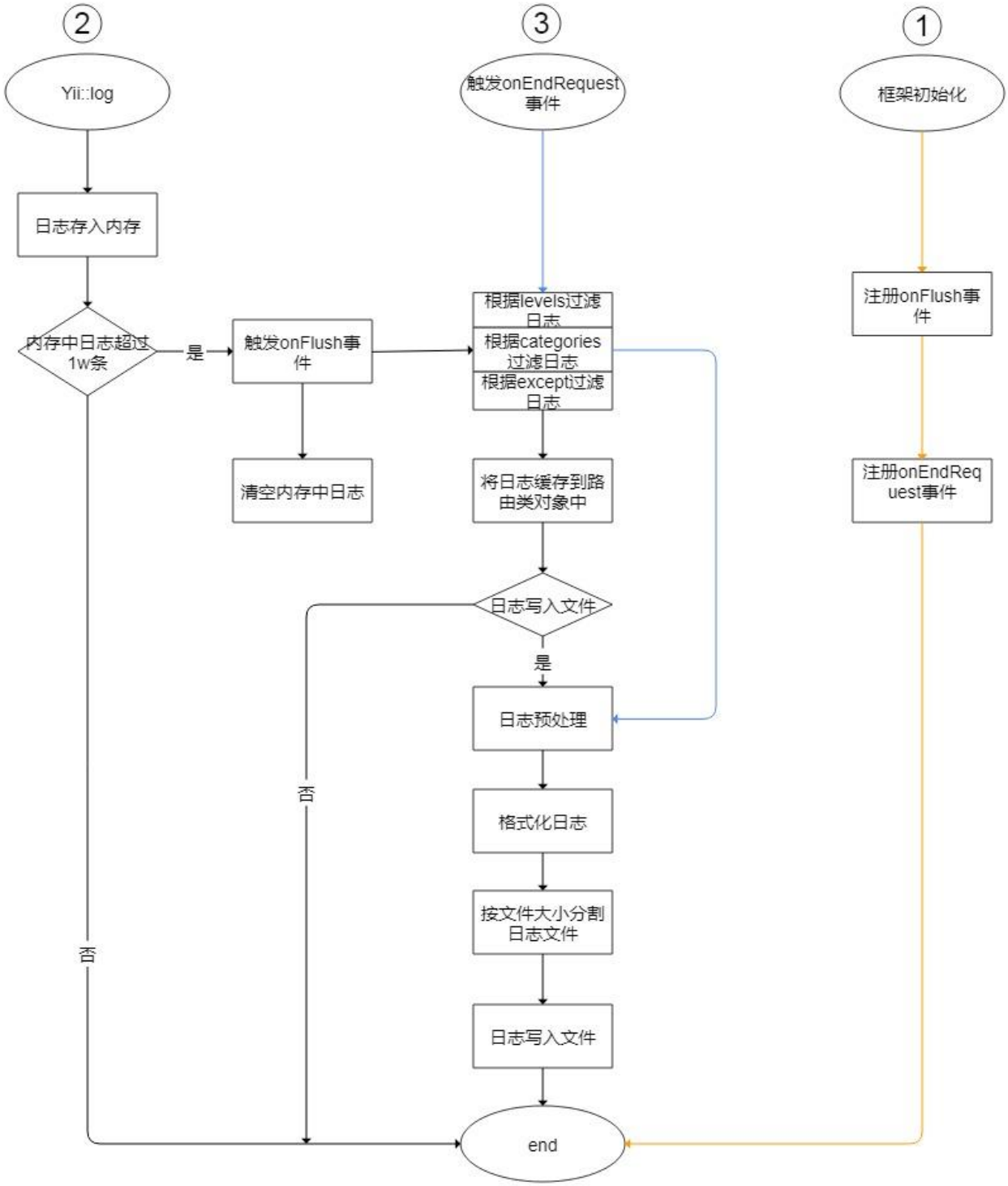
我们以 CFileLogRoute 路由类为例，CFileLogRoute 类继承 CLogRoute 类，它调用父类的 collectLogs 函数来响应 onFlush 事件，并调用 CFileLogRoute::processLogs 将日志写到文件中

```
public function collectLogs($logger, $processLogs=false)
```

```
{
     // 过滤掉不满足路由规则的日志
    $logs=$logger->getLogs($this->levels,$this->categories,$this->except);
    // 日志在写入文件前可多次触发 onFlush 事件，这里将多次事件的 log 合并
    $this->logs=empty($this->logs) ? $logs : array_merge($this->logs,$logs);
    // $processLogs  true 表示调用 processLogs 将日志路由到目的地， false 表示暂不路由日志还会保存在内存中。
    if($processLogs && !empty($this->logs))
    {
         // 是否配置了日志过滤，稍后讲解
        if($this->filter!==null)
            Yii::createComponent($this->filter)->filter($this->logs);
        if($this->logs!==array())
            $this->processLogs($this->logs); // 负责格式化日志并对日志进行路由
              // 清空本路由类的日志内存
        $this->logs=array();
    }
}
```

当内存中的日志并没有达到 1w 条(参考：CLogger:: autoFlush)时，并且请求结束了怎么办。请求结束时会主动触发 onEndRequest 事件，此事件触发 onFlush 事件将日志写入文件(或路由到其它目的地)。

日志路由流程图(以 CFileLogRoute 类为例)：



2.2 日志预处理

日志每次路由到目的地前可以进行预处理，比如在每条日志前加上 traceid，在日志前加上请求上下文等。

预处理通过 filter 配置项来实现，如：

```
'log' => array(
    'class' => 'CLogRouter',
    'routes' => array(
        array(
            'class' => 'CFileLogRoute',
            'levels' => 'warning',
            'filter' => 'CLogFilter',
        ),
    ),
),
```

CLogFilter 是 Yii 框架实现的预处理类，在触发 onFlush 日志，并将日志路由到目的地前调用以下语句对日志预处理。

```
if($this->filter!==null)
    Yii::createComponent($this->filter)->filter($this->logs);
```

CLogFilter::filter:

```
public function filter(&$logs)
{
    if (!empty($logs))
    {
        if(($message=$this->getContext())!=='')
            // 在所有日志的前面加上$messge，注意：并不是每条日志前面都加
            array_unshift($logs,array($message,CLogger::LEVEL_INFO,'application',YII_BEGIN_TIME));
        $this->format($logs);
    }
    return $logs;
}

protected function getContext()
{
    $context=array();
    // 获取用户信息
    if($this->logUser && ($user=Yii::app()->getComponent('user',false))!==null)
        $context[]='User: '.$user->getName().' (ID: '.$user->getId().')';

    if($this->dumper==='var_export' || $this->dumper==='print_r')
    {
        // $logVars=array('_GET','_POST','_FILES','_COOKIE','_SESSION','_SERVER')
            // 获取全局变量的值
        foreach($this->logVars as $name)
```

```
            if(($value=$this->getGlobalsValue($name))!==null)
                $context[]="\${$name}=".call_user_func($this->dumper,$value,true);
    }
    else
    {
        foreach($this->logVars as $name)
            if(($value=$this->getGlobalsValue($name))!==null)
                $context[]="\${$name}=".call_user_func($this->dumper,$value);
    }

    // 将数组转为字符串
    return implode("\n\n",$context);
}

protected function format(&$logs)
{
    $prefix='';
    // 是否在每条日志前加上 sessionId
    if($this->prefixSession && ($id=session_id())!=='')
        $prefix.="[$id]";
    // 是否在每条日志前加上用户信息
    if($this->prefixUser && ($user=Yii::app()->getComponent('user',false))!==null)
        $prefix.='['.$user->getName().']['.$user->getId().']';
    if($prefix!=='')
    {
         // 将$prefix 附加到每条日志的前面
        foreach($logs as &$log)
            $log[0]=$prefix.' '.$log[0];
    }
}
```

除了使用 CLogFilter 来预处理日志外，我们还可以自定义类来处理，自定义类需继承 CComponent 类实现 IlogFilter 接口。

2.3 CFileLogRoute

CFileLogRoute 类将日志存储到文件中，默认情况日志文件大小超过 1M 就进行分割，分割的文件名在原文件名上加上后缀.[1|2|3|4|5], 当分割的文件数超过 5 个时, 新分割的文件会覆盖老文件。假设当前有 app.log、app.log.1、app.log.2、app.log.3、app.log.4、app.log.5，当 app.log 文件大小超过 1M 后，app.log.5 文件

