Yii 2.0 权威指南

http://www.yiiframework.com/doc/guide

Qiang Xue,
Alexander Makarov,
Carsten Brandt,
Klimov Paul,
and
many contributors from the Yii community

简体中文 translation provided by:
https://www.yiichina.com,
cuileon,
qiansen1386,
deepziyu,
shi-yang,
iyuanc,
xiaoliushifu,
jhq0113,
funson86,
riverlet,
abrahamgreyson,
fengyh

This tutorial is released under the Terms of Yii Documentation.

Copyright 2014 Yii Software LLC. All Rights Reserved.

Contents

1 介绍（Introduction） 1

1.1 Yii 是什么 1

1.2 从 Yii 1.1 升级 2

2 入门（Getting Started） 13

2.1 你需要了解什么（What do you need to know） 13

2.2 安装 Yii 14

2.3 运行应用 20

2.4 说声 Hello 23

2.5 使用表单 26

2.6 使用数据库 31

2.7 使用 Gii 生成代码 36

2.8 更上一层楼（Looking Ahead） 42

3 应用结构（Application Structure） 45

3.1 总览 45

3.2 入口脚本 46

3.3 应用主体 48

3.4 应用组件 58

3.5 控制器 60

3.6 模型 68

3.7 视图 77

3.8 模块 89

3.9 过滤器 94

3.10 小部件 102

3.11 资源 105

3.12 扩展 122

4 请求处理（Handling Requests） 133

4.1 运行机制概述 133

4.2 启动引导（Bootstrapping） 134

4.3 路由 135

4.4 请求 148

4.5 响应 151
4.6 Sessions 和 Cookies 156
4.7 错误处理 163
4.8 日志 166

5 关键概念（Key Concepts） 175
5.1 组件（Component） 175
5.2 属性（Properties） 177
5.3 事件（Events） 178
5.4 行为 186
5.5 配置（Configurations） 192
5.6 别名（Aliases） 197
5.7 类自动加载（Autoloading） 199
5.8 服务定位器（Service Locator） 201
5.9 依赖注入容器（Dependency Injection Container） 203

6 配合数据库工作（Working with Databases） 213
6.1 数据库访问对象（Database Access Objects） 213
6.2 查询构建器 226
6.3 活动记录（Active Record） 243
6.4 数据库迁移 276

7 接收用户数据（Getting Data from Users） 297
7.1 创建表单 297
7.2 输入验证 302
7.3 文件上传 318
7.4 收集列表输入 322
7.5 多模型的复合表单 324
7.6 在客户端扩展 ActiveForm 325

8 显示数据（Displaying Data） 331
8.1 数据格式器（Data Formatting） 331
8.2 分页 336
8.3 排序 337
8.4 数据提供者 339
8.5 数据小部件 346
8.6 客户端脚本使用 360
8.7 主题 363

9 安全（Security） 367
9.1 安全（Security） 367
9.2 认证 367
9.3 授权 371
9.4 处理密码 388

9.5 加密（Cryptography） 389
9.6 最佳安全实践 390

10 缓存（Caching） 397
10.1 缓存 397
10.2 数据缓存 397
10.3 片段缓存 405
10.4 页面缓存 409
10.5 HTTP 缓存 409

11 RESTful Web 服务（RESTful Web Services） 413
11.1 快速入门 413
11.2 资源 417
11.3 控制器 421
11.4 路由 425
11.5 响应格式 427
11.6 认证 430
11.7 限流 (Rate Limiting) 433
11.8 版本 434
11.9 错误处理 436

12 开发工具（Development Tools） 439

13 测试（Testing） 441
13.1 测试 441
13.2 测试环境设置 442
13.3 单元测试 443
13.4 功能测试 443
13.5 验收测试 444
13.6 Fixtures 444

14 高级专题（Special Topics） 453
14.1 创建你自己的应用程序结构 453
14.2 控制台命令 454
14.3 核心验证器（Core Validators） 461
14.4 Yii 和 Docker 473
14.5 国际化（Internationalization） 475
14.6 收发邮件 489
14.7 性能优化 493
14.8 共享托管环境 498
14.9 使用模板引擎（Using template engines） 499
14.10引入第三方代码（Working with Third-Party Code） 500
14.11使用 Yii 作为微框架 504

15 小部件（Widgets） 509

16 助手类（Helpers） 511
16.1 助手类（Helpers） 511
16.2 数组助手类（ArrayHelper） 512
16.3 Html 帮助类（Html helper） 521
16.4 Url 助手类（Url Helper） 528

Chapter 1

介绍（Introduction）

1.1 Yii 是什么

Yii 是一个高性能，基于组件的 PHP 框架，用于快速开发现代 Web 应用程序。 名字 Yii （读作 易）在中文里有“极致简单与不断演变”两重含义， 也可看作 **Yes It Is**! 的缩写。

1.1.1 Yii 最适合做什么?

Yii 是一个通用的 Web 编程框架，即可以用于开发各种用 PHP 构建的 Web 应用。 因为基于组件的框架结构和设计精巧的缓存支持，它特别适合开发大型应用， 如门户网站、社区、内容管理系统（CMS）、 电子商务项目和 RESTful Web 服务等。

1.1.2 Yii 和其他框架相比呢?

如果你有其它框架使用经验，那么你会很开心看到 Yii 所做的努力：

- 和其他 PHP 框架类似，Yii 实现了 MVC（Model-View-Controller）设计模式并基于该模式组织代码。
- Yii 的代码简洁优雅，这是它的编程哲学。 它永远不会为了刻板地遵照某种设计模式而对代码进行过度的设计。
- Yii 是一个全栈框架，提供了大量久经考验，开箱即用的特性： 对关系型和 NoSQL 数据库都提供了查询生成器和 ActiveRecord；RESTful API 的开发支持；多层缓存支持，等等。
- Yii 非常易于扩展。你可以自定义或替换几乎任何一处核心代码。你还会受益于 Yii 坚实可靠的扩展架构，使用、再开发或再发布扩展。
- 高性能始终是 Yii 的首要目标之一。

Yii 不是一场独角戏，它由一个强大的开发者团队[1]提供支持， 也有一个庞大的专家社区，持续不断地对 Yii 的开发作出贡献。 Yii 开发者团队始终对 Web 开发趋势和其他框架及项目中的最佳实践和特性保持密切关注， 那些

[1] https://www.yiiframework.com/team

有意义的最佳实践及特性会被不定期的整合进核心框架中， 并提供简单优雅的接口。

1.1.3 Yii 版本

Yii 当前有两个主要版本：1.1 和 2.0。 1.1 版是上代的老版本，现在处于维护状态。 2.0 版是一个完全重写的版本，采用了最新的技术和协议，包括依赖包管理器 Composer、PHP 代码规范 PSR、命名空间、Traits（特质）等等。 2.0 版代表新一代框架，是未来几年中我们的主要开发版本。 本指南主要基于 2.0 版编写。

1.1.4 系统要求和先决条件

Yii 2.0 需要 PHP 7.3.0 或以上版本支持。你可以通过运行任何 Yii 发行包中附带的系统要求检查器查看每个具体特性所需的 PHP 配置。

使用 Yii 需要对面向对象编程（OOP）有基本了解，因为 Yii 是一个纯面向对象的框架。Yii 2.0 还使用了 PHP 的最新特性， 例如命名空间[2]和Trait（特质）[3]。 理解这些概念将有助于你更快地掌握 Yii 2.0。

1.2 从 Yii 1.1 升级

2.0 版框架是完全重写的，在 1.1 和 2.0 两个版本之间存在相当多差异。 因此从 1.1 版升级并不像小版本间的跨越那么简单， 通过本指南你将会了解两个版本间主要的不同之处。

如果你之前没有用过 Yii 1.1，可以跳过本章，直接从”入门篇“开始读起。

请注意，Yii 2.0 引入了很多本章并没有涉及到的新功能。 强烈建议你通读整部权威指南来了解所有新特性。 这样有可能会发现一些以前你要自己开发的功能，而现在已经被包含在核心代码中了。

1.2.1 安装

Yii 2.0 完全拥抱 Composer[4]，它是事实上的 PHP 依赖管理工具。 核心框架以及扩展的安装都通过 Composer 来处理。想要了解更多如何安装 Yii 2.0 请参阅本指南的 安装 Yii 章节。如果你想创建新扩展， 或者把你已有的 Yii 1.1 的扩展改写成兼容 2.0 的版本， 你可以参考 创建扩展 章节。

1.2.2 PHP 需求

Yii 2.0 需要 PHP 5.4 或更高版本，该版本相对于 Yii 1.1 所需求的 PHP 5.2 而言有巨大的改进。 因此在语言层面上有很多的值得注意的不同之处。 下面是 PHP 层的主要变化汇总:

[2] https://www.php.net/manual/zh/language.namespaces.php
[3] https://www.php.net/manual/zh/language.oop5.traits.php
[4] https://getcomposer.org/

- 命名空间[5]
- 匿名函数[6]
- 数组短语法 `[...元素...]` 用于取代 `array(...元素...)`
- 视图文件中的短格式 echo 标签 `<?=`，自 PHP 5.4 起总会被识别并且合法，无论 short_open_tag 的设置是什么，可以安全使用。
- SPL 类和接口[7]
- 延迟静态绑定[8]
- 日期和时间[9]
- Traits[10]
- intl[11] Yii 2.0 使用 PHP 扩展 `intl` 来支持国际化的相关功能。

1.2.3 命名空间

Yii 2.0 里最明显的改动就数命名空间的使用了。几乎每一个核心类都引入了命名空间， 比如 `yii\web\Request`。1.1 版类名前缀 "C" 已经不再使用。当前的命名方案与目录结构相吻合。例如，`yii\web\Request` 就表明对应的类文件是 Yii 框架文件夹下的 `web/Request.php` 文件。

（有了 Yii 的类自动加载器， 你可以直接使用全部核心类而不需要显式包含具体文件。）

1.2.4 组件（Component）与对象（BaseObject）

Yii 2.0 把 1.1 中的 `CComponent` 类拆分成了两个类：`yii\base\Object` 和 `yii\base\Component`。 `Object` 类是一个轻量级的基类，你可以通过 getters 和 setters 来定义对象的属性。 `Component` 类继承自 `Object`， 同时进一步支持 事件 和 行为。

如果你不需要用到事件或行为， 应该考虑使用 `Object` 类作为基类。 这种类通常用来表示基本的数据结构。

1.2.5 对象的配置

`Object` 类引入了一种统一对象配置的方法。 所有 `Object` 的子类都应该用以下方法声明它的构造方法（如果需要的话）， 以正确配置它自身：

```
class MyClass extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... 配置生效前的初始化过程
```

[5] https://www.php.net/manual/zh/language.namespaces.php
[6] https://www.php.net/manual/zh/functions.anonymous.php
[7] https://www.php.net/manual/zh/book.spl.php
[8] https://www.php.net/manual/zh/language.oop5.late-static-bindings.php
[9] https://www.php.net/manual/zh/book.datetime.php
[10] https://www.php.net/manual/zh/language.oop5.traits.php
[11] https://www.php.net/manual/zh/book.intl.php

```
        parent::__construct($config);
    }

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

        // ... 配置生效后的初始化过程
    }
}
```

在上面的例子里，构造方法的最后一个参数必须传入一个配置数组， 包含一系列用于在方法结尾初始化相关属性的键值对。 你可以重写 `init()` 方法来执行一些需要在配置生效后进行的初始化工作。

你可以通过遵循以下约定俗成的编码习惯， 来使用配置数组创建并配置新的对象：

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

更多有关配置的细节可以在配置章节找到。

1.2.6 事件（Event）

在 Yii 1 中，通常通过定义 `on` 开头的方法（例如 `onBeforeSave`）来创建事件。 而在 Yii 2 中，你可以使用任意的事件名了。同时通过调用 `trigger()` 方法来触发相关事件：

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

要给事件附加一个事件事件处理器，需要使用 `on()` 方法：

```
$component->on($eventName, $handler);
// 解除事件处理器，使用 off 方法：
// $component->off($eventName, $handler);
```

事件功能还有更多增强之处。要了解它们，请查看事件章节。

1.2.7 路径别名（Path Alias）

Yii 2.0 将路径别名的应用扩大至文件/目录路径和 URL。Yii 2.0 中路径别名必须以 `@` 符号开头， 以区别于普通文件目录路径或 URL。例如 `@yii` 就是指向 Yii 安装目录的别名。 绝大多数 Yii 核心代码都支持别名。 例如 `yii\caching\FileCache::$cachePath` 就同时支持路径别名或普通的目录地址。

路径别名也和类的命名空间密切相关。建议给每一个根命名空间定义一个路径别名， 从而无须额外配置，便可启动 Yii 的类自动加载机制。 例如，因为有 `@yii` 指向 Yii 安装目录， 那类似 `yii\web\Request` 的类就能被 Yii 自动加载。同理，若你用了一个第三方的类库， 如 Zend Framework，你只需定义一个名为 `@Zend` 的路径别名指向该框架的安装目录。 之后 Yii 就可以自动加载任意 Zend Framework 中的类了。

更多路径别名信息请参阅路径别名章节。

1.2.8 视图（View）

Yii 2 中视图最明显的改动是视图内的特殊变量 `$this` 不再指向当前控制器或小部件， 而是指向**视图**对象，它是 2.0 中引入的全新概念。**视图**对象为 `yii\web\View` 的实例， 他代表了 MVC 模式中的视图部分。如果你想要在视图中访问一个控制器或小部件， 可以使用 `$this->context`。

要在其他视图里渲染一个局部视图，使用 `$this->render()`，而不是 `$this->renderPartial()`。 `render()` 现在只返回渲染结果，而不是直接显示它，所以现在你必须显式地把它 **echo** 出来。像这样：

```
echo $this->render('_item', ['item' => $item]);
```

除了使用 PHP 作为主要的模板语言，Yii 2.0 也装备了两种流行模板引擎的官方支持：Smarty 和 Twig。 过去的 Prado 模板引擎不再被支持。要使用这些模板引擎， 你需要配置 `view` 应用组件， 给它设置 `View::$renderers` 属性。 具体请参阅模板引擎章节。

1.2.9 模型（Model）

Yii 2.0 使用 `yii\base\Model` 作为模型基类，类似于 1.1 的 `CModel` 。`CFormModel` 被完全弃用了，现在要创建表单模型类，可以通过继承 `yii\base\Model` 类来实现。

Yii 2.0 引进了名为 `scenarios()` 的新方法来声明支持的场景， 并指明在哪个场景下某属性必须经过验证，可否被视为安全值等等。如：

```
public function scenarios()
{
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!role'],
    ];
}
```

上面的代码声明了两个场景：`backend` 和 `frontend` 。对于 `backend` 场景，`email` 和 `role` 属性值都是安全的， 且能进行批量赋值。对于 `frontend` 场景，`email` 能批量赋值而 `role` 不能。 `email` 和 `role` 都必须通过规则验证。

`rules()` 方法仍用于声明验证规则。注意，由于引入了 `scenarios()`，现在已经没有 `unsafe` 验证器了。

大多数情况下，如果 `rules()` 方法内已经完整地指定场景了， 那就不必覆写 `scenarios()`， 也不必声明 `unsafe` 属性值。

要了解更多有关模型的细节，请参考模型章节。

1.2.10 控制器（Controller）

Yii 2.0 使用 `yii\web\Controller` 作为控制器的基类， 它类似于 1.1 的 `CController`。使用 `yii\base\Action` 作为操作类的基类。

这些变化最明显的影响是，当你在写控制器操作的代码时， 应该返回（return）要渲染的内容而不是输出（echo）它：

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

请查看控制器（Controller）章节了解有关控制器的更多细节。

1.2.11 小部件（Widget）

Yii 2.0 使用 `yii\base\Widget` 作为小部件基类，类似于 1.1 的 `CWidget`。

为了让框架获得更好的 IDE 支持，Yii 2.0 引进了一个调用小部件的新语法。 包含 `begin()`，`end()` 和 `widget()` 三个静态方法，用法如下：

```
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// 注意必须 **"echo"** 结果以显示内容
echo Menu::widget(['items' => $items]);

// 传递一个用于初始化对象属性的数组
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... 表单输入栏都在这里 ...
ActiveForm::end();
```

更多细节请参阅小部件章节。

1.2.12 主题（Theme）

2.0 主题的运作方式跟以往完全不同了。它们现在基于**路径映射机制**，该机制会把一个源视图文件的路径映射到一个主题视图文件路径。 举例来说，如果路径映射为 `['/web/views' => '/web/themes/basic']`， 那么 `/web/views/site/index.php`

视图经过主题修饰的版本就会是 `/web/themes/basic/site/index.php`。 也因此让主题现在可以应用在任何视图文件之上， 甚至是渲染控制器上下文环境之外的视图文件或小部件。

同样，`CThemeManager` 组件已经被移除了。 取而代之的 `theme` 成为了 `view` 应用组件的一个可配置属性。

更多细节请参考主题章节。

1.2.13 控制台应用（Console Application）

控制台应用现在如普通的 Web 应用程序一样，由控制器组成， 控制台的控制器继承自 `yii\console\Controller`，类似于 1.1 的 `CConsoleCommand`。

运行控制台命令使用 `yii <route>`， 其中 `<route>` 代表控制器的路由（如 `sitemap/index`）。 额外的匿名参数传递到对应的控制器操作方法， 而有名的参数根据 `yii\console\Controller::options()` 的声明来解析。

Yii 2.0 支持基于代码注释自动生成相的关命令行帮助（help）信息。

更多细节请参阅控制台命令章节。

1.2.14 国际化（I18N）

Yii 2.0 移除了原来内置的日期格式器和数字格式器，为了支持 PECL intl PHP module[12]（PHP 的国际化扩展）的使用。

消息翻译现在由 `i18n` 应用组件执行。 该组件管理一系列消息源， 允许使用基于消息类别的不同消息源。

更多细节请参阅国际化（Internationalization）章节。

1.2.15 动作过滤器（Action Filters）

操作的过滤现在通过行为（behavior）来实现。要定义一个新的，自定义的过滤器，请继承 `yii\base\ActionFilter` 类。 要使用一个过滤器，需要把过滤器类作为一个 `behavior` 绑定到控制器上。 例如，要使用 `yii\filters\AccessControl` 过滤器，你需要在控制器内添加如下代码：

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

更多细节请参考过滤器章节。

[12] https://pecl.php.net/package/intl

1.2.16 前端资源（Assets）

Yii 2.0 引入了一个新的概念，称为**资源包**（Asset Bundle），以代替 1.1 的脚本包概念。

一个资源包是一个目录下的资源文件集合（如 JavaScript 文件、CSS 文件、图片文件等）。每一个资源包被表示为一个类，该类继承自 `yii\web\AssetBundle`。用 `yii\web\AssetBundle::register()` 方法注册一个资源包后，就使它的资源可被 Web 访问了，注册了资源包的页面会自动包含和引用资源包内指定的 JS 和 CSS 文件。

更多细节请参阅 前端资源管理（Asset） 章节。

1.2.17 助手类（Helpers）

Yii 2.0 很多常用的静态助手类，包括：

- `yii\helpers\Html`
- `yii\helpers\ArrayHelper`
- `yii\helpers\StringHelper`
- `yii\helpers\FileHelper`
- `yii\helpers\Json`

请参考助手一览 章节来了解更多。

1.2.18 表单（Forms）

Yii 2.0 引进了**表单栏（field）**的概念，用来创建一个基于 `yii\widgets\ActiveForm` 的表单。一个表单栏是一个由标签、输入框、错误消息（可能还有提示文字）组成的容器，被表示为一个 `ActiveField` 对象。使用表单栏建立表单的过程比以前更整洁利落：

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>
<?php yii\widgets\ActiveForm::end(); ?>
```

请参考创建表单章节来了解更多细节。

1.2.19 查询生成器（Query Builder）

Yii 1.1 中，查询语句的生成分散在多个类中，包括 `CDbCommand`，`CDbCriteria` 以及 `CDbCommandBuilder`。Yii 2.0 以 `Query` 对象的形式表示一个数据库查询，这个对象使用 `QueryBuilder` 在幕后生成 SQL 语句。例如：

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);
```

```
$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

最重要的是，这些查询生成方法还可以和活动记录配合使用。

请参考查询生成器（Query Builder）章节了解更多内容。

1.2.20 活动记录（Active Record）

Yii 2.0 的活动记录改动了很多。 两个最显而易见的改动分别涉及查询语句的生成（query building）和关联查询的处理（relational query handling）。

1.1 中的 `CDbCriteria` 类在 Yii 2 中被 `yii\db\ActiveQuery` 所替代。 这个类是继承自 `yii\db\Query`，因此也继承了所有查询生成方法。开始拼装一个查询可以调用 `yii\db\ActiveRecord::find()` 方法进行：

```
// 检索所有“活动的”客户和订单，并以 ID 排序：
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

要声明一个关联关系，只需简单地定义一个 getter 方法来返回一个 `ActiveQuery` 对象。 getter 方法定义的属性名代表关联表名称。 如，以下代码声明了一个名为 `orders` 的关系（1.1 中必须在 `relations()` 方法内声明关系）：

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

现在你就可以通过调用 `$customer->orders` 来访问关联表中某用户的订单了。你还可以用以下代码进行一场指定条件的实时关联查询：

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

当贪婪加载一段关联关系时，Yii 2.0 和 1.1 的运作机理并不相同。 具体来说，在 1.1 中使用一条 JOIN 语句同时查询主表和关联表记录。 在 Yii 2.0 中会使用两个没有 JOIN 的 SQL 语句：第一条语句取回主表记录， 第二条通过主表记录经主键筛选后查询关联表记录。

当生成返回大量记录的查询时，可以链式书写 `asArray()` 方法， 这样会以数组的形式返回查询结果，而不必返回 `ActiveRecord` 对象，这能显著降低因大量记录读取所消耗的 CPU 时间和内存。如：

```
$customers = Customer::find()->asArray()->all();
```

另一个改变是你不能再通过公共变量定义属性（Attribute）的默认值了。如果你需要这么做的话，可以在你的记录类的 `init` 方法中设置它们。

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

曾几何时，在 1.1 中重写一个活动记录类的构造方法会导致一些问题。它们不会在 2.0 中出现了。 需要注意的是，如果你需要在构造方法中添加一些参数，恐怕必须重写 `yii\db\ActiveRecord::instantiate()` 方法。

活动记录方面还有很多其他的变化与改进， 请参考活动记录章节以了解更多细节。

1.2.21 活动记录行为（Active Record Behaviors）

在 2.0 中遗弃了活动记录行为基类 `CActiveRecordBehavior`。 如果你想创建活动记录行为，需要直接继承 `yii\base\Behavior`。 如果行为类中需要表示一些事件，需要像这样覆写 `events()` 方法：

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

1.2.22 用户及身份验证接口（IdentityInterface）

1.1 中的 `CWebUser` 类现在被 `yii\web\User` 所取代，随之 `CUserIdentity` 类也不在了。 与之相对的，为达到相同目的，你可以实现 `yii\web\IdentityInterface` 接口，它使用起来更直观。 在高级应用模版里提供了一个这样的一个例子。

要了解更多细节请参考认证（Authentication），授权（Authorization）以及高级应用模版 这三个章节。

1.2.23 URL 管理

Yii 2.0 的 URL 管理跟 1.1 中很像。一个主要的改进是现在的 URL 管理支持**可选参数**了。 比如，如果你在 2.0 中定义了一个下面这样的规则，那么它可以同时匹配 `post/popular` 和 `post/1/popular` 两种 URL。 而在 1.1 中为达成相同效果，必须要使用两条规则。

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]
```

请参考URL 解析和生成 章节，以了解更多细节。.

一个重要变化是路由命名的约定， 现在的控制器和动作的驼峰命名会被转换成小写字母， 其中每个单词由一个专有的连字号分隔，例如，`CamelCaseController` 将为 `camel-case`。 有关更多详细信息，请参阅关于 controller IDs 和 action IDs 的部分。

1.2.24 同时使用 Yii 1.1 和 2.x

如果你有一些遗留的 Yii 1.1 代码，需要跟 Yii 2.0 一起使用， 可以参考 1.1 和 2.0 共用章节。

Chapter 2

入门（Getting Started）

2.1 你需要了解什么（What do you need to know）

Yii 的学习曲线并不像其他 PHP 框架那样陡峭，但仍然需要一些基础知识。

2.1.1 PHP

Yii 是一个 PHP 框架，因此请确保您 阅读并理解语言参考[1]。 用 Yii 进行开发时，您将以面向对象的方式编写代码，因此请确保您熟悉类与对象[2]以及命名空间[3]。

2.1.2 面向对象编程（Object oriented programming）

对面向对象编程的基本理解是必需的。如果您不熟悉它，请选中其中一个可用的教程，如 tuts+[4]。

请注意，您的应用程序越复杂，您需要学习的更高级的 OOP 概念才能成功 掌握这种复杂度。

2.1.3 命令行和 Composer（Command line and composer）

Yii 广泛使用了标准的 PHP 包管理器 Composer[5]，因此请确保您阅读 并理解其指南。如果您不熟悉命令行，现在该开始尝试了。 一旦你学会了基础知识，你就永远不想在没有它的情况下工作。

[1] https://www.php.net/manual/zh/langref.php
[2] https://www.php.net/manual/en/language.oop5.basic.php
[3] https://www.php.net/manual/en/language.namespaces.php
[4] https://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762
[5] https://getcomposer.org/

2.2 安装 Yii

你可以通过两种方式安装 Yii：使用 Composer[6] 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。

标准安装完Yii之后，框架和一个项目模板两者都下载并安装好了。 一个项目模板是实现了一些基本特性的一个 可行的Yii项目，比如登录，联系表单，等等。 它的代码是以推荐的方式组织的。因此，它能够适合作为你项目的一个好的起点。

在本章节和以后的章节，我们将会介绍如何去安装Yii和所谓的基本的应用程序模板和如何去实现这个模板上的新特性。 Yii当然也提供了其它模板叫 高级的应用程序模板[7]， 它是更好应用于在一个团队开发环境中去开发多层级的应用程序。

> **信息：** 这个基本的应用程序模板是适合于开发90它不同于高级的应用程序模板主要地在如何使它们的代码是有组织的。 如果你是刚接触Yii，我们强烈建议你坚持使用简单并有足够的功能的基础的应用程序模板。

2.2.1 通过 Composer 安装

安装 Composer

如果还没有安装 Composer，你可以按 getcomposer.org[8] 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

```
curl -sS https://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[9]。

如果遇到任何问题或者想更深入地学习 Composer， 请参考 Composer 文档[10]。 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。

在本指南中，所有 composer 命令都假定您已经安装了全局[11] 的 composer， 这样它可以作为 `composer` 命令。如果您在本地目录中使用 `composer.phar`，则必须相应地调整示例命令。

如果您之前已安装 Composer，请确保使用最新版本。 您可以通过运行 `composer self-update` 来更新Composer。

---

[6] https://getcomposer.org/
[7] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-zh-CN/README.md
[8] https://getcomposer.org/download/
[9] https://getcomposer.org/Composer-Setup.exe
[10] https://getcomposer.org/doc/
[11] https://getcomposer.org/doc/00-intro.md#globally

> **注意：** 在安装 Yii 期间，Composer 需要从 Github API 请求很多信息。 请求的数量取决于您的应用程序所依赖的数量， 并可能大于 **Github API 速率限制**。如果达到此限制，Composer 可能会要求您提供 Github 登录凭据以获取 Github API 访问令牌。在快速连接上，您可能比 Composer 能够处理的时间早，因此我们建议您在安装 Yii 之前配置访问令牌。 有关如何执行此操作的说明，请参阅 Composer documentation about Github API tokens[12]。

安装 Yii

安装 Composer 后，您可以通过在 Web 可访问的文件夹下运行以下命令来安装Yii应用程序模板：

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

这将在一个名为 `basic` 的目录中安装Yii应用程序模板的最新稳定版本。 如果需要，您可以选择不同的目录名称。

> **信息：** 如果 `composer create-project` 命令失败，您也可以参考 Composer 文档的疑难解答[13] 部分中的常见错误。修复错误后，您可以通过在 `basic` 目录内运行 `composer update` 来恢复中止安装。
>
> **提示：** 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[14]：
>
> ```
> composer create-project --prefer-dist --stability=dev
> yiisoft/yii2-app-basic basic
> ```
>
> 请注意，Yii的开发版本不应该用于生产，因为它可能会破坏您的运行代码。

2.2.2 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[15] 下载归档文件。
2. 将下载的文件解压缩到 Web 访问的文件夹中。
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）：

[12] https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens
[13] https://getcomposer.org/doc/articles/troubleshooting.md
[14] https://getcomposer.org/doc/04-schema.md#minimum-stability
[15] https://www.yiiframework.com/download/

```
// !!! 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',
```

2.2.3 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。 这个方法对大多数的大或者小的项目是一个不错的起点。 如果你正好开始学习Yii，这是特别适合的。

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， 你可以考虑安装高级应用模版[16]。

2.2.4 安装 Assets

Yii依靠 Bower[17] 和/或 NPM[18] 软件包来安装 asset（CSS 和 JavaScript）库。 它使用Composer来获取这些库，允许 PHP 和 CSS/JavaScript 包版本同时解析。 这可以通过使用 asset-packagist.org[19] 或 composer asset plugin[20] 来实现。 有关更多详细信息，请参阅 Assets 文档。

您可能希望通过本地 Bower/NPM 客户端管理您的 assets，使用 CDN 或完全避免 assets 的安装。 为了防止通过 Composer 安装 assets，请将以下几行添加到您的 ‘composer.json’ 中：

```
"replace": {
    "bower-asset/jquery": ">=1.11.0",
    "bower-asset/inputmask": ">=3.2.0",
    "bower-asset/punycode": ">=1.3.0",
    "bower-asset/yii2-pjax": ">=2.0.0"
},
```

> **注意：** 在通过 Composer 绕过 assets 安装的情况下，您负责 assets 的安装和解决版本冲突。 准备来自不同扩展名的 assets 文件之间的可能不一致。

2.2.5 验证安装的结果

当安装完成之后， 或配置你的Web服务器(看下面的文章)或使用内置Web Server[21]， 当在项目 `web` 目录下可以通过下面的命令：

```
php yii serve
```

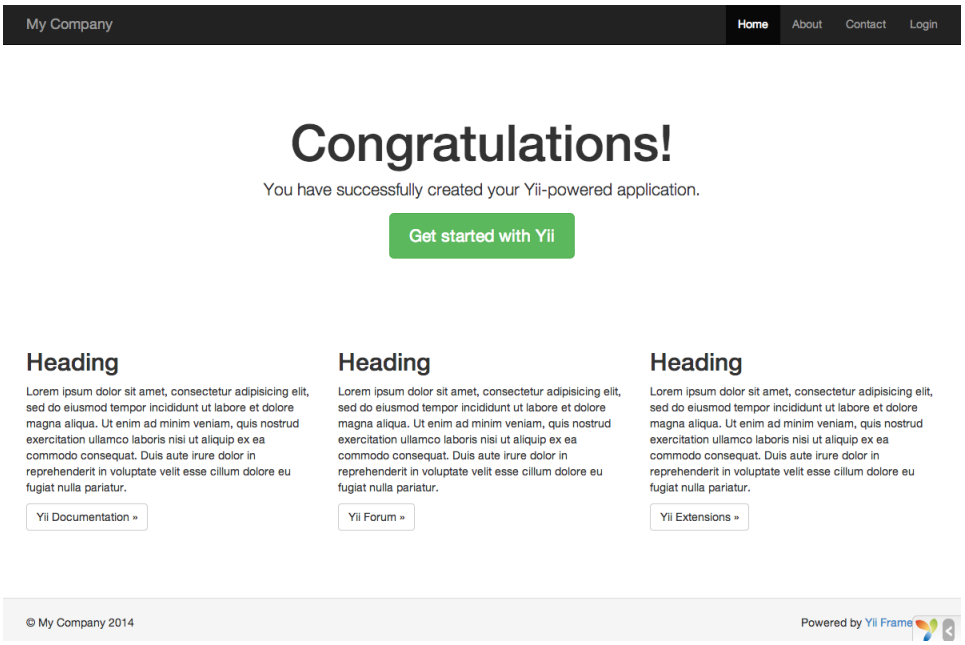
[16] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-zh-CN/README.md
[17] https://bower.io/
[18] https://www.npmjs.com/
[19] https://asset-packagist.org
[20] https://github.com/fxpio/composer-asset-plugin
[21] https://www.php.net/manual/zh/features.commandline.webserver.php

> **注意：** 默认情况下Https-server将监听8080。可是如果这个端口已经使用或者你想通过这个方式运行多个应用程序，你可以指定使用哪些端口。 只加上 –port 参数：

```
php yii serve --port=8888
```

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:8080/
```



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：

- 复制 `/requirements.php` 到 `/web/requirements.php`，然后通过浏览器访问 URL `http://localhost/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 或 以上版本。 如果应用需要用到数据库，那还要安装 PDO PHP 扩展[22] 和相应的数据库驱动（例如访问 MySQL 数据库所需的 `pdo_mysql`）。

2.2.6 配置 Web 服务器

> **提示：** 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。

[22] https://www.php.net/manual/zh/pdo.installation.php

通过上述方法安装的应用程序在 Windows，Max OS X， Linux 中的 Apache HTTP 服务器[23] 或 Nginx HTTP 服务器[24]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[25]， 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 `https://www.example.com/index.php` 访问而不是 `https://www.example.com/basic/web/index.php`。这种配置需要将 Web 服务器的文档根目录(document root)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。

> **信息：** 将 `basic/web` 设置为文档根目录(document root)，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。
>
> **信息：** 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。
>
> **信息：** 如果您在反向代理后面运行Yii应用程序， 则可能需要在请求组件中配置 Trusted proxies and headers。

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用如下配置。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

    # if $showScriptName is false in UrlManager, do not allow accessing URLs
    with script name
    RewriteRule ^index.php/ - [L,R=404]

    # ...其它设置...
</Directory>
```

[23] https://httpd.apache.org/
[24] https://nginx.org/
[25] https://hhvm.com/

推荐使用的 Nginx 配置

为了使用 Nginx[26]，你应该已经将 PHP 安装为 FPM SAPI[27] 了。 你可以使用如下 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; ## listen for ipv4
    #listen [::]:80 default_server ipv6only=on; ## listen for ipv6

    server_name mysite.test;
    root        /path/to/basic/web;
    index       index.php;

    access_log  /path/to/basic/log/access.log;
    error_log   /path/to/basic/log/error.log;

    location / {
        # Redirect everything that isn't a real file to index.php
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # uncomment to avoid processing of calls to non-existing static files by
    Yii
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    # deny accessing php files for the /assets directory
    location ~ ^/assets/.*\.php$ {
        deny all;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
        try_files $uri =404;
    }

    location ~* /\. {
        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

[26] https://www.nginx.com/resources/wiki/
[27] https://www.php.net/manual/zh/install.fpm.php

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。

2.3 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 `https://hostname/basic/web/index.php` 或 `https://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。

> **信息：** 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 `https://hostname/index.php` 或类似的。 请按需调整 URL。

注意项目模板和框架完全不同，安装完之后全都归你了。你可以根据你的需要自由的添加或删除代码和 修改全部的。

2.3.1 功能

一个安装完的基本应用包含四页：

- 主页，当你访问 `https://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。

除了 web 应用程序，还有一个控制台脚本叫 `yii` ,它位于应用程序根目录。 它可以用于程序的后台运行和维护任务，在控制台应用程序章节 中描述。

2.3.2 应用结构

应用中最重要的目录和文件（假设应用根目录是 `basic`）：

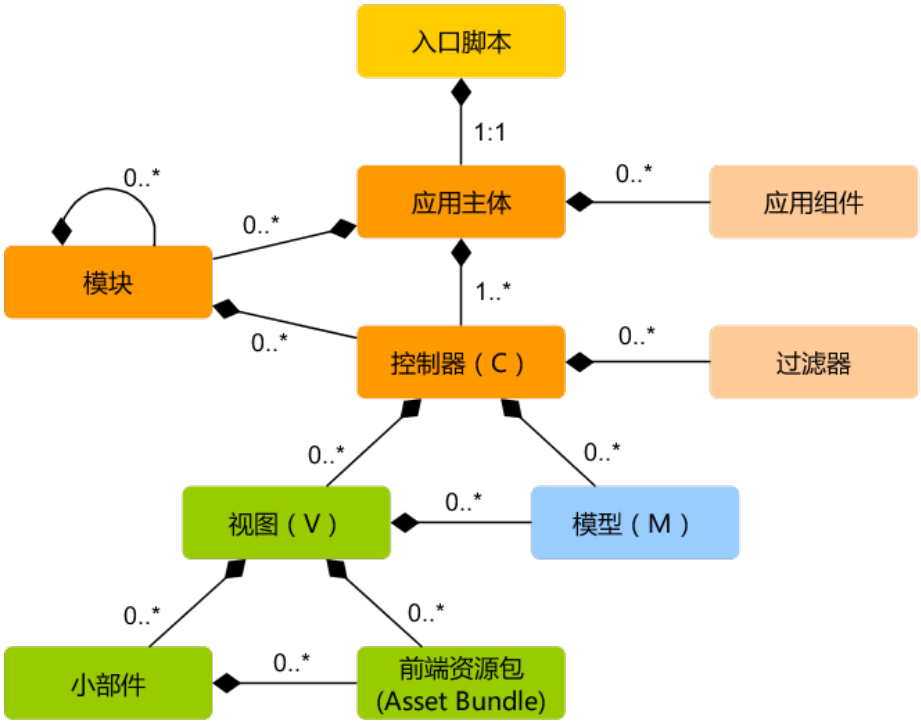
```
basic/                  应用根目录
    composer.json       Composer 配置文件，描述包信息
    config/             包含应用配置及其它配置
        console.php     控制台应用配置信息
        web.php         Web 应用配置信息
    commands/           包含控制台命令类
```

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)[28]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：

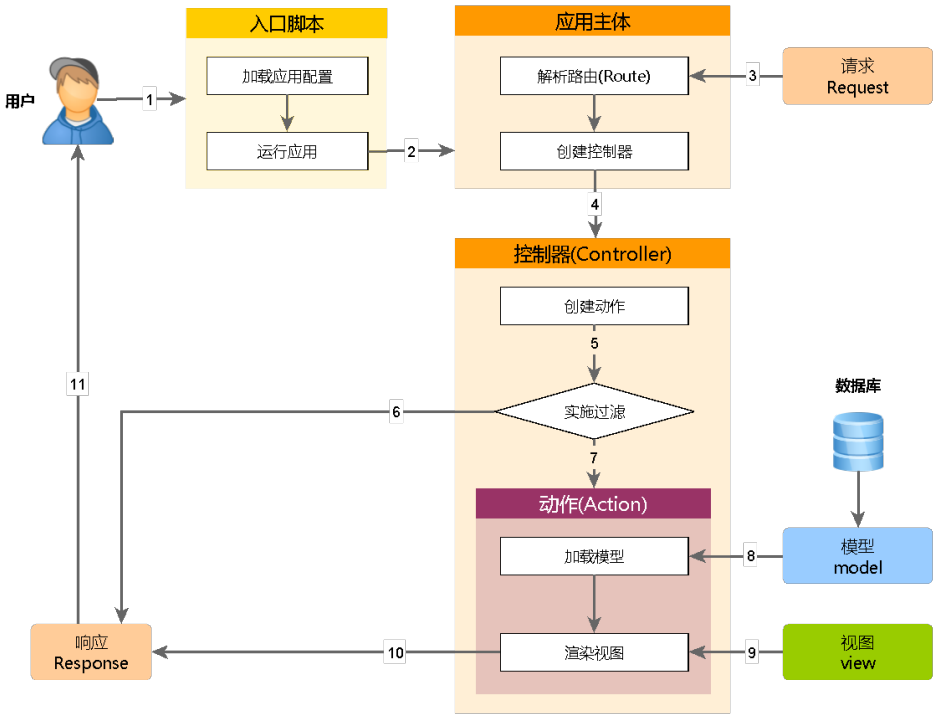


每个应用都有一个入口脚本 web/index.php，这是整个应用中唯一可以访问的 PHP 脚本。 入口脚本接受一个 Web 请求并创建应用实例去处理它。应用在它的组件辅助下解析请求， 并分派请求至 MVC 元素。视图使用小部件 去创建复杂和动态的用户界面。

[28] https://wikipedia.org/wiki/Model-view-controller

2.3.3 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. 控制器创建一个动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，则动作取消。
7. 如果所有过滤器都通过，动作将被执行。
8. 动作会加载一个数据模型，或许是来自数据库。
9. 动作会渲染一个视图，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。

2.4 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动作
- 动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. 如何创建一个动作去响应请求，
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动作。

2.4.1 创建动作

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"。

> **信息:** 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
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...现存的代码...

    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 ‘‘Goodbye’’， 操作方法中的 $message 变量也将被填充为 ‘‘Goodbye’’。

在操作方法中，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）。

2.4.2 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 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 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[29]。

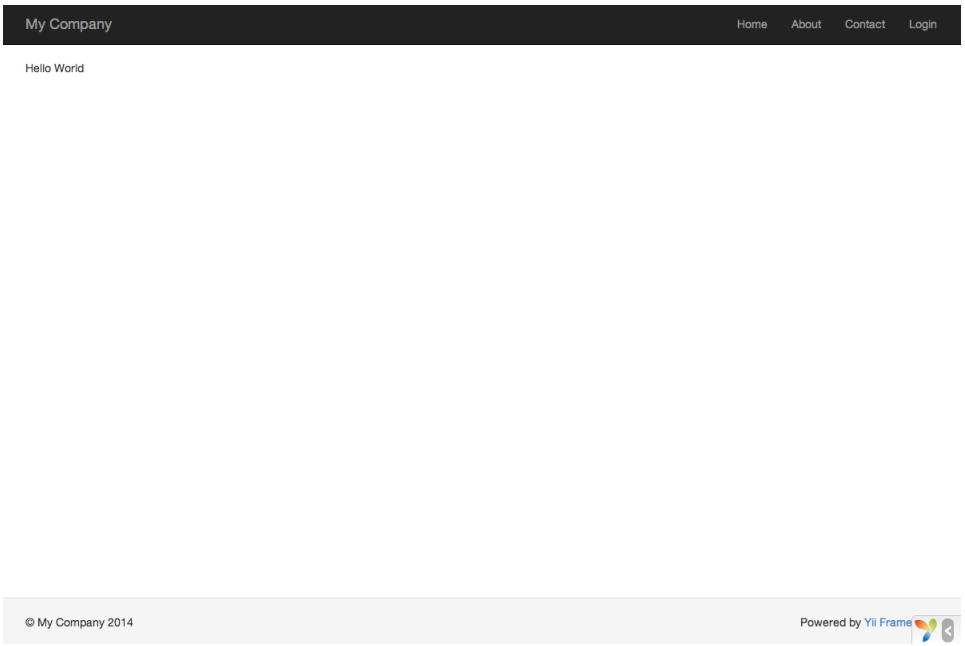
当然了，你大概会在 say 视图里放入更多内容。内容可以由 HTML 标签，纯文本， 甚至 PHP 语句组成。实际上 say 视图就是一个由 render() 执行的 PHP 脚本。 视图脚本输出的内容将会作为响应结果返回给应用。应用将依次输出结果给最终用户。

2.4.3 试运行

创建完动作和视图后，你就可以通过下面的 URL 访问新页面了：

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

[29] https://en.wikipedia.org/wiki/Cross-site_scripting



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。

> **信息：** 新页面和其它页面使用同样的头部和尾部是因为 `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。

> **信息：** 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`。

2.4.4 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动作之间的桥梁。

在下一章节中，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。

2.5 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。

通过本教程，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

2.5.1 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
<?php

namespace app\models;

use Yii;
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。

> **信息：** 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 validate() 方法触发数据验证。 如果有数据验证失败，将把 hasErrors 属性设为 ture， 想要知道具体发生什么错误就调用 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
}
```

2.5.2 创建动作

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
{
    // ...现存的代码...

    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 `validate()` 去确保用户提交的是有效数据。

> **信息：** 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 `refresh()` 或 `redirect()` 去避免表单重复提交问题[30]。

2.5.3 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。视图文件应该保存在 `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。视图文件应该保存在 `views/site/entry.php`。

[30] https://en.wikipedia.org/wiki/Post/Redirect/Get

```
<?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(); ?>
```

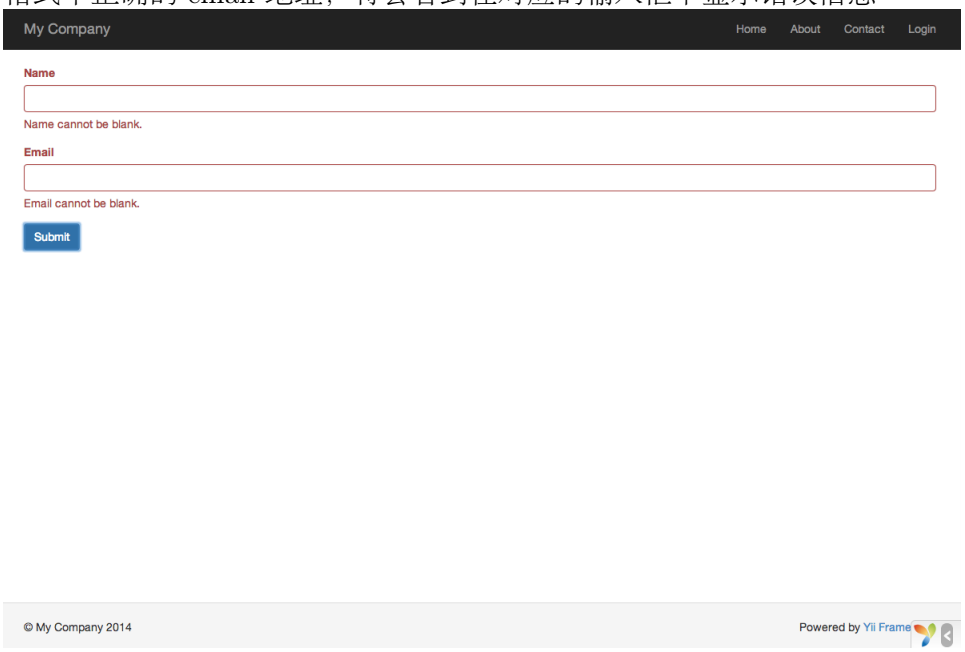
视图使用了一个功能强大的小部件 `ActiveForm` 去生成 HTML 表单。 其中的 `begin()` 和 `end()` 分别用来渲染表单的开始和关闭标签。 在这两个方法之间使用了 `field()` 方法去创建输入框。 第一个输入框用于 “name”，第二个输入框用于 “email”。 之后使用 `yii\helpers\Html::submitButton()` 方法生成提交按钮。

2.5.4 尝试下

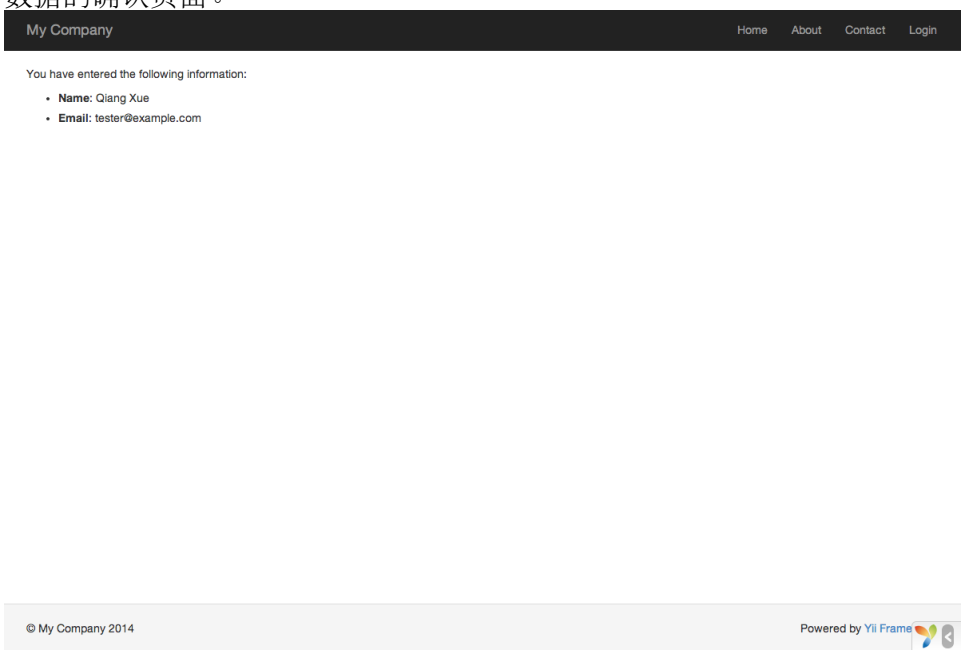
用浏览器访问下面的 URL 看它能否工作:

`https://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。



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。



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 按如下方法：

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

> **信息：** 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。

2.5.5 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。

在下一章节中，你将学习如何使用数据库，几乎每个应用都需要数据库。

2.6 使用数据库

本章节将介绍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。

2.6.1 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，PostregSQL，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。

> **信息：** 虽然 MariaDB 曾经是 MySQL 的直接替代品，但现在已经不再完全正确。如果您希望在 MariaDB 中使用“JSON”支持等高级功能，请查看下面列出的 MariaDB 扩展。

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。

2.6.2 配置数据库连接

开始之前，请确保你已经安装了 PHP PDO[31] 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
<?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。

> **信息：** `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。

如果想要使用 Yii 没有捆绑支持的数据库，你可以查看以下插件：

- Informix[32]
- IBM DB2[33]
- Firebird[34]
- MariaDB[35]

[31] https://www.php.net/manual/zh/book.pdo.php
[32] https://github.com/edgardmessias/yii2-informix
[33] https://github.com/edgardmessias/yii2-ibm-db2
[34] https://github.com/edgardmessias/yii2-firebird
[35] https://github.com/sam-it/yii2-mariadb

2.6.3 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。

> **信息：** 如果类名和数据表名不能直接对应， 可以覆写 `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();
```

> **信息：** 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。

2.6.4 创建动作

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
<?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。

2.6.5 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
<?php
use yii\helpers\Html;
```

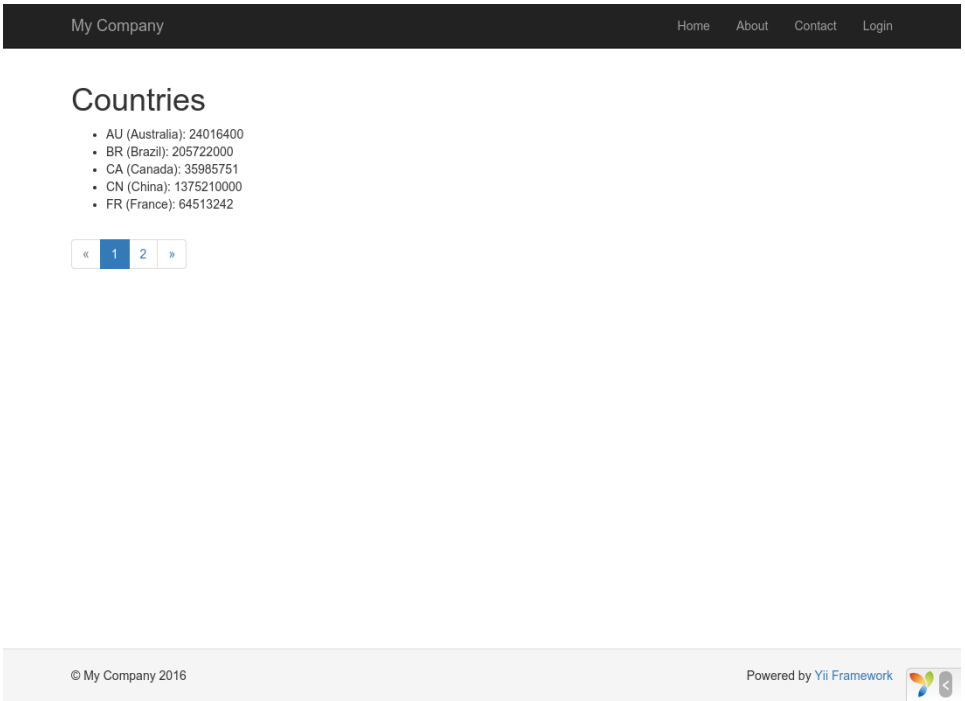
```
use yii\widgets\LinkPager;
?>
<h1>Countries</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

<?= 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。

2.6.6 试运行

浏览器访问下面的 URL 看看能否工作:

```
https://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：

```
https://hostname/index.php?r=country/index&page=2
```

在这个场景里，`Pagination` 提供了为数据结果集分页的所有功能：

- 首先 `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 `LinkPager` 使用 `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。 `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。

2.6.7 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。

2.7 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

2.7.1 开始 Gii

Gii 是 Yii 中的一个模块。 可以通过配置应用的 `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'] = [
        'class' => '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`。

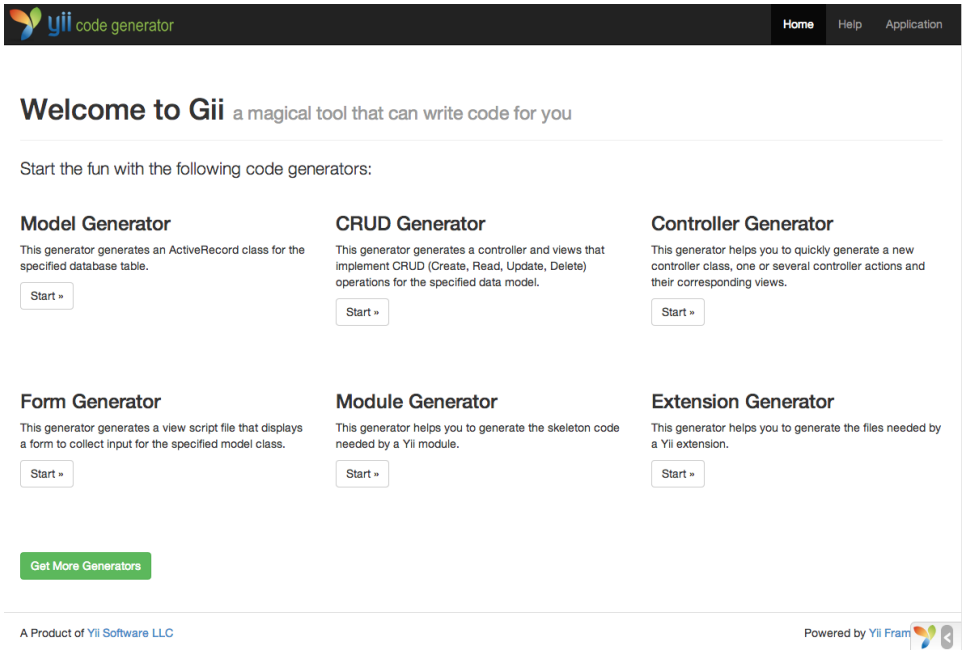
如果你检查应用的入口脚本 `web/index.php`， 将看到这行代码将 `YII_ENV_DEV` 设为 true:

```
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:

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

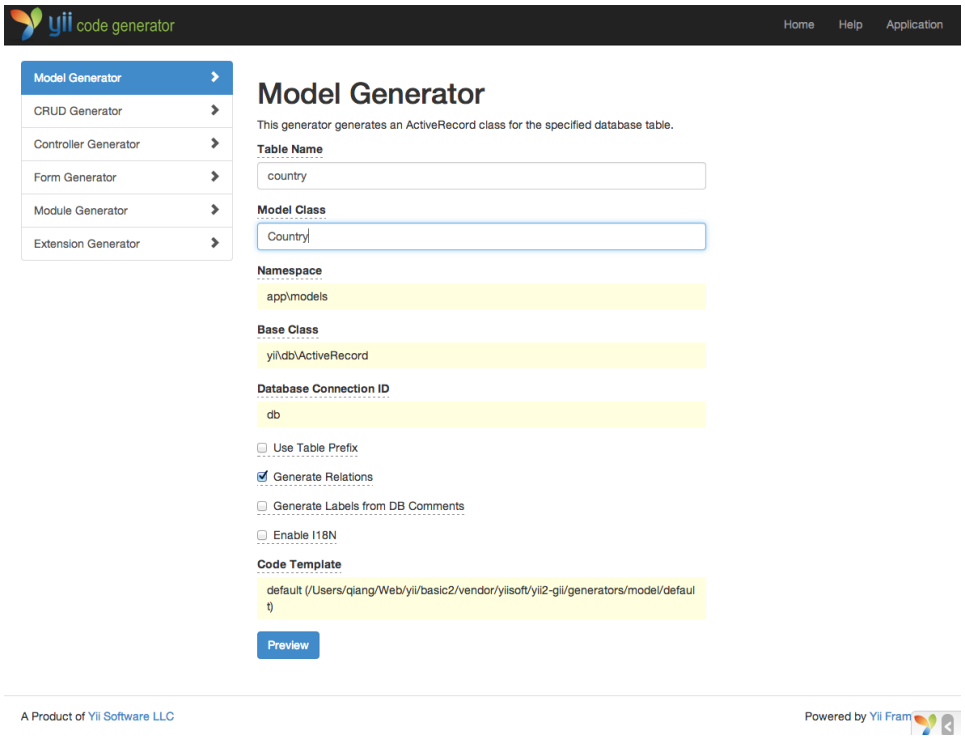
> **信息：** 如果你通过本机以外的机器访问 Gii，请求会被出于安全原因拒绝。 你可以配置 Gii 为其添加允许访问的 IP 地址:
>
> ```
> 'gii' => [
>  'class' => 'yii\gii\Module',
>  'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*',
>  '192.168.178.20'] // 按需调整这里
> ],
> ```



2.7.2 生成活动记录类

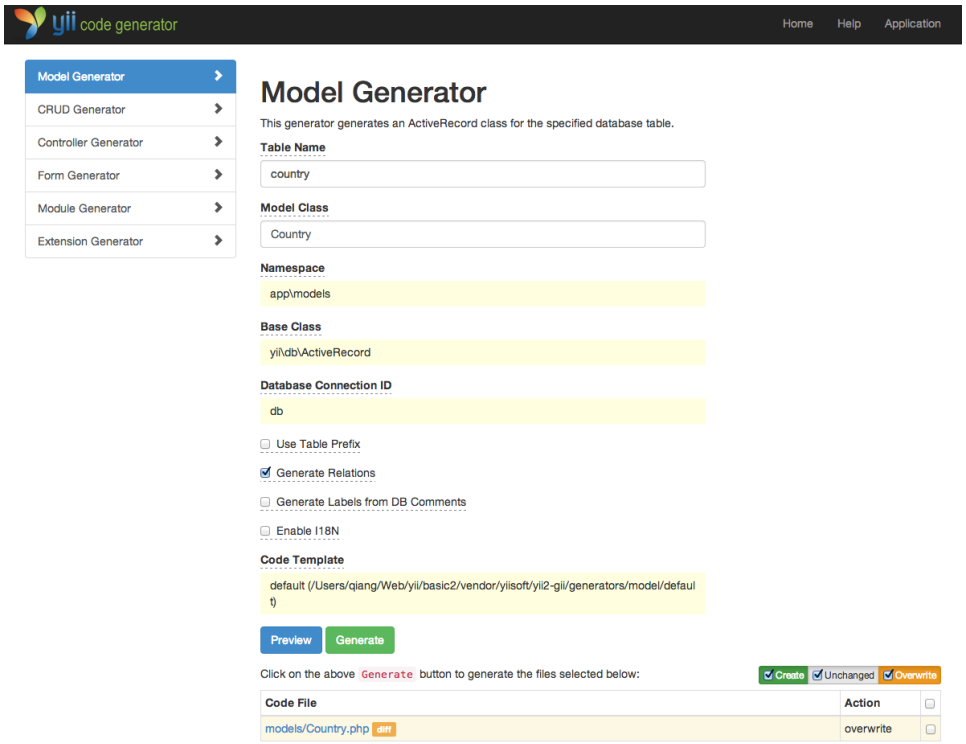
选择 “Model Generator” （点击 Gii 首页的链接）去生成活动记录类。并像这样填写表单:

- Table Name: `country`
- Model Class: `Country`



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。



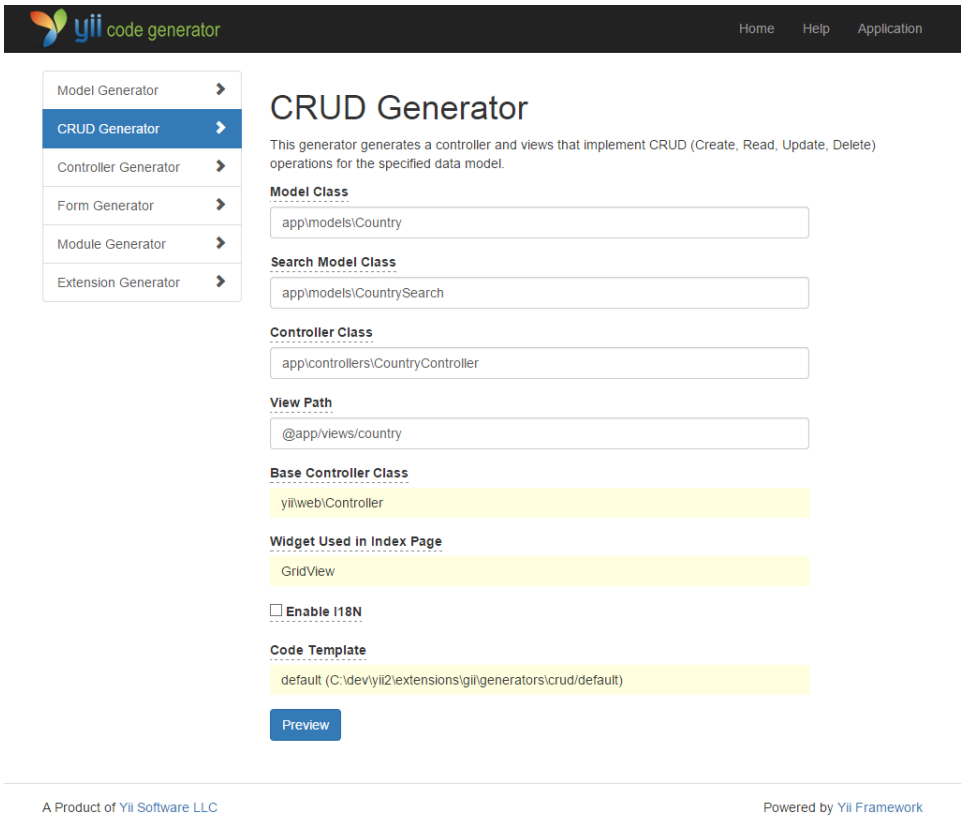
想要覆写已存在文件，选中 “overwrite” 下的复选框然后点击 “Generator”。如果是新文件，只点击 “Generator” 就好。

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。

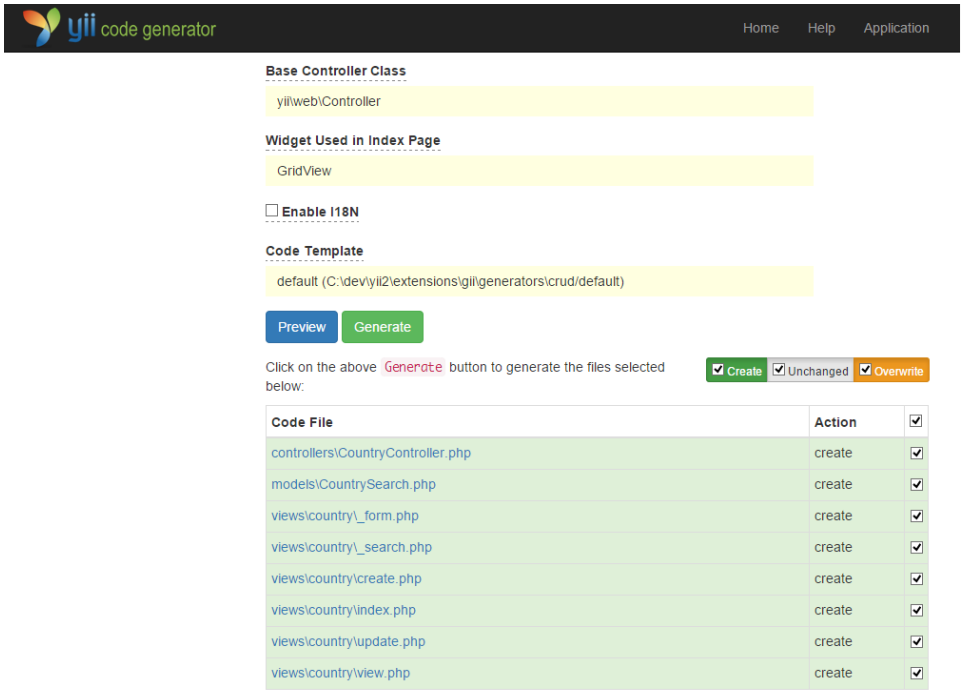
2.7.3 生成 CRUD 代码

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`



然后点击 “Preview” 按钮。你会看到下述将要生成的文件列表。



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）。

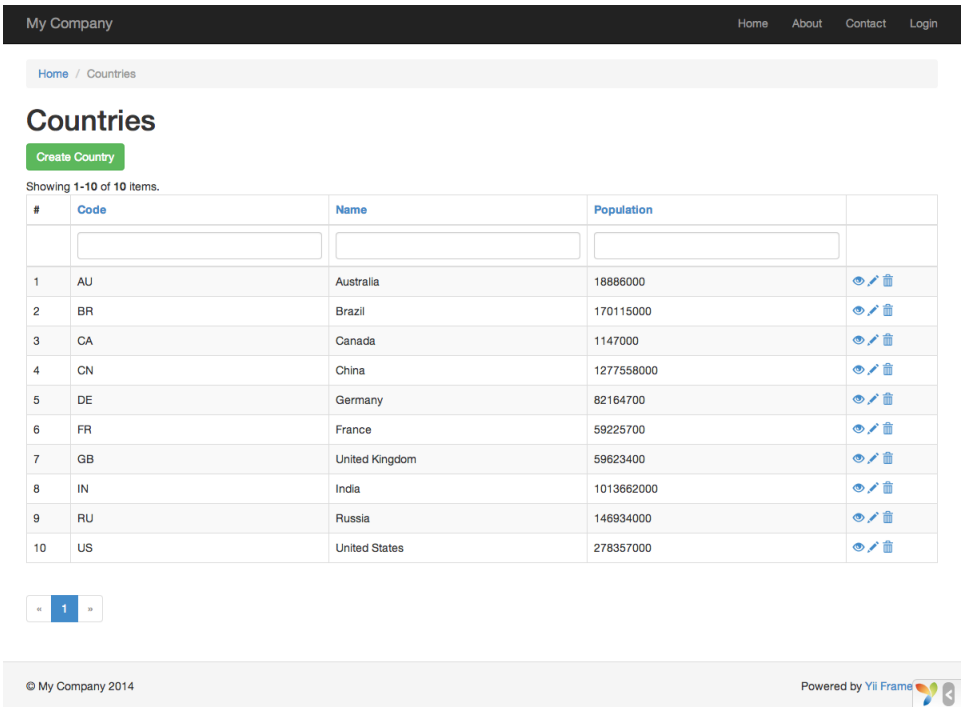
2.7.4 试运行

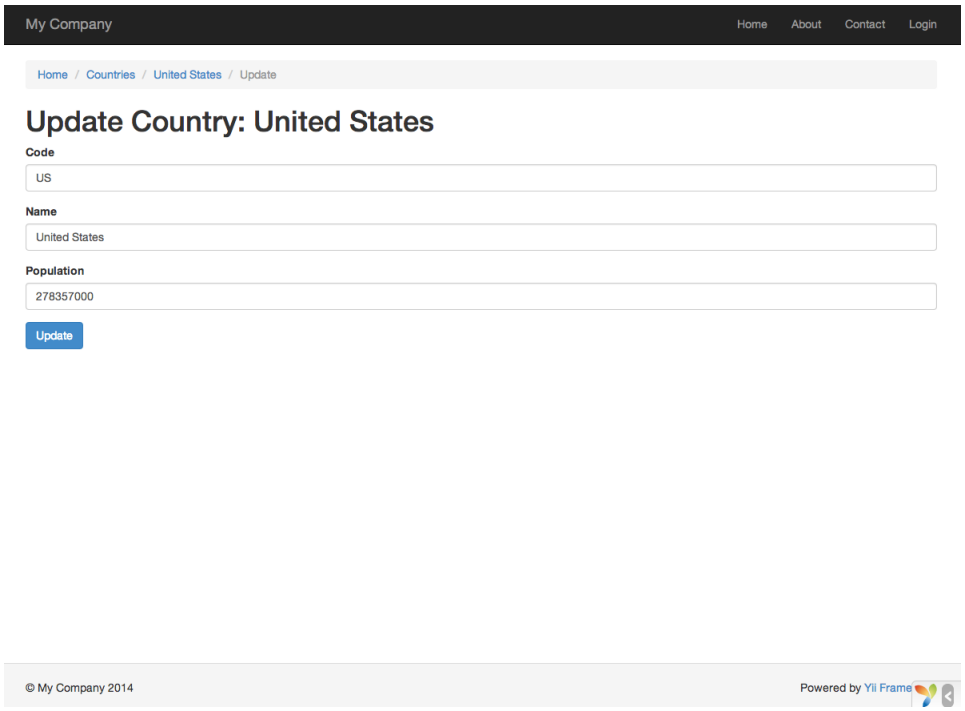
用浏览器访问下面的 URL 查看生成代码的运行：

```
https://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。





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`

> **信息：** 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。

2.7.5 总结

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

2.8 更上一层楼（Looking Ahead）

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[36]：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：https://forum.yiiframework.com/
  - IRC 聊天室：Libera 网络上的 #yii 频道 (ircs://irc.libera.chat:6697)
  - Slack chanel：https://yii.slack.com
  - Gitter chat：https://gitter.im/yiisoft/yii2
  - 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
  - Stackoverflow：https://stackoverflow.com/questions/tagged/yii2

[36] https://www.yiiframework.com/extensions/

Chapter 3

应用结构（Application Structure）

3.1 总览

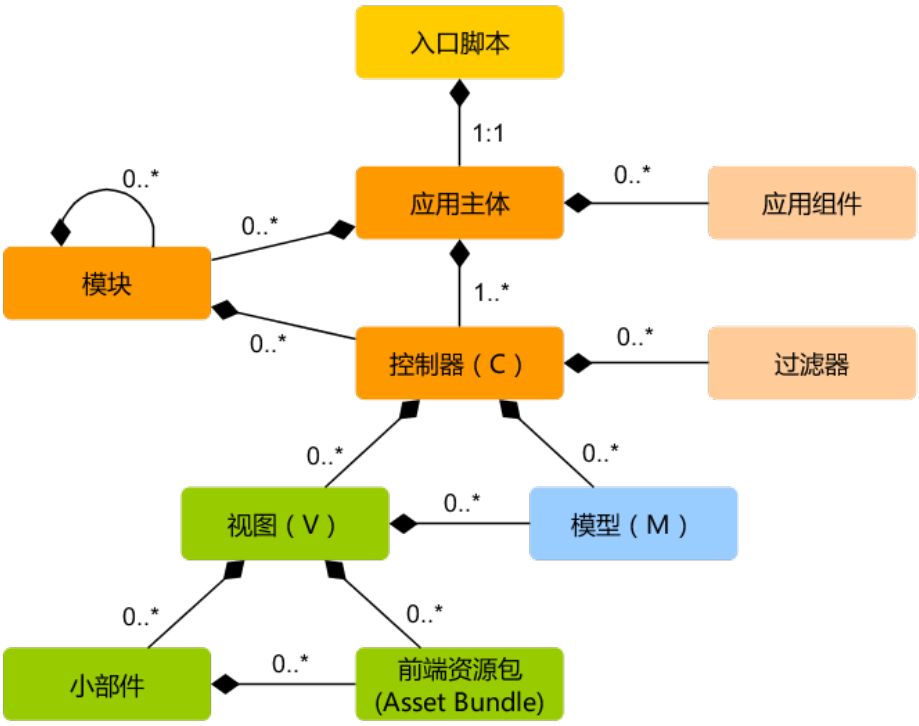
Yii 应用参照模型-视图-控制器 （MVC）[1] 设计模式来组织。 模型代表数据、业务逻辑和规则； 视图展示模型的输出；控制器 接受出入并将其转换为模型和视图命令。

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：

[1] https://zh.wikipedia.org/wiki/MVC



3.2 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[2];
- 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。

[2] https://getcomposer.org/doc/01-basic-usage.md#autoloading

3.2.1 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();
```

3.2.2 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 https://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license https://www.yiiframework.com/license/
 */

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/console.php';

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

3.2.3 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。

3.3 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。

> **信息：** 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: 网页应用主体 和 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。

3.3.1 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。

3.3.2 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。 以下我们简述这些属性。

必要属性

在一个应用中，至少要配置2个属性: `id` 和 `basePath`。

`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。

`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 `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。

`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。

`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。

`bootstrap` 这个属性很实用，它允许你用数组指定启动阶段 `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,
        ],

        // 匿名函数
        function () {
            return new app\components\Profiler();
        }
    ],
]
```

> **信息:** 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 `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';
}
```

> **注意：** 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。

`catchAll` 该属性仅 `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',
    ],
]
```

> **信息：** 当开启这个属性时，开发环境下的调试面板将不能工作。

`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.

`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。

`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”。

如果你想打破上述的规则， 可以配置 controllerMap 属性。

`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\jui\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[3] 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.

`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
```

[3] https://zh.wikipedia.org/wiki/IETF%E8%AA%9E%E8%A8%80%E6%A8%99%E7%B1%A4

```
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

更多详情请参考 模块 一节。

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 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。

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。

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[4]. 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.

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()[5]，例如：

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

有关设置时区含义的更多详细信息，请查看关于日期格式的部分。

[4] https://zh.wikipedia.org/wiki/IETF%E8%AA%9E%E8%A8%80%E6%A8%99%E7%B1%A4
[5] https://www.php.net/manual/zh/function.date-default-timezone-set.php

`version` 该属性指定应用的版本，默认为 `'1.0'`， 其他代码不使用的话可以不配置。

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。

`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。

`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 `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 `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。

`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[6] 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 ...

    ],
]
```

[6] https://getcomposer.org

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

`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。

`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。

`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。

`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.

`vendorPath` 该属性指定 Composer[7] 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。

`enableCoreCommands` 该属性仅 `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`。

3.3.3 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：

[7] https://getcomposer.org

```
[
    '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) {
    // ...
});
```

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。

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 `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。

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。

`EVENT_AFTER_ACTION`

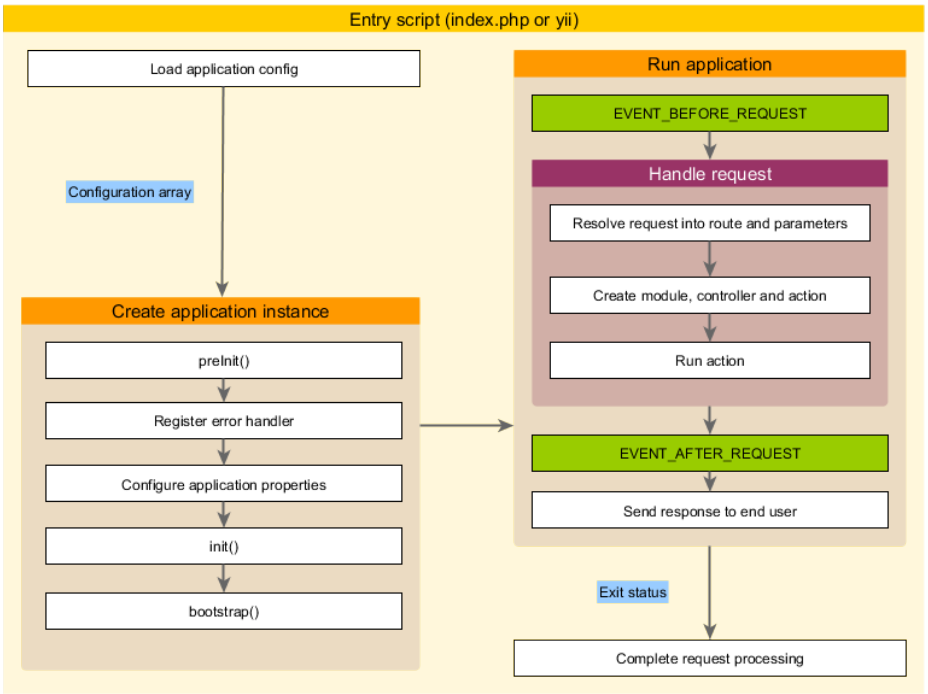
该事件在每个 控制器动作 运行 *after* 之后会被触发， 实际的事件名为 `afterAction`。

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。

3.3.4 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 `preInit()` 配置几个高级别应用主体属性， 比如 `basePath`。
   - 注册 `error handler` 错误处理方法。
   - 配置应用主体属性。
   - 调用 `init()` 初始化，该函数会调用 `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 `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 `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。

3.4 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::$app->db` 来获取到已注册到应用的 `DB connection`，使用 `\Yii::$app->cache` 来获取到已注册到应用的 `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;
        },
    ],
]
```

> **信息：** 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。

3.4.1 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 `bootstrap` 属性中。

你还可以使用闭包来引导启动自定义的组件。不需要直接返回一个实例化的组件。 在应用主体 `yii\base\Application` 实例化后，闭包也会被调用。

例如, 如下的应用主体配置保证了 `log` 组件一直被加载。

```
[
    'bootstrap' => [
        'log',
        function($app){
            return new ComponentX();
        },
        function($app){
            // 可以写自定义的代码
           return;
        }
    ],
    'components' => [
        'log' => [
            // "log" 组件的配置
        ],
    ],
]
```

3.4.2 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 `request` 组件 用来收集用户请求并解析 路由； `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。

- `assetManager`: 管理资源包和资源发布， 详情请参考 管理资源 一节。
- `db`: 代表一个可以执行数据库操作的数据库连接， 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。
- `errorHandler`: 处理 PHP 错误和异常， 详情请参考 错误处理 一节。
- `formatter`: 格式化输出显示给终端用户的数据，例如数字可能要带分隔符， 日期使用长格式。 详情请参考 格式化输出数据 一节。
- `i18n`: 支持信息翻译和格式化。 详情请参考 国际化 一节。
- `log`: 管理日志对象。 详情请参考 日志 一节。
- `yii\swiftmailer\Mailer`: 支持生成邮件结构并发送， 详情请参考 邮件 一节。
- `response`: 代表发送给用户的响应， 详情请参考 响应 一节。
- `request`: 代表从终端用户处接收到的请求， 详情请参考 请求 一节。
- `session`: 代表会话信息， 仅在`Web applications` 网页应用中可用，详情请参考 Sessions (会话) and Cookies 一节。
- `urlManager`: 支持URL地址解析和创建， 详情请参考 URL 解析和生成 一节。
- `user`: 代表认证登录用户信息， 仅在`Web applications` 网页应用中可用， 详情请参考 认证 一节。
- `view`: 支持渲染视图， 详情请参考 Views 一节。

3.5 控制器

控制器是 MVC[8] 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。

3.5.1 动作

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动作`view` and `create` 的控制器`post`:

```
namespace app\controllers;

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

[8] https://zh.wikipedia.org/wiki/MVC

```
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        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。

3.5.2 路由

终端用户通过所谓的路由寻找到动作，路由是包含以下部分的字符串:

- 模块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 `https://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。

3.5.3 创建控制器

在`Web applications`网页应用中，控制器应继承`yii\web\Controller` 或它的子类。 同理在`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` 是真是的控制器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 `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:

1. 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;
2. 去掉中横杠，将正斜杠替换为反斜杠;
3. 增加`Controller`后缀;
4. 在前面增加`controller namespace`控制器命名空间.

下面为一些示例，假设`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/post-comment应在@app/controllers/admin/PostCommentController.php文件中。

> **信息：** 最后一个示例 admin/post-comment 表示你可以将控制器放在 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 controller map 来强制上述的控制器ID和类名对应， 通常用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Web applications网页应用，它的值为 'site'，对于 console applications 控制台应用，它的值为 help，所以URL为 https://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',
]
```

3.5.4 创建动作

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';
    }
}
```

动作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。

内联动作

内联动作指的是根据我们刚描述的操作方法。

动作方法的名字是根据操作ID遵循如下规则衍生：

1. 将每个单词的第一个字母转为大写;
2. 去掉中横杠;
3. 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。

独立动作

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" 动作
        'error' => 'yii\web\ErrorAction',

        // 用配置数组申明 "view" 动作
        '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";
    }
}
```

动作结果

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console applications控制台应用，返回值可为整数， 表示命令行下执行的 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 https://example.com
    return $this->redirect('https://example.com');
}
```

动作参数

内联动作的操作方法和独立动作的 run() 方法可以带参数，称为动作参数。参数值从请求中获取，对于Web applications网页应用， 每个动作参数的值从$_GET中获得，参数名作为键； 对于console applications控制台应用, 动作参数对应命令行参数。

如下例，动作view (内联动作) 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)
    {
        // ...
    }
}
```

动作参数会被不同的参数填入，如下所示:

- https://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;
- https://hostname/index.php?r=post/view&id=123&version=2: $id 和$version 分别填入‘123’ 和‘2’‘;
- https://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;
- https://hostname/index.php?r=post/view&id[]=123: 会抛出yii\web\BadRequestHttpExceptio 异常 因为 $id 参数收到数组值 ['123'] 而不是字符串.

