JeffreyC RP 1.7k 发布于 一步一个脚印
2018-05-21 发布

深入解析 composer 的自动加载原理

深入解析 composer 的自动加载原理

前言

PHP 自5.3的版本之后，已经重焕新生，命名空间、性状（trait）、闭包、接口、PSR 规范、以及 composer 的出现已经让 PHP 变成了一门现代化的脚本语言。PHP 的生态系统也一直在演进，而 composer 的出现更是彻底的改变了以往构建 PHP 应用的方式，我们可以根据 PHP 的应用需求混合搭配最合适的 PHP 组件。当然这也得益于 PSR 规范的提出。

大纲

- PHP 自动加载功能
- PSR 规范
- comoposer 的自动加载过程
- composer 源码分析

一、PHP 自动加载功能

PHP 自动加载功能的由来

在 PHP 开发过程中，如果希望从外部引入一个 Class ，通常会使用 `include` 和 `require` 方法，去把定义这个 Class 的文件包含进来。这个在小规模开发的时候，没什么大问题。但在大型的开发项目中，

使用其它的类，那么要保证每个文件都包含正确的类文件肯定是一个噩梦， 况且 require或 incloud 的性能代价很大。

PHP5 为这个问题提供了一个解决方案，这就是 `类的自动加载(autoload)机制` 。 `autoload机制` 可以使得 PHP 程序有可能在使用类时才自动包含类文件，而不是一开始就将所有的类文件 `include` 进来，这种机制也称为 `Lazy loading (惰性加载)` 。

- 总结起来，自动加载功能带来了几处优点：

> 1. 使用类之前无需 `include / require`
> 2. 使用类的时候才会 `include / require` 文件，实现了 `lazy loading` ，避免了 `include / require` 多余文件。
> 3. 无需考虑引入 **类的实际磁盘地址** ，实现了逻辑和实体文件的分离。

PHP 自动加载函数 __autoload()

- 从 PHP5 开始，当我们在使用一个类时，如果发现这个类没有加载，就会自动运行 __autoload() 函数，这个函数是我们在程序中自定义的，在这个函数中我们可以加载需要使用的类。下面是个简单的示例：

```
<?php

function __autoload($classname) {
        require_once ($classname . ".class.php");
}
```

- 在我们这个简单的例子中，我们直接将类名加上扩展名 `.class.php` 构成了类文件名，然后使用 `require_once` 将其加载。

> 从这个例子中，我们可以看出 __autoload 至少要做三件事情：
> 1. 根据类名确定类文件名；
> 2. 确定类文件所在的磁盘路径；
> 3. 将类从磁盘文件中加载到系统中。

- 第三步最简单，只需要使用 `include / require` 即可。要实现第一步，第二步的功能，必须在开发时约定类名与磁盘文件的映射方法，只有这样我们才能根据类名找到它对应的磁盘文件。

- 当有大量的类文件要包含的时候，我们只要确定相应的规则，然后在 `__autoload()` **函数中，将类名与实际的磁盘文件对应起来，就可以实现** `lazy loading` **的效果**。
- 如果想详细的了解关于 autoload 自动加载的过程，可以查看手册资料：PHP autoload函数说明

__autoload() 函数存在的问题

- 如果在一个系统的实现中，如果需要使用很多其它的类库，这些类库可能是由不同的开发人员编写的， 其类名与实际的磁盘文件的映射规则不尽相同。这时如果要实现类库文件的自动加载，就必须 **在 __autoload() 函数中将所有的映射规则全部实现**，这样的话 `__autoload()` 函数有可能会非常复杂，甚至无法实现。最后可能会导致 `__autoload()` 函数十分臃肿，这时即便能够实现，也会给将来的维护和系统效率带来很大的负面影响。
- 那么问题出现在哪里呢？问题出现在 **__autoload() 是全局函数只能定义一次** ，不够灵活，所以所有的类名与文件名对应的逻辑规则都要在一个函数里面实现，造成这个函数的臃肿。那么如何来解决这个问题呢？答案就是使用一个 **__autoload调用堆栈** ，不同的映射关系写到不同的 `__autoload函数` 中去，然后统一注册统一管理，这个就是 PHP5 引入的 `SPL Autoload` 。