会被 app.log.4 覆盖，app.log.4 被 app.log.3 覆盖等等，最后将 app.log 文件重命名为 app.log.1。

CFileLogRoute 类提供了以下属性可以配置文件中修改属性值:

maxFileSize：日志文件的大小(单位:kB)，超过此值会进行分割，最小值 1kB

maxLogFiles：最多能分割多少个日志文件，默认 5 个，最小值 1 个。

logPath：日志文件路径，默认是 runtime 路径

logFile：文件名，默认为 application.log

rotateByCopy：在分割日志的过程中，是否直接将日志他复制到分割的文件中，并清空当前文件。

例如：将 app.log 文件中的内容复制到 app.log.1 中，并清空 app.log。

CFileLogRoute 类没有实现 collectLogs 方法，使用父类的 collectLogs 方法来响应 onFlush 事件。

CFileLogRoute.php:

```
protected function processLogs($logs)
{
    $text='';
    foreach($logs as $log)
        // 调用父类的方法格式化日志，日志格式如：
        // 2020/06/09 17:34:14 [$level] [$category] $message
        $text.=$this->formatLogMessage($log[0],$log[1],$log[2],$log[3]);

    // 获取日志文件绝对路径
    $logFile=$this->getLogPath().DIRECTORY_SEPARATOR.$this->getLogFile();
    // 以追加方式打开文件
    $fp=@fopen($logFile,'a');
    // 对文件加独占锁
    @flock($fp,LOCK_EX);
    // 是否大小 1M
    if(@filesize($logFile)>$this->getMaxFileSize()*1024)
    {
        // 分割文件
        $this->rotateFiles();
        // 解锁文件
        @flock($fp,LOCK_UN);
        // 关闭文件
        @fclose($fp);
        // 将日志写入文件，写入时加锁，这里不使用 fwrite 写入的原因是 rotateFiles 可能将文件重命名了
        @file_put_contents($logFile,$text,FILE_APPEND|LOCK_EX);
```

```
    }
    else
    {
        // 将日志写入文件
        @fwrite($fp,$text);
        @flock($fp,LOCK_UN);
        @fclose($fp);
    }
}

protected function rotateFiles()
{
    $file=$this->getLogPath().DIRECTORY_SEPARATOR.$this->getLogFile();
    // 得到分割的最大文件数，默认为 5
    $max=$this->getMaxLogFiles();
    for($i=$max;$i>0;--$i)
    {
        $rotateFile=$file.'.'.$i;
        if(is_file($rotateFile))
        {
            // suppress errors because it's possible multiple processes enter into this section
            if($i===$max)
                // 当分割后的文件个数超过 5 个时，删除第 5 个
                @unlink($rotateFile);
            else // 文件重命名，如：app.log.1->app.log.2
                @rename($rotateFile,$file.'.'.($i+1));
        }
    }
    // 上面的 for 循环只处理了分割的文件，原文件并没有处理
    if(is_file($file))
    {
        // suppress errors because it's possible multiple processes enter into this section
        // 默认为 false
        if($this->rotateByCopy)
        {
            // 先将$file 中的文件复制到分割后的文件中，再将当前文件清空
            @copy($file,$file.'.1');
            if($fp=@fopen($file,'a'))
            {
                @ftruncate($fp,0);
                @fclose($fp);
            }
        }
```

```
        else // 重命名原文件
            @rename($file,$file.'.1');
    }
    // clear stat cache after moving files so later file size check is not cached
    clearstatcache();
}
```

2.4 CDbLogRoute

CDbLogRoute 将日志保存**到**数据库中，如果不指定数据库，默认将保存到 sqlite 数据库中。由于每条日志都 insert 一次，当日志量太大时会给 db 带来压力。

CDbLogRoute 提供了以下配置项可在配置文件中配置：

connectionID：指定 db 组件 id，CDbLogRoute 通过此组件访问 db

logTableName：存放日志的表名，默认为 YiiLog

autoCreateLogTable：是否提前创建日志表，默认为 true

在加载 log 组件解析配置项时调用 init 方法：

CDbLogRoute.php

```
public function init()
{
    parent::init();

    if($this->autoCreateLogTable)
    {
        // 获取 db 连接
        $db=$this->getDbConnection();
        try
        {
            // 测试 db 是否访问
            $db->createCommand()->delete($this->logTableName,'0=1');
        }
        catch(Exception $e)
        {
            // 创建表，默认表名为:YiiLog
            $this->createLogTable($db,$this->logTableName);
```

```
        }
    }
}

protected function getDbConnection()
{
    if($this->_db!==null)
        return $this->_db;
    // 通过指定的 db 组件访问数据库
    elseif(($id=$this->connectionID)!==null)
    {
        if(($this->_db=Yii::app()->getComponent($id)) instanceof CDbConnection)
            return $this->_db;
        else
            throw new CException(Yii::t('yii','CDbLogRoute.connectionID "{id}" does not point to a valid CDbConnection application component.',
                array('{id}'=>$id)));
    }
    else
    {
         // 默认访问 sqlite 数据库
        $dbFile=Yii::app()->getRuntimePath().DIRECTORY_SEPARATOR.'log-'.Yii::getVersion().'.db';
        return $this->_db=new CDbConnection('sqlite:'.$dbFile);
    }
}
```

路由日志的方法：

CDbLogRoute:: processLogs:

```
protected function processLogs($logs)
{
    $command=$this->getDbConnection()->createCommand();
    foreach($logs as $log)
    {
         // 每条语句一次 insert，当日志量较大时，会给 db 带来很大的压力
        $command->insert($this->logTableName,array(
            'level'=>$log[1],
            'category'=>$log[2],
            'logtime'=>(int)$log[3],
            'message'=>$log[0],
        ));
    }
}
```

2.5 CEmailLogRoute

CEmailLogRoute 将日志发送到指定的邮件中，每次触发 onFlush 时会将内存中的日志格式化成一封邮件发送给每位接收者。

CEmailLogRoute 类提供以下的配置项：

utf8：邮件是否 utf-8 编码，默认为 false

email：收件人，格式如下：

'user@example.com'

'user@example.com, anotheruser@example.com'

'User <user@example.com>'

'User <user@example.com>, Another User <anotheruser@example.com>'

或者

['user@example.com', 'anotheruser@example.com']

subject：邮件主题

from：发件人，格式参考收件人，但不支持数组格式

headers：发送邮件的请求头

路由日志的方法：

```
protected function processLogs($logs)
{
    $message='';
    foreach($logs as $log)
        $message.=$this->formatLogMessage($log[0],$log[1],$log[2],$log[3]);
    // 将消息折行处理，每行 70 个字符
    $message=wordwrap($message,70);
    // 获取主题
    $subject=$this->getSubject();
    if($subject===null)
        $subject=Yii::t('yii','Application Log');
    foreach($this->getEmails() as $email)
        $this->sendEmail($email,$subject,$message);
}

protected function sendEmail($email,$subject,$message)
{
```

```
    // 存放发送邮件的请求头
    $headers=$this->getHeaders();
    if($this->utf8)
    {
        $headers[]="MIME-Version: 1.0";
        $headers[]="Content-Type: text/plain; charset=UTF-8";
        // MIME 编码格式，编码用=?和?=括起来，UTF-8 表示字符，B 表示用 base64 加密
        $subject='=?UTF-8?B?'.base64_encode($subject).'?=';
    }
    if(($from=$this->getSentFrom())!==null)
    {
        $matches=array();
        // $from 格式：example<example@126.com> 或 <example@126.com> 或 example@126.com
        preg_match_all('/([^<]*)<([^>]*)>/iu',$from,$matches);
        if(isset($matches[1][0],$matches[2][0]))
        {
            $name=$this->utf8?'=?UTF-8?B?'.base64_encode(trim($matches[1][0])).'?='
                    : trim($matches[1][0]);
            $from=trim($matches[2][0]);
            $headers[]="From: {$name} <{$from}>";
        }
        else
            $headers[]="From: {$from}";
        $headers[]="Reply-To: {$from}";
    }
    // 发送邮件, $message 每行不能超过 70 个字符
    mail($email,$subject,$message,implode("\r\n",$headers));
}
```

