

安装 Yii

你可以通过两种方式安装 Yii：使用 [Composer](#) 或下载一个归档文件。推荐使用前者，这样只需执行一条简单的命令就可以安装新的[扩展](#)或更新 Yii 了。

注意：和 Yii 1 不同，以标准方式安装 Yii 2 时会同时下载并安装框架本身和一个应用程序的基本骨架。

通过 Composer 安装

如果还没有安装 Composer，你可以按 getcomposer.org 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

```
curl -s http://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

在 Windows 中，你需要下载并运行 [Composer-Setup.exe](#)。

如果遇到任何问题或者想更深入地学习 Composer，请参考 [Composer 文档 \(英文\)](#)，[Composer 中文](#)。

如果你已经安装有 Composer 请确保使用的是最新版本，你可以用 `composer self-update` 命令更新 Composer 为最新版本。

Composer 安装后，切换到一个可通过 Web 访问的目录，执行如下命令即可安装 Yii：

```
composer global require "fxp/composer-asset-plugin:1.0.0-beta4"
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

第一条命令安装 [Composer asset plugin](#)，它是通过 Composer 管理 bower 和 npm 包所必须的，此命令全局生效，一劳永逸。第二条命令会将 Yii 安装在名为 `basic` 的目录中，你也可以随便选择其他名称。

注意：在安装过程中 Composer 可能会询问你 GitHub 账户的登录信息，因为可能在使用中超过了 GitHub API（对匿名用户的）使用限制。因为 Composer 需要为所有扩展包从 GitHub 中获取大量信息，所以超限非常正常。（译注：也意味着作为程序猿没有 GitHub 账号，就真不能愉快地玩耍了）登陆 GitHub 之后可以得到更高的 API 限额，这样 Composer 才能正常运行。更多细节请参考 [Composer 文档](#)（该段 [Composer 中文文档](#) [期待您的参与](#)）。

技巧：如果你想安装 Yii 的最新开发版本，可以使用以下命令代替，它添加了一个 `stability` 选项（[中文版](#)）：

```
composer create-project --prefer-dist --stability=dev yiisoft/yii2-app-basic basic
```

注意，Yii 的开发版(dev 版)不应该用于生产环境中，它可能会破坏运行中的代码。

通过归档文件安装

通过归档文件安装 Yii 包括三个步骤：

- 1.从 yiiframework.com 下载归档文件。
- 2.将下载的文件解压缩到 Web 目录中。
- 3.修改 `config/web.php` 文件，给 `cookieValidationKey` 配置项添加一个密钥（若你通过 Composer 安装，则此步骤会自动完成）：

```
// !!! 在下面插入一段密钥（若为空） - 以供 cookie validation 的需要
```

```
'cookieValidationKey' => '在此处输入你的密钥',
```

其他安装方式

上文介绍了两种安装 Yii 的方法，安装的同时也会创建一个立即可用的 Web 应用程序。对于小的项目或用于学习上手，这都是一个不错的起点。

但是其他的安装方式也存在：

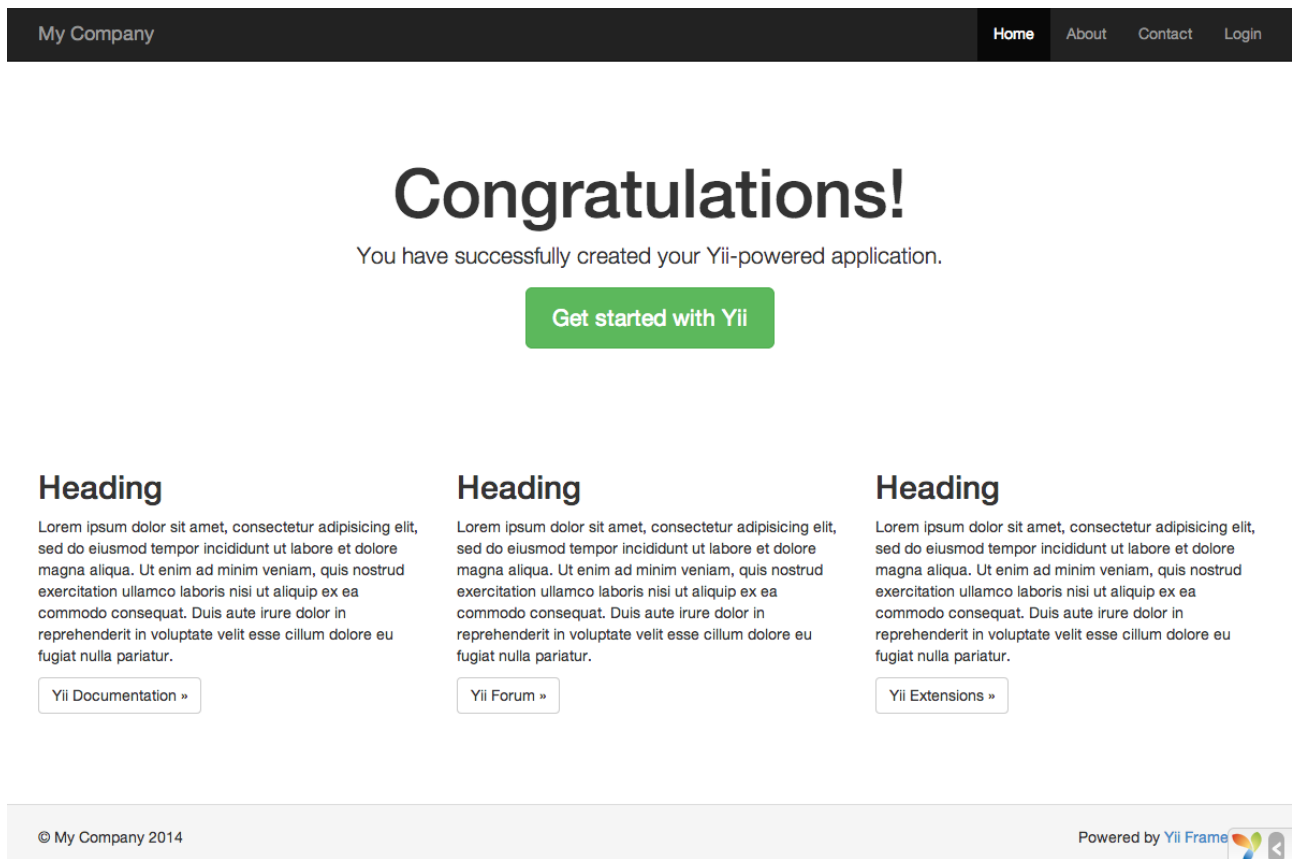
- 如果你只想安装核心框架，然后从零开始构建整个属于你自己的应用程序模版，可以参考[从头构建自定义模版](#)一节的介绍。
- 如果你要开发一个更复杂的应用，可以更好地适用于团队开发环境的，可以考虑安装[高级应用模版](#)。

验证安装的结果

安装完成后，就可以使用浏览器通过如下 URL 访问刚安装完的 Yii 应用了：

```
http://localhost/basic/web/index.php
```

这个 URL 假设你将 Yii 安装到了一个位于 Web 文档根目录下的 `basic` 目录中，且该 Web 服务器正运行在你自己的电脑上（`localhost`）。你可能需要将其调整为适应自己的安装环境。



你应该可以在浏览器中看到如上所示的“Congratulations!”页面。如果没有，请通过以下任意一种方式，检查当前 PHP 环境是否满足 Yii 最基本需求：

- 通过浏览器访问 URL <http://localhost/basic/requirements.php>
- 执行如下命令：

```
cd basic  
  
php requirements.php
```

你需要配置好 PHP 安装环境，使其符合 Yii 的最小需求。主要是需要 PHP 5.4 以上版本。如果应用需要用到数据库，那还要安装 [PDO PHP 扩展](#) 和相应的数据库驱动（例如访问 MySQL 数据库所需的 [pdo_mysql](#)）。

配置 Web 服务器

>补充：如果你现在只是要试用 Yii 而不是将其部署到生产环境中，本小节可以跳过。

通过上述方法安装的应用程序在 Windows，Mac OS X，Linux 中的 [Apache HTTP 服务器](#)或 [Nginx HTTP 服务器](#)且 PHP 版本为 5.4 或更高都可以直接运行。Yii 2.0 也兼容 Facebook 公司的 HHVM，由于 HHVM 和标准 PHP 在边界案例上有些地方略有不同，在使用 HHVM 时需稍作处理。

在生产环境的服务器上，你可能会想配置服务器让应用程序可以通过 URL <http://www.example.com/index.php> 访问而不是 <http://www.example.com/basic/web/index.php>。这种配置需要将 Web 服务器的文档根目录指向 [basic/web](#) 目录。可能你还会想隐藏掉 URL 中的 [index.php](#)，具体细节在 [URL 解析和生成](#)一章中有介绍，你将学到如何配置 Apache 或 Nginx 服务器实现这些目标。

>补充：将 **basic/web** 设置为文档根目录，可以防止终端用户访问 **basic/web** 相邻目录中的私有应用代码和敏感数据文件。禁止对其他目录的访问是一个不错的安全改进。

>补充：如果你的应用程序将来要运行在共享虚拟主机环境中，没有修改其 **Web** 服务器配置的权限，你依然可以通过调整应用的结构来提升安全性。详情请参考[共享主机环境](#) 一章。

推荐使用的 **Apache** 配置

在 **Apache** 的 **httpd.conf** 文件或在一个虚拟主机配置文件中使⤵如下配置。注意，你应该将 **path/to/basic/web** 替换为实际的 **basic/web** 目录。

```
# 设置文档根目录为 “basic/web”

DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">

    # 开启 mod_rewrite 用于美化 URL 功能的支持（译注：对应 pretty URL 选项）

    RewriteEngine on

    # 如果请求的是真实存在的文件或目录，直接访问

    RewriteCond %{REQUEST_FILENAME} !-f

    RewriteCond %{REQUEST_FILENAME} !-d

    # 如果请求的不是真实文件或目录，分发请求至 index.php

    RewriteRule . index.php

    # ...其它设置...

</Directory>
```

推荐使用的 **Nginx** 配置

为了使用 **Nginx**，你应该已经将 **PHP** 安装为 **FPM SAPI** 了。使用如下 **Nginx** 配置，将 **path/to/basic/web** 替换为实际的 **basic/web** 目录，**mysite.local** 替换为实际的主机名以提供服务。

```
server {

    charset utf-8;

    client_max_body_size 128M;

    listen 80; ## 监听 ipv4 上的 80 端口
```

```
#listen [::]:80 default_server ipv6only=on; ## 监听 ipv6 上的 80 端口
```

```
server_name mysite.local;
```

```
root    /path/to/basic/web;
```

```
index    index.php;
```

```
access_log /path/to/basic/log/access.log main;
```

```
error_log /path/to/basic/log/error.log;
```

```
location / {
```

```
    # 如果找不到真实存在的文件，把请求分发至 index.php
```

```
    try_files $uri $uri /index.php?$args;
```

```
}
```

```
# 若取消下面这段的注释，可避免 Yii 接管不存在文件的处理过程（404）
```

```
#location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
```

```
#    try_files $uri =404;
```

```
#}
```

```
#error_page 404 /404.html;
```

```
location ~ \.php$ {
```

```
    include fastcgi.conf;
```

```
    fastcgi_pass 127.0.0.1:9000;
```

```
    #fastcgi_pass unix:/var/run/php5-fpm.sock;
```

```
    try_files $uri =404;
```

```
}
```

```
location ~ /\. (ht|svn|git) {  
  
    deny all;  
  
}  
  
}
```

使用该配置时，你还应该在 `php.ini` 文件中设置 `cgi.fix_pathinfo=0`，能避免掉很多不必要的 `stat()` 系统调用。

还要注意当运行一个 HTTPS 服务器时，需要添加 `fastcgi_param HTTPS on;` 一行，这样 Yii 才能正确地判断连接是否安全。

运行应用

安装 Yii 后，就有了一个可运行的 Yii 应用，根据配置的不同，可以通过 `http://hostname/basic/web/index.php` 或 `http://hostname/index.php` 访问。本章节将介绍应用的内建功能，如何组织代码，以及一般情况下应用如何处理请求。

补充：为简单起见，在整个“入门”板块都假定你已经把 `basic/web` 设为 Web 服务器根目录并配置完毕，你访问应用的地址会是 `http://lostdname/index.php` 或类似的。请按需调整 URL。

功能

一个安装完的基本应用包含四页：

- 主页，当你访问 `http://hostname/index.php` 时显示，
- “About” 页，
- “Contact” 页，显示一个联系表单，允许终端用户通过 Email 联系你，
- “Login” 页，显示一个登录表单，用来验证终端用户。试着用 “admin/admin” 登录，你可以看到当前是登录状态，已经可以“退出登录”了。

这些页面使用同一个头部和尾部。头部包含了一个可以在不同页面间切换的导航栏。

在浏览器底部可以看到一个工具栏。这是 Yii 提供的很有用的调试工具，可以记录并显示大量的调试信息，例如日志信息，响应状态，数据库查询等等。

应用结构

应用中最重要目录和文件（假设应用根目录是 `basic`）：

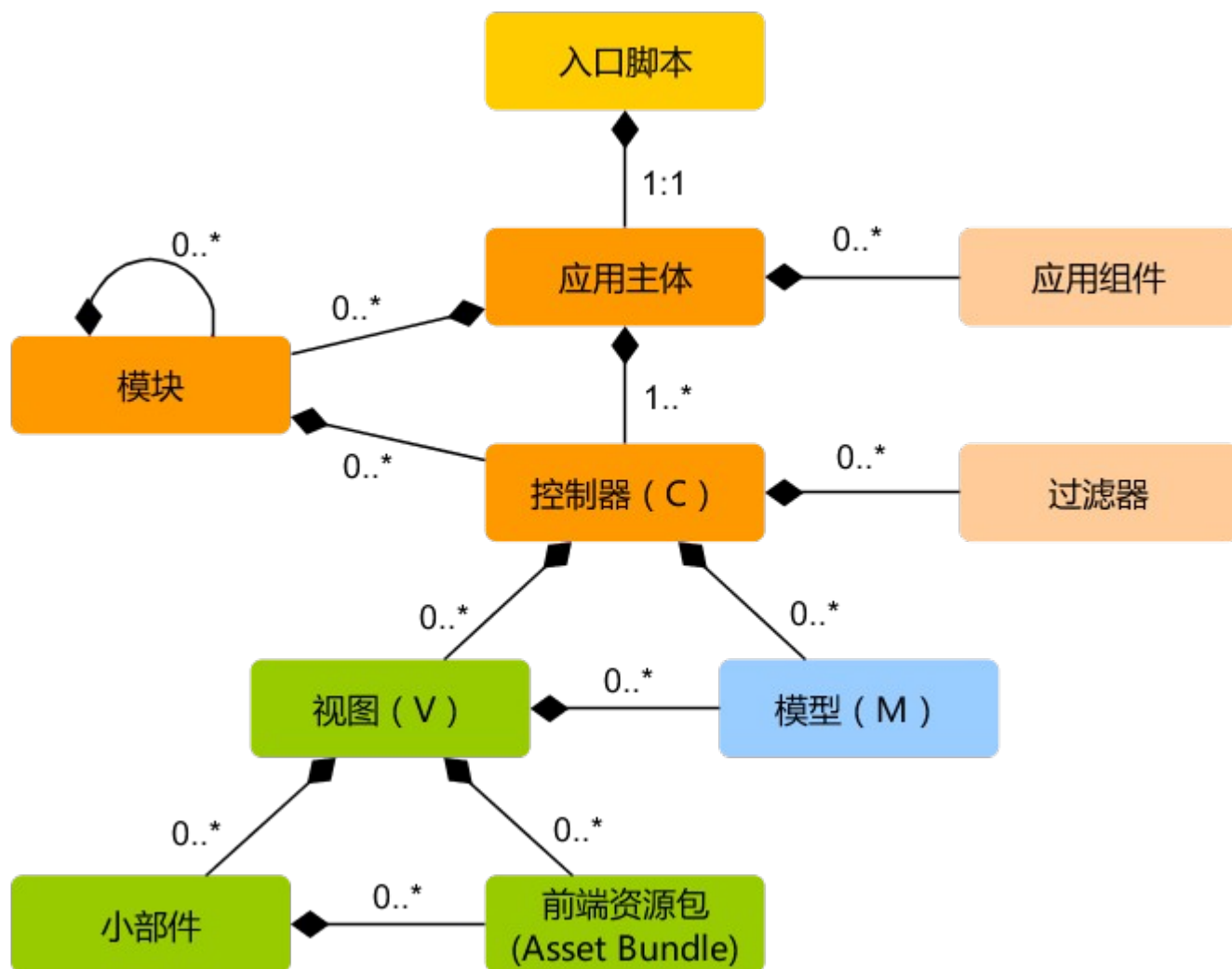
<code>basic/</code>	应用根目录
<code>composer.json</code>	Composer 配置文件, 描述包信息
<code>config/</code>	包含应用配置及其它配置

console.php	控制台应用配置信息
web.php	Web 应用配置信息
commands/	包含控制台命令类
controllers/	包含控制器类
models/	包含模型类
runtime/	包含 Yii 在运行时生成的文件，例如日志和缓存文件
vendor/	包含已经安装的 Composer 包，包括 Yii 框架自身
views/	包含视图文件
web/	Web 应用根目录，包含 Web 入口文件
assets/	包含 Yii 发布的资源文件（javascript 和 css）
index.php	应用入口文件
yii	Yii 控制台命令执行脚本

一般来说，应用中的文件可被分为两类：在 **basic/web** 下的和在其它目录下的。前者可以直接通过 HTTP 访问（例如浏览器），后者不能也不应该被直接访问。

Yii 实现了**模型-视图-控制器 (MVC)**设计模式，这点在上述目录结构中也得以体现。**models** 目录包含了所有**模型类**，**views** 目录包含了所有**视图脚本**，**controllers** 目录包含了所有**控制器类**。

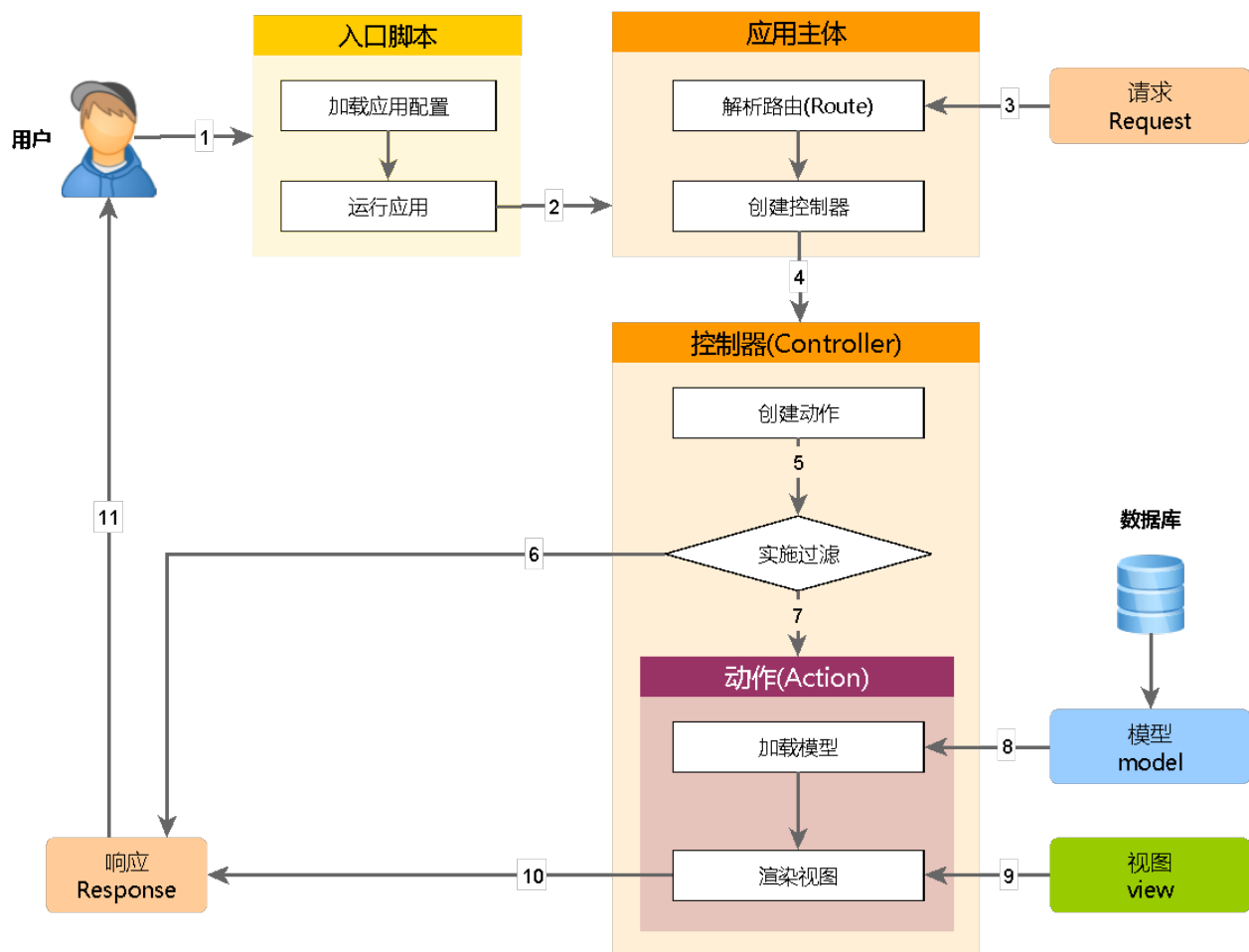
以下图表展示了一个应用的静态结构：



每个应用都有一个入口脚本 `web/index.php`，这是整个应用中唯一可以访问的 **PHP** 脚本。入口脚本接受一个 **Web** 请求并创建**应用**实例去处理它。**应用**在它的**组建**辅助下解析请求，并分派请求至 **MVC** 元素。**视图**使用**小部件**去创建复杂和动态的用户界面。

请求生命周期

以下图表展示了一个应用如何处理请求:



- 1.用户向**入口脚本** `web/index.php` 发起请求。
- 2.入口脚本加载应用**配置**并创建一个**应用**实例去处理请求。
- 3.应用通过**请求**组件解析请求的**路由**。
- 4.应用创建一个**控制器**实例去处理请求。
- 5.控制器创建一个**操作**实例并针对操作执行过滤器。
- 6.如果任何一个过滤器返回失败，则操作退出。
- 7.如果所有过滤器都通过，操作将被执行。
- 8.操作会加载一个数据模型，或许是来自数据库。
- 9.操作会渲染一个视图，把数据模型提供给它。
- 10.渲染结果返回给**响应**组件。
- 11.响应组件发送渲染结果给用户浏览器。

第一次问候：

说声 **Hello**

本章描述了如何在你的应用中创建一个新的 “Hello” 页面。为了实现这一目标，将会创建一个[操作](#)和一个[视图](#)：

- 应用将会分派页面请求给操作
- 操作将会依次渲染视图呈现 “Hello” 给最终用户

贯穿整个章节，你将会掌握三件事：

- 1.如何创建一个[操作](#)去响应请求，
- 2.如何创建一个[视图](#)去构造响应内容，
- 3.以及一个应用如何分派请求给[操作](#)。

创建操作

为了 “Hello”，需要创建一个 [say 操作](#)，从请求中接收 [message](#) 参数并显示给最终用户。如果请求没有提供 [message](#) 参数，操作将显示默认参数 “Hello”。

补充：[操作](#)是最终用户可以直接访问并执行的对象。操作被组织在[控制器](#)中。

一个操作的执行结果就是最终用户收到的响应内容。

操作必须声明在[控制器](#)中。为了简单起见，你可以直接在 [SiteController](#) 控制器里声明 [say](#) 操作。这个控制器是由文件 [controllers/SiteController.php](#) 定义的。以下是一个操作的声明：

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...其它代码...

    public function actionSay($message = 'Hello')
    {
        return $this->render('say', ['message' => $message]);
    }
}
```

```
}  
  
}
```

在上述 `SiteController` 代码中，`say` 操作被定义为 `actionSay` 方法。Yii 使用 `action` 前缀区分普通方法和操作。`action` 前缀后面的名称被映射为操作的 ID。

涉及到给操作命名时，你应该理解 Yii 如何处理操作 ID。操作 ID 总是被以小写处理，如果一个操作 ID 由多个单词组成，单词之间将由连字符连接（如 `create-comment`）。操作 ID 映射为方法名时移除了连字符，将每个单词首字母大写，并加上 `action` 前缀。例子：操作 ID `create-comment` 相当于方法名 `actionCreateComment`。

上述代码中的操作方法接受一个参数 `$message`，它的默认值是 `“Hello”`（就像你设置 PHP 中其它函数或方法的默认值一样）。当应用接收到请求并确定由 `say` 操作来响应请求时，应用将从请求的参数中寻找对应值传入进来。换句话说，如果请求包含一个 `message` 参数，它的值是 `“Goodybye”`，操作方法中的 `$message` 变量也将被填充为 `“Goodybye”`。

在操作方法中，`[[yii\web\Controller::render()|render()]]` 被用来渲染一个名为 `say` 的视图文件。`message` 参数也被传入视图，这样就可以在里面使用。操作方法会返回渲染结果。结果会被应用接收并显示给最终用户的浏览器（作为整页 HTML 的一部分）。

创建视图

视图是你用来生成响应内容的脚本。为了说 `“Hello”`，你需要创建一个 `say` 视图，以便显示从操作方法中传来的 `message` 参数。

```
<?php  
  
use yii\helpers\Html;  
  
?>  
  
<?= Html::encode($message) ?>
```

`say` 视图应该存为 `views/site/say.php` 文件。当一个操作中调用了 `[[yii\web\Controller::render()|render()]]` 方法时，它将会按 `views/控制器 ID/视图名.php` 路径加载 PHP 文件。

注意以上代码，`message` 参数在输出之前被 `[[yii\helpers\Html::encode()|HTML-encoded]]` 方法处理过。这很有必要，当参数来自于最终用户时，参数中可能隐含的恶意 JavaScript 代码会导致[跨站脚本 \(XSS\) 攻击](#)。

当然了，你大概会在 `say` 视图里放入更多内容。内容可以由 HTML 标签，纯文本，甚至 PHP 语句组成。实际上 `say` 视图就是一个由 `[[yii\web\Controller::render()|render()]]` 执行的 PHP 脚本。视图脚本输出的内容将会作为响应结果返回给应用。应用将依次输出结果给最终用户。

试运行

创建完操作和视图后，你就可以通过下面的 URL 访问新页面了：

```
http://hostname/index.php?r=site/say&message=Hello+World
```

这个 URL 将会输出包含 “Hello World” 的页面，页面和应用里的其它页面使用同样的头部和尾部。

如果你省略 URL 中的 `message` 参数，将会看到页面只显示 “Hello”。这是因为 `message` 被作为一个参数传给 `actionSay()` 方法，当省略它时，参数将使用默认的 “Hello” 代替。

补充：新页面和其它页面使用同样的头部和尾部是因为 `[[yii\web\Controller::render()|render()]]` 方法会自动把 `say` 视图执行的结果嵌入称为布局的文件中，本例中是 `views/layouts/main.php`。

上面 URL 中的参数 `r` 需要更多解释。它代表路由，是整个应用级的，指向特定操作的独立 ID。路由格式是 控制器 ID/操作 ID。应用接受请求的时候会检查参数，使用控制器 ID 去确定哪个控制器应该被用来处理请求。然后相应控制器将使用操作 ID 去确定哪个操作方法将被用来做具体工作。上述例子中，路由 `site/say` 将被解析至 `SiteController` 控制器和其中的 `say` 操作。因此 `SiteController::actionSay()` 方法将被调用处理请求。

补充：与操作一样，一个应用中控制器同样有唯一的 ID。控制器 ID 和操作 ID 使用同样的命名规则。控制器的类名源自于控制器 ID，移除了连字符，每个单词首字母大写，并加上 `Controller` 后缀。例子：控制器 ID `post-comment` 相当于控制器类名 `PostCommentController`。

总结

通过本章节你接触了 MVC 设计模式中的控制器和视图部分。创建了一个操作作为控制器的一部分去处理特定请求。然后又创建了一个视图去构造响应内容。在这个小例子中，没有模型调用，唯一涉及到数据的地方是 `message` 参数。

你同样学习了 Yii 路由的相关内容，它是用户请求与控制器操作之间的桥梁。

下一章，你将学习如何创建一个模型，以及添加一个包含 HTML 表单的页面。

使用表单

本章节介绍如何创建一个让用户提交数据的表单页。该页将显示一个包含 **name** 输入框和 **email** 输入框的表单。当提交这两部分信息后，页面将会显示用户所输入的信息。

为了实现这个目标，除了创建一个[操作](#)和两个[视图](#)外，还需要创建一个[模型](#)。

贯穿整个小节，你将会学到：

- 创建一个[模型](#)代表用户通过表单输入的数据
- 声明规则去验证输入的数据
- 在[视图](#)中生成一个 HTML 表单

创建模型

模型类 **EntryForm** 代表从用户那请求的数据，该类如下所示并存储在 **models/EntryForm.php** 文件中。请参考[类自动加载](#)章节获取更多关于类命名约定的介绍。

```
<?php

namespace app\models;

use yii\base\Model;

class EntryForm extends Model
{
    public $name;

    public $email;

    public function rules()
    {
        return [

            [['name', 'email'], 'required'],
```

```

        ['email', 'email'],

    ];

}
}

```

该类继承自 Yii 提供的一个基类 `[[yii\base\Model]]`，该基类通常用来表示数据。

补充: `[[yii\base\Model]]` 被用于普通模型类的父类并与数据表无关。

`[[yii\db\ActiveRecord]]` 通常是普通模型类的父类但与数据表有关联（译注: `[[yii\db\ActiveRecord]]` 类其实也是继承自 `[[yii\base\Model]]`，增加了数据库处理）。

`EntryForm` 类包含 `name` 和 `email` 两个公共成员，用来储存用户输入的数据。它还包含一个名为 `rules()` 的方法，用来返回数据验证规则的集合。上面声明的验证规则表示:

- `name` 和 `email` 值都是必须的
- `email` 的值必须满足 `email` 规则验证

如果你有一个处理用户提交数据的 `EntryForm` 对象，你可以调用它的

`[[yii\base\Model::validate()|validate()]]` 方法触发数据验证。如果有数据验证失败，将把 `[[yii\base\Model::hasErrors|hasErrors]]` 属性设为 `true`，想要知道具体发生什么错误就调用 `[[yii\base\Model::getErrors|getErrors]]`。

```

<?php

$model = new EntryForm();

$model->name = 'Qiang';

$model->email = 'bad';

if ($model->validate()) {

    // 验证成功！

} else {

    // 失败！

    // 使用 $model->getErrors() 获取错误详情

}

```

创建操作

下面你得在 `site` 控制器中创建一个 `entry` 操作用于新建的模型。操作的创建和使用已经在[说一声你好](#)小节中解释了。

```

<?php

```

```

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ...其它代码...

    public function actionEntry()
    {
        $model = new EntryForm;

        if ($model->load(Yii::$app->request->post()) && $model->validate()) {
            // 验证 $model 收到的数据

            // 做些有意义的事 ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // 无论是初始化显示还是数据验证错误

            return $this->render('entry', ['model' => $model]);
        }
    }
}

```

该操作首先创建了一个 `EntryForm` 对象。然后尝试从 `$_POST` 搜集用户提交的数据，由 `Yii` 的 `[[yii\web\Request::post()]]` 方法负责搜集。如果模型被成功填充数据（也就是说用户已经提交了 HTML 表单），操作将调用 `[[yii\base\Model::validate()|validate()]]` 去确保用户提交的是有效数据。

补充：表达式 `Yii::$app` 代表应用实例，它是一个全局可访问的单例。同时它也是一个服务定位器，能提供 `request`，`response`，`db` 等等特定功能的组件。在上面的代码里就是使用 `request` 组件来访问应用实例收到的 `$_POST` 数据。

用户提交表单后，操作将会渲染一个名为 `entry-confirm` 的视图去确认用户输入的数据。如果没填表单就提交，或数据包含错误（译者：如 `email` 格式不对），`entry` 视图将会渲染输出，连同表单一起输出的还有验证错误的详细信息。

注意：在这个简单例子里我们只是呈现了有效数据的确认页面。实践中你应该考虑使用 `[[yii\web\Controller::refresh()|refresh()]]` 或 `[[yii\web\Controller::redirect()|redirect()]]` 去避免表单重复提交问题。

创建视图

最后创建两个视图文件 `entry-confirm` 和 `entry`。他们会被刚才创建的 `entry` 操作渲染。

`entry-confirm` 视图简单地显示提交的 `name` 和 `email` 数据。视图文件保存在 `views/site/entry-confirm.php`。

```
<?php
use yii\helpers\Html;
?>

<p>You have entered the following information:</p>

<ul>

    <li><label>Name</label>: <?= Html::encode($model->name) ?></li>

    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>

</ul>
```

`entry` 视图显示一个 HTML 表单。视图文件保存在 `views/site/entry.php`。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(); ?>

<?= $form->field($model, 'name') ?>
```



```
<?= $form->field($model, 'email') ?>
```

```
<div class="form-group">
```

```
<?= Html::submitButton('Submit', ['class' => 'btn btn-primary']) ?>
```

```
</div>
```

```
<?php ActiveForm::end(); ?>
```

视图使用了一个功能强大的小部件 `[[yii\widgets\ActiveForm|ActiveForm]]` 去生成 HTML 表单。其中的 `begin()` 和 `end()` 分别用来渲染表单的开始和关闭标签。在这两个方法之间使用了 `[[yii\widgets\ActiveForm::field()|field()]]` 方法去创建输入框。第一个输入框用于 “name”，第二个输入框用于 “email”。之后使用 `[[yii\helpers\Html::submitButton()]]` 方法生成提交按钮。

尝试下

用浏览器访问下面的 URL 看它能否工作:

<http://hostname/index.php?r=site/entry>

你会看到一个包含两个输入框的表单的页面。每个输入框的前面都有一个标签指明应该输入的数据类型。如果什么都不填就点击提交按钮，或填入格式不正确的 email 地址，将会看到在对应的输入框下显示错误信息。

My Company

HomeAboutContactLogin

Name

Name cannot be blank.

Email

Email cannot be blank.

Submit

© My Company 2014

Powered by  Yii Framework

输入有效的 `name` 和 `email` 信息并提交后，将会看到一个显示你所提交数据的确认页面。

My Company

[Home](#)[About](#)[Contact](#)[Login](#)

You have entered the following information:

- **Name:** Qiang Xue
- **Email:** tester@example.com

© My Company 2014

Powered by [Yii Framework](#)

效果说明

你可能会好奇 `HTML` 表单暗地里是如何工作的呢，看起来它可以为每个输入框显示文字标签，而当你没输入正确的信息时又不需要刷新页面就能给出错误提示，似乎有些神奇。

是的，其实数据首先由客户端 `JavaScript` 脚本验证，然后才会提交给服务器通过 `PHP` 验证。`[[yii\widgets\ActiveForm]]` 足够智能到把你在 `EntryForm` 模型中声明的验证规则转化成客户端 `JavaScript` 脚本去执行验证。如果用户浏览器禁用了 `JavaScript`，服务器端仍然会像 `actionEntry()` 方法里这样验证一遍数据。这保证了任何情况下用户提交的数据都是有效的。

警告：客户端验证是提高用户体验的手段。无论它是否正常启用，服务端验证则都是必须的，请不要忽略它。

输入框的文字标签是 `field()` 方法生成的，内容就是模型中该数据的属性名。例如模型中的 `name` 属性生成的标签就是 `Name`。

你可以在视图中自定义标签：

```
<?= $form->field($model, 'name')->label('自定义 Name') ?>

<?= $form->field($model, 'email')->label('自定义 Email') ?>
```

补充：Yii 提供了相当多类似的小部件去帮你生成复杂且动态的视图。在后面你还会了解到自己写小部件是多么简单。你可能会把自己的很多视图代码转化成小部件以提高重用，加快开发效率。

总结

本章节指南中你接触了 **MVC** 设计模式的每个部分。学到了如何创建一个模型代表用户数据并验证它的有效性。

你还学到了如何从用户那获取数据并在浏览器上回显给用户。这本来是开发应用的过程中比较耗时的任务，好在 **Yii** 提供了强大的小部件让它变得如此简单。

下一章你将学习如何使用数据库，几乎每个应用都需要数据库。

使用数据库

本章节将介绍如何如何创建一个从数据表 **country** 中读取国家数据并显示出来的页面。为了实现这个目标，你将会配置一个数据库连接，创建一个[活动记录](#)类，并且创建一个[操作](#)及一个[视图](#)。

贯穿整个章节，你将会学到：

- 配置一个数据库连接
- 定义一个活动记录类
- 使用活动记录从数据库中查询数据
- 以分页方式在视图中显示数据

请注意，为了掌握本章你应该具备最基本的数据库知识和使用经验。尤其是应该知道如何创建数据库，如何通过数据库终端执行 **SQL** 语句。

准备数据库

首先创建一个名为 **yii2basic** 的数据库，应用将从这个数据库中读取数据。你可以创建 **SQLite**，**MySQL**，**PostgreSQL**，**MSSQL** 或 **Oracle** 数据库，**Yii** 内置多种数据库支持。简单起见，后面的内容将以 **MySQL** 为例做演示。

然后在数据库中创建一个名为 **country** 的表并插入简单的数据。可以执行下面的语句：

```
CREATE TABLE `country` (  
    `code` CHAR(2) NOT NULL PRIMARY KEY,  
    `name` CHAR(52) NOT NULL,  
    `population` INT(11) NOT NULL DEFAULT '0'  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
INSERT INTO `country` VALUES ('AU','Australia',18886000);  
INSERT INTO `country` VALUES ('BR','Brazil',170115000);  
INSERT INTO `country` VALUES ('CA','Canada',1147000);  
INSERT INTO `country` VALUES ('CN','China',1277558000);
```

```
INSERT INTO `country` VALUES ('DE','Germany',82164700);
INSERT INTO `country` VALUES ('FR','France',59225700);
INSERT INTO `country` VALUES ('GB','United Kingdom',59623400);
INSERT INTO `country` VALUES ('IN','India',1013662000);
INSERT INTO `country` VALUES ('RU','Russia',146934000);
INSERT INTO `country` VALUES ('US','United States',278357000);
```

此时便有了一个名为 `yii2basic` 的数据库，在这个数据库中有一个包含三个字段的数据表 `country`，表中有十行数据。

配置数据库连接

开始之前，请确保你已经安装了 `PHP PDO` 扩展和你所使用的数据库的 `PDO` 驱动（例如 `MySQL` 的 `pdo_mysql`）。对于使用关系型数据库来讲，这是基本要求。

驱动和扩展安装可用后，打开 `config/db.php` 修改里面的配置参数对应你的数据库配置。该文件默认包含这些内容：

```
<?php

return [

    'class' => 'yii\db\Connection',

    'dsn' => 'mysql:host=localhost;dbname=yii2basic',

    'username' => 'root',

    'password' => '',

    'charset' => 'utf8',

];
```

`config/db.php` 是一个典型的基于文件的配置工具。这个文件配置了数据库连接 `[[yii\db\Connection]]` 的创建和初始化参数，应用的 `SQL` 查询正是基于这个数据库。

上面配置的数据库连接可以在应用中通过 `Yii::$app->db` 表达式访问。

补充：`config/db.php` 将被包含在应用配置文件 `config/web.php` 中，后者指定了整个应用如何初始化。请参考配置章节了解更多信息。

创建活动记录

创建一个继承自 `活动记录` 类的类 `Country`，把它放在 `models/Country.php` 文件，去代表和读取 `country` 表的数据。

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

这个 **Country** 类继承自 `[[yii\db\ActiveRecord]]`。你不用在里面写任何代码。只需要像现在这样，Yii 就能根据类名去猜测对应的数据表名。

补充：如果类名和数据表名不能直接对应，可以覆写 `[[yii\db\ActiveRecord::tableName()|tableName()]]` 方法去显式指定相关表名。

使用 **Country** 类可以很容易地操作 **country** 表数据，就像这段代码：

```
use app\models\Country;

// 获取 country 表的所有行并以 name 排序
$countries = Country::find()->orderBy('name')->all();

// 获取主键为 “US” 的行
$country = Country::findOne('US');

// 输出 “United States”
echo $country->name;

// 修改 name 为 “U.S.A.” 并在数据库中保存更改
$country->name = 'U.S.A.';
$country->save();
```

补充：活动记录是面向对象、功能强大的访问和操作数据库数据的方式。你可以在[活动记录](#)章节了解更多信息。除此之外你还可以使用另一种更原生的

被称做[数据访问对象](#)的方法操作数据库数据。

创建操作

为了向最终用户显示国家数据，你需要创建一个操作。相比之前小节掌握的在 `site` 控制器中创建操作，在这里为所有和国家有关的数据新建一个控制器更加合理。新控制器名为 `CountryController`，并在其中创建一个 `index` 操作，如下：

```
<?php

namespace app\controllers;

use yii\web\Controller;
use yii\data\Pagination;
use app\models\Country;

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();
    }
}
```

```

return $this->render('index', [

    'countries' => $countries,

    'pagination' => $pagination,

]);

}

}

```

把上面的代码保存在 `controllers/CountryController.php` 文件中。

`index` 操作调用了活动记录 `Country::find()` 方法，去生成查询语句并从 `country` 表中取回所有数据。为了限定每个请求所返回的国家数量，查询在 `[[yii\data\Pagination]]` 对象的帮助下进行分页。

`Pagination` 对象的使命主要有两点：

- 为 SQL 查询语句设置 `offset` 和 `limit` 从句，确保每个请求只需返回一页数据（本例中每页是 5 行）。
- 在视图中显示一个由页码列表组成的分页器，这点将在后面的段落中解释。

在代码末尾，`index` 操作渲染一个名为 `index` 的视图，并传递国家数据和分页信息进去。

创建视图

在 `views` 目录下先创建一个名为 `country` 的子目录。这个目录存储所有由 `country` 控制器渲染的视图。在 `views/country` 目录下创建一个名为 `index.php` 的视图文件，内容如下：

```

<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>

<h1>Countries</h1>

<ul>

<?php foreach ($countries as $country): ?>

    <li>

        <?= Html::encode("{ $country->name } ( { $country->code } )") ?>:

        <?= $country->population ?>

    </li>

<?php endforeach; ?>

</ul>

```

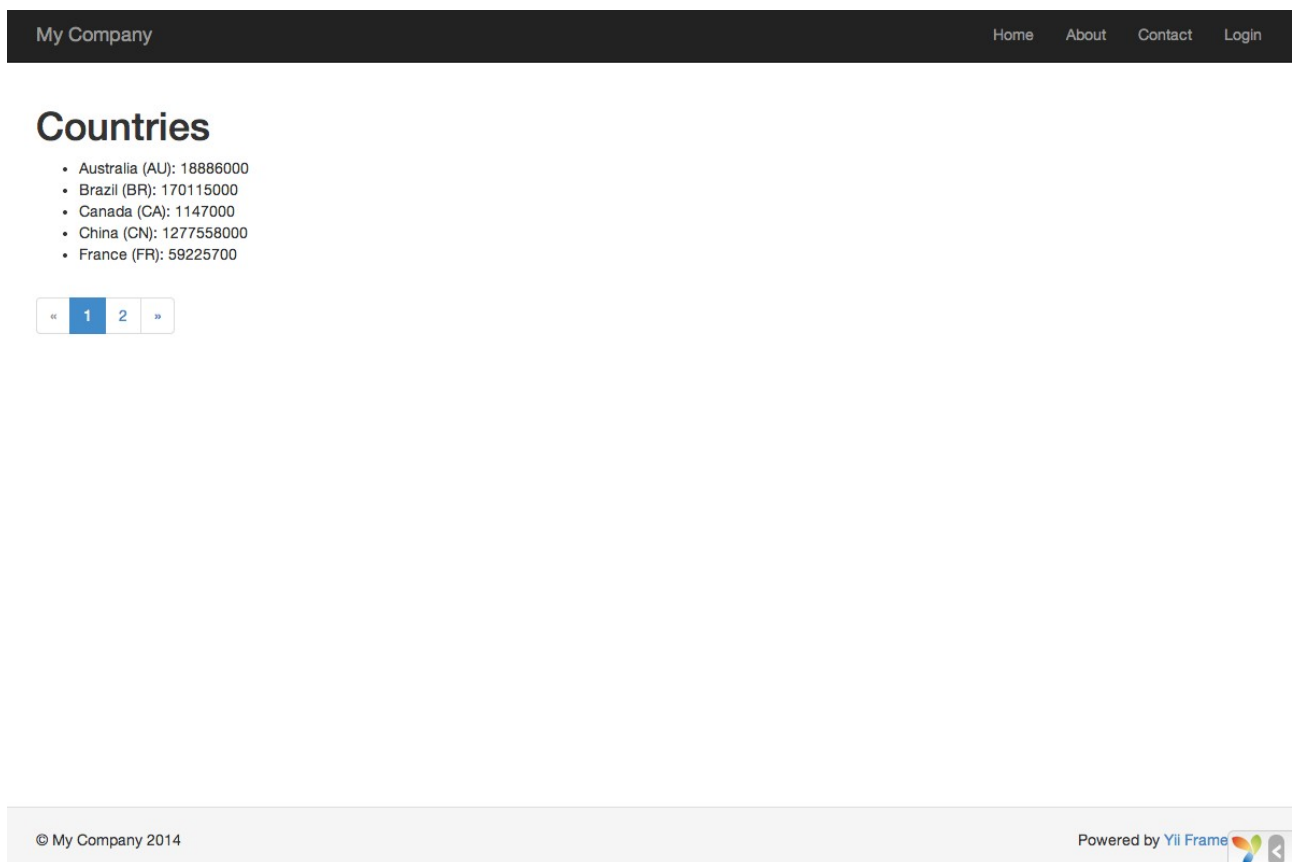
```
<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

这个视图包含两部分用以显示国家数据。第一部分遍历国家数据并以无序 **HTML** 列表渲染出来。第二部分使用 `[[yii\widgets\LinkPager]]` 去渲染从操作中传来的分页信息。小部件 `LinkPager` 显示一个分页按钮的列表。点击任何一个按钮都会跳转到对应的分页。

试运行

浏览器访问下面的 URL 看看能否工作:

<http://hostname/index.php?r=country/index>



首先你会看到显示着五个国家的列表页面。在国家下面，你还会看到一个包含四个按钮的分页器。如果你点击按钮 “2”，将会跳转到显示另外五个国家的页面，也就是第二页记录。如果观察仔细点你还会看到浏览器的 URL 变成了:

<http://hostname/index.php?r=country/index&page=2>

在这个场景里，`[[yii\data\Pagination|Pagination]]` 提供了为数据结果集分页的所有功能:

- 首先 `[[yii\data\Pagination|Pagination]]` 把 **SELECT** 的子查询 `LIMIT 5 OFFSET 0` 数据表示成第一页。因此开头的五条数据会被取出并显示。
- 然后小部件 `[[yii\widgets\LinkPager|LinkPager]]` 使用 `[[yii\data\Pagination::createUrl()|Pagination::createUrl()]]` 方法生成的 URL 去渲染翻页按钮。URL 中包含必要的参数 `page` 才能查询不同的页面编号。

- 如果你点击按钮 “2”，将会发起一个路由为 `country/index` 的新请求。
[[yii\data\Pagination\Pagination]] 接收到 URL 中的 `page` 参数把当前的页码设为 2。新的数据库请求将会以 `LIMIT 5 OFFSET 5` 查询并显示。

总结

本章节中你学到了如何使用数据库。你还学到了如何取出并使用 [[yii\data\Pagination]] 和 [[yii\widgets\LinkPager]] 显示数据。

下一章中你会学到如何使用 Yii 中强大的代码生成器 Gii，去帮助你实现一些常用的功能需求，例如增查改删（CRUD）数据表中的数据。事实上你之前所写的代码全部都可以由 Gii 自动生成。

使用 Gii 生成代码

本章将介绍如何使用 Gii 去自动生成 Web 站点常用功能的代码。使用 Gii 生成代码非常简单，只要按照 Gii 页面上的介绍输入正确的信息即可。

贯穿本章节，你将会学到：

- 在你的应用中开启 Gii
- 使用 Gii 去生成活动记录类
- 使用 Gii 去生成数据表操作的增查改删（CRUD）代码
- 自定义 Gii 生成的代码

开始 Gii

Gii 是 Yii 中的一个模块。可以通过配置应用的 [[yii\base\Application::modules|modules]] 属性开启它。通常来讲在 `config/web.php` 文件中会有以下配置代码：

```
$config = [ ... ];

if (YII_ENV_DEV) {

    $config['bootstrap'][] = 'gii';

    $config['modules']['gii'] = 'yii\gii\Module';

}
```

这段配置表明，如果当前是开发环境，应用会包含 gii 模块，模块类是 [[yii\gii\Module]]。

如果你检查应用的入口脚本 `web/index.php`，将看到这行代码将 `YII_ENV_DEV` 设为 `true`：

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

鉴于这行代码的定义，应用处于开发模式下，按照上面的配置会打开 Gii 模块。你可以直接通过 URL 访问 Gii：

```
http://hostname/index.php?r=gii
```

补充： 如果你通过本机以外的机器访问 **Gii**，请求会被出于安全原因拒绝。
你可以配置 **Gii** 为其添加允许访问的 **IP** 地址：

```
'gii' => [  
    'class' => 'yii\gii\Module',  
    'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'] // 按需调整这里  
,
```

Welcome to Gii

a magical tool that can write code for you

Start the fun with the following code generators:

Model Generator

This generator generates an ActiveRecord class for the specified database table.

[Start »](#)

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

[Start »](#)

Controller Generator

This generator helps you to quickly generate a new controller class, one or several controller actions and their corresponding views.

[Start »](#)

Form Generator

This generator generates a view script file that displays a form to collect input for the specified model class.

[Start »](#)

Module Generator

This generator helps you to generate the skeleton code needed by a Yii module.

[Start »](#)

Extension Generator

This generator helps you to generate the files needed by a Yii extension.

[Start »](#)[Get More Generators](#)

生成活动记录类

选择 “Model Generator” （点击 [Gii](#) 首页的链接）去生成活动记录类。并像这样填写表单：

•Table Name: **country**

•Model Class: **Country**

Model Generator >

CRUD Generator >

Controller Generator >

Form Generator >

Module Generator >

Extension Generator >

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name

country

Model Class

Country

Namespace

app\models

Base Class

yii\db\ActiveRecord

Database Connection ID

db

☐ Use Table Prefix☒ Generate Relations☐ Generate Labels from DB Comments☐ Enable I18N**Code Template**

```
default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)
t)
```

Preview



然后点击 “Preview” 按钮。你会看到 `models/Country.php` 被列在将要生成的文件列表中。可以点击文件名预览内容。

如果你已经创建过同样的文件，使用 Gii 会覆写它，点击文件名旁边的 `diff` 能查看现有文件与将要生成

的文件的内容区别。

Home Help Application

Model Generator

CRUD Generator

Controller Generator

Form Generator

Module Generator

Extension Generator

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name
country

Model Class
Country

Namespace
app\models

Base Class
yii\db\ActiveRecord

Database Connection ID
db

☐ Use Table Prefix

☒ Generate Relations

☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template
default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

Preview

Generate

Click on the above **Generate** button to generate the files selected below:

☒ Create

☒ Unchanged

☒ Overwrite

Code File	Action
models/Country.php diff	overwrite

想要覆写已存在文件，选中 “overwrite” 下的复选框然后点击 “Generator”。如果是新文件，只点击 “Generator” 就好。

接下来你会看到一个包含已生成文件的说明页面。如果生成过程中覆写过文件，还会有一条信息说明代码是重新生成覆盖的。

生成 **CRUD** 代码

CRUD 代表增，查，改，删操作，这是绝大多数 Web 站点常用的数据处理方式。选择 Gii 中的 “CRUD Generator”（点击 Gii 首页的链接）去创建 CRUD 功能。本例 “country” 中需要这样填写表单：

- Model Class: `app\models\Country`
- Search Model Class: `app\models\CountrySearch`

•Controller Class: `app\controllers\CountryController`

yii code generator

Home Help Application

Model Generator >

CRUD Generator >

Controller Generator >

Form Generator >

Module Generator >

Extension Generator >

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Model Class

Search Model Class

Controller Class

Base Controller Class

Module ID

Widget Used in Index Page

☐ Enable I18N

Code Template

Preview

A Product of Yii Software LLC

Powered by [Yii Framework](#)

然后点击 “Preview” 按钮。你会看到下述将要生成的文件列表。

[[NEED THE IMAGE HERE / 等待官方补充图片]]

如果你之前创建过 `controllers/CountryController.php` 和 `views/country/index.php` 文件（在指南的使用数据库章节），选中 “overwrite” 下的复选框覆写它们（之前的文件没能全部支持 CRUD）。

试运行

用浏览器访问下面的 URL 查看生成代码的运行：

`http://hostname/index.php?r=country/index`

可以看到一个栅格显示着从数据表中读取的国家数据。支持在列头对数据进行排序，输入筛选条件进行筛选。

可以浏览详情，编辑，或删除栅格中的每个国家。还可以点击栅格上方的 “Create Country” 按钮通过表单创建新国家。

Countries

[Create Country](#)

Showing 1-10 of 10 items.

#	Code	Name	Population	
1	AU	Australia	18886000	  
2	BR	Brazil	170115000	  
3	CA	Canada	1147000	  
4	CN	China	1277558000	  
5	DE	Germany	82164700	  
6	FR	France	59225700	  
7	GB	United Kingdom	59623400	  
8	IN	India	1013662000	  
9	RU	Russia	146934000	  
10	US	United States	278357000	  



Update Country: United States

Code

Name

Population



下面列出由 **Gii** 生成的文件，以便你研习功能和实现，或修改它们。

- 控制器: `controllers/CountryController.php`
- 模型: `models/Country.php` 和 `models/CountrySearch.php`
- 视图: `views/country/*.php`

补充: **Gii** 被设计成高度可定制和可扩展的代码生成工具。使用它可以大幅提高应用开发速度。请参考 [Gii](#) 章节了解更多内容。

总结

本章学习了如何使用 **Gii** 去生成数据表中数据实现完整 **CRUD** 功能的代码。

更上一层楼

通篇阅读完整个“入门”部分，你就完成了一个完整 **Yii** 应用的创建。在此过程中你学到了如何实现一些常用功能，例如通过 **HTML** 表单从用户那获取数据，从数据库中获取数据并以分页形式显示。你还学到了如何通过 **Gii** 去自动生成代码。使用 **Gii** 生成代码把 **Web** 开发中多数繁杂的过程转化为仅仅填写几个表单就行。

本章将介绍一些有助于更好使用 **Yii** 的资源：

- 文档
 - 权威指南：顾名思义，指南详细描述了 **Yii** 的工作原理并提供了如何使用它的常规引导。这是最重要的 **Yii** 辅助资料，强烈建议在开始写 **Yii** 代码之前阅读。
 - 类参考手册：描述了 **Yii** 中每个类的用法。在编码过程中这极为有用，能够帮你理清某个特定类，方法，和属性的用法。类参考手册最好在整个框架的语境下去理解。
 - Wiki 文章：Wiki 文章是 **Yii** 用户在其自身经验基础上分享出来的。大多数是使用教程或如何使用 **Yii** 解决特定问题。虽然这些文章质量可能并不如权威指南，但它们往往覆盖了更广泛的话题，并常常提供解决方案，所以它们也很有用。
 - 书籍
- 扩展：**Yii** 拥有数以千计用户提供的扩展，这些扩展能非常方便的插入到应用中，使你的应用开发过程更加方便快捷。
- 社区
 - 官方论坛：<http://www.yiiframework.com/forum/>
 - IRC 聊天室：Freenode 网络上的 **#yii** 频道 (<irc://irc.freenode.net/yii>)（使用英文哦，无需反馈上游的问题可以加 QQ-Yii2 中国交流群）
 - GitHub：<https://github.com/yiisoft/yii2>
 - Facebook：<https://www.facebook.com/groups/yiitalk/>
 - Twitter：<https://twitter.com/yiiframework>
 - LinkedIn：<https://www.linkedin.com/groups/yii-framework-1483367>

应用结构：

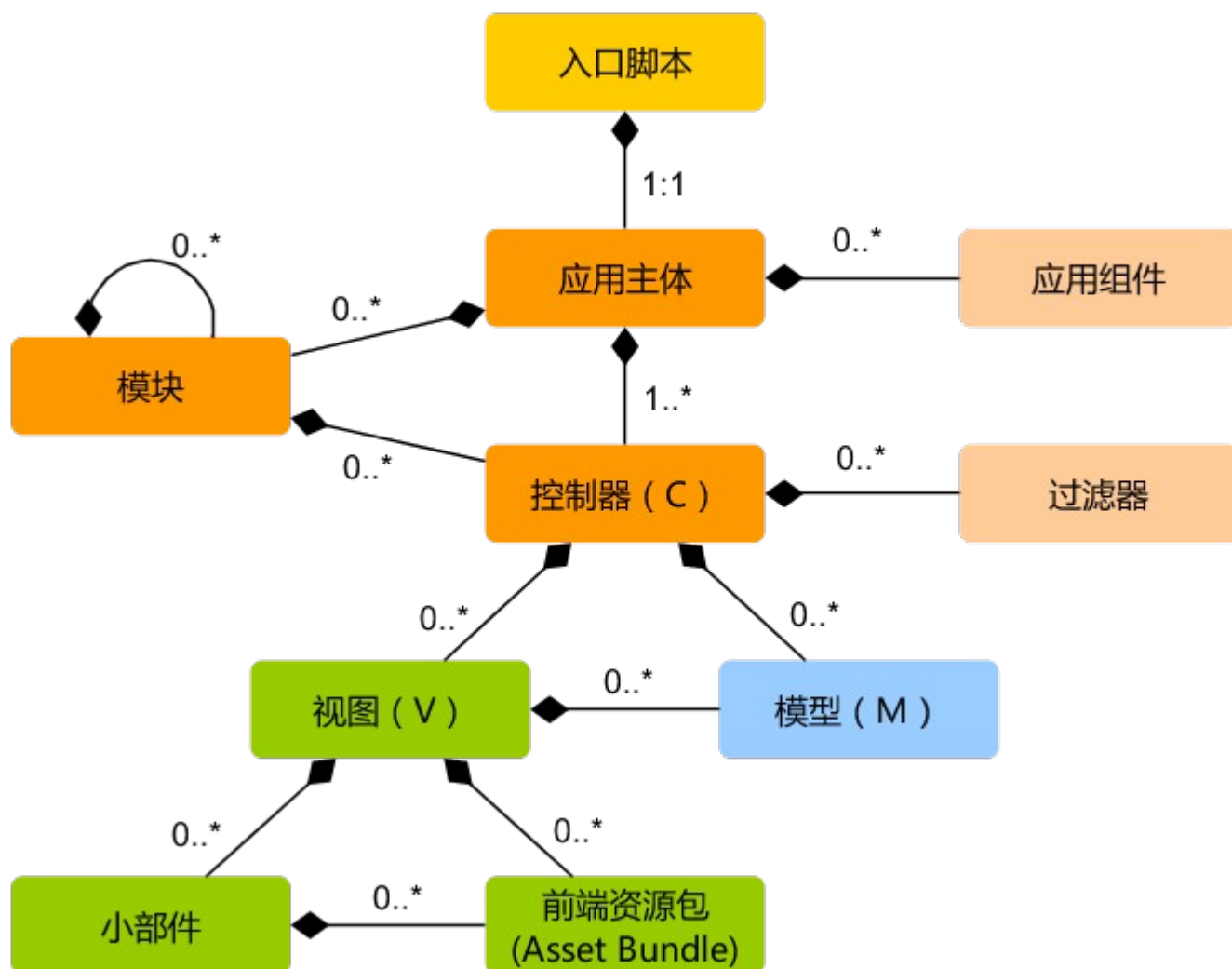
总览

Yii 应用参照**模型-视图-控制器 (MVC)** 设计模式来组织。**模型**代表数据、业务逻辑和规则；**视图**展示模型的输出；**控制器**接受出入并将其转换为**模型**和**视图**命令。

除了 MVC, Yii 应用还有以下部分：

- **入口脚本**：终端用户能直接访问的 PHP 脚本，负责启动一个请求处理周期。
- **应用**：能全局范围内访问的对象，管理协调组件来完成请求。
- **应用组件**：在应用中注册的对象，提供不同的功能来完成请求。
- **模块**：包含完整 MVC 结构的独立包，一个应用可以由多个模块组建。
- **过滤器**：控制器在处理请求之前或之后需要触发执行的代码。
- **小部件**：可嵌入到**视图**中的对象，可包含控制器逻辑，可被不同视图重复调用。

下面的示意图展示了 Yii 应用的静态结构：



入口脚本

入口脚本是应用启动流程中的第一环，一个应用（不管是网页应用还是控制台应用）只有一个入口脚本。终端用户的请求通过入口脚本实例化应用并将请求转发到应用。

Web 应用的入口脚本必须放在终端用户能够访问的目录下，通常命名为 `index.php`，也可以使用 Web 服务器能定位到的其他名称。

控制台应用的入口脚本一般在应用根目录下命名为 `yii`（后缀为`.php`），该文件需要有执行权限，这样用户就能通过命令 `./yii <route> [arguments] [options]` 来运行控制台应用。

入口脚本主要完成以下工作：

- 定义全局常量；
- 注册 [Composer 自动加载器](#)；
- 包含 `[[Yii]]` 类文件；
- 加载应用配置；
- 创建一个[应用实例](#)并配置；
- 调用 `[[yii\base\Application::run()]]` 来处理请求。

Web 应用

以下是[基础应用模版](#)入口脚本的代码：

```
<?php

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// 注册 Composer 自动加载器
require(__DIR__ . '/../vendor/autoload.php');

// 包含 Yii 类文件
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

// 加载应用配置
$config = require(__DIR__ . '/../config/web.php');
```

```
// 创建、配置、运行一个应用
```

```
(new yii\web\Application($config))->run();
```

控制台应用

以下是一个控制台应用的入口脚本：

```
#!/usr/bin/env php
```

```
<?php
```

```
/**
```

```
 * Yii console bootstrap file.
```

```
 *
```

```
 * @link http://www.yiiframework.com/
```

```
 * @copyright Copyright (c) 2008 Yii Software LLC
```

```
 * @license http://www.yiiframework.com/license/
```

```
 */
```

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

```
// fcgi 默认没有定义 STDIN 和 STDOUT
```

```
defined('STDIN') or define('STDIN', fopen('php://stdin', 'r'));
```

```
defined('STDOUT') or define('STDOUT', fopen('php://stdout', 'w'));
```

```
// 注册 Composer 自动加载器
```

```
require(__DIR__ . '/vendor/autoload.php');
```

```
// 包含 Yii 类文件
```

```
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');
```

```
// 加载应用配置
```

```
$config = require(__DIR__ . '/config/console.php');

$application = new yii\console\Application($config);

$exitCode = $application->run();

exit($exitCode);
```

定义常量

入口脚本是定义全局常量的最好地方，Yii 支持以下三个常量：

- **YII_DEBUG**：标识应用是否运行在调试模式。当在调试模式下，应用会保留更多日志信息，如果抛出异常，会显示详细的错误调用堆栈。因此，调试模式主要适合在开发阶段使用，**YII_DEBUG** 默认值为 **false**。
- **YII_ENV**：标识应用运行的环境，详情请查阅[配置](#)章节。**YII_ENV** 默认值为 **'prod'**，表示应用运行在线上产品环境。
- **YII_ENABLE_ERROR_HANDLER**：标识是否启用 Yii 提供的错误处理，默认为 **true**。

当定义一个常量时，通常使用类似如下代码来定义：

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

上面的代码等同于：

```
if (!defined('YII_DEBUG')) {

    define('YII_DEBUG', true);

}
```

显然第一段代码更加简洁易懂。

常量定义应该在入口脚本的开头，这样包含其他 **PHP** 文件时，常量就能生效。

应用主体

应用主体是管理 Yii 应用系统整体结构和生命周期的对象。每个 Yii 应用系统只能包含一个应用主体，应用主体在 [入口脚本](#) 中创建并能通过表达式 **\Yii::\$app** 全局范围内访问。

补充：当我们说“一个应用”，它可能是一个应用主体对象，也可能是一个应用系统，是根据上下文来决定[译：中文为避免歧义，**Application** 翻译为应用主体]。

Yii 有两种应用主体：[[yii\web\Application|网页应用主体]] and [[yii\console\Application|控制台应用主体]]，如名称所示，前者主要处理网页请求，后者处理控制台请求。

应用主体配置

如下所示，当 [入口脚本](#) 创建了一个应用主体，它会加载一个 [配置](#) 文件并传给应用主体。

```
require(__DIR__ . '/../vendor/autoload.php');

require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

// 加载应用主体配置

$config = require(__DIR__ . '/../config/web.php');

// 实例化应用主体、配置应用主体

(new yii\web\Application($config))->run();
```

类似其他 [配置](#) 文件，应用主体配置文件标明如何设置应用对象初始属性。由于应用主体配置比较复杂，一般保存在多个类似如上 `web.php` 的 [配置文件](#) 当中。

应用主体属性

应用主体配置文件中有许多重要的属性要配置，这些属性指定应用主体的运行环境。比如，应用主体需要知道如何加载 [控制器](#)，临时文件保存到哪儿等等。下面我们简述这些属性。

必要属性

在一个应用中，至少要配置 2 个属性：[[yii\base\Application::id|id]] 和 [[yii\base\Application::basePath|basePath]]。

[[yii\base\Application::id|id]]

[[yii\base\Application::id|id]] 属性用来区分其他应用的唯一标识 ID。主要给程序使用。为了方便协作，最好使用数字作为应用主体 ID，但不强制要求为数字。

[[yii\base\Application::basePath|basePath]]

[[yii\base\Application::basePath|basePath]] 指定该应用的根目录。根目录包含应用系统所有受保护的源代码。在根目录下可以看到对应 MVC 设计模式的 `models`, `views`, `controllers` 等子目录。

可以使用路径或 [路径别名](#) 来在配置 [[yii\base\Application::basePath|basePath]] 属性。两种格式所对应的目录都必须存在，否则系统会抛出一个异常。系统会使用 `realpath()` 函数规范化配置的路径。

[[yii\base\Application::basePath|basePath]] 属性经常用于派生一些其他重要路径（如 `runtime` 路径），因此，系统预定义 `@app` 代表这个路径。派生路径可以通过这个别名组成（如 `@app/runtime` 代表 `runtime` 的路径）。

重要属性

本小节所描述的属性通常需要设置，因为不同的应用属性不同。

`[[yii\base\Application::aliases|aliases]]`

该属性允许你用一个数组定义多个 [别名](#)。数组的 **key** 为别名名称，值为对应的路径。例如：

```
[  
    'aliases' => [  
        '@name1' => 'path/to/path1',  
        '@name2' => 'path/to/path2',  
    ],  
]
```

使用这个属性来定义别名，代替 `[[Yii::setAlias()]]` 方法来设置。

`[[yii\base\Application::bootstrap|bootstrap]]`

这个属性很实用，它允许你用数组指定启动阶段`[[yii\base\Application::bootstrap()|bootstrapping process]]`需要运行的组件。比如，如果你希望一个 [模块](#) 自定义 [URL 规则](#)，你可以将模块 ID 加入到 `bootstrap` 数组中。

属性中的每个组件需要指定以下一项：

- 应用 [组件 ID](#)。
- [模块](#) ID。
- 类名。
- 配置数组。
- 创建并返回一个组件的无名称函数。

例如：

```
[  
    'bootstrap' => [  
        // 应用组件 ID 或模块 ID  
        'demo',  
        // 类名  
        'app\components\Profiler',  
    ],  
]
```

```

        // 配置数组

        [

            'class' => 'app\components\Profiler',

            'level' => 3,

        ],

        // 无名称函数

        function () {

            return new app\components\Profiler();

        }

    ],

]

```

补充: 如果模块 ID 和应用组件 ID 同名, 优先使用应用组件 ID, 如果你想用模块 ID, 可以使用如下无名称函数返回模块 ID。

```

function () {

    return Yii::$app->getModule('user');

},

```

```

]`

```

在启动阶段, 每个组件都会实例化。如果组件类实现接口 `[[yii\base\BootstrapInterface]]`, 也会调用 `[[yii\base\BootstrapInterface::bootstrap()|bootstrap()]]` 方法。

举一个实际的例子, [Basic Application Template](#) 应用主体配置中, 开发环境下会在启动阶段运行 `debug` 和 `gii` 模块。

```

if (YII_ENV_DEV) {

    // configuration adjustments for 'dev' environment

    $config['bootstrap'][] = 'debug';

    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';

```



```
$config['modules']['gii'] = 'yii\gii\Module';  
}
```

注：启动太多的组件会降低系统性能，因为每次请求都需要重新运行启动组件，因此谨慎配置启动组件。

[[yii\web\Application::catchAll|catchAll]]

该属性仅 [[yii\web\Application|Web applications]] 网页应用支持。它指定一个要处理所有用户请求的 [控制器方法](#)，通常在维护模式下使用，同一个方法处理所有用户请求。

该配置为一个数组，第一项指定动作的路由，剩下的数组项(key-value 成对)指定传递给动作的参数，例如：

```
[  
    'catchAll' => [  
        'offline/notice',  
        'param1' => 'value1',  
        'param2' => 'value2',  
    ],  
]
```

[[yii\base\Application::components|components]]

这是最重要的属性，它允许你注册多个在其他地方使用的[应用组件](#)。例如

```
[  
    'components' => [  
        'cache' => [  
            'class' => 'yii\caching\FileCache',  
        ],  
        'user' => [  
            'identityClass' => 'app\models\User',  
            'enableAutoLogin' => true,  
        ],  
    ],  
]
```

每一个应用组件指定一个 key-value 对的数组，key 代表组件 ID，value 代表组件类名或 [配置](#)。

在应用中可以任意注册组件，并可以通过表达式 `\Yii::$app->ComponentID` 全局访问。

详情请阅读 [应用组件](#) 一节。

[[yii\base\Application::controllerMap|controllerMap]]

该属性允许你指定一个控制器 ID 到任意控制器类。Yii 遵循一个默认的 [规则](#)指定控制器 ID 到任意控制器类（如 `post` 对应 `app\controllers\PostController`）。通过配置这个属性，可以打破这个默认规则，在下面的例子中，`account` 对应到 `app\controllers\UserController`，`article` 对应到 `app\controllers\PostController`。

```
[
    'controllerMap' => [
        [
            'account' => 'app\controllers\UserController',
            'article' => [
                'class' => 'app\controllers\PostController',
                'enableCsrfValidation' => false,
            ],
        ],
    ],
],
```

数组的键代表控制器 ID，数组的值代表对应的类名。

[[yii\base\Application::controllerNamespace|controllerNamespace]]

该属性指定控制器类默认的命名空间，默认为 `app\controllers`。比如控制器 ID 为 `post` 默认对应 `PostController`（不带命名空间），类全名为 `app\controllers\PostController`。

控制器类文件可能放在这个命名空间对应目录的子目录下，例如，控制器 ID `admin/post` 对应的控制器类全名为 `app\controllers\admin\PostController`。

控制器类全面能被 [自动加载](#)，这点是非常重要的，控制器类的实际命名空间对应这个属性，否则，访问时你会收到“Page Not Found”[译：页面找不到]。

如果你想打破上述的规则，可以配置 `controllerMap` 属性。

[[yii\base\Application::language|language]]

该属性指定应用展示给终端用户的语言，默认为 `en` 标识英文。如果需要之前其他语言可以配置该属性。

该属性影响各种 [国际化](#)，包括信息翻译、日期格式、数字格式等。例如 `[[yii\ui\DatePicker]]` 小部件会根据该属性展示对应语言的日历以及日期格式。

推荐遵循 [IETF language tag](#) 来设置语言，例如 `en` 代表英文，`en-US` 代表英文(美国)。

该属性的更多信息可参考 [国际化](#) 一节。

[[yii\base\Application::modules|modules]]

该属性指定应用所包含的 [模块](#)。

该属性使用数组包含多个模块类 [配置](#)，数组的键为模块 ID，例：

```
[
    'modules' => [
        // "booking" 模块以及对应的类
        'booking' => 'app\modules\booking\BookingModule',

        // "comment" 模块以及对应的配置数组
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

更多详情请参考 [模块](#) 一节。

[[yii\base\Application::name|name]]

该属性指定你可能想展示给终端用户的应用名称，不同于需要唯一性的 `[[yii\base\Application::id|id]]` 属性，该属性可以不唯一，该属性用于显示应用的用途。

如果其他地方的代码没有用到，可以不配置该属性。

[[yii\base\Application::params|params]]

该属性为一个数组，指定可以全局访问的参数，代替程序中硬编码的数字和字符，应用中的参数定义到一个单独的文件并随时可以访问是一个好习惯。例如用参数定义缩略图的长宽如下：

```
[  
  
    'params' => [  
  
        'thumbnail.size' => [128, 128],  
  
    ],  
  
]
```

然后简单的使用如下代码即可获取到你需要的长宽参数：

```
$size = \Yii::$app->params['thumbnail.size'];  
$width = \Yii::$app->params['thumbnail.size'][0];
```

以后想修改缩略图长宽，只需要修改该参数而不需要相关的代码。

[[yii\base\Application::sourceLanguage|sourceLanguage]]

该属性指定应用代码的语言，默认为 `'en-US'` 标识英文（美国），如果应用不是英文请修改该属性。

和 [语言](#) 属性类似，配置该属性需遵循 [IETF language tag](#)。例如 `en` 代表英文，`en-US` 代表英文(美国)。

该属性的更多信息可参考 [国际化](#) 一节。

[[yii\base\Application::timeZone|timeZone]]

该属性提供一种方式修改 PHP 运行环境中的默认时区，配置该属性本质上就是调用 PHP 函数 `date_default_timezone_set()`，例如：

```
[  
  
    'timeZone' => 'America/Los_Angeles',  
  
]
```

[[yii\base\Application::version|version]]

该属性指定应用的版本，默认为 `'1.0'`，其他代码不使用的可以不配置。

实用属性

本小节描述的属性不经常设置，通常使用系统默认值。如果你想改变默认值，可以配置这些属性。

[[yii\base\Application::charset|charset]]