SPL Autoload

- SPL是 Standard PHP Library(标准PHP库)的缩写。它是 PHP5 引入的一个扩展标准库，包括 spl autoload 相关的函数以及各种数据结构和迭代器的接口或类。spl autoload 相关的函数具体可见 php中spl_autoload

```
<?php

// __autoload 函数
//
// function __autoload($class) {
//     include 'classes/' . $class . '.class.php';
// }


function my_autoloader($class) {
    include 'classes/' . $class . '.class.php';
}

spl_autoload_register('my_autoloader');


// 定义的 autoload 函数在 class 里
```

```
  public static function autoload($className) {
    // ...
  }
}
```

spl_autoload_register() 就是我们上面所说的__autoload调用堆栈，我们可以向这个函数注册多个我们自己的 autoload() 函数，当 PHP 找不到类名时，PHP就会调用这个堆栈，然后去调用自定义的 autoload() 函数，实现自动加载功能。如果我们不向这个函数输入任何参数，那么就会默认注册 spl_autoload() 函数。

二、PSR 规范

与自动加载相关的规范是 PSR4，在说 PSR4 之前先介绍一下 PSR 标准。PSR 标准的发明和推出组织是：PHP-FIG，它的网站是：www.php-fig.org。由几位开源框架的开发者成立于 2009 年，从那开始也选取了很多其他成员进来，虽然不是 “官方” 组织，但也代表了社区中不小的一块。组织的目的在于：以最低程度的限制，来统一各个项目的编码规范，避免各家自行发展的风格阻碍了程序员开发的困扰，于是大伙发明和总结了 PSR，PSR 是 PHP Standards Recommendation 的缩写，截止到目前为止，总共有 14 套 PSR 规范，其中有 7 套PSR规范已通过表决并推出使用，分别是：

> PSR-0 **自动加载标准**（已废弃，一些旧的第三方库还有在使用）
>
> PSR-1 **基础编码标准**
>
> PSR-2 **编码风格向导**
>
> PSR-3 **日志接口**
>
> PSR-4 **自动加载的增强版，替换掉了 PSR-0**
>
> PSR-6 **缓存接口规范**
>
> PSR-7 **HTTP 消息接口规范**

具体详细的规范标准可以查看PHP 标准规范

PSR4 标准

PSR-4 规范了如何指定文件路径从而自动加载类定义，同时规范了自动加载文件的位置。

1）一个完整的类名需具有以下结构：

\<命名空间>\<子命名空间>\<类名>

- 完整的类名**必须**要有一个顶级命名空间，被称为 "vendor namespace"；
- 完整的类名**可以**有一个或多个子命名空间；
- 完整的类名**必须**有一个最终的类名；
- 完整的类名中**任意一部分**中的下滑线都是没有特殊含义的；
- 完整的类名**可以**由任意大小写字母组成；
- 所有类名都**必须**是大小写敏感的。

2）根据完整的类名载入相应的文件

- 完整的类名中，去掉最前面的命名空间分隔符，前面连续的一个或多个命名空间和子命名空间，作为「命名空间前缀」，其必须与至少一个「文件基目录」相对应；
- 紧接命名空间前缀后的子命名空间 必须 与相应的「文件基目录」相匹配，其中的命名空间分隔符将作为目录分隔符。
- 末尾的类名**必须**与对应的以 .php 为后缀的文件同名。
- 自动加载器（autoloader）的实现**一定不可**抛出异常、**一定不可**触发任一级别的错误信息以及**不应该**有返回值。

3) 例子

PSR-4风格

> 类名：ZendAbc
> 命名空间前缀：Zend
> 文件基目录：/usr/includes/Zend/
> 文件路径：/usr/includes/Zend/Abc.php

