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。

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。

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。

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。

系统要求和先决条件

Yii 2.0 需要 PHP 5.4.0 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，例如命名空间和Trait（特质）。理解这些概念将有助于你更快地掌握 Yii 2.0。

配置

在 Yii 中，创建新对象和初始化已存在对象时广泛使用配置。配置通常包含被创建对象的类名和一组将要赋值给对象属性的初始值。还可能包含一组将被附加到对象事件上的句柄。和一组将被附加到对象上的行为。

以下代码中的配置被用来创建并初始化一个数据库连接：

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];

$db = Yii::createObject($config);
```

[[Yii::createObject()]] 方法接受一个配置数组并根据数组中指定的类名创建对象。对象实例化后，剩余的参数被用来初始化对象的属性，事件处理和行为。

对于已存在的对象，可以使用 [[Yii::configure()]] 方法根据配置去初始化其属性，就像这样：

```
Yii::configure($object, $config);
```

请注意，如果配置一个已存在的对象，那么配置数组中不应该包含指定类名的 `class` 元素。

配置的格式

一个配置的格式可以描述为以下形式：

```
[
    'class' => 'ClassName',
    'propertyName' => 'propertyValue',
```

```
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

其中

- `class` 元素指定了将要创建的对象的完全限定类名。
- `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 ...
    ],
]
```

使用配置

Yii 中的配置可以用在很多场景。本章开头我们展示了如何使用 [[Yii::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

应用的配置可能是最复杂的配置之一。因为 [[yii\web\Application|application]] 类拥

有很多可配置的属性和事件。更重要的是它的 [[yii\web\Application::components|components]] 属性可以接收配置数组并通过应用注册为组件。以下是一个针对基础应用模板的应用配置概要：

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.p
    '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。

小部件的配置

使用小部件时，常常需要配置以便自定义其属性。 [[yii\base\Widget::widget()]] 和 [[yii\base\Widget::begin()]] 方法都可以用来创建小部件。它们可以接受配置数组：

```
use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'Products', 'url' => ['product/index']],
        ['label' => 'Login', 'url' => ['site/login'], 'visible' => Yii
    ],
]);
```

上述代码创建了一个小部件 `Menu` 并将其 `activateItems` 属性初始化为 false。 `item` 属性也配置成了将要显示的菜单条目。

请注意，代码中已经给出了类名 `yii\widgets\Menu'，配置数组**不应该**再包含` class` 键。

配置文件

当配置的内容十分复杂，通用做法是将其存储在一或多个 PHP 文件中，这些文件被称为**配置文件**。一个配置文件返回的是 PHP 数组。例如，像这样把应用配置信息存储在名为 `web.php` 的文件中：

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.p
    'components' => require(__DIR__ . '/components.php'),
];
```

鉴于 `components` 配置也很复杂，上述代码把它们存储在单独的 `components.php` 文件中，并且包含在 `web.php` 里。 `components.php` 的内容如下：

```
return [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
```

```
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
    'log' => [
        'class' => 'yii\log\Dispatcher',
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
            ],
        ],
    ],
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'mysql:host=localhost;dbname=stay2',
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
    ],
];
```

仅仅需要 “require”，就可以取得一个配置文件的配置内容，像这样：

```
$config = require('path/to/web.php');
(new yii\web\Application($config))->run();
```

默认配置

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

环境常量

配置经常要随着应用运行的不同环境更改。例如在开发环境中，你可能使用名为 `mydb_dev` 的数据库，而生产环境则使用 `mydb_prod` 数据库。为了便于切换使用环境，Yii 提供了一个定义在入口脚本中的 `YII_ENV` 常量。如下：

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

你可以把 `YII_ENV` 定义成以下任何一种值：

- `prod`：生产环境。常量 `YII_ENV_PROD` 将被看作 true。如果你没修改过，这就是 `YII_ENV` 的默认值。
- `dev`：开发环境。常量 `YII_ENV_DEV` 将被看作 true。
- `test`：测试环境。常量 `YII_ENV_TEST` 将被看作 true。

有了这些环境常量，你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;
```

别名（Aliases）

别名用来表示文件路径和 URL，这样就避免了在代码中硬编码一些绝对路径和 URL。一个别名必须以 `@` 字符开头，以区别于传统的文件路径和 URL。Yii 预定义了大量可用的别名。例如，别名 `@yii` 指的是 Yii 框架本身的安装目录，而 `@web` 表示的是当前运行应用的根 URL。

定义别名

你可以调用 [[Yii::setAlias()]] 来给文件路径或 URL 定义别名：

```
// 文件路径的别名
Yii::setAlias('@foo', '/path/to/foo');

// URL 的别名
Yii::setAlias('@bar', 'http://www.example.com');
```

> Note: 别名所指向的文件路径或 URL 不一定是真实存在的文件或资源。

可以通过在一个别名后面加斜杠 `/` 和一至多个路径分段生成新别名（无需调用 [[Yii::setAlias()]]）。我们把通过 [[Yii::setAlias()]] 定义的别名称为**根别名**，而用他们衍生出去的别名成为**衍生别名**。例如， `@foo` 就是根别名，而 `@foo/bar/file.php` 是一个衍生别名。

你还可以用别名去定义新别名（根别名与衍生别名均可）：

```
Yii::setAlias('@foobar', '@foo/bar');
```

根别名通常在引导阶段定义。比如你可以在入口脚本里调用 [[Yii::setAlias()]]。为了方便起见，应用提供了一个名为 `aliases` 的可写属性，你可以在应用配置中设置它，就像这样：

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
```

```
        '@bar' => 'http://www.example.com',
    ],
];
```

解析别名

你可以调用 [[Yii::getAlias()]] 命令来解析根别名到对应的文件路径或 URL。同样的页面也可以用于解析衍生别名。例如：

```
echo Yii::getAlias('@foo');               // 输出：/path/to/foo
echo Yii::getAlias('@bar');               // 输出：http://www.example.c
echo Yii::getAlias('@foo/bar/file.php');  // 输出：/path/to/foo/bar/fil
```

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。

> Note: [[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/f
echo Yii::getAlias('@foo/bar/file.php');   // 输出：/path2/bar/file.php
```

若 `@foo/bar` 未被定义为根别名，最后一行语句会显示为 `/path/to/foo/bar/file.php`。

使用别名

别名在 Yii 的很多地方都会被正确识别，无需调用 [[Yii::getAlias()]] 来把它们转换为路径/URL。例如，[[yii\caching\FileCache::cachePath]] 能同时接受文件路径或是指向文件路径的别名，因为通过 `@` 前缀能区分它们。

```
use yii\caching\FileCache;
```

```php
$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

请关注 API 文档了解特定属性或方法参数是否支持别名。

预定义的别名

Yii 预定义了一系列别名来简化常用路径和 URL 的使用：

- `@yii` - `BaseYii.php` 文件所在的目录（也被称为框架安装目录）
- `@app` - 当前运行的应用 [[yii\base\Application::basePath|根路径（base path）]]
- `@runtime` - 当前运行的应用的 [[yii\base\Application::runtimePath|运行环境（runtime）路径]]
- `@vendor` - [[yii\base\Application::vendorPath|Composer 供应商目录]]
- `@webroot` - 当前运行应用的 Web 入口目录
- `@web` - 当前运行应用的根 URL

`@yii` 别名是在入口脚本里包含 `Yii.php` 文件时定义的，其他的别名都是在配置应用的时候，于应用的构造方法内定义的。

扩展的别名

每一个通过 Composer 安装的 扩展 都自动添加了一个别名。该别名会以该扩展在 `composer.json` 文件中所声明的根命名空间为名，且他直接代指该包的根目录。例如，如果你安装有 `yiisoft/yii2-jui` 扩展，会自动得到 `@yii/jui` 别名，它定义于引导启动阶段：

```php
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

组件（Component）

组件是 Yii 应用的主要基石。是 [[yii\base\Component]] 类或其子类的实例。三个用以区分它和其它类的主要功能有：

- 属性（Property）
- 事件（Event）
- 行为（Behavior）

或单独使用，或彼此配合，这些功能的应用让 Yii 的类变得更加灵活和易用。以小部件 [[yii\jui\DatePicker|日期选择器]] 来举例，这是个方便你在 视图 中生成一个交互式日期选择器的 UI 组件：

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'zh-CN',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

这个小部件继承自 [[yii\base\Component]]，它的各项属性改写起来会很容易。

正是因为组件功能的强大，他们比常规的对象（Object）稍微重量级一点，因为他们要使用额外的内存和 CPU 时间来处理 事件 和 行为 。如果你不需要这两项功能，可以继承 [[yii\base\Object]] 而不是 [[yii\base\Component]]。这样组件可以像普通 PHP 对象一样高效，同时还支持属性（Property）功能。

当继承 [[yii\base\Component]] 或 [[yii\base\Object]] 时，推荐你使用如下的编码风格：

- 若你需要重写构造方法（Constructor），传入 `$config` 作为构造器方法**最后一个**参数，然后把它传递给父类的构造方法。
- 永远在你重写的构造方法**结尾处**调用一下父类的构造方法。
- 如果你重写了 [[yii\base\Object::init()]] 方法，请确保你在 `init` 方法的**开头处**调用了父类的 `init` 方法。

例子如下：

```
namespace yii\components\MyClass;

use yii\base\Object;

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

    public function __construct($param1, $param2, $config = [])
    {
        // ... 配置生效前的初始化过程

        parent::__construct($config);
    }

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

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

另外，为了让组件可以在创建实例时能被正确配置，请遵照以下操作流程：

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// 方法二：
$component = \Yii::createObject([
    'class' => MyClass::className(),
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> Info: 尽管调用 [[Yii::createObject()]] 的方法看起来更加复杂，但这主要因为它更加灵活强大，它是基于依赖注入容器实现的。

[[yii\base\Object]] 类执行时的生命周期如下：

1. 构造方法内的预初始化过程。你可以在这儿给各属性设置缺省值。
2. 通过 `$config` 配置对象。配置的过程可能会覆盖掉先前在构造方法内设置的

默认值。

3. 在 [[yii\base\Object::init()|init()]] 方法内进行初始化后的收尾工作。你可以通过重写此方法，进行一些良品检验，属性的初始化之类的工作。
4. 对象方法调用。

前三步都是在对象的构造方法内发生的。这意味着一旦你获得了一个对象实例，那么它就已经初始化就绪可供使用。

事件

事件可以将自定义代码“注入”到现有代码中的特定执行点。附加自定义代码到某个事件，当这个事件被触发时，这些代码就会自动执行。例如，邮件程序对象成功发出消息时可触发 `messageSent` 事件。如想追踪成功发送的消息，可以附加相应追踪代码到 `messageSent` 事件。

Yii 引入了名为 [[yii\base\Component]] 的基类以支持事件。如果一个类需要触发事件就应该继承 [[yii\base\Component]] 或其子类。

事件处理器（Event Handlers）

事件处理器是一个PHP 回调函数，当它所附加到的事件被触发时它就会执行。可以使用以下回调函数之一：

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

事件处理器的格式是：

```
function ($event) {
    // $event 是 yii\base\Event 或其子类的对象
}
```

通过 `$event` 参数，事件处理器就获得了以下有关事件的信息：

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

附加事件处理器

调用 [[yii\base\Component::on()]] 方法来附加处理器到事件上。如：

```php
$foo = new Foo;

// 处理器是全局函数
$foo->on(Foo::EVENT_HELLO, 'function_name');

// 处理器是对象方法
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// 处理器是静态类方法
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// 处理器是匿名函数
$foo->on(Foo::EVENT_HELLO, function ($event) {
    //事件处理逻辑
});
```

