Composer

目录

早期代码加载

单文件

代码示例

> 一个文件就能实现所有的功能，不存在依赖或者加载关系的

```
<?php
echo "hello";

//output:
hello
```

优劣分析

- 优势：
  - 代码管理方便，仅仅单一文件对应单一业务。
- 劣势：
  - 仅仅对规模小的业务友好，较大业务会导致代码可读性差，且代码文件过大时，导致加载内存过大，影响性能。

require/include/require_once/include_once

require/include

> The require() function is identical to include(), except that it handles errors differently. If an error occurs, the include() function generates a warning, but the script will continue execution. The require() generates a fatal error, and the script will stop. 文件载入只是载入目标文件内的代码并执行，与载入的文件类型无关。文件载入属于执行阶段，当执行到require等语句时，才载入该文件的代码，编译并执行，然后回到require等语句位置继续执行下面的语句 注意：在载入开始时，先退出PHP模式；再载入目标文件代码，执行该代码；结束时，再进入PHP模式。 require：处理失败，产生 E_COMPILE_ERROR 错误，脚本中止。include：处理失败，产生 E_WARNING 错误，脚本继续执行。

- MyTestClass.php

```
<?php

/**
 * User: wangjstu
 * Date: 2018/8/12 16:06
 */
class MyTestClass
{
    private static $staticInstance = null;

    private function __construct()
```

```
    {
    }

    public static function getInstance()
    {
        if (is_null(static::$staticInstance)) {
            static::$staticInstance = new static();
        }
        return static::$staticInstance;
    }

    public function hello()
    {
        echo "Hello TestClass\n";
    }
}
```

- index.php

```
<?php
<?php
var_dump(class_exists('MyTestClass'));
//require "MyTestClass.php";
include "MyTestClass.php";
var_dump(class_exists('MyTestClass'));
echo "hello index\n";
MyTestClass::getInstance()->hello();

//output:
bool(false)
bool(true)
hello index
Hello TestClass
```

require_once/include_once

> The require_once() statement is identical to require() except PHP will check if the file has already been included, and if so, not include (require) it again.

优劣分析

- 优势：
  - 代码依赖拆分清晰。
  - 方便依赖代码加载。
- 劣势：
  - 许多业务用到同样几个class，于是在不同地方都需要加载一次。
  - 当类多了起来，会显得很乱、忘记加载时还会出现error(require的error与业务error)，
  - 若偷懒一次加载了所有的类，所有的类一次加载，导致内存会被吃爆。

__autoload

> PHP 5开始提供__autoload这种俗称“magic method”的函式。 当你要使用的类别PHP找不到时，它会将类别名称当成字串丢进这个函式，在PHP喷error投降之前，做最后的尝试 **Warning** This feature has been DEPRECATED as of PHP 7.2.0. Relying on this feature is highly discouraged.

- MyTestClass.php

```
<?php

/**
 * User: wangjstu
 * Date: 2018/8/12 16:06
 */
class MyTestClass
{
    private static $staticInstance = null;

    private function __construct()
    {
    }

    public static function getInstance()
    {
        if (is_null(static::$staticInstance)) {
            static::$staticInstance = new static();
        }
        return static::$staticInstance;
    }

    public function hello()
    {
        echo "Hello TestClass\n";
    }
}
```

- MyTestSubClass.php

```
<?php

/**
 * User: wangjstu
 * Date: 2018/8/12 16:10
 */
class MyTestSubClass extends MyTestClass
{
    private static $staticInstance = null;

    private function __construct()
    {
    }

    public static function getInstance()
    {
```

```
        if (is_null(static::$staticInstance)) {
            static::$staticInstance = new static();
        }
        return static::$staticInstance;
    }

    public function hello()
    {
        echo "Hello TestSubClass\n";
    }
}
```

- index.php

```
<?php
var_dump(class_exists('MyTestClass'));
// autoload.php
function __autoload($classname)
{
    if ($classname === 'MyTestClass') {
        $filename = "./" . $classname . ".php";
        include_once($filename);
    } elseif ($classname === 'MyTestSubClass') {
        $filename = "./" . $classname . ".php";
        include_once($filename);
    }
}
var_dump(class_exists('MyTestClass'));
echo "hello index\n";
MyTestClass::getInstance()->hello();
MyTestSubClass::getInstance()->hello();

//output:
bool(true)
bool(true)
hello index
Hello TestClass
Hello TestSubClass
```