> 类名：SymfonyCoreRequest
> 命名空间前缀：SymfonyCore
> 文件基目录：./vendor/Symfony/Core/
> 文件路径：./vendor/Symfony/Core/Request.php

目录结构

```
-vendor/
| -vendor_name/
| | -package_name/
| | | -src/
| | | | -ClassName.php       # Vendor_Name\Package_Name\ClassName
| | | -tests/
| | | | -ClassNameTest.php   # Vendor_Name\Package_Name\ClassNameTest
```

Composer自动加载过程

Composer 做了哪些事情

- 你有一个项目依赖于若干个库。
- 其中一些库依赖于其他库。
- 你声明你所依赖的东西。
- Composer 会找出哪个版本的包需要安装，并安装它们（将它们下载到你的项目中）。

例如，你正在创建一个项目，需要做一些单元测试。你决定使用 `phpunit` 。为了将它添加到你的项目中，你所需要做的就是在 `composer.json` 文件里描述项目的依赖关系。

```
{
  "require": {
    "phpunit/phpunit":"~6.0",
  }
}
```

然后在 `composer require` 之后我们只要在项目里面直接 `use` phpunit 的类即可使用。

执行 composer require 时发生了什么

- composer 会找到符合 PR4 规范的第三方库的源
- 将其加载到 vendor 目录下
- 初始化顶级域名的映射并写入到指定的文件里

（如：`'PHPUnit\\Framework\\Assert' => __DIR__ . '/..' . '/phpunit/phpunit/src/Framework/Assert.php',`）

- 写好一个 autoload 函数，并且注册到 spl_autoload_register()里

题外话：现在很多框架都已经帮我们写好了顶级域名映射了，我们只需要在框架里面新建文件，在新建的文件中写好命名空间，就可以在任何地方 use 我们的命名空间了。

Composer 源码分析

下面我们通过对源码的分析来看看 composer 是如何实现 `PSR4标准` 的自动加载功能。

很多框架在初始化的时候都会引入 composer 来协助自动加载的，以 Laravel 为例，它入口文件 index.php 第一句就是利用 composer 来实现自动加载功能。

启动

```
<?php
  define('LARAVEL_START', microtime(true));

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

去 vendor 目录下的 `autoload.php` ：

```
<?php
  require_once __DIR__ . '/composer' . '/autoload_real.php';

  return ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29::getLoader();
```

这里就是 Composer 真正开始的地方了

Composer自动加载文件

首先，我们先大致了解一下Composer自动加载所用到的源文件。

> 1. autoload_real.php: 自动加载功能的引导类。
>   - composer 加载类的初始化 (顶级命名空间与文件路径映射初始化) 和注册
>   (spl_autoload_register())

- composer 自动加载功能的核心类。

3. autoload_static.php : 顶级命名空间初始化类，
    - 用于给核心类初始化顶级命名空间。
4. autoload_classmap.php : 自动加载的最简单形式，
    - 有完整的命名空间和文件目录的映射；
5. autoload_files.php : 用于加载全局函数的文件，
    - 存放各个全局函数所在的文件路径名；
6. autoload_namespaces.php : 符合 PSR0 标准的自动加载文件，
    - 存放着顶级命名空间与文件的映射；
7. autoload_psr4.php : 符合 PSR4 标准的自动加载文件，
    - 存放着顶级命名空间与文件的映射；

autoload_real 引导类

在 vendor 目录下的 `autoload.php` 文件中我们可以看出，程序主要调用了引导类的静态方法 `getLoader()` ，我们接着看看这个函数。

```
<?php
    public static function getLoader()
    {
      if (null !== self::$loader) {
          return self::$loader;
      }

      spl_autoload_register(
        array('ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29', 'loadClassLoade
      );

      self::$loader = $loader = new \Composer\Autoload\ClassLoader();

      spl_autoload_unregister(
        array('ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29', 'loadClassLoade
      );

