参考：<http://blog.csdn.net/qq_27416209/article/details/53883590>

[composer基本用法](http://blog.csdn.net/qq_27416209/article/details/53883590)

2016-12-26 09:38 263人阅读 [评论](http://blog.csdn.net/qq_27416209/article/details/53883590#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/qq_27416209/article/details/53883590#report)

 分类：

php（17）

目录[(?)[+]](http://blog.csdn.net/qq_27416209/article/details/53883590)

[**Java**](http://lib.csdn.net/base/java)有Maven, [**Node.js**](http://lib.csdn.net/base/nodejs)有npm, ROR有gem, [**Python**](http://lib.csdn.net/base/python)有pip.这些语言的程序员在开心地使用包管理工具加速开发效率时，PHPer们还在复制粘贴的黑暗中。[**PHP**](http://lib.csdn.net/base/php)在Composer之前，包管理的历史不堪回首。

在相当长的一段时间内，如果应用依赖于第三方库，PHPer需要拷贝这些库的源代码， 或者通过PEAR、PECL安装。如果第三方库又依赖于更多的第三方库，那么很快就会进入依赖的黑洞。直到Composer出现，PHPer们看到了属于[**php**](http://lib.csdn.net/base/php)的包管理的曙光。

注：Composer更新很慢，很难成功的

下面将以创建一个电商网站为例，介绍Composer的使用方法。

**配置文件**

在我们开始一个项目的时候，首先会给项目取一个名字，我们暂且叫丝绸之路吧，代号silk。首先要写一个Composer的配置文件（me:composer.json），来描述项目，为此，在项目的根目录下，建立文件名为composer.json的配置文件。内容如下：

<!-- lang: js -->

{

"name": "meta/silk",

"description": "another e-commerce website",

"keywords": ["silk", "online shop", "good"],

"homepage": "http://www.xxx.com ",

"time": "2014-12-30",

"license": "MIT",

"authors": [

{

"name": "Elvis Lim",

"email": "elvis@xxx.com",

"homepage": "http://www.xxx.com",

"role": "Engineer"

}

]}

如果您熟悉JSON格式，那么上面这段内容不言而喻。事实上，这些键值对都是可选的。也就是说，可以都不写。但是如果要把项目打包成公共包发布，那么这些还是需要写上的，给你的包取个名字总不为过。让我们来过一下这些键值对的意义吧。

<!-- lang: js -->

"name": "meta/silk",

name, 表示包的名称。如果你经常在Github上混，那这个值的表达方式一定非常熟悉啦。解释下，通常包名包含两部分，并且以 / 分隔。斜杆前面部分，代表包的所有者。目前大部分的包作者都喜欢用Github的用户名作为这部分的值。斜杆后面部分代表包的名称。尽量保持简单和有意义些，便于记忆和传播。大部分情况下，很多人会用Github的代码库名称来命名，当然，这种情况下，代码要存在Github比较有意义。

<!-- lang: js -->

"description": "another e-commerce website",

应用简介，这部分尽量简洁介绍下项目，别长篇大论。如果确实有很多话要说，那么可以写在README.md文件里。

<!-- lang: js -->

"keywords": ["silk", "online shop", "good"],

关键词的值是一个字符串数组，在发布成公用库的是时候，作为元数据信息，有利于包的搜索和发现。

<!-- lang: js -->

"homepage": "http://www.xxx.com ",

主页，可以放你想放的任何页面地址。

<!-- lang: js -->

"license": "MIT",

如果你决定将包公开发布，那么记得选择一个合适的许可证。这样别的程序员在引用包的时候，通过查看许可证，确保没有法律上的问题。

<!-- lang: js -->

"authors":[{}]

作者字段可以包含一个对象数组，也就是说可以提供多个作者信息。

目前为止，都是关于包本身的信息描述。作为一个电商网站，能够发送电子邮件、导出订单到Excel表是基本需求，这个时候自然想到了使用现有的库来实现这些功能。要获取这些库，最简单的方式是，搜索下这些库，找到下载地址，下载个zip包，然后解压到相应目录下，根据文档引入相应的文件。使用Composer，可以更加自动和优雅地完成这个过程，这就是Composer的依赖管理。

**依赖管理**

在composer.json文件里增加一个新的字段：require。这个字段的值是一个对象，同样以键值对的形式构成。以上述提到的两个依赖位置，写成composer管理的方式如下：

