深入理解 Yii2.0

Linuor

March 16, 2015

Contents

CHAPTER 1

导读

《深入理解 Yii2.0》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、新理念、新模式。采用的方式是分析框架的源代码，尝试从根上进行理解和阐述，并融入个人使用 Yii 开发的一些经验和教训。通过本书，你将不仅仅了解到 Yii 怎么使用的实操技巧，还将掌握其实现的技术原理和内幕。更为重要的是，接触当前 Web 开发中最为流行又相对成熟的设计模式和开发思路。衷心希望读者朋友们通过本书能有所收获。

本书将随着 Yii 官方对 Yii2 的开发而不断更新、丰富内容，读者朋友们可以订阅《深入理解 Yii2.0》的 RSS Feed[1]，第一时间了解本书的最近更新。同时，建议收藏《深入理解 Yii2.0》网站[2]，以便今后访问。

由于本人的水平有限，技术欠精，写的内容难免有纰漏，事实上已经有许多 supporter 指出了书中的错误，我们都及时进行更正。一如既往地欢迎和感谢读者朋友们通过每个页面底部的评论区，把阅读过程中的疑问、需求、批评、建议与我们进行分享，帮助我们创作更好的内容。

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 中刚刚实现的命名空间等 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、

[1] http://www.digpage.com/rss.xml
[2] http://www.digpage.com/index.html

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开发而言，肯定是充分、够用的了。

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最重要的特点。对于框架的使用者，也就是开发人员来讲，更重要的是开发效率。由于 Yii 架构合理，Web 开发中常用的思路和模式都可以很顺地套上使用。在 Web 开发中经常遇到的一些细节上的问题，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。开发效率高，对于开发者、开发团队而言，更为重要。

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 Yii2.0 **的亮点**

Yii 有两个最主要的版本：Yii1.1 和 Yii2.0。Yii1.1 是老的版本，在写这本书时，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，吸收了许多当前最新的技术和开发中的主流约定，是最新一代 Web 开发框架的代表。附录 1：Yii2.0 对比 Yii1.1 的重大改进 部分介绍了 Yii2.0 对比 Yii1.1 的重大改进。下面我们简单介绍 Yii2.0 的一些比较突出的特点：

- 运用了 PHP 命名空间、Trait、PSR 标准[3] 、Composer 和 Bower 包管理器，等新技术新标准。
- 实现了依赖注入和依赖注入容器 以及服务定位器（Service Locator） 等新架构新模式。
- Yii2.0 格外重视安全性，采取一系列手段有效防止 SQL 注入、XSS 攻击、CSRF 攻击、cookie 篡改等。
- 广泛支持各类 SQL 和 NOSQL 数据库，高效实现了 Active Record 等数据库查询、操作界面，提供数据库迁移、复制、读写分离等功能。
- 只需极少量的代码就可以实现完全符合标准的 RESTful API。
- 支持各种粒度、介质的缓存机制。
- 提供多种认证和授权手段基于 cookie 和基于令牌的认证，RBAC 等权限控制手段，支持 OpenID，OAuth1，OAuth2 等。
- 支持 Bootstrap，jQuery UI，提供了丰富的 Widget 挂件供使用。
- 完善的国际化支持，提供符合 ICU 标准[4] 的时间、复数等格式化工具和消息翻译、视图翻译等功能。
- 除了支持 Twig 和 Smarty 2 个主流的 PHP 模版引擎外，开发者还可以自己写扩展支持其他引擎。
- 为苦命码农提供了 Yii 调试工具条[5] 和 Gii 代码生成器[6] 以及 文档生成器[7] 等高效开发工具。

[3] http://www.php-fig.org/psr/
[4] http://icu-project.org/apiref/icu4c/classMessageFormat.html
[5] http://www.yiiframework.com/doc-2.0/guide-tool-debugger.html
[6] http://www.yiiframework.com/doc-2.0/guide-tool-gii.html
[7] https://github.com/yiisoft/yii2/tree/master/extensions/apidoc

- 集成了 Codeception 和 Faker，并与 DB Migration 相结合，提供了一个 fixture 框架，方便测试开发。
- 提供了一个简单应用模板和高级应用模板，适用于不同的开发场景，都可以帮助开发者尽快搭建起自己的应用。

详细的 Yii2.0 功能特性，请查看 官方说明[8] 。

1.3 背景知识

请留意，虽然本书以 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，但在讲解过程中，本书会帮助没有 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，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，本书也会在涉及到时，进行专门讲解。

1.4 如何阅读本书

这不是一本“快速入门”、“一周精通”的书，本书的一个非常重要的目的和出发点是剖析 Yii2.0 的原理，使读者既知其然，又知其所以然，这就注定了不可能一蹴而就。书中有大量的代码，需要剖析数据结构、追溯调用堆栈、跟踪代码流程。虽然我们力求生动、简洁，但读者朋友们还是要有“古佛青灯”的心理准备，不浮、不躁，潜心练好内功，扎实锤炼底子，早日练成绝活。

在内容上，本书直指 Yii2.0 的本质，求精不求全，对于相关的 PHP、HTTP 等诸多知识最多在讲到时点一点，浅尝辄止，没能够展开介绍。因此，对于涉及到的关联知识、背景知识，要充分利用搜索引擎等工具，进行自学和补充。

在篇章结构上，本书总体上按照逐步深入的格局安排内容。同时，各部分的内容又保持相对独立，尽量做到每个单独页面的内容，都可独立成篇。因此，初次接触 Yii2.0 的读者朋友，建议循序渐进，依次阅读。而具有一定基础的读者则可以根据个人兴趣爱好，自由选择学习切入点，在遇到有疑问的知识点时，再按图索骥查找相关的页面。

在Yii 基础 部分，依次介绍了属性（Property），事件（Event），行为（Behavior）等 Yii 中最基础的知识，是理解整个 Yii 框架的最基本的概念。

[8] http://www.yiiframework.com/news/81/yii-2-0-0-is-released/

在Yii 约定 部分，主要讲解了 Yii 约定俗成的一些套路、设定，解决的是在开发者未作任何指定的情况下，Yii 的默认行为方式的问题，用于加深对 Yii 实际使用的理解。这一部分主要包括Yii 应用的目录结构和入口脚本，别名 (Alias)，Yii 的类自动加载机制，环境和配置文件，配置项（Configuration） 等内容。

在Yii 模式 部分，剖析了 Yii 是如何实现一些当前 Web 开发中最主流和成熟的设计模式。学习这些设计模式，有助于深入理解 Yii 的机制，更难得的是提高读者朋友自身的开发设计水平。这一部分主要讲了MVC，依赖注入和依赖注入容器，服务定位器（Service Locator） 三种设计模式。

后续我们还将创作更多的内容，敬请期待。

CHAPTER 2

Yii 基础

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。

这些都是 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。因此，我们放在最前面来讲。

2.1 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。但是，成员变量是就类的结构构成而言的概念，而属性是就类的功能逻辑而言的概念，两者紧密联系又相互区别。比如，我们说 People 类有一个成员变量 int $age，表示年龄。那么这里年龄就是属性，$age 就是成员变量。

再举个更学术化点的例子，与非门:

```
1  class NotAndGate extends Object{
2      private $_key1;
3      private $_key2;
4
5      public function setKey1($value){
6          $this->_key1 = $value;
7      }
8
9      public function setKey2($value){
10         $this->_key2 = $value;
11     }
12
13     public function getOutput(){
14         if (!$this->_key1 || !$this->_key2)
15             return true;
```

```
        else if ($this->_key1 && $this->_key2)
            return false;
    }
}
```

与非门有两个输入，当两个输入都为真时，与非门的输出为假，否则，输出为真。上面的代码中，与非门类有两个成员变量，$_key1 和 $_key2 。但是有 3 个属性，表示 2 个输入的 key1 和 key2 ，以及表示输出的 output 。

成员变量和属性的区别与联系在于：

- 成员变量是一个“内”概念，反映的是类的结构构成。属性是一个“外”概念，反映的是类的逻辑意义。
- 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，或可读可写。
- 成员变量不对读出作任何后处理，不对写入作任何预处理，而属性则可以。
- public 成员变量可以视为一个可读可写、没有任何预处理或后处理的属性。而 private 成员变量由于外部不可见，与属性“外”的特性不相符，所以不能视为属性。
- 虽然大多数情况下，属性会由某个或某些成员变量来表示，但属性与成员变量没有必然的对应关系，比如与非门的 output 属性，就没有一个所谓的 $output 成员变量与之对应。

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:

```
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) . '::' . $name);
    } else {
        throw new UnknownPropertyException('Getting unknown property: '
            . get_class($this) . '::' . $name);
    }
}

// $name 是属性名，$value 是拟写入的属性值
public function __set($name, $value)
{
    $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) . '::' . $name);
    } else {
        throw new UnknownPropertyException('Setting unknown property: '
            . get_class($this) . '::' . $name);
    }
}
```

2.1.1 实现属性的步骤

我们知道，在读取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:

```
class Post extends 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，那么毫无疑问，每个写入语句都要调用 trim() 。你能保证没有一处遗漏?

因此，使用 public $title 只是一时之快，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。

但是，世事无绝对。由于 __get() 和 __set() 是在遍历所有成员变量，找不到匹配的成员变量时才被调用。因此，其效率天生地低于使用成员变量的形式。在一些表示数据结构、数据集合等简单情况下，且不需读写控制等，可以考虑使用成员变量作为属性，这样可以提高一点效率。

另外一个提高效率的小技巧就是：使用 $pro = $object->getPro() 来代替 $pro = $object->pro ，用 $objcect->setPro($value) 来代替 $object->pro = $value 。这在功能上是完全一样的效果，但是避免了使用 __get() 和 __set() ，相当于绕过了遍历的过程。

这里估计有人该骂我了，Yii 好不容易实现了属性的机制，就是为了方便开发者，结果我却在这里教大家怎么使用原始的方式，去提高所谓的效率。嗯，确实，开发的便利性与执行高效率存在一定的矛盾。我个人的观点更倾向于以便利为先，用好、用足 Yii 为我们创造的便利条件。至于效率的事情，更多的是框架自身需要注意的，我们只要别写出格外 2 的代码就 OK 了。

不过你完全可以放心，在 Yii 的框架中，极少出现 $app->request 之类的代码，而是使用 $app->getRequest() 。换句话说，框架自身还是格外地注重效率的，至于便利性，则留给了开发者。总之，这里只是点出来有这么一个知识点，至于用不用，怎么用，完全取决于你了。

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。即属性名也是不区分大小写的。
- 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。

2.1.2 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。只要定义了 getter，属性即可读。同时，如果 $checkVars 为 true 。那么只要类定义了成员变量，不管是 public，private 还是 protected，都认为是可读。
- canSetProperty() 测试一个属性是否可写，参数 $checkVars 的意义同上。只要定义了 setter，属性即可写。同时，在 $checkVars 为 ture 。那么只要类定义了成员变量，不管是 public，private 还是 protected，都认为是可写。

2.1.3 Object **和** Component

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。

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 。

在 Yii1.1 时，就已经有了 component 了，那时是 CComponent。Yii2 将 Yii1.1 中的 CComponent 拆分成两个类：yii\base\Object 和 yii\base\Component 。

其中，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。因此，Component 在 Yii 中的地位极高。

在提供更多功能、更多便利的同时，Component 由于增加了 event 和 behavior 这两个特性，在方便开发的同时，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，比如表示一些数据的类。那么，可以不从 Component 继承，而从 Object 继承。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。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。

2.1.4 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:

```
$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);
```

$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:

```
public function __construct($config = [])
{
    if (!empty($config)) {
        Yii::configure($this, $config);
    }
    $this->init();
}
```

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:

```
public static function configure($object, $properties)
{
    foreach ($properties as $name => $value) {
        $object->$name = $value;
    }

    return $object;
}
```

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 。

- 为对象属性提供 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，其实是同质的。

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 Yii::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。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 validation
        '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',
    ],
],
```

这是一个典型嵌套配置数组。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，而是放在父类 yii\di\ServiceLocator 里面。至于 ServiceLocator 是何方神圣，在后面服务定位器（Service Locator）部分会讲到，这里你只需要知道它是 Application 的父类，提供 components 属性的 setter 方法就可以了：

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

这里有个成员函数，$this->set()，他是服务定位器用来注册服务的方法。我们暂时不讲这个东西，留待服务定位器（Service Locator）部分再讲。现在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。

从 yii\base\Object::__construct() 来看，对于所有 Object，包括 Component 的属性，都经历这么 4 个阶段：

1. 预初始化阶段。这是最开始的阶段，就是在构造函数 __construct() 的开头可以设置 property 的默认值。
2. 对象配置阶段。也就是前面提到构造函数调用 Yii::configure($this, $config) 阶段。这一阶段可以覆盖前一阶段设置的 property 的默认值，并补充没有默认值的参数，也就是必备参数。$config 通常由外部代码传入或者通过配置文件传入。
3. 后初始化阶段。也就是构造函数调用 init() 成员函数。通过在 init() 写入代码，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，由类的构造函数一口气调用的。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，不存在不确定的 property。所有的属性要么是默认值，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，可以在特定的时点，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:

```
1 $person = new Person;
2
3 // 使用 PHP 全局函数作为 handler 来进行绑定
4 $person->on(Person::EVENT_GREET, 'person_say_hello');
5
6 // 使用对象 $obj 的成员函数 say_hello 来进行绑定
7 $person->on(Person::EVENT_GREET, [$obj, 'say_hello']);
8
9 // 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。原理可以参见配置项（Configuration） 部分的内容。

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:

```
$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:

```
// 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, $data] 的数组，如$_event[] 数组的数据结构示意图 所示。

在事件的绑定逻辑上，按照以下顺序:

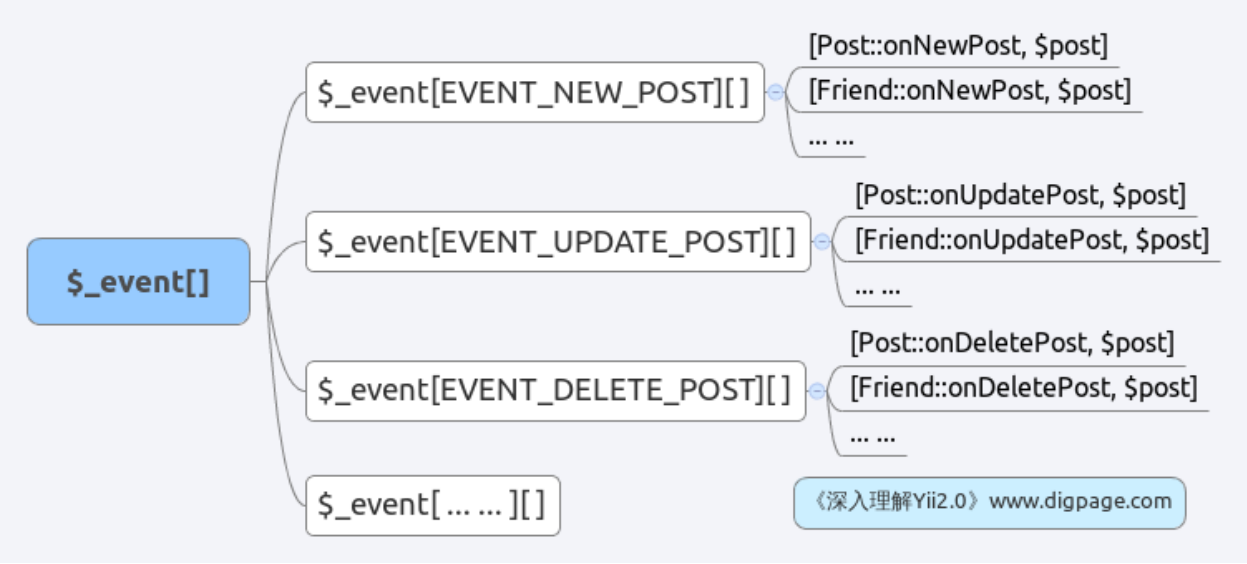

Fig. 2.1: $_event[] 数组的数据结构示意图

- 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:

```
1  public function off($name, $handler = null)
2  {
3      $this->ensureBehaviors();
4      if (empty($this->_events[$name])) {
5          return false;
6      }
7
8      // $handler === null 时解除所有的 handler
9      if ($handler === null) {
```

```
        unset($this->_events[$name]);
        return true;
    } else {
        $removed = false;

        // 遍历所有的 $handler
        foreach ($this->_events[$name] as $i => $event) {
            if ($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;
```

```
13
14          // 遍历 handler 数组，并依次调用
15          foreach ($this->_events[$name] as $handler) {
16              $event->data = $handler[1];
17
18              // 使用 PHP 的 call_user_func 调用 handler
19              call_user_func($handler[0], $event);
20
21              // 如果在某一 handler 中，将 $evnet->handled 设为 true,
22              // 就不再调用后续的 handler
23              if ($event->handled) {
24                  return;
25              }
26          }
27      }
28      Event::trigger($this, $name, $event);   // 触发类一级的事件
29  }
```

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:

```
1   abstract class Application extends Module
2   {
3       // 定义了两个事件
4       const EVENT_BEFORE_REQUEST = 'beforeRequest';
5       const EVENT_AFTER_REQUEST = 'afterRequest';
6
7       public function run()
8       {
9           try {
10
11              $this->state = self::STATE_BEFORE_REQUEST;
12
13              // 先触发 EVENT_BEFORE_REQUEST
14              $this->trigger(self::EVENT_BEFORE_REQUEST);
15
16              $this->state = self::STATE_HANDLING_REQUEST;
17
18              // 处理 Request
19              $response = $this->handleRequest($this->getRequest());
20
21              $this->state = self::STATE_AFTER_REQUEST;
22
23              // 处理完毕后触发 EVENT_AFTER_REQUEST
24              $this->trigger(self::EVENT_AFTER_REQUEST);
```

```
25
26          $this->state = self::STATE_SENDING_RESPONSE;
27          $response->send();
28
29          $this->state = self::STATE_END;
30
31          return $response->exitStatus;
32
33      } catch (ExitException $e) {
34
35          $this->end($e->statusCode, isset($response) ? $response : null);
36          return $e->statusCode;
37
38      }
39  }
40 }
```

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:

```
1  // 定义事件的关联数据
2  class MsgEvent extend yii\base\Event
3  {
4      public $dateTime;   // 微博发出的时间
5      public $author;     // 微博的作者
6      public $content;    // 微博的内容
7  }
8
9  // 在发布新的微博时，准备好要传递给 handler 的数据
10 $event = new MsgEvent;
11 $event->title = $title;
12 $event->author = $auhtor;
13
14 // 触发事件
15 $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，你指定新发微博的事件 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，国家安监局对某个煤矿进行监管，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，这个小煤窑的老板，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)
```

```
    {
        // 将事件标记为已经处理完毕，阻止后续事件 handler 介入。
        $event->handled = true;
    }
}
```

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);

// 删除一个 PHP 全局函数的 handler
$coal->off(Coal::EVENT_DISASTER, 'global_onDisaster');

// 删除一个对象的成员函数的 handler
$coal->off(Coal::EVENT_DISASTER, [$baddy, 'onDisaster']);

// 删除一个类的静态成员函数的 handler
$coal->off(Coal::EVENT_DISASTER, ['path\to\Baddy', 'static_onDisaster']);

// 删除一个匿名函数的 handler
$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，对于数百个工人，即数百个 Worker 实例，工头难道要一个一个去绑定自己的 handler 么？这也太低级了吧？其实，他只需要绑定一个 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()

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

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，你就可以向小煤窑老板那样，通过 $event->handled = true ，来终止事件处理。

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：

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

// Event 中的 $_event[] 数组
$_event[$eventName][$calssName][] = [$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，指向的是类名。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 。

同时，行为单独靠 Behavior 一方是实现不了的，就好像爱情不是一厢情愿。为了支持 Behavior，Yii 对于 yii\base\Component 也进行了精心设计，这两者共同配合，才有了神奇的行为。

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.';
```

```
12
13      // 行为的一个方法
14      public function method1()
15      {
16          return 'Method in MyBehavior is called.';
17      }
18  }
19
20  $myClass = new MyClass();
21  $myBehavior = new MyBehavior();
22
23  // Step 3: 将行为绑定到类上
24  $myClass->attachBehavior('myBehavior', $myBehavior);
25
26  // 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
27  echo $myClass->property1;
28  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，表现的财大气粗，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 行为的要素

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:

```
1   class Behavior extends Object
2   {
3       // 指向行为本身所绑定的 Component 对象
4       public $owner;
5
6       // Behavior 基类本身没用，主要是子类使用，重载这个函数返回一个数组表
7       // 示行为所关联的事件
8       public function events()
```

```
9     {
10        return [];
11    }
12
13    // 绑定行为到 $owner
14    public function attach($owner)
15    {
16        ... ...
17    }
18
19    // 解除绑定
20    public function detach()
21    {
22        ... ...
23    }
24 }
```

这就是 Behavior 的全部代码了，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() 用于表示行为所有要响应的事件；
- 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。

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 来访问 Component。

行为所要响应的事件

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。为此，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:

```
namespace app\Components;

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

class MyBehavior extends Behavior
{
    // 重载 events() 使得在事件触发时，调用行为中的一些方法
    public function events()
    {
        // 在 EVENT_BEFORE_VALIDATE 事件触发时，调用成员函数 beforeValidate
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    // 注意 beforeValidate 是行为的成员函数，而不是绑定的类的成员函数。
    // 还要注意，这个函数的签名，要满足事件 handler 的要求。
    public function beforeValidate($event)
    {
        // ...
    }
}
```

上面的代码中，events() 返回一个数组，表示所要做出响应的事件，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事件 handler，上例中是 beforeValidate()，事件 handler 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 中使用字符串时，是表示 PHP 全局函数，而这里表示行为类内部的方法。
- 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。

对于事件响应函数的签名，要求与事件 handler 一样:

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

具体内容，请参考事件（Event） 的内容。

行为的绑定与解除

说到绑定与解除，这意味着这个事情有 2 方，行为和 Component。单独一方是没有绑定或解除的说法的。因此，这里我们先卖一关子，等后面讲绑定和解除的原理时，再来讲有关的内容。

这里你只需要知道，对于绑定和解除，Behavior 分别使用 attach() 和 detach() 来实现就 OK 了。

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->_prop3 = $value;
    }

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

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

上面的代码通过定义一个 app\Components\MyBehavior 类而定义一个行为。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() 。另外，还有一个 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（Property） 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() ，因为他们都是 public 的。而 private 的 $_prop4 和 protected 的 bar 就得不到了。至于原因么，后面讲行为注入的原理时，我们再解释。

行为的绑定

行为的绑定通常是由 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。

静态方法绑定行为

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:

```php
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',
        ]
    ];
}
}
```

还有一个静态的绑定办法，就是通过配置文件来绑定:

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

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

具体参见配置项（Configuration） 部分的内容。

动态方法绑定行为

动态绑定行为，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。

在上面的这些例子中，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。

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();
```

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()

4 个方法中，Behavior 只占其一，更多的代码，是在 Component 中完成的。

yii\base\Component::behaviors() 上面讲静态方法绑定行为时已经提到了，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？就像名字所表明的，他的作用在于“ensure”。其实只是确保 behaviors() 中所描述的行为已经进行了绑定而已：

```
1  public function ensureBehaviors()
2  {
3      // 为 null 表示尚未绑定
4      // 多说一句，为空数组表示没有绑定任何行为
5      if ($this->_behaviors === null) {
6          $this->_behaviors = [];
7
8          // 遍历 $this->behaviors() 返回的数组，并绑定
9          foreach ($this->behaviors() as $name => $behavior) {
10             $this->attachBehaviorInternal($name, $behavior);
11         }
12     }
13 }
```

这个方法主要是对子类用的，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。

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      // 不是 Behavior 实例，说是只是类名、配置数组，那么就创建出来吧
```

```
    if (!($behavior instanceof Behavior)) {
        $behavior = Yii::createObject($behavior);
    }

    // 匿名行为
    if (is_int($name)) {
        $behavior->attach($this);
        $this->_behaviors[] = $behavior;

    // 命名行为
    } else {

        // 已经有一个同名的行为，要先解除，再将新的行为绑定上去。
        if (isset($this->_behaviors[$name])) {
            $this->_behaviors[$name]->detach();
        }
        $behavior->attach($this);
        $this->_behaviors[$name] = $behavior;
    }
    return $behavior;
}
```

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()，这也是前面我们讲行为的要素时没讲完的。先看看代码：

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

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。

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。不过我们可以一不做二不休，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() ，其代码如下：

```
public function detach()
{
    // 这得是个名花有主的行为才有解除一说
    if ($this->owner) {

        // 遍历行为定义的事件，一一解除
        foreach ($this->events() as $event => $handler) {
            $this->owner->off($event, is_string($handler) ? [$this,
                $handler] : $handler);
        }
        $this->owner = null;
    }
}
```

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。

行为响应的事件实例

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\Behavior::events() 方法。

上面 Behavior 基类的代码中，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。我们以这个行为为例，来分析行为响应事件的原理。

在 yii\behaviors\AttributeBehavior::event() 中，代码如下:

```
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，数组元素的值为成员函数 evaluateAttributes 。

而在 yii\behaviors\TimeStampBehavior::init() 中，有以下的代码:

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

    if (empty($this->attributes)) {
        // 重点看这里
        $this->attributes = [
            BaseActiveRecord::EVENT_BEFORE_INSERT =>
                [$this->createdAtAttribute, $this->updatedAtAttribute],
            BaseActiveRecord::EVENT_BEFORE_UPDATE =>
                $this->updatedAtAttribute,
        ];
    }
}
```

上面的代码重点看的是对于 $this->attributes 的初始化部分。结合上面 2 个方法的代码，对于 yii\base\Behavior::events() 的返回数组，其格式应该是这样的:

```
return [
    BaseActiveRecord::EVENT_BEFORE_INSERT => 'evaluateAttributes',
    BaseActiveRecord::EVENT_BEFORE_UPDATE => 'evaluateAttributes',
];
```

数组的键值用于指定要响应的事件，这里是 BaseActiveRecord::EVENT_BEFORE_INSERT 和 BaseActiveRecord::EVENT_BEFORE_UPDATE 。数组的值是一个事件 handler，如上面的 evaluateAttributes 。

那么一旦 TimeStampBehavior 与某个 ActiveRecord 绑定，就会调用 yii\behaviors\TimeStampBehavior::attach() ，那么就会有:

```
// 这里 $owner 是某个 ActiveRecord
public function attach($owner)
{
    $this->owner = $owner;

    // 遍历上面提到的 events() 所定义的数组
    foreach ($this->events() as $event => $handler) {

        // 调用 ActiveRecord::on 来绑定事件
        // 这里 $handler 为字符串 `evaluateAttributes`
        // 因此，相当于调用 on(BaseActiveRecord::EVENT_BEFORE_INSERT,
        // [$this, 'evaluateAttributes'])
        $owner->on($event, is_string($handler) ? [$this, $handler] :
            $handler);
    }
}
```

因此，事件 BaseActiveRecord::EVENT_BEFORE_INSERT 和 BaseActiveRecord::EVENT_BEFORE_UPDATE 就绑定到了 ActiveRecord 上了。当新建记录或更新记录时，TimeStampBehavior::evaluateAttributes 就会被触发。从而实现时间戳的功能。具体可以看看 yii\behaviors\AttributeBehavior::evaluateAttributes() 和 yii\behaviors\TimeStampBehavior::getValues() 的代码。这里因为只是具体功能实现，对于行为的理解关系不大。就不把代码粘出来占用篇幅了。

2.3.4 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)) {

                // 属性在行为中须为 public。否则不可能通过下面的形式访问呀。
                return $behavior->$name;
            }
        }
    }
    if (method_exists($this, 'set' . $name)) {
        throw new InvalidCallException('Getting write-only property: ' .
            get_class($this) . '::' . $name);
    } else {
        throw new UnknownPropertyException('Getting unknown property: ' .
            get_class($this) . '::' . $name);
    }
}
```

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() 。这个方法已经在前面讲过了，主要是确保行为已经绑定。
- 在确保行为已经绑定后，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，通过行为的 canGetProperty() 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 。完成属性的读取。至于 canGetProperty() 已经在:ref::property 部分已经简单讲过了，后面还会有针对性地一个介绍。

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 还是先是调用了 $this->ensureBehaviors() 确保行为已经绑定。

然后，也是遍历 yii\base\Component::$_behaviros[] 数组。通过 hasMethod() 方法判断方法是否存在。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。至于 hasMethod() 方法，我们后面再讲。

注入属性与方法的访问控制

在前面我们针对行为中 public 和 private、protected 的成员在所绑定的类中是否可访问举出了具体例子。这里我们从代码层面解析原因。

在上面的内容，我们知道，一个属性可不可访问，主要看行为的 canGetProperty() 和 canSetProperty()。而一个方法可不可调用，主要看行为的 hasMethod()。由于 yii\base\Behavior 继承自我们的老朋友 yii\base\Object，所以上面提到的三个判断方法，事实上代码都在 Object 中。我们一个一个来看:

```
public function canGetProperty($name, $checkVars = true)
{
    return method_exists($this, 'get' . $name) || $checkVars &&
        property_exists($this, $name);
}

public function canSetProperty($name, $checkVars = true)
{
    return method_exists($this, 'set' . $name) || $checkVars &&
        property_exists($this, $name);
}

public function hasMethod($name)
{
```

```
15      return method_exists($this, $name);
16  }
```

这三个方法真的谈不上复杂。对此，我们可以得出以下结论：

- 当向 Component 绑定的行为读取（写入）一个属性时，如果行为为该属性定义了一个 getter (setter)，则可以访问。或者，如果行为确实具有该成员变量即可通过上面的判断，此时，该成员变量可为 public, private, protected。但最终只有 public 的成员变量才能正确访问。原因在上面讲注入的原理时已经交待了。
- 当调用 Component 绑定的行为的一个方法时，如果行为已经定义了该方法，即可通过上面的判断。此时，这个方法可以为 public, private, protected。但最终只有 public 的方法才能正确调用。如果你理解了上一款的原因，那么这里也就理解了。

2.3.5 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 3

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，主要包含应用的目录结构、别名、自动加载机制、环境、对象配置等内容：

3.1 Yii 应用的目录结构和入口脚本

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
-- yii.bat
.
```

```
-- backend
-- common
-- console
-- environments
-- frontend
-- nbproject
-- tests
-- vendor
-- composer.json
-- composer.lock
-- init
-- init.bat
-- LICENSE.md
-- README.md
-- requirements.php
-- yii
-- 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 应用的目录结构。

3.1.1 公共目录

这里的公共目录可不止 common 目录，但这个目录从字面上来看，是所有公共目录里最“公共”的。common 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：

```
.
-- config
-- mail
-- models
```

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。

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，就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\digpage 目录下写代码。

3.1.2 前台的目录结构

其实，前台和后台是一样的，只是我们逻辑上的一个划分。典型的，frontend 具有如下的一个目录结构：

```
.
-- assets
-- config
-- controllers
-- models
-- runtime
-- views
-- web
-- widgets
```

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 等前端资源就可以了。
- config 用于存放本应用的配置文件，包含主配置文件 main.php 和全局参数配置文件 params.php 。
- models views controllers 3 个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 这个目录是要求权限为 chmod 777 ，即允许 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，目的在于分解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。

3.1.3 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/bootstrap.php');
require(__DIR__ . '/../config/bootstrap.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();
```

设置调试模式和代码环境

前两行是两个 define **语句**:: defined('YII_DEBUG') or define('YII_DEBUG', true); defined('YII_ENV') or define('YII_ENV', 'dev');

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 表示产品环境。

这些环境只是一个名称，具体的意义和环境内容要看环境的定义。dev prod 是 Yii 安装后默认的两个环境，分别表示开发环境和最终的产品环境。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) {
    // 以下配置项仅在测试环境中起作用
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

其实，这个 YII_ENV 的定义就是一个与 init 脚本环境切换的相互补充。如果各环境比较明晰，用 init 来切换各种环境的配置是完全够用的。不必在脚本中再有如 YII_ENV_TEST 之类的判断语句，这会使本来已经显得很长的配置文件看上去更臃肿。

引入必要的文件

紧接着两个 define 语句之后，就是 4 个 require 语句：

```
require(__DIR__ . '/../../vendor/autoload.php');
require(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
require(__DIR__ . '/../../common/config/bootstrap.php');
require(__DIR__ . '/../config/bootstrap.php');
```

这 4 个语句的前 3 个中都使用到了相对于当前目录的叔伯目录中的 php 文件，第 4 个语句使用了相对于当前目录的兄弟目录。

我们知道，__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。若 index.php 的当前目录是 /path/to/digpapge.com/frontend/web，那么爸爸目录就是 frontend，爷爷目录就是 digpage.com 了。

第一个 require 引入了 /path/to/digpage.com/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。

第二个引入了 vendor 下面的 yiisoft/yii2/Yii.php，这是 Yii 的工具类文件。引入了这个类文件后，才能使用 Yii 的提供的各种工具，才有 Yii::createObject() Yii::$app 之类的东东可以使用。

第三个引入了 /path/to/digpage.com/common 下面的 config/bootstrap.php 。这个文件主要用于执行一些 Yii 应用引导的代码，比如定义一系列的路径别名：

```
<?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');
```

第四个 require 引入了 path/to/digpage.com/frontend 下面的 config/bootstrap.php 。作用与上面类似，只是其中的代码仅适用于当前应用（frontend）。而第三个 require 中的，是适应于所有应用（common）。

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、合并应用的各配置文件并保存在 $config 变量中。这里我们看到一共是读取了 4 个配置文件:

```
require('path/to/digpage.com/common/config/main.php'),
require('path/to/digpage.com/common/config/main-local.php'),
require('path/to/digpage.com/frontend/config/main.php'),
require('path/to/digpage.com/frontend/config/main-local.php')
```

依次是通用目录 common 下的 2 个配置文件，和当前应用 frontend 下的 2 个配置文件。在优先顺序上，当前的配置覆盖通用的配置。同时，带有 -local 的配置文件在后，所以，本地配置文件覆盖团队配置文件。

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。

3.1.4 命令行应用入口脚本

命令行应用的入口脚本是 path/to/digpage.com/yii 文件。这个文件被 init 脚本设为可执行的。他的内容如下:

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

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

// fcgi doesn't have STDIN and STDOUT defined by default
defined('STDIN') or define('STDIN', fopen('php://stdin', 'r'));
defined('STDOUT') or define('STDOUT', fopen('php://stdout', 'w'));

require(__DIR__ . '/vendor/autoload.php');
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');
require(__DIR__ . '/common/config/bootstrap.php');
require(__DIR__ . '/console/config/bootstrap.php');

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

```
);

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

对比于 Web 应用的 index.php 入口脚本，yii 并没有太多的新东西，其中核心的东西根本就没变。

我们先来看看这个这个 yii 是什么？

首先，它没有扩展名，我们不好知道其具体类型。但是从文件内容的第一行 #!/usr/bin/env php 来看，这是一个 bash 脚本。第一行在告诉 bash，也在告诉我们，这是一个使用 PHP 运行的脚本。

但第二行的 <?php 又清楚的向我们表明，这货其实也是个 PHP 文件，只是没有加上 PHP 后缀而已。

接下来，define('STDIN') 和 define('STDOUT') 则为 fcgi 定义了标准输入和标准输出。

在各 require 语句中，由于 yii 的位置与 index.php 不同，是位于应用根目录下，所以目录结构上更简单些。

最后，在 Yii 应用跑起来后，还要获取其返回值，并以该返回值退出脚本，通知操作系统退出时的状态。

对于 Windows 系统而言，命令行的入口脚本仍然是 yii，但是命令行下无法直接运行。所以，细心的 Yii 为我们准备了一个 yii.bat。这个文件会以 php yii 形式调用 PHP 来运行入口脚本。

3.2 **别名** (Alias)

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。

3.2.1 **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。别名的定义一般放在应用的最开始的阶段进行，比如引导阶段、初始化阶段等。这样可以保证后续代码可以使用这些定义好的别名。

配置文件中的别名

别名一般放在 digpage.com\common\config\bootstrap.php，或者 digpage.com\frontend\config\bootstrap.php 等 bootstrap.php 文件中定义。比如:

```
<?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');
```

在此定义的别名通过入口脚本引入 Yii 应用中，具体可以看看入口文件 index.php 部分的内容。上面的 bootstrap.php 文件定义了 @common，@frontend，@backend 和 @console 4 个别名。开发者也可以自己在 bootstrap.php 中加入自己的别名定义，这是最常运用的定义别名的方式。

Yii **预定义的别名**

相比较于通过 bootstrap.php 的方式定义别名，有的别名就不那么直观了，可能没法一下子看到。而且，这些别名相对比较固定，与 Yii 框架和应用自身息息相关。这类别名直接写到 Yii 的代码中去了。对于这类 Yii 框架预定义的别名，一般不去修改他。改他的收益远小于造成的副作用，这种得不偿失的亏本买卖，高智商人群是不会去干的。

这些预定义的别名，主要分布在 yii\BaseYii 和 yii\base\Application 等类中。

在 yii\BaseYii 中:

```
// 定义了 @yii 别名
public static $aliases = ['@yii' => __DIR__];
```

yii\BaseYii::$aliases 用于保存整个 Yii 应用的所有的别名。这里默认地把 yii\BaseYii.php 所在的目录作为 @yii 别名。

另外，对于 yii\base\Application 在其构造函数 __construct() 中，会调用以下代码:

```
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 required.');
    }

    // @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，就使用配置的值，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 5 个别名：@app，@vendor @bower @npm，@runtime。上面的代码中，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/digpage.com/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，与设置 basePath 类似，在配置 vendorPath runtimePath 时，Yii 会调用 setVendorPath() setRuntimePath()。如果未在配置文件中对这两个配置项作出设置，Yii 会调用 getVendorPath() 和 getRuntimePath()，这两个函数最终也会调用相应的 set 函数对这些别名进行定义。

@vendor，@bower，@npm 和 @runtime 这 4 个别名就由这两个 set 函数定义：

```
public function getVendorPath()
{
    // 在未设置 vendorPath 时，使用默认值
    if ($this->_vendorPath === null) {
        $this->setVendorPath($this->getBasePath() . DIRECTORY_SEPARATOR . 'vendor');
    }

    return $this->_vendorPath;
}

// 这里定义了 3 个别名
public function setVendorPath($path)
{
    $this->_vendorPath = Yii::getAlias($path);
```

```
    Yii::setAlias('@vendor', $this->_vendorPath);
    Yii::setAlias('@bower', $this->_vendorPath . DIRECTORY_SEPARATOR . 'bower');
    Yii::setAlias('@npm', $this->_vendorPath . DIRECTORY_SEPARATOR . 'npm');
}

public function getRuntimePath()
{
    // 在未设置 runtimePath 时，使用默认值
    if ($this->_runtimePath === null) {
        $this->setRuntimePath($this->getBasePath() . DIRECTORY_SEPARATOR . 'runtime');
    }

    return $this->_runtimePath;
}

// 这里定义了 @runtime 别名
public function setRuntimePath($path)
{
    $this->_runtimePath = Yii::getAlias($path);
    Yii::setAlias('@runtime', $this->_runtimePath);
}
```

对于上面的代码，默认情况下，会有：

- @app，必须由开发者在配置文件中提供，一般为配置文件的 dirname(__DIR__) 。即 digpage.com/frontend 之类的目录。
- @vendor，一般定义为 @app/vendor，高级模板中则定义为 @app/../vendor
- @bower，定义为 @vendor/bower
- @npm，定义为 @vendor/npm
- @runtime，定义为 @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。放在 frontend 或 backend 都不合适，干脆就地提拔吧。这一点我们在前面Yii 应用的目录结构和入口脚本 已经讲过了。

因此，实际上高级应用模版的 @vendor 应该是 @app/../vendor，上面的代码显然不适用。对此，贴心的 Yii 也已经考虑到了。在使用高级模板创建应用时，digpage.com/common/config/main.php 配置文件会重新设定 vendorPath

```
'vendorPath' => dirname(dirname(__DIR__)) . '/vendor'
```

这样就避免了使用代码中默认的值。

对于 Web 应用，yii\base\Web\Application 中又定义了 @webroot 和 @web 2 个别名:

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

    parent::bootstrap();
}
```