附加事件处理器时可以提供额外数据作为 [[yii\base\Component::on()]] 方法的第三个参数。数据在事件被触发和处理器被调用时能被处理器使用。如：

```php
// 当事件被触发时以下代码显示 "abc"
// 因为 $event->data 包括被传递到 "on" 方法的数据
$foo->on(Foo::EVENT_HELLO, function ($event) {
    echo $event->data;
}, 'abc');
```

事件处理器顺序

可以附加一个或多个处理器到一个事件。当事件被触发，已附加的处理器将按附加次序依次调用。如果某个处理器需要停止其后的处理器调用，可以设置 `$event` 参数的 [[yii\base\Event::handled]] 属性为真，如下：

```php
$foo->on(Foo::EVENT_HELLO, function ($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：

```php
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // 这个处理器将被插入到处理器队列的第一位...
}, $data, false);
```

触发事件

事件通过调用 [[yii\base\Component::trigger()]] 方法触发，此方法须传递**事件名**，还可以传递一个事件对象，用来传递参数到事件处理器。如：

```php
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。

> Tip: 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：

```php
namespace app\components;

use yii\base\Component;
use yii\base\Event;
```

```php
class MessageEvent extends Event
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ...发送 $message 的逻辑...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}
```

当 [[yii\base\Component::trigger()]] 方法被调用时，它将调用所有附加到命名事件（trigger 方法第一个参数）的事件处理器。

移除事件处理器

从事件移除处理器，调用 [[yii\base\Component::off()]] 方法。如：

```php
// 处理器是全局函数
$foo->off(Foo::EVENT_HELLO, 'function_name');

// 处理器是对象方法
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// 处理器是静态类方法
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// 处理器是匿名函数
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

注意当匿名函数附加到事件后一般不要尝试移除匿名函数，除非你在某处存储了它。以上示例中，假设匿名函数存储为变量 `$anonymousFunction` 。

移除事件的全部处理器，简单调用 [[yii\base\Component::off()]] 即可，不需要第二个参数：

```
$foo->off(Foo::EVENT_HELLO);
```

类级别的事件处理器

以上部分，我们叙述了在**实例级别**如何附加处理器到事件。有时想要一个类的所有实例而不是一个指定的实例都响应一个被触发的事件，并不是一个个附加事件处理器到每个实例，而是通过调用静态方法 [[yii\base\Event::on()]] 在**类级别**附加处理器。

例如，活动记录对象要在每次往数据库新增一条新记录时触发一个 [[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] 事件。要追踪每个活动记录对象的新增记录完成情况，应如下写代码：

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

Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT,
    Yii::trace(get_class($event->sender) . ' is inserted');
});
```

每当 [[yii\db\BaseActiveRecord|ActiveRecord]] 或其子类的实例触发 [[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] 事件时，这个事件处理器都会执行。在这个处理器中，可以通过 `$event->sender` 获取触发事件的对象。

当对象触发事件时，它首先调用实例级别的处理器，然后才会调用类级别处理器。

可调用静态方法[[yii\base\Event::trigger()]]来触发一个**类级别**事件。类级别事件不与特定对象相关联。因此，它只会引起类级别事件处理器的调用。如：

```
use yii\base\Event;

Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // 显示 "null"
});

Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

注意这种情况下 `$event->sender` 指向触发事件的类名而不是对象实例。

> Note: 因为类级别的处理器响应类和其子类的所有实例触发的事件，必须谨慎使用，尤其是底层的基类，如 [[yii\base\Object]]。

移除类级别的事件处理器只需调用[[yii\base\Event::off()]]，如：

```
// 移除 $handler
Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);

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

全局事件

所谓**全局事件**实际上是一个基于以上叙述的事件机制的戏法。它需要一个全局可访问的单例，如应用实例。

事件触发者不调用其自身的 `trigger()` 方法，而是调用单例的 `trigger()` 方法来触发全局事件。类似地，事件处理器被附加到单例的事件。如：

```
use Yii;
use yii\base\Event;
use app\components\Foo;

Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);  // 显示 "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

全局事件的一个好处是当附加处理器到一个对象要触发的事件时，不需要产生该对象。相反，处理器附加和事件触发都通过单例（如应用实例）完成。

然而，因为全局事件的命名空间由各方共享，应合理命名全局事件，如引入一些命名空间（例："frontend.mail.sent", "backend.mail.sent"）。

属性（Property）

在 PHP 中，类的成员变量也被称为**属性（properties）**。它们是类定义的一部分，用来表现一个实例的状态（也就是区分类的不同实例）。在具体实践中，常常会想用一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\Object]] 的基类，它支持基于类内的 **getter** 和 **setter**（读取器和设定器）方法来定义属性。如果某类需要支持这个特性，只需要继承 [[yii\base\Object]] 或其子类即可。

> Info: 几乎每个 Yii 框架的核心类都继承自 [[yii\base\Object]] 或其子类。这意味着只要在核心类中见到 getter 或 setter 方法，就可以像调用属性一样调用它。

getter 方法是名称以 `get` 开头的方法，而 setter 方法名以 `set` 开头。方法名中 `get` 或 `set` 后面的部分就定义了该属性的名字。如下面代码所示，getter 方法 `getLabel()` 和 setter 方法 `setLabel()` 操作的是 `label` 属性，：

```
namespace app\components;

use yii\base\Object;

class Foo extend Object
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

    public function setLabel($value)
```

```
    {
        $this->_label = trim($value);
    }
}
```

（详细解释：getter 和 setter 方法创建了一个名为 `label` 的属性，在这个例子里，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 [[yii\base\InvalidCallException|InvalidCallException]] （无效调用）异常。类似的，只有 setter 方法而没有 getter 方法定义的属性是**只写属性**，尝试读取这种属性也会触发异常。使用只写属性的情况几乎没有。

通过 getter 和 setter 定义的属性也有一些特殊规则和限制：

- 这类属性的名字是**不区分大小写**的。如，`$object->label` 和 `$object->Label` 是同一个属性。因为 PHP 方法名是不区分大小写的。
- 如果此类属性名和类成员变量相同，以后者为准。例如，假设以上 `Foo` 类有个 `label` 成员变量，然后给 `$object->label = 'abc'` 赋值，将赋给成员变量而不是 setter `setLabel()` 方法。
- 这类属性不支持可见性（访问限制）。定义属性的 getter 和 setter 方法是 public、protected 还是 private 对属性的可见性没有任何影响。
- 这类属性的 getter 和 setter 方法只能定义为**非静态**的，若定义为静态方法（static）则不会以相同方式处理。

回到开头提到的问题，与其处处要调用 `trim()` 函数，现在我们只需在 setter `setLabel()` 方法内调用一次。如果 label 首字母变成大写的新要求来了，我们只需要修改 `setLabel()` 方法，而无须接触任何其它代码。

行为

行为是 [[yii\base\Behavior]] 或其子类的实例。行为，也称为 mixins，可以无须改变类继承关系即可增强一个已有的 [[yii\base\Component|组件]] 类功能。当行为附加到组件后，它将“注入”它的方法和属性到组件，然后可以像访问组件内定义的方法和属性一样访问它们。此外，行为通过组件能响应被触发的事件，从而自定义或调整组件正常执行的代码。

定义行为

要定义行为，通过继承 [[yii\base\Behavior]] 或其子类来建立一个类。如：

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

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

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

    public function foo()
    {
        // ...
    }
}
```

以上代码定义了行为类 `app\components\MyBehavior` 并为要附加行为的组件提供了两个属性 `prop1` 、 `prop2` 和一个方法 `foo()` 。注意属性 `prop2` 是通过

getter `getProp2()` 和 setter `setProp2()` 定义的。能这样用是因为 [[yii\base\Object]] 是 [[yii\base\Behavior]] 的祖先类，此祖先类支持用 getter 和 setter 方法定义属性

> Tip: 在行为内部可以通过 [[yii\base\Behavior::owner]] 属性访问行为已附加的组件。

处理事件

如果要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
    }
}
```

[[yii\base\Behavior::events()|events()]] 方法返回事件列表和相应的处理器。上例声明了 [[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]] 事件和它的处理器 `beforeValidate()` 。当指定一个事件处理器时，要使用以下格式之一：

- 指向行为类的方法名的字符串，如上例所示；
- 对象或类名和方法名的数组，如 `[$object, 'methodName']`；

- 匿名方法。

处理器的格式如下，其中 `$event` 指向事件参数。关于事件的更多细节请参考事件：

```
function ($event) {
}
```

附加行为

可以静态或动态地附加行为到[[yii\base\Component|组件]]。前者在实践中更常见。

要静态附加行为，覆写行为要附加的组件类的 [[yii\base\Component::behaviors()|behaviors()]] 方法即可。[[yii\base\Component::behaviors()|behaviors()]] 方法应该返回行为配置列表。每个行为配置可以是行为类名也可以是配置数组。如：

```
namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // 匿名行为，只有行为类名
            MyBehavior::className(),

            // 命名行为，只有行为类名
            'myBehavior2' => MyBehavior::className(),

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

            // 命名行为，配置数组
            'myBehavior4' => [
```

```
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop2' => 'value2',
            ]
        ];
    }
}
```

通过指定行为配置数组相应的键可以给行为关联一个名称。这种行为称为**命名行为**。上例中，有两个命名行为：`myBehavior2` 和 `myBehavior4` 。如果行为没有指定名称就是**匿名行为**。

要动态附加行为，在对应组件里调用 [[yii\base\Component::attachBehavior()]] 方法即可，如：

```
use app\components\MyBehavior;

// 附加行为对象
$component->attachBehavior('myBehavior1', new MyBehavior);

// 附加行为类
$component->attachBehavior('myBehavior2', MyBehavior::className());

// 附加配置数组
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::className(),
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

可以通过 [[yii\base\Component::attachBehaviors()]] 方法一次附加多个行为：

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior,  // 命名行为
    MyBehavior::className(),          // 匿名行为
]);
```

还可以通过配置去附加行为：

```
[
    'as myBehavior2' => MyBehavior::className(),
```

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

详情请参考配置章节。

使用行为

使用行为，必须像前文描述的一样先把它附加到 [[yii\base\Component|component]] 类或其子类。一旦行为附加到组件，就可以直接使用它。

行为附加到组件后，可以通过组件访问一个行为的**公共**成员变量或 getter 和 setter 方法定义的属性：

```php
// "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：

```php
// 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：

```php
$behavior = $component->getBehavior('myBehavior');
```

也能获取附加到这个组件的所有行为：

```
$behaviors = $component->getBehaviors();
```

移除行为

要移除行为，可以调用 [[yii\base\Component::detachBehavior()]] 方法用行为相关联的名字实现：

```
$component->detachBehavior('myBehavior1');
```

也可以移除**全部**行为：

```
$component->detachBehaviors();
```

使用 `TimestampBehavior`

最后以 [[yii\behaviors\TimestampBehavior]] 的讲解来结尾，这个行为支持在 [[yii\db\ActiveRecord|Active Record]] 存储时自动更新它的时间戳属性。

首先，附加这个行为到计划使用该行为的 [[yii\db\ActiveRecord|Active Record]] 类：