<!-- lang: js -->

“require”: {

"swiftmailer/swiftmailer": 5.3.\*[@dev](https://my.oschina.net/Thinker277),

"phpoffice/phpexcel": "dev-master"

}

以swiftmailer为例，swiftmailer/swiftmailer 代表的是包名称，5.3.[@dev](https://my.oschina.net/Thinker277), 是版本信息。合起来的意思就是说，我们将要开发的应用，依赖于swiftmailer的5.3.版本。其中：

5.3.\*表示，可以使用5.3.1版本，也可以使用5.3.2版本，composer在获取的时候，将寻找5.3版本下最新的版本。版本号支持一些更加宽泛的约束，比如>=1.0, >=1.0, <2.0，更加具体的信息可以查看：<http://docs.phpcomposer.com/01-basic-usage.md#The-require-Key>

@dev表示可以获取开发版本。通常，开发版本意味非稳定版本，很可能存在bug。稳定性标签可以作用于特定的依赖项，也可以作用于全局。

作用特定依赖项：默认情况下，composer只会获取稳定版本，如果这个例子我们不加@dev约束，而5.3.\*版本都是开发版本，那么在获取的时候composer就会报错，指出改版本不符合要求。如果确定这个开发版本没有问题，那么就可以通过加@dev，让Composer获取这个开发版本。

全局稳定性设置：通过设置minimum-stability的值，来告诉Composer当前开发的项目的依赖要求的包的全局稳定性级别，它的值包括：dev、alpha、beta、RC、stable，stable是默认值。

至此，两个依赖添加完毕，我们可以运行下Composer包更新命令，看看效果啦。

<!-- lang: shell -->

composer install

成功运行完毕，会在根目录下发现vendor文件夹，里面包含了刚刚我们列出来的两个包文件代码。

**require-dev**

有时候，我们会发现，有些包依赖只会在开发过程中使用，正式发布的程序不需要这些包，这个时候，就需要用到另外一个键，即require-dev。例如，我们想用codeception进行单元[**测试**](http://lib.csdn.net/base/softwaretest)，那么就可以通过require-dev引入这个开发环境下的依赖包：

<!-- lang: js -->

“require-dev”: {

"codeception/codeception": "2.0.0 "

}

加了这个依赖后，再运行下命令看看效果。

<!-- lang: shell -->

composer install

**自动加载**

自此，composer已经帮我们把需要的库文件下载下来啦，接下去想到的就是如何引用这些库文件。最简单的方式就是require或者include，但这就不够高大上了啊，需要花时间去库文件里查看需要引入哪些文件，费事而且容易出错。好在composer可以帮我们解决这个问题。那就是autoload。

在运行完composer install命令后，怎么调用PHPExcel库呢？很简单，只要引入vendor目录下的autoload.php文件就可以了。可以在根目录下，建一个index.php文件，加入一下内容：

<!-- lang: php -->

include “vendor/autoload.php”

$excel = new PHPExcel();

var\_dump($excel);

用浏览器访问一下这个页面，就会发现PHPExcel对象已经被成功创建啦，是不是很方便？

其实到目前为止，我们并没用在composer.json文件里加入autoload字段，那么什么时候需要加入呢？ 那就是当我们想让composer帮我们自动加载我们自己定义的类的时候。例如，我们自己写了个订单管理类，取名OrderManager，放在lib目录下的OrderManager.php文件里。内容如下：

<!-- lang: php -->

class OrderManager

{

public function test()

{

echo "hello";

}

}

那么如何让composer帮我们自动加载这个类呢？ 在composer.json里加入下面的内容：

<!-- lang: js -->

“autoload”:{

"files":["lib/OrderManager.php"]

}

files键对应的值是一个数组，数组元素是文件的路径，路径是相对于应用的根目录。加上上述内容后，运行命令：

<!-- lang: js -->

composer dump-autoload

让composer重建自动加载的信息，完成之后，就可以在index.php里调用OrderManager类啦。

通过文件引入的方法虽然直观，但是很费劲，每个文件都得引入一次，实在不是好的解决办法。有没有更好的办法呢？尝试将autoload的值改成：

<!-- lang: js -->

"classmap":["lib"]