      $useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HHVM_VERSION');

      if ($useStaticLoader) {
          require_once __DIR__ . '/autoload_static.php';
```

我把自动加载引导类分为 5 个部分。

第一部分——单例

第一部分很简单，就是个最经典的单例模式，自动加载类只能有一个。

```
<?php
  if (null !== self::$loader) {
      return self::$loader;
  }
```

第二部分——构造ClassLoader核心类

第二部分 new 一个自动加载的核心类对象。

```
<?php
  /**********************获得自动加载核心类对象********************/
  spl_autoload_register(
    array('ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29', 'loadClassLoader'), tr
  );

  self::$loader = $loader = new \Composer\Autoload\ClassLoader();

  spl_autoload_unregister(
    array('ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29', 'loadClassLoader')
  );
```

`loadClassLoader()` 函数：

```
<?php
public static function loadClassLoader($class)
{
    if ('Composer\Autoload\ClassLoader' === $class) {
        require __DIR__ . '/ClassLoader.php';
    }
}
```

从程序里面我们可以看出，composer 先向 PHP 自动加载机制注册了一个函数，这个函数 require 了 ClassLoader 文件。成功 new 出该文件中核心类 ClassLoader() 后，又销毁了该函数。

```php
<?php
  /**********************初始化自动加载核心类对象*******************/
  $useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HHVM_VERSION');
  if ($useStaticLoader) {
     require_once __DIR__ . '/autoload_static.php';

     call_user_func(
       \Composer\Autoload\ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::getInitia
     );
  } else {
      $map = require __DIR__ . '/autoload_namespaces.php';
      foreach ($map as $namespace => $path) {
         $loader->set($namespace, $path);
      }

      $map = require __DIR__ . '/autoload_psr4.php';
      foreach ($map as $namespace => $path) {
         $loader->setPsr4($namespace, $path);
      }

      $classMap = require __DIR__ . '/autoload_classmap.php';
      if ($classMap) {
          $loader->addClassMap($classMap);
```

这一部分就是对自动加载类的初始化，主要是给自动加载核心类初始化顶级命名空间映射。

初始化的方法有两种：

1. 使用 autoload_static 进行静态初始化；
2. 调用核心类接口初始化。

autoload_static 静态初始化（PHP >= 5.6）

静态初始化只支持 PHP5.6 以上版本并且不支持 HHVM 虚拟机。我们深入 `autoload_static.php` 这个文件发现这个文件定义了一个用于静态初始化的类，名字叫 `ComposerStaticInit7b790917ce8899df9af8ed53631a1c29` ，仍然为了避免冲突而加了 hash 值。这个类很简单：

```php
<?php
  class ComposerStaticInit7b790917ce8899df9af8ed53631a1c29{
     public static $files = array(...);
     public static $prefixLengthsPsr4 = array(...);
```

```
    public static $classMap = array (...);

    public static function getInitializer(ClassLoader $loader)
    {
      return \Closure::bind(function () use ($loader) {
          $loader->prefixLengthsPsr4
                        = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$prefix

          $loader->prefixDirsPsr4
                        = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$prefixD

          $loader->prefixesPsr0
                        = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$prefixe

          $loader->classMap
                        = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$classMa
```

这个静态初始化类的核心就是 `getInitializer()` 函数，它将自己类中的顶级命名空间映射给了 ClassLoader 类。值得注意的是这个函数返回的是一个匿名函数，为什么呢？原因就是 `ClassLoader类` 中的 `prefixLengthsPsr4` 、 `prefixDirsPsr4` 等等变量都是 private的。利用匿名函数的绑定功能就可以将这些 private 变量赋给 ClassLoader 类 里的成员变量。

关于匿名函数的绑定功能。

接下来就是命名空间初始化的关键了。

classMap（命名空间映射）

```
<?php
  public static $classMap = array (
      'App\\Console\\Kernel'
             => __DIR__ . '/../..' . '/app/Console/Kernel.php',

      'App\\Exceptions\\Handler'
             => __DIR__ . '/../..' . '/app/Exceptions/Handler.php',