```
namespace app\models\User;

use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::className(),
```

```
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at'
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'
                ],
            ],
        ];
    }
}
```

以上指定的行为数组：

- 当记录插入时，行为将当前时间戳赋值给 `created_at` 和 `updated_at` 属性；
- 当记录更新时，行为将当前时间戳赋值给 `updated_at` 属性。

保存 `User` 对象，将会发现它的 `created_at` 和 `updated_at` 属性自动填充了当前时间戳：

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

[[yii\behaviors\TimestampBehavior|TimestampBehavior]] 行为还提供了一个有用的方法 [[yii\behaviors\TimestampBehavior::touch()|touch()]]，这个方法能将当前时间戳赋值给指定属性并保存到数据库：

```
$user->touch('login_time');
```

与 PHP traits 的比较

尽管行为在 "注入" 属性和方法到主类方面类似于 traits ，它们在很多方面却不相同。如上所述，它们各有利弊。它们更像是互补的而不是相互替代。

行为的优势

行为类像普通类支持继承。另一方面，traits 可以视为 PHP 语言支持的复制粘贴功能，它不支持继承。

行为无须修改组件类就可动态附加到组件或移除。要使用 traits，必须修改使用它的

类。

行为是可配置的而 traits 不能。

行为以响应事件来自定义组件的代码执行。

当不同行为附加到同一组件产生命名冲突时，这个冲突通过先附加行为的优先权自动解决。而由不同 traits 引发的命名冲突需要通过手工重命名冲突属性或方法来解决。

traits 的优势

traits 比起行为更高效，因为行为是对象，消耗时间和内存。

IDE 对 traits 更友好，因为它们是语言结构。

数据库访问 (DAO)

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.

Yii 默认支持以下数据库 (DBMS):

- MySQL
- MariaDB
- SQLite
- PostgreSQL
- CUBRID: 版本 >= 9.3 . (由于PHP PDO 扩展的一个bug 引用值会无效,所以你需要在 CUBRID的客户端和服务端都使用 9.3)
- Oracle
- MSSQL: 版本>=2005.

##配置

开始使用数据库首先需要配置数据库连接组件，通过添加 db 组件到应用配置实现（"基础的" Web 应用是 config/web.php），DSN(Data Source Name)是数据源名称，用于指定数据库信息.如下所示：

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase', // MySQ
            //'dsn' => 'sqlite:/path/to/database/file', // SQLite
            //'dsn' => 'pgsql:host=localhost;port=5432;dbname=mydataba
            //'dsn' => 'cubrid:dbname=demodb;host=localhost;port=33000
            //'dsn' => 'sqlsrv:Server=localhost;Database=mydatabase',
            //'dsn' => 'dblib:host=localhost;dbname=mydatabase', // MS
            //'dsn' => 'mssql:host=localhost;dbname=mydatabase', // MS
            //'dsn' => 'oci:dbname=//localhost:1521/mydatabase', // Or
            'username' => 'root', //数据库用户名
```

```
            'password' => '', //数据库密码
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

请参考PHP manual获取更多有关 DSN 格式信息。 配置连接组件后可以使用以下语法访问：

```
$connection = \Yii::$app->db;
```

请参考`[[yii\db\Connection]]`获取可配置的属性列表。 如果你想通过ODBC连接数据库，则需要配置[[yii\db\Connection::driverName]] 属性，例如：

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

注意：如果需要同时使用多个数据库可以定义多个连接组件：

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
        'secondDb' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'sqlite:/path/to/database/file',
        ],
    ],
    // ...
];
```

在代码中通过以下方式使用:

```
$primaryConnection = \Yii::$app->db;
$secondaryConnection = \Yii::$app->secondDb;
```

如果不想定义数据库连接为全局应用组件，可以在代码中直接初始化使用：

```
$connection = new \yii\db\Connection([
    'dsn' => $dsn,
     'username' => $username,
     'password' => $password,
]);
$connection->open();
```

> 小提示：如果在创建了连接后需要执行额外的 SQL 查询，可以添加以下代码到应用配置文件：

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            // ...
            'on afterOpen' => function($event) {
                $event->sender->createCommand("SET time_zone = 'UTC'")
            }
        ],
    ],
    // ...
];
```

##SQL 基础查询

一旦有了连接实例就可以通过[[yii\db\Command]]执行 SQL 查询。

###SELECT 查询 查询返回多行：

```
$command = $connection->createCommand('SELECT * FROM post');
```

```
$posts = $command->queryAll();
```

返回单行：

```
$command = $connection->createCommand('SELECT * FROM post WHERE id=1')
$post = $command->queryOne();
```

查询多行单值：

```
$command = $connection->createCommand('SELECT title FROM post');
$titles = $command->queryColumn();
```

查询标量值/计算值：

```
$command = $connection->createCommand('SELECT COUNT(*) FROM post');
$postCount = $command->queryScalar();
```

###UPDATE, INSERT, DELETE 更新、插入和删除等

如果执行 SQL 不返回任何数据可使用命令中的 execute 方法：

```
$command = $connection->createCommand('UPDATE post SET status=1 WHERE
$command->execute();
```

你可以使用 `insert`，`update`，`delete` 方法，这些方法会根据参数生成合适的SQL并执行.

```
// INSERT
$connection->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// INSERT 一次插入多行
$connection->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
```

```
])->execute();

// UPDATE
$connection->createCommand()->update('user', ['status' => 1], 'age > 3

// DELETE
$connection->createCommand()->delete('user', 'status = 0')->execute();
```

###引用的表名和列名

大多数时间都使用以下语法来安全地引用表名和列名：

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

以上代码 `[[$column]]` 会转变为引用恰当的列名，而 `{{table}}` 就转变为引用恰当的表名。 表名有个特殊的变量 {{%Y}}，如果设置了表前缀使用该变体可以自动在表名前添加前缀：

```
$sql = "SELECT COUNT([[$column]]) FROM {{%$table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

如果在配置文件如下设置了表前缀，以上代码将在 tbl_table 这个表查询结果：

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

手工引用表名和列名的另一个选择是使用[[yii\db\Connection::quoteTableName()]]和 [[yii\db\Connection::quoteColumnName()]]：

```
$column = $connection->quoteColumnName($column);
```

```
$table = $connection->quoteTableName($table);
$sql = "SELECT COUNT($column) FROM $table";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

###预处理语句

为安全传递查询参数可以使用预处理语句,首先应当使用 `:placeholder` 占位，再将变量绑定到对应占位符：

```
$command = $connection->createCommand('SELECT * FROM post WHERE id=:id
$command->bindValue(':id', $_GET['id']);
$post = $command->query();
```

另一种用法是准备一次预处理语句而执行多次查询：

```
$command = $connection->createCommand('DELETE FROM post WHERE id=:id')
$command->bindParam(':id', $id);

$id = 1;
$command->execute();

$id = 2;
$command->execute();
```

> 提示，在执行前绑定变量，然后在每个执行中改变变量的值（一般用在循环中）比较高效.

##事务

当你需要顺序执行多个相关的的 `query` 时，你可以把他们封装到一个事务中去保护数据一致性.Yii提供了一个简单的接口来实现事务操作. 如下执行 SQL 事务查询语句：

```
$transaction = $connection->beginTransaction();
try {
    $connection->createCommand($sql1)->execute();
     $connection->createCommand($sql2)->execute();
    // ... 执行其他 SQL 语句 ...
    $transaction->commit();
} catch(Exception $e) {
    $transaction->rollBack();
```

```
}
```

我们通过[[yii\db\Connection::beginTransaction()|beginTransaction()]]开始一个事务，通过 `try catch` 捕获异常.当执行成功，通过 [[yii\db\Transaction::commit()|commit()]]提交事务并结束，当发生异常失败通过 [[yii\db\Transaction::rollBack()|rollBack()]]进行事务回滚.

如需要也可以嵌套多个事务：

```
// 外部事务
$transaction1 = $connection->beginTransaction();
try {
    $connection->createCommand($sql1)->execute();

    // 内部事务
    $transaction2 = $connection->beginTransaction();
    try {
        $connection->createCommand($sql2)->execute();
        $transaction2->commit();
    } catch (Exception $e) {
        $transaction2->rollBack();
    }

    $transaction1->commit();
} catch (Exception $e) {
    $transaction1->rollBack();
}
```

> 注意你使用的数据库必须支持 `Savepoints` 才能正确地执行，以上代码在所有关系数据中都可以执行，但是只有支持 `Savepoints` 才能保证安全性。

Yii 也支持为事务设置隔离级别 `isolation levels`，当执行事务时会使用数据库默认的隔离级别，你也可以为事物指定隔离级别. Yii 提供了以下常量作为常用的隔离级别

- [[\yii\db\Transaction::READ_UNCOMMITTED]] - 允许读取改变了的还未提交的数据,可能导致脏读、不可重复读和幻读
- [[\yii\db\Transaction::READ_COMMITTED]] - 允许并发事务提交之后读取，可以避免脏读，可能导致重复读和幻读。
- [[\yii\db\Transaction::REPEATABLE_READ]] - 对相同字段的多次读取结果一致，可导致幻读。

- [[\yii\db\Transaction::SERIALIZABLE]] - 完全服从ACID的原则，确保不发生脏读、不可重复读和幻读。

你可以使用以上常量或者使用一个string字符串命令，在对应数据库中执行该命令用以设置隔离级别，比如对于`postgres`有效的命令为`SERIALIZABLE READ ONLY DEFERRABLE`.

> Note: 某些数据库只能针对连接来设置事务隔离级别，所以你必须要为连接明确制定隔离级别.目前受影响的数据库: `MSSQL SQLite`

> Note: SQLite 只支持两种事务隔离级别，所以你只能设置`READ UNCOMMITTED`和`SERIALIZABLE`.使用其他隔离级别会抛出异常.

> Note: PostgreSQL 不允许在事务开始前设置隔离级别，所以你不能在事务开始时指定隔离级别.你可以在事务开始之后调用[[yii\db\Transaction::setIsolationLevel()]] 来设置.

关于隔离级别[isolation levels]:
http://en.wikipedia.org/wiki/Isolation_(database_systems)#Isolation_levels

##数据库复制和读写分离

很多数据库支持数据库复制 database replication来提高可用性和响应速度. 在数据库复制中，数据总是从主服务器 到 从服务器. 所有的插入和更新等写操作在主服务器执行，而读操作在从服务器执行.

通过配置[[yii\db\Connection]]可以实现数据库复制和读写分离.

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

    // 配置主服务器
    'dsn' => 'dsn for master server',
    'username' => 'master',
    'password' => '',

    // 配置从服务器
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
```

```
        ],
    ],

    // 配置从服务器组
    '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'],
    ],
]
```

以上的配置实现了一主多从的结构，从服务器用以执行读查询，主服务器执行写入查询，读写分离的功能由后台代码自动完成.调用者无须关心.例如：

```
// 使用以上配置创建数据库连接对象
$db = Yii::createObject($config);

// 通过从服务器执行查询操作
$rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();

// 通过主服务器执行更新操作
$db->createCommand("UPDATE user SET username='demo' WHERE id=1")->exec
```

> Note: 通过[[yii\db\Command::execute()]] 执行的查询被认为是写操作，所有使用[[yii\db\Command]]来执行的其他查询方法被认为是读操作.你可以通过 `$db->slave` 得到当前正在使用能够的从服务器.

