http://www.cnblogs.com/yue-blog/p/5901891.html

http://www.cnblogs.com/yue-blog/p/5904275.html

[**PHP中类自动加载的方式**](http://www.cnblogs.com/yue-blog/p/5901891.html)

　　最近在学习composer，发现从接触PHP到现在已经遇到了三种关于PHP中类的自动加载方式，这其中包括PHP自带的类的自动加载方式、PHP的第三方的依赖管理工具composer的加载方式以及PHP的Yaf框架下的自动加载方式。本篇博客主要是针对PHP5自带的加载方式进行详细介绍，composer和Yaf下类的自动加载将在接下来的时间里分两篇和大家一起学习。

**1.手动加载方式**

像C和C++等语言，在PHP中需要使用另一个文件中的相关的类、方法时，可以使用include、include\_once、require或者require\_once将所用的文件包含进工程里面。其中，四者的区别如下。

* include将套用一个文件，如果文件不存在，则给出一个提示，跳过继续执行；
* include\_once也是套用一个文件，但是只会套用一次，如果文件不存在，则继续执行；
* require表示套用一个文件，如果文件不存在，则中断程序的执行；
* require\_once也是套用一个文件，且只会套用一次，如果文件不存在，则中断程序的执行；

　　以上四种方式是需要什么文件的时候，手动在程序当中包含进文件。这在项目的规模比较小的时候，是可以的；但是随着项目规模的扩大，要通过手动的方式加载每个文件所需要的类简直是一场噩梦。

　　为了省事，在加载的时候可以通过set\_include\_path()设置加载的路径，同样也可以通过get\_include\_path()获取加载的路径。关于set\_include\_path()和get\_include\_path(),我也是刚刚接触，这里只对set\_include\_path()作简要的介绍，以后遇到问题再加以补充。

　　首先，set\_include\_path()是在脚本中动态的对php.ini中的include\_path进行动态的修改，而这个include\_path就是对include和require(下文中如果不进行特别的说明，include代表include和include\_once，require代表着require和require\_once)的路径进行设置，或者说是预定义。假如，我们在一个main.php文件中需要使用projname/home/lib/mylib/test文件夹下的a.php、b.php、c.php......，如果没有设置包含的路径的话，那么写成如下的形式:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | < ? php            include("projname/home/lib/mylib/test/a.php");          include("projname/home/lib/mylib/test/b.php");          include("projname/home/lib/mylib/test/c.php");        ...... |

　　这样，每个include都需要包含绝对路径，显得很麻烦。如果在需要被包含的文件之前加上set\_include\_path(“projname/home/lib/mylib/test”),那么就可以写成如下所示的形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | < ? php        set\_include\_path("projname/home/lib/mylib/test");      include("a.php");      include("b.php");      include("c.php");      ...... |

　　相比于第一种费时费力的写法，第二种明显省去了很多的时间，但是仍然是要将每个文件包含进来，只是简化了包含的路径而已。当然，上面所说的情况是所需要的文件都存在于一个文件夹中，如果文件存在于不同的文件夹中，那么可以添加多条的set\_include\_path()语句，此时如果include或者require中的文件包含的文件名在多个目录下出现，那么只会包含最先出现在set\_include\_path目录中的文件；如果所有的set\_include\_path指定的文件夹中都没有对应的文件，而文件名恰好出现在当前的文件夹中，则直接包含当前目录下的对应的文件。

　　get\_include\_path()函数只适用于获取当前的包含路径。

**2.\_autoload和spl\_autoload\_register()自动加载方式**

　　为了将双手从类的加载方式中解放出来，在PHP5及以后的版本中提供了一个自动加载的机制---autoload。Autoload可以使类在确实被需要的情况下才会被加载进来，也就是所谓的lazy loading,而不是一开始就include或者require所有的类文件。其中PHP提供的自动加载机制又分为两种---\_\_autoload()以及spl\_autoload\_register()。

