深入理解Yii2.0

Release 1

Linuor

September 14, 2014

当前版本号：*Ver 20140901*

本书将随着Yii官方对Yii2的开发而不断丰富内容和进行更新，需要阅读最新的内容请访问 《深入理解Yii2.0》的最新版 。建议收藏网址，以便今后访问。

有任何疑问、需求、批评、建议，都欢迎读者朋友通过底部评论留言告知，谢谢支持！

CHAPTER

ONE

导读

这是一本干货。主要讲解Yii2.0及所代表的最新一代Web开发框架的新特性、新技术、新思路、新模式。 学习完这里面的全部内容，你得到的，不仅仅是Yii怎么使用的实操技巧，还将了解其实现的技术原理和内幕， 更为重要的，将是Web开发中最为流行和成熟的设计模式和开发思路。希望读者朋友能够从中有所收获吧。

由于本人的水平有限，技术欠精，写的内容不一定对，欢迎读者朋友们在网页底部留下评论，批评指正。

1.1 Yii是什么

Yii是一个PHP框架，用于开发各种类型的Web应用。Yii官方将其定义为高性能、基于组件的框架。

就个人的经验而言，总结Yii具有以下特点：

- Yii比较“潮”。 Yii开发团队一直关注业内Web开发的最新技术，很注意吸收当下最为流行的技术。可以说，近年来Web开发中最潮的技术都可以在Yii身上或多或少的看到影子。 比如，刚刚开始的时候Yii带有明显的Ruby on Rails风格； 比如Yii2中刚刚现实的命名空间等PHP最新特性支持等。一个跟得上潮流和趋势的框架，才具有吸引力和生命力，学习起来才有意思、有意义。
- Yii比较“易”。 正如其名字的发音，Yii是一个比较易学、易用的框架。代码质量很高，有许多可以学习的地方。 注释清晰、文档丰富阅读代码难度不高。社区活跃，官方论坛有中文区，国内论坛人气也还OK，知识获取容易。 架构相对稳定，从Yii1.1到Yii2的变化看，许多原来的约定和沉淀的经验都还适用。
- Yii比较“全”。 就Web开发而言，无论是哪种类型的应用、无论是哪个开发阶段的常见问题，Yii都有成熟、高效、可靠的解决方案。 对于典型的Web开发而言，这已经是足够了。比如，伪静态化、国际化、RESTful等，Yii都有提供编程的框架。 但是，从规模上来讲，Yii还算不上一个大型框架。 个人对其的评价是一个中型偏轻点的框架，对于绝大多数的应用而言，肯定是充分的了。
- Yii比较“快”。 Yii官方把运行效率作为一个重要的特点来宣传。从实际使用看，在诸多PHP框架中，确实效率上具有一定优势。 但个人认为这点其实不是特点重要，作为框架的使用者，也就是开发人员来讲，更重要的是开发效率。 于由Yii架构合理，Web中开发常用的思路和模式都可以很顺地套上使用。 在Web开发中常遇以的细节上，Yii也提供了许多现成解决方案，拿来就可以使用，非常高效、方便。

目前，Yii有两个最主要的版本，Yii1.1和Yii2.0。Yii1.1是老的版本，我写这个教程时，最新版本号是1.1.15。 Yii1.1现在已经不再进行新的开发了，官方只是进行维护，更新安全漏洞等，不会再有新的功能特性的引入。 而Yii2.0是在Yii1.1的基础上完全推倒重新写的一个框架，吸收了当前最新的技术和开发中主流的约定。 Yii2.0是最新一代Web开发框架的代表。

要感谢Yii开发团队精益求精的不懈努力，为广大Web开发者创造了如此优秀的框架。 本人自Yii1.1起就开始接触并使用Yii了，由于工作和爱好关系，也接触过一些框架了。总的说，至今对Yii很满意， 最最心仪的是两点：学了Yii，就学到了许多当下最流行、最成熟的东西；开发起快，改进来快。

1.2 背景知识

请留意，虽然本教程以Yii2为主要内容，但并不要求读者具有Yii1.1的开发经验。 虽然具有这些背景知识可以更快的掌握Yii2，但在教程中，我会帮助没有Yii1.1相关知识的读者补充有关的概念。 只要有了这些概念，读者并不需要从头学习Yii1.1，就可以直接上手Yii2了。

当然，Yii作为一个PHP框架，读者朋友最好能够了解一下PHP，并不需要多精通，只需要看得懂代码，会写简单的代码，编程的时候大概知道要使用哪些函数，就基本足够了，边用边学，也是一种学习方法。

同时，Yii还是一个面向对象的框架。这意味着在代码组织和问题解决的思路上，Yii都体现了面向对象的思想。 要用Yii来开发，最好也要遵循这一思想。因此，读者最好对面向对象编程有一定的了解。 其实，看一个程序员水平的高低，不单是对于某种语言、某种开发框架的熟练程度。 更重要的，是看其解决问题的思路和方法。其中一大类方法就是面向对象方法。 从这点来看，虽然学习和使用Yii并不需要多高深的面向对象的方法。 但作为过来人，我还是希望各位读者朋友可以系统地、全面地学习面向对象的开发方法。 特别是Web开发中常用的设计模式，本教程也会在涉及到时，进行专门讲解。

1.3 Yii2.0目前的状态

Yii2.0目前是Beta版本。自Alpha版开始，官方建议不要将Yii2.0用于产品，但可以用于研究学习。在发布Beta版之后，官方又说明自Alpha以来，已经有许多网站和应用在使用Yii2.0开发了，而且使用状况稳定。 而且，Beta版不会涉及到架构及接口方面的修改，主要是文档完善和Bug修复。

从个人的使用经验看，在Alpha时，确实存在由于框架代码的更新，导致原来自己写的能够正常运行代码， 出现莫明妙错误的情况。大概有7、8次吧。但每次出现问题时，根据Yii的调试信息，可以定位错误代码的位置和原因， 很快可以判断是由于框架修改的原因造成的。随后，只需要查看最新的Yii2.0代码库，查看最近提交的comment， 就可以发现代码库中修改的内容了。根据新的框架代码，对自己的代码作出相应的调整即可。

应该说在Alpha版本时，虽然存在代码不稳定的情况，但出现状况的次数对我个人而言是可以接受的， 排查起来很容易，代码调整起来也不困难。而到了Beta版本，目前为止，还未出现类似问题。

这里只是客观讲述Yii2.0目前所处的状态。对于学习研究而言，一点不妨碍我们对Yii的深入学习。对于生产开发而言，如果对于大型应用、团队大规模开发的场景，建议可以先等等，等正式版发布后再使用。 而如果是小型应用，2－3人小团队开发、个人开发，那么，直接上Yii2.0是没有太大问题的。 只是每次更新框架代码前，留意框架代码修改的内容，并针对相应的修改作出调试，确保自己的代码不受影响。

1.4 Yii2.0 对比 Yii1.1 的重大变化(TBD)

这一节是专门为已经有Yii1.1基础的读者朋友写的。将Yii2.0与Yii1.1的不同着重写出来，对比学起来会快得多。 而对于从未接触过Yii的读者朋友，这一节的内容扫一扫就可以了，作为对过往历史的一个了解就够了。 如果有的内容你一时没看明白，也不要紧，后面会讲清楚的。另外，没有Yii1.1的经验，并不妨碍对Yii2.0的学习。

Yii官方有专门的文档归纳总结1.1版本和2.0版本的不同。以下内容，主要来自于官方的文档，我做了下精简， 选择比较重要的变化，并加入了一些个人的经验。

1.4.1 PHP新特性

从对PHP新特性的使用上，两者就存在很大不同。Yii2.0大量使用了PHP的新特性，这在Yii1.1中是没有的。因此，PHP2.0对于PHP的版本要求更高，要求PHP5.4及以上。Yii2.0中使用到的PHP新特性，主要有：

- 命名空间(Namespace)

- 匿名函数
- 数组短语法形式：`[1,2,3]` 取代 `array(1,2,3)` 。这在多维数组、嵌套数组中，代码更清晰、简短。
- 在视图文件中使用PHP的 `<?=` 标签，取代 `echo` 语句。
- 标准PHP库(SPL) 类和接口，具体可以查看 SPL Class and Interface
- 延迟静态绑定， 具体可以查看 Late Static Bindings
- PHP标准日期时间
- 特性(Traits)
- 使用PHP `intl` 扩展实现国际化支持， 具体可以查看 PECL init 。

了解Yii2.0使用了PHP的新特性，可以避免开发时由于环境不当，特别是开发生产环境切换时，产生莫名其妙的错误。 同时，也是让读者朋友借机学习PHP新知识的意思。

1.4.2 命名空间(Namespace)

Yii2.0与Yii1.1之间最显著的不同是对于PHP命名空间的使用。Yii1.1中没有命名空间一说， 为避免Yii核心类与用户自定义类的命名冲突，所有的Yii核心类的命名，均冠以 `C` 前缀，以示区别。

而Yii2.0中所有核心类都使用了命名空间，因此， `C` 前缀也就人老珠黄，退出历史舞台了。

命名空间与实际路径相关联，比如 `yii\base\Object` 对应Yii目录下的 `base/Object.php` 文件。

1.4.3 基础类

Yii1.1中使用了一个基础类 `CComponent` ，提供了属性支持等基本功能，因此几乎所有的Yii核心类都派生自该类。 到了Yii2.0，将一家独大的 `CComponent` 进行了拆分。拆分成了 `yii\base\Object` 和 `yii\base\Component` 。 拆分的考虑主要是 `CComponent` 尾大不掉，有影响性能之嫌。 于是，Yii2.0中，把 `yii\base\Object` 定位于只需要属性支持，无需事件、行为。 而 `yii\base\Component` 则在前者的基础上，加入对于事件和行为的支持。 这样，开发者可以根据需要，选择继承自哪基础类。

这一功能上的明确划分，带来了效率上的提升。在仅表示基础数据结构，而非反映客观事物的情况下， `yii\base\Object` 比较适用。

值得一提的是， `yii\base\Object` 与 `yii\base\Component` 两者并不是同一层级的，前者是后者他爹。

1.4.4 事件(Event)

在Yii1.1中，通过一个 `on` 前缀的方法来创建事件，比如 `CActiveRecord` 中的 `onBeforeSave()` 。在Yii2.0中，可以任意定义事件的名称，并自己触发它:

```
$event = new \yii\base\Event;
// 使用 trigger() 触发事件
$component->trigger($eventName, $event);

// 使用 on() 前事件handler与对象绑定
$component->on($eventName, $handler);
// 使用 off() 解除绑定
$component->off($eventName, $handler);
```

1.4.5 别名(Alias)

Yii2.0中改变了Yii1.1中别名的使用形式，并扩大了别名的范畴。 Yii1.1中，别名以 . 的形式使用:

```
RootAlias.path.to.target
```

而在Yii2.0中，别名以 @ 前缀的方式使用:

```
@yii/jui
```

另外，Yii2.0中，不仅有路径别名，还有URL别名:

```
// 路径别名
Yii::setAlias('@foo', '/path/to/foo');

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

另名与命名空间是紧密相关的，Yii建议为所有根命名空间都定义一个别名，比如上面提到的 `yii\base\Object` ， 事实上是定义了 `@yii` 的名别，表示Yii在系统中的安装路径。 这样一来，Yii就能根据命名空间找到实际的类文件所在路径，并自动加载。这一点上，Yii2.0与Yii1.1并没有本质区别。

而如果没有为根命名空间定义别名，则需要进行额外的配置。将命名空间与实际路径的映射关系，告知Yii。

关于别名的更详细内容请看 别名 。

1.4.6 视图(View)

Yii1.1中，MVC（model-view-controller）中的视图一直是依赖于Controller的，并非是真正意义上独立的View。 Yii2.0引入了 `yii\web\View` 类，使得View完全独立。这也是一个相当重要变化。

首先，Yii2.0中，View作为Application的一个组件，可以在全局中代码中进行访问。 因此，视图渲染代码不必再局限于Controller中或Widget中。

其次，Yii1.1中视图文件中的 `$this` 指的是当前控制器，而在 Yii2.0中，指的是视图本身。 要在视图中访问控制器，使用 `$this->context` 。

同时，Yii1.1中的 `CClientScript` 也被淘汰了，相关的前端资源及其依赖关系的管理，交由Assert Bundle `yii\web\AssertBundle` 来专职处理。 一个Assert Bundle代表一系列的前端资源，这些前端资源以目录形式进行管理，这样显得更有序。 更为重要的是，Yii1.1中需要你格外注意资源在HTML中的顺序，比如CSS文件的顺序（后面的会覆盖前面的）， JavaScript文件的顺序（前后顺序出错会导致有的库未加载）等。 而在Yii2.0中，使用一个Assert Bundle可以定义依赖于另外的一个或多个Assert Bundle的关系， 这样在向HTML页面注册这些CSS或者JavaScript时，Yii2.0会自动把所依赖的文件先注册上。

在视图模版引擎方面，Yii2.0仍然使用PHP作为主要的模版语言。 同时官方提供了两个扩展以支持当前两大主流PHP模版引擎：Smarty和Twig，而对于Pardo引擎官方不再提供支持了。 当然，开发者可以通过设置 `yii\web\View::$renderers` 来使用其他模版。

另外，Yii1.1中，调用 `$this->render('viewFile', ...)` 是不需要使用 `echo` 命令的。而Yii2.0中，记得 `render()` 只是返回视图渲染结果，并不对直接显示出来，需要自己调用 `echo`

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

如果有一天你发现怎么Yii输出了个空白页给你，就要注意是不是忘记使用 `echo` 了。 还别说，这个错误很常见，特别是在对Ajax请求作出响应时，会更难发现这一错误。请你们编程时留意。

在视图的主题(Theme)化方面，Yii2.0的运作机理采用了完全不同的方式。 在Yii2.0中，使用路径映射的方式，将一个源视图文件路径，映射成一个主题化后的视图文件路径。 因此， `['/web/views' => '/web/themes/basic']` 定义了一个主题映射关系， 源视图文件 `/web/views/site/index.php` 主题化后将是 `/web/themes/basic/site/index.php` 。 因此，Yii1.1中的 `CThemeManager` 也被淘汰了。

1.4.7 模型(Model)

MVC中的M指的就是模型，Yii1.1中使用 `CModel` 来表示，而Yii2.0使用 `yii\base\Model` 来表示。

Yii1.1中， `CFormModel` 用来表示用户的表单输入，以区别于数据库中的表。 这在Yii2.0中也被淘汰，Yii2.0倾向于使用继承自 `yii\base\Model` 来表示提交的表单数据。

另外，Yii2.0为Model引入了 `yii\base\Model::load()` 和 `yii\base\Model::loadMutiple()` 两个新的方法， 用于简化将用户输入的表单数据赋值给Model:

```
// Yii2.0使用load()等同于下面的用法
$model = new Post;
if ($model->load($_POST)) {
    ... ...
}

// Yii1.1中常用的套路
if (isset($_POST['Post'])) {
    $model->attributes = $_POST['Post'];
}
```

另外一个重要变化就是Yii2.0中引入了场景的概念，使得一个Model可以适用于不同的场景， 并针对不同的场景，决定哪些字段有效。比如，对于用户模型，在注册场景中，通常需要用户输入两次密码进行确认， 而在登陆场景中，密码只需要输入一次。在Yii1.1中，针对这种情况，通常要有两个Model来实现。 而在Yii2.0中，只需要一个Model，并通过 `yii\base\Model::secnarios()`

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

这也意味着， `unsafe` 验证在Yii2.0中也没有了立足之地，凡是 `unsafe` 的字段，就不在特定的场景中列出来。 或者给这个字段加上 ! 前缀。

1.4.8 控制器(Controller)

除了上面讲到的控制器中要使用 `echo` 来显示渲染视图的输出这点区别外， Yii1.1与Yii2.0的控制器还表现出更为明显的区别，那就是动作过滤器(Action Filter) 的不同。

在Yii2.0中，动作过滤器以行为(behavior)的方式出现， 一般继承自 `yii\base\ActionFilter` ，并注入到一个控制器中，以发生作用。比如，Yii1.1中很常见的:

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

看着是不是有点像，但又确实不一样?

1.4.9 Active Record

还记得么？在Yii1.1中，数据库查询被分散成 CDbCommand ， CDbCriteria 和 CDbCommandBuilder 。所谓天下大势分久必合，到了Yii2.0，采用 yii\db\Query 来表示数据库查询:

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

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

最最最爽的是， yii\db\Query 可以在 Active Record中使用，而在Yii1.1中，要结合两者，并不容易。

Active Record在Yii2.0中最大的变化一个是查询的构建，另一个是关联查询的处理。

Yii1.1中的 CDbCriteria 在Yii2.0中被 yii\db\ActiveQuery 所取代，这个把前辈拍死在沙滩上的家伙，继承自 yii\db\Query ，所以可以进行类似上面代码的查询。调用 yii\db\ActiveRecord::find() 就可以启动查询的构建了:

```
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

这在Yii1.1中，是不容易实现的。特别是比较复杂的查询关系。

在关联查询方面，Yii1.1是在一个统一的地方 relations() 定义关联关系。而Yii2.0改变了这一做法，定义一个关联关系:

- 定义一个getter方法
- getter方法的方法名表示关联关系的名称，如 getOrders() 表示关系 orders
- getter方法中定义关联的依据，通常是外键关系
- getter返回一个 yii\db\ActiveQuery 对象

比如以下代码就定义了 Customer 的 orders 关联关系:

```
class Customer extends \yii\db\ActiveRecord
{
    ... ...

    public function getOrders()
    {
        // 关联的依据是 Order.customer_id = Customer.id
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

这样的话，可以通过 Customer 访问关联的 Order

```
// 获取所有与当前 $customer 关联的 orders
$orders = $customer->orders;

// 获取所有关联 orders 中，status=1 的 orders
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

对于关联查询，有积极的方式也有消极的方式。区别在于采用积极方式时，关联的查询会一并执行，而消极方式时，仅在显示调用关联记录时材会执行关联的查询。

在积极方式的实现上，Yii2.0与Yii1.1也存在不同。Yii1.1使用一个JOIN查询， 来实现同时查询主记录及其关联的记录。 而Yii2.0弃用JOIN查询的方式，而使用两个顺序的SQL语句， 第一个语句查询主记录，第二个语句根据第一个语句的返回结果进行过滤。

同时，Yii2.0为Active Record引入了 `asArray()` 方法。在返回大量记录时，可以以数组形式保存，而不再以对象形式保存，这样可以节约大量的空间，提高效率。

另外一个变化是，在Yii1.1中，字段的默认值可以通过为类的public 成员变量赋初始值来指定。 而在Yii2.0中，这样的方式是行不通的，必须通过重载 `init()` 成员函数的方式实现了。