该属性指定应用使用的字符集，默认值为 `'UTF-8'`，绝大部分应用都在使用，除非已有的系统大量使用非 unicode 数据才需要更改该属性。

[[yii\base\Application::defaultRoute|defaultRoute]]

该属性指定未配置的请求的响应 [路由](#) 规则，路由规则可能包含模块 ID，控制器 ID，动作 ID。例如

`help`, `post/create`, `admin/post/create`, 如果动作 ID 没有指定, 会使用 `[[yii\base\Controller::defaultAction]]` 中指定的默认值。

对于 `[[yii\web\Application|Web applications]]` 网页应用, 默认值为 `'site'` 对应 `SiteController` 控制器, 并使用默认的动作。因此你不带路由的访问应用, 默认会显示 `app\controllers\SiteController::actionIndex()` 的结果。

对于 `[[yii\console\Application|console applications]]` 控制台应用, 默认值为 `'help'` 对应 `[[yii\console\controllers\HelpController::actionIndex()]]`。因此, 如果执行的命令不带参数, 默认会显示帮助信息。

`[[yii\base\Application::extensions|extensions]]`

该属性用数组列表指定应用安装和使用的 [扩展](#), 默认使用 `@vendor/yiisoft/extensions.php` 文件返回的数组。当你使用 [Composer](#) 安装扩展, `extensions.php` 会被自动生成和维护更新。所以大多数情况下, 不需要配置该属性。

特殊情况下你想自己手动维护扩展, 可以参照如下配置该属性:

```
[
    'extensions' => [
        [
            'name' => 'extension name',

            'version' => 'version number',

            'bootstrap' => 'BootstrapClassName', // 可选配, 可为配置数组

            'alias' => [ // 可选配
                '@alias1' => 'to/path1',
                '@alias2' => 'to/path2',
            ],
        ],

        // ... 更多像上面的扩展 ...

    ],
]
```

如上所示, 该属性包含一个扩展定义数组, 每个扩展为一个包含 `name` 和 `version` 项的数组。如果扩展要在 [引导启动](#) 阶段运行, 需要配置 `bootstrap` 以及对应的引导启动类名或 `configuration` 数组。扩

展也可以定义 [别名](#)

[[yii\base\Application::layout|layout]]

该属性指定渲染 [视图](#) 默认使用的布局名字，默认值为 `'main'` 对应[布局路径](#)下的 `main.php` 文件，如果 [布局路径](#) 和 [视图路径](#) 都是默认值，默认布局文件可以使用路径别名

`@app/views/layouts/main.php`

如果不想设置默认布局文件，可以设置该属性为 `false`，这种做法比较罕见。

[[yii\base\Application::layoutPath|layoutPath]]

该属性指定查找布局文件的路径，默认值为 [视图路径](#) 下的 `layouts` 子目录。如果 [视图路径](#) 使用默认值，默认的布局路径别名为 `@app/views/layouts`。

该属性需要配置成一个目录或 [路径别名](#)。

[[yii\base\Application::runtimePath|runtimePath]]

该属性指定临时文件如日志文件、缓存文件等保存路径，默认值为带别名的 `@app/runtime`。

可以配置该属性为一个目录或者路径 [别名](#)，注意应用运行时有对该路径的写入权限，以及终端用户不能访问该路径因为临时文件可能包含一些敏感信息。

为了简化访问该路径，Yii 预定义别名 `@runtime` 代表该路径。

[[yii\base\Application::viewPath|viewPath]]

该路径指定视图文件的根目录，默认值为带别名的 `@app/views`，可以配置它为一个目录或者路径 [别名](#)。

[[yii\base\Application::vendorPath|vendorPath]]

该属性指定 [Composer](#) 管理的供应商路径，该路径包含应用使用的包括 Yii 框架在内的所有第三方库。默认值为带别名的 `@app/vendor`。

可以配置它为一个目录或者路径 [别名](#)，当你修改时，务必修改对应的 [Composer](#) 配置。

为了简化访问该路径，Yii 预定义别名 `@vendor` 代表该路径。

[[yii\console\Application::enableCoreCommands|enableCoreCommands]]

该属性仅 `[[yii\console\Application|console applications]]` 控制台应用支持，用来指定是否启用 Yii 中的核心命令，默认值为 `true`。

应用事件

应用在处理请求过程中会触发事件，可以在配置文件配置事件处理代码，如下所示：

[

```
'on beforeRequest' => function ($event) {  
  
    // ...  
  
},  
  
]
```

`on eventName` 语法的用法在 [Configurations](#) 一节有详细描述。

另外，在应用主体实例化后，你可以在[引导启动](#) 阶段附加事件处理代码，例如：

```
\Yii::$app->on(\yii\base\Application::EVENT_BEFORE_REQUEST, function ($event) {  
  
    // ...  
  
});
```

[[yii\base\Application::EVENT_BEFORE_REQUEST|EVENT_BEFORE_REQUEST]]

该事件在应用处理请求 `before` 之前，实际的事件名为 `beforeRequest`。

在事件触发前，应用主体已经实例化并配置好了，所以通过事件机制将你的代码嵌入到请求处理过程中非常不错。例如在事件处理中根据某些参数动态设置[[yii\base\Application::language]]语言属性。

[[yii\base\Application::EVENT_AFTER_REQUEST|EVENT_AFTER_REQUEST]]

该事件在应用处理请求 `after` 之后但在返回响应 `before` 之前触发，实际的事件名为 `afterRequest`。

该事件触发时，请求已经被处理完，可以做一些请求后处理或自定义响应。

注意 [[yii\web\Response|response]] 组件在发送响应给终端用户时也会触发一些事件，这些事件都在本事件 `after` 之后触发。

[[yii\base\Application::EVENT_BEFORE_ACTION|EVENT_BEFORE_ACTION]]

该事件在每个 [控制器动作](#) 运行 `before` 之前会被触发，实际的事件名为 `beforeAction`。

事件的参数为一个 [[yii\base\ActionEvent]] 实例，事件处理中可以设置 [[yii\base\ActionEvent::isValid]] 为 `false` 停止运行后续动作，例如：

```
[  
  
    'on beforeAction' => function ($event) {  
  
        if (some condition) {
```

```

        $event->isValid = false;

    } else {

    }

},

]

```

注意 **模块** 和 **控制器** 都会触发 **beforeAction** 事件。应用主体对象首先触发该事件，然后模块触发（如果存在模块），最后控制器触发。任何一个事件处理中设置 `[[yii\base\ActionEvent::isValid]]` 设置为 **false** 会停止触发后面的事件。

[[yii\base\Application::EVENT_AFTER_ACTION|EVENT_AFTER_ACTION]]

该事件在每个 **控制器动作** 运行 **after** 之后会被触发，实际的事件名为 **afterAction**。

该事件的参数为 `[[yii\base\ActionEvent]]` 实例，通过 `[[yii\base\ActionEvent::result]]` 属性，事件处理可以访问和修改动作的结果。例如：

```

[

    'on afterAction' => function ($event) {

        if (some condition) {

            // 修改 $event->result

        } else {

        }

    },

]

```

注意 **模块** 和 **控制器** 都会触发 **afterAction** 事件。这些对象的触发顺序和 **beforeAction** 相反，也就是说，控制器最先触发，然后是模块（如果有模块），最后为应用主体。

应用主体生命周期

当运行 **入口脚本** 处理请求时，应用主体会经历以下生命周期：

1. 入口脚本加载应用主体配置数组。
2. 入口脚本创建一个应用主体实例：

- 调用 `[[yii\base\Application::preInit()|preInit()]]` 配置几个高级别应用主体属性，比

如`[[yii\base\Application::basePath|basePath]]`。

- 注册 `[[yii\base\Application::errorHandler|error handler]]` 错误处理方法。
- 配置应用主体属性。
- 调用 `[[yii\base\Application::init()|init()]]` 初始化，该函数会调用 `[[yii\base\Application::bootstrap()|bootstrap()]]` 运行引导启动组件。

3.入口脚本调用 `[[yii\base\Application::run()]]` 运行应用主体：

- 触发 `[[yii\base\Application::EVENT_BEFORE_REQUEST|EVENT_BEFORE_REQUEST]]` 事件。
- 处理请求：解析请求 [路由](#) 和相关参数；创建路由指定的模块、控制器和动作对应的类，并运行动作。
- 触发 `[[yii\base\Application::EVENT_AFTER_REQUEST|EVENT_AFTER_REQUEST]]` 事件。
- 发送响应到终端用户。

4.入口脚本接收应用主体传来的退出状态并完成请求的处理。

应用组件

应用主体是[服务定位器](#)，它部署一组提供各种不同功能的应用组件来处理请求。例如，`urlManager` 组件负责处理网页请求路由到对应的控制器。`db` 组件提供数据库相关服务等等。

在同一个应用中，每个应用组件都有一个独一无二的 ID 用来区分其他应用组件，你可以通过如下表达式访问应用组件。

```
\Yii::$app->componentID
```

例如，可以使用 `\Yii::$app->db` 来获取到已注册到应用的 `[[yii\db\Connection|DB connection]]`，使用 `\Yii::$app->cache` 来获取到已注册到应用的 `[[yii\caching\Cache|primary cache]]`。

第一次使用以上表达式时候会创建应用组件实例，后续再访问会返回此实例，无需再次创建。

应用组件可以是任意对象，可以在 [应用主体配置](#)配置 `[[yii\base\Application::components]]` 属性。例如：

```
[
    'components' => [
        // 使用类名注册 "cache" 组件
        'cache' => 'yii\caching\ApcCache',

        // 使用配置数组注册 "db" 组件
        'db' => [
            'class' => 'yii\db\Connection',
```

```

        'dsn' => 'mysql:host=localhost;dbname=demo',

        'username' => 'root',

        'password' => '',

    ],

    // 使用函数注册"search" 组件

    'search' => function () {

        return new app\components\SolrService;

    },

],

]

```

补充：请谨慎注册太多应用组件，应用组件就像全局变量，使用太多可能加大测试和维护的难度。一般情况下可以在需要时再创建本地组件。

引导启动组件

上面提到一个应用组件只会在第一次访问时实例化，如果处理请求过程没有访问的话就不实例化。有时你想在每个请求处理过程都实例化某个组件即便它不会被访问， 可以将该组件 ID 加入到应用主体的 `[[yii\base\Application::bootstrap|bootstrap]]` 属性中。

例如，如下的应用主体配置保证了 `log` 组件一直被加载。

```

[

    'bootstrap' => [

        // 将 log 组件 ID 加入引导让它始终载入

        'log',

    ],

    'components' => [

        'log' => [

            // "log" 组件的配置

        ],

    ],

],

```

核心应用组件

Yii 定义了一组固定 ID 和默认配置的核心组件，例如 `[[yii\web\Application::request|request]]` 组件用来收集用户请求并解析 [路由](#)；`[[yii\base\Application::db|db]]` 代表一个可以执行数据库操作的数据库连接。通过这些组件，Yii 应用主体能处理用户请求。

下面是预定义的核心应用组件列表，可以和普通应用组件一样配置和自定义它们。当你配置一个核心组件，不指定它的类名的话就会使用 Yii 默认指定的类。

- `[[yii\web\AssetManager|assetManager]]`: 管理资源包和资源发布，详情请参考 [管理资源](#) 一节。
- 注意配置该组件时必须指定组件类名和其他相关组件属性，如 `[[yii\db\Connection::dsn]]`。详情请参考 [数据访问对象](#) 一节。
- `[[yii\base\Application::errorHandler|errorHandler]]`: 处理 PHP 错误和异常，详情请参考 [错误处理](#) 一节。
- 日期使用长格式。详情请参考 [格式化输出数据](#) 一节。
- `[[yii|i18n|i18N|i18n]]`: 支持信息翻译和格式化。详情请参考 [国际化](#) 一节。
- `[[yii\log\Dispatcher|log]]`: 管理日志对象。详情请参考 [日志](#) 一节。
- `[[yii\swiftmailer\Mailer|mail]]`: 支持生成邮件结构并发送，详情请参考 [邮件](#) 一节。
- 详情请参考 [响应](#) 一节。
- 详情请参考 [请求](#) 一节。
- `[[yii\web\Session|session]]`: 代表会话信息，仅在 `[[yii\web\Application|Web applications]]` 网页应用中可用，详情请参考 [Sessions \(会话\) and Cookies](#) 一节。
- 详情请参考 [URL 解析和生成](#) 一节。
- `[[yii\web\User|user]]`: 代表认证登录用户信息，仅在 `[[yii\web\Application|Web applications]]` 网页应用中可用，详情请参考 [认证](#) 一节。
- `[[yii\web\View|view]]`: 支持渲染视图，详情请参考 [Views](#) 一节。

控制器

控制器是 MVC 模式中的一部分，是继承 `[[yii\base\Controller]]` 类的对象，负责处理请求和生成响应。

具体来说，控制器从 [应用主体](#) 接管控制后会分析请求数据并传送到 [模型](#)，传送模型结果到 [视图](#)，最后生成输出响应信息。

操作

控制器由 操作 组成，它是执行终端用户请求的最基础的单元，一个控制器可有一个或多个操作。

如下示例显示包含两个操作 **view** and **create** 的控制器 **post**:

```
namespace app\controllers;

use Yii;

use app\models\Post;

use yii\web\Controller;

use yii\web\NotFoundException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);

        if ($model === null) {
            throw new NotFoundException;
        }

        return $this->render('view', [
            'model' => $model,
        ]);
    }

    public function actionCreate()
    {
        $model = new Post;
```

```

        if ($model->load(Yii::$app->request->post()) && $model->save()) {

            return $this->redirect(['view', 'id' => $model->id]);

        } else {

            return $this->render('create', [

                'model' => $model,

            ]);

        }

    }

}

```

在操作 **view** (定义为 **actionView()** 方法)中，代码首先根据请求模型 ID 加载 **模型**，如果加载成功，会渲染名称为 **view** 的**视图**并显示，否则会抛出一个异常。

在操作 **create** (定义为 **actionCreate()** 方法)中，代码相似。先将请求数据填入**模型**，然后保存模型，如果两者都成功，会跳转到 ID 为新创建的模型的 **view** 操作，否则显示提供用户输入的 **create** 视图。

路由

终端用户通过所谓的路由寻找到操作，路由是包含以下部分的字符串：

- 模型 ID: 仅存在于控制器属于非应用的**模块**;
- 控制器 ID: 同应用（或同模块如果为模块下的控制器）下唯一标识控制器的字符串;
- 操作 ID: 同控制器下唯一标识操作的字符串。

路由使用如下格式：

ControllerID/ActionID

如果属于模块下的控制器，使用如下格式：

ModuleID/ControllerID/ActionID

如果用户的请求地址为 **http://hostname/index.php?r=site/index**，会执行 **site** 控制器的 **index** 操作。更多关于处理路由的详情请参阅 **路由** 一节。

创建控制器

在[[yii\web\Application|Web applications]]网页应用中，控制器应继承[[yii\web\Controller]] 或它的子类。同理在[[yii\console\Application|console applications]]控制台应用中，控制器继承[[yii\console\Controller]] 或它的子类。如下代码定义一个 **site** 控制器：

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}

```

控制器 ID

通常情况下，控制器用来处理请求有关的资源类型，因此控制器 ID 通常为和资源有关的名词。例如使用 `article` 作为处理文章的控制器 ID。

控制器 ID 应仅包含英文小写字母、数字、下划线、中横杠和正斜杠，例如 `article` 和 `post-comment` 是真正的控制器 ID，`article?`、`PostComment`、`admin\post` 不是控制器 ID。

控制器 ID 可包含子目录前缀，例如 `admin/article` 代表

`[[yii\base\Application::controllerNamespace|controller namespace]]` 控制器命名空间下 `admin` 子目录中 `article` 控制器。子目录前缀可为英文大小写字母、数字、下划线、正斜杠，其中正斜杠用来区分多级子目录(如 `panels/admin`)。

控制器类命名

控制器 ID 遵循以下规则衍生控制器类名：

- 将用正斜杠区分的每个单词第一个字母转为大写。注意如果控制器 ID 包含正斜杠，只将最后的正斜杠后的部分第一个字母转为大写；
- 去掉中横杠，将正斜杠替换为反斜杠；
- 增加 `Controller` 后缀；
- 在前面增加 `[[yii\base\Application::controllerNamespace|controller namespace]]` 控制器命名空间。

下面为一些示例，假设 `[[yii\base\Application::controllerNamespace|controller namespace]]` 控制器命名空间为 `app\controllers`：

- `article` 对应 `app\controllers\ArticleController`；
- `post-comment` 对应 `app\controllers\PostCommentController`；
- `admin/post-comment` 对应 `app\controllers\admin\PostCommentController`；
- `adminPanels/post-comment` 对应 `app\controllers\adminPanels\PostCommentController`。

控制器类必须能被 [自动加载](#)，所以在上面的例子中，控制器 `article` 类应在 [别名](#) 为 `@app/controllers/ArticleController.php` 的文件中定义，控制器 `admin/post2-comment` 应在

@app/controllers/admin/Post2CommentController.php 文件中。

补充: 最后一个示例 `admin/post2-comment` 表示你可以将控制器放在 `[[yii\base\Application::controllerNamespace|controller namespace]]` 控制器命名空间下的子目录中, 在你不想用 [模块](#) 的情况下给控制器分类, 这种方式很有用。

控制器部署

可通过配置 `[[yii\base\Application::controllerMap|controller map]]` 来强制上述的控制器 ID 和类名对应, 通常在使用第三方不能掌控类名的控制器上。

配置 [应用配置](#) 中的 `application configuration`, 如下所示:

```
[
    'controllerMap' => [

        // 用类名申明 "account" 控制器

        'account' => 'app\controllers\UserController',

        // 用配置数组申明 "article" 控制器

        'article' => [

            'class' => 'app\controllers\PostController',

            'enableCsrfValidation' => false,

        ],

    ],
]
```

默认控制器

每个应用有一个由 `[[yii\base\Application::defaultRoute]]` 属性指定的默认控制器; 当请求没有指定 [路由](#), 该属性值作为路由使用。对于 `[[yii\web\Application|Web applications]]` 网页应用, 它的值为 `'site'`, 对于 `[[yii\console\Application|console applications]]` 控制台应用, 它的值为 `help`, 所以 URL 为 `http://hostname/index.php` 表示由 `site` 控制器来处理。

可以在 [应用配置](#) 中修改默认控制器, 如下所示:

```
[
    'defaultRoute' => 'main',
]
```

创建操作

创建操作可简单地在控制器类中定义所谓的 操作方法 来完成，操作方法必须是以 **action** 开头的公有方法。操作方法的返回值会作为响应数据发送给终端用户，如下代码定义了两个操作 **index** 和 **hello-world**:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

    public function actionHelloWorld()
    {
        return 'Hello World';
    }
}
```

操作 ID

操作通常是用来执行资源的特定操作，因此，操作 ID 通常为动词，如 **view**, **update** 等。

操作 ID 应仅包含英文小写字母、数字、下划线和中横杠，操作 ID 中的中横杠用来分隔单词。例如 **view**, **update2**, **comment-post** 是真实的操作 ID，**view?**, **Update** 不是操作 ID。

可通过两种方式创建操作 ID，内联操作和独立操作。An inline action is 内联操作在控制器类中定义为方法；独立操作是继承[[yii\base\Action]]或它的子类的类。内联操作容易创建，在无需重用的情况下优先使用；独立操作相反，主要用于多个控制器重用，或重构为扩展。

内联操作

内联操作指的是根据我们刚描述的操作方法。

操作方法的名字是根据操作 ID 遵循如下规则衍生：

- 将每个单词的第一个字母转为大写；
- 去掉中横杠；
- 增加 `action` 前缀。

例如 `index` 转成 `actionIndex`, `hello-world` 转成 `actionHelloWorld`。

注意：操作方法的名字大小写敏感，如果方法名称为 `ActionIndex` 不会认为是操作方法，所以请求 `index` 操作会返回一个异常，也要注意操作方法必须是公有的，私有或者受保护的方法不能定义成内联操作。

因为容易创建，内联操作是最常用的操作，但是如果你计划在不同地方重用相同的操作，或者你想重新分配一个操作，需要考虑定义它为独立操作。

独立操作

独立操作通过继承 `[[yii\base\Action]]` 或它的子类来定义。例如 Yii 发布的 `[[yii\web\ViewAction]]` 和 `[[yii\web>ErrorAction]]` 都是独立操作。

要使用独立操作，需要通过控制器中覆盖 `[[yii\base\Controller::actions()]]` 方法在 action map 中申明，如下例所示：

```
public function actions()
{
    return [
        // 用类来申明"error" 操作
        'error' => 'yii\web>ErrorAction',

        // 用配置数组申明 "view" 操作
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

如上所示， `actions()` 方法返回键为操作 ID、值为对应操作类名或数组 `configurations` 的数组。和内联操作不同，独立操作 ID 可包含任意字符，只要在 `actions()` 方法中申明。

为创建一个独立操作类，需要继承 `[[yii\base\Action]]` 或它的子类，并实现公有的名称为 `run()` 的方法，`run()` 方法的角色和操作方法类似，例如：

```
<?php
namespace app\components;

use yii\base\Action;

class HelloWorldAction extends Action
{
    public function run()
    {
        return "Hello World";
    }
}
```

操作结果

操作方法或独立操作的 `run()` 方法的返回值非常重要，它表示对应操作结果。

返回值可为 [响应](#) 对象，作为响应发送给终端用户。

- 对于 `[[yii\web\Application|Web applications]]` 网页应用，返回值可为任意数据，它赋值给 `[[yii\web\Response::data]]`，最终转换为字符串来展示响应内容。
- 对于 `[[yii\console\Application|console applications]]` 控制台应用，返回值可为整数，表示命令行下执行的 `[[yii\console\Response::exitStatus|exit status]]` 退出状态。

在上面的例子中，操作结果都为字符串，作为响应数据发送给终端用户，下例显示一个操作通过 返回响应对象（因为 `[[yii\web\Controller::redirect()|redirect()]]` 方法返回一个响应对象）可将用户浏览器跳转到新的 URL。

```
public function actionForward()
{
    // 用户浏览器跳转到 http://example.com

    return $this->redirect('http://example.com');
}
```

操作参数

内联操作的操作方法和独立操作的 `run()` 方法可以带参数，称为操作参数。参数值从请求中获取，对于 `[[yii\web\Application|Web applications]]` 网页应用，每个操作参数的值从 `$_GET` 中获得，参数名作为键；对于 `[[yii\console\Application|console applications]]` 控制台应用，操作参数对应命令行参数。

如下例，操作 `view` (内联操作) 申明了两个参数 `$id` 和 `$version`。

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public function actionView($id, $version = null)
    {
        // ...
    }
}
```

操作参数会被不同的参数填入，如下所示：

- `http://hostname/index.php?r=post/view&id=123`: `$id` 会填入 '123'，`$version` 仍为 `null` 空因为没有 `version` 请求参数；
- `http://hostname/index.php?r=post/view&id=123&version=2`: `$id` 和 `$version` 分别填入 '123' 和 '2'；
- `http://hostname/index.php?r=post/view`: 会抛出 `[[yii\web\BadRequestHttpException]]` 异常 因为请求没有提供参数给必须赋值参数 `$id`；
- `http://hostname/index.php?r=post/view&id[]=123`: 会抛出 `[[yii\web\BadRequestHttpException]]` 异常 因为 `$id` 参数收到数字值 ['123'] 而不是字符串。

如果想让操作参数接收数组值，需要指定 `$id` 为 `array`，如下所示：

```
public function actionView(array $id, $version = null)
{
    // ...
}
```

现在如果请求为 `http://hostname/index.php?r=post/view&id[]=123`，参数 `$id` 会使用数组值 ['123']，如果请求为 `http://hostname/index.php?r=post/view&id=123`，参数 `$id` 会获取相同数组值，因为无类型的 '123' 会自动转成数组。

上述例子主要描述网页应用的操作参数，对于控制台应用，更多详情请参阅[控制台命令](#)。

默认操作

每个控制器都有一个由 `[[yii\base\Controller::defaultAction]]` 属性指定的默认操作，当[路由](#)只包含控制器 ID，会使用所请求的控制器的默认操作。

默认操作默认为 `index`，如果想修改默认操作，只需简单地在控制器类中覆盖这个属性，如下所示：

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $defaultAction = 'home';

    public function actionHome()
    {
        return $this->render('home');
    }
}
```

控制器生命周期

处理一个请求时，[应用主体](#)会根据请求[路由](#)创建一个控制器，控制器经过以下生命周期来完成请求：

- 1.在控制器创建和配置后，`[[yii\base\Controller::init()]]`方法会被调用。
- 2.控制器根据请求操作 ID 创建一个操作对象：
 - 如果操作 ID 没有指定，会使用`[[yii\base\Controller::defaultAction|default action ID]]`默认操作 ID；
 - 如果在`[[yii\base\Controller::actions()|action map]]`找到操作 ID，会创建一个独立操作；
 - 如果操作 ID 对应操作方法，会创建一个内联操作；
 - 否则会抛出`[[yii\base\InvalidRouteException]]`异常。
- 3.控制器按顺序调用应用主体、模块（如果控制器属于模块）、控制器的 `beforeAction()` 方法；
 - 如果任意一个调用返回 `false`，后面未调用的 `beforeAction()` 会跳过并且操作执行会被取

消; action execution will be cancelled.

- 默认情况下每个 `beforeAction()` 方法会触发一个 `beforeAction` 事件, 在事件中你可以追加事件处理操作;

4.控制器执行操作:

- 请求数据解析和填入到操作参数;

5.控制器按顺序调用控制器、模块 (如果控制器属于模块)、应用主体的 `afterAction()` 方法;

- 默认情况下每个 `afterAction()` 方法会触发一个 `afterAction` 事件, 在事件中你可以追加事件处理操作;

6.应用主体获取操作结果并赋值给[响应](#).

最佳实践

在设计良好的应用中, 控制器很精练, 包含的操作代码简短; 如果你的控制器很复杂, 通常意味着需要重构, 转移一些代码到其他类中。

归纳起来, 控制器

- 可访问 [请求](#) 数据;
- 可根据请求数据调用 [模型](#) 的方法和其他服务组件;
- 可使用 [视图](#) 构造响应;
- 不应处理应被[模型](#)处理的请求数据;
- 应避免嵌入 HTML 或其他展示代码, 这些代码最好在 [视图](#)中处理。

视图

视图是 [MVC](#) 模式中的一部分。它是展示数据到终端用户的代码, 在网页应用中, 根据视图模板来创建视图, 视图模板为 PHP 脚本文件, 主要包含 HTML 代码和展示类 PHP 代码, 通过 `[[yii\web\View|view]]` 应用组件来管理, 该组件主要提供通用方法帮助视图构造和渲染, 简单起见, 我们称视图模板或视图模板文件为视图。

创建视图

如前所述, 视图为包含 HTML 和 PHP 代码的 PHP 脚本, 如下代码为一个登录表单的视图, 可看到 PHP 代码用来生成动态内容如页面标题和表单, HTML 代码把它组织成一个漂亮的 HTML 页面。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
```

```

/* @var $form yii\widgets\ActiveForm */

/* @var $model app\models\LoginForm */

$this->title = 'Login';

?>

<h1><?= Html::encode($this->title) ?></h1>

<p>Please fill out the following fields to login:</p>

<?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <?= Html::submitButton('Login') ?>

<?php ActiveForm::end(); ?>

```

在视图中，可访问 `$this` 指向 `[[yii\web\View|view component]]` 来管理和渲染这个视图文件。

除了 `$this` 之外，上述示例中的视图有其他预定义变量如 `$model`，这些变量代表从[控制器](#)或其他触发[视图渲染](#)的对象 传入 到视图的数据。

技巧: 将预定义变量列到视图文件头部注释处，这样可被 IDE 编辑器识别，也是生成视图文档的好方法。

安全

当创建生成 HTML 页面的视图时，在显示之前将用户输入数据进行转码和过滤非常重要，否则，你的应用可能会被[跨站脚本](#) 攻击。

要显示纯文本，先调用 `[[yii\helpers\Html::encode()]]` 进行转码，例如如下代码将用户名在显示前先转码：

```

<?php
use yii\helpers\Html;

?>

<div class="username">

    <?= Html::encode($user->name) ?>

```

</div>

要显示 HTML 内容，先调用 `[[yii\helpers\HtmlPurifier]]` 过滤内容，例如如下代码将提交内容在显示前先过滤：

```
<?php
use yii\helpers\HtmlPurifier;
?>

<div class="post">

    <?= HtmlPurifier::process($post->text) ?>

</div>
```

技巧：HTMLPurifier 在保证输出数据安全上做的不错，但性能不佳，如果你的应用需要高性能可考虑[缓存](#) 过滤后的结果。

组织视图

与 [控制器](#) 和 [模型](#) 类似，在组织视图上有一些约定：

- 控制器渲染的视图文件默认放在 `@app/views/ControllerID` 目录下，其中 `ControllerID` 对应 [控制器 ID](#)，例如控制器类为 `PostController`，视图文件目录应为 `@app/views/post`，控制器类 `PostCommentController` 对应的目录为 `@app/views/post-comment`，如果是模块中的控制器，目录应为 `[[yii\base\Module::basePath|module directory]]` 模块目录下的 `views/ControllerID` 目录；
- 对于 [小部件](#) 渲染的视图文件默认放在 `WidgetPath/views` 目录，其中 `WidgetPath` 代表小部件类文件所在的目录；
- 对于其他对象渲染的视图文件，建议遵循和小部件相似的规则。

可覆盖控制器或小部件的 `[[yii\base\ViewContextInterface::getViewPath()]]` 方法来自定义视图文件默认目录。

渲染视图

可在 [控制器](#)，[小部件](#)，或其他地方调用渲染视图方法来渲染视图，该方法类似以下格式：

```
/**
 * @param string $view 视图名或文件路径，由实际的渲染方法决定
 * @param array $params 传递给视图的数据
 * @return string 渲染结果
```

```
*/
```

```
methodName($view, $params = [])
```

控制器中渲染

在 [控制器](#) 中，可调用以下控制器方法来渲染视图：

- `[[yii\base\Controller::render()|render()]]`: 渲染一个 [视图名](#) 并使用一个 [布局](#) 返回到渲染结果。
- `[[yii\base\Controller::renderPartial()|renderPartial()]]`: 渲染一个 [视图名](#) 并且不使用布局。
- `[[yii\web\Controller::renderAjax()|renderAjax()]]`: 渲染一个 [视图名](#) 并且不使用布局，并注入所有注册的 JS/CSS 脚本和文件，通常使用在响应 [AJAX](#) 网页请求的情况下。
-

例如：

```
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);

        if ($model === null) {
            throw new NotFoundHttpException;
        }

        // 渲染一个名称为"view"的视图并使用布局

        return $this->render('view', [
```



```
        'model' => $model,  
  
    });  
  
}  
  
}
```

小部件中渲染

在 [小部件](#) 中，可调用以下小部件方法来渲染视图： Within [widgets](#), you may call the following widget methods to render views.

- `[[yii\base\Widget::render()|render()]]`: 渲染一个 [视图名](#).
-

例如：

```
namespace app\components;  
  
use yii\base\Widget;  
use yii\helpers\Html;  
  
class ListWidget extends Widget  
{  
  
    public $items = [];  
  
    public function run()  
    {  
  
        // 渲染一个名为 "list" 的视图  
  
        return $this->render('list', [  
  
            'items' => $this->items,  
  
        ]);  
    }  
}
```

视图中渲染

可以在视图中渲染另一个视图，可以调用[[yii\base\View|view component]]视图组件提供的以下方法：

- [[yii\base\View::render()|render()]]: 渲染一个 视图名。
- [[yii\web\View::renderAjax()|renderAjax()]]: 渲染一个 视图名 并注入所有注册的 JS/CSS 脚本和文件，通常使用在响应 AJAX 网页请求的情况下。
-

例如，视图中的如下代码会渲染该视图所在目录下的 `_overview.php` 视图文件，记住视图中 `$this` 对应 [[yii\base\View|view]] 组件：

```
<?= $this->render('_overview') ?>
```

其他地方渲染

在任何地方都可以通过表达式 `Yii::$app->view` 访问 [[yii\base\View|view]] 应用组件，调用它的前所述的方法渲染视图，例如：

```
// 显示视图文件 "@app/views/site/license.php"
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

视图名

渲染视图时，可指定一个视图名或视图文件路径/别名，大多数情况下使用前者因为前者简洁灵活，我们称用名字的视图为 视图名。

视图名可以依据以下规则到对应的视图文件路径：

- 视图名可省略文件扩展名，这种情况下使用 `.php` 作为扩展，视图名 `about` 对应到 `about.php` 文件名；
- 视图名以双斜杠 `//` 开头，对应的视图文件路径为 `@app/views/ViewName`，也就是说视图文件在 [[yii\base\Application::viewPath|application's view path]] 路径下找，例如 `//site/about` 对应到 `@app/views/site/about.php`。
- 视图名以单斜杠 `/` 开始，视图文件路径以当前使用模块的 [[yii\base\Module::viewPath|view path]] 开始，如果不存在模块，使用 `@app/views/ViewName` 开始，例如，如果当前模块为 `user`，`/user/create` 对应成 `@app/modules/user/views/user/create.php`，如果不在模块中，`/user/create` 对应 `@app/views/user/create.php`。
- 如果 [[yii\base\View::context|context]] 渲染视图 并且上下文实现了 [[yii\base\ViewContextInterface]]，视图文件路径由上下文的 [[yii\base\ViewContextInterface::getViewPath()|view path]] 开始，这种主要用在控制器和小部件中渲染视图，例如 如果上下文为控制器 `SiteController`，`site/about` 对应到 `@app/views/site/about.php`。
- 如果视图渲染另一个视图，包含另一个视图文件的目录以当前视图的文件路径开始，例如被视

图 `@app/views/post/index.php` 渲染的 `item` 对应到 `@app/views/post/item`。

根据以上规则，在控制器中 `app\controllers\PostController` 调用 `$this->render('view')`，实际上渲染 `@app/views/post/view.php` 视图文件，当在该视图文件中调用 `$this->render('_overview')` 会渲染 `@app/views/post/_overview.php` 视图文件。

视图中访问数据

在视图中有两种方式访问数据：推送和拉取。

推送方式是通过视图渲染方法的第二个参数传递数据，数据格式应为名称-值的数组，视图渲染时，调用 PHP `extract()` 方法将该数组转换为视图可访问的变量。例如，如下控制器的渲染视图代码推送 2 个变量到 `report` 视图：`$foo = 1` 和 `$bar = 2`。

```
echo $this->render('report', [  
  
    'foo' => 1,  
  
    'bar' => 2,  
  
]);
```

拉取方式可让视图从 `[[yii\base\View|view component]]` 视图组件或其他对象中主动获得数据(如 `Yii::$app`)，在视图中使用如下表达式 `$this->context` 可获取到控制器 ID，可让你在 `report` 视图中获取控制器的任意属性或方法，如以下代码获取控制器 ID。

```
The controller ID is: <?= $this->context->id ?>  
?>
```

推送方式让视图更少依赖上下文对象，是视图获取数据优先使用方式，缺点是需要手动构建数组，有些繁琐，在不同地方渲染时容易出错。

视图间共享数据

`[[yii\base\View|view component]]` 视图组件提供 `[[yii\base\View::params|params]]` 参数属性来让不同视图共享数据。

例如在 `about` 视图中，可使用如下代码指定当前 `breadcrumbs` 的当前部分。

```
$this->params['breadcrumbs'][] = 'About Us';
```

在 `布局` 文件（也是一个视图）中，可使用依次加入到 `[[yii\base\View::params|params]]` 数组的值来生成显示 `breadcrumbs`：

```
<?= yii\widgets\Breadcrumbs::widget([  
  
    'links' => isset($this->params['breadcrumbs']) ? $this->params['breadcrumbs'] : [],  
  
]) ?>
```

布局

布局是一种特殊的视图，代表多个视图的公共部分，例如，大多数 **Web** 应用共享相同的页头和页尾，在每个视图中重复相同的页头和页尾，更好的方式是把这些公共放到一个布局中，渲染内容视图后在合适的地方嵌入到布局中。

创建布局

由于布局也是视图，它可像普通视图一样创建，布局默认存储在 `@app/views/layouts` 路径下，[模块](#)中使用的布局应存储在 `[[yii\base\Module::basePath|module directory]]` 模块目录下的 `views/layouts` 路径下，可配置 `[[yii\base\Module::layoutPath]]` 来自定义应用或模块的布局默认路径。

如下示例为一个布局大致内容，注意作为示例，简化了很多代码，在实际中，你可能想添加更多内容，如头部标签，主菜单等。

```
<?php
use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $content string 字符串 */
?>

<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <header>My Company</header>
    <?= $content ?>
```

```
<footer>&copy; 2014 by My Company</footer>

<?php $this->endBody() ?>

</body>

</html>

<?php $this->endPage() ?>
```

如上所示，布局生成每个页面通用的 HTML 标签，在 `<body>` 标签中，打印 `$content` 变量，`$content` 变量代表当 `[[yii\base\Controller::render()]]` 控制器渲染方法调用时传递到布局的内容视图渲染结果。

大多数视图应调用上述代码中的如下方法，这些方法触发关于渲染过程的事件，这样其他地方注册的脚本和标签会添加到这些方法调用的地方。

- 它触发表明页面开始的 `[[yii\base\View::EVENT_BEGIN_PAGE|EVENT_BEGIN_PAGE]]` 事件。
- 它触发表明页面结尾的 `[[yii\base\View::EVENT_END_PAGE|EVENT_END_PAGE]]` 时间。
- 它生成一个占位符，在页面渲染结束时会被注册的头部 HTML 代码（如，link 标签，meta 标签）替换。
- 它触发 `[[yii\web\View::EVENT_BEGIN_BODY|EVENT_BEGIN_BODY]]` 事件并生成一个占位符，会被注册的 HTML 代码（如 JavaScript）在页面主体开始处替换。
- 它触发 `[[yii\web\View::EVENT_END_BODY|EVENT_END_BODY]]` 事件并生成一个占位符，会被注册的 HTML 代码（如 JavaScript）在页面主体结尾处替换。

布局中访问数据

在布局中可访问两个预定义变量：`$this` 和 `$content`，前者对应和普通视图类似的

`[[yii\base\View|view]]` 视图组件 后者包含调用 `[[yii\base\Controller::render()|render()]]` 方法渲染内容视图的结果。

如果想在布局中访问其他数据，必须使用[视图中访问数据](#)一节介绍的拉取方式，如果想从内容视图中传递数据到布局，可使用[视图间共享数据](#)一节中的方法。

使用布局

如[控制器中渲染](#)一节描述，当控制器调用 `[[yii\base\Controller::render()|render()]]` 方法渲染视图时，会同时使用布局到渲染结果中，默认会使用 `@app/views/layouts/main.php` 布局文件。

可配置 `[[yii\base\Application::layout]]` 或 `[[yii\base\Controller::layout]]` 使用其他布局文件，前者管理所有控制器的布局，后者覆盖前者来控制单个控制器布局。例如，如下代码使 `post` 控制器渲染视图时使用 `@app/views/layouts/post.php` 作为布局文件，假如 `layout` 属性没改变，控制器默认使用 `@app/views/layouts/main.php` 作为布局文件。

```
namespace app\controllers;
```

```
use yii\web\Controller;
```

```
class PostController extends Controller
{
    public $layout = 'post';

    // ...
}
```

对于模块中的控制器，可配置模块的 `[[yii\base\Module::layout|layout]]` 属性指定布局文件应用到模块的所有控制器。

由于 `layout` 可在不同层级（控制器、模块、应用）配置，在幕后 Yii 使用两步来决定控制器实际使用的布局。

第一步，它决定布局的值和上下文模块：

- 如果控制器的 `[[yii\base\Controller::layout|layout]]` 属性不为空 `null`，使用它作为布局的值，控制器的 `[[yii\base\Controller::module|module]]` 模块 作为上下文模块。
- 如果 `[[yii\base\Controller::layout|layout]]` 为空，从控制器的祖先模块（包括应用） 开始找 第一个 `[[yii\base\Module::layout|layout]]` 属性不为空的模块，使用该模块作为上下文模块，并将它的 `[[yii\base\Module::layout|layout]]` 的值作为布局的值， 如果都没有找到，表示不使用布局。

第二步，它决定第一步中布局的值和上下文模块对应到实际的布局文件，布局的值可为：

- 路径别名（如 `@app/views/layouts/main`）。
- 绝对路径（如 `/main`）：布局的值以斜杠开始，在应用的 `[[yii\base\Application::layoutPath|layout path]]` 布局路径 中查找实际的布局文件，布局路径默认为 `@app/views/layouts`。
- 相对路径（如 `main`）：在上下文模块的 `[[yii\base\Module::layoutPath|layout path]]` 布局路径中查找实际的布局文件， 布局路径默认为 `[[yii\base\Module::basePath|module directory]]` 模块目录下的 `views/layouts` 目录。
- 布尔值 `false`：不使用布局。

布局的值没有包含文件扩展名，默认使用 `.php` 作为扩展名。

嵌套布局

有时候你想嵌套一个布局到另一个，例如，在 Web 站点不同地方，想使用不同的布局， 同时这些布局共享相同的生成全局 HTML5 页面结构的基本布局，可以在子布局中调用

`[[yii\base\View::beginContent()|beginContent()]]` 和 `[[yii\base\View::endContent()|endContent()]]` 方法，如下所示：

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>
```

```
...child layout content here...
```

```
<?php $this->endContent(); ?>
```

如上所示，子布局内容应在 `[[yii\base\View::beginContent()|beginContent()]]` 和 `[[yii\base\View::endContent()|endContent()]]` 方法之间，传给 `[[yii\base\View::beginContent()|beginContent()]]` 的参数指定父布局，父布局可为布局文件或别名。

使用以上方式可多层嵌套布局。

使用数据块

数据块可以在一个地方指定视图内容在另一个地方显示，通常和布局一起使用，例如，可在内容视图中定义数据块在布局中显示它。

调用 `[[yii\base\View::beginBlock()|beginBlock()]]` 和 `[[yii\base\View::endBlock()|endBlock()]]` 来定义数据块，使用 `$view->blocks[$blockID]` 访问该数据块，其中 `$blockID` 为定义数据块时指定的唯一标识 ID。

如下实例显示如何在内容视图使用数据块让布局使用。

首先，在内容视图中定一个或多个数据块：

```
...
```

```
<?php $this->beginBlock('block1'); ?>
```

```
...content of block1...
```

```
<?php $this->endBlock(); ?>
```

```
...
```

```
<?php $this->beginBlock('block3'); ?>
```

```
...content of block3...
```

```
<?php $this->endBlock(); ?>
```

然后，在布局视图中，数据块可用的话会渲染数据块，如果数据未定义则显示一些默认内容。

```

...

<?php if (isset($this->blocks['block1'])): ?>

    <?= $this->blocks['block1'] ?>

<?php else: ?>

    ... default content for block1 ...

<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>

    <?= $this->blocks['block2'] ?>

<?php else: ?>

    ... default content for block2 ...

<?php endif; ?>

...

<?php if (isset($this->blocks['block3'])): ?>

    <?= $this->blocks['block3'] ?>

<?php else: ?>

    ... default content for block3 ...

<?php endif; ?>

...

```

使用视图组件

[[yii\base\View|View components]]视图组件提供许多视图相关特性，可创建[[yii\base\View]]或它的子类实例来获取视图组件，大多数情况下主要使用 **view** 应用组件，可在[应用配置](#)中配置该组件，如下所示：

```

[
    // ...

```



```
'components' => [

    'view' => [

        'class' => 'app\components\View',

    ],

    // ...

],

]
```

视图组件提供如下实用的视图相关特性，每项详情会在独立章节中介绍：

- **主题**：允许为你的 Web 站点开发和修改主题；
- **片段缓存**：允许你在 Web 页面中缓存片段；
- **客户脚本处理**：支持 CSS 和 JavaScript 注册和渲染；
- **资源包处理**：支持 **资源包** 的注册和渲染；
- **模板引擎**：允许你使用其他模板引擎，如 **Twig**, **Smarty**。

开发 Web 页面时，也可能频繁使用以下实用的小特性。

设置页面标题

每个 Web 页面应有一个标题，正常情况下标题的标签显示在 **布局** 中，但是实际上标题大多由内容视图而不是布局来决定，为解决这个问题，`[[yii\web\View]]` 提供 `[[yii\web\View::title|title]]` 标题属性可让标题信息从内容视图传递到布局中。

为利用这个特性，在每个内容视图中设置页面标题，如下所示：

```
<?php
$this->title = 'My page title';
?>
```

然后在视图中，确保在 `<head>` 段中有如下代码：

```
<title><?= Html::encode($this->title) ?></title>
```

注册 Meta 元标签

Web 页面通常需要生成各种元标签提供给不同的浏览器，如 `<head>` 中的页面标题，元标签通常在布局中生成。

如果想在内容视图中生成元标签，可在内容视图中调用 `[[yii\web\View::registerMetaTag()]]` 方法，如下所示：

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework, php']);
?>
```

以上代码会在视图组件中注册一个 "keywords" 元标签，在布局渲染后会渲染该注册的元标签，然后，如下 HTML 代码会插入到布局中调用[[yii\web\View::head()]]方法处：

```
<meta name="keywords" content="yii, framework, php">
```

注意如果多次调用 [[yii\web\View::registerMetaTag()]] 方法，它会注册多个元标签，注册时不会检查是否重复。

为确保每种元标签只有一个，可在调用方法时指定键作为第二个参数，例如，如下代码注册两次 "description" 元标签，但是只会渲染第二个。

```
$this->registerMetaTag(['name' => 'description', 'content' => 'This is my cool website made with Yii!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => 'This website is about funny raccoons.'], 'description');
```

注册链接标签

和 [Meta 标签](#) 类似，链接标签有时很实用，如自定义网站图标，指定 Rss 订阅，或授权 OpenID 到其他服务器。可以和元标签相似的方式调用[[yii\web\View::registerLinkTag()]]，例如，在内容视图中注册链接标签如下所示：

```
$this->registerLinkTag([
    'title' => 'Live News for Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'http://www.yiiframework.com/rss.xml/',
]);
```

上述代码会转换成

```
<link title="Live News for Yii" rel="alternate" type="application/rss+xml"
href="http://www.yiiframework.com/rss.xml/">
```

和 [[yii\web\View::registerMetaTag()|registerMetaTags()]] 类似，调用 [[yii\web\View::registerLinkTag()|registerLinkTag()]] 指定键来避免生成重复链接标签。

视图事件

[[yii\base\View|View components]] 视图组件会在视图渲染过程中触发几个事件，可以在内容发送

给终端用户前，响应这些事件来添加内容到视图中或调整渲染结果。

- 该事件可设置 `[[yii\base\ViewEvent::isValid]]` 为 `false` 取消视图渲染。
- `[[yii\base\View::EVENT_AFTER_RENDER|EVENT_AFTER_RENDER]]`: 在布局中调用 `[[yii\base\View::beginPage()]]` 时触发，该事件可获取 `[[yii\base\ViewEvent::output]]` 的渲染结果，可修改该属性来修改渲染结果。
- `[[yii\base\View::EVENT_BEGIN_PAGE|EVENT_BEGIN_PAGE]]`: 在布局调用 `[[yii\base\View::beginPage()]]` 时触发；
- `[[yii\base\View::EVENT_END_PAGE|EVENT_END_PAGE]]`: 在布局调用 `[[yii\base\View::endPage()]]` 是触发；
- `[[yii\web\View::EVENT_BEGIN_BODY|EVENT_BEGIN_BODY]]`: 在布局调用 `[[yii\web\View::beginBody()]]` 时触发；
- `[[yii\web\View::EVENT_END_BODY|EVENT_END_BODY]]`: 在布局调用 `[[yii\web\View::endBody()]]` 时触发。

例如，如下代码将当前日期添加到页面结尾处：

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {  
  
    echo date('Y-m-d');  
  
});
```

渲染静态页面

静态页面指的是大部分内容为静态的不需要控制器传递动态数据的 **Web** 页面。

可将 **HTML** 代码放置在视图中，在控制器中使用以下代码输出静态页面：

```
public function actionAbout()  
{  
  
    return $this->render('about');  
  
}
```

如果 **Web** 站点包含很多静态页面，多次重复相似的代码显得很繁琐，为解决这个问题，可以使用一个在控制器中称为 `[[yii\web\ViewAction]]` 的[独立操作](#)。例如：

```
namespace app\controllers;  
  
use yii\web\Controller;  
  
class SiteController extends Controller  
{
```

```
public function actions()
{
    return [
        'page' => [
            'class' => 'yii\web\ViewAction',
        ],
    ];
}
```

现在如果你在 `@app/views/site/pages` 目录下创建名为 `about` 的视图， 可通过如下 `url` 显示该视图：

`http://localhost/index.php?r=site/page&view=about`

`GET` 中 `view` 参数告知 `[[yii\web\ViewAction]]` 操作请求哪个视图，然后操作在 `@app/views/site/pages` 目录下寻找该视图，可配置 `[[yii\web\ViewAction::viewPrefix]]` 修改搜索视图的目录。

最佳实践

视图负责将模型的数据展示用户想要的格式，总之，视图

- 应主要包含展示代码，如 `HTML`，和简单的 `PHP` 代码来控制、格式化和渲染数据；
- 不应包含执行数据查询代码，这种代码放在模型中；
- 应避免直接访问请求数据，如 `$_GET`，`$_POST`，这种应在控制器中执行， 如果需要请求数据，应由控制器推送到视图。
- 可读取模型属性，但不应修改它们。

为使模型更易于维护，避免创建太复杂或包含太多冗余代码的视图，可遵循以下方法达到这个目标：

- 使用 `布局` 来展示公共代码（如，页面头部、尾部）；
- 将复杂的视图分成几个小视图，可使用上面描述的渲染方法将这些小视图渲染并组装成大视图；
- 创建并使用 `小部件` 作为视图的数据块；
- 创建并使用助手类在视图中转换和格式化数据。

模型

模型是 `MVC` 模式中的一部分， 是代表业务数据、规则和逻辑的对象。

可通过继承 `[[yii\base\Model]]` 或它的子类定义模型类，基类`[[yii\base\Model]]`支持许多实用的特性：

- **属性**：代表可像普通类属性或数组一样被访问的业务数据；
- **属性标签**：指定属性显示出来的标签；
- **块赋值**：支持一步给许多属性赋值；
- **验证规则**：确保输入数据符合所声明的验证规则；
- **数据导出**：允许模型数据导出为自定义格式的数组。

Model 类也是更多高级模型如 **Active Record 活动记录** 的基类， 更多关于这些高级模型的详情请参考相关手册。

补充：模型并不强制一定要继承`[[yii\base\Model]]`，但是由于很多组件支持`[[yii\base\Model]]`，最好使用它做为模型基类。

属性

模型通过 属性 来代表业务数据，每个属性像是模型的公有可访问属性，`[[yii\base\Model::attributes()]]` 指定模型所拥有的属性。

可像访问一个对象属性一样访问模型的属性：

```
$model = new \app\models\ContactForm;
```

```
// "name" 是 ContactForm 模型的属性
```

```
$model->name = 'example';
```

```
echo $model->name;
```

也可像访问数组单元项一样访问属性，这要感谢`[[yii\base\Model]]`支持 **ArrayAccess 数组访问** 和 **ArrayIterator 数组迭代器**：

```
$model = new \app\models\ContactForm;
```

```
// 像访问数组单元项一样访问属性
```

```
$model['name'] = 'example';
```

```
echo $model['name'];
```

```
// 迭代器遍历模型
```

```
foreach ($model as $name => $value) {
```

```
    echo "$name: $value\n";
```

```
}
```

定义属性

默认情况下你的模型类直接从[[yii\base\Model]]继承，所有 `non-static public` 非静态公有 成员变量都是属性。 例如，下述 `ContactForm` 模型类有四个属性

`name`, `email`, `subject` and `body`, `ContactForm` 模型用来代表从 HTML 表单获取的输入数据。

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;

    public $email;

    public $subject;

    public $body;
}
```

另一种方式是可覆盖 [[yii\base\Model::attributes()]] 来定义属性，该方法返回模型的属性名。 例如 [[yii\db\ActiveRecord]] 返回对应数据表列名作为它的属性名， 注意可能需要覆盖魔术方法如

`__get()`, `__set()`使属性像普通对象属性被访问。

属性标签

当属性显示或获取输入时，经常要显示属性相关标签，例如假定一个属性名为 `firstName`， 在某些地方如表单输入或错误信息处，你可能想显示对终端用户来说更友好的 `First Name` 标签。

可以调用 [[yii\base\Model::getAttributeLabel()]] 获取属性的标签，例如：

```
$model = new \app\models\ContactForm;

// 显示为 "Name"

echo $model->getAttributeLabel('name');
```

默认情况下，属性标签通过[[yii\base\Model::generateAttributeLabel()]]方法自动从属性名生成. 它会自动将驼峰式大小写变量名转换为多个首字母大写的单词，例如 `username` 转换为 `Username`， `firstName` 转换为 `First Name`。

如果你不想用自动生成的标签，可以覆盖 `[[yii\base\Model::attributeLabels()]]` 方法明确指定属性标签，例如：

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;

    public $email;

    public $subject;

    public $body;

    public function attributeLabels()
    {
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
            'body' => 'Content',
        ];
    }
}
```

应用支持多语言的情况下，可翻译属性标签，可在 `[[yii\base\Model::attributeLabels()|attributeLabels()]]` 方法中定义，如下所示：

```
public function attributeLabels()
{
    return [
        'name' => \Yii::t('app', 'Your name'),
    ];
}
```

```

        'email' => \Yii::t('app', 'Your email address'),

        'subject' => \Yii::t('app', 'Subject'),

        'body' => \Yii::t('app', 'Content'),

    ];
}

```

甚至可以根据条件定义标签，例如通过使用模型的 [scenario 场景](#)，可对相同的属性返回不同的标签。

补充：属性标签是 [视图](#) 一部分，但是在模型中申明标签通常非常方便，并可行程非常简洁可重用代码。

场景

模型可能在多个 场景 下使用，例如 **User** 模块可能会在收集用户登录输入，也可能会在用户注册时使用。在不同的场景下，模型可能会使用不同的业务规则和逻辑，例如 **email** 属性在注册时强制要求有，但在登陆时不需要。

模型使用 `[[yii\base\Model::scenario]]` 属性保持使用场景的跟踪，默认情况下，模型支持一个名为 **default** 的场景，如下展示两种设置场景的方法：

```

// 场景作为属性来设置

$model = new User;

$model->scenario = 'login';

// 场景通过构造初始化配置来设置

$model = new User(['scenario' => 'login']);

```

默认情况下，模型支持的场景由模型中申明的 [验证规则](#) 来决定，但你可以通过覆盖 `[[yii\base\Model::scenarios()]]` 方法来自定义行为，如下所示：

```

namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {

```



```

return [

    'login' => ['username', 'password'],

    'register' => ['username', 'email', 'password'],

];

}
}

```

补充：在上述和下述的例子中，模型类都是继承[[yii\db\ActiveRecord]]，因为多场景的使用通常发生在 [Active Record](#) 类中。

`scenarios()` 方法返回一个数组，数组的键为场景名，值为对应的 **active attributes** 活动属性。活动属性可被 [块赋值](#) 并遵循[验证规则](#)在上述例子中，`username` 和 `password` 在 `login` 场景中启用，在 `register` 场景中，除了 `username` and `password` 外 `email` 也被启用。

`scenarios()` 方法默认实现会返回所有[[yii\base\Model::rules()]]方法申明的验证规则中的场景，当覆盖 `scenarios()` 时，如果你想在默认场景外使用新场景，可以编写类似如下代码：

```

namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        $scenarios = parent::scenarios();

        $scenarios['login'] = ['username', 'password'];

        $scenarios['register'] = ['username', 'email', 'password'];

        return $scenarios;
    }
}

```

场景特性主要在[验证](#) 和 [属性块赋值](#) 中使用。你也可以用于其他目的，例如可基于不同的场景定义不同的 [属性标签](#)。

验证规则

当模型接收到终端用户输入的数据，数据应当满足某种规则(称为 验证规则, 也称为 业务规则)。例如假定 **ContactForm** 模型，你可能想确保所有属性不为空且 **email** 属性包含一个有效的邮箱地址，如果某个属性的值不满足对应的业务规则，相应的错误信息应显示，以帮助用户修正错误。

可调用 `[[yii\base\Model::validate()]]` 来验证接收到的数据，该方法使用 `[[yii\base\Model::rules()]]` 申明的验证规则来验证每个相关属性，如果没有找到错误，会返回 **true**，否则它会将错误保存在 `[[yii\base\Model::errors]]` 属性中并返回 **false**，例如：

```
$model = new \app\models\ContactForm;

// 用户输入数据赋值到模型属性

$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {

    // 所有输入数据都有效 all inputs are valid

} else {

    // 验证失败: $errors 是一个包含错误信息的数组

    $errors = $model->errors;

}
```

通过覆盖 `[[yii\base\Model::rules()]]` 方法指定模型属性应该满足的规则来申明模型相关验证规则。下述例子显示 **ContactForm** 模型申明的验证规则：

```
public function rules()

{

    return [

        // name, email, subject 和 body 属性必须有值

        [['name', 'email', 'subject', 'body'], 'required'],

        // email 属性必须是一个有效的电子邮箱地址

        ['email', 'email'],

    ];

}
```

一条规则可用来验证一个或多个属性，一个属性可对应一条或多条规则。更多关于如何申明验证规则的详情请参考 [验证输入](#) 一节。

有时你想一条规则只在某个 **场景** 下应用，为此你可以指定规则的 **on** 属性，如下所示：

```
public function rules()
{
    return [

        // 在"register" 场景下 username, email 和 password 必须有值

        [['username', 'email', 'password'], 'required', 'on' => 'register'],

        // 在 "login" 场景下 username 和 password 必须有值

        [['username', 'password'], 'required', 'on' => 'login'],

    ];
}
```

如果没有指定 **on** 属性，规则会在所有场景下应用， 在当前`[[yii\base\Model::scenario|scenario]]` 下应用的规则称之为 **active rule** 活动规则。

一个属性只会属于 `scenarios()` 中定义的活动属性且在 `rules()` 申明对应一条或多条活动规则的情况下被验证。

块赋值

块赋值只用一行代码将用户所有输入填充到一个模型，非常方便， 它直接将输入数据对应填充到 `[[yii\base\Model::attributes]]` 属性。 以下两段代码效果是相同的，都是将终端用户输入的表单数据赋值到 `ContactForm` 模型的属性， 明显地前一段块赋值的代码比后一段代码简洁且不易出错。

```
$model = new \app\models>ContactForm;

$model->attributes = \Yii::$app->request->post('ContactForm');

$model = new \app\models>ContactForm;

$data = \Yii::$app->request->post('ContactForm', []);

$model->name = isset($data['name']) ? $data['name'] : null;

$model->email = isset($data['email']) ? $data['email'] : null;

$model->subject = isset($data['subject']) ? $data['subject'] : null;

$model->body = isset($data['body']) ? $data['body'] : null;
```

安全属性

块赋值只应用在模型当前`[[yii\base\Model::scenario|scenario]]`场景

`[[yii\base\Model::scenarios()]]`方法 列出的称之为 安全属性 的属性上，例如，如果 **User** 模型申明以下场景， 当当前场景为 **login** 时候，只有 **username** and **password** 可被块赋值，其他属性不会被赋值。

```
public function scenarios()
{
    return [
        'login' => ['username', 'password'],
        'register' => ['username', 'email', 'password'],
    ];
}
```

补充: 块赋值只应用在安全属性上，因为你想控制哪些属性会被终端用户输入数据所修改， 例如，如果 **User** 模型有一个 **permission** 属性对应用户的权限， 你可能只想让这个属性在后台界面被管理员修改。

由于默认`[[yii\base\Model::scenarios()]]`的实现会返回`[[yii\base\Model::rules()]]`所有属性和数据，如果不覆盖这个方法，表示所有只要出现在活动验证规则中的属性都是安全的。

为此，提供一个特别的别名为 **safe** 的验证器来申明哪些属性是安全的不需要被验证， 如下示例的规则申明 **title** 和 **description** 都为安全属性。

```
public function rules()
{
    return [
        [['title', 'description'], 'safe'],
    ];
}
```

非安全属性

如上所述，`[[yii\base\Model::scenarios()]]` 方法提供两个用处：定义哪些属性应被验证，定义哪些属性安全。 在某些情况下，你可能想验证一个属性但不想让他是安全的，可在 `scenarios()` 方法中属性名加一个惊叹号 **!**。 例如像如下的 **secret** 属性。

```
public function scenarios()
{
```

```
return [  
  
    'login' => ['username', 'password', '!secret'],  
  
];  
}
```

当模型在 `login` 场景下，三个属性都会被验证，但只有 `username` 和 `password` 属性会被块赋值，要对 `secret` 属性赋值，必须像如下例子明确对它赋值。

```
$model->secret = $secret;
```

数据导出

模型通常要导出成不同格式，例如，你可能想将模型的一个集合转成 JSON 或 Excel 格式，导出过程可分解为两个步骤，第一步，模型转换成数组；第二步，数组转换成所需要的格式。你只需要关注第一步，因为第二步可被通用的数据转换器如 `[[yii\web\JsonResponseFormatter]]` 来完成。

将模型转换为数组最简单的方式是使用 `[[yii\base\Model::attributes]]` 属性，例如：

```
$post = \app\models\Post::findOne(100);  
$array = $post->attributes;
```

`[[yii\base\Model::attributes]]` 属性会返回 所有 `[[yii\base\Model::attributes()]]` 申明的属性的值。

更灵活和强大的将模型转换为数组的方式是使用 `[[yii\base\Model::toArray()]]` 方法，它的行为默认和 `[[yii\base\Model::attributes]]` 相同，但是它允许你选择哪些称之为字段的数据项放入到结果数组中并同时被格式化。实际上，它是导出模型到 RESTful 网页服务开发的默认方法，详情请参阅[响应格式](#)。

字段

字段是模型通过调用 `[[yii\base\Model::toArray()]]` 生成的数组的单元名。

默认情况下，字段名对应属性名，但是你可以通过覆盖 `[[yii\base\Model::fields()|fields()]]` 和/或 `[[yii\base\Model::extraFields()|extraFields()]]` 方法来改变这种行为，两个方法都返回一个字段定义列表，`fields()` 方法定义的字段是默认字段，表示 `toArray()` 方法默认会返回这些字段。`extraFields()` 方法定义额外可用字段，通过 `toArray()` 方法指定 `$expand` 参数来返回这些额外可用字段。例如如下代码会返回 `fields()` 方法定义的所有字段和 `extraFields()` 方法定义的 `prettyName` and `fullAddress` 字段。

```
$array = $model->toArray([], ['prettyName', 'fullAddress']);
```

可通过覆盖 `fields()` 来增加、删除、重命名和重定义字段，`fields()` 方法返回值应为数组，数组的键为字段名，数组的值为对应的可为属性名或匿名函数返回的字段定义对应的值。特使情况下，如果字段名和属性定义名相同，可以省略数组键，例如：

```
// 明确列出每个字段，特别用于你想确保数据表或模型属性改变不会导致你的字段改变(保证后端的 API 兼容).  
public function fields()
```

```

{

    return [

        // 字段名和属性名相同

        'id',

        // 字段名为 "email", 对应属性名为 "email_address"

        'email' => 'email_address',

        // 字段名为 "name", 值通过 PHP 代码返回

        'name' => function () {

            return $this->first_name . ' ' . $this->last_name;

        },

    ];

}

// 过滤掉一些字段，特别用于你想继承父类实现并不想用一些敏感字段

public function fields()
{

    $fields = parent::fields();

    // 去掉一些包含敏感信息的字段

    unset($fields['auth_key'], $fields['password_hash'], $fields['password_reset_token']);

    return $fields;

}

```

警告：由于模型的所有属性会被包含在导出数组，最好检查数据确保没包含敏感数据，如果有敏感数据，应覆盖 `fields()` 方法过滤掉，在上述列子中，我们选择过滤掉 `auth_key`, `password_hash` and `password_reset_token`。

最佳实践

模型是代表业务数据、规则和逻辑的中心地方，通常在很多地方重用， 在一个设计良好的应用中，模型通常比[控制器](#)代码多。

归纳起来，模型

- 可包含属性来展示业务数据;
- 可包含验证规则确保数据有效和完整;
- 可包含方法实现业务逻辑;
- 不应直接访问请求，`session` 和其他环境数据，这些数据应该由[控制器](#)传入到模型;
- 应避免嵌入 `HTML` 或其他展示代码，这些代码最好在 [视图](#)中处理;
- 单个模型中避免太多的 [场景](#).

在开发大型复杂系统时应经常考虑最后一条建议， 在这些系统中，模型会很大并在很多地方使用，因此会包含需要规则集和业务逻辑， 最后维护这些模型代码成为一个噩梦，因为一个简单修改会影响好多地方， 为确保模型好维护，最好使用以下策略：

- 定义可被多个 [应用主体](#) 或 [模块](#) 共享的模型基类集合。 这些模型类应包含通用的最小规则集合和逻辑。
- 在每个使用模型的 [应用主体](#) 或 [模块](#)中， 通过继承对应的模型基类来定义具体的模型类，具体模型类包含应用主体或模块指定的规则和逻辑。

例如，在[高级应用模板](#)，你可以定义一个模型基类 `common\models\Post`， 然后在前台应用中，定义并使用一个继承 `common\models\Post` 的具体模型类 `frontend\models\Post`， 在后台应用中可以类似地定义 `backend\models\Post`。 通过这种策略，你清楚 `frontend\models\Post` 只对应前台应用，如果你修改它，就无需担忧修改会影响后台应用。

过滤器

过滤器是 [控制器 动作](#) 执行之前或之后执行的对象。 例如访问控制过滤器可在动作执行之前来控制特殊终端用户是否有权限执行动作， 内容压缩过滤器可在动作执行之后发给终端用户之前压缩响应内容。

过滤器可包含 预过滤（过滤逻辑在动作之前） 或 后过滤（过滤逻辑在动作之后），也可同时包含两者。

使用过滤器

过滤器本质上是一类特殊的 [行为](#)，所以使用过滤器和 [使用 行为](#)一样。 可以在控制器类中覆盖它的 `[[yii\base\Controller::behaviors()|behaviors()]]` 方法来申明过滤器，如下所示：

```
public function behaviors()
{
    return [
```

```
[
    'class' => 'yii\filters\HttpCache',

    'only' => ['index', 'view'],

    'lastModified' => function ($action, $params) {

        $q = new \yii\db\Query();

        return $q->from('user')->max('updated_at');

    },

],

];
}
```

控制器类的过滤器默认应用到该类的 所有 动作，你可以配置`[[yii\base\ActionFilter::only|only]]`属性明确指定控制器应用到哪些动作。在上述例子中，`HttpCache` 过滤器只应用到 `index` 和 `view` 动作。也可以配置`[[yii\base\ActionFilter::except|except]]`属性使一些动作不执行过滤器。

除了控制器外，可在 [模块](#)或[应用主体](#) 中申明过滤器。申明之后，过滤器会应用到所属该模块或应用主体的 所有 控制器动作，除非像上述一样配置过滤器的 `[[yii\base\ActionFilter::only|only]]` 和 `[[yii\base\ActionFilter::except|except]]` 属性。

补充: 在模块或应用主体中申明过滤器，在

`[[yii\base\ActionFilter::only|only]]` 和

`[[yii\base\ActionFilter::except|except]]` 属性中使用[路由](#) 代替动作 ID，

因为在模块或应用主体中只用动作 ID 并不能唯一指定到具体动作。

当一个动作有多个过滤器时，根据以下规则先后执行：

- 预过滤
 - 按顺序执行应用主体中 `behaviors()`列出的过滤器。
 - 按顺序执行模块中 `behaviors()`列出的过滤器。
 - 按顺序执行控制器中 `behaviors()`列出的过滤器。
 - 如果任意过滤器终止动作执行，后面的过滤器（包括预过滤和后过滤）不再执行。
- 成功通过预过滤后执行动作。
- 后过滤
 - 倒序执行控制器中 `behaviors()`列出的过滤器。
 - 倒序执行模块中 `behaviors()`列出的过滤器。
 - 倒序执行应用主体中 `behaviors()`列出的过滤器。

创建过滤器

继承 `[[yii\base\ActionFilter]]` 类并覆盖 `[[yii\base\ActionFilter::beforeAction()|beforeAction()]]`

和/或 `[[yii\base\ActionFilter::afterAction()|afterAction()]]` 方法来创建动作的过滤器，前者在动作执行之前执行，后者在动作执行之后执行。

`[[yii\base\ActionFilter::beforeAction()|beforeAction()]]` 返回值决定动作是否应该执行， 如果为 `false`，之后的过滤器和动作不会继续执行。

下面的例子申明一个记录动作执行时间日志的过滤器。

```
namespace app\components;

use Yii;
use yii\base\ActionFilter;

class ActionTimeFilter extends ActionFilter
{
    private $_startTime;

    public function beforeAction($action)
    {
        $this->_startTime = microtime(true);

        return parent::beforeAction($action);
    }

    public function afterAction($action, $result)
    {
        $time = microtime(true) - $this->_startTime;

        Yii::trace("Action '{$action->uniqueId}' spent $time second.");

        return parent::afterAction($action, $result);
    }
}
```

核心过滤器

Yii 提供了一组常用过滤器，在 `yii\filters` 命名空间下，接下来我们简要介绍这些过滤器。

[[yii\filters\AccessControl|AccessControl]]

`AccessControl` 提供基于[[yii\filters\AccessControl::rules|rules]]规则的访问控制。特别是在动作执行之前，访问控制会检测所有规则并找到第一个符合上下文的变量（比如用户 IP 地址、登录状态等等）的规则，来决定允许还是拒绝请求动作的执行，如果没有规则符合，访问就会被拒绝。

如下示例表示允许已认证用户访问 `create` 和 `update` 动作，拒绝其他用户访问这两个动作。

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['create', 'update'],
            'rules' => [
                // 允许认证用户
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // 默认禁止其他用户
            ],
        ],
    ];
}
```

更多关于访问控制的详情请参阅 [授权](#) 一节。

认证方法过滤器

认证方法过滤器通过 [HTTP Basic Auth](#) 或 [OAuth 2](#) 来认证一个用户，认证方法过滤器类在 `yii\filters\auth` 命名空间下。

如下示例表示可使用 `[[yii\filters\auth\HttpBasicAuth]]` 来认证一个用户，它使用基于 HTTP 基础认证方法的令牌。注意为了可运行，`[[yii\web\User::identityClass|user identity class]]` 类必须实现 `[[yii\web\IdentityInterface::findIdentityByAccessToken()|findIdentityByAccessToken()]]` 方法。

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [

        'basicAuth' => [

            'class' => HttpBasicAuth::className(),

        ],

    ];
}
```

认证方法过滤器通常在实现 RESTful API 中使用，更多关于访问控制的详情请参阅 [RESTful 认证](#) 一节。

[[yii\filters\ContentNegotiator|ContentNegotiator]]

`ContentNegotiator` 支持响应内容格式处理和语言处理。通过检查 `GET` 参数和 `Accept` HTTP 头部来决定响应内容格式和语言。

如下示例，配置 `ContentNegotiator` 支持 JSON 和 XML 响应格式和英语（美国）和德语。

```
use yii\filters\ContentNegotiator;

use yii\web\Response;

public function behaviors()
{
    return [

        [

            'class' => ContentNegotiator::className(),
```

```

        'formats' => [

            'application/json' => Response::FORMAT_JSON,

            'application/xml' => Response::FORMAT_XML,

        ],

        'languages' => [

            'en-US',

            'de',

        ],

    ],

];
}

```

在[应用主体生命周期](#)过程中检测响应格式和语言简单很多，因此 **ContentNegotiator** 设计可被[引导启动组件](#)调用的过滤器。如下例所示可以将它配置在[应用主体配置](#)。

```

use yii\filters\ContentNegotiator;

use yii\web\Response;

[

    'bootstrap' => [

        [

            'class' => ContentNegotiator::className(),

            'formats' => [

                'application/json' => Response::FORMAT_JSON,

                'application/xml' => Response::FORMAT_XML,

            ],

            'languages' => [

                'en-US',

                'de',

            ],

        ],

    ],

];

```

```
    ],  
  
    ],  
  
    ],  
];
```

补充: 如果请求中没有检测到内容格式和语言, 使用[[formats]]和[[languages]]第一个配置项。

[[yii\filters\HttpCache|HttpCache]]

HttpCache 利用 **Last-Modified** 和 **Etag** HTTP 头实现客户端缓存。例如:

```
use yii\filters\HttpCache;  
  
public function behaviors()  
{  
    return [  
        [  
            'class' => HttpCache::className(),  
            'only' => ['index'],  
            'lastModified' => function ($action, $params) {  
                $q = new \yii\db\Query();  
                return $q->from('user')->max('updated_at');  
            },  
        ],  
    ],  
];  
}
```

更多关于使用 HttpCache 详情请参阅 [HTTP 缓存](#) 一节。

[[yii\filters\PageCache|PageCache]]

PageCache 实现服务器端整个页面的缓存。如下示例所示, PageCache 应用在 **index** 动作, 缓存整个页面 60 秒或 **post** 表的记录数发生变化。它也会根据不同应用语言保存不同的页面版本。

```

use yii\filters\PageCache;

use yii\caching\DbDependency;

public function behaviors()
{
    return [

        'pageCache' => [

            'class' => PageCache::className(),

            'only' => ['index'],

            'duration' => 60,

            'dependency' => [

                'class' => DbDependency::className(),

                'sql' => 'SELECT COUNT(*) FROM post',

            ],

            'variations' => [

                \Yii::$app->language,

            ]

        ],

    ];
}

```

更多关于使用 PageCache 详情请参阅 [页面缓存](#) 一节。

[[yii\filters\RateLimiter|RateLimiter]]

RateLimiter 根据 [漏桶算法](#) 来实现速率限制。 主要用在实现 RESTful APIs，更多关于该过滤器详情请参阅 [Rate Limiting](#) 一节。

[[yii\filters\VerbFilter|VerbFilter]]

VerbFilter 检查请求动作的 HTTP 请求方式是否允许执行，如果不允许，会抛出 HTTP 405 异常。 如下示例，VerbFilter 指定 CRUD 动作所允许的请求方式。

```

use yii\filters\VerbFilter;

public function behaviors()
{
    return [

        'verbs' => [

            'class' => VerbFilter::className(),

            'actions' => [

                'index' => ['get'],

                'view' => ['get'],

                'create' => ['get', 'post'],

                'update' => ['get', 'put', 'post'],

                'delete' => ['post', 'delete'],

            ],

        ],

    ];
}

```

[[yii\filters\Cors|Cors]]

跨域资源共享 [CORS](#) 机制允许一个网页的许多资源（例如字体、JavaScript 等） 这些资源可以通过其他域名访问获取。 特别是 JavaScript's AJAX 调用可使用 XMLHttpRequest 机制，由于同源安全策略该跨域请求会被网页浏览器禁止。 CORS 定义浏览器和服务器交互时哪些跨域请求允许和禁止。

[[yii\filters\Cors|Cors filter]] 应在 授权 / 认证 过滤器之前定义，以保证 CORS 头部被发送。

```

use yii\filters\Cors;

use yii\helpers\ArrayHelper;

public function behaviors()
{

    return ArrayHelper::merge([

```

```

[
    'class' => Cors::className(),
],
], parent::behaviors());
}

```

Cors 可转为使用 `cors` 属性。

- `cors['Origin']`: 定义允许来源的数组，可为 `['*']` (任何用户) 或 `['http://www.myserver.net', 'http://www.myotherserver.com']`. 默认为 `['*']`.
- `cors['Access-Control-Request-Method']`: 允许动作数组如 `['GET', 'OPTIONS', 'HEAD']`. 默认为 `['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']`.
- `cors['Access-Control-Request-Headers']`: 允许请求头部数组，可为 `['*']` 所有类型头部或 `['X-Request-With']` 指定类型头部. 默认为 `['*']`.
- `cors['Access-Control-Allow-Credentials']`: 定义当前请求是否使用证书，可为 `true`, `false` 或 `null` (不设置). 默认为 `null`.
- `cors['Access-Control-Max-Age']`: 定义请求的有效时间，默认为 `86400`.