**1). \_\_autoload机制**

在PHP5中运行程序的过程中，如果发现某一个类并没有被包含进来，那么就会运行\_\_autoload自动加载机制，将所需要的类加载进来。其写法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | < ? php        public function  \_\_autoload($classname) {          $fileName = $classname."php";          if (file\_exist($fileName)) {              require\_once("$fileName");          } else {              echo $fileName." doesn't exist!"          }      } |

　　根据这个程序写法，我们可以得到如下的结论：**保证自动加载机制的的原则就是要使得类名和文件名具有一种对应关系，**类名+后缀构成了这个类所在的文件的名字。如果这个文件确实存在，那么就根据$fileName将该类加载进来。如果文件不存在，则提示用户，文件不存在。总的来说自动加载机制包括三个步骤：

* 根据类名确定文件名，也就是确定一种类名和文件名之间的统一对应规则；
* 根据文件名在磁盘上找到相应的对应文件（例子中是最简单的情况，就是类与调用他们的PHP文件都在同一个目录下）；如果不在同一个目录下，那么可以使用set\_include\_path()指定要加载的路径；
* 将磁盘文件加载到文件系统中，这一步只是用一般的include和require包含相应的类文件；

　　\_\_autoload()实现类的自动加载的原则就是：类名和文件名之间具有一种统一的对应关系，这是在一个系统中实现\_\_autoload的关键所在。但是一个系统可能是有不同的人员所开发，如果在开发之前没有约定统一的标准，则可能存在不同的对应规则，导致需要在\_\_autoload()中实现多种加载规则，那么可能导致\_\_autoload()函数非常的臃肿。为了解决这个这个问题，PHP还提供了一个自动加载机制---spl\_autoload\_register().

**2). spl\_autoload\_register()机制**

　　SPL是Standard PHP Library（标准PHP库）的缩写，是PHP5引入的一个扩展库。SPL autoload是通过将函数指针autoload\_func指向自动装载函数实现的。SPL具有两个不同的自动装载函数，分别是spl\_autoload和spl\_autoload\_call，通过将autoload\_fun指向这两个不同的加载函数地址可以实现不同的自动加载机制。

* **spl\_autoload**

　　spl\_autoload是SPL实现的默认的自动加载函数，是一个可以接受两个参数的函数。其中第一个函数为$class\_name,表示要加载的类名；第二个参数是$file\_extension为可选参数，表示类文件的扩展名。$file\_extension中可以指定多个扩展名，扩展名之间用分号隔开即可，不指定扩展名，则使用默认的扩展名.inc或者.php。spl\_autoload首先将$class\_name变为小写，然后在所有的include\_path中搜索$ class\_name.inc或者$class\_name.php文件。如果找到对应的文件，就加载对应的类。其实可以手动的使用spl\_autoload("xxxx",".php")来实现xxxx类的加载。这其实和require/include差不多，但是，spl\_autoload相对来说灵活一点，因为可以指定多个扩展名。

　　前面说到，spl\_autoload\_register中包含的函数指针autoload\_func用于指定要使用的加载函数。那么，我们必须将对应的函数地址赋值给autoload\_func，spl\_autoload\_register()正实现了给函数指针autoload\_func赋值的功能。如果spl\_autoload\_register()函数中不含有任何的参数，则默认是将spl\_autoload()赋值给autoload\_func.

* **spl\_autoload\_call**

　　SPL模块的内部其实还存在着一个autoload\_functions,其本质上是一个哈希表，或者为了直观的理解，我们将其想像成一个容器，里面的各个元素都是指向加载函数的指针。spl\_autoload\_call的实现机制其实也比较简单，按照一定的顺序遍历这个容器，执行里面的函数指针指向的加载函数，每执行一个指针之后都会检查所需要的类是否已经完成加载。如果完成了加载，则退出。否则继续接着向下执行。如果执行完所有的加载函数之后，所需要的类仍然没有完成加载，则spl\_autoload\_call()直接退出。这也就是说即使使用了autoload机制，也不一定能够完成类的加载，其关键在于看你如何创建你的自动加载函数。