优劣分析

- 优势：
  - 按需加载类
  - 减少不必要的加载类的硬编码
- 劣势：
  - __autoload函式内容会变得很巨大，因为在全局只能注册一次。
  - 查找类，一下会去根目录找、一下会去other_library资料夹、一下会去my_library资料夹寻找。在整理档案的时候，显得有些混乱。

spl_autoload_register

> PHP从5.1.2开始，多提供了一个函式。可以多写几个autoload函式，然后注册起来，效果跟直接使用__autoload相同。现在可以针对不同用途的类别，分批autoload了。

- MyTestClass.php

```
<?php

/**
 * User: wangjstu
 * Date: 2018/8/12 16:06
 */
class MyTestClass
{
    private static $staticInstance = null;

    private function __construct()
    {
    }

    public static function getInstance()
    {
        if (is_null(static::$staticInstance)) {
            static::$staticInstance = new static();
        }
        return static::$staticInstance;
    }

    public function hello()
    {
        echo "Hello TestClass\n";
    }
}
```

- MyTestSubClass.php

```
<?php

/**
 * User: wangjstu
 * Date: 2018/8/12 16:10
 */
class MyTestSubClass extends MyTestClass
{
    private static $staticInstance = null;

    private function __construct()
    {
    }

    public static function getInstance()
    {
        if (is_null(static::$staticInstance)) {
```

```
            static::$staticInstance = new static();
        }
        return static::$staticInstance;
    }

    public function hello()
    {
        echo "Hello TestSubClass\n";
    }
}
```

- index.php

```
<?php
var_dump(class_exists('MyTestClass'));
var_dump(class_exists('MyTestSubClass'));

spl_autoload_register('loaderMyTestClass');
spl_autoload_register('loaderMyTestSubClass');

function loaderMyTestClass($classname){
    $filename="./".$classname.".php";
    include_once($filename);
}
function loaderMyTestSubClass($classname){
    $filename="./".$classname.".php";
    include_once($filename);
}

var_dump(class_exists('MyTestClass'));
var_dump(class_exists('MyTestSubClass'));
echo "hello index\n";
MyTestClass::getInstance()->hello();
MyTestSubClass::getInstance()->hello();

//output
bool(false)
bool(false)
bool(true)
bool(true)
hello index
Hello TestClass
Hello TestSubClass
```

优劣分析

- 优势：
    - 可以按需多次写spl_autoload_register注册加载函数，加载顺序按谁先注册谁先调用。__aotuload由于是全局函数只能定义一次，不够灵活。
    - 可以被catch到错误，而__aotuload不能。
    - spl_autoload_register注册的加载函数可以按需被spl_autoload_unregister掉

- 劣势：
    - 还需要配置很多加载方法

现代代码加载

PSR规范

> - PSR-0 (Autoloading Standard) 自动加载标准
> - PSR-1 (Basic Coding Standard) 基础编码标准
> - PSR-2 (Coding Style Guide) 编码风格向导
> - PSR-3 (Logger Interface) 日志接口
> - PSR-4 (Improved Autoloading) 自动加载优化标准
> - PSR-6 缓存接口规范
> - PSR-7 HTTP 消息接口规范

psr-0 方式

```
{
    "autoload": {
        "psr-0": {
            "Foo\\": "src/",
        }
    }
}
```

> 这个配置也以 Map 的形式写入生成的 vendor/composer/autoload_namespaces.php 文件之中。

psr-4 方式

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
```

> 该方式是 composer 推荐使用的一种方式，因为它更易使用并能带来更简洁的目录结构。在 composer.json 里是这样进行配置的：