例如，允许来源为 `http://www.myserver.net` 和方式为 `GET`, `HEAD` 和 `OPTIONS` 的 CORS 如下：

```

use yii\filters\Cors;

use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),

            'cors' => [

                'Origin' => ['http://www.myserver.net'],

                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'],

            ],

        ],

    ], parent::behaviors());
}

```



```
}
```

可以覆盖默认参数为每个动作调整 **CORS** 头部。例如，为 **login** 动作增加 **Access-Control-Allow-Credentials** 参数如下所示：

```
use yii\filters\Cors;

use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([

        [

            'class' => Cors::className(),

            'cors' => [

                'Origin' => ['http://www.myserver.net'],

                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'],

            ],

            'actions' => [

                'login' => [

                    'Access-Control-Allow-Credentials' => true,

                ]

            ]

        ],

    ], parent::behaviors());
}
```

小部件(Widget)

小部件是在 [视图](#) 中使用的可重用单元，使用面向对象方式创建复杂和可配置用户界面单元。 例如，日期选择器小部件可生成一个精致的允许用户选择日期的日期选择器， 你只需要在视图中插入如下代码：

```
<?php
use yii\jui\DatePicker;
```

```
?>

<?= DatePicker::widget(['name' => 'date']) ?>
```

Yii 提供许多优秀的小部件，比如[[yii\widgets\ActiveForm|active form]], [[yii\widgets\Menu|menu]], [jQuery UI widgets](#), [Twitter Bootstrap widgets](#)。接下来介绍小部件的基本知识，如果你想了解某个小部件请参考对应的类 API 文档。

使用小部件

小部件基本上在 [views](#) 中使用，在视图中可调用 [[yii\base\Widget::widget()]] 方法使用小部件。该方法使用 [配置](#) 数组初始化小部件并返回小部件渲染后的结果。例如如下代码插入一个日期选择器小部件，它配置为使用俄罗斯语，输入框内容为\$model 的 `from_date` 属性值。

```
<?php

use yii\jui\DatePicker;

?>

<?= DatePicker::widget([

    'model' => $model,

    'attribute' => 'from_date',

    'language' => 'ru',

    'clientOptions' => [

        'dateFormat' => 'yy-mm-dd',

    ],

]) ?>
```

一些小部件可在[[yii\base\Widget::begin()]] 和 [[yii\base\Widget::end()]] 调用中使用数据内容。Some widgets can take a block of content which should be enclosed between the invocation of 例如如下代码使用[[yii\widgets\ActiveForm]]小部件生成一个登录表单，小部件会在 `begin()` 和 `end()` 执行处分别生成<form>的开始标签和结束标签，中间的任何代码也会被渲染。

```
<?php

use yii\widgets\ActiveForm;

use yii\helpers\Html;

?>

<?php $form = ActiveForm::begin(['id' => 'login-form']); ?>
```

```
<?= $form->field($model, 'username') ?>
```

```
<?= $form->field($model, 'password')->passwordInput() ?>
```

```
<div class="form-group">
```

```
    <?= Html::submitButton('Login') ?>
```

```
</div>
```

```
<?php ActiveForm::end(); ?>
```

注意和调用 `[[yii\base\Widget::widget()]]` 返回渲染结果不同，调用 `[[yii\base\Widget::begin()]]` 方法返回一个可构建小部件内容的小部件实例。

创建小部件

Creating Widgets

继承 `[[yii\base\Widget]]` 类并覆盖 `[[yii\base\Widget::init()]]` 和/或 `[[yii\base\Widget::run()]]` 方法可创建小部件。通常 `init()` 方法处理小部件属性，`run()` 方法包含小部件生成渲染结果的代码。渲染结果可在 `run()` 方法中直接"echoed"输出或以字符串返回。

如下代码中 `HelloWidget` 编码并显示赋给 `message` 属性的值，如果属性没有被赋值，默认会显示"Hello World"。

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public $message;

    public function init()
```

```

{

    parent::init();

    if ($this->message === null) {

        $this->message = 'Hello World';

    }

}

public function run()

{

    return Html::encode($this->message);

}

}

```

使用这个小部件只需在视图中简单使用如下代码：

```

<?php
use app\components\HelloWidget;
?>

<?= HelloWidget::widget(['message' => 'Good morning']) ?>

```

以下是另一种可在 `begin()` 和 `end()` 调用中使用的 `HelloWidget`，HTML 编码内容然后显示。

```

namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{

    public function init()

    {

        parent::init();
    }

}

```

```

        ob_start();

    }

    public function run()
    {
        $content = ob_get_clean();

        return Html::encode($content);
    }
}

```

如上所示，PHP 输出缓冲在 `init()` 启动，所有在 `init()` 和 `run()` 方法之间的输出内容都会被获取，并在 `run()` 处理和返回。

补充：当你调用 `[[yii\base\Widget::begin()]]` 时会创建一个新的小部件实例并在构造结束时调用 `init()` 方法，在 `end()` 时会调用 `run()` 方法并输出返回结果。

如下代码显示如何使用这种 `HelloWidget`：

```

<?php
use app\components\HelloWidget;
?>

<?php HelloWidget::begin(); ?>

    content that may contain <tag>'s

<?php HelloWidget::end(); ?>

```

有时小部件需要渲染很多内容，一种更好的办法是将内容放入一个视图文件，然后调用 `[[yii\base\Widget::render()]]` 方法渲染该视图文件，例如：

```

public function run()
{
    return $this->render('hello');
}

```

小部件的视图文件默认存储在 `WidgetPath/views` 目录，`WidgetPath` 代表小部件类文件所在的目录。假如上述示例小部件类文件在 `@app/components` 下，会渲染 `@app/components/views/hello.php` 视图文

件。 You may override 可以覆盖[[yii\base\Widget::getViewPath()]]方法自定义视图文件所在路径。

最佳实践

小部件是面向对象方式来重用视图代码。

创建小部件时仍需要遵循 MVC 模式，通常逻辑代码在小部件类，展示内容在[视图](#)中。

小部件设计时应是独立的，也就是说使用一个小部件时候，可以直接丢弃它而不需要额外的处理。但是当小部件需要外部资源如 CSS, JavaScript, 图片等会比较棘手， 幸运的时候 Yii 提供 [资源包](#) 来解决问题。

当一个小部件只包含视图代码，它和[视图](#)很相似， 实际上，在这种情况下，唯一的区别是小部件是可以重用类，视图只是应用中使用的普通 PHP 脚本。

模块(Module)

模块是独立的软件单元，由[模型](#)，[视图](#)，[控制器](#)和其他支持组件组成， 终端用户可以访问在[应用主体](#)中已安装的模块的控制器， 模块被当成小应用主体来看待，和[应用主体](#)不同的是， 模块不能单独部署，必须属于某个应用主体。

创建模块

模块被组织成一个称为[[yii\base\Module::basePath|base path]]的目录， 在该目录中有子目录如 [controllers](#)， [models](#)， [views](#) 分别为对应控制器，模型，视图和其他代码，和应用非常类似。 如下例子显示一个模型的目录结构：

forum/

Module.php	模块类文件
controllers/	包含控制器类文件
DefaultController.php	default 控制器类文件
models/	包含模型类文件
views/	包含控制器视图文件和布局文件
layouts/	包含布局文件
default/	包含 DefaultController 控制器视图文件
index.php	index 视图文件

模块类

每个模块都有一个继承[[yii\base\Module]]的模块类，该类文件直接放在模块的 [[yii\base\Module::basePath|base path]]目录下， 并且能被[自动加载](#)。当一个模块被访问，和 [应用主体实例](#) 类似会创建该模块类唯一实例，模块实例用来帮模块内代码共享数据和组件。

以下示例一个模块类大致定义：

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';

        // ... 其他初始化代码 ...
    }
}
```

如果 `init()` 方法包含很多初始化模块属性代码， 可将他们保存在[配置](#) 并在 `init()`中使用以下代码加载：

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

    // 从 config.php 加载配置来初始化模块

    \Yii::configure($this, require(__DIR__ . '/config.php'));
}
```

`config.php` 配置文件可能包含以下内容，类似[应用主体配置](#)。

```
<?php
return [

    'components' => [

        // list of component configurations
    ]
];
```

```
],  
  
    'params' => [  
  
        // list of parameters  
  
    ],  
  
];
```

模块中的控制器

创建模块的控制器时，惯例是将控制器类放在模块类命名空间的 `controllers` 子命名空间中，也意味着要将控制器类文件放在模块 `[[yii\base\Module::basePath|base path]]` 目录中的 `controllers` 子目录中。例如，上小节中要在 `forum` 模块中创建 `post` 控制器，应像如下申明控制器类：

```
namespace app\modules\forum\controllers;  
  
use yii\web\Controller;  
  
class PostController extends Controller  
{  
  
    // ...  
}
```

可配置 `[[yii\base\Module::controllerNamespace]]` 属性来自定义控制器类的命名空间，如果一些控制器不再该命名空间下，可配置 `[[yii\base\Module::controllerMap]]` 属性让它们能被访问，这类似于 [应用主体配置](#) 所做的。

模块中的视图

视图应放在模块的 `[[yii\base\Module::basePath|base path]]` 对应目录下的 `views` 目录，对于模块中控制器对应的视图文件应放在 `views/ControllerID` 目录下，其中 `ControllerID` 对应 [控制器 ID](#)。For example, if 例如，假定控制器类为 `PostController`，目录对应模块 `[[yii\base\Module::basePath|base path]]` 目录下的 `views/post` 目录。

模块可指定 [布局](#)，它用在模块的控制器视图渲染。布局文件默认放在 `views/layouts` 目录下，可配置 `[[yii\base\Module::layout]]` 属性指定布局名，如果没有配置 `layout` 属性名，默认会使用应用的布局。

使用模块

要在应用中使用模块，只需要将模块加入到应用主体配置的

`[[yii\base\Application::modules|modules]]`属性的列表中， 如下代码的[应用主体配置](#) 使用 `forum` 模块：

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... 模块其他配置 ...
        ],
    ],
]
```

`[[yii\base\Application::modules|modules]]` 属性使用模块配置数组，每个数组键为模块 ID， 它标识该应用中唯一的模块，数组的值为用来创建模块的 [配置](#)。

路由

和访问应用的控制器类似，[路由](#) 也用在模块中控制器的寻址， 模块中控制器的路由必须以模块 ID 开始，接下来为控制器 ID 和操作 ID。 例如，假定应用使用一个名为 `forum` 模块，路由 `forum/post/index` 代表模块中 `post` 控制器的 `index` 操作， 如果路由只包含模块 ID，默认为 `default` 的 `[[yii\base\Module::defaultRoute]]` 属性来决定使用哪个控制器/操作， 也就是说路由 `forum` 可能代表 `forum` 模块的 `default` 控制器。

访问模块

在模块中，可能经常需要获取[模块类](#)的实例来访问模块 ID，模块参数，模块组件等， 可以使用如下语句来获取：

```
$module = MyModuleClass::getInstance();
```

其中 `MyModuleClass` 对应你想要的模块类，`getInstance()` 方法返回当前请求的模块类实例， 如果模块没有被请求，该方法会返回空，注意不需要手动创建一个模块类，因为手动创建的和 `Yii` 处理请求时自动创建的不同。

补充：当开发模块时，你不能假定模块使用固定的 ID，因为在应用或其他没模块中，模块可能会对应到任意的 ID， 为了获取模块 ID，应使用上述代码获取模块实例，然后通过 `$module->id` 获取模块 ID。

也可以使用如下方式访问模块实例：

```
// 获取 ID 为 "forum" 的模块
$module = \Yii::$app->getModule('forum');
```

```
// 获取处理当前请求控制器所属的模块
```

```
$module = \Yii::$app->controller->module;
```

第一种方式仅在你知道模块 ID 的情况下有效，第二种方式在你知道处理请求的控制器下使用。

一旦获取到模块实例，可访问注册到模块的参数和组件，例如：

```
$maxPostCount = $module->params['maxPostCount'];
```

引导启动模块

有些模块在每个请求下都有运行，`[[yii\debug\Module|debug]]` 模块就是这种，为此将这种模块加入到应用主体的`[[yii\base\Application::bootstrap|bootstrap]]` 属性中。

例如，如下示例的应用主体配置会确保 `debug` 模块每次都被加载：

```
[  
    'bootstrap' => [  
        'debug',  
    ],  
  
    'modules' => [  
        'debug' => 'yii\debug\Module',  
    ],  
]
```

模块嵌套

模块可无限级嵌套，也就是说，模块可以包含另一个包含模块的模块，我们称前者为父模块，后者为子模块，子模块必须在父模块的`[[yii\base\Module::modules|modules]]`属性中申明，例如：

```
namespace app\modules\forum;
```

```
class Module extends \yii\base\Module
```

```
{
```

```
    public function init()
```

```
{
```

```
parent::init();

$this->modules = [

    'admin' => [

        // 此处应考虑使用一个更短的命名空间

        'class' => 'app\modules\forum\modules\admin\Module',

    ],

];

}
```

在嵌套模块中的控制器，它的路由应包含它所有祖先模块的 ID，例如 `forum/admin/dashboard/index` 代表 在模块 `forum` 中子模块 `admin` 中 `dashboard` 控制器的 `index` 操作。

最佳实践

模块在大型项目中常备使用，这些项目的特性可分组，每个组包含一些强相关的特性，每个特性组可以做成一个模块由特定的开发人员和开发组来开发和维护。

在特性组上，使用模块也是重用代码的好方式，一些常用特性，如用户管理，评论管理，可以开发成模块，这样在相关项目中非常容易被重用。

资源(Asset)

Yii 中的资源是和 Web 页面相关的文件，可为 CSS 文件，JavaScript 文件，图片或视频等，资源放在 Web 可访问的目录下，直接被 Web 服务器调用。

通过程序自动管理资源更好一点，例如，当你在页面中使用 `[[yii\jui\DatePicker]]` 小部件时，它会自动包含需要的 CSS 和 JavaScript 文件，而不是要求你手工去找到这些文件并包含，当你升级小部件时，它会自动使用新版本的资源文件，在本教程中，我们会详述 Yii 提供的强大的资源管理功能。

资源包

Yii 在资源包中管理资源，资源包简单的说就是放在一个目录下的资源集合，当在[视图](#)中注册一个资源包，在渲染 Web 页面时会包含包中的 CSS 和 JavaScript 文件。

定义资源包

资源包指定为继承[[yii\web\AssetBundle]]的 PHP 类，包名为可[自动加载](#)的 PHP 类名，在资源包类中，要指定资源所在位置，包含哪些 CSS 和 JavaScript 文件以及和其他包的依赖关系。

如下代码定义[基础应用模板](#)使用的主要资源包：

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';

    public $baseUrl = '@web';

    public $css = [

        'css/site.css',

    ];

    public $js = [

    ];

    public $depends = [

        'yii\web\YiiAsset',

        'yii\bootstrap\BootstrapAsset',

    ];
}
```

如上 **AppAsset** 类指定资源文件放在 **@webroot** 目录下，对应的 URL 为 **@web**，资源包中包含一个 CSS 文件 **css/site.css**，没有 JavaScript 文件，依赖其他两个包 [[yii\web\YiiAsset]] 和 [[yii\bootstrap\BootstrapAsset]]，关于[[yii\web\AssetBundle]] 的属性的更多详细如下所述：

- 当根目录不能被 Web 访问时该属性应设置，否则，应设置 `yii\web\AssetBundle::basePath|basePath` 属性和 `yii\web\AssetBundle::baseUrl|baseUrl`。 [路径别名](#) 可在此处使用；

- 当指定`[yii\web\AssetBundle::sourcePath|sourcePath]` 属性， **资源管理器** 会发布包的资源到一个可 **Web** 访问并覆盖该属性， 如果你的资源文件在一个 **Web** 可访问目录下， 应设置该属性， 这样就不用再发布了。 **路径别名** 可在此处使用。
- 和 `[yii\web\AssetBundle::basePath|basePath]` 类似， 如果你指定 `[yii\web\AssetBundle::sourcePath|sourcePath]` 属性， **资源管理器** 会发布这些资源并覆盖该属性， **路径别名** 可在此处使用。
- 每个 **JavaScript** 文件可指定为以下两种格式之一：
 - 相对路径表示为本地 **JavaScript** 文件 (如 `js/main.js`)， 文件实际的路径在该相对路径前加上 `[[yii\web\AssetManager::basePath]]`， 文件实际的 **URL** 在该路径前加上 `[[yii\web\AssetManager::baseUrl]]`。
 - 绝对 **URL** 地址表示为外部 **JavaScript** 文件， 如 `http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js` 或 `//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js`。
- `[[yii\web\AssetBundle::css|css]]`: 一个包含该资源包 **JavaScript** 文件的数组， 该数组格式和 `[yii\web\AssetBundle::js|js]` 相同。
- `[[yii\web\AssetBundle::jsOptions|jsOptions]]`: 当调用`[[yii\web\View::registerJsFile()]]` 注册该包 每个 **JavaScript** 文件时， 指定传递到该方法的选项。
- `[[yii\web\AssetBundle::cssOptions|cssOptions]]`: 当调用`[[yii\web\View::registerCssFile()]]`注册该包 每个 **css** 文件时， 指定传递到该方法的选项。
- 指定传递到该方法的选项， 仅在指定了`[yii\web\AssetBundle::sourcePath|sourcePath]`属性时使用。

资源位置

资源根据它们的位置可以分为：

- 源资源: 资源文件和 **PHP** 源代码放在一起， 不能被 **Web** 直接访问， 为了使用这些源资源， 它们要拷贝到一个可 **Web** 访问的 **Web** 目录中 成为发布的资源， 这个过程称为发布资源， 随后会详细介绍。
- 发布资源: 资源文件放在可通过 **Web** 直接访问的 **Web** 目录中；
- 外部资源: 资源文件放在你的 **Web** 应用不同的 **Web** 服务器上；

当定义资源包类时候， 如果你指定了`[yii\web\AssetBundle::sourcePath|sourcePath]` 属性， 就表示任何使用相对路径的资源会被 当作源资源； 如果没有指定该属性， 就表示这些资源为发布资源（因此应指定`[yii\web\AssetBundle::basePath|basePath]` 和`[yii\web\AssetBundle::baseUrl|baseUrl]` 让 **Yii** 知道它们的位置）。

推荐将资源文件放到 **Web** 目录以避免不必要的发布资源过程， 这就是之前的例子指定 `[yii\web\AssetBundle::basePath|basePath]` 而不是`[yii\web\AssetBundle::sourcePath|sourcePath]`。

对于 **扩展**来说， 由于它们的资源和源代码都在不能 **Web** 访问的目录下， 在定义资源包类时必须指定`[yii\web\AssetBundle::sourcePath|sourcePath]`属性。

注意: `[[yii\web\AssetBundle::sourcePath|source path]]` 属性不要用

`@webroot/assets`，该路径默认为 `[[yii\web\AssetManager|asset manager]]` 资源管理器将源资源发布后存储资源的路径，该路径的所有内容会认为是临时文件， 可能会被删除。

资源依赖

当 Web 页面包含多个 CSS 或 JavaScript 文件时，它们有一定的先后顺序以避免属性覆盖， 例如，Web 页面在使用 jQuery UI 小部件前必须确保 jQuery JavaScript 文件已经被包含了， 我们称这种资源先后次序称为资源依赖。

资源依赖主要通过 `[[yii\web\AssetBundle::depends]]` 属性来指定， 在 `AppAsset` 示例中，资源包依赖其他两个资源包：`[[yii\web\YiiAsset]]` 和 `[[yii\bootstrap\BootstrapAsset]]` 也就是该资源包的 CSS 和 JavaScript 文件要在这两个依赖包的文件包含 之后 才包含。

资源依赖关系是可传递，也就是人说 A 依赖 B，B 依赖 C，那么 A 也依赖 C。

资源选项

可指定 `[[yii\web\AssetBundle::cssOptions|cssOptions]]` 和 `[[yii\web\AssetBundle::jsOptions|jsOptions]]` 属性来自定义页面包含 CSS 和 JavaScript 文件的方式， 这些属性值会分别传递给 `[[yii\web\View::registerCssFile()]]` 和 `[[yii\web\View::registerJsFile()]]` 方法， 在[视图](#) 调用这些方法包含 CSS 和 JavaScript 文件时。

注意：在资源包类中设置的选项会应用到该包中 每个 CSS/JavaScript 文件，如果想对每个文件使用不同的选项， 应创建不同的资源包并在每个包中使用一个选项集。

例如，只想 IE9 或更高的浏览器包含一个 CSS 文件，可以使用如下选项：

```
public $cssOptions = ['condition' => 'lte IE9'];
```

这会是包中的 CSS 文件使用以下 HTML 标签包含进来：

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

为链接标签包含 `<noscript>` 可使用如下代码：

```
public $cssOptions = ['noscript' => true];
```

为使 JavaScript 文件包含在页面 head 区域（JavaScript 文件默认包含在 body 的结束处）使用以下选项：

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

Bower 和 NPM 资源

大多数 JavaScript/CSS 包通过 Bower 和/或 NPM 管理， 如果你的应用或扩展使用这些包，推荐你遵

循以下步骤来管理库中的资源:

- 1.修改应用或扩展的 `composer.json` 文件将包列入 `require` 中, 应使用 `bower-asset/PackageName` (Bower 包) 或 `npm-asset/PackageName` (NPM 包)来对应库。
- 2.创建一个资源包类并将你的应用或扩展要使用的 JavaScript/CSS 文件列入到类中, 应设置 `[[yii\web\AssetBundle::sourcePath|sourcePath]]` 属性为 `@bower/PackageName` 或 `@npm/PackageName`, 因为根据别名 Composer 会安装 Bower 或 NPM 包到对应的目录下。

注意: 一些包会将它们分布式文件放到一个子目录中, 对于这种情况, 应指定子目录作为 `[[yii\web\AssetBundle::sourcePath|sourcePath]]` 属性值, 例如, `[[yii\web\jQueryAsset]]` 使用 `@bower/jquery/dist` 而不是 `@bower/jquery`。

使用资源包

为使用资源包, 在视图调用 `[[yii\web\AssetBundle::register()]]` 方法先注册资源, 例如, 在视图模板可使用如下代码注册资源包:

```
use app\assets\AppAsset;

AppAsset::register($this); // $this 代表视图对象
```

如果在其他地方注册资源包, 应提供视图对象, 如在 `小部件` 类中注册资源包, 可以通过 `$this->view` 获取视图对象。

当在视图中注册一个资源包时, 在背后 Yii 会注册它所依赖的资源包, 如果资源包是放在 Web 不可访问的目录下, 会被发布到可访问的目录, 后续当视图渲染页面时, 会生成这些注册包包含的 CSS 和 JavaScript 文件对应的 `<link>` 和 `<script>` 标签, 这些标签的先后顺序取决于资源包的依赖关系以及在 `[[yii\web\AssetBundle::css]]` 和 `[[yii\web\AssetBundle::js]]` 的列出来的前后顺序。

自定义资源包

Yii 通过名为 `assetManager` 的应用组件实现 `[[yii\web\AssetManager]]` 来管理应用组件, 通过配置 `[[yii\web\AssetManager::bundles]]` 属性, 可以自定义资源包的行为, 例如, `[[yii\web\jQueryAsset]]` 资源包默认从 jquery Bower 包中使用 `jquery.js` 文件, 为了提高可用性和性能, 你可能需要从 Google 服务器上获取 jquery 文件, 可以在应用配置中配置 `assetManager`, 如下所示:

```
return [

    // ...

    'components' => [

        'assetManager' => [

            'bundles' => [
```

```

        'yii\web\jQueryAsset' => [

            'sourcePath' => null, // 一定不要发布该资源

            'js' => [

                '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',

            ]

        ],

    ],

],

];

```

可通过类似[[yii\web\AssetManager::bundles]]配置多个资源包，数组的键应为资源包的类名（最开头不要反斜杠），数组的值为对应的[配置数组](#)。

提示：可以根据条件判断使用哪个资源，如下示例为如何在开发环境用 **jquery.js**，否则用 **jquery.min.js**：

```

'yii\web\jQueryAsset' => [

    'js' => [

        YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'

    ]

],

```

可以设置资源包的名称对应 **false** 来禁用想禁用的一个或多个资源包，当视图中注册一个禁用资源包，视图不会包含任何该包的资源以及不会注册它所依赖的包，例如，为禁用[[yii\web\jQueryAsset]]，可以使用如下配置：

```

return [

    // ...

    'components' => [

        'assetManager' => [

            'bundles' => [

                'yii\web\jQueryAsset' => false,

            ],

        ],

    ],

];

```



```
    ],  
  
    ],  
];
```

可设置`[[yii\web\AssetManager::bundles]]`为 `false` 禁用 所有 的资源包。

资源部署

有时你想"修复" 多个资源包中资源文件的错误/不兼容，例如包 A 使用 1.11.1 版本的 `jquery.min.js`，包 B 使用 2.1.1 版本的 `jquery.js`，可自定义每个包来解决这个问题，更好的方式是使用资源部署特性来部署不正确的资源为想要的，为此，配置`[[yii\web\AssetManager::assetMap]]`属性，如下所示：

```
return [  
  
    // ...  
  
    'components' => [  
  
        'assetManager' => [  
  
            'assetMap' => [  
  
                'jquery.js' => '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',  
  
            ],  
  
        ],  
  
    ],  
  
];
```

`[[yii\web\AssetManager::assetMap|assetMap]]`的键为你想要修复的资源名，值为你想要使用的资源路径，当视图注册资源包，在`[[yii\web\AssetBundle::css|css]]` 和 `[[yii\web\AssetBundle::js|js]]` 数组中每个相关资源文件会和该部署进行对比，如果数组任何键对比为资源文件的最后文件名（如果有的话前缀为 `[[yii\web\AssetBundle::sourcePath]]`），对应的值为替换原来的资源。例如，资源文件 `my/path/to/jquery.js` 匹配键 `jquery.js`。

注意：只有相对相对路径指定的资源对应到资源部署，替换的资源路径可以为绝对路径，也可和`[[yii\web\AssetManager::basePath]]`相关的路径。

资源发布

如前所述，如果资源包放在 Web 不能访问的目录，当视图注册资源时资源会被拷贝到一个 Web 可访问的目录中，这个过程称为资源发布，`[[yii\web\AssetManager|asset manager]]`会自动处理该过程。

资源默认会发布到`@webroot/assets` 目录，对应的 URL 为`@web/assets`，可配置 `[[yii\web\AssetManager::basePath|basePath]]` 和 `[[yii\web\AssetManager::baseUrl|baseUrl]]`

属性自定义发布位置。

除了拷贝文件方式发布资源，如果操作系统和 Web 服务器允许可以使用符号链接，该功能可以通过设置 `[[yii\web\AssetManager::linkAssets|linkAssets]]` 为 `true` 来启用。

```
return [  
    // ...  
    'components' => [  
        'assetManager' => [  
            'linkAssets' => true,  
        ],  
    ],  
];
```

使用以上配置，资源管理器会创建一个符号链接到要发布的资源包源路径，这比拷贝文件方式快并能确保发布的资源一直为最新的。

常用资源包

Yii 框架定义许多资源包，如下资源包是最常用，可在你的应用或扩展代码中引用它们。

- `[[yii\web\YiiAsset]]`: 主要包含 `yii.js` 文件，该文件完成模块 JavaScript 代码组织功能，也为 `data-method` 和 `data-confirm` 属性提供特别支持和其他有用的功能。
- `[[yii\web\jQueryAsset]]`: 包含从 jQuery bower 包的 `jquery.js` 文件。
- `[[yii\bootstrap\BootstrapAsset]]`: 包含从 Twitter Bootstrap 框架的 CSS 文件。
- `[[yii\bootstrap\BootstrapPluginAsset]]`: 包含从 Twitter Bootstrap 框架的 JavaScript 文件来支持 Bootstrap JavaScript 插件。
- `[[yii\jui\JuiAsset]]`: 包含从 jQuery UI 库的 CSS 和 JavaScript 文件。

如果你的代码需要 jQuery, jQuery UI 或 Bootstrap，应尽量使用这些预定义资源包而非自己创建，如果这些包的默认配置不能满足你的需求，可以自定义配置，详情参考[自定义资源包](#)。

资源转换

除了直接编写 CSS 和/或 JavaScript 代码，开发人员经常使用扩展语法来编写，再使用特殊的工具将它们转换成 CSS/Javascript。例如，对于 CSS 代码可使用 LESS 或 SCSS，对于 JavaScript 可使用 TypeScript。

可将使用扩展语法的资源文件列到资源包的 `[[yii\web\AssetBundle::css|css]]` 和 `[[yii\web\AssetBundle::js|js]]` 中，如下所示：

```

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';

    public $baseUrl = '@web';

    public $css = [

        'css/site.less',

    ];

    public $js = [

        'js/site.ts',

    ];

    public $depends = [

        'yii\web\YiiAsset',

        'yii\bootstrap\BootstrapAsset',

    ];
}

```

当在视图中注册一个这样的资源包，`[[yii\web\AssetManager|asset manager]]`资源管理器会自动运行预处理工具将使用扩展语法的资源转换成 **CSS/JavaScript**，当视图最终渲染页面时，在页面中包含的是 **CSS/Javascript** 文件，而不是原始的扩展语法代码文件。

Yii 使用文件扩展名来表示资源使用哪种扩展语法，默认可以识别如下语法和文件扩展名：

- **LESS**: `.less`
- **SCSS**: `.scss`
- **Stylus**: `.styl`
- **CoffeeScript**: `.coffee`
- **TypeScript**: `.ts`

Yii 依靠安装的预处理工具来转换资源，例如，为使用 **LESS**，应安装 `lessc` 预处理命令。

可配置`[[yii\web\AssetManager::converter]]`自定义预处理命令和支持的扩展语法，如下所示：

```

return [

    'components' => [

        'assetManager' => [

```

```

'converter' => [

    'class' => 'yii\web\AssetConverter',

    'commands' => [

        'less' => ['css', 'lessc {from} {to} --no-color'],

        'ts' => ['js', 'tsc --out {to} {from}'],

    ],

],

],

],

];

```

如上所示，通过[[yii\web\AssetConverter::commands]] 属性指定支持的扩展语法， 数组的键为文件扩展名（前面不要.），数组的值为目标资源文件扩展名和执行资源转换的命令， 命令中的标记 **{from}** 和 **{to}** 会分别被源资源文件路径和目标资源文件路径替代。

补充: 除了以上方式，也有其他方式来处理扩展语法资源，例如，可使用编译工具如 [grunt](#) 来监控并自动转换扩展语法资源，此时，应使用资源包中编译后的 **CSS/JavaScript** 文件而不是原始文件。

合并和压缩资源

一个 **Web** 页面可以包含很多 **CSS** 和/或 **JavaScript** 文件，为减少 **HTTP** 请求和这些下载文件的大小，通常的方式是在页面中合并并压缩多个 **CSS/JavaScript** 文件为一个或很少的几个文件，并使用压缩后的文件而不是原始文件。

补充: 合并和压缩资源通常在应用在产品上线模式，在开发模式下使用原始的 **CSS/JavaScript** 更方便调试。

接下来介绍一种合并和压缩资源文件而不需要修改已有代码的方式：

- 1.找出应用中所有你想要合并和压缩的资源包，
- 2.将这些包分成一个或几个组，注意每个包只能属于其中一个组，
- 3.合并/压缩每个组里 **CSS** 文件到一个文件，同样方式处理 **JavaScript** 文件，
- 4.为每个组定义新的资源包：

- 设置[[yii\web\AssetBundle::css|css]] 和 [\[yii\web\AssetBundle::js|js\]](#) 属性分别为压缩后的 **CSS** 和 **JavaScript** 文件；
- 自定义设置每个组内的资源包，设置资源包的[[yii\web\AssetBundle::css|css]] 和 [\[yii\web\AssetBundle::js|js\]](#) 属性为空，并设置它们的 [\[yii\web\AssetBundle::depends|depends\]](#) 属性为每个组新创建的资源包。

使用这种方式，当在视图中注册资源包时，会自动触发原始包所属的组资源包的注册，然后，页面就会包含以合并/压缩的资源文件，而不是原始文件。

示例

使用一个示例来解释以上这种方式：

假定你的应用有两个页面 X 和 Y，页面 X 使用资源包 A，B 和 C，页面 Y 使用资源包 B，C 和 D。

有两种方式划分这些资源包，一种使用一个组包含所有资源包，另一种是将 (A,B,C) 放在组 X，(B, C, C) 放在组 Y，哪种方式更好？第一种方式优点是两个页面使用相同的已合并 CSS 和 JavaScript 文件使 HTTP 缓存更高效，另一方面，由于单个组包含所有文件，已合并的 CSS 和 Javascript 文件会更大，因此会增加文件传输时间，在这个示例中，我们使用第一种方式，也就是用一个组包含所有包。

补充：将资源包分组并不是无价值的，通常要求分析现实中不同页面各种资源的数据量，开始时为简便使用一个组。

在所有包中使用工具(例如 [Closure Compiler](#), [YUI Compressor](#)) 来合并和压缩 CSS 和 JavaScript 文件，注意合并后的文件满足包间的先后依赖关系，例如，如果包 A 依赖 B，B 依赖 C 和 D，那么资源文件列表以 C 和 D 开始，然后为 B 最后为 A。

合并和压缩之后，会得到一个 CSS 文件和一个 JavaScript 文件，假定它们的名称为 `all-xyz.css` 和 `all-xyz.js`，`xyz` 为使文件名唯一以避免 HTTP 缓存问题的时间戳或哈希值。

现在到最后一步了，在应用配置中配置`[[yii\web\AssetManager|asset manager]]` 资源管理器如下所示：

```
return [

    'components' => [

        'assetManager' => [

            'bundles' => [

                'all' => [

                    'class' => 'yii\web\AssetBundle',

                    'basePath' => '@webroot/assets',

                    'baseUrl' => '@web/assets',

                    'css' => ['all-xyz.css'],

                    'js' => ['all-xyz.js'],

                ],

                'A' => ['css' => [], 'js' => [], 'depends' => ['all']],

                'B' => ['css' => [], 'js' => [], 'depends' => ['all']],
```

```

        'C' => ['css' => [], 'js' => [], 'depends' => ['all']],

        'D' => ['css' => [], 'js' => [], 'depends' => ['all']],

    ],

    ],

    ],

];

```

如[自定义资源包](#)小节中所述，如上配置改变每个包的默认行为，特别是包 A、B、C 和 D 不再包含任何资源文件，都依赖包含合并后的 `all-xyz.css` 和 `all-xyz.js` 文件的包 `all`，因此，对于页面 X 会包含这两个合并后的文件而不是包 A、B、C 的原始文件，对于页面 Y 也是如此。

最后有个方法更好地处理上述方式，除了直接修改应用配置文件，可将自定义包数组放到一个文件，在应用配置中根据条件包含该文件，例如：

```

return [

    'components' => [

        'assetManager' => [

            'bundles' => require(__DIR__ . '/' . (YII_ENV_PROD ? 'assets-prod.php' :
'assets-dev.php')),

        ],

    ],

];

```

如上所示，在产品上线模式下资源包数组存储在 `assets-prod.php` 文件中，不是产品上线模式存储在 `assets-dev.php` 文件中。

使用 `asset` 命令

Yii 提供一个名为 `asset` 控制台命令来使上述操作自动处理。

为使用该命令，应先创建一个配置文件设置哪些资源包要合并以及分组方式，可使用 `asset/template` 子命令来生成一个模板，然后修改模板成你想要的。

```
yii asset/template assets.php
```

该命令在当前目录下生成一个名为 `assets.php` 的文件，文件的内容类似如下：

```

<?php

/**

 * 为控制台命令"yii asset"使用的配置文件

```

* 注意在控制台环境下，一些路径别名如 '@webroot' 和 '@web' 不会存在

* 请定义不存在的路径别名

*/

return [

// 为 JavaScript 文件压缩修改 command/callback

'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file {to}',

// 为 CSS 文件压缩修改 command/callback

'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o {to}',

// 要压缩的资源包列表

'bundles' => [

// 'yii\web\YiiAsset',

// 'yii\web\jQueryAsset',

],

// 资源包压缩后的输出

'targets' => [

'all' => [

'class' => 'yii\web\AssetBundle',

'basePath' => '@webroot/assets',

'baseUrl' => '@web/assets',

'js' => 'js/all-{'hash}.js',

'css' => 'css/all-{'hash}.css',

],

],

// 资源管理器配置:

'assetManager' => [

],

```
];
```

应修改该文件的 **bundles** 的选项指定哪些包你想要合并，在 **targets** 选项中应指定这些包如何分组，如前述的可以指定一个或多个组。

注意: 由于在控制台应用别名 **@webroot** and **@web** 不可用，应在配置中明确指定它们。

JavaScript 文件会被合并压缩后写入到 **js/all-`{hash}`.js** 文件，其中 **{hash}** 会被结果文件的哈希值替换。

jsCompressor 和 **cssCompressor** 选项指定控制台命令或 PHP 回调函数来执行 JavaScript 和 CSS 合并和压缩，Yii 默认使用 **Closure Compiler** 来合并 JavaScript 文件，使用 **YUI Compressor** 来合并 CSS 文件，你应手工安装这些工具或修改选项使用你喜欢的工具。

根据配置文件，可执行 **asset** 命令来合并和压缩资源文件并生成一个新的资源包配置文件 **assets-prod.php**:

```
yii asset assets.php config/assets-prod.php
```

生成的配置文件可以在应用配置中包含，如最后一小节所描述的。

补充: 使用 **asset** 命令并不是唯一一种自动合并和压缩过程的方法，可使用优秀的工具 **grunt** 来完成这个过程。

扩展 Extensions

Extensions are redistributable software packages specifically designed to be used in Yii applications and provide ready-to-use features. For example, the [yiisoft/yii2-debug](#) extension adds a handy debug toolbar at the bottom of every page in your application to help you more easily grasp how the pages are generated. You can use extensions to accelerate your development process. You can also package your code as extensions to share with other people your great work.

Info: We use the term "extension" to refer to Yii-specific software packages. For general purpose software packages that can be used without Yii, we will refer to them using the term "package" or "library".

Using Extensions

To use an extension, you need to install it first. Most extensions are distributed as [Composer](#) packages which can be installed by taking the following two simple steps:

- 1.modify the **composer.json** file of your application and specify which extensions (Composer packages) you want to install.
- 2.run **composer install** to install the specified extensions.

Note that you may need to install [Composer](#) if you do not have it.

By default, Composer installs packages registered on [Packagist](#) - the biggest repository for open source Composer packages. You can look for extensions on Packagist. You may also [create your own repository](#) and configure Composer to use it. This is useful if you are developing closed open extensions and want to share within your projects.

Extensions installed by Composer are stored in the `BasePath/vendor` directory, where `BasePath` refers to the application's [base path](#). Because Composer is a dependency manager, when it installs a package, it will also install all its dependent packages.

For example, to install the `yiisoft/yii2-image` extension, modify your `composer.json` like the following:

```
{  
    // ...  
  
    "require": {  
        // ... other dependencies  
  
        "yiisoft/yii2-image": "*"br/>    }  
}
```

After the installation, you should see the directory `yiisoft/yii2-image` under `BasePath/vendor`. You should also see another directory `image/image` which contains the installed dependent package.

Info: The `yiisoft/yii2-image` is a core extension developed and maintained by the Yii developer team. All core extensions are hosted on [Packagist](#) and named like `yiisoft/yii2-xyz`, where `xyz` varies for different extensions.

Now you can use the installed extensions like they are part of your application. The following example shows how you can use the `yii\image\Image` class provided by the `yiisoft/yii2-image` extension:

```
use Yii;  
  
use yii\image\Image;  
  
// generate a thumbnail image  
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)  
  
->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' => 50]);
```

Info: Extension classes are autoloaded by the [Yii class autoloader](#).

Installing Extensions Manually

In some rare occasions, you may want to install some or all extensions manually, rather than relying on Composer. To do so, you should

- 1.download the extension archive files and unpack them in the `vendor` directory.
- 2.install the class autoloaders provided by the extensions, if any.
- 3.download and install all dependent extensions as instructed.

If an extension does not have a class autoloader but follows the [PSR-4 standard](#), you may use the class autoloader provided by Yii to autoload the extension classes. All you need to do is just to declare a [root alias](#) for the extension root directory. For example, assuming you have installed an extension in the directory `vendor/mycompany/myext`, and the extension classes are under the `myext` namespace, then you can include the following code in your application configuration:

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

Creating Extensions

You may consider creating an extension when you feel the need to share with other people your great code. An extension can contain any code you like, such as a helper class, a widget, a module, etc.

It is recommended that you create an extension in terms of a [Composer package](#) so that it can be more easily installed and used by other users, liked described in the last subsection.

Below are the basic steps you may follow to create an extension as a Composer package.

- 1.Create a project for your extension and host it on a VCS repository, such as [github.com](#). The development and maintenance work about the extension should be done on this repository.
- 2.Under the root directory of the project, create a file named `composer.json` as required by Composer. Please refer to the next subsection for more details.
- 3.Register your extension with a Composer repository, such as [Packagist](#), so that other users can find and install your extension using Composer.

composer.json

Each Composer package must have a `composer.json` file in its root directory. The file contains the metadata about the package. You may find complete specification about this file in the [Composer Manual](#). The following example shows the `composer.json` file for the `yiisoft/yii2-imagine` extension:

```
{

    // package name

    "name": "yiisoft/yii2-imagine",

    // package type

    "type": "yii2-extension",

    "description": "The Imagine integration for the Yii framework",

    "keywords": ["yii2", "imagine", "image", "helper"],

    "license": "BSD-3-Clause",

    "support": {

        "issues": "https://github.com/yiisoft/yii2/issues?labels=ext%3Aimagine",

        "forum": "http://www.yiiframework.com/forum/",

        "wiki": "http://www.yiiframework.com/wiki/",

        "irc": "irc://irc.freenode.net/yii",

        "source": "https://github.com/yiisoft/yii2"

    },

    "authors": [

        {

            "name": "Antonio Ramirez",

            "email": "amigo.cobos@gmail.com"

        }

    ],

}
```

```

// package dependencies

"require": {

    "yiisoft/yii2": "*",

    "imagine/imagine": "v0.5.0"

},

// class autoloading specs

"autoload": {

    "psr-4": {

        "yii\\imagine\\": ""

    }

}

}

```

Package Name

Each Composer package should have a package name which uniquely identifies the package among all others. The format of package names is `vendorName/projectName`. For example, in the package name `yiisoft/yii2-imagine`, the vendor name and the project name are `yiisoft` and `yii2-imagine`, respectively.

Do NOT use `yiisoft` as vendor name as it is reserved for use by the Yii core code.

We recommend you prefix `yii2-` to the project name for packages representing Yii 2 extensions, for example, `myname/yii2-mywidget`. This will allow users to more easily tell whether a package is a Yii 2 extension.

Package Type

It is important that you specify the package type of your extension as `yii2-extension` so that the package can be recognized as a Yii extension when being installed.

When a user runs `composer install` to install an extension, the file `vendor/yiisoft/extensions.php` will be automatically updated to include the information about the new extension. From this file, Yii applications can know which extensions are installed (the information can be accessed via `[[yii\base\Application::extensions]]`).

Dependencies

Your extension depends on Yii (of course). So you should list it ([yiisoft/yii2](#)) in the `require` entry in `composer.json`. If your extension also depends on other extensions or third-party libraries, you should list them as well. Make sure you also list appropriate version constraints (e.g. `1.*`, `@stable`) for each dependent package. Use stable dependencies when your extension is released in a stable version.

Most JavaScript/CSS packages are managed using [Bower](#) and/or [NPM](#), instead of Composer. Yii uses the [Composer asset plugin](#) to enable managing these kinds of packages through Composer. If your extension depends on a Bower package, you can simply list the dependency in `composer.json` like the following:

```
{  
    // package dependencies  
  
    "require": {  
        "bower-asset/jquery": ">=1.11.*"  
    }  
}
```

The above code states that the extension depends on the `jquery` Bower package. In general, you can use `bower-asset/PackageName` to refer to a Bower package in `composer.json`, and use `npm-asset/PackageName` to refer to a NPM package. When Composer installs a Bower or NPM package, by default the package content will be installed under the `@vendor/bower/PackageName` and `@vendor/npm/Packages` directories, respectively. These two directories can also be referred to using the shorter aliases `@bower/PackageName` and `@npm/PackageName`.

For more details about asset management, please refer to the [Assets](#) section.

Class Autoloading

In order for your classes to be autoloaded by the Yii class autoloader or the Composer class autoloader, you should specify the `autoload` entry in the `composer.json` file, like shown below:

```
{  
    // ....  
  
    "autoload": {  
        "psr-4": {  
            "yii\\imagine\\": ""  
        }  
    }  
}
```

```
}  
  
}  
  
}
```

You may list one or multiple root namespaces and their corresponding file paths.

When the extension is installed in an application, Yii will create for each listed root namespace an [alias](#) that refers to the directory corresponding to the namespace. For example, the above [autoload](#) declaration will correspond to an alias named [@yii/Imagine](#).

Recommended Practices

Because extensions are meant to be used by other people, you often need to take extra development effort. Below we introduce some common and recommended practices in creating high quality extensions.

Namespaces

To avoid name collisions and make the classes in your extension autoloadable, you should use namespaces and name the classes in your extension by following the [PSR-4 standard](#) or [PSR-0 standard](#).

You class namespaces should start with [vendorName\ExtensionName](#), where [ExtensionName](#) is similar to the project name in the package name except that it should not contain the [yii2-](#) prefix. For example, for the [yiisoft/yii2-imagine](#) extension, we use [yii\Imagine](#) as the namespace its classes.

Do not use [yii](#), [yii2](#) or [yiisoft](#) as vendor name. These names are reserved for use by the Yii core code.

Bootstrapping Classes

Sometimes, you may want your extension to execute some code during the [bootstrapping process](#) stage of an application. For example, your extension may want to respond to the application's [beginRequest](#) event to adjust some environment settings. While you can instruct users of the extension to explicitly attach your event handler in the extension to the [beginRequest](#) event, a better way is to do this automatically.

To achieve this goal, you can create a so-called bootstrapping class by implementing `[[yii\base\BootstrapInterface]]`. For example,

```
namespace myname\mywidget;  
  
use yii\base\BootstrapInterface;  
  
use yii\base\Application;
```

```
class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // do something here
        });
    }
}
```

You then list this class in the `composer.json` file of your extension like follows,

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

When the extension is installed in an application, Yii will automatically instantiate the bootstrapping class and call its `[[yii\base\BootstrapInterface::bootstrap()|bootstrap()]]` method during the bootstrapping process for every request.

Working with Databases

Your extension may need to access databases. Do not assume that the applications that use your extension will always use `Yii::$db` as the DB connection. Instead, you should declare a `db` property for the classes that require DB access. The property will allow users of your extension to customize which DB connection they would like your extension to use. As an example, you may refer to the `[[yii\caching\DbCache]]` class and see how it declares and uses the `db` property.

If your extension needs to create specific DB tables or make changes to DB schema, you should

- provide [migrations](#) to manipulate DB schema, rather than using plain SQL files;
- try to make the migrations applicable to different DBMS;
- avoid using [Active Record](#) in the migrations.

Using Assets

If your extension is a widget or a module, chances are that it may require some [assets](#) to work. For example, a module may display some pages which contain images, JavaScript, and CSS. Because the files of an extension are all under the same directory which is not Web accessible when installed in an application, you have two choices to make the asset files directly accessible via Web:

- ask users of the extension to manually copy the asset files to a specific Web-accessible folder;
- declare an [asset bundle](#) and rely on the asset publishing mechanism to automatically copy the files listed in the asset bundle to a Web-accessible folder.

We recommend you use the second approach so that your extension can be more easily used by other people. Please refer to the [Assets](#) section for more details about how to work with assets in general.

Internationalization and Localization

Your extension may be used by applications supporting different languages! Therefore, if your extension displays content to end users, you should try to [internationalize and localize](#) it. In particular,

- If the extension displays messages intended for end users, the messages should be wrapped into `Yii::t()` so that they can be translated. Messages meant for developers (such as internal exception messages) do not need to be translated.
- If the extension displays numbers, dates, etc., they should be formatted using `[[yii\i18n\Formatter]]` with appropriate formatting rules.

For more details, please refer to the [Internationalization](#) section.

Testing

You want your extension to run flawlessly without bringing problems to other people. To reach this goal, you should test your extension before releasing it to public.

It is recommended that you create various test cases to cover your extension code rather than relying on manual tests. Each time before you release a new version of your extension, you may simply run these test cases to make sure everything is in good shape. Yii provides testing support, which can help you to more easily write unit tests, acceptance tests and functionality tests. For more details, please refer to the [Testing](#) section.

Versioning

You should give each release of your extension a version number (e.g. `1.0.1`). We recommend you follow the [semantic versioning](#) practice when determining what version numbers should be used.

Releasing

To let other people know your extension, you need to release it to public.

If it is the first time you release an extension, you should register it on a Composer repository, such as [Packagist](#). After that, all you need to do is simply creating a release tag (e.g. [v1.0.1](#)) on the VCS repository of your extension and notify the Composer repository about the new release. People will then be able to find the new release, and install or update the extension through the Composer repository.

In the releases of your extension, besides code files you should also consider including the followings to help other people learn about and use your extension:

- A readme file in the package root directory: it describes what your extension does and how to install and use it. We recommend you write it in [Markdown](#) format and name the file as [readme.md](#).
- A changelog file in the package root directory: it lists what changes are made in each release. The file may be written in Markdown format and named as [changelog.md](#).
- An upgrade file in the package root directory: it gives the instructions on how to upgrade from older releases of the extension. The file may be written in Markdown format and named as [upgrade.md](#).
- Tutorials, demos, screenshots, etc.: these are needed if your extension provides many features that cannot be fully covered in the readme file.
- API documentation: your code should be well documented to allow other people more easily read and understand it. You may refer to the [Object class file](#) to learn how to document your code.

Info: Your code comments can be written in Markdown format.

The [yiisoft/yii2-apidoc](#) extension provides a tool for you to generate pretty API documentation based on your code comments.

Info: While not a requirement, we suggest your extension adhere to certain coding styles. You may refer to the [core framework code style](#).

Core Extensions

Yii provides the following core extensions that are developed and maintained by the Yii developer team. They are all registered on [Packagist](#) and can be easily installed as described in the [Using Extensions](#) subsection.

- [yiisoft/yii2-apidoc](#): provides an extensible and high-performance API documentation generator. It is also used to generate the core framework API documentation.
- [yiisoft/yii2-authclient](#): provides a set of commonly used auth clients, such as Facebook OAuth2 client, GitHub OAuth2 client.
- [yiisoft/yii2-bootstrap](#): provides a set of widgets that encapsulate

the [Bootstrap](#) components and plugins.

- [yiisoft/yii2-codeception](#): provides testing support based on [Codeception](#).
- [yiisoft/yii2-debug](#): provides debugging support for Yii applications. When this extension is used, a debugger toolbar will appear at the bottom of every page. The extension also provides a set of standalone pages to display more detailed debug information.
- [yiisoft/yii2-elasticsearch](#): provides the support for using [Elasticsearch](#). It includes basic querying/search support and also implements the [Active Record](#) pattern that allows you to store active records in Elasticsearch.
- [yiisoft/yii2-faker](#): provides the support for using [Faker](#) to generate fake data for you.
- [yiisoft/yii2-gii](#): provides a Web-based code generator that is highly extensible and can be used to quickly generate models, forms, modules, CRUD, etc.
- [yiisoft/yii2-imagine](#): provides commonly used image manipulation functions based on [Imagine](#).
- [yiisoft/yii2-jui](#): provides a set of widgets that encapsulate the [jQuery UI](#) interactions and widgets.
- [yiisoft/yii2-mongodb](#): provides the support for using [MongoDB](#). It includes features such as basic query, Active Record, migrations, caching, code generation, etc.
- [yiisoft/yii2-redis](#): provides the support for using [redis](#). It includes features such as basic query, Active Record, caching, etc.
- [yiisoft/yii2-smarty](#): provides a template engine based on [Smarty](#).
- [yiisoft/yii2-sphinx](#): provides the support for using [Sphinx](#). It includes features such as basic query, Active Record, code generation, etc.
- [yiisoft/yii2-swiftmailer](#): provides email sending features based on [swiftmailer](#).
- [yiisoft/yii2-twig](#): provides a template engine based on [Twig](#).

请求处理： 启动引导（**Bootstrapping**）

启动引导是指：在应用开始解析并处理新接受请求之前，一个预先准备环境的过程。启动引导会在两个地方具体进行：[入口脚本\(Entry Script\)](#) 和 [应用主体 \(application\)](#)。

在[入口脚本](#)里，需注册各个类库的类文件自动加载器（**Class Autoloader**，简称自动加载器）。这主要包括通过其 [autoload.php](#) 文件加载的 **Composer** 自动加载器，以及通过 **Yii** 类加载的 **Yii** 自动加载器。之后，入口脚本会加载应用的 [配置 \(configuration\)](#) 并创建一个 [应用主体](#) 的实例。

在应用主体的构造函数中，会执行以下引导工作：

- 1.调用 `[[yii\base\Application::preInit()|preInit()]]`（预初始化）方法，配置一些高优先级的应用属性，比如 `[[yii\base\Application::basePath|basePath]]` 属性。
- 2.注册`[[yii\base\Application::errorHandler|错误处理器（ErrorHandler）]]`。
- 3.通过给定的应用配置初始化应用的各属性。
- 4.通过调用 `[[yii\base\Application::init()|init()]]`（初始化）方法，它会顺次调用 `[[yii\base\Application::bootstrap()|bootstrap()]]` 从而运行引导组件。
 - 加载扩展清单文件(extension manifest file) `vendor/yiisoft/extensions.php`。
 - 创建并运行各个扩展声明的 [引导组件（bootstrap components）](#)。
 - 创建并运行各个 [应用组件](#) 以及在应用的 [Bootstrap 属性](#)中声明的各个 [模块（modules）组件](#)（如果有）。

因为引导工作必须在处理**每一次**请求之前都进行一遍，因此让该过程尽可能轻量化就异常重要，请尽可能地优化这一步骤。

请尽量不要注册太多引导组件。只有他需要在 HTTP 请求处理的全部生命周期中都作用时才需要使用它。举一个用到它的范例：一个模块需要注册额外的 URL 解析规则，就应该把它列在应用的 [bootstrap 属性](#)之中，这样该 URL 解析规则才能在解析请求之前生效。（译注：换言之，为了性能需要，除了 URL 解析等少量操作之外，绝大多数组件都应该按需加载，而不是都放在引导过程中。）

在生产环境中，可以开启字节码缓存，比如 APC，来进一步最小化加载和解析 PHP 文件所需的时间。

一些大型应用都包含有非常复杂的应用[配置](#)，它们会被分割到许多更小的配置文件中。此时，可以考虑将整个配置数组缓存起来，并在入口脚本创建应用实例之前直接从缓存中加载。

路由(routing)

当[入口脚本](#)在调用 `[[yii\web\Application::run()|run()]]` 方法时，它进行的第一个操作就是解析输入的请求，然后实例化对应的[控制器操作](#)处理这个请求。该过程就被称为[引导路由（routing）](#)。（译注：中文里既是动词也是名词）

解析路由

路由引导的第一步，是把传入请求解析为一个路由。如我们在 [控制器（Controllers）](#) 章节中所描述的那样，路由是一个用于定位控制器操作的地址。这个过程通过 [request](#) 应用组件的 `[[yii\web\Request::resolve()|resolve()]]` 方法实现，该方法会调用 [URL 管理器](#) 进行实质上的请求解析工作。

默认情况下，传入请求会包含一个名为 `r` 的 [GET](#) 参数，它的值即被视为路由。但是如果启用 `[[yii\web\UrlManager::enablePrettyUrl|美化 URL 功能]]`，那么在确定请求的路由时，就会进行更多处理。具体的细节请参考 [URL 的解析与生成](#) 章节。

假使某路由最终实在无法被确定，那么 [request](#) 组件会抛出 `[[yii\web\NotFoundHttpException]]` 异常（译注：大名鼎鼎的 404）。

缺省路由

如果传入请求并没有提供一个具体的路由，（一般这种情况多用于对首页的请求）此时就会启用由 `[[yii\web\Application::defaultRoute]]` 属性所指定的缺省路由。该属性的默认值为 `site/index`，它指向 `site` 控制器的 `index` 操作。你可以像这样在应用配置中调整该属性的值：

```
return [  
    // ...  
    'defaultRoute' => 'main/index',  
];
```

catchAll 路由（全拦截路由）

有时候，你会想要将你的 Web 应用临时调整到维护模式，所有的请求下都会显示相同的信息页。当然，要实现这一点有很多种方法。这里面最简单快捷的方法就是在应用配置中设置下 `[[yii\web\Application::catchAll]]` 属性：

```
return [  
    // ...  
    'catchAll' => ['site/offline'],  
];
```

`catchAll` 属性需要传入一个数组做参数，该数组的第一个元素为路由，剩下的元素会（以名值对的形式）指定绑定于该操作的各个参数。

当设置了 `catchAll` 属性时，它会替换掉所有从输入的请求中解析出来的路由。如果是上文的这种设置，用于处理所有传入请求的操作都会是相同的 `site/offline`。

创建操作

一旦请求路由被确定了，紧接着的步骤就是创建一个“操作（action）”对象，用以响应该路由。

路由可以用里面的斜杠分割成多个组成片段，举个例子，`site/index` 可以分解为 `site` 和 `index` 两部分。每个片段都是指向某一模块（Module）、控制器（Controller）或操作（action）的 ID。

从路由的首个片段开始，应用会经过以下流程依次创建模块（如果有），控制器，以及操作：

1. 设置应用主体为当前模块。
2. 检查当前模块的 `[[yii\base\Module::controllerMap|controller map（控制器映射表）]]` 是否包含当前 ID。如果是，会根据该表中的配置创建一个控制器对象，然后跳到步骤五执行该路由的后续片段。
3. 检查该 ID 是否指向当前模块中 `[[yii\base\Module::modules|modules]]` 属性里的模块列表中的一个模块。如果是，会根据该模块表中的配置创建一个模块对象，然后会以新创建的模块为

环境，跳回步骤二解析下一段路由。

4. 将该 ID 视为控制器 ID，并创建控制器对象。用下个步骤解析路由里剩下的片段。

5. 控制器会在他的 `[[yii\base\Controller::actions()]]` (action map (操作映射表)) 里搜索当前 ID。如果找得到，它会根据该映射表中的配置创建一个操作对象；反之，控制器则会尝试创建一个与该 ID 相对应，由某个 `action` 方法所定义的行内操作 (inline action)。

在上面的步骤里，如果有任何错误发生，都会抛出 `[[yii\web\NotFoundException]]`，指出路由引导的过程失败了。

请求(request)

此部分参加官网地址：<http://www.yiichina.com/guide/2/runtime-requests>

请求(Request)

获取用户请求

PHP 并未提供集中的、统一的界面以获取用户请求，而是分散在 `$_SERVER` `$_POST` 等变量和其他代码中。万能的 Yii 怎么会允许群雄割据这种局面出现呢？他肯定是要一统江湖的。那么对于任何 Yii 应用而言，初始化后第一件正事，就是获取用户请求。这个代码在 `yii\base\Application::run()` 中：

```

1 public function run()
2 {
3     try {
4         $this->state = self::STATE_BEFORE_REQUEST;
5         $this->trigger(self::EVENT_BEFORE_REQUEST);
6
7         $this->state = self::STATE_HANDLING_REQUEST;
8         // 获取用户请求，并进行处理，处理的过程也是产生响应内容的过程
9         $response = $this->handleRequest($this->getRequest());
10
11         $this->state = self::STATE_AFTER_REQUEST;
12         $this->trigger(self::EVENT_AFTER_REQUEST);
13
14         $this->state = self::STATE_SENDING_RESPONSE;
15
16         // 将响应内容发送回用户
17         $response->send();
18
19         $this->state = self::STATE_END;
20
21         return $response->exitStatus;
22
23     } catch (ExitException $e) {
24
25         $this->end($e->statusCode, isset($response) ? $response : null);
26         return $e->statusCode;
27
28     }
29 }
30
31 }

```

上面的代码主要看注释的两个地方，聪明的读者朋友们一定都猜出来了，`$this->getRequest()` 就是用于获取用户请求的嘛。

其实这是一个 **getter**，用于获取 **Application** 的 **request** 组件 (component)。Yii 用这个组件来代表用户请求，他承载着所有的用户输入信息。

我们知道，Yii 应用有命令行(Console)应用和 Web 应用之分。因此，这个 **Request** 类其实涉及到了以下的类：

- yii\base\Request Request 类基类
- yii\console\Request 表示 Console 应用的 Request
- yii\web\Request 表示 Web 应用的 Request

下面我们逐一进行讲解。

基类 Request

基类是对 Console 应用和 Web 应用 Request 的抽象，他仅仅定义了两个属性和一个虚函数：

```

1 abstract class Request extends Component
2 {

```

```

3 // 属性 scriptFile, 用于表示入口脚本
4 private $_scriptFile;
5
6 // 属性 isConsoleRequest, 用于表示是否是命令行应用
7 private $_isConsoleRequest;
8
9 // 虚函数, 要求子类来实现
10 // 这个函数的功能主要是为了把 Request 解析成路由和相应的参数
12 abstract public function resolve();
13
14 // isConsoleRequest 属性的 getter 函数
15 // 使用 PHP_SAPI 常量判断当前应用是否是命令行应用
16 public function getIsConsoleRequest()
17 {
18     // 一切 PHP_SAPI 不为 'cli' 的, 都不是命令行
19     return $this->_isConsoleRequest !== null ?
20         $this->_isConsoleRequest : PHP_SAPI === 'cli';
21 }
22
23
24
25 // isConsoleRequest 属性的 setter 函数
26 public function setIsConsoleRequest($value)
27 {
28     $this->_isConsoleRequest = $value;
29 }
30
31
32 // scriptFile 属性的 getter 函数
33 // 通过 $_SERVER['SCRIPT_FILENAME'] 来获取入口脚本名
34 public function getScriptFile()
35 {
36     if ($this->_scriptFile === null) {
37         if (isset($_SERVER['SCRIPT_FILENAME'])) {
38             $this->setScriptFile($_SERVER['SCRIPT_FILENAME']);
39         } else {
40             throw new InvalidConfigException(
41                 'Unable to determine the entry script file path.');
42         }
43     }
44 }
45
46
47
48
49 return $this->_scriptFile;
50 }
51
52 // scriptFile 属性的 setter 函数
53 public function setScriptFile($value)
54 {
55     $scriptFile = realpath(Yii::getAlias($value));
56     if ($scriptFile !== false && is_file($scriptFile)) {
57         $this->_scriptFile = $scriptFile;
58     } else {
59         throw new InvalidConfigException(
60             'Unable to determine the entry script file path.');
61     }
62 }

```

```
}  
}
```

yii\base\Request 通过 **getter** 和 **setter** 提供了两个可读写的属性， `isConsoleRequest` 和 `scriptFile` 。 同时，要求子类实现一个 `resolve()` 方法。

基类的代码相对简单，主要涉及到 PHP 的一些知识，如 `PHP_SAPI` `$_SERVER['SCRIPT_FILENAME']` 等， 读者朋友们可以通过搜索引擎或 PHP 手册了解下相关的知识，相信上面的代码难不倒你们的。

命令行应用 Request

命令行应用 Request 由 `yii\console\Request` 负责实现，相比较于 `yii\base\Request` 稍有丰富：

```
1 class Request extends \yii\base\Request  
2 {  
3     // 属性 params, 用于表示命令行参数  
4     private $_params;  
5  
6     // params 属性的 getter 函数  
7     // 通过 $_SERVER['argv'] 来获取命令行参数  
8     public function getParams()  
9     {  
10         if (!isset($this-> params)) {  
11             if (isset($_SERVER['argv'])) {  
12                 $this-> params = $_SERVER['argv'];  
13  
14                 // 删除数组的第一个元素，这个元素是 PHP 脚本名。  
15                 // 因此，属性 params 中全部是参数，不带脚本名  
16                 array_shift($this-> params);  
17             } else {  
18                 $this-> params = [];  
19             }  
20         }  
21     }  
22  
23  
24     return $this-> params;  
25 }  
26  
27 // params 属性的 setter 函数  
28 public function setParams($params)  
29 {  
30     $this-> params = $params;  
31 }  
32  
33  
34 // 父类虚函数的实现  
35 public function resolve()  
36 {  
37     // 获取全部的命令行参数  
38     $rawParams = $this->getParams();  
39  
40     // 第一个命令行参数作为路由  
41  
42
```



```

        if (isset($rawParams[0])) {
            $route = $rawParams[0];
            array_shift($rawParams);
        } else {
            $route = '';
44     }
45
46     $params = [];
47
48     // 遍历剩余的全部命令行参数
49     foreach ($rawParams as $param) {
50
51         // 正则匹配每一个参数
52         if (preg_match('/^--(\w+)(=.*?)?$/ ', $param, $matches)) {
53
54             // 参数名
55             $name = $matches[1];
56
57             // yii\console\Application::OPTION_APPCONFIG = 'appconfig'
58             if ($name !== Application::OPTION_APPCONFIG) {
59                 $params[$name] = isset($matches[3]) ? $matches[3] : true;
60             }
61
62             // 无名参数，直接作为参数值
63         } else {
64             $params[] = $param;
65         }
66     }
67
68     return [$route, $params];
69 }
}

```

相比较于 `yii\base\Request`，`yii\console\Request` 提供了一个 `params` 属性，该属性以数组形式保存了入口脚本 `Yii` 的命令行参数。这是通过 `$_SERVER['argv']` 获取的。注意 `params` 属性不保存入口脚本名。入口脚本名由基类的 `scriptName` 属性保存。同时，`yii\console\Request` 还实现了父类的 `resolve()` 虚函数，这个函数主要做了这么几件事：

- 将 `params` 属性的第一个元素作为路由。如果入口脚本未提供任何参数，也即 `params` 是个空数组，那么将路由置为一个空字符串。
- 遍历 `params` 中剩余的参数，使用正则匹配 `Yii` 应用的参数名和参数值，看看是不是 `--参数名=参数值` 形式。其中，以 `--` 打头的任意字母、数字、下划线的组合，就是参数名。紧跟参数名的 `=` 后面的内容，则为参数值。对于仅有参数名，没有参数值的，视参数值为 `true`。
- 如果正则匹配不成功，则将这个命令行参数作为 `Yii` 应用的一个无名参数的值。
- 如果第二步中的参数名为 `appconfig` 则忽略该参数，`Console Application` 会专门针对该参数进行处理。

- 上面步骤中的参数和参数值，被保存进一个数组中。数组的键表示参数名，数组的值表示参数值。
- 最终 `resolve()` 返回一个数组，第一个元素是一个表示路由的字符串，第二元素则是参数数组。该方法由 `Application` 在处理 `Request` 时调用。

关于 `appconfig` 参数的问题，只要在调用 `yii` 时，指定了 `appconfig` 参数，就表明不使用默认的参数配置文件，而使用该参数所指定的配置文件。相关的代码在 `yii\console\Application` 中：

```
// 定义一个常量
const OPTION_APPCONFIG = 'appconfig';

1
2 // yii\console\Application 类的构造函数
public function __construct($config = [])
3
4 {
5     // 重点看这句，会调用 loadConfig() 成员函数
6     $config = $this->loadConfig($config);
7     parent::__construct($config);
8 }
9
10 // 如果指定的配置文件存在，那么返回其配置数组
11 // 否则，返回构造函数调用时的数组
protected function loadConfig($config)
12
13 {
14     if (!empty($_SERVER['argv'])) {
15         // 设定了一个字符串 "--appconfig="
16         $option = '--' . self::OPTION_APPCONFIG . '=';
17
18         // 遍历所有命令行参数，看看能不能找到上面说的这个字符串
19         foreach ($_SERVER['argv'] as $param) {
20             if (strpos($param, $option) !== false) {
21                 // 截取参数值部分
22                 $path = substr($param, strlen($option));
23                 if (!empty($path) && is_file($file = Yii::getAlias($path))) {
24                     // 将指定文件的内容引入进来
25                     return require($file);
26                 } else {
27                     die("The configuration file does not exist: $path\n");
28                 }
29             }
30         }
31     }
32     return $config;
33 }
34
35
36
37
38
39
```

讲完了 `Request` 基类和命令行应用的 `Request` 只是热身而已，接下来要讲的 `Web` 应用 `Request` 才是重头。毕竟最主要的，还是 `Web` 开发嘛。考虑到 `Web Request` 的内容较多，还是单独成 [Web 应用 Request](#) 来讲吧。

Web 应用 Request

前面 [请求\(Request\)](#) 部分我们讲了用户请求的基础知识和命令行应用的 Request，接下来继续讲 Web 应用的 Request。

Web 应用 Request 由 `yii\web\Request` 实现，这个类的代码将近 1400 行，主要是一些功能的封装罢了，原理上没有很复杂的东西。只是涉及到许多 HTTP 的有关知识，读者朋友们可以自行查看相关的规范文档，如 [HTTP 1.1 协议](#)，[CGI 1.1 规范](#) 等。

同时，Yii 大量引用了 `$_SERVER`，具体可以查看 [PHP 文档关于\\$_SERVER的内容](#)，此外，还涉及到 PHP 运行于不同的环境和模式下的一些细微差别。这些内容比较细节，不影响大局，但是很影响理解，不过没关系，我们在涉及到的时候，会点一点。

请求的方法

根据 [HTTP 1.1 协议](#)，HTTP 的请求可以有：GET, POST, PUT 等 8 种方法 (Request Method)。除了用不到的 CONNECT 外，Yii 支持全部的 HTTP 请求方法。

要获取当前用户请求的方法，可以使用 `yii\web\Request::getMethod()`

```
1 // 返回当前请求的方法，请留意方法名称是大小写敏感的，按规范应转换为大写字母
2 public function getMethod()
3 {
4     // $this->methodParam 默认值为 '_method'
5     // 如果指定 $_POST['_method']，表示使用 POST 请求来模拟其他方法的请求。
6     // 此时 $_POST['_method'] 即为所模拟的请求类型。
7     if (isset($_POST[$this->methodParam])) {
8         return strtoupper($_POST[$this->methodParam]);
9     }
10    // 或者使用 $_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE'] 的值作为方法名。
11    } elseif (isset($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE'])) {
12        return strtoupper($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE']);
13    }
14    // 或者使用 $_SERVER['REQUEST_METHOD'] 作为方法名，未指定时，默认为 GET 方法
15    } else {
16        return isset($_SERVER['REQUEST_METHOD']) ?
17            strtoupper($_SERVER['REQUEST_METHOD']) : 'GET';
18    }
19 }
```

这个方法使用了 3 种方法来获取当前用户的请求，优先级从高到低依次为：

- 当使用 POST 请求来模拟其他请求时，以 `$_POST['_method']` 作为当前请求的方法；
- 否则，如果存在 `X_HTTP_METHOD_OVERRIDE` HTTP 头时，以该 HTTP 头所指定的方法作为请求方法，如 `X-HTTP-Method-Override: PUT` 表示该请求所要执行的是 PUT 方法；
- 如果 `X_HTTP_METHOD_OVERRIDE` 不存在，则以 `REQUEST_METHOD` 的值作为当前请求的方法。如果连 `REQUEST_METHOD` 也不存在，则视该请求是一个 GET 请求。

前面两种方法，主要是针对一些只支持 GET 和 POST 等有限方法的 User Agent 而设计的。

其中第一种方法是从 **Ruby on Rails** 中借鉴过来的，通过在发送 **POST** 请求时，加入一个 `$_POST['_method']` 的隐藏字段，来表示所要模拟的方法，如 **PUT**，**DELETE** 等。这样，就可以使得这些功能有限的 **User Agent** 也可以正常与服务器交互。这种方法胜在简便，随手就来。

第二种方法则是使用 `X_HTTP_METHOD_OVERRIDE` **HTTP** 头的办法来指定所使用的请求类型。这种方法胜在直接明了，约定俗成，更为规范、合理。

至于 `REQUEST_METHOD` 是 [CGI 1.1 规范](#) 所定义的环境变量，专门用来表明当前请求方法的。上面的代码只是在未指定时默认为 **GET** 请求罢了。

当然，我们在开发过程中，其实并不怎么在乎当前的用户请求是什么类型的请求，我们更在乎是不是某一类型的请求。比如，对于同一个 **URL** 地

址 `http://api.digpage.com/post/123`，如果是正常的 **GET** 请求，应该是查看编号为 **123** 的文章的意思。但是如果是一个 **DELETE** 请求，则是表示删除编号为 **123** 的文章的意思。我们在开发中，很可能就会这么写：