如果你想要一个动作参数来接受数组值，你应该使用 array 来提示它，如下所示:

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

现在如果请求为 https://hostname/index.php?r=post/view&id[]=123, 参数 $id 会使用数组值 ['123']， 如果请求为 https://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。

默认动作

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');
    }
}
```

3.5.5 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`default action ID`默认操作ID;
   - 如果在`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.

3.5.6 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.

3.6 模型

模型是 MVC[9] 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申明的验证规则;
- 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。

> **信息**： 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。

3.6.1 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:

[9] https://zh.wikipedia.org/wiki/MVC

```
$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[10] 和 ArrayIterator 数组迭代器[11]:

```
$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。

[10] https://www.php.net/manual/en/class.arrayaccess.php
[11] https://www.php.net/manual/en/class.arrayiterator.php

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 `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。

> **信息：** 属性标签是 视图一部分， 但是在模型中声明标签通常非常方便，并可形成非常简洁可重用代码。

3.6.2 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
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['username', 'password'],
            self::SCENARIO_REGISTER => ['username', 'email', 'password'],
        ];
    }
}
```

> **信息:** 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
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios[self::SCENARIO_LOGIN] = ['username', 'password'];
        $scenarios[self::SCENARIO_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。

3.6.3 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`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。

3.6.4 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`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'],
    ];
}
```

> **信息：** 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;
```

The same can be done in rules() method:

```
public function rules()
{
    return [
        [['username', 'password', '!secret'], 'required', 'on' => 'login']
    ];
}
```

在这种情况下，属性 `username`, `password` 和 `secret` 是必须的，但是 `secret`必须被明确指定。

3.6.5 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:

- 模型转换成数组;
- 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 `fields()` 和/或 `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`。

3.6.6 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。

3.7 视图

视图是 MVC[12] 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`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。

3.7.1 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(); ?>
```

[12] https://zh.wikipedia.org/wiki/MVC

在视图中，可访问 `$this` 指向 `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。

> **提示：** 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[13] 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>
```

> **提示：** 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 `module directory` 模块目录下的 `views/ControllerID` 目录；

[13] https://zh.wikipedia.org/wiki/%E8%B7%A8%E7%B6%B2%E7%AB%99%E6%8C%87%E4%BB%A4%E7%A2%BC

- 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。

3.7.2 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：

- `render()`：渲染一个 视图名 并使用一个 布局 返回到渲染结果。
- `renderPartial()`：渲染一个 视图名 并且不使用布局。
- `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。
- `renderFile()`：渲染一个视图文件目录或 别名下的视图文件。
- `renderContent()`: renders a static string by embedding it into the currently applicable layout. This method is available since version 2.0.1.

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:

- `render()`: 渲染一个 视图名.
- `renderFile()`: 渲染一个视图文件目录或 别名下的视图文件。

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`view component`视图组件提供的以下方法:

- `render()`: 渲染一个 视图名.
- `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。
  - `renderFile()`: 渲染一个视图文件目录或 别名下的视图文件。

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 `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 `view` 应用组件， 调用它的如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 `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`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 `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 `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`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

view component视图组件提供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params数组的值来 生成显示breadcrumbs:

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

3.7.3 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，更好的方式是将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 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。

- `beginPage()`：该方法应在布局的开始处调用，它触发表明页面开始的 `EVENT_BEGIN_PAGE` 事件。
- `endPage()`：该方法应在布局的结尾处调用，它触发表明页面结尾的 `EVENT_END_PAGE` 事件。
- `head()`：该方法应在HTML页面的`<head>`标签中调用，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。
- `beginBody()`：该方法应在`<body>`标签的开始处调用，它触发 `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。
- `endBody()`：该方法应在`<body>`标签的结尾处调用，它触发 `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`view` 视图组件 后者包含调用`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`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 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 属性不为空null，使用它作为布局的值， 控制器的 module模块 作为上下文模块。
- 如果 layout 为空，从控制器的祖先模块（包括应用） 开始找 第一个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layout path布局路径中查找实际的布局文件， 布局路径默认为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 beginContent() 和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 beginContent() 和 endContent() 方法之间，传给 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 `beginBlock()` 和 `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中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; ?>
...
```

3.7.4 使用视图组件

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[14], Smarty[15]。

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 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：

[14] https://twig.symfony.com/
[15] https://www.smarty.net/

```
<?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' => 'https://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="https://www.yiiframework.com/rss.xml/">
```

和 `registerMetaTags()` 类似， 调用`registerLinkTag()` 指定键来避免生成重复链接标签。

3.7.5 视图事件

`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。

- EVENT_BEFORE_RENDER: 在控制器渲染文件开始时触发， 该事件可设置 yii\base\ViewEvent::$isValid 为 false 取消视图渲染。
- EVENT_AFTER_RENDER: 在布局中调用 yii\base\View::afterRender() 时触发， 该事件可获取yii\base\ViewEvent::$output的渲染结果，可修改该属性来修改渲染结果。
- EVENT_BEGIN_PAGE: 在布局调用 yii\base\View::beginPage() 时触发；
- EVENT_END_PAGE: 在布局调用 yii\base\View::endPage() 是触发；
- EVENT_BEGIN_BODY: 在布局调用 yii\web\View::beginBody() 时触发；
- 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');
});
```

3.7.6 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 的独立动作。 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` 动作请求哪个视图，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。

3.7.7 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。

3.8 模块

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。

3.8.1 创建模块

模块被组织成一个称为 `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 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 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 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。 例如，假定控制器类为 PostController， 目录对应模块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。

模块中的控制台命令

您的模块也可以声明命令，这将通过 控制台 模式可用。

当 Yii 在控制台模式下执行并将其指向命令的命名空间时。想要在命令行中查看你的命令， 你需要更改 yii\base\Module::$controllerNamespace 属性。

一种实现方法是在模块的 init() 方法中测试Yii应用程序的实例类型：

```
public function init()
{
    parent::init();
    if (Yii::$app instanceof \yii\console\Application) {
        $this->controllerNamespace = 'app\modules\forum\commands';
    }
}
```

然后您的命令将从命令行使用以下路由：

```
yii <module_id>/<command>/<sub_command>
```

3.8.2 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 ...
        ],
    ],
]
```

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。

在 yii\web\UrlManager::parseRequest() 被触发之前应该添加模块 URL 管理器规则。 这就意味着在模块的 init() 将不会起作用，因为模块将在路由开始处理时被初始化。 因此应该在 bootstrap stage 添加规则。 使用 yii\web\GroupUrlRule 去实现模块的 URL 规则也是一种很好的做法。

如果一个模块用于 version API， 它的 URL 规则应该直接添加到应用程序配置的 urlManager 中。

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。

> **信息：** 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` 模块就是这种， 为此将这种模块加入到应用主体的 `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',
    ],
]
```

3.8.3 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 `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。

> **信息：** `getModule()` 方法只返回子模块的直属的 父模块。`yii\base\Application::$loadedModules` 保存了已加所有载模块的属性，包括两者的子模块和 嵌套模块，并用他们的类名进行索引。

3.8.4 从模块内部访问组件

从 2.0.13 版本开始模块支持 tree traversal。 这允发模块开发人员通过作为其模块的服务定位器去引用（应用程序）组件。 这意味着最好使用 `$module->get('db')` 而不是 `Yii::$app->get('db')`。 在需要不同组件（配置）的情况下， 模块开发者能够指定要用于模块的特定组件。

例如，以下是部分应用程序的配置:

```
'components' => [
    'db' => [
        'tablePrefix' => 'main_',
        'class' => Connection::class,
        'enableQueryCache' => false
    ],
],
'modules' => [
    'mymodule' => [
        'components' => [
            'db' => [
                'tablePrefix' => 'module_',
                'class' => Connection::class
            ],
        ],
    ],
],
```

应用程序数据表将会以 `main_` 作为前缀，而所有模块表都将以 `module_` 为前缀。 注意上面的配置并未合并；例如模块的组件都将启用查询缓存，因为这是默认的设置。

3.8.5 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。

3.9 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。

3.9.1 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 behaviors() 方法来声明过滤器，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 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 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 only 和 except 属性。

> **注意：** 在模块或应用主体中申明过滤器，在only 和 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。

3.9.2 创建过滤器

继承 yii\base\ActionFilter 类并覆盖 beforeAction() 或 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。 `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) - $this->_startTime;
        Yii::debug("Action '{$action->uniqueId}' spent $time second.");
        return parent::afterAction($action, $result);
    }
}
```

3.9.3 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。

AccessControl

AccessControl 提供基于 `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::class,
            '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[16] 或 OAuth 2[17] 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，`user identity class` 类必须 实现 `findIdentityByAccessToken()` 方法。

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

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::class,
        ],
    ];
}
```

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。

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 [
```

[16] https://zh.wikipedia.org/wiki/HTTP%E5%9F%BA%E6%9C%AC%E8%AE%A4%E8%AF%81
[17] https://oauth.net/2/

```
        [
            'class' => ContentNegotiator::class,
            '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 Content-Negotiator 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::class,
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ],
];
```

> **信息**： 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。

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::class,
            '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。

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::class,
            'only' => ['index'],
            'duration' => 60,
            'dependency' => [
                'class' => DbDependency::class,
                '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。

RateLimiter

RateLimiter 根据 漏桶算法[18] 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。

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。

[18] https://en.wikipedia.org/wiki/Leaky_bucket

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::class,
            'actions' => [
                'index'  => ['get'],
                'view'   => ['get'],
                'create' => ['get', 'post'],
                'update' => ['get', 'put', 'post'],
                'delete' => ['post', 'delete'],
            ],
        ],
    ];
}
```

Cors

跨域资源共享 CORS[19] 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 的 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。

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

如果要将CORS过滤器添加到你的 API 中的 `yii\rest\ActiveController` 类， 还要检查 REST Controllers 中的部分。

CROS过滤器可以通过 `$cors` 属性进行调整。

- `cors['Origin']`: 定义允许来源的数组，可为 `['*']`（任何用户）或 `['https://www.myserver.net', 'https://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']`。

[19] https://developer.mozilla.org/zh-CN/docs/Web/HTTP/CORS

- 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 https://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::class,
            'cors' => [
                'Origin' => ['https://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::class,
            'cors' => [
                'Origin' => ['https://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD',
                'OPTIONS'],
            ],
            'actions' => [
                'login' => [
                    'Access-Control-Allow-Credentials' => true,
                ]
            ]
        ],
    ], parent::behaviors());
}
```

3.10 小部件

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 `active form`，`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。

3.10.1 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',
    'dateFormat' => 'php:Y-m-d',
]) ?>
```

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。 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() 方法返回一个可组建小部件内容的小部件实例。

> **注意：** 当调用 yii\base\Widget::end() 的时候，一些小部件将使用 输出缓冲[20] 来调整封闭的内容。因此，当调用 yii\base\Widget::begin() 和 yii\base\Widget::end() 时，最好在同一个视图文件里。 不遵循这个规则可能会导致意外的输出。

配置全局默认值

小部件的全局默认值可以通过 DI 容器配置:

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

有关详细信息，请参阅 依赖注入容器“实践中的应用”一节 。

3.10.2 创建小部件

可以根据需要以两种不同方式创建小部件。

1: 使用 方法

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。 渲染结果可以直接“输出”或通过 run() 方法作为字符串返回。

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';
```

[20] https://www.php.net/manual/zh/book.outcontrol.php

```
        }
    }

    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']) ?>
```

有时小部件需要渲染很多内容，虽然你可以在 `run()` 方法中嵌入内容，但更好的方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');
}
```

2: 使用 和 方法

这类似于上面的有细微差别。 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。

> **信息：** 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(); ?>

    sample content that may contain one or more <strong>HTML</strong>
<pre>tags</pre>

    If this content grows too big, use sub views

    For e.g.

    <?php echo $this->render('viewfile'); // Note: here render() method is
of class \yii\base\View as this part of code is within view file and not in
Widget class file ?>

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

默认情况下，小部件的视图应存储在 `WidgetPath/views` 目录，`WidgetPath` 代表小部件类文件所在的目录。 假如小部件类文件在 `@app/components` 下， 上述示例会渲染 `@app/components/views/hello.php` 视图文件。 您可以覆盖 `yii\base\Widget::getViewPath()` 方法自定义视图文件所在路径。

3.10.3 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 来解决这个问题。

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。

3.11 资源

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。

3.11.1 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。

3.11.2 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
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
        ['css/print.css', 'media' => 'print'],
    ];
    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：

- `sourcePath`：指定包包含资源文件的根目录， 当根目录不能被 Web 访问时该属性应设置，否则，应设置 `basePath` 属性和 `baseUrl`。 路径别名 可在此处使用；
- `basePath`：指定包含资源包中资源文件并可Web访问的目录， 当指定 `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。

- baseUrl：指定对应到 basePath 目录的 URL， 和 basePath 类似，如果你指定 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。
- css：列出此包中包含的 CSS 文件的数组。 请注意，只应使用正斜杠“/”作为目录分隔符。每个文件都可以单独指定为字符串， 也可以与属性标记及其值一起指定在数组中。
- js：列出此包中包含的 JavaScript 文件的数组。 请注意，只应使用正斜杠“/”作为目录分隔符。 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 https://ajax.googleapis.com/ajax/libs/jquery/2.1 或 //ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js。
- depends：一个列出该资源包依赖的 其他资源包（后两节有详细介绍）。
- 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。
- 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。
- publishOptions：当调用 yii\web\AssetManager::publish() 发布该包资源文件到 Web 目录时 指定传递到该方法的选项，仅在指定了 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: 资源文件放在与你的 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basePath 和 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 basePath 而不是 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sourcePath属性。

> **注意：** source path 属性不要用 @webroot/assets，该路径默认

> 为 `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，也就是说 A 依赖 B，B 依赖 C，那么 A 也依赖 C。

资源选项

可指定 `cssOptions` 和 `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'];
```

这会使包中的 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];
```

默认情况下，当发布资源包时，所有在 `yii\web\AssetBundle::$sourcePath` 目录里的内容都会发布。 你可以通过配置 `publishOptions` 属性来自定义这种行为。 比如，为了只发布 `yii\web\AssetBundle::$sourcePath` 其中的某些内容或子目录里的内容， 可以在资源类中试试下面的做法:

```
<?php
namespace app\assets;

use yii\web\AssetBundle;

class FontAwesomeAsset extends AssetBundle
{
    public $sourcePath = '@bower/font-awesome';
    public $css = [
        'css/font-awesome.min.css',
    ];
    public $publishOptions = [
        'only' => [
            'fonts/',
            'css/',
        ]
    ];
}
```

上述的代码为 "fontawesome" package[21] 定义了资源包。 通过配置发布选项的 only 下标，只有 `fonts` 和 `css` 子目录会发布。

Bower 和 NPM 资源安装

大多数 JavaScript/CSS 包使用 Bower[22] 或 NPM[23] 来管理。 在 PHP 中，我们用 Composer 来管理 PHP 依赖。像 PHP 包一样， 也可以使用 `composer.json` 管理 Bower 和 NPM 包。

要实现这一点，需要配置一下 Composer 。有两种方法:

使用 asset-packagist 库 这种方式将满足大多数需要使用 NPM 或 Bower 包项目的要求。

> **注意:** 从 2.0.13 开始，基本和高级应用程序模板都默认配置使用 asset-packagist ， 因此你可以跳过本节。

在你项目 `composer.json` 文件中，添加下面几行代码:

```
"repositories": [
    {
        "type": "composer",
```

[21] https://fontawesome.com/
[22] https://bower.io/
[23] https://www.npmjs.com/

```
        "url": "https://asset-packagist.org"
    }
]
```

在你的 配置数组 中设置 `@npm` 和 `@bower` 的 别名：

```
$config = [
    ...
    'aliases' => [
        '@bower' => '@vendor/bower-asset',
        '@npm'   => '@vendor/npm-asset',
    ],
    ...
];
```

你可以访问 asset-packagist.org[24] 来了解它是如何工作的。

使用 fxp/composer-asset-plugin 与 asset-packagist 相比，composer-asset-plugin 不需要对应用程序配置进行任何更改。 而是需要运行以下命令来全局安装一个特殊的 Composer 插件：

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
```

这个命令会全局安装 composer asset plugin[25] 插件， 以便使用 Composer 来管理对 Bower 和 NPM 包的依赖。 在这个插件安装后， 你计算机上的每个项目都可以通过 `composer.json` 来管理 Bower 和 NPM 包。

如果你想要通过 Yii 来发布它们，将以下行添加到项目的 `composer.json` 文件中， 来调整 Bower 和 NPM 安装包的放置目录：

```
"config": {
    "asset-installer-paths": {
        "npm-asset-library": "vendor/npm",
        "bower-asset-library": "vendor/bower"
    }
}
```

> **注意：** 与 asset-packagist 相比，使用 `fxp/composer-asset-plugin` 的方式， 会显著减慢 `composer update` 命令的速度。

在配置好 Composer 支持使用 Bower 和 NPM 之后：

1. 编辑项目或扩展中的 `composer.json` 文件，在该文件的 `require` 字段中列举出相关包。 你应该使用 `bower-asset/PackageName`（对于 Bower 包）或者 `npm-asset/PackageName` （对于 NPM 包）的方式来引入这些包。

[24] https://asset-packagist.org
[25] https://github.com/fxpio/composer-asset-plugin

2. 运行 `composer update`
3. 创建资源类并列出在应用程序或扩展中所需使用的 JavaScript/CSS 文件。 你还需要配置 `sourcePath` 属性为 `@bower/PackageName` 或 `@npm/PackageName`。这是因为 Composer 会将 Bower 或 NPM 软件包安装在与此别名对应的目录中。

   注意： 某些包可能会将其所有发布文件放在子目录中。在这种情况下， 你应该把子目录设为 `sourcePath` 的值。例如， 在 `yii\web\JqueryAsset` 这个资源包中，使用 `@bower/jquery/dist` 而不是 `@bower/jquery`。

3.11.3 使用资源包

为使用资源包，先在视图中调用 `yii\web\AssetBundle::register()` 方法注册资源， 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
```

> **信息：** `yii\web\AssetBundle::register()` 方法返回资源包对象，该对象包含了发布资源的信息比如 `basePath` 或 `baseUrl`。

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。

动态资源包

作为常规 PHP 类，资源包可以承担一些与之相关的额外逻辑，并可以动态调整其内部参数。 比如：你也许用到某些复杂的 JavaScript 库，它提供一些用于国际化的包，并分成单独的源文件：每个文件对应于国际化中所支持的语言。 为了让这个 JavaScript 库正常工作，你需要把这些单独的语言源文件加载到页面中。 这可以通过覆盖 `yii\web\AssetBundle::init()` 方法来实现：

```
namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class SophisticatedAssetBundle extends AssetBundle
```

```
{
    public $sourcePath = '/path/to/sophisticated/src';
    public $js = [
        'sophisticated.js' // file, which is always used
    ];

    public function init()
    {
        parent::init();
        $this->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file
        added
    }
}
```

特定的资源包也可以通过 `yii\web\AssetBundle::register()` 返回的实例进行调整。 例如:

```
use app\assets\SophisticatedAssetBundle;
use Yii;

$bundle = SophisticatedAssetBundle::register(Yii::$app->view);
$bundle->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file
added
```

> **注意:** 虽然支持资源包的动态调整，但这是一种**不好**的做法，可能导致意想不到的副作用，应尽可能避免。

自定义资源包

Yii 通过配置名为 `assetManager` 的应用组件来使用 `yii\web\AssetManager`，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 CDN 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'
>  ]
> ],
> ```

可以将资源包名称设置为 false 来禁用一个或多个资源包， 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,
            ],
        ],
    ],
];
```

可以设置 yii\web\AssetManager::$bundles 为 false 来禁用 所有 的资源包。

需要记住，使用 yii\web\AssetManager::$bundles 进行自定义时，只在创建资源包时起作用，例如，在对象的构造函数阶段。 这意味着在此之后对该资源包对象所做的任何调整都将覆盖在配置数组中 yii\web\AssetManager::$bundles 的配置。 特别是：在 yii\web\AssetBundle::init() 方法内或在注册的资源包对象上进行的调整优先于 AssetManager 配置。 下面给个例子来说明在配置数组中对 yii\web\AssetManager::$bundles 的配置是不起作用的:

```
// Program source code:

namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class LanguageAssetBundle extends AssetBundle
{
    // ...

    public function init()
    {
```

```
        parent::init();
        $this->baseUrl = '@web/i18n/' . Yii::$app->language; // 将不能通过
        `AssetManager` 来管理！
    }
}
// ...

$bundle = \app\assets\LargeFileAssetBundle::register(Yii::$app->view);
$bundle->baseUrl = YII_DEBUG ? '@web/large-files':
'@web/large-files/minified'; // 将不能通过 `AssetManager` 来管理！


// Application config :

return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'app\assets\LanguageAssetBundle' => [
                    'baseUrl' => 'https://some.cdn.com/files/i18n/en' // 这
                    行代码不起作用！
                ],
                'app\assets\LargeFileAssetBundle' => [
                    'baseUrl' => 'https://some.cdn.com/files/large-files' //
                    这行代码不起作用！
                ],
            ],
        ],
    ],
];
```

资源映射

有时你想“修复”多个资源包中资源文件的错误或者不兼容，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， 但更好的方式是使用资源部署的特性来把不正确的资源部署为想要的， 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',
            ],
        ],
    ],
];
```

`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 `css` 和 `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， 也可为和 `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，`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 `basePath` 和 `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 `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。

清除缓存

对于运行在生产模式的 Web 应用程序，通常会为资源包和其他静态资源开启 HTTP 缓存。 但这种做法有个不好的地方就是，当你更新某个资源并部署到生产环境时， 客户端可能由于 HTTP 缓存而仍然使用旧版本的资源。 为了克服该不足，你可以试试清除缓存特性，它由 2.0.3 版本引入，只需如下配置 `yii\web\AssetManager` 即可：

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'appendTimestamp' => true,
        ],
    ],
];
```

通过上述配置后，每个已发布资源的 URL 都会附加一个最后更新时间戳的信息。 比如，`yii.js` 的 URL 可能是 `/assets/5515a87c/yii.js?v=1423448645"`，这里的参数 v 表示 `yii.js` 文件的最后更新时间戳。 现在一旦你更新了某个资源，它的 URL 也会改变进而强制客户端获取该资源的最新版本。

3.11.4 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。 有关 `yii.js` 的更多信息可以在 客户端脚本部分 中找到。
- `yii\web\JqueryAsset`：包含 jQuery Bower 包的 `jquery.js` 文件。
- `yii\bootstrap\BootstrapAsset`：包含 Twitter Bootstrap 框架的 CSS 文件。
- `yii\bootstrap\BootstrapPluginAsset`：包含 Twitter Bootstrap 框架的 JavaScript 文件 来支持 Bootstrap JavaScript 插件。
- `yii\jui\JuiAsset`：包含 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。

3.11.5 资源转换

除了直接编写 CSS 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[26] 或 SCSS[27]， 对于 JavaScript 可使用 TypeScript[28]。

可将使用扩展语法的资源文件列到资源包的 `css` 和 `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',
    ];
}
```

[26] https://lesscss.org/
[27] https://sass-lang.com/
[28] https://www.typescriptlang.org/

当在视图中注册一个这样的资源包，`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/Javascipt 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[29]： `.less`
- SCSS[30]： `.scss`
- Stylus[31]： `.styl`
- CoffeeScript[32]： `.coffee`
- TypeScript[33]： `.ts`

Yii 依靠安装的预处理工具来转换资源，例如， 为使用 LESS[34]，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（前面不需要这样的一个小点：.），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。

> **信息：** 除了以上方式，也有其他的方式来处理扩展语法资源，例如，可使用编译工具如grunt[35] 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。

[29] https://lesscss.org/
[30] https://sass-lang.com/
[31] https://stylus-lang.com/
[32] https://coffeescript.org/
[33] https://www.typescriptlang.org/
[34] https://lesscss.org/
[35] https://gruntjs.com/

3.11.6 合并和压缩资源

一个 Web 页面可以包含很多 CSS 和 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。

> **信息：** 合并和压缩资源通常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 `css` 和 `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 `css` 和 `js` 属性为空, 并设置它们的 `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，D）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，从而使 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 JavaScipt 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。

> **信息：** 将资源包分组并不是无意义的，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[36]， YUI Compressor[37]) 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

[36] https://developers.google.com/closure/compiler/
[37] https://github.com/yui/yuicompressor/

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 `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。

> **注意：** 这种资源合并的机制是基于 `yii\web\AssetManager::$bundles` 能够覆盖已经注册的资源包。 但是，正如前面提到的， 并不能覆盖到使用 `yii\web\AssetBundle::init()` 方法或

在资源包对象上进行调整的资源包。 你应该避免在资源合并时使用此类动态捆绑的资源包。

使用 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， 然后修改成你想要的模板。

```
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
<?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}',
    // 是否在压缩后删除资源来源:
    'deleteSource' => false,
    // 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` 和 `@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[38] 来合并 JavaScript 文件， 使用 YUI Compressor[39] 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。

> **注意：** 如果你使用 `yii\web\AssetManager::$bundles` 或 `yii\web\AssetManager::$assetMap` 来自定义应用程序的资源包，并希望将此自定义应用于压缩的源文件中， 你应该在 asset 命令配置文件中的 `assetManager` 部分包含这些自定义的内容。

> **注意：** 在指定压缩源时，应避免使用那些根据参数动态调整的资源包（即在 `init()` 方法或注册后根据参数进行动态调整的包）， 因为在压缩后它们可能无法正常工作。

> **信息：** 使用 `asset` 命令并不是合并和压缩资源的唯一方法。 你也可以使用能够自动运行设定任务的项目构建工具 grunt[40] 来实现同样的目的。

资源包分组

上一小节，介绍了如何压缩所有的资源包到一个文件， 减少对应用中引用资源文件的 HTTP 请求数，但是在实践中很少这样做。 比如，应用有一个“前端”和一个“后端”， 每一个都用了一个不同 JavaScript 和 CSS 文件集合。 在这种情况下，把所有的资源包压缩到一个文件毫无意义，“前端”不会用到“后端”的资源文件， 当请求“前端”页面时，“后端”的资源文件也会被发送过来，浪费网络带宽。

为了解决这个问题，可以吧资源包分成若干组，每个组里面有若干个资源包。 下面的配置展示了如何对资源包分组:

```
return [
    ...
    // Specify output bundles with groups:
```

[38] https://developers.google.com/closure/compiler/
[39] https://github.com/yui/yuicompressor/
[40] https://gruntjs.com/

```
    'targets' => [
        'allShared' => [
            'js' => 'js/all-shared-{hash}.js',
            'css' => 'css/all-shared-{hash}.css',
            'depends' => [
                // 包含由'backend' 和 'frontend' 共享的资源包
                'yii\web\YiiAsset',
                'app\assets\SharedAsset',
            ],
        ],
        'allBackEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [
                // 只包含 'backend' 资源:
                'app\assets\AdminAsset'
            ],
        ],
        'allFrontEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [], // 包含所有的剩余资源
        ],
    ],
    ...
];
```

如上所示，资源包分成了三个组：`allShared`，`allBackEnd` 和 `allFrontEnd` 它们每个都依赖各自的资源包集合。比如，`allBackEnd` 依赖 `app\assets\AdminAsset`。当对该配置运行 `asset` 命令时，将会根据各自依赖合并资源包。

> **信息：** 你也可以把某个分组的 `depends` 配置留空。这样做得话，这个分组将会依赖剩余的资源包，剩余资源包是指不被其他分组依赖的那些资源包。

3.12 扩展

扩展是专门设计的在 Yii 应用中随时可拿来使用的，并可重发布的软件包。例如，yiisoft/yii2-debug[41] 扩展在你的应用的每个页面底部添加一个方便用于调试的工具栏，帮助你简单地抓取页面生成的情况。你可以使用扩展来加速你的开发过程。

> **信息：** 本文中我们使用的术语 “扩展” 特指 Yii 软件包。而用术语 “软件包” 和 “库” 指代非 Yii 专用的通常意义上的软件包。

[41] https://github.com/yiisoft/yii2-debug

3.12.1 使用扩展

要使用扩展，你要先安装它。大多数扩展以 Composer[42] 软件包的形式发布， 这样的扩展可采取下述两个步骤来安装:

1. 修改你的应用的 `composer.json` 文件，指明你要安装的是哪个扩展（Composer 软件包）。
2. 运行 `composer install` 来安装指定的扩展。

注意如果你还没有安装 Composer[43] ，你需要先安装。

默认情况，Composer安装的是在 Packagist[44] 中 注册的软件包 - 最大的开源 Composer 代码库。你可以在 Packageist 中查找扩展。 你也可以 创建你自己的代码库[45] 然后配置 Composer 来使用它。 如果是在开发私有的扩展，并且想只在你的其他工程中共享时，这样做是很有用的。

通过 Composer 安装的扩展会存放在 `BasePath/vendor` 目录下，这里的 `BasePath` 指你的应用的 base path。因为 Composer 还是一个依赖管理器，当它安装一个包时， 也将安装这个包所依赖的所有软件包。

例如想安装 `yiisoft/yii2-imagine` 扩展，可按如下示例修改你的 `composer.json` 文件:

```
{
    // ...

    "require": {
        // ... other dependencies

        "yiisoft/yii2-imagine": "~2.0.0"
    }
}
```

安装完成后，你应该能在 `BasePath/vendor` 目录下见到 `yiisoft/yii2-imagine` 目录。你也应该 见到另一个 `imagine/imagine` 目录，在其中安装了所依赖的包。

> **信息:** `yiisoft/yii2-imagine` 是 Yii 由开发团队维护一个核心扩展， 所有核心扩展均由 Packagist[46] 集中管理，命名为 `yiisoft/yii2-xyz`，其中的 `xyz`， 不同扩展有不同名称。

现在你可以使用安装好的扩展了，好比是应用的一部分。如下示例展示了如何使用 `yiisoft/yii2-imagine` 扩展 提供的 `yii\imagine\Image` 类:

[42] https://getcomposer.org/
[43] https://getcomposer.org/
[44] https://packagist.org/
[45] https://getcomposer.org/doc/05-repositories.md#repository
[46] https://packagist.org/

```
use Yii;
use yii\imagine\Image;

// generate a thumbnail image
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
    ->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>
    50]);
```

> **信息：** 扩展类由 Yii class autoloader 自动加载。

手动安装扩展

在极少情况下，你可能需要手动安装一部分或者全部扩展，而不是依赖 Composer。 想做到这一点，你应当：

1. 下载扩展压缩文件，解压到 `vendor` 目录。
2. 如果有，则安装扩展提供的自动加载器。
3. 按指导说明下载和安装所有依赖的扩展。

如果扩展没有提供类的自动加载器，但也遵循了 PSR-4 standard[47] 标准，那么你可以使用 Yii 提供的类自动加载器来加载扩展类。 你需要做的仅仅是为扩展的根目录声明一个 root alias。 例如，假设在 `vendor/mycompany/myext` 目录中安装了一个扩展，并且扩展类的命名空间为 `myext` ， 那么你可以在应用配置文件中包含如下代码：

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

3.12.2 创建扩展

在你需要将你的杰作分享给其他人的时候，你可能会考虑创建一个扩展。扩展可包括任何你喜欢的代码，例如助手类、挂件、模块，等等。

建议你按照 Composer package[48] 的条款创建扩展，以便其他人更容易安装和使用。 就像上面的章节讲述的那样。

以下是将扩展创建为一个 Composer 软件包的需遵循的基本步骤。

1. 为你的扩展建一个工程，并将它存放在版本控制代码库中，例如 github.com[49] 。 扩展的开发和维护都应该在这个代码库中进行。
2. 在工程的根目录下，建一个 Composer 所需的名为 `composer.json` 的文件。 详情请参考后面的章节。

[47] https://www.php-fig.org/psr/psr-4/
[48] https://getcomposer.org/
[49] https://github.com

3. 在一个 Composer 代码库中注册你的扩展，比如在 Packagist[50] 中，以便其他 用户能找到以及用 Composer 安装你的扩展。

每个 Composer 软件包在根目录都必须有一个 `composer.json` 文件。该文件包含软件包的元数据。 你可以在 Composer手册[51] 中找到完整关于该文件的规格。 以下例子展示了 `yiisoft/yii2-imagine` 扩展的 `composer.json` 文件。

```
{
    // 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": "https://forum.yiiframework.com/",
        "wiki": "https://www.yiiframework.com/wiki/",
        "irc": "ircs://irc.libera.chat:6697/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "authors": [
        {
            "name": "Antonio Ramirez",
            "email": "amigo.cobos@gmail.com"
        }
    ],

    // package dependencies
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // class autoloading specs
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

包名 每个 Composer 软件包都应当有一个唯一的包名以便能从其他

[50] https://packagist.org/
[51] https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup

的软件包中识别出来。 包名的格式为 `vendorName/projectName` 。例如在包名 `yiisoft/yii2-imagine` 中，vendor 名和 project 名分别是 `yiisoft` 和 `yii2-imagine` 。

不要用 `yiisoft` 作为你的 vendor 名，由于它被 Yii 的核心代码预留使用了。

我们推荐你用 `yii2-` 作为你的包名的前缀，表示它是 Yii 2 的扩展，例如，`myname/yii2-mywidget`。 这更便于用户辨别是否是 Yii 2 的扩展。

包类型 将你的扩展指明为 `yii2-extension` 类型很重要，以便安装的时候能被识别出是一个 Yii 扩展。

当用户运行 `composer install` 安装一个扩展时， `vendor/yiisoft/extensions.php` 文件会被自动更新使之包含新扩展的信息。从该文件中， Yii 应用程序就能知道安装了 哪些扩展 (这些信息可通过 `yii\base\Application::$extensions` 访问)。

依赖 你的扩展依赖于 Yii （理所当然）。因此你应当在 `composer.json` 文件中列出它 (`yiisoft/yii2`)。如果你的扩展还依赖其他的扩展或者是第三方库，你也要一并列出来。 确定你也为每一个依赖的包列出了适当的版本约束条件 (比如 `1.*`, `@stable`) 。 当你发布一个稳定版本时，你所依赖的包也应当使用稳定版本。

大多数 JavaScript/CSS 包是用 Bower[52] 来管理的，而非 Composer。你可使用 Composer asset 插件[53] 使之可以 通过 Composer 来管理这类包。如果你的扩展依赖 Bower 软件包，你可以如下例所示那样简单地 在 `composer.json` 文件的依赖中列出它。

```
{
    // package dependencies
    "require": {
        "bower-asset/jquery": ">=1.11.*"
    }
}
```

上述代码表明该扩展依赖于 `jquery` Bower 包。一般来说，你可以在 `composer.json` 中用 `bower-asset/PackageName` 指定 Bower 包，用 `npm-asset/PackageName` 指定 NPM 包。 当 Compower 安装 Bower 和 NPM 软件包时，包的内容默认会分别安装到 `@vendor/bower/PackageName` 和 `@vendor/npm/Packages` 下。这两个目录还可以分别用 `@bower/PackageName` 和 `@npm/PackageName` 别名指向。

关于 asset 管理的详细情况，请参照 Assets 章节。

类的自动加载 为使你的类能够被 Yii 的类自动加载器或者 Composer 的类自动加载器自动加载，你应当在 `composer.json` 中指定 `autoload` 条目，如下所示:

[52] https://bower.io/
[53] https://github.com/fxpio/composer-asset-plugin

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

你可以列出一个或者多个根命名空间和它们的文件目录。

当扩展安装到应用中后，Yii 将为每个所列出根命名空间创建一个 别名 指向命名空间对应的目录。 例如，上述的 `autoload` 条目声明将对应于别名 `@yii/imagine`。

推荐的做法

扩展意味着会被其他人使用，你在开发中通常需要额外的付出。 下面我们介绍一些通用的及推荐的做法，以创建高品质的扩展。

命名空间 为避免冲突以及使你的扩展中的类能被自动加载，你的类应当使用命名空间， 并使类的命名符合 PSR-4 standard[54] 或者 PSR-0 standard[55] 标准。

你的类的命名空间应以 `vendorName\extensionName` 起始，其中 `extensionName` 和项目名相同，除了它没有 `yii2-` 前缀外。例如，对 `yiisoft/yii2-imagine` 扩展 来说，我们用 `yii\imagine` 作为它的类的命名空间。

不要使用 `yii`、`yii2` 或者 `yiisoft` 作为你的 vendor 名。这些名称已由 Yii 内核代码预留使用了。

类的自举引导 有时候，你可能想让你的扩展在应用的 自举过程 中执行一些代码。 例如，你的扩展可能想响应应用的 `beginRequest` 事件，做一些环境的设置工作。 虽然你可以指导扩展的使用者显式地将你的扩展中的事件句柄附加（绑定）到 `beginRequest` 事件， 但是更好的方法是自动完成。

为实现该目标，你可以创建一个所谓 *bootstrapping class* （自举类）实现 `yii\base\BootstrapInterface` 接口。 例如，

```
namespace myname\mywidget;

use yii\base\BootstrapInterface;
use yii\base\Application;

class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
```

[54] https://www.php-fig.org/psr/psr-4/
[55] https://www.php-fig.org/psr/psr-0/

```
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // do something here
        });
    }
}
```

然后你将这个类在 `composer.json` 文件中列出来，如下所示，

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

当这个扩展安装到应用后，Yii 将在每一个请求的自举过程中 自动实例化自举类并调用其 `bootstrap()` 方法。

操作数据库 你的扩展可能要存取数据库。不要假设使用你的扩展的应用总是用 `Yii::$db` 作为数据库连接。你应当在需要访问数据库的类中申明一个 `db` 属性。 这个属性允许你的扩展的用户可定制你的扩展使用哪个 DB 连接。例如， 你可以参考 `yii\caching\DbCache` 类看一下它是如何申明和使用 `db` 属性的。

如果你的扩展需要创建特定的数据库表，或者修改数据库结构，你应当

- 提供 数据迁移 来操作数据库的结构修改，而不是使用SQL文本文件；
- 尽量使迁移文件适用于不同的 DBMS；
- 在迁移文件中避免使用 Active Record。

使用 Assets 如果你的扩展是挂件或者模块类型，它有可能需要使用一些 assets 。 例如，一个模块可能要显示一些包含图片，JavaScript 和 CSS 的页面。因为扩展的文件 都是放在同一个目录之下，安装之后 Web 无法读取，你有两个选择使得这些 asset 文件目录 可以通过 Web 读取：

- 让扩展的用户手动将这些 asset 文件拷贝到特定的 Web 可以读取的文件夹；
- 申明一个 asset bundle 并依靠 asset 发布机制自动将这些文件（asset bundle 中列出的文件） 拷贝到 Web 可读的文件夹。

我们推荐你使用第二种方法，以便其他人能更容易使用你的扩展。 更详细的关于如何处理 assets ，请参照 Assets 章节。

国际化和本地化 你的扩展可能会在支持不同语言的应用中使用！因此，如果你的扩展要显示内容给终端用户， 你应当试着实现 国际化和本地化，特别地，

- 如果扩展为终端用户显示信息，这些信息应该用 `Yii::t()` 包装起来，以便可以进行翻译。 只给开发者参考的信息（如内部异常信息）不需要做翻译。

- 如果扩展显示数字、日期等，你应该用 `yii\i18n\Formatter` 中适当的格式化规则做格式化处理。

更详细的信息，请参照 Internationalization 章节。

测试 你一定想让你的扩展可以无暇地运行而不会给其他人带来问题和麻烦。为达到这个目的， 你应当在公开发布前做测试。

推荐你创建测试用例，做全面覆盖的测试你的扩展，而不只是依赖于手动测试。 每次发布新版本前，你只要简单地运行这些测试用例确保一切完好。 Yii 提供了测试支持，使你更容易写单元测试、验收测试和功能测试。详情请参照 Testing 章节。

版本控制 你应该为每一个扩展定一个版本号（如 `1.0.1`）。我们推荐你命名版本号时参照 semantic versioning[56] 决定用什么样的版本号。

发布 为使其他人知道你的扩展，你应该公开发布。

如果你首次发布一个扩展，你应该在 Composer 代码库中注册它，例如 Packagist[57]。之后，你所需要做的仅仅是在 版本管理库中创建一个 tag（如`v1.0.1`），然后通知 Composer 代码库。 其他人就能查找到这个新的发布了，并可通过 Composer 代码库安装和更新该扩展。

在发布你的扩展时，除了代码文件，你还应该考虑包含如下内容 帮助其他人了解和使用你的扩展：

- 根目录下的 readme 文件：它描述你的扩展是干什么的以及如何安装和使用。 我们推荐你用 Markdown[58] 的格式 来写并将文件命名为 `readme.md`。
- 根目录下的修改日志文件：它列举每个版本的发布做了哪些更改。该文件可以用 Markdown 根式 编写并命名为 `changelog.md`。
- 根目录下的升级文件：它给出如何从其他就版本升级该扩展的指导。该文件可以用 Markdown 根式 编写并命名为 `changelog.md`。
- 入门指南、演示代码、截屏图示等：如果你的扩展提供了许多功能，在 readme 文件中不能完整 描述时，就要用到这些文件。
- API 文档：你的代码应当做好文档，让其他人更容易阅读和理解。 你可以参照 Object class file[59] 学习如何为你的代码做文档。

  信息： 你的代码注释可以写成 Markdown 格式。`yiisoft/yii2-apidoc` 扩展为你提供了一个从你的 代码注释生成漂亮的 API 文档。

  信息： 虽然不做要求，我们还是建议你的扩展遵循某个编码规范。 你可以参照 core framework code style[60]。

[56] https://semver.org
[57] https://packagist.org/
[58] https://daringfireball.net/projects/markdown/
[59] https://github.com/yiisoft/yii2/blob/master/framework/base/Object.php
[60] https://github.com/yiisoft/yii2/blob/master/docs/internals/core-code-style.md

3.12.3 核心扩展

Yii 提供了下列核心扩展，由 Yii 开发团队开发和维护。这些扩展全都在 Packagist[61] 中注册，并像 Using Extensions 章节描述 的那样容易安装。

- yiisoft/yii2-apidoc[62]： 提供了一个可扩展的、高效的 API 文档生成器。核心框架的 API 文档也是用它生成的。
- yiisoft/yii2-authclient[63]： 提供了一套常用的认证客户端，例如 Facebook OAuth2 客户端、GitHub OAuth2 客户端。
- yiisoft/yii2-bootstrap[64]： 提供了一套挂件，封装了 Bootstrap[65] 的组件和插件。
- yiisoft/yii2-debug[66]： 提供了对 Yii 应用的调试支持。当使用该扩展是， 在每个页面的底部将显示一个调试工具条。 该扩展还提供了一个独立的页面，以显示更详细的调试信息。
- yiisoft/yii2-elasticsearch[67]： 提供对 Elasticsearch[68] 的使用支持。它包含基本的查询/搜索支持， 并实现了 Active Record 模式让你可以将活动记录 存储在 Elasticsearch 中。
- yiisoft/yii2-faker[69]： 提供了使用 Faker[70] 的支持，为你生成模拟数据。
- yiisoft/yii2-gii[71]： 提供了一个基于页面的代码生成器，具有高可扩展性，并能用来快速生成模型、 表单、模块、CRUD 等。
- yiisoft/yii2-httpclient[72]： 提供 HTTP 客户端。
- yiisoft/yii2-imagine[73]： 提供了基于 Imagine[74] 的常用图像处理功能。
- yiisoft/yii2-jui[75]： 提供了一套封装 JQuery UI[76] 的挂件以及它们的交互。
- yiisoft/yii2-mongodb[77]： 提供了对 MongoDB[78] 的使用支持。它包含基本 的查询、活动记录、数据迁移、缓存、代码生成等特性。
- yiisoft/yii2-queue[79]： 通过队列异步提供运行任务的支持。 它支持基于 DB，Redis，RabbitMQ，AMQP，Beanstalk 和 Gearman 的队

[61] https://packagist.org/
[62] https://github.com/yiisoft/yii2-apidoc
[63] https://github.com/yiisoft/yii2-authclient
[64] https://github.com/yiisoft/yii2-bootstrap
[65] https://getbootstrap.com/
[66] https://github.com/yiisoft/yii2-debug
[67] https://github.com/yiisoft/yii2-elasticsearch
[68] https://www.elastic.co/
[69] https://github.com/yiisoft/yii2-faker
[70] https://github.com/fzaninotto/Faker
[71] https://github.com/yiisoft/yii2-gii
[72] https://github.com/yiisoft/yii2-httpclient
[73] https://github.com/yiisoft/yii2-imagine
[74] https://imagine.readthedocs.org/
[75] https://github.com/yiisoft/yii2-jui
[76] https://jqueryui.com/
[77] https://github.com/yiisoft/yii2-mongodb
[78] https://www.mongodb.com/
[79] https://www.yiiframework.com/extension/yiisoft/yii2-queue

列。

- yiisoft/yii2-redis[80]： 提供了对 redis[81] 的使用支持。它包含基本的 查询、活动记录、缓存等特性。
- yiisoft/yii2-shell[82]： 提供基于 psysh[83] 的交互式 shell。
- yiisoft/yii2-smarty[84]： 提供了一个基于 Smarty[85] 的模板引擎。
- yiisoft/yii2-sphinx[86]： 提供了对 Sphinx[87] 的使用支持。它包含基本的 查询、活动记录、代码生成等特性。
- yiisoft/yii2-swiftmailer[88]： 提供了基于 swiftmailer[89] 的邮件发送功能。
- yiisoft/yii2-twig[90]： 提供了一个基于 Twig[91] 的模板引擎。

以下官方扩展适用于 Yii 2.1 及以上版本。 您不需要为 Yii 2.0 安装它们，因为它们包含在核心框架中。

- yiisoft/yii2-captcha[92]： 提供 CAPTCHA。
- yiisoft/yii2-jquery[93]： 为 jQuery[94] 提供支持。
- yiisoft/yii2-maskedinput[95]： 提供基于 jQuery Input Mask plugin[96] 的格式化输入小部件。
- yiisoft/yii2-mssql[97]： 提供对使用 MSSQL[98] 的支持。
- yiisoft/yii2-oracle[99]： 提供对使用 Oracle[100] 的支持。
- yiisoft/yii2-rest[101]： 提供对 REST API 的支持。

[80] https://github.com/yiisoft/yii2-redis
[81] https://redis.io/
[82] https://www.yiiframework.com/extension/yiisoft/yii2-shell
[83] https://psysh.org/
[84] https://github.com/yiisoft/yii2-smarty
[85] https://www.smarty.net/
[86] https://github.com/yiisoft/yii2-sphinx
[87] https://sphinxsearch.com
[88] https://github.com/yiisoft/yii2-swiftmailer
[89] https://swiftmailer.symfony.com/
[90] https://github.com/yiisoft/yii2-twig
[91] https://twig.symfony.com/
[92] https://www.yiiframework.com/extension/yiisoft/yii2-captcha
[93] https://www.yiiframework.com/extension/yiisoft/yii2-jquery
[94] https://jquery.com/
[95] https://www.yiiframework.com/extension/yiisoft/yii2-maskedinput
[96] https://robinherbots.github.io/Inputmask/
[97] https://www.yiiframework.com/extension/yiisoft/yii2-mssql
[98] https://www.microsoft.com/sql-server/
[99] https://www.yiiframework.com/extension/yiisoft/yii2-oracle
[100] https://www.oracle.com/
[101] https://www.yiiframework.com/extension/yiisoft/yii2-rest

Chapter 4

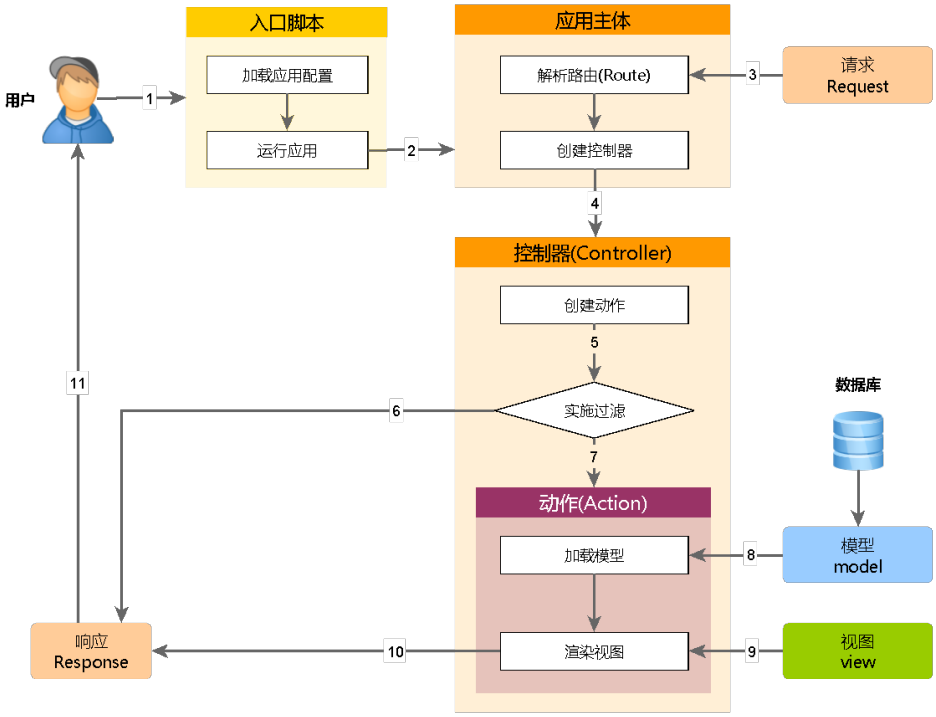
请求处理（Handling Requests）

4.1 运行机制概述

每一次 Yii 应用开始处理 HTTP 请求时，它都会进行一个近似的流程。

1. 用户提交指向 入口脚本 `web/index.php` 的请求。
2. 入口脚本会加载 配置数组 并创建一个 应用 实例用于处理该请求。
3. 应用会通过 request（请求） 应用组件 解析被请求的 路由。
4. 应用创建一个 controller（控制器） 实例具体处理请求。
5. 控制器会创建一个 action（动作） 实例并为该动作执行相关的 Filters（访问过滤器）。
6. 如果任何一个过滤器验证失败，该动作会被取消。
7. 如果全部的过滤器都通过，该动作就会被执行。
8. 动作会加载一个数据模型，一般是从数据库中加载。
9. 动作会渲染一个 View（视图），并为其提供所需的数据模型。
10. 渲染得到的结果会返回给 response（响应） 应用组件。
11. 响应组件会把渲染结果发回给用户的浏览器。

下面的示意图展示了应用是如何处理一个请求的。



在这个版块中，我们会更加详细地描述某些步骤的具体运作。

4.2 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 `preInit()`（预初始化）方法，配置一些高优先级的应用属性，比如 `basePath` 属性。
2. 注册错误处理器（`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 `init()`（初始化）方法，它会顺次调用 `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。

4.3 路由

当入口脚本在调用 `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动作处理这个请求。 该过程就被称为引导路由（*routing*）。 路由相反的操作会将给定的路由和参数生成一个可访问的URL地址， 这个操作叫做创建*URL*。 创建出来的URL被请求的时候，路由处理器可以解析成原始的路由信息和参数。

负责路由解析和创建URL的组件是 `URL`管理器，URL管理器在程序组件中被注册成 `urlManager`。 `URL`管理器 提供方法 `parseRequest()` 来 解析请求的URL并返回路由信息和参数， 方法 `createUrl()` 用来根据提供的路由和参数创建一个可访问的URL。

在程序配置中配置 `urlManager` 组件，可以让你的应用不改变现有代码的情况下 识别任意的URL格式。 例如使用下面的代码创建一个到 `post/view` 控制器的 URL：

```
use yii\helpers\Url;

// Url::to() 将调用 UrlManager::createUrl() 来创建URL
$url = Url::to(['post/view', 'id' => 100]);
```

根据 `urlManager` 中的配置，创建出来的URL可能看起来像是以下的一种格式（或者其它的格式）。 如果此URL被访问，将被解析成原来的路由和参数。

```
/index.php?r=post%2Fview&id=100
/index.php/post/100
/posts/100
```

4.3.1 URL 格式化

URL管理器提供两种URL格式:

- 默认URL格式;
- 美化URL格式。

默认URL格式使用一个参数r表示路由， 并且使用一般的参数格式表示请求参数。例如，/index.php?r=post/view&id=100表示路由为post/view， 参数id为100。 默认URL格式不需要为URL管理器做任何配置， 并且在任何Web服务器都可以正常使用。

美化URL格式在脚本名称后面使用更多的路径信息表示路由和参数信息。 例如，用适当的URL规则，/index.php/post/100中附加的路径信息/post/100表示 路由为post/view，参数id为100。 要使用美化的URL格式，你需要根据实际的需求 设计一组URL规则来规定URL的样式。

你可以仅设置URL管理器中的开启美化URL来切换两种URL格式， 而不必改动任何程序代码。

4.3.2 路由

路由处理包含两个步骤:

- 请求被解析成一个路由和关联的参数;
- 路由相关的一个控制器动作被创建出来处理这个请求。

如果使用默认URL格式，解析请求到路由只是简单的从GET请求中得到命名为r的参数。

当使用用美化URL格式时，URL管理器将检查注册的URL规则， 找到一条可以匹配的将请求转到路由的规则。 如果找不到任何匹配的规则，系统将抛出yii\web\NotFoundHttpException异常。

一旦请求解析成路由，系统将马上根据路由信息创建一个控制器动作。路由信息根据/分解成多个部分。例如，site/index将被分解成site和index两部分。 每个部分都可能被认为是一个模块、控制器或动作的ID。 从路由的第一个部分开始，系统将执行以下步骤创建所需模块（如果有模块的话）、控制器和动作:

1. 设置应用系统作为当前的模块。
2. 检查当前模块中的控制器映射是否存在当前ID。 如果存在，根据控制器映射中的定义创建一个控制器实例， 跳到步骤5处理路由剩下的部分。
3. 检查ID是否为当前模块下modules定义的子模块 如果是，创建对应子模块， 跳到步骤2使用刚创建的子模块处理路由下一部分。
4. 将ID作为一个控制器ID并创建一个控制器实例， 并用来处理路由剩下的部分。
5. 控制器在自己的动作映射中查找当前ID。 如果找到，根据映射中的定义创建一个动作。 如果没找到，控制器将尝试根据动作ID定义的动作方法创建一个行内动作。

在上面的步骤中，如果出现任何错误，系统将抛出一个yii\web\NotFoundHttpException异常， 表示路由处理程序出现的错误信息。

缺省路由

如果传入请求并没有提供一个具体的路由，（一般这种情况多为于对首页的请求）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 动作。你可以像这样在应用配置中调整该属性的值:

```
return [
    // ...
    'defaultRoute' => 'main/index',
];
```

和应用系统中的缺省路由类似，模块中也存在一个缺省路由，所以如果 有一个user模块，当请求到user模块时，模块的缺省路由 被用来决定缺省的控制器。默认的缺省控制器名称是default。如果在缺省路由中没有指定任何动作， 缺省动作将被用来决定缺省的动作。 在这个例子中，完整的路由应该是user/default/index。

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 yii\web\Application::$catchAll 属性:

```
return [
    // ...
    'catchAll' => ['site/offline'],
];
```

根据上面的配置，site/offline控制器将被用来处理所有的请求。

catchAll属性应该配置成一个数组，第一个元素为路由， 剩下的元素为键值对格式的动作的参数。

> **信息:** 如果此属性被设置，开发环境中的调试工具条[1] 将被停用。

4.3.3 创建 URLs

Yii提供了一个助手方法yii\helpers\Url::to()，用来根据提供的路由和参数创建各种各样的URL。 例如:

[1] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md

```
use yii\helpers\Url;

// 创建一个普通的路由URL: /index.php?r=post%2Findex
echo Url::to(['post/index']);

// 创建一个带路由参数的URL: /index.php?r=post%2Fview&id=100
echo Url::to(['post/view', 'id' => 100]);

// 创建一个带锚定的URL: /index.php?r=post%2Fview&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// 创建一个绝对路径URL: https://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], true);

// 创建一个带https协议的绝对路
径URL: https://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], 'https');
```

注意上面的例子中，我们假定使用默认的URL格式。如果启用美化的URL格式， 创建出来的URL根据使用的URL规则可能有所不同。

方法yii\helpers\Url::to()传入的路由是上下文相关的。 根据以下规则确认传入的路由是一个相对的路由还是绝对的路由：

- 如果路由是一个空字符串，则使用当前请求的路由；
- 如果路由中不存在/，则被认为是一个当前控制器下的动作ID， 且路由被附加到当前控制器的唯一ID后面；
- 如果路由不以/开头，被认为是当前模块下的路由， 路由将被附加到当前模块的唯一ID后面。

从版本 2.0.2 开始，你可以使用根据别名中定义的别名路由。如果是这种情况， 别名将首先被转化为实际的路由，然后根据上面的规则转化成一个绝对路由。

例如，假设当前的模块为admin，当前控制器为post，

```
use yii\helpers\Url;

// 当前请求路由: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['']);

// 只有动作ID的相对路由: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['index']);

// 相对路由: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['post/index']);

// 绝对路由: /index.php?r=post%2Findex
echo Url::to(['/post/index']);

// 假设有一个"/post/index"的别名"@posts": /index.php?r=post%2Findex
echo Url::to(['@posts']);
```

方法 yii\helpers\Url::to() 实际上调用了 URL管理器 中的 createUrl()

方法 和 `createAbsoluteUrl()` 方法。 下面的介绍中，我们将介绍如何配置 URL管理器 来创建自定义的 URL 格式。

方法 `yii\helpers\Url::to()` 同时支持创建和任何路由不相关的 URL。这种情况下，第一个参数不再传入一个数组，而是传入一个字符串。例如：

```
use yii\helpers\Url;

// 当前请求URL: /index.php?r=admin%2Fpost%2Findex
echo Url::to();

// 设定了别名的URL: https://example.com
Yii::setAlias('@example', 'https://example.com/');
echo Url::to('@example');

// 绝对URL: https://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

除了 `to()` 方法，`yii\helpers\Url` 助手类同时提供了多个其它创建 URL 的方法。 例如：

```
use yii\helpers\Url;

// 主页URL: /index.php?r=site%2Findex
echo Url::home();

// 根URL，如果程序部署到一个Web目录下的子目录时非常有用
echo Url::base();

// 当前请求的权威规范URL
// 参考 https://en.wikipedia.org/wiki/Canonical_link_element
echo Url::canonical();

// 记住当前请求的URL并在以后获取
Url::remember();
echo Url::previous();
```

4.3.4 使用美化的 URL

要使用美化的URL，像下面这样在应用配置中配置`urlManager`组件：

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

开启美化URL 属性被用来切换是否启用美化URL格式。 虽然除了开启美化URL 属性以外其它属性都是可选的，但是上面的配置是最常用到的。

- 是否显示脚本名称： 此属性决定创建的URL中是否包含入口脚本名称。 例如，默认的 `/index.php/post/100`，开启此属性后将创建成 `/post/100`。
- 是否开启严格解析：此属性决定是否开启严格的请求解析。 如果设置为启用，请求的URL必须至少匹配 规则 中设定的一条规则作为正确请求， 否则系统将抛出 `yii\web\NotFoundHttpException` 异常。 如果严格解析被关闭，当 规则 中没有任何一条匹配时， 请求URL中的路径信息将被作为请求路由使用。
- 规则：此属性包含一个规则列表，用来规定如何解析和创建URL。 这是一个主要属性，你应该根据特定的应用环境配置此属性用来生成特定格式的URL。

> **注意：** 如果你想在URL中隐藏入口脚本名称，除了要设置 `showScriptName` 为 false， 同时应该配置 Web 服务，处理当请求 URL 没有特殊指定入口脚本时确定要执行哪个PHP文件，如果你使用 Apache Web server，你可以参考安装中推荐的配置。

URL规则

一个URL规则是类 `yii\web\UrlRule` 或子类的一个实例。每个URL规则包含一个匹配URL中的路径、路由和少量参数的规则。 如果请求地址匹配一个URL规则，则此规则可以用来解析此请求。 如果生成URL时路由和参数符合一个URL规则，则此规则可以用来生成此URL。

如果开启了美化URL格式，URL管理器使用定义的规则解析请求和创建URL。 尤其注意，URL管理器按照规则中定义的顺序依次检测请求地址， 当找到第一条匹配的规则时停止。 匹配到的规则将被用来解析请求URL到指定的路由和参数。 同样的，创建URL的时候，URL管理器 查找第一条匹配的的规则并用来生成URL。

你可以配置 `yii\web\UrlManager::$rules` 为一个数组，键为匹配规则，值为路由。 每条键值对为一条URL规则。例如，下面的 规则 配置了两条URL规则。第一条规则匹配URL `posts` 映射到路由 `post/index`。 第二条规则匹配符合正则表达式 `post/(\d+)` 的URL并映射到路由 `post/view`，同时包含 一个参数 `id`。

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

> **信息：** 规则中的匹配模式用来匹配URL中的路径信息。例如，`/index.php/post/100?source=ad` 中的路径信息为 `post/100`（开始和结尾处的 `/` 将被忽略）和模式 `post/(\d+)` 匹配。

除了定义 URL 规则外，你还可以将规则定义为配置数组。 每个配置数组用来配置一个单独的 URL 规则对象。如果你需要配置 URL 规则的其它参数时可以这样用。 例如：

```
[
    // ...其它 URL 规则...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

如果你在URL规则中不配置 `class` 选项，默认将使用类 `yii\web\UrlRule`。

命名参数

一条URL规则可以对匹配模式中的参数设置格式为 `<ParamName:RegExp>` 的命名， 其中 `ParamName` 指定参数的名称，`RegExp` 是可选的用来匹配参数值得正则表达式。 如果没有设置 `RegExp`，表示参数值为不包含 `/` 的字符串。

> **注意：** 你可以仅针对参数设置正则表达式，其余部分设置普通文本。

当一条规则用来匹配URL时，符合匹配规则的相关的参数值被填充到规则中， 并且这些参数可以在 `request` 组件中使用 `$_GET` 获取到。 当规则用来创建 URL 时， 提供的参数值将被插入到规则定义的指定位置。

让我们使用一些例子来说明命名参数是如何工作的。假设我们定义了以下三条 URL 规则：

```
[
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

当规则用来解析 URL 时：

- 根据第二条规则，`/index.php/posts` 被解析成路由 `post/index`；
- 根据第一条规则，`/index.php/posts/2014/php` 被解析成路由 `post/index`，参数 `year` 的值是 2014，参数 `category` 的值是 `php`；
- 根据第三条规则，`/index.php/post/100` 被解析成路由 `post/view`， 参数 `id` 的值是 100；
- 当`yii\web\UrlManager::$enableStrictParsing` 设置为 `true` 时，`/index.php/posts/php` 将导致一个`yii\web\NotFoundHttpException` 异常， 因为无法匹配任何规则。如果 `yii\web\UrlManager::$enableStrictParsing` 设为 `false`（默认值）， 路径部分 `posts/php` 将被作为路由。

当规则用来生成 URL 时：

- 根据第二条规则 `Url::to(['post/index'])` 生成 `/index.php/posts`；
- 根据第一条规则 `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` 生成 `/index.php/posts/2014/php`；
- 根据第三条规则 `Url::to(['post/view', 'id' => 100])` 生成 `/index.php/post/100`；
- 根据第三条规则 `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` 生成 `/index.php/post/100?source=ad`。 因为参数 `source` 在规则中没有指定，将被作为普通请求参数附加到生成的 URL 后面。
- `Url::to(['post/index', 'category' => 'php'])` 生成 `/index.php/post/index?category=php`。注意因为没有任何规则适用，将把路由信息当做路径信息来生成URL，并且所有参数作为请求查询参数附加到 URL 后面。

参数化路由

你可以在 URL 规则中嵌入参数名称，这样可以允许一个 URL 规则用来匹配多个路由。 例如，下面的规则在路由中嵌入了 `controller` 和 `action` 两个参数。

```
'rules' => [
    '<controller:(post|comment)>/create' => '<controller>/create',
    '<controller:(post|comment)>/<id:\d+>/<action:(update|delete)>' =>
    '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

解析 URL `/index.php/comment/100/update` 时，第二条规则适用，设置参数 `controller` 为 `comment`， 设置参数 `action` 为 `update`。自然的，路由 `<controller>/<action>` 变成了 `comment/update`。

同样的，根据路由 `comment/index` 创建 URL 时，最后一条规则适用，将生成 URL `/index.php/comments`。

> **信息：** 使用参数化的路由，可以显著的减少 URL 规则的数量， 可以显著提高URL管理器的效率。

默认参数值

默认的，所有规则中定义的参数都是必须的。如果一个请求 URL 不存在其中一个参数， 或者创建URL时没有指定其中一个参数，则无法应用此规则。 如果需要设置某些参数为可选的，必须设置规则的默认值属性。 此属性中列出的参数将变成可选的且在没有指定时会使用此处设置的默认值。

下面设置的规则中，参数 `page` 和 `tag` 都是可选的，当没有指定时将分别使用值 1 和空字符串。

```
[
    // ...其它规则...
    [
        'pattern' => 'posts/<page:\d+>/<tag>',
```

```
        'route' => 'post/index',
        'defaults' => ['page' => 1, 'tag' => ''],
    ],
]
```

上面的规则可以用来解析或创建下面的 URL:

- `/index.php/posts`: `page` 为 1，`tag` 为 ''。
- `/index.php/posts/2`: `page` 为 2，`tag` 为 ''。
- `/index.php/posts/2/news`: `page` 为 2，`tag`为 `'news'`。
- `/index.php/posts/news`: `page` 为 1，`tag` 为 `'news'`。

如果不使用可选参数，你必须创建 4 条规则才可以实现相同的效果。

> **注意**： 如果 `pattern` 中仅包含可选参数和斜杠， 只有所有参数被忽略时第一个参数才被忽略。

带服务名称的规则

可以在URL规则中设置Web服务的名称，如果你需要使你的应用程序在不同的Web服务名称下表现不同的话。 例如，下面的规则将URL`https://admin.example.com/login`解析成路由`admin/user/login`， URL`https://www.example.com/login`解析成路由`site/login`。

```
[
    'https://admin.example.com/login' => 'admin/user/login',
    'https://www.example.com/login' => 'site/login',
]
```

你还可以在服务名称中嵌入参数用来动态的提取服务名称。例如，下面的规则 将URL`https://en.example.com/posts`解析成路由`post/index`且参数`language=en`。

```
[
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

从版本 2.0.11 开始，你还可以使用不带协议类型的模式来同时匹配 `http` 和 `https`。 规则语法和上面相比只是忽略掉 `http:` 部分，例如： `'//www.example.com/login' => 'site/login'`。

> **注意**： 带服务名称的规则**不应该**包含任何子目录。例如，如果程序入口脚本在 `https://www.example.com/sandbox/blog/index.php`，应该使用 `https://www.example.com/posts` 代替 `https://www.example.com/sandbox/blog/posts`。这样才可以将你的应用部署到任何目录而不需要更改 URL 规则。Yii 将会自动的检测应用程序所在的根目录。

URL 后缀

你可能因为各种目的需要在 URL 后面添加后缀。例如，你可以在URL后面添加 `.html` 让其看起来像是一个 HTML 页面； 也可以添加 `.json` 用来表明需要的返回值内容类型。 可以参考下面的系统配置， 通过设置 `yii\web\UrlManager::$suffix` 属性来达到此目的:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'suffix' => '.html',
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

上面的配置允许URL管理器识别或生成带 `.html` 后缀的 URL。

> **提示：** 你可以设置URL后缀为 `/` 让所有的 URL 以斜线结束。

> **注意：** 当你配置 URL 后缀时，如果请求的 URL 没有此后缀，系统将认为此 URL 无法识别。 这是 SEO（搜索引擎优化）的最佳实践。

有时你可能需要在不同的URL使用不同的后缀。可以通过在不同的URL规则下不同的设置后缀属性。 URL规则中此属性将覆盖在URL管理器中设置的值。 例如，下面的配置中全局使用 `.html` 后缀，但是定义了一个自定义的使用 `.json` 为后缀的规则。

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
]
```

HTTP 方法

当使用 RESTful 接口时，经常需要根据 HTTP 请求方法将同样的URL解析到不同的路由。 可以容易的通过将支持的 HTTP 方法设置为 URL 规则的前缀来实现这个目的。 如果一个规则需要支持多种 HTTP 方法，可以将方法名称用逗号隔开。 例如，下面的规则有相同的模式 `post/<id:\d+>` 但是支

持不同的 HTTP 方法。 一个 PUT post/100 请求将被解析到 post/update，GET post/100 请求将被解析到 post/view。

```
'rules' => [
    'PUT,POST post/<id:\d+>' => 'post/update',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

> **注意：** 如果一个 URL 规则包含 HTTP 方法，这个规则将只能用来解析请求，除非 GET 请求明确被指定在 HTTP 方法中， 否则创建 URL 时此规则将被URL管理器忽略。

> **提示：** 为了简化 RESTful 接口的路由定义，Yii 提供了一个特殊的URL规则类 yii\rest\UrlRule 支持高效的且支持一些设想中的功能，像自动多元化控制器 ID。 更多信息，请参考 RESTful 接口说明中的路由章节。

动态添加规则

URL规则可以动态添加到URL管理器。如果模块需要管理自己的URL规则时很有必要。 如果需要使路由处理过程中动态添加的规则可用，你应该在应用程序启动引导时添加。 对模块来说，需要实现 yii\base\BootstrapInterface 接口的 bootstrap() 方法， 类似下面这样动态添加规则：

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // 规则在这里定义
    ], false);
}
```

注意你需要同时在 yii\web\Application::bootstrap() 中指定这些模块，这样模块才可以参与到 启动引导过程中。

创建规则类

尽管默认的 yii\web\UrlRule 类已经足够灵活可以处理大部分项目了， 有时还是会需要创建一个自定义的规则类。例如，在一个汽车经销网站，你可能会需要使用 这样的URL格式 /Manufacturer/Model，Manufacturer 和 Model 必须同时匹配保存在数据库中的一些数据。 默认的规则类只能使用静态定义而无法适应此种情况。

我们可以创建一个自定义的 URL 规则类来解决这个问题。

```
<?php

namespace app\components;
```

```
use yii\web\UrlRuleInterface;
use yii\base\BaseObject;

class CarUrlRule extends BaseObject implements UrlRuleInterface
{
    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 false; // this rule does not apply
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\w+)(/(\w+))?$%', $pathInfo, $matches)) {
            // 检查 $matches[1] 和 $matches[3]
            // 确认是否匹配到一个数据库中保存的厂家和型号。
            // 如果匹配，设置参数 $params['manufacturer'] 和 / 或
            $params['model']
            // 返回 ['car/index', $params]
        }
        return false; // 本规则不会起作用
    }
}
```

在 `yii\web\UrlManager::$rules` 配置中设置新定义的规则类：

```
'rules' => [
    // ...其它规则...
    [
        'class' => 'app\components\CarUrlRule',
        // ...配置其它参数...
    ],
]
```

4.3.5 URL规范化

从 2.0.10 版开始`Url`管理器可以配置用`URL`规范器来处理 相同URL的不同格式，例如是否带结束斜线。因为技术上来说 `https://example.com/path` 和 `https://example.com/path/` 是完全不同的 URL，两个地址返回相同的内容会导致SEO排名降低。 默认情况下 URL 规范器会合并连续的斜线，根据配置决定是否添加或删除结尾斜线， 并且会使用永久重定向[2]将地址重新跳转到规范化后的URL。 URL规范器可以针对URL管理器全局配置，也可以针

[2] https://zh.wikipedia.org/wiki/HTTP_301

对规则单独配置 - 默认每个规则都使用URL管理器中的规范器。 你可以针对特定的URL规则设置 `UrlRule::$normalizer` 为 `false` 来关闭规范化。

下面的例子显示了一个URL规范器的配置:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'enableStrictParsing' => true,
    'suffix' => '.html',
    'normalizer' => [
        'class' => 'yii\web\UrlNormalizer',
        // 调试时使用临时跳转代替永久跳转
        'action' => UrlNormalizer::ACTION_REDIRECT_TEMPORARY,
    ],
    'rules' => [
        // ...其它规则...
        [
            'pattern' => 'posts',
            'route' => 'post/index',
            'suffix' => '/',
            'normalizer' => false, // 针对此规则关闭规范器
        ],
        [
            'pattern' => 'tags',
            'route' => 'tag/index',
            'normalizer' => [
                // 针对此规则不合并连续的斜线
                'collapseSlashes' => false,
            ],
        ],
    ],
]
```

> **注意:** 默认 `UrlManager::$normalizer` 规范器是关闭的。你需要明确配置其开启 来启用 URL 规范化。

4.3.6 性能考虑

在开发复杂的 Web 应用程序时，优化 URL 规则非常重要，以便解析请求和创建 URL 所需 的时间更少。

通过使用参数化路由，您可以减少 URL 规则的数量，这可以显著提高性能。

当解析或创建URL时，`URL manager` 按照它们声明的顺序检查 URL 规则。 因此，您可以考虑调整 URL 规则的顺序，以便在较少使用的规则之前放置更具体和/或更常用的规则。

如果多个 URL 规则使用相同的前缀，你可以考虑使用 `yii\web\GroupUrlRule`，这样作为一个组合，`URL`管理器会更高效。 特别是当应用程序由模块组合而成时，每个模块都有各自的 URL 规则且都有各自的模块 ID 作为前缀。

4.4 请求

一个应用的请求是用 `yii\web\Request` 对象来表示的，该对象提供了诸如请求参数（译者注：通常是GET参数或者POST参数）、HTTP头、cookies等信息。 默认情况下，对于一个给定的请求，你可以通过 `request` application component 应用组件（`yii\web\Request` 类的实例） 获得访问相应的请求对象。在本章节，我们将介绍怎样在你的应用中使用这个组件。

4.4.1 请求参数

要获取请求参数，你可以调用 `request` 组件的 `get()` 方法和 `post()` 方法。他们分别返回 `$_GET` 和 `$_POST` 的值。例如，

```
$request = Yii::$app->request;

$get = $request->get();
// 等价于: $get = $_GET;

$id = $request->get('id');
// 等价于: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// 等价于: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// 等价于: $post = $_POST;

$name = $request->post('name');
// 等价于: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// 等价于: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

> **信息：** 建议你像上面那样通过 `request` 组件来获取请求参数，而不是 直接访问 `$_GET` 和 `$_POST`。 这使你更容易编写测试用例，因为你可以伪造数据来创建一个模拟请求组件。

当实现 RESTful APIs 接口的时候，你经常需要获取通过 PUT， PATCH 或者其他的 request methods 请求方法提交上来的参数。你可以通过调用 `yii\web\Request::getBodyParam()` 方法来获取这些参数。例如，

```
$request = Yii::$app->request;

// 返回所有参数
$params = $request->bodyParams;

// 返回参数 "id"
$param = $request->getBodyParam('id');
```

> **信息：** 不同于 `GET` 参数，`POST`，`PUT`，`PATCH` 等等这些提交上来的参数是在请求体中被发送的。 当你通过上面介绍的方法访问

这些参数的时候，request 组件会解析这些参数。 你可以通过配置 yii\web\Request::$parsers 属性来自定义怎样解析这些参数。

4.4.2 请求方法

你可以通过 Yii::$app->request->method 表达式来获取当前请求使用的HTTP方法。 这里还提供了一整套布尔属性用于检测当前请求是某种类型。 例如，

```
$request = Yii::$app->request;

if ($request->isAjax) { /* 该请求是一个 AJAX 请求 */ }
if ($request->isGet)  { /* 请求方法是 GET */ }
if ($request->isPost) { /* 请求方法是 POST */ }
if ($request->isPut)  { /* 请求方法是 PUT */ }
```

4.4.3 请求URLs

request 组件提供了许多方式来检测当前请求的 URL。

假设被请求的 URL 是 https://example.com/admin/index.php/product?id=100，你可以像下面描述的那样获取 URL 的各个部分：

- url：返回 /admin/index.php/product?id=100，此 URL 不包括主机信息部分。
- absoluteUrl：返回 https://example.com/admin/index.php/product?id=100，包含host infode的整个URL。
- hostInfo：返回 https://example.com，只有主机信息部分。
- pathInfo：返回 /product， 这个是入口脚本之后，问号之前（查询字符串）的部分。
- queryString：返回 id=100，问号之后的部分。
- baseUrl：返回 /admin，主机信息之后， 入口脚本之前的部分。
- scriptUrl：返回 /admin/index.php，没有路径信息和查询字符串部分。
- serverName：返回 example.com，URL 中的主机名。
- serverPort：返回 80，这是 web 服务中使用的端口。

4.4.4 HTTP头

你可以通过 yii\web\Request::$headers 属性返回的 header collection 获取HTTP头信息。 例如，

```
// $headers 是一个 yii\web\HeaderCollection 对象
$headers = Yii::$app->request->headers;

// 返回 Accept header 值
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* 这是一个 User-Agent 头 */ }
```

请求组件也提供了支持快速访问常用头的方法，包括：

- `userAgent`：返回 `User-Agent` 头。
- `contentType`：返回 `Content-Type` 头的值， `Content-Type` 是请求体中MIME类型数据。
- `acceptableContentTypes`：返回用户可接受的内容MIME类型。 返回的类型是按照他们的质量得分来排序的。得分最高的类型将被最先返回。
- `acceptableLanguages`：返回用户可接受的语言。 返回的语言是按照他们的偏好层次来排序的。第一个参数代表最优先的语言。

假如你的应用支持多语言，并且你想在终端用户最喜欢的语言中显示页面， 那么你可以使用语言协商方法 `yii\web\Request::getPreferredLanguage()`。这个方法通过 `acceptableLanguages` 在你的应用中所支持的语言列表里进行比较筛选，返回最适合的语言。

> **提示：** 你也可以使用 `ContentNegotiator` 过滤器进行动态确定哪些内容类型和语言应该在响应中使用。 这个过滤器实现了上面介绍的内容协商的属性和方法。

4.4.5 客户端信息

你可以通过 `userHost` 和 `userIP` 分别获取主机名和客户机的 IP 地址， 例如，

```
$userHost = Yii::$app->request->userHost;
$userIP = Yii::$app->request->userIP;
```