这里 @webroot 就是入口脚本 index.php 所在的目录。而 @web 则是 URL 别名，表示当前应用的根 URL 地址。

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作，是由 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);
            }
        }
        ... ...
    }
}
```

经过上面这些代码，我们的各种插件也有了自己的别名，如上面的 @yii\swiftmailer，@yii\gii 等，常见的还有 @yii\bootstrap 等。

所有预定义的别名

小结一下，默认预定义别名一共有 12 个，其中路径别名 11 个，URL 别名只有 @web 1 个:

- @yii 表示 Yii 框架所在的目录，也是 yii\BaseYii 类文件所在的位置;
- @app 表示正在运行的应用的根目录，一般是 digpage.com/frontend ;
- @vendor 表示 Composer 第三方库所在目录，一般是 @app/vendor 或 @app/../vendor ;
- @bower 表示 Bower 第三方库所在目录，一般是 @vendor/bower ;
- @npm 表示 NPM 第三方库所在目录，一般是 @vendor/npm ;
- @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，一般是 @app/runtime ;
- @webroot 表示正在运行的应用的入口文件 index.php 所在的目录，一般是 @app/web;
- @web URL 别名，表示当前应用的根 URL，主要用于前端;
- @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。

3.2.2 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 @ 到第一个 / 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);
```

```
16
17          // 检查是否有 $aliases[$root],
18          // 看看是否已经定义好了根别名。如果没有，则以 $root 为键，保存这个别名
19          if (!isset(static::$aliases[$root])) {
20              if ($pos === false) {
21                  static::$aliases[$root] = $path;
22              } else {
23                  static::$aliases[$root] = [$alias => $path];
24              }
25          // 如果 $aliases[$root] 已经存在，则替换成新的路径，或增加新的路径
26          } elseif (is_string(static::$aliases[$root])) {
27              if ($pos === false) {
28                  static::$aliases[$root] = $path;
29              } else {
30                  static::$aliases[$root] = [
31                      $alias => $path,
32                      $root => static::$aliases[$root],
33                  ];
34              }
35          } else {
36              static::$aliases[$root][$alias] = $path;
37              krsort(static::$aliases[$root]);
38          }
39
40      // 当传入的 $path 为 null 时，表示要删除这个别名。
41      } elseif (isset(static::$aliases[$root])) {
42          if (is_array(static::$aliases[$root])) {
43              unset(static::$aliases[$root][$alias]);
44          } elseif ($pos === false) {
45              unset(static::$aliases[$root]);
46          }
47      }
48  }
```

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

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

Yii::setAlias('@foo/bar', 'path/to/some/where');

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:

```
// 定义别名 @foo
Yii::setAlias('@foo', 'path/to/some/where');

// 删除别名 @foo
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:

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

// BaseYii::aliases['@qux'] = 'path/to/qux'
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:

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

// 直接覆盖 BaseYii::aliases['@foo'] = 'path/to/foo2'
Yii::setAlias('@foo', 'path/to/foo2');

/**
* 新增
* BaseYii::aliases['@foo'] = [
*     '@foo/bar' => 'path/to/foo/bar',
*     '@foo' => 'path/to/foo2',
* ];
*/
Yii::setAlias('@foo/bar', 'path/to/foo/bar');
```

```
15
16  // 初始 BaseYii::aliases['@bar'] = ['@bar/qux' => 'path/to/bar/qux'];
17  Yii::setAlias('@bar/qux', 'path/to/bar/qux');
18
19  // 直接覆盖 BaseYii::aliases['@bar'] = ['@bar/qux' => 'path/to/bar/qux2'];
20  Yii::setAlias('@bar/qux', 'path/to/bar/qux2');
21
22  /**
23  * 新增
24  * BaseYii::aliases['@bar'] = [
25  *     '@bar/foo' => 'path/to/bar/foo',
26  *     '@bar/qux' => 'path/to/bar/qux2',
27  * ];
28  */
29  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 和 @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：

```
1  public static function getAlias($alias, $throwException = true)
2  {
3      // 一切不以 @ 打头的别名都是无效的
4      if (strncmp($alias, '@', 1)) {
5          return $alias;
6      }
7
8      // 先确定根别名 $root
9      $pos = strpos($alias, '/');
10     $root = $pos === false ? $alias : substr($alias, 0, $pos);
11
12     // 从根别名开始找起，如果根别名没找到，一切免谈
13     if (isset(static::$aliases[$root])) {
14         if (is_string(static::$aliases[$root])) {
15             return $pos === false ? static::$aliases[$root] :
16                 static::$aliases[$root] . substr($alias, $pos);
17         } 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');
```

3.2.3 小结

回顾上面的内容，我们有这么几个要点：

- 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。

3.3 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[1] 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。

3.3.1 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：

```
1  <?php
2  defined('YII_DEBUG') or define('YII_DEBUG', false);
3  defined('YII_ENV') or define('YII_ENV', 'prod');
4
5  // 这个是第三方的 autoloader
6  require(__DIR__ . '/../../vendor/autoload.php');
7
8  // 这个是 Yii 的 Autoloader，放在最后面，确保其插入的 autoloader 会放在最前面
9  require(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
10 // 后面不应再有 autoloader 了
```

[1] http://www.php-fig.org/psr/psr-4/

```
11
12 require(__DIR__ . '/../../common/config/aliases.php');
13
14 $config = yii\helpers\ArrayHelper::merge(
15     require(__DIR__ . '/../../common/config/main.php'),
16     require(__DIR__ . '/../../common/config/main-local.php'),
17     require(__DIR__ . '/../config/main.php'),
18     require(__DIR__ . '/../config/main-local.php')
19 );
20
21 $application = new yii\web\Application($config);
22 $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 spl_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：

```
1  <?php
2  require(__DIR__ . '/BaseYii.php');
3  class Yii extends \yii\BaseYii
4  {
5  }
6
7  // 重点看这个 spl_autoload_register
8  spl_autoload_register(['Yii', 'autoload'], true, true);
9
10 // 下面的语句读取了一个映射表
11 Yii::$classMap = include(__DIR__ . '/classes.php');
12
13 Yii::$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：

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

```
4
5   ... ...
6
7   'yii\widgets\PjaxAsset' => YII2_PATH . '/widgets/PjaxAsset.php',
8   'yii\widgets\Spaceless' => YII2_PATH . '/widgets/Spaceless.php',
9 ];
```

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()

```
1  public static function autoload($className)
2  {
3      if (isset(static::$classMap[$className])) {
4          $classFile = static::$classMap[$className];
5          if ($classFile[0] === '@') {
6              $classFile = static::getAlias($classFile);
7          }
8      } elseif (strpos($className, '\\') !== false) {
9          $classFile = static::getAlias('@' . str_replace('\\', '/',
10             $className) . '.php', false);
11         if ($classFile === false || !is_file($classFile)) {
12             return;
13         }
14     } else {
15         return;
16     }
17
18     include($classFile);
19
20     if (YII_DEBUG && !class_exists($className, false) &&
21         !interface_exists($className, false) && !trait_exists($className,
22         false)) {
23         throw new UnknownClassException(
24         "Unable to find '$className' in file: $classFile. Namespace missing?");
25     }
26 }
```

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:

```
1  // 有效的类名，因为 @yii 是一个已经预定义好的别名
2  use yii\base\Application;
3
4  // 无效的类名，因为没有 @foo 或 @foo/bar 的根别名，要提前定义好
5  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。

3.3.2 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[2] 。

3.4 环境和配置文件

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，你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。

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

问题其实不在 PHP，不在 Yii。问题在于每个成员在开发的同时，都在自己本地进行编程，都在本地搭建了自己的开发和测试环境。因此，每个人用于所用的数据库名称、用户名、密码都是根据自己的喜好命名的。这些信息是作为配置项保存在 Yii 应用的配置文件中的。

这在代码的版本控制上，出现了麻烦。每次各成员向代码库提交代码时，由于他们都修改了配置文件中的用户名、密码等信息为本地信息。因此，在提交代码时，产生了冲突。虽然这个冲突不难解决，但是负责维护代码库的 Alice 觉得这很无聊，也很苦恼。同时，Bob 是个敏感的人，他认为自己本地的数据库连接信息包含了用户名、密码等信息，关系到自己的安全，不应该提交到团队的代码仓库中去。于是，每次提交前，他都要把这些信息从配置文件中删除，然后再提交。提交完了再改回来。不然代码连不上数据库呀。因此，Bob 也很苦恼。而 Charlie 是负责把经过测试的代码部署到产品服务器上。由于开发端的环境和产品端的完全不同，于是每次部署前，他不得不小心地对照原来产品端的配置文件，将开发端的配置，改成产品端的。

于是 Bob 提议，将配置文件排除在代码库之外，不对配置文件进行版本控制。这样 Alice 不用去解决无聊的代码冲突了，Bob 也不用每次都删除配置再提交了，Charlie 也不用每次拉取代码后第一件事就是把数据库连接信息改成服务端的了。

这确实解决了一些问题。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，或者修改一个配置项，那么你们将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，不能实现自动分发。

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，Yii2.0 的基础模板应用也没这个功能。这是 Yii2.0 高级模板引入的技术。在实际运用中，各开发团队，特别是大型团队，已经摸索出自己的一套环境配置和部署的操作办法了，Yii2.0 环境功能，也是需要与团队现有的环境部署规则相契合的，并非另起一炉。

3.4.1 环境的目录结构

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，包含了入口脚本 index.php 和各类配置文件。其实他们都放在 /path/to/digpage.com/environments 目录下面，我们看看这个目录都有哪些东西

```
.
|-- dev
|   |-- api
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       |-- index.php
|   |       `-- index-test.php
|   |-- backend
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       |-- index.php
```

```
|   |       `-- index-test.php
|   |-- common
|   |   `-- config
|   |       |-- main-local.php
|   |       `-- params-local.php
|   |-- console
|   |   `-- config
|   |       |-- main-local.php
|   |       `-- params-local.php
|   |-- frontend
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       |-- index.php
|   |       `-- index-test.php
|   `-- yii
|-- prod
|   |-- api
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       `-- index.php
|   |-- backend
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       `-- index.php
|   |-- common
|   |   `-- config
|   |       |-- main-local.php
|   |       `-- params-local.php
|   |-- console
|   |   `-- config
|   |       |-- main-local.php
|   |       `-- params-local.php
|   |-- frontend
|   |   |-- config
|   |   |   |-- main-local.php
|   |   |   `-- params-local.php
|   |   `-- web
|   |       `-- index.php
|   `-- yii
```

```
`-- index.php
```

从上面的目录结构图中，可以看到，环境目录下有 3 个东东：

- 目录 dev
- 目录 prod
- 文件 index.php

其中，dev 和 prod 结构相同，分别又包含了 4 个目录和 1 个文件：

- frontend 目录，用于前台的应用，包含了存放配置文件的 config 目录和存放 web 入口脚本的 web 目录
- backend 目录，用于后台应用，内容与 frontend 相同
- console 目录，用于命令行应用，仅包含了 config 目录，因为命令行应用不需要 web 入口脚本，因此没有 web 目录。
- common 目录，用于各 web 应用和命令行应用通用的环境配置，仅包含了 config 目录，因为不同应用不可能共用相同的入口脚本。注意这个 common 的层级低于环境的层级，也就是说，他的通用，仅是某一环境下通用，并非所有环境下通用。
- yii 文件，是命令行应用的入口脚本文件。

环境目录下的 index.php 并不是通常所说的 web 入口脚本，它其实是个定义文件。定义了可以使用的环境，打开这个文件看一眼：

```
return [
    'Development' => [
        'path' => 'dev',
        'setWritable' => [
            'backend/runtime',
            'backend/web/assets',
            'frontend/runtime',
            'frontend/web/assets',
        ],
        'setExecutable' => [
            'yii',
        ],
        'setCookieValidationKey' => [
            'backend/config/main-local.php',
            'frontend/config/main-local.php',
        ],
    ],
    'Production' => [
        'path' => 'prod',
        'setWritable' => [
            'backend/runtime',
            'backend/web/assets',
```

```
            'frontend/runtime',
            'frontend/web/assets',
        ],
        'setExecutable' => [
            'yii',
        ],
        'setCookieValidationKey' => [
            'backend/config/main-local.php',
            'frontend/config/main-local.php',
        ],
    ],
];
```

不用深入去研究，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。其中：

- path 指明了当前环境所对应的配置文件存放目录。如 Productions 环境对应的目录为 prod 。
- setWritable 指明了需要 init 脚本设定为可写模式的目录，这些目录 Yii 应用在运行时会写入。
- setExecutable 指明了要将哪个文件设为可执行。这个文件就是命令行应用的入口文件了。
- setCookieValidationsKey 指明了向哪个文件写入 cookieValidationKey 配置项。

对于分散于各处的 web 和 config 目录而言，它们也是有共性的。

- 凡是 web 目录，存放的都是 web 应用的入口脚本，一个 index.php 和一个测试版本的 index-test.php
- 凡是 config 目录，存放的，都是本地配置信息 main-local.php 和 params-local.php

3.4.2 环境配置的生效规则

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 yii 入口文件和 *-local.php 配置文件。复制到哪呢？复制到了 /path/to/digpage.com/ 目录下面，并覆盖 frontend backend console common 中对应的 config 目录和入口脚本（index.php 或 yii，common 中没有入口脚本）。

在初始情况下，即未运行过 init 脚本之前，各应用目录（frontend, beckend, console）和通用目录（common），都是已经有一些配置文件了。就是 config 目录下的 bootstrap.php main.php params.php 。

总的讲，*.php 与环境、配置相关的文件都有哪些呢？有表示主配置的 main.php main-local.php，有表示全局参数的 params.php params-local.php，表示引导阶段的 bootstrap.php，有表示入口脚本的 yii 和 index.php 。其中，bootstrap.php 在别名 (Alias) 中有介绍，且在优先顺序与其他配置文件的原则是一样的，下面就不再重复讲了。

运行了 init 脚本后，环境中的文件也被复制出来。这些配置文件成套地分布在各应用目录和通用目录的 config 目录下。而 index.php 入口脚本则分布在各应用目录的 web 目录下，yii 入口脚本则只放在应用根目录下。

入口脚本我们在入口文件 index.php 已经讲了，这里就不讲。剩下的，来看看配置文件们。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/digpage.com/environments 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。

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 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？这要看看入口文件 index.php 部分的内容，但总的原则是：

- 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。还记得么？他们是不同的应用，他们是独立的。但是，这里有个 common，通用于前台、后台等。common 的内容被前台或后台的覆盖。
- 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。
- params, main。这 2 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，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。因此，这两者不存在谁覆盖谁的问题。

3.4.3 环境的使用

环境在具体使用上，把握这么几个原则：

- 与前后台无关，且与环境无关的配置项，写到 digpage.com\common\config\main.php 中去。不要写到环境中去，也不要写到前台或后台的配置文件中去。比如，当使用 FileCache 作为缓存时，这是与环境无关、与前后台无关的，或者说所有环境下，前后台的配置都相同。有关配置项要写到 digpage.com\common\config\main.php 中去。
- 无关环境的配置，不要写到环境中去，写到应用的配置中去。如，应用的 ID，无论是开发环境还是产品环境，ID 是不会变的。但前后台 ID 是不同的。因此，ID 的配置项写到 digpage.com\frontend\config\main.php 中去。而不要写到 digpage.com\environment\frontend\config\main-local.php 中去。
- 与环境有关，但与前后台无关的配置项，要写到环境的 common 配置中去。比如，有的应用前后台使用的数据库是一致的，因此，其 db 配置项应该是一样的。但在开发时，所使用的数据库服务与产品时的数据库服务肯定是不一样的。这种情况下，要所配置项写到 digpage.com\environments\dev\common\main-local.php
- 环境配置文件只提供框架。凡是环境配置文件，对于敏感信息，如，数据库地址、用户名、密码、API Key 等信息，一律留空，供团队成员在调用 init 后自行填写。
- 本地配置绝不提交代码库。因此，所有的应用目录（frontend, backend 等，并非 environment 目录）下的，所有 *-local.php 都不提交代码库。这点 Yii 已经通过 .gitignore 为我们做好了。关于 .gitignore 的有关信息，可以看看 Git 文档[3] 的有关内容。

[3] http://git-scm.com/doc

这样做能达到什么效果呢?

- 整个代码仓库中，不会有任何的敏感信息。这个代码仓库即使被外部获取，其危害程度也仅限于代码。数据库、Web Service 等密码还不至于泄露。
- 增加、删除、更新配置项的所有修改，都可以通过版本控制系统向整个团队分发。
- 所有团队成员只需要记住权限范围内的敏感信息，就可以完成工作。本地开发的队友，只要有本地的数据库连接信息就可以开发。负责测试的队友，只需要知道测试数据库服务器连接信息就可进行测试。负责部署的队友，只需要知道产品数据库的连接信息就可以完成部署。所有团队成员只需要记住各自权限范围内有限的几个敏感信息就可以了。

不足之处就是每次 pull 代码时，如果配置文件有更新，需要团队成员调用 init 将新的配置文件覆盖本地配置。然后需要手动填入敏感信息。但这种情况在初期配置不太稳定的情况下，根据团队迭代频率，一般一天一次。而等开发进入正常阶段后，配置文件相对稳定，极少有需要修改的。

3.4.4 **注意** cookieValidationKey

另外还有一个需要读者朋友们留意的地方，就是每次调用 init 脚本切换、更新环境配置时，cookieValidationKey 都会被重新生成的随机串所覆盖。这往往会导致更新前后 cookieValidationKey 不一致。从安全角度来讲，定时不定时地更新 cookieValidationKey 也是无可厚非的。但同时我们也要看到由此产生的一个副作用，那就是原先保存在用户机器上的 cookie 在下次访问时，会全部失效。一个简单表现就是已经设置了自动登录的用户，在更新 cookieValidationKey 后，全部需要重新登录。

这在大部分情况下，我们认为这是可以接受的。相信大家在使用互联网的过程中，也有遇到过重新登录的情况。因此，通常我们也可以很放心的使用 init 脚本，而不必特别地去关注自动生成的 cookieValidationKey 。

这种情况下，cookieValidationKey 是一个纯粹的、与本地环境密切相关的配置项，我们不用太操心它。

但是，有的情况下，cookieValidationKey 则不宜由 init 脚本来自动生成，而需要运维人员人工进行干预。

需要人工干预的情况，一般出现在对 cookie 要求比较严格的场景。如，当你的应用采用分布式架构提供服务，同时运行在多个节点的时候。有的负载均衡策略会将同一用户的先后 2 次请求随机分配给不同的节点进行处理。而如果这两个节点的 cookieValidationKey 不一致，那么就会出现用户就会收到很奇怪的错误信息。

因此，在分布式情况下，最简便的处理方式还是通过人工干预，确保各节点的 cookieValidationKey 始终一致。

这种情况下，cookieValidationKey 已经不是一个纯粹的本地环境配置项了，最多算是一个环境级别的配置项，而与本地、本机没有多大关系了。这种情况下，应当将 cookieValidationKey 与其他诸如数据库密码等敏感信息等同视之，由具有权限的、负责运维的人员掌握。并在每次调用 init 脚本后，将所掌握的 cookieValidationKey 填写到相应的配置项中去。

这里舍弃掉了 init 脚本自动生成的 cookieValidationKey 。更新前后，cookieValidationKey 未发生改变，因此，对于应用的用户而言没什么影响。

幸运的是，绝大多数情况下，我们还无需对于 cookieValidationKey 操太多的心，可以完全由 init 脚本自动生成。

其实，对于 Yii 开发团队而言，刚开始的时候，是将 cookieValidationKey 放在非环境配置文件 main.php 中的。后来，觉得这个配置项与虽然不一定是本地相关的，但起码与环境有关，因此后来还是放在环境配置文件 main-local.php 中。

只是，由于安全上的考虑，对于未设置 cookieValidationKey 的情况会抛出异常。同时，又为了方便开发者，又采用了让 init 脚本生成随机串的方式来自动配置。

3.5 **配置项**（Configuration）

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:

```
return [
    'id' => 'app-frontend',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'controllerNamespace' => 'frontend\controllers',

    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=yii2advanced',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
        ... ...
        'cache' => [
            'class' => 'yii\caching\MemCache',
            'servers' => [
                [
                    'host' => 'cache1.digpage.com',
                    'port' => 11211,
                    'weight' => 60,
                ],
                [
                    'host' => 'cache2.digpage.com',
                    'port' => 11211,
                    'weight' => 40,
                ],
            ],
        ],
    ],
```

```
    'params' => [...],
];
```

Yii 中许多地方都要用到配置项，Yii 应用自身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。关于属性的有关内容，请查看属性（Property）。

3.5.1 配置项的格式

一个配置文件包含了 3 个部分：

- 基本信息配置。主要指如 id basePath 等这些应用的基本信息，主要是一些简单的字符串。
- components 配置。配置文件的主体，也是我们接下来要讲的配置项。
- params 配置。主要是提供一些全局参数。

我们一般讲的配置项是指 component 配置项及里面的子项。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 属性的初始值为 $propertyValue 。
- 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。

其中，class 元素仅在特定的情况下可以没有。就是使用配置数组的时候，其类型已经是确定的。这往往是用于重新配置一个已经存在的对象，或者是在创建对象时，使用了 new 或 Yii::createObject() 指定了类型。除此以外的大多数情况 class 都是配置数组的必备元素:

```
// 使用 new 时指定了类型，配置数组中就不应再有 class 元素
$connection = new \yii\db\Connection([
    'dsn' => $dsn,
    'username' => $username,
```

```
    'password' => $password,
]);

// 使用 Yii::createObject() 时，如果第一个参数指定了类型，也不应在配置数
// 组中设定 class
$db = Yii::createObject('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// 对现有的对象重新配置时，也不应在配置数组中设定 class
Yii::configure($db, [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

上面的例子中，在没看到配置数组的内容前，已经可以确定对象的类型了。这种其他情况下，配置数组中如果再有一个 class 元素来设定类型的话，就不合理了。这种情况下，配置数组不能有 class 元素。但除此以外的其他情况，均要求配置数组提供 class 元素，以表示要创建的对象的类型。

3.5.2 配置项产生作用的原理

从环境和配置文件 部分的内容，我们了解到了一个 Yii 应用，特别是高级模版应用，是具有许多个配置文件的，这些配置文件在入口脚本 index.php 中被引入，然后按照一定的规则合并成一个配置数组 $config 并用于创建 Application 对象。具体可以看看入口文件 index.php 部分的内容。在入口脚本中，调用了:

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

在 yii\web\Application 中，会调用父类的构造函数 yii\base\Application::__construct($config)，来创建 Web Application。在这个构造函数中:

```
public function __construct($config = [])
{
    Yii::$app = $this;
    $this->setInstance($this);

    $this->state = self::STATE_BEGIN;

    // 预处理配置项
    $this->preInit($config);

```

```
    $this->registerErrorHandler($config);

    // 使用 yii\base\Component::__construct() 完成构建
    Component::__construct($config);
}
```

可以看到，其实分成两步，一是对 $config 进行预处理，二是使用 yii\base\Component::__construct($config) 进行构建。

配置项预处理

预处理配置项的 yii\base\Application::preInit() 方法其实在别名 (Alias) 部分讲过，当时主要是从预定义别名的角度来讲的。现在我们再来完整地看看这个方法和有关的属性：

```
// basePath 属性，由 Application 的父类 yii\base\Module 定义，并提供 getter 和 setter
private $_basePath;

// runtimePath 属性和 vendorPath 属性，Application 都为其定义了 getter 和 setter。
private $_runtimePath;
private $_vendorPath;

// 还有一个 timeZone 属性，Application 为其提供了 getter 和 setter，但不提供存
// 储变量。
// 而是分别调用 PHP 的 date_default_timezone_get() 和
// date_default_timezone_set()

public function preInit(&$config)
{
    // 配置数组中必须指定应用 id，这里仅判断，不赋值。
    if (!isset($config['id'])) {
        throw new InvalidConfigException(
            'The "id" configuration for the Application is required.');
    }

    // 设置 basePath 属性，这个属性在 Application 的父类 yii\base\Module 中定义。
    // 在完成设置后，删除配置数组中的 basePath 配置项
    if (isset($config['basePath'])) {
        $this->setBasePath($config['basePath']);
        unset($config['basePath']);
    } else {
        throw new InvalidConfigException(
        'The "basePath" configuration for the Application is required.');
    }

    // 设置 vendorPath 属性，并在设置后，删除 $config 中的相应配置项
```

```
32      if (isset($config['vendorPath'])) {
33          $this->setVendorPath($config['vendorPath']);
34          unset($config['vendorPath']);
35      } else {
36          // set "@vendor"
37          $this->getVendorPath();
38      }
39
40      // 设置 runtimePath 属性，并在设置后，删除 $config 中的相应配置项
41      if (isset($config['runtimePath'])) {
42          $this->setRuntimePath($config['runtimePath']);
43          unset($config['runtimePath']);
44      } else {
45          // set "@runtime"
46          $this->getRuntimePath();
47      }
48
49      // 设置 timeZone 属性，并在设置后，删除 $config 中的相应配置项
50      if (isset($config['timeZone'])) {
51          $this->setTimeZone($config['timeZone']);
52          unset($config['timeZone']);
53      } elseif (!ini_get('date.timezone')) {
54          $this->setTimeZone('UTC');
55      }
56
57      // 将 coreComponents() 所定义的核心组件配置，与开发者通过配置文件定义
58      // 的组件配置进行合并。
59      // 合并中，开发者配置优先，核心组件配置起补充作用。
60      foreach ($this->coreComponents() as $id => $component) {
61
62          // 配置文件中没有的，使用核心组件的配置
63          if (!isset($config['components'][$id])) {
64              $config['components'][$id] = $component;
65
66          // 配置文件中有的，但并未指组件的 class 的，使用核心组件的 class
67          } elseif (is_array($config['components'][$id]) &&
68              !isset($config['components'][$id]['class'])) {
69              $config['components'][$id]['class'] = $component['class'];
70          }
71      }
72  }
```

从上面的代码可以看出，这个 preInit() 对配置数组 $config 作了以下处理：

- id 属性是必不可少的。

- 从 $config 中拿掉了 basePath runtimePath vendorPath 和 timeZone 4 个属性的配置项。当然，也设置了相应的属性。
- 对 $config['components'] 配置项进行两方面的补充。一是配置文件中没有的，而核心组件有的，把核心组件的配置信息补充进去。二是配置文件中虽然也有，但没有指定组件的 class 的，使用核心组件配置信息指定的 class。

基于此，我们不难得出如下结论：

- 有的配置项如 id 是不可少的，有的配置项如 basePath 等不用我们设置也是有默认值的。
- 对于核心组件，我们不配置也可以使用。
- 核心组件的 ID 是提前安排好的，没有充足的理由一般不要改变他，否则以后接手的人会骂你的。
- 核心组件可以不指明 class，默认会使用预先安排的类型。

对于核心组件，不同的应用有不同的安排，这个我们可以看看，大致了解下，具体在于各应用的 coreComponents() 中定义:

```
// yii\base\Application 的核心组件
public function coreComponents()
{
    return [
        'log' => ['class' => 'yii\log\Dispatcher'],             // 日志组件
        'view' => ['class' => 'yii\web\View'],                  // 视图组件
        'formatter' => ['class' => 'yii\i18n\Formatter'],       // 格式组件
        'i18n' => ['class' => 'yii\i18n\I18N'],                 // 国际化组件
        'mailer' => ['class' => 'yii\swiftmailer\Mailer'],      // 邮件组件
        'urlManager' => ['class' => 'yii\web\UrlManager'],      // url 管理组件
        'assetManager' => ['class' => 'yii\web\AssetManager'], // 前端资源管理组件
        'security' => ['class' => 'yii\base\Security'],         // 安全组件
    ];
}

// yii\web\Application 的核心组件，在基类的基础上加入 Web 应用必需的组件
public function coreComponents()
{
    return array_merge(parent::coreComponents(), [
        'request' => ['class' => 'yii\web\Request'],            // HTTP 请求组件
        'response' => ['class' => 'yii\web\Response'],          // HTTP 响应组件
        'session' => ['class' => 'yii\web\Session'],            // session 组件
        'user' => ['class' => 'yii\web\User'],                  // 用户管理组件
        'errorHandler' => ['class' => 'yii\web\ErrorHandler'], // 错误处理组件
    ]);
}

// yii\console\Application 的核心组件，
```

```php
public function coreComponents()
{
    return array_merge(parent::coreComponents(), [
        'request' => ['class' => 'yii\console\Request'],        // 命令行请求组件
        'response' => ['class' => 'yii\console\Response'],      // 命令行响应组件
        'errorHandler' => ['class' => 'yii\console\ErrorHandler'], // 错误处理组件
    ]);
}
```

这些我们大致有个印象就够了，不用刻意去记住，用着用着你就自然记住了。

使用配置数组构造应用

在使用 preInit() 完成配置数组的预处理之后，Application 构造函数又直接调用 yii\base\Component::__construct() 来构造 Application 对象。

结果这个 yii\base\Component::__construct() 也是个推委扯皮的家伙，他根本就没自己定义。而是直接继承了父类的 yii\base\Object::__construct() 。因此，Application 构造函数的最后一步，实际上调用的是 yii\base\Object::__construct($config) 。这个函数的原理，我们在Object 的配置方法 部分已经作出解释，这里就不再重复。

只是这里有两类特殊的配置项需要注意，就是以 on * 打头的事件和以 as * 打头的行为。对于事件行为，可以阅读事件（Event） 和行为（Behavior） 部分的内容。

Yii 对于这两类配置项的处理，是在 yii\base\Component::__set() 中完成的，从 Component 开始，才支持事件和行为。具体处理的代码如下:

```php
public function __set($name, $value)
{
    $setter = 'set' . $name;
    if (method_exists($this, $setter)) {
        $this->$setter($value);
        return;

    // 'on ' 打头的配置项在这里处理
    } elseif (strncmp($name, 'on ', 3) === 0) {

        // 对于 'on event' 配置项，将配置值作为事件 handler 绑定到 evnet 上去
        $this->on(trim(substr($name, 3)), $value);
        return;

    // 'as ' 打头的配置项在这里处理
    } elseif (strncmp($name, 'as ', 3) === 0) {

        // 对于 'as behavior' 配置项，将配置值作为创建 Behavior 的配置，创
        // 建后绑定为 behavior
```

```
20          $name = trim(substr($name, 3));
21          $this->attachBehavior($name, $value instanceof Behavior ? $value
22              : Yii::createObject($value));
23          return;
24      } else {
25          $this->ensureBehaviors();
26          foreach ($this->_behaviors as $behavior) {
27              if ($behavior->canSetProperty($name)) {
28                  $behavior->$name = $value;
29                  return;
30              }
31          }
32      }
33      if (method_exists($this, 'get' . $name)) {
34          throw new InvalidCallException('Setting read-only property: ' .
35              get_class($this) . '::' . $name);
36      } else {
37          throw new UnknownPropertyException('Setting unknown property: '
38              . get_class($this) . '::' . $name);
39      }
40  }
```

从上面的代码中可以看到，对于 on event 形式配置项，Yii 视配置值为一个事件 handler，绑定到 event 上。而对于 as behavior 形式的配置项，视配置值为一个 Behavior，注入到当前实例中，并冠以 behavior 的名称。

CHAPTER 4

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。包括依 MVC 模式、依赖注入（Denpdency Injection, DI）和服务定位器（Service Locator）等种模式。这里将结合 Web 应用和 Yii 具体实现进行探讨，以加深印象和理解。学习这些设计模式对于提高自身的设计水平很有帮助，这也是我们学习 Yii 的一个重要出发点。

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，是上世纪 80 年代提出的，到现在已经颇有历史了。MVC 的意义在于指导开发者将数据与表现解耦，提高代码，特别是模型部分代码的复用性。

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。越觉得自己无知，就越懂得敬畏和谦逊。从这个角度讲，同学们，最好不要看不起谦逊的人。

有个这么一个段子，说一天 A 君在圈内聚会时，朋友介绍了另一个人 B 君互相认识。聚会场合嘛，这很正常，也很普遍。于是 AB 君小聊了一下。按国人的习惯，A 君就问了“先生在哪高就?”。B 君只说了句，“谈不上高就，炒炒股。”“哦，原来是炒股的。”A 君心想，虽没觉得什么不对，但心理觉得 B 有点 low，只是没说破，也没表现出来。过后了一段时间，一次偶然机会，发现原来 B 君是国内某上市公司的二股东，身家过亿。人家没说慌，确实是炒股的……

话说远了，我们还说正题。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。但从技术的成熟程度和使用的广泛程度来讲，MVC 仍是主流。

Yii 是一个 Web 框架，从 Web 开发的分工来讲，Yii 的开发工作中，承担后端的内容多一些，毕竟主要就是 PHP 开发。前端主要是在 HTML、JavaScript、CSS 上进行开发，然后通过 Yii 把前端的内容管起来，如通过 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，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.js Angular.js 等前端框架。

4.1.1 MVC **的三要素**

MVC 是模型 (Model)、视图 (View)、控制器 (Controller)3 个单词的缩写。下面我们从这 3 个方面来讲解 MVC 中的三个要素。

- 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。开发初期的最重要任务，主要也是实现 Model 的部分。这一部分写得好，后面就可以改得少，开发起来就快。

- 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，也是收集用户输入的地方。如你在某个博客上看到的某一篇文章，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、可操作的。事实上也是如此，你不会让用户看到 Model，更不会让他直接操作 Model。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，也是客户经常要求改来改去的地方。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。这俩甩手不干了，那总得有人来干吧，只能 Controller 上了。Contorller 用于决定使用哪些 Model，对 Model 执行什么操作，为视图准备哪些数据，是 MVC 中沟通的桥梁。

对于 MVC 中三者的划分并没有十分明晰的定义和界线。MVC 设计模式只是一种指导思想，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，一般博客首页的文章列表，就是一种循环结构；

- 从不调用 Model 的写方法。也就是说，View 只从 Model 获取数据，而不直接改写 Model，所以我们说他们老死不相往来。

- 一般没有任何准备数据的代码，如查询数据库、组合成一定格式的字符串等。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，都是拿来就能直接用的变量。

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，因为关于应用的大量的业务逻辑也要在这里面表示。

- Model 所提供的数据都是原始数据。也就是说，不带有任何表现层的代码。比如，一般不会在输出的数据中添加 HTML 标签，这是 View 的工作。但是 Model 可以提供有结构的数据，数组结构、队列结构、乃至其他 Model 等。这个结构并非是表现层的格式，而是数据在内存中的表现。
- 与输出不同，Model 的输入数据，可以是带有表现格式的数据。如将一篇文章保存到 Post 里面，内容中必然包含各种 HTML 标签。因此，Model 一般要对输入数据作过滤、验证和规范化等预处理。特别是对于需要保存进数据库的，一定要对所有的输入数据作预处理。这些预处理，有的其实是业务逻辑。比如只有主编才可以删除文章，这一验证规则既也是业务逻辑，也是权限控制。而有些预处理，则不属于业务逻辑，比如，文章标题前后的空格应去除。
- 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，这些工作应该交由 Models 来完成。
- 调用 Models 的读方法，获取数据，直接传递给视图，供显示。当涉及到多个 Model 时，有关的逻辑应当交给 Model 来完成。
- 调用 Models 的类方法，对 Models 进行写操作。
- 调用视图渲染函数等，形成对用户 Reqeust 的 Response。

4.1.2 Model **设计参考**

在 MVC 中，Model 排第一，是有一定暗示的。一是 Model 是整个架构中，代码量最大，复用程度最高，也是最体现程序员设计功力的地方。二是 View 和 Controller 相对于 Model 而言，在实际开发中，复用程度不高，逻辑复杂程度较低。可以说，Model 设计得好，整个 MVC 就好，应用开发起就顺。

因此，这一节将以 Model 为核心，来讲 MVC 的设计。实话说，MVC 尽管提出了 Model View Controller 的划分思想，但到了具体实操中，并不是很好把握的。下面介绍的设计参考，也仅仅是个人在实际项目中的一些体会和想法，仅作参考。在具体设计中，可以后把握这么几点：

Model **应当集中整个应用的数据和业务逻辑**

应用当中涉及到的所有业务对象都应尽可能抽像成 Model。如，博客系统当中，文章要抽象成 Post，评论要抽象成 Comment。而相关的业务逻辑，如发布新文章可以用 Post::create() ，删除评论可以用 Comment::delete() 。这样子整个应用就显得很清晰明了。

基础 Model **应当尽可能细化**

在一个应用中，特别是对于大型复杂应用，Model 间关系可能比较复杂。在构造应用时，特别是基础 Model 时，要从足够小的粒度来设计。此时，就要考虑采取继承、封装等措施了。比如，一个博客文章 Post，一般包含了若干标签，在页上一般写在作者、日期等 Post 字段的旁边。从逻辑上来看，把标签作为 Post 的一个属性，是说得通的。但是如果把标签作为一个属性像标题、正文等字段一样依附于 Post。那么在有的功能上，实现起来是有难度的。比如，客户要求，当一个 Post 含有标签“yii, model”时，可以点击“yii”，然后系统列出所有具标签中含有“yii”的文章。

为了实现这个功能，正确的设计是单独将标签抽象成 Tag。这样，Post 和 Tag 是多对多的关系，即一个 Post 有多个 Tag，一个 Tag 也对应多个 Post。这个多对多关系可以通过一张数据表 tbl_post_tag 来表示。接下来，为 Post 增加 Post::getTags() 方法，并通过 tbl_post_tag 表来查询当前 Post 的所有标签。同时，为 Tag 增加 Tag::getPosts() 方法，也通过 tbl_post_tag 表来查询当前 Tag 对应的文章。这样，就具备了实现客户要求的新功能的基础。

因此，在 Model 设计上，要以尽量小的粒度进行设计。一般而言，粒度越小，复用的可能性就越高。

有的读者可能会问了，既然要求粒度尽可能地小，那么，Post 是不是也应当再细化，把段落抽象为 Model? 是否有这个必要，看客户需求。一般情况确实没有这必要，如果这么做，那是不是再以句子为单位进行抽象? 但如果客户要求这个博客系统的评论是针对段落进行的评论的，要将评论显示在对应的段落旁边，甚至显示每个段落评论人次等功能。那么就需要把段落抽象成 Model 了。

分层次设计 Model

从设计流程上，数据库结构设计与 Model 的设计是紧密相关的。先有数据库结构设计，后有 Model 设计。在设计数据库结构的时候，也是在设计 Model。一般而言，最单元、粒度最小的 Model 就是根据每个数据库表所生成的 Model，这往往是个 Active Record。

比如标签的问题，在数据库存储过程中，Post 和 Tag 是分开存的，而且这两个表的字段，没有冗余。tbl_post_tag 表也只记录他们的 ID，没有实质内容。

在获取数据渲染视图，向用户展现时，这两个 Model 及他们的字段，是完全够用，且没有冗余的。

那么，能不能说 Post 和 Tag 这两个 Model 是够用的呢? 显然还不够。

当用户在创建文章、修改文章、审核文章时，需要采用一个表单来显示来收集用户输入。其中，对于标签的采集，一般是一个长条的文本框，让用户一次性输入多个标签，并以 , 等进行分隔的。

但是，这个文本框没有一个字段与之进行对应。我们也没办法对这个字段的用户输入进行任何的验证、预处理。

因此，Post 的功能是不够用的。不够用怎么办? 那就加吧。但直接在 Post 里面加个 public $tagString 并不好。毕竟只是在使用表单时，才会有这个问题，其他场合，这个字段是没用的。

这种情况下，一般使用继承:

```
1 public class PostForm extends Post
2 {
```

```
    public $tagString;

    ... ...
}
```

这样，当控制器发现用户在创建、修改、审核文章时，可以使用 PostForm Model 来渲染视图了，而其他场合则仍使用 Post。这样就在需要时，增加了一个 tagString 的字段用于收集用户输入的标签。

在具体设计过程中，由于 Model 本身就会包含很多代码，因此，要多使用这继承等手段，把代码组织好。

仔细为 Model 方法命名

由于 Model 的代码量比较大，又集中了大量的逻辑，因此，会在一个 Model 中有大量的方法。仍然以 Post 为例，会涉及到创建、审核、发布、回收等流程，相关的方法比较多，在命名上要用心。可能会涉及到的、名字又比较接近的方法就有：

- getPrevPost()，前一篇文章，用于导航
- getNextPost()，下一篇文章，用于导航
- getRelatedPosts($n = 10)，获取相关的 N 篇文章，用于相关文章推荐列表
- getPostsOfAuthor($n = 10)，获取同一作者的 N 篇相关文章，用于作者文章列表
- getLatestPosts($n = 10)，最新的 N 篇文章，静态方法，用于文章列表或 RSS 输出
- getHotestPosts($n = 10)，最热门的 N 篇文章，静态方法，用于热门文章列表
- getPublishPosts($n = -1)，获取已经发布的 N 篇文章，静态方法，用于文章列表
- getDraftPosts($n = -1)，获取未发布的 N 篇文章，静态方法，用于作者页面

这里只是一些获取其他 Post 的方法，命名比较合理，一看就知道意思。而且全部写成 getter 的形式，可以使用读取属性的方式进行访问。

不单单是在 Model 方法的命名上要用心，在变量名、类名、方法名等的命名上，也要养成习惯，形成规律。不要图一时之快，胡乱起名。否则，出来混，迟早要还的。

4.1.3 MVC 与前后端的配合

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。对于 Web 应用而言，包含 Yii 在内的许多 Web 开发框架，都是没有办法知道用户的操作，如鼠标、键盘等操作的。Web 应用想要了解用户的操作，只能依靠用户发送 Request。而对于鼠标、键盘等的响应，只能依靠前端技术，如 JavaScript 等来实现。

再加上这几年来 Web 浏览器的功能日臻强大。因此，Web 应用开发出现了一个新的发展苗头，就是功能从后端往前端转移。

在前端，通过 JavaScript 捕获用户操作，进行相应处理。或是发送回后端获取响应后处理，如通过 ajax 请求后端数据，实现无刷新的局部页面更新，向用户进行反馈；或直接在前端由浏览器进行处理，如使用 backbone.js、Angular.js 等前端框架的数据绑定功能等。这些都使得 Web 应用表现得越来越像桌面应用。

后端 MVC 也在为前后端的发展而改变。Controller 的功能更多的变成了识别是 ajax 请求还是普通请求，并根据请求的不同采取相应的视图渲染方式。对于普通请求，正常渲染视图，输出 HTML。对于 ajax 请求，则返回局部渲染视图，输出 HTML 片段。有的甚至输出 XML 或者 JSON。唯一在大潮流中，巍然不动的，还是我们的大 Model。

4.2 依赖注入和依赖注入容器

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[1] 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，这也是我们学习 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 Injection, 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 设计了一个 yii\di\Container 来实现了 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，两者可以结合使用。目前，Yii2.0 把这 DI 和 Service Locator 这两个东西结合起来使用，或者说通过 DI 容器，实现了 Service Locator。

[1] http://martinfowler.com/articles/injection.html

是不是云里雾里的？没错，所谓“高大上”的玩意往往就是这样，看着很炫，很唬人。卖护肤品的难道会跟你说其实皮肤表层是角质层，不具吸收功能么？这玩意又不考试，大致意会下就 OK 了。万一哪天要在妹子面前要装一把范儿的时候，张口也能来这么几个“高大上”就行了。但具体的内涵，我们还是要要通过下面的学习来加深理解，毕竟要把“高大上”的东西用好，发挥出作用来。

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:

```
// 为邮件服务定义抽象层
interface EmailSenderInterface
{
    public function send(...);
}

// 定义 Gmail 邮件服务
class GmailSender implements EmailSenderInterface
{
    ...