```
1 if ($app->request->isDelete()){
2     $post->delete();
3 } else {
4     $post->view();
5 }
```

上面的代码只是一个示意，与实际编码是有一定出入的，主要看判断分支的用法。就是判断请求是否是某一特定类型的请求。这些判断在实际开发中，是很常用的。于是 **Yii** 为我们封装了许多方法专门用于执行这些判断：

- `getIsAjax()` 是否是 **AJAX** 请求，这其实不是 **HTTP** 请求方法，但是实际使用上，这个是用得最多的。
- `getIsDelete()` 是否是 **DELETE** 请求
- `getIsFlash()` 是否是 **Adobe Flash** 或 **Adobe Flex** 发出的请求，这其实也不是 **HTTP** 请求方法。
- `getIsGet()` 是否是一个 **GET** 请求
- `getIsHead()` 是否是一个 **HEAD** 请求
- `getIsOptions()` 是否是一个 **OPTIONS** 请求
- `getIsPatch()` 是否是 **PATCH** 请求
- `getIsPjax()` 是否是一个 **PJAX** 请求，这也并非是 **HTTP** 请求方法。
- `getIsPost()` 是否是一个 **POST** 请求
- `getIsPut()` 是否是一个 **PUT** 请求

上面 10 个方法请留意其中有 3 个并未是 **HTTP** 请求方法，主要是用于特定 **HTTP** 请求类型 (**AJAX**、**Flash**、**PJAX**) 的判断。

除了这 3 个之外的其余 7 个方法，正好对应于 **HTTP 1.1** 协议定义的 7 个方法。而 **CONNECT** 方法由于 **Web** 开发在用不到，主要用于 **HTTP** 代理，因此，**Yii** 也就没有为其设计一个所谓的 `isConnect()` 了，这是无用功。

上面的 10 个方法，再加一开始说的 `getMethod()` 一共是 11 个方法，按照我们在 [属性 \(Property\)](#) 部分所说的，这相当于定义了 11 个只读属性。我们以其中几个为例，看看具体实现：

```
1 // 这个 SO EASY，啥也不说了，Yii 实现的 7 个 HTTP 方法都是这个路子。
2 public function getIsOptions()
3 {
4     // 注意在 getMethod() 时，输出的是全部大写的字符串
5     return $this->getMethod() === 'OPTIONS';
6 }
7 // AJAX 请求是通过 X_REQUESTED_WITH 消息头来判断的
8 public function getIsAjax()
9 {
10    // 注意这里的 XMLHttpRequest 没有全部大写
11    return isset($_SERVER['HTTP_X_REQUESTED_WITH']) &&
12           $_SERVER['HTTP_X_REQUESTED_WITH'] === 'XMLHttpRequest';
13 }
14 // PJAX 请求是 AJAX 请求的一种，增加了 X_PJAX 消息头的定义
15 public function getIsPjax()
16 {
17     return $this->getIsAjax() && !empty($_SERVER['HTTP_X_PJAX']);
18 }
19 // HTTP_USER_AGENT 消息头中包含 'Shockwave' 或 'Flash' 字眼的（不区分大小写），
20 // 就认为是 FLASH 请求
21 public function getIsFlash()
22 {
23     return isset($_SERVER['HTTP_USER_AGENT'])
24            && (strpos($_SERVER['HTTP_USER_AGENT'], 'Shockwave') !== false
25              || strpos($_SERVER['HTTP_USER_AGENT'], 'Flash') !== false);
26 }
```

上面提到的 AJAX、PJAX、FLASH 请求比较特殊，并非是 HTTP 协议所规定的请求类型，但是在实现中是会使用到的。比如，对于一个请求，在非 AJAX 时，需要整个页面返回给客户端，而在 AJAX 请求时，只需要返回页面片段即可。

这些特殊请求是通过特殊的消息头实现的，具体的可以自行搜索相关的定义和规范。至于那 7 个 HTTP 方法的判断，摆明了是同一个路子，换瓶不换酒，`getMethod()` 前人栽树，他们后人乘凉。

请求的参数

在实际开发中，开发者如果需要引用 `request`，最常见的情况是为了获取请求参数，以便作相应处理。PHP 有众所周知的 `$_GET` 和 `$_POST` 等。相应地，Yii 提供了一系列的方法用于获取请求参数：

```
1 // 用于获取 GET 参数，可以指定参数名和默认值
2 public function get($name = null, $defaultValue = null)
3 {
4     if ($name === null) {
5         return $this->getQueryParams();
6     } else {
```

```

7     return $this->getQueryParam($name, $defaultValue);
8 }
9 }
10
11 // 用于获取所有的 GET 参数
12 // 所有的 GET 参数保存在 $_GET 或 $this->_queryParams 中。
13 public function getQueryParams()
14 {
15     if ($this->_queryParams === null) {
16         // 请注意这里并未使用 $this->_queryParams = $_GET 进行缓存。
17         // 说明一旦指定了 $_queryParams 则 $_GET 会失效。
18         return $_GET;
19     }
20     return $this->_queryParams;
21 }
22
23 // 根据参数名获取单一的 GET 参数，不存在时，返回指定的默认值
24 public function getQueryParam($name, $defaultValue = null)
25 {
26     $params = $this->getQueryParams();
27     return isset($params[$name]) ? $params[$name] : $defaultValue;
28 }
29
30 // 类似于 get()，用于获取 POST 参数，也可以指定参数名和默认值
31 public function post($name = null, $defaultValue = null)
32 {
33     if ($name === null) {
34         return $this->getBodyParams();
35     } else {
36         return $this->getBodyParam($name, $defaultValue);
37     }
38 }
39
40 // 根据参数名获取单一的 POST 参数，不存在时，返回指定的默认值
41 public function getBodyParam($name, $defaultValue = null)
42 {
43     $params = $this->getBodyParams();
44     return isset($params[$name]) ? $params[$name] : $defaultValue;
45 }
46
47 // 获取所有 POST 参数，所有 POST 参数保存在 $this->_bodyParams 中
48 public function getBodyParams()
49 {
50     if ($this->_bodyParams === null) {
51         // 如果是使用 POST 请求模拟其他请求的
52         if (isset($_POST[$this->methodParam])) {
53             $this->_bodyParams = $_POST;
54
55             // 将 $_POST['method'] 删掉，剩余的 $_POST 就是了
56             unset($this->_bodyParams[$this->methodParam]);
57             return $this->_bodyParams;
58         }
59     }
60     // 获取 Content Type

```

```

57
58
59 // 对于 'application/json; charset=UTF-8', 得到的是 'application/json'
60 $contentType = $this->getContentType();
61 if (($pos = strpos($contentType, ';')) !== false) {
62     $contentType = substr($contentType, 0, $pos);
63 }
64
65 // 根据 Content Type 选择相应的解析器对请求体进行解析
66 if (isset($this->parsers[$contentType])) {
67
68     // 创建解析器实例
69     $parser = Yii::createObject($this->parsers[$contentType]);
70     if (!$parser instanceof RequestParserInterface) {
71         throw new InvalidConfigException(
72             "The '$contentType' request parser is invalid.
73             It must implement the yii\\web\\RequestParserInterface.");
74     }
75
76     // 将请求体解析到 $this->_bodyParams
77     $this->_bodyParams = $parser->parse($this->getRawBody(), $contentType);
78
79     // 如果没有与 Content Type 对应的解析器, 使用通用解析器
80 } elseif (isset($this->parsers['*'])) {
81     $parser = Yii::createObject($this->parsers['*']);
82     if (!$parser instanceof RequestParserInterface) {
83         throw new InvalidConfigException(
84             "The fallback request parser is invalid.
85             It must implement the yii\\web\\RequestParserInterface.");
86     }
87     $this->_bodyParams = $parser->parse($this->getRawBody(),
88         $contentType);
89
90     // 连通用解析器也没有
91     // 看看是不是 POST 请求, 如果是, PHP 已经将请求参数放到 $_POST 中了, 直接用就 OK 了
92 } elseif ($this->getMethod() === 'POST') {
93     $this->_bodyParams = $_POST;
94
95     // 以上情况都不是, 那就使用 PHP 的 mb_parse_str() 进行解析
96 } else {
97     $this->_bodyParams = [];
98     mb_parse_str($this->getRawBody(), $this->_bodyParams);
99 }
100 }
101
102 return $this->_bodyParams;
103 }
104
105
106

```

在上面的代码中, 将所有的请求参数划分为两类, 一类是包含在 URL 中的, 称为查询参数

(Query Parameter)，或 GET 参数。另一类是包含在请求体中的，需要根据请求体的内容类型 (Content Type) 进行解析，称为 POST 参数。

其中，`get()`，`getQueryParams()` 和 `getQueryParam()` 用于获取查询参数：

- `get()` 用于获取 GET 参数，可以指定所要获取的特定参数的参数名，在这个参数名不存在时，可以指定默认值。当不指定参数名时，获取所有的 GET 参数。具体功能是由下面 2 个函数来实现的。
- `getQueryParams()` 用于获取所有的 GET 参数。这些参数的内容，保存在 `$_GET` 或 `$this->_queryParams` 中。优先使用 `$this->_queryParams` 的。
- `getQueryParam()` 对应于 `get()` 用于获取特定的 GET 参数的情况。

而 `post()`，`getPostParams()` 和 `getPostParam()` 用于获取 POST 参数：

- `post()` 与 `get()` 类似，可以指定所要获取的特定参数的参数名，在这个参数名不存在时，可以指定默认值。当不指定参数名时，获取所有的 POST 参数。具体功能是由下面 2 个函数来实现的。
- `getPostParam()` 用于通过参数名获取特定的 POST 参数，需要调用 `getPostParams()` 获取所有的 POST 参数。
- `getPostParams()` 用于获取所有的 POST 参数。

上面稍微复杂点的，可能就是 `getPostParams()` 了，我们就稍稍剖析下 Yii 是怎么解析 POST 参数的。先讲讲这个方法所涉及的一些东东：内容类型、请求解析器、请求体。

内容类型 (Content-Type)

在 `getPostParams()` 中，需要先获取请求体的内容类型，然后采用相应的解析器对内容进行解析。

获取内容类型，使用 `getContentType()`

```
1 public function getContentType()  
2 {  
3     if (isset($_SERVER["CONTENT_TYPE"])) {  
4         return $_SERVER["CONTENT_TYPE"];  
5     } elseif (isset($_SERVER["HTTP_CONTENT_TYPE"])) {  
6         return $_SERVER["HTTP_CONTENT_TYPE"];  
7     }  
8  
9     return null;  
10 }
```

根据 [CGI 1.1 规范](#)，内容类型由 `CONTENT_TYPE` 环境变量来表示。而根据 [HTTP 1.1 协议](#)，内容类型则是放在 `CONTENT_TYPE` 头部中，然后由 PHP 赋值给 `$_SERVER['HTTP_CONTENT_TYPE']`。这里一般没有冲突，因此发现哪个用哪个，就怕客户端没有给出（这种情况返回 `null`）。

请求解析器

在 `getPostParams()` 中，根据不同的 **Content Type** 创建了相应的内容解析器对请求体进行解析。 `yii\web\Request` 使用成员变量 `public $parsers` 来保存一系列的解析器。 这个变量在配置时进行指定：

```
1 'request' => [  
2     ...  
3     'parsers' => [  
4         'application/json' => 'yii\web\JsonParser',  
5     ],  
6 ]
```

`$parsers` 是一个数组，数组的键是 **Content Type**，如 `application/json` 之类。而数组的值则是对应于特定 **Content Type** 的解析器，如 `yii\web\JsonParser`。这也是 Yii 实现的唯一一个现成的 **Parser**，其他 **Content-Type**，需要开发者自己写了。

而且，可以以 `*` 为键指定一个解析器。那么该解析器将在一个 **Content Type** 找不到任何匹配的解析器后被使用。

`yii\web\JsonParser` 其实很简单：

```
namespace yii\web;  
  
1 use yii\base\InvalidParamException;  
2 use yii\helpers\Json;  
3  
4 // 所有的解析器都要实现 RequestParserInterface  
5 // 这个接口也只是要求实现 parse() 方法  
6 class JsonParser implements RequestParserInterface  
7 {  
8     public $asArray = true;  
9     public $throwException = true;  
10  
11     // 具体实现 parse()  
12     public function parse($rawBody, $contentType)  
13     {  
14         try {  
15             return Json::decode($rawBody, $this->asArray);  
16         } catch (InvalidParamException $e) {  
17             if ($this->throwException) {  
18                 throw new BadRequestHttpException(  
19                     'Invalid JSON data in request body: '  
20                     . $e->getMessage(), 0, $e);  
21             }  
22         }  
23     }  
24  
25     return null;  
26 }  
27 }  
28 }
```

这里使用 `yii\helpers\Json::decode()` 对请求体进行解析。这个 `yii\helpers\Json` 是个辅助类，专门用于处理 **JSON** 格式数据。具体的内容我们这里就不做讲解了，只需要了解这里可以将 **JSON** 格式数据解析出来就 OK 了，学有余力的读者朋友可以自己看看代码。

请求体

在 `yii\web\Request::getBodyParams()` 和 `yii\web\RequestParserInterface::parse()` 中，我们可以看到，需要将请求体传入 `parse()` 进行解析，且请求体由 `yii\web\Request::getRawBody()` 可得。

`yii\web\Request::getRawBody()`:

```
1 public function getRawBody()  
2 {  
3     if ($this->_rawBody === null) {  
4         $this->_rawBody = file_get_contents('php://input');  
5     }  
6  
7     return $this->_rawBody;  
8 }
```

这个方法使用了 `php://input` 来获取请求体，这个 `php://input` 有这么几个特点：

- `php://input` 是个只读流，用于获取请求体。
- `php://input` 是返回整个 HTTP 请求中，除去 HTTP 头部的全部原始内容，而不管是什么 Content Type（或称为编码方式）。相比较之下，`$_POST` 只支持 `application/x-www-form-urlencoded` 和 `multipart/form-data-encoded` 两种 Content Type。其中前一种就是简单的 HTML 表单以 `method="post"` 提交时的形式，后一种主要是用于上传文档。因此，对于诸如 `application/json` 等 Content Type，这往往是在 AJAX 场景下使用，那么使用 `$_POST` 得到的是空的内容，这时就必须使用 `php://input`。
- 相比较于 `$HTTP_RAW_POST_DATA`，`php://input` 无需额外地在 `php.ini` 中激活 `always-populate-raw-post-data`，而且对于内存的压力也比较小。
- 当编码方式为 `multipart/form-data-encoded` 时，`php://input` 是无效的。这种情况一般为上传文档。这种情况可以使用传统的 `$_FILES` 或者 `yii\web\UploadedFile`。

请求的头部

`yii\web\Request` 使用一个成员变量 `private $_headers` 来存储请求头。而这个 `$_header` 其实是一个 `yii\web/HeaderCollection`，这是一个集合类的基本数据结构，实现了 SPL 的 `IteratorAggregate`，`ArrayAccess` 和 `Countable` 等接口。因此，这个集合可以进行迭代、像数组一样进行访问、可被用于 `count()` 函数等。

这个数据结构相对简单，我们就不展开占用篇幅了。我们要讲的是怎么获取请求的头部。这个是由 `yii\web\Request::getHeaders()` 来实现的：

```
1 public function getHeaders()  
2 {  
3     if ($this->_headers === null) {  
4         // 实例化为一个 HeaderCollection  
5         $this->_headers = new HeaderCollection;  
6  
7         // 使用 getAllheaders() 获取请求头部，以数组形式返回
```

```

        if (function_exists('getallheaders')) {
            $headers = getallheaders();
8
9            // 使用 http_get_request_headers() 获取请求头部，以数组形式返回
10        } elseif (function_exists('http_get_request_headers')) {
11            $headers = http_get_request_headers();
12
13            // 使用 $_SERVER 数组获取头部
14        } else {
15            foreach ($_SERVER as $name => $value) {
16
17                // 针对所有 $_SERVER['HTTP_*'] 元素
18                if (strncmp($name, 'HTTP ', 5) === 0) {
19                    // 将 HTTP_HEADER_NAME 转换成 Header-Name 的形式
20                    $name = str_replace(' ', '-',
21                        ucwords(strtolower(str_replace(' ', '',
22                            substr($name, 5)))));
23                    $this->headers->add($name, $value);
24                }
25            }
26        }
27        return $this->headers;
28    }
29
30    // 将数组形式的请求头部变成集合的元素
31    foreach ($headers as $name => $value) {
32        $this->headers->add($name, $value);
33    }
34    return $this->headers;
35}
36
37}
38
39}
40
41return $this->headers;
42}

```

这里用 3 种方法来尝试获取请求的头部：

- `getallheaders()`，这个方法仅在将 PHP 作为 Apache 的一个模块运行时有效。
- `http_get_request_headers()`，要求 PHP 启用 HTTP 扩展。
- `$_SERVER` 数组的方法，需要遍历整个数组，并将所有以 `HTTP_*` 元素加入到集合中去。并且，要将所有 `HTTP_HEADER_NAME` 转换成 `Header-Name` 的形式。

就是根据不同的 PHP 环境，采用有效的方法来获取请求头部，如此而已。

请求的解析

我们前面就说过了，无论是命令行应用还是 Web 应用，他们的请求都要实现接口要求的 `resolve()`，以便明确这个用户请求的路由和参数。下面就是 `yii\web\Request::resolve()` 的代码：

```

1 public function resolve()
2 {
3     // 使用 urlManager 来解析请求

```

```

4     $result = Yii::$app->getUrlManager()->parseRequest($this);
5     if ($result !== false) {
6         list($route, $params) = $result;
7
8         // 将解析出来的参数与 $_GET 参数进行合并
9         $_GET = array_merge($_GET, $params);
10
11        return [$route, $_GET];
12    } else {
13        throw new NotFoundException(Yii::t('yii', 'Page not found.'));
14    }
15 }

```

看着很简单吧？这才几行，还没有 `getBodyParams()` 的代码多呢。

虽然简单，但是有一个细节我们要留意，就是在解析出路由信息和参数的时候，会把参数的内容加入到 `$_GET` 中去，这是合理的。

比如，对于 `http://www.digpage.com/post/view/100` 这个 `100` 在路由规则中，其实定义为一个参数。其原始的形式应当是 `http://www.digpage.com/index.php?r=post/view&id=100`。你说该不该把 `id = 100` 重新写回 `$_GET` 去？至于路由规则的内容，可以看看 [路由 \(Route\)](#) 的内容。

从这个 `resolve()` 是看不出来解析过程的复杂的，这个 `yii\web\Request::resolve()` 是个没担当的家伙，他把解析过程推给了 `urlManager`。那我们就顺藤摸瓜，一睹这个 `yii\web\UrlManager::parseRequest()` 吧：

```

1 public function parseRequest($request)
2 {
3     // 启用了 enablePrettyUrl 的情况
4     if ($this->enablePrettyUrl) {
5
6         // 获取路径信息
7         $pathInfo = $request->getPathInfo();
8
9         // 依次使用所有路由规则来解析当前请求
10        // 一旦有一个规则适用，后面的规则就没有被调用的机会了
11        foreach ($this->rules as $rule) {
12            if (($result = $rule->parseRequest($this, $request)) !== false) {
13                return $result;
14            }
15        }
16    }
17
18    // 所有路由规则都不适用，又启用了 enableStrictParsing，
19    // 那只能返回 false 了。
20    if ($this->enableStrictParsing) {
21        return false;
22    }
23
24    // 所有路由规则都不适用，幸好还没启用 enableStrictParsing，
25    // 那就用默认的解析逻辑
26
27
28    Yii::trace(
29        'No matching URL rules. Using default URL parsing logic.',
30        __METHOD__);
31 }

```

```

        // 配置时所定义的 fake suffix, 诸如 ".html" 等
        $suffix = (string) $this->suffix;

        if ($suffix !== "" && $pathInfo !== "") {
32         // 这个分支的作用在于确保 $pathInfo 不能仅仅是包含一个 ".html"。
33
34         $n = strlen($this->suffix);
35         // 留意这个 -$n 的用法
36         if (substr_compare($pathInfo, $this->suffix, -$n, $n) === 0) {
37             $pathInfo = substr($pathInfo, 0, -$n);
38
39         // 仅包含 ".html" 的 $pathInfo 要之何用? 掐死算了。
40         if ($pathInfo === "") {
41             return false;
42         }
43
44         // 后缀没匹配上
45     } else {
46         return false;
47     }
48
49     return [$pathInfo, []];
50
51     // 没有启用 enablePrettyUrl 的情况, 那就更简单了,
52     // 直接使用默认的解析逻辑就 OK 了
53 } else {
54     Yii::trace(
55         'Pretty URL not enabled. Using default URL parsing logic.',
56         __METHOD__);
57     $route = $request->getQueryParam($this->routeParam, "");
58     if (is_array($route)) {
59         $route = "";
60     }
61
62     return [(string) $route, []];
63 }
64 }
65 }
66 }
67 }

```

从上面代码中可以看到, urlManager 是按这么一个顺序来解析用户请求的:

- 先判断是否启用了 enablePrettyUrl, 如果没启用, 所有的路由和参数信息都在 URL 的查询参数中, 很简单就可以处理了。
- 通常都会启用 enablePrettyUrl, 由于路由和参数信息部分或全部变成了 URL 路径。经过了美化, 使得 URL 看起来更友好, 但化妆品总是比清水芙蓉要烧银子, 解析起来就有点费功夫了。
- 既然路由和参数信息变成了 URL 路径, 那么就先从 URL 路径下手获取路径信息。
- 然后依次使用已经定义好的路由规则对当前请求进行解析, 一旦有一个规则适用, 后续的路由规则就不会起作用了。

- 然后再对配置的 .html 等 fake suffix 进行处理。

这一过程中，有两个重点，一个是获取路径信息，另一个就是使用路由规则对请求进行解析。下面我们依次进行讲解。

获取路径信息

在大多数情况下，我们还是会启用 `enablePrettyUrl` 的，特别是在产品环境下。那么从上面的代码来看，`yii\web\Request::getPathInfo()` 的调用就不可避免。其实涉及到获取路径信息的方法有很多，都在 `yii\web\Request` 中，这里暴露出来的，只是一个 `getPathInfo()`，相关的方法有：

```
1 // 这个方法其实是调用 resolvePathInfo() 来获取路径信息的
2 public function getPathInfo()
3 {
4     if ($this->_pathInfo === null) {
5         $this->_pathInfo = $this->resolvePathInfo();
6     }
7     return $this->_pathInfo;
8 }
9
10 // 这个才是重点
11 protected function resolvePathInfo()
12 {
13     // 这个 getUrl() 调用的是 resolveRequestUri() 来获取当前的 URL
14     $pathInfo = $this->getUrl();
15
16     // 去除 URL 中的查询参数部分，即 ? 及之后的内容
17     if (($pos = strpos($pathInfo, '?')) !== false) {
18         $pathInfo = substr($pathInfo, 0, $pos);
19     }
20
21     // 使用 PHP urldecode() 进行解码，所有 %## 转成对应的字符，+ 转成空格
22     $pathInfo = urldecode($pathInfo);
23
24     // 这个正则列举了各种编码方式，通过排除这些编码，来确认是 UTF-8 编码
25     // 出处可参考 http://w3.org/International/questions/qa-forms-utf-8.html
26     if (!preg_match('%^(?:
27         [\x09\x0A\x0D\x20-\x7E]          # ASCII
28         | [\xC2-\xDF][\x80-\xBF]        # non-overlong 2-byte
29         | \xE0[\xA0-\xBF][\x80-\xBF]    # excluding overlongs
30         | [\xE1-\xEC\xEE\xEF][\x80-\xBF]{2} # straight 3-byte
31         | \xED[\x80-\x9F][\x80-\xBF]    # excluding surrogates
32         | \xF0[\x90-\xBF][\x80-\xBF]{2} # planes 1-3
33         | [\xF1-\xF3][\x80-\xBF]{3}      # planes 4-15
34         | \xF4[\x80-\x8F][\x80-\xBF]{2} # plane 16
35     )*$%xs', $pathInfo)) {
36         $pathInfo = utf8_encode($pathInfo);
37     }
38 }
```

```

        // 获取当前脚本的 URL
        $scriptUrl = $this->getScriptUrl();

        // 获取 Base URL
        $baseUrl = $this->getBaseUrl();
47
48
49         if (strpos($pathInfo, $scriptUrl) === 0) {
50             $pathInfo = substr($pathInfo, strlen($scriptUrl));
51         } elseif ($baseUrl === '' || strpos($pathInfo, $baseUrl) === 0) {
52             $pathInfo = substr($pathInfo, strlen($baseUrl));
53         } elseif (isset($_SERVER['PHP_SELF']) && strpos($_SERVER['PHP_SELF'],
54             $scriptUrl) === 0) {
55             $pathInfo = substr($_SERVER['PHP_SELF'], strlen($scriptUrl));
56         } else {
57             throw new InvalidConfigException(
58                 'Unable to determine the path info of the current request.');
59         }
60
61         // 去除 $pathInfo 前的 '/'
62         if ($pathInfo[0] === '/') {
63             $pathInfo = substr($pathInfo, 1);
64         }
65
66         return (string) $pathInfo;
67     }

```

从 `resolvePathInfo()` 来看，需要调用到的方法有 `getUrl()` `resolveRequestUri()` `getScriptUrl()` `getBaseUrl()` 等，这些都是与路径信息密切相关的，让我们分别都看一看。

Request URI

`yii\web\Request::getUrl()` 用于获取 Request URI 的，实际上这只是一个属性的封装， 实际的代码是在 `yii\web\Request::resolveRequestUri()` 中：

```

1 // 这个其实调用的是 resolveRequestUri() 来获取当前 URL
2 public function getUrl()
3 {
4     if ($this->_url === null) {
5         $this->_url = $this->resolveRequestUri();
6     }
7
8     return $this->_url;
9 }
10
11 // 这个方法用于获取当前 URL 的 URI 部分，即主机或主机名之后的内容，包括查询参数。
12 // 这个方法参考了 Zend Framework 1 的部分代码，通过各种环境下的 HTTP 头来获取 URI。
13 // 返回值为 $_SERVER['REQUEST_URI'] 或 $_SERVER['HTTP_X_REWRITE_URL']。
14 // 或 $_SERVER['ORIG_PATH_INFO'] + $_SERVER['QUERY_STRING']。
15 // 即，对于 http://www.digpage.com/index.html?helloworld，
16 // 得到 URI 为 index.html?helloworld
17 protected function resolveRequestUri()
18 {
19     // 使用了开启了 ISAPI_Rewrite 的 IIS
20     if (isset($_SERVER['HTTP_X_REWRITE_URL'])) {

```



```

    $requestUri = $ _SERVER['HTTP_X_REWRITE_URL'];

    // 一般情况，需要去掉 URL 中的协议、主机、端口等内容
22 } elseif (isset($ _SERVER['REQUEST_URI'])) {
23     $requestUri = $ _SERVER['REQUEST_URI'];
24
25     // 如果 URI 不为空或以 '/' 打头，则去除 http:// 或 https:// 直到第一个 /
26     if ($requestUri !== "" && $requestUri[0] !== '/') {
27         $requestUri = preg_replace('/^(http|https):\\V[^\\]+/i',
28             "", $requestUri);
29     }
30
31     // IIS 5.0, PHP 以 CGI 方式运行，需要把查询参数接上
32 } elseif (isset($ _SERVER['ORIG_PATH_INFO'])) {
33     $requestUri = $ _SERVER['ORIG_PATH_INFO'];
34     if (!empty($ _SERVER['QUERY_STRING'])) {
35         $requestUri .= '?' . $ _SERVER['QUERY_STRING'];
36     }
37 } else {
38     throw new InvalidConfigurationException('Unable to determine the request URI.');
```

从上面的代码我们可以知道，Yii 针对不同的环境下，PHP 的不同表现形式，通过一些分支判断， 给出一个统一的路径名或文件名。 辛辛苦苦那么多，其实就是为了消除不同环境对于开发的影响，使开发者可以更加专注于核心工作。

其实，作为一个开发框架，无论是哪种语言、用于哪个领域， 都需要为开发者提供在各种环境下的都表现一致的编程界面。 这也是开发者可以放心使用的基础条件，如果在使用框架之后， 开发者仍需要考虑各种环境下会怎么样怎么样，那么这个框架注定短命。

这里有必要点一点涉及到的几个 `$_SERVER` 变量。这里面提到的，读者朋友可以自行阅读 [PHP 文档关于 `\$\_SERVER` 的内容](#)， 也可以看看 [CGI 1.1 规范的内容](#)。

REQUEST_URI

由 HTTP 1.1 协议定义，指访问某个页面的 URI，去除开头的协议、主机、端口等信息。如 `http://www.digpage.com:8080/index.php/foo/bar?queryParams`， `REQUEST_URI` 为 `/index.php/foo/bar?queryParams`。

X-REWRITE-URL

当使用以开启了 `ISAPI_Rewrite` 的 IIS 作为服务器时，`ISAPI_Rewrite` 会在未对原始 URI 作任何修改前， 将原始的 `REQUEST_URI` 以 `X-REWRITE-URL` HTTP 头保存起来。

PATH_INFO

CGI 1.1 规范所定义的环境变量。 从形式上看，
`http://www.digpage.com:8080/index.php</PATH_INFO>?queryParams`。 它是整个 URI 中，

在脚本标识之后、查询参数 ? 之前的部分。对于 Apache，需要设置 AcceptPathInfo On，且在一个 URL 没有 </PATH_INFO> 部分的时候，PATH_INFO 无效。特殊的情况，如 http://www.digpage.com/index.php/，PATH_INFO 为 /。而对于 Nginx，则需要设置：

```
fastcgi_split_path_info ^(.+?\.php)(/.*)$;  
fastcgi_param PATH_INFO $fastcgi_path_info;  
ORIG_PATH_INFO
```

在 PHP 文档中对它的解释语焉不详，“指未经 PHP 处理过的原始的 PATH_INFO”。这个在 Apache 和 Nginx 需要配置一番才行，但一般用不到，已经有 PATH_INFO 可以用了嘛。而在 IIS 中则有点怪，对于 http://www.digpage.com/index.php/ ORIG_PATH_INFO 为/index.php/；对于 http://www.digapge.com/index.php ORIG_PATH_INFO 为 /index.php。

根据上面这些背景知识，再来看 resolveRequestUri() 就简单了：

- 最广泛的情况，应当是使用 REQUEST_URI 来获取。但是 resolveRequestUri() 却先使用 X-REWRITE-URL，这是为了防止 REQUEST_URI 被 rewrite。
- 其次才是使用 REQUEST_URI，这对于绝大多数情况是完全够用的了。
- 但 REQUEST_URI 毕竟只是规范的要求，Web 服务器很有可能店大欺客、另立山头，我们又不是第一次碰见了是吧？所以，Yii 使用了平时比较少用到的 ORIG_PATH_INFO。
- 最后，按照规范要求进行规范化，该去头的去头，该续尾的续尾。去除主机信息段和查询参数段后，就大功告成了。

入口脚本路径

yii\web\Request::getScriptUrl() 用于获取入口脚本的相对路径，也涉及到不同环境下 PHP 的不同表现。我们还是先从代码入手：

```
1 // 这个方法用于获取当前入口脚本的相对路径  
2 public function getScriptUrl()  
3 {  
4     if ($this->scriptUrl === null) {  
5         // $this->getScriptFile() 用的是 $_SERVER['SCRIPT_FILENAME']  
6         $scriptFile = $this->getScriptFile();  
7         $scriptName = basename($scriptFile);  
8  
9         // 下面的这些判断分支代码，为各主流 PHP framework 所用。  
10        // Yii, Zend, Symfony 等都是大同小异。  
11        if (basename($_SERVER['SCRIPT_NAME']) === $scriptName) {  
12            $this->scriptUrl = $_SERVER['SCRIPT_NAME'];  
13        } elseif (basename($_SERVER['PHP_SELF']) === $scriptName) {  
14            $this->scriptUrl = $_SERVER['PHP_SELF'];  
15        }  
16    }
```

```

        } elseif (isset($_SERVER['ORIG_SCRIPT_NAME']) &&
            basename($_SERVER['ORIG_SCRIPT_NAME']) === $scriptName) {
17         $this->_scriptUrl = $_SERVER['ORIG_SCRIPT_NAME'];
18     } elseif (($pos = strpos($_SERVER['PHP_SELF'], '/' . $scriptName))
19         !== false) {
20         $this->_scriptUrl = substr($_SERVER['SCRIPT_NAME'], 0, $pos)
21             . '/' . $scriptName;
22     } elseif (!empty($_SERVER['DOCUMENT_ROOT'])
23         && strpos($scriptFile, $_SERVER['DOCUMENT_ROOT']) === 0) {
24         $this->_scriptUrl = str_replace('\\', '/',
25             str_replace($_SERVER['DOCUMENT_ROOT'], "", $scriptFile));
26     } else {
27         throw new InvalidConfigException(
28             'Unable to determine the entry script URL.');
```

上面的代码涉及到了一些环境问题，点一点，大家了解下就 OK 了：

SCRIPT_FILENAME

当前脚本的实际物理路径，比如 `/var/www/digpage.com/frontend/web/index.php`，或 WIN 平台的 `D:\www\digpage.com\frontend\web\index.php`。以 Nginx 为例，一般情况下，`SCRIPT_FILENAME` 有以下配置项：

```

fastcgi_split_path_info ^(.+?\.php)(/.*)$;
# 使用 document root 来得到物理路径
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name
```

SCRIPT_NAME

CGI 1.1 规范所定义的环境变量，用于标识 CGI 脚本（而非脚本的输出），如 `http://www.digapge.com/path/index.php` 中的 `/path/index.php`。仍以 Nginx 为例，`SCRIPT_NAME` 一般情况下有 `fastcgi_param SCRIPT_NAME $fastcgi_script_name` 的设置。绝大多数情况下，使用 `SCRIPT_NAME` 即可获取当前脚本。

PHP_SELF

`PHP_SELF` 是 PHP 自己实现的一个 `$_SERVER` 变量，是相对于文档根目录（document root）而言的。对于 `http://www.digpage.com/path/index.php?queryParams`，`PHP_SELF` 为 `/path/index.php`。一般 `SCRIPT_NAME` 与 `PHP_SELF` 无异。但是，在 `PHP.INI` 中，如 `cgi.fix_pathinfo=1`（默认即为 1）时，对于形如 `http://www.digpage.com/path/index.php/post/view/123`，则 `PHP_SELF` 为 `/path/index.php/post/view/123`。而根据 CGI 1.1 规范，`SCRIPT_NAME` 仅为 `/path/index.php`，至于剩余的 `/post/view/123` 则为 `PATH_INFO`。

ORIG_SCRIPT_NAME

当 PHP 以 CGI 模式运行时，默认会对一些环境变量进行调整。首当其冲的，就是 `SCRIPT_NAME` 的内容会变成 `php.cgi` 等二进制文件，而不再是 CGI 脚本文件。当然，设置 `cgi.fix_pathinfo=0` 可以关闭这一默认行为。但这导致的副作用比较大，影响范围过大，不宜使用。但天无绝人之路，九死之地总留一线生机，那就是 `ORIG_SCRIPT_NAME`，他保留了调整前 `SCRIPT_NAME` 的内容。也就是说，在 CGI 模式下，可以使用 `ORIG_SCRIPT_NAME` 来获取想要的 `SCRIPT_NAME`。请留意使用 `ORIG_SCRIPT_NAME` 前一定要先确认它是否存在。

交待完这些背景知识后，我们再来看看 `yii\web\Request::getScriptUrl()` 的逻辑：

- 先调用 `yii\web\Request::getScriptFile()`，通过 `basename($_SERVER['SCRIPT_FILENAME'])` 获取脚本文件名。一般都是我们的入口脚本 `index.php`。
- 绝大多数情况下，`base($_SERVER['SCRIPT_NAME'])` 是与第一步获取的 `index.php` 相同的。如果这样的话，则认为这个 `SCRIPT_NAME` 就是我们所要的脚本 URL。

这也是规范的定义。但是既然称为规范，说明并非事实。事实是由 Web 服务器来实现的，也就是说 Web 服务器可能进行修改。

另外，对于运行于 CGI 模式的 PHP 而言，使用 `SCRIPT_NAME` 也无法获得脚本名。

- 那么我们转而向 `PHP_SELF` 求助，这个是 PHP 内部实现的，不受 Web 服务器的影响。一般这个 `PHP_SELF` 也是可堪一用的，但也不是放之四海而皆准，对于带有 `PATH_INFO` 的 URI，`basename()` 获取的并不是脚本名。
- 于是我们再转向 `ORIG_SCRIPT_NAME` 求助，如果 PHP 是运行于 CGI 模式，那么就可行。
- 再不成功，可能 PHP 并非运行于 CGI 模式（否则第 4 步就可以成功），且 URI 中带有 `PATH_INFO`（否则第二步就可以成功）。对于这种情形，`PHP_SELF` 的前面一截就是我们要的脚本 URL。
- 万一以上情况都不符合，说明当前 PHP 运行的环境诡异莫测。那只能寄希望于将 `SCRIPT_FILENAME` 中前面那截可能是 Document Root 的部分去掉，余下的作为脚本 URL 了。前提是要有 Document Root，且 `SCRIPT_FILENAME` 前面的部分可以匹配上。

Base Url

获取路径信息的最后一个相关方法，就是 `yii\web\Request::getBaseUrl()`：

```
1 // 获取 Base Url
2 public function getBaseUrl()
3 {
4     if ($this->_baseUrl === null) {
5         // 用上面的脚本路径的父目录，再去除末尾的 \ 和 /
6         $this->_baseUrl = rtrim(dirname($this->getScriptUrl()), '\\');
7     }
8     return $this->_baseUrl;
9 }
```

这个 **Base Url** 很简单，相信聪明如你肯定一目了然，我就不浪费篇幅了。

好了，上面就是 `yii\web\Request::resolve()` 中有关获取路径信息的内容。下一步就是使用路由规则去解析当前请求了。

使用路由规则解析

上面这么多有关从请求获取路径信息的内容，其实只完成了请求解析的第一步而已。接下来，`urlManager` 就要遍历所有的路由规则来解析当前请求，直到有一个规则适用为止。路由规则层面对于请求的解析，我们在 [路由\(Route\)](#) 的 [解析 URL](#) 部分已经讲得很清楚了。而 `urlManager`

响应(response)

当应用完成处理一个[请求](#)后，会生成一个`[[yii\web\Response|response]]`响应对象并发送给终端用户。响应对象包含的信息有 HTTP 状态码，HTTP 头和主体内容等，网页应用开发的最终目的本质上就是根据不同的请求构建这些响应对象。

在大多数情况下主要处理继承自 `[[yii\web\Response]]` 的 [response 应用组件](#)，尽管如此，Yii 也允许你创建你自己的响应对象并发送给终端用户，这方面后续会阐述。

在本节，将会描述如何构建响应和发送给终端用户。

状态码

构建响应时，最先应做的是标识请求是否成功处理的状态，可通过设置 `[[yii\web\Response::statusCode]]` 属性，该属性使用一个有效的 [HTTP 状态码](#)。例如，为标识处理已被处理成功，可设置状态码为 200，如下所示：

```
Yii::$app->response->statusCode = 200;
```

尽管如此，大多数情况下不需要明确设置状态码，因为 `[[yii\web\Response::statusCode]]` 状态码默认为 200，如果需要指定请求失败，可抛出对应的 HTTP 异常，如下所示：

```
throw new \yii\web\NotFoundHttpException;
```

当[错误处理器](#)捕获到一个异常，会从异常中提取状态码并赋值到响应，对于上述的 `[[yii\web\NotFoundHttpException]]` 对应 HTTP 404 状态码，以下为 Yii 预定义的 HTTP 异常：

- `[[yii\web\BadRequestHttpException]]`: status code 400.
- `[[yii\web\ConflictHttpException]]`: status code 409.
- `[[yii\web\ForbiddenHttpException]]`: status code 403.
- `[[yii\web\GoneHttpException]]`: status code 410.
- `[[yii\web\MethodNotAllowedHttpException]]`: status code 405.
- `[[yii\web\NotAcceptableHttpException]]`: status code 406.

- [[yii\web\NotFoundHttpException]]: status code 404.
- [[yii\web\ServerErrorHttpException]]: status code 500.
- [[yii\web\TooManyRequestsHttpException]]: status code 429.
- [[yii\web\UnauthorizedHttpException]]: status code 401.
- [[yii\web\UnsupportedMediaTypeHttpException]]: status code 415.

如果想抛出的异常不在如上列表中，可创建一个[[yii\web\HttpException]]异常，带上状态码抛出，如下：

```
throw new \yii\web\HttpException(402);
```

HTTP 头部

可在 `response` 组件中操控[[yii\web\Response::headers|header collection]]来发送 HTTP 头部信息，例如：

```
$headers = Yii::$app->response->headers;

// 增加一个 Pragma 头，已存在的 Pragma 头不会被覆盖。

$headers->add('Pragma', 'no-cache');

// 设置一个 Pragma 头. 任何已存在的 Pragma 头都会被丢弃

$headers->set('Pragma', 'no-cache');

// 删除 Pragma 头并返回删除的 Pragma 头的值到数组

$values = $headers->remove('Pragma');
```

补充: 头名称是大小写敏感的，在[[yii\web\Response::send()]]方法调用前新注册的头信息并不会发送给用户。

响应主体

大多是响应应有一个主体存放你想要显示给终端用户的内容。

如果已有格式化好的主体字符串，可赋值到响应的[[yii\web\Response::content]]属性，例如：

```
Yii::$app->response->content = 'hello world!';
```

如果在发送给终端用户之前需要格式化，应设置 [[yii\web\Response::format|format]] 和 [[yii\web\Response::data|data]] 属性，[[yii\web\Response::format|format]] 属性指定 [[yii\web\Response::data|data]] 中数据格式化后的样式，例如：

```
$response = Yii::$app->response;  
$response->format = \yii\web\Response::FORMAT_JSON;  
$response->data = ['message' => 'hello world'];
```

Yii 支持以下可直接使用的格式，每个实现了[[yii\web\ResponseFormatterInterface|formatter]] 类，可自定义这些格式器或通过配置[[yii\web\Response::formatters]] 属性来增加格式器。

- [[yii\web\Response::FORMAT_HTML|HTML]]: 通过 [[yii\web\HtmlResponseFormatter]] 来实现.
- [[yii\web\Response::FORMAT_XML|XML]]: 通过 [[yii\web\XmlResponseFormatter]]来实现.
- [[yii\web\Response::FORMAT_JSON|JSON]]: 通过 [[yii\web\JsonResponseFormatter]]来实现.
- [[yii\web\Response::FORMAT_JSONP|JSONP]]: 通过 [[yii\web\JsonResponseFormatter]]来实现.

上述响应主体可明确地被设置，但是在大多数情况下是通过 [操作](#) 方法的返回值隐式地设置，常用场景如下所示：

```
public function actionIndex()  
{  
  
    return $this->render('index');  
}
```

上述的 `index` 操作返回 `index` 视图渲染结果，返回值会被 `response` 组件格式化后发送给终端用户。

因为响应格式默认为[[yii\web\Response::FORMAT_HTML|HTML]]，只需要在操作方法中返回一个字符串， 如果想使用其他响应格式，应在返回数据前先设置格式，例如：

```
public function actionInfo()  
{  
  
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;  
  
    return [  
  
        'message' => 'hello world',  
  
        'code' => 100,  
  
    ];  
}
```

如上所述，触雷使用默认的 `response` 应用组件，也可创建自己的响应对象并发送给终端用户，可在操作方法中返回该响应对象，如下所示：

```
public function actionInfo()
```

```

{
    return \Yii::createObject([
        'class' => 'yii\web\Response',
        'format' => \yii\web\Response::FORMAT_JSON,
        'data' => [
            'message' => 'hello world',
            'code' => 100,
        ],
    ]);
}

```

注意: 如果创建你自己的响应对象, 将不能在应用配置中设置 `response` 组件, 尽管如此, 可使用 [依赖注入](#) 应用通用配置到你新的响应对象。

浏览器跳转

浏览器跳转依赖于发送一个 `Location` HTTP 头, 因为该功能通常被使用, Yii 提供对它提供了特别的支持。

可调用 `[[yii\web\Response::redirect()]]` 方法将用户浏览器跳转到一个 URL 地址, 该方法设置合适的带指定 URL 的 `Location` 头并返回它自己为响应对象, 在操作的方法中, 可调用缩写版 `[[yii\web\Controller::redirect()]]`, 例如:

```

public function actionOld()
{
    return $this->redirect('http://example.com/new', 301);
}

```

在如上代码中, 操作的方法返回 `redirect()` 方法的结果, 如前所述, 操作的方法返回的响应对象会被当总响应发送给终端用户。

除了操作方法外, 可直接调用 `[[yii\web\Response::redirect()]]` 再调用 `[[yii\web\Response::send()]]` 方法来确保没有其他内容追加到响应中。

```
\Yii::$app->response->redirect('http://example.com/new', 301)->send();
```

补充: `[[yii\web\Response::redirect()]]` 方法默认会设置响应状态码为 `302`, 该状态码会告诉浏览器请求的资源 临时 放在另一个 URI 地址上, 可传递一个 `301` 状态码告知浏览器请求的资源已经 永久 重定向到新的 URI 地址。

如果当前请求为 `AJAX` 请求, 发送一个 `Location` 头不会自动使浏览器跳转, 为解决这个问题,

`[[yii\web\Response::redirect()]]` 方法设置一个值为要跳转的 URL 的 **X-Redirect** 头，在客户端可编写 JavaScript 代码读取该头部值然后让浏览器跳转对应的 URL。

补充: Yii 配备了一个 **yii.js** JavaScript 文件提供常用 JavaScript 功能，包括基于 **X-Redirect** 头的浏览器跳转，因此，如果你使用该 JavaScript 文件 (通过 `[[yii\web\YiiAsset]]` 资源包注册)，就不需要编写 AJAX 跳转的代码。

发送文件

和浏览器跳转类似，文件发送是另一个依赖指定 HTTP 头的功能，Yii 提供方法集合来支持各种文件发送需求，它们对 HTTP 头都有内置的支持。

•

这些方法都将响应对象作为返回值，如果要发送的文件非常大，应考虑使用 `[[yii\web\Response::sendStreamAsFile()]]` 因为它更节约内存，以下示例显示在控制器操作中如何发送文件：

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

如果不是在操作方法中调用文件发送方法，在后面还应调用 `[[yii\web\Response::send()]]` 没有其他内容追加到响应中。

```
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

一些浏览器提供特殊的名为 **X-Sendfile** 的文件发送功能，原理为将请求跳转到服务器上的文件，Web 应用可在服务器发送文件前结束，为使用该功能，可调用 `[[yii\web\Response::xSendFile()]]`，如下简要列出一些常用 Web 服务器如何启用 **X-Sendfile** 功能：

- Apache: [X-Sendfile](#)
- Lighttpd v1.4: [X-LIGHTTPD-send-file](#)
- Lighttpd v1.5: [X-Sendfile](#)
- Nginx: [X-Accel-Redirect](#)
- Cherokee: [X-Sendfile](#) and [X-Accel-Redirect](#)

发送响应

在 `[[yii\web\Response::send()]]` 方法调用前响应中的内容不会发送给用户，该方法默认在 `[[yii\base\Application::run()]]` 结尾自动调用，尽管如此，可以明确调用该方法强制立即发送响应。

`[[yii\web\Response::send()]]` 方法使用以下步骤来发送响应：

1. 触发 `[[yii\web\Response::EVENT_BEFORE_SEND]]` 事件。

- 2.调用 `[[yii\web\Response::prepare()]]` 来格式化 `[[yii\web\Response::data|response data]]` 为 `[[yii\web\Response::content|response content]]`.
- 3.触发 `[[yii\web\Response::EVENT_AFTER_PREPARE]]` 事件.
- 4.调用 `[[yii\web\Response::sendHeaders()]]` 来发送注册的 HTTP 头
- 5.调用 `[[yii\web\Response::sendContent()]]` 来发送响应主体内容
- 6.触发 `[[yii\web\Response::EVENT_AFTER_SEND]]` 事件.

一旦`[[yii\web\Response::send()]]` 方法被执行后，其他地方调用该方法会被忽略， 这意味着一旦响应发出后，就不能再追加其他内容。

如你所见`[[yii\web\Response::send()]]` 触发了几个实用的事件，通过响应这些事件可调整或包装响应。

Sessions 和 Cookies

[译注：Session 中文翻译为会话，Cookie 有些翻译成小甜饼，不贴切，两个单词保留英文] Sessions 和 cookies 允许数据在多次请求中保持， 在纯 PHP 中，可以分别使用全局变量`$_SESSION` 和 `$_COOKIE` 来访问，Yii 将 session 和 cookie 封装成对象并增加一些功能， 可通过面向对象方式访问它们。

Sessions

和 [请求](#) 和 [响应](#)类似， 默认可通过为`[[yii\web\Session]]` 实例的 `session` [应用组件](#) 来访问 sessions。

开启和关闭 Sessions

可使用以下代码来开启和关闭 session。

```
$session = Yii::$app->session;
```

```
// 检查 session 是否开启
```

```
if ($session->isActive) ...
```

```
// 开启 session
```

```
$session->open();
```

```
// 关闭 session
```

```
$session->close();
```

```
// 销毁 session 中所有已注册的数据
```

```
$session->destroy();
```

多次调用`[[yii\web\Session::open()|open()]]` 和`[[yii\web\Session::close()|close()]]` 方法并不会产生错误， 因为方法内部会先检查 `session` 是否已经开启。

访问 Session 数据

To access the data stored in session, you can do the following: 可使用如下方式访问 `session` 中的数据:

```
$session = Yii::$app->session;
```

```
// 获取 session 中的变量值，以下用法是相同的:
```

```
$language = $session->get('language');
```

```
$language = $session['language'];
```

```
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;
```

```
// 设置一个 session 变量，以下用法是相同的:
```

```
$session->set('language', 'en-US');
```

```
$session['language'] = 'en-US';
```

```
$_SESSION['language'] = 'en-US';
```

```
// 删除一个 session 变量，以下用法是相同的:
```

```
$session->remove('language');
```

```
unset($session['language']);
```

```
unset($_SESSION['language']);
```

```
// 检查 session 变量是否已存在，以下用法是相同的:
```

```
if ($session->has('language')) ...
```

```
if (isset($session['language'])) ...
```

```
if (isset($_SESSION['language'])) ...
```

// 遍历所有 session 变量，以下用法是相同的：

```
foreach ($session as $name => $value) ...
```

```
foreach ($_SESSION as $name => $value) ...
```

补充: 当使用 `session` 组件访问 `session` 数据时候，如果 `session` 没有开启会自动开启，这和通过 `$_SESSION` 不同，`$_SESSION` 要求先执行 `session_start()`。

当 `session` 数据为数组时，`session` 组件会限制你直接修改数据中的单元项，例如：

```
$session = Yii::$app->session;
```

// 如下代码不会生效

```
$session['captcha']['number'] = 5;
```

```
$session['captcha']['lifetime'] = 3600;
```

// 如下代码会生效：

```
$session['captcha'] = [  
    'number' => 5,  
    'lifetime' => 3600,  
];
```

// 如下代码也会生效：

```
echo $session['captcha']['lifetime'];
```

可使用以下任意一个变通方法来解决这个问题：

```
$session = Yii::$app->session;
```

// 直接使用 `$_SESSION` (确保 `Yii::$app->session->open()` 已经调用)

```
$_SESSION['captcha']['number'] = 5;
```

```
$_SESSION['captcha']['lifetime'] = 3600;
```

// 先获取 `session` 数据到一个数组，修改数组的值，然后保存数组到 `session` 中

```

$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// 使用 ArrayObject 数组对象代替数组
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// 使用带通用前缀的键来存储数组
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;

```

为更好的性能和可读性，推荐最后一种方案，也就是不用存储 **session** 变量为数组，而是将每个数组项变成有相同键前缀的 **session** 变量。

自定义 Session 存储

`[[yii\web\Session]]` 类默认存储 **session** 数据为文件到服务器上，Yii 提供以下 **session** 类实现不同的 **session** 存储方式：

- `[[yii\redis\Session]]`: 存储 **session** 数据到以 [redis](#) 作为存储媒介中
-

所有这些 **session** 类支持相同的 API 方法集，因此，切换到不同的 **session** 存储介质不需要修改项目使用 **session** 的代码。

注意：如果通过 `$_SESSION` 访问使用自定义存储介质的 **session**，需要确保 **session** 已经用 `[[yii\web\Session::open()]]` 开启，这是因为在该方法中注册自定义 **session** 存储处理器。

学习如何配置和使用这些组件类请参考它们的 API 文档，如下为一个示例 显示如何在应用配置中配置 `[[yii\web\DbSession]]` 将数据表作为 **session** 存储介质。

```

return [

    'components' => [

        'session' => [

```

```

        'class' => 'yii\web\DbSession',

        // 'db' => 'mydb', // 数据库连接的应用组件 ID，默认为'db'.

        // 'sessionTable' => 'my_session', // session 数据表名，默认为'session'.

    ],

],

];

```

也需要创建如下数据库表来存储 **session** 数据：

```

CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,

    expire INTEGER,

    data BLOB
)

```

其中'BLOB' 对应你选择的数据库管理系统的 BLOB-type 类型，以下一些常用数据库管理系统的 BLOB 类型：

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

注意：根据 `php.ini` 设置的 `session.hash_function`，你需要调整 `id` 列的长度，例如，如果 `session.hash_function=sha256`，应使用长度为 **64** 而不是 **40** 的 `char` 类型。

Flash 数据

Flash 数据是一种特别的 **session** 数据，它一旦在某个请求中设置后，只会在下次请求中有效，然后该数据就会自动被删除。常用于实现只需显示给终端用户一次的信息，如用户提交一个表单后显示确认信息。

可通过 `session` 应用组件设置或访问 `session`，例如：

```

$session = Yii::$app->session;

// 请求 #1

// 设置一个名为"postDeleted" flash 信息

```

```
$session->setFlash('postDeleted', 'You have successfully deleted your post.');
```

```
// 请求 #2
```

```
// 显示名为"postDeleted" flash 信息
```

```
echo $session->getFlash('postDeleted');
```

```
// 请求 #3
```

```
// $result 为 false，因为 flash 信息已被自动删除
```

```
$result = $session->hasFlash('postDeleted');
```

和普通 session 数据类似，可将任意数据存储为 flash 数据。

当调用[[yii\web\Session::setFlash()]]时，会自动覆盖相同名的已存在的任何数据， 为将数据追加到已存在的相同名 flash 中，可改为调用[[yii\web\Session::addFlash()]]。 例如：

```
$session = Yii::$app->session;
```

```
// 请求 #1
```

```
// 在名称为"alerts"的 flash 信息增加数据
```

```
$session->addFlash('alerts', 'You have successfully deleted your post.');
```

```
$session->addFlash('alerts', 'You have successfully added a new friend.');
```

```
$session->addFlash('alerts', 'You are promoted.');
```

```
// 请求 #2
```

```
// $alerts 为名为'alerts'的 flash 信息，为数组格式
```

```
$alerts = $session->getFlash('alerts');
```

注意：不要在相同名称的 flash 数据中使用[[yii\web\Session::setFlash()]]的同时也使用[[yii\web\Session::addFlash()]]， 因为后一个防范会自动将 flash 信息转换为数组以使新的 flash 数据可追加进来，因此， 当你调用[[yii\web\Session::getFlash()]]时，会发现有时获取到一个数组，有时获取到一个字符串， 取决于你调用这两个方法的顺序。

Cookies

Yii 使用 [[yii\web\Cookie]]对象来代表每个 cookie，[[yii\web\Request]] 和 [[yii\web\Response]]

通过名为'cookies'的属性维护一个 cookie 集合，前者的 cookie 集合代表请求提交的 cookies， 后者的 cookie 集合表示发送给用户的 cookies。

读取 Cookies

当前请求的 cookie 信息可通过如下代码获取：

```
// 从 "request"组件中获取 cookie 集合(yii\web\CookieCollection)

$cookies = Yii::$app->request->cookies;


// 获取名为 "language" cookie 的值，如果不存在，返回默认值"en"

$language = $cookies->getValue('language', 'en');


// 另一种方式获取名为 "language" cookie 的值

if (($cookie = $cookies->get('language')) !== null) {

    $language = $cookie->value;

}


// 可将 $cookies 当作数组使用

if (isset($cookies['language'])) {

    $language = $cookies['language']->value;

}


// 判断是否存在名为"language" 的 cookie

if ($cookies->has('language')) ...

if (isset($cookies['language'])) ...
```

发送 Cookies

You can send cookies to end users using the following code: 可使用如下代码发送 cookie 到终端用户：

```
// 从"response"组件中获取 cookie 集合(yii\web\CookieCollection)

$cookies = Yii::$app->response->cookies;
```

```
// 在要发送的响应中添加一个新的 cookie

$cookies->add(new \yii\web\Cookie([

    'name' => 'language',

    'value' => 'zh-CN',

]));

// 删除一个 cookie

$cookies->remove('language');

// 等同于以下删除代码

unset($cookies['language']);
```

除了上述例子定义的 `[[yii\web\Cookie::name|name]]` 和 `[[yii\web\Cookie::value|value]]` 属性 `[[yii\web\Cookie]]` 类也定义了其他属性来实现 cookie 的各种信息，如 `[[yii\web\Cookie::domain|domain]]`, `[[yii\web\Cookie::expire|expire]]` 可配置这些属性到 cookie 中并添加到响应的 cookie 集合中。

注意: 为安全起见 `[[yii\web\Cookie::httpOnly]]` 被设置为 `true`，这可减少客户端脚本访问受保护 cookie（如果浏览器支持）的风险，更多详情可阅读 [httpOnly wiki article](#) for more details.

Cookie 验证

在上两节中，当通过 `request` 和 `response` 组件读取和发送 cookie 时，你会喜欢扩展的 cookie 验证的保障安全功能，它能 使 cookie 不被客户端修改。该功能通过给每个 cookie 签发一个哈希字符串来告知服务端 cookie 是否在客户端被修改， 如果被修改，通过 `request` 组件的 `[[yii\web\Request::cookies|cookie collection]]` cookie 集合访问不到该 cookie。

注意: Cookie 验证只保护 cookie 值被修改，如果一个 cookie 验证失败，仍然可以通过 `$_COOKIE` 来访问该 cookie， 因为这是第三方库对未通过 cookie 验证自定义的操作方式。

Cookie 验证默认启用，可以设置 `[[yii\web\Request::enableCookieValidation]]` 属性为 `false` 来禁用它，尽管如此，我们强烈建议启用它。

注意: 直接通过 `$_COOKIE` 和 `setcookie()` 读取和发送的 Cookie 不会被验证。

当使用 cookie 验证，必须指定 `[[yii\web\Request::cookieValidationKey]]`，它是用来生成上述的哈希值， 可通过在应用配置中配置 `request` 组件。

```
return [
```



```
'components' => [  
  
    'request' => [  
  
        'cookieValidationKey' => 'fill in a secret key here',  
  
    ],  
  
],  
  
];
```

补充: `[[yii\web\Request::cookieValidationKey|cookieValidationKey]]`
对你的应用安全很重要， 应只被你信任的人知晓， 请不要将它放入版本控制中。

URL 解析生成:URL Management

Note: This section is under development.

The concept of URL management in Yii is fairly simple. URL management is based on the premise that the application uses internal routes and parameters everywhere. The framework itself will then translate routes into URLs, and vice versa, according to the URL manager's configuration. This approach allows you to change site-wide URLs merely by editing a single configuration file, without ever touching the application code.

Internal routes

When implementing an application using Yii, you'll deal with internal routes, often referred to as routes and parameters. Each controller and controller action has a corresponding internal route such as `site/index` or `user/create`. In the first example, `site` is referred to as the controller ID while `index` is referred to as the action ID. In the second example, `user` is the controller ID and `create` is the action ID. If the controller belongs to a module, the internal route is prefixed with the module ID. For example `blog/post/index` for a blog module (with `post` being the controller ID and `index` being the action ID).

Creating URLs

The most important rule for creating URLs in your site is to always do so using the URL manager. The URL manager is a built-in application component named `urlManager`. This component is accessible from both web and console applications via `\Yii::$app->urlManager`. The component makes available the two following URL creation methods:

- `createUrl($params)`
- `createAbsoluteUrl($params, $schema = null)`

The `createUrl` method creates an URL relative to the application root, such as `/index.php/site/index/`. The `createAbsoluteUrl` method creates an URL prefixed with the

proper protocol and hostname: `http://www.example.com/index.php/site/index`. The former is suitable for internal application URLs, while the latter is used when you need to create URLs for external resources, such as connecting to third party services, sending email, generating RSS feeds etc.

Some examples:

```
echo \Yii::$app->urlManager->createUrl(['site/page', 'id' => 'about']);  
// /index.php/site/page/id/about/  
  
echo \Yii::$app->urlManager->createUrl(['date-time/fast-forward', 'id' => 105])  
// /index.php?r=date-time/fast-forward&id=105  
  
echo \Yii::$app->urlManager->createAbsoluteUrl('blog/post/index');  
// http://www.example.com/index.php/blog/post/index/
```

The exact format of the URL depends on how the URL manager is configured. The above examples may also output:

- `/site/page/id/about/`
- `/index.php?r=site/page&id=about`
- `/index.php?r=date-time/fast-forward&id=105`
- `/index.php/date-time/fast-forward?id=105`
- `http://www.example.com/blog/post/index/`
- `http://www.example.com/index.php?r=blog/post/index`

In order to simplify URL creation there is `[[yii\helpers\Url]]` helper. Assuming we're at `/index.php?r=management/default/users&id=10` the following is how `Url` helper works:

```
use yii\helpers\Url;  
  
// currently active route  
// /index.php?r=management/default/users  
echo Url::to("");  
  
// same controller, different action  
// /index.php?r=management/default/page&id=contact  
echo Url::toRoute(['page', 'id' => 'contact']);  
  
// same module, different controller and action
```

```

// /index.php?r=management/post/index
echo Url::toRoute('post/index');

// absolute route no matter what controller is making this call
// /index.php?r=site/index
echo Url::toRoute('/site/index');

// url for the case sensitive action `actionHiTech` of the current controller
// /index.php?r=management/default/hi-tech
echo Url::toRoute('hi-tech');

// url for action the case sensitive controller, `DateTimeController::actionFastForward`
// /index.php?r=date-time/fast-forward&id=105
echo Url::toRoute(['/date-time/fast-forward', 'id' => 105]);

// get URL from alias
// http://google.com/
Yii::setAlias('@google', 'http://google.com/');
echo Url::to('@google');

// get home URL
// /index.php?r=site/index
echo Url::home();

Url::remember(); // save URL to be used later
Url::previous(); // get previously saved URL

```

Tip: In order to generate URL with a hashtag, for example `/index.php?r=site/page&id=100#title`, you need to specify the parameter named `#` using `Url::to(['post/read', 'id' => 100, '#' => 'title'])`.

There's also `Url::canonical()` method that allows you to generate `canonical URL` for the currently executing action. The method ignores all action parameters except ones passed via action arguments:

```

namespace app\controllers;

use yii\web\Controller;
use yii\helpers\Url;

class CanonicalController extends Controller
{
    public function actionTest($page)
    {
        echo Url::canonical();
    }
}

```

When accessed as `/index.php?r=canonical/test&page=hello&number=42` canonical URL will be `/index.php?r=canonical/test&page=hello`.

Customizing URLs

By default, Yii uses a query string format for URLs, such as `/index.php?r=news/view&id=100`. In order to make URLs human-friendly i.e., more readable, you need to configure the `urlManager` component in the application's configuration file. Enabling "pretty" URLs will convert the query string format to a directory-based format: `/index.php/news/view?id=100`. Disabling the `showScriptName` parameter means that `index.php` will not be part of the URLs. Here's the relevant part of the application's configuration file:

```

<?php
return [
    // ...

    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
        ],
    ],
],

```

```
];
```

Note that this configuration will only work if the web server has been properly configured for Yii, see [installation](#).

Named parameters

A rule can be associated with a few **GET** parameters. These **GET** parameters appear in the rule's pattern as special tokens in the following format:

```
<ParameterName:ParameterPattern>
```

ParameterName is a name of a **GET** parameter, and the optional **ParameterPattern** is the regular expression that should be used to match the value of the **GET** parameter. In case **ParameterPattern** is omitted, it means the parameter should match any characters except **/**. When creating a URL, these parameter tokens will be replaced with the corresponding parameter values; when parsing a URL, the corresponding GET parameters will be populated with the parsed results.

Let's use some examples to explain how URL rules work. We assume that our rule set consists of three rules:

```
[  
    'posts'=>'post/list',  
    'post/<id:\d+>'=>'post/read',  
    'post/<year:\d{4}>/<title>'=>'post/read',  
]
```

- Calling `Url::toRoute('post/list')` generates `/index.php/posts`. The first rule is applied.
- Calling `Url::toRoute(['post/read', 'id' => 100])` generates `/index.php/post/100`. The second rule is applied.
- Calling `Url::toRoute(['post/read', 'year' => 2008, 'title' => 'a sample post'])` generates `/index.php/post/2008/a%20sample%20post`. The third rule is applied.
- Calling `Url::toRoute('post/read')` generates `/index.php/post/read`. None of the rules is applied, convention is used instead.

In summary, when using `createUrl` to generate a URL, the route and the **GET** parameters passed to the method are used to decide which URL rule to be applied. If every parameter associated with a rule can be found in the **GET** parameters passed to `createUrl`, and if the route of the rule also matches the route parameter, the rule will be used to generate the URL.

If the **GET** parameters passed to `Url::toRoute` are more than those required by a rule, the additional parameters will appear in the query string. For example, if we call `Url::toRoute(['post/read', 'id' => 100, 'year' => 2008])`, we will obtain `/index.php/post/100?year=2008`.

As we mentioned earlier, the other purpose of URL rules is to parse the requesting URLs. Naturally, this is an inverse process of URL creation. For example, when a user requests for `/index.php/post/100`, the second rule in the above example will apply, which resolves in the route `post/read` and the `GET` parameter `['id' => 100]` (accessible via `Yii::$app->request->get('id')`).

Parameterizing Routes

We may reference named parameters in the route part of a rule. This allows a rule to be applied to multiple routes based on matching criteria. It may also help reduce the number of rules needed for an application, and thus improve the overall performance.

We use the following example rules to illustrate how to parameterize routes with named parameters:

```
[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>' =>
    '<controller>/<action>',

    '<controller:(post|comment)>/<id:\d+>' => '<controller>/read',

    '<controller:(post|comment)>s' => '<controller>/list',
]
```

In the above example, we use two named parameters in the route part of the rules: `controller` and `action`. The former matches a controller ID to be either post or comment, while the latter matches an action ID to be create, update or delete. You may name the parameters differently as long as they do not conflict with GET parameters that may appear in URLs.

Using the above rules, the URL `/index.php/post/123/create` will be parsed as the route `post/create` with `GET` parameter `id=123`. Given the route `comment/list` and `GET` parameter `page=2`, we can create a URL `/index.php/comments?page=2`.

Parameterizing hostnames

It is also possible to include hostnames in the rules for parsing and creating URLs. One may extract part of the hostname to be a `GET` parameter. This is especially useful for handling subdomains. For example, the URL `http://admin.example.com/en/profile` may be parsed into `GET` parameters `user=admin` and `lang=en`. On the other hand, rules with hostname may also be used to create URLs with parameterized hostnames.

In order to use parameterized hostnames, simply declare URL rules with host info, e.g.:

```
[
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
]
```

In the above example the first segment of the hostname is treated as the user parameter while the first segment of the path info is treated as the lang parameter. The rule corresponds to the `user/profile` route.

Note that `[[yii\web\UrlManager::showScriptName]]` will not take effect when a URL is being created using a rule with a parameterized hostname.

Also note that any rule with a parameterized hostname should NOT contain the subfolder if the application is under a subfolder of the Web root. For example, if the application is under `http://www.example.com/sandbox/blog`, then we should still use the same URL rule as described above without the subfolder `sandbox/blog`.

Faking URL Suffix

```
<?php
return [
    // ...

    'components' => [
        'urlManager' => [
            'suffix' => '.html',
        ],
    ],
];
```

Handling REST requests

TBD:

- RESTful routing: `[[yii\filters\VerbFilter]]`, `[[yii\web\UrlManager::$rules]]`
- Json API:
 - response: `[[yii\web\Response::format]]`
 - request: `[[yii\web\Request::$parsers]]`, `[[yii\web\JsonParser]]`

URL parsing

Complimentary to creating URLs Yii also handles transforming custom URLs back into internal routes and parameters.

Strict URL parsing

By default if there's no custom rule for a URL and the URL matches the default format such as `/site/page`, Yii tries to run the corresponding controller's action. This behavior can be disabled so if there's no custom rule match, a 404 not found error will be produced immediately.

```
<?php
return [
    // ...

    'components' => [

        'urlManager' => [

            'enableStrictParsing' => true,

        ],

    ],

];
```

Creating your own rule classes

`[[yii\web\UrlRule]]` class is used for both parsing URL into parameters and creating URL based on parameters. Despite the fact that default implementation is flexible enough for the majority of projects, there are situations when using your own rule class is the best choice. For example, in a car dealer website, we may want to support the URL format like `/Manufacturer/Model`, where `Manufacturer` and `Model` must both match some data in a database table. The default rule class will not work because it mostly relies on statically declared regular expressions which have no database knowledge.

We can write a new URL rule class by extending from `[[yii\web\UrlRule]]` and use it in one or multiple URL rules. Using the above car dealer website as an example, we may declare the following URL rules in application config:

```
// ...

'components' => [

    'urlManager' => [

        'rules' => [

            '<action:(login|logout|about)>' => 'site/<action>',

        ],

    ],

];

// ...
```



```
['class' => 'app\components\CarUrlRule', 'connectionID' => 'db', /* ... */],
],
],
```

In the above, we use the custom URL rule class `CarUrlRule` to handle the URL format `/Manufacturer/Model`. The class can be written like the following:

```
namespace app\components;

use yii\web\UrlRule;

class CarUrlRule extends UrlRule
{
    public $connectionID = 'db';

    public function init()
    {
        if ($this->name === null) {
            $this->name = __CLASS__;
        }
    }

    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {
                return $params['manufacturer'];
            }
        }
    }
}
```

```

        return $params['manufacturer'];

    }

}

return false; // this rule does not apply
}

public function parseRequest($manager, $request)
{
    $pathInfo = $request->getPathInfo();

    if (preg_match('%^(\\w+)/?(\\w+)?$%', $pathInfo, $matches)) {
        // check $matches[1] and $matches[3] to see
        // if they match a manufacturer and a model in the database
        // If so, set $params['manufacturer'] and/or $params['model']
        // and return ['car/index', $params]
    }

    return false; // this rule does not apply
}
}

```

Besides the above usage, custom URL rule classes can also be implemented for many other purposes. For example, we can write a rule class to log the URL parsing and creation requests. This may be useful during development stage. We can also write a rule class to display a special 404 error page in case all other URL rules fail to resolve the current request. Note that in this case, the rule of this special class must be declared as the last rule.