CHAPTER

TWO

YII 基础

这一章将讲解Yii框架中，最基础的部分。说基础，不是这方面的知识有多浅显。 而是说，这些内容是驱动整个Yii框架的基石。这些知识对于Yii中所有的类几乎都适用， 也是理解整个Yii所必须具备的基础。

他们分别是属性（property），事件（event），行为（hebavior）。

2.1 属性

Yii是一个纯面向对象的框架。因此，本章主要介绍Yii中有关面向对象的基础知识：属性（property）、事件（event）、行为（behavior）等。

2.1.1 对象（object）和组件（component）

前面说过，官方将Yii定位于一个基于组件的框架。可见组件这一概念是Yii的基础。如果你有兴趣阅读Yii的源代码或是API文档，你将会发现，Yii几乎所有的核心类都派生于（继承自）`yii\base\Component`。

Yii1.1是，就已经有了component了。Yii2将Yii1中的CComponent拆分成两个类：`yii\base\Object` 和 `yii\base\Component`。Object比较轻量级些，通过getter和setter定义了类的属性（property）。Component派生自Object，并支持事件（event）和行为（behavior）。因此，Component类具有三个重要的特性：

- 属性（property）
- 事件（event）
- 行为（behavior）

相信你或多或少了解过，这三个特性是丰富和拓展类功能、改变类行为的重要切入点。

2.1.2 属性（property）

属性用于表征类的状态，从访问的形式上看，属性与成员变量没有区别，你能一眼看出 `$object->foo` 中的 `foo` 是成员变量还是属性么？显然不行。成员变量和属性的区别与联系在于：

- 成员变量是属性，属性不一定是成员变量。但一般情况下，属性会由某个或某些成员变量来表示，这些成员变量通常是私有的。
- 成员变量没有读写权限控制，而属性可以指定为只读或只写。
- 成员变量不对读出作任何后处理，不对写入作任何预处理，而属性可以。

在Yii中，由 `yii\base\Object` 提供了对属性的支持，因此，如果要使你的类支持属性，必须继承自 `yii\base\Object`。Yii中属性是通过PHP的魔法函数 `__get()` `__set()` 来产生作用的。下面的代码是 `yii\base\Object` 类对于 `__get()` 和 `__set()` 的定义：

```php
public function __get($name)                // 这里$name是属性名
{
    $getter = 'get' . $name;                // getter函数的函数名
    if (method_exists($this, $getter)) {
        return $this->$getter();            // 调用了getter函数
    } elseif (method_exists($this, 'set' . $name)) {
        throw new InvalidCallException('Getting write-only property: ' . get_class($this) . '::'
    } else {
        throw new UnknownPropertyException('Getting unknown property: ' . get_class($this) . '::'
    }
}

public function __set($name, $value)       // $name是属性名，$value是拟写入的属性值
{
    $setter = 'set' . $name;                // setter函数的函数名
    if (method_exists($this, $setter)) {
        $this->$setter($value);             // 调用setter函数
    } elseif (method_exists($this, 'get' . $name)) {
        throw new InvalidCallException('Setting read-only property: ' . get_class($this) . '::' .
    } else {
        throw new UnknownPropertyException('Setting unknown property: ' . get_class($this) . '::'
    }
}
```

我们知道，在访问和写入对象的一个不存在的成员变量时，`__get()` `__set()` 会被自动调用，Yii正是利用这点，提供对属性的支持的。从上面的代码中，可以看出，如果访问一个对象的某个属性，Yii会调用名为 `get属性名()` 的函数。如，`SomeObject->Foo`，会自动调用 `SomeObject->getFoo()`。如果修改某一属性，会调用相应的setter函数。如，`SomeObject->Foo = $someValue`，会自动调用 `SomeObject->setFoo($someValue)`。

因此，要实现属性，通常有三个步骤:

- 继承自 `yii\base\Object`。
- 声明一个用于保存该属性的私有成员变量。
- 提供getter或setter函数，或两者都提供，用于访问、修改上面提到的私有成员变量。如果只提供了getter，那么该属性为只读属性，只提供了setter，则为只写。

如下的Post类，实现了可读可写的属性title:

```php
class Post extend yii\base\Object     // 第一步：继承自 yii\base\Object
{
    private $_title;                  // 第二步：声明一个私有成员变量

    public function getTitle()        // 第三步：提供getter和setter
    {
        return $this->_title;
    }

    public function setTitle($value)
    {
        $this->_title = trim($value);
    }
}
```

从理论上来讲，将 `private $_title` 写成 `public $title`，也是可以实现对 `$post->title` 的读写的。但这不是好的习惯，理由如下:

- 失去了类的封装性。一般而言，成员变量对外不可见是比较好的编程习惯。从这里你也许没看出来，但是假如有一天，你不想让用户修改标题了，你怎么改？怎么确保代码中没有直接修改标题？如果提供了setter，只要把setter删掉，那么一旦有没清理干净的对标题的写入，就会抛出异常。而使用 `public $title` 的方法的话，你改成 `private $title` 可以排查写入的异常，但是读取的也被禁止了。

- 对于标题的写入，你想去掉空格，使用setter的方法，只需要像上面的代码段一样在这个地方调用 `trim()` 就可以了。但如果使用 `public $title` 的方法，那么毫无疑义，每个写入语句都要调用 `trim()` 。你能保证没有一处遗漏?

使用 `public $title` 是一时之快，看起来简单，但今后的修改是个麻烦事。简直可以说是恶梦。这就是软件工程的意义所在，通过一定的方法，使代码易于维护、便于修改。一时看着好像没必要，但实际上吃过亏的朋友或者被客户老板逼着修改上一个程序员写的代码，问候过他亲人的，都会觉得这是十分必要的。

值得注意的是：

- 由于自动调用 `__get()` `__set()` 的时机仅仅发生在访问不存在的成员变量时。因此，如果定义了成员变量 `public $title` 那么，就算定义了 `getTitle()` `setTitle()` ，他们也不会被调用。因为 `$post->title` 时，会直接指向该 `pulic $title`，`__get()` `__set()` 是不会被调用的。从根上就被切断了。
- 由于PHP对于类方法不区分大小写，即大小写不敏感，`$post->getTitle()` 和 `$post->gettitle()` 是调用相同的函数。因此，`$post->title` 和 `$post->Title` 是同一个属性。
- 由于 `__get()` `__set()` 都是public的，无论将 `getTitle()` `setTitle()` 声明为 public, private, protected，都没有意义，外部同样都是可以访问。所以，所有的属性都是public 的。
- 由于 `__get()` `__set()` 都不是static的，因此，没有办法使用static 的属性。

2.1.3 Object的其他与属性相关的方法

除了 `__get()` `__set()` 之外， `yii\base\Object` 还提供了以下方法便于使用属性：

- `__isset()` 用于测试属性值是否不为 `null` ，在 `isset($object->property)` 时被自动调用。注意该属性要有相应的getter。
- `__unset()` 用于将属性值设为 `null` ，在 `unset($object->property)` 时被自动调用。注意该属性要有相应的setter。
- `hasProperty()` 用于测试是否有某个属性。即，定义了getter或setter。如果 `hasProperty()` 的参数 `$checkVars = true` （默认为true），那么只要具有同名的成员变量也认为具有该属性，如前面提到的 `public $title` 。
- `canGetProperty()` 测试一个属性是否可读，参数 `$checkVars` 的意义同上。
- `canSetProperty()` 测试一个属性是否可写，参数 `$checkVars` 的意义同上。

2.1.4 Component的属性

`yii\base\Component` 继承自 `yii\base\Object` ，因此，他也具有属性等基本功能。

但是，由于Componet还引入了事件、行为，因此，它并非简单继承了Object的属性实现方式，而是基于同样的机制，重载了 `__get()` `__set()` 等函数。但从实现机制上来讲，是一样的。这个不影响理解。

2.1.5 Object的配置方法

Yii提供了一个统一的配置对象的方式。这一方式贯穿整个Yii。Application对象的配置就是这种配置方式的体现：

```
1 $config = yii\helpers\ArrayHelper::merge(
2     require(__DIR__ . '/../../common/config/main.php'),
3     require(__DIR__ . '/../../common/config/main-local.php'),
4     require(__DIR__ . '/../config/main.php'),
5     require(__DIR__ . '/../config/main-local.php')
6 );
```

```
7
8  $application = new yii\web\Application($config);
```

$config 看着复杂，但本质上就是一个各种配置项的数组。Yii中就是统一使用数组的方式对对象进行配置，而实现这一切的关键就在 yii\base\Object 定义的构造函数中:

```
1  public function __construct($config = [])
2  {
3      if (!empty($config)) {
4          Yii::configure($this, $config);
5      }
6      $this->init();
7  }
```

所有 yii\base\Object 的构建流程是:

- 构建函数以 $config 数组为参数被自动调用。
- 构建函数调用 Yii::configure() 对对象进行配置。
- 在最后，构造函数调用对象的 init() 方法进行初始化。

数组配置对象的秘密在 Yii::configure() 中，但说破了其实也没有什么神奇的:

```
1  public static function configure($object, $properties)
2  {
3      foreach ($properties as $name => $value) {
4          $object->$name = $value;
5      }
6
7      return $object;
8  }
```

配置的过程就是遍历 $config 配置数组，将数组的键作为属性名，以对应的数组元素的值对对象的属性赋值。因此，实现Yii这一统一的配置方式的要点有:

- 继承自 yii\base\Object 。
- 对对象属性提供setter方法，以正确处理配置过程。
- 如果需要重载构造函数，请将 $config 作为该构造函数的最后一个参数，并将该参数传递给父构造函数。
- 重载的构造函数的最后，一定记得调用父构造函数。
- 如果重载了 yii\base\Object::Init() 函数，注意一定要在重载函数的开头调用父类的 Init() 。

只要实现了以上要点，就可以使得你编写的类可以按照Yii约定俗成的方式进行配置。这在编写代码的过程中，带来许多便利。

像你这么聪明的，肯定会提出来，如果配置数组是的某个配置项，也是一个数组，这怎么办? 如果某个对象的属性，也是一个对象，而非一个简单的数值或字符串时，又怎么办?

这两个问题，其实是同质的。如果一个对象的属性，是另一个对象，就像Application里会引入诸多的Component一样，这是很常见的。如后面会看到的 $app->request 中的 request 属性就是一个对象。那么，在配置 $app 时，必然要配置到这个 reqeust 对象。既然 request 也是一个对象，那么他的配置要是按照Yii的规矩来，也就是用一个数组来配置它。因此，上面提到的这两个问题，往往是同质的。

那么，怎么实现呢? 秘密在于setter函数。由于 $app 在进行配置时，最终会调用 configure 函数。该函数又不区分配置项是简单的数值还是数组，就直接使用 $object->$name = $value 完成属性的赋值。那么，对于对象属性，其配置值 $value 是一个数组，为了使其正确配置。你需要在其setter函数上做出正确的处理方式。典型的Yii应用 yii\web\Application 就是依靠定义专门的setter函数，实现自动处理配置项的。比如，我们在Yii的配置文件中，可以看到一个配置项 components ，一般情况下，他的内容是这样的:

```
'components' => [
    'request' => [
        // !!! insert a secret key in the following (if it is empty) - this is required by cookie
        'cookieValidationKey' => 'v7mBbyetv4ls7t8UIqQ2IBO60jY_wf_U',
    ],
    'user' => [
        'identityClass' => 'common\models\User',
        'enableAutoLogin' => true,
    ],
    'log' => [
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
                'levels' => ['error', 'warning'],
            ],
        ],
    ],
    'errorHandler' => [
        'errorAction' => 'site/error',
    ],
],
```

这是一个典型的数组，而且是嵌套数组。那么Yii是如何把他们配置好的呢？聪明的你肯定想到了，Yii一定是定义了一个名为 `setComponents` 的setter函数。当然，Yii并未将该函数放在 `yii\web\Application` 里，而是放在了 `yii\di\ServiceLocator` 里面:

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

这里有个成员函数， `$this->set()` ，我们暂不讲这个东西，只要知道这个函数把配置文件中的 `components` 配置项搞定就可以了。至于 `yii\web\Application` 和 `yii\di\ServiceLocator` 的关系，就是前者继承自后者，但不是直接继承哦，不过这个关系并不大了。

但是，正如硬币的两面，在提供便利的同时，component由于event和behavior这两个特性，也牺牲了一定的效率。如果开发中不需要使用event和behavior这两个特性，比如表示数据的类，那么，可以不从Component继承，而从Object继承。相比于Component，Object只提供了property，而没有event和behavior。典型的应用场景就是如果表示用户输入的一组数据，那么，使用Object。特别是需要以一个大数组来表示数据，那么，数组元素一般采用Object，而不是采用Component。而如果需要对对象的行为和能响应处理的事件进行处理，毫无疑问应当采用Component。从效率来讲，Object更接近原生的PHP类，因此，在可能的情况下，应当优先使用Object。

对于所有的Object，包括Component，遵循以下生命周期:

1. 预初始化阶段。可以在这一阶段设置property的默认值。
2. 对象配置阶段。也就是前面提到构造函数的 `$config` 参数。这一阶段可以覆盖前一阶段设置的property的默认值，并补充没有默认值的参数，也就是必备参数。
3. 后初始化阶段。也就是调用 `yii\base\Object::init()` ，可以对配置阶段设置的值进行检查，并规范类的property。
4. 类方法调用阶段。前面三个阶段是不可分的，也就是说一个类一但实例化，那么就至少经历了前三个阶段。此时，该对象的状态是确定且可靠的，不存在不确的property，要么是默认值，要么是传入的配置值，如果传入的配置有误或者冲突，那么也经过了检查和规范。也就是说，你就放心用吧。

2.2 事件（Event）

使用事件，可以在特点的时点，触发执行预先设定的一段代码，事件既是代码解耦的一种方式，也是设计业务流程的一种模式。现代软件中，事件无处不在，比如，你发了个微博，触发了一个事件，导致关注你的人，看到了你新发出来的内容。对于事件而言，有这么几个要素：

- 这是一个什么事件？一个软件系统里，有诸多事件，发布新微博是事件，删除微博也是一种事件。
- 谁触发了事件？你发的微博，就是你触发的事件。
- 谁负责监听这个事件？或者谁能知道这个事件发生了？服务器上处理用户注册的模块，肯定不会收到你发出新微博的事件。
- 事件怎么处理？对于发布新微博的事件，就是通知关注了你的其他用户。
- 事件相关数据是什么？对于发布新微博事件，包含的数据至少要有新微博的内容，时间等。

2.2.1 Yii中与事件相关的类

Yii中，事件是在 `yii\base\Component` 中引入的，注意， `yii\base\Object` 不支持事件。所以，当你需要使用事件时，请从 `yii\base\Component` 进行继承。同时，Yii中还有一个与事件紧密相关的 `yii\base\Event` ，他封装了与事件相关的有关数据，并提供一些功能函数作为辅助：

```
class Event extends Object
{
    public $name;                       // 事件名
    public $sender;                     // 事件发布者，通常是调用了 trigger() 的对象或类。
    public $handled = false;            // 是否终止事件的后续处理
    public $data;                       // 事件相关数据

    private static $_events = [];

    public static function on($class, $name, $handler, $data = null, $append = true)
    {
        // ... ...
        // 用于绑定事件handler
    }

    public static function off($class, $name, $handler = null)
    {
        // ... ...
        // 用于取消事件handler绑定
    }

    public static function hasHandlers($class, $name)
    {
        // ... ...
        // 用于判断是否有相应的handler与事件对应
    }

    public static function trigger($class, $name, $event = null)
    {
        // ... ...
        // 用于触发事件
    }
}
```

2.2.2 事件handler

所谓事件handler就是事件处理程序，负责事件触发后怎么办的问题。从本质上来讲，一个事件handler就是一段PHP代码，即一个PHP函数。对于一个事件handler，可以是以下的形式提供：

- 一个PHP全局函数的函数名，不带参数和括号，光秃秃的就一个函数名。如 `trim`，注意，不是 `trim($str)` 也不是 `trim()`。
- 一个对象的方法，或一个类的静态方法。如 `$person->sayHello()` 可以用为事件handler，但要改写成以数组的形式，`[$person, 'sayHello']`，而如果是类的静态方法，那应该是 `['namespace\to\Person', 'sayHello']`。
- 匿名函数。如 `function ($event) { ... }`

但无论是何种方式提供，一个事件handler必须具有以下形式：

```
function ($event) {
    // $event 就是前面提到的 yii\base\Event
}
```

也就是只有长得像上面这样的，才可以作为事件handler。

还有一点容易犯错的地方，就是对于类自己的成员函数，尽管在调用 `on()` 进行绑定时，看着这个handler是有效的，因此，有的小伙伴就写成这样了 `$this->on(EVENT_A, 'publicMethod')`，但事实上，这是一个错误的写法。以字符串的形式提供handler，只能是PHP的全局函数。这是由于handler的调用是通过 `call_user_func()` 来实现的。因此，handler的形式，与 `call_user_func()` 的要求是一致的。这将在 事件的触发 中介绍。

2.2.3 事件的绑定与解除

事件的绑定

有了事件handler，还要告诉Yii，这个handler是负责处理哪种事件的。这个过程，就是事件的绑定，把事件和事件handler这两个蚂蚱绑在一根绳上，当事件跳起来的时候，就会扯动事件handler啦。

`yii\base\Component::on()` 就是用来绑定的，很容易就猜到，`yii\base\Component::off()` 就是用来解除的。对于绑定，有以下形式：

```
$person = new Person;

// 使用PHP全局函数作为handler来进行绑定
$person->on(Person::EVENT_GREET, 'person_say_hello');

// 使用对象$obj的成员函数say_hello来进行绑定
$person->on(Person::EVENT_GREET, [$obj, 'say_hello']);

// 使用类Greet的静态成员函数say_hello进行绑定
$person->on(Person::EVENT_GREET, ['app\helper\Greet', 'say_hello']);

// 使用匿名函数
$person->on(Person::EVENT_GREET, function ($event) {
    echo 'Hello';
});
```

事件的绑定可以像上面这样在运行时以代码的形式进行绑定，也可以在配置中进行绑定，当然，这个配置生效的过程其实也是在运行时的。原理要看配置的部分章节。TODO

上面的例子只是简单的绑定了事件与事件handler，如果有额外的数据传递给handler，可以使用 `yii\base\Component::on()` 的第三个参数。这个参数将会写进 `Event` 的相关数据字段，即属性 `data`。如：

```php
$person->on(Person::EVENT_GREET, 'person_say_hello', 'Hello World!');

// 'Hello World!' 可以通过 $event访问。
function person_say_hello($event)
{
    echo $event->data;                  // 将显示 Hello World!
}
```

yii\base\Component 维护了一个handler数组，用来保存绑定的handler:

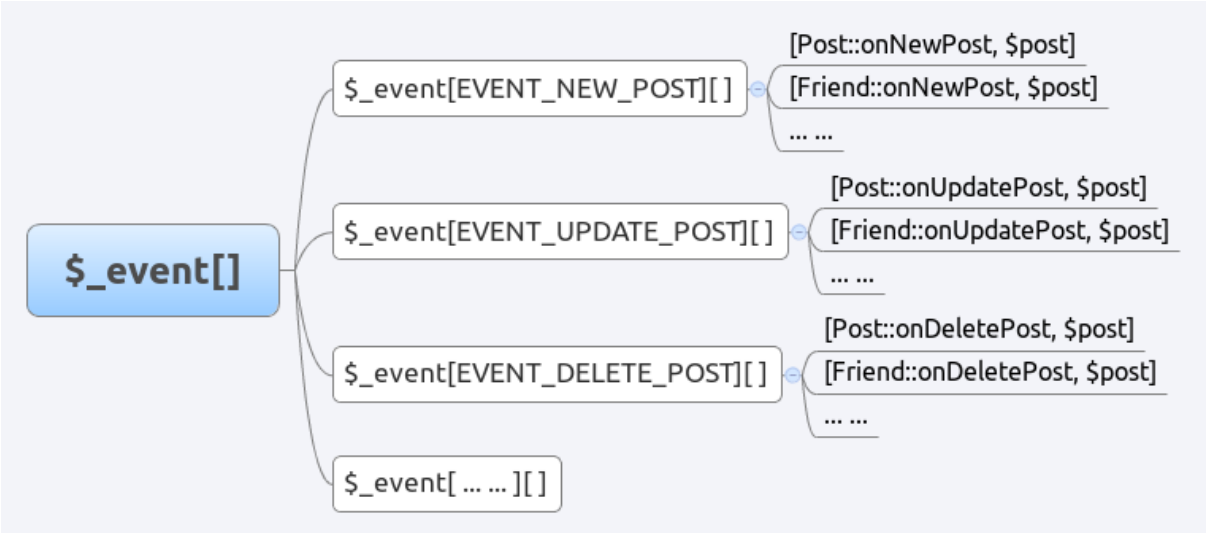
```php
// 这个就是handler数组
private _events = [];

// 绑定过程就是将handler写入_event[]
public function on($name, $handler, $data = null, $append = true)
{
    $this->ensureBehaviors();
    if ($append || empty($this->_events[$name])) {
        $this->_events[$name][] = [$handler, $data];
    } else {
        array_unshift($this->_events[$name], [$handler, $data]);
    }
}
```

保存handler的数据结构

从上面代码我们可以了解两个方向的内容，一是 $_event[] 的数据结构，二是绑定handler的逻辑。

从handler数组 $_evnet[] 的结构看，首先他是一个数组，保存了该Component的所有事件handler。该数组的下标为事件名，数组元素是形为一系列 [$handler, $data] 的数组，如下图所示:



在事件的绑定逻辑上，按照以下顺序： - 参数 $append 是否为 true 。为 true 表示所要绑定的事件handler要放在 $_event[] 数组的最后面。这也是默认的绑定方式。 - 参数 $append 是否为 false 。表示handler要放在数组的最前面。这个时候，要多进行一次判定。 - 如果所有绑定的事件还没有已经绑定好的handler，也就是说，将要绑定的handler是第一个，那么无论 $append 是否是 true ，该handler必然是第一个元素，也是最后一个元素。 - 如果 $append 为 false ，且要绑定的事件已经有了handler，那么，就将新绑定的事件插入到数组的最前面。

handler在 $event[] 数组中的位置很重要，代表的是执行的先后顺序。这个在 多个事件handler的顺序 中会讲到。

事件的解除

在解除时，就是使用 unset() 函数，处理 $_event[] 数组的相应元素。yii\base\Component::off() 如下所示:

```
public function off($name, $handler = null)
{
    $this->ensureBehaviors();
    if (empty($this->_events[$name])) {
        return false;
    }
    if ($handler === null) {                                    // $handler === null 时解除所有的handl
        unset($this->_events[$name]);
        return true;
    } else {
        $removed = false;
        foreach ($this->_events[$name] as $i => $event) {       // 遍历所有的 $handler
            if ($event[0] === $handler) {
                unset($this->_events[$name][$i]);
                $removed = true;
            }
        }
        if ($removed) {
            $this->_events[$name] = array_values($this->_events[$name]);
        }
        return $removed;
    }
}
```

要留意以下几点： - 当 $handler 为 null 时，表示解除 $name 事件的所有handler。 - 在解除 $handler 时，将会解除所有的这个事件下的 $handler 。虽然一个handler多次绑定在同一事件上的情况不多见，但这并不是没有，也不是没有意义的事情。在特定的情况下，确实有一个handler多次绑定在同一事件上。因此在解除时，所有的 $handler 都会被解除。而且没有办法只解除其中的一两个。

2.2.4 事件的触发

事件的处理程序handler有了，事件与事件handler关联好了，那么只要事件触发了，handler就会按照设计的路子走。事件的触发，需要调用 yii\base\Component::trigger()

```
public function trigger($name, Event $event = null)
{
    $this->ensureBehaviors();
    if (!empty($this->_events[$name])) {
        if ($event === null) {
            $event = new Event;
        }
        if ($event->sender === null) {
            $event->sender = $this;
        }
        $event->handled = false;
        $event->name = $name;
        foreach ($this->_events[$name] as $handler) {       // 遍历handler数组，并依次调用
            $event->data = $handler[1];
            call_user_func($handler[0], $event);            // 使用PHP的call_user_func调用handler
            if ($event->handled) {                          // 如果在某一handler中，将$evnet->handled 设
                return;
            }
        }
    }
    Event::trigger($this, $name, $event);                   // 触发类一级的事件
}
```

以 yii\base\Application 为例，他定义了两个事件， EVENT_BEFORE_REQUEST EVENT_AFTER_REQUEST 分别在处理请求的前后触发:

```
abstract class Application extends Module
{
    // 定义了两个事件
    const EVENT_BEFORE_REQUEST = 'beforeRequest';
    const EVENT_AFTER_REQUEST = 'afterRequest';

    public function run()
    {
        try {

            $this->state = self::STATE_BEFORE_REQUEST;
            $this->trigger(self::EVENT_BEFORE_REQUEST);                // 先触发EVENT_BEFORE_REQUEST

            $this->state = self::STATE_HANDLING_REQUEST;
            $response = $this->handleRequest($this->getRequest());     // 处理Request

            $this->state = self::STATE_AFTER_REQUEST;
            $this->trigger(self::EVENT_AFTER_REQUEST);                 // 处理完毕后触发EVENT_AFTER_RE

            $this->state = self::STATE_SENDING_RESPONSE;
            $response->send();

            $this->state = self::STATE_END;

            return $response->exitStatus;

        } catch (ExitException $e) {

            $this->end($e->statusCode, isset($response) ? $response : null);
            return $e->statusCode;

        }
    }
}
```

上面的代码，不用全部去读懂。只要注意是怎么定义事件，怎么触发事件的就可以了。

对于事件的定义，提倡使用const 常量的形式，可以避免写错。 trigger('Hello') 和 trigger('hello') 可是不同的事件哦。原因在于handler数组下标，就是事件名。而PHP里数组下标是区分大小写的。所以，用类常量的方式，可以避免这种头疼的问题。

在触发事件时，可以把与事件相关的数据传递给所有的handler。比如，发布新微博事件:

```
// 定义事件的关联数据
class MsgEvent extend yii\base\Event
{
    public $dateTime;    // 微博发出的时间
    public $author;      // 微博的作者
    public $content;     // 微博的内容
}

// 在发布新的微博时，准备好要传递给handler的数据
$event = new MsgEvent;
$event->title = $title;
$event->author = $auhtor;

// 触发事件
$msg->trigger(Msg::EVENT_NEW_MESSAGE, $event);
```

注意这里数据的传入，与使用 on() 绑定handler时传入数据方法的不同。在 on() 中，使用一个简单变量，传入，并在handler中通过 $event->data 进行访问。这个是在绑定时确定的数据。而有的数

据是没办法在绑定时确定的，如发出微博的时间。这个时候，就需要在触发事件时提供其他的数据了。也就是上面这段代码使用的方法了。这两种方法，一种用于提供绑定时的相关数据，一种用于提供事件触发时的数据，各有所长，互相补充。你可要一碗水端平，不要厚此薄彼了。

2.2.5 多个事件handler的顺序

使用 `yii\base\Component::on()` 可以为各种事件绑定handler，也可以为同一事件绑定多个handler。假如，你是微博系统的技术人员，刚开始的时候，你批定新发微博的事件handler就是通知关注者有新的内容发布了。现在，你不光要保留这个功能，你还要通知微博中@到的所有人。这个时候，一种做法是直接在原来的handler末尾加上新的代码，以处理这个新的需要。另一个方法，就是再写一个handler，并绑定到这个事件上。从易于维护的角度来讲，第二种方法是比较合理的。前一种方法由于修改了原来正常使用的代码，可能会影响原来的正常功能。同时，如果一直有新的需求，那么很快这个handler就会变得很杂，很大。所以，建议使用第二种方法。

Yii中是支持这种一对多的绑定的。那么，在一个事件触发时，哪个handler会被先执行呢？各handler之间总有一个先后问题吧。这个可能不同的编程语言、不同的框架有不同的实现方式。有的语言是以堆栈的形式来保存handler，可能会以后绑定上去的事件先执行的方式运作。这种方式的好处是编码的人权限大些，可以对事件进行更改、拦截、中止，移花接木、偷天换日、无中生有，各种欺骗后面的handler。而Yii是使用数组来保存handler的，并按顺序执行这些handler。这意味着一般框架上预设的handler会先执行。但是不要以为Yii的事件handler就没办法偷天换日了，要使后加上的事件handler先运行，只需在调用 `yii\base\Component::on()` 进行绑定时，将第四个参数设为 `$append` 设为 `false` 那么这个handler就会被放在数组的最前面了，它就会被最先执行，它也就有可能欺骗后面的handler了。

为了加强安全生产，国家安监局对全国煤矿进行监管，一旦发生矿难，他们会收到报警，这就是一个事件和一个handler:

```
$coal->on(Coal::EVENT_DISASTER, [$government, 'onDisaster']);

class Government extend yii\base\Component
{
    ... ...

    public function onDisaster($event)
    {
        echo 'DISASTER! from ' . $event->sender;
    }
}
```

由于煤矿自身也要进行管理，所以，政府允许煤矿可以编写自己的handler对矿难进行处理。但是，有个小煤窑的老板，你有张良计，我有过墙梯，对于发生矿难这一事件编写了一个handler专门用于瞒报:

```
// 第四个参数设为false，使得该handler在整个handler数组中处于第一个
$coal->on(Coal::EVENT_DISASTER, [$baddy, 'onDisaster'], null, false);

calss Baddy extend yii\base\Component
{
    ... ...

    public function onDisaster($event)
    {
        $event->handled = true;          // 将事件标准为已经处理完毕，阻止后续事件handler介入。
    }
}
```

坏人不可怕，会编程的坏人才可怕。我们要阻止他，所以要把绑定好的handler解除。这个解除是绑定的逆向过程，在实质上，就是把对应的handler从handler数组中删除。使用 `yii\base\Component::off()` 就能删除:

```
// 删除所有EVENT_DISASTER事件的handler
$coal->off(Coal::EVENT_DISASTER);
```

```
3
4  // 删除一个PHP全局函数的handler
5  $coal->off(Coal::EVENT_DISASTER, 'global_onDisaster');
6
7  // 删除一个对象的成员函数的handler
8  $coal->off(Coal::EVENT_DISASTER, [$baddy, 'onDisaster']);
9
10 // 删除一个类的静态成员函数的handler
11 $coal->off(Coal::EVENT_DISASTER, ['path\to\Baddy', 'static_onDisaster']);
12
13 // 删除一个匿名函数的handler
14 $coal->off(Coal::EVENT_DISASTER, $anonymousFunction);
```

其中，第三种方法就可以把小煤窑老板的handler解除下来。

细心的读者朋友可能留意到，在删除匿名函数handler时，需要使用一个变量。请读者朋友留意，就算你调用 `yii\base\Component::on()` `yii\base\Component::off()` 时，写了两个一模一样的匿名函数，你也没办法把你前面的匿名handler解除。从本质上来讲，两个匿名函数就是两个不同的存在，为了能够正确解除，需要先把匿名handler保存成一个变量，如上面的 `$anonymousFunction` ，然后再依次绑定、解除。但是，使用了变量后，就失去了匿名函数的一大心理上的优势，你本不用去关心他的，我的建议是在这种情况下，就不要使用匿名函数了。因此，在作为handler时，要慎重使用匿名函数。只有在确定不需要解除时，才可以使用。

2.2.6 事件的级别

前面的事件，都是针对类的实例而言的，也就是事件的触发、处理全部都在实例范围内。这种级别的事件用情专一，不与类的其他实例发生关系，也不与其他类、其他实例发生关系。除了实例级别的事件外，还有类级别的事件。对于Yii，由于Application是一个单例，所有的代码都可以访问这个单例。因此，有一个特殊级别的事件，全局事件。但是，本质上，他只是一个实例级别的事件。

这就好比是公司里的不同阶层。底层的码农们只能自己发发牢骚，个人的喜怒哀乐只发生在自己身上，影响不了其他人。而团队负责人如果心情不好，整个团队的所有成员今天都要战战兢兢，慎言慎行。到了公司老总那里，他今天不爽，哪个不长眼的敢上去触霉头？事件也是这样的，不同层次的事件，决定了他影响到的范围。

类级别事件

先讲讲类级别的事件。类级别事件用于响应所有类实例的事件。比如，工头需要了解所有工人的下班时间，那么，他就绑定一个handler到Worker类，这样每个工人下班时，他都能知道了。与实例级别的事件不同，类级别事件的绑定需要使用 `yii\base\Event::on()`

```
1  Event::on(
2      Worker::className(),                    // 第一个参数表示事件发生的类
3      Worker::EVENT_OFF_DUTY,                 // 第二个参数表示是什么事件
4      function ($event) {                     // 对事件的处理
5          echo $event->sender . ' 下班了';
6      }
7  );
```

这样，每个工人下班时，会触发自己的事件处理函数，比如去打卡。之后，会触发类级别事件。类级别事件的触发仍然是在 `yii\base\Component::trigger()` 中，还记得该函数的最后一个语句么:

```
Event::trigger($this, $name, $event);                    // 触发类一级的事件
```

这个语句就触发了类级别的事件。类级别事件，总是在实例事件后触发。既然触发时机靠后，那么如果有一天你要早退又不想老板知道，你会不会向小煤窑老板那样，通过 `$event->handled = true` ，来终止事件处理？如果是，说明少年你还是图样图森破啊。无论是 `yii\base\Component::trigger()` 还是 `yii\base\Event::trigger()` 中，都对于 `$trigger` 有相同的预处理:

```php
if ($event->sender === null) {
    $event->sender = $this;
}
$event->handled = false;        // 重置了handled
$event->name = $name;           // 重置了name
```

由于上面的代码重置了 $event 的部分内容，所以，你的天真被无懈打败了。因此，得出一个重要结论： $event->handled = true 只能终止同一级别事件的处理，没办法跨事件级别进行干预。所以，正确的做法是通过绑定一个类级别事件的handler来中止。

这其实是很正常的事情，你具有一定的权限，如，可以在某一级别的事件上绑定事件，那么你就可以随意终止他。如果你只有低级别的权限，你当然就不能终止高级别事件的处理了。

所以，政府部门与小煤窑老板的斗争，可以终止了，只要政府部门不在实例上进行绑定，而对于类级别事件进行绑定，小煤窑老就没有办法瞒报了。

从 yii\base\Event::trigger() 的参数列表来看，比 yii\base\Component::trigger() 多了一个参数 $class 表示这是哪个类的事件。因此，在保存 $_event[] 数组上，yii\base\Event 也比 yii\base\Component 要多一个维度:

```php
// Component中的$_event[] 数组
$_event[$eventName][] = [$handler, $data];

// Event中的$_event[] 数组
$_event[$calssName][$eventName][] = [$handler, $data];
```

那么，反过来的话，低级别的handler可以在高级别事件发生时发生作用么？这当然也是不行的。由于类级别事件不与任意的实例相关联，所以，类级别事件触发时，类的实例可能都还没有呢，怎么可能进行处理呢?