    // 实现发送邮件的类方法
    public function send(...)
    {
        ...
    }
}

// 定义评论类
class Comment extend yii\db\ActiveRecord
{
    // 用于引用发送邮件的库
    private $_eMailSender;

    // 初始化时，实例化 $_eMailSender
    public function init()
    {
        ...
        // 这里假设使用 Gmail 的邮件服务
        $this->_eMailSender = GmailSender::getInstance();
        ...
    }

```

```
    // 当有新的评价，即 save() 方法被调用之后中，会触发以下方法
    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;
        ...
    }
```

```
14
15      // 当有新的评价，即 save() 方法被调用之后中，会触发以下方法
16      public function afterInsert()
17      {
18          ...
19          //
20          $this->_eMailSender->send(...);
21          ...
22      }
23  }
24
25  // 实例化两种不同的邮件服务，当然，他们都实现了 EmailSenderInterface
26  sender1 = new GmailSender();
27  sender2 = new MyEmailSender();
28
29  // 用构造函数将 GmailSender 注入
30  $comment1 = new Comment(sender1);
31  // 使用 Gmail 发送邮件
32  $comment1.save();
33
34  // 用构造函数将 MyEmailSender 注入
35  $comment2 = new Comment(sender2);
36  // 使用 MyEmailSender 发送邮件
37  $comment2.save();
```

上面的代码对比原来的代码，解决了 Comment 类对于 GmailSender 等具体类的依赖，通过构造函数，将相应的实现了 EmailSenderInterface 接口的类实例传入 Comment 类中，使得 Comment 类可以适用于不同的邮件服务。从此以后，无论要使用何何种邮件服务，只需写出新的 EmailSenderInterface 实现即可，Comment 类的代码不再需要作任何更改，多爽的一件事，扩展起来、测试起来都省心省力。

属性注入

与构造函数注入类似，属性注入通过 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:

```
1   // 这是属性注入的例子
2   class Comment extend yii\db\ActiveRecord
3   {
4       // 用于引用发送邮件的库
5       private $_eMailSender;
6
7       // 定义了一个 setter()
8       public function setEmailSender($value)
9       {
10          $this->_eMailSender = $value;
```

```
    }

    // 当有新的评价，即 save() 方法被调用之后中，会触发以下方法
    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。

与构造函数注入类似，属性注入也是将 Comment 类所依赖的 EmailSenderInterface 的实例化过程放在 Comment 类以外。这就是依赖注入的本质所在。为什么称为注入？从外面把东西打进去，就是注入。什么是外，什么是内？要解除依赖的类内部就是内，实例化所依赖单元的地方就是外。

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，前后关系也需要特别地注意，必须将被依赖的放在需要注入依赖的前面进行实例化。这实在是一件既没技术含量，又吃力不出成果的工作，这类工作是高智商（懒）人群的天敌，我们是不会去做这么无聊的事情的。

就像极其不想洗衣服的人发明了洗衣机（我臆想的，未考证）一样，为了解决这一无聊的问题，DI 容器被设计出来了。Yii 的 DI 容器是 yii\di\Container，这个容器继承了发明人的高智商，他知道如何对对象及

对象的所有依赖，和这些依赖的依赖，进行实例化和配置。

DI 容器中的内容

DI 容器中实例的表示

容器顾名思义是用来装东西的，DI 容器里面的东西是什么呢？Yii 使用 yii\di\Instance 来表示容器中的东西。当然 Yii 中还将这个类用于 Service Locator，这个在讲 Service Locator 时再具体谈谈。

yii\di\Instance 本质上是 DI 容器中对于某一个类实例的引用，它的代码看起来并不复杂:

```
class Instance
{
    // 仅有的属性，用于保存类名、接口名或者别名
    public $id;

    // 构造函数，仅将传入的 ID 赋值给 $id 属性
    protected function __construct($id)
    {
    }

    // 静态方法创建一个 Instance 实例
    public static function of($id)
    {
        return new static($id);
    }

    // 静态方法，用于将引用解析成实际的对象，并确保这个对象的类型
    public static function ensure($reference, $type = null, $container = null)
    {
    }

    // 获取这个实例所引用的实际对象，事实上它调用的是
    // yii\di\Container::get() 来获取实际对象
    public function get($container = null)
    {
    }
}
```

对于 yii\di\Instance ，我们要了解:

- 表示的是容器中的内容，代表的是对于实际对象的引用。
- DI 容器可以通过他获取所引用的实际对象。
- 类仅有的一个属性 id 一般表示的是实例的类型。

DI **容器的数据结构**

在 DI 容器中，维护了 5 个数组，这是 DI 容器功能实现的基础:

```
// 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 = [];

// 用于缓存依赖信息，以类名或接口名为键
private $_dependencies = [];
```

DI 容器的 5 个数组内容和作用如DI 容器 5 个数组示意图 所示。

注册依赖

使用 DI 容器，首先要告诉容器，类型及类型之间的依赖关系，声明一这关系的过程称为注册依赖。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,
```

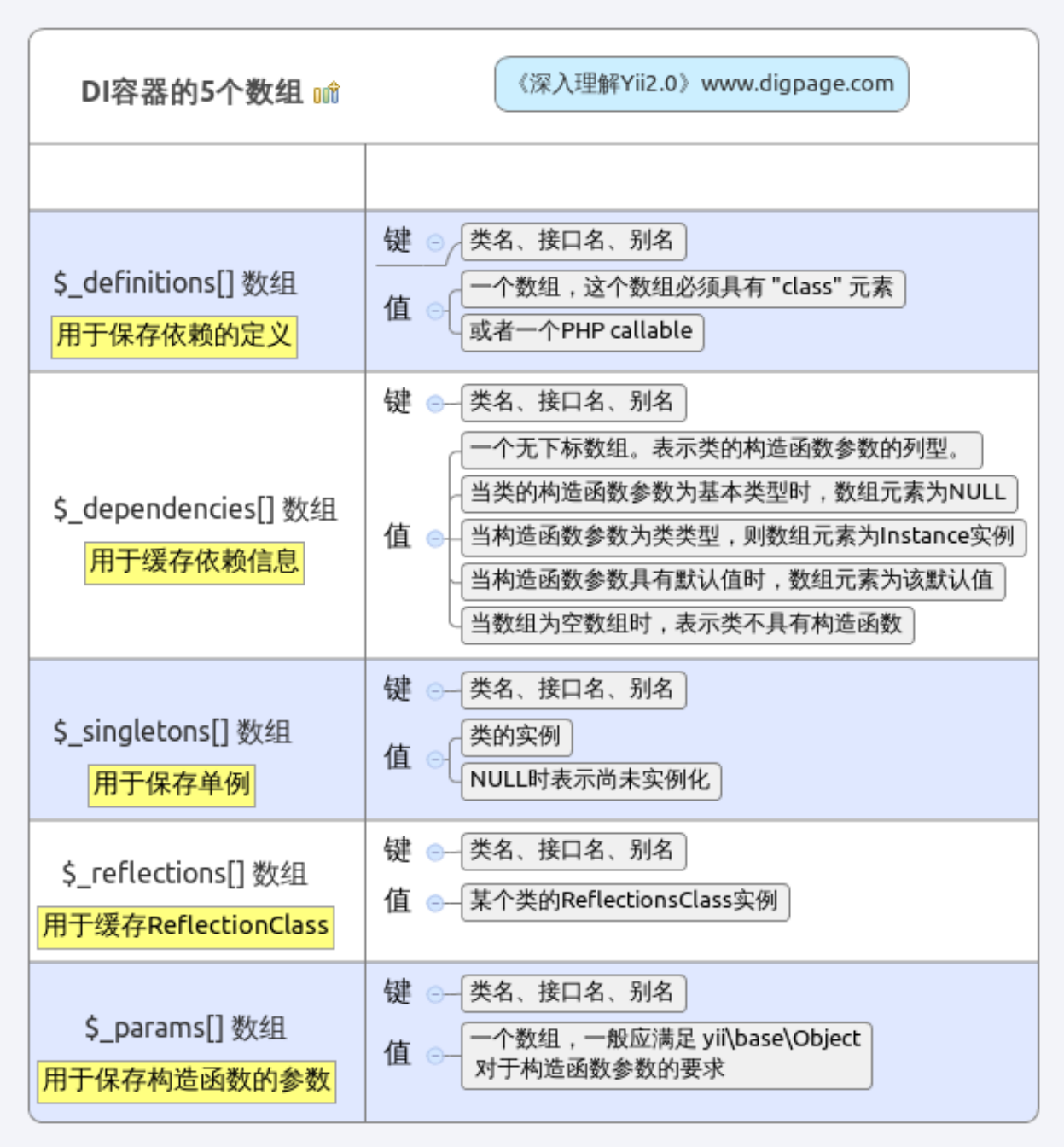

Fig. 4.1: DI 容器 5 个数组示意图

```
        $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。

表现在数据结构上，就是 set() 在注册依赖时，会把使用 setSingleton() 注册的依赖删除。否则，在解析依赖时，你让 Yii 究竟是依赖续弦还是原配？因此，在 DI 容器中，依赖关系的定义是唯一的。后定义的同名依赖，会覆盖前面定义好的依赖。

从形参来看，这两个函数的 $class 参数接受一个类名、接口名或一个别名，作为依赖的名称。$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 set() 和 setSingleton() 中，首先调用 yii\di\Container::normalizeDefinition() 对依赖的定义进行规范化处理，其代码如下：

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

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

    // $definition 是 PHP callable 或对象，则直接将其作为依赖的定义
    } elseif (is_callable($definition, true) || is_object($definition)) {
        return $definition;

    // $definition 是数组则确保该数组定义了 class 元素
    } elseif (is_array($definition)) {
        if (!isset($definition['class'])) {
            if (strpos($class, '\\') !== false) {
                $definition['class'] = $class;
            } else {
                throw new InvalidConfigException(
                    "A class definition requires a \"class\" member.");
            }
```

```
        }
        return $definition;
    // 这也不是，那也不是，那就抛出异常算了
    } else {
        throw new InvalidConfigException(
            "Unsupported definition type for \"$class\": "
            . gettype($definition));
    }
}
```

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，那么认为这个字符串就是所依赖的类名、接口名或别名，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。
- 上一步中，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。

总之，对于 $_definitions 数组中的元素，它要么是一个包含了"class" 元素的数组，要么是一个 PHP callable，再要么就是一个具体对象。这就是规范化后的最终结果。

在调用 normalizeDefinition() 对依赖的定义进行规范化处理后，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 ，表示定义了一个 Singleton，但是并未实现化。

这么讲可能不好理解，举几个具体的依赖定义及相应数组的内容变化为例，以加深理解：

```
$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。
// $_definition['yii\db\Connection'] = 'yii\db\Connetcion'
$container->set('yii\db\Connection');

// 注册一个接口，当一个类依赖于该接口时，定义中的类会自动被实例化，并供
// 有依赖需要的类使用。
// $_definition['yii\mail\MailInterface', 'yii\swiftmailer\Mailer']
$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
```

```php
// yii\db\Connection 实例。
// $_definition['foo', 'yii\db\Connection']
$container->set('foo', 'yii\db\Connection');

// 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。
// $_definition['yii\db\Connection'] = [...]
$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。
// $_definition['db'] = [...]
$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。
// $_definition['db'] = function(...){...}
$container->set('db', function ($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。
// $_definition['pageCache'] = anInstanceOfFileCache
$container->set('pageCache', new FileCache);
```

setSingleton() 对于 $_definition 和 $_params 数组产生的影响与 set() 是一样一样的。不同之处在于，使用 set() 会 unset $_singltons 中的对应元素，Yii 认为既然你都调用 set() 了，说明你希望这个依赖不再是单例了。而 setSingleton() 相比较于 set()，会额外地将 $_singletons[$class] 置为 null。以此来表示这个依赖已经定义了一个单例，但是尚未实例化。

从 set() 和 setSingleton() 来看，可能还不容易理解 DI 容器，比如我们说 DI 容器中维护了 5 个数组，但是依赖注册过程只涉及到其中 3 个。剩下的 $_reflections 和 $_dependencies 是在解析依赖的过程中完成构建的。

从 DI 容器的 5 个数组来看也好，从容器定义了 set() 和 setSingleton() 两个定义依赖的方法来看也好，不难猜出 DI 容器中装了两类实例，一种是单例，每次向容器索取单例类型的实例时，得到的都是同一个实

例；另一类是普通实例，每次向容器索要普通类型的实例时，容器会根据依赖信息创建一个新的实例给你。

单例类型主要用于节省构建实例的时间、节省保存实例的内存、共享数据等。而普通类型主要用于避免数据冲突。

对象的实例化

对象的实例化过程要比依赖的定义过程复杂得多。毕竟依赖的定义只是往特定的数据结构 $_singletons $_definitions 和 $_params 3 个数组写入有关的信息。稍复杂的东西也就是定义的规范化处理了。其它真没什么复杂的。像你这么聪明的，肯定觉得这太没挑战了。

而对象的实例化过程要相对复杂，这一过程会涉及到复杂依赖关系的解析、涉及依赖单元的实例化等过程。且让我们抽丝剥茧地进行分析。

解析依赖信息

容器在获取实例之前，必须解析依赖信息。这一过程会涉及到 DI 容器中尚未提到的另外 2 个数组 $_reflections 和 $_dependencies 。yii\di\Container::getDependencies() 会向这 2 个数组写入信息，而这个函数又会在创建实例时，由 yii\di\Container::build() 所调用。如它的名字所示意的，yii\di\Container::getDependencies() 方法用于获取依赖信息，让我们先来看看这个函数的代码

```
protected function getDependencies($class)
{
    // 如果已经缓存了其依赖信息，直接返回缓存中的依赖信息
    if (isset($this->_reflections[$class])) {
        return [$this->_reflections[$class], $this->_dependencies[$class]];
    }

    $dependencies = [];

    // 使用 PHP5 的反射机制来获取类的有关信息，主要就是为了获取依赖信息
    $reflection = new ReflectionClass($class);

    // 通过类的构建函数的参数来了解这个类依赖于哪些单元
    $constructor = $reflection->getConstructor();
    if ($constructor !== null) {
        foreach ($constructor->getParameters() as $param) {
            if ($param->isDefaultValueAvailable()) {

                // 构造函数如果有默认值，将默认值作为依赖。即然是默认值了，
                // 就肯定是简单类型了。
                $dependencies[] = $param->getDefaultValue();
            } else {
                $c = $param->getClass();

```

```
            // 构造函数没有默认值，则为其创建一个引用。
            // 就是前面提到的 Instance 类型。
            $dependencies[] = Instance::of($c === null ? null :
                $c->getName());
        }
    }
}

// 将 ReflectionClass 对象缓存起来
$this->_reflections[$class] = $reflection;

// 将依赖信息缓存起来
$this->_dependencies[$class] = $dependencies;

return [$reflection, $dependencies];
}
```

前面讲了 $_reflections 数组用于缓存 ReflectionClass 实例，$_dependencies 数组用于缓存依赖信息。这个 yii\di\Container::getDependencies() 方法实质上就是通过 PHP5 的反射机制，通过类的构造函数的参数分析他所依赖的单元。然后统统缓存起来备用。

为什么是通过构造函数来分析其依赖的单元呢？因为这个 DI 容器设计出来的目的就是为了实例化对象及该对象所依赖的一切单元。也就是说，DI 容器必然构造类的实例，必然调用构造函数，那么必然为构造函数准备并传入相应的依赖单元。这也是我们开头讲到的构造函数依赖注入的后续延伸应用。

可能有的读者会问，那不是还有 setter 注入么，为什么不用解析 setter 注入函数的依赖呢？这是因为要获取实例不一定需要为某属性注入外部依赖单元，但是却必须为其构造函数的参数准备依赖的外部单元。当然，有时候一个用于注入的属性必须在实例化时指定依赖单元。这个时候，必然在其构造函数中有一个用于接收外部依赖单元的形式参数。使用 DI 容器的目的是自动实例化，只是实例化而已，就意味着只需要调用构造函数。至于 setter 注入可以在实例化后操作嘛。

另一个与解析依赖信息相关的方法就是 yii\di\Container::resolveDependencies() 。它也是关乎 $_reflections 和 $_dependencies 数组的，它使用 yii\di\Container::getDependencies() 在这两个数组中写入的缓存信息，作进一步具体化的处理。从函数名来看，他的名字表明是用于解析依赖信息的。下面我们来看看它的代码:

```
protected function resolveDependencies($dependencies, $reflection = null)
{
    foreach ($dependencies as $index => $dependency) {

        // 前面 getDependencies() 函数往 $_dependencies[] 中
        // 写入的是一个 Instance 数组
        if ($dependency instanceof Instance) {
            if ($dependency->id !== null) {

                // 向容器索要所依赖的实例，递归调用 yii\di\Container::get()
```

```
                $dependencies[$index] = $this->get($dependency->id);
            } elseif ($reflection !== null) {
                $name = $reflection->getConstructor()
                    ->getParameters()[$index]->getName();
                $class = $reflection->getName();
                throw new InvalidConfigException(
                "Missing required parameter \"$name\" when instantiating \"$class\".");
            }
        }
    }
    return $dependencies;
}
```

上面的代码中可以看到，yii\di\Container::resolveDependencies() 作用在于处理依赖信息，将依赖信息中保存的 Istance 实例所引用的类或接口进行实例化。

综合上面提到的 yii\di\Container::getDependencies() 和 yii\di\Container::resolveDependencies() 两个方法，我们可以了解到：

- $_reflections 以类（接口、别名）名为键，缓存了这个类（接口、别名）的 ReflcetionClass。一经缓存，便不会再更改。
- $_dependencies 以类（接口、别名）名为键，缓存了这个类（接口、别名）的依赖信息。
- 这两个缓存数组都是在 yii\di\Container::getDependencies() 中完成。这个函数只是简单地向数组写入数据。
- 经过 yii\di\Container::resolveDependencies() 处理，DI 容器会将依赖信息转换成实例。这个实例化的过程中，是向容器索要实例。也就是说，有可能会引起递归。

实例的创建

解析完依赖信息，就万事俱备了，那么东风也该来了。实例的创建，秘密就在 yii\di\Container::build() 函数中：

```
protected function build($class, $params, $config)
{
    // 调用上面提到的 getDependencies 来获取并缓存依赖信息，留意这里 list 的用法
    list ($reflection, $dependencies) = $this->getDependencies($class);

    // 用传入的 $params 的内容补充、覆盖到依赖信息中
    foreach ($params as $index => $param) {
        $dependencies[$index] = $param;
    }

    // 这个语句是两个条件：
```

```
12      // 一是要创建的类是一个 yii\base\Object 类，
13      // 留意我们在《Yii 基础》一篇中讲到，这个类对于构造函数的参数是有一定要求的。
14      // 二是依赖信息不为空，也就是要么已经注册过依赖，
15      // 要么为 build() 传入构造函数参数。
16      if (!empty($dependencies) && is_a($class, 'yii\base\Object', true)) {
17          // 按照 Object 类的要求，构造函数的最后一个参数为 $config 数组
18          $dependencies[count($dependencies) - 1] = $config;
19
20          // 解析依赖信息，如果有依赖单元需要提前实例化，会在这一步完成
21          $dependencies = $this->resolveDependencies($dependencies, $reflection);
22
23          // 实例化这个对象
24          return $reflection->newInstanceArgs($dependencies);
25      } else {
26          // 会出现异常的情况有二：
27          // 一是依赖信息为空，也就是你前面又没注册过，
28          // 现在又不提供构造函数参数，你让 Yii 怎么实例化？
29          // 二是要构造的类，根本就不是 Object 类。
30          $dependencies = $this->resolveDependencies($dependencies, $reflection);
31          $object = $reflection->newInstanceArgs($dependencies);
32          foreach ($config as $name => $value) {
33              $object->$name = $value;
34          }
35          return $object;
36      }
37  }
```

从这个 yii\di\Container::build() 来看：

- DI 容器只支持 yii\base\Object 类。也就是说，你只能向 DI 容器索要 yii\base\Object 及其子类。再换句话说，如果你想你的类可以放在 DI 容器里，那么必须继承自 yii\base\Object 类。但 Yii 中几乎开发者在开发过程中需要用到的类，都是继承自这个类。一个例外就是上面提到的 yii\di\Instance 类。但这个类是供 Yii 框架自己使用的，开发者无需操作这个类。
- 递归获取依赖单元的依赖在于 dependencies = $this->resolveDependencies($dependencies, $reflection) 中。
- getDependencies() 和 resolveDependencies() 为 build() 所用。也就是说，只有在创建实例的过程中，DI 容器才会去解析依赖信息、缓存依赖信息。

容器内容实例化的大致过程

与注册依赖时使用 set() 和 setSingleton() 对应，获取依赖实例化对象使用 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，说明它不依赖其他单元，或者依赖信息不用定义，
    // 则根据传入的参数创建一个实例
    } elseif (!isset($this->_definitions[$class])) {
        return $this->build($class, $params, $config);
    }

    // 注意这里创建了 $_definitions[$class] 数组的副本
    $definition = $this->_definitions[$class];

    // 依赖的定义是个 PHP callable，调用之
    if (is_callable($definition, true)) {
        $params = $this->resolveDependencies($this->mergeParams($class,
            $params));
        $object = call_user_func($definition, $this, $params, $config);

    // 依赖的定义是个数组，合并相关的配置和参数，创建之
    } elseif (is_array($definition)) {
        $concrete = $definition['class'];
        unset($definition['class']);

        // 合并将依赖定义中配置数组和参数数组与传入的配置数组和参数数组合并
        $config = array_merge($definition, $config);
        $params = $this->mergeParams($class, $params);

        if ($concrete === $class) {
            // 这是递归终止的重要条件
            $object = $this->build($class, $params, $config);
        } else {
            // 这里实现了递归解析
            $object = $this->get($concrete, $params, $config);
        }

    // 依赖的定义是个对象则应当保存为单例
    } elseif (is_object($definition)) {
        return $this->_singletons[$class] = $definition;
    } else {
        throw new InvalidConfigException(
            "Unexpected object definition type: " . gettype($definition));
    }
```

```
46
47      // 依赖的定义已经定义为单例的，应当实例化该对象
48      if (array_key_exists($class, $this->_singletons)) {
49          $this->_singletons[$class] = $object;
50      }
51
52      return $object;
53  }
```

get() 用于返回一个对象或一个别名所代表的对象。可以是已经注册好依赖的，也可以是没有注册过依赖的。无论是哪种情况，Yii 均会自动解析将要获取的对象对外部的依赖。

get() 接受 3 个参数：

- $class 表示将要创建或者获取的对象。可以是一个类名、接口名、别名。
- $params 是一个用于这个要创建的对象的构造函数的参数，其参数顺序要与构造函数的定义一致。通常用于未定义的依赖。
- $config 是一个配置数组，用于配置获取的对象。通常用于未定义的依赖，或覆盖原来依赖中定义好的配置。

get() 解析依赖获取对象是一个自动递归的过程，也就是说，当要获取的对象依赖于其他对象时，Yii 会自动获取这些对象及其所依赖的下层对象的实例。同时，即使对于未定义的依赖，DI 容器通过 PHP 的 Reflection API，也可以自动解析出当前对象的依赖来。

get() 不直接实例化对象，也不直接解析依赖信息。而是通过 build() 来实例化对象和解析依赖。

get() 会根据依赖定义，递归调用自身去获取依赖单元。因此，在整个实例化过程中，一共有两个地方会产生递归：一是 get()，二是 build() 中的 resolveDependencies()。

DI 容器解析依赖实例化对象过程大体上是这么一个流程：

- 以传入的 $class 看看容器中是否已经有实例化好的单例，如有，直接返回这一单例。
- 如果这个 $class 根本就未定义依赖，则调用 build() 创建之。具体创建过程等下再说。
- 对于已经定义了这个依赖，如果定义为 PHP callable，则解析依赖关系，并调用这个 PHP callable。具体依赖关系解析过程等下再说。
- 如果依赖的定义是一个数组，首先取得定义中对于这个依赖的 class 的定义。然后将定义中定义好的参数数组和配置数组与传入的参数数组和配置数组进行合并，并判断是否达到终止递归的条件。从而选择继续递归解析依赖单元，或者直接创建依赖单元。

从 get() 的代码可以看出：

- 对于已经实例化的单例，使用 get() 时只能返回已经实例化好的实例，$params 参数和 $config 参数失去作用。这点要注意，Yii 不会提示你，所给出的参数不会发生作用的。有的时候发现明明已经给定配置数组了，怎么配置不起作用呀？就要考虑是不是因为这个原因了。

- 对于定义为数组的依赖，在合并配置数组和构造函数参数数组过程中，定义中定义好的两个数组会被传入的 $config 和 $params 的同名元素所覆盖，这就提供了获取不同实例的可能。
- 在定义依赖时，无论是使用 set() 还是使用 setSingleton() 只要依赖定义为特定对象或特定实例的，Yii 均将其视为单例。在获取时，也将返回这一单例。

实例分析

为了加深理解，我们以官方文档上的例子来说明 DI 容器解析依赖的过程。假设有以下代码:

```php
namespace app\models;

use yii\base\Object;
use yii\db\Connection;

// 定义接口
interface UserFinderInterface
{
    function findUser();
}

// 定义类，实现接口
class UserFinder extends Object implements UserFinderInterface
{
    public $db;

    // 从构造函数看，这个类依赖于 Connection
    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends Object
{
    public $finder;

    // 从构造函数看，这个类依赖于 UserFinderInterface 接口
    public function __construct(UserFinderInterface $finder, $config = [])
    {
```

```
        $this->finder = $finder;
        parent::__construct($config);
    }
}
```

从依赖关系看,这里的 UserLister 类依赖于接口 UserFinderInterface ,而接口有一个实现就是 UserFinder 类，但这类又依赖于 Connection 。

那么，按照一般常规的作法，要实例化一个 UserLister 通常这么做:

```
$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$lister = new UserLister($finder);
```

就是逆着依赖关系，从最底层的 Connection 开始实例化，接着是 UserFinder 最后是 UserLister 。在写代码的时候，这个前后顺序是不能乱的。而且，需要用到的单元，你要自己一个一个提前准备好。对于自己写的可能还比较清楚，对于其他团队成员写的，你还要看他的类究竟是依赖了哪些，并一一实例化。这种情况，如果是个别的、少量的还可以接受，如果有个 10 — 20 个的，那就麻烦了。估计光实例化的代码，就可以写满一屏幕了。

而且，如果是团队开发，有些单元应当是共用的，如邮件投递服务。不能说你写个模块，要用到邮件服务了，就自己实例化一个邮件服务吧? 那样岂不是有 N 模块就有 N 个邮件服务了? 最好的方式是使邮件服务成为一个单例，这样任何模块在需要邮件服务时，使用的其实是同一个实例。用传统的这种实例化对象的方法来实现的话，就没那么直接了。

那么改成 DI 容器的话，应该是怎么样呢? 他是这样的:

```
use yii\di\Container;

// 创建一个 DI 容器
$container = new Container;

// 为 Connection 指定一个数组作为依赖，当需要 Connection 的实例时，
// 使用这个数组进行创建
$container->set('yii\db\Connection', [
    'dsn' => '...',
]);

// 在需要使用接口 UserFinderInterface 时，采用 UserFinder 类实现
$container->set('app\models\UserFinderInterface', [
    'class' => 'app\models\UserFinder',
]);

// 为 UserLister 定义一个别名
$container->set('userLister', 'app\models\UserLister');

```

```
20 // 获取这个 UserList 的实例
21 $lister = $container->get('userLister');
```

采用 DI 容器的办法，首先各 set() 语句没有前后关系的要求，set() 只是写入特定的数据结构，并未涉及具体依赖关系的解析。所以，前后关系不重要，先定义什么依赖，后定义什么依赖没有关系。

其次，上面根本没有在 DI 容器中定义 UserFinder 对于 Connection 的依赖。但是 DI 容器通过对 UserFinder 构造函数的分析，能了解到这个类会对 Connection 依赖。这个过程是自动的。

最后，上面只有一个 get() 看起来好像根本没有实例化其他如 Connection 单元一样，但事实上，DI 容器已经安排好了一切。在获取 userLister 之前，Connection 和 UserFinder 都会被自动实例化。其中，Connection 是根据依赖定义中的配置数组进行实例化的。

经过上面的几个 set() 语句之后，DI 容器的 $_params 数组是空的，$_singletons 数组也是空的。$_definintions 数组却有了新的内容:

```
1 $_definitions = [
2     'yii\db\Connection' => [
3         'class' => 'yii\db\Connection',    // 注意这里
4         'dsn' => ...
5     ],
6     'app\models\UserFinderInterface' => ['class' => 'app\models\UserFinder'],
7     'userLister' => ['class' => 'app\models\UserLister']    // 注意这里
8 ];
```

在调用 get('userLister') 过程中又发生了什么呢？说实话，这个过程不是十分复杂，但是由于涉及到递归和回溯，写这里的时候，我写了改，改了写，示意图画了好几回，折腾了好久，都不满意，就怕说不清楚，读者朋友们理解起来费劲。最后画了一个简单的示意图，请你们对照DI 容器解析依赖获取实例的过程示意图，以及前面关于 get() build() getDependencies() resolveDependencies() 等函数的源代码，了解大致流程。如果有任何疑问、建议，也请在底部留言。

在DI 容器解析依赖获取实例的过程示意图 中绿色方框表示 DI 容器的 5 个数组，浅蓝色圆边方框表示调用的函数和方法。蓝色箭头表示读取内存，红色箭头表示写入内存，虚线箭头表示参照的内存对象，粗线绿色箭头表示回溯过程。图中 3 个圆柱体表示实例化过程中，创建出来的 3 个实例。

对于 get() 函数：

- 在第 1 步中调用 get('userLister') 表示要获得一个 userLister 实例。这个 userLister 不是一个有效的类名，说明这是一个别名。那么要获取的是这个别名所代表的类的实例。
- 查找 $_definitions 数组，发现 $_definitions['userLister'] = ['class'=>'app\models\UserLister'] 。这里 userLister 不等于 app\models\UserLister ，说明要获取的这个 userLister 实例依赖于 app\models\UserLister 。这是查找依赖定义数组的第一种情况。
- 而在第 22、23 步中，get(yii\db\Connection) 调用 get() 时指定要获取的实例的类型，与依赖定义数组 $_definitions 定义的所依赖的类型是相同的，都是 yii\db\Connection 。也就是说，自己依赖于自己，这就基本达到了停止递归调用 get() 的条件，差不多可以开始反溯了。这是查找依赖定义数组的第二种情况。

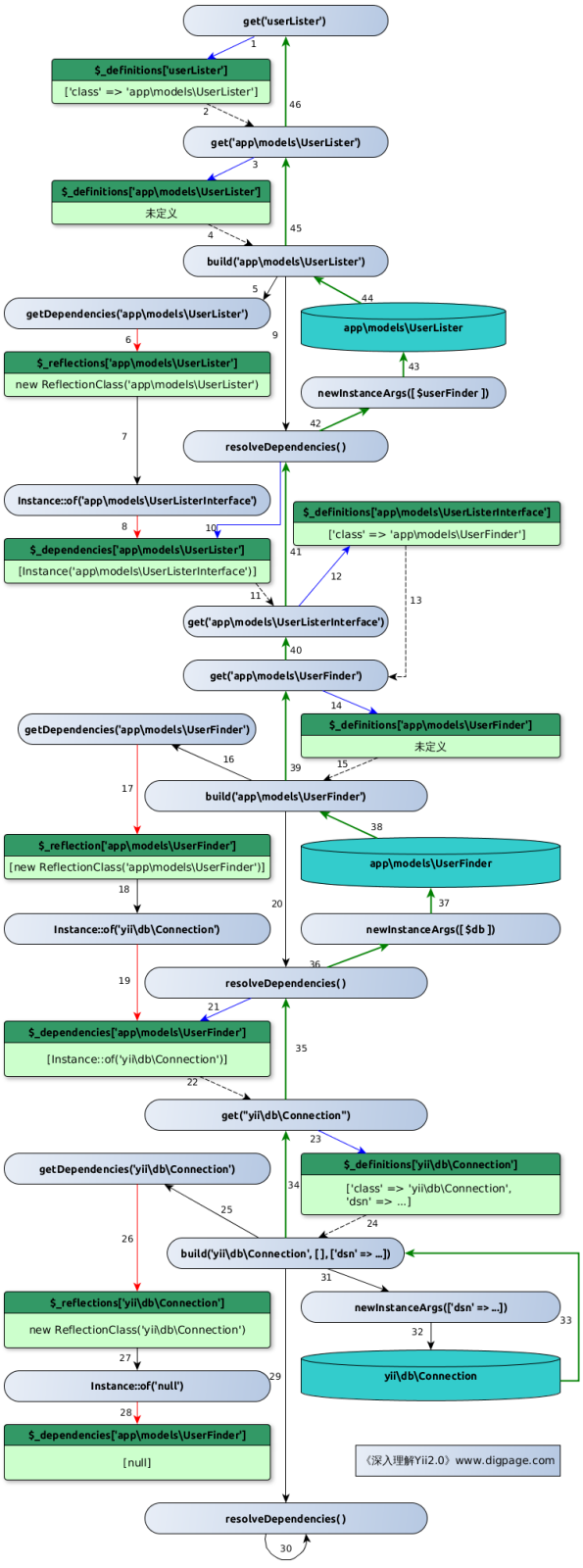

Fig. 4.2: DI 容器解析依赖获取实例的过程示意图

- 第三种情况是第 3、4 步、第 13、14 步查找依赖定义数组，发现依赖不存在。说明所要获取的类型的依赖关系未在容器中注册。对于未注册依赖关系的，DI 容器认为要么是一个没有外部依赖的简单类型，要么是一个容器自身可以自动解析其依赖关系的类型。
- 对于第一种情况，要获取的类型依赖于其他类型的，递归调用 get() 获取所依赖的类型。
- 对于第二、三种情况，直接调用 build() 尝试获取该类型的实例。

build() 在实例化过程中，干了这么几件事：

- 调用 getDependencies() 获取依赖信息。
- 调用 resolveDependencies() 解析依赖信息。
- 将定义中的配置数组、构造函数参数与调用 get() 时传入的配置数组和构造参数进行合并。这一步并未在上面的示意图中体现，请参阅 build() 的源代码部分。
- 根据解析回来的依赖单元，调用 newInstanceArgs() 创建实例。请留意第 36、42 步，并非直接由 resolveDependencies() 调用 newInstanceArgs() 。而是 resolveDependencies() 将依赖单元返回后，由 build() 来调用。就像第 31 步一样。
- 将获取的类型实例返回给调用它的 get() 。

getDependencies() 函数总是被 build() 调用，他干了这么几件事：

- 创建 ReflectionClass，并写入 $_reflections 缓存数组。如第 6 步中，$_reflections['app\models\UserLister'] = new ReflectionClass('app\models\UserLister') 。
- 利用 PHP 的 Reflection API，通过分析构造函数的形式参数，了解到当前类型对于其他单元、默认值的依赖。
- 将上一步了解到的依赖，在 $_dependencies 缓存数组中写入一个 Instance 实例。如第 7、8 步。
- 当一个类型的构造函数的参数列表中，没有默认值、参数都是简单类型时，得到一个 [null] 。如第 28 步。

resolveDependencies() 函数总是被 build() 调用，他在实例化时，干了这么几件事：

- 根据缓存在 $_dependencies 数组中的 Instance 实例的 id ，递归调用容器的 get() 实例化依赖单元。并返回给 build() 接着运行。
- 对于像第 28 步之类的依赖信息为 [null] 的，则什么都不干。

newInstanceArgs() 函数是 PHP Reflection API 的函数，用于创建实例，具体请看 PHP 手册[2] 。

这里只是简单的举例子而已，还没有涉及到多依赖和单例的情形，但是在原理上是一样的。希望继续深入了解的读者朋友可以再看看上面有关函数的源代码就行了，有疑问请随时留言。

从上面的例子中不难发现，DI 容器维护了两个缓存数组 $_reflections 和 $_dependencies 。这两个数组只写入一次，就可以无限次使用。因此，减少了对 ReflectionClass 的使用，提高了 DI 容器解析依赖和获取实例的效率。

[2] http://php.net/manual/zh/reflectionclass.newinstanceargs.php

另一方面，我们看到，获取一个实例，步骤其实不少。但是，对于典型的 Web 应用而言，有许多模块其实应当注册为单例的，比如上面的 yii\db\Connection 。一个 Web 应用一般使用一个数据库连接，特殊情况下会用多几个，所以这些数据库连接一般是给定不同别名加以区分后，分别以单例形式放在容器中的。因此，实际获取实例时，步骤会简单得。对于单例，在第一次 get() 时，直接就返回了。而且，省去不重复构造实例的过程。

这两个方面，都体现出 Yii 高效能的特点。

上面我们分析了 DI 容器，这只是其中的原理部分，具体的运用，我们将结合服务定位器（Service Locator） 来讲。

4.3 服务定位器（Service Locator）

跟 DI 容器类似，引入 Service Locator 目的也在于解耦。有许多成熟的设计模式也可用于解耦，但在 Web 应用上，Service Locator 绝对占有一席之地。对于 Web 开发而言，Service Locator 天然地适合使用，主要就是因为 Service Locator 模式非常贴合 Web 这种基于服务和组件的应用的运作特点。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。

4.3.1 Service Locator 的基本功能

在 Yii 中 Service Locator 由 yii\di\ServiceLocator 来实现。从代码组织上，Yii 将 Service Locator 放到与 DI 同一层次来对待，都组织在 yii\di 命名空间下。下面是 Service Locator 的源代码：

```
1  class ServiceLocator extends Component
2  {
3      // 用于缓存服务、组件等的实例
4      private $_components = [];
5
6      // 用于保存服务和组件的定义，通常为配置数组，可以用来创建具体的实例
7      private $_definitions = [];
8
9
10     // 重载了 getter 方法，使得访问服务和组件就跟访问类的属性一样。
11     // 同时，也保留了原来 Component 的 getter 所具有的功能。
12     // 请留意，ServiceLocator 并未重载 __set(),
13     // 仍然使用 yii\base\Component::__set()
14     public function __get($name)
15     {
```

```
16      ... ...
17  }
18
19  // 对比 Component，增加了对是否具有某个服务和组件的判断。
20  public function __isset($name)
21  {
22      ... ...
23  }
24
25  // 当 $checkInstance === false 时，用于判断是否已经定义了某个服务或组件
26  // 当 $checkInstance === true 时，用于判断是否已经有了某人服务或组件的实例
27  public function has($id, $checkInstance = false)
28  {
29      return $checkInstance ? isset($this->_components[$id]) :
30          isset($this->_definitions[$id]);
31  }
32
33  // 根据 $id 获取对应的服务或组件的实例
34  public function get($id, $throwException = true)
35  {
36     ... ...
37  }
38
39  // 用于注册一个组件或服务，其中 $id 用于标识服务或组件。
40  // $definition 可以是一个类名，一个配置数组，一个 PHP callable，或者一个对象
41  public function set($id, $definition)
42  {
43      ... ...
44  }
45
46  // 删除一个服务或组件
47  public function clear($id)
48  {
49      unset($this->_definitions[$id], $this->_components[$id]);
50  }
51
52  // 用于返回 Service Locator 的 $_components 数组或 $_definitions 数组，
53  // 同时也是 components 属性的 getter 函数
54  public function getComponents($returnDefinitions = true)
55  {
56      ... ...
57  }
58
59  // 批量方式注册服务或组件，同时也是 components 属性的 setter 函数
60  public function setComponents($components)
```

```
61      {
62          ... ...
63      }
64  }
```

从代码可以看出，Service Locator 继承自 yii\base\Component ，这是 Yii 中的一个基础类，提供了属性、事件、行为等基本功能，关于 Component 的有关知识，可以看看属性（Property） 、事件（Event）和行为（Behavior） 。

Service Locator 通过 __get() __isset() has() 等方法，扩展了 yii\base\Component 的最基本功能，提供了对于服务和组件的属性化支持。

从功能来看，Service Locator 提供了注册服务和组件的 set() setComponents() 等方法，用于删除的 clear() 。用于读取的 get() 和 getComponents() 等方法。

细心的读者可能一看到 setComponents() 和 getComponents() 就猜到了，Service Locator 还具有一个可读写的 components 属性。