错误处理

Yii 内置了一个[[yii\web\ErrorHandler|error handler]]错误处理器，它使错误处理更方便，Yii 错误处理器做以下工作来提升错误处理效果：

- 所有非致命 PHP 错误（如，警告，提示）会转换成可获取异常；
- 异常和致命的 PHP 错误会被显示，在调试模式会显示详细的函数调用栈和源代码行数。
- 支持使用专用的 [控制器操作](#) 来显示错误；
- 支持不同的错误响应格式；

`[[yii\web\ErrorHandler|error handler]]` 错误处理器默认启用， 可通过在应用的[入口脚本](#)中定义常量 `YII_ENABLE_ERROR_HANDLER` 来禁用。

使用错误处理器

`[[yii\web\ErrorHandler|error handler]]` 注册成一个名称为 `errorHandler` [应用组件](#)， 可以在应用配置中配置它类似如下：

```
return [  
    'components' => [  
        'errorHandler' => [  
            'maxSourceLines' => 20,  
        ],  
    ],  
];
```

使用如上代码，异常页面最多显示 20 条源代码。

如前所述，错误处理器将所有非致命 PHP 错误转换成可获取异常，也就是说可以使用如下代码处理 PHP 错误：

```
use Yii;  
  
use yii\base\Exception;  
  
try {  
    10/0;  
} catch (Exception $e) {  
    Yii::warning("Division by zero.");  
}  
  
// execution continues...
```

如果你想显示一个错误页面告诉用户请求是无效的或无法处理的，可简单地抛出一个 `[[yii\web\HttpException|HTTP exception]]` 异常，如 `[[yii\web\NotFoundHttpException]]`。错误处理器会正确地设置响应的 HTTP 状态码并使用合适的错误视图页面来显示错误信息。

```
use yii\web\NotFoundHttpException;

throw new NotFoundHttpException();
```

自定义错误显示

`[[yii\web\ErrorHandler|error handler]]` 错误处理器根据常量 `YII_DEBUG` 的值来调整错误显示，当 `YII_DEBUG` 为 `true` (表示在调试模式)，错误处理器会显示异常以及详细的函数调用栈和源代码行数来帮助调试，当 `YII_DEBUG` 为 `false`，只有错误信息会被显示以防止应用的敏感信息泄漏。

补充: 如果异常是继承 `[[yii\base\UserException]]`，不管 `YII_DEBUG` 为何值，函数调用栈信息都不会显示，这是因为这种错误会被认为是用户产生的错误，开发人员不需要去修正。

`[[yii\web\ErrorHandler|error handler]]` 错误处理器默认使用两个视图显示错误：

- `@yii/views/errorHandler/error.php`: 显示不包含函数调用栈信息的错误信息是使用，当 `YII_DEBUG` 为 `false` 时，所有错误都使用该视图。
- `@yii/views/errorHandler/exception.php`: 显示包含函数调用栈信息的错误信息时使用。

可以配置错误处理器的 `[[yii\web\ErrorHandler::errorView|errorView]]` 和 `[[yii\web\ErrorHandler::exceptionView|exceptionView]]` 属性 使用自定义的错误显示视图。

使用错误操作

使用指定的错误操作来自定义错误显示更方便，为此，首先配置 `errorHandler` 组件的 `[[yii\web\ErrorHandler::errorAction|errorAction]]` 属性，类似如下：

```
return [

    'components' => [

        'errorHandler' => [

            'errorAction' => 'site/error',

        ],

    ]

];
```

`[[yii\web\ErrorHandler::errorAction|errorAction]]` 属性使用路由到一个操作，上述配置表示不用显示函数调用栈信息的错误会通过执行 `site/error` 操作来显示。

可以创建 `site/error` 操作如下所示：

```
namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}
```

上述代码定义 `error` 操作使用 `[[yii\web\ErrorAction]]` 类，该类渲染名为 `error` 视图来显示错误。

除了使用 `[[yii\web\ErrorAction]]`，可定义 `error` 操作使用类似如下的操作方法：

```
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;

    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

现在应创建一个视图文件为 `views/site/error.php`，在该视图文件中，如果错误操作定义为 `[[yii\web\ErrorAction]]`，可以访问该操作中定义的如下变量：

- `name`: 错误名称

- **message**: 错误信息
- **exception**: 更多详细信息的异常对象，如 HTTP 状态码，错误码，错误调用栈等。

补充: 如果你使用 [基础应用模板](#) 或 [高级应用模板](#), 错误操作和错误视图已经定义好了。

自定义错误格式

错误处理器根据[响应](#)设置的格式来显示错误， 如果`[[yii\web\Response::format|response format]]` 响应格式为 **html**, 会使用错误或异常视图来显示错误信息，如上一小节所述。 对于其他的响应格式，错误处理器会将错误信息作为数组赋值给`[[yii\web\Response::data]]`属性，然后转换到对应的格式， 例如，如果响应格式为 **json**, 可以看到如下响应信息：

```
HTTP/1.1 404 Not Found

Date: Sun, 02 Mar 2014 05:31:43 GMT

Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y

Transfer-Encoding: chunked

Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

可在应用配置中响应 **response** 组件的 **beforeSend** 事件来自定义错误响应格式。

```
return [
    // ...

    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
```

```

        if ($response->data !== null) {

            $response->data = [

                'success' => $response->isSuccessful,

                'data' => $response->data,

            ];

            $response->statusCode = 200;

        }

    },

],

],

];

```

上述代码会重新格式化错误响应，类似如下：

HTTP/1.1 200 OK

Date: Sun, 02 Mar 2014 05:31:43 GMT

Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y

Transfer-Encoding: chunked

Content-Type: application/json; charset=UTF-8

```

{

    "success": false,

    "data": {

        "name": "Not Found Exception",

        "message": "The requested resource was not found.",

        "code": 0,

        "status": 404

    }

}

```

日志(Logging)

Note: This section is under development.

Yii provides flexible and extensible logger that is able to handle messages according to severity level or their type. You may filter messages by multiple criteria and forward them to files, email, debugger etc.

Logging basics

Basic logging is as simple as calling one method:

```
\Yii::info('Hello, I am a test log message');
```

You can log simple strings as well as more complex data structures such as arrays or objects. When logging data that is not a string the default yii log targets will serialize the value using `[[yii\helpers\Vardumper::export()]]`.

Message category

Additionally to the message itself message category could be specified in order to allow filtering such messages and handling these differently. Message category is passed as a second argument of logging methods and is `application` by default.

Severity levels

There are multiple severity levels and corresponding methods available:

- `[[Yii::trace]]` used mainly for development purpose to indicate workflow of some code. Note that it only works in development mode when `YII_DEBUG` is set to `true`.
- `[[Yii::error]]` used when there's unrecoverable error.
- `[[Yii::warning]]` used when an error occurred but execution can be continued.
- `[[Yii::info]]` used to keep record of important events such as administrator logins.

Log targets

When one of the logging methods is called, message is passed to `[[yii\log\Logger]]` component accessible as `Yii::getLogger()`. Logger accumulates messages in memory and then when there are enough messages or when the current request finishes, sends them to different log targets, such as file or email.

You may configure the targets in application configuration, like the following:

```
[
```



```

'bootstrap' => ['log'], // ensure logger gets loaded before application starts

'components' => [

    'log' => [

        'targets' => [

            'file' => [

                'class' => 'yii\log\FileTarget',

                'levels' => ['trace', 'info'],

                'categories' => ['yii\*'],

            ],

            'email' => [

                'class' => 'yii\log\EmailTarget',

                'levels' => ['error', 'warning'],

                'message' => [

                    'to' => ['admin@example.com', 'developer@example.com'],

                    'subject' => 'New example.com log message',

                ],

            ],

        ],

    ],

],

],

]

```

In the config above we are defining two log targets: `[[yii\log\FileTarget|file]]` and `[[yii\log\EmailTarget|email]]`. In both cases we are filtering messages handles by these targets by severity. In case of file target we're additionally filter by category. `yii\*` means all categories starting with `yii\`.

Each log target can have a name and can be referenced via the `[[yii\log\Logger::targets|targets]]` property as follows:

```
Yii::$app->log->targets['file']->enabled = false;
```

When the application ends or `[[yii\log\Logger::flushInterval|flushInterval]]` is reached,

Logger will call `[[yii\log\Logger::flush()|flush()]]` to send logged messages to different log targets, such as file, email, web.

Note: In the above configuration we added the log component to the list of [bootstrap](#) components that get initialized when the application is initialized to ensure logging is enabled from the start.

Profiling

Performance profiling is a special type of message logging that can be used to measure the time needed for the specified code blocks to execute and find out what the performance bottleneck is.

To use it we need to identify which code blocks need to be profiled. Then we mark the beginning and the end of each code block by inserting the following methods:

```
\Yii::beginProfile('myBenchmark');  
...code block being profiled...  
\Yii::endProfile('myBenchmark');
```

where `myBenchmark` uniquely identifies the code block.

Note, code blocks need to be nested properly such as

```
\Yii::beginProfile('block1');  
  
    // some code to be profiled  
  
    \Yii::beginProfile('block2');  
  
        // some other code to be profiled  
  
    \Yii::endProfile('block2');  
  
\Yii::endProfile('block1');
```

Profiling results [could be displayed in debugger](#).

关键概念：

组件（Component）

组件是 Yii 应用的主要基石。是 `[[yii\base\Component]]` 类或其子类的实例。三个用以区分它和其它类的主要功能有：

- 属性 (Property)
- 事件 (Event)
- 行为 (Behavior)

或单独使用，或彼此配合，这些功能的应用让 Yii 的类变得更加灵活和易用。以小部件 `[[yii\jui\DatePicker|日期选择器]]` 来举例，这是个方便你在 [视图](#) 中生成一个交互式日期选择器的 UI 组件：

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'zh-CN',
    'name' => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

这个小部件继承自 `[[yii\base\Component]]`，它的各项属性改写起来会很容易。

正是因为组件功能的强大，他们比常规的对象（Object）稍微重量级一点，因为他们要使用额外的内存和 CPU 时间来处理 [事件](#) 和 [行为](#)。如果你不需要这两项功能，可以继承 `[[yii\base\Object]]` 而不是 `[[yii\base\Component]]`。这样组件可以像普通 PHP 对象一样高效，同时还支持 [属性 \(Property\)](#) 功能。

当继承 `[[yii\base\Component]]` 或 `[[yii\base\Object]]` 时，推荐你使用如下的编码风格：

- 若你需要重写构造方法（Constructor），传入 `$config` 作为构造器方法**最后**一个参数，然后把它传递给父类的构造方法。
- 永远在你重写的构造方法**结尾处**调用一下父类的构造方法。
- 如果你重写了 `[[yii\base\Object::init()]]` 方法，请确保你在 `init` 方法的**开头处**调用了父类的 `init` 方法。

例子如下：

```
namespace yii\components\MyClass;

use yii\base\Object;

class MyClass extends Object
```

```

{

    public $prop1;

    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... 配置生效前的初始化过程

        parent::__construct($config);
    }

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

        // ... 配置生效后的初始化过程
    }
}

```

另外，为了让组件可以在创建实例时[能被正确配置](#)，请遵照以下操作流程：

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
```

// 方法二：

```

$component = \Yii::createObject([
    'class' => MyClass::className(),
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);

```

补充：尽管调用 `[[Yii::createObject()]]` 的方法看起来更加复杂，但这主要因为它更加灵活强大，它是基于[依赖注入容器](#)实现的。

[[yii\base\Object]] 类执行时的生命周期如下：

- 1.构造方法内的预初始化过程。你可以在这儿给各属性设置缺省值。
- 2.通过 `$config` 配置对象。配置的过程可能会覆盖掉先前在构造方法内设置的默认值。
- 3.在 [[yii\base\Object::init()|init()]] 方法内进行初始化后的收尾工作。你可以通过重写此方法，进行一些良品检验，属性的初始化之类的工作。
- 4.对象方法调用。

前三步都是在对象的构造方法内发生的。这意味着一旦你获得了一个对象实例，那么它就已经初始化就绪可供使用。

属性（Property）

在 PHP 中，类的成员变量也被称为**属性（properties）**。它们是类定义的一部分，用来表现一个实例的状态（也就是区分类的不同实例）。在具体实践中，常常会想用一個稍微特殊些的方法实现属性的读写。例如，如果有需求每次都要对 `label` 属性执行 `trim` 操作，就可以用以下代码实现：

```
$object->label = trim($label);
```

上述代码的缺点是只要修改 `label` 属性就必须再次调用 `trim()` 函数。若将来需要用其它方式处理 `label` 属性，比如首字母大写，就不得不修改所有给 `label` 属性赋值的代码。这种代码的重复会导致 `bug`，这种实践显然需要尽可能避免。

为解决该问题，Yii 引入了一个名为 [[yii\base\Object]] 的基类，它支持基于类内的 **getter** 和 **setter**（读取器和设定器）方法来定义属性。如果某类需要支持这个特性，只需要继承 [[yii\base\Object]] 或其子类即可。

补充：几乎每个 Yii 框架的核心类都继承自 [[yii\base\Object]] 或其子类。

这意味着只要在核心类中见到 `getter` 或 `setter` 方法，就可以像调用属性一样调用它。

`getter` 方法是名称以 `get` 开头的方法，而 `setter` 方法名以 `set` 开头。方法名中 `get` 或 `set` 后面的部分就定义了该属性的名字。如下面代码所示，`getter` 方法 `getLabel()` 和 `setter` 方法 `setLabel()` 操作的是 `label` 属性，：

```
namespace app\components;
```

```
use yii\base\Object;
```

```
class Foo extend Object
```

```
{
```

```
    private $_label;
```

```

public function getLabel()

{

    return $this->_label;

}


public function setLabel($value)

{

    $this->_label = trim($value);

}

}

```

（详细解释：getter 和 setter 方法创建了一个名为 **label** 的属性，在这个例子中，它指向一个私有的内部属性 **_label**。）

getter/setter 定义的属性用法与类成员变量一样。两者主要的区别是：当这种属性被读取时，对应的 getter 方法将被调用；而当属性被赋值时，对应的 setter 方法就调用。如：

```

// 等效于 $label = $object->getLabel();

$label = $object->label;


// 等效于 $object->setLabel('abc');

$object->label = 'abc';

```

只定义了 getter 没有 setter 的属性是**只读属性**。尝试赋值给这样的属性将导致 `[[yii\base\InvalidCallException|InvalidCallException]]`（无效调用）异常。类似的，只有 setter 方法而没有 getter 方法定义的属性是**只写属性**，尝试读取这种属性也会触发异常。使用只写属性的情况几乎没有。

通过 getter 和 setter 定义的属性也有一些特殊规则和限制：

- 这类属性的名字是不区分大小写的。如，`$object->label` 和 `$object->Label` 是同一个属性。因为 PHP 方法名是不区分大小写的。
- 如果此类属性名和类成员变量相同，以后者为准。例如，假设以上 `Foo` 类有个 `label` 成员变量，然后给 `$object->label = 'abc'` 赋值，将赋给成员变量而不是 setter `setLabel()` 方法。
- 这类属性不支持可见性（访问限制）。定义属性的 getter 和 setter 方法是 `public`、`protected` 还是 `private` 对属性的可见性没有任何影响。
- 这类属性的 getter 和 setter 方法只能定义为**非静态**的，若定义为静态方法（`static`）则不会以相同方式处理。

回到开头提到的问题，与其处处要调用 `trim()` 函数，现在我们只需在 setter `setLabel()` 方法内调用一次。如果 `label` 首字母变成大写的新要求来了，我们只需要修改 `setLabel()` 方法，而无须接触任何其它代码。

事件(Event)

事件可以将自定义代码“注入”到现有代码中的特定执行点。附加自定义代码到某个事件，当这个事件被触发时，这些代码就会自动执行。例如，邮件程序对象成功发出消息时可触发 `messageSent` 事件。如想追踪成功发送的消息，可以附加相应追踪代码到 `messageSent` 事件。

Yii 引入了名为 `[[yii\base\Component]]` 的基类以支持事件。如果一个类需要触发事件就应该继承 `[[yii\base\Component]]` 或其子类。

事件处理器 (Event Handlers)

事件处理器是一个 PHP 回调函数，当它所附加到的事件被触发时它就会执行。可以使用以下回调函数之一：

- 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) { ... }`。

事件处理器的格式是：

```
function ($event) {  
  
    // $event 是 yii\base\Event 或其子类的对象  
  
}
```

通过 `$event` 参数，事件处理器就获得了以下有关事件的信息：

- `[[yii\base\Event::name|event name]]`：事件名
- `[[yii\base\Event::sender|event sender]]`：调用 `trigger()` 方法的对象
- `[[yii\base\Event::data|custom data]]`：附加事件处理器时传入的数据，默认为空，后文详述

附加事件处理器

调用 `[[yii\base\Component::on()]]` 方法来附加处理器到事件上。如：

```
$foo = new Foo;
```

```
// 处理器是全局函数

$foo->on(Foo::EVENT_HELLO, 'function_name');


// 处理器是对象方法

$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);


// 处理器是静态类方法

$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);


// 处理器是匿名函数

$foo->on(Foo::EVENT_HELLO, function ($event) {

    //事件处理逻辑

});
```

附加事件处理器时可以提供额外数据作为 `[[yii\base\Component::on()]]` 方法的第三个参数。数据在事件被触发和处理器被调用时能被处理器使用。如：

```
// 当事件被触发时以下代码显示 "abc"

// 因为 $event->data 包括被传递到 "on" 方法的数据

$foo->on(Foo::EVENT_HELLO, function ($event) {

    echo $event->data;

}, 'abc');
```

事件处理器顺序

可以附加一个或多个处理器到一个事件。当事件被触发，已附加的处理器将按附加次序依次调用。如果某个处理器需要停止其后的处理器调用，可以设置 `$event` 参数的 `[[yii\base\Event::handled]]` 属性为真，如下：

```
$foo->on(Foo::EVENT_HELLO, function ($event) {

    $event->handled = true;

});
```

默认新附加的事件处理器排在已存在处理器队列的最后。因此，这个处理器将在事件被触发时最后一个调用。在处理器队列最前面插入新处理器将使该处理器最先调用，可以传递第四个参数 `$append` 为假并调用 `[[yii\base\Component::on()]]` 方法实现：


```
$foo->on(Foo::EVENT_HELLO, function ($event) {  
  
    // 这个处理器将被插入到处理器队列的第一位...  
  
}, $data, false);
```

触发事件

事件通过调用 `[[yii\base\Component::trigger()]]` 方法触发，此方法须传递**事件名**，还可以传递一个事件对象，用来传递参数到事件处理器。如：

```
namespace app\components;  
  
use yii\base\Component;  
use yii\base\Event;  
  
class Foo extends Component  
{  
  
    const EVENT_HELLO = 'hello';  
  
    public function bar()  
    {  
  
        $this->trigger(self::EVENT_HELLO);  
  
    }  
}
```

以上代码当调用 `bar()`，它将触发名为 `hello` 的事件。

提示：推荐使用类常量来表示事件名。上例中，常量 `EVENT_HELLO` 用来表示 `hello`。这有两个好处。第一，它可以防止拼写错误并支持 IDE 的自动完成。第二，只要简单检查常量声明就能了解一个类支持哪些事件。

有时想要在触发事件时同时传递一些额外信息到事件处理器。例如，邮件程序要传递消息信息到 `messageSent` 事件的处理器以便处理器了解哪些消息被发送了。为此，可以提供一个事件对象作为 `[[yii\base\Component::trigger()]]` 方法的第二个参数。这个事件对象必须是 `[[yii\base\Event]]` 类或其子类的实例。如：

```
namespace app\components;  
  
use yii\base\Component;
```

```

use yii\base\Event;

class MessageEvent extends Event
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ...发送 $message 的逻辑...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}

```

当 `[[yii\base\Component::trigger()]]` 方法被调用时，它将调用所有附加到命名事件（`trigger` 方法第一个参数）的事件处理器。

移除事件处理器

从事件移除处理器，调用 `[[yii\base\Component::off()]]` 方法。如：

```

// 处理器是全局函数

$foo->off(Foo::EVENT_HELLO, 'function_name');

// 处理器是对象方法

```

```
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);
```

// 处理器是静态类方法

```
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);
```

// 处理器是匿名函数

```
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

注意当匿名函数附加到事件后一般不要尝试移除匿名函数，除非你在某处存储了它。以上示例中，假设匿名函数存储为变量`$anonymousFunction`。

移除事件的全部处理器，简单调用 `[[yii\base\Component::off()]]` 即可，不需要第二个参数：

```
$foo->off(Foo::EVENT_HELLO);
```

类级别的事件处理器

以上部分，我们叙述了在**实例级别**如何附加处理器到事件。有时想要一个类的所有实例而不是一个指定的实例都响应一个被触发的事件，并不是一个个附加事件处理器到每个实例，而是通过调用静态方法 `[[yii\base\Event::on()]]` 在**类级别**附加处理器。

例如，[活动记录](#)对象要在每次往数据库新增一条新记录时触发一个 `[[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]]` 事件。要追踪每个[活动记录](#)对象的新增记录完成情况，应如下写代码：

```
use Yii;

use yii\base\Event;

use yii\db\ActiveRecord;

Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT, function ($event) {

    Yii::trace(get_class($event->sender) . ' is inserted');

});
```

每当 `[[yii\db\BaseActiveRecord|ActiveRecord]]` 或其子类的实例触发 `[[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]]` 事件时，这个事件处理器都会执行。在这个处理器中，可以通过 `$event->sender` 获取触发事件的对象。

当对象触发事件时，它首先调用实例级别的处理器，然后才会调用类级别处理器。

可调用静态方法 `[[yii\base\Event::trigger()]]` 来触发一个**类级别**事件。类级别事件不与特定对象相关联。因此，它只会引起类级别事件处理器的调用。如：

```
use yii\base\Event;
```

```
Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {  
  
    echo $event->sender; // 显示 "app\models\Foo"  
  
});
```

```
Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

注意这种情况下 `$event->sender` 指向触发事件的类名而不是对象实例。

注意：因为类级别的处理器响应类和其子类的所有实例触发的事件，必须谨慎使用，尤其是底层的基类，如 `[[yii\base\Object]]`。

移除类级别的事件处理器只需调用`[[yii\base\Event::off()]]`，如：

```
// 移除 $handler  
  
Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);  
  
  
  
// 移除 Foo::EVENT_HELLO 事件的全部处理器  
  
Event::off(Foo::className(), Foo::EVENT_HELLO);
```

全局事件

所谓**全局事件**实际上是一个基于以上叙述的事件机制的戏法。它需要一个全局可访问的单例，如[应用实例](#)。

事件触发者不调用其自身的 `trigger()` 方法，而是调用单例的 `trigger()` 方法来触发全局事件。类似地，事件处理器被附加到单例的事件。如：

```
use Yii;  
  
use yii\base\Event;  
  
use app\components\Foo;  
  
  
Yii::$app->on('bar', function ($event) {  
  
    echo get_class($event->sender); // 显示 "app\components\Foo"  
  
});  
  
  
  
Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

全局事件的一个好处是当附加处理器到一个对象要触发的事件时，不需要产生该对象。相反，处理器附

加和事件触发都通过单例（如应用实例）完成。

然而，因为全局事件的命名空间由各方共享，应合理命名全局事件，如引入一些命名空间（例： "frontend.mail.sent", "backend.mail.sent"）。

行为(Behavior)

行为是 `[[yii\base\Behavior]]` 或其子类的实例。行为，也称为 **mixins**，可以无须改变类继承关系即可增强一个已有的 `[[yii\base\Component|组件]]` 类功能。当行为附加到组件后，它将“注入”它的方法和属性到组件，然后可以像访问组件内定义的方法和属性一样访问它们。此外，行为通过组件能响应被触发的**事件**，从而自定义或调整组件正常执行的代码。

定义行为

要定义行为，通过继承 `[[yii\base\Behavior]]` 或其子类来建立一个类。如：

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

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

    public function setProp2($value)
    {

```

```

        $this->_prop2 = $value;

    }

    public function foo()

    {

        // ...

    }

}

```

以上代码定义了行为类 `app\components\MyBehavior` 并为要附加行为的组件提供了两个属性 `prop1`、`prop2` 和一个方法 `foo()`。注意属性 `prop2` 是通过 getter `getProp2()` 和 setter `setProp2()` 定义的。能这样用是因为 `[[yii\base\Object]]` 是 `[[yii\base\Behavior]]` 的祖先类，此祖先类支持用 `getter` 和 `setter` 方法定义[属性](#)

提示：在行为内部可以通过 `[[yii\base\Behavior::owner]]` 属性访问行为已附加的组件。

处理事件

如果要想行为响应对应组件的事件触发，就应覆写 `[[yii\base\Behavior::events()]]` 方法，如：

```

namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{

    // 其它代码

    public function events()

    {

        return [

            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',

```

```

    ];

}

public function beforeValidate($event)

{

    // 处理器方法逻辑

}

}

```

[[yii\base\Behavior::events()|events()]] 方法返回事件列表和相应的处理器。上例声明了 [[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]] 事件和它的处理器 `beforeValidate()`。当指定一个事件处理器时，要使用以下格式之一：

- 指向行为类的方法名的字符串，如上例所示；
- 对象或类名和方法名的数组，如 `[$object, 'methodName']`；
- 匿名方法。

处理器的格式如下，其中 `$event` 指向事件参数。关于事件的更多细节请参考[事件](#)：

```

function ($event) {

}

```

附加行为

可以静态或动态地附加行为到[[yii\base\Component|组件]]。前者在实践中更常见。

要静态附加行为，覆盖行为要附加的组件类的 [[yii\base\Component::behaviors()|behaviors()]] 方法即可。[[yii\base\Component::behaviors()|behaviors()]] 方法应该返回行为[配置](#)列表。每个行为配置可以是行为类名也可以是配置数组。如：

```

namespace app\models;

use yii\db\ActiveRecord;

use app\components\MyBehavior;

class User extends ActiveRecord
{

    public function behaviors()

```

```

{

    return [

        // 匿名行为，只有行为类名

        MyBehavior::className(),

        // 命名行为，只有行为类名

        'myBehavior2' => MyBehavior::className(),

        // 匿名行为，配置数组

        [

            'class' => MyBehavior::className(),

            'prop1' => 'value1',

            'prop2' => 'value2',

        ],

        // 命名行为，配置数组

        'myBehavior4' => [

            'class' => MyBehavior::className(),

            'prop1' => 'value1',

            'prop2' => 'value2',

        ]

    ];

}
}

```

通过指定行为配置数组相应的键可以给行为关联一个名称。这种行为称为**命名行为**。上例中，有两个命名行为：**myBehavior2** 和 **myBehavior4**。如果行为没有指定名称就是**匿名行为**。

要动态附加行为，在对应组件里调用 `[[yii\base\Component::attachBehavior()]]` 方法即可，如：

```
use app\components\MyBehavior;
```


// 附加行为对象

```
$component->attachBehavior('myBehavior1', new MyBehavior);
```

// 附加行为类

```
$component->attachBehavior('myBehavior2', MyBehavior::className());
```

// 附加配置数组

```
$component->attachBehavior('myBehavior3', [  
    'class' => MyBehavior::className(),  
    'prop1' => 'value1',  
    'prop2' => 'value2',  
]);
```

可以通过 `[[yii\base\Component::attachBehaviors()]]` 方法一次附加多个行为:

```
$component->attachBehaviors([  
    'myBehavior1' => new MyBehavior, // 命名行为  
    MyBehavior::className(),        // 匿名行为  
]);
```

还可以通过[配置](#)去附加行为:

```
[  
    'as myBehavior2' => MyBehavior::className(),  
  
    'as myBehavior3' => [  
        'class' => MyBehavior::className(),  
        'prop1' => 'value1',  
        'prop2' => 'value2',  
    ],  
]
```

详情请参考[配置](#)章节。

使用行为

使用行为，必须像前文描述的一样先把它附加到 `[[yii\base\Component|component]]` 类或其子类。一旦行为附加到组件，就可以直接使用它。

行为附加到组件后，可以通过组件访问一个行为的公共成员变量或 `getter` 和 `setter` 方法定义的属性：

```
// "prop1" 是定义在行为类的属性  
  
echo $component->prop1;  
  
$component->prop1 = $value;
```

类似地也可以调用行为的公共方法：

```
// foo() 是定义在行为类的公共方法  
  
$component->foo();
```

如你所见，尽管 `$component` 未定义 `prop1` 和 `foo()`，它们用起来也像组件自己定义的一样。

如果两个行为都定义了一样的属性或方法，并且它们都附加到同一个组件，那么首先附加上的行为在属性或方法被访问时有优先权。

附加行为到组件时的命名行为，可以使用这个名称来访问行为对象，如下所示：

```
$behavior = $component->getBehavior('myBehavior');
```

也能获取附加到这个组件的所有行为：

```
$behaviors = $component->getBehaviors();
```

移除行为

要移除行为，可以调用 `[[yii\base\Component::detachBehavior()]]` 方法用行为相关联的名字实现：

```
$component->detachBehavior('myBehavior1');
```

也可以移除全部行为：

```
$component->detachBehaviors();
```

使用 **TimestampBehavior**

最后以 `[[yii\behaviors\TimestampBehavior]]` 的讲解来结尾，这个行为支持在 `[[yii\db\ActiveRecord|Active Record]]` 存储时自动更新它的时间戳属性。

首先，附加这个行为到计划使用该行为的 `[[yii\db\ActiveRecord|Active Record]]` 类：

```
namespace app\models\User;
```

```

use yii\db\ActiveRecord;

use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::className(),
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
            ],
        ];
    }
}

```

以上指定的行为数组：

- 当记录插入时，行为将当前时间戳赋值给 `created_at` 和 `updated_at` 属性；
- 当记录更新时，行为将当前时间戳赋值给 `updated_at` 属性。

保存 `User` 对象，将会发现它的 `created_at` 和 `updated_at` 属性自动填充了当前时间戳：

```

$user = new User;

$user->email = 'test@example.com';

$user->save();

echo $user->created_at; // 显示当前时间戳

```

`[[yii\behaviors\TimestampBehavior|TimestampBehavior]]` 行为还提供了一个有用的方法 `[[yii\behaviors\TimestampBehavior::touch()|touch()]]`，这个方法能将当前时间戳赋值给指定属性并保存到数据库：

```
$user->touch('login_time');
```

与 PHP traits 的比较

尽管行为在 "注入" 属性和方法到主类方面类似于 `traits`，它们在很多方面却不相同。如上所述，它们各有利弊。它们更像是互补的而不是相互替代。

行为的优势

行为类像普通类支持继承。另一方面，`traits` 可以视为 PHP 语言支持的复制粘贴功能，它不支持继承。

行为无须修改组件类就可动态附加到组件或移除。要使用 `traits`，必须修改使用它的类。

行为是可配置的而 `traits` 不能。

行为以响应事件来自定义组件的代码执行。

当不同行为附加到同一组件产生命名冲突时，这个冲突通过先附加行为的优先权自动解决。而由不同 `traits` 引发的命名冲突需要通过手工重命名冲突属性或方法来解决。

traits 的优势

`traits` 比起行为更高效，因为行为是对象，消耗时间和内存。

IDE 对 `traits` 更友好，因为它们是语言结构。

配置(Configurations)

在 Yii 中，创建新对象和初始化已存在对象时广泛使用配置。配置通常包含被创建对象的类名和一组将要赋值给对象属性的初始值。还可能包含一组将被附加到对象事件上的句柄。和一组将被附加到对象上的行为。

以下代码中的配置被用来创建并初始化一个数据库连接：

```
$config = [  
    'class' => 'yii\db\Connection',  
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',  
    'username' => 'root',  
    'password' => '',
```

```
'charset' => 'utf8',  
  
];  
  
$db = Yii::createObject($config);
```

`[[Yii::createObject()]]` 方法接受一个配置数组并根据数组中指定的类名创建对象。对象实例化后，剩余的参数被用来初始化对象的属性，事件处理和行为。

对于已存在的对象，可以使用 `[[Yii::configure()]]` 方法根据配置去初始化其属性，就像这样：

```
Yii::configure($object, $config);
```

请注意，如果配置一个已存在的对象，那么配置数组中不应该包含指定类名的 `class` 元素。

配置的格式

一个配置的格式可以描述为以下形式：

```
[  
  
    'class' => 'ClassName',  
  
    'propertyName' => 'propertyValue',  
  
    'on eventName' => $eventHandler,  
  
    'as behaviorName' => $behaviorConfig,  
  
]
```

其中

- `class` 元素指定了将要创建的对象完全限定类名。
- `propertyName` 元素指定了对象属性的初始值。键名是属性名，值是该属性对应的初始值。只有公共成员变量以及通过 `getter/setter` 定义的属性可以被配置。
- `on eventName` 元素指定了附加到对象事件上的句柄是什么。请注意，数组的键名由 `on` 前缀加事件名组成。请参考[事件](#)章节了解事件句柄格式。
- `as behaviorName` 元素指定了附加到对象的行为。请注意，数组的键名由 `as` 前缀加行为名组成。`$behaviorConfig` 值表示创建行为的配置信息，格式与我们之前描述的配置格式一样。

下面是一个配置了初始化属性值，事件句柄和行为的示例：

```
[  
  
    'class' => 'app\components\SearchEngine',  
  
    'apiKey' => 'xxxxxxxx',  
  
    'on search' => function ($event) {
```

```

        Yii::info("搜索的关键词: " . $event->keyword);

    },

    'as indexer' => [

        'class' => 'app\components\IndexerBehavior',

        // ... 初始化属性值 ...

    ],

]

```

使用配置

Yii 中的配置可以用在很多场景。本章开头我们展示了如何使用 `[[Yii::createObject()]]` 根据配置信息创建对象。本小节将介绍配置的两种主要用法 —— 配置应用与配置小部件。

应用的配置

[应用](#)的配置可能是最复杂的配置之一。因为 `[[yii\web\Application|application]]` 类拥有很多可配置的性质和事件。更重要的是它的 `[[yii\web\Application::components|components]]` 属性可以接收配置数组并通过应用注册为组件。以下是一个针对[基础应用模板](#)的应用配置概要：

```

$config = [

    'id' => 'basic',

    'basePath' => dirname(__DIR__),

    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),

    'components' => [

        'cache' => [

            'class' => 'yii\caching\FileCache',

        ],

        'mailer' => [

            'class' => 'yii\swiftmailer\Mailer',

        ],

        'log' => [

```

```

        'class' => 'yii\log\Dispatcher',

        'traceLevel' => YII_DEBUG ? 3 : 0,

        'targets' => [

            [

                'class' => 'yii\log\FileTarget',

            ],

        ],

    ],

    'db' => [

        'class' => 'yii\db\Connection',

        'dsn' => 'mysql:host=localhost;dbname=stay2',

        'username' => 'root',

        'password' => '',

        'charset' => 'utf8',

    ],

],

];

```

配置中没有 `class` 键的原因是这段配置应用在下方的入口脚本中，类名已经指定了。

```
(new yii\web\Application($config))->run();
```

更多关于应用 `components` 属性配置的信息可以查阅[应用](#)以及[服务定位器](#)章节。

小部件的配置

使用[小部件](#)时，常常需要配置以便自定义其属性。[[yii\base\Widget::widget()]] 和 [[yii\base\Widget::begin()]] 方法都可以用来创建小部件。它们可以接受配置数组：

```

use yii\widgets\Menu;

echo Menu::widget([

    'activateItems' => false,

```

```

        'items' => [

            ['label' => 'Home', 'url' => ['site/index']],

            ['label' => 'Products', 'url' => ['product/index']],

            ['label' => 'Login', 'url' => ['site/login'], 'visible' => Yii::$app->user->isGuest],

        ],

    ]);

```

上述代码创建了一个小部件 `Menu` 并将其 `activateItems` 属性初始化为 `false`。 `item` 属性也配置成了将要显示的菜单条目。

请注意，代码中已经给出了类名 `yii\widgets\Menu`，配置数组**不应该**再包含 `class`` 键。

配置文件

当配置的内容十分复杂，通用做法是将其存储在一或多个 `PHP` 文件中，这些文件被称为**配置文件**。一个配置文件返回的是 `PHP` 数组。例如，像这样把应用配置信息存储在名为 `web.php` 的文件中：

```

return [

    'id' => 'basic',

    'basePath' => dirname(__DIR__),

    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),

    'components' => require(__DIR__ . '/components.php'),

];

```

鉴于 `components` 配置也很复杂，上述代码把它们存储在单独的 `components.php` 文件中，并且包含在 `web.php` 里。 `components.php` 的内容如下：

```

return [

    'cache' => [

        'class' => 'yii\caching\FileCache',

    ],

    'mailer' => [

        'class' => 'yii\swiftmailer\Mailer',

    ],

    'log' => [

```



```

        'class' => 'yii\log\Dispatcher',

        'traceLevel' => YII_DEBUG ? 3 : 0,

        'targets' => [

            [

                'class' => 'yii\log\FileTarget',

            ],

        ],

    ],

    'db' => [

        'class' => 'yii\db\Connection',

        'dsn' => 'mysql:host=localhost;dbname=stay2',

        'username' => 'root',

        'password' => '',

        'charset' => 'utf8',

    ],

];

```

仅仅需要 “require”，就可以取得一个配置文件的配置内容，像这样：

```

$config = require('path/to/web.php');

(new yii\web\Application($config))->run();

```

默认配置

[[Yii::createObject()]] 方法基于[依赖注入容器](#)实现。使用 [[Yii::createObject()]] 创建对象时，可以附加一系列**默认配置**到指定类的任何实例。默认配置还可以在[入口脚本](#)中调用 `Yii::$container->set()` 来定义。

例如，如果你想自定义 [[yii\widgets\LinkPager]] 小部件，以便让分页器最多只显示 5 个翻页按钮（默认是 10 个），你可以用下述代码实现：

```

Yii::$container->set('yii\widgets\LinkPager', [

    'maxButtonCount' => 5,

]);

```

不使用默认配置的话，你就得在任何使用分页器的地方，都配置 `maxButtonCount` 的值。

环境常量

配置经常要随着应用运行的不同环境更改。例如在开发环境中，你可能使用名为 `mydb_dev` 的数据库，而生产环境则使用 `mydb_prod` 数据库。为了便于切换使用环境，Yii 提供了一个定义在入口脚本中的 `YII_ENV` 常量。如下：

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

你可以把 `YII_ENV` 定义成以下任何一种值：

- **prod**：生产环境。常量 `YII_ENV_PROD` 将被看作 `true`。如果你没修改过，这就是 `YII_ENV` 的默认值。
- **dev**：开发环境。常量 `YII_ENV_DEV` 将被看作 `true`。
- **test**：测试环境。常量 `YII_ENV_TEST` 将被看作 `true`。

有了这些环境常量，你可以根据当下应用运行环境的不同，进行差异化配置。例如，应用可以包含下述代码只在开发环境中开启[调试工具](#)。

```
$config = [...];

if (YII_ENV_DEV) {
    // 根据 `dev` 环境进行的配置调整

    $config['bootstrap'][] = 'debug';

    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

类自动加载（Autoloading）

Yii 依靠[类自动加载机制](#)来定位和包含所需的类文件。它提供一个高性能且完美支持 [PSR-4 标准](#)（[中文汉化](#)）的自动加载器。该自动加载器会在引入框架文件 `Yii.php` 时安装好。

注意：为了简化叙述，本文档中我们只会提及类的自动加载。不过，要记得文中的描述同样也适用于接口和 **Trait**（特质）的自动加载哦。

使用 Yii 自动加载器

要使用 Yii 的类自动加载器，你需要在创建和命名类的时候遵循两个简单的规则：

- 每个类都必须置于命名空间之下（比如 `foo\bar\MyClass`）。
- 每个类都必须保存为单独文件，且其完整路径能用以下算法取得：

```
// $className 是一个开头包含反斜杠的完整类名（译注：请自行谷歌：fully qualified class name）
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) . '.php');
```

举例来说，若某个类名为 `foo\bar\MyClass`，对应类的文件路径别名会是 `@foo/bar/MyClass.php`。为了让该别名能被正确解析为文件路径，`@foo` 或 `@foo/bar` 中的一个必须是根别名。

当我们使用基本应用模版时，可以把你的类放置在顶级命名空间 `app` 下，这样它们就可以被 Yii 自动加载，而无需定义一个新的别名。这是因为 `@app` 本身是一个预定义别名，且类似于 `app\components\MyClass` 这样的类名，基于我们刚才所提到的算法，可以正确解析出 `AppBasePath/components/MyClass.php` 路径。

在高级应用模版里，每一逻辑层级会使用他自己的根别名。比如，前端层会使用 `@frontend` 而后端层会使用 `@backend`。因此，你可以把前端的类放在 `frontend` 命名空间，而后端的类放在 `backend`。这样这些类就可以被 Yii 自动加载了。

类映射表（Class Map）

Yii 类自动加载器支持类映射表功能，该功能会建立一个从类的名字到类文件路径的映射。当自动加载器加载一个文件时，他首先检查映射表里有没有该类。如果有，对应的文件路径就直接加载了，省掉了进一步的检查。这让类的自动加载变得超级快。事实上所有的 Yii 核心类都是这样加载的。

你可以用 `Yii::$classMap` 方法向映射表中添加类，

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

别名可以被用于指定类文件的路径。你应该在引导启动的过程中设置类映射表，这样映射表就可以在你使用具体类之前就准备好。

用其他自动加载器

因为 Yii 完全支持 Composer 管理依赖包，所以推荐你也同时安装 Composer 的自动加载器，如果你用了一些自带自动加载器的第三方类库，你应该也安装下它们。

当你同时使用其他自动加载器和 Yii 自动加载器时，应该在其他自动加载器安装成功之后，再包含 `Yii.php` 文件。这将使 Yii 成为第一个响应任何类自动加载请求的自动加载器。举例来说，以下代码提取自基本应用模版的入口脚本。第一行安装了 Composer 的自动加载器，第二行才是 Yii 的自动加载器：

```
require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
```

你也可以只使用 **Composer** 的自动加载，而不用 **Yii** 的自动加载。不过这样做的话，类的加载效率会下降，且你必须遵循 **Composer** 所设定的规则，从而让你的类满足可以被自动加载的要求。

补充：若你不想要使用 **Yii** 的自动加载器，你必须创建一个你自己版本的 **Yii.php** 文件，并把它包含进你的入口脚本里。

自动加载扩展类

Yii 自动加载器支持自动加载扩展的类。唯一的要求是它需要在 **composer.json** 文件里正确地定义 **autoload** 部分。请参考 [Composer 文档（英文）](#)（[中文汉化](#)），来了解如何正确描述 **autoload** 的更多细节。

在你不使用 **Yii** 的自动加载器时，**Composer** 的自动加载器仍然可以帮你自动加载扩展内的类。

别名（Aliases）

别名用来表示文件路径和 **URL**，这样就避免了在代码中硬编码一些绝对路径和 **URL**。一个别名必须以 **@** 字符开头，以区别于传统的文件路径和 **URL**。**Yii** 预定义了大量可用的别名。例如，别名 **@yii** 指的是 **Yii** 框架本身的安装目录，而 **@web** 表示的是当前运行应用的根 **URL**。

定义别名

你可以调用 `[[Yii::setAlias()]]` 来给文件路径或 **URL** 定义别名：

```
// 文件路径的别名

Yii::setAlias('@foo', '/path/to/foo');

// URL 的别名

Yii::setAlias('@bar', 'http://www.example.com');
```

注意：别名所指向的文件路径或 **URL** 不一定是真实存在的文件或资源。

可以通过在一个别名后面加斜杠 **/** 和一至多个路径分段生成新别名（无需调用 `[[Yii::setAlias()]]`）。我们把通过 `[[Yii::setAlias()]]` 定义的别名称为**根别名**，而用他们衍生出去的别名成为**衍生别名**。例如，**@foo** 就是根别名，而 **@foo/bar/file.php** 是一个衍生别名。

你还可以用别名去定义新别名（根别名与衍生别名均可）：

```
Yii::setAlias('@foobar', '@foo/bar');
```

根别名通常在[引导](#)阶段定义。比如你可以在[入口脚本](#)里调用 `[[Yii::setAlias()]]`。为了方便起见，[应用](#)提供了一个名为 **aliases** 的可写属性，你可以在应用[配置](#)中设置它，就像这样：

```
return [
```

```
// ...

'aliases' => [

    '@foo' => '/path/to/foo',

    '@bar' => 'http://www.example.com',

],

];
```

解析别名

你可以调用 `[[Yii::getAlias()]]` 命令来解析根别名到对应的文件路径或 URL。同样的页面也可以用于解析衍生别名。例如：

```
echo Yii::getAlias('@foo');           // 输出: /path/to/foo

echo Yii::getAlias('@bar');           // 输出: http://www.example.com

echo Yii::getAlias('@foo/bar/file.php'); // 输出: /path/to/foo/bar/file.php
```

由衍生别名所解析出的文件路径和 URL 是通过替换掉衍生别名中的根别名部分得到的。

注意： `[[Yii::getAlias()]]` 并不检查结果路径/URL 所指向的资源是否真实存在。

根别名可能也会包含斜杠 `/`。 `[[Yii::getAlias()]]` 足够智能到判断一个别名中的哪部分是根别名，因此能正确解析文件路径/URL。例如：

```
Yii::setAlias('@foo', '/path/to/foo');

Yii::setAlias('@foo/bar', '/path2/bar');

echo Yii::getAlias('@foo/test/file.php'); // 输出: /path/to/foo/test/file.php

echo Yii::getAlias('@foo/bar/file.php'); // 输出: /path2/bar/file.php
```

若 `@foo/bar` 未被定义为根别名，最后一行语句会显示为 `/path/to/foo/bar/file.php`。

使用别名

别名在 Yii 的很多地方都会被正确识别，无需调用 `[[Yii::getAlias()]]` 来把它们转换为路径/URL。例如， `[[yii\caching\FileCache::cachePath]]` 能同时接受文件路径或是指向文件路径的别名，因为通过 `@` 前缀能区分它们。

```
use yii\caching\FileCache;
```

```
$cache = new FileCache([  
    'cachePath' => '@runtime/cache',  
]);
```

请关注 [API 文档](#) 了解特定属性或方法参数是否支持别名。

预定义的别名

Yii 预定义了一系列别名来简化常用路径和 URL 的使用：

- **@yii** - **BaseYii.php** 文件所在的目录（也被称为框架安装目录）
- **@app** - 当前运行的应用 `[[yii\base\Application::basePath|根路径（base path）]]`
- **@runtime** - 当前运行的应用的 `[[yii\base\Application::runtimePath|运行环境（runtime）路径]]`
- **@vendor** - `[[yii\base\Application::vendorPath|Composer 供应商目录]]`
- **@webroot** - 当前运行应用的 Web 入口目录
- **@web** - 当前运行应用的根 URL

@yii 别名是在[入口脚本](#)里包含 **Yii.php** 文件时定义的，其他的别名都是在[配置应用](#)的时候，于应用的构造方法内定义的。

扩展的别名

每一个通过 Composer 安装的 [扩展](#) 都自动添加了一个别名。该别名会以该扩展在 **composer.json** 文件中所声明的根命名空间为名，且他直接代指该包的根目录。例如，如果你安装有 **yiisoft/yii2-jui** 扩展，会自动得到 **@yii/jui** 别名，它定义于[引导启动阶段](#)：

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

服务定位器 Service Locator

服务定位器是一个了解如何提供各种应用所需的服务（或组件）的对象。在服务定位器中，每个组件都只有一个单独的实例，并通过 ID 唯一地标识。用这个 ID 就能从服务定位器中得到这个组件。

在 Yii 中，服务定位器是 `[[yii\di\ServiceLocator]]` 或其子类的一个实例。

最常用的服务定位器是 **application**（应用）对象，可以通过 **\Yii::\$app** 访问。它所提供的服务被称为 **application components**（应用组件），比如：**request**、**response**、**urlManager** 组件。可以通过服务定位器所提供的功能，非常容易地配置这些组件，或甚至是用你自己的实现替换掉他们。

除了 `application` 对象，每个模块对象本身也是一个服务定位器。

要使用服务定位器，第一步是要注册相关组件。组件可以通过 `[[yii\di\ServiceLocator::set()]]` 方法进行注册。以下的方法展示了注册组件的不同方法：

```
use yii\di\ServiceLocator;

use yii\caching\FileCache;

$locator = new ServiceLocator;

// 通过一个可用于创建该组件的类名，注册 "cache"（缓存）组件。
$locator->set('cache', 'yii\caching\ApcCache');

// 通过一个可用于创建该组件的配置数组，注册 "db"（数据库）组件。
$locator->set('db', [

    'class' => 'yii\db\Connection',

    'dsn' => 'mysql:host=localhost;dbname=demo',

    'username' => 'root',

    'password' => '',

]);

// 通过一个能返回该组件的匿名函数，注册 "search" 组件。
$locator->set('search', function () {

    return new app\components\SolrService;

});

// 用组件注册 "pageCache" 组件
$locator->set('pageCache', new FileCache);
```

一旦组件被注册成功，你可以任选以下两种方式之一，通过它的 ID 访问它：

```
$cache = $locator->get('cache');

// 或者

$cache = $locator->cache;
```

如上所示，`[[yii\di\ServiceLocator]]` 允许通过组件 ID 像访问一个属性值那样访问一个组件。当你第一次访问某组件时，`[[yii\di\ServiceLocator]]` 会通过该组件的注册信息创建一个该组件的实例，并返回它。之后，如果再次访问，则服务定位器会返回同一个实例。

你可以通过 `[[yii\di\ServiceLocator::has()]]` 检查某组件 ID 是否被注册。若你用一个无效的 ID 调用 `[[yii\di\ServiceLocator::get()]]`，则会抛出一个异常。

因为服务定位器，经常会在创建时附带[配置信息](#)，因此我们提供了一个可写的属性，名为 `[[yii\di\ServiceLocator::setComponents()|components]]`，这样就可以配置该属性，或一次性注册多个组件。下面的代码展示了如何用配置数组，配置一个应用并注册"db"，"cache" 和 "search" 三个组件：

```
// ...  
'components' => [  
    'db' => [  
        'class' => 'yii\db\Connection',  
        'dsn' => 'mysql:host=localhost;dbname=demo',  
        'username' => 'root',  
        'password' => '',  
    ],  
    'cache' => 'yii\caching\ApcCache',  
    'search' => function () {  
        return new app\components\SolrService;  
    },  
],  
];
```

依赖注入容器(DI Container)

依赖注入（Dependency Injection，DI）容器就是一个对象，它知道怎样初始化并配置对象及其依赖的所有对象。[Martin 的文章](#) 已经解释了 DI 容器为什么很有用。这里我们主要讲解 Yii 提供的 DI 容器的使用方法。

依赖注入

Yii 通过 `[[yii\di\Container]]` 类提供 DI 容器特性。它支持如下几种类型的依赖注入：

- 构造方法注入；
- Setter 和属性注入；
- PHP 回调注入。

构造方法注入

在参数类型提示的帮助下，DI 容器实现了构造方法注入。当容器被用于创建一个新对象时，类型提示会告诉它要依赖什么类或接口。容器会尝试获取它所依赖的类或接口的实例，然后通过构造器将其注入新的对象。例如：

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}
```

```
$foo = $container->get('Foo');
```

// 上面的代码等价于：

```
$bar = new Bar;
```

```
$foo = new Foo($bar);
```

Setter 和属性注入

Setter 和属性注入是通过[配置](#)提供支持的。当注册一个依赖或创建一个新对象时，你可以提供一个配置，该配置会提供给容器用于通过相应的 Setter 或属性注入依赖。例如：

```
use yii\base\Object;
```

```
class Foo extends Object
```

```
{
```

```
    public $bar;
```

```

private $_qux;

public function getQux()
{
    return $this->_qux;
}

public function setQux(Qux $qux)
{
    $this->_qux = $qux;
}
}

$container->get('Foo', [], [
    'bar' => $container->get('Bar'),
    'qux' => $container->get('Qux'),
]);

```

PHP 回调注入

这种情况下，容器将使用一个注册过的 **PHP** 回调创建一个类的新实例。回调负责解决依赖并将其恰当地注入新创建的对象。例如：

```

$container->set('Foo', function () {
    return new Foo(new Bar);
});

$foo = $container->get('Foo');

```

注册依赖关系

可以用 `[[yii\di\Container::set()]]` 注册依赖关系。注册会用到一个依赖关系名称和一个依赖关系的定义。依赖关系名称可以是一个类名，一个接口名或一个别名。依赖关系的定义可以是一个类名，一个配置数组，或者一个 PHP 回调。

```
$container = new \yii\di\Container;

// 注册一个同类名一样的依赖关系，这个可以省略。

$container->set('yii\db\Connection');

// 注册一个接口

// 当一个类依赖这个接口时，相应的类会被初始化作为依赖对象。

$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// 注册一个别名。

// 你可以使用 $container->get('foo') 创建一个 Connection 实例

$container->set('foo', 'yii\db\Connection');

// 通过配置注册一个类

// 通过 get() 初始化时，配置将会被使用。

$container->set('yii\db\Connection', [

    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',

    'username' => 'root',

    'password' => '',

    'charset' => 'utf8',

]);

// 通过类的配置注册一个别名

// 这种情况下，需要通过一个 “class” 元素指定这个类

$container->set('db', [
```

```

        'class' => 'yii\db\Connection',

        'dsn' => 'mysql:host=127.0.0.1;dbname=demo',

        'username' => 'root',

        'password' => '',

        'charset' => 'utf8',

    ]);

// 注册一个 PHP 回调

// 每次调用 $container->get('db') 时，回调函数都会被执行。

$container->set('db', function ($container, $params, $config) {

    return new \yii\db\Connection($config);

});

// 注册一个组件实例

// $container->get('pageCache') 每次被调用时都会返回同一个实例。

$container->set('pageCache', new FileCache);

```

Tip: 如果依赖关系名称和依赖关系的定义相同，则不需要通过 DI 容器注册该依赖关系。

通过 `set()` 注册的依赖关系，在每次使用时都会产生一个新实例。可以使用 `[[yii\di\Container::setSingleton()]]` 注册一个单例的依赖关系：

```

$container->setSingleton('yii\db\Connection', [

    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',

    'username' => 'root',

    'password' => '',

    'charset' => 'utf8',

]);

```

解决依赖关系

注册依赖关系后，就可以使用 DI 容器创建新对象了。容器会自动解决依赖关系，将依赖实例化并注入新

创建的对象。依赖关系的解决是递归的，如果一个依赖关系中还有其他依赖关系，则这些依赖关系都会被自动解决。

可以使用 `[[yii\di\Container::get()]]` 创建新的对象。该方法接收一个依赖关系名称，它可以是一个类名，一个接口名或一个别名。依赖关系名或许是通过 `set()` 或 `setSingleton()` 注册的。你可以随意地提供一个类的构造器参数列表和一个 `configuration` 用于配置新创建的对象。例如：

```
// "db" 是前面定义过的一个别名

$db = $container->get('db');

// 等价于: $engine = new \app\components\SearchEngine($apiKey, ['type' => 1]);

$engine = $container->get('app\components\SearchEngine', [$apiKey], ['type' => 1]);
```

代码背后，DI 容器做了比创建对象多的多的工作。容器首先将检查类的构造方法，找出依赖的类或接口名，然后自动递归解决这些依赖关系。

如下代码展示了一个更复杂的示例。`UserLister` 类依赖一个实现了 `UserFinderInterface` 接口的对象；`UserFinder` 类实现了这个接口，并依赖于一个 `Connection` 对象。所有这些依赖关系都是通过类构造器参数的类型提示定义的。通过属性依赖关系的注册，DI 容器可以自动解决这些依赖关系并能通过一个简单的 `get('userLister')` 调用创建一个新的 `UserLister` 实例。

```
namespace app\models;

use yii\base\Object;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends Object implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
```

```

{

    $this->db = $db;

    parent::__construct($config);

}

public function findUser()

{

}

}

```

class **UserLister** extends **Object**

```

{

    public $finder;

    public function __construct(UserFinderInterface $finder, $config = [])

    {

        $this->finder = $finder;

        parent::__construct($config);

    }

}

```

```

$container = new Container;
$container->set('yii\db\Connection', [

    'dsn' => '...',

]);
$container->set('app\models\UserFinderInterface', [

    'class' => 'app\models\UserFinder',

]);

```

```
$container->set('userLister', 'app\models\UserLister');
```

```
$lister = $container->get('userLister');
```

// 等价于:

```
$db = new \yii\db\Connection(['dsn' => '...']);
```

```
$finder = new UserFinder($db);
```

```
$lister = new UserLister($finder);
```

实践中的运用

当在应用程序的入口脚本中引入 `Yii.php` 文件时，Yii 就创建了一个 DI 容器。这个 DI 容器可以通过 `[[Yii::$container]]` 访问。当调用 `[[Yii::createObject()]]` 时，此方法实际上会调用这个容器的 `[[yii\di\Container::get()|get()]]` 方法创建新对象。如上所述，DI 容器会自动解决依赖关系（如果有）并将其注入新创建的对象中。因为 Yii 在其多数核心代码中都使用了 `[[Yii::createObject()]]` 创建新对象，所以您可以通过 `[[Yii::$container]]` 全局性地自定义这些对象。

例如，你可以全局性自定义 `[[yii\widgets\LinkPager]]` 中分页按钮的默认数量：

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

这样如果你通过如下代码在一个视图里使用这个挂件，它的 `maxButtonCount` 属性就会被初始化为 5 而不是类中定义的默认值 10。

```
echo \yii\widgets\LinkPager::widget();
```

然而你依然可以覆盖通过 DI 容器设置的值：

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

另一个例子是借用 DI 容器中自动构造方法注入带来的好处。假设你的控制器类依赖一些其他对象，例如一个旅馆预订服务。你可以通过一个构造器参数声明依赖关系，然后让 DI 容器帮你自动解决这个依赖关系。

```
namespace app\controllers;
```

```
use yii\web\Controller;
```

```
use app\components\BookingInterface;
```

```
class HotelController extends Controller
```

```
{
```

```
protected $bookingService;

public function __construct($id, $module, BookingInterface $bookingService, $config = [])
{
    $this->bookingService = $bookingService;

    parent::__construct($id, $module, $config);
}
}
```

如果你从浏览器中访问这个控制器，你将看到一个报错信息，提醒你 `BookingInterface` 无法被实例化。这是因为你需要告诉 DI 容器怎样处理这个依赖关系。

```
\Yii::$container->set('app\components\BookingInterface', 'app\components\BookingService');
```

现在如果你再次访问这个控制器，一个 `app\components\BookingService` 的实例就会被创建并被作为第三个参数注入到控制器的构造器中。

什么时候注册依赖关系

由于依赖关系在创建新对象时需要解决，因此它们的注册应该尽早完成。以下是推荐的实践：

- 如果你是一个应用程序的开发者，你可以在应用程序的[入口脚本](#)或者被入口脚本引入的脚本中注册依赖关系。
- 如果你是一个可再分发[扩展](#)的开发者，你可以将依赖关系注册到扩展的引导类中。

总结

依赖注入和[服务定位器](#)都是流行的设计模式，它们使你可以用充分解耦且更利于测试的风格构建软件。强烈推荐你阅读 [Martin 的文章](#)，对依赖注入和服务定位器有个更深入的理解。

Yii 在依赖注入（DI）容器之上实现了它的[服务定位器](#)。当一个服务定位器尝试创建一个新的对象实例时，它会把调用转发到 DI 容器。后者将会像前文所述那样自动解决依赖关系。

配合数据库工作： 数据访问对象 **DAO**：

参见官网：<http://www.yiichina.com/guide/2/db-dao>

查询生成器(Query Builder and Query)

Note: This section is under development.

Yii provides a basic database access layer as described in the [Database basics](#) section. The database access layer provides a low-level way to interact with the database. While useful in some situations, it can be tedious and error-prone to write raw SQLs. An alternative approach is to use the Query Builder. The Query Builder provides an object-oriented vehicle for generating queries to be executed.

A typical usage of the query builder looks like the following:

```
$rows = (new \yii\db\Query())
```

```
->select('id, name')
```

```
->from('user')
```

```
->limit(10)
```

```
->all();
```

// which is equivalent to the following code:

```
$query = (new \yii\db\Query())
```

```
->select('id, name')
```

```
->from('user')
```

```
->limit(10);
```

// Create a command. You can get the actual SQL using \$command->sql

```
$command = $query->createCommand();
```

// Execute the command:

```
$rows = $command->queryAll();
```

Query Methods

As you can see, `[[yii\db\Query]]` is the main player that you need to deal with. Behind the

scene, `Query` is actually only responsible for representing various query information. The actual query building logic is done by `[[yii\db\QueryBuilder]]` when you call the `createCommand()` method, and the query execution is done by `[[yii\db\Command]]`.

For convenience, `[[yii\db\Query]]` provides a set of commonly used query methods that will build the query, execute it, and return the result. For example,

- `[[yii\db\Query::all()|all()]]`: builds the query, executes it and returns all results as an array.
- `[[yii\db\Query::one()|one()]]`: returns the first row of the result.
- `[[yii\db\Query::column()|column()]]`: returns the first column of the result.
- `[[yii\db\Query::scalar()|scalar()]]`: returns the first column in the first row of the result.
- `[[yii\db\Query::exists()|exists()]]`: returns a value indicating whether the query results in anything.
- `[[yii\db\Query::count()|count()]]`: returns the result of a `COUNT` query. Other similar methods include `sum($q)`, `average($q)`, `max($q)`, `min($q)`, which support the so-called aggregational data query. `$q` parameter is mandatory for these methods and can be either the column name or expression.

Building Query

In the following, we will explain how to build various clauses in a SQL statement. For simplicity, we use `$query` to represent a `[[yii\db\Query]]` object.

SELECT

In order to form a basic `SELECT` query, you need to specify what columns to select and from what table:

```
$query->select('id, name')  
  
->from('user');
```

Select options can be specified as a comma-separated string, as in the above, or as an array. The array syntax is especially useful when forming the selection dynamically:

```
$query->select(['id', 'name'])  
  
->from('user');
```

Info: You should always use the array format if your `SELECT` clause contains SQL expressions. This is because a SQL expression like `CONCAT(first_name, last_name) AS full_name` may contain commas. If you list it together with other columns in a string, the expression may be split into several parts by commas, which is not what you want to see.

When specifying columns, you may include the table prefixes or column aliases, e.g., `user.id`, `user.id AS user_id`. If you are using array to specify the columns, you may also use the array keys to specify the column aliases, e.g., `['user_id' => 'user.id', 'user_name' => 'user.name']`.

To select distinct rows, you may call `distinct()`, like the following:

```
$query->select('user_id')->distinct()->from('post');
```

FROM

To specify which table(s) to select data from, call `from()`:

```
$query->select('*')->from('user');
```

You may specify multiple tables using a comma-separated string or an array. Table names can contain schema prefixes (e.g. `'public.user'`) and/or table aliases (e.g. `'user u'`). The method will automatically quote the table names unless it contains some parenthesis (which means the table is given as a sub-query or DB expression). For example,

```
$query->select('u.*', 'p.*')->from(['user u', 'post p']);
```

When the tables are specified as an array, you may also use the array keys as the table aliases (if a table does not need alias, do not use a string key). For example,

```
$query->select('u.*', 'p.*')->from(['u' => 'user', 'p' => 'post']);
```

You may specify a sub-query using a `Query` object. In this case, the corresponding array key will be used as the alias for the sub-query.

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');  
$query->select('*')->from(['u' => $subQuery]);
```

WHERE

Usually data is selected based upon certain criteria. Query Builder has some useful methods to specify these, the most powerful of which being `where`. It can be used in multiple ways.

The simplest way to apply a condition is to use a string:

```
$query->where('status=:status', [':status' => $status]);
```

When using strings, make sure you're binding the query parameters, not creating a query by string concatenation. The above approach is safe to use, the following is not:

```
$query->where("status=$status"); // Dangerous!
```

Instead of binding the status value immediately, you can do so using `params` or `addParams`:

```
$query->where('status=:status');  
$query->addParams([':status' => $status]);
```

Multiple conditions can simultaneously be set in `where` using the hash format:

```
$query->where([
    'status' => 10,
    'type' => 2,
    'id' => [4, 8, 15, 16, 23, 42],
]);
```

That code will generate the following SQL:

```
WHERE (`status` = 10) AND (`type` = 2) AND (`id` IN (4, 8, 15, 16, 23, 42))
```

NULL is a special value in databases, and is handled smartly by the Query Builder. This code:

```
$query->where(['status' => null]);
```

results in this WHERE clause:

```
WHERE (`status` IS NULL)
```

You can also create sub-queries with `Query` objects like the following,

```
$userQuery = (new Query)->select('id')->from('user');
$query->where(['id' => $userQuery]);
```

which will generate the following SQL:

```
WHERE `id` IN (SELECT `id` FROM `user`)
```

Another way to use the method is the operand format which is `[operator, operand1, operand2, ...]`.

Operator can be one of the following:

- **and**: the operands should be concatenated together using **AND**. For example, `['and', 'id=1', 'id=2']` will generate `id=1 AND id=2`. If an operand is an array, it will be converted into a string using the rules described here. For example, `['and', 'type=1', ['or', 'id=1', 'id=2']]` will generate `type=1 AND (id=1 OR id=2)`. The method will NOT do any quoting or escaping.
- **or**: similar to the **and** operator except that the operands are concatenated using **OR**.
- **between**: operand 1 should be the column name, and operand 2 and 3 should be the starting and ending values of the range that the column is in. For example, `['between', 'id', 1, 10]` will generate `id BETWEEN 1 AND 10`.
- **not between**: similar to **between** except the **BETWEEN** is replaced with **NOT BETWEEN** in the generated condition.
- **in**: operand 1 should be a column or DB expression. Operand 2 can be either an array or a `Query` object. It will generate an **IN** condition. If Operand 2 is an array, it will represent the range of the values that the column or DB expression should be; If Operand 2 is a `Query` object, a sub-query will be generated and used as the range of

the column or DB expression. For example, `['in', 'id', [1, 2, 3]]` will generate `id IN (1, 2, 3)`. The method will properly quote the column name and escape values in the range. The `in` operator also supports composite columns. In this case, operand 1 should be an array of the columns, while operand 2 should be an array of arrays or a `Query` object representing the range of the columns.

- **not in**: similar to the `in` operator except that `IN` is replaced with `NOT IN` in the generated condition.
- **like**: operand 1 should be a column or DB expression, and operand 2 be a string or an array representing the values that the column or DB expression should be like. For example, `['like', 'name', 'tester']` will generate `name LIKE '%tester%'`. When the value range is given as an array, multiple `LIKE` predicates will be generated and concatenated using `AND`. For example, `['like', 'name', ['test', 'sample']]` will generate `name LIKE '%test%' AND name LIKE '%sample%'`. You may also provide an optional third operand to specify how to escape special characters in the values. The operand should be an array of mappings from the special characters to their escaped counterparts. If this operand is not provided, a default escape mapping will be used. You may use `false` or an empty array to indicate the values are already escaped and no escape should be applied. Note that when using an escape mapping (or the third operand is not provided), the values will be automatically enclosed within a pair of percentage characters.

Note: When using PostgreSQL you may also use `ilike` instead of `like` for case-insensitive matching.

- **or like**: similar to the `like` operator except that `OR` is used to concatenate the `LIKE` predicates when operand 2 is an array.
- **not like**: similar to the `like` operator except that `LIKE` is replaced with `NOT LIKE` in the generated condition.
- **or not like**: similar to the `not like` operator except that `OR` is used to concatenate the `NOT LIKE` predicates.
- **exists**: requires one operand which must be an instance of `[[yii\db\Query]]` representing the sub-query. It will build a `EXISTS (sub-query)` expression.
- **not exists**: similar to the `exists` operator and builds a `NOT EXISTS (sub-query)` expression.

Additionally you can specify anything as operator:

```
$userQuery = (new Query)->select('id')->from('user');  
$query->where(['>=', 'id', 10]);
```

It will result in:

```
SELECT id FROM user WHERE id >= 10;
```

If you are building parts of condition dynamically it's very convenient to use `andWhere()` and `orWhere()`:

```
$status = 10;
```

```

$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {

    $query->andWhere(['like', 'title', $search]);

}

```

In case `$search` isn't empty the following SQL will be generated:

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Building Filter Conditions

When building filter conditions based on user inputs, you usually want to specially handle "empty inputs" by ignoring them in the filters. For example, you have an HTML form that takes username and email inputs. If the user only enters something in the username input, you may want to build a query that only tries to match the entered username. You may use the `filterWhere()` method achieve this goal:

```

// $username and $email are from user inputs

$query->filterWhere([

    'username' => $username,

    'email' => $email,

]);

```

The `filterWhere()` method is very similar to `where()`. The main difference is that `filterWhere()` will remove empty values from the provided condition. So if `$email` is "empty", the resulting query will be `...WHERE username=:username`; and if both `$username` and `$email` are "empty", the query will have no `WHERE` part.

A value is empty if it is null, an empty string, a string consisting of whitespaces, or an empty array.

You may also use `andFilterWhere()` and `orWhereWhere()` to append more filter conditions.

ORDER BY

For ordering results `orderBy` and `addOrderBy` could be used:

```

$query->orderBy([

    'id' => SORT_ASC,

    'name' => SORT_DESC,

]);

```

Here we are ordering by `id` ascending and then by `name` descending.

GROUP BY and HAVING

In order to add `GROUP BY` to generated SQL you can use the following:

```
$query->groupBy('id, status');
```

If you want to add another field after using `groupBy`:

```
$query->addGroupBy(['created_at', 'updated_at']);
```

To add a `HAVING` condition the corresponding `having` method and its `andHaving` and `orHaving` can be used. Parameters for these are similar to the ones for `where` methods group:

```
$query->having(['status' => $status]);
```

LIMIT and OFFSET

To limit result to 10 rows `limit` can be used:

```
$query->limit(10);
```

To skip 100 first rows use:

```
$query->offset(100);
```

JOIN

The `JOIN` clauses are generated in the Query Builder by using the applicable join method:

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

This left join selects data from two related tables in one query:

```
$query->select(['user.name AS author', 'post.title as title'])  
->from('user')  
->leftJoin('post', 'post.user_id = user.id');
```

In the code, the `leftJoin()` method's first parameter specifies the table to join to. The second parameter defines the join condition.

If your database application supports other join types, you can use those via the generic `join` method:

```
$query->join('FULL OUTER JOIN', 'post', 'post.user_id = user.id');
```

The first argument is the join type to perform. The second is the table to join to, and the

third is the condition.

Like **FROM**, you may also join with sub-queries. To do so, specify the sub-query as an array which must contain one element. The array value must be a **Query** object representing the sub-query, while the array key is the alias for the sub-query. For example,

```
$query->leftJoin(['u' => $subQuery], 'u.id=author_id');
```

UNION

UNION in SQL adds results of one query to results of another query. Columns returned by both queries should match. In Yii in order to build it you can first form two query objects and then use **union** method:

```
$query = new Query();  
  
$query->select("id, category_id as type, name")->from('post')->limit(10);  
  
$anotherQuery = new Query();  
$anotherQuery->select('id, type, name')->from('user')->limit(10);  
  
$query->union($anotherQuery);
```

Batch Query

When working with large amount of data, methods such as `[[yii\db\Query::all()]]` are not suitable because they require loading all data into the memory. To keep the memory requirement low, Yii provides the so-called batch query support. A batch query makes uses of data cursor and fetches data in batches.

Batch query can be used like the following:

```
use yii\db\Query;  
  
$query = (new Query())  
    ->from('user')  
    ->orderBy('id');  
  
foreach ($query->batch() as $users) {  
    // $users is an array of 100 or fewer rows from the user table  
}
```



```
// or if you want to iterate the row one by one

foreach ($query->each() as $user) {

    // $user represents one row of data from the user table

}
```

The method `[[yii\db\Query::batch()]]` and `[[yii\db\Query::each()]]` return an `[[yii\db\BatchQueryResult]]` object which implements the `Iterator` interface and thus can be used in the `foreach` construct. During the first iteration, a SQL query is made to the database. Data are since then fetched in batches in the iterations. By default, the batch size is 100, meaning 100 rows of data are being fetched in each batch. You can change the batch size by passing the first parameter to the `batch()` or `each()` method.

Compared to the `[[yii\db\Query::all()]]`, the batch query only loads 100 rows of data at a time into the memory. If you process the data and then discard it right away, the batch query can help keep the memory usage under a limit.