2.6 CSysLogRoute

将日志发送到 syslog 中，默认 syslog 日志存储在/var/log/messages 文件中。

syslog 提供了以下配置项：

identity：添加在每条日志前面的字符标识。

facility：日志的分类，默认为 LOG_SYSLOG，值参考如下：

LOG_EMERG

LOG_ALERT

LOG_CRIT

LOG_ERR

LOG_WARNING

LOG_NOTICE

LOG_INFO

LOG_DEBUG

路由日志的方法：

```
protected function processLogs($logs)
{
    static $syslogLevels=array(
        CLogger::LEVEL_TRACE=>LOG_DEBUG,
        CLogger::LEVEL_WARNING=>LOG_WARNING,
        CLogger::LEVEL_ERROR=>LOG_ERR,
        CLogger::LEVEL_INFO=>LOG_INFO,
        CLogger::LEVEL_PROFILE=>LOG_DEBUG,
    );

    // 打开一个系统日志连接, $identity 被添加到每条消息的前面，$facility 表示日志的类别
    openlog($this->identity,LOG_ODELAY|LOG_PID,$this->facility);
    foreach($logs as $log)
        // 生成一条系统日志

syslog($syslogLevels[$log[1]],$this->formatLogMessage(str_replace("\n",',',$log[0]),$log[1],$log[2],$log[3]));
    closelog();
}
```

第四章 国际化

Yii 提供了 I18N 服务，支持信息和文件的翻译，以及本地化服务。Yii 中定义了区域的概念，区域 id(LocaleID)由语言 id+地区 id 组成，例如，区域 ID en_us 表示英语地区和美国。

1 信息翻译

将信息从源语言翻译成目标语言 Yii 提供了 Yii::t()函数，源语言和目标语言可以使用配置项配置，如：在 main.php 文件中添加如下内容：

```
return {
    ......
    'sourceLanguage' => 'en_us',  // 源语言
    'language' => 'zh_cn',  // 目标语言
    ......
}
```

Yii::t()函数原型如下：

Yii:: ($category,$message,$params=array(),$source=null,$language=null)

$category：信息的分类，一条信息在相同区域的情况下可以按分类来实现不同的翻译。

$message：被翻译的信息。

$params：信息可以包含占位符，此参数用来替代信息中的占位符，例如下面的信息在翻译后占位符{alias}被别名变量$alias 替换：

```
Yii::t('app', 'Path alias "{alias}" is redefined.', array('{alias}'=>$alias))
```

choice format 信息时，$params[0]值被赋值给特殊变量 n，参考下面的 choice format 介绍。

$source：翻译过的信息 Yii 称它为信息源，$source 用来指定处理翻译以及信息源的类，默认使用 CPhpMessageSource 做为信息源，它将翻过的信息存储在 PHP 的键值对数组中，翻译文件位于 application.messages.{$language}目录中，文件名为{$category}.php。

$language：指定目标语言，如果未指定，默认读取 language 配置项。

用 CPhpMessageSource 类来处理信息源时，扩展类（例如一个 widget 小物件，一个模块）中的信息可以以一种特殊的方式管理并使用。 具体来说，如果一条信息属于一个类名为 Xyz 的扩展，那么分类的名字可以以 Xyz.categoryName 的格式指定。 相应的信息文件就是 BasePath/messages/LocaleID/categoryName.php ，其中 BasePath 是指包含此扩展类文件的那个目录。 当使用 Yii::t() 翻译一条扩展信息时，需要使用如下格式：

Yii::t('Xyz.categoryName', '要翻译的信息');

Yii 支持 choice format，它的意思是根据表达式的值从多条翻译结果中选取一条，多条翻译结果由|分隔开，如：

'expr1#message1|expr2#message2|expr3#message3'

用#符号将 php 表达式和信息分开，结果选取过程中从左至右执行布尔表达式，当值为真时相应的信息会被选取，表达式中可以包括一个特殊变量 n(注意，不是 $n)，通过 Yii::t()函数第二个参数$param[0]传递，如：

'n==1#one book|n>1#many books'

例如当调用 Yii::t()时$param[0]=2，就会选取 many books 做为翻译的结果。当$param[0]值为数字时，表达式可以简写为如下形式，1 被视为 n==1。

'1#one book|n>1#many books'

总的来说，通过以下三步来实现信息翻译：

1） 在配置文件中添加 sourceLanguage 和 language 配置项。

2） BasePath/messages/LocaleID /categoryName.php 文件中记录翻译信息。

3） 调用 Yii::t()函数。

以下举例来说明：

在应用程序配置文件 main.php 中新增如下：

```
[
    ......
    'sourceLanguage' => 'en_us',  // 源语言
    'language' => 'zh_cn',  // 目标语言
    ......
]
```

假设项目 basePath 为/home/www/testyii，category 为 example，

在/home/www/testyii/messge/zh_cn/category.php 文件中添加以下内容：

```
return [
      'hello world'   => '你好',
]
```

执行以下语句将英文翻译成中文：

Yii::t('example', 'hello world')

信息翻译源代码流程：

在请求的处理过程中 CApplication::registerCoreComponents 函数注册了 coreMessages 和 messages 组件，两组件默认使用 CPhpMessageSource 做为处理类。Yii::t()函数首先找到翻译文件翻译信息，然后判断翻译结果是否是 choice format，如果是通过$params[0]参数计算各表达式的值，选择表达式值为 true 的信息做为最终的翻译结果。

Yii::t()：

```
public static function t($category,$message,$params=array(),$source=null,$language=null)
{
    if(self::$_app!==null)
    {
        if($source===null)
            // yii、zii 分类由 coreMessages 组件来处理，其余的分类由 messages 组件来处理
            $source=($category==='yii'||$category==='zii')?'coreMessages':'messages';
        if(($source=self::$_app->getComponent($source))!==null)
            // 默认调用 CPhpMessageSource::translate 翻译信息
            $message=$source->translate($category,$message,$language);
    }
    // $params 为空，不是 choice 格式
    if($params===array())
        return $message;
    if(!is_array($params))
        $params=array($params);
    if(isset($params[0])) // number choice
    {
        // 翻译结果: '1#one book|n>1#many books'
        if(strpos($message,'|')!==false)
        {
            // 当结果中不包含#符号时，通过区域的复数表达式来组合
            if(strpos($message,'#')===false)
            {
                $chunks=explode('|',$message);
                // 调用区域类 CLocale 获取区域的复数表达式，有些区域没有复数表达式
                $expressions=self::$_app->getLocale($language)->getPluralRules();
                if($n=min(count($chunks),count($expressions)))
```

```
                {
                    for($i=0;$i<$n;$i++)
                        // 组合结果
                        $chunks[$i]=$expressions[$i].'#'.$chunks[$i];

                    $message=implode('|',$chunks);
                }
            }
            // 记录表达式，选取翻译结果
            $message=CChoiceFormat::format($message,$params[0]);
        }
        if(!isset($params['{n}']))
            $params['{n}']=$params[0]; // 统一格式，下面采用 strtr 一起替换
        unset($params[0]);
    }
    // 替换占位符
    return $params!==array() ? strtr($message,$params) : $message;
}
```

调用 CPhpMessageSource::translate 翻译信息

```
public function translate($category,$message,$language=null)
{
    if($language===null)
        // 获取目标语言，如果未配置默认等于源语言
        $language=Yii::app()->getLanguage();
    // forceTranslation 表示无论源语言和目的语言是否一致，强制翻译
    if($this->forceTranslation || $language!==$this->getLanguage())
        return $this->translateMessage($category,$message,$language);
    else // 源语言和目的语言一致，返回原信息
        return $message;
}
protected function translateMessage($category,$message,$language)
{
    $key=$language.'.'.$category;
        // _messages 中存放翻译文件内容
    if(!isset($this->_messages[$key]))
        $this->_messages[$key]=$this->loadMessages($category,$language);
    // 匹配到翻译结果，直接返回
    if(isset($this->_messages[$key][$message]) && $this->_messages[$key][$message]!=='')
        return $this->_messages[$key][$message];
    // 当翻译结果不存在时，触发 onMissingTranslation 事件，默认 yii 没有注册此事件
    elseif($this->hasEventHandler('onMissingTranslation'))
    {
        $event=new CMissingTranslationEvent($this,$category,$message,$language);
```

```
        $this->onMissingTranslation($event);
        return $event->message;
    }
    else // 翻译结果不存在，直接返回原文
        return $message;
}
```