`Connection` 组件支持从服务器的负载均衡和故障转移，当第一次执行读查询时，会随即选择一个从服务器进行连接，如果连接失败则又选择另一个，如果所有从服务器都不可用，则会连接主服务器。你可以配置[[yii\db\Connection::serverStatusCache|server status cache]]来记住那些不能连接的从服务器，使Yii 在一段时间[[yii\db\Connection::serverRetryInterval]].内不会重复尝试连接那些根本不可用的从服务器.

> Note: 在上述配置中，每个从服务器连接超时时间被指定为10s. 如果在10s内不能连接，则被认为该服务器已经挂掉.你也可以自定义超时参数.

你也可以配置多主多从的结构，例如:

```
[
```

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

    // 配置主服务器
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // 配置主服务器组
    'masters' => [
        ['dsn' => 'dsn for master server 1'],
        ['dsn' => 'dsn for master server 2'],
    ],

    // 配置从服务器
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // 配置从服务器组
    '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'],
    ],
]
```

上述配置制定了2个主服务器和4个从服务器. `Connection` 组件也支持主服务器的负载均衡和故障转移，与从服务器不同的是，如果所有主服务器都不可用，则会抛出异常.

> Note: 当你使用[[yii\db\Connection::masters|masters]]来配置一个或多个主服务器时， `Connection` 中关于数据库连接的其他属性（例如： `dsn` , `username` , `password` ）都会被忽略.

事务默认使用主服务器的连接，并且在事务执行中的所有操作都会使用主服务器的

连接，例如:

```
// 在主服务器连接上开始事务
$transaction = $db->beginTransaction();

try {
    // 所有的查询都在主服务器上执行
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAl
    $db->createCommand("UPDATE user SET username='demo' WHERE id=1")->

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

如果你想在从服务器上执行事务操作则必须要明确地指定，比如:

```
$transaction = $db->slave->beginTransaction();
```

有时你想强制使用主服务器来执行读查询，你可以调用`seMaster()`方法.

```
$rows = $db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll
});
```

你也可以设置`$db->enableSlaves`为`false`来使所有查询都在主服务器上执行.

##操作数据库模式

###获得模式信息

你可以通过 [[yii\db\Schema]]实例来获取Schema信息:

```
$schema = $connection->getSchema();
```

该实例包括一系列方法来检索数据库多方面的信息:

```php
$tables = $schema->getTableNames();
```

更多信息请参考[[yii\db\Schema]]

###修改模式

除了基础的 SQL 查询，[[yii\db\Command]]还包括一系列方法来修改数据库模式：

- 创建/重命名/删除/清空表
- 增加/重命名/删除/修改字段
- 增加/删除主键
- 增加/删除外键
- 创建/删除索引

使用示例:

```php
// 创建表
$connection->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

完整参考请查看[[yii\db\Command]].

Active Record

> Note: 该章节还在开发中。

Active Record （活动记录，以下简称AR）提供了一个面向对象的接口， 用以访问数据库中的数据。一个 AR 类关联一张数据表， 每个 AR 对象对应表中的一行，对象的属性（即 AR 的特性Attribute）映射到数据行的对应列。 一条活动记录（AR对象）对应数据表的一行，AR对象的属性则映射该行的相应列。 您可以直接以面向对象的方式来操纵数据表中的数据，妈妈再不用担心我需要写原生 SQL 语句啦。

例如，假定 `Customer` AR 类关联着 `customer` 表，且该类的 `name` 属性代表 `customer` 表的 `name` 列。 你可以写以下代码来哉 `customer` 表里插入一行新的记录:

用 AR 而不是原生的 SQL 语句去执行数据库查询，可以调用直观方法来实现相同目标。如，调用 [[yii\db\ActiveRecord::save()|save()]] 方法将执行插入或更新轮询，将在该 AR 类关联的数据表新建或更新一行数据：

```
$customer = new Customer();
$customer->name = '李狗蛋';
$customer->save();  // 一行新数据插入 customer 表
```

上面的代码和使用下面的原生 SQL 语句是等效的，但显然前者更直观， 更不易出错，并且面对不同的数据库系统（DBMS, Database Management System）时更不容易产生兼容性问题。

```
$db->createCommand('INSERT INTO customer (name) VALUES (:name)', [
    ':name' => '李狗蛋',
])->execute();
```

下面是所有目前被 Yii 的 AR 功能所支持的数据库列表：

- MySQL 4.1 及以上：通过 [[yii\db\ActiveRecord]]
- PostgreSQL 7.3 及以上：通过 [[yii\db\ActiveRecord]]
- SQLite 2 和 3：通过 [[yii\db\ActiveRecord]]
- Microsoft SQL Server 2010 及以上：通过 [[yii\db\ActiveRecord]]

- Oracle: 通过 [[yii\db\ActiveRecord]]
- CUBRID 9.1 及以上：通过 [[yii\db\ActiveRecord]]
- Sphinx：通过 [[yii\sphinx\ActiveRecord]]，需求 `yii2-sphinx` 扩展
- ElasticSearch：通过 [[yii\elasticsearch\ActiveRecord]]，需求 `yii2-elasticsearch` 扩展
- Redis 2.6.12 及以上：通过 [[yii\redis\ActiveRecord]]，需求 `yii2-redis` 扩展
- MongoDB 1.3.0 及以上：通过 [[yii\mongodb\ActiveRecord]]，需求 `yii2-mongodb` 扩展

如你所见，Yii 不仅提供了对关系型数据库的 AR 支持，还提供了 NoSQL 数据库的支持。 在这个教程中，我们会主要描述对关系型数据库的 AR 用法。 然而，绝大多数的内容在 NoSQL 的 AR 里同样适用。

声明 AR 类

要想声明一个 AR 类，你需要扩展 [[yii\db\ActiveRecord]] 基类， 并实现 `tableName` 方法，返回与之相关联的的数据表的名称：

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    /**
     * @return string 返回该AR类关联的数据表名
     */
    public static function tableName()
    {
        return 'customer';
    }
}
```

访问列数据

AR 把相应数据行的每一个字段映射为 AR 对象的一个个特性变量（Attribute） 一个特性就好像一个普通对象的公共属性一样（public property）。 特性变量的名称和对应字段的名称是一样的，且大小姓名。

使用以下语法读取列的值：

```
// "id" 和 "mail" 是 $customer 对象所关联的数据表的对应字段名
$id = $customer->id;
$email = $customer->email;
```

要改变列值，只要给关联属性赋新值并保存对象即可：

```
$customer->email = '哪吒@example.com';
$customer->save();
```

建立数据库连接

AR 用一个 [[yii\db\Connection|DB connection]] 对象与数据库交换数据。 默认的，它使用 `db` 组件作为其连接对象。详见数据库基础章节， 你可以在应用程序配置文件中设置下 `db` 组件，就像这样，

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

如果在你的应用中应用了不止一个数据库，且你需要给你的 AR 类使用不同的数据库链接（DB connection） ，你可以覆盖掉 [[yii\db\ActiveRecord::getDb()|getDb()]] 方法：

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        return \Yii::$app->db2;  // 使用名为 "db2" 的应用组件
```

```
    }
}
```

查询数据

AR 提供了两种方法来构建 DB 查询并向 AR 实例里填充数据：

- [[yii\db\ActiveRecord::find()]]
- [[yii\db\ActiveRecord::findBySql()]]

以上两个方法都会返回 [[yii\db\ActiveQuery]] 实例，该类继承自[[yii\db\Query]]，因此，他们都支持同一套灵活且强大的 DB 查询方法，如 `where()`， `join()`， `orderBy()`，等等。 下面的这些案例展示了一些可能的玩法：

```php
// 取回所有活跃客户(状态为 *active* 的客户)并以他们的 ID 排序:
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// 返回ID为1的客户:
$customer = Customer::find()
    ->where(['id' => 1])
    ->one();

// 取回活跃客户的数量:
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// 以客户ID索引结果集:
$customers = Customer::find()->indexBy('id')->all();
// $customers 数组以 ID 为索引

// 用原生 SQL 语句检索客户:
$sql = 'SELECT * FROM customer';
$customers = Customer::findBySql($sql)->all();
```

> Tip: 在上面的代码中， `Customer::STATUS_ACTIVE` 是一个在 `Customer` 类里定义的常量。（译注：这种常量的值一般都是tinyint）相较于直接在代码中写死字符串或数字，使用一个更有意义的常量名称是一种更好的编程习惯。

有两个快捷方法：`findOne` 和 `findAll()` 用来返回一个或者一组 `ActiveRecord` 实例。前者返回第一个匹配到的实例，后者返回所有。 例如：

```
// 返回 id 为 1 的客户
$customer = Customer::findOne(1);

// 返回 id 为 1 且状态为 *active* 的客户
$customer = Customer::findOne([
    'id' => 1,
    'status' => Customer::STATUS_ACTIVE,
]);

// 返回id为1、2、3的一组客户
$customers = Customer::findAll([1, 2, 3]);

// 返回所有状态为 "deleted" 的客户
$customer = Customer::findAll([
    'status' => Customer::STATUS_DELETED,
]);
```

以数组形式获取数据

有时候，我们需要处理很大量的数据，这时可能需要用一个数组来存储取到的数据， 从而节省内存。你可以用 `asArray()` 函数做到这一点：

```
// 以数组而不是对象形式取回客户信息：
$customers = Customer::find()
    ->asArray()
    ->all();
// $customers 的每个元素都是键值对数组
```

批量获取数据

在 Query Builder（查询构造器） 里，我们已经解释了当需要从数据库中查询大量数据时，你可以用 *batch query（批量查询）*来限制内存的占用。 你可能也想在 AR 里使用相同的技巧，比如这样……

```
// 一次提取 10 个客户信息
foreach (Customer::find()->batch(10) as $customers) {
    // $customers 是 10 个或更少的客户对象的数组
}
```

```php
// 一次提取 10 个客户并一个一个地遍历处理
foreach (Customer::find()->each(10) as $customer) {
    // $customer 是一个 "Customer" 对象
}
// 贪婪加载模式的批处理查询
foreach (Customer::find()->with('orders')->each() as $customer) {
}
```

操作数据

AR 提供以下方法插入、更新和删除与 AR 对象关联的那张表中的某一行：

- [[yii\db\ActiveRecord::save()|save()]]
- [[yii\db\ActiveRecord::insert()|insert()]]
- [[yii\db\ActiveRecord::update()|update()]]
- [[yii\db\ActiveRecord::delete()|delete()]]

AR 同时提供了一下静态方法，可以应用在与某 AR 类所关联的整张表上。 用这些方法的时候千万要小心，因为他们作用于整张表！ 比如， `deleteAll()` 会删除掉表里**所有**的记录。

- [[yii\db\ActiveRecord::updateCounters()|updateCounters()]]
- [[yii\db\ActiveRecord::updateAll()|updateAll()]]
- [[yii\db\ActiveRecord::updateAllCounters()|updateAllCounters()]]
- [[yii\db\ActiveRecord::deleteAll()|deleteAll()]]

下面的这些例子里，详细展现了如何使用这些方法：

```php
// 插入新客户的记录
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();  // 等同于 $customer->insert();

// 更新现有客户记录
$customer = Customer::findOne($id);
$customer->email = 'james@example.com';
$customer->save();  // 等同于 $customer->update();

// 删除已有客户记录
$customer = Customer::findOne($id);
```

```
$customer->delete();

// 删除多个年龄大于20，性别为男（Male）的客户记录
Customer::deleteAll('age > :age AND gender = :gender', [':age' => 20,