If you specify the query result to be indexed by some column via `[[yii\db\Query::indexBy()]]`, the batch query will still keep the proper index. For example,

```
use yii\db\Query;

$query = (new Query())

    ->from('user')

    ->indexBy('username');

foreach ($query->batch() as $users) {

    // $users is indexed by the "username" column

}

foreach ($query->each() as $username => $user) {

}
```

活动记录 **Active Record**

注意：该章节还在开发中。

Active Record（活动记录，以下简称 **AR**）提供了一个面向对象的接口，用以访问数据库中的数据。一个 **AR** 类关联一张数据表，每个 **AR** 对象对应表中的一行，对象的属性（即 **AR** 的特性 **Attribute**）映

射到数据行的对应列。一条活动记录（AR 对象）对应数据表的一行，AR 对象的属性则映射该行的相应列。您可以直接以面向对象的方式来操纵数据表中的数据，妈妈再也不用担心我需要写原生 SQL 语句啦。

例如，假定 **Customer** AR 类关联着 **customer** 表，且该类的 **name** 属性代表 **customer** 表的 **name** 列。你可以写以下代码来在 **customer** 表里插入一行新的记录：

用 AR 而不是原生的 SQL 语句去执行数据库查询，可以调用直观方法来实现相同目标。如，调用 `[[yii\db\ActiveRecord::save()|save()]]` 方法将执行插入或更新轮询，将在该 AR 类关联的数据表新建或更新一行数据：

```
$customer = new Customer();  
  
$customer->name = '李狗蛋';  
  
$customer->save(); // 一行新数据插入 customer 表
```

上面的代码和使用下面的原生 SQL 语句是等效的，但显然前者更直观，更不易出错，并且面对不同的数据库系统（DBMS, Database Management System）时更不容易产生兼容性问题。

```
$db->createCommand('INSERT INTO customer (name) VALUES (:name)', [  
  
    ':name' => '李狗蛋',  
  
])->execute();
```

下面是所有目前被 Yii 的 AR 功能所支持的数据库列表：

- MySQL 4.1 及以上：通过 `[[yii\db\ActiveRecord]]`
- PostgreSQL 7.3 及以上：通过 `[[yii\db\ActiveRecord]]`
- SQLite 2 和 3：通过 `[[yii\db\ActiveRecord]]`
- Microsoft SQL Server 2010 及以上：通过 `[[yii\db\ActiveRecord]]`
- Oracle：通过 `[[yii\db\ActiveRecord]]`
- CUBRID 9.1 及以上：通过 `[[yii\db\ActiveRecord]]`
- Sphinx：通过 `[[yii\sphinx\ActiveRecord]]`，需求 `yii2-sphinx` 扩展
- ElasticSearch：通过 `[[yii\elasticsearch\ActiveRecord]]`，需求 `yii2-elasticsearch` 扩展
- Redis 2.6.12 及以上：通过 `[[yii\redis\ActiveRecord]]`，需求 `yii2-redis` 扩展
- MongoDB 1.3.0 及以上：通过 `[[yii\mongodb\ActiveRecord]]`，需求 `yii2-mongodb` 扩展

如你所见，Yii 不仅提供了对关系型数据库的 AR 支持，还提供了 NoSQL 数据库的支持。在这个教程中，我们会主要描述对关系型数据库的 AR 用法。然而，绝大多数的内容在 NoSQL 的 AR 里同样适用。

声明 AR 类

要想声明一个 AR 类，你需要扩展 `[[yii\db\ActiveRecord]]` 基类，并实现 `tableName` 方法，返回与之相关联的数据表的名称：

```

namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    /**
     * @return string 返回该 AR 类关联的数据表名
     */
    public static function tableName()
    {
        return 'customer';
    }
}

```

访问列数据

AR 把相应数据行的每一个字段映射为 AR 对象的一个个特性变量（Attribute） 一个特性就好像一个普通对象的公共属性一样（public property）。特性变量的名称和对应字段的名称是一样的，且大小姓名。

使用以下语法读取列的值：

```

// "id" 和 "mail" 是 $customer 对象所关联的数据表的对应字段名

$id = $customer->id;

$email = $customer->email;

```

要改变列值，只要给关联属性赋新值并保存对象即可：

```

$customer->email = '哪吒@example.com';

$customer->save();

```

建立数据库连接

AR 用一个 `[[yii\db\Connection|DB connection]]` 对象与数据库交换数据。默认的，它使用 `db` 组件作为其连接对象。详见[数据库基础](#)章节， 你可以在应用程序配置文件中设置下 `db` 组件，就像这样，

```
return [  
  
    'components' => [  
  
        'db' => [  
  
            'class' => 'yii\db\Connection',  
  
            'dsn' => 'mysql:host=localhost;dbname=testdb',  
  
            'username' => 'demo',  
  
            'password' => 'demo',  
  
        ],  
  
    ],  
  
];
```

如果在你的应用中应用了不止一个数据库，且你需要给你的 AR 类使用不同的数据库链接（DB connection）， 你可以覆盖掉 `[[yii\db\ActiveRecord::getDb()|getDb()]]` 方法：

```
class Customer extends ActiveRecord  
{  
  
    // ...  
  
    public static function getDb()  
    {  
  
        return \Yii::$app->db2; // 使用名为 "db2" 的应用组件  
  
    }  
  
}
```

查询数据

AR 提供了两种方法来构建 DB 查询并向 AR 实例里填充数据：

- `[[yii\db\ActiveRecord::find()]]`

•[[yii\db\ActiveRecord::findBySql()]]

以上两个方法都会返回 [[yii\db\ActiveQuery]] 实例，该类继承自[[yii\db\Query]]， 因此，他们都支持同一套灵活且强大的 DB 查询方法，如 `where()`，`join()`，`orderBy()`，等等。 下面的这些案例展示了一些可能的玩法：

```
// 取回所有活跃客户(状态为 *active* 的客户) 并以他们的 ID 排序：
```

```
$customers = Customer::find()

    ->where(['status' => Customer::STATUS_ACTIVE])

    ->orderBy('id')

    ->all();
```

```
// 返回 ID 为 1 的客户：
```

```
$customer = Customer::find()

    ->where(['id' => 1])

    ->one();
```

```
// 取回活跃客户的数量：
```

```
$count = Customer::find()

    ->where(['status' => Customer::STATUS_ACTIVE])

    ->count();
```

```
// 以客户 ID 索引结果集：
```

```
$customers = Customer::find()->indexBy('id')->all();
```

```
// $customers 数组以 ID 为索引
```

```
// 用原生 SQL 语句检索客户：
```

```
$sql = 'SELECT * FROM customer';
```

```
$customers = Customer::findBySql($sql)->all();
```

小技巧：在上面的代码中，`Customer::STATUS_ACTIVE` 是一个在 `Customer` 类里定义的常量。（译注：这种常量的值一般都是 `tinyint`）相较于直接在代码中写死字符串或数字，使用一个更有意义的常量名称是一种更好的编

程习惯。

有两个快捷方法：`findOne` 和 `findAll()` 用来返回一个或者一组 `ActiveRecord` 实例。前者返回第一个匹配的实例，后者返回所有。例如：

```
// 返回 id 为 1 的客户

$customer = Customer::findOne(1);

// 返回 id 为 1 且状态为 *active* 的客户

$customer = Customer::findOne([

    'id' => 1,

    'status' => Customer::STATUS_ACTIVE,

]);

// 返回 id 为 1、2、3 的一组客户

$customers = Customer::findAll([1, 2, 3]);

// 返回所有状态为 "deleted" 的客户

$customer = Customer::findAll([

    'status' => Customer::STATUS_DELETED,

]);
```

以数组形式获取数据

有时候，我们需要处理大量的数据，这时可能需要用一个数组来存储取到的数据，从而节省内存。你可以用 `asArray()` 函数做到这一点：

```
// 以数组而不是对象形式取回客户信息：

$customers = Customer::find()

    ->asArray()

    ->all();

// $customers 的每个元素都是键值对数组
```

批量获取数据

在 [Query Builder \(查询构造器\)](#) 里，我们已经解释了当需要从数据库中查询大量数据时，你可以用 `batch query` (批量查询) 来限制内存的占用。 你可能也想在 **AR** 里使用相同的技巧，比如这样…

```
// 一次提取 10 个客户信息

foreach (Customer::find()->batch(10) as $customers) {

    // $customers 是 10 个或更少的客户对象的数组

}

// 一次提取 10 个客户并一个一个地遍历处理

foreach (Customer::find()->each(10) as $customer) {

    // $customer 是一个 "Customer" 对象

}

// 贪婪加载模式的批处理查询

foreach (Customer::find()->with('orders')->each() as $customer) {

}
```

操作数据

AR 提供以下方法插入、更新和删除与 **AR** 对象关联的那张表中的某一行：

- `[[yii\db\ActiveRecord::save()|save()]]`
- `[[yii\db\ActiveRecord::insert()|insert()]]`
- `[[yii\db\ActiveRecord::update()|update()]]`
- `[[yii\db\ActiveRecord::delete()|delete()]]`

AR 同时提供了一下静态方法，可以应用在与某 **AR** 类所关联的整张表上。 用这些方法的时候千万要小心，因为他们作用于整张表！ 比如，`deleteAll()` 会删除掉表里所有的记录。

- `[[yii\db\ActiveRecord::updateCounters()|updateCounters()]]`
- `[[yii\db\ActiveRecord::updateAll()|updateAll()]]`
- `[[yii\db\ActiveRecord::updateAllCounters()|updateAllCounters()]]`
- `[[yii\db\ActiveRecord::deleteAll()|deleteAll()]]`

下面的这些例子里，详细展现了如何使用这些方法：

```
// 插入新客户的记录

$customer = new Customer();
```

```

$customer->name = 'James';

$customer->email = 'james@example.com';

$customer->save(); // 等同于 $customer->insert();

// 更新现有客户记录

$customer = Customer::findOne($id);

$customer->email = 'james@example.com';

$customer->save(); // 等同于 $customer->update();

// 删除已有客户记录

$customer = Customer::findOne($id);

$customer->delete();

// 删除多个年龄大于 20，性别为男（Male）的客户记录

Customer::deleteAll('age > :age AND gender = :gender', [':age' => 20, ':gender' => 'M']);

// 所有客户的 age（年龄）字段加 1:

Customer::updateAllCounters(['age' => 1]);

```

须知: `save()` 方法会调用 `insert()` 和 `update()` 中的一个，用哪个取决于当前 AR 对象是不是新对象（在函数内部，他会检查 `[[yii\db\ActiveRecord::isNewRecord]]` 的值）。若 AR 对象是由 `new` 操作符 初始化出来的，`save()` 方法会在表里插入一条数据；如果一个 AR 是由 `find()` 方法获取来的，则 `save()` 会更新表里的对应行记录。

数据输入与有效性验证

由于 AR 继承自 `[[yii\base\Model]]`，所以它同样也支持 `Model` 的数据输入、验证等特性。例如，你可以声明一个 `rules` 方法用来覆盖掉 `[[yii\base\Model::rules()|rules()]]` 里的；你也可以给 AR 实例批量赋值；你也可以通过调用 `[[yii\base\Model::validate()|validate()]]` 执行数据验证。

当你调用 `save()`、`insert()`、`update()` 这三个方法时，会自动调用 `[[yii\base\Model::validate()|validate()]]` 方法。如果验证失败，数据将不会保存进数据库。

下面的例子演示了如何使用 AR 获取/验证用户输入的数据并将他们保存进数据库：

```

// 新建一条记录

```



```

$model = new Customer;

if ($model->load(Yii::$app->request->post()) && $model->save()) {

    // 获取用户输入的数据，验证并保存

}

// 更新主键为$id 的 AR

$model = Customer::findOne($id);

if ($model === null) {

    throw new NotFoundException;

}

if ($model->load(Yii::$app->request->post()) && $model->save()) {

    // 获取用户输入的数据，验证并保存

}

```

读取默认值

你的表列也许定义了默认值。有时候，你可能需要在使用 **web** 表单的时候给 **AR** 预设一些值。如果你需要这样做，可以在显示表单内容前通过调用 `loadDefaultValues()` 方法来实现：

```

`php $customer =
new Customer(); $customer->loadDefaultValues(); // ... 渲染 $customer 的 HTML 表单 ... `

```

AR 的生命周期

理解 **AR** 的生命周期对于你操作数据库非常重要。生命周期通常都会有些典型的事件存在。对于开发 **AR** 的 **behaviors** 来说非常有用。

当你实例化一个新的 **AR** 对象时，我们将获得如下的生命周期：

1.constructor

2. `[[yii\db\ActiveRecord::init()|init()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]]` 事件

当你通过 `[[yii\db\ActiveRecord::find()|find()]]` 方法查询数据时，每个 **AR** 实例都将有以下生命周期：

1.constructor

2. `[[yii\db\ActiveRecord::init()|init()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]]` 事件

3. `[[yii\db\ActiveRecord::afterFind()|afterFind()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_AFTER_FIND|EVENT_AFTER_FIND]]` 事件

当通过 `[[yii\db\ActiveRecord::save()|save()]]` 方法写入或者更新数据时, 我们将获得如下生命周期:

- 1.`[[yii\db\ActiveRecord::beforeValidate()|beforeValidate()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]]` 事件
- 2.`[[yii\db\ActiveRecord::afterValidate()|afterValidate()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_AFTER_VALIDATE|EVENT_AFTER_VALIDATE]]` 事件
- 3.`[[yii\db\ActiveRecord::beforeSave()|beforeSave()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_BEFORE_INSERT|EVENT_BEFORE_INSERT]]` 或 `[[yii\db\ActiveRecord::EVENT_BEFORE_UPDATE|EVENT_BEFORE_UPDATE]]` 事件
- 4.执行实际的数据写入或更新
- 5.`[[yii\db\ActiveRecord::afterSave()|afterSave()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]]` 或 `[[yii\db\ActiveRecord::EVENT_AFTER_UPDATE|EVENT_AFTER_UPDATE]]` 事件

最后, 当调用 `[[yii\db\ActiveRecord::delete()|delete()]]` 删除数据时, 我们将获得如下生命周期:

- 1.`[[yii\db\ActiveRecord::beforeDelete()|beforeDelete()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_BEFORE_DELETE|EVENT_BEFORE_DELETE]]` 事件
- 2.执行实际的数据删除
- 3.`[[yii\db\ActiveRecord::afterDelete()|afterDelete()]]`: 会触发一个 `[[yii\db\ActiveRecord::EVENT_AFTER_DELETE|EVENT_AFTER_DELETE]]` 事件

查询关联的数据

使用 `AR` 方法也可以查询数据表的关联数据（如, 选出表 **A** 的数据可以拉出表 **B** 的关联数据）。有了 `AR`, 返回的关联数据连接就像连接关联主表的 `AR` 对象的属性一样。

建立关联关系后, 通过 `$customer->orders` 可以获取 一个 `Order` 对象的数组, 该数组代表当前客户对象的订单集。

定义关联关系使用一个可以返回 `[[yii\db\ActiveQuery]]` 对象的 `getter` 方法, `[[yii\db\ActiveQuery]]`对象有关联上下文的相关信息, 因此可以只查询关联数据。

例如:

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        // 客户和订单通过 Order.customer_id -> id 关联建立一对多关系

        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

```

}

class Order extends \yii\db\ActiveRecord
{
    // 订单和客户通过 Customer.id -> customer_id 关联建立一对一关系

    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id']);
    }
}

```

以上使用了 `[[yii\db\ActiveRecord::hasMany()]]` 和 `[[yii\db\ActiveRecord::hasOne()]]` 方法。以上两例分别是关联数据多对一关系和一对一关系的建模范例。如，一个客户有很多订单，一个订单只归属一个客户。两个方法都有两个参数并返回 `[[yii\db\ActiveQuery]]` 对象。

- **\$class**: 关联模型类名，它必须是一个完全合格的类名。
- **\$link**: 两个表的关联列，应为键值对数组的形式。数组的键是 **\$class** 关联表的列名，而数组值是关联类 **\$class** 的列名。基于表外键定义关联关系是最佳方法。

建立关联关系后，获取关联数据和获取组件属性一样简单，执行以下相应 **getter** 方法即可：

```

// 取得客户的订单

$customer = Customer::findOne(1);

$orders = $customer->orders; // $orders 是 Order 对象数组

```

以上代码实际执行了以下两条 SQL 语句：

```

SELECT * FROM customer WHERE id=1;

SELECT * FROM order WHERE customer_id=1;

```

提示:再次用表达式 **\$customer->orders** 将不会执行第二次 SQL 查询，SQL 查询只在该表达式第一次使用时执行。数据库访问只返回缓存在内部前一次取回的结果集，如果你想查询新的 关联数据，先要注销现有结果集：
unset(\$customer->orders);。

有时候需要在关联查询中传递参数，如不需要返回客户全部订单，只需要返回购买金额超过设定值的大订单，通过以下 **getter** 方法声明一个关联数据 **bigOrders**：

```

class Customer extends \yii\db\ActiveRecord
{
    public function getBigOrders($threshold = 100)

```

```

{

    return $this->hasMany(Order::className(), ['customer_id' => 'id'])

        ->where('subtotal > :threshold', [':threshold' => $threshold])

        ->orderBy('id');

}
}

```

`hasMany()` 返回 `[[yii\db\ActiveQuery]]` 对象，该对象允许你通过 `[[yii\db\ActiveQuery]]` 方法定制查询。

如上声明后，执行 `$customer->bigOrders` 就返回 总额大于 100 的订单。使用以下代码更改设定值：

```
$orders = $customer->getBigOrders(200)->all();
```

>注意：关联查询返回的是 `[[yii\db\ActiveQuery]]` 的实例，如果像特性（如类属性）那样连接关联数据，返回的结果是关联查询的结果，即 `[[yii\db\ActiveRecord]]` 的实例，或者是数组，或者是 `null`，取决于关联关系的多样性。如，`$customer->getOrders()` 返回 `ActiveQuery` 实例，而 `$customer->orders` 返回 `Order` 对象数组（如果查询结果为空则返回空数组）。

中间关联表

有时，两个表通过中间表关联，定义这样的关联关系，可以通过调用 `[[yii\db\ActiveQuery::via()|via()]]` 方法或 `[[yii\db\ActiveQuery::viaTable()|viaTable()]]` 方法来定制 `[[yii\db\ActiveQuery]]` 对象。

举例而言，如果 `order` 表和 `item` 表通过中间表 `order_item` 关联起来，可以在 `Order` 类声明 `items` 关联关系取代中间表：

```

class Order extends \yii\db\ActiveRecord
{

    public function getItems()

    {

        return $this->hasMany(Item::className(), ['id' => 'item_id'])

            ->viaTable('order_item', ['order_id' => 'id']);

    }

}

```

两个方法是相似的，除了 `[[yii\db\ActiveQuery::via()|via()]]` 方法的第一个参数是使用 `AR` 类中定义的关联名。以上方法取代了中间表，等价于：

```
class Order extends \yii\db\ActiveRecord
```

```

{

    public function getOrderItems()

    {

        return $this->hasMany(OrderItem::className(), ['order_id' => 'id']);

    }

    public function getItems()

    {

        return $this->hasMany(Item::className(), ['id' => 'item_id'])

            ->via('orderItems');

    }

}

```

延迟加载和即时加载（又称惰性加载与贪婪加载）

如前所述，当你第一次连接关联对象时，AR 将执行一个数据库查询来检索请求数据并填充到关联对象的相应属性。如果再次连接相同的关联对象，不再执行任何查询语句，这种数据库查询的执行方法称为“延迟加载”。如：

```

// SQL executed: SELECT * FROM customer WHERE id=1

$customer = Customer::findOne(1);

// SQL executed: SELECT * FROM order WHERE customer_id=1

$orders = $customer->orders;

// 没有 SQL 语句被执行

$orders2 = $customer->orders; //取回上次查询的缓存数据

```

延迟加载非常实用，但是，在以下场景中使用延迟加载会遭遇性能问题：

```

// SQL executed: SELECT * FROM customer LIMIT 100

$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {

```

```
// SQL executed: SELECT * FROM order WHERE customer_id=...

$orders = $customer->orders;

// ...处理 $orders...

}
```

假设数据库查出的客户超过 100 个，以上代码将执行多少条 SQL 语句？ 101 条！第一条 SQL 查询语句取回 100 个客户，然后，每个客户要执行一条 SQL 查询语句以取回该客户的所有订单。

为解决以上性能问题，可以通过调用 `[[yii\db\ActiveQuery::with()]]` 方法使用即时加载解决。

```
// SQL executed: SELECT * FROM customer LIMIT 100;

//      SELECT * FROM orders WHERE customer_id IN (1,2,...)

$customers = Customer::find()->limit(100)

    ->with('orders')->all();

foreach ($customers as $customer) {

    // 没有 SQL 语句被执行

    $orders = $customer->orders;

    // ...处理 $orders...

}
```

如你所见，同样的任务只需要两个 SQL 语句。 >须知：通常，即时加载 N 个关联关系而通过 `via()` 或者 `viaTable()` 定义了 M 个关联关系，将有 $1+M+N$ 条 SQL 查询语句被执行：一个查询取回主表行数，一个查询给每一个 (M) 中间表，一个查询给每个 (N) 关联表。注意：当用即时加载定制 `select()` 时，确保连接到关联模型的列都被包括了，否则，关联模型不会载入。如：

```
$orders = Order::find()->select(['id', 'amount'])->with('customer')->all();

// $orders[0]->customer 总是空的，使用以下代码解决这个问题：

$orders = Order::find()->select(['id', 'amount', 'customer_id'])->with('customer')->all();
```

有时候，你想自由的自定义关联查询，延迟加载和即时加载都可以实现，如：

```
$customer = Customer::findOne(1);

// 延迟加载: SELECT * FROM order WHERE customer_id=1 AND subtotal>100

$orders = $customer->getOrders()->where('subtotal>100')->all();

// 即时加载: SELECT * FROM customer LIMIT 100

//      SELECT * FROM order WHERE customer_id IN (1,2,...) AND subtotal>100
```

```
$customers = Customer::find()->limit(100)->with([
    'orders' => function($query) {
        $query->andWhere('subtotal>100');
    },
])->all();
```

逆关系

关联关系通常成对定义，如：Customer 可以有个名为 orders 关联项，而 Order 也有个名为 customer 的关联项：

```
class Customer extends ActiveRecord
{
    ....

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    ....

    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id']);
    }
}
```

如果我们执行以下查询，可以发现订单的 customer 和 找到这些订单的客户对象并不是同一个。连接 customer->orders 将触发一条 SQL 语句 而连接一个订单的 customer 将触发另一条 SQL 语句。

```
// SELECT * FROM customer WHERE id=1
```

```

$customer = Customer::findOne(1);

// 输出 "不相同"

// SELECT * FROM order WHERE customer_id=1

// SELECT * FROM customer WHERE id=1

if ($customer->orders[0]->customer === $customer) {

    echo '相同';

} else {

    echo '不相同';

}

```

为避免多余执行的后一条语句，我们可以为 `customer` 或 `orders` 关联关系定义相反的关联关系，通过调用 `[[yii\db\ActiveQuery::inverseOf()|inverseOf()]]` 方法可以实现。

```

class Customer extends ActiveRecord
{
    ....

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' =>
'id'])->inverseOf('customer');
    }
}

```

现在我们同样执行上面的查询，我们将得到：

```

// SELECT * FROM customer WHERE id=1

$customer = Customer::findOne(1);

// 输出相同

// SELECT * FROM order WHERE customer_id=1

if ($customer->orders[0]->customer === $customer) {

    echo '相同';

} else {

    echo '不相同';

}

```


以上我们展示了如何在延迟加载中使用相对关联关系， 相对关系也可以用在即时加载中：

```
// SELECT * FROM customer

// SELECT * FROM order WHERE customer_id IN (1, 2, ...)

$customers = Customer::find()->with('orders')->all();

// 输出相同

if ($customers[0]->orders[0]->customer === $customers[0]) {

    echo '相同';

} else {

    echo '不相同';

}
```

>注意:相对关系不能在包含中间表的关联关系中定义。 即是，如果你的关系是通过 `[[yii\db\ActiveQuery::via()|via()]]` 或 `[[yii\db\ActiveQuery::viaTable()|viaTable()]]` 方法定义的，就不能调用 `[[yii\db\ActiveQuery::inverseOf()]]` 方法了。

JOIN 类型关联查询

使用关系数据库时，普遍要做的是连接多个表并明确地运用各种 JOIN 查询。 JOIN SQL 语句的查询条件和参数，使用 `[[yii\db\ActiveQuery::joinWith()]]` 可以重用已定义关系并调用 而不是使用 `[[yii\db\ActiveQuery::join()]]` 来实现目标。

```
// 查找所有订单并以客户 ID 和订单 ID 排序，并贪婪加载 "customer" 表

$orders = Order::find()->joinWith('customer')->orderBy('customer.id, order.id')->all();

// 查找包括书籍的所有订单，并以 `INNER JOIN` 的连接方式即时加载 "books" 表

$orders = Order::find()->innerJoinWith('books')->all();
```

以上，方法 `[[yii\db\ActiveQuery::innerJoinWith()|innerJoinWith()]]` 是访问 **INNER JOIN** 类型的 `[[yii\db\ActiveQuery::joinWith()|joinWith()]]` 的快捷方式。

可以连接一个或多个关联关系，可以自由使用查询条件到关联查询， 也可以嵌套连接关联查询。如：

```
// 连接多重关系

// 找出 24 小时内注册客户包含书籍的订单

$orders = Order::find()->innerJoinWith([

    'books',

    'customer' => function ($query) {

        $query->where('customer.created_at > ' . (time() - 24 * 3600));

    }

]);
```

```

    }

    ]->all();

    // 连接嵌套关系：连接 books 表及其 author 列

    $orders = Order::find()->joinWith('books.author')->all();

```

代码背后，Yii 先执行一条 JOIN SQL 语句把满足 JOIN SQL 语句查询条件的主要模型查出，然后为每个关系执行一条查询语句，bing 填充相应的关联记录。

[[yii\db\ActiveQuery::joinWith()|joinWith()]] 和 [[yii\db\ActiveQuery::with()|with()]] 的区别是前者连接主模型类和关联模型类的数据表来检索主模型，而后者只查询和检索主模型类。检索主模型

由于这个区别，你可以应用只针对一条 JOIN SQL 语句起效的查询条件。如，通过关联模型的查询条件过滤主模型，如前例，可以使用关联表的列来挑选主模型数据，

当使用 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 方法时可以响应没有歧义的列名。In the above examples, we use `item.id` and `order.id` to disambiguate the `id` column references 因为订单表和项目表都包括 `id` 列。

当连接关联关系时，关联关系默认使用即时加载。你可以 通过传参数 `$eagerLoading` 来决定在指定关联查询中是否使用即时加载。

默认 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 使用左连接来连接关联表。你也可以传 `$joinType` 参数来定制连接类型。你也可以使用 [[yii\db\ActiveQuery::innerJoinWith()|innerJoinWith()]]。

以下是 **INNER JOIN** 的简短例子：

```

// 查找包括书籍的所有订单，但 "books" 表不使用即时加载

$orders = Order::find()->innerJoinWith('books', false)->all();

// 等价于：

$orders = Order::find()->joinWith('books', false, 'INNER JOIN')->all();

```

有时连接两个表时，需要在关联查询的 ON 部分指定额外条件。这可以通过调用 [[yii\db\ActiveQuery::onCondition()]] 方法实现：

```

class User extends ActiveRecord
{
    public function getBooks()
    {
        return $this->hasMany(Item::className(), ['owner_id' =>
'id'])->onCondition(['category_id' => 1]);
    }
}

```

在上面，`[[yii\db\ActiveRecord::hasMany()|hasMany()]]` 方法回传了一个 `[[yii\db\ActiveQuery]]` 对象，当你用 `[[yii\db\ActiveQuery::joinWith()|joinWith()]]` 执行一条查询时，取决于正被调用的是哪个 `[[yii\db\ActiveQuery::onCondition()|onCondition()]]`，返回 `category_id` 为 1 的 items

当你用 `[[yii\db\ActiveQuery::joinWith()|joinWith()]]` 进行一次查询时，“on-condition”条件会被放置在相应查询语句的 ON 部分，如：

```
// SELECT user.* FROM user LEFT JOIN item ON item.owner_id=user.id AND category_id=1
// SELECT * FROM item WHERE owner_id IN (...) AND category_id=1
$users = User::find()->joinWith('books')->all();
```

注意：如果通过 `[[yii\db\ActiveQuery::with()]]` 进行贪婪加载或使用惰性加载的话，则 on 条件会被放置在对应 SQL 语句的 WHERE 部分。因为，此时此处并没有发生 JOIN 查询。比如：

```
// SELECT * FROM user WHERE id=10
$user = User::findOne(10);
// SELECT * FROM item WHERE owner_id=10 AND category_id=1
$books = $user->books;
```

关联表操作

AR 提供了下面两个方法来建立和解除两个关联对象之间的关系：

- `[[yii\db\ActiveRecord::link()|link()]]`
- `[[yii\db\ActiveRecord::unlink()|unlink()]]`

例如，给定一个 customer 和 order 对象，我们可以通过下面的代码使得 customer 对象拥有 order 对象：

```
$customer = Customer::findOne(1);
$order = new Order();
$order->subtotal = 100;
$customer->link('orders', $order);
```

`[[yii\db\ActiveRecord::link()|link()]]` 调用上述将设置 `customer_id` 的顺序是 `$customer` 的主键值，然后调用 `[[yii\db\ActiveRecord::save()|save()]]` 要将顺序保存到数据库中。

作用域

当你调用 `[[yii\db\ActiveRecord::find()|find()]]` 或 `[[yii\db\ActiveRecord::findBySql()|findBySql()]]` 方法时，将会返回一个 `[[yii\db\ActiveQuery|ActiveQuery]]` 实例。之后，你可以调用其他查询方法，如 `[[yii\db\ActiveQuery::where()|where()]]`，`[[yii\db\ActiveQuery::orderBy()|orderBy()]]`，进一步的指定查询条件。

有时候你可能需要在不同的地方使用相同的查询方法。如果出现这种情况，你应该考虑定义所谓的作用域。作用域是本质上要求一组查询方法来修改查询对象的自定义查询类中定义的方法。之后你就可以像使用普通方法一样使用作用域。

只需两步即可定义一个作用域。首先给你的 **model** 创建一个自定义的查询类，在此类中定义的所需的范围方法。例如，给 **Comment** 模型创建一个 **CommentQuery** 类，然后在 **CommentQuery** 类中定义一个 **active()** 的方法为作用域，像下面的代码：

```
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    public function active($state = true)
    {
        $this->andWhere(['active' => $state]);

        return $this;
    }
}
```

重点：

- 1.类必须继承 **yii\db\ActiveQuery** (或者是其他的 **ActiveQuery**，比如 **yii\mongodb\ActiveQuery**)。
- 2.必须是一个 **public** 类型的方法且必须返回 **\$this** 实现链式操作。可以传入参数。
- 3.检查 **[[yii\db\ActiveQuery]]** 对于修改查询条件是非常有用的方法。

其次，覆盖 **[[yii\db\ActiveRecord::find()]]** 方法使其返回自定义的查询对象而不是常规的 **[[yii\db\ActiveQuery|ActiveQuery]]**。对于上述例子，你需要编写如下代码：

```
namespace app\models;

use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    /**
```

```

* @inheritdoc

* @return CommentQuery

*/

public static function find()
{
    return new CommentQuery(get_called_class());
}
}

```

就这样，现在你可以使用自定义的作用域方法了：

```

$comments = Comment::find()->active()->all();

$inactiveComments = Comment::find()->active(false)->all();

```

你也能在定义的关联里使用作用域方法，比如：

```

class Post extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::className(), ['post_id' => 'id']->active());
    }
}

```

或者在执行关联查询的时候使用（on-the-fly 是啥？）：

```

$posts = Post::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();

```

默认作用域

如果你之前用过 Yii 1.1 就应该知道默认作用域的概念。一个默认的作用域可以作用于所有查询。你可以很容易的通过重写 `[[yii\db\ActiveRecord::find()]]` 方法来定义一个默认作用域，例如：

```
public static function find()
{
    return parent::find()->where(['deleted' => false]);
}
```

注意，你之后所有的查询都不能用 `[[yii\db\ActiveQuery::where()|where()]]`，但是可以用 `[[yii\db\ActiveQuery::andWhere()|andWhere()]]` 和 `[[yii\db\ActiveQuery::orWhere()|orWhere()]]`，他们不会覆盖掉默认作用域。（译注：如果你要使用默认作用域，就不能在 `xxx::find()`后使用 `where()`方法，你必须使用 `andXXX()`或者 `orXXX()`系的方法，否则默认作用域不会起效果，至于原因，打开 `where()`方法的代码一看便知）

事务操作

当执行几个相关联的数据库操作的时候

TODO: FIXME: WIP, TBD, <https://github.com/yiisoft/yii2/issues/226>

, `[[yii\db\ActiveRecord::afterSave()|afterSave()]]`,
`[[yii\db\ActiveRecord::beforeDelete()|beforeDelete()]]` and/or
`[[yii\db\ActiveRecord::afterDelete()|afterDelete()]]` 生命周期周期方法(life cycle methods 我觉得这句翻译成“模板方法”会不会更好点？)。开发者可以通过重写 `[[yii\db\ActiveRecord::save()|save()]]`方法然后在控制器里使用事务操作，严格地说是似乎不是一个好的做法（召回“瘦控制器 / 肥模型”基本规则）。

这些方法在这里(如果你不明白自己实际在干什么，请不要使用他们)，Models:

```
class Feature extends \yii\db\ActiveRecord
{
    // ...

    public function getProduct()
    {
        return $this->hasOne(Product::className(), ['id' => 'product_id']);
    }
}
```

```
class Product extends \yii\db\ActiveRecord
{
    // ...

    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['product_id' => 'id']);
    }
}
```

重写 [[yii\db\ActiveRecord::save()|save()]] 方法:

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

(译注: 我觉得上面应该是原手册里的 **bug**)

在控制器层使用事务:

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

作为这些脆弱方法的替代, 你应该使用原子操作方案特性。

```
class Feature extends \yii\db\ActiveRecord
{
```

```

// ...

public function getProduct()
{
    return $this->hasOne(Product::className(), ['product_id' => 'id']);
}

public function scenarios()
{
    return [
        'userCreates' => [
            'attributes' => ['name', 'value'],
            'atomic' => [self::OP_INSERT],
        ],
    ];
}
}

class Product extends \yii\db\ActiveRecord
{
    // ...

    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['id' => 'product_id']);
    }
}

```



```

public function scenarios()
{
    return [

        'userCreates' => [

            'attributes' => ['title', 'price'],

            'atomic' => [self::OP_INSERT],

        ],

    ];
}

public function afterValidate()
{
    parent::afterValidate();

    // FIXME: TODO: WIP, TBD
}

public function afterSave($insert)
{
    parent::afterSave($insert);

    if ($this->getScenario() === 'userCreates') {

        // FIXME: TODO: WIP, TBD

    }

}
}

```

Controller 里的代码将变得很简洁:

```

class ProductController extends \yii\web\Controller
{

```

```
public function actionCreate()

{

    // FIXME: TODO: WIP, TBD

}

}
```

控制器非常简洁:

```
class ProductController extends \yii\web\Controller

{

    public function actionCreate()

    {

        // FIXME: TODO: WIP, TBD

    }

}
```

乐观锁（**Optimistic Locks**）

TODO

被污染属性

TODO

另见

- [模型](#) ([Model](#))
- [\[\[yii\db\ActiveRecord\]\]](#)

数据库迁移 **Database Migration**

Note: This section is under development.

Like source code, the structure of a database evolves as a database-driven application is

developed and maintained. For example, during development, a new table may be added; Or, after the application goes live, it may be discovered that an additional index is required. It is important to keep track of these structural database changes (called **migration**), just as changes to the source code is tracked using version control. If the source code and the database become out of sync, bugs will occur, or the whole application might break. For this reason, Yii provides a database migration tool that can keep track of database migration history, apply new migrations, or revert existing ones.

The following steps show how database migration is used by a team during development:

1. Tim creates a new migration (e.g. creates a new table, changes a column definition, etc.).
2. Tim commits the new migration into the source control system (e.g. Git, Mercurial).
3. Doug updates his repository from the source control system and receives the new migration.
4. Doug applies the migration to his local development database, thereby syncing his database to reflect the changes Tim made.

Yii supports database migration via the `yii migrate` command line tool. This tool supports:

- Creating new migrations
- Applying, reverting, and redoing migrations
- Showing migration history and new migrations

Creating Migrations

To create a new migration, run the following command:

```
yii migrate/create <name>
```

The required `name` parameter specifies a very brief description of the migration. For example, if the migration creates a new table named news, you'd use the command:

```
yii migrate/create create_news_table
```

As you'll shortly see, the `name` parameter is used as part of a PHP class name in the migration. Therefore, it should only contain letters, digits and/or underscore characters.

The above command will create a new file named `m101129_185401_create_news_table.php`. This file will be created within the `@app/migrations` directory. Initially, the migration file will be generated with the following code:

```
class m101129_185401_create_news_table extends \yii\db\Migration
{
    public function up()
    {
```

```

    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }
}

```

Notice that the class name is the same as the file name, and follows the pattern `m<timestamp>_<name>`, where:

- `<timestamp>` refers to the UTC timestamp (in the format of `yymmdd_hhmmss`) when the migration is created,
- `<name>` is taken from the command's `name` parameter.

In the class, the `up()` method should contain the code implementing the actual database migration. In other words, the `up()` method executes code that actually changes the database. The `down()` method may contain code that reverts the changes made by `up()`.

Sometimes, it is impossible for the `down()` to undo the database migration. For example, if the migration deletes table rows or an entire table, that data cannot be recovered in the `down()` method. In such cases, the migration is called irreversible, meaning the database cannot be rolled back to a previous state. When a migration is irreversible, as in the above generated code, the `down()` method returns `false` to indicate that the migration cannot be reverted.

As an example, let's show the migration about creating a news table.

```

use yii\db\Schema;

class m101129_185401_create_news_table extends \yii\db\Migration
{
    public function up()
    {
        $this->createTable('news', [

            'id' => 'pk',

```

```

        'title' => Schema::TYPE_STRING . ' NOT NULL',

        'content' => Schema::TYPE_TEXT,

    ]);

}

public function down()

{

    $this->dropTable('news');

}

}

```

The base class `[[yii\db\Migration]]` exposes a database connection via `db` property. You can use it for manipulating data and schema of a database.

The column types used in this example are abstract types that will be replaced by Yii with the corresponding types depended on your database management system. You can use them to write database independent migrations. For example `pk` will be replaced by `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY` for MySQL and `integer PRIMARY KEY AUTOINCREMENT NOT NULL` for sqlite. See documentation of `[[yii\db\QueryBuilder::getColumnType()]]` for more details and a list of available types. You may also use the constants defined in `[[yii\db\Schema]]` to define column types.

Note: You can add constraints and other custom table options at the end of the table description by specifying them as simple string. For example in the above migration, after `content` attribute definition you can write `'CONSTRAINT ...'` or other custom options.

Transactional Migrations

While performing complex DB migrations, we usually want to make sure that each migration succeed or fail as a whole so that the database maintains the consistency and integrity. In order to achieve this goal, we can exploit DB transactions. We could use special methods `safeUp` and `safeDown` for these purposes.

```

use yii\db\Schema;

```

```

class m101129_185401_create_news_table extends \yii\db\Migration

```

```

{

    public function safeUp()
    {
        $this->createTable('news', [

            'id' => 'pk',

            'title' => Schema::TYPE_STRING . ' NOT NULL',

            'content' => Schema::TYPE_TEXT,

        ]);

        $this->createTable('user', [

            'id' => 'pk',

            'login' => Schema::TYPE_STRING . ' NOT NULL',

            'password' => Schema::TYPE_STRING . ' NOT NULL',

        ]);

    }

    public function safeDown()
    {
        $this->dropTable('news');

        $this->dropTable('user');

    }

}

```

When your code uses more than one query it is recommended to use `safeUp` and `safeDown`.

Note: Not all DBMS support transactions. And some DB queries cannot be put into a transaction. In this case, you will have to implement `up()` and `down()`, instead. And for MySQL, some SQL statements may cause `implicit commit`.

Applying Migrations

To apply all available new migrations (i.e., make the local database up-to-date), run the following command:

```
yii migrate
```

The command will show the list of all new migrations. If you confirm to apply the migrations, it will run the `up()` method in every new migration class, one after another, in the order of the timestamp value in the class name.

After applying a migration, the migration tool will keep a record in a database table named `migration`. This allows the tool to identify which migrations have been applied and which are not. If the `migration` table does not exist, the tool will automatically create it in the database specified by the `db` application component.

Sometimes, we may only want to apply one or a few new migrations. We can use the following command:

```
yii migrate/up 3
```

This command will apply the 3 new migrations. Changing the value 3 will allow us to change the number of migrations to be applied.

We can also migrate the database to a specific version with the following command:

```
yii migrate/to 101129_185401
```

That is, we use the timestamp part of a migration name to specify the version that we want to migrate the database to. If there are multiple migrations between the last applied migration and the specified migration, all these migrations will be applied. If the specified migration has been applied before, then all migrations applied after it will be reverted (to be described in the next section).

Reverting Migrations

To revert the last one or several applied migrations, we can use the following command:

```
yii migrate/down [step]
```

where the optional `step` parameter specifies how many migrations to be reverted back. It defaults to 1, meaning reverting back the last applied migration.

As we described before, not all migrations can be reverted. Trying to revert such migrations will throw an exception and stop the whole reverting process.

Redoing Migrations

Redoing migrations means first reverting and then applying the specified migrations. This can be done with the following command:

```
yii migrate/redo [step]
```

where the optional **step** parameter specifies how many migrations to be redone. It defaults to 1, meaning redoing the last migration.

Showing Migration Information

Besides applying and reverting migrations, the migration tool can also display the migration history and the new migrations to be applied.

```
yii migrate/history [limit]
```

```
yii migrate/new [limit]
```

where the optional parameter **limit** specifies the number of migrations to be displayed. If **limit** is not specified, all available migrations will be displayed.

The first command shows the migrations that have been applied, while the second command shows the migrations that have not been applied.

Modifying Migration History

Sometimes, we may want to modify the migration history to a specific migration version without actually applying or reverting the relevant migrations. This often happens when developing a new migration. We can use the following command to achieve this goal.

```
yii migrate/mark 101129_185401
```

This command is very similar to **yii migrate/to** command, except that it only modifies the migration history table to the specified version without applying or reverting the migrations.

Customizing Migration Command

There are several ways to customize the migration command.

Use Command Line Options

The migration command comes with four options that can be specified in command line:

- **interactive**: boolean, specifies whether to perform migrations in an interactive mode. Defaults to true, meaning the user will be prompted when performing a specific migration. You may set this to false should the migrations be done in a background process.
- **migrationPath**: string, specifies the directory storing all migration class files. This must be specified in terms of a path alias, and the corresponding directory must exist. If not specified, it will use the **migrations** sub-directory under the application base path.
- **migrationTable**: string, specifies the name of the database table for storing

migration history information. It defaults to `migration`. The table structure is `version varchar(255) primary key, apply_time integer`.

- `db`: string, specifies the ID of the database application component. Defaults to 'db'.
- `templateFile`: string, specifies the path of the file to be served as the code template for generating the migration classes. This must be specified in terms of a path alias (e.g. `application.migrations.template`). If not set, an internal template will be used. Inside the template, the token `{ClassName}` will be replaced with the actual migration class name.

To specify these options, execute the migrate command using the following format

```
yii migrate/up --option1=value1 --option2=value2 ...
```

For example, if we want to migrate for a `forum` module whose migration files are located within the module's `migrations` directory, we can use the following command:

```
yii migrate/up --migrationPath=@app/modules/forum/migrations
```

Configure Command Globally

While command line options allow us to configure the migration command on-the-fly, sometimes we may want to configure the command once for all. For example, we may want to use a different table to store the migration history, or we may want to use a customized migration template. We can do so by modifying the console application's configuration file like the following,

```
'controllerMap' => [  
    'migrate' => [  
        'class' => 'yii\console\controllers\MigrateController',  
        'migrationTable' => 'my_custom_migrate_table',  
    ],  
]
```

Now if we run the `migrate` command, the above configurations will take effect without requiring us to enter the command line options every time. Other command options can be also configured this way.

Migrating with Multiple Databases

By default, migrations will be applied to the database specified by the `db` application component. You may change it by specifying the `--db` option, for example,

```
yii migrate --db=db2
```

The above command will apply all migrations found in the default migration path to the `db2` database.

If your application works with multiple databases, it is possible that some migrations should be applied to one database while some others should be applied to another database. In this case, it is recommended that you create a base migration class for each different database and override the `[[yii\db\Migration::init()]]` method like the following,

```
public function init()
{
    $this->db = 'db2';

    parent::init();
}
```

To create a migration that should be applied to a particular database, simply extend from the corresponding base migration class. Now if you run the `yii migrate` command, each migration will be applied to its corresponding database.

Info: Because each migration uses hardcoded DB connection, the `--db` option of the `migrate` command will have no effect. Also note that the migration history will be stored in the default `db` database.

If you want to support changing DB connection via the `--db` option, you may take the following alternative approach to work with multiple databases.

For each database, create a migration path and save all corresponding migration classes there. To apply migrations, run the command as follows,

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

Info: The above approach stores the migration history in different databases specified via the `--db` option.

Sphinx:

<http://www.yiichina.com/guide/2/db-sphinx>

Redis:

<http://www.yiichina.com/guide/2/db-redis>

MongoDB:

<http://www.yiichina.com/guide/2/db-mongodb>

ElasticSearch:

<http://www.yiichina.com/guide/2/db-elastic-search>

以上部分因为官网英文地址 **404** 错误，
故无法翻译入档

接受用户数据：

创建表单 **Working with Forms**

Note: This section is under development.

The primary way of using forms in Yii is through `[[yii\widgets\ActiveForm]]`. This approach should be preferred when the form is based upon a model. Additionally, there are some useful methods in `[[yii\helpers\Html]]` that are typically used for adding buttons and help text to any form.

When creating model-based forms, the first step is to define the model itself. The model can be either based upon the Active Record class, or the more generic Model class. For this login example, a generic model will be used:

```
use yii\base\Model;
```

```
class LoginForm extends Model
```

```
{
```

```
    public $username;
```

```
    public $password;
```

```
    /**
```

```
     * @return array the validation rules.
```

```
     */
```

```
    public function rules()
```

```
    {
```

```
        return [
```

```

        // username and password are both required

        [['username', 'password'], 'required'],

        // password is validated by validatePassword()

        ['password', 'validatePassword'],

    ];
}

/**
 * Validates the password.
 * This method serves as the inline validation for password.
 */
public function validatePassword()
{
    $user = User::findByUsername($this->username);

    if (!$user || !$user->validatePassword($this->password)) {
        $this->addError('password', 'Incorrect username or password.');
```

```

    }
}

/**
 * Logs in a user using the provided username and password.
 * @return boolean whether the user is logged in successfully
 */
public function login()
{
    if ($this->validate()) {
```

```

        $user = User::findByUsername($this->username);

        return true;

    } else {

        return false;

    }

}

}

```

The controller will pass an instance of that model to the view, wherein the `[[yii\widgets\ActiveForm|ActiveForm]]` widget is used:

```

use yii\helpers\Html;

use yii\widgets\ActiveForm;

<?php $form = ActiveForm::begin([

    'id' => 'login-form',

    'options' => ['class' => 'form-horizontal'],

]); ?>

<?= $form->field($model, 'username') ?>

<?= $form->field($model, 'password')->passwordInput() ?>

<div class="form-group">

    <div class="col-lg-offset-1 col-lg-11">

        <?= Html::submitButton('Login', ['class' => 'btn btn-primary']) ?>

    </div>

</div>

<?php ActiveForm::end() ?>

```

In the above code, `[[yii\widgets\ActiveForm::begin()|ActiveForm::begin()]]` not only creates a form instance, but also marks the beginning of the form. All of the content placed between `[[yii\widgets\ActiveForm::begin()|ActiveForm::begin()]]` and `[[yii\widgets\ActiveForm::end()|ActiveForm::end()]]` will be wrapped within the `<form>` tag. As with any widget, you can specify some options as to how the widget should be

configured by passing an array to the `begin` method. In this case, an extra CSS class and identifying ID are passed to be used in the opening `<form>` tag.

In order to create a form element in the form, along with the element's label, and any application JavaScript validation, the `[[yii\widgets\ActiveForm::field()|ActiveForm::field()]]` method of the Active Form widget is called. When the invocation of this method is echoed directly, the result is a regular (text) input. To customize the output, you can chain additional methods to this call:

```
<?= $form->field($model, 'password')->passwordInput() ?>

// or

<?= $form->field($model, 'username')->textInput()->hint('Please enter your
name')->label('Name') ?>
```

This will create all the `<label>`, `<input>` and other tags according to the template defined by the form field. To add these tags yourself you can use the `Html` helper class.

If you want to use one of HTML5 fields you may specify input type directly like the following:

```
<?= $form->field($model, 'email')->input('email') ?>
```

Specifying the attribute of the model can be done in more sophisticated ways. For example when an attribute may take an array value when uploading multiple files or selecting multiple items you may specify it by appending `[]` to the attribute name:

```
// allow multiple files to be uploaded:

echo $form->field($model, 'uploadFile[]')->fileInput(['multiple'=>'multiple']);

// allow multiple items to be checked:

echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' => 'Item B', 'c' => 'Item C']);
```

Tip: in order to style required fields with asterisk you can use the following CSS:

```
div.required label:after {

    content: " *";

    color: red;

}
```

Handling multiple models with a single form

Sometimes you need to handle multiple models of the same kind in a single form. For example, multiple settings where each setting is stored as name-value and is represented by `Setting` model. The following shows how to implement it with Yii.

Let's start with controller action:

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use app\models\Setting;

class SettingsController extends Controller
{
    // ...

    public function actionUpdate()
    {
        $settings = Setting::find()->indexBy('id')->all();

        if (Model::loadMultiple($settings, Yii::$app->request->post()) &&
            Model::validateMultiple($settings)) {

            foreach ($settings as $setting) {

                $setting->save(false);

            }

            return $this->redirect('index');
        }
    }
}
```

```

        return $this->render('update', ['settings' => $settings]);
    }
}

```

In the code above we're using `indexBy` when retrieving models from database to make array indexed by model ids. These will be later used to identify form fields. `loadMultiple` fills multiple models with the form data coming from POST and `validateMultiple` validates all models at once. In order to skip validation when saving we're passing `false` as a parameter to `save`.

Now the form that's in `update` view:

```

<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo Html::encode($setting->name) . ' : ' . $form->field($setting, "[$index]value");
}

ActiveForm::end();

```

Here for each setting we are rendering name and an input with a value. It is important to add a proper index to input name since that is how `loadMultiple` determines which model to fill with which values.

输入验证

一般说来，程序猿永远不应该信任从最终用户直接接收到的数据，并且使用它们之前应始终先验证其可靠性。

要给 `model` 填充其所需的用户输入数据，你可以调用 `[[yii\base\Model::validate()]]` 方法验证它们。该方法会返回一个布尔值，指明是否通过验证。若没有通过，你能通过 `[[yii\base\Model::errors]]` 属性获取相应的报错信息。比如，

```

$model = new \app\models\ContactForm;

```



```
// 用用户输入来填充模型的特性

$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {

    // 若所有输入都是有效的

} else {

    // 有效性验证失败: $errors 属性就是存储错误信息的数组

    $errors = $model->errors;

}
```

validate() 方法，在幕后为执行验证操作，进行了以下步骤：

1. 通过从 `[[yii\base\Model::scenarios()]]` 方法返回基于当前 `[[yii\base\Model::scenario|场景 (scenario)]]` 的特性属性列表，算出哪些特性应该进行有效性验证。这些属性被称作 **active attributes**（激活特性）。
2. 通过从 `[[yii\base\Model::rules()]]` 方法返回基于当前 `[[yii\base\Model::scenario|场景 (scenario)]]` 的验证规则列表，这些规则被称作 **active rules**（激活规则）。
3. 用每个激活规则去验证每个与之关联的激活特性。若失败，则记录下对应模型特性的错误信息。

声明规则（Rules）

要让 **validate()** 方法起作用，你需要声明与需验证模型特性相关的验证规则。为此，需要重写 `[[yii\base\Model::rules()]]` 方法。下面的例子展示了如何声明用于验证 **ContactForm** 模型的相关验证规则：

```
public function rules()

{

    return [

        // name, email, subject 和 body 特性是 `require`（必填）的

        [['name', 'email', 'subject', 'body'], 'required'],

        // email 特性必须是一个有效的 email 地址

        ['email', 'email'],

    ];

}
```

```
}
```

`[[yii\base\Model::rules()]rules()]` 方法应返回一个由规则所组成的数组，每一个规则都呈现为以下这类格式的小数组：

```
[  
    // 必须项，用于指定那些模型特性需要通过此规则的验证。  
    // 对于只有一个特性的情况，可以直接写特性名，而不必用数组包裹。  
    ['attribute1', 'attribute2', ...],  
  
    // 必填项，用于指定规则的类型。  
    // 它可以是类名，验证器昵称，或者是验证方法的名称。  
    'validator',  
  
    // 可选项，用于指定在场景（scenario）中，需要启用该规则  
    // 若不提供，则代表该规则适用于所有场景  
    // 若你需要提供除了某些特定场景以外的所有其他场景，你也可以配置 "except" 选项  
    'on' => ['scenario1', 'scenario2', ...],  
  
    // 可选项，用于指定对该验证器对象的其他配置选项  
    'property1' => 'value1', 'property2' => 'value2', ...  
]
```

对于每个规则，你至少需要指定该规则适用于哪些特性，以及本规则的类型是什么。你可以指定以下的规则类型之一：

- 核心验证器的昵称，比如 `required`、`in`、`date`，等等。请参考[核心验证器](#)章节查看完整的核心验证器列表。
- 模型类中的某个验证方法的名称，或者一个匿名方法。请参考[行内验证器](#)小节了解更多。
- 验证器类的名称。请参考[独立验证器](#)小节了解更多。

一个规则可用于验证一个或多个模型特性，且一个特性可以被一个或多个规则所验证。一个规则可以施用于特定[场景（scenario）](#)，只要指定 `on` 选项。如果你不指定 `on` 选项，那么该规则会适配于所有场景。

当调用 `validate()` 方法时，它将运行以下几个具体的验证步骤：

1. 检查从声明自 `[[yii\base\Model::scenarios()]]` 方法的场景中所挑选出的当前

[[yii\base\Model::scenario|场景]]的信息，从而确定出那些特性需要被验证。这些特性被称为激活特性。

2.检查从声明自 [[yii\base\Model::rules()]] 方法的众多规则中所挑选出的适用于当前 [[yii\base\Model::scenario|场景]]的规则，从而确定出需要验证哪些规则。这些规则被称为激活规则。

3.用每个激活规则去验证每个与之关联的激活特性。

基于以上验证步骤，有且仅有声明在 `scenarios()` 方法里的激活特性，且它还必须与一或多个声明自 `rules()` 里的激活规则相关联才会被验证。

自定义错误信息

大多数的验证器都有默认的错误信息，当模型的某个特性验证失败的时候，该错误信息会被返回给模型。比如，用 [[yii\validators\RequiredValidator|required]] 验证器的规则检验 `username` 特性失败的话，会返还给模型 "Username cannot be blank." 信息。

你可以通过在声明规则的时候同时指定 `message` 属性，来定制某个规则的错误信息，比如这样：

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Please choose a username.'],
    ];
}
```

一些验证器还支持用于针对不同原因的验证失败返回更加准确的额外错误信息。比如，[[yii\validators\NumberValidator|number]] 验证器就支持 [[yii\validators\NumberValidator::tooBig|tooBig]] 和 [[yii\validators\NumberValidator::tooSmall|tooSmall]] 两种错误消息用于分别返回输入值是太大还是太小。你也可以像配置验证器的其他属性一样配置它们俩各自的错误信息。

验证事件

当调用 [[yii\base\Model::validate()]] 方法的过程里，它同时会调用两个特殊的方法，把它们重写掉可以实现自定义验证过程的目的：

- [[yii\base\Model::beforeValidate()]]：在默认的实现中会触发 [[yii\base\Model::EVENT_BEFORE_VALIDATE]] 事件。你可以重写该方法或者响应此事件，来在验证开始之前，先进行一些预处理的工作。（比如，标准化数据输入）该方法应该返回一个布尔值，用于标明验证是否通过。
- [[yii\base\Model::afterValidate()]]：在默认的实现中会触发 [[yii\base\Model::EVENT_AFTER_VALIDATE]] 事件。你可以重写该方法或者响应此事件，来在验证结束之后，再进行一些收尾的工作。

条件式验证

若要只在某些条件满足时，才验证相关特性，比如：是否验证某特性取决于另一特性的值，你可以通过 `[[yii\validators\Validator::when|when]]` 属性来定义相关条件。举例而言，

```
[
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }],
]
```

`[[yii\validators\Validator::when|when]]` 属性会读入一个如下所示结构的 PHP callable 函数对象：

```
/**
 * @param Model $model 要验证的模型对象
 * @param string $attribute 待测特性名
 * @return boolean 返回是否启用该规则
 */
function ($model, $attribute)
```

若你需要支持客户端的条件验证，你应该配置 `[[yii\validators\Validator::whenClient|whenClient]]` 属性，它会读入一条包含有 JavaScript 函数的字符串。这个函数将被用于确定该客户端验证规则是否被启用。比如，

```
[
    ['state', 'required', 'when' => function ($model) {
        return $model->country == 'USA';
    }, 'whenClient' => "function (attribute, value) {
        return $('#country').value == 'USA';
    }"],
]
```

数据预处理

用户输入经常需要进行数据过滤，或者叫预处理。比如你可能会需要先去掉 `username` 输入的收尾空格。你可以通过使用验证规则来实现此目的。

下面的例子展示了如何去掉输入信息的首尾空格，并将空输入返回为 `null`。具体方法为通过调用 `trim`

和 `default` 核心验证器:

```
[
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
]
```

也还可以用更加通用的 `filter` (滤镜) 核心验证器来执行更加复杂的数据过滤。

如你所见，这些验证规则并不真的对输入数据进行任何验证。而是，对输入数据进行一些处理，然后把它们存回当前被验证的模型特性。

处理空输入

当输入数据是通过 **HTML** 表单，你经常会需要给空的输入项赋默认值。你可以通过调整 `default` 验证器来实现这一点。举例来说，

```
[
    // 若 "username" 和 "email" 为空，则设为 null
    [['username', 'email'], 'default'],

    // 若 "level" 为空，则设其为 1
    ['level', 'default', 'value' => 1],
]
```

默认情况下，当输入项为空字符串，空数组，或 `null` 时，会被视为“空值”。你也可以通过配置 `[[yii\validators\Validator::isEmpty]]` 属性来自定义空值的判定规则。比如，

```
[
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }],
]
```

注意：对于绝大多数验证器而言，若其

`[[yii\base\Validator::skipOnEmpty]]` 属性为默认值 `true`，则它们不会对空值进行任何处理。也就是当他们的关联特性接收到空值时，相关验证会被直接略过。在 **核心验证器** 之中，只有 `captcha` (验证码)，`default` (默认值)，`filter` (滤镜)，`required` (必填)，以及 `trim` (去首尾空

格)，这几个验证器会处理空输入。

临时验证

有时，你需要对某些没有绑定任何模型类的值进行 **临时验证**。

若你只需要进行一种类型的验证（e.g. 验证邮箱地址），你可以调用所需验证器的 `[[yii\validators\Validator::validate()|validate()]]` 方法。像这样：

```
$email = 'test@example.com';

$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {

    echo '有效的 Email 地址。';

} else {

    echo $error;

}
```

注意：不是所有的验证器都支持这种形式的验证。比如 **unique（唯一性）**

核心验证器就就是一个例子，它的设计初衷就是只作用于模型类内部的。

若你需要针对一系列值执行多项验证，你可以使用 `[[yii\base\DynamicModel]]`。它支持即时添加特性和验证规则的定义。它的使用规则是这样的：

```
public function actionSearch($name, $email)

{

    $model = DynamicModel::validateData(compact('name', 'email'), [

        [['name', 'email'], 'string', 'max' => 128],

        ['email', 'email'],

    ]);

    if ($model->hasErrors()) {

        // 验证失败

    } else {

        // 验证成功

    }

}
```

```
}
```

`[[yii\base\DynamicModel::validateData()]]` 方法会创建一个 `DynamicModel` 的实例对象，并通过给定数据定义模型特性（以 `name` 和 `email` 为例），之后用给定规则调用 `[[yii\base\Model::validate()]]` 方法。

除此之外呢，你也可以用如下的更加“传统”的语法来执行临时数据验证：

```
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));

    $model->addRule(['name', 'email'], 'string', ['max' => 128])

        ->addRule('email', 'email')

        ->validate();

    if ($model->hasErrors()) {

        // 验证失败

    } else {

        // 验证成功

    }
}
```

验证之后你可以通过调用 `[[yii\base\DynamicModel::hasErrors()|hasErrors()]]` 方法来检查验证通过与否，并通过 `[[yii\base\DynamicModel::errors|errors]]` 属性获得验证的错误信息，过程与普通模型类一致。你也可以访问模型对象内定义的动态特性，就像：`$model->name` 和 `$model->email`。

创建验证器（Validators）

除了使用 Yii 的发布版里所包含的[核心验证器](#)之外，你也可以创建你自己的验证器。自定义的验证器可以是行内验证器，也可以是独立验证器。

行内验证器（Inline Validators）

行内验证器是一种以模型方法或匿名函数的形式定义的验证器。这些方法/函数的结构如下：

```
/**
 * @param string $attribute 当前被验证的特性
```

```
* @param array $params 以名-值对形式提供的额外参数
*/
```

```
function ($attribute, $params)
```

若某特性的验证失败了，该方法/函数应该调用 `[[yii\base\Model::addError()]]` 保存错误信息到模型内。这样这些错误就能在之后的操作中，被读取并展现给终端用户。

下面是一些例子：

```
use yii\base\Model;

class MyForm extends Model
{
    public $country;

    public $token;

    public function rules()
    {
        return [

            // 以模型方法 validateCountry() 形式定义的行内验证器

            ['country', 'validateCountry'],

            // 以匿名函数形式定义的行内验证器

            ['token', function ($attribute, $params) {

                if (!ctype_alnum($this->$attribute)) {

                    $this->addError($attribute, '令牌本身必须包含字母或数字。');

                }

            }],

        ];
    }
}
```



```

public function validateCountry($attribute, $params)
{
    if (!in_array($this->$attribute, ['天朝', '墙外'])) {
        $this->addError($attribute, '国家必须为 "天朝" 或 "墙外" 中的一个。');
    }
}
}

```

注意：缺省状态下，行内验证器不会在关联特性的输入值为空或该特性已经在其他验证中失败的情况下起效。若你想要确保该验证器始终启用的话，你可以在定义规则时，酌情将

[[yii\validators\Validator::skipOnEmpty|skipOnEmpty]] 以及 [[yii\validators\Validator::skipOnError|skipOnError]] 属性设为 false，比如，

```

[
    ['country', 'validateCountry', 'skipOnEmpty' => false, 'skipOnError' => false],
]

```

独立验证器（Standalone Validators）

独立验证器是继承自 [[yii\validators\Validator]] 或其子类的类。你可以通过重写 [[yii\validators\Validator::validateAttribute()]] 来实现它的验证规则。若特性验证失败，可以调用 [[yii\base\Model::addError()]] 以保存错误信息到模型内，操作与 [inline validators](#) 所需操作完全一样。比如，

```

namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['天朝', '墙外'])) {
            $this->addError($attribute, '国家必须为 "天朝" 或 "墙外" 中的一个。');
        }
    }
}

```

```
    }  
  
    }  
}
```

若你想要验证器支持不使用 `model` 的数据验证，你还应该重写 `[[yii\validators\Validator::validate()]]` 方法。你也可以通过重写 `[[yii\validators\Validator::validateValue()]]` 方法替代 `validateAttribute()` 和 `validate()`，因为默认状态下，后两者的实现使用过调用 `validateValue()` 实现的。

客户端验证器（Client-Side Validation）

当终端用户通过 HTML 表单提供相关输入信息时，我们可能会需要用到基于 JavaScript 的客户端验证。因为，它可以让用户更快速的得到错误信息，也因此可以提供更好的用户体验。你可以使用或自己实现除服务器端验证之外，**还能额外**客户端验证功能的验证器。

补充：尽管客户端验证为加分项，但它不是必须项。它存在的主要意义在于给用户提供更好的客户体验。正如“永远不要相信来自终端用户的输入信息”，也同样永远不要相信客户端验证。基于这个理由，你应该始终如前文所描述的那样，通过调用 `[[yii\base\Model::validate()]]` 方法执行服务器端验证。

使用客户端验证

许多**核心验证器**都支持开箱即用的客户端验证。你只需要用 `[[yii\widgets\ActiveForm]]` 的方式构建 HTML 表单即可。比如，下面的 `LoginForm`（登录表单）声明了两个规则：其一为 `required` 核心验证器，它同时支持客户端与服务器端的验证；另一个则采用 `validatePassword` 行内验证器，它只支持服务器端。

```
namespace app\models;  
  
use yii\base\Model;  
use app\models\User;  
  
class LoginForm extends Model  
{  
  
    public $username;  
  
    public $password;  
  
    public function rules()
```

```

{

    return [

        // username 和 password 都是必填项

        [['username', 'password'], 'required'],

        // 用 validatePassword() 验证 password

        ['password', 'validatePassword'],

    ];

}

public function validatePassword()

{

    $user = User::findByUsername($this->username);

    if (!$user || !$user->validatePassword($this->password)) {

        $this->addError('password', 'Incorrect username or password.');
```

使用如下代码构建的 HTML 表单包含两个输入框 `username` 以及 `password`。如果你在没有输入任何东西之前提交表单，就会在没有任何与服务器端的通讯的情况下，立刻收到一个要求你填写空白项的错误信息。

```

<?php $form = yii\widgets\ActiveForm::begin(); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <?= Html::submitButton('Login') ?>

<?php yii\widgets\ActiveForm::end(); ?>
```

幕后的运作过程是这样的：[[yii\widgets\ActiveForm]] 会读取声明在模型类中的验证规则，并生成那些支持支持客户端验证的验证器所需的 JavaScript 代码。当用户修改输入框的值，或者提交表单时，就

会触发相应的客户端验证 JS 代码。

若你需要完全关闭客户端验证，你只需配置 `[[yii\widgets\ActiveForm::enableClientValidation]]` 属性为 `false`。你同样可以关闭各个输入框各自的客户端验证，只要把它们的 `[[yii\widgets\ActiveField::enableClientValidation]]` 属性设为 `false`。

自己实现客户端验证

要穿件一个支持客户端验证的验证器，你需要实现 `[[yii\validators\Validator::clientValidateAttribute()]]` 方法，用于返回一段用于运行客户端验证的 JavaScript 代码。在这段 JavaScript 代码中，你可以使用以下预定义的变量：

- `attribute`：正在被验证的模型特性的名称。
- `value`：进行验证的值。
- `messages`：一个用于暂存模型特性的报错信息的数组。

在下面的例子里，我们会创建一个 `StatusValidator`，它会通过比对现有的状态数据，验证输入值是否为一个有效的状态。该验证器同时支持客户端以及服务器端验证。

```
namespace app\components;

use yii\validators\Validator;
use app\models>Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();

        $this->message = '无效的状态输入。';
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;

        if (!Status::find()->where(['id' => $value])->exists()) {
```

```

        $model->addError($attribute, $this->message);

    }

}

public function clientValidateAttribute($model, $attribute, $view)
{
    $statuses = json_encode(Status::find()->select('id')->asArray()->column());

    $message = json_encode($this->message);

    return <<<JS
if (!$.inArray(value, $statuses)) {
    messages.push($message);
}
JS;
    }
}

```

技巧：上述代码主要是演示了如何支持客户端验证。在具体实践中，你可以使用 `in` 核心验证器来达到同样的目的。比如这样的验证规则：

```

[
    ['status', 'in', 'range' => Status::find()->select('id')->asArray()->column()],
]

```

文件上传 **Uploading Files**

Note: This section is under development.

Uploading files in Yii is done via form model, its validation rules and some controller code. Let's review what's needed to handle uploads properly.

Form model

First of all, you need to create a model that will handle file upload.

Create `models/UploadForm.php` with the following content:

```

namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

/**
 * UploadForm is the model behind the upload form.
 */
class UploadForm extends Model
{
    /**
     * @var UploadedFile|null file attribute
     */
    public $file;

    /**
     * @return array the validation rules.
     */
    public function rules()
    {
        return [
            [['file'], 'file'],
        ];
    }
}

```

In the code above, we created a model `UploadForm` with an attribute `$file` that will become `<input type="file">` in the HTML form. The attribute has the validation rule named `file` that uses `[[yii\validators\FileValidator|FileValidator]]`.

Form view

Next create a view that will render the form.

```
<?php
use yii\widgets\ActiveForm;

$form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data']]); ?>

<?= $form->field($model, 'file')->fileInput() ?>

<button>Submit</button>

<?php ActiveForm::end(); ?>
```

The `'enctype' => 'multipart/form-data'` is important since it allows file uploads. `fileInput()` represents a form input field.

Controller

Now create the controller that connects form and model together:

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();
```

```

if (Yii::$app->request->isPost) {

    $model->file = UploadedFile::getInstance($model, 'file');

    if ($model->validate()) {

        $model->file->saveAs('uploads/' . $model->file->baseName . '.' .
$model->file->extension);

    }

}

return $this->render('upload', ['model' => $model]);

}
}

```

Instead of `model->load(...)` we are using `UploadedFile::getInstance(...)`. `[[yii\web\UploadedFile|UploadedFile]]` does not run the model validation. It only provides information about the uploaded file. Therefore, you need to run validation manually via `$model->validate()`. This triggers the `[[yii\validators\FileValidator|FileValidator]]` that expects a file:

```
$file instanceof UploadedFile || $file->error == UPLOAD_ERR_NO_FILE //in code framework
```

If validation is successful, then we're saving the file:

```
$model->file->saveAs('uploads/' . $model->file->baseName . '.' . $model->file->extension);
```

If you're using "basic" application template then folder `uploads` should be created under `web`.

That's it. Load the page and try uploading. Uploads should end up in `basic/web/uploads`.

Additional information

Required rule

If you need to make file upload mandatory use `skipOnEmpty` like the following:

```
public function rules()
```

```
{
```

```
    return [
```



```
[[ 'file', 'file', 'skipOnEmpty' => false ],
];
}
```

MIME type

It is wise to validate type of the file uploaded. FileValidator has property `$extensions` for the purpose:

```
public function rules()
{
    return [
        [ 'file', 'file', 'extensions' => 'gif, jpg', ],
    ];
}
```

The thing is that it validates only file extension and not the file content. In order to validate content as well use `mimeType` property of `FileValidator`:

```
public function rules()
{
    return [
        [ 'file', 'file', 'extensions' => 'jpg, png', 'mimeType' => 'image/jpeg, image/png', ],
    ];
}
```

[List of common media types](#)

Validating uploaded image

If you upload an image, `[[yii\validators\ImageValidator|ImageValidator]]` may come in handy. It verifies if an attribute received a valid image that can be then either saved or processed using [Imagine Extension](#).

Uploading multiple files

If you need download multiple files at once some adjustments are required. View:

```
<?php
```

```

use yii\widgets\ActiveForm;

$form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data']]);

if ($model->hasErrors()) { //it is necessary to see all the errors for all the files.

    echo '<pre>';

    print_r($model->getErrors());

    echo '</pre>';
}

?>

<?= $form->field($model, 'file[]')->fileInput(['multiple' => '']) ?>

<button>Submit</button>

<?php ActiveForm::end(); ?>

```

The difference is the following line:

```

<?= $form->field($model, 'file[]')->fileInput(['multiple' => '']) ?>

```

Controller:

```

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{

    public function actionUpload()
    {

```

```

$model = new UploadForm();

if (Yii::$app->request->isPost) {

    $files = UploadedFile::getInstances($model, 'file');

    foreach ($files as $file) {

        $_model = new UploadForm();

        $_model->file = $file;

        if ($_model->validate()) {

            $_model->file->saveAs('uploads/' . $_model->file->baseName . '.' .
$model->file->extension);

        } else {

            foreach ($_model->getErrors('file') as $error) {

                $model->addError('file', $error);

            }

        }

    }

    if ($model->hasErrors('file')){

        $model->addError(

            'file',

            count($model->getErrors('file')) . ' of ' . count($files) . ' files not uploaded'

        );

    }
}

```

```
}

return $this->render('upload', ['model' => $model]);

}

}
```

The difference is `UploadedFile::getInstances($model, 'file');` instead of `UploadedFile::getInstance($model, 'file');`. Former returns instances for **all** uploaded files while the latter gives you only a single instance.

多模型同时输入:

<http://www.yiichina.com/guide/2/input-multiple-models>

显示数据:

格式化输出数据(数据格式器)

Yii 提供一个格式化类来格式化输出，以使输出数据对终端用户更友好易读，`[[yii\i18n\Formatter]]` 是一个助手类，作为 [应用组件](#) 使用，默认名为 `formatter`。

它提供一些方法用来格式化数据，如日期/时间、数字或其他常用的本地化格式，两种方式使用格式器：

1. 直接使用格式化方法(所有的格式器方法以 `as` 做前缀)：

```
echo Yii::$app->formatter->asDate('2014-01-01', 'long'); // 输出: January 1, 2014

echo Yii::$app->formatter->asPercent(0.125, 2); // 输出: 12.50%

echo Yii::$app->formatter->asEmail('cebe@example.com'); // 输出: <a
href="mailto:cebe@example.com">cebe@example.com</a>

echo Yii::$app->formatter->asBoolean(true); // 输出: Yes
```

```
// 也可处理 null 值的输出显示:
```

```
echo Yii::$app->formatter->asDate(null); // 输出: (Not set)
```

2.使用 `[[yii\i18n\Formatter::format()|format()]]` 方法和格式化名，该方法也被一些小部件如`[[yii\grid\GridView]]`和`[[yii\widgets\DetailView]]`使用，在小部件配置中可以指定列的数据格式。

```
echo Yii::$app->formatter->format('2014-01-01', 'date'); // 输出: January 1, 2014
```

```
// 可使用数组来指定格式化方法的参数:
```

```
// `2` 是 asPercent()方法的参数$decimals 的值
```

```
echo Yii::$app->formatter->format(0.125, ['percent', 2]); // 输出: 12.50%
```

当 [PHP intl extension](#) 安装时，格式器的输出会本地化，为此可配置格式器的 `[[yii\i18n\Formatter::locale|locale]]` 属性，如果没有配置，应用配置 `[[yii\base\Application::language|language]]` 作为当前区域，更多详情参考 [国际化](#) 一节。然后格式器根据当前区域为日期和数字选择正确的格式，包括月份和星期也会转换到当前语言，日期格式也会被 `[[yii\i18n\Formatter::timeZone|timeZone]]` 参数影响，该参数如果没有明确配置会使用应用的 `[[yii\base\Application::timeZone|from the application]]` 参数。

日期格式根据不同区域输出不同的结果，如下例所示： For example the date format call will output different results for different locales:

```
Yii::$app->formatter->locale = 'en-US';
```

```
echo Yii::$app->formatter->asDate('2014-01-01'); // 输出: January 1, 2014
```

```
Yii::$app->formatter->locale = 'de-DE';
```

```
echo Yii::$app->formatter->asDate('2014-01-01'); // 输出: 1. Januar 2014
```

```
Yii::$app->formatter->locale = 'ru-RU';
```

```
echo Yii::$app->formatter->asDate('2014-01-01'); // 输出: 1 января 2014 г.
```

注意不管 [PHP intl extension](#) 有没有安装，PHP 编译的 ICU 库不同，格式化结果可能不同，所以为确保不同环境下得到相同的输出，推荐在每个环境下安装 PHP intl 扩展以及相同的 ICU 库，可参考：[为国际化设置 PHP 环境](#)。

配置格式器

可配置 `[[yii\i18n\Formatter|formatter class]]` 的属性来调整格式器方法的默认格式，可以在[应用主体配置](#)中配置 `formatter` 组件应用到整个项目，配置样例如下所示，更多关于可用属性的详情请参考 `[[yii\i18n\Formatter|API documentation of the Formatter class]]` 和接下来一小节。

```
'components' => [
```

```
'formatter' => [  
  
    'dateFormat' => 'dd.MM.yyyy',  
  
    'decimalSeparator' => ',',  
  
    'thousandSeparator' => '',  
  
    'currencyCode' => 'EUR',  
  
],  
],
```

格式化日期和时间

格式器类为格式化日期和时间提供了多个方法： The formatter class provides different methods for formatting date and time values. These are:

- `[[yii\i18n\Formatter::asDate()|date]]` - 值被格式化成日期，如 **January, 01 2014**.
- `[[yii\i18n\Formatter::asTime()|time]]` - 值被格式化成时间，如 **14:23**.
- `[[yii\i18n\Formatter::asDatetime()|datetime]]` - 值被格式化成日期和时间，如 **January, 01 2014 14:23**.
- `[[yii\i18n\Formatter::asTimestamp()|timestamp]]` - 值被格式化成 **unix 时间戳** 如 **1412609982**.
- `[[yii\i18n\Formatter::asRelativeTime()|relativeTime]]` - 值被格式化成和当前时间比较的时间间隔并用人们易读的格式，如 **1 hour ago**.

可配置格式器的属性`[[yii\i18n\Formatter::$dateFormat|$dateFormat]]`, `[[yii\i18n\Formatter::$timeFormat|$timeFormat]]` 和 `[[yii\i18n\Formatter::$datetimeFormat|$datetimeFormat]]` 来全局指定`[[yii\i18n\Formatter::asDate()|date]]`, `[[yii\i18n\Formatter::asTime()|time]]` 和 `[[yii\i18n\Formatter::asDatetime()|datetime]]` 方法的日期和时间格式。

格式器默认会使用一个快捷格式，它根据当前启用的区域来解析， 这样日期和时间会格式化成用户国家和语言通用的格式， 有四种不同的快捷格式：

- en_GB** 区域的 **short** 会打印日期为 **06/10/2014**，时间为 **15:58**
- medium** 会分别打印 **6 Oct 2014** 和 **15:58:42**,
- long** 会分别打印 **6 October 2014** 和 **15:58:42 GMT**,
- full** 会分别打印 **Monday, 6 October 2014** 和 **15:58:42 GMT**.

另外你可使用 [ICU 项目](#) 定义的语法来自定义格式， ICU 项目在该 [URL](#)：

<http://userguide.icu-project.org/formatparse/datetime> 下的手册有介绍， 或者可使用 PHP `date()` 方法的语法字符串并加上前缀 **php:**.