类级别事件的触发，应使用 yii\base\Event::trigger() 。这个函数不会触发实例级别的事件。值得注意的是， $event->sender 在实例级别事件中， $event->sender 指向触发事件的实例，而在类级别事件中， 指向的时类名。在 yii\base\Event::trigger() 代码中，有:

```php
if (is_object($class)) {                    // $class 是trigger()的第一个参数，表示类名
    if ($event->sender === null) {
        $event->sender = $class;
    }
    $class = get_class($class);             // 传入的是一个实例，则以类名替换之
} else {
    $class = ltrim($class, '\\');
}
```

这段代码会对 $evnet->sender 进行设置，如果传入的时候，已经指定了他的值，那么这个值会保留，否则，就会替换成类名。

对于类级别事件，有一个要格外注意的地方，就是他不光会触发自身这个类的事件，这个类的所有祖先类的同一事件也会被触发。但是，自身类事件与所有祖先类的事件，视为同一级别:

```php
// 最外面的循环遍历所有祖先类
do {
    if (!empty(self::$_events[$name][$class])) {
        foreach (self::$_events[$name][$class] as $handler) {
            $event->data = $handler[1];
            call_user_func($handler[0], $event);

            // 所有的事件都是同一级别，可以随时终止
            if ($event->handled) {
                return;
            }
        }
    }
} while (($class = get_parent_class($class)) !== false);
```

上面的嵌套循环的深度，或者叫时间复杂度，受两个方面影响，一是类继承结构的深度，二是`$_event[$name][$class][]`数组的元素个数，即已经绑定的handler的数量。从实践经验看，一般软件工程继承深度超过十层的就很少见，而事件绑定上，同一事件的绑定handler超过十几个也比较少见。因此，上面的嵌套循环运算数量级大约在100～1000之间，这是可以接受的。

但是，从机制上来讲，由于类级别事件会被类自身、类的实例、后代类、后代类实例所触发，所以，对于越底层的类而言，其类事件的影响范围就越大。因此，在使用类事件上要注意，尽可能往后代类安排，以控制好影响范围，尽可能不在基础类上安排类事件。

全局事件

接下来再讲讲全局级别事件。上面提到过，所谓的全局事件，本质上只是一个实例事件罢了。他只是利用了Application实例在整个应用的生命周期中全局可访问的特性，来实现这个全局事件的。当然，你也可以将他绑定在任意全局可访问的的Component上。

全局事件一个最大优势在于：在任意需要的时候，都可以触发全局事件，也可以在任意必要的时候绑定，或解除一个事件:

```
Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);          // 显示当前触发事件的对象的类名称
});

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

上面的`Yii::$app->on()`可以在任何地方调用，就可以完成事件的绑定。而`Yii::$app->trigger()`只要在绑定之后的任何时候调用就OK了。

2.3 行为（Behavior）

使用行为（behavior）可以在不修改现有类的情况下，对类的功能进行扩充。通过将行为绑定到一个类，可以使类具有行为本身所定义的属性和方法，就好像类本来就有这些属性和方法一样。而且不需要写一个新的类去继承或包含现有类。

Yii中的行为，其实是`yii\base\Behavior`类的实例，只要将一个behavior实例绑定到任意的`yii\base\Component`实例上，这个Component就可以拥有该behavior所定义的属性和方法了。而如果将行为与事件关联起来，可以玩的花样就更多了。

但有一点需要注意，behavior只能与Component类绑定，他们是天生的一对，爱情不是你想买，想买就能买的，必要的物质是少不了的，奋斗吧少年。所以，如果你写了一个类，需要使用到行为，那么就果断地继承自`yii\base\Component`。

另外一点，爱情也是双方你情我愿的。行为也是需要双方共同支持的，Yii对于行为的支持，除了靠`yii\base\Behavior`之外，`yii\base\Component`也是经过了精心设计。

2.3.1 使用行为

一个绑定了行为的类，表现起来是这样的:

```
// Step 1: 定义一个将绑定行为的类
class MyClass extends yii\base\Component
{
    // 空的
}

// Step 2: 定义一个行为类，他将绑定到MyClass上
class MyBehavior extends yii\base\Behavior
{
    // 行为的一个属性
    public $property1 = 'This is property in MyBehavior.';
```

```php

    // 行为的一个方法
    public function method1()
    {
        return 'Method in MyBehavior is called.';
    }
}

$myClass = new MyClass();
$myBehavior = new MyBehavior();

// Step 3: 将行为绑定到类上
$myClass->attachBehavior('myBehavior', $myBehavior);

// Step 4: 访问行为中的属性和方法，就和访问类自身的属性和方法一样
echo $myClass->property1;
echo $myClass->method1();
```

上面的代码你不用全都看懂，虽然你可能已经用脚趾头猜到了这些代码的意思，但这里你只需要记住行为中的属性和方法可以被所绑定的类像访问自身的属性和方法一样直接访问就OK了。代码中，`$myClass` 是没有 `property1` `method()` 成员的。这俩是 `$myBehavior` 的成员。但是，通过 `attachBehavior()` 将行为绑定到对象之后，`$myCalss` 就好像练成了吸星大法、化功大法，现表的财大气粗，将别人的属性和方法都变成了自己的。

另外，从上面的代码中，你还要掌握使用行为的大致流程：

- 从 `yii\base\Component` 派生自己的类，以便使用行为；
- 从 `yii\base\Behavior` 派生自己的行为类，里面定义行为涉及到的属性、方法；
- 将Component和Behavior绑定起来；
- 像使用Component自身的属性和方法一样，尽情使用行为中定义的属性和方法。

2.3.2 Behavior类的要素

我们提到了行为只是 `yii\base\Behavior` 类的实例。那么这个类究竟有什么秘密呢？其实说破了也没有什么的他只是一个简单的封装而已，非常的简单：

```php
class Behavior extends Object
{
    // 指向行为本身所绑定的Component对象
    public $owner;

    // Behavior 基类本身没用，主要是自类使用，返回一个数组表示行为所关联的事件
    public function events()
    {
        return [];
    }

    // 绑定行为到 $owner
    public function attach($owner)
    {
        $this->owner = $owner;
        foreach ($this->events() as $event => $handler) {
            $owner->on($event, is_string($handler) ? [$this, $handler] : $handler);
        }
    }

    // 解除绑定
    public function detach()
    {
        if ($this->owner) {
```

```
            foreach ($this->events() as $event => $handler) {
                $this->owner->off($event, is_string($handler) ? [$this, $handler] : $handler);
            }
            $this->owner = null;
        }
    }
}
```

这就是Behavior的全部了代码，是不是很简单？Behavior类的要素的确很简单：

- `$owner` 成员变量，用于指向行为的依附对象；
- `events()` 用于表示行为所有响应的事件；
- `attach()` 用于将行为与Component绑定起来；
- `deatch()` 用于将行为从Component上解除。

下面分别进行讲解。

2.3.3 定义一个行为

定义一个行为，就是准备好要注入到现有类中去的属性和方法，这些属性和方法要写到一个 `yii\base\Behavior` 类中。所以，定义一个行为，就是写一个 Behavior的子类，子类中包含了所要注入的属性和方法:

```
namespace app\Components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;
    private $_prop3;
    private $_prop4

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

    public function setProp3($value)
    {
        $this->_prop2 = $value;
    }

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

    protected function bar()
    {
        // ...
    }
}
```

上面的代码通过定义一个 `app\Components\MyBehavior` 类。由于 MyBehavior 继承自 `yii\base\Behavior` 从而间接地继承自 `yii\base\Object` 。没错，这是我们的老朋友了。因此，这个类有一个public的成员变量 `prop1` 一个只读属性 `prop2` 一个只写属性 `prop3` 一个public的方

法 foo()，同时，有一个private 的成员变量 $_prop4 一个protected 的方法 bar()。如果你不清楚只读属性和只写属性，最好回头看看 属性 部分的内容。

当这MyBehavior与一个Component绑定后，绑定的Component也就拥有了 prop1 prop2 这两个属性和方法 foo()。

2.3.4 行为的依附对象

yii\base\Behavior::$owner 指向的是Behavior实例本身所依附的对象。这是行为中引用所依附对象的唯一手段了。通过这个 $owner 行为才能访问所依附的Component，才能将本身的方法作为事件handler绑定到Component上。

$owner 由 yii\base\Behavior::attach() 进行赋值，也就是在将行为绑定到某个Component时，$owner 就已经名花有主了。一般情况下，不需要你自己手动去指定 $owner 的值，在调用 yii\base\Componet::attachBehavior() 将行为与对象绑定时，Component会自动地将 $this 作为参数，调用 yii\base\Behavior::attach()。

有一点需要格外注意，由于行为从本质来讲是一个PHP类，其方法就是类方法，就是成员函数。所以，在行为的方法中，$this 引用的是行为本身，试图通过 $this 来访问行为所依附的Component是行不通的。正确的方法是通过 yii\base\Behavior::$owner。

2.3.5 行为的绑定

行为的绑定是通常由Component来发起。有两种方式可以将一个Behavior绑定到一个 yii\base\Component。一种是静态的方法，另一种是动态的。静态的方法在实践中用得比较多一些。因为一般情况下，在你的代码没跑起来之前，一个类应当具有何种行为，是确定的。动态绑定的方法主要是提供了更灵活的方式，但实际使用中并不多见。

静态绑定行为，只需要重载 yii\base\Component::behaviors() 就可以了。这个方法用于描述类所具有的行为。如何描述呢？使用配置来描述，可以是Behavior类名，也可以是Behavior类的配置数组:

```
namespace app\models;

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

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // 匿名的行为，仅直接给出行为的类名称
            MyBehavior::className(),

            // 名为myBehavior2的行为，也是仅给出行为的类名称
            'myBehavior2' => MyBehavior::className(),

            // 匿名行为，给出了MyBehavior类的配置数组
            [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop3' => 'value3',
            ],

            // 名为myBehavior4的行为，也是给出了MyBehavior类的配置数组
            'myBehavior4' => [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop3' => 'value3',
            ]
```

```
        ];
    }
}
```

在上面的例子中，以数组的键作为行为的命名，而对于没有提供键名的行为，就是匿名行为。上面的例子有两个匿名行为，两个命名行为 myBehavior2 myBehaviro4 。

对于命名的行为，可以调用 yii\base\Component::getBehavior() 来取得这个绑定好的行为:

```
$behavior = $Component->getBehavior('myBehavior2');
```

对于匿名的行为，则没有办法直接引用了。但是，可以获取所有的绑定好的行为:

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

至于动态为绑定行为，需要调用 yii\base\Compoent::attachBehaviors():

```
$Component->attachBehaviors([
    'myBehavior1' => new MyBehavior,  // 这是一个命名行为
    MyBehavior::className(),          // 这是一个匿名行为
]);
```

这个方法接受一个数组参数，参数的含义与上面静态绑定行为是一样一样的。

还有一个办法，就是通过配置文件来绑定:

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

    'as myBehavior3' => [
        'class' => MyBehavior::className(),
        'prop1' => 'value1',
        'prop3' => 'value3',
    ],
]
```

具体参见配置部分的内容。

2.3.6 绑定的内部原理

只是重载一个 yii\base\Component::behaviors() 就可以这么神奇地使用行为了? 这只是冰山的一角，实际上关系到绑定的过程，有关的方面有:

- yii\base\Component::behaviors()
- yii\base\Component::ensureBehaviors()
- yii\base\Component::attachBehaviorInternal()
- yii\base\Behavior::attach()

yii\base\Component::behaviors() 上面提到了，就是返回一个数组用于描述行为。那么 yii\base\Component::ensuerBehaviors() 呢? 这个方法会在Component的诸多地方调用 __get() __set() __isset() __unset() __call() canGetProperty() hasMethod() hasEventHandlers() on() off() 等用到，看到这么多是不是头疼? 一点都不复杂，一句话，只要涉及到类的属性、方法、事件这个函数都会被调用到。

这么众星拱月，被诸多凡人所需要的 ensureBehaviors() 究竟是何许人也? 他其实只是将 behaviors() 中所描述的行为进行绑定而已:

```
public function ensureBehaviors()
{
    if ($this->_behaviors === null) {
        $this->_behaviors = [];
```

```
5
6           // 遍历 $this->behaviors() 返回的数组，并绑定
7           foreach ($this->behaviors() as $name => $behavior) {
8               $this->attachBehaviorInternal($name, $behavior);
9           }
10      }
11  }
```

从上面的代码中，自然就看到了接下来要说的第三个东东，`yii\base\Component\attachBehaviorInternal`

```
1   private function attachBehaviorInternal($name, $behavior)
2   {
3       if (!($behavior instanceof Behavior)) {
4           $behavior = Yii::createObject($behavior);
5       }
6       if (is_int($name)) {
7           $behavior->attach($this);
8           $this->_behaviors[] = $behavior;
9       } else {
10          if (isset($this->_behaviors[$name])) {
11              $this->_behaviors[$name]->detach();
12          }
13          $behavior->attach($this);
14          $this->_behaviors[$name] = $behavior;
15      }
16      return $behavior;
17  }
```

首先要注意到，这是一个private成员。其实在Yii中，所有后缀为 `*Internal` 的方法，都是私有的。这个方法干了这么几件事：

- 如果 `$behavior` 参数并非是一个 Behavior 实例，就以之为参数，用 `Yii::createObject()` 创建出来。
- 如果以匿名行为的形式绑定行为，那么直接将行为附加在这个类上
- 如果是命名行为，先看看是否有同名的行为已经绑定在这个类上，如果有，用后来的行为取代之前的行为。

在 `yii\base\Component::attachBehaviorInternal()` 中，以 `$this` 为参数调用了 `yii\base\Behavior::attach()` 。从而，引出了跟绑定相关的最后一个家伙 `yii\base\Behavior::attach()`

```
1   public function attach($owner)
2   {
3       $this->owner = $owner;
4       foreach ($this->events() as $event => $handler) {
5           $owner->on($event, is_string($handler) ? [$this, $handler] : $handler);
6       }
7   }
```

上面的代码干了两件事：

- 设置好行为的 `$owner` ，使得行为可以访问、操作所依附的对象
- 遍历行为中的 `events()` 返回的数组，将准备响应的事件，通过所依附类的 `on()` 绑定到类上

说了这么多，关于绑定，做个小结：

- 绑定的动作从Component发起；
- 静态绑定通过重载 `yii\base\Componet::behaviors()` 实现；
- 动态绑定通过调用 `yii\base\Component::attachBehaviors()` 实现；
- 行为还可以通过为 Component 配置 `as` 配置项进行绑定；

- 行为有匿名行为和命名行为之分，区别在于绑定时是否给出命名。命名行为可以通过其命名进行标识，从而有针对性地进行解除等操作；
- 绑定过程中，后绑定的行为会取代已经绑定的同名行为；
- 绑定的意义有两点，一是为行为设置 `$owner` 。二是将行为中拟响应的事件的handler绑定到类中去。

2.3.7 解除行为

解除行为只需调用 `yii\base\Component::detachBehavior()` 就OK了:

```
$Component->detachBehavior('myBehavior2');
```

这样就可以解除已经绑定好的名为 `myBehavior2` 的行为了。但是，对于匿名行为，这个方法就无从下手了。但是，可以解除所有绑定好的行为:

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

这上面两种方法，都会调用到 `yii\base\Behavior::detach()` 。与 `yii\base\Behavior::attach()` 相反，解除的过程就是干两件事：一是将 `$owner` 设置为 `null` ，表示这个行为没有依附到任何类上。二是通过Component的 `off()` 将绑定到类上的事件hanlder解除下来。一句话，善始善终。

2.3.8 用行为响应事件

行为与事件结合后，可以在不对类作修改的情况下，补充类在事件触发后的各种不同反应。只需要重载 `yii\base\Behavior::events()` 方法，表示这个行为将对类的何种事件进行何种反馈即可:

```
1  namespace app\Components;
2
3  use yii\db\ActiveRecord;
4  use yii\base\Behavior;
5
6  class MyBehavior extends Behavior
7  {
8      // ...
9
10     // 重载events() 使得在事件触发时，调用行为中的一些方法
11     public function events()
12     {
13         return [
14             // 在EVENT_BEFORE_VALIDATE事件触发时，调用成员函数 beforeValidate
15             ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
16         ];
17     }
18
19     // 注意beforeValidate 是行为的成员函数，而不是绑定的类的成员函数。
20     // 还要注意，这个函数的签名，要满足事件handler的要求。
21     public function beforeValidate($event)
22     {
23         // ...
24     }
25 }
```

上面的代码中， `events()` 返回一个数组，表示所要做好反应的事件，上例中的事件是 `ActiveRecord::EVENT_BEFORE_VALIDATE` ，以数组的键来表示，而数组的值则表示做好反应的代码，上例中是 `beforeValidate()` ，可以是以下形式:

- 字符串，表示行为类的方法，如上面的例就是这种情况。这个是与事件handler不同的，事件handler中使用字符串是表示PHP全局函数。

- 一个对象或类的成员函数，以数组的形式，如 `[$object, 'methodName']` 。这个与事件handler是一致的。
- 一个匿名函数。

对于反应函数的签名，要求与事件handler一样:

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

具体内容，请参考 事件（*Event*） 的内容。

2.3.9 行为响应的事件原理

上面的绑定和解除过程，我们看到Yii费了那么大劲，主要就是为了将行为中的事件handler绑定到类中去。在实际编程时，行为用得最多的，也是对于Compoent各种事件的响应。通过行为注入，可以在不修改现有类的代码的情况下，更改、扩展类对于事件的响应和支持，使用这个技巧，可以玩出很炫的花样。而要将行为与Component的事件关联起来，就要通过 `yii\base\events()` 方法。上面的代码中，这个方法只是返回了一个空数组，说明不对所依附的Compoent的任何事件产生关联。但是在实际使用时，往往通过重载这个方法来告诉Yii，这个行为将对Compoent的何种事件，使用哪个方法进行处理。

比如，Yii自带的 `yii\behaviors\AttributeBehavior` 类，定义了在一个 ActiveRecord 对象的某些事件发生时，自动对某些字段进行修改的行为。他有一个很常用的子类 `yii\behaviors\TimeStampBehavior` 用于将指定的字段设置为一个当前的时间戳，常用于表示最后修改日期、上次登陆时间等场景。其中， `yii\behaviors\AttributeBehavior::event()` 如下:

```
public function events()
{
    return array_fill_keys(array_keys($this->attributes), 'evaluateAttributes');
}
```

这段代码的意思这里不作过多深入，学有余力的读者朋友可以自行研究，难度并不高。这里，你只需要大致知道，这段代码将返回一个数组，其键值为 `$this->attributes` 数组的键值，其值为成员函数 `evaluateAttributes` 。而在 `yii\behaviors\TimeStampBehavior::init()` 中，有以下的代码:

```
1  public function init()
2  {
3      parent::init();
4
5      if (empty($this->attributes)) {
6          // 重点看这里
7          $this->attributes = [
8              BaseActiveRecord::EVENT_BEFORE_INSERT => [$this->createdAtAttribute, $this->updatedAt
9              BaseActiveRecord::EVENT_BEFORE_UPDATE => $this->updatedAtAttribute,
10         ];
11     }
12 }
```

上面的代码重点看的是对于 `$this->attributes` 的初始化部分。结合上面两段代码，对于 `yii\base\Behavior::events()` 的返回数组，其格式应该是这样的:

- 数组的键值用于指定要响应的事件，如 `BaseActiveRecord::EVENT_BEFORE_INSERT` 等
- 数组的值是一个事件handler，如上面的 `evaluateAttributes` ，至于对事件handler的要求，请看看 事件（*Event*） 部分的内容。

2.3.10 行为的属性和方法注入原理

上面我们了解到了行为的用意在于将自身的属性和方法注入给所依附的类，那么Yii中是如何将一

个行为 yii\base\Behavior 的属性和方法注入到一个 yii\base\Component 中的呢? 对于属性而言，是通过 __get() 和 __set() 魔术方法来实现的，对于方法，是通过 __call() 方法。

属性的注入

以读取为例，如果访问 $Component->property1 ，Yii在幕后干了些什么呢？这个看看 yii\base\Component::__get()

```
public function __get($name)
{
    $getter = 'get' . $name;
    if (method_exists($this, $getter)) {
        return $this->$getter();
    } else {
        // 注意这个 else 分支的内容，正是与 yii\base\Object::__get() 的不同之处
        $this->ensureBehaviors();
        foreach ($this->_behaviors as $behavior) {
            if ($behavior->canGetProperty($name)) {
                return $behavior->$name;
            }
        }
    }
    if (method_exists($this, 'set' . $name)) {
        throw new InvalidCallException('Getting write-only property: ' . get_class($this) . '::'
    } else {
        throw new UnknownPropertyException('Getting unknown property: ' . get_class($this) . '::'
    }
}
```