// 所有客户的age（年龄）字段加1：
Customer::updateAllCounters(['age' => 1]);
```

> 须知：`save()` 方法会调用 `insert()` 和 `update()` 中的一个， 用哪个取决于当前 AR 对象是不是新对象（在函数内部，他会检查 [[yii\db\ActiveRecord::isNewRecord]] 的值）。 若 AR 对象是由 `new` 操作符初始化出来的， `save()` 方法会在表里插入一条数据； 如果一个 AR 是由 `find()` 方法获取来的， 则 `save()` 会更新表里的对应行记录。

数据输入与有效性验证

由于AR继承自[[yii\base\Model]]，所以它同样也支持Model的数据输入、验证等特性。例如，你可以声明一个rules方法用来覆盖掉[[yii\base\Model::rules()|rules()]]里的；你也可以给AR实例批量赋值；你也可以通过调用[[yii\base\Model::validate()|validate()]]执行数据验证。

当你调用 `save()`、`insert()`、`update()` 这三个方法时，会自动调用[[yii\base\Model::validate()|validate()]]方法。如果验证失败，数据将不会保存进数据库。

下面的例子演示了如何使用AR 获取/验证用户输入的数据并将他们保存进数据库：

```
// 新建一条记录
$model = new Customer;
if ($model->load(Yii::$app->request->post()) && $model->save()) {
    // 获取用户输入的数据，验证并保存
}

// 更新主键为$id的AR
$model = Customer::findOne($id);
if ($model === null) {
    throw new NotFoundHttpException;
}
if ($model->load(Yii::$app->request->post()) && $model->save()) {
    // 获取用户输入的数据，验证并保存
}
```

读取默认值

你的表列也许定义了默认值。有时候，你可能需要在使用web表单的时候给AR预设一些值。如果你需要这样做，可以在显示表单内容前通过调用 `loadDefaultValues()` 方法来实现：

```
$customer = new Customer();
$customer->loadDefaultValues();
// ... 渲染 $customer 的 HTML 表单 ...
```

AR的生命周期

理解AR的生命周期对于你操作数据库非常重要。生命周期通常都会有些典型的事件存在。对于开发AR的behaviors来说非常有用。

当你实例化一个新的AR对象时，我们将获得如下的生命周期：

1. constructor
2. [[yii\db\ActiveRecord::init()|init()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]] 事件

当你通过 [[yii\db\ActiveRecord::find()|find()]] 方法查询数据时，每个AR实例都将有以下生命周期：

1. constructor
2. [[yii\db\ActiveRecord::init()|init()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]] 事件
3. [[yii\db\ActiveRecord::afterFind()|afterFind()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_AFTER_FIND|EVENT_AFTER_FIND]] 事件

当通过 [[yii\db\ActiveRecord::save()|save()]] 方法写入或者更新数据时, 我们将获得如下生命周期：

1. [[yii\db\ActiveRecord::beforeValidate()|beforeValidate()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]]] 事件
2. [[yii\db\ActiveRecord::afterValidate()|afterValidate()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_AFTER_VALIDATE|EVENT_AFTER_VALIDATE]] 事件

3. [[yii\db\ActiveRecord::beforeSave()|beforeSave()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_BEFORE_INSERT|EVENT_BEFORE_INSERT]] 或 [[yii\db\ActiveRecord::EVENT_BEFORE_UPDATE|EVENT_BEFORE_UPDATE]] 事件
4. 执行实际的数据写入或更新
5. [[yii\db\ActiveRecord::afterSave()|afterSave()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] 或 [[yii\db\ActiveRecord::EVENT_AFTER_UPDATE|EVENT_AFTER_UPDATE]] 事件

最后，当调用 [[yii\db\ActiveRecord::delete()|delete()]] 删除数据时, 我们将获得如下生命周期：

1. [[yii\db\ActiveRecord::beforeDelete()|beforeDelete()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_BEFORE_DELETE|EVENT_BEFORE_DELETE]] 事件
2. 执行实际的数据删除
3. [[yii\db\ActiveRecord::afterDelete()|afterDelete()]]: 会触发一个 [[yii\db\ActiveRecord::EVENT_AFTER_DELETE|EVENT_AFTER_DELETE]] 事件

查询关联的数据

使用 AR 方法也可以查询数据表的关联数据（如，选出表A的数据可以拉出表B的关联数据）。 有了 AR， 返回的关联数据连接就像连接关联主表的 AR 对象的属性一样。

建立关联关系后，通过 `$customer->orders` 可以获取 一个 `Order` 对象的数组，该数组代表当前客户对象的订单集。

定义关联关系使用一个可以返回 [[yii\db\ActiveQuery]] 对象的 getter 方法，[[yii\db\ActiveQuery]]对象有关联上下文的相关信息，因此可以只查询关联数据。

例如：

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        // 客户和订单通过 Order.customer_id -> id 关联建立一对多关系
        return $this->hasMany(Order::className(), ['customer_id' => 'i
    }
```

```
}

class Order extends \yii\db\ActiveRecord
{
    // 订单和客户通过 Customer.id -> customer_id 关联建立一对一关系
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer
    }
}
```

以上使用了 [[yii\db\ActiveRecord::hasMany()]] 和 [[yii\db\ActiveRecord::hasOne()]] 方法。 以上两例分别是关联数据多对一关系和一对一关系的建模范例。 如，一个客户有很多订单，一个订单只归属一个客户。 两个方法都有两个参数并返回 [[yii\db\ActiveQuery]] 对象。

- `$class`： 关联模型类名，它必须是一个完全合格的类名。
- `$link`：两个表的关联列，应为键值对数组的形式。 数组的键是 `$class` 关联表的列名， 而数组值是关联类 $class 的列名。 基于表外键定义关联关系是最佳方法。

建立关联关系后，获取关联数据和获取组件属性一样简单， 执行以下相应getter方法即可：

```
// 取得客户的订单
$customer = Customer::findOne(1);
$orders = $customer->orders; // $orders 是 Order 对象数组
```

以上代码实际执行了以下两条 SQL 语句：

```
SELECT * FROM customer WHERE id=1;
SELECT * FROM order WHERE customer_id=1;
```

> Tip: 再次用表达式 `$customer->orders` 将不会执行第二次 SQL 查询， SQL 查询只在该表达式第一次使用时执行。 数据库访问只返回缓存在内部前一次取回的结果集，如果你想查询新的 关联数据，先要注销现有结果集： `unset($customer->orders);` 。

有时候需要在关联查询中传递参数，如不需要返回客户全部订单， 只需要返回购买金额超过设定值的大订单， 通过以下getter方法声明一个关联数据 `bigOrders` ：

```
class Customer extends \yii\db\ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'i
            ->where('subtotal > :threshold', [':threshold' => $thresho
            ->orderBy('id');
    }
}
```

`hasMany()` 返回 [[yii\db\ActiveQuery]] 对象，该对象允许你通过 [[yii\db\ActiveQuery]] 方法定制查询。

如上声明后，执行 `$customer->bigOrders` 就返回 总额大于100的订单。使用以下代码更改设定值：

```
$orders = $customer->getBigOrders(200)->all();
```

> Note: 关联查询返回的是 [[yii\db\ActiveQuery]] 的实例，如果像特性（如类属性）那样连接关联数据， 返回的结果是关联查询的结果，即 [[yii\db\ActiveRecord]] 的实例， 或者是数组，或者是 null ，取决于关联关系的多样性。如， `$customer->getOrders()` 返回 `ActiveQuery` 实例，而 `$customer->orders` 返回 `Order` 对象数组 （如果查询结果为空则返回空数组）。

中间关联表

有时，两个表通过中间表关联，定义这样的关联关系， 可以通过调用 [[yii\db\ActiveQuery::via()|via()]] 方法或 [[yii\db\ActiveQuery::viaTable()|viaTable()]] 方法来定制 [[yii\db\ActiveQuery]] 对象。

举例而言，如果 `order` 表和 `item` 表通过中间表 `order_item` 关联起来， 可以在 `Order` 类声明 `items` 关联关系取代中间表：

```
class Order extends \yii\db\ActiveRecord
{
    public function getItems()
    {
```

```
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

两个方法是相似的，除了 [[yii\db\ActiveQuery::via()|via()]] 方法的第一个参数是使用 AR 类中定义的关联名。 以上方法取代了中间表，等价于：

```
class Order extends \yii\db\ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::className(), ['order_id' => '
    }

    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

延迟加载和即时加载（又称惰性加载与贪婪加载）

如前所述，当你第一次连接关联对象时， AR 将执行一个数据库查询 来检索请求数据并填充到关联对象的相应属性。 如果再次连接相同的关联对象，不再执行任何查询语句，这种数据库查询的执行方法称为“延迟加载”。如：

```
// SQL executed: SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// SQL executed: SELECT * FROM order WHERE customer_id=1
$orders = $customer->orders;
// 没有 SQL 语句被执行
$orders2 = $customer->orders; //取回上次查询的缓存数据
```

延迟加载非常实用，但是，在以下场景中使用延迟加载会遭遇性能问题：

```
// SQL executed: SELECT * FROM customer LIMIT 100
$customers = Customer::find()->limit(100)->all();
```

```
foreach ($customers as $customer) {
    // SQL executed: SELECT * FROM order WHERE customer_id=...
    $orders = $customer->orders;
    // ...处理 $orders...
}
```

假设数据库查出的客户超过100个，以上代码将执行多少条 SQL 语句？ 101 条！第一条 SQL 查询语句取回100个客户，然后， 每个客户要执行一条 SQL 查询语句以取回该客户的所有订单。

为解决以上性能问题，可以通过调用 [[yii\db\ActiveQuery::with()]] 方法使用即时加载解决。

```
// SQL executed: SELECT * FROM customer LIMIT 100;
//               SELECT * FROM orders WHERE customer_id IN (1,2,...)
$customers = Customer::find()->limit(100)
    ->with('orders')->all();

foreach ($customers as $customer) {
    // 没有 SQL 语句被执行
    $orders = $customer->orders;
    // ...处理 $orders...
}
```

如你所见，同样的任务只需要两个 SQL 语句。

> 须知：通常，即时加载 N 个关联关系而通过 via() 或者 viaTable() 定义了 M 个关联关系， 将有 1+M+N 条 SQL 查询语句被执行：一个查询取回主表行数，一个查询给每一个 (M) 中间表，一个查询给每个 (N) 关联表。 注意:当用即时加载定制 select() 时，确保连接 到关联模型的列都被包括了，否则，关联模型不会载入。如：

```
$orders = Order::find()->select(['id', 'amount'])->with('customer')->a
// $orders[0]->customer 总是空的，使用以下代码解决这个问题：
$orders = Order::find()->select(['id', 'amount', 'customer_id'])->with
```

有时候，你想自由的自定义关联查询，延迟加载和即时加载都可以实现，如：

```
$customer = Customer::findOne(1);
// 延迟加载: SELECT * FROM order WHERE customer_id=1 AND subtotal>100
```

```php
$orders = $customer->getOrders()->where('subtotal>100')->all();

// 即时加载: SELECT * FROM customer LIMIT 100
//          SELECT * FROM order WHERE customer_id IN (1,2,...) AND sub
$customers = Customer::find()->limit(100)->with([
    'orders' => function($query) {
        $query->andWhere('subtotal>100');
    },
])->all();
```