4.4.6 受信任的代理和报头

在上一节中，您已经了解了如何获取主机和 IP 地址等用户信息。 这将在正常设置中开箱即用，其中使用单个网络服务器为网站提供服务。 然而，如果您的 Yii 应用程序在反向代理后面运行，则需要添加其他配置来检索此信息， 因为直接客户端现在是代理， 并且用户 IP 地址通过代理设置的报头传递给 Yii 应用程序。

除非您明确信任代理，否则不应盲目信任代理提供的报头。 从 2.0.13 开始，Yii 支持通过 `request` 组件的 `trustedHosts`， `secureHeaders`， `ipHeaders` 和 `secureProtocolHeaders` 属性配置可信代理。

以下是在反向代理数组后面运行的应用程序的请求配置， 它们位于 `10.0.2.0/24` IP 网络中：

```
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24',
    ],
],
```

默认情况下，IP 由代理在 `X-Forwarded-For` 头中发送，协议（“http”或“https”）在 `X-Forwarded-Proto` 中发送。

如果您的代理使用不同的报头，您可以使用请求配置来调整它们，例如：

```
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24' => [
            'X-ProxyUser-Ip',
            'Front-End-Https',
        ],
    ],
    'secureHeaders' => [
        'X-Forwarded-For',
        'X-Forwarded-Host',
        'X-Forwarded-Proto',
        'X-Proxy-User-Ip',
        'Front-End-Https',
    ],
    'ipHeaders' => [
        'X-Proxy-User-Ip',
    ],
    'secureProtocolHeaders' => [
        'Front-End-Https' => ['on']
    ],
],
```

使用上面的配置，`secureHeaders` 中列出的所有报头都会从请求中过滤掉，除了 `X-ProxyUser-Ip` 和 `Front-End-Https` 报头，以防请求由代理发出。 在这种情况下，前者用于检索 `ipHeaders` 中配置的用户IP， 后者将用于确定 `yii\web\Request::getIsSecureConnection()` 的结果。

4.5 响应

当一个应用在处理完一个请求后, 这个应用会生成一个 `response` 响应对象并把这个响应对象发送给终端用户 这个响应对象包含的信息有 HTTP 状态码，HTTP 头和主体内容等, 从本质上说，网页应用开发最终的目标就是根据不同的请求去构建这些响应对象。

在大多数实际应用情况下，你应该主要地去处理 `response` 这个 应用组件， 在默认情况下，它是一个继承自 `yii\web\Response` 的实例 然而，Yii 也允许你创建自己的响应对象并发送给终端用户，这方面在后续会阐述。

在本节，我们将会讲述如何组装和构建响应并把它发送给终端用户。

4.5.1 状态码

构建响应要做的第一件事就是声明请求是否被成功处理，我们可通过设置 `yii\web\Response::$statusCode` 这个属性来做到这一点，该属性接受一

个有效的 HTTP 状态码[3]。例如，表明该请求已被成功处理， 可以设置状态码为 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`：状态码 400。
- `yii\web\ConflictHttpException`：状态码 409。
- `yii\web\ForbiddenHttpException`：状态码 403。
- `yii\web\GoneHttpException`：状态码 410。
- `yii\web\MethodNotAllowedHttpException`：状态码 405。
- `yii\web\NotAcceptableHttpException`：状态码 406。
- `yii\web\NotFoundHttpException`：状态码 404。
- `yii\web\ServerErrorHttpException`：状态码 500。
- `yii\web\TooManyRequestsHttpException`：状态码 429。
- `yii\web\UnauthorizedHttpException`：状态码 401。
- `yii\web\UnsupportedMediaTypeHttpException`：状态码 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);
```

4.5.2 HTTP 头部

可在 `response` 组件中操控 `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');
```

> **信息：** 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。

[3] https://tools.ietf.org/html/rfc2616#section-10

4.5.3 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 format 和 data 属性，format 属性指定 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。

- HTML：通过 yii\web\HtmlResponseFormatter 来实现。
- XML：通过 yii\web\XmlResponseFormatter 来实现。
- JSON：通过 yii\web\JsonResponseFormatter 来实现。
- JSONP：通过 yii\web\JsonResponseFormatter 来实现。
- RAW：如果要直接发送响应而不应用任何格式，请使用此格式。

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 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，除了使用默认的 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。

4.5.4 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('https://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。

除了动作方法外，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('https://example.com/new', 301)->send();
```

> **信息：** `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 URId 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。

> **信息：** 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。

4.5.5 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。

- `yii\web\Response::sendFile()`：发送一个已存在的文件到客户端
- `yii\web\Response::sendContentAsFile()`：发送一个文本字符串作为文件到客户端
- `yii\web\Response::sendStreamAsFile()`：发送一个已存在的文件流作为文件到客户端

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[4]
- Lighttpd v1.4: X-LIGHTTPD-send-file[5]
- Lighttpd v1.5: X-Sendfile[6]
- Nginx: X-Accel-Redirect[7]
- Cherokee: X-Sendfile and X-Accel-Redirect[8]

4.5.6 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。

[4] https://tn123.org/mod_xsendfile
[5] https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[6] https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[7] https://www.nginx.com/resources/wiki/start/topics/examples/xsendfile/
[8] https://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

2. 调用 `yii\web\Response::prepare()` 来格式化 `response data` 为 `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。

4.6 Sessions 和 Cookies

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。

4.6.1 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`open()` 和`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

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) ...
```

> **信息:** 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\web\DbSession`：存储 session 数据在数据表中
- `yii\web\CacheSession`：存储 session 数据到缓存中，缓存和配置中的缓存组件相关
- `yii\redis\Session`：存储 session 数据到以 redis[9] 作为存储媒介中
- `yii\mongodb\Session`：存储 session 数据到 MongoDB[10]。

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。

> **注意：** 如果使用自定义会话存储，则可能需要显式配置会话垃圾收集器。 一些安装的 PHP（例如 Debian）使用垃圾收集器概率为 0，并在计划任务中离线清理会话文件。 此过程不适用于您的自定义存储， 因此您需要配置 `yii\web\Session::$GCProbability` 以使用非零值。

[9] https://redis.io/
[10] https://www.mongodb.com/

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。

或者，可以通过以下迁移完成：

```
<?php

use yii\db\Migration;

class m170529_050554_create_table_session extends Migration
{
    public function up()
    {
        $this->createTable('{{%session}}', [
            'id' => $this->char(64)->notNull(),
            'expire' => $this->integer(),
            'data' => $this->binary()
        ]);
        $this->addPrimaryKey('pk-id', '{{%session}}', 'id');
    }
```

```
    public function down()
    {
        $this->dropTable('{{%session}}');
    }
}
```

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。

提示： 要显示 Flash 消息，您可以通过以下方式使用 `yii\bootstrap\Alert` 小部件：

```
echo Alert::widget([
   'options' => ['class' => 'alert-info'],
   'body' => Yii::$app->session->getFlash('postDeleted'),
]);
```

4.6.2 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。

控制器是直接处理请求和响应的部分。 因此, 应当在控制器中读取和发送 cookie 。 (译者注：意思在控制器中处理 cookie 是安全的。)

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

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 name 和 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 domain，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[11]。

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 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 验证时，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',
        ],
    ],
];
```

> **信息：** 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。

[11] https://owasp.org/www-community/HttpOnly

4.7 错误处理

Yii 内置了一个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；

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。

4.7.1 使用错误处理器

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\ErrorException;

try {
    10/0;
} catch (ErrorException $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 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();
```

4.7.2 自定义错误显示

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。

> **信息**： 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。

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 errorView 和 exceptionView 属性 使用自定义的错误显示视图。

使用错误动作

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 errorAction 属性， 类似如下:

```
return [
    'components' => [
        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
    ]
];
```

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 动作如下所示:

```
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 动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，如果错误动作定义为yii\web\ErrorAction， 可以访问该动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。

  注意: 如果你使用 基础应用模板 或 高级应用模板, 错误动作和错误视图已经定义好了。

  注意: 如果您需要在错误处理程序中重定向，请执行以下操作:

  ```
  Yii::$app->getResponse()->redirect($url)->send();
  return;
  ```

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
    }
}
```

4.8 日志

Yii提供了一个强大的日志框架，这个框架具有高度的可定制性和可扩展性。使用这个框架， 你可以轻松地记录各种类型的消息，过滤它们， 并且将它们收集到不同的目标，诸如文件，数据库，邮件。

使用Yii日志框架涉及下面的几个步骤：

- 在你代码里的各个地方记录 log messages;
- 在应用配置里通过配置 log targets 来过滤和导出日志消息;
- 检查由不同的目标导出的已过滤的日志消息（例如：Yii debugger）。

在这部分，我们主要描述前两个步骤。

4.8.1 日志消息

记录日志消息就跟调用下面的日志方法一样简单：

- `Yii::trace()`：记录一条消息去跟踪一段代码是怎样运行的。这主要在开发的时候使用。
- `Yii::info()`：记录一条消息来传达一些有用的信息。
- `Yii::warning()`：记录一个警告消息用来指示一些已经发生的意外。
- `Yii::error()`：记录一个致命的错误，这个错误应该尽快被检查。

这些日志记录方法针对 严重程度 和 类别 来记录日志消息。 它们共享相同的函数签名 `function ($message, $category = 'application')`，`$message`代表要被 记录的日志消息，而 `$category` 是日志消息的类别。在下面的示例代码中，在默认的类别 `application` 下 记录了一条跟踪消息：

```
Yii::trace('start calculating average revenue');
```

> **注意：** 日志消息可以是字符串，也可以是复杂的数据，诸如数组或者对象。 log targets 的义务是正确处理日志消息。默认情况下， 假如一条日志消息不是一个字符串，它将被导出为一个字符串，通过调用 `yii\helpers\VarDumper::export()`。

为了更好地组织和过滤日志消息，我们建议您为每个日志消息指定一个适当的类别。您可以为类别选择一个分层命名方案， 这将使得 log targets 在基于它们的分类来过滤消息变得更加容易。 一个简单而高效的命名方案是使用PHP魔术常量 `__METHOD__` 作为分类名称。 这种方式也在Yii框架的核心代码中得到应用， 例如，

```
Yii::trace('start calculating average revenue', __METHOD__);
```

`__METHOD__` 常量计算值作为该常量出现的地方的方法名（完全限定的类名前缀）。 例如，假如上面那行代码在这个方法内被调用，则它将等于字符串 `'app\controllers\RevenueController::calculate'`。

> **注意：** 上面所描述的日志方法实际上是 `logger object` 对象（一个通过表达式 `Yii::getLogger()` 可访问的单例） 的方法 `log()` 的一个快捷方式。当足够的消息被记录或者当应用结束时， 日志对象将会调用一个 `message dispatcher` 调度对象将已经记录的日志消息发送到已注册的 log targets 目标中。

4.8.2 日志目标

一个日志目标是一个 `yii\log\Target` 类或者它的子类的实例。 它将通过他们的严重层级和类别来过滤日志消息，然后将它们导出到一些媒介中。 例如，一个 `database target` 目标导出已经过滤的日志消息到一个数据的表里面， 而一个 `email target`目标将日志消息导出到指定的邮箱地址里。

在一个应用里，通过配置在应用配置里的 `log` application component，你可以注册多个日志目标。 就像下面这样：

```
return [
    // 必须在引导期间加载“log”组件
    'bootstrap' => ['log'],
    // “log”组件处理带时间戳的消息。 设置 PHP 时区以创建正确的时间戳
    'timeZone' => 'PRC',
    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error'],
                    'categories' => ['yii\db\*'],
                    'message' => [
                       'from' => ['log@example.com'],
                       'to' => ['admin@example.com',
                       'developer@example.com'],
                       'subject' => 'Database errors at example.com',
                    ],
                ],
            ],
        ],
    ],
];
```

> **注意：** `log` 组件必须在 bootstrapping 期间就被加载，以便于它能够及时调度日志消息到目标里。 这是为什么在上面的代码中，它被列在 `bootstrap` 数组中的原因。

在上面的代码中，在 `yii\log\Dispatcher::$targets` 属性里有两个日志目标被注册：

- 第一个目标选择的是错误和警告层级的消息，并且在数据库表里保存他们；
- 第二个目标选择的是错误层级的消息并且是在以 `yii\db\` 开头的分类下，并且在一个邮件里将它们发送到 `admin@example.com` 和 `developer@example.com`。

Yii配备了以下的内建日志目标。请参考关于这些类的API文档， 并且学习怎样配置和使用他们。

- `yii\log\DbTarget`：在数据库表里存储日志消息。

- yii\log\EmailTarget：发送日志消息到预先指定的邮箱地址。
- yii\log\FileTarget：保存日志消息到文件中.
- yii\log\SyslogTarget：通过调用PHP函数 syslog() 将日志消息保存到系统日志里。

下面，我们将描述所有日志目标的公共特性。

消息过滤

对于每一个日志目标，你可以配置它的 levels 和 categories 属性来指定哪个消息的严重程度和分类目标应该处理。

levels 属性是由一个或者若干个以下值组成的数组：

- error：相应的消息通过 Yii::error() 被记录。
- warning：相应的消息通过 Yii::warning() 被记录。
- info：相应的消息通过 Yii::info() 被记录。
- trace：相应的消息通过 Yii::trace() 被记录。
- profile：相应的消息通过 Yii::beginProfile() 和 Yii::endProfile() 被记录。更多细节将在 Profiling 分段解释。

如果你没有指定 levels 的属性， 那就意味着目标将处理 任何 严重程度的消息。

categories 属性是一个包含消息分类名称或者模式的数组。 一个目标将只处理那些在这个数组中能够找到对应的分类或者其中一个相匹配的模式的消息。 一个分类模式是一个以星号 * 结尾的分类名前缀。假如一个分类名与分类模式具有相同的前缀， 那么该分类名将和分类模式相匹配。例如， yii\db\Command::execute 和 yii\db\Command::query 都是作为分类名称运用在 yii\db\Command 类来记录日志消息的。 它们都是匹配模式 yii\db*。

假如你没有指定 categories 属性， 这意味着目标将会处理 任何 分类的消息。

除了通过 categories 属性设置白名单分类，你也可以通过 except 属性来设置某些分类作为黑名单。假如一条消息的分类在这个属性中被发现或者是匹配其中一个， 那么它将不会在目标中被处理。

在下面的目标配置中指明了目标应该只处理错误和警告消息，当分类的名称匹配 yii\db* 或者是 yii\web\HttpException:* 的时候， 但是除了 yii\web\HttpException:404。

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

> **注意：** 当一个HTTP异常通过 error handler 被捕获的时候，一个错误消息将以 `yii\web\HttpException:ErrorCode` 这样的格式的分类名被记录下来。例如，`yii\web\NotFoundHttpException` 将会引发一个分类是 `yii\web\HttpException:404` 的 错误消息。

消息格式化

日志目标以某种格式导出过滤过的日志消息。例如， 假如你安装一个 `yii\log\FileTarget` 类的日志目标， 你应该能找出一个日志消息类似下面的 `runtime/log/app.log` 文件：

```
2014-10-04 18:10:15 [::1][][-][trace][yii\base\Module::getModule] Loading
module: debug
```

默认情况下，日志消息将被格式化，格式化的方式遵循 `yii\log\Target::formatMessage()`：

```
Timestamp [IP address][User ID][Session ID][Severity Level][Category]
Message Text
```

你可以通过配置 `yii\log\Target::$prefix` 的属性来自定义格式，这个属性是一个 PHP 可调用体返回的自定义消息前缀。 例如，下面的代码配置了一个日志目标的前缀是每个日志消息中当前用户的 ID（IP 地址和 Session ID 被删除是由于隐私的原因）。

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') :
        null;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

除了消息前缀以外，日志目标也可以追加一些上下文信息到每组日志消息中。 默认情况下，这些全局的PHP变量的值被包含在：`$_GET`，`$_POST`，`$_FILES`，`$_COOKIE`，`$_SES` 和 `$_SERVER` 中。 你可以通过配置 `yii\log\Target::$logVars` 属性适应这个行为， 这个属性是你想要通过日志目标包含的全局变量名称。 举个例子，下面的日志目标配置指明了只有 `$_SERVER` 变量的值将被追加到日志消息中。

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

你可以将 `logVars` 配置成一个空数组来完全禁止上下文信息包含。 或者假如你想要实现你自己提供上下文信息的方式， 你可以重写 `yii\log\Target::getContextMessage()` 方法。

消息跟踪级别

在开发的时候，通常希望看到每个日志消息来自哪里。这个是能够被实现的，通过配置 `log` 组件的 `traceLevel` 属性， 就像下面这样：

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

上面的应用配置设置了 `traceLevel` 的层级，假如 `YII_DEBUG` 开启则是3，否则是0。 这意味着，假如 `YII_DEBUG` 开启，每个日志消息在日志消息被记录的时候， 将被追加最多3个调用堆栈层级；假如 `YII_DEBUG` 关闭， 那么将没有调用堆栈信息被包含。

> **注意：** 获得调用堆栈信息并不是不重要。因此， 你应该只在开发或者调试一个应用的时候使用这个特性。

消息刷新和导出

如上所述，通过 `logger object` 对象，日志消息被保存在一个数组里。 为了这个数组的内存消耗，当数组积累了一定数量的日志消息， 日志对象每次都将刷新被记录的消息到 log targets 中。 你可以通过配置 `log` 组件的 `flushInterval` 属性来自定义数量：

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,   // default is 1000
            'targets' => [...],
        ],
    ],
];
```

> **注意：** 当应用结束的时候，消息刷新也会发生，这样才能确保日志目标能够接收完整的日志消息。

当 `logger object` 对象刷新日志消息到 log targets 的时候，它们并 不能立即获取导出的消息。相反，消息导出仅仅在一个日志目标累积了一定数量的过滤消息的时候才会发生。你可以通过配置 个别的 log targets 的 `exportInterval` 属性来 自定义这个数量，就像下面这样：

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,  // default is 1000
]
```

因为刷新和导出层级的设置，默认情况下，当你调用 `Yii::trace()` 或者任何其他的记录方法，你将不能在日志目标中立即看到日志消息。 这对于一些长期运行的控制台应用来说可能是一个问题。为了让每个日志消息在日志目标中能够立即出现， 你应该设置 `flushInterval` 和 `exportInterval` 都为1， 就像下面这样：

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

注意： 频繁的消息刷新和导出将降低你的应用性能。

切换日志目标

你可以通过配置 `enabled` 属性来开启或者禁用日志目标。 你可以通过日志目标配置去做，或者是在你的代码中放入下面的PHP申明：

```
Yii::$app->log->targets['file']->enabled = false;
```

上面的代码要求您将目标命名为 `file`，像下面展示的那样， 在 `targets` 数组中使用字符串键：

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

从版本 2.0.13 开始，您可以使用可调用的配置 `enabled`， 以定义是否应启用日志目标的动态条件。 有关示例，请参阅 `yii\log\Target::setEnabled()` 文档。

创建新的目标

创建一个新的日志目标类非常地简单。你主要需要实现 `yii\log\Target::export()` 方法来发送 `yii\log\Target::$messages` 数组的 内容到一个指定的媒体中。你可以调用 `yii\log\Target::formatMessage()` 方法去格式化每个消息。 更多细节，你可以参考任何一个包含在 Yii 发行版中的日志目标类。

> **提示：** 您可以使用 PSR log target extension[12] 尝试任何兼容 PSR-3 的日志记录器， 例如 Monolog[13]， 而不是创建自己的日志记录器。

4.8.3 性能分析

性能分析是一个特殊的消息记录类型，它通常用在测量某段代码块的时间， 并且找出性能瓶颈是什么。举个例子，`yii\db\Command` 类 使用性能分析找出每个数据库查询的时间。

为了使用性能分析，首先确定需要进行分析的代码块。 然后像下面这样围住每个代码块:

```
\Yii::beginProfile('myBenchmark');

...code block being profiled...

\Yii::endProfile('myBenchmark');
```

这里的 `myBenchmark` 代表一个唯一标记来标识一个代码块。之后当你检查分析结果的时候， 你将使用这个标记来定位对应的代码块所花费的时间。

对于确保 `beginProfile` 和 `endProfile` 对能够正确地嵌套，这是很重要的。 例如，

```
\Yii::beginProfile('block1');

    // some code to be profiled

    \Yii::beginProfile('block2');
        // some other code to be profiled
    \Yii::endProfile('block2');

\Yii::endProfile('block1');
```

假如你漏掉 `\Yii::endProfile('block1')` 或者切换了 `\Yii::endProfile('block1')` 和 `\Yii::endProfile('block2')` 的 顺序，那么性能分析将不会工作。

对于每个被分析的代码块，一个带有严重程度为 `profile` 的日志消息将被记录。 你可以配置一个 log target 去收集这些 消息，并且导出他们。Yii debugger 有一个内建的性能分析面板能够展示分析结果。

[12] https://github.com/samdark/yii2-psr-log-target
[13] https://github.com/Seldaek/monolog

Chapter 5

关键概念（Key Concepts）

5.1 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` 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\BaseObject::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：

```
<?php

namespace yii\components\MyClass;

use yii\base\BaseObject;

class MyClass extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... 在应用配置之前初始化

        parent::__construct($config);
    }

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

        // ... 应用配置后进行初始化
    }
}
```

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::class,
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> **信息：** 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\BaseObject` 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 `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（即一个对象），那么它就已经初始化就绪可供使用。

5.2 属性（Properties）

在 PHP 中，类的成员变量也被称为属性。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，常常会想用一个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\BaseObject` 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\BaseObject` 或其子类即可。

> **信息:** 几乎每个 Yii 框架的核心类都继承自 `yii\base\BaseObject` 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\BaseObject;

class Foo extends BaseObject
{
    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， 在这个例子里，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 `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。
- 对不确定有无魔术方法（getter 或 setter）的属性正常调用 `property_exists()` 将不会生效。你应该分别调用 `canGetProperty()` 或 `canSetProperty()` 。

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。

5.3 事件（Events）

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。

5.3.1 事件处理器（Event Handlers）

事件处理器是一个PHP 回调函数[1]， 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：

- `event name`：事件名
- `event sender`：调用 `trigger()` 方法的对象
- `custom data`：附加事件处理器时传入的数据，默认为空，后文详述

5.3.2 附加事件处理器（Attaching Event Handlers）

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
});
```

你也可以通过 配置 附加事件处理器。 请 参考 配置的格式 小节了解更多.

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_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

[1] https://www.php.net/manual/zh/language.types.callable.php

5.3.3 事件处理器顺序（Event Handler Order）

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);
```

5.3.4 触发事件（Triggering Events）

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。

5.3.5 移除事件处理器（Detaching Event Handlers）

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);
```

5.3.6 类级别的事件处理器（Class-Level Event Handlers）

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 `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::class, ActiveRecord::EVENT_AFTER_INSERT, function
($event) {
    Yii::debug(get_class($event->sender) . ' is inserted');
});
```

每当 `ActiveRecord` 或其子类的实例触发 `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::class, Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // 显示 "null"
});

Event::trigger(Foo::class, 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::class, Foo::EVENT_HELLO, $handler);

// 移除 Foo::EVENT_HELLO 事件的全部处理器
Event::off(Foo::class, Foo::EVENT_HELLO);
```

5.3.7 使用接口事件（Events using interfaces）

有更多的抽象方式来处理事件。你可以为特殊的事件创建一个独立的接口， 然后在你需要的类中实现它。

例如，我们可以先创建下面这个接口:

```
namespace app\interfaces;

interface DanceEventInterface
{
    const EVENT_DANCE = 'dance';
}
```

然后在两个类中实现:

```
class Dog extends Component implements DanceEventInterface
{
    public function meetBuddy()
    {
        echo "Woof!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}

class Developer extends Component implements DanceEventInterface
{
    public function testsPassed()
    {
        echo "Yay!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}
```

要处理由这些类触发的 `EVENT_DANCE` ， 调用 `Event::on()` 并将接口类名作为第一个参数:

```
Event::on('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE, function ($event) {
    Yii::trace(get_class($event->sender) . ' just danced'); // Will log that
    Dog or Developer danced
});
```

你可以在这些类中触发这个事件:

```
// trigger event for Dog class
Event::trigger(Dog::class, DanceEventInterface::EVENT_DANCE);

// trigger event for Developer class
Event::trigger(Developer::class, DanceEventInterface::EVENT_DANCE);
```

但是请注意, 你不能让所有实现这个接口的类都触发事件:

```
// 不会生效。实现此接口的类不会触发事件。
Event::trigger('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE);
```

调用 `Event::off()` 移除事件处理器。例如：

```
// 移除 $handler
Event::off('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE, $handler);

// 移除所有 `DanceEventInterface::EVENT_DANCE` 的处理器
Event::off('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE);
```

5.3.8 全局事件（Global Events）

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”）。

5.3.9 通配符事件（Wildcard Events）

自 2.0.14 以来，您可以为多个匹配通配符模式的事件设置事件处理程序。例如：

```
use Yii;

$foo = new Foo();

$foo->on('foo.event.*', function ($event) {
    // 触发任何事件，该名称以 'foo.event.' 开头
    Yii::debug('trigger event: ' . $event->name);
});
```

通配符模式也可以用于类级别的事件。 例如：

```
use yii\base\Event;
use Yii;

Event::on('app\models\*', 'before*', function ($event) {
    // 触发命名空间 'app\models' 中的任何类的任何事件，名称以 'before' 开头。
    Yii::debug('trigger event: ' . $event->name . ' for class: ' .
    get_class($event->sender));
});
```

这允许您使用以下代码通过单个处理程序捕获所有应用程序事件：

```
use yii\base\Event;
use Yii;

Event::on('*', '*', function ($event) {
    // 触发任何类的任何事件
    Yii::debug('trigger event: ' . $event->name);
});
```

> **注意：** 事件处理程序设置的使用通配符可能会降低应用程序的性能。 如果可能，最好避免。

为了移除由通配符模式指定的事件处理程序，您应该在 `yii\base\Component::off()` 或 `yii\base\Event::off()` 调用中重复相同的模式。 请记住，在移除事件处理程序期间传递通配符将移除为此通配符指定的处理程序， 而为常规事件名称附加的处理程序将保留，即使它们与模式匹配。 例如：

```
use Yii;

$foo = new Foo();

// 附加常规处理
$foo->on('event.hello', function ($event) {
    echo 'direct-handler'
});

// 附加通配符处理程序
$foo->on('*', function ($event) {
    echo 'wildcard-handler'
});

// 仅移除通配符处理程序！
$foo->off('*');

$foo->trigger('event.hello'); // 输出：'direct-handler'
```

5.4 行为

行为是 yii\base\Behavior 或其子类的实例。 行为，也称为 mixins[2]， 可以无须改变类继承关系即可增强一个已有的 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。

5.4.1 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

因为这是一个行为类，当它附加到一个组件时，该组件也将具有 prop1 和 prop2 属性和 foo() 方法。

> **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。

[2] https://zh.wikipedia.org/wiki/Mixin

> **注意**：如果 yii\base\Behavior::__get() 和/或 yii\base\Behavior::__set() 行为方法被覆盖，则需要覆盖 yii\base\Behavior::canGetProperty() 和/或 yii\base\Behavior::canSetProperty()。

5.4.2 处理事件

如果要让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
    }
}
```

events() 方法返回事件列表和相应的处理器。上例声明了 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) {
}
```

5.4.3 附加行为

可以静态或动态地附加行为到组件。前者在实践中更常见。

要静态附加行为，覆写行为要附加的组件类的 behaviors() 方法即可。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::class,

            // 命名行为，只有行为类名
            'myBehavior2' => MyBehavior::class,

            // 匿名行为，配置数组
            [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],

            // 命名行为，配置数组
            'myBehavior4' => [
                'class' => MyBehavior::class,
                '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::class);

// 附加配置数组
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::class,
    '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::class,          // 匿名行为
]);
```

还可以通过配置去附加行为:

```
[
    'as myBehavior2' => MyBehavior::class,

    'as myBehavior3' => [
        'class' => MyBehavior::class,
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

详情请参考 配置章节。

5.4.4 使用行为

使用行为，必须像前文描述的一样先把它附加到 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();
```

5.4.5 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();
```

5.4.6 使用

最后以 `yii\behaviors\TimestampBehavior` 的讲解来结尾，这个行为支持在 `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 `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::class,
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at',
                    'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                // if you're using datetime instead of UNIX timestamp:
                // 'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

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。

> **注意：** 对于上述实现使用MySQL数据库，请将列 (`created_at`, `updated_at`) 定义为 int(11) 作为 UNIX 时间戳。

有了以上这段代码，如果你有一个 `User` 对象并且试图保存它，你会发现它的 `created_at` 和 `updated_at` 被当前的UNIX时间戳自动填充：

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // 显示当前时间戳
```

`TimestampBehavior` 行为还提供了一个有用的方法 `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');
```

5.4.7 其它行为

有几种内置和外部行为可用：

- `yii\behaviors\BlameableBehavior` - 使用当前用户 ID 自动填充指定的属性。
- `yii\behaviors\SluggableBehavior` - 自动填充指定的属性，其值可以在 URL 中用作 slug。
- `yii\behaviors\AttributeBehavior` - 在发生特定事件时自动为 ActiveRecord 对象的一个或多个属性 指定一个指定的值。
- yii2tech\ar\softdelete\SoftDeleteBehavior[3] - 提供软删除和软恢复 ActiveRecord 的 方法。即将记录标记为已删除的设置标记或状态。
- yii2tech\ar\position\PositionBehavior[4] - 允许通过提供重新排序方法来 管理整数字段中的记录顺序。

5.4.8 比较行为与 Traits

虽然行为类似于 traits[5]，它们都将自己的属性和方法“注入”到主类中， 但它们在许多方面有所不同。如下所述，他们都有优点和缺点。 它们更像互补类而非替代类。

使用行为的原因

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 则不可行。

行为可以通过响应事件来定制组件的代码执行。

当附属于同一组件的不同行为之间可能存在名称冲突时， 通过优先考虑附加到该组件的行为， 自动解决冲突。由不同 traits 引起的名称冲突需要通过 重命名受影响的属性或方法进行手动解决。

[3] https://github.com/yii2tech/ar-softdelete
[4] https://github.com/yii2tech/ar-position
[5] https://www.php.net/manual/zh/language.oop5.traits.php

使用 Traits 的原因

Traits 比行为更有效，因为行为是既需要时间又需要内存的对象。

因为 IDE 是一种本地语言结构，所以它们对 Traits 更友好。

5.5 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。

5.5.1 配置的格式（Configuration Format）

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的完全限定类名。
- `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 ...
    ],
]
```

5.5.2 使用配置（Using Configurations）

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::creatObject()` 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（Application Configurations）

应用的配置可能是最复杂的配置之一。 因为 `application` 类拥有很多可配置的属性和事件。 更重要的是它的 `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面的入口脚本中， 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。

自版本 2.0.11 开始，系统配置支持使用 `container` 属性来配置依赖注入容器 例如：

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'container' => [
        'definitions' => [
            'yii\widgets\LinkPager' => ['maxButtonCount' => 5]
        ],
        'singletons' => [
            // 依赖注入容器单例配置
        ]
    ]
];
```

请参考 依赖注入容器 下面的 高级应用实例 获取更多 `definitions` 和 `singletons` 配置项和实际使用的例子。

小部件的配置（Widget Configurations）

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。

5.5.3 配置文件（Configuration Files）

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();
```

5.5.4 默认配置（Default Configurations）

`Yii::createObject()` 方法基于依赖注入容器实现。 使用 `Yii::creatObject()` 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。

5.5.5 环境常量（Environment Constants）

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，你就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;
```

5.6 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。 没有前导 `@` 定义的别名将以 `@` 字符作为前缀。

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。

5.6.1 定义别名（Defining Aliases）

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', 'https://www.example.com');

// 包含 \foo\Bar 类的具体文件的别名
Yii::setAlias('@foo/Bar.php', '/definitely/not/foo/Bar.php');
```

> **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()`）。 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' => 'https://www.example.com',
    ],
];
```

5.6.2 解析别名（Resolving Aliases）

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: https://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`。

5.6.3 使用别名（Using Aliases）

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。

5.6.4 预定义的别名（Predefined Aliases）

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 根路径（base path）。
- `@runtime`，当前运行的应用的 运行环境（runtime）路径。默认为 `@app/runtime`。
- `@webroot`，当前运行的Web应用程序的Web根目录。 它是根据包含 入口脚本 的目录确定的。
- `@web`，当前运行的Web应用程序的 base URL。它的值与 `yii\web\Request::$baseUrl` 相同。
- `@vendor`，Composer vendor 目录。

- `@bower`，包含 bower 包[6] 的根目录。默认为 `@vendor/bower`。
- `@npm`，包含 npm 包[7] 的根目录。默认为 `@vendor/npm`。

`@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。

> **注意：** `@web` 和 `@webroot` 别名，因为它们的描述表明是在 `Web application` 中定义的，因此默认情况下不适用于 `Console application` 应用程序。

5.6.5 扩展的别名（Extension Aliases）

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');
```

5.7 类自动加载（Autoloading）

Yii 依靠类自动加载机制[8]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[9] 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，本篇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。

5.7.1 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');
```

[6] https://bower.io/
[7] https://www.npmjs.com/
[8] https://www.php.net/manual/zh/language.oop5.autoload.php
[9] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md

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。

要将自定义命名空间添加到自动加载器，您需要使用 `Yii::setAlias()` 为命名空间的根目录定义别名。 例如，要加载位于 `path/to/foo` 目录中 `foo` 命名空间中的类，您将调用 `Yii::setAlias('@foo', 'path/to/foo')`。

5.7.2 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。

5.7.3 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。

> **信息：** 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。

5.7.4 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[10]， 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。

5.8 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;
```

[10] https://getcomposer.org/doc/04-schema.md#autoload

```
});

// 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 `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一个配置数组，配置一个应用并注册 `db`，`cache`，`tz` 和 `search` 组件：

```
return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'tz' => function() {
            return new \DateTimeZone(Yii::$app->formatter->defaultTimeZone);
        },
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');
            // ... other initializations ...
            return $solr;
        },
    ],
];
```

在上面的代码中，有一个替代方法来配置 `search` 组件。 而不是直接写一个 PHP 回调建立 `SolrService`实例，你可以使用一个静态类方法来返回这样的回调， 如下所示：

```
class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... other initializations ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];
```

当你发布一个 Yii 组件封装一些非 Yii 第三方库时，这种替代方法是最好的。 当您使用如上所示的静态方法来表示构建复杂逻辑的第三方对象时，您的组件用户只需要调用静态方法来配置组件。

5.8.1 遍历树（Tree traversal）

模块允许任意嵌套; Yii 应用程序本质上是一个模块树。 由于这些模块中的每一个都是服务定位器，所以子模块有权限访问其父模块。 这允许模块使用 `$this->get('db')` 而不是引用根服务定位器 `Yii::$app->get('db')`。 增加的好处是开发人员可以覆盖模块中的配置。

如果模块无法满足要求，则从模块中检索服务的请求将被传递给它的父模块。

请注意，模块中组件的配置决不会与来自父模块中组件的配置合并。Service Locator 模式允许我们定义命名服务，但不能假定具有相同名称的服务使用相同的配置参数。

5.9 依赖注入容器（Dependency Injection 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[11] 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。

5.9.1 依赖注入（Dependency Injection）

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:

[11] https://martinfowler.com/articles/injection.html

- 构造方法注入;
- 方法注入;
- Setter 和属性注入;
- PHP 回调注入.

构造方法注入（Constructor Injection）

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);
```

方法注入（Method Injection）

通常，类的依赖关系传递给构造函数，并且在整个生命周期中都可以在类内部使用。 通过方法注入，可以提供仅由类的单个方法需要的依赖关系，并将其传递给构造函数可能不可行，或者可能会在大多数用例中导致太多开销。

类方法可以像下面例子中的 `doSomething()` 方法一样定义:

```
class MyClass extends \yii\base\Component
{
    public function __construct(/*Some lightweight dependencies here*/,
    $config = [])
    {
        // ...
    }

    public function doSomething($param1, \my\heavy\Dependency $something)
    {
        // do something with $something
    }
}
```

你可以自己通过一个实例 `\my\heavy\Dependency` 调用这个方法或使用 `yii\di\Container::invoke()` 如下:

```
$obj = new MyClass(/*...*/);
Yii::$container->invoke([$obj, 'doSomething'], ['param1' => 42]); //
$something will be provided by the DI container
```

Setter 和属性注入（Setter and Property Injection）

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\BaseObject;

class Foo extends BaseObject
{
    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'),
]);
```

> **信息：** `yii\di\Container::get()` 方法将其第三个参数作为配置数组应用于正在创建的对象。 如果该类实现 `yii\base\Configurable` 接口（例如 `yii\base\BaseObject`），则配置数组将作为最后一个参数传递给类构造函数； 否则，将在创建对象后应用该配置。

PHP 回调注入（PHP Callable Injection）

在这种情况下，容器将使用已注册的 PHP 回调来构建类的新实例。 每次调用 `yii\di\Container::get()`，相应的回调将被调用。 调用方负责解析依赖项，并适当地将它们注入到新创建的对象中。 例如，

```
$container->set('Foo', function () {
    $foo = new Foo(new Bar);
    // ... 其他初始化 ...
    return $foo;
});

$foo = $container->get('Foo');
```

要省略构建新对象的复杂逻辑，可以使用静态类方法作为可调用的方法。 例如，

```
class FooBuilder
{
    public static function build()
    {
        $foo = new Foo(new Bar);
        // ... 其他初始化 ...
        return $foo;
    }
}

$container->set('Foo', ['app\helper\FooBuilder', 'build']);

$foo = $container->get('Foo');
```

这样做的话，想要配置 `Foo` 类的人不再需要知道它是如何构建的。

5.9.2 注册依赖关系（Registering Dependencies）

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);
```

> **提示：** 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',
]);
```

5.9.3 解决依赖关系（Resolving Dependencies）

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\BaseObject;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends BaseObject implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends BaseObject
{
    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);
```

5.9.4 实践中的运用（Practical Usage）

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::$container` 访问。 当调用 `Yii::createObject()` 时，此方法实际上会调用这个容器的 `get()` 方法创建新对象。 如上所述，DI 容器会自动解决依赖关系（如果有）并将其注入新创建的对象中。 因为 Yii 在其多数核心代码中都使用了`Yii::createObject()` 创建新对象， 所以你可以通过 `Yii::$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]);
```

> **注意：** 在部件调用中给出的属性将始终覆盖DI容器中的定义。
> 即使您指定了一个数组，例如 `'options' => ['id' => 'mypager']`
> 这些将不会与其他选项合并， 而是替换它们。

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。

5.9.5 高级实用性（Advanced Practical Usage）

比如说我们在 API 应用方面工作：

- `app\components\Request` 类继承了 `yii\web\Request` 并提供了额外的功能
- `app\components\Response` 类继承了 `yii\web\Response` 并且在创建时应该将 `format` 属性设置为 `json`
- `app\storage\FileStorage` 和 `app\storage\DocumentsReader` 用于处理位于某些文件存储中的文档的某些逻辑：

  ```
  class FileStorage
  {
      public function __construct($root) {
          // whatever
      }
  }

  class DocumentsReader
  {
      public function __construct(FileStorage $fs) {
          // whatever
      }
  }
  ```

可以一次配置多个定义，将配置数组传递给 `setDefinitions()` 或 `setSingletons()` 方法。 遍历配置数组，将分别为每个对象分别调用 `set()` 或 `setSingleton()` 方法。

配置数组格式为：

- `key`：类名称，接口名称或别名。 该 key 将作为第一个参数 `$class` 传递给 `set()` 方法。
- `value`：与 `$class` 关联的定义。`$definition` 参数的值可能在 `set()` 文档中描述。`$definition` 将作为第二个参数传递给 `set()` 方法。

例如，让我们配置我们的容器以遵循上述要求：

```
$container->setDefinitions([
    'yii\web\Request' => 'app\components\Request',
```

```
    'yii\web\Response' => [
        'class' => 'app\components\Response',
        'format' => 'json'
    ],
    'app\storage\DocumentsReader' => function () {
        $fs = new app\storage\FileStorage('/var/tempfiles');
        return new app\storage\DocumentsReader($fs);
    }
]);

$reader = $container->get('app\storage\DocumentsReader');
// 将按照配置中的描述创建 DocumentReader 对象及其依赖关系
```

> **提示：** 自 2.0.11 版以后，可以使用应用程序配置以声明式风格配置容器。 查看配置指南文章的 应用程序配置小节。

一切正常，但如果我们需要创建 `Document Writer` 类， 我们将复制粘贴创建 `FileStorage` 对象的行，这显然不是最聪明的方式。

如 解决依赖关系 子节中所述，`set()` 和 `setSingleton()` 可以选择将依赖项的构造函数参数作为第三个参数。 要设置构造函数参数，可以使用以下配置数组格式：

- `key`：类名称，接口名称或别名。该 key 将作为第一个参数 `$class` 传递给 `set()` 方法。
- `value`：两个元素的数组。第一个元素将传递给 `set()` 方法 作为第二个参数 `$definition`，第二个元素为 `$params`。

让我们来修改我们的例子：

```
$container->setDefinitions([
    'tempFileStorage' => [ // 我们为了方便创建了一个别名
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles'] // 可以从一些配置文件中获取
    ],
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader');
// 将与前面示例中的行为完全相同。
```

你可能会注意到 `Instance::of('tempFileStorage')` 符号。这意味着，`Container` 将隐含地提供一个用 `tempFileStorage` 名称注册的依赖项， 并将其作为 `app\storage\DocumentsWriter` 构造函数的第一个参数传递。

> **注意：** `setDefinitions()` 和 `setSingletons()` 方法从版本 2.0.11 开始可用。

配置优化的另一个步骤是将某些依赖项注册为单例。 通过 `set()` 注册的依赖项将在每次需要时实例化。 某些类在运行时不会更改状态， 因此它们可能会被注册为单例以提高应用程序的性能。

一个很好的例子可以是 `app\storage\FileStorage` 类，它用一个简单的 API （例如 `$fs->read()`， `$fs->write()`）在文件系统上执行一些操作。 这些操作不会更改内部类的状态，因此我们可以创建一次实例并多次使用它。

```
$container->setSingletons([
    'tempFileStorage' => [
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles']
    ],
]);

$container->setDefinitions([
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader');
```

5.9.6 什么时候注册依赖关系（When to Register Dependencies）

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。 如下是推荐的实践：

- 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。

5.9.7 总结（Summary）

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[12] ， 对依赖注入和服务定位器有个更深入的理解。

Yii 在依赖住入（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。

[12] https://martinfowler.com/articles/injection.html

Chapter 6

配合数据库工作（Working with Databases）

6.1 数据库访问对象（Database Access Objects）

Yii 包含了一个建立在 PHP PDO 之上的数据访问层 (DAO)。DAO为不同的数据库提供了一套统一的API。 其中 `ActiveRecord` 提供了数据库与模型(MVC 中的 M,Model) 的交互，`QueryBuilder` 用于创建动态的查询语句。DAO提供了简单高效的SQL查询，可以用在与数据库交互的各个地方.

使用 Yii DAO 时，你主要需要处理纯 SQL 语句和 PHP 数组。因此，这是访问数据库最高效的方法。 然而，因为不同数据库之间的 SQL 语法往往是不同的， 使用 Yii DAO 也意味着你必须花些额外的工夫来创建一个"数据库无关"的应用。

Yii DAO 支持下列现成的数据库:

- MySQL[1]
- MariaDB[2]
- SQLite[3]
- PostgreSQL[4]：版本 8.4 或更高
- CUBRID[5]：版本 9.3 或更高。
- Oracle[6]
- MSSQL[7]：版本 2008 或更高。

> **信息**： 在Yii 2.1及更高版本中，DAO 支持 CUBRID，Oracle 和 MSSQL 不再作为框架的内置核心组件提供。它们必须作为

[1] https://www.mysql.com/
[2] https://mariadb.com/
[3] https://sqlite.org/
[4] https://www.postgresql.org/
[5] https://www.cubrid.org/
[6] https://www.oracle.com/database/
[7] https://www.microsoft.com/en-us/sqlserver/default.aspx

独离的 扩展 安装。 yiisoft/yii2-oracle[8] 和 yiisoft/yii2-mssql[9] 都属于 官方扩展[10]。

注意： 供 PHP 7 使用的新版 pdo_oci 扩展目前仅发布了源代码，如果你想编译使用请参照 社区用户提供的编译安装指引[11]。 或者你也可以在你的应用中使用 PDO模拟层[12]。

6.1.1 创建数据库连接（Creating DB Connections）

想要访问数据库，你首先需要通过创建一个 `yii\db\Connection` 实例来与之建立连接。

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

因为数据库连接经常需要在多个地方使用到， 一个常见的做法是以应用组件的方式来配置它，如下:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=example',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

之后你就可以通过语句 `Yii::$app->db` 来使用数据库连接了。

提示： 如果你的应用需要访问多个数据库，你可以配置多个 DB 应用组件。

配置数据库连接时， 你应该总是通过 `dsn` 属性来指明它的数据源名称 (DSN) 。 不同的数据库有着不同的 DSN 格式。 请参考 PHP manual[13] 来获得更多细节。下面是一些例子:

[8] https://www.yiiframework.com/extension/yiisoft/yii2-oracle
[9] https://www.yiiframework.com/extension/yiisoft/yii2-mssql
[10] https://www.yiiframework.com/extensions/official
[11] https://github.com/yiisoft/yii2/issues/10975#issuecomment-248479268
[12] https://github.com/taq/pdooci
[13] https://www.php.net/manual/zh/pdo.construct.php

- MySQL, MariaDB: `mysql:host=localhost;dbname=mydatabase`
- SQLite: `sqlite:/path/to/database/file`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID: `cubrid:dbname=demodb;host=localhost;port=33000`
- MS SQL Server (via sqlsrv driver): `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (via dblib driver): `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (via mssql driver): `mssql:host=localhost;dbname=mydatabase`
- Oracle: `oci:dbname=//localhost:1521/mydatabase`

请注意，如果你是通过 ODBC 来连接数据库，你应该配置 `yii\db\Connection::$driverName` 属性， 以便 Yii 能够知道实际的数据库种类。例如：

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

除了 `dsn` 属性， 你常常需要配置 `username` 和 `password`。请参考 `yii\db\Connection` 来获取完整的可配置属性列表。

> **信息：** 当你实例化一个 DB Connection 时，直到你第一次执行 SQL 或者你明确地调用 `open()` 方法时， 才建立起实际的数据库连接。
>
> **提示：** 有时你可能想要在建立起数据库连接时立即执行一些语句来初始化一些环境变量 (比如设置时区或者字符集)，你可以通过为数据库连接的 `afterOpen` 事件注册一个事件处理器来达到目的。 你可以像这样直接在应用配置中注册处理器：
>
> ```
> 'db' => [
>  // ...
>  'on afterOpen' => function($event) {
>  // $event->sender refers to the DB connection
>  $event->sender->createCommand("SET time_zone =
>  'UTC'")->execute();
>  }
> ],
> ```

6.1.2 执行 SQL 查询（Executing SQL Queries）

一旦你拥有了 DB Connection 实例，你可以按照下列步骤来执行 SQL 查询：

1. 使用纯SQL查询来创建出 `yii\db\Command`；
2. 绑定参数 (可选的)；

3. 调用 `yii\db\Command` 里 SQL 执行方法中的一个。

下列例子展示了几种不同的从数据库取得数据的方法:

```
// 返回多行. 每行都是列名和值的关联数组.
// 如果该查询没有结果则返回空数组
$posts = Yii::$app->db->createCommand('SELECT * FROM post')
            ->queryAll();

// 返回一行 (第一行)
// 如果该查询没有结果则返回 false
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=1')
           ->queryOne();

// 返回一列 (第一列)
// 如果该查询没有结果则返回空数组
$titles = Yii::$app->db->createCommand('SELECT title FROM post')
             ->queryColumn();

// 返回一个标量值
// 如果该查询没有结果则返回 false
$count = Yii::$app->db->createCommand('SELECT COUNT(*) FROM post')
             ->queryScalar();
```

> **注意:** 为了保持精度, 即使对应的数据库列类型为数值型, 所有从数据库取得的数据都被表现为字符串。

绑定参数（Binding Parameters）

当使用带参数的 SQL 来创建数据库命令时, 你几乎总是应该使用绑定参数的方法来防止 SQL 注入攻击, 例如:

```
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
           ->bindValue(':id', $_GET['id'])
           ->bindValue(':status', 1)
           ->queryOne();
```

在 SQL 语句中, 你可以嵌入一个或多个参数占位符(例如, 上述例子中的 `:id`)。 一个参数占位符应该是以冒号开头的字符串。 之后你可以调用下面绑定参数的方法来绑定参数值:

- `bindValue()`: 绑定一个参数值
- `bindValues()`: 在一次调用中绑定多个参数值
- `bindParam()`: 与 `bindValue()` 相似, 但是也支持绑定参数引用。

下面的例子展示了几个可供选择的绑定参数的方法:

```
$params = [':id' => $_GET['id'], ':status' => 1];

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
           ->bindValues($params)
```

```
            ->queryOne();

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status', $params)
            ->queryOne();
```

绑定参数是通过 预处理语句[14] 实现的。 除了防止 SQL 注入攻击，它也可以通过一次预处理 SQL 语句， 使用不同参数多次执行，来提升性能。例如：

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
// ...
```

因为 `bindParam()` 支持通过引用来绑定参数， 上述代码也可以像下面这样写：

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id')
              ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();

$id = 2;
$post2 = $command->queryOne();
// ...
```

请注意，在执行语句前你将占位符绑定到 `$id` 变量， 然后在之后的每次执行前改变变量的值（这通常是用循环来完成的）。 以这种方式执行查询比为每个不同的参数值执行一次新的查询要高效得多得多。

> **信息：** 参数绑定仅用于需要将值插入到包含普通SQL的字符串中的地方。 在 Query Builder 和 Active Record 等更高抽象层的许多地方，您经常会指定一组将被转换为 SQL 的值。 在这些地方，参数绑定是由 Yii 内部完成的，因此不需要手动指定参数。

执行非查询语句（Executing Non-SELECT Queries）

上面部分中介绍的 `queryXyz()` 方法都处理的是从数据库返回数据的查询语句。对于那些不取回数据的语句， 你应该调用的是 `yii\db\Command::execute()` 方法。例如，

```
Yii::$app->db->createCommand('UPDATE post SET status=1 WHERE id=1')
   ->execute();
```

[14] https://www.php.net/manual/zh/mysqli.quickstart.prepared-statements.php

`yii\db\Command::execute()` 方法返回执行 SQL 所影响到的行数。

对于 INSERT, UPDATE 和 DELETE 语句，不再需要写纯SQL语句了， 你可以直接调用 `insert()`、`update()`、`delete()`， 来构建相应的 SQL 语句。这些方法将正确地引用表和列名称以及绑定参数值。例如,

```
// INSERT (table name, column values)
Yii::$app->db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// UPDATE (table name, column values, condition)
Yii::$app->db->createCommand()->update('user', ['status' => 1], 'age >
30')->execute();

// DELETE (table name, condition)
Yii::$app->db->createCommand()->delete('user', 'status = 0')->execute();
```

你也可以调用 `batchInsert()` 来一次插入多行， 这比一次插入一行要高效得多:

```
// table name, column names, column values
Yii::$app->db->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();
```

另一个有用的方法是 `upsert()`。Upsert 是一种原子操作，如果它们不存在（匹配唯一约束），则将行插入到数据库表中， 或者在它们执行时更新它们:

```
Yii::$app->db->createCommand()->upsert('pages', [
    'name' => 'Front page',
    'url' => 'https://example.com/', // url is unique
    'visits' => 0,
], [
    'visits' => new \yii\db\Expression('visits + 1'),
], $params)->execute();
```

上面的代码将插入一个新的页面记录或自动增加访问计数器。

请注意，上述的方法只是构建出语句， 你总是需要调用 `execute()` 来真正地执行它们。

6.1.3 引用表和列名称（Quoting Table and Column Names）

当写与数据库无关的代码时，正确地引用表和列名称总是一件头疼的事，因为不同的数据库有不同的名称引用规则，为了克服这个问题， 你可以使用下面由 Yii 提出的引用语法。

- `[[column name]]`: 使用两对方括号来将列名括起来;
- `{{table name}}`: 使用两对大括号来将表名括起来。

Yii DAO 将自动地根据数据库的具体语法来将这些结构转化为对应的 被引用的列或者表名称。 例如,

```
// 在 MySQL 中执行该 SQL : SELECT COUNT(`id`) FROM `employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM
{{employee}}")
            ->queryScalar();
```

使用表前缀（Using Table Prefix）

如果你的数据库表名大多都拥有一个共同的前缀， 你可以使用 Yii DAO 所提供的表前缀功能。

首先，通过应用配置中的 `yii\db\Connection::$tablePrefix` 属性来指定表前缀:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

接着在你的代码中，当你需要涉及到一张表名中包含该前缀的表时， 应使用语法 `{{%table_name}}`。百分号将被自动地替换为你在配置 DB 组件时指定的表前缀。 例如,

```
// 在 MySQL 中执行该 SQL: SELECT COUNT(`id`) FROM `tbl_employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM
{{%employee}}")
            ->queryScalar();
```

6.1.4 执行事务（Performing Transactions）

当顺序地执行多个相关的语句时， 你或许需要将它们包在一个事务中来保证数据库的完整性和一致性。 如果这些语句中的任何一个失败了， 数据库将回滚到这些语句执行前的状态。

下面的代码展示了一个使用事务的典型方法:

```
Yii::$app->db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...
});
```

上述代码等价于下面的代码，但是下面的代码给予了你对于错误处理代码的更多掌控：

```
$db = Yii::$app->db;
$transaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

通过调用 `beginTransaction()` 方法， 一个新事务开始了。 事务被表示为一个存储在 `$transaction` 变量中的 `yii\db\Transaction` 对象。 然后，被执行的语句都被包含在一个 `try...catch...` 块中。 如果所有的语句都被成功地执行了， `commit()` 将被调用来提交这个事务。 否则， 如果异常被触发并被捕获， `rollBack()` 方法将被调用， 来回滚事务中失败语句之前所有语句所造成的改变。 `throw $e` 将重新抛出该异常， 就好像我们没有捕获它一样， 因此正常的错误处理程序将处理它。

指定隔离级别（Specifying Isolation Levels）

Yii 也支持为你的事务设置隔离级别[15]。默认情况下，当我们开启一个新事务， 它将使用你的数据库所设定的隔离级别。你也可以向下面这样重载默认的隔离级别，

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

Yii::$app->db->transaction(function ($db) {
    ....
}, $isolationLevel);

// or alternatively

$transaction = Yii::$app->db->beginTransaction($isolationLevel);
```

Yii 为四个最常用的隔离级别提供了常量：

- `yii\db\Transaction::READ_UNCOMMITTED` - 最弱的隔离级别，脏读、不可重复读以及幻读都可能发生。
- `yii\db\Transaction::READ_COMMITTED` - 避免了脏读。

[15] https://zh.wikipedia.org/wiki/%E4%BA%8B%E5%8B%99%E9%9A%94%E9%9B%A2#.E9.9A.94.E7.A6.BB.E7.BA.A7.E5.88.AB

- `yii\db\Transaction::REPEATABLE_READ` - 避免了脏读和不可重复读。
- `yii\db\Transaction::SERIALIZABLE` - 最强的隔离级别， 避免了上述所有的问题。

除了使用上述的常量来指定隔离级别，你还可以使用你的数据库所支持的具有有效语法的字符串。 比如，在 PostgreSQL 中，你可以使用 `SERIALIZABLE READ ONLY DEFERRABLE`。

请注意，一些数据库只允许为整个连接设置隔离级别， 即使你之后什么也没指定，后来的事务都将获得与之前相同的隔离级别。 使用此功能时，你需要为所有的事务明确地设置隔离级别来避免冲突的设置。 在本文写作之时，只有 MSSQL 和 SQLite 受这些限制的影响。

> **注意：** SQLite 只支持两种隔离级别，所以你只能使用 `READ UNCOMMITTED` 和 `SERIALIZABLE`。 使用其他级别将导致异常的抛出。
>
> **注意：** PostgreSQL 不支持在事务开启前设定隔离级别， 因此，你不能在开启事务时直接指定隔离级别。 你必须在事务开始后再调用 `yii\db\Transaction::setIsolationLevel()`。

嵌套事务（Nesting Transactions）

如果你的数据库支持保存点，你可以像下面这样嵌套多个事务：

```
Yii::$app->db->transaction(function ($db) {
    // outer transaction

    $db->transaction(function ($db) {
        // inner transaction
    });
});
```

或者，

```
$db = Yii::$app->db;
$outerTransaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();

    $innerTransaction = $db->beginTransaction();
    try {
        $db->createCommand($sql2)->execute();
        $innerTransaction->commit();
    } catch (\Exception $e) {
        $innerTransaction->rollBack();
        throw $e;
    } catch (\Throwable $e) {
        $innerTransaction->rollBack();
        throw $e;
    }
```

```
    $outerTransaction->commit();
} catch (\Exception $e) {
    $outerTransaction->rollBack();
    throw $e;
} catch (\Throwable $e) {
    $outerTransaction->rollBack();
    throw $e;
}
```

6.1.5 复制和读写分离（Replication and Read-Write Splitting）

许多数据库支持数据库复制[16]来获得更好的数据库可用性， 以及更快的服务器响应时间。通过数据库复制功能， 数据从所谓的主服务器被复制到从服务器。所有的写和更新必须发生在主服务器上， 而读可以发生在从服务器上。

为了利用数据库复制并且完成读写分离， 你可以按照下面的方法来配置 `yii\db\Connection` 组件：

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

    // 主库的配置
    'dsn' => 'dsn for master server',
    'username' => 'master',
    'password' => '',

    // 从库的通用配置
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // 使用一个更小的连接超时
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // 从库的配置列表
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

上述的配置指定了一主多从的设置。 这些从库其中之一将被建立起连接并执行读操作，而主库将被用来执行写操作。 这样的读写分离将通过上述配

[16] https://en.wikipedia.org/wiki/Replication_(computing)#Database_replication

置自动地完成。比如，

```
// 使用上述配置来创建一个 Connection 实例
Yii::$app->db = Yii::createObject($config);

// 在从库中的一个上执行语句
$rows = Yii::$app->db->createCommand('SELECT * FROM user LIMIT
10')->queryAll();

// 在主库上执行语句
Yii::$app->db->createCommand("UPDATE user SET username='demo' WHERE
id=1")->execute();
```

> **信息：** 通过调用 `yii\db\Command::execute()` 来执行的语句都被视为写操作， 而其他所有通过调用 `yii\db\Command` 中任一 "query" 方法来执行的语句都被视为读操作。 你可以通过 `Yii::$app->db->slave` 来获取当前有效的从库连接。

`Connection` 组件支持从库间的负载均衡和失效备援， 当第一次执行读操作时，`Connection` 组件将随机地挑选出一个从库并尝试与之建立连接， 如果这个从库被发现为"挂掉的"，将尝试连接另一个从库。 如果没有一个从库是连接得上的，那么将试着连接到主库上。 通过配置 `server status cache`， 一个"挂掉的"服务器将会被记住，因此，在一个 yii\db\Connection::serverRetryInterval 内将不再试着连接该服务器。

> **信息：** 在上面的配置中， 每个从库都共同地指定了 10 秒的连接超时时间，这意味着，如果一个从库在 10 秒内不能被连接上，它将被视为"挂掉的"。 你可以根据你的实际环境来调整该参数。

你也可以配置多主多从。例如，

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

    // 主库通用的配置
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // 主库配置列表
    'masters' => [
        ['dsn' => 'dsn for master server 1'],
        ['dsn' => 'dsn for master server 2'],
    ],
```

```
    // 从库的通用配置
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // 从库配置列表
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

上述配置指定了两个主库和两个从库。 `Connection` 组件在主库之间，也支持如从库间般的负载均衡和失效备援。 唯一的差别是，如果没有主库可用，将抛出一个异常。

> **注意：** 当你使用 `masters` 属性来配置一个或多个主库时， 所有其他指定数据库连接的属性 (例如 `dsn`, `username`, `password`) 与 `Connection` 对象本身将被忽略。

默认情况下，事务使用主库连接， 一个事务内，所有的数据库操作都将使用主库连接，例如，

```
$db = Yii::$app->db;
// 在主库上启动事务
$transaction = $db->beginTransaction();

try {
    // 两个语句都是在主库上执行的
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE
    id=1")->execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

如果你想在从库上开启事务，你应该明确地像下面这样做：

```
$transaction = Yii::$app->db->slave->beginTransaction();
```

有时，你或许想要强制使用主库来执行读查询。 这可以通过 `useMaster()` 方法来完成：

```
$rows = Yii::$app->db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

你也可以明确地将 `Yii::$app->db->enableSlaves` 设置为 false 来将所有的读操作指向主库连接。

6.1.6 操纵数据库模式（Working with Database Schema）

Yii DAO 提供了一套完整的方法来让你操纵数据库模式， 如创建表、从表中删除一列，等等。这些方法罗列如下：

- `createTable()`：创建一张表
- `renameTable()`：重命名一张表
- `dropTable()`：删除一张表
- `truncateTable()`：删除一张表中的所有行
- `addColumn()`：增加一列
- `renameColumn()`：重命名一列
- `dropColumn()`：删除一列
- `alterColumn()`：修改一列
- `addPrimaryKey()`：增加主键
- `dropPrimaryKey()`：删除主键
- `addForeignKey()`：增加一个外键
- `dropForeignKey()`：删除一个外键
- `createIndex()`：增加一个索引
- `dropIndex()`：删除一个索引

这些方法可以如下地使用：

```
// CREATE TABLE
Yii::$app->db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

上面的数组描述要创建的列的名称和类型。 对于列的类型， Yii 提供了一套抽象数据类型来允许你定义出数据库无关的模式。 这些将根据表所在数据库的种类，被转换为特定的类型定义。 请参考 `createTable()`-method 的 API 文档来获取更多信息。

除了改变数据库模式， 你也可以通过 DB Connection 的 `getTableSchema()` 方法来检索某张表的定义信息。例如，

```
$table = Yii::$app->db->getTableSchema('post');
```

该方法返回一个 `yii\db\TableSchema` 对象， 它包含了表中的列、主键、外键，等等的信息。 所有的这些信息主要被 query builder 和 active record 所使用，来帮助你写出数据库无关的代码。

6.2 查询构建器

查询构建器建立在 Database Access Objects 基础之上，可让你创建 程序化的、DBMS无关的SQL语句。相比于原生的SQL语句，查询构建器可以帮你写出可读性更强的SQL相关的代码，并生成安全性更强的SQL语句。

使用查询构建器通常包含以下两个步骤：

1. 创建一个 `yii\db\Query` 对象来代表一条 SELECT SQL 语句的不同子句（例如 `SELECT`, `FROM`）。
2. 执行 `yii\db\Query` 的一个查询方法（例如：`all()`）从数据库当中检索数据。

如下所示代码是查询构造器的一个典型用法：

```
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->all();
```

上面的代码将会生成并执行如下的SQL语句，其中 `:last_name` 参数绑定了字符串 `'Smith'`。

```
SELECT `id`, `email`
FROM `user`
WHERE `last_name` = :last_name
LIMIT 10
```

> **提示：** 你平时更多的时候会使用 `yii\db\Query` 而不是 `yii\db\QueryBuilder`。 当你调用其中一个查询方法时，后者将会被前者隐式的调用。`yii\db\QueryBuilder`主要负责将 DBMS 不相关的 `yii\db\Query` 对象转换成 DBMS 相关的 SQL 语句（例如， 以不同的方式引用表或字段名称）。

6.2.1 创建查询

为了创建一个 `yii\db\Query` 对象，你需要调用不同的查询构建方法来代表SQL语句的不同子句。 这些方法的名称集成了在SQL语句相应子句中使用的关键字。例如，为了指定 SQL 语句当中的 `FROM` 子句，你应该调用 `from()` 方法。所有的查询构建器方法返回的是查询对象本身， 也就是说，你可以把多个方法的调用串联起来。

接下来，我们会对这些查询构建器方法进行一一讲解：

select()

select() 方法用来指定 SQL 语句当中的 SELECT 子句。 你可以像下面的例子一样使用一个数组或者字符串来定义需要查询的字段。当 SQL 语句 是由查询对象生成的时候，被查询的字段名称将会自动的被引号括起来。

```
$query->select(['id', 'email']);

// 等同于:

$query->select('id, email');
```

就像写原生 SQL 语句一样，被选取的字段可以包含表前缀，以及/或者字段别名。 例如:

```
$query->select(['user.id AS user_id', 'email']);

// 等同于:

$query->select('user.id AS user_id, email');
```

如果使用数组格式来指定字段，你可以使用数组的键值来表示字段的别名。 例如，上面的代码可以被重写为如下形式:

```
$query->select(['user_id' => 'user.id', 'email']);
```

如果你在组建查询时没有调用 select() 方法，那么选择的将是 '*' ， 也即选取的是所有的字段。

除了字段名称以外，你还可以选择数据库的表达式。当你使用到包含逗号的数据库表达式的时候， 你必须使用数组的格式，以避免自动的错误的引号添加。例如:

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name",
'email']);
```

与所有涉及原始 SQL 的地方一样，当在 select 中编写 DB 表达式时，可以对表名和列名使用 与 DBMS 无关的引用语法。

从 2.0.1 的版本开始你就可以使用子查询了。在定义每一个子查询的时候， 你应该使用 yii\db\Query 对象。例如:

```
$subQuery = (new Query())->select('COUNT(*)')->from('user');

// SELECT `id`, (SELECT COUNT(*) FROM `user`) AS `count` FROM `post`
$query = (new Query())->select(['id', 'count' => $subQuery])->from('post');
```

你应该调用 distinct() 方法来去除重复行，如下所示:

```
// SELECT DISTINCT `user_id` ...
$query->select('user_id')->distinct();
```

你可以调用 addSelect() 方法来选取附加字段，例如:

```
$query->select(['id', 'username'])
    ->addSelect(['email']);
```

from()

from() 方法指定了 SQL 语句当中的 FROM 子句。例如：

```
// SELECT * FROM `user`
$query->from('user');
```

你可以通过字符串或者数组的形式来定义被查询的表名称。就像你写原生的 SQL 语句一样， 表名称里面可包含数据库前缀，以及/或者表别名。例如：

```
$query->from(['public.user u', 'public.post p']);

// 等同于：

$query->from('public.user u, public.post p');
```

如果你使用的是数组的格式，那么你同样可以用数组的键值来定义表别名，如下所示：

```
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

除了表名以外，你还可以从子查询中再次查询，这里的子查询是由 yii\db\Query 创建的对象。 例如：

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');

// SELECT * FROM (SELECT `id` FROM `user` WHERE status=1) u
$query->from(['u' => $subQuery]);
```

前缀 from 还可以应用默认的 tablePrefix 前缀，实现细节请参考 “数据库访问对象指南”的“Quoting Tables”章节.

where()

where() 方法定义了 SQL 语句当中的 WHERE 子句。 你可以使用如下四种格式来定义 WHERE 条件：

- 字符串格式，例如： 'status=1'
- 哈希格式，例如： ['status' => 1, 'type' => 2]
- 操作符格式，例如： ['like', 'name', 'test']
- 对象格式，例如： new LikeCondition('name', 'LIKE', 'test')

字符串格式 在定义非常简单的查询条件的时候，字符串格式是最合适的。 它看起来和原生 SQL 语句差不多。例如：

```
$query->where('status=1');

// 或使用参数绑定来绑定动态参数值
$query->where('status=:status', [':status' => $status]);

// 原生 SQL 在日期字段上使用 MySQL YEAR() 函数
$query->where('YEAR(somedate) = 2015');
```

千万不要像如下的例子一样直接在条件语句当中嵌入变量，特别是当这些变量来源于终端用户输入的时候， 因为这样我们的软件将很容易受到 SQL 注入的攻击。

```
// 危险！千万别这样干，除非你非常的确定 $status 是一个整型数值。
$query->where("status=$status");
```

当使用 参数绑定 的时候，你可以调用 `params()` 或者 `addParams()` 方法 来分别绑定不同的参数。

```
$query->where('status=:status')
    ->addParams([':status' => $status]);
```

与涉及原生 SQL 的所有地方一样，在以字符串格式写入条件时，可以对表名和列名使用 与 DBMS 无关的引用语法。

哈希格式 哈希格式最适合用来指定多个 AND 串联起来的简单的"等于断言"子条件。 它是以数组的形式来书写的，数组的键表示字段的名称，而数组的值则表示 这个字段需要匹配的值。例如：

```
// ...WHERE (`status` = 10) AND (`type` IS NULL) AND (`id` IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

就像你所看到的一样，查询构建器非常的智能，能恰当地处理数值当中的空值和数组。

你也可以像下面那样在子查询当中使用哈希格式：

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE `id` IN (SELECT `id` FROM `user`)
$query->where(['id' => $userQuery]);
```

使用哈希格式，Yii 在内部对相应的值进行参数绑定，与 字符串格式 相比， 此处你不需要手动添加参数绑定。但请注意，Yii 不会帮你转义列名，所以如果你 从用户端获得的变量作为列名而没有进行任何额外的检查，对于 SQL 注入攻击， 你的程序将变得很脆弱。为了保证应用程序的安全，请不要将变量用作列名 或者你必须用白名单过滤变量。如果你实在需要从用户获取列名，请阅读 过滤数据 章节。例如，以下代码易受攻击：

```
// 易受攻击的代码：
$column = $request->get('column');
$value = $request->get('value');
$query->where([$column => $value]);
// $value 是安全的，但是 $column 名不会被转义处理！
```

操作符格式 操作符格式允许你指定类程序风格的任意条件语句，如下所示：

```
[操作符, 操作数1, 操作数2, ...]
```

其中每个操作数可以是字符串格式、哈希格式或者嵌套的操作符格式， 而操作符可以是如下列表中的一个：

- `and`: 操作数会被 `AND` 关键字串联起来。例如，`['and', 'id=1', 'id=2']` 将会生成 `id=1 AND id=2`。如果操作数是一个数组，它也会按上述规则转换成 字符串。例如，`['and', 'type=1', ['or', 'id=1', 'id=2']]` 将会生成 `type=1 AND (id=1 OR id=2)`。 这个方法不会自动加引号或者转义。
- `or`: 用法和 `and` 操作符类似，这里就不再赘述。
- `not`: 只需要操作数 1，它将包含在 `NOT()` 中。例如，`['not', 'id = 1']` 将生成 `['not', 'id=1']`。操作数 1 也可以是个描述多个表达式的数组。例如 `['not', ['status' => 'draft', 'name' => 'example']]` 将生成 `NOT ((status='draft') AND (name='example'))`。
- `between`: 第一个操作数为字段名称，第二个和第三个操作数代表的是这个字段 的取值范围。例如，`['between', 'id', 1, 10]` 将会生成 `id BETWEEN 1 AND 10`。 如果你需要建立一个值在两列之间的查询条件（比如 `11 BETWEEN min_id AND max_id`）， 你应该使用 `BetweenColumnsCondition`。请参阅 条件-对象格式 一章以了解有关条件的对象定义的更多信息。
- `not between`: 与 `between` 类似，除了 `BETWEEN` 被 `NOT BETWEEN` 替换 在生成条件时。
- `in`: 第一个操作数应为字段名称或者 DB 表达式。第二个操作符既可以是一个数组， 也可以是一个 `Query` 对象。它会转换成`IN` 条件语句。如果第二个操作数是一个 数组，那么它代表的是字段或 DB 表达式的取值范围。如果第二个操作数是 `Query` 对象，那么这个子查询的结果集将会作为第一个操作符的字段或者 DB 表达式的取值范围。 例如， `['in', 'id', [1, 2, 3]]` 将生成 `id IN (1, 2, 3)`。 该方法将正确地为字段名加引号以及为取值范围转义。`in` 操作符还支持组合字段，此时， 操作数1应该是一个字段名数组，而操作数2应该是一个数组或者 `Query` 对象， 代表这些字段的取值范围。
- `not in`: 用法和 `in` 操作符类似，这里就不再赘述。
- `like`: 第一个操作数应为一个字段名称或 DB 表达式， 第二个操作数可以使字符串或数组， 代表第一个操作数需要模糊查询的值。比如，`['like', 'name', 'tester']` 会生成 `name LIKE '%tester%'`。 如果范围值是一个数组，那么将会生成用 `AND` 串联起来的 多个 `like` 语句。例如，`['like', 'name', ['test', 'sample']]` 将会生成 `name LIKE '%test%' AND name LIKE '%sample%'`。 你也可以提供第三个可选的操作数来指定应该如何转义数值当中的特殊字符。 该操作数是一个从需要被转义的特殊字符到转义副本的数组映射。 如果没有提供这个操作数，将会使用默认的转义映射。如果需要禁用转义的功能， 只需要将参数设置为 `false` 或者传入一个空数组即可。需要注意的是， 当使用转义映射（又或者没有提供第三个操作数的时候），第二个操作数的值的前后将会被加上百分号。

注意： 当使用 PostgreSQL 的时候你还可以使用 `ilike`[17]， 该方法对大小写不敏感。

- `or like`: 用法和 `like` 操作符类似，区别在于当第二个操作数为数组时， 会使用 `OR` 来串联多个 `LIKE` 条件语句。
- `not like`: 用法和 `like` 操作符类似，区别在于会使用 `NOT LIKE` 来生成条件语句。
- `or not like`: 用法和 `not like` 操作符类似，区别在于会使用 `OR` 来串联多个 `NOT LIKE` 条件语句。
- `exists`: 需要一个操作数，该操作数必须是代表子查询 `yii\db\Query` 的一个实例， 它将会构建一个 `EXISTS (sub-query)` 表达式。
- `not exists`: 用法和 `exists` 操作符类似，它将创建一个 `NOT EXISTS (sub-query)` 表达式。
- `>`，`<=` 或者其他包含两个操作数的合法 DB 操作符：第一个操作数必须为字段的名称， 而第二个操作数则应为一个值。例如，`['>', 'age', 10]` 将会生成 `age>10`。

使用操作符格式，Yii 在内部对相应的值进行参数绑定，因此与 字符串格式相比， 此处你不需要手动添加参数。但请注意，Yii 不会帮你转义列名，所以如果你 从用户端获得的变量作为列名而没有进行任何额外的检查，对于 SQL 注入攻击， 你的程序将变得很脆弱。为了保证应用程序的安全，请不要将变量用作列名 或者你必须用白名单过滤变量。如果你实在需要从用户获取列名，请阅读 过滤数据 章节。例如，以下代码易受攻击：

```
// 易受攻击的代码:
$column = $request->get('column');
$value = $request->get('value');
$query->where(['=', $column, $value]);
// $value 是安全的，但是 $column 名不会被转义处理！
```

对象格式（Object Form） 对象格式自 2.0.14 开始提供，是定义条件的最强大和最复杂的方法。 如果你要在查询构建器上构建自己的抽象方法或者如果你要实现自己的复杂条件， 你需要它（Object Form）

条件类的实例是不可变的。他们唯一的用途是存储条件数据 并为条件构建器提供 getters 属性。条件构建器是一个包含转换逻辑的类， 它将存储的条件数据转换为 SQL 表达式。

在内部，上面描述的格式在构建 SQL 之前被隐式转换为对象格式， 因此可以在单一条件语句下组合适合的格式：

```
$query->andWhere(new OrCondition([
    new InCondition('type', 'in', $types),
    ['like', 'name', '%good%'],
    'disabled=false'
]))
```

[17] https://www.postgresql.org/docs/8.3/functions-matching.html#FUNCTIONS-LIKE

操作符格式与对象格式的对应关系是在 `QueryBuilder::conditionClasses` 属性中定义， 这里列举一些比较有代表性的映射关系：

- `AND, OR -> yii\db\conditions\ConjunctionCondition`
- `NOT -> yii\db\conditions\NotCondition`
- `IN, NOT IN -> yii\db\conditions\InCondition`
- `BETWEEN, NOT BETWEEN -> yii\db\conditions\BetweenCondition`

等等

使用对象格式可以定义自己的条件集，并且可以更容易维护别人定义的条件集。（注：这里是说对象比数组更可靠） 更多细节请参考 Adding Custom Conditions and Expressions 章节。

附加条件 你可以使用 `andWhere()` 或者 `orWhere()` 在原有条件的基础上 附加额外的条件。你可以多次调用这些方法来分别追加不同的条件。 例如，

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

如果 `$search` 不为空，那么将会生成如下 SQL 语句：

```
... WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

过滤条件 当 `WHERE` 条件来自于用户的输入时，你通常需要忽略用户输入的空值。 例如，在一个可以通过用户名或者邮箱搜索的表单当中，用户名或者邮箱 输入框没有输入任何东西，这种情况下你想要忽略掉对应的搜索条件， 那么你就可以使用 `filterWhere()` 方法来实现这个目的：

```
// $username 和 $email 来自于用户的输入
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

`filterWhere()` 和 `where()` 唯一的不同就在于，前者 将忽略在条件当中的hash format的空值。所以如果 `$email` 为空而 `$username` 不为空，那么上面的代码最终将生产如下 SQL `...WHERE username=:username`。

> **提示：** 当一个值为 `null`、空数组、空字符串或者一个只包含空格的字符串时，那么它将被判定为空值。

类似于 andWhere() 和 orWhere()， 你可以使用 andFilterWhere() 和 orFilterWhere() 方法 来追加额外的过滤条件。

此外，yii\db\Query::andFilterCompare() 可以根据值中的内容智能地确定运算符：

```
$query->andFilterCompare('name', 'John Doe');
$query->andFilterCompare('rating', '>9');
$query->andFilterCompare('value', '<=100');
```

您还可以显式指定运算符：

```
$query->andFilterCompare('name', 'Doe', 'like');
```

Yii 自 2.0.11 版起 ，提供了 HAVING 条件的一些构建方法：

- filterHaving()
- andFilterHaving()
- orFilterHaving()

orderBy()

orderBy() 方法是用来指定 SQL 语句当中的 ORDER BY 子句的。例如，

```
// ... ORDER BY `id` ASC, `name` DESC
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

如上所示，数组当中的键指代的是字段名称，而数组当中的值则表示的是排序的方式。 PHP 的常量 SORT_ASC 指的是升序排列，SORT_DESC 指的则是降序排列。

如果 ORDER BY 仅仅包含简单的字段名称，你可以使用字符串来声明它，就像写原生的 SQL 语句一样。例如，

```
$query->orderBy('id ASC, name DESC');
```

> **注意：** 当 ORDER BY 语句包含一些 DB 表达式的时候，你应该使用数组的格式。

你可以调用 addOrderBy() 来为 ORDER BY 片断添加额外的子句。 例如，

```
$query->orderBy('id ASC')
    ->addOrderBy('name DESC');
```

groupBy()

groupBy() 方法是用来指定 SQL 语句当中的 GROUP BY 片断的。例如，

```
// ... GROUP BY `id`, `status`
$query->groupBy(['id', 'status']);
```

如果 GROUP BY 仅仅包含简单的字段名称，你可以使用字符串来声明它， 就像写原生的 SQL 语句一样。例如，

```
$query->groupBy('id, status');
```

> **注意：** 当 GROUP BY 语句包含一些 DB 表达式的时候，你应该使用数组的格式。

你可以调用 addOrderBy() 来为 GROUP BY 子句添加额外的字段。例如，

```
$query->groupBy(['id', 'status'])
    ->addGroupBy('age');
```

having()

having() 方法是用来指定 SQL 语句当中的 HAVING 子句。它带有一个条件， 和 where() 中指定条件的方法一样。例如，

```
// ... HAVING `status` = 1
$query->having(['status' => 1]);
```

请查阅 where() 的文档来获取更多有关于如何指定一个条件的细节。

你可以调用 andHaving() 或者 orHaving() 方法来为 HAVING 子句追加额外的条件，例如，

```
// ... HAVING (`status` = 1) AND (`age` > 30)
$query->having(['status' => 1])
    ->andHaving(['>', 'age', 30]);
```

limit() 和 offset()

limit() 和 offset() 是用来指定 SQL 语句当中 的 LIMIT 和 OFFSET 子句的。例如，

```
// ... LIMIT 10 OFFSET 20
$query->limit(10)->offset(20);
```