重点来看 yii\base\Compoent::__get() 与 yii\base\Object::__get() 的不同之处，就是在于对于未定义getter函数之后的处理， yii\base\Object 是直接抛出异常，告诉你想要访问的属性不存在之类。但是 yii\base\Component 则是在不存在getter之后，还要看看是不是注入的行为的属性。

- 首先，调用了 $this->ensureBehaviors() 。这个主要是对子类用的，yii\base\Compoent 没有任何预先注入的行为，所以，这个调用没有用。但是对于子类，你可能重载了 yii\base\Compoent::behaviros() 来预先注入一些行为，那么，这个函数会将这些行为先注入进来。
- 然后，开始遍历 $this->_behaviors 。Yii将类所有绑定的行为都保存在 yii\base\Compoent::$_behaviors[] 数组中。
- 最后，判断这个属性，是否是所绑定行为的可读属性，如果是，就返回这个行为的这个属性 $behavior->name 。

对于setter，代码类似，这里就不占用篇幅了。

方法的注入

与属性的注入通过 __get() __set() 魔术方法类似，Yii通过 __call() 魔术方法实现对行为中方法的注入:

```
public function __call($name, $params)
{
    $this->ensureBehaviors();
    foreach ($this->_behaviors as $object) {
        if ($object->hasMethod($name)) {
            return call_user_func_array([$object, $name], $params);
        }
    }
    throw new UnknownMethodException('Calling unknown method: ' . get_class($this) . "::$name()");
}
```

从上面的代码中可以看出，Yii先是调用了 `$this->ensureBehaviors()` 。这个方法在前面讲过了。也就是说，访问类本身不存在的属性、方法时，这个方法都会被调用的。这点性能上的损失，是为了提供编码上提供便利，是划算的。

然后，也是遍历 `yii\base\Component::$_behaviros[]` 数组，如果所绑定的行为中要调用的方法存在，则使用PHP的 `call_user_func_array()` 调用之。从这里也可以看出，`yii\base\Components::$_behaviros[]` 对于行为的方法具有形式上的要求，或者说对方法的签名有要求。而这个要求，是由 `call_user_func_array` 所决定的。具体可以看看PHP手册。

2.3.11 行为与继承和特性（Traits） 的区别

从实现的效果看，你是不是会认为Yii真是多此一举？PHP中要达到这样的效果，可以使用继承呀，可以使用PHP新引入的特性（Traits）呀。但是，行为具有继承和特性所没有的优点，从实际使用的角度讲，继承和特性更靠底层点。靠底层，就意味着开发效率低，运行效率高。行为的引入，是以可以接受的运行效率牺牲为成本，谋取开发效率大提升的一笔买卖。

行为与继承

首先来讲，拿行为与继承比较，从逻辑上是不对的，这两者是在完全不同的层面上的事物，是不对等的。之所以进行比较，是因为在实现的效果上，两者有的类似的地方。看起来，行为和继承都可以使一个类具有另一个类的属性和方法，从而达到扩充类的功能的目的。

相比较于使用继承的方式来扩充类功能，使用行为的方式，一是不必对现有类进行修改，二是PHP不支持多继承，但是Yii可以绑定多个行为，从而达到类似多继承的效果。

反过来，行为是绝对无法替代继承的。亚洲人，美洲人都是地球人，你可以将亚洲人和美洲人当成地球人来对待。但是，你绝对不能把一只在某些方面表现得像人的猴子，真的当成人来对待。

这里就不展开讲了。从本质上来讲，行为只是一种设计模式，是解决问题的方法学。继承则是PHP作为编程语言所提供的特性，根本不在一个层次上。

行为与特性

特性是PHP5.4之后引入的一个新feature。从实现效果看，行为与特性都达到把自身的public 变量、属性、方法注入到当前类中去的目的。在使用上，他们也各有所长，但总的原则可以按下面的提示进行把握。

倾向于使用行为的情况:

- 行为从本质上讲，也是PHP的类，因此一个行为可以继承自另一个行为，从而实现代码的复用。而特性只是PHP的一种语法，效果上类似于把特性的代码导入到了类中从而实现代码的注入，特性是不支持继承的。
- 行为可以动态地绑定、解除，而不必要对类进行修改。但是特性必须在类在使用 `use` 语句，要解除特性时，则要删除这个语句。换句话说，需要对类进行修改。
- 行为还以在在配置阶段进行绑定，特性就不行了。
- 行为可以用于对事件进行反馈，而特性不行。
- 当出现命名冲突时，行为会自行排除冲突，自动使用先绑定的行为。而特性在发生冲突时，需要人为干预，修改发生冲突的变量名、属性名、方法名。

倾向于使用特性的情况:

- 特性比行为在效率上要高一点，因为行为其实是类的实例，需要时间和空间进行分配。
- 特性是PHP的语法，因此，IDE的支持要好一些。目前还没有IDE能支持行为。

CHAPTER

THREE

YII 约定

这一部份主要讲的是Yii中以约定的方式来实现的功能，或者说是惯用的模式。最常见的约定莫过于默认值了。 Yii通过约定一些最最通用的内容，使得这部分内容在编程的过程中，你不必再花费精力去指定或编码。 这也是提高效率的一种方式。

当然，既然称之为约定，就说明仅是推荐性、建议性的，而并非是强制性。也就是说，你是可以更改这些约定的内容的。 但是，除非有绝对的理由，否则，不建议随意更改Yii设定的约定。 而且，一旦对约定内容有所更改，一定要在代码中进行说明。

Yii的约定内容，主要包含应用的别名、对象配置、目录结构、自动加载等内容:

3.1 别名

可以将别名视为特殊的常量变量，他的作用在于避免将一些文件路径、URL以硬编码的方式写入代码中，或者多处出现一长串的文件路径、URL。

3.1.1 预定义的别名

Yii中，别名以 @ 开头，以区别于正常的文件路径和URL。Yii中预定义了许多常用的别名，从入口脚本文件 index.php 中不难看出，别名主要是放在 aliases.php 文件中:

```
<?php
defined('YII_DEBUG') or define('YII_DEBUG', false);
defined('YII_ENV') or define('YII_ENV', 'prod');

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

// 重点看这里
require(__DIR__ . '/../../common/config/aliases.php');

$config = yii\helpers\ArrayHelper::merge(
    require(__DIR__ . '/../../common/config/main.php'),
    require(__DIR__ . '/../../common/config/main-local.php'),
    require(__DIR__ . '/../config/main.php'),
    require(__DIR__ . '/../config/main-local.php')
);

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

在代码中提到的 aliases.php 文件中，保存了许多预定义和开发者自定义的别名:

```
<?php
Yii::setAlias('common', dirname(__DIR__));
Yii::setAlias('frontend', dirname(dirname(__DIR__)) . '/frontend');
```

```
Yii::setAlias('backend', dirname(dirname(__DIR__)) . '/backend');
Yii::setAlias('console', dirname(dirname(__DIR__)) . '/console');
```

这是直接可以看到的别名，还有的别名你可能没法一下子看到，这个别名直接写到Yii的代码中去了，位于 yii\BaseYii 中:

```
public static $aliases = ['@yii' => __DIR__];
```

yii\BaseYii::$aliases 用于保存所有的别名，默认地，把 yii\BaseYii.php 所在的目录作为 @yii 别名。

另外，对于 yii\base\Application 在其初始化过程中，有以下代码:

```
public function preInit(&$config)
{
    ... ...

    // basePath必须在配置文件中给出，否则会抛出弃常
    if (isset($config['basePath'])) {
        // 这里会设置 @app
        $this->setBasePath($config['basePath']);
        unset($config['basePath']);
    } else {
        throw new InvalidConfigException('The "basePath" configuration for the Application is requ
    }

    // @vendor 如果配置文件中设置了 vendorPath 使用配置的值，否则使用默认的
    if (isset($config['vendorPath'])) {
        $this->setVendorPath($config['vendorPath']);
        unset($config['vendorPath']);
    } else {
        $this->getVendorPath();
    }

    // @runtime 如果配置文件中设置了 runtimePath 使用配置的值，否则使用默认的
    if (isset($config['runtimePath'])) {
        $this->setRuntimePath($config['runtimePath']);
        unset($config['runtimePath']);
    } else {
        $this->getRuntimePath();
    }

    ... ...
}
```

上面的代码中，预定义了3个别名： @app @vendor @runtime 。其中， basePath 不是别名，但必须由开发者自己在配置文件中设定，表示应用的根目录，对于frontend而言，就是目录 path/to/yii/application/frontend 。在定义 basePath 时，Yii顺便定义了 @app ，代码在 yii\base\Application::setBasePath() 中:

```
public function setBasePath($path)
{
    parent::setBasePath($path);
    Yii::setAlias('@app', $this->getBasePath());
}
```

可以看出， @app 与 basePath 是一致的。

在 yii\base\Application 的初始化过程中，如果没在配置文件配置 vendorPath runtimePath ，则Yii会调用 getVendorPath() getRuntimePath() ，这两个方法分别使用默认值定义 @vendor 和 @runtime

```
public function getVendorPath()
{
```

```
    if ($this->_vendorPath === null) {
        $this->setVendorPath($this->getBasePath() . DIRECTORY_SEPARATOR . 'vendor');
    }

    return $this->_vendorPath;
}

public function getRuntimePath()
{
    if ($this->_runtimePath === null) {
        $this->setRuntimePath($this->getBasePath() . DIRECTORY_SEPARATOR . 'runtime');
    }

    return $this->_runtimePath;
}
```

上面的代码中，会分别将 `@vendor` 设为 `@app/vendor` ，将 `@runtime` 设为 `@app/runtime` 。

但是，这里有一个比较特殊的，就是 `@vendor` 对于使用Yii基础模版创建的应用而言，会使用上面提到的 `@app/vendor` ，但是，对于使用高级模版创建的应用，你会发现，vendor目录并不在 frontend 或 backend 目录下，而是跟他们是兄弟目录。这是因为对于整个工程而言，这个vendor的内容是 frontend 和 backend等共用的。因此，实际上 `@vendor` 应该是 `@app/../vendor` 。而这个，Yii也已经考虑到了，在使用高级模板创建应用时，默认的 `config/main.php` 配置文件会设定 `vendorPath` 配置项为 `dirname(dirname(__DIR__)) . '/vendor'` ，这样就避免了使用代码中默认的值。

还有一个含有别名的代码在 `yii\base\Web\Application` 中:

```
protected function bootstrap()
{
    $request = $this->getRequest();
    Yii::setAlias('@webroot', dirname($request->getScriptFile()));
    Yii::setAlias('@web', $request->getBaseUrl());

    parent::bootstrap();
}
```

这里加入了两个新的别名， `@webroot` `@web` 。

最后一个藏有别名的地方，在于Yii的扩展（extensions），当使用Composer安装扩展后，会向 `@vendor/yiisoft/extensions.php` 写入信息，其中就包含相应的别名，只不过这些别名通常都是二级别名。然后，在 `yii\base\Application::bootstrap()` 中，将这些扩展的别名进行注册。

首先看看一个典型的 `extensions.php`

```
<?php

$vendorDir = dirname(__DIR__);

return array (
  'yiisoft/yii2-swiftmailer' =>
  array (
    'name' => 'yiisoft/yii2-swiftmailer',
    'version' => '9999999-dev',
    'alias' =>
    array (
      '@yii/swiftmailer' => $vendorDir . '/yiisoft/yii2-swiftmailer',
    ),
  ),

 ... ...

 'yiisoft/yii2-gii' =>
  array (
```

```
    'name' => 'yiisoft/yii2-gii',
    'version' => '9999999-dev',
    'alias' =>
    array (
      '@yii/gii' => $vendorDir . '/yiisoft/yii2-gii',
    ),
  ),
);
```

注意上面这段代码中的 alias ，这个键对应的就是一个别名及其所代表的实际路径。至于具体对这个 extensions.php 的内容进行处理并注册成别名的工具由 yii\base\Application::bootstrap() 完成:

```
protected function bootstrap()
{
    // 将 extensions.php 的内容读取进 $this->extensions 备用
    if ($this->extensions === null) {
        $file = Yii::getAlias('@vendor/yiisoft/extensions.php');
        $this->extensions = is_file($file) ? include($file) : [];
    }

    // 遍历 $this->extensions 并注册别名
    foreach ($this->extensions as $extension) {
        if (!empty($extension['alias'])) {
            foreach ($extension['alias'] as $name => $path) {
                Yii::setAlias($name, $path);
            }
        }
        ... ...
    }
}
```

小结一下，默认预定义别名一共有10个，其中路径别名9个，URL别名只有 @web 1个：

- @yii 表示Yii框架所在的目录；
- @app 表示正在运行的应用的根目录；
- @vendor 表示第三方库所有目录；
- @runtime 表示正在运行的应用的运行时用于存放运行时文件的目录；
- @webroot 表示正在运行的应用的入口文件 index.php 所在的目录；
- @web URL别名，表示当前应用的根URL；
- @common 表示通用文件夹；
- @frontend 表示前台应用所在的文件夹；
- @backend 表示后台应用所在的文件夹；
- @console 表示命令行应用所在的文件夹；
- 其他使用Composer安装的Yii扩展注册的二级别名。

这样，在整个Yii应用中，只要使用上述别名，就可方便、且统一地表示特定的路径或URL。

3.1.2 定义与解析别名

Yii使用 Yii::$aliases[] 来保存别名，定义别名就是将别名及其代表的实际不路径或URL写入这个数组，而解析别名就是将别名的信息从数组读取出去并组合。

别名的定义过程

除了像上面的代码那样定义一个别名之外，还有其他的用法:

```
// 使用一个路径定义一个路径别名
Yii::setAlias('@foo', 'path/to/foo');

// 使用一个URL定义一个URL别名
Yii::setAlias('@bar', 'http://www.example.com');

// 使用一个别名定义另一个别名
Yii::setAlias('@fooqux', '@foo/qux');

// 定义一个"二级"别名
Yii::setAlias('@foo/bar', 'path/to/foo/bar');
```

从上面的代码中可以了解到， `Yii::setAlias()` 是定义别名的关键。实际上，该方法的代码在 `BaseYii::setAlias()` 中:

```
public static function setAlias($alias, $path)
{
    // 如果拟定义的别名并非以@打头，则在前面加上@
    if (strncmp($alias, '@', 1)) {
        $alias = '@' . $alias;
    }

    // 找到别名的第一段，即@ 到 / 之间的内容，如@foo/bar/qux的@foo
    $pos = strpos($alias, '/');
    $root = $pos === false ? $alias : substr($alias, 0, $pos);

    if ($path !== null) {
        // 去除路径末尾的 \ / 。如果路径本身就是一个别名，直接解析出来
        $path = strncmp($path, '@', 1) ? rtrim($path, '\\/') : static::getAlias($path);

        // 检查是否有 $aliases[$root] ，看看是否已经定义好了根别名。如果没有，则以$root为键，保存这个别名
        if (!isset(static::$aliases[$root])) {
            if ($pos === false) {
                static::$aliases[$root] = $path;
            } else {
                static::$aliases[$root] = [$alias => $path];
            }
        // 如果 $aliases[$root] 已经存在，则替换成新的路径，或增加新的路径
        } elseif (is_string(static::$aliases[$root])) {
            if ($pos === false) {
                static::$aliases[$root] = $path;
            } else {
                static::$aliases[$root] = [
                    $alias => $path,
                    $root => static::$aliases[$root],
                ];
            }
        } else {
            static::$aliases[$root][$alias] = $path;
            krsort(static::$aliases[$root]);
        }

    // 当传入的 $path 为 null 时，表示要删除这个别名。
    } elseif (isset(static::$aliases[$root])) {
        if (is_array(static::$aliases[$root])) {
            unset(static::$aliases[$root][$alias]);
        } elseif ($pos === false) {
            unset(static::$aliases[$root]);
        }
```

```
45         }
46     }
```

对于别名的定义过程:

别名规范化 如果要定义的别名 `$alias` 并非以 `@` 打头，自动为这个别名加上 `@` 前缀。 总之，只要是别名，必然以 `@` 打头。下面的两个语句，都定义了相同的别名 `@foo`

```
Yii::setAlias('foo', 'path/to/foo');

Yii::setAlias('@foo', 'path/to/foo');
```

获取根别名 `$alias` 的根别名，就是 `@` 加上第一个 `/` 之间地内容，以 `$root` 表示。 这里可以看出，别名是分层次的。下面3个语句的根别名都是 `@foo`

```
1  Yii::setAlias('@foo', 'path/to/some/where');
2
3  Yii::setAlias('@foo/bar', 'path/to/some/where');
4
5  Yii::setAlias('@foo/bar/qux', 'path/to/some/where');
```

新定义别名还是删除别名 如果传入的 `$path` 不是 `null` ，说明是正常的别名定义。 对于正常的别名定义，就是往 `BaseYii::$aliases[]` 里写入信息。 而如果 `$path` 为 `null` ，说明是要删除别名:

```
1  // 定义别名@foo
2  Yii::setAlias('@foo', 'path/to/some/where');
3
4  // 删除别名@foo
5  Yii::setAlias('@foo', null);
```

解析 `$path` 对于新定义别名，既然 `$path` 不为 `null` ，那么先进行解析： 如果 `$path` 以 `@` 打头，说明这是也是一个别名，则调用 `Yii::getAlias()` ， 并将解析后的结果作为新的 `$path` ； 如果 `$path` 不以 `@` 打头，说明是一个正常的path 或 URL， 那么去除 `$path` 末尾的 `/` 和 `\` 。

别名的写入 对于全新的别名，也即其根别名是新的， `BaseYii::aliases[$root]` 不存在。 那么全新别名的写入分两种情况： 如果全新别名本身就是根别名，那么直接 `BaseYii::aliases[$alias] = $path` ； 而如果全新的别名并非是一个根别名，即形如 `@foo/bar` 带有二级、三级等路径的， `BaseYii::aliases[$root] = [$alias => $path]` 。比如:

```
1  // BaseYii::aliases['@foo'] = ['@foo/bar' => 'path/to/foo/bar']
2  Yii::setAlias('@foo/bar', 'path/to/foo/bar');
3
4  // BaseYii::aliases['@qux'] = 'path/to/qux'
5  Yii::setAlias('@qux', 'path/to/qux');
```

而对于根别名已经存在的别名，在写入时，就要考虑覆盖、新增的问题了:

```
1  // 初始 BaseYii::aliases['@foo'] = 'path/to/foo'
2  Yii::setAlias('@foo', 'path/to/foo');
3
4  // 直接覆盖 BaseYii::aliases['@foo'] = 'path/to/foo2'
5  Yii::setAlias('@foo', 'path/to/foo2');
6
7  /**
8  * 新增
9  * BaseYii::aliases['@foo'] = [
10 *     '@foo/bar' => 'path/to/foo/bar',
11 *     '@foo' => 'path/to/foo2',
12 * ];
13 */
14 Yii::setAlias('@foo/bar', 'path/to/foo/bar');
15
```

```
// 初始 BaseYii::aliases['@bar'] = ['@bar/qux' => 'path/to/bar/qux'];
Yii::setAlias('@bar/qux', 'path/to/bar/qux');

// 直接覆盖 BaseYii::aliases['@bar'] = ['@bar/qux' => 'path/to/bar/qux2'];
Yii::setAlias('@bar/qux', 'path/to/bar/qux2');

/**
 * 新增
 * BaseYii::aliases['@bar'] = [
 *     '@bar/foo' => 'path/to/bar/foo',
 *     '@bar/qux' => 'path/to/bar/qux2',
 * ];
 */
Yii::setAlias('@bar/foo', 'path/to/bar/foo');
```

注意如果根别名对应的是一个数组，在新增、覆盖后， Yii会调用PHP的 `krsort()` 把数组按照键值重新逆向排序。 这可以有效确保长的别名会放在短的类以别名前面， 比如， `@foo/bar/qux` 会被放在 `@foo/bar` 同样被放在根别名 `@foo` 之下， 但长的那个，会被放在前面。

别名的删除 传入的 `$path` 为 `null` 表示要删除别名。 Yii使用PHP的 `unset()` 注销 `BaseYii::$aliases[]` 数组中的对应元素， 达到删除别名的目的。 注意删除别名后，不需要调用 `krsort()` 对数组进行处理。

别名的解析过程

与定义过程使用 `Yii::setAlias()` 相对应，别名的解析过程使用 `Yii::getAlias()` ，实际代码在 `BaseYii::getAlias()` 中:

```
public static function getAlias($alias, $throwException = true)
{
    // 一切不以@打头的别名都是无效的
    if (strncmp($alias, '@', 1)) {
        return $alias;
    }

    // 先确定根别名 $root
    $pos = strpos($alias, '/');
    $root = $pos === false ? $alias : substr($alias, 0, $pos);

    // 从根别名开始找起，如果根别名没找到，一切免谈
    if (isset(static::$aliases[$root])) {
        if (is_string(static::$aliases[$root])) {
            return $pos === false ? static::$aliases[$root] : static::$aliases[$root] . substr($a
        } else {
            // 由于写入前使用了 krsort() 所以，较长的别名会被先遍历到。
            foreach (static::$aliases[$root] as $name => $path) {
                if (strpos($alias . '/', $name . '/') === 0) {
                    return $path . substr($alias, strlen($name));
                }
            }
        }
    }

    if ($throwException) {
        throw new InvalidParamException("Invalid path alias: $alias");
    } else {
        return false;
    }
}
```

别名的解析过程相对简单:

- 先按根别名找到可能保存别名的分支。
- 遍历这个分支下的所有树叶。由于之前叶子（别名）是按键值逆排序的，所以优先匹配长别名。
- 将找到的最长匹配别名替换成其所对应的值，再接上 `@alias` 的后半截，成为新的别名。

别名的解析过程可以这么看:

```
// 无效的别名，别名必须以@打头，别名不能放在中间
// 但是语句不会出错，会认为这是一个路径，一字不变的路径： path/to/@foo/bar
Yii::getAlias('path/to/@foo/bar');

// 定义 @foo @foo/bar @foo/bar/qux 3个别名
Yii::setAlias('@foo', 'path/to/foo');
Yii::setAlias('@foo/bar', 'path/2/bar');
Yii::setAlias('@foo/bar/qux', 'path/to/qux');

// 找不到 @foobar根别名，抛出异常
Yii::getAlias('@foobar/index.php');

// 匹配@foo，相当于 path/to/foo/qux/index.php
Yii::getAlias('@foo/qux/index.php');

// 匹配@foo/bar/qux，相当于 path/to/qux/2/index.php
Yii::getAlias('@foo/bar/qux/2/index.php');

// 匹配@foo/bar，相当于 path/to/bar/2/2/index.php
Yii::getAlias('@foo/bar/2/index.php');
```

3.1.3 小结

- 别名需在使用前定义，因此通常来讲，定义别名应当在放在应用的初始化阶段。
- 别名必然以 `@` 打头。
- 别名的定义可以使用之前已经定义过的别名。
- 别名在储存时，至多只分成两级，第一级的键是根别名。 第二级别名的键是完整的别名，而不是去除根别名后剩下的所谓的“二级”别名。
- Yii通过分层的树结构来保存别名最主要是为高效检索作准备。
- 很多地方可以直接使用别名，而不用调用 `Yii::getAlias()` 转换成真实的路径或URL。
- 别名解析时，优先匹配较长的别名。
- Yii预定义了许多常用的别名供编程时使用。
- 使用别名时，要将别名放在最前面，不能放在中间。

3.2 配置项（configuration）(TBD)

说到配置项，读者朋友们第一反应是不是Yii的配置文件？这是一段配置文件的代码:

```
'db' => [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=mylocaldb',
    'username' => 'username',
    'password' => 'password',
    'charset' => 'utf8',
    'tablePrefix' => 'tbl_',
],
'mail' => [
```

```
    'class' => 'yii\swiftmailer\Mailer',
    'viewPath' => '@common/mail',
    'useFileTransport' => true,
],
```

上面两个配置项都取自实际项目，截取的是 components 的配置内容。

Yii中许多地方都要用到配置项，应用和其他几乎一切类对象的创建、初始化、配置都要用到配置项。配置项是针对对象而言的，也就是说，配置项一定是用于配置某一个对象，用于初始化或配置对象的 属性<*property.html*> 。

3.2.1 配置项的格式

简单来讲，一个配置项采用下面的格式:

```
[
    'class' => 'path\to\ClassName',
    'propertyName' => 'propertyValue',
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

作为配置项:

- 配置项以数组进行组织;
- class 数组元素表示将要创建的对象的完整类名;
- propertyName 数组元素表示指定为 propertyName 属性的初始值;
- on eventName 数组元素表示将 $eventHandler 绑定到对象的 eventName 事件中;
- as behaviorName 数组元素表示用 $behaviorConfig 创建一个行为，并注入到对象中。 这里的 $behaviroConfig 也是一个配置项;
- 配置项可以嵌套。

3.2.2 配置项的原理

Yii调用 Yii::createObject() 和 Yii::configure() 将传入的配置项作用于对象，前者用于创建一个对象，后者用于配置一个对象。

Yii::createObject() 的代码在 BaseYii::createObject() 中:

```
public static function createObject($type, array $params = [])
{
    if (is_string($type)) {
        return static::$container->get($type, $params);
    } elseif (is_array($type) && isset($type['class'])) {
        $class = $type['class'];
        unset($type['class']);
        return static::$container->get($class, $params, $type);
    } elseif (is_callable($type, true)) {
        return call_user_func($type, $params);
    } elseif (is_array($type)) {
        throw new InvalidConfigException('Object configuration must be an array containing a "cla
    } else {
        throw new InvalidConfigException("Unsupported configuration type: " . gettype($type));
    }
}
```

3.3 Yii应用的目录结构

以下是一个通过高级模版安装后典型的Yii应用的目录结构:

```
.
├── backend
├── common
├── composer.json
├ composer.lock
├── console
├── environments
├── frontend
├── init
├── init.bat
├── LICENSE.md
├── README.md
├── requirements.php
├── vendor
└── yii.bat
```

对于高级应用而言，相当于有 backend frontend console 三个独立的Yii应用。由于 console 类的应用比较特殊，我们稍后再讲。这里讲典型的Web应用的目录结果。

3.3.1 公共目录

这里的公共目录可不止 common 目录，这从字面来看，没有任何疑义，就是公共目录。 common 的作用表示，这一目录下的东西，对于本高级应用的任一独立的应用而言，都是可见、可用的。一般情况下， common 具有以下结构:

```
.
├── codeception.yml
├── config
├── mail
├── models
└── tests
```

其中:

- config 就是通用的配置，这些配置将作用于前后台和命令行。
- mail 就是应用的前后台和命令行的与邮件相关的布局文件等。
- models 就是前后台和命令行都可能用到的数据模型。这也是 common 中最主要的部分。
- tests 就是应用的通用测试的内容。

除了 common 之外，还有一个很重要的公共目录， vendor 这个目录从字面的意思看，就是各种第三方的程序。这是Composer安装的其他程序的存放目录，包含Yii框架本身，也放在这个目录下面。如果你向 composer.json 目录增加了新的需要安装的程序，那么下次调用Composer的时候，将会把新安装的目录也安装在这个 vendor 下面。

好了，现在问题来了。对于 frontend backend console 等独立的应用而言，他们的内容放在各自的目录下面，他们的运作必然用到Yii框架等 vendor 中的程序。他们是如何关联起来的。这个秘密，或者说整个Yii应用的目录结构的秘密，就包含在一个传说中的称为入口文件的地方。

但是在了解入口文件index.php之前，有必要先看看诸如 frontend 等独立应用的目录结构。这比起整个Yii应用的目录结构面言，更为重要。因为你往往是在 frontend 等目录下写代码，但是，不大会在 path\to\yii-application 目录下写代码。

3.3.2 frontend 应用的目录结构

典型的， frontend 具有如下的一个目录结构:

TODO

按照顺序来讲:

- assets 目录用于存放前端资源包PHP类。这里不需要了解什么是前端资源包，只要大致知道是用于管理CSS、js等前面资源的就可以来。
- config 用于存在存放本应用的配置文件，包含主配置文件 main.php 和全局中参数配置文件 params.php 。
- models views controllers 目录分别用于存放数据模型类、视图文件、控制器类。这个是我们编码的核心，也是我们工作最多的目录。
- widgets 目录用于存放一些常用的小挂件的类文件。
- tests 目录用于存放测试类。
- web 目录从名字可以看出，这是一个对于Web服务器可以访问的目录。除了这一目录，其他所有的目录不应对Web用户暴露出来。这是安全的需要。
- runtime 这个目录是要求是 chmod 777 ，这要求Web服务器具有完全的权限，因为可能会涉及到写入临时文件等。但是一个目录并未对Web用户可见。也就是说，权限给了，但是并不是Web用户可以访问到的。

backend 目录与 frontend 的结构、内容是一样一样的。所谓的前台和后台，只是人为从逻辑上对Web应用的功能划分，目的在于降低应用的规模和复杂程度，便于维护和使用。从代码角度来讲，Yii压根就不认得哪个是前台，哪个是后台。

前面提到的，传说中的入口文件 index.php 就位于 web 目录下面。

3.3.3 入口文件index.php

首先来看看 index.php 文件的内容:

```
<?php
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');
require(__DIR__ . '/../../common/config/aliases.php');

$config = yii\helpers\ArrayHelper::merge(
    require(__DIR__ . '/../../common/config/main.php'),
    require(__DIR__ . '/../../common/config/main-local.php'),
    require(__DIR__ . '/../config/main.php'),
    require(__DIR__ . '/../config/main-local.php')
);

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

前两行是两个 define 语句，定义当前的运行模式和环境。如果定义了 YII_DEBUG 那么，表示当前为调试状态，应用在运行过程中，会有一些调试信息的输出，在抛出异常时，也会有一个详细的调用栈的显示。默认情况下， YII_DEBUG 为 false 。但在开发过程中，最好按上面写的那样，定义为 true 这样便于查找和分析错误。

如果定义了 YII_ENV ，那么就是指定了当前应用的运行环境。上面的代码显示应用将运行于 dev 环境。默认情况下， YII_ENV 为 prod 。这些环境只是一个名称，具体的内容要看环境的定义。但是 dev prod 是安装后默认的两个环境，分别表示开发环境和最终的成品环境。此外还有一个 test 环境，表示测试环境。

环境与模式的作用不同。环境在代码中主要是影响配置文件。 YII_ENV 的 dev prod test 三种环境，会分别使 YII_ENV_DEV YII_ENV_PROD YII_ENV_TEST 的值为 true 。这样，在应用的配置中，特别是在相同的一个配置文件中，可以对不同环境作出不同的配置。

比如，你希望在测试环境下，使用另一个数据库。在开发环境下，启用调试工作条，等等。那么，可以这么做:

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

if (!YII_ENV_TEST) {
    // configuration adjustments for 'test' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

其实，这个 YII_ENV 的定义只是一个与 init 脚本环境切换的一个相互补充。如果各环境比较明晰，用 init 来切换各种环境的配置完全够了。不必在脚本中再有如 YII_ENV_TEST 之类的判断语句。

紧接着两个 define 语句之后，就是三个 require 语句。三个语句中都使用到了相对于当前目录的其他目录中的php文件。 __DIR__ 表示当前文件 index.php 所在的目录。 /../../ 表示的是当前目录的爷爷目录， index.php 的当前目录是 /path/to/yii-application/frontend/web ，爸爸目录就是 frontend ，爷爷目录就是 yii-application 了。

第一个目录引入了 /path/to/yii-application/vendor 下面的 autoload.php 。这个是composer的类自动加载机制注册文件。引入这个文件后，可以使用composer的类自动加载功能。

第二个目录引入了 vendor 下面的 yiisoft/yii2/Yii.php ，这是Yii的类文件。引入了这个类文件后，才能使用Yii的各种类。

第三个目录引入了 /path/to/yii-application/common 下面的 config/aliases.php 这个文件定义了一系列的路径别名:

```
<?php
Yii::setAlias('common', dirname(__DIR__));
Yii::setAlias('frontend', dirname(dirname(__DIR__)) . '/frontend');
Yii::setAlias('backend', dirname(dirname(__DIR__)) . '/backend');
Yii::setAlias('console', dirname(dirname(__DIR__)) . '/console');
Yii::setAlias('vendor', dirname(dirname(__DIR__)) . '/vendor');
```

这是默认安装后定义好的 common frontend backend console vendor 5个路径别名，如果你要新增一个用于表示插件的目录 plugin 可以自己在这个文件里面加一行:

```
Yii::setAlias('plugin', dirname(dirname(__DIR__)) . '/plugins');
```

其实，是不提倡直接修改所有的配置文件的。对于任何配置文件的修改，最好都是通过对应环境的修改。这是先搞清楚这个配置项的修改影响到哪些环境。然后选择对应的环境配置文件进行修改。完了，再调用 init 重新加载一次环境就OK了。不然，这次万一修改了，以后在环境切换时，可能就会把修改的内容不小心覆盖掉了。

再接下来，是一个函数 yii\helpers\ArrayHelper::merge() 这个函数的作用在于合并参数所指定的各个数组。其中，后面的数组会把前面数组中，相同下标的元素覆盖掉。这个语句的作用，就是读取、合并应用的各项配置并保存在 $config 变量中。这里我们看到一共是读取了4个配置文件:

```
require(__DIR__ . '/../../common/config/main.php'),
require(__DIR__ . '/../../common/config/main-local.php'),
require(__DIR__ . '/../config/main.php'),
require(__DIR__ . '/../config/main-local.php')
```

依次是 common 通用目录下的2个配置文件，和当前的2个配置文件。因此，这就是前面提到的配置文件的优先顺序。当前的配置覆盖通用的配置。在通用配置和当前配置中，带有 -local 的配置文件在后，所以，本地配置文件优先。

最后，以 `$config` 为参数，实例化了一个 `Application` 对象，并调用他的 `run()` 函数。这时，Yii应用就跑起来了。

3.4 Yii的类自动加载机制

在Yii中，所有类、接口、Traits都可以使用类的自动加载机制实现在调用前自动加载。Yii借助了PHP的类自动加载机制高效实现了类的定位、导入，这一机制兼容 PSR-4 的标准。在Yii中，类仅在调用时才会被加载，特别是核心类，其定位非常快，这也是Yii高效高性能的一个重要体现。

3.4.1 自动加载机制的实现

Yii的类自动加载，依赖于PHP的 `spl_autoload_register()` ，注册一个自己的自动加载函数（autoloader），并插入到自动加载函数栈的最前面，确保Yii的autoloader会被最先调用。

类自动加载的这个机制的引入要从入口文件 `index.php` 开始说起:

```
<?php
defined('YII_DEBUG') or define('YII_DEBUG', false);
defined('YII_ENV') or define('YII_ENV', 'prod');

// 这个是第三方的autoloader
require(__DIR__ . '/../../vendor/autoload.php');

// 这个是Yii的Autoloader，放在最后面，确保其插入的autoloader会放在最前面
require(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
// 后面不应再有autoloader了

require(__DIR__ . '/../../common/config/aliases.php');

$config = yii\helpers\ArrayHelper::merge(
    require(__DIR__ . '/../../common/config/main.php'),
    require(__DIR__ . '/../../common/config/main-local.php'),
    require(__DIR__ . '/../config/main.php'),
    require(__DIR__ . '/../config/main-local.php')
);

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

这个文件主要看点在于第三方autoloader与Yii 实现的autoloader的顺序。不管第三方的代码是如何使用 `spa_autoload_register()` 来注册自己的autoloader的，只要Yii 的代码在最后面，就可以确保其可以将自己的autoloader插入到整个autoloder 栈的最前面，从而在需要时最先被调用。

接下来，看看Yii是如何调用 `spl_autoload_register()` 注册autoloader的，这要看 `Yii.php` 里发生了些什么:

```
<?php
require(__DIR__ . '/BaseYii.php');
class Yii extends \yii\BaseYii
{
}

// 重点看这个 spl_autoload_register
spl_autoload_register(['Yii', 'autoload'], true, true);

// 下面的语句读取了一个映射表
Yii::$classMap = include(__DIR__ . '/classes.php');

Yii::$container = new yii\di\Container;
```

这段代码，调用了 `spl_autoload_register(['Yii', 'autoload', true, true])`，将 `Yii::autoload()` 作为autoloader插入到栈的最前面了。并将 `classes.php` 读取到 `Yii::$classMap` 中，保存了一个映射表。

在上面的代码中，Yii类是里面没有任何代码，并未对 `BaseYii::autoload()` 进行重载，所以，这个 `spl_autoload_register()` 实际上将 `BaseYii::autoload()` 注册为autoloader。如果，你要实现自己的autoloader，可以在 Yii 类的代码中，对 `autoload()` 进行重载。