CPhpMessageSource.php:

```
protected function loadMessages($category,$language)
{
     // 获取翻译文件路径
    $messageFile=$this->getMessageFile($category,$language);

    // 如果配置了缓存就从缓存中获取
    if($this->cachingDuration>0
      &&$this->cacheID!==false
      && ($cache=Yii::app()->getComponent($this->cacheID))!==null)
    {
        $key=self::CACHE_KEY_PREFIX . $messageFile;
        if(($data=$cache->get($key))!==false)
            return unserialize($data);
    }

    if(is_file($messageFile))
    {    // 翻译文件必需用 return 返回一个数组
        $messages=include($messageFile);
        if(!is_array($messages))
            $messages=array();
        if(isset($cache))
        {
            $dependency=new CFileCacheDependency($messageFile);
            $cache->set($key,serialize($messages),$this->cachingDuration,$dependency);
        }
        return $messages;
    }
    else
        return array();
}
protected function getMessageFile($category,$language)
{
    if(!isset($this->_files[$category][$language]))
    {
         // 分类中包含扩展类名
```

```
        if(($pos=strpos($category,'.'))!==false)
        {
            $extensionClass=substr($category,0,$pos);
            $extensionCategory=substr($category,$pos+1);
            // First check if there's an extension registered for this class.
                    // 从扩展类路径中找类名
            if(isset($this->extensionPaths[$extensionClass]))

$this->_files[$category][$language]=Yii::getPathOfAlias($this->extensionPaths[$extensionClass]).DIRECTO
RY_SEPARATOR.$language.DIRECTORY_SEPARATOR.$extensionCategory.'.php';
            else
            {
                // No extension registered, need to find it.
                $class=new ReflectionClass($extensionClass);
                // 调用 getFileName 获取扩展类名所在文件的路径

$this->_files[$category][$language]=dirname($class->getFileName()).DIRECTORY_SEPARATOR.'messages'.
DIRECTORY_SEPARATOR.$language.DIRECTORY_SEPARATOR.$extensionCategory.'.php';
            }
        }
        else
            // 分类中不包含扩展名，在 application.messages 中查找文件

$this->_files[$category][$language]=$this->basePath.DIRECTORY_SEPARATOR.$language.DIRECTORY_SE
PARATOR.$category.'.php';
    }
    return $this->_files[$category][$language];
}
```

信息翻译后调用 CLocale:: getPluralRules()获取区域的复数表达式。

CLocale.php:

```
protected function __construct($id)
{
    $this->_id=strtolower(str_replace('-','_',$id));
    $dataPath=self::$dataPath===null ? dirname(__FILE__).DIRECTORY_SEPARATOR.'data' : self::$dataPath;
    // 在 Yii_PATH/i18n/data 目录下存放区域数据文件
    $dataFile=$dataPath.DIRECTORY_SEPARATOR.$this->_id.'.php';
    if(is_file($dataFile))
        $this->_data=require($dataFile);
    else
        throw new CException(Yii::t('yii','Unrecognized locale "{locale}".',array('{locale}'=>$id)));
}
```

```
public function getPluralRules()
{
    return isset($this->_data['pluralRules']) ? $this->_data['pluralRules'] : array(0=>'true');
}
```

从 choice format 中选取翻译结果

CchoiceFormat.php:

```
public static function format($messages, $number)
{
    $n=preg_match_all('/\s*([^#]*)\s*#([^\|]*)\|/',$messages.'|',$matches);
    if($n===0)
        return $messages;
    for($i=0;$i<$n;++$i)
    {
        $expression=$matches[1][$i];
        $message=$matches[2][$i];
        // 表达式是数字
        if($expression===(string)(int)$expression)
        {
             // 简便格式：'1#one book|n>1#many books'
                  // 简便表达式和参数相等时，说明匹配此信息
            if($expression==$number)
                return $message;
        }
        // 调用 eval 函数运行布尔表达式
        elseif(self::evaluate(str_replace('n','$n',$expression),$number))
            return $message;
    }
    return $message; // return the last choice
}

protected static function evaluate($expression,$n)
{
    try
    {
        return @eval("return $expression;");
    }
    catch (ParseError $e)
    {
        return false;
    }
}
```

2 文件翻译

Yii 的文件翻译比较简单，给定一个要翻译的文件路径，在区域 ID 子目录下查找同名的文件，如果找到，就返回此文件的路径，否则返回原文件路径。文件翻译主要用于渲染一个视图。 当在控制器或小物件中调用任一渲染方法时，视图文件将会被自动翻译。例如，如果 目标语言 是 zh_cn 而 源语言 是 en_us，渲染一个名为 edit 的视图时，程序将会查找 protected/views/ControllerID/zh_cn/edit.php 视图文件。 如果此文件找到，就会通过此翻译版本渲染。否则，就会使用文件 protected/views/ControllerID/edit.php 渲染。

Yii 的文件翻译由 CApplication::findLocalizedFile()来完成：

```
public function findLocalizedFile($srcFile,$srcLanguage=null,$language=null)
{
    if($srcLanguage===null)
        $srcLanguage=$this->sourceLanguage;
    if($language===null)
        $language=$this->getLanguage();
    if($language===$srcLanguage)
        return $srcFile;
    // $language 最好是小写的格式，如：en_us

$desiredFile=dirname($srcFile).DIRECTORY_SEPARATOR.$language.DIRECTORY_SEPARATOR.basename($src
cFile);
    return is_file($desiredFile) ? $desiredFile : $srcFile;
}
```

3 本地化服务

本地化服务包括日期、时间和数字的本地格式化，在不同国家和地区按不同的格式显示。Yii 提供了 CDateFormatter 类来格式化日期和时间。

要使用日期、时间格式化功能，可通过调用 CApplication::dateFormatter 属性来实现，如：Yii::app()->dateFormatter。CApplication::getDateFormatter 方法首先根据目标语言申请一个 CLocale 对象，然后再申请一个 CDateFormatter 对象关联到_dateFormatter 属性。

```
public function CApplication::getDateFormatter()
{
    return $this->getLocale()->getDateFormatter();
}
public function CLocale::getDateFormatter()
```

```
{
    if($this->_dateFormatter===null)
        $this->_dateFormatter=new CDateFormatter($this);
    return $this->_dateFormatter;
}
```

CdateFormatter 类提供了 format 和 formatDateTime 两个方法来格式化日期和时间。

format()将时间为一个字符串，如：Yii::app()->dateFormatter->format("yyyy-MM-dd", time())。

formatDateTime()根据目标语言将 UNIX 时间戳转换为给定的日期模式和时间模式。日期和时间模式定义在区域数据文件中，由 CLocale 对象初始化时加载。如：区域 ID 为 zh_cn 的区域数据文件为 YII_PATH/i18n/data/zh_cn.php，日期和时间模式位于此文件中，如下所示：

```
'dateFormats' =>
  array (
    'full' => 'y 年 M 月 d 日 EEEE',
    'long' => 'y 年 M 月 d 日',
    'medium' => 'yyyy-M-d',
    'short' => 'yy-M-d',
  ),
  'timeFormats' =>
  array (
    'full' => 'zzzzah 时 mm 分 ss 秒',
    'long' => 'zah 时 mm 分 ss 秒',
    'medium' => 'ah:mm:ss',
    'short' => 'ah:mm',
  ),
```

如下语句将 unix 时间戳转换为 full 模式：

语句：

```
echo Yii::app()->dateFormatter->formatDateTime(time(), 'full', 'full');
```

结果：

2020 年 6 月 13 日星期六 CST 下午 11 时 43 分 33 秒

数字与日期时间类似，数字在不同的国家和地区也有不同的格式，数字格式化包含十进制格式化，货币格式化和百分比格式化，Yii 提供了 CNumberFormatter 类来实现这些功能。可通过 CApplication::numberFormatter 属性来访问 CNumberFormatter 类，如：Yii::app()->numberFormatter。CApplication::numberFormatter 跟 CApplication::getDateFormatter 方法一样，首先也根据目标语言申请一个 CLocale 对