如果你指定了一个无效的 limit 或者 offset（例如，一个负数），那么它将会被忽略掉。

> **提示：** 在不支持 LIMIT 和 OFFSET 的 DBMS 中（例如，MSSQL），查询构建器将生成一条模拟 LIMIT/OFFSET 行为的 SQL 语句。

join()

join() 是用来指定 SQL 语句当中的 JOIN 子句的。例如，

```
// ... LEFT JOIN `post` ON `post`.`user_id` = `user`.`id`
$query->join('LEFT JOIN', 'post', 'post.user_id = user.id');
```

join() 带有四个参数：

- $type：连接类型，例如，'INNER JOIN'，'LEFT JOIN'。
- $table：将要连接的表名称。
- $on：可选的，连接条件，即 ON 片段。有关指定条件的详细信息，请参阅 where()。请注意，数组语法 **不能** 用于指定基于列的条件，例如，['user.id' => 'comment.userId'] 将导致用户 id 必须等于字符串 'comment.userId' 的情况。您应该使用字符串语法，并将条件指定为 'user.id = comment.userId'。
- $params：可选参数，与连接条件绑定的参数。

你可以分别调用如下的快捷方法来指定 INNER JOIN, LEFT JOIN 和 RIGHT JOIN。

- innerJoin()
- leftJoin()
- rightJoin()

例如，

```
$query->leftJoin('post', 'post.user_id = user.id');
```

可以通过多次调用如上所述的连接方法来连接多张表，每连接一张表调用一次。

除了连接表以外，你还可以连接子查询。方法如下，将需要被连接的子查询指定 为一个 yii\db\Query 对象，例如，

```
$subQuery = (new \yii\db\Query())->from('post');
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

在这个例子当中，你应该将子查询放到一个数组当中，而数组当中的键，则为这个子查询的别名。

union()

union() 方法是用来指定 SQL 语句当中的 UNION 子句的。例如，

```
$query1 = (new \yii\db\Query())
    ->select("id, category_id AS type, name")
    ->from('post')
    ->limit(10);

$query2 = (new \yii\db\Query())
    ->select('id, type, name')
    ->from('user')
    ->limit(10);

$query1->union($query2);
```

你可以通过多次调用 union() 方法来追加更多的 UNION 子句。

6.2.2 查询方法

yii\db\Query 提供了一整套的用于不同查询目的的方法。

- all()：将返回一个由行组成的数组，每一行是一个由名称和值构成的关联数组（译者注：省略键的数组称为索引数组）。
- one()：返回结果集的第一行。
- column()：返回结果集的第一列。
- scalar()：返回结果集的第一行第一列的标量值。
- exists()：返回一个表示该查询是否包结果集的值。
- count()：返回 COUNT 查询的结果。
- 其它集合查询方法：包括 sum($q)，average($q)，max($q)，min($q) 等。$q 是一个必选参数， 既可以是一个字段名称，又可以是一个 DB 表达式。

例如，

```
// SELECT `id`, `email` FROM `user`
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->all();

// SELECT * FROM `user` WHERE `username` LIKE `%test%`
$row = (new \yii\db\Query())
    ->from('user')
    ->where(['like', 'username', 'test'])
    ->one();
```

> **注意：** one() 方法只返回查询结果当中的第一条数据， 条件语句中不会加上 LIMIT 1 条件。如果你清楚的知道查询将会只返回一行或几行数据 （例如， 如果你是通过某些主键来查询的），这很好也提倡这样做。但是，如果查询结果 有机会返回大量的数据时，那么你应该显示调用 limit(1) 方法，以改善性能。 例如，(new \yii\db\Query())->from('user')->limit(1)->one()。

所有的这些查询方法都有一个可选的参数 $db，该参数指代的是 DB connection，执行一个 DB 查询时会用到。如果你省略了这个参数，那么 db application component 将会被用作 默认的 DB 连接。 如下是另外一个使用 count() 查询的例子：

```
// 执行 SQL: SELECT COUNT(*) FROM `user` WHERE `last_name`=:last_name
$count = (new \yii\db\Query())
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->count();
```

当你调用 yii\db\Query 当中的一个查询方法的时候，实际上内在的运作机制如下：

- 在当前 yii\db\Query 的构造基础之上，调用 yii\db\QueryBuilder 来生成一条 SQL 语句；

- 利用生成的 SQL 语句创建一个 yii\db\Command 对象；
- 调用 yii\db\Command 的查询方法（例如，queryAll()）来执行这条 SQL 语句，并检索数据。

有时候，你也许想要测试或者使用一个由 yii\db\Query 对象创建的 SQL 语句。 你可以使用以下的代码来达到目的：

```
$command = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->createCommand();

// 打印 SQL 语句
echo $command->sql;
// 打印被绑定的参数
print_r($command->params);

// 返回查询结果的所有行
$rows = $command->queryAll();
```

索引查询结果

当你在调用 all() 方法时，它将返回一个以连续的整型数值为索引的数组。 而有时候你可能希望使用一个特定的字段或者表达式的值来作为索引结果集数组。那么你可以在调用 all() 之前使用 indexBy() 方法来达到这个目的。 例如，

```
// 返回 [100 => ['id' => 100, 'username' => '...', ...], 101 => [...], 103
=> [...], ...]
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

如需使用表达式的值做为索引，那么只需要传递一个匿名函数给 indexBy() 方法即可：

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

该匿名函数将带有一个包含了当前行的数据的 $row 参数，并且返回用作当前行索引的 标量值（译者注：就是简单的数值或者字符串，而不是其他复杂结构，例如数组）。

> **注意：** 与 `groupBy()` 或者 `orderBy()` 等查询方法不同，他们将转换为 SQL 查询语句的一部分，而这个方法（indexBy）在从数据库取回数据后才生效执行的。这意味着只能使用那些在你的 SELECT 查询中的列名。此外，你用表名连接取列名的时候，比如 `customer.id`，结果中将只包含 `id` 列，因此你必须调用 `->indexBy('id')` 不要带表名前缀。

批处理查询

当需要处理大数据的时候，像 `yii\db\Query::all()` 这样的方法就不太合适了，因为它们会把所有查询的数据都读取到客户端内存上。为了解决这个问题，Yii 提供了批处理查询的支持。服务端先保存查询结果，然后客户端使用游标（cursor）每次迭代出固定的一批结果集回来。

> **警告：** MySQL 批处理查询的实现存在已知的局限性和变通方法。见下文。

批处理查询的用法如下：

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users 是一个包含100条或小于100条用户表数据的数组
}

// or to iterate the row one by one
foreach ($query->each() as $user) {
    // 数据从服务端中以 100 个为一组批量获取，
    // 但是 $user 代表 user 表里的一行数据
}
```

`yii\db\Query::batch()` 和 `yii\db\Query::each()` 方法将会返回一个实现了`Iterator` 接口 `yii\db\BatchQueryResult` 的对象，可以用在 `foreach` 结构当中使用。在第一次迭代取数据的时候，数据库会执行一次 SQL 查询，然后在剩下的迭代中，将直接从结果集中批量获取数据。默认情况下，一批的大小为 100，也就意味着一批获取的数据是 100 行。你可以通过给 `batch()` 或者 `each()` 方法的第一个参数传值来改变每批行数的大小。

相对于 `yii\db\Query::all()` 方法，批处理查询每次只读取 100 行的数据到内存。

如果你通过 `yii\db\Query::indexBy()` 方法为查询结果指定了索引字段，那么批处理查询将仍然保持相对应的索引方案，例如，

```
$query = (new \yii\db\Query())
    ->from('user')
```

```
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users 的 "username" 字段将会成为索引
}

foreach ($query->each() as $username => $user) {
    // ...
}
```

MySQL中批量查询的局限性（Limitations of batch query in MySQL）

MySQL 是通过 PDO 驱动库实现批量查询的。默认情况下，MySQL 查询是 带缓存的[18]， 这违背了使用游标（cursor）获取数据的目的， 因为它不阻止驱动程序将整个结果集加载到客户端的内存中。

> **注意：** 当使用 `libmysqlclient` 时（PHP5 的标配），计算 PHP 的内存限制时，用于数据结果集的内存不会计算在内。 看上去批量查询是正确运行的，实际上整个数据集都被加载到了客户端的内存中， 而且这个使用量可能还会再增长。

要禁用缓存并减少客户端内存的需求量，PDO 连接属性 `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` 必须设置为 `false`。 这样，直到整个数据集被处理完毕前，通过此连接是无法创建其他查询的。 这样的操作可能会阻碍 `ActiveRecord` 执行表结构查询。 如果这不构成问题（表结构已被缓存过了）， 我们可以通过切换原本的连接到非缓存模式，然后在批量查询完成后再切换回来。

```
Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY,
false);

// 执行批量查询

Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, true);
```

> **注意：** 对于 MyISAM，在执行批量查询的过程中，表可能将被锁， 将延迟或拒绝其他连接的写入操作。 当使用非缓存查询时，尽量缩短游标打开的时间。

如果表结构没有被缓存，或在批量查询被处理过程中需要执行其他查询，你可以创建一个单独的非缓存链接到数据库：

```
$unbufferedDb = new \yii\db\Connection([
    'dsn' => Yii::$app->db->dsn,
    'username' => Yii::$app->db->username,
    'password' => Yii::$app->db->password,
    'charset' => Yii::$app->db->charset,
]);
$unbufferedDb->open();
$unbufferedDb->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY,
false);
```

[18] https://www.php.net/manual/zh/mysqlinfo.concepts.buffering.php

如果你除了 `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` 是 `false` 之外， 要确保 `$unbufferedDb` 拥有和原来缓存 `$db` 完全一样的属性， 请参阅实现 `$db` 的深度拷贝[19]， 手动方法将它设置为 false 即可。

然后使用此连接正常创建查询，新连接用于运行批量查询， 逐条或批量进行结果处理：

```
// 获取 1000 为一组的批量数据
foreach ($query->batch(1000, $unbufferedDb) as $users) {
    // ...
}


// 每次从服务端批量获取1000个数据，但是逐个遍历进行处理
foreach ($query->each(1000, $unbufferedDb) as $user) {
    // ...
}
```

当结果集已处理完毕不再需要连接时，可以关闭它：

```
$unbufferedDb->close();
```

> **注意：** 非缓存查询在 PHP 端使用更少的缓存，但会增加 MySQL 服务器端的负载。 建议您使用生产实践设计自己的代码以获取额外的海量数据，例如，将数字键分段，使用非缓存的查询遍历[20]。

添加自定义查询条件和表达式（Adding custom Conditions and Expressions）

我们在 查询条件-对象格式 章节中提到过，可以创建自定义的查询条件类。 举个栗子，我们需要创建一个查询条件，它可以检查某些字段小于特定值的情况。 当使用操作符格式时，代码如下：

```
[
    'and',
    '>', 'posts', $minLimit,
    '>', 'comments', $minLimit,
    '>', 'reactions', $minLimit,
    '>', 'subscriptions', $minLimit
]
```

当这样的查询条件仅被应用一次，没什么问题。当它在一个查询语句中被多次使用时，就有很多优化点了。 我们创建一个自定义查询条件对象来证实它。

Yii 有 `ConditionInterface` 接口类，必须用它来标识这是一个表示查询条件的类。 它需要实现 `fromArrayDefinition()` 方法，用来从数组格式创建查询条件。 如果我们不需要它，抛出一个异常来完成此方法即可。

创建自定义查询条件类，我们就可以构建最适合当前需求的 API。

[19] https://github.com/yiisoft/yii2/issues/8420#issuecomment-301423833
[20] https://github.com/yiisoft/yii2/issues/8420#issuecomment-296109257

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
{
    private $columns;
    private $value;

    /**
     * @param string[] $columns 要大于 $value 的字段名数组
     * @param mixed $value 每个 $column 要比较的数值
     */
    public function __construct(array $columns, $value)
    {
        $this->columns = $columns;
        $this->value = $value;
    }

    public static function fromArrayDefinition($operator, $operands)
    {
        throw new InvalidArgumentException('Not implemented yet, but we will
        do it later');
    }

    public function getColumns() { return $this->columns; }
    public function getValue() { return $this->vaule; }
}
```

我们现在创建了一个查询条件对象：

```
$conditon = new AllGreaterCondition(['col1', 'col2'], 42);
```

但是 `QueryBuilder` 还不知道怎样从此对象生成 SQL 查询条件。 因此我们还需要为这个条件对象创建一个构建器（Builder）。 这个构建器必须实现 `yii\db\ExpressionBuilderInterface` 接口和 `build()` 方法。

```
namespace app\db\conditions;

class AllGreaterConditionBuilder implements
\yii\db\ExpressionBuilderInterface
{
    use \yii\db\ExpressionBuilderTrait; // Contains constructor and
    `queryBuilder` property.

    /**
     * @param ExpressionInterface $condition 要构建的查询条件对象
     * @param array $params 绑定的参数
     * @return AllGreaterCondition
     */
    public function build(ExpressionInterface $expression, array &$params =
    [])
    {
        $value = $condition->getValue();
```

```
        $conditions = [];
        foreach ($expression->getColumns() as $column) {
            $conditions[] = new SimpleCondition($column, '>', $value);
        }

        return $this->queryBuilder->buildCondition(new
        AndCondition($conditions), $params);
    }
}
```

接下来，让 QueryBuilder]知道我们的新查询条件对象 – 添加一个映射到 expressionBuilders 数组中，在应用配置（application config）中完成即可：

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' =>
            'app\db\conditions\AllGreaterConditionBuilder',
        ],
    ],
],
```

现在我们可以在 where() 中使用此查询条件对象了：

```
$query->andWhere(new AllGreaterCondition(['posts', 'comments', 'reactions',
'subscriptions'], $minValue));
```

如果我们想要自定义操作符查询条件，可以在 QueryBuilder::conditionClasses 中 这样声明：

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' =>
            'app\db\conditions\AllGreaterConditionBuilder',
        ],
        'conditionClasses' => [
            'ALL>' => 'app\db\conditions\AllGreaterCondition',
        ],
    ],
],
```

并在 app\db\conditions\AllGreaterCondition 对象中实现 AllGreaterCondition::fromArrayDefinition() 法：

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
```

```
{
    // ... 这里省略其他方法的实现

    public static function fromArrayDefinition($operator, $operands)
    {
        return new static($operands[0], $operands[1]);
    }
}
```

然后呢，我们就可以使用更简短的操作符格式来创建自定义查询条件了：

```
$query->andWhere(['ALL>', ['posts', 'comments', 'reactions',
'subscriptions'], $minValue]);
```

你可能注意到了，这里使用到了两个概念：表达式对象和条件对象。表达式对象实现了 `yii\db\ExpressionInterface` 接口， 它还依赖于一个表达式构建器（Expression Builder）来执行构建逻辑，而表达式构建器实现了 `yii\db\ExpressionBuilderInterface` 接口。 而条件对象实现了 `yii\db\condition\ConditionInterface` 接口，它是继承了 `ExpressionInterface` 接口， 如上面的栗子所写的，它用于数组定义的条件的场景，当然条件对象也需要构建器。

总结起来就是：

- Expression – 表达式对象，是数据集的数据转换对象（DTO），它可以被编译为一些特定 SQL 语句 （操作符、字符串、数组、JSON等等）。
- Condition – 条件对象，是表达式对象超集，它可以聚合多个表达式对象（或标量值），然后编译成一条 SQL 查询条件。

你可以创建自己的类来实现 `ExpressionInterface` 接口，达到封装的目的：隐藏复杂的 SQL 语句拼装过程。 想学习到更多关于表达式对象的实践请听 下回分解；

6.3 活动记录（Active Record）

Active Record[21] 提供了一个面向对象的接口， 用以访问和操作数据库中的数据。Active Record 类与数据库表关联， Active Record 实例对应于该表的一行， Active Record 实例的属性表示该行中特定列的值。 您可以访问 Active Record 属性并调用 Active Record 方法来访问和操作存储在数据库表中的数据， 而不用编写原始 SQL 语句。

例如，假定 `Customer` Active Record 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哉 `customer` 表里插入一行新的记录：

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

[21] https://zh.wikipedia.org/wiki/%E4%B8%BB%E5%8A%A8%E8%AE%B0%E5%BD%95

对于 MySQL，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' => 'Qiang',
])->execute();
```

Yii 为以下关系数据库提供 Active Record 支持：

- MySQL 4.1 及以上：通过 `yii\db\ActiveRecord` 支持
- PostgreSQL 7.3 及以上：通过 `yii\db\ActiveRecord` 支持
- SQLite 2 and 3：通过 `yii\db\ActiveRecord` 支持
- Microsoft SQL Server 2008 及以上：通过 `yii\db\ActiveRecord` 支持
- Oracle：通过 `yii\db\ActiveRecord` 支持
- CUBRID 9.3 及以上：通过 `yii\db\ActiveRecord` 支持 (提示， 由于 CUBRID PDO 扩展的 bug[22]， 给变量加引用将不起作用，所以你得使用 CUBRID 9.3 客户端及服务端。
- Sphinx：通过 `yii\sphinx\ActiveRecord` 支持，依赖 `yii2-sphinx` 扩展
- ElasticSearch：通过 `yii\elasticsearch\ActiveRecord` 支持，依赖 `yii2-elasticsearch` 扩展

此外，Yii 的 Active Record 功能还支持以下 NoSQL 数据库：

- Redis 2.6.12 及以上：通过 `yii\redis\ActiveRecord` 支持，依赖 `yii2-redis` 扩展
- MongoDB 1.3.0 及以上：通过 `yii\mongodb\ActiveRecord` 支持，依赖 `yii2-mongodb` 扩展

在本教程中，我们会主要描述对关系型数据库的 Active Record 用法。 然而，绝大多数的内容在 NoSQL 的 Active Record 里同样适用。

6.3.1 声明 Active Record 类（Declaring Active Record Classes）

要想声明一个 Active Record 类，你需要声明该类继承 `yii\db\ActiveRecord`。

设置表的名称（Setting a table name）

默认的，每个 Active Record 类关联各自的数据库表。 经过 `yii\helpers\Inflector::camel2id()` 处理，`tableName()` 方法默认返回的表名称是通过类名转换来得。 如果这个默认名称不正确，你得重写这个方法。

此外，`tablePrefix` 表前缀也会起作用。例如，如果 `tablePrefix` 表前缀是 `tbl_`，`Customer` 的类名将转换成 `tbl_customer` 表名，`OrderItem` 转换成 `tbl_order_item`。

[22] http://jira.cubrid.org/browse/APIS-658

如果你定义的表名是 `{{%TableName}}`，百分比字符 `%` 会被替换成表前缀。例如，`{{%post}}` 会变成 `{{tbl_post}}`。表名两边的括号会被 SQL 查询引用处理。

下面的例子中，我们给 `customer` 数据库表定义叫 `Customer` 的 Active Record 类。

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string Active Record 类关联的数据库表名称
     */
    public static function tableName()
    {
        return '{{customer}}';
    }
}
```

将 Active Record 称为模型（Active records are called “models”）

Active Record 实例称为模型。因此，我们通常将 Active Record 类 放在 `app\models` 命名空间下（或者其他保存模型的命名空间）。

因为 `yii\db\ActiveRecord` 继承了模型 `yii\base\Model`，它就拥有所有模型特性， 比如说属性（attributes），验证规则（rules），数据序列化（data serialization），等等。

6.3.2 建立数据库连接（Connecting to Databases）

活动记录 Active Record 默认使用 `db` 组件 作为连接器 `DB connection` 访问和操作数据库数据。 基于数据库访问中的解释，你可以在系统配置中 这样配置 `db` 组件。

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

如果你要用不同的数据库连接，而不仅仅是 `db` 组件， 你可以重写 `getDb()` 方法。

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // 使用 "db2" 组件
        return \Yii::$app->db2;
    }
}
```

6.3.3 查询数据（Querying Data）

定义 Active Record 类后，你可以从相应的数据库表中查询数据。 查询过程大致如下三个步骤:

1. 通过 `yii\db\ActiveRecord::find()` 方法创建一个新的查询生成器对象;
2. 使用查询生成器的构建方法来构建你的查询;
3. 调用查询生成器的查询方法来取出数据到 Active Record 实例中。

正如你看到的，是不是跟查询生成器的步骤差不多。 唯一有区别的地方在于你用 `yii\db\ActiveRecord::find()` 去获得一个新的查询生成器对象，这个对象是 `yii\db\ActiveQuery`， 而不是使用 `new` 操作符创建一个查询生成器对象。

下面是一些例子，介绍如何使用 Active Query 查询数据:

```
// 返回 ID 为 123 的客户:
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// 取回所有活跃客户并以他们的 ID 排序:
// SELECT * FROM `customer` WHERE `status` = 1 ORDER BY `id`
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// 取回活跃客户的数量:
// SELECT COUNT(*) FROM `customer` WHERE `status` = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// 以客户 ID 索引结果集:
// SELECT * FROM `customer`
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

上述代码中，`$customer` 是个 `Customer` 对象，而 `$customers` 是个以 `Customer` 对象为元素的数组。 它们两都是以 `customer` 表中取回的数据结果集填充的。

> **提示：** 由于 `yii\db\ActiveQuery` 继承 `yii\db\Query`， 你可以使用 Query Builder 章节里所描述的所有查询方法。

根据主键获取数据行是比较常见的操作，所以 Yii 提供了两个快捷方法：

- `yii\db\ActiveRecord::findOne()`：返回一个 Active Record 实例，填充于查询结果的第一行数据。
- `yii\db\ActiveRecord::findAll()`：返回一个 Active Record 实例的数据，填充于查询结果的全部数据。

这两个方法的传参格式如下：

- 标量值：这个值会当作主键去查询。 Yii 会通过读取数据库模式信息来识别主键列。
- 标量值的数组：这数组里的值都当作要查询的主键的值。
- 关联数组：键值是表的列名，元素值是相应的要查询的条件值。 可以到 哈希格式 查看更多信息。

如下代码描述如何使用这些方法：

```
// 返回 id 为 123 的客户
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// 返回 id 是 100, 101, 123, 124 的客户
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// 返回 id 是 123 的活跃客户
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// 返回所有不活跃的客户
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

> **警告：** 如果你需要将用户输入传递给这些方法，请确保输入值是标量或者是 数组条件，确保数组结构不能被外部所改变：
>
> ```
> // yii\web\Controller 确保了 $id 是标量
> public function actionView($id)
> {
>  $model = Post::findOne($id);
>  // ...
> }
> ```

```
// 明确了指定要搜索的列，在此处传递标量或数组将始终只是查找出单个记录而已
$model = Post::findOne(['id' => Yii::$app->request->get('id')]);

// 不要使用下面的代码！可以注入一个数组条件来匹配任意列的值！
$model = Post::findOne(Yii::$app->request->get('id'));
```

> **提示：** `yii\db\ActiveRecord::findOne()` 和 `yii\db\ActiveQuery::one()` 都不会添加 `LIMIT 1` 到 生成的 SQL 语句中。如果你的查询会返回很多行的数据， 你明确的应该加上 `limit(1)` 来提高性能，比如 `Customer::find()->limit(1)->one()`。

除了使用查询生成器的方法之外，你还可以书写原生的 SQL 语句来查询数据，并填充结果集到 Active Record 对象中。 通过使用 `yii\db\ActiveRecord::findBySql()` 方法:

```
// 返回所有不活跃的客户
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, [':status' =>
Customer::STATUS_INACTIVE])->all();
```

不要在 `findBySql()` 方法后加其他查询方法了， 多余的查询方法都会被忽略。

6.3.4 访问数据（Accessing Data）

如上所述，从数据库返回的数据被填充到 Active Record 实例中， 查询结果的每一行对应于单个 Active Record 实例。 您可以通过 Active Record 实例的属性来访问列值，例如，

```
// "id" 和 "email" 是 "customer" 表中的列名
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

> **提示：** Active Record 的属性以区分大小写的方式为相关联的表列命名的。 Yii 会自动为关联表的每一列定义 Active Record 中的一个属性。 您不应该重新声明任何属性。

由于 Active Record 的属性以表的列名命名，可能你会发现你正在编写像这样的 PHP 代码： `$customer->first_name`，如果你的表的列名是使用下划线分隔的，那么属性名中的单词 以这种方式命名。 如果您担心代码风格一致性的问题，那么你应当重命名相应的表列名 （例如使用骆驼拼写法）。

数据转换（Data Transformation）

常常遇到，要输入或显示的数据是一种格式，而要将其存储在数据库中是另一种格式。 例如，在数据库中，您将客户的生日存储为 UNIX 时间

截（虽然这不是一个很好的设计），而在大多数情况下，你想以字符串 `'YYYY/MM/DD'` 的格式处理生日数据。为了实现这一目标，您可以在 `Customer` 中定义 数据转换 方法 定义 Active Record 类如下：

```
class Customer extends ActiveRecord
{
    // ...

    public function getBirthdayText()
    {
        return date('Y/m/d', $this->birthday);
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

现在你的 PHP 代码中，你可以访问 `$customer->birthdayText`，来以 `'YYYY/MM/DD'` 的格式输入和显示客户生日，而不是访问 `$customer->birthday`。

> **提示：** 上述示例显示了以不同格式转换数据的通用方法。如果你正在使用 日期值，您可以使用 DateValidator 和 `yii\jui\DatePicker` 来操作，这将更易用，更强大。

以数组形式获取数据（Retrieving Data in Arrays）

通过 Active Record 对象获取数据十分方便灵活，与此同时，当你需要返回大量的数据的时候，这样的做法并不令人满意，因为这将导致大量内存占用。在这种情况下，您可以 在查询方法前调用 `asArray()` 方法，来获取 PHP 数组形式的结果：

```
// 返回所有客户
// 每个客户返回一个关联数组
$customers = Customer::find()
    ->asArray()
    ->all();
```

> **提示：** 虽然这种方法可以节省内存并提高性能，但它更靠近较低的 DB 抽象层 你将失去大部分的 Active Record 提供的功能。一个非常重要的区别在于列值的数据类型。当您在 Active Record 实例中返回数据时，列值将根据实际列类型，自动类型转换；然而，当您以数组返回数据时，列值将为 字符串（因为它们是没有处理过的 PDO 的结果），不管它们的实际列是什么类型。

批量获取数据（Retrieving Data in Batches）

在 查询生成器 中，我们已经解释说可以使用 批处理查询 来最小化你的内存使用， 每当从数据库查询大量数据。你可以在 Active Record 中使用同样的技巧。例如，

```
// 每次获取 10 条客户数据
foreach (Customer::find()->batch(10) as $customers) {
    // $customers 是个最多拥有 10 条数据的数组
}

// 每次获取 10 条客户数据，然后一条一条迭代它们
foreach (Customer::find()->each(10) as $customer) {
    // $customer 是个 `Customer` 对象
}

// 贪婪加载模式的批处理查询
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer 是个 `Customer` 对象，并附带关联的 ``'orders'``
}
```

6.3.5 保存数据（Saving Data）

使用 Active Record，您可以通过以下步骤轻松地将数据保存到数据库：

1. 准备一个 Active Record 实例
2. 将新值赋给 Active Record 的属性
3. 调用 `yii\db\ActiveRecord::save()` 保存数据到数据库中。

例如，

```
// 插入新记录
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// 更新已存在的记录
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

`save()` 方法可能插入或者更新表的记录，这取决于 Active Record 实例的状态。 如果实例通过 `new` 操作符实例化，调用 `save()` 方法将插入新记录；如果实例是一个查询方法的结果，调用 `save()` 方法 将更新这个实例对应的表记录行。

你可以通过检查 Active Record 实例的 `isNewRecord` 属性值来区分这两个状态。 此属性也被使用在 `save()` 方法内部， 代码如下：

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

> **提示：** 你可以直接调用 `insert()` 或者 `update()` 方法来插入或更新一条记录。

数据验证（Data Validation）

因为 `yii\db\ActiveRecord` 继承于 `yii\base\Model`，它共享相同的 输入验证 功能。 你可以通过重写 `rules()` 方法声明验证规则并执行， 通过调用 `validate()` 方法进行数据验证。

当你调用 `save()` 时，默认情况下会自动调用 `validate()`。 只有当验证通过时，它才会真正地保存数据; 否则将简单地返回 `false`， 您可以检查 `errors` 属性来获取验证过程的错误消息。

> **提示：** 如果你确定你的数据不需要验证（比如说数据来自可信的场景）， 你可以调用 `save(false)` 来跳过验证过程。

块赋值（Massive Assignment）

和普通的 模型 一样，你亦可以享受 Active Record 实例的 块赋值 特性。 使用此功能，您可以在单个 PHP 语句中，给 Active Record 实例的多个属性批量赋值， 如下所示。 记住，只有 安全属性 才可以批量赋值。

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

更新计数（Updating Counters）

在数据库表中增加或减少一个字段的值是个常见的任务。我们将这些列称为“计数列”。 您可以使用 `updateCounters()` 更新一个或多个计数列。 例如，

```
$post = Post::findOne(100);

// UPDATE `post` SET `view_count` = `view_count` + 1 WHERE `id` = 100
$post->updateCounters(['view_count' => 1]);
```

> **注意：** 如果你使用 `yii\db\ActiveRecord::save()` 更新一个计数列，你最终将得到错误的结果， 因为可能发生这种情况，多个请求间并发读写同一个计数列。

脏属性（Dirty Attributes）

当您调用 `save()` 保存 Active Record 实例时，只有 脏属性 被保存。如果一个属性的值已被修改，则会被认为是 脏，因为它是从 DB 加载出来的或者 刚刚保存到 DB 。请注意，无论如何 Active Record 都会执行数据验证不管有没有脏属性。

Active Record 自动维护脏属性列表。 它保存所有属性的旧值， 并其与最新的属性值进行比较，就是酱紫个道理。你可以调用 `yii\db\ActiveRecord::getDirtyAttributes()` 获取当前的脏属性。你也可以调用 `yii\db\ActiveRecord::markAttributeDirty()` 将属性显式标记为脏。

如果你有需要获取属性原先的值，你可以调用 `getOldAttributes()` 或者 `getOldAttribute()`。

> 注：属性新旧值的比较是用 `===` 操作符，所以一样的值但类型不同， 依然被认为是脏的。当模型从 HTML 表单接收用户输入时，通常会出现这种情况， 其中每个值都表示为一个字符串类型。 为了确保正确的类型，比如，整型需要用过滤验证器：`['attributeName', 'filter', 'filter' => 'intval']`。其他 PHP 类型转换函数一样适用，像 intval()[23]， floatval()[24]， boolval[25]，等等

默认属性值（Default Attribute Values）

某些表列可能在数据库中定义了默认值。有时，你可能想预先填充 具有这些默认值的 Active Record 实例的 Web 表单。 为了避免再次写入相同的默认值， 您可以调用 `loadDefaultValues()` 来填充 DB 定义的默认值 进入相应的 Active Record 属性：

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz 将被 "zyz" 列定义的默认值赋值
```

属性类型转换（Attributes Typecasting）

在查询结果填充 `yii\db\ActiveRecord` 时，将自动对其属性值执行类型转换，基于 数据库表模式 中的信息。 这允许从数据表中获取数据， 声明为整型的，使用 PHP 整型填充 ActiveRecord 实例，布尔值（boolean）的也用布尔值填充，等等。 但是，类型转换机制有几个限制：

[23] https://www.php.net/manual/zh/function.intval.php
[24] https://www.php.net/manual/zh/function.floatval.php
[25] https://www.php.net/manual/zh/function.boolval.php

- 浮点值不被转换，并且将被表示为字符串，否则它们可能会使精度降低。
- 整型值的转换取决于您使用的操作系统的整数容量。尤其是： 声明为“无符号整型”或“大整型”的列的值将仅转换为 64 位操作系统的 PHP 整型， 而在 32 位操作系统中 - 它们将被表示为字符串。

值得注意的是，只有在从查询结果填充 ActiveRecord 实例时才执行属性类型转换。 而从 HTTP 请求加载的值或直接通过属性访问赋值的，没有自动转换。 在准备用于在 ActiveRecord 保存时，准备 SQL 语句还使用了表模式，以确保查询时 值绑定到具有正确类型的。但是，ActiveRecord 实例的属性值不会 在保存过程中转换。

> **提示：** 你可以使用 `yii\behaviors\AttributeTypecastBehavior` 来简化属性的类型转换 在 ActiveRecord 验证或者保存过程中。

从 2.0.14 开始，Yii ActiveRecord 支持了更多的复杂数据类型，例如 JSON 或多维数组。

MySQL 和 PostgreSQL 中的 JSON（JSON in MySQL and PostgreSQL） 数据填充后，基于 JSON 标准解码规则， 来自 JSON 列的值将自动解码。

另一方面，为了将属性值保存到 JSON 列中，ActiveRecord 会自动创建一个 `JsonExpression` 对象， 这对象将在 QueryBuilder 层被编码成 JSON 字符串。

PostgreSQL 中的数组（Arrays in PostgreSQL） 数据填充后，来自 Array 列的值将自动从 PgSQL 的编码值解码为 一个 `ArrayExpression` 对象。它继承于 PHP 的 `ArrayAccess` 接口，所以你可以把它当作一个数组用，或者调用 `->getValue()` 来获取数组本身。

另一方面，为了将属性值保存到数组列，ActiveRecord 会自动创建一个 `ArrayExpression` 对象， 这对象将在 QueryBuilder 中被编码成数组的 PgSQL 字符串表达式。

你还可以这样使用 JSON 列的条件：

```
$query->andWhere(['=', 'json', new ArrayExpression(['foo' => 'bar'])
```

要详细了解表达式构建系统，可以访问 Query Builder – 增加自定义条件和语句 文章。

更新多个数据行（Updating Multiple Rows）

上述方法都可以用于单个 Active Record 实例，以插入或更新单条 表数据行。 要同时更新多个数据行，你应该调用 `updateAll()` 这是一个静态方法。

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com%`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',
'@example.com']);
```

同样，你可以调用 updateAllCounters() 同时更新多条记录的计数列。

```
// UPDATE `customer` SET `age` = `age` + 1
Customer::updateAllCounters(['age' => 1]);
```

6.3.6 删除数据（Deleting Data）

要删除单行数据，首先获取与该行对应的 Active Record 实例，然后调用 yii\db\ActiveRecord::delete() 方法。

```
$customer = Customer::findOne(123);
$customer->delete();
```

你可以调用 yii\db\ActiveRecord::deleteAll() 方法删除多行甚至全部的数据。例如，

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

> **提示：** 调用 deleteAll() 时要非常小心，因为如果在指定条件时出错， 它可能会完全擦除表中的所有数据。

6.3.7 Active Record 的生命周期（Active Record Life Cycles）

当你实现各种功能的时候，会发现了解 Active Record 的生命周期很重要。在每个生命周期中，一系列的方法将被调用执行，您可以重写这些方法 以定制你要的生命周期。您还可以响应触发某些 Active Record 事件 以便在生命周期中注入您的自定义代码。这些事件在开发 Active Record 的 行为时特别有用， 通过行为可以定制 Active Record 生命周期的 。

下面，我们将总结各种 Active Record 的生命周期，以及生命周期中 所涉及的各种方法、事件。

实例化生命周期（New Instance Life Cycle）

当通过 new 操作符新建一个 Active Record 实例时，会发生以下生命周期：

1. 类的构造函数调用.
2. init()：触发 EVENT_INIT 事件。

查询数据生命周期（Querying Data Life Cycle）

当通过 查询方法 查询数据时，每个新填充出来的 Active Record 实例 将发生下面的生命周期：

1. 类的构造函数调用。
2. init()：触发 EVENT_INIT 事件。
3. afterFind()：触发 EVENT_AFTER_FIND 事件。

保存数据生命周期（Saving Data Life Cycle）

当通过 `save()` 插入或更新 Active Record 实例时 会发生以下生命周期：

1. `beforeValidate()`：触发 `EVENT_BEFORE_VALIDATE` 事件。如果这方法返回 `false` 或者 `yii\base\ModelEvent::$isValid` 值为 `false`，接下来的步骤都会被跳过。
2. 执行数据验证。如果数据验证失败，步骤 3 之后的步骤将被跳过。
3. `afterValidate()`：触发 `EVENT_AFTER_VALIDATE` 事件。
4. `beforeSave()`：触发 `EVENT_BEFORE_INSERT` 或者 `EVENT_BEFORE_UPDATE` 事件。 如果这方法返回 `false` 或者 `yii\base\ModelEvent::$isValid` 值为 `false`，接下来的步骤都会被跳过。
5. 执行真正的数据插入或者更新。
6. `afterSave()`：触发 `EVENT_AFTER_INSERT` 或者 `EVENT_AFTER_UPDATE` 事件。

删除数据生命周期（Deleting Data Life Cycle）

当通过 `delete()` 删除 Active Record 实例时， 会发生以下生命周期：

1. `beforeDelete()`：触发 `EVENT_BEFORE_DELETE` 事件。 如果这方法返回 `false` 或者 `yii\base\ModelEvent::$isValid` 值为 `false`，接下来的步骤都会被跳过。
2. 执行真正的数据删除。
3. `afterDelete()`：触发 `EVENT_AFTER_DELETE` 事件。

   提示： 调用以下方法则不会启动上述的任何生命周期， 因为这些方法直接操作数据库，而不是基于 Active Record 模型：

   - `yii\db\ActiveRecord::updateAll()`
   - `yii\db\ActiveRecord::deleteAll()`
   - `yii\db\ActiveRecord::updateCounters()`
   - `yii\db\ActiveRecord::updateAllCounters()`

刷新数据生命周期（Refreshing Data Life Cycle）

当通过 `refresh()` 刷新 Active Record 实例时， 如刷新成功方法返回 `true`，那么 `EVENT_AFTER_REFRESH` 事件将被触发。

6.3.8 事务操作（Working with Transactions）

Active Record 有两种方式来使用事务。

第一种方法是在事务块中显式地包含 Active Record 的各个方法调用，如下所示，

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ...其他 DB 操作...
});

// 或者

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ...other DB operations...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

> **提示：** 在上面的代码中，我们有两个catch块用于兼容 PHP 5.x 和 PHP 7.x。 `\Exception` 继承于 `\Throwable` interface[26] 由于 PHP 7.0 的改动，如果您的应用程序仅使用 PHP 7.0 及更高版本，您可以跳过 `\Exception` 部分。

第二种方法是在 `yii\db\ActiveRecord::transactions()` 方法中列出需要事务支持的 DB 操作。 例如，

```
class Customer extends ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // 上面等价于:
            // 'api' => self::OP_ALL,
        ];
    }
}
```

[26] https://www.php.net/manual/zh/class.throwable.php

`yii\db\ActiveRecord::transactions()` 方法应当返回以 场景 为键、 以需要放到事务中的 DB 操作为值的数组。以下的常量 可以表示相应的 DB 操作：

- `OP_INSERT`：插入操作用于执行 `insert()`；
- `OP_UPDATE`：更新操作用于执行 `update()`；
- `OP_DELETE`：删除操作用于执行 `delete()`。

使用 `|` 运算符连接上述常量来表明多个操作。您也可以使用 快捷常量 `OP_ALL` 来指代上述所有的三个操作。

这个事务方法的原理是：相应的事务在调用 `beforeSave()` 方法时开启， 在调用 `afterSave()` 方法时被提交。

6.3.9 乐观锁（Optimistic Locks）

乐观锁是一种防止此冲突的方法：一行数据 同时被多个用户更新。例如，同一时间内，用户 A 和用户 B 都在编辑 相同的 wiki 文章。用户 A 保存他的编辑后，用户 B 也点击“保存”按钮来 保存他的编辑。实际上，用户 B 正在处理的是过时版本的文章， 因此最好是，想办法阻止他保存文章并向他提示一些信息。

乐观锁通过使用一个字段来记录每行的版本号来解决上述问题。 当使用过时的版本号保存一行数据时，`yii\db\StaleObjectException` 异常 将被抛出，这阻止了该行的保存。乐观锁只支持更新 `yii\db\ActiveRecord::update()` 或者删除 `yii\db\ActiveRecord::delete()` 已经存在的单条数据行。

使用乐观锁的步骤，

1. 在与 Active Record 类相关联的 DB 表中创建一个列，以存储每行的版本号。 这个列应当是长整型（在 MySQL 中是 `BIGINT DEFAULT 0`）。

2. 重写 `yii\db\ActiveRecord::optimisticLock()` 方法返回这个列的命名。

3. 在你的 Model 类里实现 `OptimisticLockBehavior` 行为（注：这个行为类在 2.0.16 版本加入），以便从请求参数里自动解析这个列的值。然后从验证规则中删除 version 属性，因为 `OptimisticLockBehavior` 已经处理它了.

4. 在用于用户填写的 Web 表单中，添加一个隐藏字段（hidden field）来存储正在更新的行的当前版本号。

5. 在使用 Active Record 更新数据的控制器动作中，要捕获（try/catch）`yii\db\StaleObjectException` 异常。 实现一些业务逻辑来解决冲突（例如合并更改，提示陈旧的数据等等）。

例如，假定版本列被命名为 `version`。您可以使用下面的代码来实现乐观锁。

```php
// ------ 视图层代码 -------

use yii\helpers\Html;

// ...其他输入栏
echo Html::activeHiddenInput($model, 'version');


// ------ 控制器代码 -------

use yii\db\StaleObjectException;

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    try {
        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } catch (StaleObjectException $e) {
        // 解决冲突的代码
    }
}

// ------ Model 代码 -------

use yii\behaviors\OptimisticLockBehavior;

public function behaviors()
{
    return [
        OptimisticLockBehavior::class,
    ];
}

public function optimisticLock()
{
    return 'version';
}
```

注意： 因为 `OptimisticLockBehavior` 仅仅在保存记录的时候被确认，如果用户提交的有效版本号被直接解析：`getBodyParam()`，那么你的 Model 将扩展成这样：触发在步骤 3 中子类的行为，与此同时，调用步骤 2 中的父类的定义，这样你在把 Model 绑定到负责接收用户输入的控制器的同时，有一个专门用于内部逻辑处理的实例，或者，您可以通过配置其 `value` 的属性来实现自己的逻辑。（注：这一堆都是在解释 Behaviors 的原理）

6.3.10 使用关联数据（Working with Relational Data）

除了处理单个数据库表之外，Active Record 还可以将相关数据集中进来，使其可以通过原始数据轻松访问。 例如，客户数据与订单数据相关 因为一个客户可能已经存放了一个或多个订单。这种关系通过适当的声明， 你可以使用 `$customer->orders` 表达式访问客户的订单信息 这表达式将返回包含 `Order` Active Record 实例的客户订单信息的数组。

声明关联关系（Declaring Relations）

你必须先在 Active Record 类中定义关联关系，才能使用 Active Record 的关联数据。 简单地为每个需要定义关联关系声明一个 关联方法 即可，如下所示，

```
class Customer extends ActiveRecord
{
    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    // ...

    public function getCustomer()
    {
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}
```

上述的代码中，我们为 `Customer` 类声明了一个 `orders` 关联， 和为 `Order` 声明了一个 `customer` 关联。

每个关联方法必须这样命名：`getXyz`。然后我们通过 `xyz`（首字母小写）调用这个关联名。 请注意关联名是大小写敏感的。

当声明一个关联关系的时候，必须指定好以下的信息：

- 关联的对应关系：通过调用 `hasMany()` 或者 `hasOne()` 指定。在上面的例子中，您可以很容易看出这样的关联声明： 一个客户可以有很多订单，而每个订单只有一个客户。
- 相关联 Active Record 类名：用来指定为 `hasMany()` 或者 `hasOne()` 方法的第一个参数。 推荐的做法是调用 `Xyz::class` 来获取类名称的字符串，以便您 可以使用 IDE 的自动补全，以及让编译阶段的错误检测生效。
- 两组数据的关联列：用以指定两组数据相关的列（hasOne()/hasMany() 的第二个参数）。 数组的值填的是主数据的列（当前要声明关联的 Active Record 类为主数据）， 而数组的键要填的是相关数据的列。

一个简单的口诀，先附表的主键，后主表的主键。 正如上面的例子，customer_id 是 Order 的属性，而 id是 Customer 的属性。 （译者注：hasMany() 的第二个参数，这个数组键值顺序不要弄反了）

访问关联数据（Accessing Relational Data）

定义了关联关系后，你就可以通过关联名访问相应的关联数据了。就像 访问一个由关联方法定义的对象一样，具体概念请查看 属性。 因此，现在我们可以称它为 关联属性 了。

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
// $orders 是由 Order 类组成的数组
$orders = $customer->orders;
```

> **提示：** 当你通过 getter 方法 getXyz() 声明了一个叫 xyz 的关联属性，你就可以像 属性 那样访问 xyz。注意这个命名是区分大小写的。

如果使用 hasMany() 声明关联关系，则访问此关联属性 将返回相关的 Active Record 实例的数组； 如果使用 hasOne() 声明关联关系，访问此关联属性 将返回相关的 Active Record 实例，如果没有找到相关数据的话，则返回 null。

当你第一次访问关联属性时，将执行 SQL 语句获取数据，如 上面的例子所示。如果再次访问相同的属性，将返回先前的结果，而不会重新执行 SQL 语句。要强制重新执行 SQL 语句，你应该先 unset 这个关联属性，如：unset（$ customer-> orders）。

> **提示：** 虽然这个概念跟 这个 属性 特性很像， 但是还是有一个很重要的区别。普通对象属性的属性值与其定义的 getter 方法的类型是相同的。 而关联方法返回的是一个 yii\db\ActiveQuery 活动查询生成器的实例。只有当访问关联属性的的时候， 才会返回 yii\db\ActiveRecord Active Record 实例，或者 Active Record 实例组成的数组。
>
> ```
> $customer->orders; // 获得 `Order` 对象的数组
> $customer->getOrders(); // 返回 ActiveQuery 类的实例
> ```
>
> 这对于创建自定义查询很有用，下一节将对此进行描述。

动态关联查询（Dynamic Relational Query）

由于关联方法返回 yii\db\ActiveQuery 的实例，因此你可以在执行 DB 查询之前， 使用查询构建方法进一步构建此查询。例如，

```
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200
ORDER BY `id`
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

与访问关联属性不同，每次通过关联方法执行动态关联查询时， 都会执行 SQL 语句，即使你之前执行过相同的动态关联查询。

有时你可能需要给你的关联声明传递参数，以便您能更方便地执行 动态关系查询。例如，您可以声明一个 `bigOrders` 关联如下，

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100) // 老司机的提
    醒：$threshold 参数一定一定要给个默认值
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

然后你就可以执行以下关联查询：

```
// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200
ORDER BY `id`
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 100
ORDER BY `id`
$orders = $customer->bigOrders;
```

中间关联表（Relations via a Junction Table）

在数据库建模中，当两个关联表之间的对应关系是多对多时， 通常会引入一个连接表[27]。例如，`order` 表 和 `item` 表可以通过名为 `order_item` 的连接表相关联。一个 order 将关联多个 order items， 而一个 order item 也会关联到多个 orders。

当声明这种表关联后，您可以调用 `via()` 或 `viaTable()` 指明连接表。`via()` 和 `viaTable()` 之间的区别是 前者是根据现有的关联名称来指定连接表，而后者直接使用 连接表。例如，

```
class Order extends ActiveRecord
{
    public function getItems()
    {
```

[27] https://zh.wikipedia.org/wiki/%E5%85%B3%E8%81%94%E5%AE%9E%E4%BD%93

```
        return $this->hasMany(Item::class, ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

或者，

```
class Order extends ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::class, ['order_id' => 'id']);
    }

    public function getItems()
    {
        return $this->hasMany(Item::class, ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

使用连接表声明的关联和正常声明的关联是等同的，例如，

```
// SELECT * FROM `order` WHERE `id` = 100
$order = Order::findOne(100);

// SELECT * FROM `order_item` WHERE `order_id` = 100
// SELECT * FROM `item` WHERE `item_id` IN (...)
// 返回 Item 类组成的数组
$items = $order->items;
```

通过多个表来连接关联声明（Chaining relation definitions via multiple tables）

通过使用 `via()` 方法，它还可以通过多个表来定义关联声明。 再考虑考虑上面的例子，我们有 `Customer`，`Order` 和 `Item` 类。 我们可以添加一个关联关系到 `Customer` 类，这个关联可以列出了 `Customer`（客户） 的订单下放置的所有 `Item`（商品）， 这个关联命名为 `getPurchasedItems()`，关联声明如下代码示例所示：

```
class Customer extends ActiveRecord
{
    // ...

    public function getPurchasedItems()
    {
        // 客户的商品，将 Item 中的 'id' 列与 OrderItem 中的 'item_id' 相匹配
        return $this->hasMany(Item::class, ['id' => 'item_id'])
                    ->via('orderItems');
    }

    public function getOrderItems()
```

```
    {
        // 客户订单中的商品，将 `Order` 的 'id' 列和 OrderItem 的 'order_id'
        列相匹配
        return $this->hasMany(OrderItem::class, ['order_id' => 'id'])
                    ->via('orders');
    }

    public function getOrders()
    {
        // 见上述列子
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

延迟加载和即时加载（Lazy Loading and Eager Loading）

在 访问关联数据 中，我们解释说可以像问正常的对象属性那样 访问 Active Record 实例的关联属性。SQL 语句仅在 你第一次访问关联属性时执行。我们称这种关联数据访问方法为 延迟加载。 例如，

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$orders = $customer->orders;

// 没有 SQL 语句被执行
$orders2 = $customer->orders;
```

延迟加载使用非常方便。但是，当你需要访问相同的具有多个 Active Record 实例的关联属性时， 可能会遇到性能问题。请思考一下以下代码示例。 有多少 SQL 语句会被执行?

```
// SELECT * FROM `customer` LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM `order` WHERE `customer_id` = ...
    $orders = $customer->orders;
}
```

你瞅瞅，上面的代码会产生 101 次 SQL 查询！ 这是因为每次你访问 for 循环中不同的 `Customer` 对象的 `orders` 关联属性时，SQL 语句 都会被执行一次。

为了解决上述的性能问题，你可以使用所谓的 即时加载，如下所示，

```
// SELECT * FROM `customer` LIMIT 100;
// SELECT * FROM `orders` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
```

```
    ->all();

foreach ($customers as $customer) {
    // 没有任何的 SQL 执行
    $orders = $customer->orders;
}
```

通过调用 `yii\db\ActiveQuery::with()` 方法，你使 Active Record 在一条 SQL 语句里就返回了这 100 位客户的订单。 结果就是，你把要执行的 SQL 语句从 101 减少到 2 条！

你可以即时加载一个或多个关联。 你甚至可以即时加载 嵌套关联 。嵌套关联是一种 在相关的 Active Record 类中声明的关联。例如，`Customer` 通过 `orders` 关联属性 与 `Order` 相关联， `Order` 与 `Item` 通过 `items` 关联属性相关联。 当查询 `Customer` 时，您可以即时加载 通过嵌套关联符 `orders.items` 关联的 `items`。

以下代码展示了 `with()` 的各种用法。我们假设 `Customer` 类 有两个关联 `orders` 和 `country`，而 `Order` 类有一个关联 `items`。

```
//  即时加载 "orders" and "country"
$customers = Customer::find()->with('orders', 'country')->all();
// 等同于使用数组语法 如下
$customers = Customer::find()->with(['orders', 'country'])->all();
// 没有任何的 SQL 执行
$orders= $customers[0]->orders;
// 没有任何的 SQL 执行
$country = $customers[0]->country;

// 即时加载“订单”和嵌套关系“orders.items”
$customers = Customer::find()->with('orders.items')->all();
// 访问第一个客户的第一个订单中的商品
// 没有 SQL 查询执行
$items = $customers[0]->orders[0]->items;
```

你也可以即时加载更深的嵌套关联，比如 `a.b.c.d`。所有的父关联都会被即时加载。 那就是，当你调用 `with()` 来 with `a.b.c.d`，你将即时加载 `a`，`a.b`，`a.b.c` and `a.b.c.d`。

> **提示：** 一般来说，当即时加载 `N` 个关联，另有 `M` 个关联 通过连接表 声明，则会有 `N+M+1` 条 SQL 语句被执行。 请注意这样的的嵌套关联 `a.b.c.d` 算四个关联。

当即时加载一个关联，你可以通过匿名函数自定义相应的关联查询。 例如，

```
// 查找所有客户，并带上他们国家和活跃订单
// SELECT * FROM `customer`
// SELECT * FROM `country` WHERE `id` IN (...)
// SELECT * FROM `order` WHERE `customer_id` IN (...) AND `status` = 1
$customers = Customer::find()->with([
```

```
    'country',
    'orders' => function ($query) {
        $query->andWhere(['status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

自定义关联查询时，应该将关联名称指定为数组的键 并使用匿名函数作为相应的数组的值。匿名函数将接受一个 `$query` 参数 它用于表示这个自定义的关联执行关联查询的 `yii\db\ActiveQuery` 对象。 在上面的代码示例中，我们通过附加一个关于订单状态的附加条件来修改关联查询。

> **提示：** 如果你在即时加载的关联中调用 `select()` 方法，你要确保 在关联声明中引用的列必须被 select。否则，相应的模型（Models）可能 无法加载。例如，
>
> ```
> $orders = Order::find()->select(['id',
> 'amount'])->with('customer')->all();
> // $orders[0]->customer 会一直是 `null`。你应该这样写，以解决这个问题:
> $orders = Order::find()->select(['id', 'amount',
> 'customer_id'])->with('customer')->all();
> ```

关联关系的 JOIN 查询（Joining with Relations）

> **提示：** 这小节的内容仅仅适用于关系数据库， 比如 MySQL，PostgreSQL 等等。

到目前为止，我们所介绍的关联查询，仅仅是使用主表列 去查询主表数据。实际应用中，我们经常需要在关联表中使用这些列。例如， 我们可能要取出至少有一个活跃订单的客户。为了解决这个问题，我们可以 构建一个 join 查询，如下所示：

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id`
// WHERE `order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->select('customer.*')
    ->leftJoin('order', '`order`.`customer_id` = `customer`.`id`')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->with('orders')
    ->all();
```

> **提示：** 在构建涉及 JOIN SQL 语句的连接查询时，清除列名的歧义很重要。 通常的做法是将表名称作为前缀加到对应的列名称前。

但是，更好的方法是通过调用 `yii\db\ActiveQuery::joinWith()` 来利用已存在的关联声明：

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

两种方法都执行相同的 SQL 语句集。然而，后一种方法更干净、简洁。

默认的，`joinWith()` 会使用 `LEFT JOIN` 去连接主表和关联表。 你可以通过 `$joinType` 参数指定不同的连接类型（比如 `RIGHT JOIN`）。 如果你想要的连接类型是 `INNER JOIN`，你可以直接用 `innerJoinWith()` 方法代替。

调用 `joinWith()` 方法会默认 即时加载 相应的关联数据。 如果你不需要那些关联数据，你可以指定它的第二个参数 `$eagerLoading` 为 `false`。

> **注意：** 即使在启用即时加载的情况下使用 `joinWith()` 或 `innerJoinWith()`， 相应的关联数据也**不会**从这个 `JOIN` 查询的结果中填充。 因此，每个连接关系还有一个额外的查询，正如即时加载部分所述。

和 `with()` 一样，你可以 join 多个关联表；你可以动态的自定义 你的关联查询；你可以使用嵌套关联进行 join。你也可以将 `with()` 和 `joinWith()` 组合起来使用。例如：

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

有时，当连接两个表时，你可能需要在 JOIN 查询的 `ON` 部分中指定一些额外的条件。 这可以通过调用 `yii\db\ActiveQuery::onCondition()` 方法来完成，如下所示：

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id` AND
`order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->onCondition(['order.status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

以上查询取出 所有 客户，并为每个客户取回所有活跃订单。 请注意，这与我们之前的例子不同，后者仅取出至少有一个活跃订单的客户。

> **提示：** 当通过 `onCondition()` 修改 `yii\db\ActiveQuery` 时，如果查询涉及到 JOIN 查询，那么条件将被放在 `ON` 部分。如果查询不涉及 JOIN ，条件将自动附加到查询的 `WHERE` 部分。 因此，它可以只包含 包含了关联表的列 的条件。（译者注：意思是 onCondition() 中可以只写关联表的列，主表的列写不写都行）

关联表别名（Relation table aliases） 如前所述，当在查询中使用 JOIN 时，我们需要消除列名的歧义。因此通常为一张表定义 一个别名。可以通过以下列方式自定义关联查询来设置关联查询的别名：

```
$query->joinWith([
    'orders' => function ($q) {
        $q->from(['o' => Order::tableName()]);
    },
])
```

然而，这看起来很复杂和耦合，不管是对表名使用硬编码或是调用 `Order::tableName()`。从 2.0.7 版本起，Yii 为此提供了一个快捷方法。您现在可以定义和使用关联表的别名，如下所示：

```
// 连接 `orders` 关联表并根据 `orders.id` 排序
$query->joinWith(['orders o'])->orderBy('o.id');
```

上述语法适用于简单的关联。如果在 join 嵌套关联时， 需要用到中间表的别名，例如 `$query->joinWith(['orders.product'])`， 你需要嵌套 joinWith 调用，如下例所示：

```
$query->joinWith(['orders o' => function($q) {
        $q->joinWith('product p');
    }])
    ->where('o.amount > 100');
```

反向关联（Inverse Relations）

两个 Active Record 类之间的关联声明往往是相互关联的。例如，`Customer` 是 通过 `orders` 关联到 `Order` ， 而`Order` 通过 `customer` 又关联回到了 `Customer`。

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}
```

现在考虑下面的一段代码：

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// SELECT * FROM `customer` WHERE `id` = 123
$customer2 = $order->customer;

// 显示 "not the same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

我们原本认为 `$customer` 和 `$customer2` 是一样的，但不是！其实他们确实包含相同的 客户数据，但它们是不同的对象。 访问 `$order->customer` 时，需要执行额外的 SQL 语句， 以填充出一个新对象 `$customer2`。

为了避免上述例子中最后一个 SQL 语句被冗余执行，我们应该告诉 Yii `customer` 是 `orders` 的 反向关联，可以通过调用 `inverseOf()` 方法声明， 如下所示：

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' =>
        'id'])->inverseOf('customer');
    }
}
```

这样修改关联声明后：

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// No SQL will be executed
$customer2 = $order->customer;

// 输出 "same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

> **注意：** 反向关联不能用在有 连接表 关联声明中。 也就是说，如果一个关联关系通过 `via()` 或 `viaTable()` 声明， 你就不能再调用 `inverseOf()` 了。

6.3.11 保存关联数据（Saving Relations）

在使用关联数据时，您经常需要建立不同数据之间的关联或销毁 现有关联。这需要为定义的关联的列设置正确的值。通过使用 Active Record， 你就可以编写如下代码：

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// 为 Order 设置属性以定义与 "customer" 的关联关系
$order->customer_id = $customer->id;
$order->save();
```

Active Record 提供了 `link()` 方法，可以更好地完成此任务：

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

`link()` 方法需要指定关联名 和要建立关联的目标 Active Record 实例。该方法将修改属性的值 以连接两个 Active Record 实例，并将其保存到数据库。在上面的例子中，它将设置 `Order` 实例的 `customer_id` 属性 为 `Customer` 实例的 `id` 属性的值，然后保存 到数据库。

> **注意：** 你不能关联两个新的 Active Record 实例。

使用 `link()` 的好处在通过 junction table 定义关系时更加明显。 例如，你可以使用以下代码关联 `Order` 实例 和 `Item` 实例：

```
$order->link('items', $item);
```

上述代码会自动在 `order_item` 关联表中插入一行，以关联 order 和 item 这两个数据记录。

> **信息：** `link()` 方法在保存相应的 Active Record 实例时， 将不会执行任何数据验证。在调用此方法之前， 您应当验证所有的输入数据。

`link()` 方法的反向操作是 `unlink()` 方法， 这将可以断掉两个 Active Record 实例间的已经存在了的关联关系。例如，

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

默认情况下，`unlink()` 方法将设置指定的外键值， 以把现有的关联指定为 `null`。此外，你可以选择通过将 `$delete` 参数设置为`true` 传递给方法， 删除包含此外键值的表记录行。

当关联关系中有连接表时，调用 `unlink()` 时， 如果 `$delete` 参数是 `true` 的话，将导致 连接表中的外键或相应的行被删除。

6.3.12 跨数据库关联（Cross-Database Relations）

Active Record 允许您在不同数据库驱动的 Active Record 类之间声明关联关系。 这些数据库可以是不同的类型（例如 MySQL 和 PostgreSQL ，或是 MS SQL 和 MongoDB），它们也可以运行在 不同的服务器上。你可以使用相同的语法来执行关联查询。例如，

```
// Customer 对应的表是关系数据库中（比如 MySQL）的 "customer" 表
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

    public function getComments()
    {
        // 一个 customer 有很多条评论（comments）
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

// Comment 对应的是 MongoDB 数据库中的 "comment" 集合（译者注：MongoDB 中的集合相当于 MySQL 中的表）
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // 一条评论对应一位 customer
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}

$customers = Customer::find()->with('comments')->all();
```

本节中描述的大多数关联查询功能，你都可以抄一抄。

> **注意：** `joinWith()` 这个功能限制于某些数据库是否支持跨数据库 JOIN 查询。 因此，你再上述的代码里就不能用此方法了，因为 MongoDB 不支持 JOIN 查询。

6.3.13 自定义查询类（Customizing Query Classes）

默认情况下，`yii\db\ActiveQuery` 支持所有 Active Record 查询。要在 Active Record 类中使用自定义的查询类， 您应该重写 `yii\db\ActiveRecord::find()` 方法并返回一个你自定义查询类的实例。 例如，

```php
// file Comment.php
namespace app\models;

use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

现在，对于 Comment 类，不管你执行查询（比如 find()、findOne()），还是定义一个关联（比如 hasOne()），你都将调用到 CommentQuery 实例，而不再是 ActiveQuery 实例。

现在你可以定义 CommentQuery 类了，发挥你的奇技淫巧，以改善查询构建体验。例如，

```php
// file CommentQuery.php
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    // 默认加上一些条件（可以跳过）
    public function init()
    {
        $this->andOnCondition(['deleted' => false]);
        parent::init();
    }

    // ... 在这里加上自定义的查询方法 ...

    public function active($state = true)
    {
        return $this->andOnCondition(['active' => $state]);
    }
}
```

> **注意：** 作为 onCondition() 方法的替代方案，你应当调用 andOnCondition() 或 orOnCondition() 方法来附加新增的条件，不然在一个新定义的查询方法，已存在的条件可能会被覆盖。

然后你就可以先下面这样构建你的查询了：

```php
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

> **提示：** 在大型项目中，建议您使用自定义查询类来容纳大多数与查询相关的代码， 以使 Active Record 类保持简洁。

您还可以在 `Comment` 关联关系的定义中或在执行关联查询时，使用刚刚新建查询构建方法：

```
class Customer extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::class, ['customer_id' =>
        'id'])->active();
    }
}

$customers = Customer::find()->joinWith('activeComments')->all();

// 或者这样
class Customer extends \yii\db\ActiveRecord
{
    public function getComments()
    {
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

$customers = Customer::find()->joinWith([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

> **提示：** 在 Yii 1.1 中，有个概念叫做 命名范围。命名范围在 Yii 2.0 中不再支持， 你依然可以使用自定义查询类、查询方法来达到一样的效果。

6.3.14 选择额外的字段（Selecting extra fields）

当 Active Record 实例从查询结果中填充时，从数据结果集中， 其属性的值将被相应的列填充。

你可以从查询中获取其他列或值，并将其存储在 Active Record 活动记录中。 例如，假设我们有一个名为 `room` 的表，其中包含有关酒店可用房间的信息。 每个房间使用字段 `length`，`width`，`height` 存储有关其空间大小的信息。 想象一下，我们需要检索出所有可用房间的列表，并按照体积大小倒序排列。 你不可能使用 PHP 来计算体积，但是，由于我们需要按照它的值对这些记录进行排序，你依然需要 `volume` （体积） 来显示在这个列表中。 为了达到这个目标，你需要在你的 `Room` 活动记录类中声明一个额外的字段，它将存储 `volume` 的值：

```
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

然后，你需要撰写一个查询，它可以计算房间的大小并执行排序：

```
$rooms = Room::find()
    ->select([
        '{{room}}.*', // select all columns
        '([[length]] * [[width]] * [[height]]) AS volume', // 计算体积
    ])
    ->orderBy('volume DESC') // 使用排序
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // 包含了由 SQL 计算出的值
}
```

额外字段的特性对于聚合查询非常有用。 假设您需要显示一系列客户的订单数量。 首先，您需要使用 `orders` 关系声明一个 `Customer` 类，并指定额外字段来存储 count 结果：

```
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

然后你可以编写一个查询来 JOIN 订单表，并计算订单的总数：

```
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // select customer 表所有的字段
        'COUNT({{order}}.id) AS ordersCount' // 计算订单总数
    ])
    ->joinWith('orders') // 连接表
    ->groupBy('{{customer}}.id') // 分组查询，以确保聚合函数生效
    ->all();
```

使用此方法的一个缺点是，如果数据不是从 SQL 查询上加载的，它必须再单独计算一遍。 因此，如果你通过常规查询获取个别的数据记录时，它没有额外的 select 语句，那么它 将无法返回额外字段的实际值。新保存的记录一样会发生这种情。

```
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // 为 `null`, 因为它没有被声明（赋值）
```

通过 `__get()` 和 `__set()` 魔术方法 我们可以将属性赋予行为特性：

```
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }

    public function getVolume()
    {
        if (empty($this->length) || empty($this->width) ||
        empty($this->height)) {
            return null;
        }

        if ($this->_volume === null) {
            $this->setVolume(
                $this->length * $this->width * $this->height
            );
        }

        return $this->_volume;
    }

    // ...
}
```

当 select 查询不提供 volume 体积时，这模型将能够自动计算体积的值出来， 当访问模型的属性的时候。

当定义关联关系的时候，你也可以计算聚合字段：

```
class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }

    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
```

```
            return null; // 这样可以避免调用空主键进行查询
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount($this->getOrders()->count()); // 根据关联
            关系按需计算聚合字段
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

使用此代码，如果 'select' 语句中存在 'ordersCount' - 它会从查询结果集获取数据填充 `Customer::ordersCount` 属性， 否则它会在被访问的时候，使用 `Customer::orders` 关联按需计算。

这种方法也适用于创建一些关联数据的快捷访问方式，特别是对于聚合。 例如:

```
class Customer extends \yii\db\ActiveRecord
{
    /**
     * 为聚合数据定义一个只读的虚拟属性
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; //  这样可以避免调用空主键进行查询
        }

        return empty($this->ordersAggregation) ? 0 :
        $this->ordersAggregation[0]['counted'];
    }

    /**
     * 声明一个常规的 'orders' 关联
     */
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }

    /**
     * 基于 'orders' 关联，声明一个用于查询聚合的新关联
     */
    public function getOrdersAggregation()
    {
```

```
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // 输出关联的聚合数据，而不需要额外的查询，因
    为我们用了即时加载
}

$customer = Customer::findOne($pk);
$customer->ordersCount; // 从延迟加载的关联中，输出聚合数据
```

6.4 数据库迁移

在开发和维护一个数据库驱动的应用程序时， 数据库的结构会像代码一样不断演变。 例如，在开发应用程序的过程中，会增加一张新表且必须得加进来； 在应用程序被部署到生产环境后，需要建立一个索引来提高查询的性能等等。 因为一个数据库结构发生改变的时候源代码也经常会需要做出改变， Yii 提供了一个 数据库迁移 功能，该功能可以记录数据库的变化，以便使数据库和源代码一起受版本控制。

如下的步骤向我们展示了数据库迁移工具是如何为开发团队所使用的：

1. Tim 创建了一个新的迁移对象（例如，创建一张新表，改变字段的定义等）。
2. Tim 将这个新的迁移对象提交到代码管理系统（例如，Git，Mercurial）。
3. Doug 从代码管理系统当中更新版本并获取到这个新的迁移对象。
4. Doug 把这个迁移对象提交到本地的开发数据库当中， 这样一来，Doug 同步了 Tim 所做的修改。

如下的步骤向我们展示了如何发布一个附带数据库迁移的新版本到生产环境当中：

1. Scott 为一个包含数据库迁移的项目版本创建了一个发布标签。
2. Scott 把发布标签的源代码更新到生产环境的服务器上。
3. Scott 把所有的增量数据库迁移提交到生产环境数据库当中。

Yii 提供了一整套的迁移命令行工具，通过这些工具你可以：

- 创建新的迁移；

- 提交迁移;
- 恢复迁移;
- 重新提交迁移;
- 展示迁移历史和状态。

所有的这些工具都可以通过 `yii migrate` 命令来进行操作。 在这一章节，我们将详细的介绍如何使用这些工具来完成各种各样的任务。 你也可以通过 `yii help migrate` 命令来获取每一种工具的具体使用方法。

> **提示**： 迁移不仅仅只作用于数据库表， 它同样会调整现有的数据来适应新的表单、创建 RBAC 分层、又或者是清除缓存。

> **注意**： 当你使用迁移来操作数据时，你会发现使用 活动记录类 很有帮助，因为一些逻辑已经在那里实现了。 然而别忘了，相比于天生保持恒定不变的迁移类中的代码，程序逻辑是随时变化的。 所以当你在迁移中使用活动记录类时，活动记录层中逻辑的变化可能会意外打断现有的迁移。 基于这个原因， 迁移代码应该保持独立于其他程序逻辑，比如活动记录类。

6.4.1 创建迁移

使用如下命令来创建一个新的迁移:

```
yii migrate/create <name>
```

必填参数 `name` 的作用是对新的迁移做一个简要的描述。 例如，如果这个迁移是用来创建一个叫做 *news* 的表， 那么你可以使用 `create_news_table` 这个名称并运行如下命令:

```
yii migrate/create create_news_table
```

> **注意**： 因为 `name` 参数会被用来生成迁移的类名的一部分， 所以该参数应当只包含字母、数字和下划线。

如上命令将会在 `@app/migrations` 目录下创建一个新的名为 `m150101_185401_create_news_table.php` 的 PHP 类文件。 该文件包含如下的代码，它们用来声明一个迁移类 `m150101_185401_create_news_table`， 并附有代码框架:

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }
```

```
    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

每个数据库迁移都会被定义为一个继承自 `yii\db\Migration` 的 PHP 类。类的名称按照 `m<YYMMDD_HHMMSS>_<Name>` 的格式自动生成，其中

- `<YYMMDD_HHMMSS>` 指执行创建迁移命令的 UTC 时间。
- `<Name>` 和你执行命令时所带的 `name` 参数值相同。

在迁移类当中，你应当在 `up()` 方法中编写改变数据库结构的代码。 你可能还需要在 `down()` 方法中编写代码来恢复由 `up()` 方法所做的改变。 当你通过 migration 升级数据库时， `up()` 方法将会被调用，反之， `down()` 将会被调用。 如下代码展示了如何通过迁移类来创建一张 `news` 表:

```
use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends \yii\db\Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }

}
```

> **注意：** 并不是所有迁移都是可恢复的。例如，如果 `up()` 方法删除了表中的一行数据， 这将无法通过 `down()` 方法来恢复这条数

据。有时候，你也许只是懒得去执行 `down()` 方法了， 因为它在恢复数据库迁移方面并不是那么的通用。在这种情况下， 你应当在 `down()` 方法中返回 `false` 来表明这个 migration 是无法恢复的。

migration 的基类 `yii\db\Migration` 通过 `db` 属性来连接数据库。 你可以通过 操作数据库模式（schema） 章节中所描述的那些方法来操作数据库模式（schema）。

当你创建一张表或者一个字段的时候，你应该使用 抽象类型 而不是 实体类型， 这样一来你的迁移对象就可以从特定的 DBMS 当中抽离出来。 `yii\db\Schema` 类定义了一整套可用的抽象类型常量。这些常量的格式为 `TYPE_<Name>`。 例如，`TYPE_PK` 指代自增主键类型；`TYPE_STRING` 指代字符串类型。 当迁移对象被提交到某个特定的数据库的时候，这些抽象类型将会被转换成相对应的实体类型。 以 MySQL 为例，`TYPE_PK` 将会变成 `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`， 而 `TYPE_STRING` 则变成 `varchar(255)`。

在使用抽象类型的时候，你可以添加额外的约束条件。在上面的例子当中， `NOT NULL` 被添加到 `Schema::TYPE_STRING` 当中来指定该字段不能为空。

> **提示：** 抽象类型和实体类型之间的映射关系是由每个具体的 `QueryBuilder` 类当中的 `$typeMap` 属性所指定的。

从 2.0.6 版本开始，你可以使用能提供更便捷的方法定义字段模式（schema）的新模式（schema）构建器。 这样上面的迁移就可以写成这样：

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

所有用来定义字段类型的方法都可以到 `yii\db\SchemaBuilderTrait` 的 API 文档中查到。

6.4.2 生成迁移

从 2.0.7 版本开始，迁移终端提供了一种创建迁移的便捷方法。

如果迁移对象的名称遵循某种特定的格式，比如 `create_xxx` 或者 `drop_xxx`，那么生成的迁移代码中将包含创建/删除表的额外代码。 在下文中将描述该特型的所有变型。

创建表

```
yii migrate/create create_post
```

生成

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

利用 `--fields` 选项指定字段参数，可以立即创建字段。

```
yii migrate/create create_post --fields="title:string,body:text"
```

生成

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
```

```
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

你可以指定更多的字段参数。

```
yii migrate/create create_post
--fields="title:string(12):notNull:unique,body:text"
```

生成

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

> **注意：** 主键会被自动添加同时默认名称为 `id`。 如果你想使用其他名称可以使用 `--fields="name:primaryKey"` 来指定名称。

外键 从 2.0.8 版本开始，生成器通过使用 `foreignKey` 关键字支持外键。

```
yii migrate/create create_post
--fields="author_id:integer:notNull:foreignKey(user),category_id:integer:defaultValue(1):foreignK
```

生成

```
/**
 * Handles the creation for table `post`.
 * Has foreign keys to the tables:
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // creates index for column `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // add foreign key for table `user`
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
            'id',
            'CASCADE'
        );

        // creates index for column `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );

        // add foreign key for table `category`
```

```
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        // drops foreign key for table `user`
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // drops index for column `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
        );

        // drops foreign key for table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // drops index for column `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
        );

        $this->dropTable('post');
    }
}
```

`foreignKey` 关键字在字段描述中的位置不会影响生成的代码。 那意味着:

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

以上都会生成相同的代码。

`foreignKey` 关键字可以在圆括号中间接收一个参数， 这个参数将会成为生成外键所需的关联表的名称。 如果不传入任何参数，那么表名将会根据字段名来生成。

在上面的例子中， `author_id:integer:notNull:foreignKey(user)` 会生成一

个 带有关联 user 表的外键，名称为 author_id 的字段， category_id:integer:defaultValue(1):forei
会生成一个 带有关联 category 表的外键，名称为 category_id 的字段。

从 2.0.11 版本开始，foreignKey 接收第二个参数，跟第一个参数用空格分开。 这个参数表示生成的外键所关联字段的名称。 如果不传入第二个参数，字段名将从表模式（schema）中获得。 如果模式（schema）不存在，或者未设置主键，又或者是联合主键，字段将使用 id 作为默认名称。

删除表

```
yii migrate/create drop_post
--fields="title:string(12):notNull:unique,body:text"
```

生成

```
class m150811_220037_drop_post extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

添加字段

如果迁移的名称遵循 add_xxx_to_yyy 这样的格式， 生成的类文件将会包含必要的 addColumn 和 dropColumn。

添加字段:

```
yii migrate/create add_position_to_post --fields="position:integer"
```

生成

```
class m150811_220037_add_position_to_post extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

你可以像如下这样指定多个字段:

```
yii migrate/create add_xxx_column_yyy_column_to_zzz_table
--fields="xxx:integer,yyy:text"
```

删除字段

如果迁移的名称遵循 `drop_xxx_from_yyy` 这样的格式， 生成的类文件将会包含必要的 `addColumn` 和 `dropColumn`。

```
yii migrate/create drop_position_from_post --fields="position:integer"
```

生成

```
class m150811_220037_drop_position_from_post extends Migration
{
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

添加连接表

如果迁移的名称遵循 `create_junction_xxx_and_yyy` 这样的格式， 创建连接表的必要代码将会被生成。

```
yii migrate/create create_junction_post_and_tag
--fields="created_at:dateTime"
```

生成

```
/**
 * Handles the creation for table `post_tag`.
 * Has foreign keys to the tables:
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_post_and_tag extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post_tag', [
```

```
        'post_id' => $this->integer(),
        'tag_id' => $this->integer(),
        'created_at' => $this->dateTime(),
        'PRIMARY KEY(post_id, tag_id)',
    ]);

    // creates index for column `post_id`
    $this->createIndex(
        'idx-post_tag-post_id',
        'post_tag',
        'post_id'
    );

    // add foreign key for table `post`
    $this->addForeignKey(
        'fk-post_tag-post_id',
        'post_tag',
        'post_id',
        'post',
        'id',
        'CASCADE'
    );

    // creates index for column `tag_id`
    $this->createIndex(
        'idx-post_tag-tag_id',
        'post_tag',
        'tag_id'
    );

    // add foreign key for table `tag`
    $this->addForeignKey(
        'fk-post_tag-tag_id',
        'post_tag',
        'tag_id',
        'tag',
        'id',
        'CASCADE'
    );
}

/**
 * @inheritdoc
 */
public function down()
{
    // drops foreign key for table `post`
    $this->dropForeignKey(
        'fk-post_tag-post_id',
        'post_tag'
    );

    // drops index for column `post_id`
    $this->dropIndex(
```

```
            'idx-post_tag-post_id',
            'post_tag'
        );

        // drops foreign key for table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // drops index for column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );

        $this->dropTable('post_tag');
    }
}
```

从 2.0.11 版本开始，连接表的外键字段名将从表模式（schema）中获得。如果模式（schema）不存在，或者未设置主键，又或者是联合主键，字段将使用 `id` 作为默认名称。

事务迁移

当需要实现复杂的数据库迁移的时候，确定每一个迁移的执行是否成功或失败就变得相当重要了， 因为这将影响到数据库的完整性和一致性。为了达到这个目标，我们建议你把每个迁移里面的 数据库操作都封装到一个 transaction 里面。

实现事务迁移的一个更为简便的方法是把迁移的代码都放到 `safeUp()` 和 `safeDown()` 方法里面。 它们与 `up()` 和 `down()` 的不同点就在于它们是被隐式的封装到事务当中的。 如此一来，只要这些方法里面的任何一个操作失败了，那么所有之前的操作都会被自动的回滚。

在如下的例子当中，除了创建 `news` 表以外，我们还插入了一行初始化数据到表里面。

```
use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),,
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
```

```
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

需要注意的是，当你在 `safeUp()` 当中执行多个数据库操作的时候，你应该在 `safeDown()` 方法当中反转它们的执行顺序。 在上面的例子当中，我们在 `safeUp()` 方法当中首先创建了一张表，然后插入了一条数据；而在 `safeDown()` 方法当中， 我们首先删除那一行数据，然后才删除那张表。

> **注意：** 并不是所有的数据库都支持事务。有些数据库查询也是不能被放到事务里面的。 在 implicit commit[28] 章节当中有相关的例子可以参考。 如果遇到这种情况的话，那么你应该使用 `up()` 和 `down()` 方法进行替代。

访问数据库的方法

迁移的基类 `yii\db\Migration` 提供了一整套访问和操作数据库的方法。你可能会发现这些方法的命名和 `yii\db\Command` 类提供的 DAO 方法 很类似。 例如，`yii\db\Migration::createTable()` 方法可以创建一张新的表， 这和 `yii\db\Command::createTable()` 的功能是一模一样的。

使用 `yii\db\Migration` 所提供的方法的好处在于你不需要再显式的创建 `yii\db\Command` 实例， 而且在执行每个方法的时候都会显示一些有用的信息来告诉我们数据库操作是不是都已经完成， 还有它们完成这些操作花了多长时间等等。