      'App\\Http\\Controllers\\Auth\\ForgotPasswordController'
             => __DIR__ . '/../..' . '/app/Http/Controllers/Auth/ForgotPasswordControlle

      'App\\Http\\Controllers\\Auth\\LoginController'
             => __DIR__ . '/../..' . '/app/Http/Controllers/Auth/LoginController.php',

      'App\\Http\\Controllers\\Auth\\RegisterController'
             => __DIR__ . '/../..' . '/app/Http/Controllers/Auth/RegisterController.php',
  ...)
```

直接命名空间全名与目录的映射，简单粗暴，也导致这个数组相当的大。

PSR4 标准顶级命名空间映射数组：

```
<?php
  public static $prefixLengthsPsr4 = array(
      'p' => array (
        'phpDocumentor\\Reflection\\' => 25,
    ),
      'S' => array (
        'Symfony\\Polyfill\\Mbstring\\' => 26,
        'Symfony\\Component\\Yaml\\' => 23,
        'Symfony\\Component\\VarDumper\\' => 28,
        ...
    ),
  ...);

  public static $prefixDirsPsr4 = array (
      'phpDocumentor\\Reflection\\' => array (
        0 => __DIR__ . '/..' . '/phpdocumentor/reflection-common/src',
        1 => __DIR__ . '/..' . '/phpdocumentor/type-resolver/src',
        2 => __DIR__ . '/..' . '/phpdocumentor/reflection-docblock/src',
    ),
       'Symfony\\Polyfill\\Mbstring\\' => array (
        0 => __DIR__ . '/..' . '/symfony/polyfill-mbstring',
    ),
      'Symfony\\Component\\Yaml\\' => array (
        0 =>   DIR   . '/..' . '/symfony/yaml',
```

PSR4 标准顶级命名空间映射用了两个数组，第一个是用命名空间第一个字母作为前缀索引，然后是 顶级命名空间，但是最终并不是文件路径，而是 顶级命名空间的长度。为什么呢？

因为 PSR4 标准是用顶级命名空间目录替换顶级命名空间，所以获得顶级命名空间的长度很重要。

具体说明这些数组的作用：

假如我们找 `Symfony\Polyfill\Mbstring\example` 这个命名空间，通过前缀索引和字符串匹配我们得到了

```
<?php
    'Symfony\\Polyfill\\Mbstring\\' => 26,
```

这条记录，键是顶级命名空间，值是命名空间的长度。拿到顶级命名空间后去 `$prefixDirsPsr4`数组获取它的映射目录数组：**(注意映射目录可能不止一条)**

```
<?php
  'Symfony\\Polyfill\\Mbstring\\' => array (
              0 => __DIR__ . '/..' . '/symfony/polyfill-mbstring',
          )
```

然后我们就可以将命名空间 `Symfony\\Polyfill\\Mbstring\\example` 前26个字符替换成目录 `__DIR__ . '/..' . '/symfony/polyfill-mbstring` ，我们就得到了 `__DIR__ . '/..' . '/symfony/polyfill-mbstring/example.php` ，先验证磁盘上这个文件是否存在，如果不存在接着遍历。如果遍历后没有找到，则加载失败。

ClassLoader 接口初始化（ PHP < 5.6 ）

如果PHP版本低于 5.6 或者使用 HHVM 虚拟机环境，那么就要使用核心类的接口进行初始化。

```
<?php
    // PSR0 标准
    $map = require __DIR__ . '/autoload_namespaces.php';
    foreach ($map as $namespace => $path) {
       $loader->set($namespace, $path);
    }

    // PSR4 标准
    $map = require __DIR__ . '/autoload_psr4.php';
    foreach ($map as $namespace => $path) {
       $loader->setPsr4($namespace, $path);
    }

    $classMap = require __DIR__ . '/autoload_classmap.php';
    if ($classMap) {
       $loader->addClassMap($classMap);
    }
```

PSR4 标准的映射

autoload_psr4.php 的顶级命名空间映射