逆关系

关联关系通常成对定义，如：Customer 可以有个名为 orders 关联项， 而 Order 也有个名为customer 的关联项：

```php
class Customer extends ActiveRecord
{
    ....
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'i
    }
}

class Order extends ActiveRecord
{
    ....
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_
    }
}
```

如果我们执行以下查询，可以发现订单的 customer 和 找到这些订单的客户对象并不是同一个。连接 customer->orders 将触发一条 SQL 语句 而连接一个订单的 customer 将触发另一条 SQL 语句。

```php
// SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// 输出 "不相同"
// SELECT * FROM order WHERE customer_id=1
// SELECT * FROM customer WHERE id=1
```

```php
if ($customer->orders[0]->customer === $customer) {
    echo '相同';
} else {
    echo '不相同';
}
```

为避免多余执行的后一条语句，我们可以为 customer或 orders 关联关系定义相反的关联关系，通过调用 [[yii\db\ActiveQuery::inverseOf()|inverseOf()]] 方法可以实现。

```php
class Customer extends ActiveRecord
{
    ....
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'i
    }
}
```

现在我们同样执行上面的查询，我们将得到：

```php
// SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// 输出相同
// SELECT * FROM order WHERE customer_id=1
if ($customer->orders[0]->customer === $customer) {
    echo '相同';
} else {
    echo '不相同';
}
```

以上我们展示了如何在延迟加载中使用相对关联关系， 相对关系也可以用在即时加载中：

```php
// SELECT * FROM customer
// SELECT * FROM order WHERE customer_id IN (1, 2, ...)
$customers = Customer::find()->with('orders')->all();
// 输出相同
if ($customers[0]->orders[0]->customer === $customers[0]) {
    echo '相同';
} else {
```

```
    echo '不相同';
}
```

> Note: 相对关系不能在包含中间表的关联关系中定义。 即是，如果你的关系是通过[[yii\db\ActiveQuery::via()|via()]] 或 [[yii\db\ActiveQuery::viaTable()|viaTable()]]方法定义的， 就不能调用 [[yii\db\ActiveQuery::inverseOf()]]方法了。

JOIN 类型关联查询

使用关系数据库时，普遍要做的是连接多个表并明确地运用各种 JOIN 查询。 JOIN SQL语句的查询条件和参数，使用 [[yii\db\ActiveQuery::joinWith()]] 可以重用已定义关系并调用 而不是使用 [[yii\db\ActiveQuery::join()]] 来实现目标。

```
// 查找所有订单并以客户 ID 和订单 ID 排序，并贪婪加载 "customer" 表
$orders = Order::find()->joinWith('customer')->orderBy('customer.id, o
// 查找包括书籍的所有订单，并以 `INNER JOIN` 的连接方式即时加载 "books" 表
$orders = Order::find()->innerJoinWith('books')->all();
```

以上，方法 [[yii\db\ActiveQuery::innerJoinWith()|innerJoinWith()]] 是访问 `INNER JOIN` 类型的 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 的快捷方式。

可以连接一个或多个关联关系，可以自由使用查询条件到关联查询， 也可以嵌套连接关联查询。如：

```
// 连接多重关系
// 找出24小时内注册客户包含书籍的订单
$orders = Order::find()->innerJoinWith([
    'books',
    'customer' => function ($query) {
        $query->where('customer.created_at > ' . (time() - 24 * 3600))
    }
])->all();
// 连接嵌套关系：连接 books 表及其 author 列
$orders = Order::find()->joinWith('books.author')->all();
```

代码背后， Yii 先执行一条 JOIN SQL 语句把满足 JOIN SQL 语句查询条件的主要模型查出， 然后为每个关系执行一条查询语句， bing填充相应的关联记录。

[[yii\db\ActiveQuery::joinWith()|joinWith()]] 和 [[yii\db\ActiveQuery::with()|with()]] 的区别是 前者连接主模型类和关联模型类的数据表来检索主模型， 而后者只查询和检索主模型类。 检索主模型

由于这个区别，你可以应用只针对一条 JOIN SQL 语句起效的查询条件。 如，通过关联模型的查询条件过滤主模型，如前例， 可以使用关联表的列来挑选主模型数据，

当使用 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 方法时可以响应没有歧义的列名。 In the above examples, we use `item.id` and `order.id` to disambiguate the `id` column references 因为订单表和项目表都包括 `id` 列。

当连接关联关系时，关联关系默认使用即时加载。你可以 通过传参数 `$eagerLoading` 来决定在指定关联查询中是否使用即时加载。

默认 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 使用左连接来连接关联表。 你也可以传 `$joinType` 参数来定制连接类型。 你也可以使用 [[yii\db\ActiveQuery::innerJoinWith()|innerJoinWith()]]。

以下是 `INNER JOIN` 的简短例子：

```
// 查找包括书籍的所有订单，但 "books" 表不使用即时加载
$orders = Order::find()->innerJoinWith('books', false)->all();
// 等价于：
$orders = Order::find()->joinWith('books', false, 'INNER JOIN')->all()
```

有时连接两个表时，需要在关联查询的 ON 部分指定额外条件。 这可以通过调用 [[yii\db\ActiveQuery::onCondition()]] 方法实现：

```
class User extends ActiveRecord
{
    public function getBooks()
    {
        return $this->hasMany(Item::className(), ['owner_id' => 'id'])
    }
}
```

在上面， [[yii\db\ActiveRecord::hasMany()|hasMany()]] 方法回传了一个 [[yii\db\ActiveQuery]] 对象， 当你用 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 执行一条查询时，取决于正被调用的是哪个

[[yii\db\ActiveQuery::onCondition()|onCondition()]]， 返回 `category_id` 为 1 的 items

当你用 [[yii\db\ActiveQuery::joinWith()|joinWith()]] 进行一次查询时，“on-condition”条件会被放置在相应查询语句的 ON 部分， 如：

```
// SELECT user.* FROM user LEFT JOIN item ON item.owner_id=user.id AND
// SELECT * FROM item WHERE owner_id IN (...) AND category_id=1
$users = User::find()->joinWith('books')->all();
```

注意：如果通过 [[yii\db\ActiveQuery::with()]] 进行贪婪加载或使用惰性加载的话，则 on 条件会被放置在对应 SQL语句的 `WHERE` 部分。 因为，此时此处并没有发生 JOIN 查询。比如：

```
// SELECT * FROM user WHERE id=10
$user = User::findOne(10);
// SELECT * FROM item WHERE owner_id=10 AND category_id=1
$books = $user->books;
```

关联表操作

AR 提供了下面两个方法用来建立和解除两个关联对象之间的关系：

- [[yii\db\ActiveRecord::link()|link()]]
- [[yii\db\ActiveRecord::unlink()|unlink()]]

例如，给定一个customer和order对象，我们可以通过下面的代码使得customer对象拥有order对象：

```
$customer = Customer::findOne(1);
$order = new Order();
$order->subtotal = 100;
$customer->link('orders', $order);
```

[[yii\db\ActiveRecord::link()|link()]] 调用上述将设置 customer_id 的顺序是 $customer 的主键值，然后调用 [[yii\db\ActiveRecord::save()|save()]] 要将顺序保存到数据库中。

作用域

当你调用[[yii\db\ActiveRecord::find()|find()]] 或
[[yii\db\ActiveRecord::findBySql()|findBySql()]]方法时，将会返回一个
[[yii\db\ActiveQuery|ActiveQuery]]实例。之后，你可以调用其他查询方法，如
[[yii\db\ActiveQuery::where()|where()]],
[[yii\db\ActiveQuery::orderBy()|orderBy()]], 进一步的指定查询条件。

有时候你可能需要在不同的地方使用相同的查询方法。如果出现这种情况，你应该考虑定义所谓的作用域。作用域是本质上要求一组的查询方法来修改查询对象的自定义查询类中定义的方法。 之后你就可以像使用普通方法一样使用作用域。

只需两步即可定义一个作用域。首先给你的model创建一个自定义的查询类，在此类中定义的所需的范围方法。例如，给Comment模型创建一个 CommentQuery 类，然后在CommentQuery类中定义一个active()的方法为作用域，像下面的代码：

```
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    public function active($state = true)
    {
        $this->andWhere(['active' => $state]);
        return $this;
    }
}
```

重点:

1. 类必须继承 yii\db\ActiveQuery (或者是其他的 ActiveQuery ， 比如 yii\mongodb\ActiveQuery)。
2. 必须是一个public类型的方法且必须返回 $this 实现链式操作。可以传入参数。
3. 检查 [[yii\db\ActiveQuery]] 对于修改查询条件是非常有用的方法。

其次，覆盖[[yii\db\ActiveRecord::find()]] 方法使其返回自定义的查询对象而不是常规的[[yii\db\ActiveQuery|ActiveQuery]]。对于上述例子，你需要编写如下代码：

```
namespace app\models;
```

```php
use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    /**
     * @inheritdoc
     * @return CommentQuery
     */
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

就这样，现在你可以使用自定义的作用域方法了：

```php
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

你也能在定义的关联里使用作用域方法，比如：

```php
class Post extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::className(), ['post_id' => 'id'

    }
}
```

或者在执行关联查询的时候使用（on-the-fly 是啥？）：

```php
$posts = Post::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

默认作用域

如果你之前用过 Yii 1.1 就应该知道默认作用域的概念。一个默认的作用域可以作用于所有查询。你可以很容易的通过重写[[yii\db\ActiveRecord::find()]]方法来定义一个默认作用域，例如：

```
public static function find()
{
    return parent::find()->where(['deleted' => false]);
}
```

注意，你之后所有的查询都不能用 [[yii\db\ActiveQuery::where()|where()]]，但是可以用 [[yii\db\ActiveQuery::andWhere()|andWhere()]] 和 [[yii\db\ActiveQuery::orWhere()|orWhere()]]，他们不会覆盖掉默认作用域。（译注：如果你要使用默认作用域，就不能在 xxx::find()后使用where()方法，你必须使用andXXX()或者orXXX()系的方法，否则默认作用域不会起效果，至于原因，打开where()方法的代码一看便知）

事务操作

当执行几个相关联的数据库操作的时候

TODO: FIXME: WIP, TBD, https://github.com/yiisoft/yii2/issues/226

, [[yii\db\ActiveRecord::afterSave()|afterSave()]], [[yii\db\ActiveRecord::beforeDelete()|beforeDelete()]] and/or [[yii\db\ActiveRecord::afterDelete()|afterDelete()]] 生命周期周期方法(life cycle methods 我觉得这句翻译成“模板方法”会不会更好点？)。开发者可以通过重写[[yii\db\ActiveRecord::save()|save()]]方法然后在控制器里使用事务操作，严格地说是似乎不是一个好的做法 （召回"瘦控制器 / 肥模型"基本规则）。

这些方法在这里(如果你不明白自己实际在干什么，请不要使用他们)，Models：

```
class Feature extends \yii\db\ActiveRecord
{
    // ...

    public function getProduct()
    {
        return $this->hasOne(Product::className(), ['id' => 'product_i
    }
}
```

```
class Product extends \yii\db\ActiveRecord
{
    // ...

    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['product_id' => '
    }
}
```

重写 [[yii\db\ActiveRecord::save()|save()]] 方法：

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

(译注：我觉得上面应该是原手册里的bug)

在控制器层使用事务：

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

作为这些脆弱方法的替代，你应该使用原子操作方案特性。

```
class Feature extends \yii\db\ActiveRecord
{
    // ...

    public function getProduct()
    {
        return $this->hasOne(Product::className(), ['product_id' => 'i
```

```php
    }

    public function scenarios()
    {
        return [
            'userCreates' => [
                'attributes' => ['name', 'value'],
                'atomic' => [self::OP_INSERT],
            ],
        ];
    }
}

class Product extends \yii\db\ActiveRecord
{
    // ...