```
// ICU 格式化
```

```
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06

// PHP date()-格式化

echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

时区

当格式化日期和时间时，Yii 会将它们转换为对应的 `[[yii\i18n\Formatter::timeZone|configured time zone]]` 时区，输入的值在没有指定时区时候会被当作 UTC 时间，因此，推荐存储所有的日期和时间为 UTC 而不是 UNIX 时间戳，UNIX 通常也是 UTC。如果输入值所在的时区不同于 UTC，时区应明确指定，如下所示：

```
// 假定 Yii::$app->timeZone = 'Europe/Berlin';

echo Yii::$app->formatter->asTime(1412599260); // 14:41:00

echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00

echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

注意：时区从属于全世界各国政府定的规则，可能会频繁的变更，因此你的系统的时区数据库可能不是最新的信息，可参考 [ICU manual](#) 关于更新时区数据库的详情，也可参考：[为国际化设置 PHP 环境](#)。

格式化数字

格式器类提供如下方法格式化数值：For formatting numeric values the formatter class provides the following methods:

- `[[yii\i18n\Formatter::asInteger()|integer]]` - 值被格式化成整型，如 **42**。
- `[[yii\i18n\Formatter::asDecimal()|decimal]]` - 值被格式化成十进制数字并带有小数位和千分位，如 **42.123**。
- `[[yii\i18n\Formatter::asPercent()|percent]]` - 值被格式化成百分率，如 **42%**。
- `[[yii\i18n\Formatter::asScientific()|scientific]]` - 值被格式化成科学计数型，如 **4.2E4**。
- `[[yii\i18n\Formatter::asCurrency()|currency]]` - 值被格式化成货币格式，如 **£420.00**。
- `[[yii\i18n\Formatter::asSize()|size]]` - 字节值被格式化成易读的值，如 **410 kibibytes**。

可配置 `[[yii\i18n\Formatter::decimalSeparator|decimalSeparator]]` 和 `[[yii\i18n\Formatter::thousandSeparator|thousandSeparator]]` 属性来调整数字格式化的格式，默认和当前区域相同。

更多高级配置，`[[yii\i18n\Formatter::numberFormatterOptions]]` 和 `[[yii\i18n\Formatter::numberFormatterTextOptions]]` 可用于配置内部使用 [Numberformatter class](#)

为调整数字的小数部分的最大值和最小值，可配置如下属性：

[

```
NumberFormatter::MIN_FRACTION_DIGITS => 0,  
  
NumberFormatter::MAX_FRACTION_DIGITS => 2,
```

其他格式器

除了日期、时间和数字格式化外，Yii 提供其他用途提供一些实用的格式器： Additional to date, time and number formatting, Yii provides a set of other useful formatters for different purposes:

- `[[yii\i18n\Formatter::asRaw()|raw]]` - 输出值和原始值一样，除了 `null` 值会用 `[[nullDisplay]]` 格式化，这是一个伪格式器；
- `[[yii\i18n\Formatter::asText()|text]]` - 值会经过 HTML 编码； 这是 `GridView DataColumn` 默认使用的格式；
- `[[yii\i18n\Formatter::asNtext()|ntext]]` - 值会格式化成 HTML 编码的纯文本，新行会转换成换行符；
- `[[yii\i18n\Formatter::asParagraphs()|paragraphs]]` - 值会转换成 HTML 编码的文本段落，用 `<p>` 标签包裹；
- `[[yii\i18n\Formatter::asHtml()|html]]` - 值会被 `[[HtmlPurifier]]` 过滤来避免 XSS 跨域攻击，可传递附加选项如 `['html', ['Attr.AllowedFrameTargets' => ['_blank']]]`；
- `[[yii\i18n\Formatter::asEmail()|email]]` - 值会格式化成 `mailto-` 链接；
- `[[yii\i18n\Formatter::asImage()|image]]` - 值会格式化成图片标签；
- `[[yii\i18n\Formatter::asUrl()|url]]` - 值会格式化成超链接；
- `[[yii\i18n\Formatter::asBoolean()|boolean]]` - 值会格式化成布尔型值，默认情况下 `true` 对应 `Yes`，`false` 对应 `No`，可根据应用语言配置进行翻译，可以配置 `[[yii\i18n\Formatter::booleanFormat]]`-属性来调整；

null-值

对于 PHP 的 `null` 值，格式器类会打印一个占位符而不是空字符串，空字符串默认会显示对应当前语言 (`not set`)，可配置 `[[yii\i18n\Formatter::nullDisplay|nullDisplay]]`-属性配置一个自定义占位符， 如果对处理 `null` 值没有特殊要求，可设置 `[[yii\i18n\Formatter::nullDisplay|nullDisplay]]` 为 `null`。

分页 (pagination)

<http://www.yiichina.com/guide/2/output-pagination>

下面小伙介绍一下 **ActiveDataProvider** 类分页的使用

use yii\data\ActiveDataProvider; 这是必须的步骤, 因为我们要使用 **ActiveDataProvider** 类

```
use common\models\User;
```

```
public function actionList()
```

```
{
```

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

```
    $dataProvider = new ActiveDataProvider([
```

```
        'query' => $model->find();
```

```
        'pagination' => [
```

```
            'pagesize' => '10',
```

```
        ]
```

```
    ]);
```

```
    return $this->render('list', ['model' => $model, 'dataProvider' => $dataProvider]);
```

```
}
```

view 代码 list.php

```
use yii\grid\GridView; 这是必须的;
```

```
GridView::widget([
```

```
    'dataProvider' => $dataProvider,
```

```

        'columns' => [

            'attribute', (attribute 为字段的名称，开发时候根据自己的需要进行修改)

            [

                'attribute' => 'create_time',

                'format' => ['data', 'Y-m-d H:i:s'],

                'options' => ['width' => '200']

            ],

            ['class' => 'yii\grid\ActionColumn', 'header' => '操作', 'headerOptions' => ['width' =>
'80']],
        ]);

```

下面小伙为大家介绍第二种分页方法：

控制器 **CommentController** 里面的任意一个方法，在这里我的方法是 **actionComment()**;

```

<?php

use yii\data\Pagination;
use app\models\Comment;

public function actionComment(){

    $data = Comment::find()->andWhere(['id' => '10']);

    $pages = new Pagination(['totalCount' => $data->count(), 'pageSize' => '2']);

    $model = $data->offset($pages->offset)->limit($pages->limit)->all();

    return $this->render('comment',[

        'model' => $model,

        'pages' => $pages,

    ]);

}

```

```
?>
```

好的，到这里，控制器部分基本就结束了。我们接续看 **view** 里面的代码：

Comment.php 文件代码如下所示

```
<?php

use yii\widgets\LinkPager;

?>


    foreach($model as $key=>$val)

    {

        这里就是遍历数据了，省略.....

    }


    <?= LinkPager::widget(['pagination' => $pages]); ?>
```

排序(sorting)

<http://www.yiichina.com/guide/2/output-sorting>

数据提供者 Data providers

Note: This section is under development.

Data provider abstracts data set via `[[yii\data\DataProviderInterface]]` and handles pagination and sorting. It can be used by [grids](#), [lists](#) and [other data widgets](#).

In Yii there are three built-in data providers: `[[yii\data\ActiveDataProvider]]`, `[[yii\data\ArrayDataProvider]]` and `[[yii\data\SqlDataProvider]]`.

Active data provider

`ActiveDataProvider` provides data by performing DB queries using `[[yii\db\Query]]` and `[[yii\db\ActiveQuery]]`.

The following is an example of using it to provide ActiveRecord instances:

```
$provider = new ActiveDataProvider([
```

```
'query' => Post::find(),

'pagination' => [

    'pageSize' => 20,

],

]);
```

// get the posts in the current page

```
$posts = $provider->getModels();
```

And the following example shows how to use `ActiveDataProvider` without `ActiveRecord`:

```
$query = new Query();
```

```
$provider = new ActiveDataProvider([
```

```
    'query' => $query->from('post'),
```

```
    'sort' => [
```

```
        // Set the default sort by name ASC and created_at DESC.
```

```
        'defaultOrder' => [
```

```
            'name' => SORT_ASC,
```

```
            'created_at' => SORT_DESC
```

```
        ]
```

```
    ],
```

```
    'pagination' => [
```

```
        'pageSize' => 20,
```

```
    ],
```

```
]);
```

// get the posts in the current page

```
$posts = $provider->getModels();
```

Array data provider

ArrayDataProvider implements a data provider based on a data array.

The `[[yii\data\ArrayDataProvider::$allModels]]` property contains all data models that may be sorted and/or paginated. ArrayDataProvider will provide the data after sorting and/or pagination. You may configure the `[[yii\data\ArrayDataProvider::$sort]]` and `[[yii\data\ArrayDataProvider::$pagination]]` properties to customize the sorting and pagination behaviors.

Elements in the `[[yii\data\ArrayDataProvider::$allModels]]` array may be either objects (e.g. model objects) or associative arrays (e.g. query results of DAO). Make sure to set the `[[yii\data\ArrayDataProvider::$key]]` property to the name of the field that uniquely identifies a data record or false if you do not have such a field.

Compared to [ActiveDataProvider](#), [ArrayDataProvider](#) could be less efficient because it needs to have `[[yii\data\ArrayDataProvider::$allModels]]` ready.

ArrayDataProvider may be used in the following way:

```
$query = new Query();  
$provider = new ArrayDataProvider([  
    'allModels' => $query->from('post')->all(),  
    'sort' => [  
        'attributes' => ['id', 'username', 'email'],  
    ],  
    'pagination' => [  
        'pageSize' => 10,  
    ],  
]);  
  
// get the posts in the current page  
$posts = $provider->getModels();
```

Note: if you want to use the sorting feature, you must configure the `[[sort]]` property so that the provider knows which columns can be sorted.

SQL data provider

SqlDataProvider implements a data provider based on a plain SQL statement. It provides data in terms of arrays, each representing a row of query result.

Like other data providers, `SqlDataProvider` also supports sorting and pagination. It does so by modifying the given `[[yii\data\SqlDataProvider::$sql]]` statement with "ORDER BY" and "LIMIT" clauses. You may configure the `[[yii\data\SqlDataProvider::$sort]]` and `[[yii\data\SqlDataProvider::$pagination]]` properties to customize sorting and pagination behaviors.

`SqlDataProvider` may be used in the following way:

```
$count = Yii::$app->db->createCommand('

    SELECT COUNT(*) FROM user WHERE status=:status

', [':status' => 1])->queryScalar();

$dataProvider = new SqlDataProvider([

    'sql' => 'SELECT * FROM user WHERE status=:status',

    'params' => [':status' => 1],

    'totalCount' => $count,

    'sort' => [

        'attributes' => [

            'age',

            'name' => [

                'asc' => ['first_name' => SORT_ASC, 'last_name' => SORT_ASC],

                'desc' => ['first_name' => SORT_DESC, 'last_name' => SORT_DESC],

                'default' => SORT_DESC,

                'label' => 'Name',

            ],

        ],

    ],

    'pagination' => [

        'pageSize' => 20,

    ],

]);
```

```
// get the user records in the current page
```

```
$models = $dataProvider->getModels();
```

Note: if you want to use the pagination feature, you must configure the `[[yii\data\SqlDataProvider::$totalCount]]` property to be the total number of rows (without pagination). And if you want to use the sorting feature, you must configure the `[[yii\data\SqlDataProvider::$sort]]` property so that the provider knows which columns can be sorted.

Implementing your own custom data provider

TBD

数据小部件:Data widgets

Note: This section is under development.

Listview

DetailView

DetailView displays the detail of a single data `[[yii\widgets\DetailView::$model|model]]`.

It is best used for displaying a model in a regular format (e.g. each model attribute is displayed as a row in a table).

The model can be either an instance of `[[yii\base\Model]]` or an associative array.

DetailView uses the `[[yii\widgets\DetailView::$attributes]]` property to determines which model attributes should be displayed and how they should be formatted.

A typical usage of Detailview is as follows:

```
echo DetailView::widget([

    'model' => $model,

    'attributes' => [

        'title',          // title attribute (in plain text)

        'description:html', // description attribute in HTML

        [                 // the owner name of the model
```

```

        'label' => 'Owner',

        'value' => $model->owner->name,

    ],

],

]);

```

GridView

Data grid or GridView is one of the most powerful Yii widgets. It is extremely useful if you need to quickly build admin section of the system. It takes data from [data provider](#) and renders each row using a set of columns presenting data in a form of a table.

Each row of the table represents the data of a single data item, and a column usually represents an attribute of the item (some columns may correspond to complex expression of attributes or static text).

Grid view supports both sorting and pagination of the data items. The sorting and pagination can be done in AJAX mode or normal page request. A benefit of using GridView is that when the user disables JavaScript, the sorting and pagination automatically degrade to normal page requests and are still functioning as expected.

The minimal code needed to use GridView is as follows:

```

use yii\grid\GridView;

use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([

    'query' => Post::find(),

    'pagination' => [

        'pageSize' => 20,

    ],

]);

echo GridView::widget([

    'dataProvider' => $dataProvider,

]);

```

The above code first creates a data provider and then uses GridView to display every attribute in every row taken from data provider. The displayed table is equipped with

sorting and pagination functionality.

Grid columns

Yii grid consists of a number of columns. Depending on column type and settings these are able to present data differently.

These are defined in the columns part of GridView config like the following:

```
echo GridView::widget([

    'dataProvider' => $dataProvider,

    'columns' => [

        ['class' => 'yii\grid\SerialColumn'],

        // A simple column defined by the data contained in $dataProvider.

        // Data from model's column1 will be used.

        'id',

        'username',

        // More complex one.

        [

            'class' => 'yii\grid\DataColumn', // can be omitted, default

            'value' => function ($data) {

                return $data->name;

            },

        ],

    ],

]);
```

Note that if columns part of config isn't specified, Yii tries to show all possible data provider model columns.

Column classes

Grid columns could be customized by using different column classes:

```
echo GridView::widget([
```

```
'dataProvider' => $dataProvider,

'columns' => [

    [

        'class' => 'yii\grid\SerialColumn', // <-- here

        // you may configure additional properties here

    ],
```

Additionally to column classes provided by Yii that we'll review below you can create your own column classes.

Each column class extends from `[[yii\grid\Column]]` so there some common options you can set while configuring grid columns.

- **header** allows to set content for header row.
- **footer** allows to set content for footer row.
- **visible** is the column should be visible.
- **content** allows you to pass a valid PHP callback that will return data for a row. The format is the following:

```
function ($model, $key, $index, $column) {

    return 'a string';

}
```

You may specify various container HTML options passing arrays to:

- **headerOptions**
- **contentOptions**
- **footerOptions**
- **filterOptions**

Data column

Data column is for displaying and sorting data. It is default column type so specifying class could be omitted when using it.

The main setting of the data column is its format. It could be specified via **format** attribute. Its values are corresponding to methods in **format** application component that is `[[yii\i18n\Formatter|Formatter]]` by default:

```
<?= GridView::widget([

    'columns' => [

        [
```

```

        'attribute' => 'name',

        'format' => 'text'

    ],

    [

        'attribute' => 'birthday',

        'format' => ['date', 'Y-m-d']

    ],

],

]); ?>

```

In the above `text` corresponds to `[[yii\i18n\Formatter::asText()]]`. The value of the column is passed as the first argument. In the second column definition `date` corresponds to `[[yii\i18n\Formatter::asDate()]]`. The value of the column is, again, passed as the first argument while 'Y-m-d' is used as the second argument value.

For a list of available formatters see the [section about Data Formatting](#).

Action column

Action column displays action buttons such as update or delete for each row.

```

echo GridView::widget([

    'dataProvider' => $dataProvider,

    'columns' => [

        [

            'class' => 'yii\grid\ActionColumn',

            // you may configure additional properties here

        ],

    ],

]);

```

Available properties you can configure are:

- `controller` is the ID of the controller that should handle the actions. If not set, it will use the currently active controller.
- `template` the template used for composing each cell in the action column. Tokens enclosed within curly brackets are treated as controller action IDs (also called button names in the context of action column). They will be replaced by the corresponding button rendering callbacks specified in `[[yii\grid\ActionColumn::$buttons|buttons]]`. For example, the token `{view}` will be replaced by the result of

the callback `buttons['view']`. If a callback cannot be found, the token will be replaced with an empty string. Default is `{view} {update} {delete}`.

- `buttons` is an array of button rendering callbacks. The array keys are the button names (without curly brackets), and the values are the corresponding button rendering callbacks. The callbacks should use the following signature:

```
function ($url, $model) {  
  
    // return the button HTML code  
  
}
```

In the code above `$url` is the URL that the column creates for the button, and `$model` is the model object being rendered for the current row.

- `urlCreator` is a callback that creates a button URL using the specified model information. The signature of the callback should be the same as that of `[[yii\grid\ActionColumn::createUrl()]]`. If this property is not set, button URLs will be created using `[[yii\grid\ActionColumn::createUrl()]]`.

Checkbox column

CheckboxColumn displays a column of checkboxes.

To add a CheckboxColumn to the `[[yii\grid\GridView]]`, add it to the `[[yii\grid\GridView::$columns|columns]]` configuration as follows:

```
echo GridView::widget([  
  
    'dataProvider' => $dataProvider,  
  
    'columns' => [  
  
        // ...  
  
        [  
  
            'class' => 'yii\grid\CheckboxColumn',  
  
            // you may configure additional properties here  
  
        ],  
  
    ],  
],
```

Users may click on the checkboxes to select rows of the grid. The selected rows may be obtained by calling the following JavaScript code:

```
var keys = $('#grid').yiiGridView('getSelectedRows');  
  
// keys is an array consisting of the keys associated with the selected rows
```

Serial column

Serial column renders row numbers starting with **1** and going forward.

Usage is as simple as the following:

```
echo GridView::widget([  
    'dataProvider' => $dataProvider,  
    'columns' => [  
        ['class' => 'yii\grid\SerialColumn'], // <-- here  
        // ...  
    ],  
]);
```

Sorting data

- <https://github.com/yiisoft/yii2/issues/1576>

Filtering data

For filtering data the GridView needs a **model** that takes the input from the filtering form and adjusts the query of the dataProvider to respect the search criteria. A common practice when using **active records** is to create a search Model class that provides needed functionality (it can be generated for you by Gii). This class defines the validation rules for the search and provides a **search()** method that will return the data provider.

To add search capability for the **Post** model we can create **PostSearch** like in the following example:

```
<?php  
  
namespace app\models;  
  
use Yii;  
use yii\base\Model;  
use yii\data\ActiveDataProvider;  
  
class PostSearch extends Post  
{  
    public function rules()  
{
```

```

// only fields in rules() are searchable

return [

    [['id'], 'integer'],

    [['title', 'creation_date'], 'safe'],

];

}

public function scenarios()

{

    // bypass scenarios() implementation in the parent class

    return Model::scenarios();

}

public function search($params)

{

    $query = Post::find();

    $dataProvider = new ActiveDataProvider([

        'query' => $query,

    ]);

    // load the search form data and validate

    if (!$this->load($params) && $this->validate()) {

        return $dataProvider;

    }

    // adjust the query by adding the filters

```

```

$query->andWhere(['id' => $this->id]);

$query->andWhere(['like', 'title', $this->name])

->andWhere(['like', 'creation_date', $this->creation_date]);

return $dataProvider;

}
}

```

You can use this function in the controller to get the dataProvider for the GridView:

```

$searchModel = new PostSearch();

$dataProvider = $searchModel->search(Yii::$app->request->get());

return $this->render('myview', [

    'dataProvider' => $dataProvider,

    'searchModel' => $searchModel,

]);

```

And in the view you then assign the `$dataProvider` and `$searchModel` to the GridView:

```

echo GridView::widget([

    'dataProvider' => $dataProvider,

    'filterModel' => $searchModel,

]);

```

Working with model relations

When displaying active records in a GridView you might encounter the case where you display values of related columns such as the post's author's name instead of just his `id`. You do this by defining the attribute name in columns as `author.name` when the `Post` model has a relation named `author` and the author model has an attribute `name`. The GridView will then display the name of the author but sorting and filtering are not enabled by default. You have to adjust the `PostSearch` model that has been introduced in the last section to add this functionality.

To enable sorting on a related column you have to join the related table and add the sorting rule to the Sort component of the data provider:

```

$query = Post::find();

```

```

$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);

// join with relation `author` that is a relation to the table `users`
// and set the table alias to be `author`
$query->joinWith(['author' => function($query) { $query->from(['author' => 'users']); }]);
// enable sorting for the related column
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...

```

Filtering also needs the `joinWith` call as above. You also need to define the searchable column in attributes and rules like this:

```

public function attributes()
{
    // add related fields to searchable attributes

    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],

        [['title', 'creation_date', 'author.name'], 'safe'],

    ];
}

```


In `search()` you then just add another filter condition with:

```
$query->andWhere(['LIKE', 'author.name', $this->getAttribute('author.name')]);
```

Info: In the above we use the same string for the relation name and the table alias, however when your alias and relation name differ, you have to pay attention on where to use the alias and where to use the relation name. A simple rule for this is to use the alias in every place that is used to build the database query and the relation name in all other definitions like in `attributes()` and `rules()` etc.

For example you use the alias `au` for the author relation table, the `joinWith` statement looks like the following:

```
$query->joinWith(['author' => function($query) { $query->from(['au' => 'users']); }]);
```

It is also possible to just call `$query->joinWith(['author'])`; when the alias is defined in the relation definition.

The alias has to be used in the filter condition but the attribute name stays the same:

```
$query->andWhere(['LIKE', 'au.name', $this->getAttribute('author.name')]);
```

Same is true for the sorting definition:

```
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['au.name' => SORT_ASC],
    'desc' => ['au.name' => SORT_DESC],
];
```

Also when specifying the `[[yii\data\Sort::defaultOrder|defaultOrder]]` for sorting you need to use the relation name instead of the alias:

```
$dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
```

Info: For more information on `joinWith` and the queries performed in the background, check the [active record docs on eager and lazy loading](#).

Using sql views for filtering, sorting and displaying data

There is also other approach that can be faster and more useful - sql views. So for example if we need to show gridview with users and their profiles we can do it in this way:

```
CREATE OR REPLACE VIEW vw_user_info AS

SELECT user.*, user_profile.lastname, user_profile.firstname

FROM user, user_profile
```

```
WHERE user.id = user_profile.user_id
```

Then you need to create ActiveRecord that will be representing this view:

```
namespace app\models\views\grid;

use yii\db\ActiveRecord;

class UserView extends ActiveRecord
{

    /**
     * @inheritdoc
     */

    public static function tableName()
    {
        return 'vw_user_info';
    }

    public static function primaryKey()
    {
        return ['id'];
    }

    /**
     * @inheritdoc
     */

    public function rules()
    {

```

```

return [

    // define here your rules

];

}

/**
 * @inheritdoc
 */

public static function attributeLabels()
{
    return [

        // define here your attribute labels

    ];
}
}

```

After that you can use this UserView active record with search models, without additional specifying of sorting and filtering attributes. All attributes will be working out of the box. Note that this approach has several pros and cons:

- you don't need to specify different sorting and filtering conditions and other things. Everything works out of the box;
- it can be much faster because of data size, count of sql queries performed (for each relation you will need additional query);
- since this is a just simple mapping UI on sql view it lacks of some domain logic that is in your entities, so if you will have some methods like `isActive`, `isDeleted` or other that will influence on UI you will need to duplicate them in this class too.

Multiple GridViews on one page

You can use more than one GridView on a single page but some additional configuration is needed so that they do not interfere. When using multiple instances of GridView you have

to configure different parameter names for the generated sort and pagination links so that each GridView has its individual sorting and pagination. You do so by setting the `[[yii\data\Sort::sortParam|sortParam]]` and `[[yii\data\Pagination::pageParam|pageParam]]` of the dataProviders `[[yii\data\BaseDataProvider::$sort|sort]]` and `[[yii\data\BaseDataProvider::$pagination|pagination]]` instance.

Assume we want to list `Post` and `User` models for which we have already prepared two data providers in `$userProvider` and `$postProvider`:

```
use yii\grid\GridView;

$userProvider->pagination->pageParam = 'user-page';
$userProvider->sort->sortParam = 'user-sort';

$postProvider->pagination->pageParam = 'post-page';
$postProvider->sort->sortParam = 'post-sort';

echo '<h1>Users</h1>';
echo GridView::widget([
    'dataProvider' => $userProvider,
]);

echo '<h1>Posts</h1>';
echo GridView::widget([
    'dataProvider' => $postProvider,
]);
```

Using GridView with Pjax

主题:Theming

Note: This section is under development.

A theme is a directory of view and layout files. Each file of the theme overrides corresponding file of an application when rendered. A single application may use multiple themes and each may provide totally different experience. At any time only one theme can be active.

Note: Themes usually do not meant to be redistributed since views are too application specific. If you want to redistribute customized look and feel consider CSS and JavaScript files in form of [asset bundles](#) instead.

Configuring a theme

Theme configuration is specified via `view` component of the application. In order to set up a theme to work with basic application views the following should be in your application config file:

```
'components' => [  
  'view' => [  
    'theme' => [  
      'pathMap' => ['@app/views' => '@app/themes/basic'],  
      'baseUrl' => '@web/themes/basic',  
    ],  
  ],  
],
```

In the above `pathMap` defines a map of original paths to themed paths while `baseUrl` defines base URL for resources referenced from theme files.

In our case `pathMap` is `['@app/views' => '@app/themes/basic']`. That means that every view in `@app/views` will be first searched under `@app/themes/basic` and if a view exists in the theme directory it will be used instead of the original view.

For example, with a configuration above a themed version of a view file `@app/views/site/index.php` will be `@app/themes/basic/site/index.php`. It basically replaces `@app/views` in `@app/views/site/index.php` with `@app/themes/basic`.

Theming modules

In order to theme modules `pathMap` may look like the following:

```
'components' => [  
  'view' => [  
    'theme' => [  
      'pathMap' => [  
        '@app/views' => '@app/themes/basic',
```

```

        '@app/modules' => '@app/themes/basic/modules', // <-- !!!
    ],
],
],
],
],

```

It will allow you to theme `@app/modules/blog/views/comment/index.php` with `@app/themes/basic/modules/blog/views/comment/index.php`.

Theming widgets

In order to theme a widget view located at `@app/widgets/currency/views/index.php` you need the following config for view component theme:

```

'components' => [
    'view' => [
        'theme' => [
            'pathMap' => ['@app/widgets' => '@app/themes/basic/widgets'],
        ],
    ],
],

```

With the config above you can create themed version of `@app/widgets/currency/index.php` view in `@app/themes/basic/widgets/currency/index.php`.

Using multiple paths

It is possible to map a single path to multiple theme paths. For example,

```

'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
],

```

In this case, the view will be searched in `@app/themes/christmas/site/index.php` then if it's not found it will check `@app/themes/basic/site/index.php`. If there's no view there as well application view will be used.

This ability is especially useful if you want to temporary or conditionally override some views.

安全:

认证 Authentication

Note: This section is under development.

Authentication is the act of verifying who a user is, and is the basis of the login process. Typically, authentication uses the combination of an identifier--a username or email address--and a password. The user submits these values through a form, and the application then compares the submitted information against that previously stored (e.g., upon registration).

In Yii, this entire process is performed semi-automatically, leaving the developer to merely implement `[[yii\web\IdentityInterface]]`, the most important class in the authentication system. Typically, implementation of `IdentityInterface` is accomplished using the `User` model.

You can find a fully featured example of authentication in the [advanced application template](#). Below, only the interface methods are listed:

```
class User extends ActiveRecord implements IdentityInterface
{
    // ...

    /**
     * Finds an identity by the given ID.
     *
     * @param string|integer $id the ID to be looked for
     * @return IdentityInterface|null the identity object that matches the given ID.
     */
    public static function findIdentity($id)
    {
```

```

        return static::findOne($id);

    }

    /**
     * Finds an identity by the given token.
     *
     * @param string $token the token to be looked for
     * @return IdentityInterface|null the identity object that matches the given token.
     */

    public static function findIdentityByAccessToken($token, $type = null)

    {

        return static::findOne(['access_token' => $token]);

    }

    /**
     * @return int|string current user ID
     */

    public function getId()

    {

        return $this->id;

    }

    /**
     * @return string current user auth key
     */

    public function getAuthKey()

    {

```



```

        return $this->auth_key;

    }

    /**
     * @param string $authKey
     * @return boolean if auth key is valid for current user
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}

```

Two of the outlined methods are simple: `findIdentity` is provided with an ID value and returns a model instance associated with that ID. The `getId` method returns the ID itself. Two of the other methods – `getAuthKey` and `validateAuthKey` – are used to provide extra security to the "remember me" cookie. The `getAuthKey` method should return a string that is unique for each user. You can reliably create a unique string using `Yii::$app->getSecurity()->generateRandomString()`. It's a good idea to also save this as part of the user's record:

```

public function beforeSave($insert)
{
    if (parent::beforeSave($insert)) {
        if ($this->isNewRecord) {
            $this->auth_key = Yii::$app->getSecurity()->generateRandomString();
        }

        return true;
    }

    return false;
}

```

The `validateAuthKey` method just needs to compare the `$authKey` variable, passed as

parameter (itself retrieved from a cookie), with the value fetched from database.

授权 Authorization

Note: This section is under development.

Authorization is the process of verifying that a user has enough permission to do something. Yii provides two authorization methods: Access Control Filter (ACF) and Role-Based Access Control (RBAC).

Access Control Filter

Access Control Filter (ACF) is a simple authorization method that is best used by applications that only need some simple access control. As its name indicates, ACF is an action filter that can be attached to a controller or a module as a behavior. ACF will check a set of `[[yii\filters\AccessControl::rules|access rules]]` to make sure the current user can access the requested action.

The code below shows how to use ACF which is implemented as `[[yii\filters\AccessControl]]`:

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
                    ],
                ],
            ],
        ];
    }
}
```

```

        ],
        [
            'allow' => true,
            'actions' => ['logout'],
            'roles' => ['@'],
        ],
    ],
],
];
}
// ...
}

```

In the code above ACF is attached to the `site` controller as a behavior. This is the typical way of using an action filter. The `only` option specifies that the ACF should only be applied to `login`, `logout` and `signup` actions. The `rules` option specifies the `[[yii\filters\AccessRule|access rules]]`, which reads as follows:

- Allow all guest (not yet authenticated) users to access 'login' and 'signup' actions. The `roles` option contains a question mark `?` which is a special token recognized as "guests".
- Allow authenticated users to access 'logout' action. The `@` character is another special token recognized as authenticated users.

When ACF performs authorization check, it will examine the rules one by one from top to bottom until it finds a match. The `allow` value of the matching rule will then be used to judge if the user is authorized. If none of the rules matches, it means the user is NOT authorized and ACF will stop further action execution.

By default, ACF does only of the followings when it determines a user is not authorized to access the current action:

- If the user is a guest, it will call `[[yii\web\User::loginRequired()]]`, which may redirect the browser to the login page.
- If the user is already authenticated, it will throw a `[[yii\web\ForbiddenHttpException]]`.

You may customize this behavior by configuring the `[[yii\filters\AccessControl::denyCallback]]` property:

```

[

```

```
'class' => AccessControl::className(),

'denyCallback' => function ($rule, $action) {

    throw new \Exception('You are not allowed to access this page');

}
```

[[yii\filters\AccessRule|Access rules]] support many options. Below is a summary of the supported options. You may also extend [[yii\filters\AccessRule]] to create your own customized access rule classes.

- [[yii\filters\AccessRule::allow|allow]]: specifies whether this is an "allow" or "deny" rule.
- [[yii\filters\AccessRule::actions|actions]]: specifies which actions this rule matches. This should be an array of action IDs. The comparison is case-sensitive. If this option is empty or not set, it means the rule applies to all actions.
- [[yii\filters\AccessRule::controllers|controllers]]: specifies which controllers this rule matches. This should be an array of controller IDs. The comparison is case-sensitive. If this option is empty or not set, it means the rule applies to all controllers.
- [[yii\filters\AccessRule::roles|roles]]: specifies which user roles that this rule matches. Two special roles are recognized, and they are checked via [[yii\web\User::isGuest]]:
 - `?`: matches a guest user (not authenticated yet)
 - `@`: matches an authenticated user Using other role names requires RBAC (to be described in the next section), and [[yii\web\User::can()]] will be called. If this option is empty or not set, it means this rule applies to all roles.
- [[yii\filters\AccessRule::ips|ips]]: specifies which [[yii\web\Request::userIP|client IP addresses]] this rule matches. An IP address can contain the wildcard `*` at the end so that it matches IP addresses with the same prefix. For example, '192.168.*' matches all IP addresses in the segment '192.168.'. If this option is empty or not set, it means this rule applies to all IP addresses.
- [[yii\filters\AccessRule::verbs|verbs]]: specifies which request method (e.g. `GET`, `POST`) this rule matches. The comparison is case-insensitive.
- [[yii\filters\AccessRule::matchCallback|matchCallback]]: specifies a PHP callable that should be called to determine if this rule should be applied.
- [[yii\filters\AccessRule::denyCallback|denyCallback]]: specifies a PHP callable that should be called when this rule will deny the access.

Below is an example showing how to make use of the `matchCallback` option, which allows you to write arbitrary access check logic:

```
use yii\filters\AccessControl;
```

```
class SiteController extends Controller
```

```
{
```

```
  public function behaviors()
```

```
  {
```

```
    return [
```

```
      'access' => [
```

```
        'class' => AccessControl::className(),
```

```
        'only' => ['special-callback'],
```

```
        'rules' => [
```

```
          [
```

```
            'actions' => ['special-callback'],
```

```
            'allow' => true,
```

```
            'matchCallback' => function ($rule, $action) {
```

```
              return date('d-m') === '31-10';
```

```
            }
```

```
          ],
```

```
        ],
```

```
      ],
```

```
    ];
```

```
  }
```

```
  // Match callback called! This page can be accessed only each October 31st
```

```
  public function actionSpecialCallback()
```

```
  {
```

```
    return $this->render('happy-halloween');
```

```
  }
```

```
}
```

Role based access control (RBAC)

Role-Based Access Control (RBAC) provides a simple yet powerful centralized access control. Please refer to the [Wiki article](#) for details about comparing RBAC with other more traditional access control schemes.

Yii implements a General Hierarchical RBAC, following the [NIST RBAC model](#). It provides the RBAC functionality through the `[[yii\rbac\ManagerInterface|authManager]]` application component.

Using RBAC involves two parts of work. The first part is to build up the RBAC authorization data, and the second part is to use the authorization data to perform access check in places where it is needed.

To facilitate our description next, we will first introduce some basic RBAC concepts.

Basic Concepts

A role represents a collection of permissions (e.g. creating posts, updating posts). A role may be assigned to one or multiple users. To check if a user has a specified permission, we may check if the user is assigned with a role that contains that permission.

Associated with each role or permission, there may be a rule. A rule represents a piece of code that will be executed during access check to determine if the corresponding role or permission applies to the current user. For example, the "update post" permission may have a rule that checks if the current user is the post creator. During access checking, if the user is NOT the post creator, he/she will be considered not having the "update post" permission.

Both roles and permissions can be organized in a hierarchy. In particular, a role may consist of other roles or permissions; and a permission may consist of other permissions. Yii implements a partial order hierarchy which includes the more special tree hierarchy. While a role can contain a permission, it is not true vice versa.

Configuring RBAC Manager

Before we set off to define authorization data and perform access checking, we need to configure the `[[yii\base\Application::authManager|authManager]]` application component. Yii provides two types of authorization managers: `[[yii\rbac\PhpManager]]` and `[[yii\rbac\DbManager]]`. The former uses a PHP script file to store authorization data, while the latter stores authorization data in database. You may consider using the former if your application does not require very dynamic role and permission management.

The following code shows how to configure `authManager` in the application configuration:

```
return [  
  
    // ...  
  
    'components' => [  

```

```

    'authManager' => [

        'class' => 'yii\rbac\PhpManager',

    ],

    // ...

],

];

```

The `authManager` can now be accessed via `\Yii::$app->authManager`.

Tip: By default, `[[yii\rbac\PhpManager]]` stores RBAC data in the file `@app/data/rbac.php`. Sometime you need to create this file manually.

Building Authorization Data

Building authorization data is all about the following tasks:

- defining roles and permissions;
- establishing relations among roles and permissions;
- defining rules;
- associating rules with roles and permissions;
- assigning roles to users.

Depending on authorization flexibility requirements the tasks above could be done in different ways.

If your permissions hierarchy doesn't change at all and you have a fixed number of users you can create a console command that will initialize authorization data once via APIs offered by `authManager`:

```

<?php

namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{

    public function actionInit()

```

```
{

    $auth = Yii::$app->authManager;

    // add "createPost" permission

    $createPost = $auth->createPermission('createPost');

    $createPost->description = 'Create a post';

    $auth->add($createPost);

    // add "updatePost" permission

    $updatePost = $auth->createPermission('updatePost');

    $updatePost->description = 'Update post';

    $auth->add($updatePost);

    // add "author" role and give this role the "createPost" permission

    $author = $auth->createRole('author');

    $auth->add($author);

    $auth->addChild($author, $createPost);

    // add "admin" role and give this role the "updatePost" permission

    // as well as the permissions of the "author" role

    $admin = $auth->createRole('admin');

    $auth->add($admin);

    $auth->addChild($admin, $updatePost);

    $auth->addChild($admin, $author);

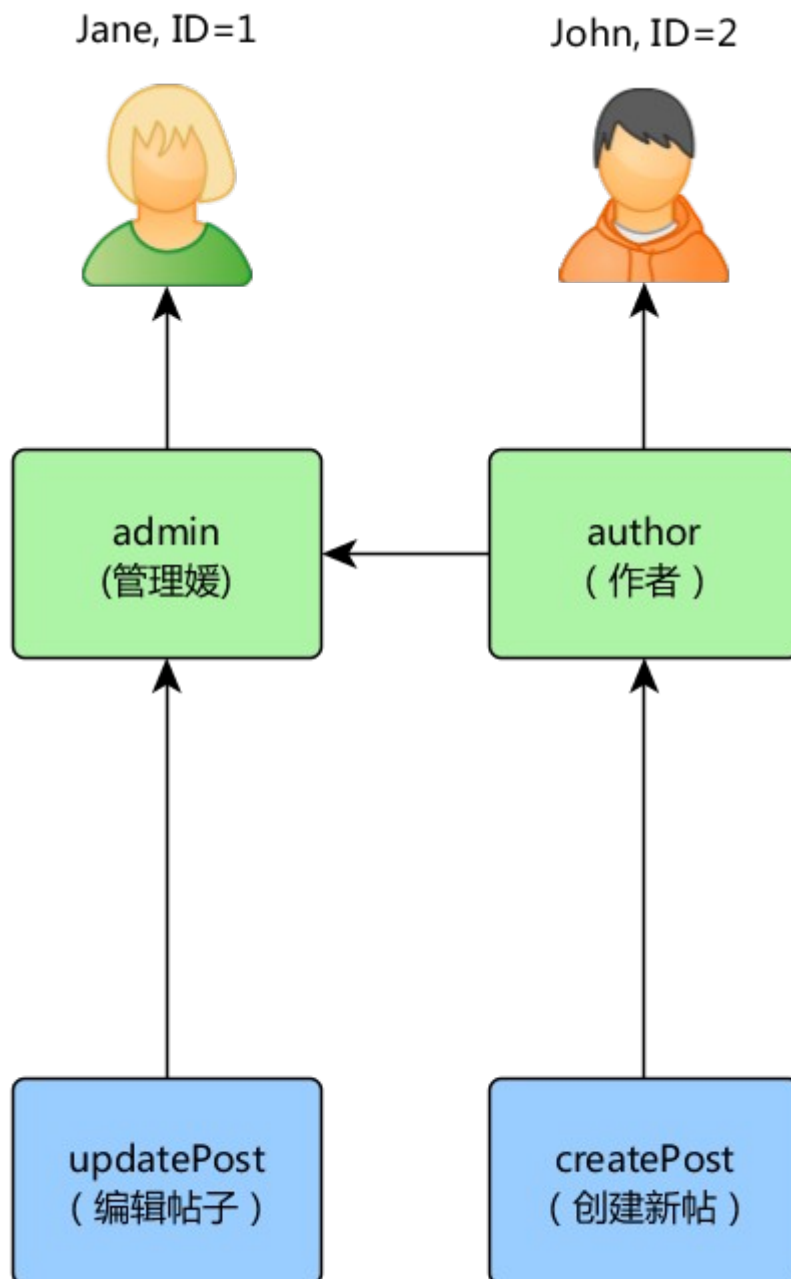
    // Assign roles to users. 1 and 2 are IDs returned by IdentityInterface::getId()

    // usually implemented in your User model.
```



```
$auth->assign($author, 2);  
  
$auth->assign($admin, 1);  
  
}  
  
}
```

After executing the command we'll get the following hierarchy:



Author can create post, admin can update post and do everything author can.

If your application allows user signup you need to assign roles to these new users once. For example, in order for all signed up users to become authors you in advanced application template you need to modify `frontend\models\SignupForm::signup()` as follows:

```
public function signup()
```

```

{

    if ($this->validate()) {

        $user = new User();

        $user->username = $this->username;

        $user->email = $this->email;

        $user->setPassword($this->password);

        $user->generateAuthKey();

        $user->save(false);


        // the following three lines were added:

        $auth = Yii::$app->authManager;

        $authorRole = $auth->getRole('author');

        $auth->assign($authorRole, $user->getId());


        return $user;

    }

    return null;

}

```

For applications that require complex access control with dynamically updated authorization data, special user interfaces (i.e. admin panel) may need to be developed using APIs offered by `authManager`.

Tip: By default, `[[yii\rbac\PhpManager]]` stores RBAC data in the file `@app/data/rbac.php`. Sometimes when you want to make some minor changes to the RBAC data, you may directly edit this file.

Using Rules

As aforementioned, rules add additional constraint to roles and permissions. A rule is a class extending from `[[yii\rbac\Rule]]`. It must implement the `[[yii\rbac\Rule::execute()|execute()]]` method. In the hierarchy we've created previously author cannot edit his own post. Let's fix it. First we need a rule to verify that the user is the post author:

```

namespace app\rbac;

use yii\rbac\Rule;

/**
 * Checks if authorID matches user passed via params
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';

    /**
     * @param string|integer $user the user ID.
     * @param Item $item the role or permission that this rule is associated with
     * @param array $params parameters passed to ManagerInterface::checkAccess().
     * @return boolean a value indicating whether the rule permits the role or permission it is
    associated with.
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user : false;
    }
}

```

The rule above checks if the `post` is created by `$user`. We'll create a special permission `updateOwnPost` in the command we've used previously:

```

// add the rule

$rule = new \app\rbac\AuthorRule;

$auth->add($rule);

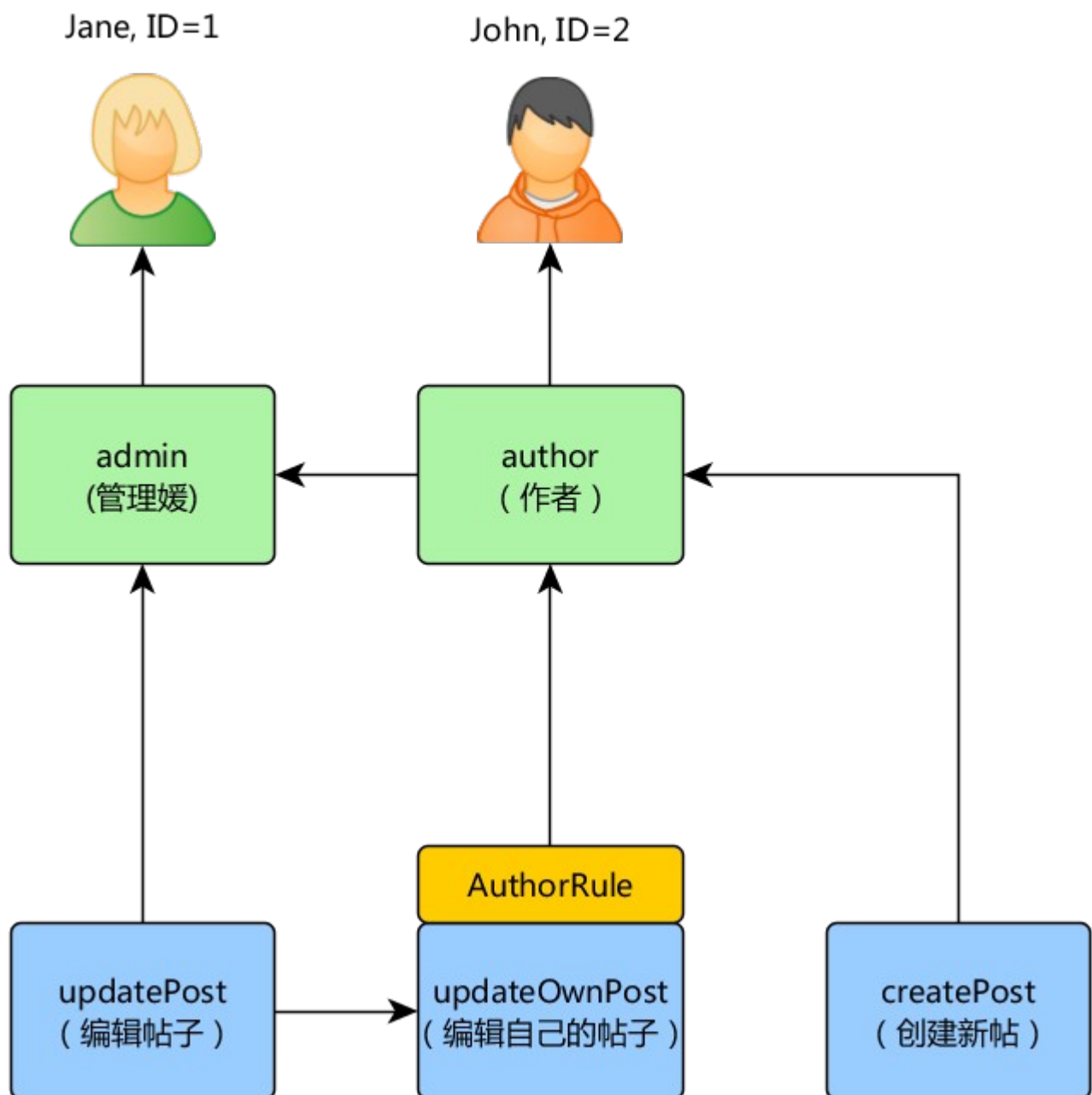
```

```
// add the "updateOwnPost" permission and associate the rule with it.
$updateOwnPost = $this->auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" will be used from "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// allow "author" to update their own posts
$auth->addChild($author, $updateOwnPost);
```

Now we've got the following hierarchy:

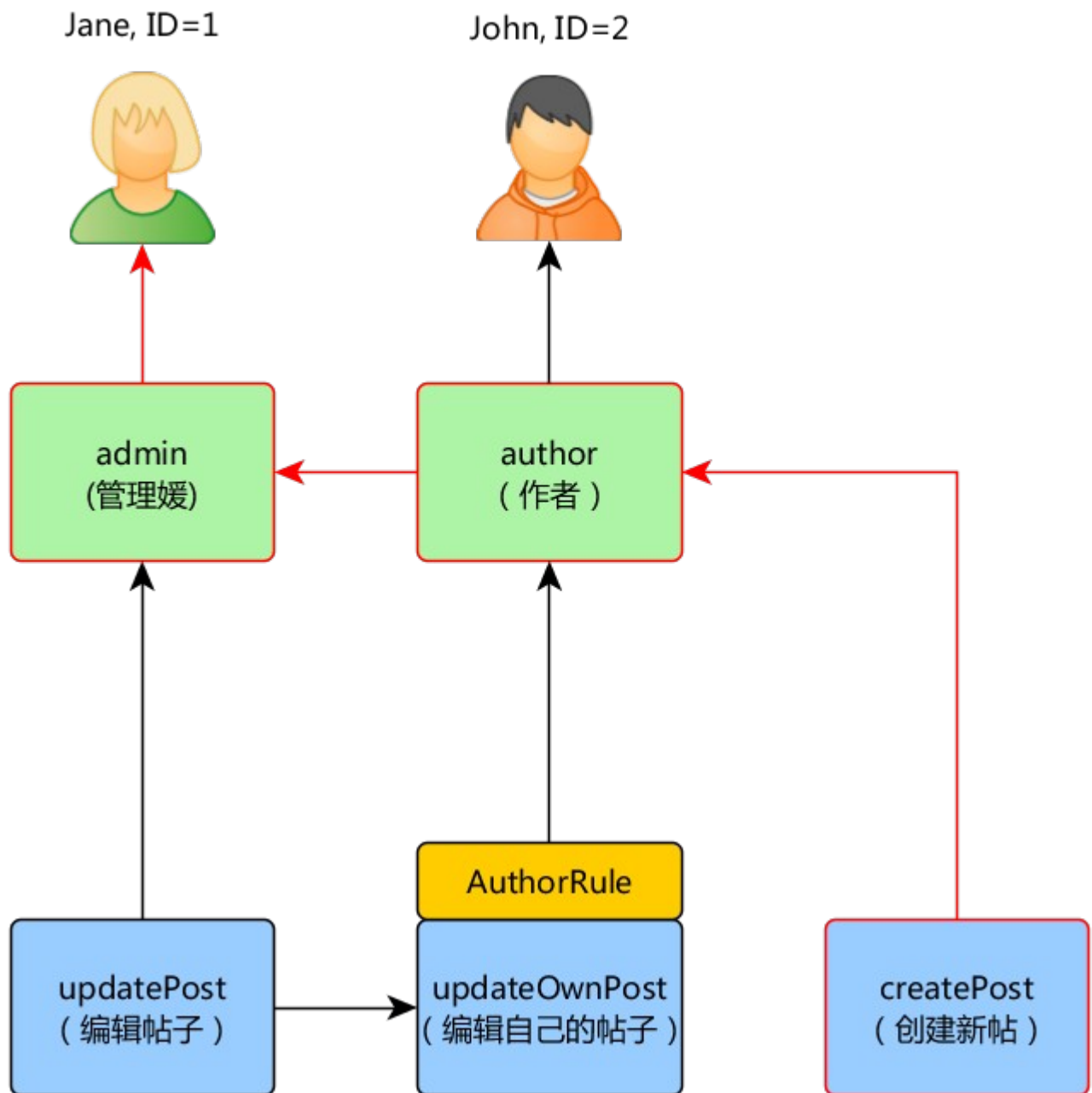


Access Check

With the authorization data ready, access check is as simple as a call to the `[[yii\rbac\ManagerInterface::checkAccess()]]` method. Because most access check is about the current user, for convenience Yii provides a shortcut method `[[yii\web\User::can()]]`, which can be used like the following:

```
if (\Yii::$app->user->can('createPost')) {  
    // create post  
}
```

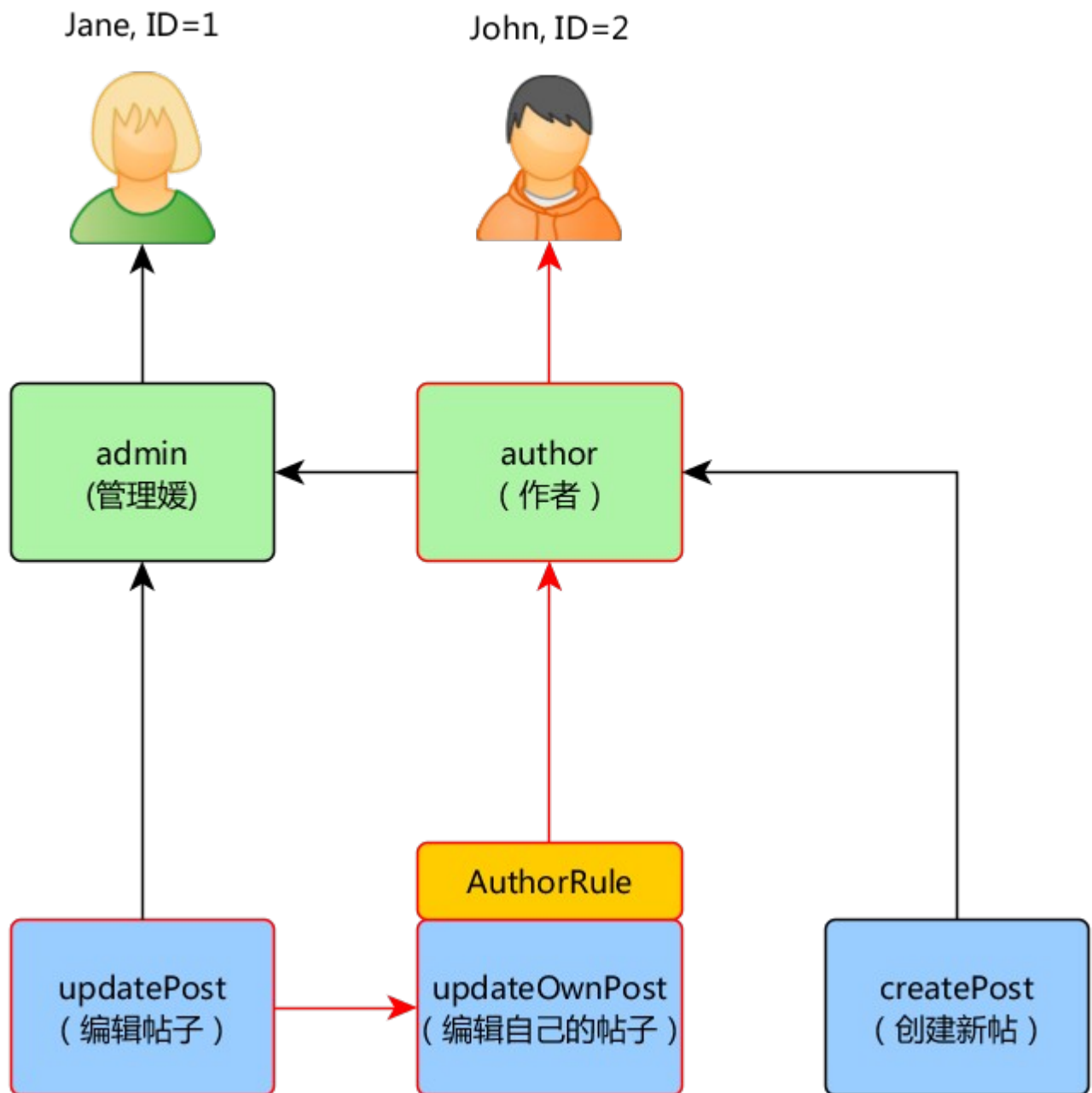
If the current user is Jane with ID=1 we're starting at `createPost` and trying to get to `Jane`:



In order to check if user can update post we need to pass an extra parameter that is required by the **AuthorRule** described before:

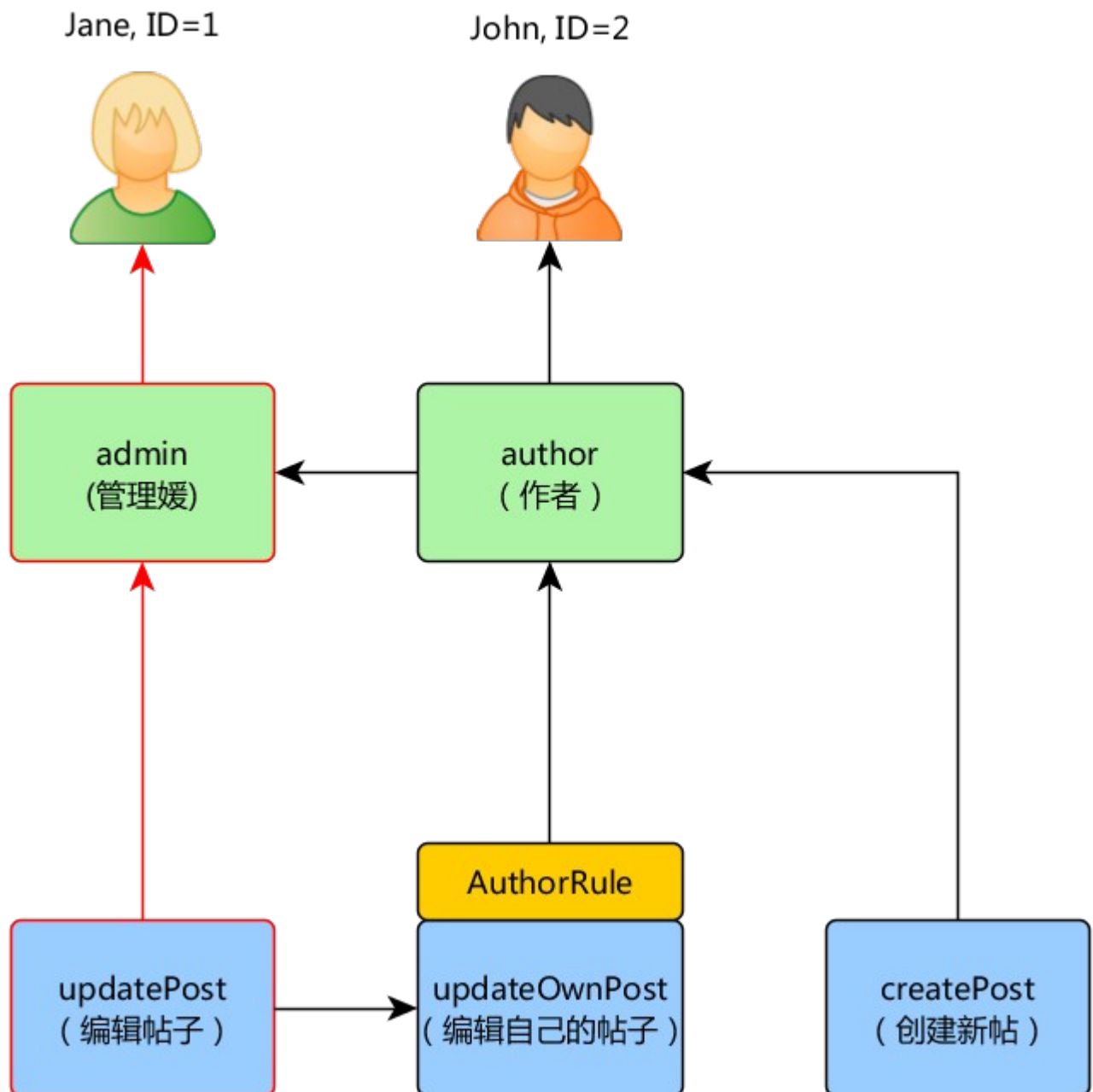
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {  
    // update post  
}
```

Here's what happens if current user is John:



We're starting with the `updatePost` and going through `updateOwnPost`. In order to pass it `AuthorRule` should return `true` from its `execute` method. The method receives its `$params` from `can` method call so the value is `['post' => $post]`. If everything is OK we're getting to `author` that is assigned to John.

In case of Jane it is a bit simpler since she's an admin:



Using Default Roles

A default role is a role that is implicitly assigned to all users. The call to `[[yii\rbac\ManagerInterface::assign()]]` is not needed, and the authorization data does not contain its assignment information.

A default role is usually associated with a rule which determines if the role applies to the user being checked.

Default roles are often used in applications which already have some sort of role assignment. For example, an application may have a "group" column in its user table to represent which privilege group each user belongs to. If each privilege group can be mapped to a RBAC role, you can use the default role feature to automatically assign each user to a RBAC role. Let's use an example to show how this can be done.

Assume in the user table, you have a **group** column which uses 1 to represent the administrator group and 2 the author group. You plan to have two RBAC roles **admin** and **author** to represent the permissions for these two groups, respectively. You can create set up the RBAC data as follows,

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Checks if user group matches
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;

            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }

        return false;
    }
}
```

```

$rule = new \app\rbac\UserGroupRule;

$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);

// ... add permissions as children of $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);

$auth->addChild($admin, $author);

// ... add permissions as children of $admin ...

```

Note that in the above, because "author" is added as a child of "admin", when you implement the `execute()` method of the rule class, you need to respect this hierarchy as well. That is why when the role name is "author", the `execute()` method will return true if the user group is either 1 or 2 (meaning the user is in either "admin" group or "author" group).

Next, configure `authManager` by listing the two roles in `[[yii\rbac\BaseManager::$defaultRoles]]`:

```

return [

    // ...

    'components' => [

        'authManager' => [

            'class' => 'yii\rbac\PhpManager',

            'defaultRoles' => ['admin', 'author'],

        ],

        // ...

    ],

];

```

Now if you perform an access check, both of the `admin` and `author` roles will be checked by evaluating the rules associated with them. If the rule returns true, it means the role applies to the current user. Based on the above rule implementation, this means if the `group` value of a user is 1, the `admin` role would apply to the user; and if the `group` value is 2, the `author` role would apply.

处理密码 Security

Note: This section is under development.

Good security is vital to the health and success of any application. Unfortunately, many developers cut corners when it comes to security, either due to a lack of understanding or because implementation is too much of a hurdle. To make your Yii powered application as secure as possible, Yii has included several excellent and easy to use security features.

Hashing and verifying passwords

Most developers know that passwords cannot be stored in plain text, but many developers believe it's still safe to hash passwords using `md5` or `sha1`. There was a time when using the aforementioned hashing algorithms was sufficient, but modern hardware makes it possible to reverse such hashes very quickly using brute force attacks.

In order to provide increased security for user passwords, even in the worst case scenario (your application is breached), you need to use a hashing algorithm that is resilient against brute force attacks. The best current choice is `bcrypt`. In PHP, you can create a `bcrypt` hash using the `crypt` function. Yii provides two helper functions which make using `crypt` to securely generate and verify hashes easier.

When a user provides a password for the first time (e.g., upon registration), the password needs to be hashed:

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

The hash can then be associated with the corresponding model attribute, so it can be stored in the database for later use.

When a user attempts to log in, the submitted password must be verified against the previously hashed and stored password:

```
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {  
    // all good, logging user in  
} else {  
    // wrong password  
}
```

Generating Pseudorandom data

Pseudorandom data is useful in many situations. For example when resetting a password via email you need to generate a token, save it to the database, and send it via email to end user which in turn will allow them to prove ownership of that account. It is very important that this token be unique and hard to guess, else there is a possibility that attacker can predict the token's value and reset the user's password.

Yii security helper makes generating pseudorandom data simple:

```
$key = Yii::$app->getSecurity()->generateRandomString();
```

Note that you need to have the `openssl` extension installed in order to generate cryptographically secure random data.

Encryption and decryption

Yii provides convenient helper functions that allow you to encrypt/decrypt data using a secret key. The data is passed through the encryption function so that only the person which has the secret key will be able to decrypt it. For example, we need to store some information in our database but we need to make sure only the user which has the secret key can view it (even if the application database is compromised):

```
// $data and $secretKey are obtained from the form
```

```
$encryptedData = Yii::$app->getSecurity()->encrypt($data, $secretKey);
```

```
// store $encryptedData to database
```

Subsequently when user wants to read the data:

```
// $secretKey is obtained from user input, $encryptedData is from the database
```

```
$data = Yii::$app->getSecurity()->decrypt($encryptedData, $secretKey);
```

Confirming data integrity

There are situations in which you need to verify that your data hasn't been tampered with by a third party or even corrupted in some way. Yii provides an easy way to confirm data integrity in the form of two helper functions.

Prefix the data with a hash generated from the secret key and data

```
// $secretKey our application or user secret, $genuineData obtained from a reliable source
```

```
$data = Yii::$app->getSecurity()->hashData($genuineData, $secretKey);
```

Checks if the data integrity has been compromised

```
// $secretKey our application or user secret, $data obtained from an unreliable source
```

```
$data = Yii::$app->getSecurity()->validateData($data, $secretKey);
```

todo: XSS prevention, CSRF prevention, cookie protection, refer to 1.1 guide

You also can disable CSRF validation per controller and/or action, by setting its property:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $enableCsrfValidation = false;

    public function actionIndex()
    {
        // CSRF validation will not be applied to this and other actions
    }
}
```

To disable CSRF validation per custom actions you can do:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function beforeAction($action)
    {
        // ...set `$this->enableCsrfValidation` here based on some conditions...

        // call parent method that will check CSRF if such property is true.

        return parent::beforeAction($action);
    }
}
```

```
}
```

Securing Cookies

- validation
- httpOnly is default

See also

- [Views security](#)

客户端认证(官方待定)

<http://www.yiichina.com/guide/2/security-auth-clients>

安全领域的最佳实践(官方待定)

<http://www.yiichina.com/guide/2/security-best-practices>

缓存:

缓存是提升 Web 应用性能简便有效的方式。通过将相对静态的数据存储到缓存并在收到请求时取回缓存，应用程序便节省了每次重新生成这些数据所需的时间。

缓存可以应用在 Web 应用程序的任何层级任何位置。在服务器端，在较低层面，缓存可能用于存储基础数据，例如从数据库中取出的最新文章列表；在较高的层面，缓存可能用于存储一段或整个 Web 页面，例如最新文章的渲染结果。在客户端，HTTP 缓存可能用于将最近访问的页面内容存储到浏览器缓存中。

Yii 支持如上所有缓存机制：

- [数据缓存](#)
- [片段缓存](#)
- [页面缓存](#)
- [HTTP 缓存](#)

数据缓存

数据缓存是指将一些 PHP 变量存储到缓存中，使用时再从缓存中取回。它也是更高级缓存特性的基础，

例如[查询缓存](#)和[内容缓存](#)。

如下代码是一个典型的数据缓存使用模式。其中 `$cache` 指向[缓存组件](#)：

```
// 尝试从缓存中取回 $data

$data = $cache->get($key);

if ($data === false) {

    // $data 在缓存中没有找到，则重新计算它的值

    // 将 $data 存放到缓存供下次使用

    $cache->set($key, $data);

}

// 这儿 $data 可以使用了。
```

缓存组件

数据缓存需要[缓存组件](#)提供支持，它代表各种缓存存储器，例如内存，文件，数据库。

缓存组件通常注册为应用程序组件，这样它们就可以在全局进行配置与访问。如下代码演示了如何配置应用程序组件 `cache` 使用两个 `memcached` 服务器：

```
'components' => [

    'cache' => [

        'class' => 'yii\caching\MemCache',

        'servers' => [

            [

                'host' => 'server1',

                'port' => 11211,

                'weight' => 100,

            ],

        ],

    ],

];
```

```
[
    [
        'host' => 'server2',

        'port' => 11211,

        'weight' => 50,

    ],

],

],

],
```

然后就可以通过 `Yii::$app->cache` 访问上面的缓存组件了。

由于所有缓存组件都支持同样的一系列 API，并不需要修改使用缓存的业务代码就能直接替换为其他底层缓存组件，只需在应用配置中重新配置一下就可以。例如，你可以将上述配置修改为使用 `[[yii\caching\ApcCache|APC cache]]`:

```
'components' => [

    'cache' => [

        'class' => 'yii\caching\ApcCache',

    ],

],
```

Tip: 你可以注册多个缓存组件，很多依赖缓存的类默认调用名为 `cache` 的组件（例如 `[[yii\web\UrlManager]]`）。

支持的缓存存储器

Yii 支持一系列缓存存储器，概况如下：

- `[[yii\caching\ApcCache]]`: 使用 PHP APC 扩展。这个选项可以认为是集中式应用程序环境中（例如：单一服务器，没有独立的负载均衡器等）最快的缓存方案。
- `[[yii\caching\DbCache]]`: 使用一个数据库的表存储缓存数据。要使用这个缓存，你必须创建一个与 `[[yii\caching\DbCache::cacheTable]]` 对应的表。
- `[[yii\caching\DummyCache]]`: 仅作为一个缓存占位符，不实现任何真正的缓存功能。这个组件的目的是为了简化那些需要查询缓存有效性的代码。例如，在开发中如果服务器没有实际的缓存支持，用它配置一个缓存组件。一个真正的缓存服务启用后，可以再切换为使用相应的缓存组件。两种条件下你都可以使用同样的代码 `Yii::$app->cache->get($key)` 尝试从缓存中取回数据而不用担心 `Yii::$app->cache` 可能是 `null`。
- `[[yii\caching\FileCache]]`: 使用标准文件存储缓存数据。这个特别适用于缓存大块数据，例如一个整页的内容。

- `[[yii\caching\MemCache]]`: 使用 PHP [memcache](#) 和 [memcached](#) 扩展。这个选项被看作分布式应用环境中（例如：多台服务器，有负载均衡等）最快的缓存方案。
- `[[yii\redis\Cache]]`: 实现了一个基于 [Redis](#) 键值对存储器的缓存组件（需要 [redis 2.6.12](#) 及以上版本的支持）。
- `[[yii\caching\WinCache]]`: 使用 PHP [WinCache](#)（[另可参考](#)）扩展。
- `[[yii\caching\XCache]]`: 使用 PHP [XCache](#) 扩展。
- `[[yii\caching\ZendDataCache]]`: 使用 [Zend Data Cache](#) 作为底层缓存媒介。

Tip: 你可以在同一个应用程序中使用不同的缓存存储器。一个常见的策略是使用基于内存的缓存存储器存储小而常用的数据（例如：统计数据），使用基于文件或数据库的缓存存储器存储大而不太常用的数据（例如：网页内容）。

缓存 API

所有缓存组件都有同样的基类 `[[yii\caching\Cache]]`，因此都支持如下 API：

- `[[yii\caching\Cache::get()|get()]]`: 通过一个指定的键（**key**）从缓存中取回一项数据。如果该项数据不存在于缓存中或者已经过期/失效，则返回值 **false**。
- `[[yii\caching\Cache::set()|set()]]`: 将一项数据指定一个键，存放到缓存中。
- `[[yii\caching\Cache::add()|add()]]`: 如果缓存中未找到该键，则将指定数据存放到缓存中。
- `[[yii\caching\Cache::mget()|mget()]]`: 通过指定的多个键从缓存中取回多项数据。
- `[[yii\caching\Cache::mset()|mset()]]`: 将多项数据存储在缓存中，每项数据对应一个键。
- `[[yii\caching\Cache::madd()|madd()]]`: 将多项数据存储在缓存中，每项数据对应一个键。如果某个键已经存在于缓存中，则该项数据会被跳过。
- `[[yii\caching\Cache::exists()|exists()]]`: 返回一个值，指明某个键是否存在于缓存中。
- `[[yii\caching\Cache::delete()|delete()]]`: 通过一个键，删除缓存中对应的值。
- `[[yii\caching\Cache::flush()|flush()]]`: 删除缓存中的所有数据。

有些缓存存储器如 **MemCache**，**APC** 支持以批量模式取回缓存值，这样可以节省取回缓存数据的开支。

`[[yii\caching\Cache::mget()|mget()]]` 和 `[[yii\caching\Cache::madd()|madd()]]` API 提供对该特性的支持。如果底层缓存存储器不支持该特性，Yii 也会模拟实现。

由于 `[[yii\caching\Cache]]` 实现了 PHP [ArrayAccess](#) 接口，缓存组件也可以像数组那样使用，下面是几个例子：

```
$cache['var1'] = $value1; // 等价于: $cache->set('var1', $value1);
$value2 = $cache['var2']; // 等价于: $value2 = $cache->get('var2');
```

缓存键

存储在缓存中的每项数据都通过键作唯一识别。当你在缓存中存储一项数据时，必须为它指定一个键，稍后从缓存中取回数据时，也需要提供相应的键。

你可以使用一个字符串或者任意值作为一个缓存键。当键不是一个字符串时，它将会自动被序列化为一个字符串。

定义一个缓存键常见的一个策略就是在一个数组中包含所有的决定性因素。例如，`[[yii\db\Schema]]` 使用如下键存储一个数据表的结构信息。

```
[
    __CLASS__,          // 结构类名

    $this->db->dsn,        // 数据源名称

    $this->db->username,    // 数据库登录用户名

    $name,               // 表名
];
```

如你所见，该键包含了可唯一指定一个数据库表所需的所有必要信息。

当同一个缓存存储器被用于多个不同的应用时，应该为每个应用指定一个唯一的缓存键前缀以避免缓存键冲突。可以通过配置 `[[yii\caching\Cache::keyPrefix]]` 属性实现。例如，在应用配置中可以编写如下代码：

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',    // 唯一键前缀
    ],
],
```

为了确保互通性，此处只能使用字母和数字。

缓存过期

默认情况下，缓存中的数据会永久存留，除非它被某些缓存策略强制移除（例如：缓存空间已满，最老的数据会被移除）。要改变此特性，你可以在调用 `[[yii\caching\Cache::set()|set()]]` 存储一项数据时提供一个过期时间参数。该参数代表这项数据在缓存中可保持有效多少秒。当你调用 `[[yii\caching\Cache::get()|get()]]` 取回数据时，如果它已经过了超时时间，该方法将返回 `false`，表明在缓存中找不到这项数据。例如：

```
// 将数据在缓存中保留 45 秒
```

```
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);

if ($data === false) {

    // $data 已过期，或者在缓存中找不到

}
```

缓存依赖

除了超时设置，缓存数据还可能受到**缓存依赖**的影响而失效。例如，`[[yii\caching\FileDependency]]` 代表对一个文件修改时间的依赖。这个依赖条件发生变化也就意味着相应的文件已经被修改。因此，缓存中任何过期的文件内容都应该被置为失效状态，对 `[[yii\caching\Cache::get()|get()]]` 的调用都应该返回 `false`。

缓存依赖用 `[[yii\caching\Dependency]]` 的派生类所表示。当调用 `[[yii\caching\Cache::set()|set()]]` 在缓存中存储一项数据时，可以同时传递一个关联的缓存依赖对象。例如：

```
// 创建一个对 example.txt 文件修改时间的缓存依赖

$dependency = new \yii\caching\FileDependency(['fileName' => 'example.txt']);

// 缓存数据将在 30 秒后超时

// 如果 example.txt 被修改，它也可能被更早地置为失效状态。

$cache->set($key, $data, 30, $dependency);

// 缓存会检查数据是否已超时。

// 它还会检查关联的依赖是否已变化。

// 符合任何一个条件时都会返回 false。

$data = $cache->get($key);
```

下面是可用的缓存依赖的概况：

- `[[yii\caching\ChainedDependency]]`：如果依赖链上任何一个依赖产生变化，则依赖改变。
- `[[yii\caching\DbDependency]]`：如果指定 SQL 语句的查询结果发生了变化，则依赖改变。

- `[[yii\caching\ExpressionDependency]]`: 如果指定的 PHP 表达式执行结果发生变化，则依赖改变。
- `[[yii\caching\FileDependency]]`: 如果文件的最后修改时间发生变化，则依赖改变。
- `[[yii\caching\GroupDependency]]`: 将一项缓存数据标记到一个组名，你可以通过调用 `[[yii\caching\GroupDependency::invalidate()]]` 一次性将相同组名的缓存全部置为失效状态。

查询缓存

查询缓存是一个建立在数据缓存之上的特殊缓存特性。它用于缓存数据库查询的结果。

查询缓存需要一个 `[[yii\db\Connection|数据库连接]]` 和一个有效的 `cache` 应用组件。查询缓存的基本用法如下，假设 `$db` 是一个 `[[yii\db\Connection]]` 实例：

```
$duration = 60; // 缓存查询结果 60 秒

$dependency = ...; // 可选的缓存依赖

$db->beginCache($duration, $dependency);

// ...这儿执行数据库查询...