```
<?php
    return array(
    'XdgBaseDir\\'
        => array($vendorDir . '/dnoegel/php-xdg-base-dir/src'),
```

```
    'TijsVerkoyen\\CssToInlineStyles\\'
        => array($vendorDir . '/tijsverkoyen/css-to-inline-styles/src'),

    'Tests\\'
        => array($baseDir . '/tests'),

    'Symfony\\Polyfill\\Mbstring\\'
        => array($vendorDir . '/symfony/polyfill-mbstring'),
    ...
    )
```

PSR4 标准的初始化接口:

```
<?php
    public function setPsr4($prefix, $paths)
    {
        if (!$prefix) {
            $this->fallbackDirsPsr4 = (array) $paths;
        } else {
            $length = strlen($prefix);
            if ('\\' !== $prefix[$length - 1]) {
                throw new \InvalidArgumentException(
                  "A non-empty PSR-4 prefix must end with a namespace separator."
                );
            }
            $this->prefixLengthsPsr4[$prefix[0]][$prefix] = $length;
            $this->prefixDirsPsr4[$prefix] = (array) $paths;
        }
    }
```

总结下上面的顶级命名空间映射过程：

```
( 前缀 -> 顶级命名空间，顶级命名空间 -> 顶级命名空间长度 )
( 顶级命名空间 -> 目录 )
```

这两个映射数组。具体形式也可以查看下面的 `autoload_static` 的 $prefixLengthsPsr4 、$prefixDirsPsr4 。

命名空间映射

autoload_classmap：

```
    'App\\Console\\Kernel'
        => __DIR__ . '/../..' . '/app/Console/Kernel.php',

    'App\\Exceptions\\Handler'
        => __DIR__ . '/../..' . '/app/Exceptions/Handler.php',
    ...
)
```

addClassMap:

```
<?php
    public function addClassMap(array $classMap)
    {
        if ($this->classMap) {
            $this->classMap = array_merge($this->classMap, $classMap);
        } else {
            $this->classMap = $classMap;
        }
    }
```

自动加载核心类 ClassLoader 的静态初始化到这里就完成了！

其实说是5部分，真正重要的就两部分——初始化与注册。初始化负责顶层命名空间的目录映射，注册负责实现顶层以下的命名空间映射规则。

第四部分 —— 注册

讲完了 Composer 自动加载功能的启动与初始化，经过启动与初始化，自动加载核心类对象已经获得了顶级命名空间与相应目录的映射，也就是说，如果有命名空间 'App\Console\Kernel，我们已经可以找到它对应的类文件所在位置。那么，它是什么时候被触发去找的呢？

这就是 composer 自动加载的核心了，我们先回顾一下自动加载引导类：

```
 public static function getLoader()
 {
    /**************************经典单例模式*******************/
    if (null !== self::$loader) {
        return self::$loader;
    }

    /**********************获得自动加载核心类对象*******************/
    spl_autoload_register(array('ComposerAutoloaderInit
```

```
    self::$loader = $loader = new \Composer\Autoload\ClassLoader();

    spl_autoload_unregister(array('ComposerAutoloaderInit
    7b790917ce8899df9af8ed53631a1c29', 'loadClassLoader'));

    /************************初始化自动加载核心类对象********************/
    $useStaticLoader = PHP_VERSION_ID >= 50600 &&
    !defined('HHVM_VERSION');

    if ($useStaticLoader) {
        require_once __DIR__ . '/autoload_static.php';
```

现在我们开始引导类的第四部分：注册自动加载核心类对象。我们来看看核心类的 register() 函数：

```
public function register($prepend = false)
{
    spl_autoload_register(array($this, 'loadClass'), true, $prepend);
}
```

其实奥秘都在自动加载核心类 ClassLoader 的 loadClass() 函数上：