象，然后再申请一个 CNumberFormatter 对象关联到_numberFormatter 属性。

```
public function CApplication::getNumberFormatter()
{
    return $this->getLocale()->getNumberFormatter();
}
public function CLocale::getNumberFormatter()
{
    if($this->_numberFormatter===null)
        $this->_numberFormatter=new CNumberFormatter($this);
    return $this->_numberFormatter;
}
```

CNumberFormatter 类提供了如下方法来格式化数字：

format($pattern,$value,$currency=null) ：将数字格式化成指定的字符串，如：Yii::app->numberFormatter->format('#,##0.00', $number)。

formatDecimal($value) :根据区域十进制模式格式化数字，十进制模式参考区域数据文件 decimalFormat 字段，如区域 id 为 zh_cn 的十进制模式为：'decimalFormat' => '#,##0.###'。

formatPercentage($value) :根据区域百分比模式格式化数字，百分比模式参考区域数据文件 percentFormat 字段，如区域 id 为 zh_cn 的十进制模式为：'percentFormat' => '#,##0%'。

formatCurrency($value,$currency) ：根据区域货币模式格式化数字，货币模式参考区域数据文件 currencyFormat 字段，如区域 id 为 zh_cn 的十进制模式为：'currencyFormat' => '¤#,##0.00',。

如以下语句格式化十进制数字：

语句：

```
echo Yii::app()->numberFormatter->formatDecimal(123456789)
```

结果：

123,456,789

第五章 控制台应用

1 命令运行流程

跟 web 请求处理流程一样，控制台应用也需要有一个入口脚本，此脚本创建 CConsoleApplication 对方并调用 run 方法运行命令。我们假设入口脚本文件名为 console.php:

batch.php:

```
<?php

// change the following paths if necessary
$yii=dirname(__FILE__).'/../../framework/yii.php';
$config=dirname(__FILE__).'/../config/console.php';

// remove the following lines when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);
// specify how many levels of call stack should be shown in each log message
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);

require_once($yii);
Yii::createConsoleApplication($config)->run();
```

batch.php 跟 index.php 文件非常相似，主要的不同点是创建 CConsoleApplication 对象而不是 CwebApplication 对象，并且 include 的配置文件也不一样。

控制台应用的配置文件格式跟项目 web 请求处理配置文件格式一样，components 等配置项都可以复用，配置项的解析流程完全一样。控制台应用的配置文件新增了 commandMap 配置项，此配置项用于配置命令名到命令文件的映射。console.php 文件内容如下：

```
<?php

// This is the configuration for yiic console application.
// Any writable CConsoleApplication properties can be configured here.
return array(
	'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
	'name'=>'My Console Application',

	// preloading 'log' component
	'preload'=>array('log'),


	'commandMap' => array(
		'email'=>array(
```

```
                        'class'=>'path.to.Mailer',
                        'interval'=>3600,
                ),
                'log'=>'path/to/LoggerCommand.php',
        ),

        // application components
        'components'=>array(
                // database settings are configured in database.php
                'db'=>require(dirname(__FILE__).'/database.php'),

                'log'=>array(
                        'class'=>'CLogRouter',
                        'routes'=>array(
                                array(
                                        'class'=>'CFileLogRoute',
                                        'levels'=>'error, warning',
                                ),
                        ),
                ),

        ),
);
```

从类的自动加载、配置项的解析、到根据 preload 配置项创建组件，这些流程都跟请求的处理流程一样。

创建 CConsoleApplication 对象时运行 CApplication::__construct 方法调用 init()方法时控制台应用跟 web 请求应用开始执行不同的流程，CConsoleApplication::init()方法负责查找所有命令，源代码如下：

CConsoleApplication.php:

```
protected function init()
{
    parent::init();
    if(empty($_SERVER['argv'])) //  检查传递给/weboot/batch.php 的参数是否为空
        die('This script must be run from the command line.');
    //  创建 CConsoleCommandRunner 对象
    $this->_runner=$this->createCommandRunner();
    // commandMap：命令名到命令文件的配置
    $this->_runner->commands=$this->commandMap;
    //  在$basePath/commands 目录下查找所有命令
    $this->_runner->addCommands($this->getCommandPath());
}
```

CConsoleCommandRunner.php:

```
public function addCommands($path)
{
     // 在$path 目录下查找所有命令
    if(($commands=$this->findCommands($path))!==array())
    {
        foreach($commands as $name=>$file)
        {
             // 不允许重名的命令存在
            if(!isset($this->commands[$name]))
                $this->commands[$name]=$file;
        }
    }
}
public function findCommands($path)
{
    if(($dir=@opendir($path))===false)
        return array();
    $commands=array();
    while(($name=readdir($dir))!==false)
    {
        $file=$path.DIRECTORY_SEPARATOR.$name;
        // 文件名以 Command.php 结尾，并且区分大小写
        if(!strcasecmp(substr($name,-11),'Command.php') && is_file($file))
            $commands[strtolower(substr($name,0,-11))]=$file;
    }
    closedir($dir);
    return $commands;
}
```

CConsoleApplication::init()函数运行完后，调用 CApplication::run()方法运行命令，run 方法先处理 onBeginRequest 事件，再调用 CConsoleApplication::processRequest 方法执行命令。

CConsoleApplication.php:

```
public function processRequest()
{
     // 传入参数，执行命令
    $exitCode=$this->_runner->run($_SERVER['argv']);
    if(is_int($exitCode))
        $this->end($exitCode);
}
```

CConsoleCommandRunner.php:

```
public function run($args)
```

```
{
     // 第一个参数是脚本名
    $this->_scriptName=$args[0];
    array_shift($args);
    if(isset($args[0]))
    {
                // 获取第二个参数是命令名
        $name=$args[0];
        array_shift($args);
    }
    else
        $name='help';

    $oldCommand=$this->_command;
    // 根据$name 加载类名，并创建类对象
    if(($command=$this->createCommand($name))===null)
        $command=$this->createCommand('help');
    $this->_command=$command;
    // 调用命令类的 init 方法
    $command->init();
    // 运行命令
    $exitCode=$command->run($args);
    $this->_command=$oldCommand;
    return $exitCode;
}

public function createCommand($name)
{
    $name=strtolower($name);

    $command=null;
    // CConsoleApplication::init()方法中已装载所有命令
    if(isset($this->commands[$name]))
        $command=$this->commands[$name];
    else
    {
         // 将$this->commands 中的所有键名转换成小写，
         // addCommands 函数中存储在 commands 中的键名已经全是小写，但通过 commandMap 配置
的命令可能不是小写的
        $commands=array_change_key_case($this->commands);
        if(isset($commands[$name]))
            $command=$commands[$name];
    }
```

```
    if($command!==null)
    {
        if(is_string($command)) // class file path or alias
        {
            if(strpos($command,'/')!==false || strpos($command,'\\')!==false)
            {
                 // 获取类名
                $className=substr(basename($command),0,-4);
                if(!class_exists($className,false))
                    require_once($command);
            }
            else // an alias，类名跟类文件名必须一致
                $className=Yii::import($command);
            return new $className($name,$this);   // 创建类对象
        }
        else // an array configuration
            return Yii::createComponent($command,$name,$this);
    }
    elseif($name==='help')
        return new CHelpCommand('help',$this);
    else
        return null;
}
```

CConsoleCommandRunner::run()方法先获取命令名，根据命令名创建类对象，然后通过类对象调用 init 方法，最后调用 CConsoleCommand::run 方法。run 方法负责解释命令行选择和参数，在带参数调用 action 之前会调用 beforeAction 方法，并且在 action 方法运行后调用 afterAction 方法。