```
{
    "autoload": {
        "psr-4": {
            "Foo\\": "src/",
        }
    }
}
```

> key 和 value 就定义出了 namespace 以及到相应 path 的映射。按照 PSR-4 的规则，当试图自动加载 “Foo\Bar\Baz” 这个 class时，会去寻找 “src/Bar/Baz.php” 这个文件，如果它存在则进行加载。 注意，“Foo\”并没有出现在文件路径中，这是与 PSR-0 不同的一点，如果 PSR-0 有此配置，那么会去寻找 ”src/Foo/Bar/Baz.php” 这个文件。另外注意 PSR-4 和 PSR-0 的配置里，”Foo\” 结尾的命名空间分隔符必须加上并且进行转义，以防出现 ”Foo” 匹配到了 ”FooBar” 这样的意外发生。在 composer 安装或更新完之后，psr-4 的配置换被转换成 namespace 为 key，dir path 为 value 的Map 的形式，并写入生成的 vendor/composer/autoload_psr4.php 文件之中。

Class-map 方式

> 通过配置指定的目录或文件，然后在 Composer 安装或更新时，它会扫描指定目录下以 .php 或 .inc 结尾的文件中的class，生成class 到指定 file path 的映射，并加入新生成的 vendor/composer/autoload_classmap.php 文件中（前提是目录和文件已经存在，否则 composer 在扫描时会报错）

```
{
    "autoload": {
        "classmap": ["src/", "lib/", "Something.php"]
    }
}
```

> 例如 src/ 下有一个 BaseController 类，那么在 autoload_classmap.php 文件中，就会生成这样的配置: `'BaseController' => $baseDir . '/src/BaseController.php'`

Files方式

> 就是手动指定供直接加载的文件。比如说我们有一系列全局的 helper functions，可以放到一个 helper 文件里然后直接进行加载

```
{
    "autoload": {
        "files": ["src/MyLibrary/functions.php"]
    }
}
```

> 它会生成一个 array，包含这些配置中指定的 files，再写入新生成的 vendor/composer/autoload_files.php 文件中，以供autoloader 直接进行加载。

Composer

入门

> 配置composer.json文档，

```
{
    "autoload":{
        "classmap":[
            "myLibrary"
        ]
    }
}
```

> 运行`composer install`，生成vendor目录，

```
#wangjstu@wangjst-pc /e/test/vendor
$ ll -tr ./*
-rw-r--r-- 1 wangjstu 197121 178 八月 12 21:11 ./autoload.php

./composer:
total 29
-rw-r--r-- 1 wangjstu 197121     3 八月 12 21:11 installed.json
-rw-r--r-- 1 wangjstu 197121   149 八月 12 21:11 autoload_namespaces.php
-rw-r--r-- 1 wangjstu 197121   143 八月 12 21:11 autoload_psr4.php
-rw-r--r-- 1 wangjstu 197121   147 八月 12 21:11 autoload_classmap.php
-rw-r--r-- 1 wangjstu 197121   317 八月 12 21:11 autoload_static.php
-rw-r--r-- 1 wangjstu 197121 13420 八月 12 21:11 ClassLoader.php
-rw-r--r-- 1 wangjstu 197121  1762 八月 12 21:11 autoload_real.php
-rw-r--r-- 1 wangjstu 197121  1070 八月 12 21:11 LICENSE
```

> 然后引入文件`require 'vendor/autoload.php';`，然后'composer install'指令除了自动加载你的类别之外，还会自动下载你需要的类别，然后自动加载它们。

```
{
    "require":{
        "google/apiclient":"1.0.*@beta",
        "guzzlehttp/guzzle":"~4.0",
        "doctrine/dbal":"~2.4"
    },

    "autoload":{
        "classmap":[
            "my_library"
        ]
    }
}

目录如下:
$ ll ./*
-rw-r--r-- 1 wangjstu 197121 152 八月 12 21:25 ./composer.json

./myLibrary:
total 0

./vendor:
total 5
-rw-r--r-- 1 wangjstu 197121 178 八月 12 21:11 autoload.php
drwxr-xr-x 1 wangjstu 197121   0 八月 12 21:31 composer/
drwxr-xr-x 1 wangjstu 197121   0 八月 12 21:31 guzzlehttp/
```

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

composer.json语法

- 示例composer.json文件