如下是所有这些数据库访问方法的列表：

- `execute()`：执行一条 SQL 语句
- `insert()`：插入单行数据
- `batchInsert()`：插入多行数据
- `update()`：更新数据
- `delete()`：删除数据
- `createTable()`：创建表
- `renameTable()`：重命名表名
- `dropTable()`：删除一张表
- `truncateTable()`：清空表中的所有数据
- `addColumn()`：加一个字段
- `renameColumn()`：重命名字段名称
- `dropColumn()`：删除一个字段

[28] https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html

- alterColumn()：修改字段
- addPrimaryKey()：添加一个主键
- dropPrimaryKey()：删除一个主键
- addForeignKey()：添加一个外键
- dropForeignKey()：删除一个外键
- createIndex()：创建一个索引
- dropIndex()：删除一个索引
- addCommentOnColumn()：添加字段的注释
- dropCommentFromColumn()：删除字段的注释
- addCommentOnTable()：添加表的注释
- dropCommentFromTable()：删除表的注释

> **提示：** yii\db\Migration 并没有提供数据库的查询方法。 这是因为通常你是不需要去数据库把数据一行一行查出来再显示出来的。 另外一个原因是你完全可以使用强大的 Query Builder 查询构建器 来构建和查询。
> 你可以像这样在迁移中使用查询构建器：

```
// 更新所有用户的 status 字段
foreach((new Query)->from('users')->each() as $user) {
    $this->update('users', ['status' => 1], ['id' => $user['id']]);
}
```

6.4.3 提交迁移

为了将数据库升级到最新的结构，你应该使用如下命令来提交所有新的迁移：

```
yii migrate
```

这条命令会列出迄今为止所有未提交的迁移。如果你确定你需要提交这些迁移， 它将会按照类名当中的时间戳的顺序，一个接着一个的运行每个新的迁移类里面的 up() 或者是 safeUp() 方法。 如果其中任意一个迁移提交失败了， 那么这条命令将会退出并停止剩下的那些还未执行的迁移。

> **提示：** 如果你的服务器没有命令行， 你可以尝试 web shell[29] 这个扩展。

对于每一个成功提交的迁移，这条命令都会在一个叫做 migration 的数据库表中插入一条包含应用程序成功提交迁移的记录， 该记录将帮助迁移工具判断哪些迁移已经提交，哪些还没有提交。

> **信息：** 迁移工具将会自动在数据库当中创建 migration 表， 该数据库是在该命令的 db 选项当中指定的。 默认情况下，是由 db application component 指定的。

[29] https://github.com/samdark/yii2-webshell

有时，你可能只需要提交一个或者少数的几个迁移， 你可以使用该命令指定需要执行的条数，而不是执行所有的可用迁移。 例如，如下命令将会尝试提交前三个可用的迁移:

```
yii migrate 3
```

你也可以指定一个特定的迁移，按照如下格式使用 `migrate/to` 命令 来指定数据库应该提交哪一个迁移:

```
yii migrate/to 150101_185401                      # using timestamp to
specify the migration 使用时间戳来指定迁移
yii migrate/to "2015-01-01 18:54:01"              # using a string that can
be parsed by strtotime() 使用一个可以被 strtotime() 解析的字符串
yii migrate/to m150101_185401_create_news_table   # using full name 使用全名
yii migrate/to 1392853618                         # using UNIX timestamp 使
用 UNIX 时间戳
```

如果在指定要提交的迁移前面还有未提交的迁移，那么在执行这个被指定的迁移之前， 这些还未提交的迁移会先被提交。

如果被指定提交的迁移在之前已经被提交过，那么在其之后的那些迁移将会被还原。

6.4.4 还原迁移

你可以使用如下命令来还原其中一个或多个意见被提交过的迁移:

```
yii migrate/down     # revert the most recently applied migration 还原最近一
次提交的迁移
yii migrate/down 3   # revert the most 3 recently applied migrations 还原最
近三次提交的迁移
```

> **注意**： 并不是所有的迁移都能被还原。 尝试还原这类迁移将可能导致报错甚至是终止所有的还原进程。

6.4.5 重做迁移

重做迁移的意思是先还原指定的迁移，然后再次提交。 如下所示:

```
yii migrate/redo        # 重做最近一次提交的迁移
yii migrate/redo 3      # 重做最近三次提交的迁移
```

> **注意**： 如果一个迁移是不能被还原的，那么你将无法对它进行重做。

6.4.6 刷新迁移

从 2.0.13 版本开始，你可以从数据库中删除所有的表和外键，从头开始重新提交所有迁移。

```
yii migrate/fresh       # 清空数据库并从头开始应用所有迁移。
```

6.4.7 列出迁移

你可以使用如下命令列出那些提交了的或者是还未提交的迁移:

```
yii migrate/history     # 显示最近10次提交的迁移
yii migrate/history 5   # 显示最近5次提交的迁移
yii migrate/history all # 显示所有已经提交过的迁移

yii migrate/new         # 显示前10个还未提交的迁移
yii migrate/new 5       # 显示前5个还未提交的迁移
yii migrate/new all     # 显示所有还未提交的迁移
```

6.4.8 修改迁移历史

有时候你也许需要简单的标记一下你的数据库已经升级到一个特定的迁移， 而不是实际提交或者是还原迁移。 这个经常会发生在你手动的改变数据库的一个特定状态，而又不想相应的迁移被重复提交。 那么你可以使用如下命令来达到目的:

```
yii migrate/mark 150101_185401                      # 使用时间戳来指定迁移
yii migrate/mark "2015-01-01 18:54:01"              # 使用一个可以被
strtotime() 解析的字符串
yii migrate/mark m150101_185401_create_news_table   # 使用全名
yii migrate/mark 1392853618                         # 使用 UNIX 时间戳
```

该命令将会添加或者删除 `migration` 表当中的某几行数据来表明数据库已经提交到了指定的某个迁移上。 执行这条命令期间不会有任何的迁移会被提交或还原。

6.4.9 自定义迁移

有很多方法可以自定义迁移命令。

使用命令行选项

迁移命令附带了几个命令行选项，可以用来自定义它的行为:

- `interactive`: boolean (默认值为 true)，指定是否以交互模式来运行迁移。 当被设置为 true 时，在命令执行某些操作前，会提示用户。如果你希望在后台执行该命令， 那么你应该把它设置成 false。
- `migrationPath`: string|array (默认值为 `@app/migrations`)，指定存放所有迁移类文件的目录。该选项可以是一个目录的路径， 也可以是 路径别名。需要注意的是指定的目录必选存在， 否则将会触发一个错误。 从 2.0.12 版本开始，可以用一个数组来指定从多个来源读取迁移类文件。
- `migrationTable`: string (默认值为 `migration`)，指定用于存储迁移历史信息的数据库表名称。 如果这张表不存在，那么迁移命令将自动创建这张表。当然你也可以使用这样的字段结构： `version varchar(255) primary key, apply_time integer` 来手动创建这张表。

- `db`: string (默认值为 `db`)，指定数据库 application component 的 ID。它指的是将会被该命令迁移的数据库。
- `templateFile`: string (默认值为 `@yii/views/migration.php`)， 指定生产迁移框架代码类文件的模版文件路径。 该选项即可以使用文件路径来指定，也可以使用路径 别名 来指定。 该模版文件是一个可以使用预定义变量 `$className` 来获取迁移类名称的 PHP 脚本。
- `generatorTemplateFiles`: array (defaults to '[

  ```
  'create_table' => '@yii/views/createTableMigration.php',
  'drop_table' => '@yii/views/dropTableMigration.php',
  'add_column' => '@yii/views/addColumnMigration.php',
  'drop_column' => '@yii/views/dropColumnMigration.php',
  'create_junction' => '@yii/views/createJunctionMigration.php'
  ```

  ]')，指定生成迁移代码的模版文件，查看 "Generating Migrations" 了解更多细节。
- `fields`: 由用来创建迁移代码的多个字段定义字符串所组成的数组。默认是 `[]`。 字段定义的格式是 `COLUMN_NAME:COLUMN_TYPE:COLUMN_DECORATOR`。例如，`--fields=name:string(12):notNull` 会创建 一个长度为 12 的，非空的，字符串类型的字段。

如下例子向我们展示了如何使用这些选项：

例如，如果我们需要迁移一个 `forum` 模块， 而该迁移文件放在该模块下的 `migrations` 目录当中， 那么我们可以使用如下命令：

```
# 在 forum 模块中以非交互模式进行迁移
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

全局配置命令

为了避免运行迁移命令的时候每次都要重复的输入一些同样的参数， 你可以选择在应用程序配置当中进行全局配置，一劳永逸：

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

如上所示配置，在每次运行迁移命令的时候， `backend_migration` 表将会被用来记录迁移历史。 你再也不需要通过 `migrationTable` 命令行参数来指定这张历史纪录表了。

使用命名空间的迁移

自从 2.0.10 版本开始，你可以为迁移类使用命名空间。 你可以通过 `migrationNamespaces` 来指定迁移会用到的命名空间。 使用命名空间将允许你利用多个源的位置进行迁移。例如：

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => null, // disable non-namespaced migrations if
            app\migrations is listed below
            'migrationNamespaces' => [
                'app\migrations', // Common migrations for the whole
                application
                'module\migrations', // Migrations for the specific
                project's module
                'some\extension\migrations', // Migrations for the specific
                extension
            ],
        ],
    ],
];
```

> **注意：** 从不同命名空间提交的迁移会创建出一份**单一的**迁移历史， 比如你不能只从某个特定的命名空间去提交或者还原迁移。

当你正在操作使用命名空间的迁移时：比如创建迁移，还原迁移等，你应当在迁移名称前指明完整的命名空间。 要注意反斜线(\)在 shell 中会被当成特殊字符，你应该对反斜线进行编码， 避免 shell 报错或者产生不正确的结果。例如：

```
yii migrate/create 'app\\migrations\\createUserTable'
```

> **注意：** 通过 `migrationPath` 声明的迁移不能包含命名空间， 使用命名空间的迁移只能通过 `yii\console\controllers\MigrateController::$migrationNamespaces` 的属性来提交。

从版本 2.0.12 开始，`migrationPath` 属性也接收一个数组作为参数, 这个数组指明了没有使用命名空间的多个迁移的目录。 这个参数主要用于已经从多个位置进行了迁移的现有项目。 这些迁移主要来自外部资源，比如其他开发人员开发的 Yii 扩展， 当使用新方法时，这些迁移很难改成使用命名空间。

分离的迁移

有时候我们并不想整个项目所有的迁移都记录到同一份迁移历史中。 例如：你可能安装了 'blog' 扩展，它有自己独立的功能和迁移， 不会影响到用于项目主要功能的其他扩展。

如果你想完全分开提交和追踪多个迁移， 你可以同时配置使用不同命名空间和历史记录表的多条迁移命令：

```
return [
    'controllerMap' => [
```

```
        // Common migrations for the whole application
        'migrate-app' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['app\migrations'],
            'migrationTable' => 'migration_app',
            'migrationPath' => null,
        ],
        // Migrations for the specific project's module
        'migrate-module' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['module\migrations'],
            'migrationTable' => 'migration_module',
            'migrationPath' => null,
        ],
        // Migrations for the specific extension
        'migrate-rbac' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => '@yii/rbac/migrations',
            'migrationTable' => 'migration_rbac',
        ],
    ],
];
```

请注意，要同步数据库，您现在需要运行多个命令而不是一个：

```
yii migrate-app
yii migrate-module
yii migrate-rbac
```

6.4.10 迁移多个数据库

默认情况下，迁移将会提交到由 `db` application component 所定义的同一个数据库当中。 如果你需要提交到不同的数据库，你可以像下面那样指定 `db` 命令行选项，

```
yii migrate --db=db2
```

上面的命令将会把迁移提交到 `db2` 数据库当中。

有些时候你需要提交 一些 迁移到一个数据库，而另外一些则提交到另一个数据库。 为了达到这个目的，你应该在实现一个迁移类的时候指定需要用到的数据库组件的 ID ， 如下所示：

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
```

```
        parent::init();
    }
}
```

即使你使用 db 命令行选项指定了另外一个不同的数据库，上面的迁移还是会被提交到 db2 当中。 需要注意的是这个时候迁移的历史信息依然会被记录到 db 命令行选项所指定的数据库当中。

如果有多个迁移都使用到了同一个数据库，那么建议你创建一个迁移的基类，里面包含上述的 init() 代码。 然后每个迁移类都继承这个基类就可以了。

> **提示：** 除了在 db 参数当中进行设置以外， 你还可以通过在迁移类中创建新的数据库连接来操作不同的数据库。 然后通过这些连接再使用 DAO 方法 来操作不同的数据库。

另外一个可以让你迁移多个数据库的策略是把迁移存放到不同的目录下，然后你可以通过如下命令分别对不同的数据库进行迁移:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

第一条命令将会把 @app/migrations/db1 目录下的迁移提交到 db1 数据库当中， 第二条命令则会把 @app/migrations/db2 下的迁移提交到 db2 数据库当中，以此类推。

Chapter 7

接收用户数据（Getting Data from Users）

7.1 创建表单

7.1.1 基于活动记录（ActiveRecord）的表单：ActiveForm

在yii中使用表单的主要方式是通过 `yii\widgets\ActiveForm`。 当某个表单是基于一个模型时，应该首选这种方式。 此外，在 `yii\helpers\Html` 中有很多实用的方法为表单添加按钮和帮助文档。

在客户端显示的表单，大多数情况下都有一个相应的模型， 用来在服务器上验证其输入的数据（可在输入验证一节获取关于验证的细节）。 当创建一个基于模型的表单时，第一步是定义模型本身。该模型可以是一个基于活动记录的类， 表示数据库中的数据，也可以是一个基于通用模型的类（继承自`yii\base\Model`）， 来获取任意的输入数据，如登录表单。

> **提示：** 如果一个表单的输入域与数据库的字段不匹配，或者它存在只适用于它的特殊的格式或者方法， 则最好为它创建一个单独的继承自 `yii\base\Model` 的模型。

在接下来的例子中，我们展示了通用模型如何用于登录表单：

```
<?php

class LoginForm extends \yii\base\Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // 在这里定义验证规则
        ];
    }
}
```

在控制器中，我们将传递一个模型的实例到视图，其中 ActiveForm 小部件用来显示表单：

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

$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() ?>
```

用 和 包裹

在上面的代码中，ActiveForm::begin() 不仅创建了一个表单实例，同时也标志着表单的开始。 放在 ActiveForm::begin() 与 ActiveForm::end() 之间的所有内容都被包裹在 HTML 的 <form> 标签中。 与任何小部件一样，你可以指定一些选项，通过传递数组到 begin 方法中来配置该小部件。在这种情况下， 一个额外的 CSS 类和 ID 会在 <form> 标签中使用。要查看所有可用的选项， 请参阅 API 文档的 yii\widgets\ActiveForm。

ActiveField

为了在表单中创建表单元素与元素的标签，以及任何适用的 JavaScript 验证，ActiveForm::field() 方法在调用时，会返回一个 yii\widgets\ActiveField 的实例。 直接输出该方法时，结果是一个普通的（文本）输入。要自定义输出，可以附加上 ActiveField 的其它方法来一起调用：

```
// 一个密码输入框
<?= $form->field($model, 'password')->passwordInput() ?>
// 增加一个提示标签
<?= $form->field($model, 'username')->textInput()->hint('Please enter your
name')->label('Name') ?>
// 创建一个 HTML5 邮箱输入框
<?= $form->field($model, 'email')->input('email') ?>
```

它会通过在 template 中定义的表单字段来创建 <label>，<input> 以及其它的标签。 input 输入框的 name 属性会自动地根据 form name 以及属性名来创建。 例如，对于在上面的例子中 username 输入字段的 name 属性将是 LoginForm[username]。 这种命名规则使所有属性的数组的登录表单在服务器端的 $_POST['LoginForm'] 数组中是可用的。

> **提示：** 如果在表单中只有一个模型并且想要简化输入名称，则可以通过覆盖模型的 `formName()` 方法来返回一个空字符串。这对于 GridView 中使用的过滤器模型来创建更好的URL很有用。

指定模型的属性可以以更复杂的方式来完成。例如，当上传时，多个文件或选择多个项目的属性，可能需要一个数组值，你可以通过附加 `[]` 来 指定它的属性名称：

```
// 允许多个文件被上传：
echo $form->field($model,
'uploadFile[]')->fileInput(['multiple'=>'multiple']);

// 允许进行选择多个项目：
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' =>
'Item B', 'c' => 'Item C']);
```

命名表单元素，如提交按钮时要小心。在 jQuery 文档[1] 中有一些保留的名称，可能会导致冲突：

> 表单和它们的子元素不应该使用与表单的属性冲突的 input name 或 id， 例如 `submit`，`length`，或者 `method`。 要检查你的标签是否存在这些问题，一个完整的规则列表详见 DOMLint[2]。

额外的 HTML 标签可以使用纯 HTML 或者 `Html`-辅助类中的方法来添加到表单中，就如上面例子中的 `Html::submitButton()`。

> **提示：** 如果你正在你的应用程序中使用 Twitter Bootstrap CSS 你可以使用`yii\bootstrap\ActiveForm` 来代替 `yii\widgets\ActiveForm`。 前者继承自后者并在生成表单字段时使用 Bootstrap 特有的样式。
>
> **提示：** 为了设计带星号的表单字段，你可以使用下面的 CSS：
>
> ```
> div.required label:after {
>  content: " *";
>  color: red;
> }
> ```

7.1.2 创建下拉列表

有三种类型的列表：

- 下拉列表
- 单选列表
- 复选框列表

[1] https://api.jquery.com/submit/
[2] https://kangax.github.io/domlint/

要创建一个列表，你必须准备这些项目。可以手动完成：

```
$items = [
    1 => 'item 1',
    2 => 'item 2'
]
```

或从 DB 中检索：

```
$items = Category::find()
        ->select(['label'])
        ->indexBy('id')
        ->column();
```

这些 `$items` 必须由不同的列表小部件处理。 表单域（和当前活动项目）的值将由 `$model` 属性的当前值自动设置。

创建一个下拉列表 我们可以使用 ActiveField `yii\widgets\ActiveField::dropDownList()` 方法创建一个下拉列表：

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

echo $form->field($model, 'category')->dropdownList([
        1 => 'item 1',
        2 => 'item 2'
    ],
    ['prompt'=>'Select Category']
);
```

创建一个单选列表 我们可以使用 ActiveField `yii\widgets\ActiveField::radioList()` 方法创建一个单选列表：

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

echo $form->field($model, 'category')->radioList([
    1 => 'radio 1',
    2 => 'radio 2'
]);
```

创建一个复选框列表 我们可以使用 ActiveField `yii\widgets\ActiveField::checkboxList()` 方法创建一个复选框列表：

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

echo $form->field($model, 'category')->checkboxList([
    1 => 'checkbox 1',
    2 => 'checkbox 2'
]);
```

7.1.3 与 Pjax 一起工作

Pjax 小部件允许您更新某个部分 而不是重新加载整个页面。 您可以使用它来仅更新表单并在提交后更换其内容。

你可以配置 $formSelector 来指定表单提交可能会触发 pjax。如果没有设置，所有封装 Pjax 内容的 data-pjax 属性的表单都会触发 pjax 请求。

```
use yii\widgets\Pjax;
use yii\widgets\ActiveForm;

Pjax::begin([
    // Pjax 配置项
]);
    $form = ActiveForm::begin([
        'options' => ['data' => ['pjax' => true]],
        // more ActiveForm options
    ]);

        // ActiveForm content

    ActiveForm::end();
Pjax::end();
```

> **提示：** 请小心处理 Pjax 小部件中的链接， 因为响应也将在小部件内呈现。为了防止这种情况， 使用 data-pjax="0" HTML 属性。

提交按钮和文件上传中的值 在处理 文件[3] 和 提交按钮值[4] 时使用 jQuery.serializeArray() 有已知的问题，这将不会被解决，而是被弃用，以支持 HTML5 中引入的 FormData 类。

这意味着对 ajax 或使用 Pjax 小部件的文件和提交按钮值的唯一官方支持取决于 FormData 类的 浏览器支持[5]。

7.1.4 延伸阅读

下一节 输入验证 处理提交的表单数据的服务器端验证，以及 ajax 和客户端验证。

要学会有关表格的更复杂的用法，你可以查看以下几节：

- 收集列表输入 同一种类型的多个模型的采集数据。
- 多模型同时输入 在同一窗口中处理多个不同的模型。
- 文件上传 如何使用表格来上传文件。

[3] https://github.com/jquery/jquery/issues/2321
[4] https://github.com/jquery/jquery/issues/2321
[5] https://developer.mozilla.org/zh-CN/docs/Web/API/FormData#%E6%B5%8F%E8%A7%88%E5%99%A8%E5%85%BC%E5%AE%B9%E6%80%A7

7.2 输入验证

根据经验，您永远不应该信任从最终用户收到的数据， 并且应该在充分利用之前对其进行验证。

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();

// 根据用户的输入填充到模型的属性中
$model->load(\Yii::$app->request->post());
// 等效于下面这样:
// $model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // 所有输入通过验证
} else {
    // 验证失败: $errors 是一个包含错误信息的数组
    $errors = $model->errors;
}
```

7.2.1 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'],
    ];
}
```

`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:

```
[
    // 必填，指出哪个属性由这个规则验证。
    // 如果是一个属性，可以直接写属性的名字
    // 不必写在数组里。
    ['attribute1', 'attribute2', ...],

    // 必填，指出规则的类型。
    // 可以是一个类名，验证器别名，或者是一个验证器方法名
```

```
    'validator',

    // 可选，指出这个规则在哪个（些）场景下生效
    // 如果没有给出，意味着这个规则在所有场景都生效
    // 如果你希望在所有场景下生效，但是在排除的场景里不生效。
    // 可以配置 "except" 选项
    'on' => ['scenario1', 'scenario2', ...],

    // 可选，为验证器对象指定额外的配置
    '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。

> **注意：** 可以方便地给规则命名比如：
>
> ```
> public function rules()
> {
>  return [
>  // ...
>  'password' => [['password'], 'string', 'max' => 60],
>  ];
> }
> ```
>
> 你可以在子模型中使用：

```
public function rules()
{
    $rules = parent::rules();
    unset($rules['password']);
    return $rules;
}
```

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 `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，`number` 验证器就支持 `tooBig` 和 `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`when` 属性来定义相关条件。举例而言，

```
['state', 'required', 'when' => function($model) {
    return $model->country == 'USA';
}]
```

`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 $attribute 待测特性名
 * @return bool 返回是否启用该规则
 */
function ($model, $attribute)
```

若你需要支持客户端的条件验证，你应该配置`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：

```
return [
    [['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。

下面的代码示例展示了对用户输入的完整处理， 这将确保只将整数值存储在一个属性中：

```
['age', 'trim'],
['age', 'default', 'value' => null],
['age', 'integer', 'integerOnly' => true, 'min' => 0],
['age', 'filter', 'filter' => 'intval', 'skipOnEmpty' => true],
```

以上代码将对输入执行以下操作：

1. 从输入值中去除前后空白。
2. 确保空输入在数据库中存储为 `null`；我们区分 未设置 值和实际值为 `0` 之间的区别。 如果值不允许为 `null`，则可以在此处设置另一个默认值。

3. 如果该值不为空，则验证该值是否为大于0的整数。大多数验证器的 `$skipOnEmpty` 属性都被设置为`true`。
4. 确保该值为整数类型，例如将字符串 `'42'` 转换为整数 `42`。在这里，我们将 `$skipOnEmpty` 设置为 `true`，默认情况下，在 `filter` 验证器里这个属性是 `false`。

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，

```
return [
    // 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。

7.2.2 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 `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 `hasErrors()` 方法来检查验证通过与否，并通过 `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`。

7.2.3 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
 * @param \yii\validators\InlineValidator $validator 相关的 InlineValidator
实例。
 * 此参数自版本 2.0.11 起可用。
 */
function ($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 [
            // 定义为模型方法 validateCountry() 的行内验证器
            ['country', 'validateCountry'],

            // 定义为匿名函数的行内验证器
            ['token', function ($attribute, $params) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'The token must contain
                    letters or digits.');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'The country must be either "USA" or
            "Web".');
        }
    }
}
```

> **注意：** 从 2.0.11 版本开始，你可以用 `yii\validators\InlineValidator::addError()` 方法添加错误信息到模型里。用这种方法 的话，

错误信息可以通过 yii\i18n\I18N::format() 格式化。 你还可以在错误信息里分别用 {attribute} 和 {value} 来引用 属性的名字（不必手动去写）和属性的值：

```
$validator->addError($this, $attribute, 'The value "{value}" is not
acceptable for {attribute}.');
```

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 skipOnEmpty 以及 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，

比如上述行内验证器也可以转移到新的验证类 [[components/validators/CountryValidator]]。这种情况下，我们可以用 yii\validators\Validator::addError() 来给模型设置自定义的错误信息。

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'The country must be either
            "USA" or "Web".');
        }
    }
}
```

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， 后两者的实现是通过调用 validateValue() 实现的。

下面就是一个怎样把自定义验证器在模型中使用的例子。

```
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::class],
            ['email', 'email'],
        ];
    }
}
```

7.2.4 多属性验证

某些情况下验证器可以包含多个属性。考虑下面的情况:

```
class MigrationForm extends \yii\base\Model
{
    /**
     * 一个成年人的最少花销
     */
    const MIN_ADULT_FUNDS = 3000;
    /**
     * 一个孩子的最小花销
     */
    const MIN_CHILD_FUNDS = 1500;

    public $personalSalary;
    public $spouseSalary;
    public $childrenCount;
    public $description;

    public function rules()
    {
        return [
            [['personalSalary', 'description'], 'required'],
            [['personalSalary', 'spouseSalary'], 'integer', 'min' =>
            self::MIN_ADULT_FUNDS],
            ['childrenCount', 'integer', 'min' => 0, 'max' => 5],
            [['spouseSalary', 'childrenCount'], 'default', 'value' => 0],
            ['description', 'string'],
        ];
    }
}
```

创建验证器

比如我们需要检查下家庭收入是否足够给孩子们花销。此时我们可以创建一个行内验证器 `validateChildrenFunds` 来解决这个问题，它仅仅在 `childrenCount` 大于 0 的时候才去检查。

请注意，我们不要把所有需要验证的属性 (`['personalSalary', 'spouseSalary', 'childrenCount']`) 都附加到 验证器上。因为这样做同一个验证器将会对每个属性都执行一遍验证（总共三次），但是实际上我们只需要对整个属性集执行一次验证而已。

你可以使用属性集合里的任何一个（或者使用你认为最相关的那个属性）：

```
['childrenCount', 'validateChildrenFunds', 'when' => function ($model) {
    return $model->childrenCount > 0;
}],
```

`validateChildrenFunds` 的实现可以是下面这样的：

```
public function validateChildrenFunds($attribute, $params)
{
    $totalSalary = $this->personalSalary + $this->spouseSalary;
    // Double the minimal adult funds if spouse salary is specified
    $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 :
    self::MIN_ADULT_FUNDS;
    $childFunds = $totalSalary - $minAdultFunds;
    if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
        $this->addError('childrenCount', 'Your salary is not enough for
        children.');
    }
}
```

你可以忽略 `$attribute` 参数，因为这个验证过程不仅仅关联一个属性。

添加错误信息

在添加错误信息的时候，如果是多个属性，可以根据自己想要的格式使用多种情况：

- 选择一个你认为最相关的字段把错误信息添加到它的属性里：

```
$this->addError('childrenCount', 'Your salary is not enough for children.');
```

- 选择多个相关的属性乃至所有属性给它们添加同样的错误信息。在使用 `addError` 之前我们可以先把错误信息存储到 一个独立的变量里，这样可以减少代码重复性。

```
$message = 'Your salary is not enough for children.';
$this->addError('personalSalary', $message);
$this->addError('wifeSalary', $message);
$this->addError('childrenCount', $message);
```

或者使用循环：

```
$attributes = ['personalSalary', 'wifeSalary', 'childrenCount'];
foreach ($attributes as $attribute) {
    $this->addError($attribute, 'Your salary is not enough for children.');
}
```

- 添加通用错误信息（不相关于特定的属性）。我们可以用一个不存在的属性名添加错误信息 比如 `*`，因为这时是不检查属性的存在性的。

```
$this->addError('*', 'Your salary is not enough for children.');
```

这种情况下，我们不会在表单域里看到错误信息。为了展示这个错误信息，我们可以在视图里使用错误汇总：

```
<?= $form->errorSummary($model) ?>
```

> **注意：** 创建一次验证多个属性的验证器的参考说明在这里 community cookbook[6].

7.2.5 客户端验证

当终端用户通过 HTML 表单提供输入数据时，基于 JavaScript 的客户端验证是可取的， 因为它允许用户更快地找出输入错误，从而提供更好的用户体验。你可以尝试使用或者自己实现一个 除了支持服务端验证 之外 还支持客户端验证的验证器。

> **信息：** 尽管客户端验证是值得的，但它不是必须的。客户端验证的主要目的是给终端用户提供 一个较好的体验。正如 “永远不用相信终端用户的输入数据” 一样，你也不能完全信任客户端验证。 基于这个考虑的话，正如前文描述所说，你应该永远在服务端通过调用 `yii\base\Model::validate()` 方法 进行服务端验证。

使用客户端验证

许多 核心验证器 支持开箱即用的客户端验证。你需要做的 就是使用 `yii\widgets\ActiveForm` 构建你的 HTML 表单。 比如，下面的 `LoginForm` 声明了两个 规则：一个使用 required 核心验证器，它支持客户端的验证，也支持服务端的 验证；另一个使用 `validatePassword` 行内验证器，它只支持在服务端 验证。

```
namespace app\models;

use yii\base\Model;
```

[6] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-validator-multiple-attributes.md

```
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'],

            // password 用 validatePassword() 方法验证
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Incorrect username or password.');
        }
    }
}
```

下面的代码构建了包含 `username` 和 `password` 两个表单项的 HTML 表单。如果不输入任何内容直接提交表单，你就会发现提示你输入内容的错误信息立刻出现， 而这并没有和服务端交互。

```
<?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` 读取在模型中声明的规则，然后 生成验证器支持客户端验证对应的 JavaScript 代码。当用户 改变表单项或者提交整个表单的时候，客户端验证的 JavaScript 就会触发。

如果你想完全关闭客户端验证，你可以设置 `yii\widgets\ActiveForm::$enableClientValidation` 属性为 `false` 。你也可以通过设置它们的 `yii\widgets\ActiveField::$enableClientValidation` 属性为 `false` 来单独关闭某一个表单项。 当在表单项级别和表单级别都设置了 `enableClientValidation` 的时候， 前者（表单项）的级别优先生效。

> **信息：** 从 2.0.11 版本开始，所有继承 `yii\validators\Validator` 的验证器都可以通过

- `yii\validators\Validator::getClientOptions()` 这个单独的方法接收客户端选项。可以这样使用：

- 如果你想自己实现自定义的客户端验证但是保留服务端的验证器选项；
- 继承或者自定义符合你特殊的需求：

```
public function getClientOptions($model, $attribute)
{
    $options = parent::getClientOptions($model, $attribute);
    // 修改 $options

    return $options;
}
```

实现客户端验证

为了创建一个支持客户端验证的验证器，你应该实现 `yii\validators\Validator::clientValidateAttribute()` 方法，该方法返回一段 JavaScript 代码 用来在客户端执行验证。在这段 JavaScript 代码里，你可以使用下面几个 预定义的变量：

- `attribute`：被验证的属性名。
- `value`：被验证的值。
- `messages`：一个给属性保存验证错误信息的数组。
- `deferred`：一个支持添加 deferred 对象的数组（下一部分再说）。

下面的例子，我们创建了一个 `StatusValidator` 验证器，它用来验证一个输入和存在的状态相比， 是否是有效的状态输入。这个验证器支持服务端验证也支持客户端验证。

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = 'Invalid status input.';
    }

    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, JSON_UNESCAPED_SLASHES |
        JSON_UNESCAPED_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

> **提示：** 上面给出的代码主要展示如何支持客户端验证。在实际使用中，你可以使用 in 核心验证器来实现同样的目标。你也可以 像下面那样写验证规则：
>
> ```
> [
>  ['status', 'in', 'range' =>
>  Status::find()->select('id')->asArray()->column()],
> ]
> ```
>
> **提示：** 如果你想手动调整客户端的验证，比如动态地添加表单项或者做一些自定义的 UI 逻辑，请参考 Working with ActiveForm via JavaScript[7] 在 Yii 2.0 Cookbook。

Deferred 验证

如果你需要执行异步客户端验证，你可以创建 Deferred objects[8]。 比如要执行一段自定义的 AJAX 验证，可以使用下面的代码：

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

上面这个 `deferred` 变量是由 Yii 提供的，它是一个 Deferred 对象的数组。这个 `$.get()` jQuery 方法用来产生一个 Deferred 对象然后推送到 `deferred` 数组里。

[7] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-activeform-js.md

[8] https://api.jquery.com/category/deferred-object/

你也可以明确地创建一个 `deferred` 对象，当异步回调触发的时候调用它的 `resolve()` 方法。 下面的例子展示了如何在客户端验证一个上传图片的尺寸。

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Image too wide!!');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> **注意：** `resolve()` 方法必须在所有属性都验证完之后调用。不然表单不会 完成整体的验证流程。

为了简单起见，`deferred` 数组封装了一个快捷方法 `add()`，它可以自动创建 Deferred 对象 然后把它添加到 `deferred` 数组里。用这个方法，你可以简化上面的例子：

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                    messages.push('Image too wide!!');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

7.2.6 AJAX 验证

一些验证器只能工作在服务端，因为只有服务端才有必要的信息。 比如，验证一个用户名是否唯一，需要在服务端检查用户表。 这时候你可以使用基于 AJAX 的验证。它会在背后触发一个 AJAX 请求用来验证输入项而且还能 保持和通常客户端验证一样的用户体验。

给一个单独的表单项开启 AJAX 验证，你只需要设置 `enableAjaxValidation` 属性为 `true`，然后指定一个唯一的表单 `id`:

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

如果要给所有的表单项开启 AJAX 验证，可以在表单级别设置 `enableAjaxValidation` 属性为 `true` 就行:

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> **注意:** 当在表单项级别和表单级别都设置了 `enableAjaxValidation` 属性的时候， 前者（表单项级别）优先生效。

你也需要在服务端准备处理这样的 AJAX 请求。 这个可以在控制器的动作里通过如下的代码片段来实现:

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
{
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

上述代码将会检测当前的请求是否源自 AJAX。如果是的话，它将运行验证过程, 然后 返回一段 JSON 格式的错误信息来响应这次请求。

> **信息:** 你也可以用 Deferred 验证 执行 AJAX 验证。 话说回来，这里描述的 AJAX 验证还是比较系统并且需要较少的代码开销。

当 `enableClientValidation` 和 `enableAjaxValidation` 都设置为 `true` 时，只有客户端验证成功之后 才会触发 AJAX 的验证请求。注意，如果验证某个表单项的时候凑巧 `validateOnChange`，`validateOnBlur` 或者 `validateOnType` 其中之一设置了 `true`,那么这个表单项在单独通过这样的客户端验证时， 也会发起 AJAX 请求。

7.3 文件上传

在Yii里上传文件通常使用 `yii\web\UploadedFile` 类， 它把每个上传的文件封装成 `UploadedFile` 对象。 结合 `yii\widgets\ActiveForm` 和 models，你可以轻松实现安全的上传文件机制。

7.3.1 创建模型

和普通的文本输入框类似，当要上传一个文件时，你需要创建一个模型类并且用其中的某个属性来接收上传的文件实例。 你还需要声明一条验证规则以验证上传的文件。 举例来讲，

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
            . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

在以上代码里，`imageFile` 属性用于接收上传的文件实例。它对应一条 `file` 验证规则， 该规则使用 `yii\validators\FileValidator` 来确保只上传扩展名为 `png` 或 `jpg` 的文件。 `upload()` 方法会执行该验证并且把上传的文件保存在服务器上。

通过 `file` 验证器，你可以检查文件的扩展名，大小，MIME类型等等。详情请查阅 Core Validatators 章节。

> **提示：** 如果你要上传的是一张图片，可以考虑使用 `image` 验证器。 `image` 验证器是通过 `yii\validators\ImageValidator` 实现验证的，确保对应的模型属性 收到的文件是有效的图片文件，然后才保存，或者使用扩展类 Imagine Extension[9] 进行处理.

7.3.2 渲染文件输入

接下来，在视图里创建一个文件输入控件

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

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

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

    <button>Submit</button>

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

需要注意的是要记得在表单选项里加入 `enctype` 属性以确保文件能被正常上传。 `fileInput()` 方法会渲染一个 `<input type="file">` 标签，让用户可以选择一个文件上传。

> **提示：** 自2.0.8版本开始，当使用文件输入字段时，`yii\web\widgets\ActiveField::fileInput` 会自动向表单添加 `enctype` 选项。

7.3.3 视图和模型的连接

现在，在控制器方法里编写连接模型和视图的代码以实现文件上传。

```
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();
```

[9] https://github.com/yiisoft/yii2-imagine

```
        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model,
            'imageFile');
            if ($model->upload()) {
                // 文件上传成功
                return;
            }
        }

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

在上面的代码里，当提交表单的时候，`yii\web\UploadedFile::getInstance()`方法就被调用， 上传的文件用一个 `UploadedFile` 实例表示。然后，我们依靠模型的验证规则确保上传的文件是有效的， 并将文件保存在服务器上。

7.3.4 上传多个文件

将前面所述的代码做一些调整，也可以一次性上传多个文件。

首先你得调整模型类，在 `file` 验证规则里增加一个 `maxFiles` 选项，用以限制一次上传文件的最大数量。 `upload()`方法也得修改,以便一个一个地保存上传的文件。将 `maxFiles` 设置为 `0` 意味着可以同时上传的文件数量没有限制。 允许同时上传的文件的最大数量也受到 PHP 指令 `max_file_uploads`[10]的限制， 默认为20。还应该更新 `upload()` 方法以逐个保存上传的文件。

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg', 'maxFiles' => 4],
        ];
    }

    public function upload()
    {
```

[10] https://www.php.net/manual/zh/ini.core.php#ini.max-file-uploads

```
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' .
                $file->extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

在视图文件里，你需要把 `multiple` 选项添加到 `fileInput()` 函数调用里， 这样文件输入控件就可以接收多个文件。

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

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' =>
true, 'accept' => 'image/*']) ?>

    <button>Submit</button>

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

最后，在控制器的 action 方法中，你应该调用 `UploadedFile::getInstances()` 而不是 `UploadedFile::getInstance()` 来把 `UploadedFile` 实例数组赋值给 `UploadForm::imageFiles`。

```
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) {
            $model->imageFiles = UploadedFile::getInstances($model,
            'imageFiles');
            if ($model->upload()) {
                // 文件上传成功
                return;
            }
        }
```

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

7.4 收集列表输入

有时你需要在一个表单中以单一的形式处理多个模型。例如，有多个设置， 每个设置存储为一个 name-value，并通过 `Setting` 活动记录 模型来表示。这种形式也常被称为“列表输入”。与此相反， 处理不同模型的不同类型，在 多模型同时输入章节中介绍。

下面展示了如何在 Yii 中收集列表输入。

在三种不同的情况下，所需处理的略有不同：

- 从数据库中更新一组固定的记录
- 创建一个动态的新记录集
- 更新、创建和删除一页记录

与之前介绍的单一模型表单相反，我们现在用的是一个数组类的模型。这个数组将 每个模型传递到视图并以一种类似于表格的方式来显示表单字段。 我们使用 `yii\base\Model` 助手类方法来一次性地加载和验证多模型数据：

- `Model::loadMultiple()` 将数据加载到一个数组中。
- `Model::validateMultiple()` 验证一系列模型。

更新一组固定的记录

让我们从控制器的动作开始：

```
<?php

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]);
    }
}
```

在上面的代码中，当用模型来从数据库获取数据时，我们使用 `indexBy()` 来让模型的主键成为一个数组的索引。其中 `Model::loadMultiple()` 用于接收以 POST 方式提交的表单数据并填充多个模型， `Model::validateMultiple()` 一次验证多个模型。 正如我们之前验证的模型，使用了 `validateMultiple()`，现在通过传递 `false` 作为 `save()`的一个参数使其不会重复验证两次。

现在在 `update` 视图的表单：

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

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo $form->field($setting, "[$index]value")->label($setting->name);
}

ActiveForm::end();
```

在这里，我们为每个设置渲染了名字和一个带值的输入。重要的是给 input name 增加添加适当的索引， 因为这是由 `Model::loadMultiple()` 来决定以哪些值来填补哪个模型。

创建一组动态的新记录

创造新的记录与修改记录很相似，除部分实例化模型不同之外：

```
public function actionCreate()
{
    $count = count(Yii::$app->request->post('Setting', []));
    $settings = [new Setting()];
    for($i = 1; $i < $count; $i++) {
        $settings[] = new Setting();
    }

    // ...
}
```

在这里，我们创建了一个初始的 `$settings` 数组包含一个默认的模型，所以始终至少有一个文本字段是可见的。 此外，我们为每个可能会收到的输入行添加更多的模型。

在视图中，可以使用 JavaScript 来动态地添加新的输入行。

把更新，创建和删除结合在一个页面上

> **注意：** 此章节正在开发中。
>
> 还没有内容。

TBD

7.5 多模型的复合表单

当需要处理复杂数据，很可能你需要使用多个不同的模型来收集用户提交的数据。 举例来说，假设用户登录信息保存在 `user` 表，但是用户基本信息保存在 `profile` 表， 你可能需要同时使用 `User` 模型和 `Profile` 模型来获取用户登录信息和基本信息。 使用 Yii 提供的模型和表单支持，解决这样的问题和处理单一模型并不会有太大的区别。

下面，我们将为你展示怎样创建一个表单并同时处理 `User` 和 `Profile` 这两个模型。

首先，控制器中收集用户提交数据的动作(action)可以按照下面写的这样，

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
            throw new NotFoundHttpException("没有找到用户登录信息。");
        }

        $profile = Profile::findOne($user->profile_id);

        if (!$profile) {
            throw new NotFoundHttpException("没有找到用户基本信息。");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) &&
        $profile->load(Yii::$app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
```

```
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}
```

在 `update` 动作中，我们首先从数据库中获取需要更新的 `$user` 和 `$profile` 这两个模型。 我们可以调用 `yii\base\Model::load()` 方法把用户提交数据填充到两个模型中。 如果加载成功，验证两个表单并保存 — 请注意我们使用了 `save(false)` 方法用来忽略内部保存时的二次验证， 因为用户输入的数据已经手动验证过了。 如果填充数据失败，直接显示下面的 `update` 视图内容:

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

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...other input fields...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('更新数据', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

你可以看到，在 `update` 视图中，我们同时显示了两个模型 `$user` 和 `$profile` 的属性的输入栏。

7.6 在客户端扩展 ActiveForm

`yii\widgets\ActiveForm` 小部件附带一组用于客户端验证的 JavaScript 方法。 它的实现非常灵活，可以让你以不同的方式扩展它。 下面我们来看描述。

7.6.1 ActiveForm 事件

ActiveForm 触发一系列专用事件。使用类似以下的代码，您可以订阅这些代码 事件并处理它们:

```
$('#contact-form').on('beforeSubmit', function (e) {
        if (!confirm("Everything is correct. Submit?")) {
                return false;
        }
        return true;
});
```

在下文中，我们将查看可用的事件。

`beforeValidate` 是在验证整个表单之前触发的。

事件处理程序的签名应该是：

```
function (event, messages, deferreds)
```

其中

- `event`：一个 Event 对象。
- `messages`：一个关联数组，其中键是属性ID， 值是相应属性的错误消息数组。
- `deferreds`：一个 Deferred 对象数组。你可以使用 `deferreds.add(callback)` 来添加一个新的延迟验证。

如果处理程序返回一个布尔型 `false`，它将在此事件后停止进一步的表单验证。 结果 `afterValidate` 事件将不会被触发。

`afterValidate` 是在验证整个表单后触发的。

事件处理程序的签名应该是：

```
function (event, messages, errorAttributes)
```

其中

- `event`：一个 Event 对象。
- `messages`：一个关联数组，其中键是属性ID， 值是相应属性的错误消息数组。
- `errorAttributes`：一个具有验证错误的属性数组。有关此参数的结构，请参阅 `attributeDefaults`。

`beforeValidateAttribute` 事件是在验证属性之前触发的。 事件处理程序的签名应该是：

```
function (event, attribute, messages, deferreds)
```

其中

- `event`: 一个 Event 对象。
- `attribute`: 要验证的属性。 请参阅这个参数的 `attributeDefaults` 结构。
- `messages`: 可以为其添加指定属性的验证错误消息的数组。
- `deferreds`: 一个 Deferred 对象数组。你可以使用 `deferreds.add(callback)` 来添加一个新的延迟验证。

如果处理程序返回布尔型 `false`，它将停止进一步验证指定的属性。 结果，`afterValidateAttribute` 事件将不会被触发。

`afterValidateAttribute` 事件在验证整个表单和每个属性后触发的。

事件处理程序的签名应该是:

```
function (event, attribute, messages)
```

其中

- `event`: 一个 Event 对象。
- `attribute`: 该属性会被验证。有关此参数的结构， 请参阅 `attributeDefaults`。
- `messages`: 可以为其添加指定属性的其他验证错误消息的 数组。

`beforeSubmit` 是在所有验证通过后且提交表单之前触发的。

事件处理程序的签名应该是:

```
function (event)
```

其中 `event` 是一个 Event 对象。

如果处理返回布尔型“false”，它将停止表单提交。

`ajaxBeforeSend` 事件是在发送用于基于AJAX的验证的AJAX请求之前触发的。

事件处理程序的签名应该是:

```
function (event, jqXHR, settings)
```

其中

- `event`: 一个 Event 对象。
- `jqXHR`: 一个 jqXHR 对象
- `settings`: AJAX 请求的设置

`ajaxComplete` 事件是在完成基于 AJAX 验证的 AJAX 请求后触发的。

事件处理程序的签名应该是：

```
function (event, jqXHR, textStatus)
```

where

- `event`: 一个 Event 对象。
- `jqXHR`: 一个 jqXHR 对象
- `textStatus`: 请求的状态 ("success", "notmodified", "error", "timeout", "abort", or "parsererror")。

7.6.2 通过 AJAX 提交表单

虽然可以在客户端或通过 AJAX 请求进行验证，但作为默认的正常请求 表单提交本身已完成。如果你想通过 AJAX 提交表单，你可以 通过以下方式处理表单的 `beforeSubmit` 事件做到这一点：

```
var $form = $('#formId');
$form.on('beforeSubmit', function() {
    var data = $form.serialize();
    $.ajax({
        url: $form.attr('action'),
        type: 'POST',
        data: data,
        success: function (data) {
            // 执行成功
        },
        error: function(jqXHR, errMsg) {
            alert(errMsg);
        }
     });
     return false; // 防止默认提交
});
```

要了解更多关于 jQuery `ajax()` 函数的信息，请参阅 jQuery 文档[11]。

7.6.3 动态添加字段

在现代 Web 应用程序中，您经常需要在向用户显示表单后更改表单。 这可以例如在点击"加号"图标之后添加新的字段。 要为这些字段启用客户端验证，他们必须注册 ActiveForm JavaScript 插件。

您必须自行添加一个字段，然后将其添加到验证列表中：

```
$('#contact-form').yiiActiveForm('add', {
    id: 'address',
    name: 'address',
```

[11] https://api.jquery.com/jQuery.ajax/

```
    container: '.field-address',
    input: '#address',
    error: '.help-block',
    validate:  function (attribute, value, messages, deferred, $form) {
        yii.validation.required(value, messages, {message: "Validation
        Message Here"});
    }
});
```

要从验证列表中删除一个字段，使它不被验证，您可以执行以下操作：

```
$('#contact-form').yiiActiveForm('remove', 'address');
```

Chapter 8

显示数据（Displaying Data）

8.1 数据格式器（Data Formatting）

你可以使用 `formatter` application component 来格式化数据。 默认 `fomatter` 由 `yii\i18n\Formatter` 来实现，这个组件提供了一系列关于日期/时间，数字，货币等的格式化方法。 使用方法如下：

```
$formatter = \Yii::$app->formatter;

// output: January 1, 2014
echo $formatter->asDate('2014-01-01', 'long');

// output: 12.50%
echo $formatter->asPercent(0.125, 2);

// output: <a href="mailto:cebe@example.com">cebe@example.com</a>
echo $formatter->asEmail('cebe@example.com');

// output: Yes
echo $formatter->asBoolean(true);
// it also handles display of null values:

// output: (Not set)
echo $formatter->asDate(null);
```

我们可以看到，所有的方法都形似 `asXyz()`，这个 `Xzy` 就是所支持的格式化类型。 当然你也可以使用类方法 `format()` 来进行格式化，通过这个类方法，你可以更自由地控制格式化的数据，这时候，类方法通常配合 `yii\grid\GridView` 或者 `yii\widgets\DetailView` 来使用。

举个例子：

```
// output: January 1, 2014
echo Yii::$app->formatter->format('2014-01-01', 'date');

// 你可以在第二个参数指定一个数组，这个数组提供了一些配置的参数
// 例如这个 2 就是 asPercent() 方法的 $decimals 参数
```

```
// output: 12.50%
echo Yii::$app->formatter->format(0.125, ['percent', 2]);
```

> **注意：** formatter 组件用来格式化最终展示给用户的数据。 如果你想要将用户的输入进行格式化或者只是将一些别的日期数据进行格式化（这里的格式化说的是机器可读的格式化）， 不要使用这个组件， 而应该使用 yii\validators\DateValidator 和 yii\validators\NumberValidator 进行用户输入格式化 对于机器可读的日期和时间格式之间的简单转换， PHP 方法 date()[1] 就足够了。

8.1.1 配置 Formatter（Configuring Formatter）

可以对 formatter 组件在 application configuration 中进行配置。 例如，

```
return [
    'components' => [
        'formatter' => [
            'dateFormat' => 'dd.MM.yyyy',
            'decimalSeparator' => ',',
            'thousandSeparator' => ' ',
            'currencyCode' => 'EUR',
        ],
    ],
];
```

可以参考 yii\i18n\Formatter 的配置

8.1.2 格式化时间/日期数据（Formatting Date and Time Values）

默认支持一下几种格式化格式

- date：这个变量将被格式化为日期 January 01, 2014。
- time：这个变量将被格式化为时间 14:23。
- datetime：这个变量将被格式化为日期+时间 January 01, 2014 14:23。
- timestamp：这个变量将被格式化为 UNIX 时间戳 unix timestamp[2]，例如 1412609982。
- relativeTime：这个变量将被格式化为人类可读的 当前相对时间 1 hour ago。
- duration：这个变量将被格式化为人类可读的时长 1 day, 2 minutes。

时间/日期数据默认使用 date, time， datetime 方法进行格式化，你可以对他们进行一些自己的配置，只需在配置文件里配置 dateFormat, timeFormat, 和 datetimeFormat 即可。

[1] https://www.php.net/manual/zh/function.date.php
[2] https://zh.wikipedia.org/wiki/UNIX%E6%97%B6%E9%97%B4

同时，你还可以配置它使用 ICU syntax[3]， 同时你也可以配置它使用 PHP date() 语法[4]，只需要加上 `php:` 前缀即可。 例如，

```
// ICU format
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06

// PHP date()-format
echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

> **信息：** ICU 不支持 PHP 格式语法的某些字母，并且这些 PHP intl 扩展名不能在 Yii 格式化程序中使用。 大多数（`w`，`t`，`L`，`B`，`u`，`I`，`Z`）对于格式化日期并不是真的有用，而是在进行数学计算时有用。 然而 `S` 和 `U` 可能是有用的。他们的行为可以通过以下方式实现：
>
> - 比如 `S`，这是当月的英文序数后缀（例如：st，nd，rd 或 th。），可以使用以下代码来替换：
>
>  ```
>  $f = Yii::$app->formatter;
>  $d = $f->asOrdinal($f->asDate('2017-05-15', 'php:j'));
>  echo "On the $d day of the month."; // 打印 "On the 15th day of the month."
>  ```
>
> - 比如 `U`，Unix 纪元，你能够使用 `timestamp` 来格式化。

当使用需要支持多种语言的应用程序时，您经常需要为不同的区域设置指定不同的日期和时间格式。 为了简化这项操作，你可以使用格式快捷键（例如：`long`，`short`）来代替。 格式化程序将根据当前有效的地区的 `locale` 将格式快捷方式转换为适当的程序。 支持以下格式的快捷方式（示例是假设设置 locale 为 `en_GB`）：

- `short`：将输出日期 `06/10/2014` 和时间 `15:58`；
- `medium`：将输出 `6 Oct 2014` 和 `15:58:42`；
- `long`：将输出 `6 October 2014` 和 `15:58:42 GMT`；
- `full`：将输出 `Monday, 6 October 2014` 和 `15:58:42 GMT`。

版本 2.0.7 起，支持格式化日期为不同的系统时钟， 请参阅格式化程序 `$calendar` 的 API 文档 - 有关如何设置其他日历的属性。

时区（Time Zones）

格式化时间/日期数据时，你会将他们转换成 `time zone` 这个时候，默认的时区为 UTC，除非你另外指定 `yii\i18n\Formatter::$defaultTimeZone`。

下面使用 `Europe/Berlin` 作为默认 `time zone`

```
// formatting a UNIX timestamp as a time
echo Yii::$app->formatter->asTime(1412599260); // 14:41:00

// formatting a datetime string (in UTC) as a time
```

[3] https://unicode-org.github.io/icu/userguide/format_parse/datetime/
[4] https://www.php.net/manual/zh/function.date.php

```
echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00

// formatting a datetime string (in CEST) as a time
echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

如果 time zone 未在格式化组件上明确设置， time zone configured in the application 将会被使用， 这与 PHP 配置中设置的时区相同。

> 不同的政府和地区政策决定不同的时区， 你在你的时区数据库中可能拿不到最新的数据。 这时你可以戳 ICU manual[5] 来查看如何更新时区。 同时，这篇也可以作为参考 Setting up your PHP environment for internationalization。

8.1.3 格式化数字（Formatting Numbers）

formatter 支持如下的方法

- integer: 这个变量将被格式化为整形 e.g. 42.
- decimal: 这个变量将被格式化为带着逗号的指定精度的浮点型， 例如: 2,542.123 或 2.542,123。
- percent: 这个变量将被格式化为百分比 e.g. 42%.
- scientific: 这个变量将被格式化为科学计数法 e.g. 4.2E4.
- currency: 这个变量将被格式化为货币， 例如: £420.00。 使用这个方法前请确认是否已经正确配置 locale
- size: 这个变量将被格式化为人类可读的字节数 e.g. 410 kibibytes.
- shortSize: 这个变量将被格式化为人类可读的字节数（缩写） size，例如: 410 KiB。

你可以使用 decimalSeparator 和 thousandSeparator 来进行调整。 他们都会根据当前的 locale 来进行格式化.

如果你想要进行更高级的配置, 可以使用 yii\i18n\Formatter::$numberFormatterOptions 和 yii\i18n\Formatter::$numberFormatterTextOptions， NumberFormatter class[6] 来进行格式化。 例如，为了调整小数部分的最大值和最小值，你可以配置 yii\i18n\Formatter::$numberFormatterOptions 如下:

```
'numberFormatterOptions' => [
    NumberFormatter::MIN_FRACTION_DIGITS => 0,
    NumberFormatter::MAX_FRACTION_DIGITS => 2,
]
```

8.1.4 其他的格式化（Other Formats）

除了时间/日期和数字的格式化，Yii 还支持如下的常用格式化

- raw: 这个值会被原样输出，这是一个无效的伪格式化程序， 除了null 将使用 nullDisplay 来格式化。

[5] https://unicode-org.github.io/icu/userguide/datetime/timezone/#updating-the-time-zone-data

[6] https://www.php.net/manual/zh/class.numberformatter.php

- text：这个值是 HTML 来编码的。 这是 GridView DataColumn 使用的默认格式。
- ntext：这个值将会被格式化用 HTML 编码的纯文本， 其中新的一行将用换行符来隔开。
- paragraphs：这个值被格式化为包含在 <p> 标签中的 HTML 编码的文本段落。
- html：这个值使用 HtmlPurifier 来格式化以避免遭受 XSS 攻击。你能够设置一切其他选项比如 ['html', ['Attr.AllowedFrameTargets' => ['_blank']]]。
- email：这个值被格式化为 mailto-链接。
- image：这个值被格式化为图像标签。
- url：这个值被格式化为超链接。
- boolean：这个值被格式化为布尔值。 默认情况下，转化为当前的引用程序语言，true 表现为 Yes 并且 false 表现为 No。 您可以通过配置 yii\i18n\Formatter::$booleanFormat 的属性来调整它。

8.1.5 空值（Null Values）

空值（null）会被特殊格式化。fommater 默认会将空值格式化为 (not set) 对应的当前的语言。 你可以配置 nullDisplay 属性来进行个性化。

8.1.6 本地日期格式化（Localizing Data Format）

如上所述，格式化应用程序可以使用当前有效的 locale 来确定 如何格式化适用于目标国家/地区的值。 例如，对于不同的区域设置，相同的日期可能会有不同的格式：

```
Yii::$app->formatter->locale = 'en-US';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: January 1, 2014

Yii::$app->formatter->locale = 'de-DE';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: 1. Januar 2014

Yii::$app->formatter->locale = 'ru-RU';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: 1 января 2014
г.
```

默认配置下，当前 locale 决定于 yii\base\Application::$language. 你可以覆盖 yii\i18n\Formatter::$locale 属性来满足不同的需要。

> **注意：** Yii formatter 依赖 PHP intl extension[7] 来进行本地数据格式化 因为不同的 ICU 库可能会导致不同的输出，所以请在你的所有机器上保持 ICU 库的一致性。 请参阅 Setting up your PHP environment for internationalization。
>
> 如果 intl 扩展没有被安装，数据格式化不会考虑本地化。

[7] https://www.php.net/manual/zh/book.intl.php

在 32 位系统中，1901 年前或者 2038 年后的日期数据将不会被本地化， 因为 ICU 使用的是 32 位的 UNIX 时间戳。

8.2 分页

当一次要在一个页面上显示很多数据时，通常需要将其分成几部分， 每个部分都包含一些数据列表并且一次只显示一部分。这些部分在网页上被称为分页。

Yii 使用 `yii\data\Pagination` 对象来代表分页方案的有关信息。特别地，

- `total count` 指定数据条目的总数。 注意，这个数字通常远远大于需要在一个页面上展示的数据条目。
- `page size` 指定每页包含多少数据条目。 默认值为 20。
- `current page` 给出当前的页码。默认值为 0，表示第一页。

通过一个已经完全明确的 `yii\data\Pagination` 对象，你可以部分地检索并且展示数据。 比如，如果你正在从数据库取回数据，你可以使用分页对象提供的对应值来指定 DB 查询语句中的 `OFFSET` 和 `LIMIT` 子句。 下面是个例子，

```
use yii\data\Pagination;

// 创建一个 DB 查询来获得所有 status 为 1 的文章
$query = Article::find()->where(['status' => 1]);

// 得到文章的总数（但是还没有从数据库取数据）
$count = $query->count();

// 使用总数来创建一个分页对象
$pagination = new Pagination(['totalCount' => $count]);

// 使用分页对象来填充 limit 子句并取得文章数据
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

上述例子中，文章的哪一页将被返回？它取决于是否给出一个名为 `page` 的参数。 默认情况下，分页对象将尝试将 `current page` 设置为 `page` 参数的值。 如果没有提供该参数，那么它将默认为 0。

为了促成创建支持分页的 UI 元素，Yii 提供了 `yii\widgets\LinkPager` 挂件来展示一栏页码按钮， 用户可以通过点击它来指示应该显示哪一页的数据。 该挂件接收一个分页对象，因此它知道当前的页数和应该展示多少页码按钮。 比如，

```
use yii\widgets\LinkPager;

echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

如果你想手动地创建 UI 元素，你可以使用 yii\data\Pagination::createUrl() 来创建指向不同页面的 URLs。 该方法需要一个页码来作为参数，并且将创建一个包含页码并且格式正确的 URL， 例如，

```
// 指定要被创建的 URL 应该使用的路由
// 如果不指定，则使用当前被请求的路由
$pagination->route = 'article/index';

// 显示: /index.php?r=article%2Findex&page=100
echo $pagination->createUrl(100);

// 显示: /index.php?r=article%2Findex&page=101
echo $pagination->createUrl(101);
```

> **提示：** 创建分页对象时，你可以通过配置 pageParam 属性来自定义查询参数 page 的名字。

8.3 排序

展示多条数据时，通常需要对数据按照用户指定的列进行排序。 Yii 使用 yii\data\Sort 对象来代表排序方案的有关信息。 特别地，

- attributes 指定 属性，数据按照其排序。 一个属性可以就是简单的一个 model attribute， 也可以是结合了多个 model 属性或者 DB 列的复合属性。下面将给出更多细节。
- attributeOrders 给出每个属性当前设置的 排序方向。
- orders 按照低级列的方式给出排序方向。

使用 yii\data\Sort，首先要声明什么属性能进行排序。 接着从 attributeOrders 或者 orders 取得当前设置的排序信息， 然后使用它们来自定义数据查询。 例如，

```
use yii\data\Sort;

$sort = new 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',
        ],
    ],
]);

$articles = Article::find()
    ->where(['status' => 1])
    ->orderBy($sort->orders)
    ->all();
```

上述例子中，为 Sort 对象声明了两个属性： age 和 name。

age 属性是 Article 与 Active Record 类中 age 属性对应的一个简单属性。 上述声明与下述等同：

```
'age' => [
    'asc' => ['age' => SORT_ASC],
    'desc' => ['age' => SORT_DESC],
    'default' => SORT_ASC,
    'label' => Inflector::camel2words('age'),
]
```

name 属性是由 Article 的 firsr_name 和 last_name 定义的一个复合属性。 使用下面的数组结构来对它进行声明：

- asc 和 desc 元素指定了如何按照该属性进行升序和降序的排序。 它们的值代表数据真正地应该按照什么列和方向进行排序。 你可以指定一列或多列来指出到底是简单排序还是多重排序。
- default 元素指定了当一次请求时，属性应该按照什么方向来进行排序。 它默认为升序方向，意味着如果之前没有进行排序，并且 你请求按照该属性进行排序，那么数据将按照该属性来进行升序排序。
- label 元素指定了调用 yii\data\Sort::link() 来创建一个排序链接时应该使用什么标签。 如果不设置，将调用 yii\helpers\Inflector::camel2words() 来通过属性名生成一个标签。 注意，它并不是HTML编码的。

> **信息：** 你可以将 orders 的值直接提供给数据库查询来构建其 ORDER BY 子句。 不要使用 attributeOrders， 因为一些属性可能是复合的，是不能被数据库查询识别的。

你可以调用 yii\data\Sort::link() 来生成一个超链接，用户可以通过点击它来请求按照指定的属性对数据进行排序。 你也可以调用 yii\data\Sort::createUrl() 来生成一个可排序的 URL。 例如，

```
// 指定被创建的 URL 应该使用的路由
// 如果你没有指定，将使用当前被请求的路由
$sort->route = 'article/index';

// 显示链接，链接分别指向以 name 和 age 进行排序
echo $sort->link('name') . ' | ' . $sort->link('age');

// 显示: /index.php?r=article%2Findex&sort=age
echo $sort->createUrl('age');
```

yii\data\Sort 查看 sort 查询参数来决定哪一个属性正在被请求来进行排序。 当该参数不存在时，你可以通过 yii\data\Sort::$defaultOrder 来指定默认的排序。 你也可以通过配置 sortParam 属性来自定义该查询参数的名字。

8.4 数据提供者

在 Pagination 和 Sorting 部分, 我们已经介绍了如何允许终端用户选择一个特定的数据页面，根据一些字段对它们进行展现与排序。 因为分页和排序数据的任务是很常见的，所以Yii提供了一组封装好的*data provider*类。

数据提供者是一个实现了 `yii\data\DataProviderInterface` 接口的类。 它主要用于获取分页和数据排序。它经常用在 data widgets 小物件里，方便终端用户进行分页与数据排序。

下面的数据提供者类都包含在Yii的发布版本里面:

- `yii\data\ActiveDataProvider`: 使用 `yii\db\Query` 或者 `yii\db\ActiveQuery` 从数据库查询数据并且以数组项的方式或者 Active Record 实例的方式返回。
- `yii\data\SqlDataProvider`: 执行一段SQL语句并且将数据库数据作为数组返回。
- `yii\data\ArrayDataProvider`: 将一个大的数组依据分页和排序规格返回一部分数据。

所有的这些数据提供者遵守以下模式:

```
// 根据配置的分页以及排序属性来创建一个数据提供者
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// 获取分页和排序数据
$models = $provider->getModels();

// 在当前页获取数据项的数目
$count = $provider->getCount();

// 获取所有页面的数据项的总数
$totalCount = $provider->getTotalCount();
```

你可以通过配置 `pagination` 和 `sort`的属性来设定数据提供者的分页和排序行为。 属性分别对应于 `yii\data\Pagination` 和 `yii\data\Sort`。 你也可以对它们配置false来禁用分页和排序特性。

Data widgets,诸如 `yii\grid\GridView`，有一个属性名叫 `dataProvider`，这个属性能够提供一个 数据提供者的示例并且可以显示所提供的数据，例如，

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

这些数据提供者的主要区别在于数据源的指定方式上。在下面的部分， 我们将详细介绍这些数据提供者的使用方法。

8.4.1 活动数据提供者

为了使用 yii\data\ActiveDataProvider，你应该配置其 query 的属性。它既可以是一个 yii\db\Query 对象，又可以是一个 yii\db\ActiveQuery 对象。假如是前者，返回的数据将是数组； 如果是后者，返回的数据可以是数组也可以是 Active Record 对象。 例如，

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// 返回一个Post实例的数组
$posts = $provider->getModels();
```

假如在上面的例子中，$query 用下面的代码来创建，则数据提供者将返回原始数组。

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['status' => 1]);
```

> **注意：** 假如查询已经指定了 orderBy 从句，则终端用户给定的新的排序说明（通过 sort 来配置的） 将被添加到已经存在的从句中。任何已经存在的 limit 和 offset 从句都将被终端用户所请求的分页 （通过 pagination 所配置的）所重写。

默认情况下，yii\data\ActiveDataProvider使用 db 应用组件来作为数据库连接。你可以通过配置 yii\data\ActiveDataProvider::$db 的属性来使用不同数据库连接。

8.4.2 SQL数据提供者

yii\data\SqlDataProvider 应用的时候需要结合需要获取数据的SQL语句。 基于 sort 和 pagination 规格， 数据提供者会根据所请求的数据页面（期望的顺序）来调整 ORDER BY 和 LIMIT 的SQL从句。

为了使用 yii\data\SqlDataProvider，你应该指定 sql 属性以及 totalCount 属性，例如，

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// 返回包含每一行的数组
$models = $provider->getModels();
```

> **信息：** `totalCount` 的属性只有你需要 分页数据的时候才会用到。这是因为通过 `sql` 指定的SQL语句将被数据提供者所修改并且只返回当 前页面数据。数据提供者为了正确计算可用页面的数量仍旧需要知道数据项的总数。

8.4.3 数组数据提供者

`yii\data\ArrayDataProvider` 非常适用于大的数组。数据提供者允许你返回一个 经过一个或者多个字段排序的数组数据页面。为了使用 `yii\data\ArrayDataProvider`， 你应该指定 `allModels` 属性作为一个大的数组。这个大数组的元素既可以是一些关联数组（例如：DAO查询出来的结果）也可以是一些对象（例如：Active Record实例） 例如，

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];

$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
```

```
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
    ],
]);

// 获取当前请求页的每一行数据
$rows = $provider->getModels();
```

> **注意：** 数组数据提供者与 Active Data Provider 和 SQL Data Provider 这两者进行比较的话， 会发现数组数据提供者没有后面那两个高效，这是因为数组数据提供者需要加载所有的数据到内存中。

8.4.4 数据键的使用

当使用通过数据提供者返回的数据项的时候，你经常需要使用一个唯一键来标识每一个数据项。 举个例子，假如数据项代表的是一些自定义的信息，你可能会使用自定义ID作为键去标识每一个自定义数据。 数据提供者能够返回一个通过 `yii\data\DataProviderInterface::getModels()` 返回的键与数据项相对应的列表。 例如，

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => Post::find(),
]);

// 返回包含Post对象的数组
$posts = $provider->getModels();

// 返回与$posts相对应的主键值
$ids = $provider->getKeys();
```

在上面的例子中，因为你提供给 `yii\data\ActiveDataProvider` 一个 `yii\db\ActiveQuery` 对象， 它是足够智能地返回一些主键值作为键。你也可以明确指出键值应该怎样被计算出来， 计算的方式是通过使用一个字段名或者一个可调用的计算键值来配置。 例如，

```
// 使用 "slug" 字段作为键值
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// 使用md5(id)的结果作为键值
$provider = new ActiveDataProvider([
    'query' => Post::find(),
```

```
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.4.5 创建自定义数据提供者

为了创建自定义的数据提供者类，你应该实现 `yii\data\DataProviderInterface` 接口。 一个简单的方式是从 `yii\data\BaseDataProvider` 去扩展，这种方式允许你关注数据提供者的核心逻辑。 这时，你主要需要实现下面的一些方法：

- `prepareModels()`：准备好在当前页面可用的数据模型， 并且作为一个数组返回它们。
- `prepareKeys()`：接受一个当前可用的数据模型的数组， 并且返回一些与它们相关联的键。
- `prepareTotalCount`：在数据提供者中返回一个 标识出数据模型总数的值。

下面是一个数据提供者的例子，这个数据提供者可以高效地读取CSV数据：

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string 要读取的 CSV 文件的名称
     */
    public $filename;

    /**
     * @var string|callable 键列的名称或返回它的可调用列表
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject; // SplFileObject 非常便于搜索文件中的特定行


    /**
     * @inheritdoc
     */
    public function init()
    {
        parent::init();

        // open file
        $this->fileObject = new SplFileObject($this->filename);
    }
```

```
/**
 * @inheritdoc
 */
protected function prepareModels()
{
    $models = [];
    $pagination = $this->getPagination();

    if ($pagination === false) {
        // in case there's no pagination, read all lines
        while (!$this->fileObject->eof()) {
            $models[] = $this->fileObject->fgetcsv();
            $this->fileObject->next();
        }
    } else {
        // in case there's pagination, read only a single page
        $pagination->totalCount = $this->getTotalCount();
        $this->fileObject->seek($pagination->getOffset());
        $limit = $pagination->getLimit();

        for ($count = 0; $count < $limit; ++$count) {
            $models[] = $this->fileObject->fgetcsv();
            $this->fileObject->next();
        }
    }

    return $models;
}

/**
 * @inheritdoc
 */
protected function prepareKeys($models)
{
    if ($this->key !== null) {
        $keys = [];

        foreach ($models as $model) {
            if (is_string($this->key)) {
                $keys[] = $model[$this->key];
            } else {
                $keys[] = call_user_func($this->key, $model);
            }
        }

        return $keys;
    } else {
        return array_keys($models);
    }
}

/**
 * @inheritdoc
```

```
     */
    protected function prepareTotalCount()
    {
        $count = 0;

        while (!$this->fileObject->eof()) {
            $this->fileObject->next();
            ++$count;
        }

        return $count;
    }
}
```

8.4.6 使用数据过滤器过滤数据提供者

虽然您可以手动为活动数据提供构建条件， 例如使用 过滤数据 和 单独过滤表格 数据小部件， 但如果你需要灵活的过滤条件，Yii 的数据过滤器会非常有用。 以下是数据过滤器使用的方式：

```
$filter = new ActiveDataFilter([
    'searchModel' => 'app\models\PostSearch'
]);

$filterCondition = null;

// 您可以从任何来源加载过滤器。例如：
// 如果你更喜欢请求体中的 JSON，
// 使用 Yii::$app->request->getBodyParams() 如下：
if ($filter->load(\Yii::$app->request->get())) {
    $filterCondition = $filter->build();
    if ($filterCondition === false) {
        // Serializer would get errors out of it
        return $filter;
    }
}

$query = Post::find();
if ($filterCondition !== null) {
    $query->andWhere($filterCondition);
}

return new ActiveDataProvider([
    'query' => $query,
]);
```

`PostSearch` 模型用于定义允许过滤的属性和值：

```
use yii\base\Model;

class PostSearch extends Model
{
    public $id;
```

```
    public $title;

    public function rules()
    {
        return [
            ['id', 'integer'],
            ['title', 'string', 'min' => 2, 'max' => 200],
        ];
    }
}
```

数据过滤器非常灵活。您可以自定义构建的条件以及允许的运算符。 更多的细节请查看 `yii\data\DataFilter` 的 API 文档。

8.5 数据小部件

Yii提供了一套数据小部件 widgets ，这些小部件可以用于显示数据。 DetailView 小部件能够用于显示一条记录数据， ListView 和 GridView 小部件能够用于显示一个拥有分页、 排序和过滤功能的一个列表或者表格。

8.5.1 DetailView

`DetailView` 小部件显示的是单一 `model` 数据的详情。

它非常适合用常规格式显示一个模型（例如在一个表格的一行中显示模型的每个属性）。 这里说的模型可以是 `yii\base\Model` 或者其子类的一个实例，例如子类 active record，也可以是一个关联数组。

DetailView使用 `$attributes` 属性来决定显示模型哪些属性以及如何格式化。 可用的格式化选项，见 formatter section 章节。

一个典型的DetailView的使用方法如下：

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',                                          // title
        attribute (in plain text)
        'description:html',                               // description
        attribute formatted as HTML
        [                                                 // the owner name
        of the model
            'label' => 'Owner',
            'value' => $model->owner->name,
            'contentOptions' => ['class' => 'bg-red'],    // HTML
            attributes to customize value tag
            'captionOptions' => ['tooltip' => 'Tooltip'], // HTML
            attributes to customize label tag
        ],
        'created_at:datetime',                            // creation date
        formatted as datetime
    ],
]);
```

请记住，与处理一组模型的 yii\widgets\GridView 不同， DetailView 只处理一个。 因为 $model 是唯一一个用于显示的模型，并且可以作为变量在视图中使用。

但是有些情况下可以使闭包有用。 例如指定了 visible，并且你不想让value 的结果为 false:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        [
            'attribute' => 'owner',
            'value' => function ($model) {
                return $model->owner->name;
            },
            'visible' => \Yii::$app->user->can('posts.owner.view'),
        ],
    ],
]);
```

8.5.2 ListView

ListView 小部件用于显示数据提供者 data provider 提供的数据。 每个数据模型用指定的视图文件 view file 来渲染。 因为它提供开箱即用式的（译者注：封装好的）分页、排序以及过滤这样一些特性， 所以它可以很方便地为最终用户显示信息并同时创建数据管理界面。

一个典型的用法如下例所示:

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);
```

_post 视图文件可包含如下代码:

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="post">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>
```

在上面的视图文件中，当前的数据模型 `$model` 是可用的。另外，下面的这些变量也是可用的：

- `$key`：混合类型，键的值与数据项相关联。
- `$index`：整型，是由数据提供者返回的数组中以0起始的数据项的索引。
- `$widget`：类型是ListView，是小部件的实例。

假如你需要传递附加数据到每一个视图中，你可以像下面这样用 `$viewParams` 属性传递键值对：

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

在视图中，上述这些附加数据也是可以作为变量来使用的。

8.5.3 GridView

数据网格或者说 GridView 小部件是Yii中最强大的部件之一。如果你需要快速建立系统的管理后台， GridView 非常有用。它从数据提供者 data provider 中取得数据并使用 `columns` 属性的一组列配置，在一个表格中渲染每一行数据。

表中的每一行代表一个数据项的数据，并且一列通常表示该项的属性（某些列可以对应于属性或静态文本的复杂表达式）。

使用GridView的最少代码如下：

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

上面的代码首先创建了一个数据提供者，然后使用GridView显示每一行的每个属性，每一行的数据是从数据提供者取来的。 展现出来的表格封装了排序以及分页功能。

表格列

表格的列是通过 yii\grid\Column 类来配置的，这个类是通过 GridView 配置项中的 columns 属性配置的。根据列的类别和设置的不同，各列能够以不同方式展示数据。 默认的列类是 yii\grid\DataColumn，用于展现模型的某个属性， 并且可以排序和过滤。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // 数据提供者中所含数据所定义的简单的列
        // 使用的是模型的列的数据
        'id',
        'username',
        // 更复杂的列数据
        [
            'class' => 'yii\grid\DataColumn', //由于是默认类型，可以省略
            'value' => function ($data) {
                return $data->name; // 如果是数组数据则为 $data['name'] ，例
                如，使用 SqlDataProvider 的情形。
            },
        ],
    ],
]);
```

请注意，假如配置中没有指定 columns 属性， 那么 Yii 会试图显示数据提供者的模型中所有可能的列。

列类

通过使用不同类，网格列可以自定义：

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- 这里
            // 你还可以在此配置其他属性
        ],
```

除了我们下面将要展开讨论的Yii自带的列类，你还可以创建你自己的列类。

每个列类是从 yii\grid\Column 扩展而来， 从而在配置网格列的时候，你可以设置一些公共的选项。

- header 允许为头部行设置内容。
- footer 允许为尾部行设置内容。
- visible 定义某个列是否可见。
- content 允许你传递一个有效的PHP回调来为一行返回数据，格式如下：

```
function ($model, $key, $index, $column) {
    return 'a string';
}
```

你可以传递数组来指定各种容器式的HTML选项:

- headerOptions
- footerOptions
- filterOptions
- contentOptions

数据列 Data column 用于显示和排序数据。这是默认的列的类型，所以在使用 DataColumn 为列类时，可省略类的指定（译者注：不需要'class'选项的意思）。

数据列的主要配置项是 format 属性。它的值对应于 formatter application component 应用组件里面的一些方法，默认是使用 Formatter 应用组件:

```
echo GridView::widget([
    'columns' => [
        [
            'attribute' => 'name',
            'format' => 'text'
        ],
        [
            'attribute' => 'birthday',
            'format' => ['date', 'php:Y-m-d']
        ],
        'created_at:datetime', // shortcut format
        [
            'label' => 'Education',
            'attribute' => 'education',
            'filter' => ['0' => 'Elementary', '1' => 'Secondary', '2' =>
            'Higher'],
            'filterInputOptions' => ['prompt' => 'All educations', 'class'
            => 'form-control', 'id' => null]
        ],
    ],
]);
```

在上述代码中，text 对应于 yii\i18n\Formatter::asText()。列的值作为第一个参数传递。在第二列的定义中，date 对应于 yii\i18n\Formatter::asDate()。该列的值再次作为第一个参数传递同时 'php:Y-m-d' 被用作第二个参数。

有关可用格式化程序的列表，请参阅 关于数据格式的部分。

对于配置数据列，还有一种快捷方式格式，请参阅 API 文档 columns。

使用 filter 和 filterInputOptions 去控制过滤器输入的 HTML。

默认情况下，列的头部有 yii\data\Sort::link() 来呈现。它还可以使用 yii\grid\Column::$header 来调整。要更改头部文本，您应该像上面的示例中那样设置 yii\grid\DataColumn::$label。默认情况下，

标签应该从数据模型中填充。更多细节请参阅 `yii\grid\DataColumn::getHeaderCellLabel()`。

动作列 `Action column` 用于显示一些动作按钮，如每一行的更新、删除操作。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\ActionColumn',
            // 您可以在此处配置其他属性
        ],
```

可配置的属性如下：

- `controller` 是应该执行这些动作的控制器ID。 如果没有设置，它将使用当前控制器。
- `template` 定义在动作列中使用的构建每个单元格的模板。 在大括号内括起来的的令牌被当做是控制器的 action 方法ID (在动作列的上下文中也称作按钮名称)。 它们将会被 `buttons` 中指定的对应按钮的关联的渲染回调函数替代。 例如，令牌 `{view}` 将被 `buttons['view']` 关联的渲染回调函数的返回结果所替换。 如果没有找到回调函数，令牌将被替换成一个空串。默认的令牌有 `{view} {update} {delete}` 。
- `buttons` 是一个按钮的渲染回调数数组。数组中的键是按钮的名字（没有花括号），并且值是对应的按钮渲染回调函数。 这些回调函数须使用下面这种原型：