Service Locator **的数据结构**

从上面的代码中，可以看到 Service Locator 维护了两个数组，$_components 和 $_definitions 。这两个数组均是以服务或组件的 ID 为键的数组。

其中，$_components 用于缓存存 Service Locator 中的组件或服务的实例。Service Locator 为其提供了 getter 和 setter。使其成为一个可读写的属性。$_definitions 用于保存这些组件或服务的定义。这个定义可以是：

- 配置数组。在向 Service Locator 索要服务或组件时，这个数组会被用于创建服务或组件的实例。与 DI 容器的要求类似，当定义是配置数组时，要求配置数组必须要有 class 元素，表示要创建的是什么类。不然你让 Yii 调用哪个构造函数？
- PHP callable。每当向 Service Locator 索要实例时，这个 PHP callable 都会被调用，其返回值，就是所要的对象。对于这个 PHP callable 有一定的形式要求，一是它要返回一个服务或组件的实例。二是它不接受任何的参数。至于具体原因，后面会讲到。
- 对象。这个更直接，每当你索要某个特定实例时，直接把这个对象给你就是了。
- 类名。即，使得 is_callable($definition, true) 为真的定义。

从 yii\di\ServiceLocator::set() 的代码:

```php
public function set($id, $definition)
{
    // 当定义为 null 时，表示要从 Service Locator 中删除一个服务或组件
    if ($definition === null) {
        unset($this->_components[$id], $this->_definitions[$id]);
        return;
    }
```

```
8
9       // 确保服务或组件 ID 的唯一性
10      unset($this->_components[$id]);
11
12      // 定义如果是个对象或 PHP callable，或类名，直接作为定义保存
13      // 留意这里 is_callable 的第二个参数为 true，所以，类名也可以。
14      if (is_object($definition) || is_callable($definition, true)) {
15          // 定义的过程，只是写入了 $_definitions 数组
16          $this->_definitions[$id] = $definition;
17
18      // 定义如果是个数组，要确保数组中具有 class 元素
19      } elseif (is_array($definition)) {
20          if (isset($definition['class'])) {
21              // 定义的过程，只是写入了 $_definitions 数组
22              $this->_definitions[$id] = $definition;
23          } else {
24              throw new InvalidConfigException(
25  "The configuration for the \"$id\" component must contain a \"class\" element.");
26          }
27
28      // 这也不是，那也不是，那么就抛出异常吧
29      } else {
30          throw new InvalidConfigException(
31              "Unexpected configuration type for the \"$id\" component: "
32              . gettype($definition));
33      }
34  }
```

服务或组件的 ID 在 Service Locator 中是唯一的，用于区别彼此。在任何情况下，Service Locator 中同一 ID 只有一个实例、一个定义。也就是说，Service Locator 中，所有的服务和组件，只保存一个单例。这也是正常的逻辑，既然称为服务定位器，你只要给定一个 ID，它必然返回一个确定的实例。这一点跟 DI 容器是一样的。

Service Locator 中 ID 仅起标识作用，可以是任意字符串，但通常用服务或组件名称来表示。如，以 db 来表示数据库连接，以 cache 来表示缓存组件等。

至于批量注册的 yii\di\ServiceLocator::setCompoents() 只不过是简单地遍历数组，循环调用 set() 而已。就算我不把代码贴出来，像你这么聪明的，一下子就可以自己写出来了。

向 Service Locator 注册服务或组件，其实就是向 $_definitions 数组写入信息而已。

访问 Service Locator 中的服务

Service Locator 重载了 __get() 使得可以像访问类的属性一样访问已经实例化好的服务和组件。下面是重载的 __get() 方法:

```php
public function __get($name)
{
    // has() 方法就是判断 $_definitions 数组中是否已经保存了服务或组件的定义
    // 请留意，这个时候服务或组件仅是完成定义，不一定已经实例化
    if ($this->has($name)) {

        // get() 方法用于返回服务或组件的实例
        return $this->get($name);

    // 未定义的服务或组件，那么视为正常的属性、行为，
    // 调用 yii\base\Component::__get()
    } else {
        return parent::__get($name);
    }
}
```

在注册好了服务或组件定义之后，就可以像访问属性一样访问这些服务（组件）。前提是已经完成注册，不要求已经实例化。访问这些服务或属性，被转换成了调用 yii\di\ServiceLocator::get() 来获取实例。下面是使用这种形式访问服务或组件的例子:

```php
// 创建一个 Service Locator
$serviceLocator = new yii\di\ServiceLocator;

// 注册一个 cache 服务
$serviceLocator->set('cache', [
    'class' => 'yii\cache\MemCache',
    'servers' => [
        ... ...
    ],
]);

// 使用访问属性的方法访问这个 cache 服务
$serviceLocator->cache->flushValues();

// 上面的方法等效于下面这个
$serviceLocator->get('cache')->flushValues();
```

在 Service Locator 中，并未重载 __set() 。所以，Service Locator 中的服务和组件看起来就好像只读属性一样。要向 Service Locator 中“写”入服务和组件，没有 setter 可以使用，需要调用 yii\di\ServiceLocator::set() 对服务和组件进行注册。

4.3.2 通过 Service Locator 获取实例

与注册服务和组件的简单之极相反，Service Locator 在创建获取服务或组件实例的过程要稍微复杂一点。这一点和 DI 容器也是很像的。Service Locator 通过 yii\di\ServiceLocator::get() 来创建、获取服务或组件的实例：

```
public function get($id, $throwException = true)
{
    // 如果已经有实例化好的组件或服务，直接使用缓存中的就 OK 了
    if (isset($this->_components[$id])) {
        return $this->_components[$id];
    }

    // 如果还没有实例化好，那么再看看是不是已经定义好
    if (isset($this->_definitions[$id])) {
        $definition = $this->_definitions[$id];

        // 如果定义是个对象，且不是 Closure 对象，那么直接将这个对象返回
        if (is_object($definition) && !$definition instanceof Closure) {
            // 实例化后，保存进 $_components 数组中，以后就可以直接引用了
            return $this->_components[$id] = $definition;

        // 是个数组或者 PHP callable，调用 Yii::createObject() 来创建一个实例
        } else {
            // 实例化后，保存进 $_components 数组中，以后就可以直接引用了
            return $this->_components[$id] = Yii::createObject($definition);
        }
    } elseif ($throwException) {
        throw new InvalidConfigException("Unknown component ID: $id");

    // 即没实例化，也没定义，万能的 Yii 也没办法通过一个任意的 ID，
    // 就给你找到想要的组件或服务呀，给你个 null 吧。
    // 表示 Service Locator 中没有这个 ID 的服务或组件。
    } else {
        return null;
    }
}
```

Service Locator 创建获取服务或组件实例的过程是：

- 看看缓存数组 $_components 中有没有已经创建好的实例。有的话，皆大欢喜，直接用缓存中的就可以了。
- 缓存中没有的话，那就要从定义开始创建了。
- 如果服务或组件的定义是个对象，那么直接把这个对象作为服务或组件的实例返回就可以了。但有一点要注意，当使用一个 PHP callable 定义一个服务或组件时，这个定义是一个 Closure 类的对象。这

种定义虽然也对象，但是可不能把这种对象直接当成服务或组件的实例返回。

- 如果定义是一个数组或者一个 PHP callable，那么把这个定义作为参数，调用 Yii::createObject() 来创建实例。

这个 Yii::createObject() 在讲配置时我们介绍过，当时只是点一点，这里会讲得更深一点。但别急，先放一放，知道他能为 Service Locator 创建对象就 OK 了。我们等下还会讲这个方法的。

4.3.3 在 Yii 应用中使用 Service Locator 和 DI 容器

我们在讲 DI 容器时，提到了 Yii 中是把 Service Locator 和 DI 容器结合起来用的，Service Locator 是建立在 DI 容器之上的。那么一个 Yii 应用，是如何使用 Service Locator 和 DI 容器的呢?

DI 容器的引入

我们知道，每个 Yii 应用都有一个入口脚本 index.php 。在其中，有一行不怎么显眼:

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

这一行看着普通，也就是引入一个 Yii.php 的文件。但是，让我们来看看这个 Yii.php

```
1  <?php
2
3  require(__DIR__ . '/BaseYii.php');
4
5  class Yii extends \yii\BaseYii
6  {
7  }
8
9  spl_autoload_register(['Yii', 'autoload'], true, true);
10 Yii::$classMap = include(__DIR__ . '/classes.php');
11
12 // 重点看这里。创建一个 DI 容器，并由 Yii::$container 引用
13 Yii::$container = new yii\di\Container;
```

Yii 是一个工具类，继承自 yii\BaseYii 。但这里对父类的代码没有任何重载，意味之父类和子类在功能上其实是相同的。但是，Yii 提供了让你修改默认功能的机会。就是自己写一个 Yii 类，来扩展、重载 Yii 默认的、由 yii\BaseYii 提供的特性和功能。尽管实际使用中，我们还从来没有需要改写过这个类，主要是因为没有必要在这里写代码，可以通过别的方式实现。但 Yii 确实提供了这么一个可能。这个在实践中不常用，有这么个印象就足够了。

这里重点看最后一句代码，创建了一个 DI 容器，并由 Yii::$container 引用。也就是说，Yii 类维护了一个 DI 容器，这是 DI 容器开始介入整个应用的标志。同时，这也意味着，在 Yii 应用中，我们可以随时使用 Yii::$container 来访问 DI 容器。一般情况下，如无必须的理由，不要自己创建 DI 容器，使用 Yii::$container 完全足够。

Application **的本质**

再看看入口脚本 index.php 的最后两行:

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

创建了一个 yii\web\Application 实例，并调用其 run() 方法。那么，这个 yii\web\Application 是何方神圣? 首先，yii\web\Application 继承自 yii\base\Application，这从 yii\web\Application 的代码可以看出来

```
class Application extends \yii\base\Application
{
    ... ...
}
```

而 yii\base\Application 又继承自 yii\base\Module，说明所有的 Application 都是 Module

```
abstract class Application extends Module
{
    ... ...
}
```

那么 yii\base\Module 又继承自哪个类呢? 不知道你猜到没，他继承自 yii\di\ServiceLocator

```
class Module extends ServiceLocator
{
    ... ...
}
```

所有的 Module 都是服务定位器 Service Locator，因此，所有的 Application 也都是 Service Locator。

同时，在 Application 的构造函数中，yii\base\Application::__construct()

```
1 public function __construct($config = [])
2 {
3     Yii::$app = $this;
4     ... ...
5 }
```

第一行代码就把 Application 当前的实例，赋值给 Yii::$app 了。这意味着 Yii 应用创建之后，可以随时通过 Yii::$app 来访问应用自身，也就是访问 Service Locator。

至此，DI 容器有了，Service Locator 也出现了。那么 Yii 是如何摆布这两者的呢? 这两者又是如何千里姻缘一线牵的呢?

实例创建方法

Service Locator 和 DI 容器的亲密关系就隐藏在 yii\di\ServiceLocator::get() 获取实例时，调用的 Yii::createObject() 中。前面我们说到这个 Yii 继承自 yii\BaseYii，因此这个函数实际上是 BaseYii::createObject()，其代码如下:

```
// static::$container 就是上面说的引用了 DI 容器的静态变量

public static function createObject($type, array $params = [])
{
    // 字符串，代表一个类名、接口名、别名。
    if (is_string($type)) {
        return static::$container->get($type, $params);

    // 是个数组，代表配置数组，必须含有 class 元素。
    } elseif (is_array($type) && isset($type['class'])) {
        $class = $type['class'];
        unset($type['class']);

        // 调用 DI 容器的 get() 来获取、创建实例
        return static::$container->get($class, $params, $type);

    // 是个 PHP callable 则调用其返回一个具体实例。
    } elseif (is_callable($type, true)) {

        // 是个 PHP callable，那就调用它，并将其返回值作为服务或组件的实例返回
        return call_user_func($type, $params);

    // 是个数组但没有 class 元素，抛出异常
    } elseif (is_array($type)) {
        throw new InvalidConfigException(
        'Object configuration must be an array containing a "class" element.');

    // 其他情况，抛出异常
    } else {
        throw new InvalidConfigException(
            "Unsupported configuration type: " . gettype($type));
    }
}
```

这个 createObject() 提供了一个向 DI 容器获取实例的接口，对于不同的定义，除了 PHP callable 外，createObject() 都是调用了 DI 容器的 yii\di\Container::get()，来获取实例的。Yii::createObject() 就是 Service Locator 和 DI 容器亲密关系的证明，也是 Service Locator 构建于 DI 容器之上的证明。而 Yii 中所有的 Module，包括 Application 都是 Service Locator，因此，它们也都构建在 DI 容器之上。

同时，在 Yii 框架代码中，只要创建实例，就是调用 Yii::createObject() 这个方法来实现。可以说，Yii

中所有的实例（除了 Application，DI 容器自身等入口脚本中实例化的），都是通过 DI 容器来获取的。

同时，我们不难发现，Yii 的基类 yii\BaseYii，所有的成员变量和方法都是静态的，其中的 DI 容器是个静态成员变量 $container。因此，DI 容器就形成了最常见形式的单例模式，在内存中仅有一份，所有的 Service Locator（Module 和 Application）都共用这个 DI 容器。就就节省了大量的内存空间和反复构造实例的时间。

更为重要的是，DI 容器的单例化，使得 Yii 不同的模块共用组件成为可能。可以想像，由于共用了 DI 容器，容器里面的内容也是共享的。因此，你可以在 A 模块中改变某个组件的状态，而 B 模块中可以了解到这一状态变化。但是，如果不采用单例模式，而是每个模块（Module 或 Application）维护一个自己的 DI 容器，要实现这一点难度会大得多。

所以，这种共享 DI 容器的设计，是必然的，合理的。

另外，前面我们讲到，当 Service Locator 中服务或组件的定义是一个 PHP callable 时，对其形式有一定要求。一是返回一个实例，二是不接收任何参数。这在 Yii::createObject() 中也可以看出来。

由于 Yii::createObject() 为 yii\di\ServiceLocator::get() 所调用，且没有提供第二参数，因此，当使用 Service Locator 获取实例时，Yii::createObject() 的 $params 参数为空。因此，使用 call_user_func($type, $params) 调用这个 PHP callable 时，这个 PHP callable 是接收不到任何参数的。

4.3.4 Yii **创建实例的全过程**

可能有的读者朋友会有疑问：不对呀，前面讲过 DI 容器的使用是要先注册依赖，后获取实例的。但 Service Locator 在注册服务、组件时，又没有向 DI 容器注册依赖。那在获取实例的时候，DI 容器怎么解析依赖并创建实例呢？

请留意，在向 DI 容器索要一个没有注册过依赖的类型时，DI 容器视为这个类型不依赖于任何类型可以直接创建，或者这个类型的依赖信息容器本身可以通过 Reflection API 自动解析出来，不用提前注册。

可能还有的读者会想：还是不对呀，在我开发 Yii 的过程中，又没有写过注册服务的代码：

```
Yii::$app->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=db.digpage.com;dbname=digpage.com',
    'username' => 'www.digpage.com',
    'password' => 'www.digapge.com',
    'charset' => 'utf8',
]);

Yii::$app->set('cache', [
    'class' => 'yii\caching\MemCache',
    'servers' => [
        [
            'host' => 'cache1.digpage.com',
            'port' => 11211,
            'weight' => 60,
```

```
        ],
        [
            'host' => 'cache2.digpage.com',
            'port' => 11211,
            'weight' => 40,
        ],
    ],
]);
```

为何可以在没有注册的情况下获取服务的实例并使用服务呢?

其实，你也不是什么都没写，至少肯定是在某个配置文件中写了有关的内容的:

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=yii2advanced',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
        'cache' => [
            'class' => 'yii\caching\MemCache',
            'servers' => [
                [
                    'host' => 'cache1.digpage.com',
                    'port' => 11211,
                    'weight' => 60,
                ],
                [
                    'host' => 'cache2.digpage.com',
                    'port' => 11211,
                    'weight' => 40,
                ],
            ],
        ],
        ... ...
    ],
];
```

只不过，在配置项（Configuration） 和Object 的配置方法 部分，我们了解了配置文件是如何产生作用的，配置到应用当中的。这个数组会被 Yii::configure($config) 所调用，然后会变成调用 Application 的 setComponents()，而 Application 其实就是一个 Service Locator。setComponents() 方法又会遍历传入的配置数组，然后使用使用 Service Locator 的 set() 方法注册服务。

到了这里，就可以了解到：每次在配置文件的 components 项写入配置信息，最终都是在向 Application 这个 Service Locator 注册服务。

让我们回顾一下，DI 容器、Service Locator 是如何配合使用的：

- Yii 类提供了一个静态的 $container 成员变量用于引用 DI 容器。在入口脚本中，会创建一个 DI 容器，并赋值给这个 $container 。
- Service Locator 通过 Yii::createObject() 来获取实例，而这个 Yii::createObject() 是调用了 DI 容器的 yii\di\Container::get() 来向 Yii::$container 索要实例的。因此，Service Locator 最终是通过 DI 容器来创建、获取实例的。
- 所有的 Module，包括 Application 都继承自 yii\di\ServiceLocator ，都是 Service Locator。因此，DI 容器和 Service Locator 就构成了整个 Yii 的基础。

CHAPTER 5

请求与响应 (TBD)

5.1 路由 (Route)

Web 开发中不可避免的要使用到 URL。用得最多的，就是生成一个指向应用中其他某个页面的 URL 了。开发者需要一个简洁的、集中的、统一的方法来完成这一过程。

否则的话，在代码中写入大量的诸如 http://www.digpage.com/post/view/100 的代码，一是过于冗长，二是易出错且难排查，三是日后修改起来容易有遗漏。因此，从开发角度来讲，需要一种更简洁、可以统一管理、又能排查错误的解决方案。

同时，我们在附录 2: Yii 的安装 部分讲解了如何为 Yii 配置 Web 服务器，从中可以发现，所有的用户请求都是发送给入口脚本 index.php 来处理的。那么，Yii 需要提供一种高效的分派请求的方法，来判断请求应当采用哪个 controller 哪个 action 进行处理。

结合以上 2 点需求，Yii 为开发者提供了路由和 URL 管理组件。

所谓路由是指 URL 中用于标识用于处理用户请求的 module, controller, action 的部分，一般情况下由 r 查询参数来指定。如 http://www.digpage.com/index.php?r=post/view&id=100 ，表示这个请求将由 PostController 的 actionView 来处理。

同时，Yii 也提供了一种美化 URL 的功能，使得上面的 URL 可以用一个比较整洁、美观的形式表现出来，如 http://www.digpage.com/post/view/100 。这个功能的实现是依赖于一个称为 urlManager 的应用组件。

使用 urlManager 开发者可以解析用户的请求，并指派相应的 module, controller 和 action 来进行处理，还可以根据预义的路由规则，生成需要的 URL 返回给用户使用。简而言之，urlManger 具有解析请求以便确定指派谁来处理请求和根据路由规则生成 URL 2 个功能。

5.1.1 美化 URL

一般情况下，Yii 应用生成和接受形如 http://www.digpage.com/index.php?r=post/view&id=100 的 URL。这个 URL 分成几个部分：

- 表示主机信息的 http://www.digapge.com
- 表示入口脚本的 index.php
- 表示路由的 r=post/view
- 表示普通查询参数的 id=100

其中，主机信息部分从 URL 来讲，一般是不能少的。当然内部链接可以使用相对路径，这种情况下看似可以省略，但是 User Agent 最终发出 Request 时，也是包含主机信息的。换句话说，Web Server 接收并转交给 Yii 处理的 URL，是完整的、带有主机信息的 URL。

而入口脚本 index.php 我们知道，Web Server 会将所有的请求都是交由其进行处理。也就是说，Web Server 应当视所有的 URL 为请求 index.php 脚本。这在 :ref:install 部分我们已经对 Web Server 进行过相应配置了。如 Nginx：

```
location / {
    try_files $uri $uri/ /index.php?$args;
}
```

即然这样，URL 中有没有指定 index.php 已经不重要了，反正都是请求的它。在 URL 里面假惺惺地留个 index.php，实在是画蛇添足。因此，Yii 允许我们不在 URL 中出现入口脚本 index.php 。

其次，路由信息对于 Yii 应用而言也必不可少，表明应当使用哪个 controller 和 action 来处理请求，否则 Yii 只能使用默认的路由来处理请求。这个形式比较固定，采用的是一种类似路径的形式，一般为 module/controller/action 之类的。

如果将 URL 省略掉入口脚本，并将路由信息转换成路径，上面的 URL 就会变成：http://www.digpage.com/post/view?id=100，是不是看起来舒服很多？

这样的链接看起来简洁美观，对于用户比较友好。同时，也比较适合搜索引擎的胃口，据说是 SEO 的手段之一。

但到了这里还没完，对于查询参数 id=100 而言，这个 URL 请求的是编号为 100 的一个 POST，并执行 view 操作。那么我们可以再进一步改成 http://www.digpage.com/post/view/100 。这样是不是更爽？

有处女座的说了，这个编号 100 跟前面的字母们放一起显得另类呀，要是都是字母的就更好了。那我们假如所请求的编号 100 的文章，其标题为 Route，那么不妨使用用 http://www.digpage.com/post/view/Route 来访问。

这样的话，干脆再加上 .html 好了。变成 http://www.digpage.com/post/view/Route.html，这样的 URL 对比原来，堪称完美了吧？岂不是连处女座也满意了？

我们把 URL http://www.digpage.comindex.php?r=post/view&id=100 变成 http://www.digpage.com/post/view/Route.html 的过程就称为 URL 美化。

Yii 有专门的 yii\web\UrlManager 来进行处理，其中：

- 隐藏入口脚本可以通过 yii\web\UrlManager::showScriptName = false 来实现
- 路由的路径化可以通过 yii\web\UrlManager::enablePrettyUrl = true 来实现

- 参数的路径化可以通过路由规则来实现
- 假后缀 (fake suffix) .html 可以通过 yii\web\UrlManager::suffix = '.html' 来实现

这里点一点，有个印象就可以下，在Url 管理 部分就会讲到了。

5.1.2 路由规则

所谓孤掌难鸣，urlManager 要发挥功能靠单打独斗是不行的，还要有另外一个的东东来配合。这就是我们本篇要重点讲的：路由规则。

路由规则是指 urlManager 用于解析请求或生成 URL 的规则。一个路由规则必须实现 yii\web\UrlRuleInterface 接口，这个接口定义了两个方法：

- 用于解析请求的 yii\web\UrlRuleInterface::parseRequest()
- 用于生成 URL 的 yii\web\UrlRuleInterface::createUrl()

Yii 中，使用 yii\web\UrlRule 来表示路由规则，一般这个类是足够开发者使用的。但是，如果开发者想自己实现解析请求或生成 URL 的逻辑，可以以这个类为基类进行派生，并重载 parseRuquest() 和 createUrl() 。

以下是配置文件中 urlManager 组件的路由规则配置部分，以几个相对简单、典型的路由规则的为例，先有个感性认识：

```
'rules' => [
    // 为路由指定了一个别名，以 post 的复数形式来表示 post/index 路由
    'posts' => 'post/index',

    // id 是命名参数，post/100 形式的 URL，其实是 post/view&id=100
    'post/<id:\d+>' => 'post/view',

    // controller action 和 id 以命名参数形式出现
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
        => '<controller>/<action>',

    // 包含了 HTTP 方法限定，仅限于 DELETE 方法
    'DELETE <controller:\w+>/<id:\d+>' => '<controller>/delete',

    // 需要将 Web Server 配置成可以接收 *.digpage.com 域名的请求
    'http://<user:\w+>.digpage.com/<lang:\w+>/profile' => 'user/profile',
]
```

上面的例子并没有穷尽路由规则的例子，可以玩的花样还有很多。至于这些例子所表达的规则，读者朋友们可以发挥想像去猜测，相信你们绝对可以猜个八九不离十。

目前不需要了解太多，只需大致了解上面这个数组用于为 urlManager 声明路由规则。数组的键相当于请求（需要解析的或将要生成的），而元素的值则对应的路由，即 controller/action 。请求部分可称为

pattern，路由部分则可称为 route。对于这 2 个部分的形式，大致上可以这么看：

- pattern 是从正则表达式变形而来。去除了两端的 / # 等分隔符。特别注意别在 pattern 两端画蛇添足加上分隔符。
- pattern 中可以使用正则表达式的命名参数，以供 route 部分引用。这个命名参数也是变形了的。对于原来 (?P<name>pattern) 的命名参数，要变形成 <name:pattern> 。
- pattern 中可以使用 HTTP 方法限定。
- route 不应再含有正则表达式，但是可以按 <name> 的形式引用命名参数。

也就是说，解析请求时，Yii 从左往右使用这个数组；而生成 URL 时 Yii 从右往左使用这个数组。

至于具体实现过程，我们马上就会讲。

首先是 yii\web\UrlRule 的代码，让我们来大致看一看：

```
class UrlRule extends Object implements UrlRuleInterface
{
    // 用于 $mode 表示路由规则的 2 种工作模式：仅用于解析请求和仅用于生成 URL。
    // 任意不为 1 或 2 的值均表示两种模式同时适用，
    // 一般未设定或为 0 时即表示两种模式均适用。
    const PARSING_ONLY = 1;
    const CREATION_ONLY = 2;

    // 路由规则名称
    public $name;

    // 用于解析请求或生成 URL 的模式，通常是正则表达式
    public $pattern;

    // 用于解析或创建 URL 时，处理主机信息的部分，如 http://www.digpage.com
    public $host;

    // 指向 controller 和 action 的路由
    public $route;

    // 以一组键值对数组指定若干 GET 参数，在当前规则用于解析请求时，
    // 这些 GET 参数会被注入到 $_GET 中去
    public $defaults = [];

    // 指定 URL 的后缀，通常是诸如 ".html" 等，
    // 使得一个 URL 看起来好像指向一个静态页面。
    // 如果这个值未设定，使用 UrlManager::suffix 的值。
    public $suffix;

    // 指定当前规则适用的 HTTP 方法，如 GET, POST, DELETE 等。
    // 可以使用数组表示同时适用于多个方法。
```

```
    // 如果未设定，表明当前规则适用于所有方法。
    // 当然，这个属性仅在解析请求时有效，在生成 URL 时是无效的。
    public $verb;

    // 表明当前规则的工作模式，取值可以是 0, PARSING_ONLY, CREATION_ONLY。
    // 未设定时等同于 0。
    public $mode;

    // 表明 URL 中的参数是否需要进行 url 编码，默认是进行。
    public $encodeParams = true;

    // 用于生成新 URL 的模板
    private $_template;

    // 一个用于匹配路由部分的正则表达式，用于生成 URL
    private $_routeRule;

    // 用于保存一组匹配参数的正则表达式，用于生成 URL
    private $_paramRules = [];

    // 保存一组路由中使用的参数
    private $_routeParams = [];

    // 初始化
    public function init() {...}

    // 用于解析请求，由 UrlRequestInterface 接口要求
    public function parseRequest($manager, $request) {...}

    // 用于生成 URL，由 UrlRequestInterface 接口要求
    public function createUrl($manager, $route, $params) {...}
}
```

从上面代码看，UrlRule 的属性（可配置项）比较多。各属性的意义在注释中已经写清楚了，这里就不再复述。但是我们要着重分析一下初始化函数 yii\web\UrlRule::init()，来加深对这些属性的理解:

```
public function init()
{
    // 一个路由规则必定要有 pattern，否则是没有意义的，
    // 一个什么都没规定的规定，要来何用？
    if ($this->pattern === null) {
        throw new InvalidConfigException('UrlRule::pattern must be set.');
    }

    // 不指定规则匹配后所要指派的路由，Yii 怎么知道将请求交给谁来处理？
```

```
10      // 不指定路由，Yii 怎么知道这个规则可以为谁创建 URL?
11      if ($this->route === null) {
12          throw new InvalidConfigException('UrlRule::route must be set.');
13      }
14
15      // 如果定义了一个或多个 verb，说明规则仅适用于特定的 HTTP 方法。
16      // 既然是 HTTP 方法，那就要全部大写。
17      // verb 的定义可以是字符串（单一的 verb）或数组（单一或多个 verb）。
18      if ($this->verb !== null) {
19          if (is_array($this->verb)) {
20              foreach ($this->verb as $i => $verb) {
21                  $this->verb[$i] = strtoupper($verb);
22              }
23          } else {
24              $this->verb = [strtoupper($this->verb)];
25          }
26      }
27
28      // 若未指定规则的名称，那么使用最能区别于其他规则的 $pattern
29      // 作为规则的名称
30      if ($this->name === null) {
31          $this->name = $this->pattern;
32      }
33
34      // 删除 pattern 两端的 "/"，特别是重复的 "/"，
35      // 在写 pattern 时，虽然有正则的成分，但不需要在两端加上 "/"，
36      // 更不能加上 "#" 等其他分隔符
37      $this->pattern = trim($this->pattern, '/');
38
39      // 如果定义了 host，将 host 部分加在 pattern 前面，作为新的 pattern
40      if ($this->host !== null) {
41          // 写入的 host 末尾如果已经包含有 "/" 则去掉，特别是重复的 "/"
42          $this->host = rtrim($this->host, '/');
43          $this->pattern = rtrim($this->host . '/' . $this->pattern, '/');
44
45      // 既未定义 host，pattern 又是空的，那么 pattern 匹配任意字符串。
46      // 而基于这个 pattern 的，用于生成的 URL 的 template 就是空的，
47      // 意味着使用该规则生成所有 URL 都是空的。
48      // 后续也无需再作其他初始化工作了。
49      } elseif ($this->pattern === '') {
50          $this->_template = '';
51          $this->pattern = '#^$#u';
52          return;
53
54      // pattern 不是空串，且包含有 '://'，以此认定该 pattern 包含主机信息
```

```
55      } elseif (($pos = strpos($this->pattern, '://')) !== false) {
56          // 除 '://' 外，第一个 '/' 之前的内容就是主机信息
57          if (($pos2 = strpos($this->pattern, '/', $pos + 3)) !== false) {
58              $this->host = substr($this->pattern, 0, $pos2);
59
60          // '://' 后再无其他 '/'，那么整个 pattern 其实就是主机信息
61          } else {
62              $this->host = $this->pattern;
63          }
64
65      // pattern 不是空串，且不包含主机信息，两端加上 '/'，形成一个正则
66      } else {
67          $this->pattern = '/' . $this->pattern . '/';
68      }
69
70      // route 也要去掉两头的 '/'
71      $this->route = trim($this->route, '/');
72
73      // 从这里往下，请结合流程图来看
74
75      // route 中含有 < 参数 >，则将所有参数提取成 [参数 => < 参数 >]
76      // 存入 _routeParams[],
77      // 如 ['controller' => '<controller>', 'action' => '<action>'],
78      // 留意这里的短路判断，先使用 strpos()，快速排除无需使用正则的情况
79      if (strpos($this->route, '<') !== false &&
80          preg_match_all('/<(\w+)>/', $this->route, $matches)) {
81          foreach ($matches[1] as $name) {
82              $this->_routeParams[$name] = "<$name>";
83          }
84      }
85
86      // 这个 $tr[] 和 $tr2[] 用于字符串的转换
87      $tr = [
88          '.' => '\\.',
89          '*' => '\\*',
90          '$' => '\\$',
91          '[' => '\\[',
92          ']' => '\\]',
93          '(' => '\\(',
94          ')' => '\\)',
95      ];
96      $tr2 = [];
97
98      // pattern 中含有 < 参数名: 参数 pattern> ,
99      // 其中 ': 参数 pattern' 部分是可选的。
```

```php
if (preg_match_all('/<(\w+):?([^>]+)?>/', $this->pattern, $matches,
    PREG_OFFSET_CAPTURE | PREG_SET_ORDER)) {
    foreach ($matches as $match) {
        // 获取 “参数名”
        $name = $match[1][0];

        // 获取 “参数 pattern” ，如果未指定，使用 '[^\/]' ，
        // 表示匹配除 '/' 外的所有字符
        $pattern = isset($match[2][0]) ? $match[2][0] : '[^\/]+';

        // 如果 defaults[] 中有同名参数，
        if (array_key_exists($name, $this->defaults)) {
            // $match[0][0] 是整个 < 参数名: 参数 pattern> 串
            $length = strlen($match[0][0]);
            $offset = $match[0][1];

            // pattern 中 < 参数名: 参数 pattern> 两头都有 '/'
            if ($offset > 1 && $this->pattern[$offset - 1] === '/'
                && $this->pattern[$offset + $length] === '/') {
                // 留意这个 (?P<name>pattern) 正则，这是一个命名分组。
                // 仅冠以一个命名供后续引用，使用上与直接的 (pattern) 没有区别
                // 见：http://php.net/manual/en/regexp.reference.subpatterns.php
                $tr["/<$name>"] = "(/(?P<$name>$pattern))?";
            } else {
                $tr["<$name>"] = "(?P<$name>$pattern)?";
            }

        // defaults[] 中没有同名参数
        } else {
            $tr["<$name>"] = "(?P<$name>$pattern)";
        }

        // routeParams[] 中有同名参数
        if (isset($this->_routeParams[$name])) {
            $tr2["<$name>"] = "(?P<$name>$pattern)";

        // routeParams[] 中没有同名参数，则将 参数 pattern 存入 _paramRules[] 中。
        // 留意这里是怎么对 参数 pattern 进行处理后再保存的。
        } else {
            $this->_paramRules[$name] = $pattern === '[^\/]+' ? '' :
                "#^$pattern$#u";
        }
    }
}

```

```
145     // 将 pattern 中所有的 < 参数名: 参数 pattern> 替换成 < 参数名 > 后作为 _template
146     $this->_template = preg_replace('/<(\w+):?([^>]+)?>/', '<$1>', $this->pattern);
147
148     // 将 _template 中的特殊字符及字符串使用 tr[] 进行转换，并作为最终的 pattern
149     $this->pattern = '#^' . trim(strtr($this->_template, $tr), '/') . '$#u';
150
151     // 如果指定了 routePrams 还要使用 tr2[] 对 route 进行转换，
152     // 并作为最终的 _routeRule
153     if (!empty($this->_routeParams)) {
154         $this->_routeRule = '#^' . strtr($this->route, $tr2) . '$#u';
155     }
156 }
```

上面的代码难点在于 pattern 等的转换过程，有点翻来覆去，转换过去、转换回来的感觉，这里我们先放一放，秋后再找他们来算帐，注意力先放在 init() 的前半部分，这些代码提醒我们：

- 规则的 $pattern 和 $route 是必须配置的。
- 规则的名称 $name 和主机信息 $host 在未配置的情况下，可以从 $pattern 来获取。
- $pattern 虽然含有正则的成分，但不需要在两端加入 / ，更不能使用 # 等其他分隔符。Yii 会自动为我们加上。
- 指定 $pattern 为空串，可以使该规则匹配任意的 URL。此时基于该规则所生成的所有 URL 也都是空串。
- $pattern 中含有 :\\ 时，Yii 会认为其中包含了主机信息。此时就不应当再指定 host 。否则，Yii 会将 host 接在这个 pattern 前，作为新的 pattern。这会造成该 pattern 两段 :\\ ，而这显然不是我们要的。

接下来要啃稍硬点的骨头了，就是 init() 的后半段，我们以一个普通的 ['post/<action:\w+>/<id:\d+>' => 'post/<action>'] 为例。同时，我们假设这个路由规则默认有 $defaults['id'] = 100 ，表示在未指定 post 的 id 时，使用 100 作为默认的 id。那么这个 UrlRule 的初始过程如UrlRule 路由规则初始化过程示意图 所示。

后续的初始化过程具体如下：

0. 从 ['post/<action:\w+>/<id:\d+>' => 'post/<action>'] ，我们有 $pattern = '/post/<action:\w+>/<id:\d+>/' 和 $route = 'post/<action>' 。
1. 首先从 $route 中提取出由 < > 所包含的部分。这里可以得到 ['action' => '<action>'] 。将其存入 $_routeParams[ ] 中。
2. 再从 $pattern 中提取出由 < > 所包含的部分，这里匹配 2 个部分，1 个 <action:\w+> 和 1 个 <id\d+> 。下面对这 2 个部分进行分别处理。
3. 对于 <action:\w+> 由于 $defaults[ ] 中不存在下标为 action 的元素，于是向 $tr[ ] 写入 (?P<$name>$pattern) 形式的元素，得到 $tr['<action>'] = '(?P<$name>\w+)' 。而对于 <id:\d+> ，由于 $defaults['id'] = 100 ，所以写入 $tr[ ] 的元素形式有所不同，变成 (/(?P<$name>$pattern))? 。于是有 $tr['<id>'] = (/(?P<$id>\d+))? 。

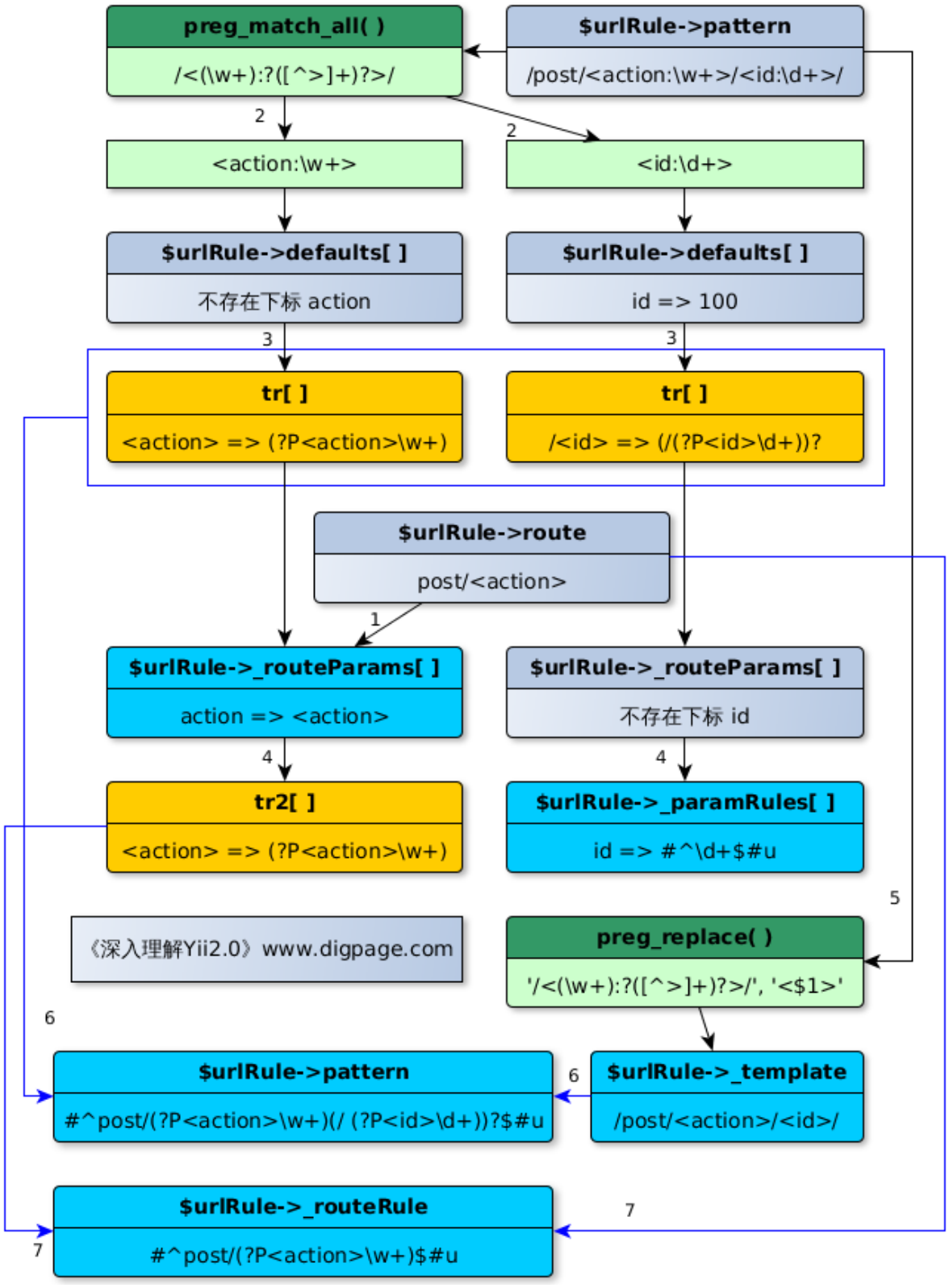

Fig. 5.1: UrlRule 路由规则初始化过程示意图

4. 由于在第 1 步只有 $_routeParams['action'] = '<actoin>'，而没有下标为 id 的元素。所以对于 <action:\w+>，往 tr2[ ] 中写入 ['<action>' => '(?P<action>\w+)']，而对于 <id:\d+> 则往 $_paramRules[ ] 中写入 ['id' => '#^\d+$#u']。

5. 上面只是准备工作，接下来开始各种替换。首先将 $pattern 中所有 <name:pattern> 替换成 <name> 并作为 $_template。因此，$_template = '/post/<action>/<id>/'。

6. 接下来用 $tr[ ] 对 $_template 进行替换，并在两端加上分隔符作为 $pattern。于是有 $pattern = '#^post/(?P<action>\w+)(/(?P<id>\d+))?$#u'。

7. 最后，由于第 1 步中 $_routeParams 不为空，所以需要使用 $tr2[ ] 对 $route 进行替换，并在两端加上分隔符后，作为 $_routeRule，于是有 $_routeRule = '#^post/(?P<action>\w+)$#'。

这些替换的意义在于方便开发者以简洁的方式在配置文件中书写路由规则，然后将这些简洁的规则，再替换成规范的正则表达式。让我们来看看这个 init() 的成果吧。仍然以上面的 ['post/<action:\w+>/<id:\d+>' => 'post/<action>'] 为例，经过 init() 处理后，我们最终得到了：