　　既然，存在一个autoload\_functions,那么如何将创建的自动加载函数添加到其中呢？spl\_autoload一样，同样使用spl\_autoload\_register()将加载函数注册到autoload\_functions中。当然可以通过spl\_autoload\_unregister()函数将已经注册的函数从autoload\_functions从哈希表中删除。这和前面所写过的工厂设计模式是一致的，详见：http://www.cnblogs.com/yue-blog/p/5771352.html。

　　这里需要说明的一点是spl\_autoload\_register实现自动加载的顺序。spl\_autoload的自动加载顺序为:首先判断autoload\_func是否为空，如果autoload\_func为空，则查看是否定义了\_\_autoload函数，如果没有定义，则返回，并报错；如果定义了\_\_autoload()函数，则返回加载的结果。如果autoload\_func不为空，直接执行autoload\_func指针指向的函数，不会检查\_\_autoload是否定义。**也就是说优先使用spl\_autoload\_register()注册过的函数。**

根据以上介绍，如果autoload\_func为非空是就不能自动执行\_\_autoload()函数了。如果想在使用spl\_autoload\_register()函数的情况下，依然可以使用\_\_autoload()函数，则可以将\_\_autoload函数通过spl\_autoload\_register()添加到哈希表中，即，spl\_autoload\_register(\_\_autoload())。下面的代码示例分别说明了如何注册普通的方法和类的静态公有方法。

　　普通函数的注册方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <? php        /\*\*      \* @ 普通函数的调用方法，可以调用后缀名分别为.php和.class.php的类文件      \*/      function loadFielEndOfPhp($classname) {          $fileName = $classname.".php";          if (file\_exist($fileName)) {              require\_once("$fileName");          } else {              echo $fileName." doesn't exist!"          }      }        function loadFielEndOfClassPhp($classname) {          $fileName = $classname.".class.php";          if (file\_exist($fileName)) {              require\_once("$fileName");          } else {              echo $fileName." doesn't exist!"          }      spl\_autoload\_register("loadFielEndOfPhp");      spl\_autoload\_register("loadFielEndOfClassPhp");    } |

　　类中静态的加载函数的注册方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <? php        /\*\*      \* @ 类中静态成员函数的调用方法，可调用后缀名为.php和.class.php的文件      \*/      class test {          public static function loadFielEndOfPhp($classname) {              $fileName = $classname.".php";              if (file\_exist($fileName)) {                  require\_once("$fileName");              }              else {                  echo $fileName." doesn't exist!"              }          }            public static function loadFielEndOfClassPhp($classname) {              $fileName = $classname.".class.php";              if (file\_exist($fileName)) {                  require\_once("$fileName");              }              else {                  echo $fileName." doesn't exist!"              }      }        spl\_autoload\_register(array("test","loadFielEndOfPhp"));      //spl\_autoload\_register("test::loadFielEndOfPhp");         //上一行的另一种写法，不是使用数组的形式完成注册；      spl\_autoload\_register(array("test","loadFielEndOfClassPhp"));      //spl\_autoload\_register("test::loadFielEndOfClassPhp");    //第三行的另一种写法，不是使用数组的形式完成注册；    } |

[**Composer实现PHP中类的自动加载**](http://www.cnblogs.com/yue-blog/p/5904275.html)

　　本篇博客承接上一篇，学习一下Composer实现的PHP的类的自动加载方式。首先说明一下，Composer是PHP针对PHP语言的第三方的依赖管理工具，将工程所用到的依赖文件包含在composer.json文件中，使用composer install命令就可以将所使用对应库或者文件加载进工程里面。下面分两部分介绍composer的基础，分别是composer的依赖管理和自动加载。

**依赖管理**

　　在composer出现之前，如果咋PHP项目中需要第三方的依赖文件，则需要程序员将所需要的源代码拷贝进工程中或者将源代码对应的文件下载下来手动添加到工程中。如果所需要的依赖文件依赖于更多的第三方文件，则程序员会陷入拷贝依赖文件的黑洞中，费时费力不说，有可能还会出现一些失误，composer就是在这种情况下出现的用于减轻程序员的对于依赖管理的负担的工具。