$db->endCache();
```

如你所见，`beginCache()` 和 `endCache()` 中间的任何查询结果都会被缓存起来。如果缓存中找到了同样查询的结果，则查询会被跳过，直接从缓存中提取结果。

查询缓存可以用于 `ActiveRecord` 和 `DAO`。

Info: 有些 DBMS（例如：`MySQL`）也支持数据库服务器端的查询缓存。你可以选择使用任一查询缓存机制。上文所述的查询缓存的好处在于你可以指定更灵活的缓存依赖因此可能更加高效。

配置

查询缓存有两个通过 `[[yii\db\Connection]]` 设置的配置项：

- `[[yii\db\Connection::queryCacheDuration|queryCacheDuration]]`: 查询结果在缓存中的有效期，以秒表示。如果在调用 `[[yii\db\Connection::beginCache()]]` 时传递了一个显式的时值参数，则配置中的有效期时值会被覆盖。
- `[[yii\db\Connection::queryCache|queryCache]]`: 缓存应用组件的 ID。默认为 `'cache'`。只有在设置了一个有效的缓存应用组件时，查询缓存才会有效。

限制条件

当查询结果中含有资源句柄时，查询缓存无法使用。例如，在有些 DBMS 中使用了 **BLOB** 列的时候，缓存结果会为该数据列返回一个资源句柄。

有些缓存存储器有大小限制。例如，**memcache** 限制每条数据最大为 **1MB**。因此，如果查询结果的大小超出了该限制，则会导致缓存失败。

片段缓存

片段缓存指的是缓存页面内容中的某个片段。例如，一个页面显示了逐年销售额的摘要表格，可以把表格缓存下来，以消除每次请求都要重新生成表格的耗时。片段缓存是基于[数据缓存](#)实现的。

在[视图](#)中使用以下结构启用片段缓存：

```
if ($this->beginCache($id)) {  
  
    // ... 在此生成内容 ...  
  
    $this->endCache();  
}
```

调用 `[[yii\base\View::beginCache()|beginCache()]]` 和 `[[yii\base\View::endCache()|endCache()]]` 方法包裹内容生成逻辑。如果缓存中存在该内容，`[[yii\base\View::beginCache()|beginCache()]]` 方法将渲染内容并返回 **false**，因此将跳过内容生成逻辑。否则，内容生成逻辑被执行，一直执行到 `[[yii\base\View::endCache()|endCache()]]` 时，生成的内容将被捕获并存储在缓存中。

和[数据缓存](#)一样，每个片段缓存也需要全局唯一的 **\$id** 标记。

缓存选项

如果要为片段缓存指定额外配置项，请通过向 `[[yii\base\View::beginCache()|beginCache()]]` 方法第二个参数传递配置数组。在框架内部，该数组将被用来配置一个 `[[yii\widget\FragmentCache]]` 小部件用以实现片段缓存功能。

过期时间（duration）

或许片段缓存中最常用的一个配置选项就是 `[[yii\widgets\FragmentCache::duration|duration]]` 了。它指定了内容被缓存的秒数。以下代码缓存内容最多一小时：

```
if ($this->beginCache($id, ['duration' => 3600])) {
```

```
// ... 在此生成内容 ...

$this->endCache();
}
```

如果该选项未设置，则默认为 `0`，永不过期。

依赖

和[数据缓存](#)一样，片段缓存的内容一样可以设置缓存依赖。例如一段被缓存的文章，是否重新缓存取决于它是否被修改过。

通过设置 `[[yii\widgets\FragmentCache::dependency|dependency]]` 选项来指定依赖，该选项的值可以是一个 `[[yii\caching\Dependency]]` 类的派生类，也可以是创建缓存对象的配置数组。以下代码指定了一个片段缓存，它依赖于 `update_at` 字段是否被更改过的。

```
$dependency = [

    'class' => 'yii\caching\DbDependency',

    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

变化

缓存的内容可能需要根据一些参数的更改而变化。例如一个 **Web** 应用支持多语言，同一段视图代码也许需要生成多个语言的内容。因此可以设置缓存根据应用当前语言而变化。

通过设置 `[[yii\widgets\FragmentCache::variations|variations]]` 选项来指定变化，该选项的值应该是一个标量，每个标量代表不同的变化系数。例如设置缓存根据当前语言而变化可以用以下代码：

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {
```

```
// ... 在此生成内容 ...

$this->endCache();
}
```

开关

有时你可能只想在特定条件下开启片段缓存。例如，一个显示表单的页面，可能只需要在初次请求时缓存表单（通过 **GET** 请求）。随后请求所显示（通过 **POST** 请求）的表单不该使用缓存，因为此时表单中可能包含用户输入内容。鉴于此种情况，可以使用

`[[yii\widgets\FragmentCache::enabled|enabled]]` 选项来指定缓存开关，如下所示：

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

缓存嵌套

片段缓存可以被嵌套使用。一个片段缓存可以被另一个包裹。例如，评论被缓存在里层，同时整个评论的片段又被缓存在外层的文章中。以下代码展示了片段缓存的嵌套使用：

```
if ($this->beginCache($id1)) {

    // ...在此生成内容...

    if ($this->beginCache($id2, $options2)) {

        // ...在此生成内容...

        $this->endCache();
    }
}
```

```
// ...在此生成内容...

$this->endCache();
}
```

可以为嵌套的缓存设置不同的配置项。例如，内层缓存和外层缓存使用不同的过期时间。甚至当外层缓存的数据过期失效了，内层缓存仍然可能提供有效的片段缓存数据。但是，反之则不然。如果外层片段缓存没有过期而被视为有效，此时即使内层片段缓存已经失效，它也将继续提供同样的缓存副本。因此，你必须谨慎处理缓存嵌套中的过期时间和依赖，否则外层的片段很有可能返回的是不符合你预期的失效数据。

译注：外层的失效时间应该短于内层，外层的依赖条件应该低于内层，以确保最小的片段，返回的是最新的数据。

动态内容

使用片段缓存时，可能会遇到一大段较为静态的内容中有少许动态内容的情况。例如，一个显示着菜单栏和当前用户名的页面头部。还有一种可能是缓存的内容可能包含每次请求都需要执行的 **PHP** 代码（例如注册资源包的代码）。这两个问题都可以使用**动态内容**功能解决。

动态内容的意思是这部分输出的内容不该被缓存，即便是它被包裹在片段缓存中。为了使内容保持动态，每次请求都执行 **PHP** 代码生成，即使这些代码已经被缓存了。

可以在片段缓存中调用 `[[yii\base\View::renderDynamic()]]` 去插入动态内容，如下所示：

```
if ($this->beginCache($id1)) {

    // ...在此生成内容...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // ...在此生成内容...

    $this->endCache();
}
```

`[[yii\base\View::renderDynamic()]]renderDynamic()` 方法接受一段 **PHP** 代码作为参数。代码的返回值被看作是动态内容。这段代码将在每次请求时都执行，无论其外层的片段缓存是否被存储。

页面缓存

页面缓存指的是在服务器端缓存整个页面的内容。随后当同一个页面被请求时，内容将从缓存中取出，而不是重新生成。

页面缓存由 `[[yii\filters\PageCache]]` 类提供支持，该类是一个[过滤器](#)。它可以像这样在控制器类中使用：

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index'],
            'duration' => 60,
            'variations' => [
                \Yii::$app->language,
            ],
            'dependency' => [
                'class' => 'yii\caching\DbDependency',
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
        ],
    ];
}
```

上述代码表示页面缓存只在 `index` 操作时启用，页面内容最多被缓存 `60` 秒，会随着当前应用的语言更改而变化。如果文章总数发生变化则缓存的页面会失效。

如你所见，页面缓存和[片段缓存](#)极其相似。它们都支持 `duration`，`dependencies`，`variations` 和 `enabled` 配置选项。它们的主要区别是页面缓存是由[过滤器](#)实现，而片段缓存则是一个[小部件](#)。

你可以在使用页面缓存的同时，使用[片段缓存](#)和[动态内容](#)。

HTTP 缓存

除了前面章节讲到的服务器端缓存外，Web 应用还可以利用客户端缓存去节省相同页面内容的生成和传输时间。

通过配置 `[[yii\filters\HttpCache]]` 过滤器，控制器操作渲染的内容就能缓存在客户端。

`[[yii\filters\HttpCache|HttpCache]]` 过滤器仅对 **GET** 和 **HEAD** 请求生效，它能为这些请求设置三种与缓存有关的 HTTP 头。

- `[[yii\filters\HttpCache::lastModified|Last-Modified]]`
- `[[yii\filters\HttpCache::etagSeed|Etag]]`
- `[[yii\filters\HttpCache::cacheControlHeader|Cache-Control]]`

Last-Modified 头

Last-Modified 头使用时间戳标明页面自上次客户端缓存后是否被修改过。

通过配置 `[[yii\filters\HttpCache::lastModified]]` 属性向客户端发送 **Last-Modified** 头。该属性的值应该为 PHP callable 类型，返回的是页面修改时的 Unix 时间戳。该 callable 的参数和返回值应该如下：

```
/**
 * @param Action $action 当前处理的操作对象
 * @param array $params “params” 属性的值
 * @return integer 页面修改时的 Unix 时间戳
 */
function ($action, $params)
```

以下是使用 **Last-Modified** 头的示例：

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
```

```

        return $q->from('post')->max('updated_at');

    },

],

];

}

```

上述代码表明 HTTP 缓存只在 **index** 操作时启用。它会基于页面最后修改时间生成一个 **Last-Modified HTTP** 头。当浏览器第一次访问 **index** 页时，服务器将会生成页面并发送至客户端浏览器。之后客户端浏览器在页面没被修改期间访问该页，服务器将不会重新生成页面，浏览器会使用之前客户端缓存下来的内容。因此服务端渲染和内容传输都将省去。

ETag 头

“Entity Tag”（实体标签，简称 **ETag**）使用一个哈希值表示页面内容。如果页面被修改过，哈希值也会随之改变。通过对比客户端的哈希值和服务器端生成的哈希值，浏览器就能判断页面是否被修改过，进而决定是否应该重新传输内容。

通过配置 `[[yii\filters\HttpCache::etagSeed]]` 属性向客户端发送 **ETag** 头。该属性的值应该为 **PHP callable** 类型，返回的是一段种子字符用来生成 **ETag** 哈希值。该 **callable** 的参数和返回值应该如下：

```

/**
 * @param Action $action 当前处理的操作对象
 * @param array $params “params” 属性的值
 * @return string 一段种子字符用来生成 ETag 哈希值
 */
function ($action, $params)

```

以下是使用 **ETag** 头的示例：

```

public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',

            'only' => ['view'],

            'etagSeed' => function ($action, $params) {

```

```

        $post = $this->findModel(\Yii::$app->request->get('id'));

        return serialize([$post->title, $post->content]);

    },

    ],

    ];

}

```

上述代码表明 HTTP 缓存只在 **view** 操作时启用。它会基于用户请求的标题和内容生成一个 **ETag** HTTP 头。当浏览器第一次访问 **view** 页时，服务器将会生成页面并发送至客户端浏览器。之后客户端浏览器标题和内容没被修改在期间访问该页，服务器将不会重新生成页面，浏览器会使用之前客户端缓存下来的内容。因此服务端渲染和内容传输都将省去。

ETag 相比 **Last-Modified** 能实现更复杂和更精确的缓存策略。例如，当站点切换到另一个主题时可以使 **ETag** 失效。

复杂的 **Etag** 生成种子可能会违背使用 **HttpCache** 的初衷而引起不必要的性能开销，因为响应每一次请求都需要重新计算 **Etag**。请试着找出一个最简单的表达式去触发 **Etag** 失效。

注意：为了遵循 **RFC 7232 (HTTP 1.1 协议)**，如果同时配置了 **ETag** 和 **Last-Modified** 头，**HttpCache** 将会同时发送它们。并且如果客户端同时发送 **If-None-Match** 头和 **If-Modified-Since** 头，则只有前者会被接受。

Cache-Control 头

Cache-Control 头指定了页面的常规缓存策略。可以通过配置 `[[yii\filters\HttpCache::cacheControlHeader]]` 属性发送相应的头信息。默认发送以下头：

```
Cache-Control: public, max-age=3600
```

会话缓存限制器

当页面使 **session** 时，PHP 将会按照 **PHP.INI** 中所设置的 **session.cache_limiter** 值自动发送一些缓存相关的 HTTP 头。这些 HTTP 头有可能会干扰你原本设置的 **HttpCache** 或让其失效。为了避免此问题，默认情况下 **HttpCache** 禁止自动发送这些头。想改变这一行为，可以配置

`[[yii\filters\HttpCache::sessionCacheLimiter]]` 属性。该属性接受一个字符串值，包括 **public**，**private**，**private_no_expire**，和 **nocache**。请参考 **PHP 手册中的缓存限制器** 了解这些值的含义。

SEO 影响

搜索引擎趋向于遵循站点的缓存头。因为一些爬虫的抓取频率有限制，启用缓存头可以减少重复请

求数量，增加爬虫抓取效率（译者：大意如此，但搜索引擎的排名规则不了解，好的缓存策略应该是可以为用户体验加分的）。

快 RESTful Web 服务：

快速入门

Yii 提供了一整套用来简化实现 RESTful 风格的 Web Service 服务的 API。特别是，Yii 支持以下关于 RESTful 风格的 API：

- 支持 [Active Record](#) 类的通用 API 的快速原型
- 涉及的响应格式（在默认情况下支持 JSON 和 XML）
- 支持可选输出字段的定制对象序列化
- 适当的格式的数据采集和验证错误
- 支持 [HATEOAS](#)
- 有适当 HTTP 动词检查的高效的路由
- 内置 [OPTIONS](#) 和 [HEAD](#) 动词的支持
- 认证和授权
- 数据缓存和 HTTP 缓存
- 速率限制

如下， 我们用一个例子来说明如何用最少的编码来建立一套 RESTful 风格的 API。

假设你想通过 RESTful 风格的 API 来展示用户数据。用户数据被存储在用户 DB 表， 你已经创建了 `[[yii\db\ActiveRecord|ActiveRecord]]` 类 `app\models\User` 来访问该用户数据。

创建一个控制器

首先，创建一个控制器类 `app\controllers\UserController` 如下，

```
namespace app\controllers;

use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

```
}
```

控制器类扩展自 `[[yii\rest\ActiveController]]`。通过指定 `[[yii\rest\ActiveController::modelClass|modelClass]]` 作为 `app\models\User`，控制器就能知道使用哪个模型去获取和处理数据。

配置 URL 规则

然后，修改有关在应用程序配置的 `urlManager` 组件的配置：

```
'urlManager' => [  
    'enablePrettyUrl' => true,  
    'enableStrictParsing' => true,  
    'showScriptName' => false,  
    'rules' => [  
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],  
    ],  
]
```

上面的配置主要是为 `user` 控制器增加一个 URL 规则。这样，用户的数据就能通过美化的 URL 和有意义的 http 动词进行访问和操作。

尝试

随着以上所做的最小的努力，你已经完成了创建用于访问用户数据的 RESTful 风格的 API。你所创建的 API 包括：

- `GET /users`: 逐页列出所有用户
- `HEAD /users`: 显示用户列表的概要信息
- `POST /users`: 创建一个新用户
- `GET /users/123`: 返回用户 123 的详细信息
- `HEAD /users/123`: 显示用户 123 的概述信息
- `PATCH /users/123` and `PUT /users/123`: 更新用户 123
- `DELETE /users/123`: 删除用户 123
- `OPTIONS /users`: 显示关于末端 `/users` 支持的动词
- `OPTIONS /users/123`: 显示有关末端 `/users/123` 支持的动词

补充：Yii 将在末端使用的控制器的名称自动变为复数。（译注：个人感觉

这里应该变为注意)

你可以访问你的 API 用 `curl` 命令如下,

```
$ curl -i -H "Accept:application/json" "http://localhost/users"
```

HTTP/1.1 200 OK

Date: Sun, 02 Mar 2014 05:31:43 GMT

Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y

X-Powered-By: PHP/5.4.20

X-Pagination-Total-Count: 1000

X-Pagination-Page-Count: 50

X-Pagination-Current-Page: 1

X-Pagination-Per-Page: 20

Link: <http://localhost/users?page=1>; rel=self,
<http://localhost/users?page=2>; rel=next,
<http://localhost/users?page=50>; rel=last

Transfer-Encoding: chunked

Content-Type: application/json; charset=UTF-8

```
[  
  {  
    "id": 1,  
    ...  
  },  
  {  
    "id": 2,  
    ...  
  },  
  ...  
]
```

试着改变可接受的内容类型为 **application/xml**，你会看到结果以 **XML** 格式返回：

```
$ curl -i -H "Accept:application/xml" "http://localhost/users"
```

HTTP/1.1 200 OK

Date: Sun, 02 Mar 2014 05:31:43 GMT

Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y

X-Powered-By: PHP/5.4.20

X-Pagination-Total-Count: 1000

X-Pagination-Page-Count: 50

X-Pagination-Current-Page: 1

X-Pagination-Per-Page: 20

Link: <http://localhost/users?page=1>; rel=self,
 <http://localhost/users?page=2>; rel=next,
 <http://localhost/users?page=50>; rel=last

Transfer-Encoding: chunked

Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<response>
```

```
  <item>
```

```
    <id>1</id>
```

```
    ...
```

```
  </item>
```

```
  <item>
```

```
    <id>2</id>
```

```
    ...
```

```
  </item>
```

```
  ...
```

```
</response>
```


技巧：你还可以通过 Web 浏览器中输入 URL `http://localhost/users` 来访问你的 API。尽管如此，你可能需要一些浏览器插件来发送特定的 `headers` 请求。

如你所见，在 `headers` 响应，有关于总数，页数的信息，等等。还有一些链接，让你导航到其他页面的数据。例如：`http://localhost/users?page=2` 会给你的用户数据的下一个页面。

使用 `fields` 和 `expand` 参数，你也可以指定哪些字段应该包含在结果内。例如：URL `http://localhost/users?fields=id,email` 将只返回 `id` 和 `email` 字段。

补充：你可能已经注意到了 `http://localhost/users` 的结果包括一些敏感字段，例如 `password_hash`, `auth_key` 你肯定不希望这些出现在你的 API 结果中。你应该在 [响应格式](#) 部分中过滤掉这些字段。

总结

使用 Yii 框架的 RESTful 风格的 API，在控制器的操作中实现 API 末端，使用 控制器来组织末端接口为一个单一的资源类型。

从 `[[yii\base\Model]]` 类扩展的资源被表示为数据模型。如果你在使用（关系或非关系）数据库，推荐你使用 `[[yii\db\ActiveRecord|ActiveRecord]]` 来表示资源。

你可以使用 `[[yii\rest\UrlRule]]` 简化路由到你的 API 末端。

为了方便维护你的 WEB 前端和后端，建议你开发接口作为一个单独的应用程序，虽然这不是必须的。

资源 Resources

RESTful APIs are all about accessing and manipulating resources. You may view resources as `models` in the MVC paradigm.

While there is no restriction in how to represent a resource, in Yii you usually would represent resources in terms of objects of `[[yii\base\Model]]` or its child classes (e.g. `[[yii\db\ActiveRecord]]`), for the following reasons:

- `[[yii\base\Model]]` implements the `[[yii\base\Arrayable]]` interface, which allows you to customize how you want to expose resource data through RESTful APIs.
- `[[yii\base\Model]]` supports `input validation`, which is useful if your RESTful APIs need to support data input.
- `[[yii\db\ActiveRecord]]` provides powerful DB data access and manipulation support, which makes it a perfect fit if your resource data is stored in databases.

In this section, we will mainly describe how a resource class extending from `[[yii\base\Model]]` (or its child classes) can specify what data may be returned via RESTful APIs. If the resource class does not extend from `[[yii\base\Model]]`, then all its public member variables will be returned.

Fields

When including a resource in a RESTful API response, the resource needs to be serialized into a string. Yii breaks this process into two steps. First, the resource is converted into an array by `[[yii\rest\Serializer]]`. Second, the array is serialized into a string in a requested format (e.g. JSON, XML) by `[[yii\web\ResponseFormatterInterface|response formatters]]`. The first step is what you should mainly focus when developing a resource class.

By overriding `[[yii\base\Model::fields()|fields()]]` and/or `[[yii\base\Model::extraFields()|extraFields()]]`, you may specify what data, called `fields`, in the resource can be put into its array representation. The difference between these two methods is that the former specifies the default set of fields which should be included in the array representation, while the latter specifies additional fields which may be included in the array if an end user requests for them via the **expand** query parameter. For example,

```
// returns all fields as declared in fields()
http://localhost/users

// only returns field id and email, provided they are declared in fields()
http://localhost/users?fields=id,email

// returns all fields in fields() and field profile if it is in extraFields()
http://localhost/users?expand=profile

// only returns field id, email and profile, provided they are in fields() and extraFields()
http://localhost/users?fields=id,email&expand=profile
```

Overriding **fields()**

By default, `[[yii\base\Model::fields()]]` returns all model attributes as fields, while `[[yii\db\ActiveRecord::fields()]]` only returns the attributes which have been populated from DB.

You can override **fields()** to add, remove, rename or redefine fields. The return value of **fields()** should be an array. The array keys are the field names, and the array values are the corresponding field definitions which can be either property/attribute names or anonymous functions returning the corresponding field values. In the special case when a field name is the same as its defining attribute name, you can omit the array key. For example,

```
// explicitly list every field, best used when you want to make sure the changes
// in your DB table or model attributes do not cause your field changes (to keep API backward compatibility).
```

```

public function fields()
{
    return [

        // field name is the same as the attribute name

        'id',

        // field name is "email", the corresponding attribute name is "email_address"

        'email' => 'email_address',

        // field name is "name", its value is defined by a PHP callback

        'name' => function () {

            return $this->first_name . ' ' . $this->last_name;

        },

    ];
}

// filter out some fields, best used when you want to inherit the parent implementation
// and blacklist some sensitive fields.

public function fields()
{
    $fields = parent::fields();

    // remove fields that contain sensitive information

    unset($fields['auth_key'], $fields['password_hash'], $fields['password_reset_token']);

    return $fields;
}

```

Warning: Because by default all attributes of a model will be included in the API result, you should examine your data to make sure they do not contain sensitive information. If there is such information, you should override `fields()` to filter them out. In the above example, we choose to filter

out `auth_key`, `password_hash` and `password_reset_token`.

Overriding `extraFields()`

By default, `[[yii\base\Model::extraFields()]]` returns nothing, while `[[yii\db\ActiveRecord::extraFields()]]` returns the names of the relations that have been populated from DB.

The return data format of `extraFields()` is the same as that of `fields()`. Usually, `extraFields()` is mainly used to specify fields whose values are objects. For example, given the following field declaration,

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

the request with `http://localhost/users?fields=id,email&expand=profile` may return the following JSON data:

```
[
    {
        "id": 100,
        "email": "100@example.com",
        "profile": {
            "id": 100,
            "age": 30,
        }
    },
    ...
]
```

Links

[HATEOAS](#), an abbreviation for Hypermedia as the Engine of Application State, promotes that RESTful APIs should return information that allow clients to discover actions supported for the returned resources. The key of HATEOAS is to return a set of hyperlinks with relation information when resource data are served by the APIs.

Your resource classes may support HATEOAS by implementing the `[[yii\web\Linkable]]` interface. The interface contains a single method `[[yii\web\Linkable::getLinks()|getLinks()]]` which should return a list of `[[yii\web\Link|links]]`. Typically, you should return at least the **self** link representing the URL to the resource object itself. For example,

```
use yii\db\ActiveRecord;

use yii\web\Link;

use yii\web\Linkable;

use yii\helpers\Url;

class User extends ActiveRecord implements Linkable
{
    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id], true),
        ];
    }
}
```

When a **User** object is returned in a response, it will contain a **_links** element representing the links related to the user, for example,

```
{
    "id": 100,
    "email": "user@example.com",
    // ...
    "_links" => [
        "self": "https://example.com/users/100"
```

```
]
}
```

Collections

Resource objects can be grouped into collections. Each collection contains a list of resource objects of the same type.

While collections can be represented as arrays, it is usually more desirable to represent them as [data providers](#). This is because data providers support sorting and pagination of resources, which is a commonly needed feature for RESTful APIs returning collections. For example, the following action returns a data provider about the post resources:

```
namespace app\controllers;

use yii\rest\Controller;
use yii\data\ActiveDataProvider;
use app\models\Post;

class PostController extends Controller
{
    public function actionIndex()
    {
        return new ActiveDataProvider([
            'query' => Post::find(),
        ]);
    }
}
```

When a data provider is being sent in a RESTful API response, `[[yii\rest\Serializer]]` will take out the current page of resources and serialize them as an array of resource objects. Additionally, `[[yii\rest\Serializer]]` will also include the pagination information by the following HTTP headers:

- **X-Pagination-Total-Count**: The total number of resources;
- **X-Pagination-Page-Count**: The number of pages;
- **X-Pagination-Current-Page**: The current page (1-based);

- **X-Pagination-Per-Page**: The number of resources in each page;
- **Link**: A set of navigational links allowing client to traverse the resources page by page.

An example may be found in the [Quick Start](#) section.

路由

随着资源和控制器类准备，您可以使用 URL 如 <http://localhost/index.php?r=user/create> 访问资源，类似于你可以用正常的 Web 应用程序做法。

在实践中，你通常要用美观的 URL 并采取有优势的 HTTP 动词。例如，请求 **POST /users** 意味着访问 **user/create** 动作。这可以很容易地通过配置 **urlManager** 应用程序组件来完成 如下所示：

```
'urlManager' => [  
  
    'enablePrettyUrl' => true,  
  
    'enableStrictParsing' => true,  
  
    'showScriptName' => false,  
  
    'rules' => [  
  
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],  
  
    ],  
  
]
```

相比于 URL 管理的 Web 应用程序，上述主要的新东西是通过 RESTful API 请求 `[[yii\rest\UrlRule]]`。这个特殊的 URL 规则类将会 建立一整套子 URL 规则来支持路由和 URL 创建的指定的控制器。例如，上面的代码中是大致按照下面的规则：

```
[  
  
    'PUT,PATCH users/<id>' => 'user/update',  
  
    'DELETE users/<id>' => 'user/delete',  
  
    'GET,HEAD users/<id>' => 'user/view',  
  
    'POST users' => 'user/create',  
  
    'GET,HEAD users' => 'user/index',  
  
    'users/<id>' => 'user/options',  
  
    'users' => 'user/options',  
  
]
```

该规则支持下面的 API 末端:

- **GET /users**: 逐页列出所有用户;
- **HEAD /users**: 显示用户列表的概要信息;
- **POST /users**: 创建一个新用户;
- **GET /users/123**: 返回用户为 123 的详细信息;
- **HEAD /users/123**: 显示用户 123 的概述信息;
- **PATCH /users/123** and **PUT /users/123**: 更新用户 123;
- **DELETE /users/123**: 删除用户 123;
- **OPTIONS /users**: 显示关于末端 **/users** 支持的动词;
- **OPTIONS /users/123**: 显示有关末端 **/users/123** 支持的动词。

您可以通过配置 **only** 和 **except** 选项来明确列出哪些行为支持, 哪些行为禁用。例如,

```
[  
    'class' => 'yii\rest\UrlRule',  
  
    'controller' => 'user',  
  
    'except' => ['delete', 'create', 'update'],  
],
```

您也可以通过配置 **patterns** 或 **extraPatterns** 重新定义现有的模式或添加此规则支持的新模式。例如, 通过末端 **GET /users/search** 可以支持新行为 **search**, 按照如下配置 **extraPatterns** 选项,

```
[  
    'class' => 'yii\rest\UrlRule',  
  
    'controller' => 'user',  
  
    'extraPatterns' => [  
        'GET search' => 'search',  
    ],  
],
```

您可能已经注意到控制器 ID **user** 以复数形式出现在 **users** 末端。这是因为 `[[yii\rest\UrlRule]]` 能够为他们使用的末端全自动复数化控制器 ID。您可以通过设置 `[[yii\rest\UrlRule::pluralize]]` 为 **false** 来禁用此行为, 如果您想使用一些特殊的名字您可以通过配置 `[[yii\rest\UrlRule::controller]]` 属性。

格式化响应(响应格式)

当处理一个 RESTful API 请求时, 一个应用程序通常需要如下步骤 来处理响应格式:

1.确定可能影响响应格式的各种因素， 例如媒介类型， 语言， 版本， 等等。 这个过程也被称为 [content negotiation](#)。

2.资源对象转换为数组， 如在 [Resources](#) 部分中所描述的。 通过 `[[yii\rest\Serializer]]` 来完成。

3.通过内容协商步骤将数组转换成字符串。

`[[yii\web\ResponseFormatterInterface|response formatters]]` 通过 `[[yii\web\Response::formatters|response]]` 应用程序组件来注册完成。

内容协商

Yii 提供了通过 `[[yii\filters\ContentNegotiator]]` 过滤器支持内容协商。RESTful API 基于 控制器类 `[[yii\rest\Controller]]` 在 [contentNegotiator](#) 下配备这个过滤器。 文件管理器提供了涉及的响应格式和语言。 例如， 如果一个 RESTful API 请求中包含以下 header，

```
Accept: application/json; q=1.0, */*; q=0.1
```

将会得到 JSON 格式的响应， 如下：

```
$ curl -i -H "Accept: application/json; q=1.0, */*; q=0.1" "http://localhost/users"
```

```
HTTP/1.1 200 OK
```

```
Date: Sun, 02 Mar 2014 05:31:43 GMT
```

```
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
```

```
X-Powered-By: PHP/5.4.20
```

```
X-Pagination-Total-Count: 1000
```

```
X-Pagination-Page-Count: 50
```

```
X-Pagination-Current-Page: 1
```

```
X-Pagination-Per-Page: 20
```

```
Link: <http://localhost/users?page=1>; rel=self,  
      <http://localhost/users?page=2>; rel=next,  
      <http://localhost/users?page=50>; rel=last
```

```
Transfer-Encoding: chunked
```

```
Content-Type: application/json; charset=UTF-8
```

```
[
```

```
{
```

```

        "id": 1,

        ...

    },

    {

        "id": 2,

        ...

    },

    ...

]

```

幕后，执行一个 RESTful API 控制器动作之前，`[[yii\filters\ContentNegotiator]]` filter 将检查 `Accept` HTTP header 在请求时和配置 `[[yii\web\Response::format|response format]]` 为 `'json'`。之后的动作被执行并返回得到的资源对象或集合，`[[yii\rest\Serializer]]` 将结果转换成一个数组。最后，`[[yii\web\JsonResponseFormatter]]` 该数组将序列化为 JSON 字符串，并将其包括在响应主体。

默认，RESTful APIs 同时支持 JSON 和 XML 格式。为了支持新的格式，你应该在 `contentNegotiator` 过滤器中配置 `[[yii\filters\ContentNegotiator::formats|formats]]` 属性，类似如下 API 控制器类：

```

use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();

    $behaviors['contentNegotiator']['formats']['text/html'] = Response::FORMAT_HTML;

    return $behaviors;
}

```

`formats` 属性的 keys 支持 MIME 类型，而 values 必须在 `[[yii\web\Response::formatters]]` 中支持被响应格式名称。

数据序列化

正如我们上面所描述的，`[[yii\rest\Serializer]]` 负责转换资源的中间件 对象或集合到数组。它将对象 `[[yii\base\ArrayableInterface]]` 作为 `[[yii\data\DataProviderInterface]]`。前者主要由资源对象实现，而 后者是资源集合。

你可以通过设置 `[[yii\rest\Controller::serializer]]` 属性和一个配置数组。例如，有时你可能想通过直接在响应主体内包含分页信息来简化客户端的开发工作。这样做，按照如下规则配置 `[[yii\rest\Serializer::collectionEnvelope]]` 属性：

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';

    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

那么你的请求可能会得到的响应如下 <http://localhost/users>:

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "items": [
```

```
{
  {
    "id": 1,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
],
"_links": {
  "self": "http://localhost/users?page=1",
  "next": "http://localhost/users?page=2",
  "last": "http://localhost/users?page=50"
},
"_meta": {
  "totalCount": 1000,
  "pageCount": 50,
  "currentPage": 1,
  "perPage": 20
}
}
```

授权验证 **Authentication**

Unlike Web applications, RESTful APIs are usually stateless, which means sessions or cookies should not be used. Therefore, each request should come with some sort of

authentication credentials because the user authentication status may not be maintained by sessions or cookies. A common practice is to send a secret access token with each request to authenticate the user. Since an access token can be used to uniquely identify and authenticate a user, **API requests should always be sent via HTTPS to prevent from man-in-the-middle (MitM) attacks.**

There are different ways to send an access token:

- HTTP Basic Auth**: the access token is sent as the username. This should only be used when an access token can be safely stored on the API consumer side. For example, the API consumer is a program running on a server.
- Query parameter**: the access token is sent as a query parameter in the API URL, e.g., <https://example.com/users?access-token=xxxxxxx>. Because most Web servers will keep query parameters in server logs, this approach should be mainly used to serve **JSONP** requests which cannot use HTTP headers to send access tokens.
- OAuth 2**: the access token is obtained by the consumer from an authorization server and sent to the API server via **HTTP Bearer Tokens**, according to the OAuth2 protocol.

Yii supports all of the above authentication methods. You can also easily create new authentication methods.

To enable authentication for your APIs, do the following steps:

1. Configure the `[[yii\web\User::enableSession|enableSession]]` property of the **user** application component to be false.
2. Specify which authentication methods you plan to use by configuring the **authenticator** behavior in your REST controller classes.
3. Implement `[[yii\web\IdentityInterface::findIdentityByAccessToken()]]` in your `[[yii\web\User::identityClass|user identity class]]`.

Step 1 is not required but is recommended for RESTful APIs which should be stateless. When `[[yii\web\User::enableSession|enableSession]]` is false, the user authentication status will NOT be persisted across requests using sessions. Instead, authentication will be performed for every request, which is accomplished by Step 2 and 3.

Tip: You may configure `[[yii\web\User::enableSession|enableSession]]` of the **user** application component in application configurations if you are developing RESTful APIs in terms of an application. If you develop RESTful APIs as a module, you may put the following line in the module's `init()` method, like the following:

```
parent::init();  
  
\Yii::$app->user->enableSession = false;  
  
}
```

For example, to use HTTP Basic Auth, you may configure **authenticator** as follows,

```
use yii\filters\auth\HttpBasicAuth;
```

```

public function behaviors()
{
    $behaviors = parent::behaviors();

    $behaviors['authenticator'] = [

        'class' => HttpBasicAuth::className(),

    ];

    return $behaviors;
}

```

If you want to support all three authentication methods explained above, you can use **CompositeAuth** like the following,

```

use yii\filters\auth\CompositeAuth;
use yii\filters\auth\HttpBasicAuth;
use yii\filters\auth\HttpBearerAuth;
use yii\filters\auth\QueryParamAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();

    $behaviors['authenticator'] = [

        'class' => CompositeAuth::className(),

        'authMethods' => [

            HttpBasicAuth::className(),

            HttpBearerAuth::className(),

            QueryParamAuth::className(),

        ],

    ];

    return $behaviors;
}

```

Each element in `authMethods` should be an auth method class name or a configuration array.

Implementation of `findIdentityByAccessToken()` is application specific. For example, in simple scenarios when each user can only have one access token, you may store the access token in an `access_token` column in the user table. The method can then be readily implemented in the `User` class as follows,

```
use yii\db\ActiveRecord;

use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }
}
```

After authentication is enabled as described above, for every API request, the requested controller will try to authenticate the user in its `beforeAction()` step.

If authentication succeeds, the controller will perform other checks (such as rate limiting, authorization) and then run the action. The authenticated user identity information can be retrieved via `Yii::$app->user->identity`.

If authentication fails, a response with HTTP status 401 will be sent back together with other appropriate headers (such as a `WWW-Authenticate` header for HTTP Basic Auth).

Authorization

After a user is authenticated, you probably want to check if he or she has the permission to perform the requested action for the requested resource. This process is called authorization which is covered in detail in the [Authorization section](#).

If your controllers extend from `[[yii\rest\ActiveController]]`, you may override the `[[yii\rest\Controller::checkAccess()|checkAccess()]]` method to perform authorization check. The method will be called by the built-in actions provided by `[[yii\rest\ActiveController]]`.

速率限制

为防止滥用，你应该考虑增加速率限制到您的 API。例如，您可以限制每个用户的 API 的使用是在 10

分钟内最多 100 次的 API 调用。如果一个用户同一个时间段内太多的请求被接收，将返回响应状态代码 429 (这意味着过多的请求)。

要启用速率限制, `[[yii\web\User::identityClass|user identity class]]` 应该实现 `[[yii\filters\RateLimitInterface]]`. 这个接口需要实现以下三个方法:

- `getRateLimit()`: 返回允许的请求的最大数目及时间, 例如, `[100, 600]` 表示在 600 秒内最多 100 次的 API 调用。
- `loadAllowance()`: 返回剩余的允许的请求和相应的 UNIX 时间戳数 当最后一次速率限制检查时。
- `saveAllowance()`: 保存允许剩余的请求数和当前的 UNIX 时间戳。

你可以在 `user` 表中使用两列来记录容差和时间戳信息。 `loadAllowance()` 和 `saveAllowance()` 可以通过实现对符合当前身份验证的用户 的这两列值的读和保存。为了提高性能, 你也可以 考虑使用缓存或 NoSQL 存储这些信息。

一旦 `identity` 实现所需的接口, `Yii` 会自动使用 `[[yii\filters\RateLimiter]]` 为 `[[yii\rest\Controller]]` 配置一个行为过滤器来执行速率限制检查。如果速度超出限制 该速率限制器将抛出一个 `[[yii\web\TooManyRequestsHttpException]]`。你可以在你的 REST 控制器类里配置速率限制,

```
public function behaviors()
{
    $behaviors = parent::behaviors();

    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;

    return $behaviors;
}
```

当速率限制被激活, 默认情况下每个响应将包含以下 HTTP 头发送 目前的速率限制信息:

- `X-Rate-Limit-Limit`: 同一个时间段所允许的请求的最大数目;
- `X-Rate-Limit-Remaining`: 在当前时间段内剩余的请求的数量;
- `X-Rate-Limit-Reset`: 为了得到最大请求数所等待的秒数。

你可以禁用这些头信息通过配置 `[[yii\filters\RateLimiter::enableRateLimitHeaders]]` 为 `false`, 就像在上面的代码示例所示。

版本(化)

你的 API 应该是版本化的。不像你完全控制在客户端和服务端 Web 应用程序代码, 对于 API, 您通常没有对 API 的客户端代码的控制权。因此, 应该尽可能的保持向后兼容性(BC), 如果一些不能向后兼容的变化必须引入 APIs, 你应该增加版本号。你可以参考 [Semantic Versioning](#) 有关设计的 API 的版本号的详细信息。

关于如何实现 API 版本, 一个常见的做法是在 API 的 URL 中嵌入版本号。例如, `http://example.com/v1/users` 代表 `users` 版本 1 的 API. 另一种 API 版本化的方法最近用的非常多的是

把版本号放入 HTTP 请求头，通常是通过 **Accept** 头， 如下：

```
// 通过参数
```

```
Accept: application/json; version=v1
```

```
// 通过 vendor 的内容类型
```

```
Accept: application/vnd.company.myapp-v1+json
```

这两种方法都有优点和缺点， 而且关于他们也有很多争论。 下面我们描述在一种 **API** 版本混合了这两种方法的一个实用的策略：

- 把每个主要版本的 **API** 实现在一个单独的模块 ID 的主版本号（例如 **v1**, **v2**）。 自然， **API** 的 url 将包含主要的版本号。
- 在每一个主要版本（在相应的模块），使用 **Accept** HTTP 请求头 确定小版本号编写条件代码来响应相应的次要版本。

为每个模块提供一个主要版本， 它应该包括资源类和控制器类 为特定服务版本。 更好的分离代码， 你可以保存一组通用的 基础资源和控制器类， 并用在每个子类版本模块。 在子类中， 实现具体的代码例如 **Model::fields()**。

你的代码可以类似于如下的方法组织起来：

```
api/
```

```
    common/
```

```
        controllers/
```

```
            UserController.php
```

```
            PostController.php
```

```
        models/
```

```
            User.php
```

```
            Post.php
```

```
    modules/
```

```
        v1/
```

```
            controllers/
```

```
                UserController.php
```

```
                PostController.php
```

```
            models/
```

```
                User.php
```

```
    Post.php

v2/

    controllers/

        UserController.php

        PostController.php

    models/

        User.php

        Post.php
```

你的应用程序配置应该这样:

```
return [

    'modules' => [

        'v1' => [

            'basePath' => '@app/modules/v1',

            ],

        'v2' => [

            'basePath' => '@app/modules/v2',

            ],

    ],

    'components' => [

        'urlManager' => [

            'enablePrettyUrl' => true,

            'enableStrictParsing' => true,

            'showScriptName' => false,

            'rules' => [

                ['class' => 'yii\rest\UrlRule', 'controller' => ['v1/user', 'v1/post']],

                ['class' => 'yii\rest\UrlRule', 'controller' => ['v2/user', 'v2/post']],

            ],

        ],

    ],

];
```

```
        ],
    ],
    ],
];
```

因此，<http://example.com/v1/users> 将返回版本 1 的用户列表，而 <http://example.com/v2/users> 将返回版本 2 的用户。

使用模块，将不同版本的代码隔离。通过共用基类和其他类跨模块重用代码也是有可能的。

为了处理次要版本号，可以利用内容协商功能通过 `[[yii\filters\ContentNegotiator|contentNegotiator]]` 提供的行为。`contentNegotiator` 行为可设置 `[[yii\web\Response::acceptParams]]` 属性当它确定支持哪些内容类型时。

例如，如果一个请求通过 `Accept: application/json; version=v1` 被发送，内容协商后，`[[yii\web\Response::acceptParams]]` 将包含值 `['version' => 'v1']`。

基于 `acceptParams` 的版本信息，你可以写条件代码如 actions, resource classes, serializers 等等。

由于次要版本需要保持向后兼容性，希望你的代码不会有太多的版本检查。否则，有机会你可能需要创建一个新的主要版本。

错误处理

处理一个 RESTful API 请求时，如果有一个用户请求错误或服务器发生意外时，你可以简单地抛出一个异常来通知用户出错了。如果你能找出错误的原因（例如，所请求的资源不存在），你应该考虑抛出一个适当的 HTTP 状态代码的异常（例如，`[[yii\web\NotFoundHttpException]]` 意味着一个 404 HTTP 状态代码）。Yii 将通过 HTTP 状态码和文本发送相应的响应。它还将包括在响应主体异常的序列化表示形式。例如，

```
HTTP/1.1 404 Not Found
```

```
Date: Sun, 02 Mar 2014 05:31:43 GMT
```

```
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
```

```
Transfer-Encoding: chunked
```

```
Content-Type: application/json; charset=UTF-8
```

```
{
    "type": "yii\\web\\NotFoundHttpException",
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
```

```
"code": 0,  
  
"status": 404  
}
```

下面的列表总结了 Yii 的 REST 框架的 HTTP 状态代码:

- **200**: OK。一切正常。
- **201**: 响应 **POST** 请求时成功创建一个资源。 **Location** header 包含的 URL 指向新创建的资源。
- **204**: 该请求被成功处理, 响应不包含正文内容 (类似 **DELETE** 请求)。
- **304**: 资源没有被修改。可以使用缓存的版本。
- **400**: 错误的请求。可能通过用户方面的多种原因引起的, 例如在请求体内有无效的 **JSON** 数据, 无效的操作参数, 等等。
- **401**: 验证失败。
- **403**: 已经经过身份验证的用户不允许访问指定的 **API** 末端。
- **404**: 所请求的资源不存在。
- **405**: 不被允许的方法。 请检查 **Allow** header 允许的 **HTTP** 方法。
- **415**: 不支持的媒体类型。 所请求的内容类型或版本号是无效的。
- **422**: 数据验证失败 (例如, 响应一个 **POST** 请求)。 请检查响应体内详细的错误消息。
- **429**: 请求过多。 由于限速请求被拒绝。
- **500**: 内部服务器错误。 这可能是由于内部程序错误引起的。

测试(官方待定)

<http://www.yiichina.com/guide/2/rest-testing>

核心验证器 (Core Validators)

Yii 提供一系列常用的核心验证器, 主要存在于 `yii\validators` 命名空间之下。为了避免使用冗长的类名, 你可以直接用**昵称**来指定相应的核心验证器。比如你可以用 `required` 昵称代指 `[[yii\validators\RequiredValidator]]` 类:

```
public function rules()  
{
```

```
return [  
  
    [['email', 'password'], 'required'],  
  
];  
}
```

`[[yii\validators\Validator::builtInValidators]]` 属性声明了所有被支持的验证器昵称。

下面，我们将详细介绍每一款验证器的主要用法和属性。

[[yii\validators\BooleanValidator|boolean（布尔型）]]

```
[  
  
    // 检查 "selected" 是否为 0 或 1，无视数据类型  
  
    ['selected', 'boolean'],  
  
    // 检查 "deleted" 是否为布尔类型，即 true 或 false  
  
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false, 'strict' => true],  
  
]
```

该验证器检查输入值是否为一个布尔值。

- **trueValue**: 代表**真**的值。默认为 `'1'`。
- **falseValue**: 代表**假**的值。默认为 `'0'`。
- **strict**: 是否要求待测输入必须严格匹配 **trueValue** 或 **falseValue**。默认为 `false`。

注意：因为通过 HTML 表单传递的输入数据都是字符串类型，所以一般情况下你都需要保持 `[[yii\validators\BooleanValidator::strict|strict]]` 属性为假。

[[yii\captcha\CaptchaValidator|captcha（验证码）]]

```
[  
  
    ['verificationCode', 'captcha'],  
  
]
```

该验证器通常配合 `[[yii\captcha\CaptchaAction]]` 以及 `[[yii\captcha\Captcha]]` 使用，以确保某一输入与 `[[yii\captcha\Captcha|CAPTCHA]]` 小部件所显示的验证代码（verification code）相同。

- **caseSensitive**: 对验证代码的比对是否要求大小写敏感。默认为 `false`。
- **captchaAction**: 指向用于渲染 CAPTCHA 图片的 `[[yii\captcha\CaptchaAction|CAPTCHA action]]` 的 [路由](#)。默认为 `'site/captcha'`。
- **skipOnEmpty**: 当输入为空时，是否跳过验证。默认为 `false`，也就是输入值为必需项。

`[[yii\validators\CompareValidator|compare（比较）]]`

```
[  
    // 检查 "password" 特性的值是否与 "password_repeat" 的值相同  
    ['password', 'compare'],  
  
    // 检查年龄是否大于等于 30  
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],  
]
```

该验证器比较两个特定输入值之间的关系是否与 **operator** 属性所指定的相同。

- **compareAttribute**: 用于与原特性相比较的特性名称。当该验证器被用于验证某目标特性时，该属性会默认为目标属性加后缀 **_repeat**。举例来说，若目标特性为 **password**，则该属性默认为 **password_repeat**。
- **compareValue**: 用于与输入值相比较的常量值。当该属性与 **compareAttribute** 属性同时被指定时，该属性优先被使用。
- **operator**: 比较操作符。默认为 **==**，意味着检查输入值是否与 **compareAttribute** 或 **compareValue** 的值相等。该属性支持如下操作符：
 - **==**: 检查两值是否相等。比对为非严格模式。
 - **===**: 检查两值是否全等。比对为严格模式。
 - **!=**: 检查两值是否不等。比对为非严格模式。
 - **!==**: 检查两值是否不全等。比对为严格模式。
 - **>**: 检查待测目标值是否大于给定被测值。
 - **>=**: 检查待测目标值是否大于等于给定被测值。
 - **<**: 检查待测目标值是否小于给定被测值。
 - **<=**: 检查待测目标值是否小于等于给定被测值。

[[yii\validators\DateValidator|date (日期)]]

```
[  
  
    [['from', 'to'], 'date'],  
  
]
```

该验证器检查输入值是否为适当格式的 **date**，**time**，或者 **datetime**。另外，它还可以帮你把输入值转换为一个 **UNIX** 时间戳并保存到

[[yii\validators\DateValidator::timestampAttribute|timestampAttribute]] 属性所指定的特性里。

- **format**: 待测的 日期/时间 格式。请参考 [date_create_from_format\(\)](#) 相关的 [PHP 手册](#) 了解设定格式字符串的更多细节。默认值为 **'Y-m-d'**。
- **timestampAttribute**: 用于保存用输入时间/日期转换出来的 **UNIX** 时间戳的特性。

[[yii\validators\DefaultValueValidator| default (默认值)]]

```
[  
  
    // 若 "age" 为空，则将其设为 null  
  
    ['age', 'default', 'value' => null],  
  
    // 若 "country" 为空，则将其设为 "USA"  
  
    ['country', 'default', 'value' => 'USA'],  
  
    // 若 "from" 和 "to" 为空，则分别给他们分配自今天起，3 天后和 6 天后的日期。  
  
    [['from', 'to'], 'default', 'value' => function ($model, $attribute) {  
        return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6 days'));  
    }],  
  
]
```

该验证器并不进行数据验证。而是，给为空的待测特性分配默认值。

- **value**: 默认值，或一个返回默认值的 **PHP Callable** 对象（即回调函数）。它们会分配给检测为空的待测特性。PHP 回调方法的样式如下：

```
function foo($model, $attribute) {  
  
    // ... 计算 $value ...  
  
    return $value;  
  
}
```

补充：如何判断待测值是否为空，被写在另外一个话题的[处理空输入](#)章节。

[[yii\validators\NumberValidator|double（双精度浮点型）]]

```
[  
  
    // 检查 "salary" 是否为浮点数  
  
    ['salary', 'double'],  
  
]
```

该验证器检查输入值是否为双精度浮点数。他等效于 [number](#) 验证器。

- **max**: 上限值（含界点）。若不设置，则验证器不检查上限。
- **min**: 下限值（含界点）。若不设置，则验证器不检查下限。

[[yii\validators>EmailValidator|email（电子邮件）]]

```
[  
  
    // 检查 "email" 是否为有效的邮箱地址  
  
    ['email', 'email'],  
  
]
```

该验证器检查输入值是否为有效的邮箱地址。

- **allowName**: 检查是否允许带名称的电子邮件地址（e.g. [张三 <John.san@example.com>](#)）。默认为 **false**。
- **checkDNS**: 检查邮箱域名是否存在，且有没有对应的 **A** 或 **MX** 记录。不过要知道，有的时候该项检查可能会因为临时性 **DNS** 故障而失败，哪怕它其实是有效的。默认为 **false**。
- **enableIDN**: 验证过程是否应该考虑 **IDN**（internationalized domain names，国际化域名，也称多语种域名，比如中文域名）。默认为 **false**。要注意但是为使用 **IDN** 验证功能，请先确保安装并开启 **intl** PHP 扩展，不然会导致抛出异常。

[[yii\validators\ExistValidator|exist (存在性)]]

[

// a1 需要在 "a1" 特性所代表的字段内存在

['a1', 'exist'],

// a1 必需存在，但检验的是 a1 的值在字段 a2 中的存在性

['a1', 'exist', 'targetAttribute' => 'a2'],

// a1 和 a2 的值都需要存在，且它们都能收到错误提示

[['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

// a1 和 a2 的值都需要存在，只有 a1 能接收到错误信息

['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

// 通过同时在 a2 和 a3 字段中检查 a2 和 a1 的值来确定 a1 的存在性

['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

// a1 必需存在，若 a1 为数组，则其每个子元素都必须存在。

['a1', 'exist', 'allowArray' => true],

]

该验证器检查输入值是否在某表字段中存在。它只对[活动记录](#)类型的模型类特性起作用，能支持对一个或多过字段的验证。

- **targetClass**: 用于查找输入值的目标 [AR](#) 类。若不设置，则会使用正在进行验证的当前模型类。
- **targetAttribute**: 用于检查输入值存在性的 **targetClass** 的模型特性。
 - 若不设置，它会直接使用待测特性名（整个参数数组的首元素）。
 - 除了指定为字符串以外，你也可以用数组的形式，同时指定多个用于验证的表字段，数组的键和值都是代表字段的特性名，值表示 **targetClass** 的待测数据源字段，而键表示当前模型的待测特性名。

- 若键和值相同，你可以只指定值。（如:['a2'] 就代表 ['a2'=>'a2']）
- filter**: 用于检查输入值存在性必然会进行数据库查询，而该属性为用于进一步筛选该查询的过滤条件。可以为代表额外查询条件的字符串或数组（关于查询条件的格式，请参考 `[[yii\db\Query::where()]]`）；或者样式为 `function ($query)` 的匿名函数，`$query` 参数为你希望在该函数内进行修改的 `[[yii\db\Query|Query]]` 对象。
- allowArray**: 是否允许输入值为数组。默认为 `false`。若该属性为 `true` 且输入值为数组，则数组的每个元素都必须在目标字段中存在。值得注意的是，若用吧 **targetAttribute** 设为多元素数组来验证被测值在多字段中的存在性时，该属性不能设置为 `true`。

译注: **exist** 和 **unique** 验证器的机理和参数都相似，有点像一体两面的阴和阳。

- 他们的区别是 **exist** 要求 **targetAttribute** 键所代表的属性在其值所代表字段中找得到；而 **unique** 正相反，要求键所代表的属性不能在其值所代表字段中被找到。
- 从另一个角度来理解：他们都会在验证的过程中执行数据库查询，查询的条件即为 `where $v=$k` (假设 **targetAttribute** 的其中一对键值对为 `$k => $v`)。 **unique** 要求查询的结果数 `$count==0`，而 **exist** 则要求查询的结果数 `$count>0`
- 最后别忘了，**unique** 验证器不存在 **allowArray** 属性哦。

[[yii\validators\FileValidator|file（文件）]]

```
[
    // 检查 "primaryImage" 是否为 PNG, JPG 或 GIF 格式的上传图片。

    // 文件大小必须小于 1MB

    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'], 'maxSize' => 1024*1024*1024],
]
```

该验证器检查输入值是否为一个有效的上传文件。

- extensions**: 可接受上传的文件扩展名列表。它可以是数组，也可以是用空格或逗号分隔各个扩展名的字符串 (e.g. "gif, jpg")。扩展名大小写不敏感。默认为 `null`，意味着所有扩展名都被接受。
- mimeTypes**: 可接受上传的 MIME 类型列表。它可以是数组，也可以是用空格或逗号分隔各个 MIME 的字符串 (e.g. "image/jpeg, image/png")。Mime 类型名是大小写不敏感的。默认为 `null`，意味着所有 MIME 类型都被接受。
- minSize**: 上传文件所需最少多少 Byte 的大小。默认为 `null`，代表没有下限。
- maxSize**: 上传文件所需最多多少 Byte 的大小。默认为 `null`，代表没有上限。

- **maxFiles**: 给定特性最多能承载多少个文件。默认为 **1**，代表只允许单文件上传。若值大于一，那么输入值必须为包含最多 **maxFiles** 个上传文件元素的数组。
- **checkExtensionByMimeType**: 是否通过文件的 **MIME** 类型来判断其文件扩展。若由 **MIME** 判定的文件扩展与给定文件的扩展不一样，则文件会被认为无效。默认为 **true**，代表执行上述检测。

FileValidator 通常与 `[[yii\web\UploadedFile]]` 共同使用。请参考 [文件上传](#) 章节来了解有关文件上传与上传文件的检验的全部内容。

`[[yii\validators\FilterValidator|filter` (滤镜) `]]`

```
[  
  
    // trim 掉 "username" 和 "email" 输入  
  
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' => true],  
  
    // 标准化 "phone" 输入  
  
    ['phone', 'filter', 'filter' => function ($value) {  
        // 在此处标准化输入的电话号码  
  
        return $value;  
    }],  
  
]
```

该验证器并不进行数据验证。而是，给输入值应用一个滤镜，并在检验过程之后把它赋值回特性变量。

- **filter**: 用于定义滤镜的 **PHP** 回调函数。可以为全局函数名，匿名函数，或其他。该函数的样式必须是 `function ($value) { return $newValue; }`。该属性不能省略，必须设置。
- **skipOnArray**: 是否在输入值为数组时跳过滤镜。默认为 **false**。请注意如果滤镜不能处理数组输入，你就应该把该属性设为 **true**。否则可能会导致 **PHP Error** 的发生。

技巧：如果你只是想要用 **trim** 处理下输入值，你可以直接用 **trim** 验证器的。

`[[yii\validators\ImageValidator|image` (图片) `]]`

```
[
```

```
// 检查 "primaryImage" 是否为适当尺寸的有效图片

['primaryImage', 'image', 'extensions' => 'png, jpg',

    'minWidth' => 100, 'maxWidth' => 1000,

    'minHeight' => 100, 'maxHeight' => 1000,

],

]
```

该验证器检查输入值是否为代表有效的图片文件。它继承自 [file](#) 验证器，并因此继承有其全部属性。除此之外，它还支持以下为图片检验而设的额外属性：

- **minWidth**：图片的最小宽度。默认为 `null`，代表无下限。
- **maxWidth**：图片的最大宽度。默认为 `null`，代表无上限。
- **minHeight**：图片的最小高度。默认为 `null`，代表无下限。
- **maxHeight**：图片的最大高度。默认为 `null`，代表无上限。

[[yii\validators\RangeValidator|in（范围）]]

```
[

    // 检查 "level" 是否为 1、2 或 3 中的一个

    ['level', 'in', 'range' => [1, 2, 3]],

]
```

该验证器检查输入值是否存在于给定列表的范围之中。

- **range**：用于检查输入值的给定值列表。
- **strict**：输入值与给定值直接的比较是否为严格模式（也就是类型与值都要相同，即全等）。默认为 `false`。
- **not**：是否对验证的结果取反。默认为 `false`。当该属性被设置为 `true`，验证器检查输入值是否不在给定列表内。
- **allowArray**：是否接受输入值为数组。当该值为 `true` 且输入值为数组时，数组内的每一个元素都必须在给定列表内存在，否则返回验证失败。

[[yii\validators\NumberValidator|integer（整数）]]

```
[  
    // 检查 "age" 是否为整数  
    ['age', 'integer'],  
]
```

该验证器检查输入值是否为整形。

- **max**: 上限值（含界点）。若不设置，则验证器不检查上限。
- **min**: 下限值（含界点）。若不设置，则验证器不检查下限。

[[yii\validators\RegularExpressionValidator|match（正则表达式）]]

```
[  
    // 检查 "username" 是否由字母开头，且只包含单词字符  
    ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']  
]
```

该验证器检查输入值是否匹配指定正则表达式。

- **pattern**: 用于检测输入值的正则表达式。该属性是必须的，若不设置则会抛出异常。
- **not**: 是否对验证的结果取反。默认为 **false**，代表输入值匹配正则表达式时验证成功。如果设为 **true**，则输入值不匹配正则时返回匹配成功。

[[yii\validators\NumberValidator|number（数字）]]

```
[  
    // 检查 "salary" 是否为数字  
    ['salary', 'number'],  
]
```

该验证器检查输入值是否为数字。他等效于 **double** 验证器。

- **max**: 上限值（含界点）。若不设置，则验证器不检查上限。
- **min**: 下限值（含界点）。若不设置，则验证器不检查下限。

[[yii\validators\RequiredValidator|required (必填)]]

```
[  
    // 检查 "username" 与 "password" 是否为空  
    [['username', 'password'], 'required'],  
]
```

该验证器检查输入值是否为空，还是已经提供了。

- **requiredValue**: 所期望的输入值。若没设置，意味着输入不能为空。
- **strict**: 检查输入值时是否检查类型。默认为 **false**。当没有设置 **requiredValue** 属性时，若该属性为 **true**，验证器会检查输入值是否严格为 **null**；若该属性设为 **false**，该验证器会用一个更加宽松的规则检验输入值是否为空。

当设置了 **requiredValue** 属性时，若该属性为 **true**，输入值与 **requiredValue** 的比对会同时检查数据类型。

补充：如何判断待测值是否为空，被写在另外一个话题的[处理空输入](#)章节。

[[yii\validators\SafeValidator|safe (安全)]]

```
[  
    // 标记 "description" 为安全特性  
    ['description', 'safe'],  
]
```

该验证器并不进行数据验证。而是把一个特性标记为[安全特性](#)。

[[yii\validators\StringValidator|string (字符串)]]

```
[
```

```
// 检查 "username" 是否为长度 4 到 24 之间的字符串
```

```
['username', 'string', 'length' => [4, 24]],
```

```
]
```

该验证器检查输入值是否为特定长度的字符串。并检查特性的值是否为某个特定长度。

- **length**: 指定待测输入字符串的长度限制。该属性可以被指定为以下格式之一:
 - 证书: the exact length that the string should be of;
 - 单元素数组: 代表输入字符串的最小长度 (e.g. **[8]**)。这会重写 **min** 属性。
 - 包含两个元素的数组: 代表输入字符串的最小和最大长度(e.g. **[8, 128]**)。这会同时重写 **min** 和 **max** 属性。
- **min**: 输入字符串的最小长度。若不设置, 则代表不设下限。
- **max**: 输入字符串的最大长度。若不设置, 则代表不设上限。
- **encoding**: 待测字符串的编码方式。若不设置, 则使用应用自身的 `[[yii\base\Application::charset|charset]]` 属性值, 该值默认为 **UTF-8**。

[[yii\validators\FilterValidator|trim (译为修剪/裁边)]]

```
[
```

```
// trim 掉 "username" 和 "email" 两侧的多余空格
```

```
[['username', 'email'], 'trim'],
```

```
]
```

该验证器并不进行数据验证。而是, **trim** 掉输入值两侧的多余空格。注意若该输入值为数组, 那它会忽略掉该验证器。

[[yii\validators\UniqueValidator|unique e (唯一性)]]

```
[
```

```
// a1 需要在 "a1" 特性所代表的字段内唯一
```

```
['a1', 'unique'],
```

```
// a1 需要唯一, 但检验的是 a1 的值在字段 a2 中的唯一性
```

```

['a1', 'unique', 'targetAttribute' => 'a2'],

// a1 和 a2 的组合需要唯一，且它们都能收到错误提示

[['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],

// a1 和 a2 的组合需要唯一，只有 a1 能接收错误提示

['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

// 通过同时在 a2 和 a3 字段中检查 a2 和 a3 的值来确定 a1 的唯一性

['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],

]

```

该验证器检查输入值是否在某表字段中唯一。它只对[活动记录](#)类型的模型类特性起作用，能支持对一个或多过字段的验证。

- **targetClass**: 用于查找输入值的目标 [AR](#) 类。若不设置，则会使用正在进行验证的当前模型类。
- **targetAttribute**: 用于检查输入值唯一性的 **targetClass** 的模型特性。
 - 若不设置，它会直接使用待测特性名（整个参数数组的首元素）。
 - 除了指定为字符串以外，你也可以用数组的形式，同时指定多个用于验证的表字段，数组的键和值都是代表字段的特性名，值表示 **targetClass** 的待测数据源字段，而键表示当前模型的待测特性名。
 - 若键和值相同，你可以只指定值。（如:['a2'] 就代表 ['a2'=>'a2']）
- **filter**: 用于检查输入值唯一性必然会进行数据库查询，而该属性为用于进一步筛选该查询的过滤条件。可以为代表额外查询条件的字符串或数组（关于查询条件的格式，请参考 `[[yii\db\Query::where()]]`）；或者样式为 `function ($query)` 的匿名函数，**\$query** 参数为你希望在该函数内进行修改的 `[[yii\db\Query|Query]]` 对象。

译注：[exist](#) 和 [unique](#) 验证器的机理和参数都相似，有点像一体两面的阴和阳。

- 他们的区别是 **exist** 要求 **targetAttribute** 键所代表的属性在其值所代表字段中找得到；而 **unique** 正相反，要求键所代表的属性不能在其值所代表字段中被找到。
- 从另一个角度来理解：他们都会验证的过程中执行数据库查询，查询的条件即为 `where $v=$k`（假设 **targetAttribute** 的其中一对键值对为 `$k => $v`）。**unique** 要求查询的结果数 `$count==0`，而 **exist** 则要求查询的结果数 `$count>0`
- 最后别忘了，**unique** 验证器不存在 **allowArray** 属性哦。

[[yii\validators\UrlValidator|url（网址）]]

```
[  
    // 检查 "website" 是否为有效的 URL。若没有 URI 方案，则给 "website" 特性加 "http://" 前缀  
    ['website', 'url', 'defaultScheme' => 'http'],  
]
```

该验证器检查输入值是否为有效 URL。

- **validSchemes**: 用于指定那些 URI 方案会被视为有效的数组。默认为 ['http', 'https']，代表 http 和 https URLs 会被认为有效。
- **defaultScheme**: 若输入值没有对应的方案前缀，会使用的默认 URI 方案前缀。默认为 null，代表不修改输入值本身。
- **enableIDN**: 验证过程是否应该考虑 IDN（internationalized domain names，国际化域名，也称多语种域名，比如中文域名）。默认为 false。要注意但是为使用 IDN 验证功能，请先确保安装并开启 intl PHP 扩展，不然会导致抛出异常。

引入第三方代码

有时，你可能会需要在 Yii 应用中使用第三方的代码。又或者是你想要在第三方系统中把 Yii 作为类库引用。在下面这个板块中，我们向你展示如何实现这些目标。

在 Yii 中使用第三方类库

要想在 Yii 应用中使用第三方类库，你主要需要确保这些库中的类文件都可以被正常导入或可以被自动加载。

使用 Composer 包

目前很多第三方的类库都以 Composer 包的形式发布。你只需要以下两个简单的步骤即可安装他们：

1. 修改你应用的 composer.json 文件，并注明需要安装哪些 Composer 包。
2. 运行 php composer.phar install 安装这些包。

这些 Composer 包内的类库，可以通过 Composer 的自动加载器实现自动加载。不过请确保你应用的入口脚本包含以下几行用于加载 Composer 自动加载器的代码：

```
// install Composer autoloader（安装 Composer 自动加载器）
```

```
require(__DIR__ . '/../vendor/autoload.php');

// include Yii class file （加载 Yii 的类文件）

require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
```

使用下载类库

若你的类库并未发布为一个 **Composer** 包，你可以参考以下安装说明来安装它。在大多数情况下，你需要预先下载一个发布文件，并把它解压缩到 **BasePath/vendor** 目录，这里的 **BasePath** 代指你应用程序自身的 **base path**（主目录）。

若该类库包含他自己的类自动加载器，你可以把它安装到你应用的**入口脚本**里。我们推荐你把它的安装代码置于 **Yii.php** 的导入之前，这样 **Yii** 的官方自动加载器可以拥有更高的优先级。

若一个类库并没有提供自动加载器，但是他的类库命名方式符合 **PSR-4** 标准，你可以使用 **Yii** 官方的自动加载器来自动加载这些类。你只需给他们的每个根命名空间声明一下**根路径别名**。比如，假设说你已经在目录 **vendor/foo/bar** 里安装了一个类库，且这些类库的根命名空间为 **xyz**。你可以把以下代码放入你的应用配置文件中：

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

若以上情形都不符合，最可能是这些类库需要依赖于 **PHP** 的 **include_path** 配置，来正确定位并导入类文件。只需参考它的安装说明简单地配置一下 **PHP** 导入路径即可。

最悲催的情形是，该类库需要显式导入每个类文件，你可以使用以下方法按需导入相关类文件：

- 找出该库内包含哪些类。
- 在应用的**入口脚本**里的 **Yii::\$classMap** 数组中列出这些类，和他们各自对应的文件路径。

举例来说，

```
Yii::$classMap['Class1'] = 'path/to/Class1.php';
Yii::$classMap['Class2'] = 'path/to/Class2.php';
```

在第三方系统内使用 **Yii**

因为 **Yii** 提供了很多牛逼的功能，有时，你可能会想要使用它们中的一些功能用来支持开发或完善某些第三方的系统，比如：**WordPress**，**Joomla**，或是用其他 **PHP** 框架开发的应用程序。举两个例子吧，

你可能会想念方便的 `[[yii\helpers\ArrayHelper]]` 类，或在第三方系统中使用 [Active Record](#) 活动记录功能。要实现这些目标，你只需两个步骤：安装 Yii，启动 Yii。

若这个第三方系统支持 **Composer** 管理他的依赖文件，你可以直接运行一下命令来安装 Yii：

```
php composer.phar require yiisoft/yii2-framework:*  
php composer.phar install
```

不然的话，你可以[下载](#) Yii 的发布包，并把它解压到对应系统的 `BasePath/vendor` 目录内。

之后，你需要修改该第三方应用的入口脚本，在开头位置添加 Yii 的引入代码：

```
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');  
  
$yiiConfig = require(__DIR__ . '/../config/yii/web.php');  
  
new yii\web\Application($yiiConfig); // 千万别在这调用 run() 方法。（笑）
```

如你所见，这段代码与典型的 Yii 应用的[入口脚本](#)非常相似。唯一的不同之处在于在 Yii 应用创建成功之后，并不会紧接着调用 `run()` 方法。因为，`run()` 方法的调用会接管 HTTP 请求的处理流程。（译注：换言之，这就不是第三方系统而是 Yii 系统了，URL 规则也会跟着换成 Yii 的规则了）

与 Yii 应用中一样，你可以依据运行该第三方系统的环境，针对性地配置 Yii 应用实例。比如，为了使用[活动记录](#)功能，你需要先用该第三方系统的 DB 连接信息，配置 Yii 的 `db` 应用组件。

现在，你就可以使用 Yii 提供的绝大多数功能了。比如，创建 **AR** 类，并用它们来操作数据库。

配合使用 Yii 2 和 Yii 1

如果你之前使用 Yii 1，大概你也有正在运行的 Yii 1 应用吧。不必用 Yii 2 重写整个应用，你也可以通过增添对哪些 Yii 2 独占功能的支持来增强这个系统。下面我们就来详细描述一下具体的实现过程。

注意：Yii 2 需要 PHP 5.4+ 的版本。你需要确保你的服务器以及现有应用都可以支持 PHP 5.4。

首先，参考前文板块中给出的方法，在已有的应用中安装 Yii 2。

之后，如下修改 Yii 1 应用的入口脚本：

```
// 导入下面会详细说定制 Yii 类文件。  
require(__DIR__ . '/../components/Yii.php');  
  
// Yii 2 应用的配置文件  
$yii2Config = require(__DIR__ . '/../config/yii2/web.php');  
new yii\web\Application($yii2Config); // Do NOT call run()
```

```
// Yii 1 应用的配置文件

$yii1Config = require(__DIR__ . '/../config/yii1/main.php');

Yii::createWebApplication($yii1Config)->run();
```

因为，Yii 1 和 Yii 2 都包含有 **Yii** 这个类，你应该创建一个定制版的 **Yii** 来把他们组合起来。上面的代码里包含了的这个定制版的 **Yii** 类，可以用以下代码创建出来：

```
$yii2path = '/path/to/yii2';

require($yii2path . '/BaseYii.php'); // Yii 2.x


$yii1path = '/path/to/yii1';

require($yii1path . '/YiiBase.php'); // Yii 1.x


class Yii extends \yii\BaseYii
{
    // 复制粘贴 YiiBase (1.x) 文件中的代码于此
}


Yii::$classMap = include($yii2path . '/classes.php');


// 通过 Yii 1 注册 Yii2 的类自动加载器

Yii::registerAutoloader(['Yii', 'autoload']);
```

大功告成！此时，你可以在你代码的任意位置，调用 **Yii::\$app** 以访问 **Yii 2** 的应用实例，而用 **Yii::app()** 则会返回 **Yii 1** 的应用实例：

```
echo get_class(Yii::app()); // 输出 'CWebApplication'

echo get_class(Yii::$app); // 输出 'yii\web\Application'
```

以下部分官方版本尚不明确：

开发工具

- 编撰中 [调试工具栏和调试器](#)
- 编撰中 [使用 Gii 生成代码](#)
- 待定中 [生成 API 文档](#)

测试

- 编撰中 [概述](#)
- 编撰中 [搭建测试环境](#)
- 编撰中 [单元测试](#)
- 编撰中 [功能测试](#)
- 编撰中 [验收测试](#)
- 编撰中 [测试夹具](#)

高级专题

- 编撰中 [高级应用模版](#)
- 编撰中 [从头构建自定义模版](#)
- 编撰中 [控制台命令](#)
- 已定稿 [核心验证器](#)
- 编撰中 [国际化](#)
- 编撰中 [收发邮件](#)
- 编撰中 [性能优化](#)
- 待定中 [共享主机环境](#)
- 编撰中 [模板引擎](#)
- 已定稿 [集成第三方代码](#)

小部件

- 表格视图（GridView）：[链接到 demo 页](#)
- 列表视图（ListView）：[链接到 demo 页](#)

- 详情视图（DetailView）：链接到 [demo 页](#)
- 活动表单（ActiveForm）：链接到 [demo 页](#)
- Pjax: 链接到 [demo 页](#)
- 菜单（Menu）：链接到 [demo 页](#)
- LinkPager: 链接到 [demo 页](#)
- LinkSorter: 链接到 [demo 页](#)
- 已完成 [Bootstrap 小部件](#)
- 已完成 [Jquery UI 小部件](#)

助手类

- 编撰中 [助手一览](#)
- 待定中 [ArrayHelper](#)
- 待定中 [Html](#)
- 待定中 [Url](#)
- 待定中 [security](#)

不在目录内的文件

.