```
{
    "name": "vendor-name/project-name",
    "description": "This is a very cool package!",
    "version": "0.3.0",
    "type": "library",
    "keywords": ["logging", "cool", "awesome"],
    "homepage": "https://jolicode.com",
    "time": "2012-12-21",
    "license": "MIT",
    "authors": [
        {
            "name": "Xavier Lacot",
            "email": "xlacot@jolicode.com",
            "homepage": "http://www.lacot.org",
            "role": "Developer"
        },
        {
            "name": "Benjamin Clay",
            "email": "bclay@jolicode.com",
            "homepage": "https://github.com/ternel",
            "role": "Developer"
        }
    ],
    "support": {
        "email": "support@exemple.org",
        "issues": "https://github.com/jolicode/jane/issues",
        "forum": "http://www.my-forum.com/",
        "wiki": "http://www.my-wiki.com/",
        "irc": "irc://irc.freenode.org/composer",
        "source": "https://github.com/jolicode/jane",
        "docs": "https://github.com/jolicode/jane/wiki"
    },
    "require": {
        "monolog/monolog": "1.0.*",
        "joli/ternel": "@dev",
        "joli/ternel-bundle": "@stable",
        "joli/semver": "^2.0",
        "joli/package": ">=1.0 <1.1",
        "acme/foo": "dev-master#2eb0c097"
    },
    "require-dev": {
        "debug/dev-only": "1.0.*"
    },
    "conflict": {
```

```
        "another-vendor/conflict": "1.0.*"
    },
    "replace": {
        "debug/dev-only": "1.0.*"
    },
    "provide": {
        "debug/dev-only": "1.0.*"
    },
    "suggest": {
        "jolicode/gif-exception-bundle": "For fun!"
    },
    "autoload": {
        "psr-4": {
            "Monolog\\": "src/",
            "Vendor\\Namespace\\": ""
        },
        "psr-0": {
            "Monolog": "src/",
            "Vendor\\Namespace": ["src/", "lib/"],
            "Pear_Style": "src/",
            "": "src/"
        },
        "classmap": ["src/", "lib/", "Something.php"],
        "files": ["src/MyLibrary/functions.php"]
    },
    "autoload-dev": {
        "psr-0": {
            "MyPackage\\Tests": "test/"
        }
    },
    "target-dir": "Symfony/Component/Yaml",
    "minimum-stability": "stable",
    "repositories": [
        {
            "type": "composer",
            "url": "http://packages.example.com"
        },
        {
            "type": "vcs",
            "url": "https://github.com/Seldaek/monolog"
        },
        {
            "type": "pear",
            "url": "http://pear2.php.net"
        },
        {
            "type": "package",
            "package": {
              "name": "smarty/smarty",
              "version": "3.1.7",
              "dist": {
                "url": "http://www.smarty.net/Smarty-3.1.7.zip",
                "type": "zip"
```

```
            },
            "source": {
              "url": "http://smarty-php.googlecode.com/svn/",
              "type": "svn",
              "reference": "tags/Smarty_3_1_7/distribution/"
            }
          }
    },
    {
        "type": "artifact",
        "url": "path/to/directory/with/zips/"
    },
    {
        "type": "path",
        "url": "../../packages/my-package"
    }
],
"config": {
    "process-timeout": 300,
    "use-include-path": false,
    "preferred-install": "auto",
    "store-auths": "prompt",
    "github-protocols": ["git", "https", "http"],
    "github-oauth": {"github.com": "oauthtoken"},
    "gitlab-oauth": {"gitlab.com": "oauthtoken"},
    "github-domains": ["entreprise-github.me.com"],
    "gitlab-domains": ["entreprise-gitlab.me.com"],
    "github-expose-hostname": true,
    "disable-tls": false,
    "cafile": "/var/certif.ca",
    "capath": "/var/",
    "http-basic": {"me.io":{"username":"foo","password":"bar"},
    "platform": {"php": "5.4", "ext-something": "4.0"},
    "vendor-dir": "vendor",
    "bin-dir": "bin",
    "data-dir": "/home/ternel/here",
    "cache-dir": "$home/cache",
    "cache-files-dir": "$cache-dir/files",
    "cache-repo-dir": "$cache-dir/repo",
    "cache-vcs-dir": "$cache-dir/vcs",
    "cache-files-ttl": 15552000,
    "cache-files-maxsize": "300MiB",
    "bin-compat": "auto",
    "prepend-autoloader": true,
    "autoloader-suffix": "pony",
    "optimize-autoloader": false,
    "sort-packages": false,
    "classmap-authoritative": false,
    "notify-on-install": true,
    "discard-changes": false,
    "archive-format": "tar",
    "archive-dir": "."
},
```