```
$urlRule->route = 'post/<action>';
$urlRule->pattern = '#^post/(?P<action>\w+)(/(?P<id>\d+))?$#u';
$urlRule->_template = '/post/<action>/<id>/';
$urlRule->_routeRule = '#^post/(?P<action>\w+)$#';
$urlRule->_routeParams = ['action' => '<action>'];
$urlRule->_paramRules = ['id' => '#^\d+$#u'];
// $tr 和 $tr2 作为局部变量已经完成历史使命光荣退伍了
```

下面我们来讲讲 UrlRule 是如何创建和解析 URL 的。

5.1.3 **创建** URL

URL 的创建就 UrlRule 层面来讲，是由 yii\web\UrlRule::createUrl() 负责的，这个方法可以根据传入的路由和参数创建一个相应的 URL 来。具体代码如下：

```
public function createUrl($manager, $route, $params)
{
    // 判断规则是否仅限于解析请求，而不适用于创建 URL
    if ($this->mode === self::PARSING_ONLY) {
        return false;
    }
    $tr = [];

    // 如果传入的路由与规则定义的路由不一致，
    // 如 post/view 与 post/<action> 并不一致
    if ($route !== $this->route) {

        // 使用 $_routeRule 对 $route 作匹配测试
        if ($this->_routeRule !== null && preg_match($this->_routeRule,
```

```
15          $route, $matches)) {
16
17          // 遍历所有的 _routeParams
18          foreach ($this->_routeParams as $name => $token) {
19              // 如果该路由规则提供了默认的路由参数,
20              // 且该参数值与传入的路由相同，则可以省略
21              if (isset($this->defaults[$name]) &&
22                  strcmp($this->defaults[$name], $matches[$name]) === 0) {
23                  $tr[$token] = '';
24              } else {
25                  $tr[$token] = $matches[$name];
26              }
27          }
28      // 传入的路由完全不能匹配该规则，返回
29      } else {
30          return false;
31      }
32  }
33
34  // 遍历所有的默认参数
35  foreach ($this->defaults as $name => $value) {
36      // 如果默认参数是路由参数，如 <action>
37      if (isset($this->_routeParams[$name])) {
38          continue;
39      }
40
41      // 默认参数并非路由参数，那么看看传入的 $params 里是否提供该参数的值。
42      // 如果未提供，说明这个规则不适用，直接返回。
43      if (!isset($params[$name])) {
44          return false;
45
46      // 如果 $params 提供了该参数，且参数值一致，则 $params 可省略该参数
47      } elseif (strcmp($params[$name], $value) === 0) {
48          unset($params[$name]);
49
50          // 且如果有该参数的转换规则，也可置为空。等下一转换就消除了。
51          if (isset($this->_paramRules[$name])) {
52              $tr["<$name>"] = '';
53          }
54
55      // 如果 $params 提供了该参数，但又与默认参数值不一致，
56      // 且规则也未定义该参数的正则，那么规则无法处理这个参数。
57      } elseif (!isset($this->_paramRules[$name])) {
58          return false;
59      }
```

```
60      }
61
62      // 遍历所有的参数匹配规则
63      foreach ($this->_paramRules as $name => $rule) {
64
65          // 如果 $params 传入了同名参数，且该参数不是数组，且该参数匹配规则，
66          // 则使用该参数匹配规则作为转换规则，并从 $params 中去掉该参数
67          if (isset($params[$name]) && !is_array($params[$name])
68              && ($rule === '' || preg_match($rule, $params[$name]))) {
69              $tr["<$name>"] = $this->encodeParams ?
70                  urlencode($params[$name]) : $params[$name];
71              unset($params[$name]);
72
73          // 否则一旦没有设置该参数的默认值或 $params 提供了该参数，
74          // 说明规则又不匹配了
75          } elseif (!isset($this->defaults[$name]) || isset($params[$name])) {
76              return false;
77          }
78      }
79
80      // 使用 $tr 对 $_template 时行转换，并去除多余的 '/'
81      $url = trim(strtr($this->_template, $tr), '/');
82
83      // 将 $url 中的多个 '/' 变成一个
84      if ($this->host !== null) {
85          // 再短的 host 也不会短于 8
86          $pos = strpos($url, '/', 8);
87          if ($pos !== false) {
88              $url = substr($url, 0, $pos) . preg_replace('#/+#', '/',
89                  substr($url, $pos));
90          }
91      } elseif (strpos($url, '//') !== false) {
92          $url = preg_replace('#/+#', '/', $url);
93      }
94
95      // 加上 .html 之类的假后缀
96      if ($url !== '') {
97          $url .= ($this->suffix === null ? $manager->suffix : $this->suffix);
98      }
99
100     // 加上查询参数们
101     if (!empty($params) && ($query = http_build_query($params)) !== '') {
102         $url .= '?' . $query;
103     }
104     return $url;
```

```
105 }
```

我们以上面提到 ['post/<action:\w+>/<id:\d+>' => 'post/<action>'] 路由规则来创建一个 URL，就假设要创建路由为 post/view ，id=100 的 URL 吧。具体的流程如UrlRule 创建 URL 的流程示意图 所示。

结合代码UrlRule 创建 URL 的流程示意图 ，URL 的创建过程大体上分 4 个阶段：

第一阶段 调用 createUrl(Yii::$app->urlManager, 'post/view', ['id'=>101]) 。

传入的路由为 post/view 与规则定义的路由 post/<action> 不同。但是，post/view 可以匹配路由规则的 $_routeRule = '#^post/(?P<action>\w+)$#' 。所以，认为该规则是适用的，可以接着处理。而如果连正则也匹配不上，那就说明该规则不适用，返回 false 。

遍历路由规则的所有 $_routeParams ，这个例子中，$_routeParams['action' => '<action>'] 。这个我们称为路由参数规则，即出现在路由部分的参数。

对于这个路由参数规则，我们并未为其设置默认值。但实际使用中，有的时候可能会提供默认的路由参数，比如对于形如 post/index 之类的，我们经常想省略掉 index ，那么就可以为 <action> 提供一个默认值 index 。

对于有默认值的情况，我们的头脑要清醒，目前是要用路由规则来创建 URL，规则所定义的默认值并非意味着可以处理不提供这个路由参数值的路由，而是说在处理路由参数值与默认值相等的路由时，最终生成的 URL 中可以省略该默认值。

即默认是体现在最终的 URL 上，是体现在 URL 解析过程中的默认，而不是体现在创建 URL 的过程中。也就是说，对于 post/index 类的路由，如果默认 index ，则生成的 URL 中可以不带有 index 。

这里没有默认值，相对简单。由于 $_routeRule 正则中，使用了命名分组，即 (?P<action>...) 。所以，可以很方便地使用 $matches['action'] 来捕获 \w+ 所匹配的部分，这里匹配的是 view 。故写入 $tr['<action>'] = view 。

第二阶段 接下来遍历所有的默认参数，当然不包含路由参数部分，因为这个在前面已经处理过了。这里只处理余下的参数。注意这个默认值的含义，如同我们前面提到的，这里是创建时必须提供，而生成出来的 URL 可以省略的意思。因此，对于 $params 中未提供相应参数的，或提供了参数值但与默认值不一致，且规则没定义参数的正则的，均说明规则不适用。只有在 $params 提供相应参数，且参数值与默认值一致或匹配规则时方可进行后续处理。

这里我们有一个默认参数 ['id' => 100] 。传入的 $params['id'] => 100 。两者一致，规则适用。

于将该参数从 $params 中删去。

接下来，看看是否为参数 id 定义了匹配规则。还真有 $_paramRules['id'] => '#^\d+$#u' 。但这也用不上了，因为这是在创建 URL，该参数与默认值一致，等下的 id 是要从 URL 中去除的。因此，写入 $tr['<id>'] = '' 。

第三阶段 再接下来，就是遍历所有参数匹配规则 $_paramRules 了。对于这个例子，只有 $_paramRules = ['id' => '#^\d+$#u'] 。

如果 $params 中并未定义该 id 参数，那么这一步什么也不用做，因为没有东西要写到 URL 中去。

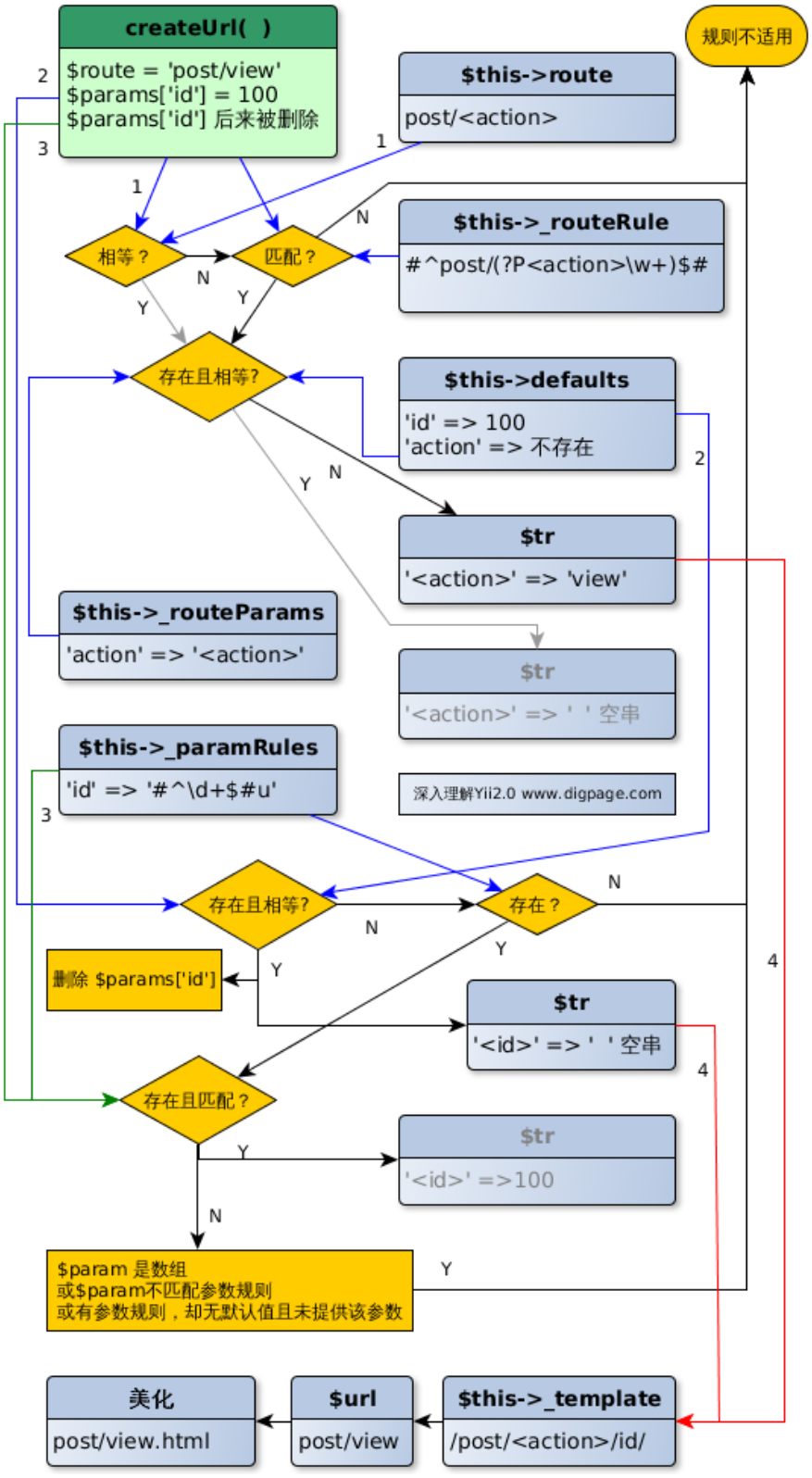

Fig. 5.2: UrlRule 创建 URL 的流程示意图

而一旦定义了 id，那么就需要看看当前路由规则是否适用了。判断的标准是所提供的参数不是数组，且匹配 $_paramRules 所定义的规则。而如果 $parasm['id'] 是数组，或不与规则匹配，或定义了 id 的参数规则却没有定义其默认值而 $params['id'] 又未提供，则规则不适用。

这里，我们在是在前面处理默认参数时，已经将 id 从 $params 中删去。但判断到规则为 id 定义了默认值的，所以认为规则仍然适用。只是，这里实际上不用做任何处理。如果需要处理的情况，也是将该参数从 $params 中删去，然后写入 $tr['<id>'] = 100 。

第四阶段 上面一切准备就绪之后，就可以着手生成 URL 了。主要用 $tr 对路由规则的 $_template 进行转换。这里，$_template = '/post/<action>/<id>/'，因此，转换后再去除多余的 / 就变成了 $url = 'post/view' 。其中 id=100 被省略掉了。

最后再分别接上 .html 的后缀和查询参数，一个 URL post/view.html 就生成了。其中，查询参数串是 $params 中剩下的内容，使用 PHP 的 http_build_query( ) 生成的。

从创建 URL 的过程来看，重点是完成这么几项工作：

- 看规则是否适用，主要标准是路由与规则定义的是否匹配，规则通过默认值或正则所定义的参数，是否都提供了。
- 看看当前要创建的 URL 是否与规则定义的默认的路由参数和查询参数一致，对于一致的，可以省略。
- 看将这些与默认值一致的，规则已经定义了的参数从 $params 删除，余下的，转换成最终 URL 的查询参数串。

5.1.4 **解析** URL

说完了路由规则生成 URL 的过程，再来看看其逻辑上的逆过程，即 URL 的解析。

先从路由规则 yii\web\UrlRule::parseRequest() 的代码入手：

```
1  public function parseRequest($manager, $request)
2  {
3      // 当前路由规则仅限于创建 URL，直接返回 false。
4      // 该方法返回 false 表示当前规则不适用于当前的 URL。
5      if ($this->mode === self::CREATION_ONLY) {
6          return false;
7      }
8
9      // 如果规则定义了适用的 HTTP 方法，则要看当前请求采用的方法是否可以接受
10     if (!empty($this->verb) && !in_array($request->getMethod(),
11         $this->verb, true)) {
12         return false;
13     }
14
15     // 获取 URL 中入口脚本之后、查询参数 ? 号之前的全部内容，即为 PATH_INFO
16     $pathInfo = $request->getPathInfo();
17     // 取得配置的 .html 等假后缀，留意 (string)null 转成空串
```

```
18      $suffix = (string) ($this->suffix === null ? $manager->suffix :
19          $this->suffix);
20
21      // 有假后缀且有 PATH_INFO
22      if ($suffix !== '' && $pathInfo !== '') {
23          $n = strlen($suffix);
24
25          // 当前请求的 PATH_INFO 以该假后缀结尾，留意 -$n 的用法
26          if (substr_compare($pathInfo, $suffix, -$n, $n) === 0) {
27              $pathInfo = substr($pathInfo, 0, -$n);
28
29              // 整个 PATH_INFO 仅包含一个假后缀，这是无效的。
30              if ($pathInfo === '') {
31                  return false;
32              }
33
34          // 应用配置了假后缀，但是当前 URL 却不包含该后缀，返回 false
35          } else {
36              return false;
37          }
38      }
39
40      // 规则定义了主机信息，即 http://www.digpage.com 之类，那要把主机信息接回去。
41      if ($this->host !== null) {
42          $pathInfo = strtolower($request->getHostInfo()) .
43              ($pathInfo === '' ? '' : '/' . $pathInfo);
44      }
45
46      // 当前 URL 是否匹配规则，留意这个 pattern 是经过 init() 转换的
47      if (!preg_match($this->pattern, $pathInfo, $matches)) {
48          return false;
49      }
50
51      // 遍历规则定义的默认参数，如果当前 URL 中没有，则加入到 $matches 中待统一处理，
52      // 默认值在这里发挥作用了，虽然没有，但仍视为捕获到了。
53      foreach ($this->defaults as $name => $value) {
54          if (!isset($matches[$name]) || $matches[$name] === '') {
55              $matches[$name] = $value;
56          }
57      }
58      $params = $this->defaults;
59      $tr = [];
60
61      // 遍历所有匹配项，注意这个 $name 的由来是 (?P<name>...) 的功劳
62      foreach ($matches as $name => $value) {
```

```
63          // 如果是匹配一个路由参数
64          if (isset($this->_routeParams[$name])) {
65              $tr[$this->_routeParams[$name]] = $value;
66              unset($params[$name]);
67
68          // 如果是匹配一个查询参数
69          } elseif (isset($this->_paramRules[$name])) {
70              // 这里可能会覆盖掉 $defaults 定义的默认值
71              $params[$name] = $value;
72          }
73      }
74
75      // 使用 $tr 进行转换
76      if ($this->_routeRule !== null) {
77          $route = strtr($this->route, $tr);
78      } else {
79          $route = $this->route;
80      }
81      Yii::trace("Request parsed with URL rule: {$this->name}", __METHOD__);
82      return [$route, $params];
83  }
```

我们以 http://www.digpage.com/post/view.html 为例，来看看上面的代码是如何解析成路由 ['post/view', ['id'=>100]] 的。注意，这里我们仍然假设路由规则提供了 id=100 默认值。而如果路由规则未提供该默认值，则请求形式要变成 http://www.digapge.com/post/view/100.html 。同时，规则的 $pattern 也会不同:

```
1  // 未提供默认值，id 必须提供，否则匹配不上
2  $pattern = '#^post/(?P<action>\w+)/(?P<id>\d+)?$#u';
3
4  // 提供了默认值，id 可以不提供，照样可以匹配上
5  $pattern = '#^post/(?P<action>\w+)(/(?P<id>\d+))?$#u';
```

这个不同的原因在于 UrlRule::init() ，读者朋友们可以回头看看。

在讲 URL 的解析前，让我们先从请求的第一个经手人 Web Server 说起，在附录 2: Yii 的安装 中讲到 Web Server 的配置时，我们将所有未命中的请求转交给入口脚本来处理:

```
location / {
    try_files $uri $uri/ /index.php?$args;
}

# fastcgi.conf
fastcgi_split_path_info ^(.+?\.php)(/.*)$;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

以 Nginx 为例，try_files 会依次尝试处理：

1. /post/view.html ，这个如果真有一个也就罢了，但其实多半不存在。
2. /post/view.html/ ，这个目录一般也不存在。
3. /index.php ，这个正是入口脚本，可以处理。至此，Nginx 与 Yii 顺利交接。

由于请求最终交给入口脚本来处理，且我们隐藏了 URL 中入口脚本名，上述请求还原回来的话，应该是 http://www.digapge.com/index.php/post/view.html 。自然，这 post/view.html 就是 PATH_INFO 了。有关 PATH_INFO 的更多知识，请看Web 应用 Request 部分的内容。

好了，在 Yii 从 Web Server 取得控制权之后，就是我们大显身手的时候了。在解析过程中，UrlRule 主要做了这么几件事：

- 通过 PATH_INFO 还原请求，如去除假后缀，开头接上主机信息等。还原后的请求为 post/view 。
- 看看当前请求是否匹配规则，这个匹配包含了主机、路由、参数等各方面的匹配。如不匹配，说明规则不适用，返回 false 。在这个例子中，规则并未定义主机信息方面的规则，规则中 $pattern = '#^post/(?P<action>\w+)(/(?P<id>\d+))?$#u' 。这与还原后的请求完全匹配。如果 URL 没有使用默认值 id = 100 ，如 post/view/101.html ，也是同样匹配的。
- 看看请求是否提供了规则已定义了默认值的所有参数，如果未提供，视为请求提供了这些参数，且他的值为默认值。这里 URL 中并未提供 id 参数，所以，视为他提供了 id = 100 的参数。简单粗暴而有效。
- 使用规则定义的路由进行转换，生成新的路由。再把上一步及当前所有参数作为路由的参数，共同组装成一个完整路由。

具体的转换流程可以看看UrlRule 路由规则解析 URL 的过程示意图 。

5.2 Url **管理**

在 Web 开发中，对于 URL 有一些共性的需求，如：

- 统一、简洁的 URL 创建方式
- URL 的伪静态化（美化）处理
- 从 URL 中解析出相应的路由信息，引导应用执行后续处理

这些功能在前面我们讲的 UrlRule 层面已经得到了一定程度的实现。但从层次上来讲，UrlRule 更偏向于基础一些，直接使用 UrlRule 相对而言还不是很方便。

比如，针对各种类型的 URL，我们需要提供相应的 UrlRule 实例来进行处理。这些实例如何进行统一管理，相互关系怎么处理？都无法在 UrlRule 自身层面解决。

我们需要更贴近开发的接口。于是 Yii 把 Web 应用中对于 URL 的常用要求抽象到了 urlManager 中，并作为 Web 应用的核心组件，更便于开发者使用。

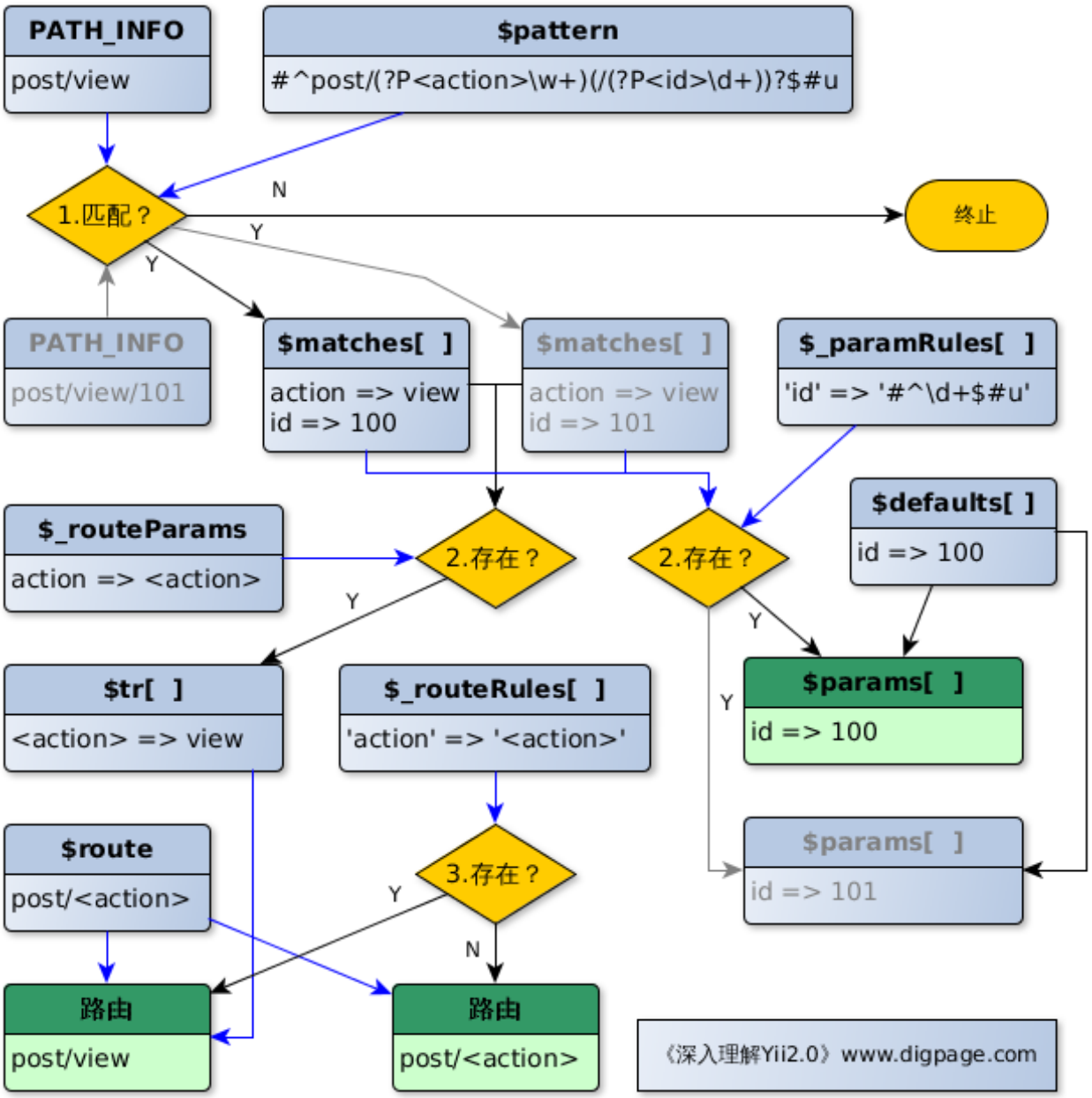

Fig. 5.3: UrlRule 路由规则解析 URL 的过程示意图

5.2.1 urlManager 概览

urlManager 组件由 yii\web\UrlManager 类定义:

```
class UrlManager extends Component
{

    // 用于表明 urlManager 是否启用 URL 美化功能，在 Yii1.1 中称为 path 格式 URL，
    // Yii2.0 中改称美化。
    // 默认不启用。但实际使用中，特别是产品环境，一般都会启用。
    public $enablePrettyUrl = false;

    // 是否启用严格解析，如启用严格解析，要求当前请求应至少匹配 1 个路由规则，
    // 否则认为是无效路由。
    // 这个选项仅在 enablePrettyUrl 启用后才有效。
    public $enableStrictParsing = false;

    // 保存所有路由规则的配置数组，并不在这里保存路由规则的实例
    public $rules = [];

    // 指定续接在 URL 后面的一个后缀，如 .html 之类的。仅在 enablePrettyUrl 启用时有效。
    public $suffix;

    // 指定是否在 URL 在保留入口脚本 index.php
    public $showScriptName = true;

    // 指定不启用 enablePrettyUrl 情况下，URL 中用于表示路由的查询参数，默认为 r
    public $routeParam = 'r';

    // 指定应用的缓存组件 ID，编译过的路由规则将通过这个缓存组件进行缓存。
    // 由于应用的缓存组件默认为 cache ，所以这里也默认为 cache 。
    // 如果不想使用缓存，需显式地置为 false
    public $cache = 'cache';

    // 路由规则的默认配置，注意上面的 rules[] 中的同名规则，优先于这个默认配置的规则。
    public $ruleConfig = ['class' => 'yii\web\UrlRule'];

    private $_baseUrl;
    private $_scriptUrl;
    private $_hostInfo;

    // urlManager 初始化
    public function init()
    {
        parent::init();
```

```
42          // 如果未启用 enablePrettyUrl 或者没有指定任何的路由规则，
43          // 这个 urlManager 不需要进一步初始化。
44          if (!$this->enablePrettyUrl || empty($this->rules)) {
45              return;
46          }
47
48          // 初始化前， $this->cache 是缓存组件的 ID，是个字符串，需要获取其实例。
49          if (is_string($this->cache)) {
50              // 如果获取不到实例，说明应用不提供缓存功能，
51              // 那么置这个 $this->cache 为 false
52              $this->cache = Yii::$app->get($this->cache, false);
53          }
54
55          // 如果顺利引用到了缓存组件，那么就将路由规则缓存起来
56          if ($this->cache instanceof Cache) {
57
58              // 以当前 urlManager 类的类名为缓存的键
59              $cacheKey = __CLASS__;
60              // urlManager 所有路由规则转换为 json 格式编码后的 HASH 值，
61              // 用于确保缓存中的路由规则没有变化。
62              // 即外部没有对已经缓存起来的路由规则有增加、修改、
63              // 删除、调整前后位置等操作。
64              $hash = md5(json_encode($this->rules));
65
66              // cache 中是一个数组，0 号元素用于缓存创建好的路由规则，
67              // 1 号元素用于保存 HASH 值。这个判断用于确认是否有缓存、且缓存仍有效。
68              // 是的话，直接使用缓存中的内容作为当前的路由规则数组。
69              if (($data = $this->cache->get($cacheKey)) !== false
70                  && isset($data[1]) && $data[1] === $hash) {
71                  $this->rules = $data[0];
72
73              // 如果尚未缓存或路由规则已经被修改导致缓存失效，
74              // 那么重新创建路由规则并缓存。
75              } else {
76                  $this->rules = $this->buildRules($this->rules);
77                  $this->cache->set($cacheKey, [$this->rules, $hash]);
78              }
79
80          // 要么是应用不提供缓存功能，要么是开发者将 $this->cache 手动置为 false，
81          // 总之，就是不使用缓存。那么就直接创建吧，也无需缓存了。
82          } else {
83              $this->rules = $this->buildRules($this->rules);
84          }
85      }
86
```

```
87      // 增加新的规则
88      public function addRules($rules, $append = true){ ... }
89
90      // 创建路由规则
91      protected function buildRules($rules){ ... }
92
93      // 用于解析请求
94      public function parseRequest($request){ ... }
95
96      // 这 2 个用于创建 URL
97      public function createUrl($params){ ... }
98      public function createAbsoluteUrl($params, $scheme = null){ ... }
99  }
```

在 urlManager 的使用上，用得最多的配置项就是：

- $enablePrettyUrl，是否开启 URL 美化功能。关于美化功能，我们在路由 (Route) 部分已经介绍过了。注意如果 $enablePrettyUrl 不开启，表明使用原始的格式，那么所有路由规则都是无效的。
- $showScriptName，是否在 URL 中显示入口脚本。是对美化功能的进一步补充。
- suffix 设置一个 .html 之类的假后缀，是对美化功能的进一步补充。
- rules 保存路由规则们的声明，注意并非保存其实例。
- $enableStrictParsing 是否开启严格解析。该选项仅在开启美化功能后生效。在开启严格解析模式时，所有请求必须匹配 $rules[] 所声明的至少一个路由规则。如果未开启，请求的 PATH_INFO 部分将作为所请求的路由进行后续处理。

在 UrlManager::init() 初始化过程中，可以发现 urlManager 使用了应用所提供的缓存组件（有果有的话），对所有路由规则的实例进行缓存。

从架构上来讲，将所有请求交由入口脚本统一接收，再分发到相应模块进行处理的这种方式，就注定了入口脚本有产生性能瓶颈的可能。但是带来的开发上的便利，却是实实在在的。可以想像，在由 Web Server 进行请求分发的情景下，每个接收请求的脚本都要执行相同或类似的代码，这会造成很冗余。而且会将权限控制、日志记录等逻辑上就应当作为所有请求第一关的的模块都分散到各处去。

因此，目前这种单一入口脚本的设计成为事实上的标准，几乎所有的 Web 开发框架都采用这种方式。但这同时也对各框架的性能提出挑战。

在前面讲路由规则时，我们就体会到了初始化过程的繁琐，转换来转换去的。如果采用简单粗暴的方式，Yii 完全可以牺牲一定的开发便利性，在代码层面提高路由规则的性能。比如，直接使用正则表达式。

但是，Yii 没有这样做，而是很好地平稳了性能与开发便利性，通过将路由规则进行缓存来克服这个瓶颈。

TBD

5.3 请求 (Reqeust)

5.3.1 获取用户请求

PHP 并未提供集中的、统一的界面以获取用户请求，而是分散在 $_SERVER $_POST 等变量和其他代码中。万能的 Yii 怎么会允许群雄割据这种局面出现呢？他肯定是要一统江湖的。那么对于任何 Yii 应用而言，初始化后第一件正事，就是获取用户请求。这个代码在 yii\base\Application::run() 中:

```
public function run()
{
    try {

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

        $this->state = self::STATE_HANDLING_REQUEST;

        // 获取用户请求，并进行处理，处理的过程也是产生响应内容的过程
        $response = $this->handleRequest($this->getRequest());

        $this->state = self::STATE_AFTER_REQUEST;
        $this->trigger(self::EVENT_AFTER_REQUEST);

        $this->state = self::STATE_SENDING_RESPONSE;

        // 将响应内容发送回用户
        $response->send();

        $this->state = self::STATE_END;

        return $response->exitStatus;

    } catch (ExitException $e) {

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

    }
}
```

上面的代码主要看注释的两个地方，聪明的读者朋友们一定都猜出来了，$this->getRequest() 就是用于获取用户请求的嘛。

其实这是一个 getter，用于获取 Application 的 request 组件 (component) 。Yii 用这个组件来代表用户请求，他承载着所有的用户输入信息。

我们知道，Yii 应用有命令行 (Console) 应用和 Web 应用之分。因此，这个 Request 类其实涉及到了以下的类：

- yii\base\Request Request 类基类
- yii\console\Request 表示 Console 应用的的 Request
- yii\web\Request 表示 Web 应用的 Request

下面我们逐一进行讲解。

5.3.2 **基类** Request

基类是对 Console 应用和 Web 应用 Request 的抽象，他仅仅定义了两个属性和一个虚函数:

```php
abstract class Request extends Component
{
    // 属性 scriptFile，用于表示入口脚本
    private $_scriptFile;

    // 属性 isConsoleRequest，用于表示是否是命令行应用
    private $_isConsoleRequest;


    // 虚函数，要求子类来实现
    // 这个函数的功能主要是为了把 Request 解析成路由和相应的参数
    abstract public function resolve();

    // isConsoleRequest 属性的 getter 函数
    // 使用 PHP_SAPI 常量判断当前应用是否是命令行应用
    public function getIsConsoleRequest()
    {
        // 一切 PHP_SAPI 不为 'cli' 的，都不是命令行
        return $this->_isConsoleRequest !== null ?
            $this->_isConsoleRequest : PHP_SAPI === 'cli';
    }

    // isConsoleRequest 属性的 setter 函数
    public function setIsConsoleRequest($value)
    {
        $this->_isConsoleRequest = $value;
    }

    // scriptFile 属性的 getter 函数
    // 通过 $_SERVER['SCRIPT_FILENAME'] 来获取入口脚本名
    public function getScriptFile()
    {
```

```
        if ($this->_scriptFile === null) {
            if (isset($_SERVER['SCRIPT_FILENAME'])) {
                $this->setScriptFile($_SERVER['SCRIPT_FILENAME']);
            } else {
                throw new InvalidConfigException(
                    'Unable to determine the entry script file path.');
            }
        }

        return $this->_scriptFile;
    }

    // scriptFile 属性的 setter 函数
    public function setScriptFile($value)
    {
        $scriptFile = realpath(Yii::getAlias($value));
        if ($scriptFile !== false && is_file($scriptFile)) {
            $this->_scriptFile = $scriptFile;
        } else {
            throw new InvalidConfigException(
                'Unable to determine the entry script file path.');
        }
    }
}
```

yii\base\Request 通过 getter 和 setter 提供了两个可读写的属性，isConsoleRequest 和 scriptFile 。同时，要求子类实现一个 resolve() 方法。

基类的代码相对简单，主要涉及到 PHP 的一些知识，如 PHP_SAPI $_SERVER['SCRIPT_FILENAME'] 等，读者朋友们可以通过搜索引擎或 PHP 手册了解下相关的知识，相信上面的代码难不倒你们的。

5.3.3 **命令行应用** Request

命令行应用 Request 由 yii\console\Request 负责实现，相比较于 yii\base\Request 稍有丰富：

```
class Request extends \yii\base\Request
{
    // 属性 params，用于表示命令行参数
    private $_params;

    // params 属性的 getter 函数
    // 通过 $_SERVER['argv'] 来获取命令行参数
    public function getParams()
    {
```

```
10          if (!isset($this->_params)) {
11              if (isset($_SERVER['argv'])) {
12                  $this->_params = $_SERVER['argv'];
13
14                  // 删除数组的第一个元素，这个元素是 PHP 脚本名。
15                  // 因此，属性 params 中全部是参数，不带脚本名
16                  array_shift($this->_params);
17              } else {
18                  $this->_params = [];
19              }
20          }
21
22          return $this->_params;
23      }
24
25      // params 属性的 setter 函数
26      public function setParams($params)
27      {
28          $this->_params = $params;
29      }
30
31      // 父类虚函数的实现
32      public function resolve()
33      {
34          // 获取全部的命令行参数
35          $rawParams = $this->getParams();
36
37          // 第一个命令行参数作为路由
38          if (isset($rawParams[0])) {
39              $route = $rawParams[0];
40              array_shift($rawParams);
41          } else {
42              $route = '';
43          }
44
45          $params = [];
46
47          // 遍历剩余的全部命令行参数
48          foreach ($rawParams as $param) {
49
50              // 正则匹配每一个参数
51              if (preg_match('/^--(\w+)(=(.*))?$/', $param, $matches)) {
52
53                  // 参数名
54                  $name = $matches[1];
```

```
55
56              // yii\console\Application::OPTION_APPCONFIG = 'appconfig'
57              if ($name !== Application::OPTION_APPCONFIG) {
58                  $params[$name] = isset($matches[3]) ? $matches[3] : true;
59              }
60
61          // 无名参数，直接作为参数值
62          } else {
63              $params[] = $param;
64          }
65      }
66
67      return [$route, $params];
68  }
69  }
```

相比较于 yii\base\Request ，yii\console\Request 提供了一个 params 属性，该属性以数组形式保存了入口脚本 Yii 的命令行参数。这是通过 $_SERVER['argv'] 获取的。注意 params 属性不保存入口脚本名。入口脚本名由基类的 scriptName 属性保存。

同时，yii\console\Request 还实现了父类的 resolve() 虚函数，这个函数主要做了这么几件事：

- 将 params 属性的第一个元素作为路由。如果入口脚本未提供任何参数，也即 params 是个空数组，那么将路由置为一个空字符串。
- 遍历 params 中剩余的参数，使用正则匹配 Yii 应用的参数名和参数值，看看是不是 --参数名 = 参数值形式。其中，以 -- 打头的任意字母、数字、下划线的组合，就是参数名。紧跟参数名的 = 后面的内容，则为参数值。对于仅有参数名，没有参数值的，视参数值为 true 。
- 如果正则匹配不成功，则将这个命令行参数作为 Yii 应用的一个无名参数的值。
- 如果第二步中的参数名为 appconfig 则忽略该参数，Console Application 会专门针对该参数进行处理。
- 上面步骤中的参数和参数值，被保存进一个数组中。数组的键表示参数名，数组的值表示参数值。
- 最终 resolve() 返回一个数组，第一个元素是一个表示路由的字符串，第二元素则是参数数组。该方法由 Application 在处理 Request 时调用。

关于 appconfig 参数的问题，只要在调用 yii 时，指定了 appconfig 参数，就表明不使用默认的参数配置文件，而使用该参数所指定的配置文件。相关的代码在 yii\console\Application 中：

```
1  // 定义一个常量
2  const OPTION_APPCONFIG = 'appconfig';
3
4  // yii\console\Application 类的构造函数
5  public function __construct($config = [])
6  {
7      // 重点看这句，会调用 loadConfig() 成员函数
```

```
8       $config = $this->loadConfig($config);
9       parent::__construct($config);
10  }
11
12  // 如果指定的配置文件存在，那么返回其配置数组
13  // 否则，返回构造函数调用时的数组
14  protected function loadConfig($config)
15  {
16      if (!empty($_SERVER['argv'])) {
17
18          // 设定了一个字符串 "--appconfig="
19          $option = '--' . self::OPTION_APPCONFIG . '=';
20
21          // 遍历所有命令行参数，看看能不能找到上面说的这个字符串
22          foreach ($_SERVER['argv'] as $param) {
23              if (strpos($param, $option) !== false) {
24
25                  // 截取参数值部分
26                  $path = substr($param, strlen($option));
27                  if (!empty($path) && is_file($file = Yii::getAlias($path))) {
28
29                      // 将指定文件的内容引入进来
30                      return require($file);
31                  } else {
32                      die("The configuration file does not exist: $path\n");
33                  }
34              }
35          }
36      }
37
38      return $config;
39  }
```

讲完了 Request 基类和命令行应用的 Request 只是热身而已，接下来要讲的 Web 应用 Request 才是重头。毕竟最最主要的，还是 Web 开发嘛。考虑到 Web Request 的内容较多，还是单独成Web 应用 Request 来讲吧。

5.4 Web **应用** Request

前面请求 (Reqeust) 部分我们讲了用户请求的基础知识和命令行应用的 Request，接下来继续讲 Web 应用的 Request。

Web 应用 Request 由 yii\web\Request 实现，这个类的代码将近 1400 行，主要是一些功能的封装罢了，原理上没有很复杂的东西。只是涉及到许多 HTTP 的有关知识，读者朋友们可以自行查看相关的规范文档，

如 HTTP 1.1 协议[1]，CGI 1.1 规范[2] 等。

同时，Yii 大量引用了 $_SERVER，具体可以查看 PHP 文档关于 $_SERVER 的内容[3]，此外，还涉及到 PHP 运行于不同的环境和模式下的一些细微差别。这些内容比较细节，不影响大局，但是很影响理解，不过没关系，我们在涉及到的时候，会点一点。

5.4.1 请求的方法

根据 HTTP 1.1 协议[4]，HTTP 的请求可以有：GET, POST, PUT 等 8 种方法 (Request Method)。除了用不到的 CONNECT 外，Yii 支持全部的 HTTP 请求方法。