  ```
  function ($url, $model, $key) {
      // return the button HTML code
  }
  ```

  在上面的代码中，`$url` 是列为按钮创建的URL，`$model`是当前要渲染的模型对象， 并且 `$key` 是在数据提供者数组中模型的键。
- `urlCreator` 是使用指定的模型信息来创建一个按钮URL的回调函数。 该回调的原型和 `yii\grid\ActionColumn::createUrl()` 是一样的。 假如这个属性没有设置，按钮的URL将使用 `yii\grid\ActionColumn::createUrl()` 来创建。
- `visibleButtons` 是控制每个按钮可见性条件的数组。 数组键是按钮名称 (没有大括号)，值是布尔值 true/false 或匿名函数。 如果在数组中没有指定按钮名称，将会按照默认的来显示。 回调必须像如下这样来使用：

  ```
  function ($model, $key, $index) {
      return $model->status === 'editable';
  }
  ```

  或者你可以传递一个布尔值：

  ```
  [
      'update' => \Yii::$app->user->can('update')
  ]
  ```

复选框列 Checkbox column 显示一个复选框列。

想要添加一个复选框到网格视图中，将它添加到 columns 的配置中，如下所示：

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        // ...
        [
            'class' => 'yii\grid\CheckboxColumn',
            // 你可以在这配置更多的属性
        ],
    ],
```

用户可点击复选框来选择网格中的一些行。被选择的行可通过调用下面的JavaScript代码来获得：

```
var keys = $('#grid').yiiGridView('getSelectedRows');
// keys 为一个由与被选行相关联的键组成的数组
```

序号列 Serial column 渲染行号，以 1 起始并自动增长。

使用方法和下面的例子一样简单：

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'], // <-- here
        // ...
```

数据排序

> **注意：** 这部分正在开发中。
>
> - https://github.com/yiisoft/yii2/issues/1576

数据过滤

为了过滤数据的 GridView 需要一个模型 model 来 从过滤表单接收数据，以及调整数据提供者的查询对象，以满足搜索条件。 使用活动记录 active records 时，通常的做法是 创建一个能够提供所需功能的搜索模型类（可以使用 Gii 来生成）。 这个类为搜索定义了验证规则并且提供了一个将会返回数据提供者对象 的 search() 方法。

为了给 Post 模型增加搜索能力，我们可以像下面的例子一样创建 PostSearch 模型：

```
<?php

namespace app\models;

use Yii;
```

```
use yii\base\Model;
use yii\data\ActiveDataProvider;

class PostSearch extends Post
{
    public function rules()
    {
        // 只有在 rules() 函数中声明的字段才可以搜索
        return [
            [['id'], 'integer'],
            [['title', 'creation_date'], 'safe'],
        ];
    }

    public function scenarios()
    {
        // 旁路在父类中实现的 scenarios() 函数
        return Model::scenarios();
    }

    public function search($params)
    {
        $query = Post::find();

        $dataProvider = new ActiveDataProvider([
            'query' => $query,
        ]);

        // 从参数的数据中加载过滤条件，并验证
        if (!($this->load($params) && $this->validate())) {
            return $dataProvider;
        }

        // 增加过滤条件来调整查询对象
        $query->andFilterWhere(['id' => $this->id]);
        $query->andFilterWhere(['like', 'title', $this->title])
              ->andFilterWhere(['like', 'creation_date',
              $this->creation_date]);

        return $dataProvider;
    }
}
```

> **提示：** 请参阅 Query Builder 尤其是 Filter Conditions 去学习如何构建过滤查询。

你可以在控制器中使用如下方法为网格视图获取数据提供者：

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search(Yii::$app->request->get());

return $this->render('myview', [
    'dataProvider' => $dataProvider,
```

```
    'searchModel' => $searchModel,
]);
```

然后你在视图中将 `$dataProvider` 和 `$searchModel` 对象分派给 GridView 小部件：

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        // ...
    ],
]);
```

单独过滤表单

大多数时候使用 GridView 标头过滤器就足够了，但是如果你需要一个单独的过滤器表单，你也可以很轻松的去添加。您可以使用以下内容创建部分视图 `_search.php`：

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model app\models\PostSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="post-search">
    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

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

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

    <div class="form-group">
        <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
        <?= Html::submitButton('Reset', ['class' => 'btn btn-default']) ?>
    </div>

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

并将其包含在 `index.php` 视图中，如下所示：

```
<?= $this->render('_search', ['model' => $searchModel]) ?>
```

> **注意：** 如果使用 Gii 生成 CRUD 代码，默认情况下会生成单独的过滤器表单（`_search.php`），但是在 `index.php` 视图中已经被注释了。取消注释就可以用了！

当您需要按字段过滤时，单独的过滤器表单很有用，这些字段不会在 GridView 中显示，也不适用于特殊筛选条件（如日期范围）。对于按日期范围过滤，我们可以将非 DB 属性 `createdFrom` 和 `createdTo` 添加到搜索模型：

```
class PostSearch extends Post
{
    /**
     * @var string
     */
    public $createdFrom;

    /**
     * @var string
     */
    public $createdTo;
}
```

在 `search()` 扩展查询条件的方法如下：

```
$query->andFilterWhere(['>=', 'creation_date', $this->createdFrom])
      ->andFilterWhere(['<=', 'creation_date', $this->createdTo]);
```

并将代表字段添加到过滤器表单：

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

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

处理关系型模型

当我们在一个网格视图中显示活动数据的时候，你可能会遇到这种情况，就是显示关联表的列的值，例如：发帖者的名字，而不是显示他的 `id`。当 `Post` 模型有一个关联的属性名（译者注：`Post` 模型中用 `hasOne` 定义 `getAuthor()` 函数）叫 `author` 并且作者模型（译者注：本例的作者模型是 `users`）有一个属性叫 `name`，那么你可以通过在 `yii\grid\GridView::$columns` 中定义属性名为 `author.name` 来处理。这时的网格视图能显示作者名了，但是默认是不支持按作者名排序和过滤的。你需要调整上个章节介绍的 `PostSearch` 模型，以添加此功能。

为了使关联列能够排序，你需要连接关系表，以及添加排序规则到数据提供者的排序组件中：

```
$query = Post::find();
$dataProvider = new ActiveDataProvider([
    'query' => $query,
```

```
]);

// 连接与 `users` 表相关联的 `author` 表
// 并将 `users` 表的别名设为 `author`
$query->joinWith(['author' => function($query) { $query->from(['author' =>
'users']); }]);
// since version 2.0.7, the above line can be simplified to
$query->joinWith('author AS author');
// 使得关联字段可以排序
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...
```

过滤也需要像上面一样调用joinWith方法。你也需要在属性和规则中定义该列，就像下面这样：

```
public function attributes()
{
    // 添加关联字段到可搜索属性集合
    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],
        [['title', 'creation_date', 'author.name'], 'safe'],
    ];
}
```

然后在 `search()` 方法中，你仅需要添加一个额外过滤条件：

```
$query->andFilterWhere(['LIKE', 'author.name',
$this->getAttribute('author.name')]);
```

> **信息：** 在上面的代码中，我们使用相同的字符串作为关联名称和表别名； 然而，当你的表别名和关联名称不相同的时候，你得注意在哪使用你的别名，在哪使用你的关联名称。 一个简单的规则是在每个构建数据库查询的地方使用别名，而在所有其他和定义相关的诸如： `attributes()` 和 `rules()` 等地方使用关联名称。
>
> 例如，你使用 `au` 作为作者关系表的别名，那么联查语句就要写成像下面这样：
>
> ```
> $query->joinWith(['author' => function($query) { $query->from(['au' =>
> 'users']); }]);
> ```

当别名已经在关联函数中定义了时，也可以只调用 `$query->joinWith(['author']);`。在过滤条件中，别名必须使用，但属性名称保持不变：

```
$query->andFilterWhere(['LIKE', 'au.name',
$this->getAttribute('author.name')]);
```

排序定义也同样如此：

```
$dataProvider->sort->attributes['author.name'] = [
     'asc' => ['au.name' => SORT_ASC],
     'desc' => ['au.name' => SORT_DESC],
];
```

同样，当指定使用 `defaultOrder` 来排序的时候， >你需要使用关联名称替代别名：

```
$dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
```

信息： 更多关于 `joinWith` 和在后台执行查询的相关信息， 可以查看 active record docs on joining with relations。

SQL 视图用于过滤、排序和显示数据 还有另外一种方法可以更快、更有用的 SQL 视图。例如，我们要在 `GridView` 中显示用户和他们的简介，可以这样创建 SQL 视图：

```
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
```

然后你需要创建活动记录模型来代表这个视图：

```
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 [
            // 在这定义你的规则
        ];
    }

    /**
     * @inheritdoc
     */
    public static function attributeLabels()
    {
        return [
            // 在这定义你的属性标签
        ];
    }


}
```

之后你可以使用这个 UserView 活动记录和搜索模型，无需附加的排序和过滤属性的规则。 所有属性都可开箱即用。请注意，这种方法有利有弊：

- 你不需要指定不同排序和过滤条件，一切都包装好了；
- 它可以更快，因为数据的大小，SQL 查询的执行（对于每个关联数据你都不需要额外的查询）都得到优化；
- 因为在 SQL 视图中这仅仅是一个简单的映射UI，所以在你的实体中，它可能缺乏某方面的逻辑，所以，假如你有一些诸如 `isActive`、`isDeleted` 或者其他影响到 UI 的方法， 你也需要在这个类中复制他们。

单个页面多个网格视图部件

你可以在一个单独页面中使用多个网格视图，但是一些额外的配置是必须的，为的就是它们相互之间不干扰。 当使用多个网格视图实例的时候，你必须要为生成的排序和分页对象配置不同的参数名， 以便于每个网格视图有它们各自独立的排序和分页。 你可以通过设置 `sortParam` 和 `pageParam`，对应于数据提供者的 `sort` 和 `pagination` 实例。

假如我们想要同时显示 `Post` 和 `User` 模型，这两个模型已经在 `$userProvider` 和 `$postProvider` 这两个数据提供者中准备好， 具体做法如下：

```
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,
]);
```

在 GridView 使用 Pjax

Pjax 允许您更新页面的某个部分， 而不是重新加载整个页面。 使用过滤器时，可以使用它仅更新 GridView 内容。

```
use yii\widgets\Pjax;
use yii\grid\GridView;

Pjax::begin([
    // PJax options
]);
    Gridview::widget([
        // GridView options
    ]);
Pjax::end();
```

Pjax 也适用于 Pjax 小部件之间的链接以及 Pjax::$linkSelector 指定的链接。 但是这可能是 ActionColumn 链接的问题。 要防止这种情况，请在编辑 ActionColumn::$buttons 属性时将 HTML 属性 data-pjax="0" 添加到链接中。

在 Gii 中使用 Pjax 的 GridView/ListView 从 2.0.5 开始，Gii 的 CRUD 生成器有一个 $enablePjax 选项， 可以通过 web 界面或者命令行使用。

```
yii gii/crud --controllerClass="backend\\controllers\PostController" \
  --modelClass="common\\models\\Post" \
  --enablePjax=1
```

这会生成一个由 Pjax 小部件包含的 GridView 或者 ListView。

8.5.4 延伸阅读

- Rendering Data in Yii 2 with GridView and ListView[8] by Arno Slatius.

[8] https://www.sitepoint.com/rendering-data-in-yii-2-with-gridview-and-listview/

8.6 客户端脚本使用

现代 Web 应用程序， 除了呈现并发送到浏览器的静态 HTML 页面外， 还包含 JavaScript， 用于通过操纵现有元素或通过 AJAX 加载新内容来修改浏览器中的页面。 本节介绍 Yii 提供的用于向网站添加 JavaScript 和 CSS 以及动态调整它们的方法。

8.6.1 注册脚本

使用 `yii\web\View` 对象时，可以动态注册前端脚本。 这里有两个专门的方法:

- `registerJs()` 用于内联脚本。
- `registerJsFile()` 用于注册引入外部脚本文件。

注册内联脚本

内联脚本对于配置，动态生成的代码以及由 widgets 中包含的可重用前端代码创建的小片段非常有用。 `registerJs()` 添加这些方法可以像如下使用:

```
$this->registerJs(
    "$('#myButton').on('click', function() { alert('Button clicked!'); });",
    View::POS_READY,
    'my-button-handler'
);
```

第一个参数是我们想插入的实际 JS 代码。 它将被包含到 `<script>` 标签中。 第二个参数确定脚本应插入页面的位置。 可能的值是:

- `View::POS_HEAD` 用在 HEAD 部分。
- `View::POS_BEGIN` 用在 `<body>` 标签的右边。
- `View::POS_END` 用在 `</body>` 标签的左边。
- `View::POS_READY` 为了在 `ready` 事件[9] 中执行代码。 这里将自动注册 `jQuery` 并将代码包装到适当的 jQuery 代码中。这是默认位置。
- `View::POS_LOAD` 为了在 `load` 事件[10] 中执行代码。 与上面相同，这也将自动注册 `jQuery`。

最后一个参数是一个唯一的脚本 ID，主要是用于标识一段代码块，在添加一段新的代码块时，如果当前页面已经存在同样 ID 代码块时，那么将会被新的替换。 如果你不传这个参数，JS 代码本身将会作为 ID 来使用。

注册脚本文件

`registerJsFile()` 的参数类似于 `registerCssFile()`。在以下示例中， 我们注册 `main.js` 文件，依赖于 `yii\web\JqueryAsset`。这意味着 `main.js` 文件将在 `jquery.js` 之后添加。没有这样的依赖规范， `main.js` 和 `jquery.js` 之间的相对顺序将是未定义的，代码将不起作用。

[9] https://learn.jquery.com/using-jquery-core/document-ready/
[10] https://learn.jquery.com/using-jquery-core/document-ready/

外部脚本的引入使用像下面这样：

```
$this->registerJsFile(
    '@web/js/main.js',
    ['depends' => [\yii\web\JqueryAsset::class]]
);
```

这将为 base URL 下的 `/js/main.js` 脚本添加一个标记。

我们强烈建议使用 asset bundles 来注册外部 JS 文件而非使用 `registerJsFile()` 来注册。 因为这些允许更好的灵活性和更精细的依赖配置。 此外，使用资源包允许您组合和压缩多个 JS 文件，这对于高流量网站来说是比较理想的方式。

8.6.2 注册 CSS

与 JavaScript 类似，您可以使用注册 CSS 使用 `registerCss()` 或 `registerCssFile()`。前者注册一个 CSS 代码块，而后者注册一个外部 CSS 文件。

注册内联 CSS

```
$this->registerCss("body { background: #f00; }");
```

上面的代码将导致将以下内容添加到页面的 `<head>` 部分：

```
<style>
body { background: #f00; }
</style>
```

如果要指定样式标记的其他属性，请将 name-values 数组传递给第二个参数。 最后一个参数是一个唯一的 ID，用于标识样式块，并确保仅在代码中的不同位置注册相同样式时仅添加一次。

注册 CSS 文件

可以使用以下方法注册 CSS 文件：

```
$this->registerCssFile("@web/css/themes/black-and-white.css", [
    'depends' => [\yii\bootstrap\BootstrapAsset::class],
    'media' => 'print',
], 'css-print-theme');
```

上面的代码将把 `/css/themes/black-and-white.css` 文件的链接添加到页面的 `<head>` 部分。

- 第一个参数指明被注册的 CSS 文件。
- 第二个参数指明 `<link>` 标签的 HTML 属性，选项 `depends` 是专门处理 指明 CSS 文件依赖于哪个资源包。在这种情况下，依赖资源包就是 `yii\bootstrap\BootstrapAsset`。这意味着 CSS 文件将 被添加在 `yii\bootstrap\BootstrapAsset` 之后。

- 最后一个参数指明一个 ID 来标识这个 CSS 文件。 如果参数未提供，则将使用 CSS 文件的 URL。

我们强烈建议使用 asset bundles 来注册外部 CSS 文件， 而非使用 `registerCssFile()` 来注册。 使用资源包允许你合并并且压缩多个 CSS 文件，对于高流量的网站来说，这是比较理想的方式。 它还提供了更大的灵活性，因为应用程序的所有资源依赖性都在一个位置配置。

8.6.3 注册资源包

如前所述，建议使用资源包而不是直接注册 CSS 和 JavaScript 文件。 您可以获取有关如何定义资源包的详细信息在 "Assets" 部分. 至于使用已定义的资源包，它非常简单：

```
\frontend\assets\AppAsset::register($this);
```

在上面的代码中，在视图文件的上下文中，`AppAsset` 包在当前视图上注册（由 `$this` 表示）。 从小部件中注册资产包时，您将传递小部件的 `$view` 来替代（`$this->view`）。

8.6.4 生成动态 Javascript

在视图文件中，HTML 代码通常不直接写出， 而是由某些依赖于视图变量的 PHP 代码生成。 为了使用 Javascript 操作生成的 HTML，JS 代码也必须包含动态部分，例如 jQuery 选择器的 ID。

要将PHP变量插入 JS 代码， 必须正确转义它们的值。 特别是当 JS 代码插入 HTML 而不是驻留在专用的 JS 文件中时。 为此，Yii 提供了 `Json` 助手类的 `htmlEncode()` 方法，其用法将在以下示例中显示。

注册全局 JavaScript 配置

在这个例子中， 我们使用一个数组将全局配置参数从应用程序的 PHP 部分传递给 JS 前端代码。

```
$options = [
    'appName' => Yii::$app->name,
    'baseUrl' => Yii::$app->request->baseUrl,
    'language' => Yii::$app->language,
    // ...
];
$this->registerJs(
    "var yiiOptions = ".\yii\helpers\Json::htmlEncode($options).";",
    View::POS_HEAD,
    'yiiOptions'
);
```

上面的代码将注册一个包含 JavaScript 的 `<script>` 标签定义， 例如：

```
var yiiOptions = {"appName":"My Yii
Application","baseUrl":"/basic/web","language":"en"};
```

在您的 JavaScript 代码中，您现在可以像 `yiiOptions.baseUrl` 或 `yiiOptions.language` 一样访问它们。

传递翻译的消息

您可能会遇到 JavaScript 需要打印消息以响应某些事件的情况。在使用多种语言的应用程序中，此字符串必须转换为当前的应用程序语言。 实现此目的的一种方法是使用 Yii 提供的 消息转换功能 并将结果传递给 JavaScript 代码。

```
$message = \yii\helpers\Json::htmlEncode(
    \Yii::t('app', 'Button clicked!')
);
$this->registerJs(<<<JS
    $('#myButton').on('click', function() { alert( $message ); });
JS
);
```

上面的示例代码使用 PHP Heredoc 语法[11] 以获得更好的可读性。 这也可以在大多数 IDE 中实现更好的语法突出显示，因此它是编写内联 JavaScript 的首选方式，对于长于单行的代码尤其有用。变量 `$message` 是在 PHP 中创建的，感谢 `Json::htmlEncode` 它包含有效 JS 语法中的字符串，可以将其插入到 JavaScript 代码中以放置 函数中的动态字符串调用 `alert()`。

> **注意：** 使用 Heredoc 时，请注意 JS 代码中的变量命名， 因为以 `$` 开头的变量可能会被解释为 PHP 变量， 这些变量将被其内容替换。 jQuery 函数以 `$(` 或 `$.` 的形式不被解释为 PHP 变量， 可以安全地使用。

8.6.5 脚本

> **注意：** 本节尚未编写。 它应该包含 `yii.js` 提供的功能的说明：
>
> - Yii JavaScript 模块
> - CSRF 参数处理
> - `data-confirm` 处理
> - `data-method` 处理
> - 脚本过滤
> - 重定向处理

8.7 主题

主题是一种将当前的一套视图 views 替换为另一套视图，而无需更改视图渲染代码的方法。 你可以使用主题来系统地更改应用的外观和体验。

[11] https://www.php.net/manual/zh/language.types.string.php#language.types.string.syntax.heredoc

要使用主题，你得配置 `view` 应用组件的 `theme` 属性。 这个属性配置了一个 `yii\base\Theme` 对象，这个对象用来控制视图文件怎样被替换。 你主要应该指明下面的 `yii\base\Theme` 属性：

- `yii\base\Theme::$basePath`：指定包含主题资源（CSS, JS, images, 等等）的基准目录。
- `yii\base\Theme::$baseUrl`：指定主题资源的基准URL。
- `yii\base\Theme::$pathMap`：指定视图文件的替换规则。 更多细节将在下面介绍。

例如，如果你在 `SiteController` 里面调用 `$this->render('about')`，那你将渲染 视图文件 `@app/views/site/about.php` 。然而，如果你在下面的应用配置中开启了主 题功能，那么 `@app/themes/basic/site/about.php` 文件将会被渲染。

```
return [
    'components' => [
        'view' => [
            'theme' => [
                'basePath' => '@app/themes/basic',
                'baseUrl' => '@web/themes/basic',
                'pathMap' => [
                    '@app/views' => '@app/themes/basic',
                ],
            ],
        ],
    ],
];
```

> **信息：** 主题支持路径别名。当我们在做视图替换的时候， 路径别名将被转换成实际的文件路径或者URL。

你可以通过 `yii\base\View::$theme` 属性访问 `yii\base\Theme` 对象。例如，在一个视图文件里，你可以写下面的代码， 因为 `$this` 指向视图对象：

```
$theme = $this->theme;

// returns: $theme->baseUrl . '/img/logo.gif'
$url = $theme->getUrl('img/logo.gif');

// returns: $theme->basePath . '/img/logo.gif'
$file = $theme->getPath('img/logo.gif');
```

`yii\base\Theme::$pathMap` 属性控制如何替换视图文件。它是一个键值对数组，其中， 键是原本的视图路径，而值是相应的主题视图路径。 替换是基于部分匹配的：如果视图路径以 `pathMap` 数组的 任何一个键为起始，那么匹配部分将被相应的值所替换。 使用上面配置的例子，因为 `@app/views/site/about.php` 中的起始部分与键 `@app/views` 匹配， 它将被替换成 `@app/themes/basic/site/about.php`。

8.7.1 主题化模块

要主题化模块，yii\base\Theme::$pathMap 可以配置成下面这样：

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/modules' => '@app/themes/basic/modules', // <-- !!!
],
```

它允许你将 @app/modules/blog/views/comment/index.php 主题化成 @app/themes/basic/modules/blog/views/commen

8.7.2 主题化小部件

要主题化小部件，你可以像下面这样配置 yii\base\Theme::$pathMap：

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/widgets' => '@app/themes/basic/widgets', // <-- !!!
],
```

这将允许你将 @app/widgets/currency/views/index.php 主题化成 @app/themes/basic/widgets/currency/index.php

8.7.3 主题继承

有的时候，你可能想要定义一个基本的主题，其中包含一个基本的应用外观和体验，然后根据当前的节日，你可能想要稍微地改变一下外观和体验。 这个时候，你就可以使用主题继承实现这一目标，主题继承是通过一个单视图路径去映射多个目标， 例如，

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
]
```

在这种情况下，视图 @app/views/site/index.php 将被主题化成 @app/themes/christmas/site/index.php 或者 @app/themes/basic/site/index.php， 这取决于哪个主题文件存在。假如都存在，那么第一个将被优先使用。在现实情况中， 你会将大部分的主题文件放在 @app/themes/basic 里，而一些自定义的放在 @app/themes/christmas里。

Chapter 9

安全（Security）

9.1 安全（Security）

良好的安全性对于任何应用的健康和成功都至关重要。 不幸的是，许多开发人员在安全方面偷工减料，要么是由于缺乏理解，要么是由于实现过于困难。 为了尽可能保证您的 Yii 应用程序的安全，Yii 包含了一些优秀且易于使用的安全功能。

- 认证
- 授权
- 使用密码
- 加密
- 视图安全
- 身份验证客户端[1]
- 最佳实践
- 受信任的代理和头文件

9.2 认证

认证是鉴定用户身份的过程。它通常使用一个标识符 （如用户名或电子邮件地址）和一个加密令牌（比如密码或者存取令牌）来 鉴别用户身份。认证是登录功能的基础。

Yii提供了一个认证框架，它连接了不同的组件以支持登录。欲使用这个框架， 你主要需要做以下工作：

- 设置用户组件 `user` ；
- 创建一个类实现 `yii\web\IdentityInterface` 接口。

9.2.1 配置 `yii\web\User`

用户组件 `user` 用来管理用户的认证状态。这需要你 指定一个含有实际认

[1] https://github.com/yiisoft/yii2-authclient/blob/master/docs/guide/README.md

证逻辑的认证类 identity class。 在以下web应用的配置项中，将用户用户组件 user 的 认证类 identity class 配置成 模型类 app\models\User，它的实现将在下一节中讲述。

```
return [
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
        ],
    ],
];
```

9.2.2 认证接口 yii\web\IdentityInterface 的实现

认证类 identity class 必须实现包含以下方法的 认证接口 yii\web\IdentityInterface：

- findIdentity()：根据指定的用户ID查找 认证模型类的实例，当你需要使用session来维持登录状态的时候会用到这个方法。
- findIdentityByAccessToken()：根据指定的存取令牌查找 认证模型类的实例，该方法用于 通过单个加密令牌认证用户的时候（比如无状态的RESTful应用）。
- getId()：获取该认证实例表示的用户的ID。
- getAuthKey()：获取基于 cookie 登录时使用的认证密钥。 认证密钥储存在 cookie 里并且将来会与服务端的版本进行比较以确保 cookie的有效性。
- validateAuthKey() ：是基于 cookie 登录密钥的 验证的逻辑的实现。

用不到的方法可以空着，例如，你的项目只是一个 无状态的 RESTful 应用，只需实现 findIdentityByAccessToken() 和 getId() ，其他的方法的函数体留空即可。

下面的例子是一个通过结合了 user 数据表的 AR 模型 Active Record 实现的一个认证类 identity class。

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * 根据给到的ID查询身份。
     *
     * @param string|integer $id 被查询的ID
```

```
     * @return IdentityInterface|null 通过ID匹配到的身份对象
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * 根据 token 查询身份。
     *
     * @param string $token 被查询的 token
     * @return IdentityInterface|null 通过 token 得到的身份对象
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string 当前用户ID
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string 当前用户的（cookie）认证密钥
     */
    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;
    }
}
```

如上所述，如果你的应用利用 cookie 登录， 你只需要实现 `getAuthKey()` 和 `validateAuthKey()` 方法。这样的话，你可以使用下面的代码在 `user` 表中生成和存储每个用户的认证密钥。

```
class User extends ActiveRecord implements IdentityInterface
{
    ......

    public function beforeSave($insert)
```

```
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
                $this->auth_key =
                \Yii::$app->security->generateRandomString();
            }
            return true;
        }
        return false;
    }
}
```

> **注意：** 不要混淆 `user` 认证类和用户组件 `yii\web\User`。前者是实现 认证逻辑的类，通常用关联了 持久性存储的用户信息的AR模型 Active Record 实现。后者是负责管理用户认证状态的 应用组件。

9.2.3 使用用户组件 `yii\web\User`

在 `user` 应用组件方面，你主要用到 `yii\web\User` 。

你可以使用表达式 `Yii::$app->user->identity` 检测当前用户身份。它返回 一个表示当前登录用户的认证类 `identity class` 的实例， 未认证用户（游客）则返回 null。下面的代码展示了如何从 `yii\web\User` 获取其他认证相关信息:

```
// 当前用户的身份实例。未认证用户则为 Null 。
$identity = Yii::$app->user->identity;

// 当前用户的ID。 未认证用户则为 Null 。
$id = Yii::$app->user->id;

// 判断当前用户是否是游客（未认证的）
$isGuest = Yii::$app->user->isGuest;
```

你可以使用下面的代码登录用户:

```
// 使用指定用户名获取用户身份实例。
// 请注意，如果需要的话您可能要检验密码
$identity = User::findOne(['username' => $username]);

// 登录用户
Yii::$app->user->login($identity);
```

`yii\web\User::login()` 方法将当前用户的身份登记到 `yii\web\User`。如果 session 设置为 `enabled`，则使用 session 记录用户身份，用户的 认证状态将在整个会话中得以维持。如果开启自动登录 `enabled` 则基于 cookie 登录（如：记住登录状态），它将使用 cookie 保存用户身份，这样 只要 cookie 有效就可以恢复登录状态。

为了使用 cookie 登录，你需要在应用配置文件中将 `yii\web\User::$enableAutoLogin` 设为 true。你还需要在 `yii\web\User::login()` 方法中 传递有效期（记住登录状态的时长）参数。

注销用户：

```
Yii::$app->user->logout();
```

请注意，启用 session 时注销用户才有意义。该方法将从内存和 session 中同时清理用户认证状态。默认情况下，它还会注销所有的 用户会话数据。如果你希望保留这些会话数据，可以换成 `Yii::$app->user->logout(false)` 。

9.2.4 认证事件

`yii\web\User` 类在登录和注销流程引发一些事件。

- `EVENT_BEFORE_LOGIN`：在登录 `yii\web\User::login()` 时引发。 如果事件句柄将事件对象的 `isValid` 属性设为 false， 登录流程将会被取消。
- `EVENT_AFTER_LOGIN`：登录成功后引发。
- `EVENT_BEFORE_LOGOUT`：注销 `yii\web\User::logout()` 前引发。 如果事件句柄将事件对象的 `isValid` 属性设为 false， 注销流程将会被取消。
- `EVENT_AFTER_LOGOUT`：成功注销后引发。

你可以通过响应这些事件来实现一些类似登录统计、在线人数统计的功能。例如, 在登录后 `EVENT_AFTER_LOGIN` 的处理程序，你可以将用户的登录时间和IP记录到 `user` 表中。

9.3 授权

授权是指验证用户是否允许做某件事的过程。Yii提供两种授权方法： 存取控制过滤器（ACF）和基于角色的存取控制（RBAC）。

9.3.1 存取控制过滤器

存取控制过滤器（ACF）是一种通过 `yii\filters\AccessControl` 类来实现的简单授权方法， 非常适用于仅需要简单的存取控制的应用。正如其名称所指，ACF 是一种动作过滤器 filter，可在控制器或者模块中使用。当一个用户请求一个动作时， ACF会检查 `access rules` 列表，判断该用户是否允许执 行所请求的动作。

下述代码展示如何在 `site` 控制器中使用 ACF：

```
use yii\web\Controller;
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
```

```
    {
        return [
            'access' => [
                'class' => AccessControl::class,
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
                    ],
                    [
                        'allow' => true,
                        'actions' => ['logout'],
                        'roles' => ['@'],
                    ],
                ],
            ],
        ];
    }
    // ...
}
```

上面的代码中 ACF 以行为 (behavior) 的形式附加到 `site` 控制器。 这就是很典型的使用行动过滤器的方法。 `only` 选项指明 ACF 应当只 对 `login`, `logout` 和 `signup` 方法起作用。所有其它的 `site` 控制器中的方法不受存取控制的限制。 `rules` 选项列出了 存取规则 (access rules)，解读如下：

- 允许所有访客（还未经认证的用户）执行 `login` 和 `signup` 动作。 `roles` 选项包含的问号 `?` 是一个特殊的标识，代表"访客用户"。
- 允许已认证用户执行 `logout` 动作。`@`是另一个特殊标识， 代表"已认证用户"。

ACF 自顶向下逐一检查存取规则，直到找到一个与当前 欲执行的动作相符的规则。 然后该匹配规则中的 `allow` 选项的值用于判定该用户是否获得授权。如果没有找到匹配的规则， 意味着该用户没有获得授权。（译者注：`only` 中没有列出的动作，将无条件获得授权）

当 ACF 判定一个用户没有获得执行当前动作的授权时，它的默认处理是：

- 如果该用户是访客，将调用 `yii\web\User::loginRequired()` 将用户的浏览器重定向到登录页面。
- 如果该用户是已认证用户，将抛出一个 `yii\web\ForbiddenHttpException` 异常。

你可以通过配置 `yii\filters\AccessControl::$denyCallback` 属性定制该行为：

```
[
    'class' => AccessControl::class,
    ...
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('You are not allowed to access this page');
```

```
    }
]
```

Access rules 支持很多的选项。下列是所支持选项的总览。 你可以派生 yii\filters\AccessRule 来创建自定义的存取规则类。

- allow: 指定该规则是 "允许" 还是 "拒绝" 。（译者注：true是允许，false是拒绝）
- actions: 指定该规则用于匹配哪些动作。 它的值应该是动作方法的ID数组。匹配比较是大小写敏感的。如果该选项为空，或者不使用该选项， 意味着当前规则适用于所有的动作。
- controllers: 指定该规则用于匹配哪些控制器。 它的值应为控制器ID数组。匹配比较是大小写敏感的。如果该选项为空，或者不使用该选项， 则意味着当前规则适用于所有的动作。（译者注：这个选项一般是在控制器的自定义父类中使用才有意义）
- roles: 指定该规则用于匹配哪些用户角色。 系统自带两个特殊的角色，通过 yii\web\User::$isGuest 来判断：
  - ?: 用于匹配访客用户 （未经认证）
  - @: 用于匹配已认证用户

  使用其他角色名时，将触发调用 yii\web\User::can()，这时要求 RBAC 的支持 （在下一节中阐述）。 如果该选项为空或者不使用该选项，意味着该规则适用于所有角色。
- roleParams: 指定将传递给 yii\web\User::can() 的参数。 请参阅下面描述RBAC规则的部分，了解如何使用它。 如果此选项为空或未设置，则不传递任何参数。
- ips: 指定该规则用于匹配哪些 客户端IP地址 。 IP 地址可在其末尾包含通配符 * 以匹配一批前缀相同的IP地址。 例如，192.168.* 匹配所有 192.168. 段的IP地址。 如果该选项为空或者不使用该选项，意味着该规则适用于所有角色。
- verbs: 指定该规则用于匹配哪种请求方法（例如GET，POST）。 这里的匹配大小写不敏感。
- matchCallback: 指定一个PHP回调函数用于 判定该规则是否满足条件。（译者注：此处的回调函数是匿名函数）
- denyCallback: 指定一个PHP回调函数， 当这个规则不满足条件时该函数会被调用。（译者注：此处的回调函数是匿名函数）

以下例子展示了如何使用 matchCallback 选项， 可使你设计任意的访问权限检查逻辑：

```php
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::class,
```

```
                'only' => ['special-callback'],
                'rules' => [
                    [
                        'actions' => ['special-callback'],
                        'allow' => true,
                        'matchCallback' => function ($rule, $action) {
                            return date('d-m') === '31-10';
                        }
                    ],
                ],
            ],
        ];
    }

    // 匹配的回调函数被调用了！这个页面只有每年的10月31号能访问（译者注：原文在这里说该方法是回调函数不确切，读者不要和 `matchCallback` 的值即匿名的回调函数混淆理解）。
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

9.3.2 基于角色的存取控制 （RBAC）

基于角色的存取控制 （RBAC） 提供了一个简单而强大的集中式存取控制机制。 详细的关于 RBAC 和诸多传统的存取控制方案对比的详情，请参阅 Wikipedia[2]。

Yii 实现了通用的分层的 RBAC，遵循的模型是 NIST RBAC model[3]. 它通过 `authManager` application component 提供 RBAC 功能。

使用 RBAC 涉及到两部分工作。第一部分是建立授权数据， 而第二部分是使用这些授权数据在需要的地方执行检查。

为方便后面的讲述，这里先介绍一些 RBAC 的基本概念。

基本概念

角色是 权限 的集合 （例如：建贴、改贴）。一个角色 可以指派给一个或者多个用户。要检查某用户是否有一个特定的权限， 系统会检查该包含该权限的角色是否指派给了该用户。

可以用一个规则 *rule* 与一个角色或者权限关联。一个规则用一段代码代表， 规则的执行是在检查一个用户是否满足这个角色或者权限时进行的。例如，”改帖” 的权限 可以使用一个检查该用户是否是帖子的创建者的规则。权限检查中，如果该用户 不是帖子创建者，那么他（她）将被认为不具有 “改帖”的权限。

[2] https://zh.wikipedia.org/wiki/%E4%BB%A5%E8%A7%92%E8%89%B2%E7%82%BA%E5%9F%BA%E7%A4%8E%E7%9A%84%E5%AD%98%E5%8F%96%E6%8E%A7%E5%88%B6

[3] https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf

角色和权限都可以按层次组织。特定情况下，一个角色可能由其他角色或权限构成， 而权限又由其他的权限构成。Yii 实现了所谓的 局部顺序 的层次结构，包含更多的特定的 树 的层次。 一个角色可以包含一个权限，反之则不行。（译者注：可理解为角色在上方，权限在下方，从上到下如果碰到权限那么再往下不能出现角色）

配置 RBAC

在开始定义授权数据和执行存取检查之前，需要先配置应用组件 `authManager` 。 Yii 提供了两套授权管理器： `yii\rbac\PhpManager` 和 `yii\rbac\DbManager`。前者使用 PHP 脚本存放授权数据， 而后者使用数据库存放授权数据。 如果你的应用不要求大量的动态角色和权限管理， 你可以考虑使用前者。

使用 以下代码展示使用 `yii\rbac\PhpManager` 时如何在应用配置文件中配置 `authManager`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

现在可以通过 `\Yii::$app->authManager` 访问 `authManager` 。

`yii\rbac\PhpManager` 默认将 RBAC 数据保存在 `@app/rbac` 目录下的文件中。 如果权限层次数据在运行时会被修改，需确保WEB服务器进程对该目录和其中的文件有写权限。

使用 以下代码展示使用 `yii\rbac\DbManager` 时如何在应用配置文件中配置 `authManager`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
            // uncomment if you want to cache RBAC items hierarchy
            // 'cache' => 'cache',
        ],
        // ...
    ],
];
```

> **注意：** 如果您使用的是 yii2-basic-app 模板，则有一个 `config/console.php` 配置文件，其中 `authManager` 需要另外声明在 `config/web.php`。

在 yii2-advanced-app 的情况下，`authManager` 只能在 `common/config/main.php` 中声明一次。

`DbManager` 使用4个数据库表存放它的数据：

- `itemTable`： 该表存放授权条目（译者注：即角色和权限）。默认表名为 "auth_item"。
- `itemChildTable`： 该表存放授权条目的层次关系。默认表名为 "auth_item_child"。
- `assignmentTable`： 该表存放授权条目对用户的指派情况。默认表名为 "auth_assignment"。
- `ruleTable`： 该表存放规则。默认表名为 "auth_rule"。

继续之前，你需要在数据库中创建这些表。你可以使用存放在 `@yii/rbac/migrations` 目录中的数据库迁移文件来做这件事（译者注：根据本人经验，最好是将授权数据初始化命令也写到这个 RBAC 数据库迁移文件中）：

```
yii migrate --migrationPath=@yii/rbac/migrations
```

阅读 Separated Migrations 部分中有关处理来自不同名称空间的迁移的更多信息。

现在可以通过 `\Yii::$app->authManager` 访问 `authManager` 。

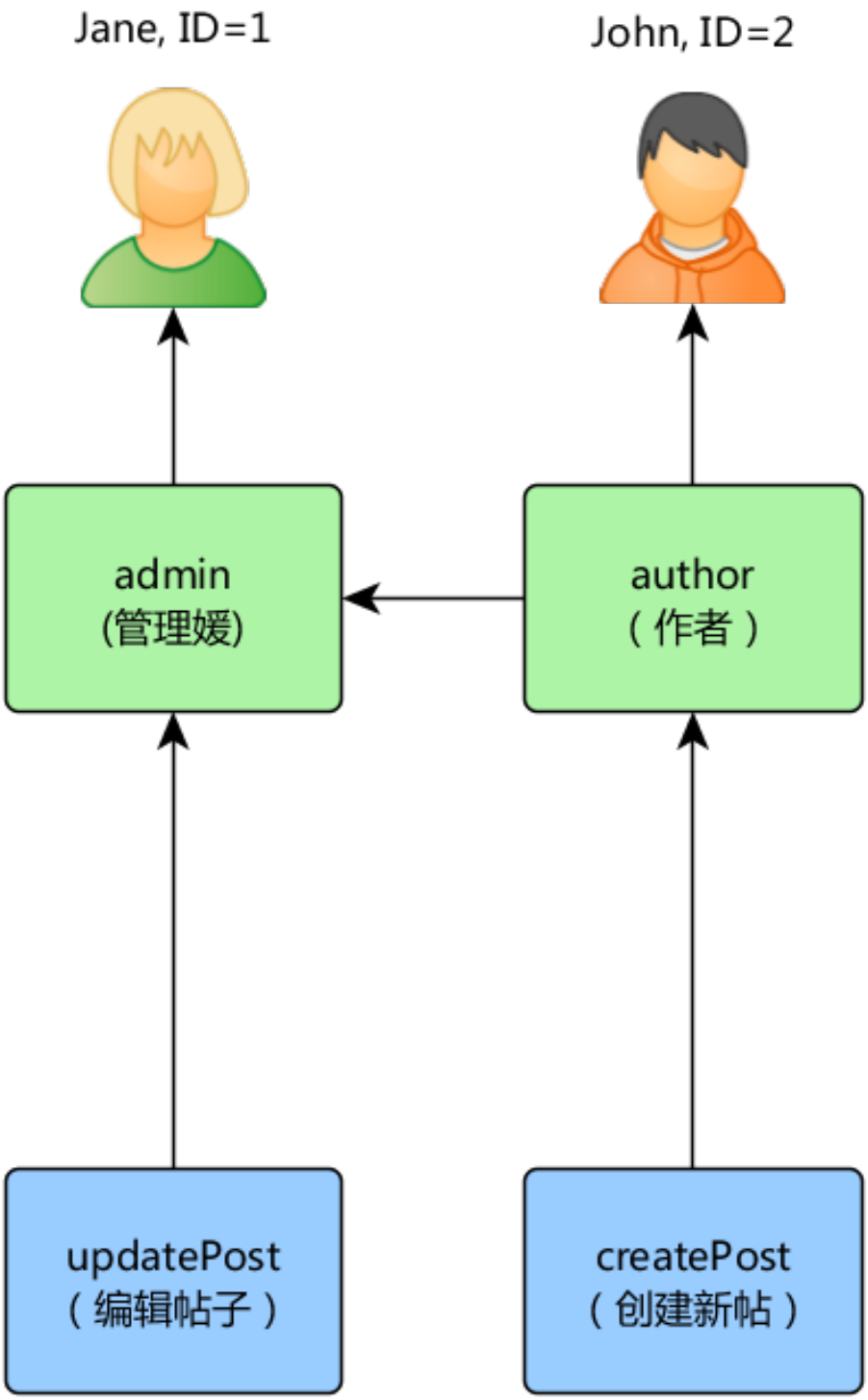
建立授权数据

所有授权数据相关的任务如下：

- 定义角色和权限；
- 建立角色和权限的关系；
- 定义规则；
- 将规则与角色和权限作关联；
- 指派角色给用户。

根据授权灵活性要求，上述任务可以通过不同的方式完成。 如果您的权限层次结构仅由开发人员更改，则可以使用迁移或控制台命令。 Migration pro 是可以与其他迁移一起执行的。 控制台命令 pro 是您可以很好地了解代码中的层次结构， 而不是分散在多个迁移中。

无论哪种方式，最终都会得到以下 RBAC 层次结构：



如果您需要动态形成权限层次结构，则需要 UI 或控制台命令。 用于构

建层次结构的 API 本身不会有所不同。

使用迁移（Using migrations） 您可以使用 migrations 通过 `authManager` 提供的 API 初始化和修改层次结构。

使用 `./yii migrate/create init_rbac` 创建新迁移，然后实现创建层次结构：

```
<?php
use yii\db\Migration;

class m170124_084304_init_rbac extends Migration
{
    public function up()
    {
        $auth = Yii::$app->authManager;

        // 添加 "createPost" 权限
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Create a post';
        $auth->add($createPost);

        // 添加 "updatePost" 权限
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Update post';
        $auth->add($updatePost);

        // 添加 "author" 角色并赋予 "createPost" 权限
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // 添加 "admin" 角色并赋予 "updatePost"
                // 和 "author" 权限
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // 为用户指派角色。其中 1 和 2 是由 IdentityInterface::getId() 返回
        的id
        // 通常在你的 User 模型中实现这个函数。
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }

    public function down()
    {
        $auth = Yii::$app->authManager;

        $auth->removeAll();
    }
}
```

> 如果您不想硬编码哪些用户具有某些角色，请不要在迁移中使用 `->assign()` 调用。 相反，请创建UI或控制台命令来管理分配。

迁移可以通过 `yii migrate` 使用。

使用控制台命令（Using console command）

如果您的权限层次根本没有改变，并且您拥有固定数量的用户， 则可以创建一个控制台命令，它将通过 `authManager` 提供的 API 初始化授权数据一次：

```
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;
        $auth->removeAll();

        // 添加 "createPost" 权限
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Create a post';
        $auth->add($createPost);

        // 添加 "updatePost" 权限
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Update post';
        $auth->add($updatePost);

        // 添加 "author" 角色并赋予 "createPost" 权限
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // 添加 "admin" 角色并赋予 "updatePost"
                // 和 "author" 权限
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // 为用户指派角色。其中 1 和 2 是由 IdentityInterface::getId() 返回
        的id
        // 通常在你的 User 模型中实现这个函数。
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
```

```
    }
}
```

> **注意：** 如果您使用高级模板，则需要将 `RbacController` 放在 `console/controllers` 目录中，并将命名空间更改为 `console/controllers`。

上面的命令可以通过以下方式从控制台执行：

```
yii rbac/init
```

> 如果您不想硬编码用户具有某些角色，请不要将 `->assign()` 调用放入命令中。相反，请创建UI或控制台命令来管理分配。

9.3.3 为用户分配角色（Assigning roles to users）

作者可以创建帖子，管理员可以更新帖子并可以做一切作者都能做的。

如果您的应用程序允许用户注册，则需要为这些新用户分配一次角色。例如，为了让所有注册用户成为高级项目模板中的作者，您需要修改 `frontend\models\SignupForm::signup()`，如下所示：

```php
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);

        // 增加了以下三行：
        $auth = \Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

对于需要动态更新授权数据的复杂访问控制的应用程序，可能需要使用 `authManager` 提供的 API 来开发特殊的用户界面（即管理面板）。

使用规则 (Rules)

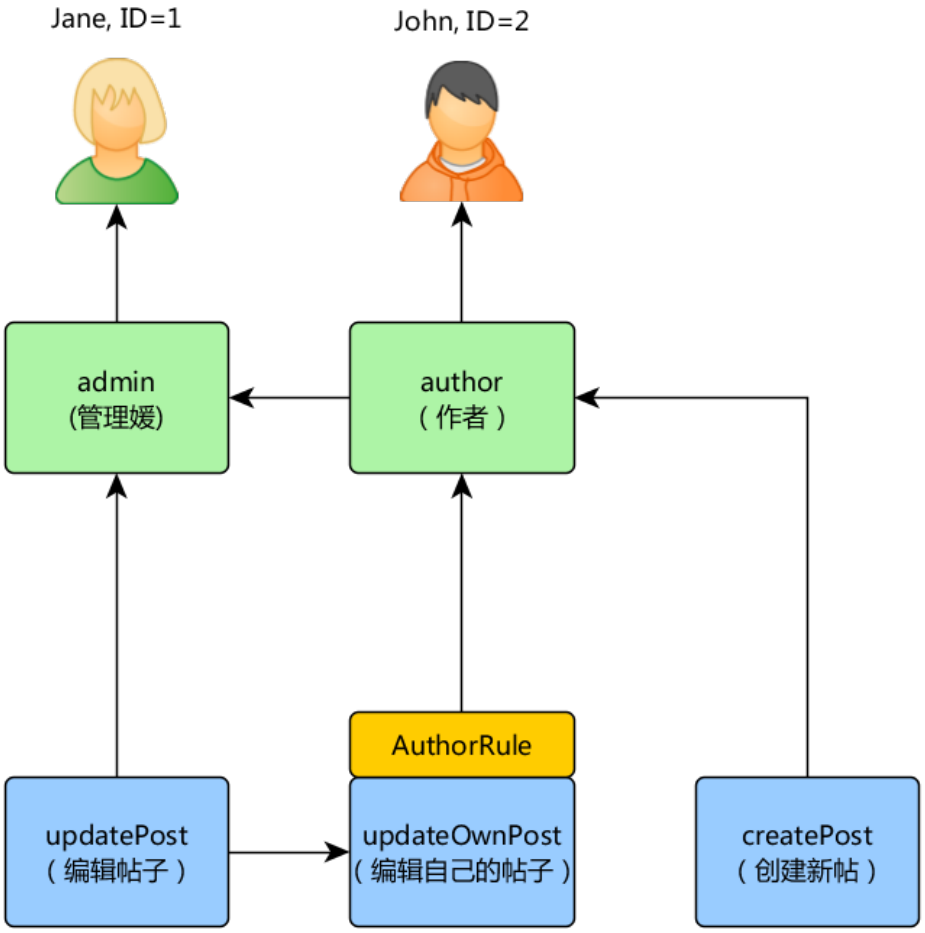
如前所述，规则给角色和权限增加额外的约束条件。规则是 `yii\rbac\Rule` 的派生类。它需要实现 `execute()` 方法。在之前我们创建的层次结构中，作者不能编辑自己的帖子，我们来修正这个问题。首先我们需要一个规则来认证当前用户是帖子的作者：

```
namespace app\rbac;

use yii\rbac\Rule;
use app\models\Post;

/**
 * 检查 authorID 是否和通过参数传进来的 user 参数相符
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';

    /**
     * @param string|integer $user 用户 ID.
     * @param Item $item 该规则相关的角色或者权限
     * @param array $params 传给 ManagerInterface::checkAccess() 的参数
     * @return boolean 代表该规则相关的角色或者权限是否被允许
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user
        : false;
    }
}
```

上述规则检查 `post` 是否是 `$user` 创建的。我们还要在之前的命令中 创建一个特别的权限 `updateOwnPost` :

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

// 添加规则
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// 添加 "updateOwnPost" 权限并与规则关联
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" 权限将由 "updatePost" 权限使用
$auth->addChild($updateOwnPost, $updatePost);

// 允许 "author" 更新自己的帖子
$auth->addChild($author, $updateOwnPost);
```

现在我们得到如下层次结构:

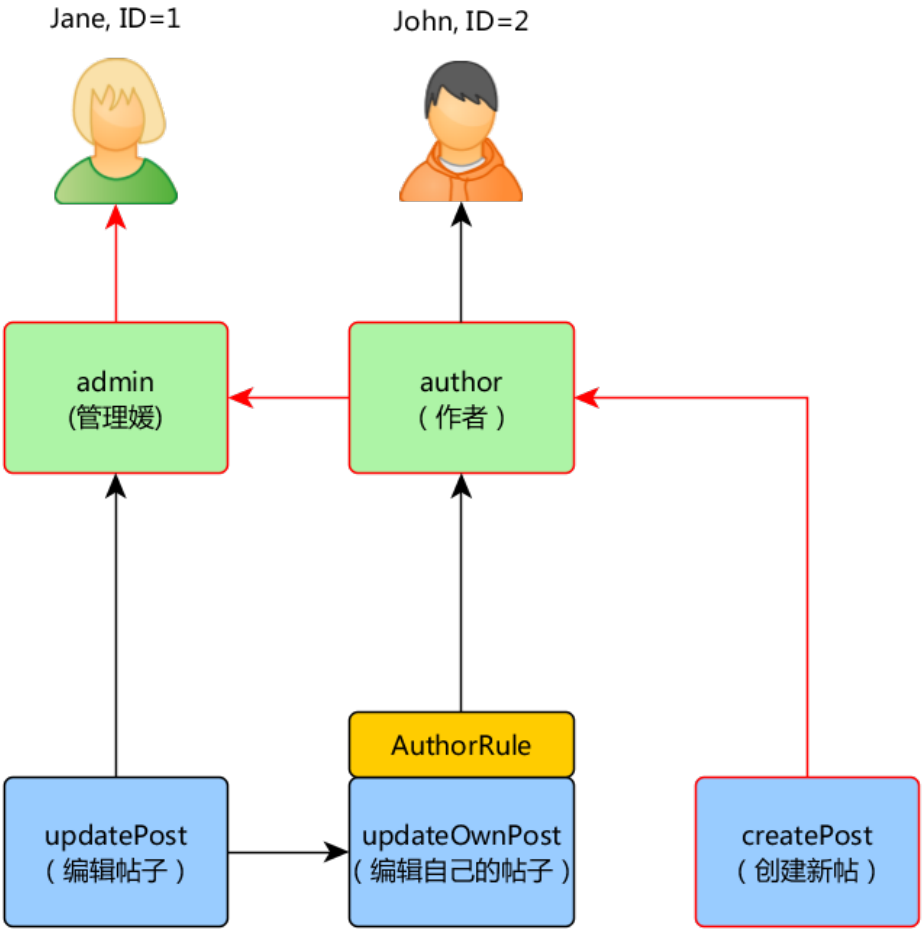


存取检查

授权数据准备好后，存取检查简单到只需要一个方法调用 `yii\rbac\ManagerInterface::checkAccess()`。 因为大多数存取检查都是针对当前用户而言，为方便起见， Yii 提供了一个快捷方法 `yii\web\User::can()`，可以如下例所示来使用：

```
if (\Yii::$app->user->can('createPost')) {
    // 建贴
}
```

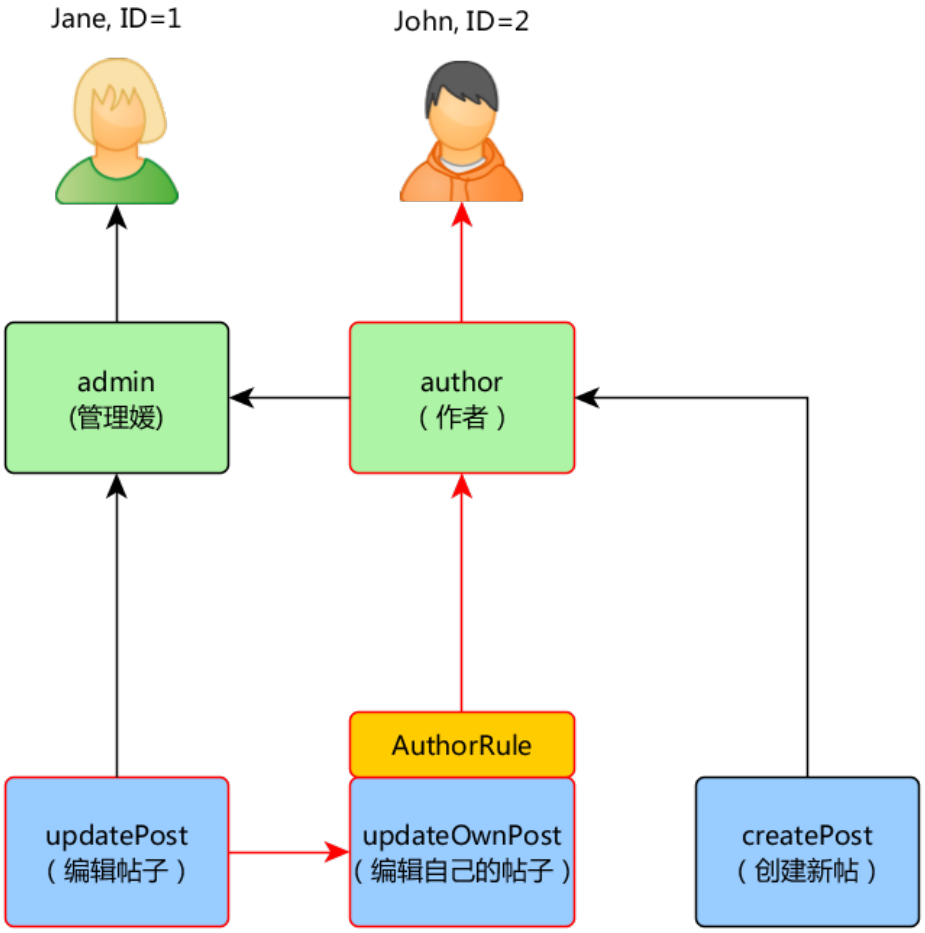
如果当前用户是 `ID=1` 的 Jane ，我们从图中的 `createPost` 开始，并试图到达 `Jane` 。 （译者注：参照图中红色路线所示的建贴授权流程）



为了检查某用户是否能更新帖子，我们需要传递一个额外的参数，该参数是 `AuthorRule` 要用的：

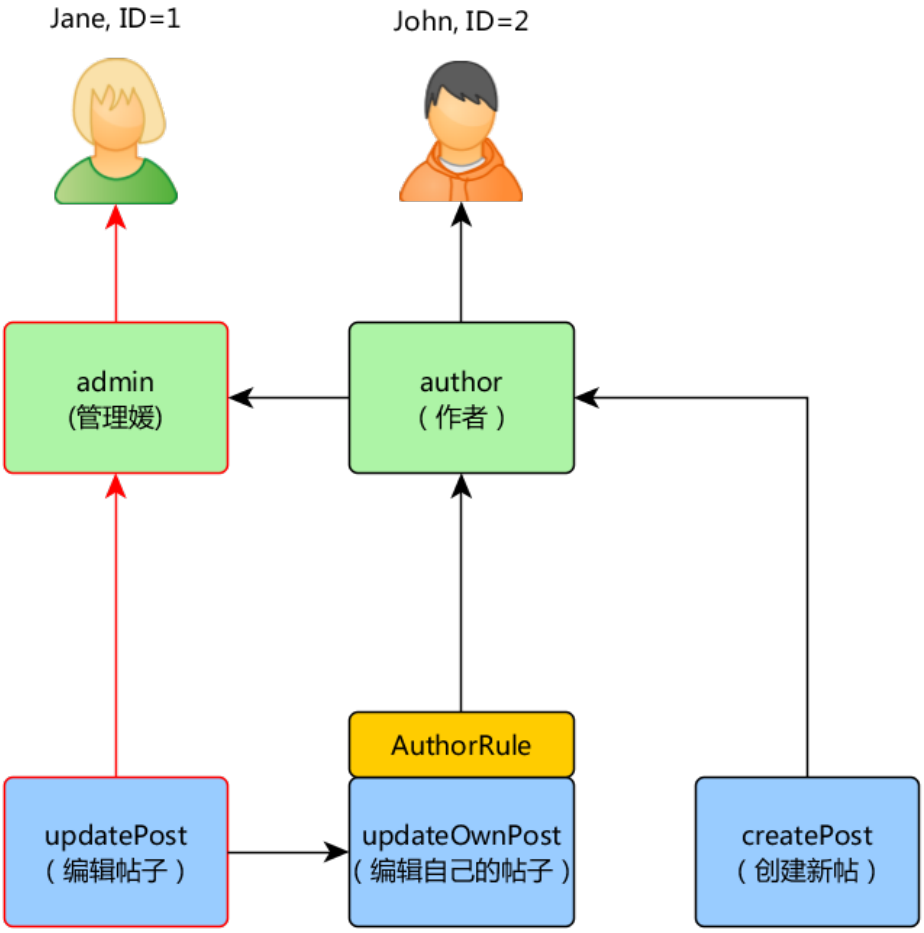
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // 更新帖子
}
```

下图所示为当前用户是 John 时所发生的事情：



我们从图中的 `updatePost` 开始，经过 `updateOwnPost`。为通过检查，`Authorrule` 规则的 `execute()` 方法应当返回 `true` 。该方法从 `can()` 方法调用接收到 `$params` 参数， 因此它的值是 `['post' => $post]` 。如果一切顺利，我们会达到指派给 John 的 `author` 角色。

对于 Jane 来说则更简单，因为她是管理员：



在您的控制器内部有几种实现授权的方式。 如果您希望细化权限来分开添加和删除的访问权限，那么您需要检查每个操作的访问权限。 您可以在每个操作方法中使用上述条件，或使用 `yii\filters\AccessControl`：

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['index'],
                    'roles' => ['managePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['view'],
                    'roles' => ['viewPost'],
                ],
```

```
                [
                    'allow' => true,
                    'actions' => ['create'],
                    'roles' => ['createPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['update'],
                    'roles' => ['updatePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['delete'],
                    'roles' => ['deletePost'],
                ],
            ],
        ],
    ];
}
```

如果所有的CRUD操作都是一起管理的，那么使用 `managePost` 这样的单一权限并且在 `yii\web\Controller::beforeAction()` 中检查它是个好主意。

在上面的例子中，没有参数与指定的访问动作的角色一起传递，但是在 `updatePost` 权限的情况下，我们需要传递 `post` 参数才能正常工作。 您可以通过在访问规则中指定 `roleParams` 将参数传递给 `yii\web\User::can()`：

```
[
    'allow' => true,
    'actions' => ['update'],
    'roles' => ['updatePost'],
    'roleParams' => function() {
        return ['post' => Post::findOne(['id' =>
        Yii::$app->request->get('id')])];
    },
],
```

在上面的例子中，`roleParams` 是一个 Closure， 将在检查访问规则时进行评估，因此模型只会在需要时加载。 如果创建角色参数是一个简单的操作，那么您可以指定一个数组，如下所示：

```
[
    'allow' => true,
    'actions' => ['update'],
    'roles' => ['updatePost'],
    'roleParams' => ['postId' => Yii::$app->request->get('id')];
],
```

使用默认角色

所谓默认角色就是 隐式 地指派给 所有 用户的角色。不需要调用 `yii\rbac\ManagerInterface::assign()` 方法做显示指派，并且授权数据中不包含指派信息。

默认角色通常与一个规则关联，用以检查该角色是否符合被检查的用户。

默认角色常常用于已经确立了一些角色的指派关系的应用（译者注：指派关系指的是应用业务逻辑层面， 并非指授权数据的结构）。比如，一个应用的 user 表中有一个 `group` 字段，代表用户属于哪个特权组。 如果每个特权组可以映射到 RBAC 的角色，你就可以采用默认角色自动地为每个用户指派一个 RBAC 角色。 让我们用一个例子展示如何做到这一点。

假设在 user 表中，你有一个 `group` 字段，用 1 代表管理员组，用 2 表示作者组。 你规划两个 RBAC 角色 `admin` 和 `author` 分别对应这两个组的权限。 你可以这样设置 RBAC 数据，

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * 检查是否匹配用户的组
 */
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;
    }
}
```

然后按 in the previous section 中的说明创建自己的 command/migration：

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

$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... 添加$author角色的子项部分代码 ... （译者注：省略部分参照之前的控制台命令）

$admin = $auth->createRole('admin');
```

```
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... 添加$admin角色的子项部分代码 ... （译者注：省略部分参照之前的控制台命令）
```

注意，在上述代码中，因为 “author” 作为 “admin” 的子角色，当你实现这个规则的 `execute()` 方法时， 你也需要遵从这个层次结构。这就是为何当角色名为 “author” 的情况下（译者注：`$item->name`就是角色名）， `execute()` 方法在组为 1 或者 2 时均要返回 true （意思是用户属于 “admin” 或者 “author” 组 ）。

接下来，在配置 `authManager` 时指定 `yii\rbac\BaseManager::$defaultRoles` 选项（译者注：在应用配置文件中的组件部分配置）：

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

现在如果你执行一个存取权限检查， 判定该规则时， `admin` 和 `author` 两个角色都将会检查。如果规则返回 true ，意思是角色符合当前用户。基于上述规则 的实现，意味着如果某用户的 `group` 值为 1 ， `admin` 角色将赋予该用户， 如果 `group` 值是 2 则将赋予 `author` 角色。

9.4 处理密码

大部分开发者知道密码不能以明文形式存储，但是许多开发者仍认为使用 `md5` 或者 `sha1` 来哈希化密码是安全的。 一度，使用上述的哈希算法是足够安全的，但是， 现代硬件的发展使得短时间内暴力破解上述算法生成的哈希串成为可能。

为了即使在最糟糕的情况下（你的应用程序被破解了）也能给用户密码提供增强的安全性， 你需要使用一个能够对抗暴力破解攻击的哈希算法。目前最好的选择是 `bcrypt`。在 PHP 中， 你可以通过 crypt 函数[4] 生成 `bcrypt` 哈希。 Yii 提供了两个帮助函数以让使用 `crypt` 来进行安全的哈希密码生成和验证更加容易。

当一个用户为第一次使用，提供了一个密码时（比如：注册时），密码就需要被哈希化。

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

[4] https://www.php.net/manual/zh/function.crypt.php

哈希串可以被关联到对应的模型属性，这样，它可以被存储到数据库中以备将来使用。

当一个用户尝试登录时，表单提交的密码需要使用之前的存储的哈希串来验证：

```
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {
    // all good, logging user in
} else {
    // wrong password
}
```

9.5 加密（Cryptography）

在本节中，我们将回顾以下安全问题：

- 生成随机数据
- 加密和解密
- 确认数据完整性

9.5.1 生成伪随机数据（Generating Pseudorandom Data）

伪随机数据在很多情况下都很有用。 例如，当通过电子邮件重置密码时，您需要生成一个令牌，将其保存到数据库中，并通过电子邮件发送给最终用户， 这反过来又会允许他们证明该帐户的所有权。 这个令牌是独一无二且难以猜测的，否则攻击者可能会预测令牌的值并重置用户的密码。

Yii 安全助手类简单生成伪随机数据：

```
$key = Yii::$app->getSecurity()->generateRandomString();
```

9.5.2 加密和解密（Encryption and Decryption）

Yii 提供了便利的帮助功能，使您可以使用密钥 加密/解密 数据。 数据通过加密功能传递，以便只有拥有密钥的人才能解密。 例如，我们需要在数据库中存储一些信息，但我们需要确保只有拥有密钥的用户才能查看它（即使应用程序数据库已被泄露）：

```
// $data 和 $secretKey 从表单中获得
$encryptedData = Yii::$app->getSecurity()->encryptByPassword($data,
$secretKey);
// 将 $encryptedData 存储到数据库
```

随后当用户想要读取数据时：

```
// $secretKey 从用户输入获得，$encryptedData 来自数据库
$data = Yii::$app->getSecurity()->decryptByPassword($encryptedData,
$secretKey);
```

也可以通过 `yii\base\Security::encryptByKey()` 和 `yii\base\Security::decryptByKey()` 使用密钥而不是密码。

9.5.3 确认数据完整性（Confirming Data Integrity）

在某些情况下，您需要验证您的数据未被第三方篡改甚至以某种方式损坏。 Yii 提供了一种简单的方法用两个帮助功能的类确认数据完整性的。

用密钥和数据生成的哈希前缀数据

```
// $secretKey 是我们的应用程序或用户密钥，$genuineData 是从可靠来源获得的
$data = Yii::$app->getSecurity()->hashData($genuineData, $secretKey);
```

检查数据完整性是否受到损害

```
// $secretKey 我们的应用程序或用户密钥，$data 从不可靠的来源获得
$data = Yii::$app->getSecurity()->validateData($data, $secretKey);
```

9.6 最佳安全实践

下面，我们将会回顾常见的安全原则，并介绍在使用 Yii 开发应用程序时，如何避免潜在安全威胁。 大多数这些原则并非您独有，而是适用于网站或软件开发， 因此，您还可以找到有关这些背后的一般概念的进一步阅读的链接。

9.6.1 基本准则

无论是开发何种应用程序，我们都有两条基本的安全准则：

1. 过滤输入
2. 转义输出

过滤输入

过滤输入的意思是，用户输入不应该认为是安全的，你需要总是验证你获得的输入值是在允许范围内。 比如，我们假设可以通过三个字段完成排序 `title`，`created_at` 和 `status`，然后，这个值是由用户输入提供的， 那么，最好在我们接收参数的时候，检查一下这个值是否是指定的范围。 对于基本的 PHP 而言，上述做法类似如下：

```
$sortBy = $_GET['sort'];
if (!in_array($sortBy, ['title', 'created_at', 'status'])) {
        throw new Exception('Invalid sort value.');
}
```

在 Yii 中，很大可能性，你会使用 表单校验器 来执行类似的检查。

进一步阅读该主题：

- https://owasp.org/www-community/vulnerabilities/Improper_Data_Validation
- https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

转义输出

转义输出的意思是，根据我们使用数据的上下文环境，数据需要被转义。比如：在 HTML 上下文， 你需要转义 <，> 之类的特殊字符。在 JavaScript 或者 SQL 中，也有其他的特殊含义的字符串需要被转义。 由于手动的给所用的输出转义容易出错， Yii 提供了大量的工具来在不同的上下文执行转义。

进一步阅读该话题：

- https://owasp.org/www-community/attacks/Command_Injection
- https://owasp.org/www-community/attacks/Code_Injection
- https://owasp.org/www-community/attacks/xss/

9.6.2 避免 SQL 注入

SQL 注入发生在查询语句是由连接未转义的字符串生成的场景，比如：

```
$username = $_GET['username'];
$sql = "SELECT * FROM user WHERE username = '$username'";
```

除了提供正确的用户名外，攻击者可以给你的应用程序输入类似 `'; DROP TABLE user; --` 的语句。 这将会导致生成如下的 SQL：

```
SELECT * FROM user WHERE username = ''; DROP TABLE user; --'
```

这是一个合法的查询语句，并将会执行以空的用户名搜索用户操作，然后，删除 `user` 表。 这极有可能导致网站出错，数据丢失。（你是否进行了规律的数据备份？）

在 Yii 中，大部分的数据查询是通过 Active Record 进行的， 而其是完全使用 PDO 预处理语句执行 SQL 查询的。在预处理语句中，上述示例中，构造 SQL 查询的场景是不可能发生的。

有时，你仍需要使用 raw queries 或者 query builder。 在这种情况下，你应该使用安全的方式传递参数。如果数据是提供给表列的值，最好使用预处理语句：

```
// query builder
$userIDs = (new Query())
    ->select('id')
    ->from('user')
    ->where('status=:status', [':status' => $status])
    ->all();

// DAO
$userIDs = $connection
    ->createCommand('SELECT id FROM user where status=:status')
    ->bindValues([':status' => $status])
    ->queryColumn();
```

如果数据是用于指定列的名字，或者表的名字，最好的方式是只允许预定义的枚举值。

```
function actionList($orderBy = null)
{
    if (!in_array($orderBy, ['name', 'status'])) {
        throw new BadRequestHttpException('Only name and status are allowed
        to order by.')
    }

    // ...
}
```

如果上述方法不行，表名或者列名应该被转义。Yii 针对这种转义提供了一个特殊的语法， 这样可以在所有支持的数据库都使用一套方案。

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

你可以在 Quoting Table and Column Names 中获取更多的语法细节。

进一步阅读该话题:

- https://owasp.org/www-community/attacks/SQL_Injection

9.6.3 防止 XSS 攻击

XSS 或者跨站脚本发生在输出 HTML 到浏览器时，输出内容没有正确的转义。 例如，如果用户可以输入其名称，那么他输入 `<script>alert('Hello!');</script>` 而非其名字 `Alexander`， 所有输出没有转义直接输出用户名的页面都会执行 JavaScript 代码 `alert('Hello!');`， 这会导致浏览器页面上出现一个警告弹出框。就具体的站点而言，除了这种无意义的警告输出外， 这样的脚本可以以你的名义发送一些消息到后台，甚至执行一些银行交易行为。

避免 XSS 攻击在 Yii 中非常简单，有如下两种一般情况:

1. 你希望数据以纯文本输出。
2. 你希望数据以 HTML 形式输出。

如果你需要的是纯文本，你可以如下简单的转义:

```
<?= \yii\helpers\Html::encode($username) ?>
```

如果是 HTML，我们可以用 HtmlPurifier 帮助类来执行:

```
<?= \yii\helpers\HtmlPurifier::process($description) ?>
```

注意 HtmlPurifier 帮助类的处理过程较为费时，建议增加缓存。

进一步阅读该话题:

- https://owasp.org/www-community/attacks/xss/

9.6.4 防止 CSRF 攻击

CSRF 是跨站请求伪造的缩写。这个攻击思想源自许多应用程序假设来自用户的浏览器请求是由用户自己产生的， 而事实并非如此。

例如，网站 an.example.com 有一个 /logout 网址，当使用简单的 GET 请求访问时, 记录用户退出。 只要用户的请求一切正常，但是有一天坏人们故意在用户经常访问的论坛上放上 <img src="https://an.example.com/logout">。浏览器在请求图像或请求页面之间没有任何区别， 所以当用户打开一个带有这样一个被操作过的 <img> 标签的页面时， 浏览器将 GET 请求发送到该 URL，用户将从 an.example.com 注销。

这是 CSRF 攻击如何运作的基本思路。可以说用户退出并不是一件严重的事情， 然而这仅仅是一个例子，使用这种方法可以做更多的事情，例如触发付款或者是改变数据。 想象一下如果某个网站有一个这样的 https://an.example.com/purse/transfer?to=anotherUser&amount=2000 网址。 使用 GET 请求访问它会导致从授权用户账户转账 $2000 给 anotherUser。我们知道，浏览器将始终发送 GET 请求来加载图像， 所以我们可以修改代码以仅接受该 URL 上的 POST 请求。 不幸的是，这并不会拯救我们，因为攻击者可以放置一些 JavaScript 代码而不是 <img> 标签，这样就可以向该 URL 发送 POST 请求。

出于这个原因，Yii 应用其他机制来防止 CSRF 攻击。

为了避免 CSRF 攻击，你总是需要:

1. 遵循 HTTP 准则，比如 GET 不应该改变应用的状态。 有关详细信息，请参阅 RFC2616[5]。
2. 保证 Yii CSRF 保护开启。

有的时候你需要对每个控制器和/或方法使用禁用 CSRF。可以通过设置其属性来实现:

```
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
    }

}
```

要对每个自定义方法禁用 CSRF 验证，您可以使用:

[5] https://www.rfc-editor.org/rfc/rfc9110.html#name-method-definitions

```
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);
    }
}
```

在 standalone actions 禁用 CSRF 必须在 `init()` 方法中设置。 不要把这段代码放在 `beforeRun()` 方法中，因为它不会起任何作用。

```
<?php

namespace app\components;

use yii\base\Action;

class ContactAction extends Action
{
    public function init()
    {
        parent::init();
        $this->controller->enableCsrfValidation = false;
    }

    public function run()
    {
          $model = new ContactForm();
          $request = Yii::$app->request;
          if ($request->referrer === 'yiipowered.com'
              && $model->load($request->post())
              && $model->validate()
          ) {
              $model->sendEmail();
          }
    }
}
```

> **警告：** 禁用 CSRF 将允许任何站点向您的站点发送 POST 请求。在这种情况下，实施额外验证非常重要，例如检查 IP 地址或秘密令牌。

进一步阅读该话题：

- https://owasp.org/www-community/attacks/csrf

9.6.5 防止文件暴露

默认的服务器 webroot 目录指向包含有 `index.php` 的 `web` 目录。在共享托管环境下，这样是不可能的， 这样导致了所有的代码，配置，日志都在webroot目录。

如果是这样，别忘了拒绝除了 `web` 目录以外的目录的访问权限。 如果没法这样做，考虑将你的应用程序托管在其他地方。

9.6.6 在生产环境关闭调试信息和工具

在调试模式下， Yii 展示了大量的错误信息，这样是对开发有用的。 同样，这些调试信息对于攻击者而言也是方便其用于破解数据结构，配置值，以及你的部分代码。 永远不要在生产模式下将你的 `index.php` 中的 `YII_DEBUG` 设置为 `true`。

你同样也不应该在生产模式下开启 Gii。它可以被用于获取数据结构信息， 代码，以及简单的用 Gii 生成的代码覆盖你的代码。

调试工具栏同样也应该避免在生产环境出现，除非非常有必要。它将会暴露所有的应用和配置的详情信息。 如果你确定需要，反复确认其访问权限限定在你自己的 IP。

进一步阅读该话题:

- https://owasp.org/www-project-.net/articles/Exception_Handling.md
- https://owasp.org/www-pdf-archive/OWASP_Top_10_2007.pdf (A6 - Information Leakage and Improper Error Handling)

9.6.7 使用 TLS 上的安全连接

Yii 提供依赖 cookie 和/或 PHP 会话的功能。如果您的连接受到威胁，这些可能会很容易受到攻击。 如果应用程序通过 TLS 使用安全连接，则风险会降低。

有关如何配置它的说明，请参阅您的 Web 服务器文档。 您还可以参考 H5BP 项目提供的示例配置:

- Nginx[6]。
- Apache[7]。
- IIS[8]。
- Lighttpd[9]。

9.6.8 安全服务器配置

本节的目的是强调在为基于 Yii 的网站提供服务配置时需要考虑的风险。除了这里涉及的要点之外， 可能还有其他与安全相关的配置选项， 所以不

[6] https://github.com/h5bp/server-configs-nginx
[7] https://github.com/h5bp/server-configs-apache
[8] https://github.com/h5bp/server-configs-iis
[9] https://github.com/h5bp/server-configs-lighttpd

要认为这部分是完整的。

避免 -header 攻击

像 `yii\web\UrlManager` 和 `yii\helpers\Url` 这样的类会使用 `currently requested host name` 来生成链接。 如果 Web 服务器配置为独立于 `Host` 标头的值提供相同的站点，这个信息并不可靠， 并且 可能由发送HTTP请求的用户伪造[10]。 在这种情况下，您应该修复您的 Web 服务器配置以便仅为指定的主机名提供站点服务 或者通过设置 `request` 应用程序组件的 `hostInfo` 属性来显式设置或过滤该值。

有关于服务器配置的更多信息，请参阅您的 web 服务器的文档：

- Apache 2：https://httpd.apache.org/docs/trunk/vhosts/examples.html#defaultallports
- Nginx：https://www.nginx.com/resources/wiki/start/topics/examples/server_blocks/

如果您无权访问服务器配置，您可以在应用程序级别设置 `yii\filters\HostControl` 过滤器， 以防此类的攻击。

```
// Web Application configuration file
return [
    'as hostControl' => [
        'class' => 'yii\filters\HostControl',
        'allowedHosts' => [
            'example.com',
            '*.example.com',
        ],
        'fallbackHostInfo' => 'https://example.com',
    ],
    // ...
];
```

> **注意：** 您应该始更倾向于使用 web 服务器配置 ‘host header attack’ 保护而不是使用过滤器。 仅当服务器配置设置不可用时 `yii\filters\HostControl` 才应该被使用。

[10] https://www.acunetix.com/vulnerabilities/web/host-header-attack

Chapter 10

缓存（Caching）

10.1 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，在较的低层面，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

10.2 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。
```

从 2.0.11 版本开始, 缓存组件 提供了 `getOrSet()` 方法来简化数据的取回、计算和存储。 下面的代码逻辑和上一个例子是完全一样的:

```
$data = $cache->getOrSet($key, function () {
    return $this->calculateSomething();
});
```

当缓存中有关联 $key 的数据时，将返回这个缓存的值。 否则就执行匿名函数来计算出将要缓存的数据并返回它。

如果匿名函数需要作用域外的数据时，可以使用 `use` 语句把这些数据传递到匿名函数中。 例如:

```
$user_id = 42;
$data = $cache->getOrSet($key, function () use ($user_id) {
    return $this->calculateSomething($user_id);
});
```

> **注意:** `getOrSet()` 方法也支持缓存持续性和缓存依赖。 请看 缓存过期 和 缓存依赖 来了解详细信息。

10.2.1 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[1] 服务器:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

[1] https://memcached.org/

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 `APC cache`：

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> **提示：** 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[2] 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::$cacheTable` 对应的表。
- `yii\caching\ArrayCache`：仅通过将值存储在数组中来为当前请求提供缓存。 为了增强 ArrayCache 的性能，您可以通过将 `yii\caching\ArrayCache::$serializer` 设置为 `false` 来禁用已存储数据的序列化。
- `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[3] 和 memcached[4] 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[5] 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 ）。

[2] https://www.php.net/manual/zh/book.apcu.php
[3] https://www.php.net/manual/zh/book.memcache.php
[4] https://www.php.net/manual/zh/book.memcached.php
[5] https://redis.io/