```
    "archive": {
        "exclude": ["/foo/bar", "baz", "/*.test", "!/foo/bar/baz"]
    },
    "prefer-stable": true,
    "scripts": {
        "pre-install-cmd": "MyVendor\\MyClass::doSomething",
        "post-install-cmd": [
            "MyVendor\\MyClass::warmCache",
            "phpunit -c app/"
        ],
        "pre-update-cmd": "MyVendor\\MyClass::doSomething",
        "post-update-cmd": "MyVendor\\MyClass::doSomething",
        "pre-status-cmd": "MyVendor\\MyClass::doSomething",
        "post-status-cmd": "MyVendor\\MyClass::doSomething",
        "pre-package-install": "MyVendor\\MyClass::doSomething",
        "post-package-install": [
            "MyVendor\\MyClass::postPackageInstall"
        ],
        "pre-package-update": "MyVendor\\MyClass::doSomething",
        "post-package-update": "MyVendor\\MyClass::doSomething",
        "pre-package-uninstall": "MyVendor\\MyClass::doSomething",
        "post-package-uninstall": "MyVendor\\MyClass::doSomething",
        "pre-autoload-dump": "MyVendor\\MyClass::doSomething",
        "post-autoload-dump": "MyVendor\\MyClass::doSomething",
        "post-root-package-install": "MyVendor\\MyClass::doStuff",
        "post-create-project-cmd": "MyVendor\\MyClass::doThis",
        "pre-archive-cmd": "MyVendor\\MyClass::doSomething",
        "post-archive-cmd": "MyVendor\\MyClass::doSomething",
    },
    "extra": { "key": "value" },
    "bin": ["./bin/toto"]
}
```

- 解析参考《composer json》与《phpcomposer官网》

composer 命令

```
composer require vendor-name/package-name
composer install
// 通过update命令，可以更新项目里所有的包，或者指定的某些包
composer update
composer update --lock
composer dump-autoload --optimize
composer about
composer archive
composer browse
composer clear-cache
composer config --list
composer create-project symfony/standard-edition dir/
composer depends vendor-name/package-name
composer diagnose
composer exec
```

```
composer global
composer help
composer info
composer init
composer licenses
composer list
composer outdated
composer prohibits
// 使用remove命令可以移除一个包及其依赖（在依赖没有被其他包使用的情况下）
composer remove
composer run-script
// 使用search命令可以进行包的搜索
composer search my keywords
composer self-update
// 使用show命令可以列出项目目前所安装的包的信息
composer show
composer status
composer suggests
composer validate
```

版本约束

- 波浪号~：下一个重要版本操作符。~操作符的用法：~1.2相当于>=1.2 <2.0.0，而~1.2.3相当于>=1.2.3 <1.3.0。
- 折音号^：允许升级版本到安全的版本。例如，^1.2.3相当于>=1.2.3 <2.0.0，因为在2.0版本前的版本应该都没有兼容性的问题。而对于1.0之前的版本，这种约束方式也考虑到了安全问题，例如^0.3会被当作>=0.3.0 <0.4.0对待。
- 版本稳定性

> 如果你没有显式的指定版本的稳定性，Composer会根据使用的操作符，默认在内部指定为-dev或者-stable。例如：

约束	内部约束
1.2.3	=1.2.3.0-stable
>1.2	>1.2.0.0-stable
>=1.2	>=1.2.0.0-dev
>=1.2-stable	>=1.2.0.0-stable
<1.3	<1.3.0.0-dev
<=1.3	<=1.3.0.0-stable
1 - 2	>=1.0.0.0-dev <3.0.0.0-dev
~1.3	>=1.3.0.0-dev <2.0.0.0-dev
1.4.*	>=1.4.0.0-dev <1.5.0.0-dev

> 如果你想指定版本只要稳定版本，你可以在版本后面添加后缀-stable。 minimum-stability 配置项定义了包在选择版本时对稳定性的选择的默认行为。默认是stable。它的值如下（按照稳定性排序）：dev，alpha，beta，RC和stable。除了修改这个配置去修改这个默认行为，我们还可以通过稳定性标识（例如@stable和@dev）来安装一个相比于默认配置不同稳定性的版本。例如：