在调用 `spl_autoload_register()` 进行autoloader注册之后，Yii将 `calsses.php` 这个文件作为一个映射表保存到 `Yii::$classMap` 当中。这个映射表，保存了一系列的类名与其所在PHP文件的映射关系，比如:

```
return [
  'yii\base\Action' => YII2_PATH . '/base/Action.php',
  'yii\base\ActionEvent' => YII2_PATH . '/base/ActionEvent.php',

  ... ...

  'yii\widgets\PjaxAsset' => YII2_PATH . '/widgets/PjaxAsset.php',
  'yii\widgets\Spaceless' => YII2_PATH . '/widgets/Spaceless.php',
];
```

这个映射表以类名为键，以实际类文件为值，Yii所有的核心类都已经写入到这个 `classes.php` 文件中，所以，核心类的加载是最便捷，最快的。现在，来看看这个关键先生 `BaseYii::autoload()`

```
public static function autoload($className)
{
    if (isset(static::$classMap[$className])) {
        $classFile = static::$classMap[$className];
        if ($classFile[0] === '@') {
            $classFile = static::getAlias($classFile);
        }
    } elseif (strpos($className, '\\') !== false) {
        $classFile = static::getAlias('@' . str_replace('\\', '/', $className) . '.php', false);
        if ($classFile === false || !is_file($classFile)) {
            return;
        }
    } else {
        return;
    }

    include($classFile);

    if (YII_DEBUG && !class_exists($className, false) && !interface_exists($className, false) &&
        throw new UnknownClassException("Unable to find '$className' in file: $classFile. Namespac
    }
}
```

从这段代码来看Yii类自动加载机制的运作原理:

- 检查 `$classMap[$className]` 看看是否在映射表中已经有拟加载类的位置信息;
- 如果有，再看看这个位置信息是不是一个路径别名，即是不是以 `@` 打头，是的话，将路径别名解析成实际路径。如果映射表中的位置信息并非一个路径别名，那么将这个路径作为类文件的所在位置。类文件的完整路径保存在 `$classFile`;
- 如果 `$classMap[$className]` 没有该类的信息，那么，看看这个类名中是否含有 `\`，如果没有，说明这是一个不符合规范要求的类名，autoloader直接返回。PHP会尝试使用其他已经注册的autoloader进行加载。如果有 `\`，认为这个类名符合规范，将其转换成路径形式。即所有的 `\` 用 `/` 替换，并加上 `.php` 的后缀。
- 将替换后的类名，加上 `@` 前缀，作为一个路径别名，进行解析。从别名的解析过程我们知道，如果根别名不存在，将会抛出异常。所以，类的命名，必须以有效的根别名打头:

```
// 有效的类名，因为@yii是一个已经预定义好的别名
use yii\base\Application;

// 无效的类名，因为没有 @foo 或 @foo/bar 的根别名，要提前定义好
use foo\bar\SomeClass;
```

- 使用PHP的 `include()` 将类文件加载进来，实现类的加载。

从其运作原理看，最快找到类的方式是使用映射表。 其次，Yii中所有的类名，除了符合规范外，还需要提前注册有效的根别名。

3.4.2 运用自动加载机制

在入口脚本中，除了Yii自己的autoloader，还有一个第三方的autoloader:

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

这个其实是Composer提供的autoloader。Yii使用Composer来作为包依赖管理器，因此，建议保留Composer的autoloader，尽管Yii的autoloader也能自动加载使用Composer安装的第三方库、扩展等，而且更为高效。但考虑到毕竟是人家安装的，人家还有一套自己专门的规则，从维护性、兼容性、扩展性来考虑，建议保留Composer的autoloader。

如果还有其他的autoloader，一定要在Yii的autoloader注册之前完成注册，以保证Yii的autoloader总是最先被调用。

如果你有自己的autoloader，也可以不安装Yii的autoloaer，只是这样未必能有Yii的高效，且还需要遵循一套类似的类命名和加载的规则。就个人的经验而言，Yii的autoloader完全够用，没必要自己重复造轮子。

至于Composer如何自动加载类文件，这里就不过多的占用篇幅了。可以看看 Composer的文档 。

CHAPTER

FOUR

YII 模式

Yii中使用了当前Web开发中最为主流和成熟的设计模式。

4.1 MVC

MVC是一种设计模式（Design pattern），也就是一种解决问题的方法和思路。MVC不仅仅存在于Web设计中，在桌面程序开发中也是一种常见的方法。MVC的出现已经有一段历史了。记得我最早了解到MVC的时候，是在Microsoft的Visual C++ 中的MFC中。当时年少无知，以为是MFC中特有的东西。后来随之着不断学习，才发现自己的天真。所以说，学得越多，就越觉得自己无知。而觉得自己无知的人，往往谦逊。从这个角度讲，最好不要看不起谦逊的人。

话说远了。MVC是三个单词的缩写：Model, View, Controller。MVC是一种设计模式，目前几乎所有的Web开发框架都建立在MVC模式之上。当然，最近几年也出现了一些诸如MVP, MVVM之类的新的设计模式。但从技术的成熟和使用的广泛程度来讲，MVC仍是主流。

Yii是一个Web框架，从Web开发来讲，Yii的开发工作中，后端的内容多一些，而且主要就是在Yii的基础上开发。前端主要是在JavaScript上进行开发，然后通过Yii把前端的内容管起来，如Assets等。这一章要讲的MVC，主要是针对后端的。前端的MVC严格来讲不属于Yii的范畴，而且也不是最主要的内容，这里我们就不作过多介绍，如果想了解前端的MVC，可以看看Backbone等前端框架。

下面是MVC中三个要素的简介。

- Model是指数据模型，是对客观事物的抽象。如一篇博客文章，我们可能会以一个Post类来表示，那么，这个Post类就是数据对象。同时，博客文章还有一些业务逻辑，如发布、回收、评论等，这一般表现为类的方法，这也是model的内容和范畴。对于model，主要是数据、业务逻辑和业务规则。相对而言，这是MVC中比较稳定的部分，一般成品后不会改变。
- View是指视图，也就是呈现给用户的一个界面，是model的具体表现形式。如你在我的博客上看到的某一篇博客文章，就是某个Post类的表现形式。View的目的在于提供与用户交互的界面。换句话说，对于用户而言，只有View是可见的。事实上也是如此，你不会让用户看到Model，也不会让他看到Controller。你只会让用户看到你想让他看的内容。这就是View要做的事，他往往是MVC中变化频繁的部分，今天你可能会以一种形式来展示你的博文，明天可能就变成别的表现形式了。
- Contorller指的是控制器，主要负责与model和view打交道。换句话说，model和view之间一般不直接打交道，他们老死不相往来，view中不会对model作任何操作，model不会输出任何用于表现的东西，如HTML代码等。Contorller用于从model获取数据、调用类方法，为view准备要输出的内容等工作，是MVC中沟通的桥梁。

4.1.1 MVC的划分原则

对于MVC中三者的划分并没有十分明晰的定义和界线，只是一种指导思想，让你按照model, view, controller三个方面来描述你的应用，并通过三者的交互，使应用功能得以正常运转。

其中，View的部分比较明确，就是负责显示嘛。一切与显示界面无关的东西，都不应该出现在View里面。因此，View中一般不会出现复杂的判断语句，不会出现复杂的运算过程。对于PHP的Web应用而言，毫无疑问，HTML是View中的主要内容。这是关于View的几个原则：

- 负责显示界面，以HTML为主；
- 一般没有复杂的判断语句或运算过程，可以有简单的循环语句、格式化语句。比如，博客首页的文章列表，就是一种循环；
- 从不调用Model的写方法。也就是说，View只从Model中获取数据，而从不改写model，所以我们说他们老死不相往来。
- 一般没有任何准备数据处理的内容，如查询数据库等。这些一般放在Controller里面，并以变量的形式传给视图。也就是说，视图里面要用到的数据，就是一个变量。

对于Model而言，最主要就是保存事物的信息，表征事物的行为和对他可以进行的操作。比如，Post类必然有一个用于保存博客文章标题的title属性，必然有一个删除的操作，这都是Model的内容。以下是关于Model的几个原则：

- 数据、行为、方法是Model的主要内容；
- 实际工作中，Model是MVC中代码量最大，逻辑最复杂的地方，因为关于应用的业务逻辑也要在这里面表示。
- 注意与Controller区分开。Model是处理业务方面的逻辑，Controller只是简单的协调Model和View之间的关系。只要是与业务有关的，就该放在Model里面。好的设计，应当是胖Model，瘦Controller。

对于Controller，主要是响应用户请求，决定使用什么视图，需要准备什么数据用来显示。以下是有关Controller的设计原则：

- 用于处理用户请求，因此，对于reqeust的访问代码应该放在Controller里面，比如 `$_GET` `$_POST` 等。但仅限于获取用户请求数据，不应该对数据有任何操作或预处理，这应该放在models里面。
- 调用models的类方法，对models进行写操作。
- 调用视图渲染函数，形成对用户reqeust的response。
- 一般不要有HTML代码等其他表现层的东本，这属于View的内容。

4.1.2 Yii中的MVC的前后端配合

从MVC的起源来讲，是从桌面应用的开发中发展起来的。从本质来讲，这是一种解决问题的思路和办法。从实践来讲，这是一种久经考验的有效方式。但是如开头我们讲的，Yii更多的是侧重于后端。因此，包含Yii在内的许多Web开发框架，只要是使用MVC的模式，就必然要注意以下这两点：

- Yii的view没有办法知道用户的操作，如鼠标、键盘的操作。对于这部份，只是依靠前端技术JavaScript等。通过JavaScript捕获用户操作，进行相应片理：或发送回后端进行处理（如ajax等），或直接在前端处理（如使用backbone.js等前端框架）。
- Yii1.1中没有一个View的类，View文件只是Controller的延伸。没有真正意义上的视图。到了Yii2，终于有了View类来独立表征视图，从逻辑上，View已经基本独立出来了。在视图文件中访问 `$this` 指向的不再是Yii1.1中当前controller，而是指向当前的视图。

4.2 Yii中的服务定位器与依赖注入(TBD)

为了降低代码耦合程度，提高项目的可维护性，Yii采用多许多当下最流行又相对成熟的设计模式，包括了依赖注入（Denpdency Injection, DI）和服务定位器（Service Locator）两种模式。关于依赖注入与服务定位器，*Inversion of Control Containers and the Dependency Injection pattern <http://martinfowler.com/articles/injection.html>* 给出了很详细的讲解，这里结合Web应用和Yii具体实现进行探讨，以加深印象和理解。这些设计模式对于提高自身的设计水平，很有帮助，这也是学习Yii的一个重要出发点。

4.2.1 有关概念

在了解Service Locator 和 Dependency Injection 之前，有必要先来了解一些高大上的概念。别担心，你只需要有个大致了解就OK了，如果展开来说，这些东西可以单独写个研究报告：

依赖倒置原则（Dependence Inversion Principle, DIP） DIP是一种软件设计的指导思想。传统软件设计中，上层代码依赖于下层代码，当下层出现变动时，上层代码也要相应变化，维护成本较高。而DIP的核心思想是上层定义接口，下层实现这个接口，从而使得下层依赖于上层，降低耦合度，提高整个系统的弹性。这是一种经实践证明的有效策略。

控制反转（Inversion of Control, IoC） IoC就是DIP的一种具体思路，DIP只是一种理念、思想，而IoC是一种实现DIP的方法。IoC的核心是将类（上层）所依赖的单元（下层）的实例化过程交由第三方来实现。一个简单的特征，就是类中不对所依赖的单元有诸如 `$component = new yii\component\SomeClass()` 的实例化语句。

依赖注入（Dependence Inversion, DI） DI是IoC的一种设计模式，是一种套路，按照DI的套路，就可以实现IoC，就能符合DIP原则。DI的核心是把类所依赖的单元的实例化过程，放到类的外面去实现。

- **控制反转容器（IoC Container）** 当项目比较大时，依赖关系可能会很复杂。而IoC Container提供了动态地创建、注入依赖单元，映射依赖关系等功能，减少了许多代码量。Yii 实现了一个 `yii\di\Container` 实现了一个 DI Container。
- **服务定位器（Service Locator）** Service Locator是IoC的另一种实现方式，其核心是把所有可能用到的依赖单元交由Service Locator进行实例化和创建、配置，把类对依赖单元的依赖，转换成类对Service Locator的依赖。DI 与 Service Locator并不冲突，两者可以结合实用。Yii把这DI和Service Locator这两个东西结合起来使用。

4.2.2 依赖注入

首先讲讲DI。在Web应用中，很常见的是使用各种第三方Web Service实现特定的功能，比如发送邮件、推送微博等。假设要实现当访客在博客上发表评论后，向博文的作者发送Email的功能，通常代码会是这样：

```
1  // 为邮件服务定义抽象层
2  interface EmailSenderInterface
3  {
4      public function send(...);
5  }
6
7  // 定义Gmail邮件服务
8  class GmailSender implements EmailSenderInterface
9  {
10     ...
11
12     public function send(...)
13     {
14         ...
15     }
16 }
17
18 // 定义评论类
19 class Comment extend yii\db\ActiveRecord
20 {
21     // 用于引用发送邮件的库
22     private $_eMailSender;
23
24     // 初始化时，实例化 $_eMailSender
25     public function init()
26     {
27         ...
28         // 这里假设使用Gmail的邮件服务
```

```
        $this->_eMailSender = GmailSender::getInstance();
        ...
    }

    // 当有新的评价时触发
    public function afterInsert()
    {
        ...
        //
        $this->_eMailSender->send(...);
        ...
    }
}
```

上面的代码只是一个示意，大致是这么个流程。

那么这种常见的设计方法有什么问题呢？ 主要问题在于 Comment 对于 GmailSender 的依赖（对于EmailSenderInterface的依赖不可避免）， 假设有一天突然不使用Gmail提供的服务了，改用Yahoo或自建的邮件服务了。 那么，你不得不修改 `Comment::init()` 里面对 `$_eMailSender` 的实例化语句:

```
$this->_eMailSender = MyEmailSender::getInstance();
```

这个问题的本质在于，你今天写完这个Comment，只能用于这个项目，哪天你开发别的项目要实现类似的功能， 你还要对针对新项目使用的邮件服务修改这个Comment。代码的复用性不高呀。有什么办法可以不改变Comment的代码，就能扩展成对各种邮件服务都支持么？ 换句话说，有办法将Comment和GmailSender解耦么？有办法提高Comment的普适性、复用性么？

依赖注入就是为了解决这个问题而生的，当然，DI也不是唯一解决问题的办法，毕竟条条大路通罗马。 Service Locator也是可以实现解耦的。

在Yii中使用DI解耦，有2种注入方式：构造函数注入、属性注入。

构造函数注入

构造函数注入通过为构造函数的形参，为类内部的抽象单元提供实例化。 具体的构造函数调用代码，由外部代码决定。具体例子如下:

```
// 这是构造函数注入的例子
class Comment extend yii\db\ActiveRecord
{
    // 用于引用发送邮件的库
    private $_eMailSender;

    // 构造函数注入
    public function __construct($emailSender)
    {
        ...
        $this->_eMailSender = $emailSender;
        ...
    }

    // 当有新的评价时触发
    public function afterInsert()
    {
        ...
        //
        $this->_eMailSender->send(...);
        ...
    }
}

// 实例化两种不同的邮件服务，当然，他们都实现了EmailSenderInterface
```

```
sender1 = new GmailSender();
sender2 = new MyEmailSender();

// 用构造函数将GmailSender注入
$comment1 = new Comment(sender1);
// 使用Gmail发送邮件
$comment1.save();

// 用构造函数将MyEmailSender注入
$comment2 = new Comment(sender2);
// 使用MyEmailSender发送邮件
$comment2.save();
```

属性注入

与构造函数注入类似，通过setter或public成员变量，将所依赖的单元注入到类内部。 具体的属性写入，由外部代码决定。具体例子如下:

```
// 这是属性注入的例子
class Comment extend yii\db\ActiveRecord
{
    // 用于引用发送邮件的库
    private $_eMailSender;

    public function setEmailSender($value)
    {
        $this->_eMailSender = $value;
    }

    // 当有新的评价时触发
    public function afterInsert()
    {
        ...
        //
        $this->_eMailSender->send(...);
        ...
    }
}

// 实例化两种不同的邮件服务，当然，他们都实现了EmailSenderInterface
sender1 = new GmailSender();
sender2 = new MyEmailSender();

$comment1 = new Comment;
// 使用属性注入
$comment1->eMailSender = sender1;
// 使用Gmail发送邮件
$comment1.save();

$comment2 = new Comment;
// 使用属性注入
$comment2->eMailSender = sender2;
// 使用MyEmailSender发送邮件
$comment2.save();
```

上面的Comment如果将 `private $_eMailSender` 改成 `public $eMailSender` 并删除 setter函数， 也是可以达到同样的效果的。

4.2.3 DI容器

从上面DI两种注入方式来看，依赖单元的实例化代码是一个重复、繁琐的过程。可以想像，一个Web应用的某一组件会依赖于若干单元，这些单元又有可能依赖于更低层级的单元，从而形成依赖嵌套的情形。那么，这些依赖单元的实例化、注入过程的代码可能会比较长，前后关系也需要注意。这实在是一件既没技术含量，又吃力不出成果的工作。

DI容器就是为了解决这一难题而设计出来的。Yii的DI容器是 `yii\di\Container`，可以实现对对象的实例化和配置，并自动实例化和配置需要用到的依赖单元。

注册依赖

要使用DI容器，首先要告诉容器，类及类之间的依赖关系。使用 `yii\di\Container::set()` 和 `yii\di\Container::setSinglton()` 可以注册依赖。DI容器是怎么管理依赖的呢？要先看看 `yii\di\Container::set()` 和 `yii\Container::setSinglton`

```
public function set($class, $definition = [], array $params = [])
{
    // 规范化 $definition 并写入 $_definitions[$class]
    $this->_definitions[$class] = $this->normalizeDefinition($class, $definition);

    // 将构造函数参数写入 $_params[$class]
    $this->_params[$class] = $params;

    // 删除$_singletons[$class]
    unset($this->_singletons[$class]);
    return $this;
}

public function setSingleton($class, $definition = [], array $params = [])
{
    // 规范化 $definition 并写入 $_definitions[$class]
    $this->_definitions[$class] = $this->normalizeDefinition($class, $definition);

    // 将构造函数参数写入 $_params[$class]
    $this->_params[$class] = $params;

    // 将$_singleton[$class]置为null，表示还未实例化
    $this->_singletons[$class] = null;
    return $this;
}
```

这两个函数功能类似没有太大区别，只是 `set()` 用于在每次请求时构造新的实例返回，而 `setSingleton()` 只维护一个单例，每次请求时都返回同一对象。

从形参来看，这两个函数的 `$class` 接受一个类名、接口名或一个别名，作为依赖的名称。`$definition` 表示依赖的定义，可以是一个类名、配置数组或一个PHP callable。