    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['id' => 'product_
    }

    public function scenarios()
    {
        return [
            'userCreates' => [
                'attributes' => ['title', 'price'],
                'atomic' => [self::OP_INSERT],
            ],
        ];
    }

    public function afterValidate()
    {
        parent::afterValidate();
        // FIXME: TODO: WIP, TBD
    }

    public function afterSave($insert)
    {
        parent::afterSave($insert);
        if ($this->getScenario() === 'userCreates') {
            // FIXME: TODO: WIP, TBD
        }
    }
}
```

Controller里的代码将变得很简洁：

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

控制器非常简洁：

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

乐观锁（Optimistic Locks）

TODO

被污染属性

当你调用[[yii\db\ActiveRecord::save()|save()]]用于保存活动记录(Active Record)实例时,只有被污染的属性才会被保存。一个属性是否认定为被污染取决于它的值自从最后一次从数据库加载或者最近一次保存到数据库后到现在是否被修改过。注意:无论活动记录(Active Record)是否有被污染属性，数据验证始终会执行。

活动记录(Active Record)会自动维护一个污染数据列表。它的工作方式是通过维护一个较旧属性值版本，并且将它们与最新的进行比较。你可以通过调用[[yii\db\ActiveRecord::getDirtyAttributes()]]来获取当前的污染属性。你也可以调用[[yii\db\ActiveRecord::markAttributeDirty()]]来显示的标记一个属性为污染属性。

如果你对最近一次修改前的属性值感兴趣，你可以调用[[yii\db\ActiveRecord::getOldAttributes()|getOldAttributes()]] 或[[yii\db\ActiveRecord::getOldAttribute()|getOldAttribute()]]。

另见

- 模型（Model）
- [[yii\db\ActiveRecord]]

HTTP 缓存

除了前面章节讲到的服务器端缓存外， Web 应用还可以利用客户端缓存去节省相同页面内容的生成和传输时间。

通过配置 [[yii\filters\HttpCache]] 过滤器，控制器操作渲染的内容就能缓存在客户端。[[yii\filters\HttpCache|HttpCache]] 过滤器仅对 `GET` 和 `HEAD` 请求生效，它能为这些请求设置三种与缓存有关的 HTTP 头。

- [[yii\filters\HttpCache::lastModified|Last-Modified]]
- [[yii\filters\HttpCache::etagSeed|Etag]]
- [[yii\filters\HttpCache::cacheControlHeader|Cache-Control]]

`Last-Modified` 头

`Last-Modified` 头使用时间戳标明页面自上次客户端缓存后是否被修改过。

通过配置 [[yii\filters\HttpCache::lastModified]] 属性向客户端发送 `Last-Modified` 头。该属性的值应该为 PHP callable 类型，返回的是页面修改时的 Unix 时间戳。该 callable 的参数和返回值应该如下：

```
/**
 * @param Action $action 当前处理的操作对象
 * @param array $params “params” 属性的值
 * @return int 页面修改时的 Unix 时间戳
 */
function ($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。

`ETag` 头

“Entity Tag”（实体标签，简称 ETag）使用一个哈希值表示页面内容。如果页面被修改过，哈希值也会随之改变。通过对比客户端的哈希值和服务器端生成的哈希值，浏览器就能判断页面是否被修改过，进而决定是否应该重新传输内容。

通过配置 [[yii\filters\HttpCache::etagSeed]] 属性向客户端发送 `ETag` 头。该属性的值应该为 PHP callable 类型，返回的是一段种子字符用来生成 ETag 哈希值。该 callable 的参数和返回值应该如下：

```
/**
 * @param Action $action 当前处理的操作对象
 * @param array $params “params” 属性的值
 * @return string 一段种子字符用来生成 ETag 哈希值
 */
function ($action, $params)
```

以下是使用 `ETag` 头的示例：

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['view'],
            'etagSeed' => function ($action, $params) {
                $post = $this->findModel(\Yii::$app->request->get('id'
```

```
                return serialize([$post->title, $post->content]);
            },
        ],
    ];
}
```

上述代码表明 HTTP 缓存只在 `view` 操作时启用。它会基于用户请求的标题和内容生成一个 `ETag` HTTP 头。当浏览器第一次访问 `view` 页时，服务器将会生成页面并发送至客户端浏览器。之后客户端浏览器标题和内容没被修改在期间访问该页，服务器将不会重新生成页面，浏览器会使用之前客户端缓存下来的内容。因此服务端渲染和内容传输都将省去。

ETag 相比 `Last-Modified` 能实现更复杂和更精确的缓存策略。例如，当站点切换到另一个主题时可以使 ETag 失效。

复杂的 Etag 生成种子可能会违背使用 `HttpCache` 的初衷而引起不必要的性能开销，因为响应每一次请求都需要重新计算 Etag。请试着找出一个最简单的表达式去触发 Etag 失效。

> Note: 为了遵循 RFC 7232（HTTP 1.1 协议），如果同时配置了 `ETag` 和 `Last-Modified` 头，`HttpCache` 将会同时发送它们。并且如果客户端同时发送 `If-None-Match` 头和 `If-Modified-Since` 头，则只有前者会被接受。

`Cache-Control` 头

`Cache-Control` 头指定了页面的常规缓存策略。可以通过配置 [[yii\filters\HttpCache::cacheControlHeader]] 属性发送相应的头信息。默认发送以下头：

```
Cache-Control: public, max-age=3600
```

会话缓存限制器

当页面使 session 时，PHP 将会按照 PHP.INI 中所设置的 `session.cache_limiter` 值自动发送一些缓存相关的 HTTP 头。这些 HTTP 头有可能会干扰你原本设置的

`HttpCache` 或让其失效。为了避免此问题，默认情况下 `HttpCache` 禁止自动发送这些头。想改变这一行为，可以配置 [[yii\filters\HttpCache::sessionCacheLimiter]] 属性。该属性接受一个字符串值，包括 `public`， `private`， `private_no_expire`，和 `nocache`。请参考 PHP 手册中的缓存限制器了解这些值的含义。

SEO 影响

搜索引擎趋向于遵循站点的缓存头。因为一些爬虫的抓取频率有限制，启用缓存头可以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）。

片段缓存

片段缓存指的是缓存页面内容中的某个片段。例如，一个页面显示了逐年销售额的摘要表格，可以把表格缓存下来，以消除每次请求都要重新生成表格的耗时。片段缓存是基于数据缓存实现的。

在视图中使用以下结构启用片段缓存：

```
if ($this->beginCache($id)) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

调用 [[yii\base\View::beginCache()|beginCache()]] 和 [[yii\base\View::endCache()|endcache()]] 方法包裹内容生成逻辑。如果缓存中存在该内容，[[yii\base\View::beginCache()|beginCache()]] 方法将渲染内容并返回 false，因此将跳过内容生成逻辑。否则，内容生成逻辑被执行，一直执行到 [[yii\base\View::endCache()|endCache()]] 时，生成的内容将被捕获并存储在缓存中。

和[数据缓存]一样，每个片段缓存也需要全局唯一的 `$id` 标记。

缓存选项

如果要为片段缓存指定额外配置项，请通过向 [[yii\base\View::beginCache()|beginCache()]] 方法第二个参数传递配置数组。在框架内部，该数组将被用来配置一个 [[yii\widget\FragmentCache]] 小部件用以实现片段缓存功能。

过期时间（duration）

或许片段缓存中最常用的一个配置选项就是 [[yii\widgets\FragmentCache::duration|duration]] 了。它指定了内容被缓存的秒数。以下代码缓存内容最多一小时：

```php
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

如果该选项未设置，则默认为 0，永不过期。

依赖

和[数据缓存]一样，片段缓存的内容一样可以设置缓存依赖。例如一段被缓存的文章，是否重新缓存取决于它是否被修改过。

通过设置 [[yii\widgets\FragmentCache::dependency|dependency]] 选项来指定依赖，该选项的值可以是一个 [[yii\caching\Dependency]] 类的派生类，也可以是创建缓存对象的配置数组。以下代码指定了一个片段缓存，它依赖于 `update_at` 字段是否被更改过的。

```php
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

变化

缓存的内容可能需要根据一些参数的更改而变化。例如一个 Web 应用支持多语言，同一段视图代码也许需要生成多个语言的内容。因此可以设置缓存根据应用当前语言而变化。

通过设置 [[yii\widgets\FragmentCache::variations|variations]] 选项来指定变化，该选项的值应该是一个标量，每个标量代表不同的变化系数。例如设置缓存根据当前语言而变化可以用以下代码：

```php
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

开关

有时你可能只想在特定条件下开启片段缓存。例如，一个显示表单的页面，可能只需要在初次请求时缓存表单（通过 GET 请求）。随后请求所显示（通过 POST 请求）的表单不该使用缓存，因为此时表单中可能包含用户输入内容。鉴于此种情况，可以使用 [[yii\widgets\FragmentCache::enabled|enabled]] 选项来指定缓存开关，如下所示：

```php
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... 在此生成内容 ...

    $this->endCache();
}
```

缓存嵌套

片段缓存可以被嵌套使用。一个片段缓存可以被另一个包裹。例如，评论被缓存在里层，同时整个评论的片段又被缓存在外层的文章中。以下代码展示了片段缓存的嵌套使用：

```php
if ($this->beginCache($id1)) {

    // ...在此生成内容...

    if ($this->beginCache($id2, $options2)) {

        // ...在此生成内容...

        $this->endCache();
    }

    // ...在此生成内容...
```

```
    $this->endCache();
}
```

可以为嵌套的缓存设置不同的配置项。例如，内层缓存和外层缓存使用不同的过期时间。甚至当外层缓存的数据过期失效了，内层缓存仍然可能提供有效的片段缓存数据。但是，反之则不然。如果外层片段缓存没有过期而被视为有效，此时即使内层片段缓存已经失效，它也将继续提供同样的缓存副本。因此，你必须谨慎处理缓存嵌套中的过期时间和依赖，否则外层的片段很有可能返回的是不符合你预期的失效数据。

> 译注：外层的失效时间应该短于内层，外层的依赖条件应该低于内层，以确保最小的片段，返回的是最新的数据。

动态内容

使用片段缓存时，可能会遇到一大段较为静态的内容中有少许动态内容的情况。例如，一个显示着菜单栏和当前用户名的页面头部。还有一种可能是缓存的内容可能包含每次请求都需要执行的 PHP 代码（例如注册资源包的代码）。这两个问题都可以使用**动态内容**功能解决。

动态内容的意思是这部分输出的内容不该被缓存，即便是它被包裹在片段缓存中。为了使内容保持动态，每次请求都执行 PHP 代码生成，即使这些代码已经被缓存了。

可以在片段缓存中调用 [[yii\base\View::renderDynamic()]] 去插入动态内容，如下所示：

```
if ($this->beginCache($id1)) {

    // ...在此生成内容...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;

    // ...在此生成内容...