要获取当前用户请求的方法，可以使用 yii\web\Request::getMethod()

```php
// 返回当前请求的方法，请留意方法名称是大小写敏感的，按规范应转换为大写字母
public function getMethod()
{
    // $this->methodParam 默认值为 '_method'
    // 如果指定 $_POST['_method']，表示使用 POST 请求来模拟其他方法的请求。
    // 此时 $_POST['_method'] 即为所模拟的请求类型。
    if (isset($_POST[$this->methodParam])) {
        return strtoupper($_POST[$this->methodParam]);

    // 或者使用 $_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE'] 的值作为方法名。
    } elseif (isset($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE'])) {
        return strtoupper($_SERVER['HTTP_X_HTTP_METHOD_OVERRIDE']);

    // 或者使用 $_SERVER['REQUEST_METHOD'] 作为方法名，未指定时，默认为 GET 方法
    } else {
        return isset($_SERVER['REQUEST_METHOD']) ?
            strtoupper($_SERVER['REQUEST_METHOD']) : 'GET';
    }
}
```

这个方法使用了 3 种方法来获取当前用户的请求，优先级从高到低依次为：

- 当使用 POST 请求来模拟其他请求时，以 $_POST['_method'] 作为当前请求的方法；
- 否则，如果存在 X_HTTP_METHOD_OVERRIDE HTTP 头时，以该 HTTP 头所指定的方法作为请求方法，如 X-HTTP-Method-Override: PUT 表示该请求所要执行的是 PUT 方法；
- 如果 X_HTTP_METHOD_OVERRIDE 不存在，则以 REQUEST_METHOD 的值作为当前请求的方法。如果连 REQUEST_METHOD 也不存在，则视该请求是一个 GET 请求。

前面两种方法，主要是针对一些只支持 GET 和 POST 等有限方法的 User Agent 而设计的。

[1] https://tools.ietf.org/html/rfc2616
[2] https://tools.ietf.org/html/rfc3875.html
[3] http://php.net/manual/en/reserved.variables.server.php
[4] https://tools.ietf.org/html/rfc2616

其中第一种方法是从 Ruby on Rails 中借鉴过来的，通过在发送 POST 请求时，加入一个 $_POST['_method'] 的隐藏字段，来表示所要模拟的方法，如 PUT，DELETE 等。这样，就可以使得这些功能有限的 User Agent 也可以正常与服务器交互。这种方法胜在简便，随手就来。

第二种方法则是使用 X_HTTP_METHOD_OVERRIDE HTTP 头的办法来指定所使用的请求类型。这种方法胜在直接明了，约定俗成，更为规范、合理。

至于 REQUEST_METHOD 是 CGI 1.1 规范[5] 所定义的环境变量，专门用来表明当前请求方法的。上面的代码只是在未指定时默认为 GET 请求罢了。

当然，我们在开发过程中，其实并不怎么在乎当前的用户请求是什么类型的请求，我们更在乎是不是某一类型的请求。比如，对于同一个 URL 地址 http://api.digpage.com/post/123 ，如果是正常的 GET 请求，应该是查看编号为 123 的文章的意思。但是如果是一个 DELETE 请求，则是表示删除编号为 123 的文章的意思。我们在开发中，很可能就会这么写:

```
if ($app->request->isDelete()){
    $post->delete();
} else {
    $post->view();
}
```

上面的代码只是一个示意，与实际编码是有一定出入的，主要看判断分支的用法。就是判断请求是否是某一特定类型的请求。这些判断在实际开发中，是很常用的。于是 Yii 为我们封装了许多方法专门用于执行这些判断：

- getIsAjax() 是否是 AJAX 请求，这其实不是 HTTP 请求方法，但是实际使用上，这个是用得最多的。
- getIsDelete() 是否是 DELETE 请求
- getIsFlash() 是否是 Adobe Flash 或 Adobe Flex 发出的请求，这其实也不是 HTTP 请求方法。
- getIsGet() 是否是一个 GET 请求
- getIsHead() 是否是一个 HEAD 请求
- getIsOptions() 是否是一个 OPTIONS 请求
- getIsPatch() 是否是 PATCH 请求
- getIsPjax() 是否是一个 PJAX 请求，这也并非是 HTTP 请求方法。
- getIsPost() 是否是一个 POST 请求
- getIsPut() 是否是一个 PUT 请求

上面 10 个方法请留意其中有 3 个并未是 HTTP 请求方法，主要是用于特定 HTTP 请求类型（AJAX、Flash、PJAX）的判断。

除了这 3 个之外的其余 7 个方法，正好对应于 HTTP 1.1 协议定义的 7 个方法。而 CONNECT 方法由于 Web 开发在用不到，主要用于 HTTP 代理，因此，Yii 也就没有为其设计一个所谓的 isConnect() 了，

[5]https://tools.ietf.org/html/rfc3875.html

这是无用功。

上面的 10 个方法，再加一开始说的 getMehtod() 一共是 11 个方法，按照我们在属性（Property）部分所说的，这相当于定义了 11 个只读属性。我们以其中几个为例，看看具体实现：

```
// 这个 SO EASY，啥也不说了，Yii 实现的 7 个 HTTP 方法都是这个路子。
public function getIsOptions()
{
    // 注意在 getMethod() 时，输出的是全部大写的字符串
    return $this->getMethod() === 'OPTIONS';
}

// AJAX 请求是通过 X_REQUESTED_WITH 消息头来判断的
public function getIsAjax()
{
    // 注意这里的 XMLHttpRequest 没有全部大写
    return isset($_SERVER['HTTP_X_REQUESTED_WITH']) &&
        $_SERVER['HTTP_X_REQUESTED_WITH'] === 'XMLHttpRequest';
}

// PJAX 请求是 AJAX 请求的一种，增加了 X_PJAX 消息头的定义
public function getIsPjax()
{
    return $this->getIsAjax() && !empty($_SERVER['HTTP_X_PJAX']);
}

// HTTP_USER_AGENT 消息头中包含 'Shockwave' 或 'Flash' 字眼的（不区分大小写），
// 就认为是 FLASH 请求
public function getIsFlash()
{
    return isset($_SERVER['HTTP_USER_AGENT'])
        && (stripos($_SERVER['HTTP_USER_AGENT'], 'Shockwave') !== false
        || stripos($_SERVER['HTTP_USER_AGENT'], 'Flash') !== false);
}
```

上面提到的 AJAX、PJAX、FLASH 请求比较特殊，并非是 HTTP 协议所规定的请求类型，但是在实现中是会使用到的。比如，对于一个请求，在非 AJAX 时，需要整个页面返回给客户端，而在 AJAX 请求时，只需要返回页面片段即可。

这些特殊请求是通过特殊的消息头实现的，具体的可以自行搜索相关的定义和规范。至于那 7 个 HTTP 方法的判断，摆明了是同一个路子，换瓶不换酒，getMethod() 前人栽树，他们后人乘凉。

5.4.2 请求的参数

在实际开发中，开发者如果需要引用 request，最常见的情况是为了获取请求参数，以便作相应处理。PHP 有众所周知的 $_GET 和 $_POST 等。相应地，Yii 提供了一系列的方法用于获取请求参数：

```php
// 用于获取 GET 参数，可以指定参数名和默认值
public function get($name = null, $defaultValue = null)
{
    if ($name === null) {
        return $this->getQueryParams();
    } else {
        return $this->getQueryParam($name, $defaultValue);
    }
}

// 用于获取所有的 GET 参数
// 所有的 GET 参数保存在 $_GET 或 $this->_queryParams 中。
public function getQueryParams()
{
    if ($this->_queryParams === null) {
        // 请留意这里并未使用 $this->_queryParams = $_GET 进行缓存。
        // 说明一旦指定了 $_queryParams 则 $_GET 会失效。
        return $_GET;
    }

    return $this->_queryParams;
}

// 根据参数名获取单一的 GET 参数，不存在时，返回指定的默认值
public function getQueryParam($name, $defaultValue = null)
{
    $params = $this->getQueryParams();
    return isset($params[$name]) ? $params[$name] : $defaultValue;
}

// 类以于 get()，用于获取 POST 参数，也可以指定参数名和默认值
public function post($name = null, $defaultValue = null)
{
    if ($name === null) {
        return $this->getBodyParams();
    } else {
        return $this->getBodyParam($name, $defaultValue);
    }
}

// 根据参数名获取单一的 POST 参数，不存在时，返回指定的默认值
public function getBodyParam($name, $defaultValue = null)
{
    $params = $this->getBodyParams();
    return isset($params[$name]) ? $params[$name] : $defaultValue;
```

```
}

// 获取所有 POST 参数，所有 POST 参数保存在 $this->_bodyParams 中
public function getBodyParams()
{
    if ($this->_bodyParams === null) {
        // 如果是使用 POST 请求模拟其他请求的
        if (isset($_POST[$this->methodParam])) {
            $this->_bodyParams = $_POST;

            // 将 $_POST['_method'] 删掉，剩余的 $_POST 就是了
            unset($this->_bodyParams[$this->methodParam]);
            return $this->_bodyParams;
        }

        // 获取 Content Type
        // 对于 'application/json; charset=UTF-8'，得到的是 'application/json'
        $contentType = $this->getContentType();
        if (($pos = strpos($contentType, ';')) !== false) {
            $contentType = substr($contentType, 0, $pos);
        }

        // 根据 Content Type 选择相应的解析器对请求体进行解析
        if (isset($this->parsers[$contentType])) {

            // 创建解析器实例
            $parser = Yii::createObject($this->parsers[$contentType]);
            if (!($parser instanceof RequestParserInterface)) {
                throw new InvalidConfigException(
                "The '$contentType' request parser is invalid.
                It must implement the yii\\web\\RequestParserInterface.");
            }

            // 将请求体解析到 $this->_bodyParams
            $this->_bodyParams = $parser->parse($this->getRawBody(), $contentType);

        // 如果没有与 Content Type 对应的解析器，使用通用解析器
        } elseif (isset($this->parsers['*'])) {
            $parser = Yii::createObject($this->parsers['*']);
            if (!($parser instanceof RequestParserInterface)) {
                throw new InvalidConfigException(
                "The fallback request parser is invalid.
                It must implement the yii\\web\\RequestParserInterface.");
            }
            $this->_bodyParams = $parser->parse($this->getRawBody(),
```

```
91              $contentType);
92
93          // 连通用解析器也没有
94          // 看看是不是 POST 请求，如果是，PHP 已经将请求参数放到 $_POST 中了，直接用就 OK 了
95          } elseif ($this->getMethod() === 'POST') {
96              $this->_bodyParams = $_POST;
97
98          // 以上情况都不是，那就使用 PHP 的 mb_parse_str() 进行解析
99          } else {
100             $this->_bodyParams = [];
101             mb_parse_str($this->getRawBody(), $this->_bodyParams);
102         }
103     }
104
105     return $this->_bodyParams;
106 }
```

在上面的代码中，将所有的请求参数划分为两类，一类是包含在 URL 中的，称为查询参数（Query Parameter），或 GET 参数。另一类是包含在请求体中的，需要根据请求体的内容类型（Content Type）进行解析，称为 POST 参数。

其中，get() , getQueryParams() 和 getQueryParam() 用于获取查询参数：

- get() 用于获取 GET 参数，可以指定所要获取的特定参数的参数名，在这个参数名不存在时，可以指定默认值。当不指定参数名时，获取所有的 GET 参数。具体功能是由下面 2 个函数来实现的。
- getQueryParams() 用于获取所有的 GET 参数。这些参数的内容，保存在 $_GET 或 $this->_queryParams 中。优先使用 $this->_queryParams 的。
- getQueryParam() 对应于 get() 用于获取特定的 GET 参数的情况。

而 post() , getPostParams() 和 getPostParam() 用于获取 POST 参数：

- post() 与 get() 类似，可以指定所要获取的特定参数的参数名，在这个参数名不存在时，可以指定默认值。当不指定参数名时，获取所有的 POST 参数。具体功能是由下面 2 个函数来实现的。
- getPostParam() 用于通过参数名获取特定的 POST 参数，需要调用 getPostParams() 获取所有的 POST 参数。
- getPostParams() 用于获取所有的 POST 参数。

上面稍微复杂点的，可能就是 getPostParams() 了，我们就稍稍剖析下 Yii 是怎么解析 POST 参数的。先讲讲这个方法所涉及到的一些东东：内容类型、请求解析器、请求体。

内容类型（Content-Type）

在 getPostParams() 中，需要先获取请求体的内容类型，然后采用相应的解析器对内容进行解析。

获取内容类型，使用 getContentType()

```
public function getContentType()
{
    if (isset($_SERVER["CONTENT_TYPE"])) {
        return $_SERVER["CONTENT_TYPE"];
    } elseif (isset($_SERVER["HTTP_CONTENT_TYPE"])) {
        return $_SERVER["HTTP_CONTENT_TYPE"];
    }

    return null;
}
```

根据 CGI 1.1 规范[6]，内容类型由 CONTENT_TYPE 环境变量来表示。而根据 HTTP 1.1 协议[7]，内容类型则是放在 CONTENT_TYPE 头部中，然后由 PHP 赋值给 $_SERVER['HTTP_CONTENT_TYPE']。这里一般没有冲突，因此发现哪个用哪个，就怕客户端没有给出（这种情况返回 null）。

请求解析器

在 getPostParams() 中，根据不同的 Content Type 创建了相应的内容解析器对请求体进行解析。yii\web\Request 使用成员变量 public $parsers 来保存一系列的解析器。这个变量在配置时进行指定:

```
'request' => [
    ... ...
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ],
]
```

$parsers 是一个数组，数组的键是 Content Type，如 applicaion/json 之类。而数组的值则是对应于特定 Content Type 的解析器，如 yii\web\JsonParser 。这也是 Yii 实现的唯一一个现成的 Parser，其他 Content-Type，需要开发者自己写了。

而且，可以以 * 为键指定一个解析器。那么该解析器将在一个 Content Type 找不到任何匹配的解析器后被使用。

yii\web\JsonParser 其实很简单:

```
namespace yii\web;

use yii\base\InvalidParamException;
use yii\helpers\Json;

// 所有的解析器都要实现 RequestParserInterface
```

[6] https://tools.ietf.org/html/rfc3875.html
[7] https://tools.ietf.org/html/rfc2616

```php
// 这个接口也只是要求实现 parse() 方法
class JsonParser implements RequestParserInterface
{
    public $asArray = true;
    public $throwException = true;

    // 具体实现 parse()
    public function parse($rawBody, $contentType)
    {
        try {
            return Json::decode($rawBody, $this->asArray);
        } catch (InvalidParamException $e) {
            if ($this->throwException) {
                throw new BadRequestHttpException(
                    'Invalid JSON data in request body: '
                    . $e->getMessage(), 0, $e);
            }

            return null;
        }
    }
}
```

这里使用 yii\helpers\Json::decode() 对请求体进行解析。这个 yii\helpers\Json 是个辅助类，专门用于处理 JSON 格式数据。具体的内容我们这里就不做讲解了，只需要了解这里可以将 JSON 格式数据解析出来就 OK 了，学有余力的读者朋友可以自己看看代码。

请求体

在 yii\web\Reqeust::getBodyParams() 和 yii\web\RequestParserInterface::parse() 中，我们可以看到，需要将请求体传入 parse() 进行解析，且请求体由 yii\web\Request::getRawBody() 可得。

yii\web\Request::getRawBody():

```php
public function getRawBody()
{
    if ($this->_rawBody === null) {
        $this->_rawBody = file_get_contents('php://input');
    }

    return $this->_rawBody;
}
```

这个方法使用了 php://input 来获取请求体，这个 php://input 有这么几个特点：

- php://input 是个只读流，用于获取请求体。

- php://input 是返回整个 HTTP 请求中，除去 HTTP 头部的全部原始内容，而不管是什么 Content Type（或称为编码方式）。相比较之下，$_POST 只支持 application/x-www-form-urlencoded 和 multipart/form-data-encoded 两种 Content Type。其中前一种就是简单的 HTML 表单以 method="post" 提交时的形式，后一种主要是用于上传文档。因此，对于诸如 application/json 等 Content Type，这往往是在 AJAX 场景下使用，那么使用 $_POST 得到的是空的内容，这时就必须使用 php://input 。
- 相比较于 $HTTP_RAW_POST_DATA , php://input 无需额外地在 php.ini 中激活 always-populate-raw-post-data ，而且对于内存的压力也比较小。
- 当编码方式为 multipart/form-data-encoded 时，php://input 是无效的。这种情况一般为上传文档。这种情况可以使用传统的 $_FILES 或者 yii\web\UploadedFile 。

5.4.3 请求的头部

yii\web\Request 使用一个成员变量 private $_headers 来存储请求头。而这个 $_header 其实是一个 yii\web\HeaderCollection , 这是一个集合类的基本数据结构, 实现了 SPL 的 IteratorAggregate , ArrayAccess 和 Countable 等接口。因此，这个集合可以进行迭代、像数组一样进行访问、可被用于 conut() 函数等。

这个数据结构相对简单，我们就不展开占用篇幅了。我们要讲的是怎么获取请求的头部。这个是由 yii\web\Request::getHeaders() 来实现的:

```
public function getHeaders()
{
    if ($this->_headers === null) {

        // 实例化为一个 HeaderCollection
        $this->_headers = new HeaderCollection;

        // 使用 getallheaders() 获取请求头部，以数组形式返回
        if (function_exists('getallheaders')) {
            $headers = getallheaders();

        // 使用 http_get_request_headers() 获取请求头部，以数组形式返回
        } elseif (function_exists('http_get_request_headers')) {
            $headers = http_get_request_headers();

        // 使用 $_SERVER 数组获取头部
        } else {
            foreach ($_SERVER as $name => $value) {

                // 针对所有 $_SERVER['HTTP_*'] 元素
                if (strncmp($name, 'HTTP_', 5) === 0) {
                    // 将 HTTP_HEADER_NAME 转换成 Header-Name 的形式
                    $name = str_replace(' ', '-',
```

```
                    ucwords(strtolower(str_replace('_', ' ',
                        substr($name, 5)))));
                $this->_headers->add($name, $value);
            }
        }

        return $this->_headers;
    }

    // 将数组形式的请求头部变成集合的元素
    foreach ($headers as $name => $value) {
        $this->_headers->add($name, $value);
    }
  }

  return $this->_headers;
}
```

这里用 3 种方法来尝试获取请求的头部：

- getallheaders()，这个方法仅在将 PHP 作为 Apache 的一个模块运行时有效。
- http_get_request_headers()，要求 PHP 启用 HTTP 扩展。
- $_SERVER 数组的方法，需要遍历整个数组，并将所有以 HTTP_* 元素加入到集合中去。并且，要将所有 HTTP_HEADER_NAME 转换成 Header-Name 的形式。

就是根据不同的 PHP 环境，采用有效的方法来获取请求头部，如此而已。

5.4.4 请求的解析

我们前面就说过了，无论是命令行应用还是 Web 应用，他们的请求都要实现接口要求的 resolve()，以便明确这个用户请求的路由和参数。下面就是 yii\web\Request::resolve() 的代码：

```
public function resolve()
{
  // 使用 urlManager 来解析请求
  $result = Yii::$app->getUrlManager()->parseRequest($this);
  if ($result !== false) {
    list ($route, $params) = $result;

    // 将解析出来的参数与 $_GET 参数进行合并
    $_GET = array_merge($_GET, $params);

    return [$route, $_GET];
  } else {
```

```
        throw new NotFoundHttpException(Yii::t('yii', 'Page not found.'));
    }
}
```

看着很简单吧？这才几行，还没有 getBodyParams() 的代码多呢。

虽然简单，但是有一个细节我们要留意，就是在解析出路由信息和参数的时候，会把参数的内容加入到 $_GET 中去，这是合理的。

比如，对于 http://www.digpage.com/post/view/100 这个 100 在路由规则中，其实定义为一个参数。其原始的形式应当是 http://www.digpage.com/index.php?r=post/view&id=100 。你说该不该把 id = 100 重新写回 $_GET 去？至于路由规则的内容，可以看看路由 (Route) 的内容。

从这个 resolve() 是看不出来解析过程的复杂的，这个 yii\web\Request::resolve() 是个没担当的家伙，他把解析过程推给了 urlManager。那我们就顺藤摸瓜，一睹这个 yii\web\UrlManager::parseRequest() 吧:

```
public function parseRequest($request)
{
    // 启用了 enablePrettyUrl 的情况
    if ($this->enablePrettyUrl) {

        // 获取路径信息
        $pathInfo = $request->getPathInfo();

        // 依次使用所有路由规则来解析当前请求
        // 一旦有一个规则适用，后面的规则就没有被调用的机会了
        foreach ($this->rules as $rule) {
            if (($result = $rule->parseRequest($this, $request)) !== false) {
                return $result;
            }
        }

        // 所有路由规则都不适用，又启用了 enableStrictParsing ,
        // 那只能返回 false 了。
        if ($this->enableStrictParsing) {
            return false;
        }

        // 所有路由规则都不适用，幸好还没启用 enableStrictParing,
        // 那就用默认的解析逻辑

        Yii::trace(
            'No matching URL rules. Using default URL parsing logic.',
            __METHOD__);

        // 配置时所定义的 fake suffix，诸如 ".html" 等
```

```
31          $suffix = (string) $this->suffix;
32
33          if ($suffix !== '' && $pathInfo !== '') {
34              // 这个分支的作用在于确保 $pathInfo 不能仅仅是包含一个 ".html"。
35
36              $n = strlen($this->suffix);
37              // 留意这个 -$n 的用法
38              if (substr_compare($pathInfo, $this->suffix, -$n, $n) === 0) {
39                  $pathInfo = substr($pathInfo, 0, -$n);
40
41                  // 仅包含 ".html" 的 $pathInfo 要之何用？掐死算了。
42                  if ($pathInfo === '') {
43                      return false;
44                  }
45
46              // 后缀没匹配上
47              } else {
48                  return false;
49              }
50          }
51
52          return [$pathInfo, []];
53
54      // 没有启用 enablePrettyUrl 的情况，那就更简单了，
55      // 直接使用默认的解析逻辑就 OK 了
56      } else {
57          Yii::trace(
58              'Pretty URL not enabled. Using default URL parsing logic.',
59              __METHOD__);
60          $route = $request->getQueryParam($this->routeParam, '');
61          if (is_array($route)) {
62              $route = '';
63          }
64
65          return [(string) $route, []];
66      }
67  }
```

从上面代码中可以看到，urlManager 是按这么一个顺序来解析用户请求的：

- 先判断是否启用了 enablePrettyUrl，如果没启用，所有的路由和参数信息都在 URL 的查询参数中，很简单就可以处理了。
- 通常都会启用 enablePrettyUrl，由于路由和参数信息部分或全部变成了 URL 路径。经过了美化，使得 URL 看起来更友好，但化妆品总是比清水芙蓉要烧银子，解析起来就有点费功夫了。

- 既然路由和参数信息变成了 URL 路径，那么就先从 URL 路径下手获取路径信息。
- 然后依次使用已经定义好的路由规则对当前请求进行解析，一旦有一个规则适用，后续的路由规则就不会起作用了。
- 然后再对配置的 .html 等 fake suffix 进行处理。

这一过程中，有两个重点，一个是获取路径信息，另一个就是使用路由规则对请求进行解析。下面我们依次进行讲解。

获取路径信息

在大多数情况下，我们还是会启用 enablePrettyUrl 的，特别是在产品环境下。那么从上面的代码来看，yii\web\Request::getPathInfo() 的调用就不可避免。其实涉及到获取路径信息的方法有很多，都在 yii\web\Request 中，这里暴露出来的，只是一个 getPathInfo()，相关的方法有:

```
// 这个方法其实是调用 resolvePathInfo() 来获取路径信息的
public function getPathInfo()
{
    if ($this->_pathInfo === null) {
        $this->_pathInfo = $this->resolvePathInfo();
    }

    return $this->_pathInfo;
}

// 这个才是重点
protected function resolvePathInfo()
{
    // 这个 getUrl() 调用的是 resolveRequestUri() 来获取当前的 URL
    $pathInfo = $this->getUrl();

    // 去除 URL 中的查询参数部分，即 ? 及之后的内容
    if (($pos = strpos($pathInfo, '?')) !== false) {
        $pathInfo = substr($pathInfo, 0, $pos);
    }

    // 使用 PHP urldecode() 进行解码，所有 %## 转成对应的字符，+ 转成空格
    $pathInfo = urldecode($pathInfo);

    // 这个正则列举了各种编码方式，通过排除这些编码，来确认是 UTF-8 编码
    // 出处可参考 http://w3.org/International/questions/qa-forms-utf-8.html
    if (!preg_match('%^(?:
        [\x09\x0A\x0D\x20-\x7E]            # ASCII
        | [\xC2-\xDF][\x80-\xBF]           # non-overlong 2-byte
        | \xE0[\xA0-\xBF][\x80-\xBF]       # excluding overlongs
```

```
        | [\xE1-\xEC\xEE\xEF][\x80-\xBF]{2}  # straight 3-byte
        | \xED[\x80-\x9F][\x80-\xBF]         # excluding surrogates
        | \xF0[\x90-\xBF][\x80-\xBF]{2}      # planes 1-3
        | [\xF1-\xF3][\x80-\xBF]{3}          # planes 4-15
        | \xF4[\x80-\x8F][\x80-\xBF]{2}      # plane 16
        )*$%xs', $pathInfo)
    ) {
        $pathInfo = utf8_encode($pathInfo);
    }

    // 获取当前脚本的 URL
    $scriptUrl = $this->getScriptUrl();

    // 获取 Base URL
    $baseUrl = $this->getBaseUrl();

    if (strpos($pathInfo, $scriptUrl) === 0) {
        $pathInfo = substr($pathInfo, strlen($scriptUrl));
    } elseif ($baseUrl === '' || strpos($pathInfo, $baseUrl) === 0) {
        $pathInfo = substr($pathInfo, strlen($baseUrl));
    } elseif (isset($_SERVER['PHP_SELF']) && strpos($_SERVER['PHP_SELF'],
        $scriptUrl) === 0) {
        $pathInfo = substr($_SERVER['PHP_SELF'], strlen($scriptUrl));
    } else {
        throw new InvalidConfigException(
            'Unable to determine the path info of the current request.');
    }

    // 去除 $pathInfo 前的 '/'
    if ($pathInfo[0] === '/') {
        $pathInfo = substr($pathInfo, 1);
    }

    return (string) $pathInfo;
}
```

从 resolvePathInfo() 来看，需要调用到的方法有 getUrl() resolveRequestUri() getScriptUrl() getBaseUrl() 等，这些都是与路径信息密切相关的，让我们分别都看一看。

Request URI

yii\web\Request::getUrl() 用于获取 Request URI 的，实际上这只是一个属性的封装，实质的代码是在 yii\web\Request::resolveRequestUri() 中：

```php
// 这个其实调用的是 resolveRequestUri() 来获取当前 URL
public function getUrl()
{
    if ($this->_url === null) {
        $this->_url = $this->resolveRequestUri();
    }

    return $this->_url;
}

// 这个方法用于获取当前 URL 的 URI 部分，即主机或主机名之后的内容，包括查询参数。
// 这个方法参考了 Zend Framework 1 的部分代码，通过各种环境下的 HTTP 头来获取 URI。
// 返回值为 $_SERVER['REQUEST_URI'] 或 $_SERVER['HTTP_X_REWRITE_URL'],
// 或 $_SERVER['ORIG_PATH_INFO'] + $_SERVER['QUERY_STRING']。
// 即，对于 http://www.digpage.com/index.html?helloworld,
// 得到 URI 为 index.html?helloworld
protected function resolveRequestUri()
{
    // 使用了开启了 ISAPI_Rewrite 的 IIS
    if (isset($_SERVER['HTTP_X_REWRITE_URL'])) {
        $requestUri = $_SERVER['HTTP_X_REWRITE_URL'];

    // 一般情况，需要去掉 URL 中的协议、主机、端口等内容
    } elseif (isset($_SERVER['REQUEST_URI'])) {
        $requestUri = $_SERVER['REQUEST_URI'];

        // 如果 URI 不为空或以'/' 打头，则去除 http:// 或 https:// 直到第一个 /
        if ($requestUri !== '' && $requestUri[0] !== '/') {
            $requestUri = preg_replace('/^(http|https):\/\/[^\/]+/i',
                '', $requestUri);
        }

    // IIS 5.0,  PHP 以 CGI 方式运行，需要把查询参数接上
    } elseif (isset($_SERVER['ORIG_PATH_INFO'])) {
        $requestUri = $_SERVER['ORIG_PATH_INFO'];
        if (!empty($_SERVER['QUERY_STRING'])) {
            $requestUri .= '?' . $_SERVER['QUERY_STRING'];
        }
    } else {
        throw new InvalidConfigException('Unable to determine the request URI.');
    }

    return $requestUri;
}
```

从上面的代码我们可以知道，Yii 针对不同的环境下，PHP 的不同表现形式，通过一些分支判断，给出一个统一的路径名或文件名。辛辛苦苦那么多，其实就是为了消除不同环境对于开发的影响，使开发者可以更加专注于核心工作。

其实，作为一个开发框架，无论是哪种语言、用于哪个领域，都需要为开发者提供在各种环境下的都表现一致的编程界面。这也是开发者可以放心使用的基础条件，如果在使用框架之后，开发者仍需要考虑各种环境下会怎么样怎么样，那么这个框架注定短命。

这里有必要点一点涉及到的几个 $_SERVER 变量。这里面提到的，读者朋友可以自行阅读 PHP 文档关于 $_SERVER 的内容[8]，也可以看看 CGI 1.1 规范的内容[9]。

REQUEST_URI 由 HTTP 1.1 协议定义，指访问某个页面的 URI，去除开头的协议、主机、端口等信息。如 http://www.digpage.com:8080/index.php/foo/bar?queryParams，REQUEST_URI 为 /index.php/foo/bar?queryParams。

X-REWRITE-URL 当使用以开启了 ISAPI_Rewrite 的 IIS 作为服务器时，ISAPI_Rewrite 会在未对原始 URI 作任何修改前，将原始的 REQUEST_URI 以 X-REWRITE-URL HTTP 头保存起来。

PATH_INFO CGI 1.1 规范所定义的环境变量。从形式上看，http://www.digpage.com:8080/index.php</PATH_INFO>?queryParams。它是整个 URI 中，在脚本标识之后、查询参数？之前的部分。对于 Apache，需要设置 AcceptPathInfo On，且在一个 URL 没有 </PATH_INFO> 部分的时候，PATH_INFO 无效。特殊的情况，如 http://www.digpage.com/index.php/，PATH_INFO 为 /。而对于 Nginx，则需要设置：

```
fastcgi_split_path_info ^(.+?\.php)(/.*)$;
fastcgi_param PATH_INFO $fastcgi_path_info;
```

ORIG_PATH_INFO 在 PHP 文档中对它的解释语焉不详，“指未经 PHP 处理过的原始的 PATH_INFO”。这个在 Apache 和 Nginx 需要配置一番才行，但一般用不到，已经有 PATH_INFO 可以用了嘛。而在 IIS 中则有点怪，对于 http://www.digpage.com/index.php/ ORIG_PATH_INFO 为 /index.php/；对于 http://www.digapge.com/index.php ORIG_PATH_INFO 为 /index.php。

根据上面这些背景知识，再来看 resolveRequestUri() 就简单了：

- 最广泛的情况,应当是使用 REQUEST_URI 来获取。但是 resolveRequestUri() 却先使用 X-REWRITE-URL，这是为了防止 REQUEST_URI 被 rewrite。
- 其次才是使用 REQUEST_URI，这对于绝大多数情况是完全够用的了。
- 但 REQUEST_URI 毕竟只是规范的要求，Web 服务器很有可能店大欺客、另立山头，我们又不是第一次碰见了是吧？所以，Yii 使用了平时比较少用到的 ORIG_PATH_INFO。
- 最后，按照规范要求进行规范化，该去头的去头，该续尾的续尾。去除主机信息段和查询参数段后，就大功告成了。

[8] http://php.net/manual/en/reserved.variables.server.php
[9] https://tools.ietf.org/html/rfc3875.html

入口脚本路径

yii\web\Request::getScriptUrl() 用于获取入口脚本的相对路径，也涉及到不同环境下 PHP 的不同表现。我们还是先从代码入手:

```
// 这个方法用于获取当前入口脚本的相对路径
public function getScriptUrl()
{
    if ($this->_scriptUrl === null) {

        // $this->getScriptFile() 用的是 $_SERVER['SCRIPT_FILENAME']
        $scriptFile = $this->getScriptFile();
        $scriptName = basename($scriptFile);

        // 下面的这些判断分支代码，为各主流 PHP framework 所用，
        // Yii, Zend, Symfony 等都是大同小异。
        if (basename($_SERVER['SCRIPT_NAME']) === $scriptName) {
            $this->_scriptUrl = $_SERVER['SCRIPT_NAME'];
        } elseif (basename($_SERVER['PHP_SELF']) === $scriptName) {
            $this->_scriptUrl = $_SERVER['PHP_SELF'];
        } elseif (isset($_SERVER['ORIG_SCRIPT_NAME']) &&
            basename($_SERVER['ORIG_SCRIPT_NAME']) === $scriptName) {
            $this->_scriptUrl = $_SERVER['ORIG_SCRIPT_NAME'];
        } elseif (($pos = strpos($_SERVER['PHP_SELF'], '/' . $scriptName))
            !== false) {
            $this->_scriptUrl = substr($_SERVER['SCRIPT_NAME'], 0, $pos)
                . '/' . $scriptName;
        } elseif (!empty($_SERVER['DOCUMENT_ROOT'])
            && strpos($scriptFile, $_SERVER['DOCUMENT_ROOT']) === 0) {
            $this->_scriptUrl = str_replace('\\', '/',
                str_replace($_SERVER['DOCUMENT_ROOT'], '', $scriptFile));
        } else {
            throw new InvalidConfigException(
                'Unable to determine the entry script URL.');
        }
    }

    return $this->_scriptUrl;
}
```

上面的代码涉及到了一些环境问题，点一点，大家了解下就 OK 了：

SCRIPT_FILENAME 当前脚本的实际物理路径，比如 /var/www/digpage.com/frontend/web/index.php ，或 WIN 平台的 D:\www\digpage.com\frontend\web\index.php 。以 Nginx 为例，一般情况下，SCRIPT_FILENAME 有以下配置项：

```
fastcgi_split_path_info ^(.+?\.php)(/.*)$;
# 使用 document root 来得到物理路径
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name
```

SCRIPT_NAME CGI 1.1 规范所定义的环境变量，用于标识 CGI 脚本（而非脚本的输出），如 http://www.digapge.com/path/index.php 中的 /path/index.php 。仍以 Nginx 为例，SCRIPT_NAME 一般情况下有 fastcgi_param SCRIPT_NAME $fastcgi_script_name 的设置。绝大多数情况下，使用 SCRIPT_NAME 即可获取当前脚本。

PHP_SELF PHP_SELF 是 PHP 自己实现的一个 $_SERVER 变量，是相对于文档根目录（document root）而言的。对于 http://www.digpage.com/path/index.php?queryParams，PHP_SELF 为 /path/index.php 。一般 SCRIPT_NAME 与 PHP_SELF 无异。但是，在 PHP.INI 中，如 cgi.fix_pathinfo=1 （默认即为 1）时，对于形如 http://www.digpage.com/path/index.php/post/view/123 ，则 PHP_SELF 为 /path/index.php/post/view/123 。而根据 CGI 1.1 规范，SCRIPT_NAME 仅为 /path/index.php ，至于剩余的 /post/view/123 则为 PATH_INFO。

ORIG_SCRIPT_NAME 当 PHP 以 CGI 模式运行时，默认会对一些环境变量进行调整。首当其冲的，就是 SCRIPT_NAME 的内容会变成 php.cgi 等二进制文件，而不再是 CGI 脚本文件。当然，设置 cgi.fix_pathinfo=0 可以关闭这一默认行为。但这导致的副作用比较大，影响范围过大，不宜使用。但天无绝人之路，九死之地总留一线生机，那就是 ORIG_SCRIPT_NAME，他保留了调整前 SCRIPT_NAME 的内容。也就是说，在 CGI 模式下，可以使用 ORIG_SCRIPT_NAME 来获取想要的 SCRIPT_NAME。请留意使用 ORIG_SCRIPT_NAME 前一定要先确认它是否存在。

交待完这些背景知识后，我们再来看看 yii\web\Request::getScriptUrl() 的逻辑：

- 先调用 yii\web\Request::getScriptFile() ，通过 basename($_SERVER['SCRIPT_FILENAME']) 获取脚本文件名。一般都是我们的入口脚本 index.php 。
- 绝大多数情况下，base($_SERVER('SCRIPT_NAME')) 是与第一步获取的 index.php 相同的。如果这样的话，则认为这个 SCRIPT_NAME 就是我们所要的脚本 URL。

  这也是规范的定义。但是既然称为规范，说明并非是事实。事实是由 Web 服务器来实现的，也就是说 Web 服务器可能进行修改。

  另外，对于运行于 CGI 模式的 PHP 而言，使用 SCRIPT_NAME 也无法获得脚本名。
- 那么我们转而向 PHP_SELF 求助，这个是 PHP 内部实现的，不受 Web 服务器的影响。一般这个 PHP_SELF 也是可堪一用的，但也不是放之四海而皆准，对于带有 PATH_INFO 的 URI，basename() 获取的并不是脚本名。
- 于是我们再转向 ORIG_SCRIPT_NAME 求助，如果 PHP 是运行于 CGI 模式，那么就可行。
- 再不成功，可能 PHP 并非运行于 CGI 模式（否则第 4 步就可以成功），且 URI 中带有 PATH_INFO（否则第二步就可以成功）。对于这种情形，PHP_SELF 的前面一截就是我们要的脚本 URL 。
- 万一以上情况都不符合,说明当前 PHP 运行的环境诡异莫测。那只能寄希望于将 SCRIPT_FILENAME 中前面那截可能是 Document Root 的部分去掉，余下的作为脚本 URL 了。前提是要有 Document

Root，且 SCRIPT_FILENAME 前面的部分可以匹配上。

Base Url

获取路径信息的最后一个相关方法，就是 yii\web\Request::getBaseUrl():

```
// 获取 Base Url
public function getBaseUrl()
{
    if ($this->_baseUrl === null) {
        // 用上面的脚本路径的父目录，再去除末尾的 \ 和 /
        $this->_baseUrl = rtrim(dirname($this->getScriptUrl()), '\\/');
    }

    return $this->_baseUrl;
}
```

这个 Base Url 很简单，相信聪明如你肯定一目了然，我就不浪费篇幅了。

好了，上面就是 yii\web\Request::resolve() 中有关获取路径信息的内容。下一步就是使用路由规则去解析当前请求了。

使用路由规则解析

上面这么多有关从请求获取路径信息的内容，其实只完成了请求解析的第一步而已。接下来，urlManager 就要遍历所有的路由规则来解析当前请求，直到有一个规则适用为止。

路由规则层面对于请求的解析，我们在路由 (Route) 的解析 URL 部分已经讲得很清楚了。

CHAPTER 6

Yii 与数据库（TBD）

6.1 数据类型

各 DBMS 间，最明显、最常见的差异就在于所支持、实现的数据类型不同。Yii 的一个重要任务，就是消除这些区别，提供一个统一的开发界面供开发者使用。所以，我们先来看看 Yii 是怎么克服这一拦路虎，实现天下的大一统的。

6.1.1 抽象数据类型

在使用 Yii 进行数据库开发时，涉及到 2 个方面的数据类型：PHP 自身的数据类型，DBMS 的数据类型。其中，PHP 数据类型比较好整，反正就那么几个，跟平台、环境的关系也比较清楚。复杂的地方在于 DBMS 的数据类型，龙生九子，各有不同。

因此，Yii 不但需要搞定各 DBMS 间数据类型的差异，还需要搞定 PHP 与 DBMS 间数据类类型的差异。因此，Yii 引入了一个逻辑层面的数据类型，来统一 PHP 与 DBMS，以及各 DBMS 之间数据类型的差异。这里我们把这个逻辑层面的数据类型称为抽象类型，在 yii\db\Schema 中进行定义:

```
1  abstract class Schema extends Object
2  {
3      // 预定义 16 种抽象字段类型
4      const TYPE_PK = 'pk';
5      const TYPE_BIGPK = 'bigpk';
6      const TYPE_STRING = 'string';
7      const TYPE_TEXT = 'text';
8      const TYPE_SMALLINT = 'smallint';
9      const TYPE_INTEGER = 'integer';
10     const TYPE_BIGINT = 'bigint';
11     const TYPE_FLOAT = 'float';
12     const TYPE_DECIMAL = 'decimal';
13     const TYPE_DATETIME = 'datetime';
14     const TYPE_TIMESTAMP = 'timestamp';
```

```
    const TYPE_TIME = 'time';
    const TYPE_DATE = 'date';
    const TYPE_BINARY = 'binary';
    const TYPE_BOOLEAN = 'boolean';
    const TYPE_MONEY = 'money';

    // ... ...
}
```

yii\db\Schema 一上来就先针对各 DBMS 间差异最明显的字段数据类型进行统一，提供了 16 种抽象的字段类型。这 16 种类型与 DBMS 无关，在具体到特定的 DBMS 时，Yii 会自动转换成合适的数据库字段类型。我们在编程中，若需要指定字段类型，比如创建数据库之类，就使用这 16 种抽象类型。这样的话，就不用考虑使用的类型具体的 DBMS 是否支持的问题了，可以使我们更加专注于开发。

这 16 种类型看着就知道是什么意思，我们就不展开讲了。只是有一点，这个数据类型是抽象的，也就是说，只是一个逻辑意义上的数据类型，不涉及到具体实现。比如上面的 Schema::TYPE_BIGINT 所要表示的是一个大数，但具体实现上，可能是用一个字符串来表示。

6.1.2 数据类型转换

既然 Yii 使用抽象数据类型来一统江山，那么无可避免的，涉及到 PHP 数据类型和 DBMS 数据类型与抽象类型的转换问题。

抽象类型转数据库类型

前面提到过，在 Yii 开发中，我们使用 16 种抽象数据类型，而不使用具体的 DBMS 的数据类型。那么，在我们要创建一个数据库时，Yii 是怎么为我们所定义的字段指定合适的数据类型的呢?