这两个函数，本质上只是将依赖的有关信息写入到容器的相应数组中去。在DI容器中，维护了5个数组:

```
// 用于保存单例Singleton对象，以对象类型为键
private $_singletons = [];
// 用于保存依赖的定义，以对象类型为键
private $_definitions = [];
// 用于保存构造函数的参数，以对象类型为键
private $_params = [];
// 用于缓存ReflectionClass对象，以类名或接口名为键
private $_reflections = [];
// 用于缓存依赖，以类名或接口名为键
private $_dependencies = [];
```

在 set() 和 setSingleton() 中，首先调用 yii\di\Container::normalizeDefinition() 对依赖的定义起规范化处理，其代码如下:

```
protected function normalizeDefinition($class, $definition)
 {
     // $definition 是空的
     if (empty($definition)) {
         return ['class' => $class];

     // $definition 是字符串
     } elseif (is_string($definition)) {
         return ['class' => $definition];

     // $definition 是PHP callable 或 对象
     } elseif (is_callable($definition, true) || is_object($definition)) {
         return $definition;
     // $definition 是数组
     } elseif (is_array($definition)) {
         if (!isset($definition['class'])) {
             if (strpos($class, '\\') !== false) {
                 $definition['class'] = $class;
             } else {
                 throw new InvalidConfigException("A class definition requires a \"class\" member
             }
         }
         return $definition;
     } else {
         throw new InvalidConfigException("Unsupported definition type for \"$class\": " . gettype
     }
 }
```

规范化处理的流程如下:

- 如果 $definition 是空的，直接返回数组 ['class' => $class]
- 如果 $definition 是字符串，那么这个字符串就是所依赖的类名或接口名， 那么直接返回数组 ['class' => $definition]
- 如果 $definition 是一个PHP callable，或是一个对象，那么直接返回该 $definition
- 如果 $definition 是一个数组，那么其应当是一个包含了元素 $definition['class'] 的配置数组。 如果该数组未定义 $definition['class'] 那么，将传入的 $class 作为该元素的值，最后返回该数组。
- 如果 definition['class'] 未定义，而 $class 不是一个有效的类名，那么抛出异常。
- 如果 $definition 不属于上述的各种情况，也抛出异常。

在调用 normalizeDefinition() 进行规范化处理后， set() 和 setSingleton() 以传入的 $class 为键， 将定义保存进 $_definition[] 中，将传入的 $param 保存进 $_params[] 中。

对于 set() 而言，还要删除 $_singleton[] 中的同名依赖。 对于 setSingleton() 而言，则要将 $_singleton[] 中的同名依赖设为 null 。

这么讲可能不好理解，举几个具体的依赖定义为例，加深理解:

```
$container = new \yii\di\Container;

// 直接以类名注册一个依赖，虽然这么做没什么意义，但会使得 $_definition['yii\db\Connetcion'] = 'yii\db\Conn
$container->set('yii\db\Connection');

// 注册一个接口，当一个类依赖于一个接口时，指定的类会自动被实例化
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// 注册一个别名，当调用$container->get('foo')时，可以得到一个 yii\db\Connection 实例
$container->set('foo', 'yii\db\Connection');
```

```php

// 用一个配置数组来注册一个类，需要这个类的实例时，这个配置数组会发生作用
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// 用一个配置数组来注册一个别名，由于别名的类型不详，因此配置数组中需要有 class 元素
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// 用一个PHP callable来注册一个别名，每次引用这个别名时，这个callable都会被调用。
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});

// 用一个对象来注册一个别名，每次引用这个别名时，这个对象都会被引用。
$container->set('pageCache', new FileCache);
```

从 set() 和 setSingleton() 来看，可能还不容易理解DI容器，下面再从解析依赖的角度讲讲。

依赖的解析

与注册依赖时使用 set() 和 setSingleton() 对应，解析依赖使用 yii\di\Container::get()，其代码如下:

```php
public function get($class, $params = [], $config = [])
{
    // 已经有一个完成实例化的单例，直接引用这个单例
    if (isset($this->_singletons[$class])) {
        return $this->_singletons[$class];
    // 是个尚未注册过的依赖，根据传入的参数创建一个实例
    } elseif (!isset($this->_definitions[$class])) {
        return $this->build($class, $params, $config);
    }

    $definition = $this->_definitions[$class];

    if (is_callable($definition, true)) {
        $params = $this->resolveDependencies($this->mergeParams($class, $params));
        $object = call_user_func($definition, $this, $params, $config);
    } elseif (is_array($definition)) {
        $concrete = $definition['class'];
        unset($definition['class']);

        $config = array_merge($definition, $config);
        $params = $this->mergeParams($class, $params);

        if ($concrete === $class) {
            $object = $this->build($class, $params, $config);
        } else {
            $object = $this->get($concrete, $params, $config);
        }
    } elseif (is_object($definition)) {
```

```
29              return $this->_singletons[$class] = $definition;
30          } else {
31              throw new InvalidConfigException("Unexpected object definition type: " . gettype($definit
32          }
33
34          if (array_key_exists($class, $this->_singletons)) {
35              $this->_singletons[$class] = $object;
36          }
37
38          return $object;
39      }
```

4.2.4 服务定位器

Service Locator也是一种设计模式，实际中有些朋友认为这是一种反模式。但我的观点是使用得当，这也是一种好模式。 这里不引起无关主题的争论了。

引入Service Locator目的在于解耦。有许多成熟的设计模式也是用于解耦，但在Web应用上， Service Locator占有一席之地。这一模式的优点有：

- Service Locator充当了一个运行时的链接器的角色，可以在运行时动态地修改一个类所要选用的服务， 而不必对类作任何的修改。
- 一个类可以在运行时，有针对性地增减、替换所要用到的服务，从而得到一定程度的优化。
- 实现服务提供方、服务使用方完全的解耦，便于独立测试和代码跨框架复用。

CHAPTER

FIVE

安装YII

请留意Yii需要PHP5.4.0以上版本。

有两种方法可以安装Yii，一种是使用Composer，另一种是直接下载压缩包。

5.1 使用Composer安装Yii

推荐使用Composer安装Yii。这样更方便后期维护，如果需要添加新的扩展或者升级Yii，只要一句命令就OK了。

用过Yii1.1的读者可能还有印象，安装Yii和构建应用是两个步骤：安装Yii，运行Yii搭建应用框架。但是，如果使用Composer方法安装Yii，安装和构建应用是一步完成了。

本教程并不打算介绍PHP和Composer的方法，这方面的内容通过官方文档或者搜索引擎都可以找到。即使是用Composer安装Yii，也有两种选择：使用基本模版或者高级模版。这两者最主要的区别在于高级模版提供了环境切换和前后台分离。由于YiiBlog是需要前后台分离的，因此，我们使用高级模版来创建应用：

```
php composer.phar create-project --prefer-dist --stability=dev yiisoft/yii2-app-advanced /path/to/
```

如果使用基本模版，也是可以的。只不过需要手工添加前后台而已。这里就先不介绍了。TODO后面的章节会讲到。使用基本模版，也是一句语句搞定

```
composer create-project --prefer-dist yiisoft/yii2-app-basic /path/to/yii-application
```

如果想使用最新的开发版本的Yii，也可像使用高级模版那样，加入 `--stability=dev` 的参数。

5.2 从压缩包安装

如果使用压缩包安装方式，请按以下步骤:

1. 从yiiframework.com下载最新的压缩包。
2. 将压缩包解压缩到任意的 `/path/to/yii-application` 目录。
3. 修改 `config/web.php` 文件，输入 `cookieValidationKey` 配置项密钥。如果使用Composer安装，则Composer会自动设置一个密钥:

```
// !!! insert a secret key in the following (if it is empty) - this is required by cookie val
'cookieValidationKey' => 'enter your secret key here',
```

5.3 设置Web服务器

常用的Web服务器有nginx＋php-fpm和Apache。而且从趋势上看，前者的比重正不断提高。

本教程不打算介绍nginx和Apache的安装，这些从官方文档和搜索引擎都可以找到相关内容。这里只介绍如何配置Web服务器，使其能够让Yii跑起来。由于YiiBlog具有前台和后台。一般来讲，前台和后台分离可以使用不同的主机名、端口，或者使用不同的路径名。使用不同的主机名的，如:

```
http://frontend.example.com/
http://backend.example.com/
```

使用不同的路径名的，如:

```
http://www.example.com/frontend/
http://www.example.com/backend/
```

目的都是分离前台和后台。对于YiiBlog，由于是在本地开发，我们使用不同的端口来分离前台和后台。体现在服务器上，不同主机名、端口的分离方式，意味着不同的虚拟主机，甚至是不同的物理服务器。而不同的路径名的，则表现为同一台主机，不同目录。这里，我们使用不同的端口来区别前后台，但在物理上，前端和后端面都部署在同一台服务器上，也就是使用虚拟主机。

使用Nginx的配置如下：

```
# for frontend
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 localhost;
    root        /path/to/application/frontend/web;
    index       index.php;

    access_log  /path/to/application/frontend/log/access.log main;
    error_log   /path/to/application/frontend/log/error.log;

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

    location ~ \.php$ {
        include fastcgi.conf;
        fastcgi_pass   127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
    }

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

# for backend
server {
    # ...other settings...
    listen 81;
    server_name localhost;
    root /path/to/application/backend/web;
    index php;
    # ...other settings...
```

```
}
```

如果使用Apache，也是分前台和后前的配置，以于前台：

```
# Set document root to be "path/to/application/frontend/web"
DocumentRoot "path/to/application/frontend/web"

<Directory "path/to/application/frontend/web">
    RewriteEngine on

    # If a directory or a file exists, use the request directly
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # Otherwise forward the request to index.php
    RewriteRule . index.php

    # ...other settings...
</Directory>
```

对于后台，也是设置一个虚拟机，路径改为 `path/to/application/backend/web` 即可。

5.4 Yii中的前后台

Yii从来不认得什么是前台，什么是后台。从本质来讲，前台和后台都是应用。换句话说，你可以先用基本模板开发出一个应用，具有前台的全部功能，然后部署。如法炮制一个具有后台全部功能的应用。

这在原理上是一样一样的。只是，按照我们的经验，前台和后台从逻辑上个讲，组成一应用比较符合我们的认知。而且，从代码复用的角度来讲，我们更希望前台和后台的代码可以互通互用，尽量不要重复造轮子。但是，对于Yii来讲，前台就是一个完备的应用，后台又是另一个应用。这点区别请读者朋友们留意。

那么Yii的高级模版是如何实现把两个应用整合成一个我们认识上的应用的呢。我们观察一下 `/path/to/application/` 目录，看看这个高级模版是如何组织代码的。不难发现其中有二个目录 `frontend` `backend` 分别代表了前台、后台。

其实你把 `frontend` `backend` 中的任意1个目录删除，是不影响剩下的目录的正常运转的。也就是说，他们相互间是独立的。只不过代码组织下，他们都放在了 `/path/to/application/` 目录下面。

如果深入下去，你会发现不光 `frontend` `backend` 其实 `console` 也是一个完备的Yii应用，它就通常是应用的维护，是个命令行。

总的说，Yii的前台、后台什么的，是我们命名的概念，他们都是独立而完备的应用。他们又都具有一定的联系，这些联系突出体现在了 `common` 目录上。这个目录的意思是通用，对于组织到一起的 `frontend` `backend` `console` 其中的内容，他们都可以使用。这是Yii中实现代码复用的技巧所在。

5.5 配置应用环境

还差最后一步就完成Yii的安装了。这最后一步，就是设置应用环境。对于Yii的高级模版，引入了环境的概念。为什么要引入环境的概念呢?

让我们假设这种非常典型的情形：你们是一个开发团队，成员有Alice, Bob, Charlie等很多人，他们在自己的计算机上进行开发。其中，Alice使用Linux操作系统，Bob使用Mac OS，而Charlie使用Windows，每个人使用的IDE也不尽相同。这都没问题，PHP可以在不同平台上跑得很好。但是，问题来了，每个成员在开发的同时，搭建了自己的测试环境，每个人用于测试的数据库名称、用户名、密码都是根据自己的喜好命名的。于是，每次团队成员从团队仓库中获取新代码后，第一件事都是要更改为本地环境的数据库名称和密码。在提交代码前，又需要将其改回去。

于是你们想到一个解决办法，将环境配置文件排除在代码库之外。这样每次pull和push代码时都不用修改环境了。这确实解决了远程仓库端和本地端环境不同的问题。但不得不说，你们的代码库是不完

整的，里面缺少一个环境配置文件。更为要命的事，某天你们的团队经过讨论，认为需要在配置文件中加入新的配置项，于是你们不得不通知所有成员，自己手动更新。因为你们未能将该配置文件纳入版本管理。

另一种情况是：你们的团队是一个高效的快速迭代的团队。你们每天都会对代码进行更新。由于开发端和产品端环境肯定是不同的。于是每次更新时，你都需要将配置文件重新按照服务器进行配置。

毫无疑问，这些来回来去运行环境的切换，一定烦透你了。于是贴心的Yii引入了环境的概念来解决这个问题。其实，Yii1.1中还没有这个特性，这是新版本中才引入的技术。尽管在实践中，他已经很成熟了。

所谓环境，就是一组与运行环境相关的配置文件。Yii对于环境的使用是这样一个原理：采用一个自动化的脚本，每次运行脚本时，确定要采用何种环境，然后将对应环境的所有配置文件都覆盖当前的配置文件。对于Yii而言，与环境相关的文件其实就只有两个：一个是入口脚本 `index.php` 另一个就是各类配置文件。而且，Yii周全地想到了用 `.gitignore` 文件来解决配置文件进入不进入版本库的问题。

因此，在切换环境时，只需要一行命令就全部搞定:

```
php /path/to/yii-application/init
```

这行命令会提示你选择何种开发环境，并确认是否覆盖当前的配置文件。如果想更加高效，可以直接指定相关的参数:

```
php /path/to/yii-application/init --env=Production overwrite=All
```

第二种方式直接在命令行中指明使用的环境，并要要全部覆盖当前配置文件。

5.6 环境的配置原则

首先了解Yii各环境文件。前面我们讲到，每个Yii环境就是一组配置文件，包含了 `index.php` 和各类配置文件。其实他们就放在 `/path/to/yii-application/enviroments` 目录下面，我们看看这个目录都有哪些东西:

- 文件 `index.php`
- 目录 `dev`
- 目录 `prod`

其中， `index.php` 定义了可以使用的环境，打开这个文件看一眼，不用深入去理解，也可以大致猜到它定义了 `Development` `Production` 两个环境，聪明如你，肯定用脚都能猜得出来。目录 `dev` 和 `prod` 都包含了同样的结构:

- 文件 `yii`
- 目录 `common`
- 目录 `frontend`
- 目录 `backend`
- 目录 `console`

这几个目录的名称，与 `/path/to/yii-application/` 是一样一样的。再看看 `frontend` `backend` 的内容，里面内容是一样的： `config/main-local.php` `config/params-local.php` `web/index.php` 。因此，一个环境，就是一组 `index.php` 入口文件和 `*-local.php` 配置文件。注意 `common` 和 `console` 中并没有 `web` 目录，也就是没有入口脚本。这是因为 `common` 不是一个完备的应用，他只是提供了通用的配置，没有说前后台通用一个入口脚本的，那不彻底成了同一个应用了？而 `console` 是命令行，不是web应用，因此，没有web入口脚本一说，但是，上面提到的 `yii` 文件，就是命令行的入口文件他存在于目录 `dev` 和 `prod` 中，也就是说，不同的环境对应了不同的命令行入口脚本。

说了这么多，现在串起来看。运行 `init` 脚本就会将某一环境的系列文件复制到当前的文件中，这些文件就是 `index.php` 入口文件和 `*-local.php` 配置文件。复制到哪呢？复制到了

/path/to/yii-application/ 下面的 frontend backend console common 中对应的 config 目录和 index.php 。

这么作会覆盖当前的配置。那么，当前配置是指哪些文件呢？就是 /path/to/yii-application/ 下面的 frontend backend console common 中对应的 config 目录中的所有 *.php 和 index.php 。

*.php 文件有哪些呢？有表示路径别名的 aliases.php ，表示主配置的 main.php main-local.php ，表示全局参数的 params.php params-local.php 。

其中，所有的 *-local.php 都来自于你选用的环境，表示本地配置的意思。他们不会被写入到代码仓库中。当然，这些环境，也就是整个 /path/to/yii-application/enviroments 目录都会被写入代码仓库。

而所有不带 *-local.php 的main和params配置文件，都不是环境的内容。但在最终的运行环境中，他们是起作用的。比如，对于开发团队的所有成员而言，应用的ID是不会变的，像本教程的 YiiBlog 这个可以写在 main.php 文件中。但是每个成员的测试环境是不同的，如数据库名称，用户名，密码等，这个要写入自己的 /path/to/yii-application/enviroments 目录中的 main-local.php 文件。这样，当提交代码时，个人环境的配置并不会写入团队的代码库中。而拉取代码时，只要运行一行 init 命令就可以完成个人环境的切换。很简单不是么?

上面讲到的配置文件有很多，有前台、后台、命令行和Common的，有带local的、不带local的，有aliases, params, main等，看起来好复杂的样子。那么一个环境发生作用时，这些文件是怎么个顺序呢?

首先，前台、后台和命令行的配置文件间，互不干扰，各管各的。没有先后顺序一说。因为Yii在任意时间，要么是在跑前台，要么是在跑后台，记得么？他们是不同的应用，他们是独立的。但是，只里有个Common，通用。那么顺序是这样的，common的内容被其余的覆盖。

其次，local和不带local的。明显的，local的是本地配置文件，不带local的是团队间通用的配置。因此，local的覆盖不带local的。

最后，aliases, params, main。这三类文件表示的配置内容并不重叠，他们逻辑上不存在谁覆盖谁的问题。如果看看源代码，可以发现，aliases是使用PHP的 require 写进去的，对于全局起作用。其次，params只是main配置的一部分。而main的内容，是作为参数传递给应用的构造函数。这三者风马牛不相及，不存在谁覆盖谁的问题。

小结一下，Yii应用配置文件生效的顺序是:

- 前台、后台、命令行互不影响;
- 前台、后台、命令行覆盖common;
- aliases和main互不影响;
- params是main的一部份;
- 带local的覆盖不带local的。

5.7 检验安装情况

前面，你已经完成了Yii的安装，可能使用了Composer，也可能使用了压缩包。接着，你配置好了Web服务器。最后，你运行了 init 命令。那么，YiiBlog的框架已经搭建好了。你可在你的浏览器中试试效果。