CConsoleCommand.php:

```
public function run($args)
{
     // 解析选项和参数
    list($action, $options, $args)=$this->resolveRequest($args);
    $methodName='action'.$action;
    if(!preg_match('/^\w+$/',$action) || !method_exists($this,$methodName))
        $this->usageError("Unknown action: ".$action);

    $method=new ReflectionMethod($this,$methodName);
    $params=array();
    // named and unnamed options
         // 遍历 action 方法的参数
```

```
foreach($method->getParameters() as $i=>$param)
{
    $name=$param->getName();
    // 存在同名的选项
    if(isset($options[$name]))
    {
        // 如果参数是数组格式，将值也转换成数组格式
        if($param->isArray())
            $params[]=is_array($options[$name]) ? $options[$name] : array($options[$name]);
        elseif(!is_array($options[$name]))
            $params[]=$options[$name];
        else
            $this->usageError("Option --$name requires a scalar. Array is given.");
    }
    elseif($name==='args') // 参数名等于 args 的参数接收所有不以--开头的参数
        $params[]=$args;
    elseif($param->isDefaultValueAvailable()) // 带默认值的参数
        $params[]=$param->getDefaultValue();
    else
        $this->usageError("Missing required option --$name.");
    unset($options[$name]);
}

// try global options
if(!empty($options)) // 给类属性赋值
{
    $class=new ReflectionClass(get_class($this));
    foreach($options as $name=>$value)
    {
        // 判断类是否有此属性
        if($class->hasProperty($name))
        {
            $property=$class->getProperty($name);
            // 只能赋值公共非静态属性
            if($property->isPublic() && !$property->isStatic())
            {
                $this->$name=$value;
                unset($options[$name]);
            }
        }
    }
}
```

```
    if(!empty($options))
        $this->usageError("Unknown options: ".implode(', ',array_keys($options)));

    $exitCode=0;
    // 调用类的 beforeAction 方法
    if($this->beforeAction($action,$params))
    {
         // 执行 action 方法
        $exitCode=$method->invokeArgs($this,$params);
        // 调用类的 afterAction 方法, afterAction 方法可改变 exitCode
        $exitCode=$this->afterAction($action,$params,is_int($exitCode)?$exitCode:0);
    }
    return $exitCode;
}

protected function resolveRequest($args)
{
    $options=array();    // named parameters
    $params=array();     // unnamed parameters
    foreach($args as $arg)
    {
        if(preg_match('/^--(\w+)(=(.*))?$/',$arg,$matches))    // an option
        {
             // 获取每个参数的名称和值
            $name=$matches[1];
            $value=isset($matches[3]) ? $matches[3] : true;
            if(isset($options[$name])) // 存在重复的参数
            {
                 // 重复的参数，用数组来保存值
                if(!is_array($options[$name]))
                    $options[$name]=array($options[$name]);
                $options[$name][]=$value;
            }
            else
                $options[$name]=$value;
        }
        elseif(isset($action))
            $params[]=$arg;
        else
            $action=$arg;
    }
    if(!isset($action))
        $action=$this->defaultAction;
```

```
    return array($action,$options,$params);
}
```

执行完 afterAction 方法后，会触发 onEndRequest 事件。

2 自定义命令

命令文件默认放在$basePath/commands 目录下，并且文件名以 Command.php(区分大小写)结尾，命令类继承 CConsoleCommand 基类，基类提供了 init、beforeAction、afterAction 等方法。

以下代码定义了 Example 命令

ExampleCommand.php:

```
class ExampleCommand extends CConsoleCommand
{
        public function actionTest()
        {
                echo "console command test";
                return 0; // 去掉此行语句，默认返回 null
        }
}
```

运行如下命令来执行，Example 表示命令类名，test 是 action 方法名：

```
/usr/bin/php  /home/www/testyii/webroot/console.php  Example  test
```

结果：

```
console command test
```

创建带选项的命令

ExampleCommand.php:

```
class ExampleCommand extends CConsoleCommand
{
        public function actionTest2($a, $b, $c=1)
        {
                $msg = '';
                $msg .= !isset($a) ? '' : (is_array($a) ? 'a=' . json_encode($a) : 'a=' . $a);
                $msg .= ' ';
                $msg .= !isset($b) ? '' : (is_array($b) ? 'b=' . json_encode($b) : 'b=' . $b);
                $msg .= ' ';
```

```
                $msg .= !isset($c) ? '' : (is_array($c) ? 'c=' . json_encode($c) : 'c=' . $c);

                echo $msg . PHP_EOL;
                return 0; // 去掉此行语句，默认返回 null
        }
}
```

命令 1：

```
/usr/bin/php  /home/www/testyii/webroot/console.php  Example  test2 --a=1 --b=2 --c=3
```

结果：

```
a=1 b=2 c=3
```

命令 2：

```
/usr/bin/php  /home/www/testyii/webroot/console.php  Example  test2 --a=1 --b=2
```

结果：

```
a=1 b=2 c=1
```

如果参数有默认值，命令执行时可以不带选项，但如果没有默认值的参数，执行命令时，未带此对应选项会报错。action 方法的参数只能通过带双横杠'--'的选项来传递，不带双横杠或带单横杠都不行。在执行命令时，传递了 action 方法不存在的参数选项也会报错，如下命令执行时会报错，会提示”未知的选项 d”：

```
/usr/bin/php  /home/www/testyii/webroot/console.php  Example  test2 --a=1 --b=2 --d=4
```

当 action 方法的参数是数组类型时，可以通过包含多个同名的选项来传参，我们将 ExampleCommand::actionTest2 方法定义改下如下：

```
public function actionTest2(array $a, $b, $c=1)
```

现在参数 a 是数组类型了，执行如下命令：

命令 3:

```
/usr/bin/php  /home/www/testyii/webroot/console.php  Example  test2 --a=1 --a=2 --a=3 --b=2
```

结果：

```
a=["1","2","3"] b=2 c=1
```

给命令类公共属性赋值

运行命令时我们还可以给命令类属性赋值，跟 action 方法赋值一样，类属性赋值也是通过双横杠'--'来实

现，这里存在优先级问题，如果存在同名的属性和 action 方法参数名，action 方法参数名优先级高于类属性。

ExampleCommand.php:

```
class ExampleCommand extends CConsoleCommand
{
	public $prop1;
	public $prop2;
	public $a=1;

	public function actionTest3($a)
	{
		echo 'local a=' . $a . PHP_EOL;
		echo 'prop1=' . $this->prop1 . ' prop2=' . $this->prop2 . ' a=' . $this->a . PHP_EOL;
		return 0;
	}
}
```

以上类包含三个属性，其中属性$a 与 actionTest3 方法的参数同名，执行如下命令：

命令 1：

```
/usr/bin/php /home/www/testyii/webroot/console.php Example test3 --prop1=1 --prop2=2 --a='abc'
```

结果：

```
local a=abc
prop1=1 prop2=2 a=1
```

通过 commandMap 配置项映射命令

在 config/console.php 文件中添加以下配置：

```
	'commandMap' => [
		'ex' => 'application.commands.ExampleCommand',
	],
```

commandMap 配置项配置命令名到命令文件的映射，以下两条命令相等：

```
/usr/bin/php /home/www/testyii/webroot/console.php Example test

/usr/bin/php /home/www/testyii/webroot/console.php Ex test
```