    $this->endCache();
}
```

[[yii\base\View::renderDynamic()|renderDynamic()]] 方法接受一段 PHP 代码作为

参数。代码的返回值被看作是动态内容。这段代码将在每次请求时都执行，无论其外层的片段缓存是否被存储。

数据缓存

数据缓存是指将一些 PHP 变量存储到缓存中，使用时再从缓存中取回。它也是更高级缓存特性的基础，例如查询缓存和内容缓存。

如下代码是一个典型的数据缓存使用模式。其中 `$cache` 指向缓存组件：

```
// 尝试从缓存中取回 $data
$data = $cache->get($key);

if ($data === false) {

    // $data 在缓存中没有找到，则重新计算它的值

    // 将 $data 存放到缓存供下次使用
    $cache->set($key, $data);
}

// 这儿 $data 可以使用了。
```

缓存组件

数据缓存需要**缓存组件**提供支持，它代表各种缓存存储器，例如内存，文件，数据库。

缓存组件通常注册为应用程序组件，这样它们就可以在全局进行配置与访问。如下代码演示了如何配置应用程序组件 `cache` 使用两个 memcached 服务器：

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
```

```
                'weight' => 50,
            ],
        ],
    ],
],
```

然后就可以通过 `Yii::$app->cache` 访问上面的缓存组件了。

由于所有缓存组件都支持同样的一系列 API，并不需要修改使用缓存的业务代码就能直接替换为其他底层缓存组件，只需在应用配置中重新配置一下就可以。例如，你可以将上述配置修改为使用 [[yii\caching\ApcCache|APC cache]]:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> Tip: 你可以注册多个缓存组件，很多依赖缓存的类默认调用名为 `cache` 的组件（例如 [[yii\web\UrlManager]]）。

支持的缓存存储器

Yii 支持一系列缓存存储器，概况如下：

- [[yii\caching\ApcCache]]：使用 PHP APC 扩展。这个选项可以认为是集中式应用程序环境中（例如：单一服务器，没有独立的负载均衡器等）最快的缓存方案。
- [[yii\caching\DbCache]]：使用一个数据库的表存储缓存数据。要使用这个缓存，你必须创建一个与 [[yii\caching\DbCache::cacheTable]] 对应的表。
- [[yii\caching\DummyCache]]: 仅作为一个缓存占位符，不实现任何真正的缓存功能。这个组件的目的是为了简化那些需要查询缓存有效性的代码。例如，在开发中如果服务器没有实际的缓存支持，用它配置一个缓存组件。一个真正的缓存服务启用后，可以再切换为使用相应的缓存组件。两种条件下你都可以使用同样的代码 `Yii::$app->cache->get($key)` 尝试从缓存中取回数据而不用担心 `Yii::$app->cache` 可能是 `null`。
- [[yii\caching\FileCache]]：使用标准文件存储缓存数据。这个特别适用于缓存大块数据，例如一个整页的内容。

- [[yii\caching\MemCache]]：使用 PHP memcache 和 memcached 扩展。这个选项被看作分布式应用环境中（例如：多台服务器，有负载均衡等）最快的缓存方案。
- [[yii\redis\Cache]]：实现了一个基于 Redis 键值对存储器的缓存组件（需要 redis 2.6.12 及以上版本的支持 ）。
- [[yii\caching\WinCache]]：使用 PHP WinCache（另可参考）扩展.
- [[yii\caching\XCache]]：使用 PHP XCache扩展。
- [[yii\caching\ZendDataCache]]：使用 Zend Data Cache 作为底层缓存媒介。

> Tip: 你可以在同一个应用程序中使用不同的缓存存储器。一个常见的策略是使用基于内存的缓存存储器存储小而常用的数据（例如：统计数据），使用基于文件或数据库的缓存存储器存储大而不太常用的数据（例如：网页内容）。

缓存 API

所有缓存组件都有同样的基类 [[yii\caching\Cache]] ，因此都支持如下 API：

- [[yii\caching\Cache::get()|get()]]：通过一个指定的键（key）从缓存中取回一项数据。如果该项数据不存在于缓存中或者已经过期/失效，则返回值 false。
- [[yii\caching\Cache::set()|set()]]：将一项数据指定一个键，存放到缓存中。
- [[yii\caching\Cache::add()|add()]]：如果缓存中未找到该键，则将指定数据存放到缓存中。
- [[yii\caching\Cache::mget()|mget()]]：通过指定的多个键从缓存中取回多项数据。
- [[yii\caching\Cache::mset()|mset()]]：将多项数据存储到缓存中，每项数据对应一个键。
- [[yii\caching\Cache::madd()|madd()]]：将多项数据存储到缓存中，每项数据对应一个键。如果某个键已经存在于缓存中，则该项数据会被跳过。
- [[yii\caching\Cache::exists()|exists()]]：返回一个值，指明某个键是否存在于缓存中。
- [[yii\caching\Cache::delete()|delete()]]：通过一个键，删除缓存中对应的值。
- [[yii\caching\Cache::flush()|flush()]]：删除缓存中的所有数据。

有些缓存存储器如 MemCache，APC 支持以批量模式取回缓存值，这样可以节省取回缓存数据的开支。 [[yii\caching\Cache::mget()|mget()]] 和 [[yii\caching\Cache::madd()|madd()]] API提供对该特性的支持。如果底层缓存存储器不支持该特性，Yii 也会模拟实现。

由于 [[yii\caching\Cache]] 实现了 PHP `ArrayAccess` 接口，缓存组件也可以像数组那样使用，下面是几个例子：

```
$cache['var1'] = $value1;  // 等价于: $cache->set('var1', $value1);
$value2 = $cache['var2'];  // 等价于: $value2 = $cache->get('var2');
```

缓存键

存储在缓存中的每项数据都通过键作唯一识别。当你在缓存中存储一项数据时，必须为它指定一个键，稍后从缓存中取回数据时，也需要提供相应的键。

你可以使用一个字符串或者任意值作为一个缓存键。当键不是一个字符串时，它将会自动被序列化为一个字符串。

定义一个缓存键常见的一个策略就是在一个数组中包含所有的决定性因素。例如，[[yii\db\Schema]] 使用如下键存储一个数据表的结构信息。

```
[
    __CLASS__,              // 结构类名
    $this->db->dsn,         // 数据源名称
    $this->db->username,    // 数据库登录用户名
    $name,                  // 表名
];
```

如你所见，该键包含了可唯一指定一个数据库表所需的所有必要信息。

当同一个缓存存储器被用于多个不同的应用时，应该为每个应用指定一个唯一的缓存键前缀以避免缓存键冲突。可以通过配置 [[yii\caching\Cache::keyPrefix]] 属性实现。例如，在应用配置中可以编写如下代码：

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',       // 唯一键前缀
    ],
],
```

为了确保互通性，此处只能使用字母和数字。

缓存过期

默认情况下，缓存中的数据会永久存留，除非它被某些缓存策略强制移除（例如：缓存空间已满，最老的数据会被移除）。要改变此特性，你可以在调用 [[yii\caching\Cache::set()|set()]] 存储一项数据时提供一个过期时间参数。该参数代表这项数据在缓存中可保持有效多少秒。当你调用 [[yii\caching\Cache::get()|get()]] 取回数据时，如果它已经过了超时时间，该方法将返回 false，表明在缓存中找不到这项数据。例如：

```
// 将数据在缓存中保留 45 秒
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // $data 已过期，或者在缓存中找不到
}
```

缓存依赖

除了超时设置，缓存数据还可能受到**缓存依赖**的影响而失效。例如，[[yii\caching\FileDependency]] 代表对一个文件修改时间的依赖。这个依赖条件发生变化也就意味着相应的文件已经被修改。因此，缓存中任何过期的文件内容都应该被置为失效状态，对 [[yii\caching\Cache::get()|get()]] 的调用都应该返回 false。

缓存依赖用 [[yii\caching\Dependency]] 的派生类所表示。当调用 [[yii\caching\Cache::set()|set()]] 在缓存中存储一项数据时，可以同时传递一个关联的缓存依赖对象。例如：

```
// 创建一个对 example.txt 文件修改时间的缓存依赖
$dependency = new \yii\caching\FileDependency(['fileName' => 'example.

// 缓存数据将在30秒后超时
// 如果 example.txt 被修改，它也可能被更早地置为失效状态。
$cache->set($key, $data, 30, $dependency);

// 缓存会检查数据是否已超时。
// 它还会检查关联的依赖是否已变化。
// 符合任何一个条件时都会返回 false。
$data = $cache->get($key);
```

下面是可用的缓存依赖的概况：

- [[yii\caching\ChainedDependency]]：如果依赖链上任何一个依赖产生变化，则依赖改变。
- [[yii\caching\DbDependency]]：如果指定 SQL 语句的查询结果发生了变化，则依赖改变。
- [[yii\caching\ExpressionDependency]]：如果指定的 PHP 表达式执行结果发生变化，则依赖改变。
- [[yii\caching\FileDependency]]：如果文件的最后修改时间发生变化，则依赖改变。
- [[yii\caching\GroupDependency]]：将一项缓存数据标记到一个组名，你可以通过调用 [[yii\caching\GroupDependency::invalidate()]] 一次性将相同组名的缓存全部置为失效状态。

查询缓存

查询缓存是一个建立在数据缓存之上的特殊缓存特性。它用于缓存数据库查询的结果。

查询缓存需要一个 [[yii\db\Connection|数据库连接]] 和一个有效的 `cache` 应用组件。查询缓存的基本用法如下，假设 `$db` 是一个 [[yii\db\Connection]] 实例：

```
$duration = 60;     // 缓存查询结果60秒
$dependency = ...;  // 可选的缓存依赖

$db->beginCache($duration, $dependency);

// ...这儿执行数据库查询...

$db->endCache();
```

如你所见， `beginCache()` 和 `endCache()` 中间的任何查询结果都会被缓存起来。如果缓存中找到了同样查询的结果，则查询会被跳过，直接从缓存中提取结果。

查询缓存可以用于 ActiveRecord 和 DAO。

> Info: 有些 DBMS （例如：MySQL）也支持数据库服务器端的查询缓存。你可以选择使用任一查询缓存机制。上文所述的查询缓存的好处在于你可以指定更灵活的缓存依赖因此可能更加高效。

配置

查询缓存有两个通过 [[yii\db\Connection]] 设置的配置项：

- [[yii\db\Connection::queryCacheDuration|queryCacheDuration]]: 查询结果在缓存中的有效期，以秒表示。如果在调用 [[yii\db\Connection::beginCache()]] 时传递了一个显式的时值参数，则配置中的有效期时值会被覆盖。
- [[yii\db\Connection::queryCache|queryCache]]: 缓存应用组件的 ID。默认为 `'cache'`。只有在设置了一个有效的缓存应用组件时，查询缓存才会有效。

限制条件

当查询结果中含有资源句柄时，查询缓存无法使用。例如，在有些 DBMS 中使用了 `BLOB` 列的时候，缓存结果会为该数据列返回一个资源句柄。

有些缓存存储器有大小限制。例如，memcache 限制每条数据最大为 1MB。因此，如果查询结果的大小超出了该限制，则会导致缓存失败。

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 支持如下所有缓存机制：

- 数据缓存
- 片段缓存
- 页面缓存
- HTTP 缓存

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。

助手类

> Note: 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');
```

> Note: 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。

核心助手类

Yii 发布版中提供以下核心助手类：

- ArrayHelper
- Console
- FileHelper
- Html
- HtmlPurifier
- Image
- Inflector
- Json
- Markdown
- Security
- StringHelper
- Url
- VarDumper

自定义助手类

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 [[yii\helpers\ArrayHelper::merge()|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/ArrayHelp
```

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。