首先来看看一个基类 yii\db\QueryBuilder

```
class QueryBuilder extends \yii\base\Object
{
    // $typeMap 用于定义抽象数据类型到 DBMS 数据类型的映射关系，
    // 具体由各 QueryBuilder 子类实现。
    public $typeMap = [];

    // 将抽象数据类型转换成合适的 DBMS 数据类型
    public function getColumnType($type)
    {
        // 映射表中已经有的，直接使用映射的类型
        if (isset($this->typeMap[$type])) {
            return $this->typeMap[$type];

        // 映射表中没有的类型，看看是不是形如 "Schema::TYPE_INT(11) DEFAULT 0" 之类的
```

```
        } elseif (preg_match('/^(\w+)\((.+?)\)(.*)$/', $type, $matches)) {
            if (isset($this->typeMap[$matches[1]])) {
                return preg_replace('/\(.+\)/', '(' . $matches[2] . ')', $this->typeMap[$matches[1]]) . $matches[3];
            }

        // 看看是不是形如 "Schema::TYPE_INT NOT NULL" 之类的,
        // 注意这一分支在第二分支之后
        } elseif (preg_match('/^(\w+)\s+/', $type, $matches)) {
            if (isset($this->typeMap[$matches[1]])) {
                return preg_replace('/^\w+/', $this->typeMap[$matches[1]], $type);
            }
        }

        // 实在匹配不上映射表中的类型，那就原封不动返回吧
        return $type;
    }

    // ... ...
}
```

上面的代码只列出了 yii\db\QueryBuilder 的部分内容。特别是其中的 $typeMap[] 是由子类来具体实现的。比如，对于 MySQL 数据库，yii\db\mysql\QueryBuilder 中：

```
namespace yii\db\mysql;

class QueryBuilder extends \yii\db\QueryBuilder
{
    public $typeMap = [
        Schema::TYPE_PK => 'int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY',
        Schema::TYPE_BIGPK => 'bigint(20) NOT NULL AUTO_INCREMENT PRIMARY KEY',
        Schema::TYPE_STRING => 'varchar(255)',
        Schema::TYPE_TEXT => 'text',
        Schema::TYPE_SMALLINT => 'smallint(6)',
        Schema::TYPE_INTEGER => 'int(11)',
        Schema::TYPE_BIGINT => 'bigint(20)',
        Schema::TYPE_FLOAT => 'float',
        Schema::TYPE_DECIMAL => 'decimal(10,0)',
        Schema::TYPE_DATETIME => 'datetime',
        Schema::TYPE_TIMESTAMP => 'timestamp',
        Schema::TYPE_TIME => 'time',
        Schema::TYPE_DATE => 'date',
        Schema::TYPE_BINARY => 'blob',
        Schema::TYPE_BOOLEAN => 'tinyint(1)',
        Schema::TYPE_MONEY => 'decimal(19,4)',
    ];
```

```
    // ... ...
}
```

因此，对于 16 种抽象数据类型而言，都可以转换成 MySQL 的特定数据类型。这里我们看到，TYPE_MONEY 本来是 MySQL 所没有的数据类型，但通过将其映射成了 decimal(19,4)，使得 MySQL 也可以支持 TYPE_MONEY 类型了。

同样的，在 yii\db\pgsql\QueryBuilder yii\db\mssql\QueryBuilder 等 QueryBuilder 子类中，也有相同的数据类型映射表 $typeMap[]，以实现从抽象数据类型到具体数据库数据类型的映射。

对于抽象数据类型的具体定义，我们选一个自己比较熟悉的 DBMS 的 Schema 进行记忆就可以了。如果实在吃不准，比如 TYPE_STRING 是定长还是变长，保留不保留空格等，可以看看 Schema 中的定义。建议读者朋友们还是熟记这 16 个基本类型，起码仔细过一遍，在具体开发时，能省时不少。

yii\db\QueryBuilder::getColumnType() 实现了抽象数据类型到具体 DBMS 数据类型的转换。在具体的转换过程中，如果指定一个字段为 Schema::TYPE_STRING 之类的，那么就会被转换成 varchar(255)。有的读者朋友可能会问，那要是想指定成 varchar(64) 该怎么做呢?

那就直接使用 Schema::TYPE_STRING(64)。yii\db\QueryBuilder::getColumnType() 会识别出来 Schema::TYPE_STRING，并将原来的 varchar(255) 转换成 varchar(64)。这点适用于其他数据类型。当然，其本身要支持。指定一个 Schema::TYPE_BOOLEAN(2) 又有什么意义呢?

我们还可以在 16 种抽象类型后面使用 NOT NULL，DEFAULT 等。yii\db\QueryBuilder::getColumnType() 也是能够识别出来并加以处理的。

数据库类型转抽象类型

所谓来而不往非礼也，说完了抽象类型转 DBMS 数据类型，就该说说反过来数据库的数据类型，怎么转换成抽象数据类型了。仍然以 MySQL 数据库为例，具体代码在 yii\db\mysql\Schema 中:

```
class Schema extends \yii\db\Schema
{
    // 定义从数据库数据类型到 16 个抽象数据类型间的映射关系
    public $typeMap = [
        'tinyint' => self::TYPE_SMALLINT,
        'bit' => self::TYPE_INTEGER,
        'smallint' => self::TYPE_SMALLINT,
        'mediumint' => self::TYPE_INTEGER,
        'int' => self::TYPE_INTEGER,
        'integer' => self::TYPE_INTEGER,
        'bigint' => self::TYPE_BIGINT,
        'float' => self::TYPE_FLOAT,
        'double' => self::TYPE_FLOAT,
        'real' => self::TYPE_FLOAT,
        'decimal' => self::TYPE_DECIMAL,
        'numeric' => self::TYPE_DECIMAL,
```

```
        'tinytext' => self::TYPE_TEXT,
        'mediumtext' => self::TYPE_TEXT,
        'longtext' => self::TYPE_TEXT,
        'longblob' => self::TYPE_BINARY,
        'blob' => self::TYPE_BINARY,
        'text' => self::TYPE_TEXT,
        'varchar' => self::TYPE_STRING,
        'string' => self::TYPE_STRING,
        'char' => self::TYPE_STRING,
        'datetime' => self::TYPE_DATETIME,
        'year' => self::TYPE_DATE,
        'date' => self::TYPE_DATE,
        'time' => self::TYPE_TIME,
        'timestamp' => self::TYPE_TIMESTAMP,
        'enum' => self::TYPE_STRING,
    ];

    // ... ...
}
```

在 yii\db\mysql\Schema::$typeMap 中，定义了 MySQL 数据类型与 16 种抽象数据类型的映射关系。当然，由于抽象数据类型只有 16 种，所以，有一些 MySQL 数据类型被映射成同一种抽象类型。比如 year date 都被映射成了 self::TYPE_DATE 了。

同样的，你可以在 yii\db\mssql\Schema 和 yii\db\pgsql\Schema 中找到类似的映射代码。在各 Schema 子类中，Yii 针对不同的 DBMS，实现了数据库数据类型到抽象数据类型间的映射关系。

从数据库类型到抽象类型的映射关系，只要掌握前面 16 种抽象类型与数据库数据类型的映射关系，那么，其逆向映射就几乎不用刻意去记忆了。额外的，凡是没有合适的抽象类型的，就用 TYPE_STRING 来表示。

上面只是一个映射表，具体转换过程在生成字段信息 yii\db\ColumnSchema 时进行。yii\db\ColumnSchema 保存了一个字段的各种相关信息，包括字段类型等。

字段信息的获取和填充，又发生在 yii\db\Schema::loadTableSchema() 中。该函数用于加载数据表信息 yii\db\TableSchema，这是一个抽象函数，具体由各子类实现。

字段信息和表信息我们后面再讲，这里大致知道就是用于保存字段和数据表的各种信息就可以了。而且在获取表信息时，必然会调用到获取字段信息的相关代码。字段信息和表信息的获取，都由 Schema 子类来具体实现。

对于 MySQL 数据库而言，字段信息的获取在 yii\db\mysql\Schema::loadColumnSchema() 中，也是他实现了从数据库类型到抽象类型的转换:

```
// $info 数组 由 SQL 语句 "SHOW FULL COLUMNS FROM ..." 而来，形式如下：
//      Field: id
//       Type: int(11)
```

```
//  Collation: NULL
//       Null: NO
//        Key: PRI
//    Default: NULL
//      Extra: auto_increment
// Privileges: select,insert,update,references
//    Comment:
protected function loadColumnSchema($info)
{
    $column = $this->createColumnSchema();

    // 字段名
    $column->name = $info['Field'];
    // 是否允许为 NULL
    $column->allowNull = $info['Null'] === 'YES';
    // 是否是主键
    $column->isPrimaryKey = strpos($info['Key'], 'PRI') !== false;
    // 是否 auto_increment
    $column->autoIncrement = stripos($info['Extra'], 'auto_increment') !== false;
    // 获取字段注释
    $column->comment = $info['Comment'];

    // 重点是这里，获取数据库字段类型，如上面的 int(11)
    $column->dbType = $info['Type'];
    // 是否是 unsigned
    $column->unsigned = stripos($column->dbType, 'unsigned') !== false;

    // 以下将把数据库类型，转换成对应的抽象类型，默认为 TYPE_STRING
    $column->type = self::TYPE_STRING;
    if (preg_match('/^(\w+)(?:\(([^\)]+)\))?/', $column->dbType, $matches)) {

        // 获取 int(11) 的 "int" 部分
        $type = strtolower($matches[1]);
        // 如果映射表里有，那就直接映射成抽象类型
        if (isset($this->typeMap[$type])) {
            $column->type = $this->typeMap[$type];
        }

        // 形如 int(11) 的括号中的内容
        if (!empty($matches[2])) {
            // 枚举类型，还需要将所有枚举值写入 $column->enumValues
            if ($type === 'enum') {
                $values = explode(',', $matches[2]);
                foreach ($values as $i => $value) {
                    $values[$i] = trim($value, "'");
```

```
            }
            $column->enumValues = $values;

        // 如果不是枚举类型，那么括号中的内容就是精度了，如 decimal(19,4)
        } else {
            $values = explode(',', $matches[2]);
            $column->size = $column->precision = (int) $values[0];
            if (isset($values[1])) {
                $column->scale = (int) $values[1];
            }

            // bit(1) 类型的，转换成 boolean
            if ($column->size === 1 && $type === 'bit') {
                $column->type = 'boolean';
            } elseif ($type === 'bit') {
                // 由于 bit 最多 64 位，如果超过 32 位，那么用一个 bigint 足以。
                if ($column->size > 32) {
                    $column->type = 'bigint';
                // 如果正好 32 位，那么用一个 interger 来表示。
                } elseif ($column->size === 32) {
                    $column->type = 'integer';
                }
            }
        }
    }

    // 获取 PHP 数据类型
    $column->phpType = $this->getColumnPhpType($column);

    // 处理默认值
    if (!$column->isPrimaryKey) {
        // timestamp 的话，要实际获取当前时间戳，而不能是字符串 'CURRENT_TIMESTAMP'
        if ($column->type === 'timestamp' && $info['Default'] === 'CURRENT_TIMESTAMP') {
            $column->defaultValue = new Expression('CURRENT_TIMESTAMP');

        // bit 的话，要截取对应的内容，并进行进制转换
        } elseif (isset($type) && $type === 'bit') {
            $column->defaultValue = bindec(trim($info['Default'],'b\''));

        // 其余类型的，直接转换成 PHP 类型的值
        } else {
            $column->defaultValue = $column->phpTypecast($info['Default']);
        }
    }
```

```
94      return $column;
95  }
```

上面的代码是完整获取字段信息的过程，这里重点要看的，是获取数据类型部分的代码。结合映射表和上面的代码，我们能够得出：

- 通过 SHOW FULL COLUMNS FROM SQL 语句获取字段信息，并存储在 $info 数组中。
- 根据 $info['Type'] 获取字段类型信息。
- 如果映射表里已经有映射关系的，直接通过映射表，获取相应的抽象类型。
- 如果映射表没有的，默认地视字段的抽象类型为 TYPE_STRING 。
- 对于枚举类型，除了转换成 TYPE_STRING 外，还要获取其枚举值，否则，类型信息不完整。
- 对于 bit 类型，在 32 位及 32 位以下时，使用 TYPE_INTEGER 抽象类型，在 32 位以上（bit 最大为 64 位）时，使用 TYPE_BIGINT 类型。

至于其他字段信息的获取，如是否允许为 NULL 之类，上面的代码中注释已经交待清楚了，大家这么聪明，相信难不倒你们。需要稍稍注意的，就是默认值的处理，特别是 timestamp 类型默认值的处理。

抽象类型转 PHP 类型

前面讲数据库类型转抽象类型时，在 yii\db\mysql\Schema::loadColumnSchema() 中，有一个语句 $column->phpType = $this->getColumnPhpType($column); 我们只是简单地说是转换成 PHP 类型，就一笔带来了。其实，他就是把抽像类型转换成 PHP 类型的关键，让我们来看看这个 yii\db\Schema::getColumnPhpType()

```
1   protected function getColumnPhpType($column)
2   {
3       // 定义从抽象类型到 PHP 类型的映射
4       static $typeMap = [
5           'smallint' => 'integer',
6           'integer' => 'integer',
7           'bigint' => 'integer',
8           'boolean' => 'boolean',
9           'float' => 'double',
10          'binary' => 'resource',
11      ];
12
13      // 除了上面的映射关系外，还有几个特殊情况：
14      // 1. bigint 字段，在 64 位环境下，且为 singed 时，使用 integer 来表示，否则 string
15      // 2. integer 字段，在 32 位环境下，且为 unsinged 时，使用 string 表示，否则 integer
16      // 3. 映射中不存在的字段类型均使用 string
17      if (isset($typeMap[$column->type])) {
18          if ($column->type === 'bigint') {
```

```
19              return PHP_INT_SIZE == 8 && !$column->unsigned ? 'integer' : 'string';
20          } elseif ($column->type === 'integer') {
21              return PHP_INT_SIZE == 4 && $column->unsigned ? 'string' : 'integer';
22          } else {
23              return $typeMap[$column->type];
24          }
25      } else {
26          return 'string';
27      }
28  }
```

首先不要惊讶于为什么这个方法不像数据库类与抽象类型转换时，放在 MySQL 子类 Schema 中实现。这是由于我们现在讨论的是抽象类型到 PHP 类型的转换。我们说过，抽象类型是与数据库无关的，PHP 类型更是与数据库没有半毛钱关系，所以，放在基类 yii\db\Schema 中是合理的。这也是我们在平时在编程时经常采用的划分基类与子类的一个常用准则。

虽然，我们说抽象数据类型有 16 种，但是到了 PHP 范畴，有的类型是没有意义的。在 PHP 中，数据库的字段类型，都可以归结了 integer boolean double resourece string 5 种。

- TYPE_PK 和 TYPE_BIGPK 表示主键，Yii 使用 yii\db\ColumnSchema::isPrimaryKey 属性来表示，不存在将其转换成何种数据类型的问题。
- TYPE_SMALLINT TYPE_INTEGER TYPE_BIGINT 等字段一般都转换成 PHP 的 integer 类型。特别是 TYPE_SMALLINT，PHP 的 integer 完全可以满足要求。
- TYPE_BIGINT 字段，如果是在 64 位环境下，且该字段并非为 unsigned 时，PHP 的 integer 足够存储。但是，在 32 位环境下，PHP 的 integer 只有 4 个字节，不够存储 8 个字节的 TYPE_BIGINT。而如果字段是 unsigned 的，由于 PHP 并没有 unsigned 一说，就算是 64 位环境，也少了 1 bit。在不够存储时，就只能选用 PHP 的 string 类型了。
- TYPE_INTEGER 字段，在 32 位环境下，如果是 unsigned 的，那也会少 1 bit 来存储。而如果是 64 位环境，或者并非是 unsigned 的，PHP 的 integer 都是够用的。同样，不够存储时，就使用字符串类型了。
- TYPE_FLOAT 字段，注意是转换成 PHP 的 double 而非 float，float 精度不够。
- TYPE_BOOLEAN 字段，顺理成章对应 PHP 的 boolean 类型。
- TYPE_BINARY 字段，理所当然对应 PHP 的 resource 类型。
- TYPE_STRING 和 TYPE_TEXT 字段，显而易见对应 PHP 的 string 类型。
- TYPE_DECIMAL 和 TYPE_MONEY 字段由于 PHP 的数值类型精度都不够，所以，只能使用 string 类型来表示。
- 对于日期、时间等字段，尽管 PHP 提供了丰富的日期、时间函数，但事实上，PHP 中并没有专门的表示日期、时间的数据类型。因此，对于这类字段，又只能求助于万能的 string 类型了。

字段内容转 PHP 变量

前面我们有了数据库类型转抽象类型，抽象类型转 PHP 类型，那么，我们就可以完成数据库类型到 PHP 类型的转换，也就是说，能够将数据库中的内容读取到 PHP 中供我们编程时使用啦。

假设有如下的 MySQL 数据表：

```
CREATE TABLE `tbl_news` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` int(10) unsigned NOT NULL DEFAULT '0',
  `status` enum('Draft','Publish','Archive','Unpublish') NOT NULL DEFAULT 'Draft',
  `title` varchar(256) NOT NULL DEFAULT '',
  `tags` varchar(256) NOT NULL DEFAULT '',
  `createtime` int(10) unsigned NOT NULL DEFAULT '0',
  `publishtime` int(10) unsigned NOT NULL DEFAULT '0',
  `abstract` varchar(512) NOT NULL DEFAULT '',
  `content` text,
  `lastupdate` int(10) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=8 DEFAULT CHARSET=utf8
```

那么，通过 PDO 获取其中一个记录，DUMP 出来后，各字段的类型为：

```
array(10) {
  ["id"]=>
  string(1) "7"
  ["userid"]=>
  string(1) "1"
  ["status"]=>
  string(5) "Draft"
  ["title"]=>
  string(6) "teste2"
  ["tags"]=>
  string(4) "test"
  ["createtime"]=>
  string(10) "1382152003"
  ["publishtime"]=>
  string(10) "1381161600"
  ["abstract"]=>
  string(3) "abs"
  ["content"]=>
  string(12) "<p>cont</p>"
  ["lastupdate"]=>
  string(10) "1382152003"
}
```

你能看到，所有的字段通过 PDO 读取后，再 DUMP 出来都是以 string 保存的。这就需要 Yii

根据 ColumnSchema 中字段信息，去把各字段以合适的 PHP 类型表现出来。而这个过程，就是 yii\db\ColumnSchema::phpTypeCast()

```php
// 该方法用于把 $value 转换成 php 变量，其中 $value 是 PDO 从数据库中读取的内容
// 主要参考的是字段的抽象类型 $type 和 PHP 类型 $phpType
public function phpTypecast($value)
{
    // 内容为空时，若不是字符串或二进制抽象类型，则是 NULL 的意思。
    if ($value === '' && $this->type !== Schema::TYPE_TEXT && $this->type !== Schema::TYPE_STRING && $this->type !=
        return null;
    }

    // 内容为 null，或者 $value 的类型与 PHP 类型一致，或者 $value 是一个数据库表达式，
    // 那么可以直接返回
    if ($value === null || gettype($value) === $this->phpType || $value instanceof Expression) {
        return $value;
    }

    // 否则，需要根据 PHP 类型来完成类型转换
    switch ($this->phpType) {
        case 'resource':
        case 'string':
            return is_resource($value) ? $value : (string) $value;
        case 'integer':
            return (int) $value;
        case 'boolean':
            return (bool) $value;
        case 'double':
            return (double) $value;
    }
    return $value;
}
```

在上面的转换过程中，PHP 类型信息已经在上一步的 yii\db\Schema::getColumnPhpType() 中获取了。这里就是把从数据库中得到的数据，转换成 phpType 所指定的 PHP 变量类型。

前面我们讲过，16 种抽象类型最终对应于 PHP 的 5 种数据类型。但是，从上面的 phpTypeCast() 来看，还要多出 2 种类型来：

- 一是 null，对于既不是文本类型，又不是二进制类型的字段，如果其内容为空，那么他的意思其实是 NULL。
- 二是 yii\db\Expression 数据库表达式类型。如前面我们提到的 timestamp 的默认值是 CURRENT_TIMESTAMP 时，就用到了一个 new Expression('CURRENT_TIMESTAMP')。

PHP **类型转** PDO **类型**

前面，我们已经实现了创建数据库时，使用抽象类型来指定字段类型，也实现了从数据库读取数据到 PHP（数据库类型—抽象类型— PHP 类型）的转换过程。

还有一个情景我们没有涉及到，那就是 PHP 类型转数据库类型的问题。由于 Yii 通过 PDO 操作数据库，因此，这个问题就成为了 PHP 类型怎么转换成 PDO 数据类型的问题，至于 PDO 转数据库字段类型，由 PDO 自己实现，我们就不用操心了。

这里，比如说的 SQL 查询参数绑定，PDO::bindParam() 要求提供参数类型。Yii 是能过 PHP 类型，直接就转换成 PDO 的类型了，具体的代码在 yii\db\Schema::getPdoType()

```
// 将一个 PHP 数据类型转换成 PDO 数据类型
public function getPdoType($data)
{
    // 定义一个 PHP 类型到 PDO 类型的映射表
    static $typeMap = [
        'boolean' => \PDO::PARAM_BOOL,
        'integer' => \PDO::PARAM_INT,
        'string' => \PDO::PARAM_STR,
        'resource' => \PDO::PARAM_LOB,
        'NULL' => \PDO::PARAM_NULL,
    ];
    $type = gettype($data);

    // 在匹配不上映射表时，采用字符串类型
    return isset($typeMap[$type]) ? $typeMap[$type] : \PDO::PARAM_STR;
}
```

前面我们一直在说，16 种抽象数据类型，可以用来表示各种 DBMS 的各种字段类型。而这 16 种抽象类型，具体到 PHP 数据类型时，逃不过 integer boolean resource string 和 double 5 种，再加上特殊的 null 一共是 6 种。

而在上面的 getPdoType() 中，与 6 种 PHP 数据类型对应的，PDO 提供了 5 种，其中 double 类型用 PDO::PARAM_STR 来表示。

至此，PHP、Yii、DBMS、PDO 间的数据类型转换问题就完全解决了。

6.2 事务（Transaction）

在 Yii 中，使用 yii\db\Transaction 来表示数据库事务。

一般情况下，我们从数据库连接启用事务，通常采用如下的形式:

```
$transaction = $connection->beginTransaction();
try {
```

```
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    // ... executing other SQL statements ...
    $transaction->commit();
} catch (Exception $e) {
    $transaction->rollBack();
}
```

在上面的代码中，先是获取一个 yii\db\Transaction 对象，之后执行若干 SQL 语句，然后调用之前 Transaction 对象的 commit() 方法。这一过程中，如果捕获了异常，那么调用 rollBack() 进行回滚。

6.2.1 创建事务

在上面代码中，我们使用数据库连接的 beginTransaction() 方法，创建了一个 yii\db\Trnasaction 对象，具体代码在 yii\db\Connection 中:

```
public function beginTransaction($isolationLevel = null)
{
    $this->open();
    // 尚未初始化当前连接使用的 Transaction 对象，则创建一个
    if (($transaction = $this->getTransaction()) === null) {
        $transaction = $this->_transaction = new Transaction(['db' => $this]);
    }

    // 获取 Transaction 后，就可以启用事务
    $transaction->begin($isolationLevel);
    return $transaction;
}
```

从创建 Transaction 对象的 new Transaction(['db' => $this]) 形式来看，这也是 Yii 一贯的风格。这里简单的初始化了 yii\db\Transaction::db 。

这表示的是当前的 Transaction 所依赖的数据库连接。如果未对其进行初始化，那么将无法正常使用事务。

在获取了 Transaction 之后，就可以调用他的 begin() 方法，来启用事务。必要的情况下，还可以指定事务隔离级别。

事务隔离级别的设定，由 yii\db\Schema::setTransactionIsolationLevel() 方法来实现，而这个方法，无非就是执行了如下的 SQL 语句：

```
SET TRANSACTION ISOLATION LEVEL ...
```

对于隔离级别，yii\db\Transaction 也提前定义了几个常量:

```
const READ_UNCOMMITTED = 'READ UNCOMMITTED';
const READ_COMMITTED = 'READ COMMITTED';
const REPEATABLE_READ = 'REPEATABLE READ';
const SERIALIZABLE = 'SERIALIZABLE';
```

如果开发者没有给出隔离级别，那么，数据库会使用默认配置的隔离级别。比如，对于 MySQL 而言，就是使用 transaction-isolation 配置项的值。

6.2.2 启用事务

上面的代码告诉我们，启用事务，最终是靠调用 Transaction::begin() 来实现的。那么就让我们来看看他的代码吧:

```
1  public function begin($isolationLevel = null)
2  {
3      // 没有初始化数据库连接的滚粗
4      if ($this->db === null) {
5          throw new InvalidConfigException('Transaction::db must be set.');
6      }
7      $this->db->open();
8
9      // _level 为 0 表示的是最外层的事务
10     if ($this->_level == 0) {
11         // 如果给定了隔离级别，那么就设定之
12         if ($isolationLevel !== null) {
13             // 设定事务隔离级别
14             $this->db->getSchema()->setTransactionIsolationLevel($isolationLevel);
15         }
16         Yii::trace('Begin transaction' . ($isolationLevel ? ' with isolation level ' . $isolationLevel : ''), __METHOD__);
17         $this->db->trigger(Connection::EVENT_BEGIN_TRANSACTION);
18         $this->db->pdo->beginTransaction();
19         $this->_level = 1;
20         return;
21     }
22
23     // 以下 _level>0 表示的是嵌套的事务
24     $schema = $this->db->getSchema();
25
26     // 要使用嵌套事务，前提是所使用的数据库要支持
27     if ($schema->supportsSavepoint()) {
28         Yii::trace('Set savepoint ' . $this->_level, __METHOD__);
29         // 使用事务保存点
30         $schema->createSavepoint('LEVEL' . $this->_level);
31     } else {
32         Yii::info('Transaction not started: nested transaction not supported', __METHOD__);
```

```
    }
    // 结合 _level == 0 分支中的 $this->_level = 1,
    // 可以得知，一旦调用这个方法， _level 就会自增 1
    $this->_level++;
}
```

对于最外层的事务，即当 _level 为 0 时，最终落到 PDO 的 beginTransaction() 来启用事务。在启用前，如果开发者给定了隔离级别，那么还需要设定隔离级别。

当 _level > 0 时，表示的是嵌套的事务，并非最外层的事务。对此，Yii 使用 SQL 的 SAVEPOINT 和 ROLLBACK TO SAVEPOINT 来实现设置事务保存点和回滚到保存点的操作。

6.2.3 嵌套事务

在开头的例子中，展现的是事务最简单的使用形式。Yii 还允许把事务嵌套起来使用。比如，可以采用如下形式来使用事务:

```
$outerTransaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();

    $innerTransaction = $db->beginTransaction();
    try {
        $db->createCommand($sql2)->execute();
        $db->createCommand($sql3)->execute();
        $innerTransaction->commit();
    } catch (Exception $e) {
        $innerTransaction->rollBack();
    }

    $db->createCommand($sql4)->execute();
    $outerTransaction->commit();
} catch (Exception $e) {
    $outerTransaction->rollBack();
}
```

为了实现这一嵌套，Yii 使用 yii\db\Transaction::_level 来表示嵌套的层级。当层级为 0 时，表示的是最外层的事务。

一般情况下，整个 Yii 应用使用了同一个数据库连接，或者说是使用了单例。具体可以看服务定位器（Service Locator） 部分。

而在 yii\db\Connection 中，又对事务对象进行了缓存:

```
class Connection extends Component
{
```

```
    // 保存当前连接的有效 Transaction 对象
    private $_transaction;

    // 已经缓存有事务对象，且事务对象有效，则返回该事务对象
    // 否则返回 null
    public function getTransaction()
    {
        return $this->_transaction && $this->_transaction->getIsActive() ? $this->_transaction : null;
    }

    // 看看启用事务时，是如何使用事务对象的
    public function beginTransaction($isolationLevel = null)
    {
        $this->open();
        // 缓存的事务对象有效，则使用缓存中的事务对象
        // 否则创建一个新的事务对象
        if (($transaction = $this->getTransaction()) === null) {
            $transaction = $this->_transaction = new Transaction(['db' => $this]);
        }
        $transaction->begin($isolationLevel);
        return $transaction;
    }
}
```

因此，可以认为整个 Yii 应用，使用了同一个 Transaction 对象，也就是说，Transaction::_level 在整个应用的生命周期中，是有延续性的。这是实现事务嵌套的关键和前提。

在这个 Transaction::_level 的基础上，Yii 实现了事务的嵌套：

- 事务对象初始化时，设 _level 为 0，表示如果要启用事务，这是一个最外层的事务。
- 每当调用 Transaction::begin() 来启用具体事务时，_level 自增 1。表示如再启用事务，将是层级为 1 的嵌套事务。
- 每当调用 Transaction::commit() 或 Transaction::rollBack() 时，_level 自减 1，表示当前层级的事务处理完毕，返回上一层级的事务中。
- 当调用了一次 begin() 且还没有调用匹配的 commit() 或 rollBack()，就再次调用 begin() 时，会使事务进行更深一层级的嵌套中。

因此，就有了我们上面代码中，当 _level 为 0 时，需要设定事务隔离级别。因为这是最外层事务。

而当 _level > 0 时，由于是“嵌套”的事务，一个大事务中的小“事务”，那么，就使用保存点及其回滚、释放操作，来模拟事务的启用、回滚和提交操作。

要注意，在这一节的开头，我们使用 2 对嵌套的 try ... catch 来实现事务的嵌套。由于内层的 catch 把可能抛出的异常吞了，不再继续抛出。那么，外层的 catch，是捕获不到内层的异常的。

也就是说，这种情况下，外层中的 $sql1 $sql4 不会由于 $sql2 或 $sql3 的失败而中止，$sql1 $sql4 可以继续执行并 commit 。

这是嵌套事务的正确使用形式，即内外层之间应当是不相干的。

如果内层事务的异常，会导致外层事务需要回滚，那么我们不应该使用事务嵌套，而是应该把内外层当成一个事务。这个道理很浅显，但是事实开发中，一个不小心，就会出昏招。所以，不要动不动就来个 beginTransaction() 。

当然，为了使代码功能有一定的层次感，在必要时，也可以使用嵌套的事务。但要考虑好，子事务是否真的要吞掉异常？有没有必要继续抛出异常，使得上一层级的事务也产生回滚？这个要根据实际的情形来确定。

6.2.4 提交和回滚

提交和回滚通过 Transaction::commit() 和 Transaction::rollBack() 来实现：

```
public function commit()
{
    if (!$this->getIsActive()) {
        throw new Exception('Failed to commit transaction: transaction was inactive.');
    }
    // 与 begin() 对应，只要调用 commit()，_level 自减 1
    $this->_level--;
    // 如果回到了最外层事务，那么应当使用 PDO 的 commit
    if ($this->_level == 0) {
        Yii::trace('Commit transaction', __METHOD__);
        $this->db->pdo->commit();
        $this->db->trigger(Connection::EVENT_COMMIT_TRANSACTION);
        return;
    }
    // 以下是尚未回到最外层的情形
    $schema = $this->db->getSchema();
    if ($schema->supportsSavepoint()) {
        Yii::trace('Release savepoint ' . $this->_level, __METHOD__);
        // 释放那么保存点
        $schema->releaseSavepoint('LEVEL' . $this->_level);
    } else {
        Yii::info('Transaction not committed: nested transaction not supported', __METHOD__);
    }
}

public function rollBack()
{
    if (!$this->getIsActive()) {
        return;
    }
```

```
31      // 调用 rollBack() 也会使 _level 自减 1
32      $this->_level--;
33
34      // 如果已经返回到最外层，那么调用 PDO 的 rollBack
35      if ($this->_level == 0) {
36          Yii::trace('Roll back transaction', __METHOD__);
37          $this->db->pdo->rollBack();
38          $this->db->trigger(Connection::EVENT_ROLLBACK_TRANSACTION);
39          return;
40      }
41      // 以下是未返回到最外层的情形
42      $schema = $this->db->getSchema();
43      if ($schema->supportsSavepoint()) {
44          Yii::trace('Roll back to savepoint ' . $this->_level, __METHOD__);
45          // 那么就回滚到保存点
46          $schema->rollBackSavepoint('LEVEL' . $this->_level);
47      } else {
48          Yii::info('Transaction not rolled back: nested transaction not supported', __METHOD__);
49          throw new Exception('Roll back failed: nested transaction not supported.');
50      }
51  }
```

对于提交和回滚：

- 提交时，会使层级＋ 1，回滚时，会使层级－ 1
- 对于最外层的提交和回滚，使用的是数据库事务的 commit 和 rollBack
- 对于嵌套的内层的提交和回滚，使用的其实是事务保存点的释放和回滚
- 释放保存点时，会释放保存点的标识符，这个标识符在下次事务嵌套达到这个层级时，会被再次使用。

6.2.5 有效的事务

在上面的提交、回滚等方法的代码中，我们多次看到了一个 this->getIsActive() 。这是用于判断当前事务是否有效的一个方法，我们通过它，来看看什么样的一个事务，算是有效的：

```
public function getIsActive()
{
    return $this->_level > 0 && $this->db && $this->db->isActive;
}
```

方法很简单明了，一个有效的事务必须同时满足 3 个条件：

- _level > 0 。这是由于为 0 是，要么是刚刚初始化，要么是所有的事务已经提交或回滚了。也就是说，只有调用过了 begin() 但还没有调用过匹配的 commit() 或 rollBack() 的事务对象，才是有效的。

- 数据库连接要已经初始化。
- 数据库连接也必须是有效的。

6.3 AcitveReocrd 事件和关联操作

ActiveRecord 预定义的事件，都在 yiidbBaseActiveRecord 中进行了明确:

```
abstract class BaseActiveRecord extends Model implements ActiveRecordInterface
{
    const EVENT_INIT = 'init';                    // 初始化对象时触发
    const EVENT_AFTER_FIND = 'afterFind';         // 执行查询结束时触发
    const EVENT_BEFORE_INSERT = 'beforeInsert';   // 插入结束时触发
    const EVENT_AFTER_INSERT = 'afterInsert';     // 插入之前触发
    const EVENT_BEFORE_UPDATE = 'beforeUpdate';   // 更新记录前触发
    const EVENT_AFTER_UPDATE = 'afterUpdate';     // 更新记录后触发
    const EVENT_BEFORE_DELETE = 'beforeDelete';   // 删除记录前触发
    const EVENT_AFTER_DELETE = 'afterDelete';     // 删除记录后触发
    // ... ...
}
```

上述常量，定义了 ActiveRecord 对象常用的几个事件。这是预定义事件，我们可以直接拿来用。事件的定义具体看事件（Event） 部分的内容。

此外，作为 ActiveRecord 类的祖宗，yiibaseModel 类也定义了 2 个事件:

```
class Model extends Component implements IteratorAggregate, ArrayAccess, Arrayable
{
    const EVENT_BEFORE_VALIDATE = 'beforeValidate';   // 在验证之前触发
    const EVENT_AFTER_VALIDATE = 'afterValidate';     // 在验证之后触发
    // ... ...
}
```

因此，上述总共 10 个事件，可供开发者在写入业务流程时使用。

从上述事件来看，可以看出大部分事件是分别以 before 和 after 打头的成对事件。有些是“读”操作时才会触发的事件，有些是“写”操作时发生的事件。

而且，“写”与“写”之间也是相互区别的。比如，增、改、删 3 个写操作，都各自有一对事件先后在不同场景下触发。但这 3 种“写”操作不会被同时触发。

6.3.1 初始化事件

首先，第一个事件，无可争议的，是 EVENT_INIT 。这是由 yii\base\Object 所决定的。该事件在 init() 方法中被触发。而我们在属性（Property） 中已经说过这个方法是最早调用的几个方法之一。具体代码:

```php
public function init()
{
    parent::init();
    // 这里触发 EVENT_INIT 事件
    $this->trigger(self::EVENT_INIT);
}
```

虽然这个事件触发得早，但是实际使用中，这个事件使用频率不高。仅是因为有的代码不得不在初始化阶段执行，所以才提供了这个事件。而且，这个事件由于所处阶段特殊，不像有的事件，可以有一定的替代性。

比如，EVENT_AFTER_VALIDATE 和 EVENT_BEFORE_UPDATE 尽管泾渭分明，但是由于是相继触发，所以某些情况下可以在一定程度上互相替代。但是，上述 10 个事件中，仅有 EVENT_INIT 是在初始化阶段触发。所以，其具有不可替代性。

EVENT_INIT 事件通常用于初始化一些东西，从模块化的角度，可以简单看成是将当前类的 init() 方法的内容，作为一个 Event Handler 单独划分为一个模块。

6.3.2 AfterFind 事件

EVENT_AFTER_FIND 事件在完成查询后触发，注意该事件少有地没有对应的 Before 事件。

另外一个区别于其他事件的不同在于,该事件并非由 ActiveRecord 自身触发。而是由 yii\db\ActiveQuery 触发。准确的触发时点，是在查询完成后，向 ActiveRecord 填充字段全部内容后触发。

具体代码在 yii\db\ActiveQuery::populate()

```php
// 该方法为 ActiveQuery 将查询到的内容 $rows 填充到 ActiveReocrd 中去的方法
public function populate($rows)
{
    if (empty($rows)) {
        return [];
    }
    $models = $this->createModels($rows);
    if (!empty($this->join) && $this->indexBy === null) {
        $models = $this->removeDuplicatedModels($models);
    }
    if (!empty($this->with)) {
        $this->findWith($this->with, $models);
    }
    if (!$this->asArray) {

        // 重点在这个 foreach 里面的 afterFind(),
        // afterFind() 不干别的，就是专门调用
        // $this->trigger(self::EVENT_AFTER_FIND) 来触发事件的。
        foreach ($models as $model) {
```

```
            $model->afterFind();
        }
    }
    return $models;
}
```

上面的代码我们可以看出，在完成查询之后，查询到了多少个记录，就会触发多少次实例级别的 EVENT_AFTER_FIND 事件。事件的级别，请看事件（Event） 部分的内容。

EVENT_AFTER_FIND 事件，用于查询后一些内容的填充。比如，有一个 ActiveRecord，专门用于表示博客文章，那么通常他有一个字段，用于表示发布的确切时间。

假设客户希望在前台显示文章时，不直接显示确切时间，而是显示如“3 分钟前”“2 个月前”之类的相对时间。那么，我们就需要有一个将绝对时间转化成相对时间的过程。

那么，就可以把这个转换过程的代码，写在 EVENT_AFTER_FIND 事件的 Event Handler 里。

6.3.3 验证事件

验证事件是在验证时先后触发的 2 个事件，这 2 个事件均由 yii\base\Model::validate 触发:

```
public function validate($attributeNames = null, $clearErrors = true)
{
    if ($clearErrors) {
        $this->clearErrors();
    }

    // 这里的 beforeValidate() 会调用
    // $this->trigger(self::EVENT_BEFORE_VALIDATE, $event)
    // 来触发 EVENT_BEFORE_VALIDATE 事件。
    if (!$this->beforeValidate()) {
        return false;
    }
    // 下面是后续的验证代码，这里不用过多关注
    $scenarios = $this->scenarios();
    $scenario = $this->getScenario();
    if (!isset($scenarios[$scenario])) {
        throw new InvalidParamException("Unknown scenario: $scenario");
    }
    if ($attributeNames === null) {
        $attributeNames = $this->activeAttributes();
    }
    foreach ($this->getActiveValidators() as $validator) {
        $validator->validateAttributes($this, $attributeNames);
    }

```

```
    // 这里的 afterValidate() 会调用
    // $this->trigger(self::EVENT_AFTER_VALIDATE)
    // 来触发 EVENT_AFTER_VALIDATE 事件。
    $this->afterValidate();
    return !$this->hasErrors();
}
```

这两个事件正如其名称所表示的，触发顺序为先 EVENT_BEFORE_VALIDATE 后 EVENT_AFTER_VALIDATE 。

这两个事件中，EVENT_BEFORE_VALIDATE 常用于验证前的一些规范化处理。仍以博客文章的发布时间字段为例，在接收用户输入时，我们的应用接收一个字符类似“2015 年 3 月 8 日”之类的字符串。

但是数据库中我们一般并不以字符串形式保存时间，而是使用一个整型字段来保存。这主要涉及存储空间，日期比较和排序，检索效率等数据库优化问题，具体不展开。反正我们就是想把时间以整型形式进行保存。

那么，在验证用户输入之前，我们就需要将字符串类型的日期时间，转换成整型类型的日期时间。否则的话，验证就通不过。

这个转换过程，就可以写在 EVENT_BEFORE_VALIDATE 的 Event Handler 里面。

EVENT_BEFORE_VALIDATE 还有一个比较吸引人的特性，它的 Event Handler 可以返回一个 boolean 值，当为 false 时，表示后续的验证没必要进行了：

```
public function beforeValidate()
{
    // 创建一个 ModelEvent，并交给 trigger 在触发事件时使用
    $event = new ModelEvent;
    $this->trigger(self::EVENT_BEFORE_VALIDATE, $event);
    return $event->isValid;
}
```