3 小结

控制台命令一般配合 crontab 命令来使用，可实现定时任务；也可以配合 supervisor 实现长驻进程，控制

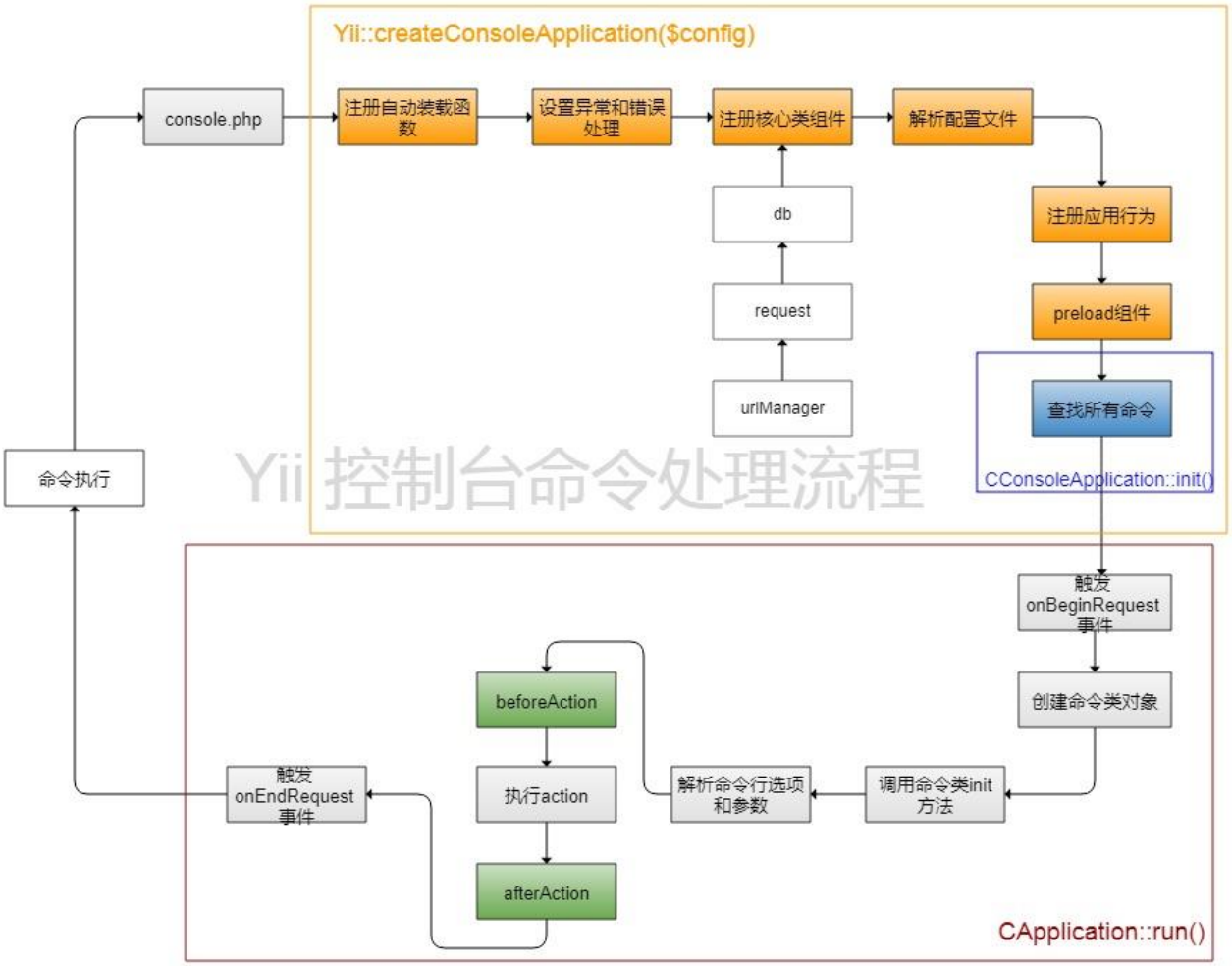
台命令不同于 web 请求，它可以长时间运行，并占用大量内存，因此最好在入口文件 console.php 中使用 set_time_limit(0)和 ini_set('memory_limit', -1)语句放开执行时间和占用内存的限制。

当使用 Yii::log 来记录日志时，对于长时间运行的控制台命令来说，Yii 默认的 log 配置并不适合，原因是命令结束后触发 onEndRequest 事件时才会将日志路由到目地的，这样会导致内存中存储大量日志，通过修改 CLogger 的 autoFlush 和 autoDump 属性实时将日志路由到目地的(写入文件或其它)。

将入口文件修改如下：

```
<?php

// change the following paths if necessary
$yii=dirname(__FILE__).'/../../framework/yii.php';
$config=dirname(__FILE__).'/../config/console.php';

// remove the following lines when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);
// specify how many levels of call stack should be shown in each log message
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);

require_once($yii);
$app= Yii::createConsoleApplication($config);

// 每记录一条日志，就将日志发送给路由类(如：CFileLogRouter)处理
Yii::getLogger ()->autoFlush =1;
// 实时将日志持久化(写入文件或发送 email 等)
Yii::getLogger()->autoDump = true;

$app->run();
```

控制台命令处理流程图：


Yii::createConsoleApplication($config)
console.php
注册自动装载函数
设置异常和错误处理
注册核心类组件
解析配置文件
db
request
urlManager
注册应用行为
preload组件
查找所有命令
CConsoleApplication::init()
命令执行
Yii 控制台命令处理流程
触发onBeginRequest事件
创建命令类对象
调用命令类init方法
解析命令行选项和参数
beforeAction
执行action
afterAction
触发onEndRequest事件
CApplication::run()

第六章 视图

1 视图的使用

视图是用来存放界面元素的文件，Yii 提供了 CController::render 方法来渲染视图，render 方法原型如下：

```
public function render($view,$data=null,$return=false)
```

$view：指视图名，视图名需与视图文件名相同，框架会根据视图名来查找同名的视图文件，视图文件的默认扩展名为.php，例如：$view 参数值等于 index，Yii 会在别名为 application.views.ControllerID 目录下查找 index.php 的视图文件。

$data：传递给视图文件的参数，格式为数组，如：['var1'=>$value1, 'var2'=>$value2]。除通过$data 传递参数外，视图文件中可以直接访问当前控制器的所有属性，如：$this->prop1，访问控制器的 prop1 属性。

$return：false:视图文件内容直接输出，true:视图文件内容以字符串形式返回，默认为 false。

以下语句渲染 index 视图，并传递了 varl1,varl2 两个变量。

```
$this->render('index', array(
    'var1'=>$value1,
    'var2'=>$value2,
));
```

除 render()方法外，Yii 还提供了 renderFile()方法来渲染视图，与 render()方法的区别是 renderFile()不渲染布局，跳过主题的查找，直接渲染视图并输出，并且视图文件要提供绝对路径。如下语句渲染 index 视图:

```
$this->renderFile(Yii::app()->basePath . '/views/site/index.php', array(
    'var1'=>$value1,
    'var2'=>$value2,
));
```

布局

布局通常用来将多个视图中的公共部分剥离出来，成形一个特殊的视图文件。当调用 render 方法渲染视图时如果不指定布局会默认使用 CWebApplication::layout 属性指定的布局文件(views/layouts/main.php 文件)，可通过修改 CWebApplication::layout 或 CController::layout 属性来改变默认值。如果要渲染不带布局的视图，可使用 renderPartial 方法。renderPartial 方法原形如下：

```
public function renderPartial($view,$data=null,$return=false,$processOutput=false)
```

前三个参数跟 rendier 方法参数一样，$processOutput 表示是否将 js、css 文件插入到视图中输出，默认为 false。

主题

主题是用来改变 web 站点外观的一种方式，每个主题代表一个目录，目录下包含 views、layouts 等资源，主题的名字就是目录名(区分大小写)，主题默认存放在 webroot/themes 目录下(可通过 themeManager 组件的 basePath 属性修改存放目录)。应用主题后，如果在主题目录下没有找到视图文件，会返回到 $basePath/views 目录下查找。

下面的例子中包含了 basic 和 fancy 两个主题：

```
webroot/
    assets
    protected/
        components/
        controllers/
        models/
        views/
            layouts/
                main.php
            site/
                index.php
    themes/
        basic/
            views/
                layouts/
                    main.php
                site/
                    index.php
        fancy/
            views/
                layouts/
                    main.php
                site/
                    index.php
```

当我们在配置文件中使用以下配置项配置主题：

```
return array(
    'theme'=>'basic',
    ......
);
```

项目当前使用 basic 主题，会到 webroot/themes/basic/views 下找视图文件，如果没有找到就在 webroot/views/目录下找视图文件。如果项目源语言和目的语言不一致，例子中的目录结构就变为：

```
webroot/
    assets
    protected/
        components/
        controllers/
        models/
        views/
            layouts/
                main.php
            site/
                index.php
    themes/
        basic/
            views/
                layouts/
                    main.php
                site/
                    index.php
        fancy/
            views/
                zh_cn/
                    layouts/
                        main.php
                    site/
                        index.php
                layouts/
                    main.php
                site/
                    index.php
```

如上例子，当采用 fancy 主题时，会先到 webroot/themes/fancy/views/zh_cn 目录下找文件，没有找到再到 webroot/themes/fancy/views 目录下找文件。

2 渲染流程

render()在渲染视图前会调用 CController::beforeRender()方法，控制器可以覆盖此方法在渲染前统一做预处理操作，并在视图输出前调用 CController::afterRender()方法。