```
{
    "require": {
        "monolog/monolog": "1.0.*@beta",
        "acme/foo": "@dev"
    }
}
```

composer最佳实践技巧

- 阅读文档composer doc。
- 注意 项目 和 库 之间的区别。一个库是一个可重用的包，你可以添加一个依赖。一个项目通常是一个应用程序，依赖于几个库. 它通常是不可重用的（没有其它项目会要求它作为依赖. 典型的例子是电子商务网站，客户支持系统等)。
- 使用特定的依赖关系-关于应用程序的版本。
- 对库依赖项使用版本范围。
- 开发应用程序要提交 composer.lock 文件到 git 版本库中。
- 开发库要把 composer.lock 文件添加到 .gitignore 文件中。
- Travis CI 构建依赖项的不同版本，composer 为安装低版本依赖项提供了一个开关 --prefer-lowest （应使用 --prefer-stable ，可阻止不稳定版本的安装）
- 按名称对 require 和 require-dev 中的包排序
- 进行版本衍合或合并时不要合并 composer.lock
- 了解 require 和 require-dev之间的区别。需要运行在应用中或者库中的包都应该被定义在 require (例如：Symfony, Doctrine, Twig, Guzzle, ...)中。如果你正在创建一个库， 注意将什么内容定义为 require。因为这个部分的 每个依赖项同时也是使用了该库的应用的依赖。开发应用程序(或库)所需的包应该定义在require-dev (例如：PHPUnit, PHP_CodeSniffer, PHPStan)中。
- 安全地升级依赖项。应该定期对依赖项升级。可用 composer outdated 命令查看哪些依赖项需要升级。追加一个 --direct （或 -D）参数开关是个聪明之举，这只会查看 composer.json 指定的依赖项。还有一个 -m 参数开关，只查看次版本号的升级列表。

> 对每一个老版本的依赖项进行升级都要尊循如下步骤：
> 1. 创建新分支
> 2. 在 composer.json 文件中更新该依赖项版本到最新版本号
> 3. 运行 composer update phpunit/phpunit --with-dependencies （使用升级过的库替换 phpunit/phpunit）
> 4. 检查 Github 上库的版本库中 CHANGELOG 文件，检查是否存在重大变化。 如果存在就升级应用程序
> 5. 本地测试应用程序（使用 Symfony 的话还能在调试栏看到弃用警告）
> 6. 提交修改（包括 composer.json 、 composer.lock 及其他新版本正常运行所做的必要修改）
> 7. 等 CI 构建结束
> 8. 合并然后部署

- 在 composer.json 中定义其他类型的依赖
- 在CI构建期间验证 composer.json
- 在 PHPStorm 中使用 Composer 插件
- 在 composer.json 中指明生产环境的PHP版本号

- 使用自有托管 Gitlab 上的私有包
- 临时使用 fork 下 bug 修复分支的方法
- 使用 prestissimo 加速你的包安装
- 当你不确定时，测试你的版本约束
- 在生产环境中使用使用权威类映射文件，应该在生产环境中 生成权威类映射文件 。这会让类映射文件中包含的所有类快速加载，而不必到磁盘文件系统进行任何检查。可以在生产环境构建时运行以下命令：`composer dump-autoload --classmap-authoritative`
- 为测试配置 autoload-dev
- 尝试 Composer 脚本

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())。
2. ClassLoader.php : composer 加载类。
    - 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', 'loadClassLoader'),
true, true
      );

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

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

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

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

          call_user_func(

\Composer\Autoload\ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::getInitializer($loader)
          );

      } 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);
          }
      }
```

```
      /**********************注册自动加载核心类对象*******************/
      $loader->register(true);

      /**********************自动加载全局函数*******************/
      if ($useStaticLoader) {
          $includeFiles =
Composer\Autoload\ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$files;
      } else {
          $includeFiles = require __DIR__ . '/autoload_files.php';
      }

      foreach ($includeFiles as $fileIdentifier => $file) {
          composerRequire7b790917ce8899df9af8ed53631a1c29($fileIdentifier, $file);
      }

      return $loader;
    }
```

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'), true,
true
  );