- `yii\caching\WinCache`：使用 PHP WinCache[6] （另可参考[7]）扩展.
- `yii\caching\XCache`：使用 PHP XCache[8]扩展。
- `yii\caching\ZendDataCache`：使用 [Zend Data Cache](https://files.zend.com/help/Zend-Server-6/zend- server.htm#data_cache_component.htm) 作为底层缓存媒介。

> **提示：** 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）。

10.2.2 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:

- `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。
- `set()`：将一个由键指定的数据项存放到缓存中。
- `add()`：如果缓存中未找到该键，则将指定数据存放到缓存中。
- `getOrSet()`：返回由键指定的缓存项，或者执行回调函数，把函数的返回值用键来关联存储到缓存中， 最后返回这个函数的返回值。
- `multiGet()`：由指定的键获取多个缓存数据项。
- `multiSet()`：一次存储多个数据项到缓存中，每个数据都由一个键来指明。
- `multiAdd()`：一次存储多个数据项到缓存中，每个数据都由一个键来指明。 如果某个键已经存在，则略过该数据项不缓存。
- `exists()`：返回一个值，指明某个键是否存在于缓存中。
- `delete()`：通过一个键，删除缓存中对应的值。
- `flush()`：删除缓存中的所有数据。

> **注意：** 千万别直接用 `false` 布尔值当做数据项缓存，因为 `get()` 方法用 `false` 作为返回值来表名对应的缓存项不存在。 你可以把 `false` 放到一个数组里然后缓存这个数组来避免上述的混淆问题。

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。 `multiGet()` 和 `multiAdd()` 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');
```

[6] https://iis.net/downloads/microsoft/wincache-extension
[7] https://www.php.net/manual/zh/book.wincache.php
[8] https://en.wikipedia.org/wiki/List_of_PHP_accelerators#XCache

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。

> **注意：** 通过 `multiSet()` 或者 `multiAdd()` 方法缓存的数据项的键，它的类型只能是字符串或整型， 如果你想使用较为复杂的键，可以通过 `set()` 或者 `add()` 方法来存储。

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 `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 `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
}
```

从 2.0.11 开始，如果想自定义缓存的持续时间，你可以在缓存组件配置中设置 `defaultDuration` 成员属性的值。 这样设置会覆盖默认的缓存持续时间，且在使用 `set()` 方法时不必每次都传递 `$duration` 参数。

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 `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 `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\CallbackDependency`：如果指定的PHP回调结果发生变化，依赖性将改变。
- `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\TagDependency`：将缓存的数据项与一个或多个标签相关联。 您可以通过调用 `yii\caching\TagDependency::invalidate()` 来检查指定标签的缓存数据项是否有效。

> **注意：** 避免对带有缓存依赖的缓存项使用 `exists()` 方法，因为它不检测缓存依赖（如果有的话）是否有效，所以调用 `get()` 可能返回 `false` 而调用 `exists()` 却返回 `true`。

10.2.3 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 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：

```
$result = $db->cache(function ($db) {

    // SQL 查询的结果将从缓存中提供
    // 如果启用查询缓存并且在缓存中找到查询结果
    return $db->createCommand('SELECT * FROM customer WHERE
    id=1')->queryOne();

});
```

查询缓存可以用在DAO和ActiveRecord上：

```
$result = Customer::getDb()->cache(function ($db) {
    return Customer::find()->where(['id' => 1])->one();
});
```

> **信息：** 有些 DBMS （例如：MySQL[9]） 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。

自 2.0.14 以后，您可以使用以下快捷方法：

```
(new Query())->cache(7200)->all();
// and
User::find()->cache(7200)->all();
```

配置

查询缓存通过 `yii\db\Connection` 有三个全局可配置选项：

- `enableQueryCache`：是否打开或关闭查询缓存。它默认为 `true`。请注意，要有效打开查询缓存，您还需要有一个由 `queryCache` 所指定的有效缓存。
- `queryCacheDuration`：这表示查询结果在缓存中保持有效的秒数。您可以使用 0 来表示查询结果永久保留在缓存中。该属性是在未指定持续时间的情况下调用 `yii\db\Connection::cache()` 使用的默认值。
- `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。

[9] https://dev.mysql.com/doc/refman/5.6/en/query-cache.html

使用

如果您有多个需要利用查询缓存的 SQL 查询，则可以使用 `yii\db\Connection::cache()`。 用法如下，

```
$duration = 60;      // 缓存查询结果 60 秒。
$dependency = ...;   // 可选的依赖关系

$result = $db->cache(function ($db) {

    // ... 在这里执行 SQL 查询 ...

    return $result;

}, $duration, $dependency);
```

在匿名函数里的任何一个 SQL 查询都将使用指定的依赖项缓存指定的持续时间 如果一个 SQL 查询的结果在缓存中有效，那么这个 SQl 语句将会被跳过而它的查询结果会直接从缓存中读取。 如果你没有指明 `$duration` 参数， 那么使用 `queryCacheDuration` 属性。

有时在`cache()`里，你可能不想缓存某些特殊的查询， 这时你可以用`yii\db\Connection::noCache()`。

```
$result = $db->cache(function ($db) {

    // 使用查询缓存的 SQL 查询

    $db->noCache(function ($db) {

        // 不使用查询缓存的 SQL 查询

    });

    // ...

    return $result;
});
```

如果您只想为单个查询使用查询缓存，则可以在构建命令时调用 `yii\db\Command::cache()`。 例如，

```
// 使用查询缓存并将查询缓存持续时间设置为 60 秒
$customer = $db->createCommand('SELECT * FROM customer WHERE
id=1')->cache(60)->queryOne();
```

您还可以使用 `yii\db\Command::noCache()` 禁用单个命令的查询缓存。例如，

```
$result = $db->cache(function ($db) {

    // 使用查询缓存的 SQL 查询
```

```
    // 对此命令不使用查询缓存
    $customer = $db->createCommand('SELECT * FROM customer WHERE
    id=1')->noCache()->queryOne();

    // ...

    return $result;
});
```

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。

缓存冲刷

当你想让所有的缓存数据失效时，可以调用 `yii\caching\Cache::flush()`。

冲刷缓存数据，你还可以从控制台调用 `yii cache/flush`。

- `yii cache`: 列出应用中可用的缓存组件
- `yii cache/flush cache1 cache2`: 刷新缓存组件`cache1`， `cache2` (可以传递多个用空格分开的缓存组件)
- `yii cache/flush-all`: 刷新应用中所有的缓存组件
- `yii cache/flush-schema db`: 清除给定连接组件的数据库表结构缓存

> **信息**: 默认情况下，控制台应用使用独立的配置文件。 所以，为了上述命令发挥作用，请确保 Web 应用和控制台应用配置相同的缓存组件。

10.3 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 beginCache() 和 endcache() 方法包裹内容生成逻辑。 如果缓存中存在该内容，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。

10.3.1 缓存选项

如果要为片段缓存指定额外配置项， 请通过向 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 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，则它将采用默认值 60，这意味着缓存的内容将在 60 秒后过期。

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 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 `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 `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();
}
```

10.3.2 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。

10.3.3 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();
}
```

`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。

> **注意：** 从版本 2.0.14 开始，动态内容 API 通过 `yii\base\DynamicContentAwareInterface` 接口及其 `yii\base\DynamicContentAwareTrait` 特质开放。 举个例子，你可以参考 `yii\widgets\FragmentCache` 类。

10.4 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。

10.5 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。 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。

- Last-Modified
- Etag
- Cache-Control

10.5.1 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 当前处理的动作对象
 * @param array $params "params" 属性的值
 * @return int 页面修改时的 Unix 时间戳
 */
function ($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。

10.5.2 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 当前处理的动作对象
 * @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）[10]， 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。

10.5.3 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
```

10.5.4 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

[10] https://datatracker.ietf.org/doc/html/rfc7232#section-2.4

`\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[11] 了解这些值的含义。

10.5.5 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）。

[11] https://www.php.net/manual/zh/function.session-cache-limiter.php

Chapter 11

RESTful Web 服务（RESTful Web Services）

11.1 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;
- 集合分页，过滤和排序;
- 支持 HATEOAS[1];
- 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 `ActiveRecord` 类 `app\models\User` 来访问该用户数据.

11.1.1 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;
```

[1] https://zh.wikipedia.org/wiki/HATEOAS

```
class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

控制器类扩展自 `yii\rest\ActiveController`。 通过指定 `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。

11.1.2 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。

> **注意：** Yii 将自动复数控制器名称以便在端点中使用（参见下面的 Trying it Out 部分）。 您可以使用 `yii\rest\UrlRule::$pluralize` 属性配置。

11.1.3 启用 JSON 输入

为了使 API 接收 JSON 格式的输入数据，配置 `request` 应用程序组件的 `parsers` 属性使用 `yii\web\JsonParser` 用于JSON输入：

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

> **信息：** 上述配置是可选的。若未按上述配置，API 将仅可以分辨 `application/x-www-form-urlencoded` 和 `multipart/form-data` 输入格式。

11.1.4 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

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
...
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
...
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>
```

以下命令将创建一个新的用户通过发送JSON格式的用户数据的 POST 请求:

```
$ curl -i -H "Accept:application/json" -H "Content-Type:application/json"
-XPOST "http://localhost/users" -d '{"username": "example", "email":
"user@example.com"}'

HTTP/1.1 201 Created
...
Location: http://localhost/users/1
Content-Length: 99
Content-Type: application/json; charset=UTF-8

{"id":1,"username":"example","email":"user@example.com","created_at":1414674789,"updated_at":1414
```

> **提示:** 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。

> **信息:** 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。

此外，您可以对 `http://localhost/users?sort=email` 或 `http://localhost/users?sort=-email` 等集合进行排序。 可以使用数据过滤器来实现过滤集合 `http://localhost/users?filter[id]=10` 或 `http://localhost/users?filter[email][like]=gmail.com`。 有关详细信息，请参阅 Resources 部分。

11.1.5 总结

使用 Yii 框架的 RESTful 风格的 API, 在控制器的动作中实现 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 `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。

11.2 资源

RESTful 的 API 都是关于访问和操作 资源，可将资源看成 MVC 模式中的模型

在如何代表一个资源没有固定的限定，在 Yii 中通常使用 `yii\base\Model` 或它的子类（如 `yii\db\ActiveRecord`） 代表资源，是为以下原因：

- `yii\base\Model` 实现了 `yii\base\Arrayable` 接口， 它允许你通过 RESTful API 自定义你想要公开的资源数据。
- `yii\base\Model` 支持 输入验证, 在你的 RESTful API 需要支持数据输入时非常有用。
- `yii\db\ActiveRecord` 提供了强大的数据库访问和操作方面的支持，如资源数据需要存到数据库它提供了完美的支持。

本节主要描述资源类如何从 `yii\base\Model` (或它的子类) 继承 并指定哪些数据可通过 RESTful API 返回，如果资源类没有 继承 `yii\base\Model` 会将它所有的公开成员变量返回。

11.2.1 字段

当 RESTful API 响应中包含一个资源时，该资源需要序列化成一个字符串。 Yii 将这个过程分成两步，首先，资源会被 `yii\rest\Serializer` 转换成数组， 然后，该数组会通过 `response formatters` 根据请求格式（如 JSON，XML）被序列化成字符串。 当开发一个资源类时应重点关注第一步。

通过覆盖 `fields()` 和/或 `extraFields()` 方法, 可指定资源中称为 字段 的数据放入展现数组中， 两个方法的差别为前者指定默认包含到展现数组的字段集合， 后者指定由于终端用户的请求包含 `expand` 参数哪些额外的字段应被包含到展现数组，例如，

```
// 返回 fields() 方法中声明的所有字段
http://localhost/users

// 只返回 fields() 方法中声明的‘‘id’’和‘‘email’’字段
http://localhost/users?fields=id,email
```

```
// 返回 fields() 方法中声明的所有字段，以及 extraFields() 方法中的“profile”字
段
http://localhost/users?expand=profile

// 返回 fields() 方法中声明的所有字段，以及 post 中的“author”
// 如果它位于 post 模型的 extraFields() 中
http://localhost/comments?expand=post.author

// 返回 fields() 方法中的“id”，“email”，以及 extraFields() 方法中
的“profile”字段
http://localhost/users?fields=id,email&expand=profile
```

覆盖 方法

`yii\base\Model::fields()` 默认返回模型的所有属性作为字段， `yii\db\ActiveRecord::fields()` 只返回和数据表关联的属性作为字段。

可覆盖 `fields()` 方法来增加、删除、重命名、重定义字段， `fields()` 的返回值应为数组，数组的键为字段名 数组的值为对应的字段定义，可为属性名或返回对应的字段值的匿名函数， 特殊情况下，如果字段名和属性名相同， 可省略数组的键，例如

```php
// 明确列出每个字段，适用于你希望数据表或
// 模型属性修改时不导致你的字段修改（保持后端API兼容性）
public function fields()
{
    return [
        // 字段名和属性名相同
        'id',
        // 字段名为"email"，对应的属性名为"email_address"
        'email' => 'email_address',
        // 字段名为"name"，值由一个PHP回调函数定义
        'name' => function (($model) {
            return $model->first_name . ' ' . $model->last_name;
        },
    ];
}

// 过滤掉一些字段，适用于你希望继承
// 父类实现同时你想屏蔽掉一些敏感字段
public function fields()
{
    $fields = parent::fields();

    // 删除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：** 模型的所有属性默认会被包含到 API 结果中， 应检查数据确保没包含敏感数据，如果有敏感数据， 应覆盖 `fields()` 过

滤掉，在上述例子中，我们选择过滤掉 `auth_key`， `password_hash` 和 `password_reset_token`。

覆盖 方法

`yii\base\Model::extraFields()` 默认返回空值， `yii\db\ActiveRecord::extraFields()` 返回和数据表关联的属性。

`extraFields()` 返回的数据格式和 `fields()` 相同， 一般 `extraFields()` 主要用于指定哪些值为对象的字段， 例如，给定以下字段申明

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

`http://localhost/users?fields=id,email&expand=profile` 的请求可能返回如下 JSON 数据:

```
[
    {
        "id": 100,
        "email": "100@example.com",
        "profile": {
            "id": 100,
            "age": 30,
        }
    },
    ...
]
```

11.2.2 链接

HATEOAS[2], 是 Hypermedia as the Engine of Application State的缩写, 提升 RESTful API 应返回允许终端用户访问的资源操作的信息， HATEOAS 的目的是在API中返回包含相关链接信息的资源数据。

资源类通过实现 `yii\web\Linkable` 接口来支持 HATEOAS， 该接口包含方法 `getLinks()` 来返回 `links` 列表，典型情况下应返回包含代表本资源对象 URL 的 `self` 链接，例如

```
use yii\base\Model;
use yii\web\Link; // represents a link object as defined in JSON Hypermedia API Language.
use yii\web\Linkable;
```

[2] https://zh.wikipedia.org/wiki/HATEOAS

```php
use yii\helpers\Url;

class UserResource extends Model implements Linkable
{
    public $id;
    public $email;

    //...

    public function fields()
    {
        return ['id', 'email'];
    }

    public function extraFields()
    {
        return ['profile'];
    }

    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id],
            true),
            'edit' => Url::to(['user/view', 'id' => $this->id], true),
            'profile' => Url::to(['user/profile/view', 'id' => $this->id],
            true),
            'index' => Url::to(['users'], true),
        ];
    }
}
```

当响应中返回一个 `User` 对象， 它会包含一个 `_links` 单元表示和用户相关的链接，例如

```
{
    "id": 100,
    "email": "user@example.com",
    // ...
    "_links" => {
        "self": {
            "href": "https://example.com/users/100"
        },
        "edit": {
            "href": "https://example.com/users/100"
        },
        "profile": {
            "href": "https://example.com/users/profile/100"
        },
        "index": {
            "href": "https://example.com/users"
        }
    }
}
```

11.2.3 集合

资源对象可以组成 集合， 每个集合包含一组相同类型的资源对象。

集合可被展现成数组，更多情况下展现成 data providers。 因为 data providers 支持资源的排序和分页， 这个特性在 RESTful API 返回集合时也用到， 例如，如下操作返回 post 资源的 data provider：

```
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(),
        ]);
    }
}
```

当在 RESTful API 响应中发送 data provider 时， `yii\rest\Serializer` 会取出资源的当前页并组装成资源对象数组， `yii\rest\Serializer` 也通过如下 HTTP 头包含页码信息：

- `X-Pagination-Total-Count`： 资源所有数量；
- `X-Pagination-Page-Count`： 页数；
- `X-Pagination-Current-Page`： 当前页（从 1 开始）；
- `X-Pagination-Per-Page`： 每页资源数量；
- `Link`： 允许客户端一页一页遍历资源的导航链接集合。

由于 REST API 中的集合是 data provider，因此它共享所有 data provider 功能，即分页和排序。

可在快速入门 一节中找到样例。

过滤集合

从 2.0.13 版本开始，Yii 提供了过滤集合的工具。一个例子可以在 快速入门 指南中找到。如果您自己实现末端， 可以按照 Data Providers 指南的 Filtering Data Providers using Data Filters 部分中的描述进行过滤。

11.3 控制器

在创建资源类和指定资源格输出式化后， 下一步就是创建控制器操作将资源通过 RESTful APIs 展现给终端用户。

Yii 提供两个控制器基类来简化创建 RESTful 操作的工作： `yii\rest\Controller` 和 `yii\rest\ActiveController`， 两个类的差别是后者提供

一系列将资源处理成 Active Record 的操作。 因此如果使用 Active Record 内置的操作会比较方便，可考虑将控制器类 继承 `yii\rest\ActiveController`，它会让你用最少的代码完成强大的 RESTful APIs。

`yii\rest\Controller` 和 `yii\rest\ActiveController` 提供以下功能，一些功能在后续章节详细描述：

- HTTP 方法验证；
- 内容协商和数据格式化；
- 认证；
- 频率限制。

`yii\rest\ActiveController` 额外提供以下功能：

- 一系列常用动作：`index`，`view`，`create`，`update`，`delete`，`options`；
- 对动作和资源进行用户认证。

11.3.1 创建控制器类

当创建一个新的控制器类，控制器类的命名最好使用资源名称的单数格式， 例如，提供用户信息的控制器 可命名为 `UserController`。

创建新的操作和 Web 应用中创建操作类似， 唯一的差别是 Web 应用中调用 `render()` 方法渲染一个视图作为返回值， 对于 RESTful 操作直接返回数据， `serializer` 和 `response object` 会处理原始数据到请求格式的转换，例如

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2 过滤器

`yii\rest\Controller` 提供的大多数 RESTful API 功能通过过滤器实现。特别是以下过滤器会按顺序执行：

- `contentNegotiator`：支持内容协商， 在 响应格式化 一节描述；
- `verbFilter`：支持 HTTP 方法验证；
- `authenticator`：支持用户认证， 在认证一节描述；
- `rateLimiter`：支持频率限制， 在频率限制 一节描述。

这些过滤器都在 `behaviors()` 方法中声明， 可覆盖该方法来配置单独的过滤器，禁用某个或增加你自定义的过滤器。 例如，如果你只想用 HTTP 基础认证，可编写如下代码：

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

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
```

```
    return $behaviors;
}
```

CORS

将 Cross-Origin Resource Sharing 过滤器添加到控制器比添加到上述其他过滤器中要复杂一些， 因为必须在认证方法之前应用 CORS 过滤器， 因此与其他过滤器相比，需要一些稍微不同的方式来实现。 并且还要为 CORS Preflight requests[3] 禁用身份验证， 这样浏览器就可以安全地确定是否可以事先做出请求， 而无需发送身份验证凭据。 下面显示了将 `yii\filters\Cors` 过滤器添加到从 `yii\rest\ActiveController` 扩展的控制器所需的代码：

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

public function behaviors()
{
    $behaviors = parent::behaviors();

    // remove authentication filter
    $auth = $behaviors['authenticator'];
    unset($behaviors['authenticator']);

    // add CORS filter
    $behaviors['corsFilter'] = [
        'class' => \yii\filters\Cors::class,
    ];

    // re-add authentication filter
    $behaviors['authenticator'] = $auth;
    // avoid authentication on CORS-pre-flight requests (HTTP OPTIONS
    method)
    $behaviors['authenticator']['except'] = ['options'];

    return $behaviors;
}
```

11.3.3 继承

如果你的控制器继承 `yii\rest\ActiveController`， 应设置 `modelClass` 属性 为通过该控制器返回给用户的资源类名，该类必须继承 `yii\db\ActiveRecord`。

自定义动作

`yii\rest\ActiveController` 默认提供一下动作：

- `index`：按页列出资源；

[3] https://developer.mozilla.org/zh-CN/docs/Web/HTTP/CORS#%E9%A2%84%E6%A3%80%E8%AF%B7%E6%B1%82

- view：返回指定资源的详情；
- create：创建新的资源；
- update：更新一个存在的资源；
- delete：删除指定的资源；
- options：返回支持的 HTTP 方法。

所有这些动作通过 actions() 方法申明，可覆盖 actions() 方法配置或禁用这些动作， 如下所示：

```
public function actions()
{
    $actions = parent::actions();

    // 禁用 "delete" 和 "create" 动作
    unset($actions['delete'], $actions['create']);

    // 使用 "prepareDataProvider()" 方法自定义数据 provider
    $actions['index']['prepareDataProvider'] = [$this,
    'prepareDataProvider'];

    return $actions;
}

public function prepareDataProvider()
{
    // 为 "index" 动作准备和返回数据 provider
}
```

请参考独立动作类的参考文档学习哪些配置项有用。

执行访问检查

通过 RESTful APIs 显示数据时，经常需要检查当前用户是否有权限访问和操作所请求的资源， 在 yii\rest\ActiveController 中， 可覆盖 checkAccess() 方法来完成权限检查。

```
/**
 * Checks the privilege of the current user.
 *
 * This method should be overridden to check whether the current user has
the privilege
 * to run the specified action against the specified data model.
 * If the user does not have access, a [[ForbiddenHttpException]] should be
thrown.
 *
 * @param string $action the ID of the action to be executed
 * @param \yii\base\Model $model the model to be accessed. If `null`, it
means no specific model is being accessed.
 * @param array $params additional parameters
 * @throws ForbiddenHttpException if the user does not have access
 */
public function checkAccess($action, $model = null, $params = [])
```

```
{
    // check if the user can access $action and $model
    // throw ForbiddenHttpException if access should be denied
    if ($action === 'update' || $action === 'delete') {
        if ($model->author_id !== \Yii::$app->user->id)
            throw new \yii\web\ForbiddenHttpException(sprintf('You can only
            %s articles that you\'ve created.', $action));
    }
}
```

checkAccess() 方法默认会被 yii\rest\ActiveController 默认动作所调用，如果创建新的操作并想执行权限检查， 应在新的动作中明确调用该方法。

> **提示：** 可使用 Role-Based Access Control (RBAC) 基于角色权限控制组件 实现 checkAccess()。

11.4 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 和 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。

> **信息:** 控制器的 ID 复数化由 yii\helpers\Inflector::pluralize() 完成。该方法遵循 特定的规则。举个例子，单词 box 会被复数化为 boxes 而不是 boxs。

如果自动复数化不能满足你的需求，你也可以配置 yii\rest\UrlRule::$controller 属性来明确指定如何将端点URL中使用的名称映射到 控制器ID。例如，以下代码将名称 u 映射到控制器ID user。

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

11.4.1 包含规则的额外配置

在 yii\rest\UrlRule 中所包含的每个规则中，指定额外的配置可能很有用。 一个很好的例子就是指定 expand 参数的默认值:

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['user'],
    'ruleConfig' => [
        'class' => 'yii\web\UrlRule',
        'defaults' => [
            'expand' => 'profile',
        ]
    ],
],
```

11.5 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[4]。
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。 response formatters 通过 response 应用程序 组件来注册完成。

11.5.1 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
```

[4] https://zh.wikipedia.org/wiki/%E5%86%85%E5%AE%B9%E5%8D%8F%E5%95%86

```
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 `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 `formats` 属性， 类似如下 API 控制器类:

```php
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。

11.5.2 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": {
            "href": "http://localhost/users?page=1"
        },
        "next": {
            "href": "http://localhost/users?page=2"
        },
        "last": {
            "href": "http://localhost/users?page=50"
```

```
        }
    },
    "_meta": {
        "totalCount": 1000,
        "pageCount": 50,
        "currentPage": 1,
        "perPage": 20
    }
}
```

控制 JSON 输出

JSON 响应将由 `JsonResponseFormatter` 类来生成， 并且将在内部使用 `JSON helper`。 这个格式化程序可以配置不同的选项， 比如 `$prettyPrint`，这对于开发更好的可读式响应更有用， 或者用 `$encodeOptions` 去控制 JSON 编码的输出。

格式化程序可以在 configuration 的 `response` 应用程序组件 `formatters` 的属性进行配置， 如下所示：

```
'response' => [
    // ...
    'formatters' => [
        \yii\web\Response::FORMAT_JSON => [
            'class' => 'yii\web\JsonResponseFormatter',
            'prettyPrint' => YII_DEBUG, // use "pretty" output in debug
            mode
            'encodeOptions' => JSON_UNESCAPED_SLASHES |
            JSON_UNESCAPED_UNICODE,
            // ...
        ],
    ],
],
```

当使用 DAO 数据库层从数据库返回数据时，所有的数据将会表示成字符串， 这不总是预期的结果，尤其是数值应该表现为 JSON 的中的数字时。当使用 ActiveRecord 层从数据库检索数据时， 在 `yii\db\ActiveRecord::populateRecord()` 中从数据库中提取数据，数字列的值将会被转换为整数。

11.6 认证

和 Web 应用不同，RESTful APIs 通常是无状态的， 也就意味着不应使用 sessions 或 cookies， 因此每个请求应附带某种授权凭证，因为用户授权状态可能没通过 sessions 或 cookies 维护， 常用的做法是每个请求都发送一个秘密的 access token 来认证用户， 由于 access token 可以唯一识别和认证用户， **API 请求应通过 HTTPS 来防止 man-in-the-middle（MitM）中间人攻击。**

下面有几种方式来发送 access token：

- HTTP 基本认证[5]：access token 当作用户名发送，应用在 access token 可安全存在 API 使用端的场景， 例如，API 使用端是运行在一台服务器上的程序。
- 请求参数：access token 当作 API URL 请求参数发送，例如 `https://example.com/users?access-token=x`
由于大多数服务器都会保存请求参数到日志， 这种方式应主要用于`JSONP` 请求，因为它不能使用HTTP头来发送access token
- OAuth 2[6]：使用者从认证服务器上获取基于 OAuth2 协议的 access token， 然后通过 HTTP Bearer Tokens[7] 发送到 API 服务器。

Yii 支持上述的认证方式，你也可很方便的创建新的认证方式。

为你的 APIs 启用认证，做以下步骤：

1. 配置 `user` 应用组件：
   - 设置 `enableSession` 属性为 `false`。
   - 设置 `loginUrl` 属性为`null` 显示一个HTTP 403 错误而不是跳转到登录界面。
2. 在你的REST 控制器类中配置`authenticator` 行为来指定使用哪种认证方式
3. 在你的`user identity class` 类中实现 `yii\web\IdentityInterface::findIdentityByAccessToken()` 方法。

步骤 1 不是必要的，但是推荐配置，因为 RESTful APIs 应为无状态的，当 `enableSession` 为 false， 请求中的用户认证状态就不能通过 session 来保持，每个请求的认证通过步骤 2 和 3 来实现。

> **提示：** 如果你将 RESTful APIs 作为应用开发，可以设置应用配置中 `user` 组件的 `enableSession`， 如果将 RESTful APIs 作为模块开发，可以在模块的 `init()` 方法中增加如下代码，如下所示：
>
> ```
> public function init()
> {
>  parent::init();
>  \Yii::$app->user->enableSession = false;
> }
> ```

例如，为使用 HTTP Basic Auth，可配置 `authenticator` 行为，如下所示：

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

public function behaviors()
{
    $behaviors = parent::behaviors();
```

[5] https://zh.wikipedia.org/wiki/HTTP%E5%9F%BA%E6%9C%AC%E8%AE%A4%E8%AF%81
[6] https://oauth.net/2/
[7] https://datatracker.ietf.org/doc/html/rfc6750

```
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
    return $behaviors;
}
```

如果你想支持上面解释的所有三种认证方法，可以使用 `CompositeAuth`，如下所示：

```
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::class,
        'authMethods' => [
            HttpBasicAuth::class,
            HttpBearerAuth::class,
            QueryParamAuth::class,
        ],
    ];
    return $behaviors;
}
```

`authMethods` 中每个单元应为一个认证方法名或配置数组。

`findIdentityByAccessToken()` 方法的实现是系统定义的， 例如，一个简单的场景，当每个用户只有一个 access token，可存储 access token 到 user 表的 `access_token` 列中， 方法可在 `User` 类中简单实现，如下所示：

```
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]);
    }
}
```

在上述认证启用后，对于每个 API 请求， 请求控制器都会在它的 `beforeAction()` 步骤中对用户进行认证。

如果认证成功，控制器再执行其他检查(如频率限制，操作权限)，然后再执行动作， 授权用户信息可使用 `Yii::$app->user->identity` 获取。

如果认证失败，会发送一个 HTTP 状态码为 401 的响应， 并带有其他相关信息头（如HTTP 基本认证会有 `WWW-Authenticate` 头信息）。

11.6.1 授权

在用户认证成功后，你可能想要检查他是否有权限执行对应的操作来获取资源， 这个过程称为 *authorization* ， 详情请参考 Authorization section。

如果你的控制器从 `yii\rest\ActiveController` 类继承， 可覆盖 `yii\rest\Controller::checkAccess()` 方法 来执行授权检查，该方法会被 `yii\rest\ActiveController` 内置的操作调用。

11.7 限流 (Rate Limiting)

为防止滥用，你应该考虑对您的 API 限流。 例如，您可以限制每个用户 10 分钟内最多调用 API 100 次。 如果在规定的时间内接收了一个用户大量的请求，将返回响应状态代码 429 (这意味着过多的请求)。

要启用限流, `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最后一次速率限制检查时相应的 UNIX 时间戳数。
- `saveAllowance()`: 保存剩余的允许请求数和当前的 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。

Implementation in the `User` model could look like the following:

```
public function getRateLimit($request, $action)
{
    return [$this->rateLimit, 1]; // $rateLimit requests per second
}

public function loadAllowance($request, $action)
{
    return [$this->allowance, $this->allowance_updated_at];
}

public function saveAllowance($request, $action, $allowance, $timestamp)
{
    $this->allowance = $allowance;
    $this->allowance_updated_at = $timestamp;
    $this->save();
}
```

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`。

你可以参考以下代码 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。

11.8 版本

一个好的 API 设计应该是“版本化”的：变更和新的功能应该在 API 新版本中实现，而不是在一个版本上持续更改。与Web应用程序不同，您可以完全控制客户端和服务器端 代码，APIs 是为了给超出控制的客户端使用。因此， 应该尽可能的保持向后兼容性，如果有一些变化不能向后兼容，你应该在新版本的 API 中采用它同时增加版本号。现有客户端可以继续使用旧版本的 API；新的或升级的客户端可以在新的 API 版本中获得新的功能。

> **提示：** 可以参考 Semantic Versioning[8] 来获取更多关于设计 API 版本号的信息

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， `https://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。

[8] https://semver.org/

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
            Module.php
        v2/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
            Module.php
```

你的应用程序配置应该这样:

```
return [
    'modules' => [
        'v1' => [
            'class' => 'app\modules\v1\Module',
        ],
        'v2' => [
           'class' => 'app\modules\v2\Module',
        ],
    ],
    '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，`https://example.com/v1/users` 将返回版本 1 的用户列表， 而 `https://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 `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，内容交涉后，`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。

11.9 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

{
    "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。

11.9.1 自定义错误响应

有时你可能想自定义默认的错误响应格式。例如，你想一直使用HTTP状态码200， 而不是依赖于使用不同的HTTP状态来表示不同的错误， 并附上实际的HTTP状态代码为JSON结构的一部分的响应，就像以下所示，

```
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
    }
}
```

为了实现这一目的，你可以响应该应用程序配置的 `response` 组件的 `beforeSend` 事件：

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null &&
                !empty(Yii::$app->request->get('suppress_response_code'))) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
```

```
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

当 suppress_response_code 作为 GET 参数传递时，上面的代码 将重新按照自己定义的格式响应（无论失败还是成功）。

Chapter 12

开发工具（Development Tools）

Chapter 13

测试（Testing）

13.1 测试

测试是软件开发的一个重要组成部分。不管我们是否意识到，我们一直在不断地进行测试。 例如，当我们在用 PHP 写一个类的时候，我们可能用 echo 或者 die 语句一步一步简单的调试 验证我们实现的代码是否按照最初的计划工作。在开发 web 应用的时候，我们在表单中输入 一些测试数据来确保页面能够如预期那样和我们进行交互。

测试过程可能是自动的，所以每次我们需要验证的时候，我们只需要调用它就可以测试代码了。 验证代码执行结果是否符合我们的计划叫做测试， 测试过程的创建以及进一步执行叫做自动化测试， 这是这些测试章节的主要主题。

13.1.1 带着测试进行开发

测试驱动开发（TDD）和行为驱动开发（BDD）在开始编写实际代码之前， 首先通过描述一段代码的行为或将其作为一组场景或测试的全部特征， 然后创建符合这些测试预期验证 的行为实现。

开发一个功能的过程如下：

- 创建一个描述一个功能被实现测试。
- 运行这个测试来确保功能失败，因为这是没有实现之前的预期。
- 编写简单代码确保这个测试通过。
- 运行所有测试确保所有测试都通过。
- 优化代码确保测试依然可以通过。

走完上面的过程之后，为其他功能或者扩展重复上面测试过程。如果功能发生变化，测试也需 要跟着变化。

> **提示：** 如果你觉得你做一些很小很简单的迭代是在浪费时间，请尝试覆盖更多的测试 场景，这样你就可以在执行测试之前做更多的尝试。如果你的调试过多，试着做相反的工作。

在做一些具体的实现之前创建测试的原因是，这允许我们后期专注于我们

想要的实现，并且 可以花费更多的精力到实现细节。在涉及功能调整的时候，这会使得抽象更合理、测试维护 更简单或者使得耦合元件更少。

这种做法的优点如下：

- 在计划和实现发生变更的时候，可以让你在同一时间只专注于一件事情。
- 更多功能更详细的覆盖测试的结果，如果测试都通过好比再也没有什么问题了。

在很长一段时间内，这通常会给你提供一个有效的时间节省。

13.1.2 什么时候测试，怎么测试?

在测试的时候，对于一些相对复杂的项目上面的内容是非常有意义的，但对于一些比较 简单的项目就做的有些极端了。适用场景如下：

- 项目已经很大且复杂。
- 项目需求开始变得复杂起来。项目不断发展。
- 项目历时很长。
- 失败的代价非常高。

在现有的实现行为中进行覆盖测试是非常适合的。

- 项目是一个逐步更新的遗产。
- 你有一个还没有经过测试的项目要做。

在一些情况下，任何形式的自动化测试都是过于极端的：

- 项目很简单，也不会变得复杂。
- 过期不再工作的一次性项目。

假如你有很多的时间，在这种情况下进行自动测试也很好。

13.1.3 深度阅读

- 测试驱动开发：示例 / Kent Beck。 ISBN：0321146530。

13.2 测试环境设置

Yii 2 官方兼容 `Codeception`[1] 测试框架， 你可以创建以下类型的测试：

- 单元测试 - 验证一个独立的代码单元是否按照期望的方式运行；
- 功能测试 - 在浏览器模拟器中以用户视角来验证期望的场景是否发生
- 验收测试 - 在真实的浏览器中以用户视角验证期望的场景是否发生。

Yii 为包括 `yii2-basic`[2] 和 `yii2-advanced`[3] 在内的应用模板脚手架提供全部三种类型的即用测试套件。

Codeception 预装了基本和高级项目模板。 如果您没有使用这些模板中的一个，则可以安装 Codeception 通过输入以下控制台命令：

[1] https://github.com/Codeception/Codeception
[2] https://github.com/yiisoft/yii2-app-basic
[3] https://github.com/yiisoft/yii2-app-advanced

```
composer require codeception/codeception
composer require codeception/specify
composer require codeception/verify
```

13.3 单元测试

单元测试验证单个代码单元是否按预期工作。 也就是说，给定不同的输入参数， 测试验证类方法返回预期结果。 单元测试通常由编写待测试类的人开发。

Yii的单元测试框架 Codeception 基于 PHPUnit，Codeception 建议遵从 PHPUnit 的文档的进行开发：

- Codeception for Yii framework[4]
- Codeception Unit Tests[5]
- PHPUnit docs starting from chapter 2[6]

13.3.1 运行基本和高级模板测试

如果您已经开始使用高级模板，请参阅 “testing” guide[7] 有关运行测试的更多细节。

如果您已经开始使用基本模板，请参阅 README “testing” section[8]。

13.3.2 框架单元测试

如果你想运行 Yii 框架的单元测试 “Getting started with Yii 2 development[9]”。

13.4 功能测试

功能测试从用户的角度验证场景。它类似于验收测试。 但是它不是通过 HTTP 进行通信，而是填充环境，如（填充）POST 和 GET 参数， 然后直接在代码里执行 Application 实例。

功能测试通常比验收测试快，并且在失败时提供详细的堆栈跟踪。 根据老司机的经验，功能测试应该是首选的，除非有专门的 Web 服务器设置 或者由 JavaScript 构建的复杂 UI。

功能测试是借助于具有良好文档的 Codeception 框架来实现的：

[4] https://codeception.com/for/yii
[5] https://codeception.com/docs/05-UnitTests
[6] https://phpunit.readthedocs.io/en/9.5/writing-tests-for-phpunit.html
[7] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md
[8] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing
[9] https://github.com/yiisoft/yii2/blob/master/docs/internals/getting-started.md

- Codeception for Yii framework[10]
- Codeception 的功能测试[11]

13.4.1 运行基本模板和高级模板的测试

如果你使用的是高级模板，请参阅测试指南[12] 中关于运行测试的更多细节。

如果你使用的是基本模板，请检查其 README “testing” 部分[13]。

13.5 验收测试

验收测试从用户角度验证场景。 测试的应用程序可以通过 PhpBrowser 或者 真正的浏览器。 在这两种情况下，浏览器都通过 HTTP 进行通信，因此应用程序应通过Web服务器提供。

验证测试是在 Codeception 框架的帮助下实现的，该框架有一个很好的文档:

- Codeception for Yii framework[14]
- Codeception Acceptance Tests[15]

13.5.1 运行基本和高级模板测试

如果您已经开始使用高级模板，请参阅 “测试”指南[16] 有关运行测试的更多细节。

如果您已经开始使用基本模板，请查看其 README“测试”部分[17]。

13.6 Fixtures

Fixtures 是测试中非常重要的一部分。他们的主要目的是建立一个固定/已知的环境状态以确保 测试可重复并且按照预期方式运行。Yii 提供一个简单可用的 Fixure 框架 允许你精确的定义你的 Fixtures 。

Yii 的 Fixture 框架的核心概念称之为 *fixture object* 。一个 Fixture object 代表 一个测试环境的某个特定方面，它是 `yii\test\Fixture` 或者其子类的实例。 比如，你可以使用 `UserFixture` 来确保用户DB表包含固定的数据。 你在运行一个测试之前加载一个或者多个 fixture object，并在结束后卸载他们。

[10] https://codeception.com/for/yii
[11] https://codeception.com/docs/04-FunctionalTests
[12] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md
[13] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing
[14] https://codeception.com/for/yii
[15] https://codeception.com/docs/03-AcceptanceTests
[16] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md
[17] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

一个 Fixture 可能依赖于其他的 Fixtures，通过它的 `yii\test\Fixture::$depends` 来指定。当一个 Fixture 被加载前，它依赖的 Fixture 会被自动的加载；同样，当某个 Fixture 被卸载后，它依赖的 Fixtures 也会被自动的卸载。

13.6.1 定义一个 Fixture

为了定义一个 Fixture，你需要创建一个新的 class 继承自 `yii\test\Fixture` 或者 `yii\test\ActiveFixture`。前一个类对于一般用途的 Fixture 比较适合，而后者则有一些增强功能专用于与数据库和 ActiveRecord 一起协作。

下面的代码定义一个关于 `User` ActiveRecord 和相关的用户表的 Fixture:

```
<?php
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture
{
    public $modelClass = 'app\models\User';
}
```

> **提示：** 每个 `ActiveFixture` 都会准备一个 DB 表用来测试。你可以通过设置 `yii\test\ActiveFixture::$tableName` 或 `yii\test\ActiveFixture::$modelClass` 属性来指定具体的表。如果是后者，表名会从 `modleClass` 指定的 `ActiveRecord` 中获取。
>
> **注意：** `yii\test\ActiveFixture` 仅限于 SQL 数据库，对于 NoSQL 数据库，Yii 提供以下 `ActiveFixture` 类:
>
> - Mongo DB: `yii\mongodb\ActiveFixture`
> - Elasticsearch: `yii\elasticsearch\ActiveFixture`（从版本 2.0.2 开始）

提供给 `ActiveFixture` 的 fixture data 通常放在一个路径为 `FixturePath/data/TableName.php` 的文件中，其中 `FixturePath` 代表 Fixture 类所在的路径，`TableName` 则是和 Fixture 关联的表。在以上的例子中，这个文件应该是 `@app/tests/fixtures/data/user.php`。data 文件返回一个包含要被插入用户表中的数据文件，比如:

```
<?php
return [
    'user1' => [
        'username' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' =>
        '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
```

```
    'user2' => [
        'username' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' =>
        '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

你可以给某行指定一个 alias 别名，这样在你以后的测试中，你可以通过别名来确定某行。 在上面的例子中，这两行指定别名为 `user1` 和 `user2`。

同样，你不需要特别的为自动增长（auto-incremental）的列指定数据，Yii 将会在 Fixture 被加载时自动的填充正确的列值到这些行中。

> **提示：** 你可以通过设置 `yii\test\ActiveFixture::$dataFile` 属性来自定义 data 文件的位置。 同样，你可以重写 `yii\test\ActiveFixture::getData()` 来提供数据。

如之前所述，一个 Fixture 可以依赖于其他的 Fixture 。比如一个 `UserProfileFixture` 可能需要依赖于 `UserFixture`， 因为 user profile 表包括一个指向 user 表的外键。那么， 这个依赖关系可以通过 `yii\test\Fixture::$depends` 属性来指定，比如如下：

```
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserProfileFixture extends ActiveFixture
{
    public $modelClass = 'app\models\UserProfile';
    public $depends = ['app\tests\fixtures\UserFixture'];
}
```

依赖关系确保所有的 Fixtures 能够以正常的顺序被加载和卸载。在以上的例子中， 为确保外键存在， `UserFixture` 会在 `UserProfileFixture` 之前加载，同样，也会在其卸载后同步卸载。

在上面，我们展示了如何定义一个 DB 表的 Fixture 。为了定义一个与 DB 无关的 Fixture （比如一个fixture关于文件和路径的），你可以从一个更通用的基类 `yii\test\Fixture` 继承， 并重写 `load()` 和 `unload()` 方法。

13.6.2 使用 Fixtures

如果你使用 CodeCeption[18] 作为你的 Yii 代码测试框架， 你需要考虑使用 `yii2-codeception` 扩展，这个扩展包含内置的机制来支持加载和访问 Fixtures。 如果你使用其他的测试框架，为了达到加载和访问 Fixture 的目的， 你需要在你的测试用例中使用 `yii\test\FixtureTrait`。

[18] https://codeception.com/

在以下示例中，我们会展示如何通过 `yii2-codeception` 写一个 `UserProfile` 单元来测试某个 class。

在一个继承自 `yii\codeception\DbTestCase` 或者 `yii\codeception\TestCase` 的单元测试类中， 你可以在 `fixtures()` 方法中声明你希望使用哪个 Fixture。比如:

```
namespace app\tests\unit\models;


use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends \Codeception\Test\Unit
{
    public function _fixtures()
    {
        return [
            'profiles' => [
                'class' => UserProfileFixture::class,
                // fixture data located in tests/_data/user.php
                'dataFile' => codecept_data_dir() . 'user.php'
            ],
        ];
    }

    // ...test methods...
}
```

在测试用例的每个测试方法运行前 `fixtures()` 方法列表返回的 Fixture 会被自动的加载， 并在结束后自动的卸载。同样，如前面所述，当一个 Fixture 被加载之前， 所有它依赖的 Fixture 也会被自动的加载。在上面的例子中，因为 `UserProfileFixture` 依赖于 `UserFixtrue`，当运行测试类中的任意测试方法时，两个 Fixture，`UserFixture` 和 `UserProfileFixture` 会被依序加载。

当我们通过 `fixtures()` 方法指定需要加载的 Fixture 时，我们既可以使用一个类名， 也可以使用一个配置数组。 配置数组可以让你自定义加载的 fixture 的属性名。

你同样可以给一个 Fixture 指定一个别名（alias），在上面的例子中，`UserProfileFixture` 的别名为 `profiles` 。 在测试方法中，你可以通过别名来访问一个 Fixture 对象。比如，

```
$profile = $I->grabFixture('profiles', 'user1');
```

将会返回 `UserProfileFixture` 对象。

因为 `UserProfileFixture` 从 `ActiveFixture` 处继承， 在后面，你可以通过如下的语法形式来访问 Fixture 提供的数据:

```
// 返回对应于别名为"user1"的数据行的 UserProfile 模型
$profile = $I->grabFixture('profiles', 'user1');
// 遍历 fixture 中的数据
foreach ($I->grabFixture('profiles') as $profile) ...
```

13.6.3 组织 Fixture 类和相关的数据文件

默认情况下，Fixture 类会在其所在的目录下面的 `data` 子目录寻找相关的数据文件。 在一些简单的项目中，你可以遵循此范例。对于一些大项目， 您可能经常为同一个 Fixture 类的不同测试而切换不同的数据文件。 在这种情况下，我们推荐你按照一种类似于命名空间的方式有层次地组织你的数据文件，比如：

```
# under folder tests\unit\fixtures

data\
    components\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
    models\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
# and so on
```

这样，你就可以避免在测试用例之间产生冲突，并根据你的需要使用它们。

> **注意：** 在以上的例子中，Fixture 文件只用于示例目的。在真实的环境下，你需要根据你的 Fixture 类继承的基类来决定它们的命名。 比如，如果你从 `yii\test\ActiveFixture` 继承了一个 DB Fixture， 你需要用数据库表名字作为 Fixture 的数据文件名；如果你从 `yii\mongodb\ActiveFixture` 继承了一个 MongoDB Fixture， 你需要使用 collection 名作为文件名。

组织 Fixuture 类名的方式同样可以使用前述的层次组织法，但是，为了避免跟数据文件产生冲突， 你需要用 `fixtures` 作为根目录而非 `data`。

13.6.4 使用 来管理 fixtures

Yii 通过 `yii fixture` 命令行工具来支持 fixtures 操作. 这个工具支持:

- 将 fixtures 装载到不同的存储设备，例如：RDBMS, NoSQL 等;
- 以不同方式卸载 fixtures（通常是清理存储）;
- 自动生成 fixtures 并用随机数据填充。

Fixtures 数据格式

让我们假设我们有要加载的 fixtures 数据：

`#users.php` 文件在 `fixtures` 下的数据路径，默认为 `@tests\unit\fixtures\data`

```
return [
    [
        'name' => 'Chase',
        'login' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' =>
        '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
    [
        'name' => 'Celestine',
        'login' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' =>
        '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

如果我们使用 fixture 将数据加载到数据库中，那么这些行将作用到 `users` 表。 如果我们使用的是 nosql 类型的 fixtures，例如 `mongodb` fixture然后这个数据将应用于 `users` mongodb 集合。 要了解有关实现各种加载策略的信息，请参阅官方文档[19]。

上面的 fixture 示例是由 `yii2-faker` 扩展自动生成的，在这些章节中可以了解更多相关内容。 Fixture 类名称不应为复数。

加载 Fixtures

Fixture 类应该以 `Fixture` 类作为后缀。默认的 Fixtures 能在 `tests\unit\fixtures` 命名空间下被搜索到， 你可以通过配置和命名行选项来更改这个行为，你可以通过加载或者卸载指定它名字前面的 `-` 来排除一些 Fixtures，像 `-User`。

运行如下命令去加载 Fixture:

> **注意:** 卸载数据事件优先执行，在加载数据之前。通常，这将会清除先前的 fixture 所插入的所有现有数据。

```
yii fixture/load <fixture_name>
```

必需参数 `fixture_name` 指定一个将被加载数据的 Fixture 名字。 你可以同时加载多个 Fixtures 。 以下是这个命令的正确格式:

```
// 加载 `User` fixture
yii fixture/load User

// 与上面相同，因为 ‘‘fixture’’ 命令的默认动作是‘‘加载’’
yii fixture User
```

[19] https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixtures.md

```
// 加载几个 fixtures
yii fixture User UserProfile

// 加载所有 fixtures
yii fixture/load "*"

// 与上面相同
yii fixture "*"

// 加载所有 fixtures 除了这些
yii fixture "*" -DoNotLoadThisOne

// 加载 fixtures, 但在不同的命名空间中搜索它们。默认命名空间
是: tests\unit\fixtures。
yii fixture User --namespace='alias\my\custom\namespace'

// 加载全局 fixture `some\name\space\CustomFixture` 在加载其他 fixtures 之前.
// 默认情况下, 此选项设置为 `InitDbFixture` 以禁用/启用完整性检查。您可以用用
逗号分离来
// 指定几个全局 fixtures。
yii fixture User --globalFixtures='some\name\space\Custom'
```

卸载 Fixtures

运行如下命名去卸载 Fixtures:

```
// 卸载 User fixture, 默认情况下它将清除 fixture 存储（例如“用户”表, 或“用
户”集合如果这是 mongodb fixture）。
yii fixture/unload User

// 卸载几个 fixtures
yii fixture/unload User,UserProfile

// 卸载所有 fixtures
yii fixture/unload "*"

// 卸载所有 fixtures 除了这些
yii fixture/unload "*" -DoNotUnloadThisOne
```

同样的命名选项: `namespace`, `globalFixtures` 也可以应用于该命令。

全局命令配置

当命令行选项允许我们配置迁移命令， 有时我们只想配置一次。例如， 你可以按照如下配置迁移目录:

```
'controllerMap' => [
    'fixture' => [
        'class' => 'yii\console\controllers\FixtureController',
        'namespace' => 'myalias\some\custom\namespace',
        'globalFixtures' => [
            'some\name\space\Foo',
```

```
            'other\name\space\Bar'
        ],
    ],
]
```

自动生成 fixtures

Yii 还可以为你自动生成一些基于一些模板的 Fixtures。 你能够以不同语言格式用不同的数据生成你的 Fixtures。 这些特征由 Faker[20] 库和 `yii2-faker` 扩展完成。 关注 guide[21] 扩展获取更多的文档。

13.6.5 总结

在上面，我们描述了如何定义和使用 Fixture，在下面，我们将总结出一个标准地运行与 DB 有关的单元测试的规范工作流程:

1. 使用 `yii migrate` 工具来让你的测试数据库更新到最新的版本;
2. 运行一个测试:
   - 加载 Fixture：清空所有的相关表结构，并用 Fixture 数据填充
   - 执行真实的测试用例
   - 卸载 Fixture
3. 重复步骤 2 直到所有的测试结束。

[20] https://github.com/fzaninotto/Faker
[21] https://github.com/yiisoft/yii2-faker

Chapter 14

高级专题（Special Topics）

14.1 创建你自己的应用程序结构

注：本章节正在开发中。

虽然 basic[1] 和 advanced[2] 项目模板能够满足你的大部分需求，但是， 你仍有可能需要创建你自己的项目模板来开始项目。

Yii 的项目模板是一个包含 `composer.json` 文件的仓库，并被注册为一个 Composer package。任何一个仓库都可以被标识为一个 Composer package， 只要让其可以通过 `create-project` Composer 命令安装。

由于完全从新创建一个你自己的模板工作量有点大，最好的方式是以一个内建模板为基础。 这里，我们使用基础应用模板。

14.1.1 克隆基础模板

第一步是从 Git 仓库克隆 Yii 的基础模板：

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

等待仓库下载到你的电脑。因为为调整到你自己的模板所产生的修改不会被 push 回， 你可以删除下载下来的 `.git` 目录及其内容。

14.1.2 修改文件

接下来，你需要修改 `composer.json` 以配置你自己的模板。修改 `name`, `description`, `keywords`, `homepage`, `license`, 和 `support` 的值来描述你自己的模板。 同样，调整 `require`, `require-dev`, `suggest` 和其他的参数来匹配你模板的环境需求。

> **注意：** 在 `composer.json` 文件中，使用 `extra` 下的 `writeable` 参数来指定使用模板创建的应用程序后 需要设置文件权限的文件列表。

[1] https://github.com/yiisoft/yii2-app-basic
[2] https://github.com/yiisoft/yii2-app-advanced

接下来，真正的修改你的应用程序默认的目录结构和内容。 最后，更新 README 文件以符合你的模板。

14.1.3 发布一个 Package（Make a Package）

模板调整好后，通过其创建一个 Git 仓库并提交你的代码。 如果你希望将你的应用程序模板开源，Github[3] 将是最好的托管服务。 如果你不喜欢其他的人来跟你一起协作，你可以使用任意的 Git 仓库服务。

接下来，你需要为 Composer 注册你的 package。对于公有的模板，你可以将 package 注册到 Packagist[4]。对于私有的模板， 注册 package 将会麻烦一点。 参考 Composer documentation[5] 获取更多的指示。

14.1.4 使用模板

以上就是为了创建一个新的 Yii 项目模板你需要做的事情。现在，你可以使用你自己的模板创建项目了：

```
composer create-project --prefer-dist --stability=dev
mysoft/yii2-app-coolone new-project
```

14.2 控制台命令

除了用于构建 Web 应用程序的丰富功能，Yii 中也有一个拥有丰富功能的控制台， 它们主要用于创建网站后台处理的任务。

控制台应用程序的结构非常类似于 Yii 的一个 Web 应用程序。 它由一个或多个 `yii\console\Controller` 类组成，它们在控制台环境下通常被称为“命令”。 每个控制器还可以有一个或多个动作，就像 web 控制器。

两个项目模板（基础模版和高级模版）都有自己的控制台应用程序。 你可以通过运行 `yii` 脚本，在位于仓库的基本目录中运行它。 当你不带任何参数来运行它时，会给你一些可用的命令列表：

[3] https://github.com
[4] https://packagist.org/
[5] https://getcomposer.org/doc/05-repositories.md#hosting-your-own

```
$ ./yii

This is Yii version 2.0.4-dev.

The following commands are available:

- asset                        Allows you to combine and compress your JavaScript and CSS files.
    asset/compress (default)   Combines and compresses the asset files according to the given
                               configuration.
    asset/template             Creates template of configuration file for [[actionCompress]].

- cache                        Allows you to flush cache.
    cache/flush                Flushes given cache components.
    cache/flush-all            Flushes all caches registered in the system.
    cache/flush-schema         Clears DB schema cache for a given connection component.
    cache/index (default)      Lists the caches that can be flushed.

- fixture                      Manages fixture data loading and unloading.
    fixture/load (default)     Loads the specified fixture data.
    fixture/unload             Unloads the specified fixtures.

- help                         Provides help information about console commands.
    help/index (default)       Displays available commands or the detailed information

- message                      Extracts messages to be translated from source files.
    message/config             Creates a configuration file for the "extract" command.
    message/extract (default)  Extracts messages to be translated from source code.

- migrate                      Manages application migrations.
    migrate/create             Creates a new migration.
    migrate/down               Downgrades the application by reverting old migrations.
    migrate/history            Displays the migration history.
    migrate/mark               Modifies the migration history to the specified version.
    migrate/new                Displays the un-applied new migrations.
    migrate/redo               Redoes the last few migrations.
    migrate/to                 Upgrades or downgrades till the specified version.
    migrate/up (default)       Upgrades the application by applying new migrations.


To see the help of each command, enter:

  yii help <command-name>
```

正如你在截图中看到，Yii 中已经定义了一组默认情况下可用的命令：

- `AssetController` - 允许合并和压缩你的 JavaScript 和 CSS 文件。在 资源 - 使用 asset 命令 一节可获取更多信息。
- `CacheController` - 清除应用程序缓存。
- `FixtureController` - 管理用于单元测试 fixture 的加载和卸载。 这个命令的更多细节在 Testing Section about Fixtures.
- `HelpController` - 提供有关控制台命令的帮助信息， 这是默认的命令并会打印上面截图所示的输出。
- `MessageController` - 从源文件提取翻译信息。 要了解更多关于这个命令的用法，请参阅 I18N 章节.
- `MigrateController` - 管理应用程序数据库迁移。 在 数据库迁移章节 可获取更多信息。
- `ServeController` - Allows you run PHP built-in web server.

14.2.1 用法

你可以使用以下语法来执行控制台控制器动作：

```
yii <route> [--option1=value1 --option2=value2 ... argument1 argument2 ...]
```

以上，`<route>` 指的是控制器动作的路由。选项将填充类属性， 参数是动作

方法的参数。

例如，将 `MigrateController::actionUp()` 限制 5 个数据库迁移并将 `MigrateController::$migrationTable` 设置为 `migrations` 应该这样调用：

```
yii migrate/up 5 --migrationTable=migrations
```

> **注意：** 当在控制台使用 `*` 时，不要忘记像 `"*"` 一样用引号来引起来，为了防止在 shell 中执行命令时被当成当前目录下的所有文件名。

14.2.2 入口脚本

控制台应用程序的入口脚本相当于用于 Web 应用程序的 `index.php` 入口文件。控制台入口脚本通常被称为 `yii`，位于应用程序的根目录。它包含了类似下面的代码：

```php
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */

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

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

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

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

该脚本将被创建为你应用程序中的一部分；你可以根据你的需求来修改它。如果你不需要记录错误信息或者希望提高整体性能，`YII_DEBUG` 常数应定义为 `false`。在基本的和高级的两个应用程序模板中，控制台应用程序的入口脚本在默认情况下会启用调试模式，以提供给开发者更好的环境。

14.2.3 配置

在上面的代码中可以看到，控制台应用程序使用它自己的配置文件，名为 `console.php` 。在该文件里你可以给控制台配置各种 应用组件 和属性。

如果你的 web 应用程序和控制台应用程序共享大量的配置参数和值，你可以考虑把这些值放在一个单独的文件中，该文件中包括（ web 和控制台）应用程序配置。你可以在“高级”项目模板中看到一个例子。

> **提示：** 有时，你可能需要使用一个与在入口脚本中指定的应用程序配置不同的控制台命令。例如，你可能想使用 `yii migrate`

命令来升级你的测试数据库，它被配置在每个测试套件。 要动态地更改配置，只需指定一个自定义应用程序的配置文件， 通过 appconfig选项来执行命令：

```
yii <route> --appconfig=path/to/config.php ...
```

14.2.4 控制台命令完成

自动完成命令参数在使用 shell 时非常有用。 从版本 2.0.11 开始，./yii 命令为 Bash 和 ZSH 提供了自动完成功能。

Bash 完成

确保安装完毕。对于大多数安装，它默认是可用的。

放置完成脚本在 /etc/bash_completion.d/:

```
curl -L
https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/completion/bash/yii
-o /etc/bash_completion.d/yii
```

对于临时使用，您可以将文件放入当前目录，并通过 source yii 将其包含在当前会话中。 如果全局安装，您可能需要重新启动终端或source ~/.bashrc 来激活它。

查看 Bash 手册[6] 了解将完成脚本添加到您的环境的其他方法。

ZSH 完成

将完成脚本放入完成目录中，例如使用 ~/.zsh/completion/

```
mkdir -p ~/.zsh/completion
curl -L
https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/completion/zsh/_yii
-o ~/.zsh/completion/_yii
```

将目录包含在 $fpath 中，例如，通过将其添加到 ~/.zshrc 中

```
fpath=(~/.zsh/completion $fpath)
```

确保 compinit 被加载或通过在 ~/.zshrc 中加入完成

```
autoload -Uz compinit && compinit -i
```

然后重新加载您的 shell

```
exec $SHELL -l
```

[6] https://www.gnu.org/software/bash/manual/html_node/Programmable-Completion.html

14.2.5 创建你自己的控制台命令

控制台的控制器和行为

一个控制台命令继承自 `yii\console\Controller` 控制器类。 在控制器类中，定义一个或多个与控制器的子命令相对应的动作。在每一个动作中，编写你的代码实现特定的子命令的适当的任务。

当你运行一个命令时，你需要指定一个控制器的路由。 例如，路由 `migrate/create` 调用子命令对应的`MigrateController::actionCreate()` 动作方法。 如果在执行过程中提供的路由不包含路由 ID ， 将执行默认动作（如 web 控制器）。

选项

通过覆盖在 `yii\console\Controller::options()` 中的方法， 你可以指定可用于控制台命令（controller/actionID）选项。这个方法应该返回控制器类的公共属性的列表。 当运行一个命令，你可以指定使用语法 `--OptionName=OptionValue` 选项的值。 这将分配 `OptionValue` 到控制器类的 `OptionName` 属性。

如果选项的默认值是数组类型，并且在运行该命令时设置了该选项， 通过在任何逗号分割输入字符串将选项值转换为数组。

选项别名

从版本 2.0.8 起控制台命令提供 `yii\console\Controller::optionAliases()` 方法来为选项添加别名。

要定义别名，请在控制器中覆盖 `yii\console\Controller::optionAliases()`，例如:

```
namespace app\commands;

use yii\console\Controller;

class HelloController extends Controller
{
    public $message;

    public function options($actionID)
    {
        return ['message'];
    }

    public function optionAliases()
    {
        return ['m' => 'message'];
    }

    public function actionIndex()
    {
```

```
        echo $this->message . "\n";
    }
}
```

现在，您可以使用以下语法来运行该命令：

```
./yii hello -m=hello
```

参数

除了选项，命令还可以接收参数。参数将传递给请求的子命令对应的动作方法。 第一个参数对应第一个参数，第二个参数对应第二个参数，依次类推。 命令被调用时，如果没有足够的参数，如果有定义默认值的情况下，则相应的参数将采取默认声明的值； 如果没有设置默认值，并且在运行时没有提供任何值，该命令将以一个错误退出。

你可以使用 `array` 类型提示来指示一个参数应该被视为一个数组。 该数组通过拆分输入字符串的逗号来生成。

下面的示例演示如何声明参数：

```
class ExampleController extends \yii\console\Controller
{
    // 命令 "yii example/create test" 会调用 "actionCreate('test')"
    public function actionCreate($name) { ... }

    // 命令 "yii example/index city" 会调用 "actionIndex('city', 'name')"
    // 命令 "yii example/index city id" 会调用 "actionIndex('city', 'id')"
    public function actionIndex($category, $order = 'name') { ... }

    // 命令 "yii example/add test" 会调用 "actionAdd(['test'])"
    // 命令 "yii example/add test1,test2" 会调用 "actionAdd(['test1',
    'test2'])"
    public function actionAdd(array $name) { ... }
}
```

退出代码

使用退出代码是控制台应用程序开发的最佳做法。通常，执行成功的命令会返回 `0`。 如果命令返回一个非零数字，会认为出现错误。 该返回的数字作为出错代码，用以了解错误的详细信息。例如 `1` 可能代表一个未知的错误， 所有的代码都将保留在特定的情况下：输入错误，丢失的文件等等。

要让控制台命令返回一个退出代码， 只需在控制器动作方法中返回一个整数：

```
public function actionIndex()
{
    if (/* some problem */) {
        echo "A problem occured!\n";
        return 1;
    }
    // do something
```

```
    return 0;
}
```

有一些预定义的常量可以使用。在类 `yii\console\ExitCode` 中被定义：

```
public function actionIndex()
{
    if (/* some problem */) {
        echo "A problem occurred!\n";
        return ExitCode::UNSPECIFIED_ERROR;
    }
    // do something
    return ExitCode::OK;
}
```

为控制器定义有意义的常量，以防有更多的错误代码类型，这会是一个很好的实践。

格式和颜色

Yii 支持格式化输出， 如果终端运行命令不支持的话则会自动退化为非格式化输出。

要输出格式的字符串很简单。以下展示了如何输出一些加粗的文字：

```
$this->stdout("Hello?\n", Console::BOLD);
```

如果你需要建立字符串动态结合的多种样式，最好使用 `ansiFormat` ：

```
$name = $this->ansiFormat('Alex', Console::FG_YELLOW);
echo "Hello, my name is $name.";
```

表格

从版本 2.0.13 开始，有一个小部件允许您在控制台中格式化表数据。使用方法如下：

```
echo Table::widget([
    'headers' => ['Project', 'Status', 'Participant'],
    'rows' => [
        ['Yii', 'OK', '@samdark'],
        ['Yii', 'OK', '@cebe'],
    ],
]);
```

有关详细信息，请参阅 `API documentation`.

14.3 核心验证器（Core Validators）

Yii 提供一系列常用的核心验证器，位于 `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**别名**来指定相应的核心验证器。 比如你可以用 `required` 别名代指 `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别名。

下面，我们将详细介绍每一款验证器的主要用法和属性。

14.3.1 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 `strict` 属性为假。

14.3.2 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 `CAPTCHA` 小部件所显示的验证代码（verification code）相同。

- `caseSensitive`： 对验证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 `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。

14.3.3 compare（比对）

```
[
    // 检查 "password" 属性的值是否与 "password_repeat" 的值相同
    ['password', 'compare'],

    // 和上一个相同，只是明确指定了需要对比的属性字段
    ['password', 'compare', 'compareAttribute' => 'password_repeat'],

    // 检查年龄是否大于等于 30
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],
]
```

该验证器比对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：用于与原属性相比对的属性名称。 当该验证器被用于验证某目标属性时， 该属性会默认为目标属性加后缀 _repeat。 举例来说，若目标属性为 password，则该属性默认为 password_repeat。
- compareValue：用于与输入值相比对的常量值。 当该属性与 compareAttribute 属性同时被指定时，该属性优先被使用。
- operator：比对操作符。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。
- type：默认的比对类型是'string'，此时将按照字节逐个对比。 当需要比对的值是数字时，需要设置类型$type为 'number'，启用数字对比模式。

比对日期值

比对验证器只能用来对比字符串和数字。如果你需要比对日期，有两种方式。 如果需要比对一个固定的日期值，只需要使用 date验证器并设置对应的属性 $min 或 $max 。 如果需要比对如表单提交的两个日期，比如一个fromDate和一个toDate项， 你可以结合比对验证器和日期验证器同时使用，如下所示：

```
['fromDate', 'date', 'timestampAttribute' => 'fromDate'],
['toDate', 'date', 'timestampAttribute' => 'toDate'],
```

```
['fromDate', 'compare', 'compareAttribute' => 'toDate', 'operator' => '<',
'enableClientValidation' => false],
```

因为验证器会按照顺序执行， 将首先验证 `fromDate` 和 `toDate` 字段是一个有效的日期值，最终将被转换成一个系统可识别的格式。 此后这两个值将使用比对验证器进行比对。 因日期验证器只提供服务端使用，当前不提供客户端验证， 故 `$enableClientValidation` 在比对验证器将同样被设置为 `false` 。

14.3.4 date（日期）

此日期 `date` 验证器有三种不同的使用方式:

```
[
    [['from_date', 'to_date'], 'date'],
    [['from_datetime', 'to_datetime'], 'datetime'],
    [['some_time'], 'time'],
]
```

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 `timestampAttribute` 所指定的属性里。

- `format`: 被验证值的日期/时间格式。 这里的值可以是 ICU manual[7] 中定义的日期时间格式。 另外还可以设置以 `php:` 开头的字符串，用来表示PHP可以识别的日期时间格式。 `Datetime` 日期时间类。请参考 https://www.php.net/manual/zh/datetime.createfromformat.php 获取更多支持的格式。 如果没有设置，默认值将使用 `Yii::$app->formatter->dateFormat` 中的值。 请参考 `API` 文档 以获取更详细的说明。
- `timestampAttribute`: 输入的日期时间将被转换为时间戳后设置到的属性的名称。 可以设置为和被验证的属性相同。如果相同， 原始值将在验证结束后被时间戳覆盖。 请参考 “Handling date input with the DatePicker”[8] 以获取更多使用事例。
  从版本 2.0.4 开始，支持 使用 `$timestampAttributeFormat` 设置日期时间格式 和使用 `$timestampAttributeTimeZone` 设置时区。
  注意，如果使用 `timestampAttribute`，被验证的值将被转换为UTC标准的时间戳， 所以使用 `input time zone` 输入的时区将被转换为UTC时间。
- 自版本 2.0.4 开始支持 直接对 `minimum` 和 `maximum` 设置时间戳。

如果输入的值是可选的，可以设置一个 默认值验证器 来确保空值通过验证器验证后的值是 `null`， 否则数据库可能会保存类似 `0000-00-00` 的值， 或表单日期选择器会显示 `1970-01-01` 。

[7] https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax

[8] https://github.com/yiisoft/yii2-jui/blob/master/docs/guide/topics-date-picker.md

```
[
    [['from_date', 'to_date'], 'default', 'value' => null],
    [['from_date', 'to_date'], 'date'],
],
```

14.3.5 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，给为空的待测属性分配默认值。

- `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属性。PHP 回调方法的样式如下：

```
function foo($model, $attribute) {
    // ... 计算 $value ...
    return $value;
}
```

> **信息：** 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。

14.3.6 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。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。

14.3.7 each（循环验证）

> **信息：** 此验证器自版本 2.0.4 后可用。

```
[
    // 检查是否每个分类编号都是一个整数
    ['categoryIDs', 'each', 'rule' => ['integer']],
]
```

此验证器只能验证数组格式的属性。此验证器将判断数组中的 每个 元素是否都符合验证规则。 在上面的例子中，`categoryIDs` 属性必须是一个数组 且每个元素将被使用 `integer` 验证器进行验证。

- `rule`：保存验证规则的数组。数组中第一个元素表示验证器类名或 验证器的别名。数组中其余键值对将被用来配置此验证器规则。
- `allowMessageFromRule`：是否使用规则中指定的验证器返回的错误信息。默认值为 `true`。 如果设置为 `false`，将使用 `message` 作为错误信息。

  注意： 如果被验证的值不是一个数组，将被认为验证失败， 并且返回 `message` 设定的错误信息。

14.3.8 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。

14.3.9 exist（存在性）

```
[
    // a1 需要在 "a1" 属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],

    // type_id 需要存在于 ProductType 类中定义的表中的"id"列中
    ['type_id', 'exist', 'targetClass' => ProductType::class,
    'targetAttribute' => ['type_id' => 'id']],

    // 与前一个相同，但使用已定义的关联"type"
    ['type_id', 'exist', 'targetRelation' => 'type'],
]
```

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 类型的模型类属性起作用， 能支持对一个或多过字段的验证。

可以使用此验证器验证单个数据列或多个数据列 （如多个列不同的组合是否存在）。

- `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` 的模型属性。 若不设置，它会直接使用待测属性名（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属性名，值表示 `targetClass` 的待测数据源字段，而键表示当前模型的待测属性名。 若键和值相同，你可以只指定值。（如:`['a2']` 就代表 `['a2'=>'a2']`）
- `targetRelation`: since version 2.0.14 you can use convenient attribute `targetRelation`, which overrides the `targetClass` and `targetAttribute` attributes using specs from the requested relation.
- `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 `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。

14.3.10 `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 (如 “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 (如 “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。 请参考 common media types[9] 获取更多详细内容。
- `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`: 给定属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。

14.3.11 filter（过滤器）

```
[
    // 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;
    }],

    // 标准化 "phone" 使用方法 "normalizePhone"
    ['phone', 'filter', 'filter' => [$this, 'normalizePhone']],

    public function normalizePhone($value) {
        return $value;
    }
]
```

该验证器并不进行数据验证。而是给输入值应用一个过滤器， 并在验证后把它赋值回原属性变量。

[9] https://en.wikipedia.org/wiki/Media_type

- `filter`：用于定义过滤器的 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过滤器。默认为 false。 请注意如果过滤器不能处理数组输入，你就应该把该属性设为 true。 否则可能会导致 PHP Error 的发生。

  提示： 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。

  提示： 有许多的PHP方法结构和 `filter` 需要的结构一致。 比如使用类型转换方法 (intval[10]， boolval[11], ...) 来确保属性为指定的类型， 你可以简单的设置这些方法名而不是重新定义一个匿名函数：

  ```
  ['property', 'filter', 'filter' => 'boolval'],
  ['property', 'filter', 'filter' => 'intval'],
  ```

14.3.12 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。

14.3.13 ip（IP地址）

```
[
    // 检查 "ip_address" 是否为一个有效的 IPv4 或 IPv6 地址
    ['ip_address', 'ip'],

    // 检查 "ip_address" 是否为一个有效的 IPv6 地址或子网地址，
    // 被检查的值将被展开成为一个完整的 IPv6 表示方法。
    ['ip_address', 'ip', 'ipv4' => false, 'subnet' => null, 'expandIPv6' =>
    true],
```

[10] https://www.php.net/manual/zh/function.intval.php
[11] https://www.php.net/manual/zh/function.boolval.php

```
    // 检查 "ip_address" 是否为一个有效的 IPv4 或 IPv6 地址，
    // 允许地址存在一个表示非的字符 `!`
    ['ip_address', 'ip', 'negation' => true],
]
```

此验证器检查属性的值是否是一个有效的 IPv4/IPv6 地址或子网地址。 如果标准记法或 IPv6 扩展记法被启用，原始值将被改变。

此验证器有以下参数：

- `ipv4`: 是否启用 IPv4 地址检测。默认为 true 。
- `ipv6`: 是否启用 IPv6 地址检测。默认为 true。
- `subnet`: 是否启用 CIDR 子网检测，类似 `192.168.10.0/24`
  - `true` - 子网是必须的，如果不是标准 CIDR 格式将被拒绝
  - `false` - 地址不能有 CIDR
  - `null` - CIDR 是可选的

  默认值为 false。
- `normalize`: 是否在地址没有 CIDR 时添加 CIDR 最小值作为后缀 (IPv4 为 32、IPv6 为 128) 仅当 `subnet` 不为 `false`时使用。例如：
  - `10.0.1.5` 将被转换为标准的`10.0.1.5/32`
  - `2008:db0::1` 将被转换为标准的 `2008:db0::1/128`

  默认为 false。
- `negation`: 是否允许被检测地址在首位存在代表非的字符 `!` 。默认为 false。
- `expandIPv6`: 是否将简化的 IPv6 地址扩展为标准记法格式。 例如，`2008:db0::1` 将被扩展成 `2008:0db0:0000:0000:0000:0000:0000:0001`。默认为 false。
- `ranges`: 允许或禁止的 IPv4 或 IPv6 范围的数组。

  当此数组为空或此参数没有设置时，所有IP地址都被允许。 否则，将逐个对比此规则，当找到第一条适用时停止。 如果没有任何规则适用待验证的IP地址，IP地址将被禁止。

  例如： `php [

  ```
  'client_ip', 'ip', 'ranges' => [
      '192.168.10.128'
      '!192.168.10.0/24',
      'any' // 允许任何其它IP地址
  ]
  ```

  ]` 在这个例子中，除了 `192.168.10.0/24` 子网之外的所有 IPv4 和 IPv6 地址都被允许。 IPv4 地址 `192.168.10.128` 同样时允许的，因为这条规则在约束规则之前适用。
- `networks`: 网络别名数组， 可以用在 `ranges` 中。数组格式：
  - key - 别名
  - value - 一组字符串。每个子字符串可以是一个范围，IP地址或另一个别名。子字符串也可以 设置一个表示非的字符 `!` (和 `negation` 不相关)。

  下列别名为默认定义：

```
— *: any
— any: 0.0.0.0/0, ::/0
— private: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, fd00::/8
— multicast: 224.0.0.0/4, ff00::/8
— linklocal: 169.254.0.0/16, fe80::/10
— localhost: 127.0.0.0/8', ::1
— documentation: 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, 2001:db8::/32
— system: multicast, linklocal, localhost, documentation
```

信息： 此验证器自版本 2.0.7 后可用。

14.3.14 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。

14.3.15 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。

14.3.16 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。

14.3.17 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。

14.3.18 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。

  注意： 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。

14.3.19 safe（安全）

```
[
    // 标记 "description" 为安全属性
    ['description', 'safe'],
]
```

该验证器并不进行数据验证。而是把一个属性标记为 安全属性。

14.3.20 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。并检查属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 `charset` 属性值， 该值默认为 `UTF-8`。

14.3.21 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。

14.3.22 unique（唯一性）

```
[
    // a1 需要在 "a1" 属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属性起作用， 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` 的模型属性。若不设置，它会直接使用待测属性名（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属性名， 值表示 `targetClass` 的待测数据源字段，而键表示当前模型的待测属性名。 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 `Query` 对象。

14.3.23 `url`（网址）

```
[
    // 检查 "website" 是否为有效的 URL。若没有 URI 方案，
    // 则给 "website" 属性加 "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。

> **注意**: 验证器检查 URL 结构和主机部分是否正确。 它不会检查URL的其余部分，也不是为防止XSS或任何其他攻击而设计的。请参阅 安全最佳实践 有关开发应用程序时威胁防范的更多信息的文章。

14.4 Yii 和 Docker

对于开发和部署 Yii 应用程序可以使用 Docker 容器运行。容器就像一个轻量级的独立虚拟机，它将其服务映射到主机的端口，即在端口 80 上的容器中的Web服务器在您的（本地）主机上的端口 8888 上可用。

容器可以解决许多问题，例如在开发人员计算机和服务器上具有相同的软件版本，在开发时快速部署或模拟多服务器体系结构。

您可以在 docker.com[12] 上阅读有关Docker容器的更多信息。

[12] https://www.docker.com/why-docker

14.4.1 要求

- `docker`
- `docker-compose`

访问下载页面[13]获取 Docker 工具。

14.4.2 安装

安装后，你应该可以运行 `docker ps` 并看到类似的输出：

```
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS
```

这意味着您的Docker守护进程已启动并正在运行。

另外运行 `docker-compose version`，你的输出应该是这样的

```
docker-compose version 1.20.0, build unknown
docker-py version: 3.1.3
CPython version: 3.6.4
OpenSSL version: OpenSSL 1.1.0g  2 Nov 2017
```

使用 Compose，您可以配置管理您的应用程序所需的所有服务，例如数据库和缓存。

14.4.3 资源

- 基于 PHP 的 Yii 镜像可以在这里找到 yii2-docker[14]
- Docker 支持的 yii2-app-basic[15]
- Docker 支持的 yii2-app-advanced[16] 正在开发中

14.4.4 用法

Docker的基本命令是：

```
docker-compose up -d
```

在后台启动堆栈中的所有服务

```
docker-compose ps
```

列出正在运行的服务

```
docker-compose logs -f
```

持续查看所有服务的日志

[13] https://www.docker.com/products/container-runtime
[14] https://github.com/yiisoft/yii2-docker
[15] https://github.com/yiisoft/yii2-app-basic#install-with-docker
[16] https://github.com/yiisoft/yii2-app-advanced/pull/347

```
docker-compose stop
```

优雅地停止堆栈中的所有服务

```
docker-compose kill
```

立即停止堆栈中的所有服务

```
docker-compose down -v
```

停止并删除所有服务，**在不使用 host-volumes 时注意数据丢失**
在容器中运行命令

```
docker-compose run --rm php composer install
```

在新的容器中运行 composer 安装

```
docker-compose exec php bash
```

在 运行中的 `php` 服务中执行 bash

14.4.5 高级主题

Yii 框架测试

你可以按照这里[17]描述的方式为 Yii 本身运行 dockerized 框架测试。

数据库管理工具

以 MySQL（`mysql`）的形式运行MySQL时，可以将 phpMyAdmin 容器添加到您的堆栈中，如下所示：

```
phpmyadmin:
    image: phpmyadmin/phpmyadmin
    ports:
        - '8888:80'
    environment:
        - PMA_ARBITRARY=1
        - PMA_HOST=mysql
    depends_on:
        - mysql
```