```
public function loadClass($class)
    {
        if ($file = $this->findFile($class)) {
            includeFile($file);

            return true;
        }
    }
```

这个函数负责按照 PSR 标准将顶层命名空间以下的内容转为对应的目录，也就是上面所说的将 'App\Console\Kernel 中' Console\Kernel 这一段转为目录，至于怎么转的在下面 “运行” 的部分讲。核心类 ClassLoader 将 loadClass() 函数注册到PHP SPL中的 spl_autoload_register() 里面去。这样，每当PHP遇到一个不认识的命名空间的时候，PHP会自动调用注册到 spl_autoload_register 里面的 loadClass() 函数，然后找到命名空间对应的文件。

全局函数的自动加载

Composer 不止可以自动加载命名空间，还可以加载全局函数。怎么实现的呢？把全局函数写到特定的文件里面去，在程序运行前挨个 require就行了。这个就是 composer 自动加载的第五步，加载全局函

```
if ($useStaticLoader) {
    $includeFiles = Composer\Autoload\ComposerStaticInit7b790917ce8899df9af8ed53631a1c29:
} else {
    $includeFiles = require __DIR__ . '/autoload_files.php';
}
foreach ($includeFiles as $fileIdentifier => $file) {
    composerRequire7b790917ce8899df9af8ed53631a1c29($fileIdentifier, $file);
}
```

跟核心类的初始化一样，全局函数自动加载也分为两种：静态初始化和普通初始化，静态加载只支持PHP5.6以上并且不支持HHVM。

静态初始化：

```
ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$files:
```

```
public static $files = array (
'0e6d7bf4a5811bfa5cf40c5ccd6fae6a' => __DIR__ . '/..' . '/symfony/polyfill-mbstring/bootst
'667aeda72477189d0494fecd327c3641' => __DIR__ . '/..' . '/symfony/var-dumper/Resources/fu
...
);
```

普通初始化

autoload_files:

```
$vendorDir = dirname(dirname(__FILE__));
$baseDir = dirname($vendorDir);

return array(
'0e6d7bf4a5811bfa5cf40c5ccd6fae6a' => $vendorDir . '/symfony/polyfill-mbstring/bootstrap.
'667aeda72477189d0494fecd327c3641' => $vendorDir . '/symfony/var-dumper/Resources/functio
    ....
);
```

其实跟静态初始化区别不大。

加载全局函数

```
class ComposerAutoloaderInit7b790917ce8899df9af8ed53631a1c29{
  public static function getLoader(){
      ...
      foreach ($includeFiles as $fileIdentifier => $file) {
        composerRequire7b790917ce8899df9af8ed53631a1c29($fileIdentifier, $file);
      }
      ...
  }
}

function composerRequire7b790917ce8899df9af8ed53631a1c29($fileIdentifier, $file)
 {
    if (empty(\$GLOBALS['__composer_autoload_files'][\$fileIdentifier])) {
        require $file;

        $GLOBALS['__composer_autoload_files'][$fileIdentifier] = true;
    }
}
```

第五部分 —— 运行

到这里，终于来到了核心的核心—— composer 自动加载的真相，命名空间如何通过 composer 转为对应目录文件的奥秘就在这一章。
前面说过，ClassLoader 的 register() 函数将 loadClass() 函数注册到 PHP 的 SPL 函数堆栈中，每当 PHP 遇到不认识的命名空间时就会调用函数堆栈的每个函数，直到加载命名空间成功。所以 loadClass() 函数就是自动加载的关键了。

看下 loadClass() 函数:

```
public function loadClass($class)
{
    if ($file = $this->findFile($class)) {
        includeFile($file);

        return true;
    }
}

public function findFile($class)
{
    // work around for PHP 5.3.0 - 5.3.2 https://bugs.php.net/50731
    if ('\\' == $class[0]) {
        $class = substr($class, 1);
    }
```

```
            return $this->classMap[$class];
        }
        if ($this->classMapAuthoritative) {
            return false;
        }
```

我们看到 loadClass() ，主要调用 findFile() 函数。findFile() 在解析命名空间的时候主要分为两部分：classMap 和 findFileWithExtension() 函数。classMap 很简单，直接看命名空间是否在映射数组中即可。麻烦的是 findFileWithExtension() 函数，这个函数包含了 PSR0 和 PSR4 标准的实现。还有个值得我们注意的是查找路径成功后 includeFile() 仍然是外面的函数，并不是 ClassLoader 的成员函数，原理跟上面一样，防止有用户写 $this 或 self。还有就是如果命名空间是以\开头的，要去掉\然后再匹配。

看下 findFileWithExtension 函数：