　　Composer通过使用配置文件composer.json文件完成依赖管理。composer.json文件包含了项目的简单介绍、项目对于外界的库或文件的依赖。从composer.json的扩展名就可以看得出来，composer.json中的内容是按照json标准组织的。本篇博客集中在类的自动加载机制上，因此对于composer.json中关于项目的作者等相关信息的解释忽略不解释。

　　在composer中将本项目所需要的外部依赖包写在关键字require对应的值中，require键可以对应着多个所需要的包，各个不同的包之间用逗号分隔。假设项目需要一个外部依赖包monolog，下面是在compose.json文件中对于monolog包的依赖配置项：

|  |  |
| --- | --- |
| 1  2  3  4  5 | {      "require":{          "monolog/monolog": "1.0.\* @dev"      }  } |

　　如上所示，require关键字将会映射包的名称monolog和包的版本1.0.\*。其中，包的名称有两部分组成，中间以“/”分隔，“/”之前代表的包的所有者，在Github上一般是代表的Github的用户名，“/”代表的是实际的包的名称。其中“/”之前的名称必须是唯一的，但是"/"之后的包的名称是可以存在重复的，例如“Jack/monolog”和“monolog/monolog”是可以共存的。而后面的"1.0.\*"代表的是所依赖的包的版本，其中“\*”代表的是“1.0”之下的任意一个版本，例如：1.0.1,1.0.2或者1.0.9等等。“@dev”代表了可以获取该包的开发版本。默认情况下，composer将获取所需要的包的最新的稳定版本，而不会考虑开发版本，因为开发版本一般是不太稳定的版本。但是，如果确定开发版本没有什么问题就可以加上“@dev”以允许获取开发的版本。如果没有加上“@dev”,而除了开发版本之外不存在别的版本的话，则composer加载依赖项出错。

　　在composer中配置好依赖的第三方的包之后，就可以使用composer install命令获取第三方的包了。运行成功之后就可以项目中就会出现vendor文件夹，vendor文件夹中会包含我们在require中所列出的monolog文件。

　　出了生成列在composer.json中关键字require值下的文件之外，成功运行composer install之后还会生成对应的composer.lock文件，该文件依据composer.json中的依赖项生成依赖包对应的版本。这里需要说明的是，在我们运行composer install的时候会首先判断是否存在着composer.lock文件。如果原本就存在这composer.lock文件，那么就会直接根据composer.lock中的版本下载对应的包，此时不再理会composer.json中的配置项。如果不存在composer.lock文件，则根据composer.json文件中的配置下载对应的版本，并生成composer.json对应的composer.lock文件。composer.lock又称为锁定文件，生成对应的composer.lock之后，composer.json和composer.lock共同对版本进行控制。有了composer.lock之后，及时有了新的版本也不会触发新版本的更新，除非手动使用composer update命令进行手动的更新。

**开发环境下的依赖**

有的时候我们只是在开发的时候才会依赖于某个具体的包，但是在发行的版本中并不需要这样的包，为了达到这种要求可以使用require-dev引入开发环境下的包依赖。如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | {      "require-dev":{          "xxxx/xxxx": "x.x.\* @dev"      }  } |

**自动加载类**

　　通过composer.json或者composer.lock，所依赖的第三方库已经被下载下来了，那么在我们的项目中怎么使用这些第三方库呢？最简单的方式就是通过include或者require将所需要的类文件包含进来，但是这种方式需要我们自己去寻找所使用的类对应的类文件，这就存在和PHP语言中直接使用include和require加载所需要的类存在同样的弊端，一种比较简单的方式当然就是使用composer提供的类的自动加载机制了。类似于PHP的自动加载机制，composer提供了autoload实现类的自动加载。