14.5 国际化（Internationalization）

国际化（I18N）是指在设计软件时，使它可以无需做大的改变就能够适应不同的语言和地区的需要。对于 Web 应用程序， 这有着特别重要的意义，因为潜在的用户可能会在全球范围内。 Yii 提供的国际化功能支持全方位信息翻译， 视图翻译，日期和数字格式化。

[17] https://github.com/yiisoft/yii2/blob/master/tests/README.md#dockerized-testing

14.5.1 区域和语言（Locale and Language）

区域（Locale）

区域设置是一组参数以定义用户希望能在他们的用户界面所看到用户的语言， 国家和任何特殊的偏好。 它通常是由语言 ID 和区域 ID 组成。

例如，ID “en-US” 代表英语和美国的语言环境。为了保持一致性，

在 Yii 应用程序中使用的所有区域 ID 应该规范化为 `ll-cc`， 其中 `ll` 是根据两个或三个字母的小写字母语言代码 ISO-639[18] 和 `cc` 是两个字母的国别代码 ISO-3166[19]。 有关区域设置的更多细节可以看 ICU 项目文档[20]。

语言（Language）

在 Yii中，我们经常用 “language” 来代表一个区域。

Yii 应用程序使用两种语言：

- 源语言：前者指的是写在代码中的语言，后者是向最终用户显示内容的语言。
- 目标语言：而信息翻译服务主要是将文本消息从原语言翻译到目标语言。

所谓的消息翻译服务主要将文本消息从源语言转换为目标语言。

配置（Configuration）

可以用类似下面的应用程序配置来配置应用程序语言：

```
return [
    // 设置目标语言为俄语
    'language' => 'ru-RU',

    // 设置源语言为英语
    'sourceLanguage' => 'en-US',

    ......
];
```

默认的 源语言 值是 `en-US`，即美国英语。 建议你保留此默认值不变， 因为通常让人将英语翻译成其它语言要比将其它语言翻译成其它语言容易得多。

你经常需要根据不同的因素来动态地设置 目标语言 ， 如最终用户的语言首选项。要在应用程序配置中配置它， 你可以使用下面的语句来更改目标语言：

```
// 改变目标语言为中文
\Yii::$app->language = 'zh-CN';
```

> **提示：** 如果您的源语言在代码的不同部分中有所不同，那么您可以覆盖不同消息源的源语言， 这将在下一节中介绍。

[18] https://www.loc.gov/standards/iso639-2/
[19] https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes
[20] https://unicode-org.github.io/icu/userguide/locale/#the-locale-concept

14.5.2 消息翻译（Message Translation）

从源语言到目标语言（From source language to target language）

消息翻译服务用于将一条文本信息从一种语言（通常是 源语言 ） 翻译成另一种语言（通常是 目标语言）。

它的翻译原理是通过在语言文件中查找要翻译的信息以及翻译的结果。如果要翻译的信息可以在语言文件中找到，会返回相应的翻译结果； 否则会返回原始未翻译的信息。

如何实现（How to implement）

为了使用消息翻译服务，需要做如下工作：

1. 调用 `Yii::t()` 方法且在其中包含每一条要翻译的消息；
2. 配置一个或多个消息来源，能在其中找得到要翻译的消息和翻译结果；
3. 让译者翻译信息并将它们存储在消息来源。

1. 包裹一条消息（Wrap a text message） 这个 `Yii::t()` 方法的用法如下，

```
echo \Yii::t('app', 'This is a string to translate!');
```

第一个参数指储存消息来源的类别名称， 第二个参数指需要被翻译的消息。

2. 配置一个或多个消息源（Configure one or multiple message sources） 这个 `Yii::t()` 方法会调用 `i18n` 应用组件 来实现翻译工作。这个组件可以在应用程序中按下面的代码来配置，

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                //'sourceLanguage' => 'en-US',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

在上面的代码中，正在配置 `yii\i18n\PhpMessageSource` 支持的消息源。

带 符号的类别通配符（Category wildcards with symbol） 在上面的代码中，配置了由 `yii\i18n\PhpMessageSource` 所支持的消息来源。模式 `app*` 表示所有以 `app` 开头的消息类别名称都使用这个翻译的消息来源。

3. 让译员翻译消息并将它们存储在消息源中（Let the translators translate messages and store them in the message source(s)） `yii\i18n\PhpMessageSource` 类使用PHP文件和一个简单的 PHP 数组来存储消息转换。 这些文件包含 源语言 中的消息到 目标语言 中的翻译的映射。

> **信息：** 您可以使用 `message` 命令 自动生成这些 PHP 文件， 这将在本章后面介绍。

每个PHP文件对应于单个类别的消息。 默认情况下，文件名应与类别名称相同。 `app/messages/nl-NL/main.php` 的例子：

```
<?php

/**
* Translation map for nl-NL
*/
return [
    'welcome' => 'welkom'
];
```

文件映射（File mapping） `fileMap` 来映射一个类别到不同名称的 PHP 文件。

在上面的例子中，类别 `app/error` 被映射到PHP文件 `@app/messages/ru-RU/error.php`（假设 `ru-RU` 为目标语言）。 如果没有此配置， 该类别将被映射到 `@app/messages/ru-RU/app/error.php`。

其他存储类型（Other storage types） 除了在PHP文件中存储消息来源， 也可以使用下面的消息来源在不同的存储来存储翻译的消息：

- `yii\i18n\GettextMessageSource` 使用 GNU Gettext 的 MO 或 PO 文件保存翻译的消息。
- `yii\i18n\DbMessageSource` 使用一个数据库表来存储翻译的消息。

14.5.3 消息格式化（Message Formatting）

在要翻译的消息里，你可以嵌入一些占位符，并让它们通过动态的参数值来代替。 你甚至可以根据目标语言格式的参数值来使用特殊的占位符。 在本节中，我们将介绍如何用不同的方式来格式化消息。

消息参数（Message Parameters）

在待翻译的消息，可以嵌入一个或多个占位符，以便它们可以由给定的参数值取代。 通过给不同的参数值，可以动态地改变翻译内容的消息。

在下面的例子中， 占位符 `{username}` 在 "`Hello, {username}!`" 中将分别被 `'Alexander'`和`'Qiang'` 所替换。

```
$username = 'Alexander';
// 输出: "Hello, Alexander"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);

$username = 'Qiang';
// 输出: "Hello, Qiang"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);
```

当翻译的消息包含占位符时，应该让占位符保留原样。 这是因为调用 `Yii::t()` 时，占位符将被实际参数值代替。

你可以使用 名称占位符 或者 位置占位符，但不能两者都用在同一个消息里。

前面的例子说明了如何使用名称占位符。即每个占位符的格式为 `{参数名称}`，你所提供的参数作为关联数组， 其中数组的键是参数名称（没有大括号）， 数组的值是对应的参数值。

位置占位符是使用基于零的整数序列，在调用 `Yii::t()` 时会参数值根据它们出现位置的顺序分别进行替换。 在下面的例子中，位置占位符 `{0}`，`{1}` 和 `{2}` 将分别被 `$price`，`$count` 和 `$subtotal` 所替换。

```
$price = 100;
$count = 2;
$subtotal = 200;
echo \Yii::t('app', 'Price: {0}, Count: {1}, Subtotal: {2}', $price, $count,
$subtotal);
```

在单个位置参数的情况下，它的值可以被指定而不包含在数组中：

```
echo \Yii::t('app', 'Price: {0}', $price);
```

> **提示：** 大多数情况下你应该使用名称占位符。 这是因为参数名称可以让翻译者更好的理解要被翻译的消息。

格式化参数

你可以在消息的占位符指定附加格式的规则， 这样的参数值可在替换占位符之前格式化它们。在下面的例子中， 价格参数值将视为一个数并格式化为货币值：

```
$price = 100;
echo \Yii::t('app', 'Price: {0, number, currency}', $price);
```

> **注意：** 参数的格式化需要安装 intl PHP 扩展[21]。

[21] https://www.php.net/manual/zh/intro.intl.php

可以使用缩写的形式或完整的形式来格式化占位符：

```
short form: {PlaceholderName, ParameterType}
full form: {PlaceholderName, ParameterType, ParameterStyle}
```

> **注意：** 如果您需要使用特殊字符（如 `{}`，`\mintinline{text}{}`，`'`，`#`，请使用 `'`：
>
> ```
> echo Yii::t('app', "Example of string with ''-escaped characters'':
> '{' '}' '{test}' {count,plural,other{''count'' value is # '#{}'}}",
> ['count' => 3]);
> ```

请参阅 ICU 文档[22] 关于如何指定这样的占位符的说明。接下来我们会展示一些常用的使用方法。

数字（Number） 参数值应该被格式化为一个数。例如，

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number}', $sum);
```

你可以指定参数的格式为 `integer`（整型），`currency`（货币），或者 `percent`（百分数）：

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, currency}', $sum);
```

你也可以指定一个自定义模式来格式化数字。 例如，

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, ,000,000000}', $sum);
```

自定义格式中使用的字符可以在“特殊模式字符”一节的 ICU API 参考[23] 中找到。

该值始终根据您翻译的区域设置进行格式设置，即无需更改翻译区域设置即可更改小数或千位分隔符，货币符号等。 如果你需要定制这些，你可以使用 `yii\i18n\Formatter::asDecimal()` 和 `yii\i18n\Formatter::asCurrency()`。

日期（Date） 该参数值应该被格式化为一个日期。 例如，

```
echo \Yii::t('app', 'Today is {0, date}', time());
```

你可以指定一个可选的参数格式 `short`，`medium`，`long`，或 `full`：

[22] https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classMessageFormat.html

[23] https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classDecimalFormat.html

```
echo \Yii::t('app', 'Today is {0, date, short}', time());
```

你还可以指定一个自定义模式来格式化日期:

```
echo \Yii::t('app', 'Today is {0, date, yyyy-MM-dd}', time());
```

格式化参考[24]。

时间（Time） 参数值应该被格式化为一个时间。 例如,

```
echo \Yii::t('app', 'It is {0, time}', time());
```

你可以指定一个可选的参数格式 `short` , `medium` , `long` , 或 `full` :

```
echo \Yii::t('app', 'It is {0, time, short}', time());
```

你还可以指定一个自定义模式来格式化时间:

```
echo \Yii::t('app', 'It is {0, date, HH:mm}', time());
```

格式化参考[25]。

拼写（Spellout） 参数值为一个数并被格式化为它的字母拼写形式。 例如,

```
// 输出: "42 is spelled as forty-two"
echo \Yii::t('app', '{n,number} is spelled as {n, spellout}', ['n' => 42]);
```

默认情况下，该数字拼写为基数。 它可以改变:

```
// may produce "I am forty-seventh agent"
echo \Yii::t('app', 'I am {n,spellout,%spellout-ordinal} agent', ['n' =>
47]);
```

请注意，在 `spellout`, 之后和 `%` 之前不应该有空格。

要获取可用于您正在使用的语言环境的选项列表，请查看 https://intl.rmcreative.ru/[26] 上的“编号模式，拼写”。

[24] https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1SimpleDateFormat.html#details

[25] https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1SimpleDateFormat.html#details

[26] https://intl.rmcreative.ru/

序数词（Ordinal） 参数值为一个数并被格式化为一个序数词。 例如，

```
// 输出: "You are the 42nd visitor here!"
echo \Yii::t('app', 'You are the {n, ordinal} visitor here!', ['n' => 42]);
```

Ordinal 支持更多格式化西班牙语等语言的方式:

```
// may produce 471ª
echo \Yii::t('app', '{n,ordinal,%digits-ordinal-feminine}', ['n' => 471]);
```

请注意，在 `ordinal`，之后和 `%` 之前不应该有空格。

要获取可用于您正在使用的语言环境的选项列表，请查看 https://intl.rmcreative.ru/[27] 上的“编号模式，序号”。

持续时间（Duration） 参数值为秒数并被格式化为持续的时间段。 例如，

```
// 输出: "You are here for 47 sec. already!"
echo \Yii::t('app', 'You are here for {n, duration} already!', ['n' => 47]);
```

持续时间支持更多格式化方法:

```
// may produce 130:53:47
echo \Yii::t('app', '{n,duration,%in-numerals}', ['n' => 471227]);
```

请注意，在 `duration`，之后和 `%` 之前不应该有空格。

要获取您正在使用的区域设置的可用选项列表，请查看 https://intl.rmcreative.ru/[28] 上的“编号模式，持续时间”。

复数（Plural） 不同的语言有不同的方式来表示复数。 Yii 提供一个便捷的途径， 即使是非常复杂的规则也使翻译消息时不同的复数形式行之有效。 取之以直接处理词形变化规则，它是足以面对某些词形变化语言的翻译。 例如，

```
// 当 $n = 0 时，输出: "There are no cats!"
// 当 $n = 1 时，输出: "There is one cat!"
// 当 $n = 42 时，输出: "There are 42 cats!"
echo \Yii::t('app', 'There {n, plural, =0{are no cats} =1{is one cat}
other{are # cats}}!', ['n' => $n]);
```

在上面的多个规则的参数中， `=0` 意味着 `n` 的值是 0 ， `=1` 意味着 `n` 的值是 1 ， 而 `other` 则是对于其它值， `#` 会被 `n` 中的值给替代。

复数形式可以是某些非常复杂的语言。下面以俄罗斯为例，`=1` 完全匹配 `n = 1`， 而 `one` 匹配 `21` 或 `101`:

[27] https://intl.rmcreative.ru/
[28] https://intl.rmcreative.ru/

```
Здесь {n, plural, =0{котов нет} =1{есть один кот} one{# кот} few{# кота}
many{# котов} other{# кота}}!
```

注意，上述信息主要是作为一个翻译的信息， 而不是一个原始消息，除非设置应用程序的 源语言 为 `ru-RU`。

> **注意：** 除非您将应用程序的 源语言 设置为“RU-RU”，并且从以下语言转换而来，上面的示例俄语消息主要用作翻译的消息，而不是原始消息俄语。
>
> 当在 `Yii::t()` 调用中指定的原始消息未找到翻译时， 源语言 复数规则将应用于原始消息。

对于字符串如下所示的情况，有一个 `offset` 参数：

```
$likeCount = 2;
echo Yii::t('app', 'You {likeCount,plural,
    offset: 1
    =0{did not like this}
    =1{liked this}
    one{and one other person liked this}
    other{and # others liked this}
}', [
    'likeCount' => $likeCount
]);

// You and one other person liked this
```

序数选择（Ordinal selection） `selectordinal` 的参数类型旨在为您所翻译的语言环境选择一个基于语序规则的字符串： 一个基于语序规则的字符串：

```
$n = 3;
echo Yii::t('app', 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd}
other{#th}} visitor', ['n' => $n]);
// For English it outputs:
// You are the 3rd visitor

// Translation
'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}}
visitor' => 'Вы {n,selectordinal,other{#-й}} посетитель',

// For Russian translation it outputs:
// Вы 3-й посетитель
```

格式与复数使用的格式非常接近。 要了解您应为特定语言环境指定哪些参数，请参阅 https://intl.rmcreative.ru/[29] 上的“复数规则，序数”。 或者，您可以参考 unicode.org上的规则参考[30]。

[29] https://intl.rmcreative.ru/

[30] https://unicode-org.github.io/cldr-staging/charts/37/supplemental/language_plural_rules.html

选择（Selection） 可以使用 `select` 参数类型来选择基于参数值的短语。例如，

```
// 输出: "Snoopy is a dog and it loves Yii!"
echo \Yii::t('app', '{name} is a {gender} and {gender, select, female{she}
male{he} other{it}} loves Yii!', [
    'name' => 'Snoopy',
    'gender' => 'dog',
]);
```

在上面的表达中， `female` 和 `male` 是可能的参数值， 而 `other` 用于处理不与它们中任何一个相匹配的值。对于每一个可能的参数值， 应指定一个短语并把它放在在一对大括号中。

指定默认翻译（Specifying default message source）

你可以指定使用默认的翻译，该翻译将作为一个类别，用于不匹配任何其他翻译的后备。 这种翻译应标有 `*` 。 为了做到这一点以下内容需要添加到应用程序的配置:

```
//配置 i18n 组件

'i18n' => [
    'translations' => [
        '*' => [
            'class' => 'yii\i18n\PhpMessageSource'
        ],
    ],
],
```

现在，你可以使用每一个还没有配置的类别，这跟 Yii 1.1 的行为有点类似。该类别的消息将来自在默认翻译 `basePath` 中的一个文件， 该文件在 `@app/messages` :

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

该消息将来自 `@app/messages/<LanguageCode>/not_specified_category.php` 。

翻译模块消息（Translating module messages）

如果你想翻译一个模块的消息，并避免使用单一翻译文件的所有信息，你可以按照下面的方式来翻译:

```
<?php

namespace app\modules\users;

use Yii;

class Module extends \yii\base\Module
```

```
{
    public $controllerNamespace = 'app\modules\users\controllers';

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        Yii::$app->i18n->translations['modules/users/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/modules/users/messages',
            'fileMap' => [
                'modules/users/validation' => 'validation.php',
                'modules/users/form' => 'form.php',
                ...
            ],
        ];
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('modules/users/' . $category, $message, $params,
        $language);
    }

}
```

在上面的例子中，我们使用通配符匹配，然后过滤了所需的文件中的每个类别。取之使用 `fileMap` ，你可以简单地使用类映射的同名文件。 现在你可以直接使用 `Module::t('validation', 'your custom validation message')` 或 `Module::t('form', 'some form label')`。

翻译小部件消息（Translating widgets messages）

上述模块的翻译规则也同样适用于小部件的翻译规则，例如：

```
<?php

namespace app\widgets\menu;

use yii\base\Widget;
use Yii;

class Menu extends Widget
{

    public function init()
```

```
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        $i18n = Yii::$app->i18n;
        $i18n->translations['widgets/menu/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/widgets/menu/messages',
            'fileMap' => [
                'widgets/menu/messages' => 'messages.php',
            ],
        ];
    }

    public function run()
    {
        echo $this->render('index');
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('widgets/menu/' . $category, $message, $params,
        $language);
    }

}
```

你可以简单地使用类映射的同名文件而不是使用 `fileMap` 。 现在你直接可以使用 `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` 。

> **提示：** 对于小部件也可以使用 i18n 视图，并一样以控制器的规则来应用它们。

翻译框架信息（Translating framework messages）

Yii 自带了一些默认的信息验证错误和其他一些字符串的翻译。 这些信息都是在 `yii` 类别中。有时候你想纠正应用程序的默认信息翻译。 为了做到这一点，需配置 `i18n` 应用组件 如下：

```
'i18n' => [
    'translations' => [
        'yii' => [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/messages'
        ],
    ],
],
```

现在可以把你修改过的翻译放在 `@app/messages/<language>/yii.php`。

处理缺少的翻译（Handling missing translations）

如果翻译的消息在消息源文件里找不到，Yii 将直接显示该消息内容。这样一来当你的原始消息是一个有效的冗长的文字时会很方便。 然而，有时它是不能实现我们的需求。你可能需要执行一些自定义处理的情况， 这时请求的翻译可能在消息翻译源文件找不到。 这可通过使用 `missingTranslation - yii\i18n\MessageSource` 的事件来完成。

例如，你可能想要将所有缺失的翻译做一个明显的标记，这样它们就可以很容易地在页面中找到。 为此，你需要先设置一个事件处理程序。这可以在应用程序配置中进行：

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
                'on missingTranslation' =>
                ['app\components\TranslationEventHandler',
                'handleMissingTranslation']
            ],
        ],
    ],
],
```

现在，你需要实现自己的事件处理程序：

```
<?php

namespace app\components;

use yii\i18n\MissingTranslationEvent;

class TranslationEventHandler
{
    public static function handleMissingTranslation(MissingTranslationEvent
    $event)
    {
        $event->translatedMessage = "@MISSING:
        {$event->category}.{$event->message} FOR LANGUAGE {$event->language}
        @";
    }
}
```

如果 `yii\i18n\MissingTranslationEvent::$translatedMessage` 是由事件处理程序设置，它将显示翻译结果。

> **注意：** 每个消息源会单独处理它缺少的翻译。如果是使用多个消息源，并希望他们把缺少的翻译以同样的方式来处理， 你应该给它们每一个消息源指定相应的事件处理程序。

使用 命令（Using the command）

翻译储存在 `php` 文件，[[yii\i18n\GettextMessageSource|.po 文件] 或者 数据库。具体见类的附加选项。

首先，你需要创建一个配置文件。确定应该保存在哪里，然后执行命令

```
./yii message/config path/to/config.php
```

打开创建的文件，并按照需求来调整参数。特别注意：

- `languages`: 代表你的应用程序应该被翻译成什么语言的一个数组;
- `messagePath`: 存储消息文件的路径，这应与配置中 `i18n` 的 `basePath` 参数一致。

您也可以使用 './yii message/config' 命令通过 cli 动态生成带有指定选项的配置文件。 例如，你可以像下面这样设置 `languages` 和 `messagePath` 参数：

```
./yii message/config --languages=de,ja --messagePath=messages
path/to/config.php
```

要获取可用选项列表，请执行下一个命令：

```
./yii help message/config
```

一旦你完成了配置文件，你最终可以用命令提取你的消息：

```
./yii message path/to/config.php
```

另外，您可以使用选项来动态更改提取参数。

然后你会发现你的文件（如果你已经选择基于文件的翻译）在 `messagePath` 目录。

14.5.4 视图的翻译（View Translation）

有时你可能想要翻译一个完整的视图文件，而不是翻译单条文本消息。为了达到这一目的， 只需简单的翻译视图并在它子目录下保存一个名称一样的目标语言文件。 例如，如果你想要翻译的视图文件为 `views/site/index.php` 且目标语言是 `ru-RU`， 你可以将视图翻译并保存为 `views/site/ru-RU/index.php`。现在 每当你调用 `yii\base\View::renderFile()` 或任何其它方法 (如 `yii\base\Controller::render()`) 来渲染 `views/site/index.php` 视图， 它最终会使用所翻译的 `views/site/ru-RU/index.php`。

> **注意：** 如果 目标语言 跟 源语言 相同， 在翻译视图的存在下，将呈现原始视图。

14.5.5 格式化日期和数字值（Formatting Date and Number Values）

在 格式化输出数据 一节可获取详细信息。

14.5.6 设置 PHP 环境（Setting Up PHP Environment）

Yii 使用 PHP intl 扩展[31] 来提供大多数 I18N 的功能， 如日期和数字格式的 `yii\i18n\Formatter` 类和消息格式的 `yii\i18n\MessageFormatter` 类。 当 `intl` 扩展没有安装时，两者会提供一个回调机制。然而，该回调机制只适用于目标语言是英语的情况下。 因此，当 I18N 对你来说必不可少时，强烈建议你安装 `intl`。

PHP intl 扩展[32] 是基于对于所有不同的语言环境 提供格式化规则的 ICU库[33]。 不同版本的 ICU 中可能会产生不同日期和数值格式的结果。 为了确保你的网站在所有环境产生相同的结果， 建议你安装与 `intl` 扩展相同的版本（和 ICU 同一版本）。

要找出所使用的 PHP 是哪个版本的 ICU ，你可以运行下面的脚本，它会给出你所使用的 PHP 和 ICU 的版本。

```
<?php
echo "PHP: " . PHP_VERSION . "\n";
echo "ICU: " . INTL_ICU_VERSION . "\n";
echo "ICU Data: " . INTL_ICU_DATA_VERSION . "\n";
```

此外，还建议你所使用的 ICU 版本应等于或大于 49 的版本。这确保了可以使用本文档描述的所有功能。例如， 低于 49 版本的 ICU 不支持使用 `#` 占位符来实现复数规则。 请参阅 `https://icu.unicode.org/download` 获取可用 ICU 版本的完整列表。 注意，版本编号在 4.8 之后发生了变化（如 ICU4.8，ICU49，50 ICU 等）。

另外，ICU 库中时区数据库的信息可能过时。要更新时区数据库时详情请参阅 ICU 手册[34] 。 而对于 ICU 输出格式使用的时区数据库，PHP 用的时区数据库可能跟它有关。 你可以通过安装 pecl package `timezonedb`[35] 的最新版本来更新它。

14.6 收发邮件

> **注意：** 本节正在开发中。

Yii 支持组成和发送电子邮件。然而，该框架提供的只有内容组成功能和基本接口。 实际的邮件发送机制可以通过扩展提供， 因为不同的项目可能需要不同的实现方式， 它通常取决于外部服务和库。

[31] https://www.php.net/manual/zh/book.intl.php
[32] https://www.php.net/manual/zh/book.intl.php
[33] https://icu.unicode.org/
[34] https://unicode-org.github.io/icu/userguide/datetime/timezone/#updating-the-time-zone-data
[35] https://pecl.php.net/package/timezonedb

大多数情况下你可以使用 yii2-swiftmailer[36] 官方扩展。

14.6.1 配置

邮件组件配置取决于你所使用的扩展。 一般来说你的应用程序配置应如下：

```
return [
    //....
    'components' => [
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
    ],
];
```

14.6.2 基本用法

一旦 “mailer” 组件被配置，可以使用下面的代码来发送邮件：

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->setTextBody('Plain text content')
    ->setHtmlBody('<b>HTML content</b>')
    ->send();
```

在上面的例子中所述的 `compose()` 方法创建了电子邮件消息，这是填充和发送的一个实例。 如果需要的话在这个过程中你可以用上更复杂的逻辑：

```
$message = Yii::$app->mailer->compose();
if (Yii::$app->user->isGuest) {
    $message->setFrom('from@domain.com')
} else {
    $message->setFrom(Yii::$app->user->identity->email)
}
$message->setTo(Yii::$app->params['adminEmail'])
    ->setSubject('Message subject')
    ->setTextBody('Plain text content')
    ->send();
```

> **注意：** 每个 “mailer” 的扩展也有两个主要类别：“Mailer” 和 “Message”。 “Mailer” 总是知道类名和具体的 “Message”。 不要试图直接实例 “Message” 对象 - 而是始终使用 `compose()` 方法。

你也可以一次发送几封邮件：

[36] https://github.com/yiisoft/yii2-swiftmailer

```
$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);
```

一些特定的扩展可能会受益于这种方法，使用单一的网络消息等。

14.6.3 撰写邮件内容

Yii 允许通过特殊的视图文件来撰写实际的邮件内容。默认情况下， 这些文件应该位于 “@app/mail” 路径。

一个邮件视图内容的例子：

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

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\BaseMessage instance of newly created mail
message */

?>
<h2>This message allows you to visit our site home page by one click</h2>
<?= Html::a('Go to home page', Url::home('http')) ?>
```

为了通过视图文件撰写正文可传递视图名称到 `compose()` 方法中：

```
Yii::$app->mailer->compose('home-link') // 渲染一个视图作为邮件内容
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->send();
```

你也可以在 `compose()` 方法中传递一些视图所需参数，这些参数可以在视图文件中使用：

```
Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

你可以指定不同的视图文件的 HTML 和纯文本邮件内容：

```
Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);
```

如果指定视图名称为纯字符串，它的渲染结果将被用来作为 HTML Body，同时纯文本正文将被删除所有 HTML 实体。

视图渲染结果可以被包裹进布局，可使用 `yii\mail\BaseMailer::$htmlLayout` 和 `yii\mail\BaseMailer::$textLayout` 来设置。它的运行方式跟常规应用程序的布局是一样的。布局可用于设置邮件 CSS 样式或其他共享内容：

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

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\MessageInterface the message being composed */
/* @var $content string main view render result */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?=
Yii::$app->charset ?>" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">With kind regards, <?= Yii::$app->name ?>
team</div>
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

14.6.4 文件附件

你可以使用 `attach()` 和 `attachContent()` 方法来添加附件的信息：

```
$message = Yii::$app->mailer->compose();

// 附件来自本地文件
$message->attach('/path/to/source/file.pdf');

// 动态创建一个文件附件
$message->attachContent('Attachment content', ['fileName' => 'attach.txt',
'contentType' => 'text/plain']);
```

14.6.5 嵌入图片

你可以使用 `embed()` 方法将图片插入到邮件内容。 此方法返回会图片 ID ，这将用在“img”标签中。 当通过视图文件来写信时，这种方法易于使用：

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' =>
'/path/to/image.jpg'])
    // ...
    ->send();
```

然后在该视图文件中，你可以使用下面的代码：

```
<img src="<?= $message->embed($imageFileName); ?>">
```

14.6.6 测试和调试

开发人员常常要检查一下，有什么电子邮件是由应用程序发送的，他们的内容是什么等。 这可通过 `yii\mail\BaseMailer::useFileTransport` 来检查。 如果开启这个选项，会把邮件信息保存在本地文件而不是发送它们。 这些文件保存在 `yii\mail\BaseMailer::fileTransportPath` 中，默认在 ‘@runtime/mail’ 。

> **提示：** 你可以保存这些信息到本地文件或者把它们发送出去，但不能同时两者都做。

邮件信息文件可以在一个普通的文本编辑器中打开，这样你就可以浏览实际的邮件标题，内容等。 这种机制可以用来调试应用程序或运行单元测试。

> **提示：** 该邮件信息文件是会被 `\yii\mail\MessageInterface::toString()` 转成字符串保存的， 它依赖于实际在应用程序中使用的邮件扩展。

14.6.7 创建自己的邮件解决方案

为了创建你自己的邮件解决方案，你需要创建两个类，一个用于 “Mailer”，另一个用于 “Message”。 你可以使用 `yii\mail\BaseMailer` 和 `yii\mail\BaseMessage` 作为基类。 这些类已经实现了基本的逻辑，这在本指南中有介绍。 然而，它们的使用不是必须的， 它实现了 `yii\mail\MailerInterface` 和 `yii\mail\MessageInterface` 接口。 然后，你需要实现所有 abstract 方法来构建解决方案。

14.7 性能优化

有许多因素影响你的 Web 应用程序的性能。有些是环境， 有些是你的代码，而其他一些与 Yii 本身有关。 在本节中，我们将列举这些因素并解释如何通过调整这些因素来提高应用程序的性能。

14.7.1 优化你的 PHP 环境

一个好的 PHP 环境是非常重要的。为了得到最大的性能，

- 使用最新稳定版本的 PHP。 PHP 的主要版本可能带来显著的性能提升。
- 启用字节码缓存 Opcache[37]（PHP 5.5或更高版本） 或 APC[38] （PHP 5.4或更早版本）。字节码缓存省去了每次解析和加载 PHP 脚本所带来的开销。
- Tune `realpath()` cache[39].

14.7.2 禁用调试模式

对于运行在生产环境中的应用程序，你应该禁用调试模式。 Yii 中使用名为 `YII_DEBUG` 的常量来定义调试模式是否应被激活。 若启用了调试模式，Yii 将需要额外的时间来产生和记录调试信息。

你可以将下面的代码行放在 入口脚本 的开头来禁用调试模式：

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

> **提示：** `YII_DEBUG` 的默认值是 false 。所以如果你确信你不在你应用程序代码中别的地方更改其默认值， 你可以简单地删除上述行来禁用调试模式。

14.7.3 使用缓存技术

你可以使用各种缓存技术来提高应用程序的性能。例如，如果你的应用程序允许用户以 Markdown 格式输入文字， 你可以考虑缓存解析后的 Markdown 内容，避免每个请求都重复解析相同的 Markdown 文本。 请参阅 缓存 一节， 了解 Yii 提供的缓存支持。

14.7.4 开启 Schema 缓存

Schema 缓存是一个特殊的缓存功能， 每当你使用活动记录时应该要开启这个缓存功能。如你所知， 活动记录能智能检测数据库对象的集合（例如列名、列类型、约束）而不需要手动地描述它们。 活动记录是通过执行额外的 SQL 查询来获得该信息。 通过启用 Schema 缓存，检索到的数据库对象的集合将被保存在缓存中并在将来的请求中重用。

要开启 Schema 缓存，需要配置一个 `cache` 应用组件来储存 Schema 信息， 并在 配置 中设置 `yii\db\Connection::$enableSchemaCache` 为 `true` :

```
return [
    // ...
```

[37] https://www.php.net/manual/zh/book.opcache.php
[38] https://www.php.net/manual/zh/book.apcu.php
[39] https://github.com/samdark/realpath_cache_tuner

```
    'components' => [
        // ...
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase',
            'username' => 'root',
            'password' => '',
            'enableSchemaCache' => true,

            // Duration of schema cache.
            'schemaCacheDuration' => 3600,

            // Name of the cache component used to store schema information
            'schemaCache' => 'cache',
        ],
    ],
];
```

14.7.5 合并和压缩资源文件

一个复杂的网页往往包括许多 CSS 和 JavaScript 资源文件。 为减少 HTTP 请求的数量和这些资源总下载的大小，应考虑将它们合并成一个单一的文件并压缩。 这可大大提高页面加载时间，且减少了服务器负载。 想了解更多细节，请参阅前端资源部分。

14.7.6 优化会话存储

默认会话数据被存储在文件中。 这是好的对处于发展项目或小型项目。 但是，当涉及要处理大量并发请求时， 最好使用其他的会话存储方式，比如数据库。 Yii 支持各种会话存储。 你可以通过在配置中配置 `session` 组件来使用这些存储， 如下代码：

```
return [
    // ...
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',

            // Set the following if you want to use DB component other than
            // default 'db'.
            // 'db' => 'mydb',

            // To override default session table, set the following
            // 'sessionTable' => 'my_session',
        ],
    ],
];
```

以上配置是使用数据库来存储会话数据。默认情况下， 它会使用 db 应用组件连接数据库并将会话数据存储在 session 表。 因此，你必须创建如下的 session 表，

```
CREATE TABLE session (
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

你也可以通过使用缓存来存储会话数据 yii\web\CacheSession 。 理论上讲，你可以使用只要支持数据缓存。 但是请注意，某些缓存的存储当达到存储限制会清除缓存数据。出于这个原因，你应主要在不存在存储限制时才使用这些缓存存储。 如果你的服务器支持 Redis[40]，强烈建议你通过使用 yii\redis\Session 来作为会话存储。

如果您的服务器上有 Redis[41]， 强烈建议您使用 yii\redis\Session 作为会话存储。

14.7.7 优化数据库

执行数据库查询并从数据库中取出数据往往是一个 Web 应用程序主要的性能瓶颈。 尽管使用数据缓存技术可以缓解性能下降，但它并不完全解决这个问题。 当数据库包含大量的数据且缓存数据是无效的， 获取最新的数据可能是最耗性能的假如在没有适当地设计数据库和查询条件。

一般来说，提高数据库查询的性能是创建索引。例如，如果你需要找一个用户表的“用户名”， 你应该为“用户名”创建一个索引。 注意，尽管索引可以使选择查询的速度快得多，但它会减慢插入、更新和删除的查询。

对于复杂的数据库查询，建议你创建数据库视图来保存查询分析和准备的时间。

最后，在“SELECT”中使用“LIMIT”查询。 这可以避免从数据库中取出大量数据。

14.7.8 使用普通数组

尽管活动记录对象使用起来非常方便， 但当你需要从数据库中检索大量数据时它的效率不如使用普通的数组。 在这种情况下，你可以考虑在使用活动记录查询数据时调用 asArray()， 使检索到的数据被表示为数组而不是笨重的活动记录对象。例如，

```
class PostController extends Controller
{
    public function actionIndex()
    {
        $posts = Post::find()->limit(100)->asArray()->all();
```

[40] https://redis.io/
[41] https://redis.io/

```
        return $this->render('index', ['posts' => $posts]);
    }
}
```

在上述代码中，$posts 将被表中的行填充形成数组。每一行是一个普通的数组。要访问 第 i 行的 `title` 列，你可以使用表达式 `$post[$i]['title']`。

你也可以使用 DAO 以数组的方式来构建查询和检索数据。

14.7.9 优化 Composer 自动加载

因为 Composer 自动加载用于加载大多数第三方类文件， 应考虑对其进行优化，通过执行以下命令：

```
composer dumpautoload -o
```

另外，您可以考虑使用 authoritative class maps[42] 和 APCu 缓存[43]。 请注意，这两种选择可能适用于您的特定情况，也可能不适合您。

14.7.10 处理离线数据

当一个请求涉及到一些资源密集操作， 你应该想办法在无需用户等待他们完成脱机模式时来执行这些操作。

有两种方法可以离线数据处理：推和拉。

在拉中，只要有请求涉及到一些复杂的操作，你创建一个任务，并将其保存在永久存储，例如数据库。然后， 使用一个单独的进程（如 cron 作业）拉任务，并进行处理。 这种方法很容易实现，但它也有一些缺点。 例如，该任务过程中需要定期地从任务存储拉。如果拉频率太低，这些任务可以延迟处理； 但是如果频率过高，将引起的高开销。

在推中，你可以使用消息队列（如 RabbitMQ ， ActiveMQ ， Amazon SQS 等）来管理任务。 每当一个新的任务放在队列中，它会启动或者通知任务处理过程去触发任务处理。

14.7.11 性能分析

你应该配置你的代码来找出性能缺陷，并相应地采取适当措施。 以下分析工具可能是有用的:

- Yii debug toolbar and debugger[44]
- Blackfire[45]
- XHProf[46]
- XDebug profiler[47]

[42] https://getcomposer.org/doc/articles/autoloader-optimization.md#optimization-level-2-a-authoritative-class-maps
[43] https://getcomposer.org/doc/articles/autoloader-optimization.md#optimization-level-2-b-apcu-cache
[44] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md
[45] https://blackfire.io/
[46] https://www.php.net/manual/zh/book.xhprof.php
[47] https://xdebug.org/docs/profiler

14.7.12 准备扩展应用程序

当没有任何帮助时，您可以尝试使您的应用程序可扩展。Configuring a Yii2 Application for an Autoscaling Stack[48] 中提供了一个很好的介绍。有关进一步阅读，请参阅 Web apps performance and scaling[49].

14.8 共享托管环境

共享的托管环境常常会对目录结构以及配置文件有较多的限制。 然而，在大多数情况下，你仍可以通过少量的修改以在共享托管环境下运行 Yii 2.0。

14.8.1 部署一个基础应用模板

由于共享托管环境往往只有一个 webroot，如果可能，请优先使用基础项目模板（ basic project template)构建你的应用程序。 参考 安装 Yii 章节在本地安装基础项目模板。 当你让应用程序在本地正常运行后， 我们将要做少量的修改以让它可以在共享托管服务器运行。

重命名 webroot

用 FTP 或者其他的工具连接到你的托管服务器，你可能看到类似如下的目录结构：

```
config
logs
www
```

在以上，`www` 是你的 web 服务器的 webroot 目录。不同的托管环境下名称可能各不相同，通常是类似：`www`，`htdocs` 和 `public_html` 之类的名称。

对于我们的基础项目模板而言，其 webroot 名为 `web`。 在你上传你的应用程序到 web 服务器上去之前，将你的本地 webroot 重命名以匹配服务器。 即：从 `web` 改为 `www`，`public_html` 或者其他你的托管环境的 webroot 名称。

FTP 根目录可写

如果你有 FTP 根目录的写权限， 即，有 `config`，`logs` 和 `www` 的根目录，那么，如本地根目录相同的结构上传 `assets`，`commands` 等目录。

[48] https://github.com/samdark/yii2-cookbook/blob/master/book/scaling.md
[49] https://thehighload.com/

增加 web 服务器的额外配置

如果你的 web 服务器是 Apache，你需要增加一个包含如下内容的 `.htaccess` 文件到你的 `web` 目录 (或者 `public_html` 根据实际情况而定，是你的 `index.php` 文件所在的目录)。

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

对于 nginx 而言，你不需要额外的配置文件。

检查环境要求

为了运行 Yii，你的 web 服务器必须匹配它的环境要求。最低的要求必须是 PHP 5.4。 为了检查环境配置，将 `requirements.php` 从你的根目录拷贝到 webroot 目录， 并通过浏览器输入 URL `https://example.com/requirements.php` 运行它。最后，检查结束后别忘了删除这个文件哦！

14.8.2 部署一个高级应用程序模板

将高级应用程序部署到共享主机比基本应用程序有点棘手但可以实现。 请按照高级项目模板文档[50]中的说明进行操作。

14.9 使用模板引擎（Using template engines）

默认情况下，Yii 使用 PHP 作为其默认的模板引擎语言，但是，你可以配置 Yii 以扩展的方式支持其他的渲染引擎， 比如 Twig[51] 或 Smarty[52]等。

组件 `view` 就是用于渲染视图的。 你可以重新配置这个组件的行为以增加一个自定义的模板引擎。

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
```

[50] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/topic-shared-hosting.md
[51] https://twig.symfony.com/
[52] https://www.smarty.net/

```
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // Array of twig options:
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
                ],
                // ...
            ],
        ],
    ],
]
```

在上述的代码中， Smarty 和 Twig 都被配置以让视图文件使用。但是，为了让扩展安装到项目中， 你同样需要修改你的 `composer.json` 文件，如下：

```
"yiisoft/yii2-smarty": "*",
"yiisoft/yii2-twig": "*",
```

上述代码需要增加到 `composer.json` 的 `require` 节中。在做了上述修改，并保存后，你可以运行 `composer update --prefer-dist` 命令来安装扩展。

对于特定模板引擎的使用详细，请参考其文档：

- Twig guide[53]
- Smarty guide[54]

14.10 引入第三方代码（Working with Third-Party Code）

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。

14.10.1 在 Yii 中使用第三方类库（Using Third-Party Libraries in Yii）

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。

[53] https://github.com/yiisoft/yii2-twig/tree/master/docs/guide
[54] https://github.com/yiisoft/yii2-smarty/tree/master/docs/guide

使用 Composer 包（Using Composer Packages）

目前很多第三方的类库都以 Composer[55] 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';
```

使用下载的类库（Using Downloaded Libraries）

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[56] 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:

[55] https://getcomposer.org/
[56] https://www.php-fig.org/psr/psr-4/

- 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';
```

14.10.2 在第三方系统内使用 Yii（Using Yii in Third-Party Systems）

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:

```
composer require yiisoft/yii2
```

如果您只想使用数据库抽象层或 Yii 的其他非静态资源的相关功能， 您应该需要一个特殊的 composer 包来阻止 Bower 和 NPM 包的安装。 有关详细信息，请参见 cebe/assetfree-yii2[57]。

有关 Composer 的更多信息以及安装过程中可能出现的问题的解决方案，另请参见 关于安装 Yii 的部分。

不然的话，你可以下载[58] 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); // 不要在这里调用 run() 方法。
```

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。

[57] https://github.com/cebe/assetfree-yii2
[58] https://www.yiiframework.com/download/

14.10.3 配合使用 Yii 2 和 Yii 1（Using Yii 2 with 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 应用的入口脚步：

```
// 导入下面会详细说明的定制 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
{
    // copy-paste the code from YiiBase (1.x) here
}

Yii::$classMap = include($yii2path . '/classes.php');
// 通过 Yii 1 注册 Yii 2 的类自动加载器
Yii::registerAutoloader(['Yii', 'autoload']);
// create the dependency injection container
Yii::$container = new yii\di\Container;
```

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'
```

14.11 使用 Yii 作为微框架

Yii 可以轻松使用，而不需要基本和高级模板中包含的功能。换句话说，Yii 已经是一个微框架。不需要由模板提供的目录结构与 Yii 一起工作。

当你不需要像 assets 或视图一样的所有预定义模板代码时，这一点特别方便。 其中一种情况是构建 JSON API。 在下面的部分将展示如何做到这一点。

14.11.1 安装 Yii

为您的项目创建一个目录并将工作目录更改为该路径。示例中使用的命令是基于 Unix 的，但在 Windows 中也存在类似的命令。

```
mkdir micro-app
cd micro-app
```

> **注意**： 需要一些 Composer 的知识才能继续。如果您还不知道如何使用 composer，请花些时间阅读 Composer 指南[59]。

使用您最喜爱的编辑器在 `micro-app` 目录下创建 `composer.json` 文件并添加以下内容：

```
{
    "require": {
        "yiisoft/yii2": "~2.0.0"
    },
    "repositories": [
        {
            "type": "composer",
            "url": "https://asset-packagist.org"
        }
    ]
}
```

保存文件并运行 `composer install` 命令。这将安装框架及其所有依赖项。

14.11.2 创建项目结构

安装框架之后，需要为此应用程序创建一个 入口点。入口点是您尝试打开应用程序时将执行的第一个文件。 出于安全原因，建议将入口点文件放在一个单独的目录中，并将其设置为Web根目录。

创建一个 `web` 目录并将 `index.php` 放入其中，内容如下：

```
<?php

// comment out the following two lines when deployed to production
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

[59] https://getcomposer.org/doc/00-intro.md

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

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

$config = require __DIR__ . '/../config.php';
(new yii\web\Application($config))->run();
```

还要创建一个名为 `config.php`的文件，它将包含所有的应用程序配置：

```
<?php
return [
    'id' => 'micro-app',
    // the basePath of the application will be the `micro-app` directory
    'basePath' => __DIR__,
    // this is where the application will find all controllers
    'controllerNamespace' => 'micro\controllers',
    // set an alias to enable autoloading of classes from the 'micro'
    namespace
    'aliases' => [
        '@micro' => __DIR__,
    ],
];
```

> **信息：** 尽管配置可以保存在 `index.php` 文件中，建议单独使用它。 这样它也可以用于控制台应用程序，如下所示。

您的项目现在已经准备进行编码了。尽管由您决定项目目录结构，只要您遵守命名空间即可。

14.11.3 创建第一个控制器

创建一个 `controllers` 目录并添加一个文件 `SiteController.php`，这是默认的控制器将处理没有路径信息的请求。

```
<?php

namespace micro\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return 'Hello World!';
    }
}
```

如果您想为此控制器使用不同的名称，则可以配置 `yii\base\Application::$defaultRoute` 进行更改。 例如，对于 `DefaultController` 将会是 `'defaultRoute' => 'default/index'`。

在这一点上，项目结构应该如下所示：

```
micro-app/
|- composer.json
|- config.php
|- web/
    |- index.php
|- controllers/
    |- SiteController.php
```

如果您尚未设置 Web 服务器，则可能需要查看Web服务器配置文件示例。另一种选择是使用 `yii serve` 命令，它将使用 PHP 内置 web 服务器。 您可以通过以下方式从 `micro-app /` 目录运行它：

```
vendor/bin/yii serve --docroot=./web
```

在浏览器中打开应用程序URL现在应该打印出“Hello World！”，它已经在 `SiteController::actionIndex()` 中返回。

> **信息：** 在我们的示例中，我们已将默认应用程序名称空间 `app` 更改为 `micro`， 以表明您不受此名称的限制（如果您是这样认为）， 然后调整 `controllers namespace` 并设置正确的别名。

14.11.4 创建一个 REST API

为了演示我们的“微框架”的用法，我们将为帖子创建一个简单的 REST API。

为了这个 API 来操作一些数据，我们首先需要一个数据库。 添加数据库连接配置 到应用程序配置：

```
'components' => [
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'sqlite:@micro/database.sqlite',
    ],
],
```

> **信息：** 为了简单，我们在这里使用了一个 sqlite 数据库 请参阅数据库指南 以获取更多选项。

接下来我们创建一个数据库迁移来创建一个帖子表。 确保你有一个单独的配置文件，如上所述，我们需要它来运行下面的控制台命令。 运行以下命令将创建数据库迁移文件 并将迁移应用到数据库：

```
vendor/bin/yii migrate/create --appconfig=config.php create_post_table
--fields="title:string,body:text"
vendor/bin/yii migrate/up --appconfig=config.php
```

在该目录中创建目录 `models` 和 `Post.php` 文件。以下是模型的代码：

```php
<?php

namespace micro\models;

use yii\db\ActiveRecord;

class Post extends ActiveRecord
{
    public static function tableName()
    {
        return '{{post}}';
    }
}
```

> **信息：** 这里创建的模型是一个 ActiveRecord 类，它代表 `post` 表中的数据。 有关更多信息，请参阅活动记录指南。

我们要通过 API 发布帖子，请在 `controllers` 中添加 `PostController`:

```php
<?php

namespace micro\controllers;

use yii\rest\ActiveController;

class PostController extends ActiveController
{
    public $modelClass = 'micro\models\Post';

    public function behaviors()
    {
        // remove rateLimiter which requires an authenticated user to work
        $behaviors = parent::behaviors();
        unset($behaviors['rateLimiter']);
        return $behaviors;
    }
}
```

此时我们的 API 将提供以下 URL：

- `/index.php?r=post` - 列出所有帖子
- `/index.php?r=post/view&id=1` - 显示 ID 为 1 的帖子
- `/index.php?r=post/create` - 创建一个帖子
- `/index.php?r=post/update&id=1` - 更新 ID 为 1 的帖子
- `/index.php?r=post/delete&id=1` - 删除 ID 为 1 的帖子

从这里开始，您可能需要查看以下指南以进一步开发您的应用程序:

- 该 API 目前仅将 urlencoded 表单数据理解为输入，若使其成为真正的 JSON API， 您需要配置 `yii\web\JsonParser`。
- 为了使 URL 更加友好，您需要配置路由。 请参阅 关于REST路由的指南 了解如何执行此操作。
- 请参阅 预览 部分获取更多参考。

Chapter 15

小部件（Widgets）

Chapter 16

助手类（Helpers）

16.1 助手类（Helpers）

> **注意：** 这部分正在开发中。

Yii 提供许多类来简化常见编码，如对字条串或数组的操作， HTML 代码生成，等等。这些助手类被编写在命名空间 `yii\helpers` 下，并且 全是静态类 （就是说它们只包含静态属性和静态方法，而且不能实例化）。

可以通过调用其中一个静态方法来使用助手类，如下：

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

> **注意：** 为了支持 自定义助手类，Yii 将每一个助手类 分隔成两个类：一个基类 (例如 `BaseArrayHelper`) 和一个具体的类 (例如 `ArrayHelper`)。 当使用助手类时，应该仅使用具体的类版本而不使用基类。

16.1.1 核心助手类（Core Helper Classes）

Yii 发布版中提供以下核心助手类：

- ArrayHelper
- Console
- FileHelper
- Html
- HtmlPurifier
- Imagine（由 yii2-imagine 扩展提供）
- Inflector
- Json
- Markdown
- Security
- StringHelper

- Url
- VarDumper

16.1.2 自定义助手类（Customizing Helper Classes）

如果想要自定义一个核心助手类 (例如 `yii\helpers\ArrayHelper`)，你应该创建一个新的类继承 helpers对应的基类 (例如 `yii\helpers\BaseArrayHelper`) 并同样的命 名你的这个类 (例如 `yii\helpers\ArrayHelper`)，包括它的命名空间。这个类 会用来替换框架最初的实现。

下面示例显示了如何自定义 `yii\helpers\ArrayHelper` 类的 `merge()` 方法:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // 你自定义的实现
    }
}
```

将你的类保存在一个名为 `ArrayHelper.php` 的文件中。该文件可以在任何目录，例如 `@app/components`。

接下来，在你的应用程序 入口脚本 处，在引入的 `yii.php` 文件后面 添加以下代码行，用 Yii 自动加载器 来加载自定义类 代替框架的原始助手类:

```
Yii::$classMap['yii\helpers\ArrayHelper'] =
'@app/components/ArrayHelper.php';
```

注意，自定义助手类仅仅用于如果你想要更改助手类中 现有的函数的行为。如果你想为你的应用程序添加附加功能，最好为它创建一个单独的 助手类。

16.2 数组助手类（ArrayHelper）

除了 PHP 中丰富的数组函数集[1]， Yii 数组助手类提供了额外的静态方法，让你更高效地处理数组。

16.2.1 获取值（Getting Values）

用原生PHP从一个对象、数组、或者包含这两者的一个复杂数据结构中获取数据是非常繁琐的。 你首先得使用 `isset` 检查 key 是否存在, 然后如果存在你就获取它，如果不存在， 则提供一个默认返回值:

[1] https://www.php.net/manual/zh/book.array.php

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name :
null;
```

Yii 提供了一个非常方便的方法来做这件事：

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

方法的第一个参数是我们从哪里获取值。第二个参数指定了如何获取数据， 它可以是下述几种类型中的一个：

- 数组键名或者欲从中取值的对象的属性名称；
- 以点号分割的数组键名或者对象属性名称组成的字符串，上例中使用的参数类型就是该类型；
- 返回一个值的回调函数。

回调函数如下例所示：

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

第三个可选的参数如果没有给定值，则默认为 `null`，如下例所示：

```
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

16.2.2 设定值（Setting values）

```
$array = [
    'key' => [
        'in' => ['k' => 'value']
    ]
];

ArrayHelper::setValue($array, 'key.in', ['arr' => 'val']);
// 在 `$array` 中写入值的路径可以被指定为一个数组
ArrayHelper::setValue($array, ['key', 'in'], ['arr' => 'val']);
```

结果，`$array['key']['in']` 的初始值将被新值覆盖

```
[
    'key' => [
        'in' => ['arr' => 'val']
    ]
]
```

如果路径包含一个不存在的键，它将被创建

```
// 如果 `$array['key']['in']['arr0']` 不为空，则该值将被添加到数组中
ArrayHelper::setValue($array, 'key.in.arr0.arr1', 'val');

// 如果你想完全覆盖值 `$array['key']['in']['arr0']`
ArrayHelper::setValue($array, 'key.in.arr0', ['arr1' => 'val']);
```

结果将是

```
[
    'key' => [
        'in' => [
            'k' => 'value',
            'arr0' => ['arr1' => 'val']
        ]
    ]
]
```

16.2.3 从数组中获取值（Take a value from an array）

如果你想获得一个值，然后立即从数组中删除它，你可以使用 `remove` 方法：

```
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

执行代码后，`$array` 将包含 `['options' => [1, 2]]` 且 `$type` 将包含 `A`。 请注意，与 `getValue` 方法不同，`remove` 仅支持简单的键名称。

16.2.4 检查键名的存在（Checking Existence of Keys）

`ArrayHelper::keyExists` 工作原理和 array_key_exists[2] 差不多，除了 它还可支持大小写不敏感的键名比较，比如：

```
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) ||
!ArrayHelper::keyExists('username', $data2, false)) {
    echo "Please provide username.";
}
```

[2] https://www.php.net/manual/zh/function.array-key-exists.php

16.2.5 检索列（Retrieving Columns）

通常你要从多行数据或者多个对象构成的数组中获取某列的值，一个普通的例子是获取 id 值列表。

```
$data = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

结果将是 `['123', '345']`。

如果需要额外的转换或者取值的方法比较复杂， 第二参数可以指定一个匿名函数:

```
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

16.2.6 重建数组索引（Re-indexing Arrays）

按一个指定的键名重新索引一个数组，可以用 `index` 方法。输入的数组应该是多维数组或者是一个对象数组。 键名（译者注：第二个参数）可以是子数组的键名、对象的属性名， 也可以是一个返回给定元素数组键值的匿名函数。

`$groups` 属性是一个键数组， 它将根据指定的键将输入数组分组为一个或多个子数组。

如果 `$key` 属性或其特定元素的值为 null，并且未定义 `$groups`， 则数组元素将被丢弃。否则，如果指定了 `$groups`， 则数组元素将被添加到没有任何键的结果数组中。

例如:

```
$array = [
    ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
    ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
];
$result = ArrayHelper::index($array, 'id');
```

结果将是一个关联数组，其中键是 `id` 属性的值

```
[
    '123' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    '345' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
    // 原始数组的第二个元素由于相同的 ID 而被最后一个元素覆盖
]
```

匿名函数作为 `$key` 传递，给出了相同的结果。

```
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

传递 id 作为第三个参数将 id 分配给 $ array:

```
$result = ArrayHelper::index($array, null, 'id');
```

结果将是一个多维数组，它由第一级的 id 分组，并且不在第二级索引:

```
[
    '123' => [
        ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
    ],
    '345' => [ // all elements with this index are present in the result
    array
        ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
        ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
    ]
]
```

匿名函数也可用于分组数组中:

```
$result = ArrayHelper::index($array, 'data', [function ($element) {
    return $element['id'];
}, 'device']);
```

结果将是一个多维数组，由第一级的 id 分组，第二级的 device 和第三级的 data 索引:

```
[
    '123' => [
        'laptop' => [
            'abc' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
        ]
    ],
    '345' => [
        'tablet' => [
            'def' => ['id' => '345', 'data' => 'def', 'device' => 'tablet']
        ],
        'smartphone' => [
            'hgi' => ['id' => '345', 'data' => 'hgi', 'device' =>
            'smartphone']
        ]
    ]
]
```

16.2.7 建立哈希表（Building Maps）

为了从一个多维数组或者一个对象数组中建立一个映射表（键值对），你可以使用 map 方法。$from 和 $to 参数分别指定了欲构建的映射表的键名和属性名。 根据需要，你可以按照一个分组字段 $group 将映射表进行分组，例如，

```
$array = [
    ['id' => '123', 'name' => 'aaa', 'class' => 'x'],
    ['id' => '124', 'name' => 'bbb', 'class' => 'x'],
    ['id' => '345', 'name' => 'ccc', 'class' => 'y'],
];

$result = ArrayHelper::map($array, 'id', 'name');
// 结果是:
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// 结果是:
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
//         '345' => 'ccc',
//     ],
// ]
```

16.2.8 多维排序（Multidimensional Sorting）

`multisort` 方法可用来对嵌套数组或者对象数组进行排序，可按一到多个键名排序，比如，

```
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

排序之后我们在 `$data` 中得到的值如下所示：

```
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

第二个参数指定排序的键名，如果是单键名的话可以是字符串，如果是多键名则是一个数组， 或者是如下例所示的一个匿名函数：

```
ArrayHelper::multisort($data, function($item) {
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

第三个参数表示增降顺序。单键排序时，它可以是 SORT_ASC 或者 SORT_DESC 之一。如果是按多个键名排序，你可以用一个数组为 各个键指定不同的顺序。

最后一个参数（译者注：第四个参数）是PHP的排序标识（sort flag），可使用的值和调用 PHP sort()[3] 函数时传递的值一样。

16.2.9 检测数组类型（Detecting Array Types）

想知道一个数组是索引数组还是联合数组很方便，这有个例子：

```
// 不指定键名的数组
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// 所有键名都是字符串
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

16.2.10 HTML 编码和解码值（HTML Encoding and Decoding Values）

为了将字符串数组中的特殊字符做 HTML 编解码，你可以使用下列方法：

```
$encoded = ArrayHelper::htmlEncode($data);
$decoded = ArrayHelper::htmlDecode($data);
```

默认情况只会对值做编码（译者注：原文中是编码，应为编解码）。通过给第二个参数传 false ，你也可以对键名做编码。 编码将默认使用应用程序的字符集，你可以通过第三个参数指定该字符集。

16.2.11 合并数组（Merging Arrays）

您可以使用 ArrayHelper::merge() 将两个或多个数组合并成一个递归的数组。 如果每个数组都有一个具有相同字符串键值的元素，则后者将覆盖前者 （不同于 array_merge_recursive()[4]） 。 如果两个数组都有一个数组类型的元素并且具有相同的键，则将执行递归合并。 对于整数键的元素，来自后一个数组的元素将被附加到前一个数组。 您可以使用 yii\helpers\UnsetArrayValue 对象来取消前一个数组的值或 yii\helpers\ReplaceArrayValue 以强制替换先前的值而不是递归合并。

例如：

```
$array1 = [
    'name' => 'Yii',
    'version' => '1.1',
    'ids' => [
        1,
```

[3] https://www.php.net/manual/zh/function.sort.php
[4] https://www.php.net/manual/zh/function.array-merge-recursive.phpp

```
    ],
    'validDomains' => [
        'example.com',
        'www.example.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
        'dev' => 'dev@example.com',
    ],
];

$array2 = [
    'version' => '2.0',
    'ids' => [
        2,
    ],
    'validDomains' => new \yii\helpers\ReplaceArrayValue([
        'yiiframework.com',
        'www.yiiframework.com',
    ]),
    'emails' => [
        'dev' => new \yii\helpers\UnsetArrayValue(),
    ],
];

$result = ArrayHelper::merge($array1, $array2);
```

结果将是:

```
[
    'name' => 'Yii',
    'version' => '2.0',
    'ids' => [
        1,
        2,
    ],
    'validDomains' => [
        'yiiframework.com',
        'www.yiiframework.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
    ],
]
```

16.2.12 对象转换为数组（Converting Objects to Arrays）

你经常要将一个对象或者对象的数组转换成一个数组，常见的情形是，为了通过 REST API 提供数据数组（或其他使用方式），将 AR 模型（活动记录模型）转换成数组。如下代码可完成这个工作:

```
$posts = Post::find()->limit(10)->all();
$data = ArrayHelper::toArray($posts, [
```

```
    'app\models\Post' => [
        'id',
        'title',
        // the key name in array result => property name
        'createTime' => 'created_at',
        // the key name in array result => anonymous function
        'length' => function ($post) {
            return strlen($post->content);
        },
    ],
]);
```

第一个参数包含我们想要转换的数据，在本例中，我们要转换一个叫 `Post` 的 AR 模型。

第二个参数是每个类的转换映射表，我们在此设置了一个 `Post` 模型的映射。每个映射数组包含一组的映射，每个映射可以是：

- 一个要包含的照原样的字段名（和类中属性的名称一致）；
- 一个由你可随意取名的键名和你想从中取值的模型列名组成的键值对；
- 一个由你可随意取名的键名和有返回值的回调函数组成的键值对；

这上面的转换结果将会是：

```
[
    'id' => 123,
    'title' => 'test',
    'createTime' => '2013-01-01 12:00AM',
    'length' => 301,
]
```

也可以在一个特定的类中实现 `Arrayable` 接口， 从而为其对象提供默认的转换成数组的方法。

16.2.13 测试阵列（Testing against Arrays）

通常你需要检查一个元素是否在数组中，或者一组元素是另一个元素的子集。 虽然PHP提供 `in_array()`，这不支持子集或 `\Traversable` 对象。

为了支持这些测试，`yii\base\ArrayHelper` 提供了 `yii\base\ArrayHelper::isIn()` 和 `yii\base\ArrayHelper::isSubset()` 与 `in_array()` 签名相同。

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new(ArrayObject['a']));

// true
ArrayHelper::isSubset(new(ArrayObject['a', 'c']), new(ArrayObject['a', 'b',
'c'])
```

16.3 Html 帮助类（Html helper）

任何一个 web 应用程序会生成很多 HTMl 超文本标记。如果超文本标记是静态的， 那么将 PHP 和 HTML 混合在一个文件里[5] 这种做法是非常高效的。但是，如果这些超文本标记是动态生成的，那么如果没有额外的辅助工具，这个过程将会变得复杂。 Yii 通过 HTML 帮助类来提供生成超文本标记的方法。这个帮助类包含有一系列的用于处理通用的 HTML 标签和其属性以及内容的静态方法。

> **注意：** 如果你的超文本标记接近静态的，那么最好是直接使用 HTML。 没有必要把所有的超文本标记都用 HTML 辅助类来生成。

16.3.1 基础（Basics）

由于通过字符串连接来生成动态的 HTML 会很容易变得凌乱， Yii 提供了一系列的静态方法来操作标签配置并基于这些配置来创建对应的标签。

生成标签（Generating Tags）

生成一个标签的代码类似如下：

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

这个方法的第一个参数是标签名称。第二个是要装入到开始和结束标签间的内容。 注意到我们使用 `Html::encode` 。那是因为内容不会被自动的转码以允许在有需要的时候嵌套 HTML。 第三个参数是一个 HTML 配置数组，或者换言之，标签属性。在这个数组中，数组的下标是属性名称， 比如 `class`，`href` 或者 `target`，而值则是对应属性的值。

以上代码会生成如下 HTML ：

```
<p class="username">samdark</p>
```

如果你只需要开启一个标签或者关闭一个标签，你可以使用 `Html::beginTag()` 和 `Html::endTag()` 方法。

标签属性（ Options ）在 Html 帮助类很多方法和大量的小部件中都有使用。在这些情况下， 有一些额外的处理我们需要知道：

- 如果一个值为 null ，那么对应的属性将不会被渲染。
- 如果是布尔类型的值的属性，将会被当做 布尔属性[6] 来处理。
- 属性的值将会用 `Html::encode()` 方法进行 HTML 转码处理。
- 如果一个属性的值是一个数组，那么它将会被如下处理：

[5] https://www.php.net/manual/zh/language.basic-syntax.phpmode.php

[6] https://html.spec.whatwg.org/multipage/common-microsyntaxes.html#boolean-attributes

  - 如果这个属性是一个如 `yii\helpers\Html::$dataAttributes` 所列的数据属性， 比如 `data` 或者 `ng`，一系列的属性列表将会被渲染，每个代表值数组中的元素。 比如： `'data' => ['id' => 1, 'name' => 'yii']` 将会生成 `data-id="1" data-name="yii"`； `'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok']` 生成 `data-params='{"id":1,"name":"yii"}' data-status="ok"`。 注意后者中，一个子数组被输出为 JSON 。
  - 如果这个属性不是一个数据属性，那么值将会被 JSON-encoded。 比如： `['params' => ['id' => 1, 'name' => 'yii']` 生成 `params='{"id":1,"name":"yii"}'`。

生成 CSS 类和样式（Forming CSS Classes and Styles）

当开始构造一个 HTML 标签的属性时，我们经常需要对默认的属性进行修改。 为了添加或者删除 CSS 类，你可以使用如下代码：

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Pwede na', $options);

// in case of $type of 'success' it will render
// <div class="btn btn-success">Pwede na</div>
```

基于同样的目的，针对 `style` 属性：

```
$options = ['class' => ['btn', 'btn-default']];

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

在添加或删除类时，您也可以使用数组格式：

```
$options = ['class' => 'btn'];

if ($type === 'success') {
    Html::addCssClass($options, ['btn-success', 'btn-lg']);
}

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-success btn-lg">Save</div>'
```

`Html::addCssClass()` 防止重复类，所以你不必担心同一个类可能会出现两次：

```
$options = ['class' => 'btn btn-default'];
```

```
Html::addCssClass($options, 'btn-default'); // class 'btn-default' is
already present

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

如果通过数组格式指定 CSS 类选项，则可以使用命名键来标记该类的逻辑用途。 在这种情况下，在 `Html::addCssClass()` 类中会忽略数组格式中具有相同键：

```
$options = [
    'class' => [
        'btn',
        'theme' => 'btn-default',
    ]
];

Html::addCssClass($options, ['theme' => 'btn-success']); // 'theme' 键已被使
用

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

可以使用 `style` 属性以类似的方式设置 CSS 样式：

```
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// gives style="width: 100px; height: 200px; position: absolute;"
Html::addCssStyle($options, 'height: 200px; position: absolute;');

// gives style="position: absolute;"
Html::removeCssStyle($options, ['width', 'height']);
```

当使用 `addCssStyle()` 方法时，你可以指定一个和 CSS 属性相关的名值对的数组， 也可以直接是一个类似 `width: 100px; height: 200px;` 的字符串。这些格式将会自动的被 `cssStyleFromArray()` 和`cssStyleToArray()` 方法进行转换。方法 `removeCssStyle()` 接收一个包含要被移除的属性数组作为参数。 如果只想移除一个属性，你可以直接传递一个字符串。

标签内容的转码和解码（Encoding and Decoding Content）

为了让内容能够正确安全的显示，一些 HTML 特殊字符应该被转码。在 PHP 中， 这个操作由 htmlspecialchars[7] 和 htmlspecialchars_decode[8] 完成。 直接使用这些方法的问题是，你总是需要指定转码所需的额外标志。由于标志一般总是不变的，而内容转码的过程为了避免一些安全问题， 需要和应用的默认过程匹配， Yii 提供了两个简单可用的对 PHP 原生方法的封装：

[7] https://www.php.net/manual/zh/function.htmlspecialchars.php
[8] https://www.php.net/manual/zh/function.htmlspecialchars-decode.php

```
$userName = Html::encode($user->name);
echo $userName;

$decodedUserName = Html::decode($userName);
```

16.3.2 表单（Forms）

处理表单标签是大量的重复性劳动并且易错。因此， Yii 也提供了一系列的方法来辅助处理表单标签。

> **注意：** 考虑在处理 models 以及需要验证的情形下，使用 `ActiveForm` 组件。

创建表单（Creating Forms）

表单可以用类似如下代码，使用 `beginForm()` 方法开启：

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' =>
'multipart/form-data']) ?>
```

方法的第一个参数为表单将要被提交的 URL 地址。它可以以 Yii 路由的形式被指定，并由 `Url::to()` 来接收处理。 第二个参数是使用的方法，默认为 `post` 方法。第三个参数为表单标签的属性数组。在上面的例子中， 我们把编码 POST 请求中的表单数据的方式改为 `multipart/form-data`。 如果是上传文件，这个调整是必须的。

关闭表单标签非常简单：

```
<?= Html::endForm() ?>
```

按钮（Buttons）

你可以用如下代码生成按钮：

```
<?= Html::button('Press me!', ['class' => 'teaser']) ?>
<?= Html::submitButton('Submit', ['class' => 'submit']) ?>
<?= Html::resetButton('Reset', ['class' => 'reset']) ?>
```

上述三个方法的第一个参数为按钮的标题，第二个是标签属性。标题默认没有进行转码，如果标题是由终端用输入的， 那么请自行用 `Html::encode()` 方法进行转码。

输入栏（Input Fields）

input 相关的方法有两组：以 `active` 开头的被称为 active inputs， 另一组则不以其开头。active inputs 依据指定的模型和属性获取数据， 而普通 input 则是直接指定数据。

一般用法如下：

```
type, input name, input value, options
<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>

type, model, model attribute name, options
<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

如果你知道 input 类型，更方便的做法是使用以下快捷方法：

- `yii\helpers\Html::buttonInput()`
- `yii\helpers\Html::submitInput()`
- `yii\helpers\Html::resetInput()`
- `yii\helpers\Html::textInput()`, `yii\helpers\Html::activeTextInput()`
- `yii\helpers\Html::hiddenInput()`, `yii\helpers\Html::activeHiddenInput()`
- `yii\helpers\Html::passwordInput()` / `yii\helpers\Html::activePasswordInput()`
- `yii\helpers\Html::fileInput()`, `yii\helpers\Html::activeFileInput()`
- `yii\helpers\Html::textarea()`, `yii\helpers\Html::activeTextarea()`

Radios 和 checkboxes 在方法的声明上有一点点不同：

```
<?= Html::radio('agree', true, ['label' => 'I agree']);
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement'])

<?= Html::checkbox('agree', true, ['label' => 'I agree']);
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement'])
```

Dropdown list 和 list box 将会如下渲染：

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels,
'id', 'name')) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id',
'name')) ?>
```

第一个参数是 input 的名称，第二个是当前选中的值，第三个则是一个下标为列表值，值为列表标签的名值对数组。

如果你需要使用多项选择， checkbox list 应该能够符合你的需求：

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels,
'id', 'name')) ?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

否则，用 radio list ：

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id',
'name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

Labels 和 Errors（Labels and Errors）

如同 inputs 一样，Yii 也提供了两个方法用于生成表单 label。带 ative 方法用于从 model 中取数据，另外一个则是直接接收数据。

```
<?= Html::label('User name', 'username', ['class' => 'label username']) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username'])
```

为了从一个或者一组 model 中显示表单的概要错误，你可以使用如下方法：

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

为了显示单个错误：

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```

Input 的名和值（Input Names and Values）

Yii 提供了方法用于从 model 中获取 input 的名称，ids，值。这些主要用于内部调用， 但是有时候你也需要使用它们：

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// my first post
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

在上面的例子中，第一个参数为模型，而第二个参数是属性表达式。在最简单的表单中，这个属性表达式就是属性名称，但是在一些多行输入的时候，它也可以是属性名以数组下标前缀或者后缀（也可能是同时）。

- `[0]content` 代表多行输入时第一个 model 的 content 属性的数据值。
- `dates[0]` 代表 dates 属性的第一个数组元素。
- `[0]dates[0]` 代表多行输入时第一个 model 的 dates 属性的第一个数组元素。

为了获取一个没有前缀或者后缀的属性名称，我们可以如下做：

```
// dates
echo Html::getAttributeName('dates[0]');
```

16.3.3 样式表和脚本（Styles and Scripts）

Yii 提供两个方法用于生成包含内联样式和脚本代码的标签。

```
<?= Html::style('.danger { color: #f00; }') ?>

Gives you

<style>.danger { color: #f00; }</style>


<?= Html::script('alert("Hello!");', ['defer' => true]);

Gives you

<script defer>alert("Hello!");</script>
```

如果你想要外联 css 样式文件，可以如下做：

```
<?= Html::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>

generates

<!--[if IE 5]>
    <link href="https://example.com/css/ie5.css" />
<![endif]-->
```

第一个参数是 URL。第二个参数是标签属性数组。比普通的标签配置项额外多出的是，你可以指定：

- `condition` 来让 `<link` 被条件控制注释包裹（ IE hacker ）。希望你在未来不再需要条件控制注释。
- `noscript` 可以被设置为 `true` ，这样 `<link`就会被 `<noscript>`包裹，如此那么这段代码只有在浏览器不支持 JavaScript 或者被用户禁用的时候才会被引入进来。

为了外联 JavaScript 文件：

```
<?= Html::jsFile('@web/js/main.js') ?>
```

这个方法的第一个参数同 CSS 一样用于指定外联链接。第二个参数是一个标签属性数组。 同 `cssFile` 一样，你可以指定 `condtion` 配置项。

16.3.4 超链接（Hyperlinks）

有一个方法可以用于便捷的生成超链接：

```
<?= Html::a('Profile', ['user/view', 'id' => $id], ['class' =>
'profile-link']) ?>
```

第一个参数是超链接的标题。它不会被转码，所以如果是用户输入数据，你需要使用 `Html::encode()` 方法进行转码。第二个参数是 `<a` 标签的 `href` 属

性的值。 关于该参数能够接受的更详细的数据值，请参阅 Url::to()。 第三个参数是标签的属性数组。

在需要的时候，你可以用如下代码生成 `mailto` 链接：

```
<?= Html::mailto('Contact us', 'admin@example.com') ?>
```

16.3.5 图片（Images）

为了生成图片标签，你可以如下做：

```
<?= Html::img('@web/images/logo.png', ['alt' => 'My logo']) ?>
```

```
generates
```

```
<img src="https://example.com/images/logo.png" alt="My logo" />
```

除了 aliases 之外，第一个参数可以接受 路由，查询，URLs。同 Url::to() 一样。

16.3.6 列表（Lists）

无序列表可以如下生成：

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
        ['class' => 'post']
    );
}]) ?>
```

有序列表请使用 `Html::ol()` 方法。

16.4 Url 助手类（Url Helper）

Url 帮助类提供一系列的静态方法来帮助管理 URL。

16.4.1 获得通用 URL（Getting Common URLs）

有两种获取通用 URLS 的方法 ： 当前请求的 home URL 和 base URL 。 为了获取 home URL ， 使用如下代码：

```
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

如果没有传任何参数，这个方法将会生成相对 URL 。你可以传 `true` 来获得一个针对当前协议的绝对 URL； 或者，你可以明确的指定具体的协议类型（ `https` , `http` ）。

如下代码可以获得当前请求的 base URL：

`php $relativeBaseUrl = Url::base(); $absoluteBaseUrl = Url::base(true); $httpsAbsoluteBaseUrl = Url::base('https'); `

这个方法的调用方式和 `Url::home()` 的完全一样。

16.4.2 创建 URLs（Creating URLs）

为了创建一个给定路由的 URL 地址，请使用 `Url::toRoute()`方法。 这个方法使用 `yii\web\UrlManager` 来创建一个 URL ：

```
$url = Url::toRoute(['product/view', 'id' => 42]);
```

你可以指定一个字符串来作为路由，如： `site/index` 。如果想要指定将要被创建的 URL 的附加查询参数， 你同样可以使用一个数组来作为路由。数组的格式须为：

```
// generates: /index.php?r=site/index&param1=value1&param2=value2
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

如果你想要创建一个带有 anchor 的 URL ，你可以使用一个带有 `#` 参数的数组。比如：

```
// generates: /index.php?r=site/index&param1=value1#name
['site/index', 'param1' => 'value1', '#' => 'name']
```

一个路由既可能是绝对的又可能是相对的。一个绝对的路由以前导斜杠开头（如： `/site/index`） ， 而一个相对的路由则没有（比如： `site/index` 或者 `index`）。一个相对的路由将会按照如下规则转换为绝对路由：

- 如果这个路由是一个空的字符串，将会使用当前 `route` 作为路由；
- 如果这个路由不带任何斜杠（比如 `index` ） ，它会被认为是当前控制器的一个 action ID， 然后将会把 `yii\web\Controller::$uniqueId` 插入到路由前面。
- 如果这个路由不带前导斜杠（比如： `site/index` ） ，它会被认为是相对当前模块（module）的路由， 然后将会把 `uniqueId` 插入到路由前面。

从2.0.2版本开始，你可以用 alias 来指定一个路由。 在这种情况下， alias 将会首先转换为实际的路由， 然后会按照上述规则转换为绝对路由。

以下是该方法的一些例子：

```
// /index.php?r=site/index
echo Url::toRoute('site/index');

// /index.php?r=site/index&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);
```

```
// /index.php?r=post/edit&id=100      假设别名 "@postEdit" 被定义为
"post/edit"
echo Url::toRoute(['@postEdit', 'id' => 100]);

// https://www.example.com/index.php?r=site/index
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site/index
echo Url::toRoute('site/index', 'https');
```

还有另外一个方法 `Url::to()` 和 `toRoute()` 非常类似。这两个方法的唯一区别在于，前者要求一个路由必须用数组来指定。 如果传的参数为字符串，它将会被直接当做 URL 。

`Url::to()` 的第一个参数可以是：

- 数组：将会调用 `toRoute()` 来生成URL。比如： `['site/index']`, `['post/index', 'page' => 2]` 。 详细用法请参考 `toRoute()` 。
- 带前导 `@` 的字符串：它将会被当做别名， 对应的别名字符串将会返回。
- 空的字符串：当前请求的 URL 将会被返回；
- 普通的字符串：返回本身。

当 `$scheme` 指定了（无论是字符串还是 true ），一个带主机信息（通过 `yii\web\UrlManager::$hostInfo` 获得） 的绝对 URL 将会被返回。如果 `$url` 已经是绝对 URL 了， 它的协议信息将会被替换为指定的（ https 或者 http ）。

以下是一些使用示例：

```
// /index.php?r=site/index
echo Url::to(['site/index']);

// /index.php?r=site/index&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post/edit&id=100      假设别名 "@postEdit" 被定义为
"post/edit"
echo Url::to(['@postEdit', 'id' => 100]);

// 当前请求的 URL
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

从2.0.3版本开始，你可以使用 `yii\helpers\Url::current()` 来创建一个基于当前请求路由和 GET 参数的 URL。 你可以通过传递一个 `$params` 给这个方法来添加或者删除 GET 参数。 例如：

```
// 假设 $_GET = ['id' => 123, 'src' => 'google']，当前路由为 "post/view"

// /index.php?r=post/view&id=123&src=google
echo Url::current();

// /index.php?r=post/view&id=123
echo Url::current(['src' => null]);
// /index.php?r=post/view&id=100&src=google
echo Url::current(['id' => 100]);
```

16.4.3 记住 URLs（Remember URLs）

有时，你需要记住一个 URL 并在后续的请求处理中使用它。 你可以用以下方式达到这个目的：

```
// 记住当前 URL
Url::remember();

// 记住指定的 URL。参数格式请参阅 Url::to()。
Url::remember(['product/view', 'id' => 42]);

// 记住用给定名称指定的 URL
Url::remember(['product/view', 'id' => 42], 'product');
```

在后续的请求处理中，可以用如下方式获得记住的 URL：

```
$url = Url::previous();
$productUrl = Url::previous('product');
```

16.4.4 检查相对 URLs（Checking Relative URLs）

你可以用如下代码检测一个 URL 是否是相对的（比如，包含主机信息部分）。

```
$isRelative = Url::isRelative('test/it');
```