  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。

第三部分 —— 初始化核心类对象

```
<?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::getInitializer($loader)
     );
  } 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
  class ComposerStaticInit7b790917ce8899df9af8ed53631a1c29{
     public static $files = array(...);
     public static $prefixLengthsPsr4 = array(...);
     public static $prefixDirsPsr4 = array(...);
     public static $prefixesPsr0 = array(...);
     public static $classMap = array (...);

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

          $loader->prefixDirsPsr4
                          = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$prefixDirsPsr4;

          $loader->prefixesPsr0
                          = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$prefixesPsr0;

          $loader->classMap
                          = ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$classMap;

      }, null, ClassLoader::class);
  }
```

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/ForgotPasswordController.php',

      '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'),

    'Webmozart\\Assert\\'
        => array($vendorDir . '/webmozart/assert/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：

```
<?php
public static $classMap = array (
    '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
   7b790917ce8899df9af8ed53631a1c29', 'loadClassLoader'), true, true);

   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';

       call_user_func(\Composer\Autoload\ComposerStaticInit
       7b790917ce8899df9af8ed53631a1c29::getInitializer($loader));

   } 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);
       }
```

```
    }

    /**********************注册自动加载核心类对象*******************/
    $loader->register(true);

    /**********************自动加载全局函数*******************/
    if ($useStaticLoader) {
        $includeFiles = Composer\Autoload\ComposerStaticInit
        7b790917ce8899df9af8ed53631a1c29::$files;
    } else {
        $includeFiles = require __DIR__ . '/autoload_files.php';
    }

    foreach ($includeFiles as $fileIdentifier => $file) {
        composerRequire
        7b790917ce8899df9af8ed53631a1c29($fileIdentifier, $file);
    }

    return $loader;
}
```

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，加载全局函数。

```
if ($useStaticLoader) {
    $includeFiles =
Composer\Autoload\ComposerStaticInit7b790917ce8899df9af8ed53631a1c29::$files;
} 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/bootstrap.php',
'667aeda72477189d0494fecd327c3641' => __DIR__ . '/..' . '/symfony/var-
dumper/Resources/functions/dump.php',
...
);
```

普通初始化

autoload_files:

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

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

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);
    }

    // class map lookup
    if (isset($this->classMap[$class])) {
        return $this->classMap[$class];
    }
    if ($this->classMapAuthoritative) {
        return false;
    }

    $file = $this->findFileWithExtension($class, '.php');

    // Search for Hack files if we are running on HHVM
    if ($file === null && defined('HHVM_VERSION')) {
        $file = $this->findFileWithExtension($class, '.hh');
    }

    if ($file === null) {
```

```
        // Remember that this class does not exist.
        return $this->classMap[$class] = false;
    }

    return $file;
}
```

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 = $dir . DIRECTORY_SEPARATOR .
substr($logicalPathPsr4, $length))) {
                        return $file;
                    }
                }
            }
        }
    }

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

    // PSR-0 lookup
    if (false !== $pos = strrpos($class, '\\')) {
        // namespaced class name
        $logicalPathPsr0 = substr($logicalPathPsr4, 0, $pos + 1)
            . strtr(substr($logicalPathPsr4, $pos + 1), '_', DIRECTORY_SEPARATOR);
    } else {
        // PEAR-like class name
        $logicalPathPsr0 = strtr($class, '_', DIRECTORY_SEPARATOR) . $ext;
    }

    if (isset($this->prefixesPsr0[$first])) {
```

```
        foreach ($this->prefixesPsr0[$first] as $prefix => $dirs) {
            if (0 === strpos($class, $prefix)) {
                foreach ($dirs as $dir) {
                    if (file_exists($file = $dir . DIRECTORY_SEPARATOR . $logicalPathPsr0)) {
                        return $file;
                    }
                }
            }
        }
    }

    // PSR-0 fallback dirs
    foreach ($this->fallbackDirsPsr0 as $dir) {
        if (file_exists($file = $dir . DIRECTORY_SEPARATOR . $logicalPathPsr0)) {
            return $file;
        }
    }

    // PSR-0 include paths.
    if ($this->useIncludePath && $file = stream_resolve_include_path($logicalPathPsr0)) {
        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。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！

参考

- have-you-tried-composer-scripts
- 17-tips-for-using-composer-efficiently
- psr-0自动加载规范
- PHP的PSR-0标准及namespace
- psr-4自动加载规范
- composer json
- phpcomposer官网
- PSR
- 深入解析 composer 的自动加载原理