```
private function findFileWithExtension($class, $ext)
{
    // PSR-4 lookup
    $logicalPathPsr4 = strtr($class, '\\', DIRECTORY_SEPARATOR) . $ext;

    $first = $class[0];
    if (isset($this->prefixLengthsPsr4[$first])) {
        foreach ($this->prefixLengthsPsr4[$first] as $prefix => $length) {
            if (0 === strpos($class, $prefix)) {
                foreach ($this->prefixDirsPsr4[$prefix] as $dir) {
                    if (file_exists($file = $dir . DIRECTORY_SEPARATOR . substr($logica
                        return $file;
                    }
                }
            }
        }
    }

    // PSR-4 fallback dirs
    foreach ($this->fallbackDirsPsr4 as $dir) {
        if (file_exists($file = $dir . DIRECTORY_SEPARATOR . $logicalPathPsr4)) {
            return $file;
        }
```

最后小结

我们通过举例来说下上面代码的流程：

如果我们在代码中写下 `new phpDocumentor\Reflection\Element()` ，PHP 会通过 SPL autoload_register 调用 loadClass -> findFile -> findFileWithExtension 一步骤如下：

- 将 \ 转为文件分隔符/，加上后缀php，变成 $logicalPathPsr4, 即 phpDocumentor/Reflection//Element.php;
- 利用命名空间第一个字母p作为前缀索引搜索 prefixLengthsPsr4 数组，查到下面这个数组：

```
p' =>
    array (
        'phpDocumentor\\Reflection\\' => 25,
        'phpDocumentor\\Fake\\' => 19,
  )
```

- 遍历这个数组，得到两个顶层命名空间 phpDocumentor\Reflection\ 和 phpDocumentor\Fake\
- 在这个数组中查找 phpDocumentor\Reflection\Element，找出 phpDocumentor\Reflection\ 这个顶层命名空间并且长度为25。
- 在prefixDirsPsr4 映射数组中得到phpDocumentor\Reflection\ 的目录映射为：

```
'phpDocumentor\\Reflection\\' =>
    array (
        0 => __DIR__ . '/..' . '/phpdocumentor/reflection-common/src',
        1 => __DIR__ . '/..' . '/phpdocumentor/type-resolver/src',
        2 => __DIR__ . '/..' . '/phpdocumentor/reflection-docblock/src',
    ),
```

- 遍历这个映射数组，得到三个目录映射；
- 查看 “目录+文件分隔符//+substr($logicalPathPsr4, $length)” 文件是否存在，存在即返回。这里就是 `'__DIR__/../phpdocumentor/reflection-common/src + substr(phpDocumentor/Reflection/Element.php,25)'`
- 如果失败，则利用 fallbackDirsPsr4 数组里面的目录继续判断是否存在文件

以上就是 composer 自动加载的原理解析！

The end . Thanks!

你可能感兴趣的

- **PHP自动加载功能原理解析** leoyang90 composer laravel php
- **PHP autoload 机制详解** Corwien php autoload
- **现代PHP的发展趋势** 冬暖 php
- **【转】php命名空间** 耕毅 laravel php
- **自己动手写PHP框架（二）** 沙袋 php框架 php
- **人人都要知道的PHP底层运行机制与工作原理？** 喝醉的清茶 php
- **【转】浅谈PHP5中垃圾回收算法(Garbage Collection)的演化** 在路上 php
- **Modern-php 书摘（一）namespace** Sugar_w php

1 条评论

默认排序 | 时间排序

myluke · 8月23日

讲的非常透彻。

赞 回复

文明社会，理性评论

发布评论

移动版 桌面版