　　成功运行composer install之后，只要调用生成的vendor目录下的autoload.php文件就可以调用通过composer.json加载的类了。以上面所述的项目中需要monolog包为例，通过可以通过下面的方式使用monolog包中的Logger类。

|  |  |
| --- | --- |
| 1  2 | include "vendor/autoload.php";  $log = new Monolog\Logger("name"); |

　　当然，除了使用第三方库中提供的类之外，还可以使用自己的定义的类。Composer提供了autoload关键字用于加载我们自己提供的类，假如我们定义了一个有关测试的类，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ClassTest{        public function test()      {          echo "hello world!";      }  } |

　　将该类放在lib目录下的ClassTest.php文件夹下面，那么怎么让composer加载自己定义的类呢？

**1. 在composer.json中加入autoload关键字**

|  |  |
| --- | --- |
| 1  2  3 | "autoload":{      "files":["lib/ClassTest.php"];  } |

　　files键对应的值是一个数组且改值是相对于文件应用根目录的文件的路径。在composer.json中加入上述的关键字之后，在命令行下运行composer dump-autoload就可以让composer重建加载信息，那么就可以在其他的文件中使用这个类了。

　　上述所述的方法和PHP中直接使用include和require存在一样的弊端，每个类都需要重新书写加载文件，费时费力。

**2. composer.json中加入classmap关键字**

　　相比于每个类文件都需要加载一次的做法，使用classmap关键字，能够减少程序员的负担，只需要将文件所在的目录添加在classmap的值中即可，如下：

|  |  |
| --- | --- |
| 1 | "classmap" : ["lib"] |

　　其实这需要建立一种类名到类所在的文件的映射关系。当需要相应的类的时候，composer通过类名找到对应的类文件名，将相应的类include进来。但是这同样存在一个问题就是，每增加一个类都需要重新运行一次composer dump-autoload重新创建类名到文件之间的映射关系，从而将对应的类加载进来。虽然比files关键字节省了功夫，但是依然不能完全自动加载所需要的类。

**3. 基于PHP规范的自动加载方式**

针对PHP这种编程语言，到目前FIG指定了五个规范，分别如下：

* PSR0：自动加载；
* PSR1:基本代码规范；
* PSR2：代码样式规范；
* PSR3：日志接口规范；
* PSR4：自动加载规范；

　　看上去PSR4与PSR0是重复了，但是PSR4规范比较干净，可以看成PSR0规范的升级版。二者最重要区别在于：PSR0规范中，下划线会被转换为目录分隔符，但是PSR4中下划线不具有特殊的含义。二者都是通过特定的目录、文件名以及类名，实现快速查找到类文件，并将相应的类加载进来。

　　PSR0和PSR4要求有个命名的空间，对上述的ClassTest类做相应的修改如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?php  namespace ClassTestLib;  class ClassTest{        public function test()      {          echo "hello world!";      }  } |

　　那么对应的文件的路径应该改为\lib\ClassTestLib\ClassTest.php，此时修改composer.json中的autoload如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | "autoload":{      "psr-0":{          "ClassTestLib":"lib/"      }  } |

　　可能你发现psr0的值有一些奇怪，是的。在这里ClassTestLib代表的是命名空间，而"lib"是目录名。加载对应的类文件的时候，搜索的路径是lib/ClassTestLib,而不是ClassTestLib/lib,这是在书写composer.json的时候需要注意的一点。

　　如果命名空间中存在着“\”,则在书写对应的composer.json的时候需要在相应的“\”再添加一个“\”。例如，如果命名空间改为ClassTest\Lib，相应的对应与应用根目录的路径名称应该变为\lib\ClassTest\Lib\ClassTest.php，对应的composer.json中的autoload应该变为：

|  |  |
| --- | --- |
| 1  2  3  4  5 | "autoload":{      "psr-0":{          "ClassTest\\Lib":"lib/"      }  } |

　　小结：以上是关于composer的基本使用方法，后续在遇到相关的问题，和大家在一起学习。