CController.php:

```
public function render($view,$data=null,$return=false)
{
```

```
    if($this->beforeRender($view))
    {
        // 渲染视图文件
        $output=$this->renderPartial($view,$data,true);
        // 查找布局文件
        if(($layoutFile=$this->getLayoutFile($this->layout))!==false)
             // 渲染布局文件
            $output=$this->renderFile($layoutFile,array('content'=>$output),true);

        $this->afterRender($view,$output);

        // 将注册的 js、css 等脚本插入到合适的位置
        $output=$this->processOutput($output);

        if($return)
            return $output;
        else
            echo $output;
    }
}

public function renderPartial($view,$data=null,$return=false,$processOutput=false)
{
    // 获取视图文件绝对路径
    if(($viewFile=$this->getViewFile($view))!==false)
    {
         // $output 包含视图文件内容
        $output=$this->renderFile($viewFile,$data,true);
        // 将注册的 js、css 等脚本插入到合适的位置
        if($processOutput)
            $output=$this->processOutput($output);
        if($return)
            return $output;
        else
            echo $output;
    }
    else
        throw new CException(Yii::t('yii','{controller} cannot find the requested view "{view}".',
            array('{controller}'=>get_class($this), '{view}'=>$view)));
}

public function getViewFile($viewName)
{
```

```
    // 如果设置了 theme，这里先调用 CThemeManager::getTheme 函数，然后在主题目录下找文件
    // 主题目录默认是 webroot/themes，可通过 themeManager 组件的 basePath 属性修改默认值
    if(($theme=Yii::app()->getTheme())!==null
      && ($viewFile=$theme->getViewFile($this,$viewName))!==false)
        return $viewFile;
    // 如果没有配置主题，如主题中没有找到文件，就在$basePath/views 目录下找文件
    $moduleViewPath=$basePath=Yii::app()->getViewPath();
    if(($module=$this->getModule())!==null)
        $moduleViewPath=$module->getViewPath();
    // 第二个参数等于$basePath/$controllerId
    return $this->resolveViewFile($viewName,$this->getViewPath(),$basePath,$moduleViewPath);
}

public function resolveViewFile($viewName,$viewPath,$basePath,$moduleViewPath=null)
{
    if(empty($viewName))
        return false;

    if($moduleViewPath===null)
        $moduleViewPath=$basePath;

    // Yii 框架默认没有注册 viewRenderer 组件
    if(($renderer=Yii::app()->getViewRenderer())!==null)
        $extension=$renderer->fileExtension;
    else
        $extension='.php'; // 默认视图和布局文件的扩展名是.php
    if($viewName[0]==='/')
    {
        if(strncmp($viewName,'//',2)===0)
            $viewFile=$basePath.$viewName;
        else
            $viewFile=$moduleViewPath.$viewName;
    }
    elseif(strpos($viewName,'.'))
        $viewFile=Yii::getPathOfAlias($viewName);
    else
        $viewFile=$viewPath.DIRECTORY_SEPARATOR.$viewName;

    // 根据源语言和目的语言查找文件
    if(is_file($viewFile.$extension))
        return Yii::app()->findLocalizedFile($viewFile.$extension);
    elseif($extension!=='.php' && is_file($viewFile.'.php'))
        return Yii::app()->findLocalizedFile($viewFile.'.php');
```

```
    else
        return false;
}
```

获取到视图文件后调用 renderFile 将视图文件做为字符串返回。

CBaseController.php:

```
public function renderFile($viewFile,$data=null,$return=false)
{
    $widgetCount=count($this->_widgetStack);
    // 框架默认没有注册 viewRenderer 组件，如果已注册 viewRenderer 组件，这里还需判断组件关注的扩展名是否跟$viewFile 扩展名一致
    if(($renderer=Yii::app()->getViewRenderer())!==null
       && $renderer->fileExtension==='.'.CFileHelper::getExtension($viewFile))
        $content=$renderer->renderFile($this,$viewFile,$data,$return);
    else
        // 将文件内容做为字符串返回
        $content=$this->renderInternal($viewFile,$data,$return);
    if(count($this->_widgetStack)===$widgetCount)
        return $content;
    else
    {
        $widget=end($this->_widgetStack);
        throw new CException(Yii::t('yii','{controller} contains improperly nested widget tags in its view "{view}". A {widget} widget does not have an endWidget() call.', array('{controller}'=>get_class($this), '{view}'=>$viewFile, '{widget}'=>get_class($widget))));
    }
}

public function renderInternal($_viewFile_,$_data_=null,$_return_=false)
{
    // we use special variable names here to avoid conflict when extracting data
    if(is_array($_data_))
        // 从数组中将变量导入到当前符号表中
        extract($_data_,EXTR_PREFIX_SAME,'data');
    else
        $data=$_data_;
    if($_return_)
    {
         // 将$viewFile 文件内容做为字符串返回，返回前先解释 php 变量
        ob_start();
        ob_implicit_flush(false);
        require($_viewFile_);
        return ob_get_clean();
```

```
    }
    else
        require($_viewFile_);
}
```

3 集成 smarty 模板引擎

smarty 是用 php 编写的模板引擎，它拥有较好的性能以及丰富的函数库，自带缓存技术。目前 smarty 最新版本是 smarty-3.1.34，它能很好的跟 Yii 框架结合。

下面我们以 smarty-3.1.34 为例介绍 Yii 如何集成 smarty 模板引擎。

第一步：

下载 smarty-3.1.34 并解压，将 libs 目录复制到项目的 extensions 目录下(application.extensions)，并重命为 smarty。

第二步：

Yii 框架在渲染过程中会判断是否有注册 viewRenderer 组件，如果有注册就会调用该组件的 renderFile 方法来渲染视图，我们将 smarty 封装成 viewRenderer 组件来访问。

在 extensions/smarty 目录下新建 CSmartyViewRenderer.php 文件，内容如下：

```
<?php

defined('SMARTY_VIEW_DIR') or define('SMARTY_VIEW_DIR', Yii::getPathOfAlias('application.views'));
require('Smarty.class.php');

class CSmartyViewRenderer extends CApplicationComponent implements IViewRenderer
{
    public $fileExtension='.tpl'; //  默认的文件扩展名，CBaseController::renderFile 会用到此属性
    public $_smarty;

    function init()
    {
        $this->_smarty = new Smarty();

        $this->_smarty->template_dir = SMARTY_VIEW_DIR;
        $this->_smarty->compile_dir = SMARTY_VIEW_DIR.DIRECTORY_SEPARATOR.'tpl_c';
        $this->_smarty->cache_dir = SMARTY_VIEW_DIR.DIRECTORY_SEPARATOR.'cache';
        $this->_smarty->config_dir = SMARTY_VIEW_DIR.DIRECTORY_SEPARATOR.'config';
```

```
        $this->_smarty->caching = false;
        //$this->_smarty->cache_lifetime = 3600;
    }

    public function renderFile($context,$sourceFile,$data,$return = false)
    {
        // render()方法调用 renderFile 方法时会传递绝对路径文件，控制器调用 renderFile 时传
        //递的是相对路径，这里加上模板的根目录
        if (strpos($sourceFile, '/') !== 0) {
            $sourceFile = $this->_smarty->getTemplateDir(0) . $sourceFile;
        }

        $template = $this->_smarty->createTemplate($sourceFile, null, null, $data, false);
        if ($return)
            return $template->fetch($template);
        else
            $template->display($template);
    }
}
```

第三步：

在 config/main.php 项目配置文件中增加 viewRenderer 组件配置，如下：

```
'components'=>array(
    'viewRenderer' => array(
        'class' => 'ext.smarty.CSmartyViewRenderer',
        'fileExtension' => '.tpl',
    ),
    ......
)
```

第四步：

在控制器中渲染视图

```
    $this->render('index', array(
        'var1'=>$value1,
        'var2'=>$value2,
    ));
```

或

```
    $this->renderFile('site/index.tpl', array(
        'var1'=>$value1,
```

```
        'var2'=>$value2,
));
```

附 录

Yii 框架的优点：
1、高性能。
2、严格按照 OOP 方式编写。
3、提供了事件响应式机制。
4、提供了主题功能，可以很方便的在多种主题间切换。
5、易扩展、框架是基于组件的，在配置文件中可通过配置项来定制组件的行为。