再此运行composer dump-autoload，尝试调用，依然能够成功创建OrderManager类。其实，classmap通过建立类到文件的对应关系，当程序需要OrderManager类时，compoer的自动加载类通过查找OrderManager类所在的文件，然后再将改文件include进来。因此，这又导致了一个问题，那就是每加一个新类，就需要运行一次composer dump-autoload来创建类到文件到对应关系，比files方法虽然好一点，但是还是很不够舒爽啊！于是，PSR-0出场了。先了解下什么是PSR-0。

FIG组织制定的一组PHP相关规范，简称PSR，其中

PSR-0自动加载 PSR-1基本代码规范 PSR-2代码样式 PSR-3日志接口 PSR-4 自动加载

目前就这五个规范，乍一看，PSR-0和PSR-4是重复了，实际上，在功能上确实有所重复。区别在于PSR-4的规范比较干净，去除了兼容PHP 5.3以前版本的内容，有一点PSR-0升级版的感觉。当然，PSR-4也不是要完全替代PSR-0，而是在必要的时候补充PSR-0——当然，如果你愿意，PSR-4也可以替代PSR-0。PSR-4可以和包括PSR-0在内的其他自动加载机制共同使用。

PSR-0规范的具体内容见：<https://github.com/hfcorriez/fig-standards/blob/zh_CN/%E6%8E%A5%E5%8F%97/PSR-0.md> PSR-4规范的具体内容见：<https://github.com/hfcorriez/fig-standards/blob/zh_CN/%E6%8E%A5%E5%8F%97/PSR-4-autoloader.md>

简而言之，就是希望通过一组约定的目录，文件名，类名定义方式，来实现快速通过类查找到文件，然后包含进来，实现自动加载。 PSR-4和PSR-0最大的区别是对下划线（underscore）的定义不同。PSR-4中，在类名中使用下划线没有任何特殊含义。而PSR-0则规定类名中的下划线\_会被转化成目录分隔符。

不管是PSR-0还是PSR-4，都要求有个命名空间，所以我们需要对OrderManager类进行一些小的修改，加上命名空间：

<!-- lang: php -->

namespace SilkLib;

class OrderManager

{

public function test()

{

echo "hello";

}

}

同时，文件夹的结构也要修改成：应用根目录\lib\SilkLib\OrderManager.php

然后修改composer.json里的autoload部分如下：

<!-- lang: js -->

"autoload":{

"psr-0":{

"SilkLib":"lib/"

}

}

这里需要注意的是，SlikLib是命名空间，lib是目录名，他们的组合告诉composer，文件搜索是在：lib/SilkLib/ 目录下，而不是在 SilkLib/lib 目录下，这一点要特别注意，有点绕，容易弄错。

如果我们把命名空间改成 Slik\lib, 相应的目录结构要改成：应用根目录\lib\Silk\lib\OrderManager.php，autoload部分的写法相应的也要改成：

<!-- lang: js -->

"autoload":{

"psr-0":{

"Silk\\lib":"lib/"

}

}

注意Silk\lib是双斜杆。好了，那我们试试再加一个类，然后不用运行composer dump-autoload命令，看看新类是否能加载上。在lib目录下，新增一个ShipManager.php文件，内容如下：

<!-- lang: php -->

namespace Silk\lib;

class ShipManager

{

public function test()

{

echo 'hello ship class';

}

}

尝试在index.php文件中调用：

<!-- lang: php -->

$orderMgr = new Silk\lib\OrderManager();

$orderMgr->test();

$shipMgr = new Silk\lib\ShipManager();

$shipMgr->test();

运行成功，说明使用psr-0规范进行自动加载，比classmap更加方便。下面试试psr-4方式，整理下目录结构，改成：应用根目录\lib\OrderManager.php，修改命名空间为Silk, 修改autoload部分为：

<!-- lang: php -->

"autoload":{

"psr-4":{

"Silk":"lib"

}

}

尝试调用，发现报错Fatal error: Uncaught exception 'InvalidArgumentException' with message 'A non-empty PSR-4 prefix must end with a namespace separator. 提示要加上分隔符，那就加上吧：

<!-- lang: js -->

"autoload":{

"psr-4":{

"Silk\\":"lib"

}

}

再次composer dump-autoload，运行测试，OK通过！

掌握require和autoload部分，其实就算Compoer入门啦，在详细的内容，可以通过查看composer文档来了解。Happy Coding！