上面的代码中，trigger() 将传入的第二个 $event 传递给 Event Handler，使得相关的这些个 Event Handler 可以在必要时修改 $event->isValid 的值。以此来决定是否可以取消后续的验证，直接视为验证没有通过。

EVENT_AFTER_VALIDATE 通常用于用户输入验证后的一些处理。比如，用于写入操作前的一些通用处理。因为后头接下来的事件，会分成插入、更新等独立事件。如果有一些写入前的通用处理，放在 EVENT_AFTER_VALIDATE 阶段是比较合适的。

至于验证通过与否，与 EVENT_AFTER_VALIDATE 事件没有关系，只要执行完所有验证了，这个事件就会被触发。而且，该事件的 Event Handler 没有返回值，无法干预验证结果。

6.3.4 “写”事件

“写”事件是指插入、更新、删除等写入操作时触发的事件。一般情况下，验证事件先于“写”事件被触发。

但这不是绝对的。Yii 允许在执行“写”操作时，不调用 validate() 进行验证，也就不触发验证事件。

下面，我们以更新操作 update 为例，来分析“写”事件。

首先，来看看 yii\db\BaseActiveRecord 里的有关代码:

```
public function save($runValidation = true, $attributeNames = null)
{
    // insert() 和 update() 具体实现由 ActiveRecord 定义
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}

// updateInternal() 由 update() 调用，
// 类似的有 deleteInternal()，由 ActiveRecord 定义，这里略去。
protected function updateInternal($attributes = null)
{
    // beforeSave() 会触发相应的 before 事件
    // 而且如果 beforeSave() 返回 false，就可以中止更新过程。
    if (!$this->beforeSave(false)) {
        return false;
    }
    $values = $this->getDirtyAttributes($attributes);

    // 没有字段有修改，那么实际上是不需要更新的。
    // 因此，直接调用 afterSave() 来触发相应的 after 事件。
    if (empty($values)) {
        $this->afterSave(false, $values);
        return 0;
    }

    // 以下为实际更新操作，不必细究。
    $condition = $this->getOldPrimaryKey(true);
    $lock = $this->optimisticLock();
    if ($lock !== null) {
        $values[$lock] = $this->$lock + 1;
        $condition[$lock] = $this->$lock;
    }
    $rows = $this->updateAll($values, $condition);
```

```
37      if ($lock !== null && !$rows) {
38          throw new StaleObjectException('The object being updated is outdated.');
39      }
40      $changedAttributes = [];
41      foreach ($values as $name => $value) {
42          $changedAttributes[$name] = isset($this->_oldAttributes[$name]) ? $this->_oldAttributes[$name] : null;
43          $this->_oldAttributes[$name] = $value;
44      }
45      // 重点看这里，触发了事件
46      $this->afterSave(false, $changedAttributes);
47      return $rows;
48  }
49
50  // 下面的 beforeSave() 和 afterSave() 会根据判断是更新操作还是插入操作，
51  // 以此来决定是触发 INSERT 事件还是 UPDATE 事件。
52  public function beforeSave($insert)
53  {
54      $event = new ModelEvent;
55      // $insert 为 true 时，表示当前是插入操作，是个新记录，要触发 INSERT 事件
56      // $insert 为 false 时，表示当前是插入操作，是个新记录，要触发 INSERT 事件
57      $this->trigger($insert ? self::EVENT_BEFORE_INSERT : self::EVENT_BEFORE_UPDATE, $event);
58      return $event->isValid;
59  }
60
61  public function afterSave($insert, $changedAttributes)
62  {
63      // $insert 为 true 时，表示当前是插入操作，是个新记录，要触发 INSERT 事件
64      // $insert 为 false 时，表示当前是插入操作，是个新记录，要触发 INSERT 事件
65      $this->trigger($insert ? self::EVENT_AFTER_INSERT : self::EVENT_AFTER_UPDATE, new AfterSaveEvent([
66          'changedAttributes' => $changedAttributes
67      ]));
68  }
```

就“写”操作而言，表面上调用的是 ActiveRecord 的 update() insert() delete() 等方法。

但是，更新最终调用的是 BaseActiveRecord::updateInteranl()，插入最终调用的是 ActiveRecord::insertInternal()，而删除最终调用的是 ActiveRecord::deleteInternal()。

这些 internal 方法，会触发相应的“写”事件，但不会调用验证方法，也不会触发验证事件。验证方法 validation() 由 update() insert() 调用。因此，验证事件也由这两个方法触发。

而且，这些 update() insert() 可以选择不进行验证，在压根不触发验证事件的情况下，就可以完成“写”操作。

因此，虽然 EVENT_AFTER_VALIDATE 和 EVENT_BEFORE_UPDATE 相继发生，在使用上有时可以有一定程度的替代。但是，其实两者是有严格界限的。原因就是验证事件可能在“写”操作过程中不

被触发。

此外，删除过程不触发验证事件。都要删掉的东西了，还需要验证么？

对于 internal 方法们，只是触发了相应的 before 和 after“写”事件。

其中，before 事件的 Event Handler 可以通过将 $event->isValid 设为 false 来中止“写”操作。

与在验证事件阶段中止时，视为验证没通过不同，这里的中止视为“写”操作失败。

与验证事件阶段类似，after 事件时由于生米已成熟饭，再也无法干预“写”操作的结果。

6.3.5 响应事件

前面提到的诸多预定义事件，为我们开发提供了方便。基本上使用这些预定义事件，就可以满足各种开发需求了。

但是凡事总有例外，特别是对于业务逻辑复杂的情况。比如，默认的删除事件，会在确确实实地要从数据表中删除记录时触发。但是，有的时候，我们并非真的想从数据表中删除记录，我们可能使用一个类似于“状态”的字段，在想要删除时，只是将记录的“状态”标记为“删除”。

这种需求并不少见。这样便于我们在后悔时，可以“恢复”删除。

从实质上是看，这其实是一个更新操作。那么预定义的 EVENT_BEFORE_DELETE 和 EVENT_AFTER_DELETE 就不适用了。

对此，我们可以自己定义事件来使用。具体的方法可以参见事件（Event） 部分的内容。

大致的代码可以是这样的:

```
class Post extends \yii\db\ActiveRecord {
    // 第一步：定义自己的事件
    const EVENT_BEFORE_MARK_DELETE = 'beforeMarkDelete';
    const EVENT_AFTER_MARK_DELETE = 'afterMarkDelete';

    // 第二步：定义 Event Handler
    public function onBeforeMarkDelete () {
        // ... do sth ...
    }

    // 第三步：在初始化阶段绑定事件和 Event Handler
    public function init()
    {
        parent::init();
        $this->trigger(self::EVENT_INIT);
        // 完成绑定
        $this->on(self::EVENT_BEFORE_MARK_DELETE, [$this, 'onBeforeMarkDelete']);
    }

```

```
    // 第四步：触发事件
    public function beforeSave($insert) {
        // 注意，重载之后要调用父类同名函数
        if (parent::beforeSave($insert)) {
            $status = $this->getDirtyAttributes(['status']);
            // 这个判断意会即可
            if (!empty($status) && $status['status'] == self::STATUS_DELETE) {
                // 触发事件
                $this->trigger(self::EVENT_BEFORE_MARK_DELETE);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

上面的代码理解个大致流程就 OK 了，不用细究。

在事件的响应上，我们有 2 个方法来写入我们的代码。

最直观的方式，是使用事件（Event） 中介绍的 Event Handler。也就是上面代码展现的，为类定义一个成员函数，作为 Event Handler。同时，在类的构造函数或初始化方法中，把事件和 Event Handler 绑定起来。最后，在合适的时候，触发事件即可。

另一种方式，是直接重载上面多次提到的各种 beforeSave() afterSave() beforeValidate() afterValidate() 等方法。比如，上面的例子可以改成:

```
class Post extends \yii\db\ActiveRecord {
    // 不需要定义自己的事件
    //const EVENT_BEFORE_MARK_DELETE = 'beforeMarkDelete';
    //const EVENT_AFTER_MARK_DELETE = 'afterMarkDelete';

    // 不需要定义 Event Handler
    //public function onBeforeMarkDelete () {
        // ... do sth ...
    //}

    // 不需要绑定事件和 Event Handler
    //public function init()
    //{
    //    parent::init();
    //    $this->trigger(self::EVENT_INIT);
    //    $this->on(self::EVENT_BEFORE_MARK_DELETE, [$this, 'onBeforeMarkDelete']);
    //}

```

```
19      // 只需要重载
20      public function beforeSave($insert) {
21          // 注意，重载之后要调用父类同名函数
22          if (parent::beforeSave($insert)) {
23              $status = $this->getDirtyAttributes(['status']);
24              // 这个判断意会即可
25              if (!empty($status) && $status['status'] == self::STATUS_DELETE) {
26                  // 不需要触发事件
27                  //$this->trigger(self::EVENT_BEFORE_MARK_DELETE);
28                  // 但是需要把原来 Event Handler 的内容放到这里来
29                  // ... do sth ...
30              }
31              return true;
32          } else {
33              return false;
34          }
35      }
36  }
```

对比来看，重载 beforeSave() 的方式要简洁很多。但是这种方式从严格意义上来讲，并不是正规的事件处理机制。只不过是利用了 Yii 已经预先定义好的函数调用流程。在使用中，需要格外注意的是，一定要在重载的函数中，调用父类的同名函数。否则的话，trigger() 不再被自动调用，相关事件就不会再被触发。整个类的事件机制，就全被破坏了。

6.3.6 关联操作

在实际开发中，有一种典型的场景，即对数据库某个表的某个记录进行修改时，需要对关联的表中的相关记录做相应的修改。

比如，一个典型的博客，表示文章的数据表中有一个字段用于记录当前文章有多少条评论。那么，当用户发表新评论时，另一个用于表示评论的表中，理所当然地要插入一条新记录。不可避免的，文章表中，被评论文章所对应的记录，其评论计数字段应当加 1。

那么这一过程怎么编程实现呢?

最直白的方法，是在操作评论记录的代码之前（后），写入相应的增加文章评论计数的代码。这样好理解，但是不同功能代码的界限不清晰。

另一种方法，是借助事件（Event），将增加文章评论计数的代码，写到评论 ActiveReocrd 的相应 Event Handler 中。比如，EVENT_AFTER_INSERT 。

这样子代码功能界限清晰，便于查找、修改和扩展。缺点是可能需要多看几个方法才能了解整个业务流程。实际中我们多采用这种方法。

在实现数据库记录的关联操作时，第一步就是要利用上述的各种事件，来产生关联性。其次，是要把这些关联性绑死在一起。也就是用数据库的事务。具体的原理，参考事务（Transaction） 部分的内容。

下面，我们以上面提到的博客文章新增一个评论为例，讲解如何实现关联操作。

声明需要事务支持的操作

在 ActiveRecord 中有一个方法，用于告诉 Yii 我们的哪些操作需要事务支持。对于插入、更新、删除的 1 个或多个操作需要事务支持时，可以在这个方法中进行声明。这个方法就是 ActiveRecord::transactions()

```
class ActiveRecord extends BaseActiveRecord
{
    // 定义插入、更新、删除操作，及表示 3 合 1 的 ALL
    const OP_INSERT = 0x01;
    const OP_UPDATE = 0x02;
    const OP_DELETE = 0x04;
    const OP_ALL = 0x07;

    // 需要事务支持时，重载这个方法。
    public function transactions()
    {
        return [];
    }

    // ... ...
}
```

默认情况下，这个 transactions() 返回一个空数组，表示不需要任何的事务支持。

我们的博客文章增加评论的案例中是要用到的，那么，我们可以在评论模型 Comment 中，作如下声明:

```
public function transactions() {
    return [
        'addComment' => self::OP_INSERT,
    ];
}
```

这个方法所返回的数组中，元素的键，如上面的 addComment 表示场景（scenario），元素值，表示的是操作的类型，即 OP_INSERT 等。

ActiveRecord 定义了 3 种可能会用到事务支持的操作 OP_INSERT OP_UPDATE OP_DELETE 分别表示插入、更新、删除。

可以把这 3 个操作两两组合作为 transactions() 所返回数组元素的值。如，self::OP_INSERT|self::OP_UPDATE “ 表示插入和更新操作。也可以直接使用 “OP_ALL 表示 3 种操作都包含。

启用事务

上一步中的 transactions() 被 ActiveRecord::isTransactional() 所调用:

```
// $operation 就是预定义的 OP_INSERT 等 3 种单一操作类型
public function isTransactional($operation)
{
    // 获取当前的 scenario
    $scenario = $this->getScenario();
    $transactions = $this->transactions();
    return isset($transactions[$scenario]) && ($transactions[$scenario] & $operation);
}
```

这个 isTransactional() 就是判断当前场景下，当前操作类型，是否已经在 transactions() 中声明为需要事务支持。

而这个 isTransactional() 又被各种“写”操作方法所调用。在我们的博客文章新增评论的案例中，就是被 insert() 所调用:

```
public function insert($runValidation = true, $attributes = null)
{
    if ($runValidation && !$this->validate($attributes)) {
        Yii::info('Model not inserted due to validation error.', __METHOD__);
        return false;
    }
    // 这里调用了 isTransactional()，判断当前场景下，
    // 插入操作是否需要事务支持
    if (!$this->isTransactional(self::OP_INSERT)) {
        // 无需事务支持，那就直接 insert 了事
        return $this->insertInternal($attributes);
    }
    // 以下是需要事务支持的情况，那就启用事务
    $transaction = static::getDb()->beginTransaction();
    try {
        $result = $this->insertInternal($attributes);
        if ($result === false) {
            $transaction->rollBack();
        } else {
            $transaction->commit();
        }
        return $result;
    } catch (\Exception $e) {
        $transaction->rollBack();
        throw $e;
    }
}
```

很明显的，我们只需要在 transactions() 中声明需要事务支持的操作就足够了。后续的怎么使声明生效的，Yii 框架已经替我们写好了。

在上面 insert() 的代码中，通过我们的声明，Yii 发现需要事务支持，于是就调用了 static::getDb()->beginTransaction() 来启用事务。事务的原理，请看事务（Transaction） 部分的内容。

在事件响应中写入关联操作

接下来，我们在关联的事件，如 EVENT_AFTER_INSERT 中，写入关联操作。这里，我们就是要更新博客文章模型 Post 的评论计数字段。

因此，可以在评论模型 Comment 完成插入后的 EVENT_AFTER_INSERT 阶段，写入更新 Post::comment_counter 的代码。如果使用简洁形式的事件响应方式，那么代码可以是:

```
class Comment extends \yii\db\ActiveRecord {
    // 通过重载 afterSave 来“响应”事件
    public function afterSave($insert) {
        if (parent::beforeSave($insert)) {
            // 新增一个评论
            if ($insert) {
                // 关联 Post 的操作，评论计数字段＋ 1
                $post = Post::find($this->postId);
                $post->comment_counter += 1;
                $post->save(false);
            }
        }
    }
}
```

回顾下实现关联操作的过程，其实就 2 步：

- 先是在 transactions() 中声明要事务支持的操作类型，比如上面的例子，声明的是插入操作。
- 在合适事件响应函数中，写下关联操作代码。

6.4 乐观锁与悲观锁

Web 应用往往面临多用户环境，这种情况下的并发写入控制，几乎成为每个开发人员都必须掌握的一项技能。

在并发环境下，有可能会出现脏读（Dirty Read）、不可重复读（Unrepeatable Read）、幻读（Phantom Read）、更新丢失（Lost update）等情况。具体的表现可以自行搜索。

为了应对这些问题，主流数据库都提供了锁机制，并引入了事务隔离级别的概念。这里我们都不作解释了，拿这些关键词一搜，网上大把大把的。

但是，就于具体开发过程而言，一般分为悲观锁和乐观锁两种方式来解决并发冲突问题。

6.4.1 乐观锁

乐观锁（optimistic locking）表现出大胆、务实的态度。使用乐观锁的前提是，实际应用当中，发生冲突的概率比较低。他的设计和实现直接而简洁。目前 Web 应用中，乐观锁的使用占有绝对优势。

因此，Yii 也为 ActiveReocrd 提供了乐观锁支持。

根据 Yii 的官方文档，使用乐观锁，总共分 4 步：

- 为需要加锁的表增加一个字段，用于表示版本号。当然相应的 Model 也要为该字段的加入，作出适当调整。比如，rules() 中要加入该字段。
- 重载 yii\db\ActiveRecord::optimisticLock() 方法，返回上一步中的字段名。
- 在记录的修改页面表单中，加入一个 <input type="hidden"> 用于暂存读取时的记录的版本号。
- 在保存代码的地方，使用 try ... catch 看看是否能捕获一个 yii\db\StaleObjectException 异常。如果是，说明在本次修改这个记录的过程中，该记录已经被修改过了。简单应对的话，可以作出相应提示。智能点的话，可以合并不冲突的修改，或者显示一个 diff 页面。

从本质上来讲，乐观锁并没有像悲观锁那样使用数据库的锁机制。乐观锁通过在表中增加一个计数字段，来表示当前记录被修改的次数（版本号）。

然后在更新、删除前通过比对版本号来实现乐观锁。

声明版本号字段

版本号是实现乐观锁的根本所在。所以第一步，我们要告诉 Yii，哪个字段是版本号字段。这个由 yii\db\BaseActiveRecord 负责:

```
public function optimisticLock()
{
    return null;
}
```

这个方法返回 null，表示不使用乐观锁。那么我们的 Model 中，要对此进行重载。返回一个字符串，表示我们用于标识版本号的字段。比如可以这样:

```
public function optimisticLock()
{
    return 'ver';
}
```

说明当前的 ActiveRecord 中，有一个 ver 字段，可以为乐观锁所用。那么 Yii 具体是如何借助这个 ver 字段实现乐观锁的呢?

更新过程

具体来讲，使用乐观锁之后的更新过程，就是这么一个流程：

- 读取要更新的记录。
- 对记录按照用户的意愿进行修改。当然，这个时候不会修改 ver 字段。这个字段对用户是没意义的。
- 在保存记录前，再次读取这个记录的 ver 字段，与之前读取的值进行比对。
- 如果 ver 不同，说明在用户修改过程中，这个记录被别人改动过了。那么，我们要给出提示。
- 如果 ver 相同，说明这个记录未被修改过。那么，对 ver + 1，并保存这个记录。这样子就完成了记录的更新。同时，该记录的版本号也加了 1。

由于 ActiveRecord 的更新过程最终都需要调用 yii\db\BaseActiveRecord::updateInteranl() ，理所当然地，处理乐观锁的代码，也就隐藏在这个方法中：

```
protected function updateInternal($attributes = null)
{
    if (!$this->beforeSave(false)) {
        return false;
    }
    // 获取等下要更新的字段及新的字段值
    $values = $this->getDirtyAttributes($attributes);
    if (empty($values)) {
        $this->afterSave(false, $values);
        return 0;
    }
    // 把原来 ActiveRecord 的主键作为等下更新记录的条件，
    // 也就是说，等下更新的，最多只有 1 个记录。
    $condition = $this->getOldPrimaryKey(true);

    // 获取版本号字段的字段名，比如 ver
    $lock = $this->optimisticLock();

    // 如果 optimisticLock() 返回的是 null，那么，不启用乐观锁。
    if ($lock !== null) {
        // 这里的 $this->$lock ，就是 $this->ver 的意思；
        // 这里把 ver+1 作为要更新的字段之一。
        $values[$lock] = $this->$lock + 1;

        // 这里把旧的版本号作为更新的另一个条件
        $condition[$lock] = $this->$lock;
    }
    $rows = $this->updateAll($values, $condition);

    // 如果已经启用了乐观锁，但是却没有完成更新，或者更新的记录数为 0；
```

```
    // 那就说明是由于 ver 不匹配，记录被修改过了，于是抛出异常。
    if ($lock !== null && !$rows) {
        throw new StaleObjectException('The object being updated is outdated.');
    }
    $changedAttributes = [];
    foreach ($values as $name => $value) {
        $changedAttributes[$name] = isset($this->_oldAttributes[$name]) ? $this->_oldAttributes[$name] : null;
        $this->_oldAttributes[$name] = $value;
    }
    $this->afterSave(false, $changedAttributes);
    return $rows;
}
```

从上面的代码中，我们不难得出：

- 当 optimisticLock() 返回 null 时，乐观锁不会被启用。
- 版本号只增不减。
- 通过乐观锁的条件有 2 个，一是主键要存在，二是要能够完成更新。
- 当启用乐观锁后，只有下列两种情况会抛出 StaleObjectException 异常：
  - 当记录在被别人删除后，由于主键已经不存在，更新失败。
  - 版本号已经变更，不满足更新的第二个条件。

删除过程

与更新过程相比，删除过程的乐观锁，更简单，更好理解。代码仍在 yii\db\BaseActiveRecord 中：

```
public function delete()
{
    $result = false;
    if ($this->beforeDelete()) {
        // 删除的 SQL 语句中，WHERE 部分是主键
        $condition = $this->getOldPrimaryKey(true);
        // 获取版本号字段的字段名，比如 ver
        $lock = $this->optimisticLock();
        // 如果启用乐观锁，那么 WHERE 部分再加一个条件，版本号
        if ($lock !== null) {
            $condition[$lock] = $this->$lock;
        }
        $result = $this->deleteAll($condition);
        if ($lock !== null && !$result) {
            throw new StaleObjectException('The object being deleted is outdated.');
        }
```

```
17          $this->_oldAttributes = null;
18          $this->afterDelete();
19      }
20      return $result;
21  }
```

比起更新过程，删除过程确实要简单得多。唯一的区别就是省去了版本号＋ 1 的步骤。都要删除了，版本号＋ 1 有什么意义?

6.4.2 乐观锁失效

乐观锁存在失效的情况，属小概率事件，需要多个条件共同配合才会出现。如：

- 应用采用自己的策略管理主键 ID。如，常见的取当前 ID 字段的最大值＋ 1 作为新 ID。
- 版本号字段 ver 默认值为 0 。
- 用户 A 读取了某个记录准备修改它。该记录正好是 ID 最大的记录，且之前没被修改过，ver 为默认值 0。
- 在用户 A 读取完成后，用户 B 恰好删除了该记录。之后，用户 C 又插入了一个新记录。
- 此时，阴差阳错的，新插入的记录的 ID 与用户 A 读取的记录的 ID 是一致的，而版本号两者又都是默认值 0。
- 用户 A 在用户 C 操作完成后，修改完成记录并保存。由于 ID、ver 均可以匹配上，因此用户 A 成功保存。但是，却把用户 C 插入的记录覆盖掉了。

乐观锁此时的失效，根本原因在于应用所使用的主键 ID 管理策略，正好与乐观锁存在极小程度上的不兼容。

两者分开来看，都是没问题的。组合到一起之后，大致看去好像也没问题。但是 bug 之所以成为 bug，坑之所以能够坑死人，正是由于其隐蔽性。

对此，也有一些意见提出来，使用时间戳作为版本号字段，就可以避免这个问题。但是，时间戳的话，如果精度不够，如毫秒级别，那么在高并发，或者非常凑巧情况下，仍有失效的可能。而如果使用高精度时间戳的话，成本又太高。

使用时间戳，可靠性并不比使用整型好。问题还是要回到使用严谨的主键成生策略上来。

6.4.3 悲观锁

正如其名字，悲观锁（pessimistic locking）体现了一种谨慎的处事态度。其流程如下：

- 在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。
- 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。具体响应方式由开发者根据实际需要决定。

- 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
- 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

悲观锁确实很严谨，有效保证了数据的一致性，在 C/S 应用上有诸多成熟方案。但是他的缺点与优点一样的明显：

- 悲观锁适用于可靠的持续性连接，诸如 C/S 应用。对于 Web 应用的 HTTP 连接，先天不适用。
- 锁的使用意味着性能的损耗，在高并发、锁定持续时间长的情况下，尤其严重。Web 应用的性能瓶颈多在数据库处，使用悲观锁，进一步收紧了瓶颈。
- 非正常中止情况下的解锁机制，设计和实现起来很麻烦，成本还很高。
- 不够严谨的设计下，可能产生莫名其妙的，不易被发现的，让人头疼到想把键盘一巴掌碎的死锁问题。

总体来看，悲观锁不大适应于 Web 应用，Yii 团队也认为悲观锁的实现过于麻烦，因此，ActiveRecord 也没有提供悲观锁。

作为 Yii 的构成基因之一的 Ruby on rails，他的 ActiveReocrd 模型，倒是提供了悲观锁，但是使用起来也很麻烦。

6.4.4 悲观锁的实现

虽然悲观锁在 Web 应用上存在诸多不足，实现悲观锁也需要解决各种麻烦。但是，当用户提出他就是要用悲观锁时，牙口再不好的码农，就是咬碎牙也是要啃下这块骨头来。

对于一个典型的 Web 应用而言，这里提供个人常用的方法来实现悲观锁。

首先，在要锁定的表里，加一个字段如 locked_at ，表示当前记录被锁定时的时间，当为 0 时，表示该记录未被锁定，或者认为这是 1970 年时加的锁。

当要修改某个记录时，先看看当前时间与 locked_at 字段相差是否超过预定的一个时长 T，比如 30 min ，1 h 之类的。

如果没超过，说明该记录有人正在修改，我们暂时不能打开（读取）他来修改。否则，说明可以修改，我们先将当前时间戳保存到该记录的 locked_at 字段。那么之后的时长 T 内如果有人要来改这个记录，他会由于加锁失败而无法读取，从而无法修改。

我们在完成修改后，即将保存时，要比对现在的 locked_at 。只有在 locked_at 一致时，才认为刚刚是我们加的锁，我们才可以保存。否则，说明在我们加锁后，又有人加了锁正在修改，或者已经完成了修改，使得 locked_at 归 0。

这种情况主要是由于我们的修改时长过长，超过了预定的 T。原先的加锁自动解开，其他用户可以在我们加锁时刻再过 T 之后，重新加上自己的锁。换句话说，此时悲观锁退化为乐观锁。

大致的原理性代码如下：

```
1 // 悲观锁 AR 基类，需要使用悲观锁的 AR 可以由此派生
2 class PLockAR extends \yii\db\BaseActiveRecord {
```

```php
    // 声明悲观锁使用的标记字段，作用类似于 optimisticLock() 方法
    public function pesstimisticLock()  {
        return null;
    }

    // 定义锁定的最大时长，超过该时长后，自动解锁。
    public function maxLockTime() {
        return 0;
    }

    // 尝试加锁，加锁成功则返回 true
    public function lock() {
        $lock = $this->pesstimisticLock();
        $now = time();
        $values = [$lock => $now];
        // 以下 2 句，更新条件为主键，且上次锁定时间距现在超过规定时长
        $condition = $this->getOldPrimaryKey(true);
        $condition[] = ['<', $lock, $now - $this->maxLockTime()];

        $rows = $this->updateAll($values, $condition);
        // 加锁失败，返回 false
        if (! $rows) {
            return false;
        }
        return true;
    }

    // 重载 updateInternal()
    protected function updateInternal($attributes = null)
    {
        // 这些与原来代码一样
        if (!$this->beforeSave(false)) {
            return false;
        }
        $values = $this->getDirtyAttributes($attributes);
        if (empty($values)) {
            $this->afterSave(false, $values);
            return 0;
        }
        $condition = $this->getOldPrimaryKey(true);

        // 改为获取悲观锁标识字段
        $lock = $this->pesstimisticLock();

        // 如果 $lock 为 null，那么，不启用悲观锁。
```

```
48          if ($lock !== null) {
49              // 等下保存时，要把标识字段置 0
50              $values[$lock] = 0;
51
52              // 这里把原来的标识字段值作为更新的另一个条件
53              $condition[$lock] = $this->$lock;
54          }
55          $rows = $this->updateAll($values, $condition);
56
57          // 如果已经启用了悲观锁，但是却没有完成更新，或者更新的记录数为 0；
58          // 那就说明之前的加锁已经自动失效了，记录正在被修改，
59          // 或者已经完成修改，于是抛出异常。
60          if ($lock !== null && !$rows) {
61              throw new StaleObjectException('The object being updated is outdated.');
62          }
63          $changedAttributes = [];
64          foreach ($values as $name => $value) {
65              $changedAttributes[$name] = isset($this->_oldAttributes[$name]) ? $this->_oldAttributes[$name] : null;
66              $this->_oldAttributes[$name] = $value;
67          }
68          $this->afterSave(false, $changedAttributes);
69          return $rows;
70      }
71  }
```

上面的代码对比乐观锁，主要不同点在于：

- 新增加了一个加锁方法，一个获取锁定最大时长的方法。
- 保存时不再是把标识字段＋ 1，而是把标识字段置 0。

在具体使用方法上，可以参照以下代码:

```
1  // 从 PLockAR 派生模型类
2  class Post extends PLockAR {
3      // 重载定义悲观锁标识字段，如 locked_at
4      public function pesstimisticLock() {
5          return 'locked_at';
6      }
7      // 重载定义最大锁定时长，如 1 小时
8      public function maxLockTime() {
9          return 3600000;
10     }
11 }
12
13 // 修改前要尝试加锁
14 class SectionController extends Controller {
```

```
15      public function actionUpdate($id)
16      {
17          $model = $this->findModel($id);
18
19          if ($model->load(Yii::$app->request->post()) && $model->save()) {
20              return $this->redirect(['view', 'id' => $model->id]);
21          } else {
22              // 加入一个加锁的判断
23              if (!$model->lock()) {
24                  // 加锁失败
25                  // ... ...
26              }
27              return $this->render('update', [
28                  'model' => $model,
29              ]);
30          }
31      }
32  }
```

上述方法实现的悲观锁，避免了使用数据库自身的锁机制，契合 Web 应用的特点，具有一定的适用性，但是也存在一定的缺陷：

- 最长允许锁定时长会带来一定的副作用。时间定得长了，可能要等很长时间，才能重新编辑非正常解锁的记录。时间定得短了，则经常退化成乐观锁。
- 时间戳精度问题。如果精度不够，那么在加锁时，与我们讨论过的乐观锁失效存，在同样的漏洞。
- 这种形式的锁定，只是应用层面的锁定，并非数据库层面的锁定。如果存在应用之外对于数据库的写入操作。这个锁定机制是无效的。

CHAPTER 7

附录

7.1 附录 1：Yii2.0 对比 Yii1.1 的重大改进

这部分内容是专门为已经有 Yii1.1 基础的读者朋友写的。将 Yii2.0 与 Yii1.1 的不同点着重写出来，对比学起来会快得多。而对于从未接触过 Yii 的读者朋友，这部分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，本书的正文部分会讲清楚的。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。

7.1.1 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，Yii2.0 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[1]
- 延迟静态绑定，具体可以查看 Late Static Bindings[2]
- PHP 标准日期时间[3]

[1] http://php.net/manual/en/book.spl.php
[2] http://php.net/manual/en/language.oop5.late-static-bindings.php
[3] http://php.net/manual/en/book.datetime.php

- 特性 (Traits)[4]
- 使用 PHP intl 扩展实现国际化支持，具体可以查看 PECL init[5] 。

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。

7.1.2 **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。

7.1.3 **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。

7.1.4 **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);
```

[4]http://php.net/manual/en/language.oop5.traits.php
[5]http://php.net/manual/en/book.intl.php

7.1.5 **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:

```
1 // 路径别名
2 Yii::setAlias('@foo', '/path/to/foo');
3
4 // URL 别名
5 Yii::setAlias('@bar', 'http://www.example.com');
```

别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 的别名，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 (Alias) 。

7.1.6 **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，可以使用 $this->context 。这个 $this->context 是指谁调用了 yii\base\View::renderFile() 来渲染这个视图。一般是某个控制器，也可以是其他实现了 yii\base\ViewContextInterface 接口的对象。

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。

7.1.7 **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:

```
1  // Yii2.0 使用 load() 等同于下面 Yii1.1 的用法
2  $model = new Post;
3  if ($model->load($_POST)) {
4      ... ...
5  }
6
7  // Yii1.1 中常用的套路
8  if (isset($_POST['Post'])) {
9      $model->attributes = $_POST['Post'];
10 }
```

另外一个重要变化就是 Yii2.0 中改变了 Model 应用于不同场景的逻辑。通过引入 yii\base\Model::scenarios() 来集中管理场景，使得一个 Model 所有适用的场景都比较清晰，一目了然。而 Yii1.1 是没有一个统一管理场景的方法的。

由此带来的一个很容易出现的问题就是，当你声明一个 Model 处于某一场景时，可能由于拼写错误，不小心将场景的名称写错了，那么在 Yii1.1 中，这个错误的场景并没有任何的提示。假设有以下情况:

```
1  class UserForm extends CFormModel
2  {
```

```
    public $username;
    public $email;
    public $password;
    public $password_repeat;
    public $rememberMe=false;

    public function rules()
    {
        return array(
            // username 和 password 在所有场景中都要验证
            array('username, password', 'required'),

            // email 和 password_repeat 只在注册场景中验证
            array('email, password_repeat', 'required', 'on'=>'Registration'),
            array('email', 'email', 'on'=>'Registration'),

            // rememberMe 仅在登陆场景中验证
            array('rememberMe', 'boolean', 'on'=>'Login'),
        );
    }
}
```

这里针对 UserForm 的注册和登陆两个场景，设定了不同的验证规则。接下来，你要在注册场景中使用这个 UserForm，但你一不小心将 Registration 场景设定成了 SignUp，说实在，我不是学英文出身的，这两个单词的意思在我眼里是一样一样的。只是 Yii 不会智能到把这两个场景等同起来。那么 Yii1.1 将不会有任何的提示，并自动地使用第一个验证规则，而用户注册时填写的 email 和 password_repeat 字段就被抛弃了。这在实际编程中，是经常出现的一个低级错误。

从这里可以看到，Yii1.1 中对于场景，没有一个集中统一的管理，也就是说一个 Model 可适用的场景，是不确定的、任意的。通过 rules() 你很难一眼看出来一个 Model 可以适用于多少个场景，每个场景下都有哪些字段是有效的、需要验证的。

而在 Yii2.0 中，由于引入了 yii\base\Model::scenarios() 新的方法，将本 Model 所有适用的场景，及不同场景下的有效字段都进行了声明，这个逻辑就显得清晰了。而且，如果使用了一个未声明的场景，Yii2.0 会有相应的提示，这避免了上面这个低级错误的可能：

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        return [
```

```
            'login' => ['username', 'password', 'rememberMe'],
            'registration' => ['username', 'email', 'password', 'password_repeat'],
        ];
    }
}
```

这样看来，是不是很清晰？这个 User 仅有两种场景，每种场景的有效字段也一目了然。而至于具体场景下每个字段的验证规则，仍然由 yii\base\Model::rules() 来确定。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。或者为了更加明显的表示某一字段在特定场景下是无效的，可以给这个字段加上！前缀。

在默认情况下，yii\base\Model::scenarios() 所有适用的场景和对应的字段由 yii\base\Model::rules() 的内容自动生成。也就是说，如果你的 rules() 很完备、很清晰，那么也是不需要重载这个 scenarios() 的。这种情况下，Yii1.1 和 Yii2.0 在这一点上的表现形式，是一样的。但是，个人经验看，我更倾向于将 scenarios() 声明清楚，而在 rules() 中，仅指定字段的验证规则，而不涉及场景的内容。这样的逻辑更加清晰，便于其他团队成员阅读你的代码，也便于后续的维护和开发。

7.1.8 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？

7.1.9 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

```php
// 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。

Yii2.0 还淘汰掉了原来的 CActiveRecordBehavior 类。在 Yii2.0 中，将行为与类进行绑定采用了统一的方式进行，具体请参考行为（Behavior） 的有关内容。

Yii2.0 中，ActiveRecord 得到极大的加强，在相关的章节中我们已经进行专门的讲解。

这里的内容主要是点一点 Yii2.0 之于 Yii1.1 的变化。大致了解下就可以了，主要还是要看正文专门针对每个知识点的深入讲解。

7.2 附录 2：Yii 的安装

由于 Yii2.0 使用了许多 PHP 的新特性，因此，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。

7.2.1 使用 Composer 安装 Yii

官方推荐使用 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。

这里并不打算介绍 Composer 的用法，这方面的内容通过官方文档或者搜索引擎都可以找到。使用 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。对于团队开发而言，环境切换功能很实用。对于大型应用，前后台分离既是逻辑上的划分，也是安全上的需要。高级模版功能相对丰富，因此，本书在大多数情况下，都使用高级模版所创建应用进行讲解。至于基本模版，原理上是相通的。

Yii2.0 要求 Composer 必须安装 composer asset 插件。这个插件使得 Composer 可以兼容实现 NPM 和 BOWER 包管理器的功能。NPM[6] 和 BOWER[7] 主要用于前端资源（如 JS 库等）的管理。

```
# 安装 Composer，如果已经安装过，可不必再安装
# curl -s http://getcomposer.org/installer | php

# 对于已经安装过 Composer 的，可以对其进行更新
php ../composer.phar self-update

# 为 Composer 安装 composer asset 插件
php ../composer.phar global require "fxp/composer-asset-plugin:1.0.0-beta2"

# 使用高级模版安装 Yii 应用到 digpage.com 目录下
php ../composer.phar create-project --prefer-dist yiisoft/yii2-app-advanced digpage.com

# 使用基础模版安装
# composer create-project --prefer-dist yiisoft/yii2-app-basic digpage.com
```

如果想使用最新的开发版本的 Yii 基础模版，可加入 --stability=dev 参数。

7.2.2 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 /path/to/digpage.com 目录。
3. 修改 config/web.php 文件，输入 cookieValidationKey 配置项密钥。这个密钥主要用于 cookie 验证。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 validation
    'cookieValidationKey' => 'enter your secret key here',
```

7.2.3 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 的配置安装，这些内容从官方文档和搜索引擎都可以找到相关内容。这里大体上介绍如何配置 Web 服务器，使其能够让 Yii 跑起来。

由于高级模版应用具有前台和后台。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：

[6] https://www.npmjs.org/
[7] http://bower.io/

```
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/
```

无论使用何种方式，目的都是分离前台和后台。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，采不同主机名、端口进行的分离方式，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

    server_name localhost;
    root        /path/to/digpage.com/frontend/web;
    index       index.php;

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

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        include fastcgi.conf;
        fastcgi_pass   127.0.0.1:9000;
    }

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

# for backend
server {
    # ...other settings...
```

```
    listen 81;
    server_name localhost;
    root /path/to/digpage.com/backend/web;
    # ...other settings...
}
```

如果使用 Apache，也是分前台和后前的配置，以前台为例：

```
DocumentRoot "path/to/digpage.com/frontend/web"

<Directory "path/to/digpage.com/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/digpage.com/backend/web 即可。

7.2.4 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。如法炮制另一个独立的，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/digpage.com/ 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。只不过在代码组织上，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，它通常用于维护，是个命令行应用。

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。同时，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。这个目录从字面的意思看，就是通用，对于组织到一起的 frontend backend console 而言，common 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。

更多关于 Yii 应用的目录结构的内容，请看Yii 应用的目录结构和入口脚本 部分。

7.2.5 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。在 Yii 的高级模版应用中，引入了环境的概念。

环境就是指开发环境、测试环境、产品环境等。对于 Yii 而言，所谓的环境，就是一组与运行环境相关的配置文件和入口脚本。

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，每次需要切换环境时，就运行脚本，由开发者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。在 Yii 中，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。

在切换环境时，只需要一行命令就全部搞定：

```
php /path/to/digpage.com/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。下面是输出的内容：

```
Yii Application Initialization Tool v1.0

Which environment do you want the application to be initialized in?

  [0] Development
  [1] Production

  Your choice [0-1, or "q" to quit] 0    // 这里选择了 Development 环境

  Initialize the application under 'Development' environment? [yes|no] yes

  Start initialization ...

   generate yii
   generate common/config/main-local.php
   generate common/config/params-local.php
   generate backend/config/main-local.php
   generate backend/config/params-local.php
   generate backend/web/index.php
   generate backend/web/index-test.php
   generate frontend/config/main-local.php
   generate frontend/config/params-local.php
   generate frontend/web/index.php
   generate frontend/web/index-test.php
   generate console/config/main-local.php
   generate console/config/params-local.php
```

```
  generate cookie validation key in backend/config/main-local.php
  generate cookie validation key in frontend/config/main-local.php
     chmod 0777 backend/runtime
     chmod 0777 backend/web/assets
     chmod 0777 frontend/runtime
     chmod 0777 frontend/web/assets
     chmod 0755 yii

 ... initialization completed.
```

从上面的输出可以看出来，init 脚本其实做了 3 件事：

- 复制文件到相应位置，覆盖当前配置。
- 生成 cookieValidationKey 并写入相应文件。
- 设置相关文件和目录的权限。

如果想更加便捷，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，并要求全部覆盖当前配置文件。

7.2.6 检验安装情况

在这一篇，我们已经完成了 Yii 的安装，可能使用了 Composer，也可能使用了压缩包。接着，我们配置好了 Web 服务器。最后，我们运行了 init 命令。那么，Yii 应用的基本框架就已经搭建好了。你可在你的浏览器中试试效果。使用 http://localhost:80/ 可以打开网站前台，使用 http://localhost:81